CS 380 - GPU and GPGPU Programming
Lecture 4+5: GPU Architecture 3+4

Markus Hadwiger, KAUST
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*)
• 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)
Graphics Pipeline

Scene Description

Geometry Processing

Rasterization

Fragment Operations

Vertices → Primitives → Fragments → Pixels

Raster Image
Graphics Hardware

Scene Description

Programmable Pipeline

Vertex Shader

Fragment Shader

Fragment Operations

Vertices → Primitives → Fragments → Pixels

Raster Image
Direct3D 10 Pipeline (~OpenGL 3.2)

New geometry shader stage:
- Vertex -> geometry -> pixel shaders
- Stream output after geometry shader
Direct3D 11 Pipeline (~OpenGL 4.x)

New tessellation stages

- Hull shader
  (OpenGL: *tessellation control*)

- Tessellator
  (OpenGL: *tessellation primitive generator*)

- Domain shader
  (OpenGL: *tessellation evaluation*)

- Trend of adding new stages likely to continue...

- ... or full flexibility such as in Intel MIC (Larrabee) architecture?
GPU Structure Before Unified Shaders

**Vertex Processors**
- Host
- Cull/Clip/Setup
- Z-Cull
- Rasterization

**Fragment Processors**
- Texture Cache
- Fragment Crossbar

**Memory Access**
- Z-Compare and Blending
- Memory Partition
- Memory Partition
- Memory Partition
- Memory Partition

Example
NVIDIA GeForce 6/7, 2004, 2005
Legacy Vertex Shading Unit (1)

Geforce 3 (NV20), 2001

- floating point 4-vector vertex engine

- still very instructive for understanding GPUs in general

Lindholm et al., A User-Programmable Vertex Engine, SIGGRAPH 2001
### Legacy Vertex Shading Unit (2)

<table>
<thead>
<tr>
<th>Vertex Attribute Register</th>
<th>Conventional Per-vertex Parameter</th>
<th>Conventional Per-vertex Parameter Command</th>
<th>Conventional Component Mapping</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Vertex position</td>
<td>glVertex</td>
<td>x, y, z, w</td>
</tr>
<tr>
<td>1</td>
<td>Vertex weights</td>
<td>glVertexWeightEXT</td>
<td>w, 0, 0, 1</td>
</tr>
<tr>
<td>2</td>
<td>Normal</td>
<td>glNormal</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>Primary color</td>
<td>glColor</td>
<td>r, g, b, a</td>
</tr>
<tr>
<td>4</td>
<td>Secondary color</td>
<td>glSecondaryColorEXT</td>
<td>r, g, b, 1</td>
</tr>
<tr>
<td>5</td>
<td>Fog coordinate</td>
<td>glFogCoordEXT</td>
<td>f, 0, 0, 1</td>
</tr>
<tr>
<td>6</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>7</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>8</td>
<td>Texture coord 0</td>
<td>multiTexCoordARB(GL_TEXTURE0...)</td>
<td>s, t, r, q</td>
</tr>
<tr>
<td>9</td>
<td>Texture coord 1</td>
<td>multiTexCoordARB(GL_TEXTURE1...)</td>
<td>s, t, r, q</td>
</tr>
<tr>
<td>10</td>
<td>Texture coord 2</td>
<td>multiTexCoordARB(GL_TEXTURE2...)</td>
<td>s, t, r, q</td>
</tr>
<tr>
<td>11</td>
<td>Texture coord 3</td>
<td>multiTexCoordARB(GL_TEXTURE3...)</td>
<td>s, t, r, q</td>
</tr>
<tr>
<td>12</td>
<td>Texture coord 4</td>
<td>multiTexCoordARB(GL_TEXTURE4...)</td>
<td>s, t, r, q</td>
</tr>
<tr>
<td>13</td>
<td>Texture coord 5</td>
<td>multiTexCoordARB(GL_TEXTURE5...)</td>
<td>s, t, r, q</td>
</tr>
<tr>
<td>14</td>
<td>Texture coord 6</td>
<td>multiTexCoordARB(GL_TEXTURE6...)</td>
<td>s, t, r, q</td>
</tr>
<tr>
<td>15</td>
<td>Texture coord 7</td>
<td>multiTexCoordARB(GL_TEXTURE7...)</td>
<td>s, t, r, q</td>
</tr>
</tbody>
</table>

#### Code examples

- **DP4**  
  \[ o[HPOS].x, c[0], v[OPOS] ; \]

- **MUL**  
  \[ R1, R0.zxyw, R2.yzxw ; \]

- **MAD**  
  \[ R1, R0.yzxw, R2.zxyw, -R1; \]  
  *swizzling!*
Vector instruction set, very few instructions; no branching yet!

<table>
<thead>
<tr>
<th>OpCode</th>
<th>Full Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOV</td>
<td>Move</td>
<td>vector -&gt; vector</td>
</tr>
<tr>
<td>MUL</td>
<td>Multiply</td>
<td>vector -&gt; vector</td>
</tr>
<tr>
<td>ADD</td>
<td>Add</td>
<td>vector -&gt; vector</td>
</tr>
<tr>
<td>MAD</td>
<td>Multiply and add</td>
<td>vector -&gt; vector</td>
</tr>
<tr>
<td>DST</td>
<td>Distance</td>
<td>vector -&gt; vector</td>
</tr>
<tr>
<td>MIN</td>
<td>Minimum</td>
<td>vector -&gt; vector</td>
</tr>
<tr>
<td>MAX</td>
<td>Maximum</td>
<td>vector -&gt; vector</td>
</tr>
<tr>
<td>SLT</td>
<td>Set on less than</td>
<td>vector -&gt; vector</td>
</tr>
<tr>
<td>SGE</td>
<td>Set on greater or equal</td>
<td>vector -&gt; vector</td>
</tr>
<tr>
<td>RCP</td>
<td>Reciprocal</td>
<td>scalar-&gt; replicated scalar</td>
</tr>
<tr>
<td>RSQ</td>
<td>Reciprocal square root</td>
<td>scalar-&gt; replicated scalar</td>
</tr>
<tr>
<td>DP3</td>
<td>3 term dot product</td>
<td>vector-&gt; replicated scalar</td>
</tr>
<tr>
<td>DP4</td>
<td>4 term dot product</td>
<td>vector-&gt; replicated scalar</td>
</tr>
<tr>
<td>LOG</td>
<td>Log base 2</td>
<td>miscellaneous</td>
</tr>
<tr>
<td>EXP</td>
<td>Exp base 2</td>
<td>miscellaneous</td>
</tr>
<tr>
<td>LIT</td>
<td>Phong lighting</td>
<td>miscellaneous</td>
</tr>
<tr>
<td>ARL</td>
<td>Address register load</td>
<td>miscellaneous</td>
</tr>
</tbody>
</table>
Fast Forward to Programm. Fragment Shading

Core OpenGL Fragment Texturing & Coloring

< 1999

NVIDIA Proprietary

Courtesy Mark Kilgard
Fast Forward to Programm. Fragment Shading

NV10 OpenGL Fragment Texturing & Coloring

GeForce 256, 1999

Courtesy Mark Kilgard
Fast Forward to Programm. Fragment Shading

NV20 OpenGL Fragment Texturing & Coloring

GeForce 3, 2001

Courtesy Mark Kilgard
Fast Forward to Programm. Fragment Shading

NV30 OpenGL Fragment Texturing & Coloring

GeForce FX (5), 2003

Courtesy Mark Kilgard
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**

**FP Texture Processor**

**L1 Texture Cache**

**FP32 Shader Unit 1**
- 4 FP Ops / pixel
- Dual/Co-Issue
- Texture Address Calc
- Free fp16 normalize + mini ALU

**FP32 Shader Unit 2**
- 4 FP Ops / pixel
- Dual/Co-Issue
- + mini ALU

**Branch Processor**

**Fog ALU**

**Output Shaded Fragments**

**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.matrx.mvpInverse };  
PARAM constants = {0, 0.999, 1, 2};

TEMP pos_w n, temp;

TEX pos_w n.z, unit_tc, texture[ 1 ], 2D;

ADD pos_w n.w, constants.y, -pos_w n.z;
ADD pos_w n.w;

MOV result.color.w, pos_w n.z;

MOV pos_w n.xw, unit_tc;
MAD pos_w n.xyz, pos_w n, constants.a, -constants.b;

DP4 temp.w, mvp_inv[ 3 ], pos_w n;
RCP temp.w, temp.w;
MUL pos_w n, pos_w n, temp.w;

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

END

Vertex Processor

```
Vertex Program Instructions
Input-Registers
Temporary Registers
Output-Registers

Begin Vertex

1. Copy vertex attributes to input registers
2. Fetch next instruction
3. Read input- or temporary registers
4. Mapping: Negation Swizzling
5. Execute command
6. Write to output or temp. registers

Finished?

no

yes

Emit Vertex
```
**Fragment Processor**

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

### Flowchart
- **Begin Fragment**
  - Copy fragment attributes to Input register
  - Fetch next instruction
  - Read input of temporary registers
  - Mapping: Negation Swizzling
- **Calculate texture address and sample texture**
  - Interpolate texel color
  - Execute instruction
  - Write to output or temporary registers
  - Write to output registers

- **Finished?**
  - Yes: Emit Fragment
  - No: Repeat

---

**Write to output or temporary registers**

**Texture Instruction?**

**Yes:**
- Texture Instruction

**No:**
- Finished?

---

**Copy fragment attributes to Input register**

---

**Texture Instruction?**

**Yes:**
- Texture Instruction

**No:**
- Finished?

---

**Finished?**

**Yes:**
- Emit Fragment

**No:**
- Repeat
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;
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
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

```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

```plaintext
<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)
```
<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>:

```plaintext
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

```plaintext
<diffuseShader>:
sample r0, v4, t0, s0
mul r3, v0, cb0[0]
madd r3, v1, cb0[1], r3
madd r3, v2, cb0[2], r3
c1mp 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

```
<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)

<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

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

<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, 1(0.0), 1(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, 1(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.