How a GPU Works

Kayvon Fatahalian

Feb 9, 2011
Today

1. Review: the graphics pipeline

2. History: a few old GPUs

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

4. Closer look at a real GPU design
   - NVIDIA GTX 285
Part 1:
The graphics pipeline
(an abstraction)
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”
Primitives are rasterized into “pixel fragments”

EACH PRIMITIVE IS RASTERIZED INDEPENDENTLY
Fragment processing

Fragments are shaded to compute a color at each pixel.
Fragment processing

Fragments are shaded to compute a color at each pixel

EACH FRAGMENT IS PROCESSED INDEPENDENTLY
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
## Graphics pipeline

<table>
<thead>
<tr>
<th>Vertices</th>
<th>Vertex Generation</th>
<th>Vertex Processing</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Vertex stream</td>
<td>Vertex stream</td>
</tr>
<tr>
<td>Primitives</td>
<td>Primitive Generation</td>
<td>Primitive Processing</td>
</tr>
<tr>
<td></td>
<td>Primitive stream</td>
<td>Primitive stream</td>
</tr>
<tr>
<td>Fragments</td>
<td>Fragment Generation</td>
<td>Fragment Processing</td>
</tr>
<tr>
<td></td>
<td>Fragment stream</td>
<td>Fragment stream</td>
</tr>
<tr>
<td>Pixels</td>
<td>Pixel Operations</td>
<td></td>
</tr>
</tbody>
</table>

### Memory Buffers
- **Vertex Data Buffers**
- **Textures**
- **Output image (pixels)**

**Fixed-function**

**Programmable**

**Memory Buffers**

- **Textures**
- **Output image (pixels)**
Part 2: Graphics architectures

(implementation of the graphics pipeline)
Independent

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

“graphics supercomputer”

- System Bus
- Command Processor
- Geometry Engines
- Triangle Bus
- Fragment Generators
- Image Engines
- Display Generator Board
- Raster Memory Board
- Geometry Board
- Video output

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

3dfx Voodoo
NVIDIA RIVA TNT

CPU

- Vertex Generation
- Vertex Processing
- Primitive Generation
- Primitive Processing
- Fragment Generation
- Fragment Processing
- Pixel Operations

- Clip/cull/rasterize
- Tex
- Tex
- Pixel operations
GPU* circa 1999

CPU

NVIDIA GeForce 256

- Vertex Generation
- Vertex Processing
- Primitive Generation
- Primitive Processing
- Fragment Generation
- Fragment Processing
- Pixel Operations
Direct3D 9 programmability: 2002

ATI Radeon 9700
Direct3D 10 programmability: 2006

NVIDIA GeForce 8800
(“unified shading” GPU)
Part 3: How a shader core works (three key ideas)
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 instr)

AMD Radeon HD 5870

~2.7 TFLOPS peak
2.2 billion transistors

(obtainable if you write OpenGL programs like you’ve done in this class)
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

```assembly
<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

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

### ALU (Execute)

```assembly
<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**

```plaintext
<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

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

**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
clamp 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
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)

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
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)
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

- Fetch/Decode
- ALU (Execute)
- Execution Context
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
Modifying the shader

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

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?

```
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
But what about branches?

Time (clocks)

ALU 1  ALU 2  ...  ...  ALU 8

<table>
<thead>
<tr>
<th>T</th>
<th>T</th>
<th>F</th>
<th>T</th>
<th>F</th>
<th>F</th>
<th>F</th>
<th>F</th>
<th>F</th>
</tr>
</thead>
</table>

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
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-“style” memory hierarchy

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
Hiding shader stalls

Time (clocks)

Frag 1 … 8
00000000

Frag 9… 16
00000000

Frag 17 … 24
00000000

Frag 25 … 32
00000000

Fetch/Decode

ALU ALU ALU ALU

ALU ALU ALU ALU

ALU ALU ALU ALU

ALU ALU ALU ALU
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>

1. Stall
2. Runnable
Hiding shader stalls

Time (clocks)

Frag 1 … 8

Frag 9… 16

Frag 17 … 24

Frag 25 … 32

Runnable

Stall

1

2

3

4
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>1</td>
<td>Stall</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Runnable</td>
<td>Stall</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Stall</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Stall</td>
</tr>
</tbody>
</table>

60
High-throughput computing!

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
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)
My “enthusiast” GPU!

32 cores, 16 ALUs per core (512 total) = 1 TFLOP (@ 1 GHz)
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
Remember this!

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 a real GPU:

NVIDIA GeForce GTX 285
NVIDIA GeForce GTX 285
NVIDIA GeForce GTX 285 “core”

- = SIMD functional unit, control shared across 8 units
- = multiply-add
- = multiply
- = instruction stream decode
- = execution context storage

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
There are 30 of these things on the GTX 285: 30,000 fragments!
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
  – Traditional caches
Current and future: GPU programming

• Support for alternative programming interfaces
  – Non-graphics programming: OpenCL, CUDA
  – 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)