NVIDIA CUDA Fortran Programming Guide

Release 25.5

NVIDIA Corporation

May 19, 2025

Contents

NVIDIA CUDA Fortran Programming Guide

Preface

This document describes CUDA Fortran, a small set of extensions to Fortran that supports and is built upon the CUDA computing architecture.

Intended Audience

This guide is intended for application programmers, scientists and engineers proficient in programming with the Fortran, C, and/or C++ languages. The tools are available on a variety of operating systems for the x86-64 and Arm server hardware platforms. This guide assumes familiarity with basic operating system usage.

Organization

The organization of this document is as follows:

Introduction
: contains a general introduction

Programming Guide
: serves as a programming guide for CUDA Fortran

Reference
: describes the CUDA Fortran language reference

Runtime APIs
: describes the interface between CUDA Fortran and the CUDA Runtime API

Examples
: provides sample code and an explanation of the simple example.

Conventions

This guide uses the following conventions:

italic
: is used for emphasis.

`Constant Width`
: is used for filenames, directories, arguments, options, examples, and for language statements in the text, including assembly language statements.

Bold
: is used for commands.

[item1]
: in general, square brackets indicate optional items. In this case item1 is optional. In the context of p/t-sets, square brackets are required to specify a p/t-set.

{ item2 | item 3 }
: braces indicate that a selection is required. In this case, you must select either item2 or item3.

filename ...
ellipsis indicate a repetition. Zero or more of the preceding item may occur. In this example, multiple filenames are allowed.

FORTRAN
Fortran language statements are shown in the text of this guide using a reduced fixed point size.

C/C++
C/C++ language statements are shown in the test of this guide using a reduced fixed point size.

The NVIDIA HPC compilers are supported on 64-bit variants of the Linux operating system on a variety of x86-compatible and Arm processors.

Related Publications

The following documents contain additional information related to CUDA Fortran programming.

- ISO/IEC 1539-1:1997, Information Technology – Programming Languages – FORTRAN, Geneva, 1997 (Fortran 95).
- NVIDIA CUDA Programming Guides, NVIDIA, Version 11, 11/23/2021. Available online at docs.nvidia.com/cuda.
- NVIDIA HPC Compiler User's Guide, Release 2025. Available online at docs.nvidia.com/hpc-sdk.
- NVIDIA Fortran CUDA Interfaces, Release 2025. Available online at docs.nvidia.com/hpc-sdk/fortran-cuda-interfaces.

Welcome to Release 2025 of NVIDIA CUDA Fortran, a small set of extensions to Fortran that supports and is built upon the CUDA computing architecture.

Graphic processing units or GPUs have evolved into programmable, highly parallel computational units with very high memory bandwidth, and tremendous potential for many applications. GPU designs are optimized for the computations found in graphics rendering, but are general enough to be useful in many data-parallel, compute-intensive programs.

NVIDIA introduced CUDA®, a general purpose parallel programming architecture, with compilers and libraries to support the programming of NVIDIA GPUs. CUDA comes with an extended C compiler, here called CUDA C, allowing direct programming of the GPU from a high level language. The programming model supports four key abstractions: cooperating threads organized into thread groups, shared memory and barrier synchronization within thread groups, and coordinated independent thread groups organized into a grid. A CUDA programmer must partition the program into coarse grain blocks that can be executed in parallel. Each block is partitioned into fine grain threads, which can cooperate using shared memory and barrier synchronization. A properly designed CUDA program will run on any CUDA-enabled GPU, regardless of the number of available processor cores.

CUDA Fortran includes a Fortran 2003 compiler and tool chain for programming NVIDIA GPUs using Fortran. NVIDIA 2025 includes support for CUDA Fortran on Linux. CUDA Fortran is an analog to NVIDIA's CUDA C compiler. Compared to the NVIDIA OpenACC directives-based model and compilers, CUDA Fortran is a lower-level explicit programming model with substantial runtime library components that give expert programmers direct control of all aspects of GPGPU programming.

The CUDA Fortran extensions described in this document allow the following operations in a Fortran program:

- Declaring variables that are allocated in the GPU device memory
- Allocating dynamic memory in the GPU device memory
- Copying data from the host memory to the GPU memory, and back

- Writing subroutines and functions to execute on the GPU
- Invoking GPU subroutines from the host
- Allocating pinned memory on the host
- Using asynchronous transfers between the host and GPU
- Using zero-copy and CUDA Unified Virtual Addressing features.
- Accessing read-only data through texture memory caches.
- Automatically generating GPU kernels using the kernel loop directive.
- Launching GPU kernels from other GPU subroutines running on the device using dynamic parallelism features.
- Relocatable device code: Creating and linking device libraries and calling functions defined in other modules and files.
- Interfacing to CUDA C.
- Programming access to Tensor Core hardware.

Chapter 1. Programming Guide

This section introduces the CUDA programming model through examples written in CUDA Fortran. For a reference for CUDA Fortran, refer to *Reference*.

1.1. CUDA Fortran Host and Device Code

All CUDA programs, and in general any program which uses a GPU for computation, must perform the following steps:

1. Initialize and select the GPU to run on. Oftentimes this is implicit in the program and defaults to NVIDIA device 0.
2. Allocate space for data on the GPU.
3. Move data from the host to the GPU, or in some cases, initialize the data on the GPU.
4. Launch kernels from the host to run on the GPU.
5. Gather results back from the GPU for further analysis our output from the host program.
6. Deallocate the data on the GPU allocated in step 2. This might be implicitly performed when the host program exits.

Here is a simple CUDA Fortran example which performs the required steps:

Explicit Device Selection

Host code	Device Code

`program t1`
`use cudafor`
`use mytests`
`integer, parameter :: n = 100`
`integer, allocatable, device :: iarr(:)`
`integer h(n)`
`istat = cudaSetDevice(0)`
`allocate(iarr(n))`
`h = 0; iarr = h`
`call test1<<<1,n>>> (iarr)`
`h = iarr`
`print *,&`
`"Errors: ", count(h.ne.(/ (i,i=1,n) /))`
`deallocate(iarr)`
`end program t1`
	`module mytests`
	`contains`
	`attributes(global)  &`
	`subroutine test1( a )`
	`integer, device :: a(*)`
	`i = threadIdx%x`
	`a(i) = i`
	`return`
	`end subroutine test1`
	`end module mytests`

In the CUDA Fortran host code on the left, device selection is *explicit*, performed by an API call on line 7. The provided `cudafor` module, used in line 2, contains interfaces to the full CUDA host runtime library, and in this case exposes the interface to `cudaSetDevice()` and ensures it is called correctly. An array is allocated on the device at line 8. Line 9 of the host code initializes the data on the host and the device, and, in line 10, a device kernel is launched. The interface to the device kernel is explicit, in the Fortran sense, because the module containing the kernel is used in line 3. At line 11 of the host code, the results from the kernel execution are moved back to a host array. Deallocation of the GPU array occurs on line 14.

Implicit Device Selection

Here is a CUDA Fortran example which is slightly more complicated than the preceding one.

Host code	Device Code

`program testramp`
`use cublas`
`use ramp`
`integer, parameter :: N = 20000`
`real, device :: x(N)`
`twopi = atan(1.0)*8`
`call buildramp<<<(N-1)/512+1,512>>>(x,N)`
`!$cuf kernel do`
`do i = 1, N`
`x(i) = 2.0 * x(i) * x(i)`
`end do`
`print *,"float(N) = ",sasum(N,x,1)`
`end program`
	`module ramp`
	`real, constant :: twopi`
	`contains`
	`attributes(global) &`
	`subroutine buildramp(x, n)`
	`real, device :: x(n)`
	`integer, value :: n`
	`real, shared :: term`
	`if (threadidx%x == 1) term = &`
	`twopi / float(n)`
	`call syncthreads()`
	`i = (blockidx%x-1)*blockdim%x &`
	`+ threadidx%x`
	`if (i <= n) then`
	`x(i) = cos(float(i-1)*term)`
	`end if`
	`return`
	`end subroutine`
	`end module`

In this case, the device selection is *implicit*, and defaults to NVIDIA device 0. The device array allocation in the host code at line 5 looks static, but actually occurs at program init time. Larger array sizes are handled, both in the kernel launch at line 7 in the host code, and in the device code at line 10. The device code contains examples of constant and shared data, which are described in *Reference*. There are actually two kernels launched from the host code: one explicitly provided and called from line 10, and a second, generated using the CUDA Fortran kernel loop directive, starting at line 11. Finally, this example demonstrates the use of the `cublas` module, used at line 2 in the host code, and called at line 12.

As these two examples demonstrate, all the steps listed at the beginning of this section for using a GPU are contained within the host code. It is possible to program GPUs without writing any kernels and device code, through library calls and CUDA Fortran kernel loop directives as shown, or by using higher-level directive-based models; however, programming in a lower-level model like CUDA provides the programmer control over device resource utilization and kernel execution.

1.2. CUDA Fortran Kernels

CUDA Fortran allows the definition of Fortran subroutines that execute in parallel on the GPU when called from the Fortran program which has been invoked and is running on the host or, starting in CUDA 5.0, on the device. Such a subroutine is called a *device kernel* or *kernel*.

A call to a kernel specifies how many parallel instances of the kernel must be executed; each instance will be executed by a different CUDA thread. The CUDA threads are organized into thread blocks, and each thread has a global thread block index, and a local thread index within its thread block.

A kernel is defined using the `attributes(global)` specifier on the subroutine statement; a kernel is called using special chevron syntax to specify the number of thread blocks and threads within each thread block:

```
! Kernel definition
attributes(global) subroutine ksaxpy( n, a, x, y )
    real, dimension(*) :: x,y
    real, value :: a
    integer, value :: n, i
    i = (blockidx%x-1) * blockdim%x + threadidx%x
    if( i <= n ) y(i) = a * x(i) + y(i)
end subroutine

! Host subroutine
subroutine solve( n, a, x, y )
    real, device, dimension(*) :: x, y
    real :: a
    integer :: n
    ! call the kernel
    call ksaxpy<<<n/64, 64>>>( n, a, x, y )
end subroutine
```

In this case, the call to the kernel `ksaxpy` specifies `n/64` thread blocks, each with 64 threads. Each thread is assigned a thread block index accessed through the built-in `blockidx` variable, and a thread index accessed through `threadidx`. In this example, each thread performs one iteration of the common SAXPY loop operation.

1.3. Thread Blocks

Each thread is assigned a thread block index accessed through the built-in `blockidx` variable, and a thread index accessed through `threadidx`. The thread index may be a one-, two-, or three-dimensional index. In CUDA Fortran, the thread index for each dimension starts at one.

Threads in the same thread block may cooperate by using *shared memory,* and by synchronizing at a barrier using the `SYNCTHREADS()` intrinsic. Each thread in the block waits at the call to `SYNCTHREADS()` until all threads have reached that call. The shared memory acts like a low-latency, high bandwidth software managed cache memory. Currently, the maximum number of threads in a thread block is 1024.

A kernel may be invoked with many thread blocks, each with the same thread block size. The thread blocks are organized into a one-, two-, or three-dimensional *grid* of blocks, so each thread has a thread index within the block, and a block index within the grid. When invoking a kernel, the first argument in the chevron `<<<>>>` syntax is the grid size, and the second argument is the thread block size. Thread blocks must be able to execute independently; two thread blocks may be executed in parallel or one after the other, by the same core or by different cores.

The `dim3` derived type, defined in the `cudafor` module, can be used to declare variables in host code which can conveniently hold the launch configuration values if they are not scalars; for example:

```
type(dim3) :: blocks, threads
...
blocks = dim3(n/256, n/16, 1)
threads = dim3(16, 16, 1)
call devkernel<<<blocks, threads>>>( ... )
```

1.4. Memory Hierarchy

CUDA Fortran programs have access to several memory spaces. On the host side, the host program can directly access data in the host main memory. It can also directly copy data to and from the device global memory; such data copies require DMA access to the device, so are slow relative to the host memory. The host can also set the values in the device constant memory, again implemented using DMA access.

On the device side, data in global device memory can be read or written by all threads. Data in constant memory space is initialized by the host program; all threads can read data in constant memory. Accesses to constant memory are typically faster than accesses to global memory, but it is read-only to the threads and limited in size. Threads in the same thread block can access and share data in shared memory; data in shared memory has a lifetime of the thread block. Each thread can also have private local memory; data in thread local memory may be implemented as processor registers or may be allocated in the global device memory; best performance will often be obtained when thread local data is limited to a small number of scalars that can be allocated as processor registers.

Through use of the CUDA API as exposed by the `cudafor` module, access to CUDA features such as mapped memory, peer-to-peer memory access, and the unified virtual address space are supported. Users should check the relevant CUDA documentation for compute capability restrictions for these features. For an example of device array mapping, refer to *Mapped Memory Example*.

Starting with CUDA 6.0, managed or unified memory programming is available on certain platforms. For a complete description of unified memory programming, see the Unified Memory Programing section of the *CUDA C Programming Guide*. Managed memory provides a common address space, and migrates data between the host and device as it is used by each set of processors. On the host side, the data is resident in host main memory. On the device side, it is accessed as resident in global device memory.

1.5. Subroutine / Function Qualifiers

A subroutine or function in CUDA Fortran has an additional attribute, designating whether it is executed on the host or on the device, and if the latter, whether it is a kernel, called from the host, or called from another device subprogram.

- A subprogram declared with `attributes(host)`, or with the host attribute by default, is called a *host subprogram*.
- A subprogram declared with `attributes(global)` or `attributes(device)` is called a *device subprogram*.
- A subroutine declared with `attributes(global)` is also called a *kernel subroutine*.
- A subroutine declared with `attributes(grid_global)` is supported starting on cc70 hardware or greater. Threads within the grid in these kernels are co-resident on the same device and can be synchronized.

1.5.1. Attributes(host)

The `host` attribute, specified on the subroutine or function statement, declares that the subroutine or function is to be executed on the host. Such a subprogram can only be called from another host subprogram. The default is `attributes(host)`, if none of the `host`, `global`, or `device` attributes is specified.

1.5.2. Attributes(global)

The `global` attribute may only be specified on a subroutine statement; it declares that the subroutine is a kernel subroutine, to be executed on the device, and may only be called using a kernel call containing the chevron syntax and runtime mapping parameters.

1.5.3. Attributes(device)

The `device` attribute, specified on the subroutine or function statement, declares that the subprogram is to be executed on the device; such a routine must be called from a subprogram with the `global` or `device` attribute.

1.5.4. Attributes(host,device)

The `host,device` attribute, specified on the subroutine or function statement, declares that the subprogram can be executed on both the host and device; such a routine can be called from host code, or from a subprogram with the `global` or `device` attribute. It is typically used for small target-independent functions.

1.5.5. Attributes(grid_global)

The `grid_global` attribute may only be specified on a subroutine statement; it declares that the subroutine is a kernel subroutine, to be executed on the device, and may only be launched using a kernel call containing the chevron syntax and runtime mapping parameters. The kernel is launched such that all threads within the grid group are guaranteed to be co-resident on the device. This allow a grid synchronization operation on cc70 hardware and greater.

1.5.6. Restrictions

The following restrictions apply to subprograms.

- A device subprogram must not contain variables with the SAVE attribute, or with data initialization.
- A kernel subroutine may not also have the `device` or `host` attribute.
- Calls to a kernel subroutine must specify the execution configuration, as described in “Predefined Variables in Device Subprograms,” on page 9. Such a call is *asynchronous*, that is, the calling routine making the call continues to execute before the device has completed its execution of the kernel subroutine.
- Device subprograms may not be contained in a host subroutine or function, and may not contain any subroutines or functions.

1.6. Variable Qualifiers

Variables in CUDA Fortran have a new attribute that declares in which memory the data is allocated. By default, variables declared in modules or host subprograms are allocated in the host main memory. At most one of the `device`, `managed`, `constant`, `shared`, or `pinned` attributes may be specified for a variable.

1.6.1. Attributes(device)

A variable with the `device` attribute is called a *device variable*, and is allocated in the device global memory.

- If declared in a module, the variable may be accessed by any subprogram in that module and by any subprogram that uses the module.
- If declared in a host subprogram, the variable may be accessed by that subprogram or subprograms contained in that subprogram.

A device array may be an explicit-shape array, an allocatable array, or either an assumed-size or assumed-shape dummy array. An allocatable device variable has a dynamic lifetime, from when it is allocated until it is deallocated. Other device variables have a lifetime of the entire application.

1.6.2. Attributes(managed)

Starting with CUDA 6.0, on certain platforms, a variable with the `managed` attribute is called a *managed variable*. Managed variables may be used in both host and device code. Variables with the managed attribute migrate between the host and device, depending on where the accesses to the memory originate. Managed variables may be read and written by the host, but there are access restrictions on the managed variables if kernels are active on the device. On the device, managed variables have characteristics similar to device variables, but managed variables cannot be allocated from the device, as device variables can be, starting in CUDA 5.0 in support of dynamic parallelism.

1.6.3. Attributes(constant)

A variable with the `constant` attribute is called a *device constant variable*. Device constant variables are allocated in the device constant memory space. The constant variable must be declared within a module's global data specification scope. When declared, the variable may be accessed by any subprogram in that module and by any subprogram that uses the module. Device constant data may not be assigned or modified in any device subprogram, but may be modified in host subprograms. All host accesses of constant memory must be through use or host association. Device constant variables may not be allocatable, and they have a lifetime, in the device constant memory, of the entire application.

1.6.4. Attributes(shared)

A variable with the `shared` attribute is called a device shared variable or a *shared variable*. A shared variable may only be declared in a device subprogram, and may only be accessed within that subprogram, or by other device subprograms to which it is passed as an argument. A shared variable may not be data initialized. A shared variable is allocated in the device shared memory for a thread block, and has a lifetime of the thread block. It can be read or written by all threads in the block, though a write in one thread is only guaranteed to be visible to other threads after the next call to the `SYNCTHREADS()` intrinsic.

1.6.5. Attributes(pinned)

A variable with the `pinned` attribute is called a *pinned variable*. A pinned variable must be an allocatable array. When a pinned variable is allocated, it will be allocated in host pagelocked memory. The advantage of using pinned variables is that copies from page-locked memory to device memory are faster than copies from normal paged host memory. Some operating systems or installations may restrict the use, availability, or size of page-locked memory; if the allocation in page-locked memory fails, the variable will be allocated in the normal host paged memory and required for asynchronous moves.

1.6.6. Attributes(texture)

Reading values through the texture memory interface is no longer recommended or necessary on newer GPUs and support for this feature has been dropped in CUDA 12.0.

1.6.7. Attributes(unified)

Starting with the NVHPC 24.3 release, on systems which support it, a variable with the `unified` attribute is called a *unified variable*. Similar to managed variables, unified variables may be used in both host and device code. The compiler will allow passing a unified variable for an argument expecting a device variable. Variables with the unified attribute may migrate between the host and device, but depending on the driver version and settings, may do so under different conditions than managed variables. Unified variables are created in host system memory. Similar to managed variables, care must be taken when unified variables are accessed from both host and device code, to avoid possible race conditions.

1.7. Datatypes in Device Subprograms

The following intrinsic datatypes are allowed in device subprograms and device data:

Table 1: Table 1. Intrinsic Datatypes

Type	Type Kind
`integer`	1,2,4,8
`logical`	1,2,4,8
`real`	2,4,8
`double precision`	equivalent to `real(kind=8)`
`complex`	4,8
`character(len=1)`	1

Derived types may contain members with these intrinsic datatypes or other allowed derived types.

1.7.1. Half-precision Floating Point

On NVIDIA GPUs which support CUDA Compute Capability 6.0 and above, it is possible to create variables and arrays as half precision floating point. CUDA Fortran offers support for using the `kind` attribute on real data types; allowing data to be declared as `real(2)`. The following operators are supported for this data type: `+ , -, *, /, .lt., .le., .gt., .ge., .eq.,.ne.`. The compiler will emit an error message when using `real(2)` and targeting a GPU with compute capability lower than 6.0.

Half precision is represented as IEEE 754 binary16. Out of the 16-bits available to represent the floating point value, one bit is used for sign, five bits are used for exponent, and ten bits are used for significand. When encountering values that cannot be precisely represented in the format, such as when adding two `real(2)` numbers, IEEE 754 defines rounding rules. In the case of real(2), the default rule is round-to-nearest with ties-to-even property which is described in detail in the IEEE 754-2008 standard in section 4.3.1. This format has a small dynamic range and thus values greater than 65520 are rounded to infinity.

1.8. Predefined Variables in Device Subprograms

Device subprograms have access to block and grid indices and dimensions through several built-in read-only variables. These variables are of type `dim3`; the module `cudafor` defines the derived type `dim3` as follows:

```
type(dim3)
    integer(kind=4) :: x,y,z
end type
```

These predefined variables are not accessible in host subprograms.

- The variable `threadidx` contains the thread index within its thread block; for one- or two-dimensional thread blocks, the `threadidx%y` and/or `threadidx%z` components have the value one.
- The variable `blockdim` contains the dimensions of the thread block; `blockdim` has the same value for all thread blocks in the same grid.
- The variable `blockidx` contains the block index within the grid; as with threadidx, for one-dimensional grids, `blockidx%y` and/or `blockidx%z` has the value one.
- The variable `griddim` contains the dimensions of the grid.
- The constant `warpsize` is declared to be type integer. Threads are executed in groups of 32, called *warps*; `warpsize` contains the number of threads in a warp, and is currently 32.

1.9. Execution Configuration

A call to a kernel subroutine must specify an execution configuration. The execution configuration defines the dimensionality and extent of the grid and thread blocks that execute the subroutine. It may also specify a dynamic shared memory extent, in bytes, and a stream identifier, to support concurrent stream execution on the device.

A kernel subroutine call looks like this:

```
call kernel<<<grid,block[,bytes][,streamid]>>>(arg1,arg2,...)
```

where

- `grid` and `block` are either integer expressions (for one-dimensional grids and thread blocks), or are `type(dim3)`, for one- or two-dimensional grids and thread blocks.
- If `grid` is `type(dim3)`, the value of each component must be equal to or greater than one, and the product is usually limited by the compute capability of the device.
- If `block` is `type(dim3)`, the value of each component must be equal to or greater than one, and the product of the component values must be less than or equal to 1024.
- The value of `bytes` must be an integer; it specifies the number of bytes of shared memory to be allocated for each thread block, in addition to the statically allocated shared memory. This memory is used for the assumed-size shared variables in the thread block; refer to *Shared data* for more information. If the value of `bytes` is not specified, its value is treated as zero.
- The value of `streamid` must be an integer greater than or equal to zero; it specifies the stream to which this call is associated. Nonzero stream values can be created with a call to `cudaStreamCreate`. Starting in CUDA 7.0, the constant `cudaStreamPerThread` can be specified to use a unique default stream for each CPU thread.

1.10. Asynchronous Concurrent Execution

There are two components to asynchronous concurrent execution with CUDA Fortran.

1.10.1. Concurrent Host and Device Execution

When a host subprogram calls a kernel subroutine, the call actually returns to the host program before the kernel subroutine begins execution. The call can be treated as a *kernel launch* operation, where the launch actually corresponds to placing the kernel on a queue for execution by the device. In this way, the host can continue executing, including calling or queueing more kernels for execution on the device. By calling the runtime routine `cudaDeviceSynchronize`, the host program can synchronize and wait for all previously launched or queued kernels.

Programmers must be careful when using concurrent host and device execution; in cases where the host program reads or modifies device or constant data, the host program should synchronize with the device to avoid erroneous results.

1.10.2. Concurrent Stream Execution

Operations involving the device, including kernel execution and data copies to and from device memory, are implemented using stream queues. An operation is placed at the end of the stream queue, and will only be initiated when all previous operations on that queue have been completed.

An application can manage more concurrency by using multiple streams. Each user-created stream manages its own queue; operations on different stream queues may execute out-of-order with respect to when they were placed on the queues, and may execute concurrently with each other.

The default stream, used when no stream identifier is specified, is stream zero; stream zero is special in that operations on the stream zero queue will begin only after all preceding operations on all queues are complete, and no subsequent operations on any queue begin until the stream zero operation is complete.

1.11. Kernel Loop Directive

CUDA Fortran allows automatic kernel generation and invocation from a region of host code containing one or more tightly nested loops. Launch configuration and mapping of the loop iterations onto the hardware is controlled and specified as part of the directive body using the familiar CUDA chevron syntax. As with any kernel, the launch is asynchronous. The program can use cudaDeviceSynchronize() or CUDA Events to wait for the completion of the kernel.

The work in the loops specified by the directive is executed in parallel, across the thread blocks and grid; it is the programmer's responsibility to ensure that parallel execution is legal and produces the correct answer. The one exception to this rule is a scalar reduction operation, such as summing the values in a vector or matrix. For these operations, the compiler handles the generation of the final reduction kernel, inserting synchronization into the kernel as appropriate.

1.11.1. Syntax

The general form of the kernel directive is:

```
!$cuf kernel do[(n)] <<< grid, block  [optional stream] >>>
```

The compiler maps the launch configuration specified by the grid and block values onto the outermost *n* loops, starting at loop *n* and working out. The grid and block values can be an integer scalar or a parenthesized list. Alternatively, using asterisks tells the compiler to choose a thread block shape and/or compute the grid shape from the thread block shape and the loop limits. Loops which are not mapped onto the grid and block values are run sequentially on each thread.

There are two ways to specify the optional stream argument:

```
!$cuf kernel do[(n)] <<< grid, block, 0, streamid >>>
```

Or

```
!$cuf kernel do[(n)] <<< grid, block, stream=streamid >>>
```

Kernel Loop Directive Example 1

```
!$cuf kernel do(2) <<< (*,*), (32,4) >>>
do j = 1, m
  do i = 1, n
    a(i,j) = b(i,j) + c(i,j)
  end do
end do
```

In this example, the directive defines a two-dimensional thread block of size 32x4.

The body of the doubly-nested loop is turned into the kernel body:

- ThreadIdx%x runs from 1 to 32 and is mapped onto the inner *i* loop.
- ThreadIdx%y runs from 1 to 4 and is mapped onto the outer *j* loop.

The grid shape, specified as (*,*), is computed by the compiler and runtime by dividing the loop trip counts *n* and *m* by the thread block size, so all iterations are computed.

Kernel Loop Directive Example 2

```
!$cuf kernel do <<< *, 256 >>>
do j = 1, m
  do i = 1, n
    a(i,j) = b(i,j) + c(i,j)
  end do
end do
```

Without an explicit *n* on the *do*, the schedule applies just to the outermost loop, that is, the default value is 1. In this case, only the outer *j* loop is run in parallel with a thread block size of 256. The inner *i* dimension is run sequentially on each thread.

You might consider if the code in *Kernel Loop Directive Example 2* would perform better if the two loops were interchanged. Alternatively, you could specify a configuration like the following in which the threads read and write the matrices in coalesced fashion.

```
!$cuf kernel do(2) <<< *, (256,1) >>>
do j = 1, m
  do i = 1, n
    a(i,j) = b(i,j) + c(i,j)
  end do
end do
```

Kernel Loop Directive Example 3

In *Kernel Loop Directive Example 2*, the 256 threads in each block each do one element of the matrix addition. Further expansion of the work along the *i* direction and all work across the *j* dimension is handled by the mapping onto the grid dimensions.

To “unroll” more work into each thread, specify non-asterisk values for the grid, as illustrated here:

```
!$cuf kernel do(2) <<< (1,*), (256,1) >>>
do j = 1, m
  do i = 1, n
```

(continues on next page)

(continued from previous page)

```
    a(i,j) = b(i,j) + c(i,j)
  end do
end do
```

Now the threads in a thread block handle all values in the *i* direction, in concert, incrementing by 256. One thread block is created for each *j*. Specifically, the *j* loop is mapped onto the grid x-dimension, because the compiler skips over the constant 1 in the *i* loop grid size. In CUDA built-in language, *gridDim%*x is equal to *m*.

1.11.2. Restrictions on the CUF kernel directive

The following restrictions apply to CUF kernel directives:

- If the directive specifies *n* dimensions, it must be followed by at least that many tightly-nested DO loops.
- The tightly-nested DO loops must have invariant loop limits: the lower limit, upper limit, and increment must be invariant with respect to any other loop in the kernel *do*.
- The invariant loop limits cannot be a value from an array expression, unless those arrays have the managed attribute.
- There can be no GOTO or EXIT statements within or between any loops that have been mapped onto the grid and block configuration values.
- The body of the loops may contain assignment statements, IF statements, loops, and GOTO statements.
- Only CUDA Fortran data types are allowed within the loops.
- Fortran intrinsic functions are allowed, if they are allowed and supported in device code.
- Device-specific intrinsics such as the CUDA atomic functions are allowed, but require the interfaces from the cudadevice module be explicitly used to compile correctly.
- Device-specific intrinsics such as the syncthreads and other warp or block-level cooperating, syncing, or barrier functions should be avoided except in very limited situations.
- Subroutine and function calls to attributes(device) subprograms are allowed if they are in the same module as the code containing the directive.
- Arrays used or assigned in the loop must have the device or managed attribute.
- Implicit loops and F90 array syntax are not allowed within the directive loops.
- Scalars used or assigned in the loop must either have the device attribute, or the compiler will make a device copy of that variable live for the duration of the loops, one for each thread. Except in the case of reductions; when a reduction has a scalar target, the compiler generates a correct sequence of synchronized operations to produce one copy either in device global memory or on the host.

1.11.3. Summation Example

The simplest directive form for performing a dot product on two device arrays takes advantage of the properties for scalar use outlined previously.

```
rsum = 0.0
!$cuf kernel do <<< *, * >>>
do i = 1, n
    rsum = rsum + x(i)* y(i)
end do
```

For reductions, the compiler recognizes the use of the scalar and generates just one final result.

This CUF kernel can be followed by another CUF kernel in the same subprogram:

```
!$cuf kernel do <<< *, * >>>
do i = 1, n
    rsum= x(i) * y(i)
    z(i) = rsum
end do
```

In this CUF kernel, the compiler recognizes *rsum* as a scalar temporary which should be allocated locally on every thread. However, use of *rsum* on the host following this loop is undefined.

1.11.4. Explicit Reductions

The CUDA Fortran compiler generally does a good job of identifying reductions in simple loops. When the reduction is not detected by the compiler, due to complicated control flow or other issues, starting in version 21.7, it is possible to specify explicit reductions using syntax similar to that used in the OpenACC and OpenMP programming models.

```
value = 0.0
!$cuf kernel do <<< *, * >>> reduce(+:value)
do i = 1, n
    a(i) = real(int(a(i) * 100.0 - 50.0),kind=4)
    if (a(i) .ge. 0.0) then
        value = value + a(i)
    else
        value = value + a(i) + 50.0
    end if
end do
```

Both the *reduce* and *reduction* keywords are accepted. Generally, all data types and types of reductions that are accepted in OpenACC Fortran are accepted in CUF kernels. That includes *+*, ***, *max*, *min*, *iand*, *ior*, and *ieor* for the Fortran integer type; *+*, ***, *max*, *min* for the Fortran real type; *+* for the Fortran complex type, and finally *.and.*, *.or.* for the Fortran logical type.

1.12. Using Fortran Modules

Modern Fortran uses modules to package global data, definitions, derived types, and interface blocks. In CUDA Fortran these modules can be used to easily communicate data and definitions between host and device code. This section includes a few examples of using Fortran Modules.

1.12.1. Accessing Data from Other Modules

in the following example, a set of modules are defined in one file which are accessed by another module.

Accessing data from other modules.

In one file, `moda.cuf`, you could define a set of modules:

```
module moda
  real, device, allocatable :: a(:)
end module

module modb
  real, device, allocatable :: b(:)
end module
```

In another module or file, `modc.cuf`, you could define another module which uses the two modules `moda` and `modb`:

```
module modc
  use moda
  use modb
  integer, parameter :: n = 100
  real, device, allocatable :: c(:)
  contains
    subroutine vadd()
    !$cuf kernel do <<<*,*>>>
    do i = 1, n
      c(i) = a(i) + b(i)
    end do
    end subroutine
end module
```

In the host program, you use the top-level module, and get the definition of `n` and the interface to `vadd`. You can also rename the device arrays so they do not conflict with the host naming conventions:

```
program t
use modc, a_d => a, b_d => b, c_d => c
real a,b,c(n)
allocate(a_d(n),b_d(n),c_d(n))
a_d = 1.0
b_d = 2.0
call vadd()
c = c_d
print *,all(c.eq.3.0)
end
```

1.12.2. Call Routines from Other Modules

Starting with CUDA 5.0, in addition to being able to access data declared in another module, you can also call device functions which are contained in another module. In the following example, the file `ffill.cuf` contains a device function to fill an array:

Calling routines from other modules using relocatable device code.

```
module ffill
  contains
    attributes(device) subroutine fill(a)
    integer, device :: a(*)
    i = (blockidx%x-1)*blockdim%x + threadidx%x
    a(i) = i
    end subroutine
end module
```

To generate relocatable device code, compile this file with the `-gpu=rdc` flag:

```
% nvfortran -cuda -gpu=rdc -c ffill.cuf
```

Now write another module and test program that calls the subroutine in this module. Since you are calling an attributes(device) subroutine, you do not use the chevron syntax. For convenience, an overloaded Fortran sum function is included in the file `tfill.cuf` which, in this case, takes 1-D integer device arrays.

```
module testfill
    use ffill
    contains
    attributes(global) subroutine Kernel(arr)
        integer, device :: arr(*)
        call fill(arr)
    end subroutine Kernel

    integer function sum(arr)
        integer, device :: arr(:)
        sum = 0
        !$cuf kernel do <<<*,*>>>
        do i = 1, size(arr)
          sum = sum + arr(i)
        end do
    end function sum
end module testfill

program tfill
use testfill
integer, device :: iarr(100)
iarr = 0
call Kernel<<<1,100>>>(iarr)
print *,sum(iarr)==100*101/2
end program tfill
```

This file also needs to be compiled with the `-gpu=rdc` flag and then can be linked with the previous object file:

```
% nvfortran -cuda -gpu=rdc tfill.cuf ffill.o
```

The `-gpu=rdc` option has been the default for many releases. The `-gpu=nordc` flag will override the current default.

1.12.3. Declaring Device Pointer and Target Arrays

Recently, NVIDIA added support for F90 pointers that point to device data. Currently, this is limited to pointers that are declared at module scope. The pointers can be accessed through module association, or can be passed in to global subroutines. The `associated()` function is also supported in device code. The following code shows many examples of using F90 pointers. These pointers can also be used in CUF kernels.

Declaring device pointer and target arrays in CUDA Fortran modules

```
module devptr
! currently, pointer declarations must be in a module
  real, device, pointer, dimension(:) :: mod_dev_ptr
  real, device, pointer, dimension(:) :: arg_dev_ptr
  real, device, target,  dimension(4) :: mod_dev_arr
  real, device, dimension(4) :: mod_res_arr
contains
  attributes(global) subroutine test(arg_ptr)
    real, device, pointer, dimension(:) :: arg_ptr
    ! copy 4 elements from one of two spots
    if (associated(arg_ptr)) then
      mod_res_arr = arg_ptr
    else
      mod_res_arr = mod_dev_ptr
    end if
  end subroutine test
end module devptr
```

```
program test
use devptr
real, device, target, dimension(4) :: a_dev
real  result(20)

a_dev = (/ 1.0, 2.0, 3.0, 4.0 /)

! Pointer assignment to device array declared on host,
! passed as argument.  First four result elements.
arg_dev_ptr => a_dev
call test<<<1,1>>>(arg_dev_ptr)
result(1:4) = mod_res_arr

!$cuf kernel do <<<*,*>>>
do i = 1, 4
  mod_dev_arr(i) = arg_dev_ptr(i) + 4.0
  a_dev(i)       = arg_dev_ptr(i) + 8.0
end do

! Pointer assignment to module array, argument nullified
! Second four result elements
mod_dev_ptr => mod_dev_arr
```

(continues on next page)

(continued from previous page)

```
arg_dev_ptr => null()
call test<<<1,1>>>(arg_dev_ptr)
result(5:8) = mod_res_arr

! Pointer assignment to updated device array, now associated
! Third four result elements
arg_dev_ptr => a_dev
call test<<<1,1>>>(arg_dev_ptr)
result(9:12) = mod_res_arr

!$cuf kernel do <<<*,*>>>
do i = 1, 4
  mod_dev_arr(i) = 25.0 - mod_dev_ptr(i)
  a_dev(i)       = 25.0 - arg_dev_ptr(i)
end do

! Non-contiguous pointer assignment to updated device array
! Fourth four element elements
arg_dev_ptr => a_dev(4:1:-1)
call test<<<1,1>>>(arg_dev_ptr)
result(13:16) = mod_res_arr

! Non-contiguous pointer assignment to updated module array
! Last four elements of the result
nullify(arg_dev_ptr)
mod_dev_ptr => mod_dev_arr(4:1:-1)
call test<<<1,1>>>(arg_dev_ptr)
result(17:20) = mod_res_arr

print *,all(result==(/(real(i),i=1,20)/))
end
```

1.12.4. Declaring Textures

Reading values through the texture memory interface is no longer recommended or necessary on newer GPUs and support for this feature has been dropped in CUDA 12.0.

CUDA texture memory can be fetched through a special texture attribute ascribed to F90 pointers that point to device data with the target attribute. In CUDA Fortran, textures are currently just for read-only data that travel through the texture cache. Since there is separate hardware to support this cache, in many cases using the texture attribute is a performance boost, especially in cases where the accesses are irregular and noncontiguous amongst threads. The following simple example demonstrates this capability:

Declaring textures in CUDA Fortran modules

```
module memtests
  real(8), texture, pointer :: t(:)  ! declare the texture
  contains
    attributes(device) integer function bitrev8(i)
    integer ix1, ix2, ix
    ix = i
    ix1 = ishft(iand(ix,z'0aa'),-1)
    ix2 = ishft(iand(ix,z'055'), 1)
```

(continues on next page)

(continued from previous page)

```
    ix = ior(ix1,ix2)
    ix1 = ishft(iand(ix,z'0cc'),-2)
    ix2 = ishft(iand(ix,z'033'), 2)
    ix = ior(ix1,ix2)
    ix1 = ishft(ix,-4)
    ix2 = ishft(ix, 4)
    bitrev8 = iand(ior(ix1,ix2),z'0ff')
    end function bitrev8

    attributes(global) subroutine without( a, b )
    real(8), device :: a(*), b(*)
    i = blockDim%x*(blockIdx%x-1) + threadIdx%x
    j = bitrev8(threadIdx%x-1) + 1
    b(i) = a(j)
    return
    end subroutine

    attributes(global) subroutine withtex( a, b )
    real(8), device :: a(*), b(*)
    i = blockDim%x*(blockIdx%x-1) + threadIdx%x
    j = bitrev8(threadIdx%x-1) + 1
    b(i) = t(j)  ! This subroutine accesses a through the texture
    return
    end subroutine
end module memtests
```

```
program t
use cudafor
use memtests
real(8), device, target, allocatable :: da(:)
real(8), device, allocatable :: db(:)
integer, parameter :: n = 1024*1024
integer, parameter :: nthreads = 256
integer, parameter :: ntimes = 1000
type(cudaEvent) :: start, stop
real(8) b(n)

allocate(da(nthreads))
allocate(db(n))

istat = cudaEventCreate(start)
istat = cudaEventCreate(stop)

db = 100.0d0
da = (/ (dble(i),i=1,nthreads) /)

call without<<<n/nthreads, nthreads>>> (da, db)
istat = cudaEventRecord(start,0)
do j = 1, ntimes
  call without<<<n/nthreads, nthreads>>> (da, db)
end do
istat = cudaEventRecord(stop,0)
istat = cudaDeviceSynchronize()
istat = cudaEventElapsedTime(time1, start, stop)
time1 = time1 / (ntimes*1.0e3)
```

(continues on next page)

(continued from previous page)

```
b = db
print *,sum(b)==(n*(nthreads+1)/2)

db = 100.0d0
t => da  ! assign the texture to da using f90 pointer assignment

call withtex<<<n/nthreads, nthreads>>> (da, db)
istat = cudaEventRecord(start,0)
do j = 1, ntimes
  call withtex<<<n/nthreads, nthreads>>> (da, db)
end do
istat = cudaEventRecord(stop,0)
istat = cudaDeviceSynchronize()
istat = cudaEventElapsedTime(time2, start, stop)
time2 = time2 / (ntimes*1.0e3)
b = db
print *,sum(b)==(n*(nthreads+1)/2)

print *,"Time with    textures",time2
print *,"Time without textures",time1
print *,"Speedup with textures",time1 / time2

deallocate(da)
deallocate(db)
end
```

1.13. CUDA Fortran Conditional Compilation

This section describes several ways that CUDA Fortran can be enabled in your application while minimizing the changes made for maintaining a single CPU/GPU code base.

If CUDA Fortran is enabled in compilation, either by specifying `-cuda` on the command line or renaming the file with the `.cuf` or `.CUF` extension, then for a source line that begins with the `!@cuf` sentinel the rest of the line appears as a statement, otherwise the entire line is a comment.

If CUDA Fortran is enabled in compilation, either by specifying `-cuda` on the command line, and preprocessing is enabled by either the `-Mpreprocess` compiler option or by using capital letters in the filename extension (.CUF, .F90, etc.) then the `_CUDA` macro is defined.

If CUDA Fortran is enabled in compilation, then the CUF kernel directive (denoted by `!$cuf kernel`) will generate device code for that loop. Otherwise, the code will run on the CPU.

Variable renaming can be accomplished through a combination of the above techniques, and the `use ..., only:` Fortran statements to rename module variables. For instance, you could rename device arrays contained in a module with use device_declaration_mod, only : a => a_dev, b => b_dev in combination with either the CUF sentinel or the `_CUDA` macro. Fortran associate blocks can used similarly and offer more fine-grained control of variable renaming.

This example shows a number of these techniques, and can be compiled and run with or without CUDA Fortran enabled.

```
program p
!@cuf use cudafor
real a(1000)
```

(continues on next page)

(continued from previous page)

```
!@cuf attributes(managed) :: a
real b(1000)
!@cuf real, device :: b_dev(1000)
b = 2.0
!@cuf b_dev = b
!@cuf associate(b=>b_dev)
!$cuf kernel do(1) <<<*,*>>>
do i = 1, 1000
    a(i) = real(i) * b(i)
end do
!@cuf end associate
#ifdef _CUDA
print *,"GPU sum passed? ",sum(a).eq.1000*1001
#else
print *,"CPU sum passed? ",sum(a).eq.1000*1001
#endif
end program
```

1.14. Building a CUDA Fortran Program

CUDA Fortran is supported by the NVIDIA Fortran compiler when the filename uses a CUDA Fortran extension. The `.cuf` extension specifies that the file is a free-format CUDA Fortran program; the `.CUF` extension may also be used, in which case the program is processed by the preprocessor before being compiled. To compile a fixed-format program, add the command line option `-Mfixed`. CUDA Fortran extensions can be enabled in any Fortran source file by adding the `-cuda` command line option. It is important to remember that if you compile a file with the `-cuda` command line option, you must also link the file with the `-cuda` command line option. If you compile with `-cuda`, but do not link with `-cuda`, you will receive an undefined reference to the symbol cuda_compiled.

To change the version of the CUDA Toolkit used from the default, specify `-cuda -gpu=cudaX.Y`; CUDA Toolkit version `X.Y` must be installed.

Relocatable device code is generated by default. You can override this option by specifying `-cuda -gpu=nordc`.

If you are using many instances of the CUDA kernel loop directives, that is, CUF kernels, you may want to add the `-Minfo` switch to verify that CUDA kernels are being generated where you expect and whether you have followed the restrictions outlined in the preceding sections.

1.15. Managed and Unified Memory Options and Interoperability

CUDA Fortran is one of several GPU programming models available for Fortran developers. Other models make use of compiler options which CUDA Fortran developers may find useful. A general discussion of the memory models which are now supported can be found in the NVIDIA HPC Compiler User's Guide, available online at docs.nvidia.com/hpc-sdk.

One nvfortran compiler option that has been supported for many years is `-gpu=mem:managed`. This has been especially useful in the stdpar programming models. For CUDA Fortran, what this option

does is to use cudaMallocManaged() for all Fortran allocatable data, in essence treating allocatable arrays as though they have the managed attribute (See 2.6.2). This allows these arrays to be used in global subroutines, in CUF kernels, and to be passed into library functions which normally take device arrays.

A drawback of this is that the Fortran compiler can lose the information, as arrays are passed through levels of subroutines, that the array was originally allocatable. Therefore the managed attribute behavior can get lost; it works one way in the top-level functions but not in the leaf functions where you really want it. There are a few ways to work around this, but they are usually unwanted changes to the code. Unlike OpenACC or stdpar, CUDA Fortran has no implicit data movement. It is all explicit, under the control of the developer, through data attributes, assignment statements, and API calls like cudaMemcpy.

Starting with the 23.11 release, on systems which support HMM/ATS and unified memory, the NVHPC compilers now support an option named `-gpu=mem:unified`. This is similar to the managed option, but this applies to not just allocatable data, but all host data: allocatable, local stack data, and global static data. All program data can be accessed on the GPU. In some respect, all this option does for CUDA Fortran is removes compiler errors and warnings that host data is being used where device data is expected. All the low-level movement of data back-and-forth between CPU and GPU accesses is handled by the operating system and CUDA driver, and a separate host and device copy of the data is not required.

Of course, CUDA programs which have been tuned for two discrete memories, and that make use of asynchronous operations, multiple streams, and concurrent operation of CPUs and GPUs, may experience race conditions when using these options and there is now one copy of the data, not two. To help debug these issues, the `NVCOMPILER_ACC_SYNCHRONOUS` environment variable now accepts a bit field. Setting the value of this environment variable to 2 will insert a synchronization point at the end of each CUF kernel, and setting it to 4 will insert a synchronization point at the end of each global kernel launch.

Here is an example of a simple CUDA Fortran program that can now act on unified memory when compiled with the `-gpu=mem:unified` option:

```
module m1
integer, parameter :: N = 5
integer :: m(N)

contains
  attributes(global) subroutine g1( a )
    integer :: a(*)
    i = threadIdx%x
    if (i .le. N) a(i) = m(i)
    return
  end subroutine g1
end module m1

program t1
  use m1
  use cudafor
  integer :: istat, a(N)
  m = [ ((i),i=1,N) ]  ! Init global data
  call g1 <<<1,N>>> (a)
  istat = cudaDeviceSynchronize()
  print *,a(1:N)
end program t1
```

Note that we have added a call to `cudaDeviceSynchronize()`, as the unified data is read and written on the device, printed from the host, and global kernel launches are still asynchronous with respect to

the host.

In addition, starting with the 24.3 release, and also on systems which support HMM/ATS and unified memory, the NVHPC CUDA Fortran compiler supports the `unified` attribute. In general, data with the `unified` attribute behaves similarly to managed data, however it is allocated using system memory, not with cudaMallocManaged(). The attribute allows a programmer to enable unified memory on a variable-by-variable basis, and it does not require compiling with `-gpu=mem:unified`.

When using the Managed Memory Model, the Unified Memory Model, or the managed or unified attributes on variables, refer to *cudaMemAdvise* or *cudaMemPrefetchAsync* for memory hints which have been shown to improve application performance in many cases.

Chapter 2. Reference

This section is the CUDA Fortran Language Reference.

2.1. New Subroutine and Function Attributes

CUDA Fortran adds new attributes to subroutines and functions. This section describes how to specify the new attributes, their meaning and restrictions.

A Subroutine may have the host, global, or device attribute, or may have both host and device attribute. A Function may have the host or device attribute, or both. These attributes are specified using the `attributes(attr)` prefix on the Subroutine or Function statement; if there is no attributes prefix on the subprogram statement, then default rules are used, as described in the following sections.

2.1.1. Host Subroutines and Functions

The host attribute may be explicitly specified on the Subroutine or Function statement as follows:

```
attributes(host) subroutine sub(...)
attributes(host) integer function func(...)
integer attributes(host) function func(...)
```

The host attributes prefix may be preceded or followed by any other allowable subroutine or function prefix specifiers (recursive, pure, elemental, function return datatype). A subroutine or function with the host attribute is called a host subroutine or function, or a *host subprogram*. A host subprogram is compiled for execution on the host processor. A subprogram with no attributes prefix has the host attribute by default.

2.1.2. Global and Grid_Global Subroutines

The global and grid_global attribute may be explicitly specified on the Subroutine statement as follows:

```
attributes(global) subroutine sub(...)
```

```
attributes(grid_global) subroutine subg(...)
```

Functions may not have a global attribute. A subroutine with either global attribute is called a *kernel subroutine*. A kernel subroutine may not be recursive, pure, or elemental, so no other subroutine prefixes are allowed. A kernel subroutine is compiled as a kernel for execution on the device, to be called from a host routine using an execution configuration. A kernel subroutine may not be contained in another subroutine or function, and may not contain any other subprogram. A grid_global subroutine is supported on cc70 hardware or greater, and specifies that the kernel should be launched in such a way that all threads in the grid can synchronize.

Launch bounds can optionally be specified as part of the global subroutine definition to provide optimization hints to the compiler. This will mainly aid register usage, spilling, and occupancy heuristics used in the low-level code generation. See the CUDA C Programming Guide for more information. The form used in CUDA Fortran is:

```
attributes(global) launch_bounds(maxTPB, minBPM) subroutine sub(...)
```

where maxTPB is the maxThreadsPerBlock, the maximum number of threads per block with which the application will ever launch, and minBPM is the desired minimum number of resident blocks per multiprocessor. Both values must be numeric constants.

Beginning with the 23.3 release, support for thread block clusters is enabled for Hopper (cc90) and later targets. To specify the dimensions of the cluster, use the `cluster_dims` syntax and specify each x, y, and z dimension. Values must be numeric constants. See the CUDA C Programming Guide for more information. For instance, this example in CUDA Fortran:

```
attributes(global) cluster_dims(2,2,1) subroutine sub(...)
```

will set up a 2x2 (x and y) set of thread blocks in a cluster. The launch to these kernels using the chevron syntax will be adjusted appropriately at the call site.

Also, as part of the Hopper support, the `launch_bounds` syntax has been extended to accept a third argument, an upper bound on the cluster size.

2.1.3. Device Subroutines and Functions

The device attribute may be explicitly specified on the Subroutine or Function statement as follows:

```
attributes(device) subroutine sub(...)
attributes(device) datatype function func(...)
datatype attributes(device) function func(...)
```

A subroutine or function with the device attribute is called a *device subprogram*. A device subprogram is compiled for execution on the device, and can be called from a kernel subroutine or other device subprograms. A device subprogram may also be recursive, pure, or elemental. A subroutine or function with the device attribute can be in a different file or scope than the callers, but you must use relocatable device code linking, and provide an explicit interface. Otherwise, the device routines should be in the same module as the caller.

2.1.4. Restrictions on Kernel Subroutines and Device Subprograms

A subroutine or function with the device or global attribute must satisfy the following restrictions:

- It may not contain another subprogram.
- It may not be contained in another subroutine or function.
- A kernel subroutine may not be recursive, nor have the recursive prefix on the subroutine statement.
- A kernel subroutine may not be pure or elemental, nor have the pure or elemental prefix on the subroutine statement.

For more information, refer to *Device Code*.

2.2. Variable Attributes

CUDA Fortran adds new attributes for variables and arrays. This section describes how to specify the new attributes and their meaning and restrictions.

Variables declared in a host subprogram may have one of three new attributes: they may be declared to be in device global memory, in managed memory, or in pinned memory.

Variables in modules may be declared to be in device global memory, in the managed memory space, or in constant memory space.

Variables declared in a device program units may have one of three new attributes: they may be declared to be in device global memory, in constant memory space, in the thread block shared memory, or without any additional attribute they will be allocated in thread local memory. For performance and useability reasons, the value attribute can also be used on scalar dummy arguments so they are passed by value, rather than the Fortran default to pass arguments by reference.

2.2.1. Device data

A variable or array with the device attribute is defined to reside in the device global memory. The device attribute can be specified with the `attributes` statement, or as an attribute on the type declaration statement. The following example declares two arrays, a and b, to be device arrays of size 100.

```
real :: a(100)
attributes(device) :: a
real, device :: b(100)
```

These rules apply to device data:

- An allocatable device array dynamically allocates device global memory.
- Device variables and arrays may appear in modules, but may not be in a Common block or an Equivalence statement.
- Members of a derived type may not have the device attribute unless they are allocatable.

- Device variables and arrays may be passed as actual arguments to host and device subprograms; in that case, the subprogram interface must be explicit (in the Fortran sense), and the matching dummy argument must also have the device attribute.
- Device variables and arrays declared in a host subprogram cannot have the Save attribute unless they are allocatable.

In host subprograms, device data may only be used in the following manner:

- In declaration statements
- In Allocate and Deallocate statements
- As an argument to the Allocated intrinsic function
- As the source or destination in a data transfer assignment statement
- As an actual argument to a kernel subroutine
- As an actual argument to another host subprogram or runtime API call
- As a dummy argument in a host subprogram

A device array may have the allocatable attribute, or may have adjustable extent.

2.2.2. Managed data

A variable or array with the managed attribute is managed by the unified memory system and migrates between host main memory and device global memory. The managed attribute can be specified with the `attributes` statement, or as an attribute on the type declaration statement. Managed local arrays can have an explicit shape, be automatic, or allocatable. Managed dummy arrays may be assumed-size or assumed-shape. The following example declares two arrays, a and b, to be managed arrays of size 100, and allocates a third array, c with size 200.

```
real :: a(100)
attributes(managed) :: a
real, managed :: b(100)
real, allocatable, managed :: c(:)
. . .
allocate(c(200))
```

These rules apply to managed data on the host:

- Managed variables and arrays may appear in host subprograms and modules, but may not be in a Common block or an Equivalence statement.
- Managed variables and arrays declared in a host subprogram cannot have the Save attribute unless they are allocatable.
- Derived types may have the managed attribute.
- Members of a derived type may have the managed attribute.
- Managed derived types may also contain allocatable device arrays.
- Managed variables and arrays may be passed as actual arguments to other host subprograms; if the subprogram interface is overloaded, the generic matching priority is match another managed dummy argument first, match a dummy with the device attribute next, and match a dummy with no (or host) attribute last.

- Passing a non-managed actual argument to a managed dummy argument will result in either a compilation error if the interface is explicit, or unexpected behavior otherwise.
- Managed variables and arrays may be passed as actual arguments to global subroutines just as device variables and arrays are.
- By default, managed data is allocated with global scope, i.e. the flag passed to cudaMallocManaged is `cudaMemAttachGlobal`.
- The scope of a managed variable can be changed with a call to``cudaStreamAttachMemAsync``.
- Individual managed variables can be associated with a given stream by calling `cudaforSetDefaultStream`.
- All subsequently allocated managed variables can also be associated with a given stream by calling `cudaforSetDefaultStream`.
- Accessing managed data on the host while a running kernel is accessing managed data within the same scope on the device will result in either a segmentation fault or a race condition.

These rules apply to managed data on the device:

- The managed attribute may be used on dummy arguments.
- Managed data is treated as if it were device data.
- There is no support for allocating or deallocating managed data on the device.

Note: Even if your application only uses a single GPU, if you are running on systems which have multiple GPUs that are not peer-to-peer enabled, managed memory will be allocated as zero-copy memory and performance will suffer accordingly. A workaround is to set the environment variable `CUDA_VISIBLE_DEVICES` so only one GPU is seen, or to force allocation on the GPU by setting `CUDA_MANAGED_FORCE_DEVICE_ALLOC`. The CUDA C Programming Guide has more details on this in the Unified Memory Programing section.

2.2.3. Unified data

A variable or array with the unified attribute can be accessed from both host and device code. Whether the data migrates or is read and written across the memory bus is under the control of the CUDA driver and settings. The unified attribute can be specified with the `attributes` statement, or as an attribute on the type declaration statement. Unified arrays must be local, fixed-size, automatic or allocatable. The unified attribute is not allowed on global module arrays. A local F90 pointer can have the unified attribute, then point to a global array with the target attribute. This pointer can be passed as a kernel argument into a device code kernel, similar to unified memory support in CUDA C++.

These rules apply to unified data on the host:

- Unified variables and arrays may appear in host subprograms only, not in modules.
- Derived types may have the unified attribute.
- Members of a derived type may have the unified attribute.
- Unified derived types may also contain allocatable device arrays, which can be useful for deep data structures.

- Accessing unified data on the host while a running kernel is accessing it on the device may result in race conditions.

These rules apply to unified data on the device:

- Unified data is treated as if it were device data.
- There is no support for allocating or deallocating unified data on the device.

This table may help explain how matching of actual arguments to dummy arguments, when there are generic, overloaded interfaces exposed, are computed. For each argument pair, a distance is returned. The minimum distance, less than infinity, wins.

Table 1: Table 2. Attributed Argument Matching Distance Values

Dummy Argument	Actual None (host)	Actual Device	Actual Managed	Actual Unified	Actual OpenACC use_device	Actual None (gpu= mem: unified)	Actual None (gpu= mem: managed)
None (host)	0	INF	3	3	1	3	3
Device	INF	0	2	2	0	2	2
Managed	INF	INF	0	1	INF	1	0
Unified	INF	INF	1	0	INF	0	1

It should be noted that the Fortran host modules provided in CUDA Fortran, such as those provided for CUDA libraries discussed later in this chapter, contain interfaces in which the dummy arguments are either host or device. Therefore, the implementation which operates on device data is still preferred for actual arguments of either device, managed, or unified.

Also remember that the matching is based on how and where the data can be used. Care must be taken when allocating and deallocating the data, and generally the attributes must exactly match during those two operations. The pinned attribute discussed in the next section has the same requirement.

2.2.4. Pinned arrays

An allocatable array with the pinned attribute will be allocated in special page-locked host memory, when such memory is available. The advantage of using pinned memory is that transfers between the device and pinned memory are faster and can be asynchronous. An array with the pinned attribute may be declared in a module or in a host subprogram. The pinned attribute can be specified with the `attributes` statement, or as an attribute on the type declaration statement. The following example declares two arrays, `p` and `q`, to be pinned allocatable arrays.

```
real :: p(:)
allocatable :: p
attributes(pinned) :: p
real, allocatable, pinned :: q(:)
```

Pinned arrays may be passed as arguments to host subprograms regardless of whether the interface is explicit, or whether the dummy argument has the pinned and allocatable attributes. Where the array

is deallocated, the declaration for the array must still have the pinned attribute, or the deallocation may fail.

2.2.5. Constant data

A variable or array with the `constant` attribute is defined to reside in the device constant memory space. The constant attribute can be specified with the `attributes` statement, or as an attribute on the type declaration statement. The following example declares two arrays, `c` and `d`, to be constant arrays of size 100.

```
real :: c(100)
attributes(constant) :: c
real, constant :: d(100)
```

These rules apply to constant data:

- Constant variables and arrays can appear in modules, but may not be in a Common block or an `Equivalence` statement. Constant variables appearing in modules may be accessed via the `use` statement in both host and device subprograms.
- Constant data may not have the Pointer, Target, or Allocatable attributes.
- Members of a derived type may not have the constant attribute.
- Arrays with the `constant` attribute must have fixed size.
- Constant variables and arrays may be passed as actual arguments to host and device subprograms, as long as the subprogram interface is explicit, and the matching dummy argument also has the `constant` attribute. Constant variables cannot be passed as actual arguments between a host subprogram and a device global subprogram.
- Within device subprograms, variables and arrays with the `constant` attribute may not be assigned or modified.
- Within host subprograms, variables and arrays with the `constant` attribute may be read and written.

In host subprograms, data with the `constant` attribute may only be used in the following manner:

- As a named entity within a USE statement.
- As the source or destination in a data transfer assignment statement
- As an actual argument to another host subprogram
- As a dummy argument in a host subprogram

2.2.6. Shared data

A variable or array with the shared attribute is defined to reside in the shared memory space of a thread block. A shared variable or array may only be declared and used inside a device subprogram. The shared attribute can be specified with the `attributes` statement, or as an attribute on the type declaration statement. The following example declares two arrays, `s` and `t`, to be shared arrays of size 100.

```
real :: c(100)
attributes(shared) :: c
real, shared :: d(100)
```

These rules apply to shared data:

- Shared data may not have the Pointer, Target, or Allocatable attributes.
- Shared variables may not be in a Common block or Equivalence statement.
- Members of a derived type may not have the shared attribute.
- Shared variables and arrays may be passed as actual arguments to from a device subprogram to another device subprogram, as long as the interface is explicit and the matching dummy argument has the shared attribute.

Shared arrays that are not dummy arguments may be declared as assumed-size arrays; that is, the last dimension of a shared array may have an asterisk as its upper bound:

```
real, shared :: x(*)
```

Such an array has special significance. Its size is determined at run time by the call to the kernel. When the kernel is called, the value of the `bytes` argument in the execution configuration is used to specify the number of bytes of shared memory that is dynamically allocated for each thread block. This memory is used for the assumed-size shared memory arrays in that thread block; if there is more than one assumed-size shared memory array, they are all implicitly equivalenced, starting at the same shared memory address. Programmers must take this into account when coding.

Shared arrays may be declared as Fortran automatic arrays. For automatic arrays, the bounds are declared as an expression containing constants, parameters, blockdim variables, and integer arguments passed in by value. The allocation of automatic arrays also comes from the dynamic area specified via the chevron launch configuration. If more than one automatic array is declared, the compiler and runtime manage the offsets into the dynamic area. Programmers must provide a sufficient number of bytes in the chevron launch configuration shared memory value to cover all automatic arrays declared in the global subroutine.

```
attributes(global) subroutine sub(A, n,
integer, value :: n, nb
real, shared :: s(nb*blockdim%x,nb)
```

If a shared array is not a dummy argument and not assumed-size or automatic, it must be fixed size. In this case, the allocation for the shared array does not come from the dynamically allocated shared memory area specified in the launch configuration, but rather it is declared statically within the function. If the global routine uses only fixed size shared arrays, or none at all, no shared memory amount needs to be specified at the launch.

2.2.7. Texture data

Reading values through the texture memory interface is no longer recommended or necessary on newer GPUs and support for this feature has been dropped in CUDA 12.0.

Read-only real and integer device data can be accessed in device subprograms through the texture memory by assigning an F90 pointer variable to the underlying device array. To use texture memory in this manner, follow these steps:

1. Add a declaration to a module declaration section that contains the device code, such that the declaration is available to the device subprogram through host association, and available to the host code via either host or use association:

```
real, texture, pointer :: t(:)
```

2. In your host code, add the target attribute to the device data that you wish to access via texture memory:
   - Change: `real, device :: a(n)`
   - To: `real, target, device :: a(n)`

   The target attribute is standard F90/F2003 syntax to denote an array or other data structure that may be "pointed to" by another entity.

3. Tie the texture declaration to the device array by using the F90 pointer assignment operator in your host code. A simple expression like the following one performs all the underlying CUDA texture binding operations.

```
t => a
```

   The CUDA Fortran device code that can refer to `t` through host association can now access the elements of `t` without any change in syntax.

   In the following example, accesses of `t`, targeting `a`, go through the texture cache.

```
! Vector add, s through device memory, t is through texture memory
i = threadIdx%x + (blockIdx%x-1)*blockDim%x
s(i) = s(i) + t(i)
```

2.2.8. Value dummy arguments

In device subprograms, following the rules of Fortran, dummy arguments are passed by default by reference. This means the actual argument must be stored in device global memory, and the address of the argument is passed to the subprogram. Scalar arguments can be passed by value, as is done in C, by adding the value attribute to the variable declaration.

```
attributes(global) subroutine madd( a, b, n )
      real, dimension(n,n) :: a, b
      integer, value :: n
```

In this case, the value of n can be passed from the host without needing to reside in device memory. The variable arrays corresponding to the dummy arguments a and b must be set up before the call to reside on the device.

2.3. Allocating Device Memory, Pinned Memory, and Managed Memory

This section describes extensions to the Allocate statement, specifically for dynamically allocating device arrays, host pinned arrays, managed arrays, and other supported methods for allocating memory specific to CUDA Fortran.

2.3.1. Allocating Device Memory

Device arrays can have the allocatable attribute. These arrays are dynamically allocated in host subprograms using the Allocate statement, and dynamically deallocated using the Deallocate statement. If a device array declared in a host subprogram does not have the Save attribute, it will be automatically deallocated when the subprogram returns.

```
real, allocatable, device :: b(:)
allocate(b(5024),stat=istat)
...
if(allocated(b)) deallocate(b)
```

Scalar variables can be allocated on the device using the Fortran 2003 allocatable scalar feature. To use these, declare and initialize the scalar on the host as:

```
integer, allocatable, device :: ndev
allocate(ndev)
ndev = 100
```

The language also supports the ability to create the equivalent of automatic and local device arrays without using the allocate statement. These arrays will also have a lifetime of the subprogram as is usual with the Fortran language:

```
subroutine vfunc(a,c,n)
    real, device :: adev(n)
    real, device :: atmp(4)
    ...
end subroutine vfunc   ! adev and atmp are deallocated
```

Automatic and local arrays declared in this way, not containing the allocatable attribute, cannot have the Save attribute.

2.3.2. Allocating Device Memory Using Runtime Routines

For programmers comfortable with the CUDA C programming environment, Fortran interfaces to the CUDA memory management runtime routines are provided. These functions return memory which will bypass certain Fortran allocatable properties such as automatic deallocation, and thus the arrays are treated more like C malloc'ed areas. Mixing standard Fortran allocate/deallocate with the runtime Malloc/Free for a given array is not supported.

The cudaMalloc function can be used to allocate single-dimensional arrays of the supported intrinsic data-types, and cudaFree can be used to free it:

```
real, allocatable, device :: v(:)
istat = cudaMalloc(v, 100)
...
istat = cudaFree(v)
```

For a complete list of the memory management runtime routines, refer to *Memory Management*.

2.3.3. Allocate Pinned Memory

Allocatable arrays with the pinned attribute are dynamically allocated using the Allocate statement. The compiler will generate code to allocate the array in host page-locked memory, if available. If no such memory space is available, or if it is exhausted, the compiler allocates the array in normal paged host memory. Otherwise, pinned allocatable arrays work and act like any other allocatable array on the host.

```
real, allocatable, pinned :: p(:)
allocate(p(5000),stat=istat)
...
if(allocated(p)) deallocate(p)
```

To determine whether or not the allocation from page-locked memory was successful, an additional PINNED keyword is added to the allocate statement. It returns a logical success value.

```
logical plog
allocate(p(5000), stat=istat, pinned=plog)
if (.not. plog) then
. . .
```

2.3.4. Allocating Managed Memory

Managed arrays may or may not have the allocatable attribute. These arrays are all dynamically allocated just as device arrays are.

```
real, allocatable, managed :: b(:)
allocate(b(5024),stat=istat)
...
if(allocated(b)) deallocate(b)
```

CUDA Fortran supports the ability to create the equivalent of automatic and local managed arrays without using the allocate statement. These arrays will also have a lifetime of the subprogram as is usual with the Fortran language:

```
subroutine vfunc(a,c,n)
    real, managed :: aman(n)
    real, managed :: atmp(4)
    ...
end subroutine vfunc   ! aman and atmp are deallocated
```

2.3.5. Allocating Managed Memory Using Runtime Routines

The cudaMallocManaged function can be used to allocate single-dimensional managed arrays of the supported intrinsic data-types, and cudaFree can be used to free it:

```
use cudafor
real, allocatable, managed :: v(:)
istat = cudaMallocManaged(v, 100, cudaMemAttachHost)
...
istat = cudaFree(v)
```

For a complete list of the memory management runtime routines, refer to *Memory Management*.

2.3.6. Allocating Device Memory Asynchronously

Beginning in CUDA 11.2, allocatable device arrays can be dynamically allocated in host subprograms using the Allocate statement, asynchronously, on a specified stream.

```
real, allocatable, device :: b(:)
integer(kind=cuda_stream_kind) :: istream
...
allocate(b(5024),stream=istream)
```

These arrays can also be dynamically deallocated using the Deallocate statement. It is not necessary, or allowed, to specify a stream during deallocation. If a device array declared in a host subprogram does not have the Save attribute, it will be automatically deallocated when the subprogram returns. Given the allocation above, this statement will deallocate the array b on the stream specified by istream.

```
if(allocated(b)) deallocate(b)
```

Arrays declared using the Allocate statement with a stream are associated with that stream as if the `cudaforSetDefaultStream` function were called for that combination of device data and stream. To use this data in operations outside of this stream, users should call `cudaStreamSynchronize` first to block host execution until all stream operations have completed.

2.3.7. Allocating Device Memory Asynchronously Using Runtime Routines

The cudaMallocAsync function can be used to allocate single-dimensional arrays of the supported intrinsic data-types, and cudaFreeAsync can be used to free it, asynchronously, on a given stream:

```
real, allocatable, device :: v(:)
integer(kind=cuda_stream_kind) :: istream
istat = cudaMallocAsync(v, 100, istream)
...
istat = cudaFreeAsync(v, istream)
```

For a complete list of the memory management runtime routines, refer to *Memory Management*.

2.3.8. Controlling Device Data is Managed

Beginning in the HPC SDK compiler version 21.9, it is possible to change the CUDA Fortran device data allocation behavior to actually allocate managed memory instead of device memory, with potentially no coding changes.

This can be useful in order to oversubscribe the available GPU memory, and allow the OS and driver to page memory to and from the GPU as needed, either as an experiment or for running larger problem sizes than normally available.

All CUDA Fortran device allocations go through a small wrapper layer before making the actual CUDA API call. By setting the environment variable

```
NVCOMPILER_CUDAFOR_DEVICE_IS_MANAGED=1
```

allocations in the form of the first two subsections in this section, `Allocating Device Memory` and `Allocating Device Memory Using Runtime Routines` will eventually call cudaMallocManaged rather than cudaMalloc. In addition, some prefetching hints are added to make the accesses to the newly allocated data most efficient from the GPU (the current device).

2.4. Data transfer between host and device memory

This section provides methods to transfer data between the host and device memory.

2.4.1. Data Transfer Using Assignment Statements

You can copy variables and arrays from the host memory to the device memory by using simple assignment statements in host subprograms. By default, using assignment statements to read or write device, managed, or constant data implicitly uses CUDA stream zero. This means such data copies are synchronous, and the data copy waits until all previous kernels and data copies complete. Alternatively, you can use the `cudaforSetDefaultStream` call to associate one or more device and managed variables to a particular stream. After this call has occurred, assignment statements on those variables will run asynchronously on the specified stream.

Specific information on assignment statements:

- An assignment statement where the left hand side is a device variable or device array or array section, and the right hand side is a host variable or host array or array section, copies data from the host memory to the device global memory.
- An assignment statement where the left hand side is a host variable or host array or array section, and the right hand side is a device variable or device array or array section, copies data from the device global memory to the host memory.
- An assignment statement with a device variable or device array or array section on both sides of the assignment statement copies data between two device variables or arrays.

Similarly, you can use simple assignment statements to copy or assign variables or arrays with the constant attribute.

Specific information on assignment statements and managed data:

- An assignment statement where the left hand side is a managed variable or managed array, and the right hand side is a conforming scalar constant, host variable, host array or array section, copies data from the host memory to the device global memory using cudaMemcpy, memset, or a similar operation.
- An assignment statement where the left hand side is a managed array section and the right hand side is any host variable copies data using generated host code.
- An assignment statement where the left hand side is a managed variable, managed array or array section, and the right hand side is a device variable or device array or array section, copies data from the device global memory to the host memory using cudaMemcpy or a similar operation.
- An assignment statement where the right hand side is a managed variable or managed array, and the left hand side is a host variable, host array or array section, copies data from the device global memory to the host memory using cudaMemcpy or a similar operation.
- An assignment statement where the right hand side is a managed array section and the left hand side is any host or managed variable copies data using generated host code.
- An assignment statement where the right hand side is a managed variable, managed array or array section, and the left hand side is a device variable or device array or array section, copies data using cudaMemcpy and accesses the data from the device.

More information on Memcpy and Memset behavior with managed memory can be found in the Unified Memory Programing section of the *CUDA C Programming Guide*.

2.4.2. Implicit Data Transfer in Expressions

Some limited data transfer can be enclosed within expressions. In general, the rule of thumb is all arithmetic or operations must occur on the host, which normally only allows one device array to appear on the right-hand-side of an expression. Temporary arrays are generated to accommodate the host copies of device data as needed. For instance, if `a`, `b`, and `c` are conforming host arrays, and `adev`, `bdev`, and `cdev` are conforming device arrays, the following expressions are legal:

```
a = adev
```

```
adev = a
```

```
b = a + adev
```

```
c = x * adev + b
```

The following expressions are not legal as they either promote a false impression of where the actual computation occurs, or would be more efficient written in another way, or both:

```
c = adev + bdev
```

```
adev = adev + a
```

```
b = sqrt(adev)
```

Elemental transfers are supported by the language but perform poorly. Array slices are also supported, and their performance is dependent on the size of the slice, the amount of contiguous data in the slices, and the implementation.

2.4.3. Data Transfer Using Runtime Routines

For programmers comfortable with the CUDA C programming environment, Fortran interfaces to the CUDA memory management runtime routines are provided. These functions can transfer data either from the host to device, device to host, or from one device array to another.

The `cudaMemcpy` function can be used to copy data between the host and the GPU:

```
real, device :: wrk(1024)
real cur(512)
istat = cudaMemcpy(wrk, cur, 512)
```

For those familiar with the CUDA C routines, the kind parameter to the Memcpy routines is optional in Fortran because the attributes of the arrays are explicitly declared. Counts expressed in arguments to the Fortran runtime routines are expressed in terms of data type elements, not bytes.

For a complete list of memory management runtime routines, refer to *Memory Management*.

2.5. Invoking a kernel subroutine

A call to a kernel subroutine must give the execution configuration for the call. The execution configuration gives the size and shape of the grid and thread blocks that execute the function as well as the amount of shared memory to use for assumed-size shared memory arrays and the associated stream.

The execution configuration is specified after the subroutine name in the call statement; it has the form:

```
<<< grid, block, bytes, stream >>>
```

- `grid` is an integer, a value of `type(dim3)`, or `*`. If grid is an integer, it is converted to `dim3(grid, 1,1)`. If it is `type(dim3)`, the product `grid%x*grid%y*grid%z` gives the number of thread blocks to launch. This product must be less than or equal to the maximum number of blocks supported by the device. Launching a grid_global subroutine kernel puts further restrictions on the number of blocks. Setting the grid to `*` instructs the runtime to compute the number of blocks via a call to `cudaOccupancyMaxActiveBlocksPerMultiprocessor()`, which takes grid_global (or not) into account. Setting a single grid dim3 x, y, or z value to -1 also takes this same querying path through the runtime.
- `block` is an integer, or of `type(dim3)`. If it is `type(dim3)`, the number of threads per thread block is `block%x*block%y*block%z`, which must be less than or equal to the maximum supported by the device. If `block` is an integer, it is converted to `dim3(block,1,1)`.
- `bytes` is optional; if present, it must be a scalar integer, and specifies the number of bytes of shared memory to be allocated for each thread block to use for assumed-size shared memory arrays. For more information, refer to *Shared Data*. If not specified, the value zero is used.
- stream is optional; if present, it must be an integer, and have a value of zero, or a value returned by a call to cudaStreamCreate. See Section 4.5 on page 41. It specifies the stream to which this call is enqueued. The stream constant value `cudaStreamPerThread` may be specified. This will use a unique stream for each CPU thread.

For instance, a kernel subroutine

```
attributes(global) subroutine sub( a )
```

can be called like:

```
call sub <<< DG, DB, bytes >>> ( A )
```

The function call fails if the `grid` or `block` arguments are greater than the maximum sizes allowed, or if `bytes` is greater than the shared memory available. Shared memory may also be consumed by fixed-sized shared memory declarations in the kernel and for other dedicated uses, such as function arguments and execution configuration arguments.

2.6. Device code

2.6.1. Datatypes Allowed

Variables and arrays with the device, constant, or shared attributes, or declared in device subprograms, are limited to the types described in this section. They may have any of the intrinsic datatypes in the following table.

Table 2: Table 3. Device Code Intrinsic Datatypes

Type	Type Kind
`integer`	1,2,4(default),8
`logical`	1,2,4(default),8
`real`	2,4(default),8
`double precision`	equivalent to `real(kind=8)`
`complex`	4(default),8
`character(len=1)`	1 (default)

Additionally, they may be of derived type, where the members of the derived type have one of the allowed intrinsic datatypes, or another allowed derived type.

The system module `cudafor` includes definitions of the derived type `dim3`, defined as

```
type(dim3)
    integer(kind=4) :: x,y,z
end type
```

2.6.2. Built-in Variables

Several CUDA Fortran read-only predefined variables are available in device code. They are declared as follows:

```
type(dim3) :: threadidx, blockdim, blockidx, griddim
integer(4), parameter :: warpsize = 32
```

- The variable `threadidx` contains the thread index within its thread block; for one- or two-dimensional thread blocks, the `threadidx%y` and/or `threadidx%z` components have the value one.
- The variable `blockdim` contains the dimensions of the thread block; `blockdim` has the same value for all threads in the same grid; for one- or two-dimensional thread blocks, the `blockdim%y` and/or `blockdim%z` components have the value one.
- The variable `blockidx` contains the block index within the grid; as with `threadidx`, for one-dimensional grids, `blockidx%y` has the value one. The value of `blockidx%z` is always one. The value of `blockidx` is the same for all threads in the same thread block.
- The variable `griddim` contains the dimensions of the grid. The value of `griddim` is the same for all threads in the same grid; the value of `griddim%y` and `griddim%z` is one for one-dimensional grids.
- The variables `threadidx`, `blockdim`, `blockidx`, and `griddim` are available only in device subprograms.
- The constant `warpsize` contains the number of threads in a warp. It is currently defined to be 32.

2.6.3. Fortran Intrinsics

This section lists the Fortran intrinsic functions allowed in device subprograms.

The use of system module `wmma` is required to call mathematical and some numeric intrinsics using `real(2)` data type. Information about which intrinsics are only available via wmma module can be found in *WMMA Module* description section.

Table 3: Table 4. Fortran Numeric and Logical Intrinsics

Name	Argument Datatypes		Name	Argument Datatypes
`abs`	integer, real(2,4,8), complex		`int`	integer, real(2,4,8), complex
`aimag`	complex		`logical`	logical
`aint`	real(4,8)		`max`	integer, real(2,4,8)
`anint`	real(4,8)		`min`	integer, real(2,4,8)
`ceiling`	real(4,8)		`mod`	integer, real(4,8)
`cmplx`	real(2,4,8) or (real,real)		`modulo`	integer, real(4,8)
`conjg`	complex		`nint`	real(4,8)
`dim`	integer, real(4,8)		`real`	integer, real(2,4,8), complex
`floor`	real(4,8)		`sign`	integer, real(4,8)

Table 4: Table 5. Fortran Mathematical Intrinsics

Name	Argument Datatypes		Name	Argument Datatypes
acos	real(2,4,8)		cosh	real(2,4,8)
acosh	real(4,8)		erf	real(4,8)
asin	real(2,4,8)		erfc	real(4,8)
asinh	real(4,8)		exp	real(2,4,8), complex
atan	real(2,4,8)		gamma	real(4,8)
atanh	real(4,8)		hypot	(real(4,8),real(4,8))
atan2	(real,real)		log	real(2,4,8), complex
bessel_j0	real(4,8)		log10	real(2,4,8)
bessel_j1	real(4,8)		log_gamma	real(4,8)
bessel_jn	(int,real(4,8))		sin	real(2,4,8), complex
bessel_y0	real(4,8)		sinh	real(2,4,8)
bessel_y1	real(4,8)		sqrt	real(2,4,8), complex
bessel_yn	(int,real(4,8))		tan	real(2,4,8)
cos	real(2,4,8), complex		tanh	real(2,4,8)

Table 5: Table 6. Fortran Numeric Inquiry Intrinsics

Name	Argument Datatypes		Name	Argument Datatypes
bit_size	integer		precision	real(2,4,8), complex
digits	integer, real(2,4,8)		radix	integer, real(2,4,8)
epsilon	real(2,4,8)		range	integer, real(2,4,8), complex
huge	integer, real(2,4,8)		selected_int_kind	integer
maxexponent	real(2,4,8)		selected_real_kind	(integer,integer)
minexponent	real(2,4,8)		tiny	real(2,4,8)

Table 6: Table 7. Fortran Bit Manipulation Intrinsics

Name	Argument Datatypes		Name	Argument Datatypes
btest	integer		ishft	integer
iand	integer		ishftc	integer
ibclr	integer		leadz	integer
ibits	integer		mvbits	integer
ibset	integer		not	integer
ieor	integer		popcnt	integer
ior	integer		poppar	integer

Table 7: Table 8. Fortran Reduction and Array Intrinsics

Name	Argument Datatypes		Name	Argument Datatypes
`all`	logical		`maxval`	integer, real(2,4,8)
`any`	logical		`minloc`	integer, real(4,8)
`count`	logical		`minval`	integer, real(2,4,8)
`dot_product`	real(4,8)		`norm2`	real(4,8)
`matmul`	real(4,8), complex		`product`	integer, real(4,8), complex
`maxloc`	integer, real(4,8)		`sum`	integer, real(4,8), complex

2.6.4. Synchronization Functions

This section describes the synchronization functions and subroutines supported in device subprograms.

Synchronization Functions

The synchronization functions control the synchronization of various threads during execution of thread blocks.

- `syncthreads`
- `syncthreads_count`
- `syncthreads_and`
- `syncthread_or`
- `syncwarp`
- `threadfence`
- `threadfence_block`
- `threadfence_system`

For detailed information on these functions, refer to *Thread Management*.

SYNCTHREADS

The `syncthreads` intrinsic subroutine acts as a barrier synchronization for all threads in a single thread block; it has no arguments:

```
subroutine syncthreads()
```

Sometimes threads within a block access the same addresses in shared or global memory, thus creating potential read-after-write, write-after-read, or write-after-write hazards for some of these memory accesses. To avoid these potential issues, use `syncthreads()` to specify synchronization points in the kernel. This intrinsic acts as a barrier at which all threads in the block must wait before any thread is allowed to proceed. Threads within a block cooperate and share data by synchronizing their execution to coordinate memory accesses.

Each thread in a thread block pauses at the `syncthreads` call until all threads have reached that call. If any thread in a thread block issues a call to `syncthreads`, all threads must also reach and execute the same call statement, or the kernel fails to complete correctly.

SYNCTHREADS_AND

```
integer syncthreads_and(int_value)
```

syncthreads_and. like syncthreads, acts as a barrier at which all threads in the block must wait before any thread is allowed to proceed. In addition, syncthreads_and evaluates the integer argument *int_value* for all threads of the block and returns non-zero if and only if *int_value* evaluates to non-zero for *all* of them.

SYNCTHREADS_COUNT

```
integer syncthreads_count(int_value)
```

syncthreads_count, like syncthreads, acts as a barrier at which all threads in the block must wait before any thread is allowed to proceed. In addition, syncthreads_count evaluates the integer argument *int_value* for all threads of the block and returns the number of threads for which *int_value* evaluates to non-zero.

SYNCTHREADS_OR

```
integer syncthreads_or(int_value)
```

syncthreads_or, like syncthreads, acts as a barrier at which all threads in the block must wait before any thread is allowed to proceed. In addition, syncthreads_or evaluates the integer argument *int_value* for all threads of the block and returns non-zero if and only if *int_value* evaluates to non-zero for *any* of them.

SYNCWARP

```
subroutine syncwarp(int_mask)
```

syncwarp will cause all executing threads witin a warp, and specified in the mask argument, to reach a barrier, at which point all threads in the mask must execute syncwarp before any thread is allowed to proceed.

Memory Fences

In general, when a thread issues a series of writes to memory in a particular order, other threads may see the effects of these memory writes in a different order. You can use threadfence(), threadfence_block(), and threadfence_system() to create a *memory fence* to enforce ordering.

For example, suppose you use a kernel to compute the sum of an array of N numbers in one call. Each block first sums a subset of the array and stores the result in global memory. When all blocks are done, the last block done reads each of these partial sums from global memory and sums them to obtain the final result. To determine which block is finished last, each block atomically increments a counter to signal that it is done with computing and storing its partial sum. If no fence is placed between storing the partial sum and incrementing the counter, the counter might increment before the partial sum is stored.

THREADFENCE

```
subroutine threadfence()
```

threadfence acts as a memory fence, creating a wait. Typically, when a thread issues a series of writes to memory in a particular order, other threads may see the effects of these memory writes in a different order. threadfence() is one method to enforce a specific order. All global and shared memory accesses made by the calling thread prior to threadfence() are visible to:

- All threads in the thread block for shared memory accesses
- All threads in the device for global memory accesses

THREADFENCE_BLOCK

```
subroutine threadfence_block()
```

`threadfence_block` acts as a memory fence, creating a wait until all global and shared memory accesses made by the calling thread prior to `threadfence_block()` are visible to all threads in the thread block for all accesses.

THREADFENCE_SYSTEM

```
subroutine threadfence_system()
```

`threadfence_system` acts as a memory fence, creating a wait until all global and shared memory accesses made by the calling thread prior to `threadfence_system()` are visible to:

- All threads in the thread block for shared memory accesses
- All threads in the device for global memory accesses
- Host threads for page-locked host memory accesses

`threadfence_system()` is only supported by devices of compute capability 2.0 or higher.

2.6.5. Warp-Vote Operations

New warp-vote and warp match operations have been added to NVIDIA CUDA Fortran. The older versions remain for legacy reasons; they will invoke the newer functionality with a mask specifying all threads in the warp.

ALLTHREADS

The `allthreads` function is a warp-vote operation with a single scalar logical argument:

```
if( allthreads(a(i)<0.0) ) allneg = .true.
```

The function `allthreads` evaluates its argument for all threads in the current warp. The value of the function is `.true.` only if the value of the argument is `.true.` for all threads in the warp.

ANYTHREAD

The `anythread` function is a warp-vote operation with a single scalar logical argument:

```
if( anythread(a(i)<0.0) ) allneg = .true.
```

The function `anythread` evaluates its argument for all threads in the current warp. The value of the function is `.false.` only if the value of the argument is `.false.` for all threads in the warp.

BALLOT

The `ballot` function is a warp-vote operation with a single integer argument:

```
unsigned integer ballot(int_value)
```

The function `ballot` evaluates the argument `int_value` for all threads of the warp and returns an integer whose Nth bit is set if and only if `int_value` evaluates to non-zero for the Nth thread of the warp.

This function is only supported by devices of compute capability 2.0.

Example:

```
if( ballot(int_value) ) allneg = .true.
```

ACTIVEMASK

```
unsigned integer activemask()
```

The `activemask` function returns a 32-bit integer mask of all the currently active threads in the calling warp. The Nth bit is set if the Nth lane in the warp is active when `activemask` is called.

ALL_SYNC

```
integer all_sync(int_mask, int_predicate)
```

The `all_sync` function evaluates the predicate argument for all non-exited threads in the mask and returns non-zero if the predicate is non-zero for all threads.

ANY_SYNC

```
integer any_sync(int_mask, int_predicate)
```

The `any_sync` function evaluates the predicate argument for all non-exited threads in the mask and returns non-zero if the predicate is non-zero for any of them.

BALLOT_SYNC

```
unsigned integer ballot_sync(int_mask, int_predicate)
```

The `ballot_sync` function evaluates the predicate argument for all non-exited threads set by the mask in the calling warp. The Nth bit is set in the Nth lane if the predicate is non-zero for the Nth thread.

MATCH_ALL_SYNC

```
unsigned integer match_all_sync(int_mask, value, int_predicate)
```

The `match_all_sync` function performs a broadcast and compare of the value for all threads within a warp specified by the mask argument. It returns `int_mask` if all threads have the same value, otherwise 0. The `int_predicate` is set to true in the former case, false in the latter. This function currently accepts the type of `value` to be integer(4), integer(8), real(4), or real(8).

MATCH_ANY_SYNC

```
unsigned integer match_any_sync(int_mask, value)
```

The `match_any_sync` function performs a broadcast and compare of the value for all threads within a warp specified by the mask argument. It returns a mask of threads that have the same value as `value`. This function currently accepts the type of `value` to be integer(4), integer(8), real(4), or real(8).

2.6.6. Load and Store Functions Using Cache Hints

These load and store functions can provide finer control over the caching behavior and act as optimization hints. They do not change the memory consistency behavior of the program. These functions and subroutines can operate on most supported data types, including integer(4), integer(8), real(2), real(4), real(8), complex(4), and complex(8). There is also support for integer(4) and real(4) of dimension(4), and integer(8) and real(8) of dimension(2), i.e. 128-bit loads and stores..

The cache load functions are:

Table 8: Table 9. Load Functions Using Cache Hints

Function	Caching Behavior
`value = __ldca(mem)`	`Cache at all levels`
`value = __ldcg(mem)`	`Cache at global level`
`value = __ldcs(mem)`	`Cache streaming, accessed once`
`value = __ldlu(mem)`	`Last use`
`value = __ldcv(mem)`	`Don't cache, treat as volatile`

The cache store subroutines are:

Table 9: Table 10. Store Subroutines Using Cache Hints

Subroutine	Caching Behavior
`call __stwb(mem, value)`	`Cache write-back all coherent levels`
`call __stcg(mem, value)`	`Cache at global level`
`call __stcs(mem, value)`	`Cache streaming, accessed once`
`call __stwt(mem, value)`	`Cache write-through`

2.6.7. Load and Store Functions Using Bulk TMA Operations

Starting with the NVHPC 25.3 release, initial support for asynchronous bulk load and store operations is available in CUDA Fortran. The TMA operations listed in this section are supported on Hopper (cc90) and newer architectures. The load operations are typically issued by a single CUDA thread, and load data into a section of shared memory. Barriers are needed to ensure the data has arrived and can be safely accessed. These barriers are declared as integer(8), shared variables, and are often the first argument to the runtime API calls. The store operations are similarly launched by a single CUDA thread, but do not require a barrier. Here is a simple vector add operation in CUDA Fortran using these new library functions.

```
attributes(global) subroutine stream_add(c, a, b, n)
  integer, value :: n
  real(8), device :: c(n), a(n), b(n)
  real(8), shared :: tmpa(1024), tmpb(1024)
```

(continues on next page)

(continued from previous page)

```
  integer(8), shared :: barrier1, barrier2
  integer(8) :: token1, token2
  integer(4) :: j, elem_count

  j = threadIdx%x + (blockIdx%x-1) * 1024
  if (threadIdx%x == 1) then
    call barrier_init(barrier1, blockDim%x)
    call barrier_init(barrier2, blockDim%x)
  end if
  call syncthreads()  ! All threads see SM barriers

  ! First thread does the bulk load, sets element count
  if (threadIdx%x == 1) then
    elem_count = min(1024, n-j+1)
    call tma_bulk_load(barrier1, a(j), tmpa, elem_count)
    call tma_bulk_load(barrier2, b(j), tmpb, elem_count)
  end if
  call syncthreads()
  token1 = barrier_arrive(barrier1)  ! All threads arrive
  token2 = barrier_arrive(barrier2)  ! Return a token to wait upon

  ! These runtime functions spin and wait, return 1 on success
  if (barrier_try_wait(barrier1, token1) .eq. 0) return
  if (barrier_try_wait(barrier2, token2) .eq. 0) return

  do i = threadIdx%x, 1024, blockDim%x
    tmpa(i) = tmpa(i) + tmpb(i)
  end do
  call fence_proxy_async()  ! Ensure readiness for store
  call syncthreads()

  if (threadIdx%x == 1) then
    call tma_bulk_store(tmpa, c(j), elem_count)
  end if
  return
end subroutine
```

Another version of the load and store functions take the count in terms of bytes, and can take any type for the source and destination. There are also wait functions that the user can provide the spin loop for:

```
j = threadIdx%x + (blockIdx%x-1) * 1024
if (threadIdx%x == 1) then
  call barrier_init(barrier1, blockDim%x)
  call barrier_init(barrier2, blockDim%x)
end if
call syncthreads()

! First thread does the bulk load, sets byte count
if (threadIdx%x == 1) then
  tx_count = min(1024*8, (n-j+1)*8)
  call tma_bulk_g2s(barrier1, a(j), tmpa, tx_count)
  call tma_bulk_g2s(barrier2, b(j), tmpb, tx_count)
else
  ! Other threads have a byte count of zero
  tx_count = 0
```

(continues on next page)

(continued from previous page)

```
end if
call syncthreads()
token1 = barrier_arrive(barrier1, tx_count)
token2 = barrier_arrive(barrier2, tx_count)

! Loop until condition
do
  if (barrier_try_wait_sleep(barrier1, token1, 1000000) .ne. 0) exit
end do

do
  if (barrier_try_wait_sleep(barrier2, token2, 1000000) .ne. 0) exit
end do

do i = threadIdx%x, 1024, blockDim%x
  tmpa(i) = tmpa(i) + tmpb(i)
end do
call fence_proxy_async()
call syncthreads()

if (threadIdx%x == 1) then
  call tma_bulk_s2g(tmpa, c(j), tx_count)
end if
```

There are other possible variations of these calls which may be added in future releases. The set that is supported initially are summarized in the following tables.

Table 10: Table 11. TMA Subroutines

Subroutine	Operation
`call barrier_init(barrier, count)`	Initialize the barrier object with the number of threads participating
`call tma_bulk_g2s(barrier, src, dst, nbytes)`	General bulk load from global memory to shared memory, count is in bytes
`call tma_bulk_load(barrier, src, dst, nelems)`	Type-specific bulk load from global mem to shared mem, count is in elements
`call tma_bulk_s2g(src, dst, nbytes)`	General bulk store from shared memory to global memory, count is in bytes
`call tma_bulk_store(src, dst, nelems)`	Type-specific bulk store from shared mem to global mem, count is in elements
`call fence_proxy_async()`	Synchronize shared memory and TMA engine; also called as part of barrier_init()
`call tma_bulk_commit_group()`	Called as part of these bulk store operations
`call tma_bulk_wait_group()`	Called as part of these bulk store operations

Table 11: Table 12. TMA Functions

Function	Operation
`token = barrier_arrive(barrier)`	All threads arrive on the barrier which returns an integer(8) token
`token = barrier_arrive(barrier, count)`	Alternate form in which the user provides the transaction count
`istat = barrier_try_wait(barrier, token)`	Call to wait in the runtime function for data to arrive
`istat = barrier_try_wait_sleep(barrier, token, ns)`	Call to wait with a specified time in nanoseconds, user provides wait loop

2.6.8. Atomic Functions

The atomic functions read and write the value of their first operand, which must be a variable or array element in shared memory (with the shared attribute) or in device global memory (with the device attribute). Atomic functions are only supported by devices with compute capability 1.1 and higher. Compute capability 1.2 or higher is required if the first argument has the shared attribute. Certain real(4) and real(8) atomic functions may require compute capability 2.0 and higher.

The atomic functions return correct values even if multiple threads in the same or different thread blocks try to read and update the same location without any synchronization.

Arithmetic and Bitwise Atomic Functions

These atomic functions read and return the value of the first argument. They also combine that value with the value of the second argument, depending on the function, and store the combined value back to the first argument location. For atomicadd, atomicsub, atomicmax, atomicmin, and atomicexch, the data types may be integer(4), integer(8), real(4), or real(8). For atomicand, atomicor, and atomicxor, only integer(4) arguments are supported.

Note: The return value for each of these functions is the first argument, `mem`.

These functions are:

Table 12: Table 13. Arithmetic and Bitwise Atomic Functions

Function	Additional Atomic Update
`atomicadd( mem, value )`	`mem = mem + value`
`atomicsub( mem, value )`	`mem = mem - value`
`atomicmax( mem, value )`	`mem = max(mem,value)`
`atomicmin( mem, value )`	`mem = min(mem,value)`
`atomicand( mem, value )`	`mem = iand(mem,value)`
`atomicor( mem, value )`	`mem = ior(mem,value)`
`atomicxor( mem, value )`	`mem = ieor(mem,value)`
`atomicexch( mem, value )`	`mem = value`

Counting Atomic Functions

These atomic functions read and return the value of the first argument. They also compare the first argument with the second argument, and stores a new value back to the first argument location, depending on the result of the comparison. These functions are intended to implement circular counters, counting up to or down from a maximum value specified in the second argument. Both arguments must be of type integer(kind=4).

Note: The return value for each of these functions is the first argument, `mem`.

These functions are:

Table 13: Table 14. Counting Atomic Functions

Function	Additional Atomic Update

`atomicinc( mem, imax )`	`if (mem<imax) then`
	`  mem = mem+1`
	`else`
	`  mem = 0`
	`endif`

`atomicdec( mem, imax )`	`if (mem<imax .and. mem>0) then`
	`  mem = mem-1`
	`else`
	`  mem = imax`
	`endif`

Compare and Swap Atomic Function

This atomic function reads and returns the value of the first argument. It also compares the first argument with the second argument, and atomically stores a new value back to the first argument location if the first and second argument are equal. All three arguments must be of the same type, either integer(kind=4), integer(kind=8), real(kind=4), or real(kind=8).

Note: The return value for this function is the first argument, `mem`.

The function is:

Table 14: Table 15. Compare and Swap Atomic Function

Function	Additional Atomic Update

`atomiccas(mem,comp,val)`	`if (mem == comp) then`
	`mem = val`
	`endif`

2.6.9. Fortran I/O

The NVIDIA Fortran compiler includes limited support for `PRINT` statements in GPU device code. The Fortran GPU runtime library, which is shared between CUDA Fortran and OpenACC for NVIDIA GPU targets, buffers up the output and prints an entire line in one operation. Integer, character, logical, real and complex data types are supported.

The underlying CUDA `printf` implementation limits the number of print statements in a kernel launch to 4096. Users should take this limit into account when making use of this feature.

2.6.10. PRINT Example

By adding the compiler option `-cuda=charstring`, some limited support for character strings, character substrings, character variables, and string assignment is also now available in CUDA Fortran device code. Here is a short example:

```
attributes(global) subroutine printtest()
character*12 c
i = threadIdx%x
if (i/2*2.eq.i) then
  c = "Even Thread:"
else
  c = " Odd Thread:"
endif
print *,c,c(6:11),i
end subroutine
```

2.6.11. Shuffle Functions

CUDA Fortran device code can access compute capability 3.x shuffle functions. These functions enable access to variables between threads within a warp, referred to as *lanes*. In CUDA Fortran, lanes use Fortran's 1-based numbering scheme.

__shfl()

`__shfl()` returns the value of `var` held by the thread whose ID is given by `srcLane`. If the `srcLane` is outside the range of `1:width`, then the thread's own value of `var` is returned. The `width` argument is optional in all shuffle functions and has a default value of 32, the current warp size.

```
integer(4) function __shfl(var, srcLane, width)
   integer(4) var, srcLane
   integer(4), optional :: width
```

```
integer(8) function __shfl(var, srcLane, width)
   integer(8) :: var
   integer(4) :: srcLane
   integer(4), optional :: width
```

```
real(4) function __shfl(var, srcLane, width)
   real(4) :: var
   integer(4) :: srcLane
   integer(4), optional :: width
```

```
real(8) function __shfl(var, srcLane, width)
   real(8) :: var
   integer(4) :: srcLane
   integer(4), optional :: width
```

__shfl_up()

`__shfl_up()` calculates a source lane ID by subtracting delta from the caller's thread ID. The value of `var` held by the resulting thread ID is returned; in effect, `var` is shifted up the warp by `delta` lanes.

The source lane index will not wrap around the value of `width`, so the lower `delta` lanes are unchanged.

```
integer(4) function __shfl_up(var, delta, width)
   integer(4) var, delta
   integer(4), optional :: width
```

```
integer(8) function __shfl_up(var, delta, width)
   integer(8) :: var
   integer(4) :: delta
   integer(4), optional :: width
```

```
real(4) function __shfl_up(var, delta, width)
   real(4) :: var
   integer(4) :: delta
   integer(4), optional :: width
```

```
real(8) function __shfl_up(var, delta, width)
   real(8) :: var
   integer(4) :: delta
   integer(4), optional :: width
```

__shfl_down()

`__shfl_down()` calculates a source lane ID by adding `delta` to the caller's thread ID. The value of `var` held by the resulting thread ID is returned: this has the effect of shifting `var` down the warp by `delta` lanes. The ID number of the source lane will not wrap around the value of width, so the upper `delta` lanes remain unchanged.

```
integer(4) function __shfl_down(var, delta, width)
   integer(4) var, delta
   integer(4), optional :: width
```

```
integer(8) function __shfl_down(var, delta, width)
   integer(8) :: var
   integer(4) :: delta
   integer(4), optional :: width
```

```
real(4) function __shfl_down(var, delta, width)
   real(4) :: var
   integer(4) :: delta
   integer(4), optional :: width
```

```
real(8) function __shfl_down(var, delta, width)
   real(8) :: var
   integer(4) :: delta
   integer(4), optional :: width
```

__shfl_xor()

`__shfl_xor()` uses ID-1 to calculate the source lane ID by performing a bitwise XOR of the caller's lane ID with the `laneMask`. The value of `var` held by the resulting lane ID is returned. If the resulting lane ID falls outside the range permitted by `width`, the thread's own value of `var` is returned. This mode implements a butterfly addressing pattern such as is used in tree reduction and broadcast.

```
integer(4) function __shfl_xor(var, laneMask, width)
   integer(4) var, laneMask
   integer(4), optional :: width
```

```
integer(8) function __shfl_xor(var, laneMask, width)
   integer(8) :: var
   integer(4) :: laneMask
   integer(4), optional :: width
```

```
real(4) function __shfl_xor(var, laneMask, width)
   real(4) :: var
   integer(4) :: laneMask
   integer(4), optional :: width
```

```
real(8) function __shfl_xor(var, laneMask, width)
   real(8) :: var
   integer(4) :: laneMask
   integer(4), optional :: width
```

Here is an example using `__shfl_xor()` to compute the sum of each thread's variable contribution within a warp:

```
j = . . .
k = __shfl_xor(j,1);  j = j + k
k = __shfl_xor(j,2);  j = j + k
k = __shfl_xor(j,4);  j = j + k
k = __shfl_xor(j,8);  j = j + k
k = __shfl_xor(j,16); j = j + k
```

2.6.12. Restrictions

This section lists restrictions on statements and features that can appear in device subprograms.

- Recursive subroutines and functions are not allowed.
- PAUSE statements are not allowed.
- Most Input/Output statements are not allowed at all: READ, FORMAT, NAMELIST, OPEN, CLOSE, BACKSPACE, REWIND, ENDFILE, INQUIRE.
- List-directed PRINT and WRITE statements to the default unit may be used when compiling for compute capability 2.0 and higher; all other uses of PRINT and WRITE are disallowed.
- Alternate return specifications are not allowed.
- ENTRY statements are not allowed.
- Floating point exception handling is not supported.
- Fortran intrinsic functions not listed in Section 3.6.3 are not supported.
- Cray pointers are not supported.

2.7. Host code

Host subprograms may use intrinsic functions, such as the `sizeof` intrinsic function, to find the size in bytes of Fortran data structures.

2.7.1. SIZEOF Intrinsic

A call to `sizeof(A)`, where `A` is a variable or expression, returns the number of bytes required to hold the value of `A`.

```
integer(kind=4) :: i, j
j = sizeof(i)    ! this assigns the value 4 to j
```

2.8. Fortran Device Modules

NVIDIA provides a device module by default which allows access and interfaces to many of the CUDA device built-in routines.

To access this module explicitly, do one of the following:

- Add this line to your Fortran program:

```
use cudadevice
```

- Add this line to your C program:

```
#include <cudadevice.h>
```

You can use these routines in CUDA Fortran global and device subprograms, in CUF kernels, and in NVIDIA Accelerator compute regions in Fortran as well as in C. Further, the NVIDIA HPC compilers come with implementations of these routines for host code, though these implementations are not specifically optimized for the host. In uses other than CUDA Fortran global and device subprograms, you must explicitly use the module in the host subprogram unit.

CUDA Built-in Routines lists the CUDA built-in routines that are available:

Table 15: Table 16. CUDA Built-in Routines

__brev	__brevll	clock	clock64
__clz	__clzll	__cosf	cospi
cospif	__dadd_rd	__dadd_rn	__dadd_ru
__dadd_rz	__ddiv_rd	__ddiv_rn	__ddiv_ru
__ddiv_rz	__dmul_rd	__dmul_rn	__dmul_ru
__dmul_rz	__double2float_rd	__double2float_rn	__double2float_ru
__double2float_rz	__double2hiint	__double2int_rd	__double2int_rn
__double2int_ru	__double2int_rz	__double2loint	__double2ll_rd
__double2ll_rn	__double2ll_ru	__double2ll_rz	__double2uint_rd
__double2uint_rn	__double2uint_ru	__double2uint_rz	__double2ull_rd
__double2ull_rn	__double2ull_ru	__double2ull_rz	__double_as_longlong
__drcp_rd	__drcp_rn	__drcp_ru	__drcp_rz
__dsqrt_rd	__dsqrt_rn	__dsqrt_ru	__dsqrt_rz
__exp10f	__expf	__fadd_rd	__fadd_rn
__fadd_ru	__fadd_rz	__fdiv_rd	__fdiv_rn
__fdiv_ru	__fdiv_rz	fdivide	fdividef
__fdividef	__ffs	__ffsll	__float2half_rn
__float2int_rd	__float2int_rn	__float2int_ru	__float2int_rz

continues on next page

Table 15 – continued from previous page

__float2ll_rd	__float2ll_rn	__float2ll_ru	__float2ll_rz
__float_as_int	__fma_rd	__fma_rn	__fma_ru
__fma_rz	__fmaf_rd	__fmaf_rn	__fmaf_ru
__fmaf_rz	__fmul_rd	__fmul_rn	__fmul_ru
__fmul_rz	__frcp_rd	__frcp_rn	__frcp_ru
__frcp_rz	__fsqrt_rd	__fsqrt_rn	__fsqrt_ru
__fsqrt_rz	__half2float	__hiloint2double	__int2double_rn
__int2float_rd	__int2float_rn	__int2float_ru	__int2float_rz
__int_as_float	__ll2double_rd	__ll2double_rn	__ll2double_ru
__ll2double_rz	__ll2float_rd	__ll2float_rn	__ll2float_ru
__ll2float_rz	__log10f	__log2f	__logf
__longlong_as_double	__mul24	__mulhi	__popc
__popcll	__powf	__sad	__saturatef
sincos	sincosf	sincospi	sincospif
__sinf	sinpi	sinpif	__tanf
__uint2double_rn	__uint2float_rd	__uint2float_rn	__uint2float_ru
__uint2float_rz	__ull2double_rd	__ull2double_rn	__ull2double_ru
__ull2double_rz	__ull2float_rd	__ull2float_rn	__ull2float_ru
__ull2float_rz	__umul24	__umulhi	__usad

2.8.1. LIBM Device Module

NVIDIA also provides a device module which provides interfaces to standard libm functions which are not in the Fortran intrinsic library.

To access this module, add this line to your Fortran subprogram:

```
use libm
```

These interfaces are defined in the libm device module:

Table 16: Table 17. CUDA Device libm Routines

Name	Argument Datatypes		Name	Argument Datatypes
cbrt,cbrtf	real(8),real(4) returns real		llround,llroundf	real(8),real(4) returns integer
ceil,ceilf	real(8),real(4) returns real		lrint,lrintf	real(8),real(4) returns integer
copy-sign,copysignf	2*real(8),real(4) returns real		lround,lroundf	real(8),real(4) returns integer
expm1,expm1f	real(8),real(4) returns real		logb,logbf	real(8),real(4) returns real
exp10,exp10f	real(8),real(4) returns real		log1p,log1pf	real(8),real(4) returns real
exp2,exp2f	real(8),real(4) returns real		log2,log2f	real(8),real(4) returns real
fabs,fabsf	real(8),real(4) returns real		modf,modff	2*real(8),real(4) returns real
floor,floorf	real(8),real(4) returns real		near-byint,nearbyintf	real(8),real(4) returns real
fma,fmaf	3*real(8),real(4) returns real		nextafter,nextafterf	2*real(8),real(4) returns real
fmax,fmaxf	2*real(8),real(4) returns real		remain-der,remainderf	2*real(8),real(4) returns real
fmin,fminf	2*real(8),real(4) returns real		remquo,remquof	2*real(8),real(4) integer returns real
frexp,frexpf	real(8),real(4) integer returns real		rint,rintf	real(8),real(4) returns real
ilogb,ilogbf	real(8),real(4) returns real		scalbn,scalbnf	real(8),real(4) integer returns real
ldexp,ldexpf	real(8),real(4) integer returns real		scalbln,scalblnf	real(8),real(4) integer returns real
llrint,llrintf	real(8),real(4) returns integer		trunc,truncf	real(8),real(4) returns real

Here is a simple example of using the LIBM device module:

```
attributes(global) subroutine testlibm( a, b )
  use libm
  real, device :: a(*), b(*)
  i = threadIdx%x
  b(i) = cbrt(a(i))
  end subroutine
```

2.8.2. Cooperative Groups Device Module

On NVIDIA GPUs which support CUDA Compute Capability 7.0 and above, NVIDIA provides a device module which provides interfaces to cooperative group functionality which is provided by NVIDIA starting in CUDA 9.0. In our 23.3 release, the cooperative group module also supports thread block cluster programming for Hopper (cc90) and newer architectures.

To access this module, add this line to your Fortran subprogram:

```
use cooperative_groups
```

Here is a simple example of using the cooperative_groups device module which enables a cooperative grid kernel:

```
attributes(grid_global) subroutine g1(a,b,n,some_offset)
  use cooperative_groups
  real, device :: a(n), b(n)
  integer, value :: n, some_offset
  type(grid_group) :: gg
  gg = this_grid()
  do i = gg%rank, n, gg%size
    a(i) = min(max(a(i),0.0),100.0) + 0.5
  end do
  call syncthreads(gg)
  do i = gg%rank, n, gg%size
    j = i + some_offset
    if (j.gt.n) j = j - n
    b(i) = a(i) + a(j)
  end do
return
end subroutine
```

There is currently limited functionality for cooperative groups of size less than or equal to a thread block. More functionality will be added in an upcoming release. Currently, the following types are defined within the module: `grid_group`, `thread_group`, `coalesced_group`, and `cluster_group`. Each type has two public members, the size and rank. The syncthreads subroutine is overloaded in the cooperative_groups module to take the type as an argument, to appropriately synchronize the threads in that group. Minimal code sequences supported are:

Cooperative group equal to a thread block:

```
. . .
 use cooperative_groups
 type(thread_group) :: tg
 tg = this_thread_block()
 call syncthreads(tg)
```

Cooperative group equal to a warp:

```
. . .
 use cooperative_groups
 type(coalesced_group) :: wg
 wg = this_warp()
 call syncthreads(wg)
```

Cooperative group equal to a thread block cluster:

```
. . .
 use cooperative_groups
 type(cluster_group) :: clg
 clg = this_cluster()
 call syncthreads(clg)
```

The major benefit of a thread block cluster is to take advantage of distributed shared memory, which enables keeping a larger portion of data close to the processing elements. We recommend using cray pointer syntax in accessing neighboring shared memory to keep register pressure as low as possible. Here is a short example:

```
attributes(global) cluster_dims(2,1,1) subroutine t1(rnks)
  use cooperative_groups
  integer, device :: rnks(32,*)
  type(cluster_group) :: clg  ! Defined in cooperative_groups
  integer, shared :: smem(*)
  integer, shared :: dmem(*); pointer(pmem,dmem)
  i = threadIdx%x; j = blockIdx%x
  clg = this_cluster()        ! Defined in cooperative_groups
  nrank = clg%rank
  rnks(i,j) = clg%rank        ! Initialize rnks to 1 or 2
  call syncthreads(clg)       ! Sync both blocks
  if (nrank.eq.1) then ! Get a pointer to the other
    pmem = cluster_map_shared_rank(smem, 2)
  else
    pmem = cluster_map_shared_rank(smem, 1)
  end if
  dmem(i) = 100*nrank + i ! Write to the other blocks shared memory
  call syncthreads(clg)   ! Sync both blocks
  rnks(i,j) = rnks(i,j) + smem(i) ! Read what the other block wrote
end subroutine
```

The cooperative groups module also defines new `shfl_sync()` functions. These functions are similar to the `shfl()` functions discussed in an earlier section of this document, but take an extra `mask` first argument. The 32-bit mask argument specifies which threads in the warp take part in the shuffle operation, and can be passed as an integer(4) with value z'ffffffff' for most use cases. Note that, if you use the legacy `shfl()` functions with CUDA 9.0 or higher, we implicitly use `shlf_sync()` with a mask of z'ffffffff'. This may not be correct if you have thread divergence within the warp. In that case, do use the new `shfl_sync()` functions and provide the proper mask, which can be generated using the `ballot()` device function.

2.8.3. WMMA (Warp Matrix Multiply Add) Module

On NVIDIA GPUs that support CUDA Compute Capability 7.0 and above, NVIDIA includes a device module that provides interfaces to matrix operations that leverage Tensor Cores to accelerate matrix problems. This enables scientific programmers using Fortran to take advantage of real(2) matrix operations.

To access the module, add this line to your Fortran subprogram:

```
use wmma
```

Among the API routines provided in the `wmma` module are matrix multiply operations of form `C = Matmul(A, B)`, where:

- A is a 2 dimensional real(2) array dimensioned A(m,k)
- B is a 2 dimensional real(2) array dimensioned B(k,n)
- C is a 2 dimensional real(2) or real(4) array dimensioned C(m,n)

Using the Fortran kind attribute, it is possible to declare and use data in half precision format. Details on representation and requirements for use can be found in *Half-precision Floating Point* section.

Here is a simple example using the `wmma` device module to do matrix multiplication using a single warp of threads. There are two 16×16 `real(2)` matrices being multiplied and accumulated into a 16×16 `real(4)` matrix:

```
#include "cuf_macros.CUF"
    module m
    integer, parameter :: wmma_m = 16
    integer, parameter :: wmma_n = 16
    integer, parameter :: wmma_k = 16

    contains
    ! kernel for 16 x16 matrices (a, b, and c) using wmma
    ! Should be launched with one block of 32 threads
    attributes(global) subroutine wmma_single(a, b, c)
    use wmma
    implicit none
    real(2), intent(in) :: a(wmma_m,*) , b(wmma_k,*)
    real(4) :: c(wmma_m,*)
    WMMASubMatrix(WMMAMatrixA, 16, 16, 16, Real, WMMAColMajor) :: sa
    WMMASubMatrix(WMMAMatrixB, 16, 16, 16, Real, WMMAColMajor) :: sb
    WMMASubMatrix(WMMAMatrixC, 16, 16, 16, Real, WMMAKind4) :: sc
    integer :: lda, ldb, ldc

    lda = wmma_m
    ldb = wmma_k
    ldc = wmma_m

    sc = 0.0_4
    call wmmaLoadMatrix(sa, a(1, 1), lda)
    call wmmaLoadMatrix(sb, b(1, 1), ldb)
    call wmmaMatMul(sc, sa, sb, sc)
    call wmmaStoreMatrix(c(1, 1), sc, ldc)

    end subroutine wmma_single
    end module m
```

The call site looks as follows to invoke with a single warp of threads:

```
call wmma_single<<<1,32>>>(ah_d, bh_d, c_d)
```

For this simple example, the matrices passed in as arguments to the kernel are the same size as the WMMA submatrices. Thus, to perform the matrix multiplication we simply initialize the C WMMA submatrix to 0.0, load the A and B matrices from global memory to WMMA submatrices, perform the matrix multiplication on the submatrices, and store the result from the WMMA submatrix to global memory.

You may have noticed that the thread index `threadIdx` does not appear at all in this code. This underlies the important concept to take away from this example: the threads in a warp work collectively to accomplish these tasks. So when dealing with the WMMA submatrices, we are doing warp-level programming rather than thread-level programming. This kernel is launched with a single warp of

32 threads, yet each of our WMMA submatrices has 16×16 or 256 elements. When the initialization statement:

```
sc = 0.0_4
```

is executed, each thread sets 8 elements in the 16×16 submatrix to zero. The mapping of threads to submatrix elements is opaque for this and other operations involving WMMA submatrices - from a programming standpoint we only address what happens collectively by a warp of threads on WMMA submatrices.

The statements that load the A and B from global memory to WMMA submatrices:

```
call wmmaLoadMatrix(sa, a(1, 1), lda)
    call wmmaLoadMatrix(sb, b(1, 1), ldb)
```

also work collectively. In these calls, the WMMA submatrices are specified as the first argument, and the second arguments contain the addresses of the upper left element of the tiles in global (or shared) memory to be loaded to the WMMA submatrices. The leading dimension of the the matrices in global (or shared) memory is the third argument. Note that the arguments passed to `wmmaLoadMatrix()` are the same for all threads in the warp. Because the mapping of elements to threads in a warp is opaque, each thread just passes the address of the first element in the 16×16 matrix along with the leading dimension as the third parameter, and the load operation is distributed amongst the threads in the warp.

The matrix multiplication on the WMMA submatrices is performed by the statement:

```
call wmmaMatMul(sc, sa, sb, sc)
```

which is again performed collectively by a warp of threads. Here used the same accumulator submatrix for the first and last arguments in the `wmmaMatMul()` call, which is why its initialization to zero is required.

The `wmmaStoreMatrix()` call:

```
call wmmaStoreMatrix(c(1, 1), sc, ldc)
```

is analogous to the prior `wmmaLoadMatrix` calls, but here the first argument is the address of the upper left element of the tile in global (or shared) memory and the second argument is the WMMA submatrix whose values are stored. When both `wmmaLoadMatrix()` and `wmmaStoreMatrix()` are called with accumulator (`WMMAMatrixC`) arguments, there is an optional fourth argument that specifies the storage order. In CUDA Fortran, the default is the `WMMAColMajor` or column-major storage order.

One final note on arguments to the `wmmaLoadMatrix()` and `wmmaStoreMatrix()` routines. There is a requirement that the leading dimension of the matrices, specified by the third argument of these routines, must be a multiple of 16 bytes (e.g. 8 `real(2)` words or 4 `real(4)` words).

More details about data declaration and wmma operations are available at Tensor Core Programming Using CUDA Fortran https://devblogs.nvidia.com/tensor-core-programming-using-cuda-fortran/

The `wmma` module also provides access to the following half precision mathematical intrinsics and requires `use wmma` in order to access them: `abs`, `sin`, `cos`, `tan`, `acos`, `asin`, `atan`, `atan2`, `sinh`, `cosh`, `tanh`, `log`, `log10`, `exp`, and `sqrt`. It is expected that in a future release, these intrinsics will be available without having to mention `use wmma`.

2.9. Fortran Host Modules

The primary Fortran module which NVIDIA provides for CUDA Fortran is named `cudafor`. This module contains all of the supported interfaces to the CUDA Runtime API listed in the next chapter. In addition, it contains interfaces to some Fortran array intrinsics which are described in sections below.

Beginning in the 25.3 release, the structure of the `cudafor` module has been changed slightly. The module now includes, or "uses" 3 submodules: `cuda_runtime_api`, `gpu_reductions`, and `sort`. The `cudafor` functionality has not changed. But for new users, or users who have needed to work-around name conflicts in the module, it may be better to `use cuda_runtime_api` to expose interfaces to the CUDA runtime calls described in Chapter 4 of this guide.

There are a number of other Fortran modules which interface to CUDA Libraries. Those are described thoroughly in the *NVIDIA Fortran CUDA Interfaces* document. These include libraries for computation, like `CUBLAS`, `CUFFT`, and `CUSPARSE`, for communication, `NCCL`, `NVSHMEM`, and for profiling, `NVTX`.

One other host module, which we will describe in this chapter, is `CUTENSOR`. It has been extended in a module named `cutensorex` and contains overloaded interfaces to many more Fortran array intrinsics, some of which call into the NVIDIA CUTENSOR library, and some which do not, but they use the same deferred evaluation techniques. These implementations operate on device (or managed) data, and are called from the host.

2.9.1. Overloaded Fortran Reduction Intrinsics in GPU_REDUCTIONS and CUDAFOR

The `SUM`, `MAXVAL`, `MINVAL`, `MAXLOC`, and `MINLOC` Fortran intrinsics are overloaded to accept device or managed arrays when the cudafor or gpu_reductions module is used, from host code. If the mask optional argument is used, the mask argument must be either a device logical array, or an expression containing managed operands and constants, i.e. the mask must be computable on the host but readable on the device. As in standard Fortran, the mask shape and size, if present, must conform to the data array.

Here is a complete example which performs the sum and maxval reductions on the GPU:

```
program multidimred
use cudafor
real(8), managed :: a(5,5,5,5,5)
real(8), managed :: b(5,5,5,5)
real(8) :: c
call random_number(a)
do idim = 1, 5
  b = sum(a, dim=idim)
  c = max(maxval(b), c)
end do
print *,"Max along any dimension",c
end program
```

Array slices are also supported. This may run less efficiently on the GPU, but is very powerful nonetheless, and useful for debugging:

```
real(4), managed :: a(n,m)
reslt(ix) = sum(a(2:n-1,:))
```

(continues on next page)

(continued from previous page)

```
reslt(ix) = sum(a(:,3:m-2))
reslt(ix) = sum(a(n2:n,m2:m))
reslt(ix) = sum(a(1:n3,1:m3))
reslt(ix) = sum(a(n2:n3,m2:m3))
```

By default, intrinsic reductions that are supported on the device will be executed on the device for (large enough) managed arrays. There may be occasions where one would like to perform reductions on managed data on the host. This can be accomplished using the rename feature of the "use" statement, for example:

```
program reductionRename
use cudafor, gpusum => sum
implicit none
integer, managed :: m(3000)
integer :: istat
m = 1
istat = cudaDeviceSynchronize()
write(*,*) sum(m)    ! executes on host
write(*,*) gpusum(m) ! executes on device
end program
```

Beginning in the NVHPC 23.1 release, all five functions, `SUM`, `MAXVAL`, `MINVAL`, `MAXLOC`, and `MINLOC` can now accept an optional stream argument. If a unique per-thread default stream was set via a call to `cudaforSetDefaultStream`, the reduction operation will pick that up and run on that stream. Given the new, simpler functionality, support for `cudaforReductionSetStream()` and `cudaforReductionGetStream()` has been dropped starting in 23.1 as well. For instance:

```
integer(kind=cuda_stream_kind) :: istrm
        x = sum(a, stream=istrm)
```

is now the simplest way to run a sum reduction on a specific stream.

The following sections describe each function, with current support and limitations, in more detail.

2.9.1.1 Fortran SUM Intrinsic Function

The overloaded interface for `SUM` is in the `cudafor` module. It can return either a scalar, which is most common, or an array, if the optional `dim` argument is used. The real(4), real(8), integer(4), and integer(8) data types are supported. Complex types may be added in a future release. The input array can be between one and seven dimensions. The two forms are:

```
function sum ( array, mask, stream ) result(res)
    type, device :: array(...)  ! type is real or integer, kind = 4 or 8
    logical(4), device, optional, intent(in) :: mask(...)
    integer(kind=cuda_stream_kind), optional, intent(in) :: stream
    type, intent(out) :: res ! same type as array
```

```
function sum ( array, dim, mask, stream ) result(res)
    type, device :: array(...)  ! type is real or integer, kind = 4 or 8
    integer(4), intent(in) :: dim
    logical(4), device, optional, intent(in) :: mask(...)
    integer(kind=cuda_stream_kind), optional, intent(in) :: stream
    type, allocatable, managed, intent(out) :: res(...) ! same type as array
                                                ! rank is one less than array
```

2.9.1.2 Fortran MAXVAL Intrinsic Function

The overloaded interface for `MAXVAL`, which returns the maximum value of an element in the array, is in the `cudafor` module. It can return either a scalar, which is most common, or an array, if the optional `dim` argument is used. The real(4), real(8), integer(4), and integer(8) data types are supported. The input array can be between one and seven dimensions. The two forms are:

```
function maxval ( array, mask, stream ) result(res)
    type, device :: array(...)  ! type is real or integer, kind = 4 or 8
    logical(4), device, optional, intent(in) :: mask(...)
    integer(kind=cuda_stream_kind), optional, intent(in) :: stream
    type, intent(out) :: res ! same type as array
```

```
function maxval ( array, dim, mask, stream ) result(res)
    type, device :: array(...)  ! type is real or integer, kind = 4 or 8
    integer(4), intent(in) :: dim
    logical(4), device, optional, intent(in) :: mask(...)
    integer(kind=cuda_stream_kind), optional, intent(in) :: stream
    type, allocatable, managed, intent(out) :: res(...) ! same type as array
                                                ! rank is one less than array
```

2.9.1.3 Fortran MINVAL Intrinsic Function

The overloaded interface for `MINVAL`, which returns the minimum value of an element in the array, is in the `cudafor` module. It can return either a scalar, which is most common, or an array, if the optional `dim` argument is used. The real(4), real(8), integer(4), and integer(8) data types are supported. The input array can be between one and seven dimensions. The two forms are:

```
function minval ( array, mask, stream ) result(res)
    type, device :: array(...)  ! type is real or integer, kind = 4 or 8
    logical(4), device, optional, intent(in) :: mask(...)
    integer(kind=cuda_stream_kind), optional, intent(in) :: stream
    type, intent(out) :: res ! same type as array
```

```
function minval ( array, dim, mask, stream ) result(res)
    type, device :: array(...)  ! type is real or integer, kind = 4 or 8
    integer(4), intent(in) :: dim
    logical(4), device, optional, intent(in) :: mask(...)
    integer(kind=cuda_stream_kind), optional, intent(in) :: stream
    type, allocatable, managed, intent(out) :: res(...) ! same type as array
                                                ! rank is one less than array
```

2.9.1.4 Fortran MAXLOC Intrinsic Function

The overloaded interface for `MAXLOC`, which returns an array of indices, starting at 1, identifying the maximum value of an element in the array which appears first, is in the `cudafor` module. The size of the function result is equal to the rank of the input array, and is an integer host array. The real(4), real(8), integer(4), and integer(8) data types are supported. The input array can be between one and seven dimensions. The dim argument is only supported for 1-D arrays, in which case the result is a scalar rather than an array of size=1. There are also optional kind, back, and stream arguments, the first two of those being standard Fortran, and the latter being a CUDA Fortran extension.

```
function maxloc ( array, mask, kind, back stream ) result(res)
    type, device :: array(...)  ! type is real or integer, kind = 4 or 8
    logical(4), device, optional, intent(in) :: mask(...)
    integer, optional, intent(in) :: kind
    logical, optional, intent(in) :: back
    integer(kind=cuda_stream_kind), optional, intent(in) :: stream
    integer, intent(out) :: res(*) ! Size of res is equal to rank of array
```

2.9.1.5 Fortran MINLOC Intrinsic Function

The overloaded interface for `MINLOC`, which returns an array of indices, starting at 1, identifying the minimum value of an element in the array which appears first, is in the `cudafor` module. The size of the function result is equal to the rank of the input array, and is an integer host array. The real(4), real(8), integer(4), and integer(8) data types are supported. The input array can be between one and seven dimensions. The dim argument is only supported for 1-D arrays, in which case the result is a scalar rather than an array of size=1. There are also optional kind, back, and stream arguments, the first two of those being standard Fortran, and the latter being a CUDA Fortran extension.

```
function minloc ( array, mask, kind, back stream ) result(res)
    type, device :: array(...)  ! type is real or integer, kind = 4 or 8
    logical(4), device, optional, intent(in) :: mask(...)
    integer, optional, intent(in) :: kind
    logical, optional, intent(in) :: back
    integer(kind=cuda_stream_kind), optional, intent(in) :: stream
    integer, intent(out) :: res(*) ! Size of res is equal to rank of array
```

2.9.2. Fortran Sorting Subroutines Module

Typically, for best performance, we recommend generating sort routines using CUDA Thrust, the nvcc compiler, and calling those functions from Fortran. Starting with the 23.5 release, we also include basic sort subroutines as part of the CUDA Fortran libraries, which are readily available and may provide "good enough" performance.

The interfaces to the library functions can be accessed by adding `use sort` to your code, and the overloaded sorting subroutine is named `fsort()`. The library provides a radix sort implementation for integer(4), integer(8), real(4), and real(8) arrays. The subroutines can accept either host, managed, or device arrays. The subroutines can also accept an index array, to return the sort permutations. Other optional arguments are listed below.

Here is a simple example which sorts an array of reals on the GPU:

```
program sortit
use sort
real(4), managed :: a(1000)
call random_number(a)
call fsort(a, 1000)
print *,all(a(1:999) .le. a(2:1000))
end program
```

The host and device functionality is divided into four types of calls, and the arguments for each are:

```
! Host arrays, no indices
subroutine fsort(array, n, stream)
type(kind) :: array(*)  ! Type is integer or real, kind is 4 or 8
integer(kind) :: n      ! kind is 4 or 8
integer(kind=cuda_stream_kind), optional :: stream
end subroutine
```

```
! Host arrays, with indices
subroutine fsort(array, indices, n, init_index, stream)
type(kind) :: array(*)  ! Type is integer or real, kind is 4 or 8
integer(4) :: indices(*)
integer(kind) :: n      ! kind is 4 or 8
logical(4), optional :: init_index  ! Flag to initialize the indices to 1..n
integer(kind=cuda_stream_kind), optional :: stream
end subroutine
```

```
! Managed or device arrays, no indices
subroutine fsort(array, n, workspace, worksize, stream)
type(kind), device :: array(*)  ! Type is integer or real, kind is 4 or 8
integer(kind) :: n      ! kind is 4 or 8
type(kind), device, optional :: workspace(*)  ! Same type as array
integer(8), optional :: worksize              ! Size of workspace in elements
integer(kind=cuda_stream_kind), optional :: stream
end subroutine
```

```
! Managed or device arrays, with indices
subroutine fsort(array, indices, n, init_index, workspace, worksize, stream)
type(kind), device :: array(*)  ! Type is integer or real, kind is 4 or 8
integer(4), device :: indices(*)
integer(kind) :: n      ! kind is 4 or 8
logical(4), optional :: init_index  ! Flag to initialize the indices to 1..n
type(kind), device, optional :: workspace(*)  ! Same type as array
integer(8), optional :: worksize              ! Size of workspace in elements
integer(kind=cuda_stream_kind), optional :: stream
end subroutine
```

Without a provided workspace argument, the subroutines will allocate temporary work space using either cudaMalloc(), or cudaMallocAsync(), depending on the CUDA version support and whether the stream is specified. The amount of workspace required to avoid temporary allocations for the subroutines which take the worksize argument is roughly N elements, kind equal to 4 or 8, plus up to another 2 MBytes above that. For instance, sorting an integer(4) array of size 10 million will use workspace of roughly 42 MBytes or 10.5 million elements.

2.9.3. Overloaded Fortran Reduction Intrinsics in CUTENSOREX

The ALL, ANY, and COUNT Fortran intrinsics are overloaded to accept device or managed arrays when the cutensorex module is used, from host code. As these three functions operate only on a mask, a different tact was chosen to make these functions more flexible, and recognize and efficiently evaluate commonly-used mask expressions.

Using the same deferred evaluation and assignment techniques that were used in cutensorex for matmul(), spread(), transpose(), and reshape(), beginning in the 23.1 release we now support more F90 array

intrinsic operations.

These three functions do not call into the cuTensor library, but build upon and extend the software infrastructure developed previously for those wrappers.

First, here are the mask expressions which are recognized for deferred evaluation:

For A, B, x, dx, alpha, beta

```
A is a device array of real(4), real(8), integer(4), or integer(8)
B is a device array with the same type as A.
A and B are 1-3 dimensional, (conforming arrays)
x is a scalar with the same type as A
dx is a device scalar with the same type as A
alpha and beta are host scalars with the same type as A
```

In one kernel launch, we support these mask expressions:

```
A .relop. B
A .relop. x
A .relop. dx
abs(A) .relop. B
abs(A) .relop. x
abs(A) .relop. dx
(A +/- B) .relop. x
(A +/- B) .relop. dx
abs(A +/- B) .relop. x
abs(A +/- B) .relop. dx
(alpha*A + beta*B) .relop. x
(alpha*A + beta*B) .relop. dx
abs(alpha*A + beta*B) .relop. x
abs(alpha*A + beta*B) .relop. dx
```

For relop in EQ, NE, LE, LT, GE, GT

One exception, for convenience, is if the operation is “A .relop. x”, x can be kind=4 if A is kind=8

In most cases, the B array can also be the result of the spread() or transpose() intrinsic function, to make B conform to the shape of A. General reshape() support for mask operands is not available at this time.

2.9.3.1 Overloaded Logical Array Assignment in CUTENSOREX

The result of a logical expression from the section above can be assigned to an array of type logical(4). For example:

A and B are conforming device arrays of type real(4), x is a real(4) scalar, and L a device arrray of type logical(4):

```
block; use cutensorex
L = A .LT. B
L = ABS(A) .GE. 1.0
L = ABS(A - B) .LE. x
end block
```

Of course, a logical array can be generated using any means: CUDA kernels, CUF kernels, or computed/copied from the host. These are provided as a convenience, but note that if a mask is constant over many uses, it is probably faster to compute it once and pass it into these functions rather than to re-evalute it many times.

2.9.3.2 Fortran ALL Intrinsic Function

The interface for ALL is in the `cutensorex` module.

The Fortran array reduction ALL returns true if every element of the mask is true, otherwise it returns false. The mask can be a logical array, array slice, or any of the logical expressions described above. The optional dim argument to ALL() is not supported at this time.

```
logical(4) function all ( mask )
    logical, intent(in) :: mask(...)  ! mask is 1 - 3 dimensions
```

For example, if A and B are conforming arrays with the device or managed attribute, and X is a scalar of the same type:

```
IF (ALL(A .EQ. B)) PRINT *,"PASSED"
IF (ALL(ABS(A - B) .GT. X)) CALL REDO()
```

2.9.3.3 Fortran ANY Intrinsic Function

The interface for ANY is in the `cutensorex` module.

The Fortran array reduction ANY returns true if any element of the mask is true, and returns false if none are true. The mask can be a logical array, array slice, or any of the logical expressions described above. The optional dim argument to ANY() is not supported at this time.

```
logical(4) function any ( mask )
    logical, intent(in) :: mask(...)  ! mask is 1 - 3 dimensions
```

For example, if A and B are conforming arrays with the device or managed attribute, and X is a scalar of the same type:

```
IF (ANY(A .EQ. B)) PRINT *,"FAILED"
IF (ANY(ABS(A) .GT. X)) CALL REDO()
```

2.9.3.4 Fortran COUNT Intrinsic Function

The interface for COUNT is in the `cutensorex` module.

The Fortran array reduction COUNT returns the number of true elements of the mask. The mask can be a logical array, array slice, or any of the logical expressions described above. The optional dim argument to COUNT() is not supported at this time.

```
integer function count ( mask )
    logical, intent(in) :: mask(...)  ! mask is 1 - 3 dimensions
```

For example, if A and B are conforming real(4) arrays with the device or managed attribute, EPS is a real(4) scalar, and ICNT1 and ICNT2 are integer scalars:

```
ICNT1 = COUNT(A .EQ. B)
ICNT2 = COUNT(ABS(A - B) .LE. EPS)
```

2.9.4. Overloaded Fortran Array Intrinsics in CUTENSOREX

This section lists the other overloaded functions available in the cutensorex module. Similar to the last section, these Fortran intrinsics accept device or managed arrays when the cutensorex module is used, from host code.

The first five functions in this section also take a mask argument, and accept the same mask arrays or expressions described in the previous section. The more complicated functions in this group use a scan algorithm described in this paper: *Single-pass Parallel Prefix Scan with Decoupled Look-Back*, by Duane Merrill and Michael Garland.

The second set of functions call into either the cuTensor or cuRand library, and are included here for completeness. They were previously documented in the *NVIDIA Fortran CUDA Interfaces* document.

2.9.4.1 Fortran MERGE Intrinsic Function

The interface for `MERGE` is in the `cutensorex` module.

The Fortran merge() intrinsic is an elemental selection based on the mask evaluation. It takes three arguments, an array of "true" values, one or more "false" values, and a mask. The merge() intrinsic function can take a mask expression in the form specified above as an argument, or a logical(4) device array. In the current implementation, only the second argument (the false selection) can be a scalar. Only real(4), real(8), integer(4), and integer(8) arrays are supported, and only for arrays of 1 - 3 dimensions. The tsource argument and mask argument must be conforming arrays, and if fsource is an array, it must conform as well.

```
function merge ( tsource, fsource, mask ) result(res)
    type, intent(in) :: tsource(...)  ! type is real or integer, kind = 4 or 8
    type, intent(in) :: fsource(...)  ! type same as tsource, array or scalar
    logical, intent(in) :: mask(...)  ! mask is 1 - 3 dimensions
    type, intent(out) :: res(...)     ! type, kind, rank same as tsource
```

For example: For A, B, C, arrays of type integer(4), and K a scalar of type integer(4):

```
C = MERGE(A, B, A .GT. B)
C = MERGE(A, 0, ABS(A) .LT. K)
```

2.9.4.2 Fortran PACK Intrinsic Function

The interface for `PACK` is in the `cutensorex` module.

The Fortran pack() intrinsic is useful for gathering selected data from a multiple-dimensional array into a rank-1 array. The pack intrinsic is unique in that the size of the output array is not known until the function has been completely evaluated. This Fortran pack() intrinsic function is an efficient parallel implementation and can take a mask expression specified above as the mask argument.

Currently, as part of our emphasis on performance, we do not re-allocate the LHS destination to fit the result; it is the user's responsibility to make sure the LHS destination array is large enough.

Only real(4), real(8), integer(4), and integer(8) arrays are supported, and only for arrays of 1 - 3 dimensions. This implementation does not support the vector optional argument to pack(). This implementation does add a new optional argument, "count" which can return the count of passing mask results, basically the number of elements written into the LHS result:

```
function pack ( array, mask, count ) result(res)
    type, intent(in) :: array(...)    ! type is real or integer, kind = 4 or 8
    logical, intent(in) :: mask(...)  ! mask is 1 - 3 dimensions
    integer, optional, intent(in) :: count
    type, intent(out) :: res(*)       ! type is same as array, rank is 1-D
```

For example: A and B are device arrays of type real(4), and x is a scalar of type real(4). C and D are device arrays of the same type, where C conforms to both A and B, and D is a 1-dimensional array:

```
D = PACK(C, ABS(A - B) .GT. x)
D = PACK(C, MASK=(A .EQ. B), COUNT=ICNT)
```

2.9.4.3 Fortran PACKLOC Function

The interface for PACKLOC is in the `cutensorex` module.

The Fortran packloc() function is an extension of the PACK intrinsic, but does not take a source array. Instead, it produces a packed array of indices, or locations, where the mask evaluates to true. This Fortran packloc() function uses the same efficient parallel implementation as PACK, and can take a mask expression specified above as the mask argument.

Currently, as part of our emphasis on performance, we do not re-allocate the LHS destination to fit the result; it is the user's responsibility to make sure the LHS destination array is large enough.

Only mask expressions involving 1D arrays are currently supported. This implementation does support the optional argument, "count" which can return the count of passing mask results, basically the number of elements written into the LHS result:

Similar to maxloc and other location functions, the indices begin at 1, and are not affected by non-unary strides or lower bounds of the arrays passed to the function.

```
function packloc ( mask, count ) result(res)
    logical, intent(in) :: mask(:)   ! mask is a 1D logical array or supported
↪expression
    integer(4), optional, intent(out) :: count
    integer(4), intent(out) :: res(*)
```

For example: A and B are device arrays of type real(4), and x is a scalar of type real(4). D is an integer(4) device array:

```
D = PACKLOC(ABS(A - B) .GT. x, COUNT=ICNT)
```

2.9.4.4 Fortran UNPACK Intrinsic Function

The interface for UNPACK is in the `cutensorex` module.

The Fortran unpack() intrinsic function is the complement of pack(), and can take a mask expression specified above as the mask argument. There are some limitations in the current implementation for unpack() related to the field argument. In this implementation, the field argument is optional, and if it is left off, the LHS destination is treated as the field. If the field argument is a scalar, unpack works according to the standard. If the field argument is an array, the mask operation must be a logical array, not a mask expression, and of course the mask and field must be conforming in size and shape.

Only real(4), real(8), integer(4), and integer(8) arrays are supported, and only for arrays of 1 - 3 dimensions. The output array and mask argument must be conforming arrays.

```
function unpack ( array, mask, field ) result(res)
    type, intent(in) :: array(*)      ! type is real or integer, kind = 4 or 8
    logical, intent(in) :: mask(...)  ! mask is 1 - 3 dimensions
    type, optional, intent(in) :: field(...)  ! array or scalar
    type, intent(out) :: res(...)     ! type same as array, rank same as mask
```

For example: For A and B device arrays of type real(4), x and y are scalars of type real(4), C and D of the same type, where C conforms to both A and B, and D is a 1-dimensional array:

```
C = UNPACK(D, ABS(A - B) .GT. x)
C = UNPACK(D, MASK=(ABS(A - B) .GT. x), FIELD=y)
```

2.9.4.5 Fortran COUNT_PREFIX Intrinsic Function

The interface for COUNT_PREFIX is in the cutensorex module.

The count_prefix function was defined in High Performance Fortran (HPF). It computes a running count of the number of true mask values, in array storage order. An optional logical argument, EXCLUSIVE, specifies that the current mask result does not contribute to the current output, only to succeeding counts. Another optional argument, the integer DIM, specifies to compute the counts for a multi-dimensional array only across the specific dimension.

The complete function declaration and argument list is:

```
function count_prefix ( mask, dim, exclusive ) result(res)
    logical, intent(in) :: mask(...)  ! mask is 1 - 3 dimensions
    integer, optional, intent(in) :: dim  ! 1 - 3 depending on rank of mask
    logical, optional, intent(in) :: exclusive ! Default is .false. (inclusive)
    integer, intent(out) :: res(...)  ! same size and rank as mask
```

For example: For A, B, and x of type real(4), C of type integer(4):

```
C = COUNT_PREFIX(A .GT. 0)
C = COUNT_PREFIX(A .EQ. B, DIM=1)
C = COUNT_PREFIX(MASK=ABS(A - B) .LE. x, DIM=2, EXCLUSIVE=.true.)
```

HPF also specified a SEGMENT optional argument, but that functionality is not in the current release.

2.9.4.6 Fortran SUM_PREFIX Intrinsic Function

The interface for SUM_PREFIX is in the cutensorex module.

The sum_prefix function was also defined in HPF. It computes a running sum of array element values, for which the corresponding mask is true, in array storage order. An optional logical argument, EXCLUSIVE, specifies that the array value does not contribute to the current output, only to succeeding sums. Another optional argument, the integer DIM, specifies to compute the counts for a multi-dimensional array only across the specific dimension. For this function, the MASK is also optional; without it, every array element contributes to the sums.

The complete function declaration and argument list is:

```
function sum_prefix ( array, mask, dim, exclusive ) result(res)
    type, intent(in) :: array(...)    ! type is real or integer, kind = 4 or 8
    logical, optional, intent(in) :: mask(...)  ! mask is 1 - 3 dimensions
    integer, optional, intent(in) :: dim  ! 1 - 3 depending on rank of array
```

(continues on next page)

(continued from previous page)

```
    logical, optional, intent(in) :: exclusive ! Default is .false. (inclusive)
    type, intent(out) :: res(...)  ! same size and rank as array
```

For example: For A, B, C, D, and x of type real(4):

```
D = SUM_PREFIX(C, A .GT. 0)
D = SUM_PREFIX(C, MASK=(A .NE. B), DIM=2)
D = SUM_PREFIX(C, MASK=ABS(A - B) .LE. x, EXCLUSIVE=.true.)
```

2.9.4.7 Fortran RESHAPE Intrinsic Function

The interface for RESHAPE is in the `cutensorex` module. This function is also documented thoroughly in the *NVIDIA Fortran CUDA Interfaces* document in the cuTensor chapter.

The Fortran reshape() intrinsic changes the shape of an array and possibly permutes the dimensions and layout. It is invoked as:

```
D = alpha * func(reshape(A, shape=[...], order=[...]))
```

The arrays A and D can be of type real(2), real(4), real(8), complex(4), or complex(8). The rank (number of dimensions) of A and D can be from 1 to 7. The alpha value is expected to be the same type as A, or as func(reshape(A)), if that differs. Accepted functions which can be applied to the result of reshape are listed in the Fortran CUDA Interfaces document referred to above. The pad argument to the F90 reshape function is not currently supported. This Fortran call, besides initialization and setting up cuTENSOR descriptors, maps to `cutensorPermutation()`.

```
function reshape ( source, shape, order ) result(res)
    type, intent(in) :: source(...)  ! type is real or complex
    integer, intent(in) :: shape(:)
    integer, optional, intent(in) :: order(*)
    type, intent(out) :: res(...)     ! type, kind same as source
```

```
! Example to switch the 2nd and 3rd dimension layout
D = reshape(a,shape=[ni,nk,nj], order=[1,3,2])
! Same example, take the absolute value and scale by 2.5
D = 2.5 * abs(reshape(a,shape=[ni,nk,nj], order=[1,3,2]))
```

2.9.4.8 Fortran TRANSPOSE Intrinsic Function

The interface for TRANSPOSE is in the `cutensorex` module. This function is also documented thoroughly in the *NVIDIA Fortran CUDA Interfaces* document in the cuTensor chapter.

The Fortran transpose() intrinsic transposes a matrix (a 2-dimensional array). It is invoked as:

```
D = alpha * func(transpose(A))
```

The arrays A and D can be of type real(2), real(4), real(8), complex(4), or complex(8). The rank (number of dimensions) of A and D is 2. Applying scaling (the `alpha` argument) or applying a function to the transpose result is optional. The alpha value is expected to be the same type as A, or as func(transpose(A)), if that differs. Accepted functions which can be applied to the result of the transpose are listed in the Fortran CUDA Interfaces document referred to above. This Fortran call, besides initialization and setting up cuTENSOR descriptors, maps to `cutensorPermutation()`.

```
! Example of transpose
D = transpose(A)
! Same example, take the absolute value and scale by 2.5
D = 2.5 * abs(tranpose(A))
```

The transpose() function is also supported as part of the “B” argument in mask expressions described above. For example, if A is a 2-dimension mxn array, and B is nxm:

```
ICNT = COUNT(A .GT. TRANSPOSE(B))
```

2.9.4.9 Fortran SPREAD Intrinsic Function

The interface for SPREAD is in the `cutensorex` module. This function is also documented in the *NVIDIA Fortran CUDA Interfaces* document in the cuTensor chapter.

The Fortran spread() intrinsic increases the rank of an array by one across the specified dimension and broadcasts the values over the new dimension. It is invoked as:

```
D = alpha * func(spread(A, dim=i, ncopies=n))
```

The arrays A and D can be of type real(2), real(4), real(8), complex(4), or complex(8). The rank (number of dimensions) of A and D can be from 1 to 7. The alpha value is expected to be the same type as A. Accepted functions which can be applied to the result of spread are listed in the Fortran CUDA Interfaces document referred to above. This Fortran call, besides initialization and setting up cuTENSOR descriptors, maps to `cutensorPermutation()`.

```
! Example to add and broadcast values over the new first dimension
D = spread(A, dim=1, ncopies=n1)
! Same example, take the absolute value and scale by 2.5
D = 2.5 * abs(spread(A, dim=1, ncopies=n1))
```

The spread() function is also supported as part of the “B” argument in mask expressions described above. For example, if A is a 2-dimension mxn array, and B is 1-dimensional array of length m:

```
ICNT = COUNT(A .GT. SPREAD(B, dim=2, ncopies=n))
```

2.9.4.10 Fortran MATMUL Intrinsic Function

The interface for `MATMUL` is in the `cutensorex` module. This function is also documented thoroughly in the *NVIDIA Fortran CUDA Interfaces* document in the cuTensor chapter.

The Fortran matmul() intrinsic performs matrix multiplication, one instance of tensor contractions. Either operand to matmul can be a permuted array, the result of a call to reshape(), transpose(), or spread(). The cuTENSOR library does not currently support applying an elemental function to the array operands, but the result and accumlator can be scaled. Here are some supported forms:

```
D = matmul(A, B)
```

```
D = matmul(permute(A), B)
```

```
D = matmul(A, permute(B))
```

```
D = matmul(permute(A), permute(B))
```

```
D = C + matmul(A, B)
```

```
D = C - matmul(A, B)
```

```
D = alpha * matmul(A, B) + beta * C
```

The arrays A, B, C, and D can be of type real(2), real(4), real(8), complex(4), or complex(8). The rank (number of dimensions) of A, B, C, and D must be 2, after any permutations. Arrays C and D must currently have the same shape, strides, and type. The alpha value is expected to be the same type as A and B. The beta value should have the same type as C. The Fortran wrapper does no type conversion, though cuTENSOR may. Compile-time checking of array conformance is limited. Other runtime checks for unsupported combinations may come from either the Fortran wrapper or from cuTENSOR. Fortran support for `Matmul`, besides initialization and setting up cuTENSOR descriptors, maps to `cutensorContraction()`.

```
! Example to multiply two matrices together
D = matmul(A, B)
! Same example, accumulate into C
C = C +  matmul(A, B)
! Same example, transpose the first argument
C = C + matmul(transpose(A), B)
```

2.9.4.11 Fortran DOT_PRODUCT Intrinsic Function

The interface for `DOT_PRODUCT` is in the `cutensorex` module. This function is also documented thoroughly in the *NVIDIA Fortran CUDA Interfaces* document in the cuTensor chapter.

The Fortran dot_product() intrinsic performs the dot product of two vectors, one specific instance of a tensor contraction. In the standard form, it returns a scalar of the same type as the input arguments, and the destination on the LHS of the assignment must have the device or managed attribute. Either operand to dot_product can be a permuted array, the result of a call to reshape(), creating a 1-D array. Note that the Fortran definition of dot_product for complex variables performs a conjugate of the first argument. Here are some supported forms:

```
S = dot_product(A, B)
```

```
S = dot_product(reshape(T, shape=[m*n]), B)
```

```
S = dot_product(ABS(A), B)
```

```
ZC = dot_product(ZX,ZY) ! BLAS ZDOTC equivalent
```

```
ZU = dot_product(CONJG(ZX),ZY) ! BLAS ZDOTU equivalent
```

The input arrays can be of type real(2), real(4), real(8), complex(4), or complex(8). Fortran support for `DOT_PRODUCT`, besides initialization and setting up cuTENSOR descriptors, maps to `cutensorContraction()`.

This implementation has been extended to expose more of the `cutensorContraction()` functionality at a high level. The extended interface to `DOT_PRODUCT` accepts multi-dimensional arrays and a dim argument. The dot_product will be computed only along the specified dimension, resulting in an array with rank one fewer than the input arrays.

Most of the same permutations, functions and accumulation operations that are provided with `MATMUL` are provided with `DOT_PRODUCT`. Here are a few examples: For A and B an NxN matrix, X and V are vectors of length N, and alpha a scalar:

```
X = dot_product(A, B, dim=1)
```

```
X = X + dot_product(A, transpose(B), dim=1)
```

```
X = X - alpha * dot_product(spread(V, dim=1, ncopies=N), B, dim=2)
```

2.9.4.12 Fortran RANDOM_NUMBER Intrinsic Function

The interface for RANDOM_NUMBER is actually in the curandex module, but that is included/used within the cutensorex module.

The Fortran subroutine RANDOM_NUMBER returns random numbers between the values of 0.0 and 1.0. This interface is provided as a convenience, and has not undergone extensive testing. When you pass arrays with the device or managed attribute, the subroutine will invoke a cuRAND library function to generate the values. Some additional work was done to support the types real(2), real(4), real(8), complex(4), and complex(8), some of which is non-standard. Only arrays of 1 - 3 dimensions are supported.

For example, if A is a real array and has the device or managed attribute:

```
block; use cutensorex
CALL RANDOM_NUMBER(A)
end block
```

The default generator used by the curandex module is CURAND_RNG_PSEUDO_XORWOW.

Some helper functions are provided in the curandex module to fine-tune the cuRAND library random number generator, which should be self-explanatory:

```
integer(4) function curandExSetCurandGenerator(g)
type(curandGenerator) :: g
end function

function curandExGetCurandGenerator() result(s)
type(curandGenerator) :: s
end function

integer(4) function curandExSetStream(stream)
integer(kind=cuda_stream_kind), value :: stream
end function

function curandExGetStream() result(s)
integer(kind=cuda_stream_kind) :: s
end function
```

2.9.5. Other CUDA Library Host Modules

Please refer to the *NVIDIA Fortran CUDA Interfaces* document for more detailed information on the Fortran interfaces to CUDA libraries. This section discusses some of the more-commonly used interfaces and libraries.

NVIDIA provides a module which defines interfaces to the CUBLAS Library from NVIDIA CUDA Fortran. These interfaces are made accessible by placing the following statement in the CUDA Fortran host-code program unit.

```
use cublas
```

The interfaces are currently in three forms:

- Overloaded traditional BLAS interfaces which take device arrays as arguments rather than host arrays, i.e.

```
call saxpy(n, a, x, incx, y, incy)
```

where the arguments x and y have the device attribute.

- Portable legacy CUBLAS interfaces which interface directly with CUBLAS versions < 4.0, i.e.

```
call cublasSaxpy(n, a, x, incx, y, incy)
```

where the arguments x and y must have the device attribute.

- New CUBLAS 4.0+ interfaces with access to all features of the new library.

  These interfaces are all in the form of function calls, take a handle as the first argument, and pass many scalar arguments and results by reference, i.e.

```
istat = cublasSaxpy_v2(h, n, a, x, incx, y, incy)
```

  In the case of `saxpy`, users now have the option of having `a` reside either on the host or device. Functions which traditionally return a scalar, such as `sdot()` and `isamax()`, now take an extra argument for returning the result. Functions which traditionally take a `character*1` argument, such as `t` or `n` to control transposing, now take an integer value defined in the `cublas` module.

  To support the third form, a derived type named `cublasHandle` is defined in the cublas module. You can define a variable of this type using

```
type(cublasHandle) :: h
```

  Initialize it by passing it to the `cublasCreate` function.

When using CUBLAS 4.0 and higher, the cublas module properly generates handles for the first two forms from serial and OpenMP parallel regions.

Intermixing the three forms is permitted. To access the handles used internally in the cublas module use:

```
h = cublasGetHandle()
```

The following form “istat = cublasGetHandle(h)” is also supported.

```
istat = cublasGetHandle(h)
```

Assignment and tests for equality and inequality are supported for the `cublasHandle` type.

CUDA 4.0+ helper functions defined in the cublas module:

```
integer function cublasCreate(handle)
integer function cublasDestroy(handle)
integer function cublasGetVersion(handle, version)
integer function cublasSetStream(handle, stream)
integer function cublasGetStream(handle, stream)
integer function cublasGetPointerMode(handle, mode)
integer function cublasSetPointerMode(handle, mode)
```

Refer to *Cublas Module Example* for an example that demonstrates the use of the `cublas` module, the `cublasHandle` type, and the three forms of calls.

NVIDIA provides another module which defines interfaces to the CUFFT Library from NVIDIA CUDA Fortran. These interfaces are made accessible by placing the following statement in the CUDA Fortran host-code program unit.

```
use cufft
```

Here is an example of some code which uses the cufft interfaces:

```
program cufft2dTest
  use cufft
  integer, parameter :: n=450
  complex :: a(n,n),b(n,n)
  complex, device :: a_d(n,n), b_d(n,n)
  integer :: plan, ierr
  real, dimension(3) :: res, exp

  a  = 1; a_d  = a

  ierr = cufftPlan2D(plan,n,n,CUFFT_C2C)
  ierr = ierr + cufftExecC2C(plan,a_d,b_d,CUFFT_FORWARD)
  ierr = ierr + cufftExecC2C(plan,b_d,b_d,CUFFT_INVERSE)

  b = b_d
  res(1) = maxval(abs(a-b/(n*n)))
  print *,'Max error C2C: ', res(1)
```

The distribution also contains a module which defines interfaces to the CUSPARSE Library from NVIDIA CUDA Fortran. These interfaces are made explicit by placing the following statement in the CUDA Fortran host-code program unit.

```
use cusparse
```

In addition to the function interfaces, there are several important derived types and constants which are defined in the cusparse module. Here is an example of their use:

```
! Compile with "nvfortran testLevel3.cuf -cudalib=cusparse"
program testLevel3
  use cudafor
  use cusparse

  implicit none

  integer, parameter :: nd = 20 ! # rows/cols in dense matrix

  type(cusparseHandle) :: h
  type(cusparseMatDescr) :: descrA
  type(cusparseSolveAnalysisInfo) :: saInfo
  integer :: status, version, mode, i

  ! D-data
  ! dense
  real(8) :: DAde(nd,nd), DBde(nd,nd), DCde(nd,nd), DmaxErr
  real(8), device :: DAde_d(nd,nd), DBde_d(nd,nd), DCde_d(nd,nd)
  ! csr
  real(8) :: csrValDA(nd)
  real(8), device :: csrValDA_d(nd)
  real(8) :: Dalpha, Dbeta
  real(8), device :: Dalpha_d, Dbeta_d

  ! integer data common to all data types
  integer :: nnz
```

(continues on next page)

(continued from previous page)

```
  integer :: nnzPerRowA(nd), csrRowPtrA(nd+1), csrColIndA(nd)
  integer, device :: nnzPerRowA_d(nd), csrRowPtrA_d(nd+1), csrColIndA_d(nd)

  ! initalization

  status = cusparseCreate(h)
  status = cusparseGetVersion(h, version)
  write(*,*) '... version:', version

  status = cusparseCreateMatDescr(descrA)
  status = cusparseSetMatType(descrA, CUSPARSE_MATRIX_TYPE_GENERAL)
  status = cusparseSetMatIndexBase(descrA, CUSPARSE_INDEX_BASE_ONE)
  status = cusparseCreateSolveAnalysisInfo(saInfo)

  ! Initialize matrix (Identity)

  DAde = 0.0
  i = 1, nd
      DAde(i,i) = 1.0
  end do
  DAde_d = DAde
  call random_number(DBde)
  DBde_d = DBde

  ! convert from dense to csr
  status = cusparseDnnz_v2(h, CUSPARSE_DIRECTION_ROW, nd, nd, descrA, &
      DAde_d, nd, nnzPerRowA_d, nnz)
  status = cusparseDdense2csr(h, nd, nd, descrA, DAde_d, nd, nnzPerRowA_d, &
      csrValDA_d, csrRowPtrA_d, csrColIndA_d)

  ! csrmm HPM
  Dalpha = 1.0
  Dbeta = 0.0
  status = cusparseDcsrmm(h, CUSPARSE_OPERATION_NON_TRANSPOSE, nd, nd, nd, &
      nnz, Dalpha, descrA, csrValDA_d, csrRowPtrA_d, csrColIndA_d, DBde_d, &
      nd, Dbeta, DCde_d, nd)
  if (status /= CUSPARSE_STATUS_SUCCESS) write (*,*) 'CSRMM Error:',status

  DCde = DCde_d
  DmaxErr = maxval(abs(DCde-DBde))

  status = cusparseDestroy(h)
  write(*,*) 'cusparseDestroy', status, DmaxErr

end program testLevel3
```

Chapter 3. Runtime APIs

The system module `cudafor` defines the interfaces to the Runtime API routines.

For a complete explanation of the purpose and function of each routine in this chapter, refer to docs.nvidia.com/cuda/cuda-runtime-api.

Most of the runtime API routines are integer functions that return an error code; they return a value of zero if the call was successful, and a nonzero value if there was an error. To interpret the error codes, refer to *Error Handling*.

Unless a specific kind is provided, the plain integer type implies integer(4) and the plain real type implies real(4).

3.1. Initialization

No explicit initialization is required; the runtime initializes and connects to the device the first time a runtime routine is called or a device array is allocated.

Tip: When doing timing runs, be aware that initialization can add some overhead.

3.2. Device Management

Use the functions in this section for device management.

For a complete explanation of the purpose and function of each routine listed here, refer to the Device Management section at https://docs.nvidia.com/cuda/cuda-runtime-api.

3.2.1. cudaChooseDevice

```
integer function cudaChooseDevice ( devnum, prop )
     integer, intent(out) :: devnum
     type(cudadeviceprop), intent(in) :: prop
```

`cudaChooseDevice` assigns the device number that best matches the properties given in `prop` to its first argument.

3.2.2. cudaDeviceGetAttribute

```
integer function cudaDeviceGetAttribute ( val, attribute, devicenum )
     integer, intent(out) :: val
     integer, intent(in)  :: attribute
     integer, intent(in)  :: devicenum
```

`cudaDeviceGetAttribute` returns information about the device. Specific information returned is determined by the `attribute` argument, for the specified device number.

3.2.3. cudaDeviceGetCacheConfig

```
integer function cudaDeviceGetCacheConfig ( cacheconfig )
     integer, intent(out) :: cacheconfig
```

`cudaDeviceGetCacheConfig` returns the preferred cache configuration for the current device. Current possible cache configurations are defined to be `cudaFuncCachePreferNone`, `cudaFuncCachePreferShared`, and `cudaFuncCachePreferL1`.

`cudaDeviceGetCacheConfig` is available in device code starting in CUDA 5.0.

3.2.4. cudaDeviceGetLimit

```
integer function cudaDeviceGetLimit( val, limit )
   integer(kind=cuda_count_kind) :: val
   integer :: limit
```

`cudaGetDeviceGetLimit` returns in val the current size of limit. Current possible limit arguments are `cudaLimitStackSize`, `cudaLimitPrintfSize`, and `cudaLimitMallocHeapSize`.

`cudaGetDeviceGetLimit` is available in device code starting in CUDA 5.0.

3.2.5. cudaDeviceGetSharedMemConfig

```
integer function cudaDeviceGetSharedMemConfig ( config )
integer, intent(out) :: config
```

`cudaDeviceGetSharedMemConfig` returns the current size of the shared memory banks on the current device. This routine is for use with devices with configurable shared memory banks, and is supported starting with CUDA 4.2. Current possible shared memory configurations are defined to be `cudaSharedMemBankSizeDefault`, `cudaSharedMemBankSizeFourByte`, and `cudaSharedMemBankSizeEightByte`.

3.2.6. cudaDeviceGetStreamPriorityRange

```
integer function cudaDeviceGetStreamPriorityRange ( leastpriority, greatestpriority )
integer, intent(out) :: leastpriority, greatestpriority
```

`cudaDeviceGetStreamPriorityRange` returns the numerical values that correspond to the least and greatest stream priorities for the current context and device.

3.2.7. cudaDeviceReset

```
integer function cudaDeviceReset()
```

`cudaDeviceReset` resets the current device attached to the current process.

3.2.8. cudaDeviceSetCacheConfig

```
integer function cudaDeviceSetCacheConfig ( cacheconfig )
     integer, intent(in) :: cacheconfig
```

`cudaDeviceSetCacheConfig` sets the current device preferred cache configuration. Current possible cache configurations are defined to be `cudaFuncCachePreferNone`, `cudaFuncCachePreferShared`, and `cudaFuncCachePreferL1`.

3.2.9. cudaDeviceSetLimit

```
integer function cudaDeviceSetLimit( limit, val )
  integer :: limit

  integer(kind=cuda_count_kind) :: val
```

`cudaGetDeviceSetLimit` sets the limit of the current device to val. Current possible limit arguments are `cudaLimitStackSize`, `cudaLimitPrintfSize`, and `cudaLimitMallocHeapSize`.

3.2.10. cudaDeviceSetSharedMemConfig

```
integer function cudaDeviceSetSharedMemConfig ( config )
    integer, intent(in) :: config
```

`cudaDeviceSetSharedMemConfig` sets the size of the shared memory banks on the current device. This routine is for use with devices with configurable shared memory banks, and is supported starting with CUDA 4.2. Current possible shared memory configurations are defined to be `cudaSharedMemBankSizeDefault`, `cudaSharedMemBankSizeFourByte`, and `cudaSharedMemBankSizeEightByte`.

3.2.11. cudaDeviceSynchronize

```
integer function cudaDeviceSynchronize()
```

`cudaDeviceSynchronize` blocks the current device until all preceding requested tasks have completed.

`cudaDeviceSynchronize` was available in device code starting in CUDA 5.0.

`cudaDeviceSynchronize` has been removed from CUDA Fortran device code starting with the NVHPC 22.11 Release as it is no longer supported there in the CUDA Programming Model.

3.2.12. cudaGetDevice

```
integer function cudaGetDevice( devnum )
    integer, intent(out) :: devnum
```

`cudaGetDevice` assigns the device number associated with this host thread to its first argument.

`cudaGetDevice` is available in device code starting in CUDA 5.0.

3.2.13. cudaGetDeviceCount

```
integer function cudaGetDeviceCount( numdev )
    integer, intent(out) :: numdev
```

`cudaGetDeviceCount` assigns the number of available devices to its first argument.

`cudaGetDeviceCount` is available in device code starting in CUDA 5.0.

3.2.14. cudaGetDeviceProperties

```
integer function cudaGetDeviceProperties( prop, devnum )
     type(cudadeviceprop), intent(out) :: prop
     integer, intent(in) :: devnum
```

`cudaGetDeviceProperties` returns the properties of a given device.

`cudaGetDeviceProperties` is available in device code starting in CUDA 5.0.

3.2.15. cudaSetDevice

```
integer function cudaSetDevice( devnum )
    integer, intent(in) :: devnum
```

`cudaSetDevice` selects the device to associate with this host thread.

3.2.16. cudaSetDeviceFlags

```
integer function cudaSetDevice( flags )
    integer, intent(in) :: flags
```

`cudaSetDeviceFlags` records how the CUDA runtime interacts with this host thread.

3.2.17. cudaSetValidDevices

```
integer function cudaSetValidDevices( devices, numdev )
    integer :: numdev, devices(numdev)
```

`cudaSetValidDevices` sets a list of valid devices for CUDA execution in priority order as specified in the devices array.

3.3. Thread Management

Sometimes threads within a block access the same addresses in shared or global memory, thus creating potential read-after-write, write-after-read, or write-after-write hazards for some of these memory accesses. To avoid these potential issues, use the functions in this section for thread management. These functions have been deprecated beginning in CUDA 4.0.

3.3.1. cudaThreadExit

```
integer function cudaThreadExit()
```

`cudaThreadExit` explicitly cleans up all runtime-related CUDA resources associated with the host thread. Any subsequent CUDA calls or operations will reinitialize the runtime.

Calling cudaThreadExit is optional; it is implicitly called when the host thread exits.

3.3.2. cudaThreadSynchronize

```
integer function cudaThreadSynchronize()
```

`cudaThreadSynchronize` blocks execution of the host subprogram until all preceding kernels and operations are complete. It may return an error condition if one of the preceding operations fails.

Note: This function is deprecated because its name does not reflect its behavior. Its functionality is identical to the non-deprecated function `cudaDeviceSynchronize()`, which you should use instead.

3.4. Error Handling

Use the functions in this section for error handling.

For a complete explanation of the purpose and function of each routine listed here, refer to the Error Handling section at https://docs.nvidia.com/cuda/cuda-runtime-api.

3.4.1. cudaGetErrorString

```
function cudaGetErrorString( errcode )
    integer(4), intent(in) :: errcode
    character*(*) :: cudaGetErrorString
```

`cudaGetErrorString` returns the message string associated with the given error code.

3.4.2. cudaGetLastError

```
integer function cudaGetLastError()
```

`cudaGetLastError` returns the error code that was most recently returned from any runtime call in this host thread.

3.4.3. cudaPeekAtLastError

```
integer function cudaPeekAtLastError()
```

`cudaPeekAtLastError` returns the last error code that has been produced by the CUDA runtime without resetting the error code to `cudaSuccess` like `cudaGetLastError`.

3.5. Stream Management

Use the functions in this section for stream management.

For a complete explanation of the purpose and function of each routine listed here, refer to the Stream Management section at https://docs.nvidia.com/cuda/cuda-runtime-api.

3.5.1. cudaforGetDefaultStream

```
integer(kind=cuda_stream_kind) function cudaforGetDefaultStream( devptr )
```

`cudaforGetDefaultStream` returns the default stream which has been associated with a thread, managed variable, or device variable via a call to `cudaforSetDefaultStream`. `devptr` may be any managed or device scalar or array of a supported type specified in *Device Code Intrinsic Datatypes*. The `devptr` argument is optional; if it is not specified, the function returns the stream tied to the thread, or zero (the default stream).

Streams values returned from `cudaforGetDefaultStream` can be used as the argument to other CUDA libraries, such as the routines `cublasSetStream()`, `cufftSetStream()`, and `cusparseSetStream()`.

3.5.2. cudaforSetDefaultStream

```
integer function cudaforSetDefaultStream( devptr, stream )
    integer(kind=cuda_stream_kind), intent(in) :: stream
```

`cudaforSetDefaultStream` sets the default stream for all subsequent high-level CUDA Fortran operations on managed or device data initiated by that CPU thread. The specific operations affected with managed data are allocatation via the Fortran allocate statement, assignment (both memset and memcpy types), CUF Kernel and global kernel launches, and sum(), maxval(), and minval() reduction operations. `devptr` may be any managed or device scalar or array of a supported type specified in *Device Code Intrinsic Datatypes*. The `devptr` argument is optional; if it is not specified, the function ties the specified stream to all subsequent, allowable, high-level operations executing on that thread.

3.5.3. cudaStreamAttachMemAsync

```
integer function cudaStreamAttachMemAsync( stream, devptr, length, flags )
    integer(kind=cuda_stream_kind), intent(in) :: stream
    integer(kind=cuda_count_kind), optional, intent(in) :: length
    integer, optional, intent(in) :: flags
```

`cudaStreamAttachMemAsync` initiates a stream operation to attach the managed allocation starting at address `devptr` to the specified stream. `devptr` may be any managed scalar or array of a supported type specified in *Device Code Intrinsic Datatypes*. The argument `len` is optional, but currently must be zero. The flags argument must be `cudaMemAttachGlobal`, `cudMemAttachHost`, or `cudMemAttachSingle`.

`cudaStreamAttachMemAsync` is available starting in CUDA 6.0.

3.5.4. cudaStreamCreate

```
integer function cudaStreamCreate( stream )
    integer(kind=cuda_stream_kind), intent(out) :: stream
```

`cudaStreamCreate` creates an asynchronous `stream` and assigns its identifier to its first argument.

3.5.5. cudaStreamCreateWithFlags

```
integer function cudaStreamCreateWithFlags( stream, flags )
    integer(kind=cuda_stream_kind), intent(out) :: stream
    integer, intent(in) :: flags
```

`cudaStreamCreateWithFlags` creates an asynchronous `stream` and assigns its identifier to its first argument. Valid values for flags are `cudaStreamDefault` or `cudaStreamNonBlocking`.

`cudaStreamCreateWithFlags` is available in device code starting in CUDA 5.0.

3.5.6. cudaStreamCreateWithPriority

```
integer function cudaStreamCreateWithPriority( stream, flags, priority )
    integer(kind=cuda_stream_kind), intent(out) :: stream
    integer, intent(in) :: flags, priority
```

`cudaStreamCreateWithPriority` creates an asynchronous `stream` and assigns its identifier to its first argument. Valid values for flags are `cudaStreamDefault` or `cudaStreamNonBlocking`. Lower values for priority represent higher priorities. Work in a higher priority stream may preempt work already executing in a low priority stream.

3.5.7. cudaStreamDestroy

```
integer function cudaStreamDestroy( stream )
    integer(kind=cuda_stream_kind), intent(in) :: stream
```

`cudaStreamDestroy` releases any resources associated with the given `stream`.

`cudaStreamDestroy` is available in device code starting in CUDA 5.0.

3.5.8. cudaStreamGetPriority

```
integer function cudaStreamGetPriority( stream, priority )
    integer(kind=cuda_stream_kind), intent(in) :: stream
    integer, intent(out) :: priority
```

`cudaStreamGetPriority` queries and returns the priority of the given stream in `priority`.

3.5.9. cudaStreamQuery

```
integer function cudaStreamQuery( stream )
    integer(kind=cuda_stream_kind), intent(in) :: stream
```

`cudaStreamQuery` tests whether all operations enqueued to the selected `stream` are complete; it returns zero (success) if all operations are complete, and the value `cudaErrorNotReady` if not. It may also return another error condition if some asynchronous operations failed.

3.5.10. cudaStreamSynchronize

```
integer function cudaStreamSynchronize( stream )
    integer(kind=cuda_stream_kind), intent(in) :: stream
```

`cudaStreamSynchronize` blocks execution of the host subprogram until all preceding kernels and operations associated with the given `stream` are complete. It may return error codes from previous, asynchronous operations.

3.5.11. cudaStreamWaitEvent

```
integer function cudaStreamWaitEvent( stream, event, flags )
    integer(kind=cuda_stream_kind) :: stream
    type(cudaEvent), intent(in) :: event
    integer :: flags
```

`cudaStreamWaitEvent` blocks execution on all work submitted on the `stream` until the `event` reports completion.

`cudaStreamWaitEvent` is available in device code starting in CUDA 5.0.

3.6. Event Management

Use the functions in this section to manage events.

For a complete explanation of the purpose and function of each routine listed here, refer to the Event Management section at https://docs.nvidia.com/cuda/cuda-runtime-api.

3.6.1. cudaEventCreate

```
integer function cudaEventCreate( event )
    type(cudaEvent), intent(out) :: event
```

`cudaEventCreate` creates an event object and assigns the event identifier to its first argument

3.6.2. cudaEventCreateWithFlags

```
integer function cudaEventCreateWithFlags( event, flags )
    type(cudaEvent), intent(out) :: event
    integer :: flags
```

`cudaEventCreateWithFlags` creates an event object with the specified flags. Current flags supported are `cudaEventDefault`, `cudaEventBlockingSync`, and `cudaEventDisableTiming`.

`cudaEventCreateWithFlags` is available in device code starting in CUDA 5.0.

3.6.3. cudaEventDestroy

```
integer function cudaEventDestroy( event )
    type(cudaEvent), intent(in) :: event
```

`cudaEventDestroy` destroys the resources associated with an event object.

`cudaEventDestroy` is available in device code starting in CUDA 5.0.

3.6.4. cudaEventElapsedTime

```
integer function cudaEventElapsedTime( time, start, end)
    real :: time
    type(cudaEvent), intent() :: start, end
```

`cudaEventElapsedTime` computes the elapsed time between two events (in milliseconds). It returns `cudaErrorInvalidValue` if either event has not yet been recorded. This function is only valid with events recorded on stream zero.

3.6.5. cudaEventQuery

```
integer function cudaEventQuery( event )
    type(cudaEvent), intent(in) :: event
```

`cudaEventQuery` tests whether an `event` has been recorded. It returns success (zero) if the event has been recorded, and `cudaErrorNotReady` if it has not. It returns `cudaErrorInvalidValue` if `cudaEventRecord` has not been called for this event.

3.6.6. cudaEventRecord

```
integer function cudaEventRecord( event, stream )
    type(cudaEvent), intent(in) :: event
    integer, intent(in) :: stream
```

`cudaEventRecord` issues an operation to the given `stream` to record an `event`. The event is recorded after all preceding operations in the stream are complete. If `stream` is zero, the event is recorded after all preceding operations in all streams are complete.

`cudaEventRecord` is available in device code starting in CUDA 5.0.

3.6.7. cudaEventSynchronize

```
integer function cudaEventSynchronize( event )
    type(cudaEvent), intent(in) :: event
```

`cudaEventSynchronize` blocks until the `event` has been recorded. It returns a value of `cudaErrorInvalidValue` if `cudaEventRecord` has not been called for this event.

3.7. Execution Control

CUDA Fortran does not support all API routines which duplicate the functionality of the chevron syntax. Additional functionality which has been provided with later versions of CUDA is available.

For a complete explanation of the purpose and function of each routine listed here, refer to the Execution Control section at https://docs.nv idia.com/cuda/cuda-runtime-api.

3.7.1. cudaFuncGetAttributes

```
integer function cudaFuncGetAttributes( attr, func )
    type(cudaFuncAttributes), intent(out) :: attr
    external :: func
```

cudaFuncGetAttributes gets the attributes for the function named by the func argument, which must be a global function.

cudaFuncGetAttributes is available in device code starting in CUDA 5.0.

3.7.2. cudaFuncSetAttribute

```
integer function cudaFuncSetAttribute( func, attribute, value )
    external :: func
    integer  :: attribute
    integer  :: value
```

cudaFuncSetAttribute sets the attribute for the function named by the func argument, which must be a global function.

3.7.3. cudaFuncSetCacheConfig

```
integer function cudaFuncSetCacheConfig( func, cacheconfig )
    character*(*) :: func
    integer :: cacheconfig
```

cudaFuncSetCacheConfig sets the preferred cache configuration for the function named by the func argument, which must be a global function. Current possible cache configurations are defined to be cudaFuncCachePreferNone, cudaFuncCachePreferShared, and cudaFuncCachePreferL1.

3.7.4. cudaFuncSetSharedMemConfig

```
integer function cudaFuncSetSharedMemConfig( func, cacheconfig )
character*(*) :: func
integer :: cacheconfig
```

cudaFuncSetSharedMemConfig sets the size of the shared memory banks for the function named by the func argument, which must be a global function. This routine is for use with devices with configurable shared memory banks, and is supported starting with CUDA 4.2. Current possible shared memory configurations are defined to be cudaSharedMemBankSizeDefault, cudaSharedMemBankSizeFourByte, and cudaSharedMemBankSizeEightByte

3.7.5. cudaSetDoubleForDevice

```
integer function cudaSetDoubleForDevice( d )
    real(8) :: d
```

`cudaSetDoubleForDevice` sets the argument `d` to an internal representation suitable for devices which do not support double precision arithmetic.

3.7.6. cudaSetDoubleForHost

```
integer function cudaSetDoubleForHost( d )
    real(8) :: d
```

`cudaSetDoubleForHost` sets the argument `d` from an internal representation on devices which do not support double precision arithmetic to the normal host representation.

3.8. Occupancy

The occupancy routines take a global subroutine as an argument and return values related to occupancy, available to be used in kernel launch configurations.

CUDA Fortran has extended the chevron syntax to take a * argument which will call these functions within the runtime, i.e. under the hood. This convenience may not be desirable if the kernel is launched many times as it does invoke some overhead for each call.

Use of the occupancy calls, either explicitly or via the * syntax, is particularly useful when launching grid_global kernels as the launch parameters must be sized to fit on the current device.

Use the functions in this section for explicit occupancy calculations.

For a complete explanation of the purpose and function of each routine listed here, refer to the Occupancy section at https://docs.nvidia.com/cuda/cuda-runtime-api.

3.8.1. cudaOccupancyMaxActiveBlocksPerMultiprocessor

```
integer function cudaOccupancyMaxActiveBlocksPerMultiprocessor( numBlocks, func,
↪blockSize, dynamicSMemSize )
    integer :: numBlocks
    external :: func
    integer :: blockSize
    integer :: dynamicSMemSize
```

`cudaOccupancyMaxActiveBlocksPerMultiprocessor` returns the occupancy, as the number of blocks per multiprocessor, given the global subroutine named by the func argument, the block size (number of threads) the kernel is intended to be launched with, and the amount of dynamic shared memory, in bytes, the kernel is intended to be launched with.

3.8.2. cudaOccupancyMaxActiveBlocksPerMultiprocessorWithFlags

```
integer function cudaOccupancyMaxActiveBlocksPerMultiprocessorWithFlags( numBlocks,
↪func, blockSize, dynamicSMemSize, flags )
    integer :: numBlocks
    external :: func
    integer :: blockSize
    integer :: dynamicSMemSize
    integer :: flags
```

`cudaOccupancyMaxActiveBlocksPerMultiprocessorWithFlags` returns the occupancy, as the number of blocks per multiprocessor, given the global subroutine named by the func argument, the block size (number of threads) the kernel is intended to be launched with, and the amount of dynamic shared memory, in bytes, the kernel is intended to be launched with, for the specified flags.

3.8.3. cudaOccupancyMaxPotentialClusterSize

```
integer function cudaOccupancyMaxPotentialClusterSize( csize, func, config )
    integer :: csize
    external :: func
    type(cudaLaunchConfig) :: config
```

`cudaOccupancyMaxPotentialClusterSize` returns the maximum cluster size that can be launched, given the input kernel, func, and launch configuration specified in the config argument.

3.8.4. cudaOccupancyMaxActiveClusters

```
integer function cudaOccupancyMaxActiveClusters( maxc, func, config )
    integer :: maxc
    external :: func
    type(cudaLaunchConfig) :: config
```

`cudaOccupancyMaxActiveClusters` returns the maximum number of clusters that could co-exist on the target device. The cluster size can be set in the config argument.

3.9. Memory Management

Many of the memory management routines can take device arrays as arguments. Some can also take C types, provided through the Fortran 2003 `iso_c_binding` module, as arguments to simplify interfacing to existing CUDA C code.

CUDA Fortran has extended the F2003 derived type `TYPE(C_PTR)` by providing a C device pointer, defined in the `cudafor` module, as `TYPE(C_DEVPTR)`. Consistent use of `TYPE(C_PTR)` and `TYPE(C_DEVPTR)`, as well as consistency checks between Fortran device arrays and host arrays, should be of benefit.

Currently, it is possible to construct a Fortran device array out of a `TYPE(C_DEVPTR)` by using an extension of the `iso_c_binding` subroutine c_f_pointer. Under CUDA Fortran, c_f_pointer will take a

TYPE(C_DEVPTR) as the first argument, an allocatable device array as the second argument, a shape as the third argument, and in effect transfer the allocation to the Fortran array. Similarly, there is also a function C_DEVLOC() defined which will create a TYPE(C_DEVPTR) that holds the C address of the Fortran device array argument. Both of these features are subject to change when, in the future, proper Fortran pointers for device data are supported.

Use the functions in this section for memory management.

For a complete explanation of the purpose and function of each routine listed here, refer to the Memory Management section at https://docs.nvidia.com/cuda/cuda-runtime-api.

3.9.1. cudaFree

```
integer function cudaFree(devptr)
```

cudaFree deallocates data on the device. devptr may be any allocatable device array of a supported type specified in *Device Code Intrinsic Datatypes*. Or, devptr may be of TYPE(C_DEVPTR).

cudaFree is available in device code starting in CUDA 5.0.

3.9.2. cudaFreeArray

```
integer function cudaFreeArray(carray)
    type(cudaArrayPtr) :: carray
```

cudaFreeArray frees an array that was allocated on the device.

3.9.3. cudaFreeAsync

```
integer function cudaFreeAsync(devptr, stream)
```

cudaFreeAsync deallocates data on the device, asynchronously, on the specified stream. devptr may be any allocatable device array of a supported type specified in *Device Code Intrinsic Datatypes*. Or, devptr may be of TYPE(C_DEVPTR). The stream argument is an integer of kind=cuda_stream_kind.

cudaFreeAsync is available starting in CUDA 11.2.

3.9.4. cudaFreeHost

```
integer function cudaFreeHost(hostptr)
    type(C_PTR) :: hostptr
```

cudaFreeHost deallocates pinned memory on the host allocated with cudaMalloHost.

3.9.5. cudaGetSymbolAddress

```
integer function cudaGetSymbolAddress(devptr, symbol)
    type(C_DEVPTR) :: devptr
    type(c_ptr) :: symbol
```

`cudaGetSymbolAddress` returns in the `devptr` argument the address of `symbol` on the device. A `symbol` can be set to an external device name via a character string.

The following code sequence initializes a global device array 'vx' from a CUDA C kernel:

```
type(c_ptr) :: csvx
type(c_devptr) :: cdvx
real, allocatable, device :: vx(:)
csvx = 'vx'
Istat = cudaGetSymbolAddress(cdvx, csvx)
Call c_f_pointer(cdvx, vx, 100)
Vx = 0.0
```

3.9.6. cudaGetSymbolSize

```
integer function cudaGetSymbolSize(size, symbol)
    integer :: size
    type(c_ptr) :: symbol
```

`cudaGetSymbolSize` sets the variable `size` to the size of a device area in global or constant memory space referenced by the `symbol`.

3.9.7. cudaHostAlloc

```
integer function cudaHostAlloc(hostptr, size, flags)
    type(C_PTR) :: hostptr
    integer :: size, flags
```

`cudaHostAlloc` allocates pinned memory on the host. It returns in `hostptr` the address of the page-locked allocation, or returns an error if the memory is unavailable. `Size` is in bytes. The `flags` argument enables different options to be specified that affect the allocation. The normal iso_c_binding subroutine c_f_pointer can be used to move the type(c_ptr) to a Fortran pointer.

3.9.8. cudaHostGetDevicePointer

```
integer function cudaHostGetDevicePointer(devptr, hostptr, flags)
    type(C_DEVPTR) :: devptr
    type(C_PTR) :: hostptr
    integer :: flags
```

`cudaHostGetDevicePointer` returns a pointer to a device memory address corresponding to the pinned memory on the host. `hostptr` is a pinned memory buffer that was allocated via `cudaHostAlloc()`. It returns in `devptr` an address that can be passed to, and read and written by, a kernel which

runs on the device. The `flags` argument is provided for future releases. The normal iso_c_binding subroutine c_f_pointer can be used to move the type(c_devptr)to a device array.

3.9.9. cudaHostGetFlags

```
integer function cudaHostGetFlags(flags, hostptr)
    integer :: flags
    type(C_PTR) :: hostptr
```

`cudaHostGetFlags` returns the flags associated with a host pointer.

3.9.10. cudaHostRegister

```
integer function cudaHostRegister(hostptr, count, flags)
    integer :: flags
    type(C_PTR) :: hostptr
```

`cudaHostRegister` page-locks the memory associated with the host pointer and of size provided by the `count` argument, according to the `flags` argument.

3.9.11. cudaHostUnregister

```
integer function cudaHostRegister(hostptr)
    type(C_PTR) :: hostptr
```

`cudaHostUnregister` unmaps the memory associated with the host pointer and makes it page-able again. The argument hostptr must be the same as was used with `cudaHostRegister`.

3.9.12. cudaMalloc

```
integer function cudaMalloc(devptr, count)
```

`cudaMalloc` allocates data on the device. `devptr` may be any allocatable, one-dimensional device array of a supported type specified in *Device Code Intrinsic Datatypes*. The `count` is in terms of elements. Or, `devptr` may be of TYPE(C_DEVPTR), in which case the `count` is in bytes.

`cudaMalloc` is available in device code starting in CUDA 5.0.

3.9.13. cudaMallocArray

```
integer function cudaMallocArray(carray, cdesc, width, height)
    type(cudaArrayPtr) :: carray
    type(cudaChannelFormatDesc) :: cdesc
    integer :: width, height
```

`cudaMallocArray` allocates a data array on the device.

3.9.14. cudaMallocAsync

```
integer function cudaMallocAsync(devptr, count, stream)
```

`cudaMallocAsync` allocates data on the device, asynchronously, on the specified stream. `devptr` may be any allocatable, one-dimensional device array of a supported type specified in *Device Code Intrinsic Datatypes*. The `count` is in terms of elements. Or, `devptr` may be of TYPE(C_DEVPTR), in which case the `count` is in bytes. The `stream` argument is an integer of kind=cuda_stream_kind.

`cudaFreeAsync` is available starting in CUDA 11.2.

3.9.15. cudaMallocManaged

```
integer function cudaMallocManaged(devptr, count, flags)
```

`cudaMallocManaged` allocates data that will be managed by the unified memory system. `devptr` may be any allocatable, one-dimensional managed array of a supported type specified in *Device Code Intrinsic Datatypes*. The `count` is in terms of elements. Or, `devptr` may be of TYPE(C_DEVPTR), in which case the `count` is in bytes. The flags argument must be either `cudaMemAttachGlobal` or `cudaMemAttachHost`.

`cudaMallocManaged` is available starting in CUDA 6.0.

3.9.16. cudaMallocPitch

```
integer function cudaMallocPitch(devptr, pitch, width, height)
```

`cudaMallocPitch` allocates data on the device. `devptr` may be any allocatable, two-dimensional device array of a supported type specified in *Device Code Intrinsic Datatypes*. The `width` is in terms of number of elements. The `height` is an integer.

`cudaMallocPitch` may pad the data, and the padded width is returned in the variable `pitch`. `Pitch` is an integer of kind=cuda_count_kind. `devptr` may also be of TYPE(C_DEVPTR), in which case the integer values are expressed in bytes.

3.9.17. cudaMalloc3D

```
integer function cudaMalloc3D(pitchptr, cext)
    type(cudaPitchedPtr), intent(out) :: pitchptr
    type(cudaExtent), intent(in) :: cext
```

cudaMalloc3D allocates data on the device. pitchptr is a derived type defined in the cudafor module. cext is also a derived type which holds the extents of the allocated array. Alternatively, pitchptr may be any allocatable, three-dimensional device array of a supported type specified in *Datatypes Allowed*.

3.9.18. cudaMalloc3DArray

```
integer function cudaMalloc3DArray(carray, cdesc, cext)
    type(cudaArrayPtr) :: carray
    type(cudaChannelFormatDesc) :: cdesc
    type(cudaExtent) :: cext
```

cudaMalloc3DArray allocates array data on the device.

3.9.19. cudaMemAdvise

```
integer function cudaMemAdvise(devptr, count, advice, device)
```

cudaMemAdvise lends advice to the Unified Memory subsystem about the expected usage pattern for the specified memory range. devptr may be any managed memory scalar or array, of a supported type specified in *Device Code Intrinsic Datatypes*. The count is in terms of elements. Alternatively, devptr may be of TYPE(C_DEVPTR), in which case the count is in terms of bytes.

Current possible values for advice, defined in the cudafor module, are cudaMemAdviseSetReadMostly, cudaMemAdviseUnsetReadMostly, cudaMemAdviseSetPreferredLocation, cudaMemAdviseUnsetPreferredLocation, cudaMemAdviseSetAccessedBy, and cudaMemAdviseUnsetAccessedBy

The device argument specifies the destination device. Passing in cudaCpuDeviceId for the device, which is defined as a parameter in the cudafor module, will set advice for the CPU.

3.9.20. cudaMemcpy

```
integer function cudaMemcpy(dst, src, count, kdir)
```

cudaMemcpy copies data from one location to another. dst and src may be any device or host, scalar or array, of a supported type specified in *Device Code Intrinsic Datatypes*. The count is in terms of elements. kdir may be optional; for more information, refer to *Data Transfer Using Runtime Routines*. If kdir is specified, it must be one of the defined enums cudaMemcpyHostToDevice, cudaMemcpyDeviceToHost, or cudaMemcpyDeviceToDevice. Alternatively, dst and src may be of TYPE(C_DEVPTR) or TYPE(C_PTR), in which case the count is in term of bytes.

cudaMemcpy is available in device code starting in CUDA 5.0.

3.9.21. cudaMemcpyArrayToArray

```
integer function cudaMemcpyArrayToArray(dsta, dstx, dsty,
                        srca, srcx, srcy, count, kdir)
    type(cudaArrayPtr) :: dsta, srca
    integer :: dstx, dsty, srcx, srcy, count, kdir
```

`cudaMemcpyArrayToArray` copies array data to and from the device.

3.9.22. cudaMemcpyAsync

```
integer function cudaMemcpyAsync(dst, src, count, kdir, stream)
```

`cudaMemcpyAsync` copies data from one location to another. `dst` and `src` may be any device or host, scalar or array, of a supported type specified in *Device Code Intrinsic Datatypes*. The `count` is in terms of elements. `kdir` may be optional; for more information, refer to *Data Transfer Using Runtime Routines*. If `kdir` is specified, it must be one of the defined enums `cudaMemcpyHostToDevice`, `cudaMemcpyDeviceToHost`, or `cudaMemcpyDeviceToDevice`. Alternatively, `dst` and `src` may be of TYPE(C_DEVPTR) or TYPE(C_PTR), in which case the `count` is in term of bytes.

This function operates on page-locked host memory only. The copy can be associated with a stream by passing a non-zero stream argument; otherwise the `stream` argument is optional and defaults to zero.

`cudaMemcpyAsync` is available in device code starting in CUDA 5.0.

3.9.23. cudaMemcpyFromArray

```
integer function cudaMemcpyFromArray(dst, srca, srcx, srcy, count, kdir)
    type(cudaArrayPtr) :: srca
    integer :: dstx, dsty, count, kdir
```

`cudaMemcpyFromArray` copies array data to and from the device.

3.9.24. cudaMemcpyFromSymbol

```
integer function cudaMemcpyFromSymbol(dst, symbol, count, offset, kdir, stream)
    type(c_ptr) :: symbol
    integer :: count, offset, kdir
    integer, optional :: stream
```

`cudaMemcpyFromSymbol` copies data from a device area in global or constant memory space referenced by a `symbol` to a destination on the host. `dst` may be any host scalar or array of a supported type specified in *Datatypes Allowed*. The `count` is in terms of elements.

3.9.25. cudaMemcpyFromSymbolAsync

```
integer function cudaMemcpyFromSymbolAsync(dst, symbol, count, offset, kdir, stream)
    type(c_ptr) :: symbol
    integer :: count, offset, kdir
    integer, optional :: stream
```

`cudaMemcpyFromSymbolASYNC` copies data from a device area in global or constant memory space referenced by a `symbol` to a destination on the host. `dst` may be any host scalar or array of a supported type specified in *Datatypes Allowed*. The `count` is in terms of elements.

`cudaMemcpyFromSymbolASYNC` is asynchronous with respect to the host, This function operates on page-locked host memory only. The copy can be associated with a stream by passing a non-zero stream argument.

3.9.26. cudaMemcpyPeer

```
integer function cudaMemcpyPeer(dst, dstdev, src, srcdev, count)
```

`cudaMemcpyPeer` copies data from one device to another. `dst` and `src` may be any device scalar or array, of a supported type specified in *Device Code Intrinsic Datatypes*. The `count` is in terms of elements. Alternatively, `dst` and `src` may be of TYPE(C_DEVPTR), in which case the `count` is in term of bytes.

3.9.27. cudaMemcpyPeerAsync

```
integer function cudaMemcpyPeerAsync(dst, dstdev, src, srcdev, count, stream)
```

`cudaMemcpyPeerAsync` copies data from one device to another. `dst` and `src` may be any device scalar or array, of a supported type specified in *Device Code Intrinsic Datatypes*. The `count` is in terms of elements. Alternatively, dst and src may be of TYPE(C_DEVPTR), in which case the `count` is in term of bytes. The copy can be associated with a stream by passing a non-zero stream argument.

3.9.28. cudaMemcpyToArray

```
integer function cudaMemcpyToArray(dsta, dstx, dsty, src, count, kdir)
    type(cudaArrayPtr) :: dsta
    integer :: dstx, dsty, count, kdir
```

`cudaMemcpyToArray` copies array data to and from the device.

3.9.29. cudaMemcpyToSymbol

```
integer function cudaMemcpyToSymbol(symbol, src, count, offset, kdir)
   type(c_ptr) :: symbol
   integer :: count, offset, kdir
```

`cudaMemcpyToSymbol` copies data from the source to a device area in global or constant memory space referenced by a `symbol`. `src` may be any host scalar or array of a supported type as specified in *Device Code Intrinsic Datatypes*. The `count` is in terms of elements.

3.9.30. cudaMemcpyToSymbolAsync

```
integer function cudaMemcpyToSymbolAsync(symbol, src, count, offset, kdir, stream)
    type(c_ptr) :: symbol
    integer :: count, offset, kdir
    integer, optional :: stream
```

`cudaMemcpyToSymbolAsync` copies data from the source to a device area in global or constant memory space referenced by a `symbol`. `src` may be any host scalar or array of a supported type specified in *Datatypes Allowed*. The `count` is in terms of elements.

This function operates on page-locked host memory only. The copy can be associated with a stream by passing a non-zero stream argument.

3.9.31. cudaMemcpy2D

```
integer function cudaMemcpy2D(dst, dpitch, src, spitch, width, height, kdir)
```

`cudaMemcpy2D` copies data from one location to another. `dst` and `src` may be any device or host array, of a supported type specified in *Device Code Intrinsic Datatypes*. The `width` and `height` are in terms of elements. Contrary to how Fortran programmers might view memory layout, and in order to keep compatibility with CUDA C, the width specifies the number of contiguous elements in the leading dimension, and the height is the number of such contiguous sections. `kdir` may be optional; for more information, refer to *Data Transfer Using Runtime Routines*. If `kdir` is specified, it must be one of the defined enums `cudaMemcpyHostToDevice`, `cudaMemcpyDeviceToHost`, or `cudaMemcpyDeviceToDevice`. Alternatively, `dst` and `src` may be of TYPE(C_DEVPTR) or TYPE(C_PTR), in which case the `width` and `height` are in term of bytes.

`cudaMemcpy2D` is available in device code starting in CUDA 5.0.

3.9.32. cudaMemcpy2DArrayToArray

```
integer function cudaMemcpy2DArrayToArray(dsta, dstx, dsty,
                     srca, srcx, srcy, width, height, kdir)
    type(cudaArrayPtr) :: dsta, srca
    integer :: dstx, dsty, srcx, srcy, width, height, kdir
```

`cudaMemcpy2DArrayToArray` copies array data to and from the device.

3.9.33. cudaMemcpy2DAsync

```
integer function cudaMemcpy2DAsync(dst, dpitch, src, spitch, width,
              height, kdir, stream)
```

`cudaMemcpy2D` copies data from one location to another. `dst` and `src` may be any device or host array, of a supported type specified in *Device Code Intrinsic Datatypes*. The `width` and `height` are in terms of elements. Contrary to how Fortran programmers might view memory layout, and in order to keep compatibility with CUDA C, the width specifies the number of contiguous elements in the leading dimension, and the height is the number of such contiguous sections. `kdir` may be optional; for more information, refer to *Data Transfer Using Runtime Routines*. If `kdir` is specified, it must be one of the defined enums `cudaMemcpyHostToDevice`, `cudaMemcpyDeviceToHost`, or `cudaMemcpyDeviceToDevice`. Alternatively, `dst` and `src` may be of TYPE(C_DEVPTR) or TYPE(C_PTR), in which case the `width` and `height` are in term of bytes.

This function operates on page-locked host memory only. The copy can be associated with a stream by passing a non-zero `stream` argument, otherwise the `stream` argument is optional and defaults to zero.

`cudaMemcpy2DAsync` is available in device code starting in CUDA 5.0.

3.9.34. cudaMemcpy2DFromArray

```
integer function cudaMemcpy2DFromArray(dst, dpitch, srca, srcx, srcy,
                               width, height, kdir)
    type(cudaArrayPtr) :: srca
    integer :: dpitch, srcx, srcy, width, height, kdir
```

`cudaMemcpy2DFromArray` copies array data to and from the device.

3.9.35. cudaMemcpy2DToArray

```
integer function cudaMemcpy2DToArray(dsta, dstx, dsty, src,
                     spitch, width, height, kdir)
    type(cudaArrayPtr) :: dsta
    integer :: dstx, dsty, spitch, width, height, kdir
```

`cudaMemcpy2DToArray` copies array data to and from the device.

3.9.36. cudaMemcpy3D

```
integer function cudaMemcpy3D(p)
    type(cudaMemcpy3DParms) :: p
```

cudaMemcpy3D copies elements from one 3D array to another as specified by the data held in the derived type p.

3.9.37. cudaMemcpy3DAsync

```
integer function cudaMemcpy3D(p, stream)
    type(cudaMemcpy3DParms) :: p
    integer :: stream
```

cudaMemcpy3DAsync copies elements from one 3D array to another as specified by the data held in the derived type p.

This function operates on page-locked host memory only. The copy can be associated with a stream by passing a non-zero `stream` argument.

3.9.38. cudaMemGetInfo

```
integer function cudaMemGetInfo( free, total )
    integer(kind=cuda_count_kind) :: free, total
```

`cudaMemGetInfo` returns the amount of free and total memory available for allocation on the device. The returned values units are in bytes.

3.9.39. cudaMemPrefetchAsync

```
integer function cudaMemPrefetchAsync(devptr, count, device, stream)
```

`cudaMemPrefetchAsync` prefetches memory to the specified destination device. `devptr` may be any managed memory scalar or array, of a supported type specified in *Device Code Intrinsic Datatypes*. The `count` is in terms of elements. Alternatively, `devptr` may be of TYPE(C_DEVPTR), in which case the `count` is in terms of bytes.

The `device` argument specifies the destination device. The `stream` argument specifies which stream to enqueue the prefetch operation on.

Passing in `cudaCpuDeviceId` for the device, which is defined as a parameter in the cudafor module, will prefetch the data to CPU memory.

3.9.40. cudaMemset

```
integer function cudaMemset(devptr, value, count)
```

`cudaMemset` sets a location or array to the specified value. `devptr` may be any device scalar or array of a supported type specified in *Device Code Intrinsic Datatypes*. The `value` must match in type and kind. The `count` is in terms of elements. Or, `devptr` may be of TYPE(C_DEVPTR), in which case the `count` is in term of bytes, and the lowest byte of `value` is used.

3.9.41. cudaMemsetAsync

```
integer function cudaMemsetAsync(devptr, value, count, stream)
```

`cudaMemsetAsync` sets a location or array to the specified value. `devptr` may be any device scalar or array of a supported type specified in *Device Code Intrinsic Datatypes*. The `value` must match in type and kind. The `count` is in terms of elements. Or, `devptr` may be of TYPE(C_DEVPTR), in which case the `count` is in term of bytes, and the lowest byte of `value` is used. The memory set operation is associated with the stream specified.

3.9.42. cudaMemset2D

```
integer function cudaMemset2D(devptr, pitch, value, width, height)
```

`cudaMemset2D` sets an array to the specified value. `devptr` may be any device array of a supported type specified in *Device Code Intrinsic Datatypes*. The `value` must match in type and kind. The `pitch`, `width`, and `height` are in terms of elements. Or, `devptr` may be of TYPE(C_DEVPTR), in which case the `pitch`, `width`, and `height` are in terms of bytes, and the lowest byte of `value` is used. Contrary to how Fortran programmers might view memory layout, and in order to keep compatibility with CUDA C, the width specifies the number of contiguous elements in the leading dimension, and the height is the number of such contiguous sections.

3.9.43. cudaMemset3D

```
integer function cudaMemset3D(pitchptr, value, cext)
    type(cudaPitchedPtr) :: pitchptr
    integer :: value
    type(cudaExtent) :: cext
```

`cudaMemset3D` sets elements of an array, the extents in each dimension specified by `cext`, which was allocated with `cudaMalloc3D` to a specified value.

3.10. Unified Addressing and Peer Device Memory Access

Use the functions in this section for managing multiple devices from the same process and threads.

For a complete explanation of the purpose and function of each routine listed here, refer to the Unified Addressing and Peer Device Memory Access sections at https://docs.nvidia.com/cuda/cuda-runtime-api.

3.10.1. cudaDeviceCanAccessPeer

```
integer function cudaDeviceCanAccessPeer( canAccessPeer, device, peerDevice )
    integer :: canAccessPeer, device, peerDevice
```

`cudaDeviceCanAccessPeer` returns in `canAccessPeer` the value 1 if the `device` argument can access memory in the device specified by the `peerDevice` argument.

3.10.2. cudaDeviceDisablePeerAccess

```
integer function cudaDeviceDisablePeerAccess ( peerDevice )
    integer :: peerDevice
```

`cudaDeviceDisablePeerAccess` disables the ability to access memory on the device specified by the `peerDevice` argument by the current device.

3.10.3. cudaDeviceEnablePeerAccess

```
integer function cudaDeviceEnablePeerAccess ( peerDevice, flags )
    integer :: peerDevice, flags
```

`cudaDeviceEnablePeerAccess` enables the ability to access memory on the device specified by the `peerDevice` argument by the current device. Currently, flags must be zero.

3.10.4. cudaPointerGetAttributes

```
integer function cudaPointerGetAttributes( attr, ptr )
    type(cudaPointerAttributes), intent(out) :: attr
```

`cudaPointerGetAttributes` returns the attributes of a device or host pointer in the attributes type. `ptr` may be any host or device scalar or array of a supported type specified in *Datatypes Allowed*. It may also be of type C_PTR or C_DEVPTR. It may have the host, device, managed, or pinned attribute.

3.11. Version Management

Use the functions in this section for version management.

For a complete explanation of the purpose and function of each routine listed here, refer to the Version Management section at https://docs.nvidia.com/cuda/cuda-runtime-api.

3.11.1. cudaDriverGetVersion

```
integer function cudaDriverGetVersion(iversion)
    integer :: iversion
```

`cudaDriverGetVersion` returns the version number of the installed CUDA driver as `iversion`. If no driver is installed, then it returns 0 as `iversion`.

This function automatically returns `cudaErrorInvalidValue` if the `iversion` argument is NULL.

3.11.2. cudaRuntimeGetVersion

```
integer function cudaRuntimeGetVersion(iversion)
    integer :: iversion
```

`cudaRuntimeGetVersion` returns the version number of the installed CUDA Runtime as `iversion`.

This function automatically returns `cudaErrorInvalidValue` if the `iversion` argument is NULL.

3.12. Profiling Management

Use the functions in this section for profiling management.

For a complete explanation of the purpose and function of each routine listed here, refer to the Profiler Control section at https://docs.nvidia.com/cuda/cuda-runtime-api.

3.12.1. cudaProfilerStart

```
integer function cudaProfilerStart()
```

`cudaProfilerStart` enables profile collection by the active profilng tool.

3.12.2. cudaProfilerStop

```
integer function cudaProfilerStop()
```

`cudaProfilerStop` disables profile collection by the active profilng tool.

3.13. CUDA Graph Management

Use the functions in this section for CUDA Graph management, the capturing and replaying of CUDA Graphs from CUDA Fortran.

For a complete explanation of the purpose and function of each routine listed here, refer to the CUDA Graph section here https://docs.nvidia.com/cuda/cuda-c-programming-guide.

CUDA Fortran provides three types for managing cuda graphs:

```
TYPE cudaGraph
  TYPE(C_PTR)  :: graph
END TYPE cudaGraph

TYPE cudaGraphExec
  TYPE(C_PTR)  :: exec
END TYPE cudaGraphExec

TYPE cudaGraphNode
  TYPE(C_PTR)  :: node
END TYPE cudaGraphNode
```

3.13.1. cudaGraphCreate

```
integer function cudaGraphCreate( graph, flags )
    type(cudagraph), intent(out) :: graph
    integer :: flags
```

`cudaGraphCreate` creates an empty `graph`.

3.13.2. cudaGraphDestroy

```
integer function cudaGraphDestroy( graph )
    type(cudagraph) :: graph
```

`cudaGraphDestroy` releases any resources associated with the given `graph`.

3.13.3. cudaGraphExecDestroy

```
integer function cudaGraphExecDestroy( graphExec )
    type(cudagraphexec) :: graphExec
```

cudaGraphExecDestroy releases any resources associated with the given graphExec.

3.13.4. cudaGraphInstantiate

```
integer function cudaGraphInstantiate( graphExec, graph, flags )
    type(cudagraphexec), intent(out) :: graphExec
    type(cudagraph), intent(in) :: graph
    integer, intent(in) :: flags
```

cudaGraphInstantiate instantiates the graphExec object from the specified graph.

3.13.5. cudaGraphLaunch

```
integer function cudaGraphLaunch( graphExec, stream )
    type(cudagraphexec) :: graphExec
    integer(kind=cuda_stream_kind), intent(in) :: stream
```

cudaGraphLaunch begins an asynchronous graph launch or replay on the specified stream.

3.13.6. cudaStreamBeginCapture

```
integer function cudaStreamBeginCapture( stream, mode )
    integer(kind=cuda_stream_kind), intent(in) :: stream
    integer, intent(in) :: mode
```

cudaStreamBeginCapture begins a graph capture on the specified stream.

3.13.7. cudaStreamEndCapture

```
integer function cudaStreamEndCapture( stream, graph )
    integer(kind=cuda_stream_kind), intent(in) :: stream
    type(cudagraph), intent(out) :: graph
```

cudaStreamEndCapture ends a graph capture on the specified stream and provides a cudagraph for further use.

3.13.8. cudaStreamIsCapturing

```
integer function cudaStreamIsCapturing( stream, status )
    integer(kind=cuda_stream_kind), intent(in) :: stream
    integer, intent(out) :: status
```

cudaStreamIsCapturing queries the capture status in the specified stream.

Chapter 4. Examples

This section contains examples with source code.

4.1. Matrix Multiplication Example

This example shows a program to compute the product C of two matrices A and B, as follows:

- Each thread block computes one 16x16 submatrix of C;
- Each thread within the block computes one element of the submatrix.

The submatrix size is chosen so the number of threads in a block is a multiple of the warp size (32) and is less than the maximum number of threads per thread block (512).

Each element of the result is the product of one row of A by one column of B. The program computes the products by accumulating submatrix products; it reads a block submatrix of A and a block submatrix of B, accumulates the submatrix product, then moves to the next submatrix of A rowwise and of B columnwise. The program caches the submatrices of A and B in the fast shared memory.

For simplicity, the program assumes the matrix sizes are a multiple of 16, and has not been highly optimized for execution time.

4.1.1. Source Code Listing

Matrix Multiplication

```
! start the module containing the matmul kernel
module mmul_mod
 use cudafor
contains
 ! mmul_kernel computes A*B into C where
 ! A is NxM, B is MxL, C is then NxL
 attributes(global) subroutine mmul_kernel( A, B, C, N, M, L )
  real :: A(N,M), B(M,L), C(N,L)
  integer, value :: N, M, L
  integer :: i, j, kb, k, tx, ty
  ! submatrices stored in shared memory
  real, shared :: Asub(16,16), Bsub(16,16)
  ! the value of C(i,j) being computed
  real :: Cij
```

(continues on next page)

(continued from previous page)

```
  ! Get the thread indices
  tx = threadidx%x
  ty = threadidx%y

! This thread computes C(i,j) = sum(A(i,:) * B(:,j))
i = (blockidx%x-1) * 16 + tx
j = (blockidx%y-1) * 16 + ty
Cij = 0.0
! Do the k loop in chunks of 16, the block size
do kb = 1, M, 16
  ! Fill the submatrices
  ! Each of the 16x16 threads in the thread block
  ! loads one element of Asub and Bsub
   Asub(tx,ty) = A(i,kb+ty-1)
  Bsub(tx,ty) = B(kb+tx-1,j)
  ! Wait until all elements are filled
  call syncthreads()
  ! Multiply the two submatrices
  ! Each of the 16x16 threads accumulates the
  ! dot product for its element of C(i,j)
  do k = 1,16
    Cij = Cij + Asub(tx,k) * Bsub(k,ty)
  enddo
  ! Synchronize to make sure all threads are done
  ! reading the submatrices before overwriting them
  ! in the next iteration of the kb loop
  call syncthreads()
enddo
! Each of the 16x16 threads stores its element
! to the global C array
C(i,j) = Cij
end subroutine mmul_kernel

! The host routine to drive the matrix multiplication
 subroutine mmul( A, B, C )
  real, dimension(:,:) :: A, B, C
  ! allocatable device arrays
  real, device, allocatable, dimension(:,:) :: Adev,Bdev,Cdev
  ! dim3 variables to define the grid and block shapes
  type(dim3) :: dimGrid, dimBlock

! Get the array sizes
N = size( A, 1 )
M = size( A, 2 )
L = size( B, 2 )
! Allocate the device arrays
allocate( Adev(N,M), Bdev(M,L), Cdev(N,L) )
```

```
! Copy A and B to the device
Adev = A(1:N,1:M)
Bdev(:,:) = B(1:M,1:L)

! Create the grid and block dimensions
dimGrid = dim3( N/16, L/16, 1 )
```

(continues on next page)

(continued from previous page)

```
dimBlock = dim3( 16, 16, 1 )
call mmul_kernel<<<dimGrid,dimBlock>>>( Adev, Bdev, Cdev, N, M, L)

 ! Copy the results back and free up memory
 C(1:N,1:L) = Cdev
 deallocate( Adev, Bdev, Cdev )
end subroutine mmul
end module mmul_mod
```

4.1.2. Source Code Description

This source code module `mmul_mod` has two subroutines. The host subroutine `mmul` is a wrapper for the kernel routine `mmul_kernel`.

MMUL

This host subroutine has two input arrays, `A` and `B`, and one output array, `C`, passed as assumed-shape arrays. The routine performs the following operations:

- It determines the size of the matrices in `N`, `M`, and `L`.
- It allocates device memory arrays `Adev`, `Bdev`, and `Cdev`.
- It copies the arrays `A` and `B` to `Adev` and `Bdev` using array assignments.
- It fills `dimGrid` and `dimBlock` to hold the grid and thread block sizes.
- It calls `mmul_kernel` to compute `Cdev` on the device.
- It copies `Cdev` back from device memory to `C`.
- It frees the device memory arrays.

Because the data copy operations are synchronous, no extra synchronization is needed between the copy operations and the kernel launch.

MMUL_KERNEL

This kernel subroutine has two device memory input arrays, `A` and `B`, one device memory output array, `C`, and three scalars giving the array sizes. The thread executing this routine is one of 16x16 threads cooperating in a thread block. This routine computes the dot product of `A(i,:)*B(:,j)` for a particular value of `i` and `j`, depending on the block and thread index.

It performs the following operations:

- It determines the thread indices for this thread.
- It determines the i and j indices, for which element of `` C(i,j) `` it is computing.
- It initializes a scalar in which it will accumulate the dot product.
- It steps through the arrays `A` and `B` in blocks of size 16.
- For each block, it does the following steps:
  - It loads one element of the submatrices of `A` and `B` into shared memory.
  - It synchronizes to make sure both submatrices are loaded by all threads in the block.
  - It accumulates the dot product of its row and column of the submatrices.

- It synchronizes again to make sure all threads are done reading the submatrices before starting the next block.

- Finally, it stores the computed value into the correct element of C.

4.2. Mapped Memory Example

This example demonstrates the use of CUDA API supported in the `cudafor` module for mapping page-locked host memory into the address space of the device. It makes use of the `iso_c_binding c_ptr` type and the `cudafor c_devptr` types to interface to the C routines, then the Fortran `c_f_pointer` call to map the types to Fortran arrays.

Mapped Memory

```
module atest
  contains
    attributes(global) subroutine matrixinc(a,n)
      real, device :: a(n,n)
      integer, value :: n
      i = (blockidx%x-1)*10 + threadidx%x
      j= (blockidx%y-1)*10 + threadidx%y
      if ((i .le. n) .and. (j .le. n)) then
        a(i,j) = a(i,j) + 1.0
      endif
      return
    end subroutine
end module

program test
use  cudafor
use  atest
use, intrinsic :: iso_c_binding

type(c_ptr) :: a
type(c_devptr) :: a_d
real, dimension(:,:), pointer :: fa
real, dimension(:,:), allocatable, device :: fa_d
type(dim3) :: blcks, thrds

istat= cudaSetDeviceFlags(cudadevicemaphost)

istat = cudaHostAlloc(a,100*100*sizeof(1.0),cudaHostAllocMapped)

! can move the c_ptr to an f90 pointer
call c_f_pointer(a, fa, (/ 100, 100 /) )

! update the data on the host
do j = 1, 100
  do i = 1, 100
    fa(i,j)= real(i) + j*;100.0
  end do
end do

! get a device pointer to the same array
istat= cudaHostGetDevicePointer(a_d, a, 0)
```

(continues on next page)

(continued from previous page)

```
! can move the c_devptr to an device allocatable array
call c_f_pointer(a_d, fa_d, (/ 100, 100 /) )
!
blcks = dim3(10,10,1)
thrds = dim3(10,10,1)
!
call matrixinc <<<blcks, thrds>>>(fa_d, 100)

! need to synchronize
istat = cudaDeviceSynchronize()
!
do j = 1, 100
  do i = 1, 100
    if (fa(i,j) .ne. (real(i) + j*100.0 + 1.0)) print *,"failure",i,j
  end do
end do
!
istat = cudaFreeHost(a)
end
```

4.3. Cublas Module Example

This example demonstrates the use of the cublas module, the `cublasHandle` type, the three forms of cublas calls, and the use of mapped pinned memory, all within the framework of an multi-threaded OpenMP program.

Cublas Module

```
program tdot
! Compile with "nvfortran -mp tdot.cuf -cudalib=cublas -lblas
! Set OMP_NUM_THREADS environment variable to run with
! up to 2 threads, currently.
!
use cublas
use cudafor
use omp_lib
!
integer, parameter :: N = 10000
real*8 x(N), y(N), z
real*8, device, allocatable :: xd0(:), yd0(:)
real*8, device, allocatable :: xd1(:), yd1(:)
real*8, allocatable :: zh(:)
real*8, allocatable, device :: zd(:)
integer, allocatable :: istats(:), offs(:)
real*8 reslt(3)
type(C_DEVPTR) :: zdptr
type(cublasHandle) :: h

! Max at 2 threads for now
nthr = omp_get_max_threads()
if (nthr .gt. 2) nthr = 2
call omp_set_num_threads(nthr)
```

(continues on next page)

(continued from previous page)

```
! Run on host
call random_number(x)
call random_number(y)
z = ddot(N,x,1,y,1)
print *,"HostSerial",z
```

```
! Create a pinned memory spot
!$omp PARALLEL private(i,istat)
    i = omp_get_thread_num()
    istat = cudaSetDeviceFlags(cudaDeviceMapHost)
    istat = cudaSetDevice(i)
!$omp end parallel
allocate(zh(512),align=4096)
zh = 0.0d0
istat = cudaHostRegister(C_LOC(zh(1)), 4096, cudaHostRegisterMapped)
istat = cudaHostGetDevicePointer(zdptr, C_LOC(zh(1)), 0)
call c_f_pointer(zdptr, zd, 512 )
```

```
! CUDA data allocation, run on one card, blas interface
allocate(xd0(N),yd0(N))
xd0 = x
yd0 = y
z = ddot(N,xd0,1,yd0,1)
ii = 1
reslt(ii) = z
ii = ii + 1
deallocate(xd0)
deallocate(yd0)
```

```
! Break up the array into sections
nsec = N / nthr
allocate(istats(nthr),offs(nthr))
offs = (/ (i*nsec,i=0,nthr-1) /)

! Allocate and initialize the arrays
!$omp PARALLEL private(i,istat)
    i = omp_get_thread_num() + 1
    if (i .eq. 1) then
        allocate(xd0(nsec), yd0(nsec))
        xd0 = x(offs(i)+1:offs(i)+nsec)
        yd0 = y(offs(i)+1:offs(i)+nsec)
    else
        allocate(xd1(nsec), yd1(nsec))
        xd1 = x(offs(i)+1:offs(i)+nsec)
        yd1 = y(offs(i)+1:offs(i)+nsec)
    endif
!$omp end parallel
```

```
! Run the blas kernel using cublas name
!$omp PARALLEL private(i,istat,z)
    i = omp_get_thread_num() + 1
    if (i .eq. 1) then
        z = cublasDdot(nsec,xd0,1,yd0,1)
    else
        z = cublasDdot(nsec,xd1,1,yd1,1)
```

(continues on next page)

(continued from previous page)

```
    endif
    zh(i) = z
!$omp end parallel
```

```
z = zh(1) + zh(2)
reslt(ii) = z
ii = ii + 1

zh = 0.0d0
```

```
! Now write to our pinned area with the v2 blas
!$omp PARALLEL private(h,i,istat)
    i = omp_get_thread_num() + 1
    h = cublasGetHandle()
    istat = cublasSetPointerMode(h, CUBLAS_POINTER_MODE_DEVICE)
    if (i .eq. 1) then
        istats(i) = cublasDdot_v2(h, nsec, xd0, 1, yd0, 1, zd(1))
    else
        istats(i) = cublasDdot_v2(h, nsec, xd1, 1, yd1, 1, zd(2))
    endif
    istat = cublasSetPointerMode(h, CUBLAS_POINTER_MODE_HOST)
    istat = cudaDeviceSynchronize()
!$omp end parallel
```

```
z = zh(1) + zh(2)
reslt(ii) = z

print *,"Device, 3 ways:",reslt

! Deallocate the arrays
!$omp PARALLEL private(i)
    i = omp_get_thread_num() + 1
    if (i .eq. 1) then
        deallocate(xd0,yd0)
    else
        deallocate(xd1,yd1)
    endif
!$omp end parallel
deallocate(istats,offs)

end
```

4.4. CUDA Device Properties Example

This example demonstrates how to access the device properties from CUDA Fortran.

CUDA Device Properties

```
! An example of getting device properties in CUDA Fortran
! Build with
!   nvfortran cufinfo.cuf
!
```

(continues on next page)

(continued from previous page)

```
program cufinfo
use cudafor
integer istat, num, numdevices
type(cudadeviceprop) :: prop
istat = cudaGetDeviceCount(numdevices)
do num = 0, numdevices-1
  istat = cudaGetDeviceProperties(prop, num)
  call printDeviceProperties(prop, num)
end do
end
!
subroutine printDeviceProperties(prop, num)
use cudafor
type(cudadeviceprop) :: prop
integer num
ilen = verify(prop%name, ' ', .true.)
write (*,900) "Device Number: "       ,num
write (*,901) "Device Name: "         ,prop%name(1:ilen)
write (*,903) "Total Global Memory: ",real(prop%totalGlobalMem)/1e9," Gbytes"
write (*,902) "sharedMemPerBlock: "   ,prop%sharedMemPerBlock," bytes"
write (*,900) "regsPerBlock: "        ,prop%regsPerBlock
write (*,900) "warpSize: "            ,prop%warpSize
write (*,900) "maxThreadsPerBlock: "  ,prop%maxThreadsPerBlock
write (*,904) "maxThreadsDim: "       ,prop%maxThreadsDim
write (*,904) "maxGridSize: "         ,prop%maxGridSize
write (*,903) "ClockRate: "           ,real(prop%clockRate)/1e6," GHz"
write (*,902) "Total Const Memory: "  ,prop%totalConstMem," bytes"
write (*,905) "Compute Capability Revision: ",prop%major,prop%minor
write (*,902) "TextureAlignment: "    ,prop%textureAlignment," bytes"
write (*,906) "deviceOverlap: "       ,prop%deviceOverlap
write (*,900) "multiProcessorCount: ",prop%multiProcessorCount
write (*,906) "integrated: "          ,prop%integrated
write (*,906) "canMapHostMemory: "    ,prop%canMapHostMemory
write (*,906) "ECCEnabled: "          ,prop%ECCEnabled
write (*,906) "UnifiedAddressing: "   ,prop%unifiedAddressing
write (*,900) "L2 Cache Size: "       ,prop%l2CacheSize
write (*,900) "maxThreadsPerSMP: "    ,prop%maxThreadsPerMultiProcessor
900 format (a,i0)
901 format (a,a)
902 format (a,i0,a)
903 format (a,f5.3,a)
904 format (a,2(i0,1x,'x',1x),i0)
905 format (a,i0,'.',i0)
906 format (a,l0)
return
end
```

4.5. CUDA Asynchronous Memory Transfer Example

This example demonstrates how to perform asynchronous copies to and from the device using the CUDA API from CUDA Fortran.

CUDA Asynchronous Memory Transfer

```
! This code demonstrates strategies hiding data transfers via
! asynchronous data copies in multiple streams

module kernels_m
contains
  attributes(global) subroutine kernel(a, offset)
    implicit none
    real :: a(*)
    integer, value :: offset
    integer :: i
    real :: c, s, x
    i = offset + threadIdx%x + (blockIdx%x-1)*blockDim%x
    x = threadIdx%x + (blockIdx%x-1)*blockDim%x
    s = sin(x); c = cos(x)
    a(i) = a(i) + sqrt(s**2+c**2)
  end subroutine kernel
end module kernels_m

program testAsync
  use cudafor
  use kernels_m
  implicit none
  integer, parameter :: blockSize = 256, nStreams = 8
  integer, parameter :: n = 16*1024*blockSize*nStreams
  real, pinned, allocatable :: a(:)
  real, device :: a_d(n)
  integer(kind=cuda_Stream_Kind) :: stream(nStreams)
  type (cudaEvent) :: startEvent, stopEvent, dummyEvent
  real :: time
  integer :: i, istat, offset, streamSize = n/nStreams
  logical :: pinnedFlag
  type (cudaDeviceProp) :: prop

  istat = cudaGetDeviceProperties(prop, 0)
  write(*,"(' Device: ', a,/)") trim(prop%name)

  ! allocate pinned  host memory
  allocate(a(n), STAT=istat, PINNED=pinnedFlag)
  if (istat /= 0) then
     write(*,*) 'Allocation of a failed'
     stop
  else
     if (.not. pinnedFlag) write(*,*) 'Pinned allocation failed'
     end if

  ! create events and streams
  istat = cudaEventCreate(startEvent)
```

(continues on next page)

(continued from previous page)

```
  istat = cudaEventCreate(stopEvent)
  istat = cudaEventCreate(dummyEvent)
  do i = 1, nStreams
     istat = cudaStreamCreate(stream(i))
  enddo
```

```
! baseline case - sequential transfer and execute
a = 0
istat = cudaEventRecord(startEvent,0)

a_d = a
call kernel<<<n/blockSize, blockSize>>>(a_d, 0)
a = a_d
istat = cudaEventRecord(stopEvent, 0)
istat = cudaEventSynchronize(stopEvent)
istat = cudaEventElapsedTime(time, startEvent, stopEvent)
write(*,*) 'Time for sequential transfer and execute (ms): ', time
write(*,*) '  max error: ', maxval(abs(a-1.0))

! asynchronous version 1: loop over {copy, kernel, copy}
a = 0
istat = cudaEventRecord(startEvent,0)

do i = 1, nStreams
   offset = (i-1)*streamSize
   istat = cudaMemcpyAsync(a_d(offset+1),a(offset+1),streamSize,stream(i))
   call kernel<<<streamSize/blockSize, blockSize, &
                 0, stream(i)>>>(a_d,offset)
   istat = cudaMemcpyAsync(a(offset+1),a_d(offset+1),streamSize,stream(i))
enddo
istat = cudaEventRecord(stopEvent, 0)
istat = cudaEventSynchronize(stopEvent)
istat = cudaEventElapsedTime(time, startEvent, stopEvent)
write(*,*) 'Time for asynchronous V1 transfer and execute (ms): ', time
write(*,*) '  max error: ', maxval(abs(a-1.0))

! asynchronous version 2:
! loop over copy, loop over kernel, loop over copy
a = 0
istat = cudaEventRecord(startEvent,0)
do i = 1, nStreams
   offset = (i-1)*streamSize
   istat = cudaMemcpyAsync(a_d(offset+1),a(offset+1),streamSize,stream(i))
enddo
do i = 1, nStreams
   offset = (i-1)*streamSize
   call kernel<<<streamSize/blockSize, blockSize, &
                 0, stream(i)>>>(a_d,offset)
enddo
do i = 1, nStreams
   offset = (i-1)*streamSize
   istat = cudaMemcpyAsync(a(offset+1),a_d(offset+1),streamSize,stream(i))
enddo
istat = cudaEventRecord(stopEvent, 0)
istat = cudaEventSynchronize(stopEvent)
istat = cudaEventElapsedTime(time, startEvent, stopEvent)
```

(continues on next page)

(continued from previous page)

```
write(*,*) 'Time for asynchronous V2 transfer and execute (ms): ', time
write(*,*) '  max error: ', maxval(abs(a-1.0))

! cleanup
istat = cudaEventDestroy(startEvent)
istat = cudaEventDestroy(stopEvent)
istat = cudaEventDestroy(dummyEvent)

do i = 1, nStreams
   istat = cudaStreamDestroy(stream(i))
enddo
deallocate(a)

end program testAsync
```

4.6. Managed Memory Example

This example demonstrates the use of CUDA managed memory in an OpenMP program. In the main program, one stream is created for each OpenMP thread. A call to cudaforSetDefaultStream is made to set that as the default stream for all subsequent high-level language constructs. The default stream is used explicitly in the launch configuration of the CUF kernel, and also as the thread's input argument for synchronization. Once the cudaStreamSynchronize has occurred, this thread can safely access the managed data on the host, in this case in the any() function, even while other threads may be in the middle of their kernel launch.

Managed Memory and OpenMP in CUDA Fortran

```
program ompcuf
use cudafor
use omp_lib
integer(kind=cuda_stream_kind) :: mystream

!$omp parallel private(istat,mystream)
istat = cudaStreamCreate(mystream)
istat = cudaforSetDefaultstream(mystream)
call ompworker()
!$omp end parallel
end

subroutine ompworker()
use cudafor
use omp_lib
real, managed :: a(10000)
j = omp_get_thread_num()
a = real(j)

!$cuf kernel do <<< *, *, stream=cudaforGetDefaultStream() >>>
do i = 1, 10000
  a(i) = a(i) + 1.0
end do
istat = cudaStreamSynchronize(cudaforGetDefaultStream())

if (any(a.ne.real(j+1))) then
```

(continues on next page)

(continued from previous page)

```
  print *,"Found error on ",j
else
  print *,"Looks good on ",j
endif
end
```

4.7. WMMA Tensor Core Example

This example demonstrates the use of NVIDIA Volta tensor cores to perform real(2) matrix multiply. The result is a real(4) matrix. This example utilizes the definitions in `cuf_macros.CUF`, a file which is shipped in the examples directory of the NVIDIA packages. The actual derived types currently used in Fortran tensor core programming may change at a later date, but these macros will always be supported. The program shows the use of the Fortran real(2) data type, both in host and device code. Further examples, highlighting overloaded device functions which take the `WMMASubMatrix` types, and which use a vector of real(2) data for improved performance, can be found in the examples directory of NVIDIA packages.

Tensor Core Programming in CUDA Fortran

```
#include "cuf_macros.CUF"

module params
  integer, parameter :: m = 16
  integer, parameter :: n = 16
  integer, parameter :: k = 16
end module

module mod1
use params  ! Define matrix m, n, k
contains
  attributes(global) subroutine test1(a,b,c)
  use wmma
  real(2), device :: a(m,k)
  real(2), device :: b(k,n)
  real(4), device :: c(m,n)
  WMMASubMatrix(WMMAMatrixA, 16, 16, 16, Real, WMMAColMajor) :: sa
  WMMASubMatrix(WMMAMatrixB, 16, 16, 16, Real, WMMAColMajor) :: sb
  WMMASubMatrix(WMMAMatrixC, 16, 16, 16, Real, WMMAKind4)    :: sc
  sc = 0.0
  call wmmaLoadMatrix(sa, a(1,1), m)
  call wmmaLoadMatrix(sb, b(1,1), k)
  call wmmaMatmul(sc, sa, sb)
  call wmmaStoreMatrix(c(1,1), sc, m)
  end subroutine
end module

program main
use cudafor
use mod1
real(2), managed :: a(m,k)
real(2), managed :: b(k,n)
real(4), managed :: c(m,n)
```

(continues on next page)

(continued from previous page)

```
a = real(1.0,kind=2)
b = 2.0_2
c = 0.0
call test1 <<<1,32>>> (a,b,c)
istat = cudaDeviceSynchronize()
print *,all(c.eq.2*k)
end program
```

4.8. OpenACC Interoperability Example

This example demonstrates two ways that CUDA Fortran and OpenACC can be used together in the same program, both in sharing data, and in control flow. At the lowest level, we have slightly modified the BLAS daxpy subroutine by inserting it in a module, making it an OpenACC vector routine, and adding OpenACC vector loop directives. The second file contains pure CUDA Fortran, a global subroutine which calls daxpy with the same arguments for each thread block. At the highest level, we have a Fortran main program which uses OpenACC for data management, but calls both a CUDA Fortran global function and overloaded CUDA Fortran reductions via the host_data directive. This directive instructs the compiler to pass the corresponding device pointers, which are managed implicitly by the OpenACC runtime, for the x and y arguments.

Mixing CUDA Fortran and OpenACC

```
      module daxpy_mod
      contains
      subroutine daxpy(n,da,dx,incx,dy,incy)
!$acc routine vector nohost
!
!     constant times a vector plus a vector.
!     uses unrolled loops for increments equal to one.
!     jack dongarra, linpack, 3/11/78.
!     modified 12/3/93, array(1) declarations changed to array(*)
!
      integer, value :: n, incx, incy
      double precision, value ::  da
      double precision dx(*),dy(*)
      integer i,ix,iy
!
      if(n.le.0)return
      if (da .eq. 0.0d0) return
      if(incx.eq.1.and.incy.eq.1) then
!
!        code for both increments equal to 1
!
!$acc loop vector
        do i = 1, n
          dy(i) = dy(i) + da*dx(i)
        end do
      else
!
!        code for unequal increments or equal increments
!          not equal to 1
!
```

(continues on next page)

(continued from previous page)

```
!$acc loop vector
        do i = 1, n
          if(incx.lt.0) then
            ix = 1 + (-n+i) * incx
          else
            ix = 1 + (i-1) * incx
          end if
          if(incy.lt.0) then
            iy = 1 + (-n+i) * incy
          else
            iy = 1 + (i-1) * incy
          end if
          dy(iy) = dy(iy) + da*dx(ix)
        end do
      end if
      return
      end
      end module daxpy_mod
```

```
module mdaxpy
use daxpy_mod
contains
  attributes(global) subroutine mdaxpy(x,y,n)
  integer, value :: n
  real(8), device :: x(n), y(n,n)
  real(8) :: a
  a = 0.5d0
  j = blockIdx%x
  call daxpy(n, a, x, 1, y(1,j), 1)
  return
  end subroutine
end module
```

Care must be taken in the CUDA code, where the programming model allows much flexibility in how the threads are applied, to follow what OpenACC expects when calling into an OpenACC routine. Calling an OpenACC vector routine from every thread in a thread block, passing the same parameters, is usually safe. Calling OpenACC sequential routines from a CUDA thread is also safe. This is a generally a new feature and has not yet been thoroughly tested.

```
program tdaxpy
! Compile with "nvfortran -cuda daxpy.F mdaxpy.CUF tdaxpy.F90"
use cudafor
use mdaxpy
integer, parameter :: n = 100
real(8) :: x(n), y(n,n)
x = 2.0d0
y = 3.0d0
!$acc data copyin(x), copy(y)
!$acc host_data use_device(x,y)
call mdaxpy <<<n, n>>> (x, y, n)
print *,sum(y),maxval(y).eq.minval(y)
!$acc end host_data
!$acc end data
end program
```

There are many examples of calling CUDA code from within OpenACC compute regions. The examples directory in the NVIDIA package has several, from Fortran, C, and C++. There are also many examples

of using the OpenACC host_data directive. More information on that directive, and other directives, can be found in the OpenACC Specification.

Notices

Notice

Trademarks

Copyright