Reading Assignment #3 (until Feb. 16)

Read (required):

• Programming Massively Parallel Processors book, Chapter 1 (*Introduction*)
• Programming Massively Parallel Processors book, Appendix B (*GPU Compute Capabilities*)
• OpenGL 4.0 Shading Language Cookbook, Chapter 2

Read (optional):

• OpenGL 4.0 Shading Language Cookbook, Chapter 1
• GLSL book, Chapter 7 (OpenGL Shading Language API)
Quiz #1: Feb. 16

Organization

• First 30 min of lecture
• No material (book, notes, ...) allowed

Content of questions

• Lectures (both actual lectures and slides)
• Reading assignments
• Programming assignments (algorithms, methods)
• Solve short practical examples
Programming Assignments: Schedule

Assignment #1:
• Querying the GPU (OpenGL and CUDA) due Feb 9

Assignment #2:
• Phong shading and procedural texturing (GLSL) due Mar 2

Assignment #3:
• Image Processing with (a) GLSL, and (b) CUDA due Mar 23

----- Spring Break: Apr 5 – Apr 11 ----- 

Assignment #4:
• Linear Algebra (CUDA) due Apr 20
• What you’ll learn
  • Basic graphics pipeline
  • How to use vertex shaders
  • How to use fragment shaders

• What you’ll pick up on the way
  • GLSL
  • Phong lighting and shading
  • Toon shading, procedural texturing
Programming Assignment 2 – Shaders

• What’s already there
  • Different models
  • Shader setup
    • Loading
    • Compiling
    • Variables

• Read the readme file in the framework!
Summary: three key ideas for high-throughput execution

1. Use many “slimmed down cores,” run them in parallel

2. Pack cores full of ALUs (by sharing instruction stream overhead across groups of fragments)
   – Option 1: Explicit SIMD vector instructions
   – Option 2: Implicit sharing managed by hardware

3. Avoid latency stalls by interleaving execution of many groups of fragments
   – When one group stalls, work on another group
Slimming down

Idea #1:
Remove components that help a single instruction stream run fast
Two cores (two fragments in parallel)

fragment 1

<diffuseShader>
  sample r0, v4, t0, s0
  mui r3, v6, cb0[0]
  madd r3, v1, cb0[1], r3
  madd r3, v2, cb0[2], r3
  clmp r3, r3, l(0.0), l(1.0)
  mui o0, r0, r3
  mui o1, r1, r3
  mui o2, r2, r3
  mov o3, l(1.0)
</diffuseShader>

fragment 2

<diffuseShader>
  sample r0, v4, t0, s0
  mui r3, v6, cb0[0]
  madd r3, v1, cb0[1], r3
  madd r3, v2, cb0[2], r3
  clmp r3, r3, l(0.0), l(1.0)
  mui o0, r0, r3
  mui o1, r1, r3
  mui o2, r2, r3
  mov o3, l(1.0)
</diffuseShader>
Four cores  (four fragments in parallel)
Sixteen cores  (sixteen fragments in parallel)

16 cores = 16 simultaneous instruction streams
Instruction stream sharing

But… many fragments should be able to share an instruction stream!

<diffuseShader>:

```
sample r0, v4, t0, s0
mul r3, v0, cb0[0]
madd r3, v1, cb0[1], r3
madd r3, v2, cb0[2], r3
clmp r3, r3, l(0.0), l(1.0)
mul o0, r0, r3
mul o1, r1, r3
mul o2, r2, r3
mov o3, l(1.0)
```
Add ALUs

Idea #2:

Amortize cost/complexity of managing an instruction stream across many ALUs

SIMD processing
(or SIMT, SPMD)
128 fragments in parallel

16 cores = 128 ALUs
= 16 simultaneous instruction streams
Stalls!

Stalls occur when a core cannot run the next instruction because of a dependency on a previous operation.

Texture access latency = 100’s to 1000’s of cycles

We’ve removed the fancy caches and logic that helps avoid stalls.
But we have **LOTS** of independent fragments.

**Idea #3:**
Interleave processing of many fragments on a single core to avoid stalls caused by high latency operations.
Hiding shader stalls

Time (clocks)

Frag 1 ... 8

Fetch/Decode

ALU ALU ALU ALU

ALU ALU ALU ALU

Ctx Ctx Ctx Ctx

Ctx Ctx Ctx Ctx

Shared Ctx Data
Hiding shader stalls

Time (clocks)

- Frag 1 … 8
- Frag 9 … 16
- Frag 17 … 24
- Frag 25 … 32

1 2 3 4

Frag 1 … 8

Fetch/Decode

ALU ALU ALU ALU

ALU ALU ALU ALU

1 2 3 4
Storing contexts

Pool of context storage
64 KB
Twenty small contexts

(maximal latency hiding ability)
Twelve medium contexts

![Diagram showing 12 medium contexts with ALU units at different levels](http://s09.idav.ucdavis.edu/)

SIGGRAPH 2009: Beyond Programmable Shading: http://s09.idav.ucdavis.edu/
Four large contexts

(low latency hiding ability)
Clarification

Interleaving between contexts can be managed by HW or SW (or both!)

- NVIDIA / AMD Radeon GPUs
  - HW schedules / manages all contexts (lots of them)
  - Special on-chip storage holds fragment state
- Intel MIC/Larrabee
  - HW manages four x86 (big) contexts at fine granularity
  - SW scheduling interleaves many groups of fragments on each HW context
  - L1-L2 cache holds fragment state (as determined by SW)
My chip!

16 cores

8 mul-add ALUs per core  
(128 total)

16 simultaneous instruction streams

64 concurrent (but interleaved) instruction streams

512 concurrent fragments

= 256 GFLOPs  (@ 1GHz)
My “enthusiast” chip (2009)!

32 cores, 16 ALUs per core (512 total) = 1 TFLOP (@ 1 GHz)
Summary: three key ideas for high-throughput execution

1. Use many “slimmed down cores,” run them in parallel

2. Pack cores full of ALUs (by sharing instruction stream overhead across groups of fragments)
   - Option 1: Explicit SIMD vector instructions
   - Option 2: Implicit sharing managed by hardware

3. Avoid latency stalls by interleaving execution of many groups of fragments
   - When one group stalls, work on another group
NVIDIA GeForce GTX 480 ‘core’

- Groups of 32 fragments share an instruction stream
- Up to 48 groups are simultaneously interleaved
- Up to 1536 individual contexts can be stored

Source: Fermi Compute Architecture Whitepaper
CUDA Programming Guide 3.1, Appendix G

Kayvon Fatahalian, Graphics and Imaging Architectures (CMU 15-669, Fall 2011)
NVIDIA GeForce GTX 480 “core”

- The core contains 32 functional units
- Two groups are selected each clock (decode, fetch, and execute two instruction streams in parallel)

Source: Fermi Compute Architecture Whitepaper
CUDA Programming Guide 3.1, Appendix G
**NVIDIA GeForce GTX 480 “SM”**

<table>
<thead>
<tr>
<th>Fetch/Decode</th>
<th>Execution contexts (128 KB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fetch/Decode</td>
<td></td>
</tr>
</tbody>
</table>

- “Shared” scratchpad memory (16+48 KB)

- The SM contains 32 CUDA cores
- Two warps are selected each clock (decode, fetch, and execute two warps in parallel)
- Up to 48 warps are interleaved, totaling 1536 CUDA threads

Source: Fermi Compute Architecture Whitepaper
CUDA Programming Guide 3.1, Appendix G

Kayvon Fatahalian, Graphics and Imaging Architectures (CMU 15-869, Fall 2011)
There are 15 of these things on the GTX 480:
That’s 23,000 fragments!
(or 23,000 CUDA threads!)
## Bonus slides: NVIDIA GTX 680 (2012)

NVIDIA Kepler GK104 architecture SMX unit (one “core”)

| Warp 0 | Fetch/Decode | Fetch/Decode |
| Warp 1 | Fetch/Decode | Warp Selector |
| Warp 2 | Fetch/Decode | Fetch/Decode |
| ...   | Fetch/Decode | Warp Selector |

**Warp execution contexts** (256 KB)

**“Shared” memory or L1 data cache** (64 KB)

- **SIMD function unit**, control shared across 32 units (1 MUL-ADD per clock)
- **“special” SIMD function unit**, control shared across 32 units (operations like sin/cos)
- **SIMD load/store unit** (handles warp loads/stores, gathers/scatters)

CMU 15-418, Spring 2013
Bonus slides: NVIDIA GTX 680 (2012)
NVIDIA Kepler GK104 architecture SMX unit (one “core”)

- SMX core resource limits:
  - Maximum warp execution contexts: 64 (2,048 total CUDA threads)
  - Maximum thread blocks: 16

- SMX core operation each clock:
  - Select up to four runnable warps from up to 64 resident on core (thread-level parallelism)
  - Select up to two runnable instructions per warp (instruction-level parallelism)
  - Execute instructions on available groups of SIMD ALUs, special-function ALUs, or LD/ST units
Bonus slides: NVIDIA GTX 680 (2012)
NVIDIA Kepler GK104 architecture

- 1 GHz clock
- Eight SMX cores per chip
- $8 \times 192 = 1,536$ SIMD mul-add ALUs
  - $= 3$ TFLOPs
- Up to 512 interleaved warps per chip
  - (16,384 CUDA threads/chip)
- TDP: 195 watts

192 GB/sec

Memory
256 bit interface
DDR5 DRAM
AMD Radeon HD 5870 (Cypress)

- **AMD-speak:**
  - 1600 stream processors

- **Generic speak:**
  - 20 cores
  - 16 “beefy” SIMD functional units per core
  - 5 multiply-adds per functional unit (VLIW processing)
AMD Radeon HD 5870 “core”

Groups of 64 [fragments/vertices/etc.] share instruction stream

Four clocks to execute an instruction for all fragments in a group

Source: ATI Radeon HD5000 Series: An Inside View (HPG 2010)
AMD Radeon HD 5870 “SIMD-engine”

Groups of 64 [fragments/vertices/OpenCL work items] are in a “wavefront”.

Four clocks to execute an instruction for an entire wavefront.

Source: ATI Radeon HD5000 Series: An Inside View (HPG 2010)
AMD Radeon HD 5870

There are 20 of these “cores” on the 5870: that’s about 31,000 fragments!
NVIDIA G80/GT200 Architecture

- Streaming Processor (SP)
- Streaming Multiprocessor (SM)
- Texture/Processing Cluster (TPC)

Courtesy AnandTech
NVIDIA G80/GT200 Architecture

- G80/G92: \(8 \text{ TPCs} \times (2 \times 8 \text{ SPs}) = 128 \text{ SPs}\)
- GT200: \(10 \text{ TPCs} \times (3 \times 8 \text{ SPs}) = 240 \text{ SPs}\)
- Arithmetic intensity has increased (ALUs vs. texture units)
NVIDIA GT200 GPGPU Hardware

NVIDIA Tesla 10-series
- Based on GT200 architecture
- 1 Teraflop / device
- 4GB RAM / device
- Multiple devices per node / machine
NVIDIA Fermi / GF100 Hardware

Geforce GTX 580
- 512 CUDA cores (16 SMs)
- 1.5 GB memory

Tesla 20-series
- Cards: M2070/C2070, ...
- Blades: S2050/S2070
- 3GB or 6GB / GPU, ECC memory
Names

- Compute: Fermi; product: Tesla-20 series
- Graphics: GF100 (product: GeForce GTX 480, 580, ...)

Compute capability 2.1 / 2.0; PTX ISA 3.0 / 2.x

- [http://developer.download.nvidia.com/compute/cuda/3_0/toolkit/docs/ptx_isa_2.0.pdf](http://developer.download.nvidia.com/compute/cuda/3_0/toolkit/docs/ptx_isa_2.0.pdf)

L1 and L2 caches

More CUDA cores (up to 512)

Faster double precision float performance, faster atomics, float atomics

DirectX 11 and OpenGL 4 functionality

- New shader types, scatter writes to images, ...
# NVIDIA Fermi / GF100 Stats

<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 Point Capability</td>
<td>None</td>
<td>30 FMA ops / clock</td>
<td>256 FMA ops / clock</td>
</tr>
<tr>
<td>Single Precision Floating Point Capability</td>
<td>128 MAD ops / clock</td>
<td>240 MAD ops / clock</td>
<td>512 FMA ops / clock</td>
</tr>
<tr>
<td>Special Function Units (SFUs) / SM</td>
<td>2</td>
<td>2</td>
<td>4</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>
Streaming processors are now **CUDA cores**

32 CUDA cores per Fermi streaming multiprocessor (SM)

16 SMs = 512 CUDA cores

CPU-like cache hierarchy

- L1 cache / shared memory
- L2 cache

Texture units and caches now in SM

(instead of with TPC=multiple SMs in GT200)
Dual Warp Schedulers

Warp Scheduler
Instruction Dispatch Unit
CUDA Cores (x16)
FADD
FFMA
IADD
FFMA
IADD
FFMA
FFMA
IADD
Warp Scheduler
Instruction Dispatch Unit
CUDA Cores (x16)
FADD
FFMA
IADD
MOV
ICMP
IADD
SFUs (x4)
RCP
SIN
LD/ST Units (x16)
LD
LD
ST

Markus Hadwiger, KAUST
Graphics Processor Clusters (GPC)

(instead of TPC on GT200)

4 Streaming Processors
32 CUDA cores / SM
4 SMs / GPC = 128 cores / GPC

Decentralized rasterization and geometry
  • 4 raster engines
  • 16 "PolyMorph" engines
NVIDIA Fermi / GF100 Structure

Full size

- 4 GPCs
- 4 SMs each
- 6 64-bit memory controllers (= 384 bit)
NVIDIA Fermi / GF100 Die

Full size

- 4 GPCs
- 4 SMs each
### Compute Capab. 2.0

- 1024 threads / block
- More threads / SM
- 32K registers / SM
- New synchronization functions

| Feature Support (Unlisted features are supported for all compute capabilities) | Compute Capability |
|---|---|---|---|---|---|
| | 1.0 | 1.1 | 1.2 | 1.3 | 2.0 |
| Integer atomic functions operating on 32-bit words in global memory (Section B.10) | No | | Yes | | |
| Integer atomic functions operating on 64-bit words in global memory (Section B.10) | No | Yes | | | |
| Integer atomic functions operating on 32-bit words in shared memory (Section B.10) | | Yes | | | |
| Warp vote functions (Section B.11) | | | | | |
| Double-precision floating-point numbers | No | | Yes | | |
| Floating-point atomic addition operating on 32-bit words in global and shared memory (Section B.10) | | | | Yes | |
| _ballot() (Section B.11) | | | | | |
| _thrdjoin() (Section B.5) | | | | | |
| _synchronize() (Section B.6) | | | | | |

<table>
<thead>
<tr>
<th>Technical Specifications</th>
<th>1.0</th>
<th>1.1</th>
<th>1.2</th>
<th>1.3</th>
<th>2.0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Maximum x- or y-dimension of a grid of thread blocks</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>65535</td>
</tr>
<tr>
<td>Maximum number of threads per block</td>
<td>512</td>
<td></td>
<td>1024</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Maximum x- or y-dimension of a block</td>
<td>512</td>
<td></td>
<td>1024</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Maximum z-dimension of a block</td>
<td></td>
<td>64</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Warp size</td>
<td>32</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Maximum number of resident blocks per multiprocessor</td>
<td></td>
<td></td>
<td>8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Maximum number of resident warps per multiprocessor</td>
<td>24</td>
<td>32</td>
<td>48</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Maximum number of resident threads per multiprocessor</td>
<td>768</td>
<td>1024</td>
<td>1536</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Number of 32-bit registers per multiprocessor</td>
<td>8 K</td>
<td>16 K</td>
<td>32 K</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Maximum amount of shared memory per multiprocessor</td>
<td></td>
<td>16 KB</td>
<td></td>
<td>48 KB</td>
<td></td>
</tr>
<tr>
<td>Number of shared memory banks</td>
<td>16</td>
<td></td>
<td>32</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Amount of local memory per thread</td>
<td>16 KB</td>
<td></td>
<td>512 KB</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Constant memory size</td>
<td></td>
<td></td>
<td>64 KB</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Cache working set per multiprocessor for constant memory</td>
<td></td>
<td></td>
<td></td>
<td>8 KB</td>
<td></td>
</tr>
<tr>
<td>Cache working set per multiprocessor for texture memory</td>
<td></td>
<td>Device dependent, between 6 KB and 8 KB</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Maximum width for a 1D texture reference bound to a CUDA array</td>
<td></td>
<td>8192</td>
<td></td>
<td>32768</td>
<td></td>
</tr>
<tr>
<td>Maximum width for a 1D texture reference bound to linear memory</td>
<td></td>
<td></td>
<td></td>
<td>2^27</td>
<td></td>
</tr>
<tr>
<td>Maximum width and height for a 2D texture reference bound to linear memory or a CUDA array</td>
<td></td>
<td></td>
<td>65536 x 32768</td>
<td>65536 x 65536</td>
<td></td>
</tr>
<tr>
<td>Maximum width, height, and depth for a 3D texture reference bound to linear memory or a CUDA array</td>
<td></td>
<td></td>
<td>2048 x 2048 x 2048</td>
<td>4096 x 4096 x 4096</td>
<td></td>
</tr>
<tr>
<td>Maximum number of instructions per kernel</td>
<td></td>
<td></td>
<td></td>
<td>2 million</td>
<td></td>
</tr>
</tbody>
</table>
L1 Cache vs. Shared Memory

Two different configs

- 64KB total
- 16KB shared, 48KB L1 cache
- 48KB shared, 16KB L1 cache

• Set per kernel

```c
// Device code
__global__ void MyKernel()
{
    ...
}

// Host code

// Runtime API
// cudaFuncCachePreferShared: shared memory is 48 KB
// cudaFuncCachePreferL1: shared memory is 16 KB
// cudaFuncCachePreferNone: no preference
cudaFuncSetCacheConfig(MyKernel, cudaFuncCachePreferShared)
```
Thank you.