Introduction to CUDA Fortran
Outline

• Introduction
  - Basic Concepts
  - Simple Examples
  - Kernel Loop Directives (CUF Kernels)
  - Compute Capabilities
  - Compilation

• Performance Optimization

• Examples

• Multi-GPU Programming
Introduction

• CUDA is a scalable model for parallel computing

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

• Co-defined by NVIDIA and PGI, implemented in the PGI Fortran compiler
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 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
Heterogeneous Programming

- Host = CPU and its memory
- Device = GPU and its memory

• Typical code progression
  - Allocate memory on host and device
  - Transfer data from host to device
  - Execute kernel (device computation)
  - Transfer result from device to host
  - Deallocate memory
Data Transfers

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

    a = 1.0
    a_d = a
    b_d = a_d
    b = b_d

    if (all(a == b)) &
        write(*,*) 'Test Passed'
end program copyData
```
Data Transfers

program copyData
  use cudafor
  implicit none
  integer, parameter :: n = 256
  real :: a(n), b(n)
  real, device :: a_d(n), b_d(n)
  a = 1.0
  a_d = a
  b_d = a_d
  b = b_d
  if (all(a == b)) &
      write(*,*) 'Test Passed'
end program copyData
Data Transfers

program copyData
    use cudafor
    implicit none
    integer, parameter :: n = 256
    real :: a(n), b(n)
    real, device :: a_d(n), b_d(n)
    a = 1.0
    a_d = a
    b_d = a_d
    b = b_d
    if (all(a == b)) &
       write(*,*) 'Test Passed'
end program copyData
Data Transfers

program copyData
    use cudafor
    implicit none
    integer, parameter :: n = 256
    real :: a(n), b(n)
    real, device :: a_d(n), b_d(n)

    a = 1.0
    a_d = a
    b_d = a_d
    b = b_d

    if (all(a == b)) &
       write(*,*) 'Test Passed'
end program copyData
Data Transfers

program copyData
    use cudafor
    implicit none
    integer, parameter :: n = 256
    real :: a(n), b(n)
    real, device :: a_d(n), b_d(n)

    a = 1.0
    a_d = a
    b_d = a_d
    b = b_d

    if (all(a == b)) &
        write(*,*) 'Test Passed'
end program copyData
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 :: b, n = 256
   integer, allocatable:: a(:)

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

   if (all(a == 4)) &
      write(*,*) 'Test Passed'
   deallocate(a)
end program incTest
CUDA Fortran - Host Code

**F90**

```fortran
program incTest

  use simpleOps_m
  implicit none
  integer :: b, n = 256
  integer, allocatable:: a(:)

  allocate (a(n))
  a = 1   ! array assignment
  b = 3

  call inc(a, b)

  if (all(a == 4)) &
    write(*,*) 'Test Passed'
  deallocate(a)

end program incTest
```

**CUDA Fortran**

```fortran
program incTest

  use cudafor
  use simpleOps_m
  implicit none
  integer :: b, n = 256
  integer, allocatable :: a(:)
  integer, allocatable, device :: a_d(:)

  allocate (a(n),a_d(n))
  a = 1
  b = 3

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

  if (all(a == 4)) &
    write(*,*) 'Test Passed'
  deallocate(a,a_d)

end program incTest
```
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

module simpleOps_m
contains
  attribute(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
Extending to Larger Arrays

- Previous example works with small arrays
  
  \[
  \text{call inc} \langle \langle 1, n \rangle \rangle (a_d, b)
  \]

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

- For larger arrays, change the first execution parameter \(\langle \langle 1, n \rangle \rangle\)
**Execution Model**

**Software**
- Thread

**Hardware**
- **Thread Processor**
- **Multiprocessor**
  - Threads are executed by thread processors
  - 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

**Software**
- Thread

**Hardware**
- **Thread Processor**
  - Threads are executed by thread processors

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

- Execution configuration specified in host code
  
  \[
  \text{call inc}<<<\text{blocksPerGrid, threadsPerBlock}>>>(a_d, b)
  \]

- Previous example used a single thread block
  
  \[
  \text{call inc}<<<1, n>>>(a_d, b)
  \]

- Multiple thread blocks
  
  \[
  \text{tPB} = 256 \\
  \text{call inc}<<<\text{ceiling}(\text{real}(n)/\text{tPB}), \text{tPB}>>>(a_d, b)
  \]
Mapping Arrays to Thread Blocks

• call inc<<<3,4>>>(a_d, b)

blockDim%x = 4
blockIdx%x

threadIdx%x

(blockIdx%x-1)*blockDim%x + threadIdx%x
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
Built-in Variables for Device Code

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

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

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

```plaintext
 1 <= blockIdx%x <= blockDim%x
```
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 Example - Host

- Execution Configuration

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

- Grid dimensions in blocks ($blocksPerGrid$) and block dimensions in threads ($threadsPerBlock$) can be either integer or dim3

```fortran
    type (dim3)
    integer (kind=4) :: x, y, z
    end type
```
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
2D Example - Device Code

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
Outline

• Introduction
  - Basic Concepts
  - Simple Examples
  - Kernel Loop Directives (CUF Kernels)
  - Compute Capabilities
  - Compilation

• Performance Optimization

• Examples

• Multi-GPU Programming
Kernel Loop Directives (CUF Kernels)

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

```fortran
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 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)>>> 
  ```
Reduction using CUF Kernels

• Compiler recognizes use of scalar reduction and generates one result

```c
rsum = 0.0
!$cuf kernel do <<<*,*>>>
do i = 1, nx
    rsum = rsum + a_d(i)
endo
```
## Compute Capabilities

<table>
<thead>
<tr>
<th>Architecture</th>
<th>Tesla</th>
<th>Tesla</th>
<th>Tesla</th>
<th>Tesla</th>
<th>Fermi</th>
<th>Fermi</th>
<th>Fermi</th>
<th>Fermi</th>
<th>Kepler</th>
<th>Kepler</th>
<th>Kepler</th>
<th>Kepler</th>
</tr>
</thead>
<tbody>
<tr>
<td>Compute capabilities</td>
<td>1.0</td>
<td>1.1</td>
<td>1.2</td>
<td>1.3</td>
<td>2.0</td>
<td>2.1</td>
<td>3.0</td>
<td>3.5</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Double precision</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3D grids</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Max # threads per block</td>
<td>512</td>
<td></td>
<td></td>
<td></td>
<td>1024</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Shared memory per MP</td>
<td>16Kb</td>
<td></td>
<td></td>
<td></td>
<td>48Kb</td>
<td>(16/48,48/16)</td>
<td>48Kb</td>
<td>(32/32)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>.....</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>.....</td>
<td></td>
<td>.....</td>
<td>.....</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

• *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=cc35`)
  - `-Mcuda=emu` specifies emulation mode
  - Flag to target CUDA Toolkit version (eg. `-Mcuda=cuda5.0`)
  - `-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

• *pgfortran  -Mcuda  -help* for a full list
Separate Compilation

• Device data can be shared between modules via F90 use
  - No special compilation flags needed

• Device functions called between modules requires compilation with `-Mcuda=rdc`
  - `rdc` = relocatable device code
  - Device must be cc20 or higher (Fermi and Kepler)
  - Use on compile and link lines
  - `-Mcuda=rdc` implies `-Mcuda=cuda5.0` and CC >= 2.0
  - requires 13.3 compiler
Separate Compilation

module devFunc_m
contains
  attributes(device) subroutine negate(a)
  integer :: a
  a = -a
end subroutine negate
end module devFunc_m

module kernel_m
contains
  attributes(global) subroutine difference(a,b)
    use devFunc_m
    integer :: a, b, c
    c = b
    call negate(c)
    a = a+c
end subroutine difference
end module kernel_m

program aMinusB
  use kernel_m
  integer :: a
  integer, device :: a_d, b_d
  a_d = 2; b_d = 3
  call difference<<<1,1>>>(a_d, b_d)
  a = a_d
  write(*,"
2-3=",i0)")) a
end program aMinusB

$ pgf90 -Mcuda=rdc -c devFunc.cuf
$ pgf90 -Mcuda=rdc -c kernel.cuf
$ pgf90 -Mcuda=rdc aMinusB.cuf devFunc.o kernel.o
Outline

• Introduction

• Performance Optimization
  - Host-Device Data Transfers
  - Device Memory
  - Execution Configuration

• Examples

• Multi-GPU Programming
Host-Device Transfers

- Host-device bandwidth is much lower than bandwidth within device
  - 8 GB/s peak (PCIe x16 Gen 2) vs. 250 GB/s peak (Tesla K20X)

- Minimize number of transfers
  - Intermediate data can be allocated, used, and deallocated without copying to host memory
  - Sometimes better to do low parallelism operations on the GPU if it avoids transfers to and from host
Page-Locked Data Transfers

**Pageable Data Transfer**

- Device
- DRAM
- Host
  - Pageable memory
  - Pinned buffer

**Page-locked Data Transfer**

- Device
- DRAM
- Host
  - Pin memory
Page-Locked Data Transfers

- Page-locked or pinned host memory by declaration
  - Designated by `pinned` variable attribute
  - Must be `allocatable`

```fortran
real, device :: a_d(N)
real, pinned, allocatable :: a(:)
allocate(a(N), STAT=istat, PINNED=pinnedFlag)
...
a_d = a
```

- Tesla K20/Sandy Bridge
  - Pageable: ~3.3 GB/s
  - Pinned: ~6 GB/s
Overlapping Transfers and Computation

- Kernel launches are asynchronous, normal memory copies are blocking
  ```
  a_d = a  // blocks on host until transfer completes
  call inc<<<g,b>>>(a_d, b)  // Control returns immediately to CPU
  a = a_d  // starts only after kernel completes
  ```

- Asynchronous and Stream APIs allow overlap of transfers with computation

- A stream is a sequence of operations that execute in order on the GPU
  - Operations in different (non-default) streams can be interleaved
  - Stream ID used as arguments to async transfers and kernel launches
Asynchronous Data Transfers

- Asynchronous host-device transfers return control immediately to CPU
  - `cudaMemcpyAsync(dst, src, nElements, stream)`
  - Requires pinned host memory

- Overlapping data transfer with CPU computation
  - default stream = 0

```c
istat = cudaMemcpyAsync(a_d, a_h, N, 0)
call kernel<<<grid, block>>>(a_d)
call cpuFunction(b)
```
Overlapping Transfers and Kernels

• Requires:
  - Pinned host memory
  - Kernel and transfer to use different non-zero streams

integer (kind=cuda_stream_kind) :: stream1, stream2
...
istat = cudaStreamCreate(stream1)
istat = cudaStreamCreate(stream2)
istat = cudaMemcpyAsync(a_d, a_h, N, stream1)
call kernel<<grid, block, 0, stream2>>>(b_d) overlapped
GPU/CPU Synchronization

• `cudaDeviceSynchronize()`
  - Blocks until all previously issued operations on the GPU complete

• `cudaStreamSynchronize(stream)`
  - Blocks until all previously issued operations to `stream` complete

• `cudaStreamQuery(stream)`
  - Indicates whether `stream` is idle
  - Does not block CPU code
GPU/CPU Synchronization

- Stream-based using CUDA events
  - Events can be inserted into streams
    
    ```
    type (cudaEvent) :: event
    ...
    istat = cudaEventRecord(event, stream1)
    ```

  - Event is recorded when the GPU reaches it in the stream
    - Recorded = assigned a time stamp
    - Useful for timing code

- `cudaEventSynchronize(event)`
  - Blocks CPU until event is recorded
Outline

- Introduction
- Performance Optimization
  - Host-Device Data Transfers
  - Device Memory
  - Execution Configuration
- Examples
- Multi-GPU Programming
Coalescing

- Device data accessed by a group of 16 or 32 sequential threads can result in as little as a single transaction if certain requirements are met
  - Alignment
  - Stride
  - Generation of GPU architecture
Misaligned Data Access

- Use array increment kernel with variable misalignment

```fortran
attributes(global) subroutine offset(a, s)
    real :: a(*)
    integer, value :: s
    integer :: i
    i = blockDim%x*(blockIdx%x-1)+threadIdx%x + s
    a(i) = a(i)+1
end subroutine offset
```

Array `a()` allocated with padding to avoid out-of-bounds accesses
Misaligned Data Access
Strided Data Access

- Use array increment kernel with variable stride

```fortran
attributes(global) subroutine stride(a, s)
    real :: a(*)
    integer, value :: s
    integer :: i
    i = (blockDim%x*(blockIdx%x-1)+threadIdx%x) * s
    a(i) = a(i)+1
end subroutine stride
```

Array `a()` allocated with padding to avoid out-of-bounds accesses
Strided Data Access
Shared Memory

• On-chip
• All threads in a block have access to same shared memory
• Used to reduce multiple loads of device data
• Used to accommodate coalescing
Matrix Transpose

attributes(global) subroutine transposeNaive(odata, idata)
  real, intent(out) :: odata(ny,nx)
  real, intent(in) :: idata(nx,ny)
  integer :: x, y

  x = (blockIdx%x-1) * blockDim%x + threadIdx%x
  y = (blockIdx%y-1) * blockDim%y + threadIdx%y
  odata(y,x) = idata(x,y)
end subroutine transposeNaive

idata  odata
Matrix Transpose - Shared Memory

attributes(global) subroutine transposeCoalesced(odata, idata)
  real, intent(out) :: odata(ny,nx)
  real, intent(in) :: idata(nx,ny)
  real, shared :: tile(TILE_DIM, TILE_DIM)
  integer :: x, y

  x = (blockIdx%x-1)*blockDim%x + threadIdx%x
  y = (blockIdx%y-1)*blockDim%y + threadIdx%y
  tile(threadIdx%x, threadIdx%y) = idata(x,y)

  call syncthreads()

  x = (blockIdx%y-1)*blockDim%y + threadIdx%y
  y = (blockIdx%x-1)*blockDim%x + threadIdx%x
  odata(x,y) = tile(threadIdx%y, threadIdx%x)
end subroutine transposeCoalesced
Shared Memory Bank Conflicts

- Shared memory is divided into banks that can be accessed simultaneously.
- Successive 32-bit words are assigned to successive banks.
- Requests for different data in the same bank by threads in a warp (SIMD unit of 32 successive threads) are serialized.
- Common workaround is to pad shared memory arrays.

```plaintext
real, shared :: tile(TILE_DIM +1, TILE_DIM)
```
Textures

- Different pathway for accessing data in device DRAM
- Read-only by device code
- Cached on chip in texture cache
- Uses F90 pointer notation
  - Declare texture at module scope in module where used
    real, texture, pointer :: aTex(:)
  - Bind and unbind texture in host code using pointer notation
- Equivalent to tex1Dfetch() in CUDA C
  - No filtering or wrapping modes
- 12.8 and later compilers
module mymodule
  real, texture, pointer :: aTex(:)
contains
  attributes(global) subroutine inc(b)
    real :: b(*)
    integer :: i
    i = blockDim%x*(blockIdx%x-1) + threadIdx%x
    b(i) = aTex(i)+1
  end subroutine
end module
Textures - Host Code

program tex
  use mymodule
  ...
  real, device, target :: a_d(n)
  real, device :: b_d(n)

  ! bind texture
  aTex => a_d

  call inc<<<n/tBlock, tBlock>>>(b_d)
  ...

  ! unbind texture
  nullify(aTex)
end program
Outline

• Introduction
• Performance Optimization
  - Host-Device Data Transfers
  - Device Memory
  - Execution Configuration
• Examples
• Multi-GPU Programming
Execution Configuration

• GPUs are high latency, 100s of cycles per device memory request
• For good performance, you need to ensure there is enough parallelism to hide this latency
• Such parallelism can come from:
  - Thread-level parallelism
  - Instruction-level parallelism
Thread-Level Parallelism

• Execution configuration dictates number of threads per block
  - Limit on number of threads per block for each architecture

• Number of concurrent blocks on a multiprocessor limited by
  - Register use per thread
  - Shared memory use per thread block
  - Limit on number of threads per multiprocessor

• Occupancy
  - Ratio of actual to maximum number of concurrent threads per multiprocessor
Thread-Level Parallelism

attributes(global) subroutine copy(odata, idata)
real :: odata(*), idata(*), tmp
integer :: i
i = (blockIdx%x-1)*blockDim%x + threadIdx%x
tmp = idata(i)
odata(i) = tmp
end subroutine copy
Thread-Level Parallelism

- Run on K20
  - Maximum of 2048 concurrent threads per multiprocessor
  - Maximum of 16 concurrent blocks per multiprocessor
  - Maximum of 1024 threads per block

<table>
<thead>
<tr>
<th>Thread Block Size</th>
<th>Occupancy</th>
<th>Bandwidth (GB/s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>32</td>
<td>0.25</td>
<td>96</td>
</tr>
<tr>
<td>64</td>
<td>0.5</td>
<td>125</td>
</tr>
<tr>
<td>128</td>
<td>1.0</td>
<td>136</td>
</tr>
<tr>
<td>256</td>
<td>1.0</td>
<td>137</td>
</tr>
<tr>
<td>512</td>
<td>1.0</td>
<td>137</td>
</tr>
<tr>
<td>1024</td>
<td>1.0</td>
<td>133</td>
</tr>
</tbody>
</table>
Thread-Level Parallelism

- Mimic high resource use
  - Specify enough shared memory so only one thread block can reside on a multiprocessor at a time

```cpp
    call copy<<<grid, threadBlock, 0.9*smBytes>>>(b_d, a_d)
```

<table>
<thead>
<tr>
<th>Thread Block</th>
<th>No Shared Memory</th>
<th>Shared Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Occupancy  Bandwidth</td>
<td>Occupancy  Bandwidth</td>
</tr>
<tr>
<td>32</td>
<td>0.25  96</td>
<td>0.016  8</td>
</tr>
<tr>
<td>64</td>
<td>0.5   125</td>
<td>0.031  15</td>
</tr>
<tr>
<td>128</td>
<td>1.0   136</td>
<td>0.063  29</td>
</tr>
<tr>
<td>256</td>
<td>1.0   137</td>
<td>0.125  53</td>
</tr>
<tr>
<td>512</td>
<td>1.0   137</td>
<td>0.25   91</td>
</tr>
<tr>
<td>1024</td>
<td>1.0   133</td>
<td>0.5    123</td>
</tr>
</tbody>
</table>
Instruction-Level Parallelism

- Have each thread process multiple elements

```fortran
subroutine copy_ILP(odata, idata)
  real :: odata(*), idata(*), tmp(ILP)
  integer :: i,j

  i = (blockIdx%x-1)*blockDim%x*ILP + threadIdx%x
  do j = 1, ILP
    tmp(j) = idata(i+(j-1)*blockDim%x)
  enddo
  do j = 1, ILP
    odata(i+(j-1)*blockDim%x) = tmp(j)
  enddo
end subroutine copy_ILP
```
## Instruction-Level Parallelism

<table>
<thead>
<tr>
<th>Thread Block Size</th>
<th>No Shared Memory</th>
<th>Shared Memory</th>
<th>Bandwidth No ILP</th>
<th>Bandwidth ILP = 4</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Occupancy</td>
<td>Bandwidth</td>
<td>Occupancy</td>
<td>Bandwidth No ILP</td>
</tr>
<tr>
<td>32</td>
<td>0.25</td>
<td>96</td>
<td>0.016</td>
<td>8</td>
</tr>
<tr>
<td>64</td>
<td>0.5</td>
<td>125</td>
<td>0.031</td>
<td>15</td>
</tr>
<tr>
<td>128</td>
<td>1.0</td>
<td>136</td>
<td>0.063</td>
<td>29</td>
</tr>
<tr>
<td>256</td>
<td>1.0</td>
<td>137</td>
<td>0.125</td>
<td>53</td>
</tr>
<tr>
<td>512</td>
<td>1.0</td>
<td>137</td>
<td>0.25</td>
<td>91</td>
</tr>
<tr>
<td>1024</td>
<td>1.0</td>
<td>133</td>
<td>0.5</td>
<td>123</td>
</tr>
</tbody>
</table>
Outline

• Introduction
• Performance Optimization
• Examples
  - Using CUBLAS and Thrust from CUDA Fortran
  - Convolution
  - Computing \( \pi \) using the Monte Carlo Method
• Multi-GPU Programming
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

```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) ! or sgemm(..)
  c=c_d
  write(*,*) 'Maximum error: ', maxval(abs(c-14.0))
  deallocate (a,b,c,a_d,b_d,c_d)
end program cublasTest
```
Calling Thrust from CUDA Fortran

C wrapper for Thrust: csort.cu

```c
#include <thrust/device_vector.h>
#include <thrust/device_vector.h>
#include <thrust/sort.h>

extern "C" {
    //Sort for integer arrays
    void sort_int_wrapper( int *data, int N)
    {
        // Wrap raw pointer with a device_ptr
        thrust::device_ptr<int> dev_ptr(data);
        // Use device_ptr in Thrust sort
        // algorithm
        thrust::sort(dev_ptr, dev_ptr+N);
    }

    //Sort for single precision arrays
    void sort_float_wrapper( float *data, int N)
    {
        thrust::device_ptr<float> dev_ptr(data);
        thrust::sort(dev_ptr, dev_ptr+N);
    }

    //Sort for double precision arrays
    void sort_double_wrapper(double *data, int N)
    {
        thrust::device_ptr<double> dev_ptr(data);
        thrust::sort(dev_ptr, dev_ptr+N);
    }
}
```
Calling Thrust from CUDA Fortran

Fortran interface to C wrapper using ISO C Bindings

module thrust

  interface thrustsort
    subroutine sort_int( input,N) &
      bind(C,name="sort_int_wrapper")
    use iso_c_binding
    integer(c_int),device:: input(*)
    integer(c_int),value:: N
  end subroutine

  subroutine sort_double( input,N) &
    bind(C,name="sort_double_wrapper")
    use iso_c_binding
    real(c_double),device:: input(*)
    integer(c_int),value:: N
  end subroutine

end interface

end module
CUDA Fortran Sorting with Thrust

program testsort
  use thrust
  real, allocatable :: cpuData(:)
  real, allocatable, device :: gpuData(:)
  integer :: N=10

  !Allocate CPU and GPU arrays
  allocate(cpuData(N),gpuData(N))
  !Fill the host array with random data
  do i=1,N
    cpuData(i)=random(i)
  end do

  !Print unsorted data
  print *, cpuData
  !Send data to GPU
  gpuData = cpuData
  !Sort the data
  call thrustsort(gpuData,N)
  !Copy the result back
  cpuData = gpuData
  !Print sorted data
  print *, cpuData
  !Deallocate arrays
  deallocate(cpuData,gpuData)
end program testsort

cuda Fortran Sorting with Thrust

Before sorting 4.1630346E-02 0.9124327 0.7832350 0.6540373 100.0000 0.3956419 0.2664442 0.1372465 8.0488138E-03 0.8788511
After sorting 8.0488138E-03 4.1630346E-02 0.1372465 0.2664442 0.3956419 0.6540373 0.7832350 0.8788511 0.9124327 100.0000

nvcc -c -arch sm_20 csort.cu
pgf90  -Mcuda=cc20 -O3 -o testsort thrust_module.cuf testsort.cuf csort.o

$ ./testsort
Before sorting 4.1630346E-02 0.9124327 0.7832350 0.6540373 100.0000 0.3956419 0.2664442 0.1372465 8.0488138E-03 0.8788511
After sorting 8.0488138E-03 4.1630346E-02 0.1372465 0.2664442 0.3956419 0.6540373 0.7832350 0.8788511 0.9124327 100.0000
CUDA Fortran Sorting with Thrust

program timesort
    use cudafor
    use thrust
    real, allocatable :: cpuData(:)
    real, allocatable, device :: gpuData(:)
    integer:: N=100000000, istat
    ! cuda events for elapsing
    type (cudaEvent) :: startEvent, stopEvent
    real :: time, random
    !Allocate CPU and GPU arrays
    allocate(cpuData(N), gpuData(N))
    !Fill the host array with random data
    do i=1,N
        cpuData(i)=random(i)
    end do
    ! Create events
    istat = cudaEventCreate ( startEvent )
    istat = cudaEventCreate ( stopEvent )
    ! Send data to GPU
    gpuData = cpuData
    ! Sort the data
    istat = cudaEventRecord ( startEvent , 0)
    call thrustsort(gpuData,N)
    istat = cudaEventRecord ( stopEvent , 0)
    istat = cudaEventSynchronize ( stopEvent )
    istat = cudaEventElapsedTime( time, &
        startEvent ,
        stopEvent)
    ! Copy the result back
    cpuData = gpuData
    print *,"Sorted array in: ",time," (ms)"
    print *, "After sorting", &
    cpuData(1:5),cpuData(N-4:N)
! Deallocate arrays
    deallocate(cpuData,gpuData)
end program timesort

$ ./timesort
Sorting array of 100000000 single precision
Sorted array in: 194.6642 (ms)
After sorting 7.0585919E-09 1.0318221E-08 1.9398616E-08 3.1738640E-08 0.9999999 0.9999999 1.000000 1.000000
Convolution Example

Perform convolution of a stack of 2D arrays in frequency space
1. Send arrays to GPU memory
2. Transform arrays to frequency domain using 2D FFT from CUFFT
3. Perform point-wise complex multiplication and scaling \((FFT(\text{IFFT}(A))=\text{len}(A)*A)\)
4. Transform result back to physical domain
5. Send result back to CPU memory
Naive approach

Transfer A  Transfer B  FFT(A)  FFT(B)  conv  IFFT(C)  Transfer C

Smarter approach

I/O
Transfer A  Transfer B  Transfer C

Compute
FFT(A)  FFT(B)  conv  IFFT(C)

Optimal approach

H2D
A(1)  B(1)  A(2)  B(2)  · · · · · ·  A(N)  B(N)

Compute
FFT A(1)  FFT B(1)  conv  IFFT C(1)  FFT A(2)  FFT B(2)  conv  IFFT C(2)  · · ·  FFT A(N)  FFT B(N)  conv  IFFT C(N)

D2H
C(1)  C(2)  · · ·  C(N)

NOT TO SCALE

0.3s  time

0.18s  time
program driver
use cudafor
use cufft
implicit none
complex, allocatable, dimension(:,:,), pinned :: A, B
complex, allocatable, dimension(:,:,), device :: A_d, B_d

real:: elapsed_time,scale
integer, parameter :: num_streams=8
integer:: nxx, nyy, nomega, plan, stream(num_streams)
integer :: clock_start,clock_end,clock_rate, istat, ifr, i, j, ind
logical:: plog

nxx=512; nyy=512; nomega=196

! Initialize FFT plan
call cufftPlan2d(plan,nyy,nxx,CUFFT_C2C)
do i = 1,num_streams
   istat= cudaStreamCreate(stream(i))
end do

! Find the clock rate
call SYSTEM_CLOCK(COUNT_RATE=clock_rate)
allocate(A(nxx,nyy,nomega),B(nxx,nyy,nomega))
allocate(A_d(nxx,nyy,num_streams),B_d(nxx,nyy,num_streams))

istat=cudaThreadSynchronize()
call SYSTEM_CLOCK(COUNT=clock_start) ! Start timing
scale =1./(nxx*nyy)
do ifr=1,nomega
   ind = mod(ifr,num_streams)+1

! Send data to GPU
istat= cudaMemcpyAsync(A_d(1,1,ind),A(1,1,ifr),nxx*nyy, stream(ind))
istat= cudaMemcpyAsync(B_d(1,1,ind),B(1,1,ifr),nxx*nyy, stream(ind))

! Execute FFTs on GPU
call cufftSetStream(plan,stream(ind))
call cufftExecC2C(plan ,A_d(1,1,ind),A_d(1,1,ind),CUFFT_FORWARD)
call cufftExecC2C(plan ,B_d(1,1,ind),B_d(1,1,ind),CUFFT_FORWARD)

! Convolution and scaling of the arrays
!$cuf kernel do(2) <<<*,*,stream=stream(ind)>>>
do j=1,nyy
   do i=1,nxx
      B_d(i,j,ind)= A_d(i,j,ind)*B_d(i,j,ind)*scale
   end do
end do

! Execute FFTs on GPU
call cufftExecC2C(plan ,B_d(1,1,ind),B_d(1,1,ind),CUFFT_INVERSE)

! Copy results back
istat= cudaMemcpyAsync( B(1,1,ifr),B_d(1,1,ind),nxx*nyy, stream=stream(ind))
end do
istat=cudaThreadSynchronize()
call SYSTEM_CLOCK(COUNT=clock_end) ! End timing
print *,"Elapsed time.", REAL(clock_end-clock_start)/REAL(clock_rate)
deallocate(A,B,A_d,B_d)
end program
Computing $\pi$ with CUDA Fortran

$$\pi = 4 \times \frac{(\Sigma \text{red points})}{(\Sigma \text{points})}$$

Simple example:
- Generate random numbers (CURAND)
- Compute sum using of kernel loop directive
- Compute sum using two stages reduction with Cuda Fortran kernels
- Compute sum using single stage reduction with Cuda Fortran kernel
- Accuracy
CUDA Libraries from CUDA Fortran

- All the toolkit libraries have C interfaces
- Use F90 interfaces and ISO C Binding to use from CUDA Fortran

```fortran
interface curandGenerateUniform
  !curandGenerateUniform(curandGenerator_t generator, float *outputPtr, size_t num);
  subroutine curandGenerateUniform(generator, odata, numele) &
    bind(C,name='curandGenerateUniform')
    use iso_c_binding
    integer(c_size_t),value:: generator
    !pgi$ ignore_tr odata
    real(c_float), device:: odata(*)
    integer(c_size_t),value:: numele
  end subroutine curandGenerateUniform
end interface curandGenerateUniform

!curandGenerateUniformDouble(curandGenerator_t generator, double *outputPtr, size_t num);
subroutine curandGenerateUniformDouble(generator, odata, numele) &
  bind(C,name='curandGenerateUniformDouble')
  use iso_c_binding
  integer(c_size_t),value:: generator
  !pgi$ ignore_tr odata
  real(c_double), device:: odata(*)
  integer(c_size_t),value:: numele
end subroutine curandGenerateUniformDouble
end subroutine curandGenerateUniformDouble
```
Computing $\pi$ with CUF Kernel

! Compute pi using a Monte Carlo method
program compute_pi
use precision
use cudafor ! CUDA Fortran runtime
use curand ! CURAND interface
implicit none
real(fp_kind), allocatable, pinned:: hostData(:)
real(fp_kind), allocatable, device:: deviceData(:)
real(fp_kind):: pival
integer :: inside_cpu,inside, i, iter, Nhalf
integer(kind=8) :: gen, N, seed=1234

! Define how many numbers we want to generate
N=2000
Nhalf=N/2
! Allocate arrays on CPU and GPU
allocate(hostData(N), deviceData(N))
! Create pseudonumber generator
call curandCreateGenerator(gen, &
CURAND_RNG_PSEUDO_DEFAULT)
! Set seed
call curandSetPseudoRandomGeneratorSeed( gen, seed)
! Generate N floats or double on device
call curandGenerateUniform(gen, deviceData, N)
! Copy the data back to CPU to check result later
hostData=deviceData

! Perform the test on GPU using CUF kernel
inside=0
!$cuf kernel do <<<*,*>>>
do i=1,Nhalf
  if( (deviceData(i)**2+deviceData(i+Nhalf)**2) &
    <= 1._fp_kind ) inside=inside+1
end do
! Perform the test on CPU
inside_cpu=0
do i=1,Nhalf
  if( (hostData(i)**2+hostData(i+Nhalf)**2) &
    <= 1._fp_kind ) inside_cpu=inside_cpu+1
end do
! Check the results
if (inside_cpu .ne. inside) &
  print *,"Mismatch between CPU/ &
  GPU",inside_cpu,inside
! Print the value of pi and the error
pival= 4._fp_kind*real(inside,fp_kind) &
/real(Nhalf,fp_kind)
print"(t3,a,i10,a,f10.8,a,e11.4)", "Samples=", &
Nhalf," Pi=", pival," Error=" &
abs(pival-2.0_fp_kind*asin(1.0_fp_kind))
! Deallocate data on CPU and GPU
deallocate(hostData,deviceData)
! Destroy the generator
call curandDestroyGenerator(gen)
end program compute_pi
Computing $\pi$

pgf90 -O3 -Mpreprocess -o pi_gpu precision_module.cuf curand_module.cuf pi.cuf -lcurand

Where is the difference coming from?

if( ( hostData(i)**2 + hostData(i+Nhalf)**2) <= 1._fp_kind) inside_cpu=inside_cpu+1  (CPU)
if( (deviceData(i)**2+deviceData(i+Nhalf)**2) <= 1._fp_kind ) inside=inside+1                (GPU)

- Sum of the point inside the circle is done with integers (no issues due to floating point arithmetic)
- Computation of the distance from the origin ($x^2+y^2$), no special functions just $+$ and $*$
GPU Accuracy

- FERMI GPUs are IEEE-754 compliant, both for single and double precision
- Support for Fused Multiply-Add instruction (IEEE 754-2008)
- Results with FMA could be different* from results without FMA
- In CUDA Fortran is possible to toggle FMA on/off with a compiler switch:
  
  - Mcuda=nofma

- Extremely useful to compare results to “golden” CPU output
- FMA is being supported in future CPUs

<table>
<thead>
<tr>
<th>Compute pi in single precision (seed=1234567 FMA disabled)</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Samples= 10000        Pi=3.16720009  Error= 0.2561E-01</td>
<td></td>
</tr>
<tr>
<td>Samples= 100000       Pi=3.13919997  Error= 0.2393E-02</td>
<td></td>
</tr>
<tr>
<td>Samples= 1000000      Pi=3.14109206  Error= 0.5007E-03</td>
<td></td>
</tr>
<tr>
<td>Samples= 10000000     Pi=3.14106607  Error= 0.5267E-03</td>
<td></td>
</tr>
<tr>
<td>Samples= 100000000    Pi=3.14139462  Error= 0.1981E-03</td>
<td></td>
</tr>
</tbody>
</table>

*GPU results with FMA are identical to CPU if operations are done in double precision
Reductions on GPU

- Parallelism across blocks
- Parallelism within a block
- No global synchronization
  - two-stage approach (two kernel launches), same code for both stages

Level 0: 8 blocks
Level 1: 1 block
Parallel Reduction: Sequential Addressing

**Step 1: Stride 8**

<table>
<thead>
<tr>
<th>Thread IDs</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>10</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>8</td>
</tr>
<tr>
<td>4</td>
<td>-1</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>-2</td>
</tr>
<tr>
<td>7</td>
<td>3</td>
</tr>
<tr>
<td>8</td>
<td>5</td>
</tr>
<tr>
<td>9</td>
<td>-2</td>
</tr>
<tr>
<td>10</td>
<td>2</td>
</tr>
<tr>
<td>11</td>
<td>7</td>
</tr>
<tr>
<td>12</td>
<td>0</td>
</tr>
<tr>
<td>13</td>
<td>11</td>
</tr>
<tr>
<td>14</td>
<td>0</td>
</tr>
<tr>
<td>15</td>
<td>2</td>
</tr>
<tr>
<td>16</td>
<td>0</td>
</tr>
<tr>
<td>17</td>
<td>2</td>
</tr>
</tbody>
</table>

**Step 2: Stride 4**

<table>
<thead>
<tr>
<th>Thread IDs</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>8</td>
</tr>
<tr>
<td>2</td>
<td>-2</td>
</tr>
<tr>
<td>3</td>
<td>10</td>
</tr>
<tr>
<td>4</td>
<td>6</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>9</td>
</tr>
<tr>
<td>7</td>
<td>3</td>
</tr>
<tr>
<td>8</td>
<td>7</td>
</tr>
<tr>
<td>9</td>
<td>-2</td>
</tr>
<tr>
<td>10</td>
<td>-3</td>
</tr>
<tr>
<td>11</td>
<td>2</td>
</tr>
<tr>
<td>12</td>
<td>7</td>
</tr>
<tr>
<td>13</td>
<td>0</td>
</tr>
<tr>
<td>14</td>
<td>11</td>
</tr>
<tr>
<td>15</td>
<td>0</td>
</tr>
<tr>
<td>16</td>
<td>2</td>
</tr>
<tr>
<td>17</td>
<td>0</td>
</tr>
<tr>
<td>18</td>
<td>2</td>
</tr>
</tbody>
</table>

**Step 3: Stride 2**

<table>
<thead>
<tr>
<th>Thread IDs</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>8</td>
</tr>
<tr>
<td>2</td>
<td>7</td>
</tr>
<tr>
<td>3</td>
<td>13</td>
</tr>
<tr>
<td>4</td>
<td>13</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>9</td>
</tr>
<tr>
<td>7</td>
<td>3</td>
</tr>
<tr>
<td>8</td>
<td>7</td>
</tr>
<tr>
<td>9</td>
<td>-2</td>
</tr>
<tr>
<td>10</td>
<td>-3</td>
</tr>
<tr>
<td>11</td>
<td>2</td>
</tr>
<tr>
<td>12</td>
<td>7</td>
</tr>
<tr>
<td>13</td>
<td>0</td>
</tr>
<tr>
<td>14</td>
<td>11</td>
</tr>
<tr>
<td>15</td>
<td>0</td>
</tr>
<tr>
<td>16</td>
<td>2</td>
</tr>
<tr>
<td>17</td>
<td>0</td>
</tr>
</tbody>
</table>

**Step 4: Stride 1**

<table>
<thead>
<tr>
<th>Thread IDs</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>8</td>
</tr>
<tr>
<td>2</td>
<td>7</td>
</tr>
<tr>
<td>3</td>
<td>13</td>
</tr>
<tr>
<td>4</td>
<td>13</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>9</td>
</tr>
<tr>
<td>7</td>
<td>3</td>
</tr>
<tr>
<td>8</td>
<td>7</td>
</tr>
<tr>
<td>9</td>
<td>-2</td>
</tr>
<tr>
<td>10</td>
<td>-3</td>
</tr>
<tr>
<td>11</td>
<td>2</td>
</tr>
<tr>
<td>12</td>
<td>7</td>
</tr>
<tr>
<td>13</td>
<td>0</td>
</tr>
<tr>
<td>14</td>
<td>11</td>
</tr>
<tr>
<td>15</td>
<td>0</td>
</tr>
<tr>
<td>16</td>
<td>2</td>
</tr>
<tr>
<td>17</td>
<td>0</td>
</tr>
</tbody>
</table>

**Values**:

- Step 1: 10 1 8 -1 0 -2 3 5 -2 -3 2 7 0 11 0 2
- Step 2: 8 -2 10 6 0 9 3 7 -2 -3 2 7 0 11 0 2
- Step 3: 8 7 13 13 0 9 3 7 -2 -3 2 7 0 11 0 2
- Step 4: 21 20 13 13 0 9 3 7 -2 -3 2 7 0 11 0 2

**Thread IDs**:

- Step 1: 1 2 3 4 5 6 7 8
- Step 2: 1 2 3 4
- Step 3: 1 2
- Step 4: 1
Computing $\pi$ with CUDA Fortran Kernels

```fortran
attributes(global) subroutine partial_sum(input,partial,N)
  real(fp_kind) :: input(N)
  integer :: partial(256)
  integer, shared, dimension(256) :: psum
  integer(kind=8),value :: N
  integer :: i,index, inext,interior

  index=threadIdx%x+(BlockIdx%x-1)*BlockDim%x
  ! Check if the point is inside the circle
  ! and increment local counter
  interior=0
  do i=index,N/2,BlockDim%x*GridDim%x
    if( (input(i)**2+input(i+N/2)**2) <= 1._fp_kind) &
      interior=interior+1
  end do

  ! Local reduction per block
  index=threadIdx%x
  psum(index)=interior
  call syncthreads()
  inext=blockDim%x/2
  do while ( inext >=1 )
    if (index <=inext) psum(index) = &
      psum(index) + psum(index+inext)
    inext = inext /2
    call syncthreads()
  end do

  ! Each block writes back its partial sum
  if (index == 1) partial(BlockIdx%x)=psum(1)
end subroutine

! Compute the partial sums with 256 blocks of 512 threads
! Compute the final sum with 1 block of 256 threads
```
Computing π with CUDA Fortran Kernels

```fortran
attributes(global) subroutine final_sum(partial,nthreads,total)
  integer, intent(in) :: partial(nthreads)
  integer, intent(out) :: total
  integer, shared :: psum(*)
  integer :: index, inext

  index=threadIdx%x

  ! load partial sums in shared memory
  psum(index)=partial(index)
  call syncthreads()

  inext=blockDim%x/2
  do while ( inext >=1 )
    if (index <=inext) psum(index)=psum(index)+psum(index+inext)
    inext = inext /2
    call syncthreads()
  end do

  ! First thread has the total sum, writes it back to global memory
  if (index == 1) total=psum(1)
end subroutine
```
Computing \( \pi \) with an Atomic Lock

Instead of storing back the partial sum:

! Each block writes back its partial sum
if (index == 1) partial(BlockIdx%x)=psum(1)

use an atomic lock to ensure that one block at the time updates the final sum:

if (index == 1) then
    do while ( atomiccas(lock,0,1) == 1) ! set lock
        end do
    partial(1)=partial(1)+psum(1) ! atomic update of partial(1)
    call threadfence() ! Wait for memory transaction to be visible to all the other threads
    lock =0 ! release lock
end if

partial(1)=0
call sum<<<64,256,256*4>>>(deviceData,partial,N)
inside=partial(1)
Outline

- Introduction
- Performance Optimization
- Examples
- Multi-GPU Programming
Multi-GPU Programming

• Multi-GPU configurations
  - Number of systems or nodes
  - Number of host threads per node
  - Number of GPUs per node

• For this tutorial we focus on a single host thread using multiple GPUs
Device Management API

- **cudaGetDeviceCount(nDevices)**
  - Returns number of CUDA-enabled devices attached to system
  - Devices are enumerated from 0 to nDevices-1

- **cudaGetDeviceProperties(prop, dev)**
  - Inquiry function for device dev
  - prop is of type cudaDeviceProp

- **cudaSetDevice(dev)**
  - Sets the current device to dev
  - All device operations are associated with current device
Multi-GPU Memory Allocation

- Allocation at time of declaration is on default device (0)

  ```
  real, device :: a_d(N)
  ```

- Allocatable arrays used for data on other devices

  ```
  real, device, allocatable :: a_d(:), b_d(:)
  istat = cudaSetDevice(0)
  allocate(a_d(N))
  istat = cudaSetDevice(1)
  allocate(b_d(N))
  ```
module kernel
contains
  attributes(global) subroutine assign(a, v)
    implicit none
    real :: a(*)
    real, value :: v
    a(threadIdx%x) = v
  end subroutine assign
end module kernel

program minimal
use cudafor
use kernel
implicit none
integer, parameter :: n=32
real :: a(n)
real, device, allocatable :: a0_d(:), a1_d(:)
integer :: nDevices, istat
istat = cudaGetDeviceCount(nDevices)
if (nDevices < 2) then
  write(*,*) 'Requires >= 2 GPUs'
  stop
end if
istat = cudaSetDevice(0)
allocate(a0_d(n))
call assign<<<1,n>>>(a0_d, 3.0)
a = a0_d
deallocate(a0_d)
write(*,*) 'Device 0: ', a(1)
istat = cudaSetDevice(1)
allocate(a1_d(n))
call assign<<<1,n>>>(a1_d, 4.0)
a = a1_d
deallocate(a1_d)
write(*,*) 'Device 1: ', a(1)
end program minimal
Unified Virtual Addressing

- CPU and GPUs use a unified virtual address space
  - Device can be determined from address of data
  - Requires: CUDA Toolkit $\geq 4.0$, compute capability $\geq 2.0$, 64-bit OS

- Enables peer-to-peer computing
  - Direct Transfers: copy between GPUs not staged through host
  - Direct Access: a kernel on one GPU can access memory on another
Enabling Peer-to-Peer Computing

- `cudaDeviceCanAccessPeer(accessible, devA, devB)`
  - Sets `accessible` to 1 if device `devA` can access `devB`’s memory
  - Access not available when
    - `devA` or `devB` have compute capability < 2.0
    - `devA` and `devB` are connected to different Intel IOH chips on motherboard

- `cudaDeviceEnablePeerAccess(dev, 0)`
  - Enables the current device to access memory on `dev`
Direct Transfers

• Once direct access is enabled, transfers between devices can be performed by
  – `dstArray = srcArray`
    • `dstArray` and `srcArray` are on different devices
  – `cudaMemcpy(dstArray, srcArray, N)`
    • Other variants (eg. `cudaMemcpyAsync`, `cudaMemcpy2D`) also work

• `cudaMemcpyPeer(dstArray, dstDev, srcArray, srcDev, N)`
  - Explicitly specify source and destination devices
  - If direct access is not enabled, the transfer is staged through host, otherwise direct transfer is used
Direct Transfer Example
module kernel
contains
  attributes(global) subroutine inc(a, v)
  implicit none
  real :: a(*)
  real, value :: v
  a(threadIdx%x) = a(threadIdx%x)+v
end subroutine inc
end module kernel
program minimal
  use cudafor
  use kernel
  implicit none
  integer, parameter :: n=32
  real :: a(n)
  real, device, allocatable :: a0_d(:), a1_d(:)
  integer :: nDevices, istat, access

  istat = cudaGetDeviceCount(nDevices)
  if (nDevices < 2) then
    write(*,*) 'Requires at least two GPUs'
    stop
  end if

  istat = cudaDeviceCanAccessPeer(access, 1, 0)
  if (access /= 1) then
    write(*,*) 'Peer access not available'
    stop
  end if
istat = cudaSetDevice(1)
istat = cudaDeviceEnablePeerAccess(0, 0)

istat = cudaSetDevice(0)
allocation(a0_d(n))
istat = cudaSetDevice(1)
allocation(a1_d(n))

istat = cudaSetDevice(0)
a = 0.0; a0_d = a
call inc<<<1,n>>>(a0_d, 1.0)

istat = cudaSetDevice(1)
a1_d = a0_d

call inc<<<1,n>>>(a1_d, 1.0)
a = a1_d
write(*,*) 'max error = ', maxval(abs(a-2))
deallocate(a1_d)
istat = cudaSetDevice(0)
deallocate(a0_d)
end program minimal
Introduction to CUDA Fortran