Course Contents

This week:

- Overview of GPU hardware
- Introduction to CUDA and how to program for GPU
- Programming restrictions and bottlenecks

Next week (Hybrid Programming in CUDA, OpenMP and MPI):

- Using multiple GPU within a node
- Simultaneously use multiple CPU cores with OpenMP
- Scale across many nodes using MPI
Course Contents

What won’t be covered (and where to find it):

- CUDA 5.0 (released 10/16/2012)
- Fortran (see directive-based PGI Accelerators)
- Directive based GPU programming such as OpenACC
- CUDA libraries such as those available from Accelereyes
- CUDA debugging tools such as TotalView, Allinea DDT, etc.

For webinars on these (and other topics) go to: 
What are GPU?

GPU = Graphics Processing Unit

GP-GPU = General Purpose GPU

- Originally developed to render graphics quickly
- Paced themselves out of the gaming market
- Looked to the HPC community for fresh blood ($ $)
When to Use GPU

GPU are specialized processors that are very good at solving some problems and not very good at solving others!

Some reasons why GPU are attractive:

- GPU are designed for data-parallel applications
- GPU are capable of achieving much higher throughput than CPU
- Implemented correctly, certain kinds of problems can be solved MUCH faster using GPU
When to Use GPU

Some challenges associated with GPU:

- Some algorithms do not extend well to SIMD
- GPU do not perform I/O
- Transfers between CPU and GPU memory spaces can be expensive
- Balancing workload is not straightforward when using hybrid CPU-GPU approaches
GPU for Scientific Computing

**Introduction**

**Preliminaries**

**CUDA Kernels**

**Memory Management**

**Streams and Events**

**Shared Memory**

**Toolkit Overview**

---

**Graph 1:**
- Theoretical GFLOPs over time for NVIDIA GPU Single Precision, NVIDIA GPU Double Precision, Intel CPU Single Precision, and Intel CPU Double Precision.
- Key GPUs include: GeForce GTX 580, GeForce GTX 480, GeForce GTX 280, GeForce 7800 GTX, GeForce 8800 GTX, and others.
- Timeline from Sep-01 to Dec-09.

**Graph 2:**
- Theoretical GB/s over time for CPU and GPU.
- Key GPUs include: GeForce GTX 580, GeForce GTX 480, GeForce GTX 280, GeForce 8800 GTX, and others.
- Timeline from 2003 to 2010.
Hardware Overview

- Host
- Host Memory
- Device (GPU)
- Device Memory
- PCIe

Diagram showing the flow between the host and device memory through PCIe.
Memory Hierarchy for Fermi GPU (HokieSpeed)

- Local storage (on chip)
  - register memory associated with each thread
- Shared memory / L1 (on-chip)
  - Shared memory accessible by all threads in a block
  - Configurable: 16KB/48KB or 48KB/16KB
  - Low latency
  - Throughput: \( \sim 1 \text{ TB/s} \)
- L2 (off-chip)
- Global memory (off-chip)
  - Accessible by all threads
  - Higher latency (400-800 cycles)
  - Throughput: \( \sim 120 \text{ GB/s} \)
Compute Capability of NVidia GPU

- GPU hardware is evolving rapidly
- Depending on how new your GPU is, it may or may not:
  - support double precision calculations
  - support atomic operations
  - support 3-D grid blocks
  - do something else that you need it to do
- “Compute capability” provides a way to evaluate the hardware capabilities
- Possibilities are: 1.0, 1.1, 1.2, 1.3, 2.x, 3.0, 3.5
Compiling with CUDA

On HokieSpeed, load the compiler by typing:

- `module swap intel gcc`
- `module load cuda`

To view with the modules you have loaded:

- `module list`

To see a list of available modules:

- `module avail`

To see the changes a module makes to your environment:

- `module show`
The CUDA compiler `nvcc` compiles by:

- identifying device (ie. gpu) functions and compiling them
- passing host (ie. cpu) functions to gcc/g++

To compile, type:

```
• nvcc -o runme program.cu
```

To compile with *double precision* support, type:

```
• nvcc -o runme -arch sm_13 program.cu
```
Submitting CUDA Jobs

As long as you have GPU available, job submission is similar to other ARC resources:

- `wget http://www.arc.vt.edu/resources/hpc/docs/hokiespeed_qsub_example.sh`
- Use `emacs` or `vi` to edit the runscript for your code
- Hands-on example
The CUDA Programming Model

Grid

Block (0, 0) Block (1, 0) Block (2, 0)
Block (0, 1) Block (1, 1) Block (2, 1)

Block (1, 1)

Thread (0, 0) Thread (1, 0) Thread (2, 0) Thread (3, 0)
Thread (0, 1) Thread (1, 1) Thread (2, 1) Thread (3, 1)
Thread (0, 2) Thread (1, 2) Thread (2, 2) Thread (3, 2)
CUDA kernels

Parts of your program that run on GPU must be provided as CUDA kernels:

```cpp
__global__ void foo(...) {
    // indices associated with this thread
    int i, j;
    i = blockIdx.x*blockDim.x+threadIdx.x;
    j = blockIdx.y*blockDim.y+threadIdx.y;
    // Do some work that depends on i and j
}
```
Thread Hierarchy

Example

```c
// Set up threadblocks
dim3 threadsPerBlock(8);
dim3 numBlocks(4);
// Kernel invocation
foo <<<<numBlocks,threadsPerBlock>>>> (...)```

```
threadIdx.x  threadIdx.x  threadIdx.x  threadIdx.x
0 1 2 3 4 5 6 7 0 1 2 3 4 5 6 7 0 1 2 3 4 5 6 7 0 1 2 3 4 5 6 7
blockIdx.x = 0  blockIdx.x = 1  blockIdx.x = 2  blockIdx.x = 3
```
Thread Hierarchy

Example

```c
// 1-D threadblocks
dim3 threadsPerBlock(nx);
dim3 numBlocks(Nx);

// 2-D threadblocks
dim3 threadsPerBlock(nx,ny);
dim3 numBlocks(Nx,Ny);

// 3-D threadblocks
dim3 threadsPerBlock(nx,ny,nz);
dim3 numBlocks(Nx,Ny,Nz);
```
How to Specify Thread Blocks

- Large number of memory transactions are needed to hide latency
- Increase transactions by:
  - multiple independent memory accesses from each thread
  - more threads in a threadblock
  - larger word sizes (i.e. double v. float)
Goal: saturate memory bandwidth:

- Threads per block should be a multiple of the warp size (32)
- Multiple threadblocks can be scheduled simultaneously
  - small threadblocks may fail to maximize occupancy
  - large threadblocks are less flexible
- It’s a good idea to vary the number of threads per block so that you can tune the performance for your application (32, 64, 128, 256, ...)

Example: Vector Addition

Suppose we want to compute the vector sum \( \vec{u} = \vec{v} + \vec{w} \)

- CPU computations are performed serially (i.e., in order)
- GPU computations are performed simultaneously
- Memory bandwidth limits how quickly computations are performed

Example

```cpp
// serial vector addition in C++
for (i=0; i<N; i++)
    u[i] = v[i] + w[i];
```
Example: Vector Addition

Example

/* Copy a and b from CPU to GPU */
int numThreads = 128;
dim3 numBlocks(N/128);
VecAdd<<<numBlocks, numThreads>>>(u,v,w,N)
/* Copy c from the GPU to CPU */

Example

__global__ void VecAdd(...)
{
    // thread-based vector addition in CUDA
    int i=blockIdx.x*blockDim.x+threadIdx.x;
    if (i<N) u[i] = v[i] + w[i];
}
Exercise: Matrix-Vector Multiplication

Write a CUDA kernel to compute \( \vec{u} = A \vec{v} \)

- \( A \) is an \( N \times N \) matrix with constant coefficients
- compute a sequence of \( \vec{u} \) for different \( \vec{v} \)

Example

```cpp
// Serial matrix-vector multiply in C++
for (j=0; j<N; j++)
    u[j] = 0.0;
for (j=0; j<N; i++)
    for (i=0; i<N; i++)
        u[j] += A[i][j]*v[i];
```

Is this implementation optimal?
Questions to consider:

1. How do you want to break up the problem among threads?
   - Use 1-D threadBlocks?
   - Use 2-D threadBlocks?

2. How does your code access memory?

3. How do $\vec{u}$ and $\vec{v}$ get to the GPU / CPU?
Hardware Overview

Host

Device (GPU)

Host Memory

Device Memory

PCIe
Allocating Memory in CUDA

Example

```c
// Allocate memory
double *data1, *data2, *data3;
data1 = malloc(size);
cudaMallocHost(&data2, size);
cudaMalloc((void**)&data3, size);
/* Do some work */
// Release the memory
free(data1);
cudaFreeHost(data2);
cudaFree(data3);
```
Memory Allocations in CUDA

- Host and device memory are separate entities
  - Device pointers point to GPU memory
  - Host pointers point to CPU memory

Let’s consider the implications for two functions:

**Example**

```c
__global__ void foo(double *a);
inline void bar(double *a);
```
Memory Allocations in CUDA

Example

```c
inline void foo(double *a);
__global__ void bar(double *a);
```

Example

```c
double *a;
cudaMalloc((void**)&a, size);
foo <<<grid,numthreads>>>(a);
free(a);
```
Memory Allocations in CUDA

Example

```c
__global__ void foo(double *a);
inline void bar(double *a);
```

Example

```c
double *a;
a = malloc(size);
foo(a);
free(a);
```
Memory Allocations in CUDA

Example

```c
__global__ void foo(double *a);
inline void bar(double *a);
```

Example

```c
double *a;
cudaMallocHost(a, size);
bar(a);
free(a);
```
Memory Transfers in CUDA

- CPU and GPU have their own distinct memory
- CUDA provides mechanisms to transfer data between the memory spaces
- Memory bandwidth for CPU ↔ GPU transfers is limited by PCI express (slow)
  - Avoid memory transfers whenever possible!
  - Can we develop a strategy to hide this communication?
Memory Transfers in CUDA

Example

```c
// Copy data from the CPU to the GPU
cudaMemcpy(cpu_data, gpu_data, size, cudaMemcpyHostToDevice);

/* ... perform work on the GPU ... */

// Copy the result back to the CPU
cudaMemcpy(gpu_data, cpu_data, size, cudaMemcpyDeviceToHost);
```
Exercise: Allocating Memory in CUDA

- How much memory can you allocate using `malloc`?
- How much memory can you allocate using `cudaMallocHost`?
- How much memory can you allocate using `cudaMalloc`?
CUDA Streams and Events

The CUDA driver API provides streams and events as a way to manage GPU synchronization:

- Subject to availability, GPU can compute multiple kernels simultaneously
- Synchronization is implied for events within a stream (including default stream)
- Streams belong to a particular GPU
- More than one stream can be associated with a GPU
- Streams are required if you want to perform asynchronous communication
- Streams are critical if you want concurrency with multiple GPU or multiple kernels on any single GPU.
CUDA Streams

Example

```c
// Create a pair of streams
cudaStream_t stream[2];
for (int i=0; i<2; ++i)
    cudaStreamCreate(&stream[i]);

// Launch a Kernel from each stream
KernelOne<<<100,512,0,stream[0]>>>(..)
KernelTwo<<<100,512,0,stream[1]>>>(..)

// Destroy the streams
for (int i=0; i<2; ++i)
    cudaStreamDestroy(stream[i]);
```
CUDA events provide another way to monitor the progress of a device and can be used to accurately time device operations.
CUDA Events

Example

```c
float time;
cudaEvent_t start, stop;
cudaEventCreate(&start);
cudaEventCreate(&stop);
cudaEventRecord(start, stream);
/* Do some GPU work and time it */
cudaEventRecord(stop, stream);
cudaEventSynchronize(stop);
cudaEventElapsedTime(&time, start, stop);
```
Streams can be synchronized explicitly:

- `cudaDeviceSynchronize()`: wait for all preceding commands in all streams for a device to complete.
- `cudaStreamSynchronize()`: wait for all preceding events in a specified stream to complete.
- `cudaStreamWaitEvent()`: synchronize a stream with an event (both must be specified).
- `cudaStreamQuery()`: Ask the system if preceding commands in a stream have completed.
CUDA Streams

Two streams will be synchronized implicitly if any of the following operations are issued in between them:

- a page-locked memory allocation (using `cudaMallocHost`)
- a device memory allocation (using `cudaMalloc`)
- a memory copy between two devices
- any CUDA command to the default stream
Asynchronous Memory Transfers in CUDA

- We know that CPU ↔ GPU memory transfers are expensive
- We also know that PCIe can perform simultaneous, bi-directional transfers:
  - One cudaMemcpy for Host → Device
  - One cudaMemcpy for Device → Host
- If the memory transfers belong to the same stream they will be synchronized
- We need a way to asynchronously perform transfers to get the full advantage of PCIe
Asynchronous Memory Transfers in CUDA

Example

```c
// Host data MUST be pinned!!!
cudaMallocHost(&cpuData, size);
cudaMalloc(&gpuData, size);
// Bi-directional memory transfer
cudaMemcpyAsync(gpuData, cpuData, size, cudaMemcpyHostToDevice, stream[0]);
cudaMemcpyAsync(cpuData, gpuData, size, cudaMemcpyDeviceToHost, stream[1]);
// Clean up
```
Exercise: Memory Transfers in CUDA

- Copy some data from the host to the device
- Copy some data from the device to the host
- How much does the performance of cudaMemcpy change depending on whether host memory is allocated with malloc or cudaMallocHost?
- Perform the same comparison using cudaMemcpyAsync using streams
- Can you validate the bi-directional transfers are simultaneous?
Alignment and Coalescence

Achieving high performance on GPU depends strongly on data alignment and access patterns:

- Memory transactions will access a chunk of data of fixed size (32B or 128B)
- You want to use all of the data that you bring back
- Global memory accesses for threads within a warp can be coalesced
- Optimize data structures to take advantage of this and significantly increase performance
- Restrictions vary based on compute capability of device
### Alignment and Coalescence

<table>
<thead>
<tr>
<th>Addresses</th>
<th>96</th>
<th>128</th>
<th>160</th>
<th>192</th>
<th>224</th>
<th>256</th>
<th>288</th>
</tr>
</thead>
<tbody>
<tr>
<td>Threads:</td>
<td>0</td>
<td>...</td>
<td>31</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Compute capability:</td>
<td>1.0 and 1.1</td>
<td>1.2 and 1.3</td>
<td>2.0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Memory transactions:</td>
<td>Uncached</td>
<td>Uncached</td>
<td>Cached</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>1 x 64B at 128</td>
<td>1 x 64B at 128</td>
<td>1 x 128B at 128</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>1 x 64B at 192</td>
<td>1 x 64B at 192</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Alignment and Coalescence

<table>
<thead>
<tr>
<th>Addresses:</th>
<th>96</th>
<th>128</th>
<th>160</th>
<th>192</th>
<th>224</th>
<th>256</th>
<th>288</th>
</tr>
</thead>
<tbody>
<tr>
<td>Threads:</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>31</td>
</tr>
</tbody>
</table>

#### Computed capability:
- 1.0 and 1.1
- 1.2 and 1.3
- 2.0

<table>
<thead>
<tr>
<th>Memory transactions:</th>
<th>Uncached</th>
<th>Cached</th>
</tr>
</thead>
<tbody>
<tr>
<td>8 x 32B at 128</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8 x 32B at 160</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8 x 32B at 192</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8 x 32B at 224</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
## Alignment and Coalescence

### Misaligned and sequential

<table>
<thead>
<tr>
<th>Addresses:</th>
<th>96</th>
<th>128</th>
<th>160</th>
<th>192</th>
<th>224</th>
<th>256</th>
<th>288</th>
</tr>
</thead>
<tbody>
<tr>
<td>Threads:</td>
<td>0</td>
<td>...</td>
<td>31</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Compute capability:

<table>
<thead>
<tr>
<th>Compute capability:</th>
<th>1.0 and 1.1</th>
<th>1.2 and 1.3</th>
<th>2.0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Memory transactions:</td>
<td>Uncached</td>
<td>Cached</td>
<td></td>
</tr>
<tr>
<td>7 x 32B at 128</td>
<td>1 x 128B at 128</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8 x 32B at 160</td>
<td>1 x 64B at 192</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8 x 32B at 192</td>
<td>1 x 32B at 256</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8 x 32B at 224</td>
<td>1 x 128B at 128</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 x 32B at 256</td>
<td>1 x 128B at 256</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Alignment and Coalescence

Figure F-2 Examples of Strided Shared Memory Accesses for Devices of Compute Capability 2.x

Left: Linear addressing with a stride of one 32-bit word (no bank conflict).
Middle: Linear addressing with a stride of two 32-bit words (2-way bank conflicts).
Right: Linear addressing with a stride of three 32-bit words (no bank conflict).

Figure F-3 Examples of Irregular and Colliding Shared Memory Accesses for Devices of Compute Capability 2.x

Left: Conflict-free access via random permutation.
Middle: Conflict-free access since threads 3, 4, 6, 7, and 9 access the same word within bank 5.
Right: Conflict-free broadcast access (all threads access the same word).
Alignment, Coalescence and Conditionals

What can you tell me about the memory transactions performed by a warp of threads executing the following code?

**Example**

```c
__global__ void foo(int *data, int N) {
    int i,v;
    i=blockIdx.x*blockDim.x+threadIdx.x;
    if (i < N){
        if (i%2==0) v=data[i];
        else v=data[i+1];
    }
}
```
Shared memory in CUDA:

- A limited amount of memory is accessible to threads within a threadblock
- Threads cannot access shared memory from other threadblocks
- Identified by the `__shared__` qualifier

Uses:

- Communication between threads in a block
- Cache data to reduce redundant memory accesses
- User-managed cache
- Must be managed carefully to achieve high performance
Using Shared Memory in CUDA

Suppose we want to apply a simple stencil to a 1-D vector:

\[ w_i = \sum_{j=i-1}^{i+1} v_i \]

Example

```c
__global__ void sten1d(int *in, int *out) {
    int idx, v1, v2, v3;
    idx = blockIdx.x*blockDim.x+threadIdx.x;
    v1 = in[idx];
    v2 = in[idx-1];
    v3 = in[idx+1];
    v1 += v2+v3;
    out[idx] = v1;
}
```
Using Shared Memory in CUDA

- The reads and writes should be coalesced
- Each thread loads three values to compute the local sum
- Adjacent threads access some of the same values
- Multiple transactions to get the same value!
- Solution: use shared memory to avoid redundant memory transactions.

![GPU Memory Diagram](image-url)
Using Shared Memory in CUDA

Example

```c
__global__ void sten1d(int *in, int *out){
  __shared__ int tmp[BLOCK_SIZE+2];
  int gidx,lidx;
  gidx = threadIdx.x + blockIdx.x * blockDim.x;
  lidx = threadIdx.x + 1;
  tmp[lidx] = in[gidx];
  if (lidx == 1) tmp[lidx-1] = in[gidx-1];
  // Now we can access from tmp[]
  // ... or can we?
}
```
Using Shared Memory in CUDA

The answer is NO! Remember, we don’t know what order the threads will update tmp[].

Solution: All threads must synchronize after updating tmp[] but before computing the sum.

You can synchronize threads within a threadblock with __syncthreads().
Using Shared Memory in CUDA

Example

```c
__global__ void sten1d(int *in, int *out) {
    __shared__ int tmp[BLOCK_SIZE+2];
    int gidx, lidx;
    gidx = threadIdx.x + blockIdx.x * blockDim.x;
    lidx = threadIdx.x + 1;
    tmp[lidx] = in[gidx];
    if (lidx == 1) tmp[lidx-1] = in[gidx-1];
    __syncthreads();
    // Now you can finish the job!
}
```
CUDA Toolkits (4.1)

- Suppose you want to avoid writing/optimizing CUDA code
- Good news! Others may have done it already!
- Some examples of work that is already done for you:
  - Basic Linear Algebra Subpackages (cuBLAS)
  - Fast Fourier Transform (cuFFT)
  - Sparse Matrix Libraries (cuSPARSE)
  - Random Number Generator (cuRAND)

cuBLAS

- cuBLAS 4.1 on Tesla M2090, ECC on
- MKL 10.2.3, TYAN FT72-B7015 Xeon x5680 Six-Core @ 3.33 GHz

*Performance may vary based on OS ver. and motherboard config.*
cuBLAS

- cuBLAS 4.1 on Tesla M2090, ECC on
- MKL 10.2.3, TYAN FT72-B7015 Xeon x5680 Six-Core @ 3.33 GHz

*Performance may vary based on OS ver. and motherboard config.*
cuFFT Single Precision

- Measured on sizes that are exactly powers-of-2
- cuFFT 4.1 on Tesla M2090, ECC on
- MKL 10.2.3, TYAN FT72-B7015 Xeon x5680 Six-Core @ 3.33 GHz

cuFFT Double Precision

- MKL 10.2.3, TYAN FT72-B7015 Xeon x5680 Six-Core @ 3.33 GHz
- Performance may vary based on OS version and motherboard configuration
cuSPARSE Sparse Matrix x 6 Dense Vectors (csrmm)
Useful for block iterative solve schemes

- cuSPARSE 4.1 on Tesla M2090, ECC on
- MKL 10.2.3, TYAN FT72-B7015 Xeon x5680 Six-Core @ 3.33 GHz

*Performance may vary based on OS ver. and motherboard config.*
Be sure to fill out the FDI evaluation forms:

http://www.fdi.vt.edu/training/evals/index.html