General Purpose computation on GPUs

Liangjun Zhang

2/23/2005
Outline

- Interpretation of GPGPU
- GPU Programmable interfaces
- GPU programming sample: ‘Hello, GPGPU’
- More complex programming
- GPU essentials, opportunity
Interpretation of GPGPU
GPU: Graphics Hardware

Recent Performance Trends

Multiplies per second

GFLOPS

NVIDIA NV30, 35, 40
ATI R300, 360, 420
Pentium 4

What is GPGPU?

- General Purpose Computation on GPUs
  - The GPU is a graphics-specific processor
  - Useful for general computation

- Parallel processor
- 32-bit float support, even better
- Flexible programming
GPU Programmable interfaces
GPU Pipeline

- Fixed Pipeline -- > Programmable

Vertex Data → Tessellation → Vertex Processing → Geometry Processing → Pixel Processing → Pixel Rendering

- Vertex Shader, Kernel
- Pixel Shader (Fragment Processor, Kernel)
Programmable Stream Model

- Kernel

Parallel Stream Model

Input Stream
(Framebuffer, Texture) → Pixel Shader → Output Stream
(Framebuffer, Texture)
The input of pixel shader

Input: Fragment

Attributes

<table>
<thead>
<tr>
<th>Color</th>
<th>R</th>
<th>G</th>
<th>B</th>
<th>A</th>
</tr>
</thead>
<tbody>
<tr>
<td>Position</td>
<td>X</td>
<td>Y</td>
<td>Z</td>
<td>W</td>
</tr>
<tr>
<td>Texture coordinates</td>
<td>X</td>
<td>Y</td>
<td>[Z]</td>
<td>-</td>
</tr>
<tr>
<td>Texture coordinates</td>
<td>X</td>
<td>Y</td>
<td>[Z]</td>
<td>-</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

32 bits = float
16 bits = half

Input: Texture Image

- Each element of texture is 4D vector
- Textures can be “square” or rectangular (power-of-two or not)

From http://www.cis.upenn.edu/~suvenkat/700
The output of pixel shader

- updated Color
GPU programming sample
-’Hello GPGPU’
GPU programming sample
- ‘Hello GPGPU’
Algorithm

- A simple post-process edge detection filter.

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>-1/8</td>
<td>-1/8</td>
<td>-1/8</td>
</tr>
<tr>
<td>-1/8</td>
<td>1</td>
<td>-1/8</td>
</tr>
<tr>
<td>-1/8</td>
<td>-1/8</td>
<td>-1/8</td>
</tr>
</tbody>
</table>
static const char *edgeFragSource =
"half4 edges(half2 coords : TEX0,                                  
    uniform sampler2D texture) : COLOR                 
{"                                                              
    static const half offset = 1.0 / 512.0;                    
    half4 c  = tex2D(texture, coords);                             
    half4 bl = tex2D(texture, coords + half2(-offset, -offset));   
    half4 l  = tex2D(texture, coords + half2(-offset,       0));   
    half4 tl = tex2D(texture, coords + half2(-offset,  offset));   
    half4 t  = tex2D(texture, coords + half2(      0,  offset));   
    half4 ur = tex2D(texture, coords + half2( offset,  offset));   
    half4 r  = tex2D(texture, coords + half2( offset,       0));   
    half4 br = tex2D(texture, coords + half2( offset, -offset));   
    half4 b  = tex2D(texture, coords + half2(      0, -offset));   
"    // scale by 8 to brighten the edges                        
"    return 8 * (c + -0.125 * (bl + l + tl + t + ur + r + br + b)); 
"}
"
Why move the data from FB to texture?
GPGPU Programming Concepts

1. Texture = Array
2. Fragment Program = Computational Kernel.
3. One-to-one Pixel to Texel Mapping:
   a) Data-Dimensioned Viewport, and
   b) Orthographic Projection.
4. Viewport-Sized Quad = Data Stream Generator.
5. Copy To Texture = feedback.
Source Code

- Want to take a look at the source code?
- helloGPGPU.cpp
GPGPU programming Environments

- High Level Programming VS Assemble
- DirectX9 + HLSL
  - High Level Shading Language
- OpenGL + GLSL
  - OpenGL Shading Language
- OpenGL + cgToolKit
- Some useful extensions
  - Glew, glee
More complex programming

- Multipass Rendering
- Render to Texture
- Floating point texture/PBuffer
Streaming Model

Input Stream
(Framebuffer, Texture)

Intermediate stream

Output Stream
(Framebuffer, Texture)
A streaming ray tracer
Multi pass rendering

Performance Bottlenecks
- Intermediate stream (Framebuffer) → Texture
- glCopyTexImage2D()
Render To Texture
- PBuffer

From GDC2002 -- OpenGL Render-to-Texture, NVidia
What is Dynamic Texturing?

- The creation of texture maps “on the fly” for use in real time.

Simplified view:

Loop:

1. Render an image.
2. Create a texture from that image.
3. Use the texture as you would a static texture.
Challenge of Dynamic Texturing

- Performance Bottlenecks

- Simplified view:
  
  Loop:
  
  1. Render an image.
  2. Create a texture from that image.
  3. Use the texture as you would a static texture.

*Step 2 is primary bottleneck but 1 and 3 can be relevant as well.*
Methods for Creating the Texture

How to get rendered image into a texture?

- `glReadPixels()` → `glTexImage*()`?
  Slow.

- `glCopyTexImage*()`
  Better.

- `glCopyTexSubImage*()`
  Even Better.

- Render Directly to Texture
  Eliminates “texture copy” – potentially optimal
Rendering Directly to a Texture

- Not a core part of OpenGL 1.3, but ARB extensions make this possible on most GPUs.

- Required Extensions:
  - WGL_ARB_extensions_string
  - WGL_ARB_render_texture
  - WGL_ARB_pbuffer
  - WGL_ARB_pixel_format
Rendering to a Texture: overview

- Basic Idea: Allow a p-buffer to be bound as a texture
- 1. Create a texture object
- 2. Create a “Render Texture” (i.e. the pbuffer)
- 3. Loop as necessary:
  - Make the pbuffer the current rendering target
  - Render an image
  - Make the window the current rendering target
  - Bind the pbuffer to the texture object
  - Use the texture object as you would any other
  - Release the pbuffer from the texture object
- 4. Clean Up
Creating the Texture Object

Just as you would for a regular texture -- no need to specify the actual texture data

```c
// Create a render texture object
glGenTextures( 1, &render_texture );
glBindTexture( GL_TEXTURE_2D, render_texture );

// Texture parameters
glTexParameteri( GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR_MIPMAP_LINEAR );
glTexParameteri( GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR );

// Texture wrapping
glTexParameteri( GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP_TO_EDGE );
glTexParameteri( GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP_TO_EDGE );
```
Creating the Pbuffer

- Get a valid device context
  
  ```c
  HDC hdc = wglGetCurrentDC();
  ```

- Choose a pixel format
  
  ```c
  wglChoosePixelFormat();
  ```

- Create the pbuffer
  
  ```c
  HPBUFFER hbuf = wglCreatePbufferARB( hdc, fid, width, height, pfattr );
  ```

- Get a rendering context for the pbuffer
  
  ```c
  hbufglr = wglCreateContext( hbufdc );
  ```
Rendering to the Texture

- Can be done anytime once the creation of the pbuffer is complete
  - Initialization function, display loop, every 10 frames, etc.

- Must make the rendering context for the pbuffer current using wglMakeCurrent:

  wglMakeCurrent( hbufdc, hbufglrc );
  Issue OpenGL drawing commands
  wglMakeCurrent( hwindc, hwinglrc );
Binding the pbuffer to the texture object

- Call wglBindTexImageARB to bind the pbuffer to the texture object.
- After binding the texture object…
Floating Point Texture / PBuffer

- Numerical Computation
  - High Accuracy Requirements
Floating Point Texture

- `glTexImage2D(GL_TEXTURE_2D, ..., GL_RGB, GL_FLOAT, texData);`
Floating Point Pbuffer

```
HPBUFFER hbuf = wglCreatePbufferARB(hdc, fid, width, height, pfattr);
if (WGLEW_NV_float_buffer)
{
    pfAttribs.push_back(WGL_PIXEL_TYPE_ARB);
    pfAttribs.push_back(WGL_TYPE_RGBA_ARB);

    pfAttribs.push_back(WGL_FLOAT_COMPONENTS_NV);
    pfAttribs.push_back(true);
}
else
{
    pfAttribs.push_back(WGL_PIXEL_TYPE_ARB);
    pfAttribs.push_back(WGL_TYPE_RGBA_FLOAT_ATI);
}
```
Quiz 1 😊

- Vector + Vector
Solution: Vector + Vector

- Input: How to represent the input data
- Floating Point Pbuffer
- Kernel
- Output: Render to Texture
Quiz 2

- Matrix * Vector
Matrix Representations

http://www15.in.tum.de/Research/Publications/LinAlg
Solution: Matrix * Vector

- Input: How to represent the input data
- Floating Point Rendering
- Multipass rendering
- Making use of Vector + Vector
- Output: Render to Texture
GPU essentials

- SI MD model, Parallelism
  - Single instruction multiple data
  - PS can process 16 pixels simultaneously

- Limitation
  - Hardware not optimal
  - Software environment immature
Wealth of applications Already

Data Analysis
Voronoi Diagrams
Motion Planning
Geometric Optimization

Particle Systems
Force-field simulation
Molecular Dynamics
Graph Drawing

Physical Simulation
Conjugate Gradient

Matrix Multiplication

Database queries
Sorting and Searching
Range queries

Signal Processing

... and graphics too !!

From http://www.cis.upenn.edu/~suvenkat/700
Opportunities

• Current hardware not optimal
  – Incredible opportunity for architectural innovation

• Current software environment immature
  – Incredible opportunity for reinventing parallel computing software

Acknowledge

- Thanks for all the materials I used in this presentation.
Question?

Thanks!