

CS 380 - GPU and GPGPU Programming

Lecture 6: GPU Architecture, Pt. 3

Markus Hadwiger, KAUST

Reading Assignment #3 (until Sep 18)



Read (required):

- Programming Mass. Parallel Proc. book, 4th ed., Chapter 1 (*Introduction*)
- Programming Mass. Parallel Proc. book, 2nd ed., Chapter 2 (*History of GPU Computing*)
- OpenGL 4 Shading Language Cookbook, Chapter 2
- OpenGL Shading Language 4.6 (current: Jul 10, 2019) specification: Chapter 2
<https://www.khronos.org/registry/OpenGL/specs/gl/GLSLangSpec.4.60.pdf>
- Download OpenGL 4.6 (current: May 5, 2022) specification
<https://www.khronos.org/registry/OpenGL/specs/gl/glspec46.core.pdf>

Read (optional):

- Orange (GLSL) book, Chapter 7 (OpenGL Shading Language API)
- OpenGL 4 Shading Language Cookbook, Chapter 1

Quiz #1: Sep 28



Organization

- First 30 min of lecture
- No material (book, notes, ...) allowed

Content of questions

- Lectures (both actual lectures and slides)
- Reading assignments
- Programming assignments (algorithms, methods)
- Solve short practical examples

Next Lectures



Lecture 7: Sunday, Sep 18

no lecture on Sep 21 !

Lecture 8: Sunday, Sep 25

GPU Architecture: General Architecture



From Shader Code to a **Teraflop**: How Shader Cores Work

Kayvon Fatahalian
Stanford University

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 processing core) designs
 2. Optimize shaders/compute kernels
 3. Establish intuition: what workloads might benefit from the design of these architectures?

Where this is going...



Summary: three key ideas for high-throughput execution

- 1. Use many “slimmed down cores,” run them in parallel**
- 2. Pack cores full of ALUs (by sharing instruction stream overhead 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**

Where this is going...

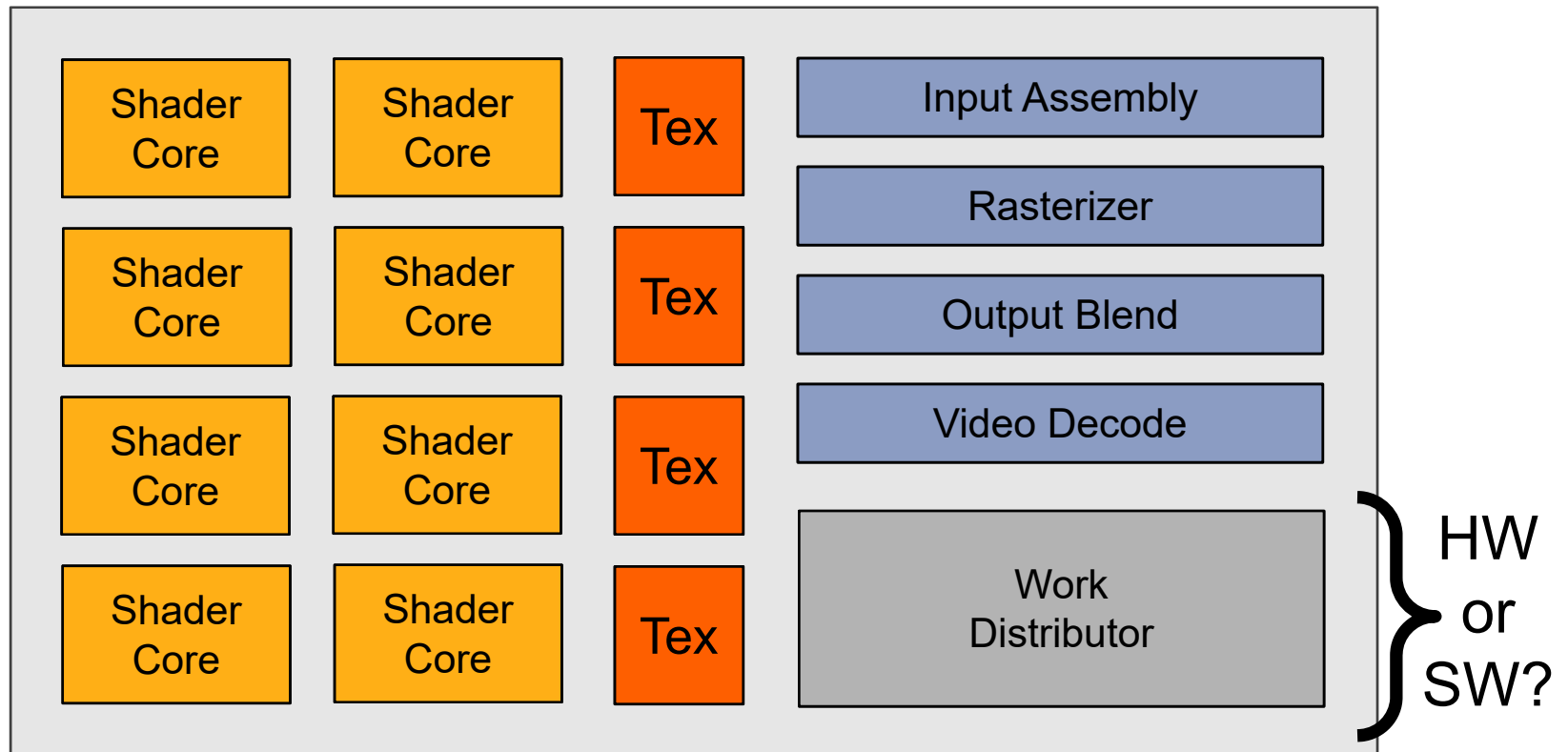


Summary: three key ideas for high-throughput execution

1. Use many “slimmed down cores,” run them in parallel
2. Pack cores full of ALUs (by sharing instruction stream overhead 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

**GPUs are here!
(usually)**

What's in a GPU?



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);
}
```

Independent, but no explicit parallelism

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);
}
```



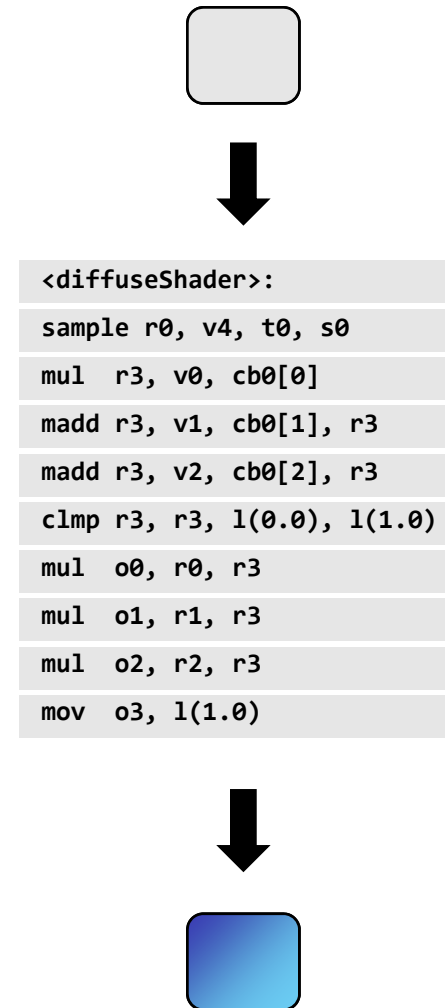
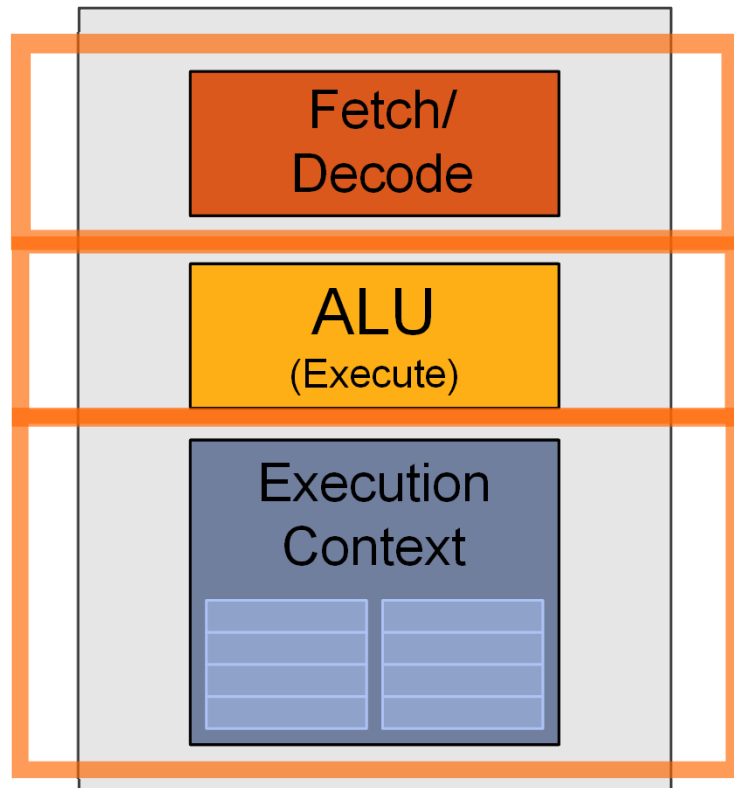
```
<diffuseShader>:
sample r0, v4, t0, s0
mul r3, v0, cb0[0]
madd r3, v1, cb0[1], r3
madd r3, v2, cb0[2], r3
clmp r3, r3, 1(0.0), 1(1.0)
mul o0, r0, r3
mul o1, r1, r3
mul o2, r2, r3
mov o3, 1(1.0)
```



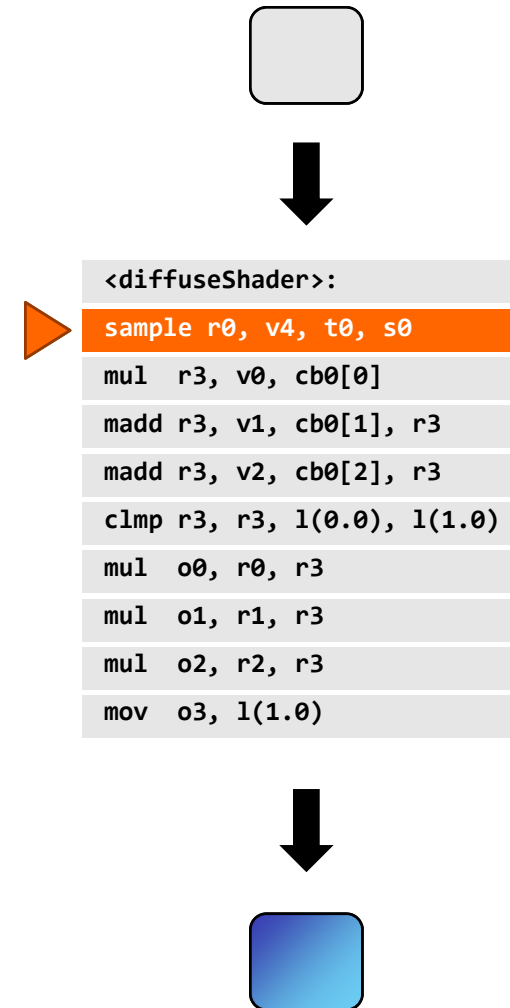
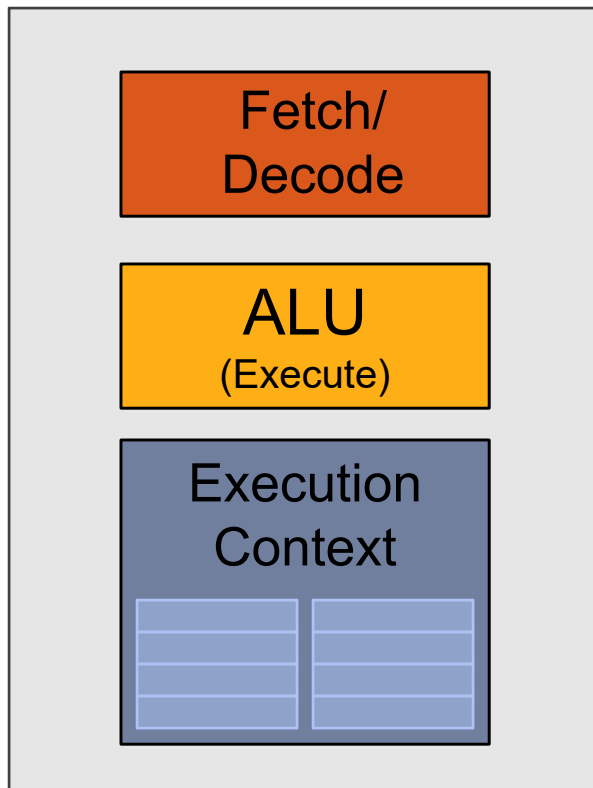
1 shaded fragment output record



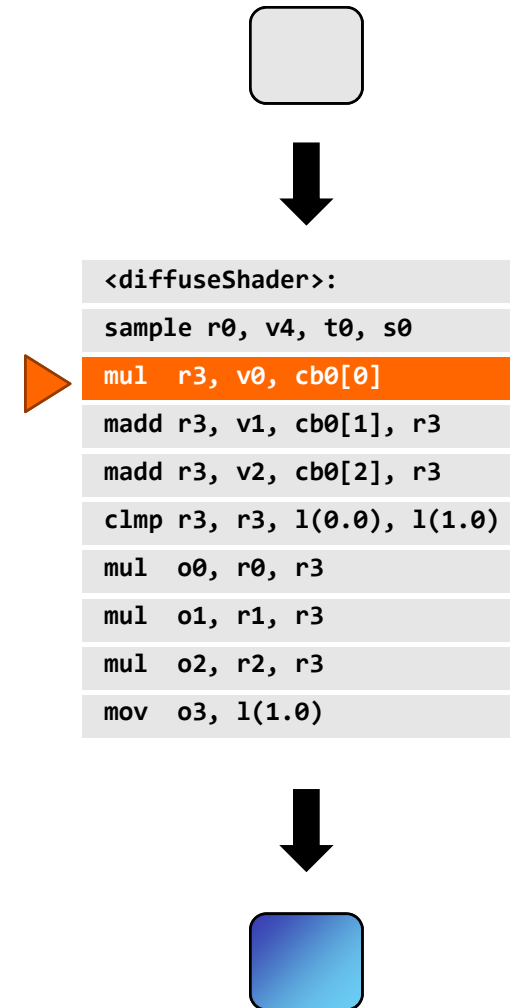
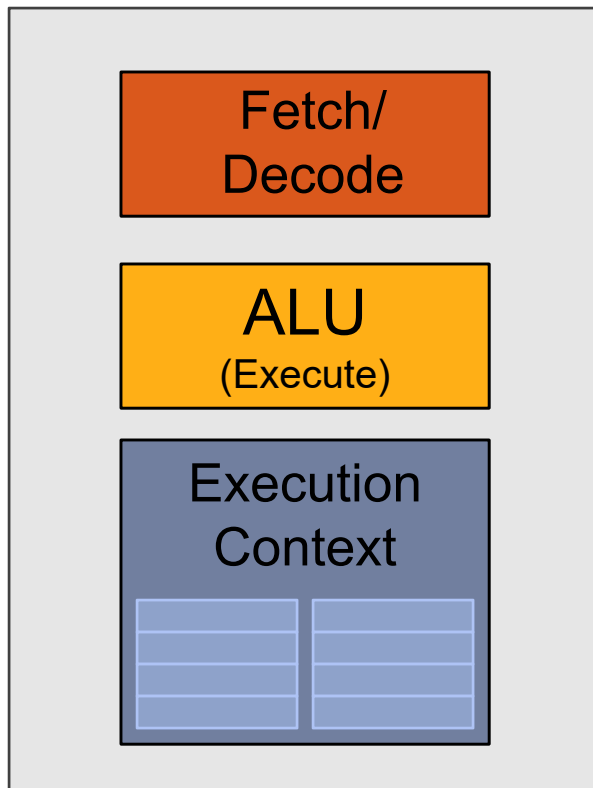
Execute shader



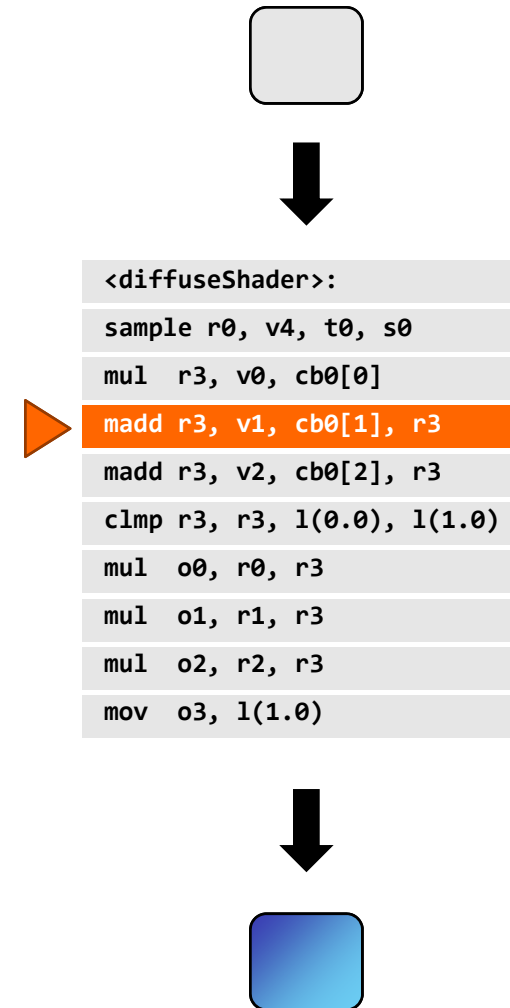
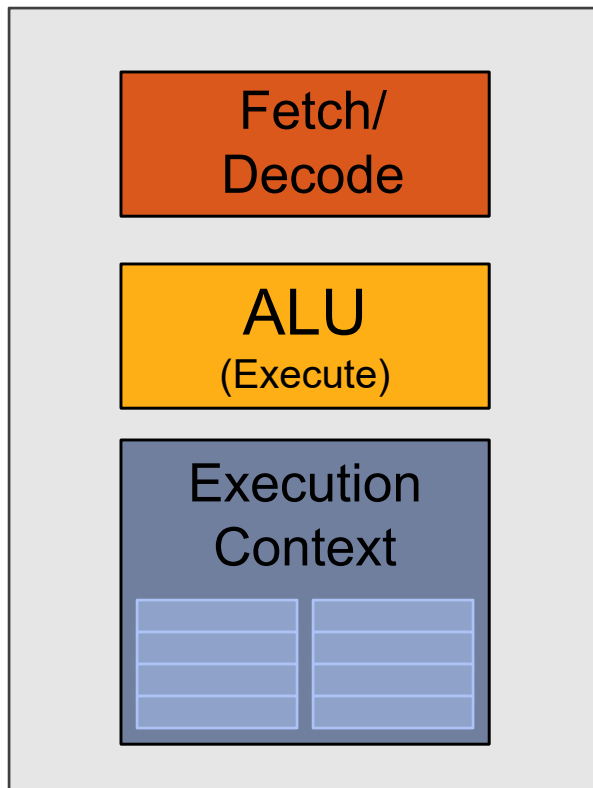
Execute shader



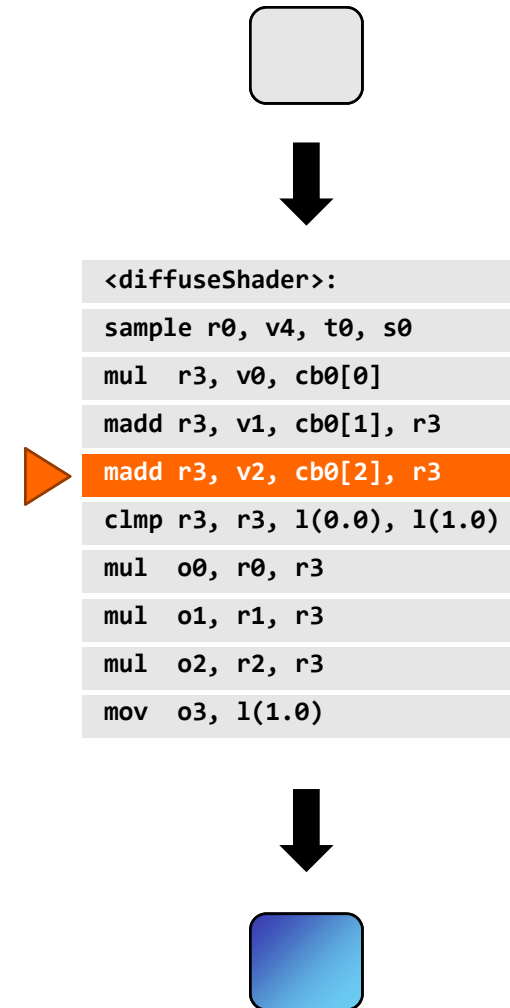
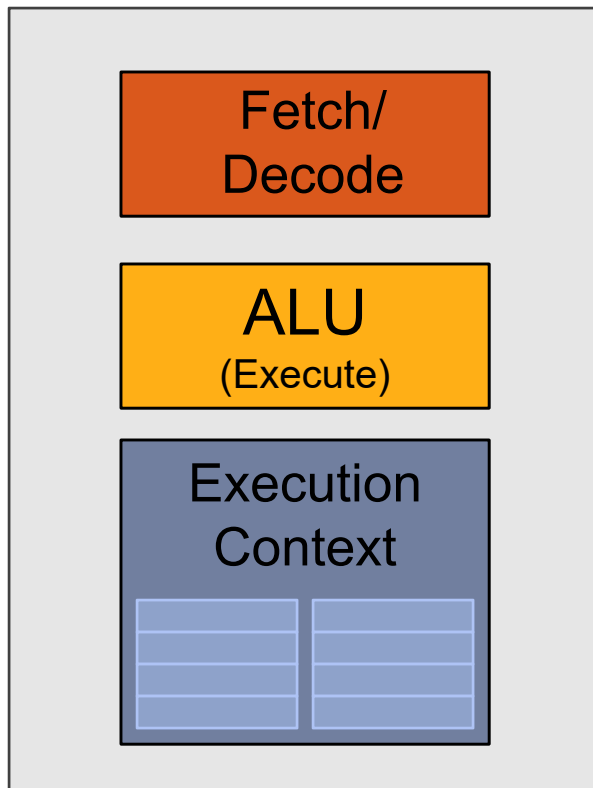
Execute shader



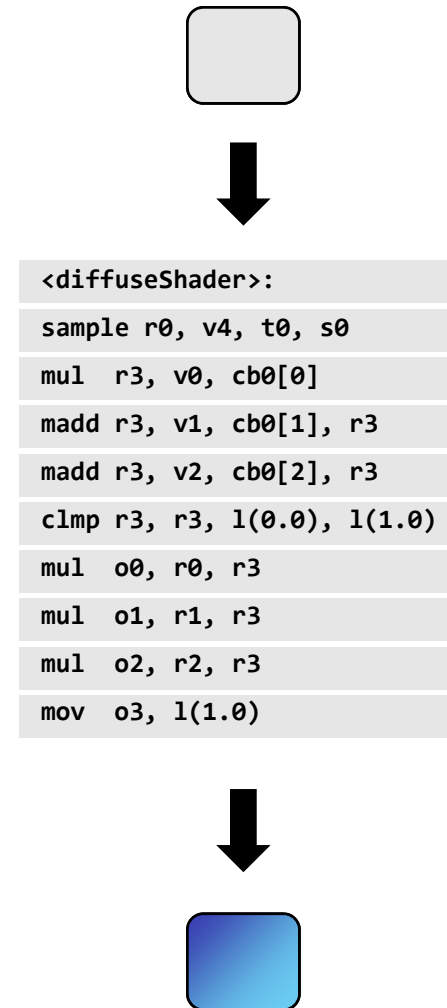
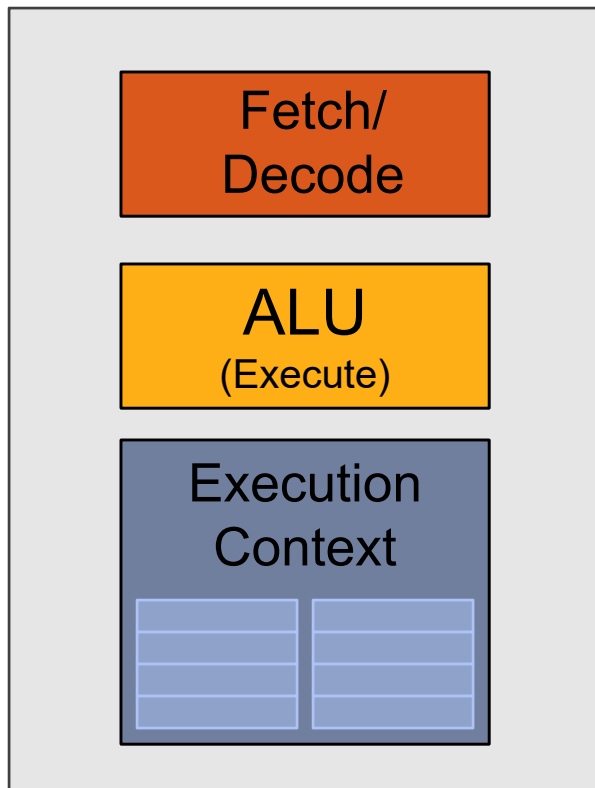
Execute shader



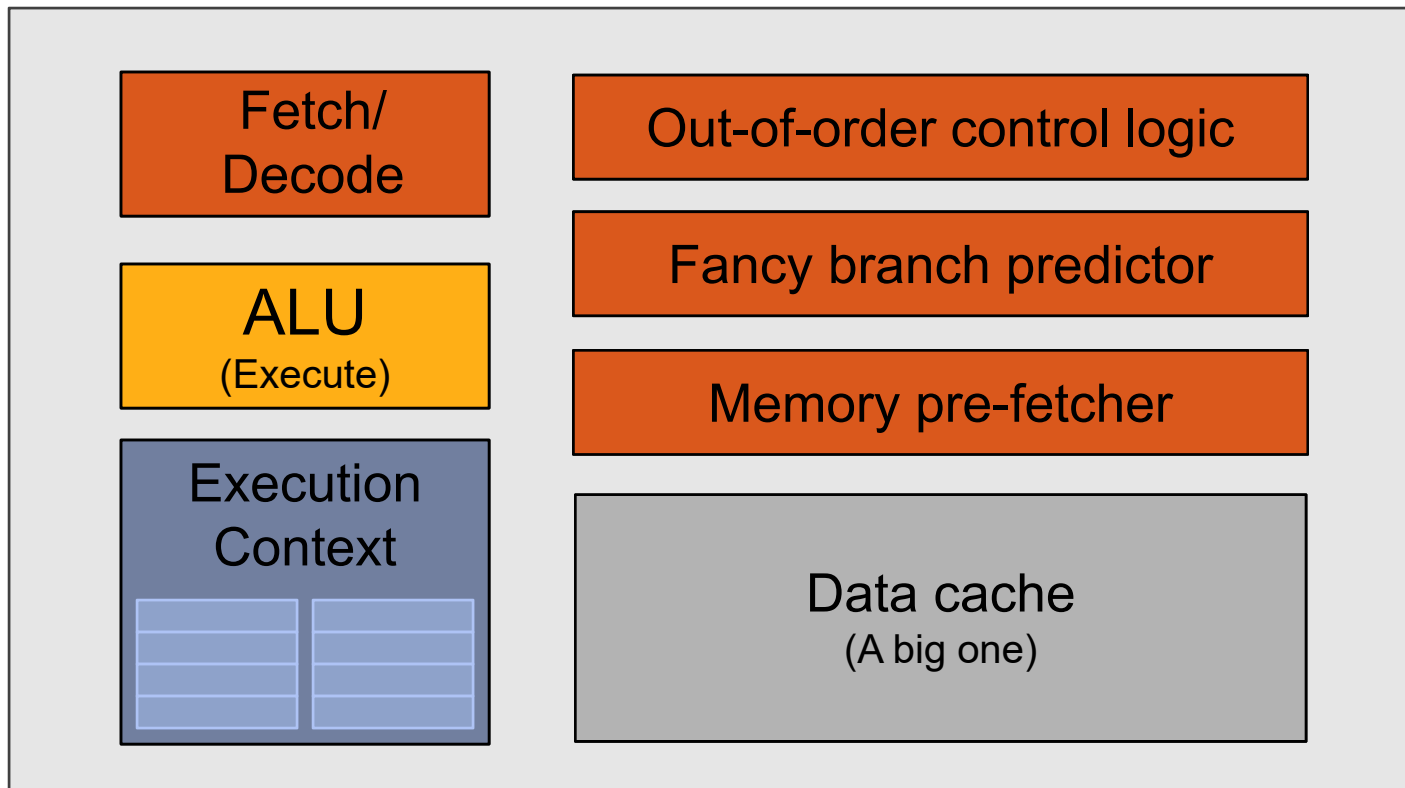
Execute shader



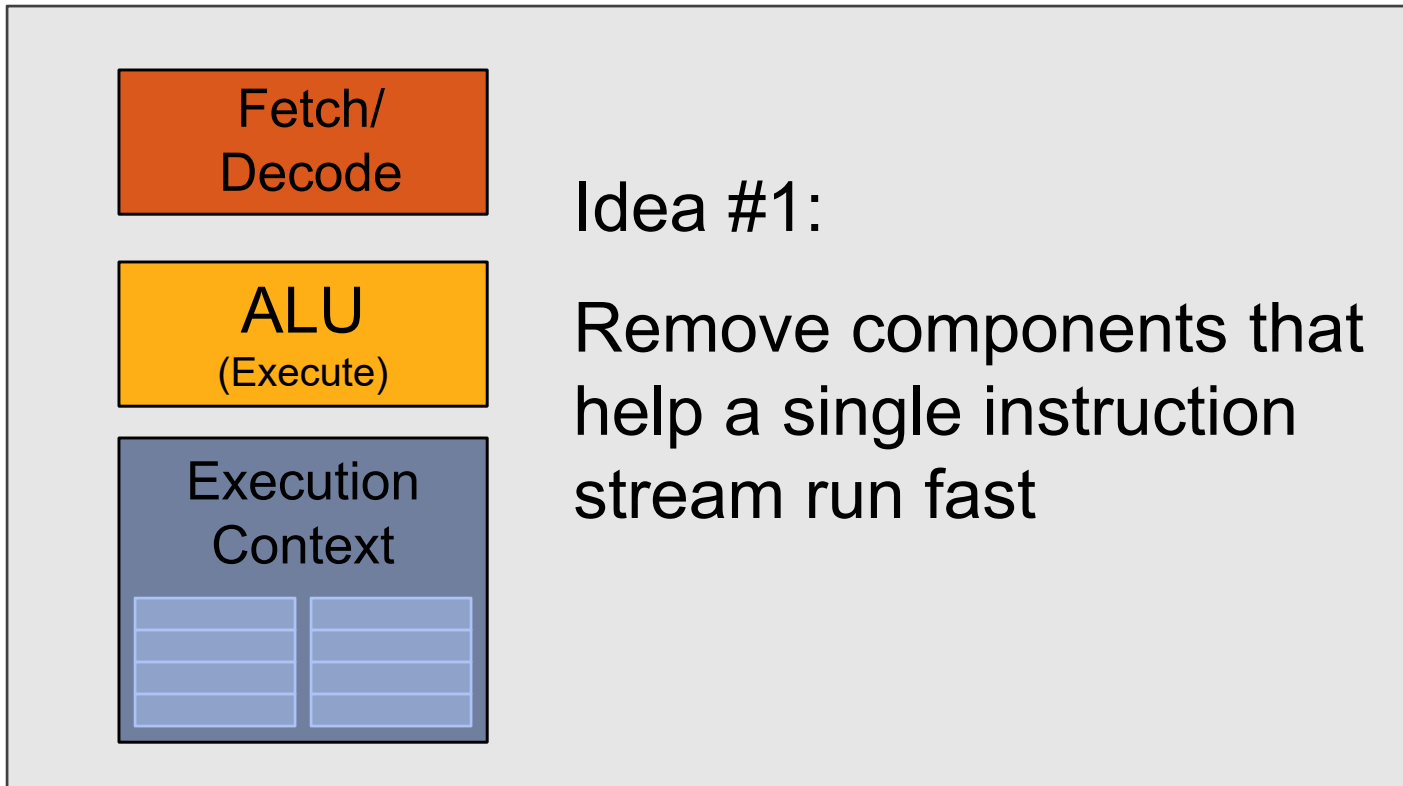
Execute shader



CPU-“style” cores

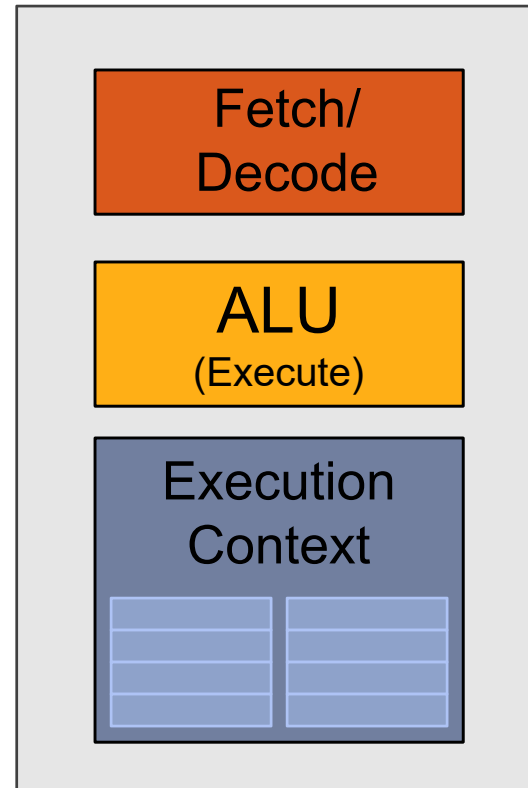
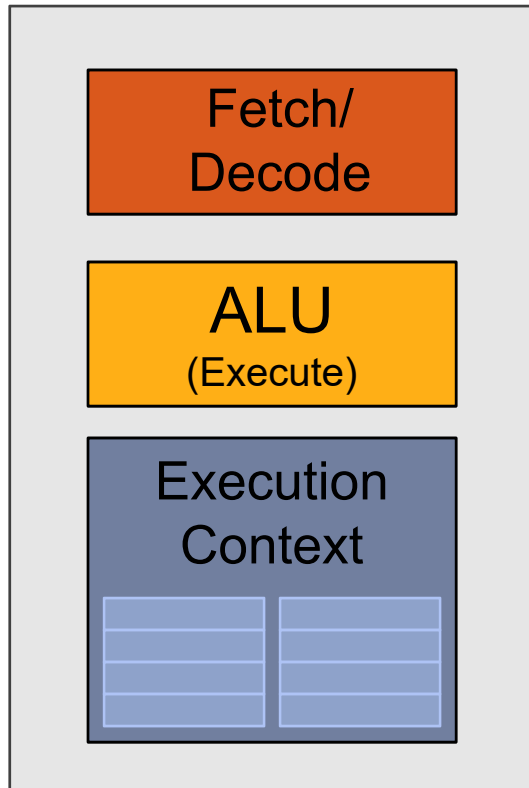
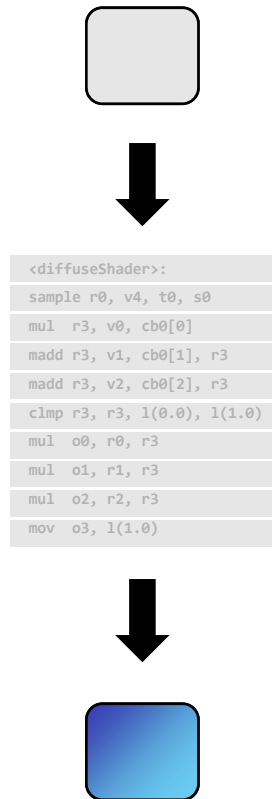


Idea #1: Slim down

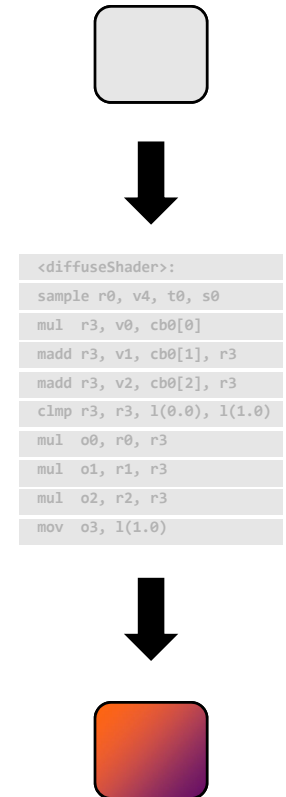


Two cores (two fragments in parallel)

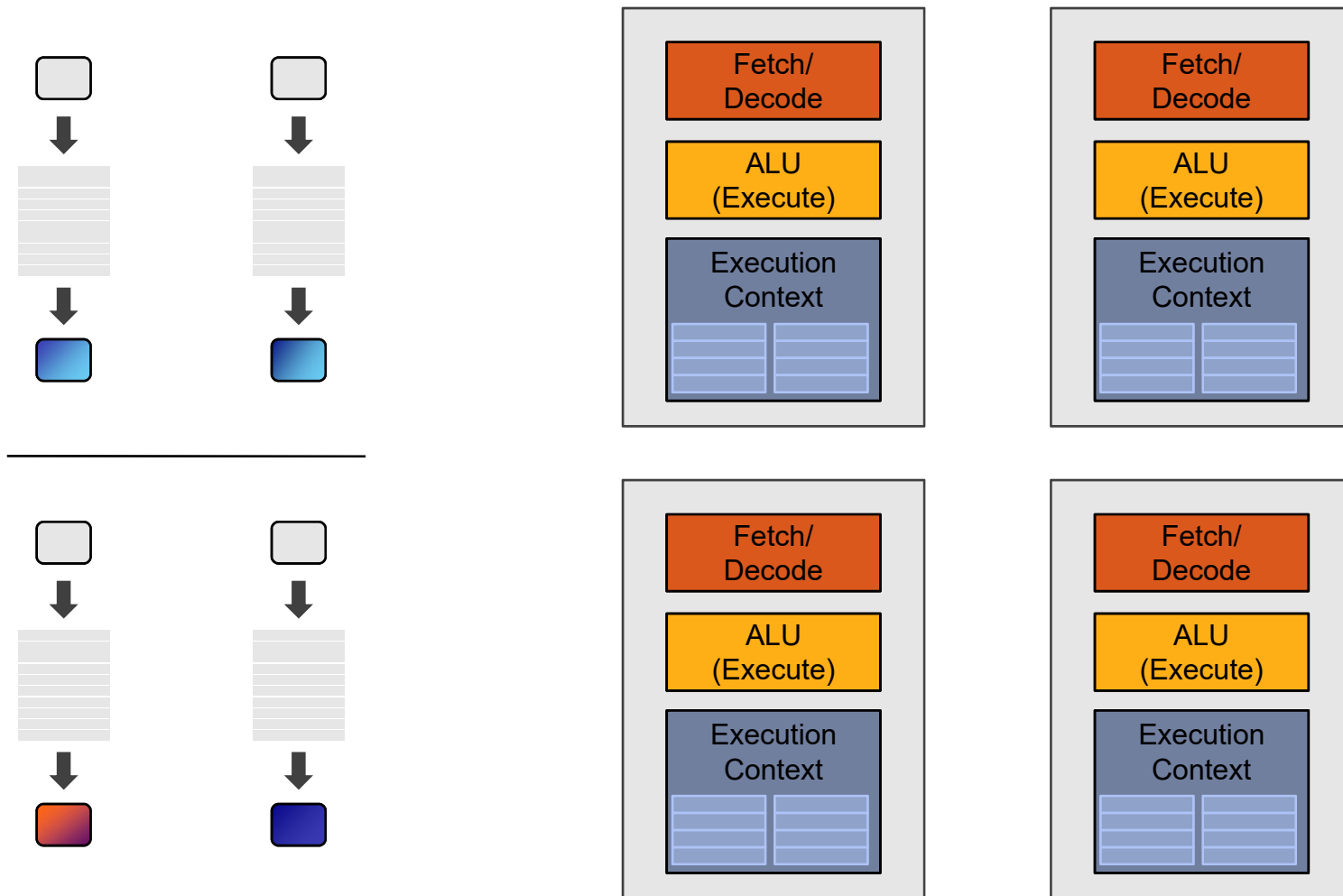
fragment 1



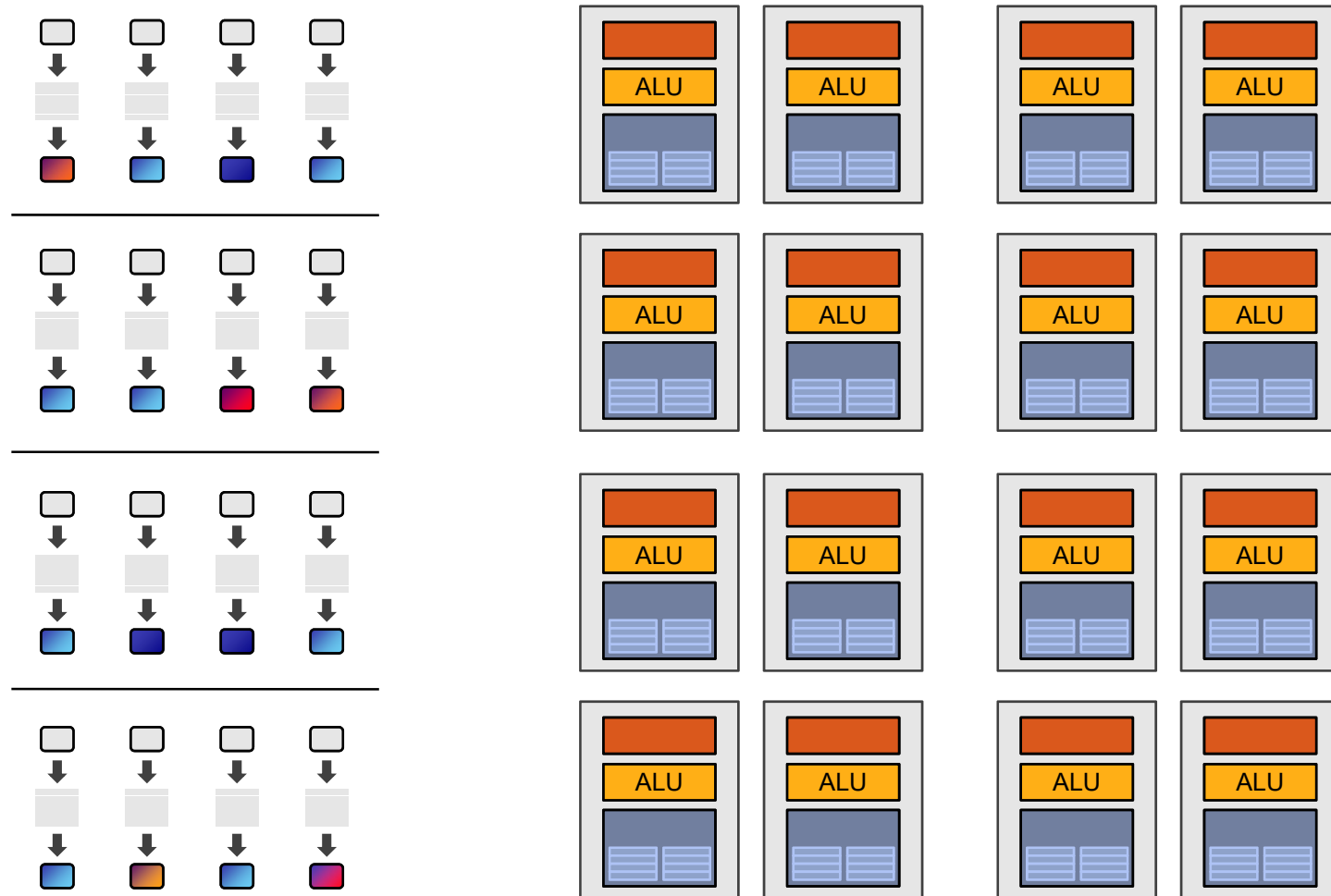
fragment 2



Four cores (four fragments in parallel)

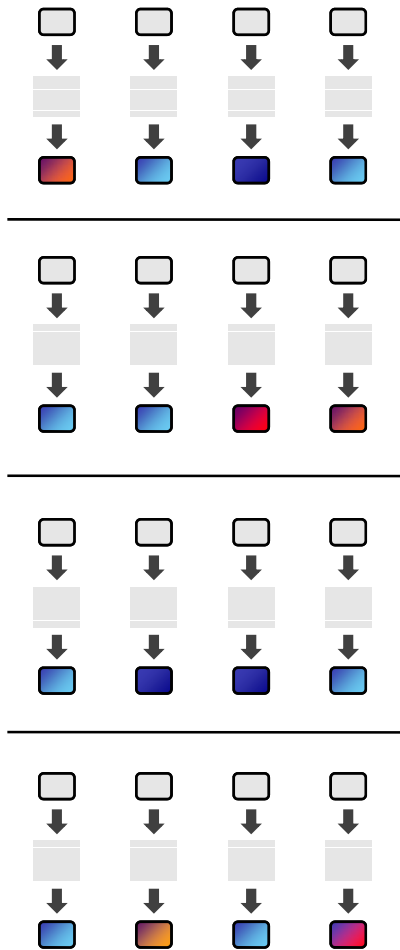


Sixteen cores (sixteen fragments in parallel)



16 cores = 16 simultaneous instruction streams

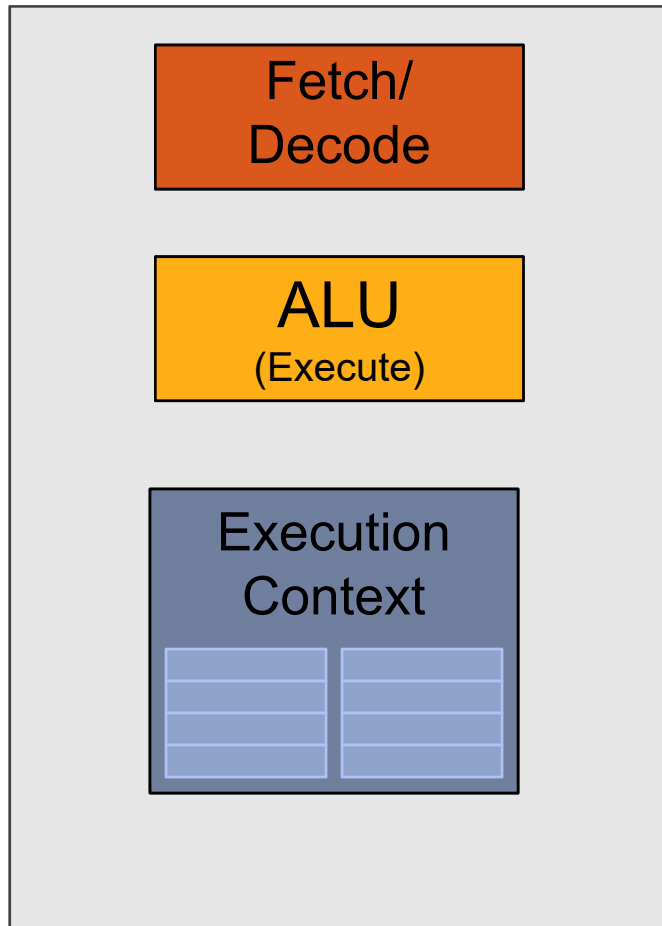
Instruction stream sharing



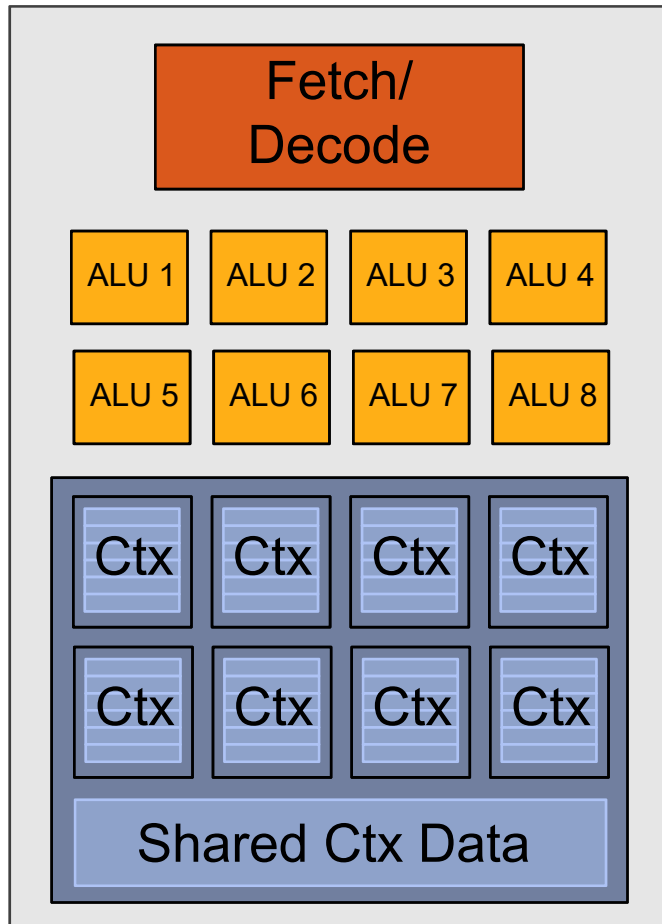
But... many fragments should be able to share an instruction stream!

```
<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



Idea #2: Add ALUs



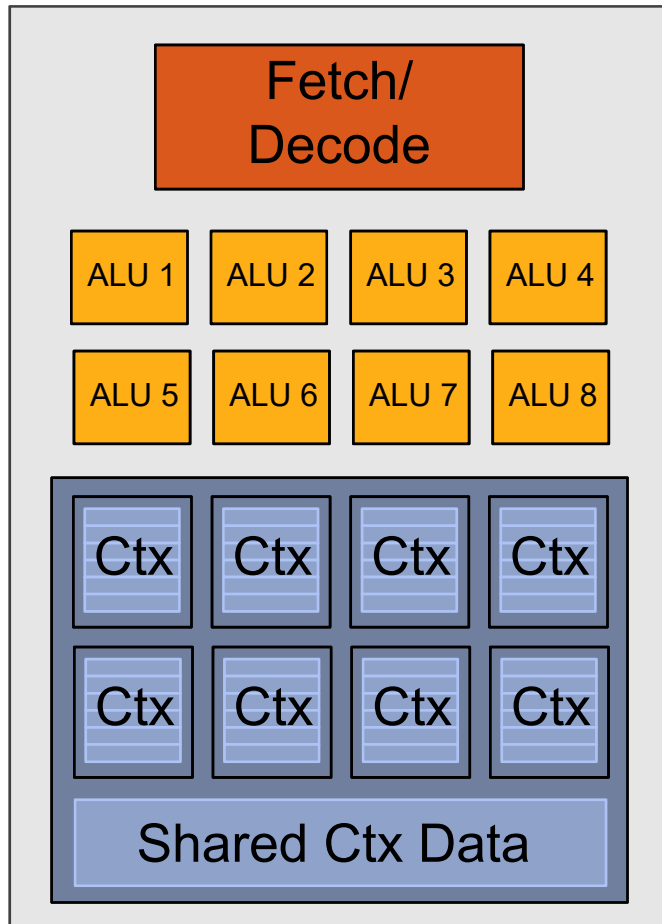
Idea #2:

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

SIMD processing

(or **SIMT**, SPMD)

How does shader execution behave?

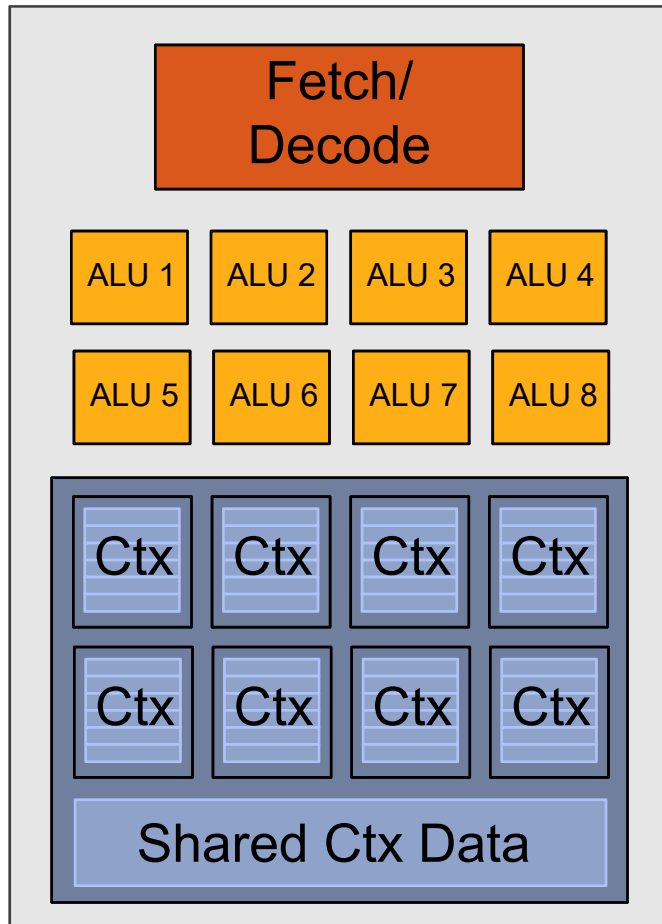


```
<diffuseShader>:  
sample r0, v4, t0, s0  
mul r3, v0, cb0[0]  
madd r3, v1, cb0[1], r3  
madd r3, v2, cb0[2], r3  
clamp r3, r3, l(0.0), l(1.0)  
mul o0, r0, r3  
mul o1, r1, r3  
mul o2, r2, r3  
mov o3, l(1.0)
```

Original compiled shader:

Processes one fragment
using scalar ops on scalar registers

How does shader execution behave?



```
<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_cImp 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 vec_o3, 1(1.0)
```

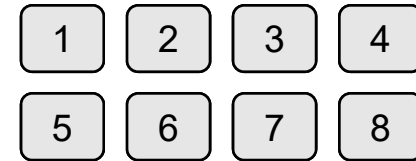
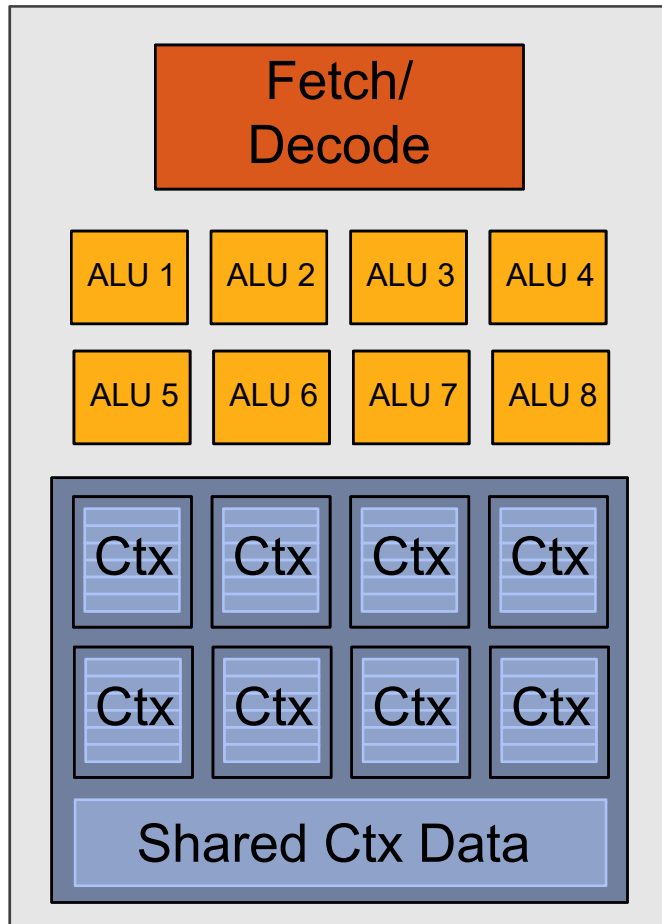
Actually executed shader:

Processes 8 fragments

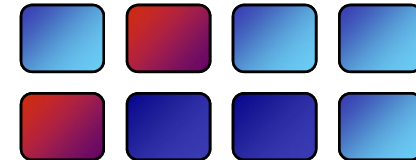
using "vector ops" on "vector registers"

(Caveat: This does NOT mean there are actual vector instructions/cores/regs! See later slide.)

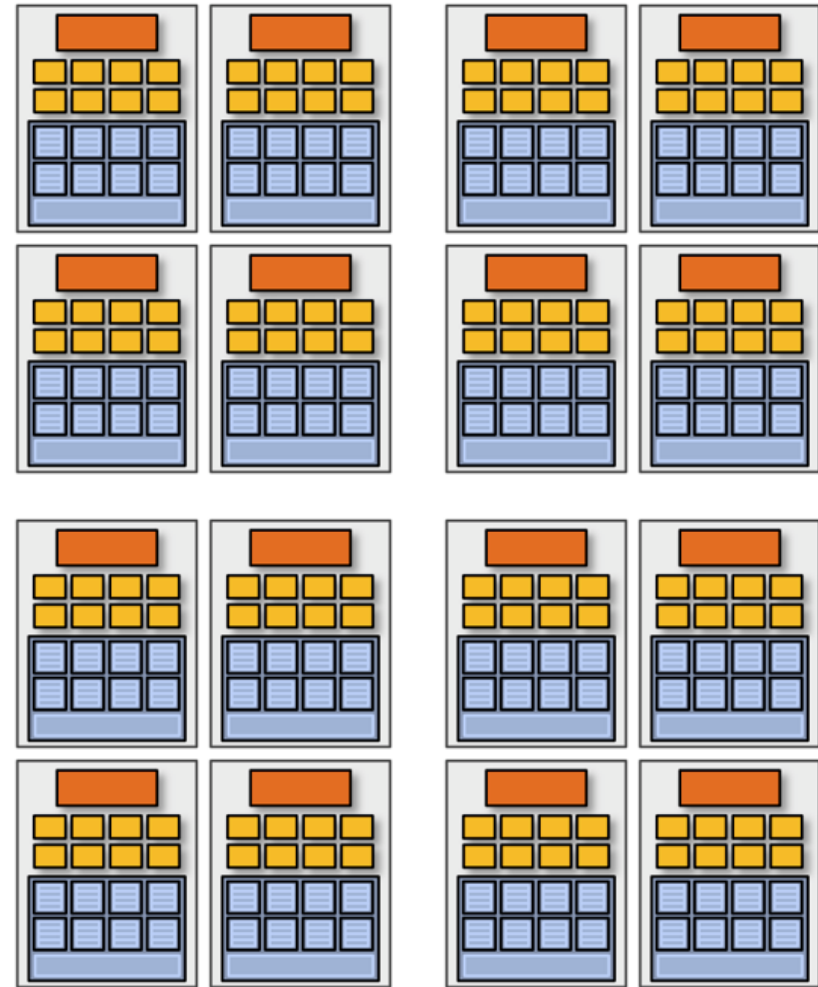
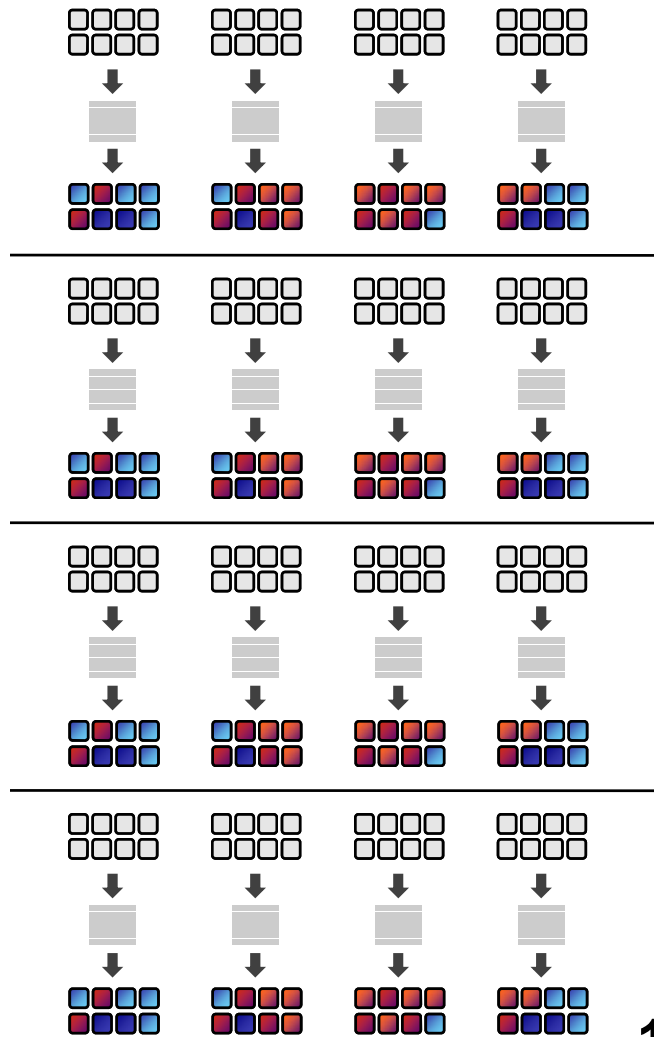
How does shader execution behave?



```
<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 vec_o3, 1(1.0)
```



128 fragments in parallel

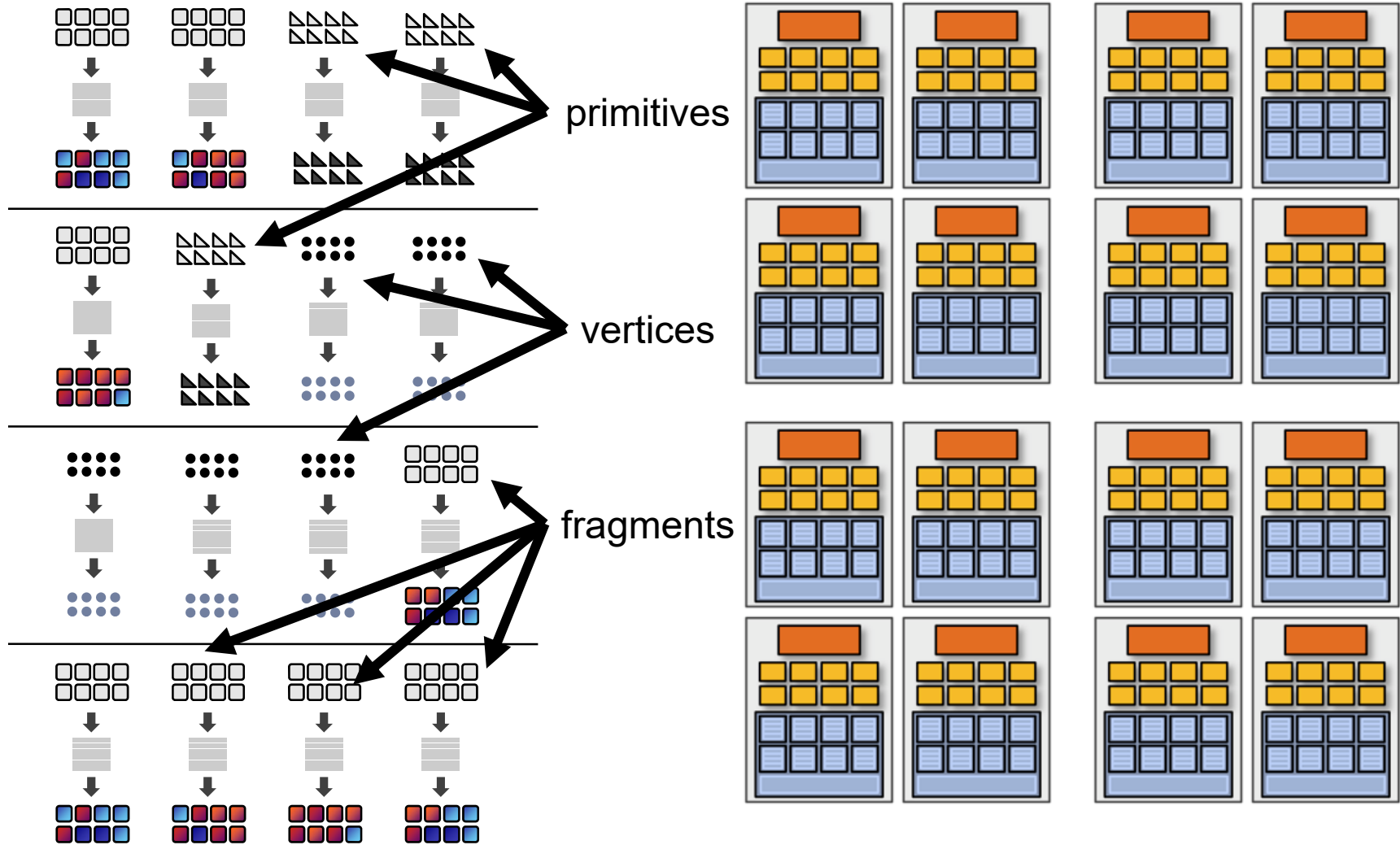


16 cores = **128** ALUs
= **16** simultaneous instruction streams

128 [

vertices / fragments
primitives
CUDA threads
OpenCL work items
compute shader threads

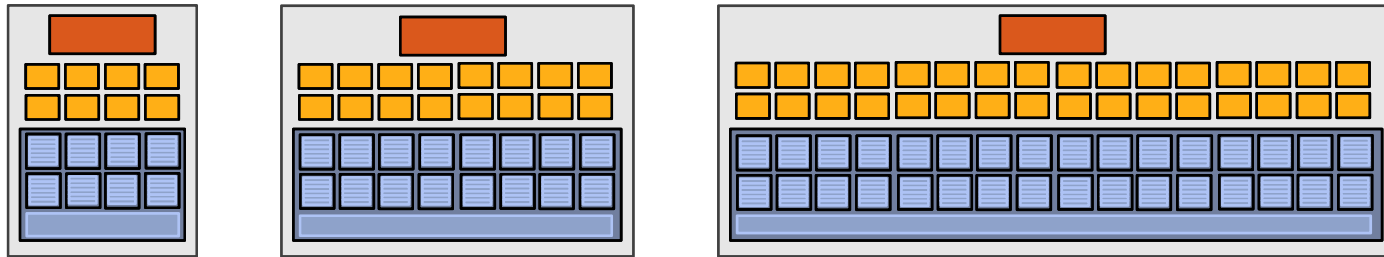
] in parallel



Clarification

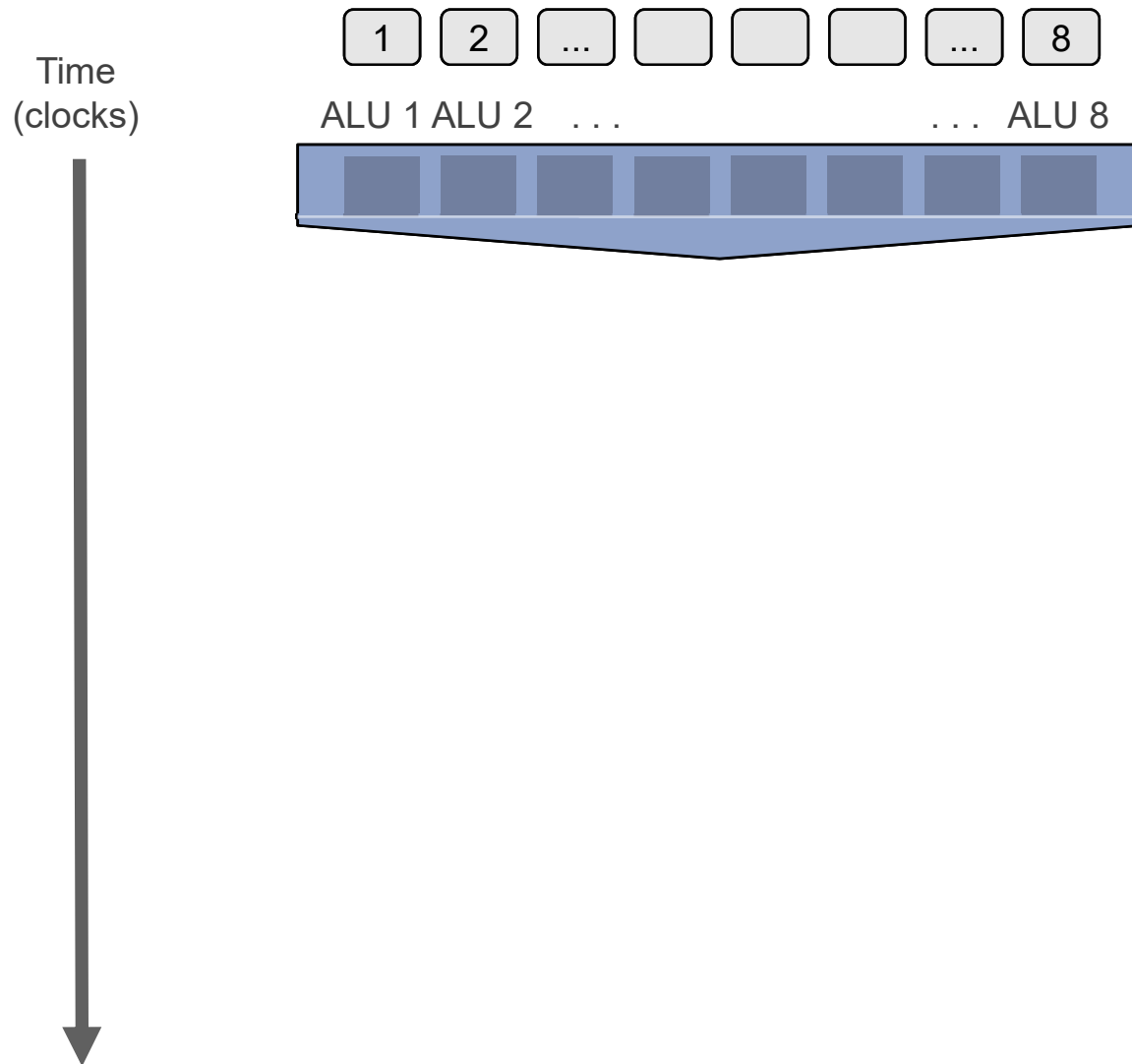
SIMD processing does not imply SIMD instructions

- Option 1: Explicit vector instructions
 - Intel/AMD x86 MMX/SSE/AVX(2), Intel Larrabee/Xeon Phi/ ...
- Option 2: Scalar instructions, implicit HW vectorization
 - HW determines instruction stream sharing across ALUs (amount of sharing hidden from software, i.e., not in ISA)
 - NVIDIA GeForce (“SIMT” warps), AMD Radeon/GNC/RDNA(2)



In practice: 16 to 64 fragments share an instruction stream

But what about branches?

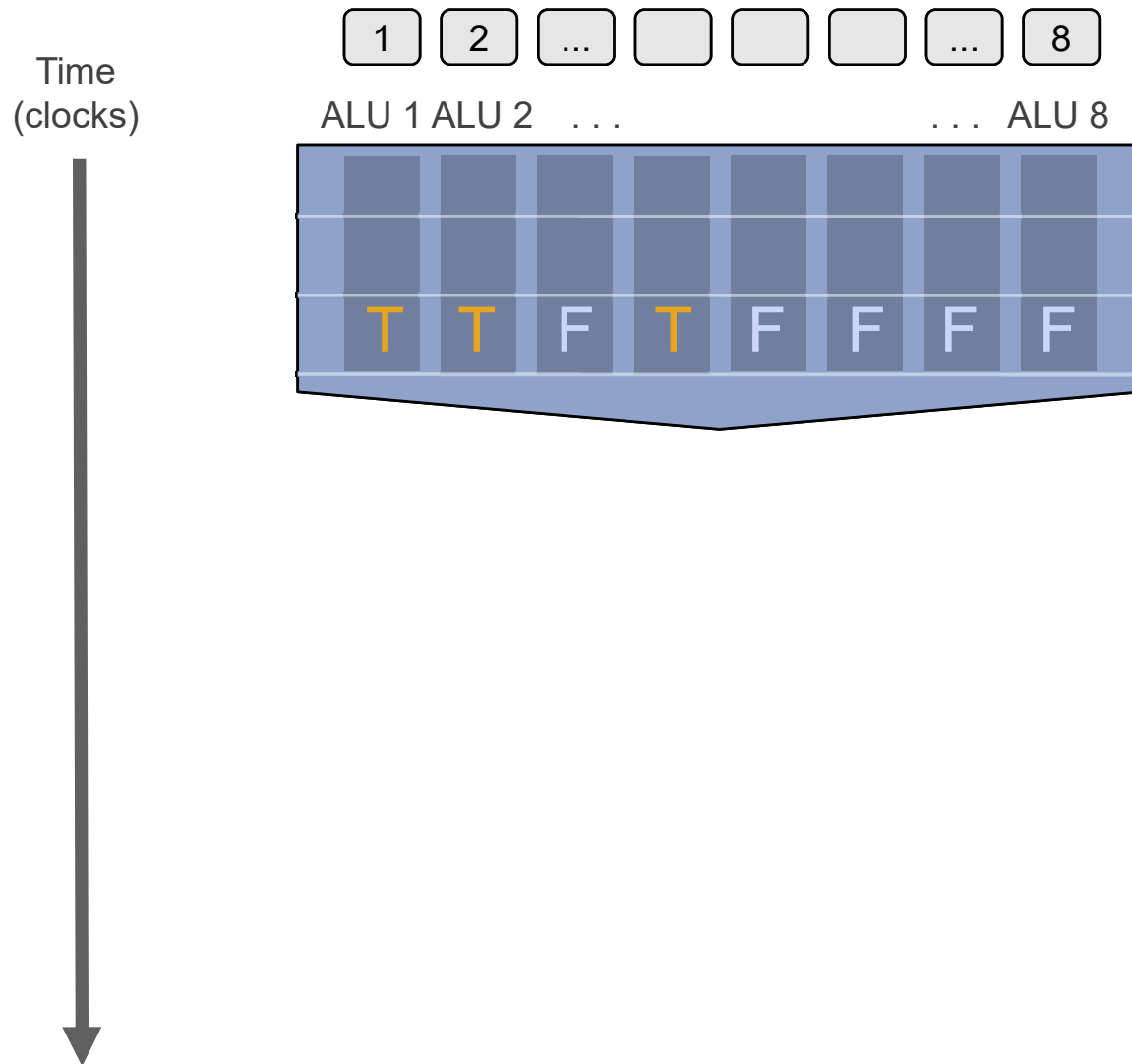


```
<unconditional  
shader code>
```

```
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?

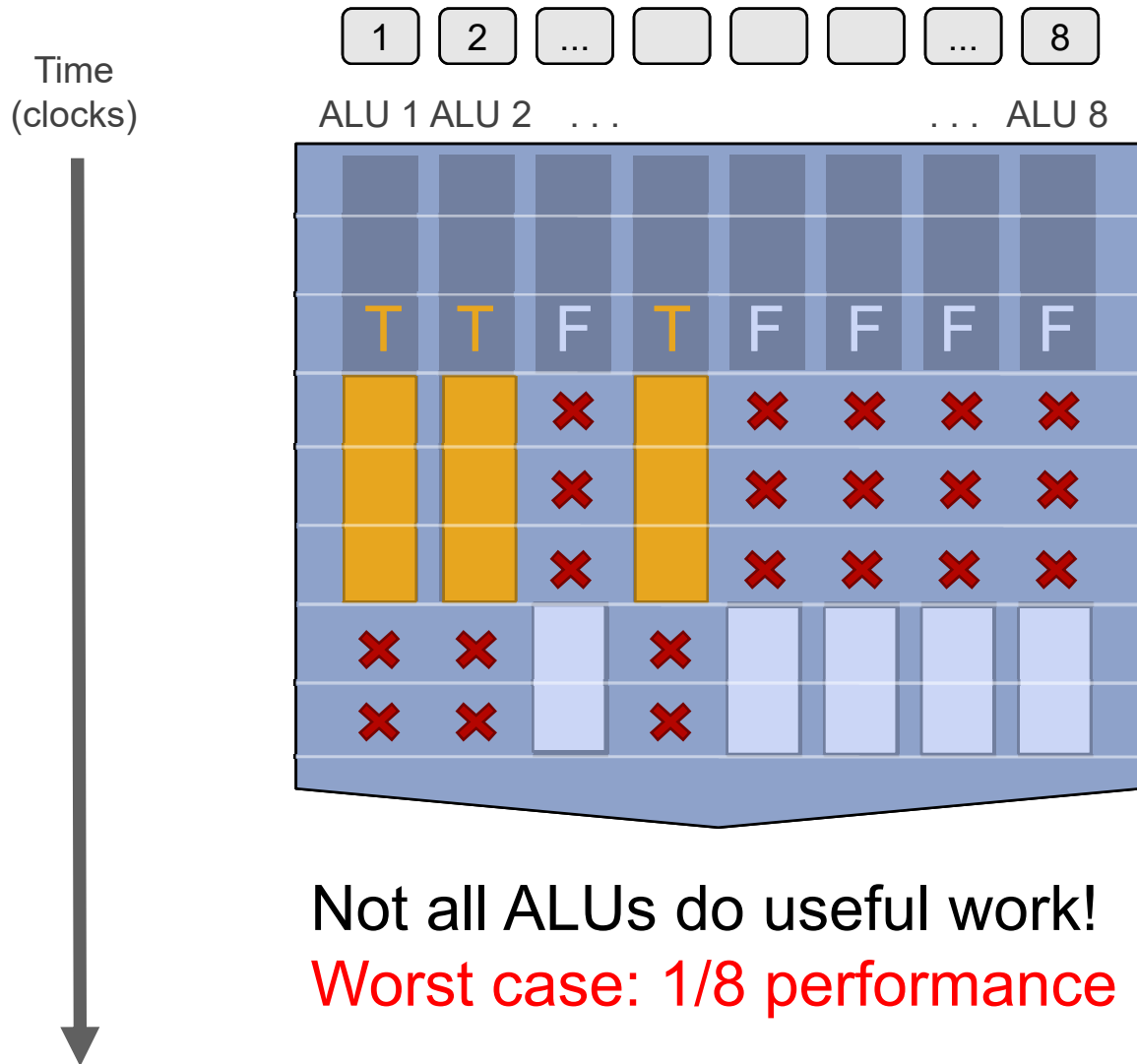


<unconditional
shader code>

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

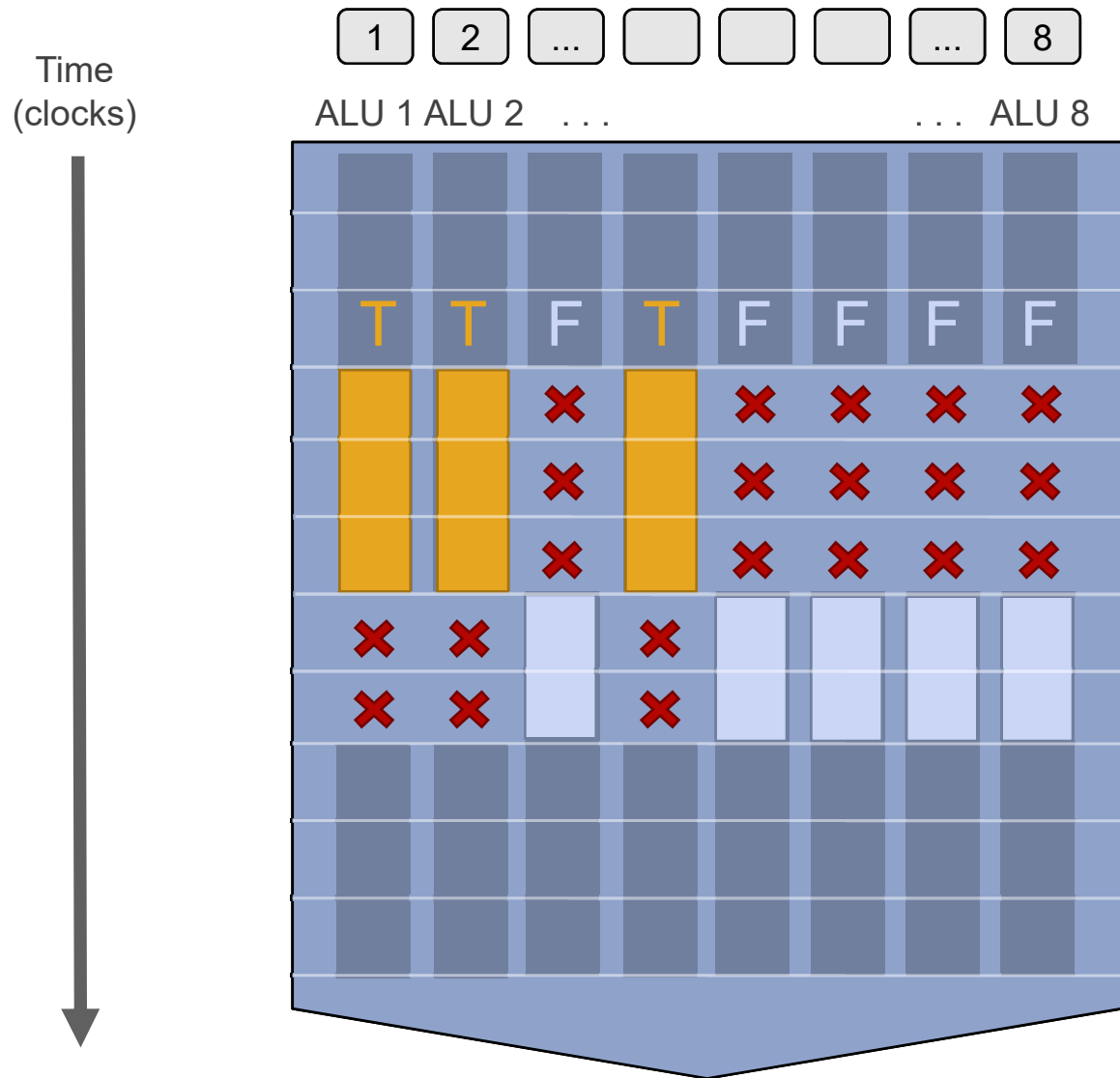
<resume unconditional
shader code>

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>
    
```

Next Problem: 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
(also: instruction pipelining hazards, ...)

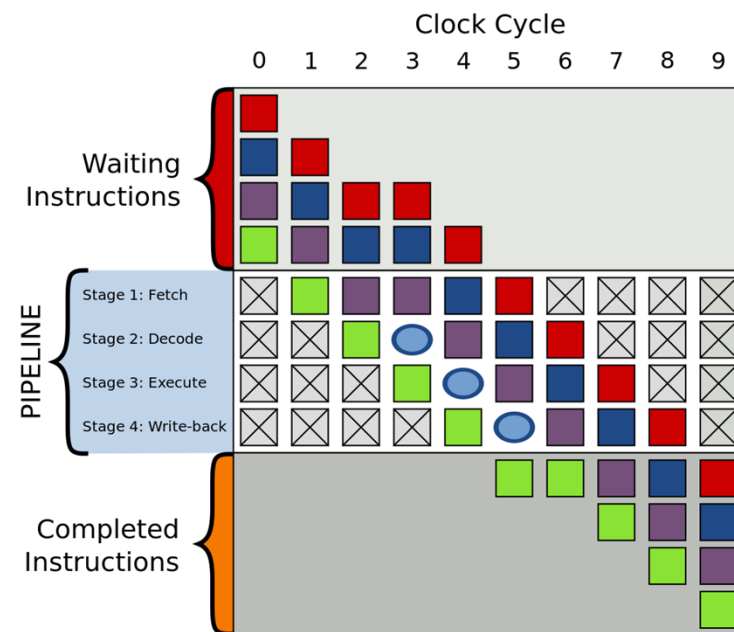
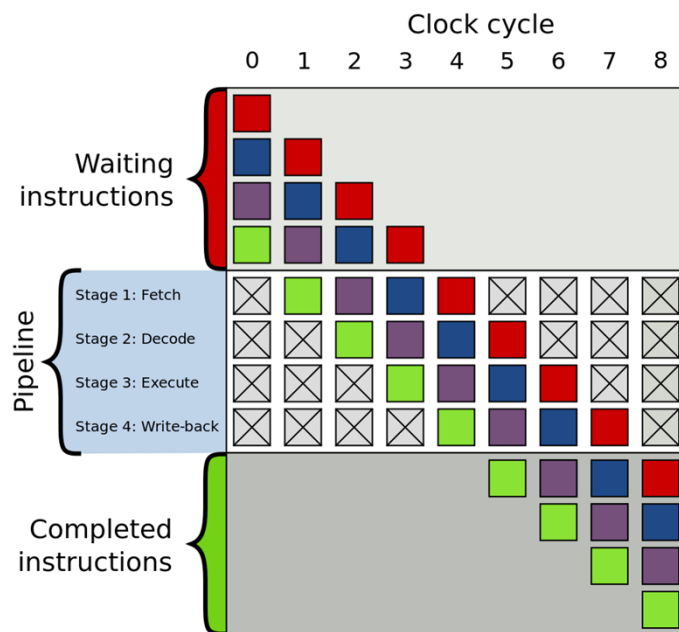
We've removed the fancy caches and logic that helps avoid stalls.

Interlude: Instruction Pipelining



Most basic way to exploit instruction-level parallelism (ILP)

Problem: hazards (different solutions: bubbles, ...)



https://en.wikipedia.org/wiki/Instruction_pipelining
https://en.wikipedia.org/wiki/Classic_RISC_pipeline

wikipedia

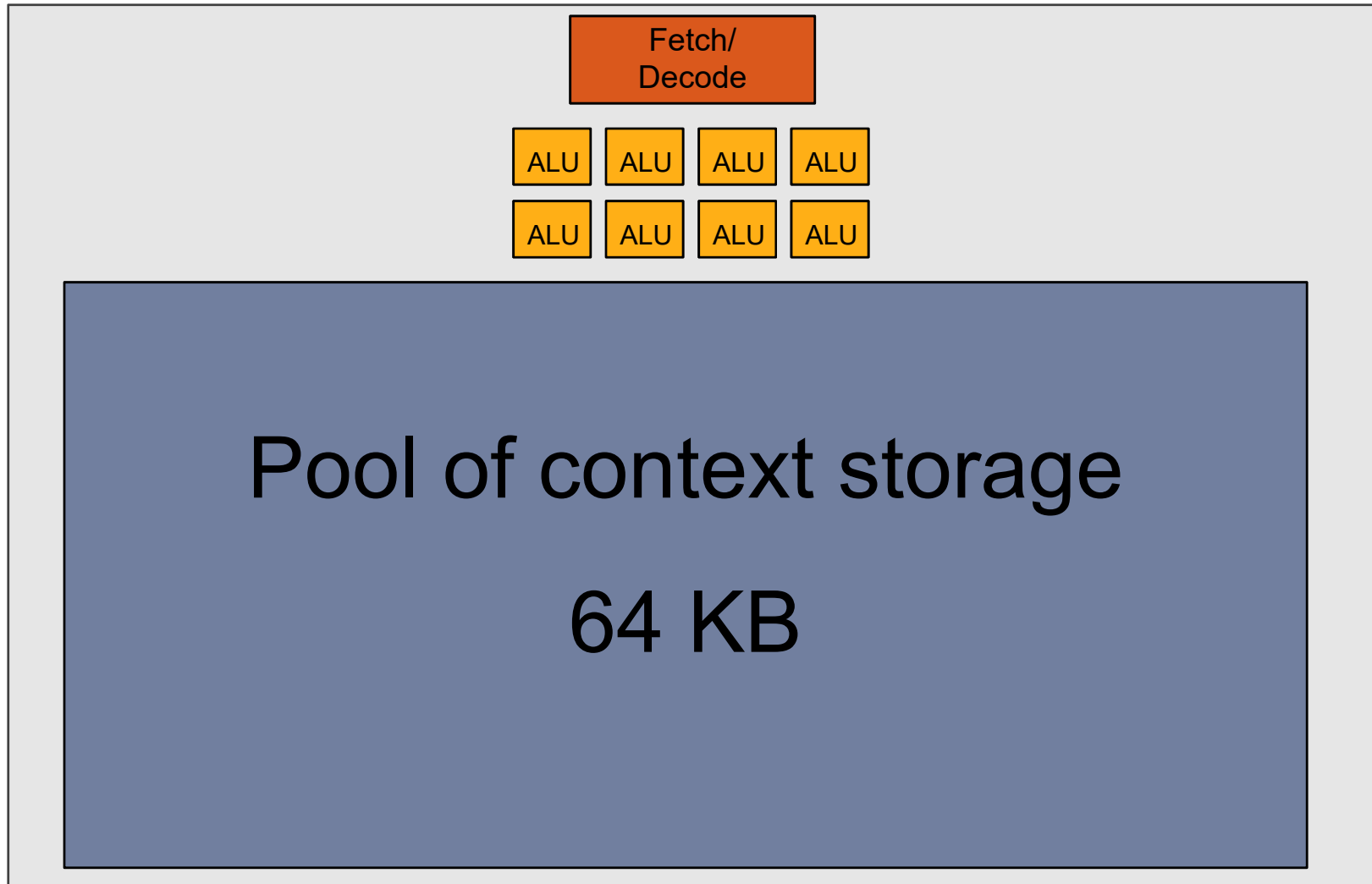
Idea #3: Interleave execution of groups

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.

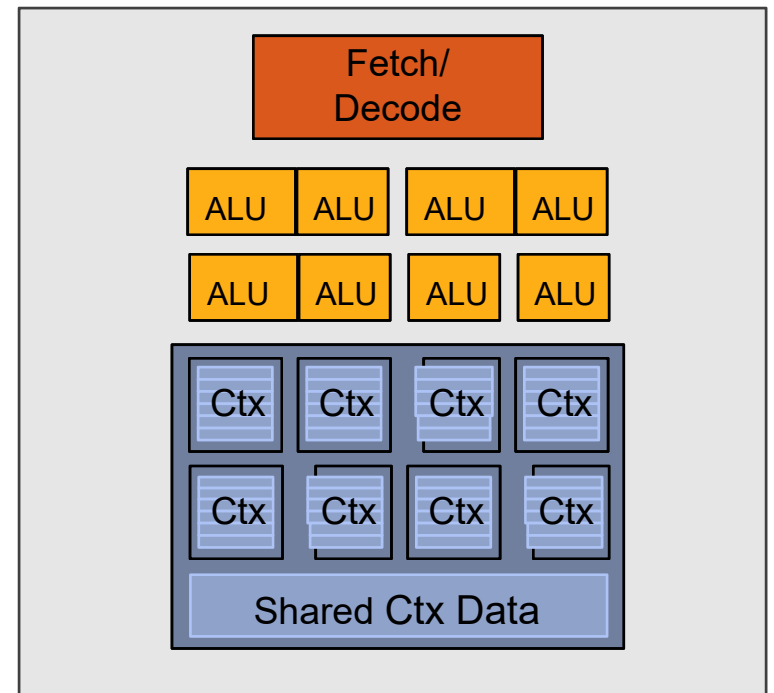
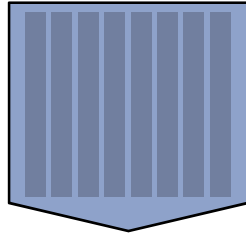
Idea #3: Store multiple group contexts



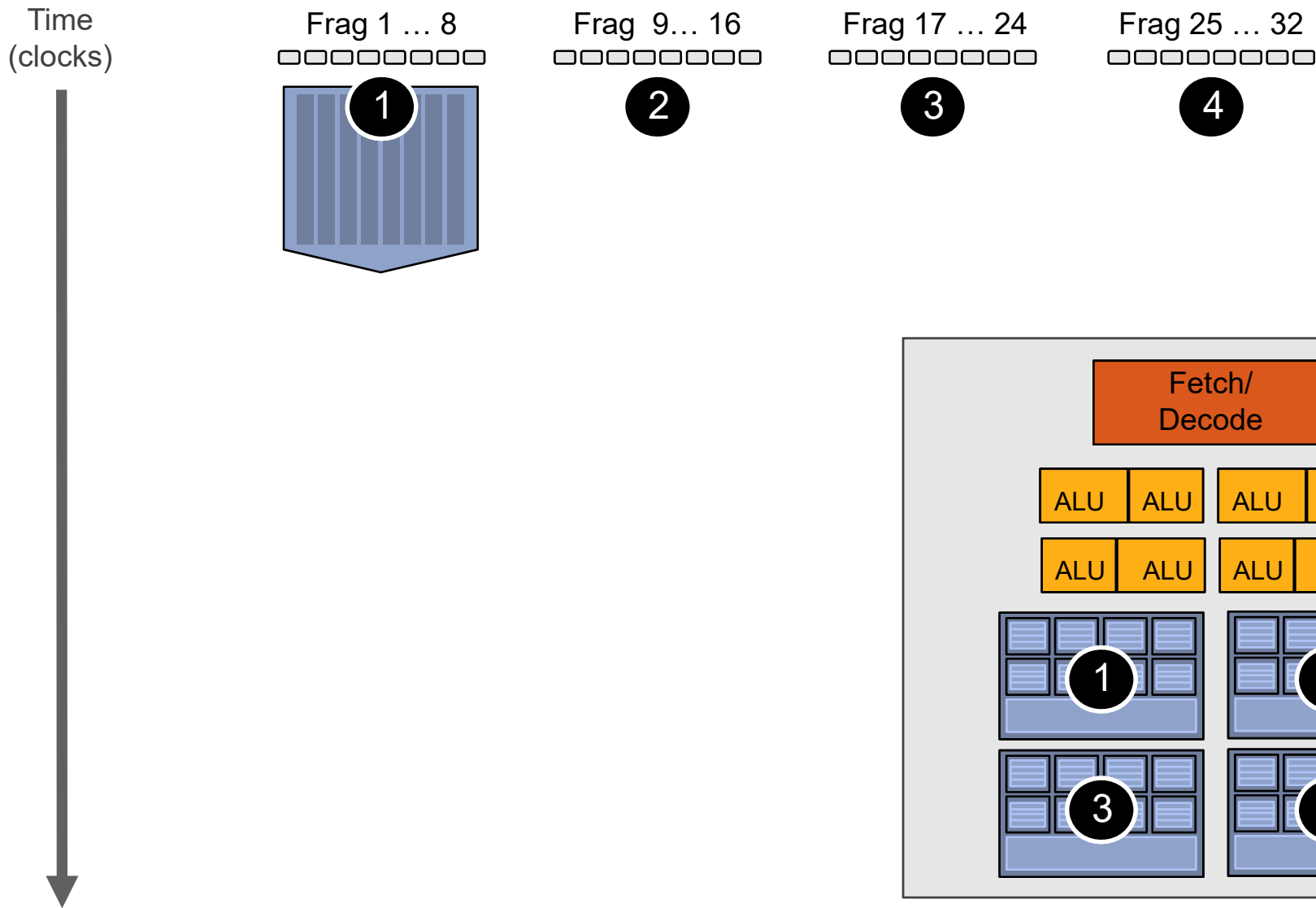
Hiding shader stalls

Time
(clocks)

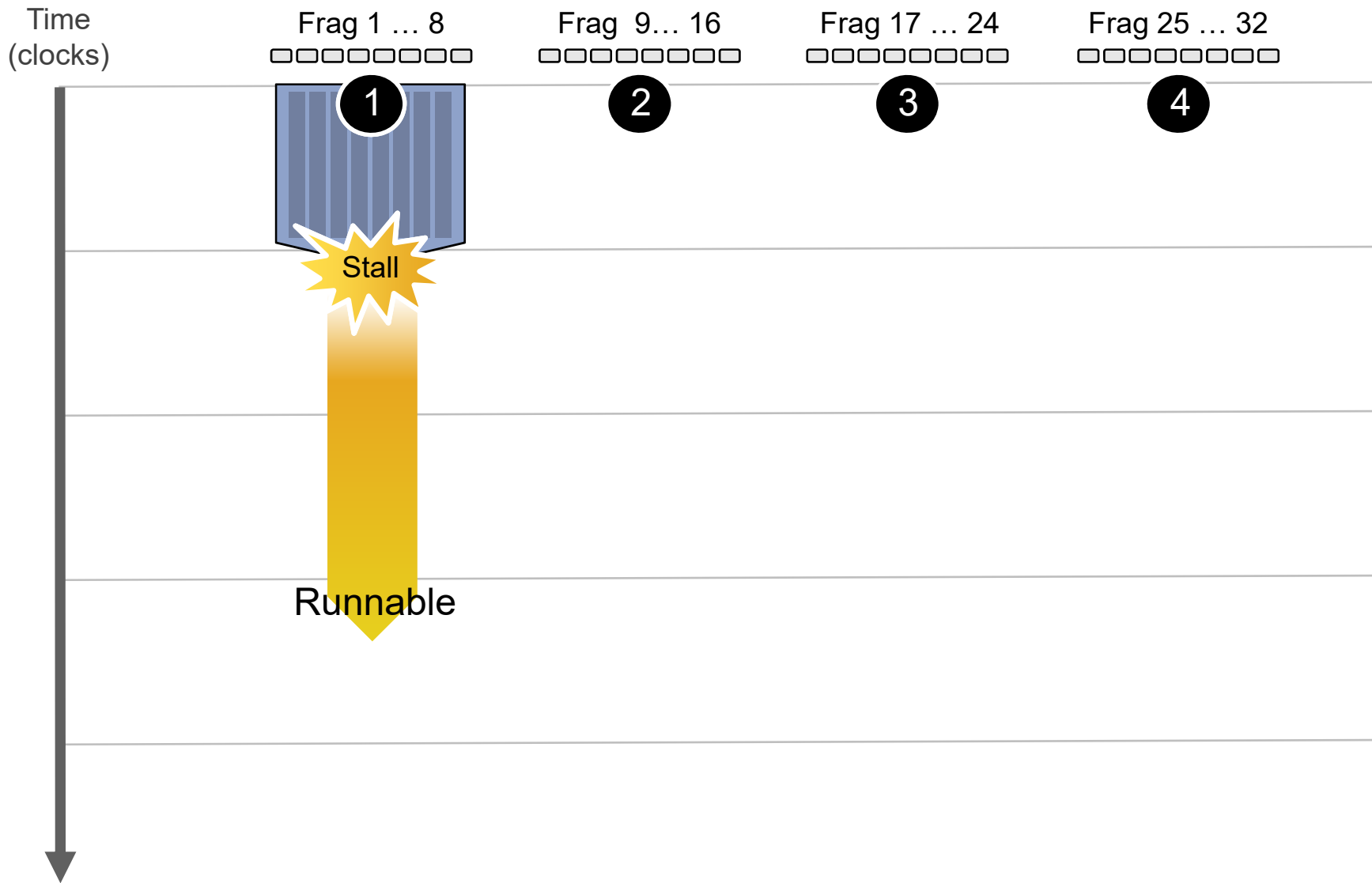
Frag 1 ... 8
□□□□□□□□



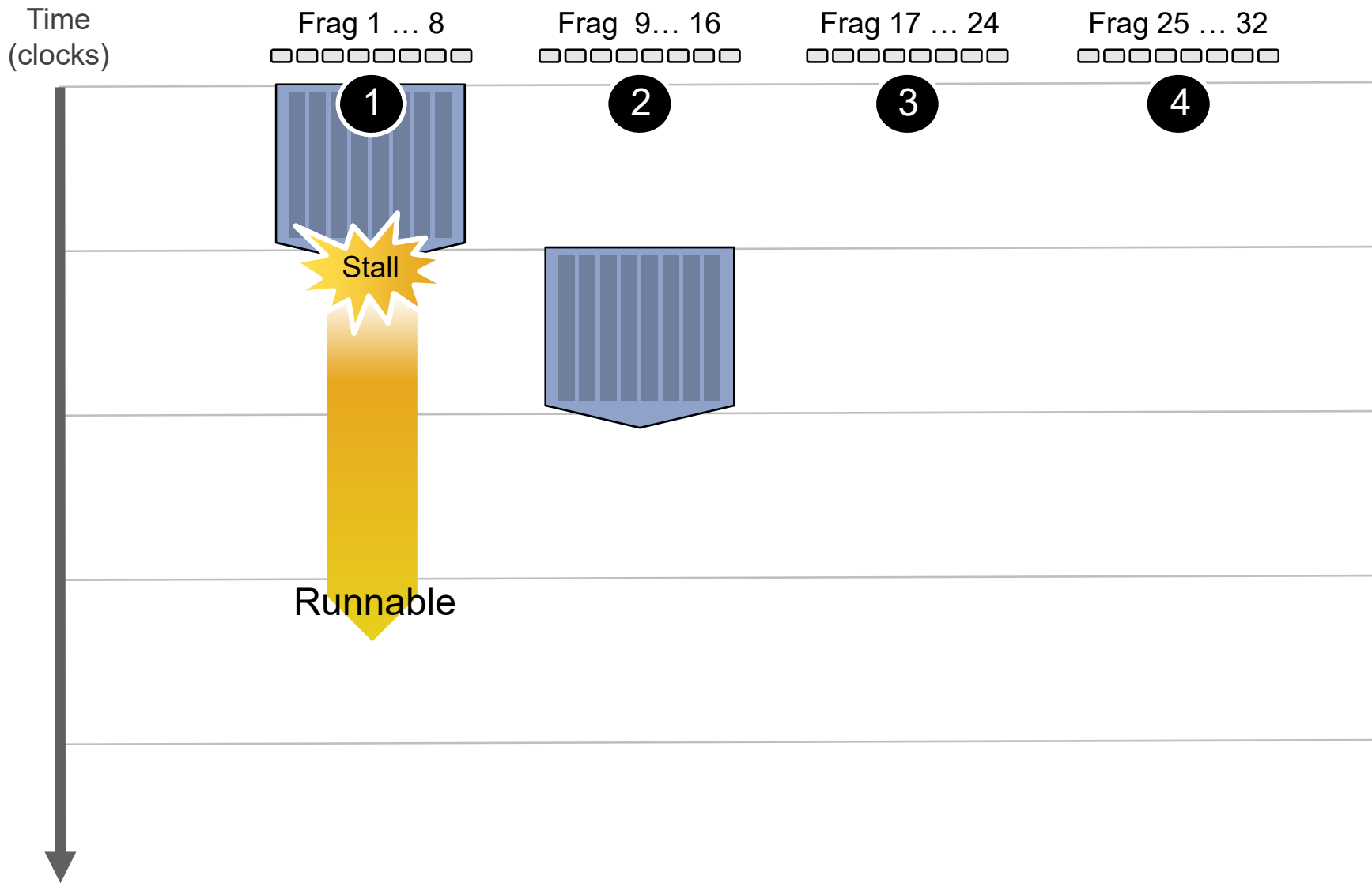
Hiding shader stalls



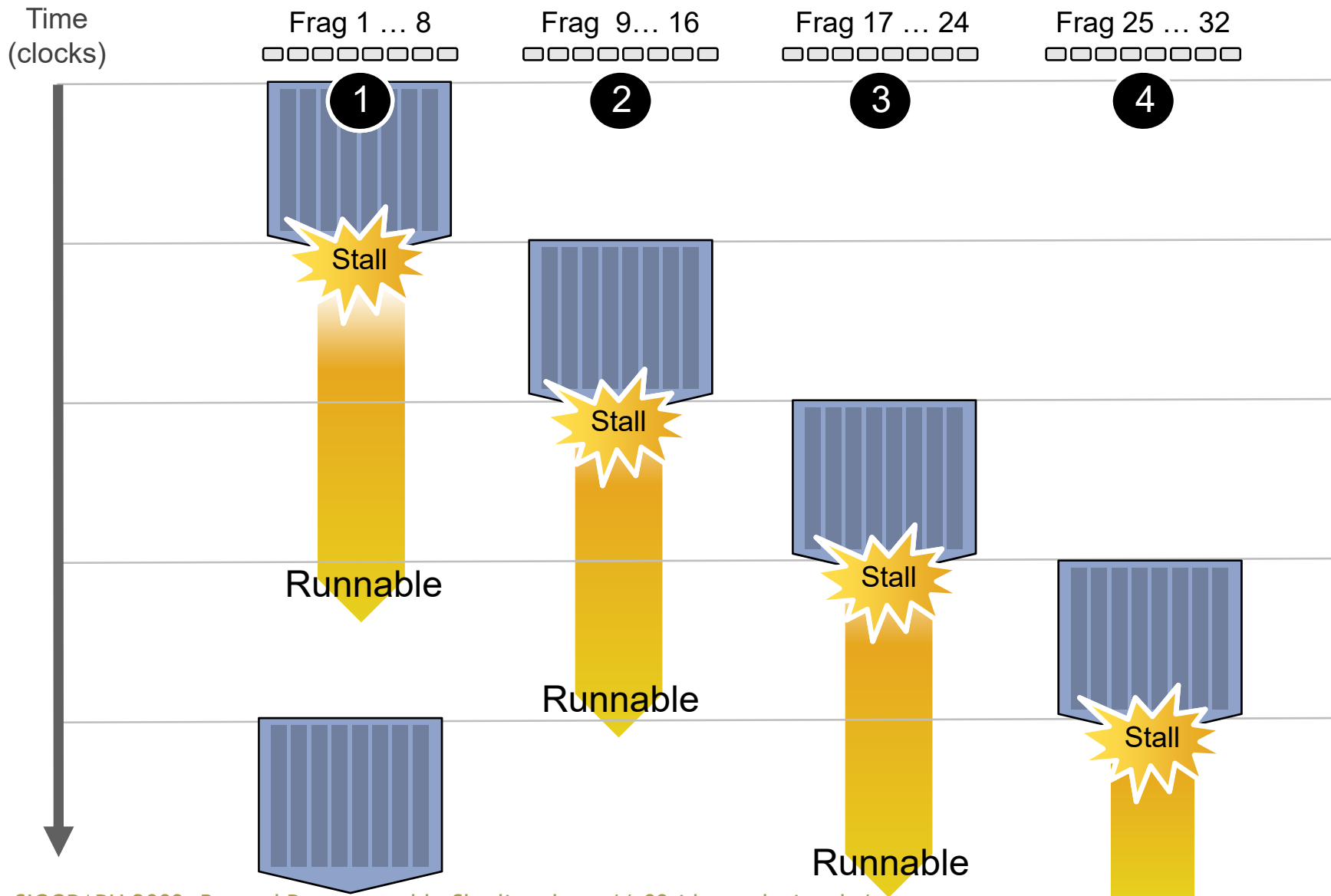
Hiding shader stalls



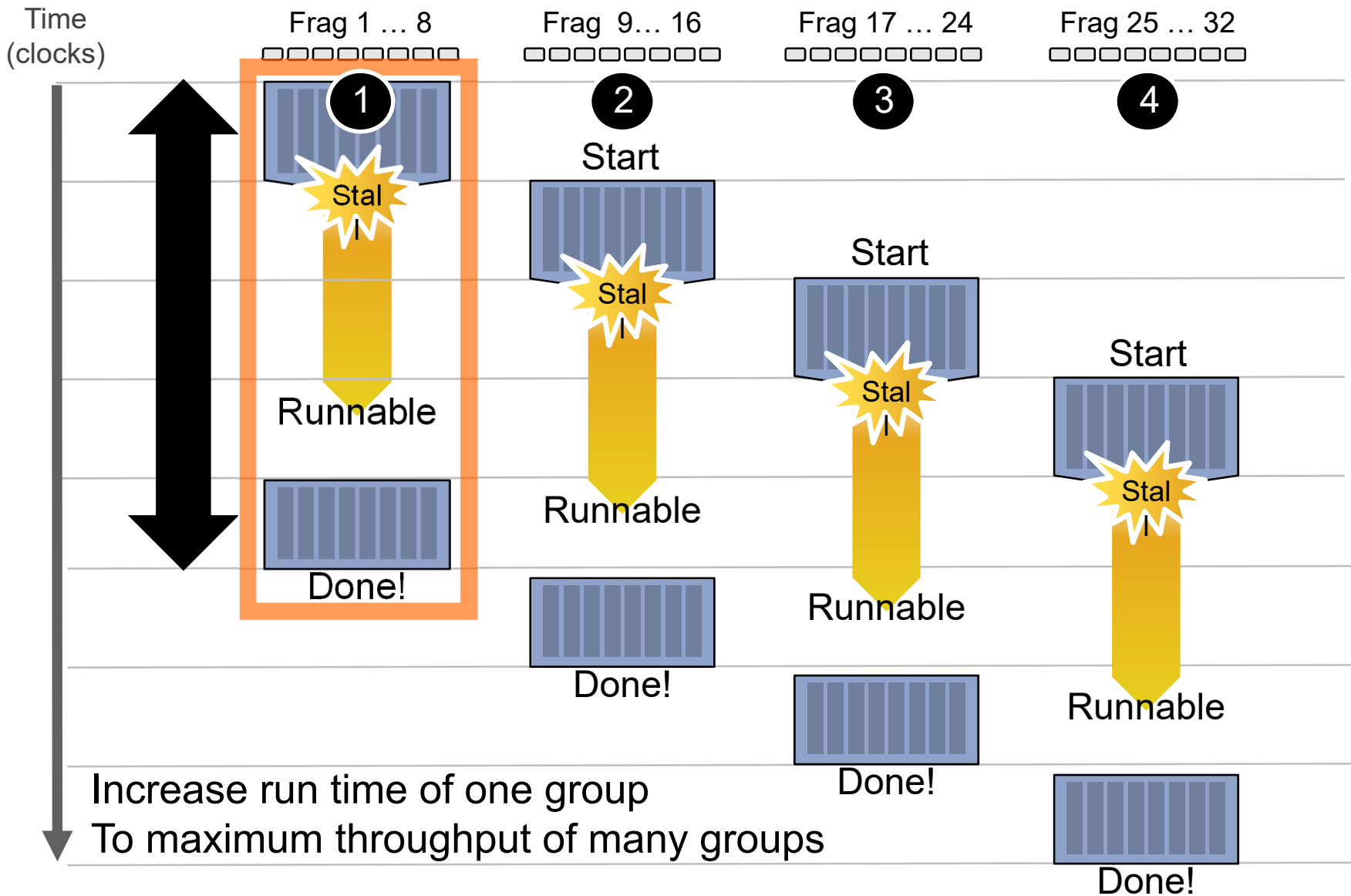
Hiding shader stalls



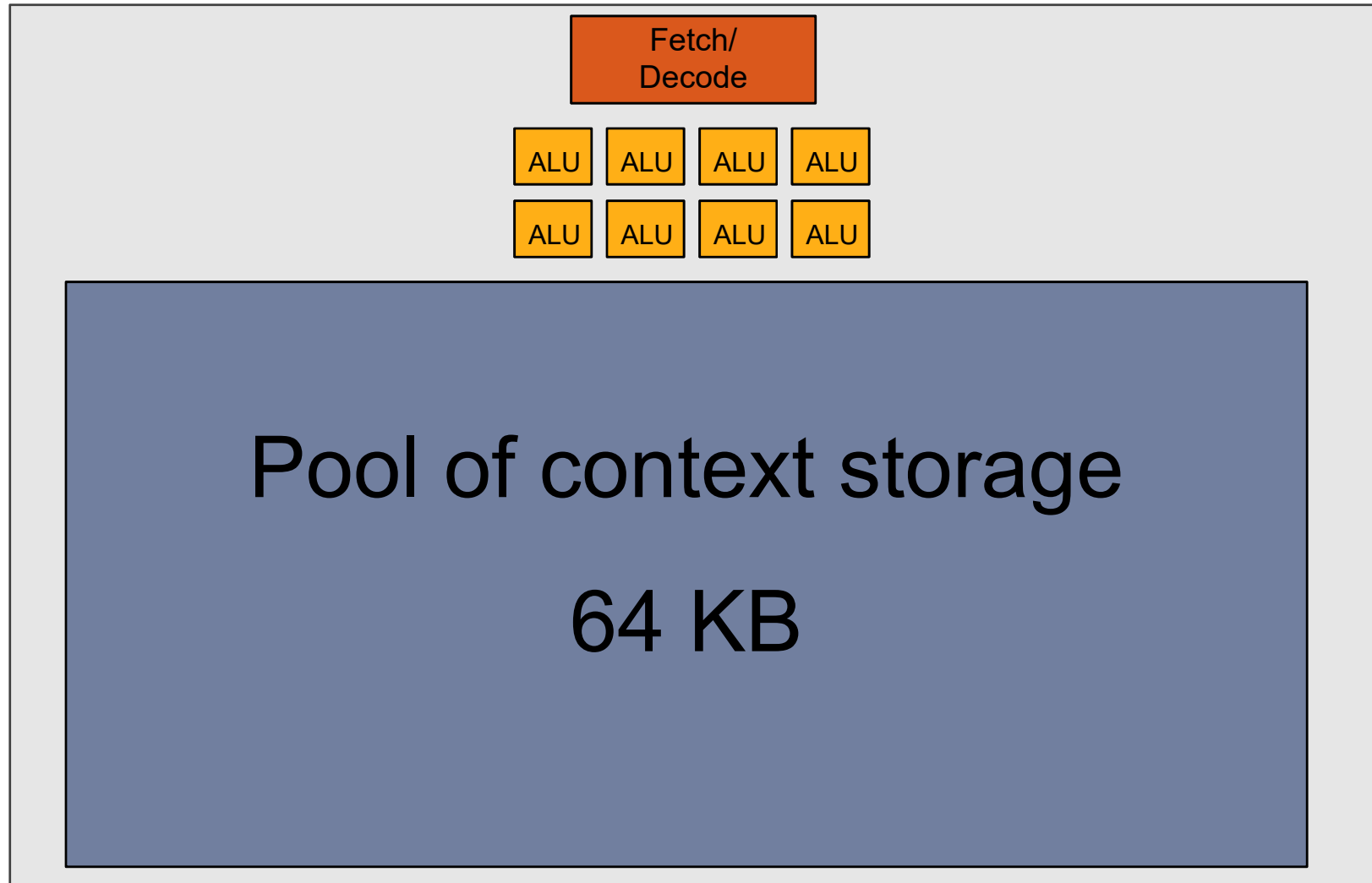
Hiding shader stalls



Throughput!

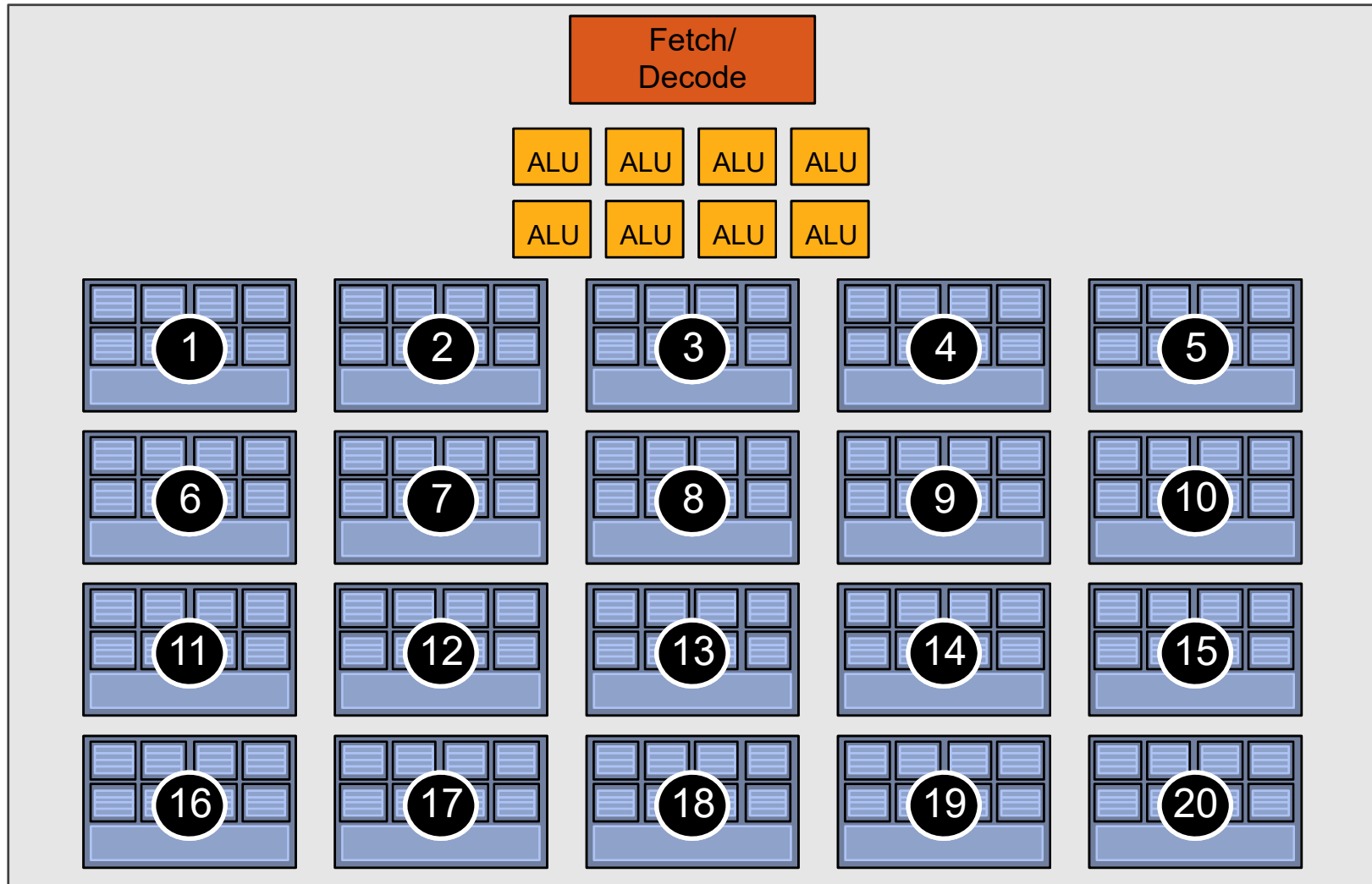


Storing contexts

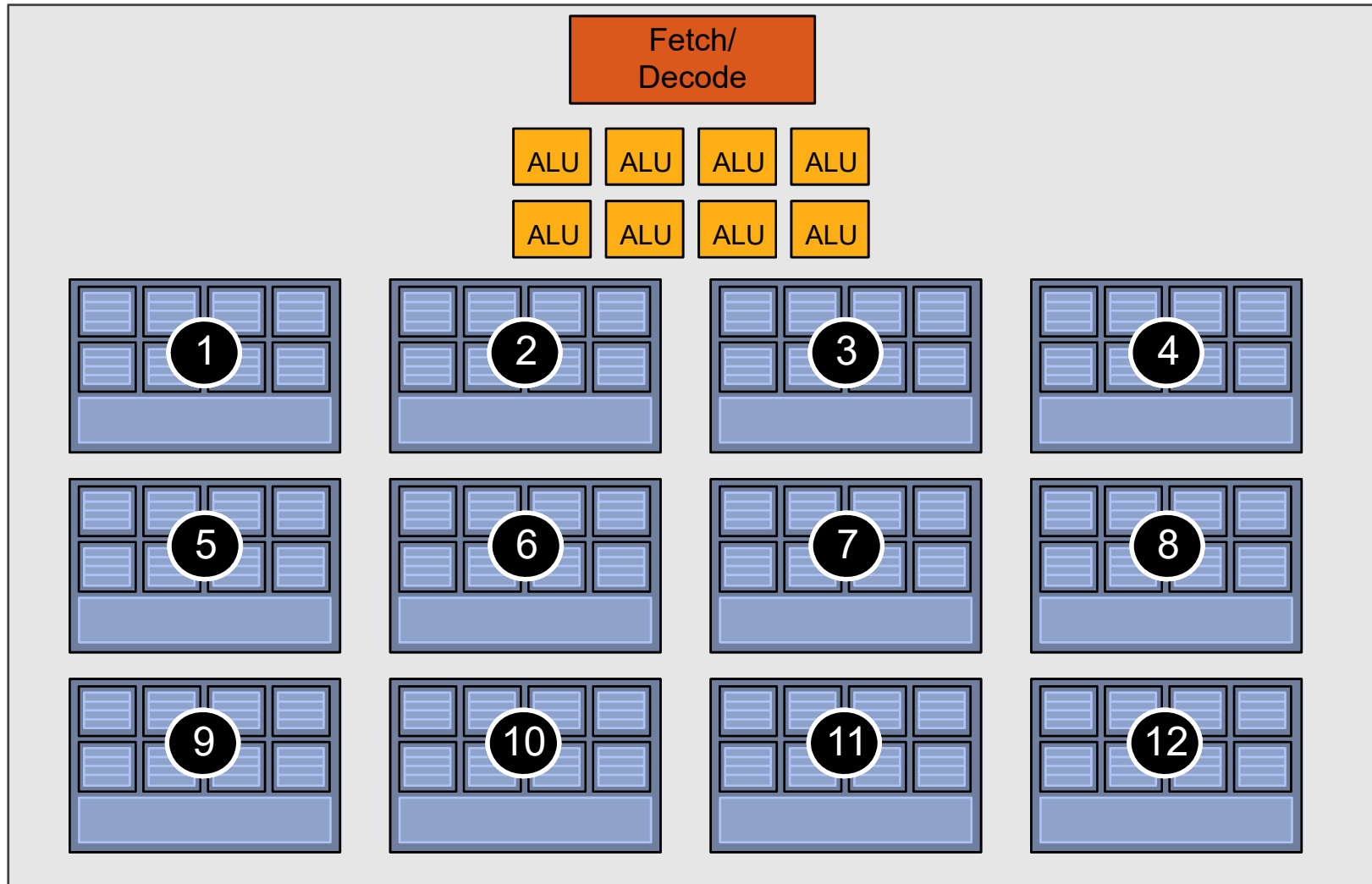


Twenty small contexts

(maximal latency hiding ability)

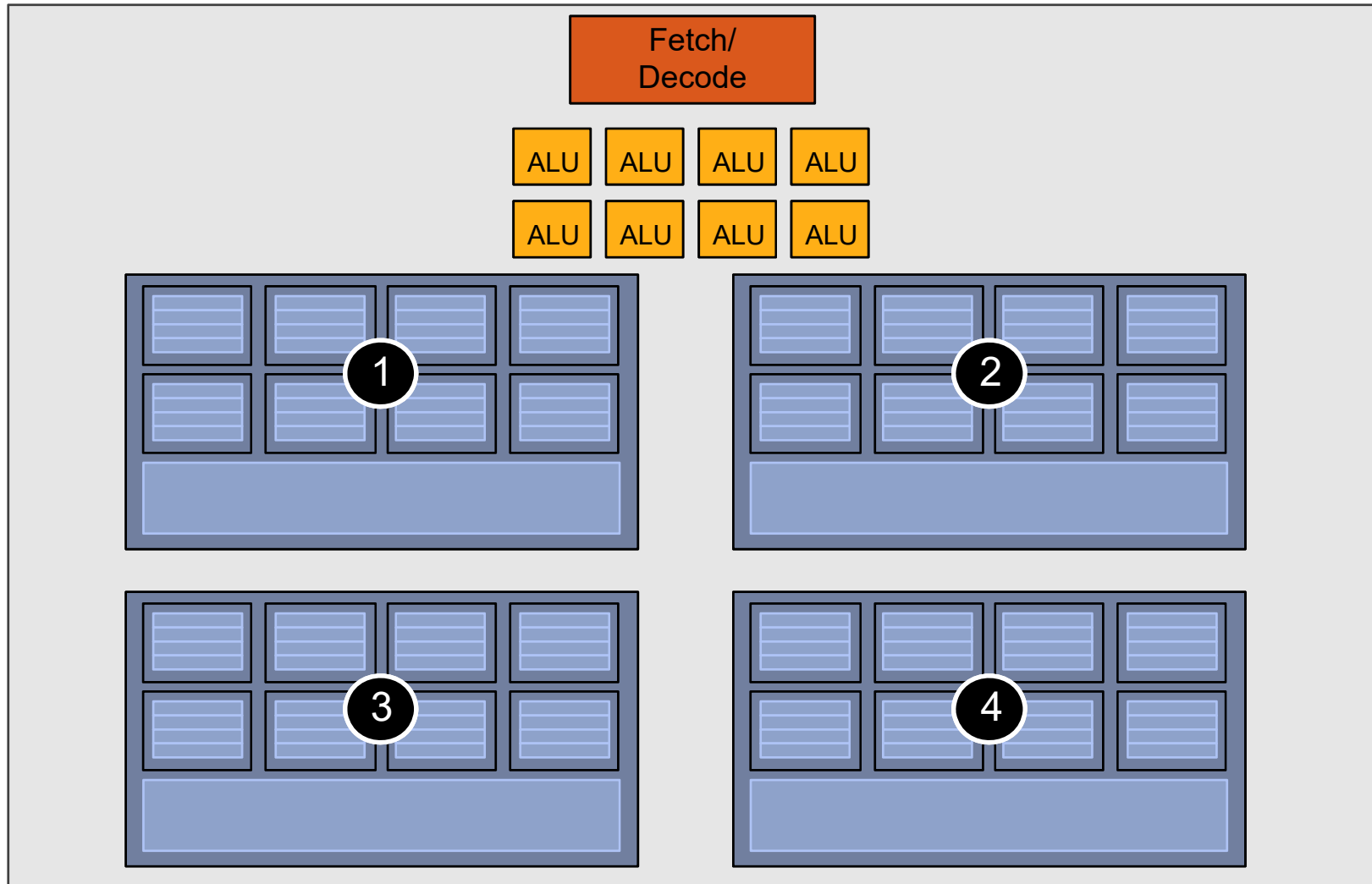


Twelve medium contexts



Four large contexts

(low latency hiding ability)



My chip!

16 cores

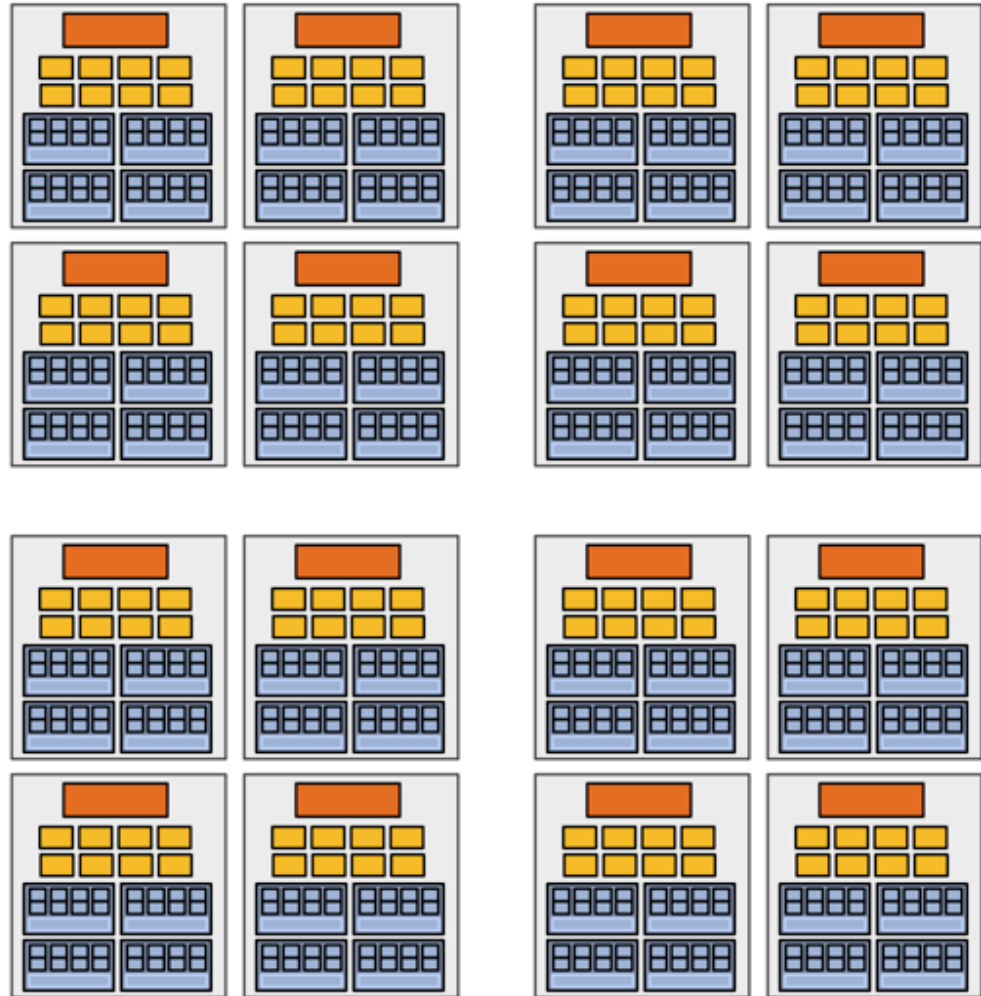
8 mul-add ALUs per core
(128 total)

16 simultaneous
instruction streams

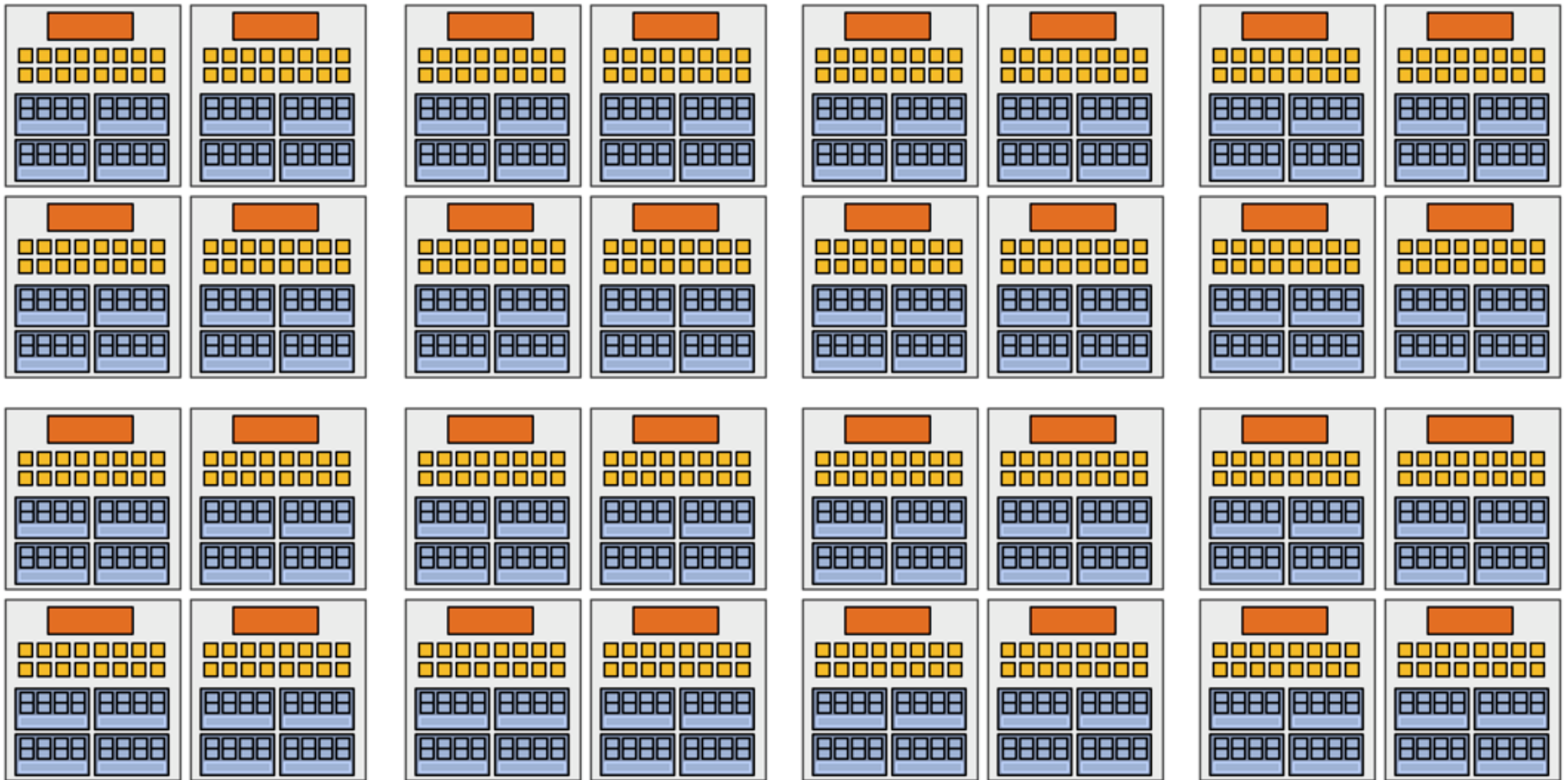
64 concurrent (but interleaved)
instruction streams

512 concurrent fragments

= 256 GFLOPs (@ 1GHz)



My “enthusiast” chip! (Some time ago :)



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

Where We've Arrived...



Summary: three key ideas for high-throughput execution

1. Use many “slimmed down cores,” run them in parallel
2. Pack cores full of ALUs (by sharing instruction stream overhead 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

**GPUs are here!
(usually)**

Thank you.