GPU Coder™

User's Guide

MATLAB®

R2020a

MathWorks®

How to Contact MathWorks

Latest news:	www.mathworks.com
Sales and services:	www.mathworks.com/sales_and_services
User community:	www.mathworks.com/matlabcentral
Technical support:	www.mathworks.com/support/contact_us
Phone:	508-647-7000

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

GPU Coder™ User's Guide

© COPYRIGHT 2017–2020 by The MathWorks, Inc.

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

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

Trademarks

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

Patents

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

Revision History

September 2017	Online only	New for Version 1.0 (Release 2017b)
March 2018	Online only	Revised for Version 1.1 (Release 2018a)
September 2018	Online only	Revised for Version 1.2 (Release 2018b)
March 2019	Online only	Revised for Version 1.3 (Release 2019a)
September 2019	Online only	Revised for Version 1.4 (Release 2019b)
March 2020	Online only	Revised for Version 1.5 (Release 2020a)

Contents

1 Functions Supported for GPU Code Generation

2 Kernel Creation

3 Troubleshooting

4 Deep Learning

5 Targeting Embedded GPU Devices

Functions Supported for GPU Code Generation

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" (MATLAB Coder). 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” (MATLAB Coder).

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” (MATLAB Coder).

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 x 2
n	Input	1 x 1
Z	Local	1 x 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” (MATLAB Coder).

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” (MATLAB Coder).
- 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".

Name	Product	Usage Notes and Limitations
`abs`	MATLAB	No known limitation
`accumneg`	Fixed-Point Designer™	No known limitation
`accumpos`	Fixed-Point Designer	No known limitation
`acos`	MATLAB	Generates an error during simulation and returns `NaN` in generated code when the input value `X` is real, but the output should be complex. To get the complex result, make the input value complex by passing in `complex(X)`.
`acosd`	MATLAB	No known limitation
`acosh`	MATLAB	Generates an error during simulation and returns `NaN` in generated code when the input value `X` is real, but the output should be complex. To get the complex result, make the input value complex by passing in `complex(X)`.
`acot`	MATLAB	No known limitation
`acotd`	MATLAB	No known limitation

`activations`	Deep Learning Toolbox™	• GPU code generation supports the following syntaxes:
		  • `features = activations(net,X,layer)`
		  • `features = activations(__,Name,Value)`
		• The input `X` must not have variable size. The size must be fixed at code generation time.
		• GPU code generation for the `activations` function supports inputs that are defined as half-precision floating point data types. For more information, see `half`.
		• The `layer` argument must be a compile-time constant.
		• Only the `'OutputAs'` and `'MiniBatchSize'` name-value pair arguments are supported for code generation. The value of the `'OutputAs'` name-value pair must be `'channels'`.
		• All name-value pairs must be compile-time constants.

`adaptthresh`	Image Processing Toolbox	The `ForegroundPolarity` and `Statistic` arguments must be compile-time constants.

Name	Product	Usage Notes and Limitations
`affine2d`	Image Processing Toolbox	When generating code, you can only specify singular objects—arrays of objects are not supported.
`alexnet`	Deep Learning Toolbox	• For code generation, you can load the network by using the syntax `net = alexnet` or by passing the `alexnet` function to `coder.loadDeepLearningNetwork`. For example: `net = coder.loadDeepLearningNetwork('alexnet')`. For more information, see "Load Pretrained Networks for Code Generation" on page 4-15. • The syntax `alexnet('Weights','none')` is not supported for GPU code generation.
`and`	MATLAB	No known limitation
`angle`	MATLAB	No known limitation
`asin`	MATLAB	Generates an error during simulation and returns `NaN` in generated code when the input value `X` is real, but the output should be complex. To get the complex result, make the input value complex by passing in `complex(X)`.
`asind`	MATLAB	No known limitation
`asinh`	MATLAB	No known limitation
`atan`	MATLAB	No known limitation
`atan2`	MATLAB	If you use `atan2` with single type and double type operands, the generated code might not produce the same result as MATLAB. See "Binary Element-Wise Operations with Single and Double Operands" (MATLAB Coder).
`atan2d`	MATLAB	If you use `atan2d` with single type and double type operands, the generated code might not produce the same result as MATLAB. See "Binary Element-Wise Operations with Single and Double Operands" (MATLAB Coder).
`atand`	MATLAB	No known limitation
`atanh`	MATLAB	Generates an error during simulation and returns `NaN` in generated code when the input value `x` is real, but the output should be complex. To get the complex result, make the input value complex by passing in `complex(x)`.
`bin2dec`	MATLAB	• Input text must be specified as a character array. Cell arrays are not supported. • When the input is empty, the answer does not match the answer in MATLAB.
`bitand`	MATLAB	No known limitation
`bitcmp`	MATLAB	No known limitation
`bitget`	MATLAB	No known limitation
`bitor`	MATLAB	No known limitation
`bitrevorder`	Signal Processing Toolbox™	No known limitation

Name	Product	Usage Notes and Limitations
bitset	MATLAB	No known limitation
bitshift	MATLAB	No known limitation
bitsll	Fixed-Point Designer	Generated code might not handle out of range shifting.
bitsra	Fixed-Point Designer	Generated code might not handle out of range shifting.
bitsrl	Fixed-Point Designer	Generated code might not handle out of range shifting.
bitxor	MATLAB	No known limitation
blkdiag	MATLAB	No known limitation
bsxfun	MATLAB	Code generation does not support sparse matrix inputs for this function.
bwareaopen	Image Processing Toolbox	• BW must be a 2-D binary image. N-D arrays are not supported. • conn must be one of the two-dimensional connectivities (4 or 8) or a 3-by-3 matrix. The 3-D connectivities (6, 18, and 26) are not supported. Matrices of size 3-by-3-by-...-by-3 are not supported. • conn must be a compile-time constant.
bwboundaries	Image Processing Toolbox	• The parameter conn must be a compile-time constant. • The parameter options must be a compile-time constant. • The return value A can only be a full matrix, not a sparse matrix.
bwconncomp	Image Processing Toolbox	• bwconncomp only supports 2-D inputs. • The conn arguments must be a compile-time constant and the only connectivities supported are 4 or 8. You can also specify connectivity as a 3-by-3 matrix, but it can only be [0 1 0;1 1 1;0 1 0] or ones(3) • The PixelIdxList field in the CC struct return value is not supported.
bwdist	Image Processing Toolbox	When generating code, the optional second input argument, method, must be a compile-time constant. Input images must have fewer than 2^{32} pixels.
bweuler	Image Processing Toolbox	No known limitation
bwlabel	Image Processing Toolbox	When generating code, the parameter n must be a compile-time constant.
bwlookup	Image Processing Toolbox	When generating code, specify an input image of class logical.
bwmorph	Image Processing Toolbox	When generating code, the character vectors or string scalars specifying the operation must be a compile-time constant and, for best results, the input image must be of class logical.

Name	Product	Usage Notes and Limitations
`bwperim`	Image Processing Toolbox	• `bwperim` supports only 2-D images. • `bwperim` does not support a no-output-argument syntax. • The connectivity matrix input argument, `conn`, must be a constant.

`bwselect`	Image Processing Toolbox	• When generating code, `bwselect` supports only these syntaxes:
		  • `BW2 = bwselect(BW, c, r)`
		  • `[BW2, idx] = bwselect(BW, c, r)`
		  • `BW2 = bwselect(BW, c, r, n)`
		  • `[BW2, idx] = bwselect(BW, c, r, n)`
		• In addition, the optional fourth input argument, `n`, must be a compile-time constant.

`bwtraceboundary`	Image Processing Toolbox	When generating code, the `dir`, `fstep`, and `conn` arguments must be compile-time constants.
`bwunpack`	Image Processing Toolbox	When generating code, all input arguments must be compile-time constants.
`cart2pol`	MATLAB	No known limitation
`cast`	MATLAB	Enumeration inputs must be scalar valued at compile time. Arrays of enumerations are not supported.
`ceil`	MATLAB	Code generation does not support `char` or `logical` data types for `X`.
`chol`	MATLAB	Only the first two syntaxes `chol(A)` and `chol(A,triangle)` with one output argument are supported.
`circshift`	MATLAB	Code generation does not support tables and cells for the first input argument.

Name	Product	Usage Notes and Limitations

`classify`	Deep Learning Toolbox	• GPU code generation supports the following syntaxes:
		  • `[YPred,scores] = classify(net,X)`
		  • `[YPred,scores] = classify(net,sequences)`
		  • `[YPred,scores] = classify(__,Name,Value)`
		• GPU code generation for the `classify` function is not supported for regression networks and networks with multiple outputs.
		• GPU code generation for the `classify` function supports inputs that are defined as half-precision floating point data types. For more information, see `half`.
		• The input `X` must not have variable size. The size must be fixed at code generation time.
		• GPU code generation supports only vector sequences. The sequence length can be variable sized. The feature dimension must be fixed at code generation time.
		• Only the `'MiniBatchSize'`, `'SequenceLength'`, `'SequencePaddingDirection'`, and `'SequencePaddingValue'` name-value pair arguments are supported for code generation. All name-value pairs must be compile-time constants.
		• Only the `'longest'` and `'shortest'` option of the `'SequenceLength'` name-value pair is supported for code generation.

`classUnderlying`	MATLAB	No known limitation
`compan`	MATLAB	No known limitation
`complex`	MATLAB	No known limitation
`conj`	MATLAB	No known limitation
`conndef`	Image Processing Toolbox	When generating code, the `num_dims` and `type` arguments must be compile-time constants.
`conv`	MATLAB	If the inputs have nonfinite values (`inf` or `NaN`), the results from the generated code may not numerically match MATLAB simulation.
`conv2`	MATLAB	If the inputs have nonfinite values (`inf` or `NaN`), the results from the generated code may not numerically match MATLAB simulation.
`cos`	MATLAB	No known limitation
`cosh`	MATLAB	No known limitation
`cot`	MATLAB	No known limitation
`coth`	MATLAB	No known limitation
`cross`	MATLAB	• If supplied, `dim` must be a constant. • Code generation does not support sparse matrix inputs for this function.
`csc`	MATLAB	No known limitation
`csch`	MATLAB	No known limitation

Name	Product	Usage Notes and Limitations
`ctranspose`	MATLAB	No known limitation
`cwt`	Wavelet Toolbox™	• Single- and double-precision input signal are supported. The precision must be set at compile time. • Timetable input signal is not supported. • Only analytic Morse (`'morse'`) and Morlet (`'amor'`) wavelets are supported. • The following input arguments are not supported: Sampling period (`ts`), `PeriodLimits` name-value pair, `NumOctave` name-value pair, and `FilterBank` name-value pair. • Scaling coefficient output and filter bank output are not supported. • Plotting is not supported.
`cummax`	MATLAB	No known limitation
`cummin`	MATLAB	No known limitation
`cumprod`	MATLAB	• Logical inputs are not supported. Cast input to `double` first. • Code generation does not support sparse matrix inputs for this function.
`cumsum`	MATLAB	• Logical inputs are not supported. Cast input to `double` first. • Code generation does not support sparse matrix inputs for this function.
`DAGNetwork`	Deep Learning Toolbox	• Only the `activations`, `predict`, and `classify` methods are supported. • To create a `DAGNetwork` object for code generation, see "Load Pretrained Networks for Code Generation" on page 4-15.
`darknet19`	Deep Learning Toolbox	• For code generation, you can load the network by using the syntax `net = darknet19` or by passing the `darknet19` function to `coder.loadDeepLearningNetwork`. For example: `net = coder.loadDeepLearningNetwork('darknet19')`. For more information, see "Load Pretrained Networks for Code Generation" on page 4-15. • The syntax `darknet19('Weights','none')` is not supported for GPU code generation.
`darknet53`	Deep Learning Toolbox	• For code generation, you can load the network by using the syntax `net = darknet53` or by passing the `darknet53` function to `coder.loadDeepLearningNetwork`. For example: `net = coder.loadDeepLearningNetwork('darknet53')`. For more information, see "Load Pretrained Networks for Code Generation" on page 4-15. • The syntax `darknet53('Weights','none')` is not supported for GPU code generation.
`deg2rad`	MATLAB	No known limitation
`del2`	MATLAB	No known limitation

Name	Product	Usage Notes and Limitations
`demosaic`	Image Processing Toolbox	`sensorAlignment` must be a compile-time constant.
`deeplabv3plusLayers`	Deep Learning Toolbox	For code generation, you must first create a DeepLab v3+ network by using the `deeplabv3plusLayers` function. Then, use the `trainNetwork` function on the resulting `lgraph` object to train the network for segmentation. Once the network is trained and evaluated, you can generate code for the deep learning network object using GPU Coder.
`densenet201`	Deep Learning Toolbox	• For code generation, you can load the network by using the syntax `net = densenet201` or by passing the `densenet201` function to `coder.loadDeepLearningNetwork`. For example: `net = coder.loadDeepLearningNetwork('densenet201')`. For more information, see “Load Pretrained Networks for Code Generation” on page 4-15. • The syntax `densenet201('Weights','none')` is not supported for GPU code generation.
`det`	MATLAB	Code generation does not support sparse matrix inputs for this function.

`diag`	MATLAB	• If you supply `k`, then it must be a real and scalar integer value.
		• For variable-size inputs that are variable-length vectors (1-by-: or :-by-1), `diag`:
		  • Treats the input as a vector
		  • Returns a matrix with the input vector along the specified diagonal
		• For variable-size inputs that are not variable-length vectors, `diag`:
		  • Treats the input as a matrix
		  • Does not support inputs that are vectors at run time
		  • Returns a variable-length vector
		
		If the input is variable-size (:m-by-:n) and has shape 0-by-0 at run time, then the output is 0-by-1, not 0-by-0. However, if the input is a constant size 0-by-0, then the output is `[]`.
		• For variable-size inputs that are not variable-length vectors (1-by-: or :-by-1), `diag` treats the input as a matrix from which to extract a diagonal vector. This behavior occurs even if the input array is a vector at run time. To force `diag` to build a matrix from variable-size inputs that are not 1-by-: or :-by-1, use:
		  • `diag(x(:))` instead of `diag(x)`
		  • `diag(x(:),k)` instead of `diag(x,k)`

Name	Product	Usage Notes and Limitations
disparitySGM	Computer Vision Toolbox™	• The input images I1 and I2 must be rectified, same size, and of same data type. • GPU code generation supports the 'UniquenessThreshold' and 'disparityMap' name-value pairs. • For very large inputs, the memory requirements of the algorithm may exceed the GPU device limits. In such cases, consider reducing the input size to proceed with code generation.
double	MATLAB	For string inputs with misplaced commas (commas that are not used as thousands separators), generated code results can differ from MATLAB results.
edge	Image Processing Toolbox	• The method, direction, and sigma arguments must be compile-time constants. • The 'approxcanny' method is not supported. • Nonprogrammatic syntaxes are not supported. For example, if you do not specify a return value, then edge displays an image. This syntax is not supported with code generation.
exp	MATLAB	No known limitation
eye	MATLAB	• typename must be a built-in MATLAB numeric type. Does not invoke the static eye method for other classes. For example, eye(m, n, 'myclass') does not invoke myclass.eye(m,n). • Size arguments must have a fixed size.
factorial	MATLAB	No known limitation
fft	MATLAB	No known limitation
fft2	MATLAB	No known limitation
fftfilt	Signal Processing Toolbox	Digital filter objects are not supported for code generation.
fftn	MATLAB	The sz argument must have a fixed size.
fftshift	MATLAB	No known limitation
filter	MATLAB	• If supplied, dim must be a constant. • See “Variable-Sizing Restrictions for Code Generation of Toolbox Functions” (MATLAB Coder). • If the inputs have nonfinite values (inf or NaN), the results from the generated code may not numerically match MATLAB simulation.
filter2	MATLAB	No known limitation
fitgeotrans	Image Processing Toolbox	• When generating code, the transformationType argument must be a compile-time constant and only the following transformation types are supported: 'nonreflectivesimilarity', 'similarity', 'affine', and 'projective'.
fix	MATLAB	Code generation does not support char or logical data types for X.
floor	MATLAB	Code generation does not support char or logical data types for X.
fspecial	Image Processing Toolbox	When generating code, all inputs must be constants at compilation time.

Name	Product	Usage Notes and Limitations
gather	MATLAB	No known limitation
ge	MATLAB	No known limitation
getrangefromclass	MATLAB	No known limitation
googlenet	Deep Learning Toolbox	• For code generation, you can load the network by using the syntax `net = googlenet` or by passing the `googlenet` function to `coder.loadDeepLearningNetwork`. For example: `net = coder.loadDeepLearningNetwork('googlenet')`. For more information, see “Load Pretrained Networks for Code Generation” on page 4-15. • The syntax `googlenet('Weights','none')` is not supported for GPU code generation.
gt	MATLAB	No known limitation
half	MATLAB	• CUDA compute capability of 5.3 or higher is required for generating and executing code with half-precision data types. • CUDA toolkit version of 10.0 or higher is required for generating and executing code with half-precision data types. • The memory allocation (`malloc`) mode for generating CUDA code must be set to `'Discrete'`. For more information, see `coder.gpuConfig`. • Half-precision complex data types are not supported for GPU code generation. • For GPU Code generation, half-precision matrix multiplication can only be performed with real inputs. • In MATLAB, the `isobject` function returns true with a half-precision input. However, in generated code, this function returns false. • If your target hardware does not have native support for half-precision, then half is used as a storage type, with arithmetic operations performed in single precision. • Some functions use half only as a storage type and the arithmetic is always performed in single-precision, regardless of the target hardware. • Code generation for 32-bit targets is not supported if your MATLAB code contains half-precision data types.
histeq	Image Processing Toolbox	When generating code, `histeq` does not support indexed images.
hough	Image Processing Toolbox	• The optional parameters `'Theta'` and `'RhoResolution'` must be compile-time string constants. • The optional `Theta` vector must have a bounded size.

Name	Product	Usage Notes and Limitations
`houghlines`	Image Processing Toolbox	The optional parameter names `'FillGap'` and `'MinLength'` must be compile-time constants. Their associated values need not be compile-time constants.
`houghpeaks`	Image Processing Toolbox	The optional parameter names `'Threshold'` and `'NHoodSize'` must be compile-time constants. Their associated values need not be compile-time constants.
`hsv2rgb`	MATLAB	No known limitation
`hypot`	MATLAB	If you use `hypot` with single type and double type operands, the generated code might not produce the same result as MATLAB. See “Binary Element-Wise Operations with Single and Double Operands” (MATLAB Coder).
`ifft`	MATLAB	• Output is complex. • Symmetry type `'symmetric'` is not supported. • For limitations related to variable-size data, see “Variable-Sizing Restrictions for Code Generation of Toolbox Functions” (MATLAB Coder).
`ifft2`	MATLAB	Symmetry type `'symmetric'` is not supported.
`ifftn`	MATLAB	• Symmetry type `'symmetric'` is not supported. • The `sz` argument must have a fixed size.
`ifftshift`	MATLAB	No known limitation
`im2double`	MATLAB	No known limitation
`im2int16`	Image Processing Toolbox	No known limitation
`im2single`	Image Processing Toolbox	No known limitation
`im2uint8`	Image Processing Toolbox	No known limitation
`imabsdiff`	Image Processing Toolbox	No known limitation
`imadjust`	Image Processing Toolbox	When generating code, `imadjust` does not support indexed images.
`imag`	MATLAB	No known limitation
`imbinarize`	Image Processing Toolbox	When generating code, all character vector input arguments must be compile-time constants.
`imbothat`	Image Processing Toolbox	• The input image `I` must be 2-D or 3-D. • The structuring element `SE` must be a compile-time constant.
`imboxfilt`	Image Processing Toolbox	When generating code, all character vector input arguments must be compile-time constants.
`imclearborder`	Image Processing Toolbox	• Supports only up to 3-D inputs. • The optional second input argument, `conn`, must be a compile-time constant.

Name	Product	Usage Notes and Limitations
`imclose`	Image Processing Toolbox	• The input image `I` must be 2-D or 3-D. • The structuring element `SE` must be a compile-time constant.
`imcomplement`	Image Processing Toolbox	`imcomplement` does not support `int64` and `uint64` data types.

`imcrop`	Image Processing Toolbox	• The interactive syntaxes are not supported, including:
		  • `J = imcrop`
		  • `J = imcrop(I)`
		  • `X2 = imcrop(X,cmap)`
		  • `J = imcrop(h)`
		• Indexed images are not supported, including the non-interactive syntax `X2 = imcrop(X,cmap,rect);`

`imdilate`	Image Processing Toolbox	• The input image, `IM`, must be 2-D or 3-D. • The structuring element argument `SE` must be a compile-time constant. • Packed binary input image (`PACKOPT` syntax) is not supported. • For 3-D input images with more than three channels, only C/C++ code is generated. • CUDA code is generated only for 1-D or 2-D structuring elements. If the structuring element is 3-D, C/C++ code is generated. Code generation is not supported for structuring elements with more than three dimensions. • For non-flat structuring elements, only C/C++ code is generated.
`imerode`	Image Processing Toolbox	
`imfill`	Image Processing Toolbox	• The optional input arguments, `conn` and `'holes'`, must be compile-time constants. • `imfill` supports up to 3-D inputs only. (No N-D support.) • The interactive syntax to select points, `imfill(BW,0,CONN)` is not supported. • With the `locations` input argument, once you select a format at compile time, you cannot change it at run time. However, the number of points in locations can be varied at run time.

Name	Product	Usage Notes and Limitations
`imfilter`	Image Processing Toolbox	• When generating code, the input image, `A`, must be 2-D or 3-D. The value of the input argument, `options`, must be a compile-time constant. • If you specify a large kernel `h`, a kernel that contains large values, or specify an image containing large values, you can see different results between MATLAB and generated code using codegen for floating point data types. This happens because of accumulation errors due to different algorithm implementations. • With CUDA toolkit v9.0, a bug in the NVIDIA® optimization causes numerical mismatch between the results from the generated code and MATLAB. As a workaround, turn off the optimization by passing the following flags to the configuration object (`cfg`) before generating the code. `cfg.GpuConfig.CompilerFlags = ‘-Xptxas -O0’` NVIDIA is expected to fix this bug in CUDA toolkit v9.1.
`imgaussfilt`	Image Processing Toolbox	• `imgaussfilt` does not support the `FilterDomain` parameter for code generation. Filtering is always done in the `'spatial'` domain in generated code. • When generating code, all character vector input arguments must be compile-time constants.
`imgradient3`	Image Processing Toolbox	When generating code, the input argument `method` must be a compile-time constant.
`imgradientxyz`	Image Processing Toolbox	When generating code, the input argument `method` must be a compile-time constant.
`imhist`	Image Processing Toolbox	• If the first input is a binary image, then `n` must be a scalar constant of value 2 at compile time. • Nonprogrammatic syntaxes are not supported. For example, the syntax `imhist(I)`, where `imhist` displays the histogram, is not supported.
`imhmax`	Image Processing Toolbox	When generating code, the optional third input argument, `conn`, must be a compile-time constant.
`immse`	Image Processing Toolbox	No known limitation
`imopen`	Image Processing Toolbox	• The input image `I` must be 2-D or 3-D. • The structuring element `SE` must be a compile-time constant.
`imoverlay`	Image Processing Toolbox	When generating code, if you specify `color` as a character vector, then the value must be a compile-time constant.
`imreconstruct`	Image Processing Toolbox	• When generating code, the optional third input argument, `conn`, must be a compile-time constant, and can only take the value `4` or `8`. • `imreconstruct` does not support `uint64` and `int64` data types for code generation.
`impyramid`	Image Processing Toolbox	`direction` must be a compile-time constant.

Name	Product	Usage Notes and Limitations
`imquantize`	Image Processing Toolbox	No known limitation
`imread`	Image Processing Toolbox	• Supports reading of 8-bit JPEG images only. The input argument `filename` must be a valid absolute path or relative path.
`imresize`	Image Processing Toolbox	• `'Colormap'` and `'Dither'` Name-Value pair arguments are not supported. • Indexed image is not supported. • Custom interpolation kernel is not supported. • For certain interpolation kernels, there may be a small numerical mismatch between the results in MATLAB and the generated code.
`imrotate`	Image Processing Toolbox	• Input images of data type categorical are not supported. • The `method` and `bbox` arguments must be compile-time constants.
`imtophat`	Image Processing Toolbox	• The image input `I` must be 2-D or 3-D. • The structuring element `SE` must be a compile-time constant.
`imwarp`	Image Processing Toolbox	• Input images of data type categorical are not supported. • The geometric transformation object input, `tform`, must be an `affine2d` or `projective2d` object and must be constant. • The interpolation method and optional parameter names must be constants. • The spatial referencing information output, `RB`, is not supported.
`inceptionresnetv2`	Deep Learning Toolbox	For code generation, you can load the network by using the syntax `net = inceptionresnetv2` or by passing the `inceptionresnetv2` function to `coder.loadDeepLearningNetwork`. For example: `net = coder.loadDeepLearningNetwork('inceptionresnetv2')` For more information, see “Load Pretrained Networks for Code Generation” on page 4-15.
`inceptionv3`	Deep Learning Toolbox	• For code generation, you can load the network by using the syntax `net = inceptionv3` or by passing the `inceptionv3` function to `coder.loadDeepLearningNetwork`. For example: `net = coder.loadDeepLearningNetwork('inceptionv3')`. For more information, see “Load Pretrained Networks for Code Generation” on page 4-15. • The syntax `inceptionv3('Weights','none')` is not supported for GPU code generation.
`int8`, `int16`, `int32`, `int64`	MATLAB	No known limitation
`integralBoxFilter`	Image Processing Toolbox	The `'NormalizationFactor'` parameter must be a compile-time constant.

Name	Product	Usage Notes and Limitations

interp2	MATLAB	• Xq and Yq must be the same size. Use meshgrid to evaluate on a grid.
		• For best results, provide X and Y as vectors. The values in these vectors must be strictly monotonic and increasing.
		• Code generation does not support the 'makima' interpolation method.
		• For the 'cubic' interpolation method, if the grid does not have uniform spacing, an error results. In this case, use the 'spline' interpolation method.
		• For best results when you use the 'spline' interpolation method:
		  • Use meshgrid to create the inputs Xq and Yq.
		  • Use a small number of interpolation points relative to the dimensions of V. Interpolating over a large set of scattered points can be inefficient.

intlut	Image Processing Toolbox	No known limitation
isUnderlying	MATLAB	No known limitation
isequal	MATLAB	No known limitation
isfloat	MATLAB	No known limitation
isinteger	MATLAB	No known limitation
islogical	MATLAB	No known limitation
ismatrix	MATLAB	No known limitation
isnumeric	MATLAB	No known limitation
isreal	MATLAB	No known limitation
isrow	MATLAB	No known limitation
issparse	MATLAB	No known limitation
issymmetric	MATLAB	Code generation does not support sparse matrix inputs for this function.
istft	Signal Processing Toolbox	The 'ConjugateSymmetric' argument is not supported for code generation.
istril	MATLAB	Code generation does not support sparse matrix inputs for this function.
istriu	MATLAB	Code generation does not support sparse matrix inputs for this function.
isvector	MATLAB	No known limitation
kron	MATLAB	Code generation does not support sparse matrix inputs for this function.
lab2rgb	Image Processing Toolbox	When generating code, all character vector input arguments must be compile-time constants.

Name	Product	Usage Notes and Limitations
`label2idx`	Image Processing Toolbox	No known limitation
`ldivide`	MATLAB	If you use `ldivide` with single type and double type operands, the generated code might not produce the same result as MATLAB. See "Binary Element-Wise Operations with Single and Double Operands" (MATLAB Coder).
`le`	MATLAB	No known limitation
`length`	MATLAB	No known limitation

`linsolve`	MATLAB	• The `opts` structure must be a constant scalar. Code generation does not support arrays of options structures.
		• Code generation only optimizes these cases:
		  • `UT`
		  • `LT`
		  • `UHESS` = `true` (the `TRANSA` can be either `true` or `false`)
		  • `SYM` = `true` and `POSDEF` = `true`
		Other options are equivalent to using `mldivide`.
		• Code generation does not support sparse matrix inputs for this function.

`log`	MATLAB	When the input value `x` is real, but the output should be complex, simulation ends with an error. To produce the complex result, make the input value complex by passing in `complex(x)`.
`log10`	MATLAB	No known limitation
`log1p`	MATLAB	No known limitation
`logical`	MATLAB	No known limitation
`lt`	MATLAB	No known limitation
`lu`	MATLAB	Code generation does not support sparse matrix inputs for this function.
`matchFeatures`	Computer Vision Toolbox	CUDA code is generated only for the `exhaustive` matching method. If the `Approximate` method is selected, GPU Coder issues a warning and generates C/C++ code for this function.
`mean`	MATLAB	• If you specify `dim`, then it must be a constant. • The `outtype` and `nanflag` options must be constant character vectors. • Integer types do not support the `'native'` output data type option.
`mean2`	Image Processing Toolbox	No known limitation
`medfilt2`	Image Processing Toolbox	When generating code, the `padopt` argument must be a compile-time constant.
`meshgrid`	MATLAB	No known limitation
`mfcc`	Audio Toolbox™	No known limitation

Name	Product	Usage Notes and Limitations
`minus`	MATLAB	If you use `minus` with single type and double type operands, the generated code might not produce the same result as MATLAB. See “Binary Element-Wise Operations with Single and Double Operands” (MATLAB Coder).
`mldivide`	MATLAB	No known limitation
`mobilenetv2`	Deep Learning Toolbox	• For code generation, you can load the network by using the syntax `net = mobilenetv2` or by passing the `mobilenetv2` function to `coder.loadDeepLearningNetwork`. For example: `net = coder.loadDeepLearningNetwork('mobilenetv2')` For more information, see “Load Pretrained Networks for Code Generation” on page 4-15. • The syntax `mobilenetv2('Weights','none')` is not supported for GPU code generation.
`mpower`	MATLAB	• If `A` is a 2-by-2 or larger matrix and `B` is `Inf` or `-Inf`, then `A^B` returns a matrix of `NaN` values. • For `A^b`, if `b` is a noninteger scalar, then at least one of `A` or `b` must be complex. • Code generation does not support sparse matrix inputs for this function.
`mrdivide`	MATLAB	Code generation does not support sparse matrix inputs for this function.
`mtimes`	MATLAB	Multiplication of pure imaginary numbers by non-finite numbers might not match MATLAB. The code generator does not specialize multiplication by pure imaginary numbers—it does not eliminate calculations with the zero real part. For example, `(Inf + 1i)*1i = (Inf*0 – 1*1) + (Inf*1 + 1*0)i = NaN + Infi`.
`multithresh`	Image Processing Toolbox	The input argument `N` must be a compile-time constant.
NaN or `nan`	MATLAB	Dimensions must be real, nonnegative, integers.
`nasnetmobile`	Deep Learning Toolbox	For code generation, you can load the network by using the syntax `net = nasnetmobile` or by passing the `nasnetmobile` function to `coder.loadDeepLearningNetwork`. For example: `net = coder.loadDeepLearningNetwork('nasnetmobile')` For more information, see “Load Pretrained Networks for Code Generation” on page 4-15.
`nasnetlarge`	Deep Learning Toolbox	For code generation, you can load the network by using the syntax `net = nasnetlarge` or by passing the `nasnetlarge` function to `coder.loadDeepLearningNetwork`. For example: `net = coder.loadDeepLearningNetwork('nasnetlarge')` For more information, see “Load Pretrained Networks for Code Generation” on page 4-15.

Name	Product	Usage Notes and Limitations
ne	MATLAB	Code generation does not support using ne to test inequality between an enumeration member and a string array, a character array, or a cell array of character arrays.
nextpow2	MATLAB	No known limitation
nnz	MATLAB	No known limitation
numel	MATLAB	No known limitation
ones	MATLAB	Dimensions must be real, nonnegative integers.
ordfilt2	Image Processing Toolbox	• GPU code generation requires the inputs to be bounded. If the input is of variable dimension, the software generates C code. • When generating code, the padopt argument must be a compile-time constant. • The generated GPU code is not optimized if the domain value that defines the neighborhood for the filtering operation is of size greater than 11x11. For better performance, consider setting the StackLimitPerThread option in the coder.gpuConfig object to Inf.
otsuthresh	Image Processing Toolbox	No known limitation
padarray	Image Processing Toolbox	• Input arrays of data type categorical are not supported. • When generating code, padarray supports only up to 3-D inputs. • The input arguments padval and direction must be compile-time constants.
pdist	Statistics and Machine Learning Toolbox™	• The supported distance input argument values (Distance) for optimized CUDA code are 'euclidean', 'squaredeuclidean', 'seuclidean', 'cityblock', 'minkowski', 'chebychev', 'cosine', 'correlation', 'hamming', and 'jaccard'. • Distance cannot be a custom distance function. • Distance must be a compile-time constant.
pdist2	Statistics and Machine Learning Toolbox	• The supported distance input argument values (Distance) for optimized CUDA code are 'euclidean', 'squaredeuclidean', 'seuclidean', 'cityblock', 'minkowski', 'chebychev', 'cosine', 'correlation', 'hamming', and 'jaccard'. • Distance cannot be a custom distance function. • Distance must be a compile-time constant. • Names in name-value pair arguments must be compile-time constants. • The sorted order of tied distances in the generated code can be different from the order in MATLAB due to numerical precision.

Name	Product	Usage Notes and Limitations
plus	MATLAB	If you use plus with single type and double type operands, the generated code might not produce the same result as MATLAB. See “Binary Element-Wise Operations with Single and Double Operands” (MATLAB Coder).
pointCloud	Computer Vision Toolbox	• GPU code generation for variable input sizes is not optimized. Consider using constant size inputs for an optimized code generation. • GPU code generation supports the 'Color', 'Normal', and 'Intensity' name-value pairs. • GPU code generation supports the findNearestNeighbors, findNeighborsInRadius, findPointsInROI, removeInvalidPoints, and select methods. • For very large inputs, the memory requirements of the algorithm may exceed the GPU device limits. In such cases, consider reducing the input size to proceed with code generation.
pol2cart	MATLAB	No known limitation
polyint	MATLAB	No known limitation
pow2	Fixed-Point Designer	No known limitation
power	MATLAB	• When both X and Y are real, but power(X,Y) is complex, simulation produces an error and generated code returns NaN. To get the complex result, make the input value X complex by passing in complex(X). For example, power(complex(X),Y). • When both X and Y are real, but X .^ Y is complex, simulation produces an error and generated code returns NaN. To get the complex result, make the input value X complex by using complex(X). For example, complex(X).^Y. • Code generation does not support sparse matrix inputs for this function.

<table>
<tr><th>Name</th><th>Product</th><th>Usage Notes and Limitations</th></tr>
<tr><td><code>predict</code></td><td>Deep Learning Toolbox</td><td>
GPU code generation supports the following syntaxes:

<code>YPred = predict(net,X)</code>
<code>[YPred1,...,YPredM] = predict(__)</code>
<code>YPred = predict(net,sequences)</code>
<code>__ = predict(__,Name,Value)</code>

The input X must not have variable size. The size must be fixed at code generation time.
GPU code generation for the <code>predict</code> function supports inputs that are defined as half-precision floating point data types. For more information, see <code>half</code>.
GPU code generation supports only vector sequences. The sequence length can be variable sized. The feature dimension must be fixed at code generation time.
Only the <code>'MiniBatchSize'</code>, <code>'SequenceLength'</code>, <code>'SequencePaddingDirection'</code>, and <code>'SequencePaddingValue'</code> name-value pair arguments are supported for code generation. All name-value pairs must be compile-time constants.
Only the <code>'longest'</code> and <code>'shortest'</code> option of the <code>'SequenceLength'</code> name-value pair is supported for code generation.
</td></tr>
<tr><td><code>predictAndUpdateState</code></td><td>Deep Learning Toolbox</td><td>
GPU code generation supports the following syntaxes:

<code>[updatedNet,YPred] = predictAndUpdateState(recNet,sequences)</code>
<code>[updatedNet,YPred] = predictAndUpdateState(__,Name,Value)</code>

GPU code generation for the <code>predictAndUpdateState</code> function is only supported for recurrent neural networks and cuDNN target library.
GPU code generation supports only vector sequences. The sequence length can be variable sized. The feature dimension must be fixed at code generation time.
Only the <code>'MiniBatchSize'</code>, <code>'SequenceLength'</code>, <code>'SequencePaddingDirection'</code>, and <code>'SequencePaddingValue'</code> name-value pair arguments are supported for code generation. All name-value pairs must be compile-time constants.
Only the <code>'longest'</code> and <code>'shortest'</code> option of the <code>'SequenceLength'</code> name-value pair is supported for code generation.
</td></tr>
<tr><td><code>prod</code></td><td>MATLAB</td><td>If you supply <code>dim</code>, it must be a constant.</td></tr>
<tr><td><code>projective2d</code></td><td>Image Processing Toolbox</td><td>When generating code, you can only specify singular objects—arrays of objects are not supported.</td></tr>
<tr><td><code>psnr</code></td><td>Image Processing Toolbox</td><td>No known limitation</td></tr>
</table>

Name	Product	Usage Notes and Limitations
`qr`	MATLAB	Code generation does not support sparse matrix inputs for this function.
`rad2deg`	MATLAB	No known limitation
`rank`	MATLAB	Code generation does not support sparse matrix inputs for this function.
`resetState`	Deep Learning Toolbox	GPU code generation for the `resetState` function is only supported for recurrent neural networks and cuDNN target library.
`rcond`	MATLAB	Code generation does not support sparse matrix inputs for this function.
`rdivide`	MATLAB	If you use `rdivide` with single type and double type operands, the generated code might not produce the same result as MATLAB. See “Binary Element-Wise Operations with Single and Double Operands” (MATLAB Coder).
`real`	MATLAB	No known limitation
`reallog`	MATLAB	No known limitation
`realsqrt`	MATLAB	No known limitation
`rectint`	MATLAB	No known limitation
`repelem`	MATLAB	The input must be a vector or matrix. The input cannot be a multidimensional array.
`repmat`	MATLAB	• Size arguments must have a fixed size. • For sparse matrices, the `repmat` function does not support trailing ones as inputs after the first two dimensions.
`reshape`	MATLAB	• If the input is a compile-time empty cell array, then the size arguments must be constants. • Size arguments must have a fixed size. • For sparse matrices, the `reshape` function does not support trailing ones as inputs after the first two dimensions.
`resnet18`	Deep Learning Toolbox	• For code generation, you can load the network by using the syntax `net = resnet18` or by passing the `resnet18` function to `coder.loadDeepLearningNetwork`. For example: `net = coder.loadDeepLearningNetwork('resnet18')` For more information, see “Load Pretrained Networks for Code Generation” on page 4-15. • The syntax `resnet18('Weights','none')` is not supported for GPU code generation.

Name	Product	Usage Notes and Limitations
`resnet50`	Deep Learning Toolbox	• For code generation, you can load the network by using the syntax `net = resnet50` or by passing the `resnet50` function to `coder.loadDeepLearningNetwork`. For example: `net = coder.loadDeepLearningNetwork('resnet50')` For more information, see "Load Pretrained Networks for Code Generation" on page 4-15. • The syntax `resnet50('Weights','none')` is not supported for GPU code generation.
`resnet101`	Deep Learning Toolbox	• For code generation, you can load the network by using the syntax `net = resnet101` or by passing the `resnet101` function to `coder.loadDeepLearningNetwork`. For example: `net = coder.loadDeepLearningNetwork('resnet101')` For more information, see "Load Pretrained Networks for Code Generation" on page 4-15. • The syntax `resnet101('Weights','none')` is not supported for GPU code generation.
`rgb2gray`	MATLAB	No known limitation
`rgb2hsv`	MATLAB	No known limitation
`rgb2lab`	Image Processing Toolbox	When generating code, all character vector input arguments must be compile-time constants.
`rot90`	MATLAB	Does not support cell arrays for the first argument.
`round`	MATLAB	• Code generation supports only the syntax `Y = round(X)`. • Code generation does not support `char` or `logical` data types for `X`.
`sec`	MATLAB	No known limitation
`sech`	MATLAB	No known limitation
`segnetLayers`	Computer Vision Toolbox	For code generation, you must first create a SegNet network by using the `segnetLayers` function. Then, use the `trainNetwork` function on the resulting `lgraph` object to train the network for segmentation. Once the network is trained and evaluated, you can generate code for the `DAGNetwork` object using GPU Coder.
`selectStrongestBboxMulticlass`	Computer Vision Toolbox	• Code generation is only supported for numeric `labels`. • Code generation is not supported for rotated rectangle bounding box inputs.
`SeriesNetwork`	Deep Learning Toolbox	• Only the `activations`, `classify`, `predict`, `predictAndUpdateState`, and `resetState` object functions are supported. • To create a `SeriesNetwork` object for code generation, see "Load Pretrained Networks for Code Generation" on page 4-15.
`sin`	MATLAB	No known limitation
`single`	MATLAB	No known limitation

Name	Product	Usage Notes and Limitations
sinh	MATLAB	No known limitation
size	MATLAB	No known limitation
sortrows	MATLAB	• The first input argument must not be a cell array. • If A is complex with all zero imaginary parts, then MATLAB might convert A to real(A) before calling sortrows(A). In this case, MATLAB sorts the rows of A by real(A), but the generated code sorts the rows of A by abs(A). To make the generated code match MATLAB, use sortrows(real(A)) or sortrows(A,'ComparisonMethod','real').
sph2cart	MATLAB	No known limitation
sqrt	MATLAB	Simulation produces an error. Generated standalone code returns NaN when the input value x is real, but the output should be complex. To get the complex result, make the input value complex by passing in complex(x).
squeeze	MATLAB	Does not support cell arrays.
squeezenet	Deep Learning Toolbox	• For code generation, you can load the network by using the syntax net = squeezenet or by passing the squeezenet function to coder.loadDeepLearningNetwork. For example: net = coder.loadDeepLearningNetwork('squeezenet'). For more information, see "Load Pretrained Networks for Code Generation" on page 4-15. • The syntax squeezenet('Weights','none') is not supported for GPU code generation.
ssdObjectDetector	Computer Vision Toolbox	• Only the detect method of the ssdObjectDetector is supported for code generation. • The bounding box output from code generation may have small numerical differences with the simulation results from MATLAB. • The code generator resizes the input image size to the detect method to that of the input layer size of the network. However, the bounding boxes output generated is with reference to the original input size. • The roi argument to the detect method must be a codegen constant (coder.const()) and a 1x4 vector. • Only the Threshold, SelectStrongest, MinSize, MaxSize, and MiniBatchSize Name-Value pairs are supported. All name-value pair must be compile time constant. • The channel and batch size of the input image must be fixed size. • The labels output is returned as a categorical array.
std	MATLAB	If you specify dim, then it must be a constant.
stft	Signal Processing Toolbox	• The 'ConjugateSymmetric' argument is not supported for code generation. • Timetables are not supported for code generation.

Name	Product	Usage Notes and Limitations
`stretchlim`	Image Processing Toolbox	No known limitation
`sub2ind`	MATLAB	• The first argument must be a valid size vector. Code generation does not support size vectors for arrays with more than `intmax` elements. • The generated code treats `NaN` inputs as out of range and throws a run-time error.
`subsasgn`	Fixed-Point Designer	No known limitation
`subsindex`	MATLAB	No known limitation
`subsref`	Fixed-Point Designer	No known limitation
`sum`	MATLAB	• If you specify `dim`, then it must be a constant. • The `outtype` and `nanflag` options must be constant character vectors.
`superpixels`	Image Processing Toolbox	• All character vector inputs must be compile-time constants. • The value of `'IsInputLab'` (`true` or `false`) must be a compile-time constant.
`svd`	MATLAB	• Code generation uses a different `SVD` implementation than MATLAB uses. Because the singular value decomposition is not unique, left and right singular vectors might differ from those computed by MATLAB. • When the input matrix contains a nonfinite value, the generated code does not issue an error. Instead, the output contains `NaN` values. • Code generation does not support sparse matrix inputs for this function.
`swapbytes`	MATLAB	Inheritance of the class of the input to `swapbytes` in a MATLAB Function block is supported only when the class of the input is `double`. For non-double inputs, the input port data types must be specified, not inherited.
`tan`	MATLAB	No known limitation
`tanh`	MATLAB	No known limitation
`times`	MATLAB	• Multiplication of pure imaginary numbers by non-finite numbers might not match MATLAB. The code generator does not specialize multiplication by pure imaginary numbers—it does not eliminate calculations with the zero real part. For example, `(Inf + 1i)*1i = (Inf*0 – 1*1) + (Inf*1 + 1*0)i = NaN + Infi`. • If you use `times` with single type and double type operands, the generated code might not produce the same result as MATLAB.
`trace`	MATLAB	Code generation does not support sparse matrix inputs for this function.
`transpose`	MATLAB	No known limitation

Name	Product	Usage Notes and Limitations
`tril`	MATLAB	If you supply the argument that represents the order of the diagonal matrix, then it must be a real and scalar integer value.
`triu`	MATLAB	If you supply the argument that represents the order of the diagonal matrix, then it must be a real and scalar integer value.
`true`	MATLAB	Dimensions must be real, nonnegative, integers.
`typecast`	MATLAB	• The value of the data type argument must be lowercase. • When you use `typecast` with inherited input port data types in MATLAB Function blocks, the software can throw a size error. To avoid this error, specify the block input port data types explicitly. • Integer input or result classes must map directly to a C type on the target hardware. • The input must be a variable-length vector or a fixed-size vector. • The output vector always has the same orientation as the input vector.
`uint8`, `uint16`, `uint32`, `uint64`	MATLAB	No known limitation
`uminus`	MATLAB	No known limitation
`unetLayers`	Computer Vision Toolbox	You can use the U-Net network for code generation. First, create the network using the `unetLayers` function. Then, use the `trainNetwork` function to train the network for segmentation. After training and evaluating the network, you can generate code for the `DAGNetwork` object by using GPU Coder.
`uplus`	MATLAB	No known limitation
`vander`	MATLAB	No known limitation
`var`	MATLAB	If specified, `dim` must be a constant.
`vertcat`	Fixed-Point Designer	No known limitation
`vgg16`	Deep Learning Toolbox	• For code generation, you can load the network by using the syntax `net = vgg16` or by passing the `vgg16` function to `coder.loadDeepLearningNetwork`. For example: `net = coder.loadDeepLearningNetwork('vgg16')` For more information, see "Load Pretrained Networks for Code Generation" on page 4-15. • The syntax `vgg16('Weights','none')` is not supported for GPU code generation.

Name	Product	Usage Notes and Limitations
vgg19	Deep Learning Toolbox	• For code generation, you can load the network by using the syntax `net = vgg19` or by passing the `vgg19` function to `coder.loadDeepLearningNetwork`. For example: `net = coder.loadDeepLearningNetwork('vgg19')` For more information, see “Load Pretrained Networks for Code Generation” on page 4-15. • The syntax `vgg19('Weights','none')` is not supported for GPU code generation.
watershed	Image Processing Toolbox	• Supports only 2-D images • Supports only `4` or `8` connectivity • Supports images containing up to 65,535 regions • Output type is always `uint16`
xception	Deep Learning Toolbox	• For code generation, you can load the network by using the syntax `net = xception` or by passing the `xception` function to `coder.loadDeepLearningNetwork`. For example: `net = coder.loadDeepLearningNetwork('xception')` For more information, see “Load Pretrained Networks for Code Generation” on page 4-15. • The syntax `xception('Weights','none')` is not supported for GPU code generation.
xor	MATLAB	No known limitation
ycbcr2rgb	Image Processing Toolbox	No known limitation
yolov2Layers	Computer Vision Toolbox	For code generation, you must first create a YOLO v2 network by using the `yolov2Layers` function. Then, use the `trainYOLOv2ObjectDetector` function on the resulting `lgraph` object to train the network for object detection. Once the network is trained and evaluated, you can generate code for the `yolov2ObjectDetector` object using GPU Coder.
yolov2Object Detector	Computer Vision Toolbox	• Only the `detect` method of the `yolov2ObjectDetector` is supported for code generation. • The `roi` argument to the `detect` method must be a codegen constant (`coder.const()`) and a 1x4 vector. • Only the `Threshold`, `SelectStrongest`, `MinSize`, `MaxSize`, and `MiniBatchSize` Name-Value pairs are supported. • The height, width, channel, and batch size of the input image must be fixed size. • The minimum batch size value passed to detect method must be fixed size. • The labels output is returned as a cell array of character vectors, such as {'car','bus'}.
zeros	MATLAB	Dimensions must be nonnegative real integers.

2

Kernel Creation

Kernels from Element-Wise Loops

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

Element-Wise Math Example

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

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

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

Preparing myFun for Code Generation

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

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

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

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

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

Generated CUDA Code

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

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

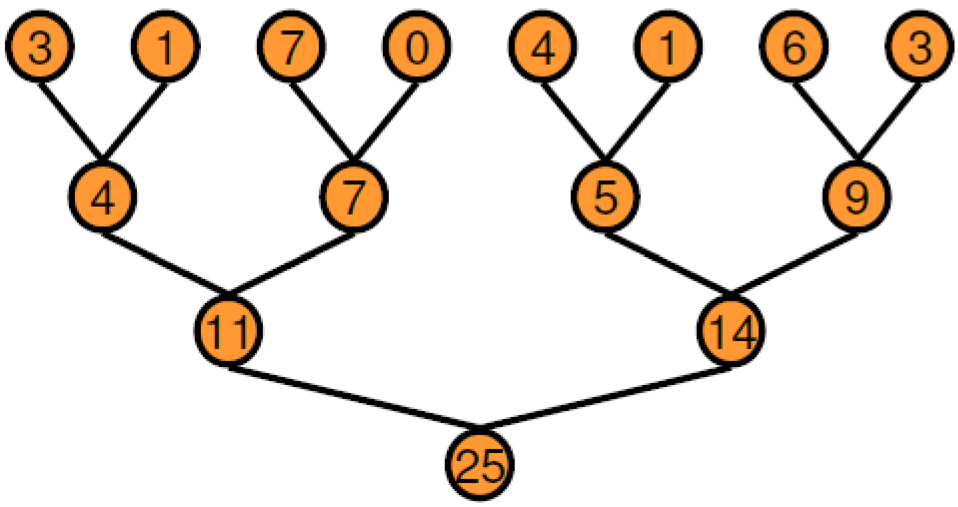
- “Design Patterns” on page 2-22
- “Kernels from Scatter-Gather Type Operations” on page 2-4
- “Kernels from Library Calls” on page 2-8
- “Legacy Code Integration” on page 2-18

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 threadId;
  uint32_T threadStride;
  uint32_T thdBlkId;
  uint32_T idx;
  real_T tmpRed;
  ;
  ;
  thdBlkId = (threadIdx.z * blockDim.x * blockDim.y + threadIdx.y * blockDim.x)
    + threadIdx.x;
  threadId = ((gridDim.x * gridDim.y * blockIdx.z + gridDim.x * blockIdx.y) +
              blockIdx.x) * (blockDim.x * blockDim.y * blockDim.z) + thdBlkId;
  threadStride = gridDim.x * blockDim.x * (gridDim.y * blockDim.y) * (gridDim.z *
    blockDim.z);
  if (!((int32_T)threadId >= 512)) {
    tmpRed = 0.0;
    for (idx = threadId; threadStride < 0U ? idx >= 511U : idx <= 511U; idx +=
         threadStride) {
      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. TheNVIDIA TensorRT is a high performance deep learning inference optimizer and runtime library. For more information, see “Code Generation for Deep Learning Networks by Using cuDNN” on page 4-17 and “Code Generation for Deep Learning Networks by Using TensorRT” on page 4-26.

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
`mtimes`	Matrix multiply	Yes	Yes
`mldivide (‘\’)`	Solve system of linear equation Ax=B for x	Yes	Yes
`lu`	LU matrix factorization	Yes	Yes
`qr`	Orthogonal-triangular decomposition	Yes	Partial
`det`	Matrix determinant	Yes	Yes

MATLAB Function	Description	MATLAB Coder LAPACK Support	cuBLAS, cuSOLVER, cuFFT, Thrust Support
`inv`	Matrix inverse	Yes	Yes
`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 `gpucoder.sort`

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` | `gpucoder.matrixMatrixKernel` | `gpucoder.sort` | `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
- "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);
  alpha1 = 1.0;
  beta1 = 0.0;
  cudaMemcpy((void *)gpu_alpha1, (void *)&alpha1, 8ULL, cudaMemcpyHostToDevice);
  cudaMemcpy((void *)gpu_A, (void *)A, 8388608ULL, cudaMemcpyHostToDevice);
  cudaMemcpy((void *)gpu_B, (void *)B, 8388608ULL, cudaMemcpyHostToDevice);
  cudaMemcpy(gpu_beta1, &beta1, 8ULL, cudaMemcpyHostToDevice);
  cublasDgemm(cublasGlobalHandle, CUBLAS_OP_N, CUBLAS_OP_N, 1024, 1024, 1024,
             (double *)gpu_alpha1, (double *)&gpu_A[0], 1024, (double *)&gpu_B
              [0], 1024, (double *)gpu_beta1, (double *)&gpu_C[0], 1024);
  cublasEnsureDestruction();
  cudaMemcpy((void *)C, (void *)gpu_C, 8388608ULL, cudaMemcpyDeviceToHost);
```

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

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

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

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

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

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

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

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

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

Prepare blas_gemm for Kernel Creation

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

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

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

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

---

cuSOLVER Example

This example solves the systems of linear equations `Ax = B` for `x` by using the cuSOLVER library. The matrices `A` and `B` must have the same number of rows. If `A` is a scalar, then `A\B` is equivalent to `A.\B`. If `A` is a square n-by-n matrix and `B` is a matrix with n rows, then `x = A\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);
blackslash_kernel1<<<dim3(1U, 1U, 1U), dim3(160U, 1U, 1U)>>>(b_gpu_A,gpu_A);
cudaMemcpy(b_A, gpu_A, 1152UL, cudaMemcpyDeviceToHost);
cusolverDnDgetrf_bufferSize(cusolverGlobalHandle, 12, 12, &gpu_A[0], 12,
  &cusolverWorkspaceReq);
cusolverWorkspaceTypeSize = 8;
```

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

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

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

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

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

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

```
P*A = L*U
```

where A is an m×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';
```

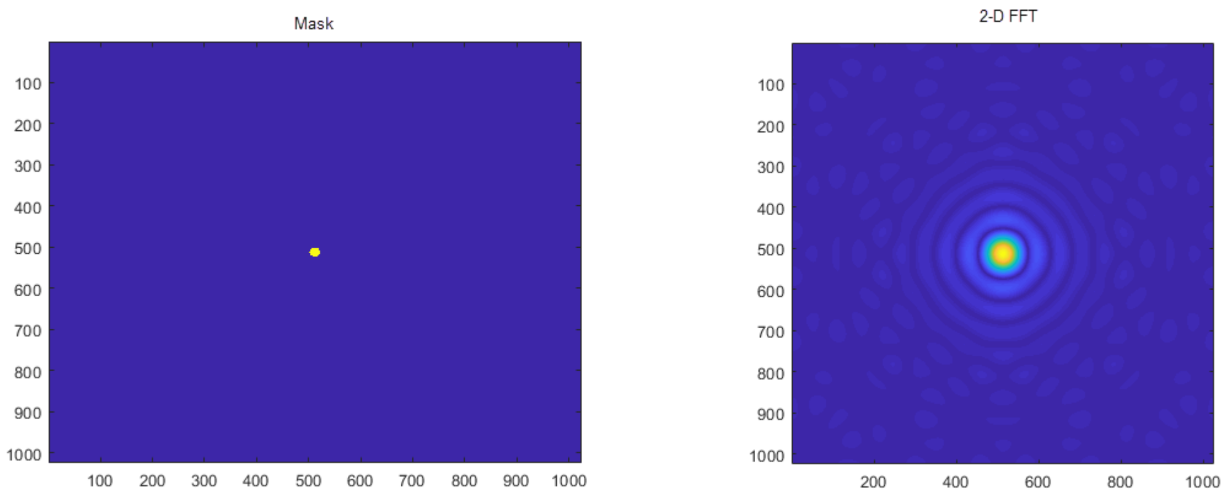
```
cfg.GenerateExampleMain = 'GenerateCodeAndCompile';

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” (MATLAB Coder).

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

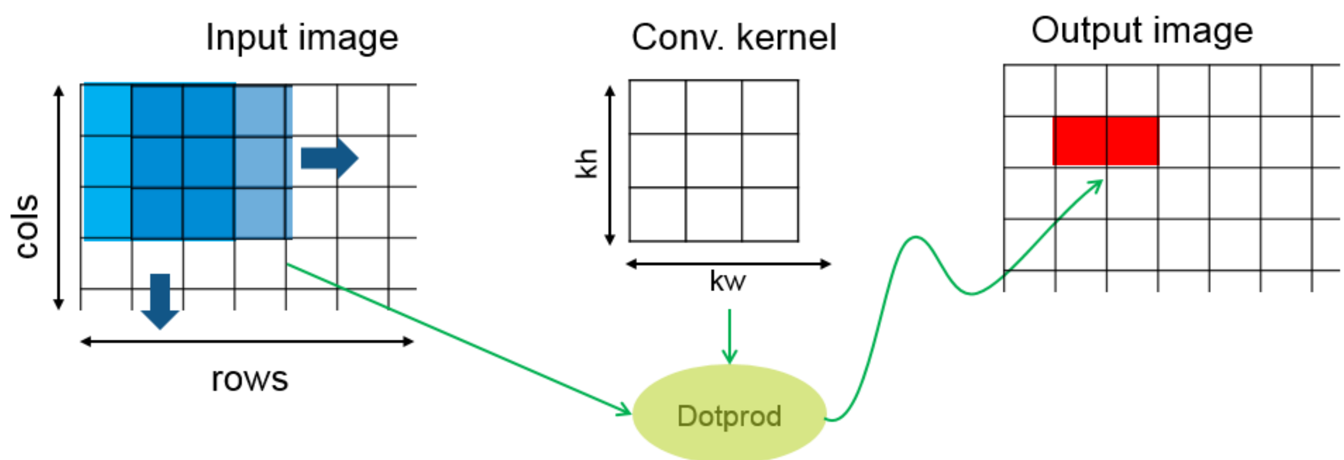
- “Design Patterns” on page 2-22
- “Kernels from Element-Wise Loops” on page 2-2
- “Kernels from Scatter-Gather Type Operations” on page 2-4
- “Kernels from Library Calls” on page 2-8

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 `gpucoder.stencilKernel` function and create CUDA code for stencil functions. For an example that demonstrates stencil preocessing, see “Stencil Processing on GPU”.

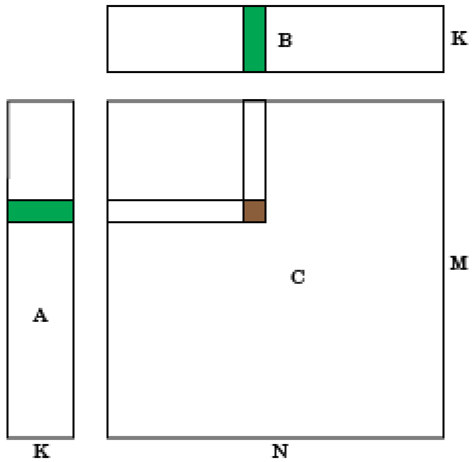
For very large input sizes, the `gpucoder.stencilKernel` function may produce CUDA code that does not numerically match the MATLAB simulation. In such cases, consider reducing the size of the input to produce accurate results..

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

See Also

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

Related Examples

- “Stencil Processing on GPU”

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.

<table>
<tr><th>Original Code</th><th>Optimized Generated Code</th></tr>
<tr><td><pre>A(:) = …
…
for i = 1:N
   gB = kernel1(gA);
   gA = kernel2(gB);

   if (somecondition)
      gC = kernel3(gA, gB);
   end
   …
end
…
… = C;</pre></td><td><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></td></tr>
</table>

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 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

Troubleshooting

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

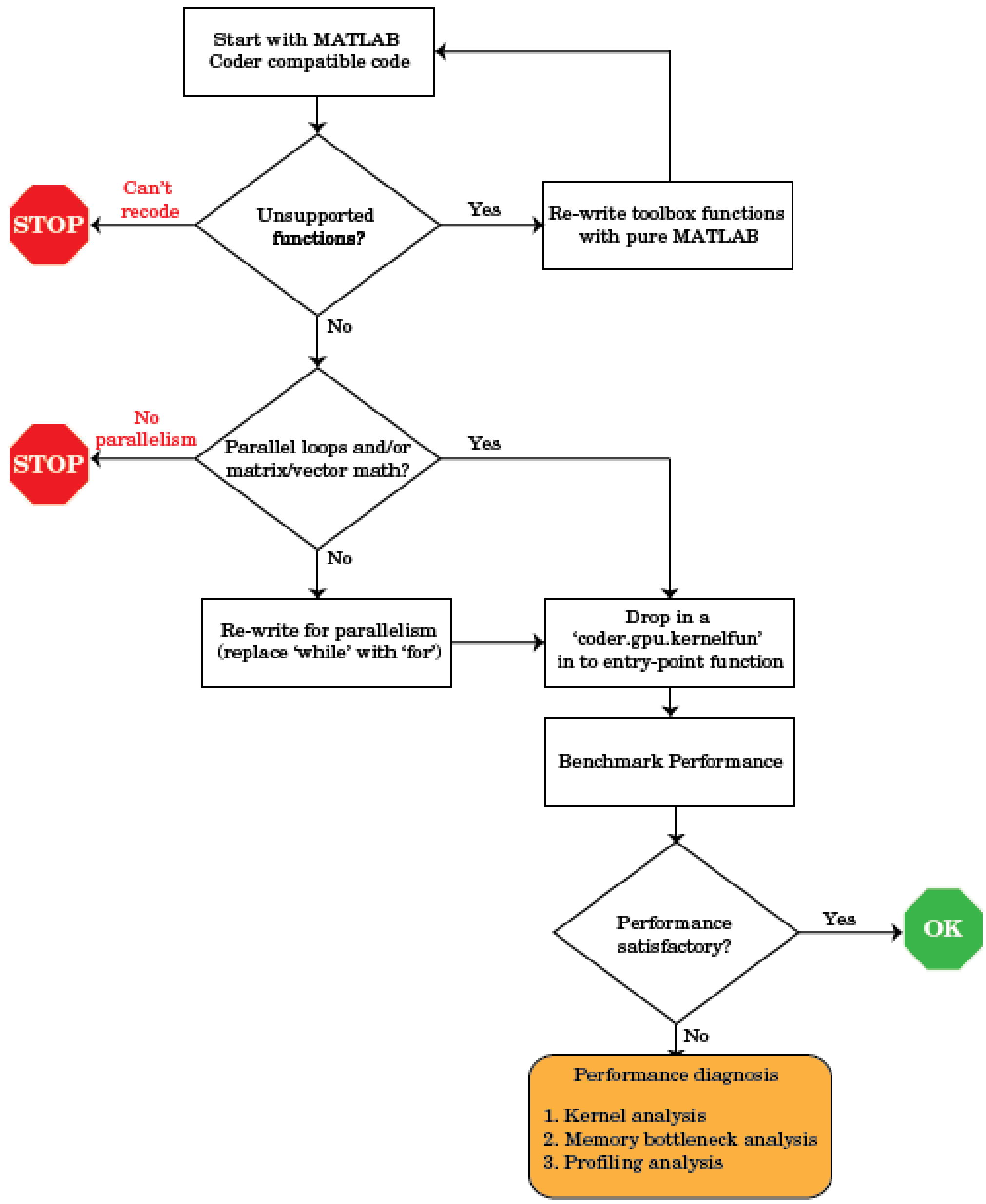
- CUDA kernels are not created.
- Host to device and device to host memory transfers (`cudaMemcpy`) are throttling performance.
- Not enough parallelism or device issues.

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” (MATLAB Coder).
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.kernelfun` 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 3-5
- "Kernel Analysis" on page 3-18

- “Memory Bottleneck Analysis” on page 3-22
- “Analyze Execution Profiles of the Generated Code” on page 3-24

Code Generation Reports

In this section...

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:

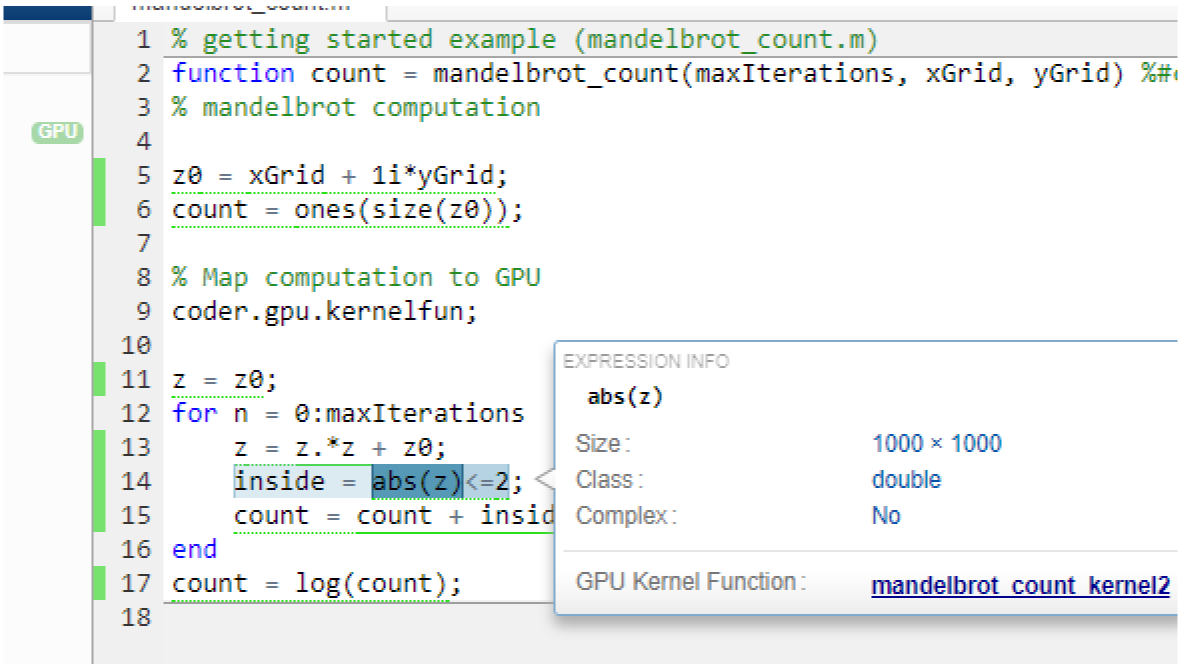
MATLAB Source

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.

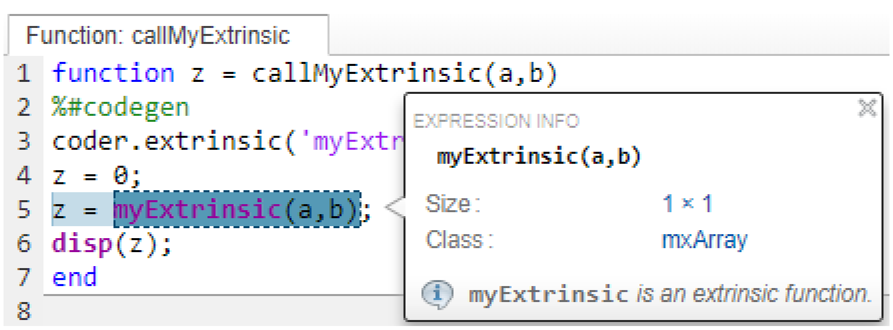
CUDA Kernels

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.



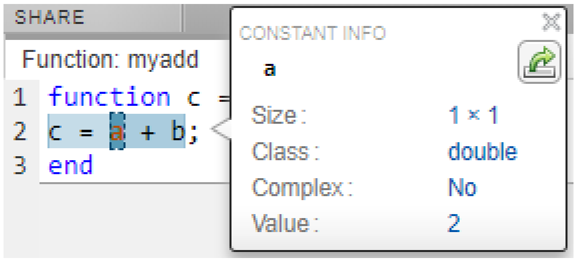
Extrinsic Functions

In the MATLAB code, the report identifies an extrinsic function with purple text. The information window indicates that the function is extrinsic.



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

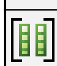
This table describes symbols, badges, and other indicators in the variables table.

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” (MATLAB Coder).

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 3-11.

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” (MATLAB Coder).
- 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” (MATLAB Coder).

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 3-15
- “Trace Between Generated CUDA Code and MATLAB Source Code” on page 3-11
- “Interactively Trace Between MATLAB Code and Generated C/C++ Code” (Embedded Coder)
- “Row-Major and Column-Major Array Layouts” (MATLAB Coder)

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 $k \to \infty$.

$$z_{k+1} = z_k^2 + z_0, \quad k = 0, 1, \ldots$$

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
  ```

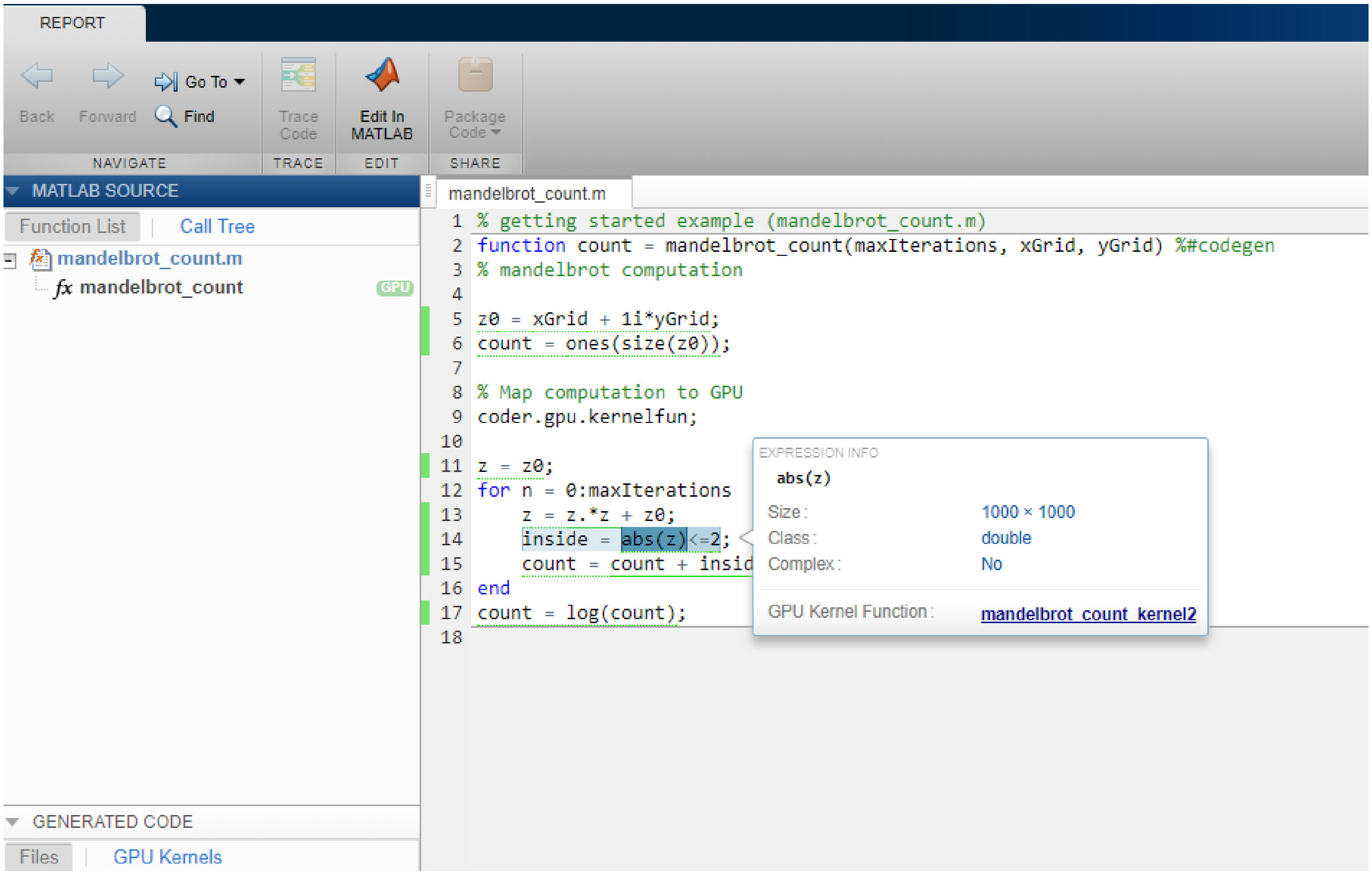
Note The `--fmad=false` flag when passed to the `nvcc`, instructs the compiler to disable Floating-Point Multiply-Add (FMAD) optimization. This option is set to prevent numerical mismatch in the generated code because of architectural differences in the CPU and the GPU. For more information, see “Numerical 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 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 3-5
- "Generating a GPU Code Metrics Report for Code Generated from MATLAB Code" on page 3-15

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 CUDAcode and produce a code generation report. See “Code Generation Reports” on page 3-5.

By default, static code metrics analysis does not run at code generation time. Instead, if and when you want to run the analysis and view the results, click **GPU Code Metrics** on the **Summary** tab of the code generation report.

Example GPU Code Metrics Report

This example runs GPU static code metrics analysis and examines a static code metrics report.

Create a MATLAB function called `mandelbrot_count.m` with the following lines of code. This code is a vectorized MATLAB implementation of the Mandelbrot set. For every point `(xGrid,yGrid)` in the grid, it calculates the iteration index `count` at which the trajectory defined by the equation reaches a distance of 2 from the origin. It then returns the natural logarithm of `count`, which is used generate the color coded plot of the Mandelbrot set.

```
function count = mandelbrot_count(maxIterations,xGrid,yGrid)
% Add kernelfun pragma to trigger kernel creation
coder.gpu.kernelfun;
% mandelbrot computation

z0 = xGrid + 1i*yGrid;
count = ones(size(z0));

z = z0;
for n = 0:maxIterations
    z = z.*z + z0;
    inside = abs(z)<=2;
    count = count + inside;
end
count = log(count);
```

Create sample data with the following lines of code. The code generates a 1000 x 1000 grid of real parts (*x*) and imaginary parts (*y*) between the limits specified by `xlim` and `ylim`.

```
maxIterations = 500;
gridSize = 1000;
xlim = [-0.748766713922161,-0.748766707771757];
ylim = [0.123640844894862,0.123640851045266];

x = linspace(xlim(1),xlim(2),gridSize);
y = linspace(ylim(1),ylim(2),gridSize);
[xGrid,yGrid] = meshgrid(x,y);
```

Enable production of a code generation report by using a configuration object for standalone code generation (static library, dynamically linked library, or executable program).

```
cfg = coder.gpuConfig('dll');
cfg.GenerateReport = true;
```

```
cfg.MATLABSourceComments = true;
cfg.GpuConfig.CompilerFlags = '--fmad=false';
```

Note The `--fmad=false` flag when passed to the `nvcc`, instructs the compiler to disable Floating-Point Multiply-Add (FMAD) optimization. This option is set to prevent numerical mismatch in the generated code because of architectural differences in the CPU and the GPU. For more information, see “Numerical Differences Between CPU and GPU”.

Alternatively, use the `codegen -report` option.

Generate code by using `codegen`. Specify the type of the input argument by providing an example input with the `-args` option. Specify the configuration object by using the `-config` option.

```
codegen -config cfg -args {maxIterations,xGrid,yGrid} mandelbrot_count
```

To open the code generation report, click **View report**.

To run the static code metrics analysis and view the code metrics report, on the **Summary** tab of the code generation report, click **GPU Code Metrics**.

Explore the code metrics report

1 To see the information on the generated CUDA kernels, click **CUDA Kernels**.

1. CUDA Kernels [hide]

Kernel Name	Thread Dimensions	Block Dimensions	Input Variables	Output Variables	Stream	Shared Memory Size	Minimum BlocksPerSM	Constant Memory	Parent Kernel
mandelbrot_count_kernel3	[512,1,1]	[1954,1,1]		gpu_count	0	0	1	0	None
mandelbrot_count_kernel2	[512,1,1]	[1954,1,1]	gpu_z0	gpu_count,gpu_z	0	0	1	0	None
mandelbrot_count_kernel1	[512,1,1]	[1954,1,1]	gpu_yGrid,gpu_xGrid	gpu_z,gpu_count,gpu_z0	0	0	1	0	None

- **Kernel Name** contains the list of generated CUDA kernels. By default, GPU Coder prepends the kernel name with the name of the entry-point function.
- **Thread Dimensions** is an array of the form `[Tx,Ty,Tz]` that identifies the number of threads in the block along dimensions `x`, `y`, and `z`.
- **Block Dimensions** is an array of the form `[Bx,By,1]` is an array that defines the number of blocks in the grid along dimensions `x` and `y` (`z` not used).
- **Shared Memory Size** and **Constant Memory** columns provide metrics on the shared and constant memory space usage in the generated code.
- **Minimum BlocksPerSM** is the minimum number of blocks per streaming multiprocessor and indicates the number of blocks with which to launch the kernels.

To navigate from the report to the generated kernel code, click a kernel name.

2 To see the variables that have memory allocated on the GPU device, go to the **CUDA Malloc** section.

2. CUDA Malloc [hide]

Variable Name	Data Size
gpu_yGrid	8000000
gpu_xGrid	8000000
gpu_z	16000000
gpu_count	8000000
gpu_z0	16000000

3 To view information on the `cudaMemCpy` calls in the generated code, click **CUDA Memcpy**.

4. CUDA Memcpy [hide]

Destination Variable Name	Source Variable Name	Data Size	Direction	Conditional Variable	Stream
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 3-5
- “Interactively Trace Between MATLAB Code and Generated C/C++ Code” (Embedded Coder)
- “Trace Between Generated CUDA Code and MATLAB Source Code” on page 3-11
- “Code Generation Using the Command Line Interface”
- “Code Generation by Using the GPU Coder App”

Kernel Analysis

In this section...

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 3-5
- “Trace Between Generated CUDA Code and MATLAB Source Code” on page 3-11
- “Generating a GPU Code Metrics Report for Code Generated from MATLAB Code” on page 3-15
- “Memory Bottleneck Analysis” on page 3-22
- “Analyze Execution Profiles of the Generated Code” on page 3-24

Memory Bottleneck Analysis

In this section...

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 3-5
- "Trace Between Generated CUDA Code and MATLAB Source Code" on page 3-11
- "Generating a GPU Code Metrics Report for Code Generated from MATLAB Code" on page 3-15
- "Kernel Analysis" on page 3-18
- "Analyze Execution Profiles of the Generated Code" on page 3-24

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.

Create a Design File

For this example create a entry-point function that performs N-D fast Fourier transform. Use the `coder.gpu.kernelfun` pragma to map the FFT to the GPU. By default, the `EnableCUFFT` property is enabled, so the code generator uses cuFFT library to perform the FFT operation.

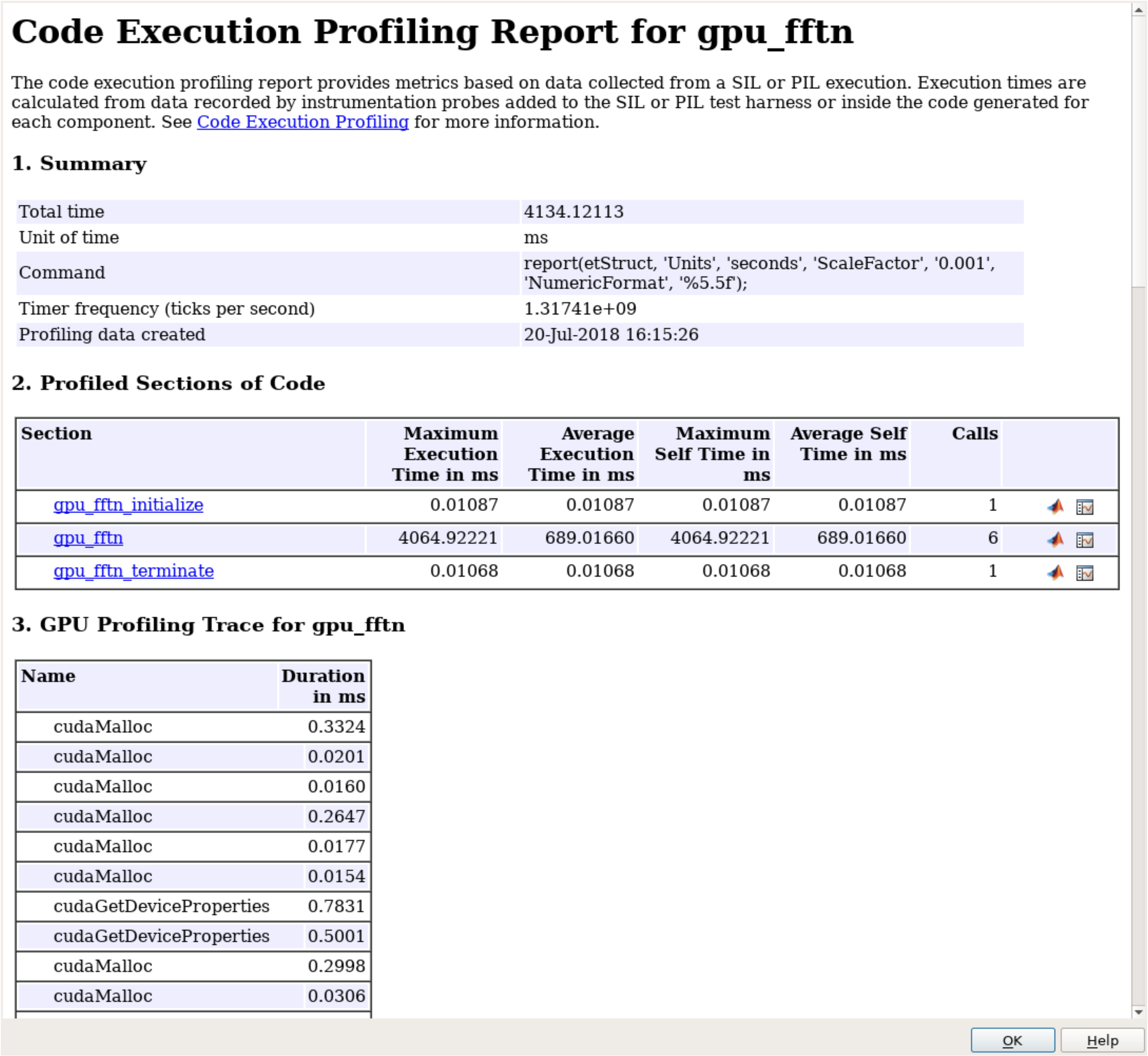
```
function [Y] = gpu_fftn(X)
  coder.gpu.kernelfun();
  Y = fftn(X);
end
```

Generate the Execution Profiling Report

Use the `gpucoder.profile` function to generate the execution profiling report.

```
cfg = coder.gpuConfig('exe');
cfg.GpuConfig.MallocMode = 'discrete';
gpucoder.profile('gpu_fftn',{rand(2,4500,4)},'CodegenConfig',cfg, ...
'CodegenArguments','-d profilingdir','Threshold',0.001)
```

The code execution profiling report opens. This report provides metrics based on data collected from a SIL execution. Execution times are calculated from data recorded by instrumentation probes added to the SIL test harness or inside the code generated for each component. See “View Execution Times” (Embedded Coder) for more information.



Code Execution Profiling Report for gpu_fftn

The code execution profiling report provides metrics based on data collected from a SIL or PIL execution. Execution times are calculated from data recorded by instrumentation probes added to the SIL or PIL test harness or inside the code generated for each component. See Code Execution Profiling for more information.

1. Summary

Total time	4134.12113
Unit of time	ms
Command	report(etStruct, 'Units', 'seconds', 'ScaleFactor', '0.001', 'NumericFormat', '%5.5f');
Timer frequency (ticks per second)	1.31741e+09
Profiling data created	20-Jul-2018 16:15:26

2. Profiled Sections of Code

Section	Maximum Execution Time in ms	Average Execution Time in ms	Maximum Self Time in ms	Average Self Time in ms	Calls	
gpu_fftn_initialize	0.01087	0.01087	0.01087	0.01087	1	
gpu_fftn	4064.92221	689.01660	4064.92221	689.01660	6	
gpu_fftn_terminate	0.01068	0.01068	0.01068	0.01068	1	

3. GPU Profiling Trace for gpu_fftn

Name	Duration in ms
cudaMalloc	0.3324
cudaMalloc	0.0201
cudaMalloc	0.0160
cudaMalloc	0.2647
cudaMalloc	0.0177
cudaMalloc	0.0154
cudaGetDeviceProperties	0.7831
cudaGetDeviceProperties	0.5001
cudaMalloc	0.2998
cudaMalloc	0.0306

OK Help

See Also

`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 3-5
- "Trace Between Generated CUDA Code and MATLAB Source Code" on page 3-11

- “Generating a GPU Code Metrics Report for Code Generated from MATLAB Code” on page 3-15

Analysis with NVIDIA Profiler

In this section...

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 3-5
- "Trace Between Generated CUDA Code and MATLAB Source Code" on page 3-11
- "Generating a GPU Code Metrics Report for Code Generated from MATLAB Code" on page 3-15
- "Kernel Analysis" on page 3-18
- "Memory Bottleneck Analysis" on page 3-22
- "Analyze Execution Profiles of the Generated Code" on page 3-24

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 1xN 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 3-5
- “Trace Between Generated CUDA Code and MATLAB Source Code” on page 3-11
- “Generating a GPU Code Metrics Report for Code Generated from MATLAB Code” on page 3-15
- “Kernel Analysis” on page 3-18
- “Memory Bottleneck Analysis” on page 3-22
- “Analyze Execution Profiles of the Generated Code” on page 3-24

Deep Learning

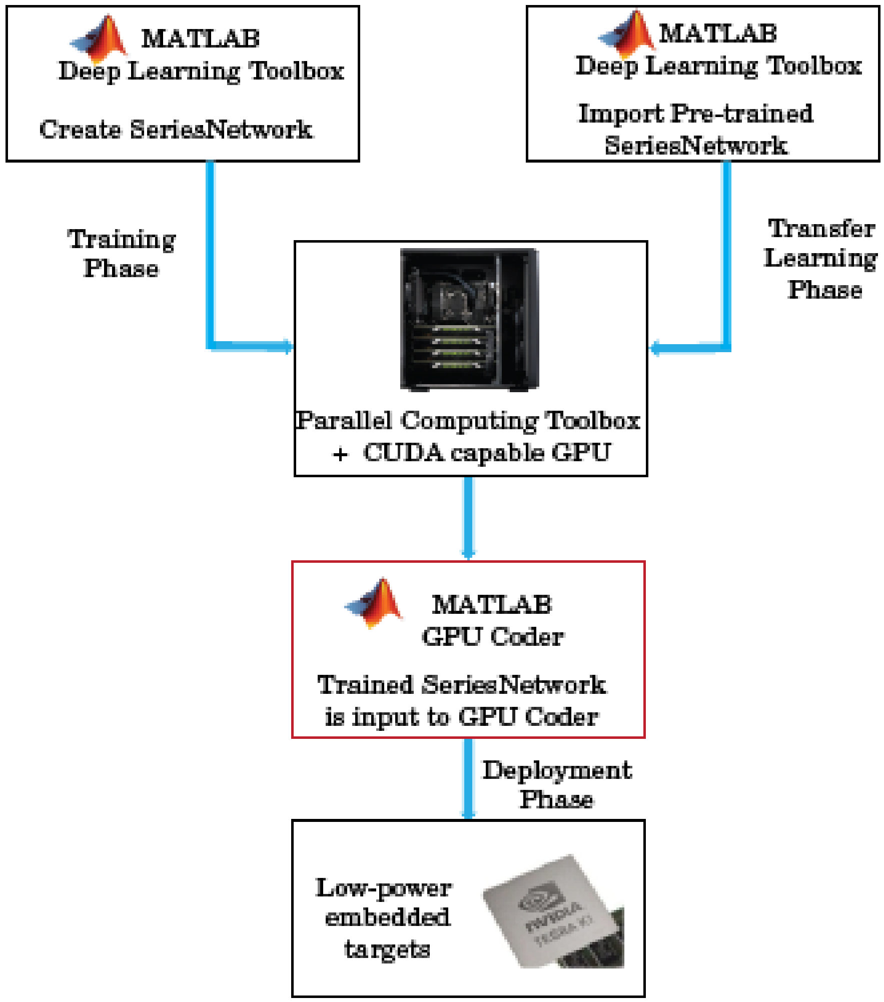
Workflow

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

codegen | coder.CodeConfig | coder.CuDNNConfig | coder.DeepLearningConfig | coder.EmbeddedCodeConfig | coder.getDeepLearningLayers | 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 and Layers” on page 4-4
- “Load Pretrained Networks for Code Generation” on page 4-15
- “Code Generation for Deep Learning Networks by Using cuDNN” on page 4-17
- “Code Generation for Deep Learning Networks by Using TensorRT” on page 4-26
- “Code Generation for Deep Learning Networks Targeting ARM Mali GPUs” on page 4-36

Supported Networks and Layers

Supported Pretrained Networks

GPU Coder supports code generation for series and directed acyclic graph (DAG) convolutional neural networks (CNNs or ConvNets). You can generate code for any trained convolutional neural network whose layers are supported for code generation. See “Supported Layers” on page 4-6. 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 `alexnet`. The syntax `alexnet('Weights','none')` is not supported for code generation.	Yes	Yes	Yes
`GoogLeNet`	GoogLeNet convolutional neural network. For the pretrained GoogLeNet model, see `googlenet`. The syntax `googlenet('Weights','none')` 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 `importCaffeNetwork`.	Yes	Yes	Yes
Darknet-19	Darknet-19 convolutional neural network. For more information, see `darknet19`. The syntax `darknet19('Weights','none')` is not supported for code generation.	Yes	Yes	Yes
Darknet-53	Darknet-53 convolutional neural network. for more information, see `darknet53`. The syntax `darknet53('Weights','none')` is not supported for code generation.	Yes	Yes	Yes
DeepLab v3+	DeepLab v3+ convolutional neural network. For more information, see `deeplabv3plusLayers`.	Yes	Yes	No

Network Name	Description	cuDNN	TensorRT	ARM Compute Library for Mali GPU
DenseNet-201	DenseNet-201 convolutional neural network. For the pretrained DenseNet-201 model, see `densenet201`. The syntax `densenet201('Weights','none')` is not supported for code generation.	Yes	Yes	Yes
Inception-v3	Inception-v3 convolutional neural network. For the pretrained Inception-v3 model, see `inceptionv3`. The syntax `inceptionv3('Weights','none')` is not supported for code generation.	Yes	Yes	Yes
Inception-ResNet-v2	Inception-ResNet-v2 convolutional neural network. For the pretrained Inception-ResNet-v2 model, see `inceptionresnetv2`.	Yes	Yes	No
Mobilenet-v2	MobileNet-v2 convolutional neural network. For the pretrained MobileNet-v2 model, see `mobilenetv2`. The syntax `mobilenetv2('Weights','none')` is not supported for code generation.	Yes	Yes	Yes
NASNet-Large	NASNet-Large convolutional neural network. For the pretrained NASNet-Large model, see `nasnetlarge`.	Yes	Yes	No
NASNet-Mobile	NASNet-Mobile convolutional neural network. For the pretrained NASNet-Mobile model, see `nasnetmobile`.	Yes	Yes	No
`ResNet`	ResNet-18, ResNet-50, and ResNet-101 convolutional neural networks. For the pretrained ResNet models, see `resnet50`, `resnet18`, and `resnet101`. The syntax `resnetXX('Weights','none')` is not supported for code generation.	Yes	Yes	Yes
`SegNet`	Multi-class pixelwise segmentation network. For more information, see `segnetLayers`.	Yes	Yes	No

Network Name	Description	cuDNN	TensorRT	ARM Compute Library for Mali GPU
SqueezeNet	Small deep neural network. For the pretrained SqueezeNet models, see squeezenet. The syntax squeezenet('Weights','none') is not supported for code generation.	Yes	Yes	Yes
VGG-16	VGG-16 convolutional neural network. For the pretrained VGG-16 model, see vgg16. The syntax vgg16('Weights','none') is not supported for code generation.	Yes	Yes	Yes
VGG-19	VGG-19 convolutional neural network. For the pretrained VGG-19 model, see vgg19. The syntax vgg19('Weights','none') is not supported for code generation.	Yes	Yes	Yes
Xception	Xception convolutional neural network. For the pretrained Xception model, see xception. The syntax xception('Weights','none') 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 yolov2Layers	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.

Once you install the support package GPU Coder Interface for Deep Learning Libraries, you can use `coder.getDeepLearningLayers` to see a list of the layers supported for a specific deep learning library. For example, `coder.getDeepLearningLayers('cudnn')` shows the list of layers supported for code generation by using the NVIDIA cuDNN library.

Input Layers

Layer Name	Product	Description	cu
imageInputLayer	Deep Learning Toolbox	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.	Ye
sequenceInputLayer	Deep Learning Toolbox	A sequence input layer inputs sequence data to a network. For code generation, only vector input types are supported. 2-D and 3-D image sequence input is not supported. Code generation does not support `'Normalization'` specified using a function handle.	Ye

Convolution and Fully Connected Layers

Layer Name	Product	Description	cu
convolution2dLayer	Deep Learning Toolbox	A 2-D convolutional layer applies sliding convolutional filters to the input.	Ye
groupedConvolution2dLayer	Deep Learning Toolbox	A 2-D grouped convolutional layer separates the input channels into groups and applies sliding convolutional filters. Use grouped convolutional layers for channel-wise separable (also known as depth-wise separable) convolution. Code generation for the ARM Mali GPU is not supported for a 2-D grouped convolution layer that has the `NumGroups` property set as `'channel-wise'` or a value greater than two.	Ye
transposedConv2dLayer	Deep Learning Toolbox	A transposed 2-D convolution layer upsamples feature maps.	Ye
fullyConnectedLayer	Deep Learning Toolbox	A fully connected layer multiplies the input by a weight matrix and then adds a bias vector.	Ye

Sequence Layers

Layer Name	Product	Description	cu
`sequenceInputLayer`	Deep Learning Toolbox	A sequence input layer inputs sequence data to a network. For code generation, only vector input types are supported. 2-D and 3-D image sequence input is not supported. Code generation does not support `'Normalization'` specified using a function handle.	Ye
`lstmLayer`	Deep Learning Toolbox	An LSTM layer learns long-term dependencies between time steps in time series and sequence data. For code generation, the `StateActivationFunction` property must be set to `'tanh'`. For code generation, the `GateActivationFunction` property must be set to `'sigmoid'`.	Ye
`bilstmLayer`	Deep Learning Toolbox	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. For code generation, the `StateActivationFunction` property must be set to `'tanh'`. For code generation, the `GateActivationFunction` property must be set to `'sigmoid'`.	Ye
`flattenLayer`	Deep Learning Toolbox	A flatten layer collapses the spatial dimensions of the input into the channel dimension.	Ye
`wordEmbeddingLayer`	Text Analytics Toolbox™	A word embedding layer maps word indices to vectors.	Ye

Activation Layers

Layer Name	Product	Description	cu
reluLayer	Deep Learning Toolbox	A ReLU layer performs a threshold operation to each element of the input, where any value less than zero is set to zero.	Ye
leakyReluLayer	Deep Learning Toolbox	A leaky ReLU layer performs a threshold operation, where any input value less than zero is multiplied by a fixed scalar.	Ye
clippedReluLayer	Deep Learning Toolbox	A clipped ReLU layer performs a threshold operation, where any input value less than zero is set to zero and any value above the *clipping ceiling* is set to that clipping ceiling.	Ye
eluLayer	Deep Learning Toolbox	An ELU activation layer performs the identity operation on positive inputs and an exponential nonlinearity on negative inputs.	Ye
tanhLayer	Deep Learning Toolbox	A hyperbolic tangent (tanh) activation layer applies the tanh function on the layer inputs.	Ye

Normalization, Dropout, and Cropping Layers

Layer Name	Product	Description	cu
batchNormalizationLayer	Deep Learning Toolbox	A batch normalization layer normalizes each input channel across a mini-batch.	Ye
crossChannelNormalizationLayer	Deep Learning Toolbox	A channel-wise local response (cross-channel) normalization layer carries out channel-wise normalization.	Ye
dropoutLayer	Deep Learning Toolbox	A dropout layer randomly sets input elements to zero with a given probability.	Ye
crop2dLayer	Deep Learning Toolbox	A 2-D crop layer applies 2-D cropping to the input.	Ye

Pooling and Unpooling Layers

Layer Name	Product	Description	cu
averagePooling2dLayer	Deep Learning Toolbox	An average pooling layer performs down-sampling by dividing the input into rectangular pooling regions and computing the average values of each region.	Ye

Layer Name	Product	Description	cu
globalAveragePooling2dLayer	Deep Learning Toolbox	A global average pooling layer performs down-sampling by computing the mean of the height and width dimensions of the input.	Ye
maxPooling2dLayer	Deep Learning Toolbox	A max pooling layer performs down-sampling by dividing the input into rectangular pooling regions, and computing the maximum of each region.	Ye
globalMaxPooling2dLayer	Deep Learning Toolbox	A global max pooling layer performs down-sampling by computing the maximum of the height and width dimensions of the input.	Ye
maxUnpooling2dLayer	Deep Learning Toolbox	A max unpooling layer unpools the output of a max pooling layer.	Ye

Combination Layers

Layer Name	Product	Description	cu
additionLayer	Deep Learning Toolbox	An addition layer adds inputs from multiple neural network layers element-wise.	Ye
depthConcatenationLayer	Deep Learning Toolbox	A depth concatenation layer takes inputs that have the same height and width and concatenates them along the third dimension (the channel dimension).	Ye
concatenationLayer	Deep Learning Toolbox	A concatenation layer takes inputs and concatenates them along a specified dimension.	Ye

Object Detection Layers

Layer Name	Product	Description	c
anchorBoxLayer	Computer Vision Toolbox	An anchor box layer stores anchor boxes for a feature map used in object detection networks.	Y
ssdMergeLayer	Computer Vision Toolbox	An SSD merge layer merges the outputs of feature maps for subsequent regression and classification loss computation.	Y
YOLOv2OutputLayer	Computer Vision Toolbox	Create output layer for YOLO v2 object detection network.	Y
YOLOv2ReorgLayer	Computer Vision Toolbox	Create reorganization layer for YOLO v2 object detection network.	Y

Layer Name	Product	Description	c
`YOLOv2TransformLayer`	Computer Vision Toolbox	Create transform layer for YOLO v2 object detection network.	Y

Output Layers

Layer Name	Product	Description	c
`softmaxLayer`	Deep Learning Toolbox	A softmax layer applies a softmax function to the input.	Y
`classificationLayer`	Deep Learning Toolbox	A classification layer computes the cross entropy loss for multi-class classification problems with mutually exclusive classes.	Y
`regressionLayer`	Deep Learning Toolbox	A regression layer computes the half-mean-squared-error loss for regression problems.	Y
`pixelClassificationLayer`	Computer Vision Toolbox	A pixel classification layer provides a categorical label for each image pixel or voxel.	Y
`dicePixelClassificationLayer`	Computer Vision Toolbox	A Dice pixel classification layer provides a categorical label for each image pixel or voxel using generalized Dice loss.	Y
`Output Layer`	Deep Learning Toolbox	All output layers including custom classification or regression output layers created by using `nnet.layer.ClassificationLayer` or `nnet.layer.RegressionLayer`. 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). 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).	Y

Keras and ONNX Layers

Layer Name	Product	Description	c
`nnet.keras.layer.FlattenCStyleLayer`	Deep Learning Toolbox	Flatten activations into 1-D assuming C-style (row-major) order.	Y
`nnet.keras.layer.GlobalAveragePooling2dLayer`	Deep Learning Toolbox	Global average pooling layer for spatial data.	Y

Layer Name	Product	Description	c
nnet.keras.layer.SigmoidLayer	Deep Learning Toolbox	Sigmoid activation layer.	Y
nnet.keras.layer.TanhLayer	Deep Learning Toolbox	Hyperbolic tangent activation layer.	Y
nnet.keras.layer.ZeroPadding2dLayer	Deep Learning Toolbox	Zero padding layer for 2-D input.	Y
nnet.onnx.layer.ElementwiseAffineLayer	Deep Learning Toolbox	Layer that performs element-wise scaling of the input followed by an addition.	Y
nnet.onnx.layer.FlattenLayer	Deep Learning Toolbox	Flattens the spatial dimensions of the input tensor to the channel dimensions.	Y
nnet.onnx.layer.IdentityLayer	Deep Learning Toolbox	Layer that implements ONNX identity operator.	Y

Supported Classes

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

Name	Product	Description	c
yolov2ObjectDetector	Computer Vision Toolbox	Detect objects using YOLO v2 object detector • Only the detect method of the yolov2ObjectDetector is supported for code generation. • The roi argument to the detect method must be a codegen constant (coder.const()) and a 1x4 vector. • Only the Threshold, SelectStrongest, MinSize, MaxSize, and MiniBatchSize Name-Value pairs are supported. • The height, width, channel, and batch size of the input image must be fixed size. • The minimum batch size value passed to detect method must be fixed size. • The labels output is returned as a cell array of character vectors, such as {'car','bus'}.	Y

Name	Product	Description	c
ssdObjectDetector	Computer Vision Toolbox	Object to detect objects using the SSD-based detector. • Only the detect method of the ssdObjectDetector is supported for code generation. • The roi argument to the detect method must be a codegen constant (coder.const()) and a 1x4 vector. • Only the Threshold, SelectStrongest, MinSize, MaxSize, and MiniBatchSize Name-Value pairs are supported. All Name-Value pairs must be compile-time constants. • The channel and batch size of the input image must be fixed size. • The labels output is returned as a categorical array. • In the generated code, the input is rescaled to the size of the input layer of the network. But the bounding box that the detect method returns is in reference to the original input size. • The bounding boxes might not numerically match the simulation results.	Y

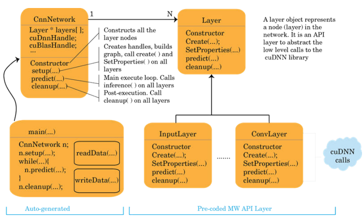
See Also
codegen | coder.CodeConfig | coder.CuDNNConfig | coder.DeepLearningConfig | coder.EmbeddedCodeConfig | coder.getDeepLearningLayers | coder.gpuConfig | coder.gpuEnvConfig

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 4-15
- “Code Generation for Deep Learning Networks by Using cuDNN” on page 4-17
- “Code Generation for Deep Learning Networks by Using TensorRT” on page 4-26
- “Code Generation for Deep Learning Networks Targeting ARM Mali GPUs” on page 4-36

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 and Layers” on page 4-4
- “Load Pretrained Networks for Code Generation” on page 4-15
- “Code Generation for Deep Learning Networks by Using cuDNN” on page 4-17
- “Code Generation for Deep Learning Networks by Using TensorRT” on page 4-26
- “Code Generation for Deep Learning Networks Targeting ARM Mali GPUs” on page 4-36

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`, or `ssdObjectDetector` 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
```

If you generate code by using `cnncodegen`, load the network object in the MATLAB workspace. Then, pass the object to `cnncodegen`. For example:

```
net = squeezenet;
cnncodegen(net,'targetlib','cudnn');
```

See Also

`DAGNetwork` | `SeriesNetwork` | `cnncodegen` | `codegen` | `coder.loadDeepLearningNetwork` | `ssdObjectDetector` | `trainNetwork` | `yolov2ObjectDetector`

More About

- "Supported Networks and Layers" on page 4-4
- "Code Generation for Deep Learning Networks by Using cuDNN" on page 4-17
- "Code Generation for Deep Learning Networks by Using TensorRT" on page 4-26
- "Code Generation for Deep Learning Networks Targeting ARM Mali GPUs" on page 4-36

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 `cnncodegen` command that generates CUDA code and builds a static library for the specified network object.
- The GPU Coder app that generates CUDA code from a MATLAB entry-point function.

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, the `cnncodegen` command, and the GPU Coder app.

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. To install the support packages, select the support package from the MATLAB **Add-Ons** menu.
4. CUDA toolkit and cuDNN libraries. 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

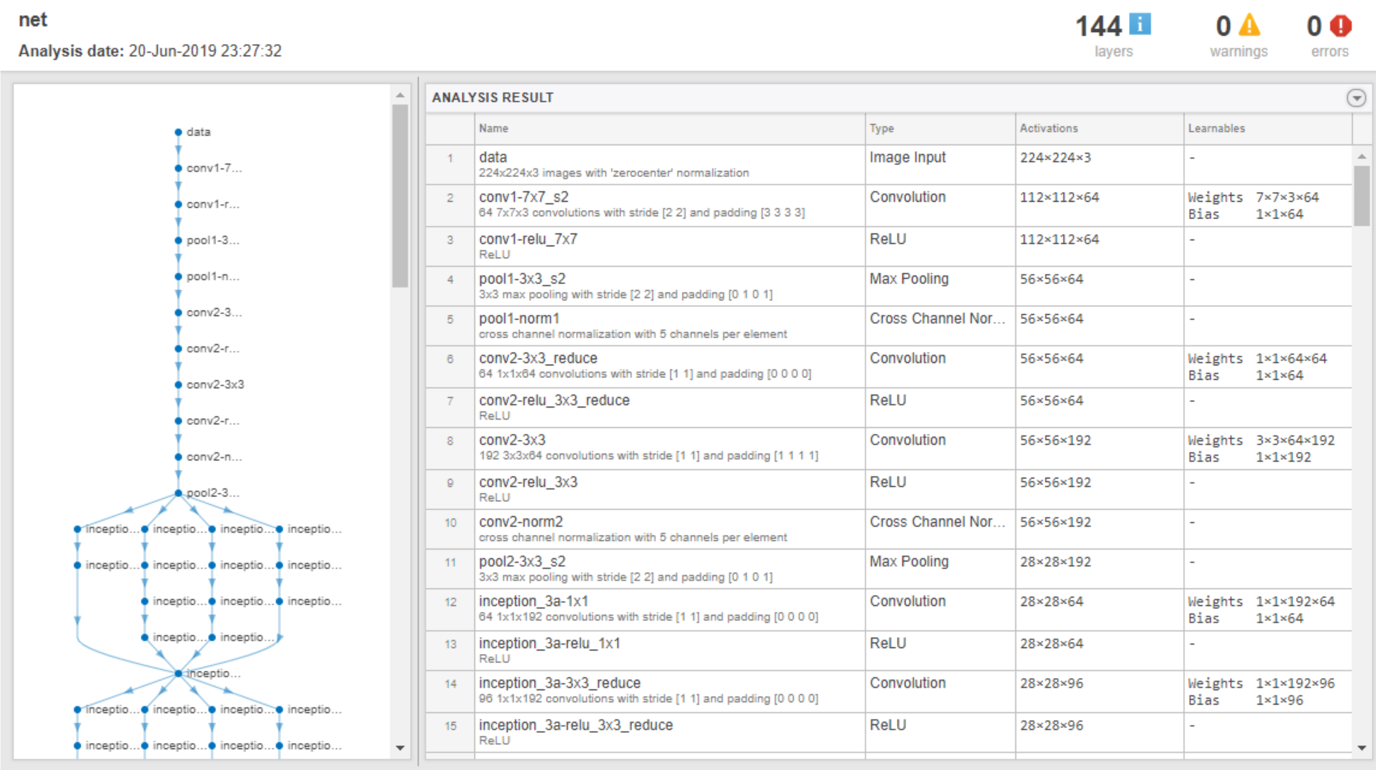
1. Load the pretrained GoogLeNet network. You can choose to load a different pretrained network for image classification. If you do not have the required support packages installed, the software provides a download link.

   ```
   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);
```



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

3 The image that you want to classify must have the same size as the input size of the network. For GoogLeNet, the size of the `imageInputLayer` is 224-by-224-by-3. The `Classes` property of the output `classificationLayer` contains the names of the classes learned by the network. View 10 random class names out of the total of 1000.

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

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

For more information, see “List of Deep Learning Layers” (Deep Learning Toolbox).

Create an Entry-Point Function

1 Write an entry-point function in MATLAB that:

a. Uses the `coder.loadDeepLearningNetwork` function to load a deep learning model and to construct and set up a CNN class. For more information, see "Load Pretrained Networks for Code Generation" on page 4-15.
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.

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 tensor data type input to the network 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.

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” (MATLAB Coder).

```
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 *inputData;
  real32_T *outputData;
  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);
}
```

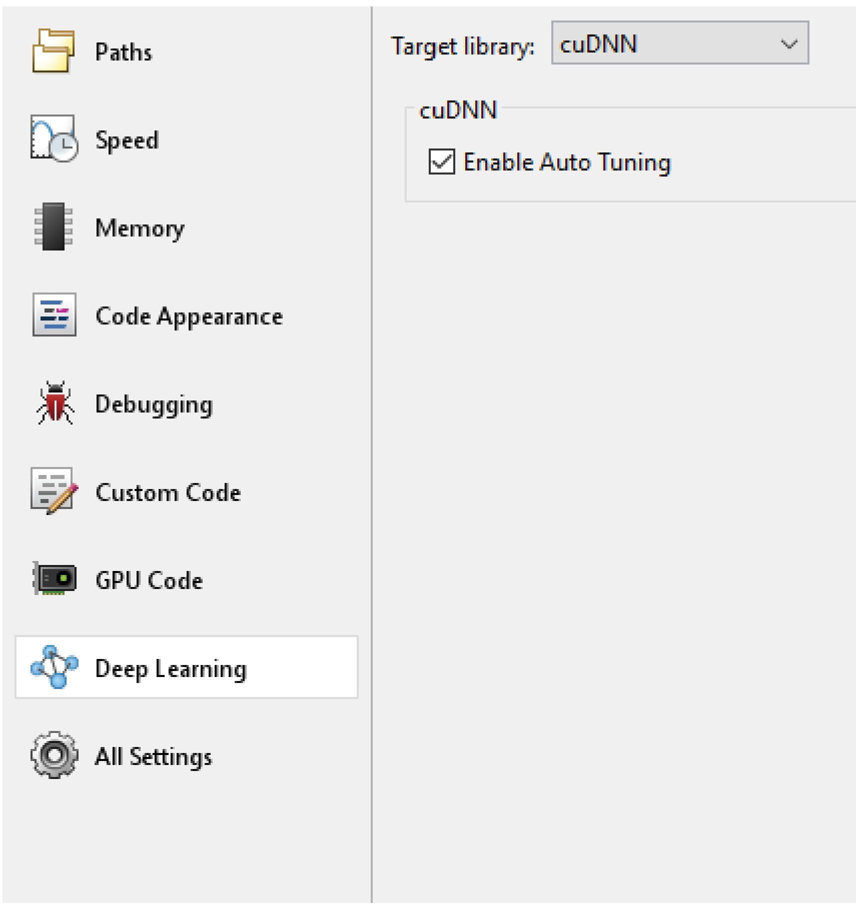
```
void googlenet_predict_init()
{
  mynet_not_empty = false;
}
```

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**.

Code Generation by Using cnncodegen

To generate code with the cuDNN library, you can use the `targetlib` option of the `cnncodegen` command. The `cnncodegen` command generates CUDA code and builds a static library for the given `SeriesNetwork` or `DAGNetwork` object.

1 Load the pretrained network. For more information, see “Load Pretrained Networks for Code Generation” on page 4-15.

2 Call `cnncodegen` with `'targetlib'` specified as `'cudnn'`. For example:

```
net = googlenet;
cnncodegen(net,'targetlib','cudnn');
```

The `cnncodegen` command generates code, a makefile, `cnnbuild_rtw.mk`, and builds the library file `cnnbuild`. The command places all the generated files in the `codegen` folder.

Generated Code

The DAG network is generated as a C++ class (`CnnMain`) 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 `cnn_exec.hpp` file is shown.

`cnn_exec.hpp` File

```
class CnnMain
{
  public:
    int32_T batchSize;
    int32_T numLayers;
    real32_T *inputData;
    real32_T *outputData;
    MWCNNLayer *layers[78];
  private:
    MWTargetNetworkImpl *targetImpl;
  public:
    void presetup();
    void allocate();
    void postsetup();
    CnnMain();
    void setup();
    void deallocate();
    void predict();
    void cleanup();
    real32_T *getInputDataPointer();
    real32_T *getOutputDataPointer();
    real32_T *getLayerOutput(int32_T layerIndex, int32_T portIndex);
    ~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 78 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 `FusedConvReLU` 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.

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.

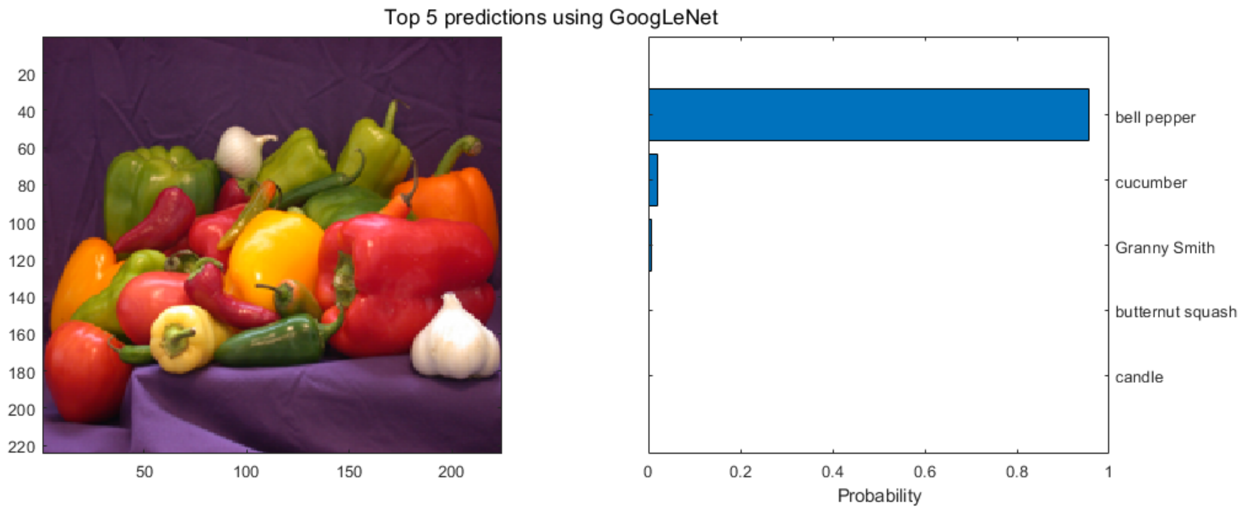
Run the Generated MEX

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(I,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

`cnncodegen` | `codegen` | `coder.CuDNNConfig` | `coder.loadDeepLearningNetwork`

More About

- “Supported Networks and Layers” on page 4-4
- “Load Pretrained Networks for Code Generation” on page 4-15
- “Code Generation for Deep Learning Networks by Using TensorRT” on page 4-26
- “Code Generation for Deep Learning Networks”
- “Code Generation for Object Detection by Using YOLO v2”
- “Deployment and Classification of Webcam Images on NVIDIA Jetson TX2 Platform”
- “Generated CNN Class Hierarchy” on page 4-14

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 `cnncodegen` command that generates CUDA code and builds a static library for the specified network object.
- The GPU Coder app that generates CUDA code from a MATLAB entry-point function.

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, the `cnncodegen` 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” example.

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. To install the support packages, select the support package from the MATLAB **Add-Ons** menu.
4. CUDA toolkit, cuDNN, and TensorRT libraries. 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”.

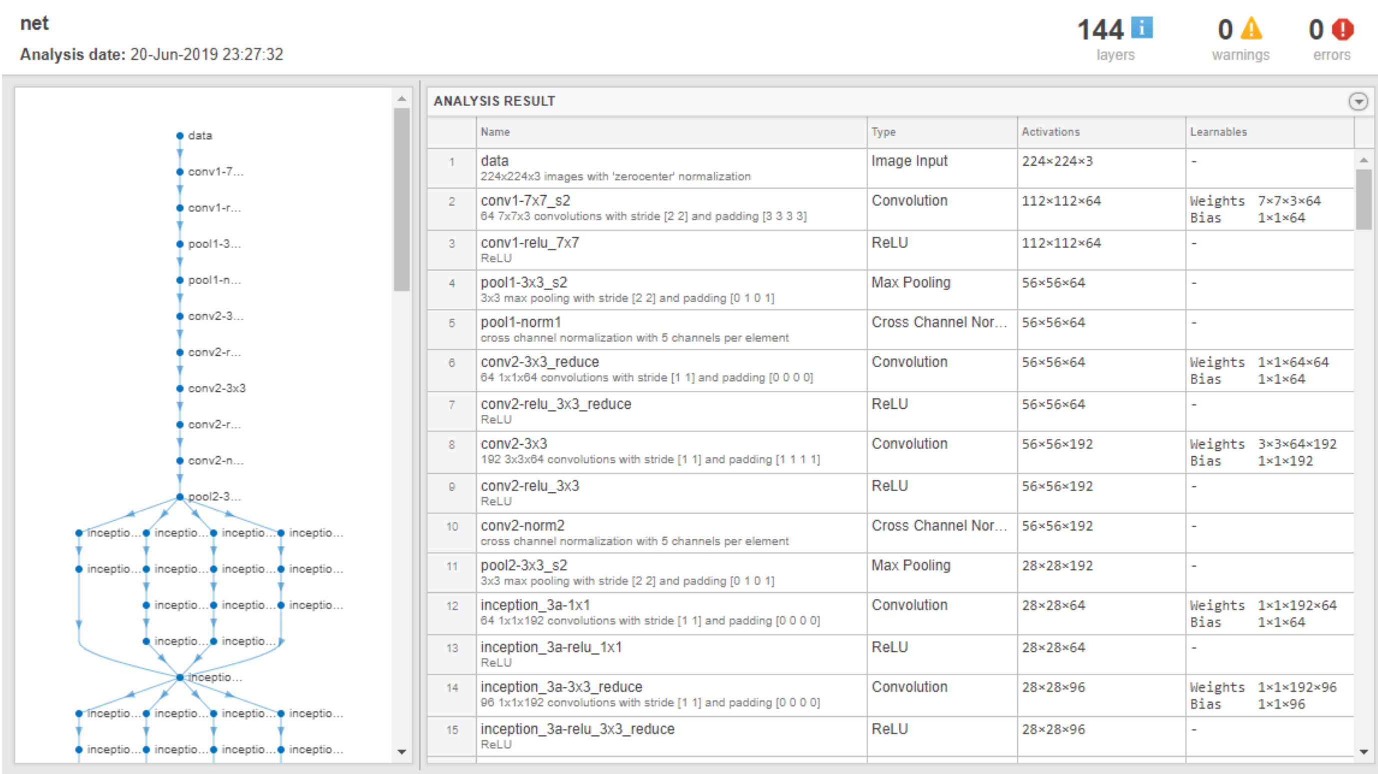
Load Pretrained Network

1 Load the pretrained GoogLeNet network. You can choose to load a different pretrained network for image classification. If you do not have the required support packages installed, the software provides a download link.

```
net = googlenet;
```

2 The object `net` contains the `DAGNetwork` object. Use the `analyzeNetwork` function to display an interactive visualization of the network architecture, to detect errors and issues in the network, and to display detailed information about the network layers. The layer information includes the sizes of layer activations and learnable parameters, the total number of learnable parameters, and the sizes of state parameters of recurrent layers.

```
analyzeNetwork(net);
```



3 The image that you want to classify must have the same size as the input size of the network. For GoogLeNet, the size of the `imageInputLayer` is 224-by-224-by-3. The `Classes` property of the output `classificationLayer` contains the names of the classes learned by the network. View 10 random class names out of the total of 1000.

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

```
    'speedboat'
    'window screen'
    'isopod'
    'wooden spoon'
    'lipstick'
    'drake'
```

```
'hyena'
'dumbbell'
'strawberry'
'custard apple'
```

For more information, see “List of Deep Learning Layers” (Deep Learning Toolbox).

Create an Entry-Point Function

1 Write an entry-point function in MATLAB that:

   a Uses the `coder.loadDeepLearningNetwork` function to load a deep learning model and to construct and set up a CNN class. For more information, see “Load Pretrained Networks for Code Generation” on page 4-15.

   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.

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 tensor data type input to the network or the tensor output of a layer 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.

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” 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" (MATLAB Coder).

```
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 *inputData;
  real32_T *outputData;
  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

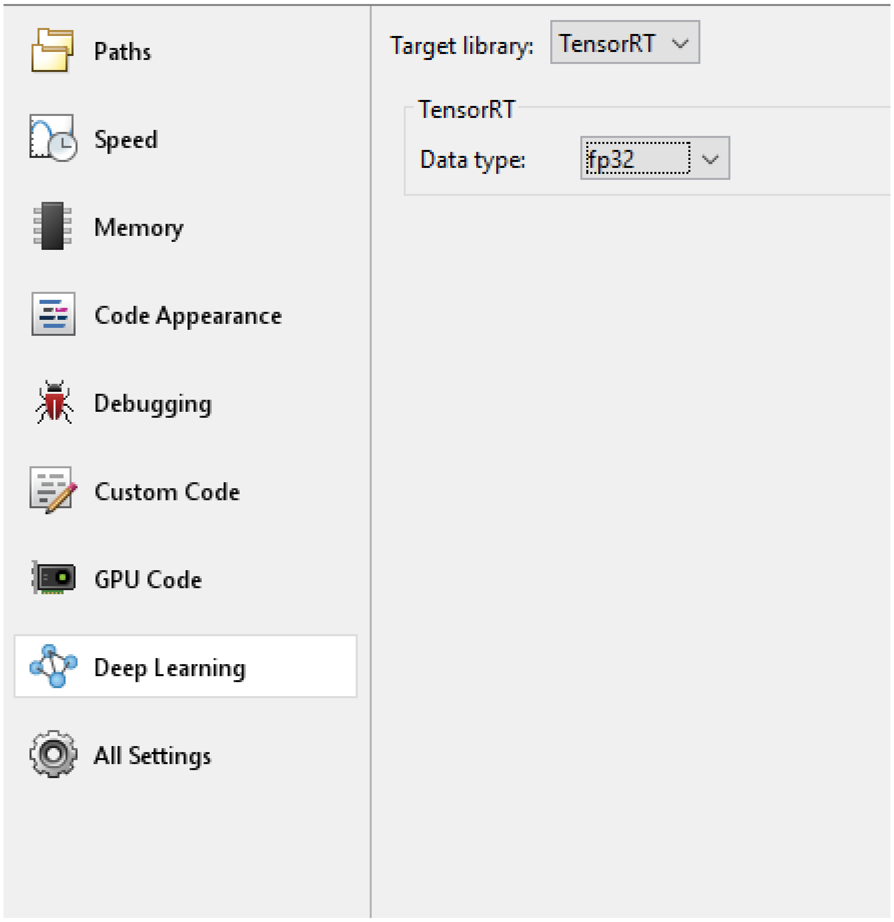
```
/* Include files */
#include "googlenet_predict.h"
#include "DeepLearningNetwork.h"
#include "predict.h"
#include "rt_nonfinite.h"

/* 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**.

Code Generation by Using cnncodegen

To generate code with the cuDNN library, use the `targetlib` option of the `cnncodegen` command. The `cnncodegen` command generates CUDA code and builds a static library for the `SeriesNetwork` or `DAGNetwork` object.

1 Load the pretrained network. For more information, see “Load Pretrained Networks for Code Generation” on page 4-15.

2 Call `cnncodegen` with `'targetlib'` specified as `'tensorrt'`. For example:

```
net = googlenet;
cnncodegen(net,'targetlib','tensorrt');
```

The `cnncodegen` command generates code, a makefile, `cnnbuild_rtw.mk`, and builds the library file `cnnbuild`. It places all the generated files in the `codegen` folder.

Generated Code

The DAG network is generated as a C++ class (`CnnMain`) containing an array of 144 layer classes. A snippet of the class declaration from `cnn_exec.hpp` file is shown.

cnn_exec.hpp File

```
class CnnMain
{
  public:
    int32_T batchSize;
    int32_T numLayers;
    real32_T *inputData;
    real32_T *outputData;
    MWCNNLayer *layers[144];
  private:
    MWTargetNetworkImpl *targetImpl;
  public:
    void presetup();
    void allocate();
    void postsetup();
    CnnMain();
    void setup();
    void deallocate();
    void predict();
    void cleanup();
    real32_T *getLayerOutput(int32_T layerIndex, int32_T portIndex);
    real32_T *getInputDataPointer();
    real32_T *getOutputDataPointer();
    ~CnnMain();
};
```

- he `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 `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.

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.

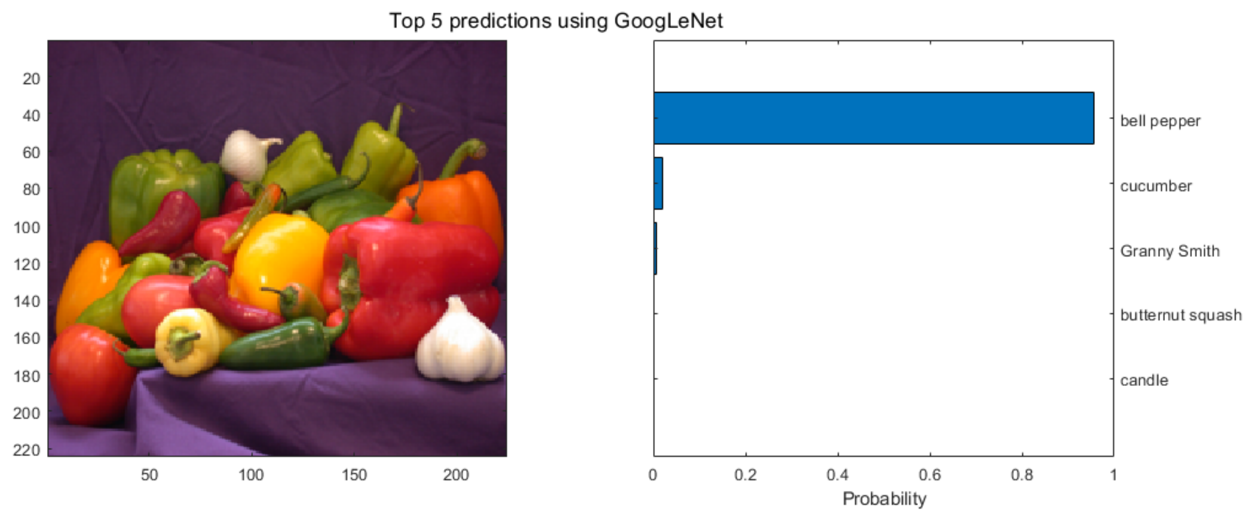
```
im = imread("peppers.png");
inputLayerSize = net.Layers(1).InputSize;
im = imresize(I,inputLayerSize(1:2));
```

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

`cnncodegen` | `codegen` | `coder.TensorRTConfig` | `coder.loadDeepLearningNetwork`

More About

- "Supported Networks and Layers" on page 4-4
- "Load Pretrained Networks for Code Generation" on page 4-15
- "Code Generation for Deep Learning Networks by Using cuDNN" on page 4-17
- "Deep Learning Prediction by Using NVIDIA TensorRT"
- "Code Generation for Deep Learning Networks"
- "Code Generation for Object Detection by Using YOLO v2"

- “Deep Learning Prediction by Using Different Batch Sizes”
- “Deployment and Classification of Webcam Images on NVIDIA Jetson TX2 Platform”

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”.

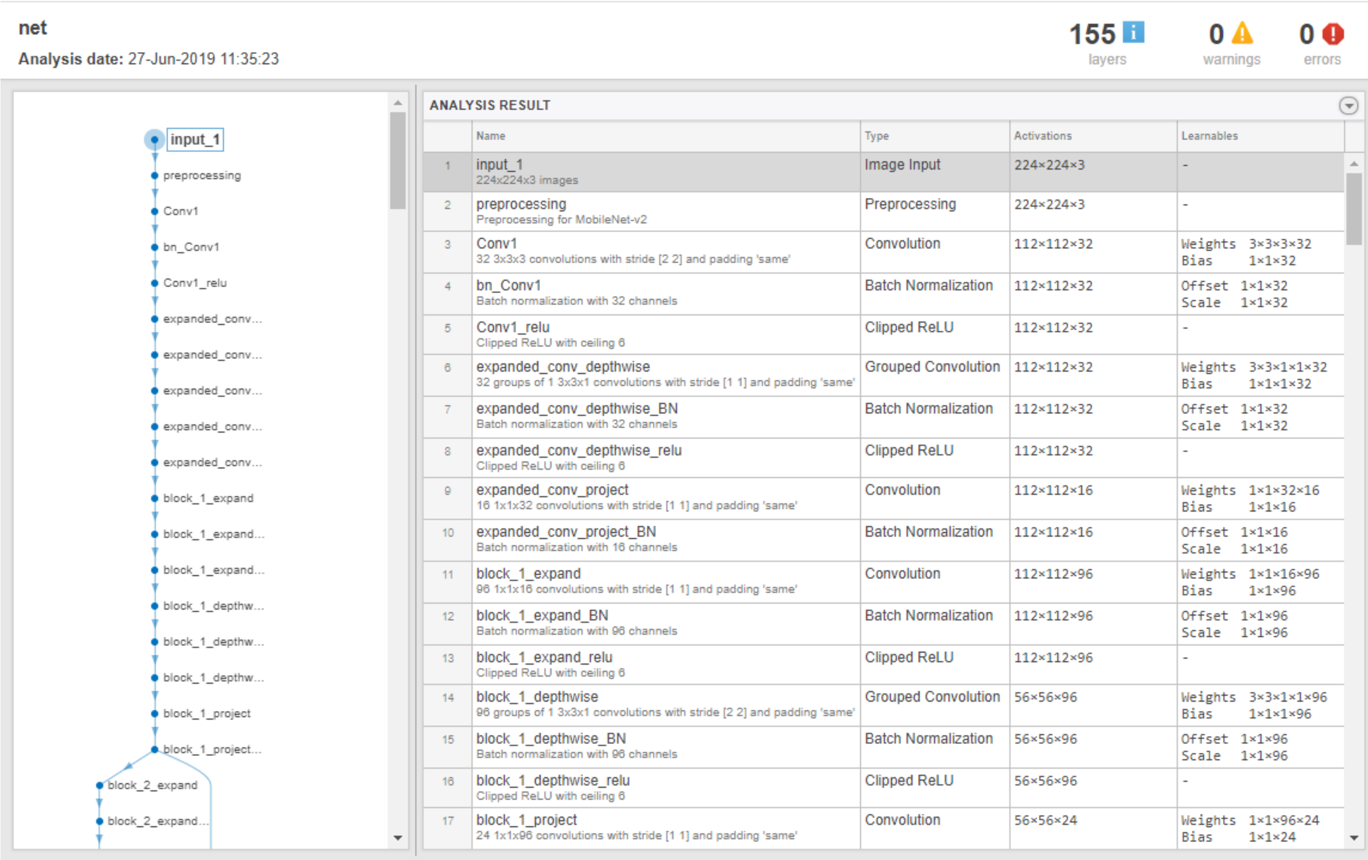
Load Pretrained Network

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);
   ```



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”

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 batchSize;
    int32_T numLayers;
    real32_T *inputData;
    real32_T *outputData;
    MWCNNLayer *layers[103];
  private:
    MWTargetNetworkImpl *targetImpl;
  public:
    void presetup();
    void allocate();
    void postsetup();
    CnnMain();
    void setup();
    void deallocate();
    void predict();
    void cleanup();
    real32_T *getLayerOutput(int32_T layerIndex, int32_T portIndex);
    real32_T *getInputDataPointer();
    real32_T *getOutputDataPointer();
    ~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

`cnncodegen` | `coder.loadDeepLearningNetwork`

More About

- “Supported Networks and Layers” on page 4-4
- “Load Pretrained Networks for Code Generation” on page 4-15
- “Code Generation for Deep Learning Networks by Using cuDNN” on page 4-17
- “Code Generation for Deep Learning Networks by Using TensorRT” on page 4-26
- “Deep Learning Prediction on ARM Mali GPU”

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:

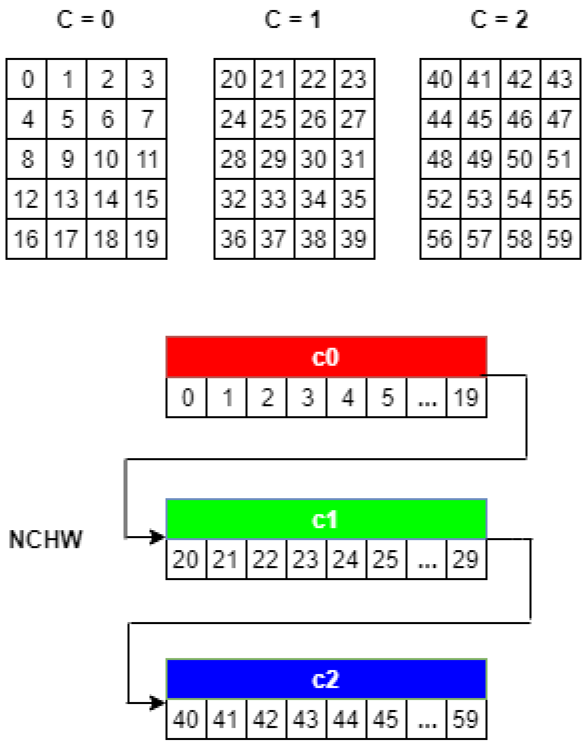
- `N` – the batch size
- `C` – the number of feature maps (number of channels)
- `H` – the height
- `W` – the width

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.



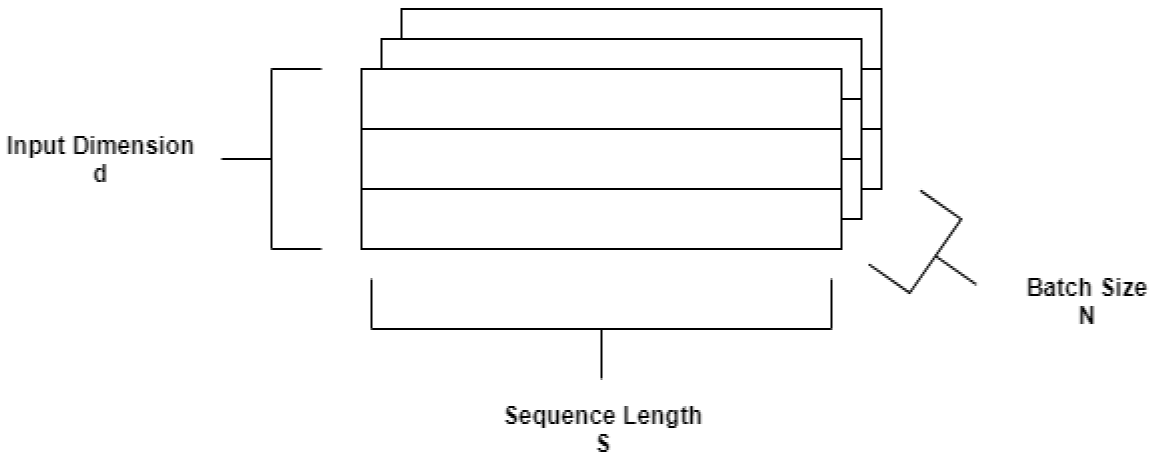
When creating the input buffer in the main function, the 4-D image is laid out in the memory in the `NCHW` format as:

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

`cnncodegen` | `codegen` | `coder.CuDNNConfig` | `coder.TensorRTConfig` | `coder.loadDeepLearningNetwork`

More About

- “Supported Networks and Layers” on page 4-4
- “Load Pretrained Networks for Code Generation” on page 4-15
- “Code Generation for Deep Learning Networks by Using cuDNN” on page 4-17
- “Code Generation for Deep Learning Networks by Using TensorRT” on page 4-26
- “Code Generation for Deep Learning Networks Targeting ARM Mali GPUs” on page 4-36
- “Lane Detection Optimized with GPU Coder”
- “Deep Learning Prediction by Using Different Batch Sizes”
- “Deployment and Classification of Webcam Images on NVIDIA Jetson TX2 Platform”

Targeting Embedded GPU Devices

You can use GPU Coder to generate CUDAcode for targeting embedded GPU platforms. Specifically, you can target the NVIDIA Tegra development boards Jetson TX2, TX1, and TK1 on either Windows or Linux systems.

Build and Run an Executable on NVIDIA Hardware

In this section...

Using GPU Coder and the GPU Coder Support Package for NVIDIA GPUs, 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.

Learning Objectives

In this tutorial, you learn how to:

- Prepare your MATLAB code for CUDA code generation by using the `kernelfun` 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 TX1/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.
- 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" (GPU Coder Support Package for NVIDIA GPUs).

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 Products". 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 GPU Coder Support Package for NVIDIA GPUs” 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('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.

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

drive | drive | jetson | jetson | killApplication | killProcess | openShell | runApplication | runExecutable | system

More About

- “Build and Run an Executable on NVIDIA Hardware Using GPU Coder App” on page 5-7
- “Code Generation Using the Command Line Interface”
- “Code Generation by Using the GPU Coder App”
- “Code Generation for Deep Learning Networks by Using cuDNN” on page 4-17
- “Code Generation for Deep Learning Networks by Using TensorRT” on page 4-26
- “Stop or Restart an Executable Running on NVIDIA Hardware” (GPU Coder Support Package for NVIDIA GPUs)
- “Run Linux Commands on NVIDIA Hardware” (GPU Coder Support Package for NVIDIA GPUs)

Build and Run an Executable on NVIDIA Hardware Using GPU Coder App

In this section...

Using GPU Coder and the GPU Coder Support Package for NVIDIA GPUs, 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

Learning Objectives

In this tutorial, you learn how to:

- Prepare your MATLAB code for CUDA code generation by using the `kernelfun` 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 TX1/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.
- 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" (GPU Coder Support Package for NVIDIA GPUs).

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 Products”. 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 GPU Coder Support Package for NVIDIA GPUs” 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

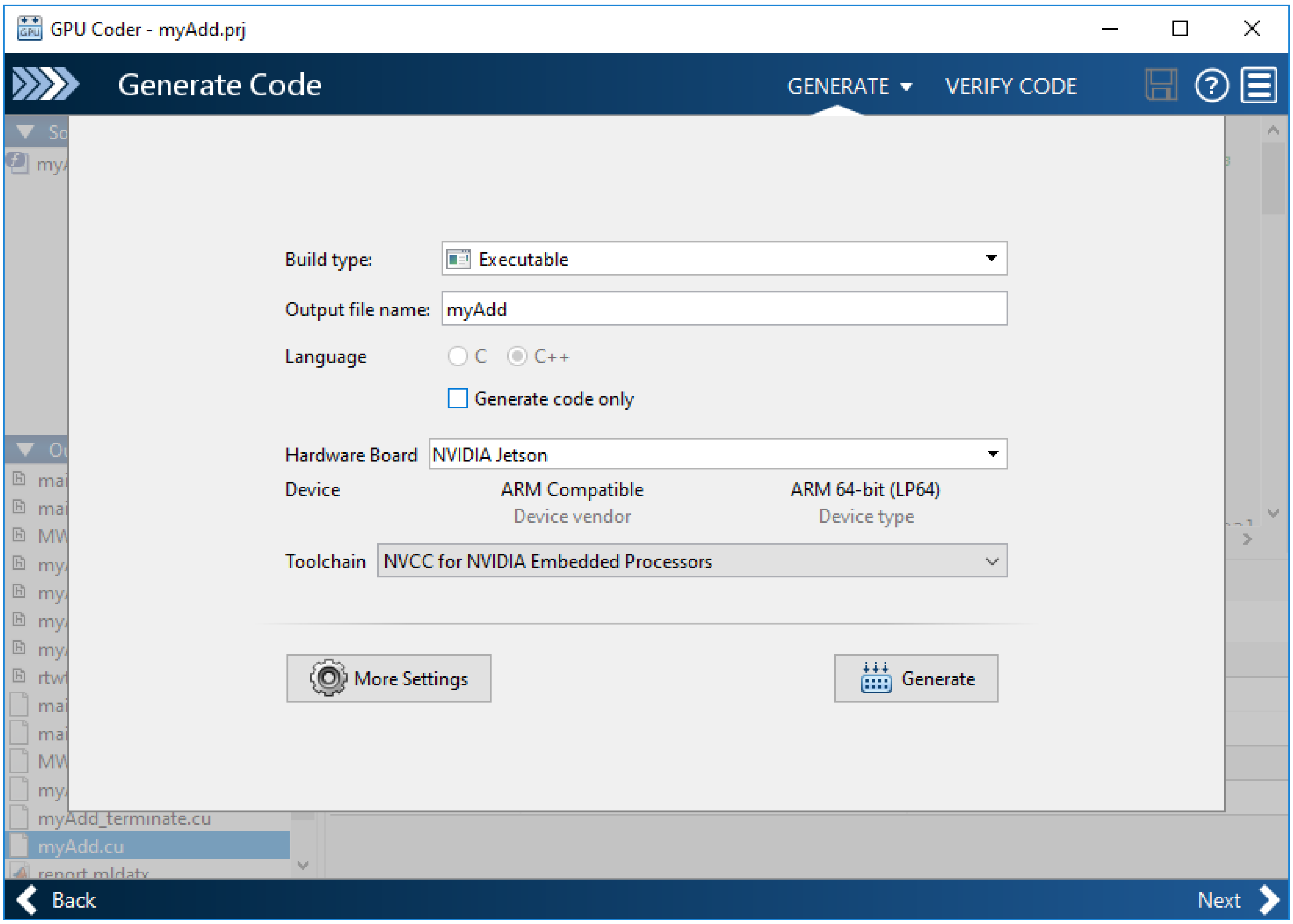
```
//main.h
#ifndef MAIN_H
#define MAIN_H

// 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

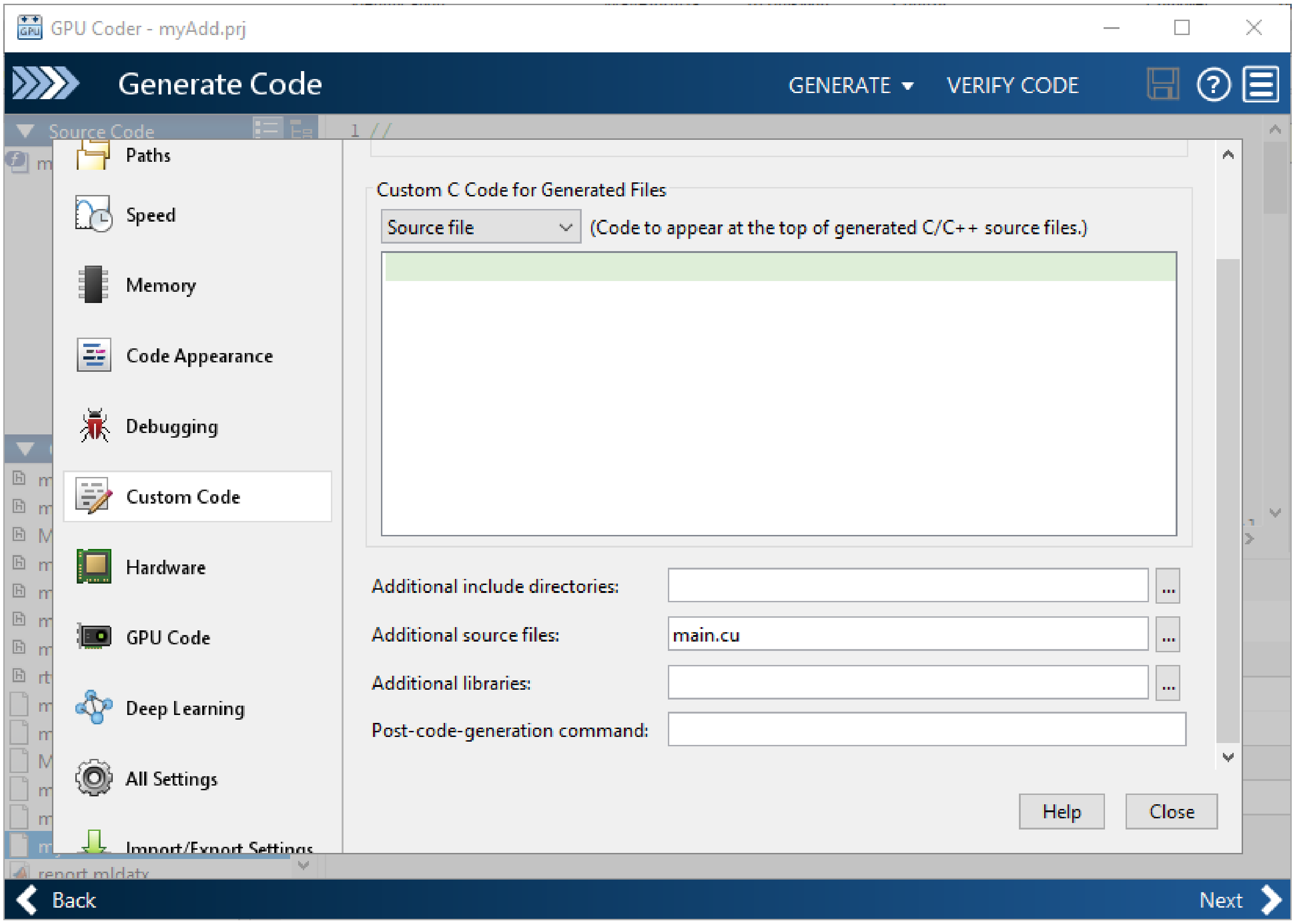
To open the GPU Coder app, on the MATLAB toolstrip **Apps** tab, under **Code Generation**, click the GPU Coder app icon. You can also open the app by typing `gpucoder` in the MATLAB Command Window.

1 The app opens the **Select** source files page. Select `myAdd.m` as the entry-point function. Click **Next**.

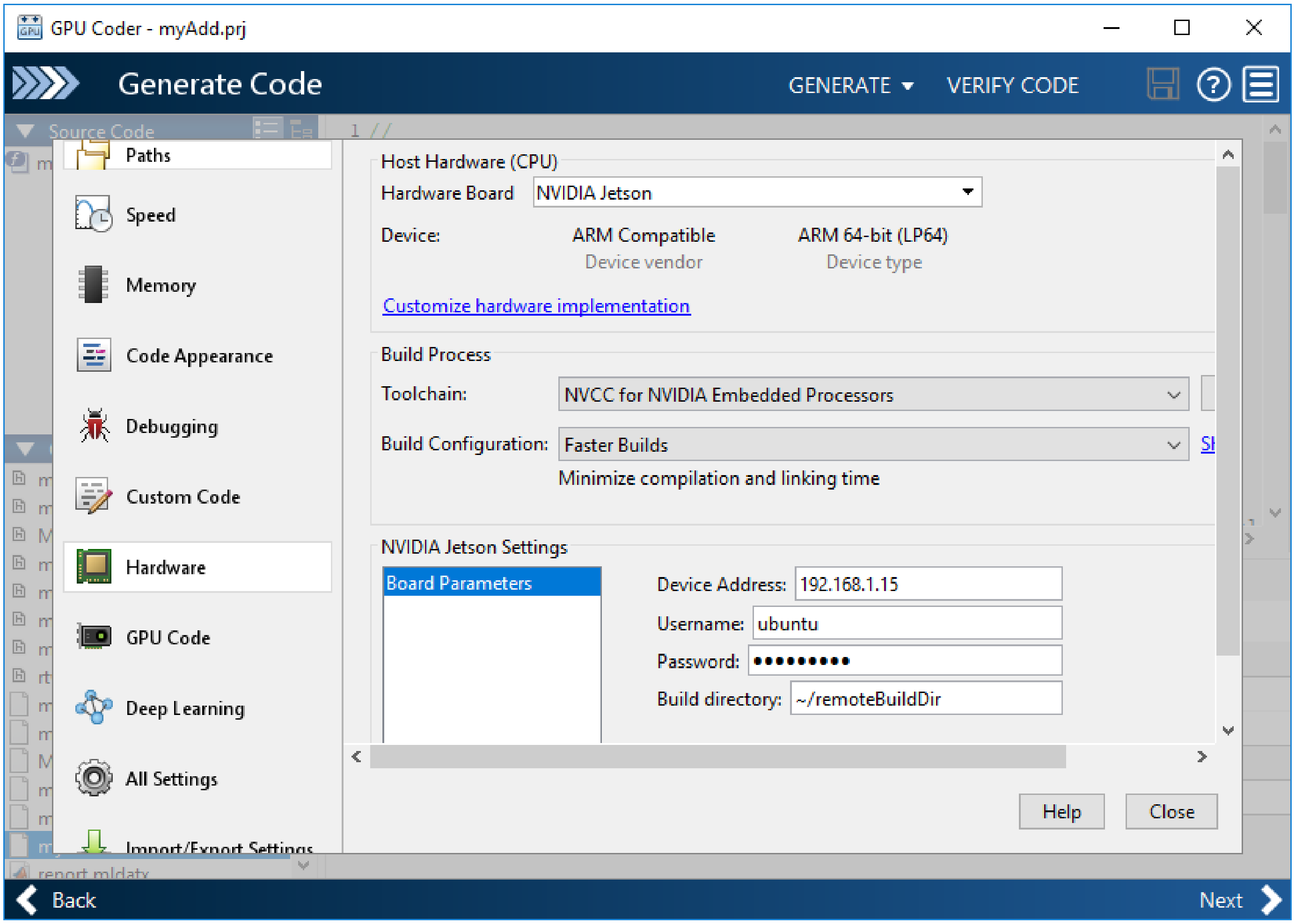
2 In the **Define Input Types** window, enter `myAdd(1:100,1:100)` and click **Autodefine Input Types**, then click **Next**.
3 You can initiate the **Check for Run-Time Issues** process or click **Next** to go to the **Generate Code** step.
4 Set the **Build type** to `Executable` and the **Hardware 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

`drive` | `drive` | `jetson` | `jetson` | `killApplication` | `killProcess` | `openShell` | `runApplication` | `runExecutable` | `system`

More About

- “Build and Run an Executable on NVIDIA Hardware” on page 5-2
- “Code Generation Using the Command Line Interface”
- “Code Generation by Using the GPU Coder App”
- “Code Generation for Deep Learning Networks by Using cuDNN” on page 4-17
- “Code Generation for Deep Learning Networks by Using TensorRT” on page 4-26
- “Stop or Restart an Executable Running on NVIDIA Hardware” (GPU Coder Support Package for NVIDIA GPUs)
- “Run Linux Commands on NVIDIA Hardware” (GPU Coder Support Package for NVIDIA GPUs)

Relocate Generated Code to Another Development Environment

In this section...

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® hardware, compilers, and libraries. For information on the supported versions of the compilers and libraries, see "Third-party Products". For setting up the environment variables, see "Setting Up the Prerequisite Products".

The 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]);
```

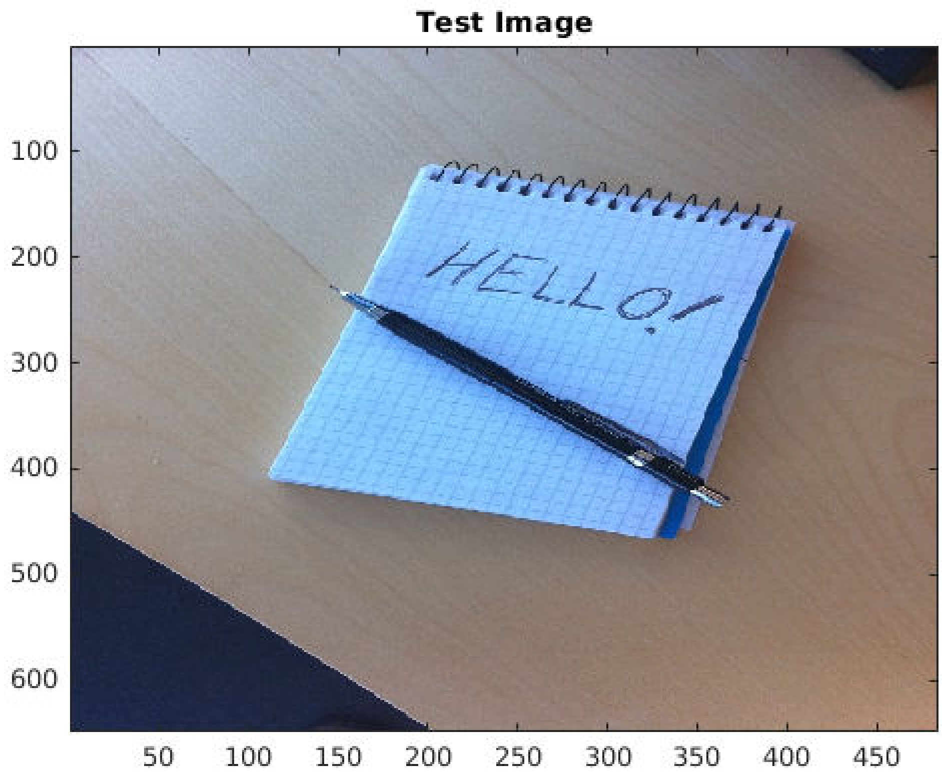
```
coder.gpu.kernelfun();



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 threhold 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');
```

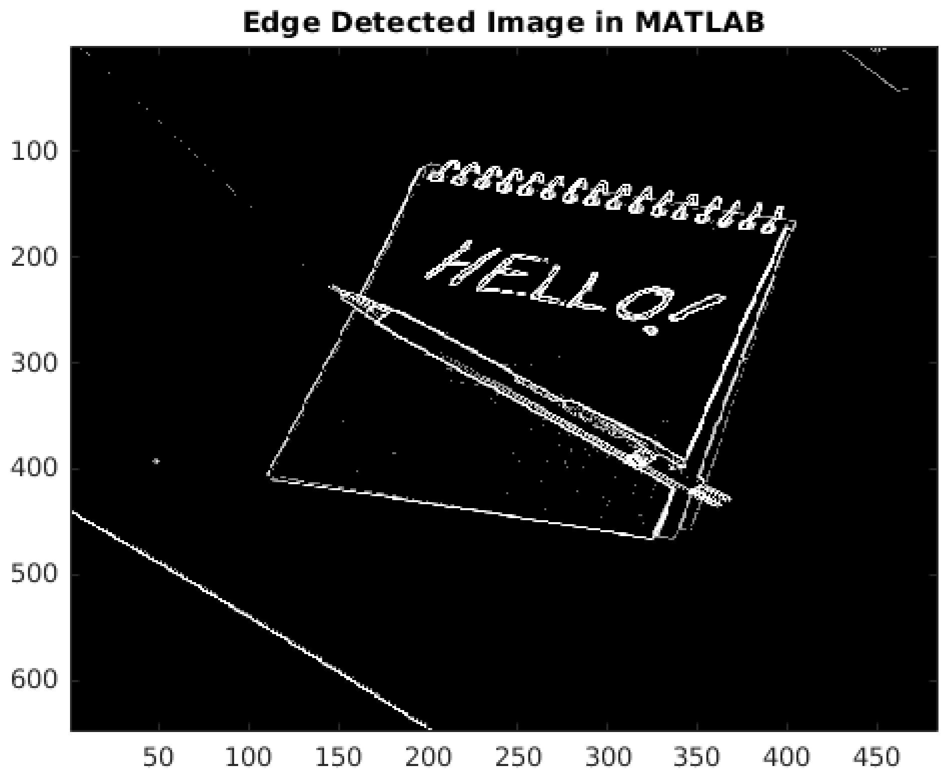


Write the matrix gray into the `imputImage.csv` file using the `writematrix` command. The Sobel edge detection application reads in this CSV file.

```
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`

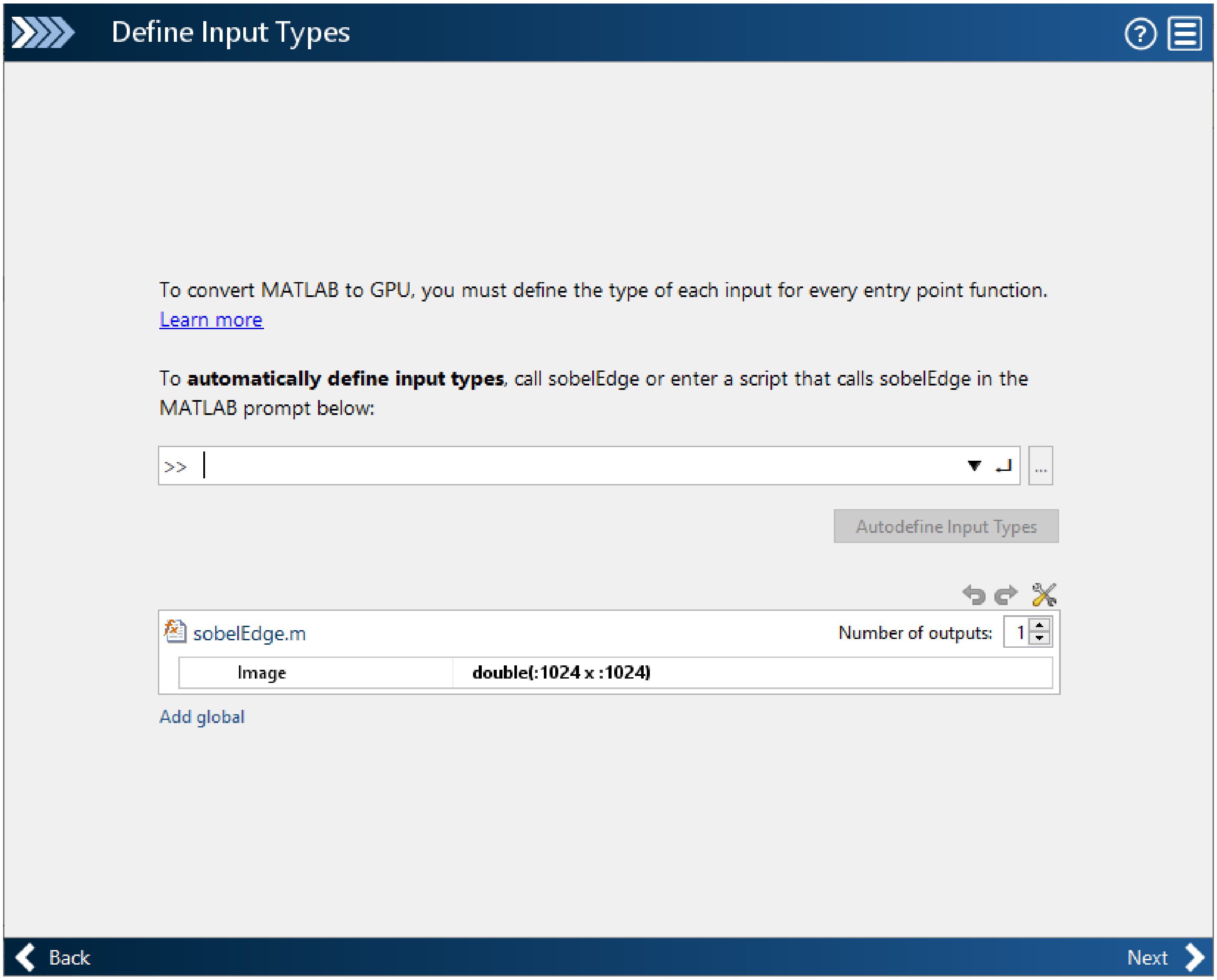
This example uses a custom main file, main_sobel.cu and its associated header file main_sobel.h. This custom main file reads the input image from the `inputImage.csv` file, calls the `sobelEdge` function in the generated `sobelEdge.cu` file, and saves the data from the edge detected image into the `outputMag.csv`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

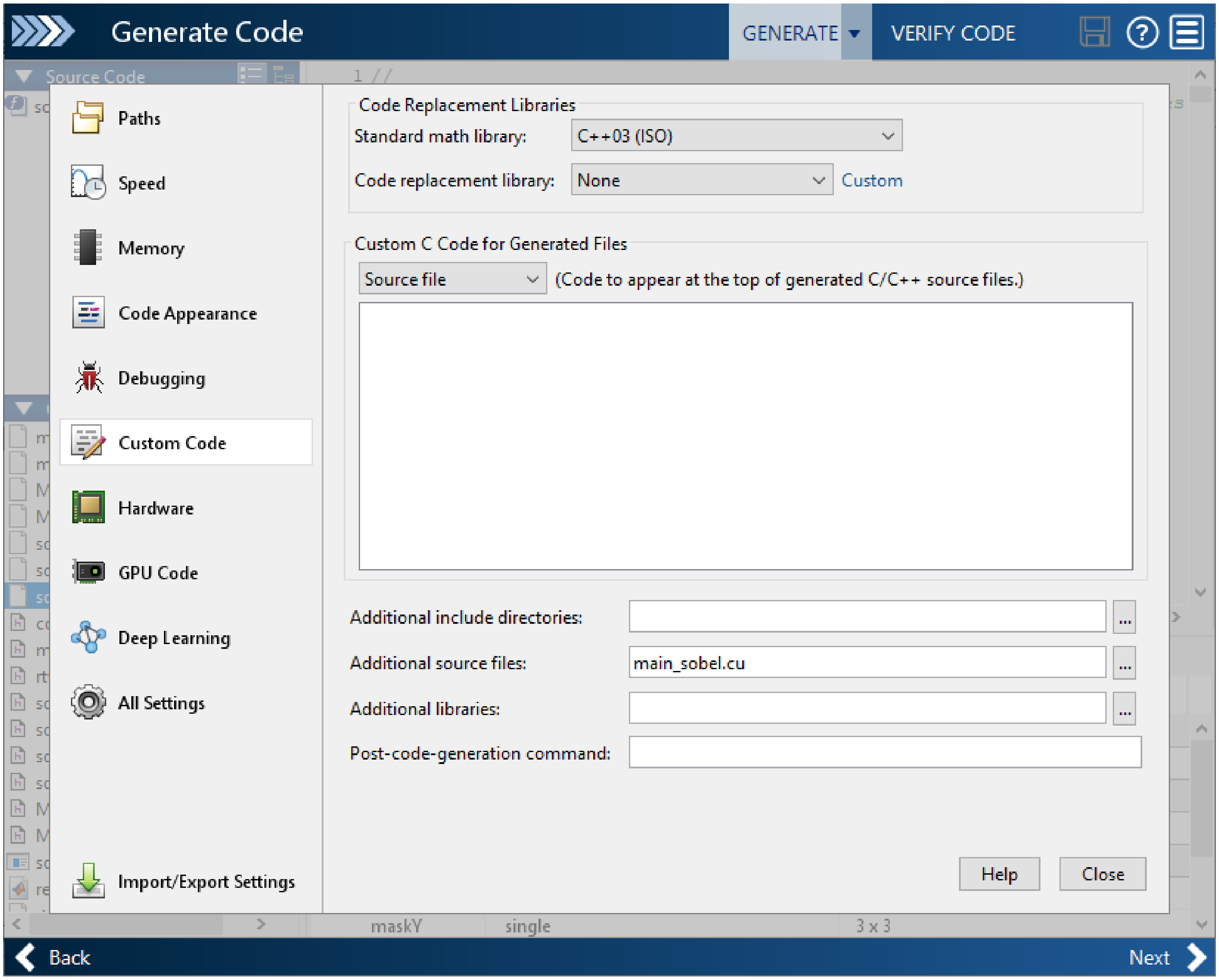
Specify that the input `Image` is of double data type and variable size with upper bound of 1024. To specify variable size with an upper bound of 1024, select `:1024`. Click **Next** to go to the **Check for Run-Time Issues** page.



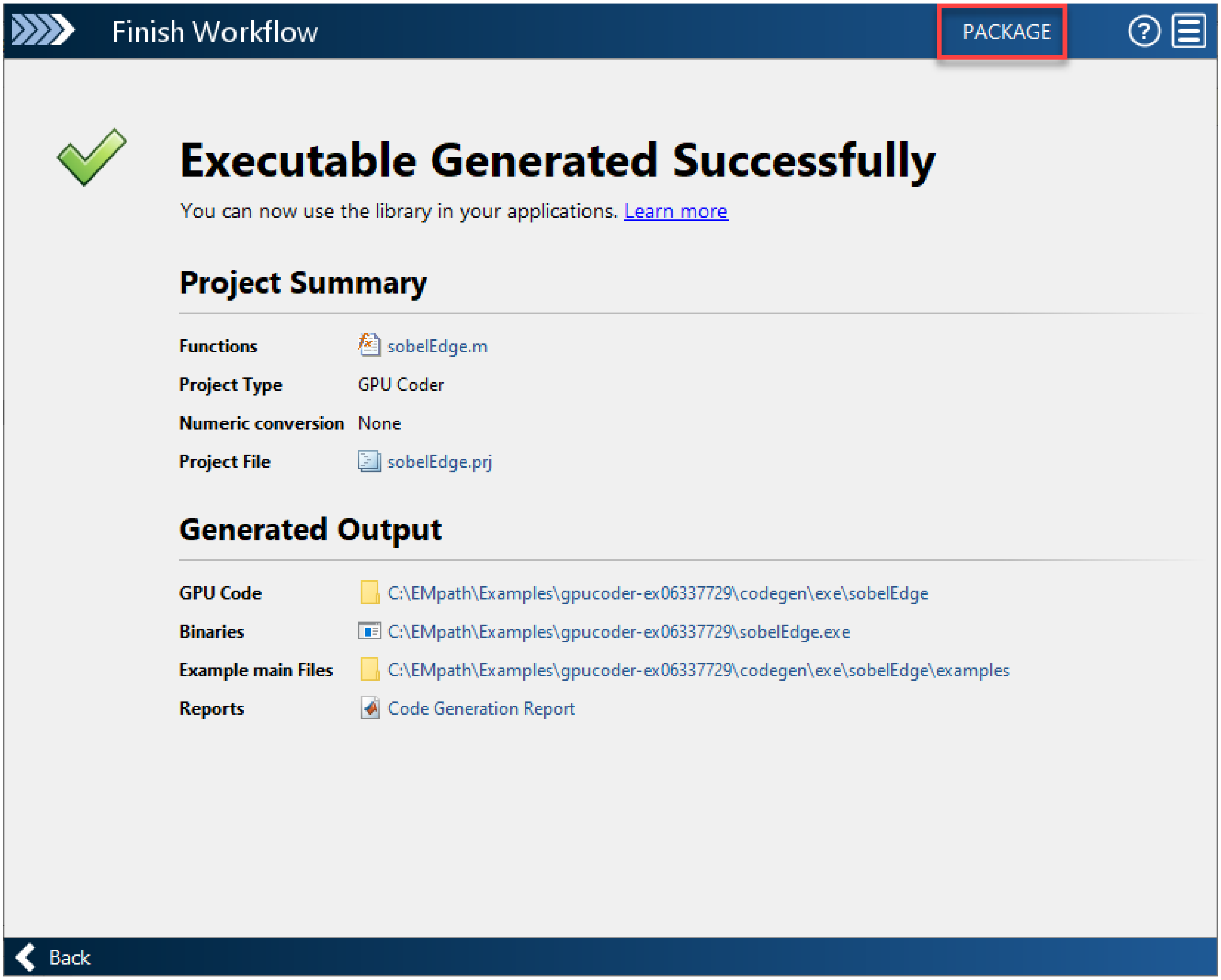
Check for run-time issues. In the **Check for Run-Time** Issues dialog box, enter code that calls `sobelEdge` with double input. For example, `sobelEdge(ones(648,484))`. Click **Check for Issues**. To check for run-time issues, the app generates and runs a MEX function. The app does not find issues for `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**.



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.

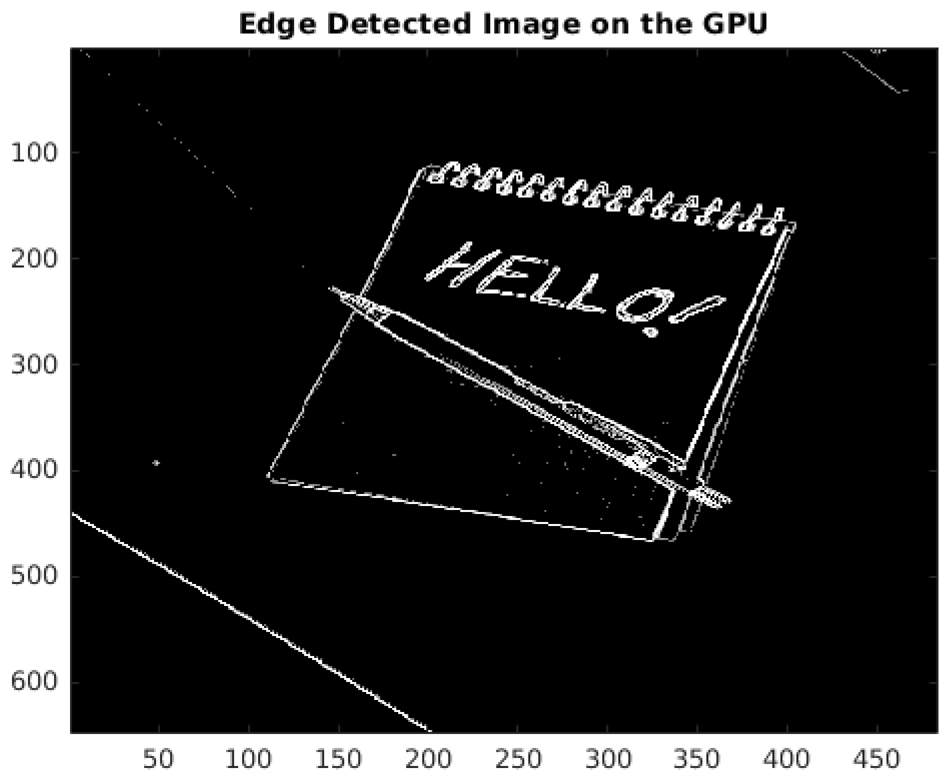
Inspect the contents of `sobelEdge.zip` in your working folder to verify that it is ready for relocation to the destination system. Depending on the zip tool that you use, you can potentially open and inspect the file without unpacking it. You can now relocate the resulting zip file to the desired development 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.	`packNGo(buildInfo,'packType','hierarchical');`
Change the structure of the file packaging to hierarchical and rename the primary zip file.	`packNGo(buildInfo,'packType','hierarchical',...` `'fileName','zippedsrcs');`
Include all header files found on the include path in the zip file (rather than the minimal header files required to build the code). For GPU Coder, this option must be set to false.	`packNGo(buildInfo,'minimalHeaders',false);`

Generate warnings for parse errors and missing files.	`packNGo(buildInfo,'ignoreParseError',`
	`true,...`
	`'ignoreFileMissing',true);`

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.