

GPU hardware

Cosmin Oancea and Troels Henriksen

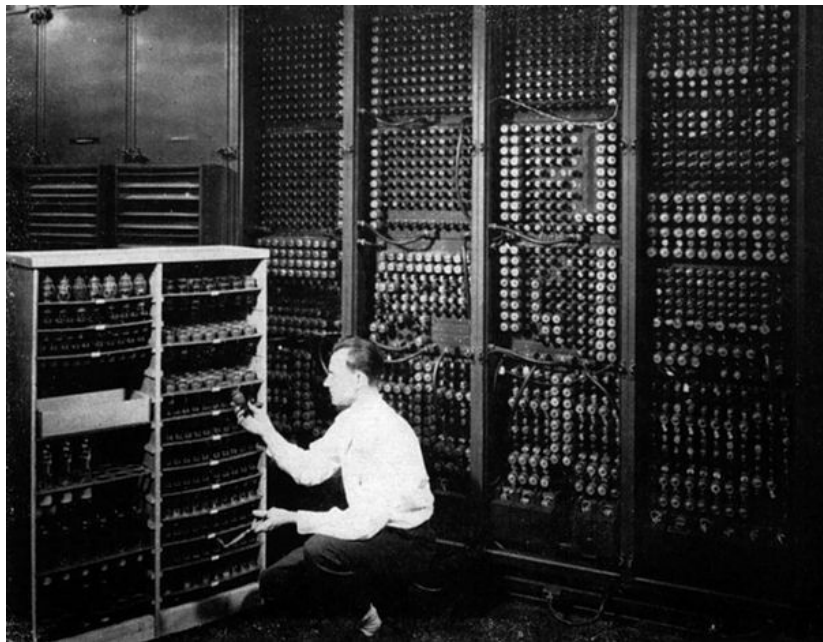
Spetember 2023

Hardware Trends

The GPU Architecture

The OpenCL Programming Model

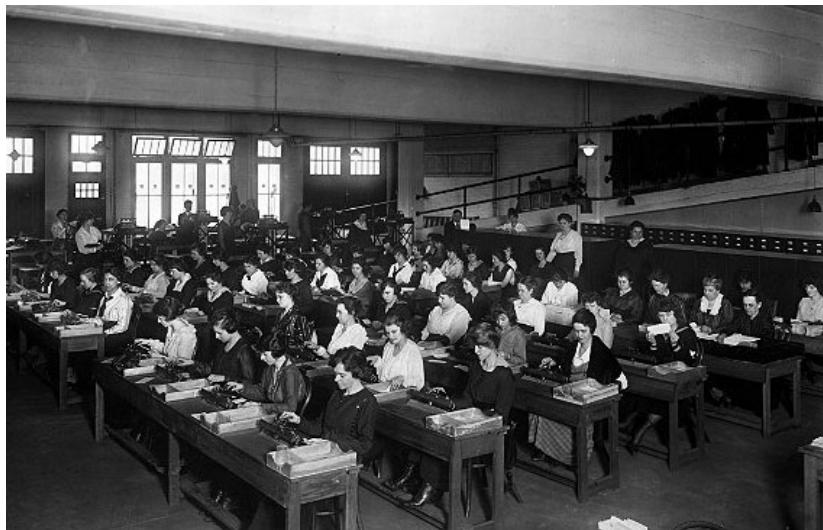
The first computers were not this



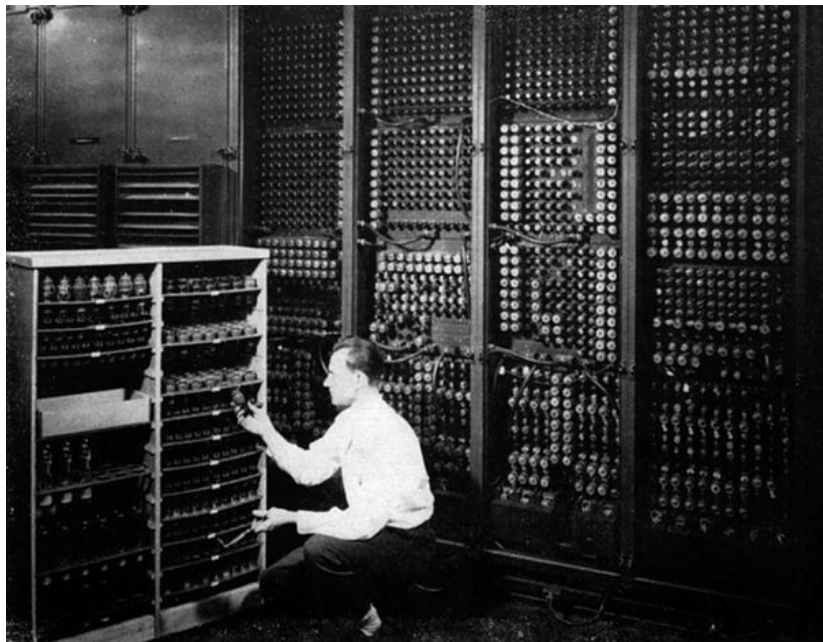
But this



And if you had a larger problem



But then they started looking like this



Then this



Then this



Then this



Then this



Then this



Then, from around 2005



Then, from around 2005



Then, from around 2005

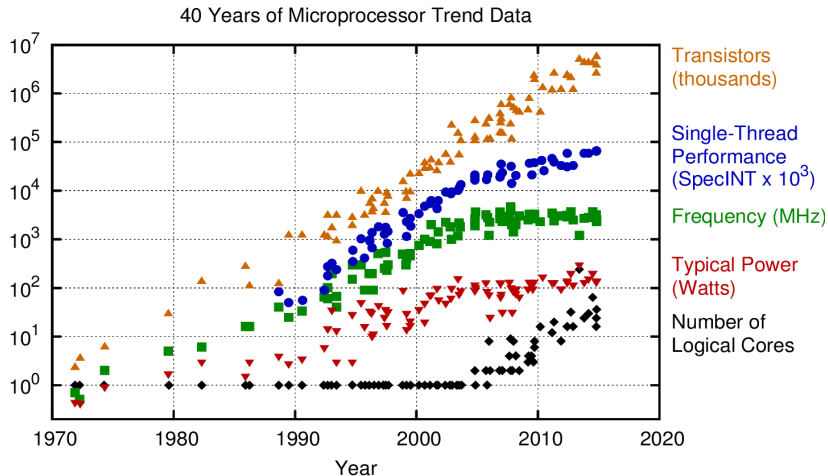


Improvements in *sequential performance* stalled, although computers still got smaller and faster.

What Changed?

- ▶ *Power complexity* $P_{dynamic} \sim Freq^3$, preventing us from increasing processor frequency.
- ▶ *Memory wall*, ever-increasing performance gap between processor and memory (which means that *memory* becomes bottleneck, not processor speed).

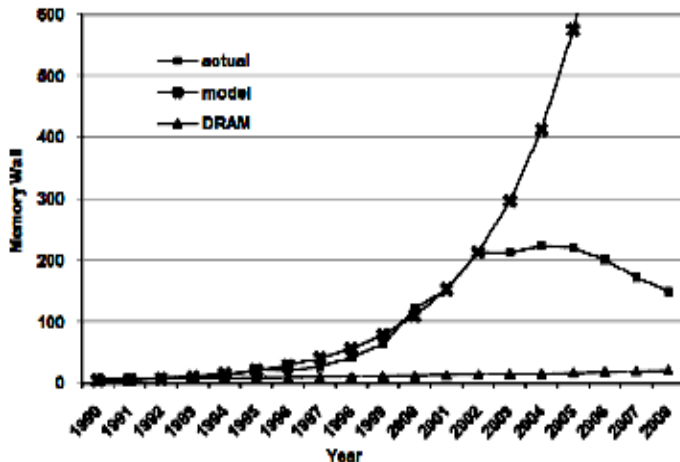
CPU progress



Original data up to the year 2010 collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond, and C. Batten
New plot and data collected for 2010-2015 by K. Rupp

Addressed with *more cores*.

The Memory Wall



Memory Wall = processor cycles/memory cycles

Addressed with caches (not scalable) and *latency hiding*.

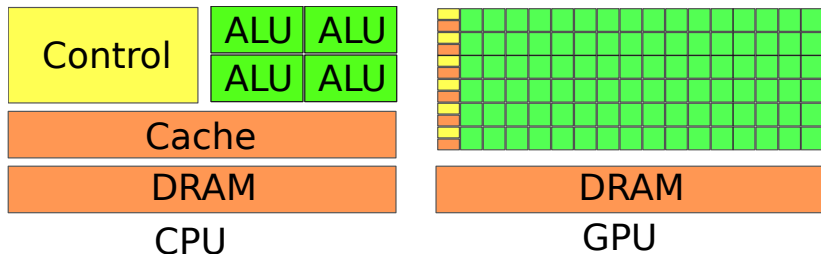
This is why GPUs are useful

The design of GPUs directly attacks these two problems.

- ▶ **Frequency scaling** becomes less of an issue because we can instead use thousands of (slower) cores.
- ▶ The **memory wall** is partially circumvented by using faster and smaller memory, but mostly by *latency hiding*. With tens of thousands of threads, we can probably find something else to do while some threads are waiting for memory!

Ultimately, GPUs do *throughput processing*, and operations have (relatively) high latency.

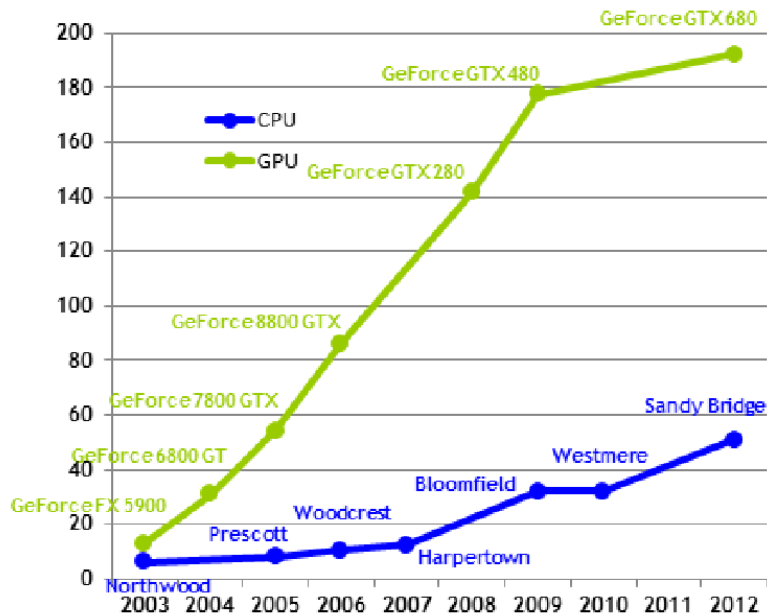
CPUs compared to GPUs



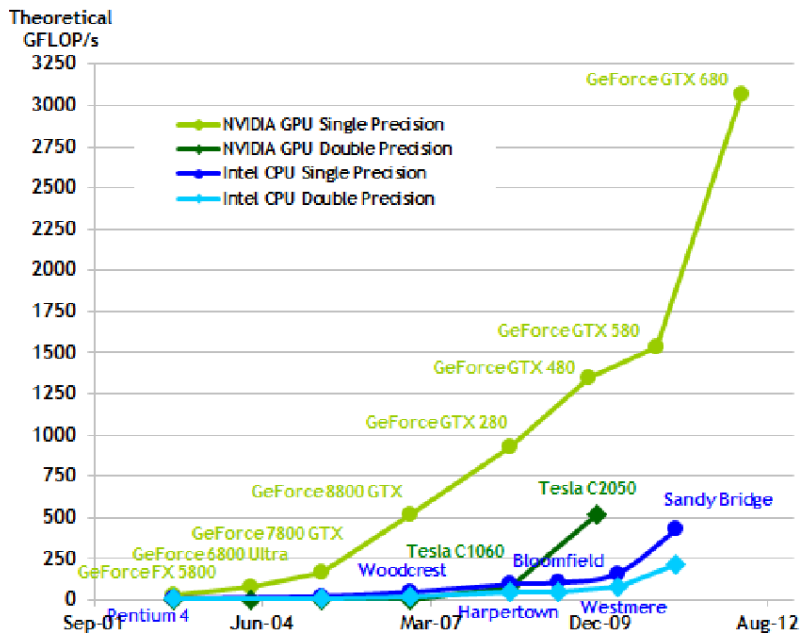
- ▶ GPUs have *thousands* of simple cores and taking full advantage of their compute power requires *tens of thousands* of threads.
- ▶ GPU threads are very *restricted* in what they can do: no stack, no allocation, limited control flow, etc.
- ▶ Potential *very high performance* and *lower power usage* compared to CPUs, but programming them is *hard*.

GPUs and Memory

Theoretical GB/s



GPUs and GFLOPS



Hardware Trends

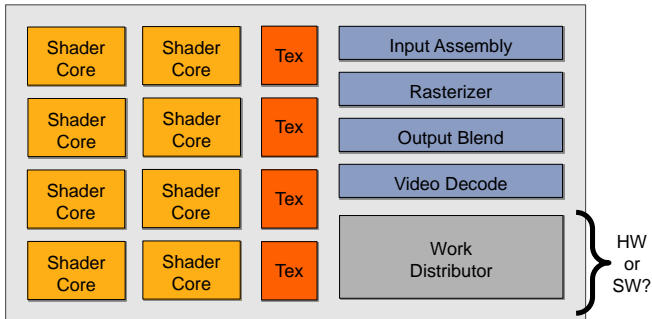
The GPU Architecture

The OpenCL Programming Model

The following slides are taken from the presentation *Introduction to GPU Architecture* by Ofer Rosenberg of AMD.

What's in a GPU?

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



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);  
}
```

Shader programming model:

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

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);  
}
```



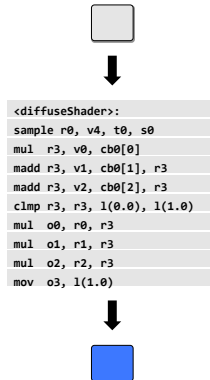
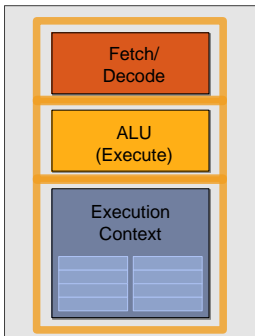
```
<diffuseShader>:  
sample r0, v4, t0, s0  
mul   r3, v0, cb0[0]  
madd  r3, v1, cb0[1], r3  
madd  r3, v2, cb0[2], r3  
clmp  r3, r3, 1(0.0), 1(1.0)  
mul   o0, r0, r3  
mul   o1, r1, r3  
mul   o2, r2, r3  
mov   o3, 1(1.0)
```



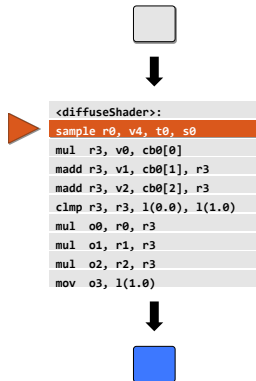
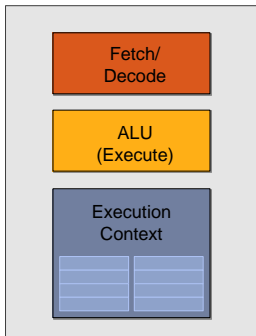
1 shaded fragment output record



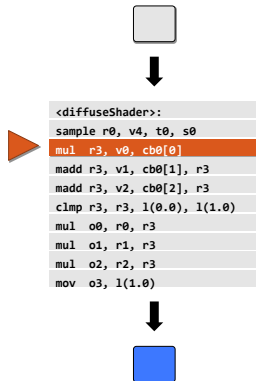
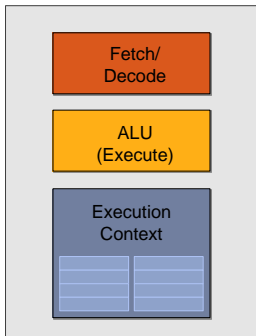
Execute shader



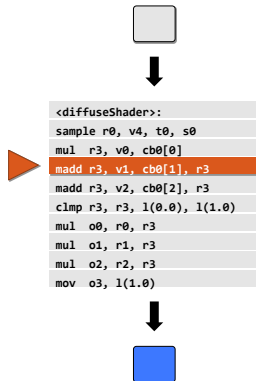
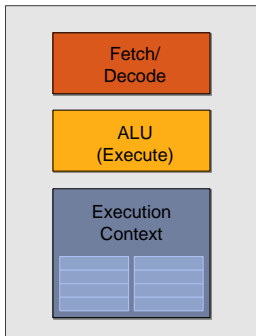
Execute shader



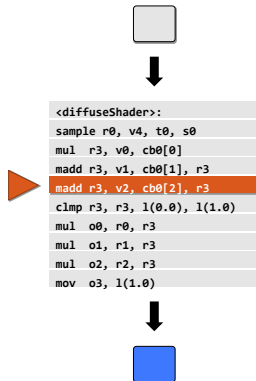
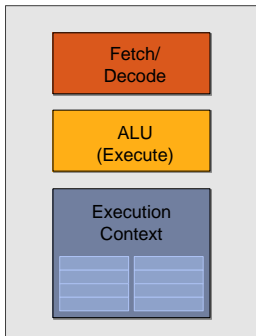
Execute shader



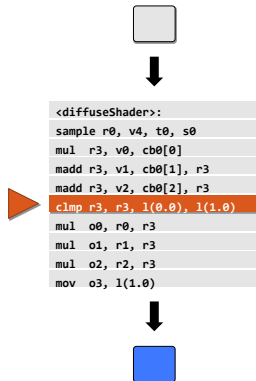
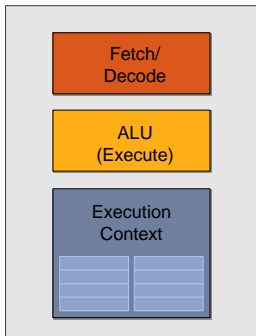
Execute shader



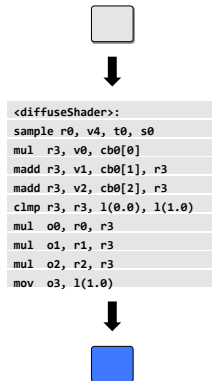
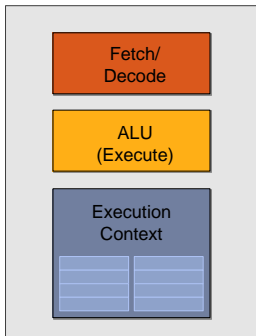
Execute shader



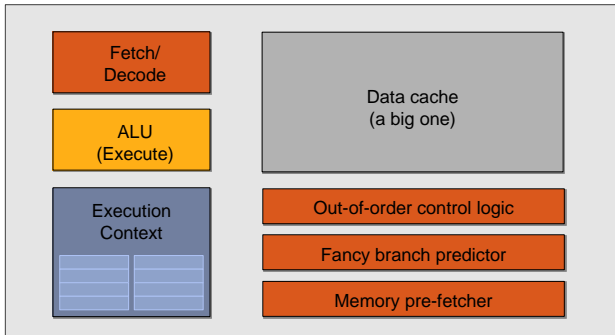
Execute shader



Execute shader



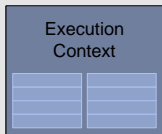
“CPU-style” cores



Slimming down

Fetch/
Decode

ALU
(Execute)



Idea #1:

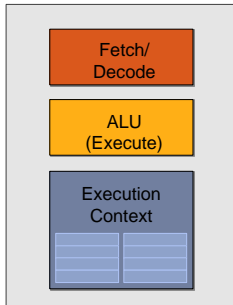
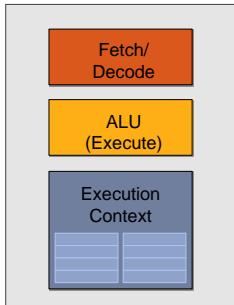
Remove components that
help a single instruction
stream run fast

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
clap 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)
```



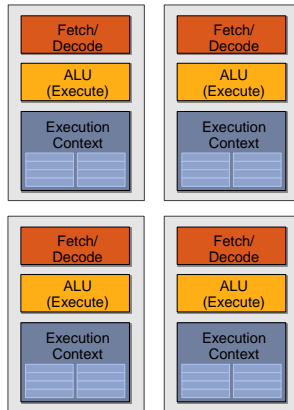
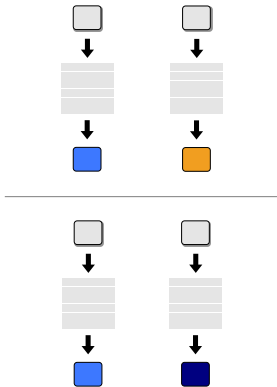
fragment 2



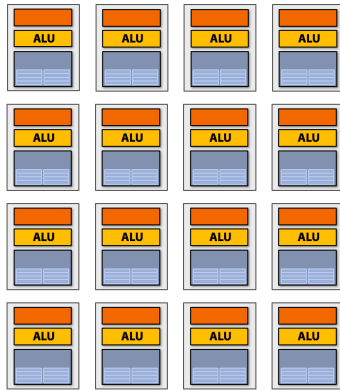
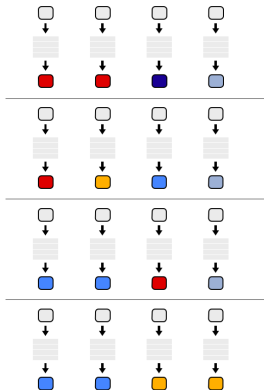
```
<diffuseShader>:
sample r0, v4, t0, s0
mul r3, v0, cb0[0]
madd r3, v1, cb0[1], r3
madd r3, v2, cb0[2], r3
clap r3, r3, 1(0.0), 1(1.0)
mul o0, r0, r3
mul o1, r1, r3
mul o2, r2, r3
mov o3, 1(1.0)
```



Four cores (four fragments in parallel)

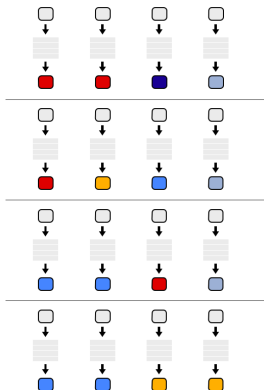


Sixteen cores (sixteen fragments in parallel)



16 cores = 16 simultaneous instruction streams

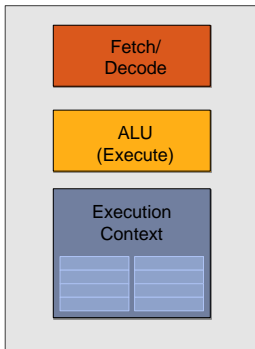
Instruction stream sharing



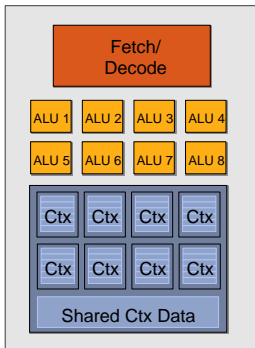
But ... many fragments
should be able to share an
instruction stream!

```
<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

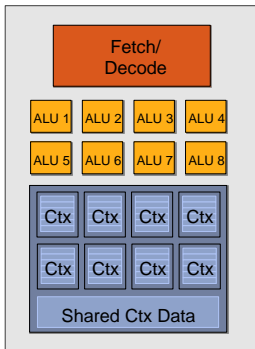


Idea #2:

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

SIMD processing

Modifying the shader

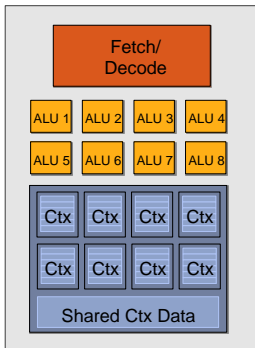


```
<diffuseShader>:  
sample r0, v4, t0, s0  
mul r3, v0, cb0[0]  
madd r3, v1, cb0[1], r3  
madd r3, v2, cb0[2], r3  
clmp r3, r3, 1(0.0), 1(1.0)  
mul o0, r0, r3  
mul o1, r1, r3  
mul o2, r2, r3  
mov o3, 1(1.0)
```

Original compiled shader:

Processes one fragment using
scalar ops on scalar registers

Modifying the shader

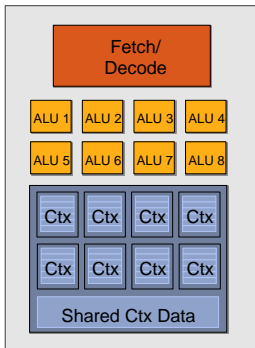


```
<VEC8_diffuseShader>:  
VEC8_sample vec_r0, vec_v4, t0, vec_s0  
VEC8_mul   vec_r3, vec_v0, cb0[0]  
VEC8_madd  vec_r3, vec_v1, cb0[1], vec_r3  
VEC8_madd  vec_r3, vec_v2, cb0[2], vec_r3  
VEC8_clmp  vec_r3, vec_r3, 1(0.0), 1(1.0)  
VEC8_mul   vec_o0, vec_r0, vec_r3  
VEC8_mul   vec_o1, vec_r1, vec_r3  
VEC8_mul   vec_o2, vec_r2, vec_r3  
VEC8_mov   o3, 1(1.0)
```

New compiled shader:

Processes eight fragments using
vector ops on vector registers

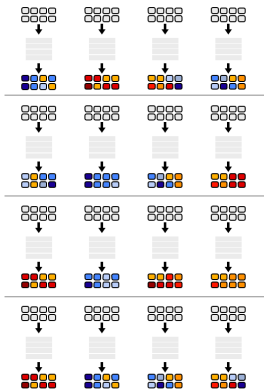
Modifying the shader



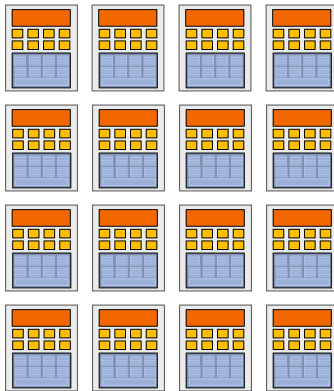
```
<VEC8_diffuseShader>:  
VEC8_sample vec_r0, vec_v4, t0, vec_s0  
VEC8_mul vec_r3, vec_v0, cb0[0]  
VEC8_madd vec_r3, vec_v1, cb0[1], vec_r3  
VEC8_madd vec_r3, vec_v2, cb0[2], vec_r3  
VEC8_clmp vec_r3, vec_r3, 1(0.0), 1(1.0)  
VEC8_mul vec_o0, vec_r0, vec_r3  
VEC8_mul vec_o1, vec_r1, vec_r3  
VEC8_mul vec_o2, vec_r2, vec_r3  
VEC8_mov o3, 1(1.0)
```



128 fragments in parallel

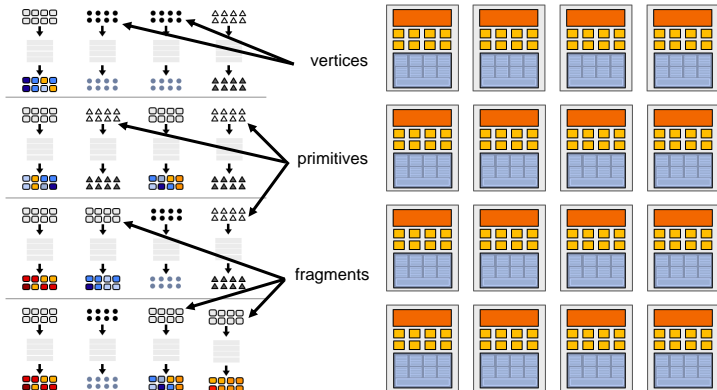


16 cores = 128 ALUs

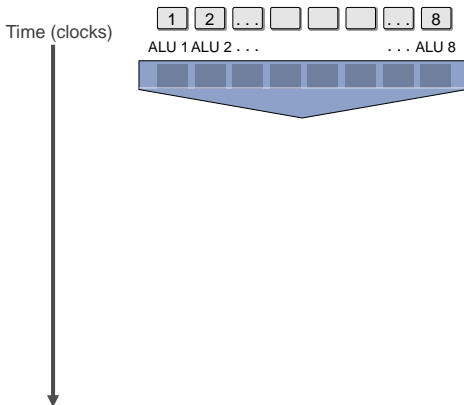


, 16 simultaneous instruction streams

128 [vertices/fragments
primitives
OpenCL work items] in parallel



But what about branches?

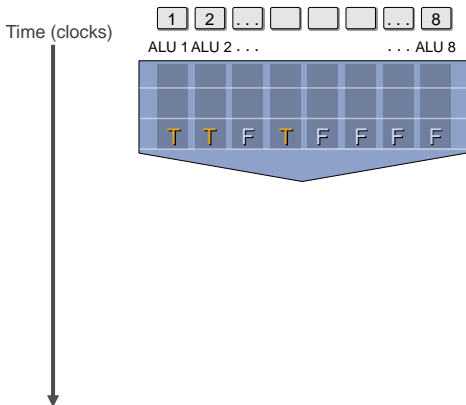


```
<unconditional  
shader code>
```

```
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?

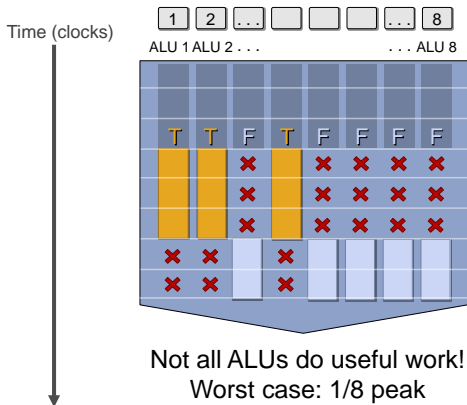


```
<unconditional  
shader code>
```

```
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?



<unconditional
shader code>

```
if (x > 0) {
```

```
    y = pow(x, exp);
```

```
    y *= Ks;
```

```
    refl = y + Ka;
```

```
} else {
```

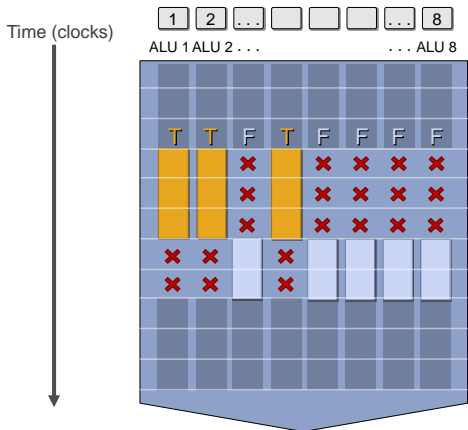
```
    x = 0;
```

```
    refl = Ka;
```

```
}
```

<resume unconditional
shader code>

But what about branches?



<unconditional
shader code>

```
if (x > 0) {
```

```
    y = pow(x, exp);
```

```
    y *= Ks;
```

```
    refl = y + Ka;
```

```
} else {
```

```
    x = 0;
```

```
    refl = Ka;
```

```
}
```

<resume unconditional
shader code>

Clarification

SIMD processing does not imply SIMD instructions

- Option 1: explicit vector instructions
 - x86 SSE, AVX, 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.

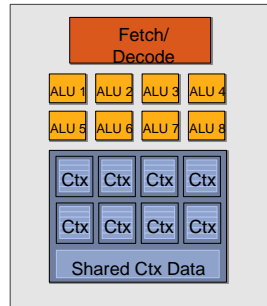
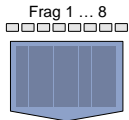
But we have **LOTS** of independent fragments.

Idea #3:

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

Hiding shader stalls

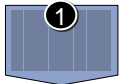
Time (clocks)



Hiding shader stalls

Time (clocks)

Frag 1 ... 8



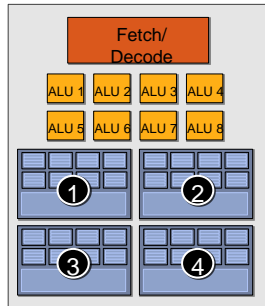
Frag 9 ... 16



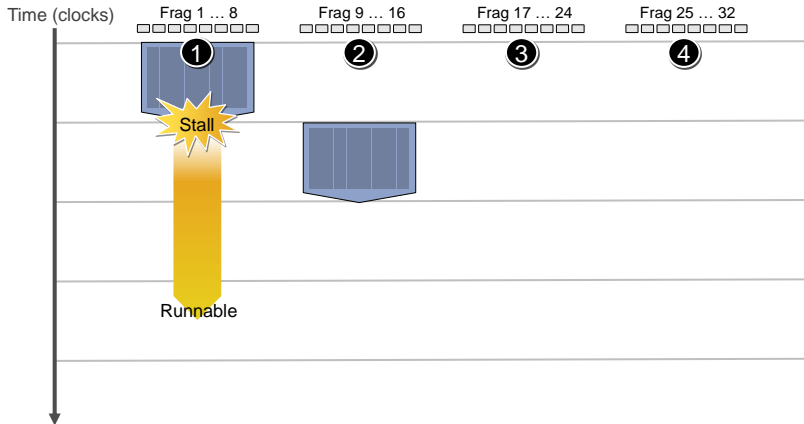
Frag 17 ... 24



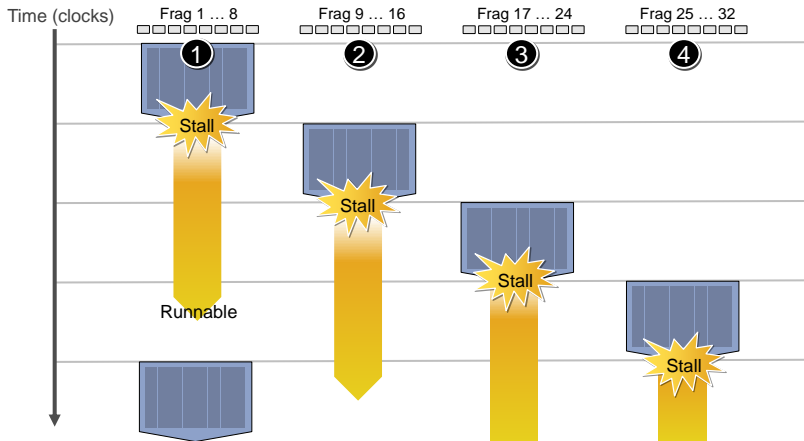
Frag 25 ... 32



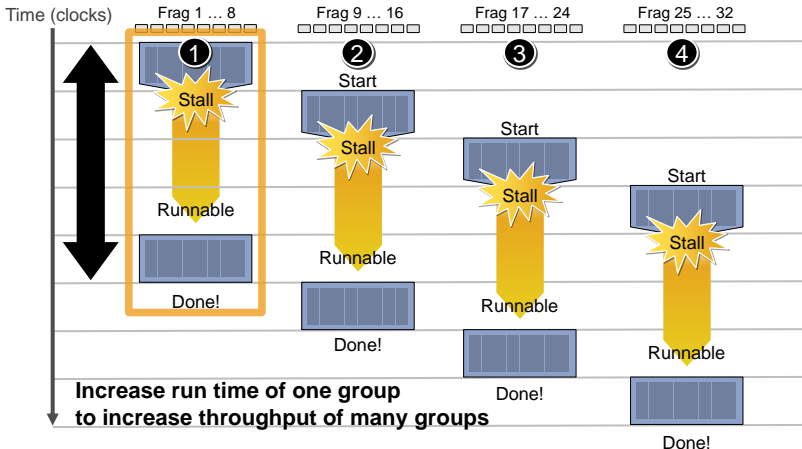
Hiding shader stalls



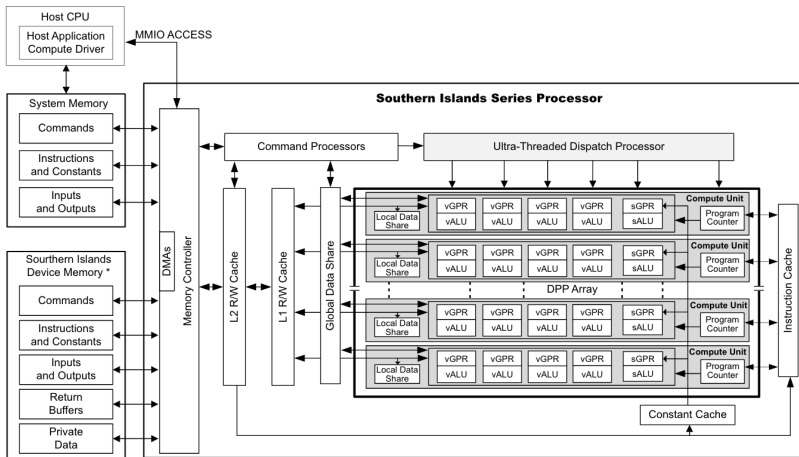
Hiding shader stalls



Throughput!

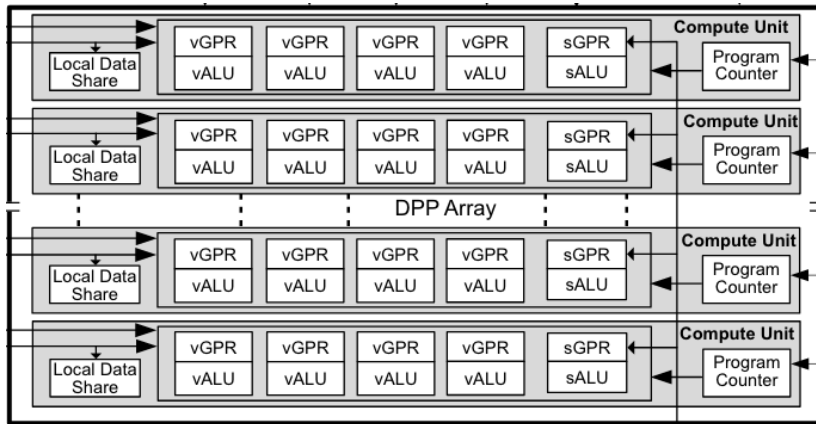


The GPU we will be using: Radeon HD 7800¹



¹https://developer.amd.com/wordpress/media/2012/12/AMD_Southern_Islands_Instruction_Set_Architecture.pdf

Zooming in on the Compute Units



- ▶ Each vector-ALU executes a *wavefront* of 64 work-items over four clock cycles.
- ▶ Many wavefronts in flight at once to hide latency.

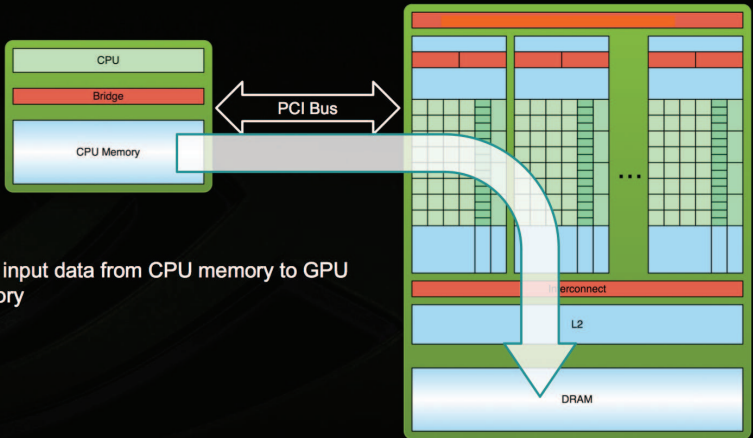
Hardware Trends

The GPU Architecture

The OpenCL Programming Model

GPU programming

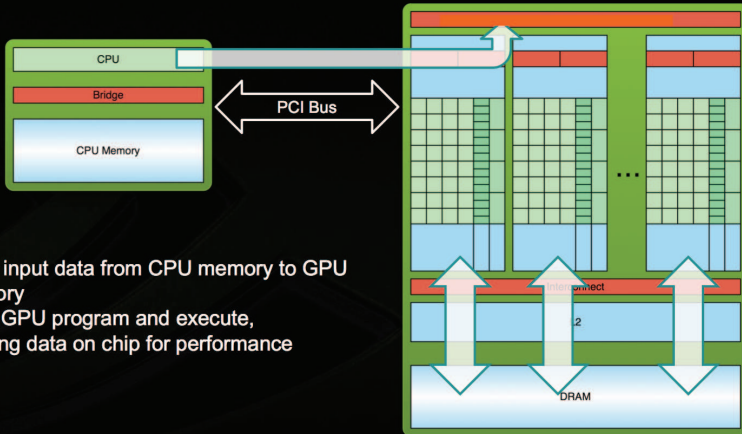
GPU-programming



1. Copy input data from CPU memory to GPU memory

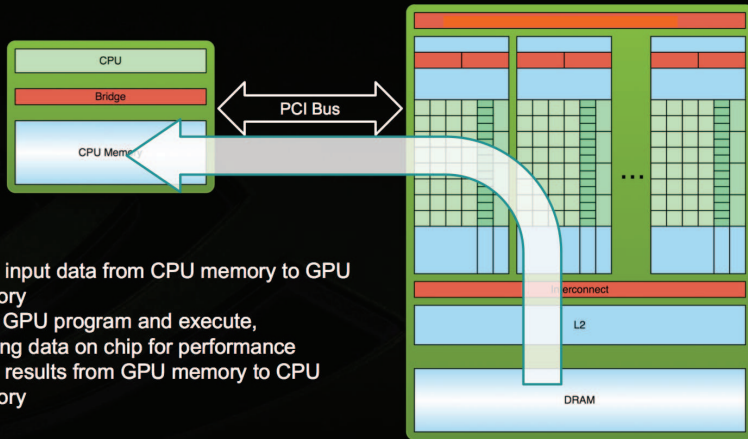
GPU programming

GPU-programming



GPU programming

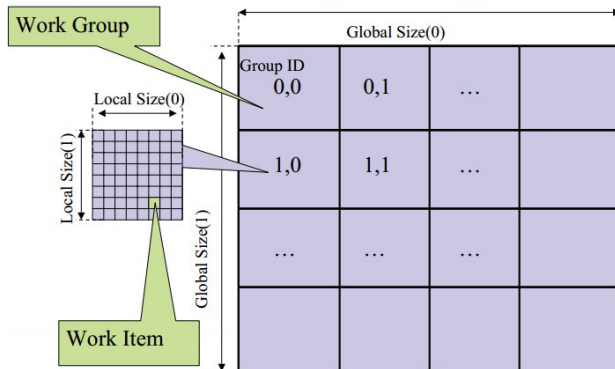
GPU-programming



OpenCL is an SIMT model

Single Instruction Multiple Threads means we provide a *sequential function* that is executed in parallel by multiple threads (“work items” in OpenCL).

OpenCL NDRange Configuration



Threads are arranged in *workgroups*, which form an *NDRange* (often called *grid*).