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

Markus Hadwiger, KAUST
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)
Vertex Processor

**Begin**

**Vertex**

- Copy vertex attributes to input registers
- Fetch next instruction
- Read input-or temporary registers
- Mapping: Negation Swizzling
- Execute command
- Write to output or temp. registers
- Finished?
  - no
  - yes

**Emit Vertex**
Begin Fragment

Copy fragment attributes to Input register

Fragment Program Instructions
Input-Registers
Temporary Registers
Texture Memory
Output Registers

Fetch next instruction
Read input of temporary registers
Mapping: Negation Swizzling

Texture Instruction? (yes/no)
Calculate texture address and sample texture
Interpolate texel color

Execute instruction
Write to output or temporary registers

Finished? (yes/no)
Emit Fragment
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

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

```cpp
!!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?
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

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
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)
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

```plaintext
<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
- ALU (Execute)
- Execution Context
- Out-of-order control logic
- Fancy branch predictor
- 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)

<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)

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)

fragment 2
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)
Thank you.