GPU Programming in CUDA

Brian Marshall

Advanced Research Computing

24 September 2015
Course Contents

This week:

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

GPU = Graphics Processing Unit

GPGPU = General Purpose GPU

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

- GPU hardware is evolving rapidly
- Depending on how new your GPU is, it might not support:
  - double precision calculations
  - certain kinds of atomic operations
  - 3-D grid blocks
  - “dynamic parallelism”
  - MPI accesses to GPU memory
- “Compute capability” provides a way to identify hardware capabilities
- Possibilities are: 1.0, 1.1, 1.2, 1.3, 2.0, 3.0, 3.5, ... 5.2
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 (in time)
- Balancing workload is not straightforward when using hybrid CPU-GPU approaches
GPU for Scientific Computing

Introduction

Preliminaries

CUDA Kernels

Memory Management

Shared Memory

Streams and Events

Toolkit Overview
Hardware Overview - GPU
Hardware Overview - Architecture View for Fermi
Memory Hierarchy

- Global memory
- Per-thread off-chip 'Local' memory
- Shared memory
- Registers
- L1 and L2 cache
- Constant memory
- Texture memory
• Registers (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: \(\sim 1\) TB/s

• L2 (off-chip)

• Global memory (off-chip)
  • Accessible by all threads
  • Higher latency (400-800 cycles)
  • Throughput: \(\sim 120\) GB/s
Getting started 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
Compiling with CUDA

To compile, type:

- `nvcc -o runme MatMul.cu`

To compile for devices with compute capability 2.0:

- `nvcc -o runme -arch sm_20 MatMul.cu`
Submitting CUDA Jobs

- As long as you have GPU available, job submission is similar to other ARC resources: http://www.arc.vt.edu/hokiespeed
- HokieSpeed equipped with Fermi GPUs
- BlueRidge has four nodes equipped with Kepler K40 GPUs
- NewRiver has eight nodes equipped with Kepler K80 GPUs
The CUDA Programming Model

The CUDA Programming Model is a framework for developing parallel applications using NVIDIA GPUs. It provides a high-level programming model that allows developers to write and execute parallel programs on CUDA-enabled GPUs.

CUDA provides a parallel programming model that is based on a grid of blocks, and each block contains a grid of threads. The grid represents the overall structure of the computation, while the blocks and threads are used to parallelize the computation across the GPU.

In the diagram, we can see a grid of blocks and threads, where each block contains a grid of threads. The threads within each block are executed in parallel on the GPU, allowing for efficient computation.

CUDA also provides a memory management model that allows developers to efficiently allocate and manage memory on the GPU. This includes shared memory, which is a high-speed memory that is shared among threads within the same block.

CUDA kernels are the functions that define the computation that will be executed on the GPU. These kernels are written in a high-level programming language, such as C or C++, and are compiled into GPU-specific code at runtime.

CUDA kernels can access both global and shared memory, allowing for efficient data transfer between the CPU and GPU. CUDA also provides a toolkit overview and preliminaries that provide additional information on how to use the CUDA programming model effectively.
CUDA kernels

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

Example

```c
__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>>>(...)
```

```
<table>
<thead>
<tr>
<th>threadIdx.x</th>
<th>threadIdx.x</th>
<th>threadIdx.x</th>
<th>threadIdx.x</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 2 3 4 5 6 7</td>
<td>0 1 2 3 4 5 6 7</td>
<td>0 1 2 3 4 5 6 7</td>
<td>0 1 2 3 4 5 6 7</td>
</tr>
</tbody>
</table>

blockIdx.x = 0  blockIdx.x = 1  blockIdx.x = 2  blockIdx.x = 3
```
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

• How do we specify the number of threads and blocks for optimal performance
• Access to global memory from threads can negatively affect performance
• Large number of memory transactions are needed to hide this latency from global memory
• Increase the computation-to-memory transaction ratio by:
  • Overlap memory transfer with computation
  • Use more threads in a block which promotes reuse of data
How to Specify Thread Blocks

Goal: saturate memory bandwidth:

- Consecutive threads in same block are bundled into warps
- 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
- 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,...)
### Allocating Memory in CUDA

#### Example

```c
// Allocate memory
double *data1, *data2, *data3;
data1 = malloc(size);
cudaMalloc((void**)&data3, size);
cudaMallocHost(&data2, size);
/* Do some work */

// Release the memory
free(data1);
cudaFree(data3);
cudaFreeHost(data2);
```
Allocating Memory in CUDA

- Host and device memory are separate entities
  - Device pointers point to GPU memory
  - Host pointers point to CPU memory
- Do not free a device pointer using a host `free()`
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);
```
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

```c
/* Copy v and w from CPU to GPU */
int numThreads = 128;
int N = 256; // length of arrays
dim3 numBlocks(N/numThreads);
VecAdd<<<numBlocks, numThreads>>>(u,v,w,N)
/* Copy u from the GPU to CPU */
```

```c
__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];
}
```
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

- In Kepler all loads from global memory done in chunks of 128B
- In Kepler memory does not have have to be accessed in order of thread, earlier compute capabilities needed this
- Memory allocated using CudaMalloc is always aligned on 256B boundaries (or a multiple of this)
- Use a structure of arrays as opposed to an array of structures that can waste bandwidth
- Use `__aligned__` keyword
Alignment and Coalescence - Example

Example

```c
__global__ void kernel(...) {
    // a is an array in global memory
    // with 96 floats
    // idx is thread id
    a[idx + 32] = ... // Coalesced access
    a[idx + 64] = ... // Coalesced access
    a[idx + 5] = ... // Uncoalesced access
}
```
Alignment and Coalescence

The following conditions can cause uncoalesced loads

- Non-sequential memory access
- Sparse memory access
- Misaligned memory access
Alignment and Coalescence - Sequential and Aligned access

Address

Thread ID

128
256

0
31
Alignment and Coalescence - Aligned but Non-sequential access
Alignment and Coalescence - Unaligned Memory Access
Using Shared Memory in CUDA

Shared memory in CUDA:
- A limited amount of memory is accessible to threads within a block
- Threads cannot access shared memory from other blocks
- 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

In this example

- 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.
Using Shared Memory in CUDA

Example

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

Example

// 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);
cudaEventElapsedTime(&time, start, stop);
```
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

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
```
Atomic Operations

- Atomic operations are performed without interference from other threads
- Used to prevent race conditions
- Works for shared memory and global memory
- Serializes code execution
A Note about Constant and Texture Memory

- Texture memory is read only
- Texture memory optimized for 2D spatial locality
- This can reduce memory traffic when a warp of threads read a spatially adjacent locations
- Constant memory is read only and won’t change over the course of kernel execution
- Constant memory is optimized for broadcast
- This can reduce memory traffic when a warp of threads read the same location
CUDA Toolkits (6.5)

- 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.
cuFFTW

- 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
- Performance may vary based on OS version and motherboard configuration
What was not covered (and where to find it):

- Fortran (i.e. 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:

https://www.udacity.com/course/cs344
Questions?

Be sure to fill out the NLI survey