Reading Assignment #3 (until Sep 16)

Read (required):

• Programming Massively Parallel Processors book, Chapter 1 (*Introduction*)
• Programming Massively Parallel Processors book, Appendix B (*GPU Compute Capabilities*)
• OpenGL 4 Shading Language Cookbook, Chapter 2

Read (optional):

• OpenGL 4 Shading Language Cookbook, Chapter 1
• GLSL (orange) book, Chapter 7 (OpenGL Shading Language API)
Quiz #1: Sep 19

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
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)

HW or SW?
A diffuse reflectance shader

```cpp
sampler mySamp;
Texture2D<float3> myTex;
float3 lightDir;

float4 diffuseShader(float3 norm, float2 uv)
{
    float3 kd;
    kd = myTex.Sample(mySamp, uv);
    kd *= clamp( dot(lightDir, norm), 0.0, 1.0);
    return float4(kd, 1.0);
}
```

Independent, but no explicit parallelism
Compile shader

1 unshaded fragment input record

```c
sampler mySamp;
Texture2D<float3> myTex;
float3 lightDir;

float4 diffuseShader(float3 norm, float2 uv)
{
    float3 kd;
    kd = myTex.Sample(mySamp, uv);
    kd *= clamp ( dot(lightDir, norm), 0.0, 1.0);
    return float4(kd, 1.0);
}
```

1 shaded fragment output record

```
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)
```
Execute shader

```
<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)
```
Execute shader

```assembly
<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)
```
Execute shader

<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)
Execute shader

<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)
Execute shader

Fetch/Decode

ALU (Execute)

Execution Context

<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)
Execute shader

Fetch/Decode

ALU (Execute)

Execution Context

<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)
CPU-“style” cores

Fetch/Decode

Out-of-order control logic

Fancy branch predictor

Memory pre-fetcher

ALU (Execute)

Execution Context

Data cache (A big one)
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, v0, 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)

fragment 2

<diffuseShader>: 
  sample r0, v4, t0, s0 
  mui 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) 
  mui o0, r0, r3 
  mui o1, r1, r3 
  mui o2, r2, r3 
  mov o3, l(1.0)
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)
```
Recall: simple processing core

- Fetch/Decode
- ALU (Execute)
- Execution Context
Add ALUs

Idea #2:
Amortize cost/complexity of managing an instruction stream across many ALUs

SIMD processing
(or SIMT, SPMD)
Modifying the shader

Original compiled shader:
Processes one fragment using scalar ops on scalar registers

<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)
Modifying the shader

New compiled shader:
Processes 8 fragments using vector ops on vector registers

<VEC8_diffuseShader>:
VEC8_sample vec_r0, vec_v4, t0, vec_s0
VEC8_mul  vec_r3, vec_v0, cb0[0]
VEC8_madd vec_r3, vec_v1, cb0[1], vec_r3
VEC8_madd vec_r3, vec_v2, cb0[2], vec_r3
VEC8_clmp vec_r3, vec_r3, l(0.0), l(1.0)
VEC8_mul  vec_o0, vec_r0, vec_r3
VEC8_mul  vec_o1, vec_r1, vec_r3
VEC8_mul  vec_o2, vec_r2, vec_r3
VEC8_mov  vec_o3, l(1.0)
Modifying the shader

<VEC8_diffuseShader>:
VEC8_sample vec_r0, vec_v4, t0, vec_s0
VEC8_mul  vec_r3, vec_v0, cb0[0]
VEC8_madd vec_r3, vec_v1, cb0[1], vec_r3
VEC8_madd vec_r3, vec_v2, cb0[2], vec_r3
VEC8_clmp vec_r3, vec_r3, l(0.0), l(1.0)
VEC8_mul  vec_o0, vec_r0, vec_r3
VEC8_mul  vec_o1, vec_r1, vec_r3
VEC8_mul  vec_o2, vec_r2, vec_r3
VEC8_mov  vec_o3, l(1.0)
128 fragments in parallel

16 cores = 128 ALUs
= 16 simultaneous instruction streams
128 [ ] in parallel

<table>
<thead>
<tr>
<th>primitives</th>
<th>vertices</th>
<th>fragments</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image1.png" alt="Primitives" /></td>
<td><img src="image2.png" alt="Vertices" /></td>
<td><img src="image3.png" alt="Fragments" /></td>
</tr>
</tbody>
</table>

vertices / fragments
primitives
CUDA threads
OpenCL work items
compute shader threads

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

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

<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;
}
```

<unconditional shader code>

<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?

Time (clocks)

1 2 ... 8

ALU 1 ALU 2 ... ALU 8

Unconditional shader code:

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

<resume unconditional shader code>
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
Fetch/Decode
ALU
ALU
ALU
ALU
ALU
ALU
ALU
ALU

Frag 9... 16

Frag 17 ... 24

Frag 25 ... 32

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

Time (clocks)

Frag 1 ... 8
Runnable

Frag 9 ... 16

Frag 17 ... 24

Frag 25 ... 32

Stall
Hiding shader stalls

Time (clocks)

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

Runnable

Stall
Hiding shader stalls

Time (clocks)

Frag 1 ... 8
Runnable

Stall

Frag 9... 16
Runnable

Stall

Frag 17 ... 24
Runnable

Stall

Frag 25 ... 32
Runnable

Stall
Throughput!

Increase run time of one group
To maximum throughput of many groups
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)
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
End of material for Quiz #1 !
Thank you.