# Declaring Fortran Device Data

- Variables / arrays with device attribute are allocated in device memory

```
real, device, allocatable :: a(:)
real, allocatable :: a(:)
attributes(device) :: a
```

- In a host subroutine or function
  - device allocatables and automatics may be declared
  - device variables and arrays may be passed to other host subroutines or functions (explicit interface)
  - device variables and arrays may be passed to kernel subroutines

# Declaring Fortran Module Data

- Variables / arrays with device attribute are allocated in device memory

```
module mm
    real, device, allocatable :: a(:)
    real, device :: x, y(10)
    real, constant :: c1, c2(10)
    integer, device :: n
  contains
    attributes(global) subroutine s( b )
  ...
```

- Module data must be fixed size, or allocatable

# Allocating Data

- Fortran allocate / deallocate statement

```
real, device, allocatable :: a(:,:), b
allocate( a(1:n,1:m), b )
....
deallocate( a, b )
```

- arrays or variables with device attribute are allocated in device memory
  - Allocate is done by the host subprogram
  - Memory is not virtual, you can run out
  - Device memory is shared among users / processes, you can have deadlock
  - `STAT=ivar` clause to catch and test for errors

# Copying Data to / from Device

- Assignment statements

```
real, device, allocatable :: a(:,:), b
allocate( a(1:n,1:m), b )
a(1:n,1:m) = x(1:n,1:m)   ! copies to device
b = 99.0
....
x(1:n,1:m) = a(1:n,1:m)! copies from device
y = b
deallocate( a, b )
```

- Data copy may be noncontiguous, but will then be slower (multiple DMAs)
- Data copy to / from host pinned memory will be faster
- Asynchronous copies currently require API interface

# Launching Kernels

- Subroutine call with chevron syntax for launch configuration

```
call vaddkernel <<<(N+31)/32,32 >>> (A,B,C,N)

type(dim3) :: g, b
g = dim3((N+31)/32, 1, 1)
b = dim3( 32, 1, 1 )
call vaddkernel <<< g, b >>> ( A, B, C, N )
```

- Interface must be explicit
  - In the same module as the host subprogram
  - In a module that the host subprogram uses
  - Declared in an interface block
- The launch is asynchronous
  - host program continues, may issue other launches

# CUDA Errors

- Out of memory
- Launch failure (array out of bounds, ...)
- No device found
- Invalid device code (compute capability mismatch)

Test for error:

```
ir = cudaGetLastError()
if( ir ) print *, cudaGetErrorString( ir )

ir = cudaGetLastError();
if( ir ) printf( "%s\n", cudaGetErrorString(ir) );
```

# Writing a CUDA Kernel (1)

- C: global attribute on the function header, must be void type
  - `__global__ void kernel ( ... ){...}`
- F: global attribute on the subroutine statement
  - `attributes(global) subroutine kernel ( A, B, C, N )`
- May declare scalars, fixed size arrays in local memory
- May declare shared memory arrays
  - `C: __shared__ float sm(16,16);`
  - `F: real, shared :: sm(16,16)`
  - Limited amount of shared memory available (16KB, 48KB)
  - shared among all threads in the same thread block
- Data types allowed
  - int (long,short,char), float, double, struct, union, ...
  - integer(1,2,4,8), logical(1,2,4,8), real(4,8), complex(4,8), derivedtype

# Writing a CUDA Kernel (2)

- Predefined variables
  - `blockIdx, threadIdx, gridDim, blockDim, warpSize`
- Executable statements in a kernel
  - assignment
  - for, do, while, if, goto, switch
  - function call to device function
  - intrinsic function call
  - most intrinsics implemented in header files

# Writing a CUDA Kernel (3)

- Fortran disallowed statements include
  - read, write, print, open, close, inquire, format, other IO except now some limited support for list-directed (print *)
  - allocate, deallocate, until PGI 13.0
  - Fortran pointer assignment
  - recursive procedure calls, direct or indirect
  - ENTRY statement, optional arguments, alternate return
  - data initialization, SAVEd data
  - assigned goto, ASSIGN statement
  - stop, pause

# Exercise Solution

- multidim_solution.cuf
- In the book on page 12

# Kernel Loop Directives (CUF Kernels)

- Automatic kernel generation and invocation of host code region (arrays used in loops must reside on GPU)

```
program incTest
  use cudafor
  implicit none
  integer, parameter :: n = 256
  integer :: a(n), b
  integer, device :: a_d(n)

  a = 1; b = 3; a_d = a

  !$cuf kernel do <<<*,*>>>
  do i = 1, n
     a_d(i) = a_d(i)+b
  enddo

  a = a_d
  if (all(a == 4)) write(*,*) 'Test Passed'
end program incTest
```

# Kernel Loop Directives (CUF Kernels)

- Multidimensional arrays

```
!$cuf kernel do(2) <<< *,* >>>
do j = 1, ny
  do i = 1, nx
    a_d(i,j) = b_d(i,j) + c_d(i,j)
  enddo
enddo
```

- Can specify parts of execution parameter

```
!$cuf kernel do(2) <<<(*,*),(32,4)>>>
```

# Kernel Loop Directives (CUF Kernels)

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.

# Kernel Loop Directive Syntax

```
!$cuf kernel do(7) <<< *,(32,8,1,1,1,1,1) >>>
 do j7 = 1, is27
  do j6 = 1, is26
   do j5 = 1, is25
    do j4 = 1, is24
     do j3 = 1, is23
      do j2 = 1, is22
       do j1 = 1, is21
        . . .
```

# Reduction using CUF Kernels

- Compiler recognizes use of scalar reduction and generates one result

```
rsum = 0.0
!$cuf kernel do <<<*,*>>>
do i = 1, nx
  rsum = rsum + a_d(i)
enddo
```

# !$CUF kernel directives

```
module madd_device_module
  use cudafor
contains
  subroutine madd_dev(a,b,c,sum,n1,n2)
    real,dimension(:,:),device :: a,b,c
    real :: sum
    integer :: n1,n2
    type(dim3) :: grid, block
!$cuf kernel do (2) <<<(*,*),(32,4)>>>
    do j = 1,n2
      do i = 1,n1
        a(i,j) = b(i,j) + c(i,j)
        sum = sum + a(i,j)
      enddo
    enddo
  end subroutine
end module
```

Equivalent
hand-written
CUDA kernels

```
module madd_device_module
  use cudafor
  implicit none
contains
  attributes(global) subroutine madd_kernel(a,b,c,blocksum,n1,n2)
    real, dimension(:,:) :: a,b,c
    real, dimension(:) :: blocksum
    integer, value :: n1,n2
    integer :: i,j,tindex,tneighbor,bindex
    real :: mysum
    real, shared :: bsum(256)
! Do this thread's work
    mysum = 0.0
    do j = threadidx%y + (blockidx%y-1)*blockdim%y, n2, blockdim%y*griddim%y
      do i = threadidx%x + (blockidx%x-1)*blockdim%x, n1, blockdim%x*griddim%x
        a(i,j) = b(i,j) + c(i,j)
        mysum = mysum + a(i,j) ! accumulates partial sum per thread
      enddo
    enddo
! Now add up all partial sums for the whole thread block
! Compute this thread's linear index in the thread block
! We assume 256 threads in the thread block
    tindex = threadidx%x + (threadidx%y-1)*blockdim%x
! Store this thread's partial sum in the shared memory block
    bsum(tindex) = mysum
    call syncthreads()
! Accumulate all the partial sums for this thread block to a single value
    tneighbor = 128
    do while( tneighbor >= 1 )
      if( tindex <= tneighbor ) &
        bsum(tindex) = bsum(tindex) + bsum(tindex+tneighbor)
      tneighbor = tneighbor / 2
      call syncthreads()
    enddo
! Store the partial sum for the thread block
    bindex = blockidx%x + (blockidx%y-1)*griddim%x
    if( tindex == 1 ) blocksum(bindex) = bsum(1)
  end subroutine

! Add up partial sums for all thread blocks to a single cumulative sum
  attributes(global) subroutine madd_sum_kernel(blocksum,dsum,nb)
    real, dimension(:) :: blocksum
    real :: dsum
    integer, value :: nb
    real, shared :: bsum(256)
    integer :: tindex,tneighbor,i
! Again, we assume 256 threads in the thread block
! accumulate a partial sum for each thread
    tindex = threadidx%x
    bsum(tindex) = 0.0
    do i = tindex, nb, blockdim%x
      bsum(tindex) = bsum(tindex) + blocksum(i)
    enddo
    call syncthreads()
! This code is copied from the previous kernel
! Accumulate all the partial sums for this thread block to a single value
! Since there is only one thread block, this single value is the final result
    tneighbor = 128
    do while( tneighbor >= 1 )
      if( tindex <= tneighbor ) &
        bsum(tindex) = bsum(tindex) + bsum(tindex+tneighbor)
      tneighbor = tneighbor / 2
      call syncthreads()
    enddo
    if( tindex == 1 ) dsum = bsum(1)
  end subroutine

  subroutine madd_dev(a,b,c,dsum,n1,n2)
    real, dimension(:,:), device :: a,b,c
    real, device :: dsum
    real, dimension(:), allocatable, device :: blocksum
    integer :: n1,n2,nb
    type(dim3) :: grid, block
    integer :: r
! Compute grid/block size; block size must be 256 threads
    grid = dim3((n1+31)/32, (n2+7)/8, 1)
    block = dim3(32,8,1)
    nb = grid%x * grid%y
    allocate(blocksum(1:nb))
    call madd_kernel<<< grid, block >>>(a,b,c,blocksum,n1,n2)
    call madd_sum_kernel<<< 1, 256 >>>(blocksum,dsum,nb)
    r = cudaThreadSynchronize() ! don't deallocate too early
    deallocate(blocksum)
  end subroutine
end module
```

# Exercise - CUF Kernels

- Modify multidim.cuf to use a CUF kernel
- Solution in multidim_CUF_solution.cuf
- Extra credit: calculate the sum of all elements in the array
- Consult section 3.7 *Kernel Loop Directives* in book if needed

# Compute Capabilities

| Architecture | Tesla | | | | Fermi | | Kepler | | | Maxwell | | Pascal | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Compute capabilities | 1 | 1.1 | 1.2 | 1.3 | 2 | 2.1 | 3 | 3.5 | 3.7 | 5.0 | 5.x | 6.0 | 6.x |
| Double precision | | | | | Full | | | Full | Full | | | Full | |
| 3D grids | | | | | | | | | | | | | |
| Max # threads per block | 512 | | | | 1024 | | | | | | | | |
| Shared memory per MP | 16Kb | | | | | 48Kb (16/48,48/16) | | | 48Kb +(32/32) | | | | |
| ..... | ..... | | | | ... | | | | ... | | | | |

- All these values are returned by **`cudaGetDeviceProperties`** or **`pgaccelinfo`**
- Target GPU can be specified with **`-Mcuda=ccxy`**

# Device Query Code

```
type (cudaDeviceProp) :: prop
integer :: nDevices=0, i, ierr

ierr = cudaGetDeviceCount(nDevices)
…
do i = 0, nDevices-1

   write(*,"('Device Number: ',i0)") i
   ierr = cudaGetDeviceProperties(prop, i)

   write(*,"('  Device Name: ',a)") trim(prop%name)
   write(*,"('  Compute Capability: ',i0,'.',i0)") &
        prop%major, prop%minor
   write(*,"('  Number of Multiprocessors: ',i0)") &
        prop%multiProcessorCount
   write(*,"('  Max Threads per Multiprocessor: ',i0)") &
        prop%maxThreadsPerMultiprocessor
   write(*,"('  Global Memory (GB): ',f9.3,/)") &
        prop%totalGlobalMem/1024.0**3
   …
```

# Compile and Run deviceQuery.cuf

- Note: devices are enumerated from 0
- Full blown deviceQuery? `pgaccelinfo` utility

# Compilation

- PGI's Fortran compiler
  - All source code with .cuf or .CUF is compiled as CUDA Fortran enabled automatically
  - Flag to target architecture (eg. -Mcuda=cc35)
  - -Mcuda=emu specifies emulation mode
  - Flag to target CUDA Toolkit version (eg. -Mcuda=cuda5.5)
  - -Mcuda=fastmath enables faster intrinsics (__sinf())
  - -Mcuda=nofma turns off fused multiply-add
  - -Mcuda=maxregcount:<n> limits register usage per thread
  - -Mcuda=ptxinfo prints memory usage per kernel

Use pgf90 -Mcuda -help for a full list