WHITE PAPER

Generating CUDA Code from MATLAB: Accelerating Embedded Vision and Deep Learning Algorithms on GPUs

MathWorks®

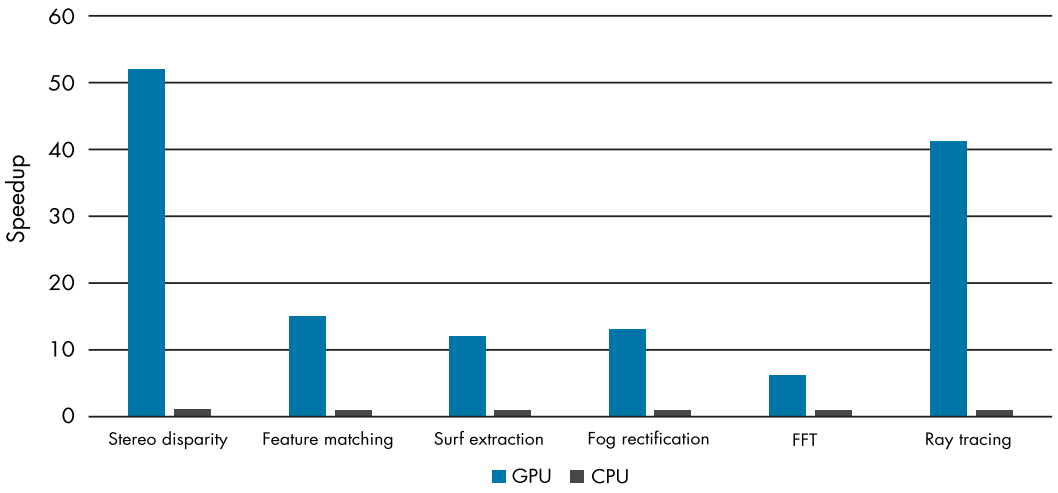
Introducing GPU Code Generation with MATLAB

GPU computing is contributing to the growth of deep learning applications because it enables parallel computations of data on a massive scale. GPU computing accelerates your applications by taking advantage of the data parallelism in your algorithms. This is key to meeting performance goals for computationally intensive algorithms in areas such as embedded vision and radar as well as deep learning.

GPU-accelerated computing follows a heterogeneous programming model; the portions of your application that can be parallelized are mapped to kernels that execute concurrently across the hundreds or thousands of parallel cores on the GPU, while the sequential portions of the code run on the CPU.

GPU Coder™ accelerates your existing *MATLAB*® algorithms on GPUs by automatically generating CUDA® code from MATLAB code for execution on NVIDIA® GPUs. The generated CUDA code is optimized for parallelism, while minimizing the overhead of data transfer between the CPU and GPU. During code generation, GPU Coder analyzes the data dependency between the CPU and GPU partitions. This analysis determines the minimum set of locations where data must be copied between the CPU and GPU. The memory bandwidth bottleneck between the CPU and the GPU is a key reason for algorithms to lose performance speed on GPUs; the ability to minimize data transfer results in significant performance gains. An automated workflow avoids the hassle and increased errors that can come from hand coding in CUDA.

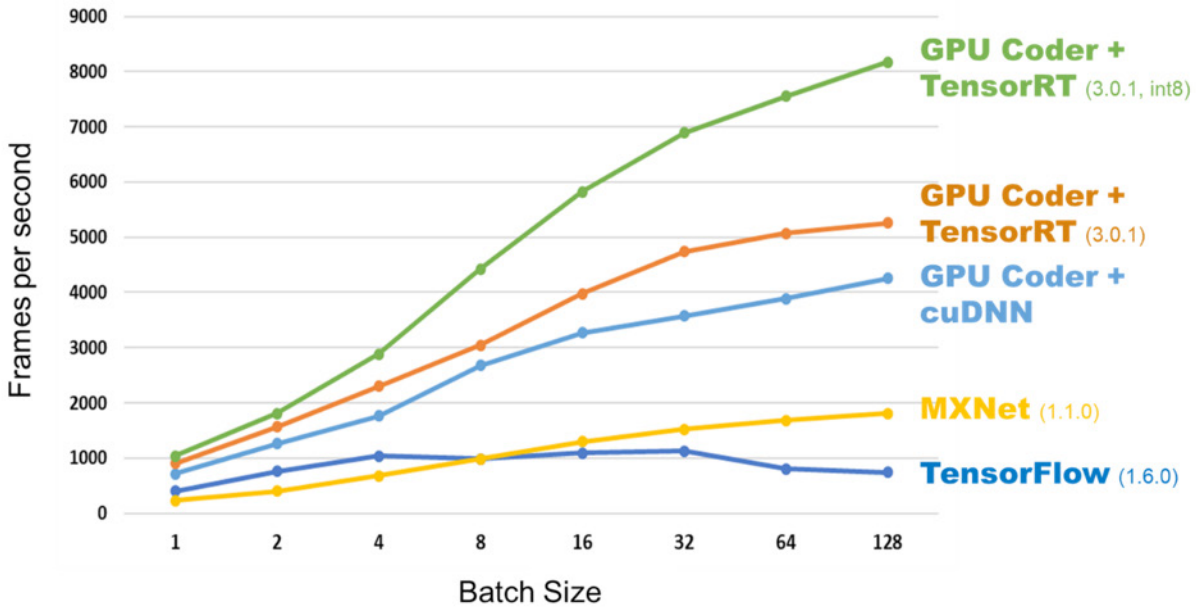
GPU Coder accelerates your algorithm on GPUs running on desktops, the cloud, or embedded devices, including NVIDIA Jetson® and NVIDIA Drive® platforms. Figures 1 and 2 highlight these performance improvements. Applications can realize up to two orders of magnitude in performance improvements for common image processing operations such as surf feature extraction, stereo disparity, and fog removal, and common signal processing operations such as FFT.



Testing Platform	MATLAB 2018a
CPU	Intel Xeon CPU E5-1650 v3 @ 3.50 GHz
GPU	NVIDIA Pascal TITAN V (Volta architecture)
CUDA	Version 9.0

Figure 1. Common image processing and signal processing performance benchmarks for GPU Coder–generated CUDA code running on GPUs compared with C code running on CPUs.

Figure 2 shows the inference performance of the CUDA code generated from GPU Coder compared with other popular deep learning frameworks. The code generated by GPU Coder runs five times faster than TensorFlow™ and twice as fast as Apache MXNet.



Testing Platform	MATLAB 2018a
CPU	Intel Xeon CPU E5-1650 v4 @ 3.60 GHz
GPU	NVIDIA Pascal TITAN Xp
cuDNN	v7

Figure 2. Deep learning performance benchmarks: AlexNet inference performance comparison of GPU Coder, TensorFlow, and MXNet running on NVIDIA Titan® Xp.

Note that a typical deep learning algorithm consists of user logic that calls one or more trained networks for inference. With GPU Coder, you can automatically generate code from the complete application, without having to do any manual coding. In the case of a very simple image classification task, for example, user logic might include preprocessing steps such as resizing the image and changing the color space, and postprocessing steps such as drawing a bounding box. GPU Coder generates code from the entire application (Figure 3).

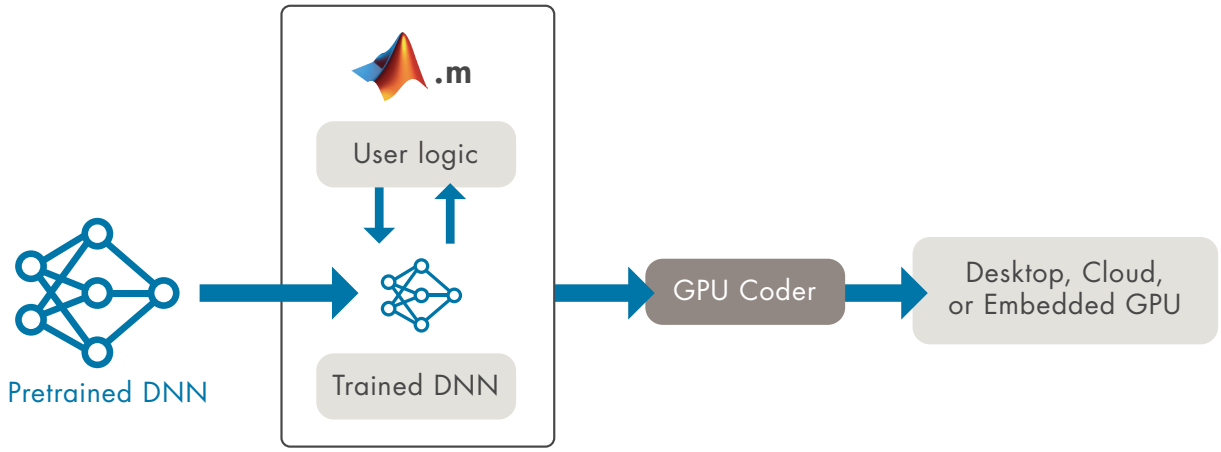


Figure 3. Generating code for the entire end-to-end application using GPU Coder.

High-Level Workflow

Figure 4 shows the typical workflow from algorithm design through code generation to deployment. You begin by preparing the MATLAB code for code generation, followed by testing the generated code to ensure it functionally behaves the same as the original MATLAB code. In the third step, you can generate code either as a MEX file (to replace the original MATLAB algorithm for acceleration purposes) or as CUDA code in the form of source code or a static or dynamic library that can be integrated into your larger CUDA project. An optional fourth step is to further optimize the MATLAB code to boost performance of the generated code.

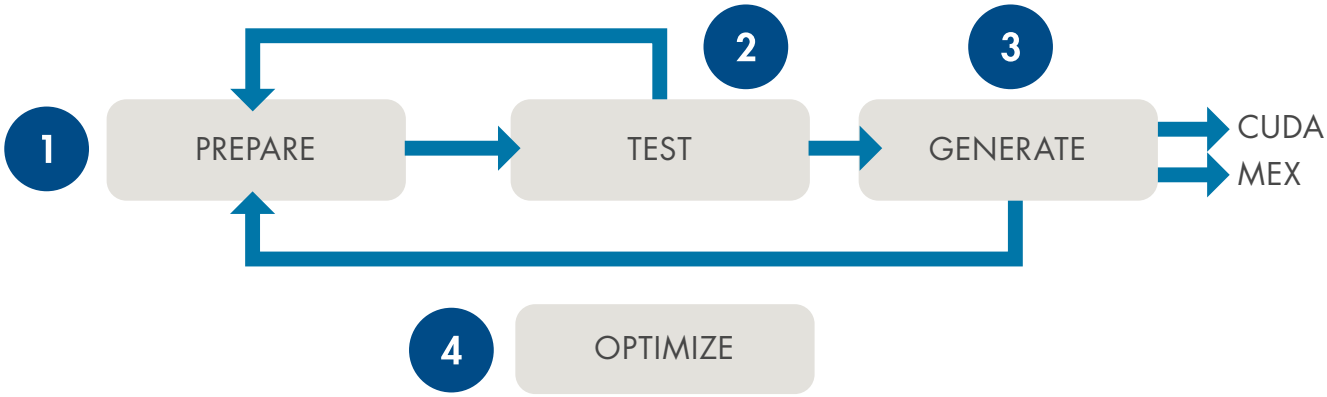


Figure 4. Workflow to generate CUDA code from MATLAB code.

Traffic Sign Detection Application

This section examines the code generation workflow in greater detail using a deep learning–based *traffic sign detection example*. The three basic steps to perform traffic sign detection and recognition are detection, non-maximal suppression (NMS), and recognition. You can start with a MATLAB algorithm to:

- Read and preprocess a test image
- Call the prediction method for the object detection network to detect the traffic sign
- Suppress overlapping detections using an NMS algorithm
- Call the prediction method for the recognition network to recognize the sign
- Display an output image with bounding boxes and corresponding detection labels

To generate CUDA code for this algorithm, you can start by preparing this MATLAB script for code generation.

Preparing MATLAB Code for GPU Code Generation

To prepare a MATLAB algorithm for code generation, you follow these steps:

1. **Separate the algorithm from the test bench**. Begin by separating the algorithm from the test bench and casting the algorithm code as a function. It is also good practice to separate your algorithm into the deep learning network inference function and the pre- and postprocessing functions.
2. **Identify computationally intensive portions of the algorithm**. Use the *MATLAB Profiler to profile* and identify hotspots in the algorithm. Isolate hotspots into separate functions to help with the next step.
3. **Direct GPU Coder to automatically map hotspots to CUDA kernels**. Insert `coder.gpu.kernelfun` pragmas at the beginning of hotspot functions. This directs GPU Coder to analyze and create the most efficient CUDA kernels from these computationally expensive functions.
4. **Ensure that the algorithm is compatible with code generation**. Use the GPU Coder app (Figure 5) to go through this process. The app begins by running a code generation readiness tool to detect any code generation compatibility issues, including unsupported functions and constructs:

- **Unsupported functions**: You will need to remove or replace *unsupported functions* from your MATLAB algorithm in order to generate code.
- **Unbounded variable-size data**: GPU Coder generates code from variables that are bounded. If the code generator cannot determine the size of an array or if it determines that the size changes, then the dimension is variable-size (Figure 6). GPU Coder issues a warning for unbounded variables or unbounded variable-size arrays. In this case, you need to specify an upper bound so that the coder can allocate the memory statically:
  - Specify upper bounds for variable-size inputs: Use the `coder.typeof` construct to specify upper bounds for variable-size inputs.
  - Specify upper bounds for local variables: To constrain the value of variables that specify the dimensions of variable-size arrays, use the assert function with relational operators or use the `coder.varsize` function to specify the upper bounds for all instances of a local variable in a function.

5. **Checking for potential run-time errors**. After checking code generation compatibility, we recommend performing run-time checks to detect and fix run-time errors. Run CPU run-time checks to confirm memory integrity, verify array bounds, and perform dimension checking. Run GPU run-time checks to identify register spills and confirm stack size conformance.

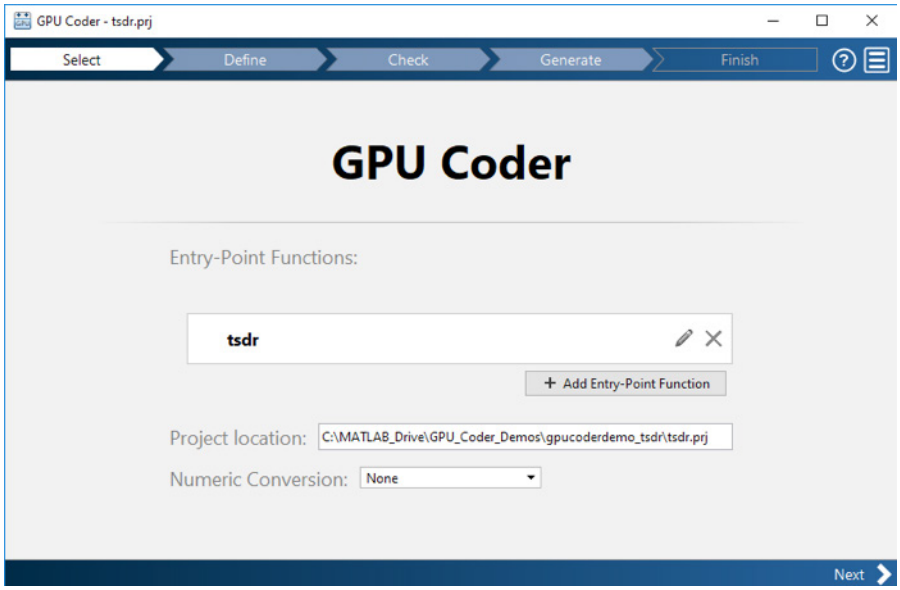


Figure 5. GPU Coder app.

Summary | All Messages (0) | Variables | Target Build Log

Order	Variable	Type	Size	Class	Complex
1	C	Output	1 x :100	double	No
2	A	Input	1 x :100	double	No
3	B	Input	1 x :?	double	No

Figure 6. The Size field showing the computed maximum size of an array. For variable-size arrays, a colon (:) indicates that a dimension is variable-size, and a question mark (?) indicates that the size is unbounded.

Preparing the Traffic Sign Detection MATLAB Script for Code Generation

Using the steps above, the first step is separate out the algorithmic code, now named the **`tsdr`** function, from the test bench script, **`tsdr _ test.m`**. When you profile the function in MATLAB, you see that there are no specific hotspots to isolate within the function. This means you can map the entire function to the GPU by inserting the **`coder.gpu.kernelfun`** pragma into the **`tsdr`** function. The GPU Coder app did not identify unsupported functions, but it did show that the function has an unbounded variable-size array. So you need to use **`coder.varsize`** to specify an upper bound (Figure 7). Run-time checks for both CPU and GPU complete successfully without any errors.

```
%% Run Non-Maximal Suppression on the detected bounding boxes
coder.varsize('selectedBbox',[98, 4],[1 0]);
[selectedBbox,~] = selectStrongestBbox(round(boxes),probs);
```

```
%% Recognition
persistent recognitionnet;
if isempty(recognitionnet)
     recognitionnet = coder.loadDeepLearningNetwork('RecognitionNet.mat', ...
          'Recognition');
end

coder.varsize('idx',98,1);
idx = zeros(size(selectedBbox,1),1);
inpImg = coder.nullcopy(zeros(48,48,3,size(selectedBbox,1)));
```

Figure 7. Specifying the upper bounds for the variable-size arrays using **`coder.varsize`**.

Testing Generated MEX

Now that the code is ready for code generation, you should first test the generated MEX within MATLAB to ensure it behaves functionally the same as the original MATLAB code. To do this, you generate a MEX function. MEX is a wrapper interface that lets you call the compiled binary in MATLAB and use it in place of the original MATLAB function to verify that it behaves in the same way as the original MATLAB function.

In the GPU Coder app, choose MEX as the build type to generate CUDA code (Figure 8) and compile it using NVIDIA's nvcc compiler into a MEX function. By calling the MEX function from the test bench in the verify code step, you can verify that the generated code provides the same functionality as the original MATLAB function.

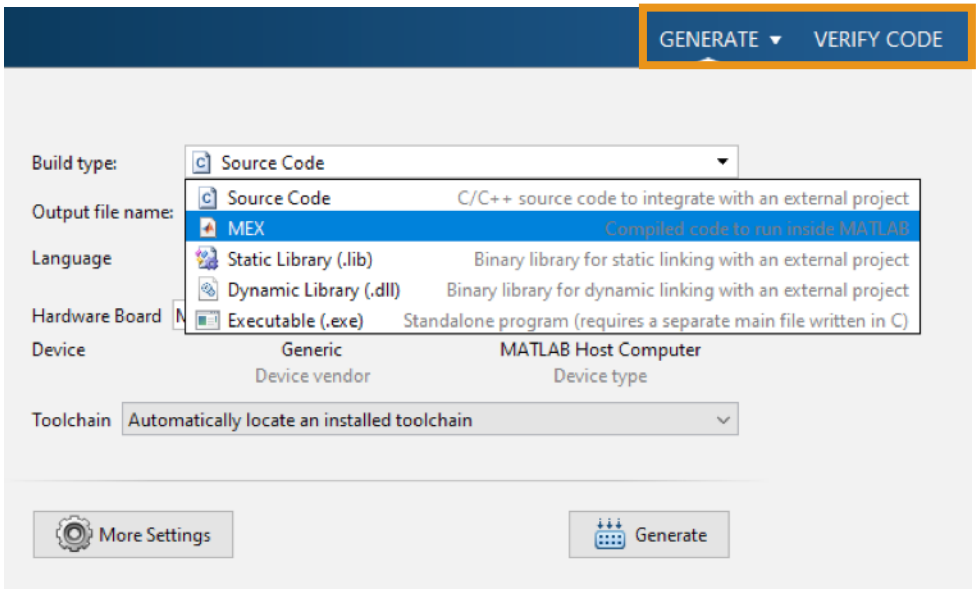


Figure 8. Selecting build targets in the GPU Coder app and the verify code step.

In this example application, you generate the MEX target and run the test with the MEX version in the verify code step to verify the behavior of the generated code.

By using GPU Coder with ***Embedded Coder®***, you can further verify the numerical behavior of the generated standalone C++ code using ***software-in-the-loop (SIL) execution*** to verify the generated code as deployed on your embedded GPU. This enables you to detect any differences between the MEX and the standalone production-quality code.

Generating Code as MEX or Source Code: Deploying on Desktop, Cloud, or Embedded GPUs

Once you have verified that the generated MEX function behaves as expected, you can generate standalone C++ code or a static library by simply changing the build type option. You can then integrate the generated library into a larger application in your development environment and deploy your algorithm to either the desktop or the cloud.

If your objective is to deploy onto embedded platforms such as NVIDIA Jetson or Drive, you can export the CUDA source code to the target and build it on the target. For NVIDIA Tegra® development boards such as the Jetson TX2, TX1, or TK1, another option is to cross-compile the generated code on your host machine and deploy it to the board by copying the executable to the target.

An important use case for generating code with the MEX build type is to accelerate the computational speed in MATLAB by calling the compiled binary running on the GPU instead of the native MATLAB function that needs to be interpreted.

Generating Code and Deploying the Traffic Sign Detection Application

In the traffic sign detection example application, the goal is to deploy the algorithm on the desktop. In this case, you choose the build type as source code and generate standalone C++ code that can be integrated into a main file and built in an executable. You can even go one step further and generate a static library that can be cross-compiled and deployed to a Jetson board, as is explained in ***an AlexNet example***.

Tips for Further Optimizing Your Code

If the generated code does not meet your performance requirements, you can take additional steps and optimizations to improve the performance of the generated code:

- **Using profile MEX**. To identify bottlenecks in the performance of the generated code, you can profile execution times for the generated MEX function by using the MATLAB Profiler. The profile for the generated code shows the number of calls and the time spent for each line of the corresponding MATLAB function. Certain bottlenecks could be addressed by making a few modifications to your MATLAB algorithm or by tuning the code generation options; we have listed a few techniques below.
- **Using the `coder.gpu.kernel` pragma**. If some parallel loops in your algorithm have not be parallelized in parts of the generated code, you can insert the **`coder.gpu.kernel`** pragma immediately before a loop to force GPU Coder to generate a kernel for the loop. Note this is an advanced maneuver and should be used with caution.

- **Using design patterns**. For certain algorithms where a given input element is accessed repeatedly for computing multiple neighboring output elements, you can use design patterns such as stencil operations (`gpucoder.stencilKernel`) or matrix-matrix operations (`gpucoder.matrixMatrixKernel`) to improve computation efficiency. These design patterns leverage the GPU's shared memory to boost performance. See the *design pattern documentation* for details.
- **Inserting custom CUDA code**. In scenarios where 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 integrate your custom code with the code generated by GPU Coder. See the *documentation on legacy code integration* for details.
- **Customizing code generation options**. You can customize the code generation configuration to further tune the code generation for your specific requirements with options such as:
  - Setting the memory allocation mode, where you choose between discrete and unified memory allocation
  - Enabling support for CUDA libraries such as cuBLAS and cuSOLVER
  - Specifying the compute capability of the target
  - Passing additional flags to the GPU compiler

Next Steps

- Explore: *GPU Code Generation from MATLAB*
- Download: *GPU Coder Trial*