

**GPU Coder™**

User's Guide



**MATLAB®**

R2021a



## How to Contact MathWorks



Latest news: [www.mathworks.com](http://www.mathworks.com)  
Sales and services: [www.mathworks.com/sales\\_and\\_services](http://www.mathworks.com/sales_and_services)  
User community: [www.mathworks.com/matlabcentral](http://www.mathworks.com/matlabcentral)  
Technical support: [www.mathworks.com/support/contact\\_us](http://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 2017–2021 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](http://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](http://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)

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

---

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

**2** | **Kernel Creation from MATLAB Code**

---

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

<b>Legacy Code Integration</b> .....	<b>2-18</b>
coder.ceval for GPU Coder .....	2-18
Legacy Code Example .....	2-18
Generate CUDA Code .....	2-20
Generated Code .....	2-20
<b>Design Patterns</b> .....	<b>2-22</b>
Stencil Processing .....	2-22
Matrix-Matrix Processing .....	2-22
<b>GPU Memory Allocation and Minimization</b> .....	<b>2-24</b>
Discrete and Managed Modes .....	2-24
Memory Minimization .....	2-24
<b>Support for GPU Arrays</b> .....	<b>2-26</b>
Considerations .....	2-26
<b>Simulate Diffraction Patterns Using CUDA FFT Libraries</b> .....	<b>2-28</b>
<b>Benchmark A\b by Using GPU Coder</b> .....	<b>2-34</b>
<b>QR Decomposition on NVIDIA GPU Using cuSOLVER Libraries</b> .....	<b>2-42</b>
<b>Stencil Processing on GPU</b> .....	<b>2-47</b>
<b>Fog Rectification</b> .....	<b>2-53</b>
<b>Stereo Disparity</b> .....	<b>2-58</b>
<b>Feature Extraction Using SURF</b> .....	<b>2-64</b>
<b>Feature Matching</b> .....	<b>2-68</b>
<b>Lane Detection on the GPU by Using the houghlines Function</b> .....	<b>2-72</b>
<b>Edge Detection with Sobel Method in Half-Precision</b> .....	<b>2-75</b>
<b>Deploy Signal Classifier on NVIDIA Jetson Using Wavelet Analysis and Deep Learning</b> .....	<b>2-79</b>

## Kernel Creation from Simulink Models

### 3

<b>Simulation Acceleration by Using GPU Coder</b> .....	<b>3-2</b>
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

<b>Code Generation from Simulink Models with GPU Coder</b> .....	<b>3-8</b>
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
<b>Deep Learning in Simulink by Using MATLAB Function Block</b> .....	<b>3-14</b>
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 .....	3-20
Limitations .....	3-21
<b>Deep Learning in Simulink by Using Deep Neural Networks Library</b> ...	<b>3-22</b>
Example: Classify Images by Using GoogLeNet .....	3-22
Create GoogLeNet Model .....	3-23
Configure the Model for GPU Acceleration .....	3-24
Build GPU Accelerated Model .....	3-25
Configure Model for Code Generation .....	3-26
Generate CUDA Code for the Model .....	3-27
Limitations .....	3-28
<b>Targeting NVIDIA Embedded Boards</b> .....	<b>3-29</b>
Configure Model for Deployment .....	3-29
Generate CUDA Code for the Model .....	3-29
<b>Numerical Equivalence Testing</b> .....	<b>3-31</b>
Target Connectivity Configuration for PIL .....	3-31
Example: The Mandelbrot Set .....	3-32
GPU Acceleration or PIL Simulation with a Top Model .....	3-33
Run Normal and PIL Simulations .....	3-34
Limitations .....	3-36
<b>Parameter Tuning and Signal Monitoring by Using External Mode</b> ....	<b>3-37</b>
Example: The Mandelbrot Set .....	3-38
Create Mandelbrot Model .....	3-39
Build Target Executable .....	3-39
Run Target Application .....	3-40
Stop Target Application .....	3-41
<b>GPU Code Generation for Lane Detection in Simulink</b> .....	<b>3-42</b>
<b>GPU Code Generation for a Fog Rectification Simulink Model</b> .....	<b>3-47</b>
<b>Code Generation for a Deep Learning Simulink Model to Classify ECG     Signals</b> .....	<b>3-50</b>
<b>Code Generation for a Deep Learning Simulink Model that Performs Lane     and Vehicle Detection</b> .....	<b>3-57</b>

<b>Workflow</b> .....	<b>4-2</b>
<b>Code Generation Reports</b> .....	<b>4-5</b>
Report Generation .....	4-5
Report Location .....	4-6
Errors and Warnings .....	4-6
Files and Functions .....	4-6
MATLAB Source .....	4-6
Generated Code .....	4-8
MATLAB Variables .....	4-8
Tracing Code .....	4-9
Code Insights .....	4-10
Additional Reports .....	4-10
Report Limitations .....	4-10
<b>Trace Between Generated CUDA Code and MATLAB Source Code</b> .....	<b>4-11</b>
Generate Traceability Tags .....	4-11
Format of Traceability Tags .....	4-13
Traceability Tag Limitations .....	4-14
<b>Generating a GPU Code Metrics Report for Code Generated from MATLAB Code</b> .....	<b>4-15</b>
Example GPU Code Metrics Report .....	4-15
Explore the code metrics report .....	4-16
Limitations .....	4-17
<b>Kernel Analysis</b> .....	<b>4-18</b>
Mapping Nested Loops to Kernels .....	4-18
For-Loops with Break .....	4-19
Dependence Analysis Parallel Loop Check Fails .....	4-19
Logical Indexing of Arrays .....	4-20
Unsupported Functions .....	4-20
Loop Interchange .....	4-20
<b>Memory Bottleneck Analysis</b> .....	<b>4-22</b>
Data Alignment .....	4-22
Small Data Sizes .....	4-22
Too Many cudaMemcpy .....	4-22
Constant Inputs .....	4-22
Stack Memory Usage .....	4-23
<b>Analyze Execution Profiles of the Generated Code</b> .....	<b>4-24</b>
Create a Design File .....	4-24
Generate the Execution Profiling Report .....	4-24
<b>Analysis with NVIDIA Profiler</b> .....	<b>4-27</b>
Not Enough Parallelism .....	4-27
Too Many Local per-Thread Registers .....	4-27
<b>GPU Coder Limitations</b> .....	<b>4-28</b>
General Limitations .....	4-28

Function Limitations .....	4-28
Unsupported CUDA Features .....	4-28
<b>GPU Execution Profiling of the Generated Code .....</b>	<b>4-30</b>

## Deep Learning

# 5

<b>Workflow .....</b>	<b>5-2</b>
<b>Supported Networks, Layers, and Classes .....</b>	<b>5-5</b>
Supported Pretrained Networks .....	5-5
Supported Layers .....	5-10
Supported Classes .....	5-27
<b>Code Generation for dlarray .....</b>	<b>5-35</b>
Define dlarray for Code Generation .....	5-35
dlarray Object Functions with Code Generation Support .....	5-36
Deep Learning Toolbox Functions with dlarray Code Generation Support .....	5-37
MATLAB Functions with dlarray Code Generation Support .....	5-37
<b>dlarray Limitations for Code Generation .....</b>	<b>5-41</b>
Recommended Usage .....	5-41
Limitations .....	5-41
<b>Generated CNN Class Hierarchy .....</b>	<b>5-44</b>
<b>Load Pretrained Networks for Code Generation .....</b>	<b>5-45</b>
Load a Network by Using coder.loadDeepLearningNetwork .....	5-45
Specify a Network Object for Code Generation .....	5-46
Specify a dlnetwork Object for Code Generation .....	5-46
<b>Code Generation for Deep Learning Networks by Using cuDNN .....</b>	<b>5-48</b>
Generate Code and Classify Images by Using GoogLeNet .....	5-48
Requirements .....	5-48
Load Pretrained Network .....	5-49
Create an Entry-Point Function .....	5-50
Code Generation by Using codegen .....	5-50
Generate Code by Using the App .....	5-53
Generated Makefile .....	5-54
Run the Generated MEX .....	5-54
<b>Code Generation for Deep Learning Networks by Using TensorRT .....</b>	<b>5-57</b>
Generate Code and Classify Images by Using GoogLeNet .....	5-57
Requirements .....	5-57
Load Pretrained Network .....	5-58
Create an Entry-Point Function .....	5-59
Code Generation by Using codegen .....	5-60
Generate Code by Using the App .....	5-63
Generated Makefile .....	5-64
Run the Generated MEX .....	5-64

<b>Code Generation for Deep Learning Networks Targeting ARM Mali GPUs</b>	
Requirements	5-66
Load Pretrained Network	5-66
Code Generation by Using cnncodegen	5-67
Limitations	5-69
<b>Data Layout Considerations in Deep Learning</b>	5-70
Data Layout Format for CNN	5-70
Data Layout Format for LSTM	5-71
<b>Quantization of Deep Neural Networks</b>	5-73
Precision and Range	5-73
Histograms of Dynamic Ranges	5-73
<b>Code Generation for Quantized Deep Learning Networks</b>	5-81
Classify Images on a GPU Using a Quantized Network	5-81
Limitations	5-89
<b>Code Generation for Deep Learning Networks</b>	5-91
<b>Lane Detection Optimized with GPU Coder</b>	5-100
<b>Traffic Sign Detection and Recognition</b>	5-111
<b>Logo Recognition Network</b>	5-120
<b>Pedestrian Detection</b>	5-125
<b>Deep Learning Prediction by Using NVIDIA TensorRT</b>	5-132
<b>Code Generation for Semantic Segmentation Network</b>	5-137
<b>Train and Deploy Fully Convolutional Networks for Semantic Segmentation</b>	5-142
<b>Code Generation for Semantic Segmentation Network That Uses U-net</b>	5-154
<b>Code Generation for Denoising Deep Neural Network</b>	5-161
<b>Code Generation for Object Detection by Using YOLO v2</b>	5-165
<b>Code Generation for a Sequence-to-Sequence LSTM Network</b>	5-169
<b>Deep Learning Prediction on ARM Mali GPU</b>	5-175
<b>Code Generation for Object Detection by Using Single Shot Multibox Detector</b>	5-178
<b>Code Generation for a Deep Learning Simulink Model to Classify ECG Signals</b>	5-182
<b>Code Generation for Lidar Point Cloud Segmentation Network</b>	5-189



<b>Code Generation for a Video Classification Network</b> .....	<b>5-196</b>
<b>Code Generation For Object Detection Using YOLO v3 Deep Learning</b>	<b>5-202</b>
<b>Generate Digit Images on NVIDIA GPU Using Variational Autoencoder</b> .....	<b>5-211</b>

## Targeting Embedded GPU Devices

# 6

<b>Build and Run an Executable on NVIDIA Hardware</b> .....	<b>6-2</b>
Learning Objectives .....	<b>6-2</b>
Tutorial Prerequisites .....	<b>6-2</b>
Example: Vector Addition .....	<b>6-3</b>
Create a Live Hardware Connection Object .....	<b>6-3</b>
Generate CUDA Executable Using GPU Coder .....	<b>6-4</b>
Run the Executable and Verify the Results .....	<b>6-5</b>
<b>Build and Run an Executable on NVIDIA Hardware Using GPU Coder App</b> .....	<b>6-7</b>
Learning Objectives .....	<b>6-7</b>
Tutorial Prerequisites .....	<b>6-7</b>
Example: Vector Addition .....	<b>6-8</b>
Custom Main File .....	<b>6-8</b>
GPU Coder App .....	<b>6-9</b>
Run the Executable and Verify the Results .....	<b>6-12</b>
<b>Relocate Generated Code to Another Development Environment</b> .....	<b>6-14</b>
Package Generated Code Using the GPU Coder .....	<b>6-14</b>
Specify packNGo Options .....	<b>6-22</b>
<b>Getting Started with the MATLAB Coder Support Package for NVIDIA Jetson and NVIDIA DRIVE Platforms</b> .....	<b>6-24</b>
<b>Sobel Edge Detection on NVIDIA Jetson Nano Using Raspberry Pi Camera Module V2</b> .....	<b>6-29</b>
<b>Semantic Segmentation on NVIDIA DRIVE</b> .....	<b>6-34</b>
<b>Top-Hat Filtering to Remove Uneven Background Illumination on NVIDIA Jetson TX2 Developer Kit</b> .....	<b>6-39</b>
<b>Deployment and Classification of Webcam Images on NVIDIA Jetson TX2 Platform</b> .....	<b>6-44</b>



# 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.varsize` functions. For example, `coder.typeof(1, [12,1],[true, false])` and `coder.varsize(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
- “Legacy Code Integration” on page 2-18
- “Design Patterns” on page 2-22
- “GPU Memory Allocation and Minimization” on page 2-24
- “Support for GPU Arrays” on page 2-26
- “Simulate Diffraction Patterns Using CUDA FFT Libraries” on page 2-28
- “Benchmark A/b by Using GPU Coder” on page 2-34
- “QR Decomposition on NVIDIA GPU Using cuSOLVER Libraries” on page 2-42
- “Stencil Processing on GPU” on page 2-47
- “Fog Rectification” on page 2-53
- “Stereo Disparity” on page 2-58
- “Feature Extraction Using SURF” on page 2-64
- “Feature Matching” on page 2-68
- “Lane Detection on the GPU by Using the houghlines Function” on page 2-72
- “Edge Detection with Sobel Method in Half-Precision” on page 2-75
- “Deploy Signal Classifier on NVIDIA Jetson Using Wavelet Analysis and Deep Learning” on page 2-79

## 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)
{
    ...
    threadIdx = (((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)(threadIdx / 512U);
    j = (int32_T)(threadIdx - (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.constantMemory](#) | [coder.gpu.kernel](#) | [coder.gpu.kernelfun](#) | [gpucoder.matrixMatrixKernel](#) | [gpucoder.stencilKernel](#)

## Related Examples

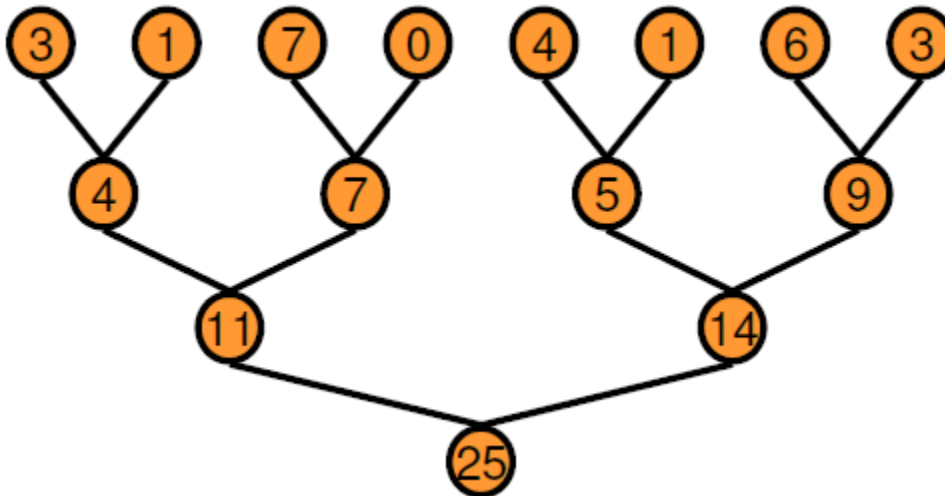
- “Design Patterns” on page 2-22
- “Kernels from Scatter-Gather Type Operations” on page 2-4
- “Kernels from Library Calls” on page 2-8
- “Legacy Code Integration” on page 2-18

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

### Related Examples

- “Design Patterns” on page 2-22

- “Kernels from Element-Wise Loops” on page 2-2
- “Kernels from Library Calls” on page 2-8
- “Legacy Code Integration” on page 2-18

## 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 5-48 and “Code Generation for Deep Learning Networks by Using TensorRT” on page 5-57.

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

## Related Examples

- “Design Patterns” on page 2-22
- “Kernels from Element-Wise Loops” on page 2-2
- “Kernels from Scatter-Gather Type Operations” on page 2-4
- “Legacy Code Integration” on page 2-18

## 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  $\alpha$  and  $\beta$  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 threadIdx;
    ;
    ;
    threadIdx = (int32_T)(((gridDim.x * gridDim.y * blockIdx.z + gridDim.x *
        blockIdx.y) + blockIdx.x) * (blockDim.x * blockDim.y * blockDim.z) +
        (int32_T)((threadIdx.z * blockDim.x * blockDim.y +
            threadIdx.y * blockDim.x) + threadIdx.x));
    if (!(threadIdx >= 144)) {
        /* Copyright 2017 The MathWorks, Inc. */
        b_A[threadIdx] = A[threadIdx];
    }
}

```

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
```

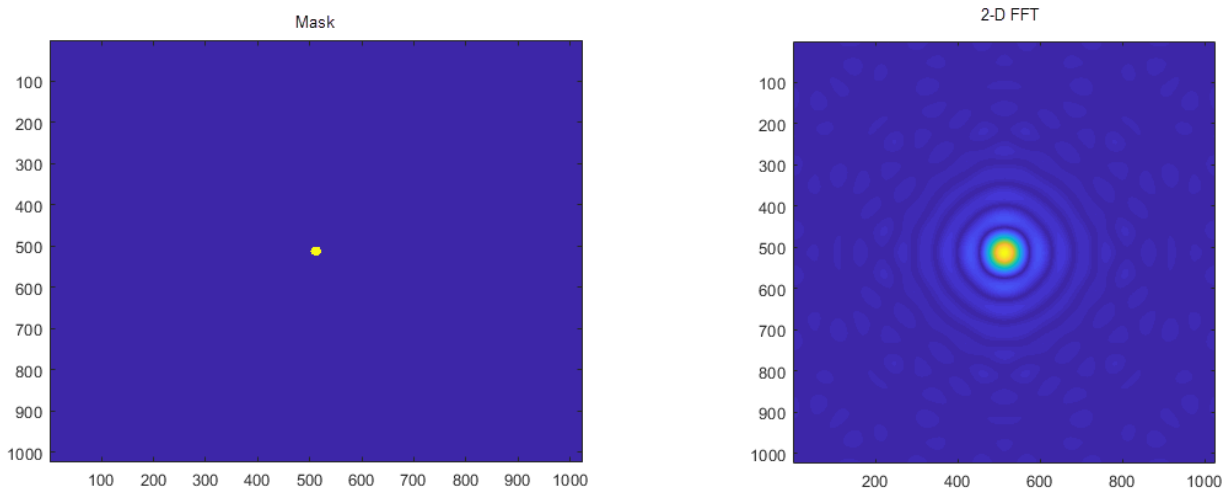
For more information, see “Speed Up Linear Algebra in Generated Standalone Code by Using LAPACK Calls”.

## 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);
...
```

## Legacy Code Integration

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.

---

**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.

---

### `coder.ceval` for GPU Coder

`coder.ceval('-gpudevicefcn', 'devicefun_name', devicefun_arguments)` is a subset of the `coder.ceval` function from MATLAB Coder that allows you to call `__device__` functions from within kernels. `'-gpudevicefcn'` indicates to `coder.ceval` that the target function is on the GPU device. `devicefun_name` is the name of the `__device__` function and `devicefun_arguments` is a comma-separated list of input arguments in the order that `devicefun_name` requires.

For code generation, you must specify the type, size, and complexity data type of the arguments before calling `coder.ceval`.

This function is a code generation function and causes errors when used otherwise.

### Legacy Code Example

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('-gpudevicefcn', '__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

```

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

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

## Related Examples

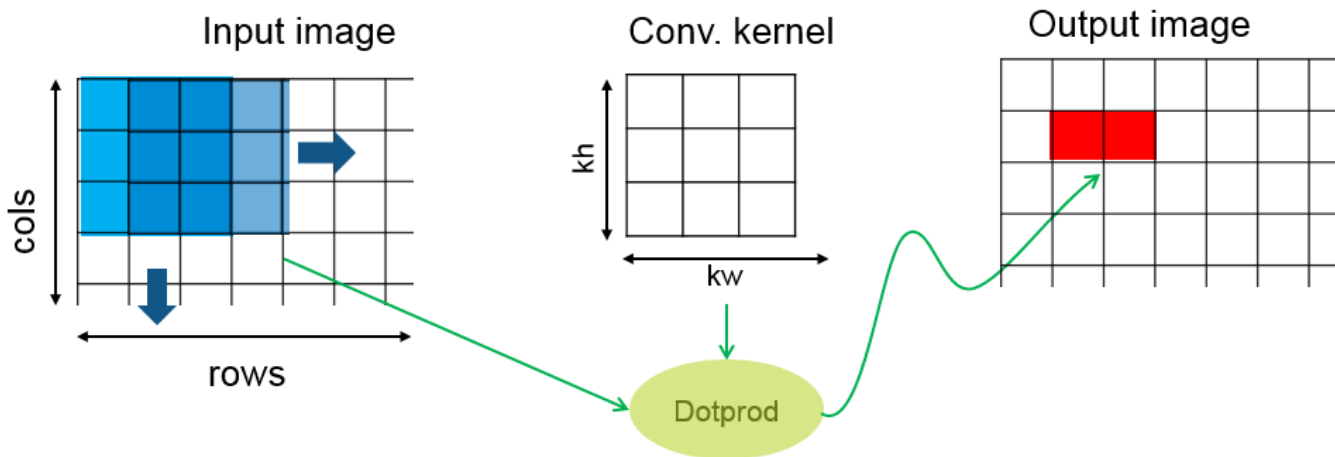
- “Design Patterns” on page 2-22
- “Kernels from Element-Wise Loops” on page 2-2
- “Kernels from Scatter-Gather Type Operations” on page 2-4
- “Kernels from Library Calls” on page 2-8

## Design Patterns

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 `gpcoder.stencilKernel` function and create CUDA code for stencil functions. For an example that demonstrates stencil preprocessing, see “Stencil Processing on GPU” on page 2-47.

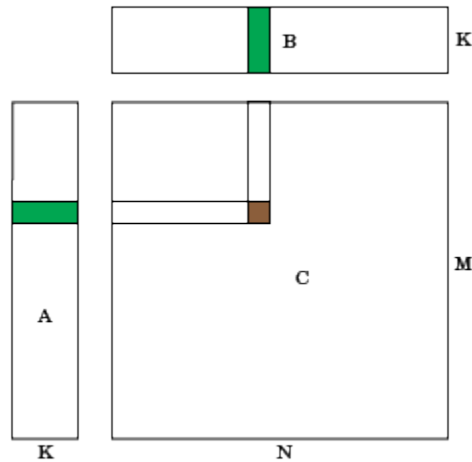
For very large input sizes, the `gpcoder.stencilKernel` function may produce CUDA code that does not numerically match the MATLAB simulation. In such cases, consider reducing the size of the input to produce accurate results.

### 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 `gpuscoder.matrixMatrixKernel` function and create CUDA code for performing matrix-matrix type operations.

## See Also

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

## Related Examples

- “Stencil Processing on GPU” on page 2-47

## 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
- “Legacy Code Integration” on page 2-18

## 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'`.

### 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.

## 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. Depending on whether a given input to the entry-point function is identified as CPU or GPU based input and depending on the usage of the variable (used on the GPU or on the CPU) `cudaMemcpy` calls are inserted efficiently in the generated code. By using the GPU array functionality you can minimize the number of `cudaMemcpy` calls in the generated code.

To use this functionality, do one of the following:

- 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 numeric and logical types. `char` and `half` data types are not supported. For using variable dimension arrays, only the bounded types are supported. Scalar GPU arrays, structures, cell-arrays, classes, enumerated types, and fixed-point data types are not supported.
- The code generator supports all target types for GPU arrays - `'mex'`, `'lib'`, `'dll'`, and `'exe'`. 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 main function. Software-In-the-Loop (SIL) is supported for `'lib'` and `'dll'`.
- The memory allocation (`malloc`) mode property of the code configuration object must be set to be `'discrete'`. For example,

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

GPU arrays are not supported in the `'unified'` memory mode.

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

## See Also

`coder.gpu.constantMemory` | `coder.gpu.kernel` | `coder.gpu.kernelfun` | `gpuCoder.matrixMatrixKernel` | `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 2-22

## 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);
```

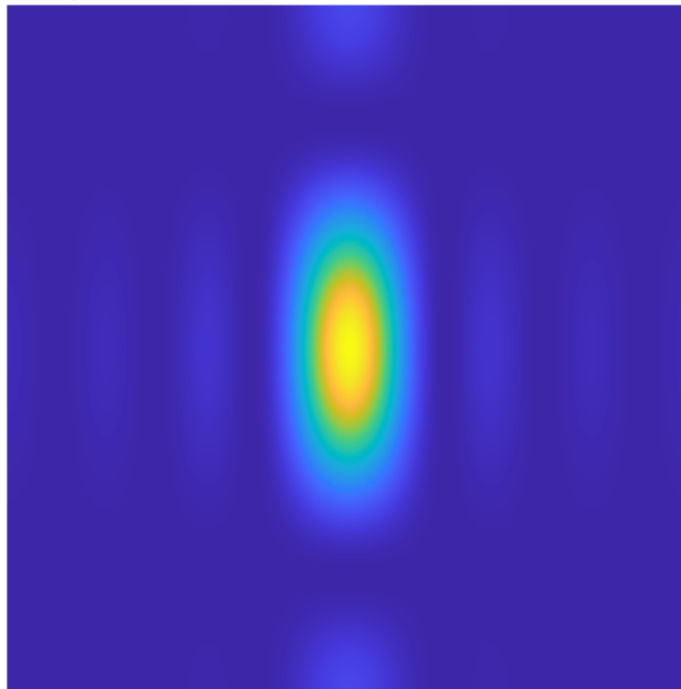
### Display the Light Intensity for a Rectangular Aperture

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

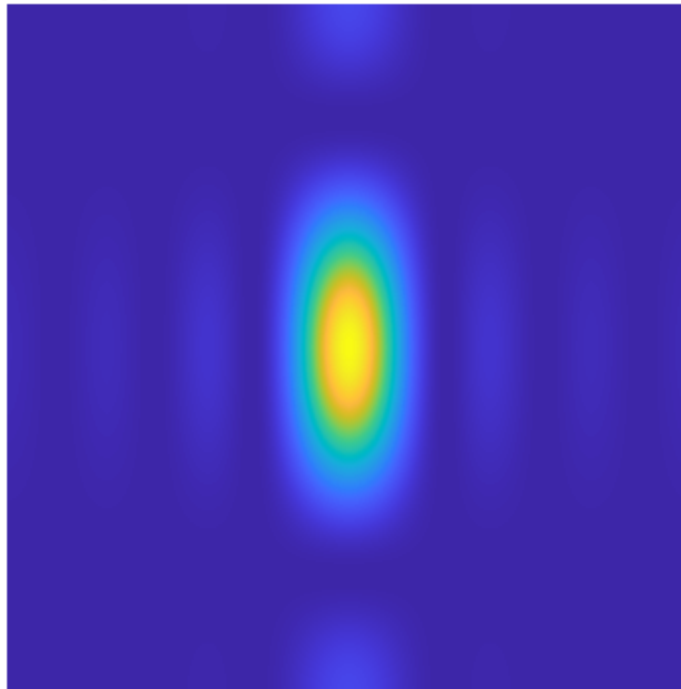
You do not have to create an entry-point function. You can directly generate code for the MATLAB® `fft2` function. To generate CUDA MEX for the MATLAB `fft2` function, in the configuration object, set the `EnablecuFFT` property and use the `codegen` function. GPU Coder replaces `fft`, `ifft`, `fft2`, `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.

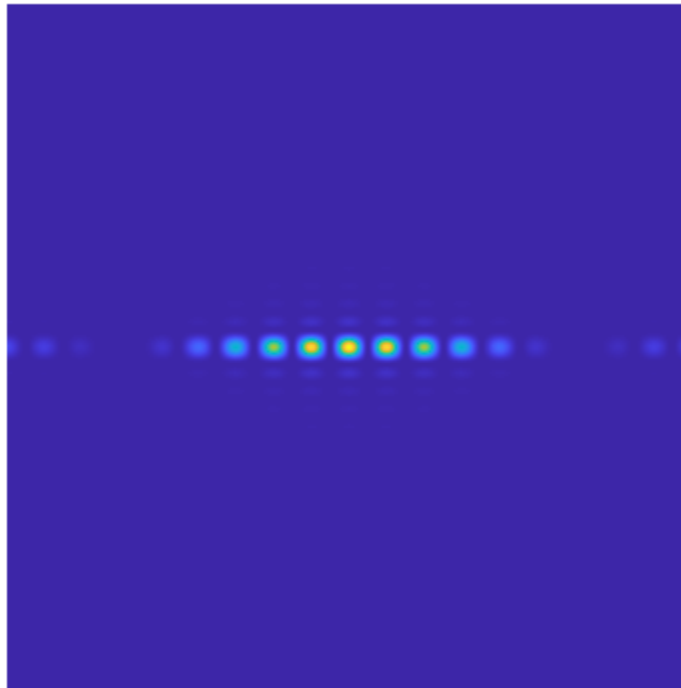
```
slits          = (abs(gx) <= 10/N2) .* (abs(gx) >= 8/N2);
aperture      = slits .* (abs(gy) < 20/N2);
lightsource   = double(aperture);
```

### 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 2-26.

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.

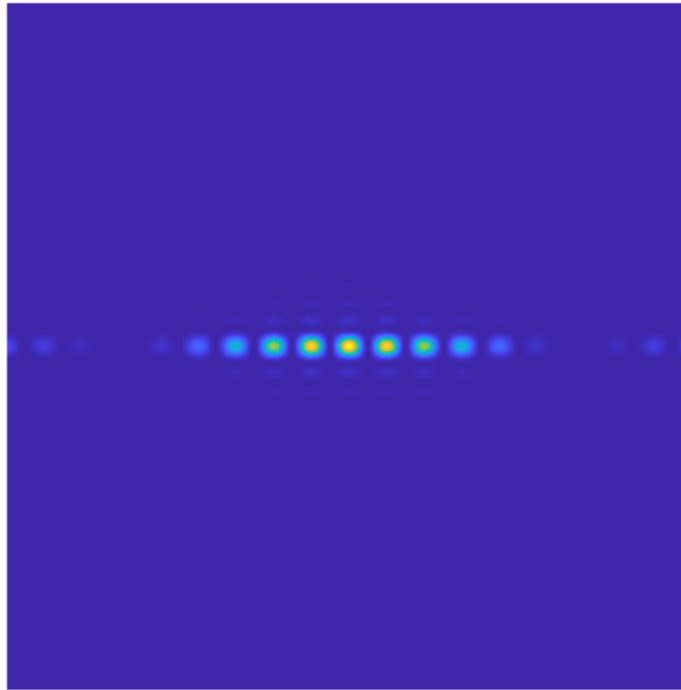
### **Verify Results of CUDA MEX Using GPU Pointer as Input**

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.checkGpuInstall` | `coder.gpu.constantMemory` | `coder.gpu.kernel` | `coder.gpu.kernelfun` | `gpuscoder.matrixMatrixKernel` | `gpuscoder.stencilKernel`

#### Objects

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

### Related Examples

- “Kernels from Library Calls” on page 2-8
- “Design Patterns” on page 2-22
- “Kernels from Scatter-Gather Type Operations” on page 2-4
- “Legacy Code Integration” on page 2-18

## Benchmark $A \setminus b$ by Using GPU Coder

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

### Prerequisites

- CUDA® enabled NVIDIA® GPU with compute capability 3.5 or higher.
- 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 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 = 0.25*g.AvailableMemory/1024^3;
```

### 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 `genData.m`

```
function [A, b] = genData(n, clz)

% Copyright 2017-2019 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 2017-2019 The MathWorks, Inc.

    coder.gpu.kernelfun();
    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 2017-2019 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.

```
% Declare the matrix sizes to be a multiple 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\b` a few times to get an accurate measure of the time it takes.

type `benchFcnMat.m`

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

% Copyright 2017-2019 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 2017-2019 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 R2020a 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 2017-2019 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);
results.sizeSingle = sizeSingle;
results.timeSingleCPU = cpu;
results.timeSingleGPU = gpu;
[cpu, gpu] = executeBenchmarks('double', sizeDouble, numReps);
results.sizeDouble = sizeDouble;
results.timeDoubleCPU = cpu;
results.timeDoubleGPU = gpu;

```

Starting benchmarks with 9 different single-precision matrices of sizes ranging from 1024-by-1024 to 25600-by-25600.

Size : 1024

Creating a matrix of size 1024-by-1024.

Time on CPU: 0.010894 sec

Time on GPU: 0.012735 sec

Size : 4096

Creating a matrix of size 4096-by-4096.

Time on CPU: 0.256912 sec

Time on GPU: 0.056594 sec

Size : 7168

Creating a matrix of size 7168-by-7168.

Time on CPU: 0.859825 sec

Time on GPU: 0.138257 sec

Size : 10240

Creating a matrix of size 10240-by-10240.

Time on CPU: 2.190538 sec

Time on GPU: 0.276720 sec

Size : 13312

Creating a matrix of size 13312-by-13312.

Time on CPU: 4.265689 sec

Time on GPU: 0.481420 sec

Size : 16384

Creating a matrix of size 16384-by-16384.

Time on CPU: 8.167554 sec

Time on GPU: 0.771961 sec

```
Size : 19456
Creating a matrix of size 19456-by-19456.
Time on CPU: 12.814798 sec
Time on GPU: 1.150009 sec
```

```
Size : 22528
Creating a matrix of size 22528-by-22528.
Time on CPU: 19.060859 sec
Time on GPU: 1.671082 sec
```

```
Size : 25600
Creating a matrix of size 25600-by-25600.
Time on CPU: 27.237500 sec
Time on GPU: 2.318981 sec
```

Starting benchmarks with 6 different double-precision matrices of sizes ranging from 1024-by-1024 to 16384-by-16384.

```
Size : 1024
Creating a matrix of size 1024-by-1024.
Time on CPU: 0.025851 sec
Time on GPU: 0.021405 sec
```

```
Size : 4096
Creating a matrix of size 4096-by-4096.
Time on CPU: 0.435886 sec
Time on GPU: 0.175009 sec
```

```
Size : 7168
Creating a matrix of size 7168-by-7168.
Time on CPU: 1.734958 sec
Time on GPU: 0.765464 sec
```

```
Size : 10240
Creating a matrix of size 10240-by-10240.
Time on CPU: 4.240617 sec
Time on GPU: 2.081403 sec
```

```
Size : 13312
Creating a matrix of size 13312-by-13312.
Time on CPU: 8.611123 sec
Time on GPU: 4.415243 sec
```

```
Size : 16384
Creating a matrix of size 16384-by-16384.
Time on CPU: 19.387437 sec
Time on GPU: 8.050974 sec
```

### Plot the Performance

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

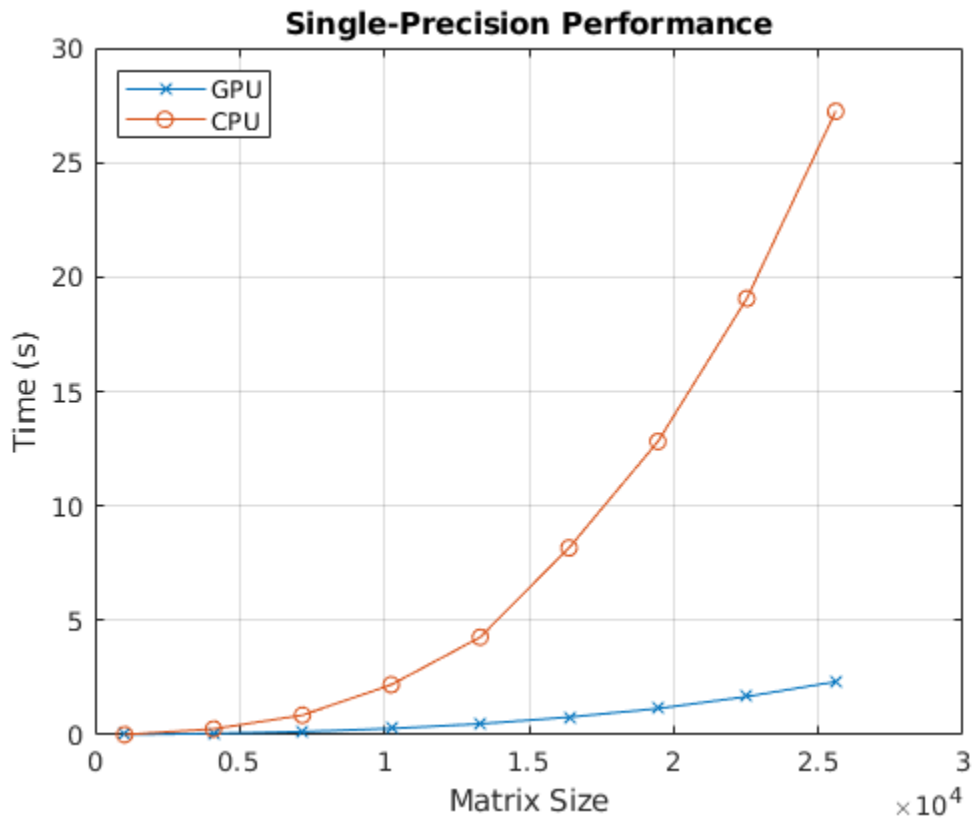
First, look at the performance of the backslash operator in single precision.

```
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;

```

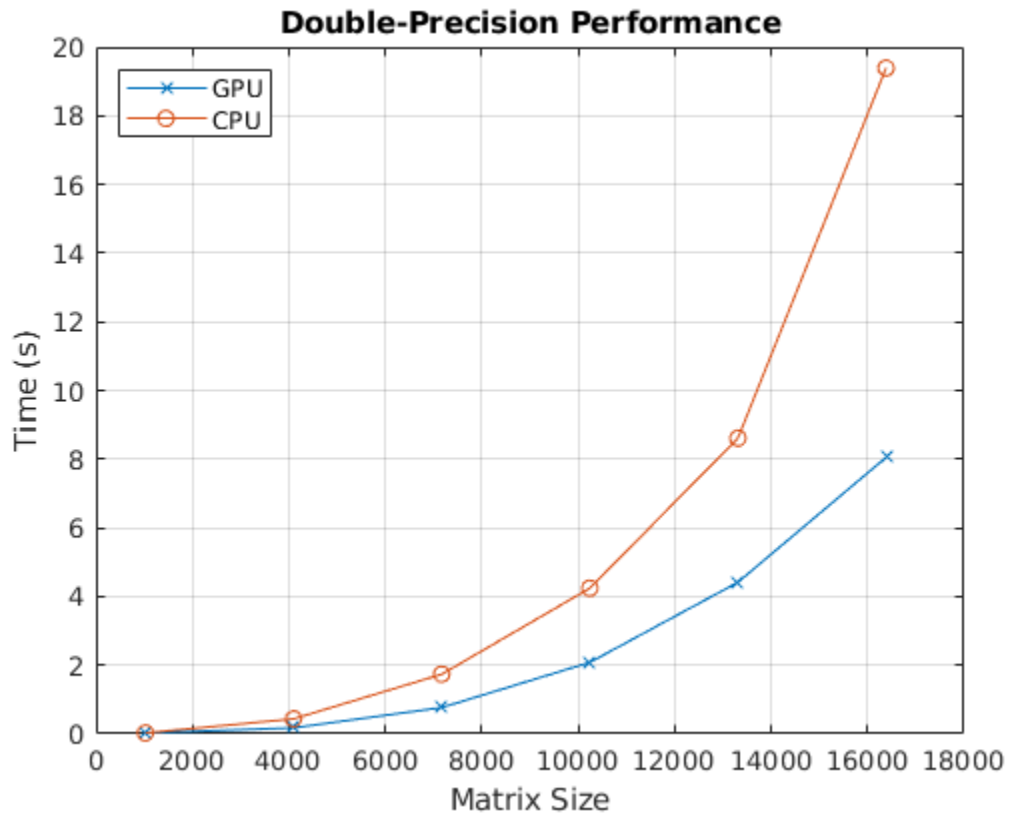


Now, look at the performance of the backslash operator in double precision.

```

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;

```



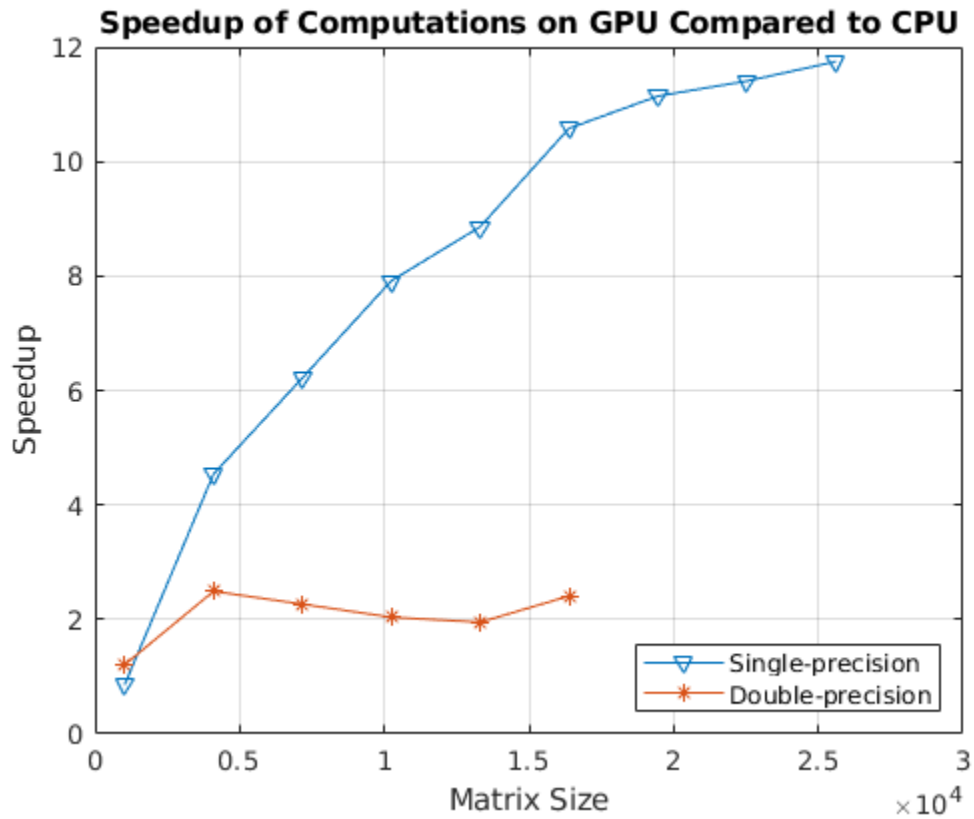
Finally, look at the speedup of the backslash operator when comparing the GPU to the CPU.

```

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

### Objects

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

## Related Examples

- “Kernels from Library Calls” on page 2-8
- “Design Patterns” on page 2-22
- “Kernels from Scatter-Gather Type Operations” on page 2-4
- “Legacy Code Integration” on page 2-18

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

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
```

## 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: To view the report, open('codegen/exe/linsolveQR/html/report.mldatx')

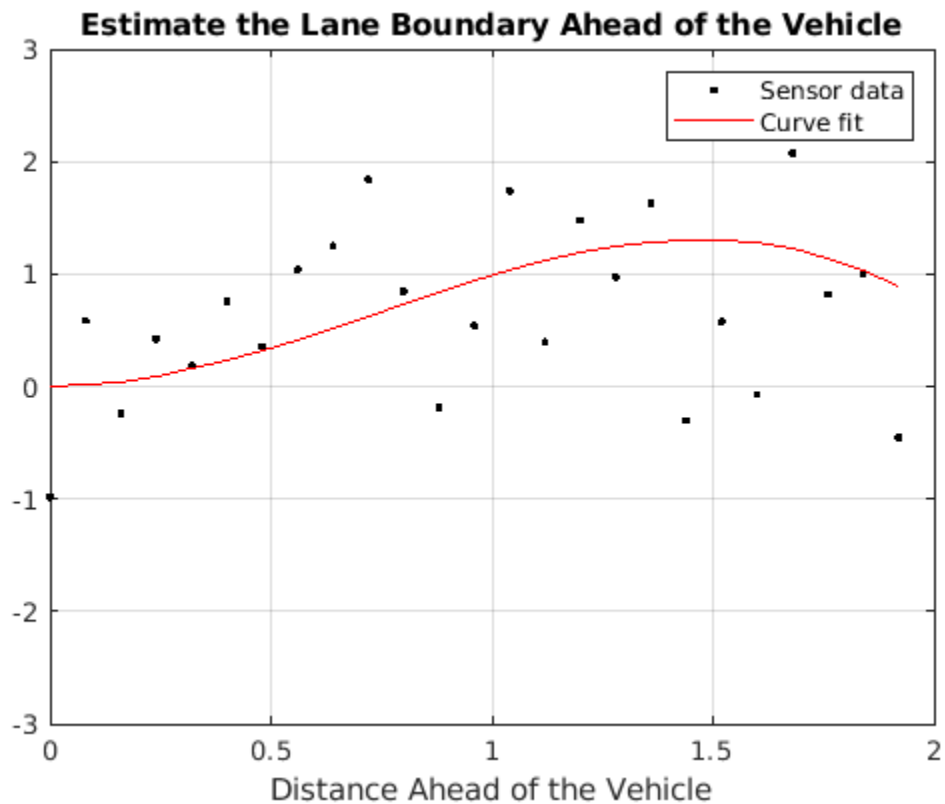
### Standalone Code Execution

When you execute the generated standalone executable, the outputs  $\hat{y}$  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.checkGpuInstall](#) | [coder.gpu.constantMemory](#) | [coder.gpu.kernel](#) | [coder.gpu.kernelfun](#) | [gpcoder.matrixMatrixKernel](#) | [gpcoder.stencilKernel](#)

### Objects

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

## Related Examples

- “Kernels from Library Calls” on page 2-8
- “Design Patterns” on page 2-22
- “Kernels from Scatter-Gather Type Operations” on page 2-4
- “Legacy Code Integration” on page 2-18

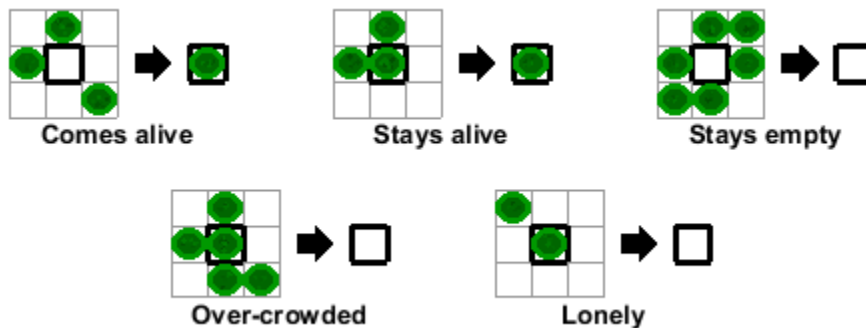
## Stencil Processing on GPU

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.

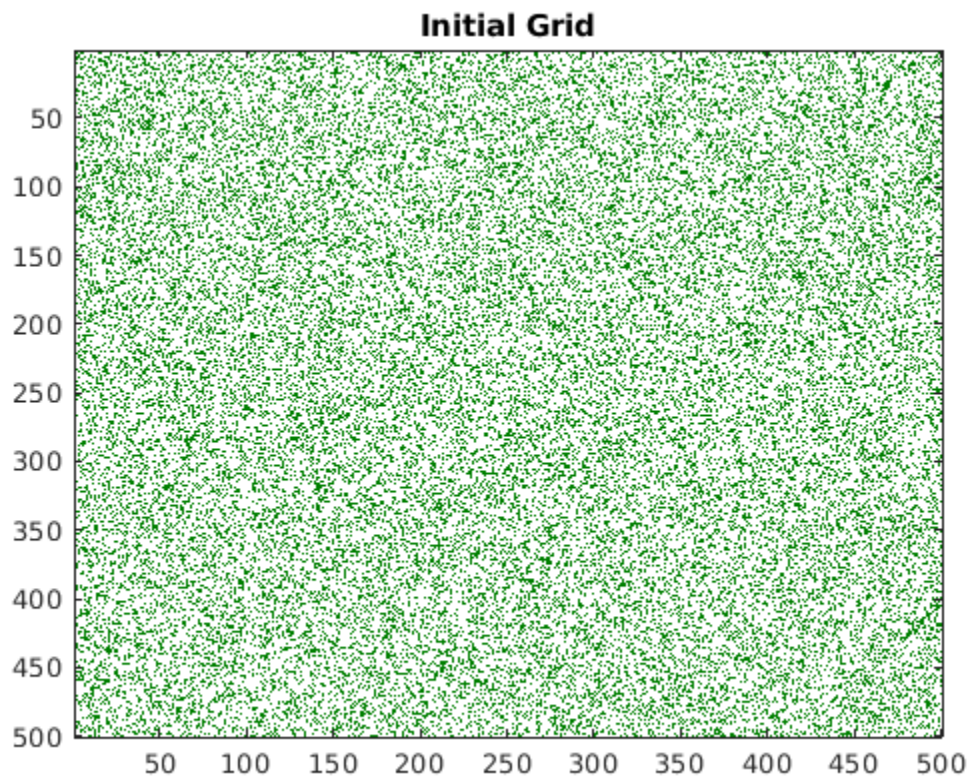
```
envCfg = coder.gpuEnvConfig('host');  
envCfg.BasicCodegen = 1;  
envCfg.Quiet = 1;  
coder.checkGpuInstall(envCfg);
```

### 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);

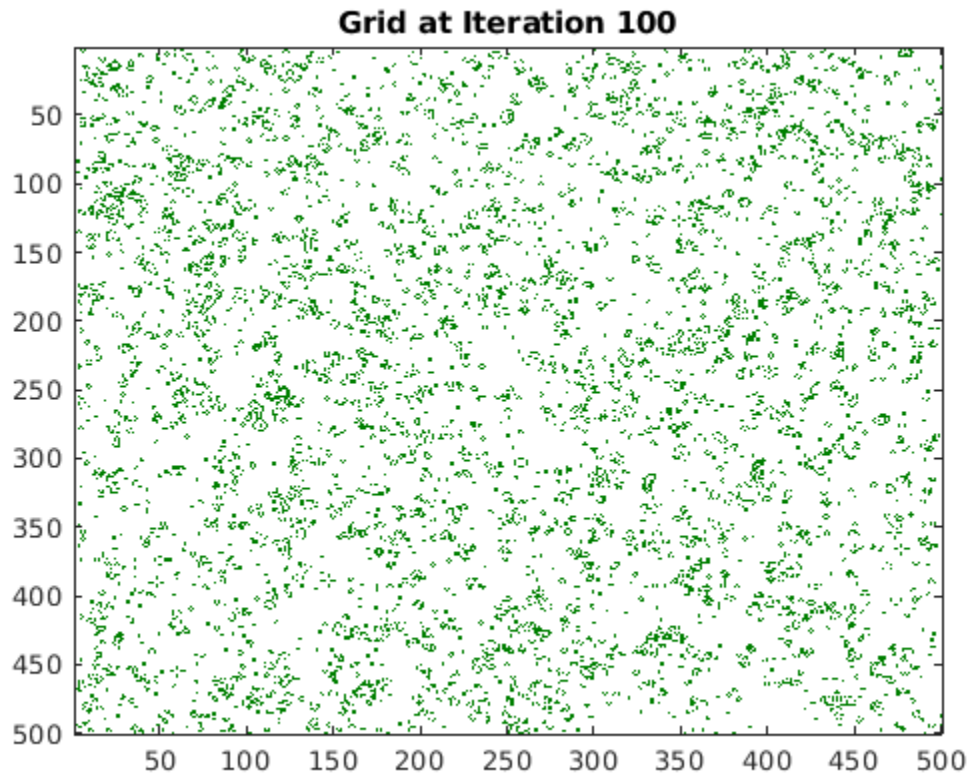
    imagesc(grid);
    colormap([1 1 1;0 0.5 0]);
    title(['Grid at Iteration ',num2str(generation)]);
    drawnow;
end

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 `gpuCoder.stencilKernel` 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 = gpuCoder.stencilKernel(@updateElem, grid, [3,3], '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

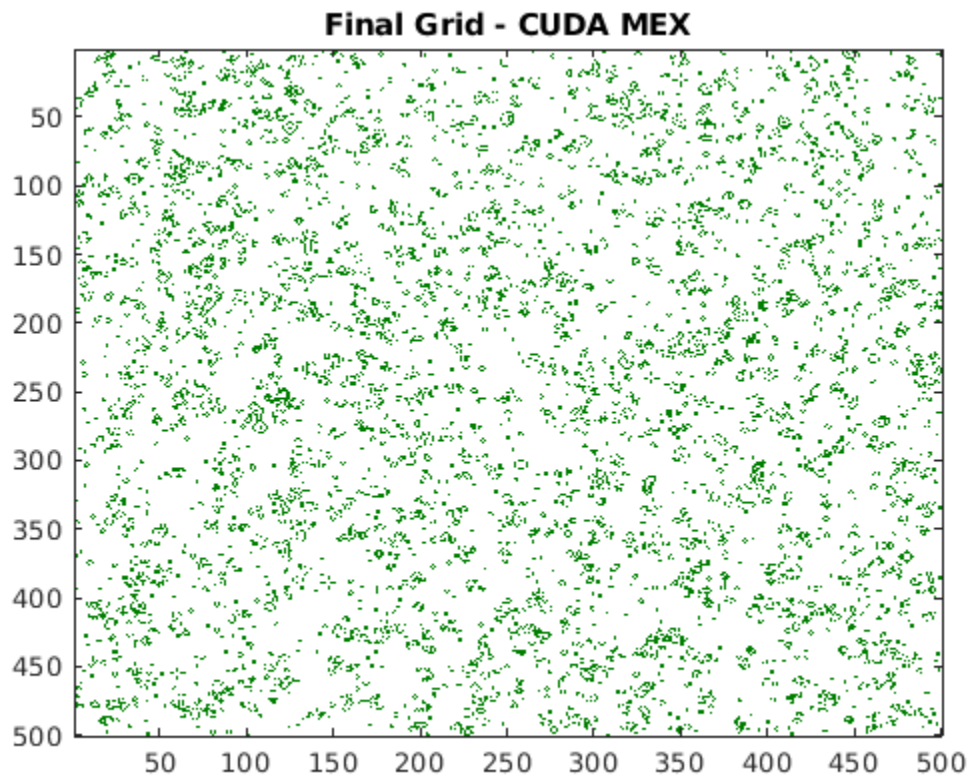
To generate CUDA MEX for the `gameoflife_stencil` function, create a GPU code configuration object, and then use the `codegen` command.

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

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

### Objects

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

## Related Examples

- “Kernels from Library Calls” on page 2-8
- “Design Patterns” on page 2-22
- “Kernels from Scatter-Gather Type Operations” on page 2-4
- “Legacy Code Integration” on page 2-18

## 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 -o fog_rectification_gpu_mex

```

Code generation successful: To view the report, open('codegen/mex/fog\_rectification/html/report.r

### Run the MEX Function with Foggy Image

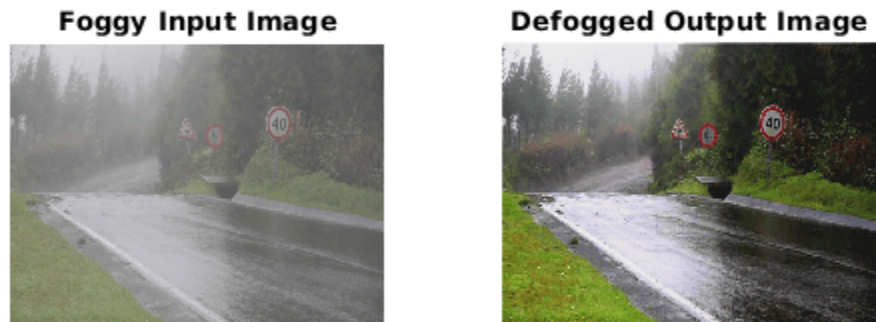
Run the generated `fog_rectification_gpu_mex` with a foggy input image, and then plot the foggy and defogged images.

```

[outputImage] = fog_rectification_gpu_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'`.

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



Code generation successful: To view the report, open('codegen/mex/fog\_rectification\_half\_precision')

## See Also

### Functions

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

### Objects

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

## Related Examples

- “Kernels from Library Calls” on page 2-8
- “Design Patterns” on page 2-22
- “Kernels from Scatter-Gather Type Operations” on page 2-4
- “Legacy Code Integration” on page 2-18

## 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).

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

            % 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((
    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
```

### **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



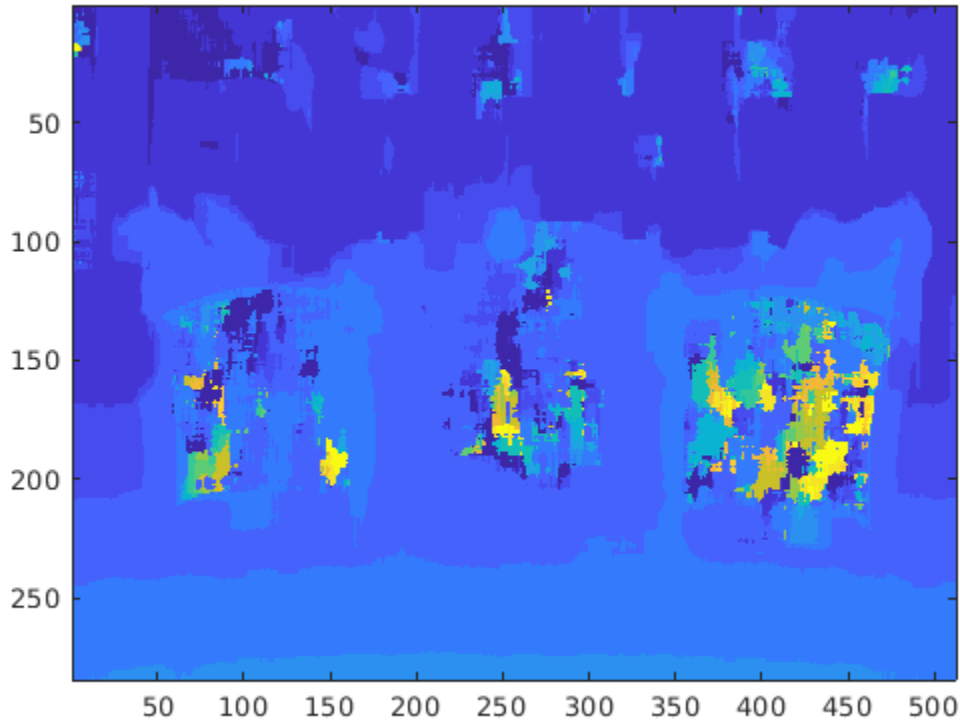
### Generate GPU Code

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

Code generation successful: To view the report, open('codegen/mex/stereoDisparity/html/report.mlx')

### 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: To view the report, open('codegen/mex/stereoDisparityHalfPrecision/h')

## See Also

### Functions

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

### Objects

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

## Related Examples

- “Kernels from Library Calls” on page 2-8
- “Design Patterns” on page 2-22
- “Kernels from Scatter-Gather Type Operations” on page 2-4
- “Legacy Code Integration” on page 2-18

## 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_gpu.m` function performs non-maximal suppression to filter out only the useful interest points from the responses obtained earlier. To generate a kernel that uses the `atomicAdd` operation, use the `coder.ceval` construct. Because this construct is not compatible when invoked directly from MATLAB®, there are two different function calls. The `NonMaxSuppression_gpu.m` function is invoked when GPU code generation is enabled and the `NonMaxSuppression.m` is invoked when you are executing the algorithm directly in MATLAB.
- 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');  
evalc('codegen -config cfg SurfDetect -args {inputImage}');
```

### Run the MEX Function on a GPU

You can invoke the generated MEX function SurfDetect\_mex to run on a GPU:

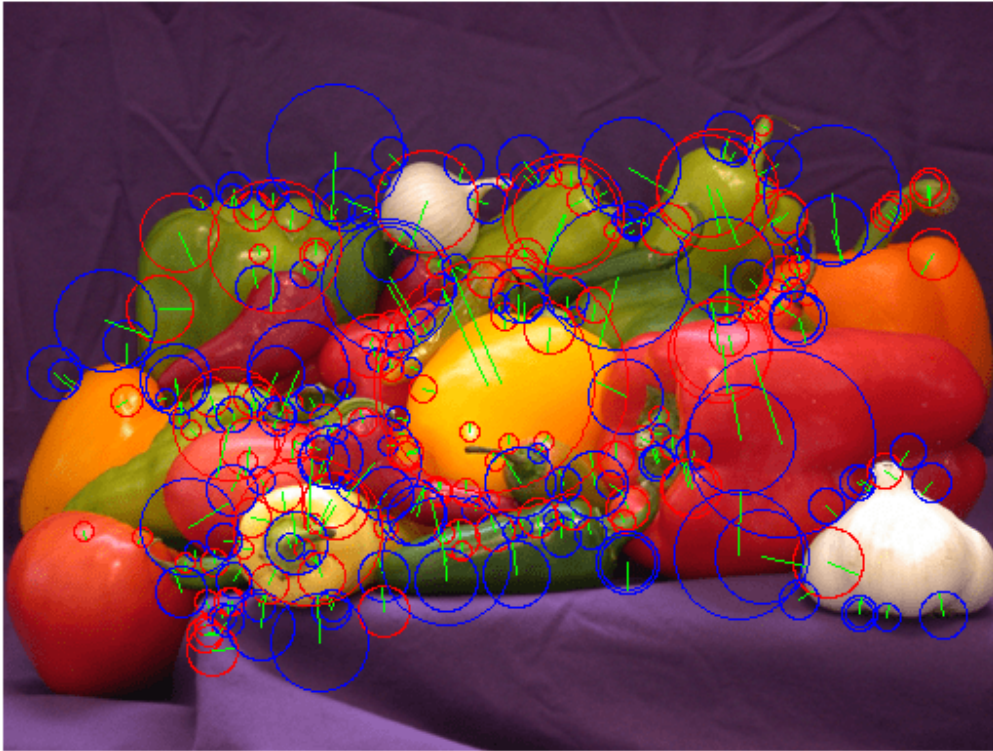
```
disp('Running GPU Coder SURF');  
interestPointsGPU = SurfDetect_mex(inputImage);  
fprintf(' GPU Coder SURF found: %d interest points\n',length(interestPointsGPU));
```

```
Running GPU Coder SURF  
GPU Coder SURF found: 249 interest points
```

### 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 SURF: Speeded-Up Robust Features by Herbert Bay, Tinne Tuytelaars, and Luc Van Gool.

## See Also

### Functions

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

### Objects

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

## Related Examples

- “Kernels from Library Calls” on page 2-8
- “Design Patterns” on page 2-22
- “Kernels from Scatter-Gather Type Operations” on page 2-4
- “Legacy Code Integration” on page 2-18

## 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.

```
% Read Image
K = imread('cameraman.tif'); % Reference image
```

```

refImage = imresize(K,3);

scale = 0.7; % Scaling the image.
J = imresize(refImage, scale);
theta = 30.0; % Rotating the image
qryImage = imrotate(J,theta); % Query image

[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)

% Copyright 2018 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;
codegen -config cfg -args {refPoints,refDescFeat,qryPoints,qryDescFeat} feature_matching -o feature_matching_gpu
[matchedRefPoints_gpu,matchedQryPoints_gpu] = feature_matching_gpu_mex(refPoints,refDescFeat,qryPoints,qryDescFeat);

Code generation successful.

```

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

### Objects

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

## Related Examples

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

- “Design Patterns” on page 2-22
- “Kernels from Scatter-Gather Type Operations” on page 2-4
- “Legacy Code Integration” on page 2-18

## 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 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 -o lane_detection_houghl

```

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_gpu_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 >
        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
end

```

```
title(p1, 'Input Image');  
title(p2, 'Lane Detected Output Image');
```



### See Also

#### Functions

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

#### Objects

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

### Related Examples

- “Kernels from Library Calls” on page 2-8
- “Design Patterns” on page 2-22
- “Kernels from Scatter-Gather Type Operations” on page 2-4
- “Legacy Code Integration” on page 2-18

## 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
% sobelEdgeDetection 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

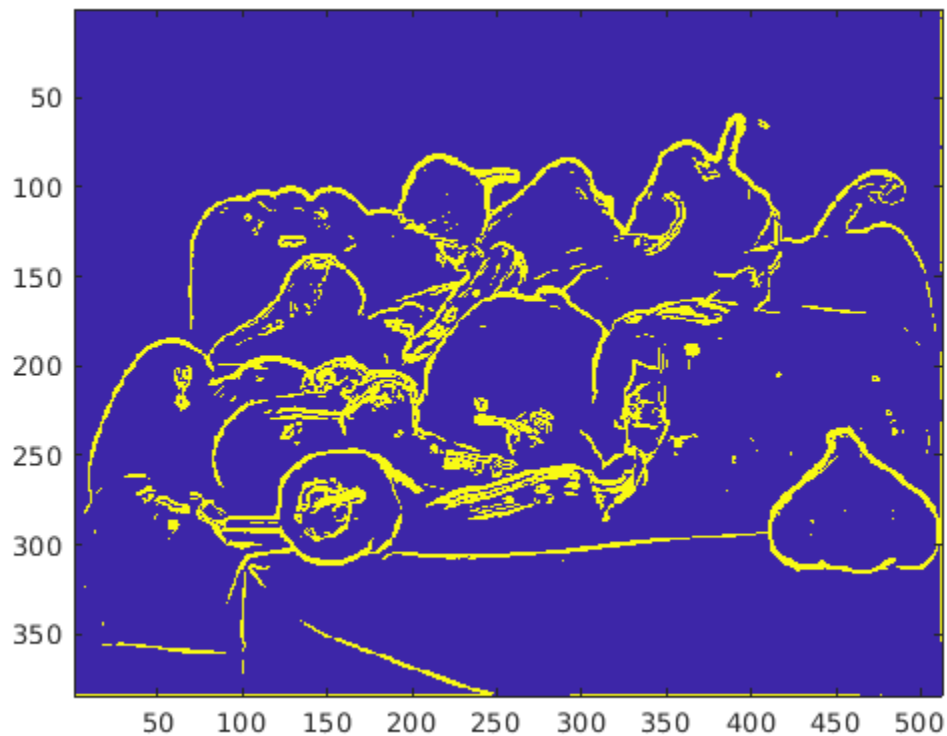
ComputeCapability property of the code configuration object to '6.0'. For half-precision, the memory allocation (malloc) mode for generating CUDA code must be set to 'Discrete'.

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

#### Objects

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

### Related Examples

- “Kernels from Library Calls” on page 2-8
- “Design Patterns” on page 2-22
- “Kernels from Scatter-Gather Type Operations” on page 2-4
- “Legacy Code Integration” on page 2-18

# 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-master.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-master.zip'), tempdir)
```

Unzipping creates the folder `physionet-ECG_data-master` 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-master`. Load the data file into your MATLAB workspace.

```
unzip(fullfile(tempdir,'physionet_ECG_data-master','ECGData.zip'),...
    fullfile(tempdir,'physionet_ECG_data-master'))
load(fullfile(tempdir,'physionet_ECG_data-master','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 2-0 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 2-0 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 convolutions v
2	'relu_conv10'	ReLU	ReLU
3	'pool10'	Global Average Pooling	Global average pooling
4	'prob'	Softmax	softmax
5	'ClassificationLayer_predictions'	Classification Output	crossentropyex with 'tenc

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] and p
2	'relu_conv10'	ReLU	ReLU
3	'pool10'	Global Average Pooling	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	00:00:02	26.67%	25.00%	4.1769	2.9
2	10	00:00:12	73.33%	59.38%	0.9877	1.7
3	20	00:00:21	60.00%	56.25%	0.9164	0.9
4	30	00:00:31	86.67%	68.75%	0.6698	0.7
5	40	00:00:40	66.67%	68.75%	0.9053	0.7

7	50	00:00:50	80.00%	78.13%	0.5422	0.4
8	60	00:00:59	100.00%	81.25%	0.4187	0.4
9	70	00:01:08	93.33%	84.38%	0.3561	0.5
10	80	00:01:18	73.33%	84.38%	0.5141	0.4
12	90	00:01:27	86.67%	84.38%	0.4220	0.4
13	100	00:01:36	93.33%	90.63%	0.1923	0.3
14	110	00:01:46	100.00%	90.63%	0.1472	0.3
15	120	00:01:55	100.00%	93.75%	0.0791	0.2
17	130	00:02:04	86.67%	93.75%	0.2486	0.2
18	140	00:02:14	100.00%	93.75%	0.0386	0.2
19	150	00:02:23	100.00%	93.75%	0.0487	0.2
20	160	00:02:32	100.00%	93.75%	0.0224	0.2

Total training time: 1.61e+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');
```

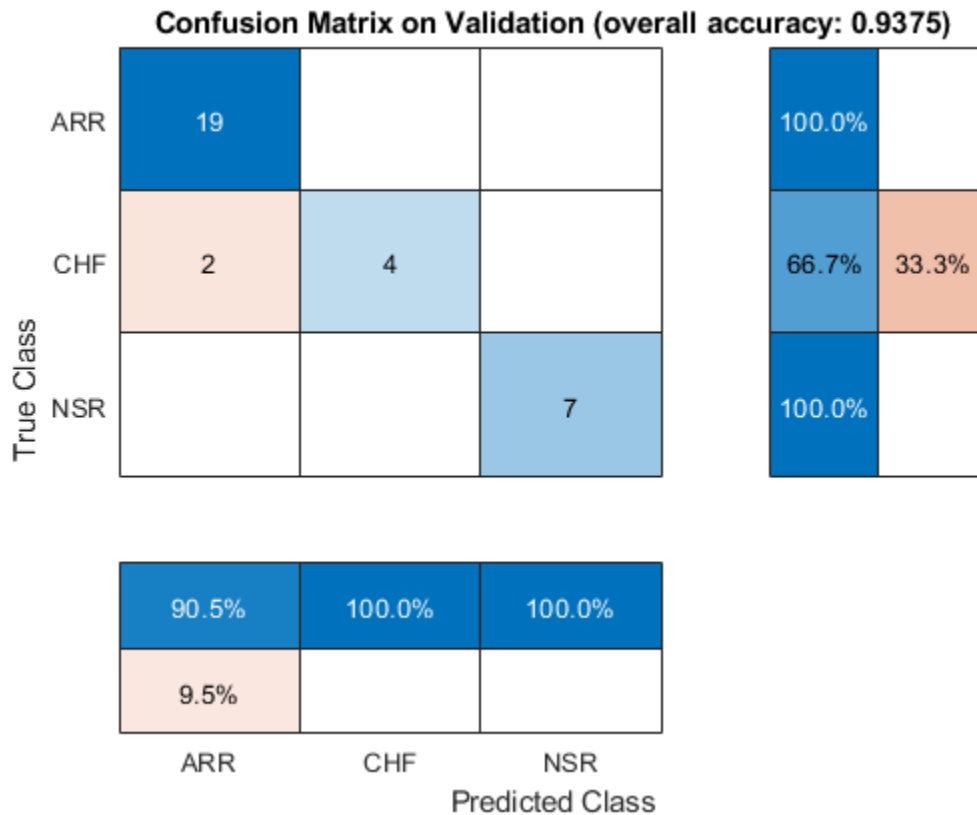
Use the trained network to predict the classes for the validation set.

```
[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: 2981.55 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): 2.94e-02 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, username, and password of the target board to create a live hardware connection object.

Create a live hardware connection object for the Jetson hardware. During the hardware live object creation checking of hardware, IO server installation and gathering peripheral information on target are performed. This information is displayed in the Command Window.

```

hwobj = jetson('gpuCoder-nano-2','ubuntu','ubuntu');

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 TX1
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
Available Webcams :
Available GPUs  : NVIDIA Tegra X1

```

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
    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)
    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

```

```

### Launching the executable on the target...
Executable launched successfully with process ID 5672.
Displaying the simple runtime log for the executable...

```

Note: For the complete log, run the following command in the MATLAB command window:  
`system(hwobj,'cat /home/ubuntu/projectECG/MATLAB_ws/R2021a/C/Users/pkostelev/OneDrive_-_MathWorks`

```
Fetch time = 9.743e+00 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 = 1.288e+01 sec
Example ECG Type: ARR
```

```
disp(PredTableJetson)
```

ARR	CHF	NSR
0.99872	0.001131	0.000153

```
PredTableMATLAB = array2table(ModPredProb(:)', 'VariableNames', matlab.lang.makeValidName(PredCat)
disp(PredTableMATLAB)
```

ARR	CHF	NSR
0.99872	0.0011298	0.00015316

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.checkGpuInstall | coder.gpu.kernel | coder.gpu.kernelfun

**Objects**

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

**Related Examples**

- “Kernels from Library Calls” on page 2-8
- “Design Patterns” on page 2-22
- “Kernels from Scatter-Gather Type Operations” on page 2-4

# 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
- “Deep Learning in Simulink by Using MATLAB Function Block” on page 3-14
- “Deep Learning in Simulink by Using Deep Neural Networks Library” on page 3-22
- “Targeting NVIDIA Embedded Boards” on page 3-29
- “Numerical Equivalence Testing” on page 3-31
- “Parameter Tuning and Signal Monitoring by Using External Mode” on page 3-37
- “GPU Code Generation for Lane Detection in Simulink” on page 3-42
- “GPU Code Generation for a Fog Rectification Simulink Model” on page 3-47
- “Code Generation for a Deep Learning Simulink Model to Classify ECG Signals” on page 3-50
- “Code Generation for a Deep Learning Simulink Model that Performs Lane and Vehicle Detection” on page 3-57

## 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.

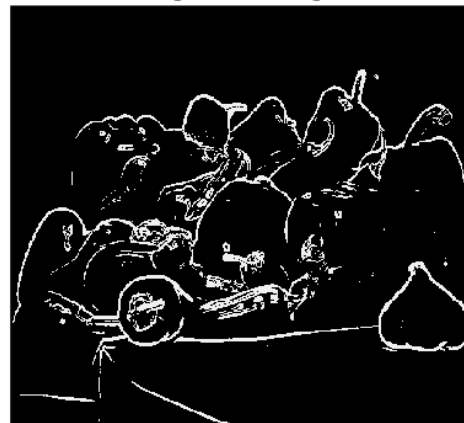
The Sobel edge algorithm computes the horizontal gradient (H) and the vertical gradient (V) of the input image by using two orthogonal filter kernels (k and k'). After the filtering operation, the algorithm computes the gradient magnitude and applies a threshold to find the regions of the images that 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



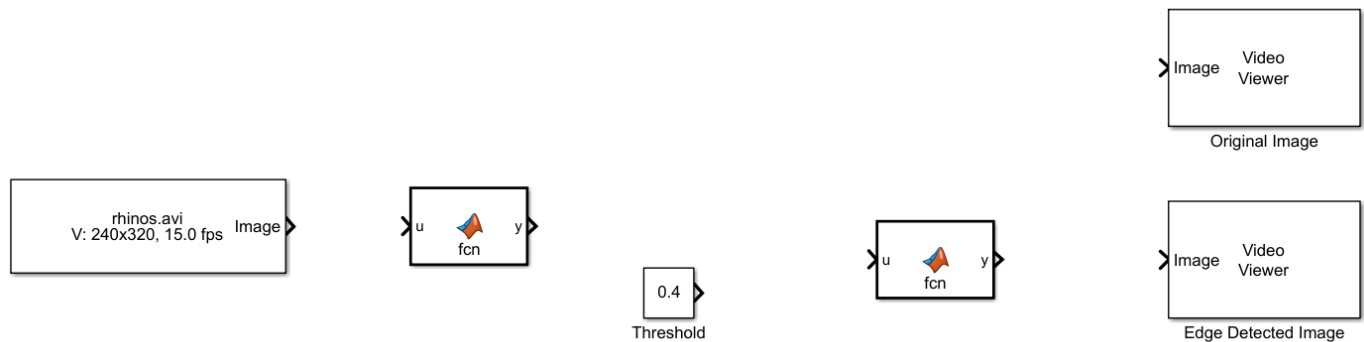


## Create Edge Detection Model

- 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.

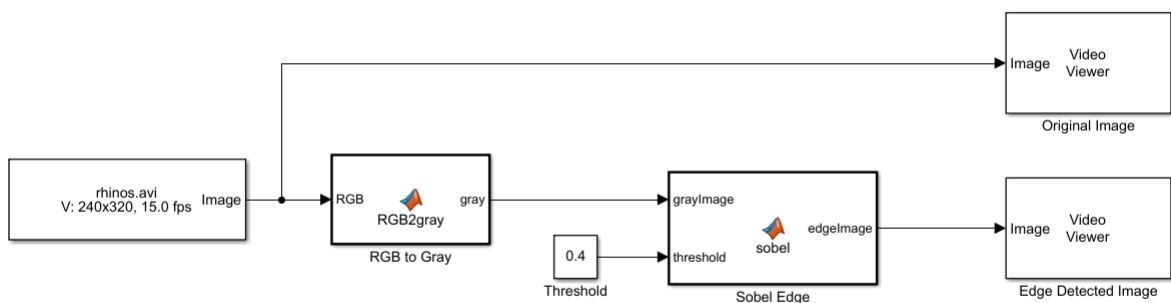
- 9 Modify the other MATLAB Function block to implement the RGB to grayscale conversion prior to the Sobel edge detection operation. Set the **Function packaging** parameter of the MATLAB Function block to Reusable function.

```
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)));

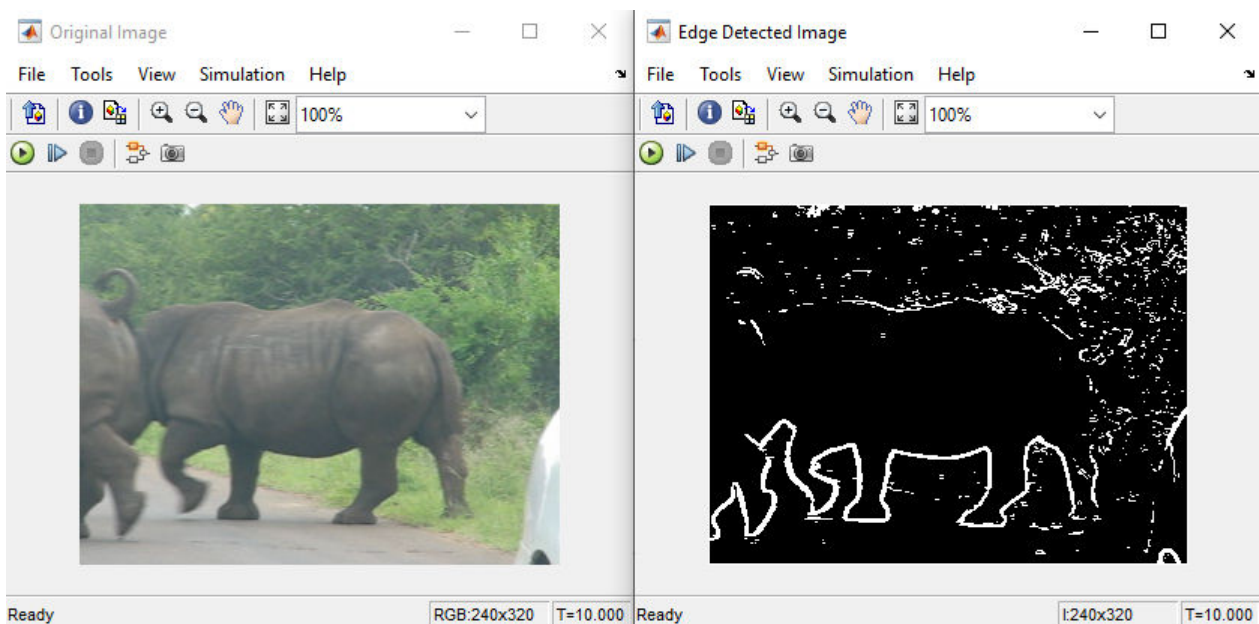
end
```

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
<b>Type</b>	Fixed-step	Maintains a constant (fixed) step size.
<b>Solver</b>	discrete (no continuous states)	Applies a fixed-step integration technique for computing the state derivative of the model.
<b>Fixed-step size</b>	auto	Simulink chooses the step size.

### Solver selection

Type: Fixed-step

Solver: discrete (no continuous states)

### ▼ Solver details

Fixed-step size (fundamental sample time):

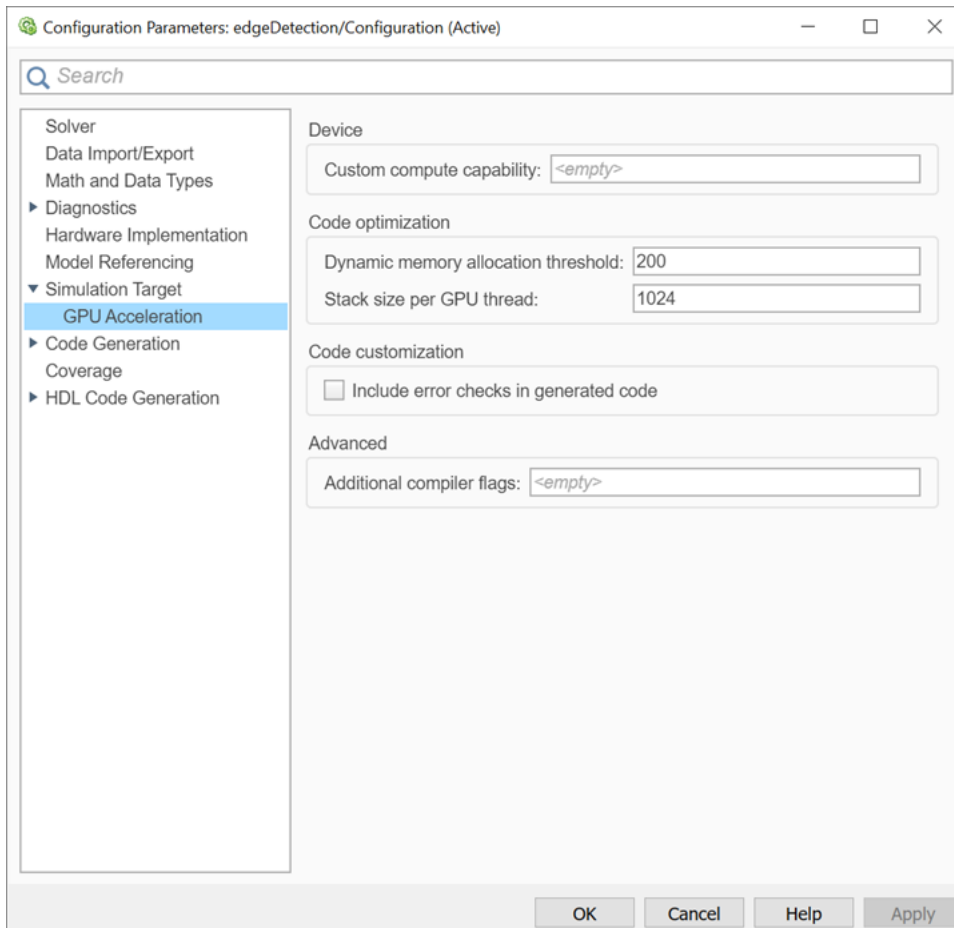
auto

- 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.

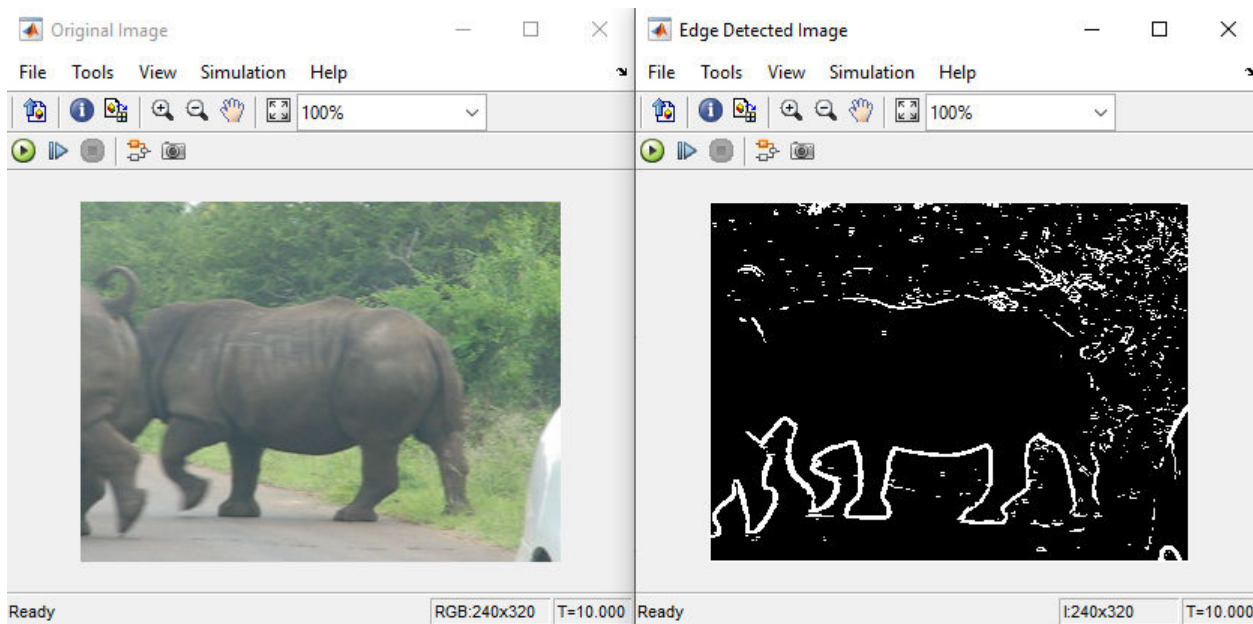
```
set_param('edgeDetection','GPUAcceleration','on');
```

## 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

`bdclose` | `close_system` | `get_param` | `load_system` | `open_system` | `save_system` | `set_param` | `sim` | `slbuild`

## More About

- “Code Generation from Simulink Models with GPU Coder” on page 3-8
- “Deep Learning in Simulink by Using MATLAB Function Block” on page 3-14
- “Deep Learning in Simulink by Using Deep Neural Networks Library” on page 3-22
- “Targeting NVIDIA Embedded Boards” on page 3-29
- “Numerical Equivalence Testing” on page 3-31
- “Parameter Tuning and Signal Monitoring by Using External Mode” on page 3-37
- “GPU Code Generation for Lane Detection in Simulink” on page 3-42
- “GPU Code Generation for a Fog Rectification Simulink Model” on page 3-47

## 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:

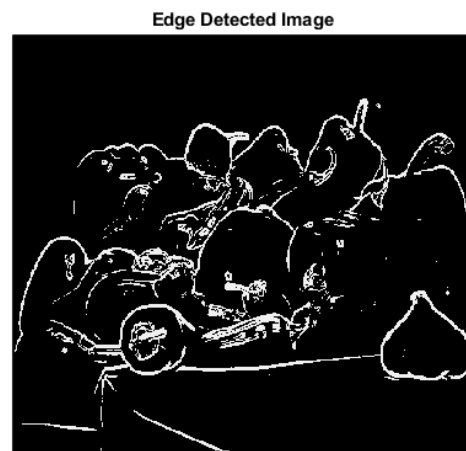
- Create or open a model.
- Configure the model for code generation by selecting the **solver**, **language**, **toolchain**, and other GPU-specific configuration parameters.
- Build the model.

### 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);
```



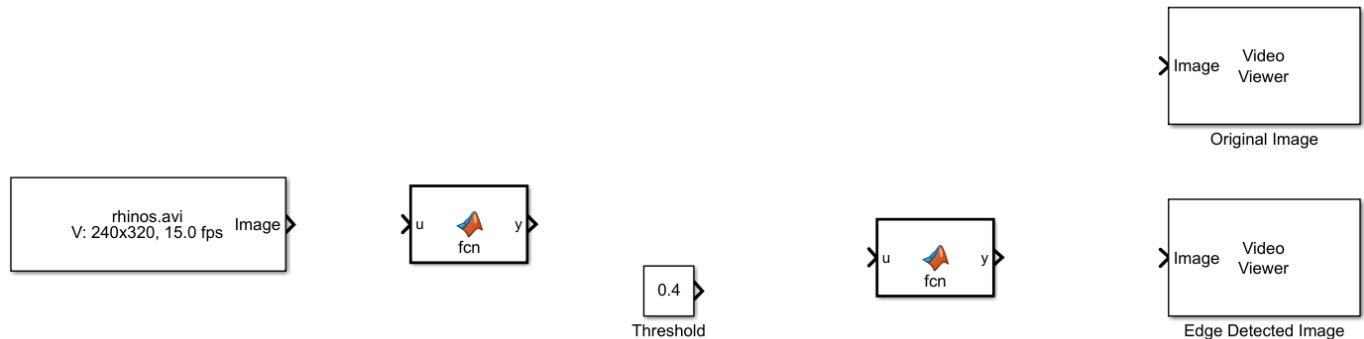
### Create Edge Detection Model

- 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.

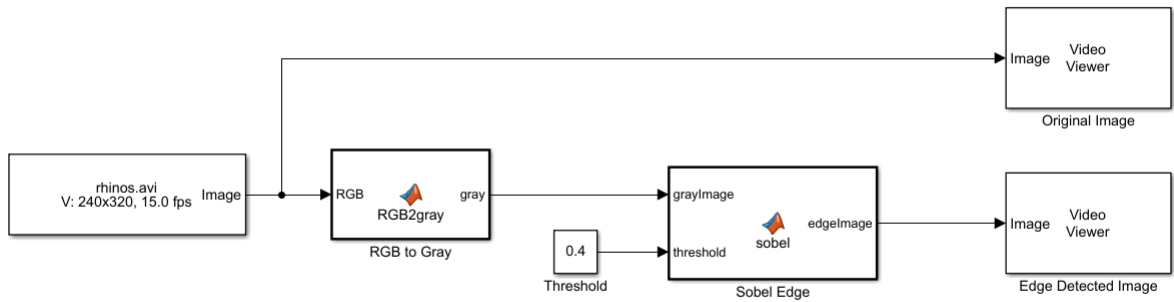
- 9 Modify the other MATLAB Function block to implement the RGB to grayscale conversion prior to the Sobel edge detection operation. Set the **Function packaging** parameter of the MATLAB Function block to **Reusable function**.

```
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)));
```

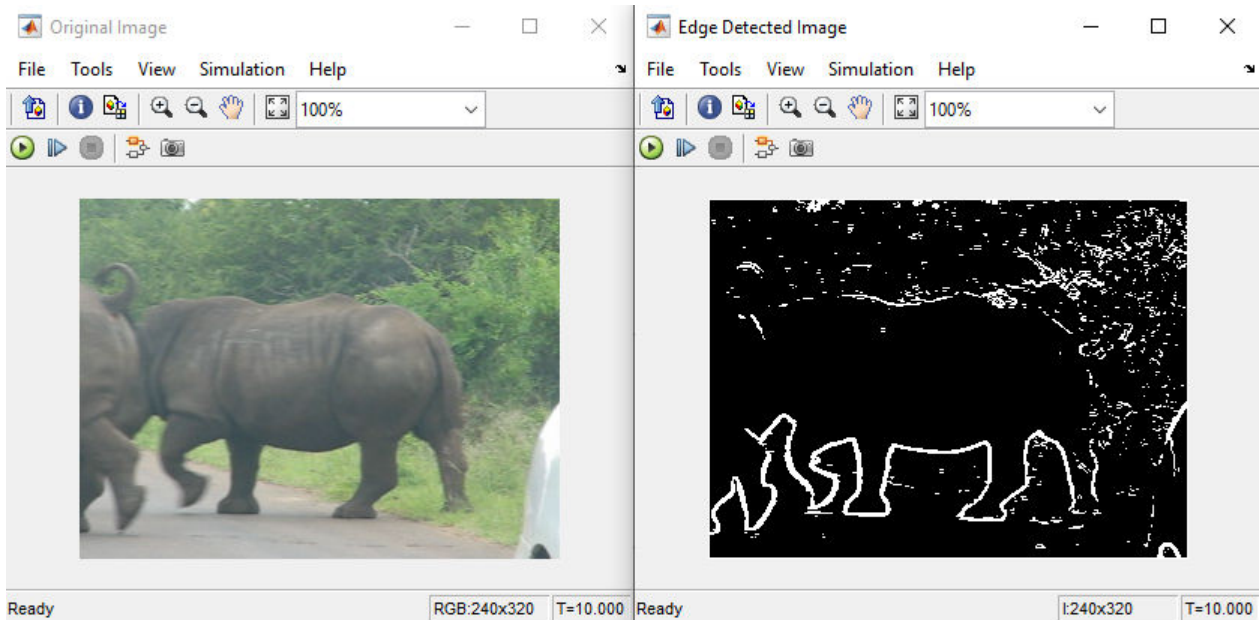
end

- 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
<b>Type</b>	Fixed-step	Maintains a constant (fixed) step size, which is required for code generation
<b>Solver</b>	discrete (no continuous states)	Applies a fixed-step integration technique for computing the state derivative of the model
<b>Fixed-step size</b>	auto	Simulink chooses the step size

#### Solver selection

Type: Fixed-step

Solver: discrete (no continuous states)

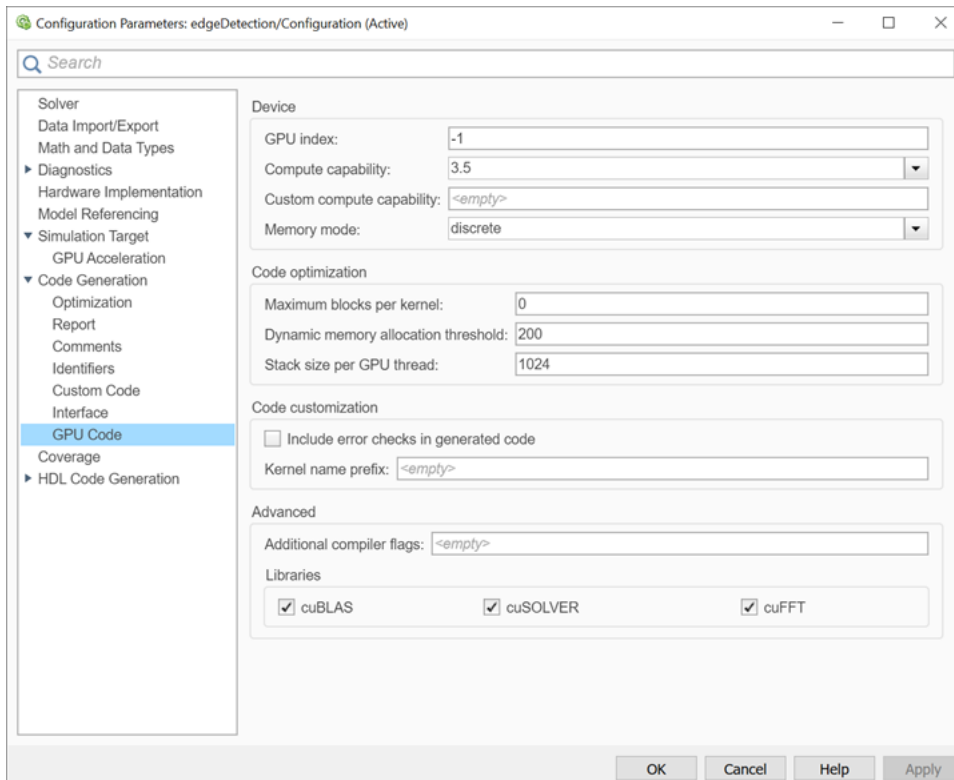
#### ▼ Solver details

Fixed-step size (fundamental sample time):

auto

- 2 On the **Code Generation** pane, set the **System target file** to `grt.tlc`.  
You can also use the Embedded Coder® target file `ert.tlc`.
- 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)**.
- 7 On the **Code Generation > Interface** pane, disable **MAT-file logging**.  
The code generator does not support MAT-file logging when generating CUDA code.
- 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.

## See Also

### Functions

`bdclose` | `close_system` | `get_param` | `load_system` | `open_system` | `save_system` | `set_param` | `sim` | `slbuild`

### More About

- “Simulation Acceleration by Using GPU Coder” on page 3-2
- “Deep Learning in Simulink by Using MATLAB Function Block” on page 3-14
- “Deep Learning in Simulink by Using Deep Neural Networks Library” on page 3-22
- “Targeting NVIDIA Embedded Boards” on page 3-29
- “Numerical Equivalence Testing” on page 3-31
- “Parameter Tuning and Signal Monitoring by Using External Mode” on page 3-37
- “GPU Code Generation for Lane Detection in Simulink” on page 3-42
- “GPU Code Generation for a Fog Rectification Simulink Model” on page 3-47

## Deep Learning in Simulink by 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);
```

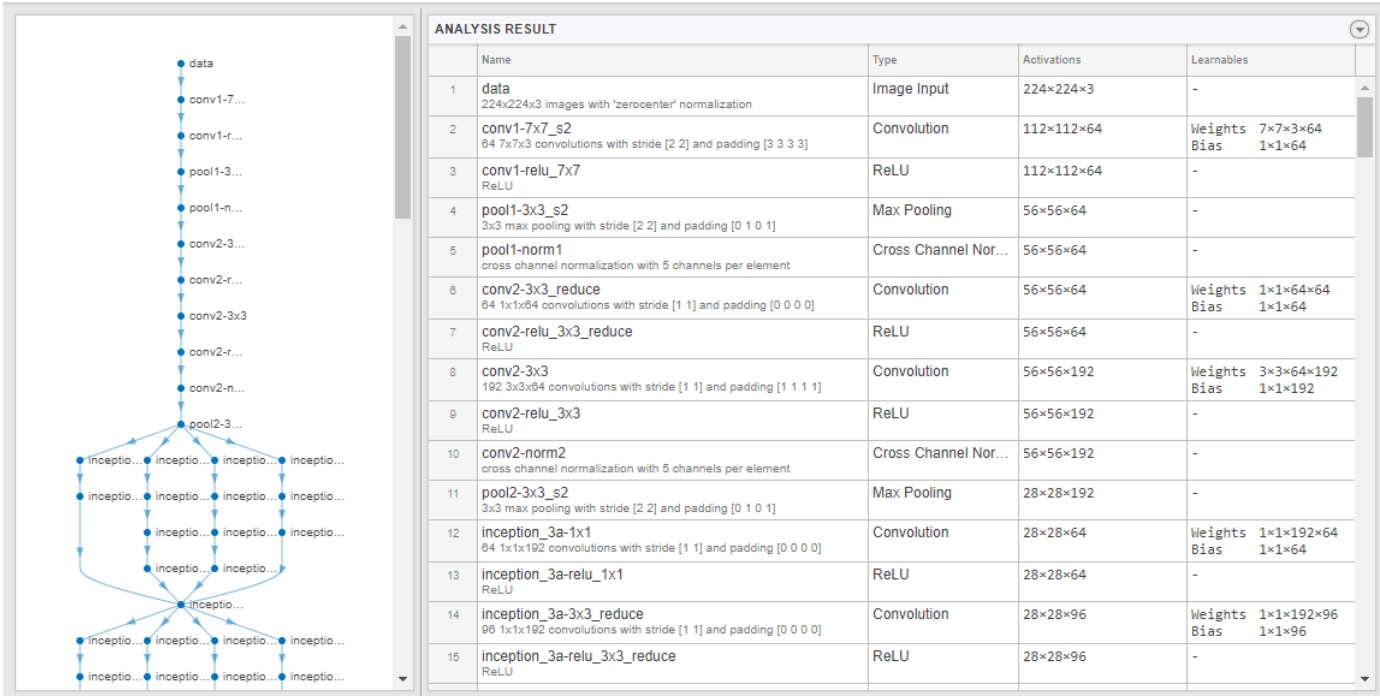
net

Analysis date: 20-Jun-2019 23:27:32

144  layers

0  warnings

0  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.

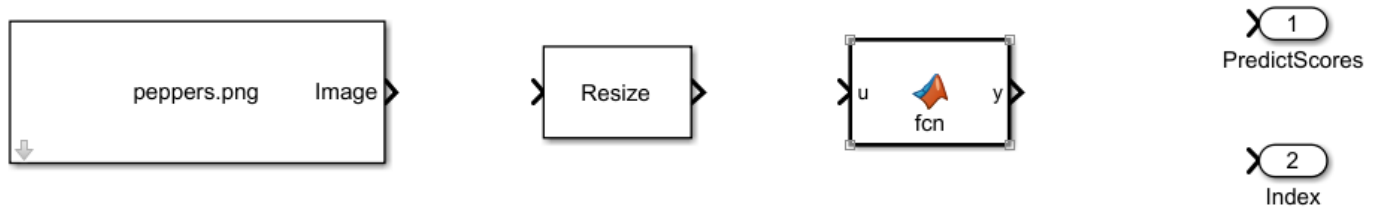
```
classNames = net.Layers(end).Classes;
numClasses = numel(classNames);
disp(classNames(randperm(numClasses,10)))
```

```
'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`.

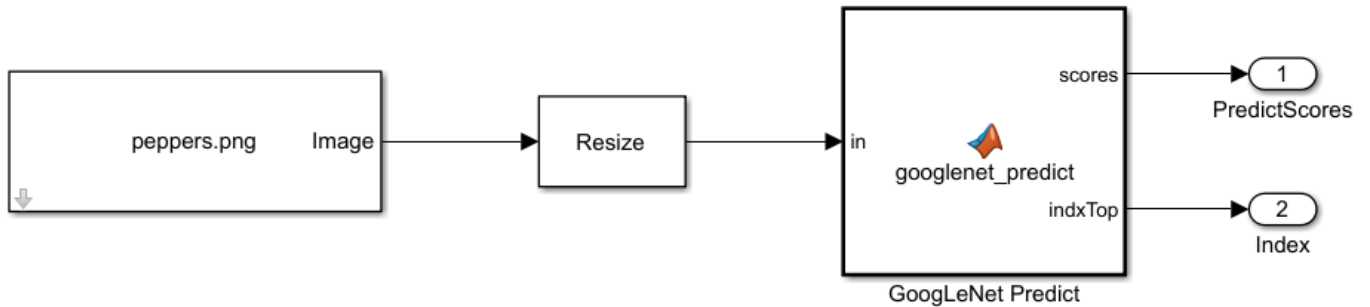
```
out = activations(mynet,in,layerIdx,'OutputAs','Channels');
```

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
<b>Type</b>	Fixed-step	Maintains a constant (fixed) step size, which is required for code generation
<b>Solver</b>	discrete (no continuous states)	Applies a fixed-step integration technique for computing the state derivative of the model
<b>Fixed-step size</b>	auto	Simulink chooses the step size

Solver selection

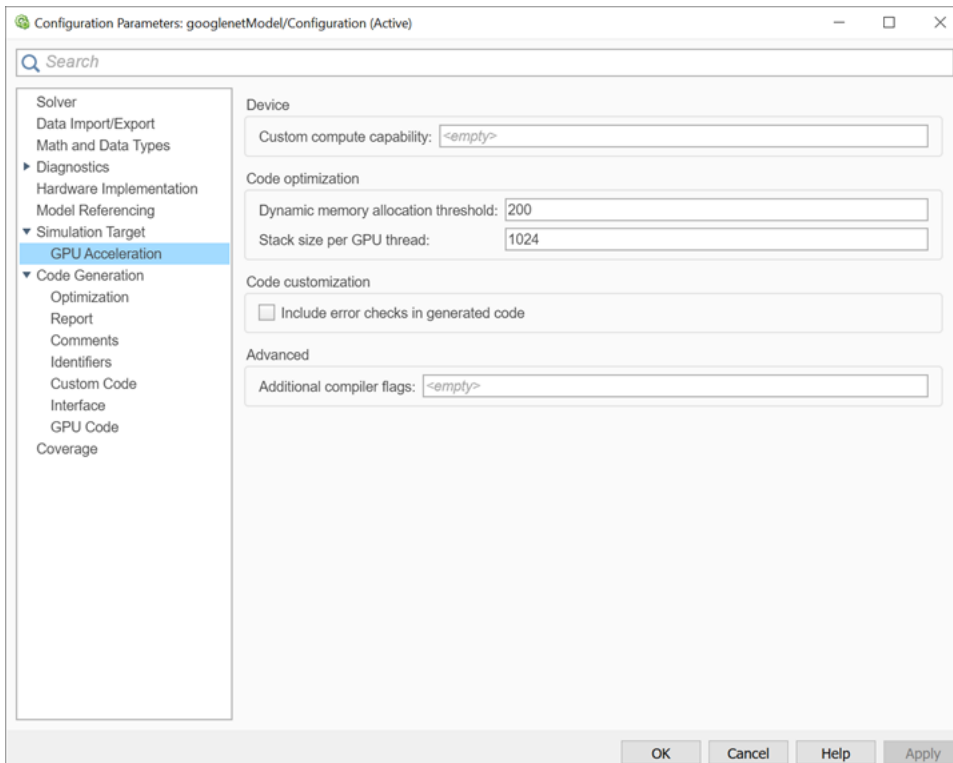
Type: Fixed-step Solver: discrete (no continuous states)

▼ 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



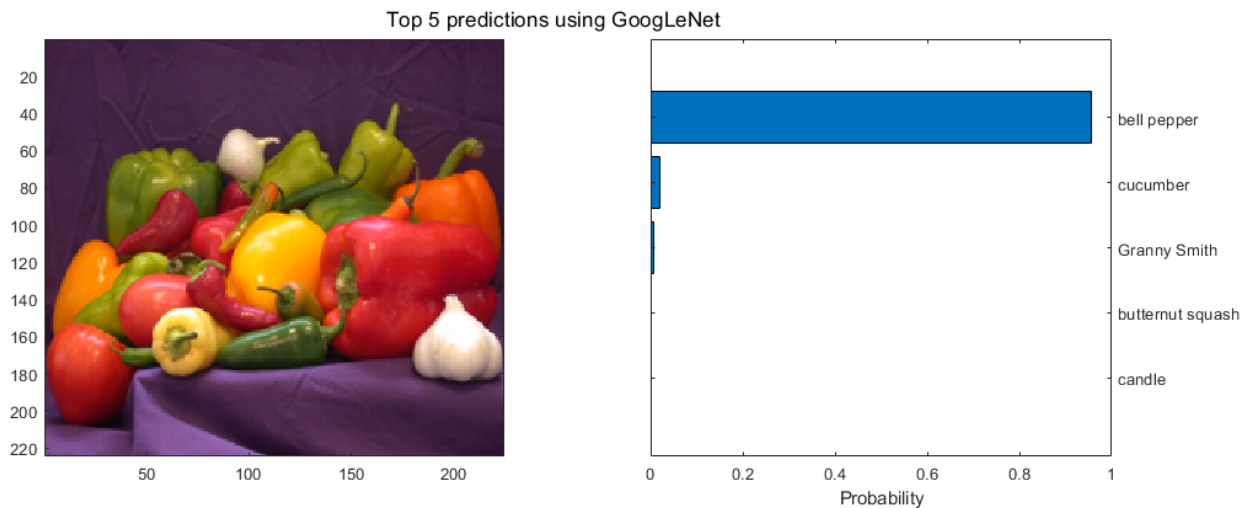
software first generates and compiles the CUDA/C++ code, and then runs the model. The code generation tool places the generated code in a subfolder of the working folder called `s\prj/_slprj/googlenetModel`.

- 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');
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:-1:1,1))
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`.

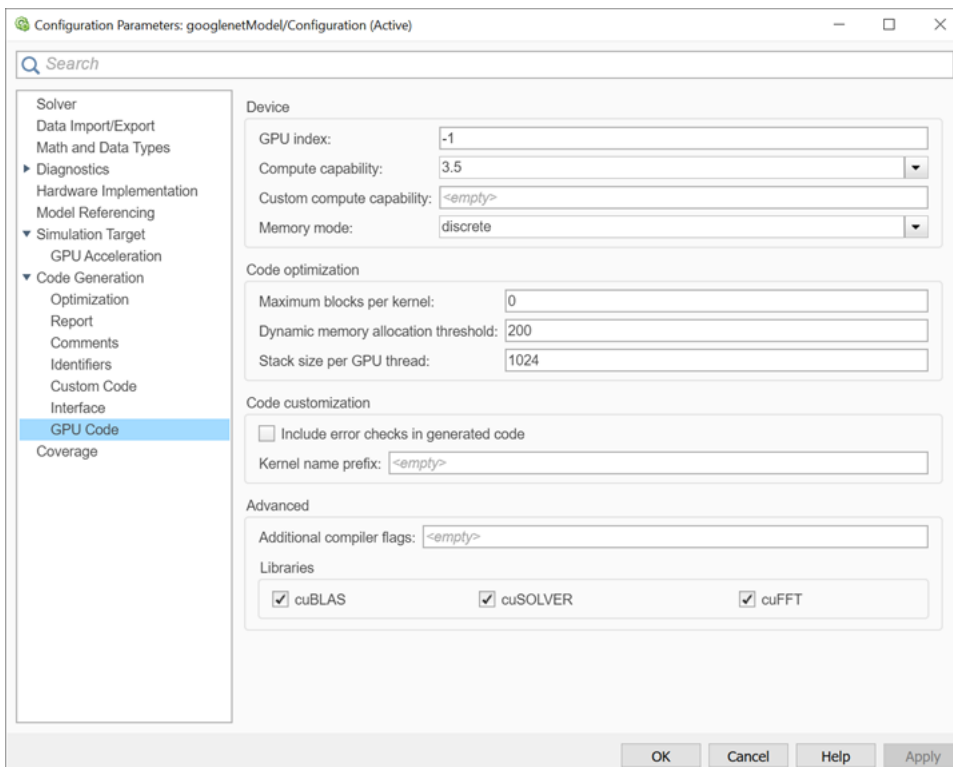
You can also use the Embedded Coder target file, `ert.tlc`.

- 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).
- 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

- Code generation for a deep learning network with custom layer is not supported in Simulink.
- 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

`bdclose` | `close_system` | `get_param` | `load_system` | `open_system` | `save_system` | `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
- “Deep Learning in Simulink by Using Deep Neural Networks Library” on page 3-22
- “Targeting NVIDIA Embedded Boards” on page 3-29
- “Numerical Equivalence Testing” on page 3-31
- “Parameter Tuning and Signal Monitoring by Using External Mode” on page 3-37

## Deep Learning in Simulink by Using 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** library. The **Deep Neural Networks** block library includes:

- **Predict block** — Predict responses using the trained network specified through the block parameter. This block enables loading of a pretrained network into the Simulink model from a MAT-file or from a MATLAB function.

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. This block enables loading of a pretrained network into the Simulink model from a MAT-file or from a MATLAB function.

For more information about working with the Image Classifier block, see “Classify ECG Signals in Simulink Using Deep Learning” (Deep Learning Toolbox).

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.

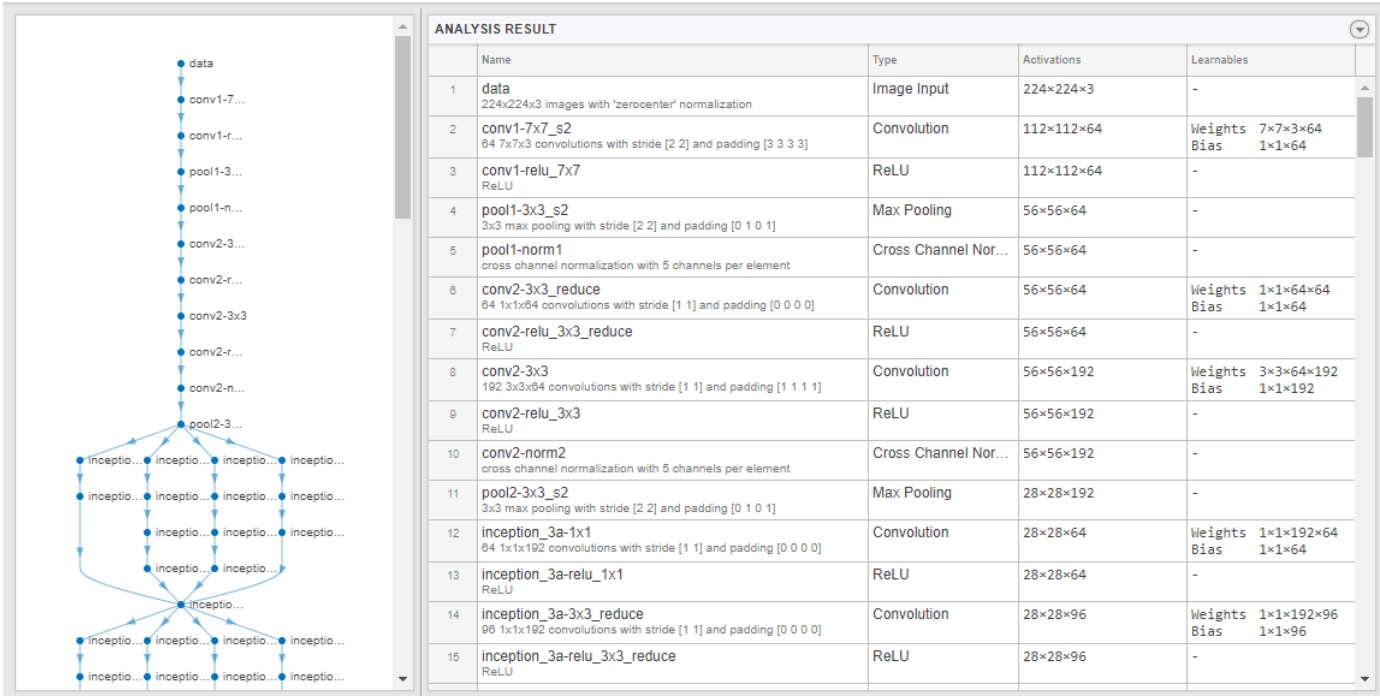
```
analyzeNetwork(net);
```

net

Analysis date: 20-Jun-2019 23:27:32

 144  layers

 0  warnings

 0  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)))

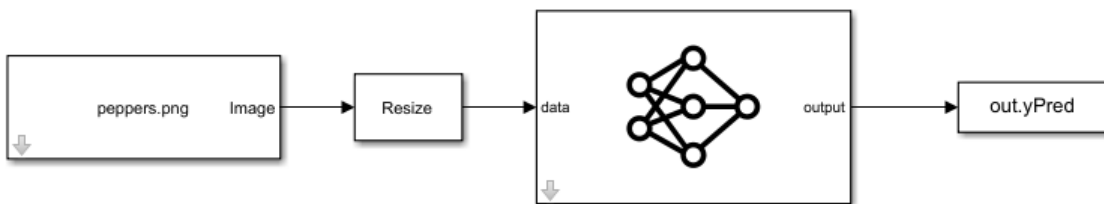
    'speedboat'
    'window screen'
    'isopod'
    'wooden spoon'
    'lipstick'
    'drake'
    'hyena'
    'dumbbell'
    'strawberry'
    'custard apple'
    
```

## Create GoogLeNet Model

- 1 Create a Simulink model and insert a 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
<b>Type</b>	Fixed-step	Maintains a constant (fixed) step size, which is required for code generation
<b>Solver</b>	discrete (no continuous states)	Applies a fixed-step integration technique for computing the state derivative of the model
<b>Fixed-step size</b>	auto	Simulink chooses the step size

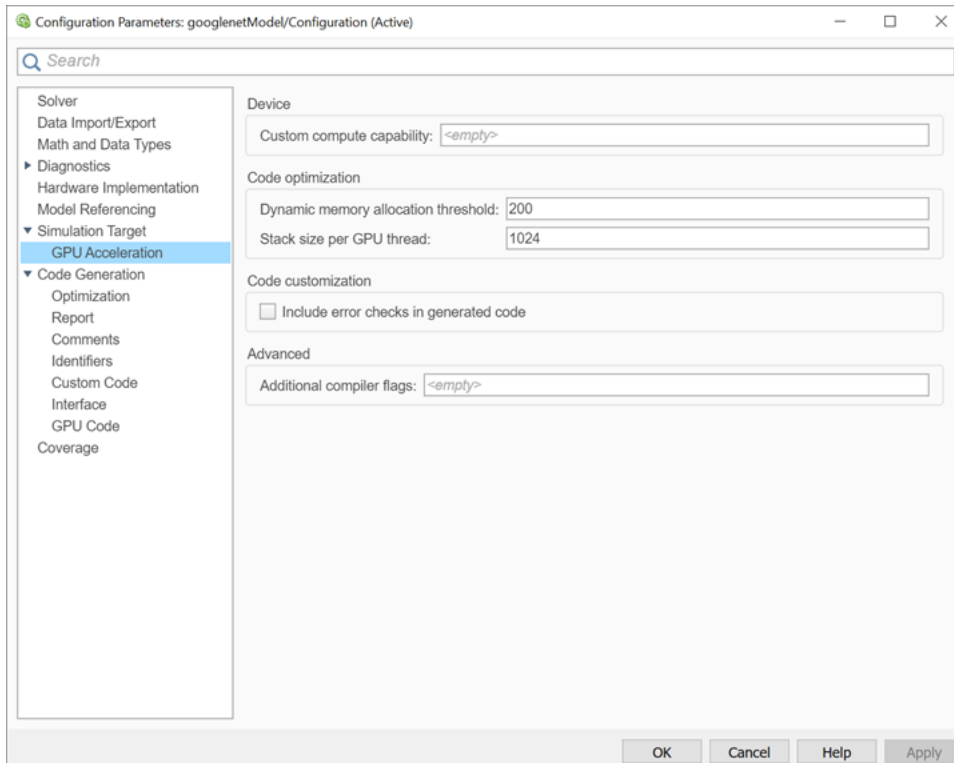
Solver selection

Type:  Solver:

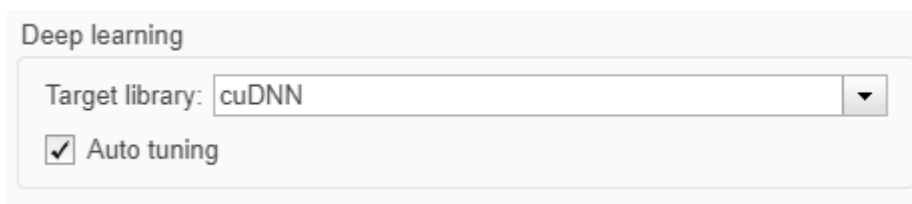
▼ Solver details

Fixed-step size (fundamental sample time):

- 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.



- 4 On the **Simulation Target** pane, set the **Target Library** in the **Deep learning** group to cuDNN. You can also select TensorRT.



- 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');
```

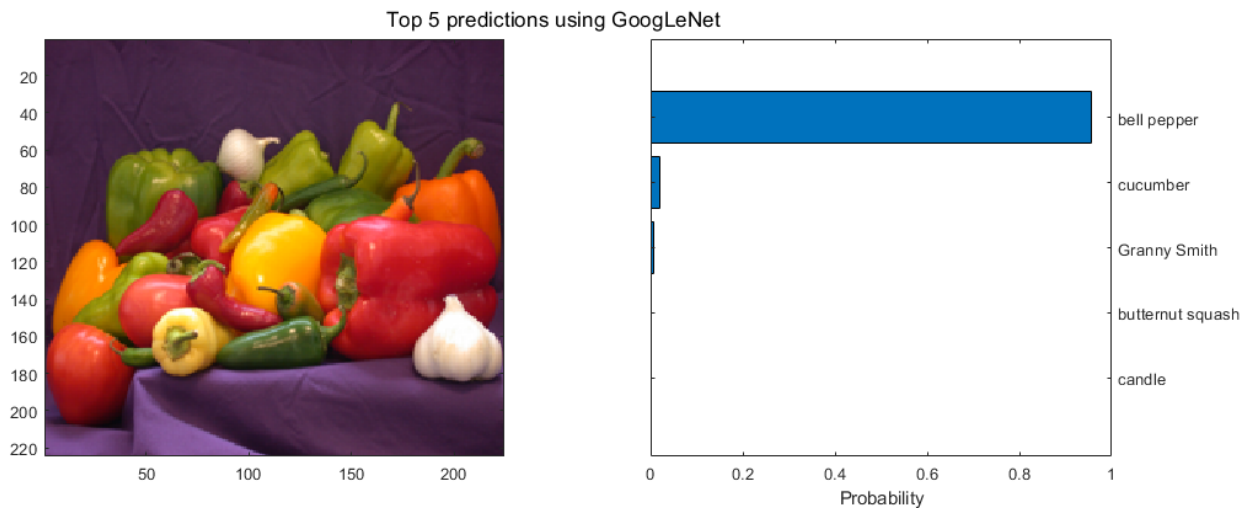
The software first checks to see if CUDA/C++ code was previously compiled for your model. If code was created previously, the software runs the model. If code was not previously built, the software first generates and compiles the CUDA/C++ code, and then runs the model. The code generation tool places the generated code in a subfolder of the working folder called `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');
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)
xlabel(ax2,'Probability')
yticklabels(ax2,classNamesTop(5:-1:1))
ax2.YAxisLocation = 'right';
sgtitle('Top 5 predictions using GoogLeNet')
```



## Configure Model for Code Generation

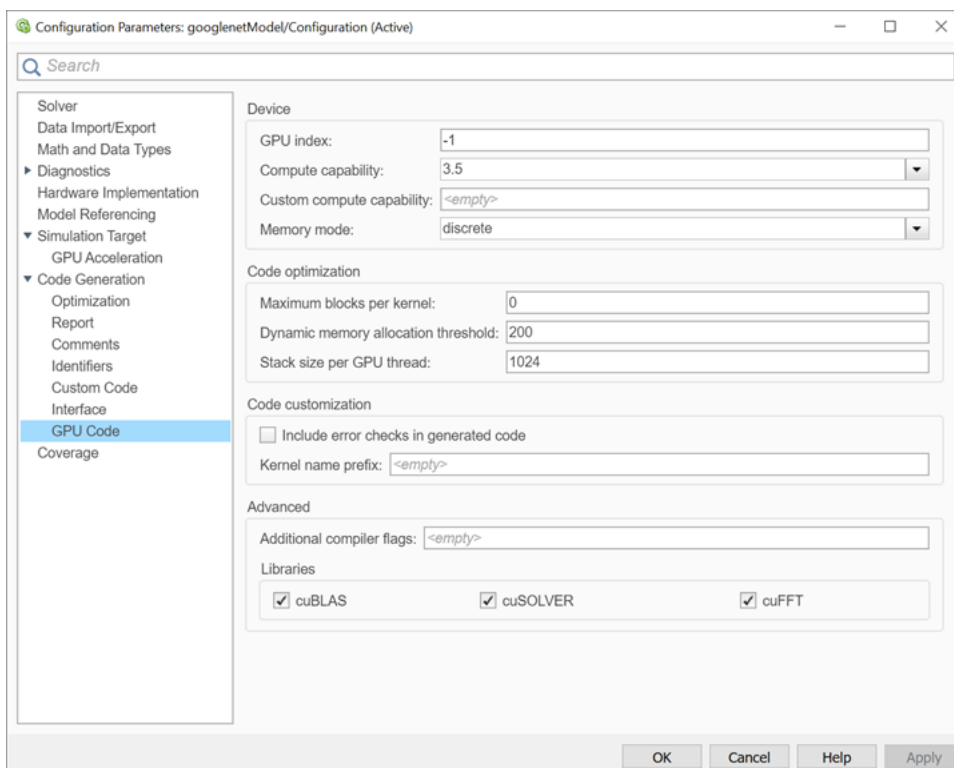
The model configuration parameters provide many options for the code generation and build process.

- 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`.



- 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).
- 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

- Code generation for a deep learning network with custom layer is not supported in Simulink.
- 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.

### See Also

#### Functions

`bdclose` | `close_system` | `get_param` | `load_system` | `open_system` | `save_system` | `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
- “Deep Learning in Simulink by Using MATLAB Function Block” on page 3-14
- “Targeting NVIDIA Embedded Boards” on page 3-29
- “Numerical Equivalence Testing” on page 3-31
- “Parameter Tuning and Signal Monitoring by Using External Mode” on page 3-37
- “Code Generation for a Deep Learning Simulink Model that Performs Lane and Vehicle Detection” on page 3-57
- “Code Generation for a Deep Learning Simulink Model to Classify ECG Signals” on page 3-50

## 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 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.

---

### 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

`bdclose` | `close_system` | `get_param` | `load_system` | `open_system` | `save_system` | `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
- “Deep Learning in Simulink by Using MATLAB Function Block” on page 3-14
- “Deep Learning in Simulink by Using Deep Neural Networks Library” on page 3-22
- “Numerical Equivalence Testing” on page 3-31
- “Parameter Tuning and Signal Monitoring by Using External Mode” on page 3-37

## 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 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.

---

### 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.

---

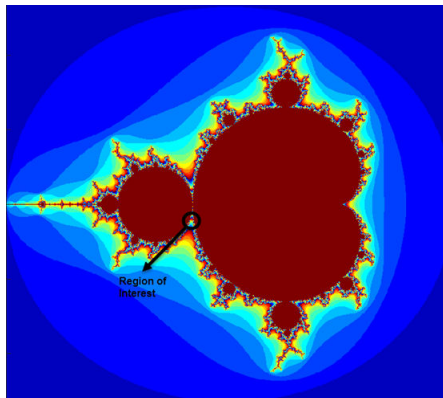
## Example: The Mandelbrot Set

### 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.

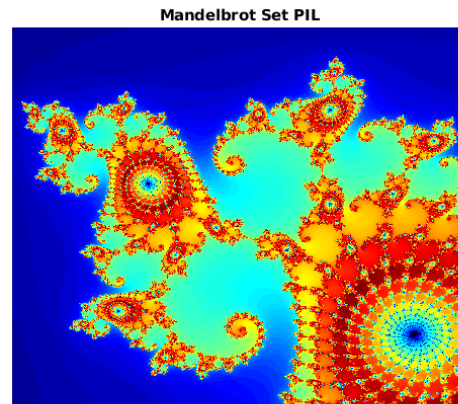
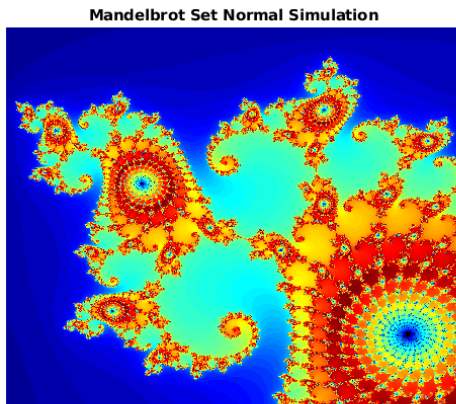
```
1 of 1 models built (0 models already up to date)
Build duration: 0h 0m 22.94s
### Target device has no native communication support. Checking connectivity
configuration registrations...
### Connectivity configuration for component "mandelbrot_top": NVIDIA Jetson ###
PIL execution is using Port 17725.
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

`bdclose` | `close_system` | `get_param` | `load_system` | `open_system` | `save_system` | `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
- “Deep Learning in Simulink by Using MATLAB Function Block” on page 3-14
- “Deep Learning in Simulink by Using Deep Neural Networks Library” on page 3-22
- “Targeting NVIDIA Embedded Boards” on page 3-29
- “Parameter Tuning and Signal Monitoring by Using External Mode” on page 3-37

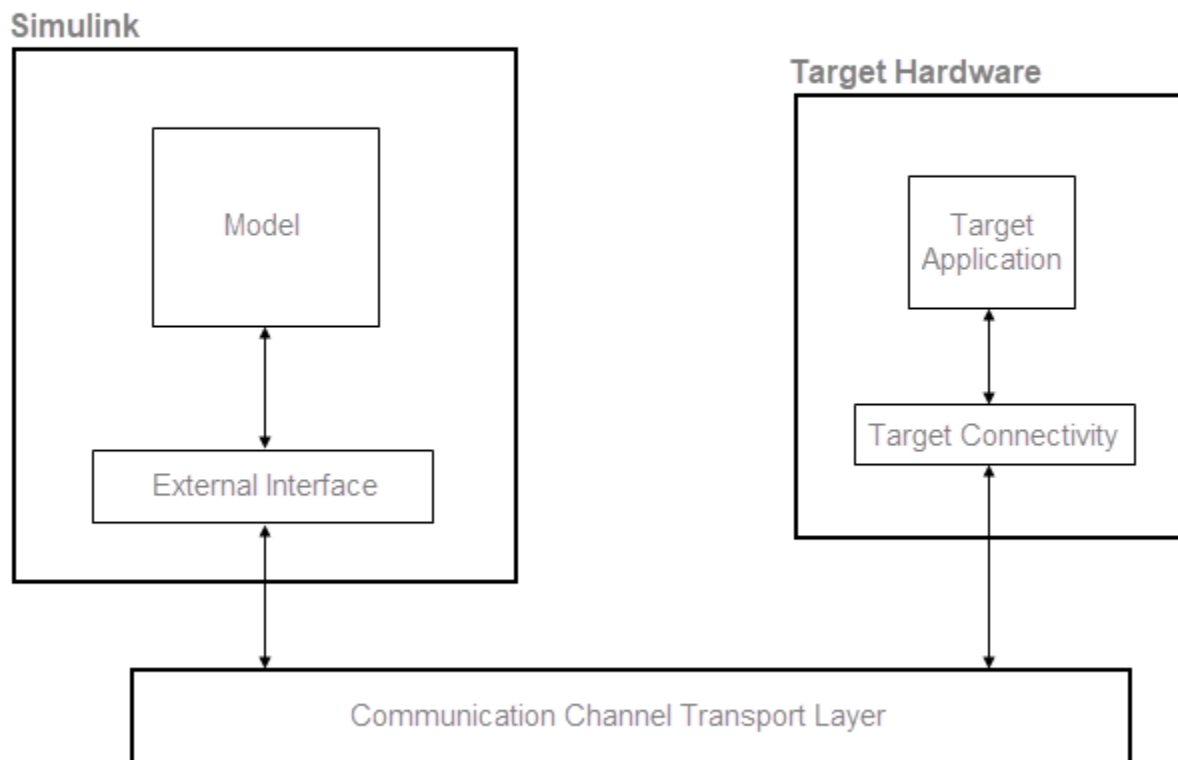
## Parameter Tuning and Signal Monitoring by Using External Mode

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.

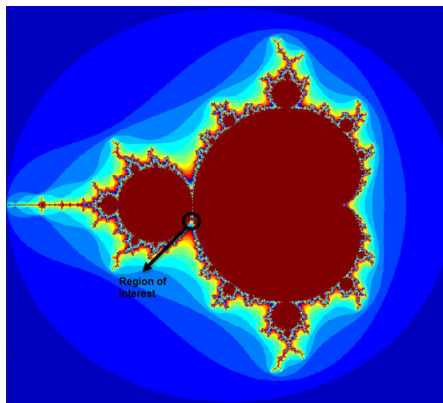
## Example: The Mandelbrot Set

### 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: tcpip MEX-file name: ext\_comm

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

`bdclose` | `close_system` | `get_param` | `load_system` | `open_system` | `save_system` | `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
- “Deep Learning in Simulink by Using MATLAB Function Block” on page 3-14
- “Deep Learning in Simulink by Using Deep Neural Networks Library” on page 3-22
- “Targeting NVIDIA Embedded Boards” on page 3-29
- “Numerical Equivalence Testing” on page 3-31

## 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` (Image Processing Toolbox), `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 2-72.

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.

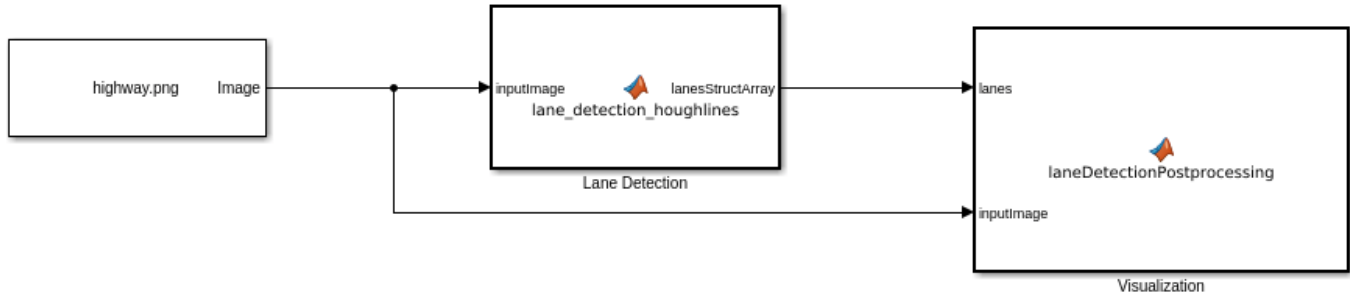
```
envCfg = coder.gpuEnvConfig('host');
envCfg.BasicCodegen = 1;
envCfg.Quiet = 1;
coder.checkGpuInstall(envCfg);
```

### Lane Detection using `houghlines` Simulink Model

The Simulink model for lane detection is shown.

```
open_system('lane_detection');
```



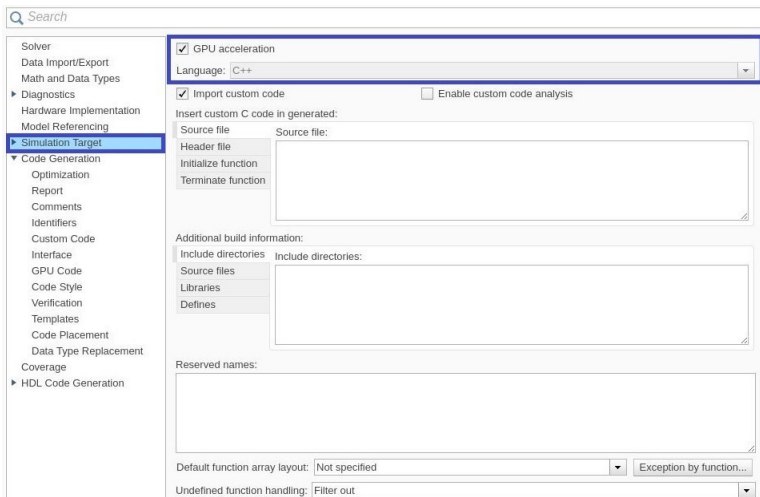


The Lane Detection subsystem contains a MATLAB Function block that takes an intensity image as input and provides detected lanes as output. This function is based on the lane detection algorithm implementation using `houghlines` as described in “Lane Detection on the GPU by Using the houghlines Function” on page 2-72 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.

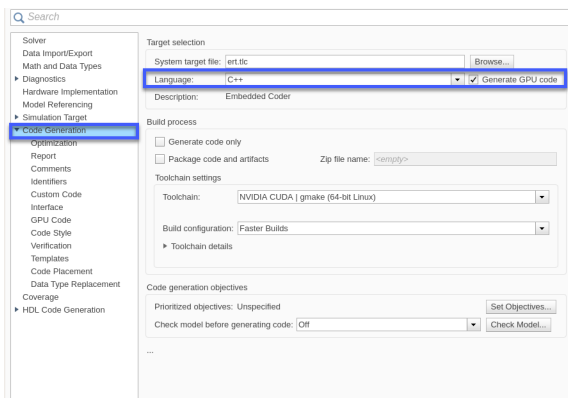
```

set_param('lane_detection', 'SimulationMode', 'Normal');
sim('lane_detection');
  
```

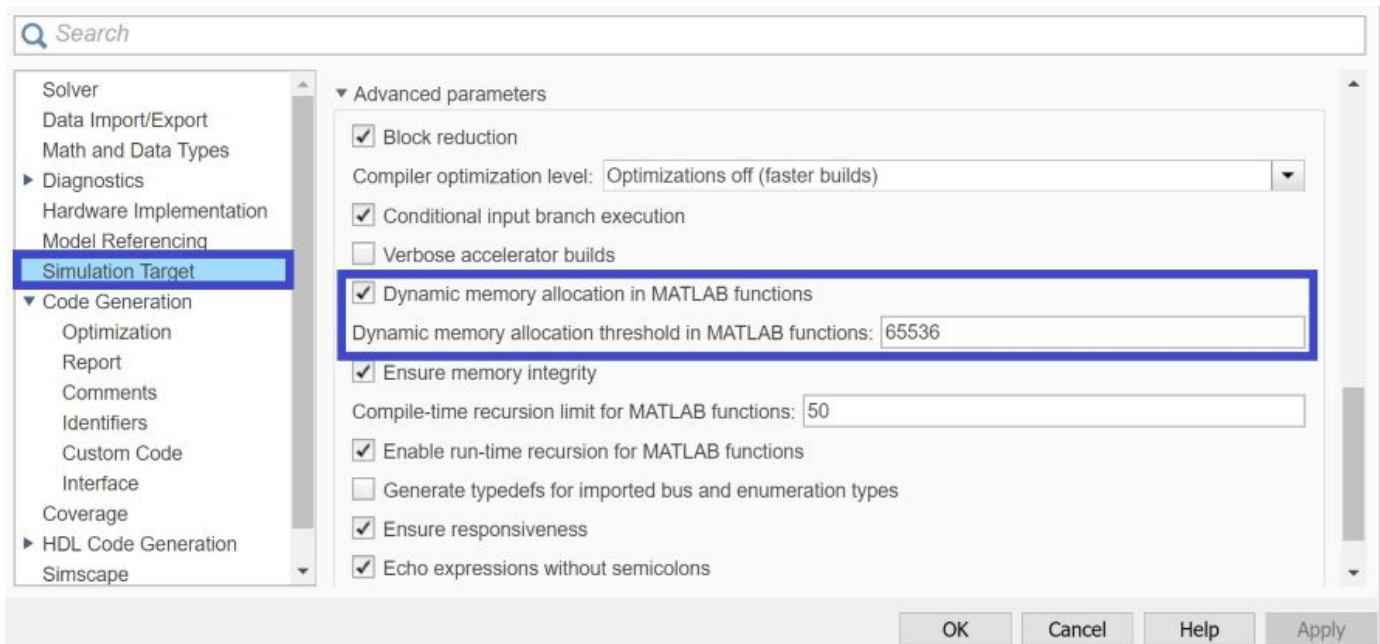


## Generate and Build the Simulink Model

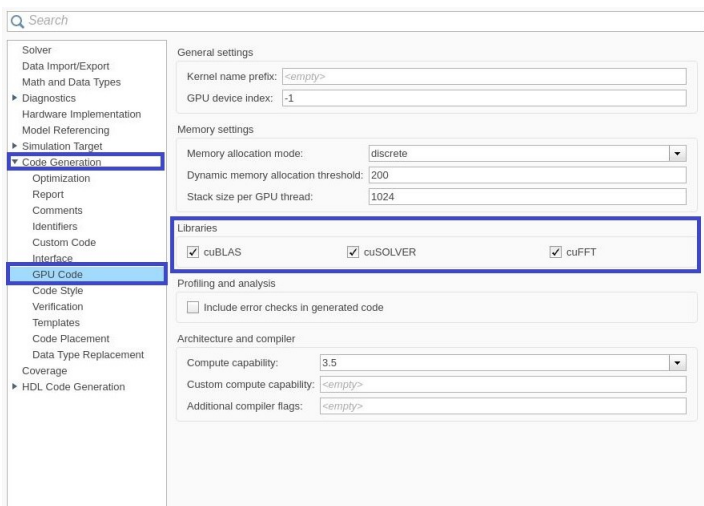
In **Code Generation** pane, select the **Language** as **C++** and enable **Generate GPU code**.



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

[bdclose](#) | [close\\_system](#) | [get\\_param](#) | [load\\_system](#) | [open\\_system](#) | [save\\_system](#) | [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
- “Deep Learning in Simulink by Using MATLAB Function Block” on page 3-14
- “Targeting NVIDIA Embedded Boards” on page 3-29
- “Numerical Equivalence Testing” on page 3-31
- “Parameter Tuning and Signal Monitoring by Using External Mode” on page 3-37

## 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 2-53 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 2-53 example. When the model runs, the `Visualization` block displays the foggy input image and defogged output image.

```
mdl = 'fog_rectification_model';  
open_system(mdl);
```



Copyright 2020 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**



## 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';
```

## 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

`bdclose` | `close_system` | `get_param` | `load_system` | `open_system` | `save_system` | `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
- “Deep Learning in Simulink by Using MATLAB Function Block” on page 3-14
- “Targeting NVIDIA Embedded Boards” on page 3-29
- “Numerical Equivalence Testing” on page 3-31
- “Parameter Tuning and Signal Monitoring by Using External Mode” on page 3-37

## 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).

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:

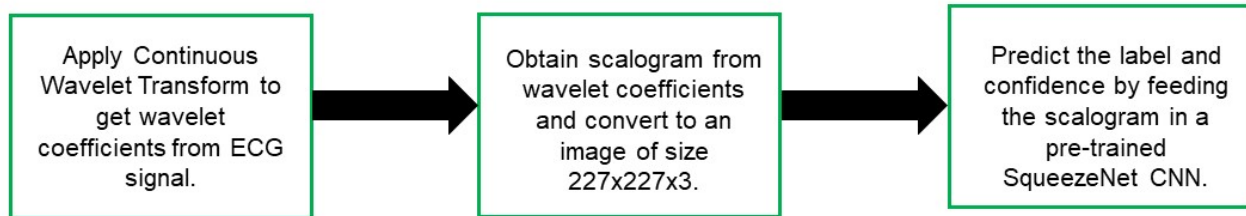
- 1 Persons with cardiac arrhythmia (ARR)
- 2 Persons with congestive heart failure (CHF)
- 3 Persons with normal sinus rhythms (NSR)

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

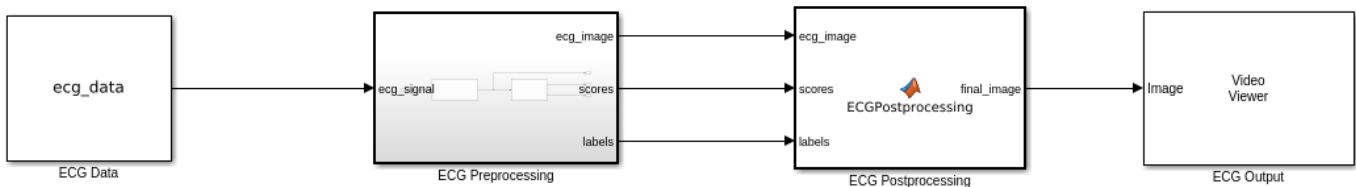
The block diagram for the algorithmic workflow of the Simulink model is shown.



## 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');
```

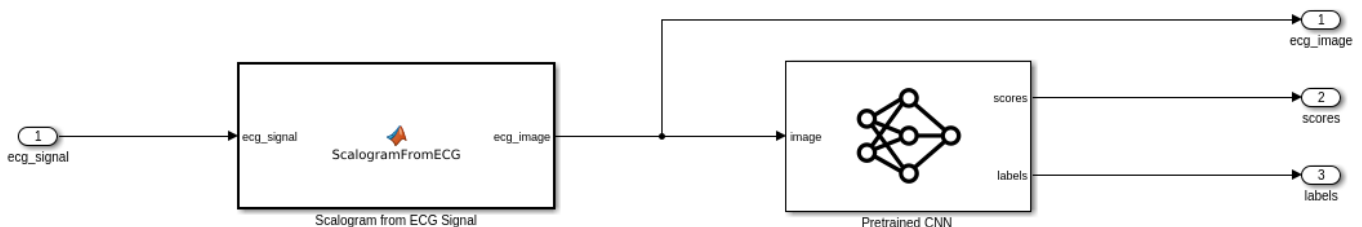


Copyright 2020 The MathWorks, Inc.

## 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 = colourmap(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 The MathWorks, Inc.

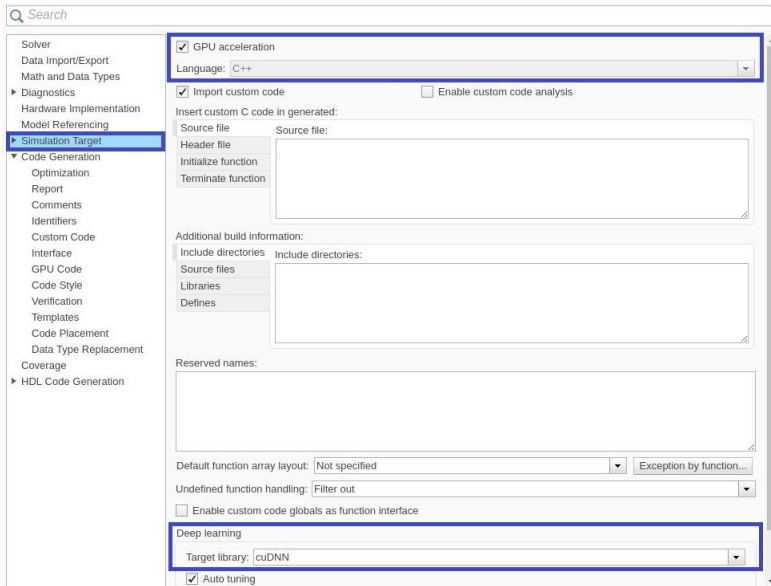
scores = double(scores);
% Obtain maximum confidence
[prob,index] = max(scores);
confidence = prob*100;
% Obtain label corresponding to maximum confidence
label = erase(char(labels(index)),'_label');
text = cell(2,1);
text{1} = ['Classification: ' label];
text{2} = ['Confidence: ' sprintf('%0.2f',confidence) '%'];
position = [135 20 0 0; 130 40 0 0];
final_image = insertObjectAnnotation(ecg_image,'rectangle',position,text,'TextBoxOpacity',0.9,'F

end
```

#### Run the Simulation

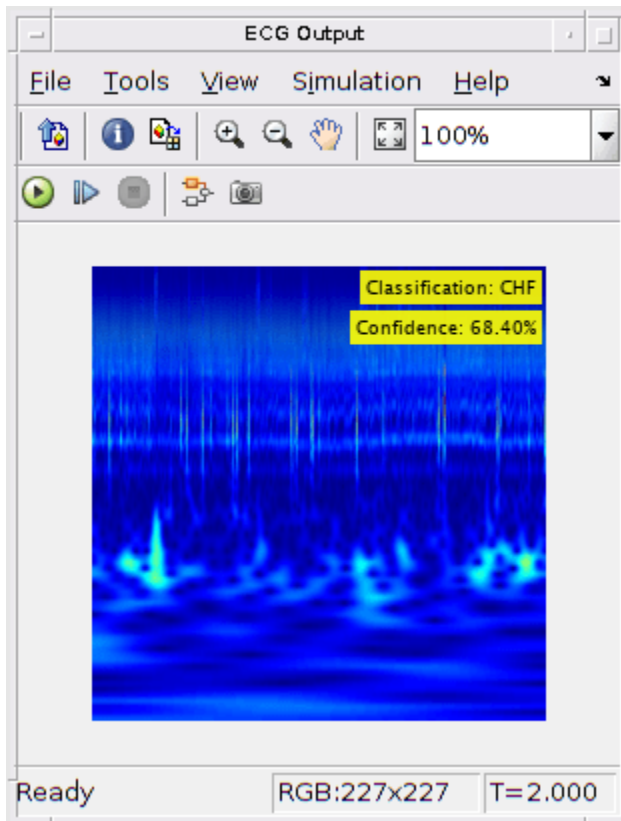
Open Configuration Parameters dialog box.

In **Simulation Target** pane, select **GPU acceleration**. In the **Deep Learning** group, select the target library as **cuDNN**.



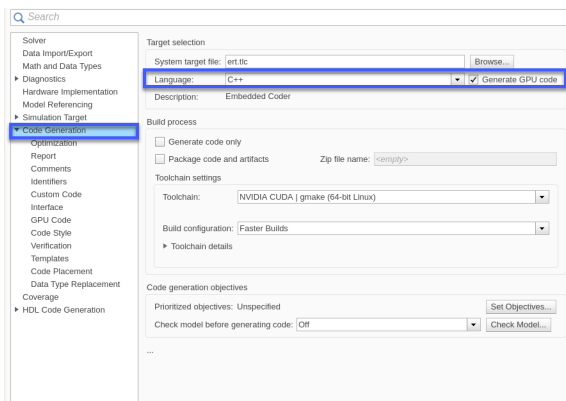
To verify the algorithm and display the labels and confidence score of the test ECG signal loaded in the workspace, run the simulation.

```
set_param('ecg_dl_cwt', 'SimulationMode', 'Normal');
sim('ecg_dl_cwt');
```

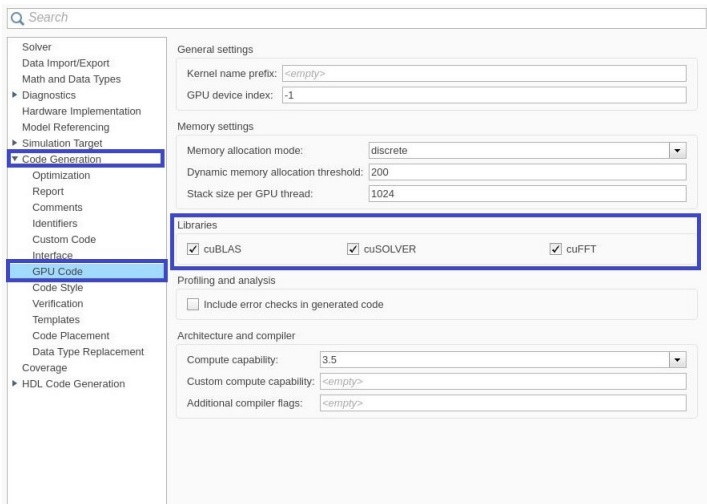


#### Generate and Build the Simulink Model

In **Code Generation** pane, select the **Language** as **C++** and enable **Generate GPU code**.

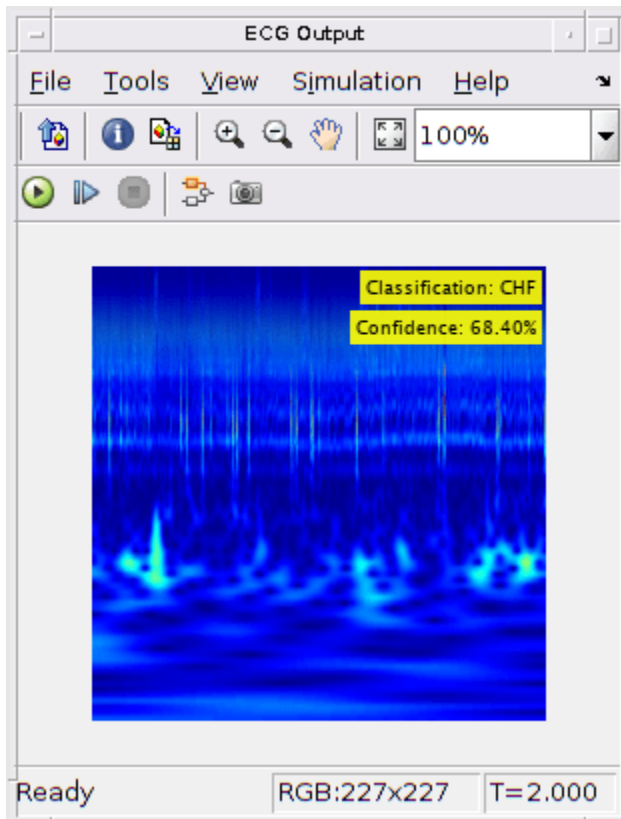


Open **Code Generation > GPU Code** pane. In the subcategory **Libraries**, enable **cuBLAS**, **cuSOLVER** and **cuFFT**.



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_d_l_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_d_l_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_d_l_cwt0_h
#define RTW_HEADER_trainedNet0_ecg_d_l_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_d_l_cwt0
{
public:
    int32_T numLayers;
private:
    MwTensorBase *inputTensors;
    MwTensorBase *outputTensors;
public:
    MwCNLayer *Layers[42];
private:
    MwTargetNetworkImpl *targetImpl;
    void allocate();
    void postsetup();
public:
    trainedNet0_ecg_d_l_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_d_l_cwt0();
};

#endif // RTW_HEADER_trainedNet0_ecg_d_l_cwt0_h
```

## Cleanup

Close the Simulink model.

```
close_system('ecg_d_l_cwt/ECG Preprocessing');
close_system('ecg_d_l_cwt');
```

## See Also

### Functions

`bdclose` | `close_system` | `get_param` | `load_system` | `open_system` | `save_system` | `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
- “Deep Learning in Simulink by Using MATLAB Function Block” on page 3-14
- “Targeting NVIDIA Embedded Boards” on page 3-29
- “Numerical Equivalence Testing” on page 3-31
- “Parameter Tuning and Signal Monitoring by Using External Mode” on page 3-37

## 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 5-100. 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”.

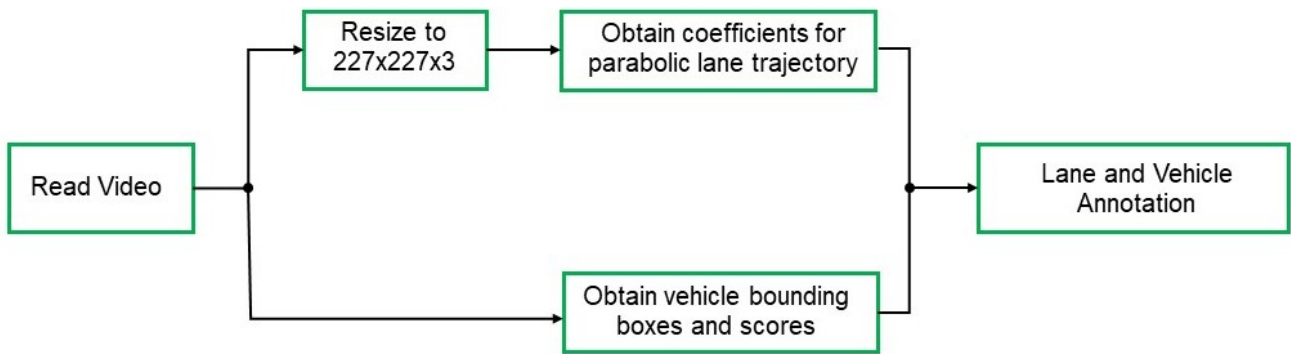
### Verify GPU Environment

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.



### Download Example Video

```

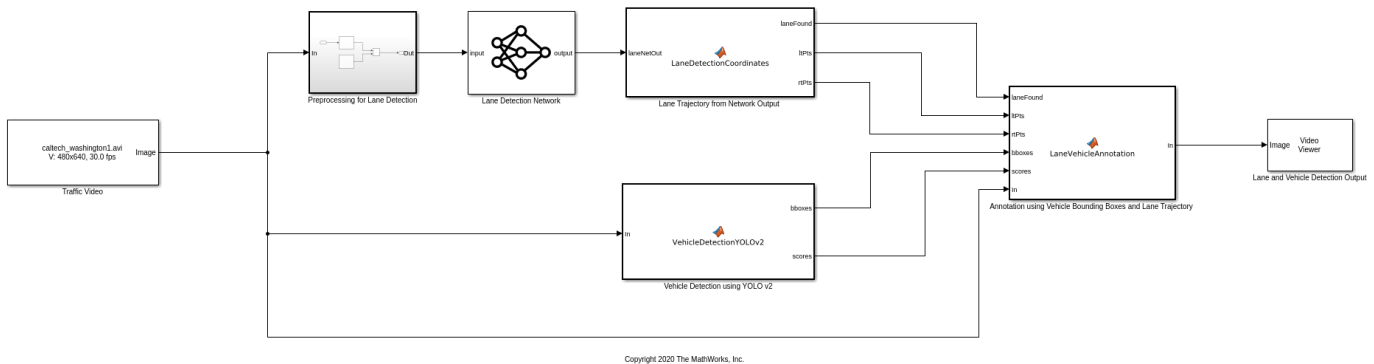
if ~exist('./caltech_washington1.avi', 'file')
    url = 'https://www.mathworks.com/supportfiles/gpuocder/media/caltech_washington1.avi';
    websave('caltech_washington1.avi', url);
end
  
```

### 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');
  
```



### 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 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;

isRightLaneFound = abs(params(6)) > 0.5; %c should be more than 0.5 for it to be a right lane
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
```

### 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

VehicleDetectionYOLOv2 MATLAB function block defines a function `vehicle_detection_yolo_v2` which loads a pretrained YOLO v2 object detector. This network takes an image as input and outputs the bounding box coordinates along with the confidence scores for vehicles in the image for subsequent annotation.

```
type vehicle_detection_yolo_v2
```

```
function [bboxes,scores] = vehicle_detection_yolo_v2(In)

% Copyright 2020 The MathWorks, Inc.

persistent yolodetector;
if isempty(yolodetector)
    yolodetector = coder.loadDeepLearningNetwork('yolov2ResNet50VehicleExample.mat');
end

[bboxes,scores,~] = yolodetector.detect(In, 'threshold', .2);

end
```

### **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 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
```

## Get Pretrained Lane and Vehicle Detection Networks

The function downloads the `trainedLaneNet.mat` and `yolov2ResNet50VehicleExample.mat` files if they are not already present.

```
getVehicleDetectionAndLaneDetectionNetworks()
```

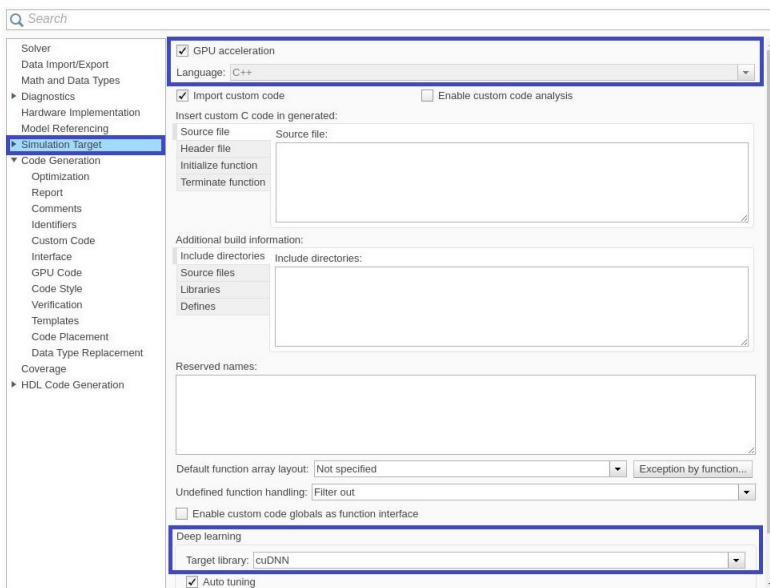
```
Downloading pretrained lane detection network (143 MB)...
Downloading pretrained vehicle detection network (98 MB)...
```

## Run the Simulation

Open Configuration Parameters dialog box.

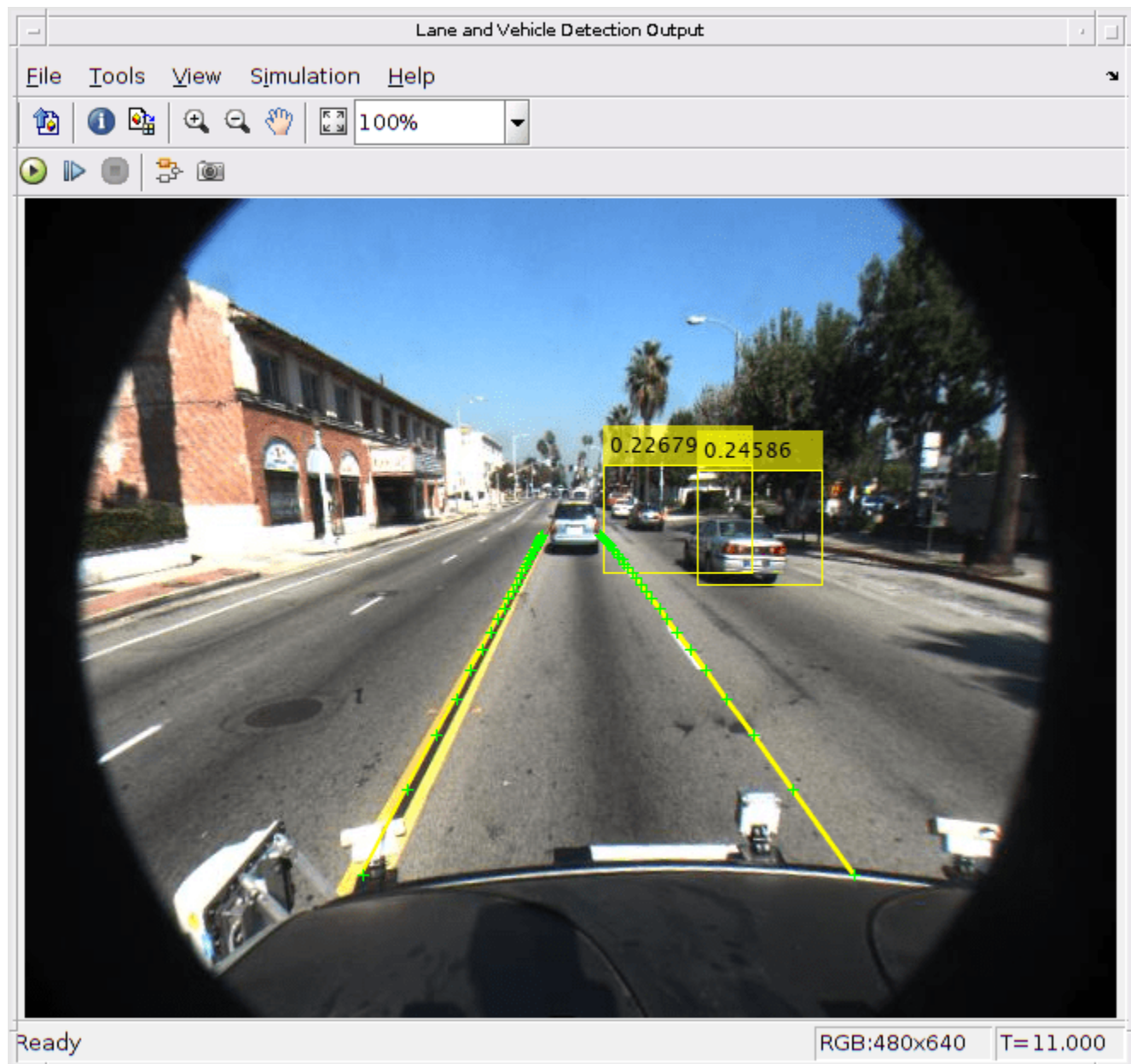
In **Simulation Target** pane, select **GPU acceleration**. In the **Deep Learning** group, select the target library as **cuDNN**.

```
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.

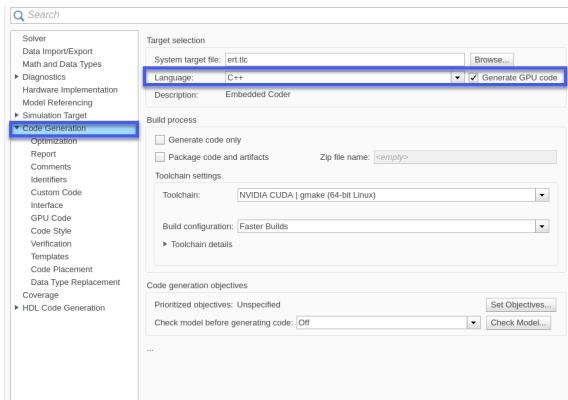
```
set_param('laneAndVehicleDetection', 'SimulationMode', 'Normal');
sim('laneAndVehicleDetection');
```



#### Generate and Build the Simulink Model

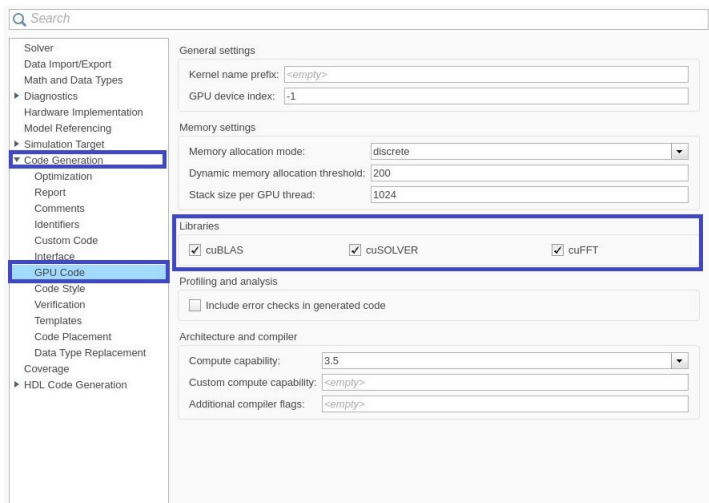
In **Code Generation** pane, select the **Language** as **C++** and enable **Generate GPU code**.

```
set_param(bdroot, 'TargetLang', 'C++');  
set_param(bdroot, 'GenerateGPUCode', 'CUDA');
```



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 "cmn_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 *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 "cmn_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_ */

```

## Cleanup

Close the Simulink model.

```
close_system('laneAndVehicleDetection/Lane and Vehicle Detection Output');  
close_system('laneAndVehicleDetection');
```

## See Also

### Functions

[bdclose](#) | [close\\_system](#) | [get\\_param](#) | [load\\_system](#) | [open\\_system](#) | [save\\_system](#) | [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
- “Deep Learning in Simulink by Using MATLAB Function Block” on page 3-14
- “Targeting NVIDIA Embedded Boards” on page 3-29
- “Numerical Equivalence Testing” on page 3-31
- “Parameter Tuning and Signal Monitoring by Using External Mode” on page 3-37





# Troubleshooting

---

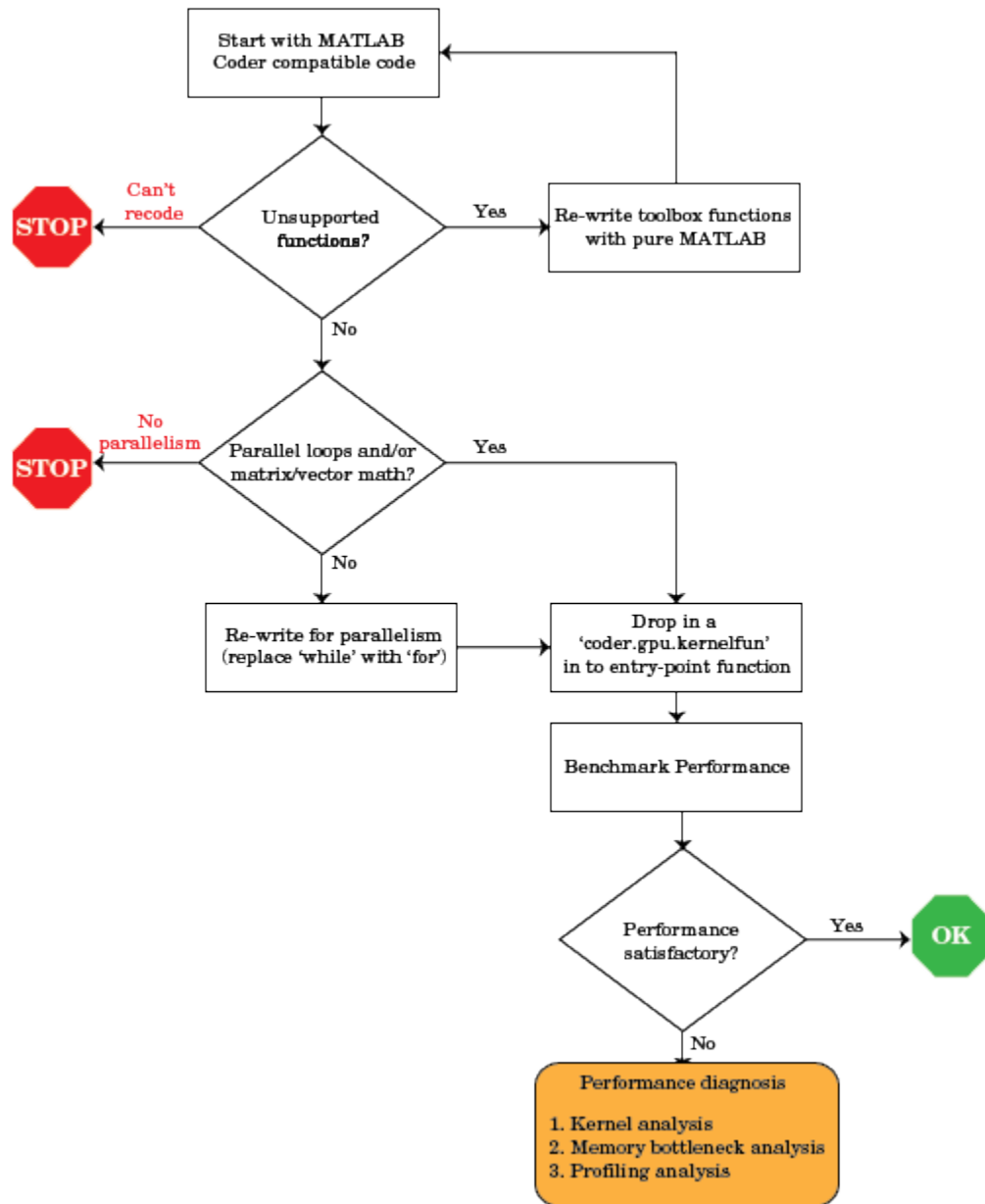
Three of the most common reasons why GPU Coder generated code is not performing as expected are:

- 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 4-5
- “Kernel Analysis” on page 4-18

- “Memory Bottleneck Analysis” on page 4-22
- “Analyze Execution Profiles of the Generated Code” on page 4-24

## Code Generation Reports

In this section...
“Report Generation” on page 4-5
“Report Location” on page 4-6
“Errors and Warnings” on page 4-6
“Files and Functions” on page 4-6
“MATLAB Source” on page 4-6
“Generated Code” on page 4-8
“MATLAB Variables” on page 4-8
“Tracing Code” on page 4-9
“Code Insights” on page 4-10
“Additional Reports” on page 4-10
“Report Limitations” on page 4-10

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

<b>abs(z)</b>	
Size:	1000 × 1000
Class:	double
Complex:	No
GPU Kernel Function:	<a href="#">mandelbrot_count_kernel2</a>

## 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

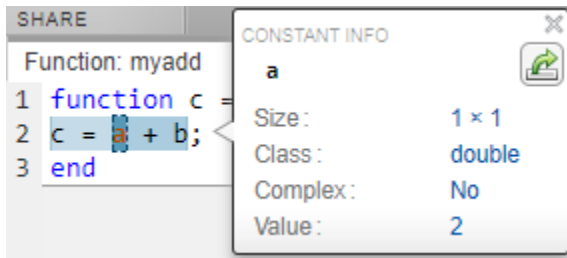
```

EXPRESSION INFO


<b>myExtrinsic(a,b)</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 4-11.

## 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 4-15
- “Trace Between Generated CUDA Code and MATLAB Source Code” on page 4-11
- “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' and contains navigation buttons (Back, Forward, Go To), a Find search bar, and buttons for Trace Code, Edit In MATLAB, and Package Code. Below this is a 'MATLAB SOURCE' section with a 'Function List' and 'Call Tree' pane on the left. The main editor displays the MATLAB source code for 'mandelbrot\_count.m'. A green 'GPU' marker is next to the function name. The code includes a function definition and a loop. A tooltip titled 'EXPRESSION INFO' is shown over the expression 'abs(z)' on line 14, displaying its size (1000 x 1000), class (double), and the GPU kernel function 'mandelbrot\_count\_kernel2'. The 'GENERATED CODE' section at the bottom 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.CodeConfig` | `coder.EmbeddedCodeConfig` | `coder.gpuConfig`

## Related Examples

- “Code Generation by Using the GPU Coder App”
- “Code Generation Using the Command Line Interface”
- “Code Generation Reports” on page 4-5
- “Generating a GPU Code Metrics Report for Code Generated from MATLAB Code” on page 4-15

## 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 4-5.

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

- 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
<a href="#">mandelbrot_count_kernel3</a>	[512,1,1]	[1954,1,1]		gpu_count	0	0	1	0	None
<a href="#">mandelbrot_count_kernel2</a>	[512,1,1]	[1954,1,1]	gpu_z0	gpu_count,gpu_z	0	0	1	0	None
<a href="#">mandelbrot_count_kernel1</a>	[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.

- 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
count	gpu_count	8000000	device->host	NO_ENCLOSING_CONDITION	0
gpu_xGrid	xGrid	8000000	host->device	NO_ENCLOSING_CONDITION	0
gpu_yGrid	yGrid	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.CodeConfig` | `coder.EmbeddedCodeConfig` | `coder.gpuConfig`

## More About

- “Code Generation Reports” on page 4-5
- “Interactively Trace Between MATLAB Code and Generated C/C++ Code” (Embedded Coder)
- “Trace Between Generated CUDA Code and MATLAB Source Code” on page 4-11
- “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 4-18  
 “For-Loops with Break” on page 4-19  
 “Dependence Analysis Parallel Loop Check Fails” on page 4-19  
 “Logical Indexing of Arrays” on page 4-20  
 “Unsupported Functions” on page 4-20  
 “Loop Interchange” on page 4-20

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 4-5
- “Trace Between Generated CUDA Code and MATLAB Source Code” on page 4-11
- “Generating a GPU Code Metrics Report for Code Generated from MATLAB Code” on page 4-15
- “Memory Bottleneck Analysis” on page 4-22
- “Analyze Execution Profiles of the Generated Code” on page 4-24

## Memory Bottleneck Analysis

### In this section...

“Data Alignment” on page 4-22

“Small Data Sizes” on page 4-22

“Too Many cudaMemcpys” on page 4-22

“Constant Inputs” on page 4-22

“Stack Memory Usage” on page 4-23

### 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 4-5
- “Trace Between Generated CUDA Code and MATLAB Source Code” on page 4-11
- “Generating a GPU Code Metrics Report for Code Generated from MATLAB Code” on page 4-15
- “Kernel Analysis” on page 4-18
- “Analyze Execution Profiles of the Generated Code” on page 4-24

## 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 [https://developer.nvidia.com/nvidia-development-tools-solutions-ERR\\_NVGPUCTRPERM-permission-issue-performance-counters](https://developer.nvidia.com/nvidia-development-tools-solutions-ERR_NVGPUCTRPERM-permission-issue-performance-counters).

---

### 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	<code>report(etStruct, 'Units', 'seconds', 'ScaleFactor', '0.001', 'NumericFormat', '%5.5f');</code>
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	
<a href="#">gpu_fftn_initialize</a>	0.01087	0.01087	0.01087	0.01087	1	
<a href="#">gpu_fftn</a>	4064.92221	689.01660	4064.92221	689.01660	6	
<a href="#">gpu_fftn_terminate</a>	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

`codegen` | `coder.EmbeddedCodeConfig` | `gpucoder.profile`

## More About

- “Code Generation Using the Command Line Interface”
- “Code Generation by Using the GPU Coder App”
- “Code Generation Reports” on page 4-5
- “Trace Between Generated CUDA Code and MATLAB Source Code” on page 4-11

- “Generating a GPU Code Metrics Report for Code Generated from MATLAB Code” on page 4-15

# Analysis with NVIDIA Profiler

<b>In this section...</b>
“Not Enough Parallelism” on page 4-27
“Too Many Local per-Thread Registers” on page 4-27

## 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 4-5
- “Trace Between Generated CUDA Code and MATLAB Source Code” on page 4-11
- “Generating a GPU Code Metrics Report for Code Generated from MATLAB Code” on page 4-15
- “Kernel Analysis” on page 4-18
- “Memory Bottleneck Analysis” on page 4-22
- “Analyze Execution Profiles of the Generated Code” on page 4-24

## 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.

### 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 4-5
- “Trace Between Generated CUDA Code and MATLAB Source Code” on page 4-11
- “Generating a GPU Code Metrics Report for Code Generated from MATLAB Code” on page 4-15
- “Kernel Analysis” on page 4-18
- “Memory Bottleneck Analysis” on page 4-22
- “Analyze Execution Profiles of the Generated Code” on page 4-24

## 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 `gpcoder.profile` function. Fog rectification is used as an example to demonstrate this concept.

### Prerequisites

- 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”.
- The profiling workflow of this example depends on the `nvprof` tool from NVIDIA. 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 [https://developer.nvidia.com/nvidia-development-tools-solutions-ERR\\_NVGPUCTRPERM-permission-issue-performance-counters](https://developer.nvidia.com/nvidia-development-tools-solutions-ERR_NVGPUCTRPERM-permission-issue-performance-counters).

### 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);
```

### Prepare for Code Generation and Profiling

The `fog_rectification.m` function takes a foggy image as input and returns a defogged image. To generate CUDA code, create a GPU code configuration object with a dynamic library ('dll') build type. Because the `gpcoder.profile` function accepts only an Embedded Coder configuration object, a `coder.EmbeddedCodeConfig` configuration object is used even if the `ecoder` option is not explicitly selected.

```
inputImage = imread('foggyInput.png');
inputs = {inputImage};
designFileName = 'fog_rectification';

cfg = coder.gpuConfig('dll');
cfg.GpuConfig.MallocMode = 'discrete';
```

### Generate Execution Profiling Report

Run `gpcoder.profile` with a threshold value of 0.003 to see the SIL execution report. The threshold value of 0.003 is just a representative number. 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. It is advisable to set a low threshold value (between 0.001-0.005) 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.

```
gpcoder.profile(designFileName, inputs, 'CodegenConfig', cfg, 'Threshold', 0.003, 'NumCalls', 10);
```

```

### Starting SIL execution for 'fog_rectification'
To terminate execution: <a href="matlab: targets_hyperlink_manager('run',1);">clear fog_rect.
Execution profiling data is available for viewing. Open <a href="matlab:Simulink.sdi.view;">
Execution profiling report available after termination.
### 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 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. 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 R2020a on a machine with an 6 core, 3.5GHz Intel® Xeon® CPU, and an NVIDIA TITAN XP GPU

### 1. Summary

Total time	999.91132
Unit of time	ms
Command	report(etStruct, 'Units', 'seconds', 'ScaleFactor', '0.001', 'NumericFormat', '%5.5f');
Timer frequency (ticks per second)	2.53122e+09
Profiling data created	22-Oct-2019 16:49:01

### 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	
<a href="#">fog_rectification_initialize</a>	0.00628	0.00628	0.00628	0.00628	1	 
<a href="#">fog_rectification</a>	952.21056	99.99007	952.21056	99.99007	10	  
<a href="#">fog_rectification_terminate</a>	0.00436	0.00436	0.00436	0.00436	1	 

### 3. GPU Profiling Trace for fog\_rectification

Section 3 shows the complete trace of GPU calls that have a runtime higher than the threshold value. The 'Threshold' parameter is defined as the fraction of the maximum execution time for a run (excluding the first run). For example, out of 9 calls to the top level fog\_rectification function, if the third call took the maximum time ( $t$ , ms), then the maximum execution time is  $t$  milliseconds. All GPU calls taking more than  $Threshold * t$  milliseconds is shown in this section. Placing your cursor over the calls shows the run-time values of other relevant non-timing related information for each call. For example, placing your cursor over fog\_rectification\_kernel10 shows the block dimensions, grid dimensions, and the static shared memory size in KiB of that call. This trace corresponds to the run that took the maximum time.

Name	Duration in ms
cudaMalloc	0.1883
cudaMalloc	0.1444
cudaMalloc	0.1389
cudaMalloc	0.1322
cudaMalloc	0.1329
cudaMalloc	0.1323
cudaMalloc	0.1370
cudaMemcpy	0.1209
CUDA memcpy HtoD	0.0766
cudaLaunchKernel	0.0235
fog_rectification_kernel1	0.0493
cudaMemcpy	0.0996
fog_rectification_kernel2	0.0244
fog_rectification_kernel6	0.0348
cudaMemcpy	0.0304
fog_rectification_kernel6	0.0354
cudaMemcpy	0.0302
fog_rectification_kernel6	0.0357
fog_rectification_kernel7	0.0248
cudaMemcpy	0.1215
fog_rectification_kernel8	0.0343
fog_rectification_kernel9	0.0232
fog_rectification_kernel10	0.0524
fog_rectification_kernel10	Name: fog_rectification_kernel10
void imhist_sm	GridDims: [1800 1 1]
cudaMemcpy	BlockDims: [512 1 1]
fog_rectification_kernel10	StaticSharedMem: 0 KiB
cudaMemcpy	0.2780
fog_rectification_kernel2...	0.0289
fog_rectification_kernel2...	0.0265
CUDA memcpy DtoH	0.0710



#### 4. GPU Profiling Summary for fog\_rectification

Section 4 in the report shows the summary of GPU calls that are shown in section 3. The `cudaFree` is called 17 times per run of `fog_rectification` and the average time taken by 17 calls of `cudaFree` over 9 runs of `fog_rectification` is 1.7154 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>	1.7154	17
<code>cudaMalloc</code>	1.0455	17
<code>cudaMemcpy</code>	0.7291	8
<code>cudaLaunchKernel</code>	0.3303	32
<code>fog_rectification_kernel6</code>	0.1059	3
CUDA memcpy HtoD	0.0799	6
CUDA memcpy DtoH	0.0722	2
<code>fog_rectification_kernel1...</code>	0.0526	1
<code>fog_rectification_kernel1</code>	0.0492	1
<code>void imhist_sm</code>	0.0399	1
<code>fog_rectification_kernel8</code>	0.0345	1
<code>fog_rectification_kernel2...</code>	0.0289	1
<code>fog_rectification_kernel2...</code>	0.0285	1
<code>fog_rectification_kernel1...</code>	0.0271	1
<code>fog_rectification_kernel2...</code>	0.0268	1
<code>fog_rectification_kernel7</code>	0.0251	1
<code>fog_rectification_kernel2</code>	0.0241	1
<code>fog_rectification_kernel9</code>	0.0235	1



# Deep Learning

---

- “Workflow” on page 5-2
- “Supported Networks, Layers, and Classes” on page 5-5
- “Code Generation for dlarray” on page 5-35
- “dlarray Limitations for Code Generation” on page 5-41
- “Generated CNN Class Hierarchy” on page 5-44
- “Load Pretrained Networks for Code Generation” on page 5-45
- “Code Generation for Deep Learning Networks by Using cuDNN” on page 5-48
- “Code Generation for Deep Learning Networks by Using TensorRT” on page 5-57
- “Code Generation for Deep Learning Networks Targeting ARM Mali GPUs” on page 5-66
- “Data Layout Considerations in Deep Learning” on page 5-70
- “Quantization of Deep Neural Networks” on page 5-73
- “Code Generation for Quantized Deep Learning Networks” on page 5-81
- “Code Generation for Deep Learning Networks” on page 5-91
- “Lane Detection Optimized with GPU Coder” on page 5-100
- “Traffic Sign Detection and Recognition” on page 5-111
- “Logo Recognition Network” on page 5-120
- “Pedestrian Detection” on page 5-125
- “Deep Learning Prediction by Using NVIDIA TensorRT” on page 5-132
- “Code Generation for Semantic Segmentation Network” on page 5-137
- “Train and Deploy Fully Convolutional Networks for Semantic Segmentation” on page 5-142
- “Code Generation for Semantic Segmentation Network That Uses U-net” on page 5-154
- “Code Generation for Denoising Deep Neural Network” on page 5-161
- “Code Generation for Object Detection by Using YOLO v2” on page 5-165
- “Code Generation for a Sequence-to-Sequence LSTM Network” on page 5-169
- “Deep Learning Prediction on ARM Mali GPU” on page 5-175
- “Code Generation for Object Detection by Using Single Shot Multibox Detector” on page 5-178
- “Code Generation for a Deep Learning Simulink Model to Classify ECG Signals” on page 5-182
- “Code Generation for Lidar Point Cloud Segmentation Network” on page 5-189
- “Code Generation for a Video Classification Network” on page 5-196
- “Code Generation For Object Detection Using YOLO v3 Deep Learning” on page 5-202
- “Generate Digit Images on NVIDIA GPU Using Variational Autoencoder ” on page 5-211

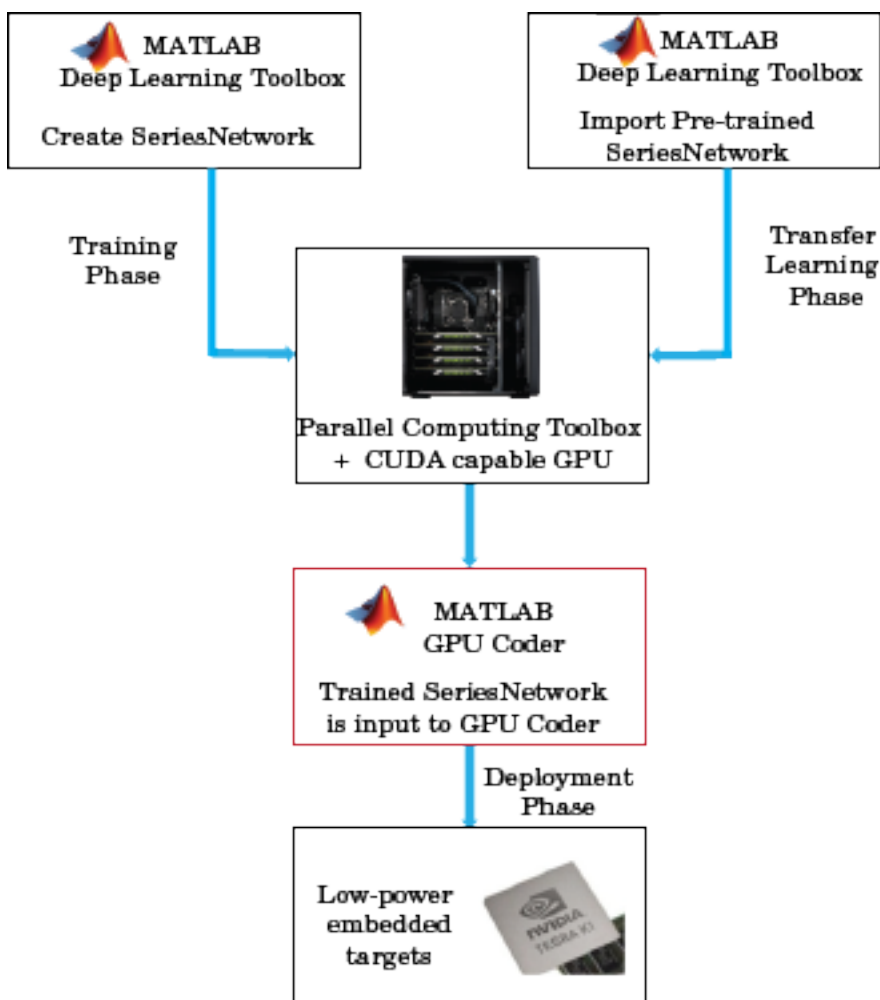
## Workflow

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

`codegen` | `coder.DeepLearningConfig` | `coder.getDeepLearningLayers`

### Objects

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

## 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 5-5
- “Load Pretrained Networks for Code Generation” on page 5-45
- “Code Generation for Deep Learning Networks by Using cuDNN” on page 5-48

- “Code Generation for Deep Learning Networks by Using TensorRT” on page 5-57
- “Code Generation for Deep Learning Networks Targeting ARM Mali GPUs” on page 5-66

## 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 5-10. 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
 imageInputLayer	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 <b>NumGroups</b> 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
 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
 crossChannelNormalizationLayer	A channel-wise local response (cross-channel) normalization layer carries out channel-wise normalization.	Yes	Yes	Yes



Layer Name	Description	cuDNN	TensorRT	ARM Compute Library for Mali GPU
 dropoutLayer	A dropout layer randomly sets input elements to zero with a given probability.	Yes	Yes	Yes
 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
 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





Layer Name	Description	cuDNN	TensorRT	ARM Compute Library for Mali GPU
 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
 maxPooling2dLayer	A max pooling layer performs down-sampling by dividing the input into rectangular pooling regions, and computing the maximum of each region.	Yes	Yes	Yes
 maxUnpooling2dLayer	A max unpooling layer unpools the output of a max pooling layer.	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
 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
 ssdMergeLayer	An SSD merge layer merges the outputs of feature maps for subsequent regression and classification loss computation.	Yes	Yes	No






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
 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

**Keras and ONNX Layers**

Layer Name	Description	cuDNN	TensorRT	ARM Compute Library for Mali GPU
<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.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.ZeroPadding2dLayer</code> (Deep Learning Toolbox)	Zero padding layer for 2-D input.	Yes	Yes	Yes
<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.FlattenLayer</code> (Deep Learning Toolbox)	Flattens the spatial dimensions of the input tensor to the channel dimensions.	Yes	Yes	Yes
<code>nnet.onnx.layer.IdentityLayer</code> (Deep Learning Toolbox)	Layer that implements ONNX identity operator.	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 5-202.</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 2-24</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>Code generation for a deep learning network with custom layer is not supported in Simulink.</p>			

## Supported Classes

The following classes are supported for code generation by GPU Coder for the target deep learning libraries specified in the table.

<b>Name</b>	<b>Description</b>	<b>cuDNN</b>	<b>TensorRT</b>	<b>ARM Compute Library for Mali GPU</b>
DAGNetwork	Directed acyclic graph (DAG) network for deep learning <ul style="list-style-type: none"><li>• Only the activations, predict, and classify methods are supported.</li></ul>	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"><li>• Code generation supports only the <code>InputNames</code> and <code>OutputNames</code> properties.</li><li>• Code generation does not support <code>dlnetwork</code> objects without input layers.</li><li>• Code generation for <code>dlnetwork</code> objects with <code>sequenceInputLayer</code> objects is not supported.</li><li>• Code generation supports only the <code>predict</code> object function. The <code>dlnetwork</code> input to the <code>predict</code> method must be a single datatype.</li></ul>	Yes	Yes	No

Name	Description	cuDNN	TensorRT	ARM Compute Library for Mali GPU
SeriesNetwork	Series network for deep learning <ul style="list-style-type: none"><li>• Only the activations, classify, predict, predictAndUpdateState, classifyAndUpdateState, and resetState object functions are supported.</li></ul>	Yes	Yes	Yes



Name	Description	cuDNN	TensorRT	ARM Compute Library for Mali GPU
ssdObjectDetector	<p>Object to detect objects using the SSD-based detector.</p> <ul style="list-style-type: none"> <li>• Only the detect method of the ssdObjectDetector is supported for code generation.</li> <li>• The roi argument to the detect method must be a codegen constant (coder.const()) and a 1x4 vector.</li> <li>• Only the Threshold, SelectStrongest, MinSize, MaxSize, and MiniBatchSize Name-Value pairs are supported. All Name-Value pairs must be compile-time constants.</li> <li>• The channel and batch size of the input image must be fixed size.</li> <li>• The labels output is returned as a categorical array.</li> <li>• In the generated code,</li> </ul>	Yes	Yes	No

Name	Description	cuDNN	TensorRT	ARM Compute Library for Mali GPU
	<p>the input is rescaled to the size of the input layer of the network. But the bounding box that the <code>detect</code> method returns is in reference to the original input size.</p> <ul style="list-style-type: none"><li>• The bounding boxes might not numerically match the simulation results.</li></ul>			

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"> <li>• Only the detect method of the yoloV2ObjectDetector is supported for code generation.</li> <li>• The roi argument to the detect method must be a codegen constant (coder.constant()) and a 1x4 vector.</li> <li>• Only the Threshold, SelectStrongest, MinSize, MaxSize, and MiniBatchSize Name-Value pairs are supported.</li> <li>• The height, width, channel, and batch size of the input image must be fixed size.</li> <li>• The minimum batch size value passed to detect method must be fixed size.</li> <li>• The labels output is returned as a cell array of character</li> </ul>	Yes	Yes	Yes

Name	Description	cuDNN	TensorRT	ARM Compute Library for Mali GPU
	vectors, such as {'car','bus'}.			

## See Also

### Functions

`codegen` | `coder.DeepLearningConfig` | `coder.getDeepLearningLayers`

### Objects

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

## 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 5-45
- “Code Generation for Deep Learning Networks by Using cuDNN” on page 5-48
- “Code Generation for Deep Learning Networks by Using TensorRT” on page 5-57
- “Code Generation for Deep Learning Networks Targeting ARM Mali GPUs” on page 5-66

## Code Generation for dlarray

### In this section...

“Define dlarray for Code Generation” on page 5-35

“dlarray Object Functions with Code Generation Support” on page 5-36

“Deep Learning Toolbox Functions with dlarray Code Generation Support” on page 5-37

“MATLAB Functions with dlarray Code Generation Support” on page 5-37

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 5-211.

## 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.
cos	The output dlarray has the same data format as the input dlarray.
cosh	
cot	
csc	
exp	
log	<ul style="list-style-type: none"> <li>The output dlarray has the same data format as the input dlarray.</li> <li>Because dlarray does not support complex numbers, the input dlarray must have nonnegative values.</li> </ul>
sec	The output dlarray has the same data format as the input dlarray.
sign	
sin	
sinh	
sqrt	<ul style="list-style-type: none"> <li>The output dlarray has the same data format as the input dlarray.</li> <li>Because dlarray does not support complex numbers, the input dlarray must have nonnegative values.</li> </ul>
tan	The output dlarray has the same data format as the input dlarray.
tanh	
uplus, +	

Function	Notes and Limitations
uminus, -	

### Extrema Functions

Function	Notes and Limitations
ceil	The output <code>darray</code> has the same data format as the input <code>darray</code> .
eps	<ul style="list-style-type: none"> <li>The output <code>darray</code> has the same data format as the input <code>darray</code>.</li> <li>Use <code>eps(ones('like', x))</code> to get a scalar epsilon value based on the data type of a <code>darray</code> <code>x</code>.</li> </ul>
fix	The output <code>darray</code> has the same data format as the input <code>darray</code> .
floor	The output <code>darray</code> has the same data format as the input <code>darray</code> .
round	<ul style="list-style-type: none"> <li>Only the syntax <code>Y = round(X)</code> is supported.</li> <li>The output <code>darray</code> has the same data format as the input <code>darray</code>.</li> </ul>

### Conversion Functions

Function	Notes and Limitations
double	The output is a <code>darray</code> that contains data of type <code>double</code> .
logical	The output is a <code>darray</code> that contains data of type <code>logical</code> .
single	The output is a <code>darray</code> that contains data of type <code>single</code> .

### Comparison Functions

Function	Notes and Limitations
isequal	<ul style="list-style-type: none"> <li>The syntax with more than two input arguments is not supported.</li> <li>Two <code>darray</code> 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.</li> </ul>
isequaln	<ul style="list-style-type: none"> <li>The syntax with more than two input arguments is not supported.</li> <li>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.</li> </ul>



## Data Type and Value Identification Functions

Function	Notes and Limitations
isfloat	The software applies the function to the underlying data of an input dlarray.
islogical	
isnumeric	
isreal	Because dlarray does not support complex numbers, this function always returns true for a dlarray input.

## Size Identification Functions

Function	Notes and Limitations
length	N/A
ndims	If the input dlarray dlX is formatted, then ndims(dlX) returns the number of dimension labels, even if some of the labeled dimensions are trailing singleton dimensions.
numel	N/A
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	

## See Also

### Objects

dlarray | dlnetwork

## Related Examples

- “Generate Digit Images on NVIDIA GPU Using Variational Autoencoder” on page 5-211

## **More About**

- “dlarray Limitations for Code Generation” on page 5-41
- “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 5-41

“Limitations” on page 5-41

### 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`.

The screenshot shows MATLAB code on the left and a tooltip on the right. The code includes a function definition, a comment, and a line where a `dlarray` object is created from random noise. The tooltip, titled 'EXPRESSION INFO', shows the expression `dlarray(randn(1,1,latentDim,25), 'SSCB')` with a size of `1 x 1` and a class of `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).

- Code generation does not support indexing with `dlarray` objects.
- 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 5-211

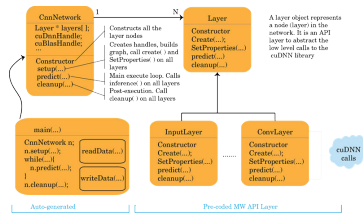
## More About

- “Code Generation for dlarray” on page 5-35
- “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 5-5
- “Load Pretrained Networks for Code Generation” on page 5-45
- “Code Generation for Deep Learning Networks by Using cuDNN” on page 5-48
- “Code Generation for Deep Learning Networks by Using TensorRT” on page 5-57
- “Code Generation for Deep Learning Networks Targeting ARM Mali GPUs” on page 5-66

## 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 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 | coder.loadDeepLearningNetwork | trainNetwork

### Objects

DAGNetwork | SeriesNetwork | dlarray | dlnetwork | ssdObjectDetector |  
yolov2ObjectDetector

## More About

- “Supported Networks, Layers, and Classes” on page 5-5
- “Code Generation for Deep Learning Networks by Using cuDNN” on page 5-48
- “Code Generation for Deep Learning Networks by Using TensorRT” on page 5-57

## 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 R2020b onwards, it is recommended to use the `codegen` command instead of the `cnncodegen` function because in a future release, the `cnncodegen` function will generate C++ code and build a static library 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, 6.3 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”.

- Environment variables for the compilers and libraries. For more information, see “Environment Variables”.

## Load Pretrained Network




- 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.

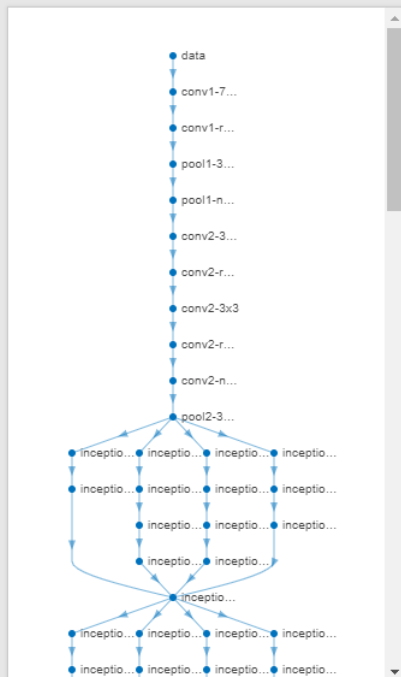
```
net = googlenet;
```

- 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  layers    0  warnings    0  errors



ANALYSIS RESULT				
	Name	Type	Activations	Learnables
1	data 224x224x3 images with 'zerocenter' normalization	Image Input	224x224x3	-
2	conv1-7x7_s2 64 7x7x3 convolutions with stride [2 2] and padding [3 3 3 3]	Convolution	112x112x64	Weights 7x7x3x64 Bias 1x1x64
3	conv1-relu_7x7 ReLU	ReLU	112x112x64	-
4	pool1-3x3_s2 3x3 max pooling with stride [2 2] and padding [0 1 0 1]	Max Pooling	56x56x64	-
5	pool1-norm1 cross channel normalization with 5 channels per element	Cross Channel Nor...	56x56x64	-
6	conv2-3x3_reduce 64 1x1x64 convolutions with stride [1 1] and padding [0 0 0 0]	Convolution	56x56x64	Weights 1x1x64x64 Bias 1x1x64
7	conv2-relu_3x3_reduce ReLU	ReLU	56x56x64	-
8	conv2-3x3 192 3x3x64 convolutions with stride [1 1] and padding [1 1 1 1]	Convolution	56x56x192	Weights 3x3x64x192 Bias 1x1x192
9	conv2-relu_3x3 ReLU	ReLU	56x56x192	-
10	conv2-norm2 cross channel normalization with 5 channels per element	Cross Channel Nor...	56x56x192	-
11	pool2-3x3_s2 3x3 max pooling with stride [2 2] and padding [0 1 0 1]	Max Pooling	28x28x192	-
12	inception_3a-1x1 64 1x1x192 convolutions with stride [1 1] and padding [0 0 0 0]	Convolution	28x28x64	Weights 1x1x192x64 Bias 1x1x64
13	inception_3a-relu_1x1 ReLU	ReLU	28x28x64	-
14	inception_3a-3x3_reduce 96 1x1x192 convolutions with stride [1 1] and padding [0 0 0 0]	Convolution	28x28x96	Weights 1x1x192x96 Bias 1x1x96
15	inception_3a-relu_3x3_reduce ReLU	ReLU	28x28x96	-

- 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 5-45.
  - 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.

---

**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;
}

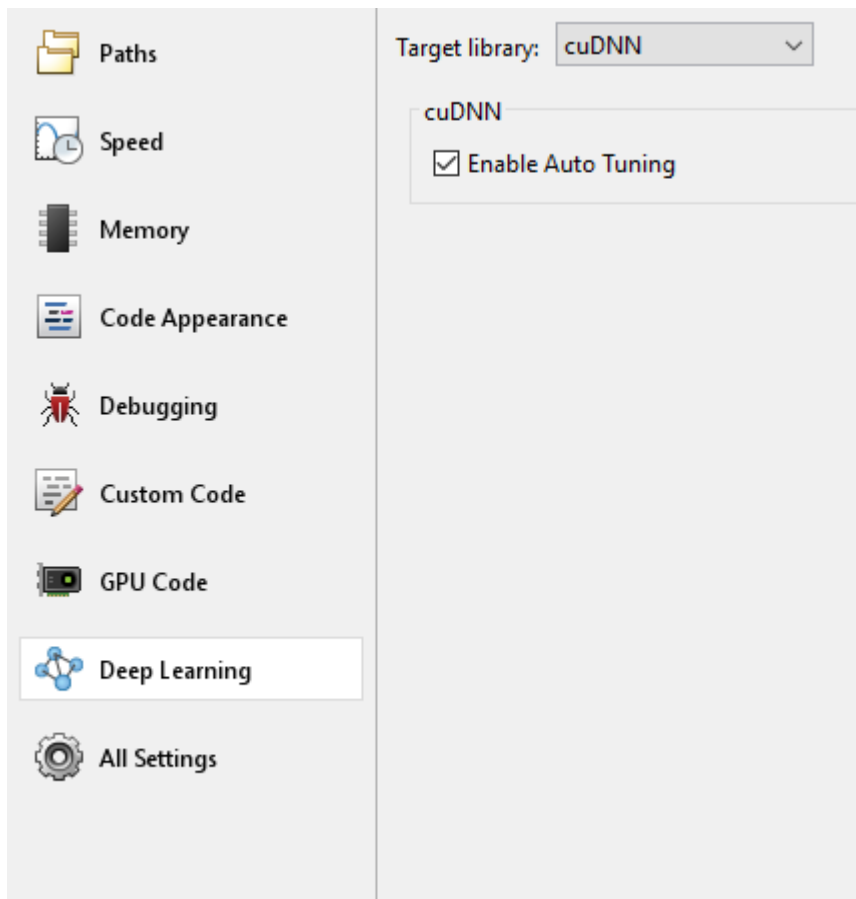
```

## 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

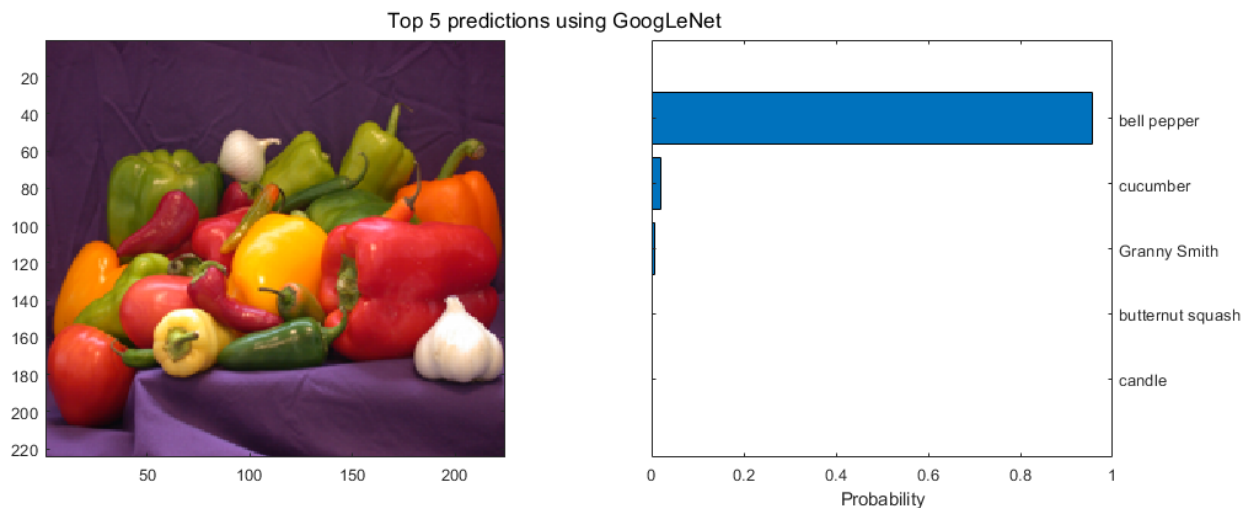


is common to consider the top-five accuracy when evaluating networks. The network classifies the image as a bell pepper with a high probability.

```
[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

`codegen` | `coder.DeepLearningConfig` | `coder.loadDeepLearningNetwork`

### Objects

`coder.CodeConfig` | `coder.CuDNNConfig` | `coder.EmbeddedCodeConfig` | `coder.gpuConfig`

## See Also

### More About

- “Supported Networks, Layers, and Classes” on page 5-5
- “Load Pretrained Networks for Code Generation” on page 5-45
- “Code Generation for Deep Learning Networks by Using TensorRT” on page 5-57
- “Pedestrian Detection” on page 5-125

- “Deployment and Classification of Webcam Images on NVIDIA Jetson TX2 Platform” on page 6-44
- “Generated CNN Class Hierarchy” on page 5-44

## 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 R2020b onwards, it is recommended to use the `codegen` command instead of the `cnncodegen` function because in a future release, the `cnncodegen` function will generate C++ code and build a static library 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 by Using NVIDIA TensorRT” on page 5-132 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, 6.3 or higher. Half-precision requires a CUDA GPU with minimum compute capability of 7.0.

### **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.

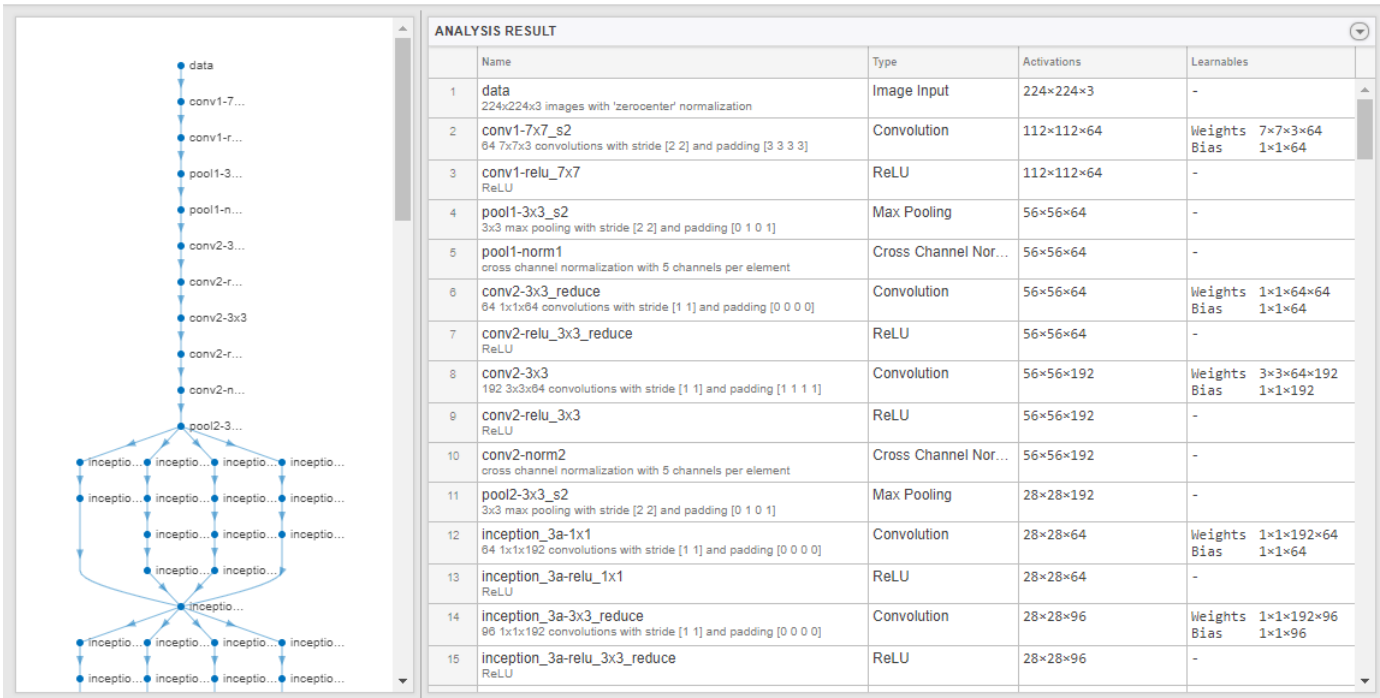
```
net = googlenet;
```

- 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 5-45.

**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 minimum compute capability of 6.1. FP16 precision requires a CUDA GPU with minimum compute capability of 7.0. 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 recalibration 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 recalibration and the preprocessing steps must be included in the entry-point file before code generation.

---

**Note** Code generation for INT8 data type does not support multiple deep learning networks in the entry-point function.

---

See the “Deep Learning Prediction by Using NVIDIA TensorRT” on page 5-132 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.

---

**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;
}

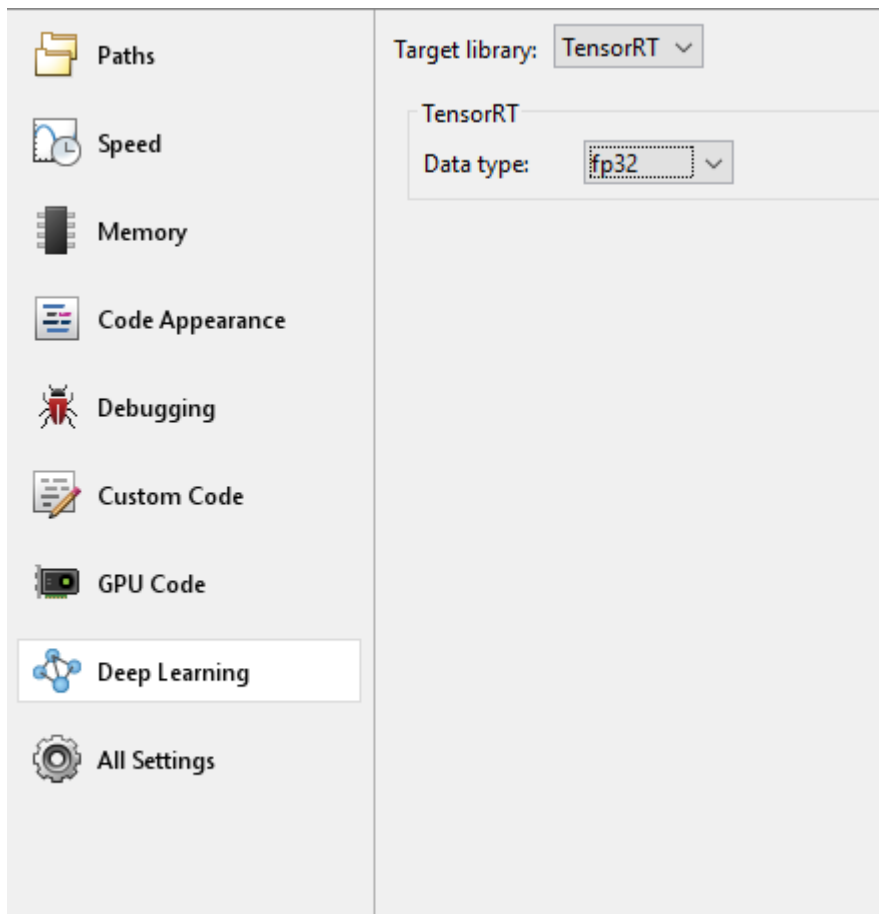
```

## 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.

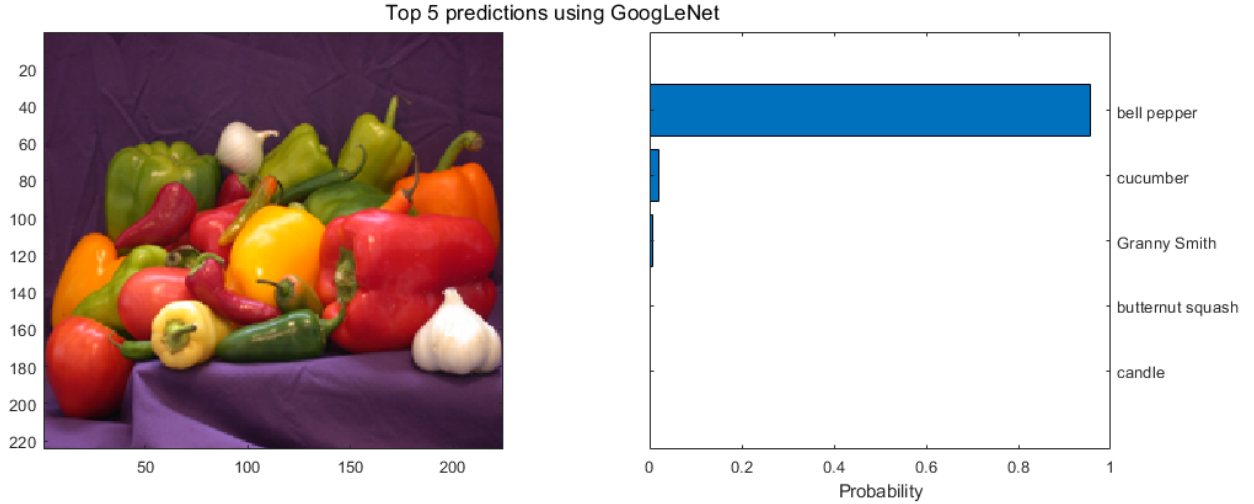
```
predict_scores = googlenet_predict_mex(im);
```

- 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

`codegen` | `coder.DeepLearningConfig` | `coder.loadDeepLearningNetwork`

### Objects

`coder.CodeConfig` | `coder.EmbeddedCodeConfig` | `coder.TensorRTConfig` | `coder.gpuConfig`

## See Also

### More About

- “Supported Networks, Layers, and Classes” on page 5-5
- “Load Pretrained Networks for Code Generation” on page 5-45
- “Code Generation for Deep Learning Networks by Using cuDNN” on page 5-48
- “Deep Learning Prediction by Using NVIDIA TensorRT” on page 5-132
- “Code Generation for Deep Learning Networks” on page 5-91
- “Code Generation for Object Detection by Using YOLO v2” on page 5-165
- “Deployment and Classification of Webcam Images on NVIDIA Jetson TX2 Platform” on page 6-44

## 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.

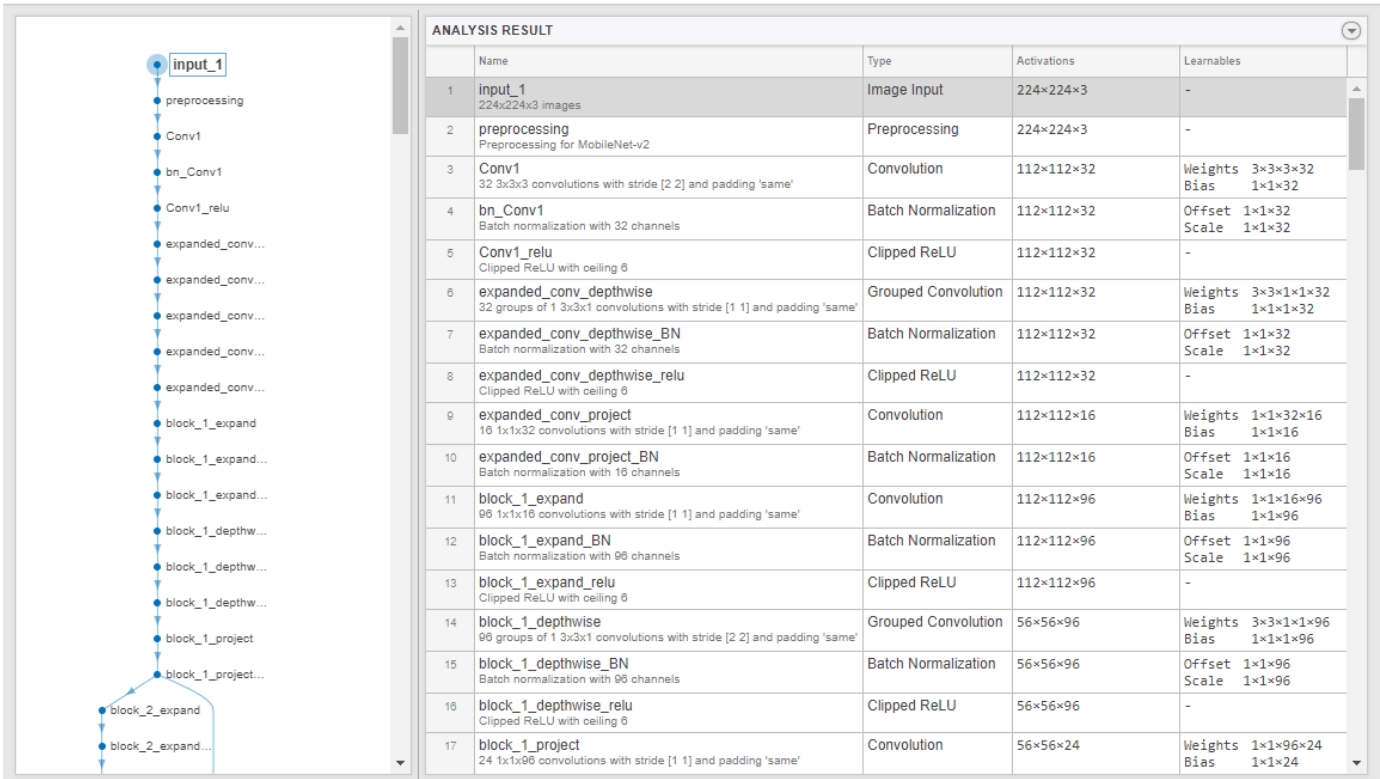
```
net = mobilenetv2;
```

- 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 5-175
- 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

`cnncodegen` | `coder.DeepLearningConfig` | `coder.getDeepLearningLayers`

## More About

- “Supported Networks, Layers, and Classes” on page 5-5
- “Load Pretrained Networks for Code Generation” on page 5-45
- “Code Generation for Deep Learning Networks by Using cuDNN” on page 5-48
- “Code Generation for Deep Learning Networks by Using TensorRT” on page 5-57
- “Deep Learning Prediction on ARM Mali GPU” on page 5-175

## 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.

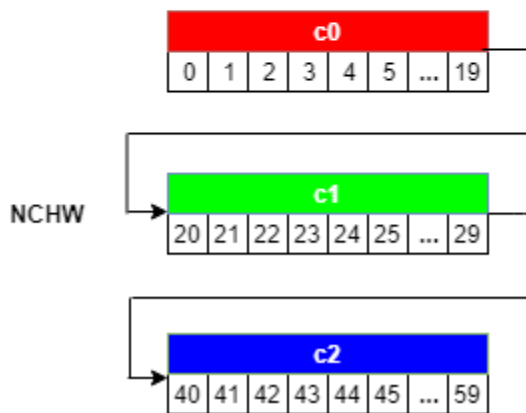
For example, consider a batch of images with the following dimensions: N=1, C=3, H=5, W=4. If the image pixel elements are represented by a sequence of integers, the input images can be pictorially represented as follows.



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



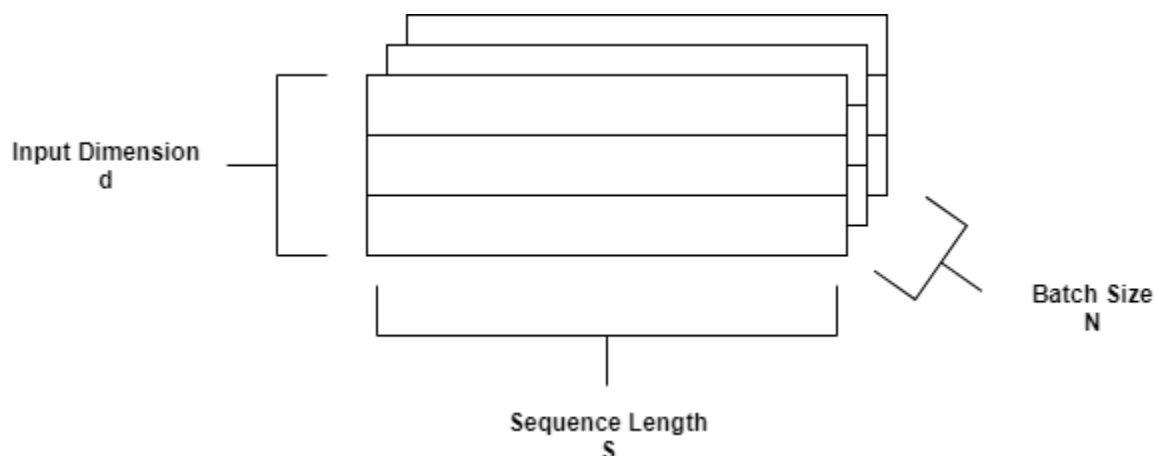
When creating the input buffer in the main function, the 4-D image is laid out in the memory in the NCHW format as:

- 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

`codegen` | `coder.DeepLearningConfig` | `coder.getDeepLearningLayers`

### Objects

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

## More About

- “Supported Networks, Layers, and Classes” on page 5-5
- “Load Pretrained Networks for Code Generation” on page 5-45
- “Code Generation for Deep Learning Networks by Using cuDNN” on page 5-48
- “Code Generation for Deep Learning Networks by Using TensorRT” on page 5-57
- “Code Generation for Deep Learning Networks Targeting ARM Mali GPUs” on page 5-66
- “Lane Detection Optimized with GPU Coder” on page 5-100
- “Deployment and Classification of Webcam Images on NVIDIA Jetson TX2 Platform” on page 6-44

# 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.

Quantization of a neural network requires a GPU, the GPU Coder Interface for Deep Learning Libraries support package, and the Deep Learning Toolbox Model Quantization Library support package. Using a GPU requires a CUDA enabled NVIDIA GPU with compute capability 6.1, 6.3 or higher.

## 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 <sup>6</sup>	2 <sup>5</sup>	2 <sup>4</sup>	2 <sup>3</sup>	2 <sup>2</sup>	2 <sup>1</sup>	2 <sup>0</sup>	2 <sup>-1</sup>	2 <sup>-2</sup>	2 <sup>-3</sup>	2 <sup>-4</sup>	2 <sup>-5</sup>	2 <sup>-6</sup>	2 <sup>-7</sup>			2 <sup>-8</sup>		
0.03125															1	0	0	0		
-0.250	✓									1	0	0	0	0	0	0	0	0		
0.250										1	0	0	0	0	0	0	0	0		
0.500									1	0	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		

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						

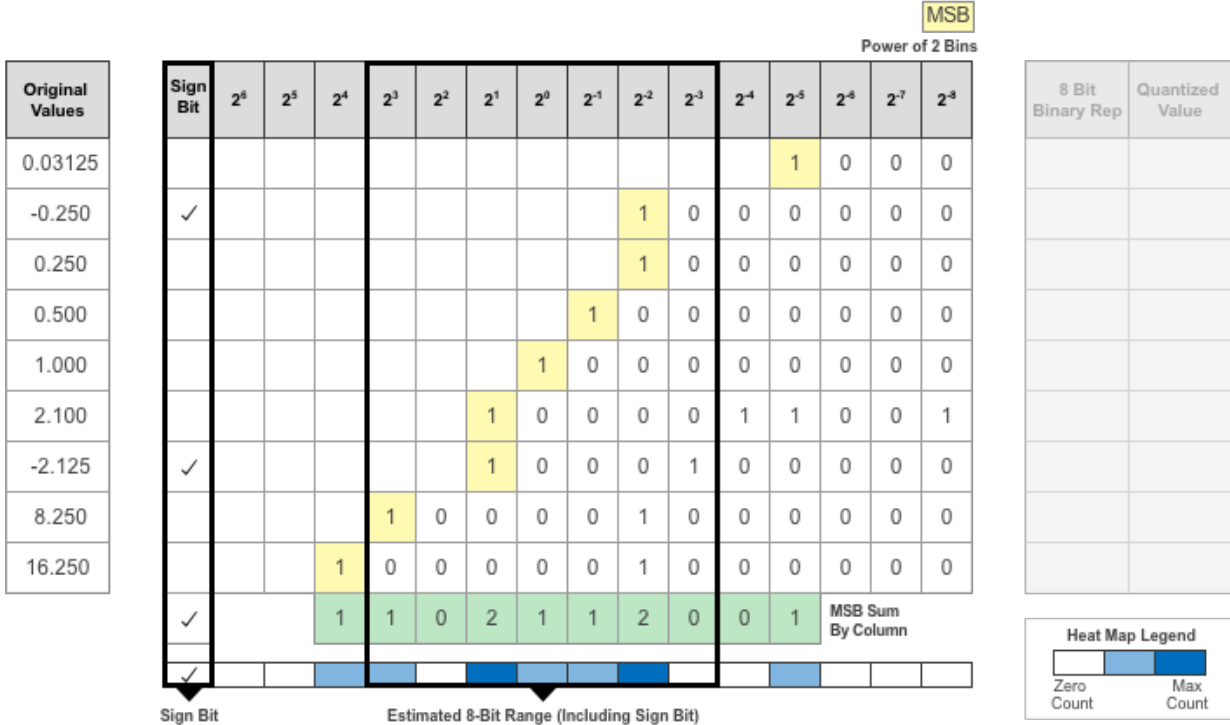
4 Display the MSB counts of each bit location as a heat map. In this heat map, darker blue regions correspond to a larger number of MSB's in the bit location.

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		
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						
✓																			

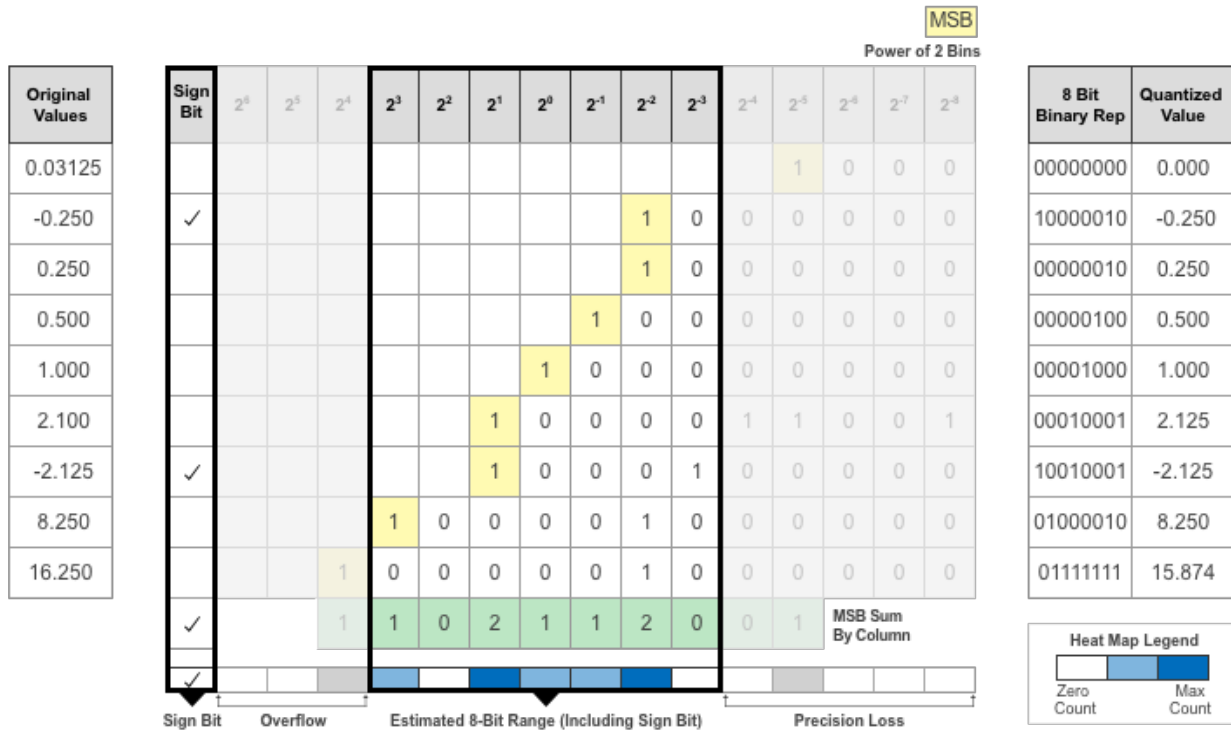
**Heat Map Legend**

Zero Count		Max Count

- 5 The software assigns a data type that can represent the bit locations that capture the most information. In this example, the software selects a data type that represents bits from  $2^3$  to  $2^{-3}$ . An additional sign bit is required to represent the signedness of the value.

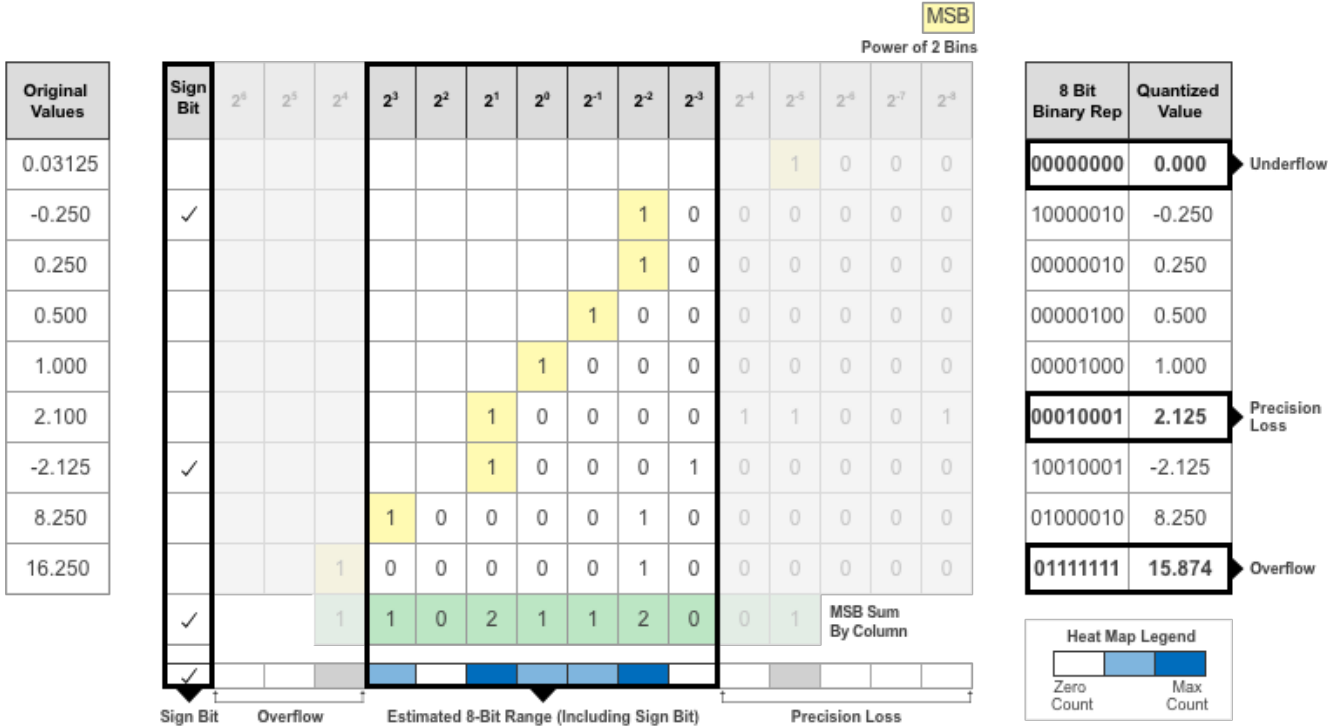


- 6 After assigning the data type, any bits outside of that data type are removed. Due to the assignment of a smaller data type of fixed length, precision loss, overflow, and underflow can occur for values that are not representable by the data type.

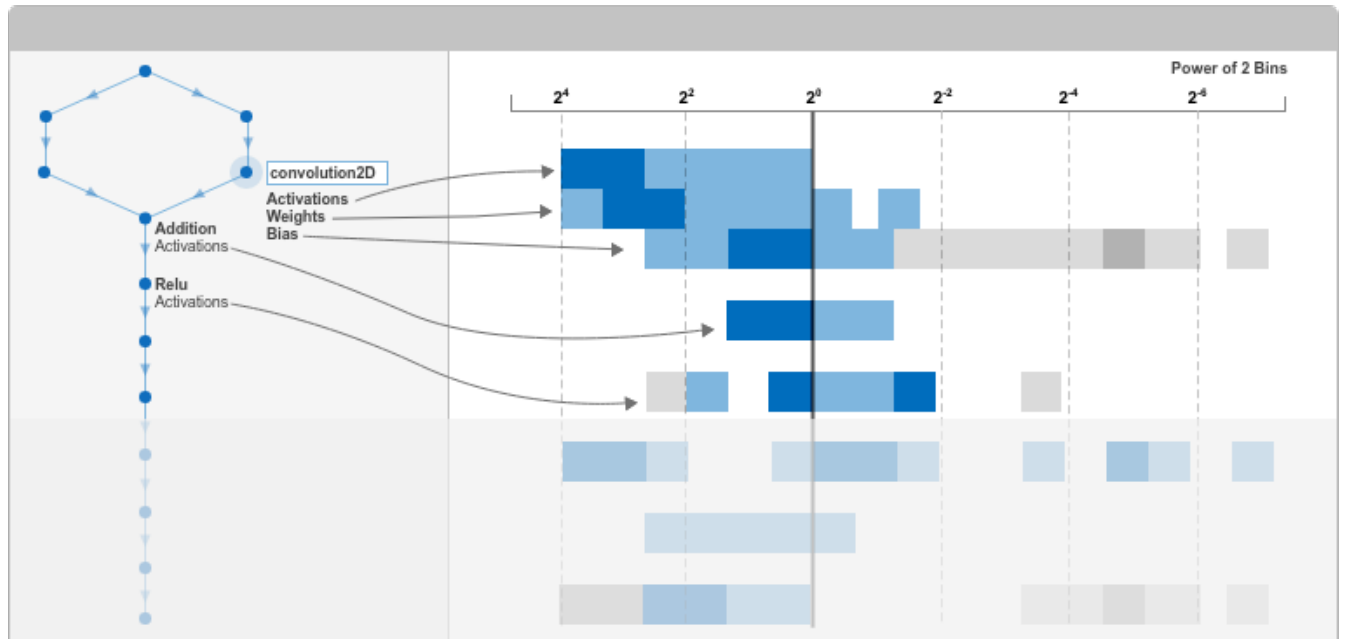


In this example, the value 0.03125, suffers from an underflow, so the quantized value is 0. The value 2.1 suffers some precision loss, so the quantized value is 2.125. The value 16.250 is larger than the largest representable value of the data type, so this value overflows and the quantized value saturates 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 | dlquantizationOptions | dlquantizer | validate

# Code Generation for Quantized 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 optimized CUDA code for the quantized 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 on a GPU Using a Quantized Network

In this example, you use GPU Coder to generate CUDA code for a quantized deep convolutional neural network and classify an image. The example uses the pretrained `squeezenet` convolutional neural network to demonstrate transfer learning, quantization, and CUDA code generation for the quantized 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.

### 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 datastore 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);
```

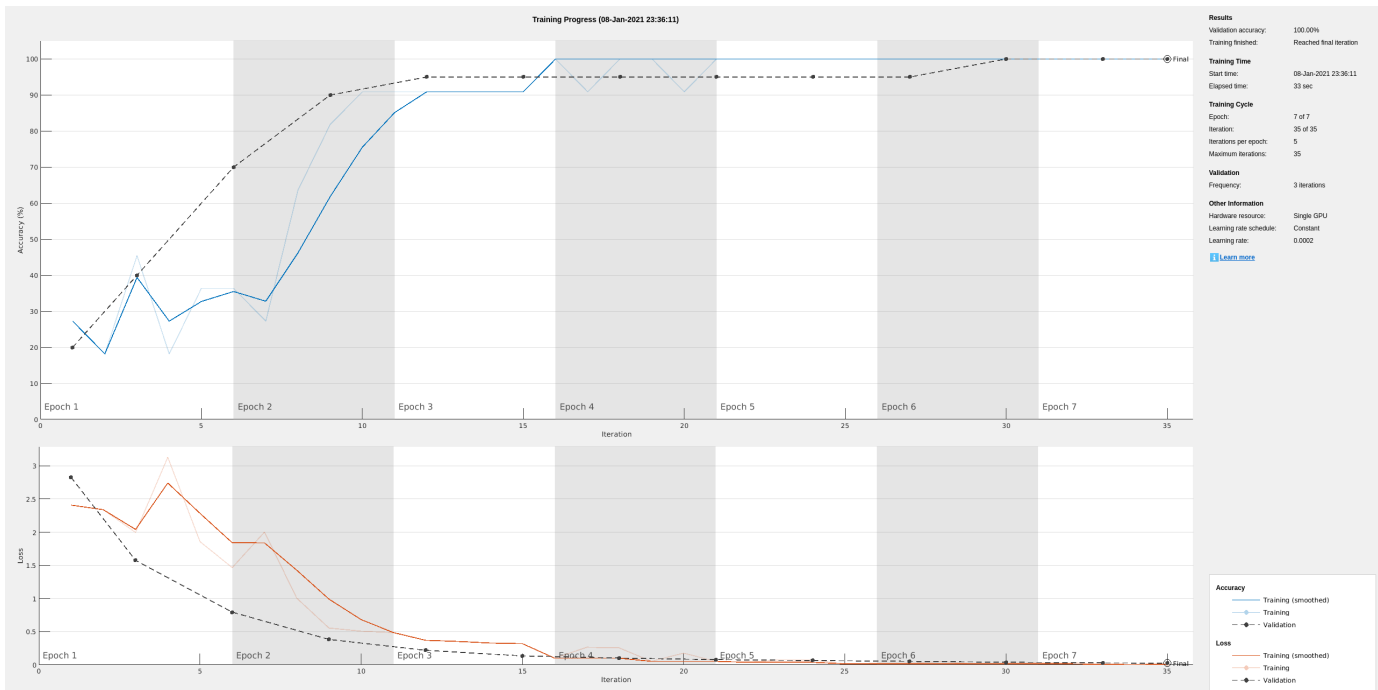
Specify the training options. For transfer learning, keep the features from the early layers of the pretrained network (the transferred layer weights). To slow down learning in the transferred layers, set the initial learning rate to a small value. In the previous step, you increased the learning rate 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');
```

## Quantize the Network

Create a `dlquantizer` object and specify the network to quantize.

```
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](#).
- 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.

```
codegen -config cfg -args {coder.Constant('mySqueezeNet.mat'),ones(inputSize,'uint8')} predict_int8.m
```

Code generation successful.

```
Warning: Name is nonexistent or not a directory: /tmp/tp99d19455_dd79_4e99_acb1_e3850de7f10e
```

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.

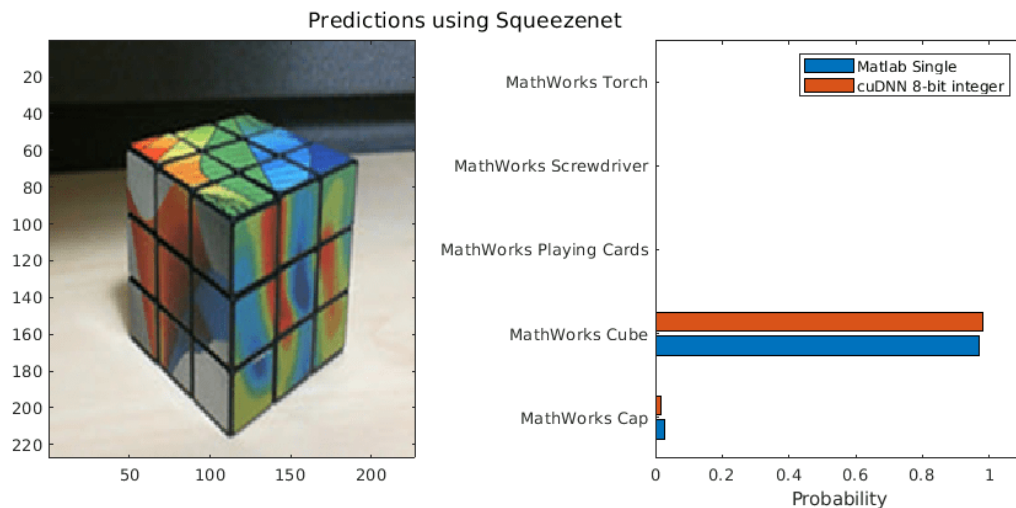
```
h = figure;
h.Position(3) = 2*h.Position(3);
```

```

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
        error('Layer graph must have a single learnable layer preceding the classification layer')
    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

calibrate | codegen | coder.loadDeepLearningNetwork | dlquantizationOptions | dlquantizer | validate

### Objects

coder.CuDNNConfig | coder.TensorRTConfig

## More About

- “Quantization of Deep Neural Networks” on page 5-73

- “Code Generation for Deep Learning Networks by Using cuDNN” on page 5-48
- “Code Generation for Deep Learning Networks by Using TensorRT” on page 5-57

## 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: To view the report, open('codegen/mex/mobilenetv2\_predict/html/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.

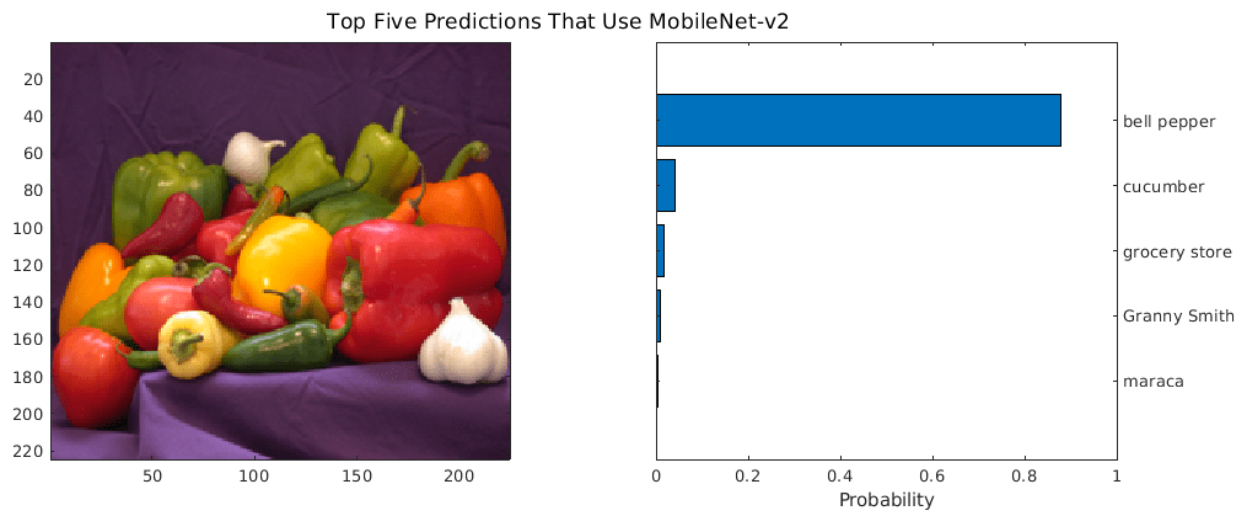
```
im = imresize(im, [224,224]);
predict_scores = mobilenetv2_predict_mex(double(im));
```

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')
```



### Classification of Video

The included helper function `mobilenet_live.m` grabs frames from a webcam, performs prediction, and displays the classification results on each of the captured video frames. This example uses the `webcam` (MATLAB Support Package for USB Webcams) function that is supported by the MATLAB® Support Package for USB Webcams™. You can download and install the support package through the Support Package Installer.

```
type('mobilenet_live.m')
```

```
% Copyright 2017-2019 The MathWorks, Inc.
```

```
function mobilenet_live
```

```
% Connect to a camera
camera = webcam;
```



```

% The labels with top 5 prediction scores are
% mapped to corresponding labels
net = mobilenetv2();
classnames = net.Layers(end).ClassNames;

imfull = zeros(224,400,3, 'uint8');

fps = 0;

ax = axes;

while true
    % Take a picture
    ipicture = camera.snapshot;

    % Resize and cast the picture to single
    picture = imresize(ipicture,[224,224]);

    % Call MEX function for MobileNet-v2 prediction
    tic;
    pout = mobilenetv2_predict(single(picture));
    newt = toc;

    % fps
    fps = .9*fps + .1*(1/newt);

    % top 5 scores
    [top5labels, scores] = getTopFive(pout,classnames);

    % display
    if isvalid(ax)
        dispResults(ax, imfull, picture, top5labels, scores, fps);
    else
        break;
    end
end

end

function dispResults(ax, imfull, picture, top5labels, scores, fps)
for k = 1:3
    imfull(:,177:end,k) = picture(:,:,k);
end

h = imshow(imfull, 'InitialMagnification',200, 'Parent', ax);
scol = 1;
srow = 20;
text(get(h, 'Parent'), scol, srow, sprintf('MobileNet-v2 Demo'), 'color', 'w', 'FontSize', 20);
srow = srow + 20;

text(get(h, 'Parent'), scol, srow, sprintf('Fps = %2.2f', fps), 'color', 'w', 'FontSize', 15);
srow = srow + 20;
for k = 1:5
    t = text(get(h, 'Parent'), scol, srow, top5labels{k}, 'color', 'w','FontSize', 15);
    pos = get(t, 'Extent');
    text(get(h, 'Parent'), pos(1)+pos(3)+5, srow, sprintf('%2.2f%%', scores(k)), 'color', 'w', 'FontSize', 15);
    srow = srow + 20;
end
end

```

```

drawnow;
end

function [labels, scores] = getTopFive(predictOut,classnames)
[val,indx] = sort(predictOut, 'descend');
scores = val(1:5)*100;
labels = classnames(indx(1:5));
end

```

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: To view the report, open('codegen/mex/resnet\_predict/html/report.mld')

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));

```

```

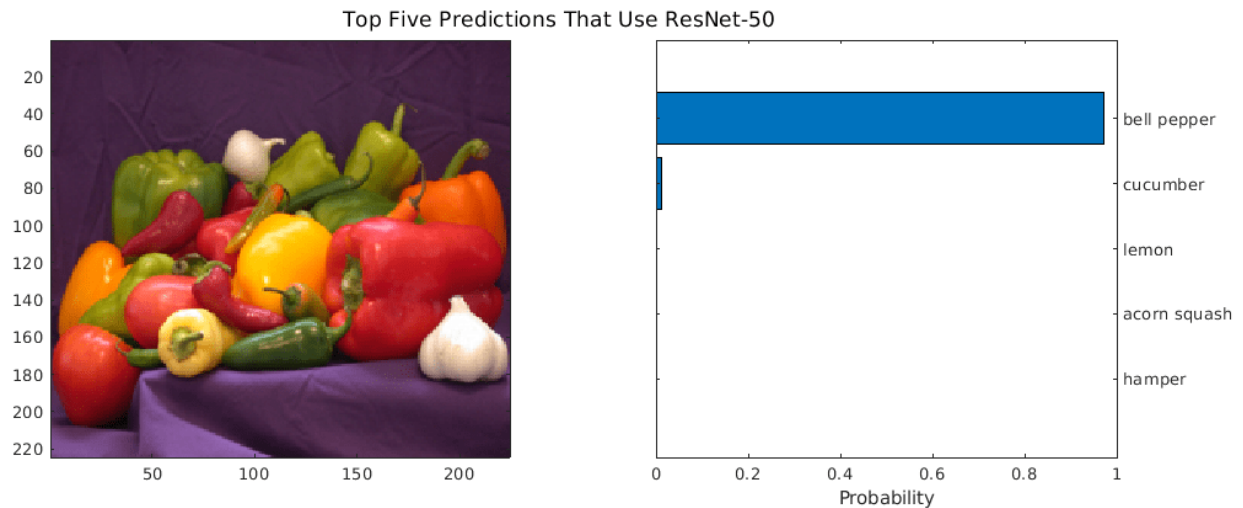
h = figure;
h.Position(3) = 2*h.Position(3);
ax1 = subplot(1,2,1);
ax2 = subplot(1,2,2);

```

```

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: To view the report, open('codegen/mex/googlenet\_predict/html/report.r

Call `googlenet_predict_mex` on the input image.

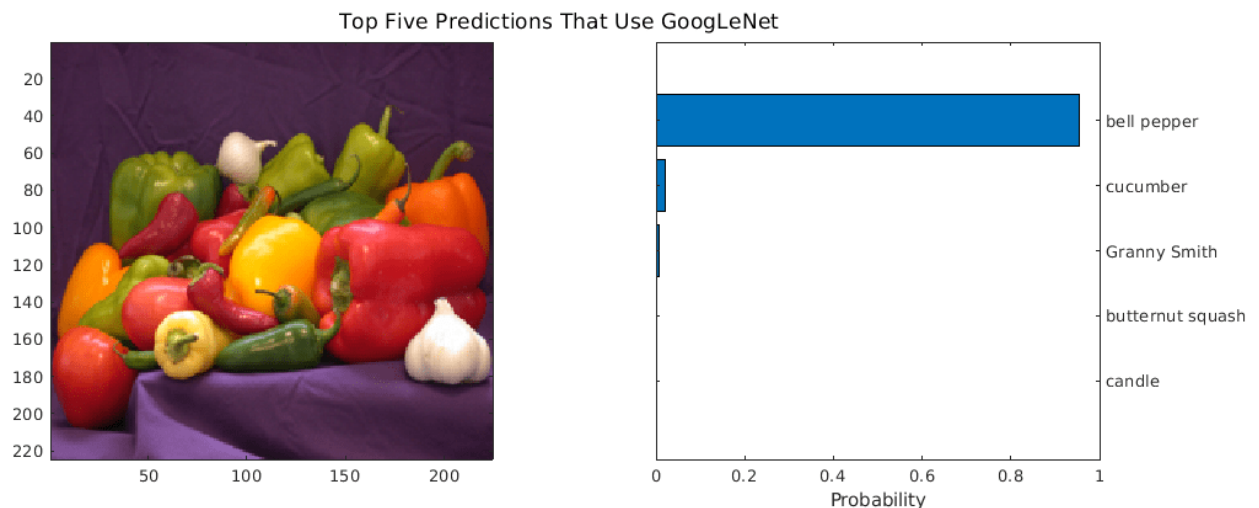
```
im = imresize(im, [224,224]);
predict_scores = googlenet_predict_mex(double(im));
```

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

`codegen` | `coder.DeepLearningConfig` | `coder.checkGpuInstall` | `coder.loadDeepLearningNetwork` | `googlenet` | `mobilenetv2` | `resnet50`

### Objects

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

## See Also

### More About

- “Deep Learning in MATLAB” (Deep Learning Toolbox)
- “Generated CNN Class Hierarchy” on page 5-44
- “Supported Networks, Layers, and Classes” on page 5-5
- “Load Pretrained Networks for Code Generation” on page 5-45
- “Code Generation for Deep Learning Networks by Using cuDNN” on page 5-48
- “Code Generation for Deep Learning Networks by Using TensorRT” on page 5-57

## Lane Detection Optimized with GPU Coder

This example shows how to generate CUDA® code from a deep learning network, represented by a `SeriesNetwork` object. In this example, the series network is a convolutional neural network that can detect and output lane marker boundaries from an image.

### Prerequisites

- CUDA enabled NVIDIA® GPU.
- NVIDIA CUDA toolkit and driver.
- NVIDIA cuDNN library.
- OpenCV libraries for video read and image display operations.
- 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 SeriesNetwork

```
[laneNet, coeffMeans, coeffStdS] = getLaneDetectionNetworkGPU();
```

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.

```
laneNet.Layers
```

```
ans =
```

```
23x1 Layer array with layers:
```

1	'data'	Image Input	227×227×3 images with 'zerocenter' normaliz.
2	'conv1'	Convolution	96 11×11×3 convolutions with stride [4 4]
3	'relu1'	ReLU	ReLU
4	'norm1'	Cross Channel Normalization	cross channel normalization with 5 channel
5	'pool1'	Max Pooling	3×3 max pooling with stride [2 2] and pad
6	'conv2'	Convolution	256 5×5×48 convolutions with stride [1 1]
7	'relu2'	ReLU	ReLU
8	'norm2'	Cross Channel Normalization	cross channel normalization with 5 channel
9	'pool2'	Max Pooling	3×3 max pooling with stride [2 2] and pad
10	'conv3'	Convolution	384 3×3×256 convolutions with stride [1 1]

11	'relu3'	ReLU	ReLU
12	'conv4'	Convolution	384 3×3×192 convolutions with stride [1 1 1]
13	'relu4'	ReLU	ReLU
14	'conv5'	Convolution	256 3×3×192 convolutions with stride [1 1 1]
15	'relu5'	ReLU	ReLU
16	'pool5'	Max Pooling	3×3 max pooling with stride [2 2] and padding [1 1]
17	'fc6'	Fully Connected	4096 fully connected layer
18	'relu6'	ReLU	ReLU
19	'drop6'	Dropout	50% dropout
20	'fcLane1'	Fully Connected	16 fully connected layer
21	'fcLane1Relu'	ReLU	ReLU
22	'fcLane2'	Fully Connected	6 fully connected layer
23	'output'	Regression Output	mean-squared-error with 'leftLane_a', 'leftLane_b', 'rightLane_a', 'rightLane_b'

### Examine Main Entry-Point Function

type `detect_lane.m`

```
function [laneFound, ltPts, rtPts] = detect_lane(frame, laneCoeffMeans, laneCoeffStds)
% From the networks output, compute left and right lane points in the
% image coordinates. The camera coordinates are described by the caltech
% mono camera model.

%#codegen

% 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 lanenet;

if isempty(lanenet)
    lanenet = coder.loadDeepLearningNetwork('laneNet.mat', 'lanenet');
end

lanecoefsNetworkOutput = lanenet.predict(permute(frame, [2 1 3]));

% Recover original coeffs by reversing the normalization steps

params = lanecoefsNetworkOutput .* laneCoeffStds + laneCoeffMeans;

isRightLaneFound = abs(params(6)) > 0.5; %c should be more than 0.5 for it to be a right lane
isLeftLaneFound = abs(params(3)) > 0.5;

vehicleXPoints = 3:30; %meters, ahead of the sensor
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

function yWorld = computeBoundaryModel(model, xWorld)
    yWorld = polyval(model, xWorld);
end

function tform = get_tformToImage
% Compute extrinsics based on camera setup
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

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

function intrinsicMat = intrinsicMatrix(FocalLength, Skew, PrincipalPoint)
intrinsicMat = ...
    [FocalLength(1) , 0 , 0; ...
     Skew , FocalLength(2) , 0; ...
     PrincipalPoint(1), PrincipalPoint(2), 1];

end

```

### Generate Code for Network and Post-Processing Code

The 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. The function `detect_lane.m` performs all these computations. Generate CUDA code for this function by creating a GPU code configuration object for a 'lib' 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.

```
cfg = coder.gpuConfig('lib');
cfg.DeepLearningConfig = coder.DeepLearningConfig('cudnn');
cfg.GenerateReport = true;
cfg.TargetLang = 'C++';
codegen -args {ones(227,227,3,'single'),ones(1,6,'double'),ones(1,6,'double')} -config cfg detect_lane.m
```

Code generation successful: To view the report, open('codegen/lib/detect\_lane/html/report.mldatx')

### Generated Code Description

The series network is generated as a C++ class containing an array of 23 layer classes.

```
class c_lanenet {
public:
    int32_T batchSize; int32_T numLayers; real32_T *inputData; real32_T
    *outputData; MWCNNLayer *layers[23];
public:
    c_lanenet(void); void setup(void); void predict(void); void
    cleanup(void); ~c_lanenet(void);
};
```

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 23 layers in the network.

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', 'lib', 'detect_lane');
dir(codegendir)
```

```
.          cnn_lanenet0_0_conv4_w.bin
..         cnn_lanenet0_0_conv5_b.bin
.gitignore  cnn_lanenet0_0_conv5_w.bin
DeepLearningNetwork.cu  cnn_lanenet0_0_data_offset.bin
DeepLearningNetwork.h  cnn_lanenet0_0_data_scale.bin
DeepLearningNetwork.o  cnn_lanenet0_0_fc6_b.bin
MWCNNLayerImpl.cu      cnn_lanenet0_0_fc6_w.bin
MWCNNLayerImpl.hpp     cnn_lanenet0_0_fcLane1_b.bin
MWCNNLayerImpl.o       cnn_lanenet0_0_fcLane1_w.bin
MWCudaDimUtility.cu    cnn_lanenet0_0_fcLane2_b.bin
MWCudaDimUtility.hpp   cnn_lanenet0_0_fcLane2_w.bin
MWCustomLayerForCuDNN.cpp  cnn_lanenet0_0_responseNames.txt
MWCustomLayerForCuDNN.hpp  codeInfo.mat
MWCustomLayerForCuDNN.o    codedescriptor.dmr
MWElementwiseAffineLayer.cpp  compileInfo.mat
MWElementwiseAffineLayer.hpp  defines.txt
MWElementwiseAffineLayer.o    detect_lane.a
MWElementwiseAffineLayerImpl.cu  detect_lane.cu
MWElementwiseAffineLayerImpl.hpp  detect_lane.h
```

```

MWElementwiseAffineLayerImpl.o      detect_lane.o
MWElementwiseAffineLayerImplKernel.cu detect_lane_data.cu
MWElementwiseAffineLayerImplKernel.o detect_lane_data.h
MWFusedConvReLULayer.cpp            detect_lane_data.o
MWFusedConvReLULayer.hpp            detect_lane_initialize.cu
MWFusedConvReLULayer.o              detect_lane_initialize.h
MWFusedConvReLULayerImpl.cu         detect_lane_initialize.o
MWFusedConvReLULayerImpl.hpp        detect_lane_ref.rsp
MWFusedConvReLULayerImpl.o          detect_lane_rtw.mk
MWKernelHeaders.hpp                 detect_lane_terminate.cu
MWTargetNetworkImpl.cu              detect_lane_terminate.h
MWTargetNetworkImpl.hpp             detect_lane_terminate.o
MWTargetNetworkImpl.o               detect_lane_types.h
buildInfo.mat                       examples
cnn_api.cpp                          gpu_codegen_info.mat
cnn_api.hpp                          html
cnn_api.o                            interface
cnn_lanenet0_0_conv1_b.bin           mean.bin
cnn_lanenet0_0_conv1_w.bin           predict.cu
cnn_lanenet0_0_conv2_b.bin           predict.h
cnn_lanenet0_0_conv2_w.bin           predict.o
cnn_lanenet0_0_conv3_b.bin           rtw_proj.tmw
cnn_lanenet0_0_conv3_w.bin           rtwtypes.h
cnn_lanenet0_0_conv4_b.bin

```

### Generate Additional Files for Post-Processing the Output

Export mean and std values from the trained network for use during execution.

```

codegen_dir = fullfile(pwd, 'codegen', 'lib', 'detect_lane');
fid = fopen(fullfile(codegen_dir, 'mean.bin'), 'w');
A = [coeffMeans coeffStds];
fwrite(fid, A, 'double');
fclose(fid);

```

### Main File

Compile the network code by using a main file. The main file uses the OpenCV VideoCapture method to read frames from the input video. Each frame is processed and classified until no more frames are read. Before displaying the output for each frame, the outputs are post-processed by using the `detect_lane` function generated in `detect_lane.cu`.

```
type main_lanenet.cu
```

```

/* Copyright 2016 The MathWorks, Inc. */

#include <stdio.h>
#include <stdlib.h>
#include <cuda.h>
#include <opencv2/opencv.hpp>
#include <opencv2/imgproc.hpp>
#include <opencv2/core/core.hpp>
#include <opencv2/core/types.hpp>
#include <opencv2/highgui.hpp>
#include <list>
#include <cmath>

```

```

#include "detect_lane.h"

using namespace cv;
void readData(float *input, Mat& orig, Mat & im)
{
    Size size(227,227);
    resize(orig,im,size,0,0,INTER_LINEAR);
    for(int j=0;j<227*227;j++)
    {
        //BGR to RGB
        input[2*227*227+j]=(float)(im.data[j*3+0]);
        input[1*227*227+j]=(float)(im.data[j*3+1]);
        input[0*227*227+j]=(float)(im.data[j*3+2]);
    }
}

void addLane(float pts[28][2], Mat & im, int numPts)
{
    std::vector<Point2f> iArray;
    for(int k=0; k<numPts; k++)
    {
        iArray.push_back(Point2f(pts[k][0],pts[k][1]));
    }
    Mat curve(iArray, true);
    curve.convertTo(curve, CV_32S); //adapt type for polylines
    polyLines(im, curve, false, CV_RGB(255,255,0), 2, LINE_AA);
}

void writeData(float *outputBuffer, Mat & im, int N, double means[6], double stds[6])
{
    // get lane coordinates
    boolean_T laneFound = 0;
    float ltPts[56];
    float rtPts[56];
    detect_lane(outputBuffer, means, stds, &laneFound, ltPts, rtPts);

    if (!laneFound)
    {
        return;
    }

    float ltPtsM[28][2];
    float rtPtsM[28][2];
    for(int k=0; k<28; k++)
    {
        ltPtsM[k][0] = ltPts[k];
        ltPtsM[k][1] = ltPts[k+28];
        rtPtsM[k][0] = rtPts[k];
        rtPtsM[k][1] = rtPts[k+28];
    }

    addLane(ltPtsM, im, 28);
    addLane(rtPtsM, im, 28);
}

void readMeanAndStds(const char* filename, double means[6], double stds[6])
{

```

```

FILE* pFile = fopen(filename, "rb");
if (pFile==NULL)
{
    fputs ("File error",stderr);
    return;
}

// obtain file size
fseek (pFile , 0 , SEEK_END);
long lSize = ftell(pFile);
rewind(pFile);

double* buffer = (double*)malloc(lSize);

size_t result = fread(buffer,sizeof(double),lSize,pFile);
if (result*sizeof(double) != lSize) {
    fputs ("Reading error",stderr);
    return;
}

for (int k = 0 ; k < 6; k++)
{
    means[k] = buffer[k];
    stds[k] = buffer[k+6];
}
free(buffer);
}

// Main function
int main(int argc, char* argv[])
{

    float *inputBuffer = (float*)calloc(sizeof(float),227*227*3);
    float *outputBuffer = (float*)calloc(sizeof(float),6);

    if ((inputBuffer == NULL) || (outputBuffer == NULL)) {
        printf("ERROR: Input/Output buffers could not be allocated!\n");
        exit(-1);
    }

    // get ground truth mean and std
    double means[6];
    double stds[6];
    readMeanAndStds("mean.bin", means, stds);

    if (argc < 2)
    {
        printf("Pass in input video file name as argument\n");
        return -1;
    }

    VideoCapture cap(argv[1]);
    if (!cap.isOpened()) {
        printf("Could not open the video capture device.\n");
        return -1;
    }
}

```

```

    cudaEvent_t start, stop;
    float fps = 0;
    cudaEventCreate(&start);
    cudaEventCreate(&stop);
    Mat orig, im;
    namedWindow("Lane detection demo", WINDOW_NORMAL);
    while(true)
    {
        cudaEventRecord(start);
        cap >> orig;
        if (orig.empty()) break;
        readData(inputBuffer, orig, im);

        writeData(inputBuffer, orig, 6, means, stds);

        cudaEventRecord(stop);
        cudaEventSynchronize(stop);

        char strbuf[50];
        float milliseconds = -1.0;
        cudaEventElapsedTime(&milliseconds, start, stop);
        fps = fps*.9+1000.0/milliseconds*.1;
        sprintf (strbuf, "%.2f FPS", fps);
        putText(orig, strbuf, Point(200,30), FONT_HERSHEY_DUPLEX, 1, CV_RGB(0,0,0), 2);
        imshow("Lane detection demo", orig);
        if( waitKey(50)%256 == 27 ) break; // stop capturing by pressing ESC    */
    }
    destroyWindow("Lane detection demo");

    free(inputBuffer);
    free(outputBuffer);

    return 0;
}

```

### Download Example Video

```

if ~exist('./caltech_cordova1.avi', 'file')
    url = 'https://www.mathworks.com/supportfiles/gpucoder/media/caltech_cordova1.avi';
    websave('caltech_cordova1.avi', url);
end

```

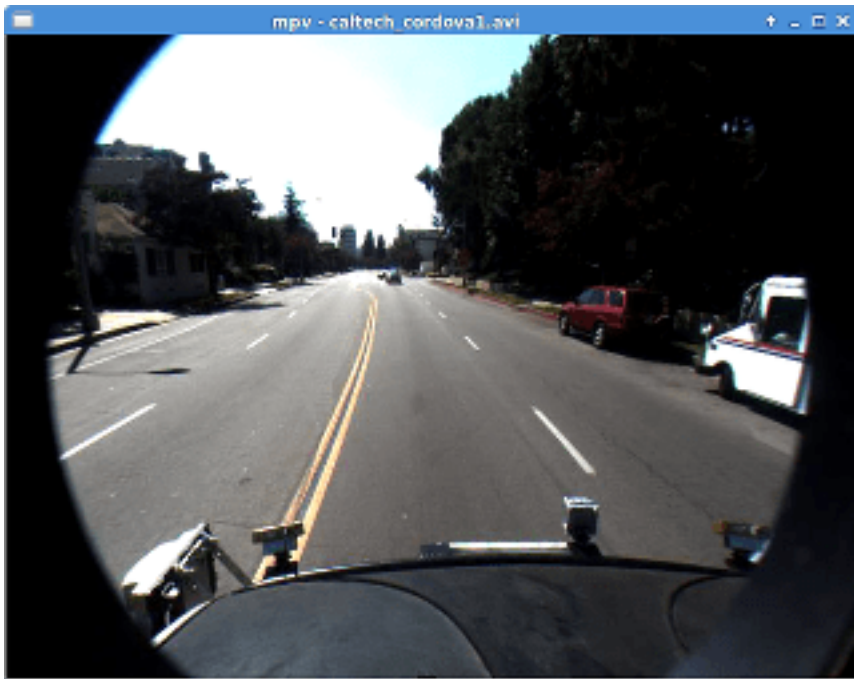
### Build Executable

```

if ispc
    setenv('MATLAB_ROOT', matlabroot);
    vcvarsall = mex.getCompilerConfigurations('C++').Details.CommandLineShell;
    setenv('VCVARSALL', vcvarsall);
    system('make_win_lane_detection.bat');
    cd(codegendir);
    system('lanenet.exe ..\..\..\caltech_cordova1.avi');
else
    setenv('MATLAB_ROOT', matlabroot);
    system('make -f Makefile_lane_detection.mk');
    cd(codegendir);
    system('./lanenet ../../../../caltech_cordova1.avi');
end

```

## Input Screenshot



## Output Screenshot



## See Also

### Functions

`codegen` | `coder.DeepLearningConfig` | `coder.checkGpuInstall` | `coder.loadDeepLearningNetwork`

### Objects

`coder.CuDNNConfig` | `coder.TensorRTConfig` | `coder.gpuConfig` | `coder.gpuEnvConfig`

## **See Also**

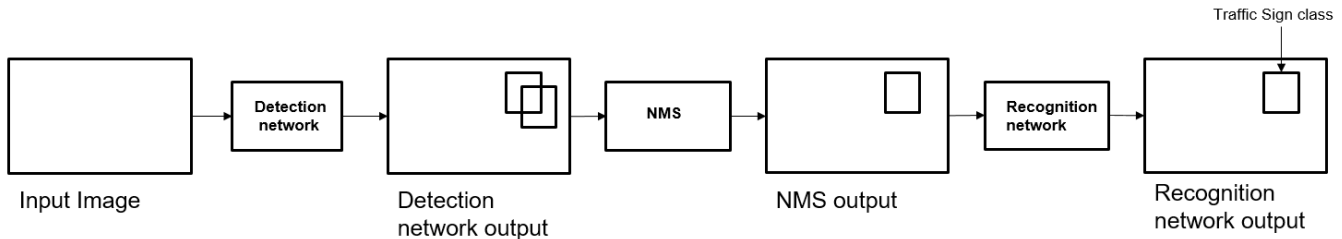
### **More About**

- “Generated CNN Class Hierarchy” on page 5-44
- “Supported Networks, Layers, and Classes” on page 5-5
- “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 SeriesNetwork

Download the detection and recognition networks.

```
getTsd();
```

The detection network contains 58 layers including convolution, leaky ReLU, and fully connected layers.

```
load('yolo_tsr.mat');
yolo.Layers
```

```
ans =
```

```
58x1 Layer array with layers:
```

1	'input'	Image Input	448×448×3 images
2	'conv1'	Convolution	64 7×7×3 convolutions with stride [2 2] and padding
3	'relu1'	Leaky ReLU	Leaky ReLU with scale 0.1
4	'pool1'	Max Pooling	2×2 max pooling with stride [2 2] and padding
5	'conv2'	Convolution	192 3×3×64 convolutions with stride [1 1] and padding
6	'relu2'	Leaky ReLU	Leaky ReLU with scale 0.1
7	'pool2'	Max Pooling	2×2 max pooling with stride [2 2] and padding
8	'conv3'	Convolution	128 1×1×192 convolutions with stride [1 1] and padding
9	'relu3'	Leaky ReLU	Leaky ReLU with scale 0.1
10	'conv4'	Convolution	256 3×3×128 convolutions with stride [1 1] and padding
11	'relu4'	Leaky ReLU	Leaky ReLU with scale 0.1
12	'conv5'	Convolution	256 1×1×256 convolutions with stride [1 1] and padding
13	'relu5'	Leaky ReLU	Leaky ReLU with scale 0.1
14	'conv6'	Convolution	512 3×3×256 convolutions with stride [1 1] and padding
15	'relu6'	Leaky ReLU	Leaky ReLU with scale 0.1
16	'pool6'	Max Pooling	2×2 max pooling with stride [2 2] and padding
17	'conv7'	Convolution	256 1×1×512 convolutions with stride [1 1] and padding
18	'relu7'	Leaky ReLU	Leaky ReLU with scale 0.1
19	'conv8'	Convolution	512 3×3×256 convolutions with stride [1 1] and padding
20	'relu8'	Leaky ReLU	Leaky ReLU with scale 0.1
21	'conv9'	Convolution	256 1×1×512 convolutions with stride [1 1] and padding
22	'relu9'	Leaky ReLU	Leaky ReLU with scale 0.1

23	'conv10'	Convolution	512 3×3×256 convolutions with stride [1 1] and
24	'relu10'	Leaky ReLU	Leaky ReLU with scale 0.1
25	'conv11'	Convolution	256 1×1×512 convolutions with stride [1 1] and
26	'relu11'	Leaky ReLU	Leaky ReLU with scale 0.1
27	'conv12'	Convolution	512 3×3×256 convolutions with stride [1 1] and
28	'relu12'	Leaky ReLU	Leaky ReLU with scale 0.1
29	'conv13'	Convolution	256 1×1×512 convolutions with stride [1 1] and
30	'relu13'	Leaky ReLU	Leaky ReLU with scale 0.1
31	'conv14'	Convolution	512 3×3×256 convolutions with stride [1 1] and
32	'relu14'	Leaky ReLU	Leaky ReLU with scale 0.1
33	'conv15'	Convolution	512 1×1×512 convolutions with stride [1 1] and
34	'relu15'	Leaky ReLU	Leaky ReLU with scale 0.1
35	'conv16'	Convolution	1024 3×3×512 convolutions with stride [1 1] and
36	'relu16'	Leaky ReLU	Leaky ReLU with scale 0.1
37	'pool16'	Max Pooling	2×2 max pooling with stride [2 2] and padding
38	'conv17'	Convolution	512 1×1×1024 convolutions with stride [1 1] and
39	'relu17'	Leaky ReLU	Leaky ReLU with scale 0.1
40	'conv18'	Convolution	1024 3×3×512 convolutions with stride [1 1] and
41	'relu18'	Leaky ReLU	Leaky ReLU with scale 0.1
42	'conv19'	Convolution	512 1×1×1024 convolutions with stride [1 1] and
43	'relu19'	Leaky ReLU	Leaky ReLU with scale 0.1
44	'conv20'	Convolution	1024 3×3×512 convolutions with stride [1 1] and
45	'relu20'	Leaky ReLU	Leaky ReLU with scale 0.1
46	'conv21'	Convolution	1024 3×3×1024 convolutions with stride [1 1] and
47	'relu21'	Leaky ReLU	Leaky ReLU with scale 0.1
48	'conv22'	Convolution	1024 3×3×1024 convolutions with stride [2 2] and
49	'relu22'	Leaky ReLU	Leaky ReLU with scale 0.1
50	'conv23'	Convolution	1024 3×3×1024 convolutions with stride [1 1] and
51	'relu23'	Leaky ReLU	Leaky ReLU with scale 0.1
52	'conv24'	Convolution	1024 3×3×1024 convolutions with stride [1 1] and
53	'relu24'	Leaky ReLU	Leaky ReLU with scale 0.1
54	'fc25'	Fully Connected	4096 fully connected layer
55	'relu25'	Leaky ReLU	Leaky ReLU with scale 0.1
56	'fc26'	Fully Connected	539 fully connected layer
57	'softmax'	Softmax	softmax
58	'classoutput'	Classification Output	crossentropyex with '1' and 538 other classes

The recognition network contains 14 layers including convolution, fully connected, and the classification output layers.

```
load('RecognitionNet.mat');
convnet.Layers
```

```
ans =
```

```
14×1 Layer array with layers:
```

1	'imageinput'	Image Input	48×48×3 images with 'zerocenter' normalization and
2	'conv_1'	Convolution	100 7×7×3 convolutions with stride [1 1] and padding
3	'relu_1'	ReLU	ReLU
4	'maxpool_1'	Max Pooling	2×2 max pooling with stride [2 2] and padding
5	'conv_2'	Convolution	150 4×4×100 convolutions with stride [1 1] and padding
6	'relu_2'	ReLU	ReLU
7	'maxpool_2'	Max Pooling	2×2 max pooling with stride [2 2] and padding
8	'conv_3'	Convolution	250 4×4×150 convolutions with stride [1 1] and padding
9	'maxpool_3'	Max Pooling	2×2 max pooling with stride [2 2] and padding
10	'fc_1'	Fully Connected	300 fully connected layer

```
11 'dropout'          Dropout          90% dropout
12 'fc_2'            Fully Connected  35 fully connected layer
13 'softmax'         Softmax         softmax
14 'classoutput'     Classification Output crossentropyex with '0' and 34 other classes
```

### 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)
%#codegen

% This function detects the traffic signs in the image using Detection Network
% (modified version of Yolo) and recognizes(classifies) using Recognition Network
%
% Inputs :
%
% im          : Input test image
%
% Outputs :
%
% selectedBbox : Detected bounding boxes
% idx          : Corresponding classes

% Copyright 2017-2019 The MathWorks, Inc.

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('yolo_tsr.mat','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);
```



```
% Resize Image
inpImg(:,:,i) = imresize(img(ymin:ymin, xmin:xmax, :), [48,48]);
end

for i = 1:size(selectedBbox,1)
    output = recognitionnet.predict(inpImg(:,:,i));
    [~,idx(i)]=max(output);
end
```

### 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');
codegen -config cfg tsdr_predict -args {ones(480,704,3,'uint8')} -report
```

Code generation successful: To view the report, open('codegen/mex/tsdr\_predict/html/report.mldata')

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);
```

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},'F',...
    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.

```
% Input video
v = VideoReader('stop.avi');
fps = 0;

while hasFrame(v)
    % Take a frame
    picture = readFrame(v);
    picture = imresize(picture,[920,1632]);
    % Call MEX function for Traffic Sign Detection and Recognition
    tic;
    [bboxes,clases] = tsdr_predict_mex(picture);
    newt = toc;

    % fps
    fps = .9*fps + .1*(1/newt);

    % display

    displayDetections(picture,bboxes,clases,fps);
end
```

Clear the static network objects that were loaded into memory.



```
clear mex;
```

## See Also

### Functions

`codegen` | `coder.DeepLearningConfig` | `coder.checkGpuInstall` |  
`coder.loadDeepLearningNetwork`

### Objects

`coder.CuDNNConfig` | `coder.TensorRTConfig` | `coder.gpuConfig` | `coder.gpuEnvConfig`

## More About

- “Supported Networks, Layers, and Classes” on page 5-5
- “Load Pretrained Networks for Code Generation” on page 5-45
- “Code Generation for Deep Learning Networks by Using cuDNN” on page 5-48
- “Code Generation for Deep Learning Networks by Using TensorRT” on page 5-57
- “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 the `codegen` command to generate a MEX function that performs prediction on a `SeriesNetwork` object called `LogoNet`.

### 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 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 (logonet) was developed in MATLAB® and can recognize 32 logos under various lighting conditions and camera motions. Because this network focuses only on recognition, you can use it in applications where localization is not required.

### Training the Network

The network is trained in MATLAB by using training data that contains around 200 images for each logo. Because the number of images for training the network is small, data augmentation increases the number of training samples. Four types of data augmentation are used: contrast normalization, Gaussian blur, random flipping, and shearing. This data augmentation helps in recognizing logos in images captured by different lighting conditions and camera motions. The input size for logonet is [227 227 3]. Standard SGDM trains by using a learning rate of 0.0001 for 40 epochs with a mini-batch size of 45. The `trainLogonet.m` helper script demonstrates the data augmentation on a sample image, architecture of the logonet, and training options.

## Get Pretrained SeriesNetwork

Download the logonet network and save it to `LogoNet.mat`.

```
getLogonet();
```

The saved network contains 22 layers including convolution, fully connected, and the classification output layers.

```
load('LogoNet.mat');
convnet.Layers
```

```
ans =
```

```
22x1 Layer array with layers:
```

1	'imageinput'	Image Input	227×227×3 images with 'zerocenter' normalization
2	'conv_1'	Convolution	96 5×5×3 convolutions with stride [1 1] and padding
3	'relu_1'	ReLU	ReLU
4	'maxpool_1'	Max Pooling	3×3 max pooling with stride [2 2] and padding
5	'conv_2'	Convolution	128 3×3×96 convolutions with stride [1 1] and padding
6	'relu_2'	ReLU	ReLU
7	'maxpool_2'	Max Pooling	3×3 max pooling with stride [2 2] and padding
8	'conv_3'	Convolution	384 3×3×128 convolutions with stride [1 1] and padding
9	'relu_3'	ReLU	ReLU
10	'maxpool_3'	Max Pooling	3×3 max pooling with stride [2 2] and padding
11	'conv_4'	Convolution	128 3×3×384 convolutions with stride [2 2] and padding
12	'relu_4'	ReLU	ReLU
13	'maxpool_4'	Max Pooling	3×3 max pooling with stride [2 2] and padding
14	'fc_1'	Fully Connected	2048 fully connected layer
15	'relu_5'	ReLU	ReLU
16	'dropout_1'	Dropout	50% dropout
17	'fc_2'	Fully Connected	2048 fully connected layer
18	'relu_6'	ReLU	ReLU
19	'dropout_2'	Dropout	50% dropout
20	'fc_3'	Fully Connected	32 fully connected layer
21	'softmax'	Softmax	softmax
22	'classoutput'	Classification Output	crossentropyex with 'adidas' and 31 other classes

## The logonet\_predict Entry-Point Function

The `logonet_predict.m` entry-point function takes an image input and performs prediction on the image 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 2017-2019 The MathWorks, Inc.

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 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 `[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: To view the report, open('codegen/mex/logonet\_predict/html/report.mlx')

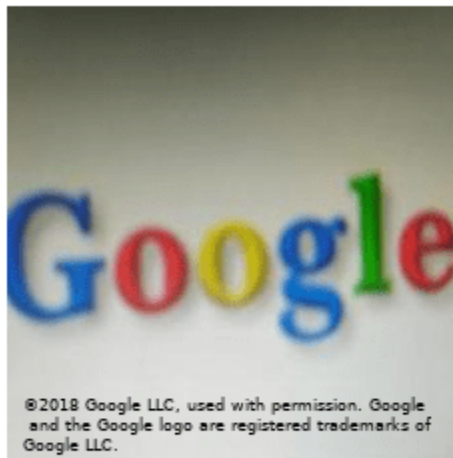
### Run Generated MEX

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 = {'adidas', 'aldi', 'apple', 'becks', 'bmw', 'carlsberg', ...
            'chimay', 'cocacola', 'corona', 'dhl', 'erdinger', 'esso', 'fedex', ...

```

```

    'ferrari', 'ford', 'fosters', 'google', 'guinness', 'heineken', 'hp',...
    'milka', 'nvidia', 'paulaner', 'pepsi', 'rittersport', 'shell', 'singha', 'starbucks', 'stel

[val,indx] = sort(predict_scores, 'descend');
scores = val(1:5)*100;
top5labels = synsetOut(indx(1:5));

Display the top five classification labels.

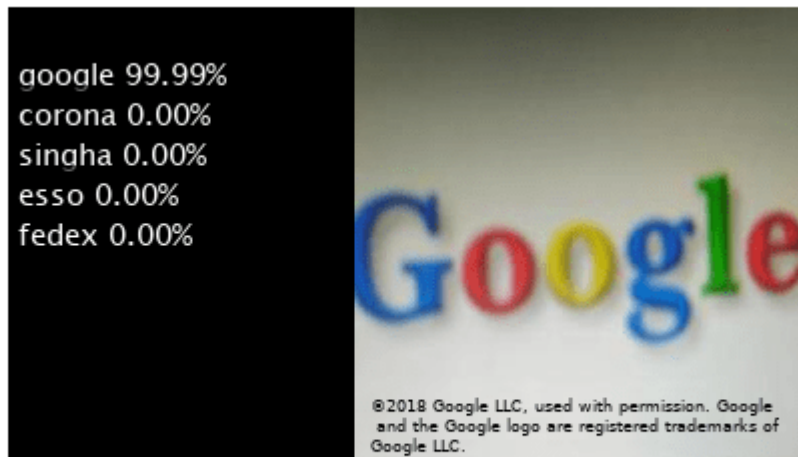
outputImage = zeros(227,400,3, 'uint8');
for k = 1:3
    outputImage(:,174:end,k) = im(:, :,k);
end

scol = 1;
srow = 20;

for k = 1:5
    outputImage = insertText(outputImage, [scol, srow], [top5labels{k}, ' ', num2str(scores(k), '%
    srow = srow + 20;
end

imshow(outputImage);

```



Clear the static network object that was loaded in memory.

```
clear mex;
```

## See Also

### Functions

codegen | coder.DeepLearningConfig | coder.checkGpuInstall |  
 coder.loadDeepLearningNetwork

**Objects**

`coder.CuDNNConfig` | `coder.TensorRTConfig` | `coder.gpuConfig` | `coder.gpuEnvConfig`

**More About**

- “Code Generation for Deep Learning Networks by Using cuDNN” on page 5-48
- “Code Generation for Deep Learning Networks by Using TensorRT” on page 5-57

## Pedestrian Detection

This example shows code generation for pedestrian detection application that uses deep learning. Pedestrian detection is a key issue in computer vision. Pedestrian detection has several applications in the fields of autonomous driving, surveillance, robotics, and so on.

### Prerequisites

- CUDA® enabled NVIDIA® GPU.
- NVIDIA CUDA toolkit and driver.
- NVIDIA cuDNN.
- 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”.
- GPU Coder Interface for Deep Learning Libraries support package. To install this support package, use the Add-On Explorer.

### Verify GPU Environment

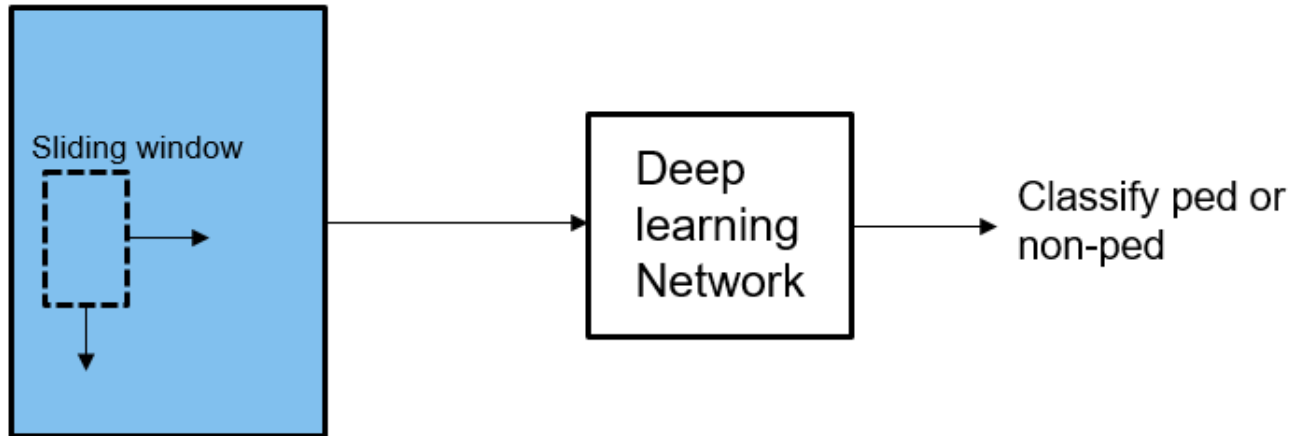
Use the `coder.checkGpuInstall` function to verify that the compilers and libraries necessary for running this example are set up correctly.

```
envCfg = coder.gpuEnvConfig('host');  
envCfg.DeepLibTarget = 'cudnn';  
envCfg.DeepCodegen = 1;  
envCfg.Quiet = 1;  
coder.checkGpuInstall(envCfg);
```

### The Pedestrian Detection Network

The pedestrian detection network was trained by using images of pedestrians and non-pedestrians. This network is trained in MATLAB® by using the `trainPedNet.m` helper script. A sliding window approach crops patches from an image of size [64 32]. Patch dimensions are obtained from a heatmap, which represents the distribution of pedestrians in the images in the data set. It indicates the presence of pedestrians at various scales and locations in the images. In this example, patches of pedestrians close to the camera are cropped and processed. Non-Maximal Suppression (NMS) is applied on the obtained patches to merge them and detect complete pedestrians.

## Input Image



The pedestrian detection network contains 12 layers which include convolution, fully connected, and classification output layers.

```
load('PedNet.mat');
PedNet.Layers
```

```
ans =
```

```
12x1 Layer array with layers:
```

1	'imageinput'	Image Input	64x32x3 images with 'zerocenter' normalization
2	'conv_1'	Convolution	20 5x5x3 convolutions with stride [1 1] and padding [0 0]
3	'relu_1'	ReLU	ReLU
4	'maxpool_1'	Max Pooling	2x2 max pooling with stride [2 2] and padding [0 0]
5	'crossnorm'	Cross Channel Normalization	cross channel normalization with 5 channels
6	'conv_2'	Convolution	20 5x5x20 convolutions with stride [1 1] and padding [0 0]
7	'relu_2'	ReLU	ReLU
8	'maxpool_2'	Max Pooling	2x2 max pooling with stride [2 2] and padding [0 0]
9	'fc_1'	Fully Connected	512 fully connected layer
10	'fc_2'	Fully Connected	2 fully connected layer
11	'softmax'	Softmax	softmax
12	'classoutput'	Classification Output	crossentropyex with classes 'NonPed' and 'Ped'

### The pedDetect\_predict Entry-Point Function

The `pedDetect_predict.m` entry-point function takes an image input and performs prediction on an image by using the deep learning network saved in the `PedNet.mat` file. The function loads the network object from the `PedNet.mat` file into a persistent variable `pednet`. Then function reuses the persistent object on subsequent calls.

```
type('pedDetect_predict.m')
```

```
function selectedBbox = pedDetect_predict(img)
%#codegen
```

```
% Copyright 2017-2019 The MathWorks, Inc.
```



```

coder.gpu.kerelfun;

persistent pednet;
if isempty(pednet)
    pednet = coder.loadDeepLearningNetwork(coder.const('PedNet.mat'),'Pedestrian_Detection');
end

[imgHt , imgWd , ~] = size(img);
VrHt = [imgHt - 30 , imgHt]; % Two bands of vertical heights are considered

% patchHt and patchWd are obtained from heat maps (heat map here refers to
% pedestrians data represented in the form of a map with different
% colors. Different colors indicate presence of pedestrians at various
% scales).
patchHt = 300;
patchWd = patchHt/3;

% PatchCount is used to estimate number of patches per image
PatchCount = ((imgWd - patchWd)/20) + 2;
maxPatchCount = PatchCount * 2;
Itmp = zeros(64 , 32 , 3 , maxPatchCount);
ltMin = zeros(maxPatchCount);
lttp = zeros(maxPatchCount);

idx = 1; % To count number of image patches obtained from sliding window
cnt = 1; % To count number of patches predicted as pedestrians

bbox = zeros(maxPatchCount , 4);
value = zeros(maxPatchCount , 1);

%% Region proposal for two bands
for VrStride = 1 : 2
    for HrStride = 1 : 20 : (imgWd - 60) % Obtain horizontal patches with stride 20.
        ltMin(idx) = HrStride + 1;
        rtMax = min(ltMin(idx) + patchWd , imgWd);
        lttp(idx) = (VrHt(VrStride) - patchHt);
        It = img(lttp(idx): VrHt(VrStride) , ltMin(idx) : rtMax , :);
        Itmp(:,:,,idx) = imresize(It,[64,32]);
        idx = idx + 1;
    end
end

for j = 1 : size (Itmp,4)
    score = pednet.predict(Itmp(:,:,,j)); % Classify ROI
    % accuracy of detected box should be greater than 0.90
    if (score(1,2) > 0.80)
        bbox(cnt,:) = [ltMin(j),lttp(j), patchWd , patchHt];
        value(cnt,:) = score(1,2);
        cnt = cnt + 1;
    end
end

%% NMS to merge similar boxes
if ~isempty(bbox)
    [selectedBbox,~] = selectStrongestBbox(bbox(1:cnt-1,:),...
        value(1:cnt-1:),'OverlapThreshold',0.002);
end

```

```
end
```

### Generate CUDA MEX for the pedDetect\_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 size of the input image. This value corresponds to the input layer size of pedestrian detection network.

```
% Load an input image.  
im = imread('test.jpg');  
im = imresize(im,[480,640]);  
  
cfg = coder.gpuConfig('mex');  
cfg.TargetLang = 'C++';  
cfg.DeepLearningConfig = coder.DeepLearningConfig('cudnn');  
codegen -config cfg pedDetect_predict -args {im} -report
```

Code generation successful: To view the report, open('codegen/mex/pedDetect\_predict/html/report.r

### Run Generated MEX

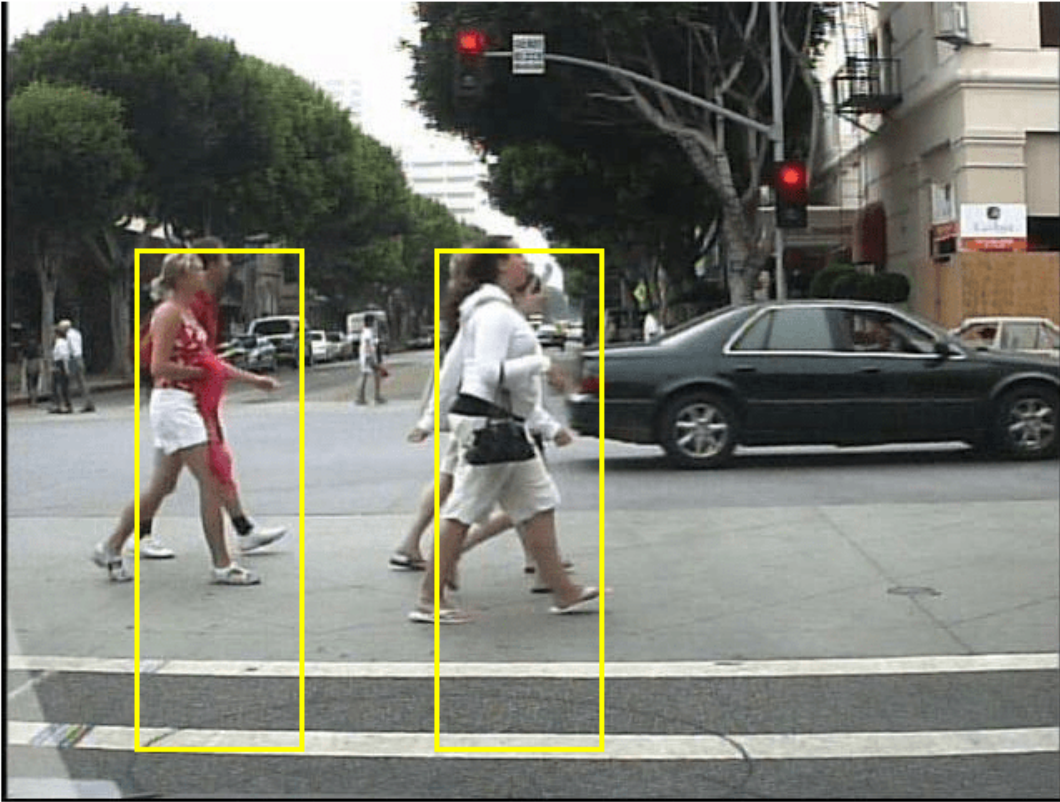
Call `pednet_predict_mex` on the input image.

```
imshow(im);  
ped_bboxes = pedDetect_predict_mex(im);
```



Display the final predictions.

```
outputImage = insertShape(im, 'Rectangle', ped_bboxes, 'LineWidth', 3);  
imshow(outputImage);
```



### Classification on Video

The included helper file `pedDetect_predict.m` grabs frames from a video, performs prediction, and displays the classification results on each of the captured video frames.

```
v = VideoReader('LiveData.avi');
fps = 0;
while hasFrame(v)
    % Read frames from video
    im = readFrame(v);
    im = imresize(im,[480,640]);

    % Call MEX function for pednet prediction
    tic;
    ped_bboxes = pedDetect_predict_mex(im);
    newt = toc;

    % fps
    fps = .9*fps + .1*(1/newt);

    % display
    outputImage = insertShape(im,'Rectangle',ped_bboxes,'LineWidth',3);
    imshow(outputImage)
    pause(0.2)
end
```

Clear the static network object that was loaded in memory.

```
clear mex;
```

## See Also

### Functions

[VideoReader](#) | [codegen](#) | [coder.DeepLearningConfig](#) | [coder.checkGpuInstall](#) | [coder.loadDeepLearningNetwork](#)

### Objects

[coder.CuDNNConfig](#) | [coder.TensorRTConfig](#) | [coder.gpuConfig](#) | [coder.gpuEnvConfig](#)

## More About

- “Code Generation for Deep Learning Networks by Using cuDNN” on page 5-48
- “Code Generation for Deep Learning Networks by Using TensorRT” on page 5-57
- “Choose Function to Visualize Detected Objects” (Computer Vision Toolbox)

## Deep Learning Prediction by Using NVIDIA TensorRT

This example shows code generation for a deep learning application by using the NVIDIA TensorRT™ library. It uses the `codegen` command to generate a MEX file to perform prediction with a ResNet-50 image classification network by using TensorRT. A second example demonstrates usage of `codegen` command to generate a MEX file that performs 8-bit integer prediction by using TensorRT for a logo classification 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 and 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);
```

### The `resnet_predict` Entry-Point Function

This example uses the DAG network ResNet-50 to show image classification by using TensorRT. 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.

The `resnet_predict.m` function loads the ResNet-50 network into a persistent network object and reuses the persistent object on subsequent prediction calls.

```
type('resnet_predict.m')

% Copyright 2020 The MathWorks, Inc.

function out = resnet_predict(in)
%#codegen

% 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, avoiding reconstructing and reloading the
% network object.
```

```
persistent mynet;
```

```
if isempty(mynet)
    % Call the function resnet50 that returns a DAG network
    % for ResNet-50 model.
    mynet = coder.loadDeepLearningNetwork('resnet50','resnet');
end
```

```
% pass in input
out = mynet.predict(in);
```

### Run MEX Code Generation

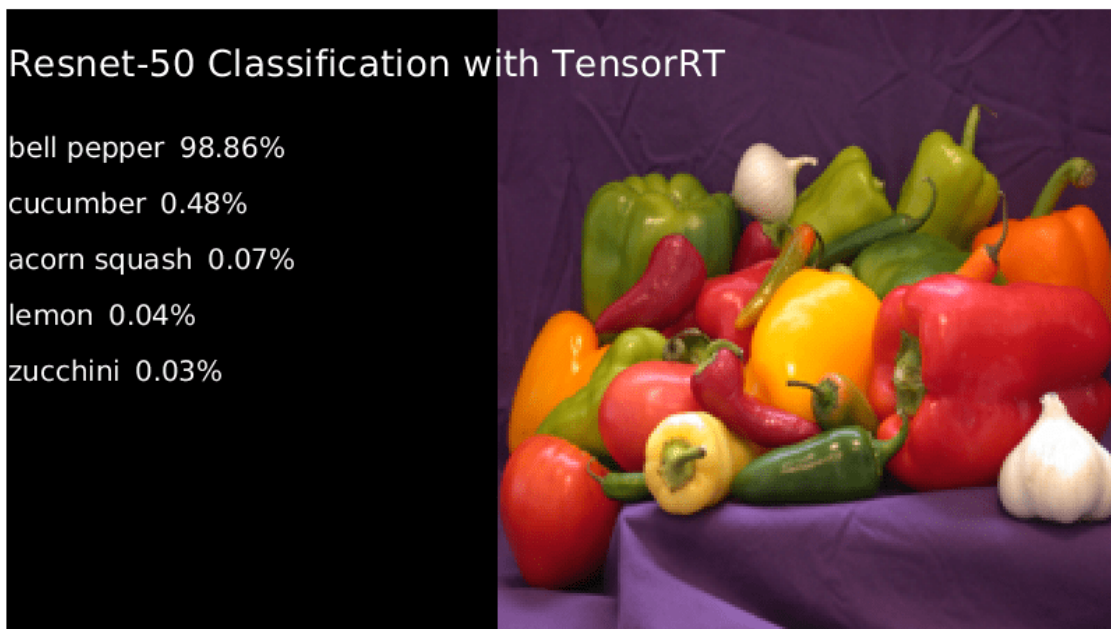
To generate CUDA code for the `resnet_predict` entry-point function, create a GPU code configuration object for a MEX target and set the target language to C++. Use the `coder.DeepLearningConfig` function to create a TensorRT deep learning configuration object and assign it to the `DeepLearningConfig` property of the GPU code configuration object. Run the `codegen` command specifying an input size of `[224,224,3]`. This value corresponds to the input layer size of ResNet-50 network.

```
cfg = coder.gpuConfig('mex');
cfg.TargetLang = 'C++';
cfg.DeepLearningConfig = coder.DeepLearningConfig('tensorrt');
codegen -config cfg resnet_predict -args {ones(224,224,3)} -report
```

Code generation successful: To view the report, open('codegen/mex/resnet\_predict/html/report.mld

### Perform Prediction on Test Image

```
im = imread('peppers.png');
im = imresize(im, [224,224]);
predict_scores = resnet_predict_mex(double(im));
%
% get top 5 probability scores and their labels
[val,indx] = sort(predict_scores, 'descend');
scores = val(1:5)*100;
net = resnet50;
classnames = net.Layers(end).ClassNames;
labels = classnames(indx(1:5));
```



Clear the static network object that was loaded in memory.

```
clear mex;
```

### Generate TensorRT Code for INT8 Prediction

Generate TensorRT code that runs inference in int8 precision. Use a pretrained logo classification network to classify logos in images. Download the pretrained LogoNet network and save it as a `logonet.mat` file. The network was developed in MATLAB. This network can recognize 32 logos under various lighting conditions and camera angles. The network is pretrained in single precision floating-point format.

```
net = getLogonet();
```

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 containing images grouped by their corresponding classification labels. For int8 support, GPU compute capability must be 6.1 or higher.

```
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 {ones(227,227,3,'int8')} -report
```

Code generation successful: To view the report, open('codegen/mex/logonet\_predict/html/report.ml

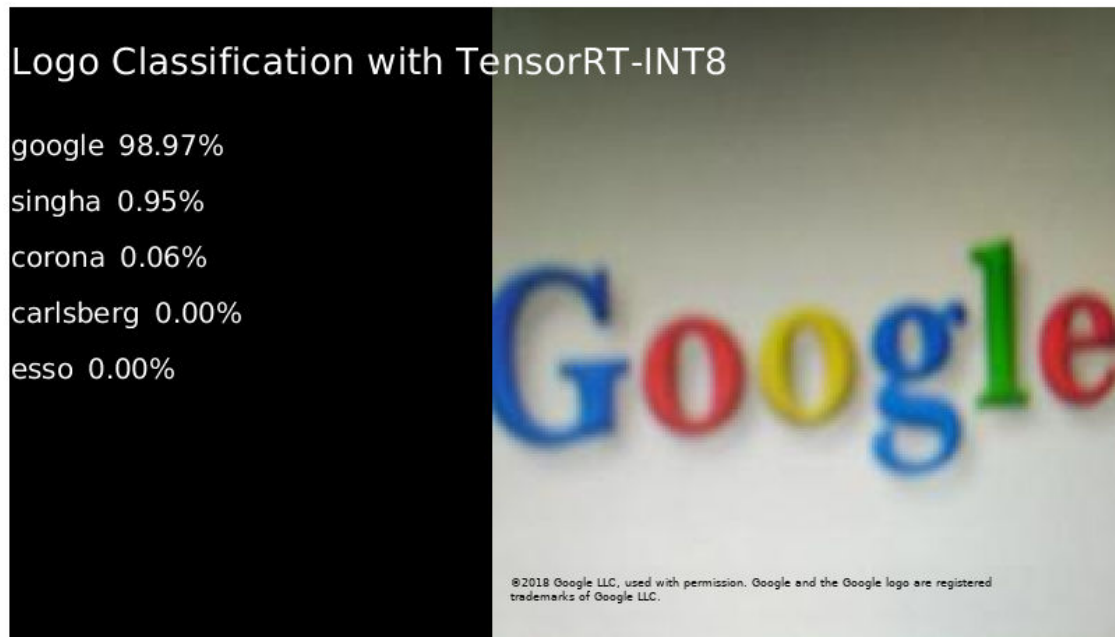


**Run INT8 Prediction on Test Image**

```

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;
labels = classnames(indx(1:5));

```



Clear the static network object that was loaded in memory.

```
clear mex;
```

**See Also****Functions**

codegen | coder.DeepLearningConfig | coder.checkGpuInstall |  
coder.loadDeepLearningNetwork

**Objects**

coder.CodeConfig | coder.EmbeddedCodeConfig | coder.TensorRTConfig |  
coder.gpuConfig | coder.gpuEnvConfig

## **See Also**

### **More About**

- “Supported Networks, Layers, and Classes” on page 5-5
- “Load Pretrained Networks for Code Generation” on page 5-45
- “Code Generation for Deep Learning Networks by Using TensorRT” on page 5-57

# 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 2018-2019 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();
```

Downloading pretrained SegNet (107 MB)...

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: To view the report, open('codegen/mex/segnet\_predict/html/report.mld')

### 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

`codegen` | `coder.DeepLearningConfig` | `coder.checkGpuInstall` |  
`coder.loadDeepLearningNetwork`

### Objects

`coder.CuDNNConfig` | `coder.gpuConfig` | `coder.gpuEnvConfig`

## 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 5-142
- “Code Generation for Semantic Segmentation Network That Uses U-net” on page 5-154

### **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.z
```

```
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.

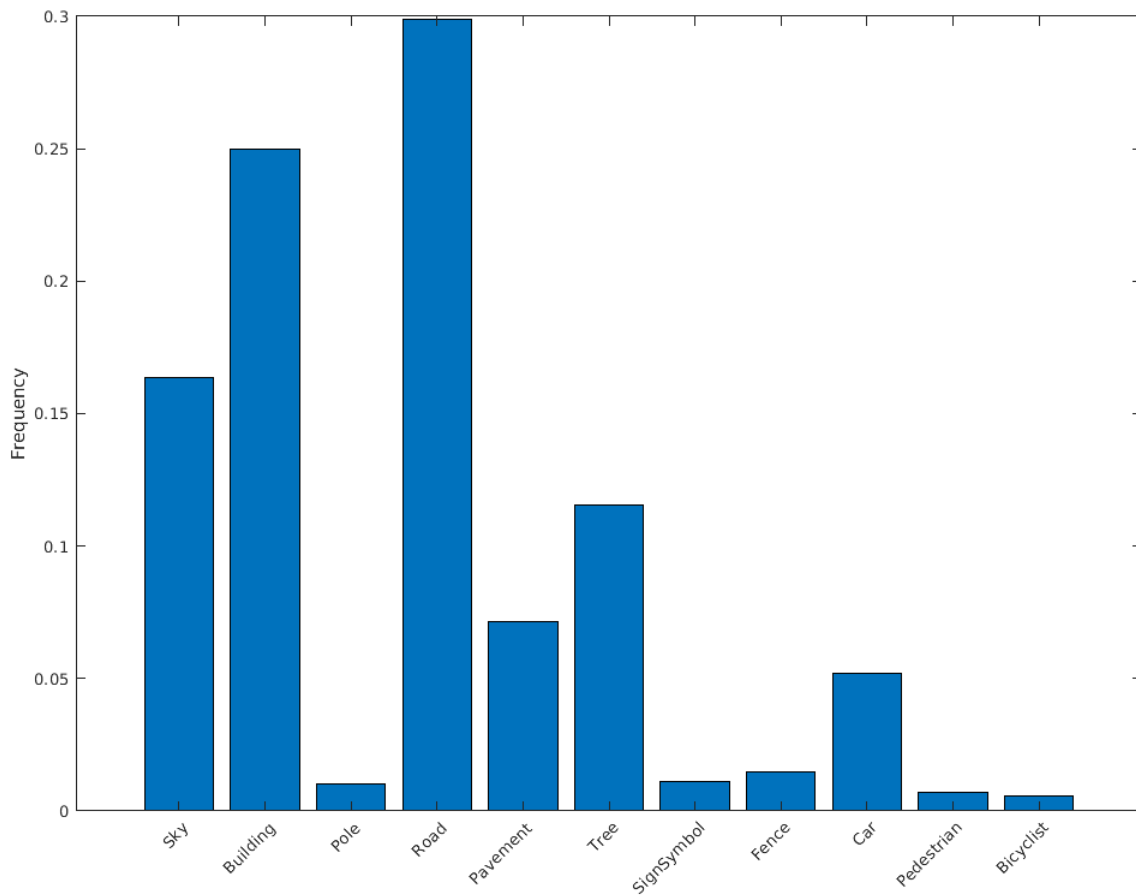
```
tbl = countEachLabel(pxds)
```

```
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 provides more examples to the network because it helps improve the accuracy of the network. Here, random left/right reflection and random X/Y translation of +/- 10 pixels is used for data augmentation. Use the `imageDataAugmenter` (Deep Learning Toolbox) function to specify these data augmentation parameters.

```
augmenter = imageDataAugmenter('RandXReflection', true, ...
    'RandXTranslation', [-10 10], 'RandYTranslation', [-10 10]);
```

The `imageDataAugmenter` function supports several other types of data augmentation. Choosing among them requires empirical analysis and is another level of hyperparameter tuning.

### Start Training

Combine the training data and data augmentation selections by using the `pixelLabelImageDatastore` (Computer Vision Toolbox) function. The `pixelLabelImageDatastore` function reads batches of training data, applies data augmentation, and sends the augmented data to the training algorithm.

```
pximds = pixelLabelImageDatastore(imdsTrain, pxdsTrain, ...
    'DataAugmentation', augmenter);
```

If the `doTraining` flag is true, start the training by using the `trainNetwork` (Deep Learning Toolbox) function.

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.

```
if doTraining
    [net, info] = trainNetwork(pximds,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"  
];
```

```
cmmap = camvidColorMap();  
SegmentedImage = labeloverlay(im, argmax, 'ColorMap', cmmap);  
figure  
imshow(SegmentedImage);  
pixelLabelColorbar(cmmap, classes);
```



### Cleanup

Clear the static network object that was loaded in memory.

```
clear mex;
```

### 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

`codegen` | `coder.DeepLearningConfig` | `coder.checkGpuInstall` |  
`coder.loadDeepLearningNetwork`

### Objects

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

## 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 5-137
- "Code Generation for Semantic Segmentation Network That Uses U-net" on page 5-154

## 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')

% OUT = segmentImageUnet(IM, PATCHSIZE) returns a semantically segmented
% image, segmented using the network multispectralUnet. The segmentation
% is performed over each patch of size PATCHSIZE.
%
% Copyright 2019-2020 The MathWorks, Inc.
function out = segmentImageUnet(im, patchSize)

%#codegen

persistent mynet;

if isempty(mynet)
    mynet = coder.loadDeepLearningNetwork('trainedUnet/multispectralUnet.mat');
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 DAG Network Object

```
trainedUnet_url = 'https://www.mathworks.com/supportfiles/vision/data/multispectralUnet.mat';
downloadTrainedUnet(trainedUnet_url,pwd);
```

Downloading Pre-trained U-net for Hamlin Beach dataset...  
This will take several minutes to download...  
done.

```
ld = load("trainedUnet/multispectralUnet.mat");
net = ld.net;
```

The DAG network contains 58 layers including convolution, max pooling, depth concatenation, and the pixel classification output layers. To display an interactive visualization of the deep learning network architecture, use the `analyzeNetwork` (Deep Learning Toolbox) function.

```
analyzeNetwork(net);
```

### Prepare Data

Download the Hamlin Beach State Park data.

```
if ~exist(fullfile(pwd, 'data'))
    url = 'http://www.cis.rit.edu/~rmk6217/rit18_data.mat';
    downloadHamlinBeachMSIData(url,pwd+"/data/");
end
```

Downloading Hamlin Beach dataset...  
This will take several minutes to download...  
done.

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 fourth, fifth, and sixth image channels. The first 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 `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 `[12446,7654,7]` and a patch size of `[1024,1024]`. 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.m` entry-point function.

```
cfg = coder.gpuConfig('mex');
cfg.TargetLang = 'C++';
cfg.DeepLearningConfig = coder.DeepLearningConfig('cudnn');
codegen -config cfg segmentImageUnet -args {ones(size(test_data),'uint16'),coder.Constant([1024 1024])}
```

Code generation successful: To view the report, open('codegen/mex/segmentImageUnet/html/report.m')

## 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` (`12446x7654x7`), it is easier to process such a large image in patches.

```
segmentedImage = segmentImageUnet_mex(test_data,[1024 1024]);
```

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.

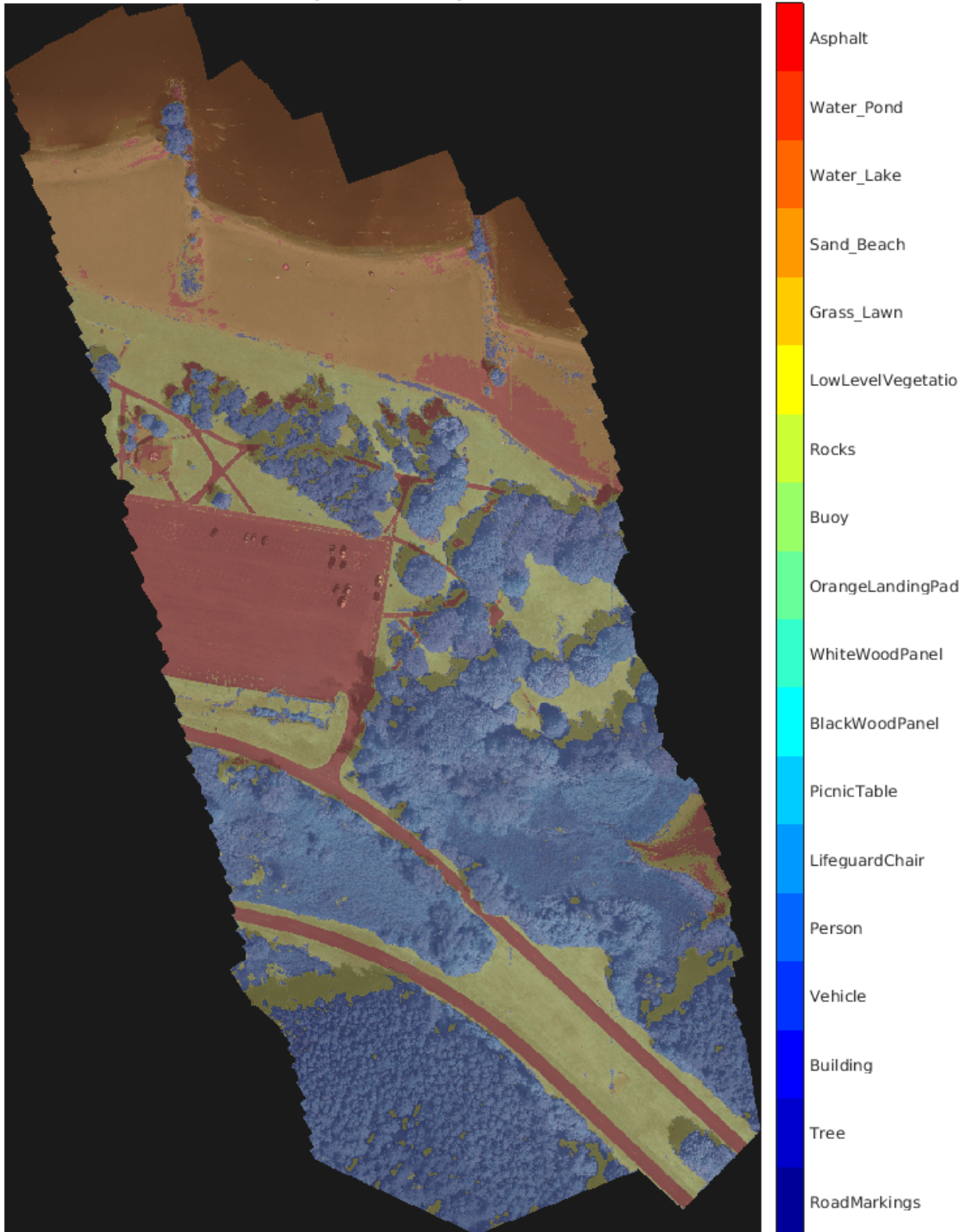
```
classNames = [ "RoadMarkings", "Tree", "Building", "Vehicle", "Person", ...  
              "LifeguardChair", "PicnicTable", "BlackWoodPanel", ...  
              "WhiteWoodPanel", "OrangeLandingPad", "Buoy", "Rocks", ...  
              "LowLevelVegetation", "Grass_Lawn", "Sand_Beach", ...  
              "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(:,:,4:6),[0 0.6],[0.1 0.9],0.55),segmentedImage,'Transparency',0.5);  
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

`codegen` | `coder.DeepLearningConfig` | `coder.checkGpuInstall` |  
`coder.loadDeepLearningNetwork`

### Objects

`coder.CuDNNConfig` | `coder.gpuConfig` | `coder.gpuEnvConfig`

## 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 5-142
- "Code Generation for Semantic Segmentation Network That Uses U-net" on page 5-154

## 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 2018-2019 The MathWorks, Inc.

persistent mynet;

if isempty(mynet)
    mynet = coder.loadDeepLearningNetwork('getDenoisingNetwork', 'DnCNN');
end

% The activations methods extracts the output from the last layer. The
% 'OutputAs' 'channels' name-value pair argument is used inorder 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: To view the report, open('codegen/mex/denoisenet\_predict/html/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

`codegen` | `coder.DeepLearningConfig` | `coder.checkGpuInstall` |  
`coder.loadDeepLearningNetwork`

#### Objects

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

### More About

- "Generated CNN Class Hierarchy" on page 5-44
- "Supported Networks, Layers, and Classes" on page 5-5
- "Load Pretrained Networks for Code Generation" on page 5-45
- "Code Generation for Deep Learning Networks by Using cuDNN" on page 5-48
- "Code Generation for Deep Learning Networks by Using TensorRT" on page 5-57

## 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

```
net = getYOLOv2();
```

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)

% Copyright 2018-2019 The MathWorks, Inc.

persistent yolov2obj;

if isempty(yolov2obj)
    yolov2obj = coder.loadDeepLearningNetwork('yolov2ResNet50VehicleExample.mat');
end

% pass in input
[bboxes,~,labels] = yolov2obj.detect(in,'Threshold',0.5);

% convert categorical labels to cell array of character vectors for MATLAB
% execution
if coder.target('MATLAB')
    labels = cellstr(labels);
end

% Annotate detections in the image.
outImg = insertObjectAnnotation(in,'rectangle',bboxes,labels);

```

### Run MEX Code Generation

To generate CUDA code for the `yolov2_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 `[224,224,3]`. This value corresponds to the input layer size of YOLOv2.

```

cfg = coder.gpuConfig('mex');
cfg.TargetLang = 'C++';
cfg.DeepLearningConfig = coder.DeepLearningConfig('cudnn');
codegen -config cfg yolov2_detect -args {ones(224,224,3,'uint8')} -report

```

Code generation successful: To view the report, open('codegen/mex/yolov2\_detect/html/report.mldat

### 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.

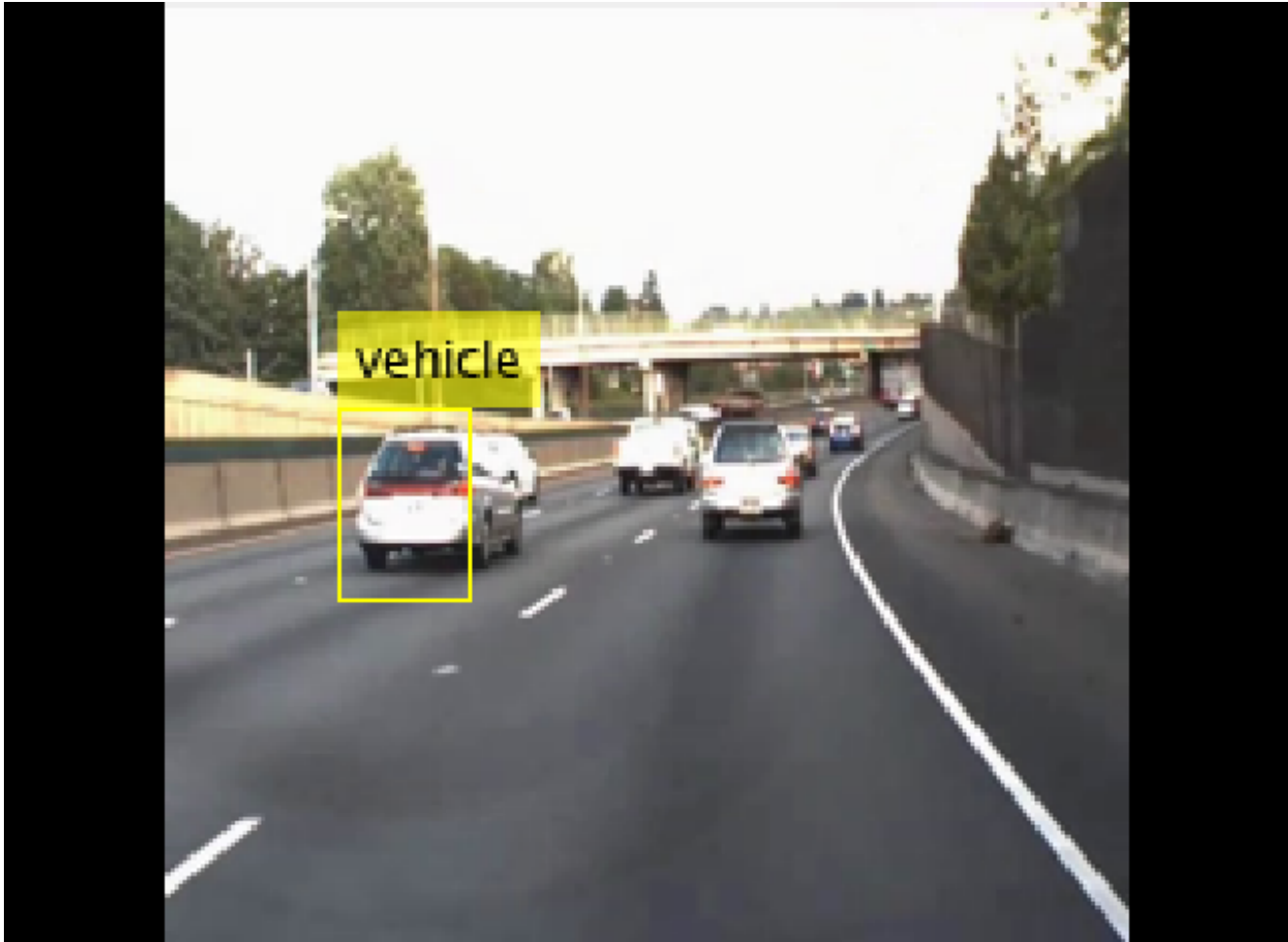
```

cont = ~isDone(videoFreader);
while cont
    I = step(videoFreader);
    in = imresize(I,[224,224]);

```



```
out = yolov2_detect_mex(in);  
step(depVideoPlayer, out);  
cont = ~isDone(videoFreader) && isOpen(depVideoPlayer); % Exit the loop if the video player  
end
```



## References

[1] Redmon, Joseph, and Ali Farhadi. "YOLO9000: Better, Faster, Stronger." 2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR). IEEE, 2017.

## See Also

### Functions

codegen | coder.DeepLearningConfig | coder.checkGpuInstall |  
coder.loadDeepLearningNetwork

### Objects

coder.CuDNNConfig | coder.gpuConfig | coder.gpuEnvConfig |  
vision.DeployableVideoPlayer | vision.VideoFileReader

## **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 5-178
- “Code Generation For Object Detection Using YOLO v3 Deep Learning” on page 5-202

## **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 5-5
- “Load Pretrained Networks for Code Generation” on page 5-45

## 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 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: To view the report, open('codegen/mex/lstmnet\_predict/html/report.ml

### 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))

```

Columns 1 through 6

```

Sitting      Sitting      Sitting      Sitting      Sitting      Sitting

```

Columns 7 through 10

```

Sitting      Sitting      Sitting      Sitting

```

### Compare Predictions with Test Data

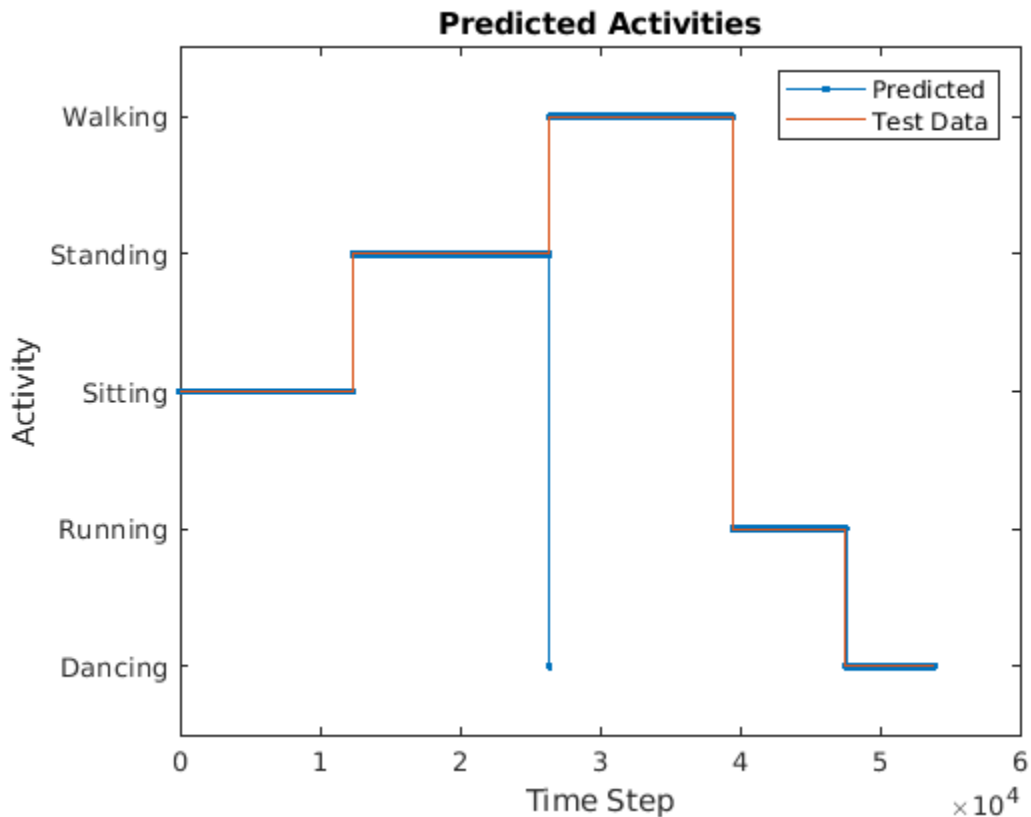
Use a plot to compare the MEX output data with the test data.

```

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))
```

Columns 1 through 6

```
Sitting    Sitting    Sitting    Sitting    Sitting    Sitting
```

Columns 7 through 10

```
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
```

```
YPred3 = lstmnet_predict_mex(XValidate);
```

Code generation successful: To view the report, open('codegen/mex/lstmnet\_predict/html/report.mlx')

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 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: To view the report, open('codegen/mex/lstmnet\_predict\_and\_update/html')

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

`codegen` | `coder.DeepLearningConfig` | `coder.checkGpuInstall` |  
`coder.loadDeepLearningNetwork`

### Objects

`coder.CuDNNConfig` | `coder.TensorRTConfig` | `coder.gpuConfig` | `coder.gpuEnvConfig`

## Related Examples

- “Sequence Classification Using Deep Learning” (Deep Learning Toolbox)
- “Code Generation for a Video Classification Network” on page 5-196

## More About

- “Long Short-Term Memory Networks” (Deep Learning Toolbox)
- “Supported Networks, Layers, and Classes” on page 5-5
- “Load Pretrained Networks for Code Generation” on page 5-45



## 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;
```

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_
```

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
```



*Copyright 2019 The MathWorks, Inc.*

## **See Also**

### **Functions**

`cnncodegen`

## **See Also**

### **More About**

- “Supported Networks, Layers, and Classes” on page 5-5
- “Code Generation for Deep Learning Networks Targeting ARM Mali GPUs” on page 5-66

## 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 DAGNetwork

```
net = getSSDNW();
```

Downloading pretrained detector (44 MB)...

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.

```
analyzeNetwork(net);
```

### 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)

% Copyright 2019-2020 The MathWorks, Inc.

persistent ssdObj;

if isempty(ssdObj)
    ssdObj = coder.loadDeepLearningNetwork('ssdResNet50VehicleExample_20a.mat');
end

% Pass in input
[bboxes,~,labels] = ssdObj.detect(in,'Threshold',0.7);

% 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 [300,300,3]. This value corresponds to the input layer size of SSD Network.

```
cfg = coder.gpuConfig('mex');
cfg.TargetLang = 'C++';
cfg.DeepLearningConfig = coder.DeepLearningConfig('cudnn');
codegen -config cfg ssdObj_detect -args {ones(300,300,3,'uint8')} -report
```

Code generation successful: To view the report, open('codegen/mex/ssdObj\_detect/html/report.mldat

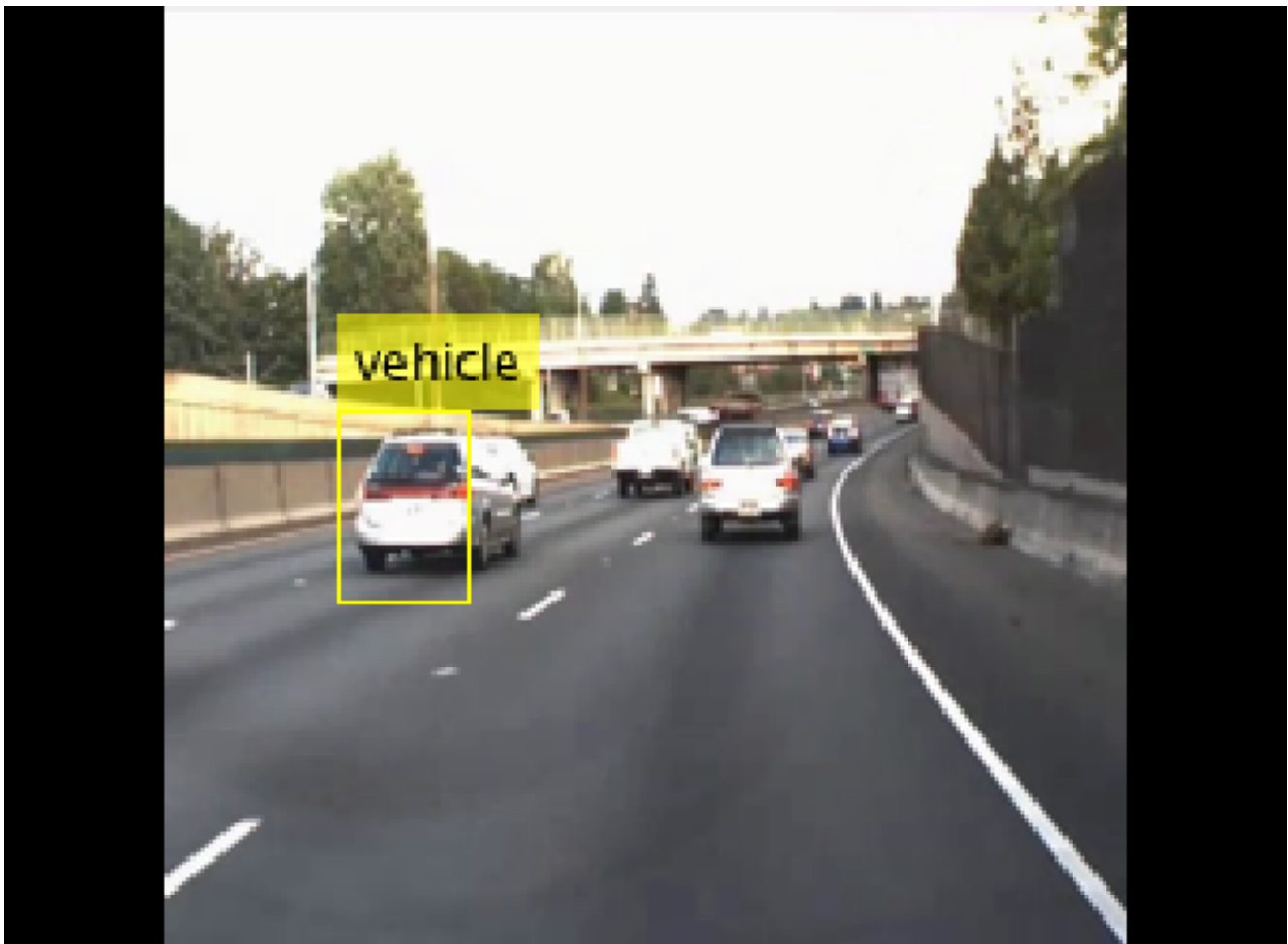
### Run Generated MEX

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, [300, 300]);  
    out = ssdObj_detect_mex(in);  
    step(depVideoPlayer, out);  
    cont = ~isDone(videoFreader) && isOpen(depVideoPlayer); % Exit the loop if the video player  
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

`codegen` | `coder.DeepLearningConfig` | `coder.checkGpuInstall` |  
`coder.loadDeepLearningNetwork`

### Objects

`coder.CuDNNConfig` | `coder.gpuConfig` | `coder.gpuEnvConfig` |  
`vision.DeployableVideoPlayer` | `vision.VideoFileReader`

## Related Examples

- "Object Detection Using SSD Deep Learning" (Computer Vision Toolbox)
- "Code Generation for Object Detection by Using YOLO v2" on page 5-165
- "Code Generation For Object Detection Using YOLO v3 Deep Learning" on page 5-202

## 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).

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:

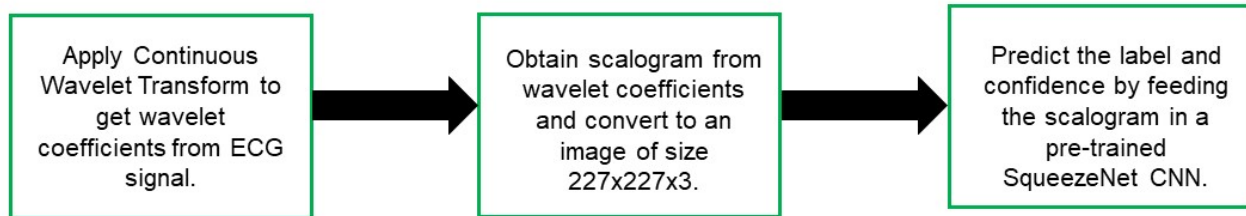
- 1 Persons with cardiac arrhythmia (ARR)
- 2 Persons with congestive heart failure (CHF)
- 3 Persons with normal sinus rhythms (NSR)

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

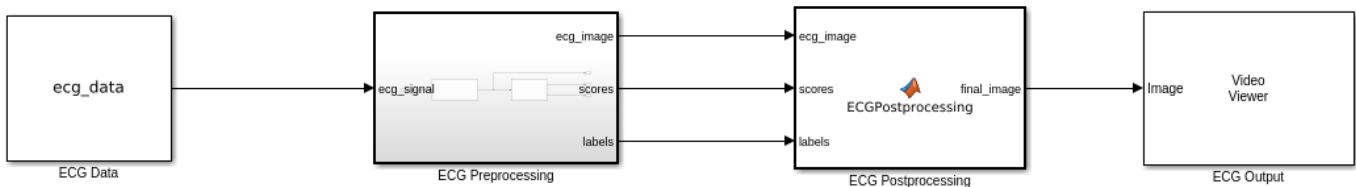
The block diagram for the algorithmic workflow of the Simulink model is shown.



## 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');
```

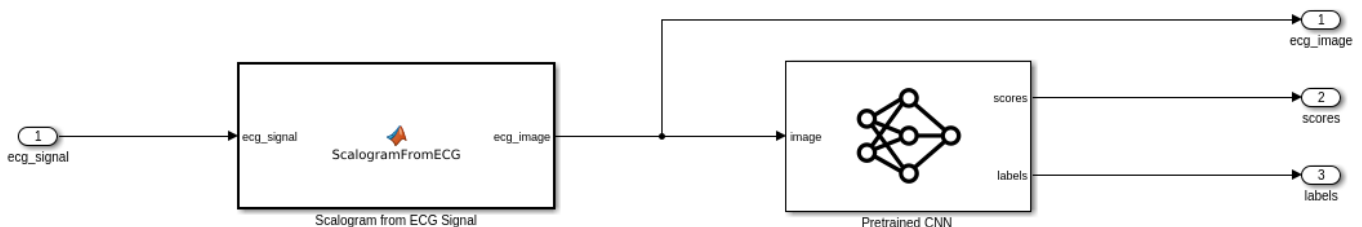


Copyright 2020 The MathWorks, Inc.

## 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 = colourmap(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 The MathWorks, Inc.

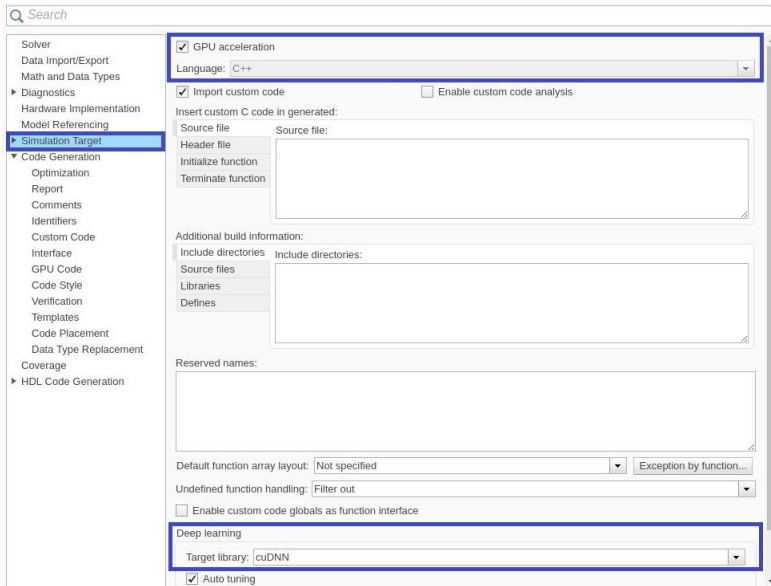
scores = double(scores);
% Obtain maximum confidence
[prob,index] = max(scores);
confidence = prob*100;
% Obtain label corresponding to maximum confidence
label = erase(char(labels(index)),'_label');
text = cell(2,1);
text{1} = ['Classification: ' label];
text{2} = ['Confidence: ' sprintf('%0.2f',confidence) '%'];
position = [135 20 0 0; 130 40 0 0];
final_image = insertObjectAnnotation(ecg_image,'rectangle',position,text,'TextBoxOpacity',0.9,'F

end
```

### Run the Simulation

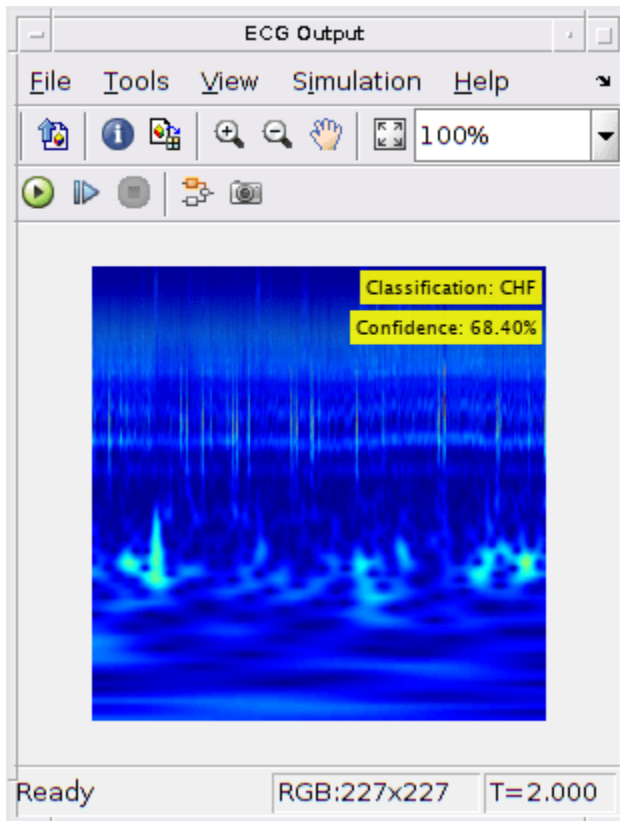
Open Configuration Parameters dialog box.

In **Simulation Target** pane, select **GPU acceleration**. In the **Deep Learning** group, select the target library as **cuDNN**.



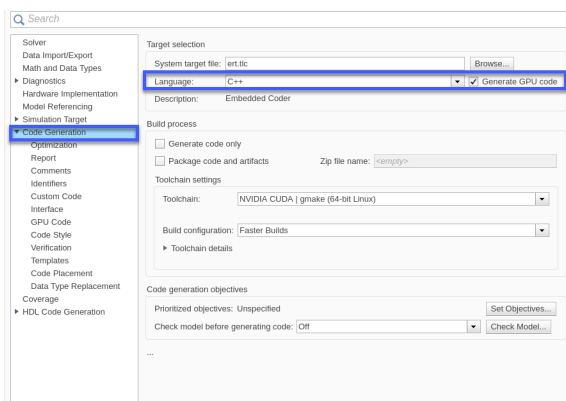
To verify the algorithm and display the labels and confidence score of the test ECG signal loaded in the workspace, run the simulation.

```
set_param('ecg_dl_cwt', 'SimulationMode', 'Normal');
sim('ecg_dl_cwt');
```

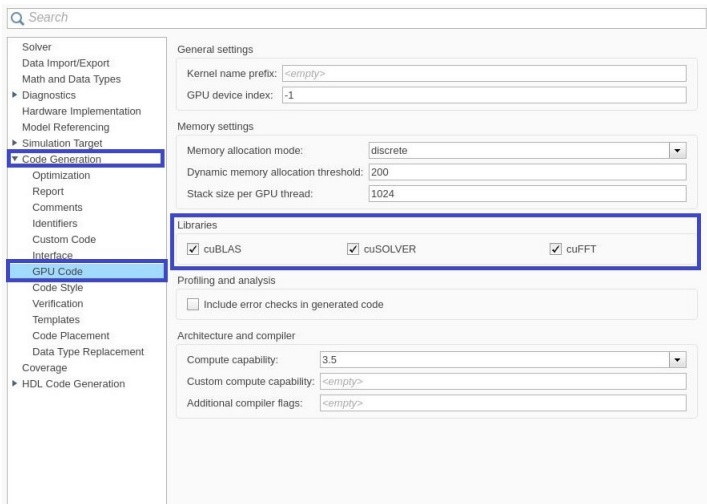


### Generate and Build the Simulink Model

In **Code Generation** pane, select the **Language** as **C++** and enable **Generate GPU code**.

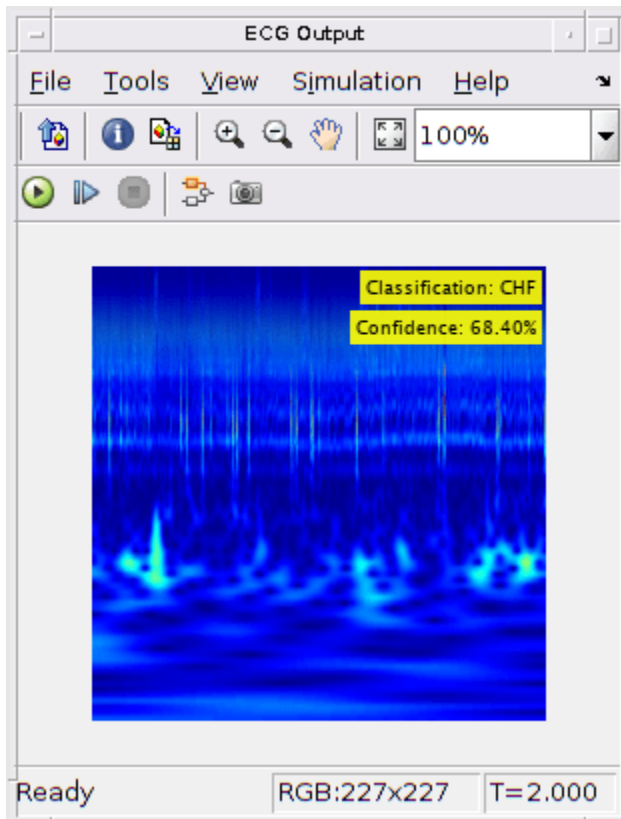


Open **Code Generation > GPU Code** pane. In the subcategory **Libraries**, enable **cuBLAS**, **cuSOLVER** and **cuFFT**.



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');
```

## 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';

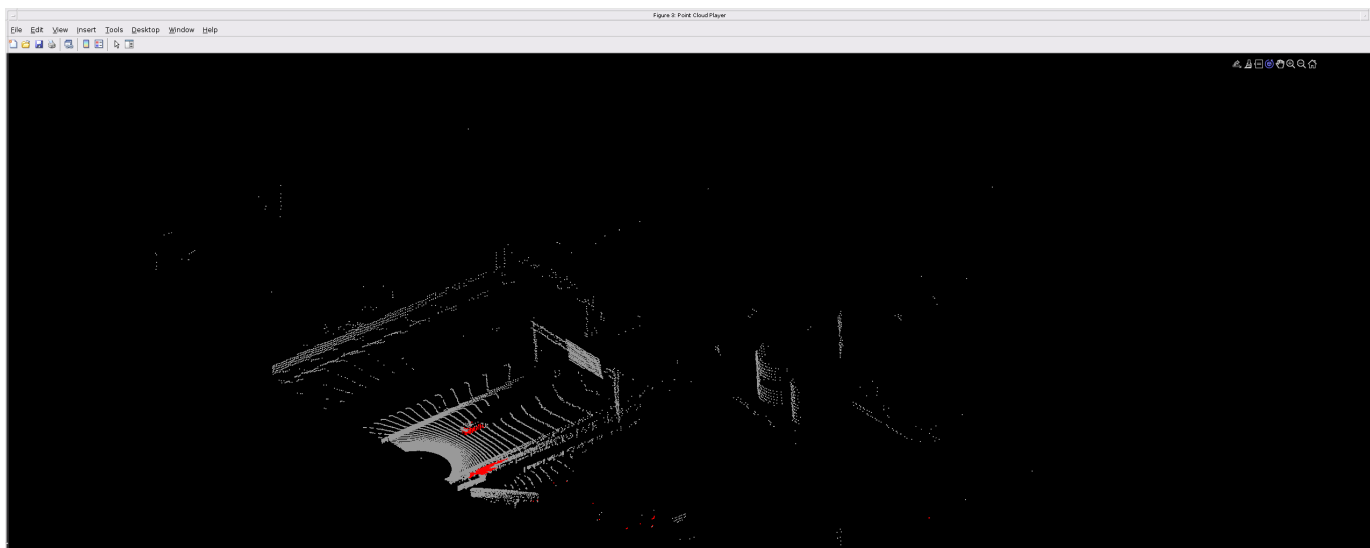
    % Extract a point cloud for each class.
    carPointCloud = select(ptCloud, carIndices, 'OutputSize', 'full');
    truckPointCloud = select(ptCloud, truckIndices, 'OutputSize', 'full');
    backgroundPointCloud = select(ptCloud, backgroundIndices, 'OutputSize', 'full');

    % 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

`codegen` | `coder.DeepLearningConfig` | `coder.checkGpuInstall` |  
`coder.loadDeepLearningNetwork`

### Objects

`coder.CuDNNConfig` | `coder.TensorRTConfig` | `coder.gpuConfig` | `coder.gpuEnvConfig`

## 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 (TM). 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 AGX Xavier, NVIDIA Jetson Xavier NX
CUDA Version    : 10.2
cuDNN Version   : 8.0
TensorRT Version : 7.1
GStreamer Version : 1.14.5
V4L2 Version    : 1.14.2-1
SDL Version     : 1.2
OpenCV Version  : 4.1.1
Available Webcams :
Available GPUs  : Xavier

```

**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.

```
hwobj.putFile(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.

```
hwobj.runApplication('net_classify');
```

```
### Launching the executable on the target...  
Executable launched successfully with process ID 4394.  
Displaying the simple runtime log for the executable...
```

Note: For the complete log, run the following command in the MATLAB command window:  
`system(hwobj,'cat /home/ubuntu/VideoClassify/MATLAB_ws/R2021a/home/lnarasim/Documents/MATLAB/Exar`

## See Also

### Functions

[VideoReader](#) | [codegen](#) | [coder.DeepLearningConfig](#) | [coder.checkGpuInstall](#) | [coder.loadDeepLearningNetwork](#) | [jetson](#) | [killApplication](#) | [runApplication](#)

### Objects

[coder.CuDNNConfig](#) | [coder.TensorRTConfig](#) | [coder.gpuConfig](#) | [coder.gpuEnvConfig](#) | [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 with custom layers. 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. This example uses the YOLO v3 network trained in the *Object Detection Using YOLO v3 Deep Learning* example from 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

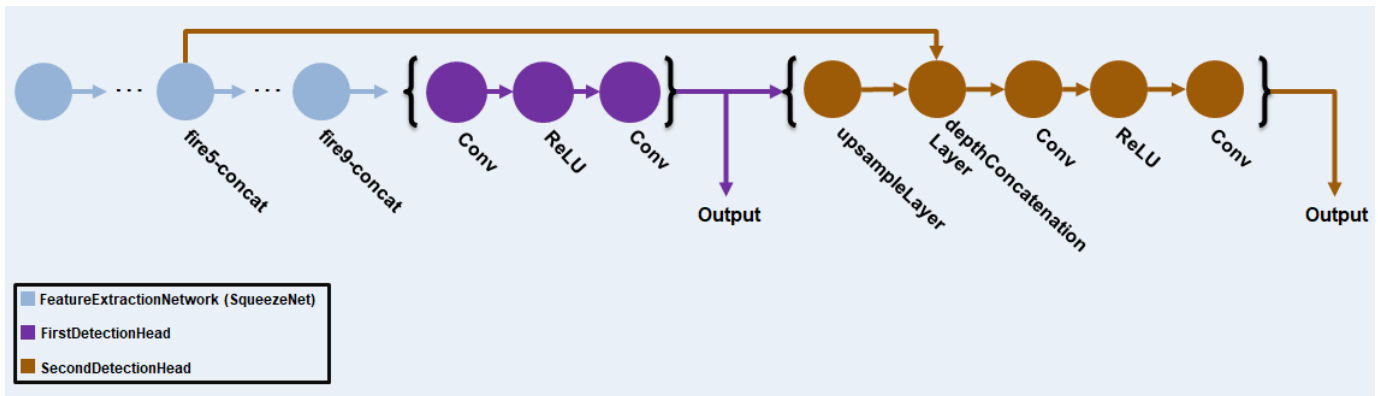
To verify that the compilers and libraries for running this example are set up correctly, use the `coder.checkGpuInstall` function.

```
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

The YOLO v3 network used in this example was trained using the steps described in “Object Detection Using YOLO v3 Deep Learning” (Computer Vision Toolbox).

```
matFile = 'yolov3SqueezeNetVehicleExample.mat';
pretrained = load(matFile);
net = pretrained.net;
```

YOLO v3 network uses a `resize2dLayer` to resize the 2-D input image by replicating the neighboring pixel values by a scaling factor of 2. The `resize2dLayer` is implemented as a custom layer supported for code generation.

### The `yolov3Detect` Entry-Point Function

The `yolov3Detect` entry-point function takes an input image and passes it to a trained network for prediction through the `yolov3Predict` function. The `yolov3Predict` function loads the network object from the MAT-file into a persistent variable and reuses the persistent object for subsequent prediction calls. Specifically, the function uses the `dlnetwork` representation of the network trained in the “Object Detection Using YOLO v3 Deep Learning” (Computer Vision Toolbox) example. The predictions from the YOLO v3 grid cell coordinates obtained from the `yolov3Predict` calls are then converted to bounding box coordinates by using the supporting functions `generateTiledAnchors` and `applyAnchorBoxOffsets`.

```
type('yolov3Detect.m')
```

```
function [bboxes,scores,labelsIndex] = yolov3Detect(matFile, im, networkInputSize, networkOutputSize)
% The yolov3Detect function detects the bounding boxes, scores, and labelsIndex in an image.
%#codegen
```

```
%% Preprocess Data
```

```
% This example applies all the preprocessing transforms to the data set
% applied during training, except data augmentation. Because the example
% uses a pretrained YOLO v3 network, the input data must be representative
% of the original data and left unmodified for unbiased evaluation.
```

```

% Specifically the following preprocessing operations are applied to the
% input data.
%     1. Resize the images to the network input size, as the images are bigger than networkInputSize.
%     2. Scale the image pixels in the range [0 1].
%     3. Convert the resized and rescaled image to a darray object.

im = darray(preprocessData(im, networkInputSize), "SSCB");
imageSize = size(im,[1,2]);

%% Define Anchor Boxes
% Specify the anchor boxes estimated on the basis of the preprocessed
% training data used when training the YOLO v3 network. These anchor box
% values are same as mentioned in
% <docid:vision_ug#mw_47d9a223-5ec7-4d36-a020-4f9d147ecdec Object Detection
% Using YOLO v3 Deep Learning> example. For details on estimating anchor
% boxes, see <docid:vision_ug#mw_f9f22f48-0ad0-4f37-8bc1-22a2046637f2
% Anchor Boxes for Object Detection>.

anchors = [
    41    34;
    163   130;
    98    93;
    144   125;
    33    24;
    69    66];

% Specify anchorBoxMasks to select anchor boxes to use in both the
% detection heads of the YOLO v3 network. anchorBoxMasks is a cell array of
% size M-by-1, where M denotes the number of detection heads. Each
% detection head consists of a 1-by-N array of row index of anchors in
% anchorBoxes, where N is the number of anchor boxes to use. Select anchor
% boxes for each detection head based on size-use larger anchor boxes at
% lower scale and smaller anchor boxes at higher scale. To do so, sort the
% anchor boxes with the larger anchor boxes first and assign the first
% three to the first detection head and the next three to the second
% detection head.

area = anchors(:, 1).*anchors(:, 2);
[~, idx] = sort(area, 'descend');
anchors = anchors(idx, :);
anchorBoxMasks = {[1,2,3]
    [4,5,6]
    };

%% Predict on Yolov3
% Predict and filter the detections based on confidence threshold.
predictions = yolov3Predict(matFile,im,networkOutputs,anchorBoxMasks);

%% Generate Detections
anchorIndex = 2:5; % indices corresponding to x,y,w,h predictions for bounding boxes
tiledAnchors = generateTiledAnchors(predictions,anchors,anchorBoxMasks,anchorIndex);
predictions = applyAnchorBoxOffsets(tiledAnchors, predictions, networkInputSize, anchorIndex);
[bboxes,scores,labelsIndex] = generateYOLOv3DetectionsForCodegen(predictions, confidenceThreshold);

end

function YPredCell = yolov3Predict(matFile,im,networkOutputs,anchorBoxMask)
% Predict the output of network and extract the confidence, x, y,

```

```

% width, height, and class.

% load the deep learning network for prediction
persistent net;

if isempty(net)
    net = coder.loadDeepLearningNetwork(matFile);
end

YPredictions = cell(coder.const(networkOutputs), 1);
[YPredictions{:}] = predict(net, im);
YPredCell = extractPredictions(YPredictions, anchorBoxMask);

% Apply activation to the predicted cell array.
YPredCell = applyActivations(YPredCell);
end

```

### Evaluate the Entry-Point Function for Object Detection

Follow these steps to evaluate the entry-point function on an image from the test data.

- Specify the confidence threshold as 0.5 to keep only detections with confidence scores above this value.
- Specify the overlap threshold as 0.5 to remove overlapping detections.
- Read an image from the input data.
- Use the entry-point function `yolov3Detect` to get the predicted bounding boxes, confidence scores, and class labels.
- Display the image with bounding boxes and confidence scores.

Define the desired thresholds.

```

confidenceThreshold = 0.5;
overlapThreshold = 0.5;

```

Specify the network input size of the trained network and the number of network outputs.

```

networkInputSize = [227 227 3];
networkOutputs = numel(net.OutputNames);

```

Read the example image data obtained from the labeled data set from the “Object Detection Using YOLO v3 Deep Learning” (Computer Vision Toolbox) example. This image contains one instance of an object of type vehicle.

```

I = imread('vehicleImage.jpg');

```

Specify the class names.

```

classNames = {'vehicle'};

```

Invoke the detect method on YOLO v3 network and display the results.

```

[bboxes,scores,labelsIndex] = yolov3Detect(matFile, I, networkInputSize, networkOutputs, confidenceThreshold, overlapThreshold);
labels = classNames{labelsIndex};

```

```

% Display the detections on the image
IAnnotated = insertObjectAnnotation(I, 'rectangle', bboxes, [labels ' - ' num2str(scores)]);

```

```
figure
imshow(IAnnotated)
```



### Generate CUDA MEX

To generate CUDA® code for the `yolov3Detect` 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');

args = {coder.Constant(matFile), I, coder.Constant(networkInputSize), coder.Constant(networkOutputs),
        confidenceThreshold, overlapThreshold, classNames};

codegen -config cfg yolov3Detect -args args -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

Call the generated CUDA MEX with the same image input `I` as before and display the results.

```
[bboxes, scores, labelsIndex] = yolov3Detect_mex(matFile, I, networkInputSize, networkOutputs,
        confidenceThreshold, overlapThreshold, classNames);
labels = classNames{labelsIndex};

figure;
```



```
IAnnotated = insertObjectAnnotation(I, 'rectangle', bboxes, [labels ' - ' num2str(scores)]);
imshow(IAnnotated);
```



## Utility Functions

The utility functions listed below are based on the ones used in “Object Detection Using YOLO v3 Deep Learning” (Computer Vision Toolbox) example and modified to make the utility functions suitable for code generation.

```
type('applyActivations.m')
```

```
function YPredCell = applyActivations(YPredCell)
%#codegen

numCells = size(YPredCell, 1);
for iCell = 1:numCells
    for idx = 1:3
        YPredCell{iCell, idx} = sigmoidActivation(YPredCell{iCell, idx});
    end
end
for iCell = 1:numCells
    for idx = 4:5
        YPredCell{iCell, idx} = exp(YPredCell{iCell, idx});
    end
end
for iCell = 1:numCells
    YPredCell{iCell, 6} = sigmoidActivation(YPredCell{iCell, 6});
end
end
```

```
function out = sigmoidActivation(x)
out = 1./(1+exp(-x));
end
```

```
type('extractPredictions.m')
```

```

function predictions = extractPredictions(YPredictions, anchorBoxMask)
%#codegen

numPredictionHeads = size(YPredictions, 1);
predictions = cell(numPredictionHeads,6);
for ii = 1:numPredictionHeads
    % Get the required info on feature size.
    numChannelsPred = size(YPredictions{ii},3);
    numAnchors = size(anchorBoxMask{ii},2);
    numPredElemsPerAnchors = numChannelsPred/numAnchors;
    allIds = (1:numChannelsPred);

    stride = numPredElemsPerAnchors;
    endIdx = numChannelsPred;

    YPredictionsData = extractdata(YPredictions{ii});

    % X positions.
    startIdx = 1;
    predictions{ii,2} = YPredictionsData(:, :, startIdx:stride:endIdx, :);
    xIds = startIdx:stride:endIdx;

    % Y positions.
    startIdx = 2;
    predictions{ii,3} = YPredictionsData(:, :, startIdx:stride:endIdx, :);
    yIds = startIdx:stride:endIdx;

    % Width.
    startIdx = 3;
    predictions{ii,4} = YPredictionsData(:, :, startIdx:stride:endIdx, :);
    wIds = startIdx:stride:endIdx;

    % Height.
    startIdx = 4;
    predictions{ii,5} = YPredictionsData(:, :, startIdx:stride:endIdx, :);
    hIds = startIdx:stride:endIdx;

    % Confidence scores.
    startIdx = 5;
    predictions{ii,1} = YPredictionsData(:, :, startIdx:stride:endIdx, :);
    confIds = startIdx:stride:endIdx;

    % Accumulate all the non-class indexes
    nonClassIds = [xIds yIds wIds hIds confIds];

    % Class probabilities.
    % Get the indexes which do not belong to the nonClassIds
    classIdx = setdiff(allIds, nonClassIds, 'stable');
    predictions{ii,6} = YPredictionsData(:, :, classIdx, :);
end
end

type('generateTiledAnchors.m')

function tiledAnchors = generateTiledAnchors(YPredCell, anchorBoxes, anchorBoxMask, anchorIndex)
% Generate tiled anchor offset for converting the predictions from the YOLO
% v3 grid cell coordinates to bounding box coordinates
%#codegen

```

```

numPredictionHeads = size(YPredCell,1);
tiledAnchors = cell(numPredictionHeads, size(anchorIndex, 2));
for i=1:numPredictionHeads
    anchors = anchorBoxes(anchorBoxMask{i}, :);
    [h,w,~,n] = size(YPredCell{i,1});
    [tiledAnchors{i,2}, tiledAnchors{i,1}] = ndgrid(0:h-1,0:w-1,1:size(anchors,1),1:n);
    [~,~,tiledAnchors{i,3}] = ndgrid(0:h-1,0:w-1,anchors(:,2),1:n);
    [~,~,tiledAnchors{i,4}] = ndgrid(0:h-1,0:w-1,anchors(:,1),1:n);
end
end

type('applyAnchorBoxOffsets.m')

function YPredCell = applyAnchorBoxOffsets(tiledAnchors,YPredCell,inputImageSize,anchorIndex)
% Convert the predictions from the YOLO v3 grid cell coordinates to bounding box coordinates
%#codegen

for i=1:size(YPredCell,1)
    [h,w,~,~] = size(YPredCell{i,1});
    YPredCell{i,anchorIndex(1)} = (tiledAnchors{i,1}+YPredCell{i,anchorIndex(1)}) ./w;
    YPredCell{i,anchorIndex(2)} = (tiledAnchors{i,2}+YPredCell{i,anchorIndex(2)}) ./h;
    YPredCell{i,anchorIndex(3)} = (tiledAnchors{i,3}.*YPredCell{i,anchorIndex(3)}) ./inputImageSi;
    YPredCell{i,anchorIndex(4)} = (tiledAnchors{i,4}.*YPredCell{i,anchorIndex(4)}) ./inputImageSi;
end
end

type('preprocessData.m')

function image = preprocessData(image, targetSize)
% Resize the images and scale the pixels to between 0 and 1.
%#codegen

imgSize = size(image);

% Convert an input image with single channel to 3 channels.
if numel(imgSize) < 1
    image = repmat(image,1,1,3);
end

image = im2single(imresize(image, coder.const(targetSize(1:2))));

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

codegen | coder.DeepLearningConfig | coder.checkGpuInstall |  
 coder.loadDeepLearningNetwork

### Objects

coder.CuDNNConfig | coder.TensorRTConfig | coder.gpuConfig | coder.gpuEnvConfig |  
 dlarray | dlnetwork

## **Related Examples**

- “Object Detection Using YOLO v3 Deep Learning” (Computer Vision Toolbox)

## **More About**

- “Code Generation for dlarray” on page 5-35
- “dlarray Limitations for Code Generation” on page 5-41

# 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.

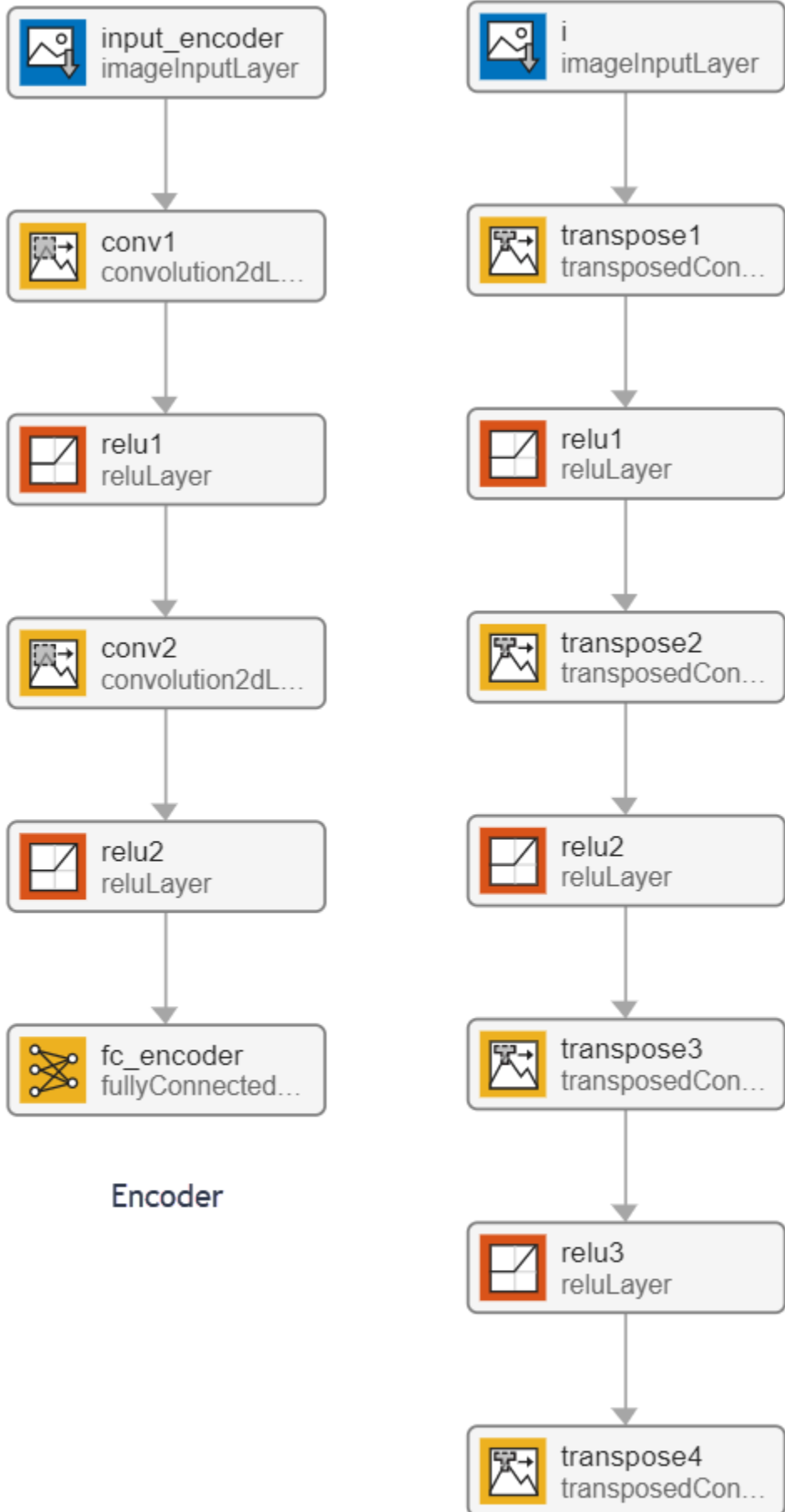
```
envCfg = coder.gpuEnvConfig('host');
envCfg.DeepLibTarget = 'cudnn';
envCfg.DeepCodegen = 1;
envCfg.Quiet = 1;
coder.checkGpuInstall(envCfg);
```

## 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](#)

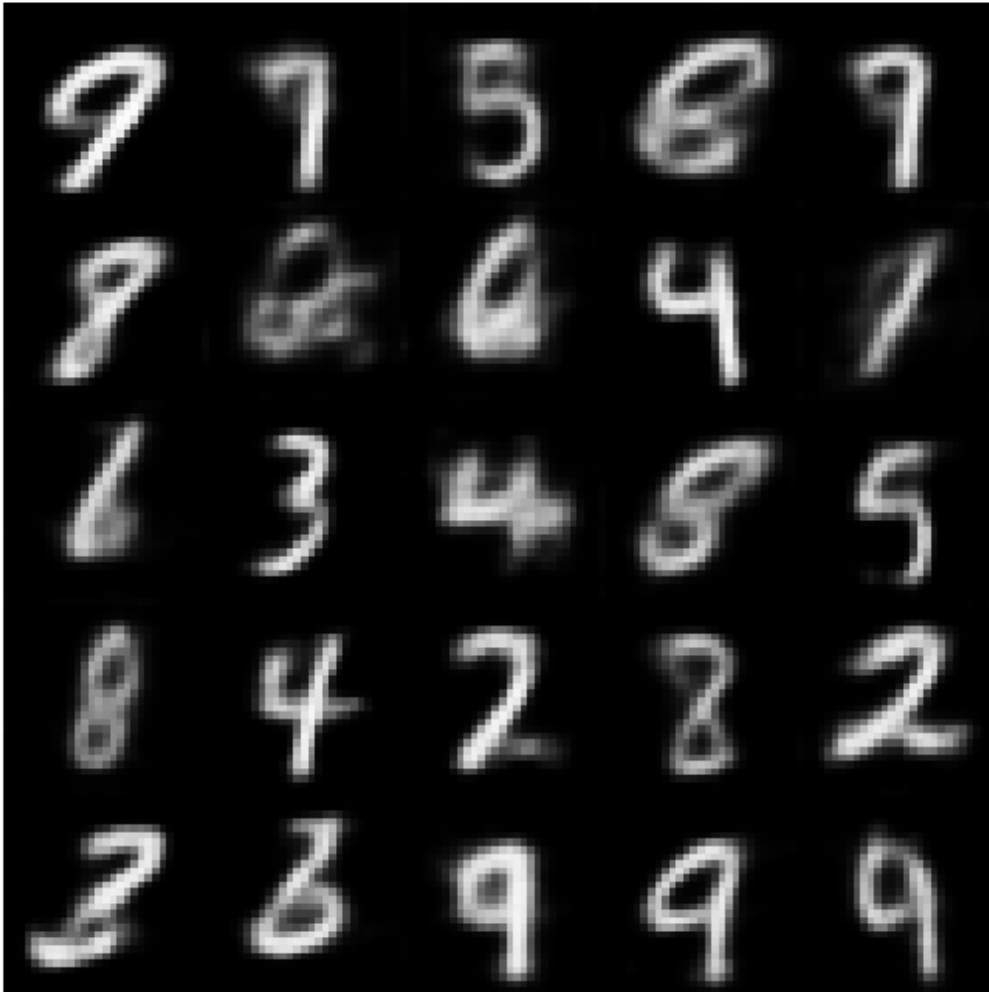
To generate CUDA code for TensorRT target, create and use a TensorRT deep learning configuration object instead of the CuDNN configuration object.

### **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]))
```



## See Also

### Functions

`codegen` | `coder.DeepLearningConfig` | `coder.checkGpuInstall` |  
`coder.loadDeepLearningNetwork`

### Objects

`coder.CuDNNConfig` | `coder.TensorRTConfig` | `coder.gpuConfig` | `coder.gpuEnvConfig` |  
`dlarray` | `dlnetwork`

## Related Examples

- “Train Variational Autoencoder (VAE) to Generate Images” (Deep Learning Toolbox)

## **More About**

- “Code Generation for dlarray” on page 5-35
- “dlarray Limitations for Code Generation” on page 5-41
- “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)

# Targeting Embedded GPU Devices

---

- “Build and Run an Executable on NVIDIA Hardware” on page 6-2
- “Build and Run an Executable on NVIDIA Hardware Using GPU Coder App” on page 6-7
- “Relocate Generated Code to Another Development Environment” on page 6-14
- “Getting Started with the MATLAB Coder Support Package for NVIDIA Jetson and NVIDIA DRIVE Platforms” on page 6-24
- “Sobel Edge Detection on NVIDIA Jetson Nano Using Raspberry Pi Camera Module V2” on page 6-29
- “Semantic Segmentation on NVIDIA DRIVE” on page 6-34
- “Top-Hat Filtering to Remove Uneven Background Illumination on NVIDIA Jetson TX2 Developer Kit” on page 6-39
- “Deployment and Classification of Webcam Images on NVIDIA Jetson TX2 Platform” on page 6-44

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 6-2
- “Tutorial Prerequisites” on page 6-2
- “Example: Vector Addition” on page 6-3
- “Create a Live Hardware Connection Object” on page 6-3
- “Generate CUDA Executable Using GPU Coder” on page 6-4
- “Run the Executable and Verify the Results” on page 6-5

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 6-24 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

`drive` | `jetson`

## More About

- “Build and Run an Executable on NVIDIA Hardware Using GPU Coder App” on page 6-7
- “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 5-48
- “Code Generation for Deep Learning Networks by Using TensorRT” on page 5-57
- “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 6-7

“Tutorial Prerequisites” on page 6-7

“Example: Vector Addition” on page 6-8

“Custom Main File” on page 6-8

“GPU Coder App” on page 6-9

“Run the Executable and Verify the Results” on page 6-12

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 6-24 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[]);

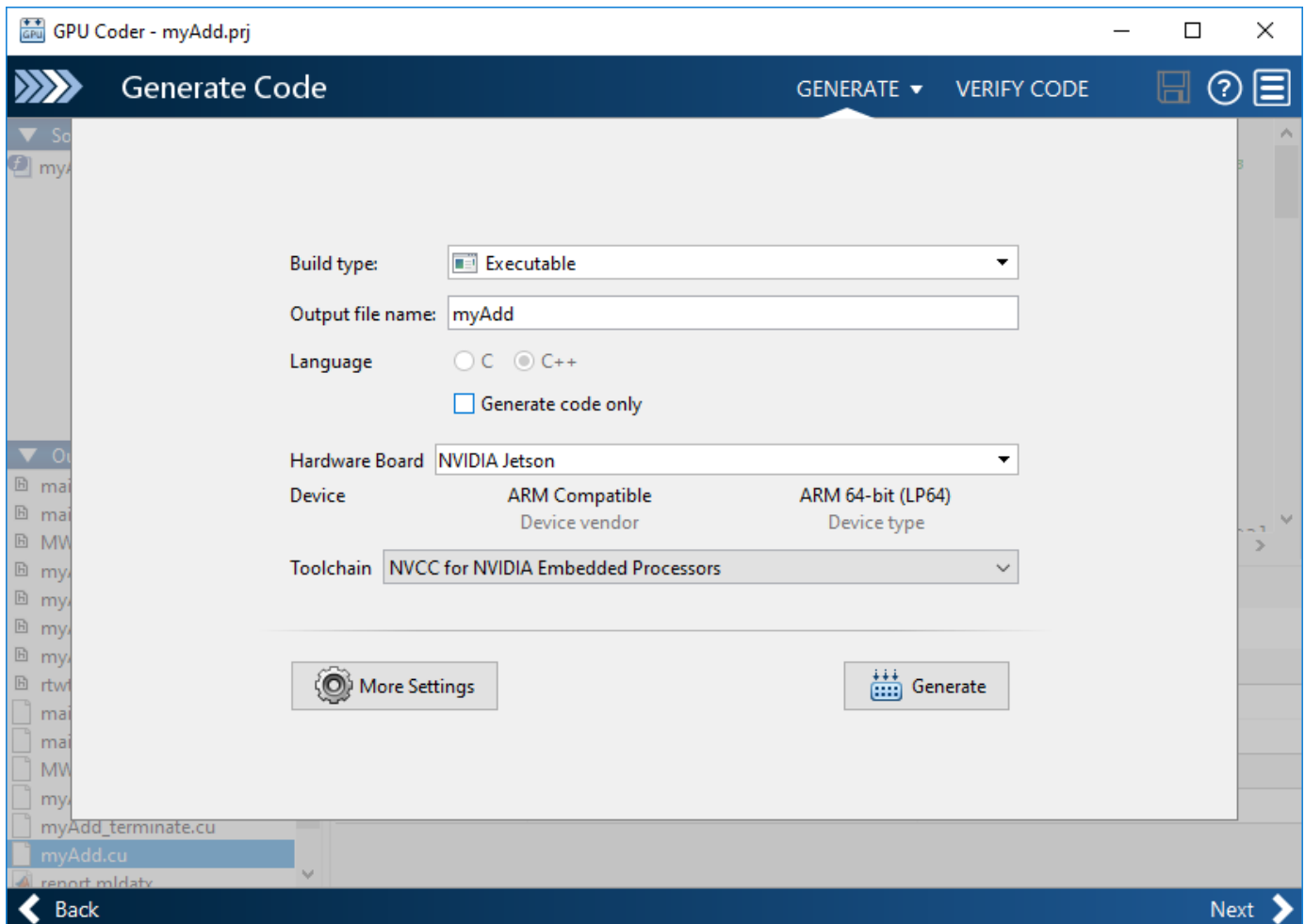
#endif

```

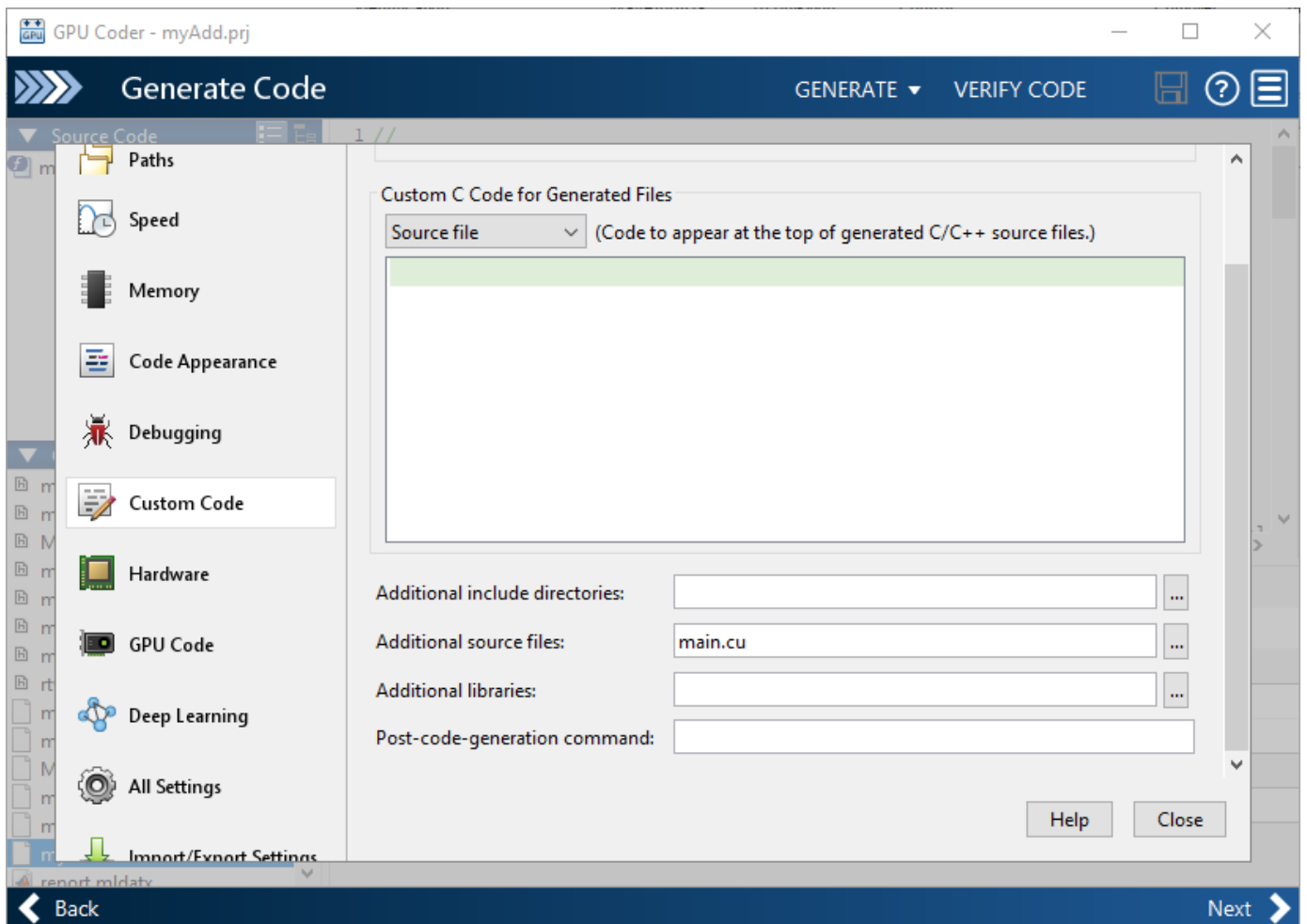
**GPU Coder App**

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.

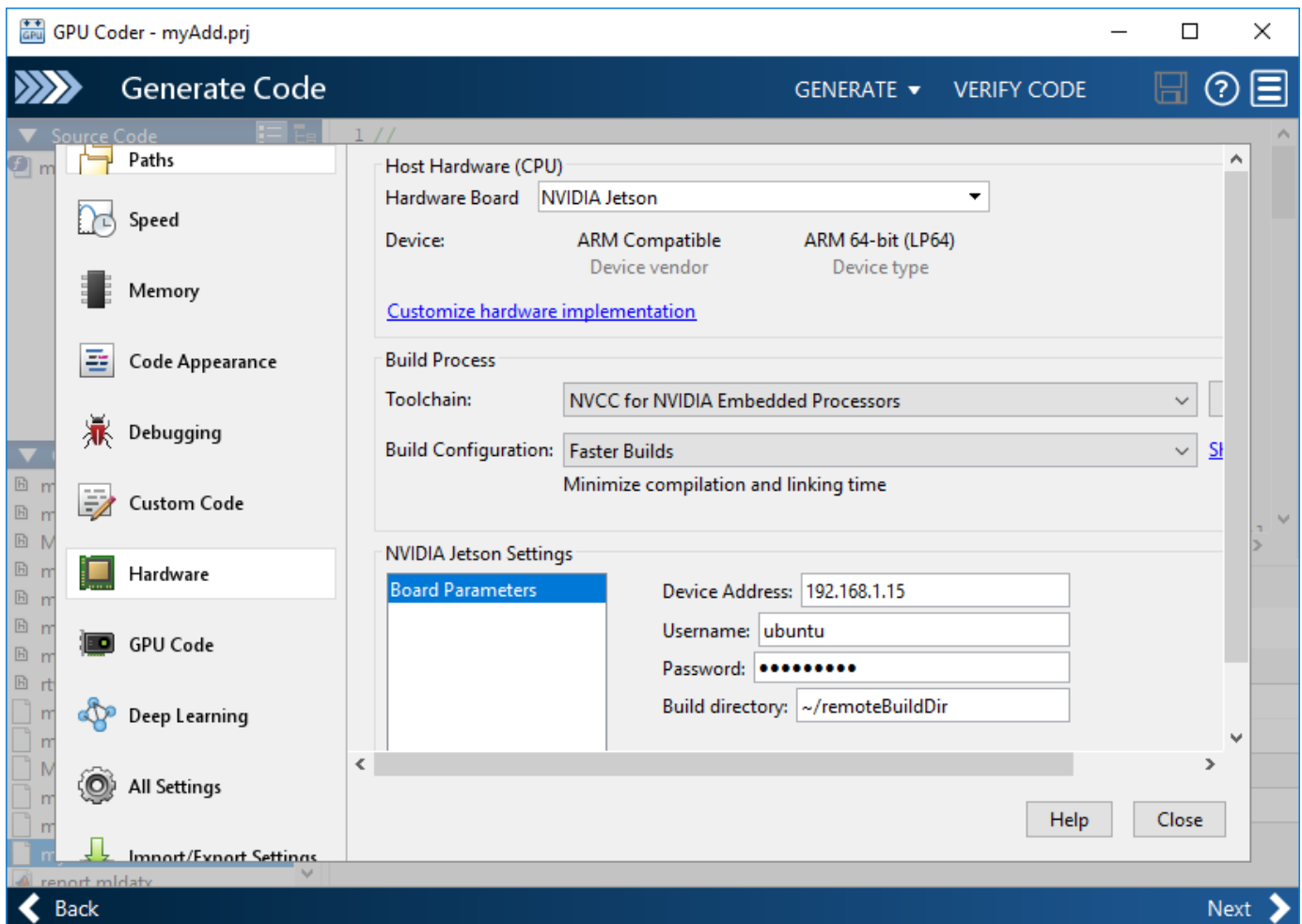
- 1 The app opens the **Select** source files page. Select `myAdd.m` as the entry-point function. Click **Next**.
- 2 In the **Define Input Types** window, enter `myAdd(1:100, 1:100)` and click **Autodefine Input Types**, then click **Next**.
- 3 You can initiate the **Check for Run-Time Issues** process 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

drive | jetson

### More About

- “Build and Run an Executable on NVIDIA Hardware” on page 6-2
- “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 5-48
- “Code Generation for Deep Learning Networks by Using TensorRT” on page 5-57
- “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 6-14

“Specify packNGo Options” on page 6-22

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 2017-2019 The MathWorks, Inc.

maskX = single([-1 0 1 ; -2 0 2; -1 0 1]);
```

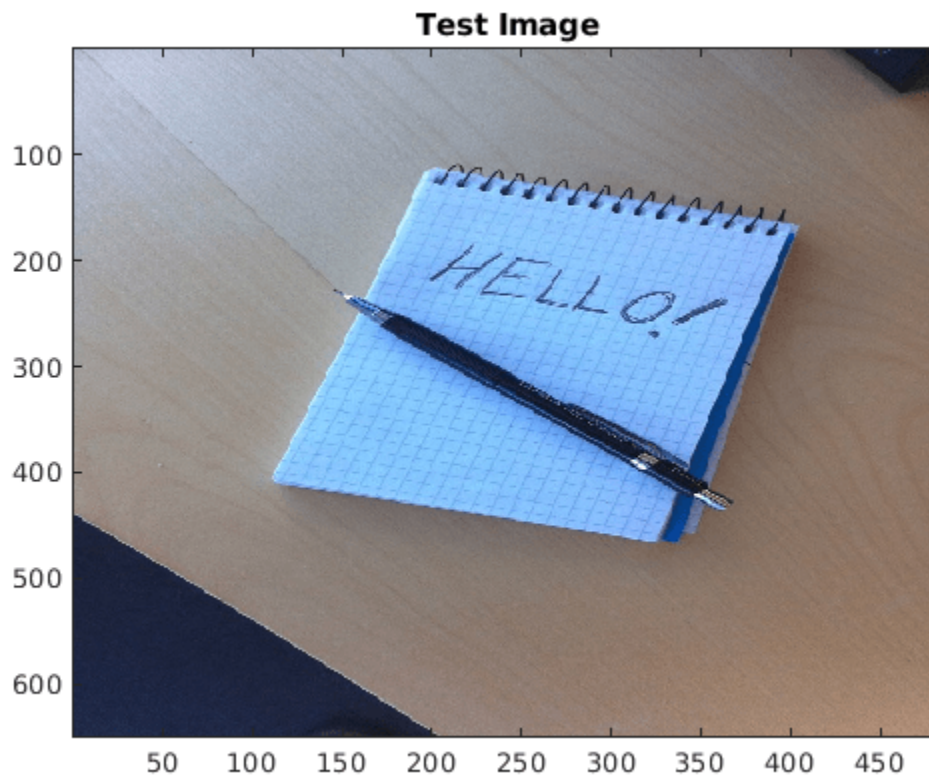
```
maskY = single([-1 -2 -1 ; 0 0 0 ; 1 2 1]);  
  
coder.gpu.kernelfun();  
  
resX = conv2(Image, maskX, 'same');  
resY = conv2(Image, maskY, 'same');  
  
magnitude = sqrt(resX.^2 + resY.^2);  
thresh = magnitude < 0.4;  
magnitude(thresh) = 0;  
  
end
```

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)))/2;  
imSize = size(imGray);  
figure();  
image(im);  
title('Test Image');
```

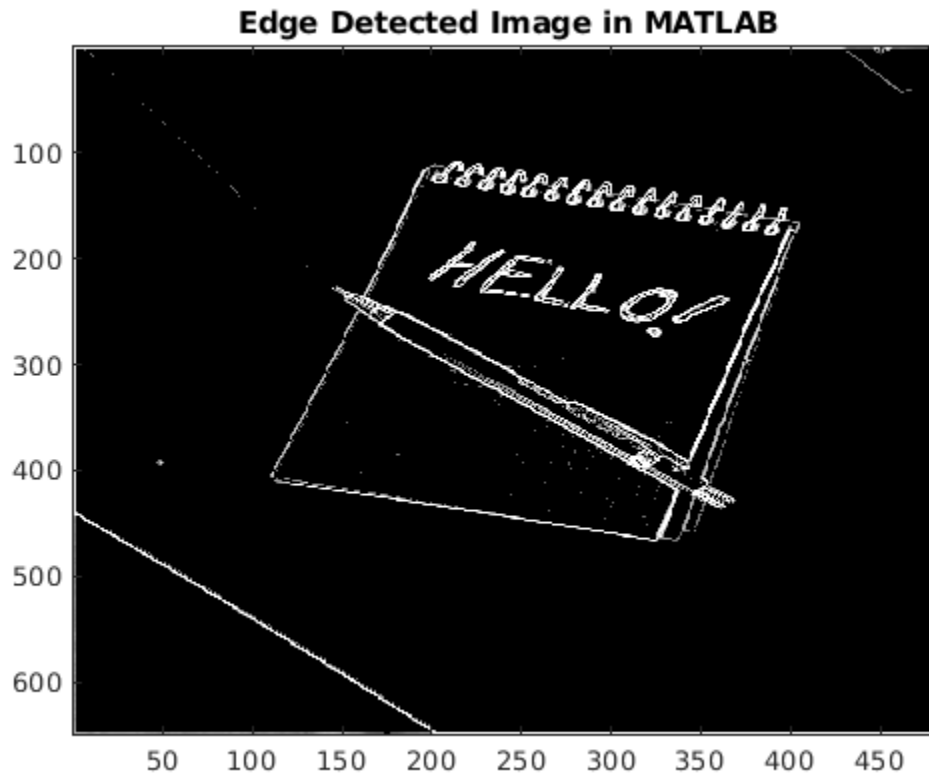


Write the matrix `gray` into the `inputImage.csv` file using the `writematrix` command. The Sobel edge detection application reads in this CSV file.

```
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`

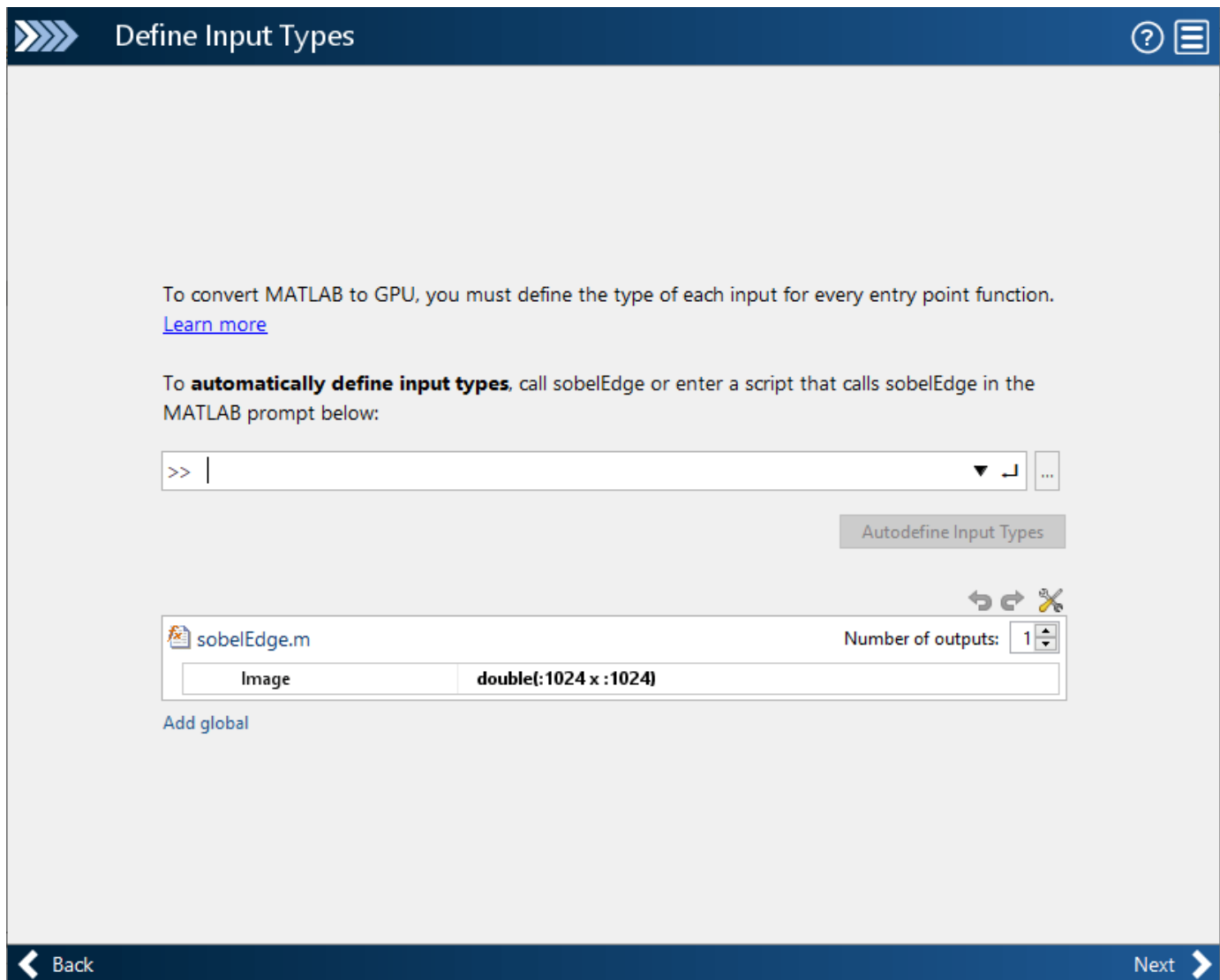
This example uses a custom main file, `main_sobel.cu` and its associated header file `main_sobel.h`. This custom main file reads the input image from the `inputImage.csv` file, calls the `sobelEdge` function in the generated `sobelEdge.cu` file, and saves the data from the edge detected image into the `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.

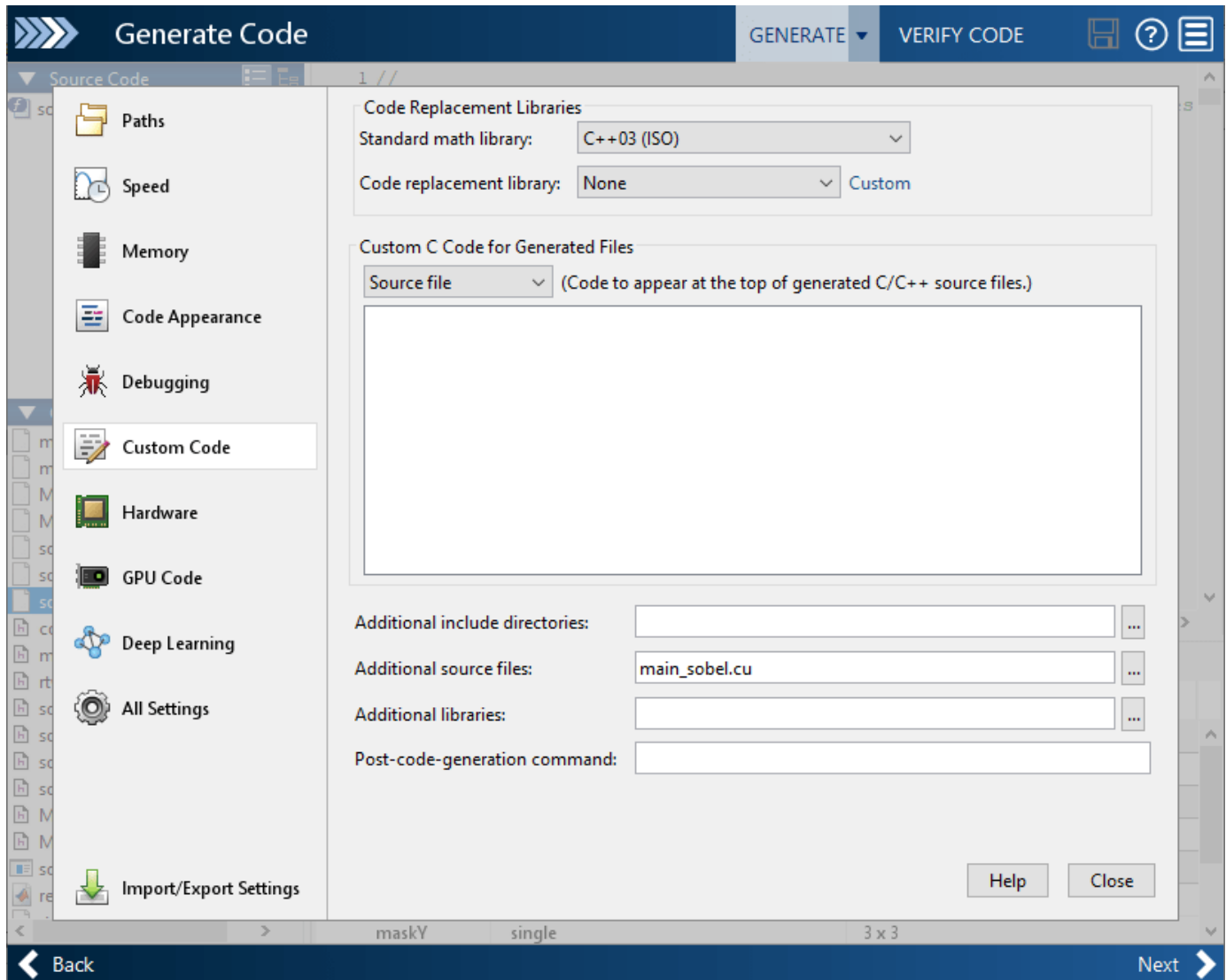
Specify that the input Image is of double data type and variable size with upper bound of 1024. To specify variable size with an upper bound of 1024, select `:1024`. Click **Next** to go to the **Check for Run-Time Issues** page.



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 main heading reads 'Executable Generated Successfully' in bold black text. Underneath this heading is a line of text: '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 the path 'C:\EMpath\Examples\gpcoder-ex06337729\codegen\exe\sobelEdge'; 'Binaries' with a file icon and the path 'C:\EMpath\Examples\gpcoder-ex06337729\sobelEdge.exe'; 'Example main Files' with a folder icon and the path '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 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.

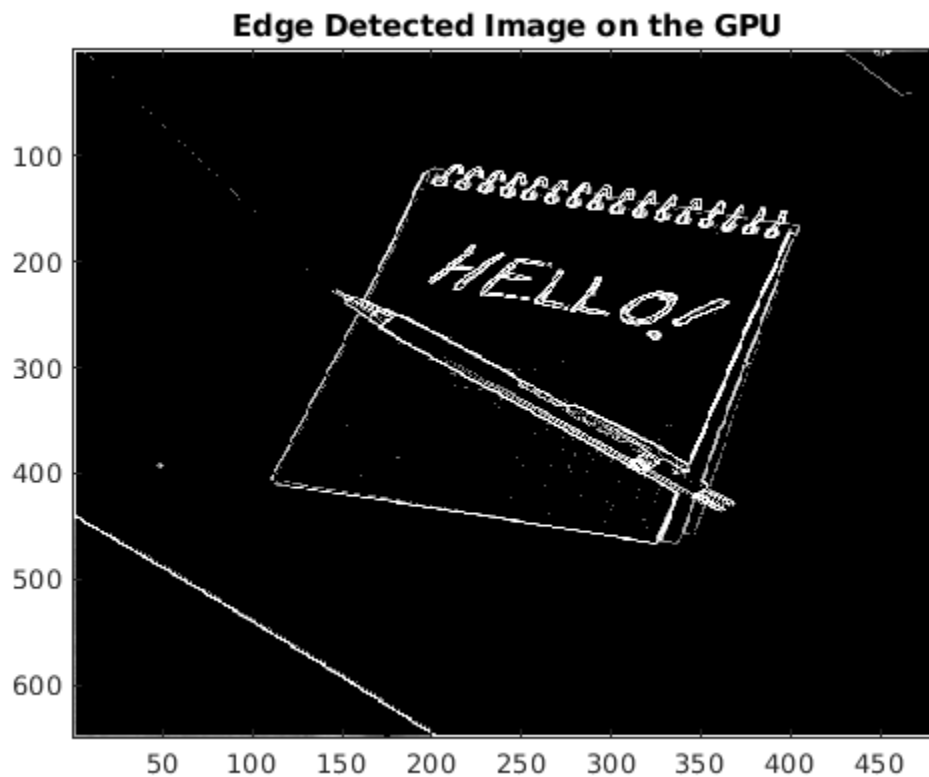
### Standalone Code Execution

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

`codegen` | `coder.gpuConfig` | `packNGo`

### 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

`drive` | `jetson`

### More About

- “Build and Run an Executable on NVIDIA Hardware” on page 6-2
- “Build and Run an Executable on NVIDIA Hardware Using GPU Coder App” on page 6-7
- “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

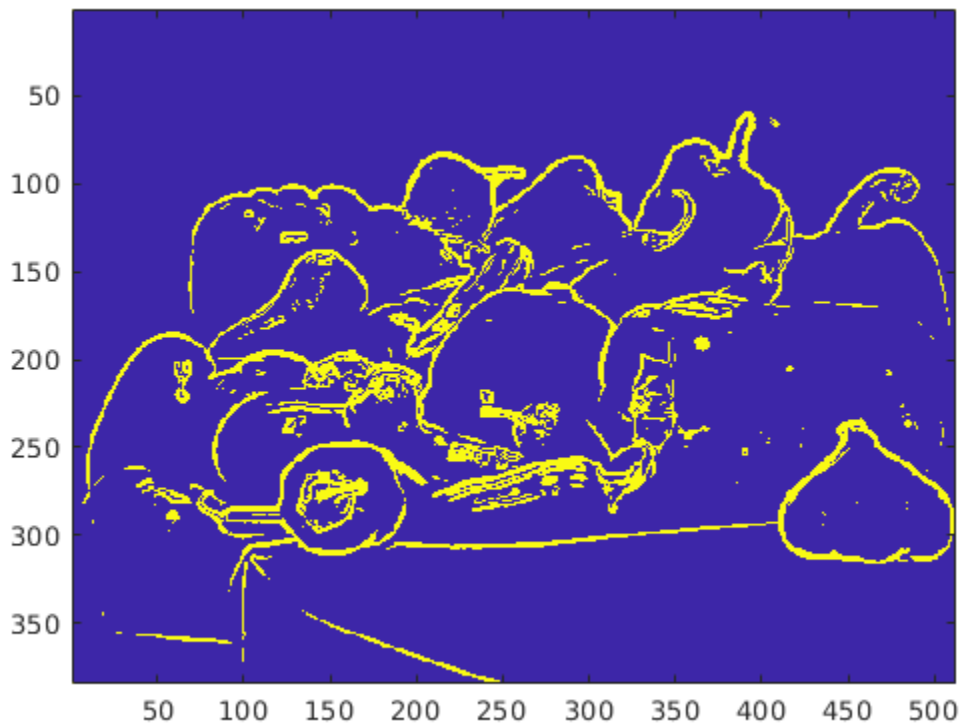
Find the gradient magnitude from the horizontal and vertical gradients (h and v).

```
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(dispatchObj, 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

drive | jetson

## More About

- “Build and Run an Executable on NVIDIA Hardware” on page 6-2
- “Build and Run an Executable on NVIDIA Hardware Using GPU Coder App” on page 6-7
- “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 5-137.

### 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.

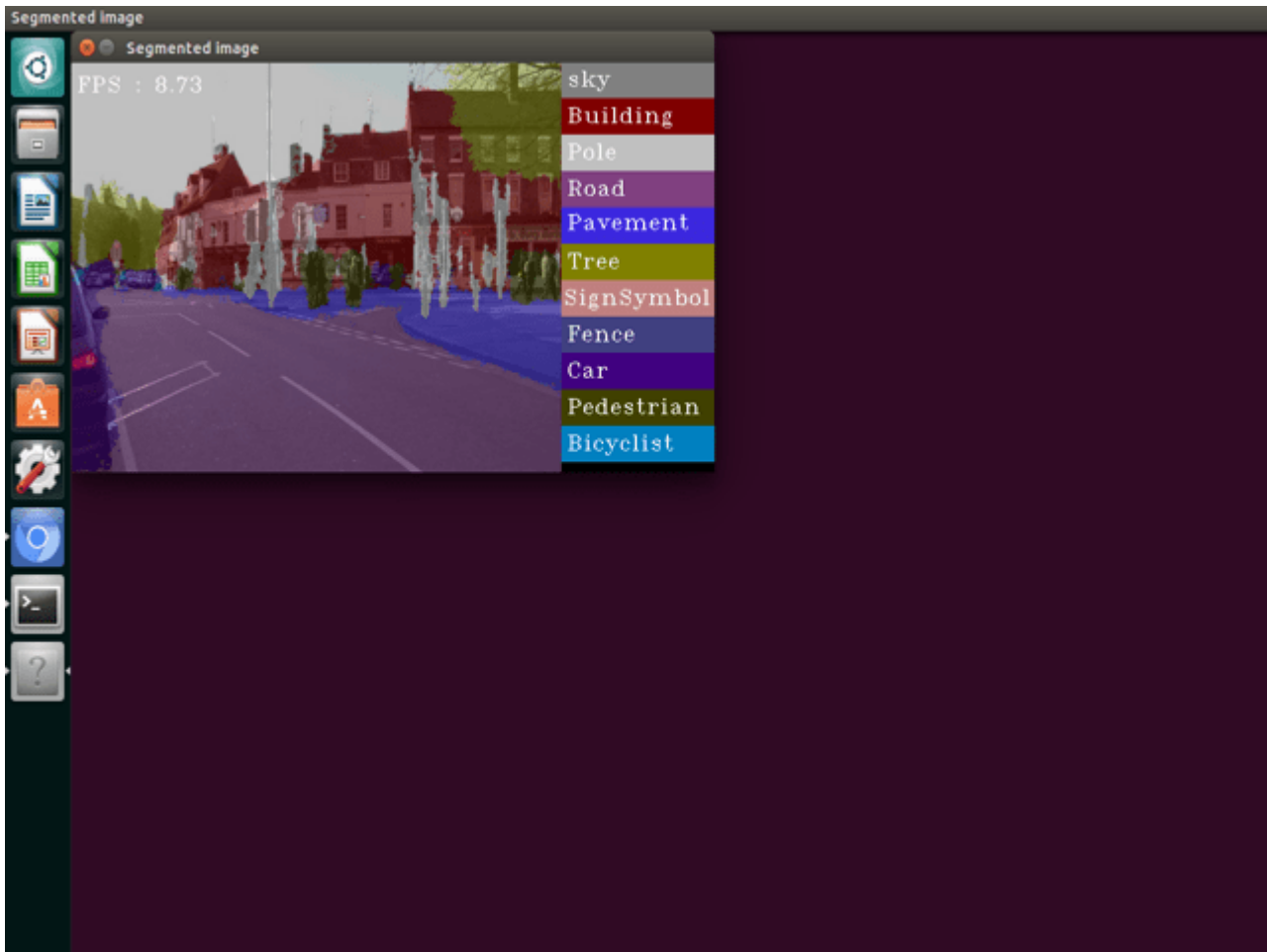
```
hwobj.putFile('CamVid.avi', hwobj.workspaceDir);
```

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

`drive` | `jetson`

### More About

- “Build and Run an Executable on NVIDIA Hardware” on page 6-2
- “Build and Run an Executable on NVIDIA Hardware Using GPU Coder App” on page 6-7

- “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);
```

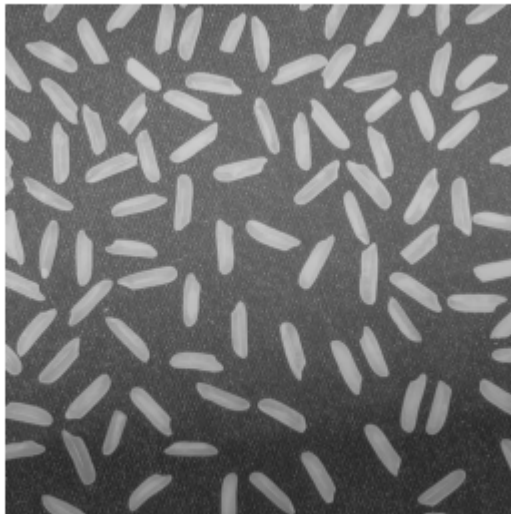
### The `imtophat` Entry-Point Function

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
```

### 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

`drive | jetson`

## **More About**

- “Build and Run an Executable on NVIDIA Hardware” on page 6-2
- “Build and Run an Executable on NVIDIA Hardware Using GPU Coder App” on page 6-7
- “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.

```
hwobj= jetson('host-name', 'username', 'password');
```



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.

```
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

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 as an 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
```

## Run the Application on the Target

Copy the `synsetWords_resnet50` text file from host computer to the target device by using the `putFile` command.

```
hwobj.putFile('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.

```
hwobj.runApplication('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.

```
hwobj.killApplication('resnet50_wrapper');
```

## See Also

### Objects

`drive | jetson`

### More About

- “Build and Run an Executable on NVIDIA Hardware” on page 6-2
- “Build and Run an Executable on NVIDIA Hardware Using GPU Coder App” on page 6-7
- “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)