CS 380 - GPU and GPGPU Programming
Lecture 5: GPU Architecture 4

Markus Hadwiger, KAUST
Quiz #1: Feb. 17

Organization

- First 15 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
Reading Assignment #3 (until Feb. 24)

Read (required):

• Programming Massively Parallel Processors book, Chapter 1 (*Introduction*)

• Programming Massively Parallel Processors book, Appendix B (*GPU Compute Capabilities*)

• GLSL book, Chapter 6 (Simple Shading Example)

• GLSL book, Chapter 8.1-8.3 (Shader Development)

Read (optional):

• GLSL book, Chapter 7 (OpenGL Shading Language API)
  You will need some of this information for programming assignment #2!
From Shader Code to a Teraflop:
How Shader Cores Work

Kayvon Fatahalian
Stanford University
Part 1: throughput processing

• Three key concepts behind how modern GPU processing cores run code

• Knowing these concepts will help you:
  1. Understand space of GPU core (and throughput CPU processing core) designs
  2. Optimize shaders/compute kernels
  3. Establish intuition: what workloads might benefit from the design of these architectures?
What’s in a GPU?

Heterogeneous chip multi-processor (highly tuned for graphics)
Slimming down

Idea #1:
Remove components that help a single instruction stream run fast
128 fragments in parallel

16 cores = 128 ALUs
= 16 simultaneous instruction streams
128 \text{ vertices / fragments}
primitives
CUDA threads
OpenCL work items
compute shader threads

] in parallel

primitives
vertices
fragments
Clarification

SIMD processing does not imply SIMD instructions

• Option 1: Explicit vector instructions
  – Intel/AMD x86 SSE, Intel Larrabee

• Option 2:Scalar instructions, implicit HW vectorization
  – HW determines instruction stream sharing across ALUs (amount of sharing hidden from software)
  – NVIDIA GeForce (“SIMT” warps), AMD Radeon architectures

In practice: 16 to 64 fragments share an instruction stream
But what about branches?

\[
\begin{align*}
\text{if} \ (x > 0) & \{
\begin{align*}
    y &= \text{pow}(x, \ exp);
    y &= y \times K_s;
    \text{refl} &= y + K_a;
\end{align*}
\} \text{ else } \{
\begin{align*}
    x &= 0;
    \text{refl} &= K_a;
\end{align*}
\}
\end{align*}
\]

<unconditional shader code>

<resume unconditional shader code>
But what about branches?

if (x > 0) {
    y = pow(x, exp);
    y *= Ks;
    refl = y + Ka;
} else {
    x = 0;
    refl = Ka;
}

<resume unconditional shader code>
But what about branches?

Not all ALUs do useful work!
Worst case: 1/8 performance

if (x > 0) {
    y = pow(x, exp);
    y *= Ks;
    refl = y + Ka;
} else {
    x = 0;
    refl = Ka;
}

<resume unconditional shader code>
But what about branches?

```plaintext
if (x > 0)
{
    y = pow(x, exp);
    y *= Ks;
    refl = y + Ka;
}
else
{
    x = 0;
    refl = Ka;
}
```

SIGGRAPH 2009: Beyond Programmable Shading: http://s09.idav.ucdavis.edu/
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
Hiding shader stalls

Time (clocks)

Frag 1 ... 8
Frag 9... 16
Frag 17 ... 24
Frag 25 ... 32

1
2
3
4

Frag 1 ... 8
Frag 9... 16
Frag 17 ... 24
Frag 25 ... 32

1
2
3
4

Fetch/Decode

ALU
ALU
ALU
ALU

ALU
ALU
ALU
ALU

SIGGRAPH 2009: Beyond Programmable Shading: http://s09.idav.ucdavis.edu/
Hiding shader stalls

Time (clocks)

Frag 1 ... 8
Frag 9 ... 16
Frag 17 ... 24
Frag 25 ... 32

Runnable

Stall
Hiding shader stalls

Time (clocks)

<table>
<thead>
<tr>
<th>Frag 1 … 8</th>
<th>Frag 9 … 16</th>
<th>Frag 17 … 24</th>
<th>Frag 25 … 32</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
</tbody>
</table>

Runnable

Stall
Hiding shader stalls

Time (clocks)

<table>
<thead>
<tr>
<th>Frag 1 ... 8</th>
<th>Frag 9... 16</th>
<th>Frag 17 ... 24</th>
<th>Frag 25 ... 32</th>
</tr>
</thead>
<tbody>
<tr>
<td>Runnable</td>
<td>Stall</td>
<td>Runnable</td>
<td>Stall</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

SIGGRAPH 2009: Beyond Programmable Shading: http://s09.idav.ucdavis.edu/
Throughput!

Increase run time of one group
To maximum throughput of many groups

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

Pool of context storage
64 KB
Twenty small contexts

(maximal latency hiding ability)
Twelve medium contexts
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!

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
Thank you.

Thanks for slides

- Kayvon Fatahalian