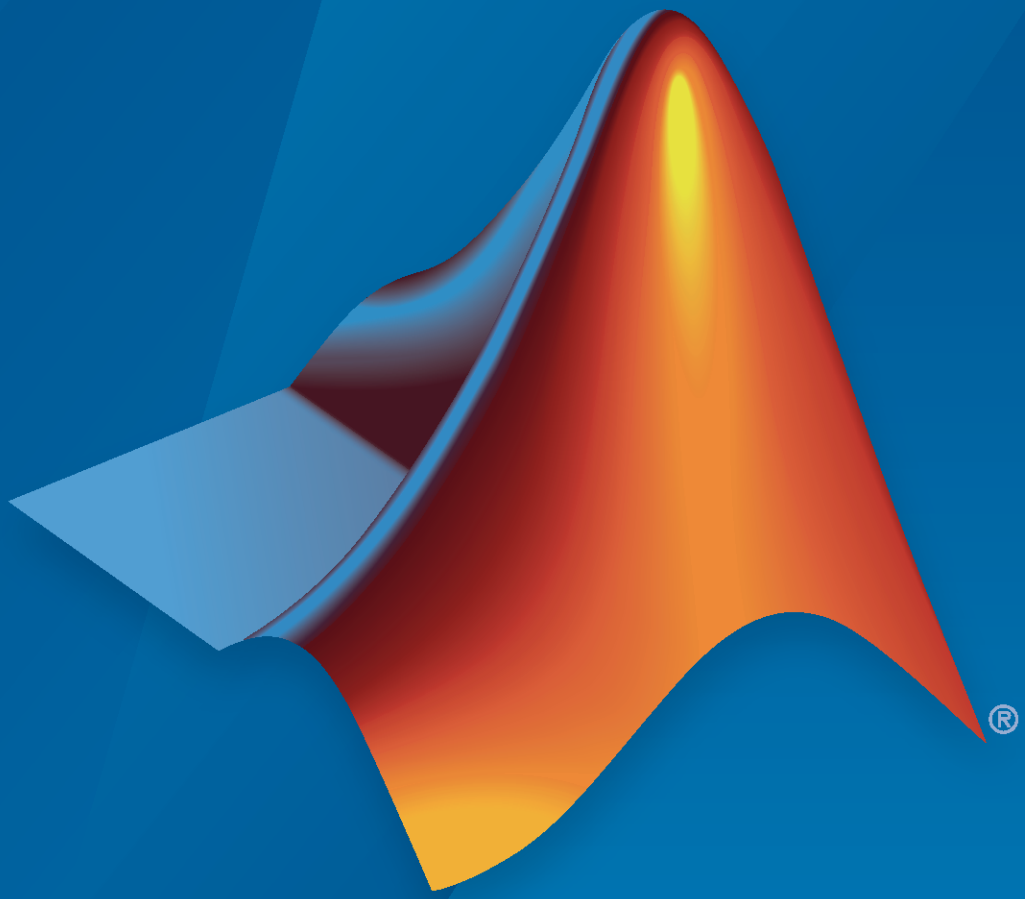
# GPU Coder™

Reference



MATLAB®

R2020a

MathWorks®

# How to Contact MathWorks

Latest news: `www.mathworks.com`

Sales and services: `www.mathworks.com/sales_and_services`

User community: `www.mathworks.com/matlabcentral`

Technical support: `www.mathworks.com/support/contact_us`

Phone: 508-647-7000

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

*GPU Coder™ Reference*



#### Trademarks

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

#### Patents

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

#### Revision History

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

# Contents

# 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`.

## Examples

- “Code Generation by Using the GPU Coder App”

### Programmatic Use

`gpucoder`

## See Also

**Apps**
**MATLAB Coder**

**Functions**
codegen

**Topics**
“Code Generation by Using the GPU Coder App”

**Introduced in R2017b**

# GPU Environment Check

Verify and set up the 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 NVIDIA compilers and libraries necessary 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. This test validates code execution by comparing the results with 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 this test, you must install the GPU Coder Support Package for NVIDIA GPUs.
- Specify the location of the libraries using the app and generate 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”.

## Open the GPU Environment Check App

- MATLAB Command Window: Enter `gpucoderSetup`.

## Examples

- “Code Generation by Using the GPU Coder App”

### See Also

**Apps**
**GPU Coder**

**Functions**
`codegen` | `coder.checkGpuInstall`

**Topics**
“Code Generation by Using the GPU Coder App”

**Introduced in R2019a**

# Functions

# 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**.

## 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.

## See Also

`codegen` | `coder.gpuEnvConfig` | `getenv` | `gpuDevice` | `gpuDeviceCount` | `setenv`

**Topics**

“Verify Setup”

**Introduced in R2017b**

# coder.gpuConfig

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

## Description

The `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 C/C++ code**
`'MEX'` | `'LIB'` | `'DLL'` | `'EXE'`

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

| Value | Description |
|---|---|
| `'MEX'` | CUDA MEX |
| `'LIB'` | Static library |
| `'DLL'` | Dynamically linked library |
| `'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.<br><br>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.<br><br>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 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.<br><br>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.<br><br>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.<br><br>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.<br><br>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.<br><br>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.

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`

**`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 code configuration object to true.

## See Also

`codegen` | `coder.CodeConfig` | `coder.EmbeddedCodeConfig` | `coder.MexCodeConfig`

### Introduced in R2017b

# 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.

## See Also

`GPU Coder` | `codegen`

### Topics

“Code Generation by Using the GPU Coder App”

### Introduced in R2017b

# 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
```

## See Also

codegen | coder.gpu.kernelfun | coder.gpuConfig

### Topics

“Kernels from Element-Wise Loops”
“Kernels from Scatter-Gather Type Operations”

### Introduced in R2017b

# 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(size(x)));

    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.

## See Also

`codegen` | `coder.gpu.kernel` | `coder.gpuConfig`

**Topics**

“Kernels from Element-Wise Loops”
“Kernels from Scatter-Gather Type Operations”

**Introduced in R2017b**

# 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.

## See Also

`coder.gpu.kernel` | `coder.gpu.kernelfun`

### Topics

“GPU Programming Paradigm”
“GPU Memory Allocation and Minimization”

### Introduced in R2017b

# gpucoder.stencilKernel

Create CUDA code for stencil functions

## 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..

## See Also

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

**Topics**

“Design Patterns”
“Kernels from Element-Wise Loops”
“Kernels from Scatter-Gather Type Operations”
“Kernels from Library Calls”

**Introduced in R2017b**

# 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 and column from matrix `A` and one row and 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.

## See Also

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

### Topics

“Design Patterns”
“Kernels from Element-Wise Loops”
“Kernels from Scatter-Gather Type Operations”
“Kernels from Library Calls”

### Introduced in R2017b

# cnncodegen

Generate code and build static library for Series or DAG Network

## Syntax

```
cnncodegen(net,'targetlib',libraryname)
cnncodegen(net,'targetlib',libraryname,Name,Value)
```

## Description

`cnncodegen(net,'targetlib',libraryname)` generates CUDA C++ code and builds a static library for the specified network object and target library by using default values for all properties.

`cnncodegen(net,'targetlib',libraryname,Name,Value)` generates CUDA C++ code and builds a static library for the specified network object and target library with additional code generation options specified by one or more `Name,Value` pair arguments.

## 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® 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'`. For `'arm-compute'`, you must provide the `'ArmArchitecture'` parameter.

```
cnncodegen(net,'targetlib','arm-compute'...
,'targetparams',struct('ArmComputeVersion','19.02','ArmArchitecture','armv8'));
```

### Generate Code for the YOLO Network to Run on NVIDIA GPU

Generate CUDA C++ code from a `SeriesNetwork` object created for the YOLO architecture, trained for classifying the PASCAL dataset. This example requires the GPU Coder product and GPU Coder Interface for Deep Learning Libraries.

Get the pretrained YOLO network and convert it into a `SeriesNetwork` object.

```
url = 'https://www.mathworks.com/supportfiles/gpucoder/cnn_models/Yolo/yolonet.mat';
websave('yolonet.mat',url);
net = coder.loadDeepLearningNetwork('yolonet.mat');
```

The `SeriesNetwork` object `net` contains 58 layers. These layers are convolution layers followed by leaky `ReLU` and fully connected layers at the end of the network architecture. You can use `net.Layers` to see the all the layers in this network.

Use the `cnncodegen` function to generate CUDA code.

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

The code generator generates the `.cu` and header files in the `'/pwd/codegen'` folder. The series network is generated as a C++ class called `CnnMain`, containing an array of 58 layer classes. 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 58 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. The files are compiled into the static library `cnnbuild.a` (on Linux®) or `cnnbuild.lib` (on Windows®).

## Input Arguments

### `net` — Name of the series or DAG network object

character vector | string scalar

Pretrained `SeriesNetwork` or `DAGNetwork` object.

### `libraryname` — Deep learning target library

character vector | string scalar

The target library and the target platform to generate code for, specified as one of the values in this table.

| Value | Description |
|---|---|
| `'arm-compute'` | Target an ARM CPU processor supporting `NEON` instructions by using the ARM Compute Library for computer vision and machine learning.<br><br>Requires the MATLAB Coder™ Interface for Deep Learning Libraries. |
| `'arm-compute-mali'` | Target an ARM GPU processor by using the ARM Compute Library for computer vision and machine learning.<br><br>Requires the GPU Coder product and the GPU Coder Interface for Deep Learning Libraries. |
| `'cudnn'` | Target NVIDIA GPUs by using the CUDA Deep Neural Network library (cuDNN).<br><br>Requires the GPU Coder product and the GPU Coder Interface for Deep Learning Libraries. |

| Value | Description |
|---|---|
| `'mkldnn'` | Target Intel® CPU processor by using the Intel Math Kernel Library for Deep Neural Networks (MKL-DNN).<br><br>Requires the MATLAB Coder Interface for Deep Learning Libraries. |
| `'tensorrt'` | Target NVIDIA GPUs by using NVIDIA TensorRT, a high performance deep learning inference optimizer and run-time library.<br><br>Requires the GPU Coder product and the GPU Coder Interface for Deep Learning Libraries. |

### Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `cnncodegen(net,'targetlib','mkldnn','codegenonly',0,'batchsize',1)` generates C++ code for the Intel processor by using MKL-DNN and builds a static library for the network object in `net`.

#### General Options

**`batchsize` — Size**
1 (default) | positive integer

A positive nonzero integer value specifying the number of observations to operate on in a single call to the network `predict()` method. When calling `network->predict()`, the size of the input data must match the `batchsize` value specified during `cnncodegen`.

If `libraryname` is `'arm-compute'` or `'arm-compute-mali'`, the value of `batchsize` must be `1`.

**`codegenonly` — Option to generate only code**
0 (default) | 1

Boolean flag that, when enabled, generates CUDA C++ code without generating and building a makefile.

**`targetparams` — Library-specific parameters**
structure

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

**Parameters for ARM Compute Library (CPU)**

| Field | Description |
|---|---|
| `ArmComputeVersion` | Version of ARM Compute Library on the target hardware, specified as `'18.05'`, `'18.08'`, `'18.11'`, `'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'`. |
| `ArmArchitecture` | ARM architecture supported on the target hardware, specified as `'armv7` or `'armv8'`. The specified architecture must be the same as the architecture for the ARM Compute Library on the target hardware.<br><br>`ArmArchitecture` is a required parameter. |

**Parameters for ARM Compute Library (Mali GPU)**

| 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'`. |

**Parameters for NVIDIA cuDNN Library**

| Field | Description |
|---|---|
| `AutoTuning` | 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. Default value is `true`.<br><br>**Note** If `AutoTuning` is enabled for `TensorRT` targets, the software generates code with auto tuning disabled. It does so without generating any warning or error messages. |
| `DataType` | Specify the precision of the tensor data type input to the network. When performing inference in 32-bit floats, use `'FP32'`. For 8-bit integer, use `'INT8'`. Default value is `'FP32'`.<br><br>The `computecapability` argument must be set to `'6.1'` or higher if the `DataType` is set to `'INT8'`. Compute capability of 6.2 does not support `INT8` precision. |
| `CalibrationResultFile` | Location of the MAT-file containing the calibration data. Default value is `''`. This option is applicable only when `DataType` is set to `'INT8'`. |

**Parameters for NVIDIA TensorRT Library**

| Field | Description |
|---|---|
| `DataType` | Specify the precision of the tensor data type input to the network or the tensor output of a layer. When performing inference in 32-bit floats, use `'FP32'`. For 8-bit integer, use `'INT8'`. For half-precision, use `'FP16'` Default value is `'FP32'`.<br><br>The `computecapability` argument must be set to `'7.0'` or higher if the `DataType` is set to `'FP16'`.<br><br>The `computecapability` argument must be set to `'6.1'` or higher if the `DataType` is set to `'INT8'`. Compute capability of 6.2 does not support `INT8` precision. |
| `DataPath` | Location of the image dataset used during recalibration. Default value is `''`. This option is applicable only when `DataType` is set to `'INT8'`.<br><br>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` | 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. Default value is 50. This option is applicable only when `DataType` is set to `'INT8'`.<br><br>NVIDIA recommends that about 500 images are sufficient for calibrating. Refer to the TensorRT documentation for more information. |

**GPU Options (GPU Coder Only)**

### `computecapability` — Compute version

character vector | string scalar

*This property affects GPU targeting only.*

Character vector or string scalar specifying the NVIDIA GPU compute capability to compile for. Argument takes the format of `major#.minor#`.

Possible values are
`'3.2'|'3.5'|'3.7'|'5.0'|'5.2'|'5.3'|'6.0'|'6.1'|'6.2'|'7.0'|'7.1'|'7.2'`.

Default value is `'3.5'`.

## See Also

`codegen` | `coder.loadDeepLearningNetwork`

### Topics

“Code Generation for Deep Learning Networks with MKL-DNN” (MATLAB Coder)
“Deep Learning Prediction with ARM Compute Using cnncodegen” (MATLAB Coder)
“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 by Using NVIDIA TensorRT”

### Introduced in R2017b

# 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`, or `ssdObjectDetector` 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`, or `ssdObjectDetector` 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`, or `ssdObjectDetector` object. `functionname` must be the name of a function existing on the MATLAB path that returns a `SeriesNetwork`, `DAGNetwork`, `yolov2ObjectDetector`, or `ssdObjectDetector` 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.

## 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`, or `ssdObjectDetector` object.

Data Types: `string`

### `functionname` — MATLAB function name

name

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

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++.

Data Types: `char`

## Output Arguments

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

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

## Limitations

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

## See Also

`DAGNetwork` | `SeriesNetwork` | `cnncodegen` | `codegen` | `ssdObjectDetector` | `yolov2ObjectDetector`

**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”

**Introduced in R2017b**

# coder.DeepLearningConfig

Create deep learning code generation configuration objects

## Syntax

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

## Description

`deepLearningCfg = coder.DeepLearningConfig(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('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 |
|---|---|
| `'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). |
| `'cudnn'` | For generating code that uses the CUDA Deep Neural Network library (cuDNN).<br><br>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.<br><br>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 |
|---|---|
| `'arm-compute'` | Creates an `ARMNEONConfig` configuration object. |
| `'mkldnn'` | Creates an `MklDNNConfig` configuration object. |
| `'cudnn'` | Creates a `CuDNNConfig` configuration object. |
| `'tensorrt'` | Creates a `TensorRTConfig` configuration object. |

## See Also

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

**Topics**
“Code Generation for Deep Learning Networks with MKL-DNN” (MATLAB Coder)
“Code Generation for Deep Learning Networks with ARM Compute Library” (MATLAB Coder)
“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”

**Introduced in R2018b**

# 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

### Syntax

```
deepLearningCfg = coder.DeepLearningConfig('mkldnn')
```

#### Description

`deepLearningCfg = coder.DeepLearningConfig('mkldnn')` creates a `coder.MklDNNConfig` object for deep learning code generation by using the MKL-DNN library.

### 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.

## See Also

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

### Topics

“Code Generation for Deep Learning Networks with MKL-DNN” (MATLAB Coder)
“Code Generation for Deep Learning Networks with ARM Compute Library” (MATLAB Coder)
“Code Generation for Deep Learning Networks by Using cuDNN”
“Code Generation for Deep Learning Networks by Using TensorRT”

### Introduced in R2018b

# 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

### Syntax

```
deepLearningCfg = coder.DeepLearningConfig('cudnn')
```

#### Description

`deepLearningCfg = coder.DeepLearningConfig('cudnn')` creates a `coder.CuDNNConfig` object for deep learning code generation by using the cuDNN library.

### 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` — Tensor input precision**
`'fp32'` (default) | `'int8'`

Specify the precision of the tensor data type input to the network. 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.

**`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'`.

### `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.

## See Also

`codegen` | `coder.CodeConfig` | `coder.DeepLearningConfig` | `coder.EmbeddedCodeConfig` | `coder.TensorRTConfig` | `coder.gpuConfig` | `coder.gpuEnvConfig` | `coder.loadDeepLearningNetwork`

**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”

**Introduced in R2018b**

# 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

### Syntax

```
deepLearningCfg = coder.DeepLearningConfig('tensorrt')
```

#### Description

`deepLearningCfg = coder.DeepLearningConfig('tensorrt')` creates a `coder.TensorRTConfig` object for deep learning code generation by using the TensorRT library.

### Properties

**`DataType` — Tensor input precision**
`'fp32'` (default) | `'fp16'` | `'int8'`

Specify the precision of the tensor data type input to the network or the tensor output of a layer. 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. Compute capability of 6.2 does not support `INT8` precision. `FP16` precision requires a CUDA GPU with minimum compute capability of 7.0. Use the `ComputeCapability` property of the `GpuConfig` object to set the appropriate compute capability value.

See the “Deep Learning Prediction by Using NVIDIA TensorRT” 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.

## See Also

`codegen` | `coder.CodeConfig` | `coder.CuDNNConfig` | `coder.DeepLearningConfig` | `coder.EmbeddedCodeConfig` | `coder.gpuConfig` | `coder.gpuEnvConfig` | `coder.loadDeepLearningNetwork`

### Topics

“Deep Learning Prediction by Using NVIDIA TensorRT”
“Code Generation for Deep Learning Networks by Using TensorRT”

### Introduced in R2018b

# coder.getDeepLearningLayers

Get convolutional neural network layers supported for code generation for a specific deep learning library

## Syntax

```
coder.getDeepLearningLayers(libraryname)
```

## Description

`coder.getDeepLearningLayers(libraryname)` returns the convolutional neural network 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 `'arm-compute'` 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.

---

## 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('mkldnn')
```

```
ans =

  17×1 cell array

    {'AdditionLayer'                }
    {'AveragePooling2DLayer'        }
    {'BatchNormalizationLayer'      }
    {'ClassificationOutputLayer'    }
    {'ClippedReLULayer'             }
    {'Convolution2DLayer'           }
    {'CrossChannelNormalizationLayer'}
    {'DepthConcatenationLayer'      }
    {'DropoutLayer'                 }
    {'FullyConnectedLayer'          }
    {'ImageInputLayer'              }
    {'LeakyReLULayer'               }
    {'MaxPooling2DLayer'            }
    {'ReLULayer'                    }
    {'RegressionOutputLayer'        }
```

```
    {'SoftmaxLayer'                  }
    {'TransposedConvolution2DLayer'  }
```

## 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.<br><br>Requires the MATLAB Coder Interface for Deep Learning Libraries. |
| `'arm-compute-mali'` | ARM Compute Library for targeting ARM GPU processors.<br><br>Requires the GPU Coder product and the GPU Coder Interface for Deep Learning Libraries. |
| `'cudnn'` | cuDNN library.<br><br>Requires the GPU Coder product and the GPU Coder Interface for Deep Learning Libraries. |
| `'mkldnn'` | Intel Math Kernel Library for Deep Neural Networks.<br><br>Requires the MATLAB Coder Interface for Deep Learning Libraries. |
| `'tensorrt'` | TensorRT library.<br><br>Requires the GPU Coder product and the GPU Coder Interface for Deep Learning Libraries. |

## See Also

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

### Topics

“Installing Prerequisite Products”
“Prerequisites for Deep Learning with MATLAB Coder” (MATLAB Coder)
“Supported Networks and Layers”
“Networks and Layers Supported for C++ Code Generation” (MATLAB Coder)
“Code Generation for Deep Learning Networks with MKL-DNN” (MATLAB Coder)
“Code Generation for Deep Learning Networks with ARM Compute Library” (MATLAB Coder)
“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”

### Introduced in R2018b

# 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
```

## See Also

`GPU Coder` | `codegen`

### Introduced in R2017b

# 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);
}
```

## See Also

`codegen` | `coder.gpu.kernel` | `coder.gpu.kernelfun` | `coder.gpuConfig`

**Introduced in R2019a**

# 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.

## See Also

`codegen` | `coder.gpu.constantMemory` | `coder.gpu.kernel` | `coder.gpu.kernelfun` | `coder.gpu.nokernel` | `coder.gpuConfig` | `gpucoder.stencilKernel`

**Introduced in R2019a**

# 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.

## See Also

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

**Topics**
“Design Patterns”
“Kernels from Element-Wise Loops”
“Kernels from Scatter-Gather Type Operations”
“Kernels from Library Calls”

**Introduced in R2018b**

# gpucoder.profile

Create an execution profile report for the 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`. `codegen_inputs` specifies the inputs to the design file. The Embedded Coder product must be installed to generate the profiling report. Profiling is only supported on the Linux platform.

**Note** The profiling workflow depends on the `nvprof` tool from NVIDIA. In CUDA toolkit v10.1, NVIDIA restricts access to performance counters to only admin users. To enable GPU performance counters for all user accounts, see the instructions provided in https://developer.nvidia.com/nvidia-development-tools-solutions-ERR_NVGPUCTRPERM-permission-issue-performance-counters.

`gpucoder.profile( ___ ,Name,Value)` generates execution profiling report with one or more profiling options specified as a name-value pair

## Examples

### Execution Profiling Report for the Generated CUDA Code

This example shows you how to perform fine grain analysis for a MATLAB algorithm and its generated CUDA code through software-in-the-loop (SIL) execution profiling. The Embedded Coder product must be installed to generate the execution profiling report.

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

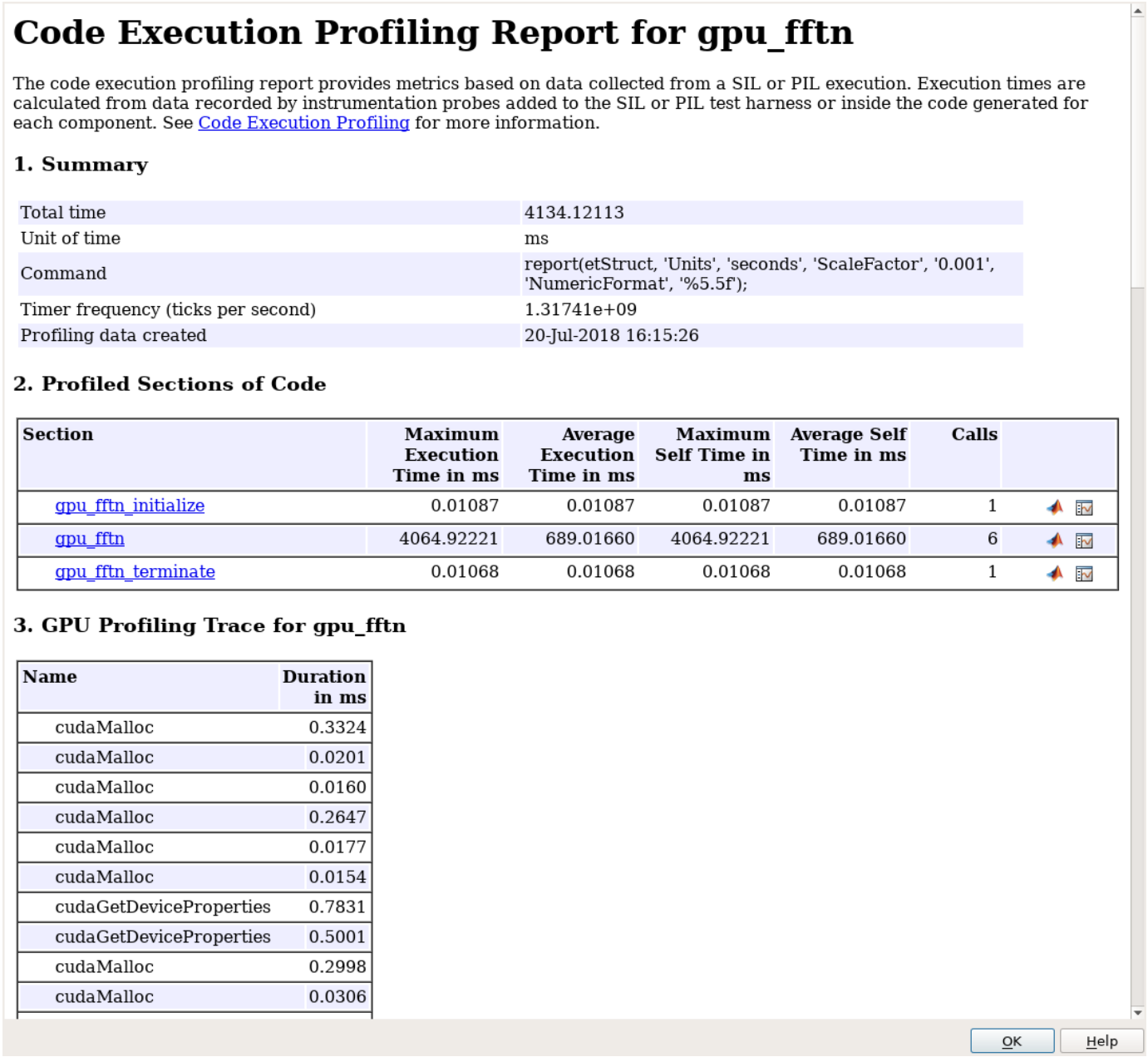
```
function [Y] = gpu_fftn(X)
  coder.gpu.kernelfun();
  Y = fftn(X);
end
```

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

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

The code execution profiling report opens. This report provides metrics based on data collected from a SIL execution. Execution times are calculated from data recorded by instrumentation probes added

to the SIL test harness or inside the code generated for each component. See “View Execution Times” (Embedded Coder) for more information.



# Code Execution Profiling Report for gpu_fftn

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

### 1. Summary

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

### 2. Profiled Sections of Code

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

### 3. GPU Profiling Trace for gpu_fftn

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

OK Help

## 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 comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

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 since 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 profile for. A default `coder.EmbeddedCodeConfig` object is used when this value is not specified.

**`CodegenArguments` — Additional codegen arguments**
`''` (default) | string

Specify any additional `codegen` arguments as a string. The default value is NULL (empty string).

**`Threshold` — Threshold value**
0.01 (default) | numeric value between 0 and 1

Use the threshold value to control the GPU calls that are displayed in the report. If the maximum execution time from the executions is `x` seconds, the software reports all GPU calls that exceed `x * threshold`.

## See Also

`codegen` | `coder.EmbeddedCodeConfig`

**Topics**
“Analyze Execution Profiles of the Generated Code”
“GPU Execution Profiling of the Generated Code”

**Introduced in R2018b**

# 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 GPU Coder Support Package for NVIDIA GPUs.

### 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 GPU Coder Support Package for NVIDIA GPUs.

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;`

## 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
```

## See Also

codegen | coder.CodeConfig | coder.EmbeddedCodeConfig | coder.MexCodeConfig

**Introduced in R2019a**

# 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` cannot be used for inputs that are greater than two dimensions.

## See Also

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

**Topics**

“Design Patterns”
“Kernels from Element-Wise Loops”
“Kernels from Scatter-Gather Type Operations”
“Kernels from Library Calls”

**Introduced in R2019a**

# 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.

## 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.

## See Also

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

**Topics**
“Kernels from Element-Wise Loops”
“Kernels from Scatter-Gather Type Operations”
“Design Patterns”
“Kernels from Library Calls”

**Introduced in R2019b**

# 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).

## 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 use with half-precision inputs.

### Math and Arithmetic

| | |
|---|---|
| abs | Absolute value and complex magnitude |
| acos | Inverse cosine in radians |
| asin | Inverse sine in radians |
| atanh | Inverse hyperbolic tangent |
| ceil | Round toward positive infinity |
| conv | Convolution and polynomial multiplication |
| conv2 | 2-D convolution |
| cos | Cosine of argument in radians |
| cospi | Compute cos(X*pi) accurately |
| dot | Dot product |

| | |
|---|---|
| exp | Exponential |
| expm1 | Compute exp(x)-1 accurately for small values of x |
| 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) |
| ldivide | Left array division |
| log | Natural logarithm |
| log1p | Compute log(1+x) accurately for small values of x |
| mean | Average or mean value of array |
| minus | Subtraction |
| mod | Remainder after division (modulo operation) |
| mtimes | Matrix multiplication |
| plus | Addition |
| pow10 | Base 10 power and scale half-precision numbers |
| pow2 | Base 2 power and scale floating-point numbers |
| prod | Product of array elements |
| rdivide | Right array division |
| rem | Remainder after division |
| round | Round to nearest decimal or integer |
| rsqrt | Reciprocal square root |
| sin | Sine of argument in radians |
| sinpi | Compute sin(X*pi) accurately |
| sqrt | Square root |
| sum | Sum of array elements |
| tanh | Hyperbolic tangent |
| times | Multiplication |
| uminus | Unary minus |
| uplus | Unary plus |

## Data Types

| | |
|---|---|
| cast | Convert variable to different data type |
| cell | Cell array |
| double | Double-precision arrays |
| eps | Floating-point relative accuracy |
| 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 |
| islogical | Determine if input is logical array |
| isnan | Determine which array elements are NaN |
| isnumeric | Determine whether input is numeric array |
| isreal | Determine whether array is real |
| logical | Convert numeric values to logicals |
| NaN | Create array of all NaN values |
| single | Single-precision arrays |
| 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 |
| any | Determine if any array elements are nonzero |
| eq | Determine equality |
| ge | Determine greater than or equal to |
| gt | Determine greater than |
| le | Determine less than or equal to |
| lt | Determine less than |
| ne | Determine inequality |
| not | Find logical NOT |
| or | Find logical OR |

## Array and Matrix Operations

| | |
|---|---|
| cat | Concatenate arrays |
| colon | Vector creation, array subscripting, and for-loop iteration |
| eye | Identity matrix |
| full | Convert sparse matrix to full storage |
| horzcat | Horizontal concatenation for heterogeneous arrays |
| iscolumn | Determine whether 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 whether input is row vector |
| isscalar | Determine whether input is scalar |
| isvector | Determine whether input is vector |
| length | Length of largest array dimension |
| 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 |
| repmat | Repeat copies of array |
| reshape | Reshape array |
| size | Array size |
| subsasgn | Redefine subscripted assignment |
| subsref | Subscripted reference |
| transpose | Transpose vector or matrix |
| vertcat | Vertical concatenation for heterogeneous arrays |
| zeros | Create array of all zeros |

## Language Fundamentals

| | |
|---|---|
| display | Show information about variable or result of expression |
| end | Terminate block of code or indicate last array index |

## Graphics

| | |
|---|---|
| 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 |

## 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

The following functions which support half-precision inputs, do not support complex half-precision inputs.

- `rsqrt`
- `fma`

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

All functions that support half-precision inputs, support code generation, except for the `rsqrt` function.

In MATLAB, the `isobject` function returns true with a half-precision input. However, in generated code, this function returns false.

If your target hardware does not have native support for half-precision, then half is used as a storage type, with arithmetic operations performed in single precision.

Some functions use half only as a storage type and the arithmetic is always performed in single-precision, regardless of the target hardware.

Code generation for 32-bit targets is not supported if your MATLAB code contains half-precision data types.

### GPU Code Generation

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

- CUDA compute capability of 5.3 or higher is required for generating and executing code with half-precision data types.
- CUDA toolkit version of 10.0 or higher is required for generating and executing code with half-precision data types.
- The memory allocation (`malloc`) mode for generating CUDA code must be set to `'Discrete'`.

  For more information, see `coder.gpuConfig`.
- Half-precision complex data types are not supported for GPU code generation.
- For GPU Code generation, half-precision matrix multiplication can only be performed with real inputs.
- In MATLAB, the `isobject` function returns true with a half-precision input. However, in generated code, this function returns false.
- If your target hardware does not have native support for half-precision, then half is used as a storage type, with arithmetic operations performed in single precision.
- Some functions use half only as a storage type and the arithmetic is always performed in single-precision, regardless of the target hardware.
- Code generation for 32-bit targets is not supported if your MATLAB code contains half-precision data types.

## See Also

`double` | `single`

### Topics

“Floating-Point Numbers” (Fixed-Point Designer)

“Edge Detection with Sobel Method in Half-Precision” (MATLAB Coder)
Edge Detection with Sobel Method in Half-Precision

**Introduced in R2018b**

# 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 types of code files in the zip file include:

- Source files (for example, `.c` and `.cpp` files)
- Header files (for example, `.h` 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.

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, 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.

## 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

If you configure zip file packaging with `set_param`, the code generator uses `packNGo` to output a zip file during the build process.

Use `set_param` to configure zip file packaging for model `zingbit` in the file `zingbit.zip`.

```
set_param('zingbit','PostCodeGenCommand', ...
   'packNGo(buildInfo);');
```

## Input Arguments

#### `buildInfo` — Object that provides build information

`buildInfo` object

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.

#### Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `'packType','flat','nestedZipFiles',true`

#### `packType` — Determines whether the primary zip file contains secondary zip files or folders

`'flat'` (default) | `'hierarchical'`

If `'flat'`, package the generated code files in a zip file as a single, flat folder.

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.

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`, 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.

## See Also

**Topics**
“Relocate Generated Code to Another Development Environment”

**Introduced in R2006b**

# 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`. `gpucoder.batchedMatrixMultiply` performs matrix-matrix multiplication of the form:

$$D = \alpha AB$$

where $\alpha$ is a scalar multiplication factor, `A`, `B`, and `D` are matrices with dimensions `m`-by-`k`, `k`-by-`n`, and `m`-by-`n` respectively. `A` and `B` can optionally be transposed or hermitian-conjugated. By default, $\alpha$ is set to one and the matrices are not transposed. Use the `Name,Value` pair arguments to specify a different scalar multiplication factor and to specify transpose operations on the input matrices.

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:

$$D_i = \alpha A_i B_i \qquad i = 1...N$$

`___ = gpucoder.batchedMatrixMultiply( ___ ,Name,Value)` performs batched matrix multiply operation using the options specified by one or more `Name,Value` pair arguments.

## Examples

### Batched Matrix-Matrix Multiplication

This example performs a simple batched matrix-matrix multiplication and uses 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`. The input matrices are not transposed, therefore 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
```

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

```
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};
```

Use the `codegen` function to generate a CUDA library.

```
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. It also contains the `cublasDgemmBatched` API calls to the cuBLAS library. The following 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 comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

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`.

## See Also

`coder.gpu.constantMemory` | `coder.gpu.kernel` | `coder.gpu.kernelfun` | `gpucoder.batchedMatrixMultiplyAdd` | `gpucoder.sort` | `gpucoder.stencilKernel` | `gpucoder.stridedMatrixMultiply` | `gpucoder.stridedMatrixMultiplyAdd`

#### Topics

“Code Generation Using the Command Line Interface”
“Kernels from Element-Wise Loops”
“Kernels from Scatter-Gather Type Operations”
“Kernels from Library Calls”
“Design Patterns”

#### Introduced in R2020a

# 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`. `gpucoder.batchedMatrixMultiplyAdd` performs matrix-matrix multiplication of the form:

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

where $\alpha$ and $\beta$ 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`, `B`, and `C` can optionally be transposed or hermitian-conjugated. By default, $\alpha$ and $\beta$ are set to one and the matrices are not transposed. Use the `Name,Value` pair arguments to specify a different scalar multiplication factor and to specify transpose operations on the input matrices.

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` pairs 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 using the options specified by one or more `Name,Value` pair arguments.

## Examples

### Batched Matrix-Matrix Multiplication with Add

This example performs a simple batched matrix-matrix multiplication with add and uses 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`. The input matrices are not transposed, therefore 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
```

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

```
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};
```

Use the `codegen` function to generate a CUDA library.

```
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. It also contains the `cublasDgemmBatched` API calls to the cuBLAS library. The following 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 comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

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`.

## See Also

`coder.gpu.constantMemory` | `coder.gpu.kernel` | `coder.gpu.kernelfun` | `gpucoder.batchedMatrixMultiply` | `gpucoder.sort` | `gpucoder.stencilKernel` | `gpucoder.stridedMatrixMultiply` | `gpucoder.stridedMatrixMultiplyAdd`

**Topics**
“Code Generation Using the Command Line Interface”
“Kernels from Element-Wise Loops”
“Kernels from Scatter-Gather Type Operations”
“Kernels from Library Calls”
“Design Patterns”

**Introduced in R2020a**

# 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. `gpucoder.stridedMatrixMultiply` performs matrix-matrix multiplication of the form:

$$D = \alpha AB$$

where $\alpha$ is a scalar multiplication factor, `A`, `B`, and `D` are matrices with dimensions `m`-by-`k`, `k`-by-`n`, and `m`-by-`n` respectively. `A` and `B` can optionally be transposed or hermitian-conjugated. By default, $\alpha$ is set to one and the matrices are not transposed. Use the `Name,Value` pair arguments to specify a different scalar multiplication factor and to specify transpose operations on the input matrices.

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 using the options specified by one or more `Name,Value` pair arguments.

## Examples

### Strided Batched Matrix-Matrix Multiplication

This example performs a simple batched matrix-matrix multiplication and uses 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`. The input matrices are not transposed, therefore use the `'nn'` option.

```
function [D] = myStridedMatMul(A,B,alpha)

[D] = gpucoder.stridedMatrixMultiply(A,B,'alpha',alpha, ...
    'transpose','nn');

end
```

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

```
A = coder.newtype('double',[15,42],[0 0]);
B = coder.newtype('double',[42,30],[0 0]);
```

```
alpha = 0.3;
inputs = {A,B,alpha};
```

Use the `codegen` function to generate a CUDA library.

```
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. It also contains the `cublasDgemmStridedBatched` API calls to the cuBLAS library. The following is a snippet of the generated code.

```
//
// File: myStridedMatMul.cu
//
...

void myStridedMatMul(const double A[630], const double B[1260], double alpha,
                     double D[450])
{
  double alpha1;
...

  myStridedMatMul_kernel1<<<dim3(1U, 1U, 1U), dim3(480U, 1U, 1U)>>>(*gpu_D);
  beta1 = 0.0;
  cudaMemcpy(gpu_alpha1, &alpha1, 8UL, cudaMemcpyHostToDevice);
  cudaMemcpy(gpu_A, (void *)&A[0], 5040UL, cudaMemcpyHostToDevice);
  cudaMemcpy(gpu_B, (void *)&B[0], 10080UL, cudaMemcpyHostToDevice);
  cudaMemcpy(gpu_beta1, &beta1, 8UL, cudaMemcpyHostToDevice);
  cublasDgemmStridedBatched(getCublasGlobalHandle(), CUBLAS_OP_N, CUBLAS_OP_N,
    15, 30, 42, (double *)gpu_alpha1, (double *)&(*gpu_A)[0], 15, 0, (double *)
    &(*gpu_B)[0], 42, 0, (double *)gpu_beta1, (double *)&(*gpu_D)[0], 15, 450, 1);
  cudaMemcpy(&D[0], gpu_D, 3600UL, cudaMemcpyDeviceToHost);
...
}
```

## 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 comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

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`.

## See Also

`coder.gpu.constantMemory` | `coder.gpu.kernel` | `coder.gpu.kernelfun` | `gpucoder.batchedMatrixMultiply` | `gpucoder.batchedMatrixMultiplyAdd` | `gpucoder.sort` | `gpucoder.stencilKernel` | `gpucoder.stridedMatrixMultiplyAdd`

**Topics**

“Code Generation Using the Command Line Interface”
“Kernels from Element-Wise Loops”
“Kernels from Scatter-Gather Type Operations”
“Kernels from Library Calls”
“Design Patterns”

**Introduced in R2020a**

# 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. `gpucoder.stridedMatrixMultiplyAdd` performs matrix-matrix multiplication of the form:

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

where $\alpha$ and $\beta$ 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`, `B`, and `C` can optionally be transposed or hermitian-conjugated. By default, $\alpha$ and $\beta$ are set to one and the matrices are not transposed. Use the `Name,Value` pair arguments to specify a different scalar multiplication factor and to specify transpose operations on the input matrices.

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 using the options specified by one or more `Name,Value` pair arguments.

## Examples

### Strided Matrix-Matrix Multiplication with Add

This example performs a simple batched matrix-matrix multiplication with add and uses 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`. The input matrices are not transposed, therefore 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
```

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

```
A = coder.newtype('double',[12,14],[0 0]);
B = coder.newtype('double',[14,16],[0 0]);
```

```
C = coder.newtype('double',[12,16],[0 0]);
alpha = 0.3;
beta = 0.6;
inputs = {A,B,C,alpha,beta};
```

Use the `codegen` function to generate a CUDA library.

```
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. It also contains the `cublasDgemmStridedBatched` API calls to the cuBLAS library. The following is a snippet of the generated code.

```
//
// File: myStridedMatMulAdd.cu
...

void myStridedMatMulAdd(const double A[168], const double B[224], const double
  C[192], double alpha, double beta, double D[192])
{
  double alpha1;
...
  cudaMemcpy(gpu_C, (void *)&C[0], 1536UL, cudaMemcpyHostToDevice);
  myStridedMatMulAdd_kernel1<<<dim3(1U, 1U, 1U), dim3(192U, 1U, 1U)>>>(*gpu_C,
    *gpu_D);
  cudaMemcpy(gpu_alpha1, &alpha1, 8UL, cudaMemcpyHostToDevice);
  cudaMemcpy(gpu_A, (void *)&A[0], 1344UL, cudaMemcpyHostToDevice);
  cudaMemcpy(gpu_B, (void *)&B[0], 1792UL, cudaMemcpyHostToDevice);
  cudaMemcpy(gpu_beta1, &beta1, 8UL, cudaMemcpyHostToDevice);
  cublasDgemmStridedBatched(getCublasGlobalHandle(), CUBLAS_OP_N, CUBLAS_OP_N,
    12, 16, 14, (double *)gpu_alpha1, (double *)&(*gpu_A)[0], 12, 0, (double *)
    &(*gpu_B)[0], 14, 0, (double *)gpu_beta1, (double *)&(*gpu_D)[0], 12, 192, 1);
  cudaMemcpy(&D[0], gpu_D, 1536UL, cudaMemcpyDeviceToHost);
...
}
```

## 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 comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

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`.

## See Also

`coder.gpu.constantMemory` | `coder.gpu.kernel` | `coder.gpu.kernelfun` | `gpucoder.batchedMatrixMultiply` | `gpucoder.batchedMatrixMultiplyAdd` | `gpucoder.sort` | `gpucoder.stencilKernel` | `gpucoder.stridedMatrixMultiply`

**Topics**

“Code Generation Using the Command Line Interface”
“Kernels from Element-Wise Loops”
“Kernels from Scatter-Gather Type Operations”
“Kernels from Library Calls”
“Design Patterns”

**Introduced in R2020a**