Lecture 23:
How GPUs work: from shader code to a TeraFLOP

Computer Graphics and Imaging
UC Berkeley CS184/284A

(slides by Kayvon Fatahalian)
Goal: Highly Complex 3D Scenes in Realtime

- Complex vertex and fragment shader computations
- 100’s of thousands to millions of triangles in a scene
- High resolution (2-4 megapixel + supersampling)
- 30-60 frames per second (even higher for VR)
A diffuse reflectance shader

sampler mySamp;
Texture2D<float3> myTex;
float3 lightDir;

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

How much compute is this?

4 multiply-adds & 1 texture fetch

4K = 8 MPixels
x 5x Overdraw = 40 MPixels / frame
x 60hz = 2.4 GPixels/sec

~10 GFLOPS
~10 GB/sec

A real game is 10s to 100s of times more!
This lecture

Three major ideas that make GPU processing cores run fast
   How can we exploit massive *parallelism* to run shaders fast?

Closer look at a real GPU design
   NVIDIA GTX 1080

The GPU memory hierarchy: moving data to processors
Part 1: throughput processing

Three key concepts behind how modern GPU processing cores run code

Knowing these concepts will help you:
1. Understand space of GPU core (and throughput CPU core) designs
2. Understand how “GPU” cores do (and don’t!) differ from “CPU” cores
3. Optimize shaders/compute kernels
4. Establish intuition: what workloads might benefit from the design of these architectures?
What’s in a GPU?

A GPU is a heterogeneous chip multi-processor (highly tuned for graphics)
A diffuse reflectance shader

```cpp
sampler mySamp;
Texture2D<float3> myTex;
float3 lightDir;

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

Shader programming model:

Fragments are processed *independently*, but there is no explicit parallel programming

Key architectural ideas:
How can we exploit **parallelism** to run faster?
Compile shader

1 unshaded fragment input record

```
sampler mySamp;
Texture2D<float3> myTex;
float3 lightDir;

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

1 shaded fragment output record

```
<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, 1(0.0), 1(1.0)
mul o0, r0, r3
mul o1, r1, r3
mul o2, r2, r3
mov o3, 1(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)
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, 1(0.0), 1(1.0)
mul o0, r0, r3
mul o1, r1, r3
mul o2, r2, r3
mov o3, 1(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
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
- Data cache (a big one)
- Out-of-order control logic
- Fancy branch predictor
- Memory pre-fetcher
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

- Fetch/Decode
- ALU (Execute)
- Execution Context
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

```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)
```
Modifying the shader

New compiled shader:

Processes eight fragments using vector ops on vector registers

```
<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 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  o3, l(1.0)
```
128 fragments in parallel

16 cores = 128 ALUs, 16 simultaneous instruction streams
128 [ vertices/fragments primitives OpenCL work items CUDA threads ] in parallel

vertices

primitives

fragments
But what about branches?

Time (clocks)

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

<unconditional shader code>

```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?

\[
\begin{array}{cccccc}
& \text{ALU 1} & \text{ALU 2} & \cdots & \cdots & \text{ALU 8} \\
1 & T & T & F & T & F & F & F & F \\
2 & T & F & T & F & F & F & F & F \\
\vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\
8 & T & T & F & T & F & F & F & F \\
\end{array}
\]

\[
\begin{align*}
\text{if } (x > 0) \{ \\
& y = \text{pow}(x, \text{exp}); \\
& y *= \text{Ks}; \\
& \text{refl} = y + \text{Ka}; \\
\} \text{ else } \{ \\
& x = 0; \\
& \text{refl} = \text{Ka}; \\
\} \\
\text{<resume unconditional shader code>}
\end{align*}
\]
But what about branches?

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

<unconditional shader code>

```c
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;
}
```

<y=
pow(x, exp);

y *= Ks;

refl = y + Ka;

{x = 0;
refl = Ka;

<resume unconditional shader code>
Clarification

SIMD processing does not imply SIMD instructions

Option 1: explicit vector instructions
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), ATI Radeon architectures (“wavefronts”)

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.
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 1  ALU 2  ALU 3  ALU 4
ALU 5  ALU 6  ALU 7  ALU 8

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 1  ALU 2  ALU 3  ALU 4
ALU 5  ALU 6  ALU 7  ALU 8

1  2  3  4
Hiding shader stalls

Time (clocks)

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

Runnable

Stall
Hiding shader stalls

Time (clocks)

Frag 1 … 8
Runnable

Frag 9 … 16
Stall

Frag 17 … 24
Stall

Frag 25 … 32
Stall
Throughput!

Increase run time of one group
to increase throughput of many groups
Storing contexts

Pool of context storage
128 KB
Eighteen small contexts  (maximal latency hiding)
Twelve medium contexts

Fetch/Decode

ALU 1
ALU 2
ALU 3
ALU 4
ALU 5
ALU 6
ALU 7
ALU 8

1  2  3  4
5  6  7  8
9  10 11 12
Four large contexts

(low latency hiding ability)
Our chip

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)
Our “enthusiast” chip

32 cores, 16 ALUs per core (512 total) = 1 TFLOP (@ 1 GHz)
Summary: three key ideas to exploit parallelism for performance

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
Part 2:
Putting the three ideas into practice:
A closer look at real GPUs

NVIDIA GeForce GTX 1080
NVIDIA GeForce GTX 1080

NVIDIA-speak:
2560 stream processors ("CUDA cores")
“SIMT execution”

Generic speak:
20 cores
4 groups of 32 SIMD functional units per core
NVIDIA GeForce GTX 1080 “core”

- Groups of 32 [fragments/vertices/CUDA threads] share an instruction stream
- Up to 64 groups are simultaneously interleaved
- Up to 2,048 individual contexts can be stored

Source: NVIDIA Pascal tuning guide
There are 20 of these things on the GTX 1080

That’s 40,960 fragments!

(Or 40,960 “CUDA threads”)

NVIDIA GeForce GTX 1080
Thank you

Original Slides:
Kayvon Fatahalian

Contributors:
Kurt Akeley
Solomon Boulos
Mike Doggett
Pat Hanrahan
Mike Houston
Jeremy Sugerman