CS 380 - GPU and GPGPU Programming
Lecture 5: GPU Architecture 3

Markus Hadwiger, KAUST
Reading Assignment #3 (until Sep 17)

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*)
Programming Assignments: Schedule (tentative)

Assignment #1:
- Querying the GPU (OpenGL and CUDA) due Aug 27
----- Eid Break: Aug 31 – Sep 7 ----- 

Assignment #2:
- Phong shading and procedural texturing (GLSL) due Oct 1
- *OpenGL + GLSL Tutorial: this week*

Assignment #3:
- Image Processing with (a) GLSL, and (b) CUDA due Oct 22

Assignment #4:
- Linear Algebra (CUDA) due Nov 12
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)
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

```shading_language
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
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)
```

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

<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

```
<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)
```
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
  mli 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)
  mli o0, r0, r3
  mli o1, r1, r3
  mli o2, r2, r3
  mov o3, l(1.0)
</diffuseShader>

fragment 2

<diffuseShader>
  sample r0, v4, t0, s0
  mli 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)
  mli o0, r0, r3
  mli o1, r1, r3
  mli 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
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
Modifying the shader

```
VEC8_diffuseShader:
VEC8_sample vec_r0, vec_v4, t0, vec_s0
VEC8_madd 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
Thank you.