MIC-GPU: High-Performance Computing for Medical Imaging on Programmable Graphics Hardware (GPUs)

SPIE Medical Imaging

GPU Architecture, Programming Model and Programming Facilities

Klaus Mueller
Computer Science
Center for Visual Computing
Stony Brook University

Overview

SPIE Medical Imaging

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

Overview

SPIE Medical Imaging

We will discuss two different programming models:
- graphics-style (first)
- GPGPU-style (second)

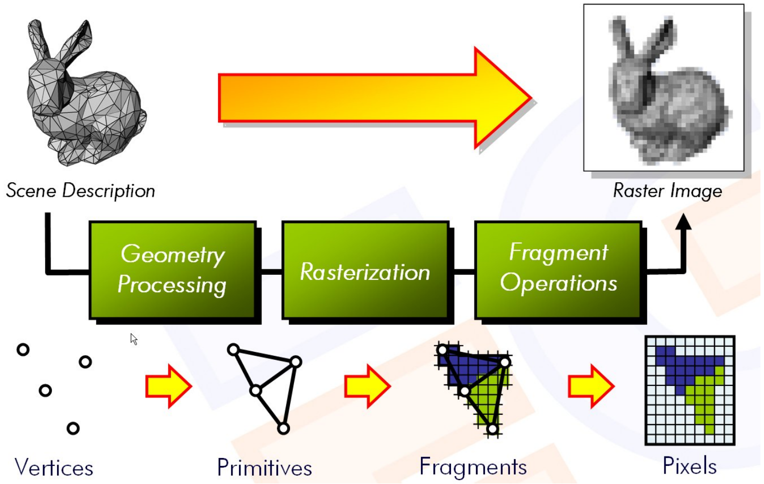
Note: both use the same underlying architecture (for example GeForce 8800)

Part 1

SPIE Medical Imaging

Graphics Style GPU Programming

Graphics Pipeline: Overview



Scene Description

Raster Image

Geometry Processing

Rasterization

Fragment Operations

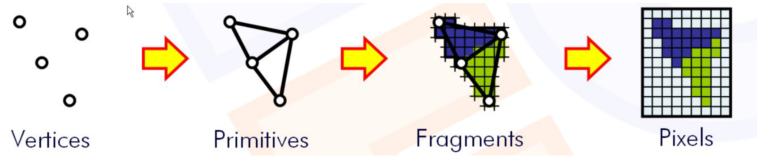
Vertices

Primitives

Fragments

Pixels

Graphics Pipeline: Overview



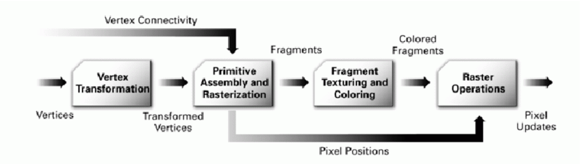
Vertices

Primitives

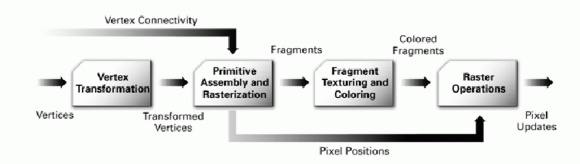
Fragments

Pixels

Graphics Hardware Pipeline



Vertex Stage



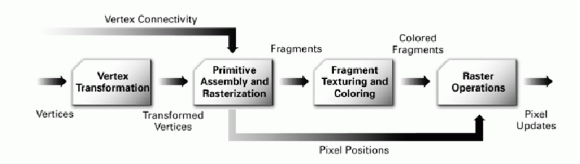
Vertex values:

- 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

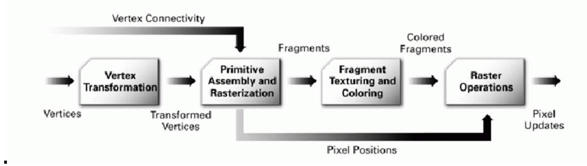
Primitive Assembly and Rasterizer



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

Primitive Assembly and Rasterizer



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

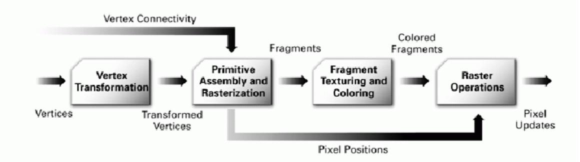
Number of vertices and fragments are unrelated!

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

---

Fragment Stage



Operations (fragment program):

- interpolation, texturing, and coloring
- math operations

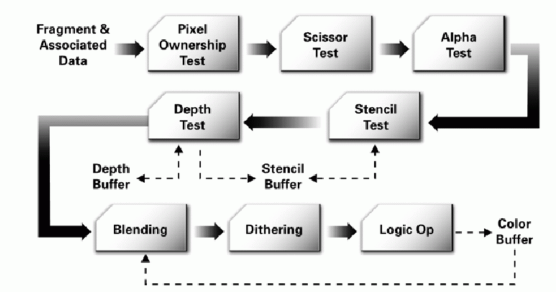
Output:

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

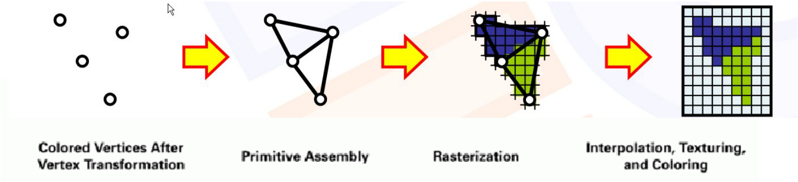
Raster Operations

Final sequence of per-fragment operations before updating the framebuffer

- fragment may be discarded here as well



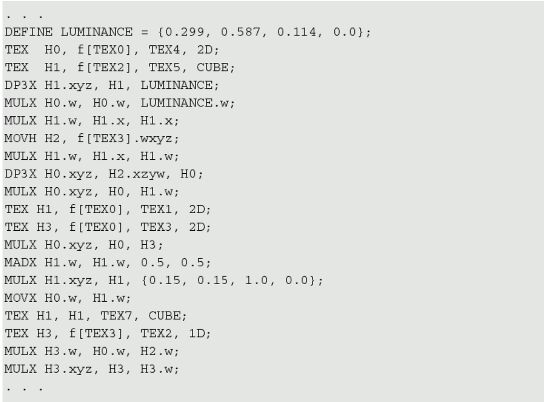
Graphics Pipeline: Summary



Vertices and fragments are vectors (up to dimension 4)

Vertex and fragment stage are programmable

Cg



```
. . .
DEFINE LUMINANCE = {0.299, 0.587, 0.114, 0.0};
TEX  H0, f[TEX0], TEX4, 2D;
TEX  H1, f[TEX2], TEX5, CUBE;
DP3X H1.xyz, H1, LUMINANCE;
MULX H0.w, H0.w, LUMINANCE.w;
MULX H1.w, H1.x, H1.x;
MOVH H2, f[TEX3].wxyz;
MULX H1.w, H1.x, H1.w;
DP3X H0.xyz, H2.xzyw, H0;
MULX H0.xyz, H0, H1.w;
TEX H1, f[TEX0], TEX1, 2D;
TEX H3, f[TEX0], TEX3, 2D;
MULX H0.xyz, H0, H3;
MADX H1.w, H1.w, 0.5, 0.5;
MULX H1.xyz, H1, {0.15, 0.15, 1.0, 0.0};
MOVX H0.w, H1.w;
TEX H1, H1, TEX7, CUBE;
TEX H3, f[TEX3], TEX2, 1D;
MULX H3.w, H0.w, H2.w;
MULX H3.xyz, H3, H3.w;
. . .
```

Cg available to overcome need for assembler programming

Cg

Developed by Nvidia

Can work with OpenGL as well as DirectX

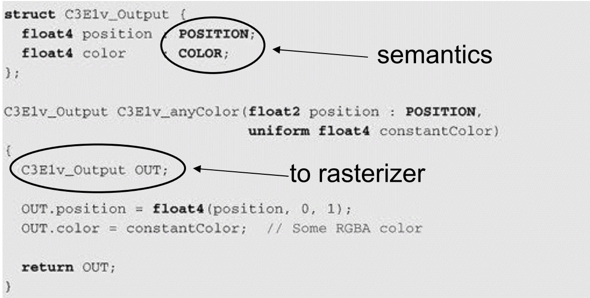
CG compiler produces OpenGL or DirectX code

- 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

Simple Vertex Program



```
struct C3E1v_Output {
  float4 position : POSITION;
  float4 color    : COLOR;
};

C3E1v_Output C3E1v_anyColor(float2 position : POSITION,
                            uniform float4 constantColor)
{
  C3E1v_Output OUT;

  OUT.position = float4(position, 0, 1);
  OUT.color = constantColor;  // Some RGBA color

  return OUT;
}
```

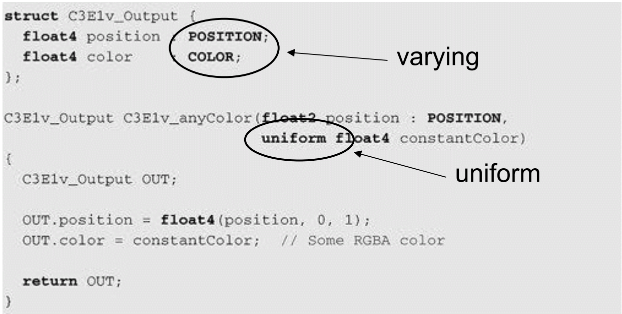
Semantics connect Cg program with graphics pipeline

- here: POSITION and COLOR

float4, float2, float4x4, etc, are packed arrays

- operations are most efficient

Uniform vs. Varying Parameters



```
struct C3E1v_Output {
  float4 position : POSITION;
  float4 color    : COLOR;
};

C3E1v_Output C3E1v_anyColor(float2 position : POSITION,
                            uniform float4 constantColor)
{
  C3E1v_Output OUT;

  OUT.position = float4(position, 0, 1);
  OUT.color = constantColor;  // Some RGBA color

  return OUT;
}
```

Varying: values vary per vertex or fragment

- interfaced via semantics

Uniform: remain constant

- interfaced via handles

Compilation

To interface Cg programs with 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

More detailed information available when you download CG from the Nvidia developer website:

- http://developer.nvidia.com
- Cg toolkit 2.0 now available (including Cg compiler)

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*

Running a Cg Program

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:

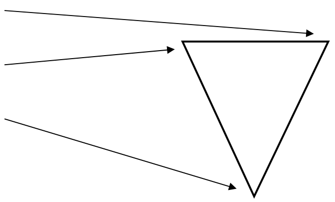
- example: modelViewMatrix

Passing Parameters into CG Programs

Set other variables via OpenGL semantics:

- glVertex, glColor, glTexCoord, glNormal,…

Example: rendering a triangle with OpenGL:



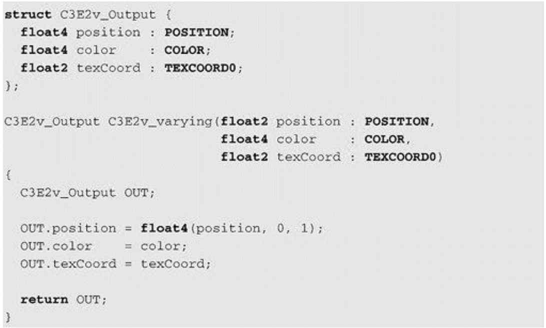
```
glBegin(GL_TRIANGLES);
  glVertex( 0.8,  0.8);
  glVertex(-0.8,  0.8);
  glVertex( 0.0, -0.8);
glEnd();
```

glVertex affects POSITION semantics and updates/sets related parameter in vertex shader

---

Example 1

Vertex program



```
struct C3E2v_Output {
  float4 position : POSITION;
  float4 color    : COLOR;
  float2 texCoord : TEXCOORD0;
};

C3E2v_Output C3E2v_varying(float2 position : POSITION,
                           float4 color    : COLOR,
                           float2 texCoord : TEXCOORD0)
{
  C3E2v_Output OUT;

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

  return OUT;
}
```

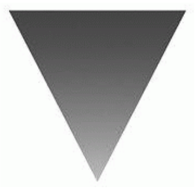
- OUT parameter values are passed to fragment shader

Example 1

Result, assuming:

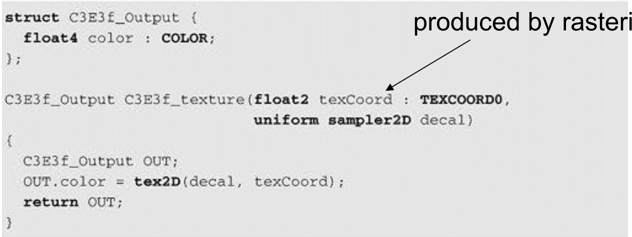
- a fragment shader that just passes values through
- OpenGl program

```
glBegin(GL_TRIANGLES);
  glVertex( 0.8,  0.8); glColor(dark);
  glVertex(-0.8,  0.8); glColor(dark);
  glVertex( 0.0, -0.8); glColor(light);
glEnd();
```



Example 2

Fragment program, following example 1 vertex program



```
struct C3E3f_Output {
  float4 color : COLOR;
};

C3E3f_Output C3E3f_texture(float2 texCoord : TEXCOORD0,
                           uniform sampler2D decal)
{
  C3E3f_Output OUT;
  OUT.color = tex2D(decal, texCoord);
  return OUT;
}
```

produced by rasterizer

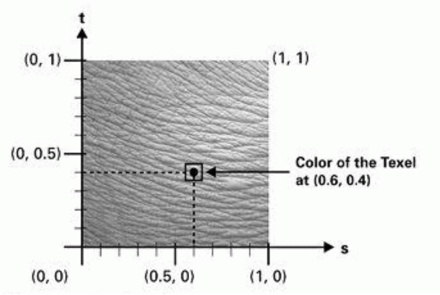
Sampler2D is a texture object

- other types exist: sampler3D, samplerCUBE, etc.

Example 2

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*)



Example 2

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

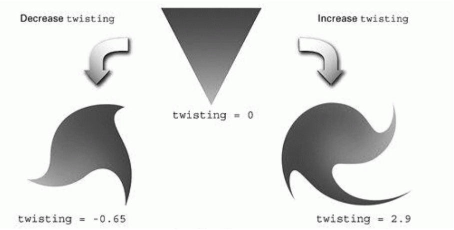
Example 3

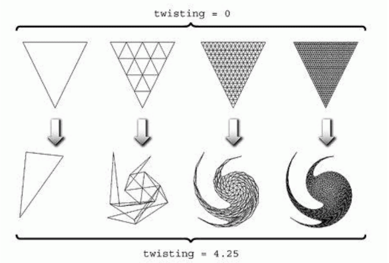
2D Twisting (vertex program)

```
struct C3E4_Output {
  float4 position : POSITION;
  float4 color    : COLOR;
};

C3E4_Output C3E4v_twist(float2 position : POSITION,
                        float4 color    : COLOR,

                        uniform float twisting)
{
  C3E4_Output OUT;
  float angle = twisting * length(position);
  float cosLength, sinLength;
  sincos(angle, sinLength, cosLength);
  OUT.position[0] = cosLength * position[0] +
                   -sinLength * position[1];
  OUT.position[1] = sinLength * position[0] +
                    cosLength * position[1];
  OUT.position[2] = 0;
  OUT.position[3] = 1;
  OUT.color = color;
  return OUT;
}
```

Example 3

Result





finer meshes give better results

Example 4

Double Vision: vertex program

```
void C3E5v_twoTextures(float2 position : POSITION,
                       float2 texCoord : TEXCOORD0,

                   out float4 oPosition : POSITION,
                   out float2 leftTexCoord : TEXCOORD0,
                   out float2 rightTexCoord : TEXCOORD1,

               uniform float2 leftSeparation,
               uniform float2 rightSeparation)
{
  C3E5v_Output OUT;

  oPosition     = 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

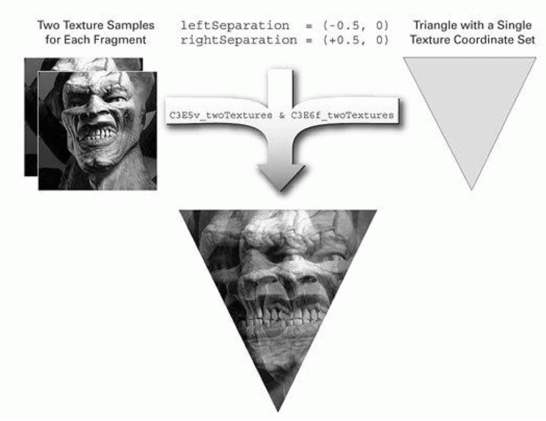
lerp(a, b, weight)

- result = (1-weight) · a + weight · b

```
void C3E6f_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);
}
```

Example 4

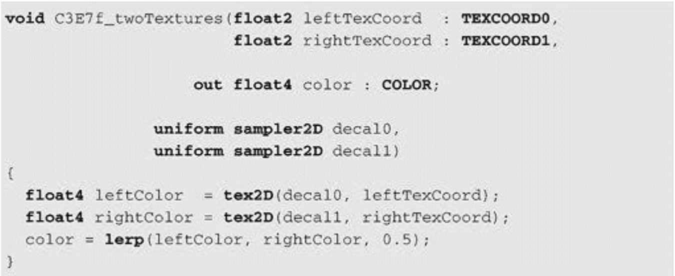
Result



Example 4

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



```
void C3E7f_twoTextures(float2 leftTexCoord  : TEXCOORD0,
                       float2 rightTexCoord : TEXCOORD1,

                   out float4 color : COLOR;

               uniform sampler2D decal0,
               uniform sampler2D decal1)
{
  float4 leftColor  = tex2D(decal0, leftTexCoord);
  float4 rightColor = tex2D(decal1, rightTexCoord);
  color = lerp(leftColor, rightColor, 0.5);
}
```

HelloWorld.cg

Prerequisites:

- NVIDIA GeForce FX or an ATI RADEON 9500
- Visual Studio .NET 2003 or your favourite compiler
  (we provide project files for VS.Net2003)
- GLUT (simple GUI library) and GLEW (extension library)
- NVidia Cg toolkit (for Cg)

```
fragout main( float4 TexCoords                         : TEXCOORD0,
              float4 WinPos                            : WPOS,
              uniform samplerRECT  inputTexture        : TEXUNIT0)
{
          fragout OUT;
          int xx=(int)(WinPos.x/8);
          int yy=(int)(WinPos.y/8);
          if(xx %2 ==yy %2){    // if in even (x,y) box write texture
                   OUT.col=texRECT(inputTexture, TexCoords.xy);
          }else{    // else write grey color
                   OUT.col=float4(0.5,0.5,0.5,1);
          }
          return OUT;
}
```

Renders the given texture covered by a chess-board pattern.

Rendering to Texture

Use FBO (Frame Buffer Object). Steps:

1. Create FBO Object
   `glGenFramebuffersEXT(1, &fbo);`
2. Create the destination texture
   ```
   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
   ```
   glBindFramebufferEXT(GL_FRAMEBUFFER_EXT, fbo);
   glFramebufferTexture2DEXT(GL_FRAMEBUFFER_EXT,
   GL_COLOR_ATTACHMENT0_EXT, GL_TEXTURE_2D, color, 0);
   ```
4. Make the FBO the current destination for rendering
   `glBindFramebufferEXT(GL_FRAMEBUFFER_EXT, fbo);`
5. When done rendering, make the FrameBuffer into rendering dest.
   `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.

Optimizations: Branching

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 useable!

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 slowdowns 6X – 8X
- exercise caution in designing/analyzing Scientific-Viz applications

---

Debugging tips

"Printf" debugging:

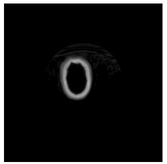
- display coordinates
- normals
- variable values
- map to [0…1] and show colors

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. Sen]
  - discontinued, not current.

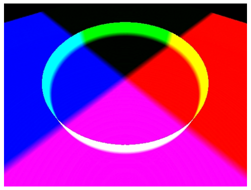
Performance "debugging"

- vary data load / compute load and observe performance change

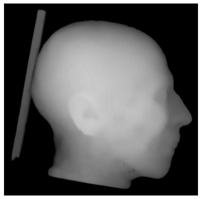


Current Rendering Slice

Depth Buffer to debug early-z



Debugging texture coordinates



Debugging Screen-Space Shading

Part 2

GPGPU Style GPU Programming

The GPU Architecture

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

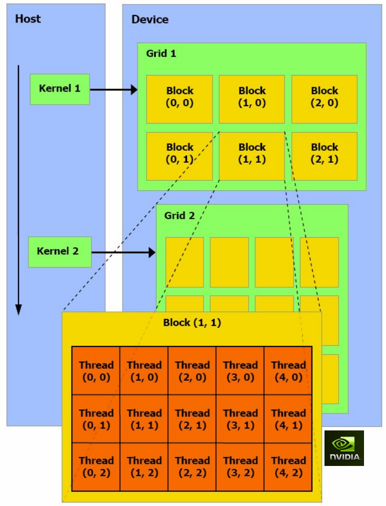
Host (CPU) tasks:

- control program flow
- perform thread management
- load SIMD kernels

Co-processor (GPU) tasks:

- load data
- perform computations

The GPU Architecture



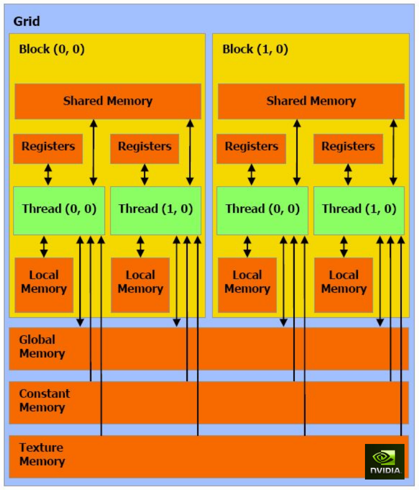
each kernel is executed as a batch of threads (G80 up to 768 threads)

this batch is organized as a grid of thread blocks

the threads in each block cooperate, share memory, and can be synchronized
→ there is a limit on the maximum number of threads within a block

each thread and each block have an ID

The GPU Architecture



each multi-processor executes a batch of blocks in sequence
→ the number of these *active* blocks is determined by the amount of registers and shared memory needed per block

each active block is split into a SIMD group of threads, called *warps*
→ each warp contains the same number of threads (the *warp-size*, 32 in G80)

the warps of the active blocks (called *active warps*) are time-sliced (switched in execution)
→ this maximizes use of computational resources (e.g., when waiting for data from memory)

note: the order of execution of blocks and warps is undefined

---

Optimizing Performance

For each multiprocessor (for G80):

- number of 32-bit registers, N_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_regs ≤ N_regs_per_thread · N_threads, else kernel will not load
- similar holds for shared memory
- want to maximize occupacy (# active warps / maximum # warps)
- maximizing occupancy covers latencies in global memory fetches

CUDA Occupancy Calculator (see references in Section 3):

- a spreadheet 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 facilties
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