GPU Shading and Rendering

Shading Compilers

Avi Bleiweiss

ATI Research Silicon Valley
Overview

• GPU evolving in fast pace
• *Shader Model 3.0*
  – Extended resources
  – Dynamic flow control
• Multi GPU systems
  – PCI Express bus
Shading Compilers

- Evidently divisible
- High level challenges
  - Encapsulate description
  - Virtualize GPU processors
- Scheduling, optimization
  - Intimate hardware
Outline

- Hiding shading languages
- Multi GPU shader partition
- Remapping GPU processors
- Source level debugger
  - Ashli shading technology
Outline

- Hiding shading languages
- Multi GPU shader partition
- Remapping GPU processors
- Source level debugger
Representation

- Content management
  - Single representation
- Ties to graphics API
- Higher level abstraction
  - Microsoft *Effect* format
Effect Format

- Parameters and functions
- Techniques and passes
- Any shading language
- Ashli for Effect - AshliFX
  - Multi lingual Effect compiler
AshliFX

Application

AshliFX

Assembly/Metadata → Shader → State → Technique/Pass → Parameter → XML → Ashli

To Graphics API

Effect Format

Effect Format
AshliFX (cnt’d)

- Implicit language binding
- Extracted shader filtering
  - Avoid semantics, annotations
- Metadata to Effect state
- Effect state observer
  - Graphics API neutral
// cartoon.fx

#pragma language GLSL

varying vec3 eyeNormal;

void mainVS()
{
    gl_Position = ftransform();
    eyeNormal = gl_NormalMatrix * gl_Normal;
}

const vec3 diffuse = vec3(1.0, 0.0, 0.0);

uniform vec4 scale = vec4(1.0, 2.0, 3.0, 4.0);

void mainPS()
{
    vec3 normal;
    vec3 color;
    float illum;
    vec3 lightDirection = vec3(1.0, 0.0, 1.0);

    lightDirection = normalize(lightDirection);
    normal = normalize(eyeNormal);
    vec3 hlf = normalize(lightDirection + vec3(0.0, 0.0, 1.0));
    illum = clamp(dot(normal, lightDirection), 0.0, 1.0);
    color = clamp(diffuse * illum + 0.2) +
            pow(max(dot(hlf, normal), 0.0), 16.0) * 0.01;
    color = scale.xyz * floor(color * 4.0) * 0.25;

    gl_FragColor = vec4(color, 1.0);
}

technique BumpReflect0 {
    pass p0 {
        VertexShader = compile vs_2_0 mainVS();
        ZEnable = true;
        ZWriteEnable = true;
        ZFunc = Always;
        CullMode = None;

        PixelShader = compile ps_2_0 mainPS();
    }
}
// texturedphong.fx (only shown pixel parameters and entry point)

float4 fV Ambient <
    string UName = "fV Ambient";
    string UIDName = "Color";
    bool UVVisible = true;
>
float4 fV Diffuse <> = float4(0.89, 0.89, 0.89, 1.00);
float4 fV Specular <> = float4(0.49, 0.49, 0.49, 1.00);
float fSpecularPower <> = float(25.00);
texture base_Tex <string ResourceName = "Textures\Fieldstone.tga">;

sampler2D baseMap = sampler_state {
    Texture = [base Tex];
    AddressU = WRAP;
    AddressV = WRAP;
};

struct PS_INPUT {
    float2 TexCoord;
    float3 ViewDirection;
    float3 LightDirection;
    float3 Normal;
};

float4 TexturedPhongPS( PS_INPUT Input ) : COLOR0 {
    float3 fV LightDirection = normalize(Input.LightDirection);
    float3 fV Normal = normalize(Input.Normal);
    float fNDotL = dot(fV Normal, fV LightDirection);
    float fV Reflection = normalize((fV Normal) * (fVNormal) - fV LightDirection);
    float3 fV ViewDirection = normalize(Input.ViewDirection);
    float fR DotV = max(0.0f, dot(fV Reflection, fV ViewDirection));
    float4 fV BaseColor = tex2D(baseMap, Input.TexCoord);
    float4 fV TotalAmbient = fV Ambient * fV BaseColor;
    float4 fV TotalDiffuse = fVDiffuse * fV BaseColor;
    float4 fV TotalSpecular = fV Specular * pow(fR DotV, fSpecularPower);
    return(saturate(fV TotalAmbient + fV TotalDiffuse + fV TotalSpecular));
}

technique TexturedPhong {
    pass Pass0 {
        VertexShader = compile vs_2_0 TexturedPhongVS();
        PixelShader = compile ps_2_0 TexturedPhongPS();
    }
}

// Auto-generated GLSL file - do not edit

// Pixel shader for program name: texturedphong

#include "GLSLoperator3.h"

uniform vec3 fV EyePosition, 0.0;
uniform vec4 fV Ambient, 0.0;
uniform vec4 fV Specular, 0.0;
uniform vec4 fV Diffuse, 0.0;
uniform float fSpecularPower, 0.0;
uniform sampler2D baseMap, 0.0;

void main() {
    vec3 fV LightDirection;
    vec3 fV Normal;
    float fNDotL,
        fV Reflection;
    vec3 fV ViewDirection;
    float fR DotV;
    vec4 fV BaseColor;
    vec4 fV TotalAmbient;
    vec4 fV TotalDiffuse;
    vec4 fV TotalSpecular;
    fV LightDirection = normalize(gl_TexCoord[2]);
    fV Normal = normalize(gl_TexCoord[3]);
    fNDotL = dot(fV Normal, fV LightDirection);
    fV Reflection = normalize(subtract(multiply(multiply(2.0f, fV Normal), fNDotL), fV LightDirection));
    fV ViewDirection = normalize(gl_TexCoord[1]);
    fR DotV = max(0.0, dot(fV Reflection, fV ViewDirection));
    fV BaseColor = texture2D(baseMap, 0.0, gl_TexCoord[0]);
    fV TotalAmbient = multiply(fV Ambient, 0.0, fV BaseColor);
    fV TotalDiffuse = multiply(fV Diffuse, 0.0, fV BaseColor);
    fV TotalSpecular = multiply(fV Specular, 0.0, power(fR DotV, fSpecularPower));
    gl_FragData[0] = clamp(add(fV TotalAmbient, add(fV TotalDiffuse, fV TotalSpecular));
}
Maya Effect
Streaming

• GPU incentive domains
• Auto generated description
  – Target platform seamless
• AshliDI -
  – Digital imaging streaming
AshliDI

- High performance
- Detail and quality
  - Extended dynamic range
- Stream based interface
  - Image tree, feedback
AshliDI (cnt’d)

To Graphics API
Geometry  Texture  State

Image Feedback  Image Tree

Assembly/Metadata  Shader

Effect Format

AshliDI

Ashli

AshliFX
Image Tree

- Directed graph, no cycles
- Nodes - image operators
  - Attributes: region of interest
- Source images - leaves
Effect Mapping

- Tree to Effect transform
  - Constitutes rendering format
- Techniques - sub trees
  - Nodes onto passes, pipe state
- Seamless language binding
Sample

- Single node
  - Gaussian filter (5x5)
  - Region of interest

```c
#define KERNEL_SIZE 5
float gaussian1mp(float2 f, float stddev)
{
    float PI = 3.141592653589;
    return (1.0 / sqrt(2.0*PI*stddev)) * exp(-(f.x*f.x + f.y*f.y) / (2.0*stddev));
}

float4 gaussian(float2 pos : VPOS,
                uniform float3 cntrl,
                uniform sampler img) : COLOR0
{
    float4 col = float4(0.0,0.0,0.0);
    float weight = 0;
    float delta = 4.0 / KERNEL_SIZE;
    float2 filter = float2(-2.0,-2.0);
    float2 init = float2(pos.x - cntrl.x*(floor(KERNEL_SIZE/2)),
                        pos.y - cntrl.y*(floor(KERNEL_SIZE/2)));
    float2 pixel = init;
    float samples = KERNEL_SIZE*KERNEL_SIZE;
    float samp=0;
    float filter_cur = 0;
    while(samp < samples) {
        filter_cur = gaussian1mp(filter, cntrl);
        weight += filter_cur;
        col += filter_cur * tex2D(img, pixel);
        pixel += delta;
        if((ceil(sample, KERNEL_SIZE) == 0) {
            filter.x = -2;
            filter.y = -2;
            pixel = init;
        } else {
            samp = samp + 1;
        }
    }
    return (col / weight);
}
```
Rendering

- Predominant stream gather
- Intermediate image results
  - Previous render-to-texture
- 2D rendering API
  - Extensible to 3D for volumes
- Feedback storage stream
Outline

- Hiding shading languages
- Multi GPU shader partition
- Remapping GPU processors
- Source level debugger
Multi GPU

- Multi GPU affordable
  - PCI Express reaching 4GB/sec
- Image, time based partition
  - Adaptive tiling more scalable
- Geometry, textures replicated
  - Vertex limited lower gains
Pipe Modality

- Vertex, pixel separate entities
  - Self resourced
- Concurrency higher on pixel
- Single amplification source
- Vertex turning into Geometry
Pipe Modality (cnt’d)
Geometry Modality

• Input, collection of vertices
  – Tessellation shader refines
• Geometry shader
  – Primitive input, output topology
• Multiplicity at top, mid pipe
• Inter shader communication
Load Balance

- Shared resources
  - Shader task scheduling
- Pipe dynamics
  - More degrees of interaction
- Walkthrough example
  - Motion blurred, displaced geometry
Displacement

- Triangular control mesh
  - Recursively refined

- Parametric space displacement
  - Neighbors for derivatives
  - Fine level of detail

Courtesy Brian Sharp 2000
Motion Blur

- Motion blur on geometry shader
  - Aperture triangle pair
  - Samples interposed in hull
Distribution

- Geometry critical path
- Pipe behavior -
  - Shader level macro threads
  - Shader results exposed
- GPU fed in pipe manner
  - Single copy scene description
Two GPUs pairing options:

1. **GPU #0**
   - **tessellation**
   - **GPU #1**
   - **geometry+pixel**

2. **GPU #0**
   - **tessellation+geometry**
   - **GPU #1**
   - **pixel**

3. **GPU #0**
   - **tessellation+pixel**
   - **GPU #1**
   - **geometry”

*inter GPU copy*
Programming

- Automate shader partition
  - Task based
- Compilers figure shader cost
  - Amplification factor, copy
- Evolve multi GPU API
  - Simple, no user intervention
Outline

- Hiding shading languages
- Multi GPU shader partition
- Remapping GPU processors
- Source level debugger
Shader Model 3.0

- Consistent instruction set
  - Vertex, pixel little adversity
- Vertex texture fetch
- Retargeting pixel onto vertex
  - Seemingly underutilized vertex
- Pixel exploits higher parallelism
Retargeting

• Vertex, pixel resource match
  – 4 vs. 16 samplers, respectively
• Automate processor remapping
• Ashli *vertex-to-pixel* conversion
  – Exploits multipass
• Analyze performance trade offs
Render to Vertex Buffer

- Vertex streams as textures
- Vertex format
  - Attributes of any type
  - Packed and Unpacked
    - Vertex storage format

```
struct VertexInput {
    float4 pos : POSITION;
    float3 normal : NORMAL;
    float color : COLOR0;
    float2 tex0 : TEXCOORD0;
};
```
Storage Format

- Contiguous vertex
  - Addressing: base, attribute stride
  - Per component fetch
- Padded vertex
  - Four component IEEE float
  - Single attribute pointer
Inputs

- Vertex and pixel inputs
  - 16 vs. 10, respectively
- Vertex buffer fetch
  - Pixel texture access
- Mapping criteria
  - vertex inputs + samplers <= 16
Outputs

- Vertex and pixel outputs
  - 12 vs. 4, respectively
- Ashli incorporates
  - Pixel *Virtual Color Outputs*
- Color outputs exceeding cap
  - Code segmentation

<table>
<thead>
<tr>
<th>Vertex Output</th>
<th>Pixel Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>Position</td>
<td>C0</td>
</tr>
<tr>
<td>Color</td>
<td>C1</td>
</tr>
<tr>
<td>SecondaryColor</td>
<td>C2</td>
</tr>
<tr>
<td>TexCoord0</td>
<td>C3</td>
</tr>
<tr>
<td>TexCoord1</td>
<td>C4</td>
</tr>
<tr>
<td>TexCoord2</td>
<td>C5</td>
</tr>
<tr>
<td>TexCoord3</td>
<td>C6</td>
</tr>
<tr>
<td>TexCoord4</td>
<td>C7</td>
</tr>
<tr>
<td>TexCoord5</td>
<td>C8</td>
</tr>
<tr>
<td>TexCoord6</td>
<td>C9</td>
</tr>
<tr>
<td>TexCoord7</td>
<td>C10</td>
</tr>
<tr>
<td>Fog</td>
<td>C11</td>
</tr>
</tbody>
</table>
Passes

- Two pass rendering
- Vertex-to-pixel
  - Texture vertex fetch
  - Processing
  - Output mapping
- *Passthrough* vertex
  - Inverse output mapping
Sample

- HLSL vertex shader
  - Vertex texture
- Ashli emits detached shaders
  - Vertex-to-pixel, passthrough
- Metadata
  - Inverse output mapping
Sample (cnt’d)

```c
struct VertexInput {
    float4 pos : POSITION;
    float2 tex : TEXCOORD0;
};

struct VertexOutput {
    float4 pos : POSITION;
    float2 energy : TEXCOORD1;
};

float4x4 mWorldViewProjection;
float3x3 mWorldInverseTranspose;
sampler shadow;

VertexOutput main(VertexInput vi) {
    VertexOutput vo;
    vo.energy = 1.0f - tex2Dlod(shadow, float4(vi.tex, 0, 1));
    vo.pos = mul(vi.pos, mWorldViewProjection);
    return vo;
}

begin fragment
    t 0 -1 0 0
    s 0 0 0 0 0 0
    s 1 0 0 0 0 0
    s 2 0 0 0 0 0
    c 3 -1 mWorldViewProjection_0_0[0] 0 0 0 0
    c 2 -1 mWorldViewProjection_0_0[1] 0 0 0 0
    c 1 -1 mWorldViewProjection_0_0[2] 0 0 0 0
    c 0 -1 mWorldViewProjection_0_0[3] 0 0 0 0
    o 0 -1 .C0_
    o 1 -1 .C4_
end

begin vertex
    a 0 -1 _Vertex_
    a 9 -1 _TexCoord1_
    o 0 -1 _Position_
    t 1 -1 _T1_
end
```

---

**SIGGRAPH 2005**
Performance

• Two pass overhead
  – Deployment and recirculation

• Speedup observed
  – Larger mesh size
  – Higher compute to fetch ratio

• Operate on vertex collection
Outline

- Hiding shading languages
- Multi GPU shader partition
- Remapping GPU processors
- Source level debugger
Tools

• Debugging increasingly important
  – Long, complex shaders

• Microsoft Visual Studio .NET
  – High level and assembly
  – File/line # and pixel area
  – No direct hardware
Tools (cnt’d)

- Shadesmith
  - Fragment assembly, on hardware
  - Register watch, inline editing
  - Platform dependent
- Source high level - Ashli
Ashli

- Language orthogonal
- Inspecting and editing code
  - Less so for hardware savvy
- Runs on graphics hardware
  - Pixel/Fragment shader
- Visual validation
API

- File/line # break points
- Debugger exposure
  - Add/remove break point
  - Continue, single step
  - Query current break point
Visual Validation

rendered image @ first break point

rendered image of complete shader
Break Point

• Two pass
  – Output substitution
  – Replace lhs with color output

• Degenerated tree nodes
  – Break point to root

• Break point delimited program
  – Valid sub tree
Flow Control

- Break point inside conditional
  - Replace lhs inside if (and else)
  - Replace lhs before conditional
- Break point in a loop
  - Conditional unrolling
  - Count set to a cap
Priorities

- Runs on hardware
- Tailored to audience
  - Content creator or
  - Hardware intimate
- Performance not critical
Summary

• Domain specific streaming
  – Hiding shading languages
• Multi GPU load distribution
  – Task based, proper API
• GPU processor virtualization
• Debugger, seriously taken
• Ashli/AshliFX on multi systems
  – 32/64 Windows and Linux, Mac OS X

• Link, contact:
  – devrel@ati.com

• Acknowledgement:
  – Raja Koduri, Evan Hart, Joshua Barczak, Nikki Lukas