Outline

1. Introduction to CUDA
2. Hardware
3. Software
4. Research
5. Synctium
Outline

1. Introduction to CUDA
2. Hardware
3. Software
4. Research
5. Synctium
About CUDA

- **Compute Unified Device Architecture**
- CUDA is both an Architecture and a Programming Style
- Scalable performance based on number of "cores"

![Processing flow on CUDA diagram]
Outline

1. Introduction to CUDA
2. Hardware
3. Software
4. Research
5. Synctium
- **Single Instruction Multiple Decode**
- Optimizes for dense repetitive tasks
GF100 Architecture

- **Graphics Processing Clusters**
- **Broken Into Simultaneous Multiprocessors**
What is inside a CUDA Core?

- Each core also has 1 Shader Module
Some Example Cards...

- We are using the GTS450

<table>
<thead>
<tr>
<th></th>
<th>GTX 480</th>
<th>GTX 460</th>
<th>GTS 450</th>
</tr>
</thead>
<tbody>
<tr>
<td>Designed for:</td>
<td>Maximum</td>
<td>The Gamers’ Sweet Spot</td>
<td>LAN Party Pwning</td>
</tr>
<tr>
<td>Target Resolution</td>
<td>Maximum</td>
<td>1920x1200 4xAA</td>
<td>1680x1050 4xAA</td>
</tr>
<tr>
<td>CUDA Cores</td>
<td>480</td>
<td>336</td>
<td>192</td>
</tr>
<tr>
<td>Polymorph Engines</td>
<td>15</td>
<td>7</td>
<td>4</td>
</tr>
<tr>
<td>Texture Units</td>
<td>60</td>
<td>56</td>
<td>32</td>
</tr>
<tr>
<td>Power Connectors</td>
<td>8-pin + 6-pin</td>
<td>6-pin + 6-pin</td>
<td>6-pin</td>
</tr>
<tr>
<td>TDP</td>
<td>250W</td>
<td>160W</td>
<td>106W</td>
</tr>
</tbody>
</table>
Compare with ATI

- How is this different than ATI’s Cards

- ATI Typically has more "Processors" (Really Just ALU’s)
- NVIDIA has fewer but more independant "Cores" (With Shaders)
- Often times performance is very similar
ATI vs NVIDIA

Benchmark

Starcraft II (DX9)
1920 x 1200, Ultra High Details, 0xAA

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>GTX 470</td>
<td>65.34</td>
<td>41</td>
</tr>
<tr>
<td>EVGA GTS 450 FTW SLI</td>
<td>63.39</td>
<td>40</td>
</tr>
<tr>
<td>HD 5850 1GB</td>
<td>61.72</td>
<td>41</td>
</tr>
<tr>
<td>GTS 450 SLI</td>
<td>61.22</td>
<td>37</td>
</tr>
<tr>
<td>GTX 460 768MB</td>
<td>47.72</td>
<td>30</td>
</tr>
<tr>
<td>EVGA GTS 450 FTW</td>
<td>47.39</td>
<td>24</td>
</tr>
<tr>
<td>HD 5770 1GB</td>
<td>42.99</td>
<td>27</td>
</tr>
<tr>
<td>HD 5770 Crossfire</td>
<td>42.81</td>
<td>29</td>
</tr>
<tr>
<td>GTS450 1GB</td>
<td>41.86</td>
<td>21</td>
</tr>
<tr>
<td>HD 5750 1GB</td>
<td>37.27</td>
<td>23</td>
</tr>
<tr>
<td>HD 5750 Crossfire</td>
<td>35.11</td>
<td>24</td>
</tr>
</tbody>
</table>

Frames Per Second
1. Introduction to CUDA
2. Hardware
3. Software
4. Research
5. Synctium
Software

- **Languages Supported**
  - C, C++, JAVA
  - Perl, Python, Ruby
  - Matlab, Mathematica
  - Fortran, Haskell, Lua, IDL, .NET

- Limited to NVIDIA Architectures

- Really Really Really Easy to set up...

- Many Useable standard libraries
Setting Up CUDA

- Go to: "http://developer.nvidia.com/cuda-toolkit-sdk"
- Download Drivers, CUDA Toolkit, SDK & Examples, and "Getting Started" manual
- Install Drivers, unpack the rest (automatically puts everything in the right place)
- run "make" in the C directory to set up libraries.
- run "make" in any of the example projects directories to build the project.
- If this works, CUDA is set up properly
Basic Steps To Use CUDA

- Allocate Local Memory
- Allocate GPU Memory
- Store Values in Local Memory
- Pass Values to GPU Memory
- Give GPU and instruction
- Pass Results Back
- Clean Up GPU Memory
sgemmExample

- Stands for Single Precision General Matrix Multiply
- Also Called Dense Matrix Multiply
- Requires multiplying two N x N Matrices
- Many Simple multiplies work well on highly parallel processors
Consider A X B = C Matrix Multiply

```c
/* Allocate host memory for the matrices */
h_A = (float *)malloc(n2 * sizeof(h_A[0]));
if (h_A == 0) {
    fprintf(stderr, "!!! host memory allocation error (A)\n");
    return EXIT_FAILURE;
}

h_B = (float *)malloc(n2 * sizeof(h_B[0]));
if (h_B == 0) {
    fprintf(stderr, "!!! host memory allocation error (B)\n");
    return EXIT_FAILURE;
}

h_C = (float *)malloc(n2 * sizeof(h_C[0]));
if (h_C == 0) {
    fprintf(stderr, "!!! host memory allocation error (C)\n");
    return EXIT_FAILURE;
}

/* Fill the matrices with test data */
for (i = 0; i < n2; i++) {
    h_A[i] = rand() / (float)RAND_MAX;
    h_B[i] = rand() / (float)RAND_MAX;
    h_C[i] = rand() / (float)RAND_MAX;
}
```
Memory Allocation: Device

```c
/* Allocate device memory for the matrices */
status = cublasAlloc(n2, sizeof(d_A[0]), (void**)&d_A);
if (status != CUBLAS_STATUS_SUCCESS) {
    fprintf(stderr, "!!! device memory allocation error (A)\n");
    return EXIT_FAILURE;
}
```

```c
status = cublasAlloc(n2, sizeof(d_B[0]), (void**)&d_B);
if (status != CUBLAS_STATUS_SUCCESS) {
    fprintf(stderr, "!!! device memory allocation error (B)\n");
    return EXIT_FAILURE;
}
```

```c
status = cublasAlloc(n2, sizeof(d_C[0]), (void**)&d_C);
if (status != CUBLAS_STATUS_SUCCESS) {
    fprintf(stderr, "!!! device memory allocation error (C)\n");
    return EXIT_FAILURE;
}
```

```c
/* Initialize the device matrices with the host matrices */
status = cublasSetVector(n2, sizeof(h_A[0]), h_A, 1, d_A, 1);
if (status != CUBLAS_STATUS_SUCCESS) {
    fprintf(stderr, "!!! device access error (write A)\n");
    return EXIT_FAILURE;
}
```

```c
status = cublasSetVector(n2, sizeof(h_B[0]), h_B, 1, d_B, 1);
if (status != CUBLAS_STATUS_SUCCESS) {
    fprintf(stderr, "!!! device access error (write B)\n");
    return EXIT_FAILURE;
}
```

```c
status = cublasSetVector(n2, sizeof(h_C[0]), h_C, 1, d_C, 1);
if (status != CUBLAS_STATUS_SUCCESS) {
    fprintf(stderr, "!!! device access error (write C)\n");
    return EXIT_FAILURE;
}
```
cublasSgemm('n', 'n', N, N, N, alpha, d_A, N, d_B, N, beta, d_C, N);
status = cublasGetError();
if (status != CUBLAS_STATUS_SUCCESS) {
    printf(stderr, "!!!! kernel execution error.\n");
    return EXIT_FAILURE;
}

/* Allocate host memory for reading back the result from device memory */
h_C = (float*)malloc(n2 * sizeof(h_C[0]));
if (h_C == 0) {
    printf(stderr, "!!!! host memory allocation error (C)\n");
    return EXIT_FAILURE;
}

/* Read the result back */
status = cublasGetVector(n2, sizeof(h_C[0]), d_C, 1, h_C, 1);
if (status != CUBLAS_STATUS_SUCCESS) {
    return EXIT_FAILURE;
}
Using GTS450 (192 Cores)
Matrix sizes of 256 were used
Why isn't this 192x Faster?
Energy Efficient Computing

- What does this mean?
  - Typically measured in Performance/Watt

- How do systems do this today?
  - Embedded Systems (Go Go Go .... Sleep)
  - Typical PC (Uh... We don't need that core right now)
  - Super Computer (Ummm... I don't care! More data please!)

- How can we improve this?
Power Saving Techniques

- **Power Gating**
  - Turn off sections that aren’t being used

- **Lower Supply Voltage**
  - Usually Requires slowing down the processor
  - Power is related to Voltage Squared
  - \[ P = CV^2f \]

- **Clock Gating**
  - Works in synchronous systems
  - Disables portions so less of the circuit switches states
  - Can lose speed, performance, and induce errors (less circuitry to check)

- **How can we save power without losing performance?**
  - We believe this should be done by optimizing the system for the job it is performing at any given time.
Optimize Hardware settings for Algorithms

We are looking into optimizing hardware settings based on the algorithms being run. These include:

- Memory Bandwidth
- Main Clock Frequency
- Supply Voltage
- Number of Cores
- Whatever else we can get our hands on...

A good example is computation vs memory bandwidth limited algorithms
Why a GPU?

- Allows for easy simulation of SIMD instructions
- Mass Parallel Execution (NVIDIA GTS 450 has over 190 Cores)
- Overclocking and Voltage Scalable
- Supply easily interrupted for Power measurements
- CUDA Environment and Libraries
- Good Relationship with NVIDIA

**GeForce GTS 450**

- CUDA Cores: 192
- Gfx / Proc Clock: 783 / 1566 MHz
- Memory Config: 1GB / 128-bit GDDR5
- Memory Speed: 3.6 Gbps
- Power Connectors: 6-pin
- SLI: 2-way
- Length: 8.25 inches
- Thermal: Dual Slot Fansink
- Outputs: DL-DVI, DL-DVI, mini-HDMI
Algorithms We are Looking At

- **Dense matrix multiply**
  - Brute force Matrix multiply
  - Memory Intensive
  - Computationally Simple for highly parallel architecture

- **Sparse matrix multiply**
  - Matrices typically with lots of zeros
  - Typically can be compressed
  - Less memory intensive
  - Computationally more complicated

These should allow for fairly straightforward comparisons for two algorithms that are known to have these limitations.
Outline

1. Introduction to CUDA
2. Hardware
3. Software
4. Research
5. Synctium
What is Synctium?

- A SIMD processor meant for testing sub-threshold operation.
- Very basic ALU with just Add and Multiply
- It shows performance/watt gains in sub-threshold region
Conclusion/Recap

1. Introduction to CUDA
2. Hardware
3. Software
4. Research
5. Synctium
Questions