High Performance Computing and GPU Programming

Lecture 3: GPU Application

GPU Intro Review

Simple Example

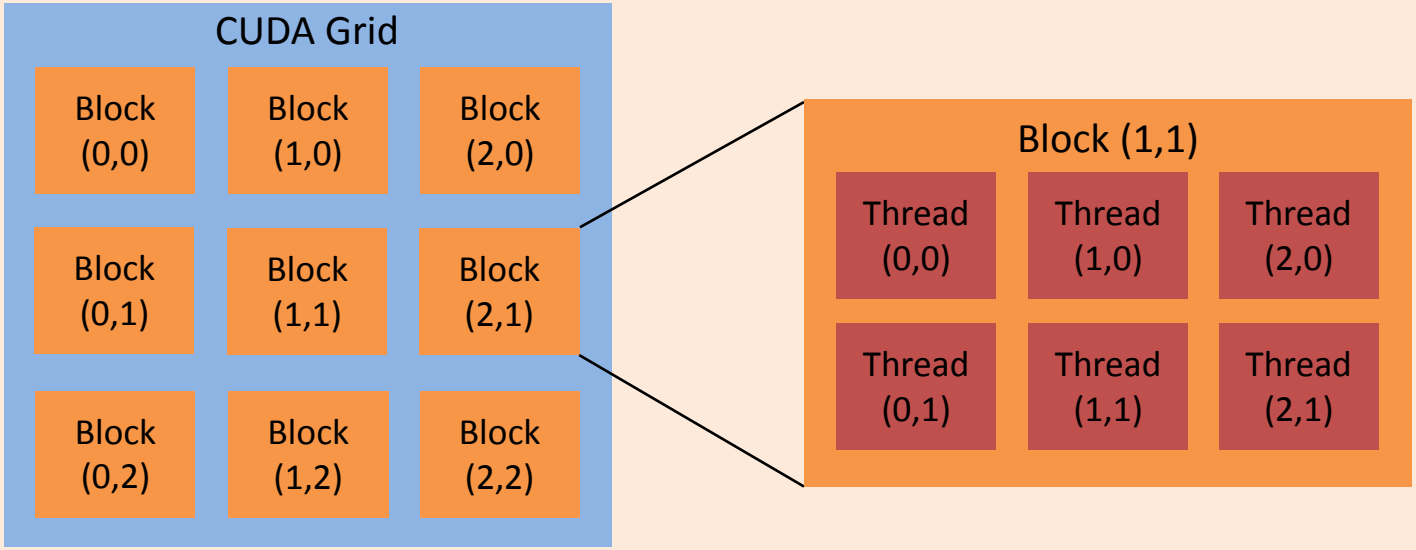
Memory Effects

GPU Intro Review

GPU Intro Review

- Shared Multiprocessors
  - “Global” parallelism
  - Assign threads into Warps (32 threads)
- Blocks
  - 1 block is executed by 1 SM
  - Reside in a 1-D or 2-D Grid
  - Composed of threads – can be 1-D, 2-D, or 3-D
- Threads
  - “Local” parallelism
  - Max = 1024 per block

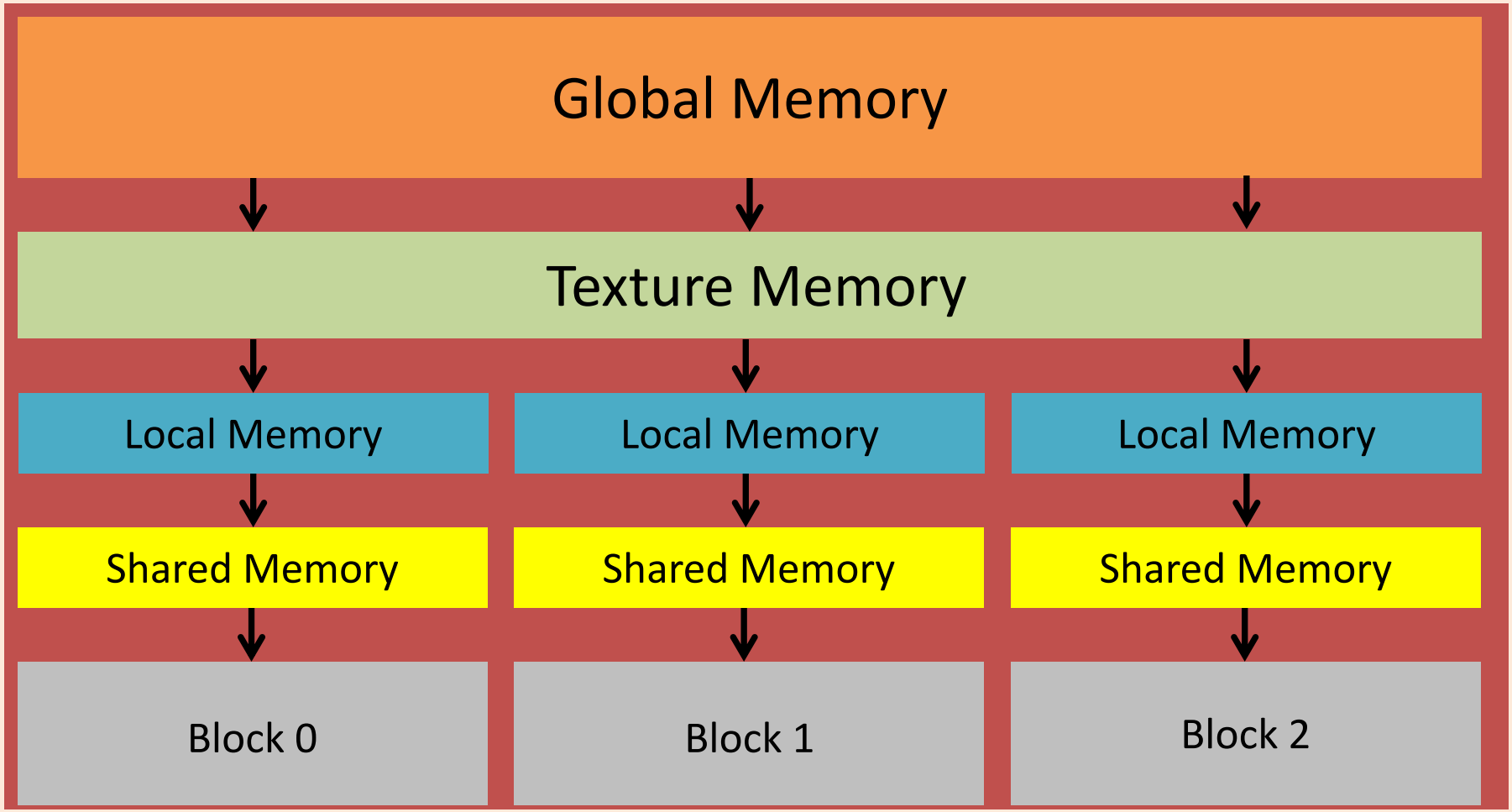
GPU Intro Review



- Remember the programming model
- GPUs like to have data independence
  - Achieve massive parallelism

GPU Into Review

- Finally recall GPU memory types
- Each type has advantages and disadvantages



Simple Example

Simple Example

- The example is:
  - Simple mathematically
  - Rather complicated implementation wise

- Consider data interpolation
  - $\widetilde{f}_i(x)$
  - $f_{i,j}$
  - $L(x)$

$$\widetilde{f}_i(x) = f_{i,1}L_1(x) + f_{i,2}L_2(x) + f_{i,3}L_3(x) + \ \ldots + f_{i,n}L_n(x)$$

Simple Example

- Just a linear combination of the value at a node multiplied by the corresponding value of the interpolation polynomial
- Let's evaluate the derivative at n=3 points

$$\frac{\partial f(x_1)}{\partial x} = f_{i1}L_1'(x_1) + f_{i2}L_2'(x_1) + f_{i3}L_3'(x_1)$$

$$\frac{\partial f(x_2)}{\partial x} = f_{i1}L_1'(x_2) + f_{i2}L_2'(x_2) + f_{i3}L_3'(x_2)$$

$$\frac{\partial f(x_3)}{\partial x} = f_{i1}L_1'(x_3) + f_{i2}L_2'(x_3) + f_{i3}L_3'(x_3)$$

Simple Example

- Data organization

$$\frac{\partial f(x_m)}{\partial x} = f_{i1}L_1'(x_m) + f_{i2}L_2'(x_m) + f_{i3}L_3'(x_m)$$

- Recall we like 1-D arrays
  - Allows faster access
  - Let's you understand access patterns

- Let's look at a CPU C++ code!

Simple Example

```
int main () {

   //Declaration of variables
   double *f, *fp, *Ld;
   int ne, np;

   //Allocate memory
   f  = (double *)malloc(sizeof(double) * ne*np);
   fp = (double *)malloc(sizeof(double) * ne*np);
   Ld = (double *)malloc(sizeof(double) * np*np);

   //Import the coefficients here (from MatLab)
   // ...
   for (int i=0; i<np; i++)
   for (int j=0; j<np; j++) {

      Ld[j + i*np] = data->coefficients[j + i*np];
   }
// ...
```

Simple Example

```
//Interpolate the data
   for (int i=0; i<ne; i++)
   for (int j=0; j<np; j++)
   {
      //Initialize to zero
      fp[j + i*np] = 0.0;

      for (int m=0; m<np; m++) {

         //Form the derivative at point j
         fp[j + i*np] += f[m + i*np] * Ld[m + j*np];
      }
   }

   return 0;
}
```

- Basic right?
- Count the loops…

Simple Example

- How do we put this on a GPU?

$$\vec{t} = [t_x, t_y, t_z] \qquad \vec{b} = [b_x, b_y]$$

- Need to first organize the blocks and threads
  - For simplicity let
    - $b = [n_e]$
    - $\vec{t} = [n_p]$
  - As a side note this is awful
    - Only 3 threads per block when we have 1024 available
    - The Warp is terrible
    - But this is easy to understand!

- Let's start the GPU code now!

Simple Example

```
int main () {

   //Declaration of variables
   double *f, *fp, *Ld;
   double *f_d, *fp_d, *Ld_d;
   int ne, np;

   //Allocate CPU memory
   f  = (double *)malloc(sizeof(double) * ne*np);
   fp = (double *)malloc(sizeof(double) * ne*np);
   Ld = (double *)malloc(sizeof(double) * np*np);

   //Allocate GPU memory
   cudaMalloc((void **) &f_d,  sizeof(double) * ne*np);
   cudaMalloc((void **) &fp_d, sizeof(double) * ne*np);
   cudaMalloc((void **) &Ld_d, sizeof(double) * np*np);

   //Load in Lagrange data
   //…
```

Simple Example

```
//Copy Data to Device
cudaMemcpy(f_d,  f,  sizeof(double) * np*ne,  cudaMemcpyHostToDevice);
cudaMemcpy(fd_d, fd, sizeof(double) * np*ne,  cudaMemcpyHostToDevice);
cudaMemcpy(Ld_d, Ld, sizeof(double) * np*np, cudaMemcpyHostToDevice);

//Threads and blocks
dim3 threads(np);
int  blocks(ne);

//Launch the kernel
interpolate<<<blocks, threads>>>(fd_d, f_d, L_d, np, ne);

cudaMemcpy(fd, fd_d, sizeof(double) * np*ne, cudaMemcpyDeviceToHost);
```

- Question for everyone
  - Where is all this data located at? Which memory?
  - What about np and ne?
  - What is the CPU doing now?

Simple Example

```
__global__ void interpolate(double *fd, double *f, double *L,
int np, int ne) {

   int j = threadIdx.x;    //Data point
   int i = blockIdx.x;     //Cell

   for (int m=0; m<np; m++) {

      fd[j + i*np] += f[m + i*np] * L[m + j*np];
   }
}
```

- That did not look to complicated
  - Performance is poor
  - Why?
  - How can we improve?

Memory Effects

Memory Effects

```
int j = threadIdx.x;    //Data point
int i = blockIdx.x;     //Cell

for (int m=0; m<np; m++) {

    fd[j + i*np] += f[m + i*np] * L[m + j*np];
}
```

- Global memory access and computation
  - Never do this
  - At 10 points and 100 elements…
    - ~0.083 milliseconds
    - Worse than the CPU code
    - NVIDIA GeForce GT 630M Vs. Intel I7 @ 2.2GHz

Memory Effects

- For your understanding – let's go through each memory one at a time

1. Texture Memory
   - We will bind the global memory into texture memory
   - Must be bound in the CPU code
   - Then accessed by GPU code

```
texture<int2, 1, cudaReadModeElementType> t_f;
texture<int2, 1, cudaReadModeElementType> t_fd;
texture<int2, 1, cudaReadModeElementType> t_Ld;
```

Double Precision

```
texture<float, 1, cudaReadModeElementType> t_f;
texture<float, 1, cudaReadModeElementType> t_fd;
texture<float, 1, cudaReadModeElementType> t_Ld;
```

Single Precision

Memory Effects

```
texture<int2, 1, cudaReadModeElementType> t_f;
texture<int2, 1, cudaReadModeElementType> t_fd;
texture<int2, 1, cudaReadModeElementType> t_Ld;

int main () {

   //Import data and initialize arrays
   // ...

   //Copy Data to Device
   cudaMemcpy(f_d,  f,  sizeof(double) * np*ne,  cudaMemcpyHostToDevice);
   cudaMemcpy(fd_d, fd, sizeof(double) * np*ne,  cudaMemcpyHostToDevice);
   cudaMemcpy(Ld_d, Ld, sizeof(double) * np*np, cudaMemcpyHostToDevice);

   //Texture Binding
   cudaBindTexture(0, t_f,  f_d,  sizeof(double) * np*ne);
   cudaBindTexture(0, t_fd, fd_d, sizeof(double) * np*ne);
   cudaBindTexture(0, t_Ld, Ld_d, sizeof(double) * np*np);

   // ...
```

- I will do everything in double precision
- Did I need to bind pointer fd?

Memory Effects

- Did I really need texture memory here?
  - Answer: No
  - Everything is coalesced access to global memory
  - Slight improvement for L_d read
- It is good practice to get used to using textures
  - Just bind everything
  - Never know when might have non-coalesced access
  - Binding is cheap – basically free

Memory Effects

2. Local / Registers

```
__global__ void interpolate(double *fd, double *f, double *L, int
np, int ne) {

   int j = threadIdx.x;    //Data point
   int i = blockIdx.x;       //Cell

   //Local memory allocation
   double fd_l = 0.0;

   for (int m=0; m<np; m++) {

      //Global memory calculation into registers
      fd_l += f[m + i*np] * L[m + j*np];
   }

   //Write registers to global
   fd[j + i*np] = fd_l;
}
```

Memory Effects

- So what happened to time?
  - 10 points 100 cells
    - Old time: ~0.083 milliseconds
    - New time: ~0.058 milliseconds

- Much better!
  - Still needs improvement

```
for (int m=0; m<np; m++) {

    //Global memory calculation into registers
    fd_l += f[m + i*np] * L[m + j*np];
}
```

  - Writes into registers – but calculation is in global memory
  - How do we fix this?

Memory Effects

- If we put the memory in registers – need to allocate memory – do you see why?

```
for (int m=0; m<np; m++) {

    //Global memory calculation into registers
    fd_l += f[m + i*np] * L[m + j*np];
}
```

- Will result in un-coalesced access to registers
- Need a memory that:
  - Is good for multiple access per thread
  - Where can I get that?

Memory Effects

3. Shared memory

```
__global__ void interpolate(double *fd, double *f, double *L, int
np, int ne) {

   int j = threadIdx.x;    //Data point
   int i = blockIdx.x;       //Cell

   //Local memory allocation
   double fd_l = 0.0;

   //Shared memory allocation
   __shared__ double f_s[np];
   __shared__ double L_s[np*np];
```

- __shared__ command
- Notice allocation size

Memory Effects

```
int2 ii;

ii      = tex1Dfetch(t_f, j + i*np);
f_s[j] = __hiloint2double(ii.y, ii.x);

for (int m=0; m<np; m++) {

        ii = tex1Dfetch(t_L, m + j*np);
        L_s[m + j*np] = __hiloint2double(ii.y, ii.x);
}
__syncthreads();
```

- int2 command access the textured memory
- Much more coding required!
- __syncthreads()

Memory Effects

```
for (int m=0; m<np; m++) {

    //Shared memory calculation into registers
    fd_l += f_s[m] * L_s[m + j*np];
}
```

- So what have we completed?
- Well…

Memory Effects

```
__global__ void interpolate(double *fd, double *f,
double *L,
int nsp, int nc) {

   int j = threadIdx.x;    //Data point
   int i = blockIdx.x;     //Cell

   for (int m=0; m<nsp; m++) {

      fd[j + i*nsp] += f[m + i*nsp] * L[m + j*nsp];
   }
}
```

- My code blew up in size
- This is only 1-D
  - Tripled in size
  - Was it worth it?
  - Only a little faster
  - Why?

```
__global__ void interpolate(double *fd, double *f,
double *L, int  nsp, int nc) {

   int j = threadIdx.x;    //Data point
   int i = blockIdx.x;       //Cell

   //Local memory allocation
   double fd_l = 0.0;

   //Shared memory allocation
   __shared__ double f_s[nsp];
   __shared__ double L_s[nsp*nsp];

   int2 ii;

   ii      = tex1Dfetch(t_f, j + i*nsp);
   f_s[j] = __hiloint2double(ii.y, ii.x);

   for (int m=0; m<nsp; m++) {

        ii = tex1Dfetch(t_L, m + j*nsp);
        L_s[m + j*nsp] = __hiloint2double(ii.y, ii.x);
   }
   __syncthreads();

  for (int m=0; m<nsp; m++) {

       //Global memory calculation into registers
       fd_l += f_s[m] * L_s[m + j*nsp];
    }

    //Write registers to global
    fd[j + i*nsp] = fd_l;
```

Memory Effects

- We are limited by our blocks and threads
  - With proper management we will see an additional factor of 2x speed-up
  - 1-D code is also very small
  - Not going to see 30 – 70x faster than a CPU version
  - If we can get a factor of 4 – 10x we are doing good

- Important
  - Use GPU registers
  - One write to global memory at the end
  - Ensures fast kernel

Wrap Up

- Simple code can become very complex
  - Every problem is very different for GPUs
- We saw roughly factor 2x from memory management only
- This problem is very basic though
- Next week
  - Thread and block management
  - Kernel thread control
  - 3-D coding problem