General Purpose Graphics Processing Unit

GPGPU

Olav Haugehåtveit
haugehat@stud.ntnu.no

Department of Electronics and Telecommunication
Norwegian University of Science and Technology

June 2006

Outline

Outline

1. **GPU**
   - Introduction
   - Performance analysis
2. **Programming model**
   - The graphics programming model
   - GPGPU programming model
3. **Conclution**
   - Conclution

What is a GPU?

GPU = Graphics Processing Unit

Purpose

- Draw graphics on the monitor

What scientists what with it?

- Non graphics application (ie. numerical simulations)

Why?
Enormous floating point power

What is a GPU?

GPU = Graphics Processing Unit

Purpose

- Draw graphics on the monitor

What scientists what with it?

- Non graphics application (ie. numerical simulations)

Why?
Enormous floating point power

What is a GPU?

GPU = Graphics Processing Unit

Purpose

- Draw graphics on the monitor

What scientists what with it?

- Non graphics application (ie. numerical simulations)

Why?

Enormous floating point power

What is a GPU?

GPU = Graphics Processing Unit

Purpose

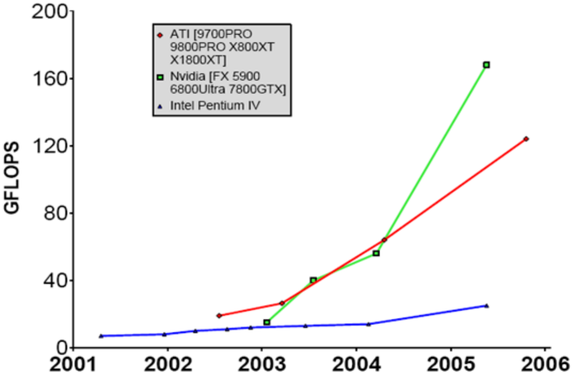
- Draw graphics on the monitor

What scientists what with it?

- Non graphics application (ie. numerical simulations)

Why?
Enormous floating point power

Floating point increment



(Data courtesy of Ian Buck, Mike Houston)

Performance analysis

CPU

- Annual growth $\approx$ 1.5x $\longrightarrow$ Decade growth $\approx$ 60x
- Follows Moore's law

GPU

- Annual growth $\approx$ 2.0x $\longrightarrow$ Decade growth $\approx$ 1000x
- Much faster than Moore's law

Performance analysis

CPU

- Annual growth $\approx$ 1.5x $\rightarrow$ Decade growth $\approx$ 60x
- Follows Moore's law

GPU

- Annual growth $\approx$ 2.0x $\rightarrow$ Decade growth $\approx$ 1000x
- Much faster than Moore's law

Performance analysis

CPU

- Annual growth ≈ 1.5x → Decade growth ≈ 60x
- Follows Moore's law

GPU

- Annual growth ≈ 2.0x → Decade growth ≈ 1000x
- Much faster than Moore's law

Performance analysis

CPU

- Annual growth $\approx$ 1.5x $\rightarrow$ Decade growth $\approx$ 60x
- Follows Moore's law

GPU

- Annual growth $\approx$ 2.0x $\rightarrow$ Decade growth $\approx$ 1000x
- Much faster than Moore's law

Performance analysis

CPU

- Annual growth $\approx$ 1.5x $\rightarrow$ Decade growth $\approx$ 60x
- Follows Moore's law

GPU

- Annual growth $\approx$ 2.0x $\rightarrow$ Decade growth $\approx$ 1000x
- Much faster than Moore's law

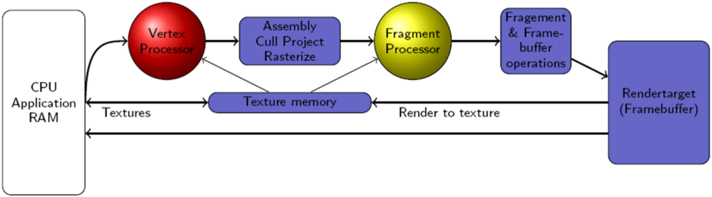
Performance analysis

Why are they so fast?

- Parallel architecture optimized for floating point arthimetic
  - 2-48 pipelines
  - $\approx$ 20 flops/pipeline pr. clock!
  - 650 MHz
- Data is read and write only
- High memory bandwidth

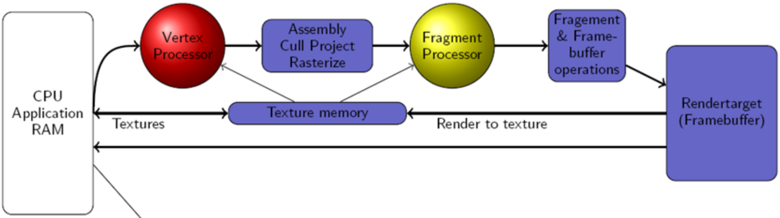
Outline

The graphics pipeline



The GPU acts as a stream computer
Given a stream of data, it executes the same operation on every data element

The graphics programming model

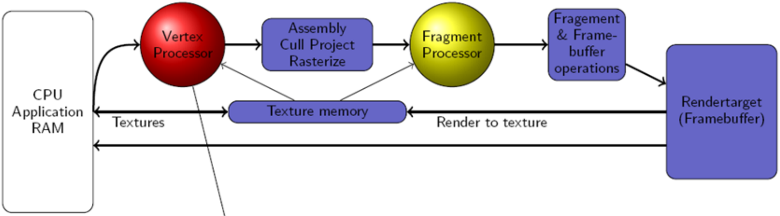


The CPU:

- Uploads shaders
- Uploads textures
- Sends geometry
- Executes the pipeline

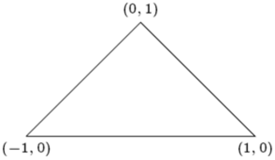
```
glBindTexture( tex, GL_TEXTURE_2D );
glUseProgram( progID );
glBegin( GL_TRIANGLES);
glNormal3f( 0.0, -0.35, 0.67 );
glVertex3f( -1.0, 0.0, 0.0 );
glNormal3f( 0.001, -0.49, -0.62 );
glVertex3f( 1.0, 0.0, 0.0 );
.
.
.
glEnd();
glFlush();
```

The graphics programming model

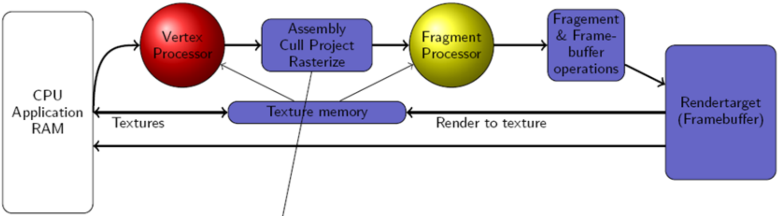


The vertex processor transforms

- Position
- Normal
- Color
- Texture coordinates
- User defined attributes

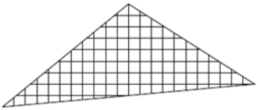


The graphics programming model

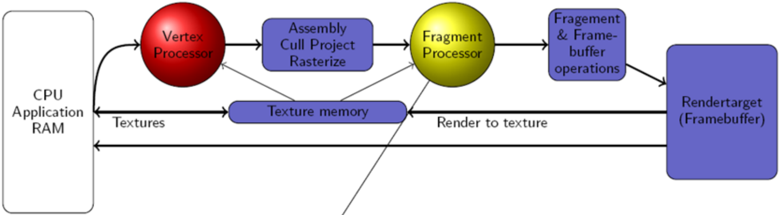


- Vertices are assembled into primitives
- Primitives are clipped
- Vertices are projected into window coordinates
- Primitives are rasterized into fragments
- Attributes are interpolated across primitives

(a fragment is a *meta-pixel* is has depth as well as (x, y)-coordinate)



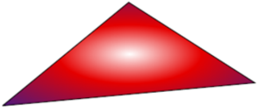
The graphics programming model



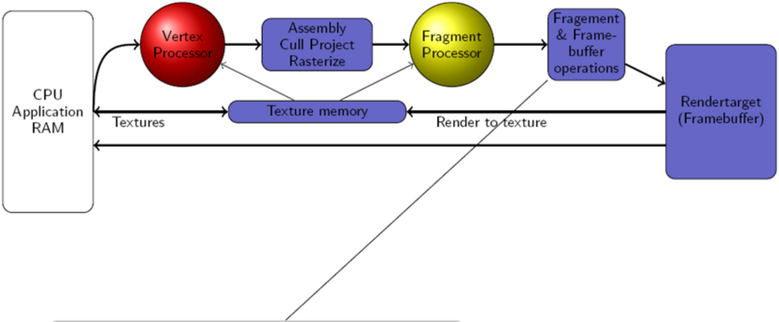
The fragment processor:

- calculates the final color and depth

This usually involves texture lookup and viewport calculations based on attributes from the vertex proce

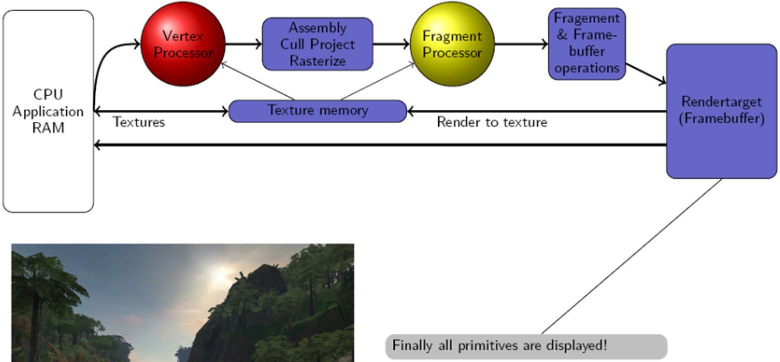


The graphics programming model



- Fragments are discarded or blended
- Writing to auxiliary buffers

Looping


CPU
Application
RAM
Vertex
Processor
Assembly
Cull Project
Rasterize
Fragment
Processor
Fragement
& Frame-
buffer
operations
Rendertarget
(Framebuffer)
Texture memory
Textures
Render to texture
Finally all primitives are displayed!

Mapping computational concepts to the GPU

CPU

- Array
- Inner loop
- Feedback
- Computational invocation
- Computational domain

GPU

- Texture
- Fragment shader
- Render to texture
- Geometry raserization
- Texture coordinates

Mapping computational concepts to the GPU

CPU

- Array
- Inner loop
- Feedback
- Computational invocation
- Computational domain

GPU

- Texture
- Fragment shader
- Render to texture
- Geometry raserization
- Texture coordinates

Mapping computational concepts to the GPU

CPU

- Array
- Inner loop
- Feedback
- Computational invocation
- Computational domain

GPU

- Texture
- Fragment shader
- Render to texture
- Geometry raserization
- Texture coordinates

Mapping computational concepts to the GPU

CPU

- Array
- Inner loop
- Feedback
- Computational invocation
- Computational domain

GPU

- Texture
- Fragment shader
- Render to texture
- Geometry raserization
- Texture coordinates

Mapping computational concepts to the GPU

CPU

- Array
- Inner loop
- Feedback
- Computational invocation
- Computational domain

GPU

- Texture
- Fragment shader
- Render to texture
- Geometry raserization
- Texture coordinates

Mapping computational concepts to the GPU

CPU

- Array
- Inner loop
- Feedback
- Computational invocation
- Computational domain

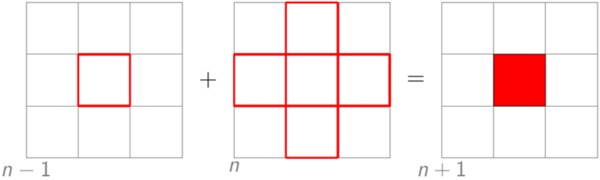
GPU

- Texture
- Fragment shader
- Render to texture
- Geometry raserization
- Texture coordinates

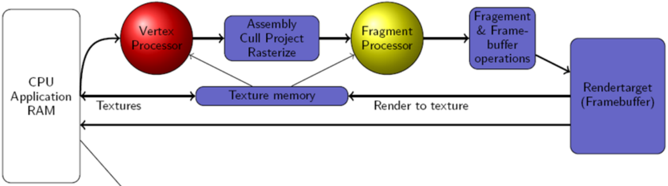
The heat equation

Example

The heat equation: $\frac{\partial^2 u}{\partial t^2} = \nabla^2 u$



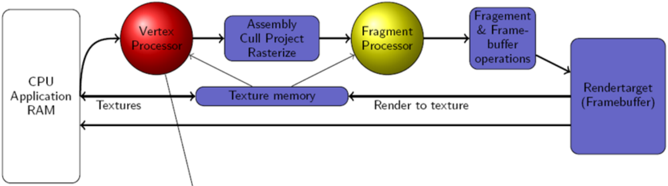
The heat equation



For every timestep
- Binds the previous rendertargets as textures
- Set up a
- Draws a single quad

```
glUseProgram( waveequation );
glUniform1i( 0, n_minus );
glUniform1i( 1, n );
glDrawBuffers( 1, &n_plus );
glBegin( GL_QUADS);
glVertex2f( 0.0, 0.0);
glVertex2f( 1.0, 0.0);
glVertex2f( 1.0, 1.0);
glVertex2f( 0.0, 1.0);
glEnd();
swap( n_plus, n_minus );
swap( n_minus, n );
```

The heat equation



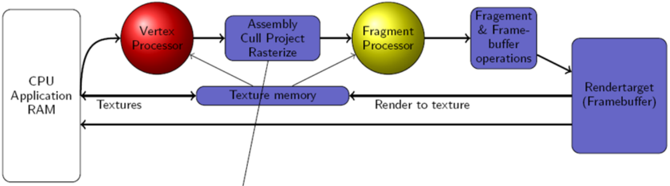
The vertex processor:

- Calculates texture coordinates
- Passes everything trough

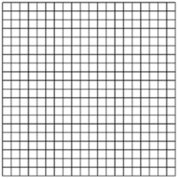
```
varying vec4 Xcoord;
varying vec4 Ycoord;

Xcoord=gl_MultiTexCoord0.yxxx +
vec4(0.0,0.0,-1.0,1.0)
Ycoord=gl_MultiTexCoord0.xyyy+
vec4(0.0,0.0,-1.0,1.0);
gl_Position = ftransform();
```

GPGPU programming model

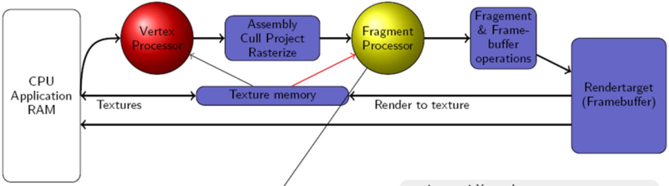
The heat equation



- The geometry is rasterized into pixels
- Texture coordinates are interpolated



The heat equation



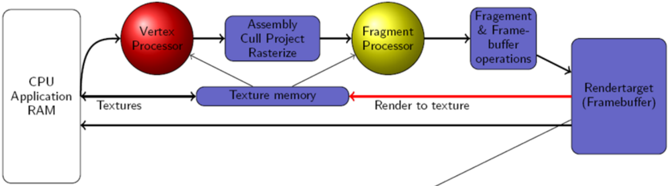
The fragment shader calculates our expression

$$u_{i,j}^{n+1} = 2u_{i,j}^{n} - u_{i,j}^{n-1} + \frac{k}{h^2}(u_{i+1,j}^{n} + u_{i-1,j}^{n} + u_{i,j+1}^{n} + u_{i,j-1}^{n} - 4u_{i,j}^{n})$$

Textures are used as arrays

```
varying vec4 Xcoord;
varying vec4 Ycoord;

uniform sampler2D n;
uniform sampler2D n_minus;

vec4 tex = texture2D(n, Xcoord.yx);
vec4 tex0 = texture2D(n, Xcoord.wx);
vec4 tex1 = texture2D(n, Xcoord.zx);
vec4 tex2 = texture2D(n, Ycoord.xw);
vec4 tex3 = texture2D(n, Ycoord.xz);
vec4 texL = texture2D(n_minus, Xcoord.yx);

gl_FragData[0] = (2.0 * tex - texL +
(2.0/4.0)*(tex0 + tex1 + tex2 + tex3 -
4.0*tex));
```

The heat equation



- Our computation is written to a texture
- It can immediately be reused in the next step
- No data is transfered to the CPU
- Nothing is displayed on screen

Outline

Well suited applications

- Large data sets
- High parallelism
- Minimal dependencies between data elements
- High arithmetic intensity
- Lots of work to do without CPU intervention

Application ported to the GPU

- Matrix Algebra
- Partial Differential Equations
- Image processing
- Fast Fourier Transform
- Ray Tracing
- Geometric computing
- Databases

Advantages and disadvantages

Advantages

- flops, Gflops, Tflops
- Sony PS3 graphics chip RSX has 1.8 Tflops!

Disadvantages

- Programming model is inherently parallel
- Programming model is tied to graphics
- Limited to 32-bit floating point
- Rapidly evolving architectures
- Largely secret architectures

End

Questions?