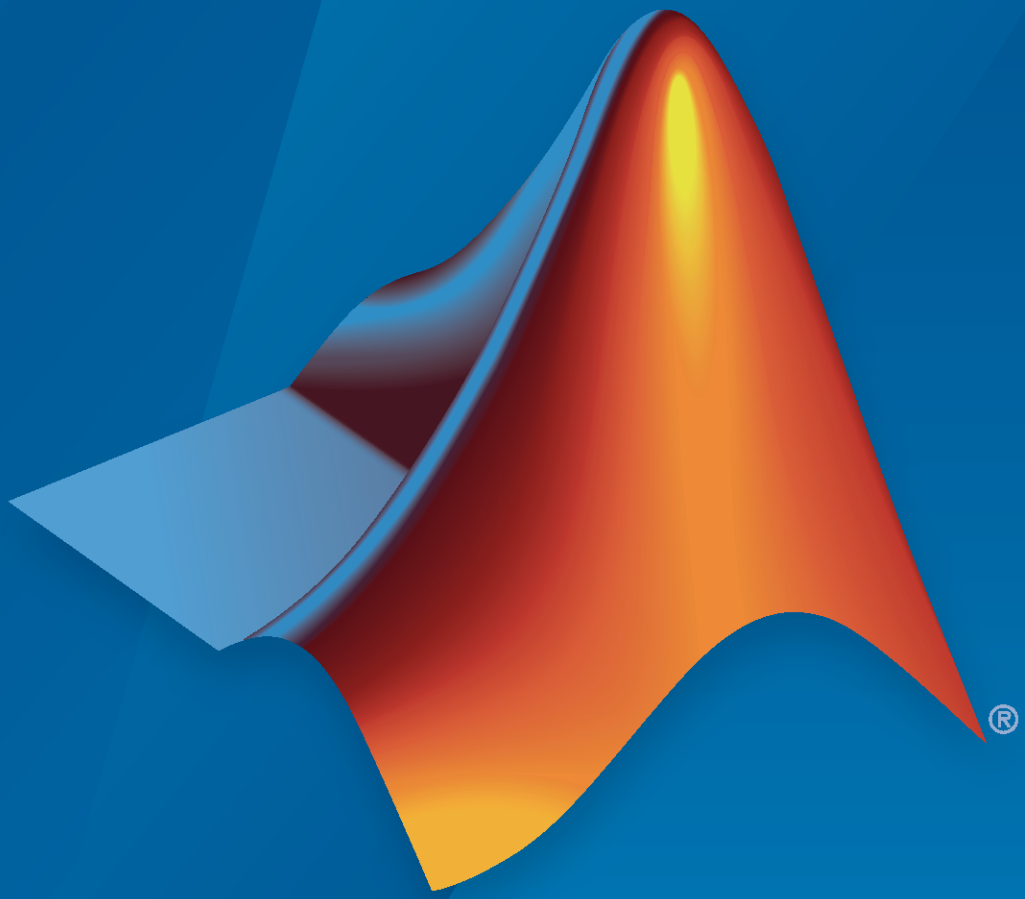
GPU Coder™

Getting Started Guide



MATLAB® & SIMULINK®

R2022a

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™ Getting Started Guide

Trademarks

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

Patents

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

Revision History

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

Contents

1 About GPU Coder

2 Tutorials

3 Verification

4 About GPU Code Generation

About GPU Coder

GPU Coder Product Description

Generate CUDA code for NVIDIA GPUs

GPU Coder™ generates optimized CUDA® code from MATLAB® code and Simulink® models. The generated code includes CUDA kernels for parallelizable parts of your deep learning, embedded vision, and signal processing algorithms. For high performance, the generated code calls optimized NVIDIA® CUDA libraries, including TensorRT, cuDNN, cuFFT, cuSolver, and cuBLAS. The code can be integrated into your project as source code, static libraries, or dynamic libraries, and it can be compiled for desktops, servers, and GPUs embedded on NVIDIA Jetson®, NVIDIA DRIVE®, and other platforms. You can use the generated CUDA within MATLAB to accelerate deep learning networks and other computationally intensive portions of your algorithm. GPU Coder lets you incorporate handwritten CUDA code into your algorithms and into the generated code.

When used with Embedded Coder®, GPU Coder lets you verify the numerical behavior of the generated code via software-in-the-loop (SIL) and processor-in-the-loop (PIL) testing.

Installing Prerequisite Products

To use GPU Coder for CUDA code generation, you must install and setup the following products. For setup instructions , see “Setting Up the Prerequisite Products” on page 1-7.

MathWorks Products and Support Packages

- MATLAB (required).
- MATLAB Coder™ (required).
- Parallel Computing Toolbox™ (required).
- Simulink (required for generating code from Simulink models).
- Computer Vision Toolbox™ (recommended).
- Deep Learning Toolbox™ (required for deep learning).
- Embedded Coder (recommended).
- Image Processing Toolbox™ (recommended).
- Simulink Coder (required for generating code from Simulink models).
- GPU Coder Interface for Deep Learning Libraries support package (required for deep learning).
- MATLAB Coder Support Package for NVIDIA Jetson and NVIDIA DRIVE Platforms (required for deployment to embedded targets such as NVIDIA Jetson and Drive).

For instructions on installing MathWorks® products, see the MATLAB installation documentation for your platform. If you have installed MATLAB and want to check which other MathWorks products are installed, enter `ver` in the MATLAB Command Window. To install the support packages, use Add-On Explorer in MATLAB.

If MATLAB is installed on a path that contains non 7-bit ASCII characters, such as Japanese characters, GPU Coder does not work because it cannot locate code generation library functions.

Third-Party Hardware

- NVIDIA GPU enabled for CUDA with a compatible graphics driver. For more information, see CUDA GPUs (NVIDIA).

  To see the CUDA compute capability requirements for code generation, consult the following table.

Target	Compute Capability
CUDA MEX	See “GPU Support by Release”.
Source code, static or dynamic library, and executables	3.2 or higher.
Deep learning applications in 8-bit integer precision	6.1, 7.0 or higher.
Deep learning applications in half-precision (16-bit floating point)	5.3, 6.0, 6.2 or higher.

- ARM® Mali graphics processor.

  For the Mali device, GPU Coder supports code generation for only deep learning networks.

Third-Party Software

Required

C/C++ Compiler:

Linux®	Windows®
GCC C/C++ compiler. For supported versions, see Supported and Compatible Compilers.	Microsoft® Visual Studio® 2013
	Microsoft Visual Studio 2015
	Microsoft Visual Studio 2017
	Microsoft Visual Studio 2019

Optional

For CUDA MEX, the code generator uses the NVIDIA compiler and libraries installed with MATLAB. Standalone code (static library, dynamically linked library, or executable program) generation has additional software requirements.

Software Name	Information
CUDA Toolkit	GPU Coder has been tested with CUDA Toolkit v9.x-v11.2. To download the CUDA Toolkit, see CUDA Toolkit Archive (NVIDIA).
NVIDIA Nsight systems	Generate an execution profiling report for the generated CUDA code. The report provides metrics that help you analyze your application algorithms and identify opportunities to optimize performance. GPU Coder has been tested with Nsight 2021.1.1
NVIDIA CUDA deep neural network library (cuDNN) for NVIDIA GPUs	For the host GPU device, GPU Coder has been tested with cuDNN v8.1.1. To download cuDNN, see cuDNN (NVIDIA).
NVIDIA TensorRT high performance inference optimizer and runtime library	For the host GPU device, GPU Coder has been tested with TensorRT v7.2.3. To download TensorRT, see TensorRT (NVIDIA).
ARM Compute Library for Mali GPUs	GPU Coder has been tested with v19.05. For more information, see Compute Library (ARM).

Software Name	Information
Open Source Computer Vision Library (OpenCV)	Required for deep learning examples. For examples targeting NVIDIA GPUs on the host development computer, use OpenCV v3.1.0. For examples targeting ARM GPUs, use OpenCV v2.4.9 on the ARM target hardware. For more information, see OpenCV.

Tips

General

- On Windows, a space or special character in the path to the tools, compilers, and libraries can create issues during the build process. You must install third-party software in locations that does not contain spaces or change Windows settings to enable creation of short names for files, folders, and paths. For more information, see *Using Windows short names* solution in MATLAB Answers.
- You must download the OpenCV source and build the libraries. The examples require separate libs such as, `opencv_core.lib` and `opencv_video.lib`. For more information, refer to the OpenCV documentation.

CUDA Toolkit

- The NVIDIA `nvcc` compiler relies on tight integration with the host development environment, including the host compiler and runtime libraries. It is recommended that you follow the CUDA Toolkit documentation for detailed information on compiler, libraries, and other platform specific requirements. See, CUDA Toolkit Documentation (NVIDIA).
- It is recommended to select the default installation options that includes `nvcc` compiler, `cuFFT`, `cuBLAS`, `cuSOLVER`, Thrust libraries, and other tools.
- The `nvcc` compiler supports multiple versions of GCC and therefore you can generate CUDA code with other versions of GCC. However, there may be compatibility issues when executing the generated code from MATLAB as the C/C++ run-time libraries that are included with the MATLAB installation are compiled for only the supported version of GCC.
- The “Analyze Execution Profiles of the Generated Code” workflow depends on the `nvprof` tool from NVIDIA. From CUDA Toolkit v10.1 onwards, NVIDIA restricts access to performance counters to only admin users. To enable GPU performance counters to be used by all users, see the instructions provided in Permission issue with Performance Counters (NVIDIA).
- GPU Coder does not support generating CUDA code by using CUDA Toolkit version 8.

Deep Learning

- Other versions of these deep learning libraries may have compatibility issues with all the features GPU Coder supports this release.

NVIDIA Embedded Targets

- On the target platforms, use the JetPack or the DriveInstall software appropriate for your board to install all the libraries. For more information, see “Install and Setup Prerequisites for NVIDIA Boards” (MATLAB Coder Support Package for NVIDIA Jetson and NVIDIA DRIVE Platforms).

ARM Mali

- This library must be installed on the ARM target hardware. Do not use a prebuilt library because it might be incompatible with the compiler on the ARM hardware. Instead, build the library from the source code. Build the library on either your host machine or directly on the target hardware. See instructions for building the library on GitHub®. You can also find information on building the library for CPUs in this post on MATLAB answers.
- When building the Compute Library, enable OpenCL support in the build options. See the ARM Compute Library documentation for instructions. OpenCL library (v1.2 or higher) on the ARM target hardware. See the ARM Compute Library documentation for version requirements. After the build is complete, rename the `build` folder containing the libraries as `lib`. Additionally, copy the OpenCL libraries present in the `build/opencl-1.2-stubs` folder into the `lib` folder. These steps are required so that the generated makefile can locate the libraries when building the generated code on the target hardware.

See Also

Apps
GPU Coder | GPU Environment Check

Functions
`codegen` | `coder.checkGpuInstall`

Objects
`coder.gpuEnvConfig`

More About

- “Setting Up the Prerequisite Products” on page 1-7
- “The GPU Environment Check and Setup App” on page 1-12
- “Code Generation by Using the GPU Coder App” on page 2-2
- “Code Generation Using the Command Line Interface” on page 2-16
- “Code Generation for Deep Learning Networks by Using cuDNN”
- “Code Generation for Deep Learning Networks by Using TensorRT”
- “Code Generation for Deep Learning Networks Targeting ARM Mali GPUs”

Setting Up the Prerequisite Products

In this section...

To use GPU Coder for CUDA code generation, install the products specified in “Installing Prerequisite Products” on page 1-3.

MEX Setup

When generating CUDA MEX with GPU Coder, the code generator uses the NVIDIA compiler and libraries included with MATLAB. Depending on the operating system on your development computer, you only need to set up the MEX code generator.

Windows Systems

If you have multiple versions of Microsoft Visual Studio compilers for the C/C++ language installed on your Windows system, MATLAB selects one as the default compiler. If the selected compiler is not compatible with the version supported by GPU Coder, change the selection. For supported Microsoft Visual Studio versions, see “Installing Prerequisite Products” on page 1-3.

To change the default compiler, use the `mex -setup C++` command. When you call `mex -setup C++`, MATLAB displays a message with links to set up a different compiler. Select a link and change the default compiler for building MEX files. The compiler that you choose remains the default until you call `mex -setup C++` to select a different default. For more information, see “Change Default Compiler”. The `mex -setup C++` command changes only the C++ language compiler. You must also change the default compiler for C by using `mex -setup C`.

Linux Platform

MATLAB and the CUDA Toolkit support only the GCC/G++ compiler for the C/C++ language on Linux platforms. For supported GCC/G++ versions, see “Installing Prerequisite Products” on page 1-3.

Environment Variables

Standalone code (static library, dynamically linked library, or executable program) generation has additional set up requirements. GPU Coder uses environment variables to locate the necessary tools, compilers, and libraries required for code generation.

Note On Windows, a space or special character in the path to the tools, compilers, and libraries can create issues during the build process. You must install third-party software in locations that does not contain spaces or change Windows settings to enable creation of short names for files, folders, and paths. For more information, see *Using Windows short names* solution in MATLAB Answers.

Platform	Variable Name	Description
Windows	`CUDA_PATH`	Path to the CUDA Toolkit installation. For example: `C:\Program Files\NVIDIA GPU Computing Toolkit\CUDA\v11.2\`
	`NVIDIA_CUDNN`	Path to the root folder of cuDNN installation. The root folder contains the bin, include, and lib subfolders. For example: `C:\Program Files\NVIDIA GPU Computing Toolkit\CUDA\v11.2\`
	`NVIDIA_TENSORRT`	Path to the root folder of TensorRT installation. The root folder contains the bin, data, include, and lib subfolders. For example: `C:\Program Files\NVIDIA GPU Computing Toolkit\CUDA\v11.2\TensorRT\`
	`OPENCV_DIR`	Path to the build folder of OpenCV on the host. This variable is required for building and running deep learning examples. For example: `C:\Program Files\opencv\build`
	`PATH`	Path to the CUDA executables. Generally, the CUDA Toolkit installer sets this value automatically. For example: `C:\Program Files\NVIDIA GPU Computing Toolkit\CUDA\v11.2\bin`
		Path to the `cudnn.dll` dynamic library. The name of this library may be different on your installation. For example: `C:\Program Files\NVIDIA GPU Computing Toolkit\CUDA\v11.2\bin`
		Path to the `nvinfer*` dynamic libraries of TensorRT. The name of this library may be different on your installation. For example: `C:\Program Files\NVIDIA GPU Computing Toolkit\CUDA\v11.2\TensorRT\lib`

Platform	Variable Name	Description
		Path to the `nsys` executable of NVIDIA Nsight systems. For example: `C:\Program Files\NVIDIA Corporation\Nsight Systems 2021.1.1\target-windows-x64`
		Path to the Dynamic-link libraries (DLL) of OpenCV. This variable is required for running deep learning examples. For example: `C:\Program Files\opencv\build\x64\vc15\bin`
Linux	`PATH`	Path to the CUDA Toolkit executable. For example: `/usr/local/cuda-11.2/bin`
		Path to the `nsys` executable of NVIDIA Nsight systems. For example: `/usr/local/Nsight Systems 2021.1.1/target-linux-x64`
		Path to the OpenCV libraries. This variable is required for building and running deep learning examples. For example: `/usr/local/lib/`
		Path to the OpenCV header files. This variable is required for building deep learning examples. For example: `/usr/local/include/opencv`
	`LD_LIBRARY_PATH`	Path to the CUDA library folder. For example: `/usr/local/cuda-11.2/lib64`
		Path to the cuDNN library folder. For example: `/usr/local/cuda-11.2/lib64/`
		Path to the TensorRT library folder. For example: `/usr/local/cuda-11.2/TensorRT/lib/`

Platform	Variable Name	Description
		Path to the ARM Compute Library folder on the target hardware. For example: `/usr/local/arm_compute/lib/` Set `LD_LIBRARY_PATH` on the ARM target hardware.
	`NVIDIA_CUDNN`	Path to the root folder of cuDNN library installation. For example: `/usr/local/cuda-11.2/`
	`NVIDIA_TENSORRT`	Path to the root folder of TensorRT library installation. For example: `/usr/local/cuda-11.2/TensorRT/`
	`ARM_COMPUTELIB`	Path to the root folder of the ARM Compute Library installation on the ARM target hardware. Set this value on the ARM target hardware. For example: `/usr/local/arm_compute`

Verify Setup

To verify that your development computer has all the tools and configuration needed for GPU code generation, use the `coder.checkGpuInstall` function. This function performs checks to verify if your environment has the all third-party tools and libraries required for GPU code generation. You must pass a `coder.gpuEnvConfig` object to the function. This function verifies the GPU code generation environment based on the properties specified in the given configuration object.

You can also use the equivalent GUI-based application that performs the same checks and can be launched using the command, **Check GPU Install**.

In the MATLAB Command Window, enter:

```
gpuEnvObj = coder.gpuEnvConfig;
gpuEnvObj.BasicCodegen = 1;
gpuEnvObj.BasicCodeexec = 1;
gpuEnvObj.DeepLibTarget = 'tensorrt';
gpuEnvObj.DeepCodeexec = 1;
gpuEnvObj.DeepCodegen = 1;
results = coder.checkGpuInstall(gpuEnvObj)
```

The output shown here is representative. Your results might differ.

```
Compatible GPU           : PASSED
CUDA Environment         : PASSED
    Runtime   : PASSED
```

```
    cuFFT     : PASSED
    cuSOLVER  : PASSED
    cuBLAS    : PASSED
cuDNN Environment        : PASSED
TensorRT Environment     : PASSED
Basic Code Generation    : PASSED
Basic Code Execution     : PASSED
Deep Learning (TensorRT) Code Generation: PASSED
Deep Learning (TensorRT) Code Execution: PASSED

results =

  struct with fields:

                gpu: 1
               cuda: 1
              cudnn: 1
           tensorrt: 1
       basiccodegen: 1
      basiccodeexec: 1
        deepcodegen: 1
       deepcodeexec: 1
   tensorrtdatatype: 1
          profiling: 0
```

See Also

Apps
GPU Coder | GPU Environment Check

Functions
`codegen` | `coder.checkGpuInstall`

Objects
`coder.gpuEnvConfig`

More About

- “Installing Prerequisite Products” on page 1-3
- “The GPU Environment Check and Setup App” on page 1-12
- “Code Generation by Using the GPU Coder App” on page 2-2
- “Code Generation Using the Command Line Interface” on page 2-16
- “Code Generation for Deep Learning Networks by Using cuDNN”
- “Code Generation for Deep Learning Networks by Using TensorRT”

The GPU Environment Check and Setup App

The **GPU Environment Check** app is an interactive tool to verify and set up the GPU code generation environment on your development computer and embedded hardware platforms such as the NVIDIA DRIVE and Jetson.

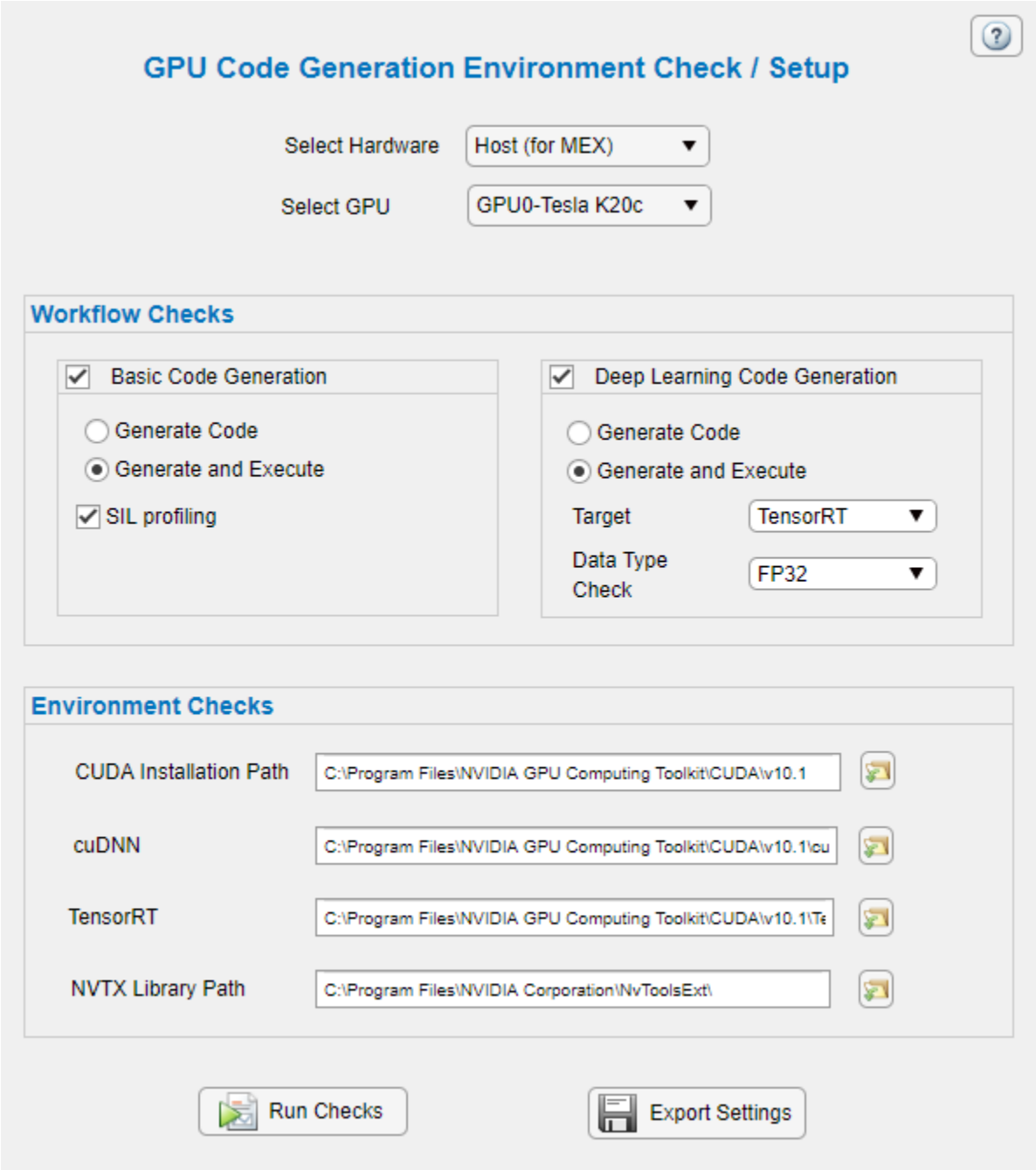
To start the app, in the MATLAB Command Window, enter:

```
gpucoderSetup
```

Using the **GPU Environment Check** app, you can:

- Verify the host development computer environment for the NVIDIA compilers and libraries necessary for GPU code generation.
- Perform basic code generation and test the execution of the generated code on the GPU device in the host computer. The tests validate code execution by comparing the results with MATLAB simulation.
- Perform deep learning code generation and execution tests on the development computer. You can target the NVIDIA cuDNN or TensorRT libraries. Requires the GPU Coder Interface for Deep Learning Libraries support package.
- Connect to embedded NVIDIA boards such as DRIVE and Jetson to perform code generation and execution tests. Requires the MATLAB Coder Support Package for NVIDIA Jetson and NVIDIA DRIVE Platforms.
- Specify the location of the libraries and generate a MATLAB script that sets up the environment variables required by GPU Coder.

Note The `gpucoderSetup` app generates a report file in the current folder. If you do not have write permissions in the current folder, before running the app, change the folder by using the MATLAB `cd` command.



Hardware Setup

The **Check/Setup** panel of the app provides drop-down lists that enable you to select a GPU device on the host development computer or hardware platforms such as the NVIDIA DRIVE and Jetson.

Option	Values	Description
Select Hardware	`Host (for MEX)`	Perform code generation, code execution, and environment checks on the host development computer. The app generates CUDA MEX to perform tests. If your development computer has multiple GPU devices, use the **Select GPU** option to select an appropriate GPU device.
	`Drive`	Perform code generation and code execution checks on an NVIDIA DRIVE target platform. After installing the MATLAB Coder Support Package for NVIDIA Jetson and NVIDIA DRIVE Platforms, use the **Board Settings** panel to specify connection parameters for the target.
	`Jetson`	Perform code generation and code execution checks on an NVIDIA Jetson target platform. After installing the MATLAB Coder Support Package for NVIDIA Jetson and NVIDIA DRIVE Platforms, use the **Board Settings** panel to specify connection parameters for the target.
Select GPU	*GPU<idx>-<device name>*	Select the GPU device to run tests. When there are multiple devices, the first device is the default. This option is visible only when the **Select Hardware** option is set to `Host (for MEX)`. For DRIVE or Jetson hardware, use the **GPU Device ID** option available in the **Board Settings** panel to select a specific GPU device.

Board Settings

Specify the connection parameters for hardware platforms such as the NVIDIA DRIVE and Jetson. The app uses the `jetson` or `drive` functions of the MATLAB Coder Support Package for NVIDIA Jetson and NVIDIA DRIVE Platforms to 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. The target platform must be on the same network as the host computer. Alternatively, you can use an Ethernet crossover cable to connect the board directly to the host computer. For more information on requirements, setup, and configuration steps for your NVIDIA boards, see “Install and Setup Prerequisites for NVIDIA Boards” (MATLAB Coder Support Package for NVIDIA Jetson and NVIDIA DRIVE Platforms).

Option	Description
Device Address	IP address or host name of the hardware. For example, `169.254.0.2` or `gpucoder-tegratx2-name` To use the host name, you must connect an Ethernet cable to the Ethernet port of the board. Then, use Linux commands to configure the hardware IP address and associate the host name with the IP address.
Username	Valid Linux user name for the operating system on the board.
Password	Valid password for the Linux user name specified.
Execution Timeout	Specify the time in seconds that the app waits for before validating the execution tests on the target. The default value is 10 seconds.
GPU Device ID	In a multi GPU environment such as NVIDIA Drive platforms, specify the CUDA GPU device to target.

Workflow Checks

There are two types of workflow checks that you can perform by using the app:

Basic code generation and execution tests on the development computer. These tests validate code execution by comparing the results with MATLAB simulation.

Option	Description
Generate Code	Test basic code generation and building. This test requires a valid CUDA code generation environment on the specified hardware.
Generate Code and Execute	Test basic code generation, building, and execution on the device in **Specified Hardware**. This test requires a valid CUDA code generation environment and GPU device on the specified hardware.

Option	Description
SIL Profiling	Perform basic SIL profiling tests on the host computer.

Deep learning code generation and execution tests on the development computer. You can target the cuDNN or TensorRT libraries.

Option	Description
Generate Code	Test deep learning code generation and building. This test requires a valid CUDA code generation environment on the specified hardware.
Generate Code and Execute	Test deep learning code generation, building, and execution on the device in **Specified Hardware**. This test requires a valid CUDA code generation environment and GPU device on the specified hardware.
Target	Specify the deep learning library to generate code for. Valid options are `cuDNN` or `TensorRT`.
Data Type Check	Specify the precision of the inference computations in supported layers. To perform inference in 32-bit floats, use `'FP32'`. For half-precision, use `'FP16'`. For 8-bit integer, use `'INT8'`. Default value is `'FP32'`. For compute capability requirements, see “Third-Party Hardware” on page 1-3.

Environment Checks

Specify the location of the libraries for checking the CUDA development environment on your host computer. Generate a MATLAB script `gpuEnvSettings.m` that sets up the environment variables required by GPU Coder. For more information, see “Setting Up the Prerequisite Products” on page 1-7.

Option	Description
CUDA Installation Path	Path to the CUDA Toolkit installation. For example: `/usr/local/cuda-10.1/bin`
cuDNN	Path to the root folder of cuDNN library installation. For example: `/usr/local/cuDNN/`

Option	Description
TensorRT	Path to the root folder of TensorRT library installation. For example: `/usr/local/TensorRT/`
NVTX Library Path	Path to the `nvtx` libraries required for profiling. To enable this item, select **SIL Profiling**. On a standard CUDA Toolkit installation, this path is usually the CUDA library folder. For example: `/usr/local/cuda-10.1/lib64`

GPU Code Generation Environment Check Report

When you select **Run Checks**, the `gpucoderSetup` app performs environment, code generation, and other checks based on the settings that you have selected. It then generates `gpucoderSetupReport` report that indicates if a test has passed and provides additional information for tests that have failed. A HTML version of the report of the same name is created in the current folder.

GPU Code Generation Environment Check Report

Results for Host GPU : TITAN Xp

ENVIRONMENT CHECKS

Check	Result	Message
Compatible GPU	✔	
CUDA Environment a. Runtime ✔ b. cuFFT ✔ c. cuSOLVER ✔ d. cuBLAS ✔	✔	
cuDNN Environment	✔	
TensorRT Environment	✔	
Profiling Environment	✔	

BASIC CODE GENERATION CHECKS

Check	Result	Message
Basic Code Generation	✔	
Basic Code Execution	✔	

DEEP LEARNING CODE GENERATION CHECKS - TensorRT

See Also

Apps
GPU Coder | GPU Environment Check

Functions
`codegen` | `coder.checkGpuInstall`

Objects
`coder.gpuEnvConfig`

More About

- “Installing Prerequisite Products” on page 1-3
- “Setting Up the Prerequisite Products” on page 1-7
- “Code Generation by Using the GPU Coder App” on page 2-2
- “Code Generation Using the Command Line Interface” on page 2-16
- “Code Generation for Deep Learning Networks by Using cuDNN”
- “Code Generation for Deep Learning Networks by Using TensorRT”

Tutorials

Code Generation by Using the GPU Coder App

The easiest way to create CUDA kernels is to place the `coder.gpu.kernelfun` pragma into your primary MATLAB function. The primary function is also known as the top-level or entry-point function. When GPU Coder encounters the `kernelfun` pragma, it attempts to parallelize all the computation within this function and then maps it to the GPU. For more information about GPU kernels, see “GPU Programming Paradigm” on page 4-2.

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.
- Define function input properties.
- Check for code generation readiness and run-time issues.
- Specify code generation properties.
- Generate CUDA code by using the GPU Coder app.

Tutorial Prerequisites

This tutorial requires the following products:

- MATLAB
- MATLAB Coder
- GPU Coder
- C++ compiler
- NVIDIA GPU enabled for CUDA
- CUDA Toolkit and driver
- Environment variables for the compilers and libraries. For more information, see “Environment Variables” on page 1-7.

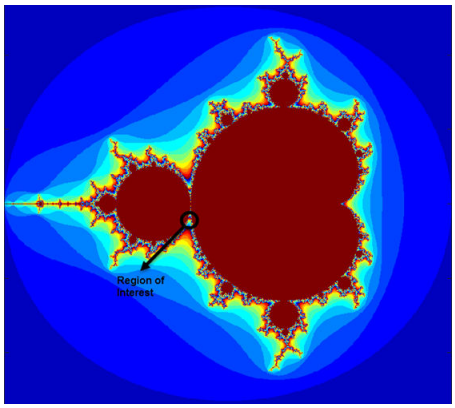
Example: The Mandelbrot Set

Description

The Mandelbrot set is the region in the complex plane consisting of the values z_0 for which the trajectories defined by this equation remain bounded at $k \to \infty$.

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

The overall geometry of the Mandelbrot set is shown in the figure. This view does not have the resolution to show the richly detailed structure of the fringe just outside the boundary of the set. At increasing magnifications, the Mandelbrot set exhibits an elaborate boundary that reveals progressively finer recursive detail.



Algorithm

For this tutorial, pick a set of limits that specify a highly zoomed part of the Mandelbrot set in the valley between the main cardioid and the *p*/*q* bulb to its left. A 1000-by-1000 grid of real parts (*x*) and imaginary parts (*y*) is created between these two limits. The Mandelbrot algorithm is then iterated at each grid location. An iteration number of 500 renders the image in full resolution.

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

This tutorial uses an implementation of the Mandelbrot set by using standard MATLAB commands running on the CPU. This calculation is vectorized such that every location is updated simultaneously.

Tutorial Files

Create a MATLAB function called `mandelbrot_count.m` with the following lines of code. This code is a baseline 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. Later in this tutorial, you modify this file to make it suitable for code generation.

```
function count = mandelbrot_count(maxIterations,xGrid,yGrid)
% 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 a MATLAB script called `mandelbrot_test.m` with the following lines of code. The script generates a 1000-by-1000 grid of real parts (*x*) and imaginary parts (*y*) between the limits specified

by `xlim` and `ylim`. It also calls the `mandelbrot_count` 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);

%% Mandelbrot computation in MATLAB
count = mandelbrot_count(maxIterations,xGrid,yGrid);

% Show
figure(1)
imagesc(x,y,count);
colormap([jet();flipud(jet());0 0 0]);
axis off
title('Mandelbrot set with MATLAB');
```

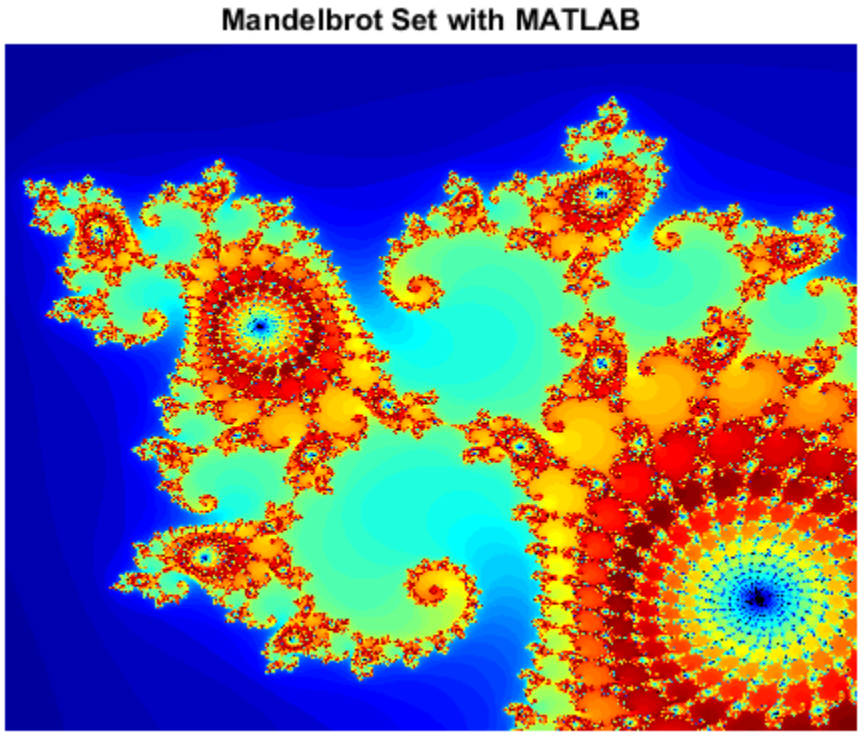
Run the Original MATLAB Code

Run the Mandelbrot Example

Before making the MATLAB version of the Mandelbrot set algorithm suitable for code generation, you can test the functionality of the original code.

1. Change the current MATLAB working folder to the location that contains `mandelbrot_count.m` and `mandelbrot_test.m`. GPU Coder places generated code in this folder. Change your current working folder if you do not have full access to this folder.
2. Run the `mandelbrot_test` script.

The test script runs and shows the geometry of the Mandelbrot within the boundary set by the variables `xlim` and `ylim`.



Prepare MATLAB Code for Code Generation

Before you generate code with GPU Coder, check for coding issues in the original MATLAB code.

Check for Issues at Design Time

There are two tools that help you detect code generation issues at design time:

- Code Analyzer tool
- Code generation readiness tool

The Code Analyzer is a tool incorporated into the MATLAB Editor that continuously checks your code as you enter it. The Code Analyzer reports issues and recommends modifications to maximize performance and maintainability of your code. To identify the warnings and errors specific to code generation from your MATLAB code, add the `%#codegen` directive to your MATLAB file. For more information, see Code Analyzer preferences.

Note The Code Analyzer does not detect all code generation issues. After eliminating the errors or warnings that the Code Analyzer detects, compile your code with GPU Coder to determine if the code has other compliance issues.

The code generation readiness tool screens the MATLAB code for features and functions that are not supported for code generation. This tool provides a report that lists issues and recommendations for

making the MATLAB code suitable for code generation. You can access the code generation readiness tool in these ways:

- In the current folder browser — right-click the MATLAB file that contains the entry-point function.
- At the command line — by using the `coder.screener` function with the `-gpu` flag.
- In the GPU Coder app — after specifying the entry-point files, the app runs the Code Analyzer and the code generation readiness tool.

Check for Issues at Code Generation Time

You can use GPU Coder to check for issues at code generation time. When GPU Coder detects errors or warnings, it generates an error report that describes the issues and provides links to the problematic MATLAB code. For more information, see “Code Generation Reports”.

Make the MATLAB Code Suitable for Code Generation

To begin the process of making your MATLAB code suitable for code generation, use the file `mandelbrot_count.m`.

1. Set your MATLAB current folder to the work folder that contains your files for this tutorial.
2. In the MATLAB Editor, open `mandelbrot_count.m`. The Code Analyzer message indicator at the top right corner of the MATLAB Editor is green. The analyzer did not detect errors, warnings, or opportunities for improvement in the code.
3. After the function declaration, add the `%#codegen` directive to turn on the error checking that is specific to code generation.

   ```
   function count = mandelbrot_count(maxIterations,xGrid,yGrid) %#codegen
   ```

   The Code Analyzer message indicator remains green, indicating that it has not detected any code generation issues.
4. To map the `mandelbrot_count` function to a CUDA kernel, modify the original MATLAB code by placing the `coder.gpu.kernelfun` pragma in the body of the function.

   ```
   function count = mandelbrot_count(maxIterations,xGrid,yGrid) %#codegen
   % 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);
   ```

   If you use the `coder.gpu.kernelfun` pragma, GPU Coder attempts to map the computations in the function `mandelbrot_count` to the GPU.
5. Save the file. You are now ready to compile your code by using the GPU Coder app.

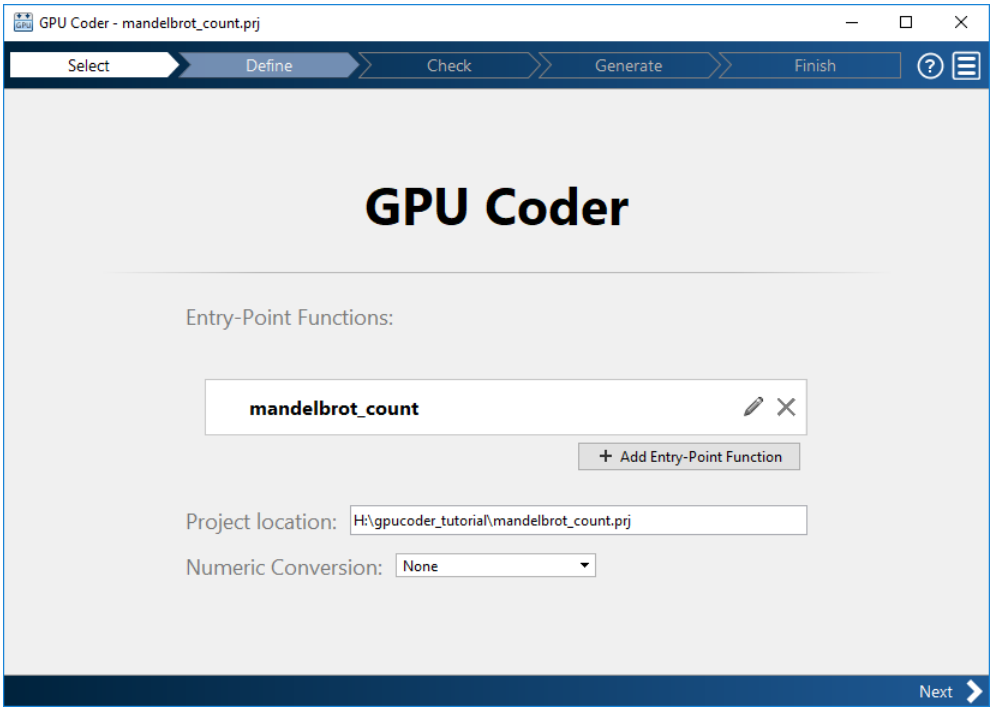
Generate Code by Using the GPU Coder App

Open the GPU Coder App

On the MATLAB toolstrip **Apps** tab, under **Code Generation**, click the GPU Coder app icon. You can also open the app by typing `gpucoder` in the MATLAB Command Window. The app opens the **Select** source files page.

Select Source Files

1 On the **Select** source files page, enter or select the name of the primary function, `mandelbrot_count`. The primary function is also known as the top-level or entry-point function. The app creates a project with the default name `mandelbrot_count.prj` in the current folder.



2 Click **Next** and go to the **Define Input Types** step. The app analyzes the function for coding issues and code generation readiness. If the app identifies issues, it opens the **Review Code Generation Readiness** page where you can review and fix issues. In this example, because the app does not detect issues, it opens the **Define Input Types** page.

Define Input Types

The code generator must determine the data types of all the variables in the MATLAB files at compile time. Therefore, you must specify the data types of all the input variables. You can specify the input data types in one of these two ways:

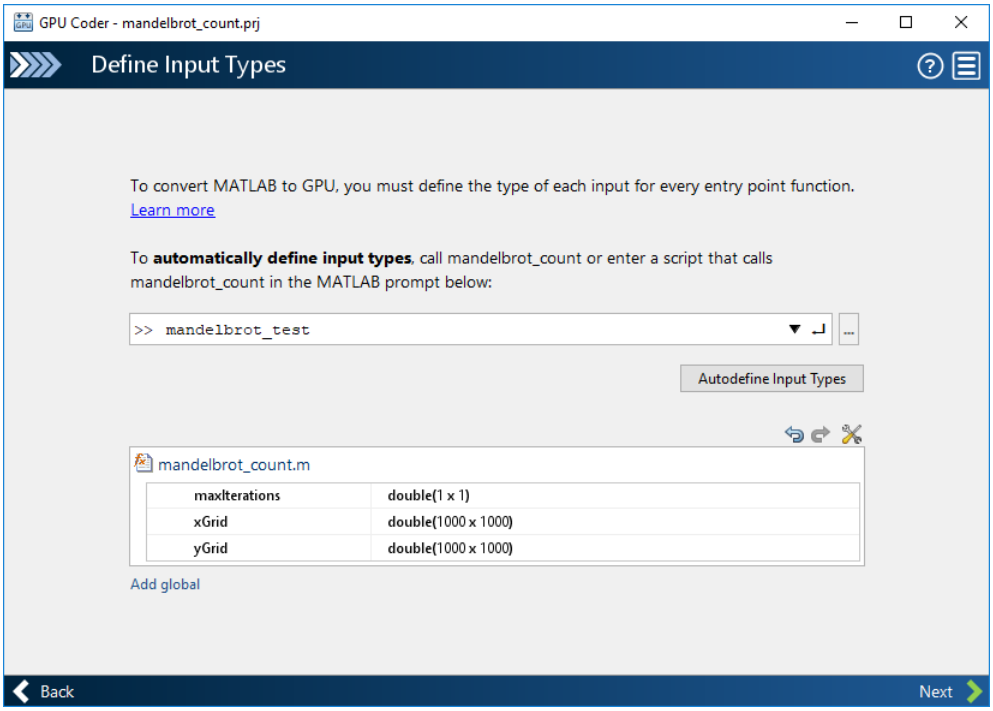
- Provide a test file that calls the project entry-point functions. The GPU Coder app can infer the input argument types by running the test file.
- Enter the input types directly.

For more information about input specifications, see “Input Specification”.

In this example, to define the properties of the inputs `maxIterations`, `xGrid`, and `yGrid`, specify the test file `mandelbrot_test.m`:

1. Enter or select the test file `mandelbrot_test.m`.
2. Click **Autodefine Input Types**.

   The test file `mandelbrot_test.m` calls the entry-point function, `mandelbrot_count.m` with the expected input types. The app infers that the input `maxIterations` is `double(1x1)` and the inputs `xGrid` and `yGrid` are `double(1000x1000)`.



3. Click **Next** go to the **Check for Run-Time Issues** step.

Check for Run-Time Issues

The **Check for Run-Time Issues** step generates a MEX file from your entry-point functions, runs the MEX function, and reports issues. This step is optional. However, it is a best practice to perform this step. Using this step, you can detect and fix defects that are harder to diagnose in the generated GPU code.

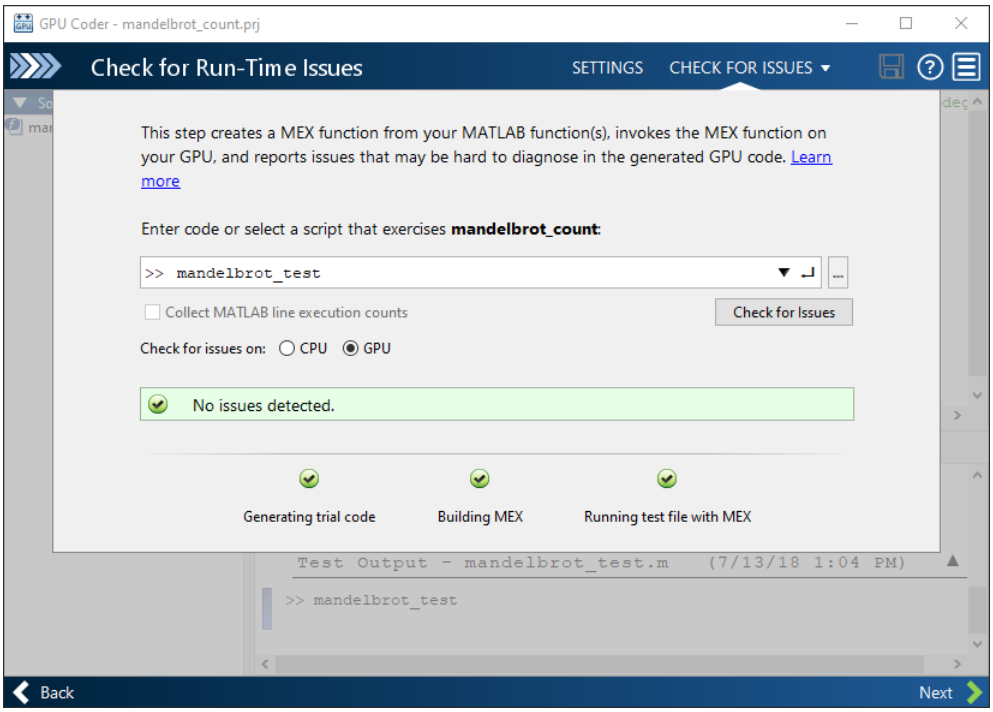
GPU Coder provides the option to perform GPU-specific checks at this point. When you select this option, GPU Coder generates CUDA code and a MEX file from your entry-point functions, runs the MEX function, and reports issues. Some of the GPU-specific run-time checks include:

- Checks for register spills.
- Stack size conformance checks.

Note There may be certain MATLAB constructs in your code that cause the **Check for Run-Time Issues** to fail CPU-specific checks but pass the GPU-specific checks.

1. To open the **Check for Run-Time Issues** dialog box, click the **Check for Issues** arrow.
2. In the **Check for Run-Time Issues** dialog box, specify a test file or enter code that calls the entry-point function with example inputs. For this example, use the test file `mandelbrot_test.m` that you used to define the input types.
3. To enable GPU-specific checks, select the **GPU** option button. Click **Check for Issues**.

   The app generates a MEX function. It runs the test script `mandelbrot_test` replacing calls to `mandelbrot_count` with calls to the generated MEX. If the app detects issues during the MEX function generation or execution, it provides warning and error messages. You can click these messages to navigate to the problematic code and fix the issue. In this example, the app does not detect issues. The MEX function has the same functionality as the original `mandelbrot_count` function.

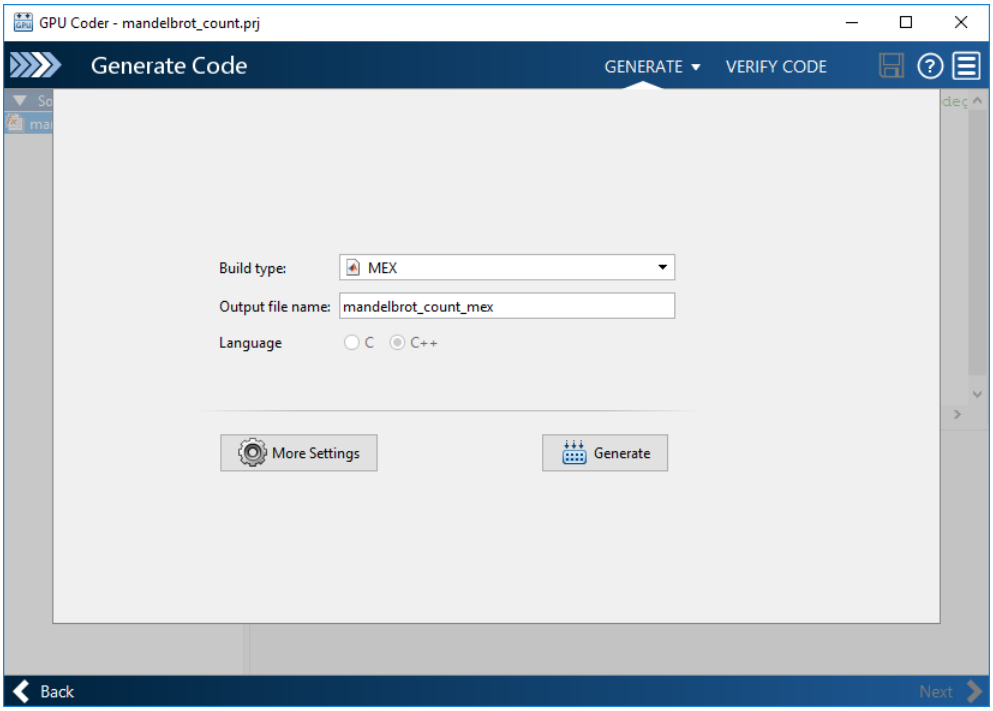


Note There may be certain MATLAB constructs in your code that cause the **Check for Run-Time Issues** to fail CPU-specific checks but pass the GPU-specific checks.

4. Click **Next** go to the **Generate Code** step.

Generate CUDA Code

1. To open the **Generate** dialog box, click the **Generate** arrow.

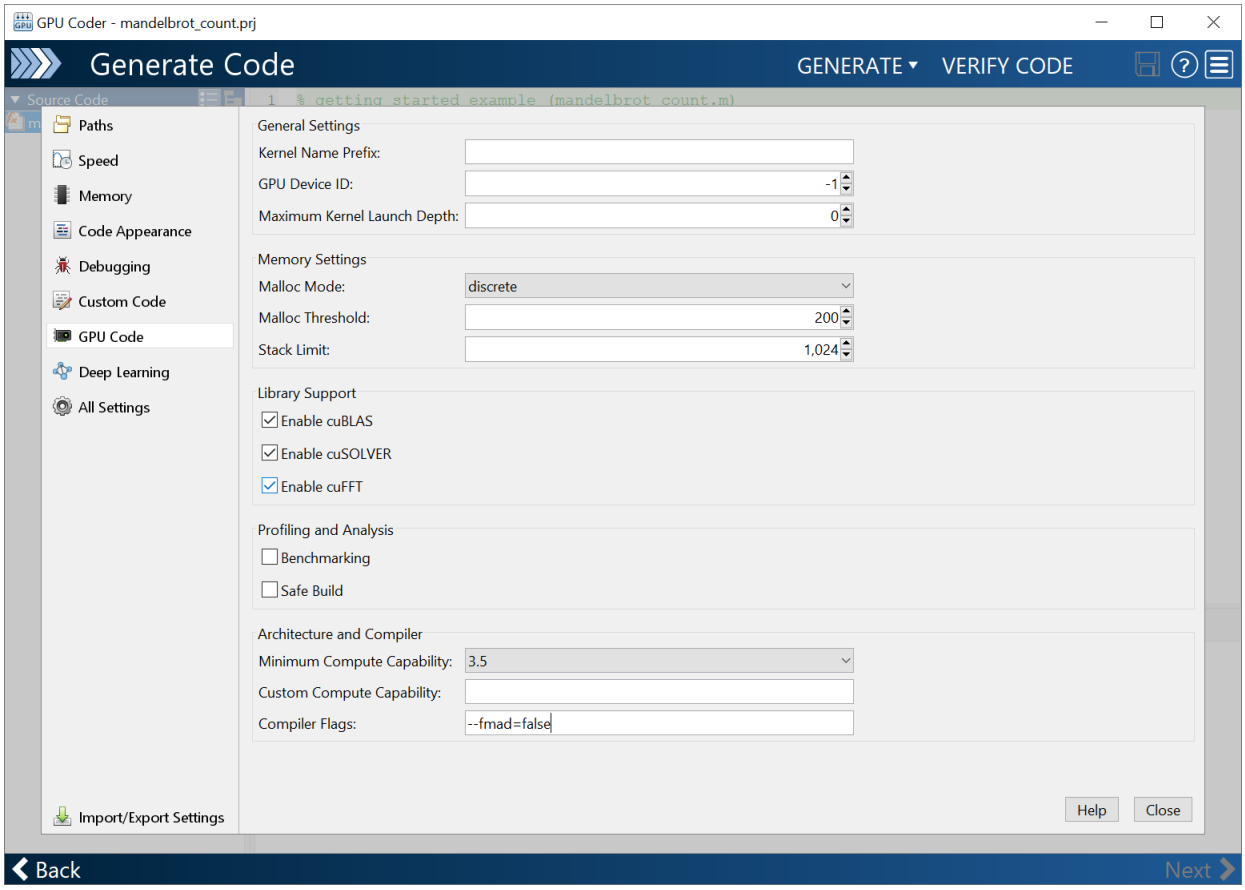


2 In the **Generate** dialog box, you can select the type of build that you want GPU Coder to perform. The available options are listed in this table.

Build Type	Description
`Source code`	CUDA source code to integrate with an external project.
`MEX`	Compiled code to run inside MATLAB.
`Static Library`	Binary library for static linking with an external project.
`Dynamic Library`	Binary library for dynamic linking with an external project.
`Executable`	Standalone program (requires a custom CUDA main file).

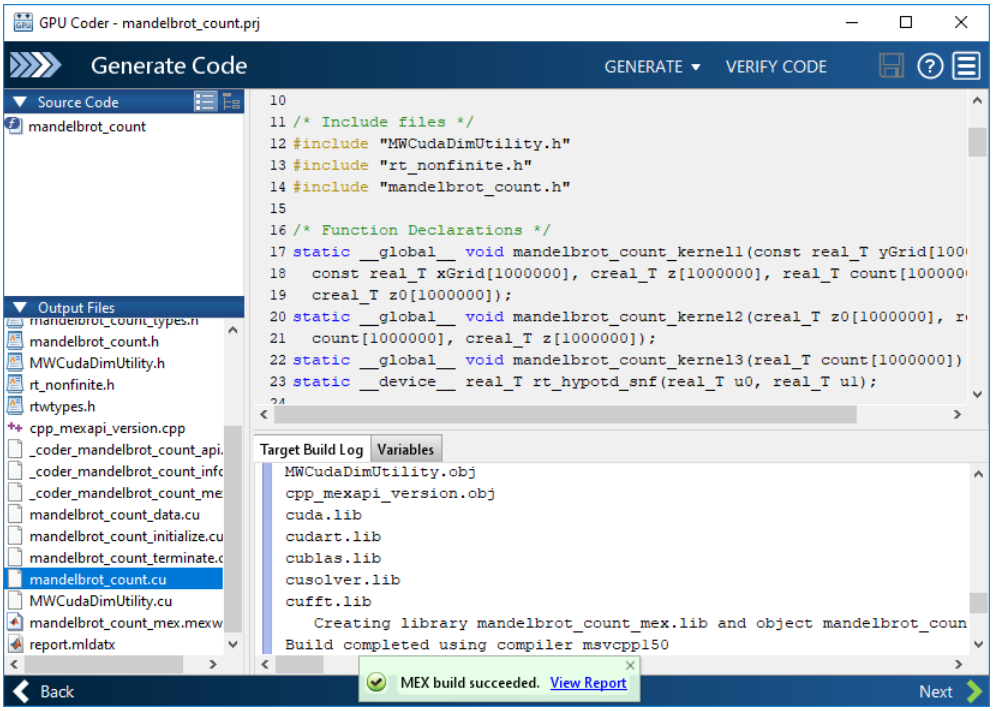
For this tutorial, set **Build type** to `MEX(.mex)`. By generating a MEX output, you can check the correctness of the generated CUDA code from within MATLAB. The MEX build type does not require additional settings like **Toolchain** and **Hardware Board**. It also does not provide the option to generate only the source code. GPU Coder can automatically select an available CUDA toolchain as long as the “Environment Variables” on page 1-7 are set properly.

To view advanced options, select **More Settings - > GPU Code**. To the **Compiler Flags** option, add `--fmad=false`. This 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 between the CPU and the GPU. For more information, see “Numerical Differences Between CPU and GPU” on page 3-4.

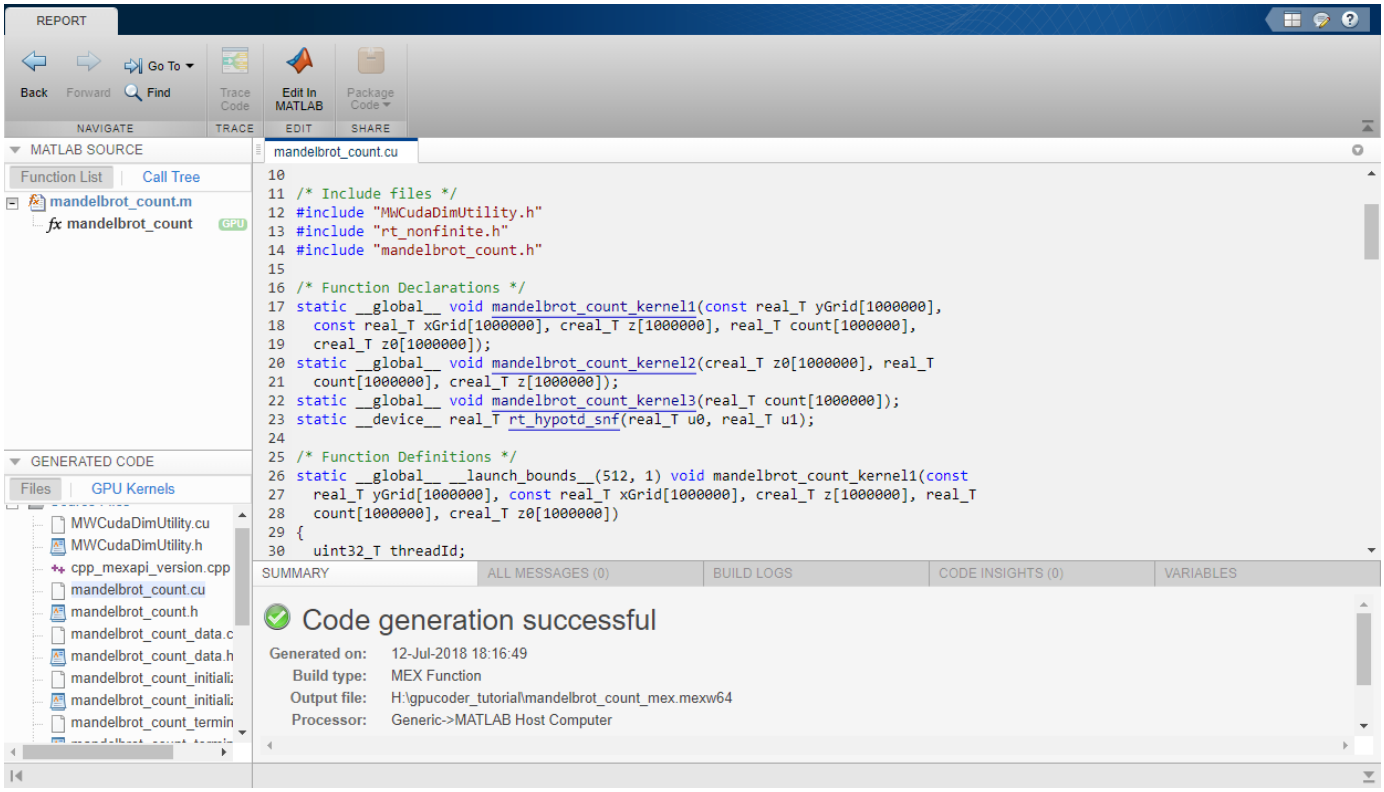


3 Click **Generate**.

GPU Coder generates the MEX executable `mandelbrot_count_mex` in your working folder. The `<pwd>\codegen\mex\mandelbrot_count` folder contains all other the generated files including the CUDA source (*.cu) and header files. The GPU Coder app indicates that the code generation succeeded. It displays the source MATLAB files and generated output files on the left side of the page. On the **Variables** tab, it displays information about the MATLAB source variables. On the **Target Build Log** tab, it displays the build log, including compiler warnings and errors. By default, in the code window, the app displays the CUDA source file `mandelbrot_count.cu`. To view a different file, in the **Source Code** or **Output Files** pane, click the file name.



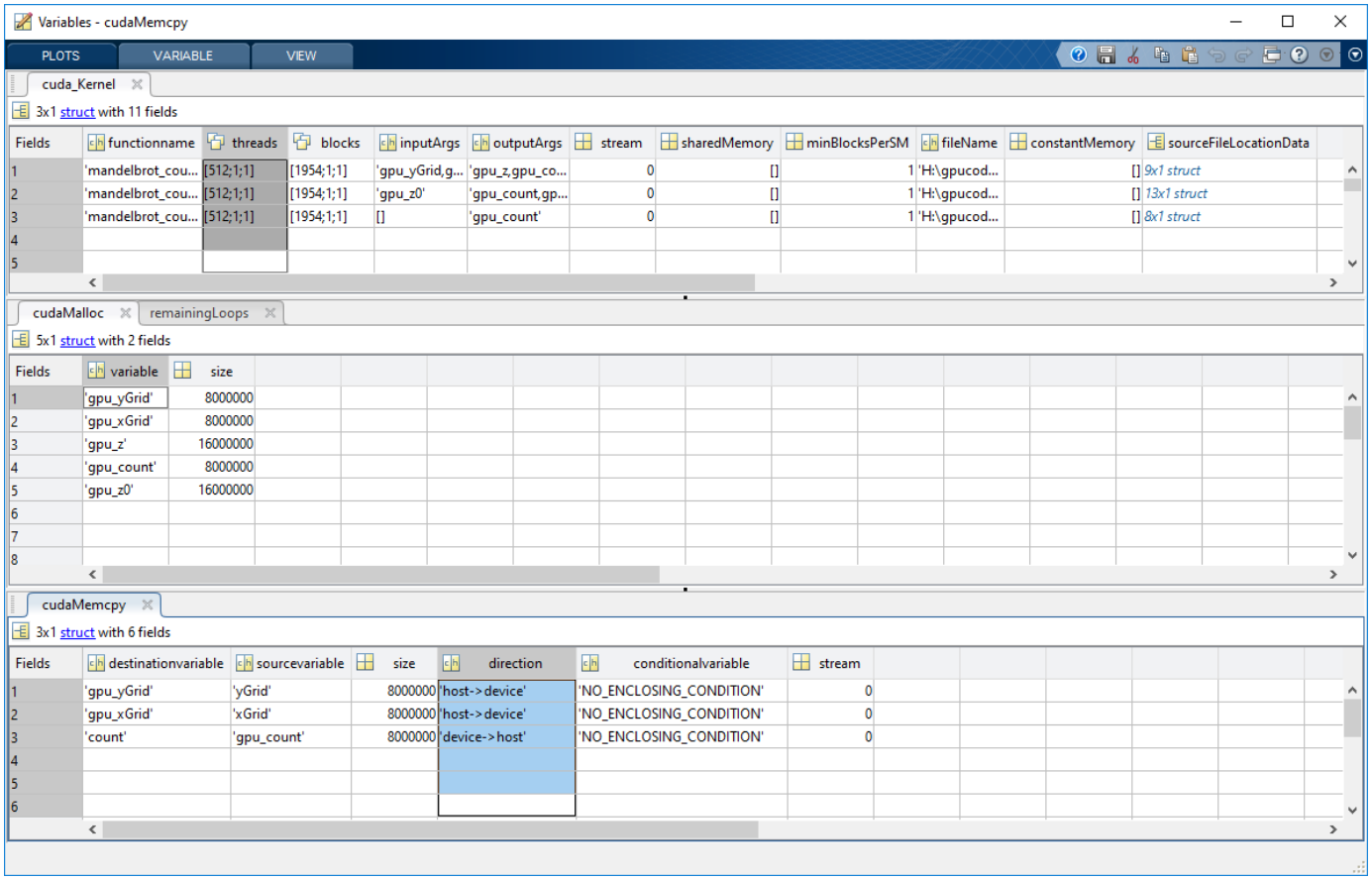
4 To view the code generation report, click **View Report**. The report provides links to your MATLAB code and the generated CUDA (*.cu) files. It also provides compile-time information for the variables and expressions in your MATLAB code. This information helps you to find sources of error and warnings. It also helps you to debug code generation issues in your code. For more information, see “Code Generation Reports”.



The **GPU Kernels** section on the **Generated Code** tab provides a list of kernels created during GPU code generation. The items in this list link to the relevant source code. For example, when you click **mandelbrot_count_kernel1**, the code section for this kernel is shown in the code browser window.

After you review the report, you can close the **Code Generation Report** window. To view the report later, open `report.mldatx` in `<pwd>\codegen\mex\mandelbrot_cout\html` folder.

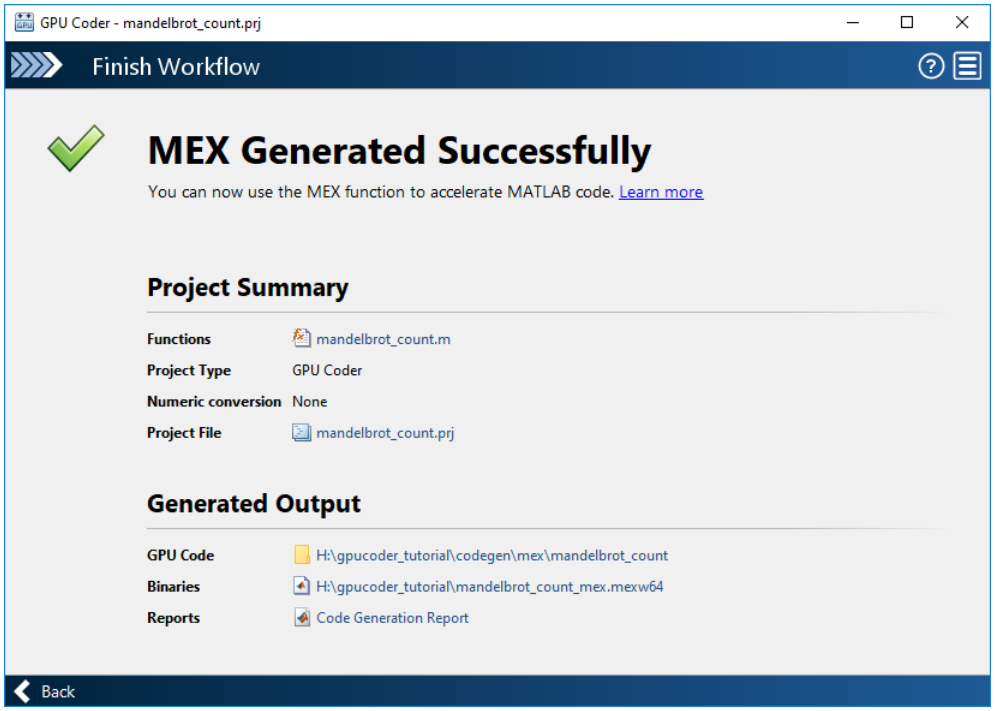
5 The `<pwd>\codegen\mex\mandelbrot_count` contains the `gpu_codegen_info.mat` MAT-file that contains the statistics for the generated GPU code. This MAT-file contains the `cuda_Kernel` variable that has information about the thread and block sizes, shared and constant memory usage, and input and output arguments of each kernel. The `cudaMalloc` and `cudaMemcpy` variables contain information about the size of all the GPU variables and the number of `memcpy` calls between the host and the device.



6 In the GPU Coder app, click **Next** to open the **Finish Workflow** page.

Review the Finish Workflow Page

The **Finish Workflow** page indicates that the code generation succeeded. It provides a project summary and links to the MATLAB source files, the code generation report, and the generated output binaries. You can save the configuration parameters of the current GPU Coder project as a MATLAB script. See “Convert MATLAB Coder Project to MATLAB Script”.



Verify Correctness of the Generated Code

To verify the correctness of the generated MEX file, see “Verify Correctness of the Generated Code” on page 3-2.

See Also

Apps
GPU Coder

Functions
`codegen` | `coder.gpuConfig` | `coder.gpu.kernelfun` | `gpucoder`

Objects
`coder.gpuConfig` | `coder.CodeConfig` | `coder.EmbeddedCodeConfig`

More About

- “GPU Programming Paradigm” on page 4-2
- “Installing Prerequisite Products” on page 1-3
- “Setting Up the Prerequisite Products” on page 1-7
- “GPU Code Generation Workflow” on page 4-3
- “The GPU Environment Check and Setup App” on page 1-12

Code Generation Using the Command Line Interface

The easiest way to create CUDA kernels is to place the `coder.gpu.kernelfun` pragma into your primary MATLAB function. The primary function is also known as the top-level or entry-point function. When the GPU Coder encounters `kernelfun` pragma, it attempts to parallelize all the computation within this function and then maps it to the GPU.

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.
- Define function input properties.
- Check for code generation readiness and run-time issues.
- Specify code generation properties.
- Generate CUDA code by using the `codegen` command.

Tutorial Prerequisites

This tutorial requires the following products:

- MATLAB
- MATLAB Coder
- GPU Coder
- C compiler
- NVIDIA GPU enabled for CUDA
- CUDA Toolkit and driver
- Environment variables for the compilers and libraries. For more information, see “Environment Variables” on page 1-7

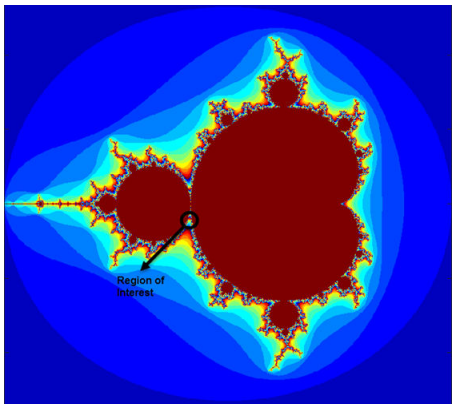
Example: The Mandelbrot Set

Description

The Mandelbrot set is the region in the complex plane consisting of the values z_0 for which the trajectories defined by this equation remain bounded at $k \to \infty$.

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

The overall geometry of the Mandelbrot set is shown in the figure. This view does not have the resolution to show the richly detailed structure of the fringe just outside the boundary of the set. At increasing magnifications, the Mandelbrot set exhibits an elaborate boundary that reveals progressively finer recursive detail.



Algorithm

For this tutorial, pick a set of limits that specify a highly zoomed part of the Mandelbrot set in the valley between the main cardioid and the *p*/*q* bulb to its left. A 1000-by-1000 grid of real parts (*x*) and imaginary parts (*y*) is created between these two limits. The Mandelbrot algorithm is then iterated at each grid location. An iteration number of 500 renders the image in full resolution.

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

This tutorial uses an implementation of the Mandelbrot set by using standard MATLAB commands running on the CPU. This calculation is vectorized such that every location is updated simultaneously.

Tutorial Files

Create a MATLAB script called `mandelbrot_count.m` with the following lines of code. This code is a baseline vectorized MATLAB implementation of the Mandelbrot set. Later in this tutorial, you modify this file to make it suitable for code generation.

```
function count = mandelbrot_count(maxIterations, xGrid, yGrid)
% 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 a MATLAB script called `mandelbrot_test.m` with the following lines of code. The script generates 1000-by-1000 grid of real parts (*x*) and imaginary parts (*y*) between the limits specified by `xlim` and `ylim`. It also calls the `mandelbrot_count` 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 );

%% Mandelbrot computation in MATLAB
count = mandelbrot_count(maxIterations, xGrid, yGrid);

% Show
figure(1)
imagesc( x, y, count );
colormap([jet();flipud( jet() );0 0 0]);
axis off
title('Mandelbrot set with MATLAB');
```

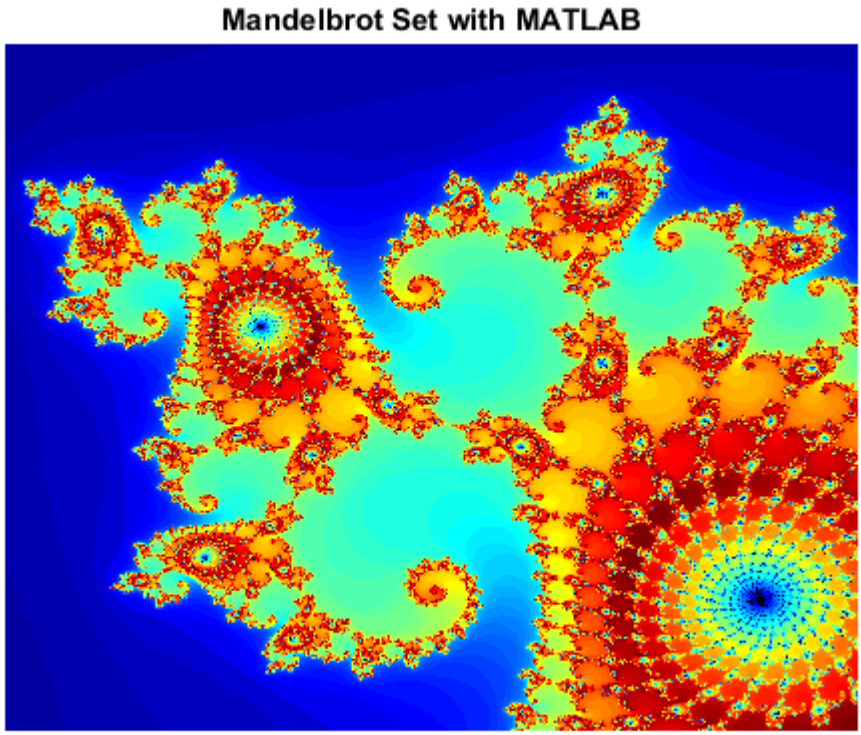
Run the Original MATLAB Code

Run the Mandelbrot Example

Before making the MATLAB version of the Mandelbrot set algorithm suitable for code generation, you can test the functionality of the original code.

1. Change the current working folder of MATLAB to the location that contains the two files you created in the previous step. GPU Coder places generated code in this folder, change your current working folder if you do not have full access to this folder.
2. Open the `mandelbrot_test` script in the MATLAB Editor.
3. Run the test script by clicking the run button  or by entering `mandelbrot_test` in the MATLAB Command Window.

   The test script runs and shows the geometry of the Mandelbrot within the boundary set by the variables `xlim` and `ylim`.



Make the MATLAB Code Suitable for Code Generation

To begin the process of making your MATLAB code suitable for code generation, use the file `mandelbrot_count.m`.

1. Set your MATLAB current folder to the work folder that contains your files for this tutorial.
2. In the MATLAB Editor, open `mandelbrot_count.m`. The file opens in the MATLAB Editor. The Code Analyzer message indicator in the top right corner of the MATLAB Editor is green. The analyzer did not detect errors, warnings, or opportunities for improvement in the code.
3. Turn on MATLAB for code generation error checking. After the function declaration, add the `%#codegen` directive.

   ```
   function count = mandelbrot_count(maxIterations, xGrid, yGrid) %#codegen
   ```

   The Code Analyzer message indicator remains green, indicating that it has not detected code generation issues.
4. To map the `mandelbrot_count` function to a CUDA kernel, modify the original MATLAB code by placing the `coder.gpu.kernelfun` pragma outside the `for`-loop body.

   ```
   function count = mandelbrot_count(maxIterations, xGrid, yGrid) %#codegen
   % mandelbrot computation

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

   % Add Kernelfun pragma to trigger kernel creation
   coder.gpu.kernelfun;
   ```

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

When using the `coder.gpu.kernelfun` pragma, GPU Coder attempts to map the computations in the function `mandelbrot_count` to the GPU.

5 Save the file. You are now ready to compile your code by using the command-line interface.

Code Generation from the Command Line

You can use the `codegen` command to translate MATLAB functions to a CUDA compatible static or dynamic library, executable, or MEX function, instead of using the GPU Coder app.

Define Input Types

At compile time, GPU Coder must know the data types of all the inputs to the entry-point function. Therefore, if your entry-point function has inputs, you must specify its data type at the time that you compile the file with the `codegen` function.

You can generate inputs and then use the `-args` option in the `codegen` function to let GPU Coder determine the class, size, and complexity of the input parameters. To generate inputs for `mandelbrot_count` function, use these commands:

```
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 );
```

Alternatively, you can specify the size, type and complexity of the inputs to the entry-point functions without generating input data by using the `coder.typeof` function.

```
ARGS = cell(1,1);
ARGS{1} = cell(3,1);
ARGS{1}{1} = coder.typeof(0);
ARGS{1}{2} = coder.typeof(0,[1000 1000]);
ARGS{1}{3} = coder.typeof(0,[1000 1000]);
```

Build Configuration

To configure build settings such as output file name, location, type, you have to create coder configuration objects. To create the objects, use the `coder.gpuConfig` function. For example, to create a `coder.MexCodeConfig` code generation object for use with `codegen` when generating a MEX function, use:

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

Other available options are:

- `cfg = coder.gpuConfig('lib');`, to create a code generation configuration object for use with `codegen` when generating a CUDA static library.

- `cfg = coder.gpuConfig('dll');`, to create a code generation configuration object for use with `codegen` when generating a CUDA dynamic library.
- `cfg = coder.gpuConfig('exe');`, to create a code generation configuration object for use with `codegen` when generating a CUDA executable.

For more information, see `coder.gpuConfig`.

Each configuration object comes with a set of parameters, initialized to default values. You can use dot notation to modify the value of one configuration object parameter at a time. Use this syntax:

```
configuration_object.property = value
```

You can enable the same settings as in the “Code Generation by Using the GPU Coder App” on page 2-2 by using the following command-line equivalents:

```
cfg = coder.gpuConfig('mex');
cfg.GpuConfig.CompilerFlags = '--fmad=false';
cfg.GenerateReport = true;
```

The `cfg` configuration object has configuration parameters that are common to MATLAB Coder and GPU Coder and parameters that are GPU Coder-specific. You can see all the GPU-specific properties available in the `cfg` configuration object by typing `cfg.GpuConfig` in the MATLAB Command Window.

```
>> cfg.GpuConfig

ans =

  config with properties:

                   Enabled: 1
                MallocMode: 'discrete'
          KernelNamePrefix: ''
              EnableCUBLAS: 1
            EnableCUSOLVER: 1
               EnableCUFFT: 1
              Benchmarking: 0
                 SafeBuild: 0
         ComputeCapability: '3.5'
    CustomComputeCapability: ''
             CompilerFlags: ''
        StackLimitPerThread: 1024
            MallocThreshold: 200
          SelectCudaDevice: -1
```

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” on page 3-4.

For more information on configuration parameters that are common to MATLAB Coder and GPU Coder, see `coder.CodeConfig` class.

Build Script

You can create a build script `mandelbrot_codegen.m` that automates the series of commands mentioned previously.

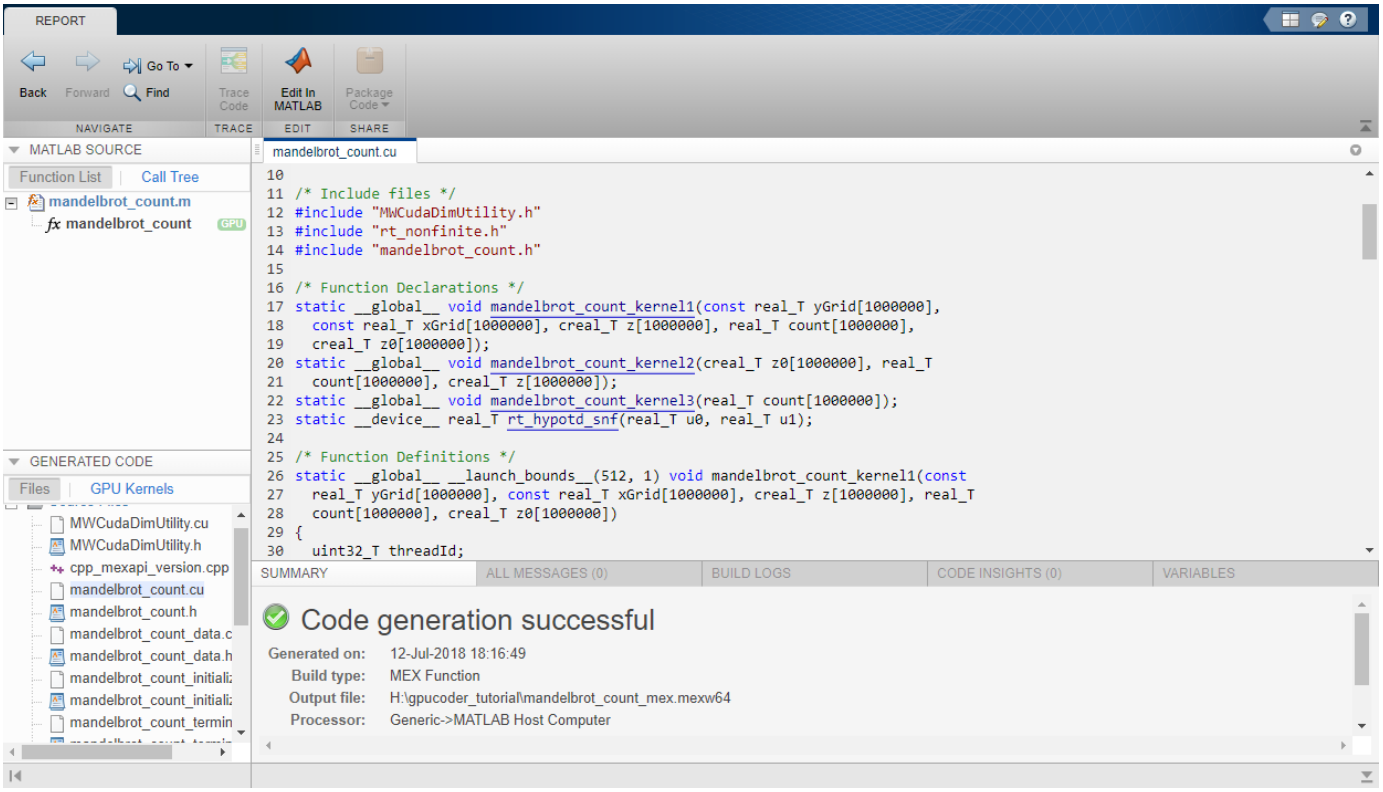
```
% GPU code generation for getting started example (mandelbrot_count.m)
%% Create configuration object of class 'coder.MexCodeConfig'.
cfg = coder.gpuConfig('mex');
cfg.GenerateReport = true;
cfg.GpuConfig.CompilerFlags = '--fmad=false';

%% Define argument types for entry-point 'mandelbrot_count'.
ARGS = cell(1,1);
ARGS{1} = cell(3,1);
ARGS{1}{1} = coder.typeof(0);
ARGS{1}{2} = coder.typeof(0,[1000 1000]);
ARGS{1}{3} = coder.typeof(0,[1000 1000]);

%% Invoke GPU Coder.
codegen -config cfg mandelbrot_count -args ARGS{1}
```

The `codegen` command opens the file `mandelbrot_count.m` and translates the MATLAB code into CUDA code.

- The `-report` option instructs `codegen` to generate a code generation report that you can use to debug your MATLAB code.
- The `-args` option instructs `codegen` to compile the file `mandelbrot_count.m` by using the class, size, and complexity of the input parameters *maxIterations*, *xGrid*, and *yGrid*.
- The `-config` option instructs `codegen` to use the specified configuration object for code generation.

When code generation is successful, you can view the resulting code generation report by clicking **View Report** in the MATLAB Command Window.

```
>> mandelbrot_codegen
Code generation successful: View report
```



Verify Correctness of the Generated Code

To verify correctness of the generated MEX file, see “Verify Correctness of the Generated Code” on page 3-2.

See Also

Apps
GPU Coder

Functions
`codegen` | `coder.gpuConfig` | `coder.gpu.kernelfun` | `gpucoder`

Objects
`coder.gpuConfig` | `coder.CodeConfig` | `coder.EmbeddedCodeConfig`

More About

- “GPU Programming Paradigm” on page 4-2
- “Installing Prerequisite Products” on page 1-3
- “Setting Up the Prerequisite Products” on page 1-7
- “GPU Code Generation Workflow” on page 4-3
- “The GPU Environment Check and Setup App” on page 1-12

GPU Code Generation: The Mandelbrot Set

This example shows how to generate CUDA® code from a simple MATLAB® function by using GPU Coder™. A Mandelbrot set implementation by using standard MATLAB commands acts as the entry-point function. This example uses the `codegen` command to generate a MEX function that runs on the GPU. You can run the MEX function to check for run-time errors.

Third-Party Prerequisites

Required

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

- CUDA enabled NVIDIA® GPU and compatible driver.

Optional

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

- NVIDIA toolkit.
- Environment variables for the compilers and libraries. For more information, see “Third-Party Hardware” on page 1-3 and “Setting Up the Prerequisite Products” on page 1-7.

Verify GPU Environment

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

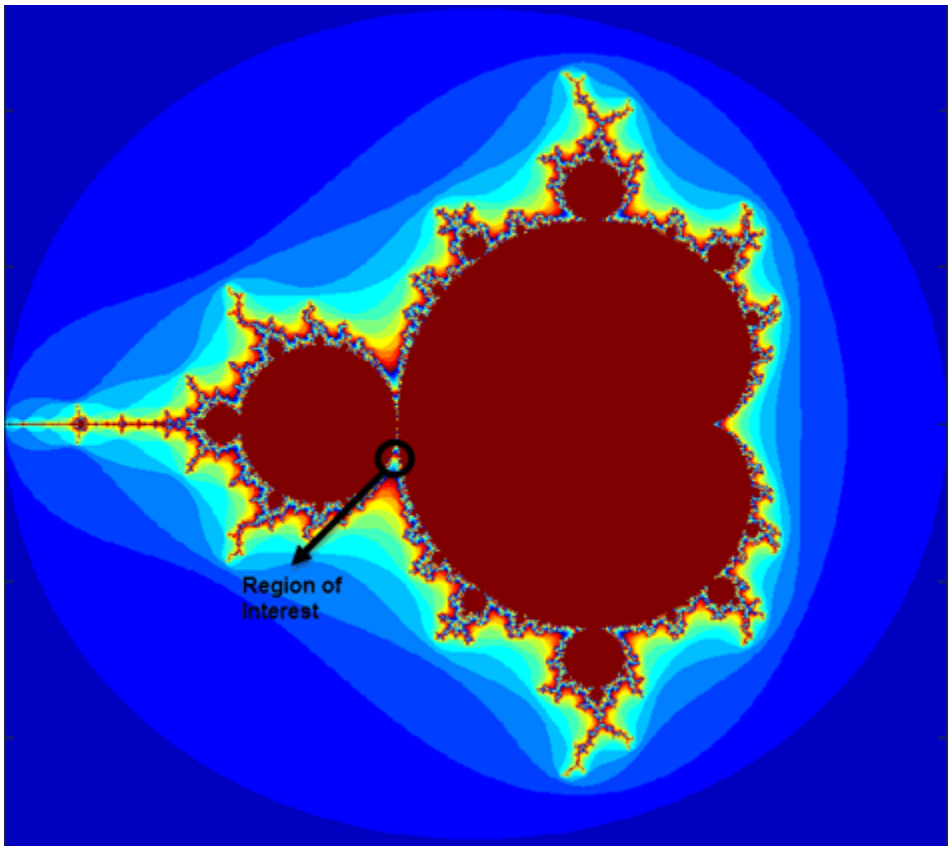
```
envCfg = coder.gpuEnvConfig('host');
envCfg.BasicCodegen = 1;
envCfg.Quiet = 1;
coder.checkGpuInstall(envCfg);
```

Mandelbrot Set

The Mandelbrot set is the region in the complex plane consisting of the values z_0 for which the trajectories defined by

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

remain bounded at $k \to \infty$. The overall geometry of the Mandelbrot set is shown in the figure. This view does not have the resolution to show the richly detailed structure of the fringe just outside the boundary of the set.



Define Input Regions

Pick a set of limits that specify a highly zoomed part of the Mandelbrot set in the valley between the main cardioid and the p/q bulb to its left. A `1000x1000` grid of $Re\{x\}$ and $Im\{y\}$ is created between these two limits. The Mandelbrot algorithm is then iterated at each grid location. An iteration number of 500 is enough to render the image in full resolution.

```
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 );
```

The Mandelbrot Entry-Point Function

The `mandelbrot_count.m` entry-point function contains a vectorized implementation of the Mandelbrot set based on the code provided in the e-book *Experiments with MATLAB* by Cleve Moler. The %#codegen directive turns on MATLAB for code generation error checking. When GPU Coder encounters the `coder.gpu.kernelfun` pragma, it attempts to parallelize all the computation within this function, and then maps it to the GPU.

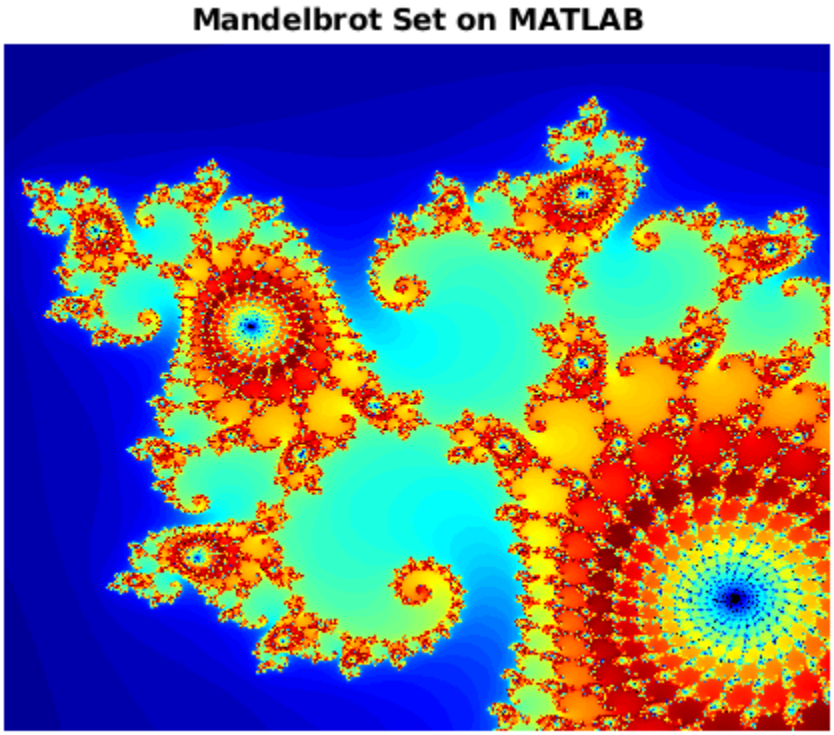
```
type mandelbrot_count
```

```
function count = mandelbrot_count(maxIterations, xGrid, yGrid) %#codegen

% Copyright 2016-2019 The MathWorks, Inc.

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

% Map computation to GPU.
coder.gpu.kernelfun;

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

Test the Functionality of `mandelbrot_count`

Run the `mandelbrot_count` function with the xGrid, yGrid values that were previously generated, and then plot the results.

```
count = mandelbrot_count(maxIterations, xGrid, yGrid);

figure(2), imagesc( x, y, count );
colormap( [jet();flipud( jet() );0 0 0] );
title('Mandelbrot Set on MATLAB');
axis off
```



Generate CUDA MEX for the Function

To generate CUDA MEX for the `mandelbrot_count` function, create a GPU code configuration object and run the `codegen` command. Because of architectural differences between the CPU and GPU, numeric verification does not always match. This scenario is true when using the single data type in your MATLAB code and performing accumulation operations on these single data type values. Like this Mandelbrot example even the double data types cause numeric errors. One reason for this mismatch is that the GPU floating-point units use fused Floating-point Multiply-Add (FMAD) instructions and the CPU does not use these instructions. The `fmad=false` option that is passed to the `nvcc` compiler turns off this FMAD optimization.

```
cfg = coder.gpuConfig('mex');
cfg.GpuConfig.CompilerFlags = '--fmad=false';
codegen -config cfg -args {maxIterations,xGrid,yGrid} mandelbrot_count
```

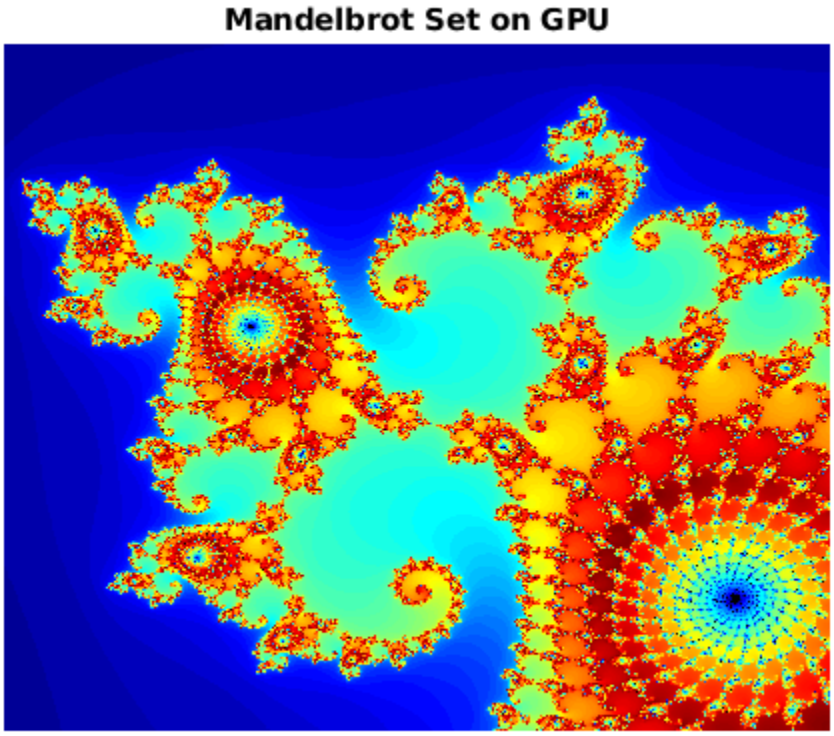
```
Code generation successful: To view the report, open('codegen/mex/mandelbrot_count/html/report.m
```

Run the MEX Function

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

```
countGPU = mandelbrot_count_mex(maxIterations, xGrid, yGrid);

figure(2), imagesc( x, y, countGPU );
colormap( [jet();flipud( jet() );0 0 0] );
```

```
title('Mandelbrot Set on GPU');
axis off
```



See Also

Apps
GPU Coder

Functions
`codegen` | `coder.gpuConfig` | `coder.gpu.kernelfun` | `gpucoder`

Objects
`coder.gpuConfig` | `coder.CodeConfig` | `coder.EmbeddedCodeConfig`

More About

- “GPU Programming Paradigm” on page 4-2
- “Installing Prerequisite Products” on page 1-3
- “Setting Up the Prerequisite Products” on page 1-7
- “GPU Code Generation Workflow” on page 4-3
- “The GPU Environment Check and Setup App” on page 1-12

Debug CUDA MEX Functions

You can debug your generated CUDA MEX function using MATLAB or a CUDA debugger. To debug your CUDA MEX functions in MATLAB, use the `disp` function to inspect the contents of your MEX function variables. You cannot use `save` to debug MEX function variables because code generation does not support it. Code generation does not support declaration of `save` as extrinsic. You can also use the `fprintf` function to inspect the contents of your MEX function variables.

Debug CUDA MEX Functions by Using a Debugger

This example shows how to debug CUDA MEX functions using a debugger.

1. Consider an entry-point function `foo` that squares each element of a matrix `x` and scales the result by a factor of `1/(i+j)`, where `i,j` are the row and column indexes.

   ```
   function [y] = foo(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
   ```

2. To build a CUDA MEX function with debugging symbols included, set the MEX configuration object property `EnableDebugging` to `1`.

   ```
   cfg = coder.gpuConfig('mex');
   cfg.EnableDebugging = 1;
   input = rand(32);

   codegen -config cfg -args {input} foo
   ```

   Alternatively, you can debug your MEX function by executing this command:

   ```
   codegen -g -args {input} foo
   ```

You can debug the generated CUDA MEX (`foo_mex`) by using the Visual Studio CUDA debugger on Windows or the CUDA GNU® debugger `cuda-gdb` on Linux systems.

Debug on Microsoft Windows Platforms

This example shows the general steps to debug `foo_mex` by using the NVIDIA Nsight Visual Studio Edition CUDADebugger. For specific information about using Nsight VSE, refer to NVIDIA documentation.

1. After generating the CUDA MEX function, start Visual Studio. Do not exit your MATLAB session.
2. Attach the debugger to the running MATLAB process by selecting **Debug > Attach to Process** or press `Ctrl + Alt + p` in Visual Studio. For more information, refer to your Visual Studio documentation.
3. Set breakpoints in code. Select **Debug > New Breakpoint** in Visual Studio. For more information, refer to your Visual Studio documentation.

4 Open MATLAB and type:

```
out = foo_mex(inputs);
```

`foo.cu` is opened in the Visual Studio CUDA debugger at the first breakpoint.

5 If you select **Debug** > **Continue**, code execution completes and the results can be verified in MATLAB.

Debug on Linux Platforms

The CUDA GNU Debugger `cuda-gdb`, available for Linux systems as part of the CUDA Toolkit installation, provides complete source code debugging, including the ability to set breakpoints, examine variables, and step through the source code line-by-line.

In this procedure, the MATLAB command prompt `>>` is shown in front of MATLAB commands, and `linux>` represents a Linux prompt; your system might show a different prompt. The debugger prompt is `<cuda-gdb>`.

1 To debug with `cuda-gdb`, at the Linux prompt, start the `cuda-gdb` debugger using the `matlab` function `-D` option.

```
linux> matlab -Dcuda-gdb
```

2 Tell `cuda-gdb` to stop for debugging.

```
<cuda-gdb> handle SIGSEGV SIGBUS nostop noprint
<cuda-gdb> handle SIGUSR1 stop print
```

3 Start MATLAB without the Java® Virtual Machine (JVM™) by using the `-nojvm` startup flag.

```
<cuda-gdb> run -nojvm
```

4 In MATLAB, enable debugging with the `dbmex` function and run your binary MEX file.

```
>> dbmex on
>> out = foo_mex(rand(32));
```

5 You are ready to start debugging.

It is often convenient to set a breakpoint at `mexFunction` so you stop at the beginning of the gateway routine.

```
<cuda-gdb> break mexFunction
<cuda-gdb> r
```

6 Once you hit one of your breakpoints, you can make full use of any commands the debugger provides to examine variables, display memory, or inspect registers.

To proceed from a breakpoint, type:

```
<cuda-gdb> continue
```

7 After stopping at the last breakpoint, type:

```
<cuda-gdb> continue
```

Code execution finishes and the results can be verified on MATLAB.

8 From the MATLAB prompt you can return control to the debugger by typing:

```
>> dbmex stop
```

Or, if you are finished running MATLAB, type:

```
>> quit
```

When you are finished with the debugger, type:

```
<cuda-gdb> quit
```

You return to the Linux prompt.

For more information on CUDA debugger, refer to NVIDIA documentation.

See Also

`dbmex` | `codegen` | `coder.gpuConfig` | `coder.gpu.kernelfun` | `gpucoder`

More About

- “GPU Programming Paradigm” on page 4-2
- “Installing Prerequisite Products” on page 1-3
- “Setting Up the Prerequisite Products” on page 1-7
- “GPU Code Generation Workflow” on page 4-3
- “Code Generation Using the Command Line Interface” on page 2-16

Verification

Verify Correctness of the Generated Code

After you generate code, GPU Coder provides you multiple options to inspect the source code and test the correctness of the generated code.

- The code generation report provides an interactive interface for inspecting the generated CUDA source files, generated data types, and other code insights. For more information, see “Code Generation Reports”.
- Verify generated MEX functions in the GPU Coder app.
- Verify generated MEX functions at the command line.
- With the MATLAB Coder Support Package for NVIDIA Jetson and NVIDIA DRIVE Platforms, you can use the processor-in-the-loop (PIL) execution to check the numerical behavior of the CUDA code that you generate from MATLAB functions. For more information, see “Processor-In-The-Loop Execution from Command Line” (MATLAB Coder Support Package for NVIDIA Jetson and NVIDIA DRIVE Platforms) and “Processor-In-The-Loop Execution with the GPU Coder App” (MATLAB Coder Support Package for NVIDIA Jetson and NVIDIA DRIVE Platforms).
- If you have Embedded Coder, you can verify the numerical behavior of the generated CUDA code by using software-in-the-loop (SIL) execution.

Verify MEX Functions in the GPU Coder App

In the GPU Coder app, after you generate a MEX function, you can verify that it has the same functionality as the original MATLAB entry-point function. Provide a test file that generates test vectors for the function under test and then calls the original MATLAB entry-point function. The test file can be a MATLAB function or script but must be in the same folder as the original entry-point function.

- On the **Generate Code** page of the GPU Coder app, click **Verify Code**.
- Type or select the test file. For example, `myfunction_test`.
- To run the test file without replacing calls to the original MATLAB function with calls to the MEX function, select **MATLAB code** for the **Run using** option. Click **Run Generated Code**.
- To run the test file, replacing calls to the original MATLAB function with calls to the MEX function, select **Generated code** for the **Run using** option. Click **Run Generated Code**.
- Compare the results of running the original MATLAB function with the results of running the generated CUDA MEX function.

Verify MEX Functions at the Command Line

You can verify the generated CUDA MEX file at the command line by using the `coder.runTest` function. The `coder.runTest` function runs the test file by replacing calls to the original MATLAB function with calls to the MEX function. For example, to test `myfunction` function with `myfunction_test` test file, enter the following code in the MATLAB Command Window.

```
coder.runTest('myfunction_test','myfunction')
```

Compare the results with the results of running the original MATLAB function. If errors occur during the run, call stack information is available for debugging. Alternatively, you can use `codegen` with the `-test` option.

```
codegen myfunction -test'myfunction_test'
```

The test file can be a MATLAB function, script, or class-based unit test.

Code Verification Through Software-In-The-Loop

GPU Coder supports software-in-the-loop (SIL) execution, which enables you to verify source code and compiled object code. During SIL execution through a MATLAB SIL interface, the software compiles and using the test vectors that you provide, runs library code on your development computer. You can reuse test vectors developed for your MATLAB functions to verify the numerical behavior of library code.

Note

- On a Microsoft Windows system, the Windows Firewall can potentially block a SIL execution. Change the Windows Firewall settings to allow access.
- When using SIL execution, make sure that the `Benchmarking` option in GPU Coder settings is `false`. Executing SIL with benchmarking results in compilation errors.

SIL Execution with the GPU Coder App

The software-in-the-loop (SIL) execution is supported only for static and dynamic library output types. If you generated a MEX function, you must change the project settings to use `Static Library` or `Dynamic Library` for **Build type** and run **Generate code** again.

1. To open the GPU Coder app, on the MATLAB toolstrip **Apps** tab, under **Code Generation**, click the app icon.
2. To open your project, click  , and then click `Open existing project`. Select the project, `myproject.prj`. On the **Generate Code** page, click the **Generate** arrow  .
3. In the **Generate** dialog box:
   - Set **Build type** to `Static Library` or `Dynamic Library`.
   - Clear the **Generate code only** check box.
   - You can leave the other settings to their default values.
4. To generate the CUDA code, click **Generate**. Click **Verify Code**.
5. In the command field, specify the test file (for example, `myfunction_test.m`) that calls the original MATLAB functions (for example, `myfunction`).
6. To start the SIL execution, click **Run Generated Code**. The GPU Coder app:
   - Generates a standalone library in `codegen\lib\myfunction`.
   - Generates SIL interface code in `codegen\lib\myfunction\sil`.
   - Runs the test file, replacing calls to the MATLAB function with calls to the generated code in the library.
   - Displays messages from the SIL execution in the **Test Output** tab.
7. Verify that the results from the SIL execution match the results from the original MATLAB functions.

8 To terminate the SIL execution process, click **Stop SIL Verification**. Alternatively, on the **Test Output** tab, click the link that follows `To terminate execution`.

SIL Execution from Command Line

To set up and start a SIL execution from the command line, you create a GPU Coder configuration object for library code generation, enable `config_obj.VerificationMode = 'SIL'`, use `codegen` function to generate the library code and the SIL interface, and use `coder.runTest` function to run the test file for your original MATLAB function. The following is a build script that automates the series of commands to perform SIL execution.

```
%% Create configuration object for static library.
config = coder.gpuConfig('lib');
config.GenerateReport = true;
config.VerificationMode = 'SIL';

%% Define argument types for entry-point 'mandelbrot_count'.
ARGS = cell(1,1);
ARGS{1} = cell(3,1);
ARGS{1}{1} = coder.typeof(0);
ARGS{1}{2} = coder.typeof(0,[1000 1000]);
ARGS{1}{3} = coder.typeof(0,[1000 1000]);

%% Invoke GPU Coder.
codegen -config config myfunction -args ARGS{1}

%% Run the test file with the sil interface
coder.runTest('myfunction_test', ['myfunction_sil.' mexext]);

%% Terminate SIL execution
clear myfunction_sil;
```

Numerical Differences Between CPU and GPU

Because of architectural differences between the CPU and GPU, numerical verification does not always match. This scenario is specially true when using `single` data type in your MATLAB code and performing accumulation operations on these `single` data type values. However, there are cases like the Mandelbrot example where even `double` data types cause numerical errors. One reason for this mismatch is that the GPU floating-point units use fused Floating-Point Multiply-Add (FMAD) instructions while the CPU does not use these instructions. It is also important to note that the CUDA compiler performs these instruction-level optimizations by default impacting the accuracy of the computed results. For example, the CUDA compiler fuses floating-point multiply and add instructions into a single instruction. This Floating-point Multiply-Add (FMAD) operation executes twice as fast compared to two single instructions but results in the loss of numerical accuracy. You can achieve tighter control over these optimizations by using intrinsic functions and compiler flags. To set compiler flags, see `coder.gpuConfig`. To integrate CUDA intrinsics, see “Legacy Code Integration”.

See Also

Apps
GPU Coder

Functions
`codegen` | `coder.gpuConfig` | `coder.gpu.kernelfun` | `gpucoder` | `coder.runTest`

Objects
`coder.gpuConfig` | `coder.CodeConfig` | `coder.EmbeddedCodeConfig`

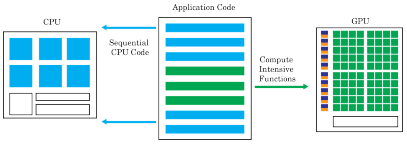
More About

- “Code Generation Reports”
- “Processor-In-The-Loop Execution from Command Line” (MATLAB Coder Support Package for NVIDIA Jetson and NVIDIA DRIVE Platforms)
- “Processor-In-The-Loop Execution with the GPU Coder App” (MATLAB Coder Support Package for NVIDIA Jetson and NVIDIA DRIVE Platforms)

About GPU Code Generation

GPU Programming Paradigm

GPU-accelerated computing follows a heterogeneous programming model. Highly parallelizable portions of the software application are mapped into kernels that execute on the physically separate GPU device, while the remainder of the sequential code still runs on the CPU. Each kernel is allocated several workers or threads, which are organized in blocks and grids. Every thread within the kernel executes concurrently with respect to each other.



The objective of GPU Coder is to take a sequential MATLAB program and generate partitioned, optimized CUDA code from it. This process involves:

- CPU/GPU partitioning — Identifying segments of code that run on the CPU and segments that run on the GPU. For the different ways GPU Coder identifies CUDA kernels, see “Kernel Creation”. Memory transfer costs between CPU and GPU are a significant consideration in the kernel creation algorithm.
- After kernel partitioning is complete, GPU Coder analyzes the data dependency between the CPU and GPU partitions. Data that is shared between the CPU and GPU are allocated on GPU memory (by using `cudaMalloc` or `cudaMallocManaged` APIs). The analysis also determines the minimum set of locations where data has to be copied between CPU and GPU by using `cudaMemcpy`. If using Unified Memory in CUDA, then the same analysis pass also determines the minimum locations in the code where `cudaDeviceSync` calls must be inserted to get the right functional behavior.
- Next, within each kernel, GPU Coder can choose to map data to shared memory or constant memory. If used wisely, these memories are part of the GPU memory hierarchy structure and can potentially result in greater memory bandwidth. For information on how GPU Coder chooses to map to shared memory, see “Stencil Processing”. For information on how GPU Coder chooses to map to constant memory, see `coder.gpu.constantMemory`.
- Once partitioning and memory allocation and transfer statements are in place, GPU Coder generates CUDA code that follows the partitioning and memory allocation decisions. The generated source code can be compiled into a MEX target to be called from within MATLAB or into a shared library to be integrated with an external project. For information, see “Code Generation Using the Command Line Interface” on page 2-16.

See Also

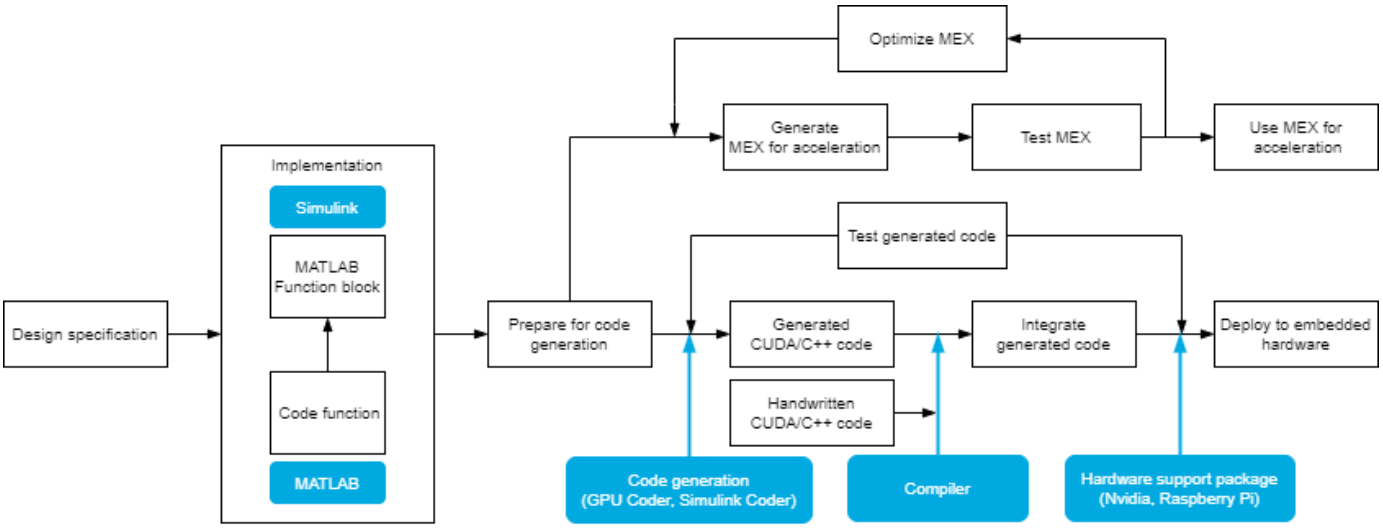
More About

- “GPU Code Generation Workflow” on page 4-3
- “Code Generation by Using the GPU Coder App” on page 2-2
- “Code Generation Using the Command Line Interface” on page 2-16
- “Kernel Creation from MATLAB Code”
- “Kernel Creation from Simulink Models”

GPU Code Generation Workflow

GPU Coder code generation technology produces CUDA C++ code and executable programs for algorithms. You can write algorithms programmatically by using MATLAB or graphically in the Simulink environment. You can generate code for MATLAB functions and Simulink MATLAB Function blocks that are useful for real-time and embedded applications. Because code generation is tightly integrated with the MATLAB and Simulink execution and simulation engines, the generated source code and executable programs match the functional behavior of MATLAB code executions and Simulink simulations to a high degree of accuracy.

The code generator supports two workflows for designing, implementing, and verifying generated CUDA code. This figure shows the design and deployment environment options.



See Also

More About

- “GPU Programming Paradigm” on page 4-2
- “Code Generation by Using the GPU Coder App” on page 2-2
- “Code Generation Using the Command Line Interface” on page 2-16
- “Kernel Creation from MATLAB Code”
- “Kernel Creation from Simulink Models”