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 18

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 (tentative)

Assignment #1:
  • Querying the GPU (OpenGL/GLSL and CUDA)  
    due Sep 4

Assignment #2:
  • Phong shading and procedural texturing (GLSL)  
    due Sep 24

Assignment #3:
  • Image Processing with GLSL  
    due Oct 7

Assignment #4:
  • Image Processing with CUDA
  • Convolutional layers with CUDA  
    due Oct 21

Assignment #5:
  • Linear Algebra (CUDA)  
    due Nov 11
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

```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);
}
```

1 shaded fragment output record

<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)
Per-Pixel(Fragment) Lighting

Simulating smooth surfaces by calculating illumination for each fragment
Example: specular highlights (Phong illumination/shading)

Phong shading: per-fragment evaluation
Gouraud shading: linear interpolation from vertices
void main(float4 position : TEXCOORD0,
           float3 normal : TEXCOORD1,
           out float4 oColor : COLOR,
           uniform float3 ambientCol,
           uniform float3 lightCol,
           uniform float3 lightPos,
           uniform float3 eyePos,
           uniform float3 Ka,
           uniform float3 Kd,
           uniform float3 Ks,
           uniform float shiny)
{
}
float3 P = position.xyz;
float3 N = normal;
float3 V = normalize(eyePosition - P);
float3 H = normalize(L + V);

float3 ambient = Ka * ambientCol;

float3 L = normalize(lightPos - P);
float diffLight = max(dot(L, N), 0);
float3 diffuse = Kd * lightCol * diffLight;

float specLight = pow(max(dot(H, N), 0), shiny);
float3 specular = Ks * lightCol * specLight;

oColor.xyz = ambient + diffuse + specular;
oColor.w = 1;
}
Legacy Fragment Shading Unit (1)

GeForce 6 (NV40), 2004

- dynamic branching

Texture Filter
Bi / Tri / Aniso
1 texture @ full speed
4-tap filter @ full speed
16:1 Aniso w/ Trilinear (128-tap)
FP16 Texture Filtering

L2 Texture Cache

L1 Texture Cache

FP Texture Processor

FP Texture Data

FP32 Shader Unit 1

FP32 Shader Unit 2

Branch Processor

Fog ALU

Output Shaded Fragments

Shader Unit 1
4 FP Ops / pixel
Dual/Co-Issue
Texture Address Calc
Free fp16 normalize
+ mini ALU

Shader Unit 2
4 FP Ops / pixel
Dual/Co-Issue
+ mini ALU

SIMD Architecture
Dual Issue / Co-Issue
FP32 Computation
Shader Model 3.0
Example code

!!ARBfp1.0

ATTRIB unit_tc = fragment.texcoord[ 0 ];
PARAM mvp_inv[] = { state.matrix.mvp.inverse };  
PARAM constants = {0, 0.999, 1, 2};

TEMP pos_win, temp;
TEX pos_win.z, unit_tc, texture[ 1 ], 2D;

ADD pos_win.w, constants.y, -pos_win.z;
KIL pos_win.w;

MOV result.color.w, pos_win.z;

MOV pos_win.xyw, unit_tc;
MAD pos_win.xyz, pos_win, constants.a, -constants.b;

DP4 temp.w, mvp_inv[ 3 ], pos_win;
RCP temp.w, temp.w;

MUL pos_win, pos_win, temp.w;

DP4 result.color.x, mvp_inv[ 0 ], pos_win;
DP4 result.color.y, mvp_inv[ 1 ], pos_win;
DP4 result.color.z, mvp_inv[ 2 ], pos_win;

END
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?
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
What’s in a GPU?

Heterogeneous chip multi-processor (highly tuned for graphics)
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

```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);
}
```

1 shaded fragment output record

```assembly
<diffuseShader>:
sample r0, v4, t0, s0
mul r3, v0, cb0[0]
madd r3, v1, cb0[1], r3
madd r3, v2, cb0[2], r3
clamp 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

<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

<stdiohemeShader>

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)
```
CPU-“style” cores

- Fetch/Decode
- Out-of-order control logic
- ALU (Execute)
- Fancy branch predictor
- Execution Context
- Memory pre-fetcher
- 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
    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)
</diffuseShader>
```

**fragment 2**

```
<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)
</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)
```
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

<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)

New compiled shader:

Processes 8 fragments using vector ops on vector registers
Modifying the shader

Fetch/Decode

ALU 1  ALU 2  ALU 3  ALU 4
ALU 5  ALU 6  ALU 7  ALU 8
Ctx  Ctx  Ctx  Ctx
Ctx  Ctx  Ctx  Ctx
Shared Ctx Data

<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

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

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?

Time (clocks)

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

<unconditional shader code>
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?

Not all ALUs do useful work!
Worst case: 1/8 performance
But what about branches?

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

<unconditional shader code>

<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

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

1 2 3 4

Fetch/Decode

ALU ALU ALU ALU
ALU ALU ALU ALU

1 2 3 4
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

Frag 9... 16

Frag 17 ... 24

Frag 25 ... 32

1

2

3

4

Stall

Runnable
Hiding shader stalls

Time (clocks)

Frag 1 ... 8
Runnable

Frag 9... 16
Runnable

Frag 17 ... 24
Runnable

Frag 25 ... 32
Runnable

Stall

Frag 1 ... 8

Stall

Frag 9... 16

Stall

Frag 17 ... 24

Stall

Frag 25 ... 32
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

Fetch/Decode

ALU ALU ALU ALU
ALU ALU ALU ALU

1 2 3 4
5 6 7 8
9 10 11 12
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
Thank you.