GPU Coder™

Reference

MATLAB® & SIMULINK®

R2022b

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™ Reference

© COPYRIGHT 2017–2022 by The MathWorks, Inc.

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

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

Trademarks

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

Patents

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

Revision History

September 2017	Online only	New for Version 1.0 (Release 2017b)
March 2018	Online only	Revised for Version 1.1 (Release 2018a)
September 2018	Online only	Revised for Version 1.2 (Release 2018b)
March 2019	Online only	Revised for Version 1.3 (Release 2019a)
September 2019	Online only	Revised for Version 1.4 (Release 2019b)
March 2020	Online only	Revised for Version 1.5 (Release 2020a)
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)
September 2022	Online only	Revised for Version 2.4 (Release 2022b)

Contents

1 Apps

2 Functions

3 GPU Acceleration Parameters

Model Configuration Parameters: GPU Acceleration **3-2**

Simulation Target: GPU Acceleration Tab Overview **3-3**

Custom compute capability **3-4**
- Description **3-4**
- Settings **3-4**
- Dependencies **3-4**
- Command-Line Information **3-4**

Dynamic memory allocation threshold **3-5**
- Description **3-5**
- Settings **3-5**
- Dependencies **3-5**
- Command-Line Information **3-5**

Stack size per GPU thread **3-6**
- Description **3-6**
- Settings **3-6**
- Dependencies **3-6**
- Command-Line Information **3-6**

Include error checks in generated code **3-7**
- Description **3-7**
- Settings **3-7**
- Dependencies **3-7**
- Command-Line Information **3-7**

Additional compiler flags **3-8**
- Description **3-8**
- Settings **3-8**
- Dependencies **3-8**

Command-Line Information 3-8

4 Code Generation Parameters: GPU Code

Model Configuration Parameters: GPU Code 4-2

Code Generation: GPU Code Tab Overview 4-3

GPU index 4-4
Description 4-4
Settings 4-4
Dependencies 4-4
Command-Line Information 4-4

Compute capability 4-5
Description 4-5
Settings 4-5
Dependencies 4-5
Command-Line Information 4-5

Custom compute capability 4-6
Description 4-6
Settings 4-6
Dependencies 4-6
Command-Line Information 4-6

Memory mode 4-7
Description 4-7
Settings 4-7
Dependencies 4-7
Command-Line Information 4-7
Compatibility Considerations 4-7

Maximum blocks per kernel 4-9
Description 4-9
Settings 4-9
Dependencies 4-9
Command-Line Information 4-9

Dynamic memory allocation threshold 4-10
Description 4-10
Settings 4-10
Dependencies 4-10
Command-Line Information 4-10

Memory Manager 4-11
Description 4-11
Settings 4-11
Dependencies 4-11
Command-Line Information 4-11

Stack size per GPU thread 4-12
Description 4-12
Settings 4-12
Dependencies 4-12
Command-Line Information 4-12

Include error checks in generated code 4-13
Description 4-13
Settings 4-13
Dependencies 4-13
Command-Line Information 4-13

Kernel name prefix 4-14
Description 4-14
Settings 4-14
Dependencies 4-14
Command-Line Information 4-14

Additional compiler flags 4-15
Description 4-15
Settings 4-15
Dependencies 4-15
Command-Line Information 4-15

cuBLAS 4-16
Description 4-16
Settings 4-16
Dependencies 4-16
Command-Line Information 4-16

cuSOLVER 4-17
Description 4-17
Settings 4-17
Dependencies 4-17
Command-Line Information 4-17

cuFFT 4-18
Description 4-18
Settings 4-18
Dependencies 4-18
Command-Line Information 4-18

Apps

GPU Coder

Generate GPU code from MATLAB code

Description

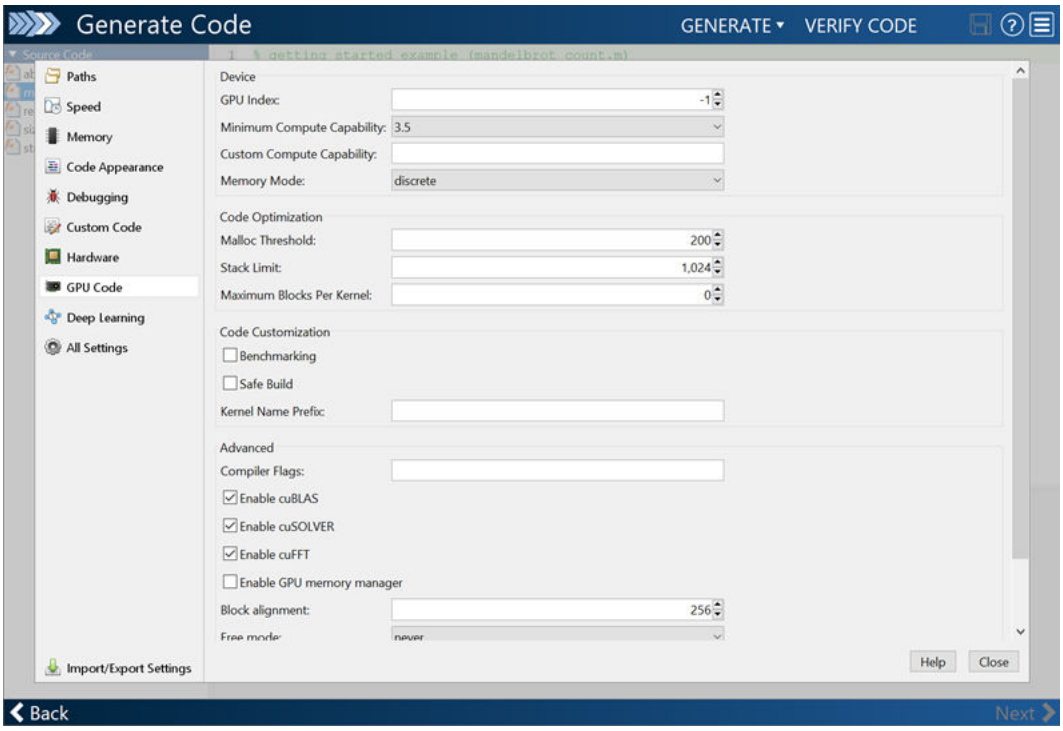
The **GPU Coder** app generates CUDA® C code from MATLAB® code. You can generate:

- CUDA C source code, static libraries, dynamically linked libraries, and executables that you can integrate into existing CUDA C applications outside of MATLAB.
- MEX functions for accelerated versions of your MATLAB functions.

The workflow-based user interface steps you through the code generation process. Using the app, you can:

- Create a project or open an existing project. The project specifies the input files, entry-point function input types, and build configuration.
- Review code generation readiness issues, including unsupported functions.
- Check your MATLAB function for run-time issues.
- Fix issues in your MATLAB code using the integrated editor.
- Convert double-precision MATLAB code to single-precision C code (requires a Fixed-Point Designer™ license).
- See static code metrics.
- Verify the numerical behavior of generated code using software-in-the-loop execution (requires an Embedded Coder® license).
- Export project settings in the form of a MATLAB script.
- Access generated files.
- Package generated files as a single zip file for deployment outside of MATLAB.

Note The app is not supported on MATLAB Online™.



Open the GPU Coder App

- MATLAB toolstrip: On the **Apps** tab, under **Code Generation**, click the app icon.
- MATLAB Command Window: Enter `gpucoder`.

Programmatic Use

```
gpucoder
```

Version History

Introduced in R2017b

See Also

Apps

MATLAB Coder | **GPU Environment Check**

Functions

`codegen` | `coder.checkGpuInstall`

Objects

`coder.gpuEnvConfig`

Topics

"GPU Programming Paradigm"
"Installing Prerequisite Products"
"Setting Up the Prerequisite Products"
"Code Generation by Using the GPU Coder App"
"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"

GPU Environment Check

Verify and set up GPU code generation environment

Description

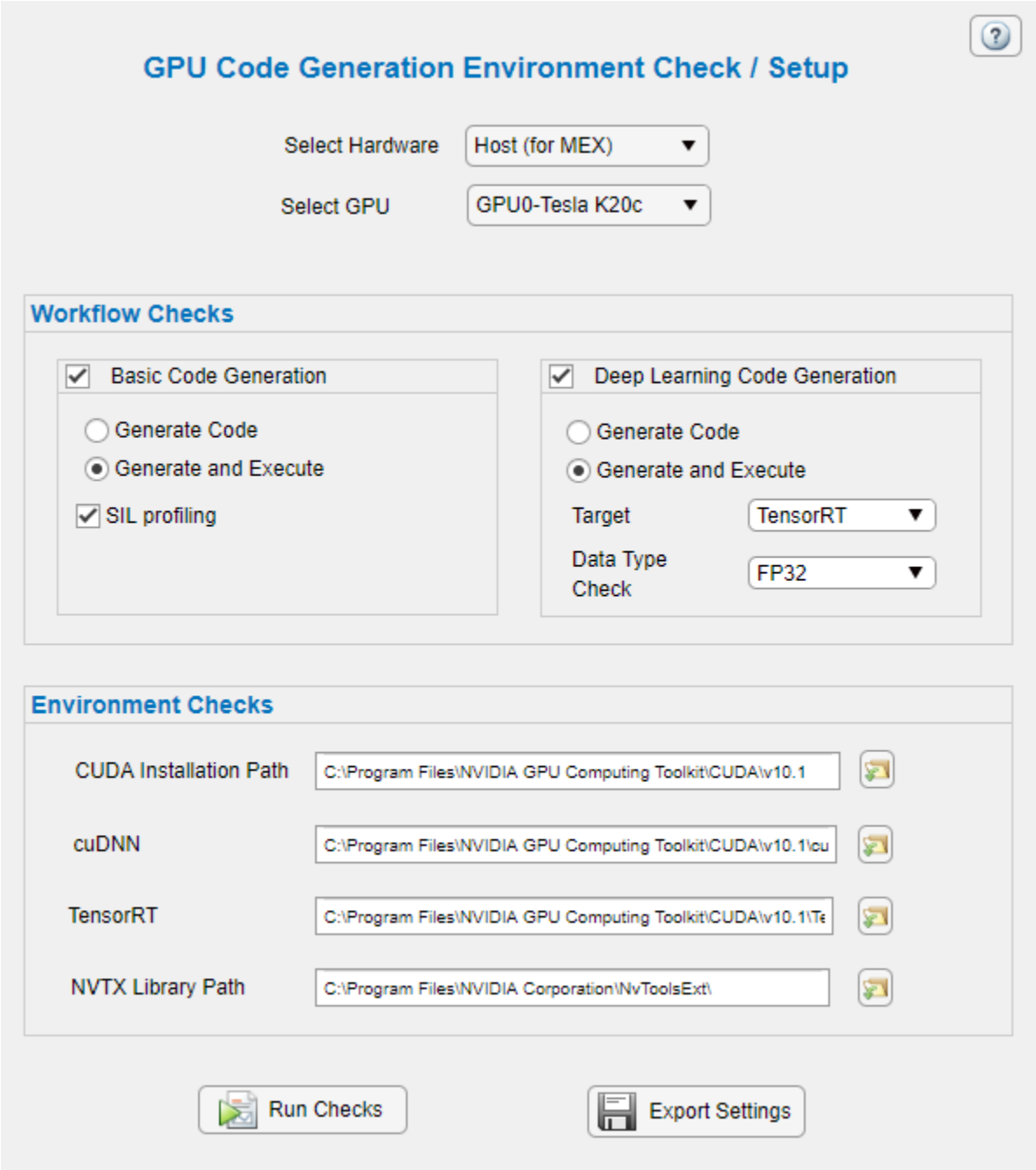
The **GPU Environment Check** app is an interactive tool to verify and set up the GPU code generation environment. You can run these checks for your development computer and hardware platforms such as the NVIDIA® DRIVE and Jetson. Using the app, you can:

- Verify your development computer environment for all the required NVIDIA compilers and libraries for GPU code generation. These tests check for the presence of:
  - A CUDA compatible GPU device.
  - The CUDA run time.
  - The cuFFT, cuSOLVER, and cuBLAS libraries.
  - The CUDA Deep Neural Network libraries.
  - NVIDIA TensorRT – high performance deep learning inference optimizer and run-time libraries.
  - NVIDIA `nvtx` libraries required for profiling.
- Perform basic code generation and execution tests on the development computer. These tests validate code execution by comparing the results to the MATLAB simulation.
- Perform deep learning code generation and execution tests on the development computer. In this test, you can target the cuDNN or TensorRT libraries.
- Connect to NVIDIA boards such as DRIVE and Jetson and perform code generation and execution tests. To perform these tests, you must install the MATLAB Coder™ Support Package for NVIDIA Jetson® and NVIDIA DRIVE® Platforms
- Specify the location of the libraries by using the app and generating a MATLAB script that sets up the environment variables required by GPU Coder.

Note The app is not supported on MATLAB Online.

For more information, see “The GPU Environment Check and Setup App”.

Before using this app, install and set up the required prerequisite third-party compilers, libraries, and tools. For more information, see “Installing Prerequisite Products” and “Setting Up the Prerequisite Products”.



Open the GPU Environment Check App

- MATLAB Command Window: Enter `gpucoderSetup`.

Examples

- “GPU Programming Paradigm”
- “Installing Prerequisite Products”
- “Setting Up the Prerequisite Products”
- “The GPU Environment Check and Setup App”
- “Verify Setup”
- “Code Generation by Using the GPU Coder App”
- “Code Generation for Deep Learning Networks by Using cuDNN”
- “Code Generation for Deep Learning Networks by Using TensorRT”
- “Code Generation by Using the GPU Coder App”

Version History

Introduced in R2019a

See Also

Apps
GPU Coder

Functions
`codegen` | `coder.checkGpuInstall`

Objects
`coder.gpuEnvConfig`

Topics
“GPU Programming Paradigm”
“Installing Prerequisite Products”
“Setting Up the Prerequisite Products”
“The GPU Environment Check and Setup App”
“Verify Setup”
“Code Generation by Using the GPU Coder App”
“Code Generation for Deep Learning Networks by Using cuDNN”
“Code Generation for Deep Learning Networks by Using TensorRT”
“Code Generation by Using the GPU Coder App”

Functions

analyzeNetworkForCodegen

Analyze deep learning network for code generation

Syntax

```
result = analyzeNetworkForCodegen(net)
___ = analyzeNetworkForCodegen( ___ ,Name,Value)
```

Description

`result = analyzeNetworkForCodegen(net)` analyzes the deep learning network `net` for code generation and reports network and layer compatibility issues. Network must be a `SeriesNetwork`, `DAGNetwork`, or `dlnetwork` object. By default, the function validates against a set of default CPU and GPU deep learning library targets.

`___ = analyzeNetworkForCodegen( ___ ,Name,Value)` analyzes the deep learning network `net` for code generation by using the options specified by one or more `Name,Value` pair arguments.

Examples

Check Network for Code Generation Compatibility

This example shows how to check code generation compatibility of the MobileNet-v2 network by using the `analyzeNetworkForCodegen` function.

You can use the the `analyzeNetworkForCodegen` function to determine network and layer compatibility issues when targeting a variety of CPU and GPU deep learning library targets.

Download MobileNet-v2 Support Package

This example uses the pretrained version of the MobileNet-v2 available through the *Deep Learning Toolbox™ Model for MobileNet-v2 Network* support package.

MobileNet-v2 is a convolutional neural network that is 53 layers deep. The pretrained version of the network is trained on more than a million images from the ImageNet database. The pretrained network has an image input size of 224-by-224 and can classify images into 1000 object categories, such as keyboard, mouse, pencil, and many animals.

```
mobilenetv2
```

```
ans = 
  DAGNetwork with properties:

         Layers: [154x1 nnet.cnn.layer.Layer]
    Connections: [163x2 table]
     InputNames: {'input_1'}
    OutputNames: {'ClassificationLayer_Logits'}
```

If the Deep Learning Toolbox Model for MobileNet-v2 Network support package is not installed, then the function provides a link to the required support package in the Add-On Explorer. To install the support package, click the link, and then click **Install**.

Analyze Network for Code Generation

Run the `analyzeNetworkForCodegen` function for `mobilenetv2`, specifying the target libraries to analyze. The `analyzeNetworkForCodegen` function requires the *MATLAB® Coder™ Interface for Deep Learning Libraries* and the *GPU Coder™ Interface for Deep Learning Libraries* support packages. To install the required support packages, use the Add-On Explorer.

```
targetLibraries = {'cudnn', 'tensorrt', 'arm-compute-mali',...
    'arm-compute', 'mkldnn','none','cmsis-nn'};
S = analyzeNetworkForCodegen(mobilenetv2,TargetLibrary  = targetLibraries);
```

```
                        Supported
                        _________    ____________________________________________________________

    cudnn                 "Yes"      ""
    tensorrt              "Yes"      ""
    arm-compute-mali      "Yes"      ""
    arm-compute           "Yes"      ""
    mkldnn                "Yes"      ""
    none                  "No"       "Found 1 unsupported layer type(s). View incompatible layer
    cmsis-nn              "No"       "Found 6 unsupported layer type(s). View incompatible layer
```

To access the analysis results for the CMSIS-NN target, use the following command.

```
disp(S(7))
```

```
             TargetLibrary: 'cmsis-nn'
                 Supported: 0
        NetworkDiagnostics: [0x0 table]
          LayerDiagnostics: [150x3 table]
    IncompatibleLayerTypes: [6x1 string]
```

Display the layer types not supported for CMSIS-NN code generation.

```
S(7).IncompatibleLayerTypes
```

```
ans = 6x1 string
    "AdditionLayer"
    "BatchNormalizationLayer"
    "ClippedReLULayer"
    "Convolution2DLayer"
    "GlobalAveragePooling2DLayer"
    "GroupedConvolution2DLayer"
```

Input Arguments

net — Deep learning network to analyze

`SeriesNetwork` object | `DAGNetwork` object | `dlnetwork` object

Network to analyze for code generation. Network can be a `SeriesNetwork`, `DAGNetwork`, or `dlnetwork` object for custom training loops or custom pruning loops.

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `resultStruct = analyzeNetworkForCodegen(mobilenetv2, TargetLibrary = ["mkldnn", "none"]);`

`TargetLibrary` — Code generation target library
character vector | string scalar

Target library for deep learning code generation, specified as one of the values in this table.

Value	Description
`'none'`	For generating code that does not use any third-party library.
`'arm-compute'`	For generating code that uses the ARM® Compute Library.
`'mkldnn'`	For generating code that uses the Intel® Math Kernel Library for Deep Neural Networks (Intel MKL-DNN).
`'cmsis-nn'`	Common Microcontroller Software Interface Standard - Neural Network (CMSIS-NN) library. Requires the MATLAB Coder Interface for Deep Learning Libraries.
`'cudnn'`	For generating code that uses the CUDA Deep Neural Network library (cuDNN). This option requires GPU Coder.
`'tensorrt'`	For generating code that takes advantage of the NVIDIA TensorRT – high performance deep learning inference optimizer and run-time library. This option requires GPU Coder.

`Quiet` — Suppress report display
`false` (default) | `true`

Scalar `boolean` value to suppress report display. By default, the function displays the analysis in verbose mode.

Output Arguments

`result` — Analysis output
structure array

Results of checking the deep learning network for code generation compatibility, returned as a `1`-by-`N` structure, where `N` is the number of target libraries to check for.

Version History

Introduced in R2022b

See Also

Functions

`codegen` | `coder.loadDeepLearningNetwork`

Topics

"Supported Networks, Layers, and Classes"
"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"
"Analyze Network for Code Generation"

gpucoder.atomicAdd

Package: `gpucoder`

Atomically add a specified value to a variable in global or shared memory

Syntax

```
[A,oldA] = gpucoder.atomicAdd(A,B)
```

Description

`[A,oldA] = gpucoder.atomicAdd(A,B)` adds `B` to the value of `A` in global or shared memory and writes the result back into A. The operation is atomic in a sense that the entire read-modify-write operation is guaranteed to be performed without interference from other threads. The order of the input and output arguments must match the syntax provided.

Examples

Simple CUDA Atomic Addition

Perform a simple atomic addition operation by using the `gpucoder.atomicAdd` function and generate CUDA code that calls appropriate CUDA `atomicAdd()` APIs.

In one file, write an entry-point function `myAtomicAdd` that accepts matrix inputs `a` and `b`.

```
function [a,oldA] = myAtomicAdd(a,b)

oldA = coder.nullcopy(a);

for i = 1:size(a,1)
    for j = 1:size(a,2)
        for k = 1:size(a,3)
            [a(i,j,k),oldA(i,j,k)] = gpucoder.atomicAdd(a(i,j,k),b(i,j,k));
        end
    end
end
```

To create a type for a matrix of doubles for use in code generation, use the `coder.newtype` function.

```
A = coder.newtype('single', [30 30 20], [1 0 1]);
B = coder.newtype('single', [30 30 20], [1 0 1]);
inputArgs = {A,B};
```

To generate a CUDA library, use the `codegen` function.

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

codegen -config cfg -args inputArgs myAtomicAdd -d myAtomicAdd
```

The generated CUDA code contains the `myAtomicAdd_kernel1` kernel with calls to the `atomicAdd()` CUDA APIs.

```
//
// File: myAtomicAdd.cu
```

```
//
...
static __global__ __launch_bounds__(1024, 1) void myAtomicAdd_kernel1(
    const float b_data[], const int b_size[3], int a_size[3],
    const int oldA_size[3], const int b_a_size, const int i, float oldA_data[],
    float a_data[])
{
  unsigned long loopEnd;
  unsigned long threadId;
...

  for (unsigned long idx{threadId}; idx <= loopEnd; idx += threadStride) {
    unsigned long tmpIndex;
    int b_i;
    int j;
    int k;
    k = static_cast<int>(idx % (static_cast<unsigned long>(b_a_size) + 1UL));
    tmpIndex = (idx - static_cast<unsigned long>(k)) /
               (static_cast<unsigned long>(b_a_size) + 1UL);
    j = static_cast<int>(tmpIndex % 30UL);
    tmpIndex = (tmpIndex - static_cast<unsigned long>(j)) / 30UL;
    b_i = static_cast<int>(tmpIndex);
    oldA_data[(b_i + oldA_size[0] * j) + oldA_size[0] * 30 * k] =
        atomicAdd(&a_data[(b_i + a_size[0] * j) + a_size[0] * 30 * k],
                  b_data[(b_i + b_size[0] * j) + b_size[0] * 30 * k]);
  }
}
...

void myAtomicAdd(float a_data[], int a_size[3], const float b_data[],
                 const int b_size[3], float oldA_data[], int oldA_size[3])
{
  dim3 block;
  dim3 grid;
...

    cudaMemcpy(gpu_a_data, a_data, a_size[0] * (30 * a_size[2]) * sizeof(float),
               cudaMemcpyHostToDevice);
    myAtomicAdd_kernel1<<<grid, block>>>(gpu_b_data, *gpu_b_size, *gpu_a_size,
                                         *gpu_oldA_size, b_a_size, i,
                                         gpu_oldA_data, gpu_a_data);
    oldA_data_dirtyOnGpu = true;
    cudaMemcpy(a_data, gpu_a_data, a_size[0] * (30 * a_size[2]) * sizeof(float),
               cudaMemcpyDeviceToHost);
  }
...

}
```

Input Arguments

A, B — Operands

scalars | vectors | matrices | multidimensional arrays

Operands, specified as scalars, vectors, matrices, or multidimensional arrays. Inputs A and B must satisfy the following requirements:

- Have the same data type.
- Have the same size or have sizes that are compatible. For example, A is an M-by-N matrix and B is a scalar or 1-by-N row vector.
- Requires CUDA device with a minimum compute capability of 6.0 when the data type is `double`.

Data Types: `double` | `single` | `int32` | `uint32` | `uint64`

Limitations

- Function handle input to the `gpucoder.stencilKernel` pragma cannot contain calls to atomic functions. For example,

```
out1 = gpucoder.stencilKernel(@myAtomicAdd,A,[3 3],'same',B);
```

Version History

Introduced in R2021b

See Also

Functions

gpucoder.atomicCAS | gpucoder.atomicDec | gpucoder.atomicExch | gpucoder.atomicInc | gpucoder.atomicMax | gpucoder.atomicMin | gpucoder.atomicSub

Topics

"GPU Programming Paradigm"
"Code Generation by Using the GPU Coder App"
"Code Generation Using the Command Line Interface"
"Kernels from Element-Wise Loops"

gpucoder.atomicAnd

Package: `gpucoder`

Atomically perform bit-wise AND between a specified value and a variable in global or shared memory

Syntax

```
[A,oldA] = gpucoder.atomicAnd(A,B)
```

Description

`[A,oldA] = gpucoder.atomicAnd(A,B)` perform bit-wise AND between `B` and the value of `A` in global or shared memory and writes the results back into A. The operation is atomic in a sense that the entire read-modify-write operation is guaranteed to be performed without interference from other threads. The order of the input and output arguments must match the syntax provided.

Examples

Bit-Wise AND Using CUDA atomicAnd

Perform a simple atomic addition operation by using the `gpucoder.atomicAnd` function and generate CUDA code that calls appropriate CUDA `atomicAnd()` APIs.

In one file, write an entry-point function `myAtomicAnd` that accepts matrix inputs `a` and `b`.

```
function a = myAtomicAnd(a,b)

coder.gpu.kernelfun;
for i =1:numel(a)
    [a(i),~] = gpucoder.atomicAnd(a(i), b);
end

end
```

To create a type for a matrix of doubles for use in code generation, use the `coder.newtype` function.

```
A = coder.newtype('uint32', [1 30], [0 1]);
B = coder.newtype('uint32', [1 1], [0 0]);
inputArgs = {A,B};
```

To generate a CUDA library, use the `codegen` function.

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

codegen -config cfg -args inputArgs myAtomicAnd -d myAtomicAnd
```

The generated CUDA code contains the `myAtomicAnd_kernel1` kernel with calls to the `atomicAnd()` CUDA APIs.

```
//
// File: myAtomicAnd.cu
//
...
```

```
static __global__ __launch_bounds__(1024, 1) void myAtomicAnd_kernel1(
    const uint32_T b, const int32_T i, uint32_T a_data[])
{
  uint64_T loopEnd;
  uint64_T threadId;
...

  for (uint64_T idx{threadId}; idx <= loopEnd; idx += threadStride) {
    int32_T b_i;
    b_i = static_cast<int32_T>(idx);
    atomicAnd(&a_data[b_i], b);
  }
}
...

void myAtomicAnd(uint32_T a_data[], int32_T a_size[2], uint32_T b)
{
  dim3 block;
  dim3 grid;
...

    cudaMemcpy(gpu_a_data, a_data, a_size[1] * sizeof(uint32_T),
               cudaMemcpyHostToDevice);
    myAtomicAnd_kernel1<<<grid, block>>>(b, i, gpu_a_data);
    cudaMemcpy(a_data, gpu_a_data, a_size[1] * sizeof(uint32_T),
               cudaMemcpyDeviceToHost);
...

}
```

Input Arguments

A, B — Operands

scalars | vectors | matrices | multidimensional arrays

Operands, specified as scalars, vectors, matrices, or multidimensional arrays. Inputs A and B must satisfy the following requirements:

- Have the same data type.
- Have the same size or have sizes that are compatible. For example, A is an M-by-N matrix and B is a scalar or 1-by-N row vector.

Data Types: `int32` | `uint32` | `uint64`

Limitations

- Function handle input to the `gpucoder.stencilKernel` pragma cannot contain calls to atomic functions. For example,

  ```
  out1 = gpucoder.stencilKernel(@myAtomicAnd,A,[3 3],'same',B);
  ```

Version History

Introduced in R2021b

See Also

Functions

`gpucoder.atomicOr` | `gpucoder.atomicXor`

Topics

"GPU Programming Paradigm"

“Code Generation by Using the GPU Coder App”
“Code Generation Using the Command Line Interface”
“Kernels from Element-Wise Loops”

gpucoder.atomicCAS

Package: `gpucoder`

Atomically compare and swap the value of a variable in global or shared memory

Syntax

```
[A,oldA] = gpucoder.atomicCAS(A,B,C)
```

Description

`[A,oldA] = gpucoder.atomicCAS(A,B,C)` compares `B` to the value of `A` in global or shared memory and if the values are the same writes the value of `C` into A. The operation is atomic in a sense that the entire read-modify-write operation is guaranteed to be performed without interference from other threads. The order of the input and output arguments must match the syntax provided.

Examples

Compare and Swap Using CUDA atomicCAS

Perform a simple atomic compare and swap operation by using the `gpucoder.atomicCAS` function and generate CUDA code that calls appropriate CUDA `atomicCAS()` APIs.

In one file, write an entry-point function `myAtomicCAS` that accepts matrix inputs `a`,`b`, and `c`.

```
function a = myAtomicCAS(a,b,c)

coder.gpu.kernelfun;
for i =1:numel(a)
    [a(i),~] = gpucoder.atomicCAS(a(i), b, c);
end

end
```

To create a type for a matrix of doubles for use in code generation, use the `coder.newtype` function.

```
A = coder.newtype('uint32', [1 30], [0 1]);
B = coder.newtype('uint32', [1 1], [0 0]);
C = coder.newtype('uint32', [1 1], [0 0]);
inputArgs = {A,B,C};
```

To generate a CUDA library, use the `codegen` function.

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

codegen -config cfg -args inputArgs myAtomicCAS -d myAtomicCAS
```

The generated CUDA code contains the `myAtomicCAS_kernel1` kernel with calls to the `atomicCAS()` CUDA APIs.

```
//
// File: myAtomicCAS.cu
//
...
```

```
static __global__ __launch_bounds__(1024, 1) void myAtomicCAS_kernel1(
    const uint32_T c, const uint32_T b, const int32_T i, uint32_T a_data[])
{
  uint64_T loopEnd;
  uint64_T threadId;

...

  for (uint64_T idx{threadId}; idx <= loopEnd; idx += threadStride) {
    int32_T b_i;
    b_i = static_cast<int32_T>(idx);
    atomicCAS(&a_data[b_i], b, c);
  }
}
...

void myAtomicCAS(uint32_T a_data[], int32_T a_size[2], uint32_T b, uint32_T c)
{
  dim3 block;
  dim3 grid;
...

  if (validLaunchParams) {
    cudaMemcpy(gpu_a_data, a_data, a_size[1] * sizeof(uint32_T),
               cudaMemcpyHostToDevice);
    myAtomicCAS_kernel1<<<grid, block>>>(c, b, i, gpu_a_data);
    cudaMemcpy(a_data, gpu_a_data, a_size[1] * sizeof(uint32_T),
               cudaMemcpyDeviceToHost);
...

}
```

Input Arguments

A, B, C — Operands

scalars | vectors | matrices | multidimensional arrays

Operands, specified as scalars, vectors, matrices, or multidimensional arrays. Inputs `A`, `B`, and `C` must satisfy the following requirements:

- Have the same data type.
- Have the same size or have sizes that are compatible. For example, `A` is an `M`-by-`N` matrix and `B`,`C` is a scalar or `1`-by-`N` row vector.

Data Types: `int32` | `uint32` | `uint64`

Version History

Introduced in R2021b

See Also

Functions

`gpucoder.atomicAdd` | `gpucoder.atomicDec` | `gpucoder.atomicExch` | `gpucoder.atomicInc` | `gpucoder.atomicMax` | `gpucoder.atomicMin` | `gpucoder.atomicSub`

Topics

"GPU Programming Paradigm"
"Code Generation by Using the GPU Coder App"
"Code Generation Using the Command Line Interface"
"Kernels from Element-Wise Loops"

gpucoder.atomicDec

Package: `gpucoder`

Atomically decrement a variable in global or shared memory within a specified upper bound

Syntax

```
[A,oldA] = gpucoder.atomicDec(A,B)
```

Description

`[A,oldA] = gpucoder.atomicDec(A,B)` decrements the value of `A` in global or shared memory within the upper bound `B`. The operation is atomic in a sense that the entire read-modify-write operation is guaranteed to be performed without interference from other threads. The order of the input and output arguments must match the syntax provided.

Examples

Decrement within a Upper Bound Using CUDA atomicDec

Perform a simple atomic wrap around decrement operation by using the `gpucoder.atomicDec` function and generate CUDA code that calls appropriate CUDA `atomicDec()` APIs.

In one file, write an entry-point function `myAtomicDec` that accepts matrix inputs `a` and `b`.

```
function a = myAtomicDec(a,b)

coder.gpu.kernelfun;
for i =1:numel(a)
    [a(i),~] = gpucoder.atomicDec(a(i), b);
end

end
```

To create a type for a matrix of doubles for use in code generation, use the `coder.newtype` function.

```
A = coder.newtype('uint32', [1 30], [0 1]);
B = coder.newtype('uint32', [1 1], [0 0]);
inputArgs = {A,B};
```

To generate a CUDA library, use the `codegen` function.

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

codegen -config cfg -args inputArgs myAtomicDec -d myAtomicDec
```

The generated CUDA code contains the `myAtomicDec_kernel1` kernel with calls to the `atomicDec()` CUDA APIs.

```
//
// File: myAtomicDec.cu
//
...
```

```
static __global__ __launch_bounds__(1024, 1) void myAtomicDec_kernel1(
    const uint32_T b, const int32_T i, uint32_T a_data[])
{
  uint64_T loopEnd;
  uint64_T threadId;

...

  for (uint64_T idx{threadId}; idx <= loopEnd; idx += threadStride) {
    int32_T b_i;
    b_i = static_cast<int32_T>(idx);
    atomicDec(&a_data[b_i], b);
  }
}
...

void myAtomicDec(uint32_T a_data[], int32_T a_size[2], uint32_T b)
{
  dim3 block;
  dim3 grid;
...

    cudaMemcpy(gpu_a_data, a_data, a_size[1] * sizeof(uint32_T),
               cudaMemcpyHostToDevice);
    myAtomicDec_kernel1<<<grid, block>>>(b, i, gpu_a_data);
    cudaMemcpy(a_data, gpu_a_data, a_size[1] * sizeof(uint32_T),
               cudaMemcpyDeviceToHost);
...

}
```

Input Arguments

A, B — Operands

scalars | vectors | matrices | multidimensional arrays

Operands, specified as scalars, vectors, matrices, or multidimensional arrays. Inputs A and B must satisfy the following requirements:

- Have the same data type.
- Have the same size or have sizes that are compatible. For example, A is an M-by-N matrix and B is a scalar or 1-by-N row vector.

Data Types: `uint32`

Limitations

- Function handle input to the `gpucoder.stencilKernel` pragma cannot contain calls to atomic functions. For example,

  ```
  out1 = gpucoder.stencilKernel(@myAtomicDec,A,[3 3],'same',B);
  ```

Version History

Introduced in R2021b

See Also

Functions

`gpucoder.atomicAdd` | `gpucoder.atomicCAS` | `gpucoder.atomicExch` | `gpucoder.atomicInc` | `gpucoder.atomicMax` | `gpucoder.atomicMin` | `gpucoder.atomicSub`

Topics

"GPU Programming Paradigm"
"Code Generation by Using the GPU Coder App"
"Code Generation Using the Command Line Interface"
"Kernels from Element-Wise Loops"

gpucoder.atomicExch

Package: `gpucoder`

Atomically exchange a variable in global or shared memory with the specified value

Syntax

```
[A,oldA] = gpucoder.atomicExch(A,B)
```

Description

`[A,oldA] = gpucoder.atomicExch(A,B)` atomically exchanges the value of `A` in global or shared memory with the value in `B` and writes the result back into A. The operation is atomic in a sense that the entire read-modify-write operation is guaranteed to be performed without interference from other threads. The order of the input and output arguments must match the syntax provided.

Examples

Simple CUDA Atomic Exchange

Perform a simple atomic exchange operation by using the `gpucoder.atomicExch` function and generate CUDA code that calls appropriate CUDA `atomicExch()` APIs.

In one file, write an entry-point function `myAtomicExch` that accepts matrix inputs `a` and `b`.

```
function a = myAtomicExch(a,b)

coder.gpu.kernelfun;
for i =1:numel(a)
    [a(i),~] = gpucoder.atomicExch(a(i), b);
end

end
```

To create a type for a matrix of doubles for use in code generation, use the `coder.newtype` function.

```
A = coder.newtype('single', [1 30], [0 1]);
B = coder.newtype('single', [1 1], [0 0]);
inputArgs = {A,B};
```

To generate a CUDA library, use the `codegen` function.

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

codegen -config cfg -args inputArgs myAtomicExch -d myAtomicExch
```

The generated CUDA code contains the `myAtomicExch_kernel1` kernel with calls to the `atomicExch()` CUDA APIs.

```
//
// File: myAtomicExch.cu
//
...
```

```
static __global__ __launch_bounds__(1024, 1) void myAtomicExch_kernel1(
    const real32_T b, const int32_T i, real32_T a_data[])
{
  uint64_T loopEnd;
  uint64_T threadId;
...

  for (uint64_T idx{threadId}; idx <= loopEnd; idx += threadStride) {
    int32_T b_i;
    b_i = static_cast<int32_T>(idx);
    atomicExch(&a_data[b_i], b);
  }
}
...

void myAtomicExch(real32_T a_data[], int32_T a_size[2], real32_T b)
{
  dim3 block;
  dim3 grid;
...
{
    cudaMemcpy(gpu_a_data, a_data, a_size[1] * sizeof(real32_T),
               cudaMemcpyHostToDevice);
    myAtomicExch_kernel1<<<grid, block>>>(b, i, gpu_a_data);
    cudaMemcpy(a_data, gpu_a_data, a_size[1] * sizeof(real32_T),
               cudaMemcpyDeviceToHost);
...

}
```

Input Arguments

A, B — Operands

scalars | vectors | matrices | multidimensional arrays

Operands, specified as scalars, vectors, matrices, or multidimensional arrays. Inputs A and B must satisfy the following requirements:

- Have the same data type.
- Have the same size or have sizes that are compatible. For example, A is an M-by-N matrix and B is a scalar or 1-by-N row vector.

Data Types: `single` | `int32` | `uint32` | `uint64`

Limitations

- Function handle input to the `gpucoder.stencilKernel` pragma cannot contain calls to atomic functions. For example,

  ```
  out1 = gpucoder.stencilKernel(@myAtomicExch,A,[3 3],'same',B);
  ```

Version History

Introduced in R2021b

See Also

Functions

`gpucoder.atomicAdd` | `gpucoder.atomicCAS` | `gpucoder.atomicDec` | `gpucoder.atomicInc` | `gpucoder.atomicMax` | `gpucoder.atomicMin` | `gpucoder.atomicSub`

Topics

"GPU Programming Paradigm"
"Code Generation by Using the GPU Coder App"
"Code Generation Using the Command Line Interface"
"Kernels from Element-Wise Loops"

gpucoder.atomicInc

Package: `gpucoder`

Atomically increment a variable in global or shared memory within a specified upper bound

Syntax

```
[A,oldA] = gpucoder.atomicInc(A,B)
```

Description

`[A,oldA] = gpucoder.atomicInc(A,B)` increments the value of `A` in global or shared memory within the upper bound `B`. If the value of `A` is greater than or equal to `B`, it is reset. The operation is atomic in a sense that the entire read-modify-write operation is guaranteed to be performed without interference from other threads. The order of the input and output arguments must match the syntax provided.

Examples

Increment within a Upper Bound Using CUDA atomicInc

Perform a simple atomic warp around increment operation by using the `gpucoder.atomicInc` function and generate CUDA code that calls appropriate CUDA `atomicInc()` APIs.

In one file, write an entry-point function `myAtomicInc` that accepts matrix inputs `a` and `b`.

```
function a = myAtomicInc(a,b)

coder.gpu.kernelfun;
for i =1:numel(a)
    [a(i),~] = gpucoder.atomicInc(a(i), b);
end

end
```

To create a type for a matrix of doubles for use in code generation, use the `coder.newtype` function.

```
A = coder.newtype('uint32', [1 30], [0 1]);
B = coder.newtype('uint32', [1 1], [0 0]);
inputArgs = {A,B};
```

To generate a CUDA library, use the `codegen` function.

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

codegen -config cfg -args inputArgs myAtomicInc -d myAtomicInc
```

The generated CUDA code contains the `myAtomicInc_kernel1` kernel with calls to the `atomicInc()` CUDA APIs.

```
//
// File: myAtomicInc.cu
//
...
```

```
static __global__ __launch_bounds__(1024, 1) void myAtomicInc_kernel1(
    const uint32_T b, const int32_T i, uint32_T a_data[])
{
  uint64_T loopEnd;
  uint64_T threadId;

...

  for (uint64_T idx{threadId}; idx <= loopEnd; idx += threadStride) {
    int32_T b_i;
    b_i = static_cast<int32_T>(idx);
    atomicInc(&a_data[b_i], b);
  }
}
...

void myAtomicInc(uint32_T a_data[], int32_T a_size[2], uint32_T b)
{
  dim3 block;
  dim3 grid;
...

    cudaMemcpy(gpu_a_data, a_data, a_size[1] * sizeof(uint32_T),
               cudaMemcpyHostToDevice);
    myAtomicInc_kernel1<<<grid, block>>>(b, i, gpu_a_data);
    cudaMemcpy(a_data, gpu_a_data, a_size[1] * sizeof(uint32_T),
               cudaMemcpyDeviceToHost);
...

}
```

Input Arguments

A, B — Operands

scalars | vectors | matrices | multidimensional arrays

Operands, specified as scalars, vectors, matrices, or multidimensional arrays. Inputs A and B must satisfy the following requirements:

- Have the same data type.
- Have the same size or have sizes that are compatible. For example, A is an M-by-N matrix and B is a scalar or 1-by-N row vector.

Data Types: uint32

Limitations

- Function handle input to the gpucoder.stencilKernel pragma cannot contain calls to atomic functions. For example,

  ```
  out1 = gpucoder.stencilKernel(@myAtomicInc,A,[3 3],'same',B);
  ```

Version History

Introduced in R2021b

See Also

Functions

gpucoder.atomicAdd | gpucoder.atomicCAS | gpucoder.atomicDec | gpucoder.atomicExch | gpucoder.atomicMax | gpucoder.atomicMin | gpucoder.atomicSub

Topics

"GPU Programming Paradigm"
"Code Generation by Using the GPU Coder App"
"Code Generation Using the Command Line Interface"
"Kernels from Element-Wise Loops"

gpucoder.atomicMax

Package: `gpucoder`

Atomically find the maximum between a specified value and a variable in global or shared memory

Syntax

```
[A,oldA] = gpucoder.atomicMax(A,B)
```

Description

`[A,oldA] = gpucoder.atomicMax(A,B)` compares `B` to the value of `A` in global or shared memory and writes the `max(A,B)` back into A. The operation is atomic in a sense that the entire read-modify-write operation is guaranteed to be performed without interference from other threads. The order of the input and output arguments must match the syntax provided.

Examples

Find the Maximum Using CUDA atomicMax

Perform a simple atomic addition operation by using the `gpucoder.atomicMax` function and generate CUDA code that calls appropriate CUDA `atomicMax()` APIs.

In one file, write an entry-point function `myAtomicMax` that accepts matrix inputs `a` and `b`.

```
function a = myAtomicMax(a,b)

coder.gpu.kernelfun;
for i =1:numel(a)
    [a(i),~] = gpucoder.atomicMax(a(i), b);
end

end
```

To create a type for a matrix of doubles for use in code generation, use the `coder.newtype` function.

```
A = coder.newtype('int32', [1 30], [0 1]);
B = coder.newtype('int32', [1 1], [0 0]);
inputArgs = {A,B};
```

To generate a CUDA library, use the `codegen` function.

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

codegen -config cfg -args inputArgs myAtomicMax -d myAtomicMax
```

The generated CUDA code contains the `myAtomicMax_kernel1` kernel with calls to the `atomicMax()` CUDA APIs.

```
//
// File: myAtomicMax.cu
//
...
```

```
static __global__ __launch_bounds__(1024, 1) void myAtomicMax_kernel1(
    const int32_T b, const int32_T i, int32_T a_data[])
{
  uint64_T loopEnd;
  uint64_T threadId;
...

  for (uint64_T idx{threadId}; idx <= loopEnd; idx += threadStride) {
    int32_T b_i;
    b_i = static_cast<int32_T>(idx);
    atomicMax(&a_data[b_i], b);
  }
}
...

void myAtomicMax(int32_T a_data[], int32_T a_size[2], int32_T b)
{
  dim3 block;
  dim3 grid;
...

    cudaMemcpy(gpu_a_data, a_data, a_size[1] * sizeof(int32_T),
               cudaMemcpyHostToDevice);
    myAtomicMax_kernel1<<<grid, block>>>(b, i, gpu_a_data);
    cudaMemcpy(a_data, gpu_a_data, a_size[1] * sizeof(int32_T),
               cudaMemcpyDeviceToHost);
...

}
```

Input Arguments

A, B — Operands

scalars | vectors | matrices | multidimensional arrays

Operands, specified as scalars, vectors, matrices, or multidimensional arrays. Inputs A and B must satisfy the following requirements:

- Have the same data type.
- Have the same size or have sizes that are compatible. For example, A is an M-by-N matrix and B is a scalar or 1-by-N row vector.

Data Types: `int32` | `uint32` | `uint64`

Limitations

- Function handle input to the `gpucoder.stencilKernel` pragma cannot contain calls to atomic functions. For example,

  ```
  out1 = gpucoder.stencilKernel(@myAtomicMax,A,[3 3],'same',B);
  ```

Version History

Introduced in R2021b

See Also

Functions

`gpucoder.atomicAdd` | `gpucoder.atomicCAS` | `gpucoder.atomicDec` | `gpucoder.atomicExch` | `gpucoder.atomicInc` | `gpucoder.atomicMin` | `gpucoder.atomicSub`

Topics

"GPU Programming Paradigm"
"Code Generation by Using the GPU Coder App"
"Code Generation Using the Command Line Interface"
"Kernels from Element-Wise Loops"

gpucoder.atomicMin

Package: `gpucoder`

Atomically find the minimum between a specified value and a variable in global or shared memory

Syntax

```
[A,oldA] = gpucoder.atomicMin(A,B)
```

Description

`[A,oldA] = gpucoder.atomicMin(A,B)` compares `B` to the value of `A` in global or shared memory and writes the `min(A,B)` back into A. The operation is atomic in a sense that the entire read-modify-write operation is guaranteed to be performed without interference from other threads. The order of the input and output arguments must match the syntax provided.

Examples

Find the Minimum Using CUDA atomicMin

Perform a simple atomic addition operation by using the `gpucoder.atomicMin` function and generate CUDA code that calls appropriate CUDA `atomicMin()` APIs.

In one file, write an entry-point function `myAtomicMin` that accepts matrix inputs `a` and `b`.

```
function a = myAtomicMin(a,b)

coder.gpu.kernelfun;
for i =1:numel(a)
    [a(i),~] = gpucoder.atomicMin(a(i), b);
end

end
```

To create a type for a matrix of doubles for use in code generation, use the `coder.newtype` function.

```
A = coder.newtype('uint32', [1 30], [0 1]);
B = coder.newtype('uint32', [1 1], [0 0]);
inputArgs = {A,B};
```

To generate a CUDA library, use the `codegen` function.

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

codegen -config cfg -args inputArgs myAtomicMin -d myAtomicMin
```

The generated CUDA code contains the `myAtomicMin_kernel1` kernel with calls to the `atomicMin()` CUDA APIs.

```
//
// File: myAtomicMin.cu
//
...
```

```
static __global__ __launch_bounds__(1024, 1) void myAtomicMin_kernel1(
    const uint32_T b, const int32_T i, uint32_T a_data[])
{
  uint64_T loopEnd;
  uint64_T threadId;
...

  for (uint64_T idx{threadId}; idx <= loopEnd; idx += threadStride) {
    int32_T b_i;
    b_i = static_cast<int32_T>(idx);
    atomicMin(&a_data[b_i], b);
  }
}
...

void myAtomicMin(uint32_T a_data[], int32_T a_size[2], uint32_T b)
{
  dim3 block;
  dim3 grid;
...

    cudaMemcpy(gpu_a_data, a_data, a_size[1] * sizeof(uint32_T),
               cudaMemcpyHostToDevice);
    myAtomicMin_kernel1<<<grid, block>>>(b, i, gpu_a_data);
    cudaMemcpy(a_data, gpu_a_data, a_size[1] * sizeof(uint32_T),
               cudaMemcpyDeviceToHost);
...

}
```

Input Arguments

A, B — Operands

scalars | vectors | matrices | multidimensional arrays

Operands, specified as scalars, vectors, matrices, or multidimensional arrays. Inputs A and B must satisfy the following requirements:

- Have the same data type.
- Have the same size or have sizes that are compatible. For example, A is an M-by-N matrix and B is a scalar or 1-by-N row vector.

Data Types: `int32` | `uint32` | `uint64`

Limitations

- Function handle input to the `gpucoder.stencilKernel` pragma cannot contain calls to atomic functions. For example,

  ```
  out1 = gpucoder.stencilKernel(@myAtomicMin,A,[3 3],'same',B);
  ```

Version History

Introduced in R2021b

See Also

Functions

`gpucoder.atomicAdd` | `gpucoder.atomicCAS` | `gpucoder.atomicDec` | `gpucoder.atomicExch` | `gpucoder.atomicInc` | `gpucoder.atomicMax` | `gpucoder.atomicSub`

Topics

"GPU Programming Paradigm"
"Code Generation by Using the GPU Coder App"
"Code Generation Using the Command Line Interface"
"Kernels from Element-Wise Loops"

gpucoder.atomicOr

Package: `gpucoder`

Atomically perform bit-wise OR between a specified value and a variable in global or shared memory

Syntax

```
[A,oldA] = gpucoder.atomicOr(A,B)
```

Description

`[A,oldA] = gpucoder.atomicOr(A,B)` perform bit-wise OR between `B` and the value of `A` in global or shared memory and writes the results back into A. The operation is atomic in a sense that the entire read-modify-write operation is guaranteed to be performed without interference from other threads. The order of the input and output arguments must match the syntax provided.

Examples

Bit-Wise OR Using CUDA atomicOr

Perform a simple atomic addition operation by using the `gpucoder.atomicOr` function and generate CUDA code that calls appropriate CUDA `atomicOr()` APIs.

In one file, write an entry-point function `myAtomicOr` that accepts matrix inputs `a` and `b`.

```
function a = myAtomicOr(a,b)

coder.gpu.kernelfun;
for i =1:numel(a)
    [a(i),~] = gpucoder.atomicOr(a(i), b);
end

end
```

To create a type for a matrix of doubles for use in code generation, use the `coder.newtype` function.

```
A = coder.newtype('uint32', [1 30], [0 1]);
B = coder.newtype('uint32', [1 1], [0 0]);
inputArgs = {A,B};
```

To generate a CUDA library, use the `codegen` function.

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

codegen -config cfg -args inputArgs myAtomicOr -d myAtomicOr
```

The generated CUDA code contains the `myAtomicOr_kernel1` kernel with calls to the `atomicOr()` CUDA APIs.

```
//
// File: myAtomicOr.cu
//
...
```

```
static __global__ __launch_bounds__(1024, 1) void myAtomicOr_kernel1(
    const uint32_T b, const int32_T i, uint32_T a_data[])
{
  uint64_T loopEnd;
  uint64_T threadId;
...

  for (uint64_T idx{threadId}; idx <= loopEnd; idx += threadStride) {
    int32_T b_i;
    b_i = static_cast<int32_T>(idx);
    atomicOr(&a_data[b_i], b);
  }
}
...

void myAtomicOr(uint32_T a_data[], int32_T a_size[2], uint32_T b)
{
  dim3 block;
  dim3 grid;
...

    cudaMemcpy(gpu_a_data, a_data, a_size[1] * sizeof(uint32_T),
               cudaMemcpyHostToDevice);
    myAtomicOr_kernel1<<<grid, block>>>(b, i, gpu_a_data);
    cudaMemcpy(a_data, gpu_a_data, a_size[1] * sizeof(uint32_T),
               cudaMemcpyDeviceToHost);
...

}
```

Input Arguments

A, B — Operands

scalars | vectors | matrices | multidimensional arrays

Operands, specified as scalars, vectors, matrices, or multidimensional arrays. Inputs A and B must satisfy the following requirements:

- Have the same data type.
- Have the same size or have sizes that are compatible. For example, A is an M-by-N matrix and B is a scalar or 1-by-N row vector.

Data Types: int32 | uint32 | uint64

Limitations

- Function handle input to the gpucoder.stencilKernel pragma cannot contain calls to atomic functions. For example,

  ```
  out1 = gpucoder.stencilKernel(@myAtomicOr,A,[3 3],'same',B);
  ```

Version History

Introduced in R2021b

See Also

Functions

gpucoder.atomicAnd | gpucoder.atomicXor

Topics

"GPU Programming Paradigm"

“Code Generation by Using the GPU Coder App”
“Code Generation Using the Command Line Interface”
“Kernels from Element-Wise Loops”

gpucoder.atomicSub

Package: `gpucoder`

Atomically subtract a specified value from a variable in global or shared memory

Syntax

```
[A,oldA] = gpucoder.atomicSub(A,B)
```

Description

`[A,oldA] = gpucoder.atomicSub(A,B)` subtracts `B` from the value of `A` in global or shared memory and writes the result back into A. The operation is atomic in a sense that the entire read-modify-write operation is guaranteed to be performed without interference from other threads. The order of the input and output arguments must match the syntax provided.

Examples

Simple CUDA Atomic Subtraction

Perform a simple atomic subtraction operation by using the `gpucoder.atomicSub` function and generate CUDA code that calls appropriate CUDA `atomicSub()` APIs.

In one file, write an entry-point function `myAtomicSub` that accepts matrix inputs `a` and `b`.

```
function a = myAtomicSub(a,b)

coder.gpu.kernelfun;
for i = 1:numel(a)
    [a(i),~] = gpucoder.atomicSub(a(i),b);
end

end
```

To create a type for a matrix of doubles for use in code generation, use the `coder.newtype` function.

```
A = coder.newtype('int32', [1 30], [0 1]);
B = coder.newtype('int32', [1 1], [0 0]);
inputArgs = {A,B};
```

To generate a CUDA library, use the `codegen` function.

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

codegen -config cfg -args inputArgs myAtomicSub -d myAtomicSub
```

The generated CUDA code contains the `myAtomicSub_kernel1` kernel with calls to the `atomicSub()` CUDA APIs.

```
//
// File: myAtomicSub.cu
//
...
```

```
static __global__ __launch_bounds__(1024, 1) void myAtomicSub_kernel1(
    const int32_T b, const int32_T i, int32_T a_data[])
{
  uint64_T loopEnd;
  uint64_T threadId;
...

  loopEnd = static_cast<uint64_T>(i - 1);
  for (uint64_T idx{threadId}; idx <= loopEnd; idx += threadStride) {
    int32_T b_i;
    b_i = static_cast<int32_T>(idx);
    atomicSub(&a_data[b_i], b);
  }
}
...

void myAtomicSub(int32_T a_data[], int32_T a_size[2], int32_T b)
{
  dim3 block;
  dim3 grid;
...

    cudaMemcpy(gpu_a_data, a_data, a_size[1] * sizeof(int32_T),
               cudaMemcpyHostToDevice);
    myAtomicSub_kernel1<<<grid, block>>>(b, i, gpu_a_data);
    cudaMemcpy(a_data, gpu_a_data, a_size[1] * sizeof(int32_T),
               cudaMemcpyDeviceToHost);
...

}
...

}
```

Input Arguments

A, B — Operands

scalars | vectors | matrices | multidimensional arrays

Operands, specified as scalars, vectors, matrices, or multidimensional arrays. Inputs A and B must satisfy the following requirements:

- Have the same data type.
- Have the same size or have sizes that are compatible. For example, A is an M-by-N matrix and B is a scalar or 1-by-N row vector.

Data Types: `int32` | `uint32`

Limitations

- Function handle input to the `gpucoder.stencilKernel` pragma cannot contain calls to atomic functions. For example,

  ```
  out1 = gpucoder.stencilKernel(@myAtomicSub,A,[3 3],'same',B);
  ```

Version History

Introduced in R2021b

See Also

Functions
`gpucoder.atomicAdd` | `gpucoder.atomicCAS` | `gpucoder.atomicDec` | `gpucoder.atomicExch` | `gpucoder.atomicInc` | `gpucoder.atomicMax` | `gpucoder.atomicMin`

Topics
"GPU Programming Paradigm"
"Code Generation by Using the GPU Coder App"
"Code Generation Using the Command Line Interface"
"Kernels from Element-Wise Loops"

gpucoder.atomicXor

Package: `gpucoder`

Atomically perform bit-wise XOR between a specified value and a variable in global or shared memory

Syntax

```
[A,oldA] = gpucoder.atomicXor(A,B)
```

Description

`[A,oldA] = gpucoder.atomicXor(A,B)` perform bit-wise XOR between `B` and the value of `A` in global or shared memory and writes the results back into A. The operation is atomic in a sense that the entire read-modify-write operation is guaranteed to be performed without interference from other threads. The order of the input and output arguments must match the syntax provided.

Examples

Bit-Wise XOR Using CUDA atomicXor

Perform a simple atomic addition operation by using the `gpucoder.atomicXor` function and generate CUDA code that calls appropriate CUDA `atomicXor()` APIs.

In one file, write an entry-point function `myAtomicXor` that accepts matrix inputs `a` and `b`.

```
function a = myAtomicXor(a,b)

coder.gpu.kernelfun;
for i =1:numel(a)
    [a(i),~] = gpucoder.atomicXor(a(i), b);
end

end
```

To create a type for a matrix of doubles for use in code generation, use the `coder.newtype` function.

```
A = coder.newtype('uint32', [1 30], [0 1]);
B = coder.newtype('uint32', [1 1], [0 0]);
inputArgs = {A,B};
```

To generate a CUDA library, use the `codegen` function.

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

codegen -config cfg -args inputArgs myAtomicXor -d myAtomicXor
```

The generated CUDA code contains the `myAtomicXor_kernel1` kernel with calls to the `atomicXor()` CUDA APIs.

```
//
// File: myAtomicXor.cu
//
...
```

```
static __global__ __launch_bounds__(1024, 1) void myAtomicXor_kernel1(
    const uint32_T b, const int32_T i, uint32_T a_data[])
{
  uint64_T loopEnd;
  uint64_T threadId;
...

  for (uint64_T idx{threadId}; idx <= loopEnd; idx += threadStride) {
    int32_T b_i;
    b_i = static_cast<int32_T>(idx);
    atomicXor(&a_data[b_i], b);
  }
}
...

void myAtomicXor(uint32_T a_data[], int32_T a_size[2], uint32_T b)
{
  dim3 block;
  dim3 grid;
...

    cudaMemcpy(gpu_a_data, a_data, a_size[1] * sizeof(uint32_T),
               cudaMemcpyHostToDevice);
    myAtomicXor_kernel1<<<grid, block>>>(b, i, gpu_a_data);
    cudaMemcpy(a_data, gpu_a_data, a_size[1] * sizeof(uint32_T),
               cudaMemcpyDeviceToHost);
...

}
```

Input Arguments

A, B — Operands

scalars | vectors | matrices | multidimensional arrays

Operands, specified as scalars, vectors, matrices, or multidimensional arrays. Inputs A and B must satisfy the following requirements:

- Have the same data type.
- Have the same size or have sizes that are compatible. For example, A is an M-by-N matrix and B is a scalar or 1-by-N row vector.

Data Types: int32 | uint32 | uint64

Limitations

- Function handle input to the gpucoder.stencilKernel pragma cannot contain calls to atomic functions. For example,

  ```
  out1 = gpucoder.stencilKernel(@myAtomicXor,A,[3 3],'same',B);
  ```

Version History

Introduced in R2021b

See Also

Functions

gpucoder.atomicAnd | gpucoder.atomicOr

Topics

"GPU Programming Paradigm"

“Code Generation by Using the GPU Coder App”
“Code Generation Using the Command Line Interface”
“Kernels from Element-Wise Loops”

coder.checkGpuInstall

Verify GPU code generation environment

Syntax

```
results = coder.checkGpuInstall(cfg)
```

Description

`results = coder.checkGpuInstall(cfg)` performs checks to verify if your environment has the all third-party tools and libraries required for GPU code generation. `cfg` must be an `coder.gpuEnvConfig` object. 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. To open this application, use the MATLAB command, **coder.checkGpuInstallApp**.

Before using this function, install and set up the required prerequisite third-party compilers, libraries, and tools. For more information, see “Installing Prerequisite Products” and “Setting Up the Prerequisite Products”.

Examples

Check GPU Code Generation Environment

Perform a complete check of all third-party tools required for GPU code generation. The output shown here is representative. Your results might differ.

```
gpuEnvObj = coder.gpuEnvConfig;
gpuEnvObj.GpuId = 1;
gpuEnvObj.BasicCodegen = 1;
gpuEnvObj.BasicCodeexec = 1;
results = coder.checkGpuInstall(gpuEnvObj)
```

```
Compatible GPU           : PASSED
CUDA Environment         : PASSED
    Runtime   : PASSED
    cuFFT     : PASSED
    cuSOLVER  : PASSED
    cuBLAS    : PASSED
Basic Code Generation    : PASSED
Basic Code Execution     : PASSED

results =

  struct with fields:

                 gpu: 1
                cuda: 1
               cudnn: 0
```

```
            tensorrt: 0
         basiccodegen: 1
        basiccodeexec: 1
          deepcodegen: 0
         deepcodeexec: 0
    tensorrtdatatype: 0
           profiling: 0
```

Input Arguments

`cfg` — `coder.gpuEnvConfig` configuration object
`object`

`coder.gpuEnvConfig` object contains the configuration parameters that `coder.checkGpuInstall` uses to verify the GPU code generation environment.

Output Arguments

`results` — GPU environment checking results
structure array

Results of checking the GPU code generation environment, returned as a 1-by-1 structure.

Version History

Introduced in R2017b

See Also

Apps
GPU Coder | GPU Environment Check

Functions
`getenv` | `setenv` | `gpuDevice` | `gpuDeviceCount` | `codegen` | `coder.gpu.kernel` | `coder.gpu.kernelfun`

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

Topics
"GPU Programming Paradigm"
"Installing Prerequisite Products"
"Setting Up the Prerequisite Products"
"The GPU Environment Check and Setup App"
"Verify Setup"
"Code Generation by Using the GPU Coder App"
"Code Generation Using the Command Line Interface"
"Code Generation for Deep Learning Networks by Using cuDNN"
"Code Generation for Deep Learning Networks by Using TensorRT"

coder.gpuConfig

Configuration parameters for CUDA code generation from MATLAB code by using GPU Coder

Description

The `coder.GpuCodeConfig` or `coder.gpuConfig` object contains the configuration parameters that `codegen` uses for generating CUDA MEX, a static library, a dynamically linked library, or an executable program with GPU Coder. Pass the object to the `codegen` function by using the `-config` option.

Creation

Syntax

```
cfg = coder.gpuConfig(build_type)
cfg = coder.gpuConfig(build_type,'ecoder',false)
cfg = coder.gpuConfig(build_type,'ecoder',true)
```

Description

`cfg = coder.gpuConfig(build_type)` creates a code generation configuration object for the specified build type, which can be CUDA MEX, a static library, a dynamically linked library, or an executable program. If the Embedded Coder product is installed, it creates a `coder.EmbeddedCodeConfig` object for static library, dynamic library, or executable build types.

`cfg = coder.gpuConfig(build_type,'ecoder',false)` creates a code generation configuration object to generate CUDA `'lib'`, `'dll'`, or `'exe'` output even if the Embedded Coder product is installed.

`cfg = coder.gpuConfig(build_type,'ecoder',true)` creates a `coder.EmbeddedCodeConfig` configuration object even if the Embedded Coder product is not installed. However, code generation using a `coder.EmbeddedCodeConfig` object requires an Embedded Coder license.

Input Arguments

build_type — Output to build from generated CUDA code
`'MEX'` | `'LIB'` | `'DLL'` | `'EXE'`

Output to build from generated CUDA code, specified as one of the values in this table.

Value	Description
`'MEX'`	CUDA MEX
`'LIB'`	Static library
`'DLL'`	Dynamically linked library

Value	Description
'EXE'	Executable program

Properties

`coder.GpuConfig` contains only GPU specific configuration parameters of the code configuration object. To see all the properties of the code configuration object, see `coder.CodeConfig` and `coder.EmbeddedCodeConfig`.

`Enabled` — Control GPU code generation
`true` (default) | `false`

Control generation of CUDA (*.cu) files by using one of the values in this table.

Value	Description
`true`	This value is the default value. Enables CUDA code generation.
`false`	Disables CUDA code generation.

Example: `cfg.GpuConfig.Enabled = true`

`MallocMode` — GPU memory allocation
`'discrete'` (default) | `'unified'`

Memory allocation (`malloc`) mode to be used in the generated CUDA code, specified as one of the values in this table.

Value	Description
`'discrete'`	This value is the default value. The generated code uses the `cudaMalloc` API for transferring data between the CPU and the GPU. From the programmers point-of-view, the discrete mode has a traditional memory architecture with separate CPU and GPU global memory address space.
`'unified'`	The generated code uses the `cudaMallocManaged` API that uses a shared (unified) CPU and GPU global memory address space. For NVIDIA embedded targets only. See “unified memory allocation mode on host being removed” on page 2-48.

For more information, see “Discrete and Managed Modes”.

Example: `cfg.GpuConfig.MallocMode = 'discrete'`

`KernelNamePrefix` — Custom kernel name prefixes
' ' (default) | character vector

Specify a custom name prefix for all the kernels in the generated code. For example, using the value `'CUDA_'` creates kernels with names `CUDA_kernel1`, `CUDA_kernel2`, and so on. If no name is provided, GPU Coder prepends the kernel name with the name of the entry-point function. Kernel names can contain upper-case letters, lowercase letters, digits 0–9, and underscore character _. GPU Coder removes unsupported characters from the kernel names and appends `alpha` to prefixes that do not begin with an alphabetic letter.

Example: `cfg.GpuConfig.KernelNamePrefix = 'myKernel'`

EnableCUBLAS — Use cuBLAS library
`true` (default) | `false`

Replacement of math function calls with NVIDIA `cuBLAS` library calls, specified as one of the values in this table.

Value	Description
`true`	This value is the default value. Allows GPU Coder to replace appropriate math function calls with calls to the `cuBLAS` library. For functions that have no replacements in CUDA, GPU Coder uses portable MATLAB functions and attempts to map them to the GPU.
`false`	Disable the use of the `cuBLAS` library in the generated code.

For more information, see “Kernels from Library Calls”.

Example: `cfg.GpuConfig.EnableCUBLAS = true`

EnableCUSOLVER — Use cuSOLVER library
`true` (default) | `false`

Replacement of math function calls with NVIDIA `cuSOLVER` library calls, specified as one of the values in this table.

Value	Description
`true`	This value is the default value. Allows GPU Coder to replace appropriate math function calls with calls to the `cuSOLVER` library. For functions that have no replacements in CUDA, GPU Coder uses portable MATLAB functions and attempts to map them to the GPU.
`false`	Disable the use of the `cuSOLVER` library in the generated code.

For more information, see “Kernels from Library Calls”.

Example: `cfg.GpuConfig.EnableCUSOLVER = true`

EnableCUFFT — Use cuFFT library
`true` (default) | `false`

Replacement of `fft` function calls with NVIDIA `cuFFT` library calls, specified as one of the values in this table.

Value	Description
`true`	This value is the default value. Allows GPU Coder to replace appropriate `fft` calls with calls to the `cuFFT` library.
`false`	Disables use of the `cuFFT` library in the generated code. With this option, GPU Coder uses C `FFTW` libraries where available or generates kernels from portable MATLAB `fft` code.

For more information, see “Kernels from Library Calls”.

Example: `cfg.GpuConfig.EnableCUFFT = true`

`Benchmarking` — Add benchmarking to the generated code
`false` (default) | `true`

Control addition of benchmarking code to the generated CUDA code by using one of the values in this table.

Value	Description
`false`	This value is the default value. The generated CUDA code does not contain benchmarking functionality.
`true`	Generates CUDA code with benchmarking functionality. This option uses CUDA APIs such as `cudaEvent` to accurately time `kernel`, `memcpy`, and other events.

Example: `cfg.GpuConfig.Benchmarking = true`

`SafeBuild` — Error checking in the generated code
`false` (default) | `true`

Add error-checking functionality to the generated CUDA code by using one of the values in this table.

Value	Description
`false`	This value is the default value. The generated CUDA code does not contain error-checking functionality.
`true`	Generates code with error-checking for CUDA API and kernel calls.

Example: `cfg.GpuConfig.SafeBuild = true`

`ComputeCapability` — Minimum compute capability for code generation
`'3.5'` (default) | `'3.2'` | `'3.7'` | `'5.0'` | `'5.2'` | `'5.3'` | `'6.0'` | `'6.1'` | `'6.2'` | `'7.0'` | `'7.1'` | `'7.2'`

Select the minimum compute capability for code generation. The compute capability identifies the features supported by the GPU hardware. It is used by applications at run time to determine which hardware features, instructions are available on the present GPU. If you specify custom compute capability, GPU Coder ignores this setting.

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

Target	Compute Capability
CUDA MEX	See “GPU Computing Requirements”.
Source code, static or dynamic library, and executables	3.2 or higher.
Deep learning applications in 8-bit integer precision	6.1, 6.3 or higher.
Deep learning applications in half-precision (16-bit floating point)	5.3, 6.0, 6.2 or higher.

Example: `cfg.GpuConfig.ComputeCapability = '6.1'`

`CustomComputeCapability` — Control GPU code generation
`''` (default) | character vector

Specify the name of the NVIDIA virtual GPU architecture for which the CUDA input files must be compiled.

For example, to specify a virtual architecture type `-arch=compute_50`. You can specify a real architecture using `-arch=sm_50`. For more information, see the *Options for Steering GPU Code Generation* topic in the CUDA Toolkit documentation.

Example: `cfg.GpuConfig.CustomComputeCapability = '-arch=compute_50'`

`CompilerFlags` — Additional flags to the GPU compiler
`''` (default) | `character vector`

Pass additional flags to the GPU compiler. For example, `--fmad=false` instructs the `nvcc` compiler to disable contraction of floating-point multiply and add to a single Floating-Point Multiply-Add (FMAD) instruction.

For similar NVIDIA compiler options, see the topic on *NVCC Command Options* in the CUDA Toolkit documentation.

Example: `cfg.GpuConfig.CompilerFlags = '--fmad=false'`

`StackLimitPerThread` — Stack limit per GPU thread
`1024` (default) | `integer`

Specify the maximum stack limit per GPU thread as an integer value.

Example: `cfg.GpuConfig.StackLimitPerThread = 1024`

`MallocThreshold` — Malloc threshold
`200` (default) | `integer`

Specify the size above which the private variables are allocated on the heap instead of the stack, as an integer value.

Example: `cfg.GpuConfig.MallocThreshold = 256`

`MaximumBlocksPerKernel` — Maximum number of blocks created during a kernel launch
`0` (default) | `integer`

Specify the maximum number of blocks created during a kernel launch.

Because GPU devices have limited streaming multiprocessor (SM) resources, limiting the number of blocks for each kernel can avoid performance losses from scheduling, loading and unloading of blocks.

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

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

Example: `cfg.GpuConfig.MaximumBlocksPerKernel = 1024`

`EnableMemoryManager` — Use GPU memory manager
`false` (default) | `true`

Select the GPU memory manager for efficient memory allocation, management, and improving run-time performance.

Value	Description
`true`	The GPU memory manager creates a collection of large GPU memory pools and manages allocation and deallocation of chunks of memory blocks within these pools. By creating large memory pools, the memory manager reduces the number of calls to the CUDA memory APIs, improving run-time performance. You can use the GPU memory manager for MEX and standalone CUDA code generation.
`false`	Disable the use GPU memory manager for memory allocation and management. This value is the default value.

Example: `cfg.GpuConfig.EnableMemoryManager = true`

`BlockAlignment` — Specify alignment of memory blocks in the pool
`256` (default) | `positive integer`

Specifies the alignment of memory blocks used by the GPU memory manager. The block sizes (bytes) in the pool are a multiple of the specified value. The value for `BlockAlignment` must be power of 2.

Example: `cfg.GpuConfig.BlockAlignment = 1024`

FreeMode — Specify when GPU memory manager frees memory
`'Never'` (default) | `'AtTerminate'` | `'AfterAllocate'`

Specify when the memory manager frees the GPU device memory by using one of the values in this table.

Value	Description
`'Never'`	Free memory when the memory manager is destroyed. This value is the default value.
`'AtTerminate'`	Free empty GPU pools when the `terminate` function is called in the generated code. For MEX targets, memory is freed after every call to the generated MEX function. For other targets, memory is freed when calling the terminate function.
`'AfterAllocate'`	Empty pools are freed after each call to CUDA memory allocate.

Example: `cfg.GpuConfig.FreeMode = 'AtTerminate'`

MinPoolSize — Specify the minimum pool size of the GPU memory manager
8 (default) | `positive integer`

Specify the minimum pool size in megabytes (MB) of the GPU memory manager. The value for `MinPoolSize` must be power of 2.

The memory manager computes the size levels using the `MinPoolSize` and `MaxPoolSize` parameters by interpolating between the two values in increasing powers of 2. For example, if the `MinPoolSize` is 4 and the `MaxPoolSize` is 1024, the size levels are {4, 8, 16, 32, 64, 128, 256, 512, 1024}.

Example: `cfg.GpuConfig.MinPoolSize = 32`

MaxPoolSize — Specify the maximum pool size of the GPU memory manager
2048 (default) | `positive integer`

Specify the maximum pool size in megabytes (MB) of the GPU memory manager. The value for `MaxPoolSize` must be power of 2.

The memory manager computes the size levels using the `MinPoolSize` and `MaxPoolSize` parameters by interpolating between the two values in increasing powers of 2. For example, if the `MinPoolSize` is 4 and the `MaxPoolSize` is 1024, the size levels are {4, 8, 16, 32, 64, 128, 256, 512, 1024}.

Example: `cfg.GpuConfig.MaxPoolSize = 4096`

SelectCudaDevice — CUDA device selection
`-1` (default) | `deviceID`

In a multi GPU environment such as NVIDIA Drive platforms, specify the CUDA device to target.

Example: `cfg.GpuConfig.SelectCudaDevice = <DeviceID>`

Note `SelectCudaDevice` can be used with `gpuArray` only if `gpuDevice` and `SelectCudaDevice` point to the same GPU. If `gpuDevice` points to a different GPU, a `CUDA_ERROR_INVALID_VALUE` runtime error is thrown.

Examples

Generate CUDA MEX

Generate CUDA MEX function from a MATLAB function that is suitable for GPU code generation. Also, enable a code generation report.

Write a MATLAB function `VecAdd`, that performs vector addition of inputs `A` and `B`.

```
function [C] = VecAdd(A,B) %#codegen
    C = coder.nullcopy(zeros(size(A)));
    coder.gpu.kernelfun();
    C = A + B;
end
```

To generate a MEX function, create a code generation configuration object.

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

Enable the code generation report.

```
cfg.GpuConfig.EnableCUBLAS = true;
cfg.GenerateReport = true;
```

Generate a MEX function in the current folder specifying the configuration object using the `-config` option.

```
% Generate a MEX function and code generation report
codegen -config cfg -args {zeros(512,512,'double'),zeros(512,512,'double')} VecAdd
```

Limitations

- GPU Coder always sets the `PassStructByReference` property of the `coder.CodeConfig` and `coder.EmbeddedCodeConfig` code configuration object to true.
- GPU Coder always sets the `EnableSignedLeftShifts` and the `EnableSignedRightShifts` property of the `coder.EmbeddedCodeConfig` code configuration object to true.
- For standalone targets such as static library, dynamically linked library, or executable program in the Windows® environment, the generated makefiles does not set `/MT` or `/MD` compiler flags. These flags indicate to the Visual Studio compiler to use the multithread library. By default, Visual Studio uses the `/MT` during compilation. To pass other compiler specific flags, use the `CompilerFlags` option. For example,

  ```
  cfg.GpuConfig.CompilerFlags = '-Xcompiler /MD';
  ```
- The `nvcc` compiler has limitations on input file suffixes. For example, if object file contains version numbers, compilation may fail. In such cases create symbolic links or pass `'-Xlinker'` to the `CompilerFlags`.

Version History

Introduced in R2017b

`unified` memory allocation mode on host being removed

Behavior change in future release

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

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

See Also

Apps

GPU Coder

Functions

`codegen` | `coder.gpu.kernel` | `gpucoder.stencilKernel` | `gpucoder.matrixMatrixKernel` | `coder.gpu.constantMemory` | `gpucoder.reduce` | `gpucoder.sort` | `coder.gpu.nokernel`

Objects

`coder.gpuConfig` | `coder.CodeConfig` | `coder.EmbeddedCodeConfig` | `coder.MexCodeConfig`

Topics

"GPU Programming Paradigm"
"Code Generation by Using the GPU Coder App"
"Code Generation Using the Command Line Interface"
"Kernels from Element-Wise Loops"
"Kernels from Scatter-Gather Type Operations"
"Kernels from Library Calls"
"Design Patterns"

gpucoder

Open GPU Coder app

Syntax

```
gpucoder
gpucoder projectname
gpucoder -open projectname
gpucoder -new projectname
gpucoder -ecoder false -new projectname
gpucoder -build projectname
gpucoder -tocode projectname -script scriptname
gpucoder -tocode projectname
```

Description

`gpucoder` opens the GPU Coder app. To create a project, provide the entry-point file name on the **Select Source Files** page. The app creates a project with the name of the first entry-point file as the default name. To open an existing project, click , and select `Open existing project`.

If the Embedded Coder product is installed, the app enables Embedded Coder features when it creates a project. To disable Embedded Coder features, in the project build settings, on the **All Settings** tab, under **Advanced**, set **Use Embedded Coder features** to `No`.

`gpucoder projectname` or `gpucoder -open projectname` opens the existing project named `projectname.prj` by using the GPU Coder app.

`gpucoder -new projectname` creates a GPU Coder project named `projectname.prj` and opens the GPU Coder app. If the Embedded Coder product is installed, the app enables Embedded Coder features when it creates a project.

`gpucoder -ecoder false -new projectname` opens the GPU Coder app creating a project named `projectname.prj`. The app creates a project with the Embedded Coder features disabled, even if the Embedded Coder product is installed.

`gpucoder -build projectname` builds the existing project named `projectname.prj`.

`gpucoder -tocode projectname -script scriptname` creates a script named `scriptname.m` containing the equivalent MATLAB commands for the project settings in `projectname.prj`.

If `scriptname.m` exists, `gpucoder` overwrites it. The script:

- Creates a configuration object named `cfg` that contains project build configuration.
- Defines the variable `ARGS` for the function input types.
- Defines the variable `GLOBALS` for global data initial values.
- Runs the `codegen` command. When you run the script, the entry-point functions that are arguments to `codegen` must be on the search path.

`cfg`, `ARGS`, and `GLOBALS` appear in the base workspace only after you run the script.

`gpucoder -tocode projectname` converts the existing project named `projectname.prj` to the equivalent list of MATLAB commands and writes them to the Command Window.

Examples

Convert a GPU Coder Project to a MATLAB Script

Convert the GPU Coder project named `myGPU_project.prj` to the MATLAB script named `myGPU_script.m`.

```
coder -tocode myGPU_project -script myGPU_script.m
```

Input Arguments

`projectname` — Name of the GPU Coder project
character vector

Name of GPU Coder project that you want to create, open, or build. The project name must not contain spaces.

`scriptname` — Name of script file
character vector

Name of script that you want to create when using the `-tocode` option with the `-script` option. The script name must not contain spaces.

Version History

Introduced in R2017b

See Also

Apps
GPU Coder | GPU Environment Check

Functions
`codegen` | `coder.checkGpuInstall`

Objects
`coder.gpuEnvConfig`

Topics
"GPU Programming Paradigm"
"Installing Prerequisite Products"
"Setting Up the Prerequisite Products"
"The GPU Environment Check and Setup App"
"Code Generation by Using the GPU Coder App"
"Code Generation for Deep Learning Networks by Using cuDNN"
"Code Generation for Deep Learning Networks by Using TensorRT"

coder.gpu.kernel

Pragma that maps `for`-loops to GPU kernels

Syntax

```
coder.gpu.kernel()
coder.gpu.kernel(B,T)
coder.gpu.kernel(B,T,M,name)
```

Description

`coder.gpu.kernel()` is a loop-level pragma that you must place immediately before a for loop. It generates a kernel with the dimensions computed from the loop parameters.

Note The `coder.gpu.kernel` pragma overrides all parallel loop analysis checks that the software performs. Use `coder.gpu.kernelfun` first before using the more advanced functionality of the `coder.gpu.kernel` pragma.

`coder.gpu.kernel(B,T)` is a loop-level pragma that you must place immediately before a for loop. It generates a kernel with the dimensions specified by `B` and `T`. `B[Bx,By,1]` is an array that defines the number of blocks in the grid along dimensions `x` and `y` (`z` not used). `T[Tx,Ty,Tz]` is an array that defines the number of threads in the block along dimensions `x`, `y`, and `z`.

A value of -1 for `B` and `T` indicates that GPU Coder must infer the grid and block dimensions automatically. The `coder.gpu.kernel` pragma generates errors for invalid grid and block dimensions.

`coder.gpu.kernel(B,T,M,name)` expects the same `B` and `T` arguments. You can specify optional arguments `M` and `name`. `M` is a positive integer specifying the minimum number of blocks per streaming multiprocessor. Sometimes, increasing `M` can reduce the register usage within a kernel and improve kernel occupancy. A value of -1 for `M` indicates that GPU Coder must use the default value of 1. `name` is a character array that allows you to customize the name of the generated kernel.

Specifying the kernel pragma overrides all parallel loop analysis checks. This override allows loops to be parallelized in situations where parallel loop analysis cannot prove that all iterations are independent of each other. First, ensure that the loop is safe to parallelize.

This function is a code generation function. It has no effect in MATLAB.

Examples

Generate CUDA Code for MATLAB Function

This example shows how to use the `kernel` pragma in a function and generate CUDA code.

In one file, write the entry-point function `scalars` that accepts two vector inputs `x,y` of size `1x4096` and one scalar input `scale`. The function has two `for`-loops of different iteration lengths, one for

vector addition and one for finding the cumulative sum. Place the `coder.gpu.kernel(1,1024)` pragma outside the first loop. This pragma creates a kernel with one block having 1024 threads. Place the `coder.gpu.kernel(8,512,512,'reduction')` pragma outside the second loop. This pragma creates a kernel with eight blocks having 512 threads per block. The kernel created for this block is named reduction.

```
function [vout, sout1] = scalars(x,y,scale)
    sout1 = 0;
    vout = coder.nullcopy(zeros(size(x)));

    coder.gpu.kernel(1,1024);
    for i=1:1024
        vout(i) = x(i) + y(i);
    end

    coder.gpu.kernel(8,512,512,'reduction');
    for i=1:4096
        sout1 = (x(i)*scale) + sout1;
    end
end
```

Use the `codegen` function to generate CUDA MEX function.

```
codegen -config coder.gpuConfig('mex')...
 -args {ones(1,4096,'double'),ones(1,4096,'double'),coder.typeof(0)}...
 -report scalars
```

GPU Coder creates two kernels: `scalars_kernel1` for vector addition and `scalarsreduction` kernel for the cumulative sum. No kernel is needed for initializing `sout1=0`.

```
  cudaMemcpy(gpu_y, y, 32768U, cudaMemcpyHostToDevice);
  cudaMemcpy(gpu_x, x, 32768U, cudaMemcpyHostToDevice);
  scalars_kernel1<<<dim3(1U, 1U, 1U), dim3(1024U, 1U, 1U)>>>(gpu_y, gpu_x, gpu_vout);
  cudaMemcpy(gpu_sout1, sout1, 8U, cudaMemcpyHostToDevice);
  scalarsreduction<<<dim3(8U, 1U, 1U), dim3(512U, 1U, 1U>>>(scale, gpu_x, gpu_sout1);
  cudaMemcpy(vout, gpu_vout, 32768U, cudaMemcpyDeviceToHost);
  cudaMemcpy(sout1, gpu_sout1, 8U, cudaMemcpyDeviceToHost);
```

`scalars_kernel1` has one block with 1024 threads per block, one for adding each element. `scalarsreduction` kernel has eight blocks with 512 threads per block, resulting in a total of 4096 threads.

You can use variables or expressions when specifying the kernel dimensions. For example, you can rewrite the `scalars` entry-point function such that the grid and block dimensions are specified at compile time.

```
function [vout, sout1] = scalars(x,y,scale, a, b)
    sout1 = 0;
    vout = zeros(size(x));

    coder.gpu.kernel(1,1024);
    for i=1:1024
        vout(i) = x(i) + y(i);
    end

    coder.gpu.kernel([a,a*b,1], [a*b, 1, 1], 'reduction');
    for i=1:length(x)
        sout1 = (x(i)*scale) + sout1;
    end
end
```

Use the `codegen` function to generate CUDA MEX function.

```
codegen -config coder.gpuConfig('mex')...
 -args {ones(1,4096,'double'),ones(1,4096,'double'),20,8,4}...
 -report scalars
```

Version History

Introduced in R2017b

See Also

Apps

GPU Coder

Functions

`codegen` | `coder.gpu.kernelfun` | `gpucoder.stencilKernel` | `coder.gpu.constantMemory` | `gpucoder.reduce` | `gpucoder.sort` | `coder.gpu.nokernel`

Objects

`coder.gpuConfig` | `coder.CodeConfig` | `coder.EmbeddedCodeConfig` | `coder.MexCodeConfig`

Topics

"GPU Programming Paradigm"
"Code Generation by Using the GPU Coder App"
"Code Generation Using the Command Line Interface"
"Kernels from Element-Wise Loops"
"Kernels from Scatter-Gather Type Operations"
"Kernels from Library Calls"
"Design Patterns"

coder.gpu.kernelfun

Pragma that maps function to GPU kernels

Syntax

```
coder.gpu.kernelfun()
```

Description

`coder.gpu.kernelfun()` is a global-level pragma that attempts to map all the computation within the function it resides in on to the GPU. Loops within this function are parallelized into GPU kernels only if they pass the parallel-loop analysis check. This analysis tries to prove that every loop iteration is independent of each other.

This pragma does not require any input parameters. It generates kernels whose dimensions are computed automatically based on loop parameters.

This function is a code generation function. It has no effect in MATLAB.

Examples

Generate CUDA Code for MATLAB Function

This example shows how to use the `kernelfun` pragma in a function and generate CUDA code.

In one file, write the entry-point function `scalars` that accepts two vector inputs `x,y` of size `1x4096` and one scalar input `scale`. The function has two `for`-loops of different iteration lengths, one for vector addition and one for finding the cumulative sum. Place the `coder.gpu.kernelfun()` pragma within the `scalars` function.

```
function [vout, sout1] = scalars(x,y,scale)
    coder.gpu.kernelfun;
    sout1 = 0;
    vout = coder.nullcopy(zeros(1,1024));

    for i=1:1024
        vout(i) = x(i) + y(i);
    end

    for i=1:4096
        sout1 = (x(i)*scale) + sout1;
    end
end
```

Use the `codegen` function to generate CUDA MEX function.

```
codegen -config coder.gpuConfig('mex')...
 -args {ones(1,4096,'double'),ones(1,4096,'double'),coder.typeof(0)}...
 -report scalars
```

GPU Coder creates three kernels: `scalars_kernel1` for initializing `sout1=0`, `scalars_kernel2` for vector addition, and `scalars_kernel3` is the reduction kernel for the cumulative sum.

```
scalars_kernel1<<<dim3(1U, 1U, 1U), dim3(32U, 1U, 1U)>>>(gpu_sout1);
cudaMemcpy(gpu_y, y, 32768U, cudaMemcpyHostToDevice);
cudaMemcpy(gpu_x, x, 32768U, cudaMemcpyHostToDevice);
scalars_kernel2<<<dim3(2U, 1U, 1U), dim3(512U, 1U, 1U)>>>(gpu_y, gpu_x, gpu_vout);
scalars_kernel3<<<dim3(8U, 1U, 1U), dim3(512U, 1U, 1U)>>>(scale, gpu_x, gpu_sout1);
cudaMemcpy(vout, gpu_vout, 32768U, cudaMemcpyDeviceToHost);
cudaMemcpy(sout1, gpu_sout1, 8U, cudaMemcpyDeviceToHost);
```

`scalars_kernel2` has two blocks with 512 threads per block for a total of 1024 threads, one for adding each element. Similarly, `scalars_kernel3` has eight blocks with 512 threads per block resulting in a total of 4096 threads. GPU Coder also performs an optimization that minimizes the number of `cudamMemcpy` function calls. In this example, a copy of the input `x` is in the GPU, no extra `cudamMemcpy` is required between `scalars_kernel2` and `scalars_kernel3`. In addition to memory optimization, any sequential code between kernels is mapped to the CUDA threads to keep data on the GPU.

Version History

Introduced in R2017b

See Also

Apps

GPU Coder

Functions

`codegen` | `coder.gpu.kernel` | `gpucoder.stencilKernel` | `coder.gpu.constantMemory` | `gpucoder.reduce` | `gpucoder.sort` | `coder.gpu.nokernel`

Objects

`coder.gpuConfig` | `coder.CodeConfig` | `coder.EmbeddedCodeConfig` | `coder.MexCodeConfig`

Topics

“GPU Programming Paradigm”
“Code Generation by Using the GPU Coder App”
“Code Generation Using the Command Line Interface”
“Kernels from Element-Wise Loops”
“Kernels from Scatter-Gather Type Operations”
“Kernels from Library Calls”
“Design Patterns”

coder.gpu.constantMemory

Pragma that maps a variable to the constant memory on GPU

Syntax

```
coder.gpu.constantMemory(v)
```

Description

`coder.gpu.constantMemory(v)` maps the variable `v` to the constant memory space on the GPU device. Place this pragma within a parallelizable loop. If GPU Coder generates a kernel for the loop, it loads `v` to a device constant memory variable. It replaces any access to this variable within the kernel by access to the constant memory variable. Within the kernel, the variable `v` must be read-only. Otherwise, GPU Coder ignores this pragma. Use this pragma when every thread accesses every element of the parameter array or matrix.

This function is a code generation function. It has no effect in MATLAB.

Examples

Map Read-Only Input to GPU Constant Memory

This example shows how to map an input to the constant memory space on the GPU by using the `coder.gpu.constantMemory` pragma.

Write an entry-point function `myFun` that accepts two inputs `a` of size `256x256` and constant `k` of size `1x3`. The function has a nested `for`-loops that adds the constants to each element of `a`. To create a kernel, place the `coder.gpu.kernel()` pragma outside the nested `for`-loop. The `coder.gpu.constantMemory(k)` places the read-only input `k` into the constant memory of the GPU.

```
function b = myFun(a,k)
  b = coder.nullcopy(zeros(size(a)));
  coder.gpu.kernel();
    for j = 1:256
      for i = 1:256
        coder.gpu.constantMemory(k);
        b(i,j) = a(i,j) + k(1) + k(2) + k(3);
      end
    end
end
```

Create a configuration object for MEX code generation.

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

Define a cell array `input` that declares the size and data type of the inputs `a,k` to the function `myFun`.

```
input = {ones(256),ones(1,3)}
```

Generate a MEX function `myFun_mex` by using `-config`, `-args`, and `-report` options to specify configuration, provide input arguments, and generate a code generation report.

```
codegen -config cfg -args input -report myFun
```

In the report, on the **C code** tab, click `myFun.cu`.

The read-only variable `k` is declared as `const_k` by using the `__constant__` qualifier as shown in the code snippet.

```
/* Variable Definitions */
__constant__ real_T const_k[3];
```

`cudaMemcpyToSymbol` call copies the value of `k` from the host to the device constant memory `const_k`.

```
  cudaMemcpyToSymbol(const_k, k, 24U, 0U, cudaMemcpyHostToDevice);
  cudaMemcpy(gpu_a, a, 524288U, cudaMemcpyHostToDevice);
  myFun_kernel1<<<dim3(128U, 1U, 1U), dim3(512U, 1U, 1U)>>>(gpu_a, gpu_b);
  cudaMemcpy(b, gpu_b, 524288U, cudaMemcpyDeviceToHost);
```

The kernel body accesses the constant `const_k` and adds it to each element of `a`

```
static __global__ __launch_bounds__(512, 1) void myFun_kernel1(const real_T *a,
 real_T *b)
{
  int32_T i;
  int32_T j;
  int32_T threadIdX;
  threadIdX = (int32_T)(blockDim.x * blockIdx.x + threadIdx.x);
  i = threadIdX / 256;
  j = threadIdX - i * 256;
  if ((!(j >= 256)) && (!(i >= 256))) {
    b[i + (j << 8)] = ((a[i + (j << 8)] + const_k[0]) + const_k[1]) + const_k[2];
  }
}
```

Input Arguments

v — Variable name

scalar | vector | matrix | multidimensional array

The name of the variable that must be mapped to the constant memory space on the GPU device.

Version History

Introduced in R2017b

See Also

Apps

GPU Coder

Functions

`codegen` | `coder.gpu.kernel` | `coder.gpu.kernelfun` | `gpucoder.stencilKernel` | `coder.gpu.nokernel` | `coder.gpu.persistentMemory`

Objects

`coder.gpuConfig` | `coder.CodeConfig` | `coder.EmbeddedCodeConfig` | `coder.MexCodeConfig`

Topics

“GPU Programming Paradigm”
“GPU Memory Allocation and Minimization”
“Kernels from Element-Wise Loops”
“Kernels from Scatter-Gather Type Operations”
“Design Patterns”
“Kernels from Library Calls”

stencilfun

Generate CUDA code for stencil functions

Syntax

```
[OUT_1,...,OUT_N] = stencilfun(func,X,W)
___ = stencilfun(___,Name,Value)
```

Description

`[OUT_1,...,OUT_N] = stencilfun(func,X,W)` applies the function `func` to each sliding window of size `W` of the input array `X`. `stencilfun` outputs a number of equally-sized output arrays equal to the number of outputs of `func`. Each call made to `func` computes a single element of each output array. The index of this element corresponds to the center of the sliding window in the input array.

`___ = stencilfun(___,Name,Value)` performs stencil operation by using the options specified by one or more `Name,Value` pair arguments.

Examples

Perform 2-D Convolution Using Stencil Operations

This example shows how to perform 2-D convolution of an array with a 5x5 filter by using the `stencilfun` function.

Define the MATLAB entry-point function

```
function Out = myconv(In, W)
fh = @(X) stencilFcn(X, W);
Out = stencilfun(fh, In, [5 5], Shape = 'same');
end
```

The stencil function `stencilFcn` is defined as:

```
function y = stencilFcn(X, W)
y = 0;
for j = 1:5
    for i = 1:5
        y = y + X(i,j) * W(i,j);
    end
end
end
```

Use `codegen` command to generate CUDA code.

Input Arguments

`func` — Function to apply
function handle

Function to apply to the elements of the input arrays, specified as a function handle. `func` must take the following form:

```
[Y_1, ..., Y_N] = FUNC(X1, IDX_1, ..., IDX_M);
```

`X1` denotes a subarray of `X` of size `W`. It is possible that the sliding window corresponding to `X1` may span outside the boundary of `X`. In this case, `X1` will be padded with a constant value wherever necessary. Each parameter `IDX_K` denotes the index along dimension `K` of the output element currently being computed and has type `int32`. `func` can accept any number of `IDX` values, but this number must be fixed-size. Each parameter `Y_K` must be a scalar value of supported data type, which computes a single output element of the `K`th output array. The type of value `Y_K` will determine the type of array `OUT_K`. `func` is permitted to be an anonymous or nested function. In this case, `func` may reference additional variables that are in scope where `func` is defined. This behavior is useful if the stencil operation requires additional parameters in addition to the input window.

Example: `out = stencilfun(fh, X, [5 5]);`

Data Types: `function_handle`

X — Input array
array

X must be an array of supported data type.

Example: `out = stencilfun(fh, In, [5 5]);`

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical`

W — Window size
positive integers

`W` must be a numeric row vector where `W(D)` specifies the window size along dimension `D`. The dimensionality of the input array and window can be arbitrary and do not need to be equal.

Example: `out = stencilfun(fh, In, [5 5], Shape = 'same');`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `out = stencilfun(fh, In, [5 5], Shape = 'same');`

Shape — Size of each output array
`'full'` (default) | `'same'` | `'valid'`

Specifies the size of each output array by determining the amount of padding to be applied on each dimension of the input. `'Shape'` must be specified as one of these values:

- `'full'` — Maximum padding will applied to the input array such that each sliding window will access at least one element of the unpadded input array.
- `'same'` — The input array will be padded so that an output array and the input array will have the same size (assuming no striding is used).

- `'valid'` — The input array will not be padded

Example: `out = stencilfun(fh, In, [5 5], Shape = 'same');`

`Preprocess` — Preprocessing function to apply
function handle

Specify the preprocess function to apply to all elements of the input array (including padding elements) before performing the stencil computation. `'Preprocess'` must be a function handle that takes the following form:

```
Y = preprocess(X);
```

where `X` denotes an input element and `Y` denotes the preprocessed result. By default, no preprocessing is applied.

Example: `out = stencilfun(fh, In, [5 5], Preprocess = @(x) single(3.14*x));`

`Stride` — Step size to traverse input array
numeric scalar (default) | row vector

Specify the step size for traversing the input array when sliding the window in each dimension. `'Stride'` must be a numeric scalar or a row vector. If `'Stride'` is a scalar, then that value specifies the stride for each dimension. If `'Stride'` is a vector, then `Stride(D)` specifies the stride along dimension `D`. The default value of `'Stride'` is 1.

Example: `out = stencilfun(fh, In, [5 5], Stride = 2);`

`PaddingValue` — Pad input array
positive integer

Specifies the value to use for padding the input array. `'PaddingValue'` must have the same class as the input array. The default value of `'PaddingValue'` is 0.

Example: `out = stencilfun(fh, In, [5 5], PaddingValue = 1);`

Output Arguments

`OUT` — Output array
array

The number of outputs equal the number of outputs of `func`. All the output arrays are of the same size.

Version History

Introduced in R2022b

See Also

Apps
GPU Coder

Functions
`codegen` | `coder.gpu.kernel` | `gpucoder.matrixMatrixKernel` | `coder.gpu.constantMemory` | `gpucoder.reduce` | `gpucoder.sort` | `coder.gpu.nokernel`

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

Topics
"GPU Programming Paradigm"
"Code Generation by Using the GPU Coder App"
"Code Generation Using the Command Line Interface"
"Kernels from Element-Wise Loops"
"Kernels from Scatter-Gather Type Operations"
"Kernels from Library Calls"
"Design Patterns"

gpucoder.stencilKernel

(Not recommended) Create CUDA code for stencil functions

Note `gpucoder.stencilKernel` is not recommended. Use `stencilfun` instead. For more information, see “Compatibility Considerations”.

Syntax

```
B = gpucoder.stencilKernel(FUN,A,[M N],shape,param1,param2...)
```

Description

`B = gpucoder.stencilKernel(FUN,A,[M N],shape,param1,param2...)` applies the function `FUN` to each `[M,N]` sliding window of the input `A`. Function `FUN` is called for each `[M,N]` submatrix of `A` and computes an element of output `B`. The index of this element corresponds to the center of the `[M,N]` window.

`FUN` is the handle to a user-defined function that returns a scalar output of the same type as the input.

```
C= FUN(X,param1,param2, ...)
```

`X` is the `[M,N]` submatrix of the original input `A`. `X` can be zero-padded when necessary, for instance at the boundaries of input `A`. `X` and the window can also be 1-D.

`C` is a scalar valued output of `FUN`. It is the output computed for the center element of the `[M,N]` array `X` and is assigned to the corresponding element of the output array `B`.

`param1,param2` are optional arguments. Pass these arguments if `FUN` requires any additional parameters in addition to the input window.

The window `[M,N]` must be less than or equal to the size of `A`, with the same shape as `A`.

If `A` is 1-D row vector, the window must be `[1,N]`.

If `A` is 1-D column vector, the window must be `[N,1]`.

`shape` determines the size of the output array `B`. It can have one of three possible values:

- `'same'` - Returns output `B` that is the same size as `A`.
- `'full'` - (default) Returns the full output. Size of `B` > size of `A`, that is, if `A` is of size (x,y). Size of `B = [x + floor(M/2), y + floor(N/2)]`
- `'valid'` - Returns only those parts of the output that are computed without the zero-padded edges of `A`. Size of `B = [x - floor(M/2), y - floor(N/2)]`

The input `A` must be a vector or matrix with a numeric type supported by `FUN`. The class of `B` is the same as the class of `A`.

Code generation is supported only for fixed size outputs. Shape and window must be compile-time constants because they determine the size of the output.

Examples

Mean Filter Using Stencil Kernel

This example shows how to use the `gpucoder.stencilKernel` and generate CUDA kernels that perform filtering of an image by using stencil operations.

This example performs mean filtering of a 2-D image. In one file, write the entry-point function `test` that accepts an image matrix `A`. Create a subfunction `my_mean` that computes the mean of the `3x3` submatrix.

```
function B = meanImgFilt(A)  %#codegen
  B = gpucoder.stencilKernel(@my_mean,A,[3 3],'same');

  function out = my_mean(A)
    out = cast(mean(A(:)), class(A));
  end
end
```

Set up the test input image for the `meanImgFilt` function.

```
inImage = im2double(imread('cameraman.tif'));
```

Use the `codegen` function to generate CUDA MEX function.

```
codegen -config coder.gpuConfig('mex') -args {inImage} -report meanImgFilt
```

GPU Coder creates three kernels: `meanImgFilt_kernel1` for initializing memory, `meanImgFilt_kernel2` for optimizing the input memory structure, and `meanImgFilt_kernel3` for mean filtering operation. The following is a snippet of the generated code.

```
cudaMalloc(&gpu_B, 524288ULL);
cudaMalloc(&gpu_A, 524288ULL);
cudaMalloc(&gpu_expanded, 532512ULL);
meanImgFilt_kernel1<<<dim3(131U, 1U, 1U), dim3(512U, 1U, 1U)>>>(gpu_expanded);
cudaMemcpy((void *)gpu_A, (void *)&A[0], 524288ULL, cudaMemcpyHostToDevice);
meanImgFilt_kernel2<<<dim3(128U, 1U, 1U), dim3(512U, 1U, 1U)>>>(gpu_A,
  gpu_expanded);
meanImgFilt_kernel3<<<dim3(8U, 8U, 1U), dim3(32U, 32U, 1U)>>>(gpu_expanded,
  gpu_B);
cudaMemcpy((void *)&B[0], (void *)gpu_B, 524288ULL, cudaMemcpyDeviceToHost);
```

`meanImgFilt_kernel3` uses shared memory (`__shared__` qualifier) to improve memory bandwidth and data locality.

Limitations

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

Version History

Introduced in R2017b

`gpucoder.stencilKernel` is not recommended

Not recommended starting in R2022b

`gpucoder.stencilKernel` is not recommended. Use `stencilfun` instead.

Starting in R2022b, generate CUDA kernels for stencil like operations by using `stencilfun` function.

This table shows typical usages of `gpucoder.stencilKernel` and how to update your code to use `stencilfun` instead.

Not Recommended	Recommended

Convolution using `gpucoder.stencilKernel`:
 	
`function Out = myconv(In)`
`Out = gpucoder.stencilKernel(@stencilFcn, In, [5 5], 'same');`
`end`
 	
`function y = stencilFcn(X)`
`W = rand(5);`
`y = 0;`
`for j = 1:5`
`  for i = 1:5`
`  y = y + X(i,j) * W(i,j);`
`  end`
`end`
`end`
	Convolution using `stencilfun`:
	
	`function Out = myconv(In)`
	`fh = @(X) stencilFcn(X);`
	`Out = stencilfun(fh, In, [5 5], Shape = 'same');`
	`end`
	
	`function y = stencilFcn(X)`
	`W = rand(5)`
	`y = 0;`
	`for j = 1:5`
	`  for i = 1:5`
	`  y = y + X(i,j) * W(i,j);`
	`  end`
	`end`
	`end`

Passing extra arguments to the stencil function.
 	
`weights = rand(5);`
`In = rand(100);`
`Out = gpucoder.stencilKernel(@stencilFcn, In, [5 5], 'same', weights);`
 	
`function y = stencilFcn(X, weights)`
`  y = 0;`
`  for i = 1 : 5`
`  for j = 1 : 5`
`  y = y + X(j,i) * weights(j,i);`
`  end`
`  end`
`end`
	Use anonymous function to pass extra arguments to the stencil function.
	
	`weights = rand(5);`
	`fh = @(X) stencilFcn(X, weights);`
	`In = rand(100);`
	`Out = stencilfun(fh, In, [5 5],Shape='same');`
	
	`function y = stencilFcn(X, weights)`
	`  y = 0;`
	`  for i = 1 : 5`
	`  for j = 1 : 5`
	`  y = y + X(j,i) * weights(j,i);`
	`  end`
	`  end`
	`end`

See Also

Apps
GPU Coder

Functions
`codegen` | `coder.gpu.kernel` | `gpucoder.matrixMatrixKernel` | `coder.gpu.constantMemory` | `gpucoder.reduce` | `gpucoder.sort` | `coder.gpu.nokernel`

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

Topics

“GPU Programming Paradigm”
“Code Generation by Using the GPU Coder App”
“Code Generation Using the Command Line Interface”
“Kernels from Element-Wise Loops”
“Kernels from Scatter-Gather Type Operations”
“Kernels from Library Calls”
“Design Patterns”

gpucoder.matrixMatrixKernel

Optimized GPU implementation of functions containing matrix-matrix operations

Syntax

```
C = gpucoder.matrixMatrixKernel(FUN,A,B)
C = gpucoder.matrixMatrixKernel(FUN,A,B,orientation)
```

Description

`C = gpucoder.matrixMatrixKernel(FUN,A,B)` generates kernels from functions that contain GEMM-like operations. For example, matching feature points between two images by using:

- The sum of absolute differences (SAD) — `F() = @(a,b)abs(a-b)`
- The sum of squared differences (SSD) — `F() = @(a,b)(a-b).*(a-b)`

`FUN` is a handle to a user-defined function. It takes one row or column from matrix `A` and one row or column from matrix `B` and outputs a vector with the same type as the input. The output vector is then summed to compute a single scalar value in `C`. Numeric inputs `A` and `B` must be either of the same size or have sizes that are compatible. For example, if `A` is an `M`-by-`K` matrix, `B` is a `K`-by-`N` matrix then `C` is an `M`-by-`N` matrix.

`C = gpucoder.matrixMatrixKernel(FUN,A,B,orientation)` has the optional argument `orientation` that specifies the orientation of `A` and `B` matrices. It can take one of four possible values:

- `'nn'` - Matrices `A` and `B` are normal.
- `'nt'` - Matrix `B` is transposed.
- `'tn'` - Matrix `A` is transposed.
- `'tt'` - Both matrices `A` and `B` are transposed.

Examples

Matrix-Matrix Multiplication

This example performs a simple matrix-matrix multiplication and uses the `matrixMatrixKernel` design pattern to generate CUDA code.

In one file, write an entry-point function `matMul_nn` that accepts two matrix inputs `f1` and `f2`. Use the MATLAB function `@times` to multiply `f1` and `f2` element by element. The sign @ creates a handle to the function `times`. Insert the `gpucoder.matrixMatrixKernel()` statement. The input matrices are not transposed, therefore use the `'nn'` option.

```
function scores = matMul_nn(f1, f2)
    scores = gpucoder.matrixMatrixKernel(@times, f1, f2, 'nn');
end
```

Use the `codegen` function to generate CUDA MEX function.

```
codegen -config coder.gpuConfig('mex') ...
    -args {ones(1024,1024,'double'),ones(1024,1024,'double')} ...
    -report matMul_nn
```

The generated CUDA code contains two kernels: `matMul_nn_kernel1` for initializing the output matrix `scores` and `matMul_nn_kernel2` that performs the `times` operation. The following is a snippet of the generated code.

```
matMul_nn_kernel1<<<dim3(2048U, 1U, 1U), dim3(512U, 1U, 1U)>>>(gpu_scores);
cudaMemcpy(gpu_f2, f2, 8388608U, cudaMemcpyHostToDevice);
cudaMemcpy(gpu_f1, f1, 8388608U, cudaMemcpyHostToDevice);
matMul_nn_kernel2<<<dim3(16U, 16U, 1U), dim3(16U, 16U, 1U)>>>(gpu_f2, gpu_f1,
                gpu_scores);
cudaMemcpy(scores, gpu_scores, 8388608U, cudaMemcpyDeviceToHost);
```

`matMul_nn_kernel2` has 2-D grid of 2-D blocks. The kernel has 16x16 blocks with 256 threads per block.

Version History

Introduced in R2017b

See Also

Apps

GPU Coder

Functions

`codegen` | `coder.gpu.kernel` | `coder.gpu.kernelfun` | `gpucoder.stencilKernel` | `coder.gpu.constantMemory` | `coder.gpu.nokernel` | `gpucoder.batchedMatrixMultiply` | `gpucoder.stridedMatrixMultiply` | `gpucoder.batchedMatrixMultiplyAdd` | `gpucoder.stridedMatrixMultiplyAdd`

Objects

`coder.gpuConfig` | `coder.CodeConfig` | `coder.EmbeddedCodeConfig` | `coder.MexCodeConfig`

Topics

“GPU Programming Paradigm”
“Code Generation by Using the GPU Coder App”
“Code Generation Using the Command Line Interface”
“Kernels from Element-Wise Loops”
“Kernels from Scatter-Gather Type Operations”
“Kernels from Library Calls”
“Design Patterns”

coder.regenerateDeepLearningParameters

Regenerate files containing network learnables and states parameters

Syntax

```
networkFileNames = coder.regenerateDeepLearningParameters(net,parameterFile)
networkFileNames = coder.regenerateDeepLearningParameters(dlnet,
parameterFile)
networkFileNames = coder.regenerateDeepLearningParameters( ___ ,Name,Value)
```

Description

`networkFileNames = coder.regenerateDeepLearningParameters(net,parameterFile)` returns a cell array of the file names containing the regenerated network learnables and states parameters. `coder.regenerateDeepLearningParameters` regenerates these files based on the learnables and states of the input `SeriesNetwork` or `DAGNetwork`network object.

`networkFileNames = coder.regenerateDeepLearningParameters(dlnet, parameterFile)` returns a cell array of the file names containing the regenerated network learnables and states parameters. `coder.regenerateDeepLearningParameters` regenerates these files based on the learnables and states of the input `dlnetwork` object.

`networkFileNames = coder.regenerateDeepLearningParameters( ___ ,Name,Value)` returns a cell array of the file names containing the regenerated network learnables and states parameters by using the options specified by one or more `Name,Value` pair arguments.

Examples

Post-Code-Generation Update of Network Learnable and State Parameters

This example shows how to update learnable and state parameters of deep learning networks without regenerating code for the network.

Write an entry-point function in MATLAB that:

1. Uses the `coder.loadDeepLearningNetwork` function to load a deep learning model. For more information, see “Load Pretrained Networks for Code Generation”.
2. Calls `predict` to predict the responses.

```
function out = mLayer(in,matFile)

myNet = coder.loadDeepLearningNetwork(coder.const(matFile));

out = predict(myNet,in);
```

Create a simple network that requires input images of size 4-by-5-by-3.

```
inputSize = [4 5 3];
im = dlarray(rand(inputSize, 'single'), 'SSCB');

outSize = 6;
```

```
layers = [
    imageInputLayer(inputSize,'Name','input','Normalization','none')
    convolution2dLayer([3 3], 5, 'Name', 'conv-1')
    batchNormalizationLayer('Name', 'batchNorm')
    reluLayer('Name','relu1')
    transposedConv2dLayer([2 2], 5, 'Name', 'transconv')
    convolution2dLayer([2 2], 5, 'Name', 'conv2')
    reluLayer('Name','relu2')
    fullyConnectedLayer(outSize, 'Name', 'fc3')
    ];
```

Create an initialized `dlnetwork` object from the layer graph.

```
rng(0);
dlnet1 = dlnetwork(layers);
save('trainedNet.mat', 'dlnet1');
```

To specify code generation parameters for cuDNN, set the `DeepLearningConfig` property to a `coder.CuDNNConfig` object that you create by using `coder.DeepLearningConfig`.

```
cfg = coder.gpuConfig('mex');
cfg.TargetLang = 'C++';
cfg.DeepLearningConfig = coder.DeepLearningConfig('TargetLibrary', 'cudnn')
cfg.DeepLearningConfig.AutoTuning = true;
cfg.DeepLearningConfig.DataType = 'fp32';
```

Run the `codegen` command. The `codegen` command generates CUDA code from the `mLayers.m` MATLAB entry-point function.

```
cnnMatFile = fullfile(pwd, 'trainedNet.mat');
inputArgs = {im, coder.Constant(cnnMatFile)};

codegen -config cfg mLayers -args inputArgs -report
```

Call predict on the input image and compare the results with MATLAB.

```
out = mLayer_mex(im,cnnMatFile)
out_MATLAB = mLayer(im,cnnMatFile)
```

```
out1 =

  6(C) x 1(B) single dlarray

   -0.0064
   -0.1422
   -0.0897
    0.2223
    0.0329
    0.0365


out_MATLAB =

  6(C) x 1(B) single dlarray

   -0.0064
   -0.1422
   -0.0897
    0.2223
```

```
    0.0329
    0.0365
```

Re-initialize `dlnetwork` to update learnables to different values.

```
rng(10);
dlnet2 = dlnetwork(layers);
save('trainedNet.mat', 'dlnet2');
```

Use the `coder.regenerateDeepLearningParameters` function to regenerate the bias files based on the new learnables and states of the network.

```
codegenDir = fullfile(pwd, 'codegen/mex/mLayer');
networkFileNames = (coder.regenerateDeepLearningParameters(dlnet2, codegenDir))'
```

```
networkFileNames =

  8×1 cell array

    {'cnn_trainedNet0_0_conv-1_b.bin'   }
    {'cnn_trainedNet0_0_conv-1_w.bin'   }
    {'cnn_trainedNet0_0_conv2_b.bin'    }
    {'cnn_trainedNet0_0_conv2_w.bin'    }
    {'cnn_trainedNet0_0_fc3_b.bin'      }
    {'cnn_trainedNet0_0_fc3_w.bin'      }
    {'cnn_trainedNet0_0_transconv_b.bin'}
    {'cnn_trainedNet0_0_transconv_w.bin'}
```

Call predict on the input image and compare the results with MATLAB.

```
clear mLayer_mex;
outNew = mLayer_mex(im,cnnMatFile)
outNew_MATLAB = mLayer(im,cnnMatFile)
```

```
outNew =

  6(C) x 1(B) single dlarray

    0.1408
   -0.0080
    0.0342
   -0.0065
    0.1843
    0.0799


outNew_MATLAB =

  6(C) x 1(B) single dlarray

    0.1408
   -0.0080
    0.0342
   -0.0065
    0.1843
    0.0799
```

Input Arguments

net — Trained network

`DAGNetwork` object | `SeriesNetwork` object

Trained network used during code generation, specified as a `SeriesNetwork` or a `DAGNetwork` object. You can use a pretrained network (for example, by using the `googlenet` function) or by training your own network using `trainNetwork`.

`dlnet` — Network for custom training loops
`dlnetwork` object

Network for custom training loops used during code generation, specified as a `dlnetwork` object.

`parameterFile` — Location of generated network parameter file
character vector | string

Path to the folder containing the generated network parameter information file. During code generation, the software creates the `networkParamsInfo_*.bin` binary file that contains the network parameter information. By default, the code generator creates this file in the `codegen` folder.

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `networkFileNames = coder.regenerateDeepLearningParameters(dlnet,"codegen",NetworkName="fooDL");`

`NetworkName` — C++ class name
`''` (default) | character vector | string

Name of the C++ class for the network in the generated code, specified as a character vector or string.

`OverrideParameterFiles` — Use original code generation folder for regenerating network parameters
`false` (default) | `true`

For MEX workflows, when the generated MEX and the associated `codegen` folder is moved from one location to another, `coder.regenerateDeepLearningParameters` cannot regenerate files containing network learnables and states parameters in the new location. Set the `'OverrideParameterFiles'` parameter to true to allow the `coder.regenerateDeepLearningParameters` function to regenerate files containing network learnables and states parameters in the original `codegen` location.

This parameter has no effect for non-MEX workflows.

Output Arguments

`networkFileNames` — File names of network learnables and states parameters
cell array

File names of the regenerated network learnables and states parameters, returned as a cell array.

Limitations

Only the network learnables and states can be updated by using the `coder.regenerateDeepLearningParameters` function. For modifications that the code generator does not support, an error message is thrown. For example, using `coder.regenerateDeepLearningParameters` after changing the scale factor of a leaky ReLU layer throws the following error message as scale factor is not a network learnable.

```
Network architecture has been modified since the last code generation. Unable
to accommodate the provided network in the generated code. Regenerate code
for the provided network to reflect changes in the network. For more
information, see Limitations to Regenerating Network Parameters After Code Generation.
```

Version History

Introduced in R2021b

See Also

Functions

`codegen` | `coder.loadDeepLearningNetwork`

Objects

`SeriesNetwork` | `DAGNetwork` | `dlarray` | `dlnetwork`

Topics

"Load Pretrained Networks for Code Generation"
"Update Network Parameters After Code Generation"
"Code Generation for Deep Learning Networks by Using cuDNN"
"Code Generation for Deep Learning Networks by Using TensorRT"

cnncodegen

Generate code for a deep learning network to target the ARM Mali GPU

Syntax

```
cnncodegen(net,'targetlib','arm-compute-mali')
cnncodegen(net,'targetlib','arm-compute-mali',targetparams)
```

Description

`cnncodegen(net,'targetlib','arm-compute-mali')` generates C++ code for the specified network object by using the ARM Compute Library for Mali GPUs.

Requires the GPU Coder product and the GPU Coder Interface for Deep Learning Libraries.

`cnncodegen(net,'targetlib','arm-compute-mali',targetparams)` generates C++ code for the specified network object by using the ARM Compute Library for Mali GPUs with additional code generation options.

Examples

Generate C++ Code for a Pretrained Network to Run on an ARM Processor

Use `cnncodegen` to generate C++ code for a pretrained network for deployment to an ARM Mali graphics processor.

Get the pretrained GoogLeNet model by using the `googlenet` function. This function requires the Deep Learning Toolbox™ Model for GoogLeNet Network. If you have not installed this support package, the function provides a download link. Alternatively, see https://www.mathworks.com/matlabcentral/fileexchange/64456-deep-learning-toolbox-model-for-googlenet-network.

```
net = googlenet;
```

Generate code by using `cnncodegen` with `'targetlib'` set to `'arm-compute-mali'`. By default, the code generator targets version `'19.05'` of the ARM. To target a different version of the Compute Library, use the `'ArmComputeVersion'` parameter.

```
cnncodegen(net,'targetlib','arm-compute-mali'...
,'targetparams',struct('ArmComputeVersion','19.02'));
```

```
---------------------------------------------------------------------
Compilation suppressed: generating code only.
---------------------------------------------------------------------
### Codegen Successfully Generated for arm device
```

The code generator generates the `.cpp` and header files in the `'/pwd/codegen'` folder. The DAG network is generated as a C++ class called `CnnMain`, containing an array of 87 layer classes. The code generator reduces the number of layers is by layer fusion optimization of convolutional and batch normalization layers. The `setup()` method of this class sets up handles and allocates resources for each layer object. The `predict()` method invokes prediction for each of the 87 layers

in the network. The `cleanup()` method releases all the memory and system resources allocated for each layer object. All the binary weights (`cnn_**_w`) and the bias files (`cnn_**_b`) for the convolution layers of the network are stored in the `codegen` folder.

To build the library, move the generated code to the ARM target platform and use the generated makefile `cnnbuild_rtw.mk`.

Input Arguments

`net` — Pretrained deep learning network object

character vector | string scalar

Pretrained `SeriesNetwork` or `DAGNetwork` object.

Note `cnncodegen` does not support `dlnetwork` objects.

`targetparams` — Library-specific parameters

structure

ARM Compute Library-specific parameters specified as a 1-by-1 structure containing the fields described in these tables.

Field	Description
`ArmComputeVersion`	Version of ARM Compute Library on the target hardware, specified as `'19.02'` or `'19.05'`. The default value is `'19.05'`. If you set `ArmComputeVersion` to a version later than `'19.05'`, `ArmComputeVersion` is set to `'19.05'`.

Version History

Introduced in R2017b

Changes to Target Library Support

From R2021b, the `cnncodegen` function generates C++ code and makefiles to build a static library for only the ARM Mali GPU processor by using the ARM Compute Library for computer vision and machine learning.

For all other targets, use the `codegen` command. Write an entry-point function in MATLAB that uses the `coder.loadDeepLearningNetwork` function to load a deep learning model and calls `predict` to predict the responses. For example,

```
function out = googlenet_predict(in) %#codegen

persistent mynet;

if isempty(mynet)
    mynet = coder.loadDeepLearningNetwork('googlenet');
end

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

This table shows some typical usages of `cnncodegen` and how to update your code to use `codegen` instead.

Target workflow	Not recommended	Recommended

ARM CPU processor supporting `NEON` instructions	Set the `'targetlib'` parameter to `'arm-compute'`. Specify the ARM Compute Library version to generate code for and the ARM architecture on the target hardware by using the `'targetparams'` parameter.
	`cnncodegen(net,'targetlib'...`
	`,'arm-compute','targetparams' ...`
	`,struct('ArmComputeVersion'...`
	`,'19.02','ArmArchitecture'...`
	`,'armv8'))`
	
	Other supported versions of ARM Compute Library are `'18.11'`, `'19.02'`, `'19.05'`, or `'20.02.1'`. The default value is `'20.02.1'`. If you set `ArmComputeVersion` to a version later than `'20.02.1'`, `ArmComputeVersion` is set to `'20.02.1'`
	
	You can specify the ARM architecture as `'armv7'` or `'armv8'`. The specified architecture must be the same as the architecture for the ARM Compute Library on the target hardware.
		Create a `coder.config` configuration object for generation of a static library.
		`cfg = coder.config('lib');`
		`cfg.TargetLang = 'C++';`
		
		Create a `coder.ARMNEONConfig` deep learning configuration object. Specify target library-specific properties of the deep learning configuration object. Assign it to the `DeepLearningConfig` property of the `cfg` configuration object.
		`dlcfg = coder.DeepLearningConfig ...`
		`('arm-compute');`
		`dlcfg.ArmArchitecture = 'armv8';`
		`dlcfg.ArmComputeVersion = '19.02';`
		`cfg.DeepLearningConfig = dlcfg;`
		
		Use the `-config` option of the `codegen` function to specify the `cfg` configuration object. The `codegen` function must determine the size, class, and complexity of MATLAB function inputs. Use the `-args` option to specify the size of the input to the entry-point function.
		`arg = {ones(224,224,3,'single')};`
		`codegen -args arg ...`
		`-config cfg googlenet_predict`
		
		For more information, see “Code Generation for Deep Learning Networks with ARM Compute Library”.

Target workflow	Not recommended	Recommended

NVIDIA GPUs by using the CUDA Deep Neural Network library (cuDNN)	Set the `'targetlib'` parameter to `'cudnn'`. Specify cuDNN library-specific properties by using the `'targetparams'` parameter.
	`cnncodegen(net,'targetlib'...`
	`,'cudnn','ComputeCapability'...`
	`,'7.0','targetparams' ...`
	`,struct('AutoTuning',true ...`
	`,'DataType','INT8'...`
	`,'CalibrationResultFile' ...`
	`'myInt8Cal.mat'))`
	The auto tuning feature allows the cuDNN library to find the fastest convolution algorithms.
	The `'DataType'` parameter specifies the precision of the inference computations in supported layers. When performing inference in 32-bit floats, use `'FP32'`.
		Create a `coder.gpuConfig` configuration object for generation of a static library.
		`cfg = coder.gpuConfig('lib');`
		`cfg.TargetLang = 'C++';`
		To set the minimum compute capability for code generation, use the `ComputeCapability` property of the GPU code configuration object.
		`cfg.GpuConfig.ComputeCapability = '7.0';`
		Create a `coder.CuDNNConfig` deep learning configuration object. Specify target library-specific properties of the deep learning configuration object. Assign it to the `DeepLearningConfig` property of the `cfg` configuration object.
		`dlcfg = coder.DeepLearningConfig('cudnn`
		`dlcfg.AutoTuning = true;`
		`dlcfg.DataType = 'int8';`
		`dlcfg.CalibrationResultFile = 'myInt8Ca`
		`cfg.DeepLearningConfig = dlcfg;`
		Use the `-config` option of the `codegen` function to specify the `cfg` configuration object. The `codegen` function must determine the size, class, and complexity of MATLAB function inputs. Use the `-args` option to specify the size of the input to the entry-point function.
		`arg = {ones(224,224,3,'single')};`
		`codegen -args arg ...`
		`-config cfg googlenet_predict`
		For more information, see “Code Generation for Deep Learning Networks by Using cuDNN”.

Target workflow	Not recommended	Recommended

Intel CPU processor	To use the Intel Math Kernel Library for Deep Neural Networks (MKL-DNN) for Intel CPUs, set the `'targetlib'` parameter to `'mkldnn'`.
	`cnncodegen(net,'targetlib'...`
	`,'mkldnn');`
		Create a `coder.config` configuration object for generation of a static library.
		`cfg = coder.config('lib');`
		`cfg.TargetLang = 'C++';`
		
		Create a `coder.MklDNNConfig` deep learning configuration object. Assign it to the `DeepLearningConfig` property of the `cfg` configuration object.
		`dlcfg = coder.DeepLearningConfig...`
		`('mkldnn');`
		`cfg.DeepLearningConfig = dlcfg;`
		
		Use the `-config` option of the `codegen` function to specify the `cfg` configuration object. The `codegen` function must determine the size, class, and complexity of MATLAB function inputs. Use the `-args` option to specify the size of the input to the entry-point function.
		`arg = {ones(224,224,3,'single')};`
		`codegen -args arg ...`
		`-config cfg googlenet_predict`
		
		For more information, see “Code Generation for Deep Learning Networks with MKL-DNN”.

Target workflow	Not recommended	Recommended

NVIDIA GPUs by using NVIDIA TensorRT, a high performance deep learning inference optimizer and run-time library	Set the `'targetlib'` parameter to `'tensorrt'`. Specify TensorRT library-specific properties by using the `'targetparams'` parameter.
	`cnncodegen(net,'targetlib'...`
	`,'tensorrt','ComputeCapability'...`
	`,'7.0','targetparams' ...`
	`,struct('DataType','INT8' ...`
	`'DataPath','image_dataset'...`
	`,'NumCalibrationBatches',50))`
		Create a `coder.gpuConfig` configuration object for generation of a static library.
		`cfg = coder.gpuConfig('lib');`
		`cfg.TargetLang = 'C++';`
		
		To set the minimum compute capability for code generation, use the `ComputeCapability` property of the GPU code configuration object.
		`cfg.GpuConfig.ComputeCapability = '7.0';`
		
		Create a `coder.TensorRTConfig` deep learning configuration object. Specify target library-specific properties of the deep learning configuration object. Assign it to the `DeepLearningConfig` property of the `cfg` configuration object.
		`dlcfg = coder.DeepLearningConfig...`
		`('cudnn');`
		`dlcfg.DataType = 'int8';`
		`dlcfg.DataPath = 'image_dataset';`
		`dlcfg.NumCalibrationBatches = 50;`
		`cfg.DeepLearningConfig = dlcfg;`
		
		Use the `-config` option of the `codegen` function to specify the `cfg` configuration object. The `codegen` function must determine the size, class, and complexity of MATLAB function inputs. Use the `-args` option to specify the size of the input to the entry-point function.
		`arg = {ones(224,224,3,'single')};`
		`codegen -args arg ...`
		`-config cfg googlenet_predict`
		
		For more information, see “Deep Learning Prediction with NVIDIA TensorRT Library”.

Target workflow	Not recommended	Recommended

General options	Generate code without generating and building a makefile. For example,
	`cnncodegen(net,'targetlib' ...`
	`,'mkldnn','codegenonly',1);`
		To produces the source code without invoking the make command or build object code, use the `GenCodeOnly` property of the `coder.CodeConfig` or `coder.GPUCodeConfig` object. For example,
		`cfg = coder.codeConfig('lib');`
		`cfg.GenCodeOnly = true;`

	Specifying the NVIDIA GPU compute capability to compile for. Argument takes the format of `major#.minor#`.
	`cnncodegen(net,'targetlib'...`
	`,'cudnn','ComputeCapability','7.0');`
		To set the minimum compute capability for code generation, use the `ComputeCapability` property of the GPU code configuration object.
		`cfg = coder.gpuConfig('lib');`
		`cfg.GpuConfig.ComputeCapability = '7.0';`

See Also

Functions
`codegen` | `coder.loadDeepLearningNetwork`

Topics
"Code Generation for Deep Learning Networks with MKL-DNN"
"Deep Learning Prediction with ARM Compute Using codegen"
"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"
"Code Generation for Object Detection by Using YOLO v2"
"Deep Learning Prediction with NVIDIA TensorRT Library"

coder.loadDeepLearningNetwork

Load deep learning network model

Syntax

```
net = coder.loadDeepLearningNetwork(filename)
net = coder.loadDeepLearningNetwork(functionname)
net = coder.loadDeepLearningNetwork( ___ ,network_name)
```

Description

`net = coder.loadDeepLearningNetwork(filename)` loads a pretrained deep learning `SeriesNetwork`, `DAGNetwork`, `yolov2ObjectDetector`, `ssdObjectDetector`, or `dlnetwork` object saved in the `filename` MAT-file. `filename` must be a valid MAT-file existing on the MATLAB path containing a single `SeriesNetwork`, `DAGNetwork`, `yolov2ObjectDetector`, `ssdObjectDetector`, or `dlnetwork` object. The MAT-file must contain only the network to be loaded.

`net = coder.loadDeepLearningNetwork(functionname)` calls a function that returns a pretrained deep learning `SeriesNetwork`, `DAGNetwork`, `yolov2ObjectDetector`, `ssdObjectDetector`, or `dlnetwork` object. `functionname` must be the name of a function existing on the MATLAB path that returns a `SeriesNetwork`, `DAGNetwork`, `yolov2ObjectDetector`, `ssdObjectDetector`, or `dlnetwork` object.

`net = coder.loadDeepLearningNetwork( ___ ,network_name)` is the same as `net = coder.loadDeepLearningNetwork(filename)` with the option to name the C++ class generated from the network. `network_name` is a descriptive name for the network object saved in the MAT-file or pointed to by the function. The network name must be a `char` type that is a valid identifier in C++.

Use this function when generating code from a network object inference. This function generates a C ++ class from this network. The class name is derived from the MAT-file name or the function name.

Note The input argument of `coder.loadDeepLearningNetwork` must be a compile-time constant.

Examples

Generate C++ Code from a MAT-File Containing the VGG-16 Network

Use of the `coder.loadDeepLearningNetwork` function to load an `VGG-16` series network and generate C++ code for this network.

Get the MAT-file containing the pretrained `VGG-16` network.

```
url = 'https://www.mathworks.com/supportfiles/gpucoder/cnn_models/VGG/vgg16.mat';
websave('vgg16.mat',url);
```

Create an entry-point function `myVGG16` that uses the `coder.loadDeepLearningNetwork` function to load the `vgg16.mat` into the persistent `mynet` `SeriesNetwork` object.

```
function out = myVGG16(in)

persistent mynet;
if isempty(mynet)
    mynet = coder.loadDeepLearningNetwork('vgg16.mat', 'myVGGnet');
end

out = predict(mynet,in);
```

The persistent object avoids reconstructing and reloading the network object during subsequent calls to the function to invoke the `predict` method on the input.

The input layer of the pretrained `VGG-16` network accepts images of size `224x224x3`. Use the following lines of code to read an input image from a graphics file and resize it to `224x224`.

```
in = imread('peppers.png');
in = imresize(in,[224,224]);
```

Create a `coder.config` configuration object for MEX code generation and set the target language to C++. On the configuration object, set `DeepLearningConfig` with `targetlib` as `'mkldnn'`. The `codegen` function must determine the size, class, and complexity of MATLAB function inputs. Use the `-args` option to specify the size of the input to the entry-point function. Use the `-config` option to pass the code configuration object.

```
cfg = coder.config('mex');
cfg.TargetLang = 'C++';
cfg.DeepLearningConfig = coder.DeepLearningConfig('mkldnn');
codegen -args {ones(224,224,3,'uint8')} -config cfg myVGG16 -report;
```

The `codegen` command places all the generated files in the `codegen` folder. The folder contains the C++ code for the entry-point function `myVGG16.cpp`, header and source files containing the C++ class definitions for the convoluted neural network (CNN), weight, and bias files.

Call `VGG-16` predict on the input image and display the top five predicted labels.

```
predict_scores = myVGG16_mex(in);
[scores,indx] = sort(predict_scores, 'descend');
net = coder.loadDeepLearningNetwork('vgg16.mat');
classNames = net.Layers(end).Classes;
disp(classNames(indx(1:5)));
```

```
     bell pepper
     cucumber
     grocery store
     acorn squash
     butternut squash
```

Code Generation for a `SeriesNetwork` Inference Loaded from a MATLAB Function

Use of the `coder.loadDeepLearningNetwork` function to load an `resnet50` series network and generate CUDA code for this network.

Create an entry-point function `resnetFun` that uses the `coder.loadDeepLearningNetwork` function to call the Deep Learning Toolbox toolbox function `resnet50`. This function returns a pretrained `ResNet-50` network.

```
function out = resnetFun(in)

persistent mynet;
if isempty(mynet)
    mynet = coder.loadDeepLearningNetwork('resnet50', 'myresnet');
end

out = predict(mynet,in);
```

The persistent object avoids reconstructing and reloading the network object during subsequent calls to the function to invoke the `predict` method on the input.

The input layer of the pretrained `ResNet-50` network accepts images of size `224x224x3`. To read an input image from a graphics file and resize it to `224x224`, use the following lines of code:

```
in = imread('peppers.png');
in = imresize(in,[224,224]);
```

Create a `coder.gpuConfig` configuration object for MEX code generation and set the target language to C++. The `codegen` function must determine the size, class, and complexity of MATLAB function inputs. Use the `-args` option to specify the size of the input to the entry-point function and the `-config` option to pass the code configuration object.

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

The `codegen` command places all the generated files in the `codegen` folder. It contains the CUDA code for the entry-point function `resnetFun.cu`, header, and source files containing the C++ class definitions for the convoluted neural network (CNN), weight, and bias files.

Input Arguments

`filename` — MAT file name

name

Specifies the name of the MAT-file containing the pretrained `SeriesNetwork`, `DAGNetwork`, `yolov2ObjectDetector`, `ssdObjectDetector`, or `dlnetwork` object.

This input argument must be a compile-time constant.

Data Types: `string`

`functionname` — MATLAB function name

name

Specifies the name of the function that returns a pretrained `SeriesNetwork`, `DAGNetwork`, `yolov2ObjectDetector`, `ssdObjectDetector`, or `dlnetwork` object.

This input argument must be a compile-time constant.

Data Types: `string`

`network_name` — Descriptive name

name

Descriptive name for the network object saved in the MAT-file. It must be a `char` type that is a valid identifier in C++.

This input argument must be a compile-time constant.

Data Types: `char`

Output Arguments

net — Network object
`SeriesNetwork` object | `DAGNetwork` object | `yolov2ObjectDetector` object | `ssdObjectDetector` object | `dlnetwork` object

Network inference, returned as a `SeriesNetwork`, `DAGNetwork`, `yolov2ObjectDetector`, `ssdObjectDetector`, or `dlnetwork` object.

Limitations

- `coder.loadDeepLearningNetwork` does not support loading MAT-files with multiple networks.
- The MAT-file must contain only the network to be loaded.

Version History

Introduced in R2017b

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

GPU Code Generation
Generate CUDA® code for NVIDIA® GPUs using GPU Coder™.

See Also

Functions
`codegen` | `cnncodegen`

Objects
`SeriesNetwork` | `DAGNetwork` | `yolov2ObjectDetector` | `ssdObjectDetector` | `dlarray` | `dlnetwork`

Topics
"Load Pretrained Networks for Code Generation"
"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"
"Code Generation for dlarray"

coder.DeepLearningConfig

Create deep learning code generation configuration objects

Syntax

```
deepLearningCfg = coder.DeepLearningConfig(TargetLibrary = targetlib)
```

Description

`deepLearningCfg = coder.DeepLearningConfig(TargetLibrary = targetlib)` creates a deep learning configuration object containing library-specific parameters that `codegen` uses to generate code for deep neural networks. Assign this deep learning configuration object to the `DeepLearningConfig` property of the code configuration object created by using `coder.config`. Pass the code configuration object to the `codegen` function by using the `-config` option.

Examples

Generate Code for the ResNet-50 Network Using Intel MKL-DNN Library

Set the code configuration parameters and generate C++ code for an `ResNet-50` series network. The generated code uses the Intel MKL-DNN deep learning libraries.

Create an entry-point function `resnet_predict` that uses the `coder.loadDeepLearningNetwork` function to load the `resnet50` `SeriesNetwork` object.

```
function out = resnet_predict(in)

persistent mynet;
if isempty(mynet)
    mynet = coder.loadDeepLearningNetwork('resnet50', 'myresnet');
end

out = predict(mynet,in);
```

The persistent object avoids reconstructing and reloading the network object during subsequent calls to the function to invoke the `predict` method on the input.

The input layer of the pretrained `ResNet-50` network accepts images of size `224x224x3`. To read an input image from a graphics file and resize it to `224x224`, use the following lines of code:

```
in = imread('peppers.png');
in = imresize(in,[224,224]);
```

Create a `coder.config` configuration object for MEX code generation and set the target language to C++. On the configuration object, set `DeepLearningConfig` with `targetlib` as `'mkldnn'`. Use the `-config` option of the `codegen` function to pass this code configuration object. The `codegen` function must determine the size, class, and complexity of MATLAB function inputs. Use the `-args` option to specify the size of the input to the entry-point function.

```
cfg = coder.config('mex');
cfg.TargetLang = 'C++';
```

```
cfg.DeepLearningConfig = coder.DeepLearningConfig(TargetLibrary = 'mkldnn');
codegen -args {ones(224,224,3,'single')} -config cfg resnet_predict;
```

The `codegen` command places all the generated files in the `codegen` folder. It contains the C++ code for the entry-point function `resnet_predict.cpp`, header and source files containing the C++ class definitions for the convoluted neural network (CNN), weight, and bias files.

Input Arguments

`targetlib` — Specify the target deep learning library

character vector | string scalar

Target library for deep learning code generation, specified as one of the values in this table.

Value	Description
`'none'`	For generating code that does not use any third-party library.
`'arm-compute'`	For generating code that uses the ARM Compute Library.
`'mkldnn'`	For generating code that uses the Intel Math Kernel Library for Deep Neural Networks (Intel MKL-DNN).
`'cmsis-nn'`	Common Microcontroller Software Interface Standard - Neural Network (CMSIS-NN) library. Requires the MATLAB Coder Interface for Deep Learning Libraries.
`'cudnn'`	For generating code that uses the CUDA Deep Neural Network library (cuDNN). This option requires GPU Coder.
`'tensorrt'`	For generating code that takes advantage of the NVIDIA TensorRT – high performance deep learning inference optimizer and run-time library. This option requires GPU Coder.

Output Arguments

`deepLearningCfg` — Deep learning configuration object

`Configuration Object`

Configuration object based on the target library specified in the input argument. This object contains library-specific parameters that are used during code generation.

Target Library	Deep Learning Configuration Object
`'none'`	Creates an `DeepLearningConfigBase` configuration object.
`'arm-compute'`	Creates an `ARMNEONConfig` configuration object.

Target Library	Deep Learning Configuration Object
`'mkldnn'`	Creates an `MklDNNConfig` configuration object.
`'cmsis-nn`	Creates a `CMSISNNConfig` configuration object.
`'cudnn'`	Creates a `CuDNNConfig` configuration object.
`'tensorrt'`	Creates a `TensorRTConfig` configuration object.

Version History

Introduced in R2018b

See Also

Functions

`codegen`

Objects

`coder.CodeConfig` | `coder.loadDeepLearningNetwork` | `coder.MklDNNConfig` | `coder.ARMNEONConfig` | `coder.TensorRTConfig` | `coder.CuDNNConfig` | `coder.CMSISNNConfig`

Topics

"Code Generation for Deep Learning Networks with MKL-DNN"
"Code Generation for Deep Learning Networks with ARM Compute Library"
"Generate Generic C/C++ Code for Deep Learning Networks"
"Code Generation for Object Detection by Using YOLO v2"
"Code Generation for Deep Learning Networks by Using cuDNN"
"Code Generation for Deep Learning Networks by Using TensorRT"

coder.MklDNNConfig

Parameters to configure deep learning code generation with the Intel Math Kernel Library for Deep Neural Networks

Description

The `coder.MklDNNConfig` object contains the Intel MKL-DNN specific parameters that `codegen` uses for generating C++ code for deep neural networks.

To use a `coder.MklDNNConfig` object for code generation, assign it to the `DeepLearningConfig` property of a code generation configuration object that you pass to `codegen`.

Creation

Create an MKL-DNN configuration object by using the `coder.DeepLearningConfig` function with target library set as `'mkldnn'`.

Properties

`TargetLib` — Target library name
'mkldnn'

Name of target library, specified as a character vector.

Examples

Specify Configuration Parameters for MEX Function Generation for the ResNet-50 Network

Create an entry-point function `resnet_predict` that uses the `coder.loadDeepLearningNetwork` function to load the `resnet50` `SeriesNetwork` object.

```
function out = resnet_predict(in)

persistent mynet;
if isempty(mynet)
    mynet = coder.loadDeepLearningNetwork('resnet50', 'myresnet');
end

out = predict(mynet,in);
```

Create a `coder.config` configuration object for MEX code generation.

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

Set the target language to C++.

```
cfg.TargetLang = 'C++';
```

Create a `coder.MklDNNConfig` deep learning configuration object. Assign it to the `DeepLearningConfig` property of the `cfg` configuration object.

```
cfg.DeepLearningConfig = coder.DeepLearningConfig('mkldnn');
```

Use the `-config` option of the `codegen` function to pass the `cfg` configuration object. The `codegen` function must determine the size, class, and complexity of MATLAB function inputs. Use the `-args` option to specify the size of the input to the entry-point function.

```
codegen -args {ones(224,224,3,'single')} -config cfg resnet_predict
```

The `codegen` command places all the generated files in the `codegen` folder. The folder contains the C++ code for the entry-point function `resnet_predict.cpp`, header, and source files containing the C++ class definitions for the convolutional neural network (CNN), weight, and bias files.

Version History

Introduced in R2018b

See Also

`codegen` | `coder.CodeConfig` | `coder.DeepLearningConfig` | `coder.TensorRTConfig` | `coder.CuDNNConfig` | `coder.ARMNEONConfig`

Topics

"Code Generation for Deep Learning Networks with MKL-DNN"
"Code Generation for Deep Learning Networks with ARM Compute Library"
"Code Generation for Deep Learning Networks by Using cuDNN"
"Code Generation for Deep Learning Networks by Using TensorRT"

coder.CuDNNConfig

Parameters to configure deep learning code generation with the CUDA Deep Neural Network library

Description

The `coder.CuDNNConfig` object contains NVIDIA cuDNN specific parameters that `codegen` uses for generating CUDA code for deep neural networks.

To use a `coder.CuDNNConfig` object for code generation, assign it to the `DeepLearningConfig` property of a `coder.gpuConfig` object that you pass to `codegen`.

Creation

Create a cuDNN configuration object by using the `coder.DeepLearningConfig` function with target library set as `'cudnn'`.

Properties

`AutoTuning` — Enable auto tuning
true (default) | false

Enable or disable auto tuning feature. Enabling auto tuning allows the cuDNN library to find the fastest convolution algorithms. This increases performance for larger networks such as `SegNet` and `ResNet`

`DataType` — Inference computation precision
`'fp32'` (default) | `'int8'`

Specify the precision of the inference computations in supported layers. When performing inference in 32-bit floats, use `'fp32'`. For 8-bit integer, use `'int8'`. Default value is `'fp32'`.

`INT8` precision requires a CUDA GPU with minimum compute capability of 6.1. Compute capability of 6.2 does not support `INT8` precision. Use the `ComputeCapability` property of the `GpuConfig` object to set the appropriate compute capability value.

Note When performing inference in `INT8` precision using cuDNN version 8.1.0, issues in the NVIDIA library may cause significant degradation in performance.

`CalibrationResultFile` — Location of calibration MAT-file
`''` (default) | character vector | string scalar

Location of the MAT-file containing the calibration data. Default value is `''`. This option is applicable only when `DataType` is set to `'int8'`.

When performing quantization of a deep convolutional neural network, the `calibrate` function exercises the network and collects the dynamic ranges of the weights and biases in the convolution

and fully connected layers of the network and the dynamic ranges of the activations in all layers of the network. To generate code for the optimized network, save the results from the `calibrate` function to a MAT-file and specify the location of this MAT-file to the code generator using this property. For more information, see “Generate INT8 Code for Deep Learning Networks”.

`TargetLib` — Target library name
'cudnn' (default) | character vector

A read-only value that specifies the name of the target library.

Examples

Specify Configuration Parameters for MEX Function Generation for the ResNet-50 Network

Create an entry-point function `resnet_predict` that uses the `coder.loadDeepLearningNetwork` function to load the `resnet50` `SeriesNetwork` object.

```
function out = resnet_predict(in)

persistent mynet;
if isempty(mynet)
    mynet = coder.loadDeepLearningNetwork('resnet50', 'myresnet');
end

out = predict(mynet,in);
```

Create a `coder.gpuConfig` configuration object for MEX code generation.

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

Set the target language to C++.

```
cfg.TargetLang = 'C++';
```

Create a `coder.CuDNNConfig` deep learning configuration object and assign it to the `DeepLearningConfig` property of the `cfg` configuration object.

```
cfg.DeepLearningConfig = coder.DeepLearningConfig('cudnn');
```

Use the `-config` option of the `codegen` function to pass the `cfg` configuration object. The `codegen` function must determine the size, class, and complexity of MATLAB function inputs. Use the `-args` option to specify the size of the input to the entry-point function.

```
codegen -args {ones(224,224,3,'single')} -config cfg resnet_predict;
```

The `codegen` command places all the generated files in the `codegen` folder. The folder contains the CUDA code for the entry-point function `resnet_predict.cu`, header, and source files containing the C++ class definitions for the convoluted neural network (CNN), weight, and bias files.

Version History

Introduced in R2018b

See Also

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

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

Topics
"Code Generation for Deep Learning Networks"
"Train and Deploy Fully Convolutional Networks for Semantic Segmentation"
"Code Generation for Deep Learning Networks by Using cuDNN"

coder.TensorRTConfig

Parameters to configure deep learning code generation with the NVIDIA TensorRT library

Description

The `coder.TensorRTConfig` object contains NVIDIA high performance deep learning inference optimizer and run-time library (TensorRT) specific parameters. `codegen` uses those parameters for generating CUDA code for deep neural networks.

To use a `coder.TensorRTConfig` object for code generation, assign it to the `DeepLearningConfig` property of a `coder.gpuConfig` object that you pass to `codegen`.

Creation

Create a TensorRT configuration object by using the `coder.DeepLearningConfig` function with target library set as `'tensorrt'`.

Properties

`DataType` — Inference computation precision
`'fp32'` (default) | `'fp16'` | `'int8'`

Specify the precision of the inference computations in supported layers. When performing inference in 32-bit floats, use `'fp32'`. For half-precision, use `'fp16'`. For 8-bit integer, use `'int8'`. Default value is `'fp32'`.

`INT8` precision requires a CUDA GPU with minimum compute capability of 6.1, 7.0 or higher. Compute capability of 6.2 does not support `INT8` precision. `FP16` precision requires a CUDA GPU with minimum compute capability of 5.3, 6.0, 6.2 or higher. Use the `ComputeCapability` property of the `GpuConfig` object to set the appropriate compute capability value.

See the “Deep Learning Prediction with NVIDIA TensorRT Library” example for 8-bit integer prediction for a logo classification network by using TensorRT.

`DataPath` — Image dataset location
`''` (default) | character vector | string scalar

Location of the image dataset used during recalibration. Default value is `''`. This option is applicable only when `DataType` is set to `'int8'`.

When you select the `'INT8'` option, TensorRT quantizes the floating-point data to `int8`. The recalibration is performed with a reduced set of the calibration data. The calibration data must be present in the image data location specified by `DataPath`.

`NumCalibrationBatches` — Number of calibration batches
`50` (default) | positive integer

Numeric value specifying the number of batches for `int8` calibration. The software uses the product of `batchsize*NumCalibrationBatches` to pick a random subset of images from the image dataset

to perform calibration. The `batchsize*NumCalibrationBatches` value must not be greater than the number of images present in the image dataset. This option is applicable only when `DataType` is set to `'int8'`.

NVIDIA recommends that about 500 images are sufficient for calibrating. Refer to the TensorRT documentation for more information.

`TargetLib` — Target library name
'tensorrt' (default) | character vector

A read-only value that specifies the name of the target library.

Examples

Specify Configuration Parameters for MEX Function Generation for the ResNet-50 Network

Create an entry-point function `resnet_predict` that uses the `coder.loadDeepLearningNetwork` function to load the `resnet50` `SeriesNetwork` object.

```
function out = resnet_predict(in)

persistent mynet;
if isempty(mynet)
    mynet = coder.loadDeepLearningNetwork('resnet50', 'myresnet');
end

out = predict(mynet,in);
```

Create a `coder.gpuConfig` configuration object for MEX code generation.

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

Set the target language to C++.

```
cfg.TargetLang = 'C++';
```

Create a `coder.TensorRTConfig` deep learning configuration object. Assign it to the `DeepLearningConfig` property of the `cfg` configuration object.

```
cfg.DeepLearningConfig = coder.DeepLearningConfig('tensorrt');
```

Use the `-config` option of the `codegen` function to pass the `cfg` configuration object. The `codegen` function must determine the size, class, and complexity of MATLAB function inputs. Use the `-args` option to specify the size of the input to the entry-point function.

```
codegen -args {ones(224,224,3,'single')} -config cfg resnet_predict;
```

The `codegen` command places all the generated files in the `codegen` folder. The folder contains the CUDA code for the entry-point function `resnet_predict.cu`, header and source files containing the C++ class definitions for the convoluted neural network (CNN), weight, and bias files.

Version History

Introduced in R2018b

See Also

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

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

Topics
"Deep Learning Prediction with NVIDIA TensorRT Library"
"Code Generation for Deep Learning Networks by Using TensorRT"

coder.getDeepLearningLayers

Get the list of layers supported for code generation for a specific deep learning library

Note `coder.getDeepLearningLayers` is not recommended. Use `analyzeNetworkForCodegen` instead. For more information, see .“Compatibility Considerations”.

Syntax

```
coder.getDeepLearningLayers
coder.getDeepLearningLayers(TargetLibrary = libraryname)
```

Description

`coder.getDeepLearningLayers` returns the layers supported for code generation that do not use any third-party libraries.

`coder.getDeepLearningLayers(TargetLibrary = libraryname)` returns the layers supported for code generation for a specific deep learning library.

Note To use `coder.getDeepLearningLayers`, you must install the support package that corresponds to `libraryname`:

- For `'none'`, `'arm-compute'`, `'cmsis-nn'`, and `'mkldnn'`, install MATLAB Coder Interface for Deep Learning Libraries.
- For `'cudnn'`, `'tensorrt'`, or `'arm-compute-mali'`, install GPU Coder Interface for Deep Learning Libraries.

Note The `coder.getDeepLearningLayers` function does not list certain custom layers if the associated support packages are not installed. For example, `keras` layers are not listed if the Deep Learning Toolbox Converter for TensorFlow™ Models support package is not installed.

Examples

Get Layers Supported for Code Generation for a Specific Deep Learning Library

Get a list of layers supported for code generation for Intel Math Kernel Library for Deep Neural Networks.

```
coder.getDeepLearningLayers(TargetLibrary = 'mkldnn')
```

```
ans =

  59×1 cell array

    {'AdditionLayer'                                }
```

```
    {'AnchorBoxLayer'                                }
    {'AveragePooling2DLayer'                         }
    {'BatchNormalizationLayer'                       }
    {'BiLSTMLayer'                                   }
    {'ClassificationOutputLayer'                     }
    {'ClippedReLULayer'                              }
    {'ConcatenationLayer'                            }
    {'Convolution2DLayer'                            }
    {'Crop2DLayer'                                   }
    {'CrossChannelNormalizationLayer'                }
    {'DepthConcatenationLayer'                       }
    {'DepthToSpace2DLayer'                           }
    {'DicePixelClassificationLayer'                  }
    {'DropoutLayer'                                  }
    {'ELULayer'                                      }
    {'FeatureInputLayer'                             }
    {'FlattenLayer'                                  }
    {'FocalLossLayer'                                }
    {'FullyConnectedLayer'                           }
    {'GRULayer'                                      }
    {'GlobalAveragePooling2DLayer'                   }
    {'GlobalMaxPooling2DLayer'                       }
    {'GroupedConvolution2DLayer'                     }
    {'ImageInputLayer'                               }
    {'LSTMLayer'                                     }
    {'LeakyReLULayer'                                }
    {'MaxPooling2DLayer'                             }
    {'MaxUnpooling2DLayer'                           }
    {'MultiplicationLayer'                           }
    {'PixelClassificationLayer'                      }
    {'RCNNBoxRegressionLayer'                        }
    {'RPNClassificationLayer'                        }
    {'ReLULayer'                                     }
    {'RegressionOutputLayer'                         }
    {'Resize2DLayer'                                 }
    {'SSDMergeLayer'                                 }
    {'SequenceFoldingLayer'                          }
    {'SequenceInputLayer'                            }
    {'SequenceUnfoldingLayer'                        }
    {'SigmoidLayer'                                  }
    {'SoftmaxLayer'                                  }
    {'SpaceToDepthLayer'                             }
    {'TanhLayer'                                     }
    {'TransposedConvolution2DLayer'                  }
    {'WordEmbeddingLayer'                            }
    {'YOLOv2OutputLayer'                             }
    {'YOLOv2ReorgLayer'                              }
    {'YOLOv2TransformLayer'                          }
    {'nnet.keras.layer.FlattenCStyleLayer'           }
    {'nnet.keras.layer.GlobalAveragePooling2dLayer'}
    {'nnet.keras.layer.SigmoidLayer'                 }
    {'nnet.keras.layer.TanhLayer'                    }
    {'nnet.keras.layer.ZeroPadding2dLayer'           }
    {'nnet.onnx.layer.ElementwiseAffineLayer'        }
    {'nnet.onnx.layer.FlattenLayer'                  }
    {'nnet.onnx.layer.IdentityLayer'                 }
```

```
    {'ScalingLayer'                                 }
    {'SoftplusLayer'                                }
```

Input Arguments

`libraryname` — Name of deep learning library

character vector | string scalar

Name of deep learning library, specified as one of the values in this table.

Value	Description
`'arm-compute'`	ARM Compute Library for targeting ARM CPU processors. Requires the MATLAB Coder Interface for Deep Learning Libraries.
`'arm-compute-mali'`	ARM Compute Library for targeting ARM GPU processors. Requires the GPU Coder product and the GPU Coder Interface for Deep Learning Libraries.
`'cmsis-nn'`	Common Microcontroller Software Interface Standard - Neural Network (CMSIS-NN) library. Requires the MATLAB Coder Interface for Deep Learning Libraries.
`'cudnn'`	NVIDIA CUDA Deep Neural Network library (cuDNN). Requires the GPU Coder product and the GPU Coder Interface for Deep Learning Libraries.
`'mkldnn'`	Intel Math Kernel Library for Deep Neural Networks. Requires the MATLAB Coder Interface for Deep Learning Libraries.
`'none'`	`coder.getDeepLearningLayers(TargetLibrary = 'none')` returns the layers supported for code generation that do not use any third-party libraries. This is equivalent to calling `coder.getDeepLearningLayers` with no input argument. Requires the MATLAB Coder Interface for Deep Learning Libraries.
`'tensorrt'`	NVIDIA TensorRT high performance deep learning inference optimizer and run-time library. Requires the GPU Coder product and the GPU Coder Interface for Deep Learning Libraries.

Version History

Introduced in R2018b

`coder.getDeepLearningLayers` is not recommended

Not recommended starting in R2022b

`coder.getDeepLearningLayers` is not recommended. Use `analyzeNetworkForCodegen` instead.

Starting in R2022b, check the code generation compatibility of a deep learning network by using the `analyzeNetworkForCodegen` function.

See Also

Functions

`codegen` | `coder.loadDeepLearningNetwork`

Topics

"Installing Prerequisite Products"
"Prerequisites for Deep Learning with MATLAB Coder"
"Supported Networks, Layers, and Classes"
"Networks and Layers Supported for Code Generation"
"Code Generation for Deep Learning Networks with MKL-DNN"
"Code Generation for Deep Learning Networks with ARM Compute Library"
"Generate Generic C/C++ Code for Deep Learning Networks"
"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"

gpucoderexamples

Product examples

Syntax

```
gpucoderexamples
```

Description

`gpucoderexamples` displays the GPU Coder examples.

Examples

Display GPU Coder Examples

Enter the following in the MATLAB Command Window:

```
gpucoderexamples
```

Version History

Introduced in R2017b

See Also

`GPU Coder` | `codegen`

coder.gpu.nokernel

Pragma to disable kernel creation for loops

Syntax

`coder.gpu.nokernel()`

Description

`coder.gpu.nokernel()` is a loop level pragma that when placed immediately before a for loop prevents the code generator from generating CUDA kernels for the statements within the loop. This pragma does not require any input parameters.

This function is a code generation function. It has no effect in MATLAB.

Examples

Generate CUDA Code for a Simple Nested Loop

This example shows how to use the `nokernel` pragma in a function and prevent the code generator from generating CUDA kernels for the statements within the loop

In one file, write the entry-point function `nestedLoop` that accepts two vector inputs `A,B` of size `32x512`. The function has two nested `for`-loops of different iteration lengths, one for operating along the column and one for operating along the row. The first nested loop computes the sum of the two vector inputs while the second nested loop scales the sum by a factor of three.

```
function [C] = nestedLoop(A, B)
    G = zeros(32, 512);
    C = zeros(32, 512);

    coder.gpu.kernelfun();
    % This nested loop will be fused
    for i = 1:32
       for j = 1:512
           G(i,j) = A(1,j) + B(1,j);
       end
    end

    coder.gpu.nokernel();
    for i = 1:32
       for j = 1:512
           C(i,j) = G(i,j) * 3;
       end
    end
end
```

Use the `codegen` function to generate CUDA MEX function.

```
cfg = coder.gpuConfig('mex');
cfg.GenerateReport = true;
codegen -config cfg -args {ones(1,512,'double'),ones(1,512,'double')} nestedLoop
```

GPU Coder creates two kernels: `nestedLoop_kernel1` to perform the computation `G(i,j) = A(1,j) + B(1,j);` of the first nested loop and `nestedLoop_kernel2` kernel to perform the computation `C(i,j) = G(i,j) * 3;` of the second nested loop. The second kernel is created for the inner loop of the second nested loop. The `noKernel` pragma is applicable only to the loop that immediately follows the statement. Snippets of the generated kernels are shown.

```
static __global__ __launch_bounds__(512, 1) void nestedLoop_kernel1(const real_T
  B[512], const real_T A[512], real_T G[16384])
{
  uint32_T threadId;
  ...
  if (i < 32) {
    G[i + (j << 5)] = A[j] + B[j];
  }
}
static __global__ __launch_bounds__(512, 1) void nestedLoop_kernel2(real_T G
  [16384], int32_T i, real_T C[16384])
{
  uint32_T threadId;
  ...;
  if (j < 512) {
    C[i + (j << 5)] = G[i + (j << 5)] * 3.0;
  }
```

A snippet of the main function shows that the code generator has fused the first nested loop as indicated by the kernel launch parameters. As mentioned earlier, the outer loop of the second nested loop is the one that is not mapped to a kernel. Hence the code generator places a `for-loop` statement just before the call to the second CUDA kernel `nestedLoop_kernel2`.

```
void nestedLoop(const real_T A[512], const real_T B[512], real_T C[16384])
{
  int32_T i;
  ...
  //  These two loops will be fused
  cudaMemcpy(gpu_B, (void *)&B[0], 4096UL, cudaMemcpyHostToDevice);
  cudaMemcpy(gpu_A, (void *)&A[0], 4096UL, cudaMemcpyHostToDevice);
  nestedLoop_kernel1<<<dim3(32U, 1U, 1U), dim3(512U, 1U, 1U)>>>(*gpu_B, *gpu_A, *
    gpu_G);
  for (i = 0; i < 32; i++) {
    nestedLoop_kernel2<<<dim3(1U, 1U, 1U), dim3(512U, 1U, 1U)>>>(*gpu_G, i,
      *gpu_C);
    C_dirtyOnGpu = true;
  }
...
  cudaFree(*gpu_C);
}
```

Version History

Introduced in R2019a

See Also

Apps

GPU Coder

Functions

`codegen` | `coder.gpu.kernel` | `gpucoder.stencilKernel` | `coder.gpu.constantMemory` | `gpucoder.reduce` | `gpucoder.sort`

Objects

`coder.gpuConfig` | `coder.CodeConfig` | `coder.EmbeddedCodeConfig` | `coder.MexCodeConfig`

Topics

“GPU Programming Paradigm”
“Code Generation by Using the GPU Coder App”
“Code Generation Using the Command Line Interface”
“Kernels from Element-Wise Loops”
“Kernels from Scatter-Gather Type Operations”
“Kernels from Library Calls”
“Design Patterns”

coder.gpu.iterations

Pragma that provides information to the code generator for making parallelization decisions on variable bound loops

Syntax

```
coder.gpu.iterations(AVG_NUM_ITER)
```

Description

`coder.gpu.iterations(AVG_NUM_ITER)` pragma can be used to specify the average number of iterations (`AVG_NUM_ITER`) for a variable-bound `for`-loop that immediately follows it. This value is used to provide heuristics towards making parallelization decisions for imperfect loops. This pragma does not affect fixed-bound `for`-loops.

This is a code generation function. It has no effect in MATLAB.

Examples

Using `coder.gpu.iterations` on a Simple Nested Loop

This example shows how to use the `coder.gpu.iterations` pragma to augment information used by the code generator to make parallelization decisions.

Consider the following MATLAB entry-point function `myFun` containing a simple nested loop.

```
function [a, c] = myFun(b, N1)

coder.gpu.kernelfun();
a = coder.nullcopy(zeros(1, N1));
c = coder.nullcopy(b);

for i = 1:N1             % Loop1
    a(i) = 1;

    for j = 1:20          % Loop2
        c(i,j) = 2 * b(i,j);
    end
end

end
```

In this case, `Loop 1` is an imperfect loop, preventing the code generator from parallelizing the outer loop `Loop 1`.

Modify the entry-point function by using the `coder.gpu.iterations` pragma to inform the code generator the average number of iterations that the loop is expected to execute.

```
function [a, c] = myFun(b, N1)
```

```
coder.gpu.kernelfun();
a = coder.nullcopy(zeros(1, N1));
c = coder.nullcopy(b);

coder.gpu.iterations(25); % AVG_NUM_ITER
for i = 1:N1             % Loop1
    a(i) = 1;

    for j = 1:20          % Loop2
        c(i,j) = 2 * b(i,j);
    end
end

end
```

Loop 1 is parallelized when the `AVG_NUM_ITER > 20` (Loop2 bound) regardless of the value of `N1`.

Input Arguments

`AVG_NUM_ITER` — Specify the average number of iterations
integer

Specify the average number of iterations (`AVG_NUM_ITER`) for a variable-bound `for`-loop that immediately follows the `coder.gpu.iterations` pragma.

Version History

Introduced in R2019a

See Also

Apps
GPU Coder

Functions
`codegen` | `coder.gpu.kernel` | `coder.gpu.kernelfun` | `gpucoder.stencilKernel` | `coder.gpu.constantMemory` | `gpucoder.reduce` | `gpucoder.sort` | `coder.gpu.nokernel`

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

Topics
“GPU Programming Paradigm”
“Code Generation by Using the GPU Coder App”
“Code Generation Using the Command Line Interface”
“Kernels from Element-Wise Loops”
“Kernels from Scatter-Gather Type Operations”
“Kernels from Library Calls”
“Design Patterns”

gpucoder.sort

Optimized GPU implementation of the MATLAB sort function

Syntax

```
B = gpucoder.sort(A)
B = gpucoder.sort(A,dim)
B = gpucoder.sort(A,direction)
[B,I] = gpucoder.sort(A,...)
```

Description

`B = gpucoder.sort(A)` sorts the elements of `A` in ascending order. The sort operation is performed on the GPU with the help of Thrust library. Thrust is a C++ template library for CUDA and is shipped with CUDA Toolkit. The sorted output in `B` has the same type and size as `A`. If `A` is a vector, `gpucoder.sort(A)` sorts the elements of A in ascending order. If `A` is a matrix, `gpucoder.sort(A)` sorts each column of A in ascending order. If `A` is an N-dimensional array, `gpucoder.sort(A)` sorts along the first non-singleton dimension.

`B = gpucoder.sort(A,dim)` has the optional argument `dim` that specifies the dimension along which the sort operation is performed.

`B = gpucoder.sort(A,direction)` has the optional argument `direction` that specifies the sort direction. `direction` can take one of two values:

- `'ascend'` - Sorts in the ascending order. This is the default option
- `'descend'` - Sorts in the descending order.

`[B,I] = gpucoder.sort(A,...)` returns a sort index `I` which specifies how the elements of `A` were rearranged to obtain the sorted output `B`.

- If `A` is a vector, then `B = A(I)`.
- If `A` is an m-by-n matrix and `dim = 1`, then

```
for j = 1:n
  B(:,j) = A(I(:,j),j);
end
```

The sort ordering is stable. Namely, when more than one element has the same value, the order of the equal elements is preserved in the sorted output B and the indices I relating to equal elements are ascending.

When `gpucoder.sort` is called from MATLAB, it uses the built-in `sort` function.

Examples

Sort a Matrix

This example generates CUDA code to sort the columns of a matrix in descending order.

In one file, write an entry-point function `mySort` that accepts a matrix inputs `A`. Use the `gpucoder.sort` function to sort the columns of `A` in descending order.

```
function B = mySort(A)
     B = gpucoder.sort(A, 1, 'descend');
end
```

Use the `codegen` function to generate CUDA MEX function.

```
codegen -config coder.gpuConfig('mex') -args {ones(1024,1024,'double')} -report mySort
```

The following is a snippet of the generated code. The Thrust library call is denoted by `thrustSortImpl`

```
...
cudaMalloc(&gpu_inDims, 8ULL);
cudaMalloc(&gpu_B, 8388608ULL);
cudaMalloc(&gpu_A, 8388608ULL);
mySort_kernel1<<<dim3(1U, 1U, 1U), dim3(32U, 1U, 1U)>>>(*gpu_inDims);
cudaMemcpy(gpu_A, (void *)&A[0], 8388608ULL, cudaMemcpyHostToDevice);
mySort_kernel2<<<dim3(2048U, 1U, 1U), dim3(512U, 1U, 1U)>>>(*gpu_A, *gpu_B);
cudaMemcpy(&inDims[0], gpu_inDims, 8ULL, cudaMemcpyDeviceToHost);
thrustSortImpl(&(*gpu_B)[0], 2, &inDims[0], 1, 'd', false);
cudaMemcpy(&B[0], gpu_B, 8388608ULL, cudaMemcpyDeviceToHost);
...
```

Input Arguments

A — Input array

vector | matrix | multidimensional array

Input array, specified as a vector, matrix, or multidimensional array.

Data Types: `double` | `single` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical` | `char`

dim — Dimension to operate along

positive integer scalar

Dimension to operate along, specified as a positive integer scalar. If no value is specified, then the default is the first array dimension whose size does not equal 1.

`sort` returns `A` if `dim` is greater than `ndims(A)`. `dim` is not supported when `A` is a cell array, that is, `sort` only operates along the first array dimension whose size does not equal 1.

Data Types: `double` | `single` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

direction — Sorting direction

`'ascend'` (default) | `'descend'`

Sorting direction, specified as `'ascend'` or `'descend'`. `direction` is not supported when `A` is a cell array, that is, `sort` only sorts in ascending order.

Output Arguments

B — Sorted array

vector | matrix | multidimensional array

Sorted array, returned as a vector, matrix, or multidimensional array. `B` is the same size and type as `A`. The order of the elements in `B` preserves the order of any equal elements in `A`.

Data Types: double | single | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64 | logical | char

I — Sort index
vector | matrix | multidimensional array

Sort index, returned as a vector, matrix, or multidimensional array. I is the same size as A. The index vectors are oriented along the same dimension that sort operates on. For example, if A is a 2-by-3 matrix, then [B,I] = sort(A,2) sorts the elements in each row of A. The output I is a collection of 1-by-3 row index vectors describing the rearrangement of each row of A.

Limitations

- gpucoder.sort does not support complex numbers.
- gpucoder.sort does not support 'MissingPlacement' and 'ComparisonMethod' name-value pairs supported by the MATLAB sort function.

Version History

Introduced in R2018b

See Also

Apps
GPU Coder

Functions
codegen | coder.gpu.kernel | coder.gpu.kernelfun | gpucoder.stencilKernel | coder.gpu.constantMemory | gpucoder.reduce

Objects
coder.gpuConfig | coder.CodeConfig | coder.EmbeddedCodeConfig | coder.MexCodeConfig

Topics
"GPU Programming Paradigm"
"Code Generation by Using the GPU Coder App"
"Code Generation Using the Command Line Interface"
"Kernels from Element-Wise Loops"
"Kernels from Scatter-Gather Type Operations"
"Kernels from Library Calls"
"Design Patterns"

gpucoder.profile

Create an execution profile report for generated CUDA code

Syntax

```
gpucoder.profile(func_name,codegen_inputs)
gpucoder.profile( ___ ,Name,Value)
```

Description

`gpucoder.profile(func_name,codegen_inputs)` generates an execution profiling report of the CUDA code generated for the design file `func_name`. The `codegen_inputs` argument specifies the inputs to the design file. You must install the Embedded Coder product to generate the profiling report.

Note The profiling workflow depends on the `nvprof` tool from NVIDIA. In CUDA Toolkit v10.1, NVIDIA restricts access to performance counters to admin users. To enable GPU performance counters for all user accounts, see the instructions in Permission issue with Performance Counters (NVIDIA).

`gpucoder.profile( ___ ,Name,Value)` generates an execution profiling report with one or more profiling options specified as a name-value pair argument.

Examples

Execution Profiling Report for the Generated CUDA Code

Perform fine-grain analysis for a MATLAB algorithm and its generated CUDA code through software-in-the-loop (SIL) execution profiling. You must install the Embedded Coder product to generate the execution profiling report.

Write an entry-point function that performs N-D fast Fourier transform. To map the FFT to the GPU, use the `coder.gpu.kernelfun` pragma. By default, the `EnableCUFFT` property is enabled, so the code generator uses the cuFFT library to perform the FFT operation.

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

To generate the execution profiling report, use the `gpucoder.profile` function.

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

The code execution profiling report provides metrics based on data collected from a SIL execution. Execution times are calculated from data recorded by instrumentation probes added to the SIL test

harness or inside the code generated for each component. For more information, see “View Execution Times” (Embedded Coder).

Code Execution Profiling Report for gpu_fftn

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

1. Summary

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

2. Profiled Sections of Code

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

3. GPU Profiling Trace for gpu_fftn

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

OK Help

Input Arguments

`func_name` — Name of the entry-point function
string

Name of the entry-point function or design file.

Example: `gpucoder.profile('xdot',{1000,rand(1000,1),1,1,rand(1000,1),1,1})`

`codegen_inputs` — Inputs to the entry-point function
cell array

Compile-time inputs to the entry-point function or design file.

Example: `gpucoder.profile('xdot',{1000,rand(1000,1),1,1,rand(1000,1),1,1})`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `gpucoder.profile('xdot', {1000,rand(1000,1),1,1,rand(1000,1),1,1},'NumCalls',2,'CodegenConfig',cfg,'CodegenArguments','-d discrete','Threshold',0.01)`

`NumCalls` — Number of executions
6 (default) | positive integer

Specify the number of times the profiled section of the code is run. The default is 6. The first run is excluded from the report because it is generally an outlier.

`CodegenConfig` — Custom code configuration object
`''` (default) | code configuration object

Specify the code generation configuration object used to generate CUDA code and the profiling report. When you do not specify this value, a default `coder.EmbeddedCodeConfig` object is used.

`CodegenArguments` — Additional codegen arguments
`''` (default) | string

Specify any additional `codegen` arguments as a string. The default value is NULL (empty string).

`Threshold` — Threshold value
0.0 (default) | numeric value

To control the GPU calls that are displayed in the report, use the threshold value. Any function call with execution time under the value for the threshold parameter will be filtered from the profiling trace.

Version History

Introduced in R2018b

See Also

Apps
GPU Coder

Functions
`codegen` | `coder.gpu.kernel` | `coder.gpu.kernelfun`

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

Topics
"GPU Programming Paradigm"
"Analyze Execution Profiles of the Generated Code"
"GPU Execution Profiling of the Generated Code"
"Code Generation by Using the GPU Coder App"
"Code Generation Using the Command Line Interface"
"Code Generation for Deep Learning Networks by Using cuDNN"
"Code Generation for Deep Learning Networks by Using TensorRT"

coder.gpuEnvConfig

Create configuration object containing the parameters passed to `coder.checkGpuInstall` for performing GPU code generation environment checks

Description

The `coder.gpuEnvConfig` object contains the configuration parameters that `coder.checkGpuInstall` uses to verify the GPU code generation environment.

Creation

Description

`gpuEnvObj = coder.gpuEnvConfig` creates a `gpuEnvConfig` configuration object for the host development computer.

`gpuEnvObj = coder.gpuEnvConfig(hw)` creates a `gpuEnvConfig` configuration object for the hardware type specified in `hw`. `hw` can take the value of `'host'`, `'jetson'`, or `'drive'`. The Jetson and DRIVE types require the MATLAB Coder Support Package for NVIDIA Jetson and NVIDIA DRIVE Platforms.

Properties

`Hardware` — Type of hardware
`'host'` (default) | `'jetson'` | `'drive'`

This field is a read-only property set at the time of creating a `gpuEnvConfig` configuration object. This field can take the value of `'host'`, `'jetson'`, or `'drive'`. The Jetson and DRIVE types require the MATLAB Coder Support Package for NVIDIA Jetson and NVIDIA DRIVE Platforms.

Example: `gpuEnvObj.Hardware`

`GpuId` — Select GPU device
`0` (default) | `integer`

Select the GPU Device ID that must be used when the environment is checked. By default, `GpuId` is set to 0.

Example: `gpuEnvObj.GpuId = 1;`

`BasicCodegen` — Enable code generation test
`false` (default) | `true`

When this field is set to true, basic GPU code generation check is performed. The generated code is not executed.

Example: `gpuEnvObj.BasicCodegen = true;`

`BasicCodeexec` — Enable code generation and execution test
`false` (default) | `true`

When this field is set to true, basic GPU code generation and execution checks are performed on the selected GPU device.

Example: `gpuEnvObj.BasicCodeexec = true;`

`DeepCodegen` — Enable deep learning code generation test
`false` (default) | `true`

When this field is set to true, deep learning GPU code generation check is performed for the library target indicated by the `DeepLibTarget` property. The generated code is not executed.

Example: `gpuEnvObj.DeepCodegen = true;`

`DeepCodeexec` — Enable deep learning code generation and execution test
`false` (default) | `true`

When this field is set to true, deep learning GPU code generation and execution checks are performed for the library target indicated by the `DeepLibTarget` property on the selected GPU device.

Example: `gpuEnvObj.DeepCodeexec = true;`

`DeepLibTarget` — Deep learning library
`''` (default) | `'cudnn'` | `'tensorrt'`

This field indicates the library target for which deep learning code generation and execution checks are performed.

Example: `gpuEnvObj.DeepLibTarget = 'cudnn';`

`DataType` — TensorRT data precision
`''` (default) | `'fp32'` | `'fp16'` | `'int8'`

This field checks if the compute capability of the selected GPU device meets the minimum compute capability required for the selected TensorRT data precision.

Example: `gpuEnvObj.DataType = 'fp32';`

`GenReport` — Enable HTML report
`false` (default) | `true`

When this field is set to true, an HTML report of the results is generated in the current working folder. The current working folder must be write-enabled.

Example: `gpuEnvObj.GenReport = true;`

`Quiet` — Suppress command-line output
`false` (default) | `true`

When this field is set to true, the output printed on the command line is suppressed.

Example: `gpuEnvObj.Quiet = true;`

`Profiling` — Check nvtx libraries for profiling
`false` (default) | `true`

Check for a properly configured NVTX library installation on the host machine. This library is used for profiling.

Example: `gpuEnvObj.Profiling = true;`

`CudaPath` — Path to the CUDA libraries
`character vector`

This field contains the path to the CUDA libraries on the host. The default value is based on the current `nvcc` location found on the Linux OS and on the "CUDA_PATH" environment variable in Windows OS. You can also modify this value to select a different location.

Example: `gpuEnvObj.CudaPath = '/usr/local/cuda';`

`CudnnPath` — Path to the cuDNN libraries
`character vector`

This field contains the path to the cuDNN libraries on the host. The default value is based on the "NVIDIA_CUDNN" environment variable if set. You can also modify this value to select a different location.

Example: `gpuEnvObj.CudnnPath = '/usr/local/cuda/cudnn';`

`TensorrtPath` — Path to the TensorRT libraries
`character vector`

This field contains the path to the TensorRT libraries on the host. The default value is based on the "NVIDIA_TENSORRT" environment variable if set. You can also modify this value to select a different location.

Example: `gpuEnvObj.TensorrtPath = '/usr/local/cuda/tensorrt';`

`NvtxPath` — Path to the NVTX libraries
`character vector`

This field contains the path to the NVTX libraries on the host. The default value is based on the "NVTOOLSEXT_PATH" environment variable on Windows OS, if set. On Linux, it is obtained from the "LD_LIBRARY_PATH". You can also modify this value to select a different location.

Example: `gpuEnvObj.NvtxPath = '/usr/local/cuda/';`

`HardwareObject` — Jetson or DRIVE object
`object`

This field accepts a "jetson" or a "drive" hardware object. This field needs (for jetson/drive) to be set before running environment checks on the board.

Example: `gpuEnvObj.Hardware = jetsonHwObj;`

`ExecTimeout` — Timeout for long-running applications
10 (default) | `positive integer`

Specify the time in seconds that the software waits before validating the execution tests on the target.

Example: `gpuEnvObj.ExecTimeout = 25;`

Examples

Verify GPU Code Generation Environment

This example shows you how to verify that your development computer has all the tools and configuration needed for GPU code generation.

Create a `coder.gpuEnvConfig` object that you can pass to the `coder.checkGpuInstall` function.

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
```

Version History

Introduced in R2019a

See Also

Apps

GPU Coder | GPU Environment Check

Functions
`getenv` | `setenv` | `gpuDevice` | `gpuDeviceCount` | `codegen` | `coder.gpu.kernel` | `coder.gpu.kernelfun`

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

Topics
"GPU Programming Paradigm"
"Installing Prerequisite Products"
"Setting Up the Prerequisite Products"
"The GPU Environment Check and Setup App"
"Verify Setup"
"Code Generation by Using the GPU Coder App"
"Code Generation Using the Command Line Interface"
"Code Generation for Deep Learning Networks by Using cuDNN"
"Code Generation for Deep Learning Networks by Using TensorRT"

gpucoder.transpose

Optimized GPU implementation of the MATLAB transpose function

Syntax

```
B = gpucoder.transpose(A)
```

Description

`B = gpucoder.transpose(A)` performs efficient out-of-place non-conjugate transpose on the GPU using shared memory. When called from MATLAB (out of the code generation context), `gpucoder.transpose` calls the built-in `transpose` function.

Examples

Transpose a Matrix

This example generates CUDA code to transpose a matrix.

In one file, write an entry-point function `myTranspose` that accepts a matrix inputs `A`. Use the `gpucoder.transpose` function to generate a GPU efficient implementation for transposing `A`.

```
function B = myTranspose(A)
     B = gpucoder.transpose(A);
end
```

Use the `codegen` function to generate CUDA MEX function.

```
codegen -config coder.gpuConfig('mex') -args {ones(1024,1024,'double')} -report myTranspose
```

Input Arguments

A — Input array
vector | matrix

Input array, specified as a vector or matrix.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical` | `char` | `string` | `struct` | `categorical` | `datetime` | `duration` | `calendarDuration`
Complex Number Support: Yes

Output Arguments

B — Transposed array
vector | matrix

Transposed array, returned as a vector or matrix.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical` | `char` | `string` | `struct` | `categorical` | `datetime` | `duration` | `calendarDuration`

Limitations

- `gpucoder.transpose` does not support inputs that are of dimension greater than two.

Version History

Introduced in R2019a

See Also

Apps

GPU Coder

Functions

`codegen` | `coder.gpu.kernel` | `coder.gpu.kernelfun` | `gpucoder.stencilKernel` | `coder.gpu.constantMemory` | `gpucoder.reduce` | `gpucoder.sort`

Objects

`coder.gpuConfig` | `coder.CodeConfig` | `coder.EmbeddedCodeConfig` | `coder.MexCodeConfig`

Topics

"GPU Programming Paradigm"
"Code Generation by Using the GPU Coder App"
"Code Generation Using the Command Line Interface"
"Kernels from Element-Wise Loops"
"Kernels from Scatter-Gather Type Operations"
"Kernels from Library Calls"
"Design Patterns"

gpucoder.reduce

Optimized GPU implementation for reduction operations

Syntax

```
S = gpucoder.reduce(A,FUN)
S = gpucoder.reduce(A,{@FUN1,@FUN2,...})
```

Description

`S = gpucoder.reduce(A,FUN)` aggregates the values present in the input array `A` to a single value using the given function handle `FUN`. The output `S` is a scalar.

`S = gpucoder.reduce(A,{@FUN1,@FUN2,...})` accepts an input array and a cell array of function handles. It aggregates the values present in the input array to a single value for every function handle provided in the cell array. The size of output is 1-by-N, where N is the number of function handles.

The code generator uses `shuffle` intrinsics to perform efficient reduction on the GPU. Multiple function handles are aggregated inside a single kernel on the GPU.

Examples

Sum and Maximum of an Array

This example generates CUDA code to find the sum and the maximum of the elements of an array.

In one file, write an entry-point function `multireduce` that accepts a matrix input `A`. Use the `gpucoder.reduce` function to perform two types of reduction operations on the elements of `A`.

```
function s = multireduce(A)
  s = gpucoder.reduce(A, {@mysum, @mymax});
end

function c = mysum(a, b)
  c = a+b;
end

function c = mymax(a, b)
  c = max(a,b);
end
```

Use the `codegen` function to generate CUDA MEX function.

```
codegen -config coder.gpuConfig('mex') -args {rand(1,1024,'double')} -report multireduce
```

The following is a snippet of the generated code.

```
...
cudaMalloc(&gpu_s, 16ULL);
cudaMalloc(&gpu_A, 8192ULL);
cudaMemcpy(gpu_A, (void *)&A[0], 8192ULL, cudaMemcpyHostToDevice);
multireduce_kernel1<<<dim3(1U, 1U, 1U), dim3(32U, 1U, 1U)>>>(*gpu_A, *gpu_s);
```

```
coder_reduce0<<<dim3(2U, 1U, 1U), dim3(512U, 1U, 1U)>>>(*gpu_A, *gpu_s);
cudaMemcpy(&s[0], gpu_s, 16ULL, cudaMemcpyDeviceToHost);
...
static __inline__ __device__ real_T shflDown2(real_T in1, uint32_T offset,
  uint32_T mask)
{
  int2 tmp;
  tmp = *(int2 *)&in1;
  tmp.x = __shfl_down_sync(mask, tmp.x, offset);
  tmp.y = __shfl_down_sync(mask, tmp.y, offset);
  return *(real_T *)&tmp;
}
...
```

Input Arguments

A — Input array

numeric | logical

The input array to perform the reduction operation on. For code generation, the input array must be of numeric or logical data type.

FUN — Function handle

function handle

Handle to a user-defined function. FUN can also be a cell array of function handles. The function handle is a binary function and must satisfy the following requirements:

- Accept two inputs and returns one output. The type of the inputs and output to the function must match the type of the input array A.
- The function must be commutative and associative, otherwise the behavior is undefined.

Name-Value Pair Arguments

Specify optional pairs of arguments as Name1=Value1,...,NameN=ValueN, where Name is the argument name and Value is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose Name *in quotes.*

Example: `gpucoder.reduce(A, {@mySum, @myMax},'dim',2);`

dim — Reduction dimension

1 (default) | numeric

Perform reduction along the specified dimension

Example: `gpucoder.reduce(A, {@mySum, @myMax},'dim',2);`

preprocess — Preprocessing function

function handle

Apply a preprocessing function to the elements of the input array before performing the reduction operation.

Example: `gpucoder.reduce(A,@mySum,'preprocess',@myScale);`

Output Arguments

S — Output
numeric | logical

Result of the reduction operation. During reduction, S is initialized to the value of one of elements of the input array A. Then, the reduction operation is performed by applying FUN to every element in A and S.

Limitations

- `gpucoder.reduce` does not support input arrays that are of complex data type.
- The user-defined function must accept two inputs and returns one output. The type of the inputs and output to the function must match the type of the input array A.
- The user-defined function must be commutative and associative, otherwise the behavior is undefined.
- For some inputs that are of the integer data type, the generated code may contain intermediate computations that reach saturation. In such cases, the results from the generated code may not match the simulation results from MATLAB.

Version History

Introduced in R2019b

See Also

Apps
GPU Coder

Functions
`codegen` | `coder.gpu.kernel` | `coder.gpu.kernelfun` | `gpucoder.stencilKernel` | `coder.gpu.constantMemory` | `gpucoder.sort`

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

Topics
"GPU Programming Paradigm"
"Code Generation by Using the GPU Coder App"
"Code Generation Using the Command Line Interface"
"Kernels from Element-Wise Loops"
"Kernels from Scatter-Gather Type Operations"
"Kernels from Library Calls"
"Design Patterns"

half

Construct half-precision numeric object

Description

Use the `half` constructor to assign a half-precision data type to a number or variable. A half-precision data type occupies 16 bits of memory, but its floating-point representation enables it to handle wider dynamic ranges than integer or fixed-point data types of the same size. For more information, see “Floating-Point Numbers” (Fixed-Point Designer) and “What is Half Precision?” (Fixed-Point Designer).

For a list of functions that support code generation with half-precision inputs, see “Half Precision Code Generation Support”.

Creation

Syntax

```
a = half(v)
```

Description

`a = half(v)` converts the values in `v` to half-precision.

Input Arguments

v — Input array
scalar | vector | matrix | multidimensional array

Input array, specified as a scalar, vector, matrix, or multidimensional array.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical`

Object Functions

These functions are supported for simulation with half-precision inputs in MATLAB. MATLAB System object™ supports half-precision data type and MATLAB System (Simulink) block supports half-precision data type with real values. For a list of functions that support code generation with half-precision inputs, see “Half Precision Code Generation Support” (Fixed-Point Designer).

Math and Arithmetic

abs	Absolute value and complex magnitude
acos	Inverse cosine in radians
acosh	Inverse hyperbolic cosine
asin	Inverse sine in radians

asinh	Inverse hyperbolic sine
atan	Inverse tangent in radians
atan2	Four-quadrant inverse tangent
atanh	Inverse hyperbolic tangent
ceil	Round toward positive infinity
conj	Complex conjugate
conv	Convolution and polynomial multiplication
conv2	2-D convolution
cos	Cosine of argument in radians
cosh	Hyperbolic cosine
cospi	Compute cos(X*pi) accurately
cumsum	Cumulative sum
dot	Dot product
exp	Exponential
expm1	Compute exp(x)-1 accurately for small values of x
fft	Fast Fourier transform
fft2	2-D fast Fourier transform
fftn	N-D fast Fourier transform
fftshift	Shift zero-frequency component to center of spectrum
fix	Round toward zero
floor	Round toward negative infinity
fma	Multiply and add using fused multiply add approach
hypot	Square root of sum of squares (hypotenuse)
ifft	Inverse fast Fourier transform
ifft2	2-D inverse fast Fourier transform
ifftn	Multidimensional inverse fast Fourier transform
ifftshift	Inverse zero-frequency shift
imag	Imaginary part of complex number
ldivide	Left array division
log	Natural logarithm
log10	Common logarithm (base 10)
log1p	Compute log(1+x) accurately for small values of x
log2	Base 2 logarithm and floating-point number dissection
mean	Average or mean value of array
minus	Subtraction
mldivide	Solve systems of linear equations Ax = B for x
mod	Remainder after division (modulo operation)
mrdivide	Solve systems of linear equations xA = B for x
mtimes	Matrix multiplication
plus	Add numbers, append strings
pow10	Base 10 power and scale half-precision numbers
pow2	Base 2 exponentiation and scaling of floating-point numbers
power	Element-wise power
prod	Product of array elements
rdivide	Right array division
real	Real part of complex number
rem	Remainder after division
round	Round to nearest decimal or integer
rsqrt	Reciprocal square root
sign	Sign function (signum function)
sin	Sine of argument in radians
sinh	Hyperbolic sine

sinpi	Compute sin(X*pi) accurately
sqrt	Square root
sum	Sum of array elements
tan	Tangent of argument in radians
tanh	Hyperbolic tangent
times	Multiplication
uminus	Unary minus
uplus	Unary plus

Data Types

allfinite	Determine if all array elements are finite
anynan	Determine if any array element is NaN
cast	Convert variable to different data type
cell	Cell array
double	Double-precision arrays
eps	Floating-point relative accuracy
flintmax	Largest consecutive integer in floating-point format
Inf	Create array of all Inf values
int16	16-bit signed integer arrays
int32	32-bit signed integer arrays
int64	64-bit signed integer arrays
int8	8-bit signed integer arrays
isa	Determine if input has specified data type
isfloat	Determine whether input is floating-point data type
isinteger	Determine whether input is integer array
islogical	Determine if input is logical array
isnan	Determine which array elements are NaN
isnumeric	Determine whether input is numeric array
isobject	Determine if input is MATLAB object
isreal	Determine whether array uses complex storage
logical	Convert numeric values to logicals
NaN	Create array of all NaN values
realmax	Largest positive floating-point number
realmin	Smallest normalized floating-point number
single	Single-precision arrays
storedInteger	Stored integer value of fi object
typecast	Convert data type without changing underlying data
uint16	16-bit unsigned integer arrays
uint32	32-bit unsigned integer arrays
uint64	64-bit unsigned integer arrays
uint8	8-bit unsigned integer arrays

Relational and Logical Operators

all	Determine if all array elements are nonzero or true
and	Find logical AND
Short-Circuit AND	Logical AND with short-circuiting
any	Determine if any array elements are nonzero
eq	Determine equality
ge	Determine greater than or equal to
gt	Determine greater than
isequal	Determine array equality

isequaln	Determine array equality, treating NaN values as equal
le	Determine less than or equal to
lt	Determine less than
ne	Determine inequality
not	Find logical NOT
or	Find logical OR
Short-Circuit OR	Logical OR with short-circuiting

Array and Matrix Operations

cat	Concatenate arrays
chol	Cholesky factorization
circshift	Shift array circularly
colon	Vector creation, array subscripting, and for-loop iteration
complex	Create complex array
ctranspose	Complex conjugate transpose
empty	Create empty array of specified class
eye	Identity matrix
flip	Flip order of elements
fliplr	Flip array left to right
flipud	Flip array up to down
horzcat	Horizontal concatenation for heterogeneous arrays
iscolumn	Determine if input is column vector
isempty	Determine whether array is empty
isfinite	Determine which array elements are finite
isinf	Determine which array elements are infinite
ismatrix	Determine whether input is matrix
isrow	Determine if input is row vector
isscalar	Determine whether input is scalar
issorted	Determine if array is sorted
isvector	Determine whether input is vector
length	Length of largest array dimension
lu	LU matrix factorization
max	Maximum elements of an array
min	Minimum elements of an array
ndims	Number of array dimensions
numel	Number of array elements
ones	Create array of all ones
permute	Permute array dimensions
repelem	Repeat copies of array elements
repmat	Repeat copies of array
reshape	Reshape array
size	Array size
sort	Sort array elements
squeeze	Remove dimensions of length 1
transpose	Transpose vector or matrix
vertcat	Vertical concatenation for heterogeneous arrays
zeros	Create array of all zeros

Graphics

area	Area of 2-D alpha shape
bar	Bar graph

barh	Horizontal bar graph
fplot	Plot expression or function
line	Create primitive line
plot	2-D line plot
plot3	3-D point or line plot
plotmatrix	Scatter plot matrix
rgbplot	Plot colormap
scatter	Scatter plot
scatter3	3-D scatter plot
xlim	Set or query x-axis limits
ylim	Set or query y-axis limits
zlim	Set or query z-axis limits

Deep Learning

activations	Compute deep learning network layer activations
classify	Classify data using trained deep learning neural network
predict	Reconstruct the inputs using trained autoencoder
predictAndUpdateState	Predict responses using a trained recurrent neural network and update the network state

To display a list of supported functions, at the MATLAB Command Window, enter:

```
methods(half(1))
```

Examples

Convert Value to Half Precision

To cast a double-precision number to half precision, use the `half` function.

```
a = half(pi)
```

```
a =

  half

    3.1406
```

You can also use the `half` function to cast an existing variable to half-precision.

```
v = single(magic(3))
```

```
v = 3x3 single matrix

     8     1     6
     3     5     7
     4     9     2
```

```
a = half(v)
```

```
a =

  3x3 half matrix
```

```
     8     1     6
     3     5     7
     4     9     2
```

Limitations

- Arithmetic operations which combine half-precision and logical types are not supported.
- For additional usage notes and limitations, see “Half Precision Code Generation Support”.

Version History

Introduced in R2018b

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

- For a list of functions that support code generation with half-precision inputs and any associated limitations, see “Half Precision Code Generation Support”.
- If your target hardware does not have native support for half-precision, then `half` is used as a storage type, with arithmetic operations performed in single-precision.
- Some functions use `half` only as a storage type and the arithmetic is performed in single-precision, regardless of the target hardware.
- For deep learning code generation, half inputs are cast to single precision and computations are performed in single precision.
- In MATLAB, the `isobject` function returns true with a half-precision input. In generated code, this function returns false.

GPU Code Generation

Generate CUDA® code for NVIDIA® GPUs using GPU Coder™.

- For a list of functions that support code generation with half-precision inputs and any associated limitations, see “Half Precision Code Generation Support”.
- CUDA compute capability of 5.3 or higher is required for generating and executing code with half-precision data types.
- CUDA toolkit version of 10.0 or later is required for generating and executing code with half-precision data types.
- You must set the memory allocation (`malloc`) mode to `'Discrete'` for generating CUDA code.

  For more information, see `coder.gpuConfig`.
- Half-precision complex data types are not supported for GPU code generation.
- If your target hardware does not have native support for half-precision, then `half` is used as a storage type, with arithmetic operations performed in single-precision.
- Some functions use `half` only as a storage type and the arithmetic is performed in single-precision, regardless of the target hardware.

- For deep learning code generation, half inputs are cast to single precision and computations are performed in single precision. To perform computations in half, set the library target to `'tensorrt'` and set the data type to `'FP16'` in `coder.DeepLearningConfig`.
- In MATLAB, the `isobject` function returns true with a half-precision input. In generated code, this function returns false.

See Also

`single` | `double`

Topics

"Half Precision Code Generation Support"
"Floating-Point Numbers" (Fixed-Point Designer)
"What is Half Precision?" (Fixed-Point Designer)
Edge Detection with Sobel Method in Half-Precision

packNGo

Package generated code in ZIP file for relocation

Syntax

```
packNGo(buildInfo,Name,Value)
```

Description

`packNGo(buildInfo,Name,Value)` packages the code files in a compressed ZIP file so that you can relocate, unpack, and rebuild them in another development environment. The list of name-value pairs is optional.

The ZIP file can include these types of files:

- Source files (for example, `.c`, `.cu`, and `.cpp` files)
- Header files (for example, `.h`, `.cuh`, and `.hpp` files)
- MAT-file that contains the build information object (`.mat` file)
- Nonbuild-related files (for example, `.dll` files and `.txt` informational files) required for a final executable
- Build-generated binary files (for example, executable `.exe` file or dynamic link library `.dll`).

  The code generator includes the build-generated binary files (if present) in the ZIP file. The **ignoreFileMissing** property does not apply to build-generated binary files.
- CMake configuration files (`CMakeLists.txt`) that you use to generate makefiles or projects for a compiler environment.

Use this function to relocate files. You can then recompile the files for a specific target environment or rebuild them in a development environment in which MATLAB is not installed. By default, the function packages the files as a flat folder structure in a ZIP file within the code generation folder. You can customize the output by specifying name-value pairs. After relocating the ZIP file, use a standard ZIP utility to unpack the compressed file.

Because the code generated by using GPU Coder relies on third-party compilers and libraries to build and run the executables, the development environment that you are relocating to must also satisfy these requirements. For more information, see “Installing Prerequisite Products” and “Setting Up the Prerequisite Products”.

The `packNGo` function can potentially modify the build information passed in the first `packNGo` argument. As part of code packaging, `packNGo` can find additional files from source and include paths recorded in the build information. When these files are found, `packNGo` adds them to the build information.

To ensure that `packNGo` finds header files, add their paths to `buildInfo` by using the `addIncludePaths` function.

Note When generating standalone code by using the `codegen` command, you can use the `-package` option to both generate code and package the code in a ZIP file in a single step.

Examples

Run packNGo from Command Window

After the build process is complete, you can run packNGo from the Command Window. Use `packNGo` for ZIP file packaging of generated code in the file `portzingbit.zip`. Maintain the relative file hierarchy.

1. Change folders to the code generation folder. For example, using MATLAB Coder, `codegen/dll/zingbit` or for Simulink® code generation, `zingbit_grt_rtw`.
2. Load the `buildInfo` object that describes the build.
3. Run `packNGo` with property settings for `packType` and `fileName`.

```
cd codegen/dll/zingbit;
load buildInfo.mat
packNGo(buildInfo,'packType', 'hierarchical', ...
   'fileName','portzingbit');
```

Configure packNGo in the Simulink Editor

If you configure ZIP file packaging from the code generation pane, the code generator uses `packNGo` to output a ZIP file during the build process.

1. Select **Code Generation** > **Package code and artifacts**. Optionally, provide a **Zip file name**. To apply the changes, click **OK**.
2. Build the model. At the end of the build process, the code generator outputs the ZIP file. The folder structure in the ZIP file is hierarchical.

Configure packNGo for Simulink from the Command Line

You can configure ZIP file packaging by using the `set_param` function. During the build process, the code generator uses `packNGo` to create a ZIP file.

To configure ZIP file packaging for the model `zingbit`, run this command:

```
set_param('zingbit', 'PackageGeneratedCodeAndArtifacts', 'on');
```

The `packNGo` function creates the file `zingbit.zip`.

To specify another name for the ZIP file, for example, `portzingbit.zip`, run these commands:

```
set_param('zingbit', 'PackageGeneratedCodeAndArtifacts', 'on');
set_param('zingbit', 'PackageName', 'portzingbit.zip')
```

Input Arguments

`buildInfo` — Object that provides build information
`buildInfo` object | path to `buildInfo.mat`

During the build process, the code generator places `buildInfo.mat` in the code generation folder. This MAT-file contains the `buildInfo` object. The object provides information that `packNGo` uses to produce the ZIP file.

You can specify the argument as a `buildInfo` object:

```
load buildInfo.mat
packNGo(buildInfo,'packType', 'hierarchical', ...
   'fileName','portzingbit');
```

Or, you can specify the argument as the path to the `buildInfo.mat` file:

```
buildInfoFile = fullfile(pathToBuildFolder, 'buildInfo.mat');
packNGo(buildInfoFile,'packType', 'hierarchical', ...
   'fileName','portzingbit');
```

Or, you can specify the argument as the path to the folder that contains `buildInfo.mat`:

```
packNGo(pathToBuildFolder,'packType', 'hierarchical', ...
   'fileName','portzingbit');
```

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `'packType','flat','nestedZipFiles',true`

`packType` — Specify structure type of ZIP file
`'flat'` (default) | `'hierarchical'`

If `'flat'`, package the generated code files in a ZIP file as a single, flat folder. The function does *not* package:

- Child `buildInfo.mat` files.
- `CMakeLists.txt` files.

If `'hierarchical'`, package the generated code files hierarchically in a primary ZIP file. The hierarchy contains top model, referenced model, and shared utility folders. The function also packages:

- The corresponding `buildInfo.mat` files for the folders.
- `CMakeLists.txt` files in the build folder.

Example: `'packType','flat'`

`nestedZipFiles` — Determines whether the paths for files in the secondary ZIP files are relative to the root folder of the primary ZIP file
`true` (default) | false

If `true`, create a primary ZIP file that contains three secondary ZIP files:

- `mlrFiles.zip` — Files in your *`matlabroot`* folder tree

- `sDirFiles.zip` — Files in and under your code generation folder
- `otherFiles.zip` — Required files not in the *`matlabroot`* or `start` folder trees

If `false`, create a primary ZIP file that contains folders, for example, your code generation folder and *`matlabroot`*.

Example: `'nestedZipFiles',true`

`fileName` — Specifies a file name for the primary ZIP file
`'modelOrFunctionName``.zip'` (default) | *`'myName'`*

If you do not specify the `'fileName'`-value pair, the function packages the files in a ZIP file named *`modelOrFunctionName`*`.zip` and places the ZIP file in the code generation folder.

If you specify `'fileName'` with the value, *`'myName'`*, the function creates *`myName`*`.zip` in the code generation folder.

To specify another location for the primary ZIP file, provide the absolute path to the location, *`fullPath/myName`*`.zip`

Example: `'fileName','/home/user/myModel.zip'`

`minimalHeaders` — Selects whether to include only the minimal header files
`true` (default) | `false`

If `true`, include only the minimal header files required to build the code in the ZIP file.

If `false`, include header files found on the include path in the ZIP file.

GPU Coder requires that the `'minimalHeaders'` option is set to `false`.

Example: `'minimalHeaders',true`

`includeReport` — Selects whether to include the `html` folder for your code generation report
`false` (default) | `true`

If `false`, do not include the `html` folder in the ZIP file.

If `true`, include the `html` folder in the ZIP file.

Example: `'includeReport',false`

`ignoreParseError` — Instruct packNGo not to terminate on parse errors
false (default) | true

If `false`, terminate on parse errors.

If `true`, do not terminate on parse errors.

Example: `'ignoreParseError',false`

`ignoreFileMissing` — Instruct packNGo not to terminate if files are missing
false (default) | true

If `false`, terminate on missing file errors.

If `true`, do not terminate on missing files errors.

Example: `'ignoreFileMissing',false`

Limitations

- The function operates on source files only, such as `*.c`, `*.cpp`, `cuh`, and `*.h` files. The function does not support compile flags, defines, or makefiles.
- The function does not package source files for reusable library subsystems.
- Unnecessary files might be included. The function might find additional files from source paths and include paths recorded in the build information, even if those files are not used.
- For GPU Coder, the function does not package example main source and header files that you generate with the default configuration settings. To package the example main files, configure code generation to generate and compile the example main function, generate your code, and then package the build files.

Version History

Introduced in R2006b

See Also

Topics

"Relocate Generated Code to Another Development Environment"

gpucoder.batchedMatrixMultiply

Optimized GPU implementation of batched matrix multiply operation

Syntax

```
[D1,D2] = gpucoder.batchedMatrixMultiply(A1,B1,A2,B2)
[D1,...,DN] = gpucoder.batchedMatrixMultiply(A1,B1,...,AN,BN)
___ = gpucoder.batchedMatrixMultiply( ___ ,Name,Value)
```

Description

`[D1,D2] = gpucoder.batchedMatrixMultiply(A1,B1,A2,B2)` performs matrix-matrix multiplication of a batch of matrices `A1,B1` and `A2,B2`. The `gpucoder.batchedMatrixMultiply` function performs matrix-matrix multiplication of the form:

$$D = \alpha AB$$

where α is a scalar multiplication factor, `A`, `B`, and `D` are matrices with dimensions `m`-by-`k`, `k`-by-`n`, and `m`-by-`n` respectively. You can optionally transpose or hermitian-conjugate `A` and `B`. By default, α is set to one and the matrices are not transposed. To specify a different scalar multiplication factor and perform transpose operations on the input matrices, use the `Name,Value` pair arguments.

All the batches passed to the `gpucoder.batchedMatrixMultiply` function must be uniform. That is, all instances must have the same dimensions `m,n,k`.

`[D1,...,DN] = gpucoder.batchedMatrixMultiply(A1,B1,...,AN,BN)` performs matrix-matrix multiplication of multiple `A`, `B` pairs of the form:

$$D_i = \alpha A_i B_i \qquad i = 1 \ldots N$$

`___ = gpucoder.batchedMatrixMultiply( ___ ,Name,Value)` performs batched matrix multiply operation by using the options specified by one or more `Name,Value` pair arguments.

Examples

Batched Matrix-Matrix Multiplication

Perform a simple batched matrix-matrix multiplication and use the `gpucoder.batchedMatrixMultiply` function to generate CUDA code that calls appropriate `cublas<t>gemmBatched` APIs.

In one file, write an entry-point function `myBatchMatMul` that accepts matrix inputs `A1`, `B1`, `A2`, and `B2`. Because the input matrices are not transposed, use the `'nn'` option.

```
function [D1,D2] = myBatchMatMul(A1,B1,A2,B2,alpha)

[D1,D2] = gpucoder.batchedMatrixMultiply(A1,B1,A2,B2, ...
    'alpha',alpha,'transpose','nn');

end
```

To create a type for a matrix of doubles for use in code generation, use the `coder.newtype` function.

```
A1 = coder.newtype('double',[15,42],[0 0]);
A2 = coder.newtype('double',[15,42],[0 0]);
B1 = coder.newtype('double',[42,30],[0 0]);
B2 = coder.newtype('double',[42,30],[0 0]);
alpha = 0.3;
inputs = {A1,B1,A2,B2,alpha};
```

To generate a CUDA library, use the `codegen` function.

```
cfg = coder.gpuConfig('lib');
cfg.GpuConfig.EnableCUBLAS = true;
cfg.GpuConfig.EnableCUSOLVER = true;
cfg.GenerateReport = true;
codegen -config cfg-args inputs myBatchMatMul
```

The generated CUDA code contains kernels `myBatchMatMul_kernelNN` for initializing the input and output matrices. The code also contains the `cublasDgemmBatched` API calls to the cuBLAS library. The following code is a snippet of the generated code.

```
//
// File: myBatchMatMul.cu
//
...
void myBatchMatMul(const double A1[630], const double B1[1260], const double A2
                   [630], const double B2[1260], double alpha, double D1[450],
                   double D2[450])
{
  double alpha1;
...

  myBatchMatMul_kernel1<<<dim3(2U, 1U, 1U), dim3(512U, 1U, 1U)>>>(*gpu_A2,
    *gpu_A1, *gpu_input_cell_f2, *gpu_input_cell_f1);
  cudaMemcpy(gpu_B2, (void *)&B2[0], 10080UL, cudaMemcpyHostToDevice);
  cudaMemcpy(gpu_B1, (void *)&B1[0], 10080UL, cudaMemcpyHostToDevice);
  myBatchMatMul_kernel2<<<dim3(3U, 1U, 1U), dim3(512U, 1U, 1U)>>>(*gpu_B2,
    *gpu_B1, *gpu_input_cell_f4, *gpu_input_cell_f3);
  myBatchMatMul_kernel3<<<dim3(1U, 1U, 1U), dim3(480U, 1U, 1U)>>>(gpu_r3, gpu_r2);
  myBatchMatMul_kernel4<<<dim3(1U, 1U, 1U), dim3(32U, 1U, 1U)>>>(gpu_r2,
    *gpu_out_cell);
  myBatchMatMul_kernel5<<<dim3(1U, 1U, 1U), dim3(32U, 1U, 1U)>>>(gpu_r3,
    *gpu_out_cell);
...

  cublasDgemmBatched(getCublasGlobalHandle(), CUBLAS_OP_N, CUBLAS_OP_N, 15, 30,
                     42, (double *)gpu_alpha1, (double **)gpu_Aarray, 15,
                     (double **)gpu_Barray, 42, (double *)gpu_beta1, (double **)
                     gpu_Carray, 15, 2);
  myBatchMatMul_kernel6<<<dim3(1U, 1U, 1U), dim3(480U, 1U, 1U)>>>(*gpu_D2,
    *gpu_out_cell, *gpu_D1);
...
}
```

Input Arguments

A, B — Operands

vectors | matrices

Operands, specified as vectors or matrices. `A` and `B` must be 2-D arrays. The number of columns in `A` must be equal to the number of rows in `B`.

Data Types: `double` | `single` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `[D1,D2] = gpucoder.batchedMatrixMultiply(A1,B1,A2,B2,'alpha',0.3,'transpose','CC');`

`alpha` — Scalar multiplication factor
1.0 (default) | scalar

Value of the scalar used for multiplication with `A`. Default value is one.

`transpose` — Operation performed on input matrices
'NN' (default) | character vector | string

Character vector or string composed of two characters, indicating the operation performed on the matrices `A` and `B` prior to matrix multiplication. Possible values are normal (`'N'`), transposed (`'T'`), or complex conjugate transpose (`'C'`).

Output Arguments

`D` — Product
scalar | vector | matrix

Product, returned as a scalar, vector, or matrix. Array `D` has the same number of rows as input `A` and the same number of columns as input `B`.

Version History

Introduced in R2020a

See Also

Apps
GPU Coder

Functions
`codegen` | `coder.gpu.kernel` | `coder.gpu.kernelfun` | `gpucoder.stridedMatrixMultiply` | `gpucoder.batchedMatrixMultiplyAdd` | `gpucoder.stridedMatrixMultiplyAdd`

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

Topics
"GPU Programming Paradigm"
"Code Generation by Using the GPU Coder App"
"Code Generation Using the Command Line Interface"
"Kernels from Element-Wise Loops"

“Kernels from Scatter-Gather Type Operations”
“Kernels from Library Calls”
“Design Patterns”

gpucoder.batchedMatrixMultiplyAdd

Optimized GPU implementation of batched matrix multiply with add operation

Syntax

```
[D1,D2] = gpucoder.batchedMatrixMultiplyAdd(A1,B1,C1,A2,B2,C2)
[D1,...,DN] = gpucoder.batchedMatrixMultiplyAdd(A1,B1,C1,...,AN,BN,CN)
___ = gpucoder.batchedMatrixMultiplyAdd( ___ ,Name,Value)
```

Description

`[D1,D2] = gpucoder.batchedMatrixMultiplyAdd(A1,B1,C1,A2,B2,C2)` performs matrix-matrix multiplication and add of a batch of matrices `A1,B1,C1` and `A2,B2,C2`. The `gpucoder.batchedMatrixMultiplyAdd` function performs matrix-matrix multiplication of the form:

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

where α and β are scalar multiplication factors and `A`, `B`, `C`, and `D` are matrices with dimensions `m`-by-`k`, `k`-by-`n`, `m`-by-`n`, and `m`-by-`n` respectively.`A` and `B` can optionally be transposed or hermitian-conjugated. By default, α and β are set to one and the matrices are not transposed. To specify a different scalar multiplication factor and perform transpose operations on the input matrices, use the `Name,Value` pair arguments.

All the batches passed to the `gpucoder.batchedMatrixMultiplyAdd` function must be uniform. That is, all instances must have the same dimensions `m,n,k`.

`[D1,...,DN] = gpucoder.batchedMatrixMultiplyAdd(A1,B1,C1,...,AN,BN,CN)` performs matrix-matrix multiplication and add of multiple `A`, `B`, `C` matrices of the form:

$$D_i = \alpha A_i B_i + \beta C_i \qquad i = 1 \ldots N$$

`___ = gpucoder.batchedMatrixMultiplyAdd( ___ ,Name,Value)` performs batched matrix multiply and add operation by using the options specified by one or more `Name,Value` pair arguments.

Examples

Batched Matrix-Matrix Multiplication with Add

Perform a simple batched matrix-matrix multiplication with add and use the `gpucoder.batchedMatrixMultiplyAdd` function to generate CUDA code that calls appropriate `cublas<t>gemmBatched` APIs.

In one file, write an entry-point function `myBatchMatMulAdd` that accepts matrix inputs `A1`, `B1`, `C1`, `A2`, `B2`, and `C2`. Because the input matrices are not transposed, use the `'nn'` option.

```
function [D1,D2] = myBatchMatMulAdd(A1,B1,C1,A2,B2,C2,alpha,beta)
```

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

end
```

To create a type for a matrix of doubles for use in code generation, use the `coder.newtype` function.

```
A1 = coder.newtype('double',[12,14],[0 0]);
A2 = coder.newtype('double',[12,14],[0 0]);
B1 = coder.newtype('double',[14,16],[0 0]);
B2 = coder.newtype('double',[14,16],[0 0]);
C1 = coder.newtype('double',[12,16],[0 0]);
C2 = coder.newtype('double',[12,16],[0 0]);
alpha = 0.3;
beta = 0.6;
inputs = {A1,B1,C1,A2,B2,C2,alpha,beta};
```

To generate a CUDA library, use the `codegen` function.

```
cfg = coder.gpuConfig('lib');
cfg.GpuConfig.EnableCUBLAS = true;
cfg.GpuConfig.EnableCUSOLVER = true;
cfg.GenerateReport = true;
codegen -config cfg-args inputs myBatchMatMulAdd
```

The generated CUDA code contains kernels `myBatchMatMulAdd_kernelNN` for initializing the input and output matrices. The code also contains the `cublasDgemmBatched` API calls to the cuBLAS library. The following code is a snippet of the generated code.

```
//
// File: myBatchMatMulAdd.cu
//
...
void myBatchMatMulAdd(const double A1[168], const double B1[224], const double
                      C1[192], const double A2[168], const double B2[224], const
                      double C2[192], double alpha, double beta, double D1[192],
                      double D2[192])
{
  double alpha1;
...

  myBatchMatMulAdd_kernel2<<<dim3(1U, 1U, 1U), dim3(224U, 1U, 1U)>>>(*gpu_B2,
    *gpu_B1, *gpu_input_cell_f4, *gpu_input_cell_f3);
  cudaMemcpy(gpu_C2, (void *)&C2[0], 1536UL, cudaMemcpyHostToDevice);
  cudaMemcpy(gpu_C1, (void *)&C1[0], 1536UL, cudaMemcpyHostToDevice);
  myBatchMatMulAdd_kernel3<<<dim3(1U, 1U, 1U), dim3(192U, 1U, 1U)>>>(*gpu_C2,
    *gpu_C1, gpu_r3, gpu_r2);
  myBatchMatMulAdd_kernel4<<<dim3(1U, 1U, 1U), dim3(32U, 1U, 1U)>>>(gpu_r2,
    *gpu_out_cell);
  myBatchMatMulAdd_kernel5<<<dim3(1U, 1U, 1U), dim3(32U, 1U, 1U)>>>(gpu_r3,
    *gpu_out_cell);
...

  cublasDgemmBatched(getCublasGlobalHandle(), CUBLAS_OP_N, CUBLAS_OP_N, 12, 16,
                     14, (double *)gpu_alpha1, (double **)gpu_Aarray, 12,
                     (double **)gpu_Barray, 14, (double *)gpu_beta1, (double **)
                     gpu_Carray, 12, 2);
  myBatchMatMulAdd_kernel6<<<dim3(1U, 1U, 1U), dim3(192U, 1U, 1U)>>>(*gpu_D2,
...

}
```

Input Arguments

A, B, C — Operands

vectors | matrices

Operands, specified as vectors or matrices. A, B, and C must be 2-D arrays. The number of columns in A must be equal to the number of rows in B. The number of rows in A must be equal to the number of rows in C. The number of columns in B must be equal to the number of columns in C.

Data Types: double | single | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64

Name-Value Pair Arguments

Specify optional pairs of arguments as Name1=Value1,...,NameN=ValueN, where Name is the argument name and Value is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose Name *in quotes.*

Example: [D1,D2] = gpucoder.batchedMatrixMultiplyAdd(A1,B1,C1,A2,B2,C2,'alpha',0.3,'beta',0.6,'transpose','CC');

alpha — Scalar multiplication factor
1.0 (default) | scalar

Value of the scalar used for multiplication with A. Default value is one.

beta — Scalar multiplication factor
1.0 (default) | scalar

Value of the scalar used for multiplication with C. Default value is one.

transpose — Operation performed on input matrices
'NN' (default) | character vector | string

Character vector or string composed of two characters, indicating the operation performed on the matrices A and B prior to matrix multiplication. Possible values are normal ('N'), transposed ('T'), or complex conjugate transpose ('C').

Output Arguments

D — Product
scalar | vector | matrix

Product, returned as a scalar, vector, or matrix. Array D has the same number of rows as input A and the same number of columns as input B.

Version History

Introduced in R2020a

See Also

Apps
GPU Coder

Functions
`codegen` | `coder.gpu.kernel` | `coder.gpu.kernelfun` | `gpucoder.stridedMatrixMultiply` | `gpucoder.stridedMatrixMultiplyAdd` | `gpucoder.batchedMatrixMultiply`

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

Topics
"GPU Programming Paradigm"
"Code Generation by Using the GPU Coder App"
"Code Generation Using the Command Line Interface"
"Kernels from Element-Wise Loops"
"Kernels from Scatter-Gather Type Operations"
"Kernels from Library Calls"
"Design Patterns"

gpucoder.stridedMatrixMultiply

Optimized GPU implementation of strided and batched matrix multiply operation

Syntax

```
D = gpucoder.stridedMatrixMultiply(A,B)
___ = gpucoder.stridedMatrixMultiply( ___ ,Name,Value)
```

Description

`D = gpucoder.stridedMatrixMultiply(A,B)` performs strided matrix-matrix multiplication of a batch of matrices. The input matrices `A` and `B` for each instance of the batch are located at fixed address offsets from their addresses in the previous instance. The `gpucoder.stridedMatrixMultiply` function performs matrix-matrix multiplication of the form:

$$D = \alpha AB$$

where α is a scalar multiplication factor, `A`, `B`, and `D` are matrices with dimensions `m`-by-`k`, `k`-by-`n`, and `m`-by-`n` respectively. You can optionally transpose or hermitian-conjugate `A` and `B`. By default, α is set to one and the matrices are not transposed. To specify a different scalar multiplication factor and perform transpose operations on the input matrices, use the `Name,Value` pair arguments.

All the batches passed to the `gpucoder.stridedMatrixMultiply` function must be uniform. That is, all instances must have the same dimensions `m,n,k`.

`___ = gpucoder.stridedMatrixMultiply( ___ ,Name,Value)` performs strided batched matrix multiply operation by using the options specified by one or more `Name,Value` pair arguments.

Examples

Strided Batched Matrix-Matrix Multiplication

Perform a simple batched matrix-matrix multiplication and use the `gpucoder.stridedMatrixMultiply` function to generate CUDA code that calls appropriate `cublas<t>gemmStridedBatched` APIs.

In one file, write an entry-point function `myStridedMatMul` that accepts matrix inputs `A` and `B`. Because the input matrices are not transposed, use the `'nn'` option.

```
function [D] = myStridedMatMul(A,B,alpha)

[D] = gpucoder.stridedMatrixMultiply(A,B,'alpha',alpha, ...
    'transpose','nn');

end
```

To create a type for a matrix of doubles for use in code generation, use the `coder.newtype` function.

```
A = coder.newtype('double',[5 4 100],[0 0]);
B = coder.newtype('double',[4 5 100],[0 0]);
```

```
alpha = 0.3;
inputs = {A,B,alpha};
```

To generate a CUDA library, use the `codegen` function.

```
cfg = coder.gpuConfig('lib');
cfg.GpuConfig.EnableCUBLAS = true;
cfg.GpuConfig.EnableCUSOLVER = true;
cfg.GenerateReport = true;
codegen -config cfg-args inputs myStridedMatMul
```

The generated CUDA code contains kernels `myStridedMatMul_kernelNN` for initializing the input and output matrices. The code also contains the `cublasDgemmStridedBatched` API calls to the cuBLAS library. The following code is a snippet of the generated code.

```
//
// File: myStridedMatMul.cu
//
...
void myStridedMatMul(const double A_data[], const int A_size[3], const double
                     B_data[], const int B_size[3], double alpha, double D_data[],
                     int D_size[3])
{
  double alpha1;
...
  beta1 = 0.0;
  cudaMemcpy(gpu_alpha1, &alpha1, 8ULL, cudaMemcpyHostToDevice);
  cudaMemcpy(gpu_A_data, (void *)A_data, A_size[0] * A_size[1] * A_size[2] *
             sizeof(double), cudaMemcpyHostToDevice);
  cudaMemcpy(gpu_B_data, (void *)B_data, B_size[0] * B_size[1] * B_size[2] *
             sizeof(double), cudaMemcpyHostToDevice);
  cudaMemcpy(gpu_beta1, &beta1, 8ULL, cudaMemcpyHostToDevice);
  if (D_data_dirtyOnCpu) {
    cudaMemcpy(gpu_D_data, &D_data[0], 25 * D_size[2] * sizeof(double),
               cudaMemcpyHostToDevice);
  }

  if (batchDimsA[2] >= batchDimsB[2]) {
    if (batchDimsA[2] >= 1) {
      ntilecols = batchDimsA[2];
    } else {
      ntilecols = 1;
    }
  } else {
    ntilecols = batchDimsB[2];
  }

  cublasDgemmStridedBatched(getCublasGlobalHandle(), CUBLAS_OP_N, CUBLAS_OP_N, 5,
    5, 4, (double *)gpu_alpha1, (double *)&gpu_A_data[0], 5, strideA, (double *)
    &gpu_B_data[0], 4, strideB, (double *)gpu_beta1, (double *)&gpu_D_data[0], 5,
    25, ntilecols);
  cudaMemcpy(&D_data[0], gpu_D_data, 25 * D_size[2] * sizeof(double),
             cudaMemcpyDeviceToHost);
...
}
```

Input Arguments

A, B — Operands

vectors | matrices

Operands, specified as vectors or matrices. `gpucoder.stridedMatrixMultiply` multiplies along the first two dimensions.

Data Types: `double` | `single` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `D = gpucoder.stridedMatrixMultiply(A,B,'alpha',0.3,'transpose','CC');`

`alpha` — Scalar multiplication factor
1.0 (default) | scalar

Value of the scalar used for multiplication with `A`. Default value is one.

`transpose` — Operation performed on input matrices
'NN' (default) | character vector | string

Character vector or string composed of two characters, indicating the operation performed on the matrices `A` and `B` prior to matrix multiplication. Possible values are normal (`'N'`), transposed (`'T'`), or complex conjugate transpose (`'C'`).

Output Arguments

`D` — Product
scalar | vector | matrix

Product, returned as a scalar, vector, or matrix. Array `D` has the same number of rows as input `A` and the same number of columns as input `B`.

Version History

Introduced in R2020a

See Also

Apps
GPU Coder

Functions
`codegen` | `coder.gpu.kernel` | `coder.gpu.kernelfun` | `gpucoder.stridedMatrixMultiplyAdd` | `gpucoder.batchedMatrixMultiply` | `gpucoder.batchedMatrixMultiplyAdd`

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

Topics
"GPU Programming Paradigm"
"Code Generation by Using the GPU Coder App"
"Code Generation Using the Command Line Interface"
"Kernels from Element-Wise Loops"

“Kernels from Scatter-Gather Type Operations”
“Kernels from Library Calls”
“Design Patterns”

gpucoder.stridedMatrixMultiplyAdd

Optimized GPU implementation of strided, batched matrix multiply with add operation

Syntax

```
D = gpucoder.stridedMatrixMultiplyAdd(A,B,C)
___ = gpucoder.stridedMatrixMultiplyAdd( ___ ,Name,Value)
```

Description

`D = gpucoder.stridedMatrixMultiplyAdd(A,B,C)` performs strided matrix-matrix multiplication and add of a batch of matrices. The input matrices `A`, `B`, and `C` for each instance of the batch are located at fixed address offsets from their addresses in the previous instance. The `gpucoder.stridedMatrixMultiplyAdd` function performs matrix-matrix multiplication of the form:

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

where α and β are scalar multiplication factors, `A`, `B`, `C`, and `D` are matrices with dimensions `m`-by-`k`, `k`-by-`n`, `m`-by-`n`, and `m`-by-`n` respectively. `A` and `B` can optionally be transposed or hermitian-conjugated. By default, α and β are set to one and the matrices are not transposed. To specify a different scalar multiplication factor and perform transpose operations on the input matrices, use the `Name,Value` pair arguments.

All the batches passed to the `gpucoder.stridedMatrixMultiplyAdd` function must be uniform. That is, all instances must have the same dimensions `m,n,k`.

`___ = gpucoder.stridedMatrixMultiplyAdd( ___ ,Name,Value)` performs batched matrix multiply and add operation by using the options specified by one or more `Name,Value` pair arguments.

Examples

Strided Matrix-Matrix Multiplication with Add

Performs a simple batched matrix-matrix multiplication with add and use the `gpucoder.stridedMatrixMultiplyAdd` function to generate CUDA code that calls appropriate `cublas<t>gemmStridedBatched` APIs.

In one file, write an entry-point function `myStridedMatMulAdd` that accepts matrix inputs `A`, `B`, and `C`. Because the input matrices are not transposed, use the `'nn'` option.

```
function [D] = myStridedMatMulAdd(A,B,C,alpha,beta)

[D] = gpucoder.stridedMatrixMultiplyAdd(A,B,C,'alpha',alpha,...
    'beta',beta,'transpose','nn');

end
```

To create a type for a matrix of doubles for use in code generation, use the `coder.newtype` function.

```
A = coder.newtype('double',[12,14 10],[0 0]);
B = coder.newtype('double',[14,16 10],[0 0]);
C = coder.newtype('double',[12,16 10],[0 0]);
alpha = 0.3;
beta = 0.6;
inputs = {A,B,C,alpha,beta};
```

To generate a CUDA library, use the `codegen` function.

```
cfg = coder.gpuConfig('lib');
cfg.GpuConfig.EnableCUBLAS = true;
cfg.GpuConfig.EnableCUSOLVER = true;
cfg.GenerateReport = true;
codegen -config cfg-args inputs myStridedMatMulAdd
```

The generated CUDA code contains kernels `myStridedMatMulAdd_kernelNN` for initializing the input and output matrices. The code also contains the `cublasDgemmStridedBatched` API calls to the cuBLAS library. The following code is a snippet of the generated code.

```
//
// File: myStridedMatMulAdd.cu
...

void myStridedMatMulAdd(const double A[1680], const double B[2240], const double
  C[1920], double alpha, double beta, double D[1920])
{
  double alpha1;

..alpha1 = alpha;
  beta1 = beta;
  cudaMemcpy(gpu_C, (void *)&C[0], 15360ULL, cudaMemcpyHostToDevice);
  myStridedMatMulAdd_kernel1<<<dim3(4U, 1U, 1U), dim3(512U, 1U, 1U)>>>(*gpu_C,
    *gpu_D);
  cudaMemcpy(gpu_alpha1, &alpha1, 8ULL, cudaMemcpyHostToDevice);
  cudaMemcpy(gpu_A, (void *)&A[0], 13440ULL, cudaMemcpyHostToDevice);
  cudaMemcpy(gpu_B, (void *)&B[0], 17920ULL, cudaMemcpyHostToDevice);
  cudaMemcpy(gpu_beta1, &beta1, 8ULL, cudaMemcpyHostToDevice);
  cublasDgemmStridedBatched(getCublasGlobalHandle(), CUBLAS_OP_N, CUBLAS_OP_N,
    12, 16, 14, (double *)gpu_alpha1, (double *)&(*gpu_A)[0], 12, 168, (double *)
    &(*gpu_B)[0], 14, 224, (double *)gpu_beta1, (double *)&(*gpu_D)[0], 12, 192,
    10);
  cudaMemcpy(&D[0], gpu_D, 15360ULL, cudaMemcpyDeviceToHost);
...
}
```

Input Arguments

A, B, C — Operands

vectors | matrices

Operands, specified as vectors or matrices. The number of columns in `A` must be equal to the number of rows in `B`. The number of rows in `A` must be equal to the number of rows in `C`. The number of columns in `B` must be equal to the number of columns in `C`.

Data Types: `double` | `single` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `D = gpucoder.stridedMatrixMultiplyAdd(A,B,C,'alpha',0.3,'beta',0.6,'transpose','CC');`

`alpha` — Scalar multiplication factor
1.0 (default) | scalar

Value of the scalar used for multiplication with `A`. Default value is one.

`beta` — Scalar multiplication factor
1.0 (default) | scalar

Value of the scalar used for multiplication with `C`. Default value is one.

`transpose` — Operation performed on input matrices
'NN' (default) | character vector | string

Character vector or string composed of two characters, indicating the operation performed on the matrices `A` and `B` prior to matrix multiplication. Possible values are normal (`'N'`), transposed (`'T'`), or complex conjugate transpose (`'C'`).

Output Arguments

`D` — Product
scalar | vector | matrix

Product, returned as a scalar, vector, or matrix. Array `D` has the same number of rows as input `A` and the same number of columns as input `B`.

Version History

Introduced in R2020a

See Also

Apps
GPU Coder

Functions
`codegen` | `coder.gpu.kernel` | `coder.gpu.kernelfun` | `gpucoder.stridedMatrixMultiply` | `gpucoder.batchedMatrixMultiply` | `gpucoder.batchedMatrixMultiplyAdd`

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

Topics
“GPU Programming Paradigm”
“Code Generation by Using the GPU Coder App”
“Code Generation Using the Command Line Interface”
“Kernels from Element-Wise Loops”
“Kernels from Scatter-Gather Type Operations”
“Kernels from Library Calls”

“Design Patterns”

coder.gpu.persistentMemory

Pragma to allocate a variable as persistent memory on the GPU

Syntax

```
coder.gpu.persistentMemory(pvar)
```

Description

`coder.gpu.persistentMemory(pvar)` maps the persistent MATLAB variable `pvar` to the CUDA enabled NVIDIA GPU as variable with persistent memory. The variable must be fixed size and of a data type supported for GPU code generation.

This is a code generation function. It has no effect in MATLAB.

Note For CUDA MEX, the persistent memory on the GPU is retained during the entirety of the MATLAB session. For freeing the GPU memory, use the `clear mex` MATLAB command. For freeing the GPU memory in static libraries, dynamic libraries, or executable targets, call the generated `<primary function name>_terminate()` housekeeping function.

Examples

Using `coder.gpu.persistentMemory` to allocate a variable on the GPU memory as persistent

This example shows how to map a persistent MATLAB variable to the persistent memory on the GPU by using the `coder.gpu.persistentMemory` pragma.

Consider the following MATLAB entry-point function `myPersistent` containing a simple loop.

```
function output = myPersistent(input)

    persistent pvar;
    if isempty(pvar)
        pvar = ones(size(input));
    end

    coder.gpu.persistentMemory(pvar);

    for i = 1:numel(input)
        pvar(i) = pvar(i) + input(i);
    end

    output = coder.nullcopy(input);
    for i = 1:numel(input)
        output(i) = pvar(i) * input(i);
    end
end
```

Create a code generation configuration object for a standalone CUDA static library.

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

Define a cell array `input` that declares the size and data type of the inputs to the function `myPersistent`.

```
input = {rand(1,1024)}
```

Generate a MEX function `myPersistent_mex` by using `codegen` function with the `-config`, `-args`, and `-report` options to specify configuration, provide input arguments, and generate a code generation report.

```
codegen -config cfg -args input -report myPersistent
```

A snippet of the generated `myPersistent.cu` file is shown.

```
//
// File: myPersistent.cu
//
// GPU Coder version                    : 2.0
// CUDA/C/C++ source code generated on  : 16-Jul-2020 20:08:46
//

// Include Files
#include "myPersistent.h"
#include "myPersistent_data.h"
#include "myPersistent_initialize.h"
#include "MWCudaDimUtility.hpp"
...
//
// Arguments    : const double input[1024]
//                double output[1024]
// Return Type  : void
//
void myPersistent(const double input[1024], double output[1024])
{
  double (*gpu_input)[1024];
  double (*gpu_output)[1024];
  if (!isInitialized_myPersistent) {
    myPersistent_initialize();
  }

  cudaMalloc(&gpu_output, 8192UL);
  cudaMalloc(&gpu_input, 8192UL);
  cudaMemcpy(gpu_input, (void *)&input[0], 8192UL, cudaMemcpyHostToDevice);
  myPersistent_kernel1<<<dim3(2U, 1U, 1U), dim3(512U, 1U, 1U)>>>(*gpu_input,
    *gpu_output, *pvar);
  cudaMemcpy(&output[0], gpu_output, 8192UL, cudaMemcpyDeviceToHost);
  cudaFree(*gpu_input);
  cudaFree(*gpu_output);
}

//
// Arguments    : void
// Return Type  : void
//
void myPersistent_init()
{
  double b_pvar[1024];
  boolean_T pvar_dirtyOnCpu;
  boolean_T pvar_dirtyOnGpu;
  pvar_dirtyOnCpu = false;
  pvar_dirtyOnGpu = true;
  for (int i = 0; i < 1024; i++) {
    if (pvar_dirtyOnGpu) {
      cudaMemcpy(&b_pvar[0], pvar, 8192UL, cudaMemcpyDeviceToHost);
      pvar_dirtyOnGpu = false;
    }

    b_pvar[i] = 1.0;
    pvar_dirtyOnCpu = true;
  }

  if (pvar_dirtyOnCpu) {
```

```
    cudaMemcpy(pvar, &b_pvar[0], 8192UL, cudaMemcpyHostToDevice);
  }
}
...
```

The persistent variable `pvar` is retained on the GPU between calls to the `myPersistent()` function.

Input Arguments

`pvar` — Name of the persistent variable
name

The name of the variable that must be mapped to the GPU memory space as a persistent variable.

Limitations

- MATLAB classes are not supported by the `coder.gpu.persistentMemory` pragma.
- The `coder.gpu.persistentMemory` pragma is not supported in Simulink.
- If the persistent variable in MATLAB is variable-sized, the code generator does not map the variable to a persistent memory on the GPU.

Version History

Introduced in R2020b

See Also

Apps
GPU Coder

Functions
`codegen` | `coder.gpu.kernel` | `coder.gpu.kernelfun` | `gpucoder.stencilKernel` | `coder.gpu.nokernel` | `coder.gpu.constantMemory`

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

Topics
"GPU Programming Paradigm"
"Kernels from Element-Wise Loops"
"Kernels from Scatter-Gather Type Operations"
"Design Patterns"
"Kernels from Library Calls"

GPU Acceleration Parameters

- “Model Configuration Parameters: GPU Acceleration” on page 3-2
- “Simulation Target: GPU Acceleration Tab Overview” on page 3-3
- “Custom compute capability” on page 3-4
- “Dynamic memory allocation threshold” on page 3-5
- “Stack size per GPU thread” on page 3-6
- “Include error checks in generated code” on page 3-7
- “Additional compiler flags” on page 3-8

Model Configuration Parameters: GPU Acceleration

The **Simulation Target > GPU Acceleration** category includes parameters for configuring GPU-specific settings of the generated code.

These parameters require a GPU Coder license.

On the Configuration Parameters dialog box, the following configuration parameters are on the **Simulation Target > GPU Acceleration** pane.

Parameter	Description
“Custom compute capability” (Simulink Coder)	Specify the name of the NVIDIA virtual GPU architecture for code generation.
“Dynamic memory allocation threshold” (Simulink Coder)	Specify the size above which the private variables are allocated on the heap instead of the stack.
“Stack size per GPU thread” (Simulink Coder)	Specify the maximum stack limit per GPU thread.
“Include error checks in generated code” (Simulink Coder)	Add run-time error-checking functionality to the generated CUDA code.
“Additional compiler flags” (Simulink Coder)	Specify additional flags to the `nvcc` compiler.

See Also

More About

- “Model Configuration Parameters: Simulation Target” (Simulink)
- “Simulation Acceleration by Using GPU Coder”

Simulation Target: GPU Acceleration Tab Overview

Set up GPU-specific information about simulation acceleration for a model's active configuration set, including device memory settings, code profiling and analysis, and device architecture and compiler.

These parameters require a GPU Coder license.

See Also

More About

- “Model Configuration Parameters: GPU Acceleration” (Simulink Coder)
- “Simulation Acceleration by Using GPU Coder”

Custom compute capability

Description

Specify the name of the NVIDIA virtual GPU architecture for code generation.

Category: Simulation Target > GPU Acceleration

Settings

Default: `''`

Specify the name of the NVIDIA virtual GPU architecture for which the CUDA input files must be compiled.

For example, to specify a virtual architecture type `-arch=compute_50`. You can specify a real architecture using `-arch=sm_50`. For more information, see the *Options for Steering GPU Code Generation* topic in the CUDA Toolkit documentation.

Dependencies

- This parameter requires a GPU Coder license.
- This parameter is enabled by **GPU acceleration** on the **Simulation Target** pane.

Command-Line Information

Parameter: `SimGPUCustomComputeCapability`
Type: character vector
Value: `''` or a valid `user-specified` virtual architecture specification
Default: `''`

See Also

Related Examples

- “Model Configuration Parameters: GPU Acceleration” (Simulink Coder)
- “Simulation Acceleration by Using GPU Coder”

Dynamic memory allocation threshold

Description

Specify the memory allocation threshold.

Category: Simulation Target > GPU Acceleration

Settings

Default: 200

Specify the size above which the private variables are allocated on the heap instead of the stack, as an integer value.

Dependencies

- This parameter requires a GPU Coder license.
- This parameter is enabled by **GPU acceleration** on the **Simulation Target** pane.

Command-Line Information

Parameter: `SimGPUMallocThreshold`
Type: integer
Value: any valid value
Default: `200`

See Also

Related Examples

- “Model Configuration Parameters: GPU Acceleration” (Simulink Coder)
- “Simulation Acceleration by Using GPU Coder”

Stack size per GPU thread

Description

Specify the stack limit per GPU thread.

Category: Simulation Target > GPU Acceleration

Settings

Default: 1024

Specify the maximum stack limit per GPU thread as an integer value.

Dependencies

- This parameter requires a GPU Coder license.
- This parameter is enabled by **GPU acceleration** on the **Simulation Target** pane.

Command-Line Information

Command-Line Information
Parameter: `SimGPUStackLimitPerThread`
Type: integer
Value: any valid value
Default: `1024`

See Also

Related Examples

- “Model Configuration Parameters: GPU Acceleration” (Simulink Coder)
- “Simulation Acceleration by Using GPU Coder”

Include error checks in generated code

Description

Add run-time error-checking functionality to the generated CUDA code.

Category: Simulation Target > GPU Acceleration

Settings

Default: Off

☑ **On**

Generates code with error-checking for CUDA API and kernel calls and performs run-time checks.

☐ **Off**

The generated CUDA code does not contain error-checking functionality.

Dependencies

- This parameter requires a GPU Coder license.
- This parameter is enabled by **GPU acceleration** on the **Simulation Target** pane.

Command-Line Information

Parameter: `SimGPUErrorChecks`
Type: character vector
Value: `'on'` | `'off'`
Default: `'off'`

See Also

Related Examples

- “Model Configuration Parameters: GPU Acceleration” (Simulink Coder)
- “Simulation Acceleration by Using GPU Coder”

Additional compiler flags

Description

Specify additional flags to the NVIDIA `nvcc` compiler.

Category: Simulation Target > GPU Acceleration

Settings

Default: `''`

Pass additional flags to the GPU compiler. For example, `--fmad=false` instructs the `nvcc` compiler to disable contraction of floating-point multiply and add to a single Floating-Point Multiply-Add (FMAD) instruction.

For similar NVIDIA compiler options, see the topic on *NVCC Command Options* in the CUDA Toolkit documentation.

Dependencies

- This parameter requires a GPU Coder license.
- This parameter is enabled by **GPU acceleration** on the **Simulation Target** pane.

Command-Line Information

Parameter: `SimGPUCompilerFlags`
Type: character vector
Value: `''` or a valid `user-specified` flag
Default: `''`

See Also

Related Examples

- “Model Configuration Parameters: GPU Acceleration” (Simulink Coder)
- “Simulation Acceleration by Using GPU Coder”

Code Generation Parameters: GPU Code

Model Configuration Parameters: GPU Code

The **Code Generation** > **GPU Code** category includes parameters for configuring GPU-specific settings of the generated code.

These parameters require a GPU Coder license.

On the Configuration Parameters dialog box, the following configuration parameters are on the **Code Generation** > **GPU Code** pane.

Parameter	Description
“GPU index” (Simulink Coder)	Specify the CUDA device to target.
“Compute capability” (Simulink Coder)	Specify the minimum compute capability for which CUDA code is generated.
“Custom compute capability” (Simulink Coder)	Specify the name of the NVIDIA virtual GPU architecture for code generation.
“Memory mode” (Simulink Coder)	Specify the Memory allocation (`malloc`) mode to be used in the generated CUDA code.
“Maximum blocks per kernel” (Simulink Coder)	Specify the maximum number of CUDA blocks created during a kernel launch.
“Dynamic memory allocation threshold” (Simulink Coder)	Specify the size above which the private variables are allocated on the heap instead of the stack.
“Memory Manager” (Simulink Coder)	Select the GPU memory manager for efficient memory allocation, management, and improving run-time performance.
“Stack size per GPU thread” (Simulink Coder)	Specify the maximum stack limit per GPU thread.
“Include error checks in generated code” (Simulink Coder)	Add run-time error-checking functionality to the generated CUDA code.
“Kernel name prefix” (Simulink Coder)	Specify custom kernel name prefixes.
“Additional compiler flags” (Simulink Coder)	Specify additional flags to the `nvcc` compiler.
“cuBLAS” (Simulink Coder)	Replacement of math function calls with NVIDIA `cuBLAS` library calls.
“cuSOLVER” (Simulink Coder)	Replacement of math function calls with NVIDIA `cuSOLVER` library calls.
“cuFFT” (Simulink Coder)	Replacement of math function calls with NVIDIA `cuFFT` library calls.

See Also

More About

- “Model Configuration” (Simulink Coder)
- “Code Generation from Simulink Models with GPU Coder”

Code Generation: GPU Code Tab Overview

Set up GPU-specific information about code generation for a model's active configuration set, including device memory settings, CUDA libraries, code profiling and analysis, and device architecture and compiler.

These parameters require a GPU Coder license.

See Also

More About

- “Model Configuration Parameters: GPU Code” (Simulink Coder)
- “Code Generation from Simulink Models with GPU Coder”

GPU index

Description

Specify the CUDA device to target.

Category: Code Generation > GPU Code

Settings

Default: -1

In a multi GPU environment such as NVIDIA Drive platforms, specify the CUDA device to target.

Note GPU index can be used with `gpuArray` only if `gpuDevice` and GPU index point to the same GPU. If `gpuDevice` points to a different GPU, a `CUDA_ERROR_INVALID_VALUE` runtime error is thrown.

Dependencies

- This parameter requires a GPU Coder license.
- To enable this parameter, select **Generate GPU code** on the **Code Generation** pane.

Command-Line Information

Parameter: `GPUDeviceID`
Type: integer
Value: `-1` or a valid `user-specified` index value
Default: `-1`

See Also

Related Examples

- “Model Configuration Parameters: GPU Code” (Simulink Coder)
- “Code Generation from Simulink Models with GPU Coder”

Compute capability

Description

Specify the minimum compute capability of the GPU device for which CUDA code is generated.

Category: Code Generation > GPU Code

Settings

Default: `3.5`

Select the minimum compute capability for code generation. The compute capability identifies the features supported by the GPU hardware. It is used by applications at run time to determine which hardware features, instructions are available on the GPU device. If you specify custom compute capability, GPU Coder ignores this setting.

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

Target	Compute Capability
CUDA MEX	See “GPU Computing Requirements”.
Source code, static or dynamic library, and executables	3.2 or higher.
Deep learning applications in 8-bit integer precision	6.1, 6.3 or higher.
Deep learning applications in half-precision (16-bit floating point)	5.3, 6.0, 6.2 or higher.

Dependencies

- This parameter requires a GPU Coder license.
- To enable this parameter, select **Generate GPU code** on the **Code Generation** pane.

Command-Line Information

Parameter: `GPUComputeCapability`
Type: character vector
Value: `'3.2'|'3.5'|'3.7'|'5.0'|'5.2'|'5.3'|'6.0'|'6.1'|'6.2'|'7.0'|'7.1'|'7.2'|'7.5'|`
Default: `'3.5'`

See Also

Related Examples

- “Model Configuration Parameters: GPU Code” (Simulink Coder)
- “Code Generation from Simulink Models with GPU Coder”

Custom compute capability

Description

Specify the name of the NVIDIA virtual GPU architecture for code generation.

Category: Code Generation > GPU Code

Settings

Default: `''`

Specify the name of the NVIDIA virtual GPU architecture for which the CUDA input files must be compiled.

For example, to specify a virtual architecture type `-arch=compute_50`. You can specify a real architecture using `-arch=sm_50`. For more information, see the *Options for Steering GPU Code Generation* topic in the CUDA Toolkit documentation.

Dependencies

- This parameter requires a GPU Coder license.
- To enable this parameter, select **Generate GPU code** on the **Code Generation** pane.

Command-Line Information

Parameter: `GPUCustomComputeCapability`
Type: character vector
Value: `''` or a valid `user-specified` virtual architecture specification
Default: `''`

See Also

Related Examples

- “Model Configuration Parameters: GPU Code” (Simulink Coder)
- “Code Generation from Simulink Models with GPU Coder”

Memory mode

Description

Specify the memory allocation (`malloc`) mode to use in the generated CUDA code.

Category: Code Generation > GPU Code

Settings

Default: `discrete`

`discrete`

The generated code uses the `cudaMalloc` API for transferring data between the CPU and the GPU. From the programmers point-of-view, the discrete mode has a traditional memory architecture with separate CPU and GPU global memory address space.

`unified`

The generated code uses the `cudaMallocManaged` API that uses a shared (unified) CPU and GPU global memory address space.

For NVIDIA embedded targets only. See “Deprecating support for unified memory allocation mode on host” on page 4-7.

Dependencies

- This parameter requires a GPU Coder license.
- To enable this parameter, select **Generate GPU code** on the **Code Generation** pane.

Command-Line Information

Parameter: `GPUMallocMode`
Type: character vector
Value: `'discrete'` | `'unified'`
Default: `'discrete'`

Compatibility Considerations

Deprecating support for unified memory allocation mode on host

In a future release, support for the unified memory allocation (`cudaMallocManaged`) mode will be removed when targeting NVIDIA GPU devices on the host development computer. When targeting GPU devices on the host, select `'discrete'` for the **Memory mode** parameter.

You can continue to use unified memory allocation mode when targeting NVIDIA embedded platforms.

See Also

Related Examples

- “Model Configuration Parameters: GPU Code” (Simulink Coder)
- “Code Generation from Simulink Models with GPU Coder”

Maximum blocks per kernel

Description

Specify the maximum number of CUDA blocks created during a kernel launch.

Because GPU devices have limited streaming multiprocessor (SM) resources, limiting the number of blocks for each kernel can avoid performance losses from scheduling, loading and unloading of blocks.

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

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

Category: Code Generation > GPU Code

Settings

Default: 0

Specify the maximum number of CUDA blocks created during a kernel launch.

Dependencies

- This parameter requires a GPU Coder license.
- To enable this parameter, select **Generate GPU code** on the **Code Generation** pane.

Command-Line Information

Parameter: `GPUMaximumBlocksPerKernel`
Type: integer
Value: any valid value
Default: `0`

See Also

Related Examples

- “Model Configuration Parameters: GPU Code” (Simulink Coder)
- “Code Generation from Simulink Models with GPU Coder”

Dynamic memory allocation threshold

Description

Specify the memory allocation threshold.

Category: Code Generation > GPU Code

Settings

Default: 200

Specify the size above which the private variables are allocated on the heap instead of the stack, as an integer value.

Dependencies

- This parameter requires a GPU Coder license.
- To enable this parameter, select **Generate GPU code** on the **Code Generation** pane.

Command-Line Information

Parameter: `GPUMallocThreshold`
Type: integer
Value: any valid value
Default: `200`

See Also

Related Examples

- “Model Configuration Parameters: GPU Code” (Simulink Coder)
- “Code Generation from Simulink Models with GPU Coder”

Memory Manager

Description

Select the GPU memory manager for efficient memory allocation, management, and improving run-time performance.

Category: Code Generation > GPU Code

Settings

Default: Off

☑ **On**

The GPU memory manager creates a collection of large GPU memory pools and manages allocation and deallocation of chunks of memory blocks within these pools. By creating large memory pools, the memory manager reduces the number of calls to the CUDA memory APIs, improving run-time performance. You can use the GPU memory manager for MEX and standalone CUDA code generation.

☐ **Off**

Disable the use GPU memory manager for memory allocation and management.

Dependencies

- This parameter requires a GPU Coder license.
- To enable this parameter, select **Generate GPU code** on the **Code Generation** pane.

Command-Line Information

Parameter: `GPUEnableMemoryManager`
Type: character vector
Value: `'on'` | `'off'`
Default: `'off'`

See Also

Related Examples

- “Model Configuration Parameters: GPU Code” (Simulink Coder)
- “Code Generation from Simulink Models with GPU Coder”

Stack size per GPU thread

Description

Specify the stack limit per GPU thread.

Category: Code Generation > GPU Code

Settings

Default: 1024

Specify the maximum stack limit per GPU thread as an integer value.

Dependencies

- This parameter requires a GPU Coder license.
- To enable this parameter, select **Generate GPU code** on the **Code Generation** pane.

Command-Line Information

Command-Line Information
Parameter: `GPUStackLimitPerThread`
Type: integer
Value: any valid value
Default: `1024`

See Also

Related Examples

- “Model Configuration Parameters: GPU Code” (Simulink Coder)
- “Code Generation from Simulink Models with GPU Coder”

Include error checks in generated code

Description

Add run-time error-checking functionality to the generated CUDA code.

Category: Code Generation > GPU Code

Settings

Default: Off

☑ **On**

Generates code with error-checking for CUDA API and kernel calls and performs run-time checks.

☐ **Off**

The generated CUDA code does not contain error-checking functionality.

Dependencies

- This parameter requires a GPU Coder license.
- To enable this parameter, select **Generate GPU code** on the **Code Generation** pane.

Command-Line Information

Parameter: `GPUErrorChecks`
Type: character vector
Value: `'on'` | `'off'`
Default: `'off'`

See Also

Related Examples

- “Model Configuration Parameters: GPU Code” (Simulink Coder)
- “Code Generation from Simulink Models with GPU Coder”

Kernel name prefix

Description

Specify custom kernel name prefixes.

Category: Code Generation > GPU Code

Settings

Default: `''`

Specify a custom name prefix for all the kernels in the generated code. For example, using the value `'CUDA_'` creates kernels with names `CUDA_kernel1`, `CUDA_kernel2`, and so on. If no name is provided, GPU Coder prepends the kernel name with the name of the entry-point function. Kernel names can contain upper-case letters, lowercase letters, digits 0–9, and underscore character _. GPU Coder removes unsupported characters from the kernel names and appends `alpha` to prefixes that do not begin with an alphabetic letter.

Dependencies

- This parameter requires a GPU Coder license.
- To enable this parameter, select **Generate GPU code** on the **Code Generation** pane.

Command-Line Information

Parameter: `GpuKernelNamePrefix`
Type: character vector
Value: `''` or a valid `user-specified` name
Default: `''`

See Also

Related Examples

- “Model Configuration Parameters: GPU Code” (Simulink Coder)
- “Code Generation from Simulink Models with GPU Coder”

Additional compiler flags

Description

Specify additional flags to the NVIDIA `nvcc` compiler.

Category: Code Generation > GPU Code

Settings

Default: `''`

Pass additional flags to the GPU compiler. For example, `--fmad=false` instructs the `nvcc` compiler to disable contraction of floating-point multiply and add to a single Floating-Point Multiply-Add (FMAD) instruction.

For similar NVIDIA compiler options, see the topic on *NVCC Command Options* in the CUDA Toolkit documentation.

Dependencies

- This parameter requires a GPU Coder license.
- To enable this parameter, select **Generate GPU code** on the **Code Generation** pane.

Command-Line Information

Parameter: `GPUCompilerFlags`
Type: character vector
Value: `''` or a valid `user-specified` flag
Default: `''`

See Also

Related Examples

- “Model Configuration Parameters: GPU Code” (Simulink Coder)
- “Code Generation from Simulink Models with GPU Coder”

cuBLAS

Description

Replacement of math function calls with NVIDIA `cuBLAS` library calls.

Category: Code Generation > GPU Code

Settings

Default: On

☑ **On**

Allows GPU Coder to replace appropriate math function calls with calls to the `cuBLAS` library. For functions that have no replacements in CUDA, GPU Coder uses portable MATLAB functions and attempts to map them to the GPU.

☐ **Off**

Disable the use of the `cuBLAS` library in the generated code.

Dependencies

- This parameter requires a GPU Coder license.
- To enable this parameter, select **Generate GPU code** on the **Code Generation** pane.

Command-Line Information

Parameter: `GPUcuBLAS`
Type: character vector
Value: `'on'` | `'off'`
Default: `'on'`

See Also

Related Examples

- “Model Configuration Parameters: GPU Code” (Simulink Coder)
- “Code Generation from Simulink Models with GPU Coder”

cuSOLVER

Description

Replacement of math function calls with NVIDIA `cuSOLVER` library calls.

Category: Code Generation > GPU Code

Settings

Default: On

☑ **On**

Allows GPU Coder to replace appropriate math function calls with calls to the `cuSOLVER` library. For functions that have no replacements in CUDA, GPU Coder uses portable MATLAB functions and attempts to map them to the GPU.

☐ **Off**

Disable the use of the `cuSOLVER` library in the generated code.

Dependencies

- This parameter requires a GPU Coder license.
- To enable this parameter, select **Generate GPU code** on the **Code Generation** pane.

Command-Line Information

Parameter: `GPUcuSOLVER`
Type: character vector
Value: `'on'` | `'off'`
Default: `'on'`

See Also

Related Examples

- “Model Configuration Parameters: GPU Code” (Simulink Coder)
- “Code Generation from Simulink Models with GPU Coder”

cuFFT

Description

Replacement of math function calls with NVIDIA `cuFFT` library calls.

Category: Code Generation > GPU Code

Settings

Default: On

☑ **On**

Allows GPU Coder to replace appropriate `fft` calls with calls to the `cuFFT` library.

☐ **Off**

Disables use of the `cuFFT` library in the generated code. With this option, GPU Coder uses C `FFTW` libraries where available or generates kernels from portable MATLAB `fft` code.

Dependencies

- This parameter requires a GPU Coder license.
- To enable this parameter, select **Generate GPU code** on the **Code Generation** pane.

Command-Line Information

Parameter: `GPUcuFFT`
Type: character vector
Value: `'on'` | `'off'`
Default: `'on'`

See Also

Related Examples

- “Model Configuration Parameters: GPU Code” (Simulink Coder)
- “Code Generation from Simulink Models with GPU Coder”