GPU Architecture

Mike Houston
Acknowledgements

• Kayvon Fatahalian
  – Most of these slides are Kayvon’s

• Justin Hensley
Today

Review: the graphics pipeline

History: a few old GPUs

How a modern GPU works (and why it is so fast!)

Closer look at a real GPU design

- NVIDIA GTX 285
- AMD HD5870
Part 1:
The graphics pipeline
Vertex processing

Vertices are transformed into “screen space”
Vertex processing

Vertices are transformed into “screen space”

Each vertex is transformed independently
Primitive processing

Then organized into primitives that are clipped and culled…
Rasterization

Primitives are rasterized into "pixel fragments"
Rasterization

Primitives are rasterized into “pixel fragments”

EACH PRIMITIVE IS RASTERIZED INDEPENDENTLY
Fragment processing

Fragments are shaded to compute a color at each pixel.

CS448S – Spring 2010 – Beyond Programmable Shading
Fragment processing

Fragments are shaded to compute a color at each pixel
Pixel operations

Fragments are blended into the frame buffer at their pixel locations (z-buffer determines visibility)
Pipeline entities

Vertices

Primitives

Fragments

Fragments (shaded)

Pixels

CS448S – Spring 2010 – Beyond Programmable Shading
Graphics pipeline

- **Vertex Generation**
  - Vertex stream
  - Fixed-function

- **Vertex Processing**
  - Vertex stream
  - Programmable

- **Primitive Generation**
  - Primitive stream

- **Primitive Processing**
  - Primitive stream

- **Fragment Generation**
  - Fragment stream

- **Fragment Processing**
  - Fragment stream

- **Pixel Operations**

- **Memory Buffers**
  - Vertex Data Buffers
  - Fixed-function
  - Programmable
  - Textures
  - Output image (pixels)
Part 2: Graphics Architectures
Independent

• What’s so important about “independent” computations?

“graphics supercomputer”

Vertex Generation
Vertex Processing
Primitive Generation
Primitive Processing
Fragment Generation
Fragment Processing
Pixel Operations
Pre-1999 PC 3D graphics accelerator

CPU

Vertex Generation

Vertex Processing

Primitive Generation

Primitive Processing

Fragment Generation

Fragment Processing

Pixel Operations

3dfx Voodoo
NVIDIA RIVA
TNT

Clip/cull/rasterize

Tex

Tex

Pixel operations
GPU circa 1999

NVIDIA GeForce 256
Direct3D 9 programmability: 2002

ATI Radeon 9700
Direct3D 10 programmability: 2006

NVIDIA GeForce 8800
("unified shading" GPU)
Part 3: How a shader core works
GPUs are fast

Intel Core i7 Quad
~100 GFLOPS peak
730 million transistors
(obtainable if you code your program to use 4 threads and SSE vector instruction)

AMD Radeon HD 5870
~2.7 TFLOPS peak
2.2 billion transistors
(obtainable if you write OpenGL/DX/OpenCL programs)
A diffuse reflectance fragment 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);
}
```

Independent, but no explicit parallelism in the code
Big guy lookin’ diffuse
Compile shader

1 unshaded fragment input record

```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);
}
```

1 shaded fragment output record

```cpp
<diffuseShader>:
    sample r0, v4, t0, s0
    mul r3, v0, cb0[0]
    madd r3, v1, cb0[1], r3
    madd r3, v2, cb0[2], r3
    clmp r3, r3, l(0.0), l(1.0)
    mul o0, r0, r3
    mul o1, r1, r3
    mul o2, r2, r3
    mov o3, l(1.0)
```
Execute shader

<diffuseShader>:
sample r0, v4, t0, s0
mul r3, v0, cb0[0]
madd r3, v1, cb0[1], r3
madd r3, v2, cb0[2], r3
clmp r3, r3, l(0.0), l(1.0)
mul o0, r0, r3
mul o1, r1, r3
mul o2, r2, r3
mov o3, l(1.0)
Execute shader

<diffuseShader>:
sample r0, v4, t0, s0
mul  r3, v0, cb0[0]
madd r3, v1, cb0[1], r3
madd r3, v2, cb0[2], r3
clmp r3, r3, l(0.0), l(1.0)
mul  o0, r0, r3
mul  o1, r1, r3
mul  o2, r2, r3
mov  o3, l(1.0)
Execute shader

Fetch/Decode

ALU (Execute)

Execution Context

<diffuseShader>:
sample r0, v4, t0, s0
mul r3, v0, cb0[0]
madd r3, v1, cb0[1], r3
madd r3, v2, cb0[2], r3
cmp r3, r3, l(0.0), l(1.0)
mul o0, r0, r3
mul o1, r1, r3
mul o2, r2, r3
mov o3, l(1.0)
Execute shader

`<diffuseShader>`:
- `sample r0, v4, t0, s0`
- `mul r3, v0, cb0[0]`
- `madd r3, v1, cb0[1], r3`
- `madd r3, v2, cb0[2], r3`
- `clmp r3, r3, l(0.0), l(1.0)`
- `mul o0, r0, r3`
- `mul o1, r1, r3`
- `mul o2, r2, r3`
- `mov o3, l(1.0)`
Execute shader

Fetch/Decode

ALU (Execute)

Execution Context

<diffuseShader>:
  sample r0, v4, t0, s0
  mul  r3, v0, cb0[0]
  madd r3, v1, cb0[1], r3
  madd r3, v2, cb0[2], r3
  clmp r3, r3, l(0.0), l(1.0)
  mul  o0, r0, r3
  mul  o1, r1, r3
  mul  o2, r2, r3
  mov  o3, l(1.0)
Execute shader

<diffuseShader>:
sample r0, v4, t0, s0
mul r3, v0, cb0[0]
madd r3, v1, cb0[1], r3
madd r3, v2, cb0[2], r3
clmp r3, r3, l(0.0), l(1.0)
mul o0, r0, r3
mul o1, r1, r3
mul o2, r2, r3
mov o3, l(1.0)
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)
Slimming down

Idea #1:
Remove components that help a single instruction stream run fast
Two cores  (two fragments in parallel)

fragment 1

<diffuseShader>:
sample r0, v4, t0, s0
mul r3, v0, cb0[0]
madd r3, v1, cb0[1], r3
madd r3, v2, cb0[2], r3
clmp r3, r3, l(0.0), l(1.0)
mul o0, r0, r3
mul o1, r1, r3
mul o2, r2, r3
mov o3, l(1.0)

fragment 2

<diffuseShader>:
sample r0, v4, t0, s0
mul r3, v0, cb0[0]
madd r3, v1, cb0[1], r3
madd r3, v2, cb0[2], r3
clmp r3, r3, l(0.0), l(1.0)
mul o0, r0, r3
mul o1, r1, r3
mul o2, r2, r3
mov o3, l(1.0)
Four cores (four fragments in parallel)
Sixteen cores (sixteen fragments in parallel)

16 cores = 16 simultaneous instruction streams
Instruction stream sharing

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

<diffuseShader>:
sample r0, v4, t0, s0
mul r3, v0, cb0[0]
madd r3, v1, cb0[1], r3
madd r3, v2, cb0[2], r3
clmp r3, r3, l(0.0), l(1.0)
mul o0, r0, r3
mul o1, r1, r3
mul o2, r2, r3
mov o3, l(1.0)
Recall: simple processing core
Add ALUs

Idea #2:

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

SIMD processing

(SIMD = single-instruction, multiple-data)
Modifying the shader

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

<diffuseShader>:
sample r0, v4, t0, s0
mul r3, v0, cb0[0]
madd r3, v1, cb0[1], r3
madd r3, v2, cb0[2], r3
clmp r3, r3, l(0.0), l(1.0)
mul o0, r0, r3
mul o1, r1, r3
mul o2, r2, r3
mov o3, l(1.0)
Modifying the shader

Processes 8 fragments using vector ops on vector registers

New compiled shader:

```
<VEC8_diffuseShader>:
VEC8_sample vec_r0, vec_v4, t0, vec_s0
VEC8_mul  vec_r3, vec_v0, cb0[0]
VEC8_madd vec_r3, vec_v1, cb0[1], vec_r3
VEC8_madd vec_r3, vec_v2, cb0[2], vec_r3
VEC8_clmp vec_r3, vec_r3, l(0.0), l(1.0)
VEC8_mul  vec_o0, vec_r0, vec_r3
VEC8_mul  vec_o1, vec_r1, vec_r3
VEC8_mul  vec_o2, vec_r2, vec_r3
VEC8_mov  vec_o3, l(1.0)
```
Modifying the shader

<VEC8_diffuseShader>:
VEC8_sample vec_r0, vec_v4, t0, vec_s0
VEC8_mul  vec_r3, vec_v0, cb0[0]
VEC8_madd vec_r3, vec_v1, cb0[1], vec_r3
VEC8_madd vec_r3, vec_v2, cb0[2], vec_r3
VEC8_clmp vec_r3, vec_r3, l(0.0), l(1.0)
VEC8_mul  vec_o0, vec_r0, vec_r3
VEC8_mul  vec_o1, vec_r1, vec_r3
VEC8_mul  vec_o2, vec_r2, vec_r3
VEC8_mov  vec_o3, l(1.0)
128 fragments in parallel

16 cores = 128 ALUs
= 16 simultaneous instruction streams
128 ] in parallel

Vertices
fragments
primitives

primitives

vertices

fragments
But what about branches?

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

<unconditional shader code>

<resume unconditional shader code>
But what about branches?

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

<unconditional shader code>

<resume unconditional shader code>
But what about branches?

Not all ALUs do useful work!

Worst case: 1/8 performance

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?

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

<unconditional shader code>
<resume unconditional shader code>
Wide SIMD processing

In practice:
16 to 64 fragments share an instruction stream
Stalls!

Stalls occur when a core cannot run the next shader instruction because it's waiting on a previous operation.
A diffuse reflectance shader

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
Recall: CPU-“style” core
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).
But we have **LOTS** of independent fragments.

**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

Fetch/Decode

ALU ALU ALU ALU

ALU ALU ALU ALU

CTX CTX CTX CTX

CTX CTX CTX CTX

Shared Ctx Data
Hiding shader stalls

Time (clocks)

Frag 1 ... 8

Frag 9 ... 16

Frag 17 ... 24

Frag 25 ... 32

Fetch/Decode

ALU

ALU

ALU

ALU

ALU

ALU

ALU

ALU
Hiding shader stalls

Time (clocks)

Frag 1 … 8

Frag 9… 16

Frag 17 … 24

Frag 25 … 32

Stall

Runnable
Hiding shader stalls

Time (clocks)

- Frag 1 … 8
- Frag 9 … 16
- Frag 17 … 24
- Frag 25 … 32

Stall

Runnable
Hiding shader stalls

Time (clocks)

Frag 1 … 8
Runnable

Frag 9 … 16
Runnable

Frag 17 … 24
Runnable

Frag 25 … 32
Runnable

Stall

Stall

Stall

Stall

Runnable

Runnable

Runnable

Runnable
High-throughput computing!

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

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

Pool of context storage
64 KB
Twenty small contexts

(maximal latency hiding ability)
Twelve medium contexts

[Diagram showing 12 medium contexts with ALU units arranged hierarchically.]
Four large contexts

(low latency hiding ability)
We just built a GPU shading system!

16 cores
8 mul-add ALUs per core (128 total)
16 simultaneous instruction streams
64 concurrent (but interleaved) instruction streams
512 concurrent fragments
= 256 GFLOPs (@ 1GHz)
Shader core 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)

3. Avoid latency stalls by interleaving execution of many groups of fragments
   - When one group stalls, work on another group
Think of a GPU as a multi-core processor optimized for maximum throughput when running vertex and fragment programs.

With special support on the side for: rasterization, clipping, culling, texturing…

and for **mapping** the graphics pipeline onto all these resources.
Bonus material:

A closer look at real GPUs:

NVIDIA GeForce GTX 285
ATI Radeon HD 5870
NVIDIA GeForce GTX 285
NVIDIA GeForce GTX 285 “core”

= SIMD functional unit, control shared across 8 units

= instruction stream decode

= execution context storage

= multiply

= multiply-add

64 KB of storage for fragment contexts (registers)
NVIDIA GeForce GTX 285 “core”

• Groups of 32 [fragments/vertices/prims] share instruction stream (they are called “WARPS”)
• Up to 32 groups are simultaneously interleaved
• Therefore: up to 1024 fragment contexts can be stored

64 KB of storage for fragment contexts (registers)
There are 30 of these things on the GTX 285: 30,000 fragments!
ATI Radeon HD 5870 “core”

- Groups of 64 [fragments/vertices/prims] share instruction stream (they are called “Wavefronts”)
ATI Radeon™ HD 5870
What is coming?
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
  – General R/W caches? (Fermi)
  – Synchronization?
Current and future: GPU programming

• Support for alternative programming interfaces
  – Non-graphics programming: OpenCL, DirectCompute
  – Applications treat GPUs as multi-core processor

• How does graphics pipeline abstraction change?
  – Direct3D 11 adds three new pipeline stages!
  – The world is very interested in ray-tracing (CS348b)
  – And other stuff (this course...)