Parallel Programming with CUDA Fortran
Outline

- What is CUDA Fortran
- Simple Examples
- CUDA Fortran Features
- Using CUBLAS with CUDA Fortran
- Compilation
CUDA Fortran

CUDA is a scalable programming model for parallel computing

CUDA Fortran is the Fortran analog of CUDA C
- Program host and device code similar to CUDA C
- Host code is based on Runtime API
- Fortran language extensions to simplify data management

Co-defined by NVIDIA and PGI, implemented in the PGI Fortran compiler
- Separate from PGI Accelerator
  - Directive-based, OpenMP-like interface to CUDA
CUDA Programming

- Heterogeneous programming model
  - CPU and GPU are separate devices with separate memory spaces
  - Host code runs on the CPU
    - Handles data management for both the host and device
    - Launches kernels which are subroutines executed on the GPU
  - Device code runs on the GPU
    - Executed by many GPU threads in parallel
    - Allows for incremental development
module simpleOps_m
contains
    subroutine inc(a, b)
        implicit none
        integer :: a(:)
        integer :: b
        integer :: i, n

        n = size(a)
        do i = 1, n
            a(i) = a(i)+b
        enddo
    end subroutine inc
end module simpleOps_m

program incTest
    use simpleOps_m
    implicit none
    integer, parameter :: n = 256
    integer :: a(n), b

    a = 1 ! array assignment
    b = 3
    call inc(a, b)

    if (all(a == 4)) then
        write(*,*) 'Success'
    endif
end program incTest
CUDA Fortran

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

  a = 1
  b = 3

  a_d = a
  call inc<<<1,n>>>(a_d, b)
  a = a_d

  if (all(a == 4)) then
    write(*,*) 'Success'
  endif
end program incTest
```

F90

```fortran
program incTest
  use simpleOps_m
  implicit none
  integer, parameter :: n = 256
  integer :: a(n), b

  a = 1
  b = 3

  call inc(a, b)

  if (all(a == 4)) then
    write(*,*) 'Success'
  endif
end program incTest
```
CUDA Fortran

module simpleOps_m
contains
  attributes(global) subroutine inc(a, b)
   implicit none
   integer :: a(:)
   integer, value :: b
   integer :: i
   i = threadIdx%x
   a(i) = a(i)+b
  end subroutine inc
end module simpleOps_m

F90

module simpleOps_m
contains
  subroutine inc(a, b)
   implicit none
   integer :: a(:)
   integer :: b
   integer :: i, n
   n = size(a)
   do i = 1, n
     a(i) = a(i)+b
   enddo
end subroutine inc
end module simpleOps_m
Extending to Larger Arrays

- Previous example works for small arrays

  \[
  \text{call inc}^{<<<1,n>>}(a_d,b)
  \]

- Limit of \(n=1024\) (Fermi) or \(n=512\) (pre-Fermi)

- For larger arrays, change the first Execution Configuration parameter (\(<<<1,n>>\))
Execution Model

**Software**
- Thread

**Hardware**
- Thread Processor
- Multiprocessor
- Thread blocks are executed on multiprocessors
- Thread blocks do not migrate
- Several concurrent thread blocks can reside on a multiprocessor

**Device**
- Grid
- A kernel is launched on a device as a grid of thread blocks
Execution Configuration

Execution configuration specified on host code

call inc<<<blocksPerGrid, threadsPerBlock>>>(a_d, b)

Previous example used a single thread block

call inc<<<1, n>>>(a_d, b)

Multiple threads blocks

tPB = 256

call inc<<<ceiling(real(n)/tPB), tPB>>>(a_d, b)
program incTest
   use cudafor
   use simpleOps_m
   implicit none
   integer, parameter :: n = 1024*1024
   integer, parameter :: tPB = 256
   integer :: a(n), b
   integer, device :: a_d(n)

   a = 1
   b = 3

   a_d = a
   call inc<<<ceiling(real(n)/tPB),tPB>>>(a_d, b)
   a = a_d

   if (all(a == 4)) then
      write(*,*) 'Success'
   endif
end program incTest
module simpleOps_m
contains
  attributes(global) subroutine inc(a, b)
  implicit none
  integer :: a(:)
  integer, value :: b
  integer :: i, n

  i = (blockIdx%x-1)*blockDim%x + threadIdx%x
  n = size(a)
  if (i <= n) a(i) = a(i)+b

end subroutine inc
end module simpleOps_m
Multidimensional Arrays - Host

**Execution Configuration**

```fortran
  call inc<<<blocksPerGrid, threadsPerBlock>>>(a_d,b)
```

- Grid dimensions in blocks (`blocksPerGrid`) and block dimensions (`threadsPerBlock`) can be either `integer` or of type `dim3`

```fortran
  type (dim3)
  integer (kind=4) :: x, y, z
end type
```
Multidimensional Arrays - Device

- Predefined variables in device subroutines
  - Grid and block dimensions: `gridDim, blockDim`
  - Block and thread indices: `blockIdx, threadIdx`
  - Of type `dim3`

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

- `blockIdx` and `threadIdx` fields have unit offset

```
1 <= blockIdx%x <= blockDim%x
```
program incTest
    use cudafor
    use simpleOps_m
    implicit none
    integer, parameter :: nx=1024, ny=512
    real :: a(nx,ny), b
    real, device :: a_d(nx,ny)
    type(dim3) :: grid, tBlock

    a = 1; b = 3

    tBlock = dim3(32,8,1)
    grid = dim3(ceiling(real(nx)/tBlock%x), ceiling(real(ny)/tBlock%y), 1)
    a_d = a
    call inc<<<grid,tBlock>>>(a_d, b)
    a = a_d

    write(*,*) 'Max error: ', maxval(abs(a-4))
end program incTest
module simpleOps_m
contains
  attributes(global) subroutine inc(a, b)
    implicit none
    real :: a(:,:,)
    real, value :: b
    integer :: i, j

    i = (blockIdx%x-1)*blockDim%x + threadIdx%x
    j = (blockIdx%y-1)*blockDim%y + threadIdx%y

    if (i<=size(a,1) .and. j<=size(a,2)) &
       a(i,j) = a(i,j) + b

  end subroutine inc
end module simpleOps_m
CUDA Fortran Features

- Variable Qualifiers
- Subroutine/Function Qualifiers
- Kernel Loop Directives (CUF Kernels)
Variable Qualifiers

- Analogous to CUDA C
  - `device`
  - `constant`
    - Read-only memory (device code) cached on-chip
  - `shared`
    - On-chip, shared between threads of a thread block
- Additional
  - `pinned`
    - Page-locked host memory
  - `value`
    - Pass-by-value dummy arguments in device code
- Textures will be available in 12.0
Function/Subroutine Qualifiers

- Designated by `attributes()` specifier
  - `attributes(host)`
    - called from host and runs on host (default)
  - `attributes(global)`
    - kernel, called from host runs on device
    - subroutine only
    - no other prefixes allowed (`recursive`, `elemental`, or `pure`)
  - `attributes(device)`
    - called from and runs on device
    - can only appear within a Fortran module
    - only additional prefix allowed is function return type
Kernel Loop Directives (CUF Kernels)

- Automatic kernel generation and invocation of host code region containing tightly nested loops
  
  ```
  !$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
  enddo
  ```

- Can specify parts of execution configuration
  
  ```
  !$cuf kernel do(2) <<<(*,*),(32,4)>>>
  ```
Reduction using CUF Kernels

Compiler recognizes use of scalar reduction and generates one result

```plaintext
rsum = 0.0
!$cuf kernel do <<<*,*>>> 
do i = 1, nx
    rsum = rsum + a_d(i)
enddo
```
Calling CUBLAS from CUDA Fortran

Module which defines interfaces to CUBLAS from CUDA Fortran

```
use cublas
```

Interfaces in three forms

- Overloaded BLAS interfaces that take device array arguments
  
  ```
call saxpy(n, a_d, x_d, incx, y_d, incy)
  ```

- Legacy CUBLAS interfaces
  
  ```
call cublasSaxpy(n, a_d, x_d, incx, y_d, incy)
  ```

- Multi-GPU version (CUDA 4.0) that utilizes a handle \( h \)
  
  ```
  istat = cublasSaxpy_v2(h, n, a_d, x_d, incx, y_d, incy)
  ```

Mixing the three forms is allowed
Calling CUBLAS from CUDA Fortran

program cublasTest
    use cublas
    implicit none

    real, allocatable :: a(:,:), b(:,:), c(:,:)
    real, device, allocatable :: a_d(:,:), b_d(:,:), c_d(:,:)
    integer :: k=4, m=4, n=4
    real :: alpha=1.0, beta=2.0, maxError

    allocate(a(m,k), b(k,n), c(m,n), a_d(m,k), b_d(k,n), c_d(m,n))

    a = 1; a_d = a
    b = 2; b_d = b
    c = 3; c_d = c

    call cublasSgemm('N','N',m,n,k,alpha,a_d,m,b_d,k,beta,c_d,m)

    c=c_d
    write(*,*) 'Maximum error: ', maxval(abs(c-14.0))

    deallocate (a,b,c,a_d,b_d,c_d)

end program cublasTest
Compilation

Source-to-source compilation (generates CUDA C)

- `pgfortran` - PGI’s Fortran compiler

All source code with `.cuf` or `.CUF` is compiled as CUDA Fortran enabled automatically

- Flag to target architecture (eg. `-Mcuda=cc20`)
  - `-Mcuda=emu` specifies emulation mode

- Flag to target toolkit version (eg. `-Mcuda=cuda4.0`)
  - `-Mcuda=fastmath` enables faster intrinsics (`__sinf()`)
  - `-Mcuda=nofma` turns off fused multiply-add
  - `-Mcuda=maxregcount:<n>` limits register use per thread
  - `-Mcuda=ptxinfo` prints memory usage per kernel
CUDA Fortran provides a convenient interface for parallel programming

Fortran analog to CUDA C
  - CUDA Fortran has strong typing that allows simplified data management
  - Fortran 90’s array features carried to GPU

More info available at
  - http://www.pgroup.com/cudafortran
Parallel Programming with CUDA Fortran
Runtime API (Host)

- Runtime API defined in `cudafor` module
  - Device management (`cudaGetDeviceCount`, `cudaSetDevice`, ...)
  - Host-device synchronization (`cudaDeviceSynchronize`)
  - Memory management (`cudaMalloc/cudaFree`, `cudaMemcpy`, `cudaMemcpyAsync`, ...)
  - Mixing `cudaMalloc/cudaFree` with Fortran allocate/deallocate on a given array is not supported
  - For device data, counts are in units of elements, not bytes
  - Stream management (`cudaStreamCreate`, `cudaStreamSynchronize`, ...)
  - Event management (`cudaEventCreate`, `cudaEventRecord`, ...)
  - Error handling (`cudaGetLastError`, ...)
Device Intrinsics

**syncthreads subroutine**
- Barrier synchronization for all threads in thread block

**gpu_time subroutine**
- Returns value of clock cycle counter on GPU

**Atomic functions**