CUDA Fortran
Programming Guide and Reference

Version 2016

PGI Compilers and Tools

TABLE OF CONTENTS

LIST OF TABLES

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 PGI tools are available on a variety of operating systems for the X86, AMD64, and Intel 64 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 PGI compilers and tools are supported on both 32-bit and 64-bit variants of the Linux, Apple OS X, and Windows operating systems on a variety of x86-compatible processors. There are a wide variety of releases and distributions of each of these types of operating systems.

> Support for 32-bit development is deprecated in PGI 2016 and will no longer be available as of the PGI 2017 release. PGI 2017 will only be available for 64-bit operating systems and will not include the ability to compile 32-bit applications for execution on either 32- or 64-bit operating systems.

Terminology

If there are terms in this guide with which you are unfamiliar, PGI provides a glossary of terms which you can access at www.pgroup.com/support/definitions.htm

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 Guide, NVIDIA, Version 3.1.1, 7/21/2010. Available online at http://www.nvidia.com/cuda.
- NVIDIA CUDA Compute Unified Device Architecture Reference Manual, NVIDIA, Version 3.1, June 2010. Available online at http://www.nvidia.com/cuda.
- PGI Compiler User"s Guide, The Portland Group, Release 2016. Available online at www.pgroup.com/doc/pgiug.pdf.

Chapter 1.
INTRODUCTION

Welcome to Release 2016 of PGI 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. PGI 2016 includes support for CUDA Fortran on Linux, Apple OS X and Windows. CUDA Fortran is an analog to NVIDIA's CUDA C compiler. Compared to the PGI Accelerator and 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 CUDA 5.0 and above dynamic parallelism features.
- ▶ Relocatable device code: Creating and linking device libraries such as the `cublas`; and calling functions defined in other modules and files.
- ▶ Interfacing to CUDA C.

Chapter 2.
PROGRAMMING GUIDE

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

2.1. 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

<pre>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</pre>
	<pre>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</pre>

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.

2.2. 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.

2.3. 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>>>( ... )
```

2.4. 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 Appendix J. 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.

2.5. 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*.

2.5.1. 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.

2.5.2. 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.

2.5.3. 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.

2.5.4. 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.
▶ A device subprogram must not have optional arguments.
▶ 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.

2.6. 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.

2.6.1. 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 an 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.

2.6.2. 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.

2.6.3. 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. When declared in a module, 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. Device constant variables may not be allocatable, and have a lifetime of the entire application.

2.6.4. 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.

2.6.5. 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.

2.6.6. Attributes(texture)

A variable with the `texture` attribute is called a *texture variable*. A texture variable must be an F90 pointer, and can be of type real or integer. Texture variables may be accessed only in device subprograms, and can only be read, not written. The advantage of using texture variables is that the accesses to texture data goes through a separate cache on the device, which may result in improved performance for many codes. Texture variables are bound to underlying device arrays in host code using F90 pointer assignments.

2.7. Datatypes in Device Subprograms

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

Table 1   Intrinsic Datatypes

Type	Type Kind
`integer`	1,2,4,8
`logical`	1,2,4,8
`real`	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.

2.8. 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.

2.9. 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.

2.10. Asynchronous Concurrent Execution

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

2.10.1. 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.

2.10.2. 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.

2.11. 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.

2.11.1. 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
    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*.

2.11.2. 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*.
- ▸ 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.
- ▸ CUDA Fortran intrinsic functions are allowed, if they are allowed in device code, but the device-specific intrinsics such as syncthreads, atomic functions, etc. are not.
- ▸ 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 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.

2.11.3. 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.

2.12. 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.

2.12.1. 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
```

2.12.2. 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 `-Mcuda=rdc` flag:

```
% pgf90 -Mcuda=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 -Mcuda=rdc flag and then can be linked with the previous object file:

```
% pgf90 -Mcuda=rdc tfill.cuf ffill.o
```

2.12.3. Declaring Device Pointer and Target Arrays

Recently, PGI 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
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
```

2.12.4. Declaring Textures

In 2012, PGI added support for CUDA texture memory fetches 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)
    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)
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
```

2.13. Building a CUDA Fortran Program

CUDA Fortran is supported by the PGI Fortran compilers 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 `-Mcuda` command line option.

To enable CUDA 7.5 or 8.0 features, use `-Mcuda=cuda7.5` or `-Mcuda=cuda8.0`. Starting in PGI 14.1, `-Mcuda=rdc` is on by default to generate relocatable device code. You can override this option by specifying `-Mcuda=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.

Starting in PGI 14.1, a separate LLVM-based back-end is also supported. This back-end is utilized when `--g` is specified as a compiler option as it enables the generation of debug information. To always use the LLVM-based back-end, use the `-Mcuda=llvm` command line option. To enable debugging on the host without using the LLVM-based back-end, specify `-Mcuda=nodebug` on the command line.

2.14. Emulation Mode

PGI Fortran compilers support an emulation mode for program development on workstations or systems without a CUDA-enabled GPU and for debugging. To build a program using emulation mode, compile and link with the `-Mcuda=emu` command line option. In emulation mode, the device code is compiled for and runs on the host, allowing the programmer to use a host debugger or full i/o capabilities.

It is important to note that the emulation is far from exact. In particular, emulation mode may execute a single thread block at a time. This will not expose certain errors, such as memory races. In emulation mode, the host floating point units and intrinsics are used, which may produce slightly different answers than the device units and intrinsics.

Chapter 3.
REFERENCE

This section is the CUDA Fortran Language Reference.

3.1. 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.

3.1.1. 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.

3.1.2. Global Subroutines

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

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

Functions may not have the global attribute. A subroutine with the 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.

3.1.3. 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 may not be recursive, pure, or elemental, so no other subroutine or function prefixes are allowed, except for the function return datatype. A subroutine or function with the device or kernel attribute is called a *device subprogram*. A device subprogram is compiled for execution on the device. A subroutine or function with the device attribute must appear within a Fortran module, and may only be called from device subprograms in the same module.

3.1.4. Restrictions on Device Subprograms

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

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

For more information, refer to Device Code.

3.2. 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. Additionally, the texture attribute can be added to read-only data declared in modules which enables reading the data through the texture cache on the device.

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.

3.2.1. 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.

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.

3.2.2. 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 arrays can be automatic or allocatable. 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.

- ▸ 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`cudaSetStreamDefault`.
- ▸ All subsequently allocated managed variables can also be associated with a given stream by calling`cudaSetStreamDefault`.
- ▸ Accessing managed data on the host while a running kernel is accessing managed data within the same scope on the device will result in a segmentation fault.

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.

> 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 Appendix J, section J.2.5.1.

3.2.3. 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.

3.2.4. 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

3.2.5. 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.

3.2.6. Texture data

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 is used in both the host and device code:
   ```
   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 use or 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)
```

3.2.7. 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.

3.3. 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.

3.3.1. 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
```

3.3.2. 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.

3.3.3. 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
. . .
```

3.3.4. 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
```

3.3.5. 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.

3.4. Data transfer between host and device memory

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

3.4.1. 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 Appendix J. of the `CUDA_C_Programming_Guide`.

3.4.2. 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.

3.4.3. 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.

3.5. 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, or of `type(dim3)`. If it is `type(dim3)`, the value of `grid%z` must be one. The product `grid%x*grid%y` gives the number of thread blocks to launch. If grid is an integer, it is converted to `dim3(grid,1,1)`. bl

▶ `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 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. Beginning with PGI 15.4, the stream constant value `cudaStreamPerThread` may be specified. This will use a unique stream for each CPU thread. Also, starting in PGI 15.4, the default stream value, normally zero, can be changed by a call to `cudaforSetDefaultStream()`.

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.

3.6. Device code

3.6.1. 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   Device Code Intrinsic Datatypes

Type	Type Kind
`integer`	1,2,4(default),8
`logical`	1,2,4(default),8
`real`	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
```

3.6.2. 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%z` is always one. The value of `griddim` is the same for all threads in the same grid; the value of `griddim%z` is always one; the value of `griddim%y` 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.

3.6.3. Fortran Intrinsics

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

Table 3   Fortran Numeric and Logical Intrinsics

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

Table 4   Fortran Mathematical Intrinsics

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

Device versions of the new F2008 intrinsics (bessel functions, gamma functions, erf, erfc, acosh, asinh, atanh, and hypot) are not currently supported in the 32-bit compilers. Support for these functions will be included in a future PGI release

Table 5 Fortran Numeric Inquiry Intrinsics

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

Table 6 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 Fortran Reduction Intrinsics

Name	Argument Datatypes		Name	Argument Datatypes
`all`	logical		`minloc`	integer, real
`any`	logical		`minval`	integer, real
`count`	logical		`product`	integer, real, complex
`maxloc`	integer, real		`sum`	integer, real, complex
`maxval`	integer, real			

Table 8 Fortran Random Number Intrinsics

Name	Argument Datatypes
`random_number`	real
`random_seed`	integer

3.6.4. New Intrinsic Functions

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

Synchronization Functions

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

syncthreads	threadfence
syncthreads_count	threadfence_block
syncthreads_and	threadfence_system
syncthread_or	

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:

```
void 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.

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

```
void 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

```
void 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

```
void 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.

3.6.5. Warp-Vote Operations

Warp-vote operations are only supported by devices with compute capability 1.2 and higher. Each of these functions has a single argument.

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.
```

3.6.6. 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.

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

These functions are:

Table 9   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).

> The return value for each of these functions is the first argument, mem.

These functions are:

Table 10   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).

> The return value for this function is the first argument, mem.

The function is:

Table 11   Compare and Swap Atomic Function

Function	Additional Atomic Update

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

3.6.7. Fortran I/O

The PGI Fortran accelerator runtime, shared between CUDA Fortran and OpenACC, buffers up the output and prints an entire line's worth in one operation. Supported data types are integer, character, logical, real and complex.

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.

Print and write statements in device code are also supported when used with the LLVM code generator `-ta=llvm`, or `-Mcuda=llvm`, and in combination with the `-mp` compiler option.

3.6.7.1. PRINT Example

By adding the compiler option `-Mcuda=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
```

3.6.8. Shuffle Functions

PGI 14.1 enables CUDA Fortran device code to 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
```

```
   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
```

```
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
```

```
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
```

```
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
```

3.6.9. Restrictions

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

- ▸ Optional arguments are not allowed.
- ▸ 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.

3.7. Host code

Host subprograms may use intrinsic functions, such as the new `sizeof` intrinsic function.

3.7.1. 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
```

3.7.2. Overloaded Reduction Intrinsics

Beginning in PGI 15.1, the `sum`, `maxval`, and `minval` host intrinsics are overloaded to accept device or managed arrays when the cudafor module is used. The tuned kernels written to implement these operations use the shuffle functions, so compute capability 3.0 or higher is required to use these reductions. 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.

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,:))
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
```

By default, these reductions will run on a nonzero stream. If a unique per-thread default stream was set via a call to `cudaforSetDefaultStream`, the reduction initialization will pick that up. Users can further control which stream reductions run on, or force stream 0, by using the `cudaforReductionSetStream()` call. The current reduction stream can be queried using the `cudaforReductionGetStream()` call.

SUM() on complex managed or device data is not currently supported. This support will be added in a future PGI release

3.8. Fortran Modules

CUDA Fortran Modules are available to help programmers access features of the CUDA runtime environment, which might otherwise not be accessible from Fortran without significant effort from the programmer. These modules might be either device modules or host modules.

3.8.1. Device Modules

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

To access this module, 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 PGI Accelerator compute regions in Fortran as well as in C. Further, the PGI compilers

come with implementations of these routines for host code, though these implementations are not specifically optimized for the host.

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

Table 12   CUDA Built-in Routines

__brev	__brevll	clock	clock64
__clz	__clzll	__cosf	__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	__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	__sinf	__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

PGI 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 13   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
copysign,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		nearbyint,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		remainder,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

On NVIDIA GPUs which support CUDA Compute Capability 3.5 and above, PGI provides a device module which provides interfaces to the device CUBLAS library.

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

```
use cublas_device
```

Here is a simple example of using the CUBLAS device library:

```
module mblasd
contains
```

```
  attributes(global) subroutine testblas( a, b, n )
  use cublas_device
  real, device :: a(*), b(*)
  type(cublasHandle) :: h
  integer, value :: n
  j = cublasCreate_v2(h)
  i = threadIdx%x
  if (i.eq.1) then
    j = cublasSswap_v2(h,n,a,1,b,1)
  end if
  j = cublasDestroy_v2(h)
  return
  end subroutine
end module mblasd

! Compile with "pgfortran -Mcuda=cc35 cublasdev.cuf -lcublas_device"
program cublasdev
use mblasd
integer, parameter :: n = 1000
real, managed :: a(n), b(n)
a = 1.0; b = 2.0
call testblas<<<1,1>>> (a,b,n)
istat = cudaDeviceSynchronize()
print *,all(a.eq.2*b)
end
```

3.8.2. Host Modules

PGI provides a module which defines interfaces to the CUBLAS Library from PGI 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 3.2 helper functions defined in the cublas module:

```
integer function cublasInit()
integer function cublasShutdown()
integer function cublasGetError()
integer function cublasAlloc(n, elemsize, devptr)
integer function cublasFree(devptr)
integer function cublasSetVector(n, elemsize, x, incx, y, incy)
integer function cublasGetVector(n, elemsize, x, incx, y, incy)
integer function cublasSetMatrix(rows, cols, elemsize, a, lda, b, ldb)
integer function cublasGetMatrix(rows, cols, elemsize, a, lda, b, ldb)
integer function cublasSetKernelStream(stream)
integer function cublasSetVectorAsync(n, elemsize, x, incx, y, incy, stream)
integer function cublasGetVectorAsync(n, elemsize, x, incx, y, incy, stream)
integer function cublasSetMatrixAsync(rows, cols, elemsize, a, lda, b, ldb,
 stream)
integer function cublasGetMatrixAsync(rows, cols, elemsize, a, lda, b, ldb,
 stream)
```

Additional 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.

PGI provides another module which defines interfaces to the CUFFT Library from PGI 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)
```

Beginning in PGI 15.1, the distribution also contains a module which defines interfaces to the CUSPARSE Library from PGI CUDA Fortran. The CUSPARSE library has had fairly substantial changes made to it over the last several CUDA releases, so the source for the module is also included in our package, and a different version of the pre-compiled module is put into a version-specific location which is searched according to the CUDA version specified on the compilation command line. 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 "pgfortran testLevel3.cuf -lcusparse"
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
  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 4.
RUNTIME APIS

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

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).

4.1. 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.

4.2. Device Management

Use the functions in this section for device management.

4.2.1. 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.

4.2.2. 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.

4.2.3. 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.

4.2.4. 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`.

4.2.5. 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.

4.2.6. cudaDeviceReset

```
integer function cudaDeviceReset()
```

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

4.2.7. 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`.

4.2.8. 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.

4.2.9. 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.

4.2.10. cudaDeviceSynchronize

```
integer function cudaDeviceSynchronize()
```

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

cudaDeviceSynchronize is available in device code starting in CUDA 5.0.

4.2.11. 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.

4.2.12. 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.

4.2.13. 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.

4.2.14. cudaSetDevice

```
integer function cudaSetDevice( devnum )
```

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

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

4.2.15. cudaSetDeviceFlags

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

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

4.2.16. 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.

4.3. 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.

4.3.1. 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.

4.3.2. 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.

> 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.

4.4. Error Handling

Use the functions in this section for error handling.

4.4.1. cudaGetErrorString

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

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

4.4.2. cudaGetLastError

```
integer function cudaGetLastError()
```

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

4.4.3. 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.

4.5. Stream Management

Use the functions in this section for stream management.

4.5.1. 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).

cudaGetStreamDefault was available starting in CUDA 6.0. In PGI 15.4, it was renamed to cudaforGetDefaultStream since this function is not available in CUDA C/C++.

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

4.5.2. 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. Starting in PGI 15.4, 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.

`cudaSetStreamDefault` was available starting in CUDA 6.0. In PGI 15.4, it was renamed to `cudaforSetDefaultStream` since this function is not available in CUDA C/C++.

4.5.3. 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.

4.5.4. 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.

4.5.5. 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.

4.5.6. 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.

4.5.7. 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.

4.5.8. 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.

4.5.9. 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.

4.5.10. 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.

4.5.11. 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.

4.6. Event Management

Use the functions in this section to manage events.

4.6.1. cudaEventCreate

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

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

4.6.2. 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.

4.6.3. 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.

4.6.4. 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.

4.6.5. 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.

4.6.6. 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.

4.6.7. 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.

4.7. 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.

4.7.1. cudaFuncGetAttributes

```
integer function cudaFuncGetAttributes( attr, func )
    type(cudaFuncAttributes), intent(out) :: attr
    character*(*) :: 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.

4.7.2. 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`.

4.7.3. 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`

4.7.4. 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.

4.7.5. 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.

4.8. 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.

4.8.1. 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.

4.8.2. cudaFreeArray

```
integer function cudaFreeArray(carray)
 type(cudaArrayPtr) :: carray
```

`cudaFreeArray` frees an array that was allocated on the device.

4.8.3. cudaFreeHost

```
integer function cudaFreeHost(hostptr)
    type(C_PTR) :: hostptr
```

`cudaFreeHost` deallocates pinned memory on the host allocated with `cudaMalloHost`.

4.8.4. 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
```

4.8.5. 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.

4.8.6. 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.

4.8.7. 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.

4.8.8. cudaHostGetFlags

```
integer function cudaHostGetFlags(flags, hostptr)
    integer :: flags
    type(C_PTR) :: hostptr
```

cudaHostGetFlags returns the flags associated with a host pointer.

4.8.9. 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.

4.8.10. 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`.

4.8.11. 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.

4.8.12. 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.

4.8.13. 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.

4.8.14. 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`. `devptr` may also be of TYPE(C_DEVPTR), in which case the integer values are expressed in bytes.

4.8.15. 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.

4.8.16. cudaMalloc3DArray

```
integer function cudaMalloc3DArray(carray, cdesc, cext)
    type(cudaArrayPtr) :: carray
    type(cudaChannelFormatDesc) :: cdesc
    type(cudaExtent) :: cext
```

`cudaMalloc3DArray` allocates array data on the device.

4.8.17. 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.

4.8.18. 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.

4.8.19. 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.

4.8.20. 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.

4.8.21. 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.

4.8.22. 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.

4.8.23. 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.

4.8.24. 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.

4.8.25. 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.

4.8.26. 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.

4.8.27. 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.

4.8.28. 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. `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.

4.8.29. 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.

4.8.30. 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. 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.

4.8.31. 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.

4.8.32. 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.

4.8.33. 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.

4.8.34. 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.

4.8.35. 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.

4.8.36. 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.

4.8.37. 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.

4.8.38. 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.

4.9. Unified Addressing and Peer Device Memory Access

Use the functions in this section for managing multiple devices from the same process and threads.

4.9.1. 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.

4.9.2. 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.

4.9.3. 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.

4.9.4. cudaPointerGetAttributes

```
integer function cudaPointerGetAttributes( attr, ptr )
    type(cudaPointerAttributes), intent(out) :: ptr
```

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.

4.10. Version Management

Use the functions in this section for version management.

4.10.1. 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.

4.10.2. 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.

Chapter 5.
EXAMPLES

This section contains examples with source code.

5.1. 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.

5.1.1. 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
  ! Get the thread indices
```

```fortran
  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
```

```fortran
! 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
```

```fortran
  ! 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) )
```

```fortran
  ! Copy A and B to the device
  Adev = A(1:N,1:M)
  Bdev(:,:) = B(1:M,1:L)
```

```fortran
  ! Create the grid and block dimensions
  dimGrid = dim3( N/16, L/16, 1 )
  dimBlock = dim3( 16, 16, 1 )
  call mmul_kernel<<<dimGrid,dimBlock>>>( Adev, Bdev, Cdev, N, M, L)
```

```fortran
  ! 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
```

5.1.2. 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.

5.2. 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)

! 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
```

5.3. 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 "pgfortran -mp tdot.cuf -lcublas -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)
! 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
!$omp PARALLEL private(i,istat,z)
    i = omp_get_thread_num() + 1
    if (i .eq. 1) then
        z = cublasDdot(nsec,xd0,1,yd0,1)
    else
        z = cublasDdot(nsec,xd1,1,yd1,1)
    endif
    zh(i) = z
!$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
```

5.4. CUDA Device Properties Example

This example demonstrates how to access the device properties from CUDA Fortran.

CUDA Device Properties

```fortran
! An example of getting device properties in CUDA Fortran
! Build with
!   pgfortran cufinfo.cuf
!
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
```

5.5. 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)
  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)
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
```

5.6. 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
  print *,"Found error on ",j
else
  print *,"Looks good on ",j
endif
end
```

Chapter 6.
CONTACT INFORMATION

You can contact PGI at:

> 20400 NW Amberwood Drive Suite 100
> Beaverton, OR 97006

Or electronically using any of the following means:

> Fax: +1-503-682-2637
> Sales: sales@pgroup.com
> Support: trs@pgroup.com
> WWW: http://www.pgroup.com

The PGI User Forum is monitored by members of the PGI engineering and support teams as well as other PGI customers. The forum newsgroups may contain answers to commonly asked questions. Log in to the PGI website to access the forum:

http://www.pgroup.com/userforum/index.php

Many questions and problems can be resolved by following instructions and the information available at our frequently asked questions (FAQ) site:

http://www.pgroup.com/support/faq.htm

All technical support is by email or submissions using an online form at:

http://www.pgroup.com/support

Phone support is not currently available.

PGI documentation is available at http://www.pgroup.com/resources/docs.htm.

Notice

ALL NVIDIA DESIGN SPECIFICATIONS, REFERENCE BOARDS, FILES, DRAWINGS, DIAGNOSTICS, LISTS, AND OTHER DOCUMENTS (TOGETHER AND SEPARATELY, "MATERIALS") ARE BEING PROVIDED "AS IS." NVIDIA MAKES NO WARRANTIES, EXPRESSED, IMPLIED, STATUTORY, OR OTHERWISE WITH RESPECT TO THE MATERIALS, AND EXPRESSLY DISCLAIMS ALL IMPLIED WARRANTIES OF NONINFRINGEMENT, MERCHANTABILITY, AND FITNESS FOR A PARTICULAR PURPOSE.

Information furnished is believed to be accurate and reliable. However, NVIDIA Corporation assumes no responsibility for the consequences of use of such information or for any infringement of patents or other rights of third parties that may result from its use. No license is granted by implication of otherwise under any patent rights of NVIDIA Corporation. Specifications mentioned in this publication are subject to change without notice. This publication supersedes and replaces all other information previously supplied. NVIDIA Corporation products are not authorized as critical components in life support devices or systems without express written approval of NVIDIA Corporation.

Trademarks

PGI Workstation, PGI Server, PGI Accelerator, PGF95, PGF90, PGFORTRAN, and PGI Unified Binary are trademarks; and PGI, PGHPF, PGF77, PGCC, PGC++, PGI Visual Fortran, PVF, PGI CDK, Cluster Development Kit, PGPROF, PGDBG, and The Portland Group are registered trademarks of NVIDIA Corporation in the U.S. and other countries. Other company and product names may be trademarks of the respective companies with which they are associated.

Copyright

PGI®