GPGPU Computing

Yong Cao
Why Graphics Card?

➤ It’s powerful!

A quiet trend

Copyright © 2009 by Yong Cao
### Why Graphics Card?

#### It’s powerful!

<table>
<thead>
<tr>
<th>Processor</th>
<th>Processing Units</th>
<th>FLOPs per Unit</th>
<th>Clock Speed</th>
<th>Processing Power</th>
</tr>
</thead>
<tbody>
<tr>
<td>High End Qual-Core CPU</td>
<td>16 (4 per core)</td>
<td>2</td>
<td>3000 MHz</td>
<td>96 GFLOPs</td>
</tr>
<tr>
<td>NVIDIA GTX285</td>
<td>240</td>
<td>3</td>
<td>1476 MHz</td>
<td>1063 GFLOPs</td>
</tr>
<tr>
<td>ATI Radeon HD 4870</td>
<td>800</td>
<td>2</td>
<td>750 MHz</td>
<td>1200 GFLOPs</td>
</tr>
</tbody>
</table>

Copyright © 2009 by Yong Cao
Why Graphics Card?

- It’s powerful!
  - Die size 576 mm\(^2\)
  - 1.4 Billion Transistors
  - Memory width: 512bit
  - Bandwidth 142GB/s

Max board Power: 236 W
GTX285: 183 W (55nm)
Why Graphics Card?

- It’s cheap and everywhere.
- E.g. NVIDIA sold 100 Million high-end GPGPU devices.
- A 128-core GeForce 9800GTX (648 GFLOPs) is $129 on Newegg.com.
Why Graphics Card NOW?

Before:

- Two years ago, everyone is using Graphics API (Cg, GLSL, HLSL) for GPGPU programming.
- Restrict random-read (using Texture), NOT be able to random-write. (No pointer!)
Why Graphics Card NOW?

Now:

- NVIDIA released CUDA two years ago, since then
  - Thousands of CUDA software engineers
  - New job title “CUDA programmer”
  - Around 200 CUDA based technical publications!

Why?

- Standard C language
- Support Pointer! Random read and write on GPU memory.
- Work with C++, Fortran
Where’s GPU in the system
NVIDIA GPU Architecture

- Lots of ALUs
- Lots of Control
- Focus on Graphics

Applicants (Data parallel)
GPGPU Computing

NVIDIA GT200 Architecture

Thread Processing Cluster

Atomic Memory Access Control

Copyright © 2009 by Yong Cao
NVIDIA GT200 Architecture

Multi-Processor

Copyright © 2009 by Yong Cao
Execution Mode – General Computing

Host

Input Assembler

Thread Execution Manager

Parallel Data Cache

Texture

Parallel Data Cache

Texture

Parallel Data Cache

Texture

Parallel Data Cache

Texture

Parallel Data Cache

Texture

Parallel Data Cache

Texture

Parallel Data Cache

Texture

Load/store

Global Memory

NVIDIA G80 GPU

Copyright © 2009 by Yong Cao, Referencing UIUC ECE498AL Course Notes
GPGPU from Graphics Point of View

- **Graphics Processing Pipeline (on GPU)**
  - Fixed function pipeline
  - Programmable pipeline with Shaders

- **GPU Processing Model**
  - Stream computing model
3D Graphics Applications

Demos.
GPU Fundamentals: The Graphics Pipeline

- Application
- Transform & Light
- Assemble Primitives
- Rasterize
- Fragment Processing
- Final Pixels (Color, Depth)
- Video Memory (Textures)
- Render-to-texture

A simplified graphics pipeline

Copyright © 2009 by Yong Cao, Referencing SIGGRAPH 2005 Course Notes from David Luebke
Programmable Graphics Pipeline - Shaders

- **Programmable vertex processor!**
- **Programmable fragment processor!**

Copyright © 2009 by Yong Cao, Referencing SIGGRAPH 2005 Course Notes from David Luebke
GPU Pipeline: Transform

- Vertex processor (multiple in parallel)
  - Transform from “world space” to “image space”
  - Compute per-vertex lighting

Copyright © 2009 by Yong Cao, Referencing SIGGRAPH 2005 Course Notes from David Luebke
Rasterizer

- Convert geometric rep. (vertex) to image rep. (fragment)
  - Fragment = image fragment
  - Pixel + associated data: color, depth, stencil, etc.

Interpolate per-vertex quantities across pixels
Pixel / Fragment Processor

- Fragment processors (multiple in parallel)
- Compute a color for each pixel
- Optionally read colors from textures (images)
GPU Programming Model: Stream

- **Stream Programming Model**
- **Streams:**
  - An array of data units
- **Kernels:**
  - Take streams as input, produce streams at output
  - Perform computation on streams
  - Kernels can be linked together

Diagram:
- Stream
- Kernel
- Stream
- Multiple streams connected through kernels.
Why Streams?

- **Ample computation by exposing parallelism**
  - Stream expose data parallelism
    - Multiple stream elements can be processed in parallel
  - **Pipeline (task) parallelism**
    - Multiple tasks can be processed in parallel

- **Efficient communication**
  - Producer-consumer locality
  - Predictable memory access pattern
    - Optimize for throughput of all elements, not latency of one
    - Processing many elements at once allows latency hiding
Reading Material

- NVIDIA CUDA Programming Guide 2.0, Chapter One
  [http://www.nvidia.com/object/cuda_develop.html](http://www.nvidia.com/object/cuda_develop.html) and looking for documentation

- NVIDIA GeForce GTX 280 Technical Brief