Parallel Computing on the GPU
GPU Computing w/ CUDA
Execution Configuration
Elements of CUDA API

September 28, 2015
Quote of the Day

“The best things in life aren’t things.”

-- Art Buchwald, Pulitzer Prize winner (1925-2007)
Before We Get Started

- Issues covered last time:
  - Big Iron HPC
  - Amdahl's Law

- Today’s topics
  - Parallel computing on GPU cards
    - Generalities
    - Execution Configuration
    - CUDA API

- Assignment:
  - HW03 – due on September 30 at 11:59 PM
    - Could be pushed two days back if I don’t make it to the slide about timing. Stay tuned for more.

- Midterm Exam: 10/09 (Friday)
  - Review on Th 10/08, at 7:15 PM, room TBA
GPU Computing with CUDA
Layout of Typical Hardware Architecture

CPU (the "host")

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

NOTE: The width of the black lines is proportional to the bandwidth.

Important slide...
Bandwidth in a CPU-GPU System

- **CPU Core**
  - 80GB/s
  - Latency: Low

- **Cache**
  - 50-100 GB/s
  - Latency: Medium

- **System Memory**
  - 8-16 GB/s
  - Latency: High

- **GPU Memory**
  - 100-240 GB/s
  - Latency: Rel. Low

- **Infiniband to Next Node**
  - 6GB/s
  - Latency: High
Bandwidth of Common Interfaces

Good-to-Know Numbers

- Almost all our applications are bound by memory speed
  - That is, most often, the execution units (cores, SPs, etc.) idle waiting for data
  - What’s important is how fast you can move data
  - Make a matter of habit to have a ballpark idea of
    - Bandwidth you are operating at
    - Latencies you are bound to experience
The Need for High Bandwidth

- Assume you have a 1 Tflops card:
  - Assume that you want to add *different* numbers and reach 1 Tflops: 1E12 ops/second
  - You need to feed 2E12 operands per second…
  - If each number is stored using 4 bytes (float), then you need to fetch 2E12*4 bytes in a second. This is 8E12 B/s, which is 8 TB/s…
  - The memory bandwidth on the GPU is in the neighborhood of 0.2 TB/s, about 40 times less than what you need
    - You haven’t taken into account that you probably want to send back the outcome of the operation that you carry out
CUDA: Making the GPU Tick…

- “Compute Unified Device Architecture” – freely distributed by NVIDIA
- When introduced it eliminated the constraints associated with GPGPU
- 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
    - Interface designed for compute, CUDA exposes a graphics-free API
  - Enables explicit GPU memory management
CUDA Programming Model: GPU as 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 (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
Fermi: Scalability through SMs
The Quantum of Scalability: SM

- GPU is a SIMD device → it works on “streams” of data
  - Each “GPU thread” executes one general instruction on the stream of data that the GPU is assigned to process
  - The NVIDIA calls this model SIMT (single instruction multiple thread)

- The number crunching power comes from a vertical hierarchy:
  - A collection of Streaming Multiprocessors (SMs)
  - Each SM has a set of 32 Scalar Processors (SPs)
    - Maxwell has 128 SPs, Kepler has 196 SPs, Fermi 2.1 had 48 SPs

- The quantum of scalability is the SM
  - The more $ you pay, the more SMs you get inside your GPU
  - Fermi can have up to 16 SMs on one GPU card
The Fermi Architecture

- Late 2009, early 2010
- 40 nm technology
- Three billion transistors
- 512 Scalar Processors (SP, “shaders”)
- 64 KB L1 cache
- 768 KB L2 uniform cache (shared by all SMs)
- Up to 6 GB of global memory
- Operates at several clock rates
  - Memory
  - Scheduler
  - Shader (SP)
- High memory bandwidth
  - Close to 200 GB/s
## My Laptop Has an NVIDIA GPU

<table>
<thead>
<tr>
<th></th>
<th>NVS 5400M</th>
<th>NVS 5200M</th>
<th>NVS 4200M</th>
<th>NVS 3100M</th>
<th>NVS 2100M</th>
</tr>
</thead>
<tbody>
<tr>
<td>CUDA Cores</td>
<td>96</td>
<td>96</td>
<td>48</td>
<td>16</td>
<td>16</td>
</tr>
<tr>
<td>PhysX capable</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>OpenCL support</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>DirectX 11 support</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>DirectCompute support</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>OpenGL 2.1 support</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Graphics Clock (MHz)</td>
<td>Up to 660</td>
<td>Up to 625</td>
<td>Up to 810</td>
<td>600</td>
<td>535</td>
</tr>
<tr>
<td>Processor Clock (MHz)</td>
<td>Up to 1320</td>
<td>Up to 1250</td>
<td>Up to 1620</td>
<td>1470</td>
<td>1230</td>
</tr>
<tr>
<td>Memory Amount</td>
<td>Up to 2 GB</td>
<td>1 GB</td>
<td>Up to 1 GB</td>
<td>Up to 512MB</td>
<td>Up to 512MB</td>
</tr>
<tr>
<td>Memory Interface</td>
<td>128 bit</td>
<td>64 bit</td>
<td>64 bit</td>
<td>64 bit</td>
<td>64 bit</td>
</tr>
<tr>
<td>CUDA compute capability</td>
<td>2.1</td>
<td>2.1</td>
<td>2.1</td>
<td>1.2</td>
<td>1.2</td>
</tr>
<tr>
<td>PCI Express 2.0 support</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

My laptop’s card: NVIDIA MVS 5200M (2 SMs w/ 48 SPs each)
# NVIDIA CUDA Devices

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

<table>
<thead>
<tr>
<th>Card</th>
<th>Compute Capability</th>
<th>Number of SMs</th>
<th>Number of SPs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tegra X1</td>
<td>5.3</td>
<td>2</td>
<td>256</td>
</tr>
<tr>
<td>TITAN X</td>
<td>5.2</td>
<td>24</td>
<td>3072</td>
</tr>
<tr>
<td>Tesla K80</td>
<td>3.7</td>
<td>2x15</td>
<td>2x2496</td>
</tr>
<tr>
<td>GTX TITAN Z</td>
<td>3.5</td>
<td>2x15</td>
<td>2x2880</td>
</tr>
<tr>
<td>Tegra K1</td>
<td>3.2</td>
<td>1</td>
<td>192</td>
</tr>
<tr>
<td>GTX 690</td>
<td>3.0</td>
<td>2x8</td>
<td>2x1536</td>
</tr>
<tr>
<td>GTX 680</td>
<td>3.0</td>
<td>8</td>
<td>1536</td>
</tr>
<tr>
<td>GTX 670</td>
<td>2.1</td>
<td>7</td>
<td>1344</td>
</tr>
<tr>
<td>GTX 590</td>
<td>2.1</td>
<td>2x16</td>
<td>2x512</td>
</tr>
<tr>
<td>GTX 560Ti</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>GTS 450, GTX 460M</td>
<td>2.1</td>
<td>4</td>
<td>192</td>
</tr>
<tr>
<td>GTX 490</td>
<td>2.0</td>
<td>2x15</td>
<td>2x480</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,</td>
<td>1.3</td>
<td>30</td>
<td>240</td>
</tr>
<tr>
<td>GTX 275</td>
<td></td>
<td></td>
<td></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>GTS 250, GTS 150,</td>
<td>1.1</td>
<td>16</td>
<td>128</td>
</tr>
<tr>
<td>9800 GTX, 9800</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>GTX+, 8800 GTS</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>512, GTX 285M,</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>GTX 280M</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8800 Ultra, 8800</td>
<td>1.0</td>
<td>16</td>
<td>128</td>
</tr>
<tr>
<td>GTX</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9800 GT, 8800 GT</td>
<td>1.1</td>
<td>14</td>
<td>112</td>
</tr>
</tbody>
</table>

**SM:** Stream Multiprocessor (the analog of a CPU core)

**SP:** Stream Processor (the analog of an ALU)
“Compute Capability of a Device” refers to hardware
  - Defined by a major revision number and a minor revision number

Example:
  - Tesla C1060 is compute capability 1.3
  - Tesla C2050 is compute capability 2.0
  - Fermi architecture is capability 2 (on Euler now)
  - Kepler architecture is capability 3
  - Titan X is capability 5.2

- 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 is 7.5

In a perfect world
  - You would run the most recent CUDA (version 7.5) software release
  - You would use the most recent architecture (compute capability 5.X)
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
# GPU Computing Applications

## Libraries and Middleware

<table>
<thead>
<tr>
<th>Libraries and Middleware</th>
<th>Libraries and Middleware</th>
</tr>
</thead>
<tbody>
<tr>
<td>CUFFT</td>
<td>CUDA</td>
</tr>
<tr>
<td>CUBLAS</td>
<td>MAGMA</td>
</tr>
<tr>
<td>CURAND</td>
<td>NPP</td>
</tr>
<tr>
<td>CUSPARSE</td>
<td>VSIPV SVM</td>
</tr>
<tr>
<td></td>
<td>OptiX</td>
</tr>
<tr>
<td></td>
<td>iray</td>
</tr>
<tr>
<td></td>
<td>MATLAB Mathematica</td>
</tr>
</tbody>
</table>

## Programming Languages

<table>
<thead>
<tr>
<th>Programming Languages</th>
<th>C</th>
<th>C++</th>
<th>Fortran</th>
<th>Java Python Wrappers</th>
<th>DirectCompute</th>
<th>Directives (e.g. OpenACC)</th>
</tr>
</thead>
</table>

## CUDA-Enabled NVIDIA GPUs

### Kepler Architecture (compute capabilities 3.x)
- **GeForce 600 Series**
- **Quadro Kepler Series**
- Tesla K20, Tesla K10

### Fermi Architecture (compute capabilities 2.x)
- **GeForce 500 Series**
- **GeForce 400 Series**
- **Quadro Fermi Series**
- Tesla 20 Series

### Tesla Architecture (compute capabilities 1.x)
- **GeForce 200 Series**
- **GeForce 9 Series**
- **GeForce 8 Series**
- **Quadro FX Series**
- **Quadro Plex Series**
- **Quadro NVS Series**
- Tesla 10 Series

---

**Entertainment**

**Professional Graphics**

**High Performance Computing**
The CUDA Execution Model
The CUDA Execution Model is Asynchronous

This is how your C code looks like

Execution of Kernel0

Execution of Kernel1

This is how the code gets executed on the hardware in heterogeneous computing. GPU calls are asynchronous…
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 too
  - There is [limited] support for C++ programming
    - Operator overloading, Templated kernels etc
## 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
CUDA, First Example

```c++
#include <cutil_inline.h>
#include <iostream>

__global__ void simpleKernel(int* data)
{
    //write something trivial to the global memory...
    data[threadIdx.x] = blockIdx.x + threadIdx.x;
}

int main()
{
    int hostArray[4], *devArray;
    //allocate memory on the device (GPU)
    cudaMalloc((void**)&devArray, sizeof(int)*4);

    //invoke GPU kernel, with one block that has four threads
    simpleKernel<<<1, 4>>>(devArray);

    //bring the result back from the GPU into the hostArray
    cudaMemcpy(&hostArray, devArray, sizeof(int)*4, cudaMemcpyDeviceToHost);

    //print out the result to confirm that things are looking good
    std::cout << "Values stored in hostArray: ";
    std::cout << hostArray[0] << ", ";
    std::cout << hostArray[1] << ", ";
    std::cout << hostArray[2] << ", ";
    std::cout << hostArray[3] << std::endl;

    //release the memory allocated on the GPU
    cudaFree(devArray);

    return 0;
}
```
First Example, Nuts and Bolts

- Here’s how you compile on euler
  ```bash
  [negrut@euler CodeBits]$ srun -p slurm_me759 --gres=gpu:1 --pty bash
  [negrut@euler01 CodeBits]$ nvcc arch=compute_20 code=sm_20 test.cu
  [negrut@euler01 CodeBits]$ ./test
  ```

- Here’s what you’d get as output
  - Shown on my machine, which ran Windows