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

Markus Hadwiger, KAUST
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
OpenGL Tutorial

With Peter Rautek
Monday, Sept. 9, 14:30 – 16:00
Bldg 9, Room 3138

Optional attendance, but highly encouraged if OpenGL is new for you!
Graphics Pipeline

Scene Description → Geometry Processing → Rasterization → Fragment Operations → Raster Image

Vertices → Primitives → Fragments → Pixels
Geometry Processing

Transformation
- Multiplication with Modelview and Projection Matrix

Per-Vertex Lighting
- Per-Vertex Local Illumination (Blinn/Phong)

Primitive Assembly
- Geometric Primitives (Points, Lines, Triangles)

Clipping, Perspect.Divide
- Clip Space To Screen Space

Vertices

Primitives
Rasterization

Geometry Processing → Rasterization → Fragment Operations

Polygon Rasterization
Texture Fetch
Texture Application

Decomposition of primitives into fragments
Interpolation of texture coordinates
Combination of primary color with texture color

Primitives → Fragments
Combination of primary color with texture color

Fragment Operations

Geometry Processing → Rasterization → Fragment Operations

- Alpha Test: Discard all fragments within a certain alpha range
- Stencil Test: Discard a fragment if the stencil buffer is set
- Depth Test: Discard all occluded fragments
- Alpha Blending: Combination of primary color with texture color
Graphics Pipeline

Scene Description

Programmable Pipeline

Vertex Shader

Fragment Shader

Fragment Operations

Vertices → Primitives → Fragments → Pixels

Raster Image
Graphics pipeline architecture
Performs operations on vertices, triangles, fragments, and pixels

1. Input: vertices in 3D space + connectivity
2. Vertex processing stage computes where vertices appear on screen given a camera position
3. Group vertices into triangles positioned on screen
4. Fragment generation creates one fragment for each pixel covered by the triangle

Fragment processing colors the fragments based on the surface characteristics at this pixel
Output image pixels contain color of the closest fragment at each pixel

Courtesy Kayvon Fatahalian, CMU
Direct3D 10 Pipeline (~OpenGL 3.2)

New geometry shader stage:
- Vertex -> geometry -> pixel shaders
- Stream output after geometry shader

Courtesy David Blythe, Microsoft
Direct3D 11 Pipeline (~OpenGL 4.x)

New tessellation stages

- Hull shader
  (OpenGL: tessellation control)

- Tessellator
  (OpenGL: tessellation primitive generator)

- Domain shader
  (OpenGL: tessellation evaluation)

- In future versions, there might be yet more stages, but for some time now all additions were outside this pipeline:
  - Compute shaders
  - Vulkan
  - Ray tracing cores
Direct3D 12 Pipeline (and Later Updates)

First version 2015 (Windows 10)
New (March 2018): DXR (DX12 Ray-Tracing)
Some Vulkan commands specify geometric objects to be drawn or computational work to be performed, while others specify state controlling how objects are handled by the various pipeline stages, or control data transfer between memory organized as images and buffers. Commands are effectively sent through a processing pipeline, either a graphics pipeline or a compute pipeline.
**Initial state**
The state when a command buffer is first allocated. The command buffer may be reset back to this state from any of the executable, recording, or invalid states. Command buffers in the initial state can only be moved to recording or freed.

**Recording state**
`vkBeginCommandBuffer` changes the state from initial to recording. Once in the recording state, `vkCmd*` commands can be used to record to the command buffer.

**Executable state**
`vkEndCommandBuffer` moves a command buffer state from recording to executable. Executable command buffers can be submitted, reset, or recorded to another command buffer.

**Pending state**
Queue submission changes the state from executable to pending, in which applications must not attempt to modify the command buffer in any way. The state reverts back to executable when current executions complete, or to invalid.

**Invalid state**
Some operations will transition the command buffer into the invalid state, in which it can only be reset or freed.
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

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)

### Input attributes

<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>glMultiTexCoordARB(GL_TEXTURE0...)</td>
<td>s,t,r,q</td>
</tr>
<tr>
<td>9</td>
<td>Texture coord 1</td>
<td>glMultiTexCoordARB(GL_TEXTURE1...)</td>
<td>s,t,r,q</td>
</tr>
<tr>
<td>10</td>
<td>Texture coord 2</td>
<td>glMultiTexCoordARB(GL_TEXTURE2...)</td>
<td>s,t,r,q</td>
</tr>
<tr>
<td>11</td>
<td>Texture coord 3</td>
<td>glMultiTexCoordARB(GL_TEXTURE3...)</td>
<td>s,t,r,q</td>
</tr>
<tr>
<td>12</td>
<td>Texture coord 4</td>
<td>glMultiTexCoordARB(GL_TEXTURE4...)</td>
<td>s,t,r,q</td>
</tr>
<tr>
<td>13</td>
<td>Texture coord 5</td>
<td>glMultiTexCoordARB(GL_TEXTURE5...)</td>
<td>s,t,r,q</td>
</tr>
<tr>
<td>14</td>
<td>Texture coord 6</td>
<td>glMultiTexCoordARB(GL_TEXTURE6...)</td>
<td>s,t,r,q</td>
</tr>
<tr>
<td>15</td>
<td>Texture coord 7</td>
<td>glMultiTexCoordARB(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!
Legacy Vertex Shading Unit (3)

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

From Primitive Assembly
- DrawPixels
- Bitmap
- Point Rasterization
- Line Rasterization
- Polygon Rasterization
- Bitmap Rasterization
- Pixel Rectangle Rasterization

Conventional Texture Fetching
- Texture Unit 0
- Texture Unit 1

Texture Environment Application
- Texture Unit 0
- Texture Unit 1

Color Sum

Fog

Coverage Application

To fragment processing

Courtesy Mark Kilgard

NVIDIA Proprietary
Fast Forward to Programm. Fragment Shading

NV10 OpenGL Fragment Texturing & Coloring

GeForce 256, 1999

NVIDIA Proprietary

Courtesy Mark Kilgard
Fast Forward to Programm. Fragment Shading

NV20 OpenGL Fragment Texturing & Coloring

GeForce 3, 2001

NVidia Proprietary

Courtesy Mark Kilgard
Fast Forward to Programm. Fragment Shading

NV30 OpenGL Fragment Texturing & Coloring

GeForce FX (5), 2003

NVIDIA Proprietary

Courtesy Mark Kilgard
**Vertex Processor**

- **Begin Vertex**: Copy vertex attributes to input registers
- **Vertex Program Instructions**
- **Input-Registers**
- **Temporary Registers**
- **Output-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?
- Interpolate texel color
- Execute instruction
- Write to output or temporary registers
- Finished?

Emit Fragment
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

```cpp
<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**

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

**FP32 Shader Unit 2**

**Branch Processor**

**Fog ALU**

**Output Shaded Fragments**

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

```glsl
!!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
```
Thank you.