ME964
High Performance Computing for Engineering Applications

Intro, GPU Computing

February 9, 2012

"The Internet is a great way to get on the net."
US Senator Bob Dole
Before We Get Started…

- Last time
  - Amdahl's law
  - Discussion on scaling of algorithms and implementations
  - Parallel computing paradigms (Flynn’s taxonomy)
  - Quick overview of common hardware topologies for parallel computing
    - Top500 & trends in HPC
  - HTC vs. HPC

- Today
  - Start GPU computing segment of the class

- Assignments
  - HW 2 due tonight, at 11:59 PM
  - Assignment 3 will be emailed to you today or early tomorrow
Acknowledgements

- A number of GPU computing slides include material developed at the University of Illinois Urbana-Champaign by Professor W. Hwu and Adjunct Professor David Kirk (the latter also former Chief Scientist at NVIDIA).
  - Slides that include material produced by professors Hwu and Kirk contain a HK-UIUC logo in the lower left corner of the slide
- Slides that include material from various NVIDIA presentations are marked with &NVIDIA
- Several other slides are lifted from other sources as indicated along the way
Parallel Programming on the GPU Card
Running Code on Parallel Computers

[one slide detour]

- You rely on compiler to figure out the parallelism in a piece of code and then map it to an underlying hardware
  - VERY hard, the holy grail in parallel computing

- You rely on parallel libraries built for a specific underlying hardware
  - Very convenient, the way to go when such libraries are available
  - Example: PETSc, leverages MPI programming and CPU clusters

- You rely on language extensions and with human interaction you facilitate the process of generating a parallel executable
  - This is the solution embraced by straight CUDA
Why Discuss GPU Computing?

- GPU Computing fast for a variety of jobs
  - Really good for data parallelism (which requires SIMD)
  - However, not impressive for task parallelism (which requires MIMD)

- It’s cheap to get a GPU card ($120 to $480)
  - High end GPUs for Scientific Computing are more like $1500

- GPUs are everywhere
  - There is incentive to produce software since there are many potential users of it…
  - NVIDIA alone sold more than 200 million units that can be user programmed

- GPU computing is not quite High Performance Computing (HPC)
  - However, it shares with HPC the aspect that they both draw on parallel programming
  - Amount of memory (max 6 GB) represents an impediment
  - However, a bunch of GPUs can together lead to a HPC cluster…
Keep this in mind…

- You should always understand the hardware that you program against
  - Be aware of the constraints you’ll be facing, deal with them smartly
  - Be aware of its strong points, capitalize on them
Layout of Typical Hardware Architecture

CPU (the "host")

GPU w/ local DRAM (the "device")
Bandwidth in a CPU-GPU System

This guy crunches numbers

Robert Strzodka, Max Plank Institute, Germany
GPU vs. CPU – Memory Bandwidth

[GB/sec]

- Tesla 8-series
- Tesla 10-series
- Tesla 20-series
- Nehalem 3 GHz
- Westmere 3 GHz
PCI-Express Latency

## Latency, DRAM Memory Access

<table>
<thead>
<tr>
<th>Year of introduction</th>
<th>Chip size</th>
<th>Slowest DRAM (ns)</th>
<th>Fastest DRAM (ns)</th>
<th>Column access strobe (CAS)/data transfer time (ns)</th>
<th>Cycle time (ns)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1980</td>
<td>64K bit</td>
<td>180</td>
<td>150</td>
<td>75</td>
<td>250</td>
</tr>
<tr>
<td>1983</td>
<td>256K bit</td>
<td>150</td>
<td>120</td>
<td>50</td>
<td>220</td>
</tr>
<tr>
<td>1986</td>
<td>1M bit</td>
<td>120</td>
<td>100</td>
<td>25</td>
<td>190</td>
</tr>
<tr>
<td>1989</td>
<td>4M bit</td>
<td>100</td>
<td>80</td>
<td>20</td>
<td>165</td>
</tr>
<tr>
<td>1992</td>
<td>16M bit</td>
<td>80</td>
<td>60</td>
<td>15</td>
<td>120</td>
</tr>
<tr>
<td>1996</td>
<td>64M bit</td>
<td>70</td>
<td>50</td>
<td>12</td>
<td>110</td>
</tr>
<tr>
<td>1998</td>
<td>128M bit</td>
<td>70</td>
<td>50</td>
<td>10</td>
<td>100</td>
</tr>
<tr>
<td>2000</td>
<td>256M bit</td>
<td>65</td>
<td>45</td>
<td>7</td>
<td>90</td>
</tr>
<tr>
<td>2002</td>
<td>512M bit</td>
<td>60</td>
<td>40</td>
<td>5</td>
<td>80</td>
</tr>
<tr>
<td>2004</td>
<td>1G bit</td>
<td>55</td>
<td>35</td>
<td>5</td>
<td>70</td>
</tr>
<tr>
<td>2006</td>
<td>2G bit</td>
<td>50</td>
<td>30</td>
<td>2.5</td>
<td>60</td>
</tr>
</tbody>
</table>

**Figure 5.13 Times of fast and slow DRAMs with each generation.** (Cycle time is defined on page 310.) Performance improvement of row access time is about 5% per year. The improvement by a factor of 2 in column access in 1986 accompanied the switch from NMOS DRAMs to CMOS DRAMs.
Parallel Computing on a GPU

- NVIDIA GPU Computing Architecture
  - Via a separate HW interface
  - In laptops, desktops, workstations, servers

- Tesla C2050 deliver 0.515 Tflops in double precision

- Multithreaded SIMT model uses application data parallelism and thread parallelism

- Programmable in C with CUDA tools
  - “Extended C”
CPU vs. GPU – Flop Rate (GFlops)

- **Tesla 8-series**
- **Tesla 10-series**
- **Tesla 20-series**
- **Westmere 3 GHz**
- **Nehalem 3 GHz**

- Single Precision
- Double Precision

GFlop/Sec

- 2003 2004 2005 2006 2007 2008 2009 2010

- 0 200 400 600 800 1000 1200

- Single Precision
- Double Precision

- Tesla 8-series
- Tesla 10-series
- Tesla 20-series
- Westmere 3 GHz
- Nehalem 3 GHz
## Key Parameters

**GPU – NVIDIA Tesla C2050**

- **Processing Cores**: 448 (8 threads)
- **Memory**: 3 GB
- **Clock speed**: 1.15 GHz
- **Memory bandwidth**: 140 GB/s
- **Floating point operations/s**: $515 \times 10^9$ Double Precision

**CPU – Intel core i7 975 Extreme**

- **Processing Cores**: 4 (8 threads)
- **Memory**: 32 KB L1 cache / core
- **Memory**: 256 KB L2 (I&D) cache / core
- **Memory**: 8 MB L3 (I&D) shared by all cores
- **Clock speed**: 3.20 GHz
- **Memory bandwidth**: 25.6 GB/s
- **Floating point operations/s**: $70 \times 10^9$ Double Precision
IBM BlueGene/L [2007]

- Entry model: 1024 dual core nodes
- 5.7 Tflop/s
- Linux OS
- Dedicated power management solution
- Dedicated IT support
- Decent options for productivity tools (debugging, profiling, etc.)
  - TotalView
- Price (2007): $1.4 million
What is the GPU so Fast?

- The GPU is specialized for compute-intensive, highly data parallel computation (owing to its graphics rendering origin)
- More transistors can be devoted to data processing rather than data caching and control flow
- Where are GPUs good: high arithmetic intensity (the ratio between arithmetic operations and memory operations)

The fast-growing video game industry exerts strong economic pressure that forces constant innovation
ALU – Arithmetic Logic Unit

- Digital circuit that performs arithmetic and logical operations
- Fundamental building block of a processing unit (CPU and GPU)

- A and B operands (the data, coming from input registers)
- F is an operator (“+”, “-”, etc.) – specified by the control unit
- R is the result, stored in output register
- D is an output flag passed back to the control unit
What is GPGPU?

- General Purpose computation using GPU in applications other than 3D graphics
  - GPU accelerates critical path of application

- Data parallel algorithms leverage GPU attributes
  - Large data arrays, streaming throughput
  - Fine-grain **SIMD** parallelism
  - Low-latency floating point (FP) computation

- Applications – see [http://GPGPU.org](http://GPGPU.org)
  - Game effects, image processing
  - Physical modeling, computational engineering, matrix algebra, convolution, correlation, sorting
A shader: set of software instructions mostly used to calculate rendering effects on graphics hardware with a good degree of flexibility

- Shaders are used to program the graphics processing unit (GPU) programmable rendering pipeline
  - Represent a set of instructions executed by a GPU thread

- Shader-programming replaced the fixed-function pipeline that allowed only common geometry transformation and pixel-shading functions

- Shaders enable customized effects
GPGPU Constraints

- Dealing with graphics API
  - Working with the corner cases of the graphics API

- Addressing modes
  - Limited texture size/dimension

- Shader capabilities
  - Limited outputs

- Instruction sets
  - Lack of Integer & bit ops

- Communication limited
  - Between pixels
  - Only gather (can read data from other pixels), but no scatter (can only write to one pixel)

Summing Up: Mapping computation problems to graphics rendering pipeline was tedious…
CUDA: Making the GPU Tick...

- “Compute Unified Device Architecture” – freely distributed by NVIDIA

- It enables a general purpose programming model
  - User kicks off batches of threads on the GPU to execute a function (kernel)

- Targeted software stack
  - Scientific computing oriented drivers, language, and tools

- Driver for loading computation programs into GPU
  - Standalone Driver - Optimized for computation
  - Interface designed for compute, graphics free, API
  - Explicit GPU memory management
CUDA Programming Model: A Highly Multithreaded Coprocessor

- The GPU is viewed as a compute device that:
  - Is a co-processor to the CPU or host
  - Has its own DRAM (device memory, or global memory in CUDA parlance)
  - Runs many threads in parallel

- Data-parallel portions of an application run on the device as kernels which are executed in parallel by many threads

- Differences between GPU and CPU threads
  - GPU threads are extremely lightweight
    - Very little creation overhead
  - GPU needs 1000s of threads for full efficiency
    - Multi-core CPU needs only a few heavy ones
GPU: Underlying Hardware

- NVIDIA nomenclature used below, reminiscent of GPU’s mission

- The hardware organized as follows:
  
  - One Stream Processor Array (SPA)…
  
    - … has a collection of Texture Processor Clusters (TPC, ten of them on C1060) …
      
      - …and each TPC has three Stream Multiprocessors (SM) …
        
        - …and each SM is made up of eight Stream or Scalar Processor (SP)
NVIDIA TESLA C1060
[Newton’s GPU cards]

- 240 Scalar Processors
- 4 GB device memory
- Memory Bandwidth: 102 GB/s
- Clock Rate: 1.3GHz
- Approx. $1,250
Keep in mind that the GPU is a SIMD device, so it works on “streams” of data
- Each “GPU thread” executes one general instruction on the stream of data that it is assigned to handle
- The NVIDIA calls this model SIMT (single instruction multiple thread)

The number crunching power comes from a vertical hierarchy:
- One Stream Processor Array (SPA)…
  - …which has a collection of Texture Processor Clusters (TPC, ten of them on Tesla C1060) …
    - …and each TPC has three Stream Multiprocessors (SM) …
    - …and each SM is made up of eight Stream or Scalar Processor (SP)

The quantum of scalability is the SM
- The more $ you pay, the more SMs you get inside your GPU
Compute Capability [of a Device] vs. CUDA Version

- “Compute Capability of a Device” refers to hardware
  - Defined by a major revision number and a minor revision number

- Example:
  - Newton’s Tesla C1060 is compute capability 1.3
  - Tesla C2050 is compute capability 2.0
  - The major revision number is up to 2 (Fermi architecture)
  - The minor revision number indicates incremental changes within an architecture class

- A higher compute capability indicates an more able piece of hardware

- The “CUDA Version” indicates what version of the software you are using to run on the hardware
  - Right now, the most recent version of CUDA, released in January 2012, is 4.1

- In a perfect world
  - You would run the most recent CUDA (version 4.1) software release
  - You would use the most recent architecture (compute capability 2.1)
Compatibility Issues

- The basic rule: the CUDA Driver API is backward, but not forward compatible
- Makes sense, the functionality in later versions increased, was not there in previous versions
NVIDIA CUDA Devices

- CUDA-Enabled Devices with Compute Capability, Number of Multiprocessors, and Number of CUDA Cores

<table>
<thead>
<tr>
<th>GeForce</th>
<th>Compute Capability</th>
<th>Number of Multiprocessors</th>
<th>Number of CUDA Cores</th>
</tr>
</thead>
<tbody>
<tr>
<td>GTX 560 Ti</td>
<td>2.1</td>
<td>8</td>
<td>384</td>
</tr>
<tr>
<td>GTX 460</td>
<td>2.1</td>
<td>7</td>
<td>336</td>
</tr>
<tr>
<td>GTX 470M</td>
<td>2.1</td>
<td>6</td>
<td>288</td>
</tr>
<tr>
<td>GTX 450, GTX 460M</td>
<td>2.1</td>
<td>4</td>
<td>192</td>
</tr>
<tr>
<td>GTX 445M</td>
<td>2.1</td>
<td>3</td>
<td>144</td>
</tr>
<tr>
<td>GTX 435M, GTX 425M, GT 420M</td>
<td>2.1</td>
<td>2</td>
<td>96</td>
</tr>
<tr>
<td>GTX 415M</td>
<td>2.1</td>
<td>1</td>
<td>48</td>
</tr>
<tr>
<td>GTX 580</td>
<td>2.0</td>
<td>16</td>
<td>512</td>
</tr>
<tr>
<td>GTX 570, GTX 480</td>
<td>2.0</td>
<td>15</td>
<td>480</td>
</tr>
<tr>
<td>GTX 470</td>
<td>2.0</td>
<td>14</td>
<td>448</td>
</tr>
<tr>
<td>GTX 465, GTX 480M</td>
<td>2.0</td>
<td>11</td>
<td>352</td>
</tr>
<tr>
<td>GTX 295</td>
<td>1.3</td>
<td>2x30</td>
<td>2x240</td>
</tr>
<tr>
<td>GTX 285, GTX 280, GTX 275</td>
<td>1.3</td>
<td>30</td>
<td>240</td>
</tr>
<tr>
<td>GTX 260</td>
<td>1.3</td>
<td>24</td>
<td>192</td>
</tr>
<tr>
<td>9800 GX2</td>
<td>1.1</td>
<td>2x16</td>
<td>2x128</td>
</tr>
<tr>
<td>GTX 250, GTX 150, 9800 GTX, 9800 GTX+, 8800 GTX, GTX 285M, GTX 280M</td>
<td>1.1</td>
<td>16</td>
<td>128</td>
</tr>
<tr>
<td>8800 Ultra, 8800 GTX</td>
<td>1.1</td>
<td>14</td>
<td>112</td>
</tr>
</tbody>
</table>
The CUDA Execution Model
GPU Computing – The Basic Idea

- The GPU is linked to the CPU by a reasonably fast connection
- The idea is to use the GPU as a co-processor
  - Farm out big parallel tasks to the GPU
  - Keep the CPU busy with the control of the execution and “corner” tasks
The CUDA Way: Extended C

- Declaration specifications:
  - global, device, shared, local, constant

- Keywords
  - threadIdx, blockIdx

- Intrinsics
  - __syncthreads

- Runtime API
  - For memory, symbol, execution management

- Kernel launch

```c
__device__ float filter[N];
__global__ void convolve (float *image) {
    __shared__ float region[M];
    ...
    region[threadIdx.x] = image[i];
    __syncthreads()
    ...
    image[j] = result;
}
```

```
// Allocate GPU memory
void *myimage = cudaMalloc(bytes)
```

```
// 100 blocks, 10 threads per block
convolve<<<100, 10>>> (myimage);
```
Example: Hello World!

```c
int main(void) {
    printf("Hello World!\n");
    return 0;
}
```

- Standard C that runs on the host
- NVIDIA compiler (nvcc) can be used to compile programs with no device code

Output, on Euler:
```
$ nvcc hello_world.cu
$ a.out
Hello World!
$  
```
Hello World! with Device Code

__global__ void mykernel(void) {
}

int main(void) {
    mykernel<<<1,1>>>();
    printf("Hello World!\n");
    return 0;
}

- Two new syntactic elements…
Hello World! with Device Code

CUDA C/C++ keyword \texttt{\_global\_} indicates a function that:

\begin{itemize}
  \item Runs on the device
  \item Is called from host code
\end{itemize}

\texttt{nvcc} separates source code into host and device components

\begin{itemize}
  \item Device functions, e.g. \texttt{mykernel()}, processed by NVIDIA compiler
  \item Host functions, e.g. \texttt{main()}, processed by standard host compiler
    \begin{itemize}
      \item \texttt{gcc, cl.exe}
    \end{itemize}
\end{itemize}
Hello World! with Device Code

mykernel<<<1,1>>>();

- Triple angle brackets mark a call from host code to device code
  - Also called a “kernel launch”
  - NOTE: we’ll return to the parameters (1,1) soon

- That’s all that is required to execute a function on the GPU…
Hello World! with Device Code

```c
__global__ void mykernel(void) {
}

int main(void) {
    mykernel<<<1,1>>>();
    printf("Hello World!\n");
    return 0;
}
```

- Actually, `mykernel()` does not do anything yet...

Output, on Euler:

```
$ nvcc hello.cu $ a.out Hello World! $
```

&NVIDIA
This is how your C code looks like

This is how the code gets executed on the hardware in heterogeneous computing
Languages Supported in CUDA

- Note that everything is done in C
  - Yet minor extensions are needed to flag the fact that a function actually represents a kernel, that there are functions that will only run on the device, etc.
    - You end up working in “C with extensions”

- FOTRAN is supported, we’ll not cover in ME964 though

- There is support for C++ programming (operator overload, new/delete, etc.)
  - Not fully supported yet
## CUDA Function Declarations

(the “C with extensions” part)

<table>
<thead>
<tr>
<th>Function Declaration</th>
<th>Executed on the:</th>
<th>Only callable from the:</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>device</strong> float myDeviceFunc()</td>
<td>device</td>
<td>device</td>
</tr>
<tr>
<td><strong>global</strong> void myKernelFunc()</td>
<td>device</td>
<td>host</td>
</tr>
<tr>
<td><strong>host</strong> float myHostFunc()</td>
<td>host</td>
<td>host</td>
</tr>
</tbody>
</table>

- **__global__** defines a kernel function, launched by host, executed on the device
  - Must return **void**
- For a full list, see CUDA Reference Manual
GPU Computing: Facts and Consequences

- You have to copy data onto the GPU and later fetch results back
  - This happens over a PCI-Express 2.0 connection

- For this to pay off, the data transfer overhead should be overshadowed by the GPU number crunching that draws on that data

- GPUs also work in asynchronous mode
  - Data transfer for future task can happen while the GPU processes current job

Consequences
1. GET THE DATA ON THE GPU AND KEEP IT THERE
2. GIVE THE GPU ENOUGH WORK TO DO
3. FOCUS ON DATA REUSE WITHIN THE GPU TO AVOID MEMORY BANDWIDTH LIMITATIONS