The Portland Group®

CUDA Fortran Programming Guide and Reference

Published: v1.1 February 2010

PGI®

Contents

1 Introduction

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.

This document describes CUDA Fortran, a small set of extensions to Fortran that supports and is built upon the CUDA computing architecture. The extensions described here allow the following operations in a Fortran program:

- declaring variables that will be 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

1.1 Structure of This Document

This document has five chapters:

- Chapter 1 is a general introduction
- Chapter 2 serves as a programming guide for CUDA Fortran
- Chapter 3 is the CUDA Fortran language reference
- Chapter 4 describes the interface between CUDA Fortran and the CUDA Runtime API
- Chapter 5 walks through the code of a simple example

Details about the capabilities and hardware in NVIDIA GPUs can be found in the appropriate NVIDIA documentation.

.

1.2 References

- ISO/IEC 1539-1:1997, *Information Technology – Programming Languages – Fortran*, Geneva, 1997 (Fortran 95).
- *NVIDIA CUDA™ Programming Guide*, NVIDIA, Version 2.1, 12/8/2008. Available online at http://www.nvidia.com/cuda.
- *NVIDIA CUDA Compute Unified Device Architecture Reference Manual*, NVIDIA, Version 2.0, June 2008. Available at http://www.nvidia.com/cuda.
- *PGI User's Guide*, The Portland Group, Release 9.0, June 2009. Available online at http://www.pgroup.com/doc/pgiug.pdf.

2 Programming Guide

This chapter introduces the CUDA programming model through examples written in CUDA Fortran. A reference for CUDA Fortran can be found in Chapter 3.

2.1 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. 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.

2.2 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. A unique thread ID is assigned to each thread, computed from the thread index. For a one-dimensional thread block, the thread index is equal to the thread ID. For a two-

dimensional thread block of size (D_x,D_y), the thread ID is equal to $(x+D_x(y-1))$. For a three-dimensional thread block of size (D_x,D_y,D_z), the thread ID is $(x+D_x(y-1+D_y(z-1))$.

Threads in the same thread block may cooperate using *shared memory*, and by synchronizing at a barrier using the `SYNCTHREADS()` intrinsic. Each thread in the block will wait 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 512.

A kernel may be invoked with many thread blocks, each with the same thread block size. The thread blocks are organized into a one- or two-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.

2.3 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.

2.4 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*.

2.4.1 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.

2.4.2 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 from the host using a kernel call containing the chevron syntax and runtime mapping parameters.

2.4.3 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. A single subroutine or function may have both `device` and `host` attributes; in this case, the subprogram is compiled once for the device and once for the host. Such a subprogram is both a device and a host subprogram.

2.4.4 Restrictions

A device subprogram must not be recursive.

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. Dummy arguments in a device subprogram must not be assumed-shape arrays, and must not have the pointer attribute.

Calls to a kernel subroutine must specify the execution configuration, as in section 2.7. Such a call is asynchronous, that is, the host routine making the call will continue execute before the device has completed its execution of the kernel subroutine.

Arguments to a kernel subroutine are currently limited to a total size of 256 bytes.

Device subprograms may not be contained in a host subroutine or function, and may not contain any subroutines or functions.

2.5 Variable Qualifiers

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

2.5.1 Attributes(device)

A variable with the `device` attribute is called a *device variable*, and will be allocated in the device global memory. If declared in a module, the variable may be accessed by any device subprogram in that module, and by any host subprogram in the module or 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, in a host subprogram, 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.

2.5.2 Attributes(constant)

A variable with the `constant` attributes is called a *device constant variable*. Device constant variables are allocated in the device constant memory space. If declared in a module, the variable may be accessed by any device subprogram in that module, and by any host

subprogram in the module or 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.

2.5.3 Attributes(shared)

A variable with the `shared` attributed 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.

2.5.4 Attributes(pinned)

A variable with the `pinned` attributes 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 page-locked 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.

2.6 Datatypes in Device Subprograms

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

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.

2.7 Predefined Variables in Device Subprograms

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

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

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 will have the value one.

The variable `blockdim` contains the dimensions of the thread block; `blockdim` will have 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` will have the value one. The value of `blockidx%z` is always one.

The variable `griddim` contains the dimensions of the grid; the value of `griddim%z` is always one.

These four variables are not accessible in host subprograms.

An additional builtin read-only variable is `warpsize`, declared to be type integer. Threads are executed in groups of 32, called *warps*; `warpsize` contains the number of threads in a warp.

2.8 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 will 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

```
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 one-, two-, or three-dimensional thread blocks. If `grid` is `type(dim3)`, the value of `grid%z` must be one, and `block%x` and `block%y` must be equal to or greater than one. 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 512.

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 will be used for the assumed-size shared variables in the thread block; see Section 3.2.3. If 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.

2.9 Asynchronous concurrent execution

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

2.9.1 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. The host program can synchronize and wait for all previously launched or queued kernels by calling the `cudaThreadSynchronize` runtime routine. 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.

2.9.2 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 will begin until the stream zero operation is complete.

2.10 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.

2.11 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.

It’s 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.

3 Reference

This chapter is the CUDA Fortran Language Reference.

3.1 New Subroutine and Function Attributes

CUDA Fortran adds new attributes to subroutines and functions. This chapter 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 below.

3.1.1 Host Subroutines and Functions

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

```
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

```
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

```
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 Device and Host Subroutines and Functions

A subroutine or function may have both the device and host attributes, if explicitly specified on the Subroutine or Function statement:

```
attributes(device,host) subroutine sub(…)
attributes(device,host) datatype function func(…)
datatype attributes(host,device) function func(…)
```

The device and host attributes keywords may appear in either order. A subprogram that has both device and host attributes must appear within a Fortran module. It will be compiled both for execution on the device and for execution on the host. It may be called from device subprograms in the same Fortran module, in which case the device code will be called. It may also be called from any host subprogram in the same module, or any subprogram that uses the module or is contained in a subprogram that uses the module. Subprograms with both device and host attributes must satisfy all the restrictions on device subprograms below, and must not refer to any data that is only accessible from device subprograms, such as the `threadidx` or `blockidx` builtin variables.

3.1.5 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

See also Section 3.6 on page 22.

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 restriction.

Variables declared in a device subprogram may have one of four attributes: they may be declared to be in device global memory, in constant memory space, in the thread block shared memory, or in thread local memory. Variables in modules may be declared to be in device global memory or constant memory space. CUDA Fortran also adds a new attribute for allocatable arrays in host memory; the array may be declared to be in pinned memory, that is, in page-locked host memory space. The advantage of using pinned memory is that transfers between the device and pinned memory are faster and can be asynchronous.

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)
```

An allocatable device array will dynamically allocate device global memory. Device variables and arrays may not have the Pointer or Target attributes. 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. 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 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)
```

Constant data may not have the Pointer, Target, or Allocatable attributes. Constant 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 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. 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:

- In declaration statements
- 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

3.2.3 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)
```

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 will have to take this into account when coding.

If a shared array is not a dummy argument and not assumed-size, it must be fixed size.

3.2.4 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.

3.2.5 Pinned arrays

An allocatable array with the pinned attribute will be allocated in special page-locked host memory, when such memory is available. 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.

3.3 Allocating Device and Pinned Arrays

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

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)
```

See section 4.4 for a complete list of the memory management runtime routines

3.3.3 Allocating 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 will allocate 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
   . . .
```

3.4 Data transfer between host and device memory

3.4.1 Data Transfer Using Assignment Statements

Variables and arrays can be copied from the host memory to the device memory by using simple assignment statements in host subprograms. An assignment statement where the left hand side is a device variable or device array or array section, and the right hand is a host variable or host array or array section, will copy 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, will copy 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 will copy data between two device variables or arrays.

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

Note that using assignment statements to read or write device or constant data implicitly uses CUDA stream zero. This means such data copies are synchronous, meaning the data copy will wait until all previous kernels and data copies complete.

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. Compiler-generated temporary arrays will be 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 will perform poorly. Array slices are also supported, and their performance is dependent on the size of the slice, 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 since 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. See section 4.4 for a complete list of the memory management runtime routines

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 will 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)`.
- `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. See Section 3.2.3 on page 18. 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 37. It specifies the stream to which this call is enqueued.

For instance, a kernel subroutine

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

can be called like:

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

The function call will fail if the `grid` or `block` arguments are greater than the maximum sizes allowed, or if `bytes` is greater than the shared memory available, allowing for static shared memory declared in the kernel and for other dedicated uses, such as the 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.

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 Builtin variables

The system module cudafor declares several predefined variables. These variables are read-only. They are declared as follows:

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

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 will have the value one.

The variable `blockdim` contains the dimensions of the thread block; `blockdim` will have 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 will have the value one

The variable `blockidx` contains the block index within the grid; as with `threadidx`, for one-dimensional grids, `blockidx%y` will have the value one. The value of `blockidx%z` is always one. The value of `blockidx` will be 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` will be 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 variable `warpsize` contains the number of threads in a warp. It has constant value, currently defined to be 32.

3.6.3 Fortran intrinsics

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

3.6.3.1 Fortran Numeric and Logical Intrinsics

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

3.6.3.2 Fortran Mathematical Intrinsics

name	argument datatypes
acos	real
asin	real
atan	real
atan2	(real,real)
cos	real, complex
cosh	real
exp	real, complex
log	real, complex

log10	real
sin	real, complex
sinh	real
sqrt	real, complex
tan	real
tanh	real

3.6.3.3 Fortran Numeric Inquiry Intrinsics

name	argument datatypes
bit_size	integer
digits	integer, real
epsilon	real
huge	integer, real
maxexponent	real
minexponent	real
precision	real, complex
radix	integer, real
range	integer, real, complex
selected_int_kind	integer
selected_real_kind	(integer,integer)
tiny	real

3.6.3.4 Fortran Bit Manipulation Intrinsics

name	argument datatypes
btest	integer
iand	integer
ibclr	integer
ibits	integer
ibset	integer
ieor	integer
ior	integer

ishft	integer
ishftc	integer
not	integer
mvbits	integer

3.6.3.5 Fortran Real Manipulation Intrinsics

name	argument datatypes
exponent	real
fraction	real
nearest	real
rrspacing	real
scale	(real,integer)
set_exponent	(real,integer)
spacing	real

3.6.3.6 Fortran Vector and Matrix Multiplication Intrinsics

name	argument datatypes
dot_product	integer, real, complex
matmul	integer, real, complex

3.6.3.7 Fortran Reduction Intrinsics

name	argument datatypes
all	logical
any	logical
count	logical
maxloc	integer, real
maxval	integer, real
minloc	integer, real
minval	integer, real

`product`	integer, real, complex
`sum`	integer, real, complex

3.6.3.8 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.

3.6.4.1 SYNCTHREADS

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

```
call syncthreads()
```

Each thread in a thread block will pause 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 will fail to complete correctly.

3.6.4.2 GPU_TIME

The `gpu_time` intrinsic returns the value of the clock cycle counter on the GPU. It has a single argument:

```
integer(8) clock
call gpu_time(clock)
```

The argument to `gpu_time` is set to the value of the clock cycle counter. . The clock frequency can be determined by calling cudaGetDeviceProperties; see Section 4.2.4.

3.6.4.3 ALLTHREADS

The `allthreads` function is a warp-vote operation; it is only supported by devices with compute capability 1.2 and higher. It has 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.

3.6.4.4 ANYTHREAD

The `anythread` function is a warp-vote operation; it is only supported by devices with compute capability 1.2 and higher. It has 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.

3.6.5 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. The atomic functions will 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.

3.6.5.1 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. Both arguments must be of type integer(kind=4). These functions are:

function	return value	additional atomic update
`atomicadd( mem, value )`	`mem`	`mem = mem + value`
`atomicsub( mem, value )`	`mem`	`mem = mem – value`
`atomicmax( mem, value )`	`mem`	`mem = max(mem,value)`
`atomicmin( mem, value )`	`mem`	`mem = min(mem,value)`
`atomicand( mem, value )`	`mem`	`mem = iand(mem,value)`
`atomicor( mem, value )`	`mem`	`mem = ior(mem,value)`
`atomicxor( mem, value )`	`mem`	`mem = ieor(mem,value)`
`atomicexch( mem, value )`	`mem`	`mem = value`

3.6.5.2 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).

These functions are:

function	return value	additional atomic update

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

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

3.6.5.3 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 type integer(kind=4).

The function is:

function	return value	additional atomic update

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

3.6.6 Restrictions

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

- Objects with the Pointer and Allocatable attribute are not allowed
- Automatic arrays must be fixed size
- Assumed-shape array arguments are not allowed
- Optional arguments are not allowed
- Objects with character type must have `LEN=1`; character substrings are not supported
- Recursive subroutines and functions are not allowed
- STOP and PAUSE statements are not allowed

- Input/Output statements are not allowed: READ, WRITE, PRINT, FORMAT, NAMELIST, OPEN, CLOSE, BACKSPACE, REWIND, ENDFILE, INQUIRE
- 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
- Subroutine and function calls are supported only if they can be inlined
- Cray pointers are not supported

3.7 Host code

3.7.1 SIZEOF Intrinsic

Host subprograms may use the new `sizeof` intrinsic function. A call to `sizeof(A)`, where `A` is a variable or expression, will return 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
```

4 Runtime API

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. See Section 4.7 on page 39 to interpret the error codes.

4.1 Initialization

No explicit initialization is required; the runtime will initialize and connect to the device the first time a runtime routine is called, or a device array is allocated. This initialization can add some overhead, so programmers need to be aware of this when doing timing runs.

4.2 Device Management

4.2.1 cudaGetDeviceCount

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

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

4.2.2 cudaSetDevice

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

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

4.2.3 cudaGetDevice

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

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

4.2.4 cudaGetDeviceProperties

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

`cudaGetDeviceProperties` returns the properties of a given device.

4.2.5 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.

4.3 Thread Management

4.3.1 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.

4.3.2 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.

4.4 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.

4.4.1 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 section 3.6.1. The `count` is in terms of elements. Or, `devptr` may be of TYPE(C_DEVPTR), in which case the `count` is in bytes.

4.4.2 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 section 3.6.1. 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.

4.4.3 cudaFree

```
integer function cudaFree(devptr)
```

`cudaFree` deallocates data on the device. `Devptr` may be any allocatable device array of a supported type specified in section 3.6.1. Or, `devptr` may be of TYPE(C_DEVPTR).

4.4.4 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.

4.4.5 cudaFreeArray

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

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

4.4.6 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 section 3.6.1. 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.

4.4.7 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 section 3.6.1. 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.

4.4.8 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 section 3.6.1. The `count` is in terms of elements. `Kdir` may be optional; see section 3.4.3. If it 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.

4.4.9 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 section 3.6.1. The width and height are in terms of elements. Kdir may be optional; see section 3.4.3. If it 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.

4.4.10 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.

4.4.11 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.

4.4.12 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.

4.4.13 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.

4.4.14cudaMemcpyArrayToArray

```
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.

4.4.15cudaMemcpy2DArrayToArray

```
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.

4.4.16cudaMalloc3D

```
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 section 3.6.1.

4.4.17cudaMalloc3DArray

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

`cudaMalloc3DArray` allocates array data on the device.

4.4.18cudaMemset3D

```
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.

4.4.19cudaMemcpy3D

```
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.

4.4.20cudaMemcpyToSymbol

```
integer function cudaMemcpyToSymbol(symbol, src, count,
offset, kdir)
 type(cudaSymbol) :: 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. another. Src may be any host scalar or array, of a supported type specified in section 3.6.1. The count is in terms of elements.

4.4.21cudaMemcpyFromSymbol

```
integer function cudaMemcpyFromSymbol(dst, symbol, count,
offset, kdir)
 type(cudaSymbol) :: symbol
 integer :: count, offset, kdir
```

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 section 3.6.1. The count is in terms of elements.

4.4.22cudaGetSymbolAddress

```
integer function cudaGetSymbolAddress(devptr, symbol)
 type(C_DEVPTR) :: devptr
 type(cudaSymbol) :: 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(cudaSymbol) :: 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
```

4.4.23cudaGetSymbolSize

```
integer function cudaGetSymbolSize(size, symbol)
 integer :: size
type(cudaSymbol) :: symbol
```

cudaGetSymbolSize sets the variable size to the size of a device area in global or constant memory space referenced by the symbol.

4.4.24 cudaMallocHost

```
integer function cudaMallocHost(hostptr, size)
 type(C_PTR) :: hostptr
 integer :: size
```

cudaMallocHost 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 normal iso_c_binding subroutine c_f_pointer can be used to move the type(c_ptr) to a Fortran pointer.

4.4.25 cudaFreeHost

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

cudaFreeHost deallocates pinned memory on the host allocated with cudaMalloHost.

4.5 Stream Management

4.5.1 cudaStreamCreate

```
integer function cudaStreamCreate( stream )
   integer, intent(out) :: stream
```

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

4.5.2 cudaStreamQuery

```
integer function cudaStreamQuery( stream )
   integer, intent(in) :: stream
```

cudaStreamQuery tests whether all operations enqueued to the selected stream are complete; it will return 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.

4.5.3 cudaStreamSynchronize

```
integer function cudaStreamSynchronize( stream )
   integer, 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.

4.5.4 cudaStreamDestroy

```
integer function cudaStreamDestroy( stream )
   integer, intent(in) :: stream
```

cudaStreamDestroy releases any resources associated with the given stream.

4.6 Event Management

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 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.

4.6.3 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 will return `cudaErrorInvalidValue` if `cudaEventRecord` has not been called for this event.

4.6.4 cudaEventSynchronize

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

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

4.6.5 cudaEventDestroy

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

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

4.6.6 cudaEventElapsedTime

```
integer function cudaEventElapsedTime( time, start, end )
  float :: 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.

4.7 Error Handling

4.7.1 cudaGetLastError

```
integer function cudaGetLastError()
```

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

4.7.2 cudaGetErrorString

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

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

5 Matrix Multiplication Example

5.1 Overview

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 submatris 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.

5.2 Source Code Listing

```
! 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
  tx = threadidx%x
  ty = threadidx%y
  ! This thread computes C(i,j) = sum(A(i,:) * B(:,j))
  i = (blockidx%x-1) * 16 + tx
  j = (blockidx%y-1) * 16 + ty
  Cij = 0.0
  ! Do the k loop in chunks of 16, the block size
  do kb = 1, M, 16
    ! Fill the submatrices
    ! Each of the 16x16 threads in the thread block
    ! loads one element of Asub and Bsub
```

```
    Asub(tx,ty) = A(i,kb+ty-1)
    Bsub(tx,ty) = B(kb+tx-1,j)
    ! Wait until all elements are filled
    call syncthreads()
    ! Multiply the two submatrices
    ! Each of the 16x16 threads accumulates the
    ! dot product for its element of C(i,j)
    do k = 1,16
      Cij = Cij + Asub(tx,k) * Bsub(k,ty)
    enddo
    ! Synchronize to make sure all threads are done
    ! reading the submatrices before overwriting them
    ! in the next iteration of the kb loop
    call syncthreads()
  enddo
  ! Each of the 16x16 threads stores its element
  ! to the global C array
  C(i,j) = Cij
 end subroutine mmul_kernel

 ! The host routine to drive the matrix multiplication
 subroutine mmul( A, B, C )
  real, dimension(:,:) :: A, B, C
  ! allocatable device arrays
  real, device, allocatable, dimension(:,:) :: Adev,Bdev,Cdev
  ! dim3 variables to define the grid and block shapes
  type(dim3) :: dimGrid, dimBlock

  ! Get the array sizes
  N = size( A, 1 )
  M = size( A, 2 )
  L = size( B, 2 )
  ! Allocate the device arrays
  allocate( Adev(N,M), Bdev(M,L), Cdev(N,L) )

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

  ! Create the grid and block dimensions
  dimGrid = dim3( N/16, M/16, 1 )
  dimBlock = dim3( 16, 16, 1 )
  call mmul_kernel<<<dimGrid,dimBlock>>>( Adev, Bdev, Cdev, &
                                          N, M, L )
  ! Copy the results back and free up memory
  C(1:N,1:L) = Cdev
  deallocate( Adev, Bdev, Cdev )
 end subroutine mmul
end module mmul_mod
```

5.3 Source Code Discussion

This source code module `mmul_mod` has two subroutines. The host subroutine `mmul` is a wrapper for the kernel routine `mmul_kernel`.

5.3.1 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.

5.3.2 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`