An Introduction to CUDA for HPC

University of Maryland, Baltimore County
April 12, 2013
Frank Willmore
willmore@tacc.utexas.edu
An Introduction to CUDA for HPC

- Some Definitions
- Comparison of CPU and GPU architectures
- CUDA programming model
- Lab exercise
- Lab exercise walk-through
What is CUDA?

- “Compute Unified Device Architecture”
- CUDA is an API for programming NVIDIA GPUs.
- CUDA can be used with C/C++ or FORTRAN (we’ll be working with the C interface).
- CUDA has a high-level/runtime API and a low-level/driver API. (We’ll be working with the runtime API).
- OpenCL has similar syntax to the driver API, usable for GPUs and CPUs (and other devices)
What is a CPU?

- CPU = Central Processing Unit
- A CPU is what we typically think of as the ‘brain’ part of the computer, or in the context of heterogeneous computing, ‘the decider’.
- Modern CPUs can have multiple cores
  - The Nehalem chips in Longhorn have four cores each
  - The Westmere chips in Lonestar have six cores each
- The CPU part of a node is the *host*
  - Processors and associated RAM
Intel Nehalem
(Longhorn nodes)
Intel Westmere
(Lonestar nodes)
What is a GPU?

- GPU = Graphics Processing Unit
- A GPU is an extension or add-on to the CPU
- GPGPU = general purpose GPU
- Each GPU has many functional units (Streaming Multiprocessors or SMs in NVIDIA devices)
  - The Tesla-class devices on Longhorn have 30 SMs
  - The Fermi-class devices on Lonestar have 14 SMs
  - Each SM is like a CPU core, but much simpler logic
- The GPU parts of a node are *devices*
  - GPU SMs and associated GRAM (graphics RAM)
NVIDIA GT200 (Longhorn nodes)
NVIDIA GF100

Fermi

(Lonestar nodes)
## Hardware Comparison
(Longhorn- and Lonestar-deployed versions)

<table>
<thead>
<tr>
<th></th>
<th>Nehalem E5540</th>
<th>Westmere X5680</th>
<th>Tesla Quadro FX 5800</th>
<th>Fermi Tesla M2070</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Functional Units</strong></td>
<td>4</td>
<td>6</td>
<td>30</td>
<td>14</td>
</tr>
<tr>
<td><strong>Speed (GHz)</strong></td>
<td>2.53</td>
<td>3.33</td>
<td>1.30</td>
<td>1.15</td>
</tr>
<tr>
<td><strong>SIMD / SIMT width</strong></td>
<td>4</td>
<td>4</td>
<td>8</td>
<td>32</td>
</tr>
<tr>
<td><strong>Instruction Streams</strong></td>
<td>16</td>
<td>24</td>
<td>240</td>
<td>448</td>
</tr>
<tr>
<td><strong>Peak Bandwidth DRAM-&gt;Chip (GB/s)</strong></td>
<td>35</td>
<td>35</td>
<td>102</td>
<td>150</td>
</tr>
</tbody>
</table>
What is a CUDA kernel?

- A CUDA kernel is a function that is called by the host or another CUDA kernel.
- A CUDA kernel executes on the device.
- A CUDA kernel is parallel code executed simultaneously by many threads.

- Mainly Tesla capability today (on Longhorn)
  - Some Fermi details, see the CUDA C Programming Guide for more
## CPU vs. GPU characteristics

<table>
<thead>
<tr>
<th>CPU</th>
<th>GPU</th>
</tr>
</thead>
<tbody>
<tr>
<td>Few computation cores</td>
<td>Many computation cores</td>
</tr>
<tr>
<td>- Supports many instruction streams, but keep few for performance</td>
<td>- Few instruction streams</td>
</tr>
<tr>
<td>More complex pipeline</td>
<td>Simple pipeline</td>
</tr>
<tr>
<td>- Out-of-order processing</td>
<td>- In-order processing</td>
</tr>
<tr>
<td>- Deep (tens of stages)</td>
<td>- Shallow (&lt; 10 stages)</td>
</tr>
<tr>
<td>- Became simpler (Pentium 4 was complexity peak)</td>
<td>- Became more complex</td>
</tr>
<tr>
<td>Optimized for serial execution</td>
<td>Optimized for parallel execution</td>
</tr>
<tr>
<td>- SIMD units less so, but lower penalty for branching than GPU</td>
<td>- Potentially heavy penalty for branching</td>
</tr>
</tbody>
</table>
Longhorn CPU architecture

Intel Nehalem processor (two sockets, four cores/socket)

- **8M L3 cache**
- **8M L3 cache**
- **48G DRAM**
Longhorn GPU architecture

Quadro FX 5800 (30 Streaming Multiprocessors)

Constant memory

Global memory
A Streaming Multiprocessor has 8 “CUDA cores”
A block of threads executes on one SM
Blocks are divided into “warps” of 32 threads
  * Each block should have a multiple of 32 threads!
Note: Lonestar’s GPUs have 32 CUDA cores per SM
Comparing Tesla versus Fermi

<table>
<thead>
<tr>
<th>GPU</th>
<th>G80</th>
<th>GT200</th>
<th>Fermi</th>
</tr>
</thead>
<tbody>
<tr>
<td>Transistors</td>
<td>681 million</td>
<td>1.4 billion</td>
<td>3.0 billion</td>
</tr>
<tr>
<td>CUDA Cores</td>
<td>128</td>
<td>240</td>
<td>512</td>
</tr>
<tr>
<td>Double Precision Floating</td>
<td>None</td>
<td>30 FMA ops / clock</td>
<td>256 FMA ops / clock</td>
</tr>
<tr>
<td>Point Capability</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Single Precision Floating</td>
<td>128 MAD ops / clock</td>
<td>240 MAD ops / clock</td>
<td>512 FMA ops / clock</td>
</tr>
<tr>
<td>Point Capability</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Special Function Units (SFUs)</td>
<td>2</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>/ SM</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Warp schedulers (per SM)</td>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>Shared Memory (per SM)</td>
<td>16 KB</td>
<td>16 KB</td>
<td>Configurable 48 KB or 16 KB</td>
</tr>
<tr>
<td>L1 Cache (per SM)</td>
<td>None</td>
<td>None</td>
<td>Configurable 16 KB or 48 KB</td>
</tr>
<tr>
<td>L2 Cache</td>
<td>None</td>
<td>None</td>
<td>768 KB</td>
</tr>
<tr>
<td>ECC Memory Support</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Concurrent Kernels</td>
<td>No</td>
<td>No</td>
<td>Up to 16</td>
</tr>
<tr>
<td>Load/Store Address Width</td>
<td>32-bit</td>
<td>32-bit</td>
<td>64-bit</td>
</tr>
</tbody>
</table>
Kernel Execution

GT200
single kernel at a time

GF100
multiple kernels simultaneously

Up to one per SM
Other Important Changes in Fermi

• ECC (Error Correction Code) memory
  – Detects and corrects soft memory errors automatically
  – Parity bits reduce memory capacity: 6 GB -> 5.25 GB
  – Definite nod to HPC, but not enabled by default

• Faster Double-Precision
  – GT200 took 8x penalty, GF100 only 2x (same as CPUs)

• Faster context switching
  – 10x improvement makes multitasking practical

• IEEE 754-2008 (floating point standard) compliance
Other Important Changes in Fermi

• Faster atomic operations
  – Ensures protected access to data
  – More hardware support and unified L2 cache
  – 20x faster than GT200

• Unified host/device address space
  – Allows full implementation of pointers in C and C++
  – Supports device-size memory allocation
    • Possible, but expensive! Reuse buffers if possible.
The CUDA programming model

- Runtime API
- Driver API

- Host thread calls CUDA functions and launches CUDA kernels
- Flow control and synchronization are the responsibility of the host thread.
CUDA Programming Model

- Data is decomposed into a hierarchy of grids and threads.

- A Grid is composed of multiple thread blocks
- A Grid can be either 1-D or 2-D (or 3-D on Fermis)
CUDA Programming Model

- A thread block contains many threads.
- A thread block can be 1-D, 2-D, or 3-D.
Hierarchy of Threads, Blocks, and Grids

- Thread -> One SIMT Lane -> Registers
- Thread Block -> SM -> Shared Memory (& L1 Cache on Fermi)
- Grid -> all SMs -> Global Memory (& L2 Cache on Fermi)

Threads control access to all memories
Compiler Flags:

• **Debugging:**
  - `-g` generates debugging information for host
  - `-G` generates debugging information for device
  - Use `cuda-gdb` to debug both host and device code
  - **More info:** "$TACC_CUDA_DIR/doc/cuda-gdb.pdf"

• **Profiling**
  - Use NVIDIA `compute-prof` to profile device code
  - **More info:** "$TACC_CUDA_DIR/compute-prof/doc"
Example: Free Volume via Widom insertion

\[ F(N,V,T) = -kT \ln Z(N,V,T) \]

\[ Z = \sum_i e^{-E_i/kT} \]

\[ \mu = \left. \frac{\partial F}{\partial N} \right|_{T,V} = -kT \left. \frac{\partial \ln Z}{\partial N} \right|_{T,V} = -kT \ln \frac{Z(N+1,V,T)}{Z(N,V,T)} = -kT \ln B_{\text{insertion}} \]

\[ B_{\text{insertion}} = \langle e^{-\frac{\psi_{\text{insertion}}}{kT}} \rangle \]

\[ FVI(x,y,z,t) \equiv e^{-\frac{\psi_{\text{repulsive}}}{kT}} \]

A GPU example: Sampling free volume
Insertion parameter as a free volume metric:

- Works with *any* forcefield model
- Well-behaved function, bounded between \([0..1)\)
- Represents solubility
- Continuity with statistical mechanics
- Easily computable (particularly with GPUs)
- Defined over *all* of space

A GPU example: Sampling free volume
A GPU example: Sampling free volume
A GPU example: Sampling free volume
A GPU example: Sampling free volume
polydimethylsiloxane
Why use GPUs for this?

• Feasability of calculation
  – Widom first formulated insertion method in 1963. For most systems, calculation was not feasible.
  – 49 years of increases in computing power / ~4 billion-fold increase per Moore’s Law (1965)
  – With the current formulation, it now takes ~one hour on a single NVIDIA Tesla M2070 ($R_{peak} = 1$TFLOP single precision) to calculate energies in a system of 1600 atoms at 1 billion ($1024 \times 1024 \times 1024$) grid points.

• GPUs excel at performing the same calculation for different values of x, y, z. For example:
  – rendering an image
  – Monte Carlo sampling
  – Stencil calculations
Lab I: My first CUDA program

• Reads from standard input
• Converts each line of text received into ALL CAPITALS
• Prints the converted line of text to standard output.
Lab I: My first CUDA program

- Compile the program
- Run the program interactively
- Run the program with a large text file
- Time program execution
- Compile program again for device emulation
- Time program execution for device emulation
Setting up the lab

- Login
- Unpack the archive
- Submit the batch job to reserve a node
- Retrieve the node name
- Connect to your node
Log in to longhorn.tacc.utexas.edu

ssh longhorn.tacc.utexas.edu

Documentation:  http://services.tacc.utexas.edu/index.php/longhorn-user-guide
User News:  http://www.tacc.utexas.edu/services/usernews/

Important System Notes:

--> To see what software packages are available, issue: "module avail"

--> Example batch job submission scripts are available in /share/doc/sge

--> Longhorn has one global high-speed Lustre file system: $SCRATCH. Users
should run jobs out of $SCRATCH (note that the "cds" alias is provided so
you can easily change to your specific $SCRATCH directory (alternatively,
you can issue "cd $SCRATCH").
Copy and unpack the archive

```bash
cp ~train00/cuda4hpc.tar .
tar -xvf cuda4hpc.tar
cd cuda4hpc
```
Submit the batch job to reserve a node

```bash
qsub qsleep
```

-- Welcome to TACC's Longhorn Visualization System, an NSF TeraGrid Resource --

--> Checking that you specified -V...
--> Checking that you specified a time limit...
--> Checking that you specified a queue...
--> Testing that the specified project type is valid...

Starting Longhorn project...

--> Checking that the minimum and maximum PE counts are the same...
--> Checking that the number of PEs requested is valid...
--> Ensuring absence of dubious h_vmem, h_data, s_vmem, s_data limits...
--> Requesting valid memory configuration (mt=31.3G)...
--> Verifying HOME file-system availability...
--> Verifying SCRATCH file-system availability...
--> Checking ssh setup...
--> Checking that you didn't request more cores than the maximum...
--> Checking that you don't already have the maximum number of jobs...
--> Checking that you don't already have the maximum number of jobs in queue development.

--> Checking that your time limit isn't over the maximum...
--> Checking available allocation...
--> Submitting job...

Your job 27546 ("sleep") has been submitted

login1%
Retrieve the name of your node

```
tail sleep-27546
```

Your number will be different.

Use your own job number.

Last line of file is name of your node
Connect to your node

ssh c210-103

Your job z 6 ("sleep") has been submitted

Your numbers will be different!

# Queue name
# project type
# runtime (hh:mm:ss) - 2 hours max

hostname

sleep 3600

TACC: Done.
c210-103.longhorn
Login1% ssh c210-103
Warning: Permanently added 'c210-103' (RSA) to the list of known hosts.
Warning: untrusted X11 forwarding setup failed: xauth key data not generated
Warning: No xauth data; using fake authentication data for X11 forwarding.
Rocks Compute Node
Rocks 5.2 (Chimichanga)
Profile built 17:40 03-Dec-2009

Kickstarted 12:45 03-Dec-2009
c210-103%

That’s it! You’re logged in!
Examine source code

cd cuda4hpc
cat shift.cu

```c
#include <stdio.h>

__device__ char d_string[65536][256];

__global__ void toUpper() {
    if ((d_string[blockIdx.x][threadIdx.x] <= 122) && (d_string[blockIdx.x][threadIdx.x]) >= 97)
        d_string[blockIdx.x][threadIdx.x] -= 32;
}

int main(int argc, char* argv[]) {
    char line[65536][256];
    int n_lines;

    for (n_lines=0; !feof(stdin); n_lines++) fgets(&line[n_lines][0], 256, stdin);
    cudaMemcpyToDevice(d_string, line, sizeof(line), 0, cudaMemcpyHostToDevice);
    toUpper<<< n_lines, 256 >>>();
    cudaMemcpyToDeviceFromSymbol(line, d_string, sizeof(line), 0, cudaMemcpyDeviceToHost);

    for (int i=0; i<n_lines; i++) printf("%s", line[i]);
}
```
Compile and run the program

module load cuda
make shift
./shift

Enter any text here. <Ctrl-D> when done.
#include <stdio.h>

__device__ char d_string[65536][256];

__global__ void toUpper() {
    if ((d_string[blockIdx.x][threadIdx.x] <= 122)
     && (d_string[blockIdx.x][threadIdx.x]) >= 97)
        d_string[blockIdx.x][threadIdx.x] -= 32;
}

int main(int argc, char* argv[]) {
    char line[65536][256];
    int n_lines;

    for (n_lines=0; !feof(stdin); n_lines++) fgets(&line[n_lines][0], 256, stdin);
    cudaMemcpyToSymbol(d_string, line, sizeof(line), 0, cudaMemcpyHostToDevice);
    toUpper<<< n_lines, 256 >>>();
    cudaMemcpyFromSymbol(line, d_string, sizeof(line), 0, cudaMemcpyDeviceToHost);
    for (int i=0; i<n_lines; i++) printf("%s", line[i]);
}
#include <stdio.h>

__device__ char d_string[65536][256];

__global__ void toUpper() {
    if (((d_string[blockIdx.x][threadIdx.x] <= 122)
         && (d_string[blockIdx.x][threadIdx.x]) >= 97)
         d_string[blockIdx.x][threadIdx.x] -= 32;
}

int main(int argc, char* argv[]) {
    char line[65536][256];
    int n_lines;

    for (n_lines=0; !feof(stdin); n_lines++) fgets(&line[n_lines][0], 256, stdin);
    cudaMemcpyToSymbol(d_string, line, sizeof(line), 0, cudaMemcpyHostToDevice);
    toUpper<<< n_lines, 256 >>>();
    cudaMemcpyFromSymbol(line, d_string, sizeof(line), 0, cudaMemcpyDeviceToHost);
    for (int i=0; i<n_lines; i++) printf("%s", line[i]);
}

#include <stdio.h>
__device__ char d_string[65536][256];

__global__ void toUpper() {
    if ((d_string[blockIdx.x][threadIdx.x] <= 122)
        && (d_string[blockIdx.x][threadIdx.x]) >= 97)
        d_string[blockIdx.x][threadIdx.x] -= 32;
}

int main(int argc, char* argv[]) {
    char line[65536][256];
    int n_lines;

    for (n_lines=0; !feof(stdin); n_lines++) fgets(&line[n_lines][0], 256, stdin);

cudaMemcpyToSymbol(d_string, line, sizeof(line), 0, cudaMemcpyHostToDevice);
toUpper<<< n_lines, 256 >>>();
cudaMemcpyFromSymbol(line, d_string, sizeof(line), 0, cudaMemcpyDeviceToHost);

    for (int i=0; i<n_lines; i++) printf("%s", line[i]);
}
#include <stdio.h>

__device__ char d_string[65536][256];

__global__ void toUpper() {
    if ((d_string[blockIdx.x][threadIdx.x] <= 122) && (d_string[blockIdx.x][threadIdx.x]) >=97)
        d_string[blockIdx.x][threadIdx.x] -= 32;
}

int main(int argc, char* argv[]) {
    char line[65536][256];
    int n_lines;

    for (n_lines=0; !feof(stdin); n_lines++) fgets(&line[n_lines][0], 256, stdin);

    cudaMemcpyToSymbol(d_string, line, sizeof(line), 0, cudaMemcpyHostToDevice);
    toUpper<<< n_lines, 256 >>>();
    cudaMemcpyFromSymbol(line, d_string, sizeof(line), 0, cudaMemcpyDeviceToHost);

    for (int i=0; i<n_lines; i++) printf("%s", line[i]);
}
Copy the result from the device

```c
#include <stdio.h>

__device__ char d_string[65536][256];

__global__ void toUpper() {
    if (((d_string[blockIdx.x][threadIdx.x] <= 122) && (d_string[blockIdx.x][threadIdx.x]) >=97))
        d_string[blockIdx.x][threadIdx.x] -= 32;
}

int main(int argc, char* argv[]) {
    char line[65536][256];
    int n_lines;

    for (n_lines=0; !feof(stdin); n_lines++) fgets(&line[n_lines][0], 256, stdin);

    cudaMemcpyToSymbol(d_string, line, sizeof(line), 0, cudaMemcpyHostToDevice);
    toUpper<<< n_lines, 256 >>>();
    cudaMemcpyFromSymbol(line, d_string, sizeof(line), 0, cudaMemcpyDeviceToHost);

    for (int i=0; i<n_lines; i++) printf("%s", line[i]);
}
```
#include <stdio.h>

__device__ char d_string[65536][256];

__global__ void toUpper() {
    if (((d_string[blockIdx.x][threadIdx.x] <= 122) && (d_string[blockIdx.x][threadIdx.x]) >=97) 
        d_string[blockIdx.x][threadIdx.x] -= 32; 
}

int main(int argc, char* argv[]) {
    char line[65536][256];
    int n_lines;

    for (n_lines=0; !feof(stdin); n_lines++) fgets(&line[n_lines][0], 256, stdin);

cudaMemcpyToSymbol(d_string, line, sizeof(line), 0, cudaMemcpyHostToDevice);
toUpper<<< n_lines, 256 >>>();
cudaMemcpyFromSymbol(line, d_string, sizeof(line), 0, cudaMemcpyDeviceToHost);

    for (int i=0; i<n_lines; i++) printf("%s", line[i]);
}
A CUDA kernel: toUpper()

```c
__global__ void toUpper() {
    if ((d_string[blockIdx.x][threadIdx.x] <= 122) && (d_string[blockIdx.x][threadIdx.x]) >= 97)
        d_string[blockIdx.x][threadIdx.x] -= 32;
}
```

- This simple CUDA kernel changes all lower case letters in a string to upper case.
- Each letter is examined simultaneously by a separate thread.
- The built-in variable `threadIdx.x` tells us which thread is referenced / which character to examine.
Launching a CUDA kernel:

```
toupper<<< n_lines, 256 >>>();
```
A set of threads (a WARP) is assigned to a multiprocessor.
32 threads = 1 warp

8 threads

One warp executes in four cycles of eight threads.
Summary

• Using GPU acceleration is beneficial only if there is enough work for the GPU to do!
  – Must amortize cost of data transfer between device and host
  – Can use streams or zero-copy transfer to improve performance (see CUDA Programming Guide for details)

• Best if there is (1) a lot of data to process, and (2) a lot of work to do for each piece of data
  – Regular distribution of work is best
  – Few conditionals, lots of FLOPs
Additional reading