A GPU Renders 3D Scenes

• A Graphics Processing Unit (GPU) accelerates rendering of 3D scenes
  - Input: description of scene
  - Output: colored pixels to be displayed on a screen
• Input:
  - Geometry (triangles), colors, lights, effects, textures
• Output:

GPU Scene Complexity Defined by Standard Interfaces (DirectX and OpenGL)

• DirectX and OpenGL define the interface between applications and the GPU
• Geometry describes the objects and layout
  - Triangles (vertices) describe all objects
  - Can have millions of triangles per scene
  - Can modify triangle surfaces
  - Bumps, ripples...
  - Lights are part of the scene geometry
• Pixel Shaders describe how to add color
  - Colors of triangle vertices
  - Textures (patterns)
  - How to determine color of pixels within a triangle
  - ...

Complexity and Quality are Orders of Magnitude Better

GPU Performance is Increasing Much Faster than CPUs
GPU and CPU Architectures are Starting to Converge

<table>
<thead>
<tr>
<th>Year</th>
<th>CPUs</th>
<th>GPUs</th>
</tr>
</thead>
<tbody>
<tr>
<td>1997</td>
<td>no explicit parallelism</td>
<td>not programmable</td>
</tr>
<tr>
<td>2000</td>
<td>explicit short vectors</td>
<td>emerging programmability</td>
</tr>
<tr>
<td>2003</td>
<td>explicit short vectors</td>
<td>fully programmable explicit &quot;infinite&quot; DP</td>
</tr>
<tr>
<td>2006</td>
<td>explicit short vectors</td>
<td>explicit threading (~2)</td>
</tr>
<tr>
<td>2009</td>
<td>explicit short vectors</td>
<td>explicit threading (~16)</td>
</tr>
</tbody>
</table>

Outline

- What is a GPU?
- Why should we care about GPUs?
- 3D graphics pipeline
- Programmable GPUs

- Many slides courtesy David Kirk (NVIDIA) and Wen-Mei Hwu (UIUC)
- Other slides courtesy Massimiliano Fatica (NVIDIA)

Filtering techniques

- **Point sampling:**
  - pixel values are calculated by choosing one texture pixel (texel) color
- **Bilinear filtering:**
  - interpolating color from 4 neighboring texels. This gives a smoothing (if somewhat blurry) effect and makes the scene look more natural and prevents abrupt transitions between neighboring texels.
- **Tri-linear filtering:**
  - interpolating bilinearly filtered samples from two mip-maps. Tri-linear mip-mapping prevents moving objects from displaying a distracting "sparkle" caused by abrupt transitions between mip-maps.
- **Anisotropic filtering:**
  - interpolating and filtering multiple samples from one or more mip-maps to better approximate very distorted textures. Gives a sharper effect when severe perspective correction is used. Tri-linear mipmapping blurs textures more.

Texture Cache

- Stores temporally local texel values to reduce bandwidth requirements
- Due to nature of texture filtering high degrees of efficiency are possible
- Efficient texture caches can achieve 75% or better hit rates
- Reduces texture (memory) bandwidth by a factor of four for bilinear filtering

Pixel Shading

- **1999 (DirectX 7):**
  - Application could select from a few simple combinations of texture and interpolated color
  - Add
  - Decal
  - Modulate
- **Next (DirectX 9):**
  - Write a general program that executes for every pixel with a nearly unlimited number of interpolated inputs, texture lookups and math operations.
  - Can afford to perform sophisticated lighting calculations at every pixel
ROP (from Raster Operations)

- C-ROP performs frame buffer blending
  - Combinations of colors and transparency
  - Anti-aliasing
  - Read/Modify/Write the Color Buffer
- Z-ROP performs the Z operations
  - Determine the visible pixels
  - Discard the occluded pixels
  - Read/Modify/Write the Z-Buffer
- ROP on GeForce also performs
  - "Coalescing" of transactions
  - Z-Buffer compression/decompression

Alpha Blending

- Alpha Blending is used to render translucent objects.
- The pixel's alpha component contains its opacity.
- Read-modify-write operation to the color framebuffer
- Result = alpha * Src + (1-alpha) * Dst

Anti-Aliasing

- Aliased rendering: color sample at pixel center is the color of the whole pixel
- Anti-aliasing accounts for the contribution of all the primitives that intersect the pixel

Frame Buffer Interface (FBI)

- Manages reading from and writing to frame buffer
- Perhaps the most performance-critical component of a GPU
- GeForce's FBI is a crossbar
- Independent memory controllers for 4+ independent memory banks for more efficient access to frame buffer

The Frame Buffer

- The primary determinant of graphics performance other than the GPU
- The most expensive component of a graphics product other than the GPU
- Memory bandwidth is the key
- Frame buffer size also determines
  - Local texture storage
  - Maximum resolutions
  - AA resolution limits
Z Buffer

- A Z buffer is a 2-D array of Z values with the same (x,y) dimensions as the color framebuffer.
- Every candidate pixel from the shader has a calculated Z value along with its R,G,B,A color.
- Before writing the color, perform the Z buffer test:
  - Read the Z value from memory
  - Compare the candidate Z to the Z from memory; if the candidate Z is NOT in front of the previous Z, discard the pixel
  - Otherwise, write the new Z value to the Z buffer and write (or blend) the new color to the color framebuffer.

Summary, so far...

- Introduction to several key 3D graphics concepts
  - Framebuffers
  - Object Representation
  - Vertex Processing
  - Lighting
  - Texturing
  - Gouraud Interpolation
  - Texture Mapping
  - Pixel Shading
  - Alpha Blending
  - Anti-Aliasing
  - Z Buffering

Outline

- What is a GPU?
- Why should we care about GPUs?
- Programmable GPUs
  - Many slides courtesy David Kirk (NVIDIA) and Wen-Mei Hwu (UIUC)
  - Other slides courtesy Massimiliano Fatica (NVIDIA)
### Vertex and Fragment Processing Share Unified Processing Elements

- Load balancing SW is easier

![Unified Shader](image1.png)

- Heavy Geometry Workload Perf = 11

- Heavy Pixel Workload Perf = 11

### Vertex and Fragment Processing is Dynamically Load Balanced

- Less Geometry
  - High pixel shader use
  - Balanced use of pixel shader and vertex shader

- More Geometry
  - Low vertex shader use

### Make the Compute Core The Focus of the Architecture

- Processors execute computing threads
- Alternative operating mode specifically for computing

### The NVIDIA GeForce Graphics Pipeline

- Host
- Vertex Control
- Vertex Shader
- Triangle Setup
- Raster
- Shader
- Texture Cache
- Frame Buffer Memory
- Frame Buffer Memory

### Another View of the 3D Graphics Pipeline

- Host
- Vertex Control
- Vertex Shader
- Triangle Setup
- Raster
- Shader
- Texture Cache
- Frame Buffer Memory
- Frame Buffer Memory

- FS 0
  - Fragment Shader
  - ROP

- FS 1
  - Fragment Shader
  - ROP

- ROP
- Frame Buffer Memory
- Frame Buffer Memory
- Frame Buffer Memory
Stream Execution Model

- Data parallel streams of data
- Processing kernels
  - Unit of execution is processing of one stream element in one kernel - defined as a thread

Stream Execution Model

- Can partition the streams into chunks
  - Streams are very long and elements are independent
  - Chunks are called strips or blocks
- Unit of execution is processing one block of data by one kernel - defined as a thread block

This talk

- Three key concepts behind how modern architectures run “shader” code
- Knowing these concepts will help you:
  1. Understand space of GPU shader core (and throughput CPU processing core) designs
  2. Optimize shader/compute kernels
  3. Establish intuition: what workloads might benefit from the design of these architectures?

From Shader Code to a Teraflop: How Shader Cores Work

Kayvon Fatahalian
Stanford University

What’s in a GPU?

A diffuse reflectance shader

```
sampler mySamp;
Texture2D<float3> myTex;
float3 lightDir;
float4 diffuseShader(float3 nore, float2 uv)
{
    float3 kd;
    kd = myTex.Sample(mySamp, uv);
    kd = clamp(dot(lightDir, nore), 0.0, 1.0);
    return float4(kd, 1.0);
}
```

Heterogeneous chip multi-processor (highly tuned for graphics)

Independent – data parallelism
Compile shader

1 unshaded fragment input record

- Shader compilation
  - Shader to GLSL
  - Compile shader
    - sampler
      - mySamp
    - Texture2D<float3>
      - myTex

1 shaded fragment output record

Execute shader

Fetch/Decode

- ALU (Execute)
  - Execution Context
    - float3 lightDir
    - float3 v0
    - float3 r0, v4, t0, s0
    - float4 diffuseShader(float3 norm, float2 uv)

- ALU (Execute)
  - Execution Context
    - float3 lightDir
    - float3 v0
    - float3 r0, v4, t0, s0
    - float4 diffuseShader(float3 norm, float2 uv)

- ALU (Execute)
  - Execution Context
    - float3 lightDir
    - float3 v0
    - float3 r0, v4, t0, s0
    - float4 diffuseShader(float3 norm, float2 uv)

- ALU (Execute)
  - Execution Context
    - float3 lightDir
    - float3 v0
    - float3 r0, v4, t0, s0
    - float4 diffuseShader(float3 norm, float2 uv)

- ALU (Execute)
  - Execution Context
    - float3 lightDir
    - float3 v0
    - float3 r0, v4, t0, s0
    - float4 diffuseShader(float3 norm, float2 uv)

- ALU (Execute)
  - Execution Context
    - float3 lightDir
    - float3 v0
    - float3 r0, v4, t0, s0
    - float4 diffuseShader(float3 norm, float2 uv)

- ALU (Execute)
  - Execution Context
    - float3 lightDir
    - float3 v0
    - float3 r0, v4, t0, s0
    - float4 diffuseShader(float3 norm, float2 uv)
Execute shader

CPU-"style" cores

Slimming down

Idea #1:
Remove components that help a single instruction stream run fast

Two cores (two fragments in parallel)

Four cores (four fragments in parallel)

Sixteen cores (sixteen fragments in parallel)

16 cores = 16 simultaneous instruction streams
**Instruction stream coherence**

But... many fragments should be able to share an instruction stream!

Recall: simple processing core

Add ALUs

Idea #2:
Amortize cost/complexity of managing an instruction stream across many ALUs

SIMD processing

Modifying the shader

Original compiled shader:
Processes one fragment using scalar ops on scalar registers

New compiled shader:
Processes 8 fragments using vector ops on vector registers
But what about branches?

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

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

16 cores = 128 ALUs
= 16 simultaneous instruction streams
**Clarification**

SIMD processing does not imply SIMD instructions

- Option 1: Explicit vector instructions
  - Intel/AMD x86 SSE, Intel Larrabee
- Option 2: Scalar instructions, implicit HW vectorization
  - HW determines instruction stream sharing across ALUs (amount of sharing hidden from software)
  - NVIDIA GeForce ("SIMT" warps), ATIRadeon architectures

In practice: 16 to 64 fragments share an instruction stream

**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

We’ve removed the fancy caches and logic that helps avoid stalls.

**Idea #3:**

Interleave processing of many fragments on a single core to avoid stalls caused by high latency operations.

**Hiding shader stalls**

Time (clocks)

Frag 1, 8, 16, 24, 32

Runnable

Shared Data

Stall

If we have LOTS of independent fragments.

But we have LOTS of independent fragments.
Hiding shader stalls

Time (clocks)

Stall
Runnable

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

Increase run time of one group
To maximum throughput of many groups

Storing contexts

Pool of context storage
32KB

Twenty small contexts

(maximal latency hiding ability)

Storing contexts

Pool of context storage
32KB

Twelve medium contexts

Throughput!
Summary: three key ideas

1. Use many “slimmed down cores” to run in parallel

2. Pack cores full of ALUs (by sharing instruction stream across groups of fragments)
   - Option 1: Explicit SIMD vector instructions
   - Option 2: Implicit sharing managed by hardware

3. Avoid latency stalls by interleaving execution of many groups of fragments
   - When one group stalls, work on another group