GPU Coder™ Release Notes

MATLAB®

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

Contents

R2020a

R2019b

R2019a

R2018b

R2018a

R2017b

R2021a

Version: 2.1

New Features

Bug Fixes

Compatibility Considerations

Code Optimization: Control the number of blocks created during kernel launch

In R2021a, you can specify the maximum number of blocks created during a kernel launch. Because GPU devices have limited streaming multiprocessor (SM) resources, limiting the number of blocks for each kernel can avoid performance losses from scheduling, loading and unloading of blocks. To specify the maximum number of blocks for each kernel, use one of these methods:

- In a GPU code configuration object (`coder.GpuCodeConfig`), set a valid value for the `MaximumBlocksPerKernel` property.
- In the GPU Coder™ app, on the **GPU Code** tab, set a valid value for **Maximum Blocks Per Kernel**.
- In the Simulink® Configuration Parameters dialog box, **Code Generation > GPU Code** pane, set a valid value for the **Maximum blocks per kernel** parameter.

If the number of iterations in a loop is greater than the maximum number of blocks per kernel, the code generator creates CUDA® kernels with striding.

When you specify the maximum number of blocks for each kernel, the code generator creates 1-D kernels. To force the code generator to create 2-D or 3-D kernels, use the `coder.gpu.kernel` pragma. The `coder.gpu.kernel` pragma takes precedence over the maximum number of kernels for each block.

Generate code from MATLAB for dlnetwork workflows that uses deep learning arrays

In R2021a, you can generate code for `dlnetwork` and `dlarray` that you use to run inference with `dlnetwork`. Code generation support includes:

- Construction of formatted and unformatted `dlarray`
- Passing `dlarray` to entry-point functions and returning `dlarray` from entry-point functions
- Invoking a subset of functions on `dlarray` objects, including the object functions `softmax`, `sigmoid`, and `fullyconnect`
- Passing formatted `dlarray` to the `dlnetwork` predict function inside an entry-point function.

See:

- Help topic: "Code Generation for dlarray"
- Example: "Generate Digit Images on NVIDIA GPU Using Variational Autoencoder"

Generate code that uses newer versions of NVIDIA cuDNN and TensorRT libraries

In R2021a, you can generate CUDA code for layers and networks that uses these newer versions of NVIDIA® CUDA deep neural network library (cuDNN) and TensorRT high performance inference library for NVIDIA GPUs.

- NVIDIA CUDA deep neural network library (cuDNN), version 8.1.0.
- NVIDIA TensorRT high performance inference library, version 7.2.x.

See "Installing Prerequisite Products".

Compatibility Considerations

When performing inference in INT8 (8-bit integer) precision using cuDNN version 8.1.0, issues in the NVIDIA library may cause significant degradation in performance.

Deep Learning Layers: Generate code for additional layers

Code generation by using the CUDA Deep Neural Network library (cuDNN) supports this additional layer:

- `featureInputLayer` inputs feature data to a network and applies data normalization.

Code generation by using the cuDNN library with inference computation in 8-bit integer precision supports this additional layer:

- `maxPooling2dLayer` performs down-sampling by dividing the input into rectangular pooling regions and computing the maximum of each region.

Code generation by using the NVIDIA TensorRT Library supports this additional layer:

- `featureInputLayer` inputs feature data to a network and applies data normalization.

Code generation for ARM® Mali GPUs using the ARM Compute Library supports this additional layer:

- `featureInputLayer` inputs feature data to a network and applies data normalization.

For more information, see Supported Networks and Layers.

Code generation for additional Computer Vision Toolbox functions

In R2021a, you can generate optimized CUDA code for these additional Computer Vision Toolbox™ toolbox functions:

- `pcregisterndt`
- `pcdownsample`
- `pcbin`
- `segmentGroundFromLidarData`

Code generation for additional Wavelet Toolbox functions

In R2021a, you can generate optimized CUDA code for these additional Wavelet Toolbox™ toolbox functions:

- **Discrete Wavelet Transforms** — `waverec` and `waverec2`
- **Denoising** — `wdenoise` and `wdenoise2`

Code generation for additional MATLAB functions

In R2021a, you can generate optimized CUDA code for these additional MATLAB® toolbox functions:

- histcounts
- interp1

GPU Coder Support Package for NVIDIA GPUs is moved to MATLAB Coder Support Package for NVIDIA Jetson and NVIDIA DRIVE Platforms

Starting in R2021a, the GPU Coder Support Package for NVIDIA GPUs is named MATLAB Coder™ Support Package for NVIDIA Jetson® and NVIDIA DRIVE Platforms. To use this support package in R2021a, you must have the MATLAB Coder product. For more information, see "GPU Coder Supported Hardware".

Functionality being removed or changed

cnncodegen Function: ARM Mali target support only
Warns

In a future release, the cnncodegen function will generate C++ code and build a static library for only the ARM Mali GPU processor. You can continue to use the `'arm-compute-mali'` value for the `'targetlib'` argument to target an ARM Mali GPU by using the ARM Compute Library for computer vision and machine learning.

For all other targets, use the codegen command. To learn about targeting cuDNN targets by using the codegen function, see "Code Generation for Deep Learning Networks by Using cuDNN". To learn about targeting TensorRT targets by using the codegen function, see "Code Generation for Deep Learning Networks by Using TensorRT".

Deprecating support for unified memory allocation mode on host
Behavior change in future release

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

When generating CUDA code from MATLAB, set the MallocMode property of the coder.gpuConfig code configuration object to `'discrete'`.

When generating CUDA code from Simulink models, select `'discrete'` for the **Memory mode** parameter in the **Code Generation>GPU Code** pane.

R2020b

Version: 2.0

New Features

Bug Fixes

Compatibility Considerations

Simulink Support: Generate, build, and deploy Simulink models to NVIDIA GPUs

In R2020b, you can use GPU Coder to generate and execute optimized CUDA C++ code from Simulink models that contain MATLAB Function blocks. The generated code calls optimized NVIDIA CUDA libraries, including cuFFT, cuSolver, and cuBLAS.

You can use the generated CUDA code within Simulink to:

- Speed up the execution of your Simulink model by using NVIDIA GPUs. When you simulate a model that contains a MATLAB Function block, the software generates CUDA MATLAB executable (MEX) code from the block and dynamically links the generated code to Simulink. For more information, see Simulation Acceleration by Using GPU Coder.
- Build an executable that you can use for rapid prototyping on NVIDIA GPUs. For more information, see Code Generation from Simulink Models by Using GPU Coder.
- Deploy the Simulink models on embedded NVIDIA GPUs such as Jetson and DRIVE by using the GPU Coder Support Package for NVIDIA GPUs. You can also remotely communicate with the NVIDIA target and control the peripheral devices for prototyping. For more information, see Targeting NVIDIA Embedded Boards.

Deep Learning Simulink Support: Generate, build, and deploy deep learning networks in Simulink models to NVIDIA GPUs

In this release, you can use GPU Coder to generate and execute optimized CUDA C++ code for deep learning networks in Simulink models. The generated code calls optimized NVIDIA CUDA libraries, including cuDNN and TensorRT.

You can use the generated CUDA code within Simulink to:

- Speed up the execution of your Simulink model by using NVIDIA GPUs. When you simulate a model that contains a MATLAB Function or Deep Learning Toolbox™ blocks, the software generates CUDA MATLAB executable (MEX) code from the block and dynamically links the generated code to Simulink.
- Build an executable that you can use for rapid prototyping on NVIDIA GPUs.

For more information, see Deep Learning in Simulink Using Deep Neural Networks Library and Deep Learning in Simulink Using MATLAB Function Block. You can also deploy the Simulink models on embedded NVIDIA GPUs such as Jetson and DRIVE by using the GPU Coder Support Package for NVIDIA GPUs. For more information, see Targeting NVIDIA Embedded Boards.

Simulink Support: SIL, PIL, and external mode simulations

Test numerical equivalence between model components and production code that you generate from the components by using software-in-the-loop (SIL) and processor-in-the-loop (PIL) simulations. With a SIL simulation, you test the behavior of generated source code on development computer. Simulation does not test code compiled for target hardware because code is compiled for the development computer. With a PIL simulation, you test the compiled object code that you intend to deploy on a target hardware by running the object code on real target hardware.

To determine whether model components and generated code are numerically equivalent, compare GPU acceleration and PIL results against normal mode results. For more information, see Numerical Equivalence Testing

Use external mode simulations for rapid prototyping. With external mode simulation, you can:

- Modify or tune block parameters in real time. When you change parameters in your model, Simulink downloads the new values to the executing target application.
- Monitor and save signal data from the executing target application.

For more information, see Parameter Tuning and Signal Monitoring Using External Mode.

Persistent Variables: Create persistent memory on the GPU

In this release, you can use the `coder.gpu.persistentMemory` pragma to allocate a variable as persistent memory on the GPU. The variable must be fixed size and of a data type supported for GPU code generation.

For example, when generating CUDA code for the entry-point function `foo`, the persistent variable `p` is mapped to the GPU as variable with persistent memory.

```
function output = foo(input)
coder.gpu.kernelfun();
persistent p;
if isempty(p)
    p = zeros(1024,1);
end

coder.gpu.persistentMemory(p);
p = p + 1;
output = input + p;
end
```

Wavelet Toolbox Code Generation: Generate code for FFT-based FIR filtering and Short-time Fourier transform functions

In R2020b, you can generate optimized CUDA code for the following additional Wavelet Toolbox toolbox functions.

- modwt
- imodwt
- modwtmra
- dwt
- idwt
- dwt2
- idwt2
- wavedec
- wavedec2
- mdwtdec

Deep Learning: Generate code for custom layers

In R2020b, you can generate CUDA code for deep learning networks that have custom deep learning layers. Custom layer code generation supports NVIDIA CUDA deep neural network library (cuDNN) and the NVIDIA TensorRT high performance inference library for NVIDIA GPUs. When targeting the cuDNN library, the code generator supports both row-major and column-major code generation for custom layers. For other usage notes and limitations, see Custom Layers.

To learn how to define custom deep learning layers, see Define Custom Deep Learning Layers (Deep Learning Toolbox) and Define Custom Deep Learning Layer for Code Generation (Deep Learning Toolbox). For an example on how to generate code for a network with custom layers, see Code Generation For Object Detection Using YOLO v3 Deep Learning.

Multi-Input Networks: Generate code for networks that have multiple inputs

In R2020b, you can generate code for networks that have multiple input layers. For information on training multiple input networks, see Multiple-Input and Multiple-Output Networks (Deep Learning Toolbox).

Convolutional Recurrent Neural Networks: Generate code for convolutional LSTM

In this release, you can generate CUDA code for convolutional LSTM networks. Convolutional LSTM is a type of LSTM network that is made up of convolutional and LSTM layers. Such a network is useful for image and video classification applications where you use convolutional layers to extract features from each frame independently. For more information, see Long Short-Term Memory Networks (Deep Learning Toolbox).

Long Short-Term Memory (LSTM) Networks: Generate code for network activations

You can generate code for the `activations` method and compute the network activations for a specific layer of an LSTM network. For example, the following line of code returns the network activations for the sequence or time series data you specify in `sequences` and the layer you specify in `layerIdx`.

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

Workflow improvements

In R2020b, when generating CUDA MEX with GPU Coder, the code generator uses the NVIDIA compiler and libraries included with MATLAB. The CUDA toolkit installed with MATLAB includes CUDA runtime, cuBLAS, cuFFT, cuSOLVER, cuDNN, and TensorRT libraries. To use CUDA MEX, you must have a compatible C/C++ compiler, a CUDA enabled GPU device and a CUDA compatible graphics driver.

For more information, see Installing Prerequisite Products.

cuFFT Library Support: Improved performance of generated code for fast Fourier transform (FFT) functions

Compared to previous releases, the code generated for FFT functions by using the cuFFT library calls can have improved performance. In R2020b, the software caches multiple cuFFT plans that have the same geometry. Because some cuFFT plans allocate memory on the GPU, caching can improve memory utilization and improve performance for repeatedly running FFT calls.

Deep Learning Networks: Generate code for additional networks

Code generation by using the CUDA Deep Neural Network library (cuDNN) supports these additional pretrained networks:

- `efficientnetb0` – EfficientNet-b0 model network trained on the ImageNet data set.

Code generation by using the NVIDIA TensorRT Library supports these additional pretrained networks:

- `efficientnetb0` – EfficientNet-b0 model network trained on the ImageNet data set.

Code generation by using the ARM Compute Library supports these additional pretrained networks:

- `efficientnetb0` – EfficientNet-b0 model network trained on the ImageNet data set.

For more information, see Supported Networks and Layers.

Deep Learning Layers: Generate code for additional layers

Code generation by using the CUDA Deep Neural Network library (cuDNN) supports these additional layers:

- `focalLossLayer` predicts object classes using focal loss.
- `gruLayer` creates a gated recurrent unit (GRU) that learns dependencies between time steps in time series and sequence data.
- `rcnnBoxRegressionLayer` refines bounding box locations by using a smooth L1 loss function.
- `rpnClassificationLayer` for region proposal networks RPNs.
- `scalingLayer` for actor or critic network.
- `sequenceFoldingLayer` to convert a batch of image sequences to a batch of images.
- `sequenceUnfoldingLayer` to restore the sequence structure of the input data after sequence folding.
- `sigmoidLayer` applies a sigmoid function to its input such that its output is bounded in the interval (0,1).
- `spaceToDepthLayer` permutes the spatial blocks of the input into the depth dimension.
- `softplusLayer` for actor or critic network.

Code generation by using the NVIDIA TensorRT Library supports these additional layers:

- `focalLossLayer` predicts object classes using focal loss.
- `gruLayer` creates a gated recurrent unit (GRU) that learns dependencies between time steps in time series and sequence data.

- `rcnnBoxRegressionLayer` refines bounding box locations by using a smooth L1 loss function.
- `rpnClassificationLayer` for region proposal networks RPNs.
- `scalingLayer` for actor or critic network.
- `sigmoidLayer` applies a sigmoid function to its input such that its output is bounded in the interval (0,1).
- `spaceToDepthLayer` permutes the spatial blocks of the input into the depth dimension.
- `softplusLayer` for actor or critic network.

Code generation for ARM Mali GPUs using the ARM Compute Library supports these additional layers:

- `focalLossLayer` predicts object classes using focal loss.
- `rcnnBoxRegressionLayer` refines bounding box locations by using a smooth L1 loss function.
- `rpnClassificationLayer` for region proposal networks RPNs.
- `scalingLayer` for actor or critic network.
- `sigmoidLayer` applies a sigmoid function to its input such that its output is bounded in the interval (0,1).
- `spaceToDepthLayer` permutes the spatial blocks of the input into the depth dimension.

For more information, see Supported Networks and Layers.

Code generation for additional Computer Vision Toolbox function

- `pcsegdist`

Code generation for additional Signal Processing Toolbox functions

- `fsst`
- `ifsst`
- `filtfilt`

New examples

This release adds the following examples:

- GPU Code Generation for Lane Detection in Simulink – This example shows how to generate CUDA code for a Simulink model that can detect and output lane marker boundaries on an image. This example takes RGB image as an input and uses the imresize, rgb2gray, ordfilt2, hough, houghpeaks, and houghlines functions that are part of Image Processing Toolbox to detect lane markings.
- GPU Code Generation for a Fog Rectification Simulink Model – Demonstrates how to generate CUDA code from the Simulink model that takes a foggy image as input and produces a defogged image as output. This example is a typical implementation of fog rectification algorithm. The example uses conv2, rgb2gray, and imhist functions.
- Code Generation for a Deep Learning Simulink Model to Classify ECG Signals – Shows how you can use powerful signal processing techniques and Convolutional Neural Networks together to classify ECG signals in Simulink. This example also shows how to generate CUDA from the Simulink model.

- Code Generation for a Deep Learning Simulink Model that Performs Lane and Vehicle Detection – Shows how to develop a CUDA application from a Simulink model that performs lane and vehicle detection using convolutional neural networks (CNN). This example takes the frames of a traffic video as an input, outputs two lane boundaries that correspond to the left and right lanes of the ego vehicle, and detects vehicles in the frame.

- Code Generation for Lidar Point Cloud Segmentation Network – Shows how to generate CUDA MEX for a lidar (light detection and ranging) semantic segmentation network that uses deep learning. This example uses the `SqueezeSegV2` network trained to segment organized lidar point clouds belonging to three classes (background, car, and truck). The generated MEX takes a point cloud input and performs prediction on the point cloud by using the `DAGNetwork` object for the `SqueezeSegV2` network.

- Code Generation for a Video Classification Network – Shows how to generate CUDA code for a deep learning network that classifies video and deploy the generated code onto the NVIDIA Jetson Xavier board using the GPU Coder Support Package for NVIDIA GPUs. The deep learning network has both convolutional and bidirectional long short-term memory (BiLSTM) layers. The generated application reads the data from a specified video file as a sequence of video frames and outputs a label that classifies the activity in the video.

- Code Generation For Object Detection Using YOLO v3 Deep Learning – This example shows how to generate CUDA® MEX for a you only look once (YOLO) v3 object detector with custom layers. The example uses YOLO v3 object detection to illustrate:

  - CUDA code generation for a deep learning network with custom layers.
  - Convert a deep learning dlnetwork object into a DAGNetwork object for code generation.

To see the full list of examples for GPU Coder, at the MATLAB command line, enter `doc gpucoder`.

Functionality being removed or changed

cnncodegen Function: ARM Mali targets support only
Behavior change in future release

In a future release, the `cnncodegen` function will generate C++ code and build a static library for only the ARM Mali GPU processor. You can continue to use the `'arm-compute-mali'` value for the `'targetlib'` argument to target an ARM Mali GPU by using the ARM Compute Library for computer vision and machine learning.

For all other targets, use the `codegen` command. To learn about targeting cuDNN targets by using the `codegen` function, see Code Generation for Deep Learning Networks by Using cuDNN. To learn about targeting TensorRT targets by using the `codegen` function, see Code Generation for Deep Learning Networks by Using TensorRT.

R2020a

Version: 1.5

New Features

Bug Fixes

Compatibility Considerations

cuBLAS Support: Generate CUDA code for strided and batched matrix multiply

In R2020a, you can generate CUDA code from MATLAB functions to compute many (small) matrix-matrix multiplies at once. This technique is known as batched matrix-matrix multiply can potentially improve device utilization and overall performance.

Use the `gpucoder.batchedMatrixMultiply` function to perform batched matrix multiply operation of the form `D = (alpha*A)xB`.

Use the `gpucoder.batchedMatrixMultiplyAdd` function to perform batched matrix multiply with add operation of the form `D = (alpha*A)xB + (beta*C)`. For example,

```
[D1,D2] = gpucoder.batchedMatrixMultiplyAdd(A1,B1,C1,A2,B2,C2,...
'alpha',0.3,'beta', 0.4,'transpose','TT');
```

You can also perform strided matrix multiplication for matrix batches, where subsequent matrices are memory-contiguous by using the `gpucoder.stridedMatrixMultiply` and `gpucoder.stridedMatrixMultiplyAdd` functions. For example,

```
D = gpucoder.stridedMatrixMultiply(A,B,'alpha',0.4,'transpose','TT');
```

Single Shot Object Detection (SSD) Networks: Object detection on NVIDIA GPU by using a single shot multibox detector

In R2020a, you can generate CUDA code for an SSD network (`ssdObjectDetector` object) and take advantage of the NVIDIA cuDNN and TensorRT libraries.

The SSD detector uses a single stage object detection network that merges detections predicted from multiscale features. The SSD is faster than two-stage detectors, such as the Faster R-CNN detector and can localize objects more accurately compared to single-scale feature detectors such as the YOLO v2 detector. For more information, see Getting Started with SSD Multibox Detection (Computer Vision Toolbox).

For more information on the SSD layers supported in this release, see Supported Networks and Layers. The Code Generation for Object Detection by Using Single Shot Multibox Detector shows how to generate CUDA code for an SSD based vehicle object detector.

Row-Major Array Layout: Simplify interfacing generated deep learning code with target libraries by storing arrays in row-major layout

The code that you generate can store array elements in column-major or row-major array layout. In column-major array layout, the elements of the columns are contiguous in memory. In row-major, the elements of the rows are contiguous. MATLAB uses column-major array layout by default, whereas the deep learning networks supported by NVIDIA cuDNN, TensorRT, and ARM Compute libraries use row-major layout by default.

In previous releases, the code generator produced CUDA C++ code that performed transpose operations on the row-major data and called predict or activation on the transposed data. In R2020a, you can choose to generate code that uses row-major array layout. Row-major layout can improve performance for certain networks and ease integration with other code that also uses row-major layout. For more information, see Array Layout (MATLAB Coder).

For more information on deep learning code generation, see Deep Learning with GPU Coder.

Long Short-Term Memory (LSTM) Networks: Generate code for bidirectional and stateful LSTM

In R2020a, you can generate CUDA code for bidirectional and stateful LSTM networks. A bidirectional LSTM network is a type of recurrent neural network (RNN) that learns bidirectional long-term dependencies between time steps of sequence data. Stateful LSTM networks can remember the state of the network between predictions. The network state can be useful when you do not have the complete time series in advance, or if you want to make multiple predictions on a long time series. For more information, see Long Short-Term Memory Networks (Deep Learning Toolbox).

For a code generation example using stateful LSTM, see Code Generation for a Sequence-to-Sequence LSTM Network

For more information on the LSTM layers supported in this release, see Supported Networks and Layers. Use the `predictAndUpdateState` to predict parts of a time series and update the network state. Use the `resetState` to reset the network state between predictions.

Multi-Output Networks: Generate code for networks with multiple outputs

In R2020a, you can generate code for networks with multiple output layers. For information on training multiple output networks, see Multiple-Input and Multiple-Output Networks (Deep Learning Toolbox).

Deep Learning Networks: Generate code for more networks

In R2020a, you can generate code for networks such as Darknet19, Darknet53, NASNet-Large, NASNet-Mobile, and Inception-ResNet-v2. For more information, see Supported Networks and Layers.

Generate code for half-precision floating point data type

In R2020a, you can generate CUDA code for half-precision floating point data types in MATLAB. Half-precision data types occupy only 16 bits of memory, but their floating-point representation enables them to handle wider dynamic ranges than integer or fixed-point data types of the same size.

For a full list of features that support half-precision code generation, see `half`. For examples that demonstrate half-precision code generation, see Edge Detection with Sobel Method in Half-Precision, Fog Rectification, and Stereo Disparity.

Deep Learning Layers: Generate code for more layers

Code generation with the CUDA Deep Neural Network library (cuDNN) supports these additional layers:

- `anchorBoxLayer` layer to store anchor boxes for object detection.

- Bidirectional long short-term memory (BiLSTM) layer (`bilstmLayer`).
- `concatenationLayer` that concatenates inputs along a specified dimension.
- Flatten layer (`flattenLayer`).
- Global max pooling layer (`globalMaxPooling2dLayer`).
- `ssdMergeLayer` layer to merge activations from several feature maps.
- Word embedding layer for deep learning networks (`wordEmbeddingLayer`).
- Layer that implements ONNX identity operator (`nnet.onnx.layer.IdentityLayer`).

Code generation with the NVIDIA TensorRT Library supports these additional layers:

- `anchorBoxLayer` layer to store anchor boxes for object detection.
- Bidirectional long short-term memory (BiLSTM) layer (`bilstmLayer`).
- `concatenationLayer` that concatenates inputs along a specified dimension.
- Global max pooling layer (`globalMaxPooling2dLayer`).
- Long short-term memory (LSTM) layer (`lstmLayer`).
- Sequence input layer (`sequenceInputLayer`).
- `ssdMergeLayer` layer to merge activations from several feature maps.
- Word embedding layer for deep learning networks (`wordEmbeddingLayer`).
- Layer that implements ONNX identity operator (`nnet.onnx.layer.IdentityLayer`).

Code generation with the ARM Compute Library supports these additional layers:

- `anchorBoxLayer` layer to store anchor boxes for object detection.
- Layer that applies 2-D cropping to the input (`crop2dLayer`).
- Global max pooling layer (`globalMaxPooling2dLayer`).
- Affine layer for the ONNX network that performs element-wise scaling of the input followed by an addition (`nnet.onnx.layer.ElementwiseAffineLayer`).
- Layer that implements ONNX identity operator (`nnet.onnx.layer.IdentityLayer`).

For more information, see Supported Networks and Layers.

Code generation for more MATLAB functions

- `filter`
- `fftshift`
- `circshift`

Code generation for more Image Processing Toolbox functions

- `bwlookup`
- `imrotate`
- `imboxfilt`
- `imgaussfilt`

Code generation for more Computer Vision Toolbox functions

- `disparitySGM`
- `pointCloud`

Code generation for more Signal Processing Toolbox functions

- `fftfilt`
- `stft`
- `istft`

Code generation for Audio Toolbox functions

- `mfcc`

Deep Learning: Generate code that uses newer versions of ARM Compute library

In R2020a, you can generate more efficient C++ code for layers and networks by using version 19.05 of the ARM Compute Library for computer vision and machine learning. To learn more about supported compilers and libraries, see Code Generation for a Sequence-to-Sequence LSTM NetworkInstalling Prerequisite Products. For an example on targeting the ARM Compute Library, see Code Generation for Deep Learning Networks Targeting ARM Mali GPUs.

New and updated examples

This release adds the following examples:

- Code Generation for Object Detection by Using Single Shot Multibox Detector – Shows how to generate CUDA code for an SSD network (ssdObjectDetector object) and take advantage of the NVIDIA cuDNN libraries. An SSD network is based on a feed-forward convolutional neural network that detect multiple objects within the image in a single shot. SSD network can be thought of as having two sub-networks. A feature extraction network, followed by a detection network..
- Edge Detection with Sobel Method in Half-Precision – Demonstrates edge detection in an image with a MEX function generated from a MATLAB function. The edge detection algorithm is implemented with half-precision data type.

This release updates the following examples:

- Code Generation for a Sequence-to-Sequence LSTM Network – demonstrates how to generate CUDA code for a long short-term memory (LSTM) network. The example generates a MEX application that makes predictions at each step of an input time series. Two methods are demonstrated: a method using a standard LSTM network, and a method leveraging the stateful behavior of the same LSTM network. This example uses accelerometer sensor data from a smartphone carried on the body and makes predictions on the activity of the wearer. User movements are classified into one of five categories, namely dancing, running, sitting, standing, and walking.
- Fog Rectification – Shows the use of image processing functions for GPU code generation. This example also shows half-precision code generation using GPU Coder.

- Stereo Disparity – Shows how to generate a MEX function from a MATLAB function that computes the stereo disparity of two images. This example also shows half-precision code generation using GPU Coder.

To see the full list of examples for GPU Coder, at the MATLAB command line, enter `doc gpucoder`.

Functionality being removed or changed

The `coder.checkGpuInstallApp` has been renamed to `gpucoderSetup`.

Compatibility Considerations

Functionality	What Happens When You Use This Functionality?	Use This Instead	Compatibility Considerations
`coder.checkGpuInstallApp`	Still runs	Use `gpucoderSetup`	Replace all instances of `coder.checkGpuInstallApp` with `gpucoderSetup`.

R2019b

Version: 1.4

New Features

Bug Fixes

Compatibility Considerations

Long Short-Term Memory (LSTM) Networks: Generate code for recurrent networks such as LSTM

In R2019b, you can generate CUDA code for an LSTM network and take advantage of the NVIDIA cuDNN library. An LSTM network is a type of recurrent neural network (RNN) that can learn long-term dependencies between time steps of sequence data. For more information on the LSTM layers supported in this release, see Supported Networks and Layers.

Deep Learning Targeting: Deploy deep learning networks to ARM Mali GPU processors

You can generate code for prediction from a pretrained convolutional neural network (CNN) and target the code to an embedded platform that uses an ARM Mali GPU processor. The code generator takes advantage of ARM Compute Library for computer vision and machine learning. The generated code implements a CNN that has the architecture, layers, and parameters specified in the input `SeriesNetwork` or `DAGNetwork` objects. For more information, see Code Generation for Deep Learning Networks Targeting ARM Mali GPUs.

For information on the networks and layers supported for code generation, see Supported Networks and Layers.

TensorRT Support: Support for NVIDIA TensorRT library on the Windows platform

In R2019b, you can take advantage of the NVIDIA low-latency, high-throughput TensorRT inference library for your deep learning applications and generate CUDA code on the Windows® platform. For information on the supported TensorRT version, see Installing Prerequisite Products. To set up your development computer for code generation, see Setting Up the Prerequisite Products. To generate CUDA code targeting the TensorRT libraries, see Code Generation for Deep Learning Networks by Using TensorRT.

Deep Learning Networks: Generate code for more networks

In R2019b, you can generate code for networks such as DeepLab-v3+, MobileNet-v2, ONNX™ (Open Neural Network Exchange), and Xception. For more information, see Supported Networks and Layers.

Deep Learning Layers: Generate code for more layers

Code generation with the CUDA Deep Neural Network library (cuDNN) supports these additional layers:

- Pixel classification layer by using generalized dice loss for semantic segmentation (`dicePixelClassificationLayer`)
- Exponential linear unit (ELU) layer (`eluLayer`)
- 2-D grouped convolutional layer (`groupedConvolution2dLayer`)
- Long short-term memory (LSTM) layer (`lstmLayer`)
- All output layers including custom classification or regression output layers created by using `nnet.layer.ClassificationLayer` or `nnet.layer.RegressionLayer`

- Sequence input layer (`sequenceInputLayer`)
- Hyperbolic tangent (tanh) layer (`tanhLayer`)
- Affine layer for the ONNX network that performs element-wise scaling of the input followed by an addition (`nnet.onnx.layer.ElementwiseAffineLayer`)
- Flatten layer for the ONNX network that flattens the spatial dimensions of the input tensor to the channel dimensions (`nnet.onnx.layer.FlattenLayer`)

Code generation with the NVIDIA TensorRT Library supports these additional layers:

- Clipped rectified linear unit (ReLU) layer (`clippedReluLayer`)
- Pixel classification layer using generalized dice loss for semantic segmentation (`dicePixelClassificationLayer`)
- Exponential linear unit (ELU) layer (`eluLayer`)
- 2-D grouped convolutional layer (`groupedConvolution2dLayer`)
- All output layers including custom classification or regression output layers created by using `nnet.layer.ClassificationLayer` or `nnet.layer.RegressionLayer`
- Hyperbolic tangent (tanh) layer (`tanhLayer`)
- Flatten layer for the ONNX network that flattens the spatial dimensions of the input tensor to the channel dimensions (`nnet.onnx.layer.FlattenLayer`)

For more information, see Supported Networks and Layers.

1-D reduction operations on the GPU

In R2019b, 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.

Workflow and generated code improvements

R2019b includes the following improvements in the generated code:

- Performance improvement in the code generated for the `cumsum` function.
- Variables and expression support for specifying dimensions in the kernel pragmas. For more information, see `coder.gpu.kernel`.

Code generation for more Image Processing Toolbox functions

- `bwconncomp`
- `bwlabel`
- `houghlines`
- `imadjust`
- `imhist`
- `imfill`
- `imreconstruct`

Code generation for more MATLAB functions

- `interp2`
- `min`
- `max`
- `rgb2gray`

Code generation for more Computer Vision Toolbox functions

- `selectStrongestBboxMulticlass`

Functionality being removed or changed

This release removes support for generating CUDA code by using CUDA toolkit version 8.

Compatibility Considerations

GPU Coder throws an error if the supported CUDA toolkit is not found on the development platform. For information on the supported compilers and libraries, see Installing Prerequisite Products.

New examples

This release adds the following examples:

- Code Generation for a Sequence-to-Sequence LSTM Network – Shows how to generate CUDA code for a long short-term memory (LSTM) network. The example generates a MEX application that makes predictions at each step of an input time series. This example uses accelerometer sensor data from a smartphone carried on the body and makes predictions on the activity of the

wearer. User movements are classified into one of five categories, namely dancing, running, sitting, standing, and walking.

- Deep Learning Prediction on ARM Mali GPU– Shows how to use the `cnncodegen` function to generate code for an image classification application that uses deep learning on ARM Mali GPUs. The example uses the MobileNet-v2 DAG network to perform image classification.

- QR Decomposition on an NVIDIA GPU Using cuSOLVER Libraries– Shows how to create a standalone CUDA executable that leverages the CUDA Solver library (cuSOLVER). The example uses a curve fitting application that mimics automatic lane tracking on a road to illustrate several topics, including:

  - Fitting an arbitrary-order polynomial to noisy data using matrix QR factorization.

  - Using the `coder.LAPACKCallback` class to provide the LAPACK library information for the code generator when generating standalone executables.

- Lane Detection on the GPU using `houghlines`– Shows how to generate CUDA MEX for a MATLAB function that can detect and output lane marker boundaries on an image. The example takes an RGB image as input and uses the `rgb2gray`, `ordfilt2`, `hough`, `houghpeaks`, and `houghlines` functions that are part of Image Processing Toolbox™ to produce the lane detected output image.

To see the full list of examples for GPU Coder, at the MATLAB command line, enter `doc gpucoder`.

R2019a

Version: 1.3

New Features

Bug Fixes

Deep Learning: Generate code for more layers

Code generation with the CUDA Deep Neural Network library (cuDNN) supports these additional layers:

- Layer that applies 2-D cropping to the input (`crop2dLayer`)
- One of the layers that allows the network to use features from earlier by making the features match the feature map size at the later layer (`YOLOv2ReorgLayer`)
- Output layer for YOLO v2 object detection network (`YOLOv2OutputLayer`).
- Transform layer for YOLO v2 object detection network (`YOLOv2TransformLayer`).
- Flatten activations into 1-D assuming C-style (row-major) order (`nnet.keras.layer.FlattenCStyleLayer`)
- Global average pooling layer for spatial data (`nnet.keras.layer.GlobalAveragePooling2dLayer`)
- Sigmoid activation layer (`nnet.keras.layer.SigmoidLayer`)
- Hyperbolic tangent activation layer (`nnet.keras.layer.TanhLayer`)
- Zero padding layer for 2-D input (`nnet.keras.layer.ZeroPadding2dLayer`)

Code generation with the NVIDIA TensorRT Library supports these additional layers:

- Layer that applies 2-D cropping to the input (`crop2dLayer`)
- Depth concatenation layer (`depthConcatenationLayer`)
- One of the layers that allows the network to use features from earlier by making the features match the feature map size at the later layer (`YOLOv2ReorgLayer`)
- Output layer for YOLO v2 object detection network (`YOLOv2OutputLayer`).
- Transform layer for YOLO v2 object detection network (`YOLOv2TransformLayer`).
- Flatten activations into 1-D assuming C-style (row-major) order (`nnet.keras.layer.FlattenCStyleLayer`)
- Global average pooling layer for spatial data (`nnet.keras.layer.GlobalAveragePooling2dLayer`)
- Sigmoid activation layer (`nnet.keras.layer.SigmoidLayer`)
- Hyperbolic tangent activation layer (`nnet.keras.layer.TanhLayer`)
- Zero padding layer for 2-D input (`nnet.keras.layer.ZeroPadding2dLayer`)

For more information on supported networks and layers, see Supported Networks and Layers.

TensorRT Support: Generate code that takes advantage of FP16 optimization in deep learning inference applications

Using TensorRT half-precision (also called FP16) arithmetic support in GPU Coder, the generated neural network code utilizes reduced memory usage compared to FP32 precision. This enables deployment of larger networks while taking less time than FP32. To enable half-precision, set the `DataType` property of the `coder.TensorRTConfig` object to `'fp16'`. Alternatively, you can also set the `DataType` property in the `Deep Learning` settings tab of the GPU Coder App.

Deep Learning: Generate code for more networks

In R2019a, you can generate code for networks such as fully convolutional neural networks (FCN), YOLOv2, and segmentation networks such as U-Net. For more information, see Deep Learning with GPU Coder.

CUDA optimized transpose function

In this release, you can use the `gpucoder.transpose` or `gpucoder.ctranspose` functions to perform efficient out-of-place non-conjugate or conjugate transpose on the GPU. This implementation uses shared memory for improved performance. For example,

```
A = rand(5,10);
B = gpucoder.transpose(A);
```

This function must not be used for inputs whose dimensions are greater than 2.

Support for unbounded variables

In this release, GPU Coder supports CUDA code generation for MATLAB code that contains unbounded variables.

Workflow and generated code quality improvements

R2019a includes these workflow and generated code quality improvements:

* Verify and set up the GPU code generation environment by using the `coder.checkGpuInstallApp`. The **Check GPU Install** app is an interactive tool to verify and set up the GPU code generation environment on your development computer and hardware platforms such as the NVIDIA DRIVE and Jetson. For more information, see Using the Check GPU Install App.

  You can also use the `coder.checkGpuInstall` function to perform the same checks from the MATLAB command line. In this release, the `coder.checkGpuInstall` function has been updated to accept a `coder.gpuEnvConfig` object. The `coder.gpuEnvConfig` object contains the configuration parameters that `coder.checkGpuInstall` uses to verify the GPU code generation environment. You can continue to use option flags with the `coder.checkGpuInstall` as in previous releases, but it is recommended to use the `coder.gpuEnvConfig` object as this functionality may be deprecated in a future release.
* Improved handling for loops with dynamic bound variables.
* CUDA profiling integration with the SIL interface.
* Support for the `gpuArray` function when performing SIL simulation.

Code generation for more MATLAB functions

* conv

Code generation for more Image Processing Toolbox functions

- hough
- houghpeaks
- ordfilt2

Code generation for more Computer Vision Toolbox functions

- rectifyStereoImages

Code generation for Statistics and Machine Learning Toolbox functions

In R2019a, you can generate optimized CUDA code for `pdist` and `pdist2` functions from the Statistics and Machine Learning Toolbox™. The supported distance input argument values are `'euclidean'`, `'squareeuclidean'`, `'seuclidean'`, `'cityblock'`, `'minkowski'`, `'chebychev'`, `'cosine'`, `'correlation'`, `'hamming'`, and `'jaccard'`.

Code generation for Wavelet Toolbox function

In R2019a, you can generate optimized CUDA code for the `cwt` Wavelet Toolbox function. For more information, see Supported Functions.

New examples

This release adds the following examples:

- Train and Deploy Fully Convolutional Networks for Semantic Segmentation – Shows how to train and deploy a fully convolutional semantic segmentation network on an NVIDIA GPU by using GPU Coder.
- Code Generation for Semantic Segmentation Network using U-net – Demonstrates code generation for an image segmentation application that uses U-Net, a popular deep learning network for image segmentation.
- Code Generation for Object Detection Using YOLO v2 – Demonstrates code generation for an object detector using a deep learning technique named you only look once (YOLO) v2.
- Top-Hat Filtering on Jetson TX2– Demonstrates code generation for a top-hat filtering application that removes uneven background illumination on NVIDIA Jetson TX2. This example requires the GPU Coder Support Package for NVIDIA GPUs.
- Deployment and Classification of Webcam Images on NVIDIA Jetson TX2 Platform– Demonstrates deployment and classification of webcam Images on NVIDIA Jetson TX2 Platform. This example requires the GPU Coder Support Package for NVIDIA GPUs.
- Edge Detection on GPU using Order statistic filters– Demonstrates code generation for edge detection algorithm on the GPU using order statistic filters.
- Image Denoising on the GPU using Median filter– Demonstrates code generation for an image denoising application on the GPU using median filter.

To see the full list of examples for GPU Coder, at the MATLAB command line, enter `doc gpucoder`.

R2018b

Version: 1.2

New Features

Bug Fixes

Compatibility Considerations

Deep Learning Retargetability: Deploy applications that use deep learning networks onto Intel MKL-DNN, and NVIDIA TensorRT by using the codegen function

When targeting Intel® MKL-DNN, and NVIDIA TensorRT libraries, GPU Coder now supports code generation for deep learning networks by using the `codegen` function. In previous releases, you could use the `codegen` function to target only NVIDIA cuDNN libraries.

To use the `codegen` function, create a GPU configuration object and set the `DeepLearningConfig.TargetLib` property to `'cudnn'`, `'mkldnn'`, or `'tensorrt'`. For more information, see Code Generation for Deep Learning Networks with TensorRT and Code Generation for Deep Learning Networks with MKL-DNN (MATLAB Coder).

Compatibility Considerations

In R2018b, you must install the MATLAB Coder Interface for Deep Learning Libraries and GPU Coder Interface for Deep Learning Libraries to generate code for deep learning networks.

In previous releases, you could target NVIDIA cuDNN libraries without specifying a target library in the code configuration object. In R2018b, you must set the `cfg.DeepLearningConfig = coder.DeepLearningConfig('cudnn')` configuration object to target cuDNN libraries.

Thrust Library Support: Generate GPU-accelerated code for sort and reduction operations by using the Thrust library

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. For more information, see Thrust Example.

Deep Learning Optimization: Improve performance and memory utilization through auto-tuning, layer fusion, and buffer minimization

When generating code for deep learning networks by using the cuDNN libraries, you can now take advantage of the auto-tuning functionality in the library to select an optimal convolutional algorithm. The convolutional algorithm selection is based on the input, kernel sizes, and memory availability resulting in improved performance. To control the auto-tuning functionality, use the `DeepLearningConfig.AutoTuning` property of the GPU code configuration object. This capability is available only when targeting cuDNN libraries and is enabled by default. For more information, see `coder.CuDNNConfig`.

In R2018b, the code generator uses layer fusion and double buffering techniques to generate optimized code for deep learning networks.

- Convolutional and Rectified Linear Unit (ReLU) layers are fused into `FusedConvRelu` layer.
- Convolutional and Batch normalization layers are also fused to a `convolutional` layer with modified weights and biases.
- Convolutional, Batch normalization layer, and Rectified Linear Unit (ReLU) layers are also fused as `FusedConvRelu` layer.

gpuArray Support: Use gpuArray arguments at the I/O of MEX targets

In R2018b, you can use `gpuArray` arguments as inputs and outputs to an entry-point function when generating CUDA MEX code. Because the `gpuArray` function copies the array to the GPU, the generated code contains fewer `cudaMemcpy` calls. To use this functionality, use `coder.Type` to represent the `gpuArray` type of an entry-point function input. For example, you can use `coder.typeof(rand(20),'Gpu',true)` or `coder.typeof(gpuArray(rand(20)))` to create a `gpuArray` type for code generation.

Support Package for NVIDIA GPUs: Target NVIDIA Jetson and DRIVE platforms

In R2018b, you can use the GPU Coder Support Package for NVIDIA GPUs to communicate with, deploy, and run CUDA code on NVIDIA platforms such as Jetson and DRIVE. To download the support package, use the Add-on Explorer. For more information on the supported workflows, see GPU Coder Support Package for NVIDIA GPUs.

Calling External CUDA Functions: Use GPU arguments that pass by reference when using coder.ceval

In R2018b, you can pass GPU arguments by reference when calling external CUDA functions with `coder.ceval`. To make `coder.ceval` pass arguments by reference, use the constructs `coder.ref`, `coder.rref`, and `coder.wref`.

Deep Learning Layers: Generate code for new network layers

In R2018b, you can now generate code for these layers:

- Dilated convolutional
- Variable-size I/O

Ease-of-use and traceability improvements

This release contains a new traceability report that highlights sections of MATLAB code that are running on the GPU, a new diagnosis report to analyze performance breakdown, and an integrated GPU profiling report to analyze execution profiles of the generated code.

You can use the traceability feature to understand how the code generator maps your algorithm to GPU kernels, debug issues in the generated code, and evaluate the quality of the generated code. For more information on using the traceability feature, see Trace Between MATLAB Code and Generated CUDA Code.

In R2018b, the code generation report has a new diagnostic section that analyzes performance issues with your MATLAB algorithm and categorizes them as kernel issues, memory issues, pragma issues, and design pattern issues. The report provides suggestions for resolving the issues so that you can generate more efficient CUDA code. To enable report generation, set the `GenerateReport` property in the code configuration object or enable the **Always create a code generation report** option in **Miore Settings ->Debugging** pane of the GPU Coder app.

For information on the GPU profiling report, see Analyze Execution Profiles of the Generated Code.

In R2018b, CUDA syntax highlighting in the MATLAB editor helps you identify the different CUDA language elements in the generated code. You can change syntax highlighting preferences. On the **Home** tab, in the **Environment** section, click **Preferences**. Select **MATLAB > Editor/Debugger > Language > CUDA**.

Code generation for more Image Processing Toolbox functions

In R2018b, you can generate optimized CUDA code for the `imresize` Image Processing Toolbox function. For more information, see Supported Functions.

Deep learning examples

This release adds two deep learning examples:

- Integrating Deep Learning with GPU Coder into Simulink – Demonstrates integration of the CUDA code generated for a deep learning network into the Simulink environment.
- Code Generation for Denoising Deep Neural Network – Shows how to generate CUDA code for a denoising convolutional neural network (DnCNN). You can use the denoising network to estimate noise in a noisy image, and then remove it to obtain a denoised image.
- Deep Learning Prediction with NVIDIA TensorRT – Shows how to generate CUDA code by using the TensorRT library.
- Deep Learning Prediction with Different Batch Sizes – Shows how to use different batch sizes when generating code for a deep learning network.

To see the full list of examples for GPU Coder, at the MATLAB command line, enter gpucoderexamples.

Functionality being removed or changed

Compatibility Considerations

- Specifying the C language for the generated code through the `TargetLang` property of `coder.config` will be removed in a future release.

Functionality	What Happens When You Use This Functionality?	Use This Instead
`TargetLang = 'C'` property of `coder.config` object.	You get a warning message.	`TargetLang = 'C++'` property of `coder.config` object.

- To perform code generation for deep learning networks, you must install the `GPU Coder Interface for Deep Learning Libraries` and `MATLAB Coder Interface for Deep Learning Libraries` support packages. To install these support packages, select the support package from the MATLAB **Add-Ons** menu.

Functionality	What Happens When You Use This Functionality?	Solution
Using `cnncodegen` or `codegen` functions to generate code for deep learning networks.	You get an error message.	To install the required support packages, follow the links in the error message.

R2018a

Version: 1.1

New Features

Bug Fixes

Directed Acyclic Graph (DAG) Networks: Generate CUDA code for deep learning networks with DAG topology

You can use GPU Coder in tandem with the Neural Network Toolbox™ to generate CUDA code for DAG networks. A DAG network is a neural network for deep learning that can have its layers arranged as a directed acyclic graph. You can use a pretrained DAG network or train one by using the Neural Network Toolbox. See, Supported Networks and Layers.

Deep Learning Layers: Generate CUDA code for popular networks such as GoogLeNet, ResNet, and SegNet

In R2018a, you can target generate CUDA code for popular convolutional neural networks such as GoogLeNet, ResNet, and SegNet. See, Supported Networks and Layers

TensorRT Support: Generate code that takes advantage of NVIDIA deep learning inference optimizer and run time

With TensorRT support in GPU Coder, you can take advantage of the NVIDIA low-latency, high throughput inference library for your deep learning applications on embedded platforms. For more information, see CNN Code Generation, and `cnncodegen`.

Multi-Platform Deep Learning Targeting: Deploy deep learning networks to Intel and ARM processors

Generate code that takes advantage of Intel Math Kernel Library for Deep Neural Networks (MKL-DNN) for Intel CPUs, and ARM Compute libraries for mobile platforms. For more information, see CNN Code Generation.

Code generation for Image Processing Toolbox functions

In R2018a, you can generate optimized code for Image Processing Toolbox functions such as `imerode`, `imdilate`, and `imwarp`. For more information, see Supported Functions.

Code generation for Computer Vision System Toolbox functions

Generate optimized CUDA code for the `matchfeatures` function. For more information, see Supported Functions.

Loop and kernel optimization

In R2018a, you can map while loops and dynamically bound for-loops to GPU kernels. This feature allows you to generate CUDA code containing kernels with variable and symbolic dimensions.

Deep learning examples

This release adds three deep learning examples:

- Pedestrian Detection – Demonstrates code generation for a pedestrian detection implementation that has several applications in the fields of autonomous driving, surveillance, and robotics.

- Traffic Sign Detection and Recognition – Demonstrates how to generate CUDA MEX code to detect traffic signs, suppress overlapping detections, and classify the detected traffic signs.
- Logo Recognition Network – Demonstrates code generation for a logo classification application that can recognize 32 logos under various lightning conditions and camera motions.

Use gpucoderexamples to see the full list of examples that ship with GPU Coder.

R2017b

Version: 1.0

CUDA C and C++ code Generation

Generate CUDA C and C++ code from MATLAB code. You can integrate the generated code into your project as source code, static libraries, or dynamic libraries. The generated code calls optimized NVIDIA CUDA libraries, including cuDNN, cuSolver, cuFFT, and cuBLAS. To generate CUDA code, you must have the following products:

- MATLAB
- GPU Coder
- MATLAB Coder
- Parallel Computing Toolbox™

For more information, see Getting Started with GPU Coder.

Deep Learning Network Support

You can use GPU Coder in tandem with the Neural Network Toolbox to generate CUDA code for deep learning networks. You can use the Neural Network Toolbox to create and train a neural network, or import pretrained networks like VGG, MNIST, AlexNET, YOLO. See, Deep Learning.

Image Processing Toolbox Support

GPU Coder supports CUDA code generation for many of the functions from MATLAB and the Image Processing Toolbox.

CUDA Kernel and memory Optimizations

GPU Coder performs program parallelism analysis to identify segments of code that run on the CPU and segments that run on the GPU. After this kernel partitioning and optimization is complete, GPU Coder performs memory optimization by analyzing the data dependency between the CPU and GPU partitions. GPU Coder also provides you pragmas and design patterns that can be used to generate optimized CUDA code.

MEX Function Generation for code Verification and Acceleration

With GPU Coder, you can also use the generated code within the MATLAB environment to accelerate computationally intensive portions of your MATLAB code. MEX functionality also allows you to verify the numerical correctness of the generated code.

Legacy CUDA 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.

Hardware Integration with NVIDIA Tegra

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

Code Profiling and Verification

By using GPU Coder with Embedded Coder®, you can verify the numerical behavior of the generated C/C++ code by using software-in-the-loop (SIL) execution.