

GPU Coder™

User's Guide



MATLAB® & SIMULINK®

R2022b



How to Contact MathWorks



Latest news: www.mathworks.com
Sales and services: www.mathworks.com/sales_and_services
User community: www.mathworks.com/matlabcentral
Technical support: www.mathworks.com/support/contact_us



Phone: 508-647-7000



The MathWorks, Inc.
1 Apple Hill Drive
Natick, MA 01760-2098

GPU Coder™ User's Guide

© COPYRIGHT 2017–2022 by The MathWorks, Inc.

The software described in this document is furnished under a license agreement. The software may be used or copied only under the terms of the license agreement. No part of this manual may be photocopied or reproduced in any form without prior written consent from The MathWorks, Inc.

FEDERAL ACQUISITION: This provision applies to all acquisitions of the Program and Documentation by, for, or through the federal government of the United States. By accepting delivery of the Program or Documentation, the government hereby agrees that this software or documentation qualifies as commercial computer software or commercial computer software documentation as such terms are used or defined in FAR 12.212, DFARS Part 227.72, and DFARS 252.227-7014. Accordingly, the terms and conditions of this Agreement and only those rights specified in this Agreement, shall pertain to and govern the use, modification, reproduction, release, performance, display, and disclosure of the Program and Documentation by the federal government (or other entity acquiring for or through the federal government) and shall supersede any conflicting contractual terms or conditions. If this License fails to meet the government's needs or is inconsistent in any respect with federal procurement law, the government agrees to return the Program and Documentation, unused, to The MathWorks, Inc.

Trademarks

MATLAB and Simulink are registered trademarks of The MathWorks, Inc. See www.mathworks.com/trademarks for a list of additional trademarks. Other product or brand names may be trademarks or registered trademarks of their respective holders.

Patents

MathWorks products are protected by one or more U.S. patents. Please see www.mathworks.com/patents for more information.

Revision History

September 2017	Online only	New for Version 1.0 (Release 2017b)
March 2018	Online only	Revised for Version 1.1 (Release 2018a)
September 2018	Online only	Revised for Version 1.2 (Release 2018b)
March 2019	Online only	Revised for Version 1.3 (Release 2019a)
September 2019	Online only	Revised for Version 1.4 (Release 2019b)
March 2020	Online only	Revised for Version 1.5 (Release 2020a)
September 2020	Online only	Revised for Version 2.0 (Release 2020b)
March 2021	Online only	Revised for Version 2.1 (Release 2021a)
September 2021	Online only	Revised for Version 2.2 (Release 2021b)
March 2022	Online only	Revised for Version 2.3 (Release 2022a)
September 2022	Online only	Revised for Version 2.4 (Release 2022b)

1 | **Functions Supported for GPU Code Generation**

MATLAB Language Features Support for GPU Coder	1-2
Code Generation for Variable-Size Arrays	1-2
Structure Definition for Code Generation	1-4
Unsupported Features	1-5
Supported Functions	1-6

2 | **Kernel Creation from MATLAB Code**

Kernels from Element-Wise Loops	2-2
Element-Wise Math Example	2-2
Preparing myFun for Code Generation	2-2
Generated CUDA Code	2-3
Limitations	2-3
Kernels from Scatter-Gather Type Operations	2-4
Vector Sum Example	2-5
Prepare vecSum for Kernel Creation	2-5
Generated CUDA Code	2-5
1-D Reduction Operations on the GPU	2-6
Kernels from Library Calls	2-8
cuBLAS Example	2-10
Generated CUDA Code	2-10
Prepare blas_gemm for Kernel Creation	2-11
cuSOLVER Example	2-12
Prepare backslash for Kernel Creation	2-12
Generated CUDA Code	2-12
cuSOLVER Standalone Code	2-13
FFT Example	2-15
Prepare myFFT for Kernel Creation	2-15
Generated CUDA Code	2-16
Thrust Example	2-17
Generated CUDA Code	2-17

Call Custom CUDA Kernels from the Generated Code	2-18
Call Custom CUDA Kernel	2-18
Call Custom CUDA Device Function from the Generated Code	2-22
Call <code>_usad4_wrap</code> CUDA Device Function	2-22
Generate CUDA Code	2-24
Generated Code	2-24
Design Patterns	2-26
Stencil Processing	2-26
Matrix-Matrix Processing	2-26
GPU Memory Allocation and Minimization	2-28
Discrete and Managed Modes	2-28
GPU Memory Manager	2-28
Memory Minimization	2-30
Use Dynamically Allocated C++ Arrays in Generated Function Interfaces	2-33
Change Interface Generation	2-33
Using the <code>coder::gpu_array</code> Class Template	2-33
Limitations	2-35
Support for GPU Arrays	2-37
Considerations	2-37
Limitations	2-37
What is Half Precision?	2-39
Half Precision Applications	2-39
Benefits of Using Half Precision in Embedded Applications	2-41
Half Precision in MATLAB	2-42
Half Precision in Simulink	2-43
Code Generation with Half Precision	2-43
Half Precision Code Generation Support	2-44
Simulate Diffraction Patterns Using CUDA FFT Libraries	2-55
Benchmark Solving a Linear System by Using GPU Coder	2-61
QR Decomposition on NVIDIA GPU Using cuSOLVER Libraries	2-69
Stencil Processing on GPU	2-74
Fog Rectification	2-80
Stereo Disparity	2-85
Feature Extraction Using SURF	2-91
Feature Matching	2-96
Lane Detection on the GPU by Using the <code>houghlines</code> Function	2-100

Edge Detection with Sobel Method in Half-Precision	2-103
Build a Map from Lidar Data using SLAM on GPU	2-107

Kernel Creation from Simulink Models

3

Simulation Acceleration by Using GPU Coder	3-2
Example: Sobel Edge Detection	3-2
Create Edge Detection Model	3-3
Configure Model for GPU Acceleration	3-5
Build GPU Accelerated Model	3-6
Limitations	3-7
Code Generation from Simulink Models with GPU Coder	3-8
Example: Sobel Edge Detection	3-8
Create Edge Detection Model	3-8
Configure Model for Code Generation	3-10
Generate CUDA Code for the Model	3-12
Limitations	3-12
GPU Code Generation for Deep Learning Networks Using MATLAB	
Function Block	3-14
Example: Classify Images by Using GoogLeNet	3-14
Create GoogLeNet Model	3-15
Configure Model for GPU Acceleration	3-17
Build GPU Accelerated Model	3-18
Configure the Model for Code Generation	3-19
Generate CUDA Code for the Model	3-21
Limitations	3-21
GPU Code Generation for Blocks from the Deep Neural Networks Library	
.....	3-22
Example: Classify Images by Using GoogLeNet	3-22
Create GoogLeNet Model	3-24
Configure the Model for GPU Acceleration	3-24
Build GPU Accelerated Model	3-26
Configure Model for Code Generation	3-27
Generate CUDA Code for the Model	3-28
Limitations	3-28
Targeting NVIDIA Embedded Boards	3-30
Configure Model for Deployment	3-30
Generate CUDA Code for the Model	3-30
Numerical Equivalence Testing	3-32
Target Connectivity Configuration for PIL	3-32
Example: The Mandelbrot Set	3-33
GPU Acceleration or PIL Simulation with a Top Model	3-34
Run Normal and PIL Simulations	3-35
Limitations	3-37

Parameter Tuning and Signal Monitoring by Using External Mode	3-38
Example: The Mandelbrot Set	3-39
Create Mandelbrot Model	3-40
Build Target Executable	3-40
Run Target Application	3-41
Stop Target Application	3-42
GPU Code Generation for Lane Detection in Simulink	3-43
GPU Code Generation for a Fog Rectification Simulink Model	3-48
Code Generation for a Deep Learning Simulink Model to Classify ECG Signals	3-53
Code Generation for a Deep Learning Simulink Model that Performs Lane and Vehicle Detection	3-60

Deep Learning

4

Workflow	4-3
Supported Networks, Layers, and Classes	4-6
Supported Pretrained Networks	4-6
Supported Layers	4-11
Supported Classes	4-33
Analyze Network for Code Generation	4-44
Check dlnetwork for Code Generation Compatibility	4-44
Analyze Classification Network for Code Generation Compatibility	4-46
Code Generation for dlarray	4-52
Define dlarray for Code Generation	4-52
dlarray Object Functions with Code Generation Support	4-53
Deep Learning Toolbox Functions with dlarray Code Generation Support	4-54
MATLAB Functions with dlarray Code Generation Support	4-54
dlarray Limitations for Code Generation	4-62
Recommended Usage	4-62
Limitations	4-62
Generated CNN Class Hierarchy	4-65
Load Pretrained Networks for Code Generation	4-66
Load a Network by Using coder.loadDeepLearningNetwork	4-66
Specify a Network Object for Code Generation	4-67
Specify a dlnetwork Object for Code Generation	4-67
Code Generation for Deep Learning Networks by Using cuDNN	4-69
Generate Code and Classify Images by Using GoogLeNet	4-69
Requirements	4-69

Load Pretrained Network	4-70
Create an Entry-Point Function	4-71
Code Generation by Using codegen	4-71
Generate Code by Using the App	4-74
Generated Makefile	4-75
Run the Generated MEX	4-75
Code Generation for Deep Learning Networks by Using TensorRT	4-78
Generate Code and Classify Images by Using GoogLeNet	4-78
Requirements	4-78
Load Pretrained Network	4-79
Create an Entry-Point Function	4-80
Code Generation by Using codegen	4-81
Generate Code by Using the App	4-84
Generated Makefile	4-85
Run the Generated MEX	4-85
Code Generation for Deep Learning Networks Targeting ARM Mali GPUs	
.....	4-88
Requirements	4-88
Load Pretrained Network	4-88
Code Generation by Using cnncodegen	4-89
Limitations	4-91
Update Network Parameters After Code Generation	4-92
Create an Entry-Point Function	4-92
Create a Network	4-92
Code Generation by Using codegen	4-93
Run the Generated MEX	4-93
Update Network with Different Learnable Parameters	4-94
Run the Generated MEX with Updated Learnables	4-94
Limitations	4-95
Data Layout Considerations in Deep Learning	4-96
Data Layout Format for CNN	4-96
Data Layout Format for LSTM	4-97
Quantization of Deep Neural Networks	4-99
Precision and Range	4-99
Histograms of Dynamic Ranges	4-99
Generate INT8 Code for Deep Learning Networks	4-107
Classify Images Using a Network Optimized for INT8 Inference	4-107
Limitations	4-115
Code Generation for Deep Learning Networks	4-117
Lane Detection Optimized with GPU Coder	4-124
Traffic Sign Detection and Recognition	4-132
Logo Recognition Network	4-140
Deep Learning Prediction with NVIDIA TensorRT Library	4-145

Code Generation for Semantic Segmentation Network	4-152
Train and Deploy Fully Convolutional Networks for Semantic Segmentation	4-157
Code Generation for Semantic Segmentation Network That Uses U-net	4-169
Code Generation for Denoising Deep Neural Network	4-176
Code Generation for Object Detection by Using YOLO v2	4-180
Code Generation for a Sequence-to-Sequence LSTM Network	4-184
Deep Learning Prediction on ARM Mali GPU	4-190
Code Generation for Object Detection by Using Single Shot Multibox Detector	4-193
Code Generation for a Deep Learning Simulink Model to Classify ECG Signals	4-197
Code Generation for Lidar Point Cloud Segmentation Network	4-204
Code Generation for a Video Classification Network	4-211
Code Generation For Object Detection Using YOLO v3 Deep Learning	4-217
Generate Digit Images on NVIDIA GPU Using Variational Autoencoder	4-222
Quantize Residual Network Trained for Image Classification and Generate CUDA Code	4-229
Quantize Layers in Object Detectors and Generate CUDA Code	4-237
Parameter Pruning and Quantization of Image Classification Network	4-249
Code Generation For Aerial Lidar Semantic Segmentation Using PointNet ++ Deep Learning	4-266
Code Generation For Lidar Object Detection Using PointPillars Deep Learning	4-272
Code Generation for Object Detection Using YOLO v4 Deep Learning	4-277

Build and Run an Executable on NVIDIA Hardware	5-2
Learning Objectives	5-2
Tutorial Prerequisites	5-2
Example: Vector Addition	5-3
Create a Live Hardware Connection Object	5-3
Generate CUDA Executable Using GPU Coder	5-4
Run the Executable and Verify the Results	5-5
Build and Run an Executable on NVIDIA Hardware Using GPU Coder App	5-7
Learning Objectives	5-7
Tutorial Prerequisites	5-7
Example: Vector Addition	5-8
Custom Main File	5-8
GPU Coder App	5-9
Run the Executable and Verify the Results	5-12
Relocate Generated Code to Another Development Environment	5-14
Package Generated Code Using the GPU Coder	5-14
Specify packNGo Options	5-22
Getting Started with the MATLAB Coder Support Package for NVIDIA Jetson and NVIDIA DRIVE Platforms	5-24
Sobel Edge Detection on NVIDIA Jetson Nano Using Raspberry Pi Camera Module V2	5-29
Semantic Segmentation on NVIDIA DRIVE	5-34
Top-Hat Filtering to Remove Uneven Background Illumination on NVIDIA Jetson TX2 Developer Kit	5-39
Deployment and Classification of Webcam Images on NVIDIA Jetson TX2 Platform	5-44
Ground Plane Segmentation and Obstacle Detection on NVIDIA Jetson Xavier™ NX Embedded platform	5-49
Deploy Signal Classifier on NVIDIA Jetson Using Wavelet Analysis and Deep Learning	5-57

Troubleshooting

Workflow	6-2
Code Generation Reports	6-5
Report Generation	6-5

Report Location	6-6
Errors and Warnings	6-6
Files and Functions	6-6
MATLAB Source	6-6
Generated Code	6-8
MATLAB Variables	6-8
Tracing Code	6-9
Code Insights	6-10
Additional Reports	6-10
Report Limitations	6-10
Trace Between Generated CUDA Code and MATLAB Source Code	6-11
Generate Traceability Tags	6-11
Format of Traceability Tags	6-13
Traceability Tag Limitations	6-14
Generating a GPU Code Metrics Report for Code Generated from MATLAB	
Code	6-15
Example GPU Code Metrics Report	6-15
Explore the code metrics report	6-16
Limitations	6-17
Kernel Analysis	6-18
Mapping Nested Loops to Kernels	6-18
For-Loops with Break	6-19
Dependence Analysis Parallel Loop Check Fails	6-19
Logical Indexing of Arrays	6-20
Unsupported Functions	6-20
Loop Interchange	6-20
Memory Bottleneck Analysis	6-22
Data Alignment	6-22
Small Data Sizes	6-22
Too Many cudaMemcpy	6-22
Constant Inputs	6-22
Stack Memory Usage	6-23
Analyze Execution Profiles of the Generated Code	6-24
Create a Design File	6-24
Generate the Execution Profiling Report	6-24
Analysis with NVIDIA Profiler	6-27
Not Enough Parallelism	6-27
Too Many Local per-Thread Registers	6-27
GPU Coder Limitations	6-28
General Limitations	6-28
Function Limitations	6-28
Unsupported CUDA Features	6-28
GPU Execution Profiling of the Generated Code	6-30

Register Count nmlink Error	7-2
Issue	7-2
Possible Solutions	7-2

Functions Supported for GPU Code Generation

- “MATLAB Language Features Support for GPU Coder” on page 1-2
- “Supported Functions” on page 1-6

MATLAB Language Features Support for GPU Coder

GPU Coder™ supports many of the MATLAB® language features supported by MATLAB Coder™, see “MATLAB Language Features Supported for C/C++ Code Generation”. However, some features may be supported in a restricted mode and others not supported. In the following sections, we highlight some of the important features that affect GPU code generation and then list the features that not supported by GPU Coder.

A common and important consideration is variable-size matrices support. This feature can really affect the way CUDA® kernels are created and the following discussion describes the feature and considerations for GPU code generation.

Code Generation for Variable-Size Arrays

For code generation, an array dimension is fixed-size or variable-size. If the code generator can determine the size of an array and that the size of the array does not change at run time, then the dimension is fixed-size. When all dimensions of an array are fixed-size, the array is a fixed-size array. In the following example, *Z* is a fixed-size array.

```
function Z = myfcn()  
Z = zeros(1,4);  
end
```

If the code generator cannot determine the size of an array or the code generator determines that the size changes, then the dimension is variable-size. When at least one of its dimensions is variable-size, an array is a variable-size array.

A variable-size dimension is either bounded or unbounded. A bounded dimension has a fixed upper size. An unbounded dimension does not have a fixed upper size.

In the following example, the second dimension of *Z* is bounded, variable-size. It has an upper bound of 32.

```
function s = myfcn(n)  
if (n > 0)  
    Z = zeros(1,4);  
else  
    Z = zeros(1,32);  
end  
s = length(Z);
```

In the following example, if the value of *n* is unknown at compile time, then the second dimension of *Z* is unbounded.

```
function s = myfcn(n)  
Z = rand(1,n);  
s = sum(Z);  
end
```

You can define variable-size arrays by:

- Using constructors, such as `zeros` or `ones`, with a nonconstant size value
- Assigning multiple, constant sizes to the same variable before using it
- Using loops to grow the dimensions of variables

- Declaring all instances of a variable to be variable-size by using `coder.typeof` or `coder.varsizes` functions. For example, `coder.typeof(1, [12,1],[true, false])` and `coder.varsizes(1, [Inf,1], [true, false])`.

For more information, see “Define Variable-Size Data for Code Generation”.

Enabling and Disabling Support for Variable-Size Arrays

Code Generation Behavior

For variable-size arrays that are bounded, GPU Coder maps these bounded variables to the GPU and CUDA kernels are created. To specify upper bounds for variable-size arrays, see “Specify Upper Bounds for Variable-Size Arrays”.

For unbounded, variable-size arrays and variable-size arrays whose size is greater than or equal to a `DynamicMemoryAllocation` threshold, GPU Coder does not map these variables to the GPU and kernels are not created. The code generator allocates memory dynamically on the CPU heap. GPU Coder issues a warning for unbounded variables in the build log and code generation report.

By default, the code generator is set to use dynamic memory allocation for variable-size arrays whose size is greater than or equal to the threshold with a threshold value of 2 GB. To change these settings:

- In the configuration object, set the `DynamicMemoryAllocation` to `Threshold` and `DynamicMemoryAllocationThreshold` to a non-negative integer.
- In the GPU Coder app, in the **Memory** settings, set **Dynamic memory allocation** to **For arrays with max size at or above threshold** and the **Dynamic memory allocation threshold** to a non-negative integer.

Variable-Size Arrays in a Code Generation Report

You can tell whether an array is fixed-size or variable-size by looking at the **Size** column of the **Variables** tab in a code generation report.

Name	Type	Size	Class
y	Output	1 × 1	double
A	Input	1 × :16	char
n	Input	1 × 1	double
X	Local	1 × :?	double

A colon (:) indicates that a dimension is variable-size. A question mark (?) indicates that the size is unbounded. For example, a size of 1-by-:? indicates that the size of the first dimension is fixed-size 1 and the size of the second dimension is unbounded, variable-size. An asterisk (*) indicates that the code generator produced a variable-size array, but the size of the array does not change during execution.

Variable	Type	Size
y	Output	1 × 2
n	Input	1 × 1
Z	Local	1 × 4 *

Structure Definition for Code Generation

To generate efficient standalone code for structures, you must define and use structures differently than you normally would when running your code in the MATLAB environment. For code generation, you must first create a scalar template version of the structure before growing it into an array. The code generation inference engine uses the type of this scalar value as the base type of the array. To generate standalone code for MATLAB structures, you are restricted to the following operations:

- Define structures as local and persistent variables by assignment and using the `struct` function
- Index structure fields using dot notation
- Define primary or entry-point function inputs as structures
- Pass structures to local functions

For more information, see “Structure Definition for Code Generation”.

Note GPU Coder generates more efficient code when you use `struct` of arrays instead of array of `structs`.

Example

This example shows how to write a MATLAB function that uses structure arrays so that it is suitable for code generation. First, you must specify the base element using the `struct` function.

```
tempS = struct('a',0,'b',0);  
numE = 2000;  
AofS = repmat(tempS,numE,1);
```

In MATLAB, when building up a structure array, you would typically add fields as you go. This "dynamic" style of building structures is not supported for code generation. One reason is that it is possible in MATLAB to have different structure fields for two different elements of a structure array, which conflicts with the more static approach of type inference. Therefore, you must specify the base scalar element first, and then grow a structure array from this fully specified element. This method guarantees that two elements of a structure array always share type (fields).

```
for ind = 1:numE  
    AofS(ind).a = rand;  
    AofS(ind).b = rand;  
end
```

Now, you can define an entry-point function `mStructSupport` that takes `AofS` as input. The local function `arrayOp` doubles `AofS.b` and stores the result in `AofS.a`.

```
function [V] = mStructSupport(AofS)  
    V = arrayOp(AofS);
```

```
end
```

```
function AofS = arrayOp(AofS)  
    n = numel(AofS);
```

```
    for i = 1:n  
        AofS(i).a = AofS(i).b * 2;  
    end
```


end

You can use any of the methods described in “Code Generation by Using the GPU Coder App” to generate CUDA code for this example.

Unsupported Features

The following list contains the features that are not currently supported.

- Memory integrity checks, see “Control Run-Time Checks”.
- Array bound and dimension checks.
- `break` statements.
- Function handles are supported only when defined within another function and not as entry-point parameter.
- Anonymous functions are supported only when defined within another function and not as an entry-point parameter.
- MATLAB classes.

Supported Functions

You can generate CUDA code for a subset of MATLAB built-in functions and toolbox functions that you call from MATLAB code. These functions appear in alphabetical order in the following table. Some of these functions especially from the Image Processing Toolbox™ contain calls to other functions, GPU Coder does not create CUDA kernels for all the loops and functions that the parent function relies on. However, GPU Coder does generate C/C++ code for sections that cannot be mapped to the GPU. The results from the code generated for functions in this list are also numerically equivalent (within tolerance) to its MATLAB counterpart. See, “Numerical Differences Between CPU and GPU”.

Link to an categorized list of all functions that support the GPU code generation: [Functions Supporting GPU Code Generation](#).

Kernel Creation from MATLAB Code

- “Kernels from Element-Wise Loops” on page 2-2
- “Kernels from Scatter-Gather Type Operations” on page 2-4
- “Kernels from Library Calls” on page 2-8
- “cuBLAS Example” on page 2-10
- “cuSOLVER Example” on page 2-12
- “FFT Example” on page 2-15
- “Thrust Example” on page 2-17
- “Call Custom CUDA Kernels from the Generated Code” on page 2-18
- “Call Custom CUDA Device Function from the Generated Code” on page 2-22
- “Design Patterns” on page 2-26
- “GPU Memory Allocation and Minimization” on page 2-28
- “Use Dynamically Allocated C++ Arrays in Generated Function Interfaces” on page 2-33
- “Support for GPU Arrays” on page 2-37
- “What is Half Precision?” on page 2-39
- “Half Precision Code Generation Support” on page 2-44
- “Simulate Diffraction Patterns Using CUDA FFT Libraries” on page 2-55
- “Benchmark Solving a Linear System by Using GPU Coder” on page 2-61
- “QR Decomposition on NVIDIA GPU Using cuSOLVER Libraries” on page 2-69
- “Stencil Processing on GPU” on page 2-74
- “Fog Rectification” on page 2-80
- “Stereo Disparity” on page 2-85
- “Feature Extraction Using SURF” on page 2-91
- “Feature Matching” on page 2-96
- “Lane Detection on the GPU by Using the houghlines Function” on page 2-100
- “Edge Detection with Sobel Method in Half-Precision” on page 2-103
- “Build a Map from Lidar Data using SLAM on GPU” on page 2-107

Kernels from Element-Wise Loops

The simplest case of CUDA kernel creation is from MATLAB functions that contain scalarized, element-wise math operations. When element-wise operations are enclosed within a for-loop body, concurrent CUDA threads can be invoked to compute each loop iteration in parallel. Because CUDA threads execute in no particular order, and are independent of each other, it is essential that no iteration in your for-loop depends on the results of other iterations.

Element-Wise Math Example

This example shows how to create CUDA kernels from functions that contain element-wise math operations. Suppose that you want to square each element of a matrix x and scale by a factor of $1/(i+j)$, where i, j are the row and column indexes. You can implement this example as a MATLAB function.

```
function [y] = myFun(x)

y = zeros(size(x));
for i = 1:size(x,1)
    for j = 1:size(x,2)
        y(i,j)=(x(i,j)^2)/(i+j);
    end
end
end
```

Preparing myFun for Code Generation

The first statement `zeros(size(A))` in the `myFun` function is to initialize result vector `y` to zeros. For CUDA code generation, pre-allocate memory for `y` without incurring the overhead of initializing the memory to zeros. Replace this line with `coder.nullcopy(zeros(size(y)))`.

To create CUDA kernels from loops, GPU Coder provides another pragma `coder.gpu.kernel`. Specifying this kernel pragma overrides all parallel-loop analysis. If you do not specify any parameters, GPU Coder determines the kernel bounds based on the loop bounds and input size. It provides a way for you to specify kernel launch parameters such as thread and block sizes. However, use it only when you know that the loop is safe to parallelize. Because the `myFun` example is simple and does not require specification of the kernel launch parameters, you can utilize the `coder.gpu.kernelfun` pragma to generate CUDA kernels.

With these modifications, the original `myFun` function is suitable for code generation.

```
function [y] = myFun(x) %#codegen

y = coder.nullcopy(zeros(size(x)));
coder.gpu.kernelfun();
for i = 1:size(x,1)
    for j = 1:size(x,2)
        y(i,j)=(x(i,j)^2)/(i+j);
    end
end
end
```

Generated CUDA Code

When you generate CUDA code by using the GPU Coder app or from the command line, GPU Coder creates a single kernel that performs squaring and scaling operation. The following is a snippet of the `myFun_kernel1` kernel code.

```
static __global__ __launch_bounds__(512, 1) void myFun_kernel1(const real_T *x,
    real_T *y)
{
    ...
    threadId = (((gridDim.x * gridDim.y * blockIdx.z + gridDim.x * blockIdx.y) +
        blockIdx.x) * (blockDim.x * blockDim.y * blockDim.z) +
        threadIdx.z * blockDim.x * blockDim.y) + threadIdx.y * blockDim.x)
        + threadIdx.x;
    i = (int32_T)(threadId / 512U);
    j = (int32_T)(threadId - (uint32_T)i * 512U);
    if (!(j <= 512) && !(i <= 512)) {
        y[i + (j << 9)] = x[i + (j << 9)] * x[i + (j << 9)] / ((real_T)(i + j) + 2.0);
    }
}
```

The following is a snippet of the main `myFun` function. Before calling `myFun_kernel1`, there is a single `cudaMemcpy` call that transfers the matrix `x` from the host (`x`) to the device (`gpu_x`). The kernel has 512 blocks containing 512 threads per block, consistent with the size of the input vector. A second `cudaMemcpy` call copies the result of the computation back to the host.

```
cudaMemcpy((void *)gpu_x, (void *)x, 2097152ULL, cudaMemcpyHostToDevice);
myFun_kernel1<<<dim3(512U, 1U, 1U), dim3(512U, 1U, 1U)>>>(gpu_x, gpu_y);
cudaMemcpy((void *)y, (void *)gpu_y, 2097152ULL, cudaMemcpyDeviceToHost);
```

Limitations

- If the loop bounds are of the unsigned data type, the code generator may add conditional checks to determine if the loop bounds are valid. These conditional checks may limit optimizations that are performed by the software and introduce reduction kernels that can affect performance.

See Also

[coder.gpu.kernel](#) | [coder.gpu.kernelfun](#) | [gpucoder.matrixMatrixKernel](#) | [coder.gpu.constantMemory](#) | [gpucoder.stencilKernel](#)

Related Examples

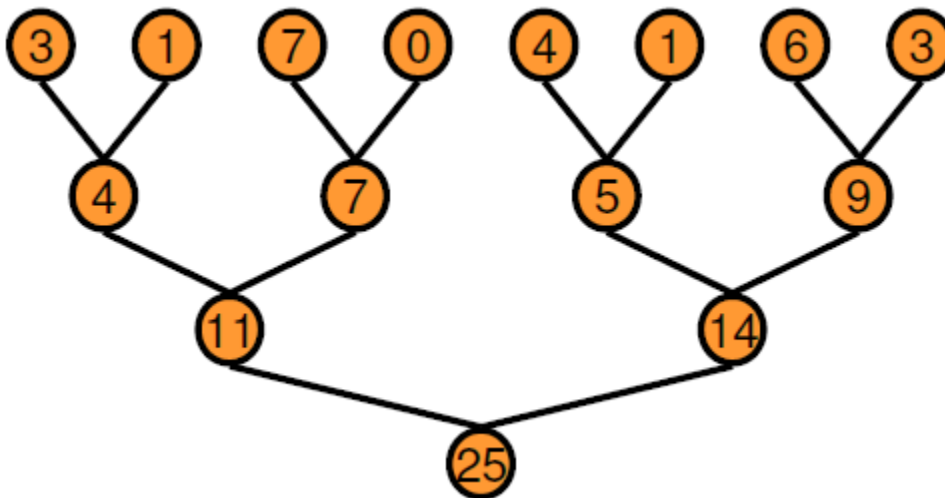
- “Design Patterns” on page 2-26
- “Kernels from Scatter-Gather Type Operations” on page 2-4
- “Kernels from Library Calls” on page 2-8
- “Call Custom CUDA Device Function from the Generated Code” on page 2-22

Kernels from Scatter-Gather Type Operations

GPU Coder also supports the concept of reductions - an important exception to the rule that loop iterations must be independent. A reduction variable accumulates a value that depends on all the iterations together, but is independent of the iteration order. Reduction variables appear on both side of an assignment statement, such as in summation, dot product, and sort. The following example shows the typical usage of a reduction variable `x`:

```
x = ...; % Some initialization of x
for i = 1:n
    x = x + d(i);
end
```

The variable `x` in each iteration gets its value either before entering the loop or from the previous iteration of the loop. This serial order type implementation is not suitable for parallel execution due to the chain of dependencies in the sequential execution. An alternative approach is to employ a binary tree-based approach.



In the tree-based approach, you can execute every horizontal level of the tree in parallel over a certain number of passes. When compared to sequential execution, the binary tree does require more memory because each pass requires an array of temporary values as output. The performance benefit that you receive far outweighs the cost of increased memory usage. GPU Coder creates reduction kernels by using this tree-based approach wherein each thread block reduces a portion of the array. Parallel reduction requires partial result data exchanges between thread blocks. In older CUDA devices, this data exchange was achieved by using shared memory and thread synchronization. Starting with the Kepler GPU architecture, CUDA provides shuffle (`shfl`) instruction and fast device memory atomic operations that make reductions even faster. Reduction kernels that the GPU Coder creates use the `shfl_down` instruction to reduce across a warp (32 threads) of threads. Then, the first thread of each warp uses the atomic operation instructions to update the reduced value.

For more information on the instructions, refer to the NVIDIA® documentation.

Vector Sum Example

This example shows how to create CUDA reduction type kernels by using GPU Coder. Suppose that you want to create a vector v and compute the sum of its elements. You can implement this example as a MATLAB function.

```
function s = VecSum(v)
    s = 0;
    for i = 1:length(v)
        s = s + v(i);
    end
end
```

Prepare vecSum for Kernel Creation

GPU Coder requires no special pragma to infer reduction kernels. In this example, use the `coder.gpu.kernelfun` pragma to generate CUDA reduction kernels. Use the modified `VecSum` function.

```
function s = VecSum(v) %#codegen
    s = 0;

    coder.gpu.kernelfun();
    for i = 1:length(v)
        s = s + v(i);
    end
end
```

Generated CUDA Code

When you generate CUDA code by using the GPU Coder app or from the command line, GPU Coder creates a single kernel that performs the vector sum calculation. The following is a snippet of `vecSum_kernel1`.

```
static __global__ __launch_bounds__(512, 1) void vecSum_kernel1(const real_T *v,
    real_T *s)
{
    uint32_T threadIdx;
    uint32_T threadIdx;
    uint32_T thdBlkId;
    uint32_T idx;
    real_T tmpRed;
    ;
    ;
    thdBlkId = (threadIdx.z * blockDim.x * blockDim.y + threadIdx.y * blockDim.x)
        + threadIdx.x;
    threadIdx = ((gridDim.x * gridDim.y * blockIdx.z + gridDim.x * blockIdx.y) +
        blockIdx.x) * (blockDim.x * blockDim.y * blockDim.z) + thdBlkId;
    threadIdx = gridDim.x * blockDim.x * (gridDim.y * blockDim.y) * (gridDim.z *
        blockDim.z);
    if (!(int32_T)threadIdx >= 512) {
        tmpRed = 0.0;
        for (idx = threadIdx; threadIdx < 0U ? idx >= 511U : idx <= 511U; idx +=
            threadIdx) {
            tmpRed += v[idx];
        }

        tmpRed = workGroupReduction1(tmpRed, 0.0);
        if (thdBlkId == 0U) {
            atomicOp1(s, tmpRed);
        }
    }
}
```

Before calling `VecSum_kernel1`, two `cudaMemcpy` calls transfer the vector `v` and the scalar `s` from the host to the device. The kernel has one thread block containing 512 threads per block, consistent with the size of the input vector. A third `cudaMemcpy` call copies the result of the computation back to the host. The following is a snippet of the main function.

```
cudaMemcpy((void *)gpu_v, (void *)v, 4096ULL, cudaMemcpyHostToDevice);
cudaMemcpy((void *)gpu_s, (void *)&s, 8ULL, cudaMemcpyHostToDevice);
VecSum_kernel1<<<dim3(1U, 1U, 1U), dim3(512U, 1U, 1U)>>>(gpu_v, gpu_s);
cudaMemcpy(&s, gpu_s, 8U, cudaMemcpyDeviceToHost);
```

Note For better performance, GPU Coder gives priority to parallel kernels over reductions. If your algorithm contains reduction inside a parallel loop, GPU Coder infers the reduction as a regular loop and generates kernels for it.

1-D Reduction Operations on the GPU

You can use the `gpuCoder.reduce` function to generate CUDA code that performs efficient 1-D reduction operations on the GPU. The generated code uses the CUDA shuffle intrinsics to implement the reduction operation.

For example, to find the sum and max elements of an array `A`:

```
function s = myReduce(A)
    s = gpuCoder.reduce(A, {@mysum, @mymax});
end

function c = mysum(a, b)
    c = a+b;
end

function c = mymax(a, b)
    c = max(a,b);
end
```

For code generation, the `gpuCoder.reduce` function has these requirements:

- The input must be of numeric or logical data type.
- The function passed through the `@handle` must be a binary function that accepts two inputs and returns one output. The inputs and outputs must be of the same data type.
- The function must be commutative and associative.

Note For some inputs that are of the integer data type, the code generated for the `gpuCoder.reduce` function may contain intermediate computations that reach saturation. In such cases, the results from the generated code may not match the simulation results from MATLAB.

See Also

`coder.gpu.kernel` | `coder.gpu.kernelfun` | `gpuCoder.matrixMatrixKernel` | `coder.gpu.constantMemory` | `gpuCoder.stencilKernel` | `gpuCoder.reduce`

Related Examples

- “Design Patterns” on page 2-26

- “Kernels from Element-Wise Loops” on page 2-2
- “Kernels from Library Calls” on page 2-8
- “Call Custom CUDA Device Function from the Generated Code” on page 2-22

Kernels from Library Calls

GPU Coder supports libraries optimized for CUDA GPUs such as cuBLAS, cuSOLVER, cuFFT, Thrust, cuDNN, and TensorRT libraries.

- The cuBLAS library is an implementation of Basic Linear algebra Subprograms (BLAS) on top of the NVIDIA CUDA run time. It allows you to access the computational resources of the NVIDIA GPU.
- The cuSOLVER library is a high-level package based on the cuBLAS and cuSPARSE libraries. It provides useful LAPACK-like features, such as common matrix factorization and triangular solve routines for dense matrices, a sparse least-squares solver, and an Eigenvalue solver.
- The cuFFT library provides a high-performance implementation of the Fast Fourier Transform (FFT) algorithm on NVIDIA GPUs. The cuBLAS, cuSOLVER, and cuFFT libraries are part of the NVIDIA CUDA Toolkit.
- Thrust is a C++ template library for CUDA. The Thrust library is shipped with CUDA Toolkit and allows you to take advantage of GPU-accelerated primitives such as sort to implement complex high-performance parallel applications.
- The NVIDIA CUDA Deep Neural Network library (cuDNN) is a GPU-accelerated library of primitives for deep neural networks. cuDNN provides highly tuned implementations for standard routines such as forward and backward convolution, pooling, normalization, and activation layers. The NVIDIA TensorRT is a high performance deep learning inference optimizer and runtime library. For more information, see “Code Generation for Deep Learning Networks by Using cuDNN” on page 4-69 and “Code Generation for Deep Learning Networks by Using TensorRT” on page 4-78.

GPU Coder does not require a special pragma to generate kernel calls to libraries. During the code generation process, when you select the **Enable cuBLAS** option in the GPU Coder app or use `config_object.GpuConfig.EnableCUBLAS = true` property in CLI, GPU Coder replaces some functionality with calls to the cuBLAS library. When you select the **Enable cuSOLVER** option in the GPU Coder app or use `config_object.GpuConfig.EnableCUSOLVER = true` property in CLI, GPU Coder replaces some functionality with calls to the cuSOLVER library. For GPU Coder to replace high-level math functions to library calls, the following conditions must be met:

- GPU-specific library replacement must exist for these functions.
- MATLAB Coder data size thresholds must be satisfied.

GPU Coder supports cuFFT, cuSOLVER, and cuBLAS library replacements for the functions listed in the table. For functions that have no replacements in CUDA, GPU Coder uses portable MATLAB functions that are mapped to the GPU.

MATLAB Function	Description	MATLAB Coder LAPACK Support	cuBLAS, cuSOLVER, cuFFT, Thrust Support
<code>mtimes</code>	Matrix multiply	Yes	Yes
<code>mldivide ('\'')</code>	Solve system of linear equation $Ax=B$ for x	Yes	Yes
<code>lu</code>	LU matrix factorization	Yes	Yes
<code>qr</code>	Orthogonal-triangular decomposition	Yes	Partial
<code>det</code>	Matrix determinant	Yes	Yes

MATLAB Function	Description	MATLAB Coder LAPACK Support	cuBLAS, cuSOLVER, cuFFT, Thrust Support
chol	Cholesky factorization	Yes	Yes
rcond	Reciprocal condition number	Yes	Yes
linsolve	Solve system of linear equations $Ax=B$	Yes	Yes
eig	Eigenvalues and eigen vectors	Yes	No
schur	Schur decomposition	Yes	No
svd	Singular value decomposition	Yes	Partial
fft, fft2, fftn	Fast Fourier Transform	Yes	Yes
ifft, ifft2, ifftn	Inverse Fast Fourier Transform	Yes	Yes
sort	Sort array elements		Yes, using <code>gpcoder.sort</code>

When you select the **Enable cuFFT** option in the GPU Coder app or use `config_object.GpuConfig.EnableCUFFT = true` property in CLI, GPU Coder maps `fft, ifft, fft2, ifft2, fftn, ifftn` function calls in your MATLAB code to the appropriate cuFFT library calls. For 2-D transforms and higher, GPU Coder creates multiple 1-D batched transforms. These batched transforms have higher performance than single transforms. GPU Coder only supports out-of-place transforms. If **Enable cuFFT** is not selected, GPU Coder uses C FFTW libraries where available or generates kernels from portable MATLAB FFT. Both single and double precision data types are supported. Input and output can be real or complex-valued, but real-valued transforms are faster. cuFFT library support input sizes that are typically specified as a power of 2 or a value that can be factored into a product of small prime numbers. In general the smaller the prime factor, the better the performance.

Note Using CUDA library names such as `cufft`, `cuBLAS`, and `cuDNN` as the names of your MATLAB function results in code generation errors.

See Also

`coder.gpu.kernel` | `coder.gpu.kernelfun` | `gpcoder.matrixMatrixKernel` | `coder.gpu.constantMemory` | `gpcoder.stencilKernel` | `gpcoder.sort`

Related Examples

- “Design Patterns” on page 2-26
- “Kernels from Element-Wise Loops” on page 2-2
- “Kernels from Scatter-Gather Type Operations” on page 2-4
- “Call Custom CUDA Device Function from the Generated Code” on page 2-22

cuBLAS Example

This example multiplies two matrices A and B by using the cuBLAS library. The MATLAB implementation of GEneral Matrix-Matrix Multiplication (GEMM) is:

```
function [C] = blas_gemm(A,B)

    C = zeros(size(A));
    C = A * B;
end
```

Generated CUDA Code

When you generate CUDA code, GPU Coder creates function calls to initialize the cuBLAS library, perform matrix-matrix operations, and release hardware resources that the cuBLAS library uses. The following is a snippet of the generated CUDA code.

```
cublasEnsureInitialization();
blas_gemm_kernel1<<<dim3(2048U, 1U, 1U), dim3(512U, 1U, 1U)>>>(gpu_C);
alpha = 1.0;
beta = 0.0;
cudaMemcpy((void *)gpu_alpha, (void *)&alpha, 8ULL, cudaMemcpyHostToDevice);
cudaMemcpy((void *)gpu_A, (void *)A, 8388608ULL, cudaMemcpyHostToDevice);
cudaMemcpy((void *)gpu_B, (void *)B, 8388608ULL, cudaMemcpyHostToDevice);
cudaMemcpy(gpu_beta, &beta, 8ULL, cudaMemcpyHostToDevice);
cublasDgemm(cublasGlobalHandle, CUBLAS_OP_N, CUBLAS_OP_N, 1024, 1024, 1024,
            (double *)gpu_alpha, (double *)&gpu_A[0], 1024, (double *)&gpu_B
            [0], 1024, (double *)gpu_beta, (double *)&gpu_C[0], 1024);
cublasEnsureDestruction();
cudaMemcpy((void *)C, (void *)gpu_C, 8388608ULL, cudaMemcpyDeviceToHost);
```

To initialize the cuBLAS library and create a handle to the cuBLAS library context, the function `cublasEnsureInitialization()` calls `cublasCreate()` cuBLAS API. It allocates hardware resources on the host and device.

```
static void cublasEnsureInitialization(void)
{
    if (cublasGlobalHandle == NULL) {
        cublasCreate(&cublasGlobalHandle);
        cublasSetPointerMode(cublasGlobalHandle, CUBLAS_POINTER_MODE_DEVICE);
    }
}
```

`blas_gemm_kernel1` initializes the result matrix C to zero. This kernel is launched with 2048 blocks and 512 threads per block. These block and thread values correspond to the size of C.

```
static __global__ __launch_bounds__(512, 1) void blas_gemm_kernel1(real_T *C)
{
    int32_T threadIdx;
    threadIdx = (int32_T)(blockDim.x * blockIdx.x + threadIdx.x);
    if (!(threadIdx >= 1048576)) {
        C[threadIdx] = 0.0;
    }
}
```

Calls to `cudaMemcpy` transfer the matrices A and B from the host to the device. The function `cublasDgemm` is a level-3 Basic Linear Algebra Subprogram (BLAS3) that performs the matrix-matrix multiplication:

$$C = \alpha AB + \beta C$$

where α and β are scalars, and A, B, and C are matrices stored in column-major format. `CUBLAS_OP_N` controls transpose operations on the input matrices.

The final calls are to `cublasEnsureDestruction()` and another `cudaMemcpy`. `cublasEnsureDestruction()` calls `cublasCreate()` cuBLAS API to release hardware resources the cuBLAS library uses. `cudaMemcpy` copies the result matrix C from the device to the host.

```
static void cublasEnsureDestruction(void)
{
    if (cublasGlobalHandle != NULL) {
        cublasDestroy(cublasGlobalHandle);
        cublasGlobalHandle = NULL;
    }
}
```

Prepare blas_gemm for Kernel Creation

GPU Coder requires no special pragma to generate calls to libraries. There are two ways to generate CUDA kernels — `coder.gpu.kernelfun` and `coder.gpu.kernel`. In this example, we utilize the `coder.gpu.kernelfun` pragma to generate CUDA kernels. The modified `blas_gemm` function is:

```
function [C] = blas_gemm(A,B) %#codegen
    C = coder.nullcopy(zeros(size(A)));

    coder.gpu.kernelfun;
    C = A * B;
end
```

Note A minimum size (128 elements) is required on the input data for replacing math operators and functions with cuBLAS library implementations.

cuSOLVER Example

This example solves the systems of linear equations $Ax = B$ for x by using the cuSOLVER library. The matrices A and B must have the same number of rows. If A is a scalar, then $A \setminus B$ is equivalent to $A \cdot \setminus B$. If A is a square n -by- n matrix and B is a matrix with n rows, then $x = A \setminus B$ is a solution to the equation $A * x = B$, if it exists. The MATLAB implementation of backslash is:

```
function [x] = backslash(A,b)
if (isscalar(A))
    x = coder.nullcopy(zeros(size(b)));
else
    x = coder.nullcopy(zeros(size(A,2),size(b,2)));
end

x = A\b;

end
```

Prepare backslash for Kernel Creation

GPU Coder requires no special pragma to generate calls to libraries. Just as before, there are two ways to generate CUDA kernels — `coder.gpu.kernelfun` and `coder.gpu.kernel`. In this example, we utilize the `coder.gpu.kernelfun` pragma to generate CUDA kernels. The modified backslash function is:

```
function [x] = backslash(A,b) %#codegen

if (isscalar(A))
    x = coder.nullcopy(zeros(size(b)));
else
    x = coder.nullcopy(zeros(size(A,2),size(b,2)));
end

coder.gpu.kernelfun()
x = A\b;

end
```

Note A minimum size is required on the input data for replacing math operators and functions with cuSOLVER library implementations. The minimum threshold is 128 elements.

Generated CUDA Code

When you generate CUDA code, GPU Coder creates function calls to initialize the cuSOLVER library, perform `mldivide` operations, and release hardware resources that the cuSOLVER library uses. A snippet of the generated CUDA code is:

```
cusolverEnsureInitialization();

/* Copyright 2017 The MathWorks, Inc. */
cudaMemcpy(b_gpu_A, A, 1152UL, cudaMemcpyHostToDevice);
backslash_kernel1<<<dim3(1U, 1U, 1U), dim3(160U, 1U, 1U)>>>(b_gpu_A,gpu_A);
cudaMemcpy(b_A, gpu_A, 1152UL, cudaMemcpyDeviceToHost);
cusolverDnDgetrf_bufferSize(cusolverGlobalHandle, 12, 12, &gpu_A[0], 12,
    &cusolverWorkspaceReq);
cusolverWorkspaceTypeSize = 8;
```

```

cusolverInitWorkspace();
cudaMemcpy(gpu_A, b_A, 1152UL, cudaMemcpyHostToDevice);
cusolverDnDgetrf(cusolverGlobalHandle, 12, 12, &gpu_A[0], 12, (real_T *)
    cusolverWorkspaceBuff, &gpu_ipiv_t[0], gpu_info_t);
A_dirtyOnGpu = true;
cudaMemcpy(&info_t, gpu_info_t, 4UL, cudaMemcpyDeviceToHost);

```

To initialize the cuSOLVER library and create a handle to the cuSOLVER library context, the function `cusolversEnsureInitialization()` calls `cusolverDnCreate()` cuSOLVER API. It allocates hardware resources on the host and device.

```

static void cusolverEnsureInitialization(void)
{
    if (cusolverGlobalHandle == NULL) {
        cusolverDnCreate(&cuSolverGlobalHandle);
    }
}

```

`backslash_kernel1` zero pads the matrix A. This kernel is launched with a single block of 512 threads.

```

static __global__ __launch_bounds__(160, 1) void backslash_kernel1(const real_T *
    A, real_T *b_A)
{
    int32_T threadId;
    ;
    ;
    threadId = (int32_T)(((gridDim.x * gridDim.y * blockDim.z + gridDim.x *
        blockDim.y) + blockDim.x) * (blockDim.x * blockDim.y * blockDim.z) +
        (int32_T)((threadIdx.z * blockDim.x * blockDim.y +
            threadIdx.y * blockDim.x) + threadIdx.x));
    if (!(threadId >= 144)) {
        /* Copyright 2017 The MathWorks, Inc. */
        b_A[threadId] = A[threadId];
    }
}

```

Calls to `cudaMemcpy` transfer the matrix A from the host to the device. The function `cusolverDnDgetrf` computes the LU factorization of an $m \times n$ matrix:

$$P * A = L * U$$

where A is an $m \times n$ matrix, P is a permutation matrix, L is a lower triangular matrix with unit diagonal, and U is an upper triangular matrix.

cuSOLVER Standalone Code

For functions like `qr` that only have partial support in cuSOLVER, GPU Coder uses LAPACK library where necessary. For MEX functions, the code generator uses the LAPACK library that is included with MATLAB. For standalone code, the code generator uses the LAPACK library that you specify. To specify the LAPACK library:

- At the command line, define your own `coder.LAPACKCallback` class containing the LAPACK library information and assign it to the `CustomLAPACKCallback` property of the code configuration object.
- In the GPU Coder app, set Custom LAPACK library callback to your LAPACK library.

For example, to generate a standalone executable, you can use the following code generation script. Here `myLAPACK` is the name of the custom `coder.LAPACKCallback` class containing the LAPACK library information.

```

cfg = coder.gpuConfig('exe');
cfg.CustomLAPACKCallback = 'myLAPACK';

```

```
cfg.GenerateExampleMain = 'GenerateCodeAndCompile';  
  
classdef myLAPACK < coder.LAPACKCallback  
    methods (Static)  
        function hn = getHeaderFilename()  
            hn = 'lapacke.h';  
        end  
        function updateBuildInfo(buildInfo, buildctx)  
            [~,linkLibExt] = buildctx.getStdLibInfo();  
            cudaPath = getenv('CUDA_PATH');  
            libPath = 'lib\x64';  
  
            buildInfo.addIncludePaths(fullfile(cudaPath,'include'));  
            libName = 'cusolver';  
            libPath = fullfile(cudaPath,libPath);  
            buildInfo.addLinkObjects([libName linkLibExt], libPath, ...  
                '', true, true);  
  
            lapackLocation = 'C:\LAPACK\win64'; % specify path to LAPACK libraries  
  
            includePath = fullfile(lapackLocation,'include');  
            buildInfo.addIncludePaths(includePath);  
            libPath = fullfile(lapackLocation,'lib');  
            libName = 'mllapack';  
  
            buildInfo.addLinkObjects([libName linkLibExt], libPath, ...  
                '', true, true);  
            buildInfo.addDefines('HAVE_LAPACK_CONFIG_H');  
            buildInfo.addDefines('LAPACK_COMPLEX_STRUCTURE');  
        end  
    end  
end
```

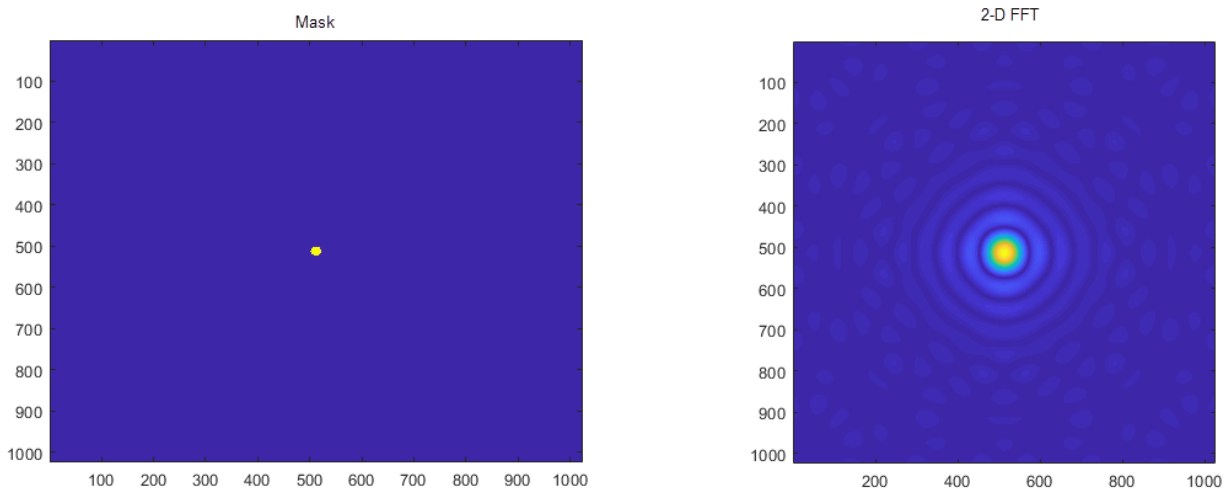
For more information, see “Speed Up Linear Algebra in Generated Standalone Code by Using LAPACK Calls”.

FFT Example

This example shows how a two-dimensional Fourier transform can be used on an optical mask to compute its diffraction pattern. Create a logical array that defines an optical mask with a small, circular aperture.

```
n = 2^10; % size of mask
M = zeros(n);
I = 1:n;
x = I-n/2; % mask x-coordinates
y = n/2-I; % mask y-coordinates
[X,Y] = meshgrid(x,y); % create 2-D mask grid
R = 10; % aperture radius
A = (X.^2 + Y.^2 <= R^2); % circular aperture of radius R
M(A) = 1; % set mask elements inside aperture to 1
figure
imagesc(M) % plot mask
axis image

DP = fftshift(fft2(M));
imagesc(abs(DP))
axis image
```



Prepare myFFT for Kernel Creation

Create an entry-point function `myFFT` that computes the 2-D Fourier transform of the mask by using the `fft2` function. Use the `fftshift` function to rearrange the output so that the zero-frequency component is at the center. To map this function to a GPU kernel, place the coder `.gpu.kernelfun` pragma within the function.

```
function [DP] = myFFT(M)

coder.gpu.kernelfun();

DP = fftshift(fft2(M));
```

Generated CUDA Code

When you generate CUDA code, GPU Coder creates function calls (`cufftEnsureInitialization`) to initialize the cuFFT library, perform FFT operations, and release hardware resources that the cuFFT library uses. A snippet of the generated CUDA code is:

```
void myFFT(myFFTStackData *SD, const real_T M[1048576], creal_T DP[1048576])
{
    ...
    cudaMemcpy((void *)gpu_M, (void *)M, 8388608ULL, cudaMemcpyHostToDevice);
    myFFT_kernel1<<<dim3(2048U, 1U, 1U), dim3(512U, 1U, 1U)>>>(gpu_M, gpu_b);
    cufftEnsureInitialization(1024, CUFFT_D2Z, 1024, 1024);
    cufftExecD2Z(*cufftGlobalHandlePtr, (cufftDoubleReal *)&gpu_b[0],
                (cufftDoubleComplex *)&gpu_y1[0]);
    ...
    myFFT_kernel2<<<dim3(2048U, 1U, 1U), dim3(512U, 1U, 1U)>>>(gpu_y1, gpu_y);
    cufftEnsureInitialization(1024, CUFFT_Z2Z, 1024, 1024);
    cufftExecZ2Z(*cufftGlobalHandlePtr, (cufftDoubleComplex *)&gpu_y[0],
                (cufftDoubleComplex *)&gpu_DP[0], CUFFT_FORWARD);
    ...
    cufftEnsureDestruction();
    ...
}
```

The first `cudaMemcpy` function call transfers the 1024x1024 double-valued input `M` to the GPU memory. The `myFFT_kernel1` kernel performs pre-processing of the input data before the cuFFT library calls. The two-dimensional Fourier transform call `fft2` is equivalent to computing `fft(fft(M) .') .'`. Because batched transforms generally have higher performance compared to single transforms, GPU Coder has two 1-D cuFFT calls `cufftExecD2Z` to compute the double-precision real-to-complex forward transform of the input `M` followed by `cufftExecZ2Z` to perform the double-precision complex-to-complex transform of the result. The `cufftEnsureDestruction()` call destroys and frees all GPU resources associated with the cuFFT library call.

Thrust Example

With Thrust library support in GPU Coder, you can take advantage of GPU-accelerated primitives such as `sort` to implement complex high-performance parallel applications. When your MATLAB code uses `gpuCoder.sort` function instead of `sort`, GPU Coder can generate calls to the Thrust sort primitives.

This example generates CUDA code to sort the columns of a matrix in descending order. In one file, write an entry-point function `mySort` that accepts a matrix inputs `A`. Use the `gpuCoder.sort` function to sort the columns of `A` in descending order.

```
function B = mySort(A)
    B = gpuCoder.sort(A, 1, 'descend');
end
```

Use the `codegen` function to generate CUDA MEX function.

```
codegen -config coder.gpuConfig('mex') -args {ones(1024,1024,'double')} -report mySort
```

Generated CUDA Code

The following is a snippet of the generated code. The Thrust library call is denoted by `thrustSortImpl`

```
...
cudaMalloc(&gpu_inDims, 8ULL);
cudaMalloc(&gpu_B, 8388608ULL);
cudaMalloc(&gpu_A, 8388608ULL);
mySort_kernel1<<<dim3(1U, 1U, 1U), dim3(32U, 1U, 1U)>>>(*gpu_inDims);
cudaMemcpy(gpu_A, (void *)&A[0], 8388608ULL, cudaMemcpyHostToDevice);
mySort_kernel2<<<dim3(2048U, 1U, 1U), dim3(512U, 1U, 1U)>>>(*gpu_A, *gpu_B);
cudaMemcpy(&inDims[0], gpu_inDims, 8ULL, cudaMemcpyDeviceToHost);
thrustSortImpl(&*gpu_B[0], 2, &inDims[0], 1, 'd', false);
cudaMemcpy(&B[0], gpu_B, 8388608ULL, cudaMemcpyDeviceToHost);
...
```

Call Custom CUDA Kernels from the Generated Code

From within your MATLAB code, you can directly call external CUDA kernels, also called custom code or legacy code. To call CUDA kernels, use `coder.ceval`. The code generator integrates your CUDA kernel into the CUDA code generated from MATLAB. Integrate code when there are external libraries, optimized code, or object files developed using CUDA that you want to use with your generated code.

The external CUDA kernel must use the `__global__` qualifier to execute the function (kernels) on the GPU device and to call the function from the host or from the device. Functions with the `__device__` qualifier are called device functions. The device functions are different from global functions in that they can only be called from other device or global functions. For information on integrating custom device functions, see “Call Custom CUDA Device Function from the Generated Code” on page 2-22.

Note Use `coder.ceval` only in MATLAB code intended for code generation. `coder.ceval` generates an error in uncompiled MATLAB code. To determine if a MATLAB function is executing in MATLAB, use `coder.target`. If the function is executing in MATLAB, call the MATLAB version of the CUDA kernel.

Call Custom CUDA Kernel

This example shows how to integrate a simple CUDA kernel with MATLAB code by using `coder.ceval`. Consider the MATLAB function, `saxpy.m`:

```
type saxpy.m
function y = saxpy(a,x,y)
    y = a*x + y;
end
```

For this example, suppose that you want to implement the $a \times x$ plus y operation by using external CUDA kernel. Consider the CUDA kernel, `saxpy_kernel`, implemented in the file `saxpy.cu`:

```
type saxpy.cu
#include "saxpy.h"

__global__
void saxpy_kernel(uint32_T n, real32_T a, real32_T *x, real32_T *y)
{
    int i = blockIdx.x*blockDim.x + threadIdx.x;
    if (i < n) y[i] = a*x[i] + y[i];
}
```

To integrate `saxpy_kernel` with your MATLAB code, you need a header file that contains the function prototype. See the file `saxpy.h`:

```
type saxpy.h
#ifndef real32_T
#define real32_T float
#define uint32_T unsigned int
```

```
#endif

#define saxpy(grid,block,n,a,x,y) saxpy_kernel<<<grid,block>>>(n,a,x,y)

__global__ void saxpy_kernel(uint32_T n, real32_T a, real32_T *x, real32_T *y);
```

This example generates CUDA MEX, `uint32_T` and `real32_T` are custom types used in the generated MEX code. The code generator produces data types in CUDA code that correspond to the data types that you use in your MATLAB code. The data types that are generated depend on the target platform and compiler. The code generator can produce either built-in CUDA/C++ data types, such as `short`, `long`, `int`, and so on, or custom data types defined by using `typedef` statements. By default, the code generator produces built-in types for standalone code (`lib`, `dll`, or `exe`) and custom types for MEX code. For more information, see “Mapping MATLAB Types to Types in Generated Code”.

Entry-Point Function

Use the `coder.ceval` command to call the CUDA kernel in the `saxpyRef.m` entry-point function. Use `coder.ceval` only in MATLAB code intended for code generation. The `coder.rref` and `coder.ref` commands instruct the code generator to pass pointers to the arrays, rather than copy them.

type `saxpyRef.m`

```
function y = saxpyRef(a,x,y)
% saxpyRef Entry-point function for computing single-precision (A*X) Plus
% Y

% Copyright 2022 The MathWorks, Inc.
coder.gpu.kernelfun;

if coder.target('MATLAB')
    y = a*x + y;
else
    coder.ceval('saxpy', uint32(floor((numel(x)+255)/256)), uint32(256), ...
        uint32(numel(x)), single(a), coder.rref(x,'gpu'), ...
        coder.ref(y,'gpu'));
end
end
```

Generate CUDA Code

To generate CUDA code, create a GPU code configuration object. Specify the location of the custom CUDA files by setting custom code properties on the configuration object. For more information, see “Configure Build for External C/C++ Code”.

```
cfg = coder.gpuConfig("mex");
cfg.GenerateReport = true;
cfg.CustomSource = "saxpy.cu";
cfg.CustomInclude = pwd;
cfg.CustomSourceCode = '#include "saxpy.h"';

aType = coder.newtype('single', [1 1], [0 0]);
xType = coder.newtype('single', [4096 256], [0 0]);
yType = coder.newtype('single', [4096 256], [0 0]);
inputArgs = {aType,xType,yType};

codegen -config cfg saxpyRef -args inputArgs
```

Code generation successful: [View report](#)

Generated Code

To compare your generated CUDA code to the original MATLAB code, open the CUDA file, `saxpyRef.cu` in the `work\codegen\mex\saxpyRef` folder.

```
#include "saxpy.h"
// Function Definitions
void saxpyRef(real32_T a, const real32_T x[1048576], real32_T y[1048576])
{
    real32_T(*gpu_x)[1048576];
    real32_T(*gpu_y)[1048576];
    cudaMalloc(&gpu_y, 4194304UL);
    cudaMalloc(&gpu_x, 4194304UL);
    // saxpyRef Entry-point function for computing single-precision (A*X) Plus
    // Y
    // Copyright 2022 The MathWorks, Inc.
    cudaMemcpy(*gpu_x, x, 4194304UL, cudaMemcpyHostToDevice);
    cudaMemcpy(*gpu_y, y, 4194304UL, cudaMemcpyHostToDevice);
    saxpy(4096U, 256U, 1048576U, a, &(*gpu_x)[0], &(*gpu_y)[0]);
    cudaMemcpy(y, *gpu_y, 4194304UL, cudaMemcpyDeviceToHost);
    cudaFree(*gpu_x);
    cudaFree(*gpu_y);
}
```

Run Generated MEX

Run the generated MEX with random inputs and compare the results with MATLAB simulation.

```
a = single(15);
x = randi(10,4096,256,'single');
y = zeros(4096,256,'single');

yMATLAB = saxpyRef(a,x,y);
yGPU = saxpyRef_mex(a,x,y);

if (yGPU - yMATLAB == 0)
    fprintf('\nMATLAB simulation matches GPU execution.\n');
end
```

MATLAB simulation matches GPU execution.

See Also

Functions

`codegen` | `coder.wref` | `coder.ceval` | `coder.rref` | `coder.ref` | `coder.cinclude`

Objects

`coder.gpuConfig` | `coder.CodeConfig` | `coder.EmbeddedCodeConfig` | `coder.MexCodeConfig`

More About

- “Call Custom CUDA Device Function from the Generated Code” on page 2-22
- “Configure Build for External C/C++ Code”

- “Mapping MATLAB Types to Types in Generated Code”
- “Kernels from Element-Wise Loops” on page 2-2

Call Custom CUDA Device Function from the Generated Code

If you have highly optimized CUDA code for certain subfunctions that you want to incorporate into your generated code, GPU Coder extends the `coder.ceval` functionality to help you achieve this goal.

The external CUDA function must use the `__device__` qualifier to execute the function on the GPU device. These device functions are different from global functions (kernels) in that they can only be called from other device or global functions. Therefore the `coder.ceval` calls to the device functions must be from within a loop that gets mapped to a kernel. For information on integrating CUDA kernels with the generated code, see “Call Custom CUDA Kernels from the Generated Code” on page 2-18.

Note Code generation fails if the loop containing the `coder.ceval` calls cannot be mapped to a kernel. See the troubleshooting topic in the GPU Coder documentation to check for issues preventing kernel creation and their suggested workarounds. If your MATLAB code section contains unsupported functions, then you must remove the `coder.ceval` calls from such sections.

Call `__usad4_wrap` CUDA Device Function

The stereo disparity example measures the distance between two corresponding points in the left and the right image of a stereo pair. The `stereoDisparity_cuda_sample` entry-point function calls the `__usad4_wrap` external device function by using the `coder.ceval` function.

```
%% modified algorithm for stereo disparity block matching
%% In this implementation instead of finding shifted image ,indices are mapped
%% accordingly to save memory and some processing RGBA column major packed
%% data is used as input for compatibility with CUDA intrinsics. Convolution
%% is performed using separable filters (Horizontal and then Vertical)

function [out_disp] = stereoDisparity_cuda_sample(img0,img1)
coder.cinclude('cuda_intrinsic.h');

% gpu code generation pragma
coder.gpu.kernelfun;

%% Stereo disparity Parameters
% WIN_RAD is the radius of the window to be operated,min_disparity is the
% minimum disparity level the search continues for, max_disparity is the maximum
% disparity level the search continues for.
WIN_RAD = 8;
min_disparity = -16;
max_disparity = 0;

%% Image dimensions for loop control
% The number of channels packed are 4 (RGBA) so as nChannels are 4
[imgHeight,imgWidth]=size(img0);
nChannels = 4;
imgHeight = imgHeight/nChannels;

%% To store the raw differences
diff_img = zeros([imgHeight+2*WIN_RAD,imgWidth+2*WIN_RAD],'int32');

%%To store the minimum cost
min_cost = zeros([imgHeight,imgWidth],'int32');
min_cost(:,:,) = 99999999;

% Store the final disparity
out_disp = zeros([imgHeight,imgWidth],'int16');

%% Filters for aggregating the differences
% filter_h is the horizontal filter used in separable convolution
% filter_v is the vertical filter used in separable convolution which
% operates on the output of the row convolution
filt_h = ones([1 17],'int32');
filt_v = ones([17 1],'int32');
```



```

%% Main Loop that runs for all the disparity levels. This loop is currently
% expected to run on CPU.
for d=min_disparity:max_disparity

    % Find the difference matrix for the current disparity level. Expect
    % this to generate a Kernel function.
    coder.gpu.kernel;
    for colIdx=1:imgWidth+2*WIN_RAD
        coder.gpu.kernel;
        for rowIdx=1:imgHeight+2*WIN_RAD
            % Row index calculation
            ind_h = rowIdx - WIN_RAD;

            % Column indices calculation for left image
            ind_w1 = colIdx - WIN_RAD;

            % Row indices calculation for right image
            ind_w2 = colIdx + d - WIN_RAD;

            % Border clamping for row Indices
            if ind_h <= 0
                ind_h = 1;
            end
            if ind_h > imgHeight
                ind_h = imgHeight;
            end

            % Border clamping for column indices for left image
            if ind_w1 <= 0
                ind_w1 = 1;
            end
            if ind_w1 > imgWidth
                ind_w1 = imgWidth;
            end

            % Border clamping for column indices for right image
            if ind_w2 <= 0
                ind_w2 = 1;
            end
            if ind_w2 > imgWidth
                ind_w2 = imgWidth;
            end

            % In this step, Sum of absolute Differences is performed
            % across Four channels. This piece of code is suitable
            % for replacement with SAD intrinsics
            tDiff = int32(0);
            tDiff = coder.ceval('-gpudevifcfn', '__usad4_wrap',
                coder.rref(img0((ind_h-1)*(nChannels)+1,ind_w1)),
                coder.rref(img1((ind_h-1)*(nChannels)+1,ind_w2)));

            %Store the SAD cost into a matrix
            diff_img(rowIdx,colIdx) = tDiff;
        end
    end

    % Aggregating the differences using separable convolution. Expect this
    % to generate two Kernel using shared memory.The first kernel is the
    % convolution with the horizontal kernel and second kernel operates on
    % its output the column wise convolution.
    cost_v = conv2(diff_img,filt_h,'valid');
    cost = conv2(cost_v,filt_v,'valid');

    % This part updates the min_cost matrix with by comparing the values
    % with current disparity level. Expect to generate a Kernel for this.
    for ll=1:imgWidth
        for kk=1:imgHeight
            % load the cost
            temp_cost = int32(cost(kk,ll));

            % compare against the minimum cost available and store the
            % disparity value
            if min_cost(kk,ll) > temp_cost
                min_cost(kk,ll) = temp_cost;
                out_disp(kk,ll) = abs(d) + 8;
            end
        end
    end
end
end

```

```
end
end
```

The definition for the `__usad4_wrap` is written in an external file `cuda_intrinsic.h`. The file is located in the same folder as the entry-point function.

```
__device__ unsigned int __usad4(unsigned int A, unsigned int B, unsigned int C=0)
{
    unsigned int result;
    #if (__CUDA_ARCH__ >= 300) // Kepler (SM 3.x) supports a 4 vector SAD SIMD
        asm("vabsdiff4.u32.u32.u32.add" " %0, %1, %2, %3;": "=r"(result):"r"(A),
            "r"(B), "r"(C));
    #else // SM 2.0          // Fermi (SM 2.x) supports only 1 SAD SIMD,
                            // so there are 4 instructions
        asm("vabsdiff.u32.u32.u32.add" " %0, %1.b0, %2.b0, %3;":
            "=r"(result):"r"(A), "r"(B), "r"(C));
        asm("vabsdiff.u32.u32.u32.add" " %0, %1.b1, %2.b1, %3;":
            "=r"(result):"r"(A), "r"(B), "r"(result));
        asm("vabsdiff.u32.u32.u32.add" " %0, %1.b2, %2.b2, %3;":
            "=r"(result):"r"(A), "r"(B), "r"(result));
        asm("vabsdiff.u32.u32.u32.add" " %0, %1.b3, %2.b3, %3;":
            "=r"(result):"r"(A), "r"(B), "r"(result));
    #endif
    return result;
}

__device__ unsigned int packBytes(const uint8_T *inBytes)
{
    unsigned int packed = inBytes[0] | (inBytes[1] << 8) |
        (inBytes[2] << 16) | (inBytes[3] << 24);
    return packed;
}

__device__ unsigned int __usad4_wrap(const uint8_T *A, const uint8_T *B)
{
    unsigned int x = packBytes(A);
    unsigned int y = packBytes(B);

    return __usad4(x, y);
}
```

Generate CUDA Code

Generate CUDA code by creating a code configuration object. Specify the location of the custom C files by setting custom code properties (`CustomInclude`) on configuration objects. The following is an example code generation script that points to the location of `cuda_intrinsic.h` file.

```
cfg = coder.gpuConfig('mex');
cfg.CustomInclude = pwd;

codegen -config cfg -args {imgRGB0, imgRGB1} stereoDisparity_cuda_sample_intrinsic;
```

Generated Code

GPU Coder creates four kernels. The following is a snippet of the generated CUDA code.

```
e_stereoDisparity_cuda_sample_i<<<dim3(704U, 1U, 1U), dim3(512U, 1U, 1U)>>>
    (gpu_img1, gpu_img0, d, gpu_diff_img);*/
/* Aggregating the differences using separable convolution.*/
/* Expect this to generate two Kernel using shared memory.*/
/* The first kernel is the convolution with the horizontal kernel and*/
/* second kernel operates on its output the column wise convolution. */
f_stereoDisparity_cuda_sample_i<<<dim3(704U, 1U, 1U), dim3(512U, 1U, 1U)>>>
    (gpu_diff_img, gpu_a);
g_stereoDisparity_cuda_sample_i<<<dim3(18U, 20U, 1U), dim3(32U, 32U, 1U)>>>
    (gpu_a, gpu_cost_v);
h_stereoDisparity_cuda_sample_i<<<dim3(17U, 20U, 1U), dim3(32U, 32U, 1U)>>>
    (gpu_a, gpu_cost_v);
/* This part updates the min_cost matrix with by comparing the values */
/* with current disparity level. Expect to generate a Kernel for this. */
i_stereoDisparity_cuda_sample_i<<<dim3(667U, 1U, 1U), dim3(512U, 1U, 1U)>>>
    (d, gpu_cost, gpu_out_disp, gpu_min_cost);
```

The `e_stereoDisparity_cuda_sample_i` kernel is the one that calls the `__usad4_wrap` device function. The following is a snippet of `e_stereoDisparity_cuda_sample_i` kernel code.

```
static __global__ __launch_bounds__(512, 1) void e_stereoDisparity_cuda_sample_i
(const uint8_T *img1, const uint8_T *img0, int32_T d, int32_T *diff_img)
{
    ...
    /* In this step, Sum of absolute Differences is performed */
    /* across Four channels. This piece of code is suitable */
    /* for replacement with SAD intrinsics */
    temp_cost = __usad4_wrap(&img0[((ind_h - 1) << 2) + 2132 * (ind_w1 - 1)],
        &img1[((ind_h - 1) << 2) + 2132 * (temp_cost - 1)]);

    /* Store the SAD cost into a matrix */
    diff_img[rowIdx + 549 * colIdx] = temp_cost;
}
}
```

See Also

Functions

`codegen` | `coder.wref` | `coder.ceval` | `coder.rref` | `coder.ref` | `coder.cinclude`

Objects

`coder.gpuConfig` | `coder.CodeConfig` | `coder.EmbeddedCodeConfig` | `coder.MexCodeConfig`

More About

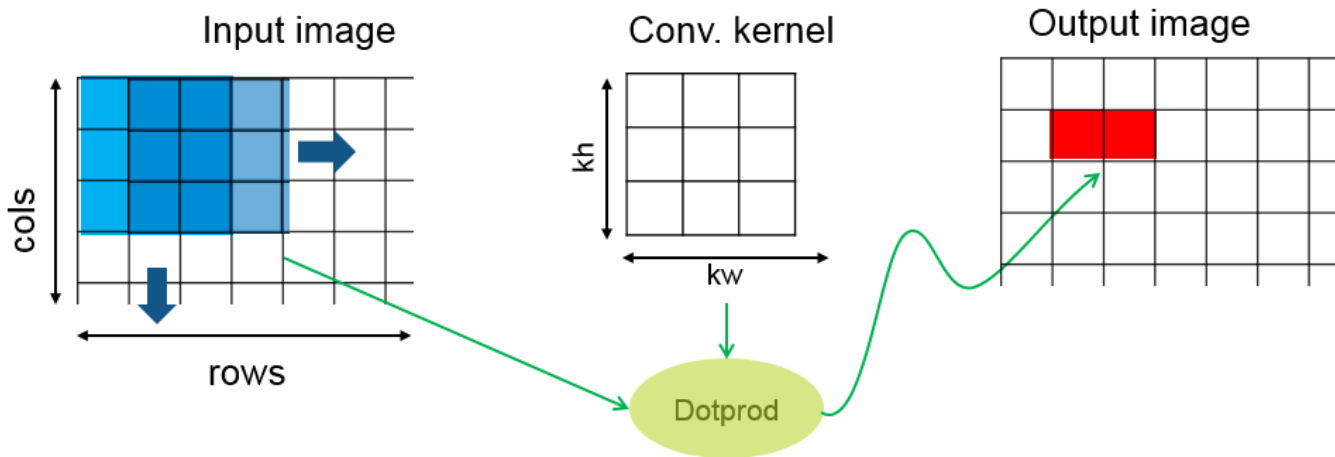
- “Call Custom CUDA Kernels from the Generated Code” on page 2-18
- “Configure Build for External C/C++ Code”
- “Mapping MATLAB Types to Types in Generated Code”
- “Kernels from Element-Wise Loops” on page 2-2

Design Patterns

GPU Coder supports some design patterns that map efficiently to GPU structures.

Stencil Processing

Stencil kernel operations compute each element of the output array as a function of a small region of the input array. You can express many filtering operations as a stencil operation. Examples include convolution, median filtering, and finite element methods.



In the GPU Coder implementation of the stencil kernel, each thread computes one element of the output array. Because a given input element is accessed repeatedly for computing multiple neighboring output elements, GPU Coder uses shared memory to improve memory bandwidth and data locality.

Use the `stencilfun` function and create CUDA code for stencil functions. For an example that demonstrates stencil preprocessing, see “Stencil Processing on GPU” on page 2-74.

Note Starting in R2022b, generate CUDA kernels for stencil like operations by using `stencilfun` function. `gpcoder.stencilKernel` is not recommended.

For very large input sizes, the `stencilfun` function may produce CUDA code that does not numerically match the MATLAB simulation. In such cases, consider reducing the size of the input to produce accurate results.

Matrix-Matrix Processing

Many scientific applications contain matrix-matrix operations including the General Matrix to Matrix Multiplication (GEMM), of the form $C = AB$ where you can optionally transpose A and B. The code for such matrix-matrix operations typically takes the pattern:

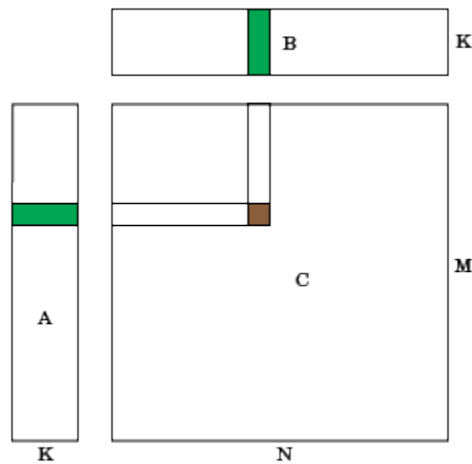
```
for x = 1:M
    for y = 1:N
```

```

    for z = 1:K
        C(x,y) = F(A(x,z),B(z,y));
    end
end
end

```

where $F()$ is a user-defined function. In these operations, a row from one input matrix and a column from the second input matrix is used to compute the corresponding element of the output matrix. Every thread reloads the row and column. This design pattern allows optimization of this structure by reusing data and making each thread compute multiple output elements.



For example, $F()$ can be a regular matrix multiply, $F()=@mtimes$. For such patterns, GPU Coder provides the `MatrixMatrix` kernel to create a highly efficient, fast implementation of matrix-matrix operations on the GPU.

Use the `gpcoder.matrixMatrixKernel` function and create CUDA code for performing matrix-matrix type operations.

See Also

`coder.gpu.kernel` | `coder.gpu.kernelfun` | `gpcoder.matrixMatrixKernel` | `coder.gpu.constantMemory` | `gpcoder.stencilKernel`

Related Examples

- “Stencil Processing on GPU” on page 2-74

More About

- “Kernels from Element-Wise Loops” on page 2-2
- “Kernels from Scatter-Gather Type Operations” on page 2-4
- “Kernels from Library Calls” on page 2-8
- “Call Custom CUDA Device Function from the Generated Code” on page 2-22

GPU Memory Allocation and Minimization

Discrete and Managed Modes

GPU Coder provides you access to two different memory allocation (`malloc`) modes available in the CUDA programming model, `cudaMalloc` and `cudaMallocManaged`. `cudaMalloc` API is applicable to the traditionally separate CPU, and GPU global memories. `cudaMallocManaged` is applicable to Unified Memory.

From a programmer point of view, a traditional computer architecture requires that data be allocated and shared between the CPU and GPU memory spaces. The need for applications to manage data transfers between these two memory spaces adds to increased complexity. Unified memory creates a pool of managed memory, shared between the CPU and the GPU. The managed memory is accessible to both the CPU and the GPU through a single pointer. Unified memory attempts to optimize memory performance by migrating data to the device that needs it, at the same time hiding the migration details from the program. Though unified memory simplifies the programming model, it requires device-sync calls when data written on the GPU is being accessed on the CPU. GPU Coder inserts these synchronization calls. According to NVIDIA, unified memory can provide significant performance benefits when by using CUDA 8.0, or when targeting embedded hardware like the NVIDIA Tegra®.

To change the memory allocation mode in the GPU Coder app, use the **Malloc Mode** drop-down box under **More Settings->GPU Coder**. When using the command-line interface, use the `MallocMode` build configuration property and set it to either `'discrete'` or `'unified'`.

Note In a future release, the unified memory allocation (`cudaMallocManaged`) mode will be removed when targeting NVIDIA GPU devices on the host development computer. You can continue to use unified memory allocation mode when targeting NVIDIA embedded platforms.

GPU Memory Manager

You can use the GPU memory manager for efficient memory allocation, management, and improving run-time performance. The GPU memory manager creates a collection of large GPU memory pools and manages allocation and deallocation of chunks of memory blocks within these pools. By creating large memory pools, the memory manager reduces the number of calls to the CUDA memory APIs, improving run-time performance. You can use the GPU memory manager for MEX and standalone CUDA code generation.

To enable the GPU memory manager, use one of these methods:

- In a GPU code configuration object (`coder.gpuConfig`), enable the `MemoryManager` property.
- In the GPU Coder app, on the **GPU Code** tab, select **GPU Memory Manager**.
- In the Simulink® Configuration Parameters dialog box, **Code Generation > GPU Code** pane, select the **Memory manager** parameter.

For CUDA code that uses NVIDIA CUDA libraries, such as `cuFFT`, `cuBLAS`, and `cuSOLVER`, you can enable the use of GPU memory manager for efficient memory allocation and management.

To use memory pools with CUDA libraries, enable the memory manager using one the methods above and:

- In the GPU code configuration object (`coder.gpuConfig`), enable the `EnableCUFFT`, `EnableCUBLAS`, or `EnableCUSOLVER` properties.
- In the GPU Coder app, on the **GPU Code** tab, select **Enable cuFFT**, **Enable cuBLAS**, or **Enable cuSOLVER**.
- In the Simulink Configuration Parameters dialog box, **Code Generation > GPU Code** pane, select the **cuFFT**, **cuBLAS**, or **cuSOLVER** parameters.

Customization options for GPU memory pools

The GPU memory manager provides additional code configuration parameters listed in the table to manage allocation and deallocation of memory blocks within GPU memory pools.

Code Configuration Parameter	Description	Value
<p>In a GPU code configuration object (<code>coder.gpuConfig</code>): <code>BlockAlignment</code></p> <p>In the GPU Coder app: on the GPU Code tab, Block Alignment</p>	<p>Controls the alignment of the blocks. The block sizes (bytes) in the pool are a multiple of the specified value.</p>	<p>Positive integer that is a power of 2. Default value is 256.</p>
<p>In a GPU code configuration object: <code>FreeMode</code></p> <p>In the GPU Coder app: on the GPU Code tab, Free Mode</p>	<p>Controls when the memory manager frees the GPU device memory.</p> <p>When set to 'Never', the memory is freed only when the memory manager is destroyed.</p> <p>Use 'AtTerminate' to free empty GPU pools when the <code>terminate</code> function is called in the generated code. For MEX targets, memory is freed after every call to the generated MEX function. For other targets, memory is freed when calling the <code>terminate</code> function.</p> <p>When set to 'AfterAllocate', empty pools are freed after each call to <code>CUDA</code> memory <code>allocate</code>.</p>	<p>'Never' (default) 'AtTerminate' 'AfterAllocate'</p>
<p>In a GPU code configuration object: <code>MinPoolSize</code></p> <p>In the GPU Coder app: on the GPU Code tab, Minimum Pool Size</p>	<p>Specify the minimum pool size in megabytes (MB).</p>	<p>Positive integer that is a power of 2. Default value is 8.</p>

Code Configuration Parameter	Description	Value
In a GPU code configuration object: <code>MaxPoolSize</code>	Specify the maximum pool size in megabytes (MB).	Positive integer that is a power of 2. Default value is 2048.
In the GPU Coder app: on the GPU Code tab, Maximum Pool Size	The memory manager computes the size levels using the <code>MinPoolSize</code> and <code>MaxPoolSize</code> parameters by interpolating between the two values in increasing powers of 2. For example, if the <code>MinPoolSize</code> is 4 and the <code>MaxPoolSize</code> is 1024, the size levels are {4, 8, 16, 32, 64, 128, 256, 512, 1024}.	

Memory Minimization

GPU Coder analyzes the data dependency between CPU and GPU partitions and performs optimizations to minimize the number of `cudaMemcpy` function calls in the generated code. The analysis also determines the minimum set of locations where data must be copied between CPU and GPU by using `cudaMemcpy`.

For example, the function `foo` has sections of code that process data sequentially on the CPU and in parallel on the GPU.

```
function [out] = foo(input1,input2)
    ...
    % CPU work
        input1 = ...
        input2 = ...
        tmp1 = ...
        tmp2 = ...
    ...
    % GPU work
        kernel1(gpuInput1, gpuTmp1);
        kernel2(gpuInput2, gpuTmp1, gpuTmp2);
        kernel3(gpuTmp1, gpuTmp2, gpuOut);
    ...
    % CPU work
        ... = out
end
```

An unoptimized CUDA implementation can potentially have multiple `cudaMemcpy` function calls to transfer all inputs `gpuInput1`, `gpuInput2`, and the temporary results `gpuTmp1`, `gpuTmp2` between kernel calls. Because the intermediate results `gpuTmp1`, `gpuTmp2` are not used outside the GPU, they can be stored within the GPU memory resulting in fewer `cudaMemcpy` function calls. These optimizations improve overall performance of the generated code. The optimized implementation is:

```
gpuInput1 = input1;
gpuInput2 = input2;
```



```
kernel1<<< >>>(gpuInput1, gpuTmp1);
kernel2<<< >>>(gpuInput2, gpuTmp1, gpuTmp2);
kernel3<<< >>>(gpuTmp1, gpuTmp2, gpuOut);
```

```
out = gpuOut;
```

To eliminate redundant `cudaMemcpy` calls, GPU Coder analyzes all uses and definitions of a given variable and uses status flags to perform minimization. An example of the original code and what the generated code looks like is shown in this table.

Original Code	Optimized Generated Code
<pre>A(:) = for i = 1:N gB = kernel1(gA); gA = kernel2(gB); if (somecondition) gC = kernel3(gA, gB); end ... end ... = C;</pre>	<pre>A(:) = ... A_isDirtyOnCpu = true; ... for i = 1:N if (A_isDirtyOnCpu) gA = A; A_isDirtyOnCpu = false; end gB = kernel1(gA); gA = kernel2(gB); if (somecondition) gC = kernel3(gA, gB); C_isDirtyOnGpu = true; end ... end ... if (C_isDirtyOnGpu) C = gC; C_isDirtyOnGpu = false; end ... = C;</pre>

The `_isDirtyOnCpu` flag tells the GPU Coder memory optimization about routines where the given variable is declared and used either on the CPU or on then GPU.

See Also

Apps

GPU Coder

Functions

`codegen` | `coder.gpu.constantMemory` | `gpucoder.reduce`

Objects

`coder.gpuConfig` | `coder.CodeConfig` | `coder.EmbeddedCodeConfig` | `coder.MexCodeConfig`

More About

- “GPU Programming Paradigm”
- “Code Generation by Using the GPU Coder App”

- “Code Generation Using the Command Line Interface”

Use Dynamically Allocated C++ Arrays in Generated Function Interfaces

In most cases, when you generate code for a MATLAB function that accepts or returns an array, there is an array at the interface of the generated CUDA function. For an array size that is unknown at compile time, or whose bound exceeds a predefined threshold, the memory for the generated array is dynamically allocated.

By default, the dynamically allocated array is implemented by using the C style `emxArray` data structure in the generated code. Alternatively, dynamically allocated array can be implemented as a class template called `coder::gpu_array` in the generated code. `coder::gpu_array` offers several advantages over `emxArray` style data structures:

- The generated code is exception safe
- Generated code is easier to read.
- Better C++ integration because of ease of initializing the input data and working with the output data.
- Because `coder::gpu_array` is defined in a header file that ships with MATLAB, you can write the interface code before the generating code.

To use dynamically allocated arrays in your custom CUDA code that you integrate with the generated CUDA C++ functions, learn to use the `coder::gpu_array` template.

Change Interface Generation

By default, the generated CUDA code uses the C style `emxArray` data structure to implement dynamically allocated arrays. Instead, you can choose to generate CUDA code that uses the `coder::gpu_array` template to implement dynamically allocated arrays. To generate the `coder::gpu_array` template, do one of the following:

- In a code configuration object (`coder.MexCodeConfig`, `coder.CodeConfig`, or `coder.EmbeddedCodeConfig`), set the `DynamicMemoryAllocationInterface` parameter to 'C++'.
- In the GPU Coder app, on the **Memory** tab, set **Dynamic memory allocation interface** to Use C++ `coder::array`.

Using the `coder::gpu_array` Class Template

When you generate CUDA code for your MATLAB functions, the code generator produces header files `coder_gpu_array.h` and `coder::array.h` in the build folder. The `coder_gpu_array.h` header file contains the definition of the class template `gpu_array` in the namespace `coder` and the definitions for the function templates `arrayCopyCpuToGpu` and `arrayCopyGpuToCpu`. The `coder::gpu_array` template implements the dynamically allocated arrays in the generated code. The declaration for this template is:

```
template <typename T, int32_T N> class gpu_array
```

The array contains elements of type `T` and has `N` dimensions. For example, to declare a two-dimensional dynamic array `myArray` that contains elements of type `int32_T` in your custom CUDA code, use:

```
coder::gpu_array<int32_T, 2> myArray
```

The function templates `arrayCopyCpuToGpu` and `arrayCopyGpuToCpu` implement data transfers between the CPU and GPU memory. On the CPU, the dynamically allocated arrays are implemented by using the `coder::array` template. For more information on the APIs you use to create and interact with dynamic arrays in your custom code, see “Use Dynamically Allocated C++ Arrays in Generated Function Interfaces”.

To use dynamically allocated arrays in your custom CUDA code that you want to integrate with the generated code (for example, a custom main function), include the `coder_gpu_array.h` and `coder_array.h` header files in your custom `.cu` files.

Generate C++ Code That Accepts and Returns a Variable-Size Numeric Array

This examples shows how to customize the generated example main function to use the `coder::gpu_array` and `coder::array` class templates in your project.

Your goal is to generate a CUDA executable for `xTest1` that can accept and return an array of `int32_T` elements. You want the first dimension of the array to be singleton and the second dimension to be unbounded.

- 1 Define a MATLAB function `xTest1` that accepts an array `X`, adds the scalar `A` to each of its elements, and returns the resulting array `Y`.

```
function Y = xTest1(X, A)
Y = X;
for i = 1:numel(X)
    Y(i) = X(i) + A;
end
```

- 2 Generate initial source code for `xTest1` and move `xTest1.h` from the code generation folder to your current folder. Use the following commands:

```
cfg = coder.gpuConfig('lib');
cfg.DynamicMemoryAllocationInterface = 'C++';
cfg.GenerateReport = true;
inputs = {coder.typeof(int32(0), [1 inf]), int32(0)};

codegen -config cfg -args inputs xTest1.m
```

The function prototype for `xTest1` in the generated code is shown here:

```
extern void xTest1(const coder::array<int, 2U> &X, int A,
                  coder::array<int, 2U> &Y);
```

Interface the generated code by providing input and output arrays that are compatible with the function prototype shown above.

- 3 Define a CUDA main function in the file `xTest1_main.cu` in your current working folder.

This main function includes the header files `coder_gpu_array.h` and `coder_array.h` that contain the `coder::gpu_array` and `coder::array` class template definitions respectively. The main function performs these actions:

- Declare `myArray` and `myResult` as two-dimensional `coder::array` dynamic arrays of `int32_T` elements.
- Dynamically set the sizes of the two dimensions of `myArray` to 1 and 100 by using the `set_size` method.

- Access the size vector of `myResult` by using `myResult.size`.

```
#include<iostream>
#include<coder_array.h>
#include<xTest1.h>

int main(int argc, char *argv[])
{
    static_cast<void>(argc);
    static_cast<void>(argv);

    // Instantiate the input variable by using coder::array template
    coder::array<int32_T, 2> myArray;

    // Allocate initial memory for the array
    myArray.set_size(1, 100);

    // Access array with standard C++ indexing
    for (int i = 0; i < myArray.size(1); i++) {
        myArray[i] = i;
    }

    // Instantiate the result variable by using coder::array template
    coder::array<int32_T, 2> myResult;

    // Pass the input and result arrays to the generated function
    xTest1(myArray, 1000, myResult);

    for (int i = 0; i < myResult.size(1); i++) {
        if (i > 0) std::cout << " ";
        std::cout << myResult[i];
        if ((i+1) % 10) == 0 std::cout << std::endl;
    }
    std::cout << std::endl;

    return 0;
}
```

4 Generate code by running this script:

```
cfg = coder.gpuConfig('exe');
cfg.DynamicMemoryAllocationInterface = 'C++';
cfg.GenerateReport = true;
cfg.CustomSource = 'xTest1_main.cu';
cfg.CustomInclude = '.';
codegen -config cfg -args inputs xTest1_main.cu xTest1.m
```

5 The code generator produces an executable file `xTest1` in your current working folder. Run the executable using the following commands:

```
if ispc
    !xtest1.exe
else
    !./xTest1
end
```

```
1000 1001 1002 1003 1004 1005 1006 1007 1008 1009
1010 1011 1012 1013 1014 1015 1016 1017 1018 1019
1020 1021 1022 1023 1024 1025 1026 1027 1028 1029
1030 1031 1032 1033 1034 1035 1036 1037 1038 1039
1040 1041 1042 1043 1044 1045 1046 1047 1048 1049
1050 1051 1052 1053 1054 1055 1056 1057 1058 1059
1060 1061 1062 1063 1064 1065 1066 1067 1068 1069
1070 1071 1072 1073 1074 1075 1076 1077 1078 1079
1080 1081 1082 1083 1084 1085 1086 1087 1088 1089
1090 1091 1092 1093 1094 1095 1096 1097 1098 1099
```

Limitations

- For generating CUDA code that uses `coder::gpu_array`, the GPU memory allocation mode must be set to `discrete`.

To change the memory allocation mode in the GPU Coder app, use the **Malloc Mode** drop-down box under **More Settings->GPU Coder**. When using the command-line interface, use the `MallocMode` build configuration property and set it to either 'discrete' or 'unified'.

- GPU Coder does not support `coder::gpu_array` in Simulink.

See Also

`coder.typeof` | `coder.varsizes`

More About

- “Use C Arrays in the Generated Function Interfaces”
- “Representation of Arrays in Generated Code”

Support for GPU Arrays

You can use GPU arrays as input and output arguments to an entry-point function when generating CUDA MEX, source code, static libraries, dynamic libraries, and executables. GPU Coder automatically takes care of CPU-GPU copies based on the input type and the usage of the variable in your MATLAB design. The GPU array functionality is useful in minimizing CPU-GPU copies when you are trying to:

- Integrate the generated code with an existing implementation that has its outputs on the GPU memory.
- Pass MATLAB `gpuArray` inputs to the generated MEX function.

To mark an input to the entry-point function as a GPU type, use one of the following approaches:

- Use `coder.typeof` to represent the `gpuArray` type of an entry-point function input. For example:

```
coder.typeof(rand(20), 'Gpu', true);
```

- Use the `gpuArray` function. For example:

```
in = gpuArray(rand(1,10));
codegen -config cfg -args {in} test
```

Considerations

- GPU Coder supports all data type supported by `gpuArray`.
- For using variable dimension arrays, only the bounded types are supported.
- For 'lib', 'dll', and 'exe' targets, you must pass the correct pointers to the entry-point function in the example main function. For example, if an input is marked as 'Gpu', a GPU pointer should be passed when the entry-point is called from the main function.
- The `MemoryMode` (memory allocation mode) property of the code configuration object must be set to 'discrete'. For example,

```
cfg.GpuConfig.MallocMode = 'discrete';
```

- During code generation, if one input to entry-point function is of the GPU array, then GPU Coder attempts to make all the output variables GPU array types, provided they are supported by `gpuArray`. For example, if the entry-point function returns a `struct` and because `struct` is not supported by `gpuArray`, the generated code returns a CPU output. However, if a supported matrix type is returned, then the generated code returns a GPU output.

Note Passing `gpuArray` inputs does not guarantee the outputs to also be `gpuArray`.

Limitations

- GPU Coder does not support the following types:
 - `char`
 - `half`
 - Scalar GPU arrays

- Structures
- Cell arrays
- Classes
- Enumerated types
- Fixed-point data types
- GPU Coder does not support the 'unified' memory mode for GPU arrays.

See Also

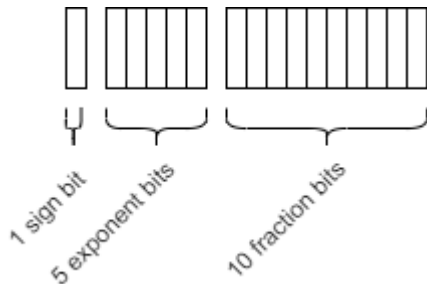
`coder.gpu.kernel` | `coder.gpu.kernelfun` | `gpucoder.matrixMatrixKernel` |
`coder.gpu.constantMemory` | `gpucoder.stencilKernel`

Related Examples

- “Kernels from Element-Wise Loops” on page 2-2
- “Kernels from Scatter-Gather Type Operations” on page 2-4
- “Kernels from Library Calls” on page 2-8
- “Design Patterns” on page 2-26

What is Half Precision?

The IEEE® 754 half-precision floating-point format is a 16-bit word divided into a 1-bit sign indicator s , a 5-bit biased exponent e , and a 10-bit fraction f .



Because numbers of type `half` are stored using 16 bits, they require less memory than numbers of type `single`, which uses 32 bits, or `double`, which uses 64 bits. However, because they are stored with fewer bits, numbers of type `half` are represented to less precision than numbers of type `single` or `double`.

The range, bias, and precision for supported floating-point data types are given in the table below.

Data Type	Low Limit	High Limit	Exponent Bias	Precision
Half	$2^{-14} \approx 6.1 \cdot 10^{-5}$	$(2-2^{-10}) \cdot 2^{15} \approx 6.5 \cdot 10^4$	15	$2^{-10} \approx 10^{-3}$
Single	$2^{-126} \approx 10^{-38}$	$2^{128} \approx 3 \cdot 10^{38}$	127	$2^{-23} \approx 10^{-7}$
Double	$2^{-1022} \approx 2 \cdot 10^{-308}$	$2^{1024} \approx 2 \cdot 10^{308}$	1023	$2^{-52} \approx 10^{-16}$

For a video introduction to the half-precision data type, see [What Is Half Precision?](#) and [Half-Precision Math in Modeling and Code Generation](#).

Half Precision Applications

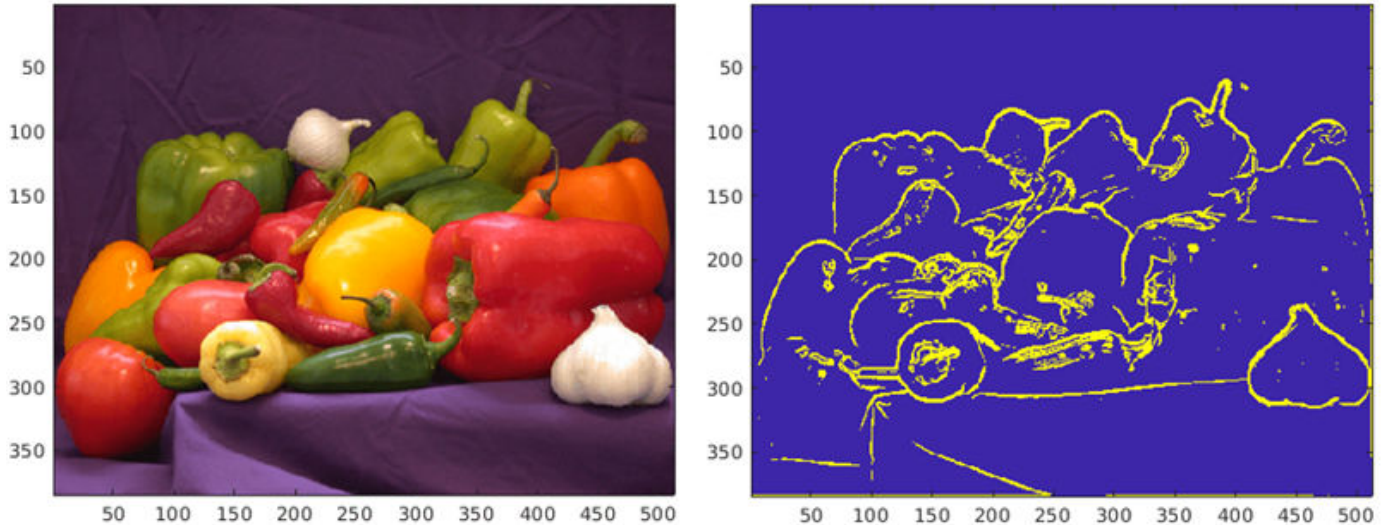
When an algorithm contains large or unknown dynamic ranges (for example integrators in feedback loops) or when the algorithm uses operations that are difficult to design in fixed-point (for example `atan2`), it can be advantageous to use floating-point representations. The half-precision data type occupies only 16 bits of memory, but its floating-point representation enables it to handle wider dynamic ranges than integer or fixed-point data types of the same size. This makes half precision particularly suitable for some image processing and graphics applications. When half-precision is used with deep neural networks, the time needed for training and inference can be reduced. By using half precision as a storage time for lookup tables, the memory footprint of the lookup table can be reduced.

MATLAB Examples

- “Fog Rectification” on page 2-80 — The fog rectification image processing algorithm uses convolution, image color space conversion, and histogram-based contrast stretching to enhance the input image. This example shows how to generate and execute CUDA MEX with half-precision data types for these image processing operations.



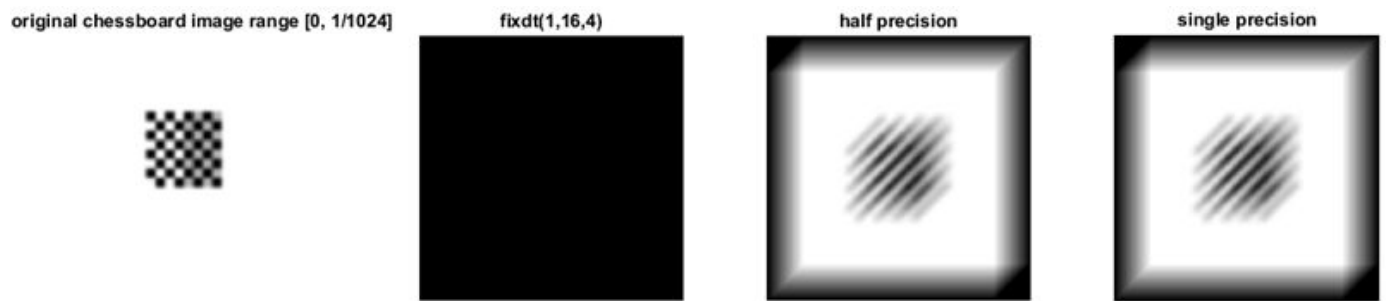
- “Edge Detection with Sobel Method in Half-Precision” on page 2-103 — The sobel edge detection algorithm takes an input image and returns an output image that emphasizes high spatial frequency regions that correspond to edges in the input image. This example shows how to generate and execute CUDA MEX with the half-precision data type used for the input image and Sobel operator kernel values.



- “Generate Code for Sobel Edge Detection That Uses Half-Precision Data Type” — This example shows how to generate a standalone C++ library from a MATLAB function that performs Sobel edge detection of images by using half-precision floating point numbers.

Simulink Examples

- “Half-Precision Field-Oriented Control Algorithm” (Fixed-Point Designer) — This example implements a Field-Oriented Control (FOC) algorithm using both single precision and half precision.
- “Image Quantization with Half-Precision Data Types” (Fixed-Point Designer) — This example shows the effects of quantization on images. While the fixed-point data type does not always produce an acceptable results, the half-precision data type, which uses the same number of bits as the fixed-point data type, produces a result comparable to the single-precision result.



- “Digit Classification with Half-Precision Data Types” (Fixed-Point Designer) — This example compares the results of a trained neural network classification model in double precision and half precision.
- “Convert Single Precision Lookup Table to Half Precision” (Fixed-Point Designer) — This example demonstrates how to convert a single-precision lookup table to use half precision. Half precision is the storage type; the lookup table computations are performed using single precision. After converting to half precision, the memory size of the Lookup Table blocks are reduced by half while maintaining the desired system performance.

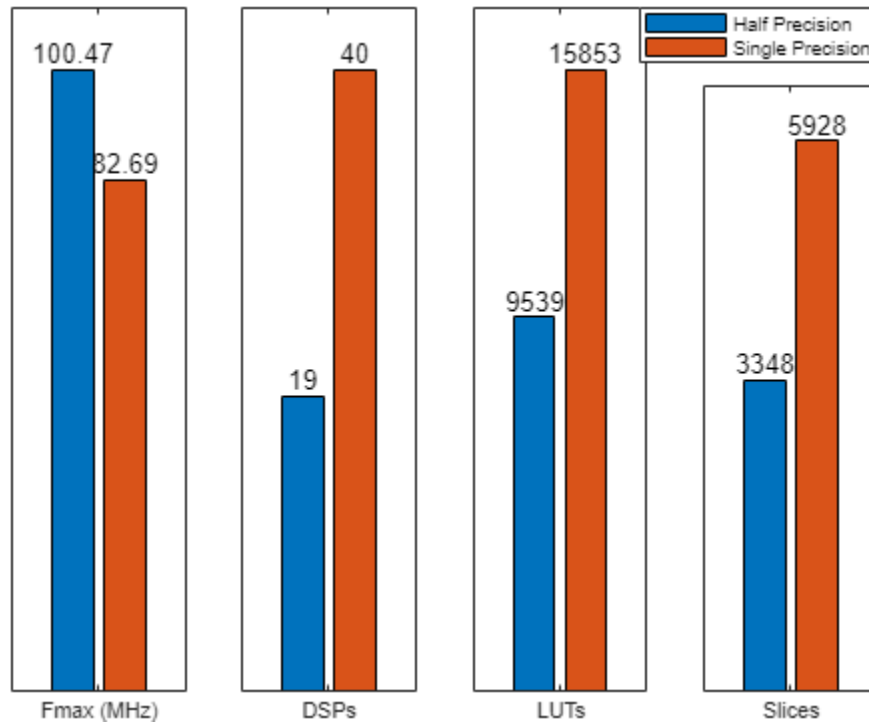
Benefits of Using Half Precision in Embedded Applications

The half precision data type uses less memory than other floating-point types like single and double. Though it occupies only 16 bits of memory, its floating-point representations enables it to handle wider dynamic ranges than integer or fixed-point data types of the same size.

FPGA

The half precision data type uses significantly less area and has low latency compared to the single precision data type when used on hardware. Half precision is particularly advantageous for low dynamic range applications.

The following plot shows the advantage of using half precision for an implementation of a field-oriented control algorithm in Xilinx® Virtex® 7 hardware.



GPU

In GPUs that support the half-precision data type, arithmetic operations are faster as compared to single or double precision.

In applications like deep learning, which require a large number of computations, using half precision can provide significant performance benefits without significant loss of precision. With GPU Coder, you can generate optimized code for prediction of a variety of trained deep learning networks from the Deep Learning Toolbox™. You can configure the code generator to take advantage of the NVIDIA TensorRT high performance inference library for NVIDIA GPUs. TensorRT provides improved latency, throughput, and memory efficiency by combining network layers and optimizing kernel selection. You can also configure the code generator to take advantage TensorRT's precision modes (FP32, FP16, or INT8) to further improve performance and reduce memory requirements.

CPU

In CPUs that support the half-precision data type, arithmetic operations are faster as compared to single or double precision. For ARM® targets that natively support half-precision data types, you can generate native half C code from MATLAB or Simulink. See “Code Generation with Half Precision” (Fixed-Point Designer).

Half Precision in MATLAB

Many functions in MATLAB support the half-precision data type. For a full list of supported functions, see `half`.

Half Precision in Simulink

Signals and block outputs in Simulink can specify a half-precision data type. The half-precision data type is supported for simulation and code generation for parameters and a subset of blocks. To view the blocks that support half precision, at the command line, type:

```
showblockdatatypetable
```

Blocks that support half precision display an X in the column labeled **Half**. For detailed information about half precision support in Simulink, see “The Half-Precision Data Type in Simulink” (Fixed-Point Designer).

Code Generation with Half Precision

The half precision data type is supported for C/C++ code generation, CUDA code generation using GPU Coder, and HDL code generation using HDL Coder™. For GPU targets, the half-precision data type uses the native half data type available in NVIDIA GPU for maximum performance.

For detailed code generation support for half precision in MATLAB and Simulink, see “Half Precision Code Generation Support” (Fixed-Point Designer) and “The Half-Precision Data Type in Simulink” (Fixed-Point Designer).

For embedded hardware targets that natively support special types for half precision, such as `_Float16` and `_fp16` data types for ARM compilers, you can generate native half precision C code using Embedded Coder® or MATLAB Coder. For more information, see “Generate Native Half-Precision C Code from Simulink Models” (Fixed-Point Designer) and “Generate Native Half-Precision C Code Using MATLAB Coder” (Fixed-Point Designer).

See Also

`half` | “The Half-Precision Data Type in Simulink” (Fixed-Point Designer) | “Half Precision” 16-bit Floating Point Arithmetic | “Floating-Point Numbers” (Fixed-Point Designer)

Related Examples

- “Fog Rectification” on page 2-80
- “Edge Detection with Sobel Method in Half-Precision” on page 2-103
- “Generate Code for Sobel Edge Detection That Uses Half-Precision Data Type”
- “Half-Precision Field-Oriented Control Algorithm” (Fixed-Point Designer)
- “Image Quantization with Half-Precision Data Types” (Fixed-Point Designer)
- “Digit Classification with Half-Precision Data Types” (Fixed-Point Designer)
- “Convert Single Precision Lookup Table to Half Precision” (Fixed-Point Designer)

Half Precision Code Generation Support

To assign a half-precision data type to a number or variable, use the `half` constructor. A half-precision data type occupies 16 bits of memory, but its floating-point representation enables it to handle wider dynamic ranges than integer or fixed-point data types of the same size. For more information, see “Floating-Point Numbers” (Fixed-Point Designer).

A subset of MATLAB functions are supported for use with half-precision inputs. Additionally, some functions support code generation with half-precision data types. C and C++ code generation requires MATLAB Coder. CUDA code generation for NVIDIA GPUs requires GPU Coder. Supported functions appear in alphabetical order in the following table. MATLAB System object™ supports half-precision data type and MATLAB System (Simulink) block supports half-precision data type with real values. For general information regarding code generation with half precision, see `half`.

Function	MATLAB Simulation Support	C/C++ Code Generation Support	GPU Code Generation Support
<code>abs</code>	✓	✓	✓
<code>acos</code>	✓	✓	✓
<code>acosh</code>	✓	✓	✓
<code>activations</code>	✓	✓ Half inputs are cast to single precision and computations are performed in single precision.	✓ Half inputs are cast to single precision and computations are performed in single precision. To perform computations in half, set the library target to 'tensorrt' and set the data type to 'FP16' in <code>coder.DeepLearningConfig</code> .
<code>all</code>	✓	✓	✓
<code>allfinite</code>	✓	✓	✓
<code>and, &</code>	✓	✓	✓
Short-Circuit AND	✓	✓	✓
<code>any</code>	✓	✓	✓
<code>anynan</code>	✓	✓	✓
<code>area</code>	✓		
<code>asin</code>	✓	✓	✓
<code>asinh</code>	✓	✓	✓
<code>atan</code>	✓	✓	✓
<code>atan2</code>	✓	✓	✓
<code>atanh</code>	✓	✓	✓

Function	MATLAB Simulation Support	C/C++ Code Generation Support	GPU Code Generation Support
bar	✓		
barh	✓		
cast	✓ Supported syntax: cast(__, 'half') cast(__, 'like', p)	✓ Supported syntax: cast(__, 'half') cast(__, 'like', p)	✓ Supported syntax: cast(__, 'half') cast(__, 'like', p)
cat	✓	✓ <ul style="list-style-type: none"> • Dimension argument must be a constant. • Dimension argument cannot be half precision. 	✓ <ul style="list-style-type: none"> • Dimension argument must be a constant. • Dimension argument cannot be half precision.
ceil	✓	✓	✓
cell	✓	✓	✓
chol	✓		
circshift	✓	✓	✓
classify	✓	✓ Half inputs are cast to single precision and computations are performed in single precision.	✓ Half inputs are cast to single precision and computations are performed in single precision. To perform computations in half, set the library target to 'tensorrt' and set the data type to 'FP16' in coder.DeepLearningConfig.
coder.ceval		✓	✓
colon, :	✓	✓	✓
complex	✓	✓	
conj	✓	✓	✓
conv	✓	✓	✓
conv2	✓	✓	✓
cos	✓	✓	✓
cosh	✓	✓	✓
cospi	✓	✓	✓

Function	MATLAB Simulation Support	C/C++ Code Generation Support	GPU Code Generation Support
ctranspose	✓	✓	✓
cumsum	✓		
dot	✓		
double	✓	✓	✓
empty	✓		
eps	✓ Supported syntax: eps('half') eps(half(1)) eps('like',half(1))	✓ eps(half(1))	✓ eps(half(1))
eq, ==	✓	✓	✓
exp	✓	✓	✓
expm1	✓	✓	✓
eye	✓ Supported syntax: eye(_, 'half') eye(_, 'like', p)	✓ Supported syntax: eye(_, 'half') eye(_, 'like', p) where p is half precision. Other input arguments cannot be half precision.	✓ Supported syntax: eye(_, 'half') eye(_, 'like', p) where p is half precision. Other input arguments cannot be half precision.
fft	✓	✓	
fft2	✓	✓	
fftn	✓	✓	
fftshift	✓	✓	✓
fix	✓	✓	✓
flintmax	✓ Supported syntax: flintmax('half') flintmax('like',half(1))		

Function	MATLAB Simulation Support	C/C++ Code Generation Support	GPU Code Generation Support
flip	✓	✓ Dimension argument cannot be half precision.	✓ Dimension argument cannot be half precision.
flipplr	✓	✓	✓
flipud	✓	✓	✓
floor	✓	✓	✓
fma	✓ Complex half-precision inputs are not supported.	✓ Complex half-precision inputs are not supported.	✓ Complex half-precision inputs are not supported.
fplot	✓		
ge, >=	✓	✓	✓
gt, >	✓	✓	✓
half	✓	✓	✓
horzcat	✓	✓	✓
hypot	✓	✓	✓
ifft	✓	✓	
ifft2	✓	✓	
ifftn	✓	✓	
ifftshift	✓	✓	✓
imag	✓	✓	
Inf	✓ Supported syntax: Inf(__, 'half') Inf(__, 'like', p)	✓ Supported syntax: Inf(__, 'half') Inf(__, 'like', p)	✓ Supported syntax: Inf(__, 'half') Inf(__, 'like', p)
int16	✓	✓	✓
int32	✓	✓	✓
int64	✓	✓	✓
int8	✓	✓	✓
isa	✓	✓	✓
iscolumn	✓	✓	✓
isempty	✓	✓	✓
isequal	✓	✓	✓
isequaln	✓	✓	✓

Function	MATLAB Simulation Support	C/C++ Code Generation Support	GPU Code Generation Support
isfinite	✓	✓	✓
isfloat	✓	✓	✓
isinf	✓	✓	✓
isinteger	✓	✓	✓
islogical	✓	✓	✓
ismatrix	✓	✓	✓
isnan	✓	✓	✓
isnumeric	✓	✓	✓
isobject	✓	✓	✓
	Returns true with half-precision input.	Returns false with half-precision input.	Returns false with half-precision input.
isreal	✓	✓	✓
isrow	✓	✓	✓
isscalar	✓	✓	✓
issorted	✓		
isvector	✓	✓	✓
ldivide	✓	✓	✓
le, <=	✓	✓	✓
length	✓	✓	✓
line	✓		
log	✓	✓	✓
log10	✓	✓	✓
log1p	✓	✓	✓
log2	✓	✓	✓
		Two output syntax is not supported.	Two output syntax is not supported.
logical	✓	✓	✓
lt, <	✓	✓	✓
lu	✓		
max	✓	✓	✓
mean	✓	✓	✓
min	✓	✓	✓
minus, -	✓	✓	✓
mldivide, \	✓		
	Left-hand side must be scalar		

Function	MATLAB Simulation Support	C/C++ Code Generation Support	GPU Code Generation Support
mod	✓	✓	✓
mrdivide, /	✓ Right-hand side must be scalar	✓ Right-hand side must be scalar	✓ Right-hand side must be scalar
mtimes, *	✓	✓	✓ For GPU Code generation, you can perform half-precision matrix multiplication with real inputs.
NaN	✓ Supported syntax: NaN(__, 'half') NaN(__, 'like', p)	✓ Supported syntax: NaN(__, 'half') NaN(__, 'like', p)	✓ Supported syntax: NaN(__, 'half') NaN(__, 'like', p)
ndims	✓	✓	✓
ne, ~=	✓	✓	✓
not	✓	✓	✓
numel	✓	✓	✓
ones	✓ Supported syntax: ones(__, 'half') ones(__, 'like', p)	✓ Supported syntax: ones(__, 'half') ones(__, 'like', p)	✓ Supported syntax: ones(__, 'half') ones(__, 'like', p)
or,	✓	✓	✓
Short-Circuit OR	✓	✓	✓
permute	✓	✓	✓
plot	✓		
plot3	✓		
plotmatrix	✓		
plus, +	✓	✓	✓
pow10	✓	✓	✓
pow2	✓	✓	✓
power, .^	✓	✓	✓

Function	MATLAB Simulation Support	C/C++ Code Generation Support	GPU Code Generation Support
predict	✓	✓ Half inputs are cast to single precision and computations are performed in single precision.	✓ Half inputs are cast to single precision and computations are performed in single precision. To perform computations in half, set the library target to 'tensorrt' and set the data type to 'FP16' in <code>coder.DeepLearningConfig</code> .
predictAndUpdateState	✓	✓ Half inputs are cast to single precision and computations are performed in single precision.	✓ Half inputs are cast to single precision and computations are performed in single precision. To perform computations in half, set the library target to 'tensorrt' and set the data type to 'FP16' in <code>coder.DeepLearningConfig</code> .

Function	MATLAB Simulation Support	C/C++ Code Generation Support	GPU Code Generation Support
prod	<p>✓</p> <p>Half inputs are cast to single precision and computations are performed in single precision. As a result, saturation behavior differs between single and half inputs:</p> <pre>maxhalf = half.realmx; isequal(prod([maxhalf 2 0.5]), maxhalf) ans = logical 1 maxsingle = realmx('single'); isequal(prod([maxsingle 2 0.5]), maxsingle) ans = logical 0</pre>	✓	✓
rdivide	✓	✓	✓
real	✓	✓	✓
realmax	<p>✓</p> <p>Supported syntax:</p> <pre>realmax('half') realmax('like',half(1))</pre>		
realmin	<p>✓</p> <p>Supported syntax:</p> <pre>realmin('half') realmin('like',half(1))</pre>		
rem	✓	✓	✓
repelem	✓	✓	✓

Function	MATLAB Simulation Support	C/C++ Code Generation Support	GPU Code Generation Support
repmat	✓	✓ Dimension argument cannot be half precision.	✓ Dimension argument cannot be half precision.
reshape	✓	✓ Dimension argument cannot be half precision.	✓ Dimension argument cannot be half precision.
rgbplot	✓		
round	✓ Only one input supported	✓ Only one input supported	✓ Only one input supported
rsqrt	✓ Complex half-precision inputs are not supported		
scatter	✓		
scatter3	✓		
sign	✓	✓	✓
sin	✓	✓	✓
single	✓	✓	✓
sinh	✓	✓	✓
sinpi	✓	✓	✓
size	✓	✓	✓
sort	✓		
sqrt	✓	✓	✓
squeeze	✓	✓	✓
storedInteger	✓		

Function	MATLAB Simulation Support	C/C++ Code Generation Support	GPU Code Generation Support
sum	<p>✓</p> <p>Half inputs are cast to single precision and computations are performed in single precision. As a result, saturation behavior differs between single and half inputs:</p> <pre>maxhalfint = half.flintmax; isequal(sum([maxhalfint, 1, -1]), maxhalfint) ans = logical 1 maxsingleint = flintmax('single'); isequal(sum([maxsingleint, 1, -1]), maxsingleint) ans = logical 0</pre>	✓	✓
tan	✓	✓	✓
tanh	✓	✓	✓
times, .*	✓	✓	✓
transpose	✓	✓	✓
typecast	✓		
uint16	✓	✓	✓
uint32	✓	✓	✓
uint64	✓	✓	✓
uint8	✓	✓	✓
uminus	✓	✓	✓
uplus	✓	✓	✓
vertcat	✓	✓	✓
xlim	✓		
ylim	✓		

Function	MATLAB Simulation Support	C/C++ Code Generation Support	GPU Code Generation Support
zeros	✓ Supported syntax: zeros(__, 'half') zeros(__, 'like', p)	✓ Supported syntax: zeros(__, 'half') zeros(__, 'like', p)	✓ Supported syntax: zeros(__, 'half') zeros(__, 'like', p)
zlim	✓		

See Also

half

More About

- “Floating-Point Numbers” (Fixed-Point Designer)
- “What is Half Precision?” (Fixed-Point Designer)
- Edge Detection with Sobel Method in Half-Precision on page 2-103

Simulate Diffraction Patterns Using CUDA FFT Libraries

This example shows how to use GPU Coder™ to leverage the CUDA® Fast Fourier Transform library (cuFFT) to compute two-dimensional FFT on a NVIDIA® GPU. The two-dimensional Fourier transform is used in optics to calculate far-field diffraction patterns. When a monochromatic light source passes through a small aperture, such as in Young's double-slit experiment, you can observe these diffraction patterns. This example also shows you how to use GPU pointers as inputs to an entry-point function when generating CUDA MEX, source code, static libraries, dynamic libraries, and executables. By using this functionality, the performance of the generated code is improved by minimizing the number of `cudaMemcpy` calls in the generated code.

Third-Party Prerequisites

Required

This example generates CUDA MEX and has the following third-party requirements.

- CUDA enabled NVIDIA GPU and compatible driver.

Optional

For non-MEX builds such as static, dynamic libraries or executables, this example has the following additional requirements.

- NVIDIA toolkit.
- Environment variables for the compilers and libraries. For more information, see “Third-Party Hardware” and “Setting Up the Prerequisite Products”.

Verify GPU Environment

To verify that the compilers and libraries necessary for running this example are set up correctly, use the `coder.checkGpuInstall` function.

```
envCfg = coder.gpuEnvConfig('host');
envCfg.BasicCodegen = 1;
envCfg.Quiet = 1;
coder.checkGpuInstall(envCfg);
```

Define the Coordinate System

Before simulating the light that has passed through an aperture, you must define your coordinate system. To get the correct numeric behavior when you call `fft2`, you must carefully arrange x and y so that the zero value is in the correct place. $N2$ is half the size in each dimension.

```
N2 = 1024;
[gx, gy] = meshgrid(-1:1/N2:(N2-1)/N2);
```

Simulate the Diffraction Pattern for a Rectangular Aperture

Simulate the effect of passing a parallel beam of monochromatic light through a small rectangular aperture. The two-dimensional Fourier transform describes the light field at a large distance from the aperture. Form `aperture` as a logical mask based on the coordinate system. The light source is a double-precision version of the aperture. Find the far-field light signal by using the `fft2` function.

```
aperture      = (abs(gx) < 4/N2) .* (abs(gy) < 2/N2);  
lightsource   = double(aperture);  
farfieldsignal = fft2(lightsource);
```

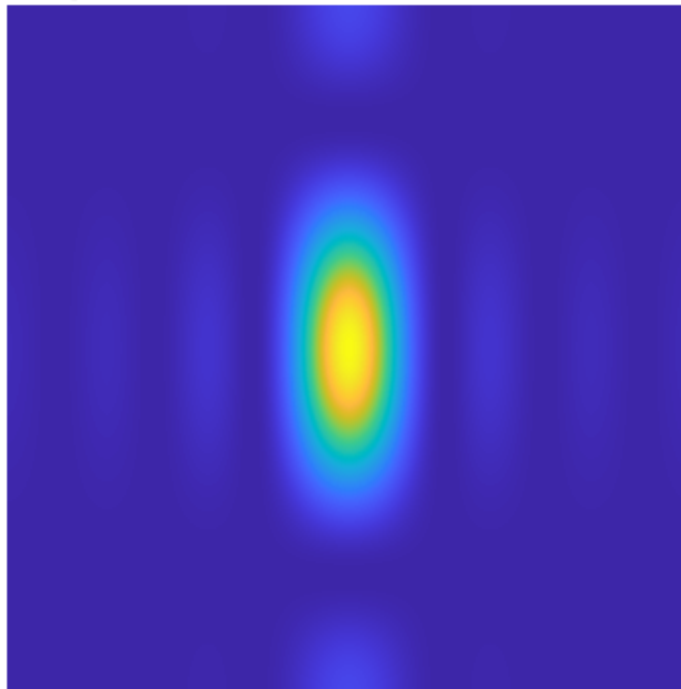
Display the Light Intensity for a Rectangular Aperture

The `visualize.m` function displays the light intensity for a rectangular aperture. Calculate the far-field light intensity from the magnitude squared of the light field. To aid visualization, use the `fftshift` function.

type `visualize`

```
function visualize(farfieldsignal, titleStr)  
  
farfieldintensity = real( farfieldsignal .* conj( farfieldsignal ) );  
imagesc( fftshift( farfieldintensity ) );  
axis( 'equal' ); axis( 'off' );  
title(titleStr);  
  
end  
  
str = sprintf('Rectangular Aperture Far-Field Diffraction Pattern in MATLAB');  
visualize(farfieldsignal,str);
```

Rectangular Aperture Far-Field Diffraction Pattern in MATLAB



Generate CUDA MEX for the Function

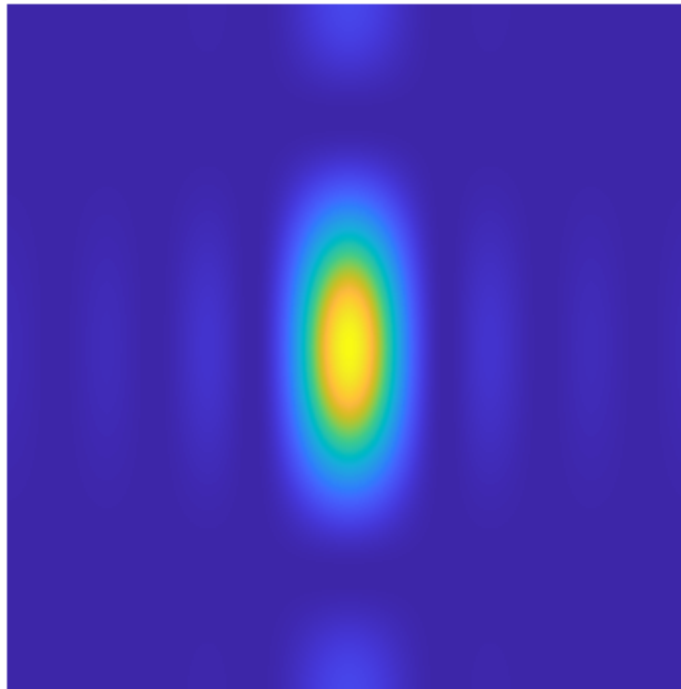
You do not have to create an entry-point function. You can directly generate code for the MATLAB® `fft2` function. To generate CUDA MEX for the MATLAB `fft2` function, in the configuration object, set the `EnablecuFFT` property and use the `codegen` function. GPU Coder replaces `fft`, `ifft`, `fft2`, `ifft2`, `fftn`, and `ifftn` function calls in your MATLAB code to the appropriate cuFFT library calls. For two-dimensional transforms and higher, GPU Coder creates multiple 1-D batched transforms. These batched transforms have higher performance than single transforms. After generating the MEX function, you can verify that it has the same functionality as the original MATLAB entry-point function. Run the generated `fft2_mex` and plot the results.

```
cfg = coder.gpuConfig('mex');
cfg.GpuConfig.EnablecuFFT = 1;
codegen -config cfg -args {lightsource} fft2

farfieldsignalGPU = fft2_mex(lightsource);
str = sprintf('Rectangular Aperture Far-Field Diffraction Pattern on GPU');
visualize(farfieldsignalGPU,str);
```

Code generation successful.

Rectangular Aperture Far-Field Diffraction Pattern on GPU



Simulate The Young's Double-Slit Experiment

Young's double-slit experiment shows light interference when an aperture comprises two parallel slits. A series of bright points is visible where constructive interference takes place. In this case, form

the aperture representing two slits. Restrict the aperture in the y direction to ensure that the resulting pattern is not entirely concentrated along the horizontal axis.

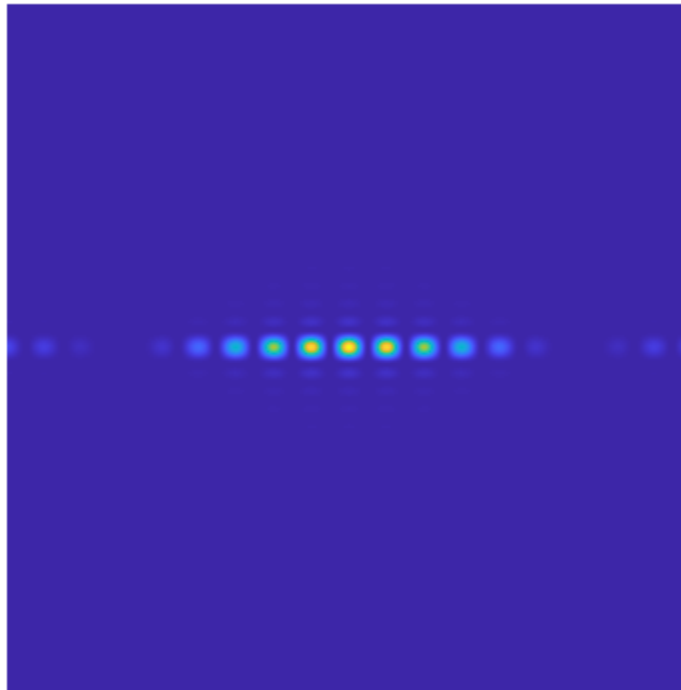
```
slits          = (abs(gx) <= 10/N2) .* (abs(gx) >= 8/N2);
aperture      = slits .* (abs(gy) < 20/N2);
lightsource   = double(aperture);
```

Display the Light Intensity for Young's Double-Slit

Because the size, type and complexity of the inputs remains the same, reuse the `fft2_mex` generated MEX-function and display the intensity as before.

```
farfieldsignalGPU = fft2_mex(lightsource);
str = sprintf('Double Slit Far-Field Diffraction Pattern on GPU');
visualize(farfieldsignalGPU,str);
```

Double Slit Far-Field Diffraction Pattern on GPU



Generate CUDA MEX Using GPU Pointer as Input

In the CUDA MEX generated above, the input provided to MEX is copied from CPU to GPU memory, the computation is performed on the GPU and the result is copied back to the CPU. Alternatively, CUDA code can be generated such that it accepts GPU pointers directly. For MEX targets, GPU pointers can be passed from MATLAB® to CUDA MEX using `gpuArray`. For other targets, GPU memory must be allocated and inputs must be copied from CPU to GPU inside the handwritten main function, before they are passed to the entry-point function.

```
lightsource_gpu = gpuArray(lightsource);
cfg = coder.gpuConfig('mex');
```

```
cfg.GpuConfig.EnableCUFFT = 1;  
codegen -config cfg -args {lightsource_gpu} fft2 -o fft2_gpu_mex
```

Code generation successful.

Only numeric and logical input matrix types can be passed as GPU pointers to the entry-point function. Other data types that are not supported can be passed as CPU inputs. During code generation, if at least one of the inputs provided to the entry-point function is a GPU pointer, the outputs returned from the function are also GPU pointers. However, if the data type of the output is not supported as a GPU pointer, such as a struct or a cell-array, the output will be returned as a CPU pointer. For more information on passing GPU pointers to entry-point function, see “Support for GPU Arrays” on page 2-37.

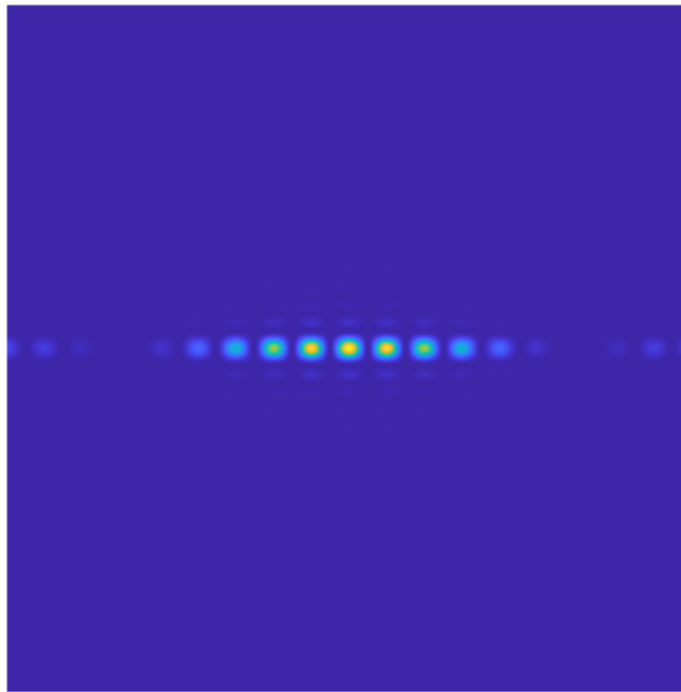
Notice the difference in the generated CUDA code when using `lightsource_gpu` GPU input. It avoids copying the input from CPU to GPU memory and avoids copying the result back from GPU to CPU memory. This results in fewer `cudaMemcpy`s and improves the performance of the generated CUDA MEX.

Verify Results of CUDA MEX Using GPU Pointer as Input

To verify that the generated CUDA MEX using `gpuArray` has the same functionality, run the generated `fft2_gpu_mex`, gather the results on the host and plot the results.

```
farfieldsignal_gpu = fft2_gpu_mex(lightsource_gpu);  
farfieldsignal_cpu = gather(farfieldsignal_gpu);  
str = sprintf('Double Slit Far-Field Diffraction Pattern on GPU using gpuArray');  
visualize(farfieldsignal_cpu,str);
```

Double Slit Far-Field Diffraction Pattern on GPU using gpuArray



See Also

Functions

[codegen](#) | [coder.gpu.kernel](#) | [coder.gpu.kerndfun](#) | [gpuCoder.matrixMatrixKernel](#) | [coder.gpu.constantMemory](#) | [gpuCoder.stencilKernel](#) | [coder.checkGpuInstall](#)

Objects

[coder.gpuConfig](#) | [coder.CodeConfig](#) | [coder.EmbeddedCodeConfig](#) | [coder.gpuEnvConfig](#)

Related Examples

- “Kernels from Library Calls” on page 2-8
- “Design Patterns” on page 2-26
- “Kernels from Scatter-Gather Type Operations” on page 2-4
- “Call Custom CUDA Device Function from the Generated Code” on page 2-22

Benchmark Solving a Linear System by Using GPU Coder

This example shows how to benchmark solving a linear system by generating CUDA® code. Use matrix left division, also known as `mldivide` or the backslash operator (`\`), to solve the system of linear equations $A*x = b$ for x (that is, compute $x = A\b$).

Third-Party Prerequisites

Required

This example generates CUDA MEX and has the following third-party requirements.

- CUDA enabled NVIDIA® GPU and compatible driver.

Optional

For non-MEX builds such as static, dynamic libraries or executables, this example has the following additional requirements.

- NVIDIA toolkit.
- Environment variables for the compilers and libraries. For more information, see “Third-Party Hardware” and “Setting Up the Prerequisite Products”.

Verify GPU Environment

To verify that the compilers and libraries necessary for running this example are set up correctly, use the `coder.checkGpuInstall` function.

```
envCfg = coder.gpuEnvConfig('host');
envCfg.BasicCodegen = 1;
envCfg.Quiet = 1;
coder.checkGpuInstall(envCfg);
```

Determine the Maximum Data Size

Choose the appropriate matrix size for the computations by specifying the amount of system memory in GB available to the CPU and the GPU. The default value is based only on the amount of memory available on the GPU. You can specify a value that is appropriate for your system.

```
g = gpuDevice;
maxMemory = 0.1*g.AvailableMemory/1024^3;
```

Note:

This example uses `cuSOLVER` libraries that have significant GPU memory requirements for creating workspaces. If you run into CUDA out-of-memory errors, reduce the `maxMemory` or the matrix step sizes in `sizeSingle` and `sizeDouble`.

The Benchmarking Function

This example benchmarks matrix left division (`\`) including the cost of transferring data between the CPU and GPU, to get a clear view of the total application time when using GPU Coder™. The application time profiling must not include the time to create sample input data. The `genData.m` function separates generation of test data from the entry-point function that solves the linear system.

```
type getData.m
function [A, b] = getData(n, clz)
% Copyright 2017-2022 The MathWorks, Inc.
    fprintf('Creating a matrix of size %d-by-%d.\n', n, n);
    A = rand(n, n, clz) + 100*eye(n, n, clz);
    b = rand(n, 1, clz);
end
```

The Backslash Entry-Point Function

The `backslash.m` entry-point function encapsulates the (`\`) operation for which you want to generate code.

```
type backslash.m
function [x] = backslash(A,b)
%#codegen
% Copyright 2017-2022 The MathWorks, Inc.
    coder.gpu.kerndefun();
    x = A\b;
end
```

Generate the GPU Code

Create a function to generate the GPU MEX function based on the particular input data size.

```
type genGpuCode.m
function [] = genGpuCode(A, b)
% Copyright 2017-2022 The MathWorks, Inc.
    cfg = coder.gpuConfig('mex');
    evalc('codegen -config cfg -args {A,b} backslash');
end
```

Choose a Problem Size

The performance of the parallel algorithms that solve a linear system depends greatly on the matrix size. This example compares the performance of the algorithm for different matrix sizes (multiples of 1024).

```
sizeLimit = inf;
if ispc
    sizeLimit = double(intmax('int32'));
end
maxSizeSingle = min(floor(sqrt(maxMemory*1024^3/4)), floor(sqrt(sizeLimit/4)));
maxSizeDouble = min(floor(sqrt(maxMemory*1024^3/8)), floor(sqrt(sizeLimit/8)));
step = 1024;
if maxSizeDouble/step >= 10
    step = step*floor(maxSizeDouble/(5*step));
end
sizeSingle = 1024:step:maxSizeSingle;
```



```
sizeDouble = 1024:step:maxSizeDouble;
numReps = 5;
```

Compare Performance: Speedup

Use the total elapsed time as a measure of performance because that enables you to compare the performance of the algorithm for different matrix sizes. Given a matrix size, the benchmarking function creates the matrix A and the right-side b once, and then solves $A \setminus b$ a few times to get an accurate measure of the time it takes.

type [benchFcnMat.m](#)

```
function time = benchFcnMat(A, b, reps)

% Copyright 2017-2022 The MathWorks, Inc.

    time = inf;
    % Solve the linear system a few times and take the best run
    for itr = 1:reps
        tic;
        matX = backslash(A, b);
        tcurr = toc;
        time = min(tcurr, time);
    end
end
```

Create a different function for GPU code execution that invokes the generated GPU MEX function.

type [benchFcnGpu.m](#)

```
function time = benchFcnGpu(A, b, reps)

% Copyright 2017-2022 The MathWorks, Inc.

    time = inf;
    gpuX = backslash_mex(A, b);
    for itr = 1:reps
        tic;
        gpuX = backslash_mex(A, b);
        tcurr = toc;
        time = min(tcurr, time);
    end
end
```

Execute the Benchmarks

When you execute the benchmarks, the computations can take a long time to complete. Print some intermediate status information as you complete the benchmarking for each matrix size. Encapsulate the loop over all the matrix sizes in a function to benchmark single- and double-precision computations.

Actual execution times can vary across different hardware configurations. This benchmarking was done by using MATLAB R2022a on a machine with a 6 core, 3.5GHz Intel® Xeon® CPU and an NVIDIA TITAN Xp GPU.

type [executeBenchmarks.m](#)

```
function [timeCPU, timeGPU] = executeBenchmarks(clz, sizes, reps)
```

```
% Copyright 2017-2022 The MathWorks, Inc.

fprintf(['Starting benchmarks with %d different %s-precision ' ...
        'matrices of sizes\nranging from %d-by-%d to %d-by-%d.\n'], ...
        length(sizes), clz, sizes(1), sizes(1), sizes(end), ...
        sizes(end));
timeGPU = zeros(size(sizes));
timeCPU = zeros(size(sizes));
for i = 1:length(sizes)
    n = sizes(i);
    fprintf('Size : %d\n', n);
    [A, b] = getData(n, clz);
    genGpuCode(A, b);
    timeCPU(i) = benchFcnMat(A, b, reps);
    fprintf('Time on CPU: %f sec\n', timeCPU(i));
    timeGPU(i) = benchFcnGpu(A, b, reps);
    fprintf('Time on GPU: %f sec\n', timeGPU(i));
    fprintf('\n');
end
end
```

Execute the benchmarks in single and double precision.

```
[cpu, gpu] = executeBenchmarks('single', sizeSingle, numReps);
```

```
Starting benchmarks with 9 different single-precision matrices of sizes
ranging from 1024-by-1024 to 17408-by-17408.
```

```
Size : 1024
```

```
Creating a matrix of size 1024-by-1024.
```

```
Time on CPU: 0.012281 sec
```

```
Time on GPU: 0.008329 sec
```

```
Size : 3072
```

```
Creating a matrix of size 3072-by-3072.
```

```
Time on CPU: 0.115839 sec
```

```
Time on GPU: 0.035071 sec
```

```
Size : 5120
```

```
Creating a matrix of size 5120-by-5120.
```

```
Time on CPU: 0.380651 sec
```

```
Time on GPU: 0.074228 sec
```

```
Size : 7168
```

```
Creating a matrix of size 7168-by-7168.
```

```
Time on CPU: 0.867239 sec
```

```
Time on GPU: 0.127977 sec
```

```
Size : 9216
```

```
Creating a matrix of size 9216-by-9216.
```

```
Time on CPU: 1.677065 sec
```

```
Time on GPU: 0.205344 sec
```

```
Size : 11264
```

```
Creating a matrix of size 11264-by-11264.
```

```
Time on CPU: 2.911081 sec
```

```
Time on GPU: 0.306867 sec
```

```
Size : 13312
```

```
Creating a matrix of size 13312-by-13312.  
Time on CPU: 4.684644 sec  
Time on GPU: 0.440095 sec
```

```
Size : 15360  
Creating a matrix of size 15360-by-15360.  
Time on CPU: 6.950956 sec  
Time on GPU: 0.608897 sec
```

```
Size : 17408  
Creating a matrix of size 17408-by-17408.  
Time on CPU: 9.833478 sec  
Time on GPU: 0.802604 sec
```

```
results.sizeSingle = sizeSingle;  
results.timeSingleCPU = cpu;  
results.timeSingleGPU = gpu;  
[cpu, gpu] = executeBenchmarks('double', sizeDouble, numReps);
```

Starting benchmarks with 6 different double-precision matrices of sizes ranging from 1024-by-1024 to 11264-by-11264.

```
Size : 1024  
Creating a matrix of size 1024-by-1024.  
Time on CPU: 0.021463 sec  
Time on GPU: 0.010796 sec
```

```
Size : 3072  
Creating a matrix of size 3072-by-3072.  
Time on CPU: 0.213805 sec  
Time on GPU: 0.093114 sec
```

```
Size : 5120  
Creating a matrix of size 5120-by-5120.  
Time on CPU: 0.689023 sec  
Time on GPU: 0.323026 sec
```

```
Size : 7168  
Creating a matrix of size 7168-by-7168.  
Time on CPU: 1.687437 sec  
Time on GPU: 0.775834 sec
```

```
Size : 9216  
Creating a matrix of size 9216-by-9216.  
Time on CPU: 3.521580 sec  
Time on GPU: 1.539601 sec
```

```
Size : 11264  
Creating a matrix of size 11264-by-11264.  
Time on CPU: 6.075310 sec  
Time on GPU: 2.694465 sec
```

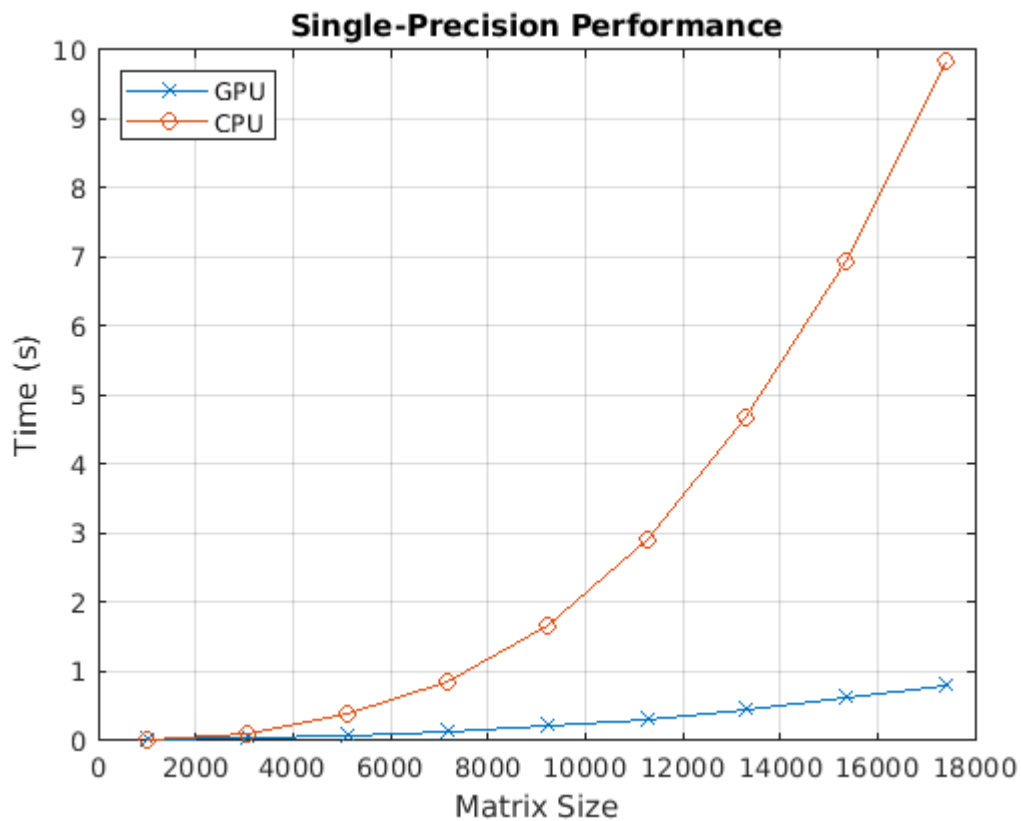
```
results.sizeDouble = sizeDouble;  
results.timeDoubleCPU = cpu;  
results.timeDoubleGPU = gpu;
```

Plot the Performance

Plot the results and compare the performance on the CPU and the GPU for single and double precision.

First, look at the performance of the backslash operator in single precision.

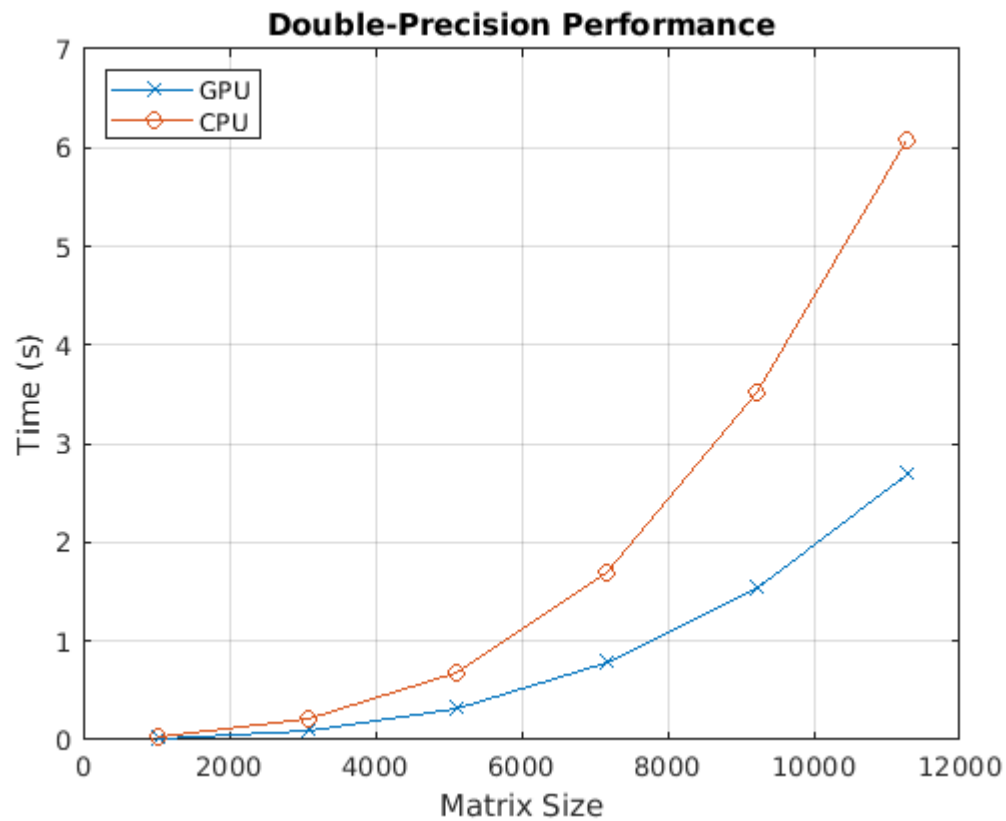
```
fig = figure;
ax = axes('parent', fig);
plot(ax, results.sizeSingle, results.timeSingleGPU, '-x', ...
      results.sizeSingle, results.timeSingleCPU, '-o')
grid on;
legend('GPU', 'CPU', 'Location', 'NorthWest');
title(ax, 'Single-Precision Performance')
ylabel(ax, 'Time (s)');
xlabel(ax, 'Matrix Size');
```



```
drawnow;
```

Now, look at the performance of the backslash operator in double precision.

```
fig = figure;
ax = axes('parent', fig);
plot(ax, results.sizeDouble, results.timeDoubleGPU, '-x', ...
      results.sizeDouble, results.timeDoubleCPU, '-o')
legend('GPU', 'CPU', 'Location', 'NorthWest');
grid on;
title(ax, 'Double-Precision Performance')
ylabel(ax, 'Time (s)');
xlabel(ax, 'Matrix Size');
drawnow;
```

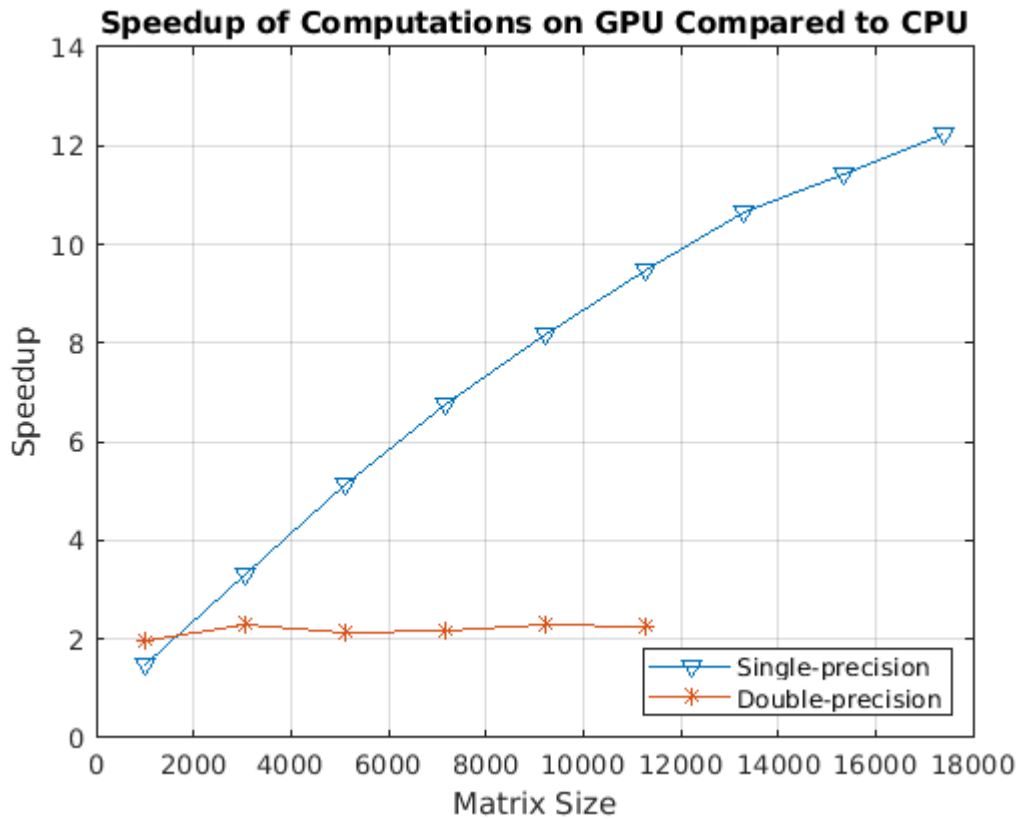


Finally, look at the speedup of the backslash operator when comparing the GPU to the CPU.

```

speedupDouble = results.timeDoubleCPU./results.timeDoubleGPU;
speedupSingle = results.timeSingleCPU./results.timeSingleGPU;
fig = figure;
ax = axes('parent', fig);
plot(ax, results.sizeSingle, speedupSingle, '-v', ...
      results.sizeDouble, speedupDouble, '-*')
grid on;
legend('Single-precision', 'Double-precision', 'Location', 'SouthEast');
title(ax, 'Speedup of Computations on GPU Compared to CPU');
ylabel(ax, 'Speedup');
xlabel(ax, 'Matrix Size');
drawnow;

```



See Also

Functions

[codegen](#) | [coder.gpu.kernel](#) | [coder.gpu.kernelfun](#) | [gpucoder.matrixMatrixKernel](#) | [coder.gpu.constantMemory](#) | [gpucoder.stencilKernel](#) | [coder.checkGpuInstall](#)

Objects

[coder.gpuConfig](#) | [coder.CodeConfig](#) | [coder.EmbeddedCodeConfig](#) | [coder.gpuEnvConfig](#)

Related Examples

- “Kernels from Library Calls” on page 2-8
- “Design Patterns” on page 2-26
- “Kernels from Scatter-Gather Type Operations” on page 2-4
- “Call Custom CUDA Device Function from the Generated Code” on page 2-22

QR Decomposition on NVIDIA GPU Using cuSOLVER Libraries

This example shows how to create a standalone CUDA® executable that leverages the CUDA Solver library (cuSOLVER). The example uses a curve fitting application that mimics automatic lane tracking on a road to illustrate:

- Fitting an arbitrary-order polynomial to noisy data by using matrix QR factorization.
- Using the `coder.LAPACKCallback` class to provide the LAPACK library information for the code generator when generating standalone executables.

Prerequisites

- CUDA enabled NVIDIA® GPU.
- NVIDIA CUDA toolkit and driver.
- LAPACK library that is optimized for your execution environment. For more information, see LAPACK vendors implementations. This example uses the `mwlapack` libraries that MATLAB® provides in `matlabroot/extern/lib`.
- Environment variables for the compilers and libraries. For information on the supported versions of the compilers and libraries, see “Third-Party Hardware”. For setting up the environment variables, see “Setting Up the Prerequisite Products”.

Verify GPU Environment

To verify that the compilers and libraries necessary for running this example are set up correctly, use the `coder.checkGpuInstall` function.

```
envCfg = coder.gpuEnvConfig('host');
envCfg.BasicCodegen = 1;
envCfg.Quiet = 1;
coder.checkGpuInstall(envCfg);
```

Solve a Linear System by Using Matrix Factorization

In curve fitting applications, the objective is to estimate the coefficients of a low-order polynomial. The polynomial is then used as a model for observed noisy data, which in this example represents the lane boundary of the road ahead of a vehicle. For example, when using a quadratic polynomial, there are three coefficients (a , b , and c) to estimate:

$$ax^2 + bx + c$$

The polynomial that fits best is defined as the one that minimizes the sum of the squared errors between itself and the noisy data. To solve this least-squares problem, you get and solve an overdetermined linear system. An explicit matrix inverse is not required to solve the system.

In this example, the unknowns are the coefficients of each term in the polynomial. Because the polynomial you use as a model always starts from the current position on the road, the constant term in the polynomial is assumed to be zero. Estimate the coefficients for the linear and higher-order terms. Set up a matrix equation $Ax=y$ such that:

- y contains the sensor outputs.
- x contains the polynomial coefficients that we need to obtain.

- A is a constant matrix related to the order of the polynomial and the locations of the sensors.

Solve the equation using the QR factorization of A :

$$Ax = QRx = y$$

and

$$x = \text{pinv}(A) * y = R^{-1}Q^T * y$$

where $\text{pinv}()$ represents pseudo-inverse. Given the matrix A , you can use the following code to implement a solution of this matrix equation. Factoring A allows for an easier solution of the system.

```
[Q,R,P] = qr(A);
z = Q' * y;
x = R \ z;
yhat = A * x;
```

Use the `linsolveQR` function to solve the equation using the QR factorization of A .

type `linsolveQR.m`

```
function [yhat,x] = linsolveQR(A,y)
%#codegen

% Copyright 2019 The MathWorks, Inc.

[Q,R,P] = qr(A);
z = Q' * y;
x = R \ z;
yhat = A * x;

end
```

Signal Model for the Road

To test the algorithm, a continuously curving road model is used, that is, a sinusoid that is contaminated with additive noise. By varying the frequency of the sinusoid in the model, you can stress the algorithm by different amounts. This code simulates noisy sensor outputs using our road model:

```
% Duration - Distance that we look ahead
% N - Total number of sensors providing estimates of road boundary
% Ts - Sample interval
% FracPeriod - Fraction of period of sinusoid to match
% y - Contains the simulated sensor outputs
Duration = 2;
N = 25;
Ts = Duration / N;
FracPeriod = 0.5;
y = sin(2*pi* (0:N-1)' * (FracPeriod/N)) + sqrt(0.3) * randn(N,1);
```

Use this code to form the Vandermonde matrix A :

```
Npoly = 3; % Order of polynomial to use in fitting
v = (0:Ts:(N-1)*Ts)';
A = zeros(length(v), Npoly);
for i = Npoly : -1 : 1
```



```

    A(:,i) = v.^i;
end

```

The Vandermonde matrix A and sensor outputs matrix y are passed as input parameters to the `linsolveQR` entry-point function. These inputs are written to comma-separated files and are read from the handwritten main `qrmain.cu`.

```

writematrix(reshape(A, 1, 75), 'inputA.csv');
writematrix(reshape(y, 1, 25), 'inputY.csv');

```

Custom Callback Class for Standalone Code Generation

The `qr` function is only partially supported in the cuSOLVER library. In such cases, GPU Coder™ uses the LAPACK library for certain linear algebra function calls. LAPACK is an external software library for numeric linear algebra. For MEX targets, the code generator uses the LAPACK library included in MATLAB.

For standalone targets, you must define a custom `coder.LAPACKCallback` class that specifies the LAPACK libraries along with the header files to use for linear algebra calls in the generated code. In this example, the `lapackCallback` callback class specifies the paths to these libraries in `updateBuildInfo` method. You must modify this file with the library names and paths for the custom LAPACK installation on your computer.

type `lapackCallback.m`

```

classdef lapackCallback < coder.LAPACKCallback
%
% Copyright 2019 The MathWorks, Inc.

methods (Static)
    function hn = getHeaderFilename()
        hn = 'lapacke.h';
    end

    function updateBuildInfo(buildInfo, buildctx)
        [~, libExt] = buildctx.getStdLibInfo();

        % Specify path to LAPACK library
        if ispc
            lapackLocation = [matlabroot, '\extern'];
            libName = ['libmwlpack' libExt];
            buildInfo.addIncludePaths([lapackLocation, '\include']);
            libPath = [lapackLocation, '\lib\win64\microsoft\'];
        else
            lapackLocation = [matlabroot];
            libName = ['libmwlpack' libExt];
            buildInfo.addIncludePaths([lapackLocation, '/extern/include']);
            libPath = [lapackLocation, '/bin/glnxa64'];
        end

        % Add include path and LAPACK library for linking
        buildInfo.addLinkObjects(libName, libPath, 1000, true, true);

        buildInfo.addDefines('HAVE_LAPACK_CONFIG_H');
        buildInfo.addDefines('LAPACK_COMPLEX_STRUCTURE');
    end
end

```

```
    end  
end
```

Standalone Code Generation

Generate a standalone executable by the specifying CustomLAPACKCallback property in the code configuration object and using a handwritten main `qrmain.cu`.

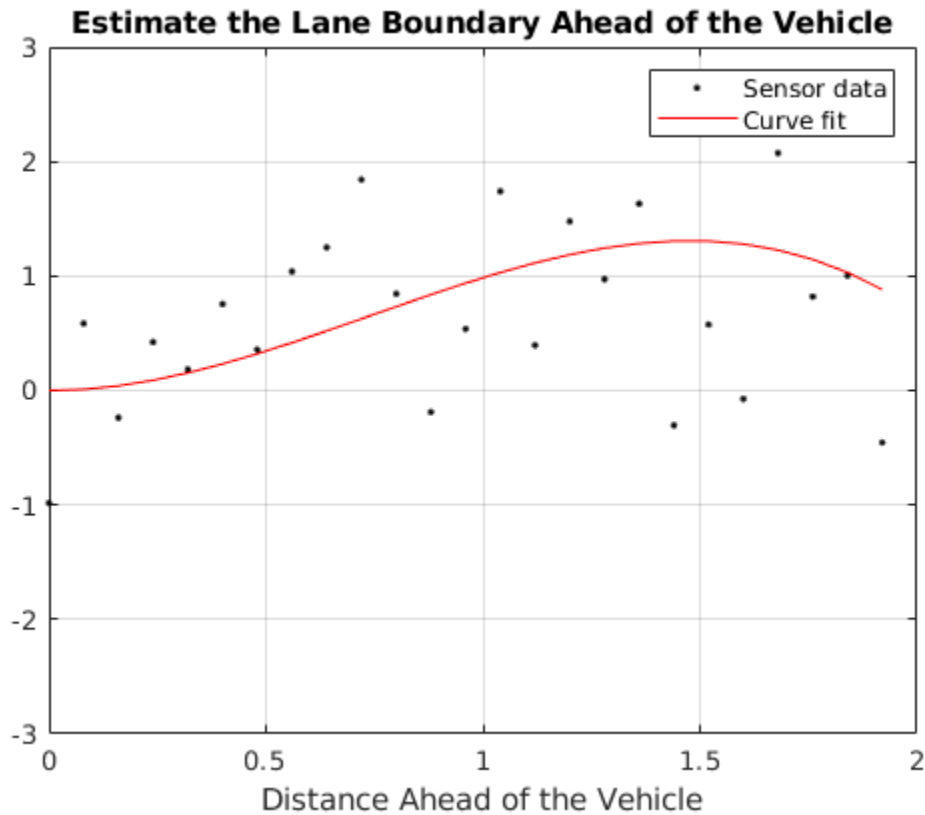
```
cfg = coder.gpuConfig('exe');  
cfg.GpuConfig.EnableCUSOLVER = 1;  
cfg.CustomLAPACKCallback = 'lapackCallback';  
cfg.CustomSource = 'qrmain.cu';  
cfg.CustomInclude = '.';  
codegen -config cfg -args {A,y} linsolveQR -report
```

Code generation successful: [View report](#)

Standalone Code Execution

When you execute the generated standalone executable, the outputs `yhat` and `x` are computed and written to comma-separated files. Read these outputs back in MATLAB and use the `plot` function to visualize the sensor data and fitted curve.

```
if ispc  
    system('linsolveQR.exe');  
else  
    system('./linsolveQR');  
end  
yhat = reshape(readmatrix('outputYhat.csv'), 25, 1);  
x = reshape(readmatrix('outputX.csv'), 3, 1);  
figure  
plot(v, y, 'k.', v, yhat, 'r')  
axis([0 N*Ts -3 3]);  
grid;  
xlabel('Distance Ahead of the Vehicle');  
legend('Sensor data','Curve fit');  
title('Estimate the Lane Boundary Ahead of the Vehicle');
```



See Also

Functions

[codegen](#) | [coder.gpu.kernel](#) | [coder.gpu.kerndfun](#) | [gpcoder.matrixMatrixKernel](#) | [coder.gpu.constantMemory](#) | [gpcoder.stencilKernel](#) | [coder.checkGpuInstall](#)

Objects

[coder.gpuConfig](#) | [coder.CodeConfig](#) | [coder.EmbeddedCodeConfig](#) | [coder.gpuEnvConfig](#)

Related Examples

- “Kernels from Library Calls” on page 2-8
- “Design Patterns” on page 2-26
- “Kernels from Scatter-Gather Type Operations” on page 2-4
- “Call Custom CUDA Device Function from the Generated Code” on page 2-22

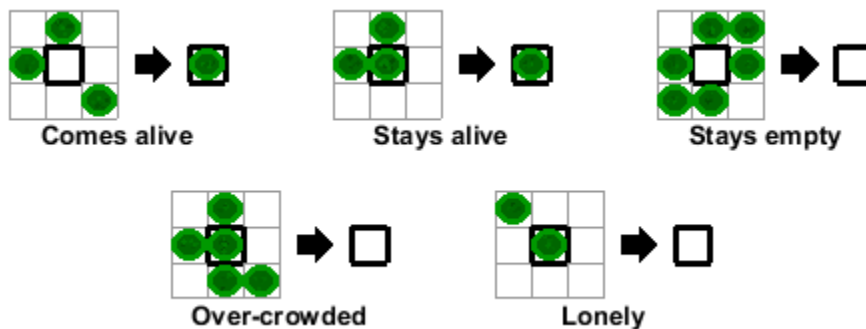
Stencil Processing on GPU

This example shows how to generate CUDA® kernels for stencil type operations by implementing "Game of Life" by John H. Conway.

"Game of Life" is a zero-player *cellular automaton* game that consists of a collection of cells (*population*) in a rectangular grid (*universe*). The cells evolve at discrete time steps known as *generations*. A set of mathematical rules applied to the cells and its neighbors control their life, death, and reproduction. This "Game of Life" implementation is based on the example provided in the e-book *Experiments with MATLAB* by Cleve Moler. The implementation follows these rules:

- Cells are arranged in a 2-D grid.
- At each step, the vitality of the eight nearest neighbors of each cell determines its fate.
- Any cell with exactly three live neighbors comes to life at the next step.
- A live cell with exactly two live neighbors remains alive at the next step.
- All other cells (including those with more than three neighbors) die at the next step or remain empty.

Here are some examples of how a cell is updated.



Many array operations can be expressed as a *stencil* operation, where each element of the output array depends on a small region of the input array. The stencil in this example is the 3-by-3 region around each cell. Finite differences, convolution, median filtering, and finite-element methods are examples of other operations that stencil processing can perform.

Third-Party Prerequisites

Required

This example generates CUDA MEX and has the following third-party requirements.

- CUDA enabled NVIDIA® GPU and compatible driver.

Optional

For non-MEX builds such as static, dynamic libraries or executables, this example has the following additional requirements.

- NVIDIA toolkit.

- Environment variables for the compilers and libraries. For more information, see “Third-Party Hardware” and “Setting Up the Prerequisite Products”.

Verify GPU Environment

To verify that the compilers and libraries necessary for running this example are set up correctly, use the `coder.checkGpuInstall` function.

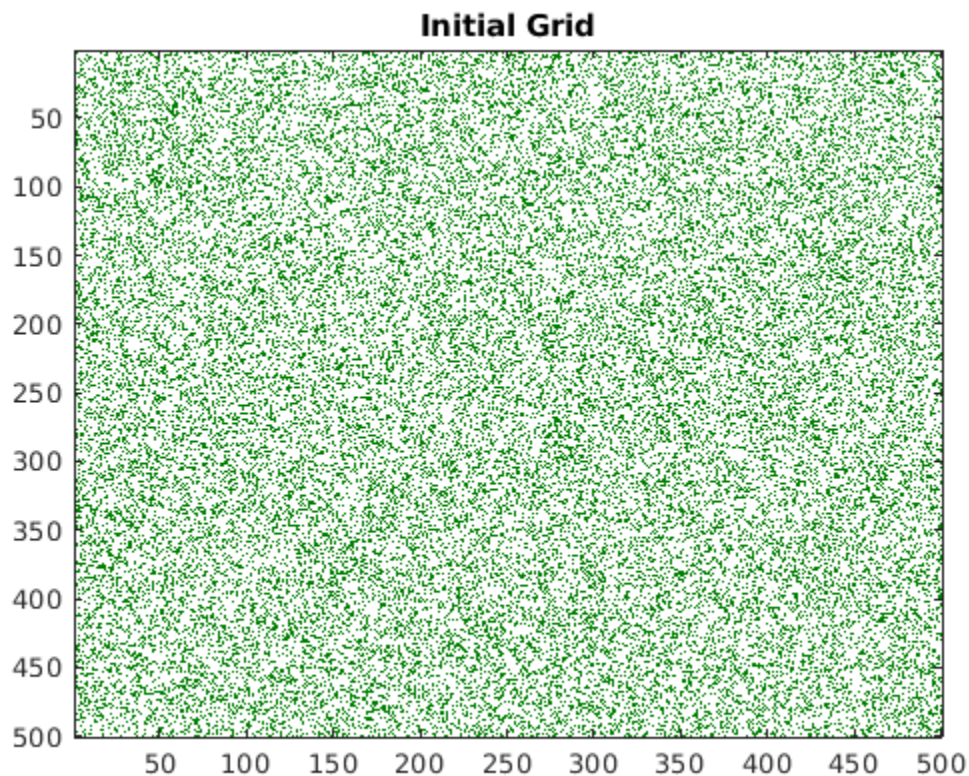
```
envCfg = coder.gpuEnvConfig('host');  
envCfg.BasicCodegen = 1;  
envCfg.Quiet = 1;  
coder.checkGpuInstall(envCfg);
```

Generate a Random Initial Population

Being that the game is zero-player, the evolution of the game is determined by its initial state. For this example, an initial population of cells is created on a two-dimensional grid with approximately 25% of the locations being alive.

```
gridSize = 500;  
numGenerations = 100;  
initialGrid = (rand(gridSize,gridSize) > .75);
```

```
% Draw the initial grid  
imagesc(initialGrid);  
colormap([1 1 1;0 0.5 0]);  
title('Initial Grid');
```



Play the Game of Life

The `gameoflife_orig.m` function is a fully vectorized implementation of "Game of Life". The function updates all cells on the grid in one pass per their generation.

type `gameoflife_orig`

```
% MATLAB vectorized implementation
function grid = gameoflife_orig(initialGrid)

% Copyright 2016-2019 The MathWorks, Inc.

numGenerations = 100;
grid = initialGrid;
[gridSize,~] = size(initialGrid);

% Loop through each generation updating the grid and displaying it.
for generation = 1:numGenerations
    grid = updateGrid(grid, gridSize);

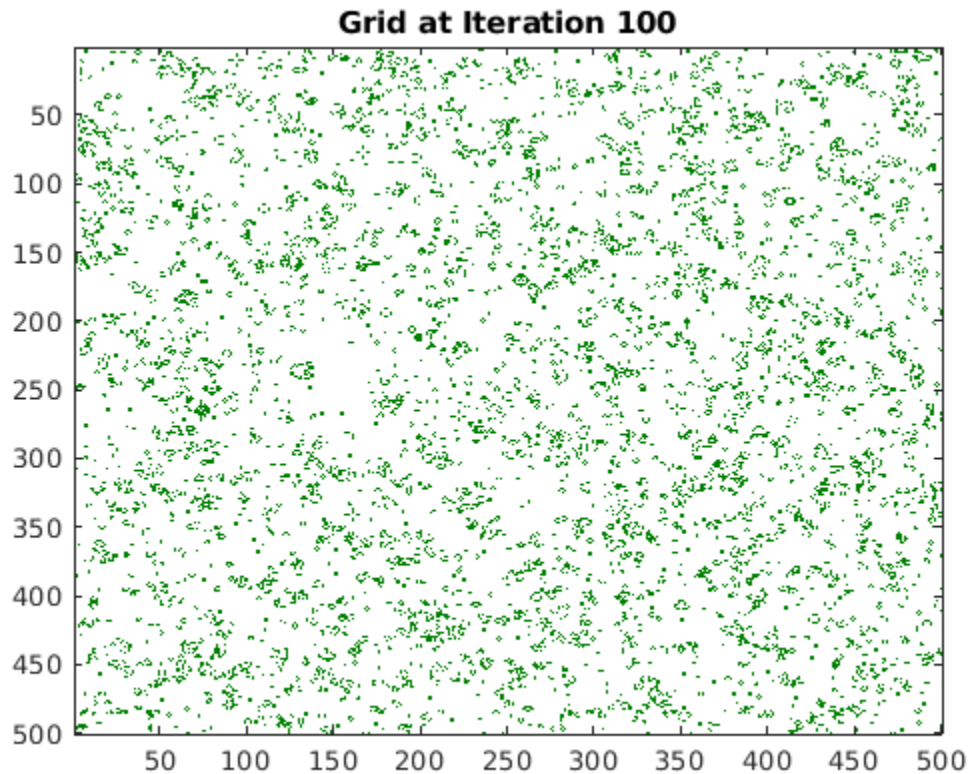
    imagesc(grid);
    colormap([1 1 1;0 0.5 0]);
    title(['Grid at Iteration ',num2str(generation)]);
    drawnow;
end

function X = updateGrid(X, N)
    % Index vectors increase or decrease the centered index by one
    % thereby accessing neighbors to the left,right,up, and down.
    p = [1 1:N-1];
    q = [2:N N];
    % Count how many of the eight neighbors are alive.
    neighbors = X(:,p) + X(:,q) + X(p,:) + X(q,:) + ...
        X(p,p) + X(q,q) + X(p,q) + X(q,p);
    % A live cell with two live neighbors, or any cell with
    % three live neighbors, is alive at the next step.
    X = (X & (neighbors == 2)) | (neighbors == 3);
end

end
```

Play the game by calling the `gameoflife_orig` function with an initial population. The game iterates through 100 generations and displays the population at each generation.

```
gameoflife_orig(initialGrid);
```



Convert the Game of Life for GPU Code Generation

Looking at the calculations in the `updateGrid` function, it is apparent that the same operations are applied at each grid location independently. However, each cell must know about its eight neighbors. The modified `gameoflife_stencil.m` function uses the `stencilfun` pragma to compute a 3-by-3 region around each cell. The GPU Coder™ implementation of the stencil kernel computes one element of the grid in each thread and uses shared memory to improve memory bandwidth and data locality.

```
type gameoflife_stencil
```

```
function grid = gameoflife_stencil(initialGrid) %#codegen
% Copyright 2016-2019 The MathWorks, Inc.

numGenerations = 100;
grid = initialGrid;

% Loop through each generation updating the grid.
for generation = 1:numGenerations
    grid = stencilfun(@updateElem, grid, [3,3], Shape='same');
end
end

function X = updateElem(window)
neighbors = window(1,1) + window(1,2) + window(1,3) ...
```

```
+ window(2,1) + window(2,3) ...  
+ window(3,1) + window(3,2) + window(3,3);  
X = (window(2,2) & (neighbors == 2)) | (neighbors == 3);  
end
```

Generate CUDA MEX for the Function

To generate CUDA MEX for the `gameoflife_stencil` function, create a GPU code configuration object, and then use the `codegen` command.

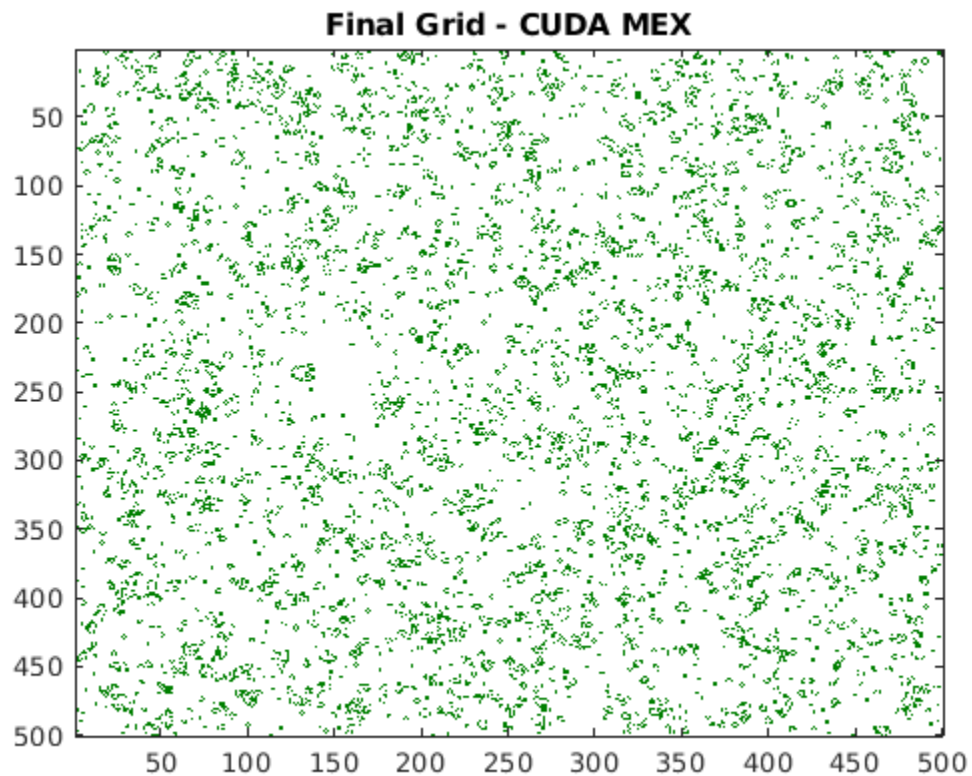
```
cfg = coder.gpuConfig('mex');  
codegen -config cfg -args {initialGrid} gameoflife_stencil
```

Code generation successful.

Run the MEX Function

Run the generated `gameoflife_stencil_mex` with the random initial population.

```
gridGPU = gameoflife_stencil_mex(initialGrid);  
% Draw the grid after 100 generations  
imagesc(gridGPU);  
colormap([1 1 1;0 0.5 0]);  
title('Final Grid - CUDA MEX');
```

See Also

Functions

[codegen](#) | [coder.gpu.kernel](#) | [coder.gpu.kerndfun](#) | [gpuscoder.matrixMatrixKernel](#) | [coder.gpu.constantMemory](#) | [stencilfun](#) | [coder.checkGpuInstall](#)

Objects

[coder.gpuConfig](#) | [coder.CodeConfig](#) | [coder.EmbeddedCodeConfig](#) | [coder.gpuEnvConfig](#)

Related Examples

- “Kernels from Library Calls” on page 2-8
- “Design Patterns” on page 2-26
- “Kernels from Scatter-Gather Type Operations” on page 2-4
- “Call Custom CUDA Device Function from the Generated Code” on page 2-22

Fog Rectification

This example shows the use of image processing functions for GPU code generation. The example takes a foggy image as input and produces a defogged image. This example is a typical implementation of fog rectification algorithm. The example uses `conv2`, `im2gray`, and `imhist` functions.

Third-Party Prerequisites

Required

This example generates CUDA® MEX and has the following third-party requirements.

- CUDA enabled NVIDIA® GPU and compatible driver. For half-precision code generation, the GPU must have a minimum compute capability of 6.0.

Optional

For non-MEX builds such as static, dynamic libraries or executables, this example has the following additional requirements.

- NVIDIA toolkit.
- Environment variables for the compilers and libraries. For more information, see “Third-Party Hardware” and “Setting Up the Prerequisite Products”.

Verify GPU Environment

To verify that the compilers and libraries necessary for running this example are set up correctly, use the `coder.checkGpuInstall` function.

```
envCfg = coder.gpuEnvConfig('host');
envCfg.BasicCodegen = 1;
envCfg.Quiet = 1;
coder.checkGpuInstall(envCfg);
```

The `fog_rectification` Entry-Point Function

The `fog_rectification.m` entry-point function takes a foggy image as input and returns a defogged image.

```
type fog_rectification

function [out] = fog_rectification(input) %#codegen

% Copyright 2017-2019 The MathWorks, Inc.

coder.gpu.kernelfun;

% restoreOut is used to store the output of restoration
restoreOut = zeros(size(input),'double');

% Changing the precision level of input image to double
input = double(input)./255;

%% Dark channel Estimation from input
darkChannel = min(input,[],3);
```

```

% diff_im is used as input and output variable for anisotropic diffusion
diff_im = 0.9*darkChannel;
num_iter = 3;

% 2D convolution mask for Anisotropic diffusion
hN = [0.0625 0.1250 0.0625; 0.1250 0.2500 0.1250; 0.0625 0.1250 0.0625];
hN = double(hN);

%% Refine dark channel using Anisotropic diffusion.
for t = 1:num_iter
    diff_im = conv2(diff_im,hN,'same');
end

%% Reduction with min
diff_im = min(darkChannel,diff_im);

diff_im = 0.6*diff_im ;

%% Parallel element-wise math to compute
% Restoration with inverse Koschmieder's law
factor = 1.0./(1.0-(diff_im));
restoreOut(:,:,1) = (input(:,:,1)-diff_im).*factor;
restoreOut(:,:,2) = (input(:,:,2)-diff_im).*factor;
restoreOut(:,:,3) = (input(:,:,3)-diff_im).*factor;
restoreOut = uint8(255.*restoreOut);
restoreOut = uint8(restoreOut);

%%
% Stretching performs the histogram stretching of the image.
% im is the input color image and p is cdf limit.
% out is the contrast stretched image and cdf is the cumulative prob.
% density function and T is the stretching function.

p = 5;
% RGB to grayscale conversion
im_gray = im2gray(restoreOut);
[row,col] = size(im_gray);

% histogram calculation
[count,~] = imhist(im_gray);
prob = count'/(row*col);

% cumulative Sum calculation
cdf = cumsum(prob(:));

% finding less than particular probability
i1 = length(find(cdf <= (p/100)));
i2 = 255-length(find(cdf >= 1-(p/100)));

o1 = floor(255*.10);
o2 = floor(255*.90);

t1 = (o1/i1)*[0:i1];
t2 = (((o2-o1)/(i2-i1))*[i1+1:i2]) - (((o2-o1)/(i2-i1))*i1)+o1;
t3 = (((255-o2)/(255-i2))*[i2+1:255]) - (((255-o2)/(255-i2))*i2)+o2;

T = (floor([t1 t2 t3]));

```

```
restoreOut(restoreOut == 0) = 1;

u1 = (restoreOut(:,:,1));
u2 = (restoreOut(:,:,2));
u3 = (restoreOut(:,:,3));

% Replacing the value from look up table
out1 = T(u1);
out2 = T(u2);
out3 = T(u3);

out = zeros([size(out1),3], 'uint8');
out(:,:,1) = uint8(out1);
out(:,:,2) = uint8(out2);
out(:,:,3) = uint8(out3);
return
```

Generate CUDA Code and MEX function

Set up the input for code generation and create a configuration for GPU code generation.

```
inputImage = imread('foggyInput.png');
cfg = coder.gpuConfig('mex');
```

Run Code Generation

Generate the `fog_rectification_mex` MEX file by using the `codegen` command.

```
codegen -args {inputImage} -config cfg fog_rectification
```

Code generation successful: [View report](#)

Run the MEX Function with Foggy Image

Run the generated `fog_rectification_mex` with a foggy input image, and then plot the foggy and defogged images.

```
[outputImage] = fog_rectification_mex(inputImage);

% plot images
p1 = subplot(1, 2, 1);
p2 = subplot(1, 2, 2);
imshow(inputImage, 'Parent', p1);
imshow(outputImage, 'Parent', p2);
title(p1, 'Foggy Input Image');
title(p2, 'Defogged Output Image');
```



Because of architectural differences between the CPU and GPU, numeric verification does not always match. This scenario is true when using the single data type or when performing integer type conversion in your MATLAB code. In this example, the integer type conversion in the `fog_rectification.m` entry-point function produces numeric differences with MATLAB simulation.

Half-Precision

Computations in this example can also be done in half-precision floating point numbers, using the `fog_rectification_half_precision.m` entry-point function. To generate and execute code with half-precision data types, CUDA compute capability of 6.0 or higher is required. Set the `ComputeCapability` property of the code configuration object to `'6.0'`. For half-precision, the memory allocation (malloc) mode for generating CUDA code must be set to `'Discrete'`.

```
inputImageHalf = half(imread('foggyInput.png'));
cfg = coder.gpuConfig('mex');
cfg.GpuConfig.ComputeCapability = '6.0';
cfg.GpuConfig.MallocMode = 'Discrete';
codegen -args {inputImageHalf} -config cfg fog_rectification_half_precision
```

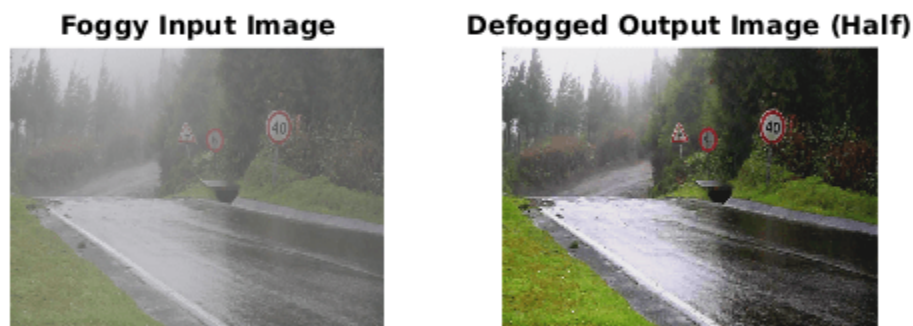
Code generation successful: [View report](#)

Run the Half-Precision MEX Function with Foggy Image

Run the generated `fog_rectification_half_precision_mex` with a foggy input image, and then plot the foggy and defogged images.

```
[outputImageHalf] = fog_rectification_half_precision_mex(inputImageHalf);
```

```
% plot images
p1 = subplot(1, 2, 1);
p2 = subplot(1, 2, 2);
imshow(inputImage, 'Parent', p1);
imshow(outputImageHalf, 'Parent', p2);
title(p1, 'Foggy Input Image');
title(p2, 'Defogged Output Image (Half)');
```



See Also

Functions

[codegen](#) | [coder.gpu.kernel](#) | [coder.gpu.kerndfun](#) | [gpcoder.matrixMatrixKernel](#) | [coder.gpu.constantMemory](#) | [gpcoder.stencilKernel](#) | [coder.checkGpuInstall](#)

Objects

[coder.gpuConfig](#) | [coder.CodeConfig](#) | [coder.EmbeddedCodeConfig](#) | [coder.gpuEnvConfig](#)

Related Examples

- “Kernels from Library Calls” on page 2-8
- “Design Patterns” on page 2-26
- “Kernels from Scatter-Gather Type Operations” on page 2-4
- “Call Custom CUDA Device Function from the Generated Code” on page 2-22

Stereo Disparity

This example shows how to generate a CUDA® MEX function from a MATLAB® function that computes the stereo disparity of two images.

Third-Party Prerequisites

Required

This example generates CUDA MEX and has the following third-party requirements.

- CUDA enabled NVIDIA® GPU and compatible driver. For half-precision code generation, the GPU device must have a minimum compute capability of 6.0.

Optional

For non-MEX builds such as static, dynamic libraries or executables, this example has the following additional requirements.

- NVIDIA toolkit.
- Environment variables for the compilers and libraries. For more information, see “Third-Party Hardware” and “Setting Up the Prerequisite Products”.

Verify GPU Environment

To verify that the compilers and libraries necessary for running this example are set up correctly, use the `coder.checkGpuInstall` function.

```
envCfg = coder.gpuEnvConfig('host');
envCfg.BasicCodegen = 1;
envCfg.Quiet = 1;
coder.checkGpuInstall(envCfg);
```

Stereo Disparity Calculation

The `stereoDisparity.m` entry-point function takes two images and returns a stereo disparity map computed from the two images.

```
type stereoDisparity
```

```
%% Modified Algorithm for Stereo Disparity Block Matching
% In this implementation, instead of finding shifted image, indices are
% mapped accordingly to save memory and some processing. RGBA column major
% packed data is used as input for compatibility with CUDA intrinsics.
% Convolution is performed using separable filters (horizontal and then
% vertical).
% Copyright 2017-2021 The MathWorks, Inc.

function [out_disp] = stereoDisparity(img0,img1) %#codegen

% Copyright 2017-2019 The MathWorks, Inc.

% GPU code generation pragma
coder.gpu.kernelFun;
```

```
%% Stereo Disparity Parameters
% |WIN_RAD| is the radius of the window to be operated. |min_disparity| is
% the minimum disparity level the search continues for. |max_disparity| is
% the maximum disparity level the search continues for.
WIN_RAD = 8;
min_disparity = -16;
max_disparity = 0;

%% Image Dimensions for Loop Control
% The number of channels packed are 4 (RGBA) so as nChannels are 4.
[imgHeight,imgWidth]=size(img0);
nChannels = 4;
imgHeight = imgHeight/nChannels;

%% Store the Raw Differences
diff_img = zeros([imgHeight+2*WIN_RAD,imgWidth+2*WIN_RAD],'int32');

% Store the minimum cost
min_cost = zeros([imgHeight,imgWidth],'int32');
min_cost(:, :) = 99999999;

% Store the final disparity
out_disp = zeros([imgHeight,imgWidth],'int16');

%% Filters for Aggregating the Differences
% |filter_h| is the horizontal filter used in separable convolution.
% |filter_v| is the vertical filter used in separable convolution which
% operates on the output of the row convolution.
filt_h = ones([1 17],'int32');
filt_v = ones([17 1],'int32');

% Main Loop that runs for all the disparity levels. This loop is
% expected to run on CPU.
for d=min_disparity:max_disparity

    % Find the difference matrix for the current disparity level. Expect
    % this to generate a Kernel function.
    coder.gpu.kernel;
    for colIdx=1:imgWidth+2*WIN_RAD
        coder.gpu.kernel;
        for rowIdx=1:imgHeight+2*WIN_RAD
            % Row index calculation.
            ind_h = rowIdx - WIN_RAD;

            % Column indices calculation for left image.
            ind_w1 = colIdx - WIN_RAD;

            % Row indices calculation for right image.
            ind_w2 = colIdx + d - WIN_RAD;

            % Border clamping for row Indices.
            if ind_h <= 0
                ind_h = 1;
            end
            if ind_h > imgHeight
                ind_h = imgHeight;
            end
        end
    end
end
```



```

% Border clamping for column indices for left image.
if ind_w1 <= 0
    ind_w1 = 1;
end
if ind_w1 > imgWidth
    ind_w1 = imgWidth;
end

% Border clamping for column indices for right image.
if ind_w2 <= 0
    ind_w2 = 1;
end
if ind_w2 > imgWidth
    ind_w2 = imgWidth;
end

% In this step, Sum of absolute Differences is performed
% across four channels.
tDiff = int32(0);
for chIdx = 1:nChannels
    tDiff = tDiff + abs(int32(img0((ind_h-1)*(nChannels)+...
        chIdx,ind_w1))-int32(img1((ind_h-1)*(nChannels)+...
        chIdx,ind_w2)));
end

% Store the SAD cost into a matrix.
diff_img(rowIdx,colIdx) = tDiff;
end
end

% Aggregating the differences using separable convolution. Expect this
% to generate two kernels using shared memory. The first kernel is the
% convolution with the horizontal kernel and second kernel operates on
% its output the column wise convolution.
cost_v = conv2(diff_img,filt_h,'valid');
cost = conv2(cost_v,filt_v,'valid');

% This part updates the min_cost matrix with by comparing the values
% with current disparity level.
for ll=1:imgWidth
    for kk=1:imgHeight
        % load the cost
        temp_cost = int32(cost(kk,ll));

        % Compare against the minimum cost available and store the
        % disparity value.
        if min_cost(kk,ll) > temp_cost
            min_cost(kk,ll) = temp_cost;
            out_disp(kk,ll) = abs(d) + 8;
        end
    end
end
end
end
end
end

```

Read Images and Pack Data Into RGBA Packed Column-Major Order

```
img0 = imread('scene_left.png');  
img1 = imread('scene_right.png');  
  
[imgRGB0] = pack_rgbData(img0);  
[imgRGB1] = pack_rgbData(img1);
```

Left Image



Right Image



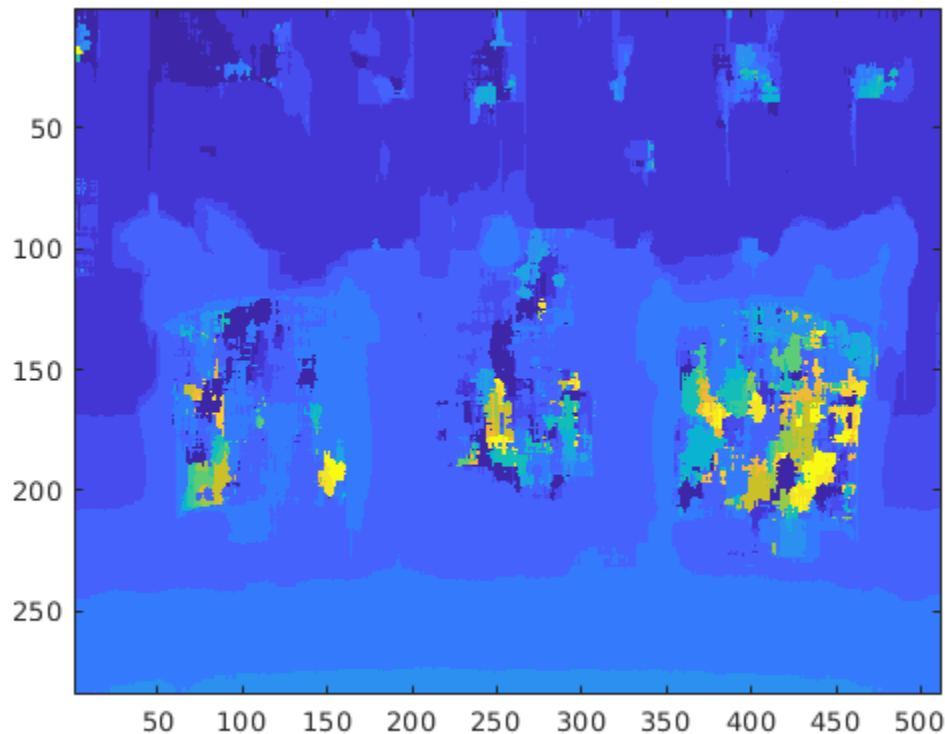
Generate GPU Code

```
cfg = coder.gpuConfig('mex');  
codegen -config cfg -args {imgRGB0, imgRGB1} stereoDisparity;
```

Code generation successful: [View report](#)

Run Generated MEX and Show the Output Disparity

```
out_disp = stereoDisparity_mex(imgRGB0,imgRGB1);
imagesc(out_disp);
```



Half Precision

Computations in this example can also be done in half-precision floating point numbers, using the `stereoDisparityHalfPrecision.m` entry-point function. To generate and execute code with half-precision data types, CUDA compute capability of 6.0 or higher is required. Set the `ComputeCapability` property of the code configuration object to `'6.0'`. For half-precision, the memory allocation (`malloc`) mode for generating CUDA code must be set to `'Discrete'`.

```
cfg.GpuConfig.ComputeCapability = '6.0';
cfg.GpuConfig.MallocMode = 'Discrete';
```

The standard `imread` command represents the RGB channels of an images with integers, one for each pixel. The integers range from 0 to 255. Simply casting inputs to half type might result in overflow during convolutions. In this case, we can scale the images to values between 0 and 1. "imread" represents the RGB channels of an images with integers, one for each pixel. The integers range from 0 to 255. Simply casting inputs to half type might result in overflow during convolutions. In this case, we can scale the images to values between 0 and 1.

```
img0 = imread('scene_left.png');
img1 = imread('scene_right.png');
```

```
[imgRGB0] = half(pack_rgbData(img0))/255;  
[imgRGB1] = half(pack_rgbData(img1))/255;
```

Generate CUDA MEX for the Function

Code generation on the `stereo_disparity_half_precision.m` function.

```
codegen -config cfg -args {imgRGB0, imgRGB1} stereoDisparityHalfPrecision;
```

Code generation successful: [View report](#)

See Also

Functions

[codegen](#) | [coder.gpu.kernel](#) | [coder.gpu.kernelfun](#) | [gpucoder.matrixMatrixKernel](#) | [coder.gpu.constantMemory](#) | [gpucoder.stencilKernel](#) | [coder.checkGpuInstall](#)

Objects

[coder.gpuConfig](#) | [coder.CodeConfig](#) | [coder.EmbeddedCodeConfig](#) | [coder.gpuEnvConfig](#)

Related Examples

- “Kernels from Library Calls” on page 2-8
- “Design Patterns” on page 2-26
- “Kernels from Scatter-Gather Type Operations” on page 2-4
- “Call Custom CUDA Device Function from the Generated Code” on page 2-22

Feature Extraction Using SURF

Object Recognition using Speeded-Up Robust Features (SURF) is composed of three steps: feature extraction, feature description, and feature matching. This example performs feature extraction, which is the first step of the SURF algorithm. The algorithm used here is based on the OpenSURF library implementation. This example shows how you can use GPU Coder™ to solve this compute intensive problem through CUDA® code generation.

Third-Party Prerequisites

Required

This example generates CUDA MEX and has the following third-party requirements.

- CUDA enabled NVIDIA® GPU and compatible driver.

Optional

For non-MEX builds such as static, dynamic libraries or executables, this example has the following additional requirements.

- NVIDIA toolkit.
- Environment variables for the compilers and libraries. For more information, see “Third-Party Hardware” and “Setting Up the Prerequisite Products”.

Verify GPU Environment

To verify that the compilers and libraries necessary for running this example are set up correctly, use the `coder.checkGpuInstall` function.

```
envCfg = coder.gpuEnvConfig('host');
envCfg.BasicCodegen = 1;
envCfg.Quiet = 1;
coder.checkGpuInstall(envCfg);
```

Feature Extraction

Feature extraction is a fundamental step in any object recognition algorithm. It refers to the process of extracting useful information referred to as *features* from an input image. The extracted features must be representative in nature, carrying important and unique attributes of the image.

The `SurfDetect.m` function is the main entry-point, that performs feature extraction. This function accepts an 8-bit RGB or an 8-bit grayscale image as the input. The output returned is an array of extracted interest points. This function is composed of the following function calls, which contain computations suitable for GPU parallelization:

- The `Convert32bitFPGray.m` function converts an 8-bit RGB image to an 8-bit grayscale image. If the input provided is already in the 8-bit grayscale format, skip this step. After this step, the 8-bit grayscale image is converted to a 32-bit floating-point representation for enabling fast computations on the GPU.
- The `MyIntegralImage.m` function calculates the integral image of the 32-bit floating-point grayscale image obtained in the previous step. The integral image is useful for simplifying finding the sum of pixels enclosed within any rectangular region of the image. Finding the sum of pixels helps in improving the speed of convolutions performed in the next step.

- The FastHessian.m function performs convolution of the image with box filters of different sizes and stores the computed responses. For this example, use these parameters:

```
Number of Octaves: 5  
Number of Intervals: 4  
Threshold: 0.0004
```

```
Filter Sizes: Octave 1 - 9, 15, 21, 27  
              Octave 2 - 15, 27, 39, 51  
              Octave 3 - 27, 51, 75, 99  
              Octave 4 - 51, 99, 147, 195  
              Octave 5 - 99, 195, 291, 387
```

- The NonMaxSuppression.m function performs non-maximal suppression to filter out only the useful interest points from the responses obtained earlier.
- The OrientationCalc.m function calculates and assigns orientation to the interest points in the previous step.

The final result is an array of interest points where an interest point is a structure that consists of these fields:

x, y (coordinates), scale, orientation, Laplacian

Read Input Image

Read an input image into MATLAB by using the imread function.

```
imageFile = 'peppers.png';  
inputImage = imread(imageFile);  
imshow(inputImage);
```



Generate CUDA MEX for the Function

To generate CUDA MEX for the SurfDetect function, create a GPU Coder configuration object, and then run the codegen function.

```
cfg = coder.gpuConfig('mex');  
codegen -config cfg SurfDetect -args {inputImage}
```

Code generation successful: [View report](#)

Run SURF Detection on MATLAB and GPU

Run the SurfDetect on MATLAB.

```
disp('Running SURF Detection on MATLAB...');
```

Running SURF Detection on MATLAB...

```
tic;  
interestPoints = SurfDetect(inputImage);  
execTime = toc;  
fprintf('Found %d SURF interest points in %f seconds.\n',length(interestPoints),execTime);
```

Found 249 SURF interest points in 7.777913 seconds.

Call the generated MEX function SurfDetect_mex to run on a GPU.

```
disp('Running GPU Coder SURF');
```

```
Running GPU Coder SURF
```

```
tic;
```

```
interestPointsGPU = SurfDetect_mex(inputImage);
```

```
execTime = toc;
```

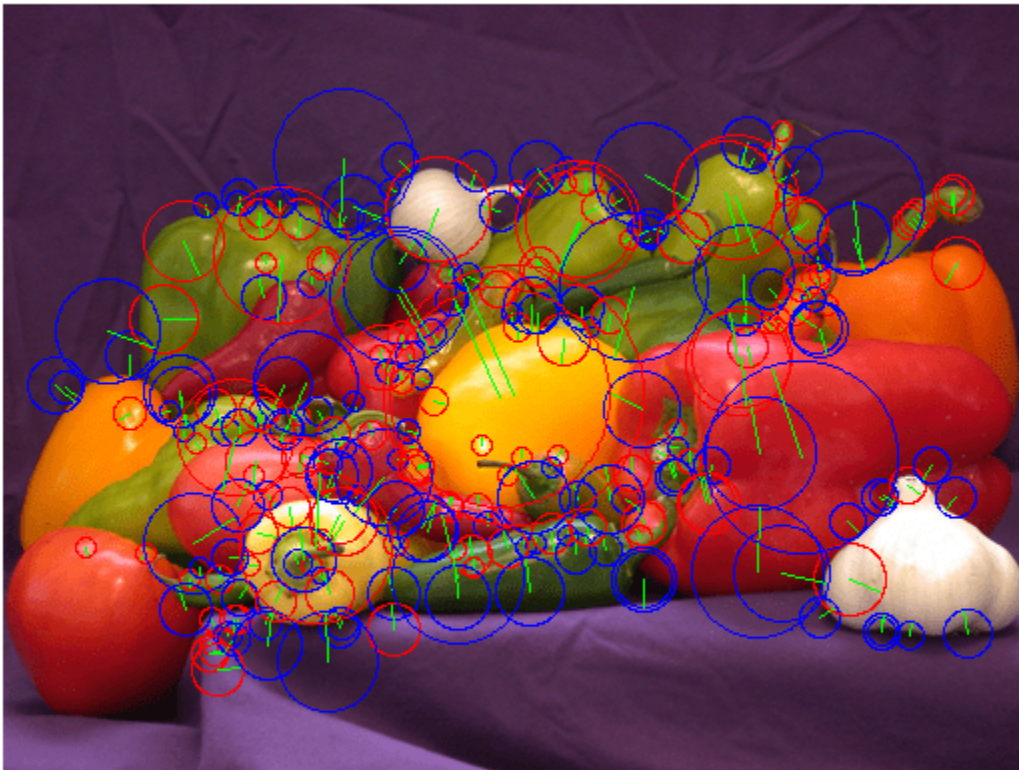
```
fprintf('GPU Coder SURF found %d interest points in %f seconds.\n',length(interestPointsGPU),execTime);
```

```
GPU Coder SURF found 249 interest points in 0.436629 seconds.
```

Depict the Extracted Interest Points

The output `interestPointsGPU` is an array of extracted interest points. These interest points are depicted over the input image in a figure window.

```
DrawIpoints(imageFile, interestPointsGPU);
```



References

[1] Notes on the OpenSURF Library by Christopher Evans.

[2] Bay, H., A. Ess, T. Tuytelaars, and L. Van Gool. "SURF:Speeded Up Robust Features." *Computer Vision and Image Understanding (CVIU)*. Vol. 110, No. 3, pp. 346-359, 2008.

See Also

Functions

`codegen` | `coder.gpu.kernel` | `coder.gpu.kernelfun` | `gpucoder.matrixMatrixKernel` | `coder.gpu.constantMemory` | `gpucoder.stencilKernel` | `coder.checkGpuInstall`

Objects

`coder.gpuConfig` | `coder.CodeConfig` | `coder.EmbeddedCodeConfig` | `coder.gpuEnvConfig`

Related Examples

- "Kernels from Library Calls" on page 2-8
- "Design Patterns" on page 2-26
- "Kernels from Scatter-Gather Type Operations" on page 2-4
- "Call Custom CUDA Device Function from the Generated Code" on page 2-22

Feature Matching

This example shows how to generate CUDA® MEX from MATLAB® code and perform feature matching between two images. This example uses the `matchFeatures` (Computer Vision Toolbox) function from the Image Processing Toolbox™ to match the feature descriptors between two images that are rotated and scaled with respect to each other. The feature descriptors of the two images are detected and extracted by using the Speeded-Up Robust Features (SURF) algorithm.

Third-Party Prerequisites

Required

This example generates CUDA MEX and has the following third-party requirements.

- CUDA enabled NVIDIA® GPU and compatible driver.

Optional

For non-MEX builds such as static, dynamic libraries or executables, this example has the following additional requirements.

- NVIDIA toolkit.
- Environment variables for the compilers and libraries. For more information, see “Third-Party Hardware” and “Setting Up the Prerequisite Products”.

Verify GPU Environment

To verify that the compilers and libraries necessary for running this example are set up correctly, use the `coder.checkGpuInstall` function.

```
envCfg = coder.gpuEnvConfig('host');  
envCfg.BasicCodegen = 1;  
envCfg.Quiet = 1;  
coder.checkGpuInstall(envCfg);
```

Feature Detection and Extraction

For this example, feature matching is performed on two images that are rotated and scaled with respect to each other. Before the two images can be matched, feature points for each image must be detected and extracted. The following `featureDetectionAndExtraction` function uses SURF (`detectSURFFeatures` (Computer Vision Toolbox)) local feature detector to detect the feature points and `extractFeatures` (Computer Vision Toolbox) to extract the features.

The function `featureDetectionAndExtraction` returns `refPoints`, which contains the feature coordinates of the reference image, `qryPoints`, containing feature coordinates of query image, `refDesc` matrix containing reference image feature descriptors and `qryDesc` matrix containing query image feature descriptors.

- `refPoints` = Reference image feature coordinates.
- `qryPoints` = Query image feature coordinates.
- `refDescFeat` = Reference image feature descriptors.
- `qryDescFeat` = Query image feature descriptors.

```

K = imread('cameraman.tif');
refImage = imresize(K,3);
scale = 0.7;
J = imresize(refImage,scale);
theta = 30.0;
qryImage = imrotate(J,theta);
[refPoints,refDescFeat,qryPoints,qryDescFeat] = featureDetectionAndExtraction(refImage,...
    qryImage);

```

The feature_matching Entry-Point Function

The `feature_matching` function takes feature points and feature descriptors extracted from two images and finds a match between them.

type `feature_matching`

```

function [matchedRefPoints,matchedQryPoints] = feature_matching(refPoints,...
    refDesc,qryPoints,qryDesc)
%#codegen
% Copyright 2018-2021 The MathWorks, Inc.

coder.gpu.kernelfun;
%% Feature Matching
[indexPairs,matchMetric] = matchFeatures(refDesc, qryDesc);
matchedRefPoints = refPoints(indexPairs(:,1),:);
matchedQryPoints = qryPoints(indexPairs(:,2),:);

```

Feature Matching Code Generation

Because the example runs on the host system, create a MEX-call configuration object with default parameters. To avoid abnormal termination of MATLAB if there are run-time errors in the generated code, select the safe-build option.

```

cfg = coder.gpuConfig;
cfg.GpuConfig.SafeBuild = 1;
inputs = {refPoints,refDescFeat,qryPoints,qryDescFeat};
codegen -config cfg -args inputs feature_matching

```

Code generation successful.

```

[matchedRefPoints_gpu,matchedQryPoints_gpu] = feature_matching_mex(refPoints,...
    refDescFeat,qryPoints,qryDescFeat);

```

Display Feature Matches

```

figure;
showMatchedFeatures(refImage, qryImage, matchedRefPoints_gpu, matchedQryPoints_gpu);
title('Putatively Matched Points (Including Outliers)');

```

Putatively Matched Points (Including Outliers)



See Also

Functions

[codegen](#) | [coder.gpu.kernel](#) | [coder.gpu.kernelfun](#) | [gpucoder.matrixMatrixKernel](#) | [coder.gpu.constantMemory](#) | [gpucoder.stencilKernel](#) | [coder.checkGpuInstall](#)

Objects

[coder.gpuConfig](#) | [coder.CodeConfig](#) | [coder.EmbeddedCodeConfig](#) | [coder.gpuEnvConfig](#)

Related Examples

- “Kernels from Library Calls” on page 2-8

- “Design Patterns” on page 2-26
- “Kernels from Scatter-Gather Type Operations” on page 2-4
- “Call Custom CUDA Device Function from the Generated Code” on page 2-22

Lane Detection on the GPU by Using the houghlines Function

This example shows how to generate CUDA® MEX for a MATLAB® function that can detect and output lane marker boundaries on an image. The example takes an RGB image as input and uses the `ordfilt2` (Image Processing Toolbox), `hough` (Image Processing Toolbox), `houghpeaks` (Image Processing Toolbox), and `houghlines` (Image Processing Toolbox) functions that are part of Image Processing Toolbox™ to produce the lane-detected output image.

Third-Party Prerequisites

Required

This example generates CUDA MEX and has the following third-party requirements.

- CUDA enabled NVIDIA® GPU and compatible driver.

Optional

For non-MEX builds such as static, dynamic libraries or executables, this example has the following additional requirements.

- NVIDIA toolkit.
- Environment variables for the compilers and libraries. For more information, see “Third-Party Hardware” and “Setting Up the Prerequisite Products”.

Verify GPU Environment

To verify that the compilers and libraries necessary for running this example are set up correctly, use the `coder.checkGpuInstall` function.

```
envCfg = coder.gpuEnvConfig('host');
envCfg.BasicCodegen = 1;
envCfg.Quiet = 1;
coder.checkGpuInstall(envCfg);
```

The lane_detection_houghlines Entry-Point Function

The `lane_detection_houghlines.m` entry-point function takes an intensity image as input and returns the lane-detected image.

```
type lane_detection_houghlines

function [lines] = lane_detection_houghlines(inputImage)%#codegen

% Copyright 2019-2021 The MathWorks, Inc.
coder.gpu.kernelfun;

% Convert RGB image to grayscale image.
grayImage = im2gray(inputImage);

% Edge detection using ordfilt2.
input = grayImage(240:end,1:end);
dom = ones(2);
minOrder = 1;
maxOrder = 4;
```

```

padopt = 'zeros';

MinImg = ordfilt2(input,minOrder,dom,padopt);
MaxImg = ordfilt2(input,maxOrder,dom,padopt);

% Edge detected output.
outImage = MaxImg - MinImg;
BW = imbinarize(outImage);

[H,T,R] = hough(BW);
P = houghpeaks(H,20,'threshold',1);
lines = houghlines(BW,T,R,P,'FillGap',200,'MinLength',150);

```

Generate CUDA MEX for the lane_detection_houghlines Function

Create a GPU code configuration object and run the codegen function.

```

inputImage = imread('highway.png');
inputResizedImage = imresize(inputImage,[480 640]);
cfg = coder.gpuConfig('mex');
codegen -args {inputResizedImage} -config cfg lane_detection_houghlines

```

Code generation successful.

Run the Generated CUDA MEX

Run the generated lane_detection_houghlines_mex with an input image and plot the input and lane-detected images.

```

[lines] = lane_detection_houghlines_mex(inputResizedImage);

% Plot images.
inputImageVGASize = imresize(inputImage,[480 640]);
outputImage = imresize(inputImage,[480 640]);
p1 = subplot(1, 2, 1);
p2 = subplot(1, 2, 2);
imshow(inputImageVGASize, 'Parent', p1);
imshow(outputImage, 'Parent', p2);hold on
max_len = 0;
for k = 1:length(lines)
    if ((lines(k).theta <= 60 && lines(k).theta >10)||...
        (lines(k).theta <= -10 && lines(k).theta > -50) )
        xy = [lines(k).point1; (lines(k).point2)];
        plot(xy(:,1),xy(:,2)+240,'LineWidth',2,'Color','green');

        % Plot beginning and end of lines.
        plot(xy(1,1),xy(1,2)+240,'x','LineWidth',2,'Color','yellow');
        plot(xy(2,1),xy(2,2)+240,'x','LineWidth',2,'Color','red');

        % Determine the endpoints of the longest line segment.
        len = norm(lines(k).point1 - lines(k).point2);
        if ( len > max_len)
            max_len = len;
            xy_long = xy;
        end
    end
end
title(p1, 'Input Image');
title(p2, 'Lane Detected Output Image');

```



See Also

Functions

[codegen](#) | [coder.gpu.kernel](#) | [coder.gpu.kerndfun](#) | [gpuCoder.matrixMatrixKernel](#) | [coder.gpu.constantMemory](#) | [gpuCoder.stencilKernel](#) | [coder.checkGpuInstall](#)

Objects

[coder.gpuConfig](#) | [coder.CodeConfig](#) | [coder.EmbeddedCodeConfig](#) | [coder.gpuEnvConfig](#)

Related Examples

- “Kernels from Library Calls” on page 2-8
- “Design Patterns” on page 2-26
- “Kernels from Scatter-Gather Type Operations” on page 2-4
- “Call Custom CUDA Device Function from the Generated Code” on page 2-22

Edge Detection with Sobel Method in Half-Precision

This example demonstrates edge detection in an image with a CUDA® MEX function generated from a MATLAB® function. The edge detection algorithm is implemented with half-precision data type.

Third-Party Prerequisites

Required

This example generates CUDA MEX and has the following third-party requirements.

- CUDA enabled NVIDIA® GPU with a minimum compute capability of 6.0 and compatible driver.

Optional

For non-MEX builds such as static, dynamic libraries or executables, this example has the following additional requirements.

- NVIDIA toolkit.
- Environment variables for the compilers and libraries. For more information, see “Third-Party Hardware” and “Setting Up the Prerequisite Products”.

Verify GPU Environment

To verify that the compilers and libraries necessary for running this example are set up correctly, use the `coder.checkGpuInstall` function.

```
envCfg = coder.gpuEnvConfig('host');
envCfg.BasicCodegen = 1;
envCfg.Quiet = 1;
coder.checkGpuInstall(envCfg);
```

Sobel Edge Detection Algorithm

In the Sobel edge detection algorithm `sobelEdgeDetectionAlg.m`, a 2-D spatial gradient operation is performed on a gray scale image. This operation emphasizes the high spatial frequency regions that correspond to the edges in the image.

type `sobelEdgeDetectionAlg`

```
function edgeImg = sobelEdgeDetectionAlg(img,thresh) %#codegen
% Sobel Edge Detection Example MATLAB function for edge detection.
% Copyright 2018 The MathWorks, Inc.

kern = half([1 2 1; 0 0 0; -1 -2 -1]);

% Finding horizontal and vertical gradients.
h = conv2(img(:,:,2),kern,'same');
v = conv2(img(:,:,2),kern','same');

% Finding magnitude of the gradients.
e = sqrt(h.*h + v.*v);

% Threshold the edges
```

```
edgeImg = uint8((e > thresh) * 240);  
end
```

The Sobel edge algorithm computes the horizontal gradient `resX` and the vertical gradient `resY` of the input image by using two orthogonal filter kernels `maskX` and `maskY`. After the filtering operation, the algorithm computes the gradient magnitude and applies a threshold to find the regions of the images that are considered to be edges.

Read Images and Pack Data Into RGBA Packed Column Major Order

Use the standard `imread` command to read the images. `imread` represents the RGB channels of an images with integers, one for each pixel. The integers range from 0 to 255. Simply casting inputs to half type might result in overflow during convolutions. In this case, we can scale the images to values between 0 and 1.

```
im = imread('peppers.png');  
figure();  
image(im);  
imPacked = half(im)/255;  
thresh = half(100)/255;
```



Generate CUDA MEX for the Function

To generate CUDA MEX for the `sobelEdgeDetectionAlg` function, create a GPU code configuration object and run the `codegen` command. To generate and execute code with half-precision data types, CUDA compute capability of 6.0 or higher is required. Set the

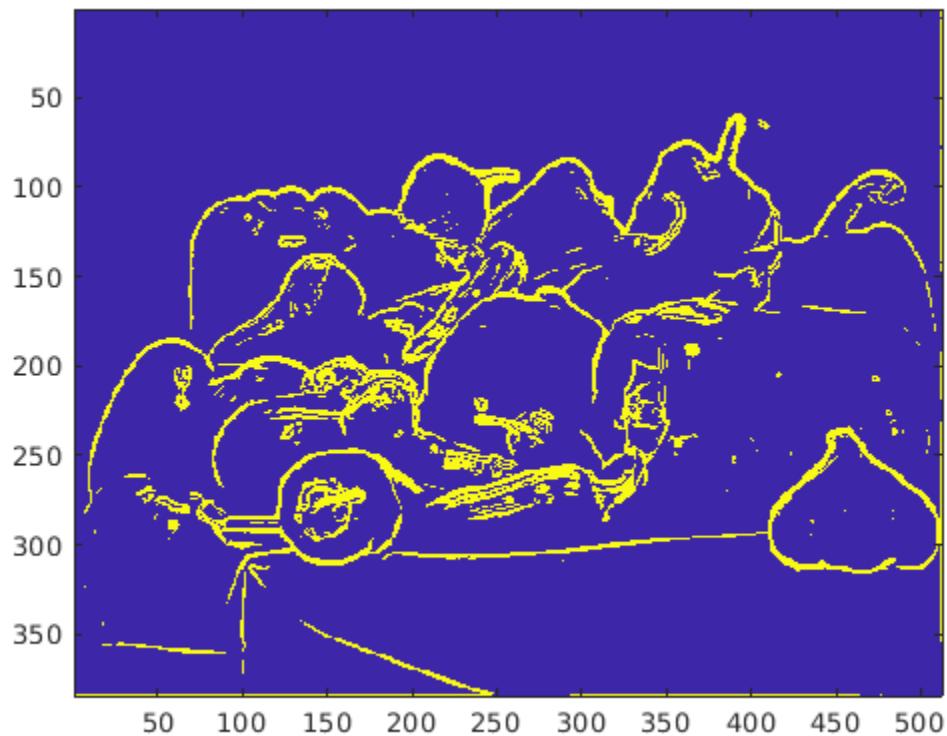
ComputeCapability property of the code configuration object to '6.0'. For half-precision, the memory allocation (malloc) mode for generating CUDA code must be set to 'Discrete'.

```
cfg = coder.gpuConfig('mex');  
cfg.GpuConfig.ComputeCapability = '6.0';  
cfg.GpuConfig.MallocMode = 'Discrete';  
  
codegen -config cfg -args {imPacked,thresh} sobelEdgeDetectionAlg;  
  
Code generation successful.
```

Run the MEX Function

After you generate a MEX function, you can verify that it has the same functionality as the original MATLAB entry-point function. Run the generated `sobelEdgeDetectionAlg_mex` and plot the results.

```
out_disp = sobelEdgeDetectionAlg_mex(imPacked,thresh);  
imagesc(out_disp);
```



Clear MEX Memory.

Clear the static network object that was loaded in memory.

```
clear mex;
```

See Also

Functions

[codegen](#) | [coder.gpu.kernel](#) | [coder.gpu.kernelfun](#) | [gpucoder.matrixMatrixKernel](#) | [coder.gpu.constantMemory](#) | [gpucoder.stencilKernel](#) | [coder.checkGpuInstall](#)

Objects

[coder.gpuConfig](#) | [coder.CodeConfig](#) | [coder.EmbeddedCodeConfig](#) | [coder.gpuEnvConfig](#)

Related Examples

- “Kernels from Library Calls” on page 2-8
- “Design Patterns” on page 2-26
- “Kernels from Scatter-Gather Type Operations” on page 2-4
- “Call Custom CUDA Device Function from the Generated Code” on page 2-22

Build a Map from Lidar Data using SLAM on GPU

This example shows how to perform 3-D Lidar simultaneous localization and mapping (SLAM) on Nvidia GPU.

This example uses 3-D lidar data from a vehicle mounted sensor to progressively build a map and estimate the trajectory of the vehicle by using the SLAM approach. This example is based on the Build a Map from Lidar Data Using SLAM example. For more information, see “Build a Map from Lidar Data Using SLAM” (Computer Vision Toolbox).

Load Recorded Data

The data used in this example is part of the Velodyne SLAM Dataset [1], and represents close to 6 minutes of recorded data. Download the data to a temporary directory. The dataset size is 153 MB. This download can take a few minutes.

```
baseDownloadURL = 'https://www.mrt.kit.edu/z/publ/download/velodyneslam/data/scenario1.zip';
dataFolder      = fullfile(tempdir, 'kit_velodyneslam_data_scenario1', filesep);
options         = weboptions('Timeout', Inf);

zipFileName     = dataFolder + "scenario1.zip";

% Get the full file path to the PNG files in the scenario1 folder.
pointCloudFilePattern = fullfile(dataFolder, 'scenario1', 'scan*.png');
numExpectedFiles     = 2513;

folderExists = exist(dataFolder, 'dir');
if ~folderExists
    % Create a folder in a temporary directory to save the downloaded zip
    % file.
    mkdir(dataFolder);

    disp('Downloading scenario1.zip (153 MB) ...')
    websave(zipFileName, baseDownloadURL, options);

    % Unzip downloaded file
    unzip(zipFileName, dataFolder);
elseif folderExists && numel(dir(pointCloudFilePattern)) < numExpectedFiles
    % Redownload the data if it got reduced in the temporary directory.
    disp('Downloading scenario1.zip (153 MB) ...')
    websave(zipFileName, baseDownloadURL, options);

    % Unzip downloaded file.
    unzip(zipFileName, dataFolder)
end
```

Use the `helperReadDataset` function to read data from the created folder in the form of a timetable. The point clouds captured by the lidar are stored in the form of PNG image files. Extract the list of point cloud file names in the `pointCloudTable` variable.

```
datasetTable = helperReadDataset(dataFolder, pointCloudFilePattern);

pointCloudTable = datasetTable(:, 1);
insDataTable     = datasetTable(:, 2:end);
```

To pass the point cloud data to entry-point function, copy the data from the point clouds into matrix. To read the point cloud data from the image file, use the `helperReadPointCloudFromFile` function. This function takes an image file name and returns a `PointCloud` object. The size of every point cloud is 64-by-870-by-3 and there are 2513 point clouds. The size of matrix is 64-by-670-by-3-by-2513.

```
pointCloudCount = height(pointCloudTable);
numColumns = 64;
numRows = 870;
location = zeros(numColumns, numRows, 3, 'single');
for idx = 1 : pointCloudCount
    filename = pointCloudTable.PointCloudFileName{idx};
    ptCloud = helperReadPointCloudFromFile(filename);
    location(:,:,,idx) = ptCloud.Location;
end
```

Build a Map Using Odometry

Use the approach explained in the “Build a Map from Lidar Data Using SLAM” (Computer Vision Toolbox) example to build a map. The approach consists of the following steps:

- Align lidar scans: Align successive lidar scans using a point cloud registration technique. This example uses `pregisterndt` for registering scans. By successively composing these transformations, each point cloud is transformed back to the reference frame of the first point cloud.
- Combine aligned scans: Generate a map by combining all the transformed point clouds.

This approach of incrementally building a map and estimating the trajectory of the vehicle is called odometry.

Examine Entry-Point Function

`ndtSLAM` is the entry-point function for GPU code generation. `ndtSLAM` takes locations of point clouds and INS data as input. Inside the for-loop, it registers two consecutive sets of point clouds in a single iteration. The first two pointclouds are registered before the for-loop, and the last two point clouds are registered after the for-loop, if required.

type `ndtSLAM.m`

```
function absTformOut = ndtSLAM(locations, insDataTable)
% ndtSLAM register multiple pointclouds and returns the absolute
% transformation for each of the pointcloud. locations is matrix of
% location of every pointcloud with size N x R x C x 3, where N is
% number of pointcloud, and R x C x 3 is size of individual pointcloud.
% insDataTable is a table of INS data. absTformOut returns transformations
% as a matrix shaped N x 4 x 4.

% Set random seed to ensure reproducibility
rng(0);

% Initialize point cloud processing parameters
gridSize = coder.const(0.8);

% Initialize transformations
absTform = rigidtform3d(eye(4, 'single')); % Absolute transformation to reference frame
relTform = rigidtform3d(eye(4, 'single')); % Relative transformation between successive scans
```

```

skipFrames = coder.const(5);
numFrames = size(locations, 4);

% allocate output variables
numTransforms = ceil(numFrames / skipFrames);
absTformOut = coder.nullcopy(zeros(4,4,numTransforms, 'single'));
outIdx = 1;

% If input locations are empty, return
if isempty(locations)
    return;
end

% Read point cloud
ptCloudFirstOrig = pointCloud(locations(:,:,1));

% Process point cloud
% - Segment and remove ground plane
% - Segment and remove ego vehicle
ptCloudFirst = helperProcessPointCloud(ptCloudFirstOrig, "rangefloodfill");

% Downsample the processed point cloud
ptCloudFirst = pcdownsampling(ptCloudFirst, "gridAverage", gridSize);

% Add first point cloud scan as a view to the view set
absTformOut(:,:,outIdx) = absTform.A;
outIdx = outIdx + 1;

ptCloudPrev = ptCloudFirst;

for n = 1 + skipFrames : skipFrames + skipFrames : numFrames - skipFrames
    % If locations are empty skip present iteration and continue to next
    % iteration.
    if isempty(locations(:,:,n)) || isempty(locations(:,:,n + skipFrames))
        continue;
    end
    %% even iteration
    % Read point cloud
    ptCloudOrig = pointCloud(locations(:,:,n));
    insData = insDataTable(n - skipFrames: n, :);
    [absTform, relTform, ptCloudPrev] = processFrame(ptCloudOrig, ptCloudPrev, ...
        insData, gridSize, relTform, absTform);

    % update output
    absTformOut(:,:,outIdx) = absTform.A;
    outIdx = outIdx + 1;

    %% odd iteration
    % Read point cloud
    ptCloudOrig = pointCloud(locations(:,:,n + skipFrames));
    insData = insDataTable(n: n + skipFrames, :);
    [absTform, relTform, ptCloudPrev] = processFrame(ptCloudOrig, ptCloudPrev, ...
        insData, gridSize, relTform, absTform);

    % update output
    absTformOut(:,:,outIdx) = absTform.A;
    outIdx = outIdx + 1;
end
end

```

```
if mod(numTransforms, 2) == 0
    % last even iteration, if required.
    ptIdx = 1 + skipFrames * (numTransforms - 1);
    % Read point cloud
    ptCloudOrig = pointCloud(locations(:,:,:),ptIdx));
    ptCloudPrev = pointCloud(locations(:,:,:),ptIdx - skipFrames));
    insData = insDataTable(ptIdx-skipFrames:ptIdx, :);
    [absTform, ~, ~] = processFrame(ptCloudOrig, ptCloudPrev, ...
        insData, gridSize, relTform, absTform);
    % update output
    absTformOut(:,:,outIdx) = absTform.A;
end
end
```

`processFrame` performs the processing and registration of two point clouds. `processFrame` is called by `ndtSLAM`.

type `processFrame.m`

```
function [absTform, relTform, ptCloudPrev] = processFrame(ptCloudOrig, ptCloudPrev, ...
    insData, gridSize, relTform, absTform)
% processFrame Process and register two pointclouds and return the
% transformations.

regGridSize = coder.const(2.5);

% Process point cloud
% - Segment and remove ground plane
% - Segment and remove ego vehicle
ptCloud = helperProcessPointCloud(ptCloudOrig, "rangefloodfill");

% Downsample the processed point cloud
moving = pcdownsampling(ptCloud, 'gridAverage', gridSize);

% Use INS to estimate an initial transformation for registration
initTform = helperComputeInitialEstimateFromINS(relTform, insData);

% Compute rigid transformation that registers current point cloud with
% previous point cloud
relTform = pcregisterndt(moving, ptCloudPrev, regGridSize, ...
    "InitialTransform", initTform);

% Update absolute transformation to reference frame (first point cloud)
absTform = rigidtransform3d(absTform.A * relTform.A);

% update prev point cloud
ptCloudPrev = moving;
end
```

`helperProcessPointCloud` processes a `pointCloud` object by removing points belonging to the ground plane and the ego vehicle.

type `helperProcessPointCloud.m`

```
function ptCloud = helperProcessPointCloud(ptCloudIn, method)
%helperProcessPointCloud Process pointCloud to remove ground and ego vehicle
% ptCloud = helperProcessPointCloud(ptCloudIn, method) processes
```



```

% ptCloudIn by removing the ground plane and the ego vehicle.
% method can be "planeFit" or "rangeFloodfill".
%
% See also pcfiteplane, pointCloud/findNeighborsInRadius.

isOrganized = ~ismatrix(ptCloudIn.Location);

if (method=="rangeFloodfill" && isOrganized)
    elevationAngleDelta = coder.const(11);
    % Segment ground using floodfill on range image
    groundFixedIdx = segmentGroundFromLidarData(ptCloudIn, ...
        "ElevationAngleDelta", elevationAngleDelta);
else
    % Segment ground as the dominant plane with reference normal
    % vector pointing in positive z-direction
    maxDistance = 0.4;
    maxAngularDistance = 5;
    referenceVector = [0 0 1];

    [~, groundFixedIdx] = pcfiteplane(ptCloudIn, maxDistance, ...
        referenceVector, maxAngularDistance);
end

if isOrganized
    groundFixed = false(size(ptCloudIn.Location,1),size(ptCloudIn.Location,2));
else
    groundFixed = false(ptCloudIn.Count, 1);
end
groundFixed(groundFixedIdx) = true;

% Segment ego vehicle as points within a given radius of sensor
sensorLocation = coder.const([0 0 0]);
radius = coder.const(3.5);
egoFixedIdx = findNeighborsInRadius(ptCloudIn, sensorLocation, radius);

if isOrganized
    egoFixed = false(size(ptCloudIn.Location,1),size(ptCloudIn.Location,2));
else
    egoFixed = false(ptCloudIn.Count, 1);
end
egoFixed(egoFixedIdx) = true;

% Retain subset of point cloud without ground and ego vehicle
if isOrganized
    indices = ~groundFixed & ~egoFixed;
else
    indices = find(~groundFixed & ~egoFixed);
end

ptCloud = select(ptCloudIn, indices);
end

```

helperComputeInitialEstimateFromINS computes initial transformation estimate from INS data.

type [helperComputeInitialEstimateFromINS.m](#)

```
function initTform = helperComputeInitialEstimateFromINS(initTform, insData)
% helperComputeInitialEstimateFromINS Compute estimate for transformation
% from INS data.

% If no INS readings are available, return
if isempty(insData)
    return;
end

% The INS readings are provided with X pointing to the front, Y to the left
% and Z up. Translation below accounts for transformation into the lidar
% frame.
insToLidarOffset = [0 -0.79 -1.73]; % See DATAFORMAT.txt
tNow = [-insData.Y(end), insData.X(end), insData.Z(end)].' + insToLidarOffset';
tBefore = [-insData.Y(1) , insData.X(1) , insData.Z(1)].' + insToLidarOffset';

% Since the vehicle is expected to move along the ground, changes in roll
% and pitch are minimal. Ignore changes in roll and pitch, use heading only.
Rnow = rotmat(Quaternion([insData.Heading(end) 0 0], 'euler', 'ZYX', 'point'), 'point');
Rbef = rotmat(Quaternion([insData.Heading(1) 0 0], 'euler', 'ZYX', 'point'), 'point');

tformMatrix = [Rbef tBefore;0 0 0 1] \ [Rnow tNow;0 0 0 1];

initTform = rigidtransform3d(cast(tformMatrix, 'like', initTform.A));
end
```

Generate CUDA mex

Generate CUDA mex for the entry-point function(ndtSLAM). To improve performance,

- 1 Enable Memory Manager
- 2 Set the compute capability to the highest supported by the GPU on the system.
- 3 Increase stack limit per thread. This example uses max integer value. Use lower value if this gives an error.

```
config = coder.gpuConfig();
config.GpuConfig.EnableMemoryManager = true;
config.GpuConfig.ComputeCapability = gpuDevice().ComputeCapability;
config.GpuConfig.StackLimitPerThread = intmax;
```

```
codegen -config config -args {location, insDataTable} ndtSLAM
```

Code generation successful: [View report](#)

Plot Map

ndtSLAM function returns the absolute transformation for each of the frame that is used to build the map. To plot the map convert the transformation matrix into rigidtransform3d object and add the point clouds and the rigidtransform3d objects into pcviewset object.

The view set object viewset, now holds views and connections. In the Views table of viewset, the AbsolutePose variable specifies the absolute pose of each view with respect to the first view. Now, build a point cloud map using the created viewset. Align the view absolute poses with the point clouds in the viewset using pcalign. Specify a grid size to control the resolution of the map. The map is returned as a pointCloud object.

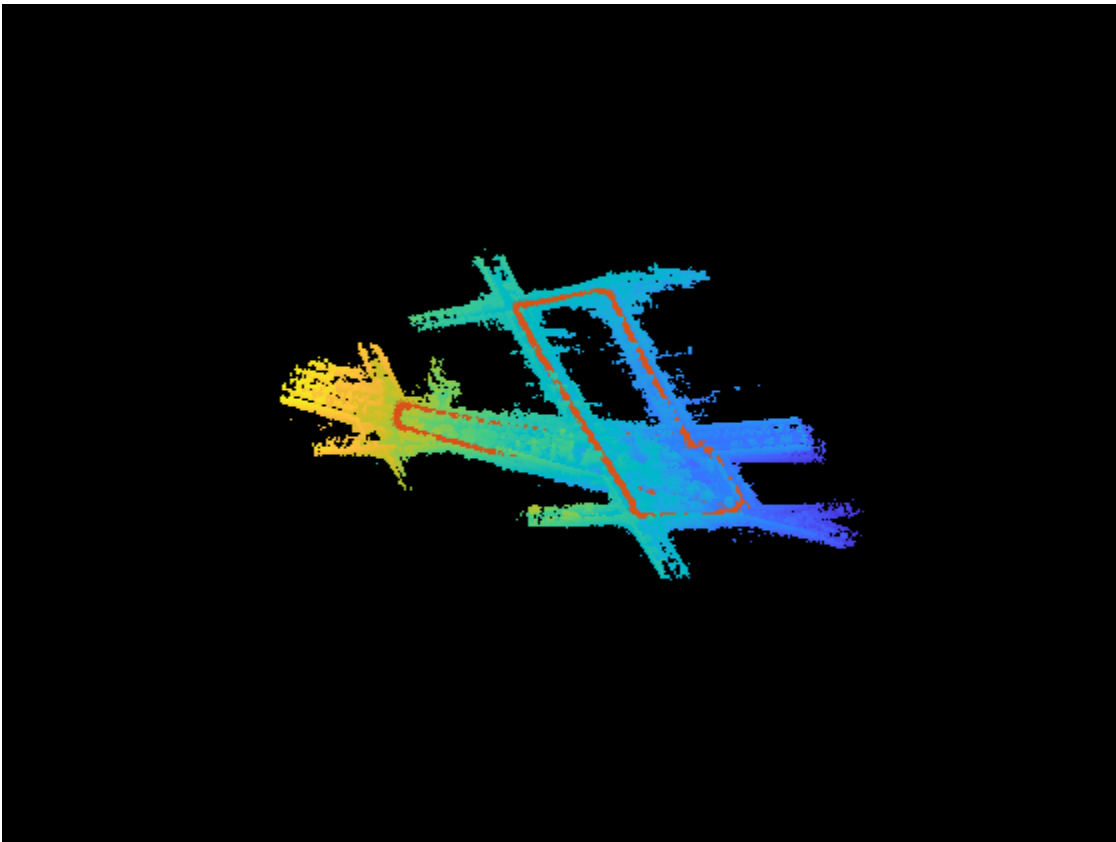
```
tforms = ndtSLAM_mex(location, insDataTable);
% Add results into pcviewset
```

```
viewset = pcviewset();
skipFrames = 5;
viewId = 1;
for idx = 1: skipFrames: 2513
    ptCloud = pointCloud(location(:,:,:),idx);
    absTforms = rigidtf3d(tforms(:,:),viewId);
    viewset = addView(viewset, viewId, absTforms, "PointCloud", ptCloud);
    if viewId > 1
        viewset = addConnection(viewset, viewId-1, viewId);
    end
    viewId = viewId + 1;
end

% plot result
ptClouds = viewset.Views.PointCloud;
absPoses = viewset.Views.AbsolutePose;
mapGridSize = 0.2;
ptCloudMap = pcalign(ptClouds, absPoses, mapGridSize);

hFigAfter = figure('Name', 'GPU SLAM');
hAxAfter = axes(hFigAfter);
pcshow(ptCloudMap, 'Parent', hAxAfter);

% Overlay view set display
hold on
plot(viewset, 'Parent', hAxAfter);
```



```
helperMakeFigurePublishFriendly(hFigAfter);
```

References

- 1 Moosmann, Frank, and Christoph Stiller. "Velodyne SLAM." *Proceedings of the IEEE Intelligent Vehicles Symposium*, 2011, pp. 393-98, http://www.mrt.kit.edu/z/publ/download/Moosmann_IV11.pdf.

Supporting Functions

convertFromSphericalToCartesianCoordinates converts coordinates from spherical to cartesian system.

```
function xyzData = convertFromSphericalToCartesianCoordinates(rangeData)
```

```
xyzData = zeros(size(rangeData), 'like', rangeData);
```

```
range = rangeData(:, :, 1);
```

```
pitch = rangeData(:, :, 2);
```

```
yaw = rangeData(:, :, 3);
```

```
xyzData(:, :, 1) = range .* cos(pitch) .* sin(yaw);
```

```
xyzData(:, :, 2) = range .* cos(pitch) .* cos(yaw);
```

```
xyzData(:, :, 3) = -range .* sin(pitch);
```

```
end
```

helperReadPointCloudFromFile reads pointcloud from PNG image file and returns a point cloud object.

```
function ptCloud = helperReadPointCloudFromFile(fileName)
```

```
%helperReadPointCloudFromFile Read pointCloud from PNG image file
```

```
%
```

```
% This is an example helper class that is subject to change or removal in  
% future releases.
```

```
%
```

```
% ptCloud = helperReadPointCloudFromFile(fileName) reads point cloud  
% data from the .png image file fileName and returns a pointCloud object.  
% This function expects file to be from the Velodyne SLAM Dataset.
```

```
% Copyright 2019-2022 The MathWorks, Inc.
```

```
% From DATAFORMAT.txt
```

```
% -----
```

```
% Each 360° revolution of the Velodyne scanner was stored as 16bit png  
% distance image (scan*.png). The scanner turned clockwise, filling the  
% image from the leftmost to the rightmost column, with the leftmost and  
% rightmost column being at the back of the vehicle. Note that measurements  
% were not corrected for vehicle movement. Thus and due to the physical  
% setup of the laser diodes, some strange effects can be seen at the cut of  
% the image when the vehicle is turning. As consequence, it is best to  
% ignore the 10 leftmost and rightmost columns of the image. To convert the  
% pixel values [0..65535] into meters, just divide by 500. This results in  
% an effective range of [0..131m]. Invalid measurements are indicated by  
% zero distance.
```

```
% To convert the distance values into 3D coordinates, use the setup in  
% "img.cfg". The yaw angles (counter-clockwise) are a linear mapping from
```

```

% the image column [0..869]->[180°..-180°] The pitch angles are specified
% for each image row separately.

validateattributes(fileName, {'char','string'}, {'scalartext'}, mfilename, 'fileName');

% Convert pixel values to range
range = single(imread(fileName)) ./ 500;
range(range==0) = NaN;

% Get yaw angles as a linear mapping of [0..869] -> [180 to -180]. Yaw and
% pitch values are obtained from img.cfg file.
yawAngles = 869 : -1 : 0;
yawAngles = -180 + yawAngles .* (360 / 869);

pitchAngles = [-1.9367; -1.57397; -1.30476; -0.871566; -0.57881; -0.180617; ...
    0.088762; 0.451829; 0.80315; 1.20124; 1.49388; 1.83324; 2.20757; ...
    2.54663; 2.87384; 3.23588; 3.53933; 3.93585; 4.21552; 4.5881; 4.91379; ...
    5.25078; 5.6106; 5.9584; 6.32889; 6.67575; 6.99904; 7.28731; 7.67877; ...
    8.05803; 8.31047; 8.71141; 9.02602; 9.57351; 10.0625; 10.4707; 10.9569; ...
    11.599; 12.115; 12.5621; 13.041; 13.4848; 14.0483; 14.5981; 15.1887; ...
    15.6567; 16.1766; 16.554; 17.1868; 17.7304; 18.3234; 18.7971; 19.3202; ...
    19.7364; 20.2226; 20.7877; 21.3181; 21.9355; 22.4376; 22.8566; 23.3224; ...
    23.971; 24.5066; 24.9992];

[yaw,pitch] = meshgrid( deg2rad(yawAngles), deg2rad(pitchAngles));
rangeData = cat(3, range, pitch, yaw);

xyzData = convertFromSphericalToCartesianCoordinates(rangeData);

% Transform points so that coordinate system faces towards the front of the
% vehicle.
ptCloud = pointCloud(xyzData.*cat(3,-1,1,1));
end

helperReadINSConfigFile reads INS configuration file and returns the data as a table.

function T = helperReadINSConfigFile(fileName)
%helperReadINSConfigFile Reads INS configuration file
%
% This is an example helper class that is subject to change or removal in
% future releases.
%
% T = helperReadINSConfigFile(fileName) reads the .cfg configuration file
% containing INS data, and returns it in a table. This function expects
% data from the Velodyne SLAM Dataset.
%
% See also timetable, readtable.

validateattributes(fileName, {'char','string'}, {'scalartext'}, mfilename, 'fileName');

% Create options to read delimited text file
opts = delimitedTextImportOptions;
opts.Delimiter = ";";
opts.DataLines = [8 inf];
opts.VariableNames = [...
    "Timestamps", ...
    "Num_Satellites", "Latitude", "Longitude", "Altitude", ...
    "Heading", "Pitch", "Roll", ...

```

```

        "Omega_Heading", "Omega_Pitch", "Omega_Roll", ...
        "V_X", "V_Y", "V_ZDown", ...
        "X", "Y", "Z"];
opts.VariableTypes(2:end) = {'double'};

T = readtable(fileName, opts);

% Remove unnecessary column
T.ExtraVar1 = [];

% Convert timestamps to datetime
T.Timestamps = datetime(T.Timestamps, 'InputFormat', 'yyyy-MM-dd HH:mm:ss.SSS');
T = table2timetable(T);
end

```

helperReadDataset reads velodyne SLAM dataset data into a timetable

```

function datasetTable = helperReadDataset(dataFolder, pointCloudFilePattern)
%helperReadDataset Read Velodyne SLAM Dataset data into a timetable
% datasetTable = helperReadDataset(dataFolder) reads data from the
% folder specified in dataFolder into a timetable. The function
% expects data from the Velodyne SLAM Dataset.
%
% See also fileDatastore, helperReadINSConfigFile.

% Create a file datastore to read in files in the right order
fileDS = fileDatastore(pointCloudFilePattern, 'ReadFcn', ...
    @helperReadPointCloudFromFile);

% Extract the file list from the datastore
pointCloudFiles = fileDS.Files;

imuConfigFile = fullfile(dataFolder, 'scenario1', 'imu.cfg');
insDataTable = helperReadINSConfigFile(imuConfigFile);

% Delete the bad row from the INS config file
insDataTable(1447, :) = [];

% Remove columns that will not be used
datasetTable = removevars(insDataTable, ...
    {'Num_Satellites', 'Latitude', 'Longitude', 'Altitude', 'Omega_Heading', ...
    'Omega_Pitch', 'Omega_Roll', 'V_X', 'V_Y', 'V_ZDown'});

datasetTable = addvars(datasetTable, pointCloudFiles, 'Before', 1, ...
    'NewVariableNames', "PointCloudFileName");
end

```

helperMakeFigurePublishFriendly adjusts figures so that screenshot captured by publish is correct.

```

function helperMakeFigurePublishFriendly(figure)
if ~isempty(figure) && isvalid(figure)
    figure.HandleVisibility = 'callback';
end
end

```

Kernel Creation from Simulink Models

- “Simulation Acceleration by Using GPU Coder” on page 3-2
- “Code Generation from Simulink Models with GPU Coder” on page 3-8
- “GPU Code Generation for Deep Learning Networks Using MATLAB Function Block” on page 3-14
- “GPU Code Generation for Blocks from the Deep Neural Networks Library” on page 3-22
- “Targeting NVIDIA Embedded Boards” on page 3-30
- “Numerical Equivalence Testing” on page 3-32
- “Parameter Tuning and Signal Monitoring by Using External Mode” on page 3-38
- “GPU Code Generation for Lane Detection in Simulink” on page 3-43
- “GPU Code Generation for a Fog Rectification Simulink Model” on page 3-48
- “Code Generation for a Deep Learning Simulink Model to Classify ECG Signals” on page 3-53
- “Code Generation for a Deep Learning Simulink Model that Performs Lane and Vehicle Detection” on page 3-60

Simulation Acceleration by Using GPU Coder

You can use GPU Coder to speed up the execution of your Simulink model on NVIDIA GPUs. GPU-accelerated computing follows a heterogeneous programming model. Highly parallelizable portions of the application are mapped into kernels that execute on thousands of GPU cores in parallel, while the remainder of the sequential code still runs on the CPU.

To perform GPU-accelerated simulation, model the compute intensive portions of your application in Simulink by using MATLAB Function blocks. When you simulate a model that contains a MATLAB Function block, the software partitions and generates CUDA MATLAB executable (MEX) code and integrates this code with the Simulink model.

The basic steps for simulation acceleration by using GPU Coder are:

- Create or open a model.
- Configure the model for GPU acceleration by selecting the **Solver**, **Language**, and other GPU-specific configuration parameters.
- Run the GPU accelerated model.

Example: Sobel Edge Detection

The Sobel edge detection algorithm is a simple edge detection algorithm that performs a 2-D spatial gradient operation on a grayscale image. This algorithm emphasizes the high spatial frequency regions that correspond to the edges of the input image.

The Sobel edge algorithm computes the horizontal gradient (H) and the vertical gradient (V) of the input image by using two orthogonal filter kernels (k and k'). After the filtering operation, the algorithm computes the gradient magnitude and applies a threshold to find the regions of the images that are considered to be edges.

```
k = single([1 2 1; 0 0 0; -1 -2 -1]);  
H = conv2(single(grayImage),k, 'same');  
V = conv2(single(grayImage),k', 'same');  
E = sqrt(H.*H + V.*V);  
edgeImage = uint8((E > threshold) * 255);
```

Test Image



Edge Detected Image

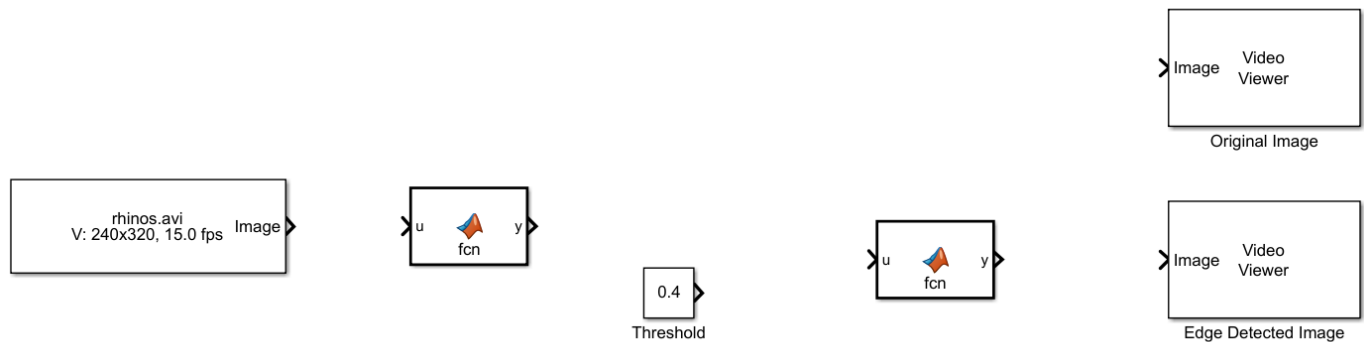


Create Edge Detection Model

- 1 Create a Simulink model and insert two MATLAB Function blocks from the **User-Defined Functions** library.
- 2 Add a Constant block and set its value to 0.4.
- 3 Add a From Multimedia File block from the **Computer Vision Toolbox™** library.
- 4 Open the **Block Parameters** dialog for the From Multimedia File block and set the **File name** parameter to `rhinos.avi`.

Set the **Image signal** parameter to `One multidimensional signal`.

- 5 Add two Video Viewer blocks from the **Computer Vision Toolbox** library to the model.



- 6 Double-click on one of the MATLAB Function blocks. A default function signature appears in the MATLAB Function Block Editor.
- 7 Define a function called `sobel`, which implements the Sobel edge detection algorithm. The function header declares `grayImage` and `threshold` as an argument to the `sobel` function, with `edgeImage` as the return value. Save Editor document to file.

```
function edgeImage = sobel(grayImage,threshold) %#codegen

% Define Kernel for Sobel edge detection
k = single([1 2 1; 0 0 0; -1 -2 -1]);

% Detect Edge
H = conv2(single(grayImage),k, 'same');
V = conv2(single(grayImage),k, 'same');
E = sqrt(H.*H + V.*V);
edgeImage = uint8((E > threshold) * 255);

end
```

- 8 Open the block parameters for the MATLAB Function block. On the **Code Generation** tab, select **Reusable function** for **Function packaging** parameter.

If the **Function packaging** parameter is set to any other value, CUDA kernels may not get generated.

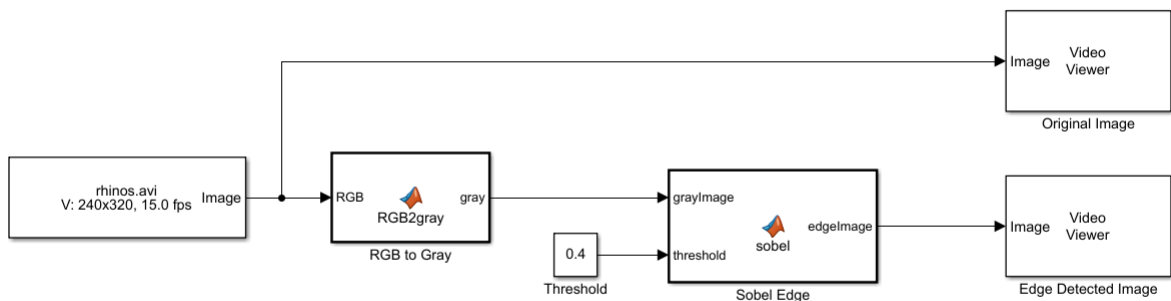
- 9 Modify the other MATLAB Function block to implement the RGB to grayscale conversion prior to the Sobel edge detection operation. Set the **Function packaging** parameter of the MATLAB Function block to **Reusable function**.

```
function gray = RGB2gray(RGB) %#codegen
% Convert color image to grey image

gray = (0.2989 * double(RGB(:,:,1)) + ...
        0.5870 * double(RGB(:,:,2)) + ...
        0.1140 * double(RGB(:,:,3)));

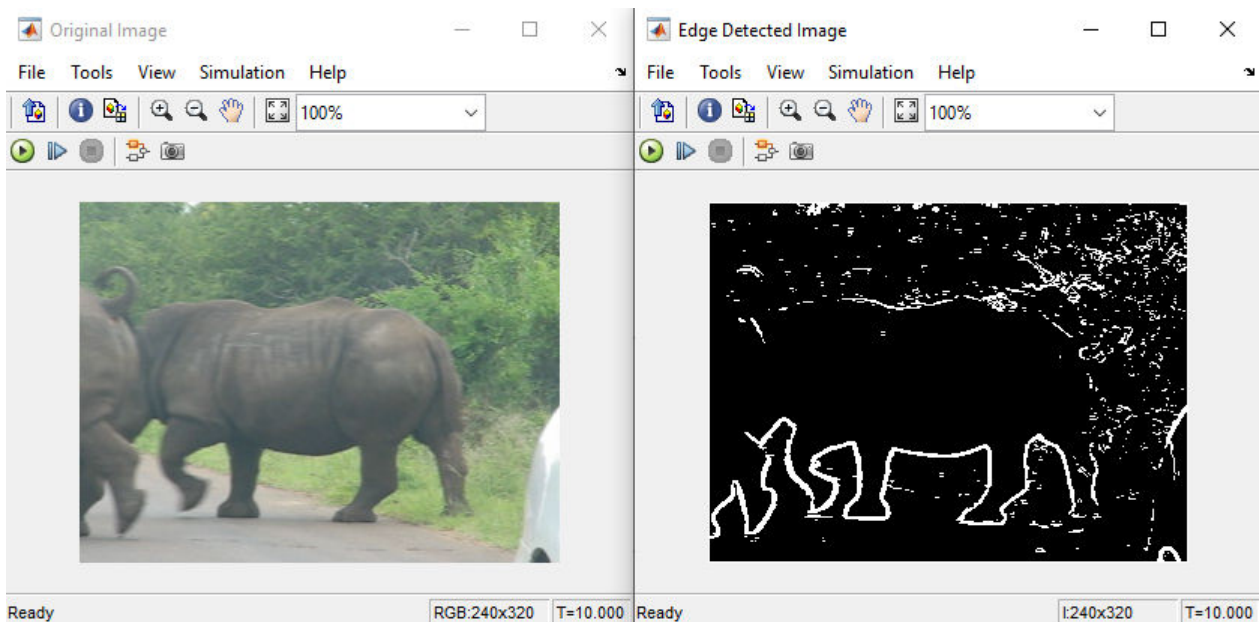
end
```

- 10 Connect these blocks as shown in the diagram. Save the model as `edgeDetection.slx`.



- 11 To test the model for errors, simulate the model in the Simulink Editor. On the toolbar, click **Run**.

To see all video frames during simulation, disable the **Simulation > Drop Frames to improve Performance** option of the Video Viewer block.



Configure Model for GPU Acceleration

Model configuration parameters determine the acceleration method used during simulation.

- 1 Open the Configuration Parameters dialog box. Open the **Solver** pane. To compile your model for acceleration and generate CUDA code, configure the model to use a fixed-step solver. This table shows the solver configuration for this example.

Parameter	Setting	Effect on Generated Code
Type	Fixed-step	Maintains a constant (fixed) step size.
Solver	discrete (no continuous states)	Applies a fixed-step integration technique for computing the state derivative of the model.
Fixed-step size	auto	Simulink chooses the step size.

Solver selection

Type: Fixed-step

Solver: discrete (no continuous states)

▼ Solver details

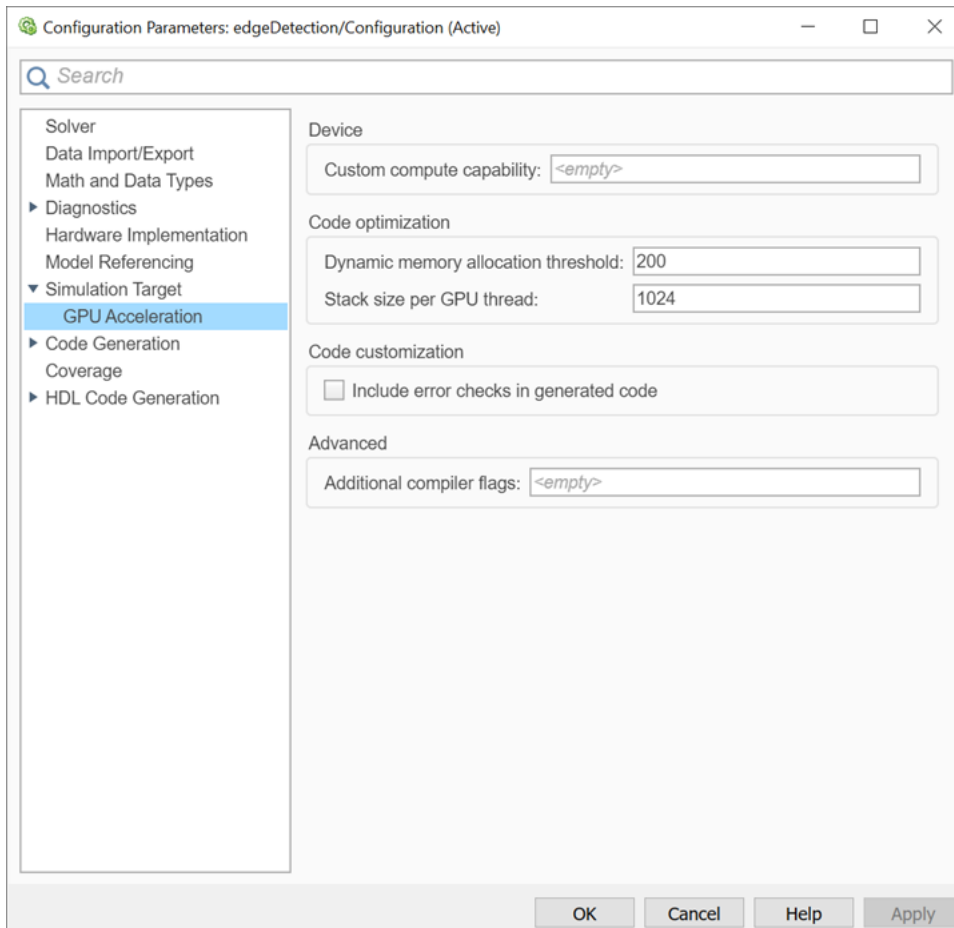
Fixed-step size (fundamental sample time):

auto

- 2 On the **Simulation Target** pane, enable **GPU acceleration** parameter.

Note The **Language** parameter is automatically set to C++.

- 3 GPU Coder specific options are now visible in the **Simulation Target > GPU Acceleration** pane. For the purposes of this example, you can use the default values for all the GPU-specific parameters.



- 4 To save and close the Configuration Parameters dialog box, click **OK**.

You can also use the `set_param` function to configure the model parameters programmatically in the MATLAB command Window.

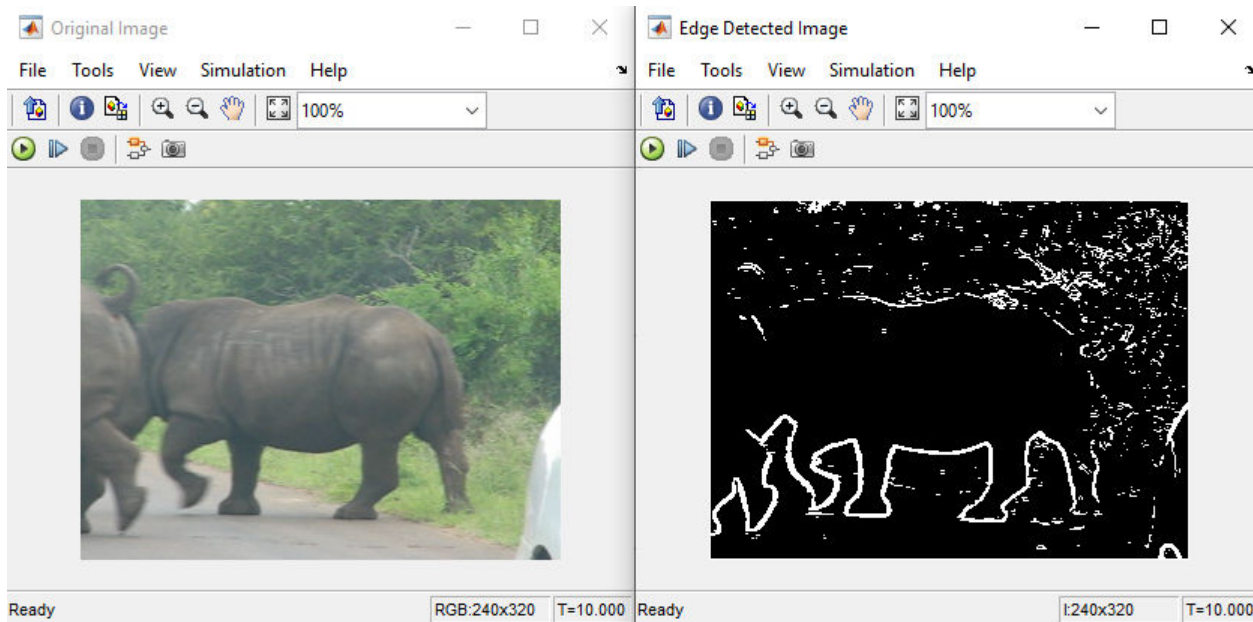
```
set_param('edgeDetection','GPUAcceleration','on');
```

Build GPU Accelerated Model

To build and simulate the GPU accelerated model, select **Run** on the **Simulation** tab or use the following MATLAB command:

```
sim('edgeDetection');
```

The software first checks to see if CUDA code was previously compiled for the model. If code was created previously, the software runs the model. If code was not previously built, the software first generates and compiles the CUDA code, and then runs the model. The code generation tool places the generated code in a subfolder of the working folder called `s\prj/_s\prj/edgeDetection`.



Limitations

- GPU code generation for MATLAB Function blocks in Stateflow® charts is not supported.
- When **GPU acceleration** is enabled, the code generator does not support **Import custom code** for importing custom authored CUDA source files (*.cu). Instead, use `coder.ceval` inside the MATLAB Function block.
- The MATLAB Function block does not support all the data types from the MATLAB language. For supported data types, refer to the block documentation.

See Also

Functions

`open_system` | `load_system` | `save_system` | `close_system` | `bdclose` | `get_param` | `set_param` | `sim` | `slbuild`

More About

- “Code Generation from Simulink Models with GPU Coder” on page 3-8
- “GPU Code Generation for Deep Learning Networks Using MATLAB Function Block” on page 3-14
- “GPU Code Generation for Blocks from the Deep Neural Networks Library” on page 3-22
- “Targeting NVIDIA Embedded Boards” on page 3-30
- “Numerical Equivalence Testing” on page 3-32
- “Parameter Tuning and Signal Monitoring by Using External Mode” on page 3-38
- “GPU Code Generation for Lane Detection in Simulink” on page 3-43
- “GPU Code Generation for a Fog Rectification Simulink Model” on page 3-48

Code Generation from Simulink Models with GPU Coder

GPU Coder generates optimized CUDA code from Simulink models containing MATLAB Function blocks. You can use the generated code and executable for rapid prototyping on NVIDIA GPUs. Code generation reports and traceability enable you to view and analyze the generated code. The basic steps for CUDA code generation by using GPU Coder are:

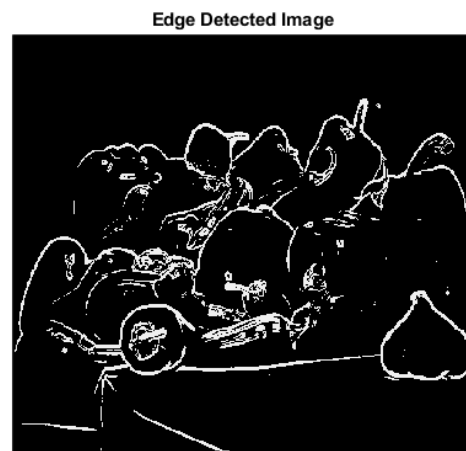
- Create or open a model.
- Configure the model for code generation by selecting the **solver**, **language**, **toolchain**, and other GPU-specific configuration parameters.
- Build the model.

Example: Sobel Edge Detection

The Sobel edge detection algorithm is a simple edge detection algorithm that performs a 2-D spatial gradient operation on a grayscale image. This algorithm emphasizes the high spatial frequency regions that correspond to the edges of the input image.

The Sobel edge algorithm computes the horizontal gradient (H) and the vertical gradient (V) of the input image by using two orthogonal filter kernels (k and k'). After the filtering operation, the algorithm computes the gradient magnitude and applies a threshold to find the regions of the images that are considered to be edges.

```
k = single([1 2 1; 0 0 0; -1 -2 -1]);
H = conv2(single(grayImage),k, 'same');
V = conv2(single(grayImage),k', 'same');
E = sqrt(H.*H + V.*V);
edgeImage = uint8((E > threshold) * 255);
```



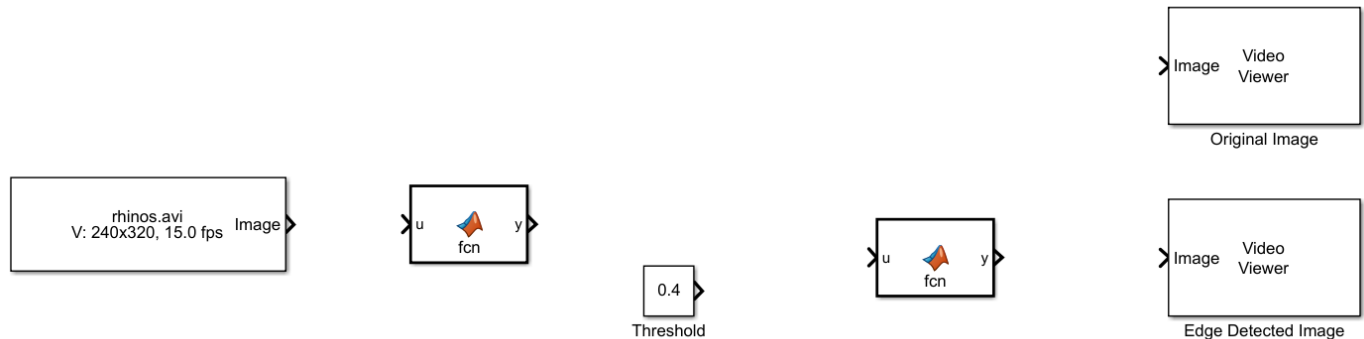
Create Edge Detection Model

- 1 Create a Simulink model and insert two MATLAB Function blocks from the **User-Defined Functions** library.
- 2 Add a Constant block and set its value to 0.4.

- 3 Add a From Multimedia File block from the **Computer Vision Toolbox** library.
- 4 Open the **Block Parameters** dialog box for the From Multimedia File block and set the **File name** parameter to rhinos.avi.

Set the **Image signal** parameter to One multidimensional signal.

- 5 Add two Video Viewer blocks from the **Computer Vision Toolbox** library to the model.



- 6 Double-click on one of the MATLAB Function blocks. A default function signature appears in the MATLAB Function Block Editor.
- 7 Define a function called `sobel`, which implements the Sobel edge detection algorithm. The function header declares `grayImage` and `threshold` as an argument to the `sobel` function, with `edgeImage` as the return value. Save Editor document to file.

```
function edgeImage = sobel(grayImage,threshold) %#codegen

% Define Kernel for Sobel edge detection
k = single([1 2 1; 0 0 0; -1 -2 -1]);

% Detect Edge
H = conv2(single(grayImage),k, 'same');
V = conv2(single(grayImage),k, 'same');
E = sqrt(H.*H + V.*V);
edgeImage = uint8((E > threshold) * 255);

end
```

- 8 Open the block parameters for the MATLAB Function block. On the **Code Generation** tab, select **Reusable function** for **Function packaging** parameter.

If the **Function packaging** parameter is set to any other value, CUDA kernels may not get generated.

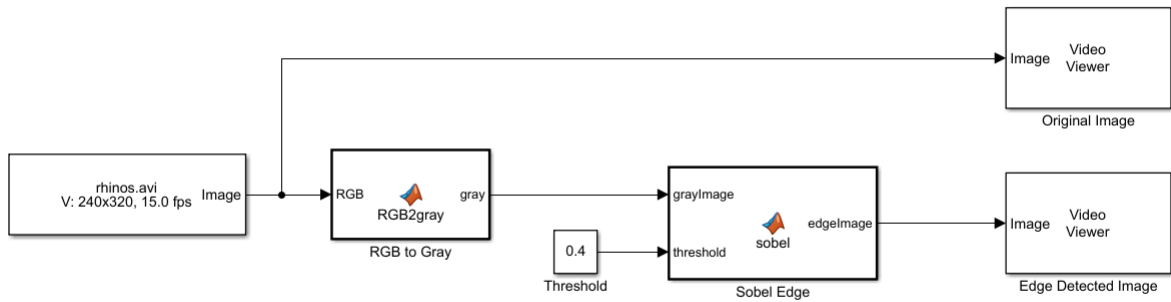
- 9 Modify the other MATLAB Function block to implement the RGB to grayscale conversion prior to the Sobel edge detection operation. Set the **Function packaging** parameter of the MATLAB Function block to **Reusable function**.

```
function gray = RGB2gray(RGB) %#codegen
% Convert color image to grey image

gray = (0.2989 * double(RGB(:,:,1)) + ...
        0.5870 * double(RGB(:,:,2)) + ...
        0.1140 * double(RGB(:,:,3)));
```

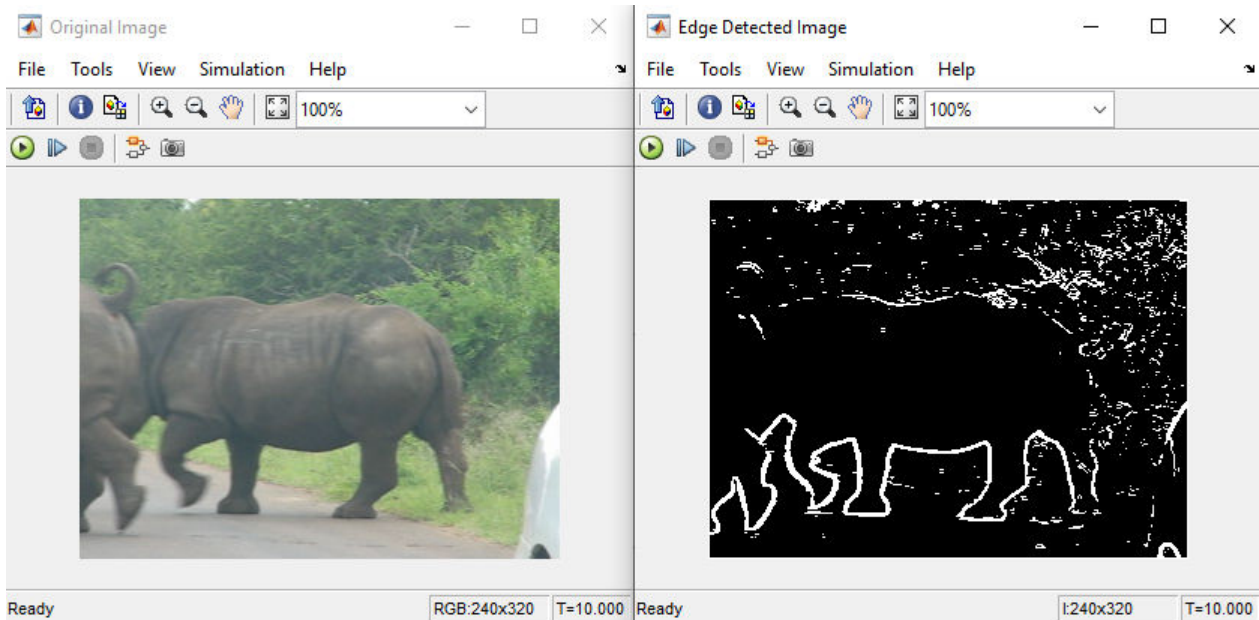
end

- 10 Connect these blocks as shown in the diagram. Save the model as `edgeDetection.slx`.



- 11 To test the model for errors, simulate the model in the Simulink Editor. On the toolbar, click **Run**.

To see all video frames during simulation, disable the **Simulation > Drop Frames to improve Performance** option of the Video Viewer block.



Configure Model for Code Generation

The model configuration parameters provide many options for the code generation and build process.

- 1 Open the Configuration Parameters dialog box. Open the **Solver** pane. To compile your model for acceleration and generate CUDA code, configure the model to use a fixed-step solver. This table shows the solver configuration for this example.

Parameter	Setting	Effect on Generated Code
Type	Fixed-step	Maintains a constant (fixed) step size, which is required for code generation
Solver	discrete (no continuous states)	Applies a fixed-step integration technique for computing the state derivative of the model
Fixed-step size	auto	Simulink chooses the step size

Solver selection

Type: Solver:

▼ Solver details

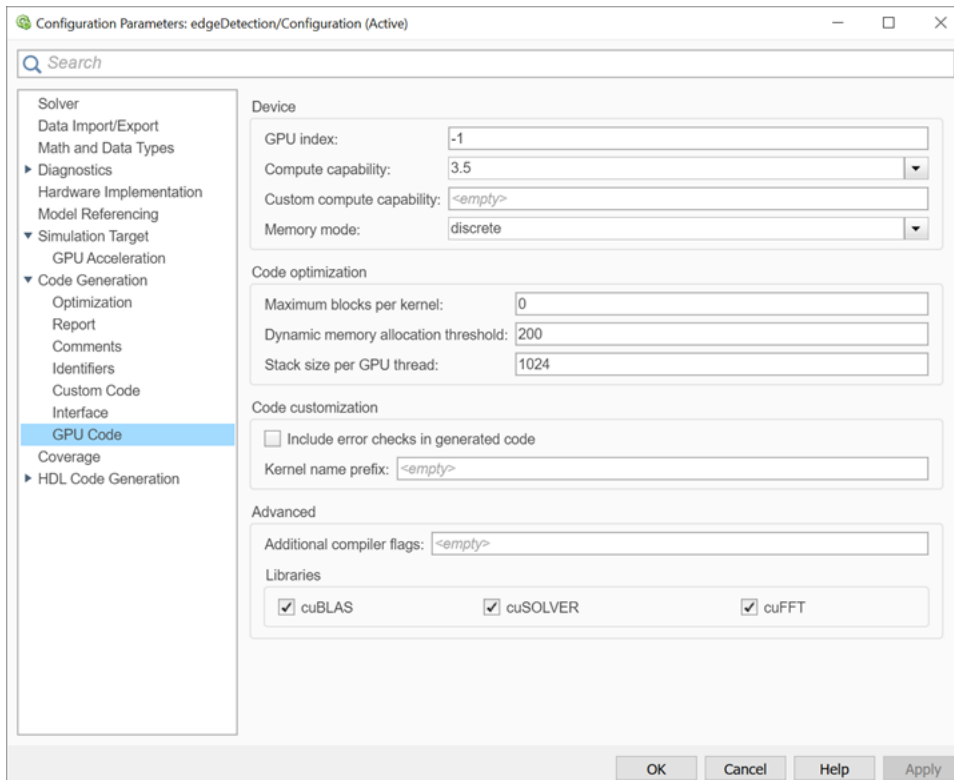
Fixed-step size (fundamental sample time):

- 2 On the **Code Generation** pane, set the **System target file** to `grt.tlc`.
You can also use the Embedded Coder target file `ert.tlc` or a custom system target file.
For GPU code generation, the custom target file must be based on `grt.tlc` or `ert.tlc`. For information on developing a custom target file, see “Customize System Target Files” (Simulink Coder).
- 3 Set the **Language** to C++.
- 4 Select **Generate GPU code**.
- 5 On the **Code Generation** pane, select **Generate code only**.
- 6 Select the **Toolchain**. For Linux® platforms, select **NVIDIA CUDA | gmake (64-bit Linux)**. For Windows® systems, select **NVIDIA CUDA (w/Microsoft Visual C++ 20XX) | nmake (64-bit windows)**.

When using a custom system target file, you must set the build controls for the toolchain approach. To learn more about toolchain approach for custom targets, see “Support Toolchain Approach with Custom Target” (Simulink Coder).

- 7 On the **Code Generation > Interface** pane, disable **MAT-file logging**.
- 8 On the **Code Generation > Report** pane, select **Create code generation report** and **Open report automatically**.
- 9 When you enable the **Generate GPU code** parameter, options specific to GPU Coder appear in the **Code Generation > GPU Code** pane.

For this example, you can use the default values of the GPU-specific parameters in **Code Generation > GPU Code** pane.



10 Click **OK** to save and close the Configuration Parameters dialog box.

You can use the `set_param` function to configure the model parameter programmatically in the MATLAB Command Window.

```
set_param('edgeDetection','GenerateGPUCode','CUDA');
```

Generate CUDA Code for the Model

- 1 In the Simulink Editor, open the **Simulink Coder** app.
- 2 Generate code.

Messages appear in the Diagnostics Viewer. The code generator produces CUDA source and header files, and an HTML code generation report. The code generator places the files in a *build folder*, a subfolder named `edgeDetection_grt_rtw` under your current working folder.

You can find the CUDA kernels in the `<model_name>_eML_blk_kernel` and `<model_name>_eML_blk_kernel_c` functions. The information within the triple chevrons is the execution configuration for the kernel.

Limitations

- GPU code generation for MATLAB Function blocks in Stateflow charts is not supported.
- The MATLAB Function block does not support all the data types from the MATLAB language. For supported data types, refer to the block documentation.
- For GPU code generation, the custom target file must be based on `grt.tlc` or `ert.tlc`.

See Also

Functions

`open_system` | `load_system` | `save_system` | `close_system` | `bdclose` | `get_param` | `set_param` | `sim` | `slbuild`

More About

- “Simulation Acceleration by Using GPU Coder” on page 3-2
- “GPU Code Generation for Deep Learning Networks Using MATLAB Function Block” on page 3-14
- “GPU Code Generation for Blocks from the Deep Neural Networks Library” on page 3-22
- “Targeting NVIDIA Embedded Boards” on page 3-30
- “Numerical Equivalence Testing” on page 3-32
- “Parameter Tuning and Signal Monitoring by Using External Mode” on page 3-38
- “GPU Code Generation for Lane Detection in Simulink” on page 3-43
- “GPU Code Generation for a Fog Rectification Simulink Model” on page 3-48

GPU Code Generation for Deep Learning Networks Using MATLAB Function Block

With GPU Coder, you can generate optimized code for Simulink models containing a variety of trained deep learning networks. You can implement the deep learning functionality in Simulink by using MATLAB Function blocks or by using blocks from the **Deep Neural Networks** library. When implementing with MATLAB Function blocks, use the `coder.loadDeepLearningNetwork` function to load a trained deep learning network and use the object functions of the network object to obtain the desired responses. You can configure the code generator to take advantage of the NVIDIA CUDA deep neural network library (cuDNN) and TensorRT high performance inference libraries for NVIDIA GPUs. The generated code implements the deep convolutional neural network (CNN) by using the architecture, the layers, and parameters that you specify in network object.

Example: Classify Images by Using GoogLeNet

GoogLeNet has been trained on over a million images and can classify images into 1000 object categories (such as keyboard, coffee mug, pencil, and animals). The network takes an image as input, and then outputs a label for the object in the image together with the probabilities for each of the object categories. This example shows you how to perform simulation and generate CUDA code for the pretrained `googlenet` deep convolutional neural network and classify an image.

- 1 Load the pretrained GoogLeNet network. You can choose to load a different pretrained network for image classification. If you do not have the required support packages installed, install the software according to the instructions provided.

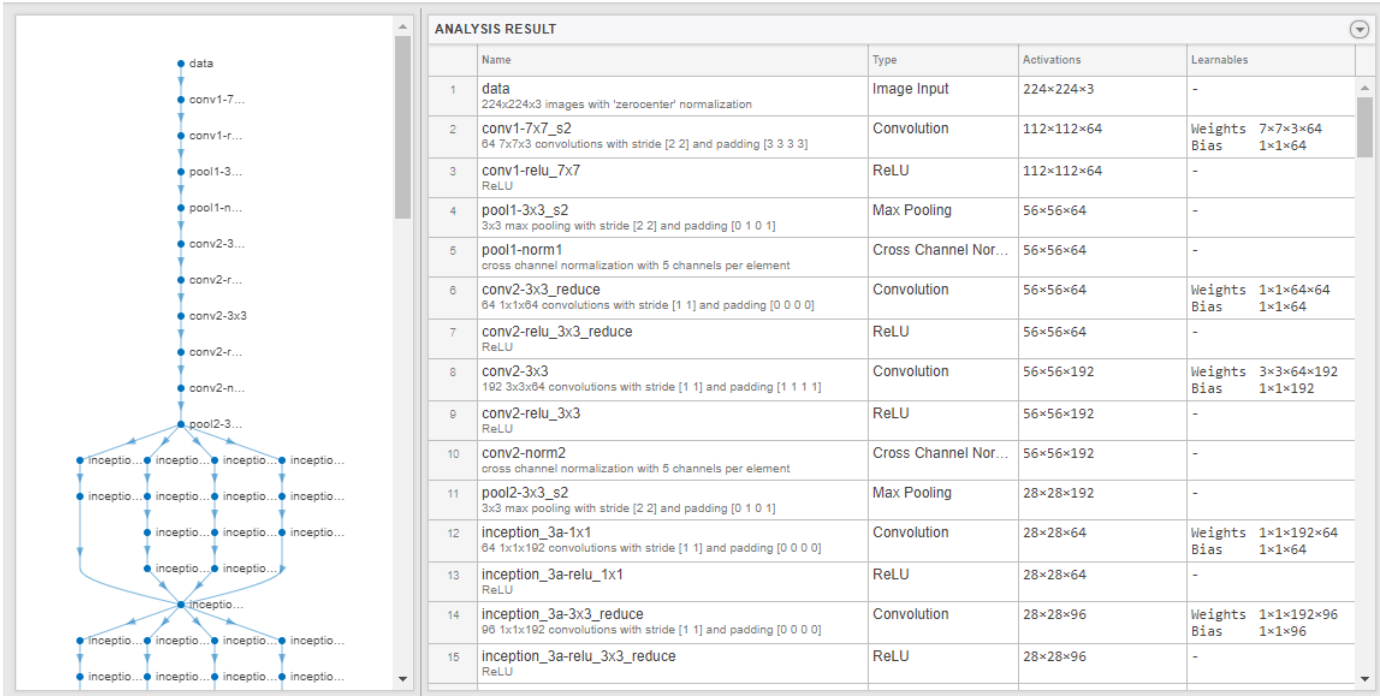
```
net = googlenet;
```

- 2 The object `net` contains the `DAGNetwork` object. Use the `analyzeNetwork` function to display an interactive visualization of the network architecture, to detect errors and issues in the network, and to display detailed information about the network layers. The layer information includes the sizes of layer activations and learnable parameters, the total number of learnable parameters, and the sizes of state parameters of recurrent layers.

```
analyzeNetwork(net);
```

net

Analysis date: 20-Jun-2019 23:27:32

144 
layers0 
warnings0 
errors

- 3 The image that you want to classify must have the same size as the input size of the network. For GoogLeNet, the size of the `imageInputLayer` is 224-by-224-by-3. The `Classes` property of the output `classificationLayer` contains the names of the classes learned by the network. View 10 random class names out of the total of 1000.

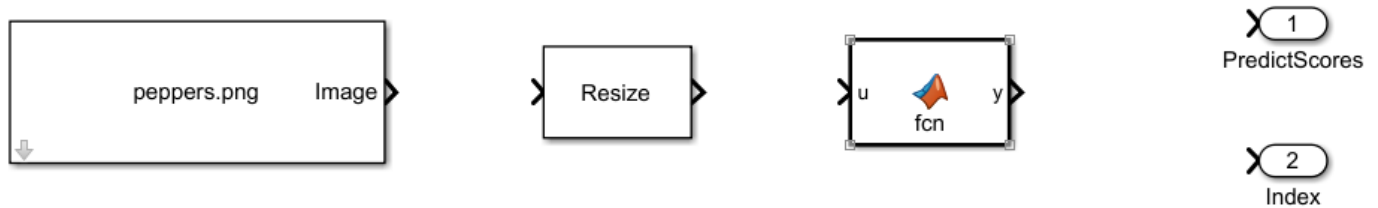
```
classNames = net.Layers(end).Classes;
numClasses = numel(classNames);
disp(classNames(randperm(numClasses,10)))
```

```
'speedboat'
'window screen'
'isopod'
'wooden spoon'
'lipstick'
'drake'
'hyena'
'dumbbell'
'strawberry'
'custard apple'
```

Create GoogLeNet Model

- 1 Create a Simulink model and insert a MATLAB Function block from the **User-Defined Functions** library.
- 2 Add an Image From File block from the **Computer Vision Toolbox** library and set the File name parameter to `peppers.png`.
- 3 Add a Resize block from the **Computer Vision Toolbox** library to the model. Set the **Specify** parameter of the Resize block to **Number of output rows and columns** and enter [224

224] as the value for **Number of output rows and columns**. This block resizes the input image to that of the input layer of the network.



- 4 Double-click the MATLAB Function block. A default function signature appears in the MATLAB Function Block Editor.
- 5 Define a function called `googlenet_predict`, which implements the prediction entry-point function. The function header declares `in` as an argument to the `googlenet_predict` function, with `scores` and `indxTop` as the return value.

```
function [scores,indxTop] = googlenet_predict(in) %#codegen
persistent mynet;
if isempty(mynet)
    mynet = coder.loadDeepLearningNetwork('googlenet');
end
% pass in input
predict_scores = predict(mynet,in);
[scores,indx] = sort(predict_scores, 'descend');
indxTop = indx(1:5);
```

A persistent object `mynet` loads the `DAGNetwork` object. At the first call to the entry-point function, the persistent object is constructed and set up. On subsequent calls to the function, the same object is reused to call `predict` on inputs, avoiding reconstructing and reloading the network object.

You can also use the `activations` method to network activations for a specific layer. For example, the following line of code returns the network activations for the layer specified in `layerIdx`.

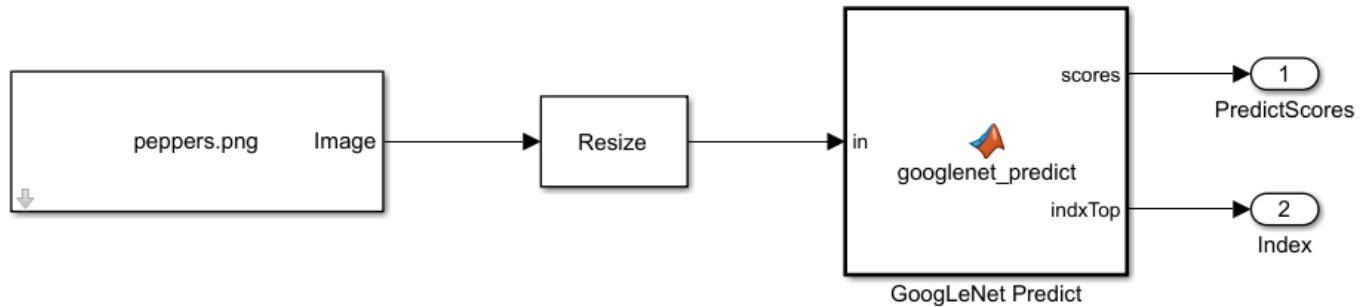
```
out = activations(mynet,in,layerIdx,'OutputAs','Channels');
```

You can also use the `classify` method to predict class labels for the image data in `in` using the trained network `mynet`.

```
[out,scores] = classify(mynet,in);
```

For LSTM networks, you can use the `predictAndUpdateState` and `resetState` methods. For usage notes and limitations of these method, see “Supported Functions” on page 1-6.

- 6 Open the block parameters of the MATLAB Function block. On the **Code Generation** tab, select **Reusable** function for **Function packaging**.
- 7 Connect these blocks as shown in the diagram. Save the model as `googlenetModel`.



Configure Model for GPU Acceleration

Model configuration parameters determine the acceleration method used during simulation.

- 1 Open the Configuration Parameters dialog box. Open the **Solver** pane. To compile your model for acceleration and generate CUDA code, configure the model to use a fixed-step solver. This table shows the solver configuration for this example.

Parameter	Setting	Effect on Generated Code
Type	Fixed-step	Maintains a constant (fixed) step size, which is required for code generation
Solver	discrete (no continuous states)	Applies a fixed-step integration technique for computing the state derivative of the model
Fixed-step size	auto	Simulink chooses the step size

Solver selection

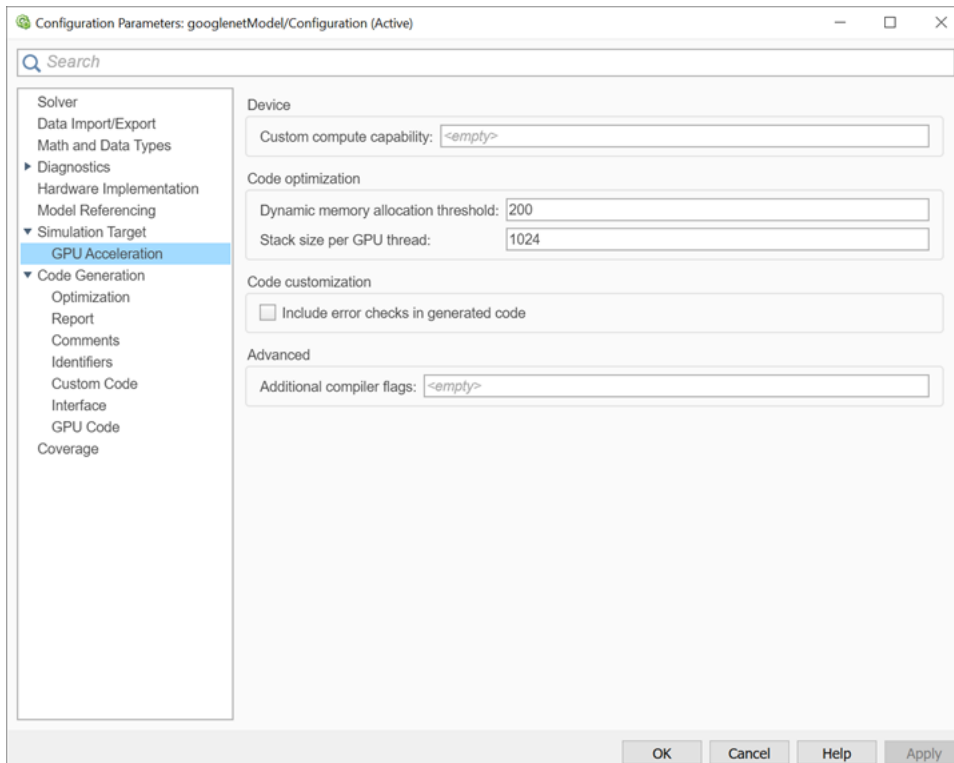
Type: Fixed-step Solver: discrete (no continuous states)

▼ Solver details

Fixed-step size (fundamental sample time): auto

- 2 Select the **Simulation Target** pane. Set the **Language** to C++.
- 3 Select **GPU acceleration**.

options specific to GPU Coder are now visible in the **Simulation Target > GPU Acceleration** pane. For this example, you can use the default values for these GPU-specific parameters.



- 4 On the **Simulation Target** pane, set the **Target Library** parameter in the **Deep learning** group to cuDNN.



You can also select TensorRT to target TensorRT high performance inference libraries for NVIDIA GPUs.

- 5 Click **OK** to save and close the Configuration Parameters dialog box.

You can use `set_param` to configure the model parameter programmatically in the MATLAB command Window.

```
set_param('googlenetModel', 'GPUAcceleration', 'on');
```

Build GPU Accelerated Model

- 1 To build and simulate the GPU accelerated model, select **Run** on the **Simulation** tab or use the MATLAB command:

```
out = sim('googlenetModel');
```

The software first checks to see if CUDA/C++ code was previously compiled for your model. If code was created previously, the software runs the model. If code was not previously built, the

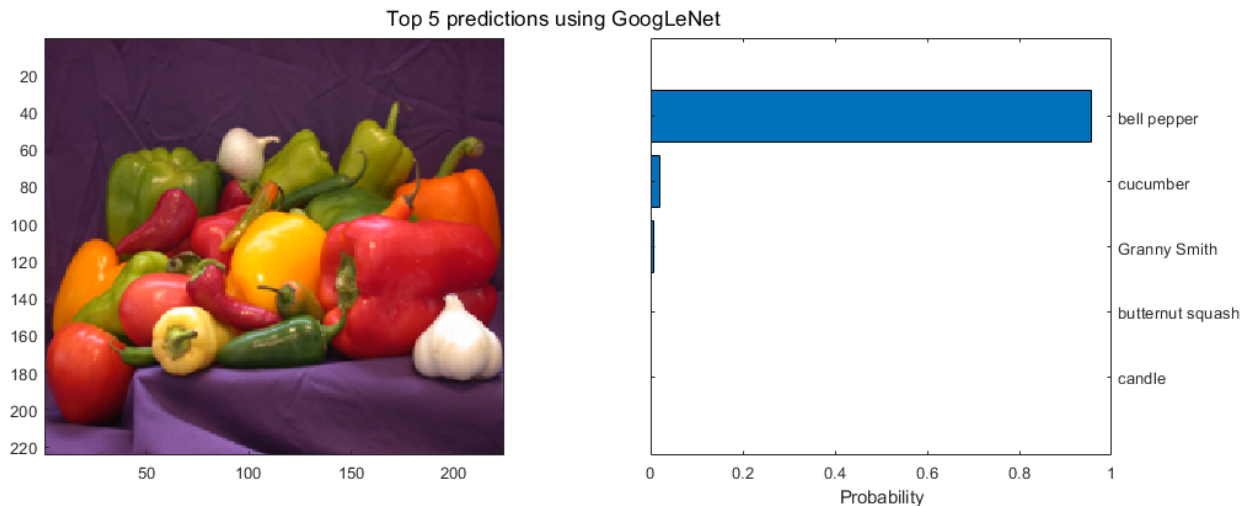
software first generates and compiles the CUDA/C++ code, and then runs the model. The code generation tool places the generated code in a subfolder of the working folder called `s_lprj/_s_lprj/googlenetModel`.

- 2 Display the top five predicted labels and their associated probabilities as a histogram. Because the network classifies images into so many object categories, and many categories are similar, it is common to consider the top-five accuracy when evaluating networks. The network classifies the image as a bell pepper with a high probability.

```
im = imread('peppers.png');
classNamesTop = classNames(out.yout{2}.Values.Data(:, :), 1)

h = figure;
h.Position(3) = 2*h.Position(3);
ax1 = subplot(1,2,1);
ax2 = subplot(1,2,2);

image(ax1, im);
barh(ax2, out.yout{1}.Values.Data(1, 5:-1:1))
xlabel(ax2, 'Probability')
yticklabels(ax2, classNamesTop(5:-1:1))
ax2.YAxisLocation = 'right';
sgtitle('Top 5 predictions using GoogLeNet')
```



Configure the Model for Code Generation

The model configuration parameters provide many options for the code generation and build process.

- 1 Select the **Code Generation** pane. Set the **System target file** to `grt.tlc`.

You can also use the Embedded Coder target file `ert.tlc` or a custom system target file.

For GPU code generation, the custom target file must be based on `grt.tlc` or `ert.tlc`. For information on developing a custom target file, see “Customize System Target Files” (Simulink Coder).

- 2 Set the **Language** to C++.

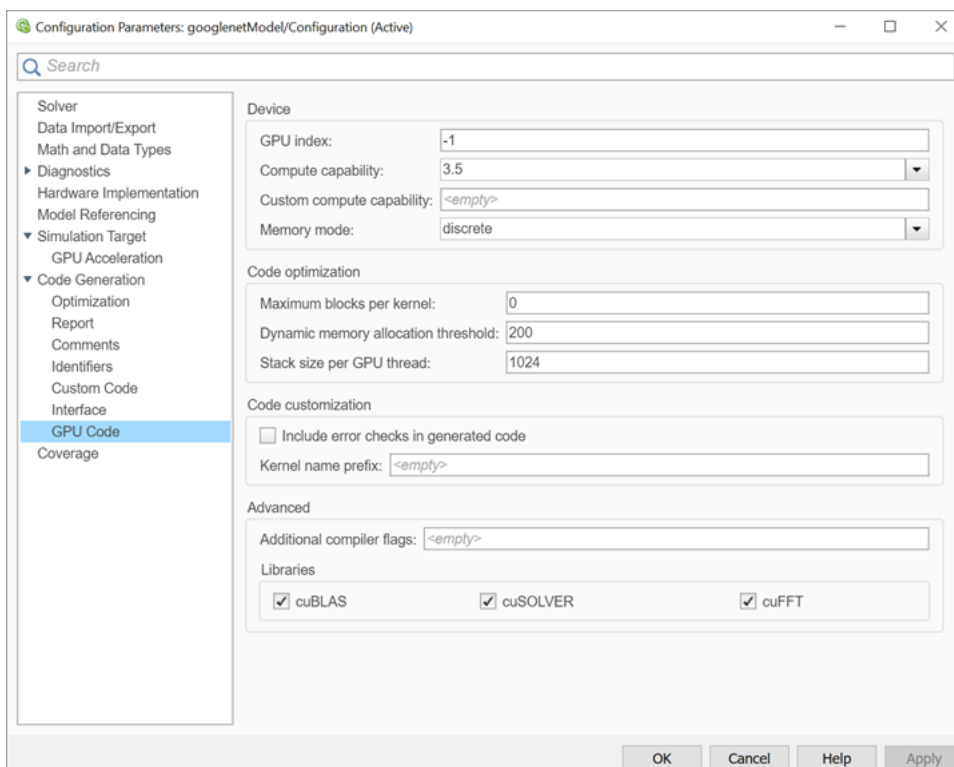
- 3 Select **Generate GPU code**.
- 4 Select **Generate code only**.
- 5 Select the **Toolchain**. For Linux platforms, select **NVIDIA CUDA | gmake (64-bit Linux)**. For Windows systems, select **NVIDIA CUDA (w/Microsoft Visual C++ 20XX) | nmake (64-bit windows)**.

When using a custom system target file, you must set the build controls for the toolchain approach. To learn more about toolchain approach for custom targets, see “Support Toolchain Approach with Custom Target” (Simulink Coder).

- 6 On the **Code Generation > Report** pane, select **Create code generation report** and **Open report automatically**.
- 7 On the **Code Generation > Interface** pane, set the **Target Library** in the **Deep learning** group to **cuDNN**.

You can also select **TensorRT** to target **TensorRT** high performance inference libraries for **NVIDIA GPUs**.

- 8 When the **Generate GPU code** parameter is enabled, options specific to GPU Coder are visible in the **Code Generation > GPU Code** pane. For this example, you can use the default values of the GPU-specific parameters in **Code Generation > GPU Code** pane.



- 9 Click **OK** to save and close the Configuration Parameters dialog box.

You can also use `set_param` function to configure the model parameter programmatically in the MATLAB Command Window.

```
set_param('googlenetModel','GenerateGPUCode','CUDA');
```

Generate CUDA Code for the Model

- 1 In the Simulink Editor, open the **Simulink Coder** app.
- 2 Generate code.

Messages appear in the Diagnostics Viewer. The code generator produces CUDA source and header files, and an HTML code generation report. The code generator places the files in a *build folder*, a subfolder named `googlenetModel_grt_rtw` under your current working folder.

Limitations

- GPU code generation for MATLAB Function blocks in Stateflow charts is not supported.
- When **GPU acceleration** is enabled, the code generator does not support **Import custom code** for importing custom authored CUDA source files (*.cu). Instead, use `coder.ceval` inside the MATLAB Function block.
- The MATLAB Function block does not support all the data types from the MATLAB language. For supported data types, refer to the block documentation.
- For GPU code generation, the custom target file must be based on `grt.tlc` or `ert.tlc`.
- For deploying the generated code, it is recommended to use the **Generate an example main program** option to generate the `ert_main.cu` module. This option requires the Embedded Coder license.

You can also use the `rt_cppclass_main.cpp` static main module provided by MathWorks®. However, the static main file must be modified such that the models class constructor points to the deep learning object. For example,

```
static googlenetModelModelClass::DeepLearning_googlenetModel_T
    googlenetModel_DeepLearning;
static googlenetModelModelClass googlenetModel_Obj{ &googlenetModel_DeepLearning};
```

See Also

Functions

`open_system` | `load_system` | `save_system` | `close_system` | `bdclose` | `get_param` | `set_param` | `sim` | `slbuild`

More About

- “Simulation Acceleration by Using GPU Coder” on page 3-2
- “Code Generation from Simulink Models with GPU Coder” on page 3-8
- “GPU Code Generation for Blocks from the Deep Neural Networks Library” on page 3-22
- “Targeting NVIDIA Embedded Boards” on page 3-30
- “Numerical Equivalence Testing” on page 3-32
- “Parameter Tuning and Signal Monitoring by Using External Mode” on page 3-38

GPU Code Generation for Blocks from the Deep Neural Networks Library

With GPU Coder, you can generate optimized code for Simulink models containing a variety of trained deep learning networks. You can implement deep learning functionality in Simulink by using MATLAB Function blocks or by using blocks from the **Deep Neural Networks** from the Deep Learning Toolbox or the **Computer Vision Toolbox > Analysis and Enhancement** library from the Computer Vision Toolbox.

GPU Coder supports the following deep learning blocks:

- Predict block — Predict responses using the trained network specified through the block parameter.

For more information about working with the Predict block, see “Lane and Vehicle Detection in Simulink Using Deep Learning” (Deep Learning Toolbox).

- Image Classifier block — Classify data using a trained deep learning neural network specified through the block parameter.

For more information about working with the Image Classifier block, see “Classify ECG Signals in Simulink Using Deep Learning” (Deep Learning Toolbox).

- Stateful Classify block — Predicts class labels for the data at the input by using the trained recurrent neural network specified through the block parameter.
- Stateful Predict block — Predicts responses for the data at the input by using the trained recurrent neural network specified through the block parameter.
- Deep Learning Object Detector block — Predicts bounding boxes, class labels, and scores for the input image by using the trained object detector specified through the block parameter.

For more information about working with the Deep Learning Object Detector block, see “Lane and Vehicle Detection in Simulink Using Deep Learning” (Deep Learning Toolbox).

These library blocks enable loading of a pretrained network into the Simulink model from a MAT-file or from a MATLAB function.

You can configure the code generator to take advantage of the NVIDIA CUDA deep neural network library (cuDNN) and TensorRT high performance inference libraries for NVIDIA GPUs. The generated code implements the deep convolutional neural network (CNN) by using the architecture, the layers, and parameters that you specify in the network object.

Example: Classify Images by Using GoogLeNet

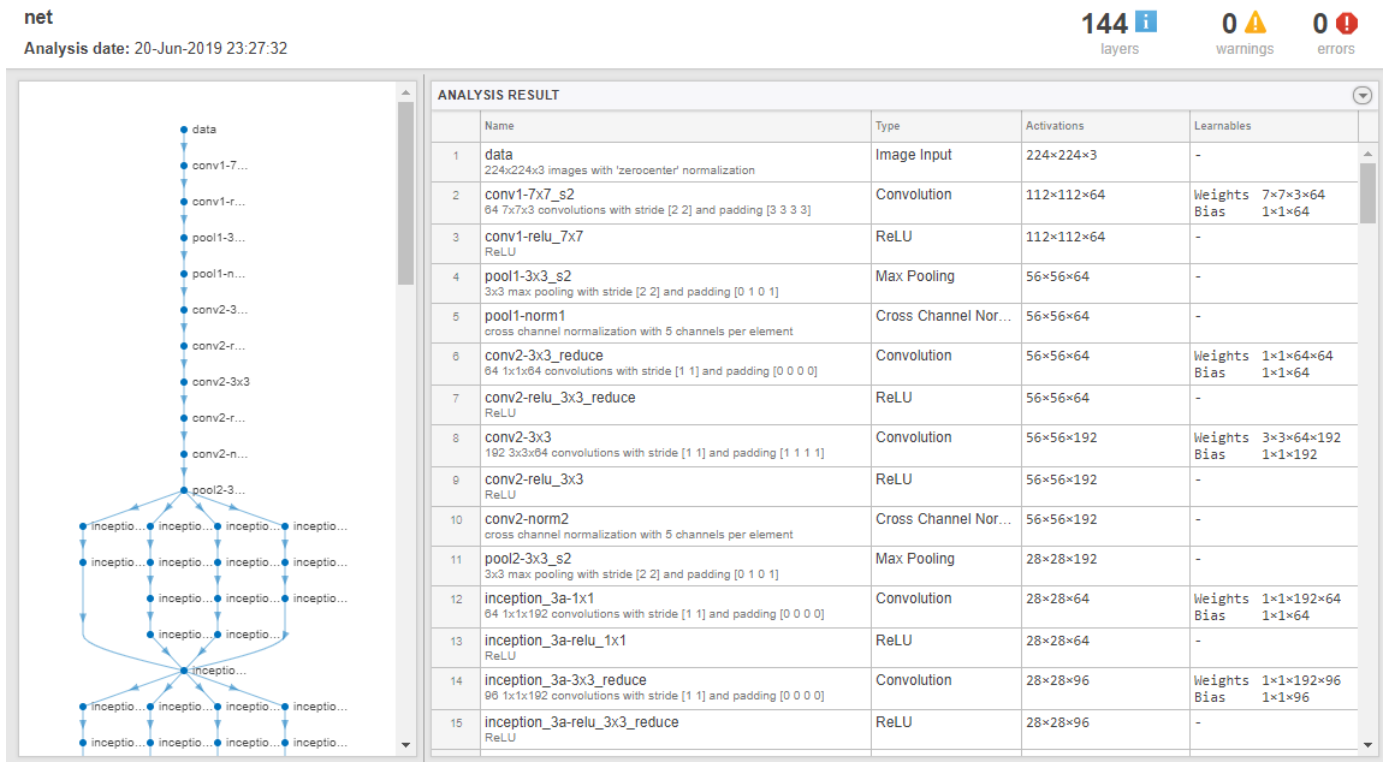
GoogLeNet has been trained on over a million images and can classify images into 1000 object categories (such as keyboard, coffee mug, pencil, and animals). The network takes an image as input, and then outputs a label for the object in the image with the probabilities for each of the object categories. This example shows how to perform simulation and generate CUDA code for the pretrained googlenet deep convolutional neural network and classify an image. The pretrained networks are available as support packages from the Deep Learning Toolbox.

- 1 Load the pretrained GoogLeNet network.

```
net = googlenet;
```

- 2 The object `net` contains the `DAGNetwork` object. Use the `analyzeNetwork` function to display an interactive visualization of the network architecture, to detect errors and issues in the network, and to display detailed information about the network layers. The layer information includes the sizes of layer activations and learnable parameters, the total number of learnable parameters, and the sizes of state parameters of recurrent layers.

```
analyzeNetwork(net);
```



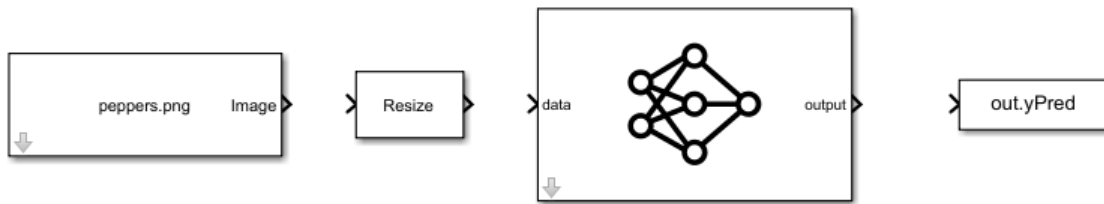
- 3 The image that you want to classify must have the same size as the input size of the network. For GoogLeNet, the size of the `imageInputLayer` is 224-by-224-by-3. The `Classes` property of the output `classificationLayer` contains the names of the classes learned by the network. View 10 random class names out of the total of 1000.

```
classNames = net.Layers(end).Classes;  
numClasses = numel(classNames);  
disp(classNames(randperm(numClasses,10)))
```

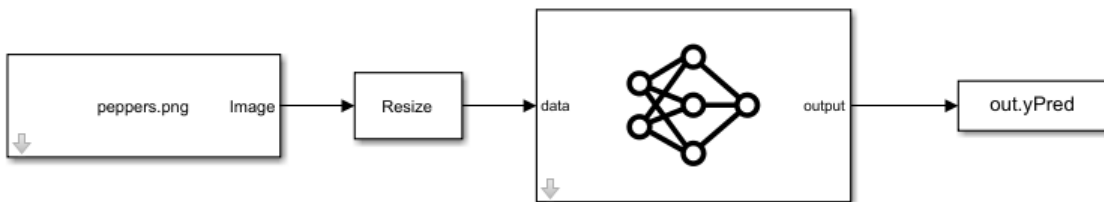
```
'speedboat'  
'window screen'  
'isopod'  
'wooden spoon'  
'lipstick'  
'drake'  
'hyena'  
'dumbbell'  
'strawberry'  
'custard apple'
```

Create GoogLeNet Model

- 1 Create a Simulink model and insert a Predict block from the **Deep Neural Networks** library.
- 2 Add an Image From File block from the **Computer Vision Toolbox** library and set the File name parameter to `peppers.png`. Add a Resize block from the **Computer Vision Toolbox** library to the model. Set the **Specify** parameter of the Resize block to Number of output rows and columns and enter `[224 224]` as the value for **Number of output rows and columns**. The resize block resizes the input image to that of the input layer of the network. Add a To Workspace to the model and change the variable name to `yPred`.



- 3 Open the **Block Parameters (subsystem)** of the Predict block. Select `Network` from `MATLAB` function for **Network** and `googlenet` for **MATLAB function**.
- 4 Connect these blocks as shown in the diagram. Save the model as `googlenetModel.slx`.



Configure the Model for GPU Acceleration

Model configuration parameters determine the acceleration method used during simulation.

- 1 Open the Configuration Parameters dialog box. Open the **Solver** pane. To compile your model for acceleration and generate CUDA code, configure the model to use a fixed-step solver. This table shows the solver configuration for this example.

Parameter	Setting	Effect on Generated Code
Type	Fixed-step	Maintains a constant (fixed) step size, which is required for code generation
Solver	discrete (no continuous states)	Applies a fixed-step integration technique for computing the state derivative of the model
Fixed-step size	auto	Simulink chooses the step size

Solver selection

Type: Fixed-step Solver: discrete (no continuous states)

▼ **Solver details**

Fixed-step size (fundamental sample time): auto

- 2 Select the **Simulation Target** pane. Set the **Language** to C++.
- 3 Select **GPU acceleration**. Options specific to GPU Coder are now visible in the **Simulation Target > GPU Acceleration** pane. For this example, you can use the default values of these parameters.

Configuration Parameters: googlenetModel/Configuration (Active)

Search

Solver
Data Import/Export
Math and Data Types
▶ Diagnostics
Hardware Implementation
Model Referencing
▼ Simulation Target
GPU Acceleration
▼ Code Generation
Optimization
Report
Comments
Identifiers
Custom Code
Interface
GPU Code
Coverage

Device
Custom compute capability: <empty>

Code optimization
Dynamic memory allocation threshold: 200
Stack size per GPU thread: 1024

Code customization
 Include error checks in generated code

Advanced
Additional compiler flags: <empty>

OK Cancel Help Apply

- 4 On the **Simulation Target** pane, set the **Target Library** in the **Deep learning** group to cuDNN. You can also select TensorRT.

Deep learning

Target library: cuDNN

Auto tuning

- 5 Click **OK** to save and close the Configuration Parameters dialog box.

You can use `set_param` to configure the model parameter programmatically in the MATLAB Command Window.

```
set_param('googlenetModel','GPUAcceleration','on');
```

Build GPU Accelerated Model

- 1 To build and simulate the GPU accelerated model, select **Run** on the **Simulation** tab or use the command:

```
out = sim('googlenetModel');
```

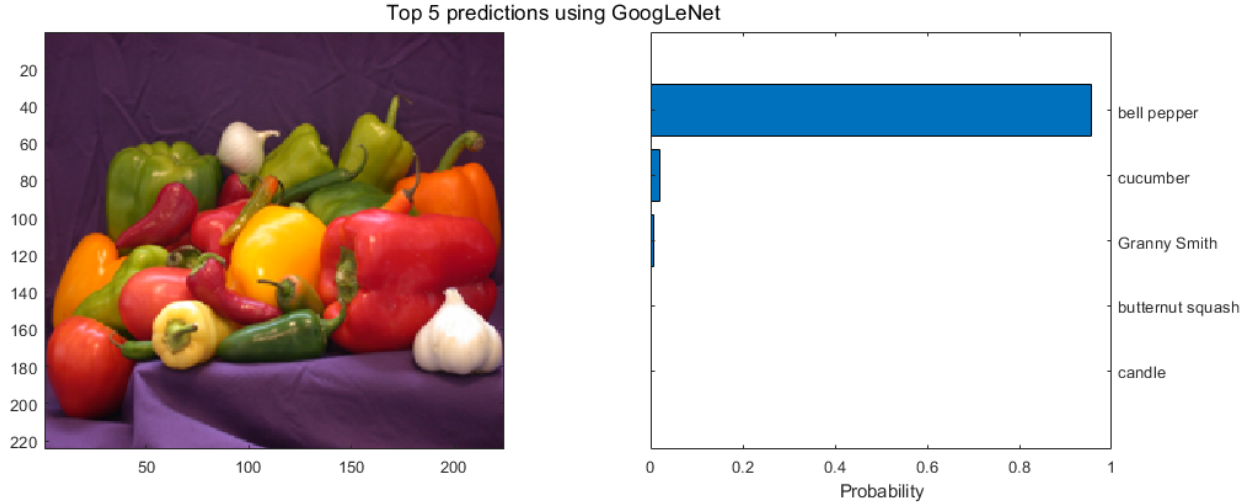
The software first checks to see if CUDA/C++ code was previously compiled for your model. If code was created previously, the software runs the model. If code was not previously built, the software first generates and compiles the CUDA/C++ code, and then runs the model. The code generation tool places the generated code in a subfolder of the working folder called `slprj/_slprj/googlenetModel`.

- 2 Display the top five predicted labels and their associated probabilities as a histogram. Because the network classifies images into so many object categories, and many categories are similar, it is common to consider the top-five accuracy when evaluating networks. The network classifies the image as a bell pepper with a high probability.

```
im = imread('peppers.png');
predict_scores = out.yPred.Data(:,:,1);
[scores,indx] = sort(predict_scores,'descend');
topScores = scores(1:5);
classNamesTop = classNames(indx(1:5))

h = figure;
h.Position(3) = 2*h.Position(3);
ax1 = subplot(1,2,1);
ax2 = subplot(1,2,2);

image(ax1,im);
barh(ax2,topScores(1,5:-1:1,1))
xlabel(ax2,'Probability')
yticklabels(ax2,classNamesTop(5:-1:1))
ax2.YAxisLocation = 'right';
sgtitle('Top 5 predictions using GoogLeNet')
```

Configure Model for Code Generation

The model configuration parameters provide many options for the code generation and build process.

- 1 In Configuration Parameters dialog box, select **Code Generation** pane. Set the **System target file** to `grt.tlc`.

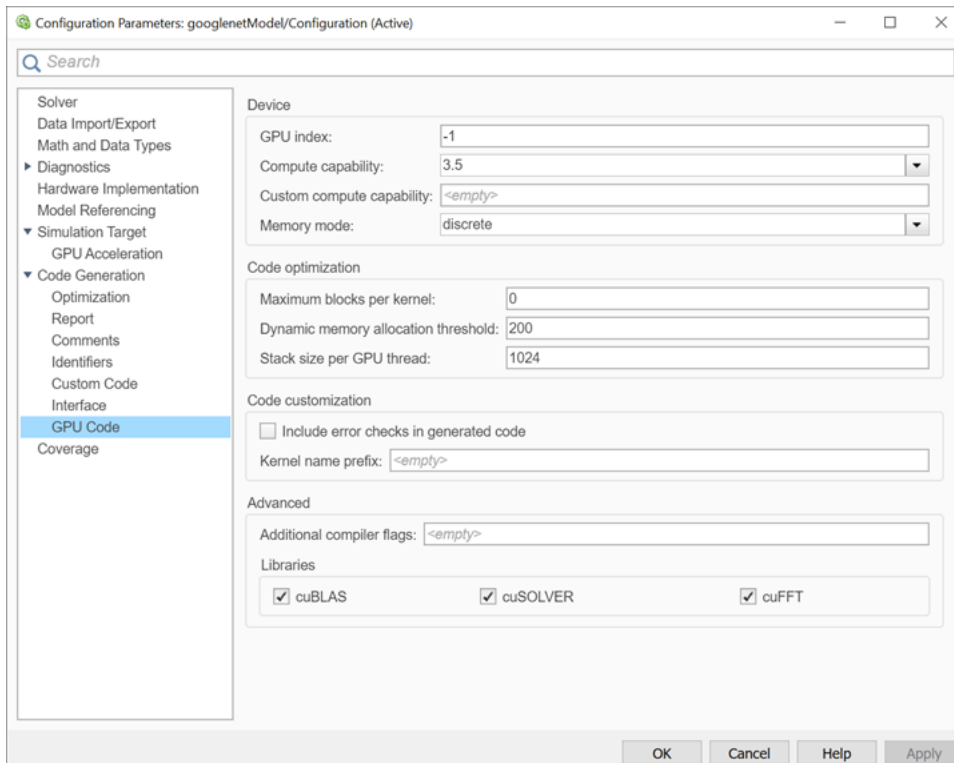
You can also use the Embedded Coder target file `ert.tlc` or a custom system target file.

For GPU code generation, the custom target file must be based on `grt.tlc` or `ert.tlc`. For information on developing a custom target file, see “Customize System Target Files” (Simulink Coder).

- 2 Set the **Language** to C++.
- 3 Select **Generate GPU code**.
- 4 Select **Generate code only**.
- 5 Select the **Toolchain**. For Linux platforms, select **NVIDIA CUDA | gmake (64-bit Linux)**. For Windows systems, select **NVIDIA CUDA (w/Microsoft Visual C++ 20XX) | nmake (64-bit windows)**.

When using a custom system target file, you must set the build controls for the toolchain approach. To learn more about toolchain approach for custom targets, see “Support Toolchain Approach with Custom Target” (Simulink Coder).

- 6 On the **Code Generation > Report** pane, select **Create code generation report** and **Open report automatically**.
- 7 On the **Code Generation > Interface** pane, set the **Target Library** in the **Deep learning** group to `cuDNN`. You can also select `TensorRT`.
- 8 Options specific to GPU Coder are in the **Code Generation > GPU Code** pane. For this example, you can use the default values of the GPU-specific parameters in **Code Generation > GPU Code** pane.



- 9 Click **OK** to save and close the Configuration Parameters dialog box.

You can also use `set_param` to configure the model parameter programmatically in the MATLAB Command Window.

```
set_param('googlenetModel', 'GenerateGPUCode', 'CUDA');
```

Generate CUDA Code for the Model

- 1 In the Simulink Editor, open the **Simulink Coder** app.
- 2 Generate code.

Messages appear in the Diagnostics Viewer. The code generator produces CUDA source and header files, and an HTML code generation report. The code generator places the files in a *build folder*, a subfolder named `googlenetModel_grt_rtw` under your current working folder.

Limitations

- GPU code generation for MATLAB Function blocks in Stateflow charts is not supported.
- The code generator does not support all the data types from the MATLAB language. For supported data types, refer to the block documentation.
- For GPU code generation, the custom target file must be based on `grt.tlc` or `ert.tlc`.
- For deploying the generated code, it is recommended to use the **Generate an example main program** option to generate the `ert_main.cu` module. This option requires the Embedded Coder license.

You can also use the `rt_cppclass_main.cpp` static main module provided by MathWorks. However, the static main file must be modified such that the models class constructor points to the deep learning object. For example,

```
static googlenetModelModelClass::DeepLearning_googlenetModel_T
    googlenetModel_DeepLearning;
static googlenetModelModelClass googlenetModel_Obj{ &googlenetModel_DeepLearning};
```

See Also

Functions

`open_system` | `load_system` | `save_system` | `close_system` | `bdclose` | `get_param` | `set_param` | `sim` | `slbuild`

More About

- “Simulation Acceleration by Using GPU Coder” on page 3-2
- “Code Generation from Simulink Models with GPU Coder” on page 3-8
- “GPU Code Generation for Deep Learning Networks Using MATLAB Function Block” on page 3-14
- “Targeting NVIDIA Embedded Boards” on page 3-30
- “Numerical Equivalence Testing” on page 3-32
- “Parameter Tuning and Signal Monitoring by Using External Mode” on page 3-38
- “Code Generation for a Deep Learning Simulink Model that Performs Lane and Vehicle Detection” on page 3-60
- “Code Generation for a Deep Learning Simulink Model to Classify ECG Signals” on page 3-53

Targeting NVIDIA Embedded Boards

With the MATLAB Coder Support Package for NVIDIA Jetson® and NVIDIA DRIVE Platforms, you can automate the deployment of Simulink models on embedded NVIDIA boards by building and deploying the generated code on the target hardware board. You can also remotely communicate with the target and control the peripheral devices for prototyping.

For an example of deployment to NVIDIA targets, see “Deploy and Classify Webcam Images on NVIDIA Jetson TX2 Platform from Simulink” (MATLAB Coder Support Package for NVIDIA Jetson and NVIDIA DRIVE Platforms).

Note Starting in R2021a, the MATLAB Coder Support Package for NVIDIA Jetson and NVIDIA DRIVE® Platforms is named MATLAB Coder Support Package for NVIDIA Jetson and NVIDIA DRIVE Platforms. To use this support package in R2021a, you must have the MATLAB Coder product.

Configure Model for Deployment

The model configuration parameters provide many options for the code generation and build process.

- 1 Open the Configuration Parameters dialog box. Select the **Hardware Implementation** pane. Set the **Hardware board** to NVIDIA Jetson. You can also use NVIDIA Drive.
- 2 Under **Target hardware resources** group, set the **Device Address**, **Username**, and **Password** of your target hardware. The device address is the IP address or host name of the target platform.
- 3 Click **OK** to save and close the Configuration Parameters dialog box.

You can also use `set_param` to configure the model parameter programmatically in the MATLAB Command Window.

```
set_param(<modelName>, 'HardwareBoard', 'NVIDIA Jetson');
```

Generate CUDA Code for the Model

- 1 Once the hardware parameters are set, in the Simulink Editor, open the **Hardware** tab.
- 2 Select **Build, Deploy & Start** to generate and deploy the code on the hardware.



See Also

Functions

`open_system` | `load_system` | `save_system` | `close_system` | `bdclose` | `get_param` | `set_param` | `sim` | `slbuild`

More About

- “Deploy and Classify Webcam Images on NVIDIA Jetson TX2 Platform from Simulink” (MATLAB Coder Support Package for NVIDIA Jetson and NVIDIA DRIVE Platforms)
- “Simulation Acceleration by Using GPU Coder” on page 3-2
- “Code Generation from Simulink Models with GPU Coder” on page 3-8
- “GPU Code Generation for Deep Learning Networks Using MATLAB Function Block” on page 3-14
- “GPU Code Generation for Blocks from the Deep Neural Networks Library” on page 3-22
- “Numerical Equivalence Testing” on page 3-32
- “Parameter Tuning and Signal Monitoring by Using External Mode” on page 3-38

Numerical Equivalence Testing

Test numerical equivalence between model components and production code that you generate from the components by using GPU acceleration and processor-in-the-loop (PIL) simulations.

With a GPU acceleration simulation, you test source code on your development computer. With a PIL simulation, you test the compiled object code that you intend to deploy on a target hardware by running the object code on real target hardware. To determine whether model components and generated code are numerically equivalent, compare GPU acceleration and PIL results to normal mode results.

Target Connectivity Configuration for PIL

Before you can run PIL simulations, you must configure target connectivity. The target connectivity configuration enables the PIL simulation to:

- Build the target application.
- Download, start, and stop the application on the target.
- Support communication between Simulink and the target.

To produce a target connectivity configuration for hardware platforms such as NVIDIA DRIVE and Jetson, install the MATLAB Coder Support Package for NVIDIA Jetson and NVIDIA DRIVE Platforms.

Note Starting in R2021a, the MATLAB Coder Support Package for NVIDIA Jetson and NVIDIA DRIVE Platforms is named MATLAB Coder Support Package for NVIDIA Jetson and NVIDIA DRIVE Platforms. To use this support package in R2021a, you must have the MATLAB Coder product.

Target Board Requirements

- NVIDIA DRIVE or Jetson embedded platform.
- Ethernet crossover cable to connect the target board and host PC (if you cannot connect the target board to a local network).
- NVIDIA CUDA Toolkit installed on the board.
- Environment variables on the target for the compilers and libraries. For information on the supported versions of the compilers, libraries, and their setup, see “Install and Setup Prerequisites for NVIDIA Boards” (MATLAB Coder Support Package for NVIDIA Jetson and NVIDIA DRIVE Platforms).

Create Live Hardware Connection Object

The support package software uses an SSH connection over TCP/IP to execute commands while building and running the generated CUDA code on the DRIVE or Jetson platforms. Connect the target platform to the same network as the host computer or use an Ethernet crossover cable to connect the board directly to the host computer. For how to set up and configure your board, see NVIDIA documentation.

To communicate with the NVIDIA hardware, create a live hardware connection object by using the `jetson` or `drive` function. To create a live hardware connection object by using the function, provide the host name or IP address, user name, and password of the target board. For example, to create live object for Jetson hardware:

```
hwobj = jetson('192.168.1.15','ubuntu','ubuntu');
```

The software performs a check of the hardware, compiler tools, libraries, IO server installation, and gathers peripheral information on target. This information is displayed in the Command Window.

```
Checking for CUDA availability on the Target...
Checking for NVCC in the target system path...
Checking for CUDNN library availability on the Target...
Checking for TensorRT library availability on the Target...
Checking for Prerequisite libraries is now complete.
Fetching hardware details...
Fetching hardware details is now complete. Displaying details.
Board name      : NVIDIA Jetson TX2
CUDA Version    : 9.0
cuDNN Version   : 7.0
TensorRT Version : 3.0
Available Webcams : UVC Camera (046d:0809)
Available GPUs   : NVIDIA Tegra X2
```

Alternatively, to create live object for DRIVE hardware:

```
hwobj = drive('92.168.1.16','nvidia','nvidia');
```

Note If there is a connection failure, a diagnostics error message is reported on the MATLAB command window. If the connection has failed, the most likely cause is incorrect IP address or host name.

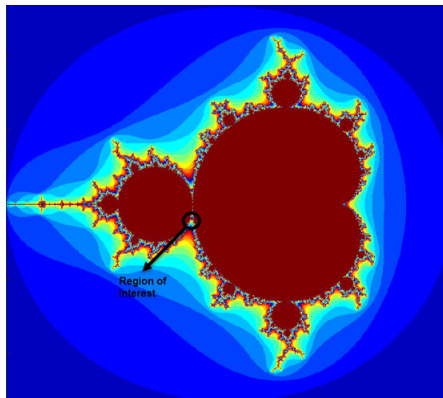
Example: The Mandelbrot Set

Description

The Mandelbrot set is the region in the complex plane consisting of the values z_0 for which the trajectories defined by this equation remain bounded at $k \rightarrow \infty$.

$$z_{k+1} = z_k^2 + z_0, \quad k = 0, 1, \dots$$

The overall geometry of the Mandelbrot set is shown in the figure. This view does not have the resolution to show the richly detailed structure of the fringe just outside the boundary of the set. At increasing magnifications, the Mandelbrot set exhibits an elaborate boundary that reveals progressively finer recursive detail.



Algorithm

For this tutorial, pick a set of limits that specify a highly zoomed part of the Mandelbrot set in the valley between the main cardioid and the p/q bulb to its left. A 1000-by-1000 grid of real parts (x) and imaginary parts (y) is created between these two limits. The Mandelbrot algorithm is then iterated at each grid location. An iteration number of 500 renders the image in full resolution.

```
maxIterations = 500;
gridSize = 1000;
xlim = [-0.748766713922161, -0.748766707771757];
ylim = [0.123640844894862, 0.123640851045266];
```

This tutorial uses an implementation of the Mandelbrot set by using standard MATLAB commands running on the CPU. This calculation is vectorized such that every location is updated simultaneously.

GPU Acceleration or PIL Simulation with a Top Model

Test the generated model code by running a top-model PIL simulation. With this approach:

- You test code generated from the top model, which uses the standalone code interface.
- You configure the model to load test vectors or stimulus inputs from the MATLAB workspace.
- You can easily switch the top model between the normal, GPU acceleration, and PIL simulation modes.

Create Mandelbrot Top Model

- 1 Create a Simulink model and insert a MATLAB Function block from the **User-Defined Functions** library.
- 2 Double-click the MATLAB Function block. A default function signature appears in the MATLAB Function Block Editor.
- 3 Define a function called `mandelbrot_count`, which implements the Mandelbrot algorithm. The function header declares `maxIterations`, `xGrid`, and `yGrid` as an argument to the `mandelbrot_count` function, with `count` as the return value.

```
function count = mandelbrot_count(maxIterations, xGrid, yGrid)
% mandelbrot computation

z0 = xGrid + 1i*yGrid;
count = ones(size(z0));

% Map computation to GPU
coder.gpu.kernelfun;

z = z0;
for n = 0:maxIterations
    z = z.*z + z0;
    inside = abs(z)<=2;
    count = count + inside;
end
count = log(count);
```

- 4 Open the block parameters for the MATLAB Function block. On the **Code Generation** tab, select Reusable function for **Function packaging** parameter.

If the **Function packaging** parameter is set to any other value, CUDA kernels may not get generated.

- 5 Add Inport blocks and Outport block from the **Sources** and **Sinks** library.
- 6 Connect these blocks as shown in the diagram. Save the model as `mandelbrot_top.slx`.



Configure the Model for GPU Acceleration

To focus on numerical equivalence testing, turn off:

- Model coverage
- Code coverage
- Execution time profiling

```
model = 'mandelbrot_top';
close_system(model,0);
open_system(model)
set_param(gcs, 'RecordCoverage','off');
coverageSettings = get_param(model, 'CodeCoverageSettings');
coverageSettings.CoverageTool='None';
set_param(model, 'CodeCoverageSettings',coverageSettings);
set_param(model, 'CodeExecutionProfiling','off');
```

Configure the input stimulus data. The following lines of code generate a 1000-by-1000 grid of real parts (x) and imaginary parts (y) between the limits specified by `xlim` and `ylim`.

```
gridSize = 1000;
xlim = [-0.748766713922161, -0.748766707771757];
ylim = [ 0.123640844894862, 0.123640851045266];
x = linspace( xlim(1), xlim(2), gridSize );
y = linspace( ylim(1), ylim(2), gridSize );
[xG, yG] = meshgrid( x, y );
maxIterations = timeseries(500,0);
xGrid = timeseries(xG,0);
yGrid = timeseries(yG,0);
```

Configure logging options in the model.

```
set_param(model, 'LoadExternalInput','on');
set_param(model, 'ExternalInput','maxIterations, xGrid, yGrid');
set_param(model, 'SignalLogging', 'on');
set_param(model, 'SignalLoggingName', 'logsOut');
set_param(model, 'SaveOutput','on')
```

Run Normal and PIL Simulations

Run a normal mode simulation.

```
set_param(model,'SimulationMode','normal')
set_param(model,'GPUAcceleration','on');
sim_output = sim(model,10);
count_normal = sim_output.yout{1}.Values.Data(:, :, 1);
```

Run a top-model PIL simulation.

```
set_param(model, 'SimulationMode', 'Processor-in-the-Loop (PIL)')
sim_output = sim(model,10);
count_pil = sim_output.yout{1}.Values.Data(:,:,1);
```

```
### Target device has no native communication support.
Checking connectivity configuration registrations...
### Starting build procedure for: mandelbrot_top
### Generating code and artifacts to 'Model specific' folder structure
### Generating code into build folder:
/mathworks/examples/sil_pil/mandelbrot_top_ert_rtw
### Generated code for 'mandelbrot_top' is up to date because no structural,
parameter or code replacement library changes were found.
### Evaluating PostCodeGenCommand specified in the model
### Using toolchain: NVCC for NVIDIA Embedded Processors
### '/mathworks/examples/sil_pil/mandelbrot_top_ert_rtw/mandelbrot_top.mk' is
up to date
### Building 'mandelbrot_top': make -f mandelbrot_top.mk buildobj
### Successful completion of build procedure for: mandelbrot_top
```

Build Summary

Top model targets built:

Model	Action	Rebuild Reason
mandelbrot_top	Code compiled	Compilation artifacts were out of date.

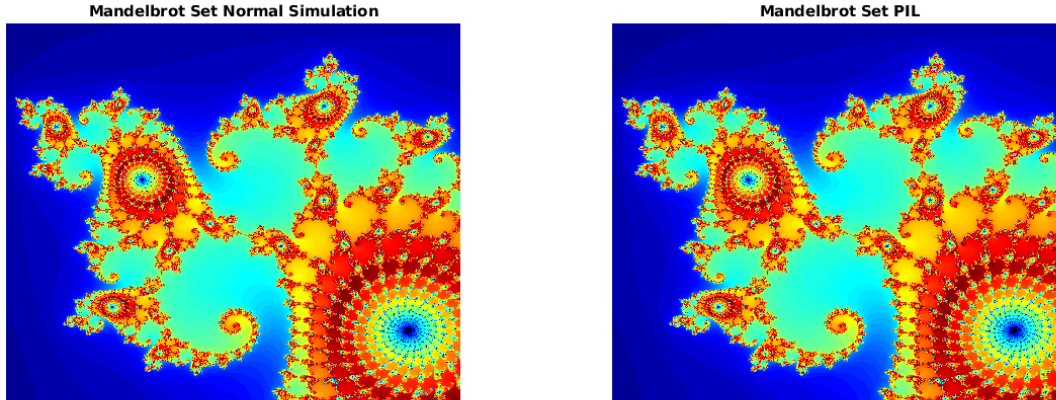
```
1 of 1 models built (0 models already up to date)
Build duration: 0h 0m 22.94s
### Target device has no native communication support. Checking connectivity
configuration registrations...
### Connectivity configuration for component "mandelbrot_top": NVIDIA Jetson ###
PIL execution is using Port 17725.
PIL execution is using 30 Sec(s) for receive time-out.
### Preparing to start PIL simulation ...
### Using toolchain: NVCC for NVIDIA Embedded Processors
### '/mathworks/examples/sil_pil/mandelbrot_top_ert_rtw/pil/mandelbrot_top.mk' is
up to date
### Building 'mandelbrot_top': make -f mandelbrot_top.mk all
### Starting application: 'mandelbrot_top_ert_rtw/pil/mandelbrot_top.elf'
### Launching application mandelbrot_top.elf...
PIL execution terminated on target.
```

Unless up-to-date code for this model exists, new code is generated and compiled. The generated code runs as a separate process on your computer.

Plot and compare the results of the normal and PIL simulations. Observe that the results match.

```
figure();
subplot(1,2,1)
imagesc(x, y, count_normal);
colormap([jet();flipud( jet() );0 0 0]);
title('Mandelbrot Set Normal Simulation');
axis off;

subplot(1,2,2)
imagesc(x, y, count_pil);
colormap([jet();flipud( jet() );0 0 0]);
title('Mandelbrot Set PIL');
axis off;
```



Clean up.

```
close_system(model,0);
if ishandle(fig1), close(fig1), end
clear fig1
simResults = {'count_sil','count_normal','model'};
save([model '_results'],simResults{:});
clear(simResults{:},'simResults')
```

Limitations

MAT-file logging is not supported for Processor-in-the-loop (PIL) simulation with GPU Coder.

See Also

Functions

open_system | load_system | save_system | close_system | bdclose | get_param | set_param | sim | slbuild

More About

- “Simulation Acceleration by Using GPU Coder” on page 3-2
- “Code Generation from Simulink Models with GPU Coder” on page 3-8
- “GPU Code Generation for Deep Learning Networks Using MATLAB Function Block” on page 3-14
- “GPU Code Generation for Blocks from the Deep Neural Networks Library” on page 3-22
- “Targeting NVIDIA Embedded Boards” on page 3-30
- “Parameter Tuning and Signal Monitoring by Using External Mode” on page 3-38

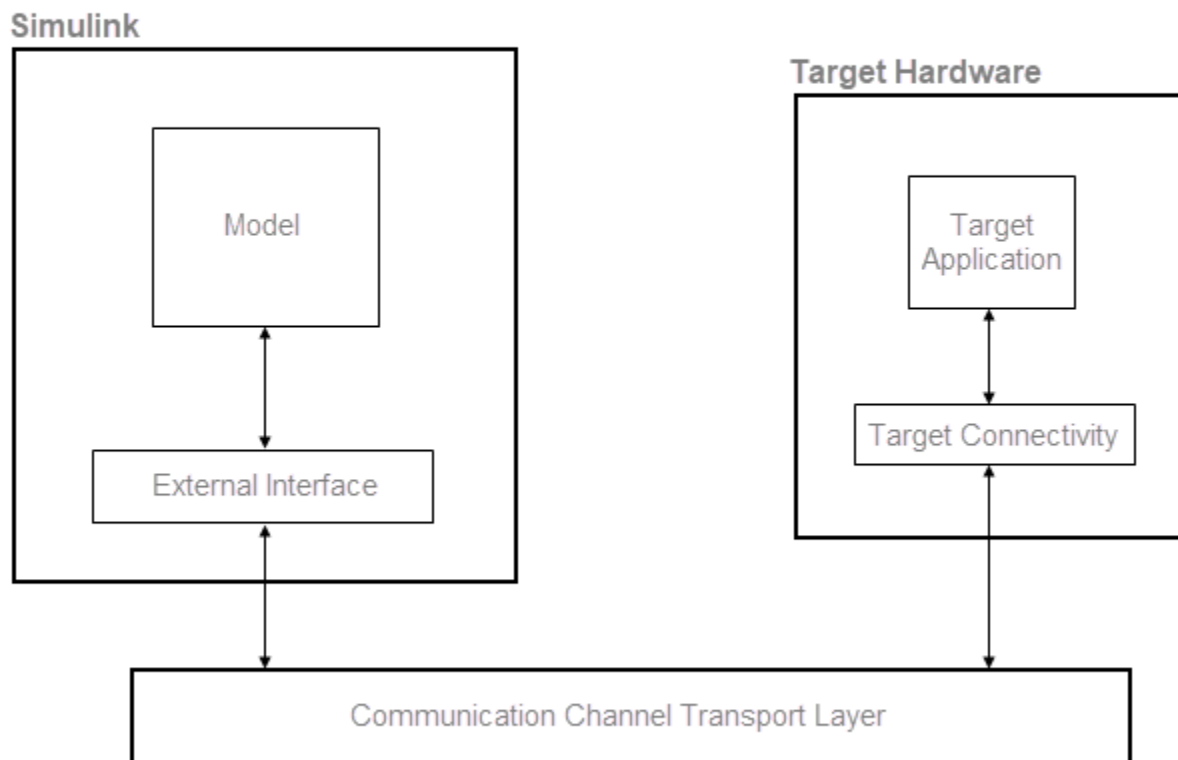
Parameter Tuning and Signal Monitoring by Using External Mode

You can use external mode simulations for rapid prototyping. An external mode simulation establishes a communication channel between Simulink on your development computer (host) and the target hardware that runs the executable file created by the code generation and build process.

Through the communication channel, you can:

- Modify or tune block parameters in real time. When you change parameters in your model, Simulink downloads the new values to the executing target application.
- Monitor and save signal data from the executing target application.

The low-level transport layer of the channel handles the transmission of messages. Simulink and the generated model code are independent of this layer. The transport layer and its interface code are isolated in separate modules that format, transmit, and receive messages and data packets.



Set up and run an external mode simulation that uses a TCP/IP or serial (RS-232) communication channel.

- 1 Create and configure a simple model.
- 2 Build the target executable file.
- 3 Run the target application.
- 4 Tune parameters.

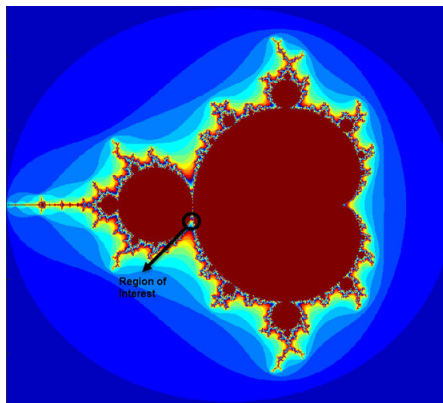
Example: The Mandelbrot Set

Description

The Mandelbrot set is the region in the complex plane consisting of the values z_0 for which the trajectories defined by this equation remain bounded at $k \rightarrow \infty$.

$$z_{k+1} = z_k^2 + z_0, \quad k = 0, 1, \dots$$

The overall geometry of the Mandelbrot set is shown in the figure. This view does not have the resolution to show the richly detailed structure of the fringe just outside the boundary of the set. At increasing magnifications, the Mandelbrot set exhibits an elaborate boundary that reveals progressively finer recursive detail.



Algorithm

For this tutorial, pick a set of limits that specify a highly zoomed part of the Mandelbrot set in the valley between the main cardioid and the p/q bulb to its left. A 1000-by-1000 grid of real parts (x) and imaginary parts (y) is created between these two limits. The Mandelbrot algorithm is then iterated at each grid location. An iteration number of 500 renders the image in full resolution.

```
maxIterations = 500;
gridSize = 1000;
xlim = [-0.748766713922161, -0.748766707771757];
ylim = [0.123640844894862, 0.123640851045266];
```

This tutorial uses an implementation of the Mandelbrot set by using standard MATLAB commands running on the CPU. This calculation is vectorized such that every location is updated simultaneously.

Create Mandelbrot Model

- 1 Create a Simulink model and insert a MATLAB Function block from the **User-Defined Functions** library.
- 2 Double-click the MATLAB Function block. A default function signature appears in the MATLAB Function Block Editor.
- 3 Define a function called `mandelbrot_count`, which implements the Mandelbrot algorithm. The function header declares `maxIterations`, `xGrid`, and `yGrid` as an argument to the `mandelbrot_count` function, with `count` as the return value.

```
function count = mandelbrot_count(maxIterations, xGrid, yGrid)
% mandelbrot computation

z0 = xGrid + 1i*yGrid;
count = ones(size(z0));

% Map computation to GPU
coder.gpu.kernelfun;

z = z0;
for n = 0:maxIterations
    z = z.*z + z0;
    inside = abs(z)<=2;
    count = count + inside;
end
count = log(count);
```

- 4 Open the block parameters for the MATLAB Function block. On the **Code Generation** tab, select **Reusable function** for **Function packaging** parameter.

If the **Function packaging** parameter is set to any other value, CUDA kernels may not get generated.

- 5 Add Inport blocks and Outport block from the **Sources** and **Sinks** library.
- 6 Connect these blocks as shown in the diagram. Save the model as `mandelbrot_top.slx`.



Build Target Executable

Set up the model and code generation parameters required for an external mode target application. Then, generate code and build the target application.

- 1 From the **Apps** tab on the Simulink toolstrip, in the **Setup to Run on Hardware** section, click **Run on Hardware Board**.
- 2 In the **Hardware Board** section, from the **Hardware Board** list, select **NVIDIA Jetson**.
- 3 In the **Prepare** section, click **Hardware Settings**. The Configuration Parameters dialog box opens, displaying **Hardware Implementation** settings that are determined by the selected board.
- 4 On the **Solver** pane:

- a In the **Type** field, select **Fixed-step**.
 - b In the **Solver** field, select **discrete (no continuous states)**.
 - c Click **Solver details**. In the **Fixed-step size** field, specify **0.1**. (Otherwise, when you generate code, the GPU Coder build process produces a warning and supplies a value.)
 - d Click **Apply**.
- 5 On the **Data Import/Export** pane, clear the **Time** and **Output** check boxes. In this example, data is not logged to the workspace or to a MAT-file. Click **Apply**.
 - 6 On the **Code Generation > Optimization** pane, make sure that **Default parameter behavior** is set to **Tunable**. If you make a change, click **Apply**.
 - 7 On the **Code Generation > Interface** pane, in the **Data exchange interface** section, select **External mode**.
 - 8 In the **External mode configuration** section, make sure that you select the default value **tcpip** for the **Transport layer** parameter.

External mode

External mode configuration

Transport layer: MEX-file name:

MEX-file arguments:

Static memory allocation

The **MEX-file name** specifies the name of a MEX-file that implements host-target communication. The default for TCP/IP is `ext_comm`, a MEX-file provided with the Simulink Coder software.

The **MEX-file arguments** field enables you specify arguments, such as a TCP/IP server port number, to be passed to the external interface program. These arguments are specific to the external interface that you are using.

This tutorial uses the default arguments. Leave the **MEX-file arguments** field blank.

The **Static memory allocation** check box controls how memory is allocated for external mode communication buffers in the target. For this tutorial, do not select the check box.

- 9 Click **Apply** to save the external mode settings.
- 10 Save the model.
- 11 Select the **Code Generation** pane. Make sure that **Generate code only** is cleared.
- 12 To generate code and create the target application, in the model window, press **Ctrl+B**. Or, on the **Hardware** tab, in the **Run on Hardware** section, click **Monitor & Tune**. Then, under **Step By Step Commands**, click **Build for Monitoring**.

The software creates the `mandelbrot_top` executable file in your working folder.

Run Target Application


Run the `mandelbrot_top` target executable and use Simulink as an interactive front end to the running target application. The executable file is in your working folder. Run the target application and establish communication between Simulink and the target.

To run the target application:

- 1 On the **Hardware** tab, in the **Run on Hardware** section:
 - a In the **Stop Time** field, specify `inf`, which makes the model run until the target application receives a stop message from Simulink
 - b Click **Monitor & Tune**. Then, under **Step By Step Commands**, click **Deploy**.

The target application begins execution, and enters a wait state.

- 2 On the **Hardware** tab, in the **Run on Hardware** section, click **Monitor & Tune**. Then, under **Step By Step Commands**, click **Connect**. When Simulink and the target are connected, the **Connect** button changes to **Disconnect**.

- 3 In the **Run on Hardware** section, click , which starts execution of the generated model code.

You have established communication between Simulink and the running target application.

Note When performing external mode simulation on Simulink models containing deep learning networks, a timeout error may occur during model initialization on the target. This timeout may be because the initialization time for the executable exceeds the default maximum loading time of 300 seconds. You can increase the timeout by using the `NVIDIA_XCP_EXTMODE_INIT_TIME` environment variable. For example, in the MATLAB Command Window, enter:

```
setenv('NVIDIA_XCP_EXTMODE_INIT_TIME', '500');
```

Stop Target Application

To simultaneously disconnect Simulink from the host/target communication and end execution of the target application, on the **Hardware** tab, in the **Run on Hardware** section, click **Stop**.

See Also

Functions

`open_system` | `load_system` | `save_system` | `close_system` | `bdclose` | `get_param` | `set_param` | `sim` | `slbuild`

More About

- “Simulation Acceleration by Using GPU Coder” on page 3-2
- “Code Generation from Simulink Models with GPU Coder” on page 3-8
- “GPU Code Generation for Deep Learning Networks Using MATLAB Function Block” on page 3-14
- “GPU Code Generation for Blocks from the Deep Neural Networks Library” on page 3-22
- “Targeting NVIDIA Embedded Boards” on page 3-30
- “Numerical Equivalence Testing” on page 3-32

GPU Code Generation for Lane Detection in Simulink

This example shows how to generate CUDA® code for a Simulink® model that can detect and output lane marker boundaries on an image. This example takes RGB image as an input and uses the `imresize`, `rgb2gray`, `ordfilt2` (Image Processing Toolbox), `hough` (Image Processing Toolbox), `houghpeaks` (Image Processing Toolbox), and `houghlines` (Image Processing Toolbox) functions that are part of Image Processing Toolbox™ to detect lane markings. This example closely follows “Lane Detection on the GPU by Using the `houghlines` Function” on page 2-100.

This example illustrates the following concepts:

- Model a lane detection application in Simulink by using image processing functions.
- Configure the model for GPU code generation.
- Generate a CUDA executable for the Simulink model.

Third-Party Prerequisites

Required

This example generates CUDA MEX and has the following third-party requirements.

- CUDA enabled NVIDIA® GPU and compatible driver.

Optional

For non-MEX builds such as static, dynamic libraries or executables, this example has the following additional requirements.

- NVIDIA toolkit.
- Environment variables for the compilers and libraries. For more information, see “Third-Party Hardware” and “Setting Up the Prerequisite Products”.

Verify GPU Environment

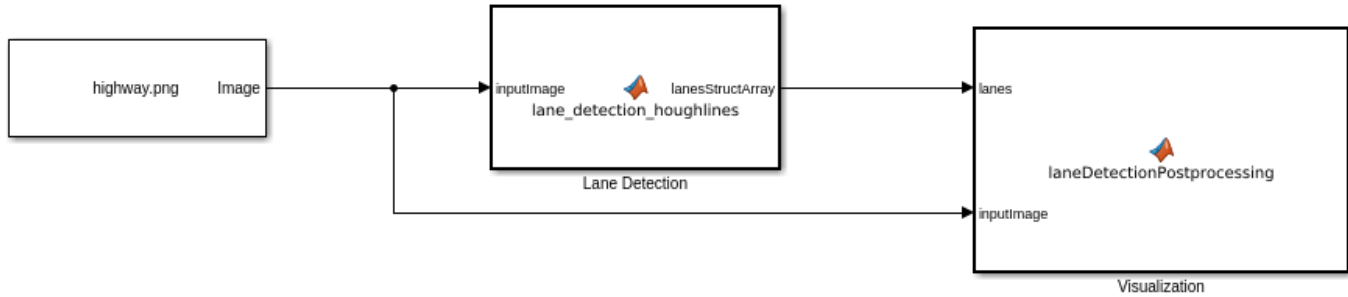
To verify that the compilers and libraries necessary for running this example are set up correctly, use the `coder.checkGpuInstall` function.

```
envCfg = coder.gpuEnvConfig('host');
envCfg.BasicCodegen = 1;
envCfg.Quiet = 1;
coder.checkGpuInstall(envCfg);
```

Lane Detection using `houghlines` Simulink Model

The Simulink model for lane detection is shown.

```
open_system('lane_detection');
```



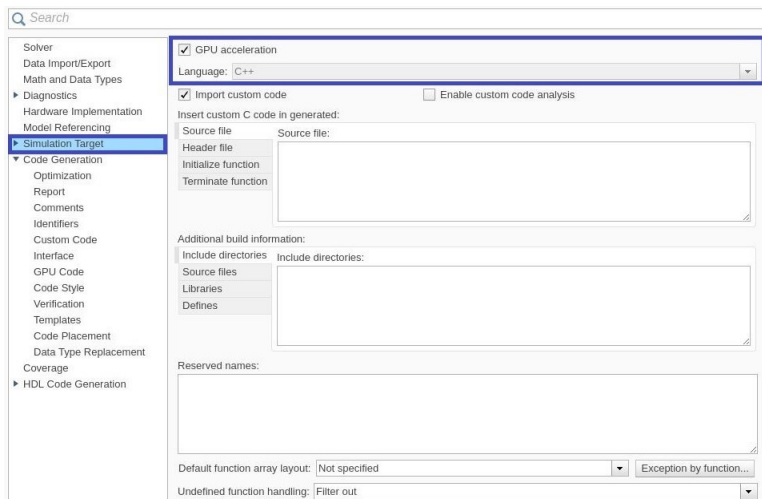
Copyright 2020-2021 The MathWorks, Inc.

The Lane Detection subsystem contains a MATLAB Function block that takes an intensity image as input and provides detected lanes as output. This function is based on the lane detection algorithm implementation using `houghlines` as described in “Lane Detection on the GPU by Using the houghlines Function” on page 2-100 example. When the model runs, the Visualization block displays the lane detected output image.

Run the Simulation

Open Configuration Parameters dialog box.

In **Simulation Target** pane, select **GPU acceleration**.



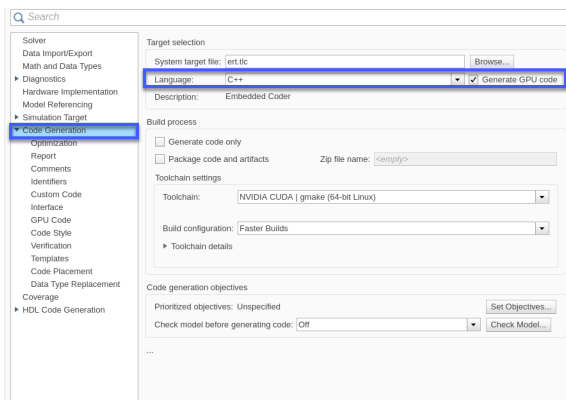
Run the simulation in Normal mode.

```
set_param('lane_detection', 'SimulationMode', 'Normal');
sim('lane_detection');
```

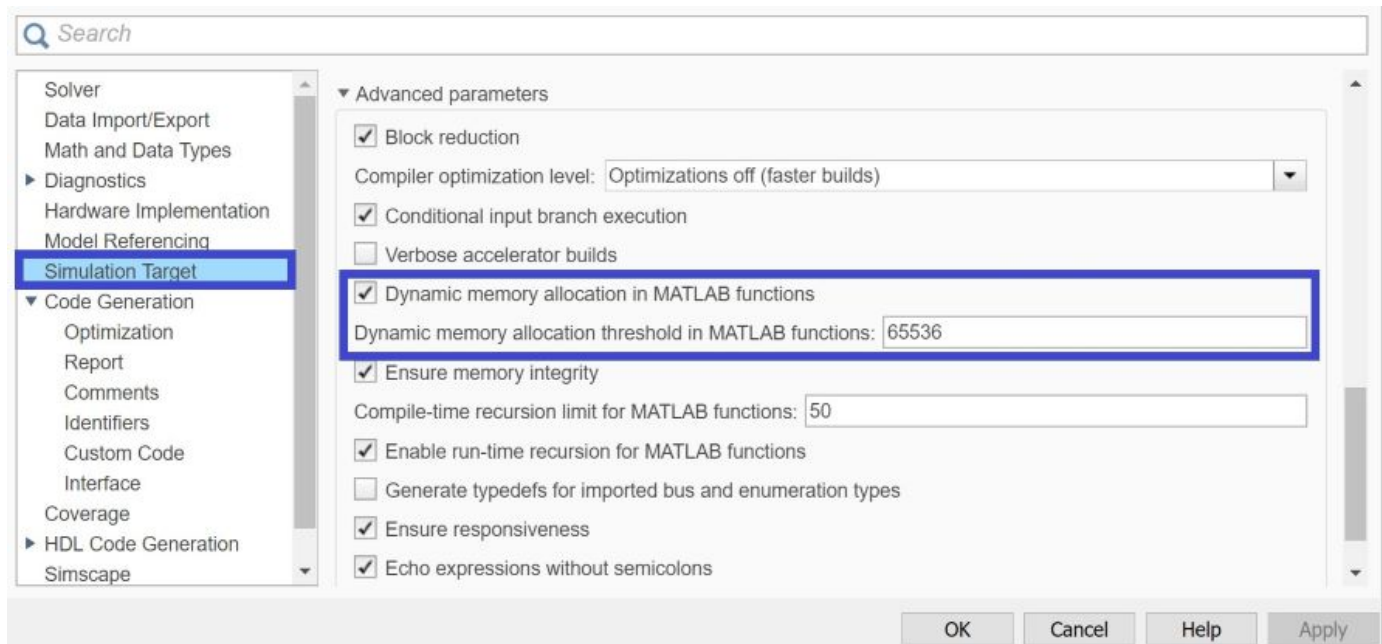


Generate and Build the Simulink Model

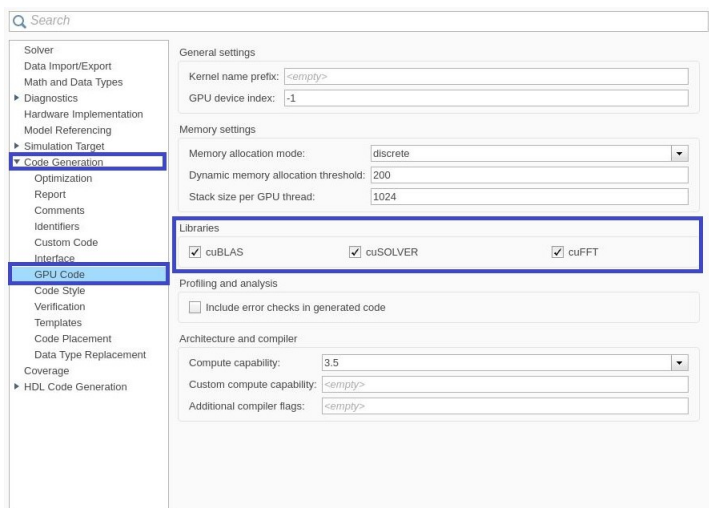
In **Code Generation** pane, select the **Language** as **C++** and enable **Generate GPU code**.



Open **Simulation Target** pane. In the **Advanced parameters**, enable **Dynamic memory allocation threshold in MATLAB functions**. For more information, see “Dynamic memory allocation in MATLAB functions” (Simulink)



Open **Code Generation > GPU Code** pane. In the subcategory **Libraries**, enable **cuBLAS**, **cuSOLVER** and **cuFFT**.



Generate and build the Simulink model on the host GPU by using the `slbuild` command. The code generator places the files in a *build folder*, a subfolder named `lane_detection_ert_rtw` under your current working folder.

```
status = evalc("slbuild('lane_detection')");
```

Cleanup

Close the Simulink model.

```
close_system('lane_detection');
```

See Also

Functions

[open_system](#) | [load_system](#) | [save_system](#) | [close_system](#) | [bdclose](#) | [get_param](#) | [set_param](#) | [sim](#) | [slbuild](#)

More About

- “Simulation Acceleration by Using GPU Coder” on page 3-2
- “Code Generation from Simulink Models with GPU Coder” on page 3-8
- “GPU Code Generation for Deep Learning Networks Using MATLAB Function Block” on page 3-14
- “Targeting NVIDIA Embedded Boards” on page 3-30
- “Numerical Equivalence Testing” on page 3-32
- “Parameter Tuning and Signal Monitoring by Using External Mode” on page 3-38

GPU Code Generation for a Fog Rectification Simulink Model

This example demonstrates how to generate CUDA® code from the Simulink® model that takes a foggy image as input and produces a defogged image as output. This example is a typical implementation of fog rectification algorithm. The example uses `conv2`, `im2gray`, and `imhist` (Image Processing Toolbox) functions. This example closely follows “Fog Rectification” on page 2-80 example. This example illustrates the following concepts:

- Verification of GPU Environment.
- Model fog rectification application in Simulink by using image processing functions.
- Configure the model for GPU code generation.
- Generate a CUDA executable for the Simulink model.

Third-Party Prerequisites

Required

This example generates CUDA MEX and has the following third-party requirements.

- CUDA enabled NVIDIA® GPU and compatible driver.

Optional

For non-MEX builds such as static, dynamic libraries or executables, this example has the following additional requirements.

- NVIDIA toolkit.
- Environment variables for the compilers and libraries. For more information, see “Third-Party Hardware” and “Setting Up the Prerequisite Products”.

Verify GPU Environment

To verify that the compilers and libraries necessary for running this example are set up correctly, use the `coder.checkGpuInstall` function.

```
envCfg = coder.gpuEnvConfig('host');  
envCfg.BasicCodegen = 1;  
envCfg.Quiet = 1;  
coder.checkGpuInstall(envCfg);
```

Fog Rectification Simulink Model

The Simulink model for fog rectification consists of Fog Rectification subsystem that contains a MATLAB Function block which takes a foggy image as input and returns a defogged image as output. It uses fog_rectification algorithm described in “Fog Rectification” on page 2-80 example. When the model runs, the Visualization block displays the foggy input image and defogged output image.

```
mdl = 'fog_rectification_model';  
open_system(mdl);
```

?

GPU Code Generation for a Fog Rectification Simulink Model



Copyright 2020-2022 The Mathworks, Inc.

Configure Model for GPU Acceleration

Model configuration parameters determine the acceleration method used during simulation.

```

set_param mdl, 'Solver', 'FixedStepAuto');
set_param mdl, 'GPUAcceleration', 'on');
set_param mdl, 'SimulationMode', 'Normal');
  
```

Build GPU Accelerated Model

To build and simulate the GPU accelerated model, select **Run** on the **Simulation** tab or use the following MATLAB command:

```

out = sim(mdl);
  
```

Foggy Input Image



Defogged Output Image



Foggy Input Image**Defogged Output Image**

Configure Model for Code Generation

Set the following parameters for code generation.

```
set_param mdl, 'TargetLang', 'C++';  
set_param mdl, 'GenerateGPUCode', 'CUDA';  
set_param mdl, 'GPUcuBLAS', 'on';  
set_param mdl, 'GPUcuSOLVER', 'on';  
set_param mdl, 'GPUcuFFT', 'on';  
set_param mdl, 'ProdLongLongMode', 'on';
```

Generate CUDA Code for the Model

Generate and build the Simulink model on the host GPU by using the `slbuild` command. The code generator places the files in a *build folder*, a subfolder named `fog_rectification_model_ert_rtw` under your current working folder.

```
status = evalc("slbuild('fog_rectification_model')");
```

Cleanup

Close the Simulink model.

```
close_system('fog_rectification_model');
```

See Also

Functions

[open_system](#) | [load_system](#) | [save_system](#) | [close_system](#) | [bdclose](#) | [get_param](#) | [set_param](#) | [sim](#) | [slbuild](#)

More About

- “Simulation Acceleration by Using GPU Coder” on page 3-2
- “Code Generation from Simulink Models with GPU Coder” on page 3-8
- “GPU Code Generation for Deep Learning Networks Using MATLAB Function Block” on page 3-14
- “Targeting NVIDIA Embedded Boards” on page 3-30
- “Numerical Equivalence Testing” on page 3-32
- “Parameter Tuning and Signal Monitoring by Using External Mode” on page 3-38

Code Generation for a Deep Learning Simulink Model to Classify ECG Signals

This example demonstrates how you can use powerful signal processing techniques and Convolutional Neural Networks together to classify ECG signals. We will also showcase how CUDA® code can be generated from the Simulink® model. This example uses the pretrained CNN network from the *Classify Time Series Using Wavelet Analysis and Deep Learning* example of the Wavelet Toolbox™ to classify ECG signals based on images from the CWT of the time series data. For information on training, see “Classify Time Series Using Wavelet Analysis and Deep Learning” (Wavelet Toolbox).

For a video demonstration on how to perform software-in-the-loop (SIL), processor-in-the-loop (PIL) simulation, and deploying this example to NVIDIA Jetson® board, see <https://www.mathworks.com/videos/deep-learning-in-simulink-for-nvidia-gpus-classification-of-ecg-signals-1621401016961.html>.

This example illustrates the following concepts:

- Model the classification application in Simulink. Use **MATLAB Function** blocks to perform preprocessing and wavelet transforms of the ECG data. Use the **Image Classifier** block from the Deep Learning Toolbox™ for loading the pretrained network and performing the classification of the ECG data.
- Configure the model for code generation.
- Generate a CUDA executable for the Simulink model.

Third-Party Prerequisites

- CUDA enabled NVIDIA GPU.
- NVIDIA CUDA toolkit and driver.
- NVIDIA cuDNN library.
- Environment variables for the compilers and libraries. For more information, see “Third-Party Hardware” and “Setting Up the Prerequisite Products”.

Verify GPU Environment

To verify that the compilers and libraries necessary for running this example are set up correctly, use the `coder.checkGpuInstall` function.

```
envCfg = coder.gpuEnvConfig('host');  
envCfg.DeepLibTarget = 'cudnn';  
envCfg.DeepCodegen = 1;  
envCfg.Quiet = 1;  
coder.checkGpuInstall(envCfg);
```

ECG Data Description

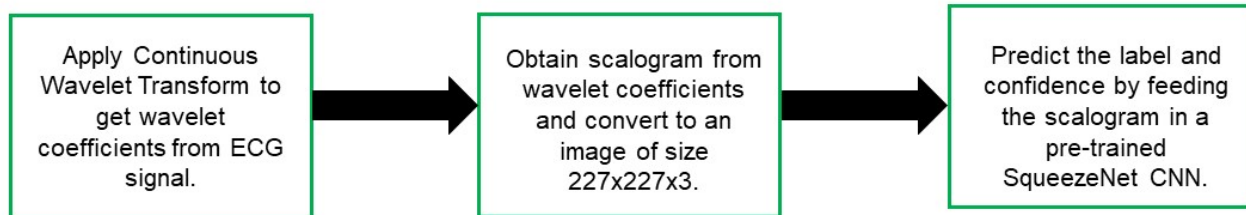
This example uses ECG data from PhysioNet database. It contains data from three groups of people:

- 1 Persons with cardiac arrhythmia (ARR)
- 2 Persons with congestive heart failure (CHF)
- 3 Persons with normal sinus rhythms (NSR)

It includes 96 recordings from persons with ARR, 30 recordings from persons with CHF, and 36 recordings from persons with NSR. The `ecg_signals` MAT-file contains the test ECG data in time series format. The image classifier in this example distinguishes between ARR, CHF, and NSR.

Algorithmic Workflow

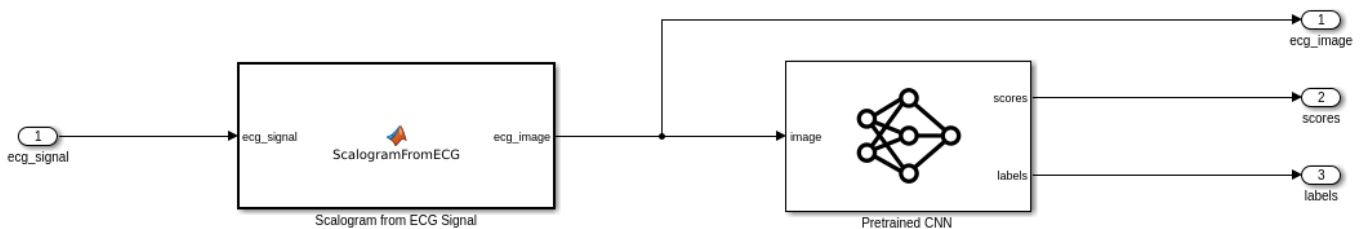
The block diagram for the algorithmic workflow of the Simulink model is shown.

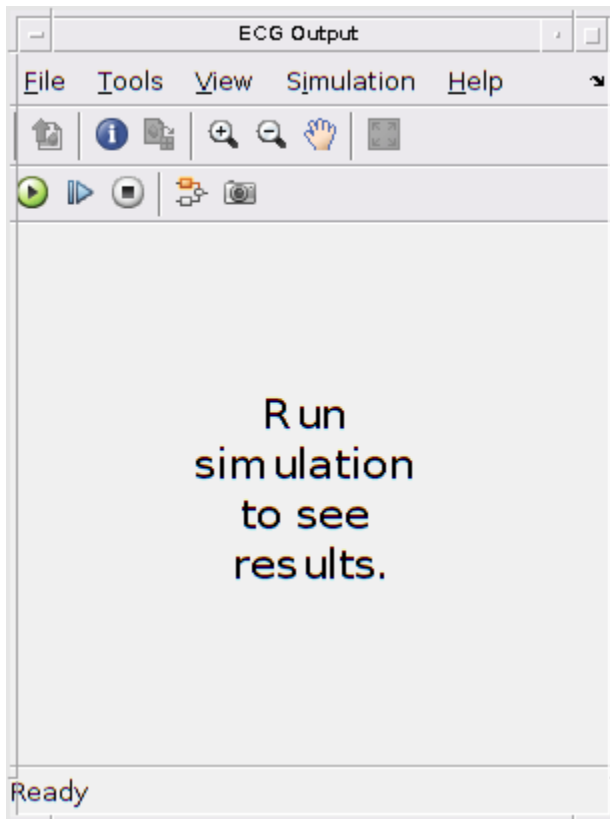


ECG Deep Learning Simulink Model

The Simulink model for classifying the ECG signals is shown. When the model runs, the Video Viewer block displays the classified ECG signal.

```
open_system('ecg_dl_cwt');
```





ECG Preprocessing Subsystem

The ECG Preprocessing subsystem contains a MATLAB Function block that performs CWT to obtain scalogram of the ECG signal and then processes the scalogram to obtain an image and an Image Classifier block that loads the pretrained network from `trainedNet.mat` and performs prediction for image classification based on SqueezeNet deep learning CNN.

```
open_system('ecg_dl_cwt/ECG Preprocessing');
```

The ScalogramFromECG function block defines a function called `ecg_to_scalogram` that:

- Uses 65536 samples of double-precision ECG data as input.
- Create time frequency representation from the ECG data by applying Wavelet transform.
- Obtain scalogram from the wavelet coefficients.
- Convert the scalogram to image of size (227x227x3).

The function signature of `ecg_to_scalogram` is shown.

```
type ecg_to_scalogram
```

```
function ecg_image = ecg_to_scalogram(ecg_signal)
```

```
% Copyright 2020 The MathWorks, Inc.
```

```
persistent jetdata;
if(isempty(jetdata))
```

```
    jetdata = ecgColorMap(128,'single');
end
% Obtain wavelet coefficients from ECG signal
cfs = cwt_ecg(ecg_signal);
% Obtain scalogram from wavelet coefficients
image = ind2rgb(im2uint8(rescale(cfs)),jetdata);
ecg_image = im2uint8(imresize(image,[227,227]));

end
```

ECG Postprocessing

The ECG Postprocessing MATLAB function block defines the `label_prob_image` function that finds the label for the scalogram image based on the highest score from the scores outputted by the image classifier. It outputs the scalogram image with the label and confidence printed on it.

type `label_prob_image`

```
function final_image = label_prob_image(ecg_image, scores, labels)

% Copyright 2020-2021 The MathWorks, Inc.

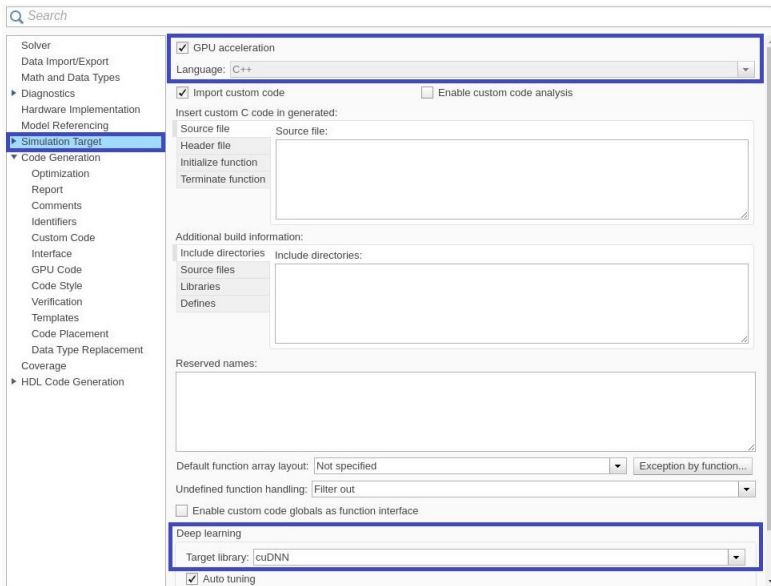
scores = double(scores);
% Obtain maximum confidence
[prob,index] = max(scores);
confidence = prob*100;
% Obtain label corresponding to maximum confidence
label = erase(char(labels(index)),'_label');
text = cell(2,1);
text{1} = ['Classification: ' label];
text{2} = ['Confidence: ' sprintf('%0.2f',confidence) '%'];
position = [135 20 0 0; 130 40 0 0];
final_image = insertObjectAnnotation(ecg_image,'rectangle',position,...
    text,'TextBoxOpacity',0.9,'FontSize',9);

end
```

Run the Simulation

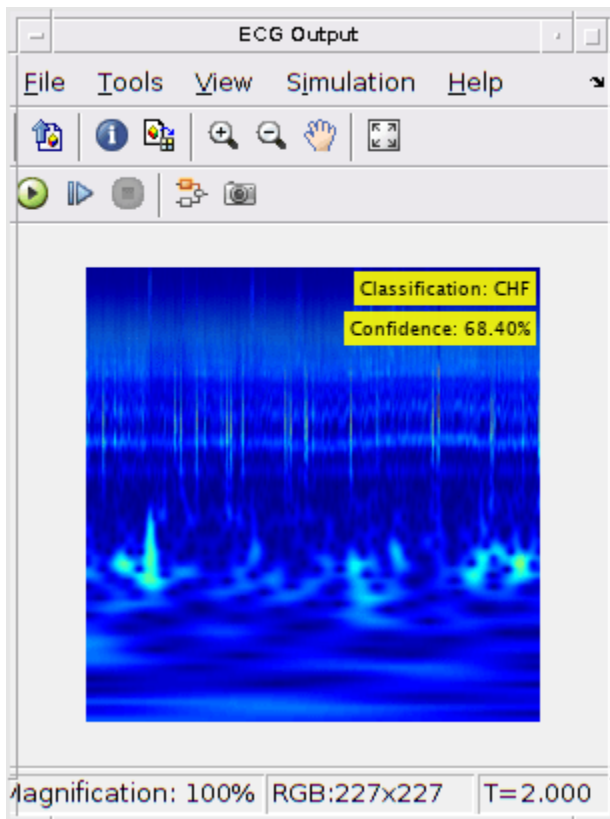
Open Configuration Parameters dialog box.

In **Simulation Target** pane, select **GPU acceleration**. In the **Deep Learning** group, select the target library as **cuDNN**.



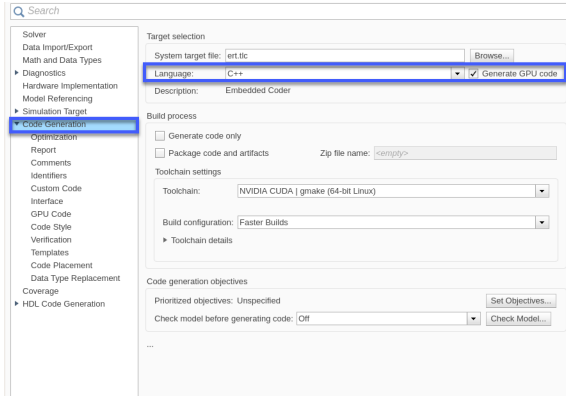
To verify the algorithm and display the labels and confidence score of the test ECG signal loaded in the workspace, run the simulation.

```
set_param('ecg_dl_cwt', 'SimulationMode', 'Normal');
sim('ecg_dl_cwt');
```

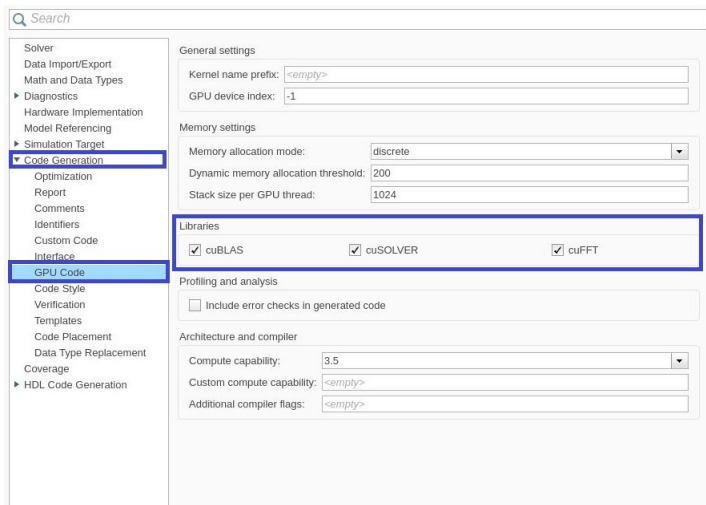


Generate and Build the Simulink Model

In **Code Generation** pane, select the **Language** as **C++** and enable **Generate GPU code**.



Open **Code Generation > GPU Code** pane. In the subcategory **Libraries**, enable **cuBLAS**, **cuSOLVER** and **cuFFT**.



Generate and build the Simulink model on the host GPU by using the `slbuild` command. The code generator places the files in a *build folder*, a subfolder named `ecg_dl_cwt_ert_rtw` under your current working folder.

```
status = evalc("slbuild('ecg_dl_cwt')");
```

Generated CUDA® Code

The subfolder named `ecg_dl_cwt_ert_rtw` contains the generated C++ codes corresponding to the different blocks in the Simulink model and the specific operations being performed in those blocks. For example, the file `trainedNet0_ecg_dl_cwt0.h` contains the C++ class which contains certain attributes such as `numLayers` and member functions such as `getBatchSize()`, `predict()`. This class represents the pretrained SqueezeNet which has been loaded in the Simulink model.


```

#ifndef RTW_HEADER_trainedNet0_ecg_dl_cwt0_h_
#define RTW_HEADER_trainedNet0_ecg_dl_cwt0_h_
#include "rtwtypes.h"
#include "MwTargetNetworkImpl.hpp"
#include "MwElementwiseAffineLayer.hpp"
#include "cnn_api.hpp"
#include "MwFusedConvReLULayer.hpp"
#include "MwConcatenationLayer.hpp"
#include "MwKernelHeaders.hpp"
#include "MwCustomLayerForCuDNN.hpp"

class trainedNet0_ecg_dl_cwt0
{
public:
    int32_T numLayers;
private:
    MwTensorBase *inputTensors;
    MwTensorBase *outputTensors;
public:
    MwCNLayer *Layers[42];
private:
    MwTargetNetworkImpl *targetImpl;
    void allocate();
    void postsetup();
public:
    trainedNet0_ecg_dl_cwt0();
private:
    void deallocate();
public:
    void setSize();
    void resetState();
    void setup();
    void predict();
    void cleanup();
    real32_T *getLayerOutput(int32_T layerIndex, int32_T portIndex);
    real32_T *getInputDataPointer(int32_T index);
    real32_T *getInputDataPointer();
    real32_T *getOutputDataPointer(int32_T index);
    real32_T *getOutputDataPointer();
    int32_T getBatchSize();
    ~trainedNet0_ecg_dl_cwt0();
};

#endif // RTW_HEADER_trainedNet0_ecg_dl_cwt0_h_

```

Cleanup

Close the Simulink model.

```
close_system('ecg_dl_cwt/ECG Preprocessing');
close_system('ecg_dl_cwt');
```

See Also

Functions

`open_system` | `load_system` | `save_system` | `close_system` | `bdclose` | `get_param` | `set_param` | `sim` | `slbuild`

More About

- “Simulation Acceleration by Using GPU Coder” on page 3-2
- “Code Generation from Simulink Models with GPU Coder” on page 3-8
- “GPU Code Generation for Deep Learning Networks Using MATLAB Function Block” on page 3-14
- “Targeting NVIDIA Embedded Boards” on page 3-30
- “Numerical Equivalence Testing” on page 3-32
- “Parameter Tuning and Signal Monitoring by Using External Mode” on page 3-38

Code Generation for a Deep Learning Simulink Model that Performs Lane and Vehicle Detection

This example shows how to develop a CUDA® application from a Simulink® model that performs lane and vehicle detection using convolutional neural networks (CNN). This example takes the frames of a traffic video as an input, outputs two lane boundaries that correspond to the left and right lanes of the ego vehicle, and detects vehicles in the frame. This example uses the pretrained lane detection network from the *Lane Detection Optimized with GPU Coder* example of the GPU Coder Toolbox™. For more information, see “Lane Detection Optimized with GPU Coder” on page 4-124. This example also uses the pretrained vehicle detection network from the *Object Detection Using YOLO v2 Deep Learning* example of the Computer Vision toolbox™. For more information, see “Object Detection Using YOLO v2 Deep Learning” (Computer Vision Toolbox).

This example illustrates the following concepts:

- Model the lane detection application in Simulink. First the traffic video is preprocessed by resizing to 227x227x3 and multiplication by a constant factor of 255. Subsequently, it is processed by the pretrained network loaded in the `Predict` block from the Deep Learning Toolbox™. Finally, if the left and right lane boundaries are detected, the parabolic coefficients to model the trajectories of the lane boundaries are obtained.
- Model the vehicle detection application in Simulink. The traffic video is processed by a pretrained YOLO v2 detector. This network detects vehicles in the video and outputs the coordinates of the bounding boxes for these vehicles and their confidence score.
- Configure the model for code generation.
- Generate a CUDA executable for the Simulink model.

Third-Party Prerequisites

- CUDA enabled NVIDIA GPU.
- NVIDIA CUDA toolkit and driver.
- NVIDIA cuDNN library.
- Environment variables for the compilers and libraries. For more information, see “Third-Party Hardware” and “Setting Up the Prerequisite Products”.

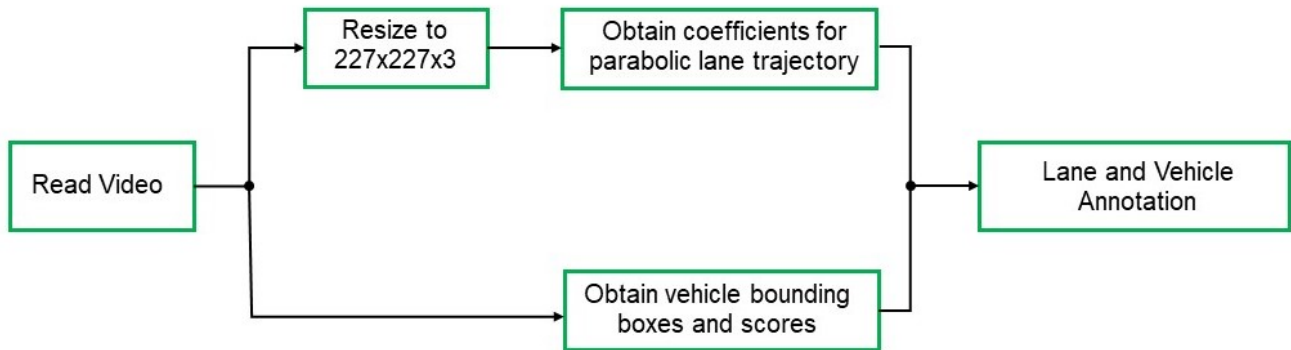
Verify GPU Environment

To verify that the compilers and libraries necessary for running this example are set up correctly, use the `coder.checkGpuInstall` function.

```
envCfg = coder.gpuEnvConfig('host');  
envCfg.DeepLibTarget = 'cudnn';  
envCfg.DeepCodegen = 1;  
envCfg.Quiet = 1;  
coder.checkGpuInstall(envCfg);
```

Algorithmic Workflow

The block diagram for the algorithmic workflow of the Simulink model is shown.



Get Pretrained Lane and Vehicle Detection Networks

This example uses the `trainedLaneNet` and `yolov2ResNet50VehicleExample` MAT-files containing the pretrained networks. The files are approximately 143MB and 98MB in size, respectively. Download the files from the MathWorks website.

```
lanenetFile = matlab.internal.examples.downloadSupportFile('gpcoder/cnn_models/lane_detection',
vehiclenetFile = matlab.internal.examples.downloadSupportFile('vision/data', 'yolov2ResNet50VehicleExample');
```

Download Test Traffic Video

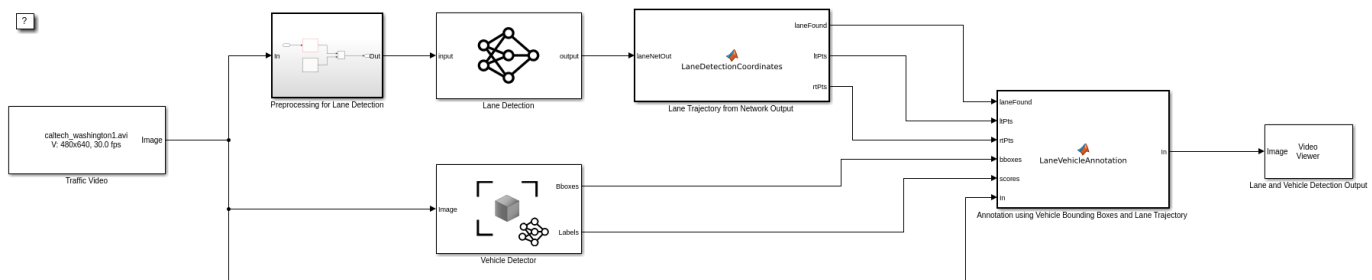
To test the model, the example uses the Caltech lanes dataset. The file is approximately 16 MB in size. Download the files from the MathWorks website.

```
mediaFile = matlab.internal.examples.downloadSupportFile('gpcoder/media', 'caltech_washington1.avi');
```

Lane and Vehicle Detection Simulink Model

The Simulink model for performing lane and vehicle detection on the traffic video is shown. When the model runs, the Video Viewer block displays the traffic video with lane and vehicle annotations.

```
open_system('laneAndVehicleDetection');
```



Copyright 2020-2021 The MathWorks, Inc.

Set the file paths of the downloaded network model in the predict and detector blocks of the Simulink model. Set the location of the test video to be loaded by the Simulink model.

```
set_param('laneAndVehicleDetection/Lane Detection', 'NetworkFilePath', lanenetFile)
set_param('laneAndVehicleDetection/Vehicle Detector', 'DetectorFilePath', vehiclenetFile)
set_param('laneAndVehicleDetection/Traffic Video', 'inputFileName', mediaFile)
```

Lane Detection

The `Predict` block loads the pretrained lane detection network from the `trainedLaneNet.mat` file. This network takes an image as an input and outputs two lane boundaries that correspond to the left and right lanes of the ego vehicle. Each lane boundary is represented by the parabolic equation:

$$y = ax^2 + bx + c$$

Here y is the lateral offset and x is the longitudinal distance from the vehicle. The network outputs the three parameters a , b , and c per lane. The network architecture is similar to `AlexNet` except that the last few layers are replaced by a smaller fully connected layer and regression output layer. The `LaneDetectionCoordinates` MATLAB function block defines a function `lane_detection_coordinates` that takes the output from the `predict` block and outputs three parameters i.e. `laneFound`, `ltPts` and `rtPts`. Thresholding is used to determine if both left and right lane boundaries are both found. If both are found, `laneFound` is set to be true and the trajectories of the boundaries are calculated and stored in `ltPts` and `rtPts` respectively.

type `lane_detection_coordinates`

```
function [laneFound,ltPts,rtPts] = lane_detection_coordinates(laneNetOut)
```

```
% Copyright 2020-2021 The MathWorks, Inc.
```

```
persistent laneCoeffMeans;
if isempty(laneCoeffMeans)
    laneCoeffMeans = [-0.0002,0.0002,1.4740,-0.0002,0.0045,-1.3787];
end
```

```
persistent laneCoeffStds;
if isempty(laneCoeffStds)
    laneCoeffStds = [0.0030,0.0766,0.6313,0.0026,0.0736,0.9846];
end
```

```
params = laneNetOut .* laneCoeffStds + laneCoeffMeans;
```

```
% 'c' should be more than 0.5 for it to be a right lane
isRightLaneFound = abs(params(6)) > 0.5;
isLeftLaneFound = abs(params(3)) > 0.5;
```

```
persistent vehicleXPoints;
if isempty(vehicleXPoints)
    vehicleXPoints = 3:30; %meters, ahead of the sensor
end
```

```
ltPts = coder.nullcopy(zeros(28,2,'single'));
rtPts = coder.nullcopy(zeros(28,2,'single'));
```

```
if isRightLaneFound && isLeftLaneFound
    rtBoundary = params(4:6);
    rt_y = computeBoundaryModel(rtBoundary, vehicleXPoints);
    ltBoundary = params(1:3);
    lt_y = computeBoundaryModel(ltBoundary, vehicleXPoints);
```

```
% Visualize lane boundaries of the ego vehicle
tform = get_tformToImage;
% Map vehicle to image coordinates
```

```

    ltPts = tform.transformPointsInverse([vehicleXPoints', lt_y']);
    rtPts = tform.transformPointsInverse([vehicleXPoints', rt_y']);
    laneFound = true;
else
    laneFound = false;
end
end
end

```

Vehicle Detection

A YOLO v2 object detection network is composed of two subnetworks: a feature extraction network followed by a detection network. This pretrained network uses a ResNet -50 for feature extraction. The detection sub-network is a small CNN compared to the feature extraction network and is composed of a few convolutional layers and layers specific to YOLO v2. The Simulink model performs vehicle detection using the `Object Detector` block from the Computer Vision Toolbox(™). This block takes an image as input and outputs the bounding box coordinates along with the confidence scores for vehicles in the image.

Annotation of Vehicle Bounding Boxes and Lane Trajectory in Traffic Video

The `LaneVehicleAnnotation` MATLAB function block defines a function `lane_vehicle_annotation` which annotates the vehicle bounding boxes along with the confidence scores. Also, if `laneFound` is true, then the left and right lane boundaries stored in `ltPts` and `rtPts` are annotated in the traffic video.

```
type lane_vehicle_annotation
```

```

function In = lane_vehicle_annotation(laneFound,ltPts,rtPts,bboxes,scores,In)

% Copyright 2020-2021 The MathWorks, Inc.

if ~isempty(bboxes)
    In = insertObjectAnnotation(In, 'rectangle',bboxes,scores);
end

pts = coder.nullcopy(zeros(28, 4, 'single'));
if laneFound
    prevpt = [ltPts(1,1) ltPts(1,2)];
    for k = 2:1:28
        pts(k,1:4) = [prevpt ltPts(k,1) ltPts(k,2)];
        prevpt = [ltPts(k,1) ltPts(k,2)];
    end
    In = insertShape(In, 'Line', pts, 'LineWidth', 2);
    prevpt = [rtPts(1,1) rtPts(1,2)];
    for k = 2:1:28
        pts(k,1:4) = [prevpt rtPts(k,1) rtPts(k,2)];
        prevpt = [rtPts(k,1) rtPts(k,2)];
    end
    In = insertShape(In, 'Line', pts, 'LineWidth', 2);
    In = insertMarker(In, ltPts);
    In = insertMarker(In, rtPts);
end

end

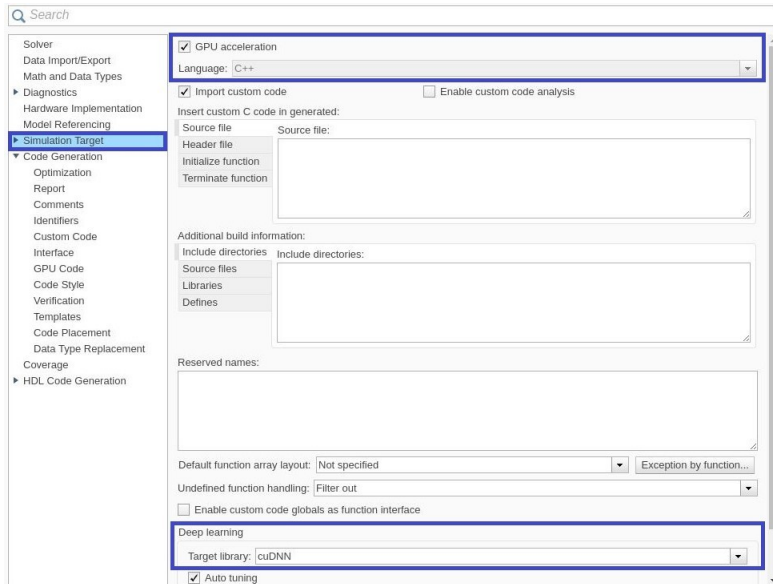
```

Run the Simulation

Open Configuration Parameters dialog box.

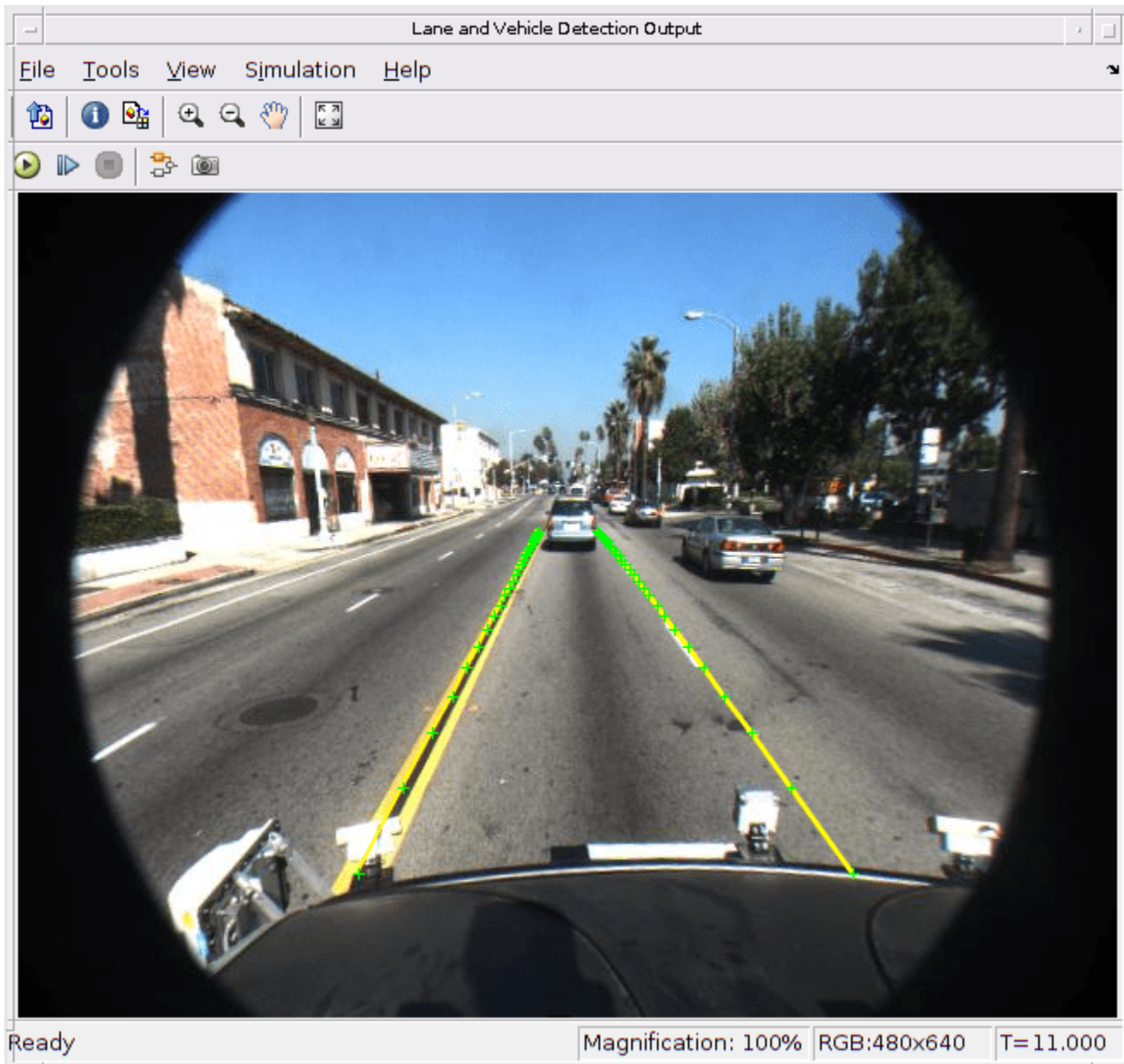
In **Simulation Target** pane, select **GPU acceleration**. In the **Deep Learning** group, select the target library as **cuDNN**.

```
set_param(bdroot, 'GPUAcceleration', 'on');
set_param(bdroot, 'SimDLTargetLibrary', 'cudnn');
set_param(bdroot, 'DLTargetLibrary', 'cudnn');
```



To verify the lane and vehicle detection algorithms and display the lane trajectories, vehicle bounding boxes and scores for the traffic video loaded in the Simulink model, run the simulation.

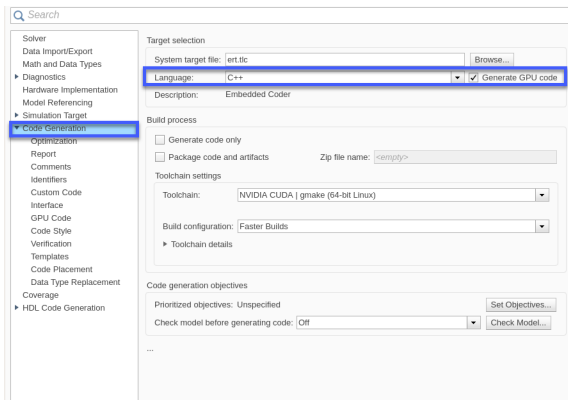
```
set_param('laneAndVehicleDetection', 'SimulationMode', 'Normal');
sim('laneAndVehicleDetection');
```



Generate and Build the Simulink Model

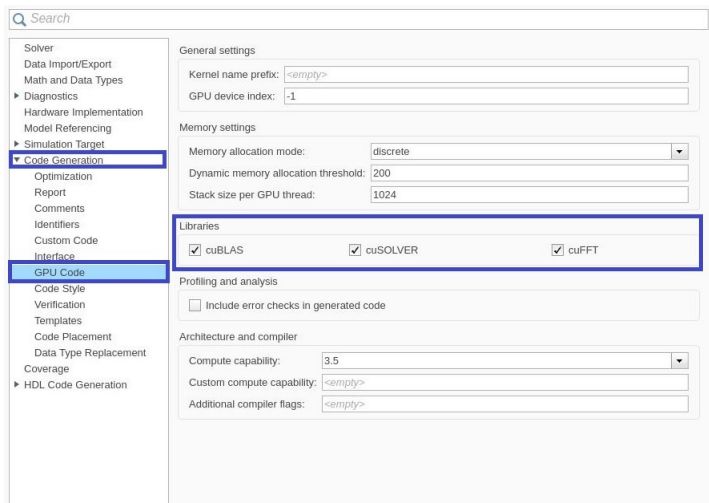
In **Code Generation** pane, select the **Language** as **C++** and enable **Generate GPU code**.

```
set_param(bdroot, 'TargetLang', 'C++');
set_param(bdroot, 'GenerateGPUCode', 'CUDA');
set_param(bdroot, 'GenCodeOnly', 'on');
if ispc
    set_param(bdroot, 'ProdHWDeviceType', 'Intel->x86-64 (Windows64)');
else
    set_param(bdroot, 'ProdHWDeviceType', 'Intel->x86-64 (Linux 64)');
end
set_param(bdroot, 'ProdLongLongMode', 'on');
```



In the subcategory **Libraries** of the **Code Generation > GPU Code** pane, enable **cuBLAS**, **cuSOLVER** and **cuFFT**.

```
set_param(bdroot, 'GPUcuBLAS', 'on');
set_param(bdroot, 'GPUcuSOLVER', 'on');
set_param(bdroot, 'GPUcuFFT', 'on');
```



Generate and build the Simulink model on the host GPU by using the `slbuild` command. The code generator places the files in a *build folder*, a subfolder named `laneAndVehicleDetection_ert_rtw` under your current working folder.

```
status = evalc("slbuild('laneAndVehicleDetection')");
```

Generated CUDA Code

The subfolder named `laneAndVehicleDetection_ert_rtw` contains the generated C++ codes corresponding to the different blocks in the Simulink model and the specific operations being performed in those blocks. For example, the file `trainedLaneNet0_laneAndVehicleDetection0.h` contains the C++ class which contains attributes and member functions representing the pretrained lane detection network.


```

#ifndef RTW_HEADER_trainedLaneNet0_laneAndVehicleDetection0_h_
#define RTW_HEADER_trainedLaneNet0_laneAndVehicleDetection0_h_
#include "rtwtypes.h"
#include "MwTargetNetworkImpl.hpp"
#include "MwYoloExtractionLayer.hpp"
#include "MwElementwiseAffineLayer.hpp"
#include "MwSigmoidLayer.hpp"
#include "MwFusedConvReLULayer.hpp"
#include "cnn_api.hpp"
#include "MwYoloSoftmaxLayer.hpp"
#include "MwConcatenationLayer.hpp"
#include "MwExponentialLayer.hpp"
#include "MwConvLayer.hpp"
#include "MwKernelHeaders.hpp"
#include "MwCustomLayerForCuDNN.hpp"

class trainedLaneNet0_laneAndVehicleDetection0
{
public:
    int32_T numLayers;
private:
    MwTensorBase *inputTensors;
    MwTensorBase *outputTensors;
public:
    MwCNLayer *layers[18];
private:
    MwTargetNetworkImpl *targetImpl;
    void allocate();
    void postsetup();
public:
    trainedLaneNet0_laneAndVehicleDetection0();
private:
    void deallocate();
public:
    void setSize();
    void resetState();
    void setup();
    void predict();
    void cleanup();
    real32_T *getLayerOutput(int32_T layerIndex, int32_T portIndex);
    real32_T *getInputDataPointer(int32_T index);
    real32_T *getInputDataPointer();
    real32_T *getOutputDataPointer(int32_T index);
    real32_T *getOutputDataPointer();
    int32_T getBatchSize();
    ~trainedLaneNet0_laneAndVehicleDetection0();
};

#endif /* RTW_HEADER_trainedLaneNet0_laneAndVehicleDetection0_h_ */

```

Similarly, the file `yolov2ResNet50VehicleExample0_laneAndVehicleDetection0.h` contains the C++ class representing the pretrained YOLO v2 detection network.

```
#ifndef RTW_HEADER_yolov2ResNet50VehicleExample0_laneAndVehicleDetection0_h_
#define RTW_HEADER_yolov2ResNet50VehicleExample0_laneAndVehicleDetection0_h_
#include "rtwtypes.h"
#include "MwTargetNetworkImpl.hpp"
#include "MwYoloExtractionLayer.hpp"
#include "MwElementwiseAffineLayer.hpp"
#include "MwSigmoidLayer.hpp"
#include "MwFusedConvReLULayer.hpp"
#include "cnn_api.hpp"
#include "MwYoloSoftmaxLayer.hpp"
#include "MwConcatenationLayer.hpp"
#include "MwExponentialLayer.hpp"
#include "MwConvLayer.hpp"
#include "MwKernelHeaders.hpp"
#include "MwCustomLayerForCuDNN.hpp"

class yolov2ResNet50VehicleExample0_laneAndVehicleDetection0
{
public:
    int32_T numLayers;
private:
    MwTensorBase *inputTensors;
    MwTensorBase *outputTensors;
public:
    MwCNLayer *layers[57];
private:
    MwTargetNetworkImpl *targetImpl;
    void allocate();
    void postsetup();
public:
    yolov2ResNet50VehicleExample0_laneAndVehicleDetection0();
private:
    void deallocate();
public:
    void setSize();
    void resetState();
    void setup();
    void predict();
    void activations(int32_T layerIdx);
    void cleanup();
    real32_T *getLayerOutput(int32_T layerIndex, int32_T portIndex);
    real32_T *getInputDataPointer(int32_T index);
    real32_T *getInputDataPointer();
    real32_T *getOutputDataPointer(int32_T index);
    real32_T *getOutputDataPointer();
    int32_T getBatchSize();
    ~yolov2ResNet50VehicleExample0_laneAndVehicleDetection0();
};

#endif
/* RTW_HEADER_yolov2ResNet50VehicleExample0_laneAndVehicleDetection0_h_ */
```

See Also

Functions

[open_system](#) | [load_system](#) | [save_system](#) | [close_system](#) | [bdclose](#) | [get_param](#) | [set_param](#) | [sim](#) | [slbuild](#)

More About

- “Simulation Acceleration by Using GPU Coder” on page 3-2
- “Code Generation from Simulink Models with GPU Coder” on page 3-8
- “GPU Code Generation for Deep Learning Networks Using MATLAB Function Block” on page 3-14
- “Targeting NVIDIA Embedded Boards” on page 3-30
- “Numerical Equivalence Testing” on page 3-32
- “Parameter Tuning and Signal Monitoring by Using External Mode” on page 3-38

Deep Learning

- “Workflow” on page 4-3
- “Supported Networks, Layers, and Classes” on page 4-6
- “Analyze Network for Code Generation” on page 4-44
- “Code Generation for dlarray” on page 4-52
- “dlarray Limitations for Code Generation” on page 4-62
- “Generated CNN Class Hierarchy” on page 4-65
- “Load Pretrained Networks for Code Generation” on page 4-66
- “Code Generation for Deep Learning Networks by Using cuDNN” on page 4-69
- “Code Generation for Deep Learning Networks by Using TensorRT” on page 4-78
- “Code Generation for Deep Learning Networks Targeting ARM Mali GPUs” on page 4-88
- “Update Network Parameters After Code Generation” on page 4-92
- “Data Layout Considerations in Deep Learning” on page 4-96
- “Quantization of Deep Neural Networks” on page 4-99
- “Generate INT8 Code for Deep Learning Networks” on page 4-107
- “Code Generation for Deep Learning Networks” on page 4-117
- “Lane Detection Optimized with GPU Coder” on page 4-124
- “Traffic Sign Detection and Recognition” on page 4-132
- “Logo Recognition Network” on page 4-140
- “Deep Learning Prediction with NVIDIA TensorRT Library” on page 4-145
- “Code Generation for Semantic Segmentation Network” on page 4-152
- “Train and Deploy Fully Convolutional Networks for Semantic Segmentation” on page 4-157
- “Code Generation for Semantic Segmentation Network That Uses U-net” on page 4-169
- “Code Generation for Denoising Deep Neural Network” on page 4-176
- “Code Generation for Object Detection by Using YOLO v2” on page 4-180
- “Code Generation for a Sequence-to-Sequence LSTM Network” on page 4-184
- “Deep Learning Prediction on ARM Mali GPU” on page 4-190
- “Code Generation for Object Detection by Using Single Shot Multibox Detector” on page 4-193
- “Code Generation for a Deep Learning Simulink Model to Classify ECG Signals” on page 4-197
- “Code Generation for Lidar Point Cloud Segmentation Network” on page 4-204
- “Code Generation for a Video Classification Network” on page 4-211
- “Code Generation For Object Detection Using YOLO v3 Deep Learning” on page 4-217
- “Generate Digit Images on NVIDIA GPU Using Variational Autoencoder ” on page 4-222
- “Quantize Residual Network Trained for Image Classification and Generate CUDA Code” on page 4-229
- “Quantize Layers in Object Detectors and Generate CUDA Code” on page 4-237

- “Parameter Pruning and Quantization of Image Classification Network” on page 4-249
- “Code Generation For Aerial Lidar Semantic Segmentation Using PointNet++ Deep Learning” on page 4-266
- “Code Generation For Lidar Object Detection Using PointPillars Deep Learning” on page 4-272
- “Code Generation for Object Detection Using YOLO v4 Deep Learning” on page 4-277

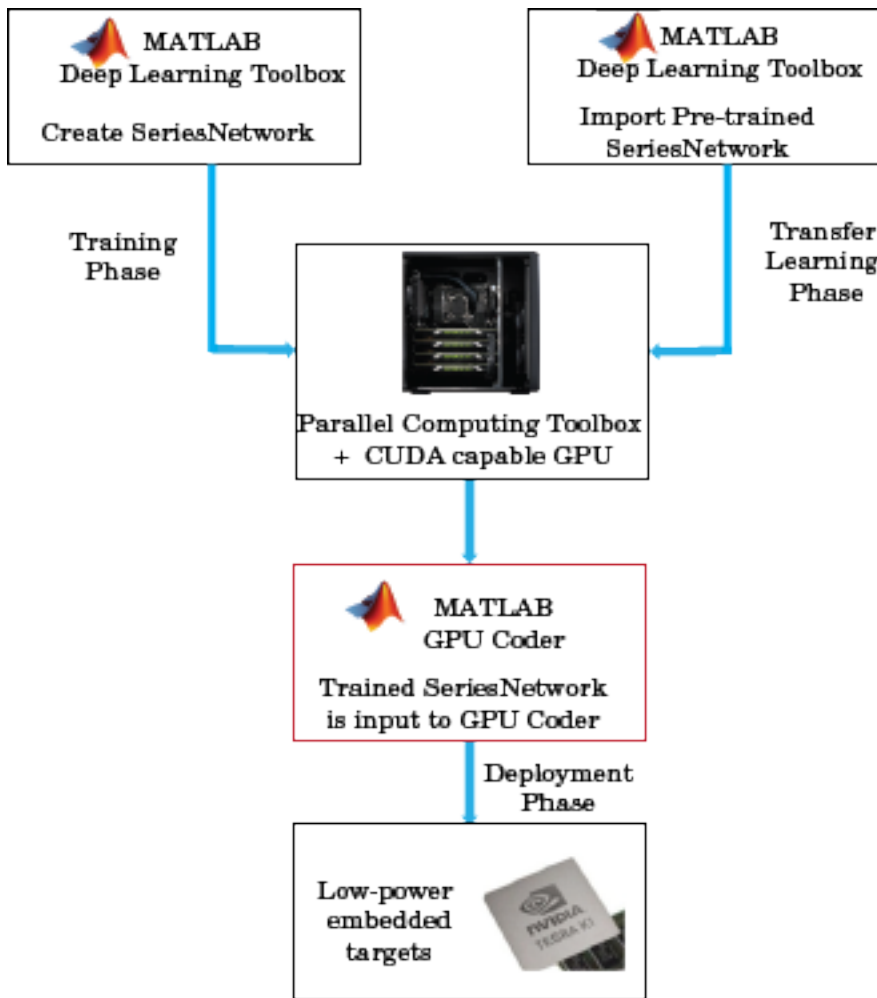
Workflow

In a typical Convolutional Neural Networks (CNN) workflow, you start with constructing a CNN architecture by using the Deep Learning Toolbox, and train the network in tandem with the Parallel Computing Toolbox™. Alternatively, you can import a ConvNet already trained on a large dataset, and transfer the learned features. Transfer learning implies taking a CNN trained for one set of classification problems and retraining it to classify a different set of classes. Here the last few layers of the CNN are relearned. Again, Parallel Computing Toolbox is used in the learning phase. You can also import a trained CNN network from other frameworks like Caffe or MatConvNet into a SeriesNetwork object.

Once you have obtained the trained network, you can use GPU Coder to generate C++ or CUDA code and deploy CNN on multiple embedded platforms that use NVIDIA or ARM GPU processors. The generated code implements the CNN by using the architecture, the layers, and parameters that you specify in the input SeriesNetwork or DAGNetwork object.

The code generator takes advantage of NVIDIA CUDA deep neural network library (cuDNN), NVIDIA TensorRT high performance inference library for NVIDIA GPUs and ARM Compute Library for computer vision and machine learning for ARM Mali GPUs.

The generated code can be integrated into your project as source code, static or dynamic libraries, or executables that you can deploy to a variety of NVIDIA and ARM Mali GPU platforms. For performing deep learning on ARM Mali GPU targets, you generate code on the host development computer. Then, to build and run the executable program move the generated code to the ARM target platform.



See Also

Functions

`coder.getDeepLearningLayers` | `codegen` | `coder.DeepLearningConfig`

Objects

`coder.gpuConfig` | `coder.CodeConfig` | `coder.EmbeddedCodeConfig` | `coder.gpuEnvConfig` | `coder.CuDNNConfig` | `coder.TensorRTConfig`

More About

- "Pretrained Deep Neural Networks" (Deep Learning Toolbox)
- "Get Started with Transfer Learning" (Deep Learning Toolbox)
- "Create Simple Deep Learning Network for Classification" (Deep Learning Toolbox)
- "Supported Networks, Layers, and Classes" on page 4-6
- "Load Pretrained Networks for Code Generation" on page 4-66
- "Code Generation for Deep Learning Networks by Using cuDNN" on page 4-69

- “Code Generation for Deep Learning Networks by Using TensorRT” on page 4-78
- “Code Generation for Deep Learning Networks Targeting ARM Mali GPUs” on page 4-88

Supported Networks, Layers, and Classes

Supported Pretrained Networks

GPU Coder supports code generation for series and directed acyclic graph (DAG) convolutional neural networks (CNNs or ConvNets). You can generate code for any trained convolutional neural network whose layers are supported for code generation. See “Supported Layers” on page 4-11. You can train a convolutional neural network on either a CPU, a GPU, or multiple GPUs by using the Deep Learning Toolbox or use one of the pretrained networks listed in the table and generate CUDA code.

Network Name	Description	cuDNN	TensorRT	ARM Compute Library for Mali GPU
AlexNet	AlexNet convolutional neural network. For the pretrained AlexNet model, see <code>alexnet</code> . The syntax <code>alexnet('Weights','none')</code> is not supported for code generation.	Yes	Yes	Yes
Caffe Network	Convolutional neural network models from Caffe. For importing a pretrained network from Caffe, see <code>importCaffeNetwork</code> .	Yes	Yes	Yes
Darknet-19	Darknet-19 convolutional neural network. For more information, see <code>darknet19</code> . The syntax <code>darknet19('Weights','none')</code> is not supported for code generation.	Yes	Yes	Yes

Network Name	Description	cuDNN	TensorRT	ARM Compute Library for Mali GPU
Darknet-53	<p>Darknet-53 convolutional neural network. for more information, see <code>darknet53</code>.</p> <p>The syntax <code>darknet53('Weights', 'none')</code> is not supported for code generation.</p>	Yes	Yes	Yes
DeepLab v3+	<p>DeepLab v3+ convolutional neural network. For more information, see <code>deeplabv3plusLayers</code>.</p>	Yes	Yes	No
DenseNet-201	<p>DenseNet-201 convolutional neural network. For the pretrained DenseNet-201 model, see <code>densenet201</code>.</p> <p>The syntax <code>densenet201('Weights', 'none')</code> is not supported for code generation.</p>	Yes	Yes	Yes
EfficientNet-b0	<p>EfficientNet-b0 convolutional neural network. For the pretrained EfficientNet-b0 model, see <code>efficientnetb0</code>.</p> <p>The syntax <code>efficientnetb0('Weights', 'none')</code> is not supported for code generation.</p>	Yes	Yes	Yes

Network Name	Description	cuDNN	TensorRT	ARM Compute Library for Mali GPU
GoogLeNet	<p>GoogLeNet convolutional neural network. For the pretrained GoogLeNet model, see googlenet.</p> <p>The syntax <code>googlenet('Weights', 'none')</code> is not supported for code generation.</p>	Yes	Yes	Yes
Inception-ResNet-v2	<p>Inception-ResNet-v2 convolutional neural network. For the pretrained Inception-ResNet-v2 model, see inceptionresnetv2.</p>	Yes	Yes	No
Inception-v3	<p>Inception-v3 convolutional neural network. For the pretrained Inception-v3 model, see inceptionv3.</p> <p>The syntax <code>inceptionv3('Weights', 'none')</code> is not supported for code generation.</p>	Yes	Yes	Yes

Network Name	Description	cuDNN	TensorRT	ARM Compute Library for Mali GPU
Mobilenet-v2	<p>MobileNet-v2 convolutional neural network. For the pretrained MobileNet-v2 model, see <code>mobilenetv2</code>.</p> <p>The syntax <code>mobilenetv2('Weights', 'none')</code> is not supported for code generation.</p>	Yes	Yes	Yes
NASNet-Large	<p>NASNet-Large convolutional neural network. For the pretrained NASNet-Large model, see <code>nasnetlarge</code>.</p>	Yes	Yes	No
NASNet-Mobile	<p>NASNet-Mobile convolutional neural network. For the pretrained NASNet-Mobile model, see <code>nasnetmobile</code>.</p>	Yes	Yes	No
ResNet	<p>ResNet-18, ResNet-50, and ResNet-101 convolutional neural networks. For the pretrained ResNet models, see <code>resnet50</code>, <code>resnet18</code>, and <code>resnet101</code>.</p> <p>The syntax <code>resnetXX('Weights', 'none')</code> is not supported for code generation.</p>	Yes	Yes	Yes


Network Name	Description	cuDNN	TensorRT	ARM Compute Library for Mali GPU
SegNet	Multi-class pixelwise segmentation network. For more information, see <code>segnetLayers</code> .	Yes	Yes	No
SqueezeNet	Small deep neural network. For the pretrained SqueezeNet models, see <code>squeezenet</code> . The syntax <code>squeezenet('Weights', 'none')</code> is not supported for code generation.	Yes	Yes	Yes
VGG-16	VGG-16 convolutional neural network. For the pretrained VGG-16 model, see <code>vgg16</code> . The syntax <code>vgg16('Weights', 'none')</code> is not supported for code generation.	Yes	Yes	Yes
VGG-19	VGG-19 convolutional neural network. For the pretrained VGG-19 model, see <code>vgg19</code> . The syntax <code>vgg19('Weights', 'none')</code> is not supported for code generation.	Yes	Yes	Yes



Network Name	Description	cuDNN	TensorRT	ARM Compute Library for Mali GPU
Xception	Xception convolutional neural network. For the pretrained Xception model, see <code>xception</code> . The syntax <code>xception('Weights', 'none')</code> is not supported for code generation.	Yes	Yes	Yes
YOLO v2	You only look once version 2 convolutional neural network based object detector. For more information, see <code>yolov2Layers</code>	Yes	Yes	Yes

Supported Layers





The following layers are supported for code generation by GPU Coder for the target deep learning libraries specified in the table.

Input Layers



Layer Name	Description	cuDNN	TensorRT	ARM Compute Library for Mali GPU
 <code>imageInputLayer</code>	An image input layer inputs 2-D images to a network and applies data normalization. Code generation does not support 'Normalization' specified using a function handle.	Yes	Yes	Yes



Layer Name	Description	cuDNN	TensorRT	ARM Compute Library for Mali GPU
 sequenceInputLayer	<p>A sequence input layer inputs sequence data to a network.</p> <p>The cuDNN library supports vector and 2-D image sequences. The TensorRT library support only vector input sequences.</p> <p>For vector sequence inputs, the number of features must be a constant during code generation.</p> <p>For image sequence inputs, the height, width, and the number of channels must be a constant during code generation.</p> <p>Code generation does not support 'Normalization' specified using a function handle.</p>	Yes	Yes	No
 featureInputLayer	<p>A feature input layer inputs feature data to a network and applies data normalization.</p>	Yes	Yes	Yes



Convolution and Fully Connected Layers



Layer Name	Description	cuDNN	TensorRT	ARM Compute Library for Mali GPU
 convolution2dLayer	A 2-D convolutional layer applies sliding convolutional filters to the input.	Yes	Yes	Yes
 fullyConnectedLayer	A fully connected layer multiplies the input by a weight matrix and then adds a bias vector.	Yes	Yes	No
 groupedConvolution2dLayer	A 2-D grouped convolutional layer separates the input channels into groups and applies sliding convolutional filters. Use grouped convolutional layers for channel-wise separable (also known as depth-wise separable) convolution. Code generation for the ARM Mali GPU is not supported for a 2-D grouped convolution layer that has the NumGroups property set as 'channel-wise' or a value greater than two.	Yes	Yes	Yes
 transposedConv2dLayer	A transposed 2-D convolution layer upsamples feature maps.	Yes	Yes	Yes

Sequence Layers




Layer Name	Description	cuDNN	TensorRT	ARM Compute Library for Mali GPU
 biLstmLayer	<p>A bidirectional LSTM (BiLSTM) layer learns bidirectional long-term dependencies between time steps of time series or sequence data. These dependencies can be useful when you want the network to learn from the complete time series at each time step.</p> <p>For code generation, the <code>StateActivationFunction</code> property must be set to 'tanh'.</p> <p>For code generation, the <code>GateActivationFunction</code> property must be set to 'sigmoid'.</p>	Yes	Yes	No
 flattenLayer	<p>A flatten layer collapses the spatial dimensions of the input into the channel dimension.</p>	Yes	No	No





Layer Name	Description	cuDNN	TensorRT	ARM Compute Library for Mali GPU
 gruLayer	<p>A GRU layer learns dependencies between time steps in time series and sequence data.</p> <p>Code generation supports only the 'after-multiplication' and 'recurrent-bias-after-multiplication' reset gate modes.</p>	Yes	Yes	No
 lstmLayer	<p>An LSTM layer learns long-term dependencies between time steps in time series and sequence data.</p> <p>For code generation, the StateActivationFunction property must be set to 'tanh'.</p> <p>For code generation, the GateActivationFunction property must be set to 'sigmoid'.</p>	Yes	Yes	No

Layer Name	Description	cuDNN	TensorRT	ARM Compute Library for Mali GPU
 sequenceFoldingLayer	<p>A sequence folding layer converts a batch of image sequences to a batch of images. Use a sequence folding layer to perform convolution operations on time steps of image sequences independently.</p>	Yes	No	No
 sequenceInputLayer	<p>A sequence input layer inputs sequence data to a network.</p> <p>The cuDNN library supports vector and 2-D image sequences. The TensorRT library support only vector input sequences.</p> <p>For vector sequence inputs, the number of features must be a constant during code generation.</p> <p>For image sequence inputs, the height, width, and the number of channels must be a constant during code generation.</p> <p>Code generation does not support 'Normalization' specified using a function handle.</p>	Yes	Yes	No



Layer Name	Description	cuDNN	TensorRT	ARM Compute Library for Mali GPU
 sequenceUnfoldingLayer	A sequence unfolding layer restores the sequence structure of the input data after sequence folding.	Yes	No	No
 wordEmbeddingLayer	A word embedding layer maps word indices to vectors.	Yes	Yes	No





Activation Layers

Layer Name	Description	cuDNN	TensorRT	ARM Compute Library for Mali GPU
 clippedReluLayer	A clipped ReLU layer performs a threshold operation, where any input value less than zero is set to zero and any value above the <i>clipping ceiling</i> is set to that clipping ceiling.	Yes	Yes	Yes
 eluLayer	An ELU activation layer performs the identity operation on positive inputs and an exponential nonlinearity on negative inputs.	Yes	Yes	No
 leakyReluLayer	A leaky ReLU layer performs a threshold operation, where any input value less than zero is multiplied by a fixed scalar.	Yes	Yes	Yes


Layer Name	Description	cuDNN	TensorRT	ARM Compute Library for Mali GPU
 reluLayer	A ReLU layer performs a threshold operation to each element of the input, where any value less than zero is set to zero.	Yes	Yes	Yes
 softplusLayer	A SoftplusLayer is a deep neural network layer that implements the softplus activation $Y = \log(1 + e^X)$, which ensures that the output is always positive.	Yes	Yes	No
 swishLayer	A swish activation layer applies the swish function on the layer inputs.	Yes	Yes	No
 tanhLayer	A hyperbolic tangent (tanh) activation layer applies the tanh function on the layer inputs.	Yes	Yes	Yes



Normalization, Dropout, and Cropping Layers


Layer Name	Description	cuDNN	TensorRT	ARM Compute Library for Mali GPU
 batchNormalizationLayer	A batch normalization layer normalizes each input channel across a mini-batch.	Yes	Yes	Yes
 crop2dLayer	A 2-D crop layer applies 2-D cropping to the input.	Yes	Yes	Yes


Layer Name	Description	cuDNN	TensorRT	ARM Compute Library for Mali GPU
 crossChannelNormalizationLayer	A channel-wise local response (cross-channel) normalization layer carries out channel-wise normalization.	Yes	Yes	Yes
 dropoutLayer	A dropout layer randomly sets input elements to zero with a given probability.	Yes	Yes	Yes
 groupNormalizationLayer	A group normalization layer normalizes a mini-batch of data across grouped subsets of channels for each observation independently.	Yes	Yes	No
 scalingLayer	Scaling layer for actor or critic network. For code generation, values for the 'Scale' and 'Bias' properties must have the same dimension.	Yes	Yes	Yes

Pooling and Unpooling Layers



Layer Name	Description	cuDNN	TensorRT	ARM Compute Library for Mali GPU
 averagePooling2dLayer	An average pooling layer performs down-sampling by dividing the input into rectangular pooling regions and computing the average values of each region.	Yes	Yes	Yes


Layer Name	Description	cuDNN	TensorRT	ARM Compute Library for Mali GPU
 globalAveragePooling2dLayer	A global average pooling layer performs down-sampling by computing the mean of the height and width dimensions of the input.	Yes	Yes	Yes
 globalMaxPooling2dLayer	A global max pooling layer performs down-sampling by computing the maximum of the height and width dimensions of the input.	Yes	Yes	Yes

Layer Name	Description	cuDNN	TensorRT	ARM Compute Library for Mali GPU
 maxPooling2dLayer	<p>A max pooling layer performs down-sampling by dividing the input into rectangular pooling regions, and computing the maximum of each region.</p> <p>If equal max values exists along the off-diagonal in a kernel window, implementation differences for the maxPooling2dLayer might cause minor numerical mismatch between MATLAB and the generated code. This issue also causes mismatch in the indices of the maximum value in each pooled region. For more information, see maxPooling2dLayer.</p>	Yes	Yes	Yes





Layer Name	Description	cuDNN	TensorRT	ARM Compute Library for Mali GPU
 maxUnpooling2d Layer	<p>A max unpooling layer unpools the output of a max pooling layer.</p> <p>If equal max values exists along the off-diagonal in a kernel window, implementation differences for the maxPooling2dLayer might cause minor numerical mismatch between MATLAB and the generated code. This issue also causes mismatch in the indices of the maximum value in each pooled region. For more information, see maxUnpooling2d Layer.</p>	Yes	Yes	No







Combination Layers

Layer Name	Description	cuDNN	TensorRT	ARM Compute Library for Mali GPU
 additionLayer	An addition layer adds inputs from multiple neural network layers element-wise.	Yes	Yes	Yes
 concatenationLayer	A concatenation layer takes inputs and concatenates them along a specified dimension.	Yes	Yes	No




Layer Name	Description	cuDNN	TensorRT	ARM Compute Library for Mali GPU
 depthConcatenationLayer	A depth concatenation layer takes inputs that have the same height and width and concatenates them along the third dimension (the channel dimension).	Yes	Yes	Yes



Object Detection Layers






Layer Name	Description	cuDNN	TensorRT	ARM Compute Library for Mali GPU
 anchorBoxLayer	An anchor box layer stores anchor boxes for a feature map used in object detection networks.	Yes	Yes	Yes
 depthToSpace2dLayer	A 2-D depth to space layer permutes data from the depth dimension into blocks of 2-D spatial data.	Yes	Yes	Yes
 focalLossLayer	A focal loss layer predicts object classes using focal loss.	Yes	Yes	Yes
 spaceToDepthLayer	A space to depth layer permutes the spatial blocks of the input into the depth dimension. Use this layer when you need to combine feature maps of different size without discarding any feature data.	Yes	Yes	Yes

Layer Name	Description	cuDNN	TensorRT	ARM Compute Library for Mali GPU
 ssdMergeLayer	An SSD merge layer merges the outputs of feature maps for subsequent regression and classification loss computation.	Yes	Yes	No
 rcnnBoxRegressionLayer	A box regression layer refines bounding box locations by using a smooth L1 loss function. Use this layer to create a Fast or Faster R-CNN object detection network.	Yes	Yes	Yes
 rpnClassificationLayer	A region proposal network (RPN) classification layer classifies image regions as either <i>object</i> or <i>background</i> by using a cross entropy loss function. Use this layer to create a Faster R-CNN object detection network.	Yes	Yes	Yes
 YOLOv2OutputLayer	Create output layer for YOLO v2 object detection network.	Yes	Yes	Yes
 YOLOv2ReorgLayer	Create reorganization layer for YOLO v2 object detection network.	Yes	Yes	Yes
 YOLOv2TransformLayer	Create transform layer for YOLO v2 object detection network.	Yes	Yes	Yes

Output Layers

Layer Name	Description	cuDNN	TensorRT	ARM Compute Library for Mali GPU
 classificationLayer	A classification layer computes the cross entropy loss for multi-class classification problems with mutually exclusive classes.	Yes	Yes	Yes
 dicePixelClassificationLayer	A Dice pixel classification layer provides a categorical label for each image pixel or voxel using generalized Dice loss.	Yes	Yes	Yes
 focalLossLayer	A focal loss layer predicts object classes using focal loss.	Yes	Yes	Yes

Layer Name	Description	cuDNN	TensorRT	ARM Compute Library for Mali GPU
 Output Layer (Deep Learning Toolbox)	<p>All output layers including custom classification or regression output layers created by using <code>nnet.layer.ClassificationLayer</code> or <code>nnet.layer.RegressionLayer</code>.</p> <p>For an example showing how to define a custom classification output layer and specify a loss function, see “Define Custom Classification Output Layer” (Deep Learning Toolbox).</p> <p>For an example showing how to define a custom regression output layer and specify a loss function, see “Define Custom Regression Output Layer” (Deep Learning Toolbox).</p>	Yes	Yes	Yes
 <code>pixelClassificationLayer</code>	<p>A pixel classification layer provides a categorical label for each image pixel or voxel.</p>	Yes	Yes	Yes

Layer Name	Description	cuDNN	TensorRT	ARM Compute Library for Mali GPU
 rcnnBoxRegressionLayer	A box regression layer refines bounding box locations by using a smooth L1 loss function. Use this layer to create a Fast or Faster R-CNN object detection network.	Yes	Yes	Yes
 regressionLayer	A regression layer computes the half-mean-squared-error loss for regression problems.	Yes	Yes	Yes
 rpnClassificationLayer	A region proposal network (RPN) classification layer classifies image regions as either <i>object</i> or <i>background</i> by using a cross entropy loss function. Use this layer to create a Faster R-CNN object detection network.	Yes	Yes	Yes
 sigmoidLayer	A sigmoid layer applies a sigmoid function to the input.	Yes	Yes	Yes
 softmaxLayer	A softmax layer applies a softmax function to the input.	Yes	Yes	Yes

Custom Keras Layers

Layer Name	Description	cuDNN	TensorRT	ARM Compute Library for Mali GPU
<code>nnet.keras.layer.ClipLayer</code> (Deep Learning Toolbox)	Clips the input between the upper and lower bounds.	Yes	Yes	No
<code>nnet.keras.layer.FlattenCStyleLayer</code> (Deep Learning Toolbox)	Flatten activations into 1-D assuming C-style (row-major) order.	Yes	Yes	Yes
<code>nnet.keras.layer.GlobalAveragePooling2dLayer</code> (Deep Learning Toolbox)	Global average pooling layer for spatial data.	Yes	Yes	Yes
<code>nnet.keras.layer.PreluLayer</code> (Deep Learning Toolbox)	Parametric rectified linear unit.	Yes	Yes	No
<code>nnet.keras.layer.SigmoidLayer</code> (Deep Learning Toolbox)	Sigmoid activation layer.	Yes	Yes	Yes
<code>nnet.keras.layer.TanhLayer</code> (Deep Learning Toolbox)	Hyperbolic tangent activation layer.	Yes	Yes	Yes
<code>nnet.keras.layer.TimeDistributedFlattenCStyleLayer</code> (Deep Learning Toolbox)	Flatten a sequence of input image into a sequence of vector, assuming C-style (or row-major) storage ordering of the input layer.	Yes	Yes	No
<code>nnet.keras.layer.ZeroPadding2dLayer</code> (Deep Learning Toolbox)	Zero padding layer for 2-D input.	Yes	Yes	Yes

Custom ONNX Layers

Layer Name	Description	cuDNN	TensorRT	ARM Compute Library for Mali GPU
<code>nnet.onnx.layer.ClipLayer</code> (Deep Learning Toolbox)	Clips the input between the upper and lower bounds.	Yes	Yes	No
<code>nnet.onnx.layer.ElementwiseAffineLayer</code> (Deep Learning Toolbox)	Layer that performs element-wise scaling of the input followed by an addition.	Yes	Yes	Yes
<code>nnet.onnx.layer.FlattenInto2dLayer</code> (Deep Learning Toolbox)	Flattens a MATLAB 2D image batch in the way ONNX does, producing a 2D output array with CB format.	Yes	Yes	No
<code>nnet.onnx.layer.FlattenLayer</code> (Deep Learning Toolbox)	Flattens the spatial dimensions of the input tensor to the channel dimensions.	Yes	Yes	Yes
<code>nnet.onnx.layer.GlobalAveragePooling2dLayer</code> (Deep Learning Toolbox)	Global average pooling layer for spatial data.	Yes	Yes	Yes
<code>nnet.onnx.layer.IdentityLayer</code> (Deep Learning Toolbox)	Layer that implements ONNX identity operator.	Yes	Yes	Yes
<code>nnet.onnx.layer.PreluLayer</code> (Deep Learning Toolbox)	Parametric rectified linear unit.	Yes	Yes	No
<code>nnet.onnx.layer.SigmoidLayer</code> (Deep Learning Toolbox)	Sigmoid activation layer.	Yes	Yes	Yes
<code>nnet.onnx.layer.TanhLayer</code> (Deep Learning Toolbox)	Hyperbolic tangent activation layer.	Yes	Yes	Yes

Layer Name	Description	cuDNN	TensorRT	ARM Compute Library for Mali GPU
nnet.onnx.layer.VerifyBatchSizeLayer (Deep Learning Toolbox)	Verify fixed batch size.	Yes	Yes	Yes

Custom Layers

Layer Name	Description	cuDNN	TensorRT	ARM Compute Library for Mali GPU
 Custom layers	<p>Custom layers, with or without learnable parameters, that you define for your problem.</p> <p>To learn how to define custom deep learning layers, see “Define Custom Deep Learning Layers” (Deep Learning Toolbox) and “Define Custom Deep Learning Layer for Code Generation” (Deep Learning Toolbox).</p> <p>For an example on how to generate code for a network with custom layers, see “Code Generation For Object Detection Using YOLO v3 Deep Learning” on page 4-217.</p> <p>The outputs of the custom layer must be fixed-size arrays.</p> <p>Using 'unified' as the MallocMode in <code>coder.gpuConfig</code> requires extra memory copies leading to slower performance. For custom layers, it is recommended to use 'discrete'</p>	Yes	Yes	No

Layer Name	Description	cuDNN	TensorRT	ARM Compute Library for Mali GPU
	<p>mode. For more information on GPU memory allocation, see “Discrete and Managed Modes” on page 2-28</p> <p>cuDNN targets support both row-major and column-major code generation for custom layers. TensorRT targets support only column-major code generation.</p> <p>For code generation, custom layers must contain the <code>##codegen</code> pragma.</p> <p>Code generation for a sequence network containing custom layer and LSTM or GRU layer is not supported.</p> <p>You can pass <code>dlarray</code> to custom layers if:</p> <ul style="list-style-type: none"> • The custom layer is in <code>dlnetwork</code>. • Custom layer is in a DAG or series network and either inherits from <code>nnet.layer.Formatable</code> or has no 			

Layer Name	Description	cuDNN	TensorRT	ARM Compute Library for Mali GPU
	<p>backward propagation.</p> <p>For unsupported <code>dlarray</code> methods, then you must extract the underlying data from the <code>dlarray</code>, perform the computations and reconstruct the data back into the <code>dlarray</code> for code generation. For example,</p> <pre>function Z = predict(layer, X) if coder.target('MATLAB') Z = doPredict(X); else if isdlarray(X) X1 = extractdata(X); Z1 = doPredict(X1); Z = dlarray(Z1); else Z = doPredict(X); end end end</pre>			

Supported Classes

The following classes are supported for code generation by GPU Coder for the target deep learning libraries specified in the table.

Name	Description	cuDNN	TensorRT	ARM Compute Library for Mali GPU
DAGNetwork	Directed acyclic graph (DAG) network for deep learning <ul style="list-style-type: none">• Only the activations, predict, and classify methods are supported.	Yes	Yes	Yes

Name	Description	cuDNN	TensorRT	ARM Compute Library for Mali GPU
dlnetwork	<p>Deep learning network for custom training loops</p> <ul style="list-style-type: none"> • Code generation supports only the <code>InputNames</code> and <code>OutputNames</code> properties. • The <code>Initialized</code> property of the <code>dlnetwork</code> object must be set to true. • You can generate code for <code>dlnetwork</code> that have vector and image sequence inputs. Code generation support includes: <ul style="list-style-type: none"> • <code>dlarray</code> containing vector sequences that have 'CT' or 'CBT' data formats. • <code>dlarray</code> containing image sequences that have 'SSCT' or 'SSCBT' data formats. 	Yes	Yes	No

Name	Description	cuDNN	TensorRT	ARM Compute Library for Mali GPU
	<ul style="list-style-type: none"> • Multi-input dlnetwork with heterogeneous input layers. For RNN networks, multiple input is not supported. • Code generation supports only the predict object function. The dlarray input to the predict method must be a single datatype. • Code generation supports dlnetwork for cuDNN and TensorRT targets. Code generation does not support dlnetwork for ARM Mali targets. • When targeting TensorRT with INT8 precision, the last layer(s) of the network must be a softmaxLayer layer. • Code generation supports MIMO dlnetworks. 			

Name	Description	cuDNN	TensorRT	ARM Compute Library for Mali GPU
	<ul style="list-style-type: none"> To create a <code>dlNetwork</code> object for code generation, see “Load Pretrained Networks for Code Generation” on page 4-66. 			
pointPillarsObjectDetector	<p>PointPillars network to detect objects in lidar point clouds</p> <ul style="list-style-type: none"> Only the <code>detect</code> method of the <code>pointPillarsObjectDetector</code> or is supported for code generation. Only the <code>Threshold</code>, <code>SelectStrongest</code>, and <code>MiniBatchSize</code> Name-Value pairs of the <code>detect</code> method are supported. 	Yes	Yes	No
SeriesNetwork	<p>Series network for deep learning</p> <ul style="list-style-type: none"> Only the <code>activations</code>, <code>classify</code>, <code>predict</code>, <code>predictAndUpdateState</code>, <code>classifyAndUpdateState</code>, and <code>resetState</code> object functions are supported. 	Yes	Yes	Yes

Name	Description	cuDNN	TensorRT	ARM Compute Library for Mali GPU
ssdObjectDetector	<p>Detect objects using the SSD-based detector.</p> <ul style="list-style-type: none"> • Only the detect method of the ssdObjectDetector is supported for code generation. • The roi argument to the detect method must be a codegen constant (coder.const()) and a 1x4 vector. • Only the Threshold, SelectStrongest, MinSize, MaxSize, and MiniBatchSize Name-Value pairs are supported. All Name-Value pairs must be compile-time constants. • The channel and batch size of the input image must be fixed size. • The labels output is returned as a categorical array. • In the generated code, the input is 	Yes	Yes	No

Name	Description	cuDNN	TensorRT	ARM Compute Library for Mali GPU
	<p>rescaled to the size of the input layer of the network. But the bounding box that the detect method returns is in reference to the original input size.</p> <ul style="list-style-type: none">• The bounding boxes might not numerically match the simulation results.			

Name	Description	cuDNN	TensorRT	ARM Compute Library for Mali GPU
yolov2ObjectDetector	<p>Detect objects using YOLO v2 object detector</p> <ul style="list-style-type: none"> • Only the detect method of the yolov2ObjectDetector is supported for code generation. • The roi argument to the detect method must be a codegen constant (coder.constant()) and a 1x4 vector. • Only the Threshold, SelectStrongest, MinSize, MaxSize, and MiniBatchSize Name-Value pairs are supported. • The height, width, channel, and batch size of the input image must be fixed size. • The minimum batch size value passed to detect method must be fixed size. 	Yes	Yes	Yes

Name	Description	cuDNN	TensorRT	ARM Compute Library for Mali GPU
yolov3ObjectDetector	<p>Detect objects using YOLO v3 object detector</p> <ul style="list-style-type: none"> • Only the detect method of the yolov3ObjectDetector is supported for code generation. • The roi argument to the detect method must be a codegen constant (coder.constant()) and a 1x4 vector. • Only the Threshold, SelectStrongest, MinSize, MaxSize, and MiniBatchSize Name-Value pairs are supported. • The height, width, channel, and batch size of the input image must be fixed size. • The minimum batch size value passed to detect method must be fixed size. 	Yes	Yes	No

Name	Description	cuDNN	TensorRT	ARM Compute Library for Mali GPU
yolov4ObjectDetector	Detect objects using YOLO v4 object detector <ul style="list-style-type: none"> • Only the detect method of the yolov3ObjectDetector is supported for code generation. • The roi argument to the detect method must be a code generation constant (coder.constant()) and a 1x4 vector. • Only the Threshold, SelectStrongest, MinSize, MaxSize, and MiniBatchSize name-value pairs for detect are supported. 	Yes	Yes	No

See Also

Functions

`coder.getDeepLearningLayers` | `codegen` | `coder.DeepLearningConfig`

Objects

`coder.gpuConfig` | `coder.CodeConfig` | `coder.EmbeddedCodeConfig` | `coder.gpuEnvConfig` | `coder.CuDNNConfig` | `coder.TensorRTConfig`

More About

- “Pretrained Deep Neural Networks” (Deep Learning Toolbox)
- “Get Started with Transfer Learning” (Deep Learning Toolbox)
- “Create Simple Deep Learning Network for Classification” (Deep Learning Toolbox)

- “Load Pretrained Networks for Code Generation” on page 4-66
- “Code Generation for Deep Learning Networks by Using cuDNN” on page 4-69
- “Code Generation for Deep Learning Networks by Using TensorRT” on page 4-78
- “Code Generation for Deep Learning Networks Targeting ARM Mali GPUs” on page 4-88

Analyze Network for Code Generation

You can analyze code generation compatibility of deep learning networks by using the `analyzeNetworkForCodegen` function. Use the network code generation analyzer to validate a `SeriesNetwork`, `DAGNetwork`, and `dlnetwork` for non-library and library targets and detect problems before code generation. Supported library targets include MKL-DNN, ARM Compute, CMSIS-NN, ARM Compute Mali, cuDNN, and TensorRT. Problems that `analyzeNetworkForCodegen` detects include unsupported layers for code generation, network issues, built-in layer specific issues, and issues with custom layers.

The `analyzeNetworkForCodegen` function requires the MATLAB Coder Interface for Deep Learning Libraries and GPU Coder Interface for Deep Learning Libraries support packages. To download and install support package, use the Add-On Explorer. You can also download the support packages from MathWorks GPU Coder Team and MathWorks MATLAB Coder Team.

Check `dlnetwork` for Code Generation Compatibility

This example shows how to check code generation compatibility of multi-input `dlnetwork` by using the `analyzeNetworkForCodegen` function.

The example checks code generation support for the following targets:

- Library free code generation.
- ARM® Compute Library.
- ARM Compute Library for Mali GPU.
- Intel® Math Kernel Library for Deep Neural Networks (Intel MKL-DNN).
- Common Microcontroller Software Interface Standard - Neural Network (CMSIS-NN) library.
- NVidia® CUDA® Deep Neural Network library (cuDNN).
- NVIDIA TensorRT high performance deep learning inference optimizer and run-time library.

Define Network Architecture

Construct a network with two branches. The network takes two inputs, with one input per branch. Connect the branches using an addition layer.

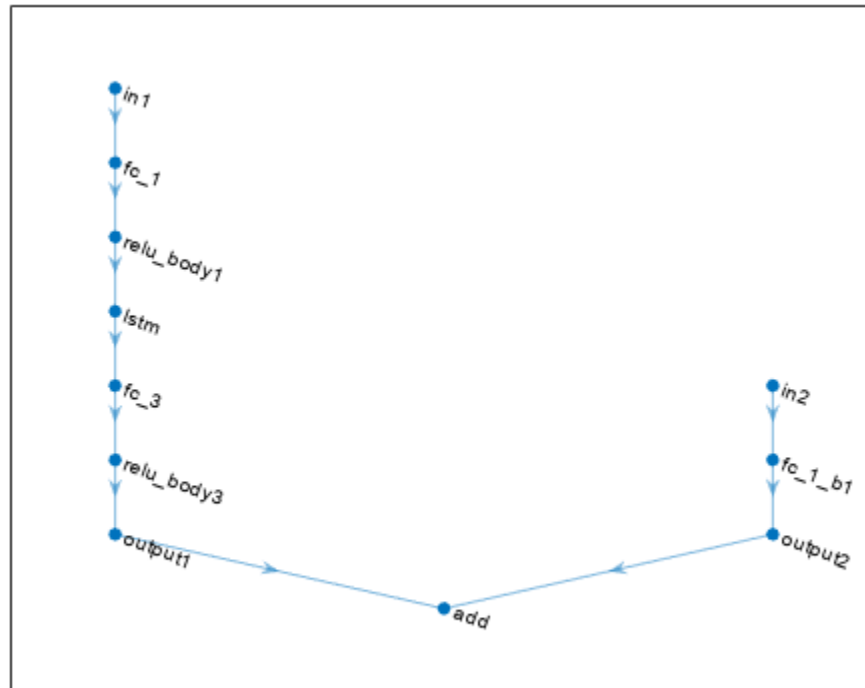
```
layersBranch1 = [
    sequenceInputLayer(1,"Name","in1","Normalization","none")
    fullyConnectedLayer(32,"Name","fc_1")
    reluLayer("Name","relu_body1")
    lstmLayer(32,"Name","lstm","OutputMode","last")
    fullyConnectedLayer(32,"Name","fc_3")
    reluLayer("Name","relu_body3")
    fullyConnectedLayer(4,"Name","output1")
    additionLayer(2,"Name","add")];

layersBranch2 = [
    imageInputLayer([5 5 3],"Name","in2","Normalization","none")
    fullyConnectedLayer(4,"Name","fc_1_b1")
    reluLayer("Name","output2")];

lgraph = layerGraph(layersBranch1);
```

```
lgraph = addLayers(lgraph, layersBranch2);
lgraph = lgraph.connectLayers('output2', 'add/in2');
```

```
figure
plot(lgraph)
```



Create the dlnetwork.

```
dlnet = dlnetwork(lgraph);
```

Analyze Network for Code Generation

Run the `analyzeNetworkForCodegen` function for `mobilenetv2`, specifying the target libraries to analyze. The `analyzeNetworkForCodegen` function requires the *MATLAB® Coder™ Interface for Deep Learning Libraries* and the *GPU Coder™ Interface for Deep Learning Libraries* support packages. To install the required support packages, use the Add-On Explorer.

```
targetLibraries = {'none', 'arm-compute', 'arm-compute-mali', ...
    'mkl-dnn', 'cmsis-nn', 'cudnn', 'tensorrt'};
S = analyzeNetworkForCodegen(dlnet, TargetLibrary = targetLibraries);
```

	Supported		
none	"Yes"	" "	
arm-compute	"Yes"	" "	
arm-compute-mali	"No"	"Found 1 issue(s). View network diagnostics."	"Found 2 u
mkl-dnn	"Yes"	" "	

```

cmsis-nn          "No"          "Found 1 issue(s). View network diagnostics."    "Found 2 u
cudnn            "Yes"         ""
tensorrt         "Yes"         ""

```

Display the layer diagnostics for CMSIS-NN code generation.

```
S(5).LayerDiagnostics
```

```
ans=4x3 table
  LayerName      LayerType      Diagnostics
  _____  _____  _____
  "add"          "AdditionLayer"  "Unsupported layer type."
  "relu_body1"  "ReLULayer"      "Unsupported layer type."
  "relu_body3"  "ReLULayer"      "Unsupported layer type."
  "output2"     "ReLULayer"      "Unsupported layer type."
```

Display the network diagnostics for CMSIS-NN code generation.

```
S(5).NetworkDiagnostics.Diagnostics
```

```
ans =
"Code generation for cmsis-nn library does not support dlnetwork objects with combinations of se
```

Analyze Classification Network for Code Generation Compatibility

This example shows how to create a simple convolutional neural network for deep learning classification and test the network for code generation compatibility. The example demonstrates how to:

- Load and explore image data.
- Define the network architecture.
- Specify training options and train the network.
- Predict the labels of new data and calculate the classification accuracy.
- Analyze the deep learning network for code generation and report network and layer compatibility issues by using `analyzeNetworkForCodegen`.

The `analyzeNetworkForCodegen` function requires the *MATLAB® Coder™ Interface for Deep Learning Libraries* and the *GPU Coder™ Interface for Deep Learning Libraries* support packages. To install the required support packages, use the Add-On Explorer.

Load and Explore Image Data

Load the digit sample data as an image datastore. Display some of the images in the datastore.

```

digitDatasetPath = fullfile(matlabroot,'toolbox','nnet','nndemos', ...
    'nndatasets','DigitDataset');
imds = imageDatastore(digitDatasetPath, ...
    'IncludeSubfolders',true,'LabelSource','foldernames');

figure;
perm = randperm(10000,20);
for i = 1:20

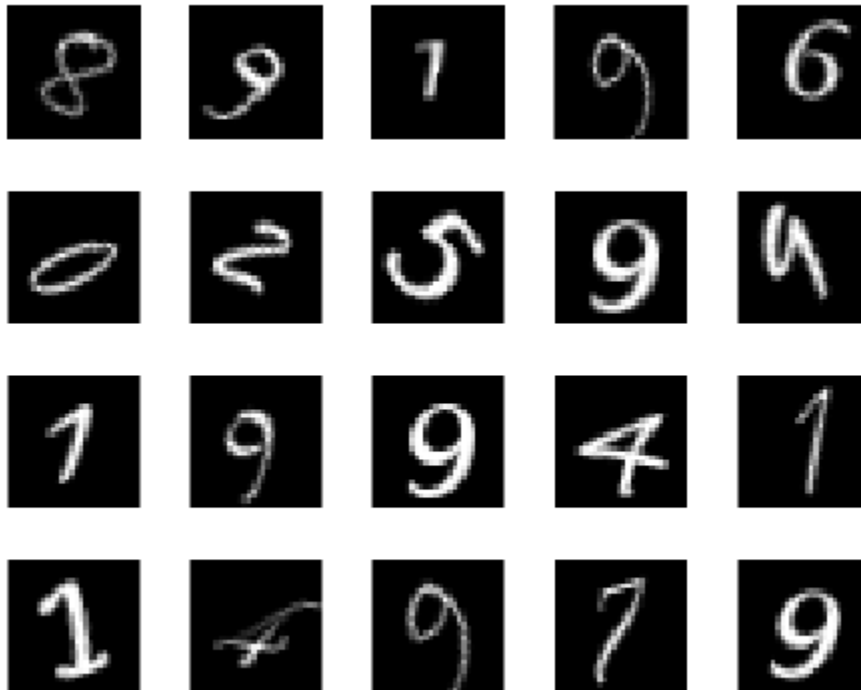
```



```

subplot(4,5,i);
imshow(imds.Files{perm(i)});
end

```



Check the size of the first image in `digitData`. Each image is 28-by-28-by-1 pixels.

```

img = readimage(imds,1);
size(img)

```

```

ans = 1x2
      28      28

```

Divide the data into training and validation data sets, so that each category in the training set contains 750 images, and the validation set contains the remaining images from each label.

```

numTrainFiles = 750;
[imdsTrain,imdsValidation] = splitEachLabel(imds,numTrainFiles,'randomize');

```

Define Network Architecture

Define the convolutional neural network architecture.

```

layers = [
    imageInputLayer([28 28 1])
    convolution2dLayer(3,8,'PaddingValue',5,'Name','conv1')

```

```
batchNormalizationLayer
reluLayer

maxPooling2dLayer(2, 'Stride', 2)

convolution2dLayer(3, 16, 'Padding', 'same', 'Name', 'conv2')
batchNormalizationLayer
reluLayer

maxPooling2dLayer(2, 'Stride', 2)

convolution2dLayer(3, 32, 'Padding', 'same', 'Name', 'conv3')
batchNormalizationLayer
reluLayer

fullyConnectedLayer(10)
softmaxLayer
classificationLayer];
```

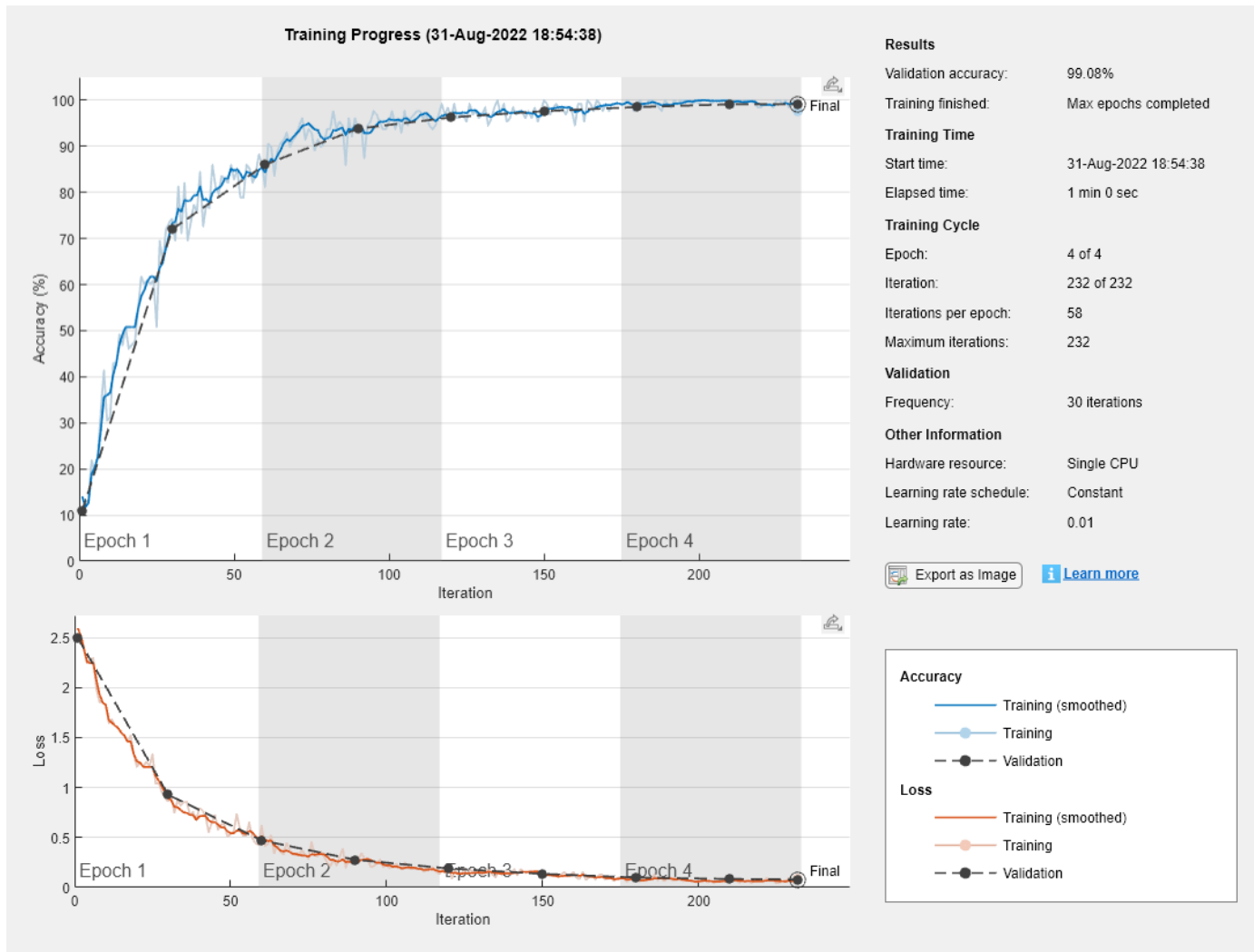
Specify Training Options and Train Network

Train the network using stochastic gradient descent with momentum (SGDM) with an initial learning rate of 0.01. Set the maximum number of epochs to 4. Monitor the network accuracy during training by specifying validation data and validation frequency. Shuffle the data every epoch. Turn on the training progress plot, and turn off the command window output.

```
options = trainingOptions('sgdm', ...
    'InitialLearnRate', 0.01, ...
    'MaxEpochs', 4, ...
    'Shuffle', 'every-epoch', ...
    'ValidationData', imdsValidation, ...
    'ValidationFrequency', 30, ...
    'Verbose', false, ...
    'Plots', 'training-progress');
```

Train the network using the architecture defined by `layers`, the training data, and the training options. The training progress plot shows the mini-batch loss and accuracy and the validation loss and accuracy.

```
net = trainNetwork(imdsTrain, layers, options);
```



Classify Validation Images and Compute Accuracy

Predict the labels of the validation data using the trained network, and calculate the final validation accuracy. Accuracy is the fraction of labels that the network predicts correctly. In this case, more than 99% of the predicted labels match the true labels of the validation set.

```
YPred = classify(net, imdsValidation);
YValidation = imdsValidation.Labels;

accuracy = sum(YPred == YValidation)/numel(YValidation)

accuracy = 0.9908
```

Analyze Network for Code Generation

To check the network for code generation compatibility, run `analyzeNetworkForCodegen`. By default, the function validates against a set of default CPU and GPU deep learning library targets. `analyzeNetworkForCodegen` returns a 1-by-N structure containing the analysis results.

```
S = analyzeNetworkForCodegen(net)
```

	Supported	
none	"No"	"Found 1 issue(s) in 1 layer(s). View layer diagnostics."
arm-compute	"No"	"Found 1 issue(s) in 1 layer(s). View layer diagnostics."
mkldnn	"No"	"Found 1 issue(s) in 1 layer(s). View layer diagnostics."
cuda	"No"	"Found 1 issue(s) in 1 layer(s). View layer diagnostics."
tensorrt	"No"	"Found 1 issue(s) in 1 layer(s). View layer diagnostics."

S=1x5 struct array with fields:

```
TargetLibrary
Supported
NetworkDiagnostics
LayerDiagnostics
IncompatibleLayerTypes
```

To view the layer issues in the network for the cuDNN target, use the following command. Alternatively, you can click on the [View layer diagnostics](#) hyperlink to display the layer issues.

S(4).LayerDiagnostics

ans=1x3 table

LayerName	LayerType	
"conv1"	"Convolution2DLayer"	"Layer 'conv1' has a non-default padding value. Code gen

The first convolution2dLayer (conv1) has non-zero padding value. For code generation, the `PaddingValue` parameter must be equal to 0, which is the default value.

Fix Network Issues and Retrain

In this example, the padding value of the convolution layer can be set to zero.

```
layers(2) = convolution2dLayer(3,8,'PaddingValue',0,'Name','conv1');
```

Retrain the modified network using the following training options.

```
options = trainingOptions('sgdm', ...
    'InitialLearnRate',0.01, ...
    'MaxEpochs',4, ...
    'Shuffle','every-epoch', ...
    'ValidationData',imdsValidation, ...
    'ValidationFrequency',30, ...
    'Verbose',false, ...
    'Plots','none');
net = trainNetwork(imdsTrain, layers, options);
```

Check the modified network for code generation compatibility.

```
S = analyzeNetworkForCodegen(net)
```

	Supported
none	"Yes"
arm-compute	"Yes"

```
mklDnn      "Yes"  
cudnn       "Yes"  
tensorrt    "Yes"
```

S=1x5 struct array with fields:

```
TargetLibrary  
Supported  
NetworkDiagnostics  
LayerDiagnostics  
IncompatibleLayerTypes
```

The `analyzeNetworkForCodegen` function reports no issues. The network is now ready for code generation.

See Also

Functions

`analyzeNetworkForCodegen` | `codegen` | `cnncodegen` | `coder.loadDeepLearningNetwork`

Related Examples

- “Supported Networks, Layers, and Classes” on page 4-6
- “Code Generation for Deep Learning Networks by Using cuDNN” on page 4-69
- “Code Generation for Deep Learning Networks by Using TensorRT” on page 4-78
- “Code Generation for Deep Learning Networks Targeting ARM Mali GPUs” on page 4-88

Code Generation for dlarray

In this section...

“Define dlarray for Code Generation” on page 4-52

“dlarray Object Functions with Code Generation Support” on page 4-53

“Deep Learning Toolbox Functions with dlarray Code Generation Support” on page 4-54

“MATLAB Functions with dlarray Code Generation Support” on page 4-54

A deep learning array stores data with optional data format labels for custom training loops, and enables functions to compute and use derivatives through automatic differentiation. To learn more about custom training loops, automatic differentiation, and deep learning arrays, see “Deep Learning Custom Training Loops” (Deep Learning Toolbox).

Code generation supports both formatted and unformatted deep learning arrays. `dlarray` objects containing `gpuArrays` are also supported for code generation. When you use deep learning arrays with CPU and GPU code generation, adhere to these restrictions:

Define dlarray for Code Generation

For code generation, use the `dlarray` function to create deep learning arrays. For example, suppose you have a pretrained `dlnetwork` network object in the `mynet.mat` MAT-file. To predict the responses for this network, create an entry-point function in MATLAB.

There are two possibilities:

Note For code generation, the `dlarray` input to the `predict` method of the `dlnetwork` object must be `single` data type.

Design 1 (Not recommended)

In this design example, the input and output to the entry-point function, `foo` are of `dlarray` types. This type of entry-point function is not recommended for code generation because in MATLAB, `dlarray` enforces the order of labels 'SCBTU'. This behavior is replicated for MEX code generation. However, for standalone code generation such as static, dynamic libraries, or executables, the data format follows the specification of the `fmt` argument of the `dlarray` object. As a result, if the input or output of an entry-point function is a `dlarray` object and its order of labels is not 'SCBTU', then the data layout will be different between the MATLAB environment and standalone code.

```
function dlOut = foo(dlIn)

persistent dlnet;
if isempty(dlnet)
    dlnet = coder.loadDeepLearningNetwork('mynet.mat');
end

dlOut = predict(dlnet, dlIn);

end
```

Design 2 (Recommended)

In this design example, the input and output to `foo` are of primitive datatypes and the `dlarray` object is created within the function. The `extractdata` method of the `dlarray` object returns the data in the `dlarray` `dIA` as the output of `foo`. The output `a` has the same data type as the underlying data type in `dIA`.

When compared to Design 1, this entry-point design has the following advantages:

- Easier integration with standalone code generation workflows such as static, dynamic libraries, or executables.
- The data format of the output from the `extractdata` function has the same order ('SCBTU') in both the MATLAB environment and the generated code.
- Improves performance for MEX workflows.
- Simplifies Simulink workflows using MATLAB Function blocks as Simulink does not natively support `dlarray` objects.

```
function a = foo(in)
    dlIn = dlarray(in, 'SSC');

    persistent dlnet;
    if isempty(dlnet)
        dlnet = coder.loadDeepLearningNetwork('mynet.mat');
    end

    dIA = predict(dlnet, dlIn);

    a = extractdata(dIA);

end
```

To see an example of `dlnetwork` and `dlarray` usage with GPU Coder, see “Generate Digit Images on NVIDIA GPU Using Variational Autoencoder” on page 4-222.

dlarray Object Functions with Code Generation Support

For code generation, you are restricted to the deep learning array object functions listed in this table.

<code>dims</code>	Dimension labels for <code>dlarray</code>
<code>extractdata</code>	Extract data from <code>dlarray</code>
<code>finddim</code>	Find dimensions with specified label
<code>stripdims</code>	Remove <code>dlarray</code> labels

Deep Learning Toolbox Functions with dlarray Code Generation Support

Deep Learning Operations

Function	Description
fullyconnect	The fully connect operation multiplies the input by a weight matrix and then adds a bias vector.
sigmoid	The sigmoid activation operation applies the sigmoid function to the input data.
softmax	The softmax activation operation applies the softmax function to the channel dimension of the input data.

MATLAB Functions with dlarray Code Generation Support

Unary Element-wise Functions

Function	Notes and Limitations
abs	The output dlarray has the same data format as the input dlarray.
atan2	The output dlarray has the same data format as the input dlarray.
cos	
cosh	
cot	
csc	
exp	
log	<ul style="list-style-type: none"> The output dlarray has the same data format as the input dlarray. Because dlarray does not support complex numbers, the input dlarray must have nonnegative values.
sec	The output dlarray has the same data format as the input dlarray.
sign	
sin	
sinh	
sqrt	<ul style="list-style-type: none"> The output dlarray has the same data format as the input dlarray. Because dlarray does not support complex numbers, the input dlarray must have nonnegative values.
tan	The output dlarray has the same data format as the input dlarray.
tanh	

Function	Notes and Limitations
uplus, +	
uminus, -	
erf	

Binary Element-wise Operators

Function	Notes and Limitations
minus, -	If the two dlarray inputs are formatted, then the output dlarray is formatted with a combination of both of their data formats. The function uses implicit expansion to combine the inputs. For more information, see “Implicit Expansion with Data Formats” (Deep Learning Toolbox).
plus, +	
power, .^	
rdivide, ./	
times, .*	

Reduction Functions

Function	Notes and Limitations
mean	<ul style="list-style-type: none"> The output dlarray has the same data format as the input dlarray. The 'omitnan' option is not supported. If the input dlarray is on the GPU, the 'native' option is not supported.
prod	<ul style="list-style-type: none"> The output dlarray has the same data format as the input dlarray. The 'omitnan' option is not supported.
sum	

Extrema Functions

Function	Notes and Limitations
ceil	The output dlarray has the same data format as the input dlarray.
eps	<ul style="list-style-type: none"> The output dlarray has the same data format as the input dlarray. Use eps(ones('like', x)) to get a scalar epsilon value based on the data type of a dlarray x.
fix	The output dlarray has the same data format as the input dlarray.
floor	The output dlarray has the same data format as the input dlarray.
max	<ul style="list-style-type: none"> When you find the maximum or minimum elements of a single dlarray, the output dlarray has the same data format as the input dlarray.

Function	Notes and Limitations
<code>min</code>	<ul style="list-style-type: none"> When you find the maximum or minimum elements between two formatted <code>darray</code> inputs, the output <code>darray</code> has a combination of both of their data formats. The function uses implicit expansion to combine the inputs. For more information, see “Implicit Expansion with Data Formats” (Deep Learning Toolbox). The index output argument is not traced and cannot be used with automatic differentiation. For more information, see “Use Automatic Differentiation In Deep Learning Toolbox” (Deep Learning Toolbox).
<code>round</code>	<ul style="list-style-type: none"> Only the syntax <code>Y = round(X)</code> is supported. The output <code>darray</code> has the same data format as the input <code>darray</code>.

Other Math Operations

Function	Notes and Limitations
<code>colon, :</code>	<ul style="list-style-type: none"> The supported operations are: <ul style="list-style-type: none"> <code>a:b</code> <code>a:b:c</code> For information on indexing into a <code>darray</code>, see “Indexing” (Deep Learning Toolbox). All inputs must be real scalars. The output <code>darray</code> is unformatted.
<code>mtimes, *</code>	<ul style="list-style-type: none"> One input can be a formatted <code>darray</code> only when the other input is an unformatted scalar. In this case, the output <code>darray</code> has the same data format as the formatted <code>darray</code> input. Multiplying a <code>darray</code> with a non-<code>darray</code> sparse matrix is supported only when both inputs are non-scalar.
<code>pagetimes</code>	<ul style="list-style-type: none"> One input can be a formatted <code>darray</code> only when the other input is unformatted, with scalar pages. In this case, the output <code>darray</code> has the same data format as the formatted <code>darray</code> input. For code generation, each transpose option of <code>pagetimes</code> must be constant.

Logical Operations

Function	Notes and Limitations
and, &	If the two <code>dlarray</code> inputs are formatted, then the output <code>dlarray</code> is formatted with a combination of both of their data formats. The function uses implicit expansion to combine the inputs. For more information, see “Implicit Expansion with Data Formats” (Deep Learning Toolbox).
eq, ==	
ge, >=	
gt, >	
le, <=	
lt, <	
ne, ~=	If the two <code>dlarray</code> inputs are formatted, then the output <code>dlarray</code> is formatted with a combination of both of their data formats. The function uses implicit expansion to combine the inputs. For more information, see “Implicit Expansion with Data Formats” (Deep Learning Toolbox).
or,	
xor	

Size Manipulation Functions

Function	Notes and Limitations
reshape	The output <code>dlarray</code> is unformatted, even if the input <code>dlarray</code> is formatted. For code generation, the size dimensions must be fixed size.
squeeze	Two-dimensional <code>dlarray</code> objects are unaffected by <code>squeeze</code> . If the input <code>dlarray</code> is formatted, the function removes dimension labels belonging to singleton dimensions. If the input <code>dlarray</code> has more than two dimensions and its third and above dimensions are singleton, then the function discards these dimensions and their labels.

Function	Notes and Limitations
repelem	<p>If you use the <code>u = repelem(v,n)</code> syntax and specify the number of times to repeat each element in <code>repelem</code>, the output <code>darray</code> is unformatted even if the input <code>darray</code> is formatted.</p> <p>If you use the <code>B = repelem(A,r1,...,rN)</code> syntax and specify the repetition factors for each dimension in <code>repelem</code>, the output <code>darray</code> has the same data format as the input <code>darray</code>.</p>
repmat	The output <code>darray</code> has the same data format as the input <code>darray</code> .

Transposition Operations

Function	Notes and Limitations
<code>ctranspose, '</code>	If the input <code>darray</code> is formatted, then the labels of both dimensions must be the same. The function performs transposition implicitly, and transposes directly only if necessary for other operations.
<code>permute</code>	<p>If the input <code>darray</code> is formatted, then the permutation must be among only those dimensions that have the same label. The function performs permutations implicitly, and permutes directly only if necessary for other operations.</p> <p>For code generation, the dimension order must be fixed size.</p>
<code>ipermute</code>	<p>If the input <code>darray</code> is formatted, then the permutation must be among only those dimensions that have the same label. The function performs permutations implicitly, and permutes directly only if necessary for other operations.</p> <p>For code generation, the dimension order must be fixed size.</p>
<code>transpose, .'</code>	If the input <code>darray</code> is formatted, then the labels of both dimensions must be the same. The function performs transposition implicitly, and transposes directly only if necessary for other operations.

Concatenation Functions

Function	Notes and Limitations
cat	<p>The dlarray inputs must have matching formats or be unformatted. Mixed formatted and unformatted inputs are supported. If any dlarray inputs are formatted, then the output dlarray is formatted with the same data format.</p> <p>For code generation, the dimension order to cat function must be fixed size.</p>
horzcat	
vertcat	

Conversion Functions

Function	Notes and Limitations
cast	<ul style="list-style-type: none"> cast(dIA, newdatatype) copies the data in the dlarray dIA into a dlarray of the underlying data type newdatatype. The newdatatype option must be 'double', 'single', or 'logical'. The output dlarray is formatted with the same data format as dIA. cast(A, 'like', Y) returns an array of the same type as Y. If Y is a dlarray, then the output is a dlarray that has the same underlying data type as Y. If Y is on the GPU, then the output is on the GPU. If both A and Y are dlarray objects, then the output dlarray is formatted with the same data format as the input A.
double	The output is a dlarray that contains data of type double.
logical	The output is a dlarray that contains data of type logical.
single	The output is a dlarray that contains data of type single.

Comparison Functions

Function	Notes and Limitations
isequal	<ul style="list-style-type: none"> The syntax with more than two input arguments is not supported. Two dlarray inputs are equal if the numeric data they represent are equal and if they both are either formatted with the same data format or unformatted.

Function	Notes and Limitations
<code>isequaln</code>	<ul style="list-style-type: none"> The syntax with more than two input arguments is not supported. Two <code>darray</code> inputs are equal if the numeric data they represent are equal (treating NaNs as equal) and if they both are either formatted with the same data format or unformatted.

Data Type and Value Identification Functions

Function	Notes and Limitations
<code>isfloat</code>	The software applies the function to the underlying data of an input <code>darray</code> .
<code>islogical</code>	
<code>isnumeric</code>	
<code>isreal</code>	Because <code>darray</code> does not support complex numbers, this function always returns <code>true</code> for a <code>darray</code> input.

Size Identification Functions

Function	Notes and Limitations
<code>iscolumn</code>	This function returns <code>true</code> for a <code>darray</code> that is a column vector, where each dimension except the first is a singleton. For example, a 3-by-1-by-1 <code>darray</code> is a column vector.
<code>ismatrix</code>	This function returns <code>true</code> for <code>darray</code> objects with only two dimensions and for <code>darray</code> objects where each dimension except the first two is a singleton. For example, a 3-by-4-by-1 <code>darray</code> is a matrix.
<code>isrow</code>	This function returns <code>true</code> for a <code>darray</code> that is a row vector, where each dimension except the second is a singleton. For example, a 1-by-3-by-1 <code>darray</code> is a row vector.
<code>isscalar</code>	N/A
<code>isvector</code>	This function returns <code>true</code> for a <code>darray</code> that is a row vector or column vector. Note that <code>isvector</code> does not consider a 1-by-1-by-3 <code>darray</code> to be a vector.
<code>length</code>	N/A
<code>ndims</code>	If the input <code>darray</code> <code>dLX</code> is formatted, then <code>ndims(dLX)</code> returns the number of dimension labels, even if some of the labeled dimensions are trailing singleton dimensions.
<code>numel</code>	N/A

Function	Notes and Limitations
size	If the input dlarray dLX is formatted, then size(dLX) returns a vector of length equal to the number of dimension labels, even if some of the labeled dimensions are trailing singleton dimensions.

Creator Functions

Function	Notes and Limitations
false	Only the 'like' syntax is supported for dlarray.
inf	
nan	
ones	
rand	
true	
zeros	

Indexing

Code generation supports indexing dlarray objects and exhibits the following behaviors:

- If you set `dLY(idx1,...,idxn) = dLX`, then dLY and dLX must be assignment compatible.
 - Size of the data must not change. Out-of-bounds assignment operation is not supported.
 - The assignment statement cannot add or drop U labels.
- Code generation does not support deleting of parts of a dlarray object by using `dLX(idx1,...,idxn) = []`.

See Also

Objects

dlarray | dlnetwork

Related Examples

- “Generate Digit Images on NVIDIA GPU Using Variational Autoencoder” on page 4-222

More About

- “dlarray Limitations for Code Generation” on page 4-62
- “Define Custom Training Loops, Loss Functions, and Networks” (Deep Learning Toolbox)
- “Train Network Using Custom Training Loop” (Deep Learning Toolbox)
- “Make Predictions Using dlnetwork Object” (Deep Learning Toolbox)

dlarray Limitations for Code Generation

In this section...

“Recommended Usage” on page 4-62

“Limitations” on page 4-62

Recommended Usage

For code generation, use the `dlarray` function to create deep learning arrays. For example, suppose you have a pretrained `dlnetwork` network object in the `mynet.mat` MAT-file. To predict the responses for this network, create an entry-point function in MATLAB as shown in this code.

```
function a = foo(in)
dlIn = dlarray(in, 'SSC');

persistent dlnet;
if isempty(dlnet)
    dlnet = coder.loadDeepLearningNetwork('mynet.mat');
end

dIA = predict(dlnet, dlIn);

a = extractdata(dIA);

end
```

Limitations

For deep learning arrays, code generation has the following limitations:

- The data format argument of the `dlarray` object must be a compile-time constant. For example,

```
function out = foo()

dIA = dlarray(ones(5,4), 'SSC'); %fmt 'SSC' is constant
.
.
.
end
```

- The data input to the `dlarray` object must be fixed-size. For example, the `dlarray` `dIA` is not supported as `A` is variable-sized.

```
function dIA = foo()

A = ones(5,4);
coder.varsize('A') %'A' is variable sized.

dIA = dlarray(A, 'SSC'); % Error: not supported.

end
```

- Code generation does not support creating a `dlarray` type object by using the `coder.typeof` function with upper bound size and variable dimensions specified. For example,


```
function dIA = foo()
A = dlarray(ones(5,4), 'SC');
A_type = coder.typeof(A,[5 10],[1 0]); % Error: not supported.
end
```

Code generation supports use of `coder.typeof` without the size arguments. For example,

```
A = dlarray(ones(5,4), 'SC');
A_type = coder.typeof(A);
```

- The code generation report does not display the size of the `dlarray` object. The size is always displayed as `1x1`.

```
7 end
8
9 % generate random noise
10 randomNoise = dlarray(randn(1,1,latentDim,25), 'SSCB');
11
12 if coder.target('MATLAB') && strcmp(Environment, 'gpu')
13     randomNoise = gpuArray(randomNoise);
```

EXPRESSION INFO
dlarray(randn(1,1,latentDim,25), 'SSCB')
Size: 1 x 1
Class: dlarray

- In MATLAB, `dlarray` enforces the order of labels 'SCBTU'. This enforcement eliminates ambiguous semantics in operations, which implicitly match labels between inputs. This behavior is mimicked during MEX code generation. However, for standalone code generation such as static, dynamic libraries, or executables, the data format follows the specification of the `fmt` argument of the `dlarray` object. As a result, if the input or output of an entry-point function is a `dlarray` object and its order of labels is not 'SCBTU', then the data layout will be different between the MATLAB environment and standalone code.

For example, consider a function `foo` with a `dlarray` object as an output.

```
function dIA = foo()
rng default
dIA = dlarray(rand(5,4), 'BC');
end
```

In MATLAB, `dIA` is 4(C)-by-5(B).

```
dIA =
    4(C) × 5(B) dlarray
    0.8147    0.9058    0.1270    0.9134    0.6324
    0.0975    0.2785    0.5469    0.9575    0.9649
    0.1576    0.9706    0.9572    0.4854    0.8003
    0.1419    0.4218    0.9157    0.7922    0.9595
```

For standalone code generation, `dIA` is 5(B)-by-4(C).

- For code generation, the `dlarray` input to the `predict` method of the `dlnetwork` object must be single data type.

See Also

Objects

`dlarray` | `dlnetwork`

Related Examples

- “Generate Digit Images on NVIDIA GPU Using Variational Autoencoder” on page 4-222

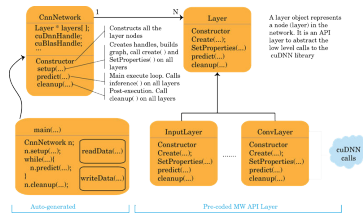
More About

- “Code Generation for dlarray” on page 4-52
- “Define Custom Training Loops, Loss Functions, and Networks” (Deep Learning Toolbox)
- “Train Network Using Custom Training Loop” (Deep Learning Toolbox)
- “Make Predictions Using dlnetwork Object” (Deep Learning Toolbox)

Generated CNN Class Hierarchy

The generated CNN code has the following class hierarchy. The Layer class and the generated Network class have three important methods:

- 1 `setup()`, which allocates memory and system resources for each layer.
- 2 `predict()`, which performs forward inference in the execution loop.
- 3 `cleanup()`, which releases all memory and system resources.



See Also

More About

- “Supported Networks, Layers, and Classes” on page 4-6
- “Load Pretrained Networks for Code Generation” on page 4-66
- “Code Generation for Deep Learning Networks by Using cuDNN” on page 4-69
- “Code Generation for Deep Learning Networks by Using TensorRT” on page 4-78
- “Code Generation for Deep Learning Networks Targeting ARM Mali GPUs” on page 4-88

Load Pretrained Networks for Code Generation

You can generate code for a pretrained convolutional neural network (CNN). To provide the network to the code generator, load a `SeriesNetwork`, `DAGNetwork`, `yolov2objectDetector`, `ssdObjectDetector`, or `dlnetwork` object from the trained network.

Load a Network by Using `coder.loadDeepLearningNetwork`

You can load a network object from any network that is supported for code generation by using `coder.loadDeepLearningNetwork`. You can specify the network from a MAT-file. The MAT-file must contain only the network to be loaded.

For example, suppose that you create a trained network object called `myNet` by using the `trainNetwork` function. Then, you save the workspace by entering `save`. This creates a file called `matlab.mat` that contains the network object. To load the network object `myNet`, enter:

```
net = coder.loadDeepLearningNetwork('matlab.mat');
```

You can also specify the network by providing the name of a function that does not accept an input argument and returns a pretrained `SeriesNetwork`, `DAGNetwork`, `yolov2objectDetector`, or `ssdObjectDetector` object, such as:

- `alexnet`
- `darknet19`
- `darknet53`
- `densenet201`
- `googlenet`
- `inceptionv3`
- `inceptionresnetv2`
- `mobilenetv2`
- `nasnetlarge`
- `nasnetmobile`
- `resnet18`
- `resnet50`
- `resnet101`
- `squeezenet`
- `vgg16`
- `vgg19`
- `xception`

For example, load a network object by entering:

```
net = coder.loadDeepLearningNetwork('googlenet');
```

The Deep Learning Toolbox functions in the previous list require that you install a support package for the function. See “Pretrained Deep Neural Networks” (Deep Learning Toolbox).

Specify a Network Object for Code Generation

If you generate code by using `codegen` or the app, load the network object inside of your entry-point function by using `coder.loadDeepLearningNetwork`. For example:

```
function out = myNet_predict(in) %#codegen
persistent mynet;
if isempty(mynet)
    mynet = coder.loadDeepLearningNetwork('matlab.mat');
end
out = predict(mynet,in);
```

For pretrained networks that are available as support package functions such as `alexnet`, `inceptionv3`, `googlenet`, and `resnet`, you can directly specify the support package function, for example, by writing `mynet = googlenet`.

Next, generate code for the entry-point function. For example:

```
cfg = coder.gpuConfig('mex');
cfg.TargetLang = 'C++';
cfg.DeepLearningConfig = coder.DeepLearningConfig('cudnn');
codegen -args {ones(224,224,3,'single')} -config cfg myNet_predict
```

Specify a dlnetwork Object for Code Generation

Suppose you have a pretrained `dlnetwork` network object in the `mynet.mat` MAT-file. To predict the responses for this network, create an entry-point function in MATLAB as shown in this code.

```
function a = myDLNet_predict(in)
dlIn = dlarray(in, 'SSC');

persistent dlnet;
if isempty(dlnet)
    dlnet = coder.loadDeepLearningNetwork('mynet.mat');
end

dlA = predict(dlnet, dlIn);

a = extractdata(dlA);

end
```

In this example, the input and output to `myDLNet_predict` are of simpler datatypes and the `dlarray` object is created within the function. The `extractdata` method of the `dlarray` object returns the data in the `dlarray` `dlA` as the output of `myDLNet_predict`. The output `a` has the same data type as the underlying data type in `dlA`. This entry-point design has the following advantages:

- Easier integration with standalone code generation workflows such as static, dynamic libraries, or executables.
- The data format of the output from the `extractdata` function has the same order ('SCBTU') in both the MATLAB environment and the generated code.
- Improves performance for MEX workflows.
- Simplifies Simulink workflows using MATLAB Function blocks as Simulink does not natively support `dlarray` objects.

Next, generate code for the entry-point function. For example:

```
cfg = coder.gpuConfig('lib');  
cfg.TargetLang = 'C++';  
cfg.DeepLearningConfig = coder.DeepLearningConfig('cudnn');  
codegen -args {ones(224,224,3,'single')} -config cfg myDLNet_predict
```

See Also

Functions

[codegen](#) | [trainNetwork](#) | [coder.loadDeepLearningNetwork](#)

Objects

[SeriesNetwork](#) | [DAGNetwork](#) | [yolov20objectDetector](#) | [ssdObjectDetector](#) | [dlarray](#) | [dlnetwork](#)

More About

- “Supported Networks, Layers, and Classes” on page 4-6
- “Code Generation for Deep Learning Networks by Using cuDNN” on page 4-69
- “Code Generation for Deep Learning Networks by Using TensorRT” on page 4-78

Code Generation for Deep Learning Networks by Using cuDNN

With GPU Coder, you can generate optimized code for prediction of a variety of trained deep learning networks from Deep Learning Toolbox. The generated code implements the deep convolutional neural network (CNN) by using the architecture, the layers, and parameters that you specify in the input `SeriesNetwork` or `DAGNetwork` object. The code generator takes advantage of NVIDIA CUDA deep neural network library (cuDNN) for NVIDIA GPUs. cuDNN is a GPU-accelerated library of primitives for deep neural networks. The generated code can be integrated into your project as source code, static or dynamic libraries, or executables that you can deploy to a variety of NVIDIA GPU platforms.

Generate code for convolutional networks by using one of the methods:

- The standard `codegen` function that generates CUDA code from a MATLAB entry-point function.
- The GPU Coder app that generates CUDA code from a MATLAB entry-point function.

Note In previous releases you could target the cuDNN library by using the `cnncodegen` function. From R2021a onwards, the `cnncodegen` function generates C++ code and make files for only the ARM Mali GPU processor.

Generate Code and Classify Images by Using GoogLeNet

In this example, you use GPU Coder to generate CUDA code for the pretrained `googlenet` deep convolutional neural network and classify an image. GoogLeNet has been trained on over a million images and can classify images into 1000 object categories (such as keyboard, coffee mug, pencil, and animals). The network has learned rich feature representations for a wide range of images. The network takes an image as input, and then outputs a label for the object in the image together with the probabilities for each of the object categories. This example show you how to generate code for the pretrained network by using the `codegen` command and the GPU Coder app.

Requirements

Required

This example generates CUDA MEX that has the following additional requirements.

- 1 Deep Learning Toolbox.
- 2 Deep Learning Toolbox Model for GoogLeNet Network support package.
- 3 GPU Coder Interface for Deep Learning Libraries support package.
- 4 CUDA enabled NVIDIA GPU and a compatible driver. For 8-bit integer precision, the CUDA GPU must have a compute capability of 6.1 or higher.

Optional

For non-MEX builds such as static, dynamic libraries, or executables, this example has the following additional requirements.

- 1 CUDA Toolkit and cuDNN libraries. For information on the supported versions of the compilers and libraries, see “Installing Prerequisite Products”.
- 2 Environment variables for the compilers and libraries. For more information, see “Environment Variables”.

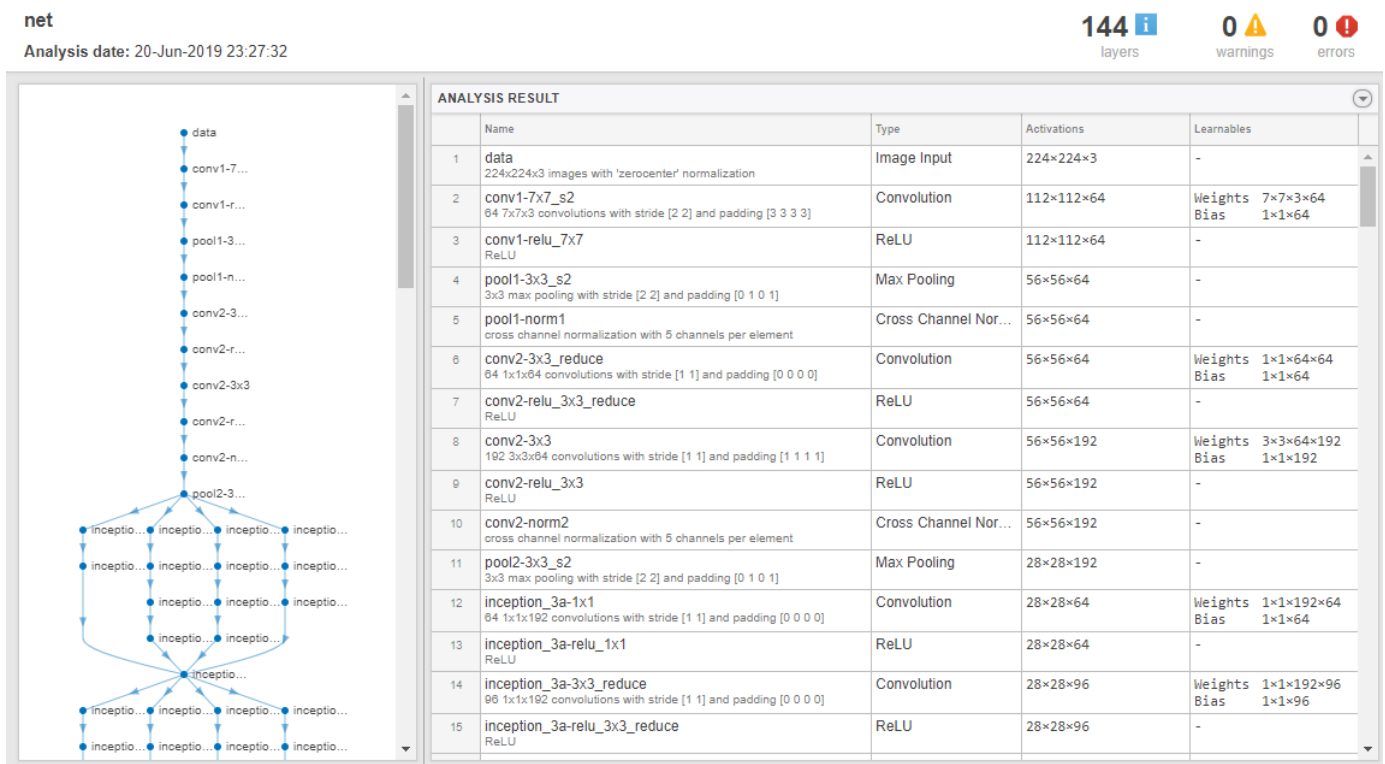
Load Pretrained Network

- 1 Load the pretrained GoogLeNet network. You can choose to load a different pretrained network for image classification. If you do not have the required support packages installed, the software provides a download link.

```
net = googlenet;
```

- 2 The object `net` contains the `DAGNetwork` object. Use the `analyzeNetwork` function to display an interactive visualization of the network architecture, to detect errors and issues in the network, and to display detailed information about the network layers. The layer information includes the sizes of layer activations and learnable parameters, the total number of learnable parameters, and the sizes of state parameters of recurrent layers.

```
analyzeNetwork(net);
```



- 3 The image that you want to classify must have the same size as the input size of the network. For GoogLeNet, the size of the `imageInputLayer` is 224-by-224-by-3. The `Classes` property of the output `classificationLayer` contains the names of the classes learned by the network. View 10 random class names out of the total of 1000.

```
classNames = net.Layers(end).Classes;
numClasses = numel(classNames);
disp(classNames(randperm(numClasses, 10)))
```

```
'speedboat'
'window screen'
'isopod'
'wooden spoon'
'lipstick'
'drake'
```



```
'hyena'
'dumbbell'
'strawberry'
'custard apple'
```

For more information, see “List of Deep Learning Layers” (Deep Learning Toolbox).

Create an Entry-Point Function

- 1 Write an entry-point function in MATLAB that:
 - a Uses the `coder.loadDeepLearningNetwork` function to load a deep learning model and to construct and set up a CNN class. For more information, see “Load Pretrained Networks for Code Generation” on page 4-66.
 - b Calls `predict` to predict the responses.

- 2 For example:

```
function out = googlenet_predict(in) %#codegen
persistent mynet;
if isempty(mynet)
    mynet = coder.loadDeepLearningNetwork('googlenet');
end
% pass in input
out = predict(mynet,in);
```

A persistent object `mynet` loads the `DAGNetwork` object. At the first call to the entry-point function, the persistent object is constructed and set up. On subsequent calls to the function, the same object is reused to call `predict` on inputs, avoiding reconstructing and reloading the network object.

Note Code generation requires the network to be loaded into a persistent object.

- 3 You can also use the `activations` method to network activations for a specific layer. For example, the following line of code returns the network activations for the layer specified in `layerIdx`.


```
out = activations(mynet,in,layerIdx,'OutputAs','Channels');
```
- 4 You can also use the `classify` method to predict class labels for the image data in `in` using the trained network, `mynet`.


```
[out,scores] = classify(mynet,in);
```

For LSTM networks, you can also use the `predictAndUpdateState` and `resetState` methods. For usage notes and limitations of these method, see the corresponding entry in the “Supported Functions” on page 1-6 table.

Code Generation by Using `codegen`

- 1 To configure build settings such as output file name, location, and type, you create `coder` configuration objects. To create the objects, use the `coder.gpuConfig` function. For example, when generating CUDA MEX using the `codegen` command, use `cfg = coder.gpuConfig('mex');`

Other available options are:

- a `cfg = coder.gpuConfig('lib');` to create a code generation configuration object for use with `codegen` when generating a CUDA C/C++ static library.
 - b `cfg = coder.gpuConfig('dll');` to create a code generation configuration object for use with `codegen` when generating a CUDA C/C++ dynamic library.
 - c `cfg = coder.gpuConfig('exe');` to create a code generation configuration object for use with `codegen` when generating a CUDA C/C++ executable.
- 2 To specify code generation parameters for cuDNN, set the `DeepLearningConfig` property to a `coder.CuDNNConfig` object that you create by using `coder.DeepLearningConfig`.

```
cfg = coder.gpuConfig('mex');  
cfg.TargetLang = 'C++';  
cfg.DeepLearningConfig = coder.DeepLearningConfig('cudnn');  
cfg.DeepLearningConfig.AutoTuning = true;  
cfg.DeepLearningConfig.DataType = 'fp32';
```

Specify the precision of the inference computations in supported layers by using the `DataType` property. When performing inference in 32-bit floats, use `'fp32'`. For 8-bit integer, use `'int8'`. Default value is `'fp32'`. INT8 precision requires a CUDA GPU with minimum compute capability of 6.1. Use the `ComputeCapability` property of the `GpuConfig` object to set the appropriate compute capability value.

Note Code generation for INT8 data type does not support multiple deep learning networks in the entry-point function.

When performing inference in INT8 precision using cuDNN version 8.1.0, issues in the NVIDIA library may cause significant degradation in performance.

- 3 Run the `codegen` command. The `codegen` command generates CUDA code from the `googlenet_predict.m` MATLAB entry-point function.

```
codegen -config cfg googlenet_predict -args {ones(224,224,3)} -report
```

- a The `-report` option instructs `codegen` to generate a code generation report that you can use to debug your MATLAB code.
- b The `-args` option instructs `codegen` to compile the file `googlenet_predict.m` by using the class, size, and complexity specified for the input `in`. The value `(224, 224, 3)` corresponds to input layer size of the GoogLeNet network.
- c The `-config` option instructs `codegen` to use the specified configuration object for code generation.

Note You can specify half-precision inputs for code generation. However, the code generator type casts the inputs to single-precision. The Deep Learning Toolbox uses single-precision, floating-point arithmetic for all computations in MATLAB.

The code generator uses column-major layout by default. To use row-major layout pass the `-rowmajor` option to the `codegen` command. Alternatively, configure your code for row-major layout by modifying the `cfg.RowMajor` parameter in the code generation configuration object.

- 4 When code generation is successful, you can view the resulting code generation report by clicking **View Report** in the MATLAB Command Window. The report is displayed in the Report Viewer window. If the code generator detects errors or warnings during code generation, the report describes the issues and provides links to the problematic MATLAB code. See “Code Generation Reports”.

Code generation successful: [View report](#)

Generated Code

The DAG network is generated as a C++ class containing an array of 78 layer classes. The code generator reduces the number of layers by using layer fusion optimization of convolutional and ReLU layers. A snippet of the class declaration from `googlenet_predict_types.h` file is shown.

`googlenet_predict_types.h` File

```
class b_googlenet_0
{
public:
    void presetup();
    void allocate();
    void postsetup();
    b_googlenet_0();
    void setup();
    void deallocate();
    void predict();
    void cleanup();
    real32_T *getLayerOutput(int32_T layerIndex, int32_T portIndex);
    ~b_googlenet_0();
    int32_T batchSize;
    int32_T numLayers;
    real32_T *getInputDataPointer();
    real32_T *getOutputDataPointer();
    MWCNNLayer *layers[78];
private:
    MWTargetNetworkImpl *targetImpl;
};
```

- The `setup()` method of the class sets up handles and allocates memory for each layer of the network object.
- The `predict()` method invokes prediction for each of the 78 layers in the network.
- The `DeepLearningNetwork.cu` file contains the definitions of the object functions for the `b_googlenet_0` class.

Binary files are exported for layers with parameters such as fully connected and convolution layers in the network. For instance, files `cnn_googlenet_conv*_w` and `cnn_googlenet_conv*_b` correspond to weights and bias parameters for the FusedConvReLU layers in the network. The code generator places these binary files in the `codegen` folder.

By default, the generated application looks for the weight files in the `codegen` folder. If you are relocating the generated application and weight files to a different location such as an embedded board, create an environment variable called `USER_DL_DATA_PATH`, whose value is the location of the relocated weight files. The generated application will then look for the weight files in this location.

Note On Windows systems, some antivirus software such as Bit Defender can incorrectly identify some weight files as infected and delete them. These cases are false positives and the files can be marked as safe in your antivirus program.

In the generated code file `googlenet_predict.cu`, the entry-point function `googlenet_predict()` constructs a static object of `b_googlenet_0` class type and invokes `setup` and `predict` on this network object.

googlenet_predict.cu File

```
/* Include files */
#include "googlenet_predict.h"
#include "DeepLearningNetwork.h"
#include "predict.h"
#include "rt_nonfinite.h"

/* Variable Definitions */
static b_googlenet_0 mynet;
static boolean_T mynet_not_empty;

/* Function Definitions */
void googlenet_predict(const real_T in[150528], real32_T out[1000])
{
    if (!mynet_not_empty) {
        DeepLearningNetwork_setup(&mynet);
        mynet_not_empty = true;
    }

    DeepLearningNetwork_predict(&mynet, in, out);
}

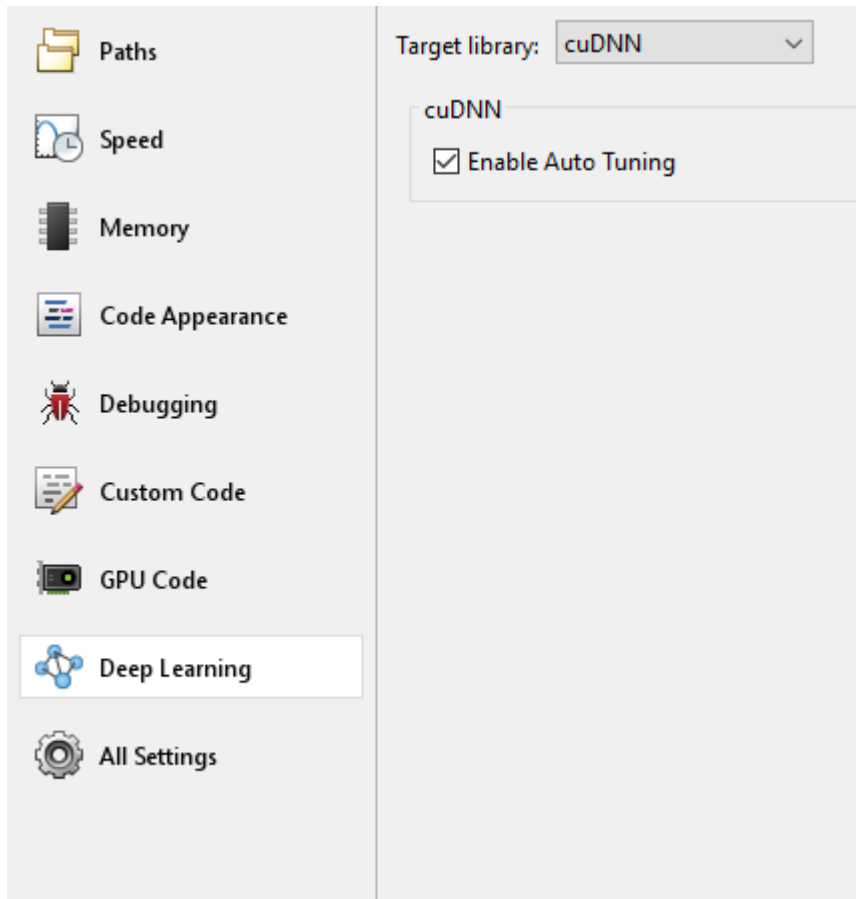
void googlenet_predict_init()
{
    mynet_not_empty = false;
}
```

Generate Code by Using the App

To specify the entry-point function and specifying input types, complete the procedure in the app. See “Code Generation by Using the GPU Coder App”.

In the **Generate Code** step:

- 1 Set the **Build** type to **MEX**.
- 2 Click **More Settings**. In the **Deep Learning** pane, set **Target library** to **cuDNN**.



- 3 Close the settings window. To generate CUDA code, click **Generate**.

Generated Makefile

For 'lib', 'dll', and 'exe' targets, the code generator creates the *_rtw.mk make file in the codegen folder. In this make file, the location of the generated code is specified by using the START_DIR variable found in the MACROS section. By default, this variable points to the path of the current working folder where the code is generated. If you plan to move the generated files and use the makefile to build, replace the generated value of START_DIR with the appropriate path location.

Run the Generated MEX

- 1 The image that you want to classify must have the same size as the input size of the network. Read the image that you want to classify and resize it to the input size of the network. This resizing slightly changes the aspect ratio of the image.

```
im = imread("peppers.png");
inputLayerSize = net.Layers(1).InputSize;
im = imresize(im,inputLayerSize(1:2));
```

- 2 Call GoogLeNet predict on the input image.

```
predict_scores = googlenet_predict_mex(im);
```

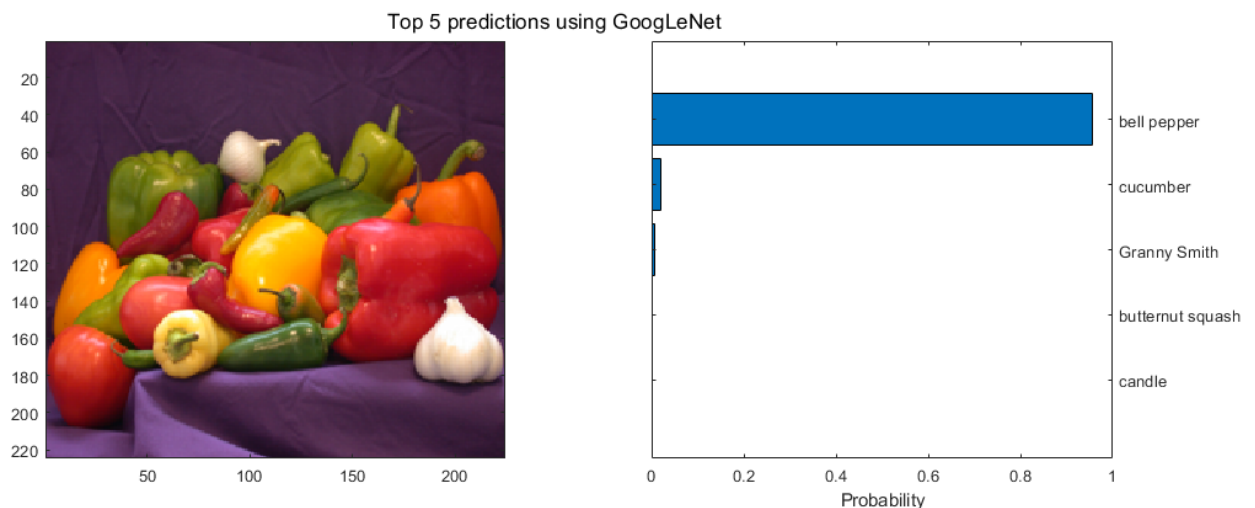
- 3 Display the top five predicted labels and their associated probabilities as a histogram. Because the network classifies images into so many object categories, and many categories are similar, it

is common to consider the top-five accuracy when evaluating networks. The network classifies the image as a bell pepper with a high probability.

```
[scores,indx] = sort(predict_scores, 'descend');
classNamesTop = classNames(indx(1:5));
```

```
h = figure;
h.Position(3) = 2*h.Position(3);
ax1 = subplot(1,2,1);
ax2 = subplot(1,2,2);
```

```
image(ax1,im);
barh(ax2,scores(5:-1:1))
xlabel(ax2,'Probability')
yticklabels(ax2,classNamesTop(5:-1:1))
ax2.YAxisLocation = 'right';
sgtitle('Top 5 predictions using GoogLeNet')
```



See Also

Functions

`coder.loadDeepLearningNetwork` | `codegen` | `coder.DeepLearningConfig`

Objects

`coder.gpuConfig` | `coder.CodeConfig` | `coder.EmbeddedCodeConfig` | `coder.CuDNNConfig`

See Also

More About

- “Supported Networks, Layers, and Classes” on page 4-6
- “Load Pretrained Networks for Code Generation” on page 4-66
- “Code Generation for Deep Learning Networks by Using TensorRT” on page 4-78
- “Deployment and Classification of Webcam Images on NVIDIA Jetson TX2 Platform” on page 5-44

- “Generated CNN Class Hierarchy” on page 4-65

Code Generation for Deep Learning Networks by Using TensorRT

With GPU Coder, you can generate optimized code for prediction of a variety of trained deep learning networks from Deep Learning Toolbox. The generated code implements the deep convolutional neural network (CNN) by using the architecture, the layers, and parameters that you specify in the input `SeriesNetwork` or `DAGNetwork` object. You can configure the code generator to take advantage of the NVIDIA TensorRT high performance inference library for NVIDIA GPUs. TensorRT provides improved latency, throughput, and memory efficiency by combining network layers and optimizing kernel selection. You can also configure the code generator to take advantage TensorRT's precision modes (FP32, FP16, or INT8) to further improve performance and reduce memory requirements. The generated code can be integrated into your project as source code, static or dynamic libraries, or executables that you can deploy to a variety of NVIDIA GPU platforms.

Note The TensorRT work flow is not supported on MATLAB Online™.

Generate code for convolutional networks by using one of the methods:

- The standard `codegen` function that generates CUDA code from a MATLAB entry-point function.
- The GPU Coder app that generates CUDA code from a MATLAB entry-point function.

Note In previous releases you could target the TensorRT library by using the `cnncodegen` function. From R2021a onwards, the `cnncodegen` function generates C++ code and make files for only the ARM Mali GPU processor.

Generate Code and Classify Images by Using GoogLeNet

In this example, you use GPU Coder to generate CUDA code for the pretrained `googlenet` deep convolutional neural network and classify an image. GoogLeNet has been trained on over a million images and can classify images into 1000 object categories (such as keyboard, coffee mug, pencil, and animals). The network has learned rich feature representations for a wide range of images. The network takes an image as input, and then outputs a label for the object in the image with the probabilities for each of the object categories. This example show you how to generate code for the pretrained network by using the `codegen` command and the GPU Coder app.

This example uses 32-bit floats (default value) as the precision for the tensor inputs. To learn more about using 8-bit integer precision for the tensors, see the “Deep Learning Prediction with NVIDIA TensorRT Library” on page 4-145 example.

Requirements

Required

This example generates CUDA MEX that has the following additional requirements.

- 1 Deep Learning Toolbox.
- 2 Deep Learning Toolbox Model for GoogLeNet Network support package.

- 3 GPU Coder Interface for Deep Learning Libraries support package.
- 4 CUDA enabled NVIDIA GPU and a compatible driver. For 8-bit integer precision, the CUDA GPU must have a compute capability of 6.1, 7.0 or higher. Half-precision requires a CUDA GPU with minimum compute capability of 5.3, 6.0, 6.2 or higher.

Optional

For non-MEX builds such as static, dynamic libraries, or executables, this example has the following additional requirements.

- 1 CUDA Toolkit, cuDNN, and TensorRT libraries. For information on the supported versions of the compilers and libraries, see “Installing Prerequisite Products”.
- 2 Environment variables for the compilers and libraries. For more information, see “Environment Variables”.

Load Pretrained Network

- 1 Load the pretrained GoogLeNet network. You can choose to load a different pretrained network for image classification. If you do not have the required support packages installed, the software provides a download link.

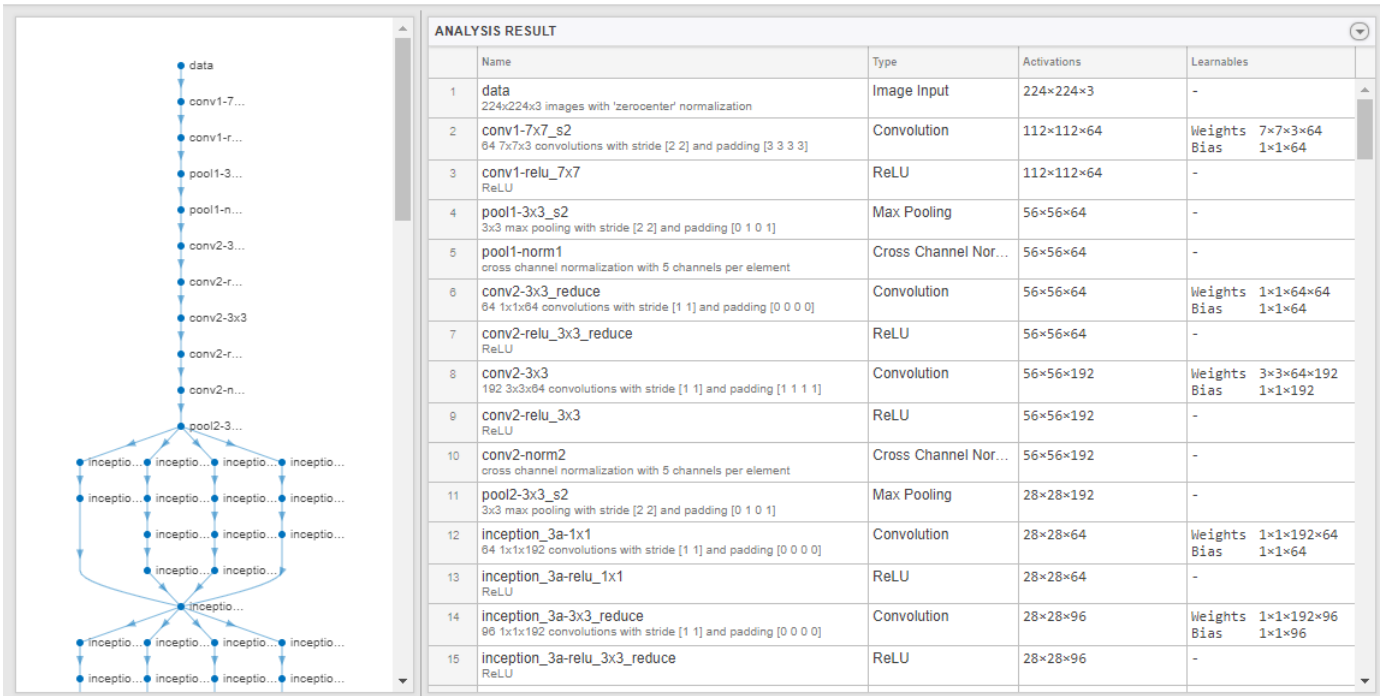
```
net = googlenet;
```

- 2 The object `net` contains the `DAGNetwork` object. Use the `analyzeNetwork` function to display an interactive visualization of the network architecture, to detect errors and issues in the network, and to display detailed information about the network layers. The layer information includes the sizes of layer activations and learnable parameters, the total number of learnable parameters, and the sizes of state parameters of recurrent layers.

```
analyzeNetwork(net);
```

net

Analysis date: 20-Jun-2019 23:27:32

144 
layers0 
warnings0 
errors

- 3 The image that you want to classify must have the same size as the input size of the network. For GoogLeNet, the size of the `imageInputLayer` is 224-by-224-by-3. The `Classes` property of the output `classificationLayer` contains the names of the classes learned by the network. View 10 random class names out of the total of 1000.

```
classNames = net.Layers(end).Classes;
numClasses = numel(classNames);
disp(classNames(randperm(numClasses,10)))
```

```
'speedboat'
>window screen'
'isopod'
'wooden spoon'
'lipstick'
'drake'
'hyena'
'dumbbell'
'strawberry'
'custard apple'
```

For more information, see “List of Deep Learning Layers” (Deep Learning Toolbox).

Create an Entry-Point Function

- 1 Write an entry-point function in MATLAB that:
 - a Uses the `coder.loadDeepLearningNetwork` function to load a deep learning model and to construct and set up a CNN class. For more information, see “Load Pretrained Networks for Code Generation” on page 4-66.

b Calls `predict` to predict the responses.

2 For example:

```
function out = googlenet_predict(in) %#codegen
persistent mynet;
if isempty(mynet)
    mynet = coder.loadDeepLearningNetwork('googlenet');
end
% pass in input
out = predict(mynet,in);
```

A persistent object `mynet` loads the `DAGNetwork` object. At the first call to the entry-point function, the persistent object is constructed and set up. On subsequent calls to the function, the same object is reused to call `predict` on inputs, avoiding reconstructing and reloading the network object.

Note Code generation requires the network to be loaded into a persistent object.

3 You can also use the `activations` method to network activations for a specific layer. For example, the following line of code returns the network activations for the layer specified in `layerIdx`.

```
out = activations(mynet,in,layerIdx,'OutputAs','Channels');
```

4 You can also use the `classify` method to predict class labels for the image data in `in` using the trained network, `mynet`.

```
[out,scores] = classify(mynet,in);
```

For LSTM networks, you can also use the `predictAndUpdateState` and `resetState` methods. For usage notes and limitations of these method, see the corresponding entry in the “Supported Functions” on page 1-6 table.

Code Generation by Using `codegen`

1 To configure build settings such as output file name, location, and type, you create `coder` configuration objects. To create the objects, use the `coder.gpuConfig` function. For example, when generating CUDA MEX by using the `codegen` command, use `cfg = coder.gpuConfig('mex');`

Other available options are:

- a** `cfg = coder.gpuConfig('lib');` to create a code generation configuration object for use with `codegen` when generating a CUDA C/C++ static library.
- b** `cfg = coder.gpuConfig('dll');` to create a code generation configuration object for use with `codegen` when generating a CUDA C/C++ dynamic library.
- c** `cfg = coder.gpuConfig('exe');` to create a code generation configuration object for use with `codegen` when generating a CUDA C/C++ executable.

2 To specify code generation parameters for TensorRT, set the `DeepLearningConfig` property to a `coder.TensorRTConfig` object that you create by using `coder.DeepLearningConfig`.

```
cfg = coder.gpuConfig('mex');
cfg.TargetLang = 'C++';
cfg.DeepLearningConfig = coder.DeepLearningConfig('tensorrt');
cfg.DeepLearningConfig.DataType = 'fp32';
```

Specify the precision of the inference computations in supported layers by using the `DataType` property. When performing inference in 32-bit floats, use `'fp32'`. For half-precision, use `'fp16'`. For 8-bit integer, use `'int8'`. Default value is `'fp32'`. INT8 precision requires a CUDA GPU with a compute capability of 6.1, 7.0, or higher. FP16 precision requires a CUDA GPU with a compute capability of 5.3, 6.0, 6.2, or higher. Use the `ComputeCapability` property of the `GpuConfig` object to set the appropriate compute capability value.

When you select the `'INT8'` option, TensorRT quantizes the floating-point data to `int8`. The calibration is performed with a reduced set of the calibration data. The calibration data must be present in the image data location specified by `DataPath`. Preprocessing of the images must be performed before calibration and the preprocessing steps must be included in the entry-point file before code generation.

Code generation by using the NVIDIA TensorRT Library with inference computation in 8-bit integer precision supports these additional networks:

- Object detector networks such as YOLOv2 and SSD.
- Regression and semantic segmentation networks. For semantic segmentation networks, the recalibration images must be in a format supported by the `imread` function.

See the “Deep Learning Prediction with NVIDIA TensorRT Library” on page 4-145 example for 8-bit integer prediction for a logo classification network by using TensorRT.

- 3 Run the `codegen` command. The `codegen` command generates CUDA code from the `googlenet_predict.m` MATLAB entry-point function.

```
codegen -config cfg googlenet_predict -args {ones(224,224,3)} -report
```

- a The `-report` option instructs `codegen` to generate a code generation report that you can use to debug your MATLAB code.
- b The `-args` option instructs `codegen` to compile the file `googlenet_predict.m` by using the class, size, and complexity specified for the input `in`. The value `(224, 224, 3)` corresponds to the input layer size of the GoogLeNet network.
- c The `-config` option instructs `codegen` to use the specified configuration object for code generation.

Note You can specify half-precision inputs for code generation. However, the code generator type casts the inputs to single-precision. The Deep Learning Toolbox uses single-precision, floating-point arithmetic for all computations in MATLAB. During code generation, you can enable inference with half-precision (16-bit floating-point) inputs by specifying the `DataType` property of `coder.TensorRTConfig` as `'fp16'`.

The code generator uses column-major layout by default. To use row-major layout pass the `-rowmajor` option to the `codegen` command. Alternatively, configure your code for row-major layout by modifying the `cfg.RowMajor` parameter in the code generation configuration object.

- 4 When code generation is successful, you can view the resulting code generation report by clicking **View Report** in the MATLAB Command Window. The report is displayed in the Report Viewer window. If the code generator detects errors or warnings during code generation, the report describes the issues and provides links to the problematic MATLAB code. See “Code Generation Reports”.

Code generation successful: View report

Generated Code

The DAG network is generated as a C++ class containing an array of 144 layer classes. A snippet of the class declaration from `googlenet_predict_types.h` file is shown.

`googlenet_predict_types.h` File

```
class b_googlenet_0
{
public:
    void presetup();
    void allocate();
    void postsetup();
    b_googlenet_0();
    void setup();
    void deallocate();
    void predict();
    void cleanup();
    real32_T *getLayerOutput(int32_T layerIndex, int32_T portIndex);
    ~b_googlenet_0();
    int32_T batchSize;
    int32_T numLayers;
    real32_T *getInputDataPointer();
    real32_T *getOutputDataPointer();
    MWCNNLayer *layers[144];
private:
    MWTargetNetworkImpl *targetImpl;
};
```

- The `setup()` method of the class sets up handles and allocates memory for each layer of the network object.
- The `predict()` method invokes prediction for each of the 144 layers in the network.
- The `DeepLearningNetwork.cu` file contains the definitions of the object functions for the `b_googlenet_0` class.

Binary files are exported for layers with parameters such as fully connected and convolution layers in the network. For instance, files `cnn_googlenet_conv*_w` and `cnn_googlenet_conv*_b` correspond to weights and bias parameters for the convolutional layers in the network. The code generator places these binary files in the `codegen` folder.

By default, the generated application looks for the weight files in the `codegen` folder. If you are relocating the generated application and weight files to a different location such as an embedded board, create an environment variable called `USER_DL_DATA_PATH`, whose value is the location of the relocated weight files. The generated application will then look for the weight files in this location.

Note On Windows systems, some antivirus software such as Bit Defender can incorrectly identify some weight files as infected and delete them. These cases are false positives and the files can be marked as safe in your antivirus program.

In the generated code file `googlenet_predict.cu`, the entry-point function `googlenet_predict()` constructs a static object of `b_googlenet_0` class type and invokes `setup` and `predict` on this network object.

googlenet_predict.cu File

```
/* Include files */
#include "googlenet_predict.h"
#include "DeepLearningNetwork.h"
#include "predict.h"
#include "rt_nonfinite.h"

/* Variable Definitions */
static b_googlenet_0 mynet;
static boolean_T mynet_not_empty;

/* Function Definitions */
void googlenet_predict(const real_T in[150528], real32_T out[1000])
{
    if (!mynet_not_empty) {
        DeepLearningNetwork_setup(&mynet);
        mynet_not_empty = true;
    }

    DeepLearningNetwork_predict(&mynet, in, out);
}

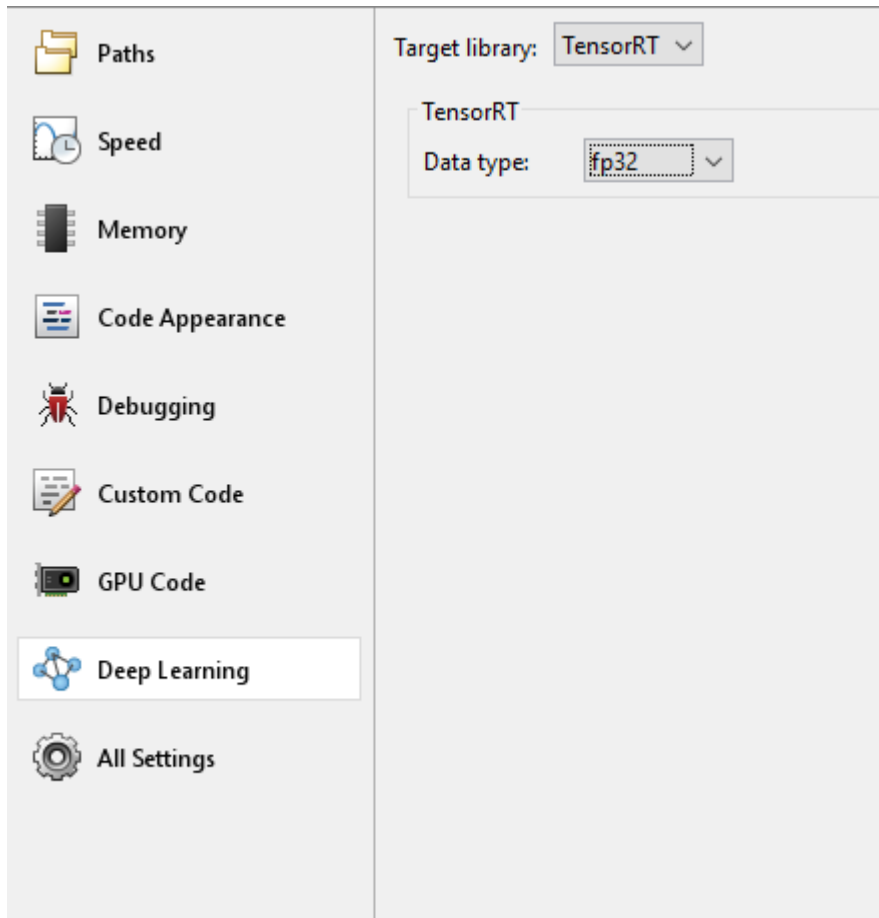
void googlenet_predict_init()
{
    mynet_not_empty = false;
}
```

Generate Code by Using the App

To specify the entry-point function and specifying input types, complete the procedure in the app. See “Code Generation by Using the GPU Coder App”.

In the **Generate Code** step:

- 1 Set the **Build** type to MEX.
- 2 Click **More Settings**. In the **Deep Learning** pane, set **Target library** to **TensorRT**.



- 3 Close the settings window. To generate CUDA code, click **Generate**.

Generated Makefile

For 'lib', 'dll', and 'exe' targets, the code generator creates the *_rtw.mk make file in the codegen folder. In this make file, the location of the generated code is specified by using the START_DIR variable found in the MACROS section. By default, this variable points to the path of the current working folder where the code is generated. If you plan to move the generated files and use the makefile to build, replace the generated value of START_DIR with the appropriate path location.

Run the Generated MEX

- 1 The image that you want to classify must have the same size as the input size of the network. Read the image that you want to classify and resize it to the input size of the network. This resizing slightly changes the aspect ratio of the image.

```
im = imread("peppers.png");
inputLayerSize = net.Layers(1).InputSize;
im = imresize(im,inputLayerSize(1:2));
```

- 2 Call GoogLeNet predict on the input image.

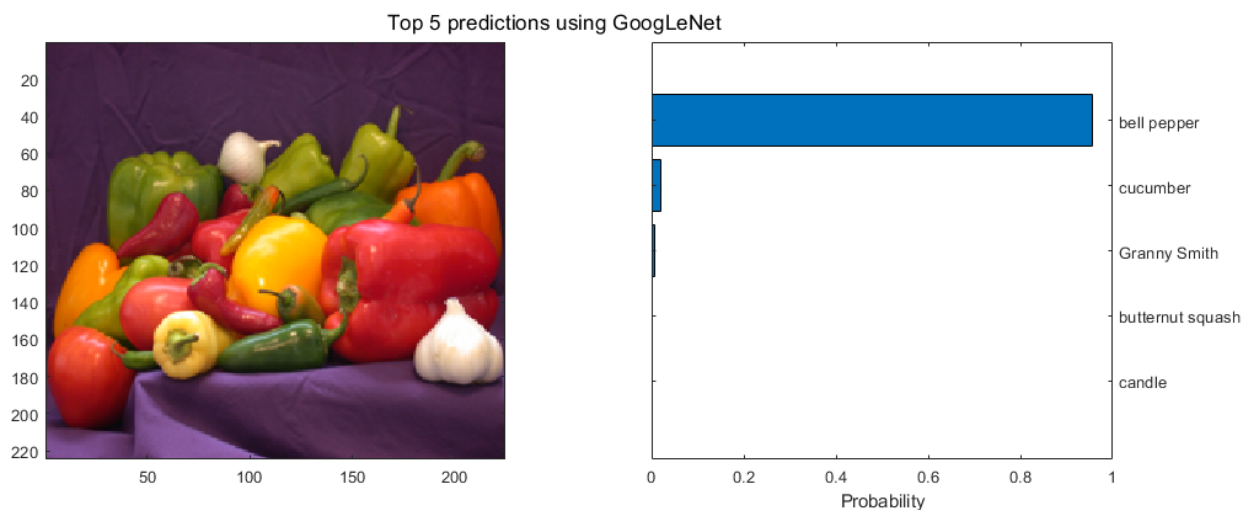
```
predict_scores = googlenet_predict_mex(im);
```

- 3 Display the top five predicted labels and their associated probabilities as a histogram. Because the network classifies images into so many object categories, and many categories are similar, it is common to consider the top-five accuracy when evaluating networks. The network classifies the image as a bell pepper with a high probability.

```
[scores,indx] = sort(predict_scores, 'descend');
classNamesTop = classNames(indx(1:5));
```

```
h = figure;
h.Position(3) = 2*h.Position(3);
ax1 = subplot(1,2,1);
ax2 = subplot(1,2,2);
```

```
image(ax1,im);
barh(ax2,scores(5:-1:1))
xlabel(ax2,'Probability')
yticklabels(ax2,classNamesTop(5:-1:1))
ax2.YAxisLocation = 'right';
sgtitle('Top 5 predictions using GoogLeNet')
```



See Also

Functions

`coder.loadDeepLearningNetwork` | `codegen` | `coder.DeepLearningConfig`

Objects

`coder.gpuConfig` | `coder.CodeConfig` | `coder.EmbeddedCodeConfig` | `coder.TensorRTConfig`

See Also

More About

- “Supported Networks, Layers, and Classes” on page 4-6
- “Load Pretrained Networks for Code Generation” on page 4-66

- “Code Generation for Deep Learning Networks by Using cuDNN” on page 4-69
- “Deep Learning Prediction with NVIDIA TensorRT Library” on page 4-145
- “Code Generation for Deep Learning Networks” on page 4-117
- “Code Generation for Object Detection by Using YOLO v2” on page 4-180
- “Deployment and Classification of Webcam Images on NVIDIA Jetson TX2 Platform” on page 5-44

Code Generation for Deep Learning Networks Targeting ARM Mali GPUs

With GPU Coder, you can generate optimized code for prediction of a variety of trained deep learning networks from Deep Learning Toolbox. The generated code implements the deep convolutional neural network (CNN) by using the architecture, the layers, and parameters that you specify in the input `SeriesNetwork` or `DAGNetwork` object. The code generator takes advantage of the ARM Compute Library for computer vision and machine learning. For performing deep learning on ARM Mali GPU targets, you generate code on the host development computer. Then, to build and run the executable program move the generated code to the ARM target platform. For example, HiKey960 is one of the target platforms that can execute the generated code.

Requirements

- 1 Deep Learning Toolbox.
- 2 Deep Learning Toolbox Model for MobileNet-v2 Network support package.
- 3 GPU Coder Interface for Deep Learning Libraries support package. To install the support packages, select the support package from the MATLAB **Add-Ons** menu.
- 4 ARM Compute Library for computer vision and machine learning must be installed on the target hardware. For information on the supported versions of the compilers and libraries, see “Installing Prerequisite Products”.
- 5 Environment variables for the compilers and libraries. For more information, see “Environment Variables”.

Load Pretrained Network

- 1 Load the pretrained MobileNet-v2 network. You can choose to load a different pretrained network for image classification. If you do not have the required support packages installed, the software provides a download link.

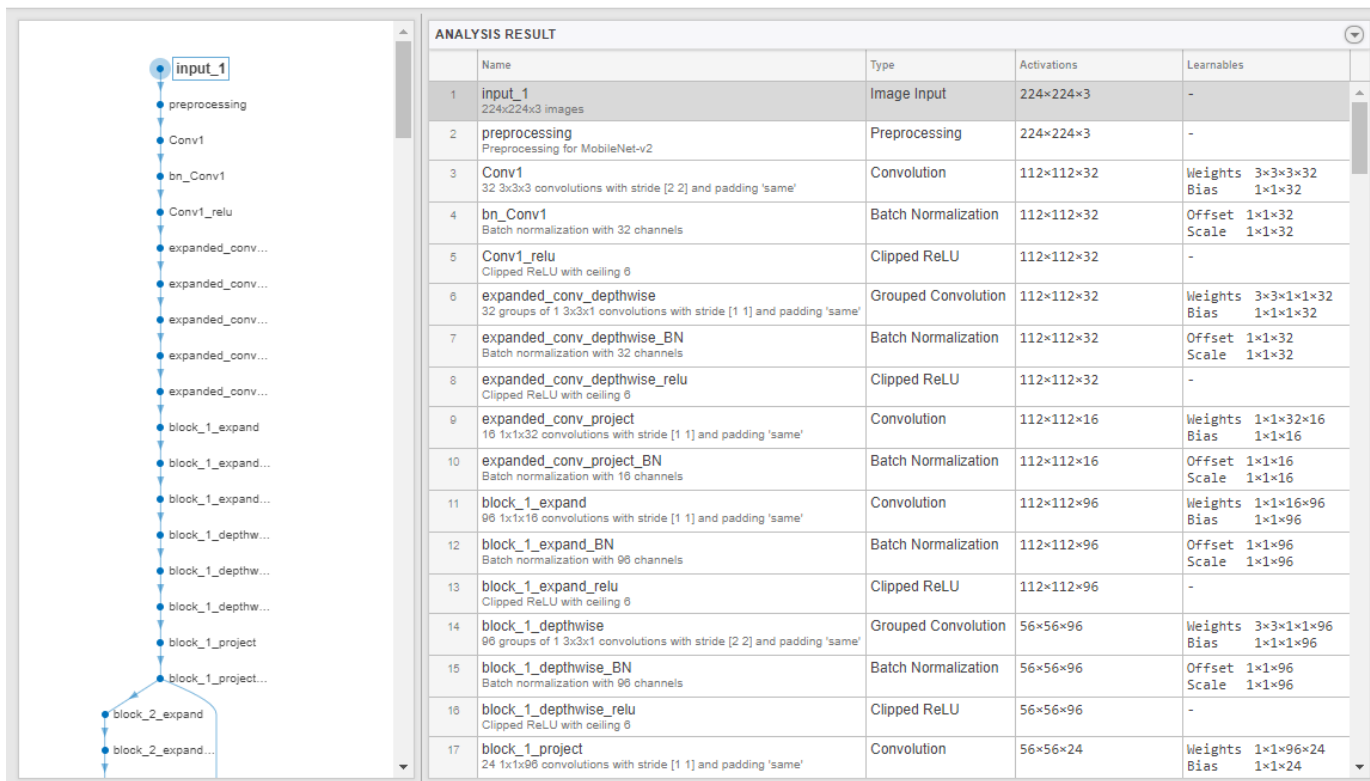
```
net = mobilenetv2;
```

- 2 The object `net` contains the `DAGNetwork` object. Use the `analyzeNetwork` function to display an interactive visualization of the network architecture, to detect errors and issues in the network, and to display detailed information about the network layers. The layer information includes the sizes of layer activations and learnable parameters, the total number of learnable parameters, and the sizes of state parameters of recurrent layers.

```
analyzeNetwork(net);
```

net

Analysis date: 27-Jun-2019 11:35:23

155 
layers0 
warnings0 
errors

- 3 The image that you want to classify must have the same size as the input size of the network. For GoogLeNet, the size of the `imageInputLayer` is 224-by-224-by-3. The `Classes` property of the output `classificationLayer` contains the names of the classes learned by the network. View 10 random class names out of the total of 1000.

```
classNames = net.Layers(end).Classes;
numClasses = numel(classNames);
disp(classNames(randperm(numClasses,10)))
```

```
cock
apiary
soap dispenser
titi
car wheel
guenon
muzzle
agaric
buckeye
megalith
```

For more information, see “List of Deep Learning Layers” (Deep Learning Toolbox).

Code Generation by Using `cnncodegen`

To generate code with the ARM Compute Library, use the `targetlib` option of the `cnncodegen` command. The `cnncodegen` command generates C++ code for the `SeriesNetwork` or `DAGNetwork` network object.

- 1 Call `cnncodegen` with `'targetlib'` specified as `'arm-compute-mali'`. For example:

```
net = googlenet;
cnncodegen(net, 'targetlib', 'arm-compute-mali', 'batchsize', 1);
```

For `'arm-compute-mali'`, the value of `batchsize` must be 1.

The `'targetparams'` name-value pair arguments that enable you to specify Library-specific parameters for the ARM Compute Library is not applicable when targeting ARM Mali GPUs.

- 2 The `cnncodegen` command generates code, a makefile, `cnnbuild_rtw.mk`, and other supporting files to build the generated code on the target hardware. The command places all the generated files in the `codegen` folder.
- 3 Write a C++ main function that calls `predict`. For an example main file that interfaces with the generated code, see “Deep Learning Prediction on ARM Mali GPU” on page 4-190
- 4 Move the generated `codegen` folder and other files from the host development computer to the ARM hardware by using your preferred Secure File Copy (SCP) and Secure Shell (SSH) client. Build the executable program on the target.

Generated Code

The DAG network is generated as a C++ class (`CnnMain`) containing an array of 103 layer classes. The code generator reduces the number of layers is by layer fusion optimization of convolutional and batch normalization layers. A snippet of the class declaration from `cnn_exec.hpp` file is shown.

`cnn_exec.hpp` File

```
class CnnMain
{
public:
    int32_T numLayers;
private:
    MWTensorBase *inputTensors[1];
    MWTensorBase *outputTensors[1];
public:
    MWCNNLayer *layers[103];
private:
    MWTargetNetworkImpl *targetImpl;
    void allocate();
    void postsetup();
public:
    CnnMain();
private:
    void deallocate();
public:
    void setup();
    void predict();
    void cleanup();
    real32_T *getLayerOutput(int32_T layerIndex, int32_T portIndex);
    real32_T *getInputDataPointer(int32_T index);
    real32_T *getInputDataPointer();
    real32_T *getOutputDataPointer(int32_T index);
    real32_T *getOutputDataPointer();
    int32_T getBatchSize();
    ~CnnMain();
};
```

- The `setup()` method of the class sets up handles and allocates memory for each layer of the network object.
- The `predict()` method invokes prediction for each of the 103 layers in the network.
- The `cnn_exec.cpp` file contains the definitions of the object functions for the `CnnMain` class.

Binary files are exported for layers with parameters such as fully connected and convolution layers in the network. For instance, files `cnn_CnnMain_Conv*_w` and `cnn_CnnMain_Conv*_b` correspond to weights and bias parameters for the convolutional layers in the network. The code generator places these binary files in the `codegen` folder. The code generator builds the library file `cnnbuild` and places all the generated files in the `codegen` folder.

Limitations

- Code generation for the ARM Mali GPU is not supported for a 2-D grouped convolution layer that has the `NumGroups` property set as `'channel-wise'` or a value greater than two.

See Also

Functions

`coder.getDeepLearningLayers` | `cnncodegen` | `coder.DeepLearningConfig`

More About

- “Supported Networks, Layers, and Classes” on page 4-6
- “Load Pretrained Networks for Code Generation” on page 4-66
- “Code Generation for Deep Learning Networks by Using cuDNN” on page 4-69
- “Code Generation for Deep Learning Networks by Using TensorRT” on page 4-78
- “Deep Learning Prediction on ARM Mali GPU” on page 4-190

Update Network Parameters After Code Generation

In this section...

“Create an Entry-Point Function” on page 4-92
 “Create a Network” on page 4-92
 “Code Generation by Using codegen” on page 4-93
 “Run the Generated MEX” on page 4-93
 “Update Network with Different Learnable Parameters” on page 4-94
 “Run the Generated MEX with Updated Learnables” on page 4-94
 “Limitations” on page 4-95

This example shows how to update learnable and state parameters of deep learning networks without regenerating code for the network. You can update the network parameters for `SeriesNetwork`, `DAGNetwork` and `dlnetwork`.

Parameter update supports MEX and standalone code generation for the NVIDIA CUDA deep neural network library (cuDNN) and the NVIDIA TensorRT high performance inference libraries.

Create an Entry-Point Function

- 1 Write an entry-point function in MATLAB that:
 - a Uses the `coder.loadDeepLearningNetwork` function to load a deep learning model. For more information, see “Load Pretrained Networks for Code Generation” on page 4-66.
 - b Calls `predict` to predict the responses.
- 2 For example:

```
function out = mLayer(in, matFile)
myNet = coder.loadDeepLearningNetwork(coder.const(matFile));
out = myNet.predict(in);
```

Create a Network

The network used in this example requires input images of size 4-by-5-by-3. Create sample network inputs of the same size format as the network inputs.

```
inputSize = [4 5 3];
im = dldarray(rand(inputSize, 'single'), 'SSCB');
```

Define the network architecture.

```
outSize = 6;
layers = [
    imageInputLayer(inputSize, 'Name', 'input', 'Normalization', 'none')
    convolution2dLayer([3 3], 5, 'Name', 'conv-1')
    batchNormalizationLayer('Name', 'batchNorm')
    reluLayer('Name', 'relu1')
    transposedConv2dLayer([2 2], 5, 'Name', 'transconv')
    convolution2dLayer([2 2], 5, 'Name', 'conv2')
    reluLayer('Name', 'relu2')
```

```
fullyConnectedLayer(outSize, 'Name', 'fc3')
];
```

Create an initialized `dlnetwork` object from the layer graph.

```
rng(0);
dlnet1 = dlnetwork(layers);
save('trainedNet.mat', 'dlnet1');
```

Code Generation by Using codegen

- 1 To configure build settings such as output file name, location, and type, you create coder configuration objects. To create the objects, use the `coder.gpuConfig` function.
- 2 To specify code generation parameters for cuDNN, set the `DeepLearningConfig` property to a `coder.CuDNNConfig` object that you create by using `coder.DeepLearningConfig`.

```
cfg = coder.gpuConfig('mex');
cfg.TargetLang = 'C++';
cfg.DeepLearningConfig = coder.DeepLearningConfig('TargetLibrary', 'cudnn')
cfg.DeepLearningConfig.AutoTuning = true;
cfg.DeepLearningConfig.DataType = 'fp32';
```

- 3 Specify the inputs.

```
cnnMatFile = fullfile(pwd, 'trainedNet.mat');
inputArgs = {im, coder.Constant(cnnMatFile)};
```

- 4 Run the `codegen` command. The `codegen` command generates CUDA code from the `mLayers.m` MATLAB entry-point function.

```
codegen -config cfg mLayer -args inputArgs -report
```

Run the Generated MEX

Call `predict` on the input image and compare the results with MATLAB.

```
out = mLayer_mex(im,cnnMatFile)
out_MATLAB = mLayer(im,cnnMatFile)
```

```
out1 =
```

```
6(C) x 1(B) single dlarray
```

```
-0.0064
-0.1422
-0.0897
0.2223
0.0329
0.0365
```

```
out_MATLAB =
```

```
6(C) x 1(B) single dlarray
```

```
-0.0064
-0.1422
-0.0897
0.2223
```

```
0.0329
0.0365
```

Update Network with Different Learnable Parameters

Re-initialize `dlnetwork` to update learnables to different values.

```
rng(10);
dlnet2 = dlnetwork(layers);
save('trainedNet.mat', 'dlnet2');
```

Use the `coder.regenerateDeepLearningParameters` function to regenerate the bias files based on the new learnables and states of the network.

The first input to the `coder.regenerateDeepLearningParameters` function is a `SeriesNetwork`, `DAGNetwork` or `dlnetwork` object. The second argument is the path to the network parameter information file emitted during code generation. You can optionally specify the `NetworkName=MYNET` name-value pair to specify the name of the C++ class for the network in the generated code.

```
codegenDir = fullfile(pwd, 'codegen/mex/mLayer');
networkFileNames = (coder.regenerateDeepLearningParameters(dlnet2, codegenDir))'
```

The `coder.regenerateDeepLearningParameters` function returns a cell-array of files containing network learnables and states.

```
networkFileNames =
    8x1 cell array

    {'cnn_trainedNet0_0_conv-1_b.bin' }
    {'cnn_trainedNet0_0_conv-1_w.bin' }
    {'cnn_trainedNet0_0_conv2_b.bin' }
    {'cnn_trainedNet0_0_conv2_w.bin' }
    {'cnn_trainedNet0_0_fc3_b.bin' }
    {'cnn_trainedNet0_0_fc3_w.bin' }
    {'cnn_trainedNet0_0_transconv_b.bin'}
    {'cnn_trainedNet0_0_transconv_w.bin'}
```

Note For MEX workflows, when the generated MEX and the associated `codegen` folder is moved from one location to another, `coder.regenerateDeepLearningParameters` cannot regenerate files containing network learnables and states parameters in the new location. Set the `'OverrideParameterFiles'` parameter of `coder.regenerateDeepLearningParameters` to true to allow the `coder.regenerateDeepLearningParameters` function to regenerate files containing network learnables and states parameters in the original `codegen` location.

For standalone workflows, `coder.regenerateDeepLearningParameters` can regenerate files containing network learnables and states parameters in the new location

Run the Generated MEX with Updated Learnables

Call `predict` on the input image and compare the results with MATLAB.

```
clear mLayer_mex;
outNew = mLayer_mex(im,cnnMatFile)
outNew_MATLAB = mLayer(im,cnnMatFile)
```



```
outNew =
    6(C) x 1(B) single dlarray
    0.1408
   -0.0080
    0.0342
   -0.0065
    0.1843
    0.0799
```

```
outNew_MATLAB =
    6(C) x 1(B) single dlarray
    0.1408
   -0.0080
    0.0342
   -0.0065
    0.1843
    0.0799
```

Limitations

Only the network learnables and states can be updated by using the `coder.regenerateDeepLearningParameters` function. For modifications that the code generator does not support, an error message is thrown. For example, using `coder.regenerateDeepLearningParameters` after changing the scale factor of a leaky ReLU layer throws the following error message as scale factor is not a network learnable.

```
Network architecture has been modified since the last code generation. Unable
to accommodate the provided network in the generated code. Regenerate code
for the provided network to reflect changes in the network. For more
information, see Limitations to Regenerating Network Parameters After Code Generation.
```

See Also

Functions

`codegen` | `coder.loadDeepLearningNetwork` | `coder.regenerateDeepLearningParameters`

Objects

`SeriesNetwork` | `DAGNetwork` | `dlarray` | `dlnetwork`

More About

- “Supported Networks, Layers, and Classes” on page 4-6
- “Load Pretrained Networks for Code Generation” on page 4-66
- “Code Generation for Deep Learning Networks by Using cuDNN” on page 4-69
- “Code Generation for Deep Learning Networks by Using TensorRT” on page 4-78

Data Layout Considerations in Deep Learning

When you build an application that uses the generated CUDA C++ code, you must provide a CUDA C++ main function that calls the generated code. By default, for code generation of source code, static libraries, dynamic libraries, and executables by using the `codegen` command, GPU Coder generates example CUDA C++ main files (`main.cu` source file and `main.h` header file in the `examples` subfolder of the build folder). This example main file is a template that helps you incorporate generated CUDA code into your application. The example main function declares and initializes data, including dynamically allocated data. It calls entry-point functions but does not use values that the entry point functions return.

When generating code for deep convolutional neural networks (CNN), the code generator takes advantage of NVIDIA cuDNN, TensorRT for NVIDIA GPUs or the ARM Compute Library for the ARM Mali GPUs. These libraries have specific data layout requirements for the input tensor holding images, video, and any other data. When authoring custom main functions for building an application, you must create input buffers that provide data to the generated entry-point functions in the format expected by these libraries.

Data Layout Format for CNN

For deep convolutional neural networks (CNN), a 4-D tensor descriptor is used to define the format for batches of 2-D images with the following letters:

- N - the batch size
- C - the number of feature maps (number of channels)
- H - the height
- W - the width

The most commonly used 4-D tensor formats is shown, where the letters are sorted in decreasing order of the strides.

- NCHW
- NHWC
- CHWN

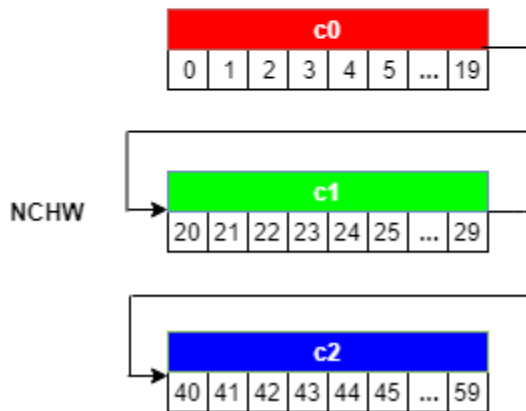
Of these, GPU Coder uses the NCHW format (column-major layout by default). To use row-major layout pass the `-rowmajor` option to the `codegen` command. Alternatively, configure your code for row-major layout by modifying the `cfg.RowMajor` parameter in the code generation configuration object.

For example, consider a batch of images with the following dimensions: N=1, C=3, H=5, W=4. If the image pixel elements are represented by a sequence of integers, the input images can be pictorially represented as follows.

C = 0			
0	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15
16	17	18	19

C = 1			
20	21	22	23
24	25	26	27
28	29	30	31
32	33	34	35
36	37	38	39

C = 2			
40	41	42	43
44	45	46	47
48	49	50	51
52	53	54	55
56	57	58	59



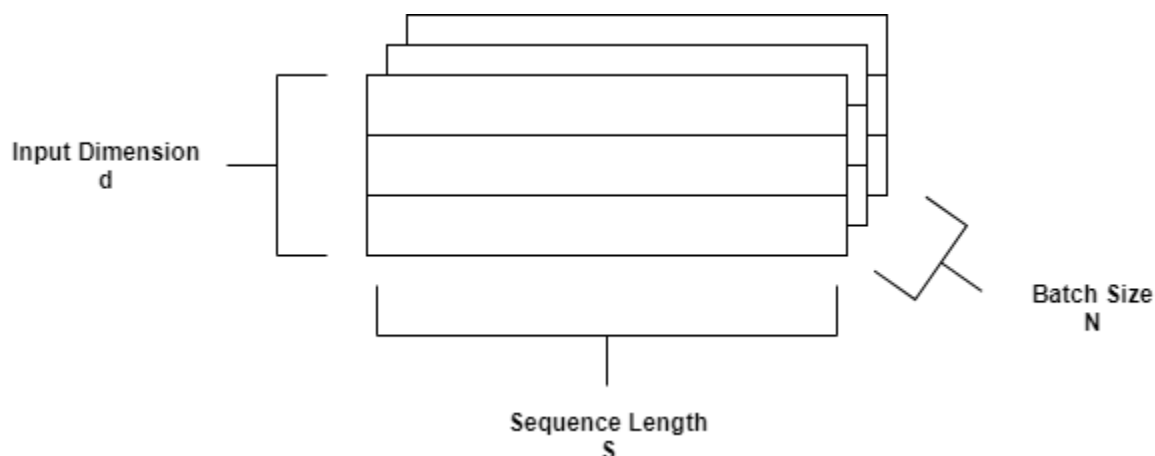
When creating the input buffer in the main function, the 4-D image is laid out in the memory in the NCHW format as:

- 1 Beginning with the first channel ($C=0$), the elements are arranged contiguously in row-major order.
- 2 Continue with second and subsequent channels until the elements of all the channels are laid out.
- 3 Proceed to the next batch (if $N > 1$).

Data Layout Format for LSTM

A long short-term memory (LSTM) network is a type of recurrent neural network (RNN) that can learn long-term dependencies between time steps of sequence data. For LSTM, the data layout format can be described with the following letters:

- N - the batch size
- S - the sequence length (number of time steps)
- d - the number of units in one input sequence



For LSTM, GPU Coder uses the SNd format by default.

See Also

Functions

`coder.getDeepLearningLayers` | `codegen` | `coder.DeepLearningConfig`

Objects

`coder.gpuConfig` | `coder.CodeConfig` | `coder.EmbeddedCodeConfig` | `coder.gpuEnvConfig` | `coder.CuDNNConfig` | `coder.TensorRTConfig`

More About

- “Supported Networks, Layers, and Classes” on page 4-6
- “Load Pretrained Networks for Code Generation” on page 4-66
- “Code Generation for Deep Learning Networks by Using cuDNN” on page 4-69
- “Code Generation for Deep Learning Networks by Using TensorRT” on page 4-78
- “Code Generation for Deep Learning Networks Targeting ARM Mali GPUs” on page 4-88
- “Lane Detection Optimized with GPU Coder” on page 4-124
- “Deployment and Classification of Webcam Images on NVIDIA Jetson TX2 Platform” on page 5-44

Quantization of Deep Neural Networks

In digital hardware, numbers are stored in binary words. A binary word is a fixed-length sequence of bits (1's and 0's). The data type defines how hardware components or software functions interpret this sequence of 1's and 0's. Numbers are represented as either scaled integer (usually referred to as fixed-point) or floating-point data types.

Most pretrained neural networks and neural networks trained using Deep Learning Toolbox use single-precision floating point data types. Even small trained neural networks require a considerable amount of memory, and require hardware that can perform floating-point arithmetic. These restrictions can inhibit deployment of deep learning capabilities to low-power microcontrollers and FPGAs.

Using the Deep Learning Toolbox Model Quantization Library support package, you can quantize a network to use 8-bit scaled integer data types.

To learn about the products required to quantize and deploy the deep learning network to a GPU, FPGA, or CPU environment, see “Quantization Workflow Prerequisites” (Deep Learning Toolbox).

Precision and Range

Scaled 8-bit integer data types have limited precision and range when compared to single-precision floating point data types. There are several numerical considerations when casting a number from a larger floating-point data type to a smaller data type of fixed length.

- Precision loss: Precision loss is a rounding error. When precision loss occurs, the value is rounded to the nearest number that is representable by the data type. In the case of a tie it rounds:
 - Positive numbers to the closest representable value in the direction of positive infinity.
 - Negative numbers to the closest representable value in the direction of negative infinity.

In MATLAB you can perform this type of rounding using the `round` function.

- Underflow: Underflow is a type of precision loss. Underflows occur when the value is smaller than the smallest value representable by the data type. When this occurs, the value saturates to zero.
- Overflow: When a value is larger than the largest value that a data type can represent, an overflow occurs. When an overflow occurs, the value saturates to the largest value representable by the data type.

Histograms of Dynamic Ranges

Use the **Deep Network Quantizer** app to collect and visualize the dynamic ranges of the weights and biases of the convolution layers and fully connected layers of a network, and the activations of all layers in the network. The app assigns a scaled 8-bit integer data type for the weights, biases, and activations of the convolution layers of the network. The app displays a histogram of the dynamic range for each of these parameters. The following steps describe how these histograms are produced.

- 1 Consider the following values logged for a parameter while exercising a network.

Original Values	Power of 2 Bins														8 Bit Binary Rep	Quantized Value			
	Sign Bit	2^8	2^5	2^4	2^3	2^2	2^1	2^0	2^{-1}	2^{-2}	2^{-3}	2^{-4}	2^{-5}	2^{-6}			2^{-7}	2^{-8}	
0.03125																			
-0.250																			
0.250																			
0.500																			
1.000																			
2.100																			
-2.125																			
8.250																			
16.250																			

2 Find the ideal binary representation of each logged value of the parameter.

The most significant bit (MSB) is the left-most bit of the binary word. This bit contributes most to the value of the number. The MSB for each value is highlighted in yellow.

Original Values	Sign Bit	Power of 2 Bins														8 Bit Binary Rep	Quantized Value		
		2 ⁶	2 ⁵	2 ⁴	2 ³	2 ²	2 ¹	2 ⁰	2 ⁻¹	2 ⁻²	2 ⁻³	2 ⁻⁴	2 ⁻⁵	2 ⁻⁶	2 ⁻⁷			2 ⁻⁸	
0.03125														1	0	0	0		
-0.250	✓									1	0	0	0	0	0	0	0		
0.250										1	0	0	0	0	0	0	0		
0.500									1	0	0	0	0	0	0	0	0		
1.000								1	0	0	0	0	0	0	0	0	0		
2.100							1	0	0	0	0	1	1	0	0	0	1		
-2.125	✓						1	0	0	0	1	0	0	0	0	0	0		
8.250					1	0	0	0	0	1	0	0	0	0	0	0	0		
16.250				1	0	0	0	0	0	1	0	0	0	0	0	0	0		

3 By aligning the binary words, you can see the distribution of bits used by the logged values of a parameter. The sum of MSB's in each column, highlighted in green, give an aggregate view of the logged values.

Original Values	Sign Bit	Power of 2 Bins														8 Bit Binary Rep	Quantized Value		
		2 ⁶	2 ⁵	2 ⁴	2 ³	2 ²	2 ¹	2 ⁰	2 ⁻¹	2 ⁻²	2 ⁻³	2 ⁻⁴	2 ⁻⁵	2 ⁻⁶	2 ⁻⁷			2 ⁻⁸	
0.03125														1	0	0	0		
-0.250	✓									1	0	0	0	0	0	0	0		
0.250										1	0	0	0	0	0	0	0		
0.500									1	0	0	0	0	0	0	0	0		
1.000							1	0	0	0	0	0	0	0	0	0	0		
2.100						1	0	0	0	0	1	1	0	0	0	0	1		
-2.125	✓					1	0	0	0	1	0	0	0	0	0	0	0		
8.250				1	0	0	0	0	1	0	0	0	0	0	0	0	0		
16.250			1	0	0	0	0	0	1	0	0	0	0	0	0	0	0		
✓			1	1	0	2	1	1	2	0	0	1	MSB Sum By Column						

- 4 The MSB counts of each bit location are displayed as a heat map. In this heat map, darker blue regions correspond to a larger number of MSB's in the bit location.

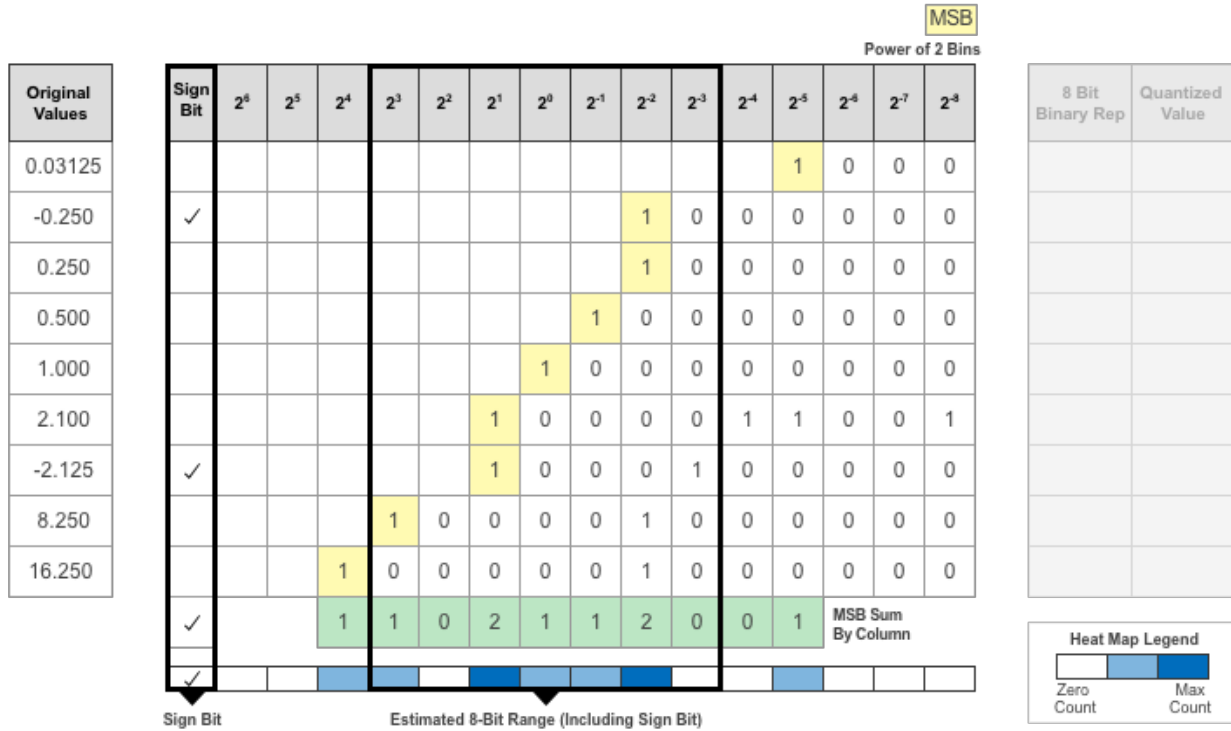
Original Values	Sign Bit	Power of 2 Bins														8 Bit Binary Rep	Quantized Value		
		2 ⁶	2 ⁵	2 ⁴	2 ³	2 ²	2 ¹	2 ⁰	2 ⁻¹	2 ⁻²	2 ⁻³	2 ⁻⁴	2 ⁻⁵	2 ⁻⁶	2 ⁻⁷			2 ⁻⁸	
0.03125													1	0	0	0			
-0.250	✓									1	0	0	0	0	0	0			
0.250										1	0	0	0	0	0	0			
0.500									1	0	0	0	0	0	0	0			
1.000							1	0	0	0	0	0	0	0	0	0			
2.100						1	0	0	0	0	1	1	0	0	0	1			
-2.125	✓					1	0	0	0	1	0	0	0	0	0	0			
8.250				1	0	0	0	0	1	0	0	0	0	0	0	0			
16.250			1	0	0	0	0	0	1	0	0	0	0	0	0	0			
✓			1	1	0	2	1	1	2	0	0	1	MSB Sum By Column						
✓																			

Heat Map Legend

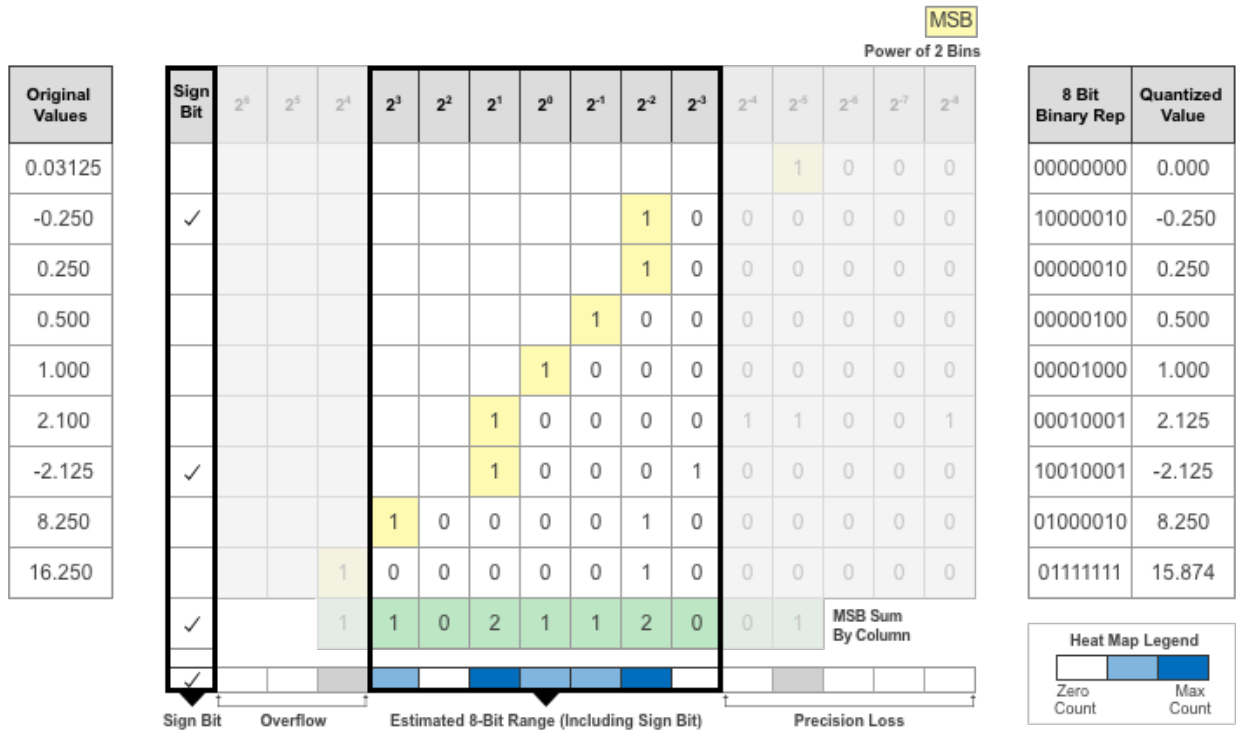
Zero Count		Max Count

- The **Deep Network Quantizer** app assigns a data type that can avoid overflow, cover the range, and allow underflow. An additional sign bit is required to represent the signedness of the value.

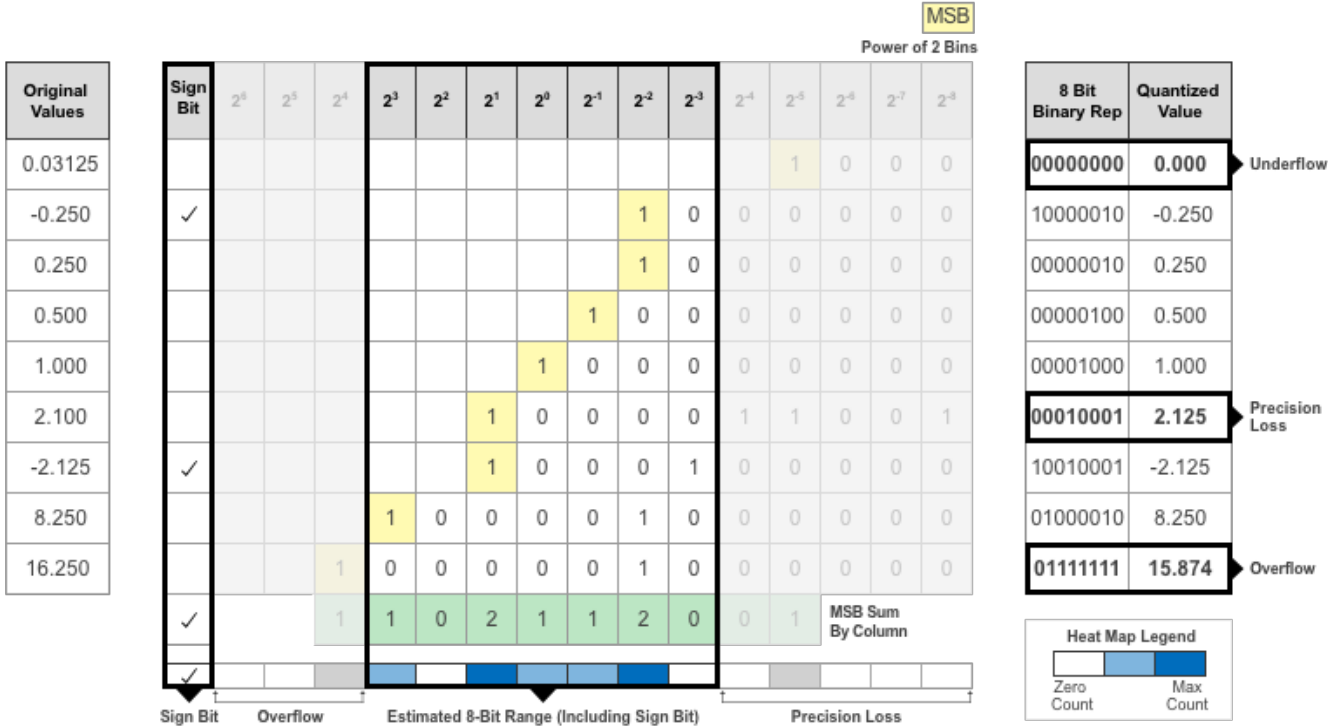
The figure below shows an example of a data type that represents bits from 2^3 to 2^{-3} , including the sign bit.



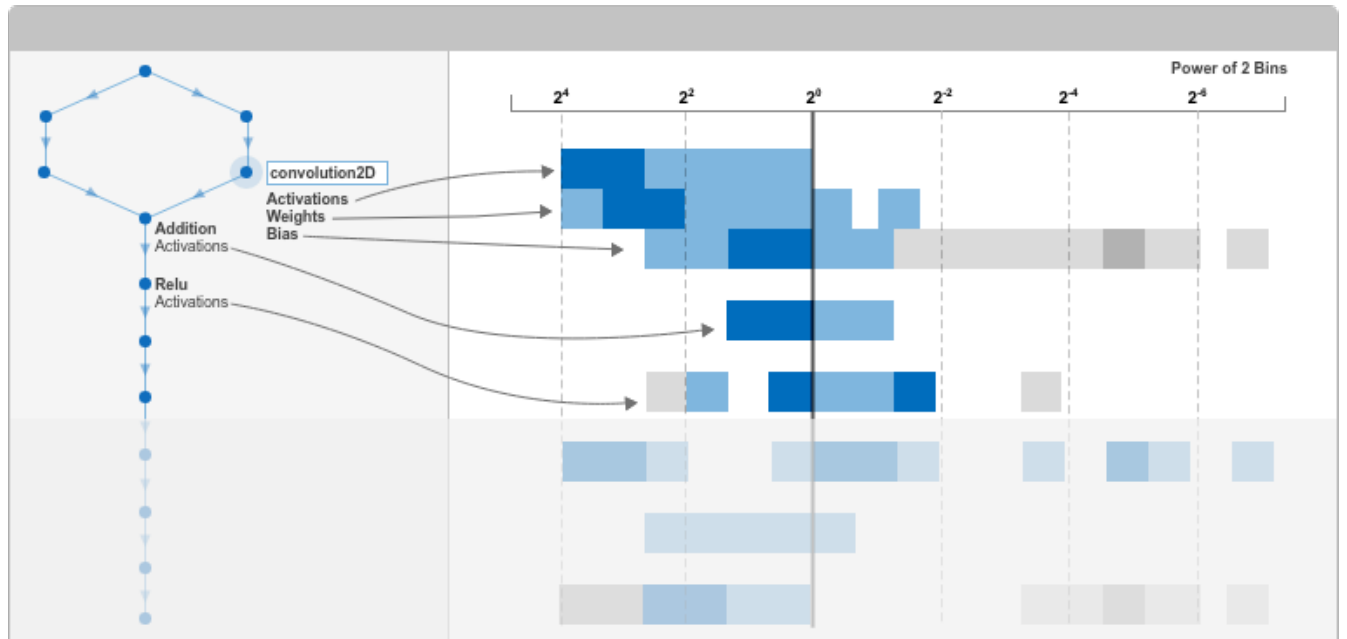
- After assigning the data type, any bits outside of that data type are removed. Due to the assignment of a smaller data type of fixed length, precision loss, overflow, and underflow can occur for values that are not representable by the data type.



In this example, the value 0.03125, suffers from an underflow, so the quantized value is 0. The value 2.1 suffers some precision loss, so the quantized value is 2.125. The value 16.250 is larger than the largest representable value of the data type, so this value overflows and the quantized value saturates to 15.874.



7 The **Deep Network Quantizer** app displays this heat map histogram for each learnable parameter in the convolution layers and fully connected layers of the network. The gray regions of the histogram show the bits that cannot be represented by the data type.



See Also

Apps

Deep Network Quantizer

Functions

calibrate | validate | dlquantizer | dlquantizationOptions

More About

- “Generate INT8 Code for Deep Learning Networks” on page 4-107
- “Quantize Residual Network Trained for Image Classification and Generate CUDA Code” on page 4-229
- “Quantize Layers in Object Detectors and Generate CUDA Code” on page 4-237

Generate INT8 Code for Deep Learning Networks

Deep learning uses neural network architectures that contain many processing layers, including convolutional layers. Deep learning models typically work on large sets of labeled data. Training these models and performing inference is computationally intensive, consuming significant amount of memory. Neural networks use memory to store input data, parameters (weights), and activations from each layer as the input propagates through the network. The majority of the pretrained neural networks and neural networks trained by using Deep Learning Toolbox use single-precision floating point data types. Even networks that are small in size require a considerable amount of memory and hardware to perform these floating-point arithmetic operations. These restrictions can inhibit deployment of deep learning models to devices that have low computational power and smaller memory resources. By using a lower precision to store the weights and activations, you can reduce the memory requirements of the network.

You can use Deep Learning Toolbox in tandem with the Deep Learning Toolbox Model Quantization Library support package to reduce the memory footprint of a deep neural network by quantizing the weights, biases, and activations of convolution layers to 8-bit scaled integer data types. Then, you can use GPU Coder to generate CUDA code for the optimized network. The generated code takes advantage of NVIDIA CUDA deep neural network library (cuDNN) or the TensorRT high performance inference library. The generated code can be integrated into your project as source code, static or dynamic libraries, or executables that you can deploy to a variety of NVIDIA GPU platforms.

Classify Images Using a Network Optimized for INT8 Inference

This example was previously named 'Classify Images on a GPU Using a Quantized Network' but renamed in R2022a to avoid confusion with quantized network objects created by the `quantize` (Deep Learning Toolbox) function. Code generation does not support quantized deep neural networks produced by the `quantize` function.

In this example, you use GPU Coder to generate optimized CUDA code for a deep convolutional neural network and classify an image. The generated code performs inference computation using 8-bit integer data type for the convolution layer. The example uses the pretrained squeezenet (Deep Learning Toolbox) convolutional neural network.

SqueezeNet has been trained on over a million images and can classify images into 1000 object categories (such as keyboard, coffee mug, pencil, and many animals). The network has learned rich feature representations for a wide range of images. The network takes an image as input and outputs a label for the object in the image together with the probabilities for each of the object categories.

This example consists of the following steps:

- Modify the SqueezeNet neural network to classify a smaller subset of images containing five object categories using transfer learning.
- Use the `calibrate` (Deep Learning Toolbox) function to exercise the network with sample inputs and collect range information to produce a calibration result file.
- Generate optimized code for the network by using the `codegen` command and the calibration result file.

Third-Party Prerequisites

Required

- CUDA enabled NVIDIA® GPU and compatible driver.

Optional

For non-MEX builds such as static, dynamic libraries or executables, this example has the following additional requirements.

- NVIDIA CUDA toolkit.
- NVIDIA cuDNN library.
- Environment variables for the compilers and libraries. For more information, see “Third-Party Hardware” and “Setting Up the Prerequisite Products”.

Transfer Learning Using SqueezeNet

To perform classification on a new set of images, you fine-tune a pretrained SqueezeNet convolutional neural network by transfer learning. In transfer learning, you can take a pretrained network and use it as a starting point to learn a new task. Fine-tuning a network with transfer learning is usually much faster and easier than training a network with randomly initialized weights from scratch. You can quickly transfer learned features to a new task using a smaller number of training images.

Load Training Data

Unzip and load the new images as an image datastore. The `imageDatastore` function automatically labels the images based on folder names and stores the data as an `ImageDatastore` object. An image datastore enables you to store large image data, including data that does not fit in memory, and efficiently read batches of images during training of a convolutional neural network. Divide the data into training and validation data sets. Use 70% of the images for training and 30% for validation. `splitEachLabel` splits the `imds` datastore into two new datastores.

```
unzip('MerchData.zip');
imds = imageDatastore('MerchData', ...
    'IncludeSubfolders',true, ...
    'LabelSource','foldernames');
[imdsTrain,imdsValidation] = splitEachLabel(imds,0.7,'randomized');

numTrainImages = numel(imdsTrain.Labels);
idx = randperm(numTrainImages,4);
img = imtile(imds, 'Frames', idx);

figure
imshow(img)
title('Random Images from Training Dataset');
```

Random Images from Training Dataset



Load Pretrained Network

Load the pretrained SqueezeNet network. If you do not have the required support packages installed, the software provides a download link.

```
net = squeezeNet;
```

The object `net` contains the `DAGNetwork` object. The first layer, the image input layer, requires input images of size 227-by-227-by-3, where 3 is the number of color channels. You can use the `analyzeNetwork` (Deep Learning Toolbox) function to display an interactive visualization of the network architecture, to detect errors and issues in the network, and to display detailed information about the network layers. The layer information includes the sizes of layer activations and learnable parameters, the total number of learnable parameters, and the sizes of state parameters of recurrent layers.

```
inputSize = net.Layers(1).InputSize;
```

Replace Final Layers

The convolutional layers of the network extract image features that the last learnable layer and the final classification layer use to classify the input image. These two layers, 'conv10' and 'ClassificationLayer_predictions' in SqueezeNet, contain information on how to combine the features that the network extracts into class probabilities, a loss value, and predicted labels.

To retrain a pretrained network to classify new images, replace these two layers with new layers adapted to the new data set. You can do this manually or use the helper function `findLayersToReplace` to find these layers automatically.

```
lgraph = layerGraph(net);
[learnableLayer,classLayer] = findLayersToReplace(lgraph);
numClasses = numel(categories(imdsTrain.Labels));

newConvLayer = convolution2dLayer([1, 1],numClasses,'WeightLearnRateFactor',...
10,'BiasLearnRateFactor',10,"Name",'new_conv');
lgraph = replaceLayer(lgraph,'conv10',newConvLayer);

newClassificatonLayer = classificationLayer('Name','new_classoutput');
lgraph = replaceLayer(lgraph,'ClassificationLayer_predictions',newClassificatonLayer);
```

Train Network

The network requires input images of size 227-by-227-by-3, but the images in the image datastores have different sizes. Use an augmented image datastore to automatically resize the training images. Specify additional augmentation operations to perform on the training images: randomly flip the training images along the vertical axis, and randomly translate them up to 30 pixels horizontally and vertically. Data augmentation helps prevent the network from over-fitting and memorizing the exact details of the training images.

```
pixelRange = [-30 30];
imageAugmenter = imageDataAugmenter( ...
    'RandXReflection',true, ...
    'RandXTranslation',pixelRange, ...
    'RandYTranslation',pixelRange);
augimdsTrain = augmentedImageDatastore(inputSize(1:2),imdsTrain, ...
    'DataAugmentation',imageAugmenter);
```

To automatically resize the validation images without performing further data augmentation, use an augmented image datastore without specifying any additional preprocessing operations.

```
augimdsValidation = augmentedImageDatastore(inputSize(1:2),imdsValidation);
```

Specify the training options. For transfer learning, keep the features from the early layers of the pretrained network (the transferred layer weights). To slow down learning in the transferred layers, set the initial learning rate to a small value. In the previous step, you increased the learning rate factors for the convolutional layer to speed up learning in the new final layers. This combination of learning rate settings results in fast learning only in the new layers and slower learning in the other layers. When performing transfer learning, you do not need to train for as many epochs. An epoch is a full training cycle on the entire training data set. Specify the mini-batch size to be 11 so that in each epoch you consider all of the data. The software validates the network every `ValidationFrequency` iterations during training.

```
options = trainingOptions('sgdm', ...
    'MiniBatchSize',11, ...
```



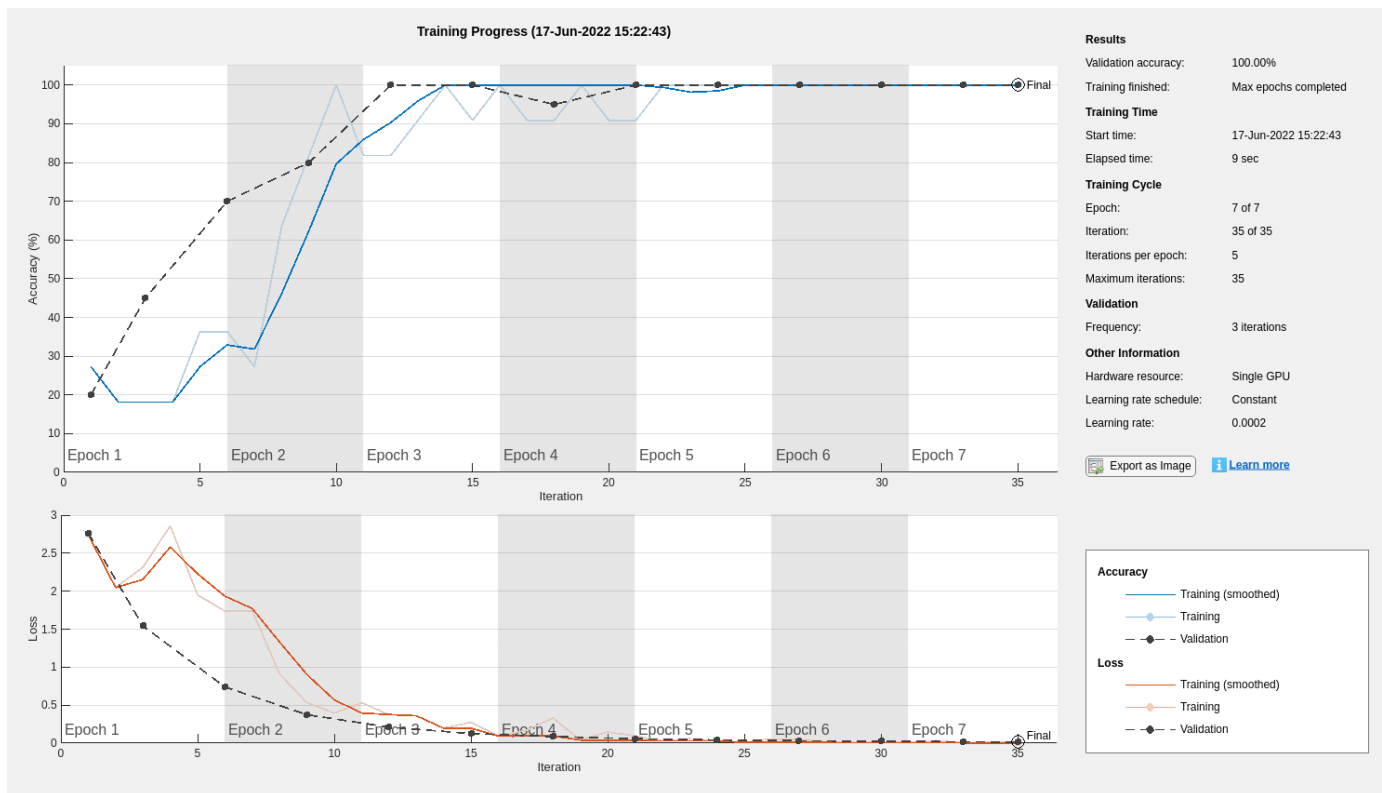
```

'MaxEpochs',7, ...
'InitialLearnRate',2e-4, ...
'Shuffle','every-epoch', ...
'ValidationData',augimdsValidation, ...
'ValidationFrequency',3, ...
'Verbose',false, ...
'Plots','training-progress');

```

Train the network that consists of the transferred and new layers.

```
netTransfer = trainNetwork(augimdsTrain,lgraph,options);
```



```

classNames = netTransfer.Layers(end).Classes;
save('mySqueezenet.mat','netTransfer');

```

Generate Calibration Result File for the Network

Create a `dlquantizer` object and specify the network.

```
quantObj = dlquantizer(netTransfer);
```

Define a metric function to use to compare the behavior of the network before and after quantization.

```
type('hComputeModelAccuracy.m');
```

```

function accuracy = hComputeModelAccuracy(predictionScores, net, datastore)
%% Computes model-level accuracy statistics

```

```

    % Load ground truth
    tmp = readall(datastore);

```

```

groundTruth = tmp.response;

% Compare with predicted label with actual ground truth
predictionError = {};
for idx=1:numel(groundTruth)
    [~, idy] = max(predictionScores(idx,:));
    yActual = net.Layers(end).Classes(idy);
    predictionError{end+1} = (yActual == groundTruth(idx)); %#ok
end

% Sum all prediction errors.
predictionError = [predictionError{:}];
accuracy = sum(predictionError)/numel(predictionError);
end

```

Specify the metric function in a `dlquantizationOptions` object.

```

quantOpts = dlquantizationOptions('MetricFcn', ...
    {@(x)hComputeModelAccuracy(x,netTransfer,augimdsValidation)});

```

Use the `calibrate` function to exercise the network with sample inputs and collect range information. The `calibrate` function exercises the network and collects the dynamic ranges of the weights and biases in the convolution and fully connected layers of the network and the dynamic ranges of the activations in all layers of the network. The function returns a table. Each row of the table contains range information for a learnable parameter of the optimized network.

```

calResults = calibrate(quantObj,augimdsTrain);
save('squeezenetCalResults.mat','calResults');
save('squeezenetQuantObj.mat','quantObj');

```

You can use the `validate` function to quantize the learnable parameters in the convolution layers of the network and exercise the network. The function uses the metric function defined in the `dlquantizationOptions` object to compare the results of the network before and after quantization.

```

valResults = validate(quantObj,augimdsValidation,quantOpts);

```

Create an Entry-Point Function

Write an entry-point function in MATLAB that:

- Uses the `coder.loadDeepLearningNetwork` function to load a deep learning model and to construct and set up a CNN class. For more information, see “Load Pretrained Networks for Code Generation” on page 4-66.
- Calls the `predict` function to predict the responses.

```

type('predict_int8.m');

function out = predict_int8(netFile, in)

    persistent mynet;
    if isempty(mynet)
        mynet = coder.loadDeepLearningNetwork(netFile);
    end
    out = predict(mynet,in);
end

```

A persistent object `mynet` loads the `DAGNetwork` object. At the first call to the entry-point function, the persistent object is constructed and set up. On subsequent calls to the function, the same object is reused to call `predict` on inputs, avoiding reconstructing and reloading the network object.

Note

Ensure that all the preprocessing operations performed in the calibration and validation steps are included in the design file.

Code Generation by Using `codegen`

To configure build settings such as output file name, location, and type, you create `coder` configuration objects. To create the objects, use the `coder.gpuConfig` function. For example, when generating CUDA MEX using the `codegen` command, use `cfg = coder.gpuConfig('mex');`

To specify code generation parameters for cuDNN, set the `DeepLearningConfig` property to a `coder.CuDNNConfig` object that you create by using `coder.DeepLearningConfig`.

```
cfg = coder.gpuConfig('mex');
cfg.TargetLang = 'C++';
cfg.GpuConfig.ComputeCapability = '6.1';
cfg.DeepLearningConfig = coder.DeepLearningConfig('cudnn');
cfg.DeepLearningConfig.AutoTuning = true;
cfg.DeepLearningConfig.CalibrationResultFile = 'squeezeNetQuantObj.mat';
cfg.DeepLearningConfig.DataType = 'int8';
```

Specify the location of the MAT-file containing the calibration data.

Specify the precision of the inference computations in supported layers by using the `DataType` property. For 8-bit integer, use `'int8'`. Use the `ComputeCapability` property of the code configuration object to set the appropriate compute capability value.

Run the `codegen` command. The `codegen` command generates CUDA code from the `predict_int8.m` MATLAB entry-point function.

```
inputs = {coder.Constant('mySqueezeNet.mat'),ones(inputSize,'uint8')};
codegen -config cfg -args inputs predict_int8
```

Code generation successful.

When code generation is successful, you can view the resulting code generation report by clicking **View Report** in the MATLAB Command Window. The report is displayed in the Report Viewer window. If the code generator detects errors or warnings during code generation, the report describes the issues and provides links to the problematic MATLAB code.

Run the Generated MEX

The image that you want to classify must have the same size as the input size of the network. Read the image that you want to classify and resize it to the input size of the network. This resizing slightly changes the aspect ratio of the image.

```
testImage = imread("MerchDataTest.jpg");
testImage = imresize(testImage,inputSize(1:2));
```

Call SqueezeNet `predict` on the input image.

```

predictScores(:,1) = predict(netTransfer,testImage)';
predictScores(:,2) = predict_int8_mex('mySqueezenet.mat',testImage);

```

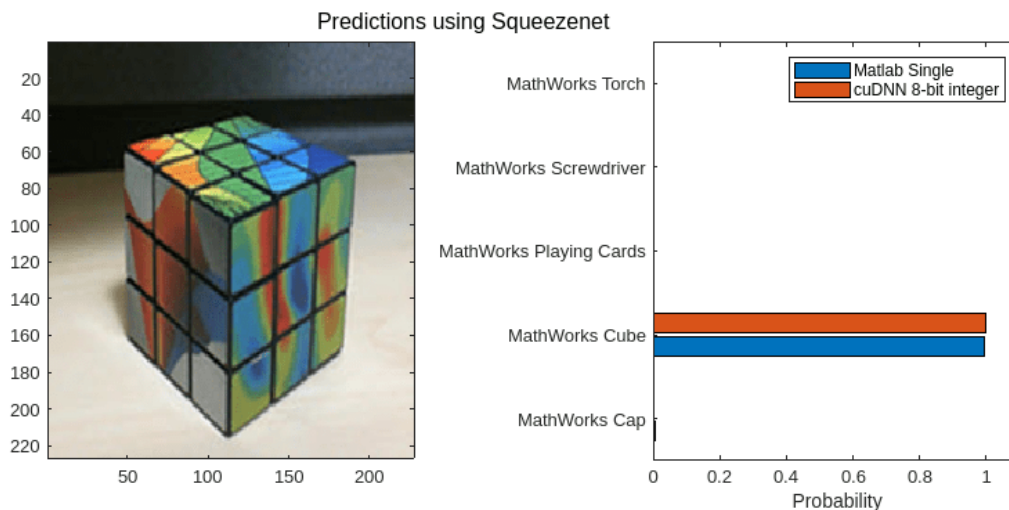
Display the predicted labels and their associated probabilities as a histogram.

```

h = figure;
h.Position(3) = 2*h.Position(3);
ax1 = subplot(1,2,1);
ax2 = subplot(1,2,2);

image(ax1,testImage);
barh(ax2,predictScores)
xlabel(ax2,'Probability')
yticklabels(ax2,classNames)
ax2.XLim = [0 1.1];
ax2.YAxisLocation = 'left';
legend('Matlab Single','cuDNN 8-bit integer');
sgtitle('Predictions using Squeezenet')

```



Helper Functions

```

function [learnableLayer,classLayer] = findLayersToReplace(lgraph)
% findLayersToReplace(lgraph) finds the single classification layer and the
% preceding learnable (fully connected or convolutional) layer of the layer
% graph lgraph.

if ~isa(lgraph,'nnet.cnn.LayerGraph')
    error('Argument must be a LayerGraph object.')
end

% Get source, destination, and layer names.
src = string(lgraph.Connections.Source);
dst = string(lgraph.Connections.Destination);
layerNames = string({lgraph.Layers.Name}');

% Find the classification layer. The layer graph must have a single
% classification layer.
isClassificationLayer = arrayfun(@(l) ...
    (isa(l,'nnet.cnn.layer.ClassificationOutputLayer') ...

```

```

|isa(l, 'nnet.layer.ClassificationLayer')), ...
    lgraph.Layers);

if sum(isClassificationLayer) ~= 1
    error('Layer graph must have a single classification layer.')
end
classLayer = lgraph.Layers(isClassificationLayer);

% Traverse the layer graph in reverse starting from the classification
% layer. If the network branches, throw an error.
currentLayerIdx = find(isClassificationLayer);
while true

    if numel(currentLayerIdx) ~= 1
        msg = ['Layer graph must have a single learnable layer ' ...
            'preceding the classification layer.'];
        error(msg)
    end

    currentLayerType = class(lgraph.Layers(currentLayerIdx));
    isLearnableLayer = ismember(currentLayerType, ...
        ['nnet.cnn.layer.FullyConnectedLayer', 'nnet.cnn.layer.Convolution2DLayer']);

    if isLearnableLayer
        learnableLayer = lgraph.Layers(currentLayerIdx);
        return
    end

    currentDstIdx = find(layerNames(currentLayerIdx) == dst);
    currentLayerIdx = find(src(currentDstIdx) == layerNames);

end

end

```

Limitations

- When performing inference in INT8 precision using cuDNN version 8.1.0, issues in the NVIDIA library may cause significant degradation in performance.
- The following layers are not supported for 8-bit integer quantization when targeting the NVIDIA CUDA deep neural network library (cuDNN) library.
 - leakyReluLayer
 - clippedReluLayer
 - globalAveragePooling2dLayer

See Also

Apps

Deep Network Quantizer

Functions

dlquantizer | dlquantizationOptions | calibrate | validate |
 coder.loadDeepLearningNetwork | codegen

Objects

`coder.CuDNNConfig` | `coder.TensorRTConfig`

More About

- “Quantization of Deep Neural Networks” on page 4-99
- “Quantize Residual Network Trained for Image Classification and Generate CUDA Code” on page 4-229
- “Quantize Layers in Object Detectors and Generate CUDA Code” on page 4-237
- “Code Generation for Deep Learning Networks by Using cuDNN” on page 4-69
- “Code Generation for Deep Learning Networks by Using TensorRT” on page 4-78

Code Generation for Deep Learning Networks

This example shows how to perform code generation for an image classification application that uses deep learning. It uses the `codegen` command to generate a MEX function that runs prediction by using image classification networks such as MobileNet-v2, ResNet, and GoogLeNet.

Third-Party Prerequisites

Required

This example generates CUDA MEX and has the following third-party requirements.

- CUDA® enabled NVIDIA® GPU and compatible driver.

Optional

For non-MEX builds such as static, dynamic libraries or executables, this example has the following additional requirements.

- NVIDIA toolkit.
- NVIDIA cuDNN library.
- Environment variables for the compilers and libraries. For more information, see “Third-Party Hardware” and “Setting Up the Prerequisite Products”.

Verify GPU Environment

Use the `coder.checkGpuInstall` function to verify that the compilers and libraries necessary for running this example are set up correctly.

```
envCfg = coder.gpuEnvConfig('host');
envCfg.DeepLibTarget = 'cudnn';
envCfg.DeepCodegen = 1;
envCfg.Quiet = 1;
coder.checkGpuInstall(envCfg);
```

mobilenetv2_predict Entry-Point Function

MobileNet-v2 is a convolutional neural network that is trained on more than a million images from the ImageNet database. The network is 155 layers deep and can classify images into 1000 object categories, such as keyboard, mouse, pencil, and many animals. The network has an image input size of 224-by-224. Use the `analyzeNetwork` (Deep Learning Toolbox) function to display an interactive visualization of the deep learning network architecture.

```
net = mobilenetv2();
analyzeNetwork(net);
```

The `mobilenetv2_predict.m` entry-point function takes an image input and runs prediction on the image using the pretrained MobileNet-v2 convolutional neural network. The function uses a persistent object `myNet` to load the series network object and reuses the persistent object for prediction on subsequent calls.

```
type('mobilenetv2_predict.m')
```

```
% Copyright 2017-2019 The MathWorks, Inc.
```

```
function out = mobilenetv2_predict(in)
%#codegen

persistent mynet;

if isempty(mynet)
    mynet = coder.loadDeepLearningNetwork('mobilenetv2','mobilenetv2');
end

% pass in input
out = mynet.predict(in);
```

Run MEX Code Generation

To generate CUDA code for the `mobilenetv2_predict` entry-point function, create a GPU code configuration object for a MEX target and set the target language to C++. Use the `coder.DeepLearningConfig` function to create a CuDNN deep learning configuration object and assign it to the `DeepLearningConfig` property of the GPU code configuration object. Run the `codegen` command and specify an input size of `[224,224,3]`. This value corresponds to the input layer size of the MobileNet-v2 network.

```
cfg = coder.gpuConfig('mex');
cfg.TargetLang = 'C++';
cfg.DeepLearningConfig = coder.DeepLearningConfig('cudnn');
codegen -config cfg mobilenetv2_predict -args {ones(224,224,3)} -report
```

Code generation successful: [View report](#)

Generated Code Description

The series network is generated as a C++ class containing an array of 155 layer classes and functions to set up, call `predict`, and clean up the network.

```
class b_mobilenetv2_0
{
    ....
public:
    b_mobilenetv2_0();
    void setup();
    void predict();
    void cleanup();
    ~b_mobilenetv2_0();
};
```

The `setup()` method of the class sets up handles and allocates memory for each layer of the network object. The `predict()` method performs prediction for each of the 155 layers in the network.

The entry-point function `mobilenetv2_predict()` in the generated code file `mobilenetv2_predict.cu` constructs a static object of `b_mobilenetv2` class type and invokes `setup` and `predict` on this network object.

```
static b_mobilenetv2_0 mynet;
static boolean_T mynet_not_empty;

/* Function Definitions */
void mobilenetv2_predict(const real_T in[150528], real32_T out[1000])
{
    if (!mynet_not_empty) {
```



```
    DeepLearningNetwork_setup(&mynet);  
    mynet_not_empty = true;  
}  
  
/* pass in input */  
DeepLearningNetwork_predict(&mynet, in, out);  
}
```

Binary files are exported for layers with parameters such as fully connected and convolution layers in the network. For instance, files `cnn_mobilenetv2_conv*_w` and `cnn_mobilenetv2_conv*_b` correspond to weights and bias parameters for the convolution layers in the network. To see a list of the generated files, use:

```
dir(fullfile(pwd, 'codegen', 'mex', 'mobilenetv2_predict'))
```

Run Generated MEX

Load an input image.

```
im = imread('peppers.png');  
imshow(im);
```



Call `mobilenetv2_predict_mex` on the input image.

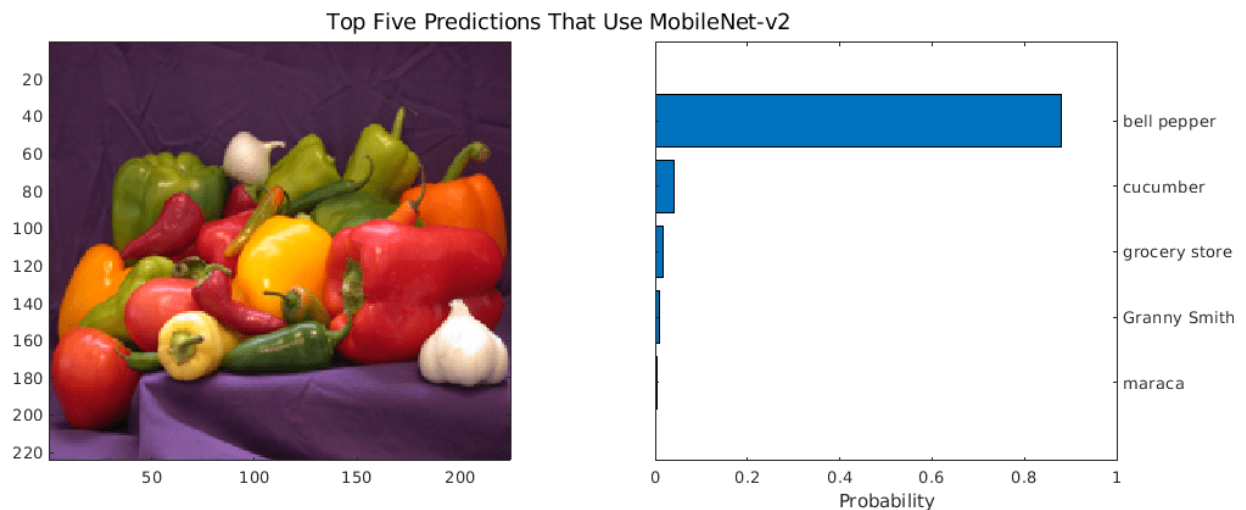
```
im = imresize(im, [224,224]);  
predict_scores = mobilenetv2_predict_mex(double(im));
```

Get the top five prediction scores and their labels.

```
[scores,indx] = sort(predict_scores, 'descend');
classNames = net.Layers(end).ClassNames;
classNamesTop = classNames(indx(1:5));

h = figure;
h.Position(3) = 2*h.Position(3);
ax1 = subplot(1,2,1);
ax2 = subplot(1,2,2);

image(ax1,im);
barh(ax2,scores(5:-1:1))
xlabel(ax2,'Probability')
yticklabels(ax2,classNamesTop(5:-1:1))
ax2.YAxisLocation = 'right';
sgtitle('Top Five Predictions That Use MobileNet-v2')
```



Clear the static network object that was loaded in memory.

```
clear mex;
```

Classification of Images by Using ResNet-50 network

You can also use the DAG network ResNet-50 for image classification. A pretrained ResNet-50 model for MATLAB is available in the ResNet-50 support package of Deep Learning Toolbox. To download and install the support package, use the Add-On Explorer. To learn more about finding and installing add-ons, see “Get and Manage Add-Ons”.

```
net = resnet50;
disp(net)
```

DAGNetwork with properties:

```
Layers: [177x1 nnet.cnn.layer.Layer]
Connections: [192x2 table]
InputNames: {'input_1'}
OutputNames: {'ClassificationLayer_fc1000'}
```

Run MEX Code Generation

To generate CUDA code for the `resnet_predict.m` entry-point function, create a GPU code configuration object for a MEX target and set the target language to C++. This entry-point function calls the `resnet50` function to load the network and perform prediction on the input image.

```
cfg = coder.gpuConfig('mex');
cfg.TargetLang = 'C++';
cfg.DeepLearningConfig = coder.DeepLearningConfig('cudnn');
codegen -config cfg resnet_predict -args {ones(224,224,3)} -report
```

Code generation successful: [View report](#)

Call `resnet_predict_mex` on the input image.

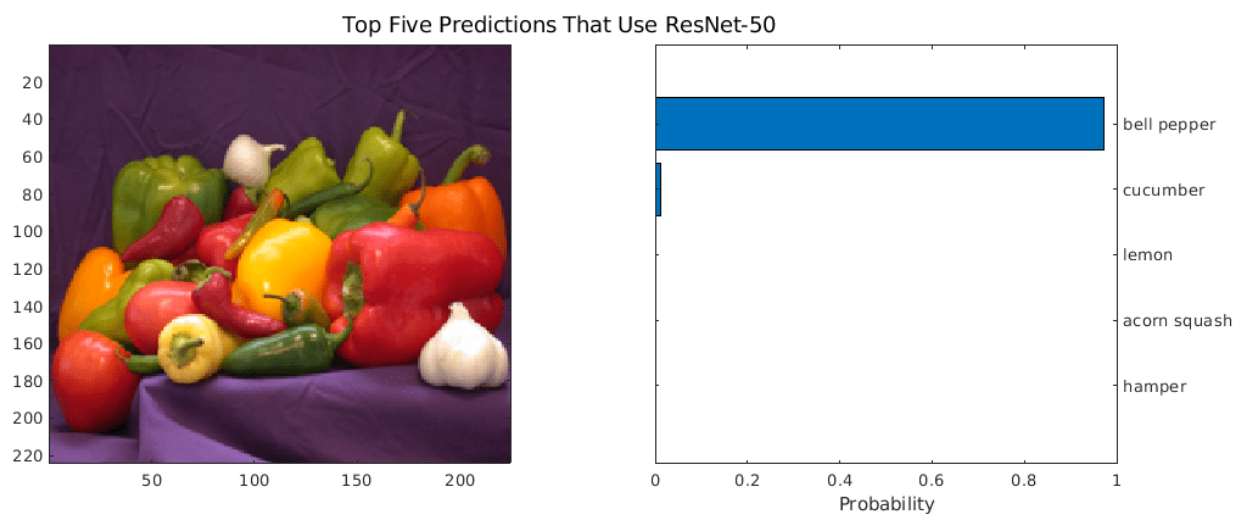
```
predict_scores = resnet_predict_mex(double(im));
```

Get the top five prediction scores and their labels.

```
[scores,indx] = sort(predict_scores, 'descend');
classNames = net.Layers(end).ClassNames;
classNamesTop = classNames(indx(1:5));
```

```
h = figure;
h.Position(3) = 2*h.Position(3);
ax1 = subplot(1,2,1);
ax2 = subplot(1,2,2);
```

```
image(ax1,im);
barh(ax2,scores(5:-1:1))
xlabel(ax2,'Probability')
yticklabels(ax2,classNamesTop(5:-1:1))
ax2.YAxisLocation = 'right';
sgtitle('Top Five Predictions That Use ResNet-50')
```



Clear the static network object that was loaded in memory.

```
clear mex;
```

Classification of Images by Using GoogLeNet (Inception) network

A pretrained GoogLeNet model for MATLAB is available in the GoogLeNet support package of Deep Learning Toolbox. To download and install the support package, use the Add-On Explorer. To learn more about finding and installing add-ons, see “Get and Manage Add-Ons”.

```
net = googlenet;  
disp(net)
```

DAGNetwork with properties:

```
Layers: [144x1 nnet.cnn.layer.Layer]  
Connections: [170x2 table]  
InputNames: {'data'}  
OutputNames: {'output'}
```

Run MEX Code Generation

Generate CUDA code for the `googlenet_predict.m` entry-point function. This entry-point function calls the `googlenet` function to load the network and perform prediction on the input image. To generate code for this entry-point function, create a GPU configuration object for MEX target.

```
cfg = coder.gpuConfig('mex');  
cfg.TargetLang = 'C++';  
cfg.DeepLearningConfig = coder.DeepLearningConfig('cudnn');  
codegen -config cfg googlenet_predict -args {ones(224,224,3)} -report
```

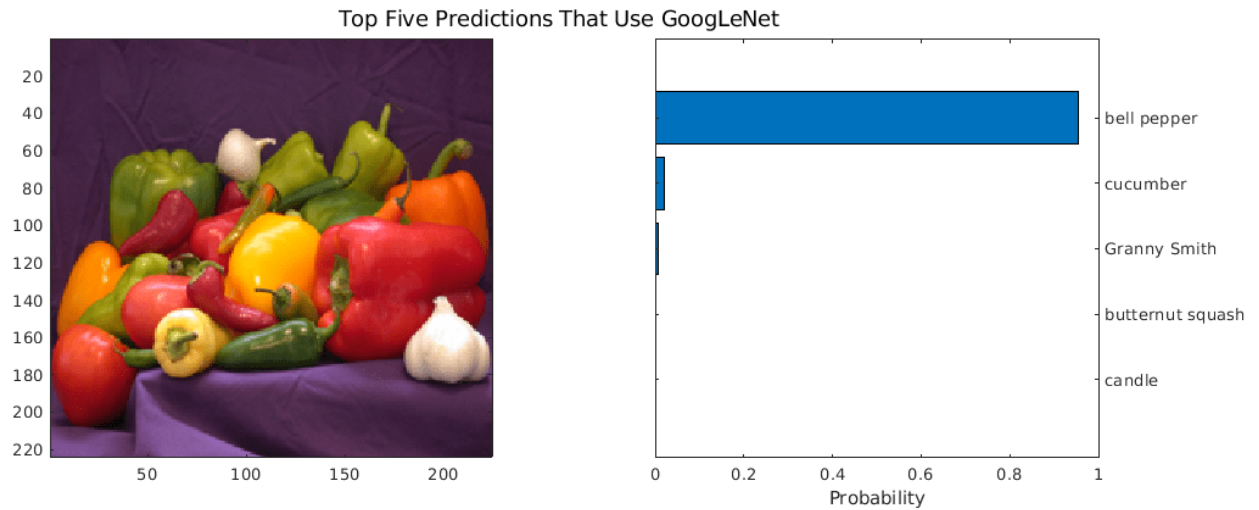
Code generation successful: [View report](#)

Call `googlenet_predict_mex` on the input image.

```
im = imresize(im, [224,224]);  
predict_scores = googlenet_predict_mex(double(im));
```

Get the top five prediction scores and their labels.

```
[scores,indx] = sort(predict_scores, 'descend');  
classNames = net.Layers(end).ClassNames;  
classNamesTop = classNames(indx(1:5));  
  
h = figure;  
h.Position(3) = 2*h.Position(3);  
ax1 = subplot(1,2,1);  
ax2 = subplot(1,2,2);  
  
image(ax1,im);  
barh(ax2,scores(5:-1:1))  
xlabel(ax2,'Probability')  
yticklabels(ax2,classNamesTop(5:-1:1))  
ax2.YAxisLocation = 'right';  
sgtitle('Top Five Predictions That Use GoogLeNet')
```



Clear the static network object that was loaded in memory.

```
clear mex;
```

See Also

Functions

`coder.checkGpuInstall` | `codegen` | `coder.DeepLearningConfig` |
`coder.loadDeepLearningNetwork` | `mobilenetv2` | `resnet50` | `googlenet`

Objects

`coder.gpuConfig` | `coder.CodeConfig` | `coder.EmbeddedCodeConfig` | `coder.gpuEnvConfig` |
`coder.CuDNNConfig` | `coder.TensorRTConfig`

See Also

More About

- “Deep Learning in MATLAB” (Deep Learning Toolbox)
- “Generated CNN Class Hierarchy” on page 4-65
- “Supported Networks, Layers, and Classes” on page 4-6
- “Load Pretrained Networks for Code Generation” on page 4-66
- “Code Generation for Deep Learning Networks by Using cuDNN” on page 4-69
- “Code Generation for Deep Learning Networks by Using TensorRT” on page 4-78

Lane Detection Optimized with GPU Coder

This example shows how to develop a deep learning lane detection application that runs on NVIDIA® GPUs.

The pretrained lane detection network can detect and output lane marker boundaries from an image and is based on the AlexNet network. The last few layers of the AlexNet network are replaced by a smaller fully connected layer and regression output layer. The example generates a CUDA executable that runs on a CUDA-enabled GPU on the host machine.

Prerequisites

- CUDA enabled NVIDIA GPU.
- NVIDIA CUDA toolkit and driver.
- NVIDIA cuDNN library.
- Environment variables for the compilers and libraries. For information on the supported versions of the compilers and libraries, see “Third-Party Hardware”. For setting up the environment variables, see “Setting Up the Prerequisite Products”.

Verify GPU Environment

Use the `coder.checkGpuInstall` function to verify that the compilers and libraries necessary for running this example are set up correctly.

```
envCfg = coder.gpuEnvConfig('host');
envCfg.DeepLibTarget = 'cudnn';
envCfg.DeepCodegen = 1;
envCfg.Quiet = 1;
coder.checkGpuInstall(envCfg);
```

Get Pretrained Lane Detection Network

This example uses the `trainedLaneNet` MAT-file containing the pretrained lane detection network. This file is approximately 143 MB size. Download the file from the MathWorks website.

```
laneNetFile = matlab.internal.examples.downloadSupportFile('gpcoder/cnn_models/lane_detection',
    'trainedLaneNet.mat');
```

This network takes an image as an input and outputs two lane boundaries that correspond to the left and right lanes of the ego vehicle. Each lane boundary is represented by the parabolic equation: $y = ax^2 + bx + c$, where y is the lateral offset and x is the longitudinal distance from the vehicle. The network outputs the three parameters a , b , and c per lane. The network architecture is similar to AlexNet except that the last few layers are replaced by a smaller fully connected layer and regression output layer.

```
load(laneNetFile);
disp(laneNet)
```

```
SeriesNetwork with properties:
```

```
    Layers: [23x1 nnet.cnn.layer.Layer]
  InputNames: {'data'}
 OutputNames: {'output'}
```

To view the network architecture, use the `analyzeNetwork` function.

```
analyzeNetwork(laneNet)
```

Download Test Video

To test the model, the example uses the a video file from the Caltech lanes dataset. The file is approximately 8 MB in size. Download the file from the MathWorks website.

```
videoFile = matlab.internal.examples.downloadSupportFile('gpucoder/media','caltech_cordova1.avi')
```

Main Entry-Point Function

The `detectLanesInVideo.m` file is the main entry-point function for code generation. The `detectLanesInVideo` function uses the `vision.VideoFileReader` (Computer Vision Toolbox) system object to read frames from the input video, calls the `predict` method of the LaneNet network object, and draws the detected lanes on the input video. A `vision.DeployableVideoPlayer` (Computer Vision Toolbox) system object is used to display the lane detected video output.

```
type detectLanesInVideo.m
```

```
function detectLanesInVideo(videoFile,net,laneCoeffMeans,laneCoeffsStds)
% detectLanesInVideo Entry-point function for the Lane Detection Optimized
% with GPU Coder example
%
% detectLanesInVideo(videoFile,net,laneCoeffMeans,laneCoeffsStds) uses the
% VideoFileReader system object to read frames from the input video, calls
% the predict method of the LaneNet network object, and draws the detected
% lanes on the input video. A DeployableVideoPlayer system object is used
% to display the lane detected video output.
%
% Copyright 2022 The MathWorks, Inc.
%#codegen
%% Create Video Reader and Video Player Object
videoFReader = vision.VideoFileReader(videoFile);
depVideoPlayer = vision.DeployableVideoPlayer(Name='Lane Detection on GPU');
%% Video Frame Processing Loop
while ~isDone(videoFReader)
    videoFrame = videoFReader();
    scaledFrame = 255.*(imresize(videoFrame,[227 227]));
    [laneFound,ltPts,rtPts] = laneNetPredict(net,scaledFrame, ...
        laneCoeffMeans,laneCoeffsStds);
    if(laneFound)
        pts = [reshape(ltPts',1,[]);reshape(rtPts',1,[])];
        videoFrame = insertShape(videoFrame, 'Line', pts, 'LineWidth', 4);
    end
    depVideoPlayer(videoFrame);
end
end
```

LaneNet Predict Function

The `laneNetPredict` function computes the right and left lane positions in a single video frame. The `laneNet` network computes parameters a , b , and c that describe the parabolic equation for the

left and right lane boundaries. From these parameters, compute the x and y coordinates corresponding to the lane positions. The coordinates must be mapped to image coordinates.

```
type laneNetPredict.m
```

```
function [laneFound,ltPts,rtPts] = laneNetPredict(net,frame,means,stds)
% laneNetPredict Predict lane markers on the input image frame using the
% lane detection network
%
% Copyright 2017-2022 The MathWorks, Inc.
%#codegen
% A persistent object lanenet is used to load the network object. At the
% first call to this function, the persistent object is constructed and
% setup. When the function is called subsequent times, the same object is
% reused to call predict on inputs, thus avoiding reconstructing and
% reloading the network object.
persistent lanenet;
if isempty(lanenet)
    lanenet = coder.loadDeepLearningNetwork(net, 'lanenet');
end
lanecoefsNetworkOutput = predict(lanenet,frame);
% Recover original coeffs by reversing the normalization steps.
params = lanecoefsNetworkOutput .* stds + means;
% 'c' should be more than 0.5 for it to be a lane.
isRightLaneFound = abs(params(6)) > 0.5;
isLeftLaneFound = abs(params(3)) > 0.5;
% From the networks output, compute left and right lane points in the image
% coordinates.
vehicleXPoints = 3:30;
ltPts = coder.nullcopy(zeros(28,2,'single'));
rtPts = coder.nullcopy(zeros(28,2,'single'));
if isRightLaneFound && isLeftLaneFound
    rtBoundary = params(4:6);
    rt_y = computeBoundaryModel(rtBoundary, vehicleXPoints);
    ltBoundary = params(1:3);
    lt_y = computeBoundaryModel(ltBoundary, vehicleXPoints);
    % Visualize lane boundaries of the ego vehicle.
    tform = get_tformToImage;
    % Map vehicle to image coordinates.
    ltPts = tform.transformPointsInverse([vehicleXPoints', lt_y]);
    rtPts = tform.transformPointsInverse([vehicleXPoints', rt_y]);
    laneFound = true;
else
    laneFound = false;
end
end
```



```

%% Helper Functions

% Compute boundary model.
function yWorld = computeBoundaryModel(model, xWorld)
yWorld = polyval(model, xWorld);
end

% Compute extrinsics.
function tform = get_tformToImage

%The camera coordinates are described by the caltech mono
% camera model.
yaw = 0;
pitch = 14; % Pitch of the camera in degrees
roll = 0;

translation = translationVector(yaw, pitch, roll);
rotation = rotationMatrix(yaw, pitch, roll);

% Construct a camera matrix.
focalLength = [309.4362, 344.2161];
principalPoint = [318.9034, 257.5352];
Skew = 0;

camMatrix = [rotation; translation] * intrinsicMatrix(focalLength, ...
    Skew, principalPoint);

% Turn camMatrix into 2-D homography.
tform2D = [camMatrix(1,:); camMatrix(2,:); camMatrix(4,:)]; % drop Z

tform = projective2d(tform2D);
tform = tform.invert();
end

% Translate to image co-ordinates.
function translation = translationVector(yaw, pitch, roll)
SensorLocation = [0 0];
Height = 2.1798; % mounting height in meters from the ground
rotationMatrix = (...
    rotZ(yaw)*... % last rotation
    rotX(90-pitch)*...
    rotZ(roll)... % first rotation
);

% Adjust for the SensorLocation by adding a translation.
sl = SensorLocation;

translationInWorldUnits = [sl(2), sl(1), Height];
translation = translationInWorldUnits*rotationMatrix;
end

% Rotation around X-axis.
function R = rotX(a)
a = deg2rad(a);
R = [...
    1 0 0;
    0 cos(a) -sin(a);

```

```

        0 sin(a) cos(a)];
end

% Rotation around Y-axis.
function R = rotY(a)
a = deg2rad(a);
R = [...
    cos(a) 0 sin(a);
    0      1 0;
    -sin(a) 0 cos(a)];
end

% Rotation around Z-axis.
function R = rotZ(a)
a = deg2rad(a);
R = [...
    cos(a) -sin(a) 0;
    sin(a) cos(a) 0;
    0      0      1];
end

% Given the Yaw, Pitch, and Roll, determine the appropriate Euler angles
% and the sequence in which they are applied to align the camera's
% coordinate system with the vehicle coordinate system. The resulting
% matrix is a Rotation matrix that together with the Translation vector
% defines the extrinsic parameters of the camera.
function rotation = rotationMatrix(yaw, pitch, roll)
rotation = (...
    rotY(180)*...           % last rotation: point Z up
    rotZ(-90)*...          % X-Y swap
    rotZ(yaw)*...          % point the camera forward
    rotX(90-pitch)*...     % "un-pitch"
    rotZ(roll)*...         % 1st rotation: "un-roll"
);
end

% Intrinsic matrix computation.
function intrinsicMat = intrinsicMatrix(FocalLength, Skew, PrincipalPoint)
intrinsicMat = ...
    [FocalLength(1) , 0 , 0; ...
    Skew , FocalLength(2) , 0; ...
    PrincipalPoint(1), PrincipalPoint(2), 1];
end

```

Generate CUDA Executable

To generate a standalone CUDA executable for the `detectLanesInVideo` entry-point function, create a GPU code configuration object for 'exe' target and set the target language to C++. Use the `coder.DeepLearningConfig` function to create a CuDNN deep learning configuration object and assign it to the `DeepLearningConfig` property of the GPU code configuration object.

```

cfg = coder.gpuConfig('exe');
cfg.DeepLearningConfig = coder.DeepLearningConfig('cudnn');
cfg.GenerateReport = true;
cfg.GenerateExampleMain = "GenerateCodeAndCompile";
cfg.TargetLang = 'C++';

```

```
inputs = {coder.Constant(videoFile),coder.Constant(laneNetFile), ...
    coder.Constant(laneCoeffMeans),coder.Constant(laneCoeffsStds)};
```

Run the codegen command.

```
codegen -args inputs -config cfg detectLanesInVideo
```

Code generation successful: [View report](#)

Generated Code Description

The series network is generated as a C++ class containing an array of 18 layer classes (after layer fusion optimization). The `setup()` method of the class sets up handles and allocates memory for each layer object. The `predict()` method invokes prediction for each of the 18 layers in the network.

```
class lanenet0_0 {
public:
    lanenet0_0();
    void setSize();
    void resetState();
    void setup();
    void predict();
    void cleanup();
    float *getLayerOutput(int layerIndex, int portIndex);
    int getLayerOutputSize(int layerIndex, int portIndex);
    float *getInputDataPointer(int b_index);
    float *getInputDataPointer();
    float *getOutputDataPointer(int b_index);
    float *getOutputDataPointer();
    int getBatchSize();
    ~lanenet0_0();

private:
    void allocate();
    void postsetup();
    void deallocate();

public:
    boolean_T isInitialized;
    boolean_T matlabCodegenIsDeleted;

private:
    int numLayers;
    MWTensorBase *inputTensors[1];
    MWTensorBase *outputTensors[1];
    MWCNNLayer *layers[18];
    MWCudnnTarget::MWTargetNetworkImpl *targetImpl;
};
```

The `cnn_lanenet*_conv*_w` and `cnn_lanenet*_conv*_b` files are the binary weights and bias file for convolution layer in the network. The `cnn_lanenet*_fc*_w` and `cnn_lanenet*_fc*_b` files are the binary weights and bias file for fully connected layer in the network.

```
codegendir = fullfile('codegen', 'exe', 'detectLanesInVideo');
dir([codegendir,filesep,'*.bin'])
```

```
cnn_lanenet0_0_conv1_b.bin
cnn_lanenet0_0_conv1_w.bin
```

```
cnn_lanenet0_0_conv3_b.bin
cnn_lanenet0_0_conv3_w.bin
```

```
cnn_lanenet0_0_conv5_b.bin
cnn_lanenet0_0_conv5_w.bin
```

```
cnn_lanenet0_0_conv2_b.bin  
cnn_lanenet0_0_conv2_w.bin
```

```
cnn_lanenet0_0_conv4_b.bin  
cnn_lanenet0_0_conv4_w.bin
```

```
cnn_lanenet0_0_data_offset.b  
cnn_lanenet0_0_data_scale.bi
```

Run the Executable

To run the executable, uncomment the following lines of code.

```
if ispc  
    [status,cmdout] = system("detectLanesInVideo.exe");  
else  
    [status,cmdout] = system("./detectLanesInVideo");  
end
```



See Also

Functions

[codegen](#) | [coder.DeepLearningConfig](#) | [coder.loadDeepLearningNetwork](#) | [coder.checkGpuInstall](#)

Objects

[coder.gpuConfig](#) | [coder.gpuEnvConfig](#) | [coder.CuDNNConfig](#) | [coder.TensorRTConfig](#)

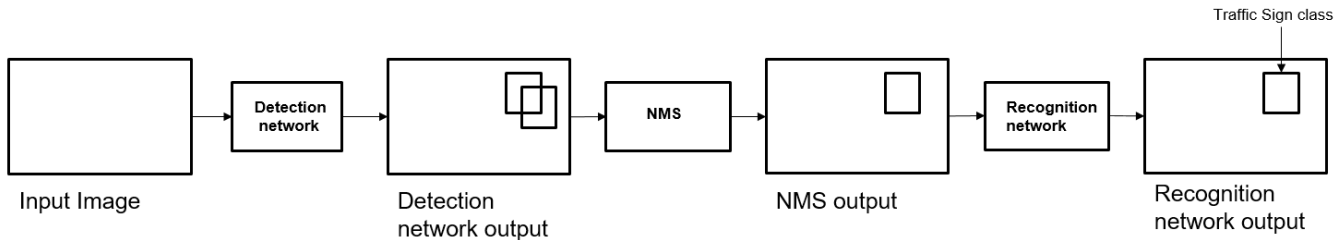
See Also

More About

- “Generated CNN Class Hierarchy” on page 4-65
- “Supported Networks, Layers, and Classes” on page 4-6
- “Deep Learning in MATLAB” (Deep Learning Toolbox)

Traffic Sign Detection and Recognition

This example shows how to generate CUDA® MEX code for a traffic sign detection and recognition application that uses deep learning. Traffic sign detection and recognition is an important application for driver assistance systems, aiding and providing information to the driver about road signs.



In this traffic sign detection and recognition example you perform three steps - detection, Non-Maximal Suppression (NMS), and recognition. First, the example detects the traffic signs on an input image by using an object detection network that is a variant of the You Only Look Once (YOLO) network. Then, overlapping detections are suppressed by using the NMS algorithm. Finally, the recognition network classifies the detected traffic signs.

Third-Party Prerequisites

Required

This example generates CUDA MEX and has the following third-party requirements.

- CUDA® enabled NVIDIA® GPU and compatible driver.

Optional

For non-MEX builds such as static, dynamic libraries or executables, this example has the following additional requirements.

- NVIDIA toolkit.
- NVIDIA cuDNN library.
- Environment variables for the compilers and libraries. For more information, see “Third-Party Hardware” and “Setting Up the Prerequisite Products”.

Verify GPU Environment

Use the `coder.checkGpuInstall` function to verify that the compilers and libraries necessary for running this example are set up correctly.

```

envCfg = coder.gpuEnvConfig('host');
envCfg.DeepLibTarget = 'cudnn';
envCfg.DeepCodegen = 1;
  
```

```
envCfg.Quiet = 1;
coder.checkGpuInstall(envCfg);
```

Detection and Recognition Networks

The detection network is trained in the Darknet framework and imported into MATLAB® for inference. Because the size of the traffic sign is relatively small with respect to that of the image and the number of training samples per class are fewer in the training data, all the traffic signs are considered as a single class for training the detection network.

The detection network divides the input image into a 7-by-7 grid. Each grid cell detects a traffic sign if the center of the traffic sign falls within the grid cell. Each cell predicts two bounding boxes and confidence scores for these bounding boxes. Confidence scores indicate whether the box contains an object or not. Each cell predicts on probability for finding the traffic sign in the grid cell. The final score is product of the preceding scores. You apply a threshold of 0.2 on this final score to select the detections.

The recognition network is trained on the same images by using MATLAB.

The trainRecognitionnet.m helper script shows the recognition network training.

Get the Pretrained Detector and Recognition Networks

This example uses the `yolo_tsr` and `RecognitionNet` MAT-files containing the pretrained networks. The files are approximately 6MB and 992MB in size, respectively. Download the files from the MathWorks website.

```
detectorNet = matlab.internal.examples.downloadSupportFile('gpucoder/cnn_models/traffic_sign_det
recognitionNet = matlab.internal.examples.downloadSupportFile('gpucoder/cnn_models/traffic_sign_
```

The detection network contains 58 layers including convolution, leaky ReLU, and fully connected layers.

```
load(detectorNet);
yolo

yolo =
  SeriesNetwork with properties:
    Layers: [58x1 nnet.cnn.layer.Layer]
    InputNames: {'input'}
    OutputNames: {'classoutput'}
```

To view the network architecture, use the `analyzeNetwork` (Deep Learning Toolbox) function.

```
analyzeNetwork(yolo)
```

The recognition network contains 14 layers including convolution, fully connected, and the classification output layers.

```
load(recognitionNet);
convnet

convnet =
  SeriesNetwork with properties:
```

```
Layers: [14x1 nnet.cnn.layer.Layer]
InputNames: {'imageinput'}
OutputNames: {'classoutput'}
```

The `tsdr_predict` Entry-Point Function

The `tsdr_predict.m` entry-point function takes an image input and detects the traffic signs in the image by using the detection network. The function suppresses the overlapping detections (NMS) by using `selectStrongestBbox` and recognizes the traffic sign by using the recognition network. The function loads the network objects from `yolo_tsr.mat` into a persistent variable `detectionnet` and the `RecognitionNet.mat` into a persistent variable `recognitionnet`. The function reuses the persistent objects on subsequent calls.

```
type('tsdr_predict.m')

function [selectedBbox,idx] = tsdr_predict(img,detectorMATFile,recogMATFile)
%#codegen

coder.gpu.kernelfun;

% resize the image
img_rz = imresize(img,[448,448]);

% Converting into BGR format
img_rz = img_rz(:,:,3:-1:1);
img_rz = im2single(img_rz);

%% TSD
persistent detectionnet;
if isempty(detectionnet)
    detectionnet = coder.loadDeepLearningNetwork(detectorMATFile,'Detection');
end

predictions = detectionnet.activations(img_rz,56,'OutputAs','channels');

%% Convert predictions to bounding box attributes
classes = 1;
num = 2;
side = 7;
thresh = 0.2;
[h,w,~] = size(img);

boxes = single(zeros(0,4));
probs = single(zeros(0,1));
for i = 0:(side*side)-1
    for n = 0:num-1
        p_index = side*side*classes + i*num + n + 1;
        scale = predictions(p_index);
        prob = zeros(1,classes+1);
        for j = 0:classes
            class_index = i*classes + 1;
            tempProb = scale*predictions(class_index+j);
            if tempProb > thresh

                row = floor(i / side);
```



```

        col = mod(i,side);

        box_index = side*side*(classes + num) + (i*num + n)*4 + 1;
        bxX = (predictions(box_index + 0) + col) / side;
        bxY = (predictions(box_index + 1) + row) / side;

        bxW = (predictions(box_index + 2)^2);
        bxH = (predictions(box_index + 3)^2);

        prob(j+1) = tempProb;
        probs = [probs;tempProb];

        boxX = (bxX-bxW/2)*w+1;
        boxY = (bxY-bxH/2)*h+1;
        boxW = bxW*w;
        boxH = bxH*h;
        boxes = [boxes; boxX,boxY,boxW,boxH];
    end
end
end
end

%% Run Non-Maximal Suppression on the detected bounding boxes
coder.varsize('selectedBbox',[98, 4],[1 0]);
[selectedBbox,~] = selectStrongestBbox(round(boxes),probs);

%% Recognition

persistent recognitionnet;
if isempty(recognitionnet)
    recognitionnet = coder.loadDeepLearningNetwork(recogMATFile,'Recognition');
end

idx = zeros(size(selectedBbox,1),1);
inpImg = coder.nullcopy(zeros(48,48,3,size(selectedBbox,1)));
for i = 1:size(selectedBbox,1)

    ymin = selectedBbox(i,2);
    ymax = ymin+selectedBbox(i,4);
    xmin = selectedBbox(i,1);
    xmax = xmin+selectedBbox(i,3);

    % Resize Image
    inpImg(:,:,i) = imresize(img(ymin:ymax,xmin:xmax,:),[48,48]);
end

for i = 1:size(selectedBbox,1)
    output = recognitionnet.predict(inpImg(:,:,i));
    [~,idx(i)]=max(output);
end

% Copyright 2017-2022 The MathWorks, Inc.

```

Generate CUDA MEX for the tsdr_predict Function

Create a GPU configuration object for a MEX target and set the target language to C++. Use the `coder.DeepLearningConfig` function to create a CuDNN deep learning configuration object and

assign it to the `DeepLearningConfig` property of the GPU code configuration object. To generate CUDA MEX, use the `codegen` command and specify the input to be of size `[480,704,3]`. This value corresponds to the input image size of the `tsdr_predict` function.

```
cfg = coder.gpuConfig('mex');  
cfg.TargetLang = 'C++';  
cfg.DeepLearningConfig = coder.DeepLearningConfig('cudnn');  
inputArgs = {ones(480,704,3,'uint8'),coder.Constant(detectorNet),...  
            coder.Constant(recognitionNet)};  
codegen -config cfg tsdr_predict -args inputArgs -report
```

Code generation successful: [View report](#)

To generate code by using TensorRT, pass `coder.DeepLearningConfig('tensorrt')` as an option to the coder configuration object instead of `'cudnn'`.

Run Generated MEX

Load an input image.

```
im = imread('stop.jpg');  
imshow(im);
```



Call `tsdr_predict_mex` on the input image.

```
im = imresize(im, [480,704]);  
[bboxes,classes] = tsdr_predict_mex(im,detectorNet,recognitionNet);
```

Map the class numbers to traffic sign names in the class dictionary.

```
classNames = {...
    'addedLane', 'slow', 'dip', 'speedLimit25', 'speedLimit35', 'speedLimit40', ...
    'speedLimit45', 'speedLimit50', 'speedLimit55', 'speedLimit65', ...
    'speedLimitUrdbl', 'doNotPass', 'intersection', 'keepRight', 'laneEnds', ...
    'merge', 'noLeftTurn', 'noRightTurn', 'stop', 'pedestrianCrossing', ...
    'stopAhead', 'rampSpeedAdvisory20', 'rampSpeedAdvisory45', ...
    'truckSpeedLimit55', 'rampSpeedAdvisory50', 'turnLeft', ...
    'rampSpeedAdvisoryUrdbl', 'turnRight', 'rightLaneMustTurn', 'yield', ...
    'yieldAhead', 'school', 'schoolSpeedLimit25', 'zoneAhead45', 'signalAhead'};
```

```
classRec = classNames(classes);
```

Display the detected traffic signs.

```
outputImage = insertShape(im, 'Rectangle', bboxes, 'LineWidth', 3);
```

```
for i = 1:size(bboxes,1)
    outputImage = insertText(outputImage, [bboxes(i,1)+ ...
        bboxes(i,3) bboxes(i,2)-20], classRec{i}, 'FontSize', 20, ...
        'TextColor', 'red');
end
```

```
imshow(outputImage);
```



Traffic Sign Detection and Recognition on a Video

The included helper file `tsdr_testVideo.m` grabs frames from the test video, performs traffic sign detection and recognition, and plots the results on each frame of the test video.

```
type tsdr_testVideo
```

```
function tsdr_testVideo

% Copyright 2017-2022 The MathWorks, Inc.

% Input video
v = VideoReader('stop.avi');

%% Generate Code for Traffic Sign Detection and Recognition
% Create a GPU Configuration object for MEX target setting target language
% to C++. Run the |codegen| command specifying an input of input video
% frame size. This corresponds to the input image size of tsdr_predict
% function.
cfg = coder.gpuConfig('mex');
cfg.TargetLang = 'C++';
cfg.DeepLearningConfig = coder.DeepLearningConfig('cudnn');
inputArgs = {ones(480,704,3,'uint8'),coder.constant(detectorNet),...
    coder.Constant(recognitionNet)};
codegen -config cfg tsdr_predict -args inputArgs -report

fps = 0;

while hasFrame(v)
    % Take a frame
    picture = readFrame(v);
    picture = imresize(picture,[480,704]);
    % Call MEX function for Traffic Sign Detection and Recognition
    tic;
    [bboxes,classes] = tsdr_predict_mex(picture,detectorNet,recognitionNet);
    newt = toc;

    % fps
    fps = .9*fps + .1*(1/newt);

    % display
        diplayDetections(picture,bboxes,classes,fps);
end

end

function diplayDetections(im,boundingBoxes,classIndices,fps)
% Function for inserting the detected bounding boxes and recognized classes
% and displaying the result
%
% Inputs :
%
% im          : Input test image
% boundingBoxes : Detected bounding boxes
% classIndices : Corresponding classes
```

```

%
% Traffic Signs (35)
classNames = {'addedLane','slow','dip','speedLimit25','speedLimit35',...
    'speedLimit40','speedLimit45','speedLimit50','speedLimit55',...
    'speedLimit65','speedLimitUrdbl','doNotPass','intersection',...
    'keepRight','laneEnds','merge','noLeftTurn','noRightTurn','stop',...
    'pedestrianCrossing','stopAhead','rampSpeedAdvisory20',...
    'rampSpeedAdvisory45','truckSpeedLimit55','rampSpeedAdvisory50',...
    'turnLeft','rampSpeedAdvisoryUrdbl','turnRight','rightLaneMustTurn',...
    'yield','yieldAhead','school','schoolSpeedLimit25','zoneAhead45',...
    'signalAhead'};

outputImage = insertShape(im,'Rectangle',boundingBoxes,'LineWidth',3);

for i = 1:size(boundingBoxes,1)

    ymin = boundingBoxes(i,2);
    xmin = boundingBoxes(i,1);
    xmax = xmin+boundingBoxes(i,3);

    % inserting class as text at YOLO detection
    classRec = classNames{classIndices(i)};
    outputImage = insertText(outputImage,[xmax ymin-20],classRec,...
        'FontSize',20,'TextColor','red');

end
outputImage = insertText(outputImage,...
    round(([size(outputImage,1) 40]/2)-20),...
    ['Frame Rate: ',num2str(fps)],'FontSize',20,'TextColor','red');
imshow(outputImage);
end

```

See Also

Functions

`coder.checkGpuInstall` | `codegen` | `coder.DeepLearningConfig` |
`coder.loadDeepLearningNetwork`

Objects

`coder.gpuConfig` | `coder.gpuEnvConfig` | `coder.CuDNNConfig` | `coder.TensorRTConfig`

More About

- “Supported Networks, Layers, and Classes” on page 4-6
- “Load Pretrained Networks for Code Generation” on page 4-66
- “Code Generation for Deep Learning Networks by Using cuDNN” on page 4-69
- “Code Generation for Deep Learning Networks by Using TensorRT” on page 4-78
- “Choose Function to Visualize Detected Objects” (Computer Vision Toolbox)

Logo Recognition Network

This example shows code generation for a logo classification application that uses deep learning. It uses a pretrained network called LogoNet and classifies an input image into 32 logo categories. This example also describes how to train the network by using preprocessed training data set. Finally, this example uses the `codegen` command to generate a MEX function and performs the prediction.

This example illustrates the following concepts:

- Preprocess the training images by extracting the logos and resizing to 227-by-227-by-3. Subsequently, use image augmentation to increase training data size.
- Train the network by using the stochastic gradient descent with momentum (SGDM) optimizer.
- Generate a CUDA® MEX and run the MEX.

Third-Party Prerequisites

Required

This example generates CUDA MEX and requires CUDA-enabled NVIDIA® GPU and compatible driver.

Optional

For non-MEX builds such as static, dynamic libraries or executables, this example has the following additional requirements.

- NVIDIA toolkit.
- NVIDIA cuDNN library.
- Environment variables for the compilers and libraries. For more information, see “Third-Party Hardware” and “Setting Up the Prerequisite Products”.

Verify GPU Environment

Use the `coder.checkGpuInstall` function to verify that the compilers and libraries for running this example are set up correctly.

```
envCfg = coder.gpuEnvConfig('host');  
envCfg.DeepLibTarget = 'cudnn';  
envCfg.DeepCodegen = 1;  
envCfg.Quiet = 1;  
coder.checkGpuInstall(envCfg);
```

Logo Recognition Network

Logos assist users in brand identification and recognition. Many companies incorporate their logos in advertising, documentation materials, and promotions. The logo recognition network was developed in MATLAB® and contains 22 layers. The network contains four sets of convolutional max pooling layers, three fully connected layers, and dropout layers that reduce computational expense. The network takes an input image of size 227-by-227-by-3 and classifies it into 32 logo categories. Because this network focuses on recognition, you can use it in applications where localization is not required. The network was trained in MATLAB by using the Flickr32Logos[1] and Flickr32 Plus[2] training data set. The two data sets contain around 200 images for each logo. The network was trained by using the stochastic gradient descent with momentum (SGDM) optimizer, a learning rate of

0.0001, 40 epochs, and a mini-batch size of 45. By default, the example uses a pretrained logo recognition network. The pretrained network enables you to run the entire example without having to wait for training to complete.

To train the network, set the `doTraining` variable in the following code to `true`. You must also download the Logos-32plus data set from Deep Learning for Logo Recognition and provide the location of the downloaded Logos-32plus_v1.0.1.zip file to `logozipPath`. The size of Logos-32plus data set is 1.95 GB. Depending on your internet connection, the download process can take time. The data set has 32 image subfolders containing a total of 7830 logo images from various brands. The groundtruth MAT-file provides the bounding box information of the logo in each image.

The `preprocessLogoData` function preprocesses the data for network training. The images in the Logos-32plus data set are of varying size. You must resize the images to input layer size of the network (227-by-227-by-3). The images also contain background information that you must remove. The `preprocessLogoData.m` performs these steps by using the bounding box information to extract the logos and creates a `imageDatastore` object that you can use for network training. The `trainLogonet` function creates logo recognition layers and trains the network by using specified training options. The network is trained using data that contains at least 110 images for each logo.

You can also increase the number of training samples by using data augmentation. Data augmentation helps prevent the network from overfitting and memorizing the exact details of the training images. To increase the training data, four types of data augmentation are provided: random flipping, Gaussian blur, shearing, and contrast normalization. To use data augmentation, set the `doAugmentation` variable in the following code to `true`.

```
doTraining = false;

if ~doTraining
    getLogonet;
else
    logozipPath = ''; % provide path of the downloaded zip file
    zipData = fullfile(logozipPath, 'Logos-32plus_v1.0.1.zip');
    unpackedData = fullfile(logozipPath, 'Logos32plus');

    if ~exist(unpackedData, 'dir')
        unzip(zipData, unpackedData);
    end

    doAugmentation = false;
    logoData = preprocessLogoData(unpackedData, doAugmentation);
    trainLogonet(logoData);
end

load('LogoNet.mat');
convnet

convnet =
    SeriesNetwork with properties:

        Layers: [22x1 nnet.cnn.layer.Layer]
        InputNames: {'imageinput'}
        OutputNames: {'classoutput'}
```

To view the network architecture, use the `analyzeNetwork` (Deep Learning Toolbox) function.

```
analyzeNetwork(convnet)
```

The `logonet_predict` Entry-Point Function

The `logonet_predict.m` entry-point function takes an image input and performs prediction on the image by using the deep learning network saved in the `LogoNet.mat` file. The function loads the network object from `LogoNet.mat` into a persistent variable `logonet` and reuses the persistent variable on subsequent prediction calls.

```
type('logonet_predict.m')

function out = logonet_predict(in)
    %#codegen

    % Copyright 2017-2022 The MathWorks, Inc.

    % A persistent object logonet is used to load the network object. At the
    % first call to this function, the persistent object is constructed and
    % setup. When the function is called subsequent times, the same object is
    % reused to call predict on inputs, thus avoiding reconstructing and
    % reloading the network object.
    persistent logonet;

    if isempty(logonet)

        logonet = coder.loadDeepLearningNetwork('LogoNet.mat','logonet');
    end

    out = logonet.predict(in);

end
```

Generate CUDA MEX for the `logonet_predict` Function

Create a GPU configuration object for a MEX target and set the target language to C++. Use the `coder.DeepLearningConfig` function to create a CuDNN deep learning configuration object. Assign it to the `DeepLearningConfig` property of the GPU code configuration object. To generate CUDA MEX, use the `codegen` command and specify the input to be of size `[227,227,3]`. This value corresponds to the input layer size of the `logonet` network.

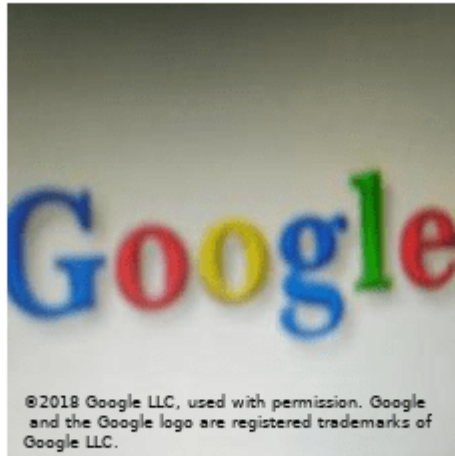
```
cfg = coder.gpuConfig('mex');
cfg.TargetLang = 'C++';
cfg.DeepLearningConfig = coder.DeepLearningConfig('cudnn');
codegen -config cfg logonet_predict -args {ones(227,227,3,'uint8')} -report
```

Code generation successful: [View report](#)

Run Generated MEX

Load an input image. Call `logonet_predict_mex` on the input image.

```
im = imread('test.png');
imshow(im);
```

```
im = imresize(im, [227,227]);
predict_scores = logonet_predict_mex(im);
```

Map the top five prediction scores to words in the Wordnet dictionary synset (logos).

```
synsetOut = convnet.Layers(end).Classes;

[val,indx] = sort(predict_scores, 'descend');
scores = val(1:5)*100;
top5labels = synsetOut(indx(1:5));
```

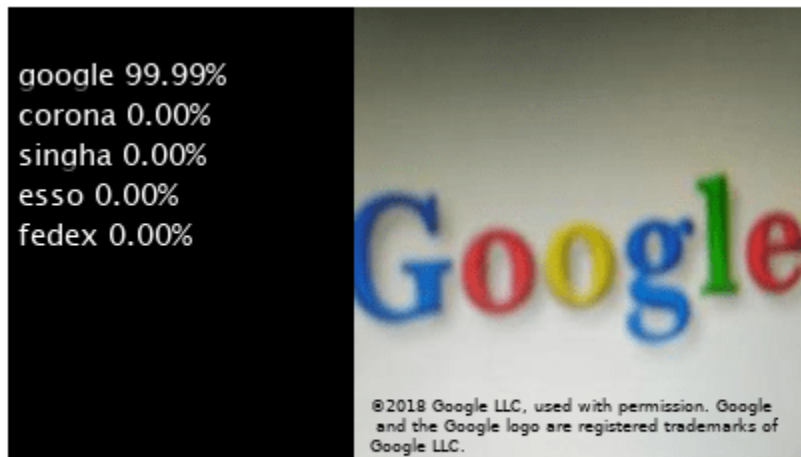
Display the top five classification labels.

```
outputImage = zeros(227,400,3, 'uint8');
for k = 1:3
    outputImage(:,174:end,k) = im(:, :, k);
end

scol = 1;
srow = 20;

for k = 1:5
    outputImage = insertText(outputImage, [scol, srow], ...
        [char(top5labels(k)), ' ', num2str(scores(k), '%2.2f'), '%'], ...
        'TextColor', 'w', 'FontSize', 15, 'BoxColor', 'black');
    srow = srow + 20;
end

imshow(outputImage);
```



Clear the static network object that was loaded in memory.

```
clear mex;
```

References

[1] Romberg, Stefan, Lluís Garcia Pueyo, Rainer Lienhart, and Roelof van Zwol. "Scalable Logo Recognition in Real-World Images." *ACM International Conference on Multimedia Retrieval 2011 (ICMR11)*: 1-8. <https://doi.org/10.1145/1991996.1992021>

[2] Bianco, Simone, Marco Buzzelli, Davide Mazzini, and Raimondo Schettini. "Deep Learning for Logo Recognition." *Neurocomputing* 245 (2017): 23-30. <https://doi.org/10.1016/j.neucom.2017.03.051>.

See Also

Functions

`coder.checkGpuInstall` | `codegen` | `coder.DeepLearningConfig` |
`coder.loadDeepLearningNetwork`

Objects

`coder.gpuConfig` | `coder.gpuEnvConfig` | `coder.CuDNNConfig` | `coder.TensorRTConfig`

More About

- "Code Generation for Deep Learning Networks by Using cuDNN" on page 4-69
- "Code Generation for Deep Learning Networks by Using TensorRT" on page 4-78

Deep Learning Prediction with NVIDIA TensorRT Library

This example shows how to generate code for a deep learning application by using the NVIDIA® TensorRT™ library. This example uses the `codegen` command to generate a MEX file that performs prediction with a Logo Recognition classification network by using TensorRT. The example also demonstrates how to use `codegen` command to generate a MEX file that performs 8-bit integer and 16-bit floating point prediction.

Third-Party Prerequisites

Required

This example generates CUDA® MEX and requires a CUDA-enabled NVIDIA GPU and compatible driver. You must have specific GPU compute capability for 8-bit integer and 16-bit floating point precision modes, see “Third-Party Hardware”.

Optional

For non-MEX builds such as static, dynamic libraries or executables, you must also have:

- NVIDIA toolkit.
- NVIDIA cuDNN and the TensorRT library.
- Environment variables for the compilers and libraries. For more information, see “Third-Party Hardware” and “Setting Up the Prerequisite Products”.

Verify GPU Environment

Use the `coder.checkGpuInstall` function to verify that the compilers and libraries necessary for running this example are set up correctly.

```
envCfg = coder.gpuEnvConfig('host');
envCfg.DeepLibTarget = 'tensorrt';
envCfg.DeepCodegen = 1;
envCfg.Quiet = 1;
coder.checkGpuInstall(envCfg);
```

Download and Load Pretrained Network

This example uses a pretrained logo recognition network to classify logos in images. Download the pretrained LogoNet network from MathWorks website and load the file. The network was developed in MATLAB and is approximately 42 MB in size. This network can recognize 32 logos under various lighting conditions and camera angles. For information on training the logo recognition network, see “Logo Recognition Network” on page 4-140.

```
net = getLogonet;
```

The `logonet_predict` Entry-Point Function

The `logonet_predict.m` entry-point function takes an image input and performs prediction on the image by using the deep learning network saved in the `LogoNet.mat` file. The function loads the network object from `LogoNet.mat` into a persistent variable `logonet` and reuses the persistent variable during subsequent prediction calls.

```
type('logonet_predict.m')
```

```
function out = logonet_predict(in)
%#codegen

% Copyright 2017-2022 The MathWorks, Inc.

% A persistent object logonet is used to load the network object. At the
% first call to this function, the persistent object is constructed and
% setup. When the function is called subsequent times, the same object is
% reused to call predict on inputs, thus avoiding reconstructing and
% reloading the network object.
persistent logonet;

if isempty(logonet)

    logonet = coder.loadDeepLearningNetwork('LogoNet.mat','logonet');

end

out = logonet.predict(in);

end
```

Run MEX Code Generation

To generate CUDA code for the `logonet_predict` entry-point function, create a GPU code configuration object for a MEX target and set the target language to C++. Use the `coder.DeepLearningConfig` function to create a TensorRT deep learning configuration object and assign it to the `DeepLearningConfig` property of the GPU code configuration object. Run the `codegen` command by specifying an input size of 227-by-227-by-3. This value corresponds to the input layer size of the Logo Recognition network. By default, generating TensorRT code runs inference in 32-bit floats.

```
cfg = coder.gpuConfig('mex');
cfg.TargetLang = 'C++';
cfg.DeepLearningConfig = coder.DeepLearningConfig('tensorrt');
codegen -config cfg logonet_predict -args {coder.typeof(single(0),[227 227 3])} -report
```

Code generation successful: [View report](#)

Perform Prediction on Test Image

Load an input image. Call `logonet_predict_mex` on the input image.

```
im = imread('gpuCoder_tensorrt_test.png');
im = imresize(im, [227,227]);
predict_scores = logonet_predict_mex(single(im));

% get top 5 probability scores and their labels
[val,indx] = sort(predict_scores, 'descend');
scores = val(1:5)*100;
classnames = net.Layers(end).ClassNames;
top5labels = classnames(indx(1:5));
```

Display the top five classification labels.

```
outputImage = zeros(227,400,3, 'uint8');
for k = 1:3
    outputImage(:,174:end,k) = im(:, :, k);
end
```

```

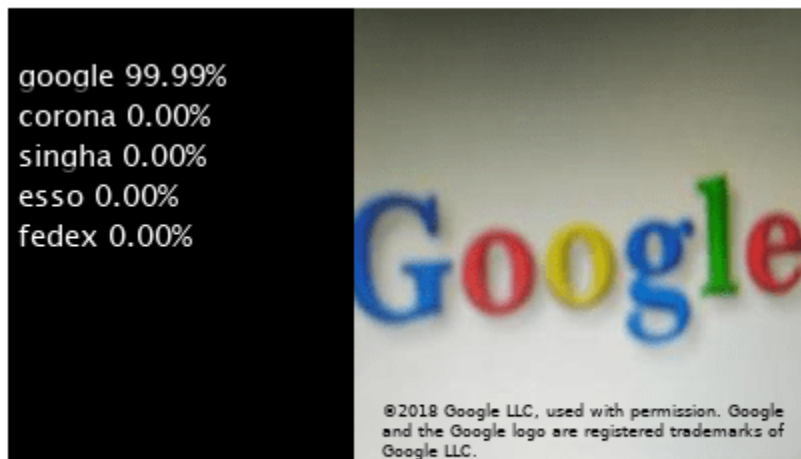
end

scol = 1;
srow = 20;

for k = 1:5
    outputImage = insertText(outputImage, [scol, srow],...
        [char(top5labels(k)), ' ', num2str(scores(k), '%2.2f'), '%'],...
        'TextColor', 'w', 'FontSize', 15, 'BoxColor', 'black');
    srow = srow + 20;
end

imshow(outputImage);

```



Free the GPU memory by removing the loaded MEX function.

```
clear mex;
```

Generate TensorRT Code for 8-Bit Integer Prediction

Generate TensorRT code that runs inference in int8 precision.

Code generation by using the NVIDIA TensorRT Library with inference computation in 8-bit integer precision supports these additional networks:

- Object detector networks, such as YOLOv2 and SSD
- Regression and semantic segmentation networks

TensorRT requires a calibration data set to calibrate a network that is trained in floating-point to compute inference in 8-bit integer precision. Set the data type to `int8` and the path to the calibration data set by using the `DeepLearningConfig`. `logos_dataset` is a subfolder that contains images grouped by their classification labels. For `int8` support, the GPU compute capability must be 6.1, 7.0, or higher.

Note that for semantic segmentation networks, the calibration data images must be of a format supported by the `imread` function.

```
unzip('logos_dataset.zip');
cfg = coder.gpuConfig('mex');
cfg.TargetLang = 'C++';
cfg.GpuConfig.ComputeCapability = '6.1';
cfg.DeepLearningConfig = coder.DeepLearningConfig('tensorrt');
cfg.DeepLearningConfig.DataType = 'int8';
cfg.DeepLearningConfig.DataPath = 'logos_dataset';
cfg.DeepLearningConfig.NumCalibrationBatches = 50;
codegen -config cfg logonet_predict -args {coder.typeof(int8(0),[227 227 3])} -report
```

Code generation successful: [View report](#)

Run INT8 Prediction on Test Image

Load an input image. Call `logonet_predict_mex` on the input image.

```
im = imread('gpucoder_tensorrt_test.png');
im = imresize(im, [227,227]);
predict_scores = logonet_predict_mex(int8(im));

% get top 5 probability scores and their labels
[val,indx] = sort(predict_scores, 'descend');
scores = val(1:5)*100;
classnames = net.Layers(end).ClassNames;
top5labels = classnames(indx(1:5));
```

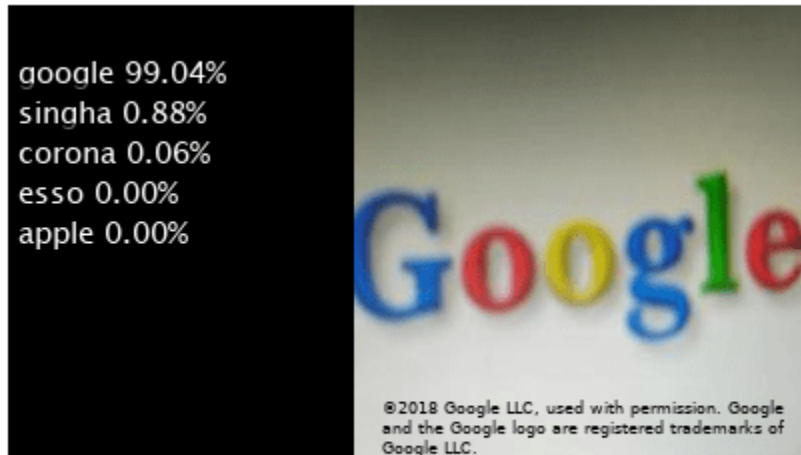
Display the top five classification labels.

```
outputImage = zeros(227,400,3, 'uint8');
for k = 1:3
    outputImage(:,174:end,k) = im(:, :, k);
end

scol = 1;
srow = 20;

for k = 1:5
    outputImage = insertText(outputImage, [scol, srow],...
        [char(top5labels(k)), ' ', num2str(scores(k), '%2.2f'), '%'],...
        'TextColor', 'w', 'FontSize', 15, 'BoxColor', 'black');
    srow = srow + 20;
end

imshow(outputImage);
```



Free the GPU memory by removing the loaded MEX function.

```
clear mex;
```

Generate TensorRT Code for 16-bit Floating Point Prediction

Generate TensorRT code that runs inference in fp16 precision. For fp16 support, the GPU compute capability must be 5.3, 6.0, 6.2 or higher.

Note that quantization error occurs when accumulating operations in single precision and converting them to half precision. For more information, see “Quantization of Deep Neural Networks” on page 4-99.

```
cfg = coder.gpuConfig('mex');
cfg.TargetLang = 'C++';
cfg.GpuConfig.ComputeCapability = '5.3';
cfg.DeepLearningConfig = coder.DeepLearningConfig('tensorrt');
cfg.DeepLearningConfig.DataType = 'fp16';
codegen -config cfg logonet_predict -args {coder.typeof(half(0),[227 227 3])} -report
```

Code generation successful: [View report](#)

Run FP16 Prediction on Test Image

Load an input image. Call `logonet_predict_mex` on the input image.

```
im = imread('gpcoder_tensorrt_test.png');

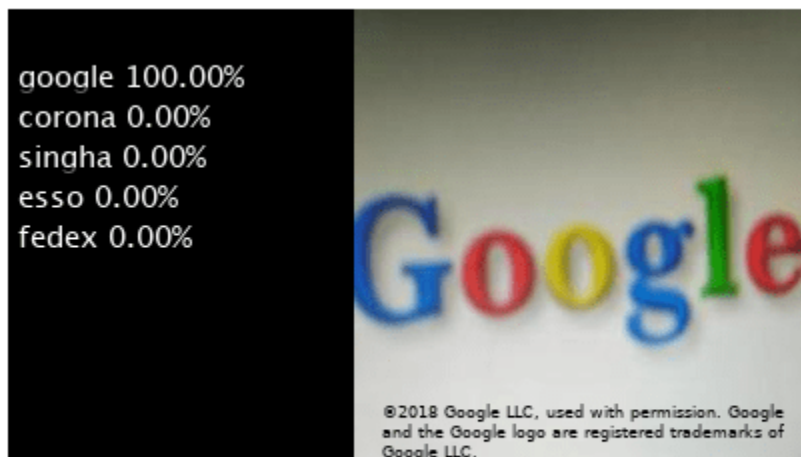
im = imresize(im, [227,227]);
predict_scores = logonet_predict_mex(half(im));

% get top 5 probability scores and their labels
[val,indx] = sort(predict_scores, 'descend');
scores = val(1:5)*100;
```

```
classnames = net.Layers(end).ClassNames;  
top5labels = classnames(indx(1:5));
```

Display the top five classification labels.

```
outputImage = zeros(227,400,3, 'uint8');  
for k = 1:5  
    outputImage(:,174:end,k) = im(:,:,k);  
end  
  
scol = 1;  
srow = 20;  
  
for k = 1:5  
    outputImage = insertText(outputImage, [scol, srow],...  
        [char(top5labels(k)), ' ', num2str(scores(k), '%2.2f'), '%'],...  
        'TextColor', 'w', 'FontSize', 15, 'BoxColor', 'black');  
    srow = srow + 20;  
end  
  
imshow(outputImage);
```



Free the GPU memory by removing the loaded MEX function.

```
clear mex;
```

See Also

Functions

`coder.checkGpuInstall` | `codegen` | `coder.DeepLearningConfig` |
`coder.loadDeepLearningNetwork`

Objects

`coder.gpuConfig` | `coder.CodeConfig` | `coder.EmbeddedCodeConfig` | `coder.gpuEnvConfig`
| `coder.TensorRTConfig`

See Also

More About

- “Supported Networks, Layers, and Classes” on page 4-6
- “Load Pretrained Networks for Code Generation” on page 4-66
- “Code Generation for Deep Learning Networks by Using TensorRT” on page 4-78

Code Generation for Semantic Segmentation Network

This example shows code generation for an image segmentation application that uses deep learning. It uses the `codegen` command to generate a MEX function that performs prediction on a DAG Network object for SegNet [1], a deep learning network for image segmentation.

Third-Party Prerequisites

Required

This example generates CUDA MEX and has the following third-party requirements.

- CUDA® enabled NVIDIA® GPU and compatible driver.

Optional

For non-MEX builds such as static, dynamic libraries or executables, this example has the following additional requirements.

- NVIDIA toolkit.
- NVIDIA cuDNN library.
- Environment variables for the compilers and libraries. For more information, see “Third-Party Hardware” and “Setting Up the Prerequisite Products”.

Verify GPU Environment

Use the `coder.checkGpuInstall` function to verify that the compilers and libraries necessary for running this example are set up correctly.

```
envCfg = coder.gpuEnvConfig('host');  
envCfg.DeepLibTarget = 'cudnn';  
envCfg.DeepCodegen = 1;  
envCfg.Quiet = 1;  
coder.checkGpuInstall(envCfg);
```

Segmentation Network

SegNet [1] is a type of convolutional neural network (CNN) designed for semantic image segmentation. It is a deep encoder-decoder multi-class pixel-wise segmentation network trained on the CamVid [2] dataset and imported into MATLAB® for inference. The SegNet [1] is trained to segment pixels belonging to 11 classes that include Sky, Building, Pole, Road, Pavement, Tree, SignSymbol, Fence, Car, Pedestrian, and Bicyclist.

For information regarding training a semantic segmentation network in MATLAB by using the CamVid [2] dataset, see “Semantic Segmentation Using Deep Learning” (Computer Vision Toolbox).

The `segnet_predict` Entry-Point Function

The `segnet_predict.m` entry-point function takes an image input and performs prediction on the image by using the deep learning network saved in the `SegNet.mat` file. The function loads the network object from the `SegNet.mat` file into a persistent variable `myNet` and reuses the persistent variable on subsequent prediction calls.

```
type('segnet_predict.m')
```

```
function out = segnet_predict(in)
%#codegen
% Copyright 2018-2021 The MathWorks, Inc.

persistent mynet;

if isempty(mynet)
    mynet = coder.loadDeepLearningNetwork('SegNet.mat');
end

% pass in input
out = predict(mynet,in);
```

Get Pretrained SegNet DAG Network Object

```
net = getSegNet();
```

The DAG network contains 91 layers including convolution, batch normalization, pooling, unpooling, and the pixel classification output layers. Use the `analyzeNetwork` (Deep Learning Toolbox) function to display an interactive visualization of the deep learning network architecture.

```
analyzeNetwork(net);
```

Run MEX Code Generation

To generate CUDA code for the `segnet_predict.m` entry-point function, create a GPU code configuration object for a MEX target and set the target language to C++. Use the `coder.DeepLearningConfig` function to create a CuDNN deep learning configuration object and assign it to the `DeepLearningConfig` property of the GPU code configuration object. Run the `codegen` command specifying an input size of `[360,480,3]`. This value corresponds to the input layer size of SegNet.

```
cfg = coder.gpuConfig('mex');
cfg.TargetLang = 'C++';
cfg.DeepLearningConfig = coder.DeepLearningConfig('cudnn');
codegen -config cfg segnet_predict -args {ones(360,480,3,'uint8')} -report
```

Code generation successful: [View report](#)

Run Generated MEX

Load and display an input image. Call `segnet_predict_mex` on the input image.

```
im = imread('gpcoder_segnet_image.png');
imshow(im);
```



```
predict_scores = segnet_predict_mex(im);
```

The *predict_scores* variable is a three-dimensional matrix that has 11 channels corresponding to the pixel-wise prediction scores for every class. Compute the channel by using the maximum prediction score to get pixel-wise labels.

```
[~,argmax] = max(predict_scores,[],3);
```

Overlay the segmented labels on the input image and display the segmented region.

```
classes = [  
    "Sky"  
    "Building"  
    "Pole"  
    "Road"  
    "Pavement"  
    "Tree"  
    "SignSymbol"  
    "Fence"  
    "Car"  
    "Pedestrian"  
    "Bicyclist"  
];
```

```
cmap = camvidColorMap();
```

```
SegmentedImage = labeloverlay(im, argmax, 'ColorMap', cmap);  
figure  
imshow(SegmentedImage);  
pixelLabelColorbar(cmap, classes);
```



References

[1] Badrinarayanan, Vijay, Alex Kendall, and Roberto Cipolla. "SegNet: A Deep Convolutional Encoder-Decoder Architecture for Image Segmentation." *arXiv preprint arXiv:1511.00561*, 2015.

[2] Brostow, Gabriel J., Julien Fauqueur, and Roberto Cipolla. "Semantic object classes in video: A high-definition ground truth database." *Pattern Recognition Letters* Vol 30, Issue 2, 2009, pp 88-97.

See Also

Functions

`coder.checkGpuInstall` | `codegen` | `coder.DeepLearningConfig` |
`coder.loadDeepLearningNetwork`

Objects

`coder.gpuConfig` | `coder.gpuEnvConfig` | `coder.CuDNNConfig`

Related Examples

- “Semantic Segmentation Using Deep Learning” (Computer Vision Toolbox)
- “Semantic Segmentation on NVIDIA DRIVE” (MATLAB Coder Support Package for NVIDIA Jetson and NVIDIA DRIVE Platforms)
- “Train and Deploy Fully Convolutional Networks for Semantic Segmentation” on page 4-157
- “Code Generation for Semantic Segmentation Network That Uses U-net” on page 4-169

More About

- “Getting Started with Semantic Segmentation Using Deep Learning” (Computer Vision Toolbox)

Train and Deploy Fully Convolutional Networks for Semantic Segmentation

This example shows how to train and deploy a fully convolutional semantic segmentation network on an NVIDIA® GPU by using GPU Coder™.

A semantic segmentation network classifies every pixel in an image, resulting in an image that is segmented by class. Applications for semantic segmentation include road segmentation for autonomous driving and cancer cell segmentation for medical diagnosis. To learn more, see “Getting Started with Semantic Segmentation Using Deep Learning” (Computer Vision Toolbox).

To illustrate the training procedure, this example trains FCN-8s [1], one type of convolutional neural network (CNN) designed for semantic image segmentation. Other types of networks for semantic segmentation include fully convolutional networks, such as SegNet and U-Net. You can apply this training procedure to those networks too.

This example uses the CamVid dataset [2] from the University of Cambridge for training. This data set is a collection of images containing street-level views obtained while driving. The data set provides pixel-level labels for 32 semantic classes including car, pedestrian, and road.

Third-party Prerequisites

Required

- CUDA® enabled NVIDIA GPU and compatible driver.

Optional

- NVIDIA CUDA toolkit.
- NVIDIA cuDNN library.
- Environment variables for the compilers and libraries. For information on the supported versions of the compilers and libraries, see “Third-Party Hardware”. For setting up the environment variables, see “Setting Up the Prerequisite Products”.

Verify GPU Environment

Use the `coder.checkGpuInstall` function to verify that the compilers and libraries necessary for running this example are set up correctly.

```
envCfg = coder.gpuEnvConfig('host');  
envCfg.DeepLibTarget = 'cudnn';  
envCfg.DeepCodegen = 1;  
envCfg.Quiet = 1;  
coder.checkGpuInstall(envCfg);
```

Setup

This example creates the fully convolutional semantic segmentation network with weights initialized from the VGG-16 network. The `vgg16` function checks for the existence of the Deep Learning Toolbox Model for VGG-16 Network support package and returns a pretrained VGG-16 model.

```
vgg16();
```

Download a pretrained version of FCN. This pretrained model enables you to run the entire example without waiting for the training to complete. The `doTraining` flag controls whether the example uses the trained network of the example or the pretrained FCN network for code generation.

```
doTraining = false;
if ~doTraining
    pretrainedURL = 'https://www.mathworks.com/supportfiles/gpuCoder/cnn_models/fcn/FCN8sCamVid.r
    disp('Downloading pretrained FCN (448 MB)...');
    websave('FCN8sCamVid.mat',pretrainedURL);
end
```

Downloading pretrained FCN (448 MB)...

Download CamVid Dataset

Download the CamVid dataset from these URLs.

```
imageURL = 'http://web4.cs.ucl.ac.uk/staff/g.brostow/MotionSegRecData/files/701_StillsRaw_full.z
labelURL = 'http://web4.cs.ucl.ac.uk/staff/g.brostow/MotionSegRecData/data/LabeledApproved_full.
```

```
outputFolder = fullfile(pwd,'CamVid');
```

```
if ~exist(outputFolder, 'dir')

    mkdir(outputFolder)
    labelsZip = fullfile(outputFolder,'labels.zip');
    imagesZip = fullfile(outputFolder,'images.zip');

    disp('Downloading 16 MB CamVid dataset labels...');
    websave(labelsZip, labelURL);
    unzip(labelsZip, fullfile(outputFolder,'labels'));

    disp('Downloading 557 MB CamVid dataset images...');
    websave(imagesZip, imageURL);
    unzip(imagesZip, fullfile(outputFolder,'images'));
end
```

The data download time depends on your Internet connection. The example execution does not proceed until the download operation is complete. Alternatively, use your web browser to first download the data set to your local disk. Then, use the `outputFolder` variable to point to the location of the downloaded file.

Load CamVid Images

Use `imageDatastore` to load CamVid images. The `imageDatastore` enables you to efficiently load a large collection of images onto a disk.

```
imgDir = fullfile(outputFolder,'images','701_StillsRaw_full');
imds = imageDatastore(imgDir);
```

Display one of the images.

```
I = readimage(imds,25);
I = histeq(I);
imshow(I)
```




Load CamVid Pixel-Labeled Images

Use `pixelLabelDatastore` (Computer Vision Toolbox) to load CamVid pixel label image data. A `pixelLabelDatastore` encapsulates the pixel label data and the label ID to a class name mapping.

Following the training method described in the SegNet paper [3], group the 32 original classes in CamVid to 11 classes. Specify these classes.

```
classes = [  
    "Sky"  
    "Building"  
    "Pole"  
    "Road"  
    "Pavement"  
    "Tree"  
    "SignSymbol"  
    "Fence"  
    "Car"  
    "Pedestrian"  
    "Bicyclist"  
];
```

To reduce 32 classes into 11 classes, multiple classes from the original data set are grouped together. For example, "Car" is a combination of "Car", "SUVPickupTruck", "Truck_Bus", "Train", and "OtherMoving". Return the grouped label IDs by using the `camvidPixelLabelIDs` supporting function.

```
labelIDs = camvidPixelLabelIDs();
```

Use the classes and label IDs to create the `pixelLabelDatastore`.

```
labelDir = fullfile(outputFolder, 'labels');
pxds = pixelLabelDatastore(labelDir, classes, labelIDs);
```

Read and display one of the pixel-labeled images by overlaying it on top of an image.

```
C = readimage(pxds, 25);
cmap = camvidColorMap;
B = labeloverlay(I, C, 'ColorMap', cmap);
imshow(B)
pixelLabelColorbar(cmap, classes);
```



Areas with no color overlay do not have pixel labels and are not used during training.

Analyze Data Set Statistics

To see the distribution of class labels in the CamVid dataset, use `countEachLabel` (Computer Vision Toolbox). This function counts the number of pixels by class label.

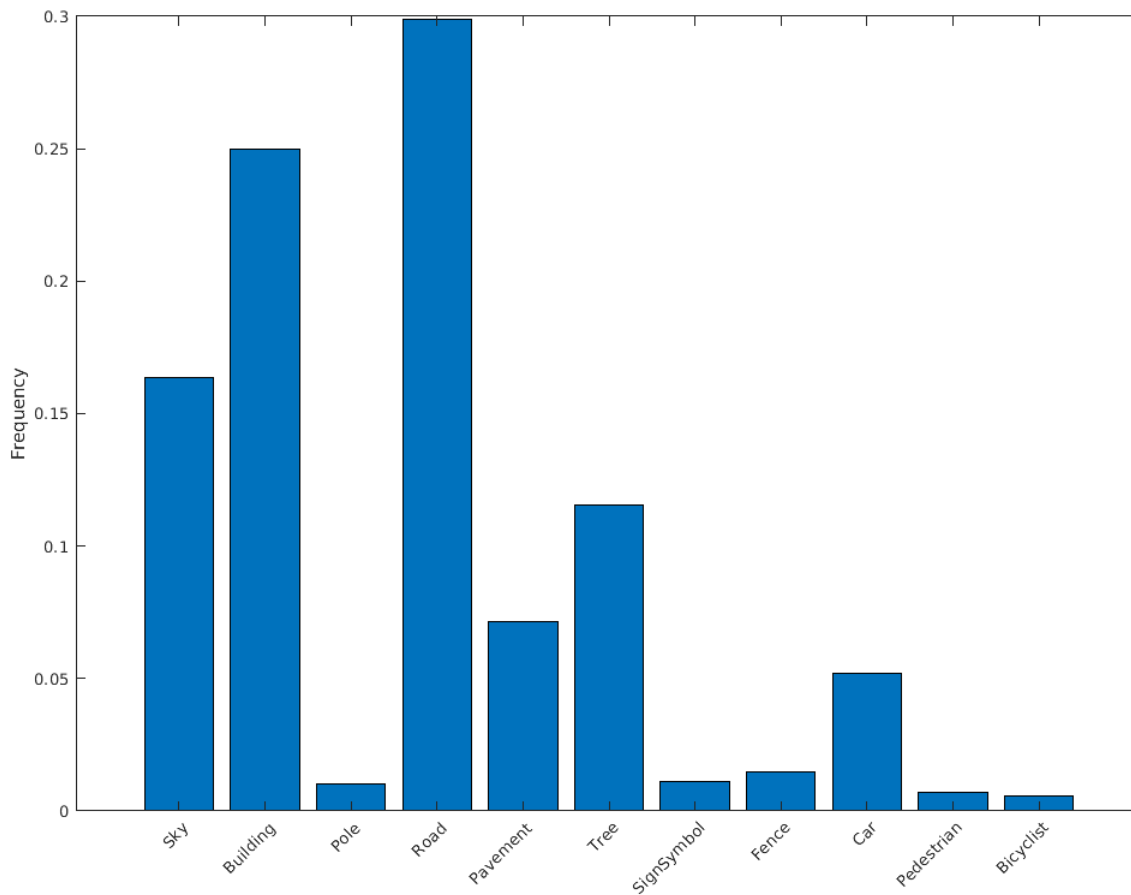
```
tbl = countEachLabel(pxds)
```

```
tbl=11x3 table
      Name      PixelCount      ImagePixelCount
      _____  _____  _____
      {'Sky'      }  7.6801e+07      4.8315e+08
      {'Building' }  1.1737e+08      4.8315e+08
      {'Pole'     }  4.7987e+06      4.8315e+08
      {'Road'     }  1.4054e+08      4.8453e+08
      {'Pavement' }  3.3614e+07      4.7209e+08
      {'Tree'     }  5.4259e+07      4.479e+08
      {'SignSymbol'}  5.2242e+06      4.6863e+08
      {'Fence'    }  6.9211e+06      2.516e+08
      {'Car'      }  2.4437e+07      4.8315e+08
      {'Pedestrian'}  3.4029e+06      4.4444e+08
      {'Bicyclist'}  2.5912e+06      2.6196e+08
```

Visualize the pixel counts by class.

```
frequency = tbl.PixelCount/sum(tbl.PixelCount);

bar(1:numel(classes),frequency)
xticks(1:numel(classes))
xticklabels(tbl.Name)
xtickangle(45)
ylabel('Frequency')
```



Ideally, all classes have an equal number of observations. The classes in CamVid are imbalanced, which is a common issue in automotive data sets of street scenes. Such scenes have more sky, building, and road pixels than pedestrian and bicyclist pixels because sky, buildings, and roads cover more area in the image. If not handled correctly, this imbalance can be detrimental to the learning process because the learning is biased in favor of the dominant classes. Later on in this example, you use class weighting to handle this issue.

Resize CamVid Data

The images in the CamVid data set are 720-by-960. To reduce training time and memory usage, resize the images and pixel label images to 360-by-480 by using the `resizeCamVidImages` and `resizeCamVidPixelLabels` supporting functions.

```
imageFolder = fullfile(outputFolder, 'imagesResized', filesep);  
imds = resizeCamVidImages(imds, imageFolder);  
  
labelFolder = fullfile(outputFolder, 'labelsResized', filesep);  
pxds = resizeCamVidPixelLabels(pxds, labelFolder);
```

Prepare Training and Test Sets

SegNet is trained by using 60% of the images from the dataset. The rest of the images are used for testing. The following code randomly splits the image and pixel label data into a training set and a test set.

```
[imdsTrain,imdsTest,pxdsTrain,pxdsTest] = partitionCamVidData(imds,pxds);
```

The 60/40 split results in the following number of training and test images:

```
numTrainingImages = numel(imdsTrain.Files)
```

```
numTrainingImages = 421
```

```
numTestingImages = numel(imdsTest.Files)
```

```
numTestingImages = 280
```

Create Network

Use `fcnLayers` (Computer Vision Toolbox) to create fully convolutional network layers initialized by using VGG-16 weights. The `fcnLayers` function performs the network transformations to transfer the weights from VGG-16 and adds the additional layers required for semantic segmentation. The output of the `fcnLayers` function is a `LayerGraph` object representing FCN. A `LayerGraph` object encapsulates the network layers and the connections between the layers.

```
imageSize = [360 480];
numClasses = numel(classes);
lgraph = fcnLayers(imageSize,numClasses);
```

The image size is selected based on the size of the images in the dataset. The number of classes is selected based on the classes in CamVid.

Balance Classes by Using Class Weighting

The classes in CamVid are not balanced. To improve training, you can use the pixel label counts computed earlier by the `countEachLabel` (Computer Vision Toolbox) function and calculate the median frequency class weights [3].

```
imageFreq = tbl.PixelCount ./ tbl.ImagePixelCount;
classWeights = median(imageFreq) ./ imageFreq;
```

Specify the class weights by using a `pixelClassificationLayer` (Computer Vision Toolbox).

```
pxLayer = pixelClassificationLayer('Name','labels','Classes',tbl.Name,'ClassWeights',classWeights);
```

```
pxLayer =
  PixelClassificationLayer with properties:
```

```
    Name: 'labels'
  Classes: [11x1 categorical]
ClassWeights: [11x1 double]
  OutputSize: 'auto'
```

```
Hyperparameters
  LossFunction: 'crossentropyex'
```

Update the SegNet network that has the new pixelClassificationLayer by removing the current pixelClassificationLayer and adding the new layer. The current pixelClassificationLayer is named 'pixelLabels'. Remove it by using the removeLayers (Deep Learning Toolbox) function, add the new one by using the addLayers (Deep Learning Toolbox) function, and connect the new layer to the rest of the network by using the connectLayers (Deep Learning Toolbox) function.

```
lgraph = removeLayers(lgraph, 'pixelLabels');  
lgraph = addLayers(lgraph, pxLayer);  
lgraph = connectLayers(lgraph, 'softmax', 'labels');
```

Select Training Options

The optimization algorithm for training is Adam, which is derived from *adaptive moment estimation*. Use the trainingOptions (Deep Learning Toolbox) function to specify the hyperparameters used for Adam.

```
options = trainingOptions('adam', ...  
    'InitialLearnRate', 1e-3, ...  
    'MaxEpochs', 100, ...  
    'MiniBatchSize', 4, ...  
    'Shuffle', 'every-epoch', ...  
    'CheckpointPath', tempdir, ...  
    'VerboseFrequency', 2);
```

A 'MiniBatchSize' of four reduces memory usage while training. You can increase or decrease this value based on the amount of GPU memory in your system.

'CheckpointPath' is set to a temporary location. This name-value pair enables the saving of network checkpoints at the end of every training epoch. If training is interrupted due to a system failure or power outage, you can resume training from the saved checkpoint. Make sure that the location specified by 'CheckpointPath' has enough space to store the network checkpoints.

Data Augmentation

Data augmentation is used to improve network accuracy by randomly transforming the original data during training. By using data augmentation, you can add more variety to the training data without increasing the number of labeled training samples. To apply the same random transformation to both image and pixel label data use datastore combine and transform. First, combine imdsTrain and pxdsTrain.

```
dsTrain = combine(imdsTrain, pxdsTrain);
```

Next, use datastore transform to apply the desired data augmentation defined in the supporting function augmentImageAndLabel. Here, random left/right reflection and random X/Y translation of +/- 10 pixels is used for data augmentation.

```
xTrans = [-10 10];  
yTrans = [-10 10];  
dsTrain = transform(dsTrain, @(data) augmentImageAndLabel(data, xTrans, yTrans));
```

Note that data augmentation is not applied to the test and validation data. Ideally, test and validation data should be representative of the original data and is left unmodified for unbiased evaluation.

Start Training

Start training using trainNetwork if the doTraining flag is true. Otherwise, load a pretrained network.

The training was verified on an NVIDIA™ Titan Xp with 12 GB of GPU memory. If your GPU has less memory, you might run out of memory. If you do not have enough memory in your system, try lowering the `MiniBatchSize` property in `trainingOptions` to 1. Training this network takes about 5 hours or longer depending on your GPU hardware.

```
doTraining = false;
if doTraining
    [net, info] = trainNetwork(dsTrain,lgraph,options);
    save('FCN8sCamVid.mat','net');
end
```

Save the DAG network object as a MAT-file named `FCN8sCamVid.mat`. This MAT-file is used during code generation.

Perform MEX Code-generation

The `fcn_predict.m` function takes an image input and performs prediction on the image by using the deep learning network saved in `FCN8sCamVid.mat` file. The function loads the network object from `FCN8sCamVid.mat` into a persistent variable `mynet` and reuses the persistent object on subsequent prediction calls.

```
type('fcn_predict.m')

function out = fcn_predict(in)
%#codegen
% Copyright 2018-2019 The MathWorks, Inc.

persistent mynet;

if isempty(mynet)
    mynet = coder.loadDeepLearningNetwork('FCN8sCamVid.mat');
end

% pass in input
out = predict(mynet,in);
```

Generate a GPU Configuration object for MEX target setting target language to C++. Use the `coder.DeepLearningConfig` function to create a cuDNN deep learning configuration object and assign it to the `DeepLearningConfig` property of the GPU code configuration object. Run the `codegen` command specifying an input size [360, 480, 3]. This size corresponds to the input layer of FCN.

```
cfg = coder.gpuConfig('mex');
cfg.TargetLang = 'C++';
cfg.DeepLearningConfig = coder.DeepLearningConfig('cudnn');
codegen -config cfg fcn_predict -args {ones(360,480,3,'uint8')} -report
```

Code generation successful: [View report](#)

Run Generated MEX

Load and display an input image.

```
im = imread('testImage.png');
imshow(im);
```



Run prediction by calling `fcn_predict_mex` on the input image.

```
predict_scores = fcn_predict_mex(im);
```

The `predict_scores` variable is a three-dimensional matrix having 11 channels corresponding to the pixel-wise prediction scores for every class. Compute the channel by using the maximum prediction score to get pixel-wise labels.

```
[~,argmax] = max(predict_scores,[],3);
```

Overlay the segmented labels on the input image and display the segmented region.

```
classes = [  
    "Sky"  
    "Building"  
    "Pole"  
    "Road"  
    "Pavement"  
    "Tree"  
    "SignSymbol"  
    "Fence"  
    "Car"  
    "Pedestrian"  
    "Bicyclist"  
];
```



```

cmap = camvidColorMap();
SegmentedImage = labeloverlay(im, argmax, 'ColorMap', cmap);
figure
imshow(SegmentedImage);
pixelLabelColorbar(cmap, classes);

```



Cleanup

Clear the static network object that was loaded in memory.

```
clear mex;
```

Supporting Functions

```

function data = augmentImageAndLabel(data, xTrans, yTrans)
% Augment images and pixel label images using random reflection and
% translation.

for i = 1:size(data,1)

    tform = randomAffine2d(...
        'XReflection', true, ...
        'XTranslation', xTrans, ...
        'YTranslation', yTrans);

```

```
% Center the view at the center of image in the output space while
% allowing translation to move the output image out of view.
rout = affineOutputView(size(data{i,1}), tform, 'BoundsStyle', 'centerOutput');

% Warp the image and pixel labels using the same transform.
data{i,1} = imwarp(data{i,1}, tform, 'OutputView', rout);
data{i,2} = imwarp(data{i,2}, tform, 'OutputView', rout);

end
end
```

References

- [1] Long, J., E. Shelhamer, and T. Darrell. "Fully Convolutional Networks for Semantic Segmentation." Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, 2015, pp. 3431-3440.
- [2] Brostow, G. J., J. Fauqueur, and R. Cipolla. "Semantic object classes in video: A high-definition ground truth database." *Pattern Recognition Letters*. Vol. 30, Issue 2, 2009, pp 88-97.
- [3] Badrinarayanan, V., A. Kendall, and R. Cipolla. "SegNet: A Deep Convolutional Encoder-Decoder Architecture for Image Segmentation." arXiv preprint arXiv:1511.00561, 2015.

See Also

Functions

`coder.checkGpuInstall` | `codegen` | `coder.DeepLearningConfig` |
`coder.loadDeepLearningNetwork`

Objects

`coder.gpuConfig` | `coder.CodeConfig` | `coder.EmbeddedCodeConfig` | `coder.gpuEnvConfig`
| `coder.CuDNNConfig`

Related Examples

- "Semantic Segmentation Using Deep Learning" (Computer Vision Toolbox)
- "Semantic Segmentation on NVIDIA DRIVE" (MATLAB Coder Support Package for NVIDIA Jetson and NVIDIA DRIVE Platforms)
- "Code Generation for Semantic Segmentation Network" on page 4-152
- "Code Generation for Semantic Segmentation Network That Uses U-net" on page 4-169

More About

- "Getting Started with Semantic Segmentation Using Deep Learning" (Computer Vision Toolbox)

Code Generation for Semantic Segmentation Network That Uses U-net

This example shows code generation for an image segmentation application that uses deep learning. It uses the `codegen` command to generate a MEX function that performs prediction on a DAG Network object for U-Net, a deep learning network for image segmentation.

For a similar example covering segmentation of images by using U-Net without the `codegen` command, see “Semantic Segmentation of Multispectral Images Using Deep Learning” (Image Processing Toolbox).

Third-Party Prerequisites

Required

This example generates CUDA MEX and has the following third-party requirements.

- CUDA® enabled NVIDIA® GPU and compatible driver.

Optional

For non-MEX builds such as static, dynamic libraries or executables, this example has the following additional requirements.

- NVIDIA toolkit.
- NVIDIA cuDNN library.
- Environment variables for the compilers and libraries. For more information, see “Third-Party Hardware” and “Setting Up the Prerequisite Products”.

Verify GPU Environment

Use the `coder.checkGpuInstall` function to verify that the compilers and libraries necessary for running this example are set up correctly.

```
envCfg = coder.gpuEnvConfig('host');  
envCfg.DeepLibTarget = 'cudnn';  
envCfg.DeepCodegen = 1;  
envCfg.Quiet = 1;  
coder.checkGpuInstall(envCfg);
```

Segmentation Network

U-Net [1] is a type of convolutional neural network (CNN) designed for semantic image segmentation. In U-Net, the initial series of convolutional layers are interspersed with max pooling layers, successively decreasing the resolution of the input image. These layers are followed by a series of convolutional layers interspersed with upsampling operators, successively increasing the resolution of the input image. Combining these two series paths forms a U-shaped graph. The network was originally trained for and used to perform prediction on biomedical image segmentation applications. This example demonstrates the ability of the network to track changes in forest cover over time. Environmental agencies track deforestation to assess and qualify the environmental and ecological health of a region.

Deep-learning-based semantic segmentation can yield a precise measurement of vegetation cover from high-resolution aerial photographs. One challenge is differentiating classes that have similar

visual characteristics, such as trying to classify a green pixel as grass, shrubbery, or tree. To increase classification accuracy, some data sets contain multispectral images that provide additional information about each pixel. For example, the Hamlin Beach State Park data set supplements the color images with near-infrared channels that provide a clearer separation of the classes.

This example uses the Hamlin Beach State Park Data [2] along with a pretrained U-Net network in order to correctly classify each pixel.

The U-Net used is trained to segment pixels belonging to 18 classes which includes:

- | | | |
|---------------------------------|------------------------|-----------------------------------|
| 0. Other Class/Image Border | 7. Picnic Table | 14. Grass |
| 1. Road Markings | 8. Black Wood Panel | 15. Sand |
| 2. Tree | 9. White Wood Panel | 16. Water (Lake) |
| 3. Building | 10. Orange Landing Pad | 17. Water (Pond) |
| 4. Vehicle (Car, Truck, or Bus) | 11. Water Buoy | 18. Asphalt (Parking Lot/Walkway) |
| 5. Person | 12. Rocks | |
| 6. Lifeguard Chair | 13. Other Vegetation | |

The `segmentImageUnet` Entry-Point Function

The `segmentImageUnet.m` entry-point function performs patchwise semantic segmentation on the input image by using the `multispectralUnet` network found in the `multispectralUnet.mat` file. The function loads the network object from the `multispectralUnet.mat` file into a persistent variable `mynet` and reuses the persistent variable on subsequent prediction calls.

```
type('segmentImageUnet.m')

function out = segmentImageUnet(im,patchSize,trainedNet)
% OUT = segmentImageUnet(IM,patchSize,trainedNet) returns a semantically
% segmented image, segmented using the multi-spectral Unet specified in
% trainedNet. The segmentation is performed over each patch of size
% patchSize.
%
% Copyright 2019-2022 The MathWorks, Inc.

%#codegen
persistent mynet;

if isempty(mynet)
    mynet = coder.loadDeepLearningNetwork(trainedNet);
end

[height, width, nChannel] = size(im);
patch = coder.nullcopy(zeros([patchSize, nChannel-1]));

% Pad image to have dimensions as multiples of patchSize
padSize = zeros(1,2);
padSize(1) = patchSize(1) - mod(height, patchSize(1));
padSize(2) = patchSize(2) - mod(width, patchSize(2));

im_pad = padarray(im, padSize, 0, 'post');
[height_pad, width_pad, ~] = size(im_pad);

out = zeros([size(im_pad,1), size(im_pad,2)], 'uint8');

for i = 1:patchSize(1):height_pad
    for j = 1:patchSize(2):width_pad
        for p = 1:nChannel-1
```

```

        patch(:,:,p) = squeeze( im_pad( i:i+patchSize(1)-1,...
                                        j:j+patchSize(2)-1,...
                                        p));
    end

    % Pass in input
    segmentedLabels = activations(mynet, patch, 'Segmentation-Layer');

    % Takes the max of each channel (6 total at this point)
    [~,L] = max(segmentedLabels,[],3);
    patch_seg = uint8(L);

    % Populate section of output
    out(i:i+patchSize(1)-1, j:j+patchSize(2)-1) = patch_seg;

end
end

% Remove the padding
out = out(1:height, 1:width);

```

Get Pretrained U-Net Network

This example uses the `multispectralUnet` MAT-file containing the pretrained U-Net network. This file is approximately 117 MB in size. Download the file from the MathWorks website.

```
trainedUnetFile = matlab.internal.examples.downloadSupportFile('vision/data','multispectralUnet.r
```

U-Net is a DAG network that contains 58 layers including convolution, max pooling, depth concatenation, and the pixel classification output layers.

```
load(trainedUnetFile);
disp(net)
```

DAGNetwork with properties:

```

    Layers: [58x1 nnet.cnn.layer.Layer]
Connections: [61x2 table]
InputNames: {'ImageInputLayer'}
OutputNames: {'Segmentation-Layer'}

```

To view the network architecture, use the `analyzeNetwork` (Deep Learning Toolbox) function.

```
analyzeNetwork(net);
```

Prepare Data

This example uses the high-resolution multispectral data from [2]. The image set was captured using a drone over the Hamlin Beach State Park, NY. The data contains labeled training, validation, and test sets, with 18 object class labels. The size of the data file is ~3.0 GB.

Download the MAT-file version of the data set using the `downloadHamlinBeachMSIData` helper function. This function is attached to the example as a supporting file.

```

if ~exist(fullfile(pwd,'data'),'dir')
    url = 'http://www.cis.rit.edu/~rmk6217/rit18_data.mat';
    downloadHamlinBeachMSIData(url,pwd+"/data/");
end

```

Load and examine the data in MATLAB.

```
load(fullfile(pwd, 'data', 'rit18_data', 'rit18_data.mat'));
```

```
% Examine data
```

```
whos test_data
```

Name	Size	Bytes	Class	Attributes
test_data	7x12446x7654	1333663576	uint16	

The image has seven channels. The RGB color channels are the third, second, and first image channels. The next three channels correspond to the near-infrared bands and highlight different components of the image based on their heat signatures. Channel 7 is a mask that indicates the valid segmentation region.

The multispectral image data is arranged as numChannels-by-width-by-height arrays. In MATLAB, multichannel images are arranged as width-by-height-by-numChannels arrays. To reshape the data so that the channels are in the third dimension, use the helper function, `switchChannelsToThirdPlane`.

```
test_data = switchChannelsToThirdPlane(test_data);
```

```
% Confirm data has the correct structure (channels last).
```

```
whos test_data
```

Name	Size	Bytes	Class	Attributes
test_data	12446x7654x7	1333663576	uint16	

Run MEX Code Generation

To generate CUDA code for the `segmentImageUnet.m` entry-point function, create a GPU Configuration object for a MEX target setting the target language to C++. Use the `coder.DeepLearningConfig` function to create a CuDNN deep learning configuration object and assign it to the `DeepLearningConfig` property of the GPU code configuration object. Run the `codegen` command specifying an input size of 12446-by-7654-by-7 and a patch size of 1024-by-1024. These values correspond to the entire `test_data` size. The smaller patch sizes speed up inference. To see how the patches are calculated, see the `segmentImageUnet` entry-point function.

```
cfg = coder.gpuConfig('mex');
cfg.ConstantInputs = 'Remove';
cfg.TargetLang = 'C++';
cfg.DeepLearningConfig = coder.DeepLearningConfig('cudnn');
inputArgs = {ones(size(test_data), 'uint16'), ...
    coder.Constant([1024 1024]), coder.Constant(trainedUnetFile)};
```

```
codegen -config cfg segmentImageUnet -args inputArgs -report
```

```
Code generation successful: View report
```

Run Generated MEX to Predict Results for test_data

This `segmentImageUnet` function takes in the data to test (`test_data`) and a vector containing the dimensions of the patch size to use. Take patches of the image, predict the pixels in a particular patch, then combine all the patches together. Due to the size of test data (12446-by-7654-by-7), it is easier to process such a large image in patches.

```
segmentedImage = segmentImageUnet_mex(test_data);
```

To extract only the valid portion of the segmentation, multiply the segmented image by the mask channel of the test data.

```
segmentedImage = uint8(test_data(:,:,7)~=0) .* segmentedImage;
```

Because the output of the semantic segmentation is noisy, remove the noise and stray pixels by using the `medfilt2` function.

```
segmentedImage = medfilt2(segmentedImage,[5,5]);
```

Display U-Net Segmented test_data

The following line of code creates a vector of the class names.

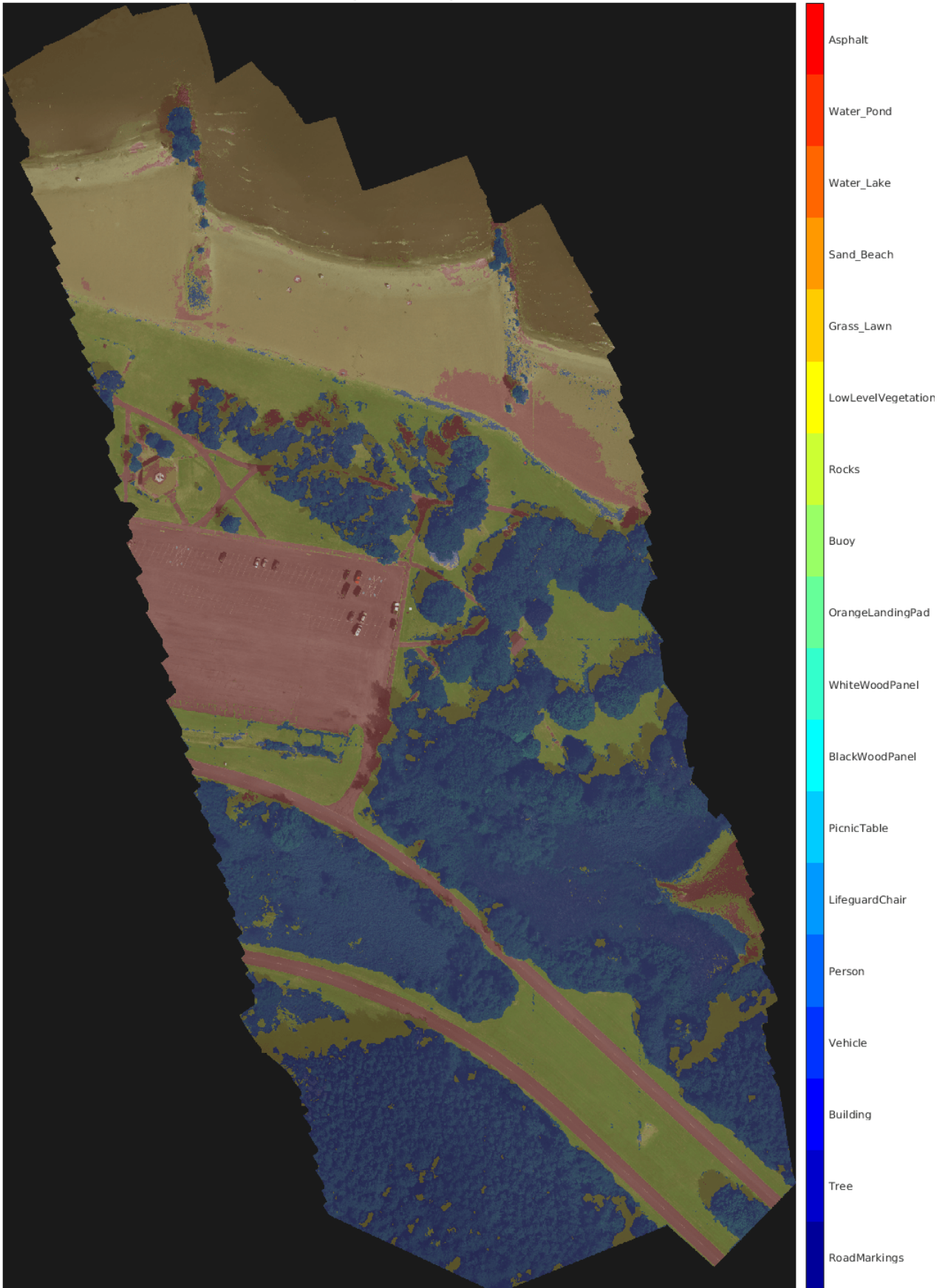
```
classNames = [ "RoadMarkings","Tree","Building","Vehicle","Person", ...
               "LifeguardChair","PicnicTable","BlackWoodPanel",...
               "WhiteWoodPanel","OrangeLandingPad","Buoy","Rocks",...
               "LowLevelVegetation","Grass_Lawn","Sand_Beach",...
               "Water_Lake","Water_Pond","Asphalt"];
```

Overlay the labels on the segmented RGB test image and add a color bar to the segmentation image.

```
cmap = jet(numel(classNames));
B = labeloverlay(imadjust(test_data(:,:, [3,2,1]),[0 0.6],[0.1 0.9],0.55),...
                segmentedImage,'Transparency',0.8,'Colormap',cmap);
figure
imshow(B)

N = numel(classNames);
ticks = 1/(N*2):1/N:1;
colorbar('TickLabels',cellstr(classNames),'Ticks',ticks,'TickLength',0,...
        'TickLabelInterpreter','none');
colormap(cmap)
title('Segmented Image');
```

Segmented Image



References

[1] Ronneberger, Olaf, Philipp Fischer, and Thomas Brox. "U-Net: Convolutional Networks for Biomedical Image Segmentation." *arXiv preprint arXiv:1505.04597*, 2015.

[2] Kemker, R., C. Salvaggio, and C. Kanan. "High-Resolution Multispectral Dataset for Semantic Segmentation." CoRR, abs/1703.01918, 2017.

See Also

Functions

`coder.checkGpuInstall` | `codegen` | `coder.DeepLearningConfig` |
`coder.loadDeepLearningNetwork`

Objects

`coder.gpuConfig` | `coder.gpuEnvConfig` | `coder.CuDNNConfig`

Related Examples

- "Semantic Segmentation of Multispectral Images Using Deep Learning" (Image Processing Toolbox)
- "Semantic Segmentation Using Deep Learning" (Computer Vision Toolbox)
- "Semantic Segmentation on NVIDIA DRIVE" (MATLAB Coder Support Package for NVIDIA Jetson and NVIDIA DRIVE Platforms)
- "Train and Deploy Fully Convolutional Networks for Semantic Segmentation" on page 4-157
- "Code Generation for Semantic Segmentation Network That Uses U-net" on page 4-169

More About

- "Getting Started with Semantic Segmentation Using Deep Learning" (Computer Vision Toolbox)

Code Generation for Denoising Deep Neural Network

This example shows how to generate CUDA® MEX from MATLAB® code and denoise grayscale images by using the denoising convolutional neural network (DnCNN [1]). You can use the denoising network to estimate noise in a noisy image, and then remove it to obtain a denoised image.

Third-Party Prerequisites

Required

This example generates CUDA MEX and has the following third-party requirements.

- CUDA® enabled NVIDIA® GPU and compatible driver.

Optional

For non-MEX builds such as static, dynamic libraries or executables, this example has the following additional requirements.

- NVIDIA toolkit.
- NVIDIA cuDNN library.
- Environment variables for the compilers and libraries. For more information, see “Third-Party Hardware” and “Setting Up the Prerequisite Products”.

Verify GPU Environment

Use the `coder.checkGpuInstall` function to verify that the compilers and libraries necessary for running this example are set up correctly.

```
envCfg = coder.gpuEnvConfig('host');  
envCfg.DeepLibTarget = 'cudnn';  
envCfg.DeepCodegen = 1;  
envCfg.Quiet = 1;  
coder.checkGpuInstall(envCfg);
```

Load Noisy Image

Load a noisy grayscale image into the workspace and display the image.

```
noisyI = imread('noisy_cameraman.png');  
figure  
imshow(noisyI);  
title('Noisy Image');
```

Noisy Image

Get Pretrained Denoising Network

Call the `getDenoisingNetwork` helper function to get a pretrained image denoising deep neural network.

```
net = getDenoisingNetwork;
```

The `getDenoisingNetwork` function returns a pretrained DnCNN [1] that you can use to detect additive white Gaussian noise (AWGN) that has unknown levels. The network is a feed-forward denoising convolutional network that implements a residual learning technique to predict a residual image. In other words, DnCNN [1] computes the difference between a noisy image and the latent clean image.

The network contains 59 layers including convolution, batch normalization, and regression output layers. To display an interactive visualization of the deep learning network architecture, use the `analyzeNetwork` (Deep Learning Toolbox) function.

```
analyzeNetwork(net);
```

The `denoisenet_predict` Function

The `denoisenet_predict` entry-point function takes a noisy image input and returns a denoised image by using a pretrained denoising network.

The function loads the network object returned by `getDenoisingNetwork` into a persistent variable `myNet` and reuses the persistent object on subsequent prediction calls.

```
type denoisenet_predict

function I = denoisenet_predict(in)
%#codegen
% Copyright 2018-2021 The MathWorks, Inc.
```

```
persistent mynet;

if isempty(mynet)
    mynet = coder.loadDeepLearningNetwork('getDenoisingNetwork', 'DnCNN');
end

% The activations method extracts the output from the last layer. The
% 'OutputAs' 'channels' name-value pair argument is used in order to call
% activations on an image whose input dimensions are greater than or equal
% to the network's imageInputLayer.InputSize.

res = mynet.activations(in, 59, 'OutputAs', 'channels');

% Once the noise is estimated, we subtract the noise from the original
% image to obtain a denoised image.

I = in - res;
```

Here, the `activations` method is called with the layer numeric index as 59 to extract the activations from the final layer of the network. The `'OutputAs' 'channels'` name-value pair argument computes activations on images larger than the `imageInputLayer.InputSize` of the network.

The `activations` method returns an estimate of the noise in the input image by using the pretrained denoising image.

Once the noise is estimated, subtract the noise from the original image to obtain a denoised image.

Run MEX Code Generation

To generate CUDA code for the `denoisenet_predict.m` entry-point function, create a GPU code configuration object for a MEX target and set the target language to C++. Use the `coder.DeepLearningConfig` function to create a CuDNN deep learning configuration object and assign it to the `DeepLearningConfig` property of the GPU code configuration object. Run the `codegen` command specifying an input size of [256,256]. This value corresponds to the size of the noisy image that you intend to denoise.

```
cfg = coder.gpuConfig('mex');
cfg.TargetLang = 'C++';
cfg.DeepLearningConfig = coder.DeepLearningConfig('cudnn');
codegen -config cfg denoisenet_predict -args {ones(256,256,'single')} -report
```

Code generation successful: [View report](#)

Run Generated MEX

The DnCNN [1] is trained on input images having an input range [0,1]. Call the `im2single` (Image Processing Toolbox) function on `noisyI` to rescale the values from [0,255] to [0,1].

Call `denoisenet_predict_predict` on the rescaled input image.

```
denoisedI = denoisenet_predict_mex(im2single(noisyI));
```

View Denoised Image

```
figure
imshowpair(noisyI,denoisedI,'montage');
title('Noisy Image (left) and Denoised Image (right)');
```

Noisy Image (left) and Denoised Image (right)**References**

[1] Zhang, K., W. Zuo, Y. Chen, D. Meng, and L. Zhang. "Beyond a Gaussian Denoiser: Residual Learning of Deep CNN for Image Denoising." IEEE Transactions on Image Processing. Vol. 26, Number 7, Feb. 2017, pp. 3142-3155.

See Also**Functions**

`coder.checkGpuInstall` | `codegen` | `coder.DeepLearningConfig` |
`coder.loadDeepLearningNetwork`

Objects

`coder.gpuConfig` | `coder.CodeConfig` | `coder.EmbeddedCodeConfig` | `coder.gpuEnvConfig`
| `coder.CuDNNConfig`

More About

- "Generated CNN Class Hierarchy" on page 4-65
- "Supported Networks, Layers, and Classes" on page 4-6
- "Load Pretrained Networks for Code Generation" on page 4-66
- "Code Generation for Deep Learning Networks by Using cuDNN" on page 4-69
- "Code Generation for Deep Learning Networks by Using TensorRT" on page 4-78

Code Generation for Object Detection by Using YOLO v2

This example shows how to generate CUDA® MEX for a you only look once (YOLO) v2 object detector. A YOLO v2 object detection network is composed of two subnetworks. A feature extraction network followed by a detection network. This example generates code for the network trained in the *Object Detection Using YOLO v2 Deep Learning* example from Computer Vision Toolbox™. For more information, see “Object Detection Using YOLO v2 Deep Learning” (Computer Vision Toolbox). You can modify this example to generate CUDA® MEX for the network imported in the *Import Pretrained ONNX YOLO v2 Object Detector* example from Computer Vision Toolbox™. For more information, see “Import Pretrained ONNX YOLO v2 Object Detector” (Computer Vision Toolbox).

Third-Party Prerequisites

Required

This example generates CUDA MEX and has the following third-party requirements.

- CUDA® enabled NVIDIA® GPU and compatible driver.

Optional

For non-MEX builds such as static, dynamic libraries or executables, this example has the following additional requirements.

- NVIDIA toolkit.
- NVIDIA cuDNN library.
- Environment variables for the compilers and libraries. For more information, see “Third-Party Hardware” and “Setting Up the Prerequisite Products”.

Verify GPU Environment

Use the `coder.checkGpuInstall` function to verify that the compilers and libraries necessary for running this example are set up correctly.

```
envCfg = coder.gpuEnvConfig('host');
envCfg.DeepLibTarget = 'cudnn';
envCfg.DeepCodegen = 1;
envCfg.Quiet = 1;
coder.checkGpuInstall(envCfg);
```

Get Pretrained DAGNetwork

This example uses the `yolov2ResNet50VehicleExample` MAT-file containing the pretrained network. The file is approximately 98MB in size. Download the file from the MathWorks website.

```
matFile = matlab.internal.examples.downloadSupportFile('vision/data','yolov2ResNet50VehicleExamp');
vehicleDetector = load(matFile);
net = vehicleDetector.detector.Network

net =
  DAGNetwork with properties:
    Layers: [150×1 nnet.cnn.layer.Layer]
    Connections: [162×2 table]
```

```
InputNames: {'input_1'}
OutputNames: {'yolov2OutputLayer'}
```

The DAG network contains 150 layers including convolution, ReLU, and batch normalization layers and the YOLO v2 transform and YOLO v2 output layers. To display an interactive visualization of the deep learning network architecture, use the `analyzeNetwork` (Deep Learning Toolbox) function.

```
analyzeNetwork(net);
```

The `yolov2_detect` Entry-Point Function

The `yolov2_detect.m` entry-point function takes an image input and runs the detector on the image using the deep learning network saved in the `yolov2ResNet50VehicleExample.mat` file. The function loads the network object from the `yolov2ResNet50VehicleExample.mat` file into a persistent variable `yolov2Obj` and reuses the persistent object on subsequent detection calls.

```
type('yolov2_detect.m')

function outImg = yolov2_detect(in,matFile)

% Copyright 2018-2021 The MathWorks, Inc.

persistent yolov2Obj;

if isempty(yolov2Obj)
    yolov2Obj = coder.loadDeepLearningNetwork(matFile);
end

% Call to detect method
[bboxes,~,labels] = yolov2Obj.detect(in,'Threshold',0.5);

% Convert categorical labels to cell array of character vectors
labels = cellstr(labels);

% Annotate detections in the image.
outImg = insertObjectAnnotation(in,'rectangle',bboxes,labels);
```

Run MEX Code Generation

To generate CUDA code for the entry-point function, create a GPU code configuration object for a MEX target and set the target language to C++. Use the `coder.DeepLearningConfig` function to create a CuDNN deep learning configuration object and assign it to the `DeepLearningConfig` property of the GPU code configuration object. Run the `codegen` command specifying an input size of 224-by-224-by-3. This value corresponds to the input layer size of YOLOv2.

```
cfg = coder.gpuConfig('mex');
cfg.TargetLang = 'C++';
cfg.DeepLearningConfig = coder.DeepLearningConfig('cudnn');
cfg.GenerateReport = true;
inputArgs = {ones(224,224,3,'uint8'),coder.Constant(matFile)};

codegen -config cfg yolov2_detect -args inputArgs
```

```
Code generation successful: View report
```

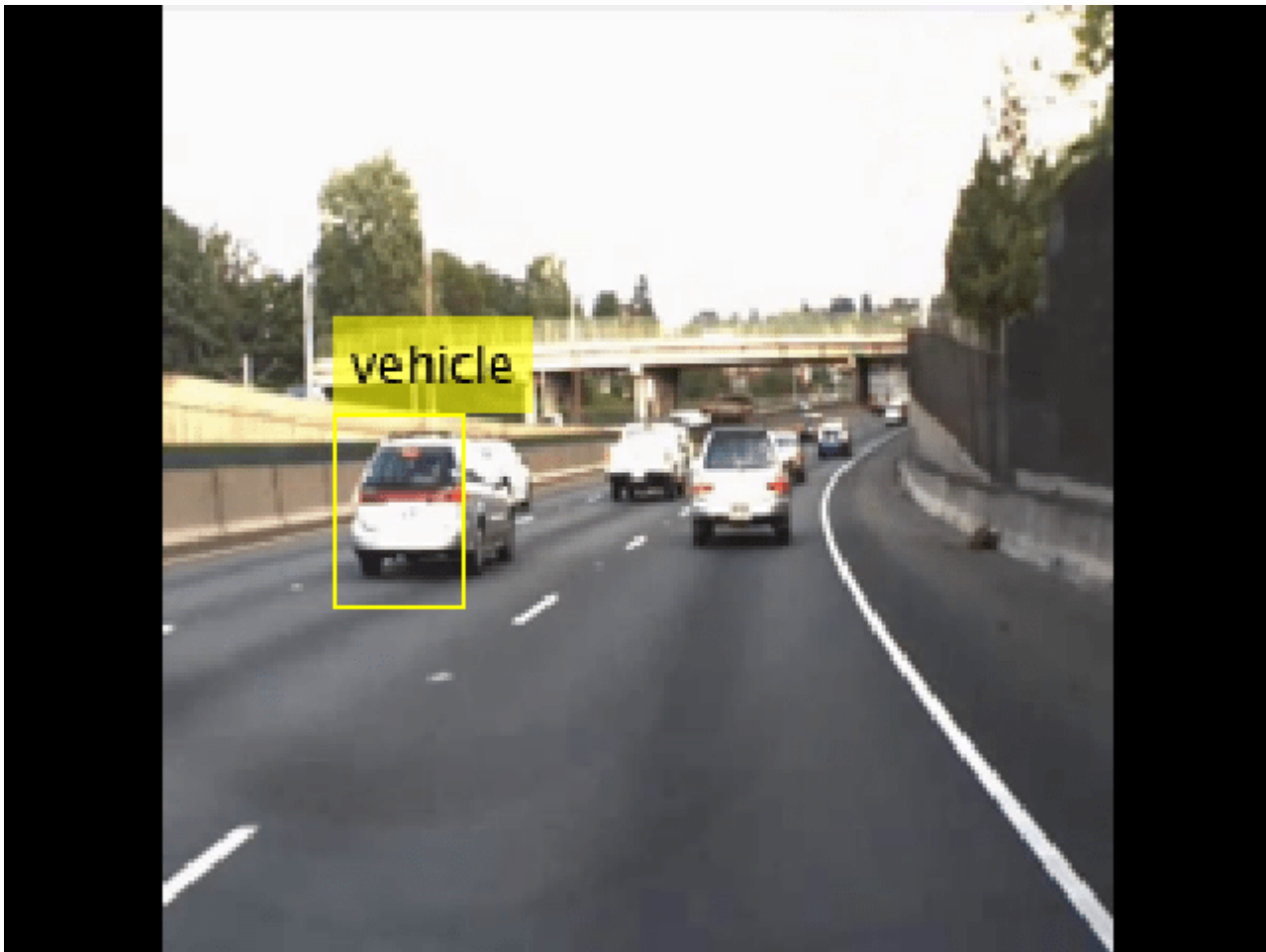
Run Generated MEX

Set up the video file reader and read the input video. Create a video player to display the video and the output detections.

```
videoFile = 'highway_lanechange.mp4';  
videoFreader = vision.VideoFileReader(videoFile, 'VideoOutputDataType', 'uint8');  
depVideoPlayer = vision.DeployableVideoPlayer('Size', 'Custom', 'CustomSize', [640 480]);
```

Read the video input frame-by-frame and detect the vehicles in the video using the detector.

```
cont = ~isDone(videoFreader);  
while cont  
    I = step(videoFreader);  
    in = imresize(I, [224, 224]);  
    out = yolov2_detect_mex(in, matFile);  
    step(depVideoPlayer, out);  
    % Exit the loop if the video player figure window is closed  
    cont = ~isDone(videoFreader) && isOpen(depVideoPlayer);  
end
```



References

[1] Redmon, Joseph, and Ali Farhadi. "YOLO9000: Better, Faster, Stronger." 2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR). IEEE, 2017.

Copyright 2017-2021 The MathWorks, Inc.

See Also

Functions

`coder.checkGpuInstall` | `codegen` | `coder.DeepLearningConfig` |
`coder.loadDeepLearningNetwork`

Objects

`coder.gpuConfig` | `coder.gpuEnvConfig` | `coder.CuDNNConfig` | `vision.VideoFileReader` |
`vision.DeployableVideoPlayer`

Related Examples

- "Object Detection Using YOLO v2 Deep Learning" (Computer Vision Toolbox)
- "Import Pretrained ONNX YOLO v2 Object Detector" (Computer Vision Toolbox)
- "Code Generation for Object Detection by Using Single Shot Multibox Detector" on page 4-193
- "Code Generation For Object Detection Using YOLO v3 Deep Learning" on page 4-217

More About

- "Getting Started with YOLO v2" (Computer Vision Toolbox)
- "Anchor Boxes for Object Detection" (Computer Vision Toolbox)
- "Supported Networks, Layers, and Classes" on page 4-6
- "Load Pretrained Networks for Code Generation" on page 4-66

Code Generation for a Sequence-to-Sequence LSTM Network

This example demonstrates how to generate CUDA® code for a long short-term memory (LSTM) network. The example generates a MEX application that makes predictions at each step of an input timeseries. Two methods are demonstrated: a method using a standard LSTM network, and a method leveraging the stateful behavior of the same LSTM network. This example uses accelerometer sensor data from a smartphone carried on the body and makes predictions on the activity of the wearer. User movements are classified into one of five categories, namely dancing, running, sitting, standing, and walking. The example uses a pretrained LSTM network. For more information on training, see the “Sequence Classification Using Deep Learning” (Deep Learning Toolbox) example from Deep Learning Toolbox™.

Third-Party Prerequisites

Required

This example generates CUDA MEX and has the following third-party requirements.

- CUDA enabled NVIDIA® GPU and compatible driver.

Optional

For non-MEX builds such as static, dynamic libraries or executables, this example has the following additional requirements.

- NVIDIA toolkit.
- NVIDIA cuDNN library.
- Environment variables for the compilers and libraries. For more information, see “Third-Party Hardware” and “Setting Up the Prerequisite Products”.

Verify GPU Environment

Use the `coder.checkGpuInstall` function to verify that the compilers and libraries necessary for running this example are set up correctly.

```
envCfg = coder.gpuEnvConfig('host');  
envCfg.DeepLibTarget = 'cudnn';  
envCfg.DeepCodegen = 1;  
envCfg.Quiet = 1;  
coder.checkGpuInstall(envCfg);
```

The `lstmnet_predict` Entry-Point Function

A sequence-to-sequence LSTM network enables you to make different predictions for each individual time step of a data sequence. The `lstmnet_predict.m` entry-point function takes an input sequence and passes it to a trained LSTM network for prediction. Specifically, the function uses the LSTM network trained in the *Sequence to Sequence Classification Using Deep Learning* example. The function loads the network object from the `lstmnet_predict.mat` file into a persistent variable and reuses the persistent object on subsequent prediction calls.

To display an interactive visualization of the network architecture and information about the network layers, use the `analyzeNetwork` (Deep Learning Toolbox) function.

```
type('lstmnet_predict.m')
```

```
function out = lstmnet_predict(in) %#codegen
% Copyright 2019-2021 The MathWorks, Inc.
persistent mynet;
if isempty(mynet)
    mynet = coder.loadDeepLearningNetwork('lstmnet.mat');
end
% pass in input
out = predict(mynet,in);
```

Generate CUDA MEX

To generate CUDA MEX for the `lstmnet_predict.m` entry-point function, create a GPU configuration object and specify the target to be MEX. Set the target language to C++. Create a deep learning configuration object that specifies the target library as cuDNN. Attach this deep learning configuration object to the GPU configuration object.

```
cfg = coder.gpuConfig('mex');
cfg.TargetLang = 'C++';
cfg.DeepLearningConfig = coder.DeepLearningConfig('cudnn');
```

At compile time, GPU Coder™ must know the data types of all the inputs to the entry-point function. Specify the type and size of the input argument to the `codegen` command by using the `coder.typeof` function. For this example, the input is of double data type with a feature dimension value of three and a variable sequence length. Specifying the sequence length as variable-sized enables us to perform prediction on an input sequence of any length.

```
matrixInput = coder.typeof(double(0),[3 Inf],[false true]);
```

Run the `codegen` command.

```
codegen -config cfg lstmnet_predict -args {matrixInput} -report
```

Code generation successful: [View report](#)

Run Generated MEX on Test Data

Load the `HumanActivityValidate` MAT-file. This MAT-file stores the variable `XValidate` that contains sample timeseries of sensor readings on which you can test the generated code. Call `lstmnet_predict_mex` on the first observation.

```
load HumanActivityValidate
YPred1 = lstmnet_predict_mex(XValidate{1});
```

`YPred1` is a 5-by-53888 numeric matrix containing the probabilities of the five classes for each of the 53888 time steps. For each time step, find the predicted class by calculating the index of the maximum probability.

```
[~, maxIndex] = max(YPred1, [], 1);
```

Associate the indices of max probability to the corresponding label. Display the first ten labels. From the results, you can see that the network predicted the human to be sitting for the first ten time steps.

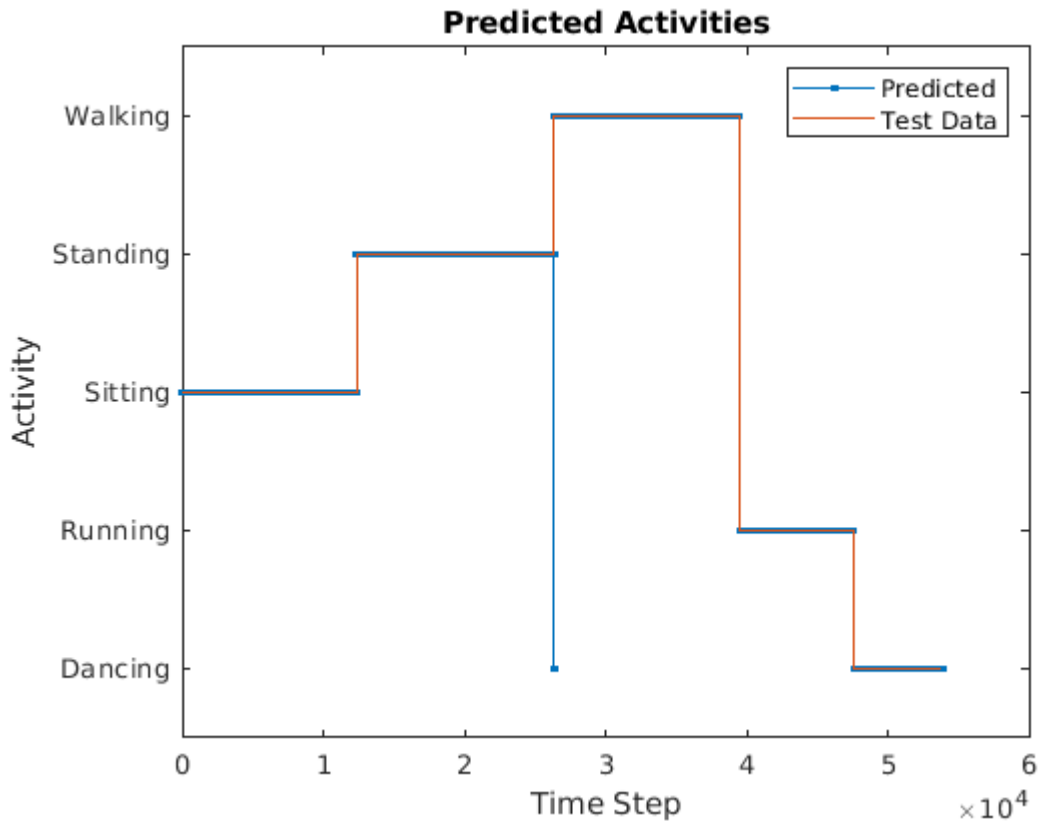
```
labels = categorical({'Dancing', 'Running', 'Sitting', 'Standing', 'Walking'});  
predictedLabels1 = labels(maxIndex);  
disp(predictedLabels1(1:10)')
```

```
Sitting  
Sitting  
Sitting  
Sitting  
Sitting  
Sitting  
Sitting  
Sitting  
Sitting  
Sitting
```

Compare Predictions with Test Data

Use a plot to compare the MEX output data with the test data.

```
figure  
plot(predictedLabels1, '-.-');  
hold on  
plot(YValidate{1});  
hold off  
  
xlabel("Time Step")  
ylabel("Activity")  
title("Predicted Activities")  
legend(["Predicted" "Test Data"])
```



Call Generated MEX on an Observation with a Different Sequence Length

Call `lstmnet_predict_mex` on the second observation with a different sequence length. In this example, `XValidate{2}` has a sequence length of 64480 whereas `XValidate{1}` had a sequence length of 53888. The generated code handles prediction correctly because we specified the sequence length dimension to be variable-size.

```
YPred2 = lstmnet_predict_mex(XValidate{2});
[~, maxIndex] = max(YPred2, [], 1);
predictedLabels2 = labels(maxIndex);
disp(predictedLabels2(1:10)')
```

```
Sitting
Sitting
Sitting
Sitting
Sitting
Sitting
Sitting
Sitting
Sitting
Sitting
Sitting
```

Generate MEX that takes in Multiple Observations

If you want to perform prediction on many observations at once, you can group the observations together in a cell array and pass the cell array for prediction. The cell array must be a column cell

array, and each cell must contain one observation. Each observation must have the same feature dimension, but the sequence lengths may vary. In this example, `XValidate` contains five observations. To generate a MEX that can take `XValidate` as input, specify the input type to be a 5-by-1 cell array. Further, specify that each cell be of the same type as `matrixInput`, the type you specified for the single observation in the previous codegen command.

```
matrixInput = coder.typeof(double(0),[3 Inf],[false true]);
cellInput = coder.typeof({matrixInput}, [5 1]);

codegen -config cfg lstmnet_predict -args {cellInput} -report

Code generation successful: View report
```

```
YPred3 = lstmnet_predict_mex(XValidate);
```

The output is a 5-by-1 cell array of predictions for the five observations passed in.

```
disp(YPred3)

    {5×53888 single}
    {5×64480 single}
    {5×53696 single}
    {5×56416 single}
    {5×50688 single}
```

Generate MEX with Stateful LSTM

Instead of passing the entire timeseries to predict in one step, we can run prediction on an input by streaming in one timestep at a time, making use of the function `predictAndUpdateState` (Deep Learning Toolbox) This function takes in an input, produces an output prediction, and updates the internal state of the network so that future predictions take this initial input into account.

The entry-point function `lstmnet_predict_and_update.m` takes in a single-timestep input and processes the input using the `predictAndUpdateState` (Deep Learning Toolbox) function. `predictAndUpdateState` outputs a prediction for the input timestep and updates the network so that subsequent inputs are treated as subsequent timesteps of the same sample. After passing in all timesteps one at a time, the resulting output is the same as if all timesteps were passed in as a single input.

```
type('lstmnet_predict_and_update.m')

function out = lstmnet_predict_and_update(in) %#codegen

% Copyright 2019-2021 The MathWorks, Inc.

persistent mynet;

if isempty(mynet)
    mynet = coder.loadDeepLearningNetwork('lstmnet.mat');
end

% pass in input
[mynet, out] = predictAndUpdateState(mynet,in);
```

Run codegen on this new design file. Since we are taking in a single timestep each call, we specify `matrixInput` to have a fixed sequence dimension of 1 instead of a variable sequence length.

```
matrixInput = coder.typeof(double(0),[3 1]);
codegen -config cfg lstmnet_predict_and_update -args {matrixInput} -report
```

Code generation successful: [View report](#)

Run the generated MEX on the first validation sample's first timestep.

```
firstSample = XValidate{1};
firstTimestep = firstSample(:,1);
YPredStateful = lstmnet_predict_and_update_mex(firstTimestep);
[~, maxIndex] = max(YPredStateful, [], 1);
predictedLabelsStateful1 = labels(maxIndex)

predictedLabelsStateful1 = categorical
    Sitting
```

Compare the output label with the ground truth.

```
YValidate{1}(1)

ans = categorical
    Sitting
```

See Also

Functions

[coder.checkGpuInstall](#) | [codegen](#) | [coder.DeepLearningConfig](#) | [coder.loadDeepLearningNetwork](#)

Objects

[coder.gpuConfig](#) | [coder.gpuEnvConfig](#) | [coder.CuDNNConfig](#) | [coder.TensorRTConfig](#)

Related Examples

- “Sequence Classification Using Deep Learning” (Deep Learning Toolbox)
- “Code Generation for a Video Classification Network” on page 4-211

More About

- “Long Short-Term Memory Networks” (Deep Learning Toolbox)
- “Supported Networks, Layers, and Classes” on page 4-6
- “Load Pretrained Networks for Code Generation” on page 4-66

Deep Learning Prediction on ARM Mali GPU

This example shows how to use the `cnncodegen` function to generate code for an image classification application that uses deep learning on ARM® Mali GPUs. The example uses the `MobileNet-v2` DAG network to perform image classification. The generated code takes advantage of the ARM Compute library for computer vision and machine learning.

Prerequisites

- ARM Mali GPU based hardware. For example, HiKey960 is one of the target platforms that contains a Mali GPU.
- ARM Compute Library on the target ARM hardware built for the Mali GPU.
- Open source Computer Vision Library (OpenCV v2.4.9) on the target ARM hardware.
- Environment variables for the compilers and libraries. Ensure that the `ARM_COMPUTE` and the `LD_LIBRARY_PATH` variables are set on the target platform. For information on the supported versions of the compilers and libraries, see “Third-Party Hardware”. For setting up the environment variables, see “Setting Up the Prerequisite Products”.

Get Pretrained DAGNetwork

Load the pretrained `MobileNet-v2` network available in the `Deep Learning Toolbox Model for MobileNet-v2 Network`.

```
net = mobilenetv2

net =
  DAGNetwork with properties:

    Layers: [154×1 nnet.cnn.layer.Layer]
  Connections: [163×2 table]
    InputNames: {'input_1'}
    OutputNames: {'ClassificationLayer_Logits'}
```

The network contains 155 layers including convolution, batch normalization, softmax, and the classification output layers. The `analyzeNetwork()` function displays an interactive plot of the network architecture and a table containing information about the network layers.

```
analyzeNetwork(net);
```

Generate Code

For deep learning on ARM targets, you generate code on the host development computer. To build and run the executable program, move the generated code to the ARM target platform. The target platform must have an ARM Mali GPU. For example, HiKey960 is one of the target platforms on which you can execute the code generated in this example.

Call the `cnncodegen` function, specifying the target library as `arm-compute-mali`.

```
cnncodegen(net, 'targetlib', 'arm-compute-mali');
```


Copy Generated Files to the Target

Move the generated codegen folder and other required files from the host development computer to the target platform by using your preferred SCP (Secure Copy Protocol) or Secure Shell File Transfer Protocol (SSH) client.

For example, on the Linux® platform, to transfer the files to the HiKey960, use the scp command with the format:

```
system('sshpass -p [password] scp (sourcefile) [username]@[targetname]:~/');
system('sshpass -p password scp main_mobilenet_arm_generic.cpp username@targetname:~/');
system('sshpass -p password scp peppers_mobilenet.png username@targetname:~/');
system('sshpass -p password scp makefile_mobilenet_arm_generic.mk username@targetname:~/');
system('sshpass -p password scp synsetWords.txt username@targetname:~/');
system('sshpass -p password scp -r codegen username@targetname:~/');
```

On the Windows® platform, you can use the pscp tool that comes with a PuTTY installation. For example:

```
system('pscp -pw password -r codegen username@targetname:/home/username');
```

PSCP utilities must be either on your PATH or in your current folder.

Build Executable

To build the library on the target platform, use the generated makefile `cnnbuild_rtw.mk`.

For example, to build the library on the HiKey960:

```
system('sshpass -p password ssh username@targetname' ...
' "make -C /home/username/codegen -f cnnbuild_rtw.mk"');
```

On the Windows platform, you can use the putty command with `-ssh` argument to log in and run the make command. For example:

```
system('putty -ssh username@targetname -pw password');
```

To build and run the executable on the target platform, use the command with the format: `make -C /home/$(username) and ./execfile -f makefile_mobilenet_arm_generic.mk`

For example, on the HiKey960:

```
make -C /home/username arm_mobilenet -f makefile_mobilenet_arm_generic.mk
```

Run the executable on the ARM platform specifying an input image file.

```
./mobilenet_exe peppers_mobilenet.png
```

The top five predictions for the input image file are:

```
Top 5 Predictions:
-----
88.976% bell pepper
4.907% cucumber
1.390% grocery store
0.512% Granny Smith
0.256% lemon
```



See Also

Functions
cnncodegen

See Also

More About

- “Supported Networks, Layers, and Classes” on page 4-6
- “Code Generation for Deep Learning Networks Targeting ARM Mali GPUs” on page 4-88

Code Generation for Object Detection by Using Single Shot Multibox Detector

This example shows how to generate CUDA® code for an SSD network (ssdObjectDetector object) and take advantage of the NVIDIA® cuDNN and TensorRT libraries. An SSD network is based on a feed-forward convolutional neural network that detect multiple objects within the image in a single shot. SSD network can be thought of as having two sub-networks. A feature extraction network, followed by a detection network.

This example generates code for the network trained in the *Object Detection Using SSD Deep Learning* example from Computer Vision Toolbox™. For more information, see “Object Detection Using SSD Deep Learning” (Computer Vision Toolbox). The *Object Detection Using SSD Deep Learning* example uses ResNet-50 for feature extraction. The detection sub-network is a small CNN compared to the feature extraction network and is composed of a few convolutional layers and layers specific to SSD.

Third-Party Prerequisites

Required

This example generates CUDA MEX and has the following third-party requirements.

- CUDA enabled NVIDIA GPU and compatible driver.

Optional

For non-MEX builds such as static, dynamic libraries or executables, this example has the following additional requirements.

- NVIDIA toolkit.
- NVIDIA cuDNN library.
- Environment variables for the compilers and libraries. For more information, see “Third-Party Hardware” and “Setting Up the Prerequisite Products”.

Verify GPU Environment

Use the `coder.checkGpuInstall` function to verify that the compilers and libraries necessary for running this example are set up correctly.

```
envCfg = coder.gpuEnvConfig('host');
envCfg.DeepLibTarget = 'cudnn';
envCfg.DeepCodegen = 1;
envCfg.Quiet = 1;
coder.checkGpuInstall(envCfg);
```

Get Pretrained DAG Network

This example uses the `ssdResNet50VehicleExample_20a` MAT-file containing the pretrained SSD network. This file is approximately 44 MB size. Download the file from the MathWorks website.

```
ssdNetFile = matlab.internal.examples.downloadSupportFile('vision/data','ssdResNet50VehicleExamp
```

The DAG network contains 180 layers including convolution, ReLU, and batch normalization layers, anchor box, SSD merge, focal loss, and other layers. To display an interactive visualization of the deep learning network architecture, use the `analyzeNetwork` (Deep Learning Toolbox) function.

```
load(ssdNetFile);
analyzeNetwork(detector.Network);
```

The `ssdObj_detect` Entry-Point Function

The `ssdObj_detect.m` entry-point function takes an image input and runs the detector on the image using the deep learning network saved in the `ssdResNet50VehicleExample_20a.mat` file. The function loads the network object from the `ssdResNet50VehicleExample_20a.mat` file into a persistent variable `ssdObj` and reuses the persistent object on subsequent detection calls.

```
type('ssdObj_detect.m')

function outImg = ssdObj_detect(in,matFile)

% Copyright 2019-2022 The MathWorks, Inc.

persistent ssdObj;

if isempty(ssdObj)
    ssdObj = coder.loadDeepLearningNetwork(matFile);
end

% Pass in input
[bboxes,~,labels] = detect(ssdObj,in,'Threshold',0.5);

% Convert categorical labels to cell array of character vectors for
% execution
labels = cellstr(labels);

% Annotate detections in the image.
if ~isempty(labels)
    outImg = insertObjectAnnotation(in,'rectangle',bboxes,labels);
else
    outImg = in;
end
```

Run MEX Code Generation

To generate CUDA code for the `ssdObj_detect.m` entry-point function, create a GPU code configuration object for a MEX target and set the target language to C++. Use the `coder.DeepLearningConfig` function to create a CuDNN deep learning configuration object and assign it to the `DeepLearningConfig` property of the GPU code configuration object. Run the `codegen` command specifying an input size of 300-by-300-by-3. This value corresponds to the input layer size of SSD Network.

```
cfg = coder.gpuConfig('mex');
cfg.TargetLang = 'C++';
cfg.DeepLearningConfig = coder.DeepLearningConfig('cudnn');
inputArgs = {ones(300,300,3,'uint8'),coder.Constant(ssdNetFile)};
codegen -config cfg ssdObj_detect -args inputArgs -report
```

Code generation successful: [View report](#)

Run Generated MEX

To test the generated MEX, the example uses a small vehicle data set that contains 295 images. Many of these images come from the Caltech Cars 1999 and 2001 data sets, available at the Caltech Research Data Respository website, created by Pietro Perona and used with permission.

Load the vehicle data set and randomly select 10 images to test the generated code.

```
unzip vehicleDatasetImages.zip
imageNames = dir(fullfile(pwd, 'vehicleImages', '*.jpg'));
imageNames = {imageNames.name}';
rng(0);
imageIndices = randi(length(imageNames),1,10);
```

Read the video input frame-by-frame and detect the vehicles in the video using the detector.

```
for idx = 1:10
    testImage = imread(fullfile(pwd, 'vehicleImages', imageNames{imageIndices(idx)}));
    resizedImage = imresize(testImage, [300,300]);
    detectorOutput = ssdObj_detect_mex(resizedImage, ssdNetFile);
    imshow(detectorOutput);
    pause(0.5)
end
```



References

[1] Liu, Wei, Dragomir Anguelov, Dumitru Erhan, Christian Szegedy, Scott Reed, Cheng Yang Fu, and Alexander C. Berg. "SSD: Single shot multibox detector." In 14th European Conference on Computer Vision, ECCV 2016. Springer Verlag, 2016.

See Also

Functions

`coder.checkGpuInstall` | `codegen` | `coder.DeepLearningConfig` |
`coder.loadDeepLearningNetwork`

Objects

`coder.gpuConfig` | `coder.gpuEnvConfig` | `coder.CuDNNConfig` | `vision.VideoFileReader` |
`vision.DeployableVideoPlayer`

Related Examples

- "Object Detection Using SSD Deep Learning" (Computer Vision Toolbox)
- "Code Generation for Object Detection by Using YOLO v2" on page 4-180
- "Code Generation For Object Detection Using YOLO v3 Deep Learning" on page 4-217

More About

- "Getting Started with SSD Multibox Detection" (Computer Vision Toolbox)
- "Anchor Boxes for Object Detection" (Computer Vision Toolbox)

Code Generation for a Deep Learning Simulink Model to Classify ECG Signals

This example demonstrates how you can use powerful signal processing techniques and Convolutional Neural Networks together to classify ECG signals. We will also showcase how CUDA® code can be generated from the Simulink® model. This example uses the pretrained CNN network from the *Classify Time Series Using Wavelet Analysis and Deep Learning* example of the Wavelet Toolbox™ to classify ECG signals based on images from the CWT of the time series data. For information on training, see “Classify Time Series Using Wavelet Analysis and Deep Learning” (Wavelet Toolbox).

For a video demonstration on how to perform software-in-the-loop (SIL), processor-in-the-loop (PIL) simulation, and deploying this example to NVIDIA Jetson® board, see <https://www.mathworks.com/videos/deep-learning-in-simulink-for-nvidia-gpus-classification-of-ecg-signals-1621401016961.html>.

This example illustrates the following concepts:

- Model the classification application in Simulink. Use **MATLAB Function** blocks to perform preprocessing and wavelet transforms of the ECG data. Use the **Image Classifier** block from the Deep Learning Toolbox™ for loading the pretrained network and performing the classification of the ECG data.
- Configure the model for code generation.
- Generate a CUDA executable for the Simulink model.

Third-Party Prerequisites

- CUDA enabled NVIDIA GPU.
- NVIDIA CUDA toolkit and driver.
- NVIDIA cuDNN library.
- Environment variables for the compilers and libraries. For more information, see “Third-Party Hardware” and “Setting Up the Prerequisite Products”.

Verify GPU Environment

To verify that the compilers and libraries necessary for running this example are set up correctly, use the `coder.checkGpuInstall` function.

```
envCfg = coder.gpuEnvConfig('host');  
envCfg.DeepLibTarget = 'cudnn';  
envCfg.DeepCodegen = 1;  
envCfg.Quiet = 1;  
coder.checkGpuInstall(envCfg);
```

ECG Data Description

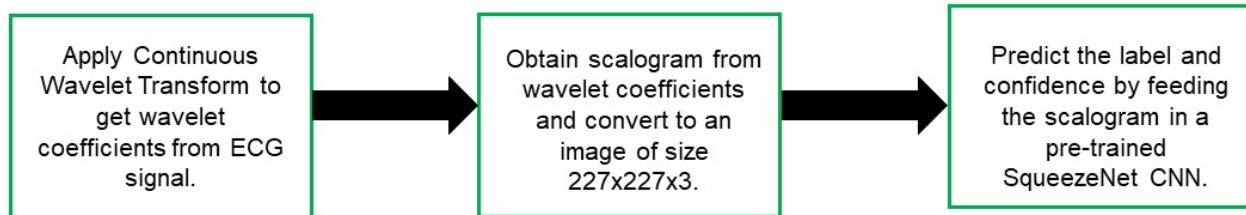
This example uses ECG data from PhysioNet database. It contains data from three groups of people:

- 1 Persons with cardiac arrhythmia (ARR)
- 2 Persons with congestive heart failure (CHF)
- 3 Persons with normal sinus rhythms (NSR)

It includes 96 recordings from persons with ARR, 30 recordings from persons with CHF, and 36 recordings from persons with NSR. The `ecg_signals` MAT-file contains the test ECG data in time series format. The image classifier in this example distinguishes between ARR, CHF, and NSR.

Algorithmic Workflow

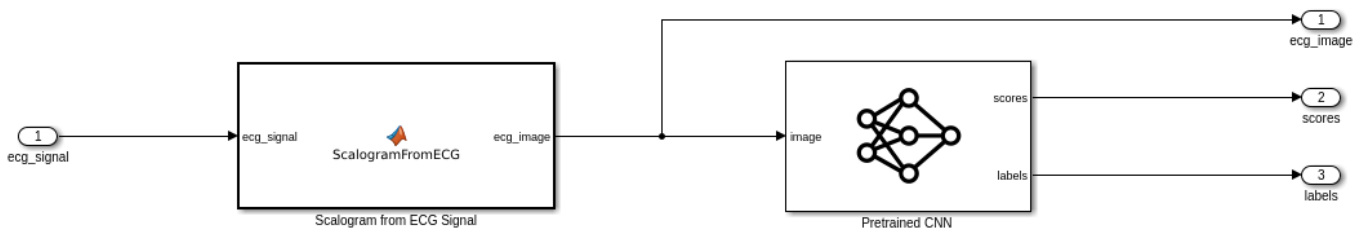
The block diagram for the algorithmic workflow of the Simulink model is shown.

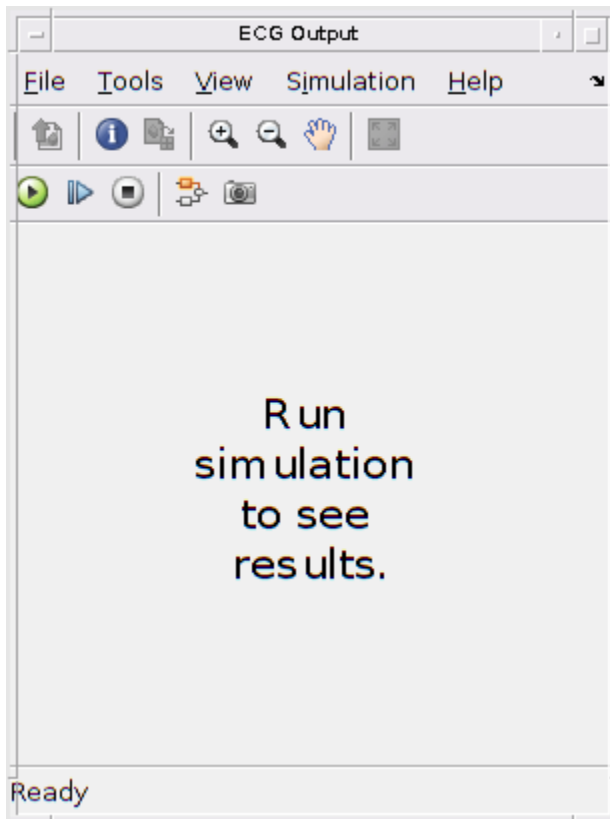


ECG Deep Learning Simulink Model

The Simulink model for classifying the ECG signals is shown. When the model runs, the Video Viewer block displays the classified ECG signal.

```
open_system('ecg_dl_cwt');
```





ECG Preprocessing Subsystem

The ECG Preprocessing subsystem contains a MATLAB Function block that performs CWT to obtain scalogram of the ECG signal and then processes the scalogram to obtain an image and an Image Classifier block that loads the pretrained network from `trainedNet.mat` and performs prediction for image classification based on SqueezeNet deep learning CNN.

```
open_system('ecg_dl_cwt/ECG Preprocessing');
```

The ScalogramFromECG function block defines a function called `ecg_to_scalogram` that:

- Uses 65536 samples of double-precision ECG data as input.
- Create time frequency representation from the ECG data by applying Wavelet transform.
- Obtain scalogram from the wavelet coefficients.
- Convert the scalogram to image of size (227x227x3).

The function signature of `ecg_to_scalogram` is shown.

```
type ecg_to_scalogram
```

```
function ecg_image = ecg_to_scalogram(ecg_signal)
```

```
% Copyright 2020 The MathWorks, Inc.
```

```
persistent jetdata;
if isempty(jetdata)
```

```
    jetdata = ecgColorMap(128,'single');
end
% Obtain wavelet coefficients from ECG signal
cfs = cwt_ecg(ecg_signal);
% Obtain scalogram from wavelet coefficients
image = ind2rgb(im2uint8(rescale(cfs)),jetdata);
ecg_image = im2uint8(imresize(image,[227,227]));

end
```

ECG Postprocessing

The ECG Postprocessing MATLAB function block defines the `label_prob_image` function that finds the label for the scalogram image based on the highest score from the scores outputted by the image classifier. It outputs the scalogram image with the label and confidence printed on it.

type `label_prob_image`

```
function final_image = label_prob_image(ecg_image, scores, labels)

% Copyright 2020-2021 The MathWorks, Inc.

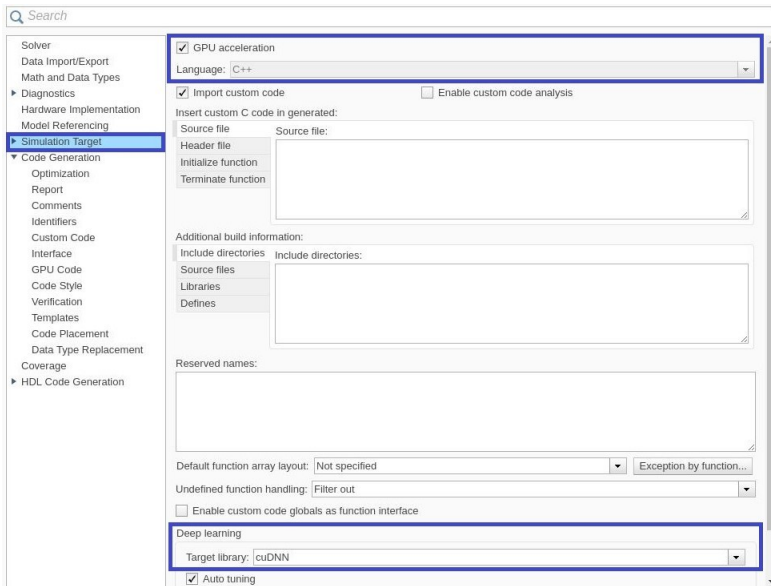
scores = double(scores);
% Obtain maximum confidence
[prob,index] = max(scores);
confidence = prob*100;
% Obtain label corresponding to maximum confidence
label = erase(char(labels(index)),'_label');
text = cell(2,1);
text{1} = ['Classification: ' label];
text{2} = ['Confidence: ' sprintf('%0.2f',confidence) '%'];
position = [135 20 0 0; 130 40 0 0];
final_image = insertObjectAnnotation(ecg_image,'rectangle',position,...
    text,'TextBoxOpacity',0.9,'FontSize',9);

end
```

Run the Simulation

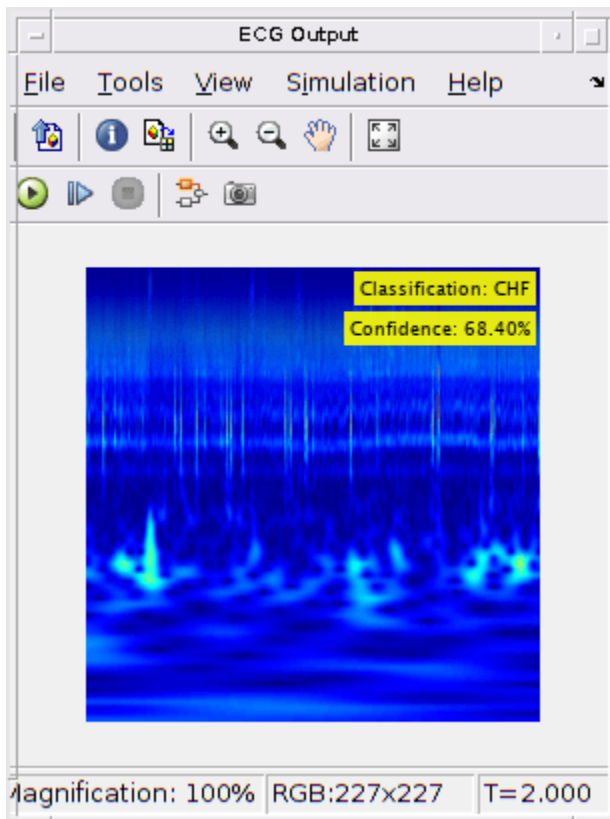
Open Configuration Parameters dialog box.

In **Simulation Target** pane, select **GPU acceleration**. In the **Deep Learning** group, select the target library as **cuDNN**.



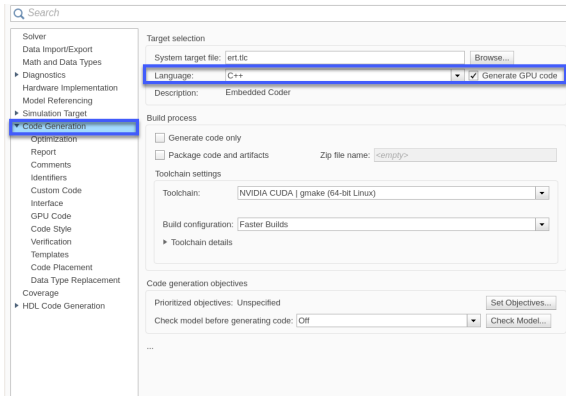
To verify the algorithm and display the labels and confidence score of the test ECG signal loaded in the workspace, run the simulation.

```
set_param('ecg_dl_cwt', 'SimulationMode', 'Normal');
sim('ecg_dl_cwt');
```

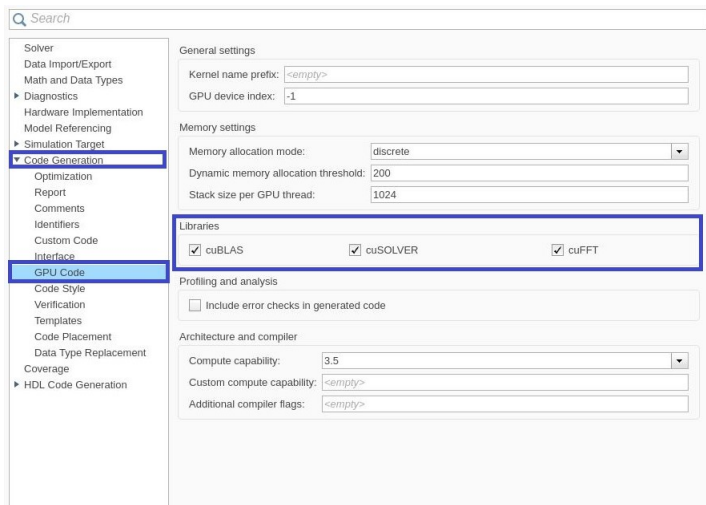


Generate and Build the Simulink Model

In **Code Generation** pane, select the **Language** as **C++** and enable **Generate GPU code**.



Open **Code Generation > GPU Code** pane. In the subcategory **Libraries**, enable **cuBLAS**, **cuSOLVER** and **cuFFT**.



Generate and build the Simulink model on the host GPU by using the `slbuild` command. The code generator places the files in a *build folder*, a subfolder named `ecg_dl_cwt_ert_rtw` under your current working folder.

```
status = evalc("slbuild('ecg_dl_cwt')");
```

Generated CUDA® Code

The subfolder named `ecg_dl_cwt_ert_rtw` contains the generated C++ codes corresponding to the different blocks in the Simulink model and the specific operations being performed in those blocks. For example, the file `trainedNet0_ecg_dl_cwt0.h` contains the C++ class which contains certain attributes such as `numLayers` and member functions such as `getBatchSize()`, `predict()`. This class represents the pretrained SqueezeNet which has been loaded in the Simulink model.

```

#ifdef RTW_HEADER_trainedNet0_ecg_dl_cwt0_h_
#define RTW_HEADER_trainedNet0_ecg_dl_cwt0_h_
#include "rtwtypes.h"
#include "MwTargetNetworkImpl.hpp"
#include "MwElementwiseAffineLayer.hpp"
#include "cnn_api.hpp"
#include "MwFusedConvReLULayer.hpp"
#include "MwConcatenationLayer.hpp"
#include "MwKernelHeaders.hpp"
#include "MwCustomLayerForCUDNN.hpp"

class trainedNet0_ecg_dl_cwt0
{
public:
    int32_T numLayers;
private:
    MwTensorBase *inputTensors;
    MwTensorBase *outputTensors;
public:
    MwCNLayer *Layers[42];
private:
    MwTargetNetworkImpl *targetImpl;
    void allocate();
    void postsetup();
public:
    trainedNet0_ecg_dl_cwt0();
private:
    void deallocate();
public:
    void setSize();
    void resetState();
    void setup();
    void predict();
    void cleanup();
    real32_T *getLayerOutput(int32_T layerIndex, int32_T portIndex);
    real32_T *getInputDataPointer(int32_T index);
    real32_T *getInputDataPointer();
    real32_T *getOutputDataPointer(int32_T index);
    real32_T *getOutputDataPointer();
    int32_T getBatchSize();
    ~trainedNet0_ecg_dl_cwt0();
};

#endif // RTW_HEADER_trainedNet0_ecg_dl_cwt0_h_

```

Cleanup

Close the Simulink model.

```

close_system('ecg_dl_cwt/ECG Preprocessing');
close_system('ecg_dl_cwt');

```

Code Generation for Lidar Point Cloud Segmentation Network

This example shows how to generate CUDA® MEX code for a deep learning network for lidar semantic segmentation. This example uses a pretrained SqueezeSegV2 [1] network that can segment organized lidar point clouds belonging to three classes (*background*, *car*, and *truck*). For information on the training procedure for the network, see “Lidar Point Cloud Semantic Segmentation Using SqueezeSegV2 Deep Learning Network” (Lidar Toolbox). The generated MEX code takes a point cloud as input and performs prediction on the point cloud by using the `DAGNetwork` object for the SqueezeSegV2 network.

Third-Party Prerequisites

Required

This example generates CUDA MEX and has the following third-party requirements.

- CUDA enabled NVIDIA® GPU and compatible driver.

Optional

For non-MEX builds such as static, dynamic libraries or executables, this example has the following additional requirements.

- NVIDIA toolkit.
- NVIDIA cuDNN library.
- NVIDIA TensorRT library.
- Environment variables for the compilers and libraries. For details, see “Third-Party Hardware” and “Setting Up the Prerequisite Products”.

Verify GPU Environment

To verify that the compilers and libraries for running this example are set up correctly, use the `coder.checkGpuInstall` function.

```
envCfg = coder.gpuEnvConfig('host');  
envCfg.DeepLibTarget = 'cudnn';  
envCfg.DeepCodegen = 1;  
envCfg.Quiet = 1;  
coder.checkGpuInstall(envCfg);
```

Segmentation Network

SqueezeSegV2 is a convolutional neural network (CNN) designed for the semantic segmentation of organized lidar point clouds. It is a deep encoder-decoder segmentation network trained on a lidar data set and imported into MATLAB® for inference. In SqueezeSegV2, the encoder subnetwork consists of convolution layers that are interspersed with max-pooling layers. This arrangement successively decreases the resolution of the input image. The decoder subnetwork consists of a series of transposed convolution layers, which successively increase the resolution of the input image. In addition, the SqueezeSegV2 network mitigates the impact of missing data by including context aggregation modules (CAMs). A CAM is a convolutional subnetwork with `filterSize` of value [7, 7] that aggregates contextual information from a larger receptive field, which improves the robustness of the network to missing data. The SqueezeSegV2 network in this example is trained to segment points belonging to three classes (background, car, and truck).

For more information on training a semantic segmentation network in MATLAB® by using the Mathworks lidar dataset, see “Lidar Point Cloud Semantic Segmentation Using PointSeg Deep Learning Network” (Lidar Toolbox).

Download the pretrained SqueezeSegV2 Network.

```
net = getSqueezeSegV2Net();
```

Downloading pretrained SqueezeSegV2 (2 MB)...

The DAG network contains 238 layers, including convolution, ReLU, and batch normalization layers, and a focal loss output layer. To display an interactive visualization of the deep learning network architecture, use the `analyzeNetwork` (Deep Learning Toolbox) function.

```
analyzeNetwork(net);
```

squeezesegv2_predict Entry-Point Function

The `squeezesegv2_predict.m` entry-point function, which is attached to this example, takes a point cloud as input and performs prediction on it by using the deep learning network saved in the `SqueezeSegV2Net.mat` file. The function loads the network object from the `SqueezeSegV2Net.mat` file into a persistent variable `mynet` and reuses the persistent variable in subsequent prediction calls.

```
type('squeezesegv2_predict.m');
```

```
function out = squeezesegv2_predict(in)
%#codegen
```

```
% A persistent object mynet is used to load the DAG network object. At
% the first call to this function, the persistent object is constructed and
% setup. When the function is called subsequent times, the same object is
% reused to call predict on inputs, thus avoiding reconstructing and
% reloading the network object.
```

```
% Copyright 2020 The MathWorks, Inc.
```

```
persistent mynet;
```

```
if isempty(mynet)
    mynet = coder.loadDeepLearningNetwork('SqueezeSegV2Net.mat');
end
```

```
% pass in input
out = predict(mynet,in);
```

Generate CUDA MEX Code

To generate CUDA MEX code for the `squeezesegv2_predict.m` entry-point function, create a GPU code configuration object for a MEX target and set the target language to C++. Use the `coder.DeepLearningConfig` function to create a CuDNN deep learning configuration object and assign it to the `DeepLearningConfig` property of the GPU code configuration object. Run the `codegen` command, specifying an input size of [64, 1024, 5]. This value corresponds to the size of the input layer of the SqueezeSegV2 network.

```
cfg = coder.gpuConfig('mex');
cfg.TargetLang = 'C++';
```

```
cfg.DeepLearningConfig = coder.DeepLearningConfig('cudnn');
codegen -config cfg squeezesegv2_predict -args {ones(64,1024,5,'uint8')} -report
```

Code generation successful: [View report](#)

To generate CUDA C++ code that takes advantage of NVIDIA TensorRT libraries, in the code, specify `coder.DeepLearningConfig('tensorrt')` instead of `coder.DeepLearningConfig('cudnn')`.

For information on how to generate MEX code for deep learning networks on Intel® processors, see “Code Generation for Deep Learning Networks with MKL-DNN”.

Prepare Data

Load an organized test point cloud in MATLAB®. Convert the point cloud to a five-channel image for prediction.

```
ptCloud = pcread('ousterLidarDrivingData.pcd');
I = pointCloudToImage(ptCloud);
```

```
% Examine converted data
whos I
```

Name	Size	Bytes	Class	Attributes
I	64x1024x5	327680	uint8	

The image has five channels. The (x,y,z) point coordinates comprise the first three channels. The fourth channel contains the lidar intensity measurement. The fifth channel contains the range information, which is computed as $r = \sqrt{x^2 + y^2 + z^2}$.

Visualize intensity channel of the image.

```
intensityChannel = I(:,:,4);

figure;
imshow(intensityChannel);
title('Intensity Image');
```



Run Generated MEX on Data

Call `squeezesegv2_predict_mex` on the five-channel image.

```
predict_scores = squeezesegv2_predict_mex(I);
```

The `predict_scores` variable is a three-dimensional matrix that has three channels corresponding to the pixel-wise prediction scores for every class. Compute the channel by using the maximum prediction score to get the pixel-wise labels

```
[~,argmax] = max(predict_scores,[],3);
```


Overlay the segmented labels on the intensity channel image and display the segmented region. Resize the segmented output and add a colorbar for better visualization.

```

classes = [
    "background"
    "car"
    "truck"
];

cmap = lidarColorMap();
SegmentedImage = labeloverlay(intensityChannel, argmax, 'ColorMap', cmap);
SegmentedImage = imresize(SegmentedImage, 'Scale', [2 1], 'method', 'nearest');
figure;
imshow(SegmentedImage);

N = numel(classes);
ticks = 1/(N*2):1/N:1;
colorbar('TickLabels', cellstr(classes), 'Ticks', ticks, 'TickLength', 0, 'TickLabelInterpreter', 'none');
colormap(cmap);
title('Semantic Segmentation Result');

```



Run Generated MEX Code on Point Cloud Sequence

Read an input point cloud sequence. The sequence contains 10 organized pointCloud frames collected using an Ouster OS1 lidar sensor. The input data has a height of 64 and a width of 1024, so each pointCloud object is of size 64-by-1024.

```
dataFile = 'highwaySceneData.mat';
```

```
% Load data in workspace.
load(dataFile);
```

Setup different colors to visualize point-wise labels for different classes of interest.

```
% Apply the color red to cars.
carClassColor = zeros(64, 1024, 3, 'uint8');
carClassColor(:,:,1) = 255*ones(64, 1024, 'uint8');
```

```
% Apply the color blue to trucks.
truckClassColor = zeros(64, 1024, 3, 'uint8');
truckClassColor(:,:,3) = 255*ones(64, 1024, 'uint8');
```

```
% Apply the color gray to background.
backgroundClassColor = 153*ones(64, 1024, 3, 'uint8');
```

Set the pcplayer function properties to display the sequence and the output predictions. Read the input sequence frame by frame and detect classes of interest using the model.

```
xlimits = [0 120.0];
ylimits = [-80.7 80.7];
zlimits = [-8.4 27];

player = pcplayer(xlimits, ylimits, zlimits);
set(get(player.Axes, 'parent'), 'units', 'normalized', 'outerposition', [0 0 1 1]);
zoom(get(player.Axes, 'parent'), 2);
set(player.Axes, 'XColor', 'none', 'YColor', 'none', 'ZColor', 'none');

for i = 1 : numel(inputData)
    ptCloud = inputData{i};

    % Convert point cloud to five-channel image for prediction.
    I = pointCloudToImage(ptCloud);

    % Call squeezesegv2_predict_mex on the 5-channel image.
    predict_scores = squeezesegv2_predict_mex(I);

    % Convert the numeric output values to categorical labels.
    [~, predictedOutput] = max(predict_scores, [], 3);
    predictedOutput = categorical(predictedOutput, 1:3, classes);

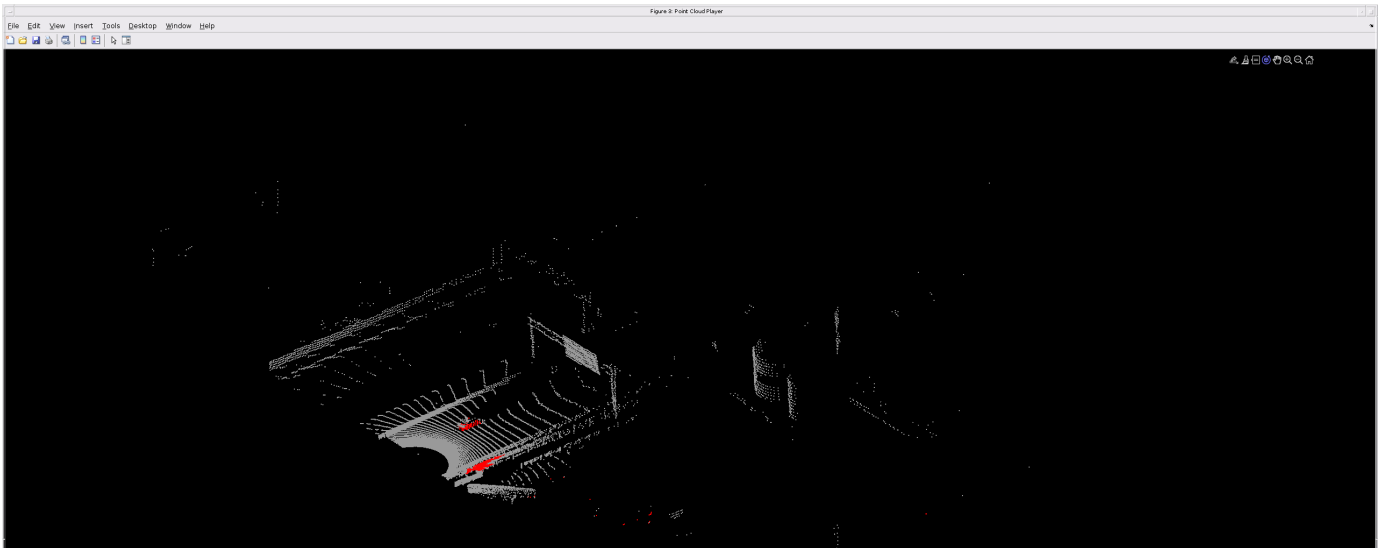
    % Extract the indices from labels.
    carIndices = predictedOutput == 'car';
    truckIndices = predictedOutput == 'truck';
    backgroundIndices = predictedOutput == 'background';

    % Extract a point cloud for each class.
    carPointCloud = select(ptCloud, carIndices, 'OutputSize', 'full');
    truckPointCloud = select(ptCloud, truckIndices, 'OutputSize', 'full');
    backgroundPointCloud = select(ptCloud, backgroundIndices, 'OutputSize', 'full');

    % Fill the colors to different classes.
    carPointCloud.Color = carClassColor;
    truckPointCloud.Color = truckClassColor;
    backgroundPointCloud.Color = backgroundClassColor;

    % Merge and add all the processed point clouds with class information.
    coloredCloud = pcmerge(carPointCloud, truckPointCloud, 0.01);
    coloredCloud = pcmerge(coloredCloud, backgroundPointCloud, 0.01);

    % View the output.
    view(player, coloredCloud);
    drawnow;
end
```



Helper Functions

The helper functions used in this example follow.

type `pointCloudToImage.m`

```
function image = pointCloudToImage(ptcloud)
%pointCloudToImage Converts organized 3-D point cloud to 5-channel
% 2-D image.
```

```
image = ptcloud.Location;
image(:,:,4) = ptcloud.Intensity;
rangeData = iComputeRangeData(image(:,:,1),image(:,:,2),image(:,:,3));
image(:,:,5) = rangeData;
```

```
% Cast to uint8.
image = uint8(image);
end
```

```
%-----
function rangeData = iComputeRangeData(xChannel,yChannel,zChannel)
rangeData = sqrt(xChannel.*xChannel+yChannel.*yChannel+zChannel.*zChannel);
end
```

type `lidarColorMap.m`

```
function cmap = lidarColorMap()

cmap = [
    0.00  0.00  0.00  % background
    0.98  0.00  0.00  % car
    0.00  0.00  0.98  % truck
];
end
```

References

[1] Wu, Bichen, Xuanyu Zhou, Sicheng Zhao, Xiangyu Yue, and Kurt Keutzer. "SqueezeSegV2: Improved Model Structure and Unsupervised Domain Adaptation for Road-Object Segmentation from a LiDAR Point Cloud." Preprint, submitted September 22, 2018. <http://arxiv.org/abs/1809.08495>.

See Also

Functions

`coder.checkGpuInstall` | `codegen` | `coder.DeepLearningConfig` |
`coder.loadDeepLearningNetwork`

Objects

`coder.gpuConfig` | `coder.gpuEnvConfig` | `coder.TensorRTConfig` | `coder.CuDNNConfig`

Related Examples

- "Lidar Point Cloud Semantic Segmentation Using PointSeg Deep Learning Network" (Lidar Toolbox)

Code Generation for a Video Classification Network

This example shows how to generate CUDA® code for a deep learning network that classifies video and deploy the generated code onto the NVIDIA® Jetson® Xavier board by using the MATLAB® Coder™ Support Package for NVIDIA Jetson and NVIDIA DRIVE Platforms. The deep learning network has both convolutional and bidirectional long short-term memory (BiLSTM) layers. The generated application reads the data from a specified video file as a sequence of video frames and outputs a label that classifies the activity in the video. This example generates code for the network trained in the *Classify Videos Using Deep Learning* example from the Deep Learning Toolbox™. For more information, see “Classify Videos Using Deep Learning” (Deep Learning Toolbox).

Third-Party Prerequisites

Target Board Requirements

- NVIDIA Jetson board.
- Ethernet crossover cable to connect the target board and host PC (if the target board cannot be connected to a local network).
- Supported Jetpack SDK that includes CUDA and cuDNN libraries
- Environment variables on the target for the compilers and libraries. For information on the supported versions of the compilers and libraries and their setup, see “Install and Setup Prerequisites for NVIDIA Boards” (MATLAB Coder Support Package for NVIDIA Jetson and NVIDIA DRIVE Platforms) for NVIDIA boards.

Verify NVIDIA Support Package Installation on Host

To generate and deploy code to an NVIDIA Jetson Xavier board, you will need the MATLAB Coder Support Package for NVIDIA Jetson and NVIDIA DRIVE Platforms. Use the `checkHardwareSupportPackageInstall` function to verify that the host system is compatible to run this example. If the function does not throw an error, the support package is correctly installed.

```
checkHardwareSupportPackageInstall();
```

Connect to the NVIDIA Hardware

The support package uses an SSH connection over TCP/IP to execute commands while building and running the generated CUDA code on the Jetson platform. You must therefore connect the target platform to the same network as the host computer or use an Ethernet crossover cable to connect the board directly to the host computer. Refer to the NVIDIA documentation on how to set up and configure your board.

To communicate with the NVIDIA hardware, you must create a live hardware connection object by using the `jetson` (MATLAB Coder Support Package for NVIDIA Jetson and NVIDIA DRIVE Platforms) function. You must know the host name or IP address, username, and password of the target board to create a live hardware connection object. For example, when connecting to the target board for the first time, create a live object for Jetson hardware by using the command:

```
hwobj = jetson('jetson-name', 'ubuntu', 'ubuntu');
```

The `jetson` object reuses these settings from the most recent successful connection to the Jetson hardware. This example establishes an SSH connection to the Jetson hardware using the settings stored in memory.

```
hwobj = jetson;
```

```
Checking for CUDA availability on the Target...
Checking for 'nvcc' in the target system path...
Checking for cuDNN library availability on the Target...
Checking for TensorRT library availability on the Target...
Checking for prerequisite libraries is complete.
Gathering hardware details...
Checking for third-party library availability on the Target...
Gathering hardware details is complete.
Board name      : NVIDIA Jetson TX2
CUDA Version    : 10.0
cuDNN Version   : 7.6
TensorRT Version : 6.0
GStreamer Version : 1.14.5
V4L2 Version    : 1.14.2-1
SDL Version     : 1.2
OpenCV Version  : 4.1.1
Available Webcams : Microsoft® LifeCam Cinema(TM)
Available GPUs  : NVIDIA Tegra X2
```

NOTE:

In case of a connection failure, a diagnostics error message is reported on the MATLAB command line. If the connection has failed, the most likely cause is incorrect IP address or hostname.

Verify GPU Environment

Use the `coder.checkGpuInstall` function to verify that the compilers and libraries necessary for running this example are set up correctly.

```
envCfg = coder.gpuEnvConfig('jetson');
envCfg.DeepLibTarget = 'cudnn';
envCfg.DeepCodegen = 1;
envCfg.Quiet = 1;
envCfg.HardwareObject = hwobj;
coder.checkGpuInstall(envCfg);
```

The net_classify Entry-Point Function

The `net_classify` entry-point function hardcodes the name of a video file. Note that this hardcoded path must be adjusted to the location of the video file on your target hardware. The entry-point function then reads the data from the file using a `VideoReader` object. The data is read into MATLAB as a sequence of images (video frames). This data is then center-cropped, and finally passed as input to a trained network for prediction. Specifically, the function uses the network trained in the *Classify Videos Using Deep Learning* example. The function loads the network object from the `net.mat` file into a persistent variable and reuses the persistent object for subsequent prediction calls.

```
type('net_classify.m')

function out = net_classify() %#codegen

if coder.target('MATLAB')
    videoFilename = 'situp.mp4';
else
    videoFilename = '/home/ubuntu/VideoClassify/situp.mp4';
end
```

```

frameSize = [1920 1080];

% read video
video = readVideo(videoFilename, frameSize);

% specify network input size
inputSize = [224 224 3];

% crop video
croppedVideo = centerCrop(video, inputSize);

% A persistent object mynet is used to load the series network object. At
% the first call to this function, the persistent object is constructed and
% setup. When the function is called subsequent times, the same object is
% reused to call predict on inputs, thus avoiding reconstructing and
% reloading the network object.

persistent mynet;

if isempty(mynet)
    mynet = coder.loadDeepLearningNetwork('net.mat');
end

% pass in cropped input to network
out = classify(mynet, croppedVideo);

% Copyright 2019-2021 The MathWorks, Inc.

```

About the Network

The network used to classify video input has a few notable features:

1. The network has a sequence input layer to accept images sequences as input.
2. The network uses a sequence folding layer followed by convolutional layers to apply the convolutional operations to each video frame independently, thereby extracting features from each frame.
3. The network uses a sequence unfolding layer and a flatten layer to restore the sequence structure and reshape the output to vector sequences, in anticipation of the BiLSTM layer.
4. Finally, the network uses the BiLSTM layer followed by output layers to classify the vector sequences.

To display an interactive visualization of the network architecture and information about the network layers, use the `analyzeNetwork` (Deep Learning Toolbox) function.

Run `net_classify` in MATLAB

Download the video classification network.

```
getVideoClassificationNetwork();
```

Loop over the individual frames of `situp.mp4` to view the test video in MATLAB.

```
videoFileName = 'situp.mp4';
video = readVideo(videoFileName);
```

```

numFrames = size(video,4);
figure
for i = 1:numFrames
    frame = video(:,:, :, i);
    imshow(frame/255);
    drawnow
end

```



Run `net_classify` and note the output label. Note that if there is a host GPU available, it will be automatically used when running `net_classify`.

```

net_classify()

ans = categorical
      situp

```

Generate & Deploy CUDA Code on the Target

To generate a CUDA executable that can be deployed to an NVIDIA target, create a new GPU coder configuration object for generating an executable. Set the target deep learning library to 'cudnn'.

```

clear cfg
cfg = coder.gpuConfig('exe');
cfg.DeepLearningConfig = coder.DeepLearningConfig('cudnn');

```

Use the `coder.hardware` function to create a configuration object for the Jetson platform and assign it to the `Hardware` property of the GPU code configuration object `cfg`.

```

cfg.Hardware = coder.hardware('NVIDIA Jetson');

```


Set the build directory on the target hardware. Change the example path below to the location on your target hardware where you would like the generated code to be placed.

```
cfg.Hardware.BuildDir = '/home/ubuntu/VideoClassify';
```

The custom main file `main.cu` is a wrapper that calls the `net_classify` function in the generated library.

```
cfg.CustomInclude = '.';
cfg.CustomSource = fullfile('main.cu');
```

Run the `codegen` command. This time, code will be generated and then copied over to the target board. The executable will then be built on the target board.

```
codegen -config cfg net_classify
```

```
Code generation successful.
```

Run the Generated Application on the Target

Copy the test video file `situp.mp4` from the host computer to the target device by using the `putFile` command. Ensure that this video file is placed in the location hardcoded in the entry-point function `net_classify`. In this example, this location happens to be the target hardware build directory.

```
putFile(hwobj,videoFileName, cfg.Hardware.BuildDir);
```

Use `runApplication` to launch the application on the target hardware. The label will be displayed in the output terminal on the target.

```
status = evalc("runApplication(hwobj,'net_classify')");
```

See Also

Functions

`coder.checkGpuInstall` | `codegen` | `coder.DeepLearningConfig` |
`coder.loadDeepLearningNetwork` | `jetson` | `runApplication` | `killApplication` |
`VideoReader`

Objects

`coder.gpuConfig` | `coder.gpuEnvConfig` | `coder.CuDNNConfig` | `coder.TensorRTConfig` |
`jetson`

Related Examples

- “Classify Videos Using Deep Learning” (Deep Learning Toolbox)
- “Getting Started with the MATLAB Coder Support Package for NVIDIA Jetson and NVIDIA DRIVE Platforms” (MATLAB Coder Support Package for NVIDIA Jetson and NVIDIA DRIVE Platforms)

More About

- “Long Short-Term Memory Networks” (Deep Learning Toolbox)
- “Build and Run an Executable on NVIDIA Hardware” (MATLAB Coder Support Package for NVIDIA Jetson and NVIDIA DRIVE Platforms)

- “Stop or Restart an Executable Running on NVIDIA Hardware” (MATLAB Coder Support Package for NVIDIA Jetson and NVIDIA DRIVE Platforms)
- “Run Linux Commands on NVIDIA Hardware” (MATLAB Coder Support Package for NVIDIA Jetson and NVIDIA DRIVE Platforms)

Code Generation For Object Detection Using YOLO v3 Deep Learning

This example shows how to generate CUDA® MEX for a you only look once (YOLO) v3 object detector. YOLO v3 improves upon YOLO v2 by adding detection at multiple scales to help detect smaller objects. Moreover, the loss function used for training is separated into mean squared error for bounding box regression and binary cross-entropy for object classification to help improve detection accuracy. The YOLO v3 network used in this example was trained from the *Object Detection Using YOLO v3 Deep Learning* example in the Computer Vision Toolbox (TM). For more information, see “Object Detection Using YOLO v3 Deep Learning” (Computer Vision Toolbox).

Third-Party Prerequisites

Required

- CUDA enabled NVIDIA® GPU and compatible driver.

Optional

For non-MEX builds such as static, dynamic libraries or executables, this example has the following additional requirements.

- NVIDIA CUDA toolkit.
- NVIDIA cuDNN library.
- Environment variables for the compilers and libraries. For more information, see “Third-Party Hardware” and “Setting Up the Prerequisite Products”.

Verify GPU Environment

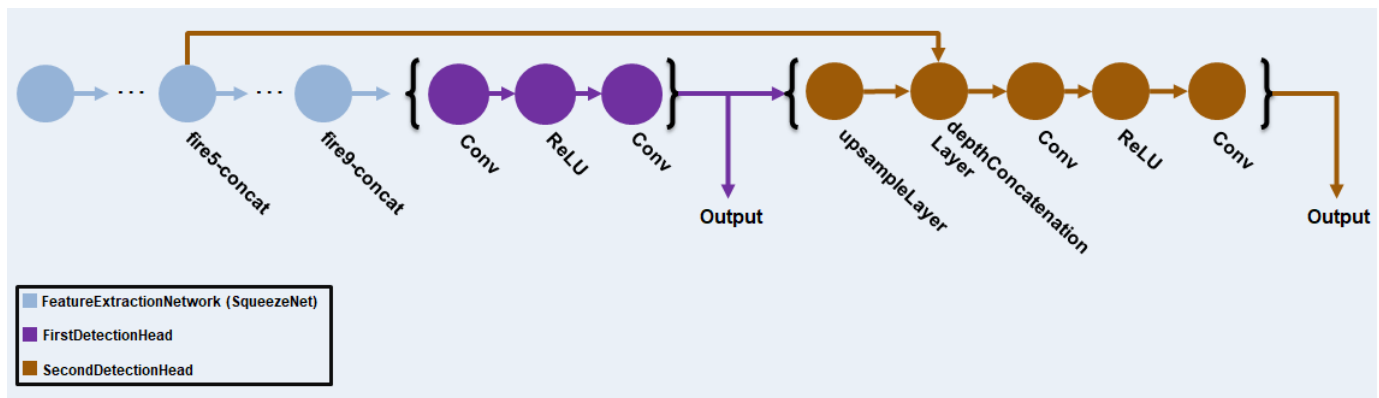
To verify that the compilers and libraries for running this example are set up correctly, use the `coder.checkGpuInstall` function.

```
envCfg = coder.gpuEnvConfig('host');  
envCfg.DeepLibTarget = 'cudnn';  
envCfg.DeepCodegen = 1;  
envCfg.Quiet = 1;  
coder.checkGpuInstall(envCfg);
```

YOLO v3 Network

The YOLO v3 network in this example is based on `squeezenet` (Deep Learning Toolbox), and uses the feature extraction network in `SqueezeNet` with the addition of two detection heads at the end. The second detection head is twice the size of the first detection head, so it is better able to detect small objects. Note that any number of detection heads of different sizes can be specified based on the size of the objects to be detected. The YOLO v3 network uses anchor boxes estimated using training data to have better initial priors corresponding to the type of data set and to help the network learn to predict the boxes accurately. For information about anchor boxes, see “Anchor Boxes for Object Detection” (Computer Vision Toolbox).

The YOLO v3 network in this example is illustrated in the following diagram.



Each detection head predicts the bounding box coordinates (x, y, width, height), object confidence, and class probabilities for the respective anchor box masks. Therefore, for each detection head, the number of output filters in the last convolution layer is the number of anchor box mask times the number of prediction elements per anchor box. The detection heads comprise the output layer of the network.

Pretrained YOLO v3 Network

This example uses the `yolov3SqueezeNetVehicleExample_21aSPKG.zip` file containing the pretrained YOLO v3 network. The file is approximately 23 MB in size. Download the file from the MathWorks website, then unzip the file.

```
fileName = matlab.internal.examples.downloadSupportFile('vision/data/', 'yolov3SqueezeNetVehicleExample_21aSPKG.zip');
data = unzip(fileName);
matFile = data{1,1};
vehicleDetector = load(matFile);
net = vehicleDetector.detector.Network
```

```
net =
  dlnetwork with properties:
    Layers: [75x1 nnet.cnn.layer.Layer]
    Connections: [84x2 table]
    Learnables: [66x3 table]
    State: [6x3 table]
    InputNames: {'data'}
    OutputNames: {'customOutputConv1' 'customOutputConv2'}
    Initialized: 1
```

Note: You can also use the pretrained detector network available through the Computer Vision Toolbox™ Model for YOLO v3 Object Detection support package.

To use this pretrained network, you must first install the Computer Vision Toolbox Model for YOLO v3 Object Detection from the Add-On Explorer. For more information about installing add-ons, see “Get and Manage Add-Ons”.

Then, save the `yolov3ObjectDetector` object to a MAT-file and proceed. For example,

```
detector = yolov3ObjectDetector('darknet53-coco');
matFile = 'pretrainedYOLOv3Detector.mat';
save(matFile, 'detector');
```

The yolov3Detect Entry-Point Function

The `yolov3Detect` entry-point function takes an image input and runs the detector on the image using the deep learning network saved in the `yolov3SqueezeNetVehicleExample_21aSPKG.mat` file. The function loads the network object from the `yolov3SqueezeNetVehicleExample_21aSPKG.mat` file into a persistent variable `yolov3Obj` and reuses the persistent object on subsequent detection calls.

```
type('yolov3Detect.m')

function outImg = yolov3Detect(in,matFile)

% Copyright 2021 The MathWorks, Inc.

persistent yolov3Obj;

if isempty(yolov3Obj)
    yolov3Obj = coder.loadDeepLearningNetwork(matFile);
end

% Call to detect method
[bboxes,~,labels] = yolov3Obj.detect(in,'Threshold',0.5);

% Convert categorical labels to cell array of character vectors
labels = cellstr(labels);

% Annotate detections in the image.
outImg = insertObjectAnnotation(in,'rectangle',bboxes,labels);
```

Generate CUDA MEX

To generate CUDA code for the entry-point function, create a GPU code configuration object for a MEX target and set the target language to C++. Use the `coder.DeepLearningConfig` function to create a CuDNN deep learning configuration object and assign it to the `DeepLearningConfig` property of the GPU code configuration object. Run the `codegen` command specifying an input size of 227-by-227-by-3. This value corresponds to the input layer size of YOLOv3.

```
cfg = coder.gpuConfig('mex');
cfg.TargetLang = 'C++';
cfg.DeepLearningConfig = coder.DeepLearningConfig('cudnn');
cfg.GenerateReport = true;
inputArgs = {ones(227,227,3,'uint8'),coder.Constant(matFile)};

codegen -config cfg yolov3Detect -args inputArgs -report
```

Code generation successful: [View report](#)

To generate CUDA® code for TensorRT target create and use a TensorRT deep learning configuration object instead of the CuDNN configuration object. Similarly, to generate code for MKLDNN target, create a CPU code configuration object and use MKLDNN deep learning configuration object as its `DeepLearningConfig` property.

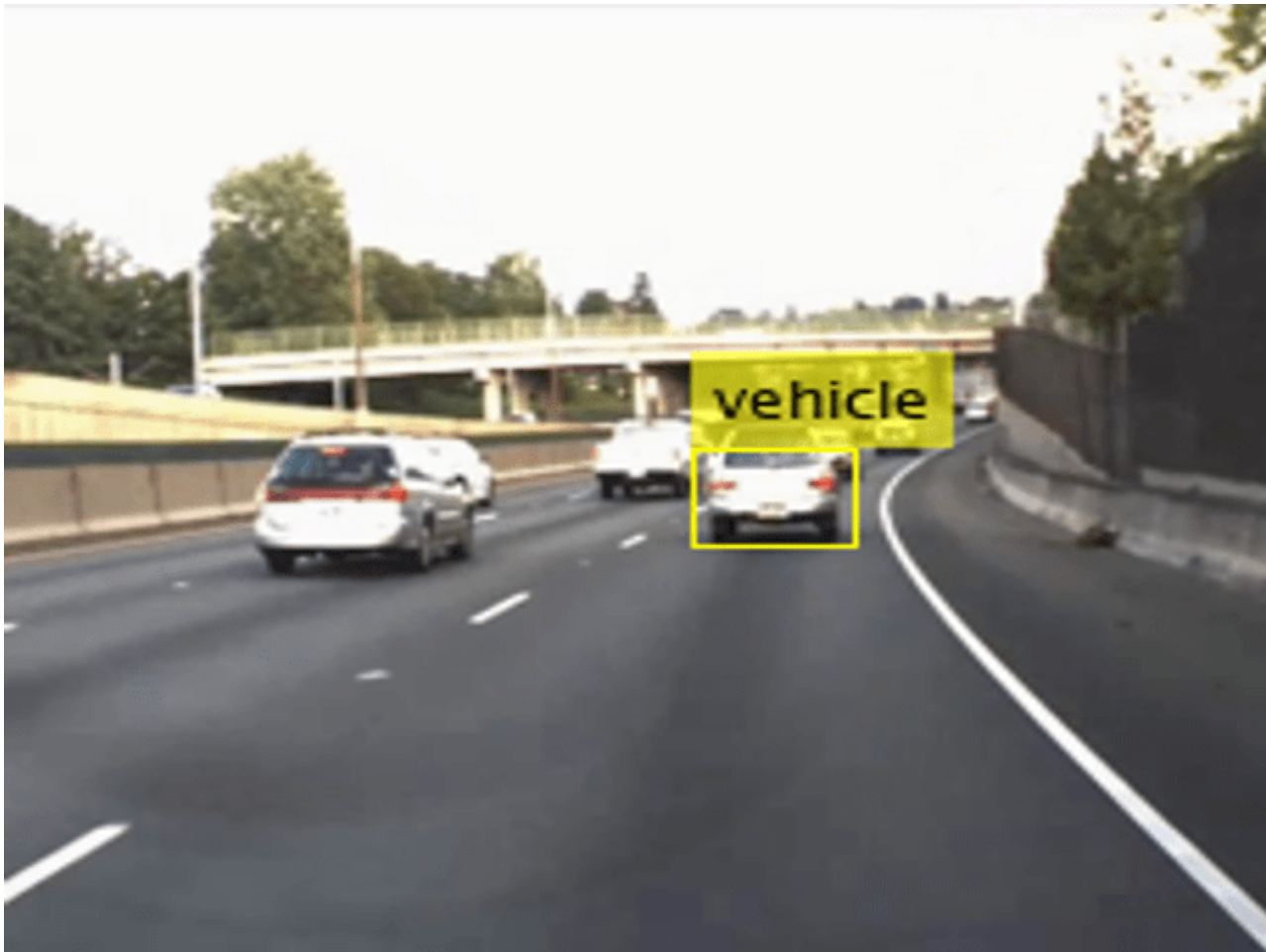
Run the Generated MEX

Set up the video file reader and read the input video. Create a video player to display the video and the output detections.

```
videoFile = 'highway_lanechange.mp4';  
videoFreader = vision.VideoFileReader(videoFile,'VideoOutputDataType','uint8');  
depVideoPlayer = vision.DeployableVideoPlayer('Size','Custom','CustomSize',[640 480]);
```

Read the video input frame-by-frame and detect the vehicles in the video using the detector.

```
cont = ~isDone(videoFreader);  
while cont  
    I = step(videoFreader);  
    in = imresize(I,[227,227]);  
    out = yolov3Detect_mex(in,matFile);  
    step(depVideoPlayer, out);  
    % Exit the loop if the video player figure window is closed  
    cont = ~isDone(videoFreader) && isOpen(depVideoPlayer);  
end
```



References

1. Redmon, Joseph, and Ali Farhadi. "YOLOv3: An Incremental Improvement." Preprint, submitted April 8, 2018. <https://arxiv.org/abs/1804.02767>.

See Also

Functions

`coder.checkGpuInstall` | `codegen` | `coder.DeepLearningConfig` |
`coder.loadDeepLearningNetwork`

Objects

`coder.gpuConfig` | `coder.gpuEnvConfig` | `coder.TensorRTConfig` | `coder.CuDNNConfig` |
`dlarray` | `dlnetwork`

Related Examples

- "Object Detection Using YOLO v3 Deep Learning" (Computer Vision Toolbox)

More About

- "Code Generation for `dlarray`" on page 4-52
- "`dlarray` Limitations for Code Generation" on page 4-62

Generate Digit Images on NVIDIA GPU Using Variational Autoencoder

This example shows how to generate CUDA® MEX for a trained variational autoencoder (VAE) network. The example illustrates:

- Generation of hand-drawn digit images in the style of the MNIST data set.
- CUDA code generation for a `dlnetwork` (Deep Learning Toolbox) object representing a deep learning network.
- Use of `dlarray` (Deep Learning Toolbox) objects in code generation.

This example uses a pretrained decoder network based on the *Train Variational Autoencoder (VAE) to Generate Images* example from the Deep Learning Toolbox™. For more information, see “Train Variational Autoencoder (VAE) to Generate Images” (Deep Learning Toolbox).

Third-Party Prerequisites

Required

- CUDA enabled NVIDIA® GPU and compatible driver.

Optional

For non-MEX builds such as static, dynamic libraries or executables, this example has the following additional requirements.

- NVIDIA CUDA toolkit.
- NVIDIA cuDNN library.
- Environment variables for the compilers and libraries. For more information, see “Third-Party Hardware” and “Setting Up the Prerequisite Products”.

Verify GPU Environment

To verify that the compilers and libraries for running this example are set up correctly, use the `coder.checkGpuInstall` function.

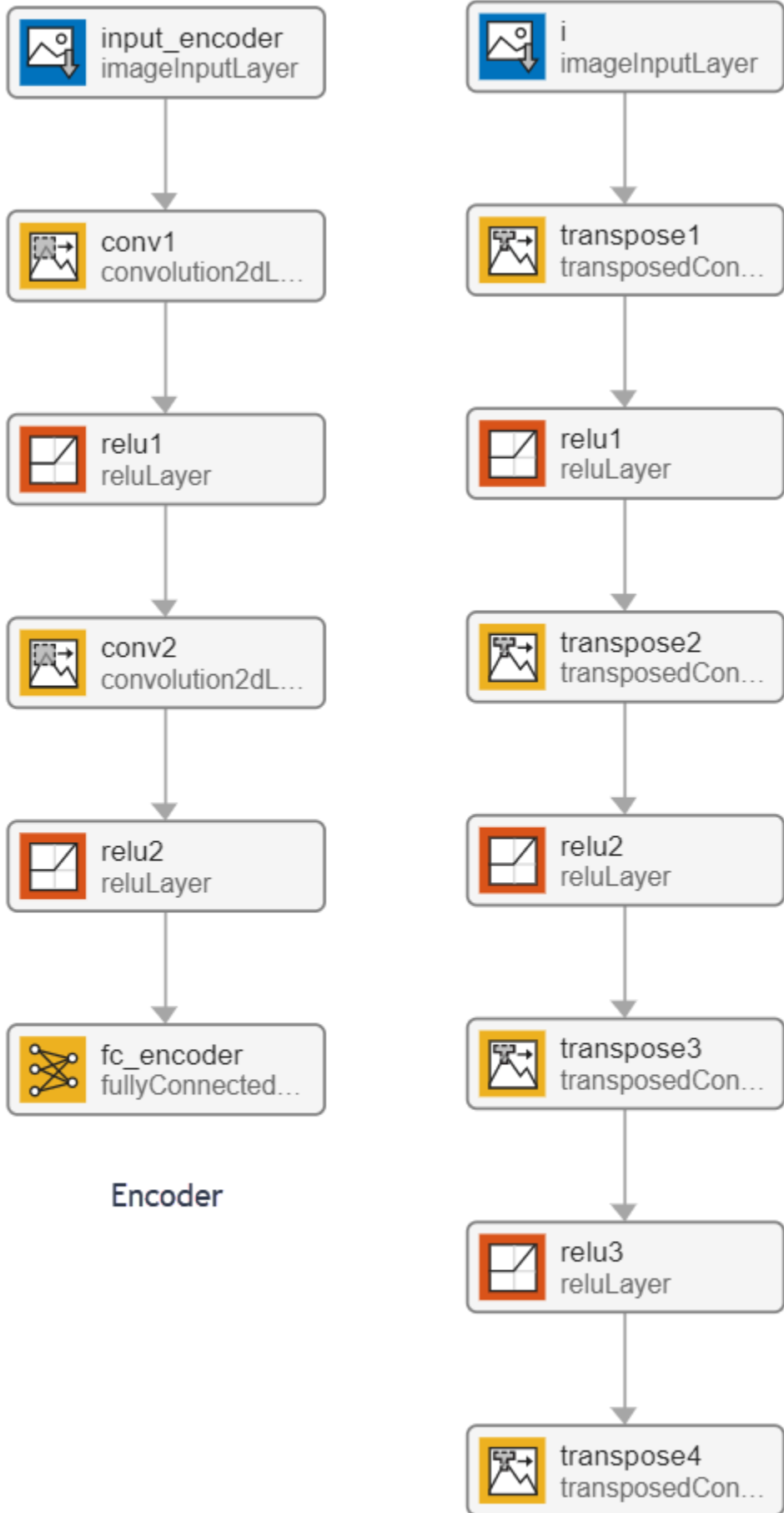
```
envCfg = coder.gpuEnvConfig('host');  
envCfg.DeepLibTarget = 'cudnn';  
envCfg.DeepCodegen = 1;  
envCfg.Quiet = 1;  
coder.checkGpuInstall(envCfg);
```

Pretrained Variational Autoencoder Network

Autoencoders have two parts: the encoder and the decoder. The encoder takes an image input and outputs a compressed representation (the encoding), which is a vector of size `latent_dim`, equal to 20 in this example. The decoder takes the compressed representation, decodes it, and recreates the original image.

VAEs differ from regular autoencoders in that they do not use the encoding-decoding process to reconstruct an input. Instead, they impose a probability distribution on the latent space, and learn the distribution so that the distribution of outputs from the decoder matches that of the observed data. Then, they sample from this distribution to generate new data.

This example uses the decoder network trained in the *Train Variational Autoencoder (VAE) to Generate Images* example. To train the network yourself, see “Train Variational Autoencoder (VAE) to Generate Images” (Deep Learning Toolbox).



The generateVAE Entry-Point Function

The `generateVAE` entry-point function loads the `dlnetwork` object from the `trainedDecoderVAENet` MAT-file into a persistent variable and reuses the persistent object for subsequent prediction calls. It initializes a `dlarray` object containing 25 randomly generated encodings, passes them through the decoder network, and extracts the numeric data of the generated image from the deep learning array object.

```
type('generateVAE.m')

function generatedImage = generateVAE(decoderNetFileName,latentDim,Environment) %#codegen
% Copyright 2020-2021 The MathWorks, Inc.

persistent decoderNet;
if isempty(decoderNet)
    decoderNet = coder.loadDeepLearningNetwork(decoderNetFileName);
end

% Generate random noise
randomNoise = dlarray(randn(1,1,latentDim,25,'single'),'SSCB');

if coder.target('MATLAB') && strcmp(Environment,'gpu')
    randomNoise = gpuArray(randomNoise);
end

% Generate new image from noise
generatedImage = sigmoid(predict(decoderNet,randomNoise));

% Extract numeric data from dlarray
generatedImage = extractdata(generatedImage);

end
```

Evaluate the Entry-Point Function

Evaluate the `generateVAE` entry-point function to generate digit images and plot the results.

```
latentDim = 20;
matfile = 'trainedDecoderVAENet.mat';
Env = '';

figure()
title("Generated samples of digits - MATLAB")

generatedImageML = generateVAE(matfile, latentDim, Env);
imshow(imtile(generatedImageML, "ThumbnailSize", [100,100]))
```



Generate CUDA MEX

To generate CUDA code for the `generateVAE` entry-point function, create a GPU code configuration object for a MEX target and set the target language to C++. Use the `coder.DeepLearningConfig` function to create a CuDNN deep learning configuration object and assign it to the `DeepLearningConfig` property of the GPU code configuration object.

```
Env = 'gpu';  
cfg = coder.gpuConfig('mex');  
cfg.TargetLang = 'C++';  
cfg.DeepLearningConfig = coder.DeepLearningConfig('cudnn');  
  
args = {coder.Constant(matfile), coder.Constant(latentDim), coder.Constant(Env)};  
  
codegen -config cfg -args args generateVAE -report
```

Code generation successful: [View report](#)

To generate CUDA code for TensorRT target, create and use a TensorRT deep learning configuration object instead of the CuDNN configuration object.

Run the Generated MEX

Call the generated CUDA MEX and display the results.

```
figure()
title("Generated samples of digits - GPU")

generatedImageGPU = generateVAE_mex(matfile, latentDim, Env);
imshow(imtile(generatedImageGPU, "ThumbnailSize", [100,100]))
```



The `generateVAE` entry-point function initializes the `dlarray` object with randomly generated encodings, passes them through the decoder network, and extracts the numeric data of the generated image from the deep learning array object. As a result, the image generated during MATLAB simulation is different from the image generated by the MEX function call.

See Also

Functions

`coder.checkGpuInstall` | `codegen` | `coder.DeepLearningConfig` |
`coder.loadDeepLearningNetwork`

Objects

`coder.gpuConfig` | `coder.gpuEnvConfig` | `coder.CuDNNConfig` | `coder.TensorRTConfig` |
`dlarray` | `dlnetwork`

Related Examples

- “Train Variational Autoencoder (VAE) to Generate Images” (Deep Learning Toolbox)

More About

- “Code Generation for `dlarray`” on page 4-52
- “`dlarray` Limitations for Code Generation” on page 4-62
- “Define Custom Training Loops, Loss Functions, and Networks” (Deep Learning Toolbox)
- “Train Network Using Custom Training Loop” (Deep Learning Toolbox)
- “Make Predictions Using `dlnetwork` Object” (Deep Learning Toolbox)

Quantize Residual Network Trained for Image Classification and Generate CUDA Code

This example shows how to quantize the learnable parameters in the convolution layers of a deep learning neural network that has residual connections and has been trained for image classification with CIFAR-10 data.

Neural networks use memory to store input data, parameters (weights), and activations from each layer as the input propagates through the network. Most neural networks that you create and train using Deep Learning Toolbox™ use single-precision floating point data types. Even small networks require a considerable amount of memory and hardware to perform these floating-point arithmetic operations. These restrictions can inhibit deployment of deep learning models to devices that have low computational power and less memory resources. By using a lower precision to store the weights and activations, you can reduce the memory requirements of the network.

In this example, you use the Deep Learning Toolbox in tandem with the Deep Learning Toolbox Model Quantization Library support package to reduce the memory footprint of a deep neural network by quantizing the weights, biases, and activations of convolution layers to 8-bit scaled integer data types.

The network in this example has been trained for image classification with CIFAR-10 data.

Residual connections are a popular element in convolutional neural network architectures. A residual network is a type of DAG network that has residual (or shortcut) connections that bypass the main network layers. Residual connections enable the parameter gradients to propagate more easily from the output layer to the earlier layers of the network, which makes it possible to train deeper networks. This increased network depth can result in higher accuracies on more difficult tasks. For information on the network architecture and training, see “Train Residual Network for Image Classification” (Deep Learning Toolbox).

To run this example, you must have the products required to quantize and deploy a deep learning network to a GPU environment. For information on these products, see “Quantization Workflow Prerequisites” (Deep Learning Toolbox).

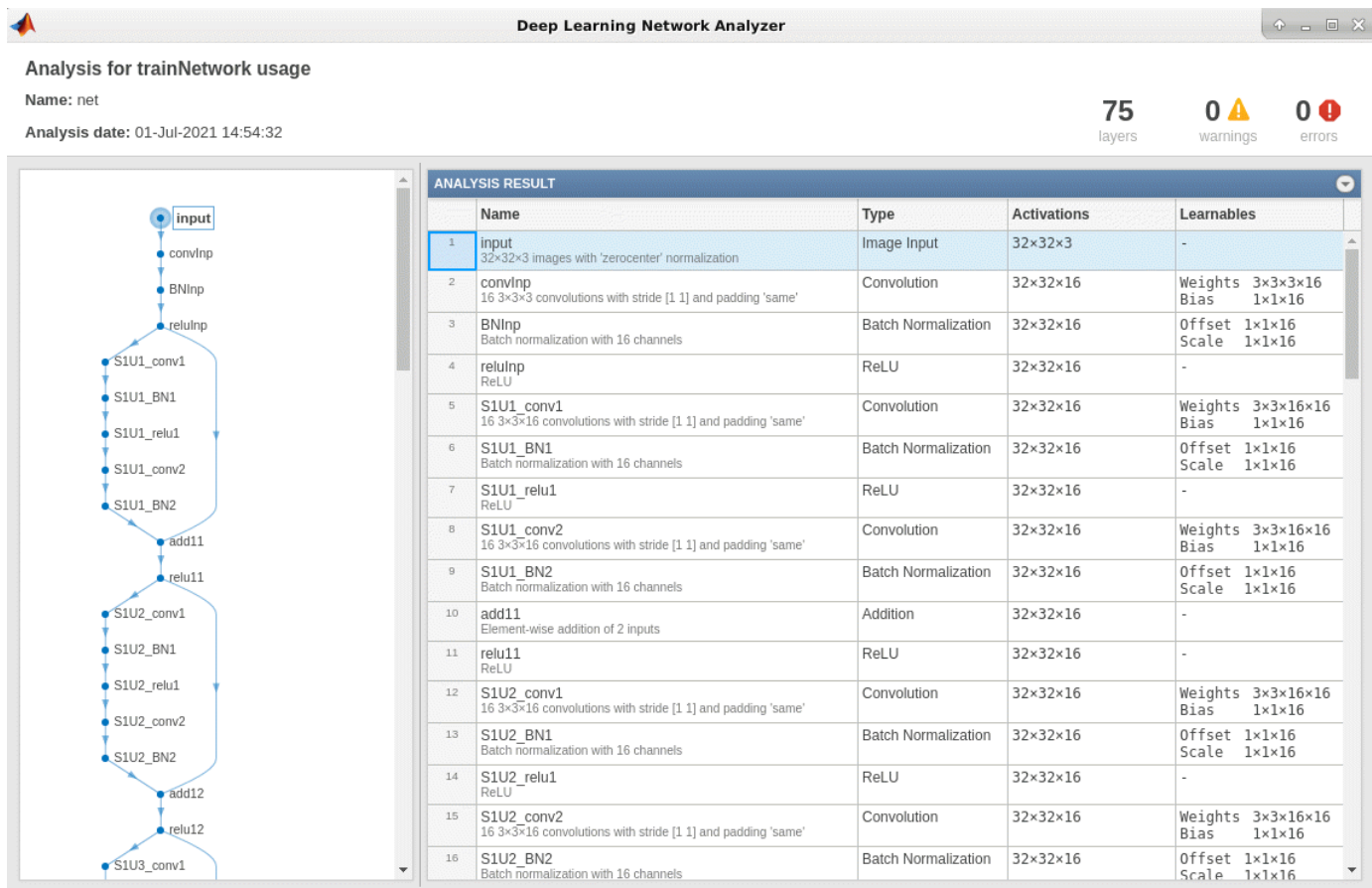
Load Pretrained Network

Load the pretrained network. For information on creating and training a network with residual connections for image classification yourself, see the *Train Residual Network for Image Classification* example.

```
load('CIFARNet-20-16.mat','trainedNet');  
net = trainedNet;
```

You can use `analyzeNetwork` to analyze the deep learning network architecture.

```
analyzeNetwork(net)
```



Load Data

Download the CIFAR-10 data set [1] by executing the code below. The data set contains 60,000 images. Each image is 32-by-32 in size and has three color channels (RGB). The size of the data set is 175 MB. Depending on your internet connection, the download process can take some time.

```
datadir = tempdir;  
downloadCIFARData(datadir);
```

Downloading CIFAR-10 dataset (175 MB). This can take a while...done.

Prepare Data for Calibration and Validation

Load the CIFAR-10 training and test images as 4-D arrays. The training set contains 50,000 images and the test set contains 10,000 images. Use the CIFAR-10 test images for network validation.

```
[XTrain,YTrain,XValidation,YValidation] = loadCIFARData(datadir);
```

You can display a random sample of the training images using the following code.

```
figure;  
idx = randperm(size(XTrain,4),20);  
im = imtile(XTrain(:,:,,idx),'ThumbnailSize',[96,96]);  
imshow(im)
```


Create an `augmentedImageDatastore` object to use for calibration and validation. Use 200 random images for calibration and 50 random images for validation.

```
inputSize = net.Layers(1).InputSize;  
  
augimdsTrain = augmentedImageDatastore(inputSize,XTrain,YTrain);  
augimdsCalibration = shuffle(augimdsTrain).subset(1:200);  
  
augimdsValidation = augmentedImageDatastore(inputSize,XValidation,YValidation);  
augimdsValidation = shuffle(augimdsValidation).subset(1:50);
```

Quantize the Network for GPU Deployment Using the Deep Network Quantizer App

This example uses a GPU execution environment. To learn about the products required to quantize and deploy the deep learning network to a GPU environment, see “Quantization Workflow Prerequisites” (Deep Learning Toolbox).

In the MATLAB® Command Window, open the Deep Network Quantizer app.

```
deepNetworkQuantizer
```

Select **New > Quantize a network**. The app automatically verifies your execution environment.

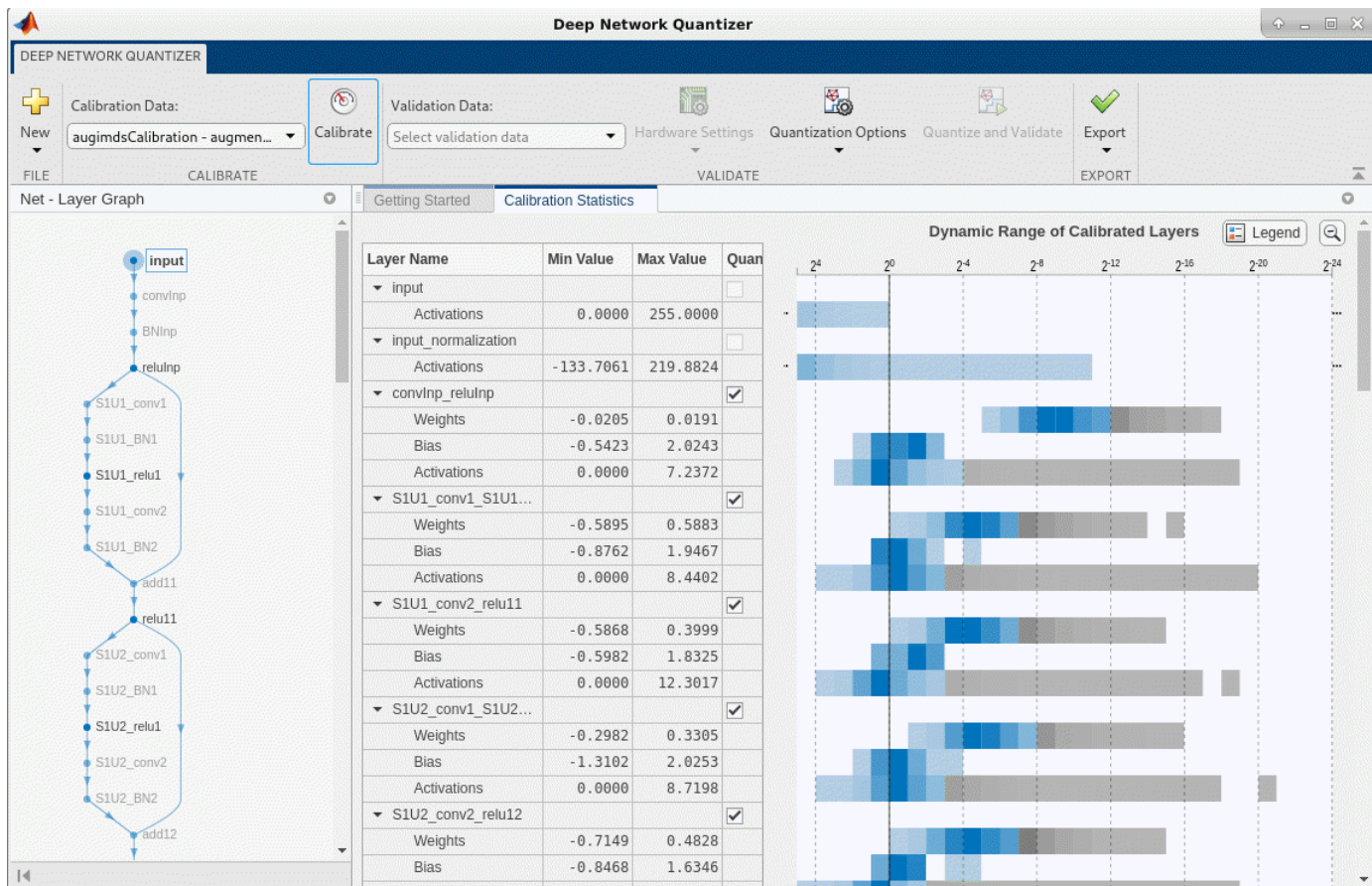
In the dialog, select the execution environment and the network to quantize from the base workspace. For this example, select a GPU execution environment and the DAG network `net`.

In the **Calibrate** section of the toolstrip, under **Calibration Data**, select the `augmentedImageDatastore` object from the base workspace containing the calibration data `augimdsCalibration`.

Click **Calibrate**.

Deep Network Quantizer uses the calibration data to exercise the network and collect range information for the learnable parameters in the network layers.

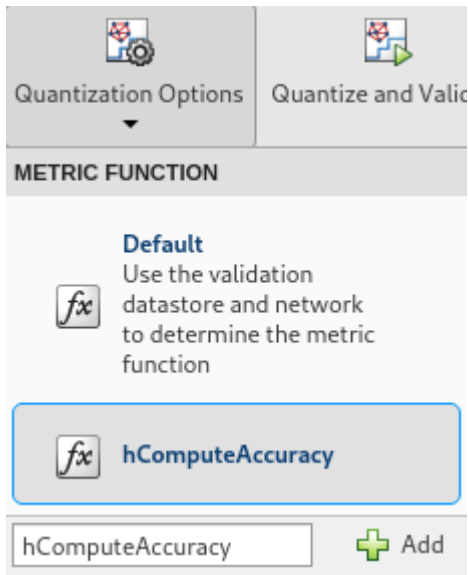
When the calibration is complete, the app displays a table containing the weights and biases in the convolution, as well as fully connected layers of the network and the dynamic ranges of the activations in all layers of the network with their minimum and maximum values during the calibration. To the right of the table, the app displays histograms of the dynamic ranges of the parameters. The gray regions of the histograms indicate data that cannot be represented by the quantized representation. For more information on how to interpret these histograms, see “Quantization of Deep Neural Networks” (Deep Learning Toolbox).



In the **Quantize Layer** column of the table, indicate whether to quantize the learnable parameters in the layer. Layers that are not convolution layers cannot be quantized, and therefore cannot be selected. Layers that are not quantized remain in single precision after quantization.

In the **Validate** section of the toolbar, under **Validation Data**, select the `augmidsValidation` object from the base workspace containing the validation data, `augmidsValidation`.

In the **Validate** section of the toolbar, under **Quantization Options**, select the metric function to use for validation. The app determines a default metric function to use for validation based on the type of network that you quantize. You can also add additional custom metric functions to use for validation. For this example, enter the name of the custom metric function `hComputeAccuracy`. Select **Add** to add `hComputeAccuracy` to the list of metric functions available in the app. Select `hComputeAccuracy` as the metric function to use for validation. This custom metric function compares the predicted label to the ground truth and returns the top-1 accuracy. The custom metric function must be on the path.



Click **Quantize and Validate**.

The app quantizes the network and displays a summary of the validation results. For this set of calibration and validation images, quantization of the network results in a 2% decrease in accuracy with a 73% reduction in learnable parameter memory for the set of 50 validation images.

Layer Name	Min Value	Max Value	Quan
input			
Activations	0.0000	255.0000	
input_normalization			
Activations	-133.7061	219.8824	
convInp_reluInp			<input checked="" type="checkbox"/>
Weights	-0.0205	0.0191	
Bias	-0.5423	2.0243	
Activations	0.0000	7.2372	
S1U1_conv1_S1U1...			<input checked="" type="checkbox"/>
Weights	-0.5895	0.5883	
Bias	-0.8762	1.9467	
Activations	0.0000	8.4402	

Metric	Floating-Point Network Results	Quantized Network Results	Percent Change
Learnable parameter memory (MB)	1.1113	0.3006	72.9544
Top-1 Accuracy	0.9800	0.9600	2.0408

After quantizing and validating the network, you can export the network or generate code. To export the network, select **Export > Export Quantizer** to create a `dlquantizer` object in the base workspace. To open the GPU Coder app and generate GPU code from the optimized neural network, select **Export > Generate Code**. To learn how to generate CUDA code for an optimized deep convolutional neural network using GPU Coder, see “Generate INT8 Code for Deep Learning Networks” on page 4-107.

Validate the Performance of the Network Using Multiple Metric Functions

You can use multiple metric functions to evaluate the performance of the network simultaneously by using the `dlquantizer` function.

To begin, load the pretrained network and data, and prepare the data for calibration and validation, as described above.

Create a `dlquantizer` object. Specify the network to quantize and the execution environment to use. Use the `calibrate` function to exercise the network with sample inputs from `augimdsCalibration` and collect range information.

```
dq = dlquantizer(net, 'ExecutionEnvironment', 'GPU');
calResults = calibrate(dq, augimdsCalibration)
```

Specify the metric functions in a `dlquantizationOptions` object. Use the `validate` function to quantize the learnable parameters in the convolution layers of the network and exercise the network. The `validate` function uses the metric functions defined in the `dlquantizationOptions` object to compare the results of the network before and after quantization. For this example, use the top-1 accuracy and top-5 accuracy metrics are used to evaluate the performance of the network.

```
dqOpts = dlquantizationOptions('MetricFcn', ...
    {@(x)hComputeAccuracy(x, net, augimdsValidation), ...
    @(x)hComputeTop_5(x, net, augimdsValidation)});

validationResults = validate(dq, augimdsValidation, dqOpts)

validationResults = struct with fields:
    NumSamples: 50
    MetricResults: [1x2 struct]
    Statistics: [2x2 table]
```

Examine the `MetricResults.Result` field of the validation output to see the performance of the optimized network as measured by each metric function used.

```
validationResults.MetricResults.Result
validationResults.Statistics
```

To visualize the calibration statistics, first save the `dlquantizer` object `dq`.

```
save('dlquantObj.mat', 'dq')
```

Then import the `dlquantizer` object `dq` in the Deep Network Quantizer app by selecting **New > Import dlquantizer object**.

Generate CUDA Code

Generate CUDA® code for a optimized deep convolutional neural network.

Create Entry-Point Function

Write an entry-point function in MATLAB® that:

- 1 Uses the `coder.loadDeepLearningNetwork` function to load a deep learning model and to construct and set up a CNN class. For more information, see “Load Pretrained Networks for Code Generation” on page 4-66.
- 2 Calls `predict` (Deep Learning Toolbox) to predict the responses.

```
type('mynet_predict.m');

function out = mynet_predict(netFile, im)
    persistent net;
    if isempty(net)
        net = coder.loadDeepLearningNetwork(netFile);
    end
    out = net.predict(im);
end
```

A persistent object `mynet` loads the `DAGNetwork` object. The first call to the entry-point function constructs and sets up the persistent object. Subsequent calls to the function reuse the same object to call `predict` on inputs, avoiding reconstructing and reloading the network object.

Code Generation by Using codegen

To configure build settings such as the output file name, location, and type, you create `coder` configuration objects. To create the objects, use the `coder.gpuConfig` function. For example, when generating CUDA MEX using the `codegen` command, use `cfg = coder.gpuConfig('mex')`.

To specify code generation parameters for cuDNN, set the `DeepLearningConfig` property to a `coder.CuDNNConfig` object that you create by using `coder.DeepLearningConfig`.

Specify the location of the MAT file containing the calibration data.

Specify the precision of the inference computations in supported layers by using the `DataType` property. For 8-bit integers, use `'int8'`. Int8 precision requires a CUDA GPU with compute capability of 6.1, 6.3, or higher. Use the `ComputeCapability` property of the GPU code configuration object to set the appropriate compute capability value.

```
cfg = coder.gpuConfig('mex');
cfg.TargetLang = 'C++';
cfg.DeepLearningConfig = coder.DeepLearningConfig('cudnn');
cfg.DeepLearningConfig.DataType = 'int8';
cfg.DeepLearningConfig.CalibrationResultFile = 'dlquantObj.mat';
netFile = 'mynet.mat';
save(netFile, 'net');
```

Run the `codegen` command. The `codegen` command generates CUDA code from the `mynet_predict.m` entry-point function.

```
codegen -config cfg mynet_predict -args {coder.Constant(netFile), ones(inputSize, 'single')} -rep
```

When code generation is successful, you can view the resulting code generation report by clicking View Report in the MATLAB Command Window. The report is displayed in the Report Viewer window. If the code generator detects errors or warnings during code generation, the report describes the issues and provides links to the problematic MATLAB code. See “Code Generation Reports”.

Code generation successful: [View report](#)

References

[1] Krizhevsky, Alex. 2009. "Learning Multiple Layers of Features from Tiny Images." <https://www.cs.toronto.edu/~kriz/learning-features-2009-TR.pdf>

See Also

Apps

Deep Network Quantizer

Functions

`dlquantizer` | `dlquantizationOptions` | `calibrate` | `validate` | `coder.loadDeepLearningNetwork` | `codegen`

Objects

`coder.CuDNNConfig` | `coder.TensorRTConfig`

More About

- "Quantization of Deep Neural Networks" on page 4-99
- "Generate INT8 Code for Deep Learning Networks" on page 4-107
- "Quantize Layers in Object Detectors and Generate CUDA Code" on page 4-237
- "Code Generation for Deep Learning Networks by Using cuDNN" on page 4-69
- "Code Generation for Deep Learning Networks by Using TensorRT" on page 4-78

Quantize Layers in Object Detectors and Generate CUDA Code

This example was previously named 'Quantize Object Detectors and Generate CUDA Code' but renamed in R2022a to avoid confusion with quantized network objects created by the `quantize` (Deep Learning Toolbox) function. Code generation does not support quantized deep neural networks produced by the `quantize` function.

This example shows how to generate CUDA® code for an SSD vehicle detector and a YOLO v2 vehicle detector that performs inference computations in 8-bit integers for the convolutional layers.

Deep learning is a powerful machine learning technique in which you train a network to learn image features and perform detection tasks. There are several techniques for object detection using deep learning, such as Faster R-CNN, You Only Look Once (YOLO v2), and SSD. For more information, see “Object Detection Using YOLO v2 Deep Learning” (Computer Vision Toolbox) and “Object Detection Using SSD Deep Learning” (Computer Vision Toolbox).

Neural network architectures used for deep learning applications contain many processing layers, including convolutional layers. Deep learning models typically work on large sets of labeled data. Performing inference on these models is computationally intensive, consuming significant amounts of memory. Neural networks use memory to store input data, parameters (weights), and activations from each layer as the input propagates through the network. Deep neural networks trained in MATLAB use single-precision floating point data types. Even networks that are small in size require a considerable amount of memory and hardware to perform these floating-point arithmetic operations. These restrictions can inhibit deployment of deep learning models to devices that have low computational power and smaller memory resources. By using a lower precision to store the weights and activations, you can reduce the memory requirements of the network.

You can use Deep Learning Toolbox in tandem with the Deep Learning Toolbox Model Quantization Library support package to reduce the memory footprint of a deep neural network by quantizing the weights, biases, and activations of convolution layers to 8-bit scaled integer data types. Then, you can use GPU Coder™ to generate CUDA code for the optimized network.

Download Pretrained Network

Download a pretrained object detector to avoid having to wait for training to complete.

```
detectorType = 
detectorType = 2
switch detectorType
    case 1
        if ~exist('ssdResNet50VehicleExample_20a.mat','file')
            disp('Downloading pretrained detector...');
            pretrainedURL = 'https://www.mathworks.com/supportfiles/vision/data/ssdResNet50VehicleExample_20a.mat';
            websave('ssdResNet50VehicleExample_20a.mat',pretrainedURL);
        end
    case 2
        if ~exist('yolov2ResNet50VehicleExample_19b.mat','file')
            disp('Downloading pretrained detector...');
            pretrainedURL = 'https://www.mathworks.com/supportfiles/vision/data/yolov2ResNet50VehicleExample_19b.mat';
            websave('yolov2ResNet50VehicleExample_19b.mat',pretrainedURL);
        end
end
```

Load Data

This example uses a small vehicle data set that contains 295 images. Many of these images come from the Caltech Cars 1999 and 2001 data sets, created by Pietro Perona and used with permission. Each image contains one or two labeled instances of a vehicle. A small data set is useful for exploring the training procedure, but in practice, more labeled images are needed to train a robust detector. Extract the vehicle images and load the vehicle ground truth data.

```
unzip vehicleDatasetImages.zip
data = load('vehicleDatasetGroundTruth.mat');
vehicleDataset = data.vehicleDataset;
```

Prepare Data for Training, Calibration, and Validation

The training data is stored in a table. The first column contains the path to the image files. The remaining columns contain the ROI labels for vehicles. Display the first few rows of the data.

```
vehicleDataset(1:4,:)
```

Split the data set into training, validation, and test sets. Select 60% of the data for training, 10% for calibration, and the remainder for validating the trained detector.

```
rng(0);
shuffledIndices = randperm(height(vehicleDataset));
idx = floor(0.6 * length(shuffledIndices) );

trainingIdx = 1:idx;
trainingDataTbl = vehicleDataset(shuffledIndices(trainingIdx),:);

calibrationIdx = idx+1 : idx + 1 + floor(0.1 * length(shuffledIndices) );
calibrationDataTbl = vehicleDataset(shuffledIndices(calibrationIdx),:);

validationIdx = calibrationIdx(end)+1 : length(shuffledIndices);
validationDataTbl = vehicleDataset(shuffledIndices(validationIdx),:);
```

Use `imageDatastore` and `boxLabelDatastore` to create datastores for loading the image and label data during training and evaluation.

```
imdsTrain = imageDatastore(trainingDataTbl{:, 'imageFilename'});
blldsTrain = boxLabelDatastore(trainingDataTbl(:, 'vehicle'));

imdsCalibration = imageDatastore(calibrationDataTbl{:, 'imageFilename'});
blldsCalibration = boxLabelDatastore(calibrationDataTbl(:, 'vehicle'));

imdsValidation = imageDatastore(validationDataTbl{:, 'imageFilename'});
blldsValidation = boxLabelDatastore(validationDataTbl(:, 'vehicle'));
```

Combine the image and box label datastores.

```
trainingData = combine(imdsTrain,blldsTrain);
calibrationData = combine(imdsCalibration,blldsCalibration);
validationData = combine(imdsValidation,blldsValidation);
```

Display one of the training images and box labels.

```
data = read(calibrationData);
I = data{1};
bbox = data{2};
```



```

annotatedImage = insertShape(I, 'Rectangle', bbox);
annotatedImage = imresize(annotatedImage, 2);
figure
imshow(annotatedImage)

```

Define Network Parameters

To reduce the computational cost of running the example, specify a network input size that corresponds to the minimum size required to run the network.

```

inputSize = [];
switch detectorType
    case 1
        inputSize = [300 300 3]; % Minimum size for SSD
    case 2
        inputSize = [224 224 3]; % Minimum size for YOLO v2
end

```

Define the number of object classes to detect.

```
numClasses = width(vehicleDataset)-1;
```

Data Augmentation

Data augmentation is used to improve network accuracy by randomly transforming the original data during training. By using data augmentation, you can add more variety to the training data without actually having to increase the number of labeled training samples.

Use transformations to augment the training data by:

- Randomly flipping the image and associated box labels horizontally.
- Randomly scaling the image and associated box labels.
- Jitter the image color.

Note that data augmentation is not applied to the test data. Ideally, test data is representative of the original data and left unmodified for unbiased evaluation.

```
augmentedCalibrationData = transform(calibrationData, @augmentVehicleData);
```

Visualize augmented training data by reading the same image multiple times.

```

augmentedData = cell(4,1);
for k = 1:4
    data = read(augmentedCalibrationData);
    augmentedData{k} = insertShape(data{1}, 'Rectangle', data{2});
    reset(augmentedCalibrationData);
end

figure
montage(augmentedData, 'BorderSize', 10)

```





Preprocess Calibration Data

Preprocess the augmented calibration data to prepare for calibration of the network.

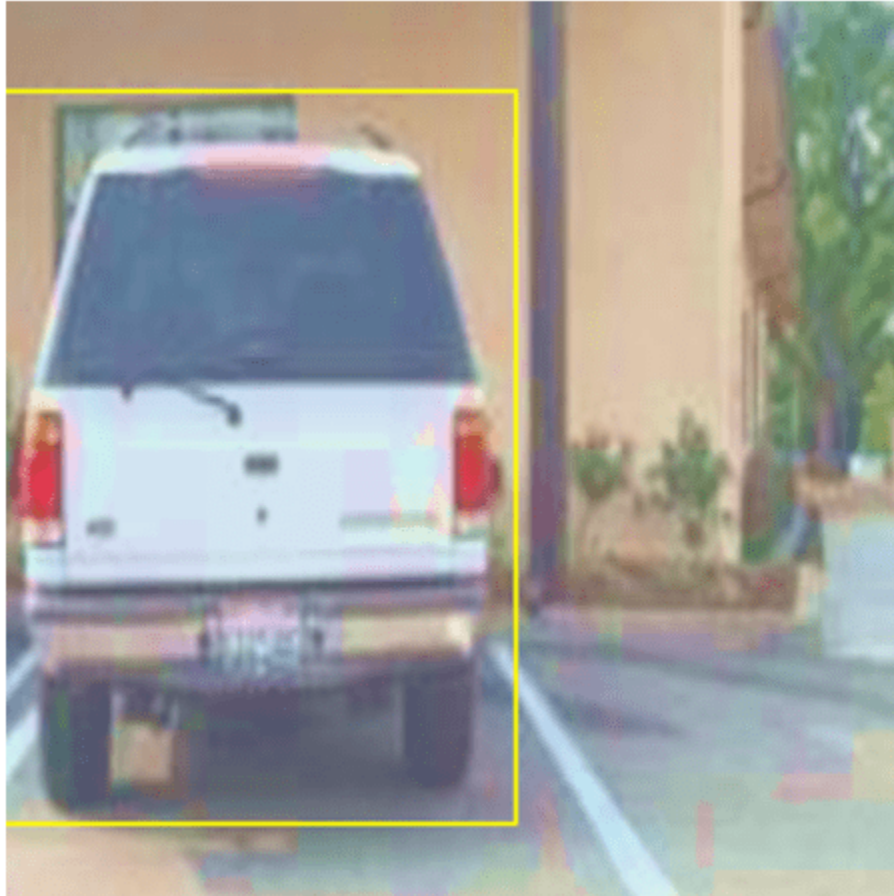
```
preprocessedCalibrationData = transform(augmentedCalibrationData,@(data)preprocessVehicleData(data));
```

Read the preprocessed calibration data.

```
data = read(preprocessedCalibrationData);
```

Display the image and bounding boxes.

```
I = data{1};  
bbox = data{2};  
annotatedImage = insertShape(I, 'Rectangle', bbox);  
annotatedImage = imresize(annotatedImage,2);  
figure  
imshow(annotatedImage)
```



Load and Test Pretrained Detector

Load the pretrained detector.

```
switch detectorType
    case 1
        % Load pretrained SSD detector for the example.
        pretrained = load('ssdResNet50VehicleExample_20a.mat');
        detector = pretrained.detector;
    case 2
        % Load pretrained YOLO v2 detector for the example.
        pretrained = load('yolov2ResNet50VehicleExample_19b.mat');
        detector = pretrained.detector;
end
```

As a quick test, run the detector on one test image.

```
data = read(calibrationData);
I = data{1,1};
```

```
I = imresize(I,inputSize(1:2));
[bboxes,scores] = detect(detector,I, 'Threshold', 0.4);
```

Display the results.

```
I = insertObjectAnnotation(I,'rectangle',bboxes,scores);
figure
imshow(I)
```



Validate Floating-Point Network

Evaluate the trained object detector on a large set of images to measure the performance. Computer Vision Toolbox™ provides functions to measure common object detector metrics, such as average precision (`evaluateDetectionPrecision`) and log-average miss rates (`evaluateDetectionMissRate`). For this example, use the average precision metric to evaluate performance. The average precision provides a single number that incorporates the ability of the detector to make correct classifications (`precision`) and the ability of the detector to find all relevant objects (`recall`).

Apply the same preprocessing transform to the test data as for the training data. Note that data augmentation is not applied to the test data. Ideally, test data is representative of the original data and left unmodified for unbiased evaluation.

```
preprocessedValidationData = transform(validationData,@(data)preprocessVehicleData(data,inputSize
```

Run the detector on all the test images.

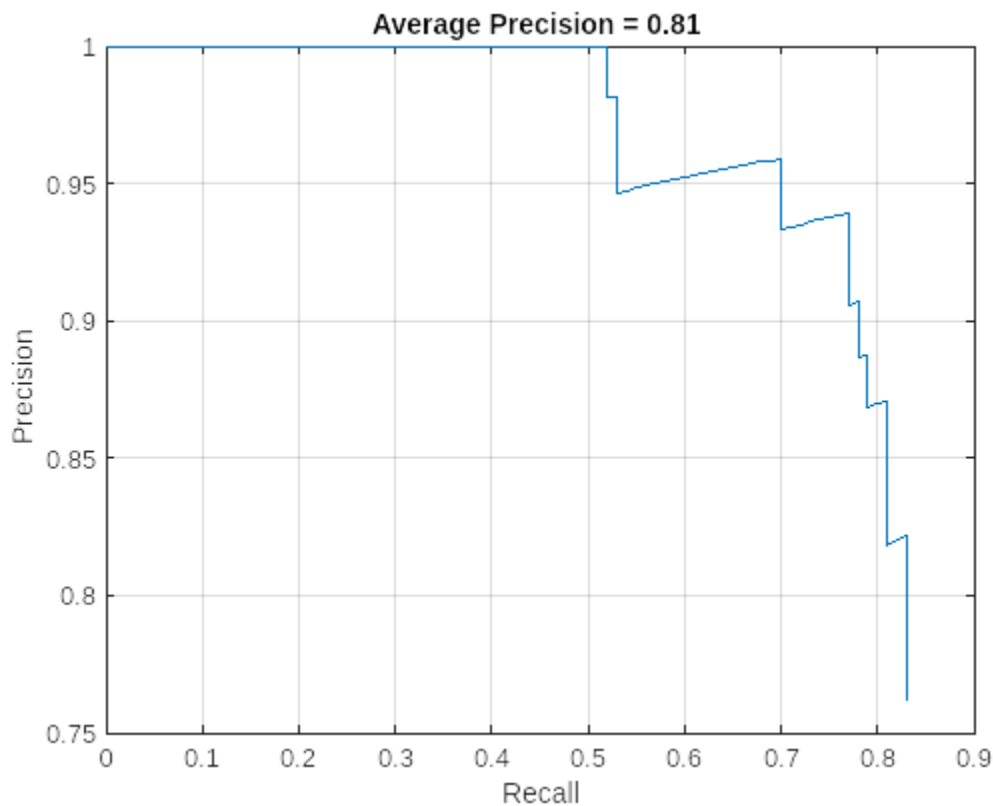
```
detectionResults = detect(detector, preprocessedValidationData,'Threshold',0.4);
```

Evaluate the object detector using average precision metric.

```
[ap,recall,precision] = evaluateDetectionPrecision(detectionResults,preprocessedValidationData);
```

The precision/recall (PR) curve highlights how precise a detector is at varying levels of recall. Ideally, the precision is 1 at all recall levels. Using more data can help improve the average precision, but might require more training time. Plot the PR curve.

```
figure
plot(recall,precision)
xlabel('Recall')
ylabel('Precision')
grid on
title(sprintf('Average Precision = %.2f',ap))
```



Generate Calibration Result File for the Network

Create a `dlquantizer` object and specify the detector to quantize. By default, the execution environment is set to GPU. To learn about the products required to quantize and deploy the detector to a GPU environment, see “Quantization Workflow Prerequisites” (Deep Learning Toolbox).

```
quantObj = dlquantizer(detector)

quantObj =
  dlquantizer with properties:
      NetworkObject: [1x1 yolov2objectDetector]
  ExecutionEnvironment: 'GPU'
```

Specify the metric function in a `dlquantizationOptions` object.

```

quantOpts = dlquantizationOptions;
quantOpts = dlquantizationOptions('MetricFcn', ...
    @(x)hVerifyDetectionResults(x, detector.Network, preprocessedValidationData));

```

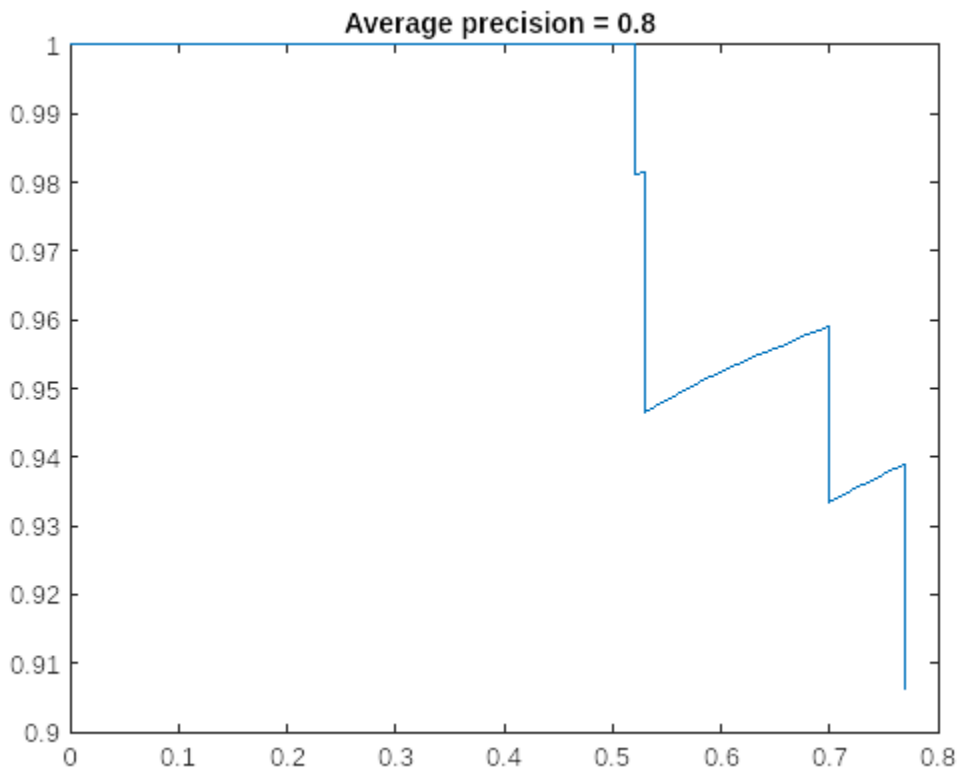
Use the `calibrate` function to exercise the network with sample inputs and collect range information. The `calibrate` function exercises the network and collects the dynamic ranges of the weights and biases in the convolution and fully connected layers of the network, as well as the dynamic ranges of the activations in all layers of the network. The function returns a table. Each row of the table contains range information for a learnable parameter of the optimized network.

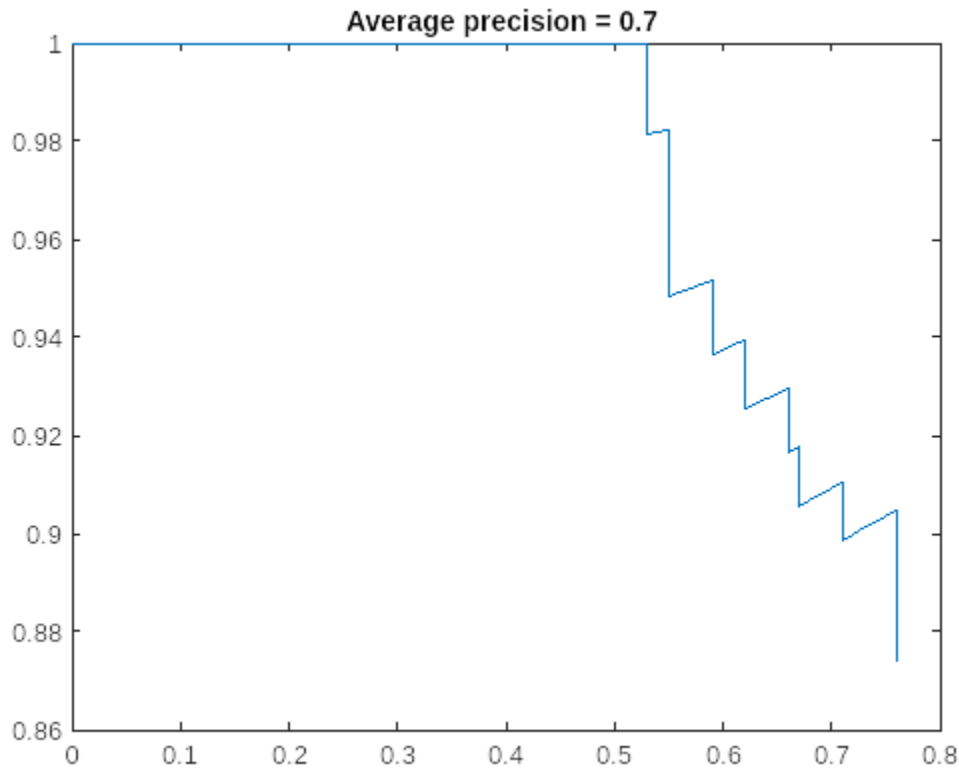
```
calResults = calibrate(quantObj,preprocessedCalibrationData)
```

Use the `validate` function to quantize the learnable parameters in the convolution layers of the network and exercise the network. The function uses the metric function defined in the `dlquantizationOptions` object to compare the results of the network before and after quantization.

Examine the `MetricResults.Result` field of the validation output to see the performance of the optimized network. The first row in the results table contains the information for the original, floating-point implementation. The second row contains the information for the quantized implementation. The output of the metric function is displayed in the `MetricOutput` column.

```
valResults = validate(quantObj,preprocessedValidationData,quantOpts)
```





```
valResults = struct with fields:
    NumSamples: 88
    MetricResults: [1x1 struct]
    Statistics: [2x2 table]
```

```
valResults.MetricResults.Result
```

The metrics show that quantization reduces the required memory by approximately 75% and the network accuracy by approximately 3%.

To visualize the calibration statistics, use the Deep Network Quantizer app. First, save the `dlquantizer` object.

```
save('dlquantObj.mat', 'quantObj')
```

In the MATLAB® Command Window, open the Deep Network Quantizer app.

```
deepNetworkQuantizer
```

Then import the `dlquantizer` object `dq` in the Deep Network Quantizer app by selecting **New > Import dlquantizer object**.

Generate CUDA Code

After you train and evaluate the detector, you can generate code for the `ssdObjectDetector` or `yolov20ObjectDetector` using GPU Coder™. For more details, see “Code Generation for Object

Detection by Using Single Shot Multibox Detector” (Computer Vision Toolbox) and “Code Generation for Object Detection by Using YOLO v2” on page 4-180.

```
cfg = coder.gpuConfig('mex');
cfg.TargetLang = 'C++';

% Check compute capability of GPU
gpuInfo = gpuDevice;
cc = gpuInfo.ComputeCapability;

% Create deep learning code generation configuration object
cfg.DeepLearningConfig = coder.DeepLearningConfig('cudnn');

% INT8 precision requires a CUDA GPU with minimum compute capability of
% 6.1, 6.3, or higher
cfg.GpuConfig.ComputeCapability = cc;
cfg.DeepLearningConfig.DataType = 'int8';
cfg.DeepLearningConfig.CalibrationResultFile = 'dlquantObj.mat';
```

Run the codegen command to generate CUDA code.

```
codegen -config cfg mynet_detect -args {coder.Constant(detectorType), ones(inputSize, 'single')}
```

When code generation is successful, you can view the resulting code generation report by clicking View Report in the MATLAB Command Window. The report is displayed in the Report Viewer window. If the code generator detects errors or warnings during code generation, the report describes the issues and provides links to the problematic MATLAB code. See “Code Generation Reports”.

References

- [1] Liu, Wei, Dragomir Anguelov, Dumitru Erhan, Christian Szegedy, Scott Reed, Cheng Yang Fu, and Alexander C. Berg. "SSD: Single Shot Multibox Detector." In Computer Vision - ECCV 2016, edited by Bastian Leibe, Jiri Matas, Nicu Sebe, and Max Welling, 9905:21-37. Cham: Springer International Publishing, 2016. https://doi.org/10.1007/978-3-319-46448-0_2
- [2] Redmon, Joseph, and Ali Farhadi. "YOLO9000: Better, Faster, Stronger." In 2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR), 6517-25. Honolulu, HI: IEEE, 2017. <https://doi.org/10.1109/CVPR.2017.690>

See Also

Apps

Deep Network Quantizer

Functions

dlquantizer | dlquantizationOptions | calibrate | validate |
coder.loadDeepLearningNetwork | codegen

Objects

coder.CuDNNConfig | coder.TensorRTConfig

More About

- “Quantization of Deep Neural Networks” on page 4-99

- “Generate INT8 Code for Deep Learning Networks” on page 4-107
- “Quantize Residual Network Trained for Image Classification and Generate CUDA Code” on page 4-229
- “Code Generation for Deep Learning Networks by Using cuDNN” on page 4-69
- “Code Generation for Deep Learning Networks by Using TensorRT” on page 4-78

Parameter Pruning and Quantization of Image Classification Network

This example shows how to prune the parameters of a trained neural network using two parameter score metrics: The Magnitude score [1] and Synaptic Flow score [2].

In many applications where transfer learning is used to retrain an image classification network for a new task or where a new network is trained from scratch, the optimal network architecture is not known, and the network might be overparameterized. An overparameterized network has redundant connections. Structured pruning, also known as sparsification, is a compression technique that aims to identify redundant, unnecessary connections you can remove without affecting the network accuracy. When you use pruning in combination with network quantization, you can reduce the inference time and memory footprint of the network making it easier to deploy.

This example shows how to:

- Perform post-training, iterative, unstructured pruning without the need for training data
- Evaluate the performance of two different pruning algorithms
- Investigate the layer-wise sparsity induced after pruning
- Evaluate the impact of pruning on classification accuracy
- Evaluate the impact of quantization on the classification accuracy of the pruned network

This example uses a simple convolutional neural network to classify handwritten digits from 0 to 9. For more information on setting up the data used for training and validation, see “Create Simple Deep Learning Network for Classification” (Deep Learning Toolbox).

Load Pretrained Network and Data

Load the training and validation data. Train a convolutional neural network for the classification task.

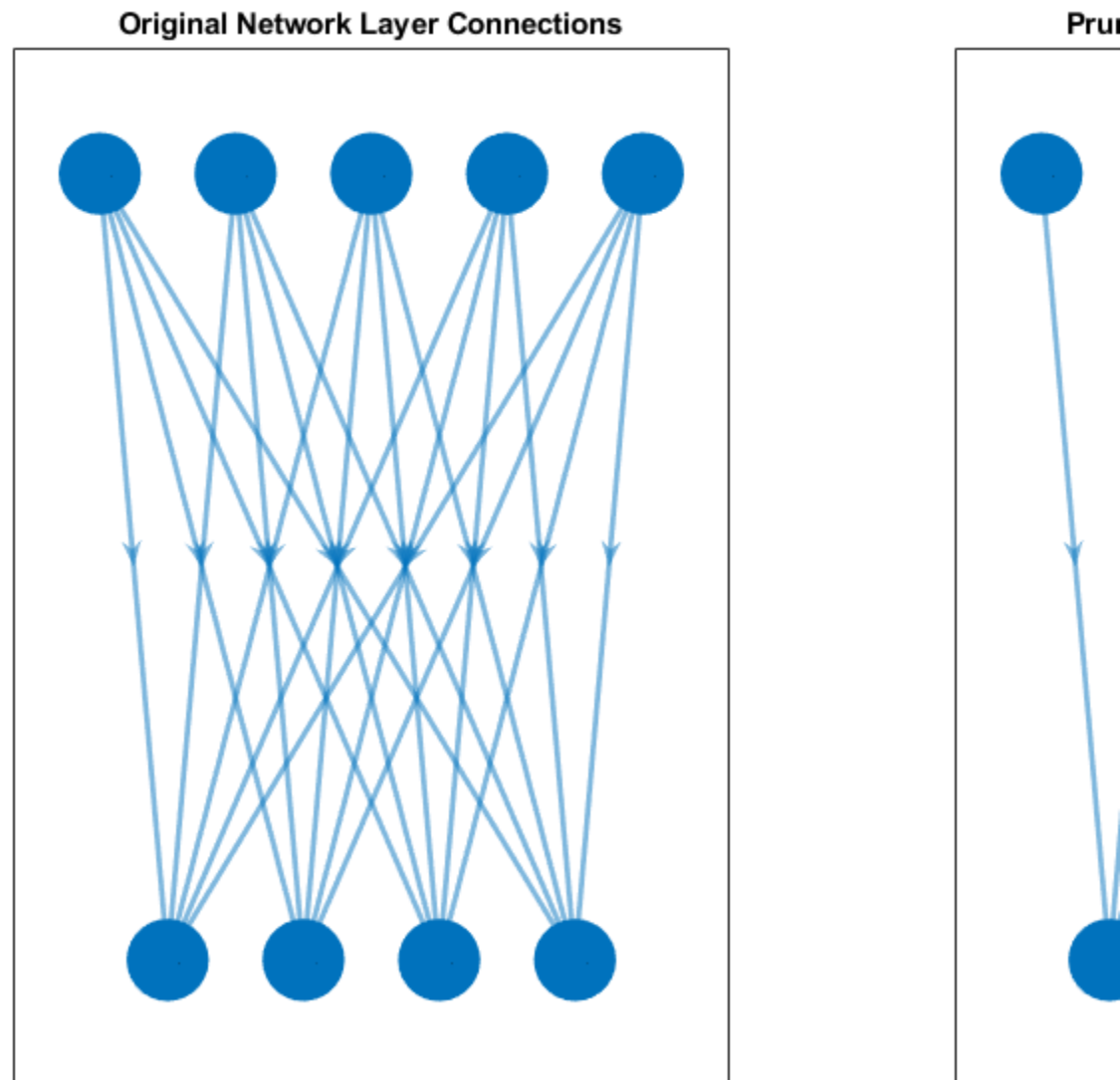
```
[imdsTrain, imdsValidation] = loadDigitDataset;
net = trainDigitDataNetwork(imdsTrain, imdsValidation);
trueLabels = imdsValidation.Labels;
classes = categories(trueLabels);
```

Create a minibatchqueue object containing the validation data. Set `executionEnvironment` to `auto` to evaluate the network on a GPU, if one is available. By default, the `minibatchqueue` object converts each output to a `gpuArray` if a GPU is available. Using a GPU requires Parallel Computing Toolbox™ and a supported GPU device. For information on supported devices, see “GPU Computing Requirements”.

```
executionEnvironment = ;
miniBatchSize = 128;
imdsValidation.ReadSize = miniBatchSize;
mbqValidation = minibatchqueue(imdsValidation,1,...
    'MiniBatchSize',miniBatchSize,...
    'MiniBatchFormat','SSCB',...
    'MiniBatchFcn',@preprocessMiniBatch,...
    'OutputEnvironment',executionEnvironment);
```

Neural Network Pruning

The goal of neural network pruning is to identify and remove unimportant connections to reduce the size of the network without affecting network accuracy. In the following figure, on the left, the network has connections that map each neuron to the neuron of the next layer. After pruning, the network has fewer connections than the original network.

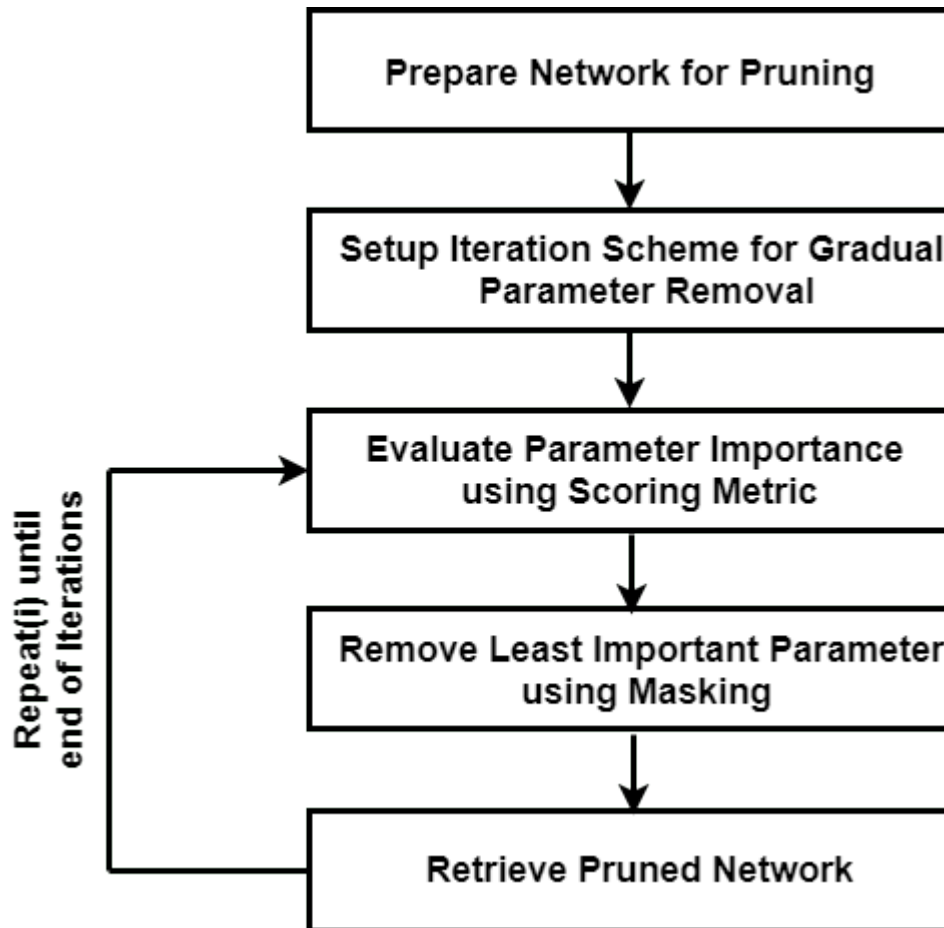


A pruning algorithm assigns a score to each parameter in the network. The score ranks the importance of each connection in the network. You can use one of two pruning approaches to achieve a target sparsity:

- One-shot pruning - Remove a specified percentage of connections based on their score in one step. This method is prone to layer collapse when you specify a high sparsity value.
- Iterative pruning - Achieve the target sparsity in a series of iterative steps. You can use this method when evaluated scores are sensitive to network structure. Scores are reevaluated at every iteration, so using a series of steps allows the network to move toward sparsity incrementally.

This example uses the iterative pruning method to achieve a target sparsity.

Iterative Pruning



Convert to dlnetwork Object

In this example, you use the Synaptic Flow algorithm, which requires that you create a custom cost function and evaluate the gradients with respect to the cost function to calculate the parameter score. To create a custom cost function, first convert the pretrained network to a `dlnetwork` (Deep Learning Toolbox).

Convert the network to a layer graph and remove the layers used for classification using `removeLayers`.

```

lgraph = layerGraph(net.Layers);
lgraph = removeLayers(lgraph, ["softmax", "classoutput"]);
dlnet = dlnetwork(lgraph);
  
```

Use `analyzeNetwork` to analyze the network architecture and learnable parameters.

```
analyzeNetwork(dlnet)
```

Evaluate the accuracy of the network before pruning.

```
accuracyOriginalNet = evaluateAccuracy(dlnet, mbqValidation, classes, trueLabels)
```

```
accuracyOriginalNet = 0.9908
```

The layers with learnable parameters are the 3 convolutional layers and one fully connected layer. The network initially consists of total 21578 learnable parameters.

```
numTotalParams = sum(cellfun(@numel, dlnet.Learnables.Value))
```

```
numTotalParams = 21578
```

```
numNonZeroPerParam = cellfun(@(w) nnz(extractdata(w)), dlnet.Learnables.Value)
```

```
numNonZeroPerParam = 8×1
```

```
    72
     8
   1152
    16
   4608
    32
  15680
    10
```

Sparsity is defined as the percentage of parameters in the network with a value of zero. Check the sparsity of the network.

```
initialSparsity = 1 - (sum(numNonZeroPerParam) / numTotalParams)
```

```
initialSparsity = 0
```

Before pruning, the network has a sparsity of zero.

Create Iteration Scheme

To define an iterative pruning scheme, specify the target sparsity and number of iterations. For this example, use linearly spaced iterations to achieve the target sparsity.

```
numIterations = 10;
targetSparsity = 0.90;
iterationScheme = linspace(0, targetSparsity, numIterations);
```

Pruning Loop

For each iteration, the custom pruning loop in this example performs the following steps:

- Calculate the score for each connection.
- Rank the scores for all connections in the network based on the selected pruning algorithm.
- Determine the threshold for removing connections with the lowest scores.
- Create the pruning mask using the threshold.

- Apply the pruning mask to learnable parameters of the network.

Network Mask

Instead of setting entries in the weight arrays directly to zero, the pruning algorithm creates a binary mask for each learnable parameter that specifies whether a connection is pruned. The mask allows you to explore the behavior of the pruned network and try different pruning schemes without changing the underlying network structure.

For example, consider the following weights.

```
testWeight = [10.4 5.6 0.8 9];
```

Create a binary mask for each parameter in testWeight.

```
testMask = [1 0 1 0];
```

Apply the mask to testWeight to get the pruned weights.

```
testWeightsPruned = testWeight.*testMask
```

```
testWeightsPruned = 1×4
```

```
    10.4000         0     0.8000         0
```

In iterative pruning, you create a binary mask for each iteration that contains pruning information. Applying the mask to the weights array does not change either the size of the array or the structure of the neural network. Therefore, the pruning step does not directly result in any speedup during inference or compression of the network size on disk.

Initialize a plot that compares the accuracy of the pruned network to the original network.

```
figure
plot(100*iterationScheme([1,end]),100*accuracyOriginalNet*[1 1], '*-b', 'LineWidth',2, "Color", "b")
ylim([0 100])
xlim(100*iterationScheme([1,end]))
xlabel("Sparsity (%)")
ylabel("Accuracy (%)")
legend("Original Accuracy", "Location", "southwest")
title("Pruning Accuracy")
grid on
```

Magnitude Pruning

Magnitude pruning [1] assigns a score to each parameter equal to its absolute value. It is assumed that the absolute value of a parameter corresponds to its relative importance to the accuracy of the trained network.

Initialize the mask. For the first iteration, you do not prune any parameters and the sparsity is 0%.

```
pruningMaskMagnitude = cell(1,numIterations);
pruningMaskMagnitude{1} = dLupdate(@(p)true(size(p)), dLnet.Learnables);
```

Below is an implementation of magnitude pruning. The network is pruned to various target sparsities in a loop to provide the flexibility to choose a pruned network based on its accuracy.

```
lineAccuracyPruningMagnitude = animatedline('Color','g','Marker','o','LineWidth',1.5);
legend("Original Accuracy", "Magnitude Pruning Accuracy", "Location", "southwest")
```

```

% Compute magnitude scores
scoresMagnitude = calculateMagnitudeScore(dlnet);

for idx = 1:numel(iterationScheme)

    prunedNetMagnitude = dlnet;

    % Update the pruning mask
    pruningMaskMagnitude{idx} = calculateMask(scoresMagnitude,iterationScheme(idx));

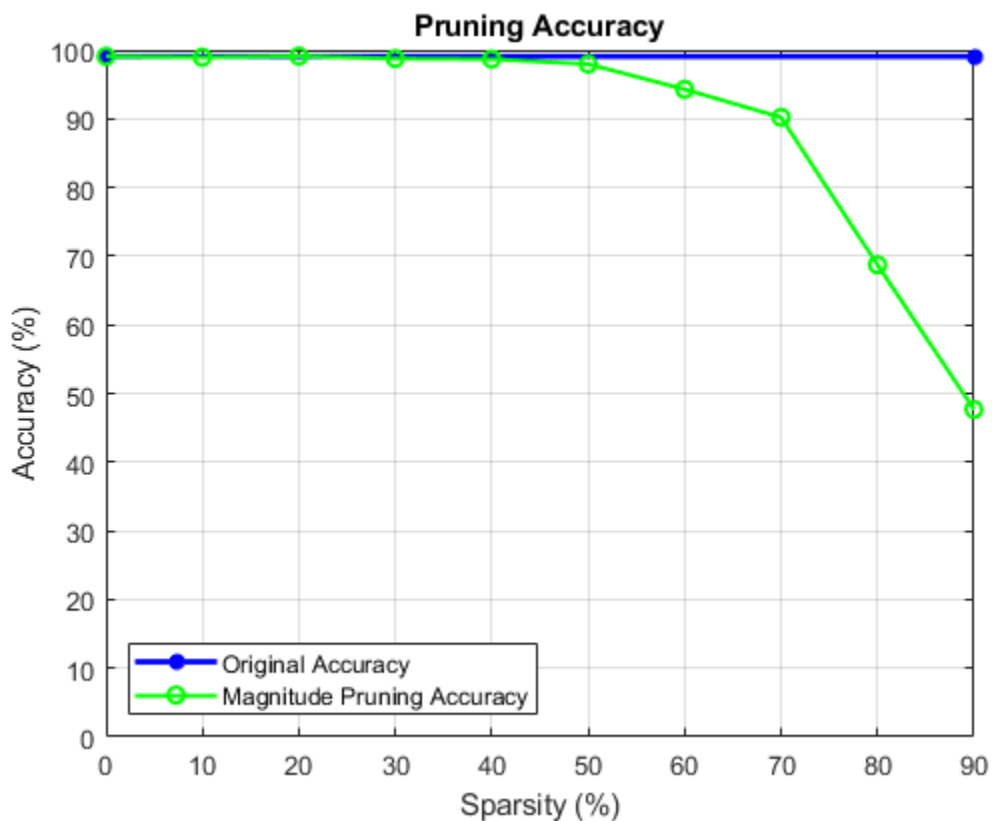
    % Check the number of zero entries in the pruning mask
    numPrunedParams = sum(cellfun(@(m)nnz(~extractdata(m)),pruningMaskMagnitude{idx}.Value));
    sparsity = numPrunedParams/numTotalParams;

    % Apply pruning mask to network parameters
    prunedNetMagnitude.Learnables = dupdate(@(W,M)W.*M, prunedNetMagnitude.Learnables, pruningMaskMagnitude{idx});

    % Compute validation accuracy on pruned network
    accuracyMagnitude = evaluateAccuracy(prunedNetMagnitude,mbqValidation,classes,trueLabels);

    % Display the pruning progress
    addpoints(lineAccuracyPruningMagnitude,100*sparsity,100*accuracyMagnitude)
    drawnow
end

```



SynFlow Pruning

Synaptic flow conservation (SynFlow) [2] scores are used for pruning. You can use this method to prune networks that use linear activation functions such as ReLU.

Initialize the mask. For the first iteration, no parameters are pruned, and the sparsity is 0%.

```
pruningMaskSynFlow = cell(1,numIterations);
pruningMaskSynFlow{1} = dlupdate(@(p)true(size(p)),dlnet.Learnables);
```

The input data you use to compute the scores is a single image containing ones. If you are using a GPU, convert the data to a `gpuArray`.

```
dlX = dlarray(ones(net.Layers(1).InputSize),'SSC');
if (executionEnvironment == "auto" && canUseGPU) || executionEnvironment == "gpu"
    dlX = gpuArray(dlX);
end
```

The below loop implements iterative synaptic flow score for pruning [2] where a custom cost function evaluates the SynFlow score for each parameter used for network pruning.

```
lineAccuracyPruningSynflow = animatedline('Color','r','Marker','o','LineWidth',1.5);
legend("Original Accuracy","Magnitude Pruning Accuracy","Synaptic Flow Accuracy","Location","sou

prunedNetSynFlow = dlnet;

% Iteratively increase sparsity
for idx = 1:numel(iterationScheme)
    % Compute SynFlow scores
    scoresSynFlow = calculateSynFlowScore(prunedNetSynFlow,dlX);

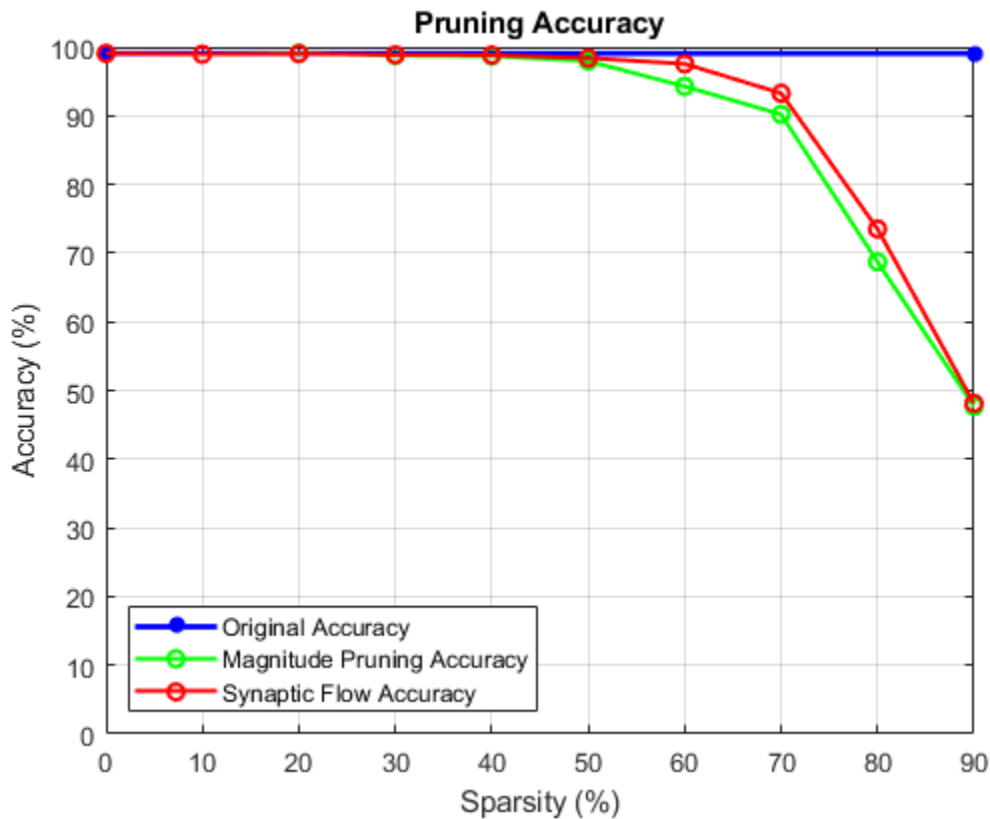
    % Update the pruning mask
    pruningMaskSynFlow{idx} = calculateMask(scoresSynFlow,iterationScheme(idx));

    % Check the number of zero entries in the pruning mask
    numPrunedParams = sum(cellfun(@(m)nnz(~extractdata(m)),pruningMaskSynFlow{idx}.Value));
    sparsity = numPrunedParams/numTotalParams;

    % Apply pruning mask to network parameters
    prunedNetSynFlow.Learnables = dlupdate(@(W,M)W.*M, prunedNetSynFlow.Learnables, pruningMaskS

    % Compute validation accuracy on pruned network
    accuracySynFlow = evaluateAccuracy(prunedNetSynFlow,mbqValidation,classes,trueLabels);

    % Display the pruning progress
    addpoints(lineAccuracyPruningSynflow,100*sparsity,100*accuracySynFlow)
    drawnow
end
```



Investigate Structure of Pruned Network

Choosing how much to prune a network is a trade-off between accuracy and sparsity. Use the sparsity versus accuracy plot to select the iteration with the desired sparsity level and acceptable accuracy.

```

pruningMethod = "SynFlow" ;
selectedIteration = 8 ;
prunedDLNet = createPrunedNet(dlNet,selectedIteration,pruningMaskSynFlow,pruningMaskMagnitude,pruningMaskMagnitude);
[sparsityPerLayer,prunedChannelsPerLayer,numOutChannelsPerLayer,layerNames] = pruningStatistics(prunedDLNet);

```

Earlier convolutional layers are typically pruned less since they contain more relevant information about the core low-level structure of the image (e.g. edges and corners) which are essential for interpreting the image.

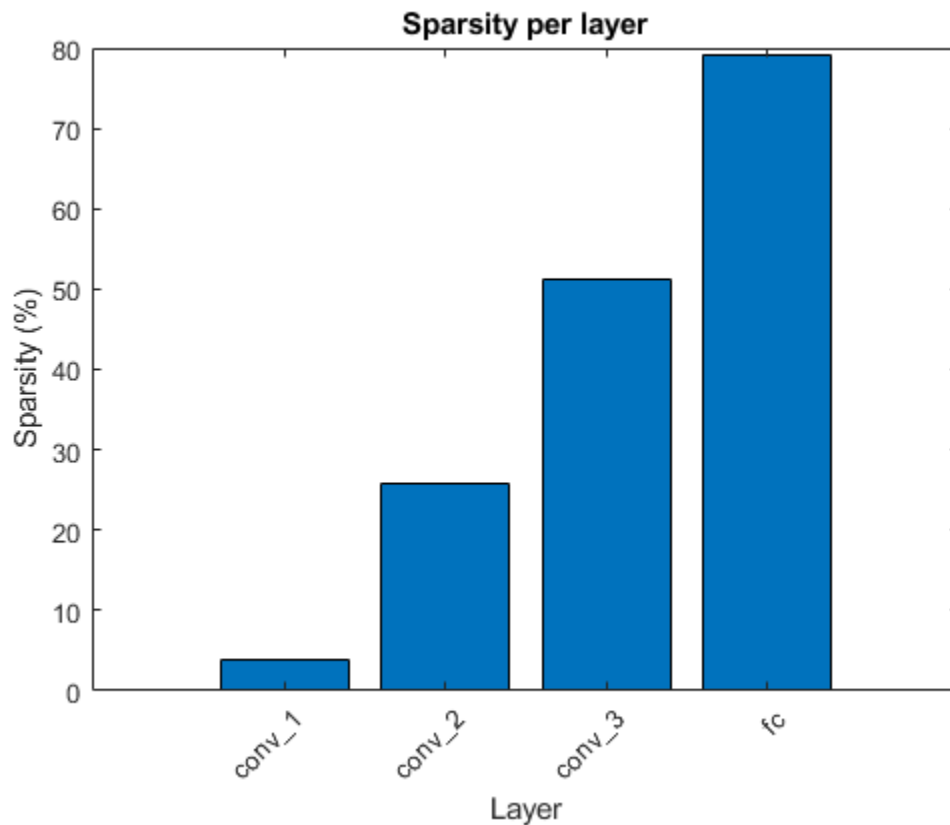
Plot the sparsity per layer for the selected pruning method and iteration.

```

figure
bar(sparsityPerLayer*100)
title("Sparsity per layer")
xlabel("Layer")
ylabel("Sparsity (%)")
xticks(1:numel(sparsityPerLayer))
xticklabels(layerNames)

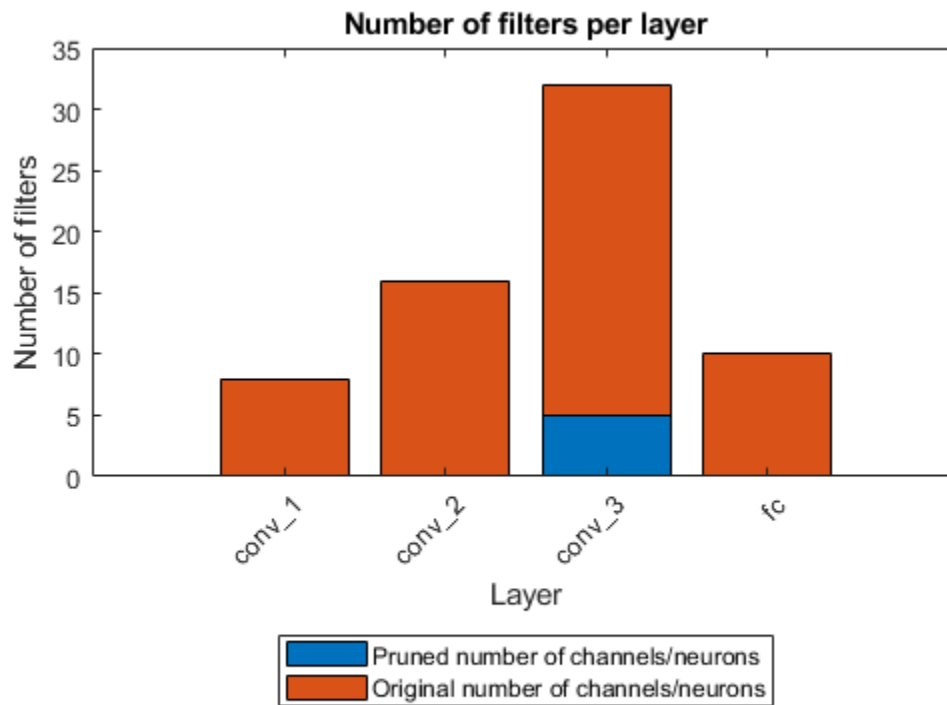
```

```
xtickangle(45)
set(gca,'TickLabelInterpreter','none')
```



The pruning algorithm prunes single connections when you specify a low target sparsity. When you specify a high target sparsity, the pruning algorithm can prune whole filters and neurons in convolutional or fully connected layers.

```
figure
bar([prunedChannelsPerLayer,numOutChannelsPerLayer-prunedChannelsPerLayer],"stacked")
xlabel("Layer")
ylabel("Number of filters")
title("Number of filters per layer")
xticks(1:(numel(layerNames)))
xticklabels(layerNames)
xtickangle(45)
legend("Pruned number of channels/neurons" , "Original number of channels/neurons","Location","s")
set(gca,'TickLabelInterpreter','none')
```



Evaluate Network Accuracy

Compare the accuracy of the network before and after pruning.

```
YPredOriginal = modelPredictions(dlnet,mbqValidation,classes);
accOriginal = mean(YPredOriginal == trueLabels)
```

```
accOriginal = 0.9908
```

```
YPredPruned = modelPredictions(prunedDLNet,mbqValidation,classes);
accPruned = mean(YPredPruned == trueLabels)
```

```
accPruned = 0.9328
```

Create a confusion matrix chart to explore the true class labels to the predicted class labels for the original and pruned network.

```
figure
confusionchart(trueLabels,YPredOriginal);
title("Original Network")
```

Original Network

0	249									1
1	3	245					2			
2			249	1						
3				249					1	
4					249		1			
5		1		4		244			1	
6	1		1				248			
7		1						249		
8			1				1		248	
9	2								1	247
	0	1	2	3	4	5	6	7	8	9

Predicted Class

The validation set of the digits data contains 250 images for each class, so if a network predicts the class of each image perfectly, all scores on the diagonal equal 250 and no values are outside of the diagonal.

```
confusionchart(trueLabels,YPredPruned);
title("Pruned Network")
```

Pruned Network

0	249									1
1	1	248					1			
2	1	3	244	1	1					
3		4		237	2	1				6
4	2	1			246					1
5		3	1	2	2	235				7
6	11	4	4		1	1	224	1		4
7		6						244		
8	26	1	6	5	7	5	16		160	24
9	1		1		3					245
	0	1	2	3	4	5	6	7	8	9

Predicted Class

When pruning a network, compare the confusion chart of the original network and the pruned network to check how the accuracy for each class label changes for the selected sparsity level. If all numbers on the diagonal decrease roughly equally, no bias is present. However, if the decreases are not equal, you might need to choose a pruned network from an earlier iteration by reducing the value of the variable `selectedIteration`.

Quantize Pruned Network

Deep neural networks trained in MATLAB use single-precision floating point data types. Even networks that are small require a considerable amount of memory and hardware to perform floating-point arithmetic operations. These restrictions can inhibit deployment of deep learning models that have low computational power and less memory resources. By using a lower precision to store the weights and activations, you can reduce the memory requirements of the network. You can use Deep Learning Toolbox in tandem with the Deep Learning Model Quantization Library support package to reduce the memory footprint of a deep neural network by quantizing the weights, biases, and activations of the convolution layers to 8-bit scaled integer data types.

Pruning a network impacts the range statistics of parameters and activations at each layer, so the accuracy of the quantized network can change. To explore this difference, quantize the pruned network and use the quantized network to perform inference.

Split the data into calibration and validation data sets.

```
calibrationDataStore = splitEachLabel(imdsTrain,0.1,'randomize');
validationDataStore = imdsValidation;
```

Create a `dlquantizer` object and specify the pruned network as the network to quantize.

```
prunedNet = assembleNetwork([prunedDLNet.Layers ; net.Layers(end-1:end)]);
```

```
quantObjPrunedNetwork = dlquantizer(prunedNet, 'ExecutionEnvironment', 'GPU');
```

Use the `calibrate` function to exercise the network with the calibration data and collect range statistics for the weights, biases, and activations at each layer.

```
calResults = calibrate(quantObjPrunedNetwork, calibrationDataStore)
```

Use the `validate` function to compare the results of the network before and after quantization using the validation data set.

```
valResults = validate(quantObjPrunedNetwork, validationDataStore);
```

Examine the `MetricResults.Result` field of the validation output to see the accuracy of the quantized network.

```
valResults.MetricResults.Result
valResults.Statistics
```

Mini Batch Preprocessing Function

The `preprocessMiniBatch` function preprocesses a mini-batch of predictors by extracting the image data from the input cell array and concatenate into a numeric array. For grayscale input, concatenating the data over the fourth dimension adds a third dimension to each image to use as a singleton channel dimension.

```
function X = preprocessMiniBatch(XCell)
% Extract image data from cell and concatenate.
X = cat(4,XCell{:});
end
```

Model Accuracy Function

Evaluate the classification accuracy of the `dlnetwork`. Accuracy is the percentage of labels correctly classified by the network.

```
function accuracy = evaluateAccuracy(dlNet,mbqValidation,classes,trueLabels)
YPred = modelPredictions(dlNet,mbqValidation,classes);
accuracy = mean(YPred == trueLabels);
end
```

SynFlow Score Function

The `calculateSynFlowScore` function calculates Synaptic Flow (SynFlow) scores. Synaptic saliency [2] is described as the class of gradient-based scores defined by the product of gradient of loss multiplied by the parameter value:

$$\text{synFlowScore} = \frac{d(\text{loss})}{d\theta} * \theta$$

The SynFlow score is a synaptic saliency score that uses the sum of all network outputs as a loss function:

$$\text{loss} = \sum f(\text{abs}(\theta), X)$$

f is the function represented by the neural network

θ are the parameters of the network

X is the input array to the network

To compute parameter gradients with respect to this loss function, use `dlfeval` and a model gradients function.

```
function score = calculateSynFlowScore(dlnet,dlX)
dlnet.Learnables = dlupdate(@abs, dlnet.Learnables);
gradients = dlfeval(@modelGradients,dlnet,dlX);
score = dlupdate(@(g,w)g.*w, gradients, dlnet.Learnables);
end
```

Model Gradients for SynFlow Score

```
function gradients = modelGradients(dlNet,inputArray)
% Evaluate the gradients on a given input to the dlnetwork
dlYPred = predict(dlNet,inputArray);
pseudoloss = sum(dlYPred,'all');
gradients = dlgradient(pseudoloss,dlNet.Learnables);
end
```

Magnitude Score Function

The `calculateMagnitudeScore` function returns the magnitude score, defined as the element-wise absolute value of the parameters.

```
function score = calculateMagnitudeScore(dlnet)
score = dlupdate(@abs, dlnet.Learnables);
end
```

Mask Generation Function

The `calculateMask` function returns a binary mask for the network parameters based on the given scores and the target sparsity.

```
function mask = calculateMask(scoresMagnitude,sparsity)
% Compute a binary mask based on the parameter-wise scores such that the mask contains a percenta

% Flatten the cell array of scores into one long score vector
flattenedScores = cell2mat(cellfun(@(S)extractdata(gather(S(:))),scoresMagnitude.Value,'UniformO
% Rank the scores and determine the threshold for removing connections for the
% given sparsity
flattenedScores = sort(flattenedScores);
k = round(sparsity*numel(flattenedScores));
if k==0
    thresh = 0;
else
    thresh = flattenedScores(k);
end
% Create a binary mask
mask = dlupdate( @(S)S>thresh, scoresMagnitude);
end
```

Model Predictions Function

The `modelPredictions` function takes as input a `dlnetwork` object `dlnet`, a `minibatchqueue` of input data `mbq`, and the network classes, and computes the model predictions by iterating over all

data in the minibatchqueue object. The function uses the onehotdecode function to find the predicted class with the highest score.

```
function predictions = modelPredictions(dlnet,mbq,classes)
predictions = [];
while hasdata(mbq)
    dlXTest = next(mbq);
    dlYPred = softmax(predict(dlnet,dlXTest));
    YPred = onehotdecode(dlYPred,classes,1)';
    predictions = [predictions; YPred];
end
reset(mbq)
end
```

Apply Pruning Function

The createPrunedNet function returns the pruned dlnetwork for the specified pruning algorithm and iteration.

```
function prunedNet = createPrunedNet(dlnet,selectedIteration,pruningMaskSynFlow,pruningMaskMagni
switch pruningMethod
    case "Magnitude"
        prunedNet = dlupdate(@(W,M)W.*M, dlnet, pruningMaskMagnitude{selectedIteration});
    case "SynFlow"
        prunedNet = dlupdate(@(W,M)W.*M, dlnet, pruningMaskSynFlow{selectedIteration});
end
end
```

Pruning Statistics Function

The pruningStatistics function extracts detailed layer-level pruning statistics such as the layer-level sparsity and the number of filters or neurons being pruned.

sparsityPerLayer - percentage of parameters pruned in each layer

prunedChannelsPerLayer - number of channels/neurons in each layer that can be removed as a result of pruning

numOutChannelsPerLayer - number of channels/neurons in each layer

```
function [sparsityPerLayer,prunedChannelsPerLayer,numOutChannelsPerLayer,layerNames] = pruningSt

layerNames = unique(dlnet.Learnables.Layer,'stable');
numLayers = numel(layerNames);
layerIDs = zeros(numLayers,1);
for idx = 1:numel(layerNames)
    layerIDs(idx) = find(layerNames(idx)=={dlnet.Layers.Name});
end

sparsityPerLayer = zeros(numLayers,1);
prunedChannelsPerLayer = zeros(numLayers,1);
numOutChannelsPerLayer = zeros(numLayers,1);

numParams = zeros(numLayers,1);
numPrunedParams = zeros(numLayers,1);
for idx = 1:numLayers
    layer = dlnet.Layers(layerIDs(idx));
```

```

% Calculate the sparsity
paramIDs = strcmp(dlnet.Learnables.Layer,layerNames(idx));
paramValue = dlnet.Learnables.Value(paramIDs);
for p = 1:numel(paramValue)
    numParams(idx) = numParams(idx) + numel(paramValue{p});
    numPrunedParams(idx) = numPrunedParams(idx) + nnz(extractdata(paramValue{p})==0);
end

% Calculate channel statistics
sparsityPerLayer(idx) = numPrunedParams(idx)/numParams(idx);
switch class(layer)
    case "nnet.cnn.layer.FullyConnectedLayer"
        numOutChannelsPerLayer(idx) = layer.OutputSize;
        prunedChannelsPerLayer(idx) = nnz(all(layer.Weights==0,2)&layer.Bias(:)==0);
    case "nnet.cnn.layer.Convolution2DLayer"
        numOutChannelsPerLayer(idx) = layer.NumFilters;
        prunedChannelsPerLayer(idx) = nnz(reshape(all(layer.Weights==0,[1,2,3]),[],1)&layer.L
    case "nnet.cnn.layer.GroupedConvolution2DLayer"
        numOutChannelsPerLayer(idx) = layer.NumGroups*layer.NumFiltersPerGroup;
        prunedChannelsPerLayer(idx) = nnz(reshape(all(layer.Weights==0,[1,2,3]),[],1)&layer.L
    otherwise
        error("Unknown layer: "+class(layer))
end
end
end

```

Load Digits Data set Function

The `loadDigitDataset` function loads the Digits data set and splits the data into training and validation data.

```

function [imdsTrain, imdsValidation] = loadDigitDataset()
digitDatasetPath = fullfile(matlabroot,'toolbox','nnet','nndemos', ...
    'nndatasets','DigitDataset');
imds = imageDatastore(digitDatasetPath, ...
    'IncludeSubfolders',true,'LabelSource','foldernames');
[imdsTrain, imdsValidation] = splitEachLabel(imds,0.75,"randomized");
end

```

Train Digit Recognition Network Function

The `trainDigitDataNetwork` function trains a convolutional neural network to classify digits in grayscale images.

```

function net = trainDigitDataNetwork(imdsTrain,imdsValidation)
layers = [
    imageInputLayer([28 28 1],"Normalization","rescale-zero-one")
    convolution2dLayer(3,8,'Padding','same')
    reluLayer
    maxPooling2dLayer(2,'Stride',2)

    convolution2dLayer(3,16,'Padding','same')
    reluLayer
    maxPooling2dLayer(2,'Stride',2)

    convolution2dLayer(3,32,'Padding','same')
    reluLayer

```

```

    fullyConnectedLayer(10)
    softmaxLayer
    classificationLayer];

% Specify the training options
options = trainingOptions('sgdm', ...
    'InitialLearnRate',0.01, ...
    'MaxEpochs',10, ...
    'Shuffle','every-epoch', ...
    'ValidationData',imdsValidation, ...
    'ValidationFrequency',30, ...
    'Verbose',false, ...
    'Plots','none','ExecutionEnvironment','auto');

% Train network
net = trainNetwork(imdsTrain, layers, options);
end

```

References

- [1] Song Han, Jeff Pool, John Tran, and William J. Dally. 2015. "Learning Both Weights and Connections for Efficient Neural Networks." *Advances in Neural Information Processing Systems 28 (NIPS 2015)*: 1135-1143.
- [2] Hidenori Tanaka, Daniel Kunin, Daniel L. K. Yamins, and Surya Ganguli 2020. "Pruning Neural Networks Without Any Data by Iteratively Conserving Synaptic Flow." *34th Conference on Neural Information Processing Systems (NeurIPS 2020)*

See Also

Apps

Deep Network Quantizer

Functions

dlquantizer | dlquantizationOptions | calibrate | validate |
 coder.loadDeepLearningNetwork | codegen

Objects

coder.CuDNNConfig | coder.TensorRTConfig

More About

- "Quantization of Deep Neural Networks" on page 4-99
- "Generate INT8 Code for Deep Learning Networks" on page 4-107
- "Quantize Residual Network Trained for Image Classification and Generate CUDA Code" on page 4-229
- "Code Generation for Deep Learning Networks by Using cuDNN" on page 4-69
- "Code Generation for Deep Learning Networks by Using TensorRT" on page 4-78

Code Generation For Aerial Lidar Semantic Segmentation Using PointNet++ Deep Learning

This example shows how to generate CUDA® MEX code for a PointNet++ [1 on page 4-271] network for lidar semantic segmentation. This example uses a pretrained PointNet++ network that can segment unorganized lidar point clouds belonging to eight classes (buildings, cars, trucks, poles, power lines, fences, ground, and vegetation). For more information on PointNet++ network, see “Getting Started with PointNet++” (Lidar Toolbox).

Third-Party Prerequisites

Required

- CUDA enabled NVIDIA® GPU and compatible driver.

Optional

For non-MEX builds such as static libraries, dynamic libraries, or executables, this example has the following additional requirements.

- NVIDIA toolkit.
- NVIDIA cuDNN library.
- Environment variables for the compilers and libraries. For more information, see “Third-Party Hardware” and “Setting Up the Prerequisite Products”.

Verify GPU Environment

To verify that the compilers and libraries for running this example are set up correctly, use the `coder.checkGpuInstall` function.

```
envCfg = coder.gpuEnvConfig('host');  
envCfg.DeepLibTarget = 'cudnn';  
envCfg.DeepCodegen = 1;  
envCfg.Quiet = 1;  
coder.checkGpuInstall(envCfg);
```

Load PointNet++ Network

Use the `getPointnetplusNet` function, attached as a supporting file to this example, to load the pretrained PointNet++ network. For more information on how to train this network, see “Aerial Lidar Semantic Segmentation Using PointNet++ Deep Learning” (Lidar Toolbox) example.

```
net = getPointnetplusNet;
```

The pretrained network is a DAG network. To display an interactive visualization of the network architecture, use the `analyzeNetwork` (Deep Learning Toolbox) function.

The sampling and grouping layer, and the interpolation layer are implemented using the `functionLayer` (Deep Learning Toolbox) function. Both `pointCloudInputLayer` and the `functionLayer` functions do not support code generation. For code generation support, replace the function layers with custom layers and the `pointCloudInputLayer` with the `imageInputLayer` by using the `helperReplaceInputAndFunctionLayers` helper function, attached to this example as a support file. This function saves the network as a MAT file with the name `pointnetplusCodegenNet.mat`.

```
net = helperReplaceInputAndFunctionLayers(net);
```

pointnetplusPredict Entry-Point Function

The `pointnetplusPredict` entry-point function takes a point cloud data matrix as input and performs prediction on it by using the deep learning network saved in the `pointnetplusCodegenNet.mat` file. The function loads the network object from the `pointnetplusCodegenNet.mat` file into a persistent variable `mynet` and reuses the persistent variable in subsequent prediction calls.

```
type('pointnetplusPredict.m');

function out = pointnetplusPredict(in)
    %#codegen

    % A persistent object mynet is used to load the DAG network object. At
    % the first call to this function, the persistent object is constructed and
    % setup. When the function is called subsequent times, the same object is
    % reused to call predict on inputs, thus avoiding reconstructing and
    % reloading the network object.

    % Copyright 2021 The MathWorks, Inc.

    persistent mynet;

    if isempty(mynet)
        mynet = coder.loadDeepLearningNetwork('pointnetplusCodegenNet.mat');
    end

    % pass in input
    out = predict(mynet,in);
```

Generate CUDA MEX Code

To generate CUDA® code for the `pointnetplusPredict` entry-point function, create a GPU code configuration object for a MEX target and set the target language to C++. Use the `coder.DeepLearningConfig` function to create a CuDNN deep learning configuration object and assign it to the `DeepLearningConfig` property of the GPU code configuration object. Run the `codegen` command with the size of point cloud data in the input layer of the network, which in this case is [8192 1 3].

```
cfg = coder.gpuConfig('mex');
cfg.TargetLang = 'C++';
cfg.DeepLearningConfig = coder.DeepLearningConfig(TargetLibrary='cudnn');
codegen -config cfg pointnetplusPredict -args {randn(8192,1,3,'single')} -report
```

Code generation successful: [View report](#)

To generate CUDA® code for the TensorRT target, create and use a TensorRT deep learning configuration object instead of the CuDNN configuration object.

Segment Aerial Point Cloud Using Generated MEX Code

The network in this example is trained on the DALES data set [2 on page 4-271]. Follow the instructions on the DALES website to download the data set to the folder specified by the `dataFolder` variable. Create a folder to store the test data.

```
dataFolder = fullfile(tempdir, 'DALES');
testDataFolder = fullfile(dataFolder, 'dales_las', 'test');
```

Each point cloud in the DALES dataset covers an area of 500-by-500 meters, which is much larger than the typical area covered by terrestrial lidar point clouds. For efficient memory processing, divide the point cloud into small, non-overlapping blocks by using a `blockedPointCloud` (Lidar Toolbox) object.

Define the block dimensions using the `blockSize` parameter. As the size of each point cloud in the dataset varies, set the z-dimension of the block to `Inf` to avoid block creation along z-axis.

```
blockSize = [51 51 Inf];
```

First, create a `blockedPointCloud` (Lidar Toolbox) object. Then, create a `blockedPointCloudDatastore` (Lidar Toolbox) object on the test data using the `blockedPointCloud` (Lidar Toolbox) object.

```
tbbc = blockedPointCloud(fullfile(testDataFolder, '5080_54470.las'), blockSize);
tbpcds = blockedPointCloudDatastore(tbbc);
```

Define the parameters used to train the network. For more details, see the “Aerial Lidar Semantic Segmentation Using PointNet++ Deep Learning” (Lidar Toolbox) example.

```
numNearestNeighbors = 20;
radius = 0.05;
numPoints = 8192;
maxLabel = 1;
classNames = [
    "ground"
    "vegetation"
    "cars"
    "trucks"
    "powerlines"
    "fences"
    "poles"
    "buildings"
];
numClasses = numel(classNames);
```

Initialize placeholders for the predicted and target labels.

```
labelsDensePred = [];
labelsDenseTarget = [];
```

Apply the same transformation used on training data to the test data, `tbpcds`, follow these steps.

- Extract the point cloud.
- Downsample the point cloud to a specified number, `numPoints`.
- Normalize the point clouds to the range [0 1].
- Convert the point cloud to make it compatible with the input layer of the network.

Perform inference on the test point cloud data to compute prediction labels. Predict the labels of the sparse point cloud using the `pointnetplusPredict_mex` function. Then interpolate the prediction labels of the sparse point cloud to obtain prediction labels of the dense point cloud and iterate this process on all the non-overlapping blocks.

```

while hasdata(tbpcds)

    % Read the block along with block information.
    [ptCloudDense,infoDense] = read(tbpcds);

    % Extract the labels from the block information.
    labelsDense = infoDense.PointAttributes.Classification;

    % Select only labeled data.
    ptCloudDense = select(ptCloudDense{1},labelsDense~=0);
    labelsDense = labelsDense(labelsDense~=0);

    % Use the helperDownsamplePoints function, attached to this example as a
    % supporting file, to extract a downsampled point cloud from the
    % dense point cloud.
    ptCloudSparse = helperDownsamplePoints(ptCloudDense, ...
        labelsDense,numPoints);

    % Make the spatial extent of the dense point cloud equal to the sparse
    % point cloud.
    limits = [ptCloudDense.XLimits;ptCloudDense.YLimits;ptCloudDense.ZLimits];
    ptCloudSparseLocation = ptCloudSparse.Location;
    ptCloudSparseLocation(1:2,:) = limits(:,1:2)';
    ptCloudSparse = pointCloud(ptCloudSparseLocation,Color=ptCloudSparse.Color, ...
        Intensity=ptCloudSparse.Intensity, Normal=ptCloudSparse.Normal);

    % Use the helperNormalizePointCloud function, attached to this example as
    % a supporting file, to normalize the point cloud between 0 and 1.
    ptCloudSparseNormalized = helperNormalizePointCloud(ptCloudSparse);
    ptCloudDenseNormalized = helperNormalizePointCloud(ptCloudDense);

    % Use the helperTransformToTestData function, defined at the end of this
    % example, to convert the point cloud to a cell array and to permute the
    % dimensions of the point cloud to make it compatible with the input layer
    % of the network.
    ptCloudSparseForPrediction = helperTransformToTestData(ptCloudSparseNormalized);

    % Get the output predictions.
    scoresPred = pointnetplusPredict_mex(single(ptCloudSparseForPrediction{1,1}));
    [~,labelsSparsePred] = max(scoresPred,[],3);
    labelsSparsePred = uint8(labelsSparsePred);

    % Use the helperInterpolate function, attached to this example as a
    % supporting file, to calculate labels for the dense point cloud,
    % using the sparse point cloud and labels predicted on the sparse point cloud.
    interpolatedLabels = helperInterpolate(ptCloudDenseNormalized, ...
        ptCloudSparseNormalized,labelsSparsePred,numNearestNeighbors, ...
        radius,maxLabel,numClasses);

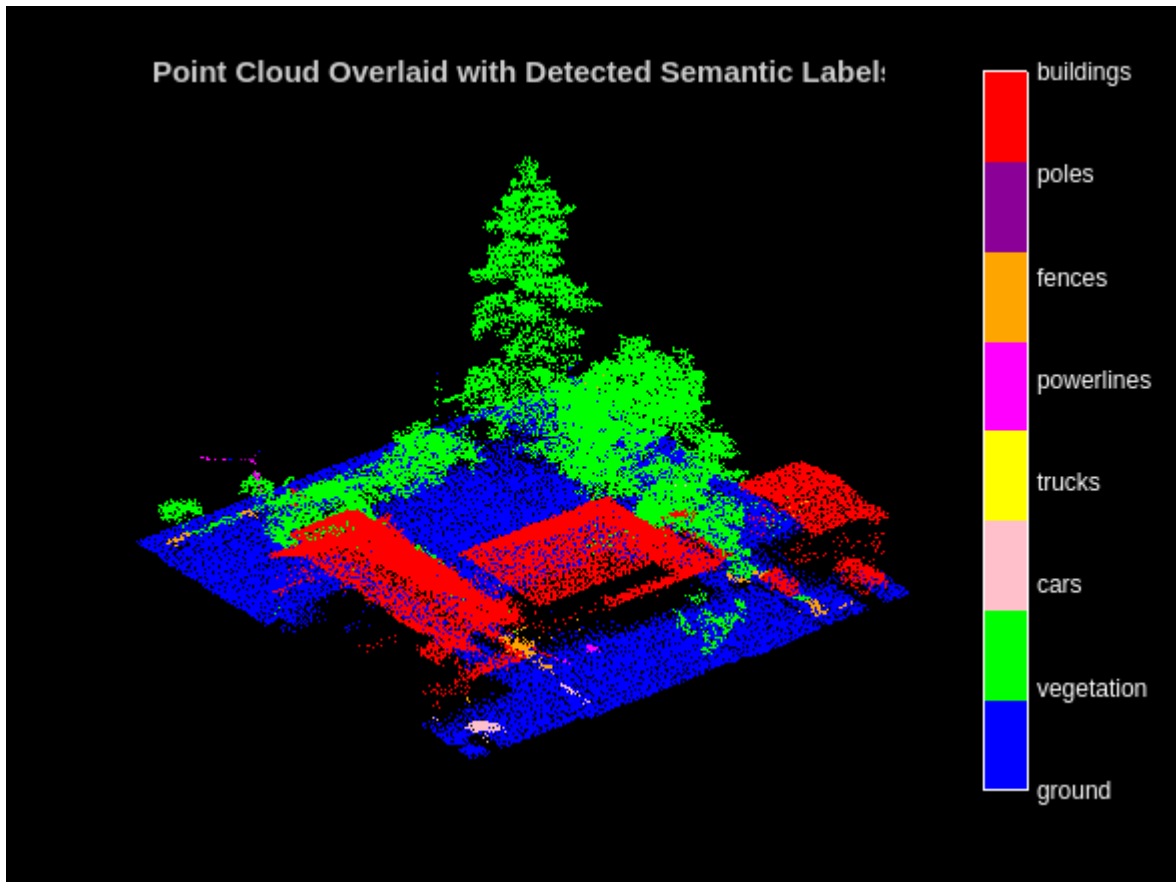
    % Concatenate the predicted and target labels from the blocks.
    labelsDensePred = vertcat(labelsDensePred,interpolatedLabels);
    labelsDenseTarget = vertcat(labelsDenseTarget,labelsDense);
end

```

Starting parallel pool (parpool) using the 'Processes' profile ...
Connected to the parallel pool (number of workers: 6).

For better visualisation, display a single block inferred from the point cloud data.

```
figure;
ax = pcshow(ptCloudDense.Location,interpolatedLabels);
axis off;
helperLabelColorbar(ax,classNames);
title("Point Cloud Overlaid with Detected Semantic Labels");
```



Supporting Functions

The `helperLabelColorbar` function adds a colorbar to the current axis. The colorbar is formatted to display the class names with the color.

```
function helperLabelColorbar(ax,classNames)
% Colormap for the original classes.
cmap = [[0,0,255];
        [0,255,0];
        [255,192,203];
        [255,255,0];
        [255,0,255];
        [255,165,0];
        [139,0,150];
        [255,0,0]];
cmap = cmap./255;
cmap = cmap(1:numel(classNames),:);
colormap(ax,cmap);

% Add colorbar to current figure.
```



```

c = colorbar(ax);
c.Color = 'w';

% Center tick labels and use class names for tick marks.
numClasses = size(classNames, 1);
c.Ticks = 1:1:numClasses;
c.TickLabels = classNames;

% Remove tick mark.
c.TickLength = 0;
end

```

The `helperTransformToTestData` function converts the point cloud into a cell array and permutes the dimensions of the point cloud to make it compatible with the input layer of the network.

```

function data = helperTransformToTestData(data)
if ~iscell(data)
    data = {data};
end
numObservations = size(data,1);
for i = 1:numObservations
    tmp = data{i,1}.Location;
    data{i,1} = permute(tmp,[1 3 2]);
end
end

```

References

[1] Qi, Charles R., Li Yi, Hao Su, and Leonidas J. Guibas. "PointNet++: Deep Hierarchical Feature Learning on Point Sets in a Metric Space." *ArXiv:1706.02413 [Cs]*, June 7, 2017. <https://arxiv.org/abs/1706.02413>.

[2] Varney, Nina, Vijayan K. Asari, and Quinn Graehling. "DALES: A Large-Scale Aerial LiDAR Data Set for Semantic Segmentation." *ArXiv:2004.11985 [Cs, Stat]*, April 14, 2020. <https://arxiv.org/abs/2004.11985>.

See Also

Functions

`coder.checkGpuInstall` | `codegen` | `coder.DeepLearningConfig` | `coder.loadDeepLearningNetwork`

Objects

`coder.gpuConfig` | `coder.gpuEnvConfig` | `coder.TensorRTConfig` | `coder.CuDNNConfig`

Related Examples

- "Lidar Point Cloud Semantic Segmentation Using PointSeg Deep Learning Network" (Lidar Toolbox)
- "Code Generation for Lidar Point Cloud Segmentation Network" on page 4-204

Code Generation For Lidar Object Detection Using PointPillars Deep Learning

This example shows how to generate CUDA® MEX for a PointPillars object detector. For more information, see “Lidar 3-D Object Detection Using PointPillars Deep Learning” (Lidar Toolbox) example from the Lidar Toolbox™.

Third-Party Prerequisites

Required

- CUDA enabled NVIDIA® GPU and compatible driver.

Optional

For non-MEX builds such as static and dynamic libraries or executables, this example has the following additional requirements.

- NVIDIA CUDA toolkit.
- NVIDIA cuDNN library.
- Environment variables for the compilers and libraries. For more information, see “Third-Party Hardware” and “Setting Up the Prerequisite Products”.

Verify GPU Environment

To verify that the compilers and libraries for running this example are set up correctly, use the `coder.checkGpuInstall` function.

```
envCfg = coder.gpuEnvConfig('host');  
envCfg.DeepLibTarget = 'cudnn';  
envCfg.DeepCodegen = 1;  
envCfg.Quiet = 1;  
coder.checkGpuInstall(envCfg);
```

Pretrained PointPillars Network

Load the pretrained `pointPillarsObjectDetector` (Lidar Toolbox) trained in the *Lidar 3-D Object Detection Using PointPillars Deep Learning example*. To train the detector yourself, see “Lidar 3-D Object Detection Using PointPillars Deep Learning” (Lidar Toolbox).

```
matFile = 'pretrainedPointPillarsDetector.mat';  
pretrainedDetector = load('pretrainedPointPillarsDetector.mat', 'detector');  
detector = pretrainedDetector.detector;
```

pointpillarsDetect Entry-Point Function

The `pointpillarsDetect` entry-point function takes in the point cloud and confidence threshold as input and passes them to a trained `pointPillarsObjectDetector` (Lidar Toolbox) for prediction through the `pointpillarDetect` function. The `pointpillarsDetect` function loads the detector object from the MAT file into a persistent variable and reuses the persistent object for subsequent prediction calls.

```
type('pointpillarsDetect.m')
```

```
function [bboxes,scores,labels] = pointpillarsDetect(matFile,dataLoc,dataInt,threshold)
% Predict the output of network and extract the confidence, x, y,
% width, height, and class.

% load the deep learning network for prediction
persistent pointPillarObj;

if isempty(pointPillarObj)
    pointPillarObj = coder.loadDeepLearningNetwork(matFile);
end

ptCloud = pointCloud(dataLoc,'Intensity',dataInt);

[bboxes,scores,labels] = pointPillarObj.detect(ptCloud,'Threshold',threshold);
end
```

Evaluate the detector for Object Detection

Read the point cloud.

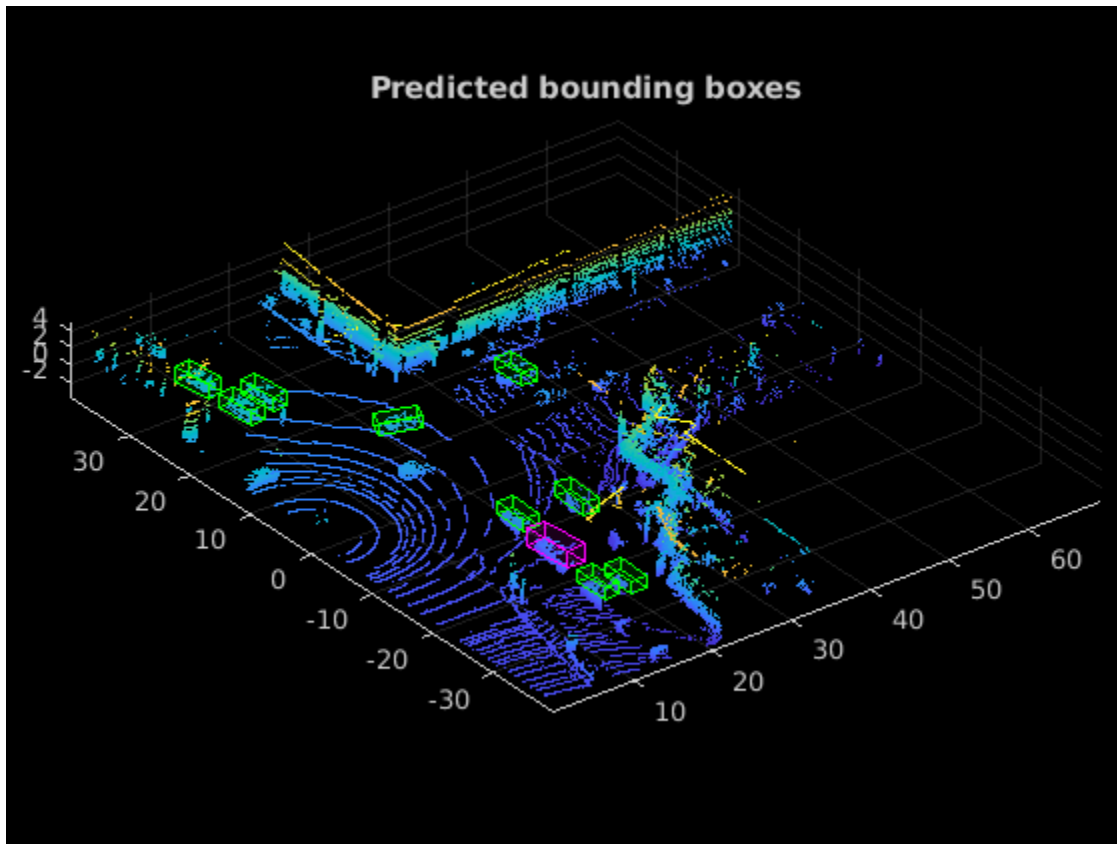
```
pc = pcread('pandasetDrivingData.pcd');
```

Use the detect method on the pretrained detector.

```
confidenceThreshold = 0.7;
[bboxes,~,labels] = detect(detector,pc,'Threshold',confidenceThreshold);
bboxesCar = bboxes(labels == 'Car',:);
bboxesTruck = bboxes(labels == 'Truck',:);
```

Display the detections on the point cloud.

```
helperDisplay3DBoxesOverlaidPointCloud(pc.Location,bboxesCar,'green',...
    bboxesTruck,'magenta','Predicted bounding boxes');
```



Generate CUDA MEXscatter

To generate CUDA® code for the `pointpillarsDetect` entry-point function, create a GPU code configuration object for a MEX target and set the target language to C++. Use the `coder.DeepLearningConfig` function to create a cuDNN deep learning configuration object and assign it to the `DeepLearningConfig` property of the GPU code configuration object.

```
cfg = coder.gpuConfig('mex');
cfg.TargetLang = 'C++';
cfg.DeepLearningConfig = coder.DeepLearningConfig(TargetLibrary='cudnn');

dataLoc = pc.Location;
dataInt = pc.Intensity;

args = {coder.Constant(matFile) coder.typeof(dataLoc,[Inf,3],[1 0]) coder.typeof(dataInt,[Inf,1])};

codegen -config cfg pointpillarsDetect -args args -report
```

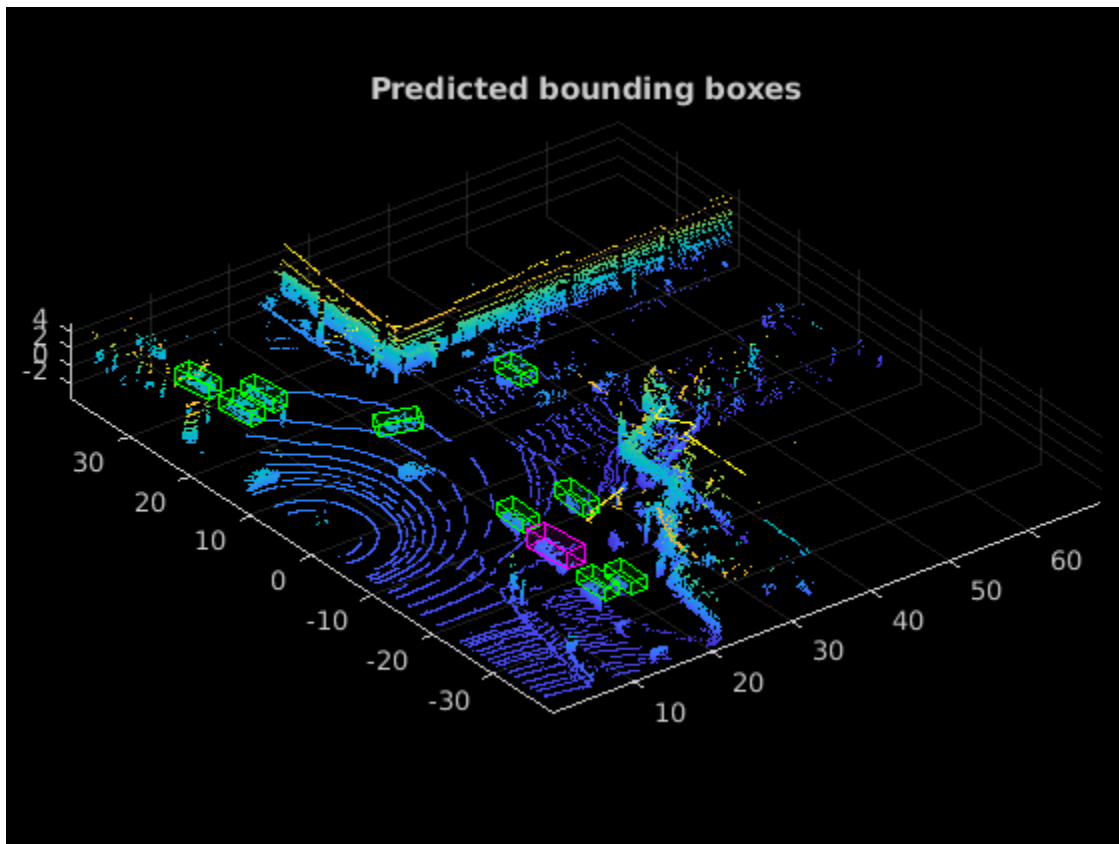
Code generation successful: [View report](#)

Run the Generated MEX

Call the generated CUDA MEX with the point cloud. Display the results.

```
[bboxes,~,labels] = pointpillarsDetect_mex(matFile,dataLoc,dataInt,confidenceThreshold);
bboxesCar = bboxes(labels == 'Car',:);
bboxesTruck = bboxes(labels == 'Truck',:);
```

```
helperDisplay3DBoxesOverlaidPointCloud(pc.Location,bboxesCar,'green',...
    bboxesTruck,'magenta','Predicted bounding boxes');
```



Helper Functions

```
function helperDisplay3DBoxesOverlaidPointCloud(ptCld,labelsCar,carColor,...
    labelsTruck,truckColor,titleForFigure)
% Display the point cloud with different colored bounding boxes for different
% classes
figure;
ax = pcshow(ptCld);
showShape('cuboid',labelsCar,'Parent',ax,'Opacity',0.1,'Color',...
    carColor,'LineWidth',0.5);
hold on;
showShape('cuboid',labelsTruck,'Parent',ax,'Opacity',0.1,'Color',...
    truckColor,'LineWidth',0.5);
title(titleForFigure);
zoom(ax,1.5);
end
```

References

[1] Lang, Alex H., Sourabh Vora, Holger Caesar, Lubing Zhou, Jiong Yang, and Oscar Beijbom. "PointPillars: Fast Encoders for Object Detection From Point Clouds." *In 2019 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 12689-12697. Long Beach, CA, USA: IEEE, 2019. <https://doi.org/10.1109/CVPR.2019.01298>.

[2] Hesai and Scale. PandaSet. <https://scale.com/open-datasets/pandaset>.

Code Generation for Object Detection Using YOLO v4 Deep Learning

This example shows how to generate standalone CUDA® executable for a You Only Look Once v4 (YOLO v4) object detector. This example uses a lightweight version of the YOLO v4 network with fewer network layers. It uses a feature pyramid network as the neck and has two YOLO v4 detection heads. The network was trained on the COCO dataset. For more information about the YOLO v4 object detection network, see “Getting Started with YOLO v4” (Computer Vision Toolbox) and `yolov4objectDetector` (Computer Vision Toolbox).

Third-Party Prerequisites

Required

- CUDA enabled NVIDIA® GPU and compatible driver

For non-MEX builds, such as static, dynamic libraries or executables, this example additionally requires:

- NVIDIA CUDA toolkit.
- NVIDIA cuDNN library.
- Environment variables for the compilers and libraries. For more information, see “Third-Party Hardware” and “Setting Up the Prerequisite Products”.

Verify GPU Environment

To verify that the compilers and libraries for this example are set up correctly, use the `coder.checkGpuInstall` function.

```
envCfg = coder.gpuEnvConfig('host');
envCfg.DeepLibTarget = 'cudnn';
envCfg.DeepCodegen = 1;
envCfg.Quiet = 1;
coder.checkGpuInstall(envCfg);
```

Load Pretrained Network

This example uses a pretrained YOLO v4 object detection network trained on the COCO dataset. The object detector can detect and identify 80 different objects. To use this network, download and install the Computer Vision Toolbox Model for YOLO v4 Object Detection from Add-On Explorer. For more information about installing add-ons, see “Get and Manage Add-Ons”.

Specify the name for the network and save the network to a MAT-file.

```
name = "tiny-yolov4-coco";
vehicleDetector = yolov4objectDetector(name);
save('tinyyolov4coco.mat', 'vehicleDetector');
net = vehicleDetector.Network;
disp(vehicleDetector)
```

`yolov4objectDetector` with properties:

```
Network: [1×1 dlnetwork]
AnchorBoxes: {2×1 cell}
```

```
ClassNames: {80x1 cell}
InputSize: [416 416 3]
ModelName: 'tiny-yolov4-coco'
```

Download Test Traffic Video

To test the model, download the video file from the MathWorks website. The file is approximately 40 MB in size.

```
if ~exist('./downtown_short.mp4', 'file')
    url = 'https://www.mathworks.com/supportfiles/gpuCoder/media/downtown_short.mp4';
    websave('downtown_short.mp4', url);
end
```

The `tinyyolov4cocoDetect` Entry-Point Function

The `tinyyolov4Detect` entry-point function runs the detector on the video file by using the deep learning network in the `tinyyolov4coco.mat` file. The function loads the network object from the `tinyyolov4coco.mat` file into a persistent variable `yolov4obj` and reuses the persistent object during subsequent detection calls. Then it sets up the video file reader to read the input video and creates a video player to display the video and the output detections.

```
type('tinyyolov4cocoDetect.m')

function tinyyolov4cocoDetect()
    %#codegen

    % Copyright 2022 The MathWorks, Inc.

    persistent yolov4obj;

    if isempty(yolov4obj)
        yolov4obj = coder.loadDeepLearningNetwork('tinyyolov4coco.mat');
    end

    % Read the input video and create a video player
    videoFile = 'downtown_short.mp4';

    videoFreader = vision.VideoFileReader(videoFile, 'VideoOutputDataType', 'uint8');
    depVideoPlayer = vision.DeployableVideoPlayer();

    cont = ~isDone(videoFreader);
    while cont
        I = step(videoFreader);
        in = imresize(I, [416,416]);
        % Call to detect method
        [bboxes, ~, labels] = detect(yolov4obj, in, Threshold = 0.3);

        % Convert categorical labels to cell array of character vectors
        labels = cellstr(labels);

        % Annotate detections in the image.
        outImg = insertObjectAnnotation(in, 'rectangle', bboxes, labels);

        step(depVideoPlayer, outImg); % display video
        cont = ~isDone(videoFreader);
        % pause(0.05); % adjust frame rate
    end
end
```


Generate Executable

To generate CUDA executable code for the entry-point function, create a GPU code configuration object and set the target language to C++. Use the `coder.DeepLearningConfig` function to create a CuDNN deep learning configuration object and assign it to the `DeepLearningConfig` property of the GPU code configuration object. Then run the `codegen` command.

```
cfg = coder.gpuConfig('exe');  
cfg.GenerateExampleMain = 'GenerateCodeAndCompile';  
cfg.DeepLearningConfig = coder.DeepLearningConfig('cudnn');
```

```
codegen -config cfg tinyyolov4cocoDetect -report
```

```
Code generation successful: View report
```

Execute Standalone Code

When you run the generated standalone executable, it displays the detection results frame-by-frame.



References

[1] Bochkovskiy, Alexey, Chien-Yao Wang, and Hong-Yuan Mark Liao. "YOLOv4: Optimal Speed and Accuracy of Object Detection." arXiv, April 22, 2020. <http://arxiv.org/abs/2004.10934>.

[2] Redmon, Joseph, Santosh Divvala, Ross Girshick, and Ali Farhadi. "You Only Look Once: Unified, Real-Time Object Detection." In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 779-88. Las Vegas, NV, USA: IEEE, 2016. <https://doi.org/10.1109/CVPR.2016.91>.

[3] Lin, Tsung-Yi, Michael Maire, Serge Belongie, James Hays, Pietro Perona, Deva Ramanan, Piotr Dollár, and C. Lawrence Zitnick. "Microsoft COCO: Common Objects in Context." In *Computer Vision - ECCV 2014*, edited by David Fleet, Tomas Pajdla, Bernt Schiele, and Tinne Tuytelaars, 740-55. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2014. https://doi.org/10.1007/978-3-319-10602-1_48.

See Also

Functions

`coder.checkGpuInstall` | `codegen` | `coder.DeepLearningConfig` |
`coder.loadDeepLearningNetwork`

Objects

`coder.gpuConfig` | `coder.gpuEnvConfig` | `coder.CuDNNConfig` | `vision.VideoFileReader` |
`vision.DeployableVideoPlayer` | `yolov4ObjectDetector`

Related Examples

- "Object Detection Using YOLO v4 Deep Learning" (Computer Vision Toolbox)
- "Code Generation for Object Detection by Using YOLO v2" on page 4-180
- "Code Generation For Object Detection Using YOLO v3 Deep Learning" on page 4-217

More About

- "Getting Started with YOLO v4" (Computer Vision Toolbox)

Targeting Embedded GPU Devices

- “Build and Run an Executable on NVIDIA Hardware” on page 5-2
- “Build and Run an Executable on NVIDIA Hardware Using GPU Coder App” on page 5-7
- “Relocate Generated Code to Another Development Environment” on page 5-14
- “Getting Started with the MATLAB Coder Support Package for NVIDIA Jetson and NVIDIA DRIVE Platforms” on page 5-24
- “Sobel Edge Detection on NVIDIA Jetson Nano Using Raspberry Pi Camera Module V2” on page 5-29
- “Semantic Segmentation on NVIDIA DRIVE” on page 5-34
- “Top-Hat Filtering to Remove Uneven Background Illumination on NVIDIA Jetson TX2 Developer Kit” on page 5-39
- “Deployment and Classification of Webcam Images on NVIDIA Jetson TX2 Platform” on page 5-44
- “Ground Plane Segmentation and Obstacle Detection on NVIDIA Jetson Xavier™ NX Embedded platform” on page 5-49
- “Deploy Signal Classifier on NVIDIA Jetson Using Wavelet Analysis and Deep Learning” on page 5-57

You can use GPU Coder to generate CUDA code for targeting embedded GPU platforms. Specifically, you can target the NVIDIA development boards such as Jetson AGX, Nano, TX2, TX1, and DRIVE PX2 platforms from either Windows or Linux host development systems.

Build and Run an Executable on NVIDIA Hardware

In this section...

“Learning Objectives” on page 5-2

“Tutorial Prerequisites” on page 5-2

“Example: Vector Addition” on page 5-3

“Create a Live Hardware Connection Object” on page 5-3

“Generate CUDA Executable Using GPU Coder” on page 5-4

“Run the Executable and Verify the Results” on page 5-5

Using GPU Coder and the MATLAB Coder Support Package for NVIDIA Jetson and NVIDIA DRIVE Platforms, you can target NVIDIA DRIVE and Jetson hardware platforms. After connecting to the hardware platforms, you can perform basic operations, generate CUDA executable from a MATLAB entry-point function, and run the executable on the hardware.

Note Starting in R2021a, the GPU Coder Support Package for NVIDIA GPUs is named MATLAB Coder Support Package for NVIDIA Jetson and NVIDIA DRIVE Platforms. To use this support package in R2021a, you must have the MATLAB Coder product.

Learning Objectives

In this tutorial, you learn how to:

- Prepare your MATLAB code for CUDA code generation by using the `kernel fun` pragma.
- Connect to the NVIDIA target board.
- Generate and deploy a CUDA executable on the target board.
- Run the executable on the board and verify the results.

Tutorial Prerequisites

Target Board Requirements

- NVIDIA DRIVE PX2 or Jetson embedded platform.
- Ethernet crossover cable to connect the target board and host PC (if the target board cannot be connected to a local network).
- NVIDIA CUDA Toolkit installed on the board.
- Environment variables on the target for the compilers and libraries. For information on the supported versions of the compilers, libraries, and their setup, see “Install and Setup Prerequisites for NVIDIA Boards” (MATLAB Coder Support Package for NVIDIA Jetson and NVIDIA DRIVE Platforms).

Development Host Requirements

- NVIDIA CUDA Toolkit on the host.
- Environment variables on the host for the compilers and libraries. For information on the supported versions of the compilers and libraries, see “Third-Party Hardware”. For setting up the environment variables, see “Environment Variables”.

Example: Vector Addition

This tutorial uses a simple vector addition example to demonstrate the build and deployment workflow on NVIDIA GPUs. Create a MATLAB function `myAdd.m` that acts as the entry-point for code generation. Alternatively, use the files in the “Getting Started with the MATLAB Coder Support Package for NVIDIA Jetson and NVIDIA DRIVE Platforms” on page 5-24 example for this tutorial. The easiest way to create CUDA code for this function is to place the `coder.gpu.kernelfun` pragma in the function. When the GPU Coder encounters `kernelfun` pragma, it attempts to parallelize the computations within this function and map them to the GPU.

```
function out = myAdd(inp1,inp2) %#codegen
coder.gpu.kernelfun();
out = inp1 + inp2;
end
```

Create a Live Hardware Connection Object

The support package software uses an SSH connection over TCP/IP to execute commands while building and running the generated CUDA code on the DRIVE or Jetson platforms. Connect the target platform to the same network as the host computer or use an Ethernet crossover cable to connect the board directly to the host computer. Refer to the NVIDIA documentation on how to set up and configure your board.

To communicate with the NVIDIA hardware, you must create a live hardware connection object by using the `jetson` or `drive` function. To create a live hardware connection object using the function, provide the host name or IP address, user name, and password of the target board. For example to create live object for Jetson hardware:

```
hwobj = jetson('jetson-board-name', 'ubuntu', 'ubuntu');
```

The software performs a check of the hardware, compiler tools, libraries, IO server installation, and gathers peripheral information on target. This information is displayed in the command window.

```
Checking for CUDA availability on the Target...
Checking for NVCC in the target system path...
Checking for CUDNN library availability on the Target...
Checking for TensorRT library availability on the Target...
Checking for Prerequisite libraries is now complete.
Gathering hardware details...
Checking for third-party library availability on the Target...
Gathering hardware details is complete.
Board name       : NVIDIA Jetson TX2
CUDA Version     : 10.0
cuDNN Version    : 7.6
TensorRT Version : 6.0
GStreamer Version : 1.14.5
V4L2 Version     : 1.14.2-1
SDL Version      : 1.2
OpenCV Version   : 4.1.1
Available Webcams : UVC Camera (046d:0809)
Available GPUs   : NVIDIA Tegra X2
```

Alternatively, to create live object for DRIVE hardware:

```
hwobj = drive('drive-board-name', 'nvidia', 'nvidia');
```

Note If there is a connection failure, a diagnostics error message is reported on the MATLAB command window. If the connection has failed, the most likely cause is incorrect IP address or host name.

Generate CUDA Executable Using GPU Coder

To generate a CUDA executable that can be deployed to a NVIDIA target, create a custom main file (`main.cu`) and header file (`main.h`). The main file calls the code generated for the MATLAB entry-point function. The main file passes a vector containing the first 100 natural numbers to the entry-point function and writes the results to a binary file (`myAdd.bin`).

`main.cu`

```
//main.cu
// Include Files
#include "myAdd.h"
#include "main.h"
#include "myAdd_terminate.h"
#include "myAdd_initialize.h"
#include <stdio.h>

// Function Declarations
static void argInit_1x100_real_T(real_T result[100]);
static void main_myAdd();

// Function Definitions
static void argInit_1x100_real_T(real_T result[100])
{
    int32_T idx1;

    // Initialize each element.
    for (idx1 = 0; idx1 < 100; idx1++) {
        result[idx1] = (real_T) idx1;
    }
}

void writeToFile(real_T result[100])
{
    FILE *fid = NULL;
    fid = fopen("myAdd.bin", "wb");
    fwrite(result, sizeof(real_T), 100, fid);
    fclose(fid);
}

static void main_myAdd()
{
    real_T out[100];
    real_T b[100];
    real_T c[100];

    argInit_1x100_real_T(b);
    argInit_1x100_real_T(c);

    myAdd(b, c, out);
    writeToFile(out); // Write the output to a binary file
}
```



```
// Main routine
int32_T main(int32_T, const char * const [])
{
    // Initialize the application.
    myAdd_initialize();

    // Invoke the entry-point functions.
    main_myAdd();

    // Terminate the application.
    myAdd_terminate();
    return 0;
}
```

main.h

```
//main.h
#ifndef MAIN_H
#define MAIN_H

// Include Files
#include <stddef.h>
#include <stdlib.h>
#include "rtwtypes.h"
#include "myAdd_types.h"

// Function Declarations
extern int32_T main(int32_T argc, const char * const argv[]);

#endif
```

Create a GPU code configuration object for generating an executable. Use the `coder.hardware` function to create a configuration object for the DRIVE or Jetson platform and assign it to the `Hardware` property of the code configuration object `cfg`. Use the `BuildDir` property to specify the folder for performing remote build process on the target. If the specified build folder does not exist on the target, then the software creates a folder with the given name. If no value is assigned to `cfg.Hardware.BuildDir`, the remote build process happens in the last specified build folder. If there is no stored build folder value, the build process takes place in the home folder.

```
cfg = coder.gpuConfig('exe');
cfg.Hardware = coder.hardware('NVIDIA Jetson');
cfg.Hardware.BuildDir = '~/remoteBuildDir';
cfg.CustomSource = fullfile('main.cu');
```

To generate CUDA code, use the `codegen` command and pass the GPU code configuration object along with the size of the inputs for and `myAdd` entry-point function. After the code generation takes place on the host, the generated files are copied over and built on the target.

```
codegen('-config ',cfg,'myAdd','-args',{1:100,1:100});
```

Run the Executable and Verify the Results

To run the executable on the target hardware, use the `runApplication()` method of the hardware object. In the MATLAB command window, enter:

```
pid = runApplication(hwobj,'myAdd');
```

```
### Launching the executable on the target...
Executable launched successfully with process ID 26432.
Displaying the simple runtime log for the executable...
```

Copy the output bin file `myAdd.bin` to the MATLAB environment on the host and compare the computed results with the results from MATLAB.

```
outputFile = [hwobj.workspaceDir '/myAdd.bin']
getFile(hwobj,outputFile);

% Simulation result from the MATLAB.
simOut = myAdd(0:99,0:99);

% Read the copied result binary file from target in MATLAB.
fId = fopen('myAdd.bin','r');
tOut = fread(fId,'double');
diff = simOut - tOut';
fprintf('Maximum deviation : %f\n', max(diff(:)));

Maximum deviation between MATLAB Simulation output and GPU coder output on Target is: 0.000000
```

See Also

Objects

jetson | drive

More About

- “Build and Run an Executable on NVIDIA Hardware Using GPU Coder App” on page 5-7
- “Code Generation Using the Command Line Interface”
- “Code Generation by Using the GPU Coder App”
- “Code Generation for Deep Learning Networks by Using cuDNN” on page 4-69
- “Code Generation for Deep Learning Networks by Using TensorRT” on page 4-78
- “Stop or Restart an Executable Running on NVIDIA Hardware” (MATLAB Coder Support Package for NVIDIA Jetson and NVIDIA DRIVE Platforms)
- “Run Linux Commands on NVIDIA Hardware” (MATLAB Coder Support Package for NVIDIA Jetson and NVIDIA DRIVE Platforms)

Build and Run an Executable on NVIDIA Hardware Using GPU Coder App

In this section...

“Learning Objectives” on page 5-7
“Tutorial Prerequisites” on page 5-7
“Example: Vector Addition” on page 5-8
“Custom Main File” on page 5-8
“GPU Coder App” on page 5-9
“Run the Executable and Verify the Results” on page 5-12

Using GPU Coder and the MATLAB Coder Support Package for NVIDIA Jetson and NVIDIA DRIVE Platforms, you can target NVIDIA DRIVE and Jetson hardware platforms. After connecting to the target platform, you can perform basic operations, generate CUDA executable from a MATLAB function, and run the executable on the hardware. The support package automates the deployment of the generated CUDA code on GPU hardware platforms such as Jetson or DRIVE

Note Starting in R2021a, the GPU Coder Support Package for NVIDIA GPUs is named MATLAB Coder Support Package for NVIDIA Jetson and NVIDIA DRIVE Platforms. To use this support package in R2021a, you must have the MATLAB Coder product.

Learning Objectives

In this tutorial, you learn how to:

- Prepare your MATLAB code for CUDA code generation by using the `kernel fun` pragma.
- Create and set up a GPU Coder project.
- Change settings to connect to the NVIDIA target board.
- Generate and deploy a CUDA executable on the target board.
- Run the executable on the board and verify the results.

Before following getting started with this tutorial, it is recommended to familiarize yourself with the GPU Coder App. For more information, see “Code Generation by Using the GPU Coder App”.

Tutorial Prerequisites

Target Board Requirements

- NVIDIA DRIVE PX2 or Jetson embedded platform.
- Ethernet crossover cable to connect the target board and host PC (if the target board cannot be connected to a local network).
- NVIDIA CUDA Toolkit installed on the board.
- Environment variables on the target for the compilers and libraries. For information on the supported versions of the compilers, libraries, and their setup, see “Install and Setup Prerequisites for NVIDIA Boards” (MATLAB Coder Support Package for NVIDIA Jetson and NVIDIA DRIVE Platforms).

Development Host Requirements

- NVIDIA CUDA Toolkit on the host.
- Environment variables on the host for the compilers and libraries. For information on the supported versions of the compilers and libraries, see “Third-Party Hardware”. For setting up the environment variables, see “Environment Variables”.

Example: Vector Addition

This tutorial uses a simple vector addition example to demonstrate the build and deployment workflow on NVIDIA GPUs. Create a MATLAB function `myAdd.m` that acts as the entry-point for code generation. Alternatively, use the files in the “Getting Started with the MATLAB Coder Support Package for NVIDIA Jetson and NVIDIA DRIVE Platforms” on page 5-24 example for this tutorial. The easiest way to create CUDA code for this function is to place the `coder.gpu.kernelfun` pragma in the function. When the GPU Coder encounters `kernelfun` pragma, it attempts to parallelize the computations within this function and maps them to the GPU.

```
function out = myAdd(inp1,inp2) %#codegen
coder.gpu.kernelfun();
out = inp1 + inp2;
end
```

Custom Main File

To generate a CUDA executable that can be deployed to a NVIDIA target, create a custom main file (`main.cu`) and header file (`main.h`). The main file calls the code generated for the MATLAB entry-point function. The main file passes a vector containing the first 100 natural numbers to the entry-point function and writes the results to a binary file (`myAdd.bin`).

`main.cu`

```
//main.cu
// Include Files
#include "myAdd.h"
#include "main.h"
#include "myAdd_terminate.h"
#include "myAdd_initialize.h"
#include <stdio.h>

// Function Declarations
static void argInit_1x100_real_T(real_T result[100]);
static void main_myAdd();

// Function Definitions
static void argInit_1x100_real_T(real_T result[100])
{
    int32_T idx1;

    // Initialize each element.
    for (idx1 = 0; idx1 < 100; idx1++) {
        result[idx1] = (real_T) idx1;
    }
}

void writeToFile(real_T result[100])
```

```

{
    FILE *fid = NULL;
    fid = fopen("myAdd.bin", "wb");
    fwrite(result, sizeof(real_T), 100, fid);
    fclose(fid);
}

static void main_myAdd()
{
    real_T out[100];
    real_T b[100];
    real_T c[100];

    argInit_1x100_real_T(b);
    argInit_1x100_real_T(c);

    myAdd(b, c, out);
    writeToFile(out); // Write the output to a binary file
}

// Main routine
int32_T main(int32_T, const char * const [])
{
    // Initialize the application.
    myAdd_initialize();

    // Invoke the entry-point functions.
    main_myAdd();

    // Terminate the application.
    myAdd_terminate();
    return 0;
}

```

main.h

```

//main.h
#ifndef MAIN_H
#define MAIN_H

// Include Files
#include <stddef.h>
#include <stdlib.h>
#include "rtwtypes.h"
#include "myAdd_types.h"

// Function Declarations
extern int32_T main(int32_T argc, const char * const argv[]);

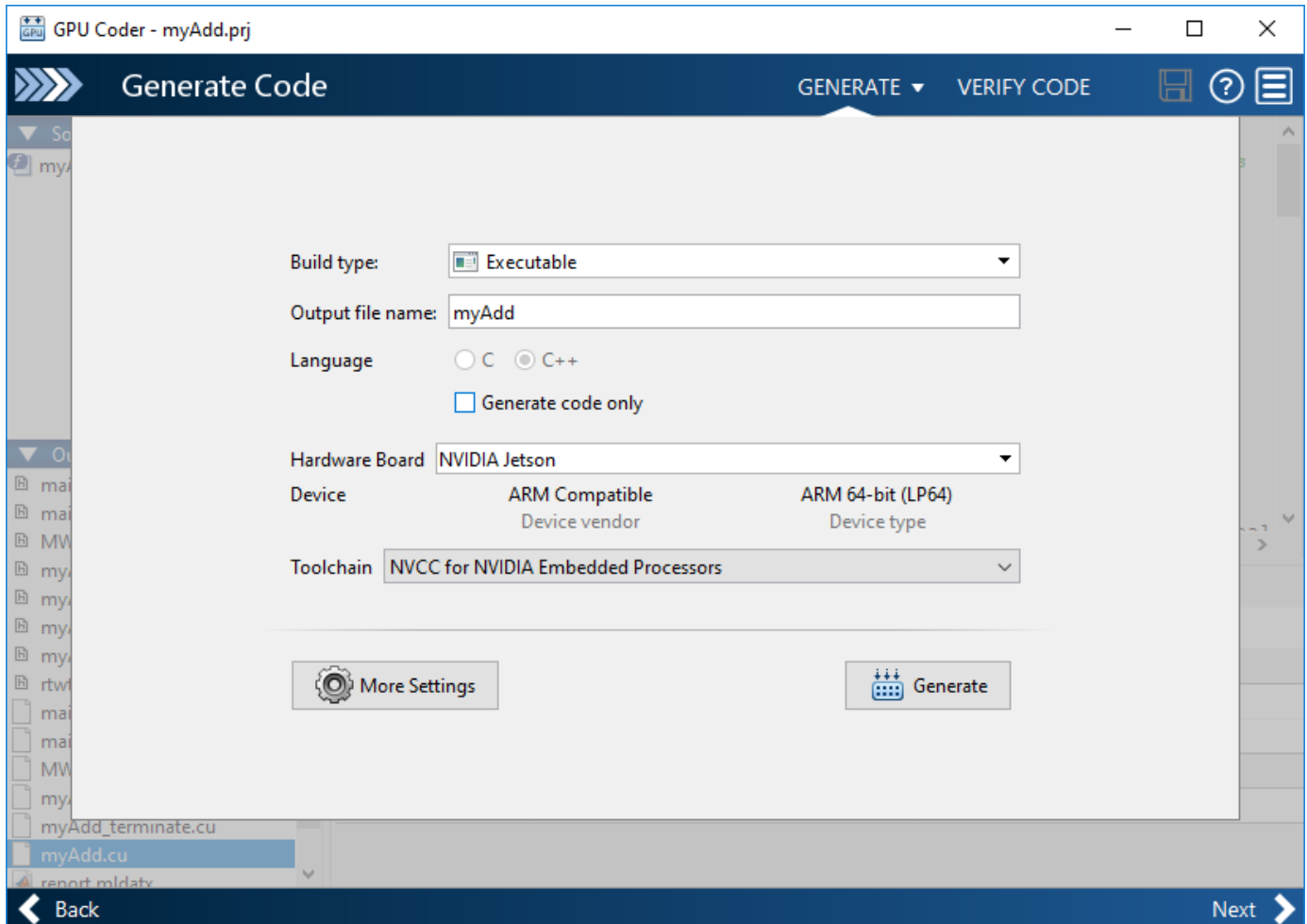
#endif

```

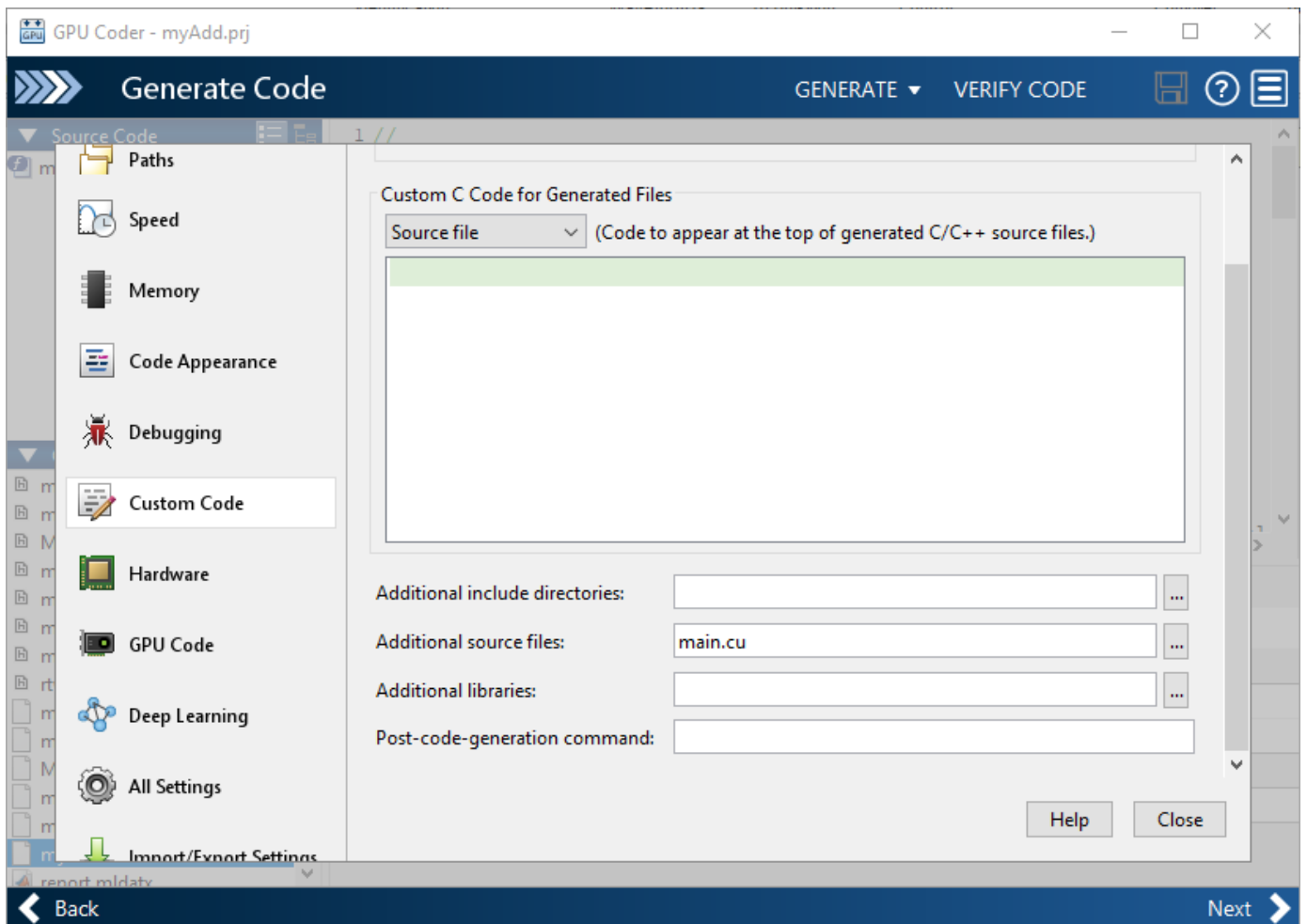
GPU Coder App

To open the GPU Coder app, on the MATLAB toolstrip **Apps** tab, under **Code Generation**, click the GPU Coder app icon. You can also open the app by typing `gpcoder` in the MATLAB Command Window.

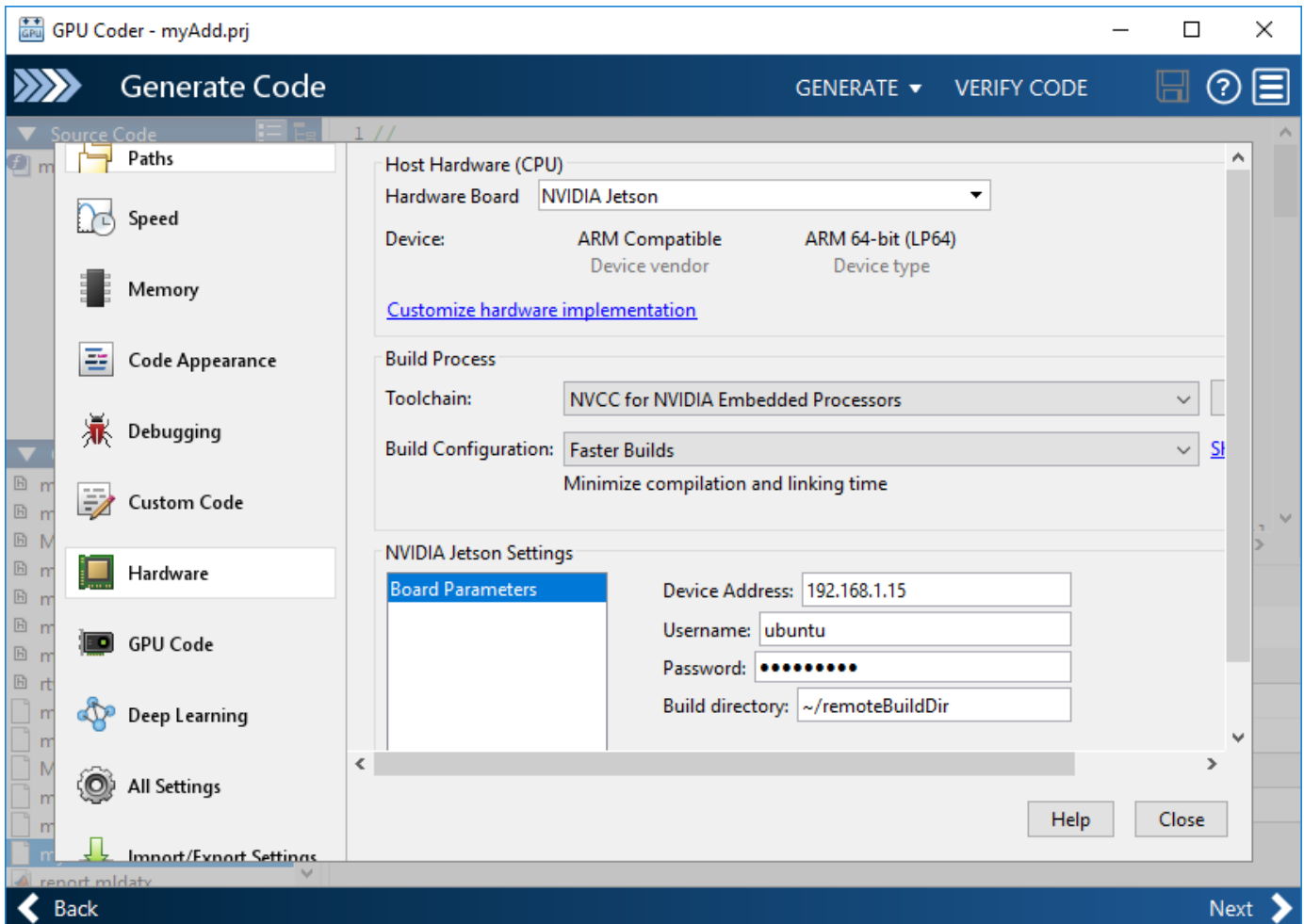
- 1 The app opens the **Select** source files page. Select `myAdd.m` as the entry-point function. Click **Next**.
- 2 In the **Define Input Types** window, enter `myAdd(1:100, 1:100)` and click **Autodefine Input Types**, then click **Next**.
- 3 You can initiate the **Check for Run-Time Issues** process or click **Next** to go to the **Generate Code** step.
- 4 Set the **Build type** to Executable and the **Hardware Board** to NVIDIA Jetson.



- 5 Click **More Settings**, on the **Custom Code** panel, enter the custom main file `main.cu` in the field for **Additional source files**. The custom main file and the header file must be in the same location as the entry-point file.



- 6 Under the **Hardware** panel, enter the device address, user name, password, and build folder for the board.



- 7 Close the **Settings** window and click **Generate**. The software generates CUDA code and deploys the executable to the folder specified. Click **Next** and close the app.

Run the Executable and Verify the Results

In the MATLAB command window, use the `runApplication()` method of the hardware object to start the executable on the target hardware.

```
hwobj = jetson;
pid = runApplication(hwobj, 'myAdd');
```

```
### Launching the executable on the target...
Executable launched successfully with process ID 26432.
Displaying the simple runtime log for the executable...
```

Copy the output bin file `myAdd.bin` to the MATLAB environment on the host and compare the computed results with the results from MATLAB.

```
outputFile = [hwobj.workspaceDir '/myAdd.bin']
getFile(hwobj, outputFile);
```

```
% Simulation result from the MATLAB.
simOut = myAdd(0:99, 0:99);
```



```
% Read the copied result binary file from target in MATLAB.  
fId = fopen('myAdd.bin','r');  
tOut = fread(fId,'double');  
diff = simOut - tOut';  
fprintf('Maximum deviation is: %f\n', max(diff(:)));
```

Maximum deviation between MATLAB Simulation output and GPU coder output on Target is: 0.000000

See Also

Objects

jetson | drive

More About

- “Build and Run an Executable on NVIDIA Hardware” on page 5-2
- “Code Generation Using the Command Line Interface”
- “Code Generation by Using the GPU Coder App”
- “Code Generation for Deep Learning Networks by Using cuDNN” on page 4-69
- “Code Generation for Deep Learning Networks by Using TensorRT” on page 4-78
- “Stop or Restart an Executable Running on NVIDIA Hardware” (MATLAB Coder Support Package for NVIDIA Jetson and NVIDIA DRIVE Platforms)
- “Run Linux Commands on NVIDIA Hardware” (MATLAB Coder Support Package for NVIDIA Jetson and NVIDIA DRIVE Platforms)

Relocate Generated Code to Another Development Environment

In this section...

“Package Generated Code Using the GPU Coder” on page 5-14

“Specify packNGo Options” on page 5-22

If you need to relocate the generated code files to another development environment, such as a system or an integrated development environment (IDE) that does not include MATLAB, you can use the `packNGo` function at the command line or the **Package** option in the GPU Coder app. The files are packaged in a compressed file that you can relocate and unpack using a standard zip utility.

Because the code generated by using GPU Coder relies on third-party compilers, libraries to build and run the executables, the development environment that you are relocating to must also satisfy these requirements. For more information, see “Installing Prerequisite Products” and “Setting Up the Prerequisite Products”.

Note GPU Coder requires that the `'minimalHeaders'` option of the `packNGo` command is set to `false`. This setting instructs the software to include all the header files found on the include path in the zip file (rather than the minimal header files required to build the code). For example, `packNGo(buildInfo, 'minimalHeaders', false)`.

Package Generated Code Using the GPU Coder

This example shows how to package generated code into a zip file for relocation using the Package option in the GPU Coder app. The example uses a Sobel edge detection application to demonstrate this concept. By default, GPU Coder creates the zip file in the current working folder.

Prerequisites

- NVIDIA® CUDA® enabled GPU
- CUDA toolkit and drivers.
- For information on the supported versions of the compilers and libraries, see “Third-Party Hardware”. For setting up the environment variables, see “Setting Up the Prerequisite Products”.

Sobel Edge Detection Entry-Point Function

In the Sobel edge detection algorithm, a 2-D spatial gradient operation on a grayscale image is performed. This operation emphasizes the high spatial frequency regions which corresponds to edges.

```
type sobelEdge.m

function [ magnitude ] = sobelEdge( Image )
%#codegen

% Copyright 2017-2021 The MathWorks, Inc.
```

```
maskX = single([-1 0 1 ; -2 0 2; -1 0 1]);
maskY = single([-1 -2 -1 ; 0 0 0 ; 1 2 1]);

coder.gpu.kernelfun();

resX = conv2(Image, maskX, 'same');
resY = conv2(Image, maskY, 'same');

magnitude = sqrt(resX.^2 + resY.^2);
thresh = magnitude < 0.4;
magnitude(thresh) = 0;

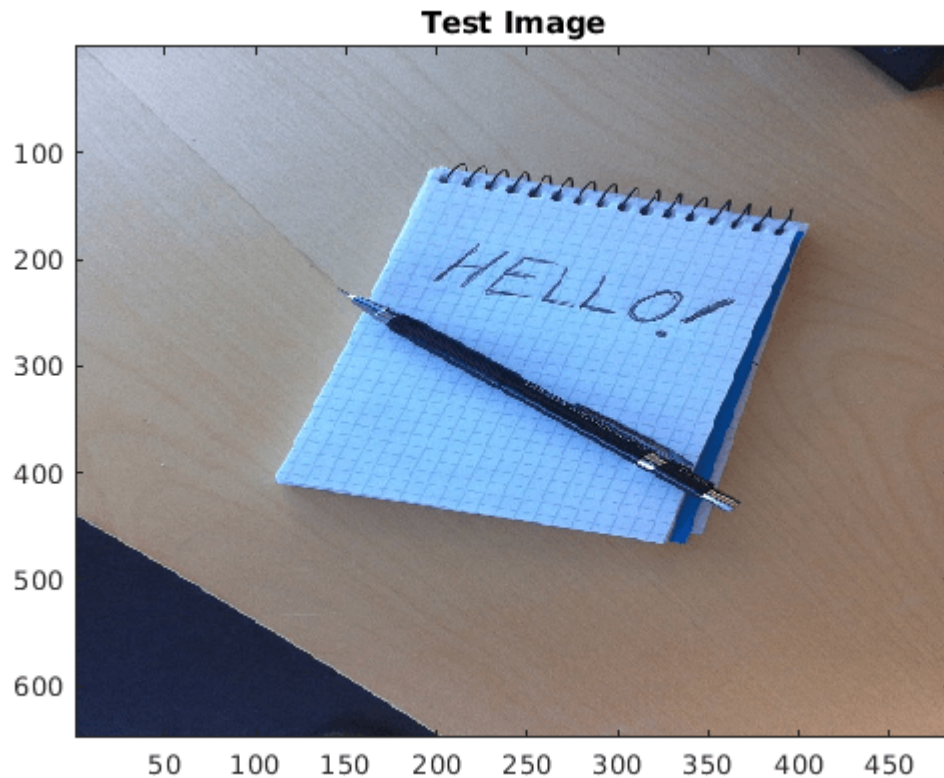
end
```

The Sobel edge algorithm computes the horizontal gradient (`resX`) and the vertical gradient (`resY`) of the input image by using two orthogonal filter kernels (`maskX` and `maskY`). After the filtering operation, the algorithm computes the gradient magnitude and applies a threshold to find the regions of the images that are considered to be edges.

Run Sobel Edge Detection Algorithm on Test Image

The Sobel filtering algorithm operates on grayscale images. Convert the color image to an equivalent grayscale image with normalized values (0.0 for black, 1.0 for white).

```
im = imread('hello.jpg');
imGray = (0.2989 * double(im(:,:,1)) + 0.5870 * double(im(:,:,2)) + ...
    0.1140 * double(im(:,:,3)))/255;
imSize = size(imGray);
figure();
image(im);
title('Test Image');
```

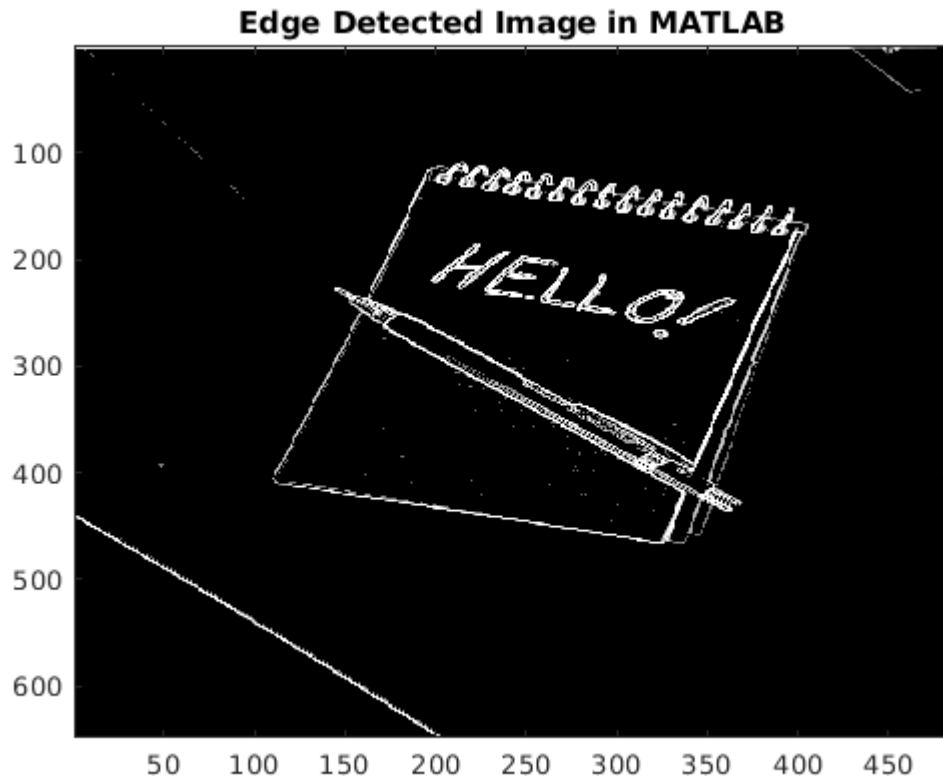


Write the matrix gray into the `inputImage.csv` file using the `writematrix` command. The Sobel edge detection application reads in this CSV file.

```
writematrix(reshape(imGray,1,[]),'inputImage.csv');  
imOut = sobelEdge(double(imGray));
```

To display the edge detected image, reformat the matrix `imOut` with the function `repmat` so that you can pass it to the `image` command.

```
figure();  
image(repmat(imOut,[1 1 3]));  
title('Edge Detected Image in MATLAB');
```



Create Custom Main Function for `sobelEdge.m`

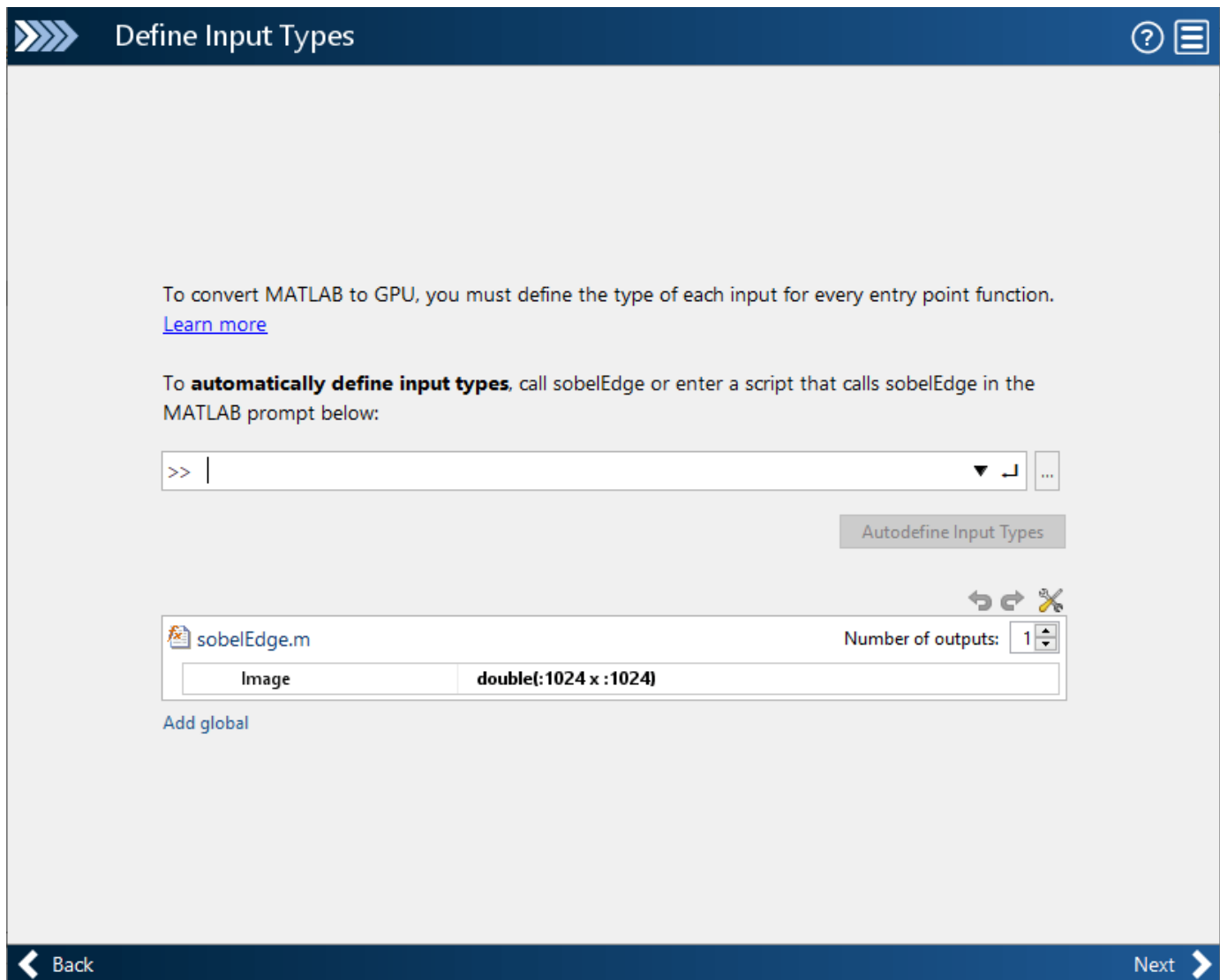
This example uses a custom main file, `main_sobel.cu` and its associated header file `main_sobel.h`. This custom main file reads the input image from the `inputImage.csv` file, calls the `sobelEdge` function in the generated `sobelEdge.cu` file, and saves the data from the edge detected image into the `outputMag.csv` file.

Package Generated Code Using the GPU Coder App

Open the GPU Coder app. On the MATLAB Toolstrip **Apps** tab, under **Code Generation**, click the GPU Coder app icon.

On the **Select Source Files** page, enter the name of the entry-point function `sobelEdge.m`. Click **Next** to go to the **Define Input Types** page.

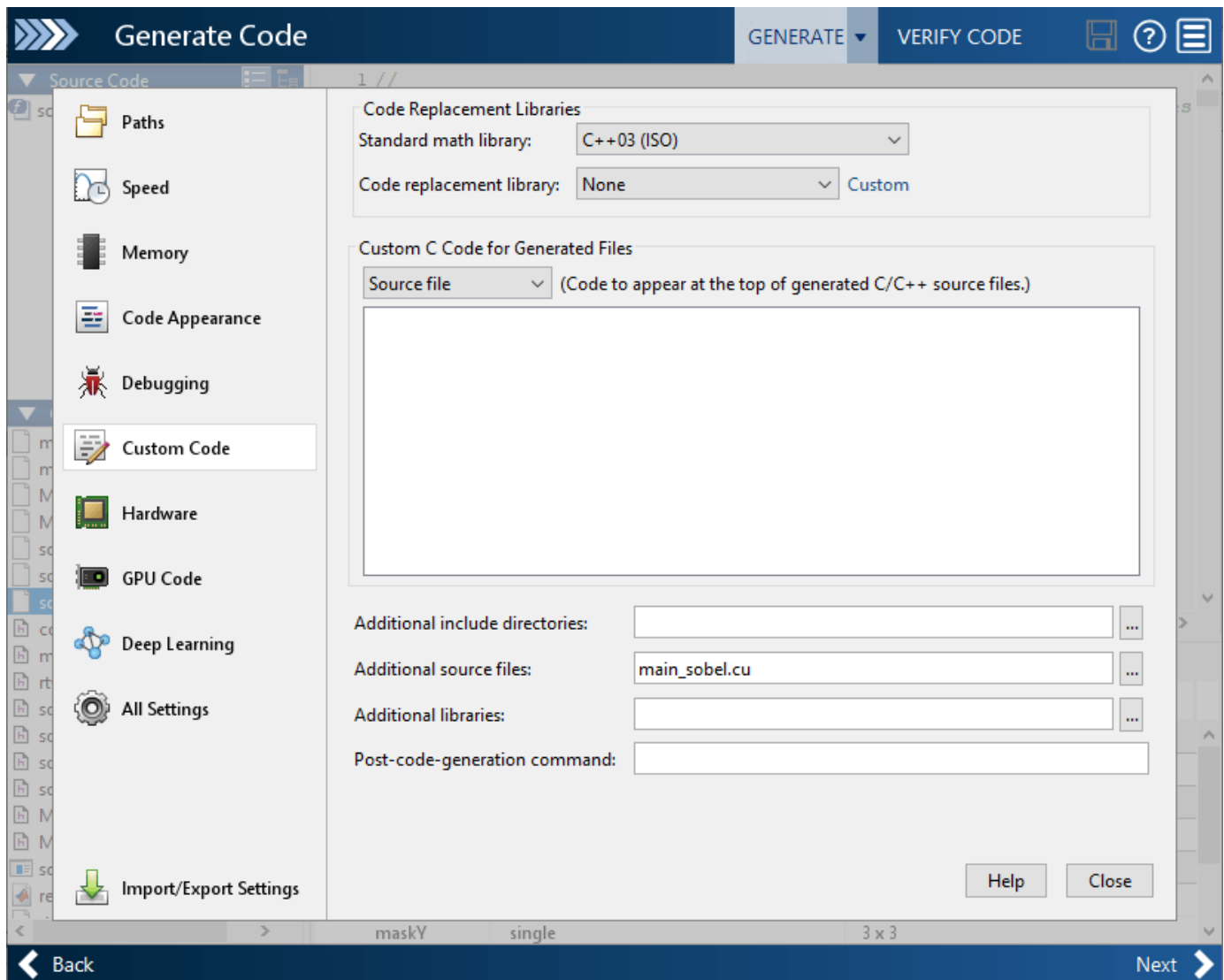
Specify that the input `Image` is of double data type and variable size with upper bound of 1024. To specify variable size with an upper bound of 1024, select `:1024`. Click **Next** to go to the **Check for Run-Time Issues** page.



Check for run-time issues. In the **Check for Run-Time** Issues dialog box, enter code that calls `sobelEdge` with double input. For example, `sobelEdge(ones(648,484))`. Click **Check for Issues**. To check for run-time issues, the app generates and runs a MEX function. The app does not find issues for `sobelEdge`. Click **Next** to go to the **Generate Code** page.

In the **Generate** dialog box, set the **Build Type** to **Executable**. You can also package the code generated for Source Code, Static Library, or Dynamic Library targets. You cannot package the code generated for MEX targets. Click **More Settings**.

On the **Custom Code** tab, under **Custom C Code for Generated Files**, set **Additional source files** to `main_sobel.cu`. Click **Close** to go to the **Generate Code** page.



Click **Generate**. Click **Next** to go to the **Finish Workflow** page. On the **Finish Workflow** page, click **Package**.

The screenshot shows a software interface with a dark blue header bar. On the left, there are three white chevrons pointing right, followed by the text 'Finish Workflow'. On the right side of the header, there is a red-bordered button labeled 'PACKAGE', a question mark icon, and a hamburger menu icon. Below the header, a large green checkmark is on the left. To its right, the text 'Executable Generated Successfully' is displayed in a large, bold font. Underneath this, a smaller line of text says 'You can now use the library in your applications. [Learn more](#)'. Below this is a section titled 'Project Summary' with a horizontal line underneath. It contains four rows of information: 'Functions' with a file icon and 'sobelEdge.m'; 'Project Type' with 'GPU Coder'; 'Numeric conversion' with 'None'; and 'Project File' with a file icon and 'sobelEdge.prj'. Below this is another section titled 'Generated Output' with a horizontal line underneath. It contains four rows: 'GPU Code' with a folder icon and 'C:\EMpath\Examples\gpcoder-ex06337729\codegen\exe\sobelEdge'; 'Binaries' with a file icon and 'C:\EMpath\Examples\gpcoder-ex06337729\sobelEdge.exe'; 'Example main Files' with a folder icon and 'C:\EMpath\Examples\gpcoder-ex06337729\codegen\exe\sobelEdge\examples'; and 'Reports' with a document icon and 'Code Generation Report'. At the bottom left of the interface, there is a white arrow pointing left and the text 'Back'.

In the **Package** dialog box, specify the package file name and packaging type. By default, the app derives the name of the package file from the project name. The app saves the file in the current working folder. By default, the app packages the generated files as a single, flat folder. For this example, use the default values, and then click **Save**.

This zip file contains the CUDA C++ code and header files required for relocation. It does not contain:

- Compile flags
- Defines
- Makefiles
- Example main files, unless you configure code generation to generate and compile the example main function.

Inspect the contents of `sobelEdge_pkg.zip` in your working folder to verify that it is ready for relocation to the destination system. Depending on the zip tool that you use, you can potentially open

and inspect the file without unpacking it. You can now relocate the resulting zip file to the desired development environment and unpack the file.

Package Generated Code at the Command Line

To generate a CUDA executable for the `sobelEdge` function, create a GPU code configuration object and run the `codegen` command.

```
cfg = coder.gpuConfig('exe');
cfg.GenerateReport = true;
cfg.CustomSource = 'main_sobel.cu';
codegen -config cfg sobelEdge -args {coder.typeof(0,[1024 1024],[1 1])}
```

Code generation successful: [View report](#)

To package the generated code into a zip file, load the `BuildInfo` object. The `BuildInfo` object contains information for compiling and linking generated code, including the list of all the source and include files and their paths.

```
buildInfoFile = fullfile(pwd,'codegen','exe','sobelEdge','buildInfo.mat');
load(buildInfoFile);
```

Create the zip file by using the `packNGo` function.

```
packNGo(buildInfo,'packType','flat','nestedZipFiles',true,...
    'minimalHeaders',false,'includeReport',false);
```

The `packNGo` function creates the `sobelEdge.zip` file in the current working folder. This zip file contains the CUDA C++ code and header files required for relocation. It does not contain:

- Compile flags
- Defines
- Makefiles
- Example main files, unless you configure code generation to generate and compile the example main function.

Inspect the contents of `sobelEdge.zip` in your working folder to verify that it is ready for relocation to the destination system. Depending on the zip tool that you use, you can potentially open and inspect the file without unpacking it. You can now relocate the resulting zip file to the desired development environment and unpack the file.

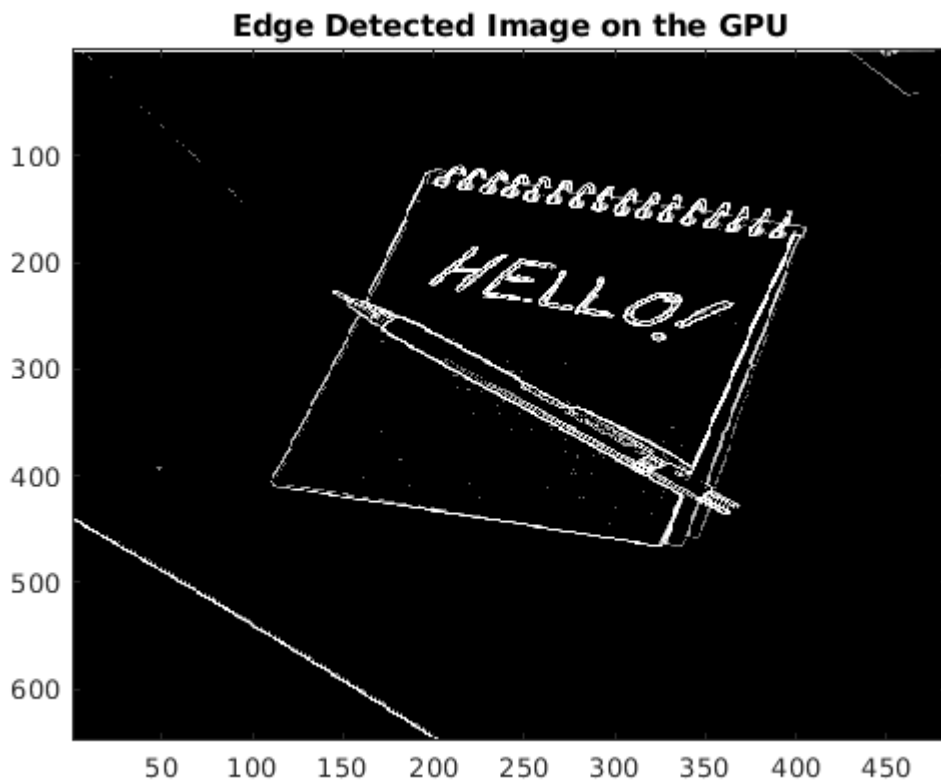
Standalone Code Execution

When you execute the generated standalone executable, the output `magnitudeData` is computed and written to a comma-separated file. Read this output back in MATLAB and use the `image` function to visualize the edge detected image.

```
if ispc
    system('sobelEdge.exe');
else
    system('./sobelEdge');
end

imOutGPU = reshape(readmatrix('outputMag.csv'),imSize);
edgeImg = repmat(imOutGPU,[1 1 3]);
figure();
```

```
image(edgeImg);
title('Edge Detected Image on the GPU');
```



Specify packNGo Options

You can specify options for the packNGo function.

To	Specify
Change the structure of the file packaging to hierarchical.	<code>packNGo(buildInfo, 'packType', 'hierarchical');</code>
Change the structure of the file packaging to hierarchical and rename the primary zip file.	<code>packNGo(buildInfo, 'packType', 'hierarchical', . .. 'fileName', 'zippedsrcs');</code>
Include all header files found on the include path in the zip file (rather than the minimal header files required to build the code).	<code>packNGo(buildInfo, 'minimalHeaders', false);</code>
For GPU Coder, this option must be set to false.	
Generate warnings for parse errors and missing files.	<code>packNGo(buildInfo, 'ignoreParseError', true, ... 'ignoreFileMissing', true);</code>

For more information, see `packNGo`.

Choose a Structure for the Zip File

Before you generate and package the files, decide whether you want to package the files in a flat or hierarchical folder structure. By default, the `packNGo` function packages the files in a single, flat folder structure. This approach is the simplest and might be the optimal choice.

If	Use
You are relocating files to an IDE that does not use the generated makefile, or the code is not dependent on the relative location of required static files	A single, flat folder structure
The target development environment must maintain the folder structure of the source environment because it uses the generated makefile, or the code depends the relative location of files	A hierarchical structure

If you use a hierarchical structure, the `packNGo` function creates two levels of zip files. There is a primary zip file, which in turn contains the following secondary zip files:

- `mlrFiles.zip` — files in your *matlabroot* folder tree
- `sDirFiles.zip` — files in and under your build folder where you initiated code generation
- `otherFiles.zip` — required files not in the *matlabroot* or *start* folder trees

Paths for the secondary zip files are relative to the root folder of the primary zip file, maintaining the source development folder structure.

See Also

Functions

`packNGo` | `codegen` | `coder.gpuConfig`

More About

- “Code Generation by Using the GPU Coder App”
- “Code Generation Using the Command Line Interface”

Getting Started with the MATLAB Coder Support Package for NVIDIA Jetson and NVIDIA DRIVE Platforms

This example shows how to use the **MATLAB® Coder™ Support Package for NVIDIA Jetson and NVIDIA DRIVE Platforms** with embedded boards from NVIDIA®. The example uses a simple vector addition algorithm to illustrate:

- Connection to the embedded board from the MATLAB environment.
- Perform basic operations such as file transfer to and from MATLAB and executing Linux® shell commands on the board.
- Generate C++ executable from a MATLAB function and run the executable on the ARM® CPU in the board.
- Generate CUDA® executable from a MATLAB function and run the executable on the NVIDIA GPU in the board.



Prerequisites

Target Board Requirements

- NVIDIA DRIVE PX2 or Jetson embedded platform.
- Ethernet crossover cable to connect the target board and host PC (if you cannot connect the target board to a local network).
- NVIDIA CUDA toolkit and libraries installed on the board.
- Environment variables on the target for the compilers and libraries. For more information, see “Install and Setup Prerequisites for NVIDIA Boards” (MATLAB Coder Support Package for NVIDIA Jetson and NVIDIA DRIVE Platforms).

Development Host Requirements

- MATLAB Coder for C++ code generation. For a tutorial, see “Get Started with MATLAB Coder”.
- GPU Coder for CUDA code generation. For a tutorial, see “Get Started with GPU Coder”.
- For CUDA code generation, NVIDIA CUDA toolkit on the host and environment variables for the compilers and libraries. For more information, see “Third-Party Hardware” and “Setting Up the Prerequisite Products”.

Create a Folder and Copy Relevant Files

The following line of code creates a folder in your current working folder on the host and copies all the relevant files into this folder. If you cannot generate files in this folder, before running this command, change your current working folder.

```
nvidiademo_setup('nvidia_gettingstarted');
```

Connect to NVIDIA Hardware

The support package uses an SSH connection over TCP/IP to execute commands while building and running the generated code on the Jetson or DRIVE platforms. Connect the target platform to the same network as the host computer or use an Ethernet crossover cable to connect the board directly to the host computer. For information on how to set up and configure your board, see NVIDIA documentation.

To communicate with the NVIDIA hardware, create a live hardware connection object by using the `drive` (MATLAB Coder Support Package for NVIDIA Jetson and NVIDIA DRIVE Platforms) or `jetson` (MATLAB Coder Support Package for NVIDIA Jetson and NVIDIA DRIVE Platforms) function. You must know the host name or IP address, user name, and password of the target board to create a live hardware connection object. For example, when connecting to the target board for the first time, create a live object for Jetson hardware by using the command:

```
hwobj = jetson('jetson-tx2-name', 'ubuntu', 'ubuntu');
```

During the hardware live object creation, the support package performs hardware and software checks, IO server installation, and gathers peripheral information on target. This information is displayed in the Command Window.

Similarly, to create live object for DRIVE hardware, use the command:

```
hwobj = drive('drive-px2-name', 'ubuntu', 'ubuntu');
```

In case of a connection failure, a diagnostics error message is reported at the MATLAB command line. If the connection has failed, the most likely cause is incorrect IP address or host name.

Run Linux Commands on NVIDIA Hardware

When a successful connection to the board is established, you can use the `system` method of the board object to execute various Linux shell commands on the NVIDIA hardware from MATLAB. For example, to list the contents of the home folder on the target board, use the command:

```
system(hwobj, 'ls -al ~');
```

The hardware object provides basic file manipulation capabilities. To transfer files from the host to the target use the `putFile()` method of the live hardware object. For example, to transfer the `test.txt` file in the current folder to the `remoteBuildDir` on the target board, use the command:

```
putFile(hwobj, 'test.txt', '~/remoteBuildDir');
```

To copy a file from the target board to the host computer, use the `getFile()` method of the hardware object. For example,

```
getFile(hwobj, '~/remoteBuildDir/test.txt', '.');
```

Generate C++ code for the ARM CPU Using MATLAB Coder

This example uses `myAdd.m`, a simple vector addition, as the entry-point function for code generation.

```
function out = myAdd(inp1,inp2) %#codegen
% Simple vector addition
% Copyright 2018-2021 The MathWorks, Inc.
out = inp1 + inp2;
end
```

To generate an executable that you can deploy on to an NVIDIA target, create a code configuration object for generating an executable.

```
cfg = coder.config('exe');
cfg.TargetLang = 'C++';
```

When there are multiple live connection objects for different targets, the code generator performs a remote build on the target board for which a recent live object was created. To choose a hardware board for performing a remote build, use the `setupCodegenContext()` method of the respective live hardware object. If only one live connection object was created, you do not need to call this method.

```
hwobj.setupCodegenContext;
```

To create a configuration object for the DRIVE or Jetson platform and assign it to the `Hardware` property of the code configuration object `cfg`, use the `coder.hardware` function. Use `'NVIDIA Jetson'` for the Jetson boards and `'NVIDIA Drive'` for the DRIVE board.

```
cfg.Hardware = coder.hardware('NVIDIA Jetson');
```

To specify the folder for performing remote build process on the target board, use the `BuildDir` property. If the specified build folder does not exist on the target board, then the software creates a folder with the given name. If no value is assigned to `cfg.Hardware.BuildDir`, the remote build process occurs in the last specified build folder. If there is no stored build folder value, the build process takes place in the home folder.

```
cfg.Hardware.BuildDir = '~/remoteBuildDir';
```

The custom `main.cpp` file is a wrapper that calls the entry point function in the generated code. This main file passes a vector containing the first 100 natural numbers to the entry-point function. The main file writes the results to the `myAdd.bin` binary file.

```
cfg.CustomSource = fullfile('main.cpp');
```

To generate C++ code, use the `codegen` function and pass the code configuration and the size of the inputs for and `myAdd.m` entry-point function. After the code generation takes place on the host, the generated files are copied over and built on the target board.

```
codegen('-config ',cfg,'myAdd','-args',{1:100,1:100});
```

Generate CUDA Code for the Target Board Using GPU Coder

Verify GPU Environment on Target Board

To verify that the compilers and libraries necessary for running this example are set up correctly, use the `coder.checkGpuInstall` function.

```
% Use 'drive' for NVIDIA DRIVE hardware
envCfg = coder.gpuEnvConfig('jetson');
envCfg.BasicCodegen = 1;
envCfg.Quiet = 1;
envCfg.HardwareObject = hwobj;
coder.checkGpuInstall(envCfg);
```

Generate CUDA Executable

To generate a CUDA executable that you can deploy on to an NVIDIA target, create a GPU code configuration object for generating an executable.

```
cfg = coder.gpuConfig('exe');
cfg.Hardware = coder.hardware('NVIDIA Jetson');
cfg.Hardware.BuildDir = '~/remoteBuildDir';
cfg.CustomSource = fullfile('main.cu');
```

Certain NVIDIA platforms such as DRIVE PX2 contain multiple GPUs. On such platforms, use the `SelectCudaDevice` property in the GPU configuration object to select a specific GPU.

```
cfg.GpuConfig.SelectCudaDevice = 0;

codegen('-config ',cfg,'myAdd','-args',{1:100,1:100});
```

Run Executable on Target Board

To run the executable on the target hardware, use the `runApplication()` method of the hardware object.

```
pid = runApplication(hwobj,'myAdd');
```

Alternatively, to run the executable, use the `runExecutable()` method of the hardware object.

```
exe = [hwobj.workspaceDir '/myAdd.elf'];
pid = runExecutable(hwobj,exe);
```

Verify Result from Target Board

Copy the output bin file `myAdd.bin` to the MATLAB environment on the host and compare the computed results to those from MATLAB. The property `workspaceDir` contains the path to the codegen folder on the target board.

```
pause(0.3); % To ensure that the executable completed the execution.
getFile(hwobj,[hwobj.workspaceDir '/myAdd.bin']);
```

Simulation result from the MATLAB:

```
simOut = myAdd(0:99,0:99);
```

Read the copied result binary file from target in MATLAB:

```
fId = fopen('myAdd.bin','r'); tOut = fread(fId,'double');
```

Find the difference between the MATLAB simulation output and the output from target board.

```
diff = simOut - tOut';
```

Display the maximum deviation between the simulation output and the output from target board.

```
fprintf('Maximum deviation between MATLAB Simulation output and the output on Target is: %f\n', r
```

Cleanup

To remove the example files and return to the original folder, call the `cleanup` function.

```
cleanup
```

See Also

Objects

`jetson` | `drive`

More About

- “Build and Run an Executable on NVIDIA Hardware” on page 5-2
- “Build and Run an Executable on NVIDIA Hardware Using GPU Coder App” on page 5-7
- “Stop or Restart an Executable Running on NVIDIA Hardware” (MATLAB Coder Support Package for NVIDIA Jetson and NVIDIA DRIVE Platforms)
- “Run Linux Commands on NVIDIA Hardware” (MATLAB Coder Support Package for NVIDIA Jetson and NVIDIA DRIVE Platforms)

Sobel Edge Detection on NVIDIA Jetson Nano Using Raspberry Pi Camera Module V2

This example shows you how to capture and process images from a Raspberry Pi Camera Module V2 connected to the NVIDIA® Jetson Nano. The MATLAB® Coder™ Support Package for NVIDIA Jetson and NVIDIA DRIVE Platforms allows you to capture images from the Camera Module V2 and bring them into the MATLAB environment for processing. In this example you learn how to develop a Sobel edge detection algorithm by using this capability.

Prerequisites

Target Board Requirements

- NVIDIA Jetson Nano embedded platform.
- Raspberry Pi Camera Module V2 connected to the CSI host port of the target.
- Ethernet crossover cable to connect the target board and host PC (if you cannot connect the target board to a local network).
- NVIDIA CUDA toolkit installed on the board.
- V4L2 and SDL (v1.2) libraries on the board.
- GStreamer libraries on the board.
- Environment variables on the target for the compilers and libraries. For more information, see “Install and Setup Prerequisites for NVIDIA Boards” (MATLAB Coder Support Package for NVIDIA Jetson and NVIDIA DRIVE Platforms).

Development Host Requirements

- GPU Coder for CUDA code generation. For a tutorial, see “Get Started with GPU Coder”.
- NVIDIA CUDA toolkit on the host.
- Environment variables for the compilers and libraries. For more information, see “Third-Party Hardware” and “Setting Up the Prerequisite Products”.

Create a Folder and Copy Relevant Files

The following line of code creates a folder in your current working folder on the host and copies all the relevant files into this folder. If you cannot generate files in this folder, before running this command, change your current working folder.

```
nvidiademo_setup('sobel_edge_detection');
```

Connect to NVIDIA Jetson Nano

The support package uses an SSH connection over TCP/IP to execute commands while building and running the generated CUDA code on the Jetson Nano platforms. Connect the target platform to the same network as the host computer or use an Ethernet crossover cable to connect the board directly to the host computer. For information on how to set up and configure your board, see NVIDIA documentation.

To communicate with the NVIDIA hardware, create a live hardware connection object by using the `jetson` (MATLAB Coder Support Package for NVIDIA Jetson and NVIDIA DRIVE Platforms) function. You must know the host name or IP address, user name, and password of the target board to create a

live hardware connection object. For example, when connecting to the target board for the first time, create a live object for Jetson hardware by using the command:

```
hwobj = jetson('jetson-nano-name', 'ubuntu', 'ubuntu');
```

During the hardware live object creation, the support package performs hardware and software checks, IO server installation, and gathers peripheral information on target. This information is displayed in the Command Window.

Run the `getCameraList` function of the `hwobj` object to find the available cameras. If this function outputs an empty table, then try re-connecting the camera and execute the function again.

```
camlist = getCameraList(hwobj);
```

Verify GPU Environment on Target Board

To verify that the compilers and libraries necessary for running this example are set up correctly, use the `coder.checkGpuInstall` function.

```
envCfg = coder.gpuEnvConfig('jetson');  
envCfg.BasicCodegen = 1;  
envCfg.Quiet = 1;  
envCfg.HardwareObject = hwobj;  
coder.checkGpuInstall(envCfg);
```

Create a Camera Object

Create a camera object by using the name from the `getCameraList` function. For example, if the camera is named `vi-output`, `imx219 6-0010`, use:

```
camObj = camera(hwobj, "vi-output, imx219 6-0010", [640 480]);
```

`camObj` is a handle to a camera object. To display the images captured from the Camera Module V2 in MATLAB, use these commands:

```
for i = 1:100  
    img = snapshot(camObj);  
    imagesc(img);  
    drawnow;  
end
```

This camera object captures RGB and 3-channel grayscale images.

Create a Display Object

To create a display object, use the `imageDisplay` function. This object is a system object that uses `imshow` function to display the images in MATLAB.

```
dispObj = imageDisplay(hwobj);  
img = snapshot(camObj);  
image(dispObj, img);
```

Sobel Edge Detection Algorithm

The Sobel edge detection algorithm is a 2-D spatial gradient operation on a grayscale image. This operation emphasizes the high spatial frequency regions of the image that corresponds to edges.

Calculate Gradients

Find horizontal gradient(h) and vertical gradient (v) of the input image with respective Sobel kernels. These two Sobel kernels are orthogonal to each other. Before processing live image data from the camera, test the algorithm on a sample image.

```
kern = [1 2 1; 0 0 0; -1 -2 -1];
img = imread('peppers.png');

imagesc(img);
```



```
h = conv2(img(:,:,2),kern,'same');
v = conv2(img(:,:,2),kern','same');
```

Calculate Gradient Magnitude

Find the gradient magnitude from the horizontal and vertical gradients (h and v).

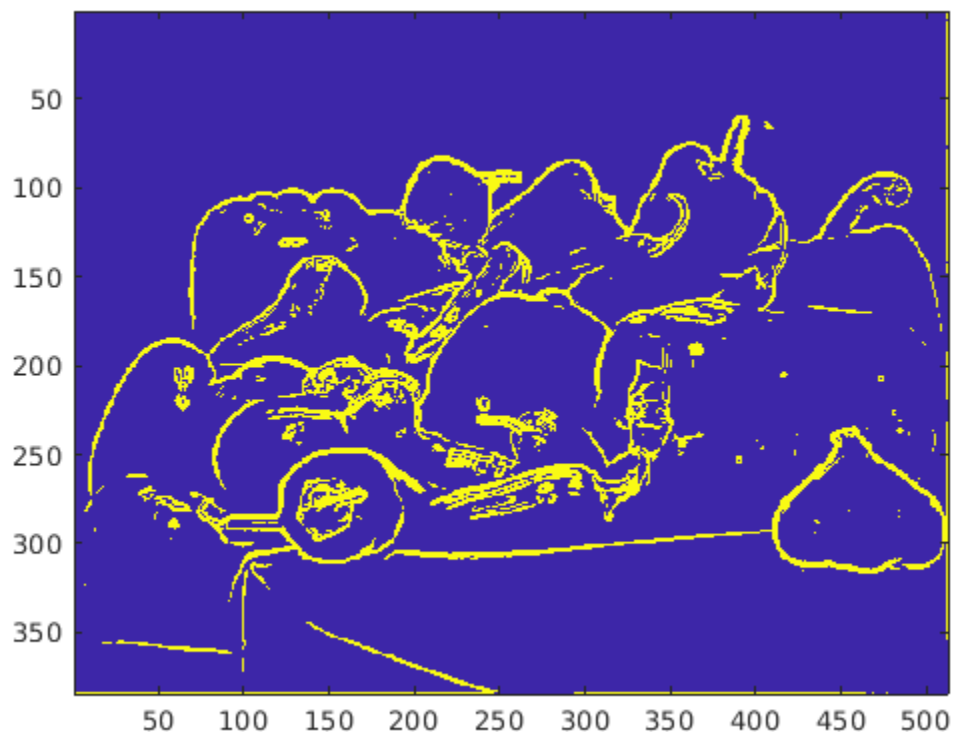
```
e = sqrt(h.*h + v.*v);
```

Threshold the Edge Image

Threshold the image to find the regions of image that are edges.

```
edgeImg = uint8((e > 100) * 240);

imagesc(edgeImg);
```



Run Sobel Edge Detection Algorithm on Live Data

Create a MATLAB entry-point function, `sobelEdgeDetectionAlg.m`, out of the MATLAB code developed in the previous sections of this example. View the code in MATLAB editor.

```
edit('sobelEdgeDetectionAlg.m');
```

The function `sobelEdgeDetectionAlg` takes image and threshold input for edge detection and returns the results of edge detection algorithm. Call this function on the images captured from inside a loop. You can vary the threshold variable `thresh` to get a proper edge image. This way you can use the camera access capability of the support package to tune the algorithm suitable for the specified camera.

```
for i = 1:200
    img = snapshot(camObj);
    thresh = 100;
    edgeImage = sobelEdgeDetectionAlg(img, thresh);
    image dispObj, edgeImage);
end
```

To deploy this example as a standalone application on the target board, see “Deploy and Run Sobel Edge Detection with I/O on NVIDIA Jetson Nano” (MATLAB Coder Support Package for NVIDIA Jetson and NVIDIA DRIVE Platforms).

Cleanup

To remove the example files and return to the original folder, call the `cleanup` function.

cleanup

See Also

Objects

jetson | drive

More About

- “Build and Run an Executable on NVIDIA Hardware” on page 5-2
- “Build and Run an Executable on NVIDIA Hardware Using GPU Coder App” on page 5-7
- “Stop or Restart an Executable Running on NVIDIA Hardware” (MATLAB Coder Support Package for NVIDIA Jetson and NVIDIA DRIVE Platforms)
- “Run Linux Commands on NVIDIA Hardware” (MATLAB Coder Support Package for NVIDIA Jetson and NVIDIA DRIVE Platforms)

Semantic Segmentation on NVIDIA DRIVE

This example shows how to generate and deploy a CUDA® executable for an image segmentation application that uses deep learning. It uses the MATLAB® Coder™ Support Package for NVIDIA Jetson and NVIDIA DRIVE Platforms to deploy the executable on the NVIDIA DRIVE™ platform. This example performs code generation on the host computer and builds the generated code on the target platform by using remote build capability of the support package. For more information, see “Code Generation for Semantic Segmentation Network” on page 4-152.

Prerequisites

Target Board Requirements

- NVIDIA DRIVE PX2 embedded platform.
- Ethernet crossover cable to connect the target board and host PC (if you cannot connect the target board to a local network).
- NVIDIA CUDA toolkit installed on the board.
- NVIDIA cuDNN library (v5 and above) on the target.
- OpenCV library on the target for reading and displaying images.
- Environment variables on the target for the compilers and libraries. For more information, see “Install and Setup Prerequisites for NVIDIA Boards” (MATLAB Coder Support Package for NVIDIA Jetson and NVIDIA DRIVE Platforms).

Development Host Requirements

- GPU Coder for CUDA code generation. For a tutorial, see “Get Started with GPU Coder”.
- Deep Learning Toolbox™ to use a DAG network object.
- GPU Coder Interface for Deep Learning Libraries support package. To install this support package, use the MATLAB® Add-On Explorer.
- NVIDIA CUDA toolkit on the host.
- Environment variables for the compilers and libraries. For more information, see “Third-Party Hardware” and “Setting Up the Prerequisite Products”.

Create a Folder and Copy Relevant Files

The following line of code creates a folder in your current working folder on the host and copies all the relevant files into this folder. If you cannot generate files in this folder, before running this command, change your current working folder.

```
nvidiademo_setup('segnet_deploy');
```

Connect to the NVIDIA Hardware

The support package uses an SSH connection over TCP/IP to execute commands while building and running the generated CUDA code on the DRIVE platforms. Connect the target platform to the same network as the host computer or use an Ethernet crossover cable to connect the board directly to the host computer. For information on how to set up and configure your board, see NVIDIA documentation.

To communicate with the NVIDIA hardware, create a live hardware connection object by using the `drive` (MATLAB Coder Support Package for NVIDIA Jetson and NVIDIA DRIVE Platforms) function.

You must know the host name or IP address, user name, and password of the target board to create a live hardware connection object. For example, when connecting to the target board for the first time, create a live object for Drive hardware by using the command:

```
hwobj = drive('drive-px2-name', 'ubuntu', 'ubuntu');
```

During the hardware live object creation, the support package performs hardware and software checks, IO server installation, and gathers peripheral information on target. This information is displayed in the Command Window.

In case of a connection failure, a diagnostics error message is reported at the MATLAB command line. If the connection has failed, the most likely cause is incorrect IP address or host name.

Verify GPU Environment on Target Board

To verify that the compilers and libraries necessary for running this example are set up correctly, use the `coder.checkGpuInstall` function.

```
envCfg = coder.gpuEnvConfig('drive');
envCfg.BasicCodegen = 1;
envCfg.Quiet = 1;
envCfg.HardwareObject = hwobj;
coder.checkGpuInstall(envCfg);
```

Get Pretrained SegNet DAG Network Object

```
net = getSegNet();
```

Downloading pre-trained SegNet (107 MB)...

The DAG network contains 91 layers including convolution, batch normalization, pooling, unpooling, and the pixel classification output layers. To see all the layers of the network, use the `analyzeNetwork` function.

Generate CUDA Code for the Target Board Using GPU Coder

This example uses `segnet_predict.m` file as the entry-point function for code generation. To generate a CUDA executable that you can deploy on to an NVIDIA target, create a GPU code configuration object for generating an executable.

```
cfg = coder.gpuConfig('exe');
```

When there are multiple live connection objects for different targets, the code generator performs a remote build on the target board for which a recent live object was created. To choose a hardware board for performing a remote build, use the `setupCodegenContext()` method of the respective live hardware object. If only one live connection object was created, you do not need to call this method.

```
hwobj.setupCodegenContext;
```

To create a configuration object for the DRIVE platform and assign it to the `Hardware` property of the code configuration object `cfg`, use the `coder.hardware` function.

```
cfg.Hardware = coder.hardware('NVIDIA Drive');
```

To specify the folder for performing remote build process on the target board, use the `BuildDir` property. If the specified build folder does not exist on the target board, then the software creates a

folder with the given name. If no value is assigned to `cfg.Hardware.BuildDir`, the remote build process occurs in the last specified build folder. If there is no stored build folder value, the build process takes place in the home folder.

```
cfg.Hardware.BuildDir = '~/remoteBuildDir';
```

On NVIDIA platforms such as DRIVE PX2 that contain multiple GPUs, use the `SelectCudaDevice` property in the GPU configuration object to select a specific GPU.

```
cfg.GpuConfig.SelectCudaDevice = 0;
```

The custom `main.cu` file is a wrapper that calls the `predict` function in the generated code. Postprocessing steps are added in the main file by using OpenCV interfaces. The output of SegNet prediction is an 11-channel image. The eleven channels here represent the prediction scores of eleven different classes. In postprocessing, each pixel is assigned a class label that has the maximum score among the 11 channels. Each class is associated with a unique color for visualization. The final output is shown by using the OpenCV `imshow` function.

```
cfg.CustomSource = fullfile('main.cu');
```

In this example, code generation uses an image as the input to the network. However, the custom main file is coded to take video as input and perform a SegNet prediction for each frame in the video sequence. The compiler and linker flags required to build the executable with OpenCV library are updated in the `buildinfo` section in the `|segnet_predict.m|file`.

Generate sample image input for code generation.

```
img = imread('peppers.png');  
img = imresize(img,[360 480]);
```

To generate CUDA code, use the `codegen` function and pass the GPU code configuration and the size of the inputs for and `segnet_predict.m` entry-point function. After the code generation takes place on the host, the generated files are copied over and built on the target board.

```
codegen('-config ', cfg, 'segnet_predict', '-args', {img}, '-report');
```

Run Executable on Target Board

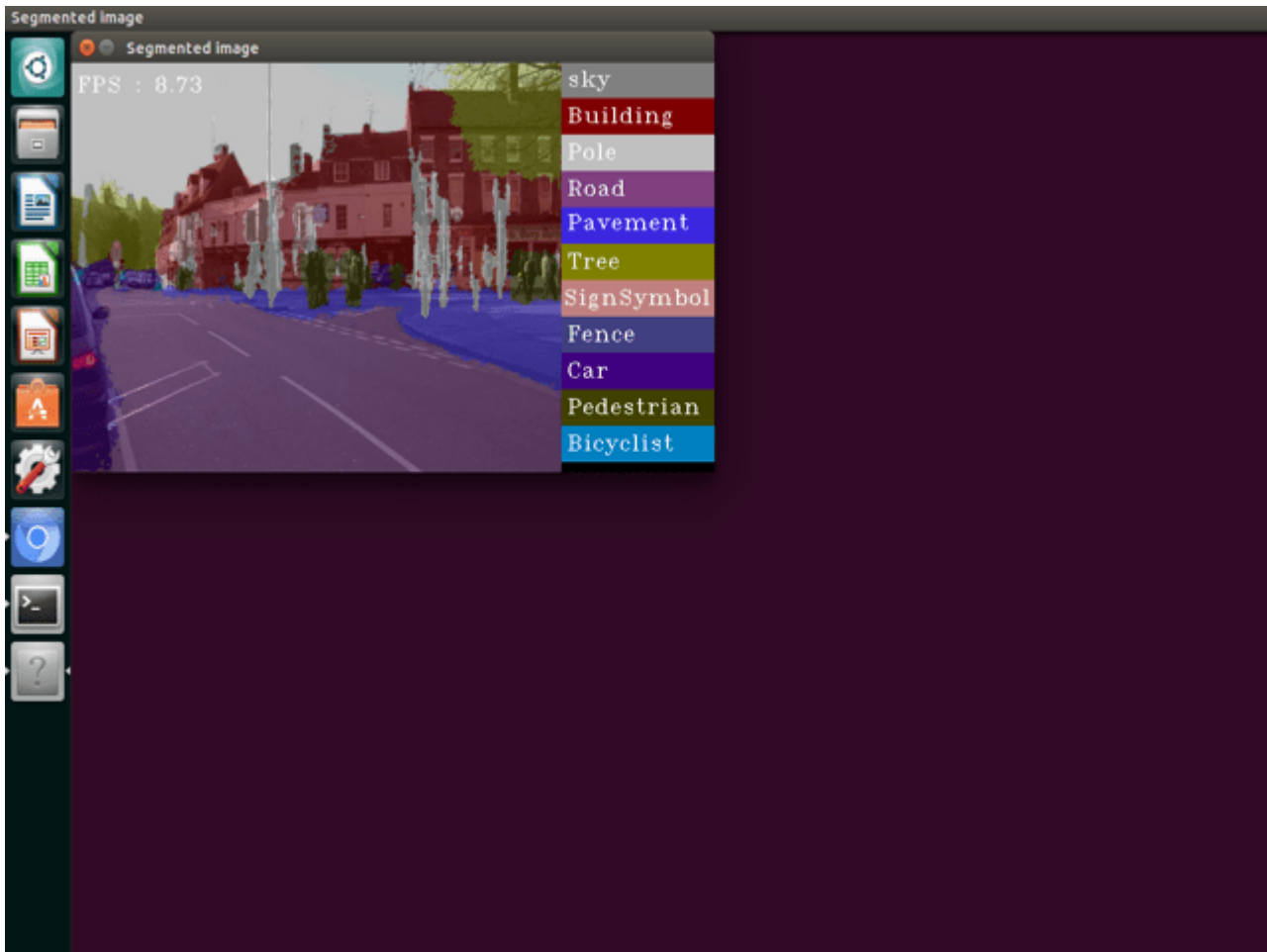
Copy the input test video to the target workspace folder, using the `workspaceDir` property of the hardware object. This property contains the path to the `codegen` folder on the target board.

```
hwobj.putFile('CamVid.avi', hwobj.workspaceDir);
```

To launch the executable on the target hardware, use the `runApplication()` method of the hardware object.

```
hwobj.runApplication('segnet_predict','CamVid.avi');
```

The segmented image output is displayed in a window on the monitor that is connected to the target board.



You can stop the running executable on the target board from the MATLAB environment on the host by using the `killApplication()` method of the hardware object. This method uses the name of the application and not the name of the executable.

```
hwobj.killApplication('segnet_predict');
```

Cleanup

To remove the example files and return to the original folder, call the `cleanup` function.

```
cleanup
```

See Also

Objects

jetson | drive

More About

- “Build and Run an Executable on NVIDIA Hardware” on page 5-2
- “Build and Run an Executable on NVIDIA Hardware Using GPU Coder App” on page 5-7

- “Stop or Restart an Executable Running on NVIDIA Hardware” (MATLAB Coder Support Package for NVIDIA Jetson and NVIDIA DRIVE Platforms)
- “Run Linux Commands on NVIDIA Hardware” (MATLAB Coder Support Package for NVIDIA Jetson and NVIDIA DRIVE Platforms)

Top-Hat Filtering to Remove Uneven Background Illumination on NVIDIA Jetson TX2 Developer Kit

This example shows how to deploy Image Processing Toolbox™ algorithms to NVIDIA® Jetson TX2 board using the GPU Coder™ Support Package for NVIDIA GPUs. The `imtophat` (Image Processing Toolbox) function that performs morphological top-hat filtering on a grayscale image is used as an example to demonstrate this concept. Top-hat filtering computes the morphological opening of the image (using `imopen` (Image Processing Toolbox)) and then subtracts the result from the original image. The generated CUDA® code uses shared memory to speed up the operations on the GPU.

Prerequisites

Target Board Requirements *

- NVIDIA Jetson TX2 embedded platform.
- Ethernet crossover cable to connect the target board and host PC (if the target board cannot be connected to a local network).
- NVIDIA CUDA toolkit installed on the board.
- OpenCV library on the target for reading and displaying images and video.
- Environment variables on the target for the compilers and libraries. For information on the supported versions of the compilers and libraries and their setup, see “Install and Setup Prerequisites for NVIDIA Boards” (MATLAB Coder Support Package for NVIDIA Jetson and NVIDIA DRIVE Platforms) for NVIDIA boards.

Development Host Requirements

- CUDA enabled NVIDIA GPU.
- NVIDIA CUDA toolkit and driver.
- Environment variables for the compilers and libraries. For information on the supported versions of the compilers and libraries, see “Third-Party Hardware”. For setting up the environment variables, see “Setting Up the Prerequisite Products”.

Verify NVIDIA Support Package Installation on Host

Use the `checkHardwareSupportPackageInstall` function to verify that the host system is compatible to run this example.

```
checkHardwareSupportPackageInstall();
```

Connect to the NVIDIA Hardware

The GPU Coder Support Package for NVIDIA GPUs uses an SSH connection over TCP/IP to execute commands while building and running the generated CUDA code on the Jetson platform. You must therefore connect the target platform to the same network as the host computer or use an Ethernet crossover cable to connect the board directly to the host computer. Refer to the NVIDIA documentation on how to set up and configure your board.

To communicate with the NVIDIA hardware, you must create a live hardware connection object by using the `jetson` (MATLAB Coder Support Package for NVIDIA Jetson and NVIDIA DRIVE Platforms) function. You must know the host name or IP address, username, and password of the target board to create a live hardware connection object.

```
hwobj = jetson('host-name', 'username', 'password');
```

When there are multiple live connection objects for different targets, the code generator performs remote build on the target for which a recent live object was created. To choose a hardware board for performing remote build, use the `setupCodegenContext()` method of the respective live hardware object. If only one live connection object was created, it is not necessary to call this method.

```
hwobj.setupCodegenContext;
```

Verify GPU Environment

To verify that the compilers and libraries necessary for running this example are set up correctly, use the `coder.checkGpuInstall` function.

```
envCfg = coder.gpuEnvConfig('jetson');  
envCfg.BasicCodegen = 1;  
envCfg.Quiet = 1;  
envCfg.HardwareObject = hwobj;  
coder.checkGpuInstall(envCfg);
```

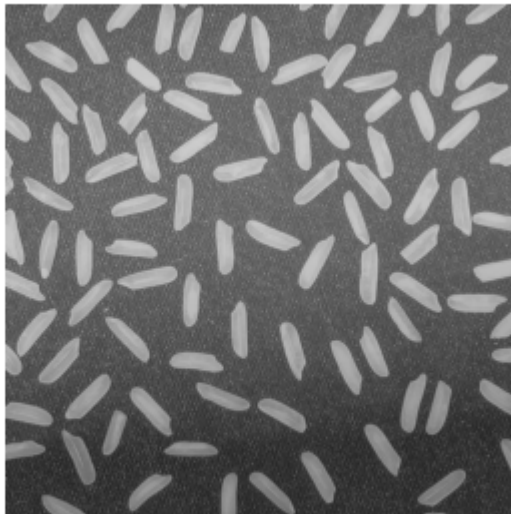
The `imtophat` Entry-Point Function

The `imtophatDemo_gpu.m` calls `imtophat` internally. The `imtophat` function performs morphological opening on the image using the `imopen` (Image Processing Toolbox) function. The result of the image is subtracted from the original image. The `imopen` operation is basically `imerode` (Image Processing Toolbox) operation followed by `imdilate` (Image Processing Toolbox).

This example is shown on an input grayscale image.

```
original = imread('rice.png');  
imshow(original), title('Input to Top-Hat Filtering');
```

Input to Top-Hat Filtering



Create a disc-shaped structuring element with a radius of 12. Neighbourhood, Nhood of this structuring element is passed as an input argument for the `imtophat` function.

```
se = strel('disk',12);
Nhood = se.Neighborhood;
type imtophatDemo_gpu

function [out] = imtophatDemo_gpu(img,Nhood,ocvFlag) %#codegen

% Copyright 2019-2021 The MathWorks, Inc.

coder.gpu.kernelfun;

% This example uses OpenCV for reading an image
% and displaying output image. Update buildinfo to link with
% OpenCV library available on target.
if ocvFlag
    % OpenCV 4 flags
    opencv_compile_flags = `pkg-config --cflags --libs opencv4`;
    opencv_link_flags = `pkg-config --libs opencv4`;
else
    % OpenCV 3 flags
    opencv_compile_flags = `pkg-config --cflags --libs opencv`;
    opencv_link_flags = `pkg-config --libs opencv`;
end

coder.updateBuildInfo('addLinkFlags',opencv_link_flags);
coder.updateBuildInfo('addCompileFlags',opencv_compile_flags);

out = imtophat(img,Nhood);

end
```

Get OpenCV Version on the Target

Use the `pkg-config` helper tool to query if OpenCV 4.x is installed on the target board. This example uses the information to update build information to link with the appropriate OpenCV library available on target.

```
try
    OpenCVver = strtrim(system(hwobj,'pkg-config --modversion opencv4'));
    isOpenCV4 = 1;
catch
    OpenCVver = strtrim(system(hwobj,'pkg-config --modversion opencv'));
    isOpenCV4 = 0;
end
```

Generate and Deploy CUDA Code on the Target

This example uses `imtophatDemo_gpu.m` as the entry-point function for code generation. To generate a CUDA executable, create a GPU code configuration object.

```
cfg = coder.gpuConfig('exe');
```

Use the `coder.hardware` function to create a configuration object for the Jetson platform and assign it to the `Hardware` property of the GPU code configuration object `cfg`.

```
cfg.Hardware = coder.hardware('NVIDIA Jetson');
```

The custom `main_tophat.cu` file is a wrapper that calls the `imtophatDemo_gpu` entry-point function in the generated code. Post processing steps are added in the main file using OpenCV interfaces. Build Flags for OpenCV libraries are included in `imtophatDemo_gpu.m` entry-point function.

```
cfg.CustomSource = fullfile('main_tophat.cu');
```

To generate CUDA code, use the `codegen` function and pass the GPU code configuration object along with input arguments. In this step, CUDA code is generated on the host, generated files are copied over and built on the target in the workspace directory. The workspace directory is available as a property, `workspaceDir` in the hardware object, `hwobj`.

```
codegen -args {original,coder.Constant(Nhood),coder.Constant(isOpenCV4)} -config cfg imtophatDemo_gpu
```

Run the Application on the Target

This application takes a grayscale image as input. Copy the `rice.png` file from host to the target device by using the `putFile` command.

```
imgLoc = which('rice.png');  
hwobj.putFile(imgLoc, hwobj.workspaceDir);
```

Use the `runApplication` method of the hardware object to launch the application on the target hardware.

```
hwobj.runApplication('imtophatDemo_gpu', 'rice.png');
```

Top-Hat Filtered Image on Jetson TX2



Kill the Application

Use the `killApplication` method of the hardware object to kill the running application on the target.

```
hwobj.killApplication('imtophatDemo_gpu');
```

See Also

Objects

`jetson | drive`

More About

- “Build and Run an Executable on NVIDIA Hardware” on page 5-2
- “Build and Run an Executable on NVIDIA Hardware Using GPU Coder App” on page 5-7
- “Stop or Restart an Executable Running on NVIDIA Hardware” (MATLAB Coder Support Package for NVIDIA Jetson and NVIDIA DRIVE Platforms)
- “Run Linux Commands on NVIDIA Hardware” (MATLAB Coder Support Package for NVIDIA Jetson and NVIDIA DRIVE Platforms)

Deployment and Classification of Webcam Images on NVIDIA Jetson TX2 Platform

This example shows how to generate CUDA® code from a DAGNetwork object and deploy the generated code onto the NVIDIA® Jetson® TX2 board using the GPU Coder™ Support Package for NVIDIA GPUs. This example uses the resnet50 deep learning network to classify images from a USB webcam video stream.

Prerequisites

Target Board Requirements

- NVIDIA Jetson Tegra TX2 embedded platform.
- Ethernet crossover cable to connect the target board and host PC (if the target board cannot be connected to a local network).
- USB camera to connect to the TX2.
- NVIDIA CUDA toolkit installed on the target board.
- NVIDIA cuDNN library on the target board.
- OpenCV library on the target for reading and displaying images/video.
- Environment variables on the target for the compilers and libraries. For information on the supported versions of the compilers and libraries and their setup, see “Install and Setup Prerequisites for NVIDIA Boards” (MATLAB Coder Support Package for NVIDIA Jetson and NVIDIA DRIVE Platforms) for NVIDIA boards.

Development Host Requirements

- NVIDIA CUDA toolkit and driver.
- Environment variables for the compilers and libraries. For information on the supported versions of the compilers and libraries, see “Third-Party Hardware”. For setting up the environment variables, see “Setting Up the Prerequisite Products”.

Verify NVIDIA Support Package Installation on Host

Use the `checkHardwareSupportPackageInstall` function to verify that the host system is compatible to run this example.

```
checkHardwareSupportPackageInstall();
```

Connect to the NVIDIA Hardware

The GPU Coder Support Package for NVIDIA GPUs uses an SSH connection over TCP/IP to execute commands while building and running the generated CUDA code on the Jetson platform. You must therefore connect the target platform to the same network as the host computer or use an Ethernet crossover cable to connect the board directly to the host computer. Refer to the NVIDIA documentation on how to set up and configure your board.

To communicate with the NVIDIA hardware, you must create a live hardware connection object by using the `jetson` (MATLAB Coder Support Package for NVIDIA Jetson and NVIDIA DRIVE Platforms) function. You must know the host name or IP address, username, and password of the target board to create a live hardware connection object. For example, when connecting to the target board for the first time, create a live object for Jetson hardware by using the command:


```
hwobj= jetson('host-name', 'username', 'password');
```

The `jetson` object reuses these settings from the most recent successful connection to the Jetson hardware. This example establishes an SSH connection to the Jetson hardware using the settings stored in memory.

```
hwobj = jetson;
```

```
Checking for CUDA availability on the Target...
Checking for 'nvcc' in the target system path...
Checking for cuDNN library availability on the Target...
Checking for TensorRT library availability on the Target...
Checking for prerequisite libraries is complete.
Gathering hardware details...
Checking for third-party library availability on the Target...
Gathering hardware details is complete.
Board name           : NVIDIA Jetson TX2
CUDA Version         : 10.0
cuDNN Version        : 7.6
TensorRT Version     : 6.0
GStreamer Version    : 1.14.5
V4L2 Version         : 1.14.2-1
SDL Version          : 1.2
OpenCV Version       : 4.1.1
Available Webcams    :
Available GPUs       : NVIDIA Tegra X2
Available Digital Pins : 7 11 12 13 15 16 18 19 21 22 23 24 29 31 32 33 35 36
```

In case of a connection failure, a diagnostics error message is reported on the MATLAB command line. If the connection has failed, the most likely cause is incorrect IP address or hostname.

When there are multiple live connection objects for different targets, the code generator performs remote build on the target for which a recent live object was created. To choose a hardware board for performing remote build, use the `setupCodegenContext()` method of the respective live hardware object. If only one live connection object was created, it is not necessary to call this method.

```
hwobj.setupCodegenContext;
```

Verify GPU Environment on the Target

Use the `coder.checkGpuInstall` function to verify that the compilers and libraries necessary for running this example are set up correctly.

```
envCfg = coder.gpuEnvConfig('jetson');
envCfg.DeepLibTarget = 'cudnn';
envCfg.DeepCodegen = 1;
envCfg.Quiet = 1;
envCfg.HardwareObject = hwobj;
coder.checkGpuInstall(envCfg);
```

ResNet-50 Entry-Point Function

The `resnet50_wrapper.m` entry-point function uses a pre-trained ResNet-50 Network to classify images. ResNet-50 is a DAG Network trained on more than a million images from the ImageNet database. The output contains the categorical scores of each class the image belongs to.

```
type resnet50_wrapper
```

```
function out = resnet50_wrapper(im,ocvFlag) %#codegen
% Wrapper function to call ResNet50 predict function.

% Copyright 2019-2021 The MathWorks, Inc.

% This example uses OpenCV for reading frames from a web camera and
% displaying output image. Update buildinfo to link with OpenCV library
% available on target.
if ocvFlag
    opencv_link_flags = `pkg-config --libs opencv4`;
    opencv_compile_flags = `pkg-config --cflags opencv4`;
else
    opencv_link_flags = `pkg-config --libs opencv`;
    opencv_compile_flags = `pkg-config --cflags --libs opencv`;
end

coder.updateBuildInfo('addLinkFlags',opencv_link_flags);
coder.updateBuildInfo('addCompileFlags',opencv_compile_flags);

% To avoid multiple loads of the network for each run, we use persistent
% rnet
persistent rnet;
if isempty(rnet)
    rnet = resnet50();
end
out = rnet.predict(im);

end
```

Get OpenCV Version on the Target

Use the `pkg-config` helper tool to query if OpenCV 4.x is installed on the target board. This example uses the information to update build information to link with the appropriate OpenCV library available on target.

```
isOpenCV4 = 1;
ocvVersion = hwobj.OpenCVVersion();
if (str2double(ocvVersion(1)) <= 3)
    isOpenCV4 = 0;
end
```

Generate and Deploy CUDA Code on the Target

To generate a CUDA executable that can be deployed on to an NVIDIA target, create a GPU coder configuration object for generating an executable.

```
cfg = coder.gpuConfig('exe');
```

Use the `coder.hardware` function to create a configuration object for the Jetson platform and assign it to the `Hardware` property of the GPU code configuration object `cfg`.

```
cfg.Hardware = coder.hardware('NVIDIA Jetson');
```

Set Deep Learning Configuration to 'cudnn' or 'tensorrt'

```
cfg.DeepLearningConfig = coder.DeepLearningConfig('cudnn');
```

In this example, code generation is done using image as an input. However, webcam stream is fed a input to the executable after deployment.

Sample image input for code generation

```
im = single(imread('peppers.png'));  
im = imresize(im,[224,224]);
```

The custom main file is coded to take video as input and classifies each frame in the video sequence. The custom `main_resnet50.cu` file is a wrapper that calls the `predict` function in the generated code. Post processing steps such as displaying output on the input frame are added in the main file using OpenCV interfaces.

```
cfg.CustomSource = fullfile('main_resnet50.h');  
cfg.CustomSource = fullfile('main_resnet50.cu');
```

To generate CUDA code and deploy it onto target, use the `codegen` function and pass the GPU code configuration object. After the code generation takes place on the host, the generated files are copied over and built on the target in the workspace directory.

```
codegen -config cfg -args {im,coder.Constant(isOpenCV4)} resnet50_wrapper -report
```

Code generation successful: [View report](#)

Run the Application on the Target

Copy the `synsetWords_resnet50` text file from host computer to the target device by using the `putFile` command.

```
putFile(hwobj, 'synsetWords_resnet50.txt', hwobj.workspaceDir);
```

Use the `runApplication` method of the hardware object to launch the application on the target hardware. The application will be located in the workspace directory.

```
runApplication(hwobj, 'resnet50_wrapper');
```

If the webcam window is not visible on the target board, it may have been directed to the incorrect display. Use the `setDisplayEnvironment` function to set the display environment used for redirecting the display on the target. The value must be the same as the `$DISPLAY` environment value set on the board.

Resnet Classification Output on Jetson TX2



Kill the Application

Use the `killApplication` method of the hardware object to kill the running application on the target.

```
killApplication(hwobj, 'resnet50_wrapper');
```

See Also

Objects

jetson | drive

More About

- “Build and Run an Executable on NVIDIA Hardware” on page 5-2
- “Build and Run an Executable on NVIDIA Hardware Using GPU Coder App” on page 5-7
- “Stop or Restart an Executable Running on NVIDIA Hardware” (MATLAB Coder Support Package for NVIDIA Jetson and NVIDIA DRIVE Platforms)
- “Run Linux Commands on NVIDIA Hardware” (MATLAB Coder Support Package for NVIDIA Jetson and NVIDIA DRIVE Platforms)

Ground Plane Segmentation and Obstacle Detection on NVIDIA Jetson Xavier™ NX Embedded platform

This example shows ground plane segmentation of 3-D lidar data from a vehicle on NVIDIA® embedded platforms to find nearby obstacles. The example uses ground plane segmentation and obstacle detection application to illustrate:

- C++ and CUDA® code generation for the ground plane segmentation and obstacle detection algorithm by using MATLAB® Coder™ and GPU Coder™.
- Verify behavior of the generated code on the target platform by using processor-in-the-loop (PIL) simulation.
- Compare of the performance of the application on the CPU (C++) and the GPU (CUDA).

Third-Party Prerequisites

Target Board Requirements

- NVIDIA Jetson Xavier™ NX Embedded platform.
- NVIDIA CUDA toolkit installed on the board.
- Environment variables on the target board for the compilers and libraries. For more information, see “Install and Setup Prerequisites for NVIDIA Boards” (MATLAB Coder Support Package for NVIDIA Jetson and NVIDIA DRIVE Platforms).

Development Host Requirements

- NVIDIA CUDA toolkit installed on the host.
- Environment variables for the compilers and libraries. For information on the supported versions of the compilers and libraries, see “Third-Party Hardware”. For setting up the environment variables, see “Setting Up the Prerequisite Products”.

Configure and Verify NVIDIA Target Platform

Connect to NVIDIA Hardware

The MATLAB Coder Support Package for NVIDIA Jetson and NVIDIA DRIVE™ Platforms uses SSH connection over TCP/IP to execute commands while building and running the generated code on the Jetson platforms. Connect the target platform to the same network as the host computer.

To communicate with the NVIDIA hardware, create a live hardware connection object by using the `jetson` (MATLAB Coder Support Package for NVIDIA Jetson and NVIDIA DRIVE Platforms) function. This example uses the device address, user name, and password settings from the most recent successful connection to the Jetson hardware.

```
hwobj = jetson;
```

Configure PIL Simulation

This example uses processor-in-the-loop (PIL) simulation to test the generated C++ and CUDA code on the Jetson board. Because the input data transfer and algorithm computations consumes time, change the default higher PIL timeout value to prevent time-out errors.

```
setPILTimeout(hwobj,100);
```

Verify GPU Environment

To verify that the compilers and libraries necessary for running this example are set up correctly, use the `coder.checkGpuInstall` function.

```
envCfg = coder.gpuEnvConfig('jetson');  
envCfg.BasicCodegen = 1;  
envCfg.Quiet = 1;  
envCfg.HardwareObject = hwobj;  
coder.checkGpuInstall(envCfg);
```

Configure Code Generation Parameters

To generate a PIL executable that runs on the ARM® CPU of the Jetson board, create a `coder.EmbeddedCodeConfig` object for a static library.

```
cfgCpu = coder.config('lib');
```

Set the target language for the generated code to C++ and enable PIL execution in the code configuration object. Then, enable execution-time profiling during PIL execution. Execution-time profiling generates metrics for tasks and functions in the generated code. For more information, see “Create Execution-Time Profile for Generated Code” (Embedded Coder). Finally, create a `coder.hardware` object for the Jetson platform and assign it to the `Hardware` property of the code configuration object.

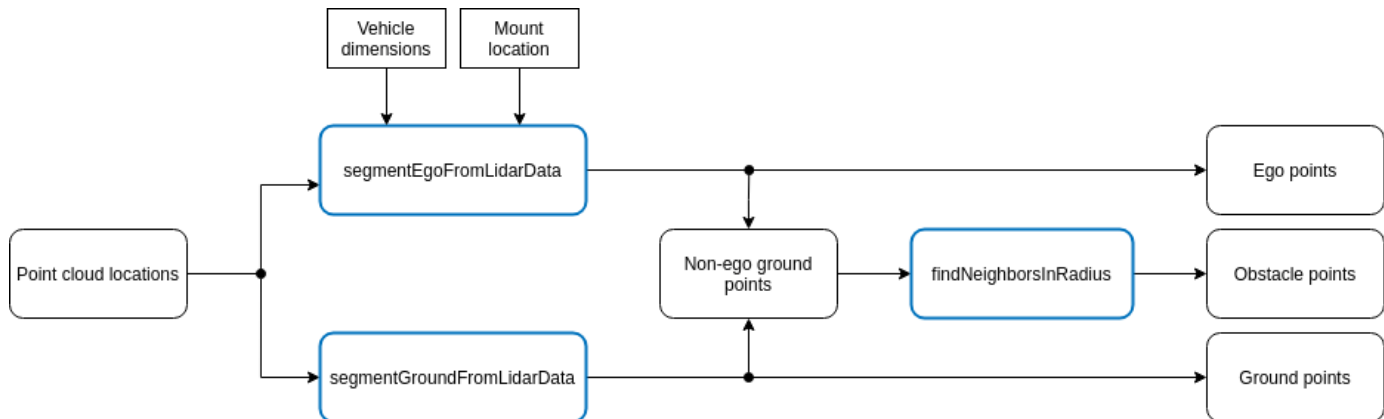
```
cfgCpu.TargetLang = 'C++';  
cfgCpu.VerificationMode = 'PIL';  
cfgCpu.CodeExecutionProfiling = true;  
cfgCpu.Hardware = coder.hardware('NVIDIA Jetson');
```

Similarly, create configuration parameters for the CUDA GPU on the Jetson board by using `coder.gpuConfig`.

```
cfgGpu = coder.gpuConfig('lib');  
cfgGpu.VerificationMode = 'PIL';  
cfgGpu.CodeExecutionProfiling = true;  
cfgGpu.Hardware = coder.hardware('NVIDIA Jetson');
```

The `segmentGroundAndObstacles` Entry-Point Function

The `segmentGroundAndObstacles` entry-point function segments points belonging to the ground plane, the ego vehicle, and nearby obstacles from the input point cloud locations. The following diagram illustrates the algorithm implemented in the entry-point function. For more information, see “Ground Plane and Obstacle Detection Using Lidar” (Automated Driving Toolbox).



type `segmentGroundAndObstacles`

Generate and Run Executables on the Target

The point cloud data from the lidar sensor is of size 32-by-1100-by-3. Due to signal misses and noise, a few points may be dropped from this point cloud data. So, the second dimension might vary with an upper bound of 1100. Create the input type for the entry-point function `segmentGroundAndObstacles` with varying dimensions for the second argument using the `coder.typeof` function.

```
codegenArgs = {coder.typeof(single(0), [32, 1100, 3], [0, 1, 0])};
```

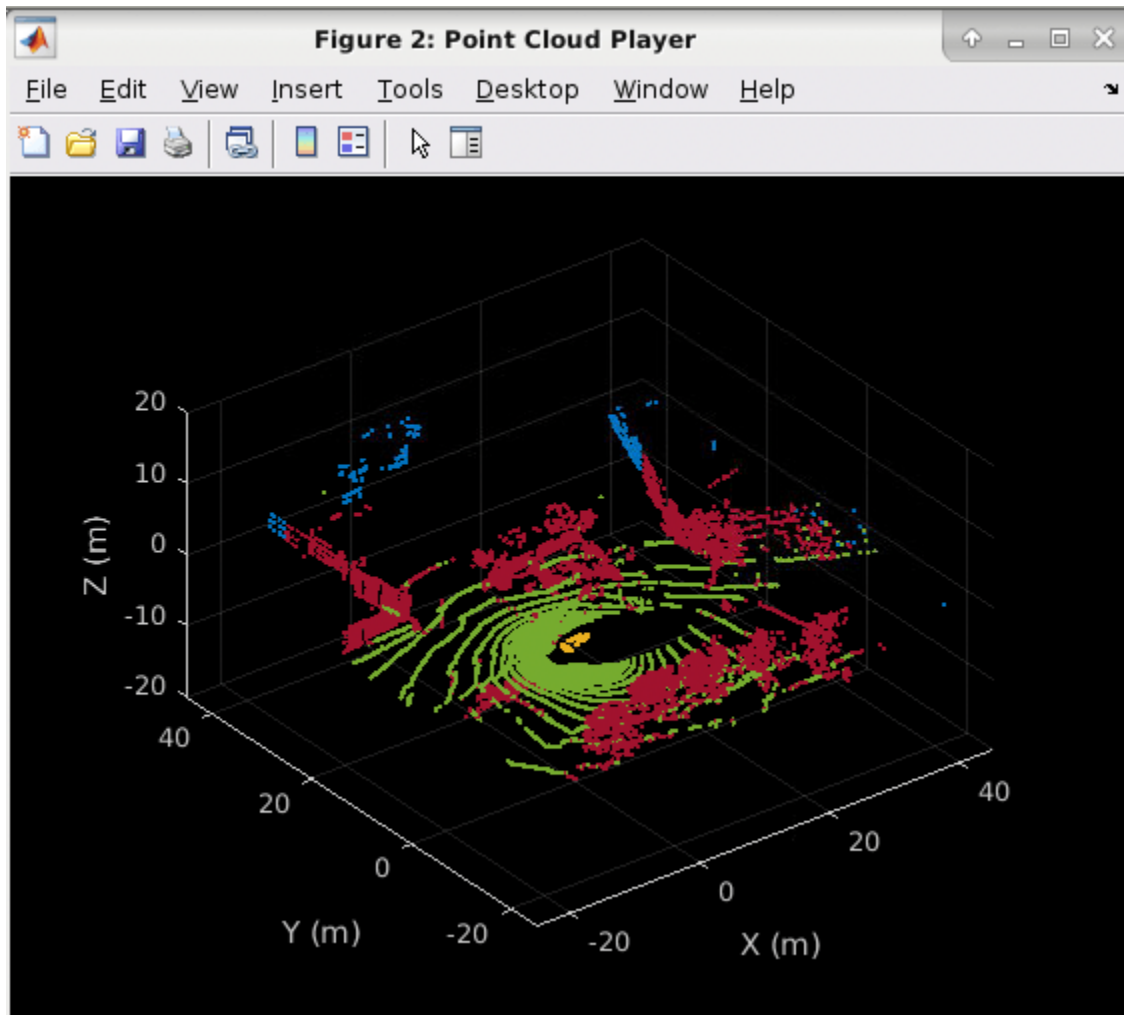
Generate and Run C++ Executable

Generate C++ code with the CPU code configuration object `cfgCpu`.

```
codegen -config cfgCpu -args codegenArgs segmentGroundAndObstacles -report
```

The `obstacleDetectionWrapper` is an execution wrapper function that processes the streaming lidar input data frame-by-frame, calls the PIL executable, and displays the 3-D point cloud with segmenting points belonging to the ground plane, the ego vehicle, and nearby obstacles. The lidar data used in this example was recorded using a Velodyne HDL32E sensor mounted on a vehicle. For an explanation of the processing performed by the `obstacleDetectionWrapper` function, see “Ground Plane and Obstacle Detection Using Lidar” (Automated Driving Toolbox).

```
obstacleDetectionWrapper();
```



Execution Profile of the C++ Executable

Clear the PIL executable and collect the execution time profile by using the `getCoderExecutionProfile` function.

```
clear segmentGroundAndObstacles_pil;
cpuExecutionProfile = getCoderExecutionProfile('segmentGroundAndObstacles');
```

Generate and Run CUDA Executable

Generate CUDA code with the GPU code configuration object `cfgGpu`.

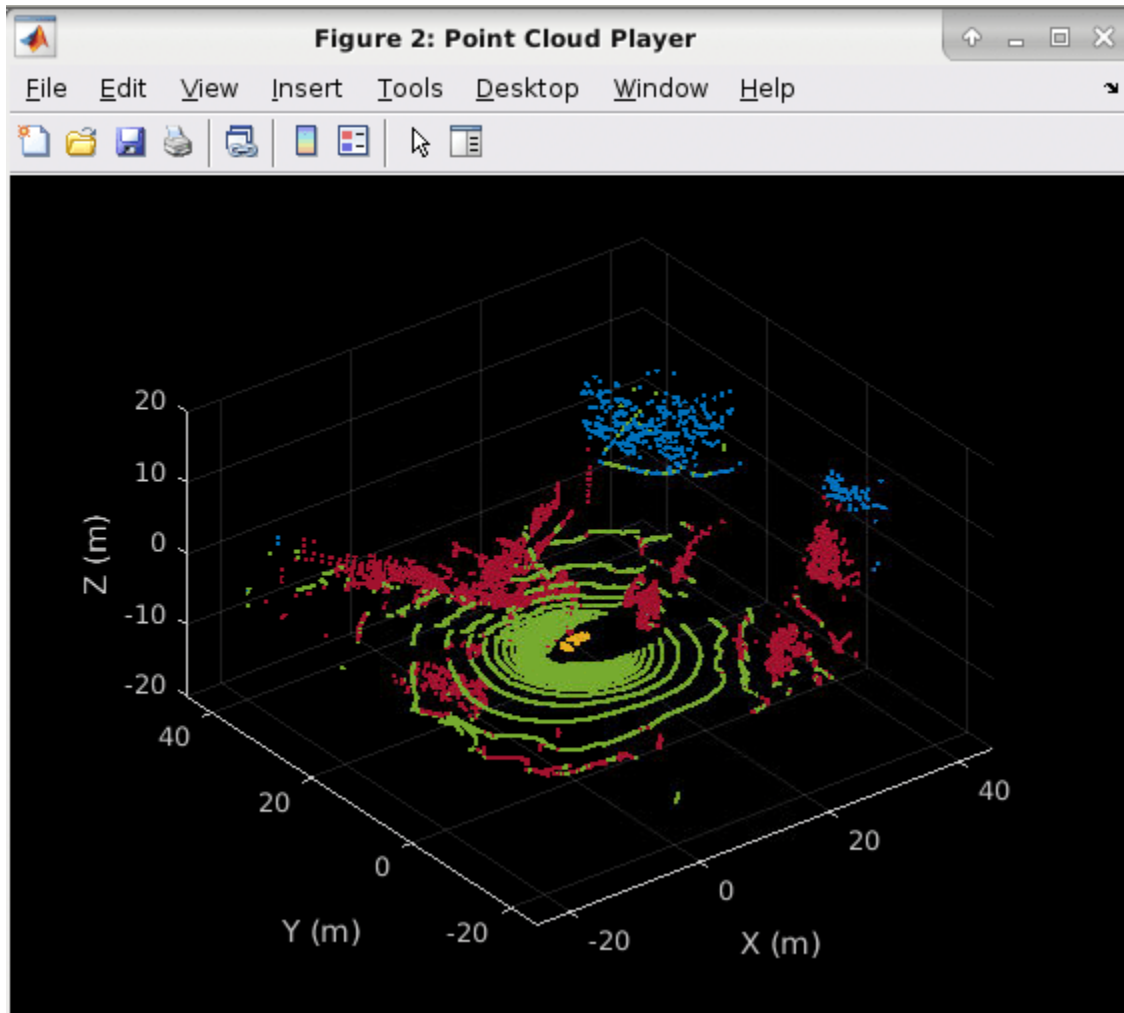
```
codegen -config cfgGpu -args codegenArgs segmentGroundAndObstacles -report
```

To maximize the GPU performance, use the `jetson_clocks.sh` script on the board. For more information, see [NVIDIA Xavier - Maximizing Performance \(RidgeRun wiki\)](#).

```
ClockFileStatus = system(hwobj, 'test -f l4t_dfs.conf && echo "1" || echo "0"');
if ~str2double(ClockFileStatus)
    system(hwobj, 'echo "ubuntu" | sudo -S jetson_clocks --store');
end
system(hwobj, 'echo "ubuntu" | sudo -S jetson_clocks');
```


Use the `obstacleDetectionWrapper` to process the streaming lidar input data, calls the PIL executable, and display the 3-D point cloud with segmenting points belonging to the ground plane, the ego vehicle, and nearby obstacles.

```
obstacleDetectionWrapper();
```



Execution Profile of the CUDA Executable

Disable Jetson clock settings, clear the PIL executable and collect the execution time profile by using the `getCoderExecutionProfile` function.

```
system(hwobj, 'echo "ubuntu" | sudo -S jetson_clocks --restore');
clear segmentGroundAndObstacles_pil;
gpuExecutionProfile = getCoderExecutionProfile('segmentGroundAndObstacles');
```

Analysis of CPU and GPU Execution Profiles

Get the per frame execution times of CPU and GPU from their execution profiles by using the `ExecutionTimeInSeconds` property.

```
[~,cpuExecTimePerFrame,~] = cpuExecutionProfile.Sections.ExecutionTimeInSeconds;
[~,gpuExecTimePerFrame,~] = gpuExecutionProfile.Sections.ExecutionTimeInSeconds;
```

To plot the per frame execution times use the code mentioned below:

```
figure;
% Plot CPU execution times.
plot(cpuExecTimePerFrame(2:end)*1000,'r');
hold on;
% Plot GPU execution times.
plot(gpuExecTimePerFrame(2:end)*1000,'b');
grid on;
% Set the title, legend and labels.
title('CPU vs GPU Per-frame Execution times (in ms)');
legend('GPU Timings', 'CPU Timings');
axis([0,1240,0,40]);
xlabel('Frame Number');
ylabel('Execution Time (in ms)');
```

Supporting Functions

obstacleDetectionWrapper processes the lidar data, calls the PIL executable, and visualize the results. For an explanation of the processing performed by the obstacleDetectionWrapper function, see “Ground Plane and Obstacle Detection Using Lidar” (Automated Driving Toolbox).

```
function obstacleDetectionWrapper()
%OBSTACLEDETECTIONWRAPPER process lidar data and visualize results
% The OBSTACLEDETECTIONWRAPPER is an execution wrapper function that
% processess the streaming lidar input data frame-by-frame, calls the PIL
% executable, and displays the 3-D point cloud with segmenting points
% belonging to the ground plane, the ego vehicle, and nearby obstacles.

fileName = 'lidarData_ConstructionRoad.pcap';
deviceModel = 'HDL32E';
veloReader = velodyneFileReader(fileName, deviceModel);

% Setup Streaming Point Cloud Display
xlimits = [-25 45]; % Limits of point cloud display, meters
ylimits = [-25 45];
zlimits = [-20 20];

% Create a pcplayer
lidarViewer = pcplayer(xlimits, ylimits, zlimits);
xlabel(lidarViewer.Axes, 'X (m)')
ylabel(lidarViewer.Axes, 'Y (m)')
zlabel(lidarViewer.Axes, 'Z (m)')

% Set the colormap for labeling the ground plane, ego vehicle, and nearby
% obstacles.
colorLabels = [...
    0      0.4470 0.7410; ... % Unlabeled points, specified as [R,G,B]
    0.4660 0.6740 0.1880; ... % Ground points
    0.9290 0.6940 0.1250; ... % Ego points
    0.6350 0.0780 0.1840]; % Obstacle points

% Define indices for each label
colors.Unlabeled = 1;
colors.Ground = 2;
colors.Ego = 3;
colors.Obstacle = 4;
```

```

% Set the colormap
colormap(lidarViewer.Axes, colorLabels)

% Stop processing the frame after specified time.
stopTime = veloReader.EndTime;

i = 1;
isPlayerOpen = true;
while hasFrame(veloReader) && veloReader.CurrentTime < stopTime && isPlayerOpen

    % Grab the next lidar scan
    ptCloud = readFrame(veloReader);

    % Segment points belonging to the ego vehicle
    [egoPoints,groundPoints,obstaclePoints] = segmentGroundAndObstacles_pil(ptCloud.Location);

    i = i+1;
    closePlayer = ~hasFrame(veloReader);

    % Update lidar display
    points = struct('EgoPoints',egoPoints, 'GroundPoints',groundPoints, ...
        'ObstaclePoints',obstaclePoints);
    isPlayerOpen = helperUpdateView(lidarViewer, ptCloud, points, colors, closePlayer);
end
snapnow
end

```

`helperUpdateView` updates the streaming point cloud display with the latest point cloud and associated color labels.

```

function isPlayerOpen = helperUpdateView(lidarViewer,ptCloud,points,colors,closePlayer)
%HELPERUPDATEVIEW update streaming point cloud display
% isPlayerOpen =
% HELPERUPDATEVIEW(lidarViewer,ptCloud,points,colors,closePlayers)
% updates the pcplayer object specified in lidarViewer with a new point
% cloud ptCloud. Points specified in the struct points are colored
% according to the colormap of lidarViewer using the labels specified by
% the struct colors. closePlayer is a flag indicating whether to close
% the lidarViewer.

if closePlayer
    hide(lidarViewer);
    isPlayerOpen = false;
    return;
end

scanSize = size(ptCloud.Location);
scanSize = scanSize(1:2);

% Initialize colormap
colormapValues = ones(scanSize, 'like', ptCloud.Location) * colors.Unlabeled;

if isfield(points, 'GroundPoints')
    colormapValues(points.GroundPoints) = colors.Ground;
end

if isfield(points, 'EgoPoints')
    colormapValues(points.EgoPoints) = colors.Ego;
end

```

```
end

if isfield(points, 'ObstaclePoints')
    colormapValues(points.ObstaclePoints) = colors.Obstacle;
end

% Update view
view(lidarViewer, ptCloud.Location, colormapValues)

% Check if player is open
isPlayerOpen = isOpen(lidarViewer);

end
```

See Also

Objects

jetson | drive

More About

- “Build and Run an Executable on NVIDIA Hardware” on page 5-2
- “Build and Run an Executable on NVIDIA Hardware Using GPU Coder App” on page 5-7
- “Stop or Restart an Executable Running on NVIDIA Hardware” (MATLAB Coder Support Package for NVIDIA Jetson and NVIDIA DRIVE Platforms)
- “Run Linux Commands on NVIDIA Hardware” (MATLAB Coder Support Package for NVIDIA Jetson and NVIDIA DRIVE Platforms)

Deploy Signal Classifier on NVIDIA Jetson Using Wavelet Analysis and Deep Learning

This example shows how to generate and deploy a CUDA® executable that classifies human electrocardiogram (ECG) signals using features extracted by the continuous wavelet transform (CWT) and a pretrained convolutional neural network (CNN).

SqueezeNet is a deep CNN originally designed to classify images in 1000 categories. We reuse the network architecture of the CNN to classify ECG signals based on their scalograms. A scalogram is the absolute value of the CWT of the signal. After training SqueezeNet to classify ECG signals, you create a CUDA executable that generates a scalogram of an ECG signal and then uses the CNN to classify the signal. The executable and CNN are both deployed to the NVIDIA hardware.

This example uses the same data as used in “Classify Time Series Using Wavelet Analysis and Deep Learning” (Wavelet Toolbox). In that example, transfer learning with GoogLeNet and SqueezeNet are used to classify ECG waveforms into one of three categories. The description of the data and how to obtain it are repeated here for convenience.

ECG Data Description and Download

The ECG data is obtained from three groups of people: persons with cardiac arrhythmia (ARR), persons with congestive heart failure (CHF), and persons with normal sinus rhythms (NSR). In total there are 162 ECG recordings from three PhysioNet databases: MIT-BIH Arrhythmia Database [2][3], MIT-BIH Normal Sinus Rhythm Database [3], and The BIDMC Congestive Heart Failure Database [1] [3]. More specifically, 96 recordings from persons with arrhythmia, 30 recordings from persons with congestive heart failure, and 36 recordings from persons with normal sinus rhythms. The goal is to train a model to distinguish between ARR, CHF, and NSR.

You can obtain this data from the MathWorks GitHub repository. To download the data from the website, click Code and select Download ZIP. Save the file `physionet_ECG_data-main.zip` in a folder where you have write permission. The instructions for this example assume you have downloaded the file to your temporary directory, `tempdir`, in MATLAB. Modify the subsequent instructions for unzipping and loading the data if you choose to download the data in a folder different from `tempdir`.

After downloading the data from GitHub, unzip the file in your temporary directory.

```
unzip(fullfile(tempdir, 'physionet_ECG_data-main.zip'), tempdir)
```

Unzipping creates the folder `physionet-ECG_data-main` in your temporary directory. This folder contains the text file `README.md` and `ECGData.zip`. The `ECGData.zip` file contains:

- `ECGData.mat`
- `Modified_physionet_data.txt`
- `License.txt`

`ECGData.mat` holds the data used in this example. The text file `Modified_physionet_data.txt` is required by PhysioNet's copying policy and provides the source attributions for the data as well as a description of the preprocessing steps applied to each ECG recording.

Unzip `ECGData.zip` in `physionet-ECG_data-main`. Load the data file into your MATLAB workspace.

```

unzip(fullfile(tempdir,'physionet_ECG_data-main','ECGData.zip'),...
    fullfile(tempdir,'physionet_ECG_data-main'))
load(fullfile(tempdir,'physionet_ECG_data-main','ECGData.mat'))

```

ECGData is a structure array with two fields: `Data` and `Labels`. The `Data` field is a 162-by-65536 matrix where each row is an ECG recording sampled at 128 hertz. `Labels` is a 162-by-1 cell array of diagnostic labels, one label for each row of `Data`. The three diagnostic categories are: 'ARR', 'CHF', and 'NSR'.

Feature Extraction

After downloading the data, you must generate scalograms of the signals. The scalograms are the "input" images to the CNN.

To store the scalograms of each category, first create an ECG data directory 'data' inside `tempdir`. Then create three subdirectories in 'data' named after each ECG category. The helper function `helperCreateECGDirectories` does this for you. `helperCreateECGDirectories` accepts `ECGData`, the name of an ECG data directory, and the name of a parent directory as input arguments. You can replace `tempdir` with another directory where you have write permission. You can find the source code for this helper function in the Supporting Functions on page 5-70 section at the end of this example.

```

parentDir = tempdir;
dataDir = 'data';
helperCreateECGDirectories(ECGData,parentDir,dataDir)

```

After making the folders, create scalograms of the ECG signals as RGB images and write them to the appropriate subdirectory in `dataDir`. To create the scalograms, first precompute a CWT filter bank. Precomputing the filter bank is the preferred method when obtaining the CWT of many signals using the same parameters. The helper function `helperCreateRGBfromTF` does this. The source code for this helper function is in the Supporting Functions on page 5-70 section at the end of this example. To be compatible with the SqueezeNet architecture, each RGB image is an array of size 227-by-227-by-3.

```

helperCreateRGBfromTF(ECGData,parentDir,dataDir)

```

Divide Data Set into Training and Validation Data

Load the scalogram images as an image datastore. The `imageDatastore` function automatically labels the images based on folder names and stores the data as an `ImageDatastore` object. An image datastore enables you to store large image data, including data that does not fit in memory, and efficiently read batches of images when training a CNN.

```

allImages = imageDatastore(fullfile(tempdir,dataDir),...
    'IncludeSubfolders',true,...
    'LabelSource','foldernames');

```

Randomly divide the images into two groups, one for training and the other for validation. Use 80% of the images for training and the remainder for validation. For purposes of reproducibility, we set the random seed to the default value.

```

rng default
[imgsTrain,imgsValidation] = splitEachLabel(allImages,0.8,'randomized');
disp(['Number of training images: ',num2str(numel(imgsTrain.Files))]);

```

```

Number of training images: 130

```

```
disp(['Number of validation images: ', num2str(numel(imgsValidation.Files))]);
```

```
Number of validation images: 32
```

SqueezeNet

SqueezeNet is a pretrained CNN that can classify images into 1000 categories. You need to retrain SqueezeNet for our ECG classification problem. Prior to retraining, you modify several network layers and set various training options. After retraining is complete, you save the CNN in a `.mat` file. The CUDA executable will use the `.mat` file.

Specify an experiment trial index and a results directory. If necessary, create the directory.

```
trial = 1;
ResultDir = 'results';
if ~exist(ResultDir, 'dir')
    mkdir(ResultDir)
end
MatFile = fullfile(ResultDir, sprintf('SqueezeNet_Trial%d.mat', trial));
```

Load SqueezeNet. Extract the layer graph and inspect the last five layers.

```
sqz = squeezenet;
lgraph = layerGraph(sqz);
lgraph.Layers(end-4:end)
```

```
ans =
```

```
5x1 Layer array with layers:
```

1	'conv10'	Convolution	1000 1x1x512 convolutio
2	'relu_conv10'	ReLU	ReLU
3	'pool10'	2-D Global Average Pooling	2-D global average poo
4	'prob'	Softmax	softmax
5	'ClassificationLayer_predictions'	Classification Output	crossentropyex with 't

To retrain SqueezeNet to classify the three classes of ECG signals, replace the `'conv10'` layer with a new convolutional layer with the number of filters equal to the number of ECG classes. Replace the classification layer with a new one without class labels.

```
numClasses = numel(categories(imgsTrain.Labels));
new_conv10_WeightLearnRateFactor = 1;
new_conv10_BiasLearnRateFactor = 1;
newConvLayer = convolution2dLayer(1, numClasses, ...
    'Name', 'new_conv10', ...
    'WeightLearnRateFactor', new_conv10_WeightLearnRateFactor, ...
    'BiasLearnRateFactor', new_conv10_BiasLearnRateFactor);
lgraph = replaceLayer(lgraph, 'conv10', newConvLayer);
newClassLayer = classificationLayer('Name', 'new_classoutput');
lgraph = replaceLayer(lgraph, 'ClassificationLayer_predictions', newClassLayer);
lgraph.Layers(end-4:end)
```

```
ans =
```

```
5x1 Layer array with layers:
```

1	'new_conv10'	Convolution	3 1x1 convolutions with stride [1 1] a
2	'relu_conv10'	ReLU	ReLU
3	'pool10'	2-D Global Average Pooling	2-D global average pooling
4	'prob'	Softmax	softmax
5	'new_classoutput'	Classification Output	crossentropyex

Create a set of training options to use with SqueezeNet.

```

OptimSolver = 'sgdm';
MiniBatchSize = 15;
MaxEpochs = 20;
InitialLearnRate = 1e-4;
Momentum = 0.9;
ExecutionEnvironment = 'cpu';

options = trainingOptions(OptimSolver,...
    'MiniBatchSize',MiniBatchSize,...
    'MaxEpochs',MaxEpochs,...
    'InitialLearnRate',InitialLearnRate,...
    'ValidationData',imgsValidation,...
    'ValidationFrequency',10,...
    'ExecutionEnvironment',ExecutionEnvironment,...
    'Momentum',Momentum);

```

Save all the parameters in a structure. The trained network and structure will be later saved in a .mat file.

```

TrialParameter.new_conv10_WeightLearnRateFactor = new_conv10_WeightLearnRateFactor;
TrialParameter.new_conv10_BiasLearnRateFactor = new_conv10_BiasLearnRateFactor;
TrialParameter.OptimSolver = OptimSolver;
TrialParameter.MiniBatchSize = MiniBatchSize;
TrialParameter.MaxEpochs = MaxEpochs;
TrialParameter.InitialLearnRate = InitialLearnRate;
TrialParameter.Momentum = Momentum;
TrialParameter.ExecutionEnvironment = ExecutionEnvironment;

```

Set the random seed to the default value and train the network. Save the trained network, trial parameters, training run time, and image datastore containing the validation images. The training process usually takes 1-5 minutes on a desktop CPU. If you want to use a trained CNN from a previous trial, set trial to the index number of that trial and LoadModel to true.

```

LoadModel = false;
if ~LoadModel
    rng default
    tic;
    trainedModel = trainNetwork(imgsTrain,lgraph,options);
    trainingTime = toc;
    fprintf('Total training time: %.2e sec\n',trainingTime);
    save(MatFile,'TrialParameter','trainedModel','trainingTime','imgsValidation');
else
    disp('Load ML model from the file')
    load(MatFile,'trainedModel','imgsValidation');
end

```

Initializing input data normalization.

Epoch	Iteration	Time Elapsed (hh:mm:ss)	Mini-batch Accuracy	Validation Accuracy	Mini-batch Loss	Validation Loss
1	1	00:00:03	26.67%	25.00%	4.1769	2.9
2	10	00:00:18	73.33%	59.38%	0.9875	1.7
3	20	00:00:35	60.00%	56.25%	0.9157	0.9
4	30	00:00:52	86.67%	68.75%	0.6708	0.7
5	40	00:01:10	66.67%	68.75%	0.9026	0.7

7	50	00:01:29	80.00%	78.12%	0.5429	0.4
8	60	00:01:48	100.00%	81.25%	0.4165	0.4
9	70	00:02:06	93.33%	84.38%	0.3590	0.5
10	80	00:02:24	73.33%	84.38%	0.5113	0.4
12	90	00:02:42	86.67%	84.38%	0.4211	0.4
13	100	00:03:00	93.33%	90.62%	0.1935	0.3
14	110	00:03:18	100.00%	90.62%	0.1488	0.3
15	120	00:03:36	100.00%	93.75%	0.0788	0.2
17	130	00:03:55	86.67%	93.75%	0.2489	0.2
18	140	00:04:13	100.00%	93.75%	0.0393	0.2
19	150	00:04:32	100.00%	93.75%	0.0522	0.2
20	160	00:04:50	100.00%	93.75%	0.0227	0.2

Training finished: Max epochs completed.

Total training time: 3.03e+02 sec

Save only the trained network in a separate .mat file. This file will be used by the CUDA executable.

```
ModelFile = fullfile(ResultDir, sprintf('SqueezeNet_Trial%d.mat', trial));
OutMatFile = fullfile('ecg_model.mat');
```

```
data = load(ModelFile, 'trainedModel');
net = data.trainedModel;
save(OutMatFile, 'net');
```

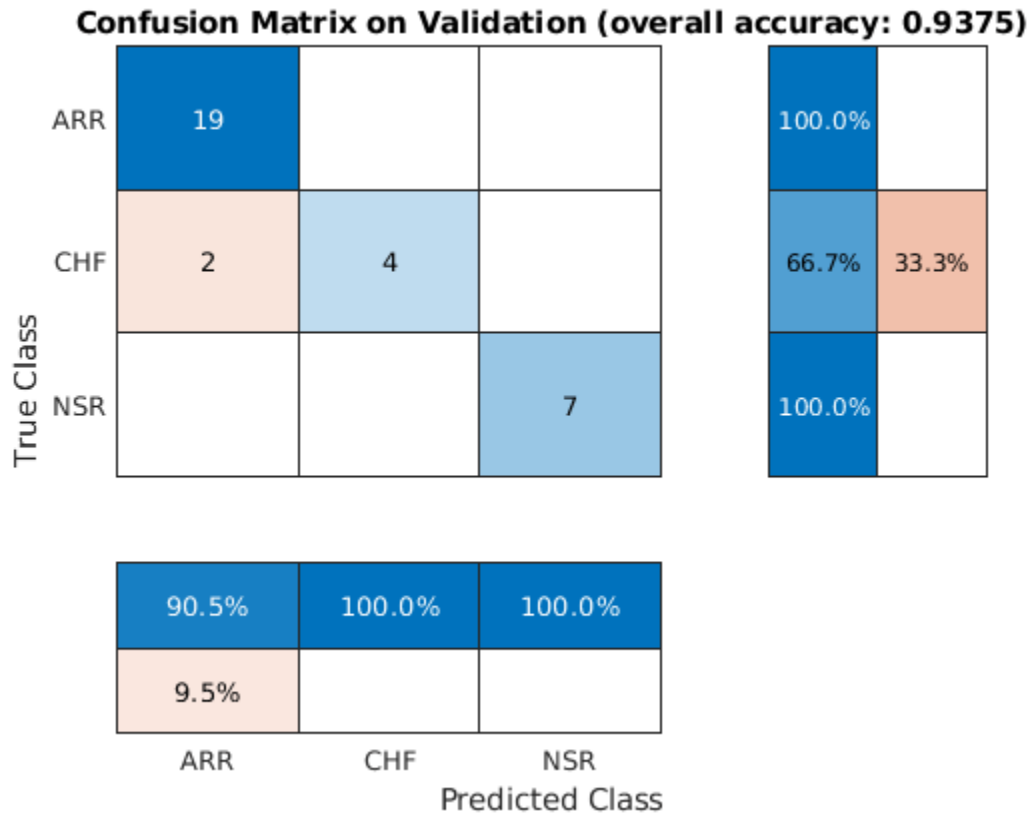
Use the trained network to predict the classes for the validation set.

```
[YPred, probs] = classify(trainedModel, imgsValidation);
accuracy = mean(YPred==imgsValidation.Labels)
```

```
accuracy = 0.9375
```

Summarize the performance of the trained network on the validation set with a confusion chart. Display the precision and recall for each class by using column and row summaries. Save the figure. The table at the bottom of the confusion chart shows the precision values. The table to the right of the confusion chart shows the recall values.

```
figure
confusionMat = confusionmat(imgsValidation.Labels, YPred);
confusionchart(imgsValidation.Labels, YPred, ...
    'Title', sprintf('Confusion Matrix on Validation (overall accuracy: %.4f)', accuracy), ...
    'ColumnSummary', 'column-normalized', 'RowSummary', 'row-normalized');
```



```
AccFigFile = fullfile(ResultDir, sprintf('SqueezeNet_ValidationAccuracy_Trial%d.fig', trial));
saveas(gcf, AccFigFile);
```

Display the size of the trained network.

```
info = whos('trainedModel');
ModelMemSize = info.bytes/1024;
fprintf('Trained network size: %g kB\n', ModelMemSize)
```

```
Trained network size: 2991.89 kB
```

Determine the average time it takes the network to classify an image.

```
NumTestForPredTime = 20;
TrialParameter.NumTestForPredTime = NumTestForPredTime;

fprintf('Test prediction time (number of tests: %d)... ', NumTestForPredTime)
```

```
Test prediction time (number of tests: 20)...
```

```
imageSize = trainedModel.Layers(1).InputSize;
PredTime = zeros(NumTestForPredTime, 1);
for i = 1:NumTestForPredTime
    x = randn(imageSize);
    tic;
    [YPred, probs] = classify(trainedModel, x, 'ExecutionEnvironment', ExecutionEnvironment);
    PredTime(i) = toc;
end
```

```
AvgPredTimePerImage = mean(PredTime);
fprintf('Average prediction time (execution environment: %s): %.2e sec \n', ...
    ExecutionEnvironment, AvgPredTimePerImage);
```

```
Average prediction time (execution environment: cpu): 1.67e-01 sec
```

Save the results.

```
if ~LoadModel
    save(MatFile, 'accuracy', 'confusionMat', 'PredTime', 'ModelMemSize', ...
        'AvgPredTimePerImage', '-append')
end
```

GPU Code Generation — Define Functions

The scalogram of a signal is the input "image" to a deep CNN. Create a function, `cwt_ecg_jetson_ex`, that computes the scalogram of an input signal and returns an image at the user-specified dimensions. The image uses the `jet(128)` colormap. The `codegen` directive in the function indicates that the function is intended for code generation. When using the `coder.gpu.kernelfun` pragma, code generation attempts to map the computations in the `cwt_ecg_jetson_ex` function to the GPU.

```
type cwt_ecg_jetson_ex.m

function im = cwt_ecg_jetson_ex(TimeSeriesSignal, ImgSize) %codegen
% This function is only intended to support wavelet deep learning examples.
% It may change or be removed in a future release.

coder.gpu.kernelfun();

%% Create Scalogram
cfs = cwt(TimeSeriesSignal, 'morse', 1, 'VoicesPerOctave', 12);
cfs = abs(cfs);

%% Image generation
cmapj128 = coder.load('cmapj128');
imx = ind2rgb_custom_ecg_jetson_ex(round(255*rescale(cfs))+1, cmapj128.cmapj128);

% resize to proper size and convert to uint8 data type
im = im2uint8(imresize(imx, ImgSize));

end
```

Create the entry-point function, `model_predict_ecg.m`, for code generation. The function takes an ECG signal as input and calls the `cwt_ecg_jetson_ex` function to create an image of the scalogram. The `model_predict_ecg` function uses the network contained in the `ecg_model.mat` file to classify the ECG signal.

```
type model_predict_ecg.m

function PredClassProb = model_predict_ecg(TimeSeriesSignal) %codegen
% This function is only intended to support wavelet deep learning examples.
% It may change or be removed in a future release.
    coder.gpu.kernelfun();

    % parameters
    ModFile = 'ecg_model.mat'; % file that saves neural network model
    ImgSize = [227 227]; % input image size for the ML model
```

```

% sanity check signal is a row vector of correct length
assert(isequal(size(TimeSeriesSignal), [1 65536]))
%% cwt transformation for the signal
im = cwt_ecg_jetson_ex(TimeSeriesSignal, ImgSize);

%% model prediction
persistent model;
if isempty(model)
    model = coder.loadDeepLearningNetwork(ModFile, 'mynet');
end

PredClassProb = predict(model, im);

end

```

To generate a CUDA executable that can be deployed to an NVIDIA target, create a custom main file (`main_ecg_jetson_ex.cu`) and a header file (`main_ecg_jetson_ex.h`). You can generate an example main file and use that as a template to rewrite new main and header files. For more information, see the `GenerateExampleMain` property of `coder.CodeConfig`. The main file calls the code generated for the MATLAB entry-point function. The main file first reads the ECG signal from a text file, passes the data to the entry-point function, and writes the prediction results to a text file (`predClassProb.txt`). To maximize computation efficiency on the GPU, the executable processes single-precision data.

```
type main_ecg_jetson_ex.cu
```

```

//
// File: main_ecg_jetson_ex.cu
//
// This file is only intended to support wavelet deep learning examples.
// It may change or be removed in a future release.

//*****
// Include Files
#include "rt_nonfinite.h"
#include "model_predict_ecg.h"
#include "main_ecg_jetson_ex.h"
#include "model_predict_ecg_terminate.h"
#include "model_predict_ecg_initialize.h"
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

// Function Definitions

/* Read data from a file*/
int readData_real32_T(const char * const file_in, real32_T data[65536])
{
    FILE* fp1 = fopen(file_in, "r");
    if (fp1 == 0)
    {
        printf("ERROR: Unable to read data from %s\n", file_in);
        exit(0);
    }
    for(int i=0; i<65536; i++)
    {
        fscanf(fp1, "%f", &data[i]);
    }
}

```

```

    }
    fclose(fp1);
    return 0;
}

/* Write data to a file*/
int writeData_real32_T(const char * const file_out, real32_T data[3])
{
    FILE* fp1 = fopen(file_out, "w");
    if (fp1 == 0)
    {
        printf("ERROR: Unable to write data to %s\n", file_out);
        exit(0);
    }
    for(int i=0; i<3; i++)
    {
        fprintf(fp1, "%f\n", data[i]);
    }
    fclose(fp1);
    return 0;
}

// model predict function
static void main_model_predict_ecg(const char * const file_in, const char * const file_out)
{
    real32_T PredClassProb[3];
    // real_T b[65536];
    real32_T b[65536];

    // readData_real_T(file_in, b);
    readData_real32_T(file_in, b);

    model_predict_ecg(b, PredClassProb);

    writeData_real32_T(file_out, PredClassProb);
}

// main function
int32_T main(int32_T argc, const char * const argv[])
{
    const char * const file_out = "predClassProb.txt";
    // Initialize the application.
    model_predict_ecg_initialize();

    // Run prediction function
    main_model_predict_ecg(argv[1], file_out); // argv[1] = file_in

    // Terminate the application.
    model_predict_ecg_terminate();
    return 0;
}

type main_ecg_jetson_ex.h

//
// File: main_ecg_jetson_ex.h
//

```

```

// This file is only intended to support wavelet deep learning examples.
// It may change or be removed in a future release.

//
//*****
#ifndef MAIN_H
#define MAIN_H

// Include Files
#include <stddef.h>
#include <stdlib.h>
#include "rtwtypes.h"
#include "model_predict_ecg_types.h"

// Function Declarations
extern int32_T main(int32_T argc, const char * const argv[]);

#endif

//
// File trailer for main_ecg_jetson_ex.h
//
// [EOF]
//

```

GPU Code Generation — Specify Target

To create an executable that can be deployed to the target device, set `CodeGenMode` equal to 1. If you want to create an executable that runs locally and connects remotely to the target device, set `CodeGenMode` equal to 2.

The `main` function reads data from the text file specified by `signalFile` and writes the classification results to `resultFile`. Set `ExampleIndex` to choose a representative ECG signal. You will use this signal to test the executable against the `classify` function. `Jetson_BuildDir` specifies the directory for performing the remote build process on the target. If the specified build directory does not exist on the target, then the software creates a directory with the given name.

```

CodeGenMode =  ;
signalFile = 'signalData.txt';
resultFile = 'predClassProb.txt'; % consistent with "main_ecg_jetson_ex.cu"
Jetson_BuildDir = '~/projectECG';
ExampleIndex = 1; % 1,4: type ARR; 2,5: type CHF; 3,6: type NSR

Function_to_Gen = 'model_predict_ecg';
ModFile = 'ecg_model.mat'; % file that saves neural network model; consistent with "main_ecg_jet
ImgSize = [227 227]; % input image size for the ML model

switch ExampleIndex
    case 1 % ARR 7
        SampleSignalIdx = 7;
    case 2 % CHF 97
        SampleSignalIdx = 97;
    case 3 % NSR 132
        SampleSignalIdx = 132;
    case 4 % ARR 31
        SampleSignalIdx = 31;
    case 5 % CHF 101

```

```

        SampleSignalIdx = 101;
    case 6 % NSR 131
        SampleSignalIdx = 131;
end
signal_data = single(ECGData.Data(SampleSignalIdx,:));
ECGtype = ECGData.Labels{SampleSignalIdx};

```

GPU Code Generation — Connect to Hardware

To communicate with the NVIDIA hardware, you create a live hardware connection object using the `jetson` function. You must know the host name or IP address, user name, and password of the target board to create a live hardware connection object.

Create a live hardware connection object for the Jetson hardware. In the following code, replace:

- `NameOfJetsonDevice` with the name or IP address of your Jetson device
- `Username` with your user name
- `password` with your password

During the creation of the object, the software performs hardware and software checks, IO server installation, and gathers information on the peripherals connected to the target. This information is displayed in the command window.

```
hwobj = jetson("NameOfJetsonDevice","Username","password");
```

```

Checking for CUDA availability on the Target...
Checking for 'nvcc' in the target system path...
Checking for cuDNN library availability on the Target...
Checking for TensorRT library availability on the Target...
Checking for prerequisite libraries is complete.
Gathering hardware details...
Checking for third-party library availability on the Target...
Gathering hardware details is complete.
Board name           : NVIDIA Jetson Nano
CUDA Version         : 10.0
cuDNN Version        : 7.3
TensorRT Version     : 5.0
GStreamer Version    : 1.14.5
V4L2 Version         : 1.14.2-1
SDL Version          : 1.2
OpenCV Version       : 3.3.1
Available Webcams    :
Available GPUs       : NVIDIA Tegra X1
Available Digital Pins : 7 11 12 13 15 16 18 19 21 22 23 24 26 29 31 32 33 35

```

Use the `coder.checkGpuInstall` function and verify that the compilers and libraries needed for running this example are set up correctly on the hardware.

```

envCfg = coder.gpuEnvConfig('jetson');
envCfg.DeepLibTarget = 'cudnn';
envCfg.DeepCodegen = 1;
envCfg.HardwareObject = hwobj;
envCfg.Quiet = 1;
coder.checkGpuInstall(envCfg)

```

```

ans = struct with fields:
    gpu: 1

```

```

        cuda: 1
        cudnn: 1
        tensorrt: 0
        basiccodegen: 0
        basiccodeexec: 0
        deepcodegen: 1
        deepcodeexec: 0
        tensorrtdatatype: 0
        profiling: 0

```

GPU Code Generation — Compile

Create a GPU code configuration object necessary for compilation. Use the `coder.hardware` function to create a configuration object for the Jetson platform and assign it to the `Hardware` property of the code configuration object `cfg`. Use `'NVIDIA Jetson'` for the Jetson TX1 or TX2 boards. The custom main file is a wrapper that calls the entry-point function in the generated code. The custom file is required for a deployed executable.

Use the `coder.DeepLearningConfig` function to create a CuDNN deep learning configuration object and assign it to the `DeepLearningConfig` property of the GPU code configuration object. The code generator takes advantage of NVIDIA® CUDA® deep neural network library (cuDNN) for NVIDIA GPUs. cuDNN is a GPU-accelerated library of primitives for deep neural networks.

```

if CodeGenMode == 1
    cfg = coder.gpuConfig('exe');
    cfg.Hardware = coder.hardware('NVIDIA Jetson');
    cfg.Hardware.BuildDir = Jetson_BuildDir;
    cfg.DeepLearningConfig = coder.DeepLearningConfig('cudnn');
    cfg.CustomSource = fullfile('main_ecg_jetson_ex.cu');
elseif CodeGenMode == 2
    cfg = coder.gpuConfig('lib');
    cfg.VerificationMode = 'PIL';
    cfg.Hardware = coder.hardware('NVIDIA Jetson');
    cfg.Hardware.BuildDir = Jetson_BuildDir;
    cfg.DeepLearningConfig = coder.DeepLearningConfig('cudnn');
end

```

To generate CUDA code, use the `codegen` function and pass the GPU code configuration along with the size and type of the input for the `model_predict_ecg` entry-point function. After code generation on the host is complete, the generated files are copied over and built on the target.

```
codegen('-config ',cfg,Function_to_Gen,'-args',{signal_data},' -report');
```

```
Code generation successful: View report
```

GPU Code Generation — Execute

If you compiled an executable to be deployed to the target, write the example ECG signal to a text file. Use the `putFile()` function of the hardware object to place the text file on the target. The `workspaceDir` property contains the path to the `codegen` folder on the target.

```

if CodeGenMode == 1
    fid = fopen(signalFile,'w');
    for i = 1:length(signal_data)
        fprintf(fid,'%f\n',signal_data(i));
    end
end

```



```

    fclose(fid);
    hwobj.putFile(signalFile, hwobj.workspaceDir);
end

```

Run the executable.

When running the deployed executable, delete the previous result file if it exists. Use the `runApplication()` function to launch the executable on the target hardware, and then the `getFile()` function to retrieve the results. Because the results may not exist immediately after the `runApplication()` function call returns, and to allow for communication delays, set a maximum time for fetching the results to 90 seconds. Use the `evalc` function to suppress the command-line output.

```

if CodeGenMode == 1 % run deployed executable
    maxFetchTime = 90;
    resultFile_hw = fullfile(hwobj.workspaceDir, resultFile);
    if ispc
        resultFile_hw = strrep(resultFile_hw, '\', '/');
    end

    ta = tic;

    hwobj.deleteFile(resultFile_hw)
    evalc('hwobj.runApplication(Function_to_Gen, signalFile)');

    tf = tic;
    success = false;
    while toc(tf) < maxFetchTime
        try
            evalc('hwobj.getFile(resultFile_hw)');
            success = true;
        catch ME
        end
        if success
            break
        end
    end
    fprintf('Fetch time = %.3e sec\n', toc(tf));
    assert(success, 'Unable to fetch the prediction')
    PredClassProb = readmatrix(resultFile);
    PredTime = toc(ta);
elseif CodeGenMode == 2 % run PIL executable
    ta = tic;
    eval(sprintf('PredClassProb = %s_pil(signal_data);', Function_to_Gen));
    PredTime = toc(ta);
    eval(sprintf('clear %s_pil;', Function_to_Gen)); % terminate PIL execution
end

```

```
Fetch time = 1.658e+01 sec
```

Use the `classify` function to predict the class labels for the example signal.

```

ModData = load(ModFile, 'net');
im = cwt_ecg_jetson_ex(signal_data, ImgSize);
[ModPred, ModPredProb] = classify(ModData.net, im);
PredCat = categories(ModPred)';

```

Compare the results.

```
PredTableJetson = array2table(PredClassProb(:)', 'VariableNames', matlab.lang.makeValidName(PredCat)
fprintf('tPred = %.3e sec\nExample ECG Type: %s\n', PredTime, ECGtype)
```

```
tPred = 2.044e+01 sec
Example ECG Type: ARR
```

```
disp(PredTableJetson)
```

ARR	CHF	NSR
0.99858	0.001252	0.000166

```
PredTableMATLAB = array2table(ModPredProb(:)', 'VariableNames', matlab.lang.makeValidName(PredCat)
disp(PredTableMATLAB)
```

ARR	CHF	NSR
0.99858	0.0012516	0.00016613

Close the hardware connection.

```
clear hwobj
```

Summary

This example shows how to create and deploy a CUDA executable that uses a CNN to classify ECG signals. You also have the option to create an executable that runs locally and connects to the remote target. A complete workflow is presented in this example. After the data is downloaded, the CWT is used to extract features from the ECG signals. Then SqueezeNet is retrained to classify the signals based on their scalograms. Two user-defined functions are created and compiled on the target NVIDIA device. Results of the executable are compared with MATLAB.

References

- 1 Baim, D. S., W. S. Colucci, E. S. Monrad, H. S. Smith, R. F. Wright, A. Lanoue, D. F. Gauthier, B. J. Ransil, W. Grossman, and E. Braunwald. "Survival of patients with severe congestive heart failure treated with oral milrinone." *Journal of the American College of Cardiology*. Vol. 7, Number 3, 1986, pp. 661-670.
- 2 Goldberger A. L., L. A. N. Amaral, L. Glass, J. M. Hausdorff, P. Ch. Ivanov, R. G. Mark, J. E. Mietus, G. B. Moody, C.-K. Peng, and H. E. Stanley. "PhysioBank, PhysioToolkit, and PhysioNet: Components of a New Research Resource for Complex Physiologic Signals." *Circulation*. Vol. 101, Number 23: e215-e220. [Circulation Electronic Pages; <http://circ.ahajournals.org/content/101/23/e215.full>]; 2000 (June 13). doi: 10.1161/01.CIR.101.23.e215.
- 3 Moody, G. B., and R. G. Mark. "The impact of the MIT-BIH Arrhythmia Database." *IEEE Engineering in Medicine and Biology Magazine*. Vol. 20, Number 3, May-June 2001, pp. 45-50. (PMID: 11446209)

Supporting Functions

helperCreateECGDirectories

```
function helperCreateECGDirectories(ECGData, parentFolder, dataFolder)
% This function is only intended to support wavelet deep learning examples.
% It may change or be removed in a future release.
```

```

rootFolder = parentFolder;
localFolder = dataFolder;
mkdir(fullfile(rootFolder,localFolder))

folderLabels = unique(ECGData.Labels);
for i = 1:numel(folderLabels)
    mkdir(fullfile(rootFolder,localFolder,char(folderLabels(i))));
end
end

```

helperPlotReps

```

function helperPlotReps(ECGData)
% This function is only intended to support wavelet deep learning examples.
% It may change or be removed in a future release.

folderLabels = unique(ECGData.Labels);

for k=1:3
    ecgType = folderLabels{k};
    ind = find(ismember(ECGData.Labels,ecgType));
    subplot(3,1,k)
    plot(ECGData.Data(ind(1),1:1000));
    grid on
    title(ecgType)
end
end

```

helperCreateRGBfromTF

```

function helperCreateRGBfromTF(ECGData,parentFolder, childFolder)
% This function is only intended to support wavelet deep learning examples.
% It may change or be removed in a future release.

imageRoot = fullfile(parentFolder,childFolder);

data = ECGData.Data;
labels = ECGData.Labels;

[~,signalLength] = size(data);

fb = cwtfilterbank('SignalLength',signalLength,'VoicesPerOctave',12);
r = size(data,1);

for ii = 1:r
    cfs = abs(fb.wt(data(ii,:)));
    im = ind2rgb(im2uint8(rescale(cfs)),jet(128));

    imgLoc = fullfile(imageRoot,char(labels(ii)));
    imFileName = strcat(char(labels(ii)),'_',num2str(ii),'.jpg');
    imwrite(imresize(im,[227 227]),fullfile(imgLoc,imFileName));
end
end

```

See Also

Functions

codegen | coder.gpu.kernel | coder.gpu.kernelfun | coder.checkGpuInstall

Objects

`coder.gpuConfig` | `coder.CodeConfig` | `coder.EmbeddedCodeConfig` | `coder.gpuEnvConfig`

Related Examples

- “Kernels from Library Calls” on page 2-8
- “Design Patterns” on page 2-26
- “Kernels from Scatter-Gather Type Operations” on page 2-4

Troubleshooting

Three of the most common reasons why GPU Coder generated code is not performing as expected are:

- CUDA kernels are not created.
- Host to device and device to host memory transfers (`cudaMemcpy`) are throttling performance.
- Not enough parallelism or device issues.

Common causes for these symptoms and the process of using the built-in screener to detect these issues are discussed in the following topics. these topics also provide information on how to work around for these issues and generate more efficient CUDA code.

Workflow

- 1 GPU Coder relies on functionality provided by MATLAB Coder, so the first step in the troubleshooting process is to ensure that you have MATLAB Coder compatible code. To see programming requirements and best practices for MATLAB Coder, see “MATLAB Programming for Code Generation”.
- 2 GPU Coder has varying support for functions compatible with MATLAB Coder and Image Processing Toolbox. A list of the functions that have been tested with GPU Coder is provided in “MATLAB Algorithm Design for GPU”. These functions are categorized into ones that are fully supported, functions that are unsupported, and functions that are supported under certain conditions. For example, there are certain functions that work in vector-based operations but not when used within a loop body. It is however recommended where possible to rewrite the toolbox functions with pure MATLAB.
- 3 GPU Coder uses program parallelism analysis to detect parallel for loops. Traditional serial algorithms can vary significantly in how parallelizable they are. Some problems are embarrassingly parallel and are easy to divide up into pieces. On the other hand, some algorithms require some amount of refactoring to expose their inherent parallelism. The parallel analysis that GPU Coder performs is conservative. As a result there are cases where loops are truly parallel, but dependence analysis fails to detect the parallelism.
- 4 Loops must be statically bound to determine kernel dimensions. For example, while loops, loops with break statements and loops whose iteration range cannot be statically determinable are not easily mappable to CUDA kernels and have to be rewritten. Refer to the section on kernel analysis for more information.
- 5 After considering and rectifying these issues, you are now ready to generate CUDA code. The easiest way to accomplish code generation is to drop in the `pragma coder.gpu.kernel fun` in to the entry point function. You can then follow the steps described in “Get Started with GPU Coder” to generate CUDA code from either the command line or by using GPU Coder app.
- 6 To assess the performance of generated CUDA code, we can use MATLAB `tic` and `toc` functions and determine execution time. If the resulting GPU acceleration is not satisfactory, you can perform advance diagnostics like:
 - Kernel analysis
 - Memory bottleneck analysis
 - Analysis with NVIDIA Visual Profiler (`nvvp`) tool



See Also

More About

- “Code Generation Using the Command Line Interface”
- “Code Generation by Using the GPU Coder App”
- “Code Generation Reports” on page 6-5
- “Kernel Analysis” on page 6-18

- “Memory Bottleneck Analysis” on page 6-22
- “Analyze Execution Profiles of the Generated Code” on page 6-24

Code Generation Reports

In this section...
“Report Generation” on page 6-5
“Report Location” on page 6-6
“Errors and Warnings” on page 6-6
“Files and Functions” on page 6-6
“MATLAB Source” on page 6-6
“Generated Code” on page 6-8
“MATLAB Variables” on page 6-8
“Tracing Code” on page 6-9
“Code Insights” on page 6-10
“Additional Reports” on page 6-10
“Report Limitations” on page 6-10

GPU Coder produces a code generation report that helps you to:

- Debug code generation issues and verify that your MATLAB code is suitable for code generation.
- View generated CUDA code.
- Trace between MATLAB source code and generated CUDA code.
- See how the code generator determines and propagates type information for variables and expressions in your MATLAB code.
- Identify potential issues in the generated code.
- Access additional reports available with Embedded Coder.

Report Generation

When you enable report generation or when an error occurs, the code generator produces a code generation report. To control production and opening of a code generation report, use app settings, `codegen` options, or configuration object properties.

In the **GPU Coder** app:

- To generate a report, set **Always create a report** to Yes.
- If you want the app to open the report for you, set **Automatically launch a report if one is generated** to Yes.

At the command line, use `codegen` options:

- To generate a report, use the `-report` option.
- To generate and open a report, use the `-launchreport` option.

Alternatively, use the configuration object properties (`coder.CodeConfig`):

- To generate a report, set `GenerateReport` to `true`.
- If you want `codegen` to open the report for you, set `LaunchReport` to `true`.

Report Location

The code generation report is named `report.mldatx`. It is located in the `html` subfolder of the code generation output folder. If you have MATLAB R2018a or later, you can open the `report.mldatx` file by double-clicking it.

Errors and Warnings

View code generation error, warning, and information messages on the **All Messages** tab. To highlight the source code for an error or warning, click the message. It is a best practice to address the first message because subsequent errors and warnings can be related to the first message.

View compilation and linking errors and warnings on the **Build Logs** tab.

Files and Functions

The report lists MATLAB source functions and generated files. In the **MATLAB Source** pane, the **Function List** view organizes functions according to the containing file. To visualize functions according to the call structure, use the **Call Tree** view.

To view a function in the code pane of the report, click the function in the list. Clicking a function opens the file that contains the function. To edit the selected file in the MATLAB Editor, click **Edit in MATLAB** or click a line number in the code pane.

If you have Embedded Coder and generate the report with traceability enabled, to view the source code and generated code next to each other in the code pane, click **Trace Code**. You can interactively trace between the source code and the generated code. See “Interactively Trace Between MATLAB Code and Generated C/C++ Code” (Embedded Coder).

If you want to move the generated files for standalone code (library or executable) to another development environment, you can put them into a zip file by clicking **Package Code**.

Specialized Functions or Classes

When a function is called with different types of inputs or a class uses different types for its properties, the code generator produces specializations. In the **MATLAB Source** pane, numbered functions (or classes) indicate specializations. For example:

```
fx fcn > 1  
fx fcn > 2
```

MATLAB Source

To view a MATLAB function in the code pane, click the function in the **MATLAB Source** pane. To see information about the type of a variable or expression, pause over the variable or expression.

In the code pane, syntax highlighting of MATLAB source code helps you to identify MATLAB syntax elements. Syntax highlighting also helps you to identify certain code generation attributes such as whether a function is extrinsic or whether an argument is constant.

CUDA Kernels

The green **GPU** marker next to `mandelbrot_count` function indicates that the generated code has both CPU and GPU sections. The green vertical bar indicates the lines of code that are mapped to the GPU. To see information about the type of a variable or expression and the name of the corresponding **GPU Kernel Function**, pause over the variable or expression. When you select highlighted code by clicking it, the code becomes blue and you can see the information even when you move your pointer away from the selection. The code remains selected until you press **Esc** or select different code.

```

1 % getting started example (mandelbrot_count.m)
2 function count = mandelbrot_count(maxIterations, xGrid, yGrid) %#
3 % mandelbrot computation
4
5 z0 = xGrid + 1i*yGrid;
6 count = ones(size(z0));
7
8 % Map computation to GPU
9 coder.gpu.kernelfun;
10
11 z = z0;
12 for n = 0:maxIterations
13     z = z.*z + z0;
14     inside = abs(z)<=2;
15     count = count + inside;
16 end
17 count = log(count);
18

```

EXPRESSION INFO

abs(z)	
Size:	1000 × 1000
Class:	double
Complex:	No
GPU Kernel Function:	mandelbrot_count_kernel2

Extrinsic Functions

In the MATLAB code, the report identifies an extrinsic function with purple text. The information window indicates that the function is extrinsic.

```

Function: callMyExtrinsic
1 function z = callMyExtrinsic(a,b)
2 %#codegen
3 coder.extrinsic('myExtrinsic');
4 z = 0;
5 z = myExtrinsic(a,b);
6 disp(z);
7 end
8

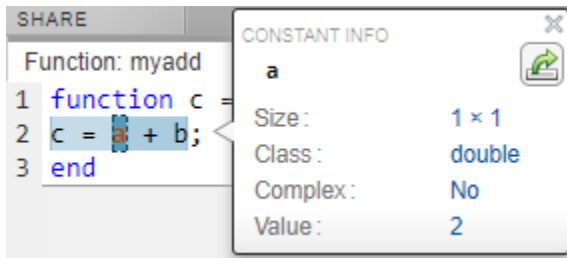
```

EXPRESSION INFO


myExtrinsic(a,b)	
Size:	1 × 1
Class:	mxArray
myExtrinsic is an extrinsic function.	

Constant Arguments

In the MATLAB code, orange text indicates a compile-time constant argument to an entry-point function or a specialized function. The information window includes the constant value.



Knowing the value of the constant arguments helps you to understand generated function signatures. It also helps you to see when code generation created function specializations for different constant argument values.

To export the value to a variable in the workspace, click .

Generated Code

To view a generated CUDA source or header file in the code pane, click the file in the **Files** tab on the **Generated Code** pane. The **GPU Kernels** tab on the **Generated Code** pane contains the list of CUDA kernels in the generated code. Click on the kernel name to navigate directly to the definition of the corresponding kernel in the generated code.

MATLAB Variables

The **Variables** tab provides information about the variables for the selected MATLAB function. To select a function, click the function in the **MATLAB Source** pane.

The variables table shows:

- Class, size, and complexity
- Properties of fixed-point types

This information helps you to debug errors, such as type mismatch errors, and to understand how the code generator propagates types and represents data in the generated code.

Visual Indicators on the Variables Tab




This table describes symbols, badges, and other indicators in the variables table.

Column in the Variables Table	Indicator	Description
Name	expander	Variable has elements or properties that you can see by clicking the expander.
Name	{:}	Heterogeneous cell array (all elements have the same properties)
Name	{n}	nth element of a heterogeneous cell array

Column in the Variables Table	Indicator	Description
Class	$v > n$	v is reused with a different class, size, and complexity. The number n identifies each unique reuse (a reuse with a unique set of properties). When you pause over a renamed variable, the report highlights only the instances of this variable that share the class, size, and complexity.
Size	:n	Variable-size dimension with an upper bound of n
Size	:?	Variable-size with no upper bound
Size	italics	Variable-size array whose dimensions do not change size during execution
Class	sparse prefix	Sparse array
Class	complex prefix	Complex number

Array Layout Indicators on the Variables Tab

This table describes the badges that indicate array layout in the variables table.

Badge	Description
	Row-major array layout.
	Column-major array layout.
	A mixture of row-major and column-major layouts.

See “Row-Major and Column-Major Array Layouts”.

Tracing Code

You can trace between MATLAB source code and generated CUDA code by using one of these methods:

- Interactively visualize the mapping between the MATLAB code and the generated code. To access interactive tracing, in the report, click **Trace Code**. The **Trace Code** button is enabled only if you have Embedded Coder and you enabled code traceability when you generated code. See “Interactively Trace Between MATLAB Code and Generated C/C++ Code” (Embedded Coder).
- Include source code as comments in the generated CUDA code. In a comment, the code generator produces a tag that helps you find the corresponding MATLAB source code. If you have Embedded Coder, the tag is a link to the source code. See “Trace Between Generated CUDA Code and MATLAB Source Code” on page 6-11.

Code Insights

The code generator can detect and report issues that can potentially occur in the generated code. View the messages on the **Code Insights** tab. The issues include:

- Potential differences between the behavior of the generated code and the behavior of the MATLAB code. The report includes potential differences messages only if you enabled potential differences reporting. See “Potential Differences Reporting”.
- GPU code generation diagnostics report that identifies issues during code generation and suggests potential solutions to maximize performance.
- Potential row-major issues. See “Code Design for Row-Major Array Layout”.

Additional Reports

The **Summary** tab can have links to these additional reports:

- GPU code metrics report. See “Generating a Static Code Metrics Report for Code Generated from MATLAB Code” (Embedded Coder).

Report Limitations

- The report does not show full information for unrolled loops. It displays data types of one arbitrary iteration.
- The report does not show information about dead code.

See Also

More About

- “Code Generation Using the Command Line Interface”
- “Code Generation by Using the GPU Coder App”
- “Generating a GPU Code Metrics Report for Code Generated from MATLAB Code” on page 6-15
- “Trace Between Generated CUDA Code and MATLAB Source Code” on page 6-11
- “Interactively Trace Between MATLAB Code and Generated C/C++ Code” (Embedded Coder)
- “Row-Major and Column-Major Array Layouts”

Trace Between Generated CUDA Code and MATLAB Source Code

This example shows how to trace (highlight sections) between MATLAB source code and the generated CUDA code. Tracing between source code and generated code helps you to:

- Understand how the code generator maps your algorithm to GPU kernels.
- Debug issues in the generated code.
- Evaluate the quality of the generated code.

You can trace by using one of these methods:

- Configure GPU Coder to generate code that includes the MATLAB source code as comments. In the comments, a traceability tag immediately precedes each line of source code. The traceability tag provides details about the location of the source code. If you have Embedded Coder, in the code generation report, the traceability tags link to the corresponding MATLAB source code.
- With Embedded Coder, produce a code generation report that includes interactive traceability. Interactive tracing in the report helps you to visualize the mapping between the MATLAB source code and the generated C/C++ code. See “Interactively Trace Between MATLAB Code and Generated C/C++ Code” (Embedded Coder).

Generate Traceability Tags

Create the MATLAB Source Code

To illustrate traceability tags, this example uses an implementation of the Mandelbrot set by using standard MATLAB commands running on the CPU. This implementation is based on the code provided in the Experiments with MATLAB e-book by Cleve Moler.

The Mandelbrot set is the region in the complex plane consisting of the values z_0 for which the trajectories defined by this equation remain bounded at $k \rightarrow \infty$.

$$z_{k+1} = z_k^2 + z_0, \quad k = 0, 1, \dots$$

Create a MATLAB function called `mandelbrot_count.m` with the following lines of code. This code is a vectorized MATLAB implementation of the Mandelbrot set. For every point $(xGrid, yGrid)$ in the grid, it calculates the iteration index count at which the trajectory defined by the equation reaches a distance of 2 from the origin. It then returns the natural logarithm of count, which is used generate the color coded plot of the Mandelbrot set.

```
function count = mandelbrot_count(maxIterations,xGrid,yGrid)
% Add kernelfun pragma to trigger kernel creation
coder.gpu.kernelfun;
% mandelbrot computation

z0 = xGrid + 1i*yGrid;
count = ones(size(z0));

z = z0;
for n = 0:maxIterations
    z = z.*z + z0;
    inside = abs(z)<=2;
```

```

        count = count + inside;
end
count = log(count);

```

Create Test Vectors

Create test vectors for the entry-point function by using the following lines of code. The script generates a 1000 x 1000 grid of real parts (x) and imaginary parts (y) between the limits specified by `xlim` and `ylim`. You can use these inputs to validate the `mandelbrot_count` entry-point function and plots the resulting Mandelbrot set.

```

maxIterations = 500;
gridSize = 1000;
xlim = [-0.748766713922161, -0.748766707771757];
ylim = [0.123640844894862, 0.123640851045266];

x = linspace(xlim(1),xlim(2),gridSize);
y = linspace(ylim(1),ylim(2),gridSize);
[xGrid,yGrid] = meshgrid(x,y);

```

Generate Traceability Tags

To produce traceability tags in the generated code, enable generation of MATLAB source code as comments.

- In the GPU Coder app, set **MATLAB source code as comments** to Yes.
- In a code generation configuration object, create a `coder.gpuConfig` object and set the `MATLABSourceComments` property to `true`.

```

cfg = coder.gpuConfig('dll','ecoder',true);
cfg.GenerateReport = true;
cfg.MATLABSourceComments = true;
cfg.GpuConfig.CompilerFlags = '--fmad=false';
codegen -config cfg -args {maxIterations,xGrid,yGrid} mandelbrot_count

```

Note The `--fmad=false` flag when passed to the `nvcc`, instructs the compiler to disable Floating-Point Multiply-Add (FMAD) optimization. This option is set to prevent numerical mismatch in the generated code because of architectural differences in the CPU and the GPU. For more information, see “Numerical Differences Between CPU and GPU”.

Access the Report

To open the code generation report, click **View report**.

The code generation report is named `report.mldatx`. It is located in the `html` subfolder of the code generation output folder. If you have MATLAB R2018a or later, you can open the `report.mldatx` file by double-clicking it.

In the **MATLAB Source** pane, select `mandelbrot_count.m`. You see the MATLAB source code in the code pane.

The screenshot shows the MATLAB IDE interface. The top bar is labeled 'REPORT'. Below it are navigation buttons: Back, Forward, Go To, Find, Trace Code, Edit In MATLAB, and Package Code. The main area is divided into three panes: 'MATLAB SOURCE', 'GENERATED CODE', and 'GPU Kernels'. The 'MATLAB SOURCE' pane shows the source code for 'mandelbrot_count.m'. A green 'GPU' marker is next to the function name. The source code is as follows:

```

1 % getting started example (mandelbrot_count.m)
2 function count = mandelbrot_count(maxIterations, xGrid, yGrid) %#codegen
3 % mandelbrot computation
4
5 z0 = xGrid + 1i*yGrid;
6 count = ones(size(z0));
7
8 % Map computation to GPU
9 coder.gpu.kernelfun;
10
11 z = z0;
12 for n = 0:maxIterations
13     z = z.*z + z0;
14     inside = abs(z)<=2;
15     count = count + inside;
16 end
17 count = log(count);
18

```

The 'EXPRESSION INFO' tooltip for 'abs(z)' shows:

- Size: 1000 x 1000
- Class: double
- Complex: No
- GPU Kernel Function: [mandelbrot_count_kernel2](#)

The 'GENERATED CODE' pane shows 'Files' and 'GPU Kernels'.

The green **GPU** marker next to `mandelbrot_count` function indicates that the generated code has both CPU and GPU sections. The green vertical bar indicates the lines of code that are mapped to the GPU. To see information about the type of a variable or expression and the name of the corresponding **GPU Kernel Function**, pause over the variable or expression. When you select highlighted code by clicking it, the code becomes blue and you can see the information even when you move your pointer away from the selection. The code remains selected until you press **Esc** or select different code.

To view the CUDA code generated for the `mandelbrot_count.m` entry-point function, select `mandelbrot_count.cu` from the **Generated Code** pane.

Format of Traceability Tags

In the generated code, traceability tags appear immediately before the MATLAB source code in the comment. The format of the tag is:
`<filename>:<line number>`.

For example, this comment indicates that the code `z0 = xGrid + 1i*yGrid;` appears at line 5 in the source file `mandelbrot_count.m`.

```
/* 'mandelbrot_count:5' z0 = xGrid + 1i*yGrid;
```

Traceability Tag Limitations

- You cannot include MATLAB source code as comments for:
 - MathWorks toolbox functions
 - P-code
- The appearance or location of comments can vary:
 - Even if the implementation code is eliminated, for example, due to constant folding, comments can still appear in the generated code.
 - If a complete function or code block is eliminated, comments can be eliminated from the generated code.
 - For certain optimizations, the comments can be separated from the generated code.
 - Even if you do not choose to include source code comments in the generated code, the generated code includes legally required comments from the MATLAB source code.
- Functions with multiple outputs do not get highlighted.
- Calls to `coder` functions such as `coder.nullcopy` will not be highlighted
- Code that gets mapped to library calls such as `cuDNN`, `cuBLAS` and `cuFFT` will not be highlighted. As a result, functions that are completely mapped to GPU may be tagged incorrectly.

See Also

`codegen` | `coder.gpuConfig` | `coder.CodeConfig` | `coder.EmbeddedCodeConfig`

Related Examples

- “Code Generation by Using the GPU Coder App”
- “Code Generation Using the Command Line Interface”
- “Code Generation Reports” on page 6-5
- “Generating a GPU Code Metrics Report for Code Generated from MATLAB Code” on page 6-15

Generating a GPU Code Metrics Report for Code Generated from MATLAB Code

The GPU static code metrics report contains the results of static analysis of the generated CUDA code, including information on the generated CUDA kernels, thread and block dimensions, memory usage and other statistics. To produce a static code metrics report, you must use GPU Coder to generate standalone CUDA code and produce a code generation report. See “Code Generation Reports” on page 6-5.

By default, static code metrics analysis does not run at code generation time. Instead, if and when you want to run the analysis and view the results, click **GPU Code Metrics** on the **Summary** tab of the code generation report.

Example GPU Code Metrics Report

This example runs GPU static code metrics analysis and examines a static code metrics report.

Create a MATLAB function called `mandelbrot_count.m` with the following lines of code. This code is a vectorized MATLAB implementation of the Mandelbrot set. For every point (`xGrid`, `yGrid`) in the grid, it calculates the iteration index `count` at which the trajectory defined by the equation reaches a distance of 2 from the origin. It then returns the natural logarithm of `count`, which is used generate the color coded plot of the Mandelbrot set.

```
function count = mandelbrot_count(maxIterations,xGrid,yGrid)
% Add kernelfun pragma to trigger kernel creation
coder.gpu.kernelfun;
% mandelbrot computation

z0 = xGrid + 1i*yGrid;
count = ones(size(z0));

z = z0;
for n = 0:maxIterations
    z = z.*z + z0;
    inside = abs(z)<=2;
    count = count + inside;
end
count = log(count);
```

Create sample data with the following lines of code. The code generates a 1000 x 1000 grid of real parts (`x`) and imaginary parts (`y`) between the limits specified by `xlim` and `ylim`.

```
maxIterations = 500;
gridSize = 1000;
xlim = [-0.748766713922161,-0.748766707771757];
ylim = [0.123640844894862,0.123640851045266];

x = linspace(xlim(1),xlim(2),gridSize);
y = linspace(ylim(1),ylim(2),gridSize);
[xGrid,yGrid] = meshgrid(x,y);
```

Enable production of a code generation report by using a configuration object for standalone code generation (static library, dynamically linked library, or executable program).

```
cfg = coder.gpuConfig('dll');
cfg.GenerateReport = true;
```

```
cfg.MATLABSourceComments = true;
cfg.GpuConfig.CompilerFlags = '--fmad=false';
```

Note The `--fmad=false` flag when passed to the `nvcc`, instructs the compiler to disable Floating-Point Multiply-Add (FMAD) optimization. This option is set to prevent numerical mismatch in the generated code because of architectural differences in the CPU and the GPU. For more information, see “Numerical Differences Between CPU and GPU”.

Alternatively, use the `codegen -report` option.

Generate code by using `codegen`. Specify the type of the input argument by providing an example input with the `-args` option. Specify the configuration object by using the `-config` option.

```
codegen -config cfg -args {maxIterations,xGrid,yGrid} mandelbrot_count
```

To open the code generation report, click **View report**.

To run the static code metrics analysis and view the code metrics report, on the **Summary** tab of the code generation report, click **GPU Code Metrics**.

Explore the code metrics report

- 1 To see the information on the generated CUDA kernels, click **CUDA Kernels**.

1. CUDA Kernels [\[hide\]](#)

Kernel Name	Thread Dimensions	Block Dimensions	Input Variables	Output Variables	Stream	Shared Memory Size	Minimum BlocksPerSM	Constant Memory	Parent Kernel
mandelbrot_count_kernel3	[512,1,1]	[1954,1,1]		gpu_count	0	0	1	0	None
mandelbrot_count_kernel2	[512,1,1]	[1954,1,1]	gpu_z0	gpu_count,gpu_z	0	0	1	0	None
mandelbrot_count_kernel1	[512,1,1]	[1954,1,1]	gpu_yGrid,gpu_xGrid	gpu_z,gpu_count,gpu_z0	0	0	1	0	None

- **Kernel Name** contains the list of generated CUDA kernels. By default, GPU Coder prepends the kernel name with the name of the entry-point function.
- **Thread Dimensions** is an array of the form $[Tx, Ty, Tz]$ that identifies the number of threads in the block along dimensions x , y , and z .
- **Block Dimensions** is an array of the form $[Bx, By, 1]$ is an array that defines the number of blocks in the grid along dimensions x and y (z not used).
- **Shared Memory Size** and **Constant Memory** columns provide metrics on the shared and constant memory space usage in the generated code.
- **Minimum BlocksPerSM** is the minimum number of blocks per streaming multiprocessor and indicates the number of blocks with which to launch the kernels.

To navigate from the report to the generated kernel code, click a kernel name.

- 2 To see the variables that have memory allocated on the GPU device, go to the **CUDA Malloc** section.

2. CUDA Malloc [\[hide\]](#)

Variable Name	Data Size
gpu_yGrid	8000000
gpu_xGrid	8000000
gpu_z	16000000
gpu_count	8000000
gpu_z0	16000000

3 To view information on the `cudaMemcpy` calls in the generated code, click **CUDA Memcpy**.

4. CUDA Memcpy [\[hide\]](#)

Destination Variable Name	Source Variable Name	Data Size	Direction	Conditional Variable	Stream
<code>count</code>	<code>gpu_count</code>	8000000	device->host	NO_ENCLOSING_CONDITION	0
<code>gpu_xGrid</code>	<code>xGrid</code>	8000000	host->device	NO_ENCLOSING_CONDITION	0
<code>gpu_yGrid</code>	<code>yGrid</code>	8000000	host->device	NO_ENCLOSING_CONDITION	0

Limitations

- If you have the Embedded Coder product, the code configuration object contains the `GenerateCodeMetricsReport` property to enable static metric report generation at compile time. GPU Coder does not honor this setting and has no effect during code generation.

See Also

`codegen` | `coder.gpuConfig` | `coder.CodeConfig` | `coder.EmbeddedCodeConfig`

More About

- “Code Generation Reports” on page 6-5
- “Interactively Trace Between MATLAB Code and Generated C/C++ Code” (Embedded Coder)
- “Trace Between Generated CUDA Code and MATLAB Source Code” on page 6-11
- “Code Generation Using the Command Line Interface”
- “Code Generation by Using the GPU Coder App”

Kernel Analysis

In this section...

“Mapping Nested Loops to Kernels” on page 6-18
 “For-Loops with Break” on page 6-19
 “Dependence Analysis Parallel Loop Check Fails” on page 6-19
 “Logical Indexing of Arrays” on page 6-20
 “Unsupported Functions” on page 6-20
 “Loop Interchange” on page 6-20

For GPU code generation, the primary mechanism for creating CUDA kernels is by using `for`-loops. The way you write loops in your MATLAB code has a significant impact on the number of kernels created as well as the performance of the generated code. When you generate GPU code, check the diagnostic report to see if your loop segment has `Loop not parallelized` notices. Calls to MATLAB functions in your code may also have `for`-loops that contain these notices. To get maximum performance, you want to ensure that compute intensive loop segments in your code are mapped to kernels and executed in parallel. The following recommendations help you in achieving this goal and generating efficient CUDA kernels.

Mapping Nested Loops to Kernels

Condition

Consider a function that has nested `for`-loops.

```
function y = foo(x)
...
for i1 = 1:N1
  for i2 = 1:N2
    for i3 = 1:N3
      for i4 = 1:N4
        ...
      end
    end
  end
end
```

Assume that one of the intermediate loop `i3` is not parallelizable. When performs loop analysis to create kernels, GPU Coder it considers only the outermost parallel loops `i1`, `i2` and creates a kernel with the outer loop dimensions `N1`, `N2`. The loops `i3`, `i4` are within the kernel body and are executed sequentially. However if the innermost `i4` is large (iteration), then better performance may be achieved by creating kernels for the innermost loop.

Action

There are three ways in which you can parallelize the innermost loop:

- Rewrite the code so that the innermost code segment is not within a nested loop.
- If the iteration size of the outer loop is small, then attach the loop to a `coder.unroll` function. This function unrolls the `for`-loop by making a copy of the loop body for each loop iteration. For more information, see `coder.unroll`.

```
function y = foo(x)
    ...
    for i1 = coder.unroll(1:N1)
        ...
    end
```

- Make the outer loop dimension as dynamic bound. This way parallel loop analysis fails on the outer loop, whereas it succeeds on the inner loops.

```
function y = foo(x,N1)
    ...
    for i1 = 1:N1
        ...
    end
```

For-Loops with Break

Condition

Loops with break are not supported.

```
while (i < N)
    ...
    ...
    if (cond2)
        ...
        ...
        break;
    end
end
```

Action

Remove breaks by creating a guard variable and conditional.

```
cond = true;
while (i < N)
    if(cond)
        ...
        ...
        if(cond2)
            cond = false;
        end
    end
end
```

Dependence Analysis Parallel Loop Check Fails

Condition

Kernel extraction use parallel loop dependence analysis. There are cases where loop dependence analysis cannot detect a parallel for loop. The `coder.gpu.kernel` allows GPU Coder to override dependence analysis and force kernel creation. The caveat is for user to be sure that the loop is “for-all” loop with no inter-iteration dependencies.

Action

Use `coder.gpu.kernel` pragma explicitly on each of your for-loops.

Logical Indexing of Arrays**Condition**

GPU Coder may not create kernels when logical indexing is used for accessing array elements.

```
i = (mag ~= 0);  
vx(i) = vx(i)./mag(i);  
vy(i) = vy(i)./mag(i);
```

Action

Rewrite the code by using a loop body and guarding with an appropriate conditional.

```
for i = 1:numel(mag)  
    if (mag(i) ~= 0)  
        vx(i) = vx(i)./mag(i);  
        vy(i) = vy(i)./mag(i);  
    end  
end
```

Unsupported Functions**Condition**

Use of unsupported functions, coder pragmas, toolbox functions etc. inside a loop prevents them from becoming a kernel.

Action

Try rewriting unsupported functions using pure MATLAB.

Loop Interchange**Condition**

If smaller loops in a loop nest are the outer most loops, then a kernel could be created with just a subset of the loops in the nesting. If algorithm allows it, always put the largest loops in the outermost nesting.

Action

Rewrite loop nesting with larger loops as outer loops.

See Also**More About**

- “Code Generation Using the Command Line Interface”

- “Code Generation by Using the GPU Coder App”
- “Code Generation Reports” on page 6-5
- “Trace Between Generated CUDA Code and MATLAB Source Code” on page 6-11
- “Generating a GPU Code Metrics Report for Code Generated from MATLAB Code” on page 6-15
- “Memory Bottleneck Analysis” on page 6-22
- “Analyze Execution Profiles of the Generated Code” on page 6-24

Memory Bottleneck Analysis

In this section...

“Data Alignment” on page 6-22

“Small Data Sizes” on page 6-22

“Too Many cudaMemcpys” on page 6-22

“Constant Inputs” on page 6-22

“Stack Memory Usage” on page 6-23

Data Alignment

Condition

MATLAB is column major but the algorithm could be implemented for an optimized row-major implementation. In the generated code, if your fastest changing dimension is not the innermost loop, then memory is not coalesced. Often, transposing the input matrices can simply fix this problem.

Action

Try transposing the data.

Small Data Sizes

Condition

If your problem/data size is too small, then the overhead of moving data to GPU (even if it is just at the I/O boundary) can offset any performance gains of running on the GPU.

Action

Try the algorithm with larger data sizes.

Too Many cudaMemcpys

Condition

If you use only `coder.gpu.kernel`, then everything outside the loop goes to the CPU. To try to keep most of the code on the GPU, use of both pragmas is recommended. Also, presence of unsupported functions or any function/statement that cannot run on the GPU, causes more `cudaMemcpys` to be generated.

Action

Use `coder.gpu.kernelfun` in addition to `coder.gpu.kernel`

Constant Inputs

Recommendation

If certain inputs of your entry-point function are constant, wrap them using the `coder.const` object. Use of `coder.const` object indicates that these variables are constant during code generation.

Without this function, GPU Coder considers these inputs to be variables and hence treats all matrices sized by these variables as variable-dimension matrices. GPU Coder does not create good kernels out of variable-dimension matrices since currently there is no support for dynamic sizing of kernels or dynamic `cudaMemcpy` function calls.

Stack Memory Usage

Recommendation

Using large stack memory inside kernels can reduce the performance of the generated code. Under such conditions consider rewriting the algorithm in a different fashion or breaking it into smaller computations to reduce stack memory usage and improve performance.

See Also

More About

- “Code Generation Using the Command Line Interface”
- “Code Generation by Using the GPU Coder App”
- “Code Generation Reports” on page 6-5
- “Trace Between Generated CUDA Code and MATLAB Source Code” on page 6-11
- “Generating a GPU Code Metrics Report for Code Generated from MATLAB Code” on page 6-15
- “Kernel Analysis” on page 6-18
- “Analyze Execution Profiles of the Generated Code” on page 6-24

Analyze Execution Profiles of the Generated Code

This example shows you how to perform fine grain analysis for a MATLAB algorithm and its generated CUDA code through software-in-the-loop (SIL) execution profiling. The Embedded Coder product must be installed to generate the execution profiling report.

Note The profiling workflow depends on the `nvprof` tool from NVIDIA. In CUDA Toolkit v10.1, NVIDIA restricts access to performance counters to only admin users. To enable GPU performance counters to be used by all users, see the instructions provided in [Permission issue with Performance Counters \(NVIDIA\)](#).

Create a Design File

For this example create an entry-point function that performs N-D fast Fourier transform. Use the `coder.gpu.kernelfun` pragma to map the FFT to the GPU. By default, the `EnableCUFFT` property is enabled, so the code generator uses `cuFFT` library to perform the FFT operation.

```
function [Y] = gpu_fftn(X)
    coder.gpu.kernelfun();
    Y = fftn(X);
end
```

Generate the Execution Profiling Report

Use the `gpucoder.profile` function to generate the execution profiling report.

```
cfg = coder.gpuConfig('exe');
cfg.GpuConfig.MallocMode = 'discrete';
gpucoder.profile('gpu_fftn',{rand(2,4500,4)},'CodegenConfig',cfg, ...
'CodegenArguments','-d profilingdir','Threshold',0.001)
```

The code execution profiling report opens. This report provides metrics based on data collected from a SIL execution. Execution times are calculated from data recorded by instrumentation probes added to the SIL test harness or inside the code generated for each component. See “View Execution Times” (Embedded Coder) for more information.

Code Execution Profiling Report for gpu_fftn

The code execution profiling report provides metrics based on data collected from a SIL or PIL execution. Execution times are calculated from data recorded by instrumentation probes added to the SIL or PIL test harness or inside the code generated for each component. See [Code Execution Profiling](#) for more information.

1. Summary

Total time	4134.12113
Unit of time	ms
Command	report(etStruct, 'Units', 'seconds', 'ScaleFactor', '0.001', 'NumericFormat', '%5.5f');
Timer frequency (ticks per second)	1.31741e+09
Profiling data created	20-Jul-2018 16:15:26

2. Profiled Sections of Code

Section	Maximum Execution Time in ms	Average Execution Time in ms	Maximum Self Time in ms	Average Self Time in ms	Calls	
gpu_fftn_initialize	0.01087	0.01087	0.01087	0.01087	1	
gpu_fftn	4064.92221	689.01660	4064.92221	689.01660	6	
gpu_fftn_terminate	0.01068	0.01068	0.01068	0.01068	1	

3. GPU Profiling Trace for gpu_fftn

Name	Duration in ms
cudaMalloc	0.3324
cudaMalloc	0.0201
cudaMalloc	0.0160
cudaMalloc	0.2647
cudaMalloc	0.0177
cudaMalloc	0.0154
cudaGetDeviceProperties	0.7831
cudaGetDeviceProperties	0.5001
cudaMalloc	0.2998
cudaMalloc	0.0306

OK

Help

See Also

`gpucoder.profile` | `codegen` | `coder.EmbeddedCodeConfig`

More About

- “Code Generation Using the Command Line Interface”
- “Code Generation by Using the GPU Coder App”
- “Code Generation Reports” on page 6-5
- “Trace Between Generated CUDA Code and MATLAB Source Code” on page 6-11

- “Generating a GPU Code Metrics Report for Code Generated from MATLAB Code” on page 6-15

Analysis with NVIDIA Profiler

In this section...
“Not Enough Parallelism” on page 6-27
“Too Many Local per-Thread Registers” on page 6-27

Not Enough Parallelism

Condition

If the kernel is doing little work, then the overhead of memcpy and kernel launches can offset any performance gains. Consider working on a larger sample set (thus increasing the loop size). To detect this condition, look at the nvvpreport.

Action

Do more work in the loop or increase sample set size

Too Many Local per-Thread Registers

Condition

If there are too many local/temp variables used in the loop body, then it causes high register pressure in the per-thread register file. You can detect this condition by running in GPU safe-build mode. Or, nvvp reports this fact.

Action

Consider using different block sizes in `coder.gpu.kernel pragma`.

See Also

More About

- “Code Generation Using the Command Line Interface”
- “Code Generation by Using the GPU Coder App”
- “Code Generation Reports” on page 6-5
- “Trace Between Generated CUDA Code and MATLAB Source Code” on page 6-11
- “Generating a GPU Code Metrics Report for Code Generated from MATLAB Code” on page 6-15
- “Kernel Analysis” on page 6-18
- “Memory Bottleneck Analysis” on page 6-22
- “Analyze Execution Profiles of the Generated Code” on page 6-24

GPU Coder Limitations

General Limitations

- Spaces in file and path names cause build errors in Linux. GPU Coder uses GNU make tools that have known limitations when file names contain spaces. It is generally a good practice to avoid spaces in file, project, and path names.
- GPU Coder disables integrity and array bounds/dimension checks that are part of MATLAB Coder.
- When using `coder.inline('never')` option during code generation, GPU Coder creates kernel for only the entry-point function containing the `coder.gpu.kernelfun` pragma and does not create kernels automatically for any sub-functions within the entry-point function. It is therefore recommended not to use the `coder.inline('never')` option.
- Generating kernels for structures with variable-size arrays is not supported.
- The CUDA compute capability that you select must match the compute capability of your hardware.
- When using `coder.ceval` with GPU pointers, the **Check for Issues** option for **CPU** is not supported.
- GPU Coder does not support code generation for Simulink blocks. You cannot use the NVIDIA Jetson and NVIDIA Drive boards from the **Hardware board** option in the **Hardware Implementation** pane and target NVIDIA GPUs.
- GPU Coder does not support SIMD code generation. Disable SIMD code generation by setting the **Leverage target hardware instruction set extensions** parameter to None.

Function Limitations

- You can generate CUDA code for only a subset of MATLAB built-in functions and toolbox functions.
- When targeting NVIDIA Tegra devices, GPU Coder does not support the *quasi-euclidean* method of `bwdist` function and image dimensions greater than 3.
- When `imfilter` is used with a $1 \times N$ kernel and N is an even integer, shared memory is not used in generated code. When `imfilter` is used with a three-dimensional image, shared memory is not used in the `conv2` implementation.
- GPU Coder has empty code replacement report even if there is a replacement. This issue has been identified with `atan` function.

Unsupported CUDA Features

List of CUDA features that are not supported:

- Texture memory
- Asynchronous streams
- Dynamic kernel invocation — calling kernels from within kernels

See Also

More About

- “Code Generation Using the Command Line Interface”
- “Code Generation by Using the GPU Coder App”
- “Code Generation Reports” on page 6-5
- “Trace Between Generated CUDA Code and MATLAB Source Code” on page 6-11
- “Generating a GPU Code Metrics Report for Code Generated from MATLAB Code” on page 6-15
- “Kernel Analysis” on page 6-18
- “Memory Bottleneck Analysis” on page 6-22
- “Analyze Execution Profiles of the Generated Code” on page 6-24

GPU Execution Profiling of the Generated Code

This example shows you how to generate an execution profiling report for the generated CUDA® code by using the `gpuCoder.profile` function.

The GPU Coder profiler runs a software-in-the-loop (SIL) execution that produces execution-time metrics for the tasks and kernels in the generated code. This example generates an execution profiling report for the *Fog Rectification* example from GPU Coder. For more information, see “Fog Rectification” on page 2-80.

Third-Party Prerequisites

- CUDA enabled NVIDIA® GPU.
- NVIDIA CUDA toolkit and driver.
- NVIDIA Nsight™ Systems. For information on the supported versions of the compilers and libraries, see “Third-Party Hardware”.
- Environment variables for the compilers and libraries. For setting up the environment variables, see “Setting Up the Prerequisite Products”.
- The profiling workflow of this example depends on the profiling tools from NVIDIA that accesses GPU performance counters. From CUDA toolkit v10.1, NVIDIA restricts access to performance counters to only admin users. To enable GPU performance counters to be used by all users, see the instructions provided in Permission issue with Performance Counters (NVIDIA).

Verify GPU Environment

To verify that the compilers and libraries necessary for running this example are set up correctly, use the `coder.checkGpuInstall` function.

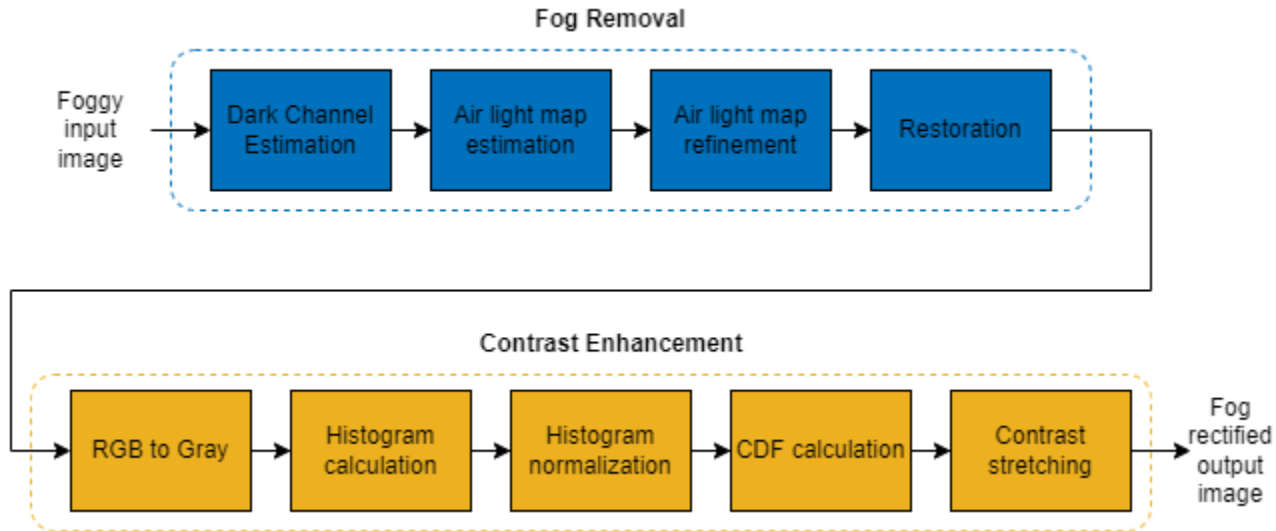
```
envCfg = coder.gpuEnvConfig('host');  
envCfg.BasicCodegen = 1;  
envCfg.Quiet = 1;  
coder.checkGpuInstall(envCfg);
```

Fog Rectification Algorithm

To improve the foggy input image, the algorithm performs fog removal and then contrast enhancement. The diagram shows the steps of both these operations.

This example takes a foggy RGB image as input. To perform fog removal, the algorithm estimates the dark channel of the image, calculates the airlight map based on the dark channel, and refines the airlight map by using filters. The restoration stage creates a defogged image by subtracting the refined airlight map from the input image.

Then, the Contrast Enhancement stage assesses the range of intensity values in the image and uses contrast stretching to expand the range of values and make features stand out more clearly.



```
type fog_rectification.m
```

```
function [out] = fog_rectification(input) %#codegen
% Copyright 2017-2019 The MathWorks, Inc.
coder.gpu.kernelFun;
% restoreOut is used to store the output of restoration
restoreOut = zeros(size(input),'double');
% Changing the precision level of input image to double
input = double(input)./255;
%% Dark channel Estimation from input
darkChannel = min(input,[],3);
% diff_im is used as input and output variable for anisotropic diffusion
diff_im = 0.9*darkChannel;
num_iter = 3;
% 2D convolution mask for Anisotropic diffusion
hN = [0.0625 0.1250 0.0625; 0.1250 0.2500 0.1250; 0.0625 0.1250 0.0625];
hN = double(hN);
%% Refine dark channel using Anisotropic diffusion.
for t = 1:num_iter
    diff_im = conv2(diff_im,hN,'same');
end
%% Reduction with min
diff_im = min(darkChannel,diff_im);
diff_im = 0.6*diff_im ;
%% Parallel element-wise math to compute
% Restoration with inverse Koschmieder's law
factor = 1.0./(1.0-(diff_im));
```

```

restoreOut(:,:,1) = (input(:,:,1)-diff_im).*factor;
restoreOut(:,:,2) = (input(:,:,2)-diff_im).*factor;
restoreOut(:,:,3) = (input(:,:,3)-diff_im).*factor;
restoreOut = uint8(255.*restoreOut);
restoreOut = uint8(restoreOut);

%%
% Stretching performs the histogram stretching of the image.
% im is the input color image and p is cdf limit.
% out is the contrast stretched image and cdf is the cumulative prob.
% density function and T is the stretching function.

p = 5;
% RGB to grayscale conversion
im_gray = im2gray(restoreOut);
[row,col] = size(im_gray);

% histogram calculation
[count,~] = imhist(im_gray);
prob = count'/(row*col);

% cumulative Sum calculation
cdf = cumsum(prob(:));

% finding less than particular probability
i1 = length(find(cdf <= (p/100)));
i2 = 255-length(find(cdf >= 1-(p/100)));

o1 = floor(255*.10);
o2 = floor(255*.90);

t1 = (o1/i1)*[0:i1];
t2 = (((o2-o1)/(i2-i1))*[i1+1:i2])-(((o2-o1)/(i2-i1))*i1)+o1;
t3 = (((255-o2)/(255-i2))*[i2+1:255])-(((255-o2)/(255-i2))*i2)+o2;

T = (floor([t1 t2 t3]));

restoreOut(restoreOut == 0) = 1;

u1 = (restoreOut(:,:,1));
u2 = (restoreOut(:,:,2));
u3 = (restoreOut(:,:,3));

% Replacing the value from look up table
out1 = T(u1);
out2 = T(u2);
out3 = T(u3);

out = zeros([size(out1),3], 'uint8');
out(:,:,1) = uint8(out1);
out(:,:,2) = uint8(out2);
out(:,:,3) = uint8(out3);
return

```

Generate Execution Profiling Report

To generate an execution profiling report, create a code configuration object with a dynamic library ('dll') build type. Because the `gpuCoder.profile` function accepts only an Embedded Coder™

configuration object, enable the option to create a `coder.EmbeddedCodeConfig` configuration object.

```
cfg = coder.gpuConfig('dll', 'ecoder', true);
cfg.GpuConfig.MallocMode = 'discrete';
```

Run `gpucoder.profile` with the default threshold value of zero seconds. If the generated code has a lot of CUDA API or kernel calls, it is likely that each call constitutes only a small proportion of the total time. In such cases, set a low (non-zero) threshold value to generate a meaningful profiling report. It is not advisable to set number of executions value to a very low number (less than 5) because it does not produce an accurate representation of a typical execution profile.

```
inputImage = imread('foggyInput.png');
inputs = {inputImage};
designFileName = 'fog_rectification';

gpucoder.profile(designFileName, inputs, ...
    'CodegenConfig', cfg, 'Threshold', 0, 'NumCalls', 10);
```

Code generation successful: View report

```
### Starting SIL execution for 'fog_rectification'
To terminate execution: clear fog_rectification_sil
Execution profiling data is available for viewing. Open Simulation Data Inspector.
Execution profiling report available after termination.
```

```
### Host application produced the following standard error (stderr) messages:
```

```
Warning: LBR backtrace method is not supported on this platform. DWARF backtrace method will be used.
Collecting data...
```

```
### Stopping SIL execution for 'fog_rectification'
```

Code Execution Profiling Report for the fog_rectification Function

The code execution profiling report provides metrics based on data collected from a SIL execution. Execution times are calculated from data recorded by instrumentation probes added to the SIL or PIL test harness or inside the code generated for each component. For more information, see “View Execution Times” (Embedded Coder).

These numbers are representative. The actual values depend on your hardware setup. This profiling was done using MATLAB R2022b on a machine with an 6 core, 3.5GHz Intel® Xeon® CPU, and an NVIDIA TITAN XP GPU

Summary








This section gives information about the creation of the report.




Total time	208.63571
Unit of time	ms
Command	report(etStruct, 'Units', 'seconds', 'ScaleFactor', '0.001', 'NumericFormat', '%5.5F');
Timer frequency (ticks per second)	3.8e+09
Profiling data created	09-Jun-2022 15:04:37

Profiled Sections of Code

This section contains information about profiled code sections. The report contains time measurements for:

- The `entry_point_fn_initialize` function, for example, `fog_rectification_initialize`.
- The entry-point function, for example, `fog_rectification`.
- The `entry_point_fn_terminate` function, for example, `fog_rectification_terminate`.

Section	Maximum Execution Time in ms	Average Execution Time in ms	Maximum Self Time in ms	Average Self Time in ms	Calls	
fog_rectification_initialize	34.34152	34.34152	34.34152	34.34152	1	 
fog_rectification	149.55285	17.42718	149.55285	17.42718	10	  
fog_rectification_terminate	0.02240	0.02240	0.02240	0.02240	1	 

- The section column lists the names of the function from which code is generated.
- Maximum execution time is the longest time between start and end of code section.
- Average Execution Time is the average time between start and end of code section.
- Maximum Self Time is the maximum execution time, excluding time in child sections.
- Average Self Time is the average execution time, excluding time in child sections.
- Calls indicate the number of calls to the code section.
- To view execution-time metrics for a code section in the Command Window, on the corresponding row, click the icon .
- To display measured execution times, click the Simulation Data Inspector icon . You can use the Simulation Data Inspector to manage and compare plots from various executions.
- To display the execution-time distribution, click the icon .

By default, the report displays time in milliseconds (10^{-3} seconds). You can specify the time unit and numeric display format. For example, to display time in microseconds (10^{-6} seconds), use the `report` (Embedded Coder) command:

```
executionProfile=getCoderExecutionProfile('fog_rectification');
report(executionProfile, ...
    'Units', 'Seconds', ...
    'ScaleFactor', '1e-06', ...
    'NumericFormat', '%0.3f')

ans =
'/local-ssd/lnarasim/MATLAB/ExampleManager/lnarasim.Bdoc22b.j1984243/gpuCoder-ex87489778/codegen,
```

The report displays time in seconds only if the timer is calibrated, that is, the number of timer ticks per second is known. On a Windows® machine, the software determines this value for a SIL simulation. On a Linux® machine, you must manually calibrate the timer. For example, if your processor speed is 3.5 GHz, specify the number of timer ticks per second:

```
executionProfile.TimerTicksPerSecond = 3.5e9;
```

Execution Times in Percentages

This section provides function execution times as percentages of caller function and total execution times, which can help you to identify performance bottlenecks in generated code.

Section	Self Time / Caller Function	Self Time / Task	Self Time / Simulation	Function / Simulation
fog_rectification_initialize	100%	100%	16.46%	16.46%
fog_rectification	100%	100%	83.529%	83.529%
fog_rectification_terminate	100%	100%	0.010738%	0.010738%

GPU Profiling Trace for fog_rectification

Section 4 shows the complete trace of GPU calls that have a runtime higher than the threshold value. A snippet of the profiling trace is shown.

cudaMalloc	0.0739
cudaMalloc	0.0041
cudaMemcpy	0.0863
cudaLaunchKernel	0.0211
cudaLaunchKernel	0.0056
cudaLaunchKernel	0.0051
cudaLaunchKernel	0.0044
cudaMemcpyToSymbol	0.1402
fog_rectification_kernel1	0.0464
fog_rectification_kernel2	0.0297
fog_rectification_kernel3	0.0065
fog_rectification_kernel4	0.0131
cudaLaunchKernel	0.0048
cudaLaunchKernel	0.0074
fog_rectification_kernel5	0.0370
cudaLaunchKernel	0.0034
cudaMemcpyToSymbol	0.0508
fog_rectification_kernel3	0.0065
fog_rectification_kernel4	0.0132
cudaLaunchKernel	0.0041

GPU Profiling Summary for fog_rectification

Section 5 in the report shows the summary of GPU calls that are shown in section 4. The `cudaFree` is called 15 times per run of `fog_rectification` and the average time taken by 15 calls of `cudaFree` over 9 runs of `fog_rectification` is 1.3790 milliseconds. This summary is sorted in descending order of time taken to give the users an idea which GPU call is taking the maximum time.

Name	Total Average Time in ms	Number of Calls per Iteration
<code>cudaFree</code>	1.3790	15
<code>cudaMalloc</code>	0.6327	15
<code>cudaMemcpy</code>	0.3588	3
<code>cudaMemcpyToSymbol</code>	0.2444	3
<code>cudaLaunchKernel</code>	0.1584	29
<code>fog_rectification_kernel5</code>	0.1100	3
<code>fog_rectification_kernel1...</code>	0.0503	1
<code>fog_rectification_kernel1</code>	0.0463	1
<code>fog_rectification_kernel7</code>	0.0460	1
<code>fog_rectification_kernel8</code>	0.0424	1
<code>fog_rectification_kernel4</code>	0.0394	3
<code>imhist_sm</code>	0.0379	1
<code>fog_rectification_kernel2</code>	0.0295	1
<code>fog_rectification_kernel6</code>	0.0255	1
<code>fog_rectification_kernel2...</code>	0.0239	1
<code>fog_rectification_kernel3</code>	0.0198	3
<code>fog_rectification_kernel9</code>	0.0194	1
<code>fog_rectification_kernel1...</code>	0.0142	1
<code>_kernel_agent</code>	0.0093	2
<code>fog_rectification_kernel1...</code>	0.0037	1
<code>fog_rectification_kernel1...</code>	0.0026	1
<code>fog_rectification_kernel1...</code>	0.0025	1

Definitions

This section provides descriptions of some metrics.

Execution Time: Time between start and end of code section.

Self Time: Execution time, excluding time in child sections.

Troubleshooting CUDA Errors

Register Count nvlink Error

Issue

The NVIDIA compiler (nvcc) may fail in the linking stage due to a mismatch in the maximum number of registers computed at compilation. A generated CUDA kernel is compiled with a max register count decided by NVCC. A kernel may call a device function in a different CUDA file. The device function may be compiled to use a larger number of registers exceeding the max register count of the kernel.

You may encounter an error message with the following pattern:

```
nvlink error : entry function 'xxx' with max regcount of n calls function 'yyy' with regcount of m.
```

```
nvlink error : entry function 'xx' with max  
regcount of n calls function 'yyy' with  
regcount of m.
```

where 'xxx' and 'yyy' are the mangled function names, n and m are integers, and m is larger than n.

Possible Solutions

Use the '-maxrregcount n' compiler flag of NVCC to specify the maximum amount of registers. Use the compiler flags option in the GPU code configuration parameters to pass compiler flags to NVCC. For example,

```
cfg = coder.gpuConfig;  
cfg.GpuConfig.compilerFlags = '-maxrregcount n';
```

where n is the smallest integer number of register count in the error message thrown by nvlink.

See Also