Tutorial on Programming GPUs

Viktor K. Decyk
UCLA

Abstract

The new NVIDIA Fermi GPU architecture has hardware for cache and fast native atomic operations. These features allow one to obtain decent performance with less programming effort than before. A short tutorial on programming GPUs will be presented, making use of simple examples. We will discuss the important concepts of data coalescence, warp divergence, and tiling. Examples will be presented in both Cuda Fortran and Cuda C. If time permits, we will illustrate how these ideas were applied to a Particle-in-Cell code.

Outline of Presentation

- Abstraction of future computer hardware
- Vector copy example (segmenting problem, data coalescence)
- Transpose example (tiling with shared memory)
- Sum reduction example (device functions)

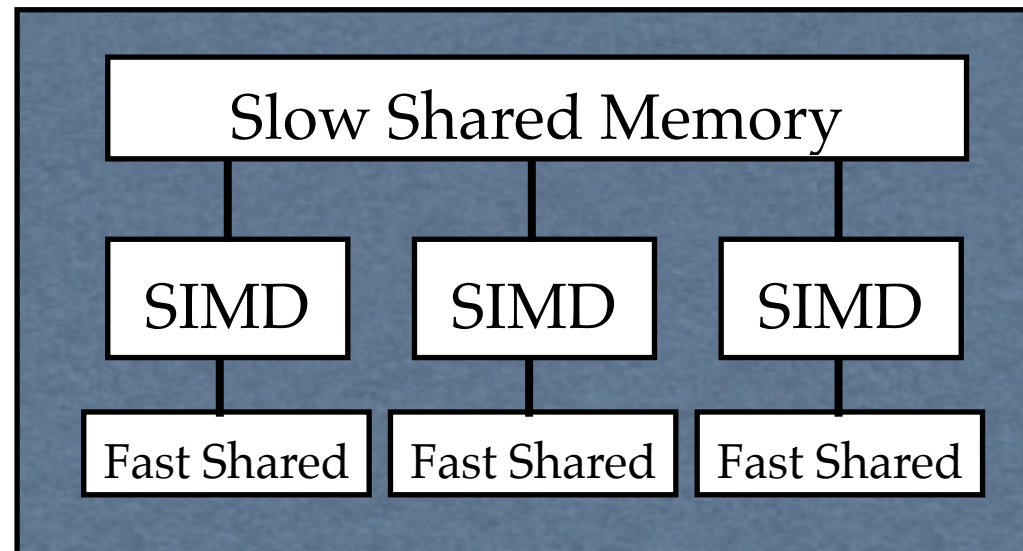All examples are in the file GPUTutorial.tar.gz
README file describes contents
There are 3 versions of each example in this file:
- Cuda C, with Fortran 2003 main program
- Cuda C, with C main program
- Cuda Fortran

Cuda Fortran will be used in this presentation

Simple abstraction of future hardware

```
┌─────────────────────────────────────────────┐
│   ┌───────────────────────────────────────┐  │
│   │          Slow Shared Memory           │  │
│   └───────────────────────────────────────┘  │
│        │              │             │         │
│   ┌─────────┐    ┌─────────┐   ┌─────────┐    │
│   │  SIMD   │    │  SIMD   │   │  SIMD   │    │
│   └─────────┘    └─────────┘   └─────────┘    │
│        │              │             │         │
│ ┌────────────┐ ┌────────────┐ ┌────────────┐ │
│ │Fast Shared │ │Fast Shared │ │Fast Shared │ │
│ └────────────┘ └────────────┘ └────────────┘ │
└─────────────────────────────────────────────┘
```

- SIMD (vector) unit has multiple cores, each executing the same instruction
- Cores work in lockstep with fast shared memory and local synchronization
- Multiple SIMD units coupled via "slow" shared memory and global synchronization

Each compute node is a powerful computer by itself
- A supercomputer is a hierarchy of such powerful computers

This abstraction applies to both GPUs as well as upcoming multi-core Intel processors
- Programming languages generally differ, however

GPUs are graphical processing units which consist of:
- 12-30 SIMD multiprocessors, each with small (16-48KB), fast (4 clocks) shared memory
- Each multi-processor contains 8-32 processor cores
- Large (0.5-6.0 GB), slow (400-600 clocks) global shared memory, readable by all units
- No cache on some units
- **Very fast (1 clock) hardware thread switching**

GPU Technology has two special features:
- High bandwidth access to global memory (>100 GBytes/sec)
- Ability to handle thousands of threads simultaneously, greatly reducing memory "stalls"

NVIDIA M2090 has 512 cores!

Programming GPUs

Programming Massively Parallel Processors: A Hands-on Approach
by David B. Kirk and Wen-mei W. Hwu [Morgan Kaufmann, 2010].

CUDA by Example: An Introduction to General-Purpose GPU Programming
by Jason Sanders and Edward Kandrot [Addison-Wesley, 2011].

CUDA Application Design and Development
by Rob Farber [Morgan Kaufmann, 2011].

CUDA Programming: A Developer's Guide to Parallel Computing with GPUs
by Shane Cook [Morgan Kaufmann, 2012].

The CUDA Handbook: A Comprehensive Guide to GPU Programming
by Nicholas Wilt [Addison-Wesley, 2013].

CUDA Fortran for Scientists and Engineers
by Gregory Ruetsch and Massimiliano Fatica [Morgan Kaufmann, 2014].

http://developer.nvidia.com/cuda-downloads

Programming GPUs: Example 1

Let's consider a very simple example: copying one 1D array to another
Each SIMD unit processes one block of the total problem

Original vector copy	Serial segmented 1d vector copy
Parallelized with OpenMP	Break up loops into blocks of size mx

```fortran
   subroutine copy0(a,b)
! simple 1d copy of length nx
! a = b
   implicit none
   real, dimension(:) :: a, b
   integer :: j, nx
   nx = min(size(a,1),size(b,1))

!$OMP PARALLEL DO PRIVATE(j)
   do j = 1, nx
      a(j) = b(j)
   enddo
!$OMP END PARALLEL DO

   end subroutine
```

```fortran
   subroutine copy1(a,b,mx)
! segmented 1d copy of length nx, with block size mx
! a = b
   implicit none
   integer :: mx
   real, dimension(:) :: a, b
   integer :: j, id, nx, nbx, joff

   nx = min(size(a,1),size(b,1))
   nbx = (nx - 1)/mx + 1

   do id = 1, nbx-1  ! outer loop over number of blocks
      joff = mx*id
      do j = 1, min(mx,nx-joff) ! loop over block
         a(j+joff) = b(j+joff)
      enddo
   enddo


   end subroutine
```

Stride 1 memory access important:

Read memory locations in order

Programming GPUs:
Copying one 1D array to another

Threads on each block (threadIdx) run on a single SIMD unit, execute the same instruction.

Different blocks (blockIdx) run on different SIMD units

```fortran
  subroutine copy1(a,b,mx)
! a = b
  implicit none
  integer :: mx
  real, dimension(:) :: a, b
  integer :: j, id, nx, nbx, joff

  nx = min(size(a,1),size(b,1))
  nbx = (nx - 1)/mx + 1

  do id = 0, nbx-1
     joff = mx*id
     do j = 1, min(mx,nx-joff)
        a(j+joff) = b(j+joff)
     enddo
  enddo

  end subroutine
```

The loop parameters set by host calling function:

```fortran
   subroutine gpu_copy1(a,b,mx)
! outer part of loop goes here
   implicit none
   integer :: mx, nx, nbx
   real, device, dimension(:) :: a, b
   type (dim3) :: dimBlock, dimGrid
   nx = min(size(a,1),size(b,1))
   nbx = (nx - 1)/mx + 1
   dimBlock = dim3(mx,1,1)    ! size of block
   dimGrid = dim3(nbx,1,1)    ! number of blocks

   call gcopy1<<<dimGrid,dimBlock>>>(a,b,nx)

   crc = cudaThreadSynchronize()
   end subroutine
```

The inner part of loop in GPU kernel function:

```fortran
   attributes(global) subroutine gcopy1(a,b,nx)
! inner part of loops goes here
   implicit none
   integer, value :: nx
   real, dimension(nx) :: a, b
   integer :: j, js, id, mx
   mx = blockDim%x      ! comes from dimBlock
   j = threadIdx%x      ! comes from dimBlock
   id = blockIdx%x      ! comes from dimGrid

   js = j + mx*(id - 1)
   if (js <= nx) a(js) = b(js)

   end subroutine
```

Programming GPUs: CUDA Fortran

In addition, you must initialize memory on the GPU

```fortran
  real, dimension(:), allocatable :: a, b
  real, device, dimension(:), allocatable :: g_a, g_b
! allocate host data
  allocate(a(nx),b(nx),c(nx))
! allocate data on GPU, using Fortran90 array syntax
  allocate(g_a(nx),g_b(nx))
```

Copy from host to GPU

```fortran
! Copy data to GPU, using Fortran90 array syntax
   g_b = b
```

Execute the subroutine:

```fortran
! Execute on GPU: g_a = g_b
     call gpu_copy1(g_a,g_b,mx)
```

Copy from GPU back to host

```fortran
! Copy data from GPU, using Fortran90 array syntax
   a = g_a
```

CUDA C is similar but more complex (no array syntax, separate memory spaces)

Programming GPUs:
Copying one **2D array** to another

Serial segmented 2d vector copy

```fortran
   subroutine copy2(a,b,mx)
! segmented 2d copy of length nx, ny
! with block size mx
! a = b
   implicit none
   integer :: mx
   real, dimension(:,:) :: a, b
   integer :: j, k, nx, ny, js, id, nbx

   nx = min(size(a,1),size(b,1))
   ny = min(size(a,2),size(b,2))
   nbx = (nx - 1)/mx + 1

   do k = 1, ny
      do id = 0, nbx-1
         joff = mx*id
         do j = 1, min(mx,nx-joff)
            a(j+joff,k) = b(j+joff,k)
         enddo
      enddo
   enddo

   end subroutine
```

In Fortran, first index is adjacent
in memory

Changes to host calling function:
Each y value is a separate block, nbx*ny blocks

```fortran
   subroutine gpu_copy2a(a,b,mx)
! outer part of loop goes here
   ...
   real, device, dimension(:,:) :: a, b
   ny = min(size(a,2),size(b,2))
   dimGrid = dim3(nbx,ny,1)


   call gcopy2a<<<dimGrid,dimBlock>>>(a,b,nx,ny)


   ...
   end subroutine
```

GPU kernel function:

```fortran
attributes(global) subroutine gcopy2a(a,b,nx,ny)
implicit none
integer, value :: nx, ny
real, dimension(nx,ny) :: a, b
integer :: j, k, js, id, mx
mx = blockDim%x
j = threadIdx%x
id = blockIdx%x
k = blockIdx%y


js = j + mx*(id - 1)
if ((js <= nx).and.(k <= ny)) a(js,k) = b(js,k)

end subroutine
```

Wednesday, May 27, 2015

Programming GPUs:
Copying one 2D array to another

Fewer blocks, more work per block

prior GPU kernel function:

```fortran
attributes(global) subroutine
                    gcopy2a(a,b,nx,ny)
implicit none
integer, value :: nx, ny
real, dimension(nx,ny) :: a, b
integer :: j, k, js, id, mx
mx = blockDim%x
j = threadIdx%x
id = blockIdx%x
k = blockIdx%y

js = j + mx*(id - 1)
if ((js <= nx).and.(k <= ny)) then
   a(js,k) = b(js,k)
endif

end subroutine
```

Changes to host calling function:
One block handles all x values

```fortran
   subroutine gpu_copy2b(a,b,mx)
! outer part of loop goes here
   ...
   dimGrid = dim3(ny,1,1)

   call gcopy2b<<<dimGrid,dimBlock>>>(a,b,nx,ny)

   ...
   end subroutine
```

GPU kernel function:

```fortran
attributes(global) subroutine gcopy2b(a,b,nx,ny)
implicit none
integer, value :: nx, ny
real, dimension(nx,ny) :: a, b
integer :: j, k, mx
mx = blockDim%x            ! id no longer needed
k = blockIdx%x

j = threadIdx%x
do while (j <= nx)
    if (k <= ny) a(j,k) = b(j,k)
    j = j + mx
enddo

end subroutine
```

Programming GPUs:
Copying one 2D array to another

Doubly segmented serial 2d vector copy

```fortran
  subroutine copy3(a,b,mx,my)
! segmented 2d copy of length nx, ny
! with block size mx, my
! a = b
  implicit none
  integer :: mx, my
  real, dimension(:,:) :: a, b
  integer :: j, k, nx, ny, joff, koff
  integer :: idx, idy, nbx, nby
  nx = min(size(a,1),size(b,1))
  ny = min(size(a,2),size(b,2))
  nbx = (nx - 1)/mx + 1
  nby = (ny - 1)/my + 1

  do idy = 0, nby-1
  koff = my*idy
  do idx = 0, nbx-1
     joff = mx*idx
     do k = 1, min(my,ny-koff)
        do j = 1, min(mx,nx-joff)
           a(j+joff,k+koff) =
           b(j+joff,k+koff)
        enddo
     enddo
  enddo
  enddo

  end subroutine
```

Changes to host calling function:
One block handles some x,y values

```fortran
   subroutine gpu_copy3(a,b,mx)
! outer part of loop goes here
   ...
   real, device, dimension(:,:) :: a, b
   dimBlock = dim3(mx,my,1)
   dimGrid = dim3(nbx,nby,1)

   call gcopy3<<<dimGrid,dimBlock>>>(a,b,nx,ny)

   ...
   end subroutine
```

GPU kernel function:

```fortran
attributes(global) subroutine gcopy3(a,b,nx,ny)
implicit none
integer, value :: nx, ny
real, dimension(nx,ny) :: a, b
integer :: j, k, js, ks, idx, idy, mx, my
mx = blockDim%x; my = blockDim%y
j = threadIdx%x; k = threadIdx%y
idx = blockIdx%x; idy = blockIdx%y

ks = k + my*(idy - 1)
js = j + mx*(idx - 1)
if ((js <= nx) .and. (ks <= ny)) then
    a(js,ks) = b(js,ks)
endif
end subroutine
```

Programming GPUs: Vector copy

Summary
- Processing must be segmented into independent blocks
- Inner loop runs on GPU, loop parameters are set on host
- Data coalescing is important (adjacent threads read adjacent memory locations)
- Each block should execute same instruction (avoid complex if statements)

Programming GPUs: CUDA Fortran

Sample output:

make cudaf

./fexample1

j=   0 :CUDA_DEVICE_NAME=Tesla M2090
    CUDA_MULTIPROCESSOR_COUNT=   16
    CUDA_GLOBAL_MEM_SIZE=   5636554752 (5.249451   GB)
    Capability=   20
 using device j=   0
 Fortran empty kernel time=   2.2300000E-04
 Fortran 1d copy time=   3.0000001E-06
 GPU 1d copy time=   7.4000003E-05
 1d copy maximum difference =   0.000000
 Fortran 2d copy time=   3.6940000E-03
 GPU 2d copy time=   4.3499999E-04
 2d copy maximum difference =   0.000000

 Homework: vary blocksize mx and data lengths nx, ny

Programming GPUs: Example 2

Transpose a 2D array:

First copy in to fast memory

Then copy out from fast memory

Copy to slow memory has stride 1

Original vector transpose

```fortran
  subroutine transpose0(a,b)
! simple 2d transpose of length nx,ny
! a = transpose(b)
  implicit none
  real, dimension(:,:) :: a, b
  integer :: j, k, nx, ny
  nx = min(size(a,2),size(b,1))
  ny = min(size(a,1),size(b,2))

  do k = 1, ny
     do j = 1, nx
         a(k,j) = b(j,k)
     enddo
  enddo

  end subroutine
```

In original, stride 1 memory access is obtained only when reading b array

```fortran
  subroutine transpose2(a,b,mx,my)
! segmented 2d transpose of length nx, ny
! with block size mx, my
! a = transpose(b)
  implicit none
  integer :: mx, my, j, k, nx, ny
  real, dimension(:,:) :: a, b
  integer :: idx, idy, joff, koff, nbx, nby
  real, dimension(mx+1,my) :: s
  nx = min(size(a,2),size(b,1))
  ny = min(size(a,1),size(b,2))
  nbx = (nx - 1)/mx + 1; nby = (ny - 1)/my + 1
  do idy = 0, nby-1
     koff = my*idy
     do idx = 0, nbx-1
         joff = mx*idx

         do k = 1, min(my,ny-koff)
            do j = 1, min(mx,nx-joff)
                s(j,k) = b(j+joff,k+koff)
            enddo
         enddo

         do j = 1, min(mx,nx-joff)
            do k = 1, min(my,ny-koff)
                a(k+koff,j+joff) = s(j,k)
            enddo
         enddo

     enddo
  enddo
  end subroutine
```

Programming GPUs: Transpose 2D array

On GPU, mx = my

```fortran
  subroutine transpose2(a,b,mx,my)
! a = transpose(b)
  ...
  real, dimension(mx+1,my) :: s
  ...
  do idy = 0, nby-1
     koff = my*idy
     do idx = 0, nbx-1
        joff = mx*idx
        do k = 1, min(my,ny-koff)
           do j = 1, min(mx,nx-joff)
              s(j,k) =
              b(j+joff,k+koff)
           enddo
        enddo
        do j = 1, min(mx,nx-joff)
           do k = 1, min(my,ny-koff)
              a(k+koff,j+joff) =
              s(j,k)
           enddo
        enddo
     enddo
  enddo
  end subroutine
```

The inner part of loop in GPU kernel function:

```fortran
        attributes(global) subroutine
                          gtranspose2(a,b,nx,ny)
! inner part of loops go here
     implicit none
     integer, value :: nx, ny
     real, dimension(ny,nx) :: a
     real, dimension(nx,ny) :: b
     integer :: j, k, js, ks, jb, kb
     integer :: joff, koff, mx, mxv
     real, shared, dimension(*) :: s
     mx = blockDim%x; mxv = mx + 1
     j = threadIdx%x; k = threadIdx%y
     idx = blockIdx%x; idy = blockIdx%y
     koff = mx*(idy - 1)
     joff = mx*(idx - 1)

     ks = k + koff
     js = j + joff
     if ((js <= nx) .and. (ks <= ny)) then
         s(j+mxv*(k-1)) = b(js,ks)
     endif


     call syncthreads() ! synchronize threads

     js = k + joff
     ks = j + koff
     if ((js <= nx) .and. (ks <= ny)) then
         a(ks,js) = s(k+mxv*(j-1))
     endif
     end subroutine
```

Programming GPUs: Transpose 2D array

GPU kernel function:

```fortran
attributes(global) subroutine
          gtranspose2(a,b,nx,ny)
...
real, shared, dimension(*) :: s
mx = blockDim%x; mxv = mx + 1
j = threadIdx%x; k = threadIdx%y
idx = blockIdx%x; idy = blockIdx%y
koff = mx*(idy - 1)
joff = mx*(idx - 1)

ks = k + koff
js = j + joff
if ((js <= nx).and.(ks <= ny)) then
   s(j+mxv*(k-1)) = b(js,ks)
endif


call syncthreads()


js = k + joff
ks = j + koff
if ((js <= nx).and.(ks <= ny)) then
   a(ks,js) = s(k+mxv*(j-1))
endif
end subroutine
```

Host calling function similar to gpu_copy3

```fortran
subroutine gpu_transpose2(a,b,mx)
...
real, device, dimension(:,:) :: a, b
...
dimBlock = dim3(mx,mx,1)
dimGrid = dim3(nbx,nby,1)
! calculate size of shared memory
ns = (mx + 1)*mx*sizeof(a(1,1))


call gtranspose2<<<dimGrid,dimBlock,ns>>>(a,b,nx,ny)


crc = cudaThreadSynchronize()


end subroutine
```

Programming GPUs: Transpose

Summary
- Stride 1 access is obtained to slow global memory
- Stride 1 access is not obtained to fast shared memory, but it is 100x faster
- Processing in pieces small enough to fit in fast memory (tiling) is important

Programming GPUs: CUDA Fortran

Sample output:

./fexample2

　using device j=　　　　0
　Fortran empty kernel time=　2.2800000E-04
　Fortran 2d transpose time=　4.2500001E-04
　GPU 2d transpose time=　2.5099999E-04
　2d transpose maximum difference =　0.000000

Programming GPUs: Example 3

Sum reduction of 1D array

Original sum reduction
Parallelized with OpenMP

```fortran
  subroutine sum0(a,sa)
! simple 1d sum reduction of length nx
! sa = sum(a)
  implicit none
  real :: sa
  real, dimension(:) :: a
  integer :: j

  sa = 0.0
!$OMP PARALLEL DO PRIVATE(j)
!$OMP& REDUCTION(+:sa)
  do j = 1, size(a,1)
     sa = sa + a(j)
  enddo
!$OMP END PARALLEL DO
  end subroutine
```

Stride 1 memory access important:

Read memory locations in order

Serial segmented 1d sum reduction
Partial sums of blocks of size mx

```fortran
  subroutine sum1(a,sa,mx)
! 1d sum reductions, each of length mx
! sa = sum(a)
  implicit none
  integer :: mx
  real :: sa
  real, dimension(:) :: a
  integer :: j, js, jb, nx, nbx
  real :: t
  nx = size(a,1)
  nbx = (nx - 1)/mx + 1

  sa = 0.0
  do jb = 1, nbx
  t = 0.0
  do js = 1, min(mx,nx-mx*(jb-1))
     j = js + mx*(jb - 1)
     t = t + a(j)
  enddo
  sa = sa + t

  enddo

  end subroutine
```

Programming GPUs:
Sum reduction of 1D array

On GPU, copy to shared memory one thread performs partial sum
Not very efficient

```fortran
  subroutine sum1(a,sa,mx)
! sa = sum(a)
  ...
  real :: sa
  real, dimension(:) :: a
  ...
  sa = 0.0
  do jb = 1, nbx
  t = 0.0
  do js = 1, min(mx,nx-mx*(jb-1))
     j = js + mx*(jb - 1)
     t = t + a(j)
  enddo
  sa = sa + t

  enddo

  end subroutine
```

The inner part of loop in GPU kernel function:

```fortran
    attributes(global) subroutine gsum1(a,sa,nx)
! inner part of loop goes here
    implicit none
    integer, value :: nx
    real, dimension(:) :: a, sa
    integer :: j, js, jb, mx, joff, mxm
    real :: t
    real, shared, dimension(*) :: s
    mx = blockDim%x
    js = threadIdx%x
    jb = blockIdx%x
    joff = mx*(jb - 1)

    j = js + joff  ! first copy to shared memory
    if (j <= nx) s(js) = a(j)
    call syncthreads()
    if (js==1) then     ! one thread performs sum
       mxm = nx - joff
       if (mxm > mx) mxm = mx
       t = 0.0
       do j = 1, mxm
          t = t + s(j)
       enddo               ! sum different blocks
       t = atomicAdd(sa(1),t)
    endif

    end subroutine
```
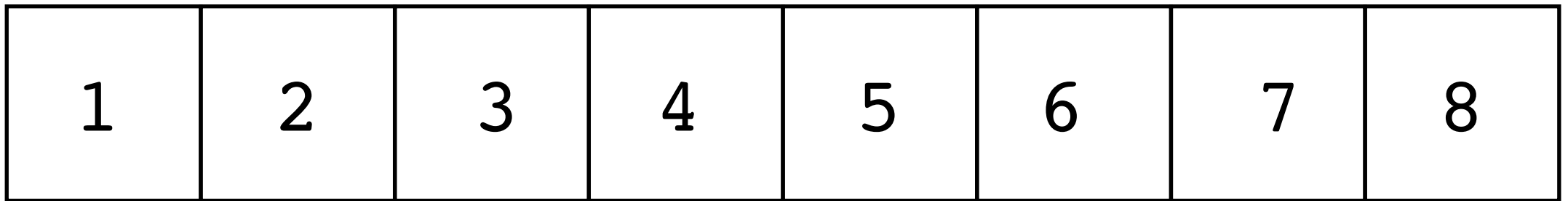
Programming GPUs:
Sum reduction of 1D array

GPU kernel function:

```fortran
attributes(global) subroutine
          gsum1(a,sa,nx)
...
real, shared, dimension(*) :: s
mx = blockDim%x
js = threadIdx%x
jb = blockIdx%x
joff = mx*(jb - 1)

j = js + joff
if (j <= nx) s(js) = a(j)
call syncthreads()
if (js==1) then      mxm = nx - joff
   if (mxm > mx) mxm = mx
   t = 0.0
   do j = 1, mxm
      t = t + s(j)
   enddo
   t = atomicAdd(sa(1),t)
endif

end subroutine
```

host calling function:

```fortran
subroutine gpu_sum1(a,sa,mx)
implicit none
integer :: mx
real, device, dimension(:) :: a, sa
integer :: nx, nbx, ns
type (dim3) :: dimBlock, dimGrid
nx = size(a,1)
nbx = (nx - 1)/mx + 1
dimBlock = dim3(mx,1,1)
dimGrid = dim3(nbx,1,1)
sa(1) = 0.0
! calculate size of shared memory
ns = mx*sizeof(a(1))

call gsum1<<<dimGrid,dimBlock,ns>>>(a,sa,nx)

end subroutine
```
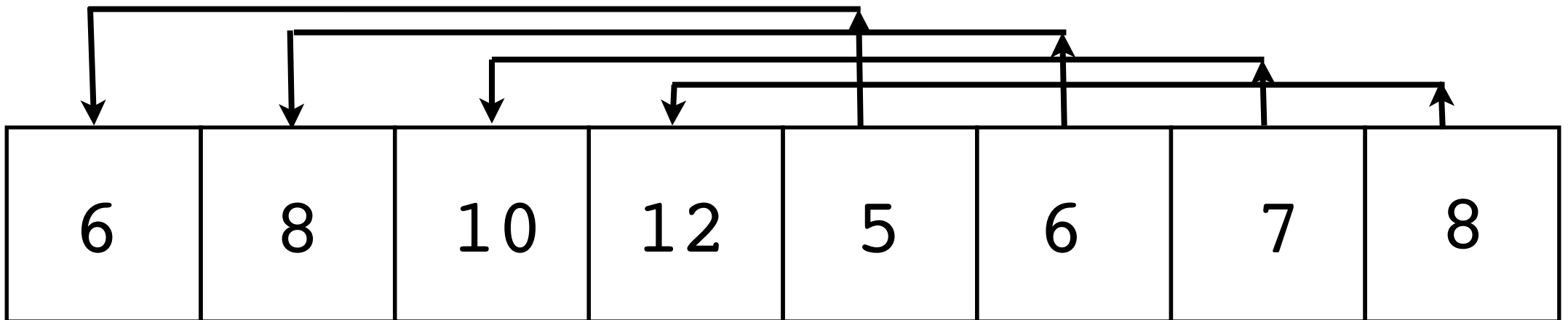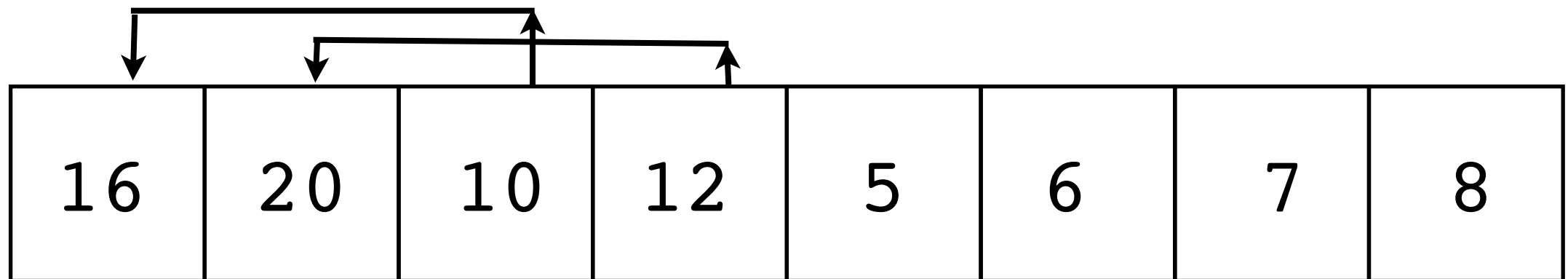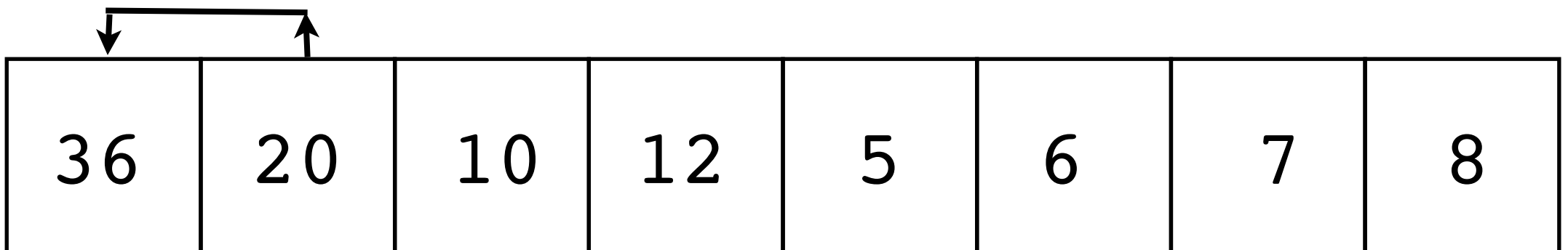
Parallel Sum Reduction:

0:

1	2	3	4	5	6	7	8

1:

6	8	10	12	5	6	7	8

2:

16	20	10	12	5	6	7	8

3:

36	20	10	12	5	6	7	8

Wednesday, May 27, 2015

Programming GPUs: Sum reduction of 1D array

Implementation of local parallel sum reduction
device functions are inlined, sdata is in shared memory

```fortran
    attributes(device) subroutine lsum2(sdata,n)
! finds local sum of n data items shared by threads
! using binary tree method. input is modified
    implicit none
    real, dimension(*) :: sdata
    integer, value :: n
    integer :: l, k
    real :: s
    l = threadIdx%x
    k = blockDim%x/2
    s = 0.0

    if (l <= n) s = sdata(l)
    do while (k > 0)
       if (l <= k) then
          if ((l+k) <= n) then
             s = s + sdata(l+k)
             sdata(l) = s
          endif
       endif
       call syncthreads()
       k = k/2
    enddo

    end subroutine
```

Programming GPUs:
Sum reduction of 1D array

Prior GPU kernel function:

```
attributes(global) subroutine
            gsum1(a,sa,nx)
...
real, shared, dimension(*) :: s
mx = blockDim%x
js = threadIdx%x
jb = blockIdx%x
joff = mx*(jb - 1)

j = js + joff
if (j <= nx) s(js) = a(j)
call syncthreads()

if (js==1) then
   mxm = nx - joff
   if (mxm > mx) mxm = mx
   t = 0.0
   do j = 1, mxm
      t = t + s(j)
   enddo
   t = atomicAdd(sa(1),t)
endif

end subroutine
```

Multiple parallel sum reductions:
write out partial sum of each block

GPU kernel function:

```
attributes(global) subroutine gsum2(a,d,nx)
implicit none
integer, value :: nx
real, dimension(:) :: a, d
integer :: j, js, jb, mx, joff, mxm
real, shared, dimension(*) :: s
mx = blockDim%x
js = threadIdx%x
jb = blockIdx%x
joff = mx*(jb - 1)

j = js + joff
if (j <= nx) s(js) = a(j)
call syncthreads()

mxm = nx - joff
if (mxm > mx) mxm = mx
call lsum2(s,mxm)
if (js==1) d(jb) = s(1)

end subroutine
```

Programming GPUs:
Sum reduction of 1D array

Multiple parallel sum reductions:
write out partial sum of each block

GPU kernel function:

host calling function:

```fortran
attributes(global) subroutine
          gsum2(a,d,nx)
implicit none
integer, value :: nx
real, dimension(:) :: a, d
integer :: j, js, jb, mx, joff, mxm
real, shared, dimension(*) :: s
mx = blockDim%x
js = threadIdx%x
jb = blockIdx%x
joff = mx*(jb - 1)

j = js + joff
if (j <= nx) s(js) = a(j)
call syncthreads()

mxm = nx - joff
if (mxm > mx) mxm = mx
call lsum2(s,mxm)
if (js==1) d(jb) = s(1)

end subroutine
```

```fortran
subroutine gpu_sum2(a,d,mx)
implicit none
integer :: mx
real, device, dimension(:) :: a, d
integer :: nx, nbx, ns
type (dim3) :: dimBlock, dimGrid
nx = size(a,1)
nbx = (nx - 1)/mx + 1
dimBlock = dim3(mx,1,1)
dimGrid = dim3(nbx,1,1)
! calculate size of shared memory
ns = mx*sizeof(a(1))

call gsum2<<<dimGrid,dimBlock,ns>>>(a,d,nx)

crc = cudaThreadSynchronize()

end subroutine
```

Programming GPUs: Sum reduction of 1D array

Many times only local sums are required
If global sum is needed, one can combine these two procedures

Perform parallel sums for each block:
  call gpu_sum2(g_a,g_d,mx)
Then add the partial sums with serial algorithm:
  call gpu_sum1(g_d,g_s,mx)

If array is long, one can iterate gpu_sum2 several times.
This is done by the host procedure gpu_sum3:
  call gpu_sum3(g_a,g_d,g_s,mx)

Sample output:

./fexample3

 Fortran empty kernel time=   5.1999999E-05
 Fortran 1d sum time=   4.0000000E-06
 GPU 1d sum time=   2.5300001E-04
 1d sum maximum difference =   0.000000
 s,t =   4501500.   4501500.

Programming GPUs: Conclusions

- Vector algorithms are relatively easy
- Processing has to be done in small blocks
- Tiling algorithms are well known from cache based machines
- Sum reductions are harder
- Irregular problems such as reordering data can be very hard
- Very useful to develop a serial version before implementing a Cuda version
- Libraries such as BLAS, FFTs, are available to avoid reinventing the wheel
- New languages such as OpenACC (similar to OpenMP) are evolving