Graphics Architectures

Adrien Treuille
Carnegie Mellon University
Outline

• Review: GPU Architectures
• ISAs vs APIs
• Cuda Memory (ISA)
• OpenGL Memory (API)
• Memory Mapping
• Advanced Streaming
• Texture Memory Layout
• Review: GPU Architectures
  • ISAs vs APIs
  • Cuda Memory (ISA)
  • OpenGL Memory (API)
  • Memory Mapping
  • Advanced Streaming
  • Texture Memory Layout
CPU-“style” cores

- Fetch/Decode
- ALU (Execute)
- Execution Context
- Out-of-order control logic
- Fancy branch predictor
- Memory pre-fetcher
- Data cache (A big one)
Add ALUs

Idea #2:

Amortize cost/complexity of managing an instruction stream across many ALUs

**SIMD processing**

(SIMD = single-instruction, multiple-data)

Source: Kayvon Fatahalian
128 [ ] in parallel

Source: Kayvon Fatahalian
But what about branches?

```plaintext
if (x > 0) {
    y = pow(x, exp);
    y *= Ks;
    refl = y + Ka;
} else {
    x = 0;
    refl = Ka;
}
```

<resume unconditional shader code>
But what about branches?

Not all ALUs do useful work! Worst case: 1/8 performance

```c
<unconditional shader code>
if (x > 0) {
    y = pow(x, exp);
    y *= Ks;
    refl = y + Ka;
} else {
    x = 0;
    refl = Ka;
}
<resume unconditional shader code>
```
Wide SIMD processing

In practice:
16 to 64 fragments share an instruction stream

Source: Kayvon Fatahalian
Stalls!

Stalls occur when a core cannot run the next shader instruction because it is waiting on a previous operation.

Source: Kayvon Fatahalian
A diffuse reflectance shader

```cpp
sampler mySampler;
Texture2D<float3> myTexture;
float3 lightDir;

float4 diffuseShader(float3 norm, float2 uv)
{
    float3 kd;
    kd = myTexture.Sample(mySampler, uv);
    kd *= clamp(dot(lightDir, norm), 0.0, 1.0);
    return float4(kd, 1.0);
}
```

Texture access latency = 100’s to 1000’s of cycles

Source: Kayvon Fatahalian
Recall: CPU-“style” core

Data cache
(big one)

OOO exec logic
branch pred.
Fetch/Decode
ALU
Execution Context
CPU-“style” memory hierarchy

- **L1 cache**: 32 KB
- **L2 cache**: 256 KB
- **L3 cache**: 8 MB (shared across cores)

CPU cores run efficiently when data is resident in cache (caches reduce latency, provide high bandwidth)
Stalls!

Stalls occur when a core cannot run the next instruction because of a dependency on a previous operation.

Texture access latency = 100’s to 1000’s of cycles

Remember: on a GPU we’ve removed the fancy caches and logic that helps avoid stalls (to fit more ALUs).

Source: Kayvon Fatahalian
Hiding shader stalls

Source: Kayvon Fatahalian
Hiding shader stalls

Source: Kayvon Fatahalian

Tuesday, January 25, 2011
Hiding shader stalls

Time (clocks)

Frag 1 ... 8
Frag 9 ... 16
Frag 17 ... 24
Frag 25 ... 32

Runnable
Stall

Source: Kayvon Fatahalian
Hiding shader stalls

<table>
<thead>
<tr>
<th>Time (clocks)</th>
<th>Frag 1 ... 8</th>
<th>Frag 9 ... 16</th>
<th>Frag 17 ... 24</th>
<th>Frag 25 ... 32</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>☀️☀️☀️☀️☀️</td>
<td>☀️☀️☀️☀️</td>
<td>☀️☀️☀️☀️</td>
<td>☀️☀️☀️☀️</td>
</tr>
</tbody>
</table>

Stall

Runnable

---

Source: Kayvon Fatahalian

Tuesday, January 25, 2011
Hiding shader stalls

Time (clocks)

1. Frag 1 … 8
2. Frag 9 … 16
3. Frag 17 … 24
4. Frag 25 … 32

Runnable

Stall

Runnable

Stall

Runnable

Stall

Source: Kayvon Fatahalian
High-throughput computing!

Increase time to complete one group
To minimize time to complete all groups!

Source: Kayvon Fatahalian
Four large contexts

(source: Kayvon Fatahalian)

(low latency hiding ability)
Twenty small contexts

(maximal latency hiding ability)
Current and future: GPU architectures

• Bigger and faster (more cores, more FLOPS)
  – 2 TFLOPS today…

• What fixed-function hardware should remain?

• Addition of (a few) CPU-like features
  – Traditional caches
Outline

- Review: GPU Architectures
- ISAs vs APIs
- Cuda Memory (ISA)
- OpenGL Memory (API)
- Memory Mapping
- Advanced Streaming
- Texture Memory Layout
Outline

- Review: GPU Architectures
- ISAs vs APIs
- Cuda Memory (ISA)
- OpenGL Memory (API)
- Memory Mapping
- Advanced Streaming
- Texture Memory Layout
ISAs vs APIs

Instruction Set Architecture

```c
// calculate the unique thread index
int index = blockIdx.x * blockDim.x + threadIdx.x;

unsigned int x = index * width;
unsigned int y = index / width;

// TODO: make each thread output a meaningful color
// according to the thread pixel's membership
// of the Mandelbrot set
unsigned char r = int((x / float(width)) * 255.0f);
unsigned char g = int((y / float(height)) * 255.0f);
unsigned char b = 128;

// Each thread writes one pixel location in the texture (texel)
pos[index].x = r;
pos[index].y = g;
pos[index].z = b;
pos[index].w = 0;
```

Advantages
- Optimization
- Simple Memory Semantics

Disadvantages
- New Language
- Special Compiler

OpenGL Programming Interface

```c
glDisable(GL_TEXTURE);
glBegin(GL_QUADS);
    glColor4f(1.0, 0.0, 0.0, 1.0); glVertex3f(0.0f, 0.0f, 0.0f);
    glColor4f(0.0, 1.0, 0.0, 1.0); glVertex3f(1.0f, 0.0f, 0.0f);
    glColor4f(0.0, 0.0, 1.0, 1.0); glVertex3f(1.0f, 1.0f, 0.0f);
    glColor4f(1.0, 1.0, 0.0, 1.0); glVertex3f(0.0f, 1.0f, 0.0f);
End;
```

Advantages
- Leverage Existing Language
- Higher Abstraction

Disadvantages
- Complex Memory Semantics
- Control Flow?
Outline

• Review: GPU Architectures
• ISAs vs APIs
• Cuda Memory (ISA)
• OpenGL Memory (API)
• Memory Mapping
• Advanced Streaming
• Texture Memory Layout
Outline

• Review: GPU Architectures
• ISAs vs APIs
• Cuda Memory (ISA)
• OpenGL Memory (API)
• Memory Mapping
• Advanced Streaming
• Texture Memory Layout
CUDA Memory

ISA Memory Semantics are Straightforward

```
char * device_str = 0;
cudaMalloc((void **) &device_str, (str_len + 1) * sizeof(char));
cudaMemcpy(device_str, host_str, (str_len + 1) * sizeof(char), cudaMemcpyHostToDevice);
```

```
__global__
void capitalize(char * str, unsigned int str_len)
{
    const unsigned int ii = blockIdx.x * n_threads_per_block + threadIdx.x;
    if (ii >= str_len)
        return;
    char cc = str[ii];
    if (cc >= 'a' && cc <= 'z') // if the character is lower case
        cc -= difference; // capitalize it
    str[ii] = cc;
}
```
Outline

• Review: GPU Architectures
• ISAs vs APIs
• Cuda Memory (ISA)
• OpenGL Memory (API)
• Memory Mapping
• Advanced Streaming
• Texture Memory Layout

Tuesday, January 25, 2011
Outline

- Review: GPU Architectures
- ISAs vs APIs
- Cuda Memory (ISA)
- OpenGL Memory (API)
- Memory Mapping
- Advanced Streaming
- Texture Memory Layout
OpenGL Memory

API Memory Semantics can be Complex

Pixel Buffer Object

Vertex Buffer Object

Texture

Frame Buffer Object
What is a **Buffer Object**?

Buffer Objects are (basically) what OpenGL calls **Arrays**.
A Pixel Buffer Object is an array of pixel colors.

- Standard Memory Layout (column/row major)
- **Cannot** be directly Displayed
Creating PBOs
(and basically any buffer object)

```c
if (pbo) {
    // set up vertex data parameter
    int num_texels = image_width * image_height;
    int num_values = num_texels * 4;
    int size_tex_data = sizeof(GLubyte) * num_values;

    // Generate a buffer ID called a PBO (Pixel Buffer Object)
    glGenBuffers(1, pbo);
    // Make this the current UNPACK buffer (OpenGL is state-based)
    glBindBuffer(GL_PIXEL_UNPACK_BUFFER, *pbo);
    // Allocate data for the buffer. 4-channel 8-bit image
    glBufferData(GL_PIXEL_UNPACK_BUFFER, size_tex_data, NULL, GL_DYNAMIC_COPY);
    cudaGLRegisterBufferObject(*pbo);
}
```
A **Texture** is also an array of pixel colors.

- Optimized (Opaque) Memory Layout
- **Can** be directly Displayed
Creating a Texture

```c
// Generate a texture identifier
 glGenTextures(1, textureID);

// Make this the current texture (remember that GL is state-based)
 glBindTexture( GL_TEXTURE_2D, *textureID);

// Allocate the texture memory. The last parameter is NULL since we only
// want to allocate memory, not initialize it
 glTexImage2D( GL_TEXTURE_2D, 0, GL_RGBA8, image_width, image_height, 0,
               GL_BGRA, GL_UNSIGNED_BYTE, NULL);

// Must set the filter mode, GL_LINEAR enables interpolation when scaling
 glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
 glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);

// Note: GL_TEXTURE_RECTANGLE_ARB may be used instead of
// GL_TEXTURE_2D for improved performance if linear interpolation is
// not desired. Replace GL_LINEAR with GL_NEAREST in the
// glTexParameteri() call
```
Pack/Unpack

Diagram showing the relationship between PBO, Framebuffer, and Texture Object with the processes of pack and unpack.
CPU->OpenGL Texture

Direct Texture Access

PBO Texture Access

source: http://www.songho.ca/opengl/gl_pbo.html
// Create a texture from the buffer
glBindBuffer(GL_PIXEL_UNPACK_BUFFER, pbo);

// bind texture from PBO
glBindTexture(GL_TEXTURE_2D, textureID);

// Note: glTexImage2D will perform a format conversion if the
// buffer is a different format from the texture. We created the
// texture with format GL_RGBA8. In glTexImage2D we specified
// GL_BGRA and GL_UNSIGNED_INT. This is a fast-path combination

// Note: NULL indicates the data resides in device memory
glTexImage2D(GL_TEXTURE_2D, 0, 0, 0, image_width, image_height,
             GL_RGBA, GL_UNSIGNED_BYTE, NULL);
A VBO is an array of vertex coordinates (possibly interleaved with other data).

```c
// bind VBOs for vertex array and index array
glBindBufferARB(GL_ARRAY_BUFFER_ARB, vboId1); // for vertex coordinates
glBindBufferARB(GL_ELEMENT_ARRAY_BUFFER_ARB, vboId2); // for indices

// do same as vertex array except pointer
glEnableClientState(GL_VERTEX_ARRAY); // activate vertex coords array
glVertexPointer(3, GL_FLOAT, 0, 0); // last param is offset, not ptr

// draw 6 quads using offset of index array
glDrawElements(GL_QUADS, 24, GL_UNSIGNED_BYTE, 0);

glDisableClientState(GL_VERTEX_ARRAY); // deactivate vertex array

// bind with 0, so, switch back to normal pointer operation
glBindBufferARB(GL_ARRAY_BUFFER_ARB, 0);
glBindBufferARB(GL_ELEMENT_ARRAY_BUFFER_ARB, 0);
```
A **FBO** is set of Buffer Objects (everything that you need to draw in OpenGL)
Outline

- Review: GPU Architectures
- ISAs vs APIs
- Cuda Memory (ISA)
- OpenGL Memory (API)
- Memory Mapping
- Advanced Streaming
- Texture Memory Layout
Outline

- Review: GPU Architectures
- ISAs vs APIs
- Cuda Memory (ISA)
- OpenGL Memory (API)
- Memory Mapping
- Advanced Streaming
- Texture Memory Layout
CUDA->OpenGL Texture

// map OpenGL buffer object for writing from CUDA on a single GPU
// no data is moved (Win & Linux). When mapped to CUDA, OpenGL
// should not use this buffer
cudaGLMapBufferObject((void**)&dptr, pbo);

// execute the kernel
const int image_width = 512;
const int image_height = 512;
launch_kernel(dptr, image_width, image_height);

// unmap buffer object
cudaGLUnmapBufferObject(pbo);
Memory Mapping

Pixel Buffer Object

CUDA Memory

CPU
- Previous Instructions
- Start Memory Map
- Launch Kernel
- End Memory Map
- More Instructions

GPU
- Previous Kernel
- Previous Kernel
- Previous Kernel
- Start Memory Map
- Run Kernel
- End Memory Map
- Next Kernel

Tuesday, January 25, 2011
Project 1 Outline

You implement this!

Simple_Kernel

CUDA Memory

Pixel Buffer Object

Texture

Tuesday, January 25, 2011
Outline

- Review: GPU Architectures
- ISAs vs APIs
- Cuda Memory (ISA)
- OpenGL Memory (API)
- Memory Mapping
- Advanced Streaming
- Texture Memory Layout
Outline

• Review: GPU Architectures
• ISAs vs APIs
• Cuda Memory (ISA)
• OpenGL Memory (API)
• Memory Mapping
• Advanced Streaming
• Texture Memory Layout
Streaming Texture Uploads
Asynchronous Readback
Outline

- Review: GPU Architectures
- ISAs vs APIs
- Cuda Memory (ISA)
- OpenGL Memory (API)
- Memory Mapping
- Advanced Streaming
- Texture Memory Layout
Outline

- Review: GPU Architectures
- ISAs vs APIs
- Cuda Memory (ISA)
- OpenGL Memory (API)
- Memory Mapping
- Advanced Streaming
- Texture Memory Layout
Texture Memory Layout

Chalk Board
Outline

- Review: GPU Architectures
- ISAs vs APIs
- Cuda Memory (ISA)
- OpenGL Memory (API)
- Memory Mapping
- Advanced Streaming
- Texture Memory Layout
Outline

• Review: GPU Architectures
• ISAs vs APIs
• Cuda Memory (ISA)
• OpenGL Memory (API)
• Memory Mapping
• Advanced Streaming
• Texture Memory Layout