We will discuss two different programming models:

- graphics-style (first)
- GPGPU-style (second)

Note: both use the same underlying architecture (for example GeForce 8800)

Overview

Graphics-style
- parallelism exposed as fragments
- programming in CG, GLSL, HLSL
- utilizes the graphics sub-system of GPUs
- scatter operations (calculations affecting many targets) not possible

GPGPU-style
- parallelism exposed as threads
- programmed in CUDA (NVIDIA), CTM (AMD-ATI)
- does not utilize the graphics sub-system of GPUs
- provides a more elaborate application programmer interface (API)
- allows scatter operations
Graphics Pipeline: Overview

Graphics Hardware Pipeline

Vertex Stage

Vertex values:
- position, color, texture coordinate(s), normal vector

Operations (in a vertex program):
- perform a sequence of math ops on each vertex
- transform world position into screen position for rasterizer
- generate texture coordinates for texturing
- perform vertex lighting to determine its color
**Primitive Assembly and Rasterizer**

Operations:
- assemble vertices into geometric primitives (triangles, lines, points)
- clipping to the view frustrum and other clip planes
- eliminate backward-facing polygons (culling)
- rasterize geometric primitives into fragments

**Fragment Stage**

Operations (fragment program):
- interpolation, texturing, and coloring
- math operations

Output:
- final color (RGBA), depth
- output is either 1 or 0 fragments (may be discarded)

**Raster Operations**

Final sequence of per-fragment operations before updating the framebuffer
- fragment may be discarded here as well

Rasterization:
- yields a set pixel locations and fragments

What is a fragment?
- potential pixel (still subject to fragment kill)
- values: color, depth, location, texture coordinate sets

Number of vertices and fragments are unrelated!

Early fragment kill
- set up *early depth culling* or *early stencil culling* to reject fragment before it is subjected to computations → major source of speedup
Vertices and fragments are vectors (up to dimension 4)
Vertex and fragment stage are programmable

Cg available to overcome need for assembler programming

Developed by Nvidia
Can work with OpenGL as well as DirectX
CG compiler produces OpenGL or DirectX code
  • e.g., OpenGL’s ARB_fragment_program language
OpenGL or DirectX drivers perform final translation into hardware-executable code
  • core CG runtime library (CG prefix)
  • cgGL and cgD3D libraries

Semantics connect Cg program with graphics pipeline
  • here: POSITION and COLOR
  float4, float2, float4x4, etc, are packed arrays
  • operations are most efficient
Uniform vs. Varying Parameters

- **Varying**: values vary per vertex or fragment
  - interfaced via semantics
- **Uniform**: remain constant
  - interfaced via handles

Compilation

To interface Cg programs with an application:
- compile the program with appropriate profile
  - dependent on underlying hardware
- range of profiles will grow with GPU advances
  - OpenGL: arbvp1 (basic), vp20, vp30, vp40 (advanced Nvidia)

Link the program to the application program

Can perform compilation at
- compile time (static)
- runtime (dynamic)

Cg Runtime

Can take advantage of
- latest profiles
- optimization of existing profiles

No dependency issues
- register names, register allocations

In the following, use OpenGL to illustrate
- DirectX similar methods

Preparing a Cg Program

First, create a context:
- context = cgCreateContext()

Compile a program by adding it to the context:
- program = cgCreateProgram(context, programString, profile, name, args)

Loading a program (pass to the 3D API):
- cgGLLoadProgram(program)
Running a Cg Program

Executing the profile:
• cgEnableProfile(CG_PROFILE_ARBVP1)

Bind the program:
• cgGLBindProgram(program)

After binding, the program will execute in subsequent drawing calls
• for every vertex (for vertex programs)
• for every fragment (for fragment programs)
• these programs are often called shaders

Only one vertex / fragment program can be bound at a time
• the same program will execute unless another program is bound

Disable a profile by:
• cgGLDisableProfile(CG_PROFILE_ARBVP1)

Release resources:
• cgDestroyProgram(program)
• cgDestroyContext(context)
• the latter destroys all programs as well

Error Handling

There are core CG routines that retrieve global error variables:
• error = cgGetError()
• cgGetErrorString(error)
• cgSet ErrorCallback(MyErrorCallback)

Passing Parameters into CG Programs

Assume these shader variables:
• float4 position : POSITION
• float4 color : COLOR0

Get the handle for color by:
• color = cgGetNamedParameter(program, "IN.color")

Can set the value for color by:
• cgGLSetParameter4f(color, 0.5f, 1.0f, 0.5f, 1.0f)

Uniform variables are set infrequently:
• example: modelViewMatrix
Passing Parameters into CG Programs

Set other variables via OpenGL semantics:
- glVertex, glColor, glTexCoord, glNormal,…

Example: rendering a triangle with OpenGL:
```gl
glBegin(GL_TRIANGLES);
    glVertex(0.8, 0.8);
    glVertex(-0.8, 0.8);
    glVertex(0.0, -0.8);

    glVertex3f(position, 0.0, 0.0);
    glVertex3f(position, 1.0, 0.0);
    glVertex3f(position, 0.0, 1.0);

    glColor4f(dark, dark, dark, 1.0);
    glColor4f(light, light, light, 1.0);
    glColor4f(white, white, white, 1.0);

    glEnable(GL_TEXTURE_2D);
    glBindTexture(GL_TEXTURE_2D, texture_id);
    glTexCoord2f(0.0, 0.0);
    glVertex2f(0.0, 0.0);
    glTexCoord2f(1.0, 1.0);
    glVertex2f(1.0, 1.0);
    glTexCoord2f(1.0, 0.0);
    glVertex2f(1.0, 0.0);
    glTexCoord2f(0.0, 1.0);
    glVertex2f(0.0, 1.0);
    glEnd();
```

Example 1

Vertex program
```gl
struct CGFV_Output {
    float4 color : COLOR;  //\n    float4 texCoord : TEXCOORD0;  //\n};
```
```gl
//\n//\n//\n//\n//\n```

```gl
OUT.position = float4(position, 0, 1);
OUT.color = color;
OUT.texCoord = texCoord;
return OUT;
```

- OUT parameter values are passed to fragment shader

Example 1 Example 1

Result, assuming:
- a fragment shader that just passes values through
- OpenGL program
```gl
glBegin(GL_TRIANGLES);
    glVertex(0.8, 0.8);
    glVertex(-0.8, 0.8);
    glVertex(0.0, -0.8);

    glVertex3f(position, 0.0, 0.0);
    glVertex3f(position, 1.0, 0.0);
    glVertex3f(position, 0.0, 1.0);

    glColor4f(dark, dark, dark, 1.0);
    glColor4f(light, light, light, 1.0);
    glColor4f(white, white, white, 1.0);

    glEnable(GL_TEXTURE_2D);
    glBindTexture(GL_TEXTURE_2D, texture_id);
    glTexCoord2f(0.0, 0.0);
    glVertex2f(0.0, 0.0);
    glTexCoord2f(1.0, 1.0);
    glVertex2f(1.0, 1.0);
    glTexCoord2f(1.0, 0.0);
    glVertex2f(1.0, 0.0);
    glTexCoord2f(0.0, 1.0);
    glVertex2f(0.0, 1.0);
    glEnd();
```

Example 2

Fragment program, following example 1 vertex program
```gl
struct CGFSL_output {
    float4 color : COLOR;
};
```
```gl
//\n//\n//\n//\n//\n```
```gl
CGFSL_output CGFSL_texture(float2 texCoord : TEXCOORD0, uniform sampler2D decal) {
    CGFSL_output OUT;
    OUT.color = tex2D(decal, texCoord);  //\n    return OUT;
```

Sampler2D is a texture object
- other types exist: sampler3D, samplerCUBE, etc.
Tex2D(decal, texCoord) performs a texture-lookup

- sampling, filtering, and interpolation depends on texture type and texture parameters
- advanced fragment profiles allow sampling using texture coordinate sets from other texture units (dependent textures)

Result

Math Support

A rich set of math operators and library functions
- +, -, *, sin, cos, floor, etc...
- no bit-wise operators yet, but operators reserved

Latest hardware full floating point on framebuffer operations
- half-floats are also available

Function overloading frequent
- for example, abs() function accepts float4, float2

Syntax

IN keyword
- call by value
- parameter passing by value

OUT keyword
- indicates when the program returns
Example 3

2D Twisting (vertex program)

```c
struct CS4D_output {
  float4 position : POSITION;
  float4 color : COLOR;
};

CS4D_output CS4Dv_twist(float2 position : POSITION,
                        float4 color : COLOR,
                        uniform float twisting)
{
  float4 OUT;
  float angle = twisting * length(position);
  float coelenLength, sinLenLength;
  sin(angle), sinLength = cos(length, coelenLength);
  OUT.position[0] = coelenLength * position[0] +
    -sinLength * position[1];
  OUT.position[1] = sinLength * position[0] +
    coelenLength * position[1];
  OUT.position[2] = 0;
  OUT.color = color;
  return OUT;
}
```

Result

finer meshes give better results

Example 4

Double Vision: vertex program

```c
void CS4Dv_twoTextures(float2 position : POSITION,
                        float2 texCoord : TEXCOORD0,
                        out float4 sPosition : POSITION,
                        out float2 leftTexCoord : TEXCOORD0,
                        out float2 rightTexCoord : TEXCOORD1,
                        uniform float2 leftSeparation,
                        uniform float2 rightSeparation)
{
  sPosition = float4(position, 0, 1);
  leftTexCoord = texCoord + leftSeparation;
  rightTexCoord = texCoord + rightSeparation;
  return OUT;
}
```

OUT is defined via semantics in the prototype

• optional

Example 4

Double Vision: fragment program #1

• advanced fragment profiles
• samples the same texture (named decal) twice

```c
lerp(a, b, weight)
• result = (1-weight) * a + weight * b
```

```c
void CS4Dv_twoTextures(float2 leftTexCoord : TEXCOORD0,
                        float2 rightTexCoord : TEXCOORD1,
                        out float4 color : COLOR,
                        uniform sampler2D decal)
{
  float4 leftColor = tex2D(decal, leftTexCoord);
  float4 rightColor = tex2D(decal, rightTexCoord);
  color = lerp(leftColor, rightColor, 0.5);  
}
```
Double Vision: fragment program #2

- basic fragment profiles
- samples two different textures (decal0 and decal1)
- textures must be bound to two texture units in advance

```c
void C3Df_twoTextures(float2 leftTexCoord : TEXCOORD0,
float2 rightTexCoord : TEXCOORD0,
out float4 color : COLOR,
uniform samplerRECT inputTexture : TEXUNIT0)
{
float4 leftColor = tex2D(decal0, leftTexCoord);
float4 rightColor = tex2D(decal1, rightTexCoord);
color = lerp(leftColor, rightColor, 0.5);
}
```

Rendering to Texture

Use FBO (Frame Buffer Object). Steps:

1. Create FBO Object
   ```c
   glGenFramebuffersEXT(1, &fbo);
   ```

2. Create the destination texture
   ```c
   glGenTextures(1, &color);
   glBindTexture(GL_TEXTURE_2D, color);
   glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA8, width, height, 0, GL_RGBA, GL_UNSIGNED_BYTE, NULL);
   ```

3. Bind FBO and attach the texture
   ```c
   glBindFramebufferEXT(GL_FRAMEBUFFER_EXT, fbo);
   glBindTexture(GL_TEXTURE_2D, color);
   ```

4. Make the FBO the current destination for rendering
   ```c
   glBindFramebufferEXT(GL_FRAMEBUFFER_EXT, fbo);
   ```

5. When done rendering, make the FrameBuffer into rendering dest.
   ```c
   glBindFramebufferEXT(GL_FRAMEBUFFER_EXT, 0);
   ```
Multiple Render Targets

Steps using FBO:
1. Create FBO Object
2. Create the destination textures
   ```
   glGenTextures(4, colors);
   glBindTexture(GL_TEXTURE_2D, color[0]);
   ……………
   ```
3. Attach destination textures
   ```
   glFramebufferTexture2DEXT(GL_FRAMEBUFFER_EXT,
   GL_COLOR_ATTACHMENT0_EXT, GL_TEXTURE_2D, color[0], 0);
   glFramebufferTexture2DEXT(GL_FRAMEBUFFER_EXT,
   GL_COLOR_ATTACHMENT1_EXT, GL_TEXTURE_2D, color[1], 0);
   ```
4. Setup DrawBuffers
   ```
   GLuint drawBuffers[4] = {GL_COLOR_ATTACHMENT0_EXT,
   GL_COLOR_ATTACHMENT1_EXT, GL_COLOR_ATTACHMENT2_EXT,
   GL_COLOR_ATTACHMENT3_EXT};
   glDrawBuffers(4, drawBuffers);
   ```
5. Make the FBO the current destination for rendering
6. When done rendering, make the FrameBuffer into rendering dest.

What NVidia said about the 6800 series:
SIMD branching
- incoherent branching can hurt performance
- should have coherent regions of ~1000 pixels
  - that is, only about 30x30 pixels, so still very usable!
Don’t ignore overhead of branch instructions
- branching over < 5 instructions may not be worth it
Use branching for early exit from loops
  - saves a lot of computation
Our Experience:
- incoherent branching in Volume Rendering causes severe slow downs 6X – 8X
  - exercise caution in designing/analyzing Scientific-Viz applications

Optimizations: Branching

Debugging tips

"Printf" debugging:
- display coordinates
- normals
- variable values
- map to [0…1] and show colors
Tools, suitable for our development:
- Graphic Remedy gDebugger – useful for state debugging
  - free for students/academic (ARB license program)
- Microsoft Shader Debugger Tool – software emulation
  - slow
- Imdebug – The Image Debugger [B. Baxter]
  - dumps textures to images relatively easily, and lets user explore values.
- Shadesmith – [T. Purcell, P. See]
  - discontinued, not current.
Performance "debugging"
- vary data load / compute load and observe performance change

GPGPU Style GPU Programming

Part 2
The GPU Architecture

Viewed as a highly parallel co-processor to the CPU (the host)

Host (CPU) tasks:
- control program flow
- perform thread management
- load SIMD kernels

Co-processor (GPU) tasks:
- load data
- perform computations

For each multiprocessor (for G80):
- number of 32-bit registers, \( N_{\text{regs}} = 8192 \)
- shared memory: 16 KB (16 banks)

Thread occupancy:
- determined by the amount of shared memory and registers used by each thread block (compiler helps to minimize these)
- \( N_{\text{regs}} \leq N_{\text{regs\_per\_thread}} \cdot N_{\text{threads}}, \) else kernel will not load
- similar holds for shared memory
- want to maximize occupancy (\# active warps / maximum \# warps)
- maximizing occupancy covers latencies in global memory fetches

CUDA Occupancy Calculator (see references in Section 3):
- a spreadsheet to help users in choosing best thread block size for a given kernel in order to achieve highest occupancy of the GPU
Programming Facilities

CUDA is a C-like language
- uses C-syntax
- can be used within C/C++ (and GL, DirectX)

Compilation using nvcc
Profiler available

gdb debugger for the GPU available in March, 2008

Course Schedule

1:30 – 2:00: Introduction
2:00 – 2:30: GPU architecture, programming model, and programming facilities
2:30 – 3:00: GPU programming examples (image processing)

Coffee Break

3:30 – 4:00: CT reconstruction pipeline components
4:00 – 4:30: GPU-acceleration of individual components
4:30 – 5:00: Various CT reconstruction pipelines, load balancing and load estimation
5:00 – 5:30: Reconstruction visualization and final remarks