

GPUs

(based on Spring 2019 Lec 23)

Computer Graphics and Imaging
UC Berkeley CS184

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)



Unreal Engine Kite Demo (Epic Games 2015)



Unreal Engine 5 (2020)

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!

Today

- Key ideas/motivation behind GPU architecture
- High-performance graphics
- RTX

Part 1: throughput processing

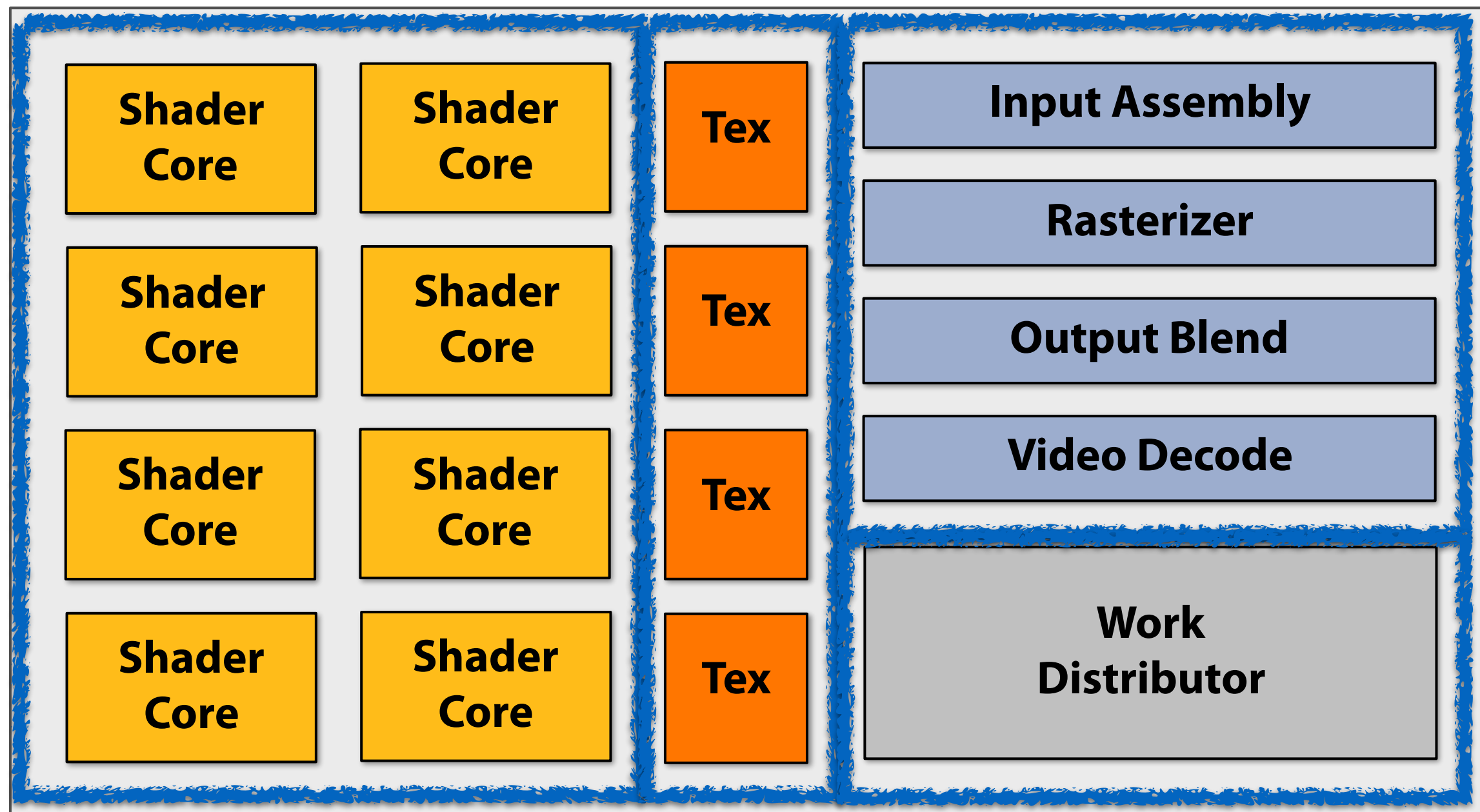
Three key concepts behind how modern GPU processing cores run code

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

```
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);  
}
```



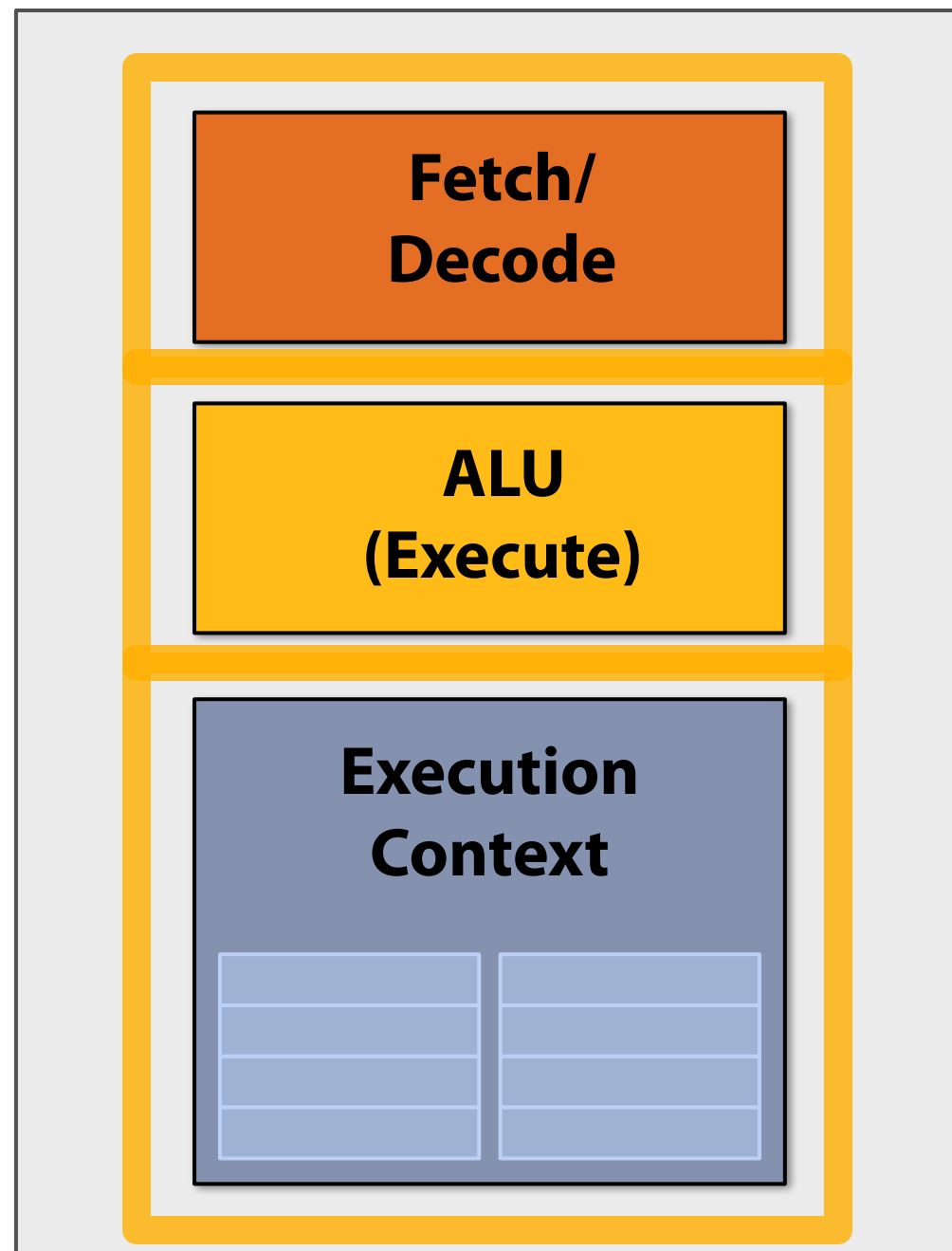
```
<diffuseShader>:  
sample r0, v4, t0, s0  
mul   r3, v0, cb0[0]  
madd  r3, v1, cb0[1], r3  
madd  r3, v2, cb0[2], r3  
clmp  r3, r3, 1(0.0), 1(1.0)  
mul   o0, r0, r3  
mul   o1, r1, r3  
mul   o2, r2, r3  
mov   o3, 1(1.0)
```



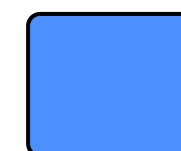
1 shaded fragment output record



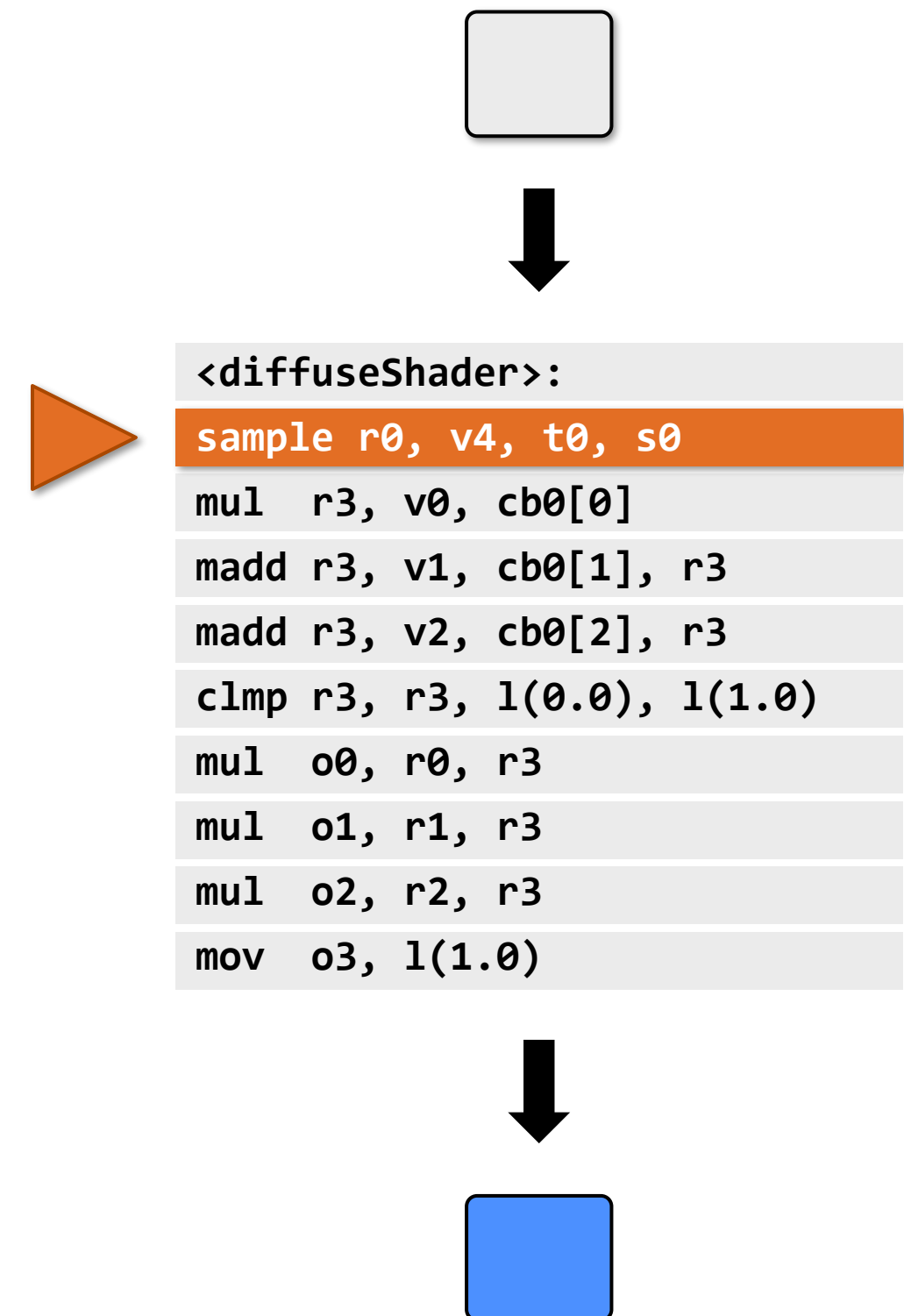
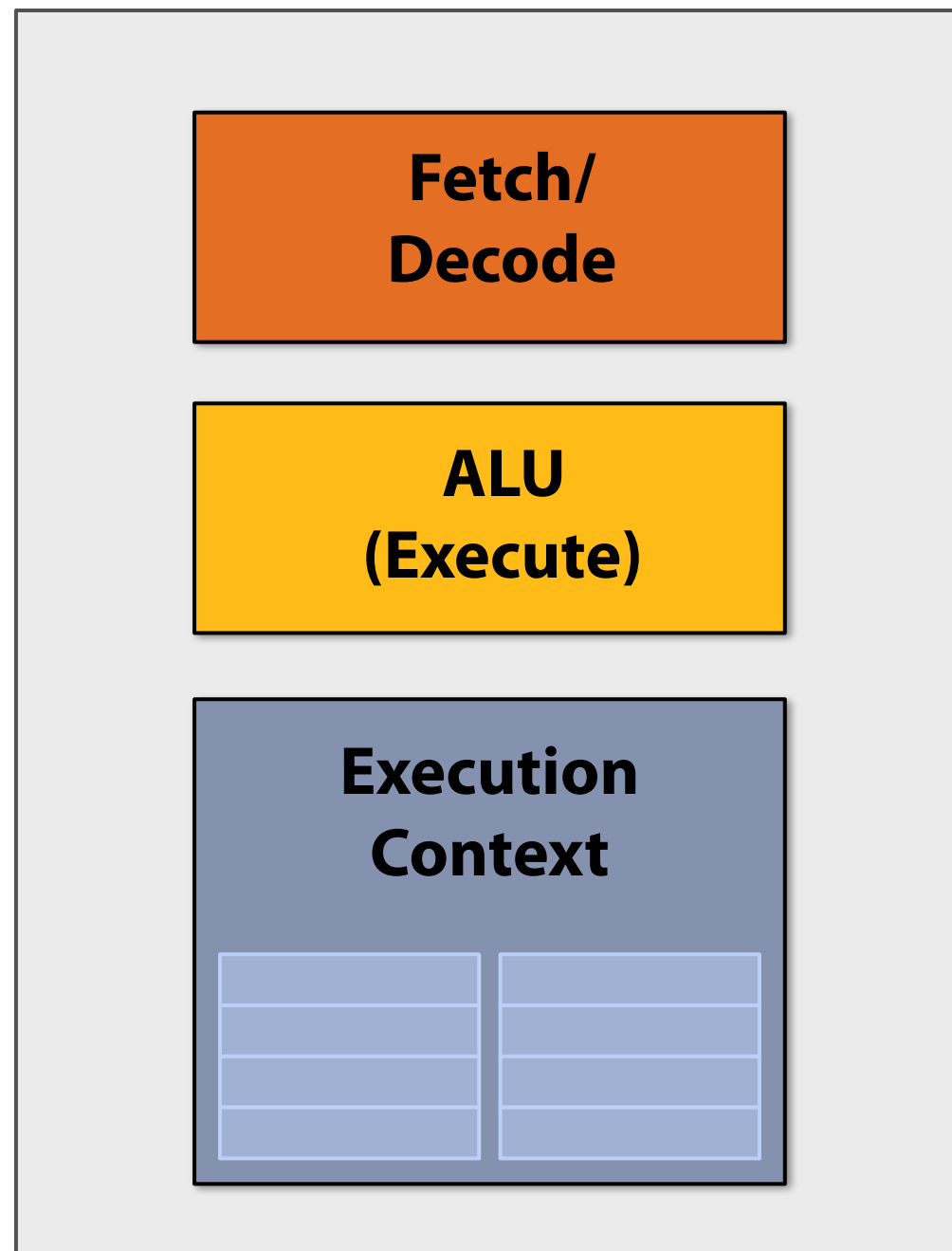
Execute shader



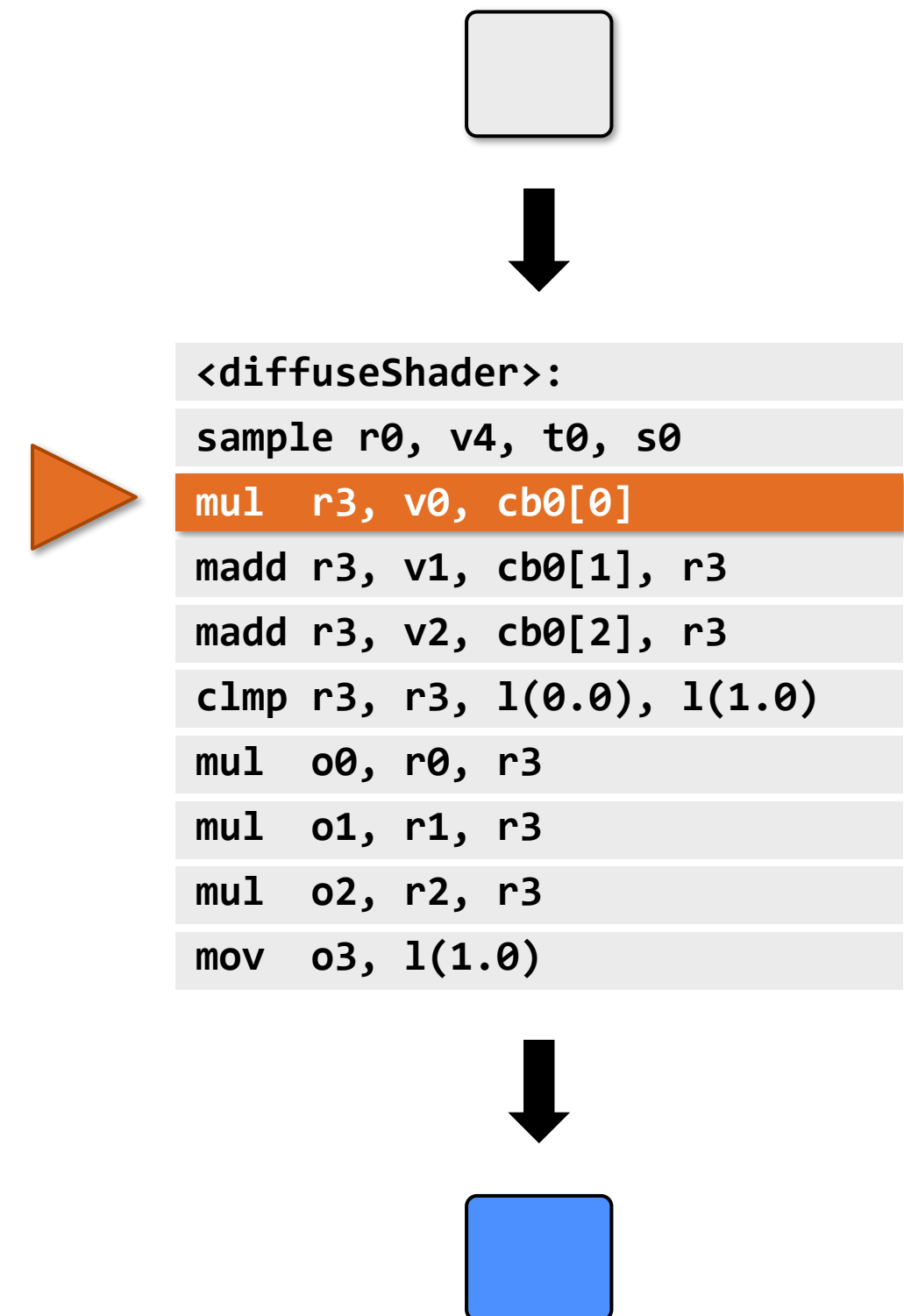
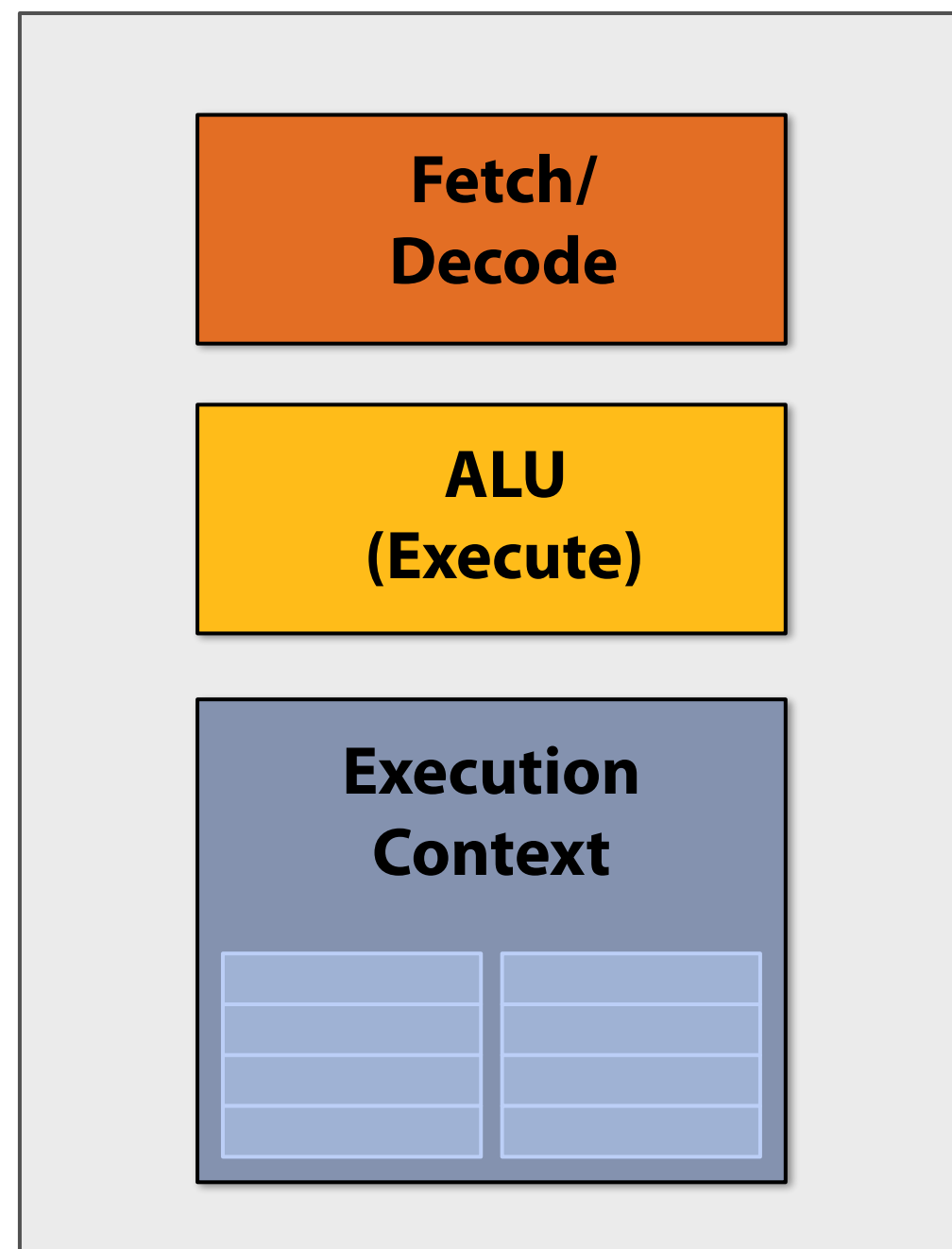
```
<diffuseShader>:  
sample r0, v4, t0, s0  
mul  r3, v0, cb0[0]  
madd r3, v1, cb0[1], r3  
madd r3, v2, cb0[2], r3  
clmp r3, r3, 1(0.0), 1(1.0)  
mul  o0, r0, r3  
mul  o1, r1, r3  
mul  o2, r2, r3  
mov  o3, 1(1.0)
```



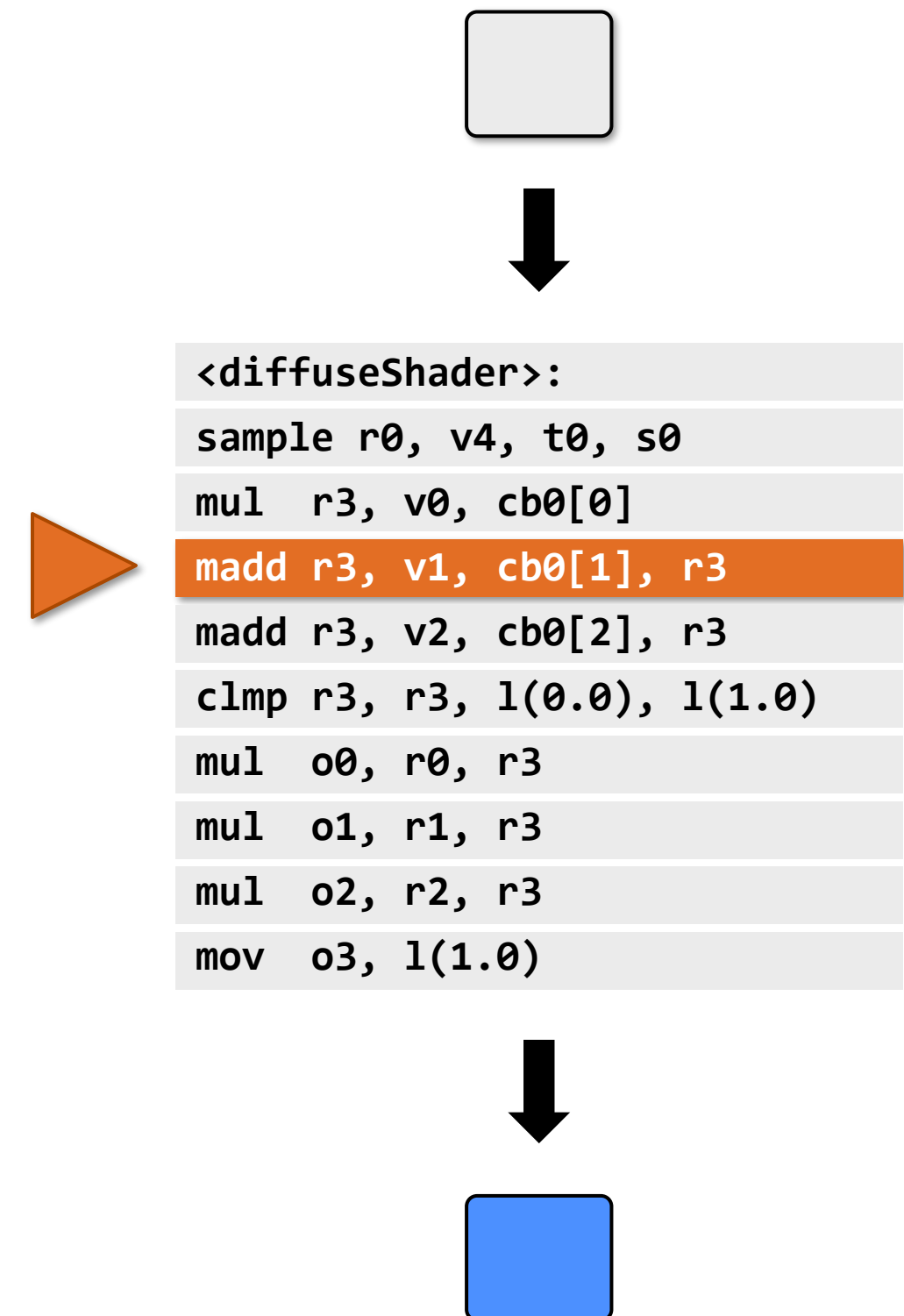
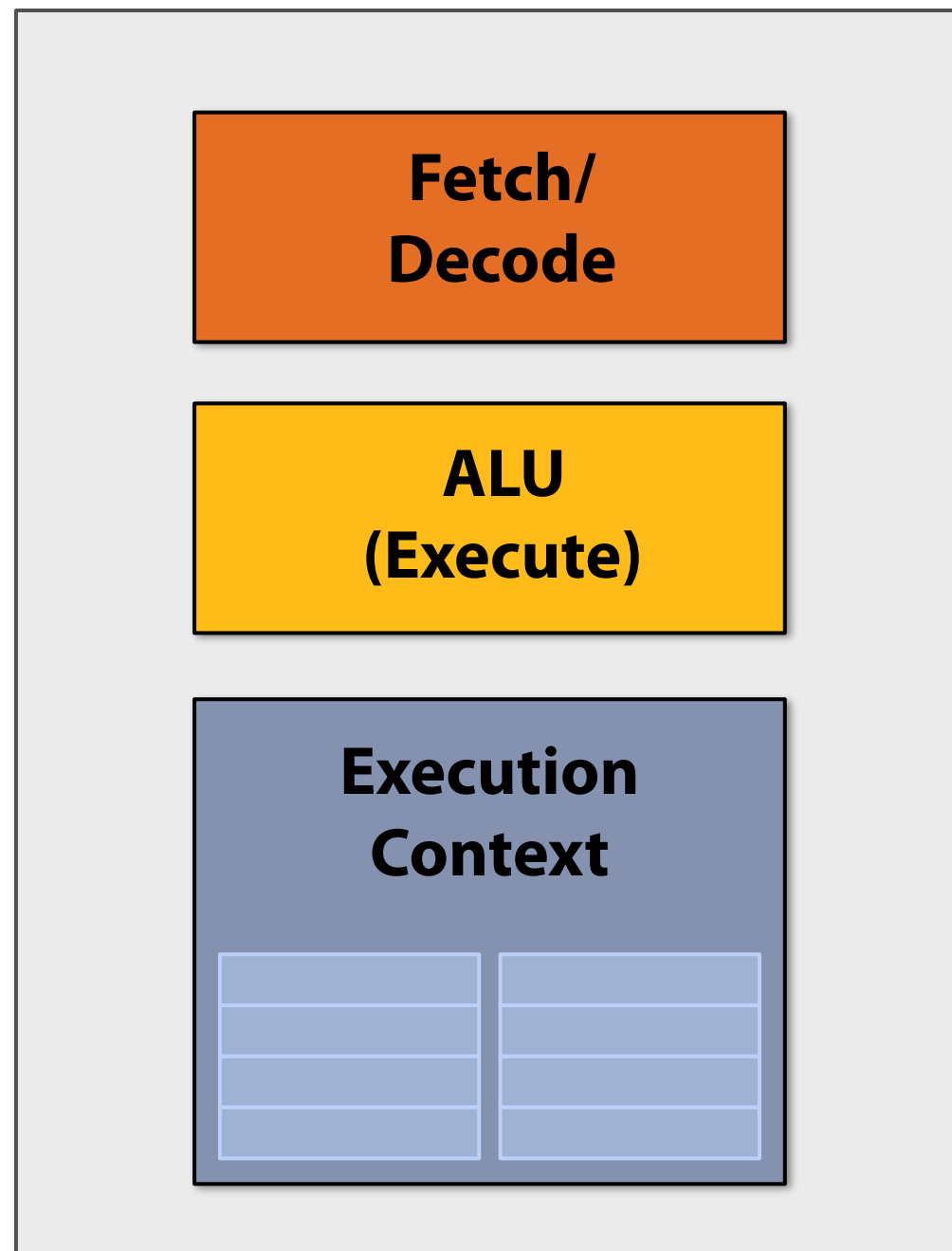
Execute shader



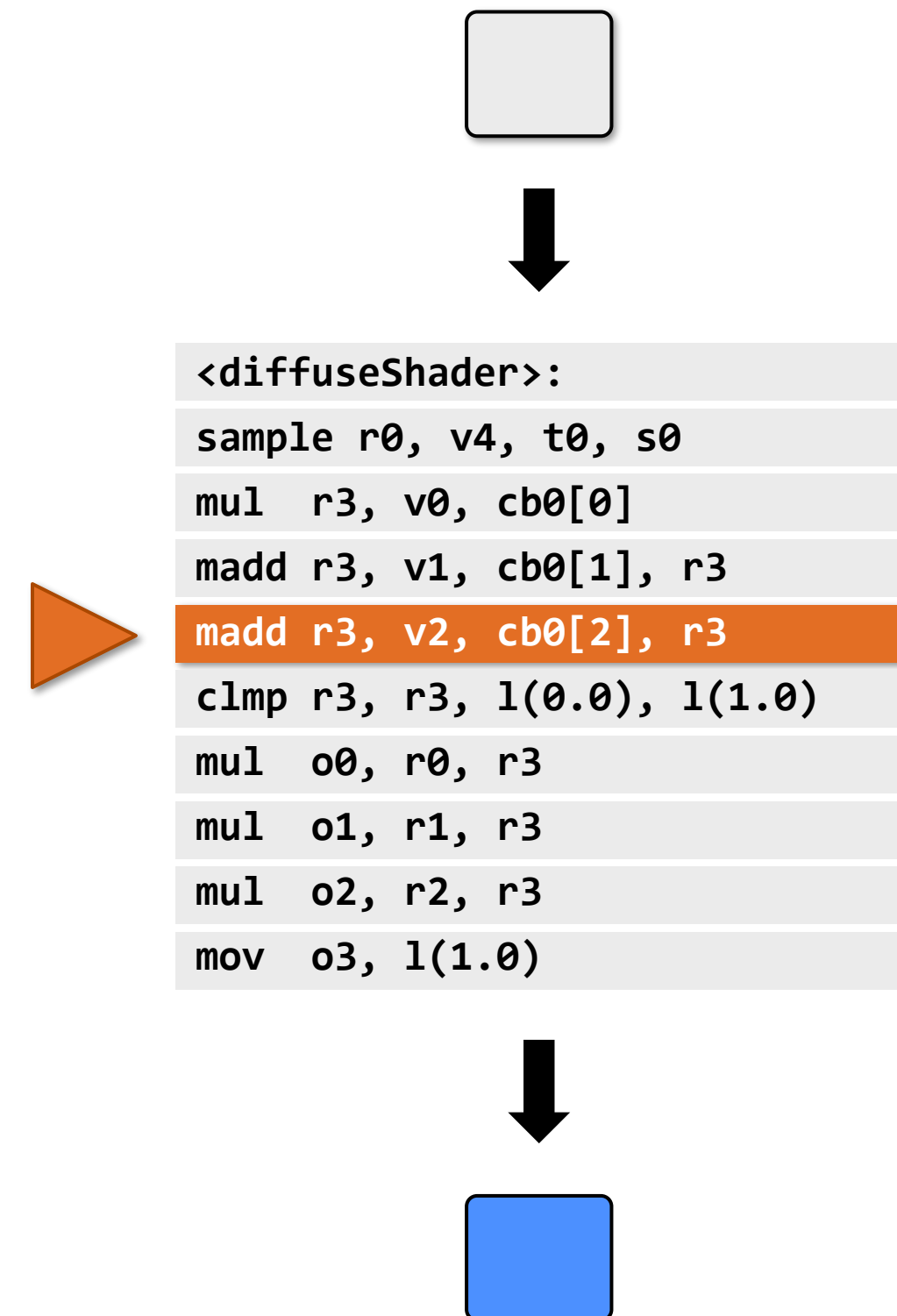
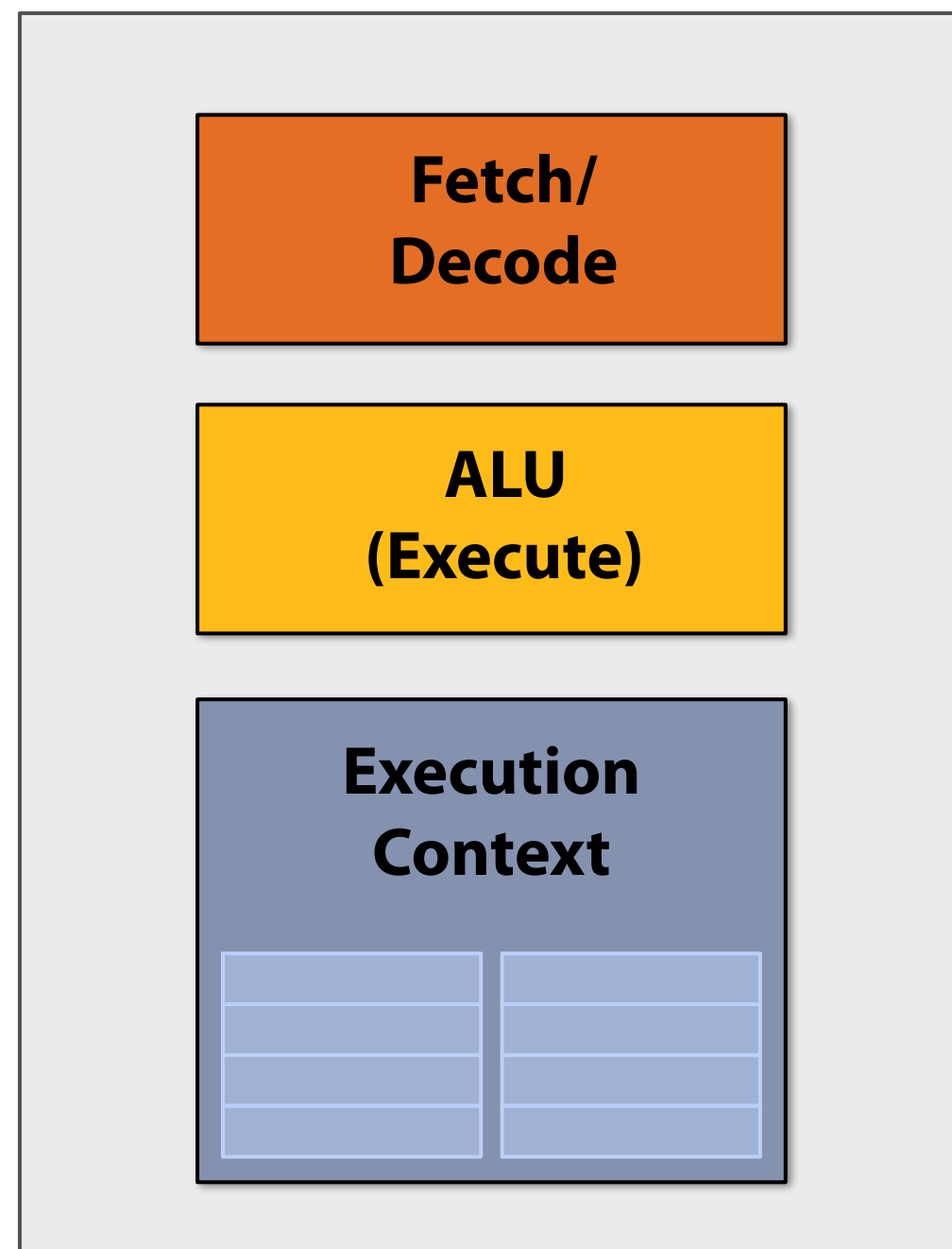
Execute shader



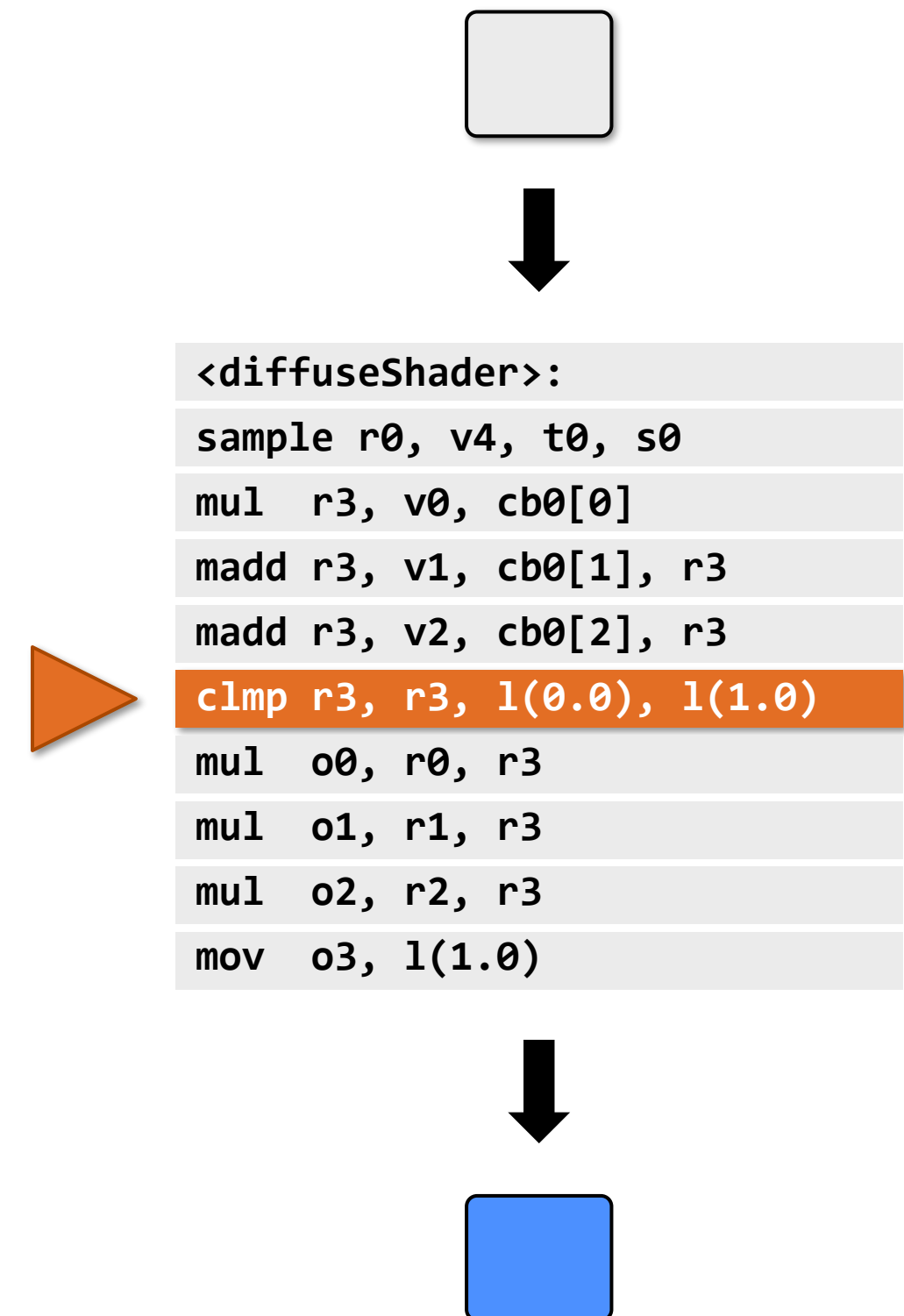
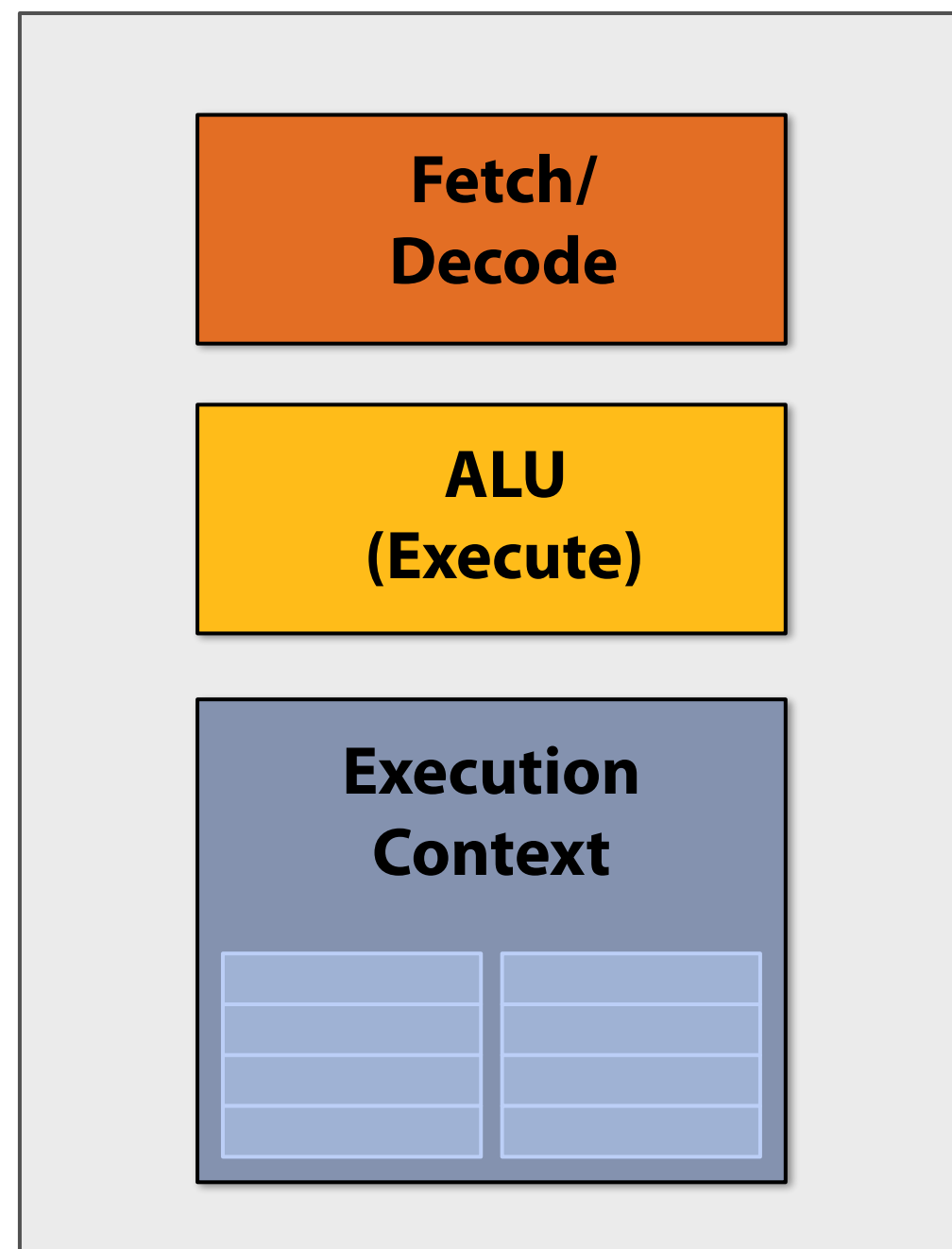
Execute shader



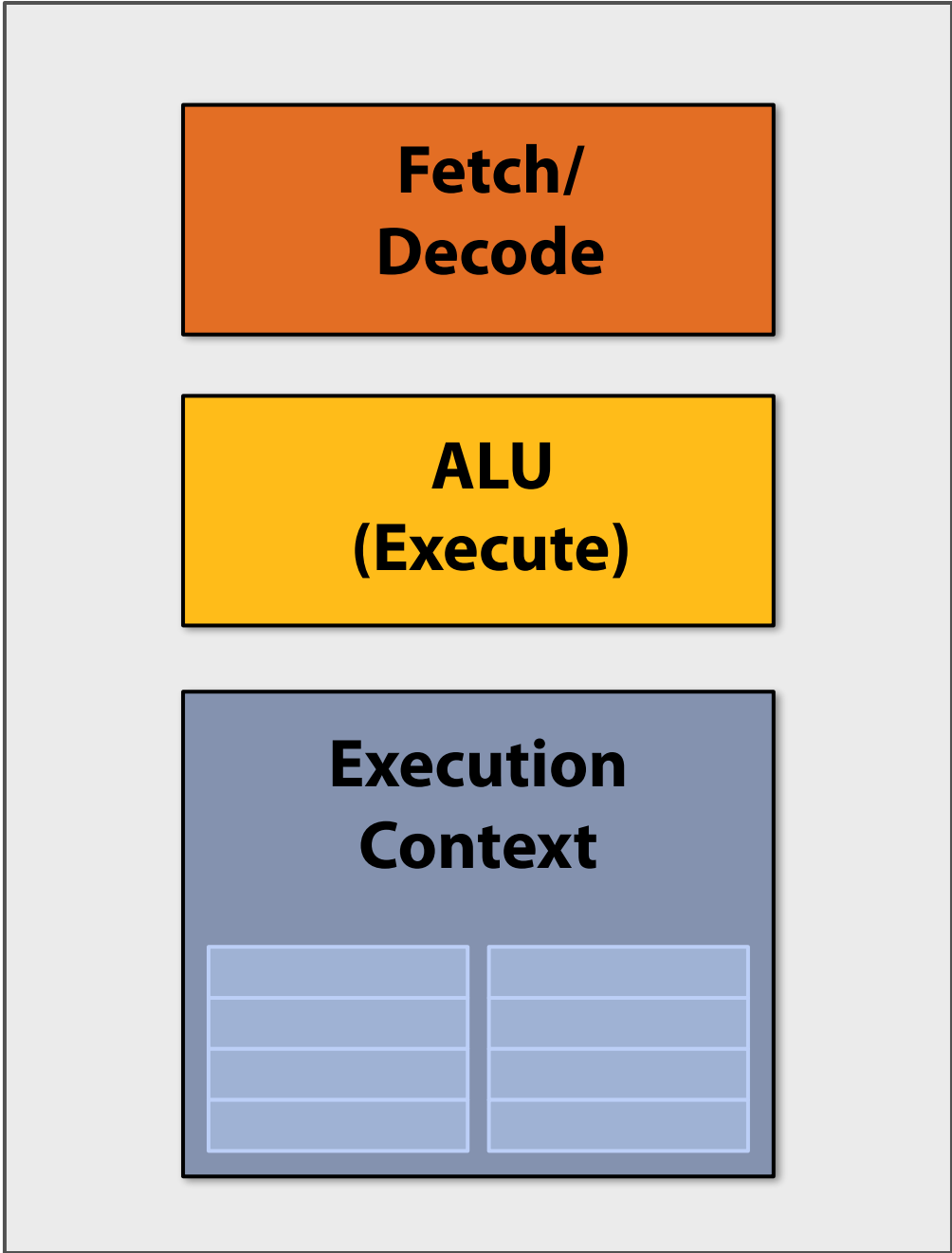
Execute shader



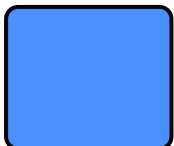
Execute shader



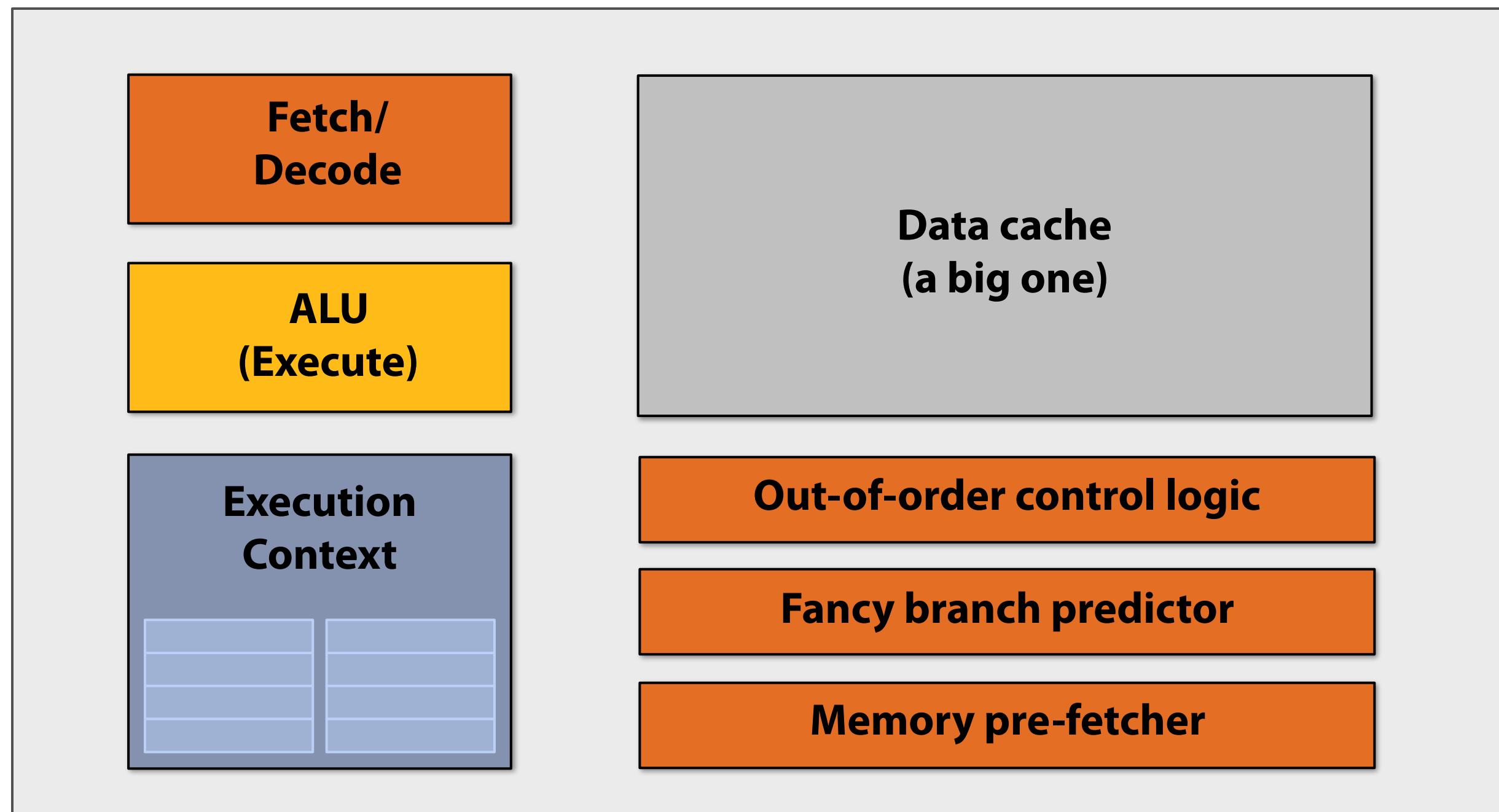
Execute shader



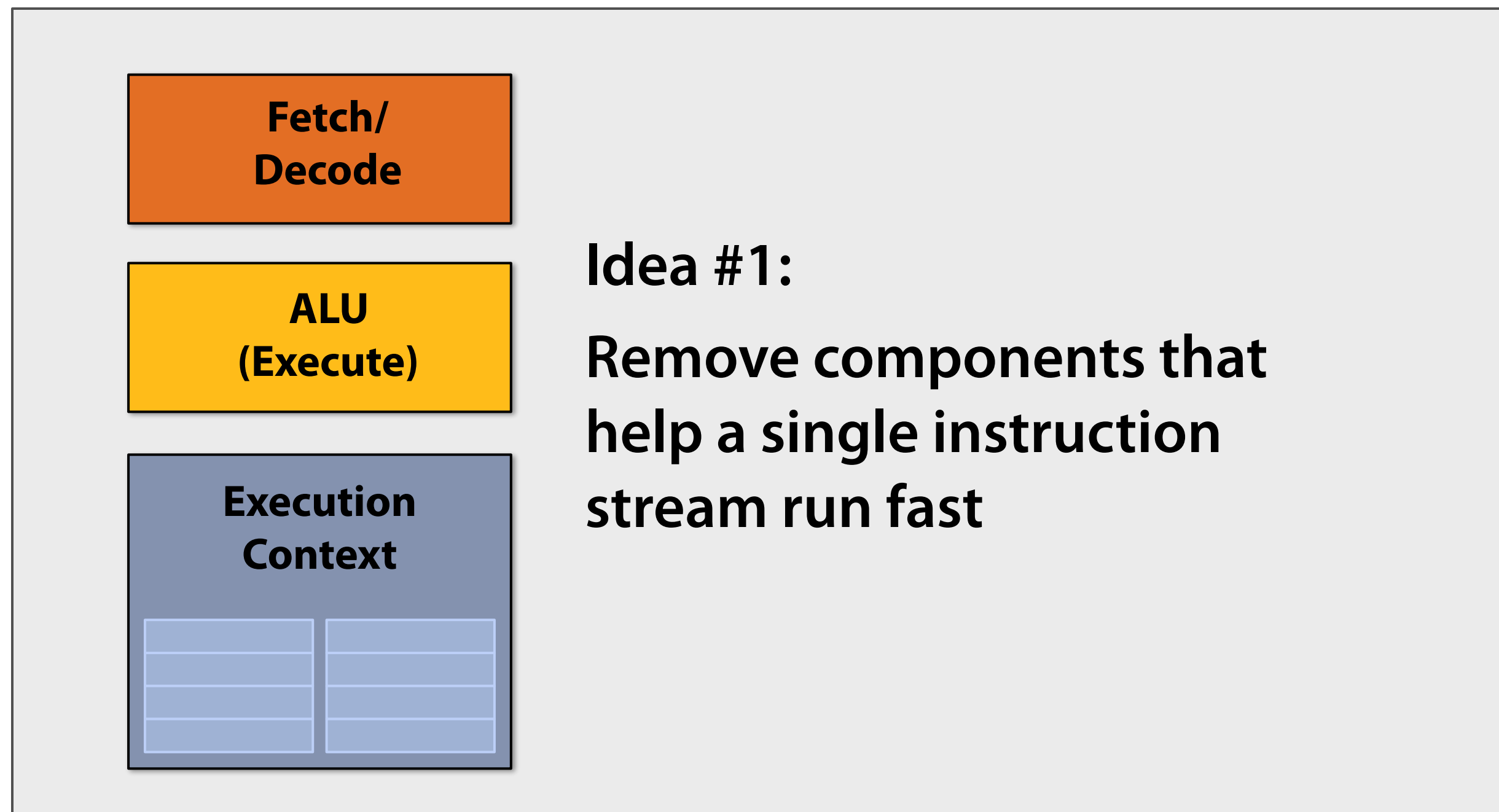
```
<diffuseShader>:  
sample r0, v4, t0, s0  
mul  r3, v0, cb0[0]  
madd r3, v1, cb0[1], r3  
madd r3, v2, cb0[2], r3  
clmp r3, r3, 1(0.0), 1(1.0)  
mul  o0, r0, r3  
mul  o1, r1, r3  
mul  o2, r2, r3  
mov  o3, 1(1.0)
```



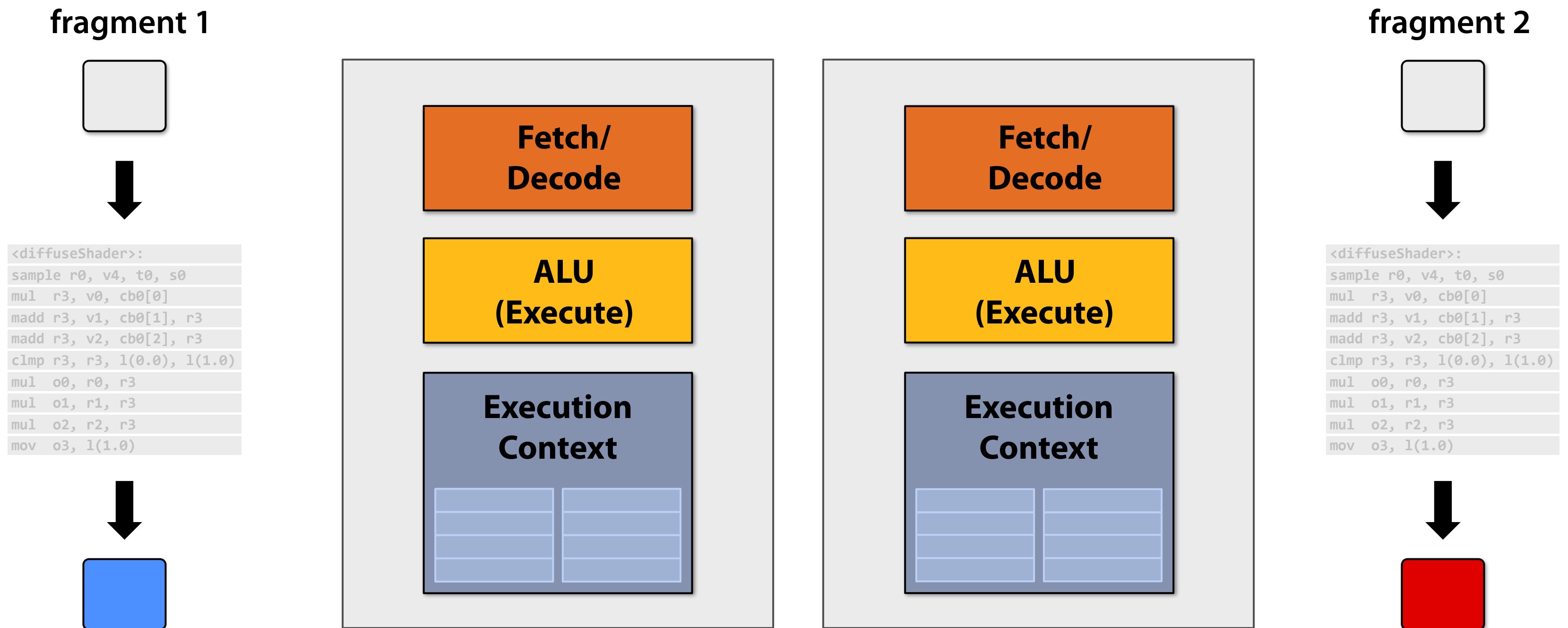
“CPU-style” cores



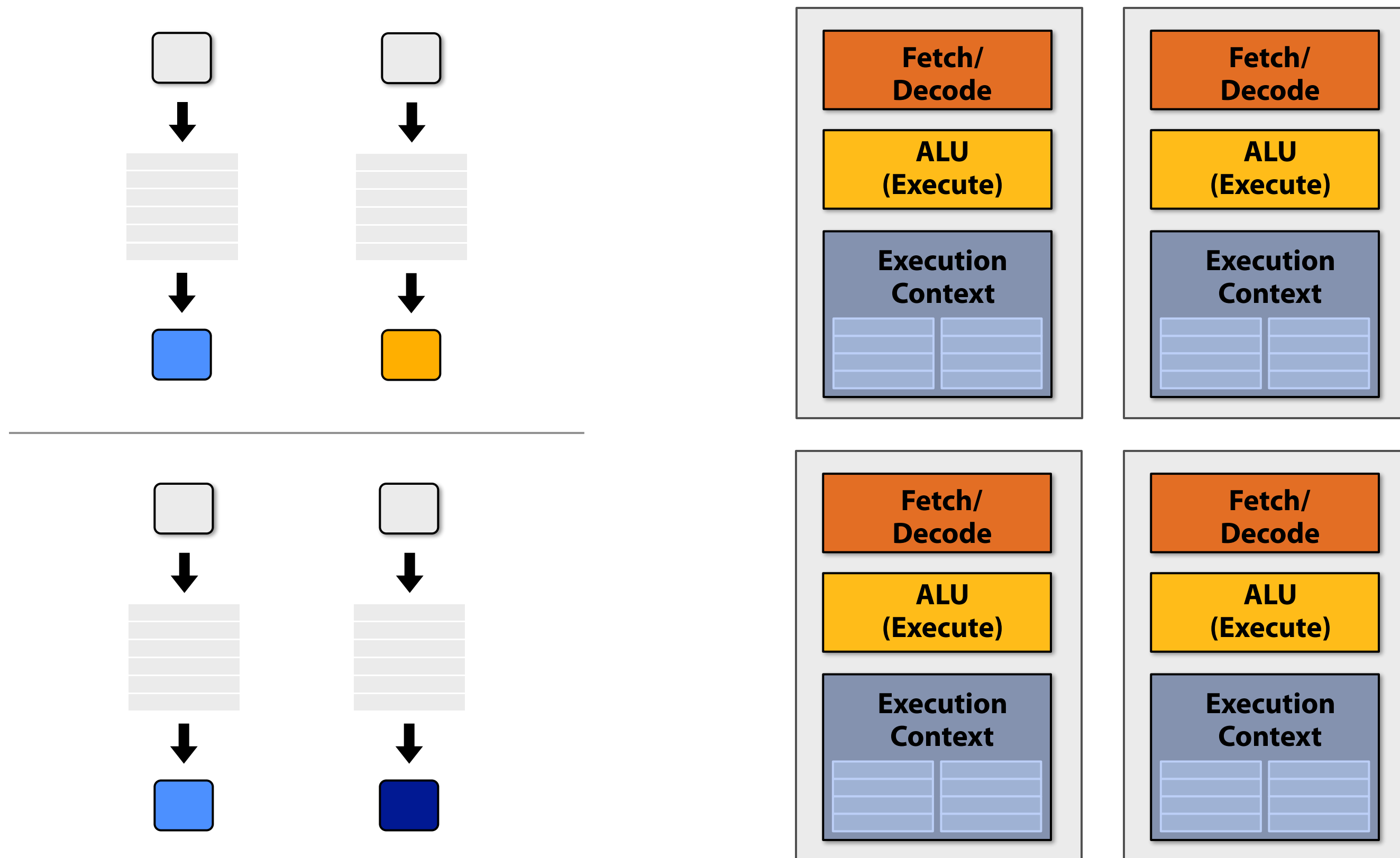
Slimming down



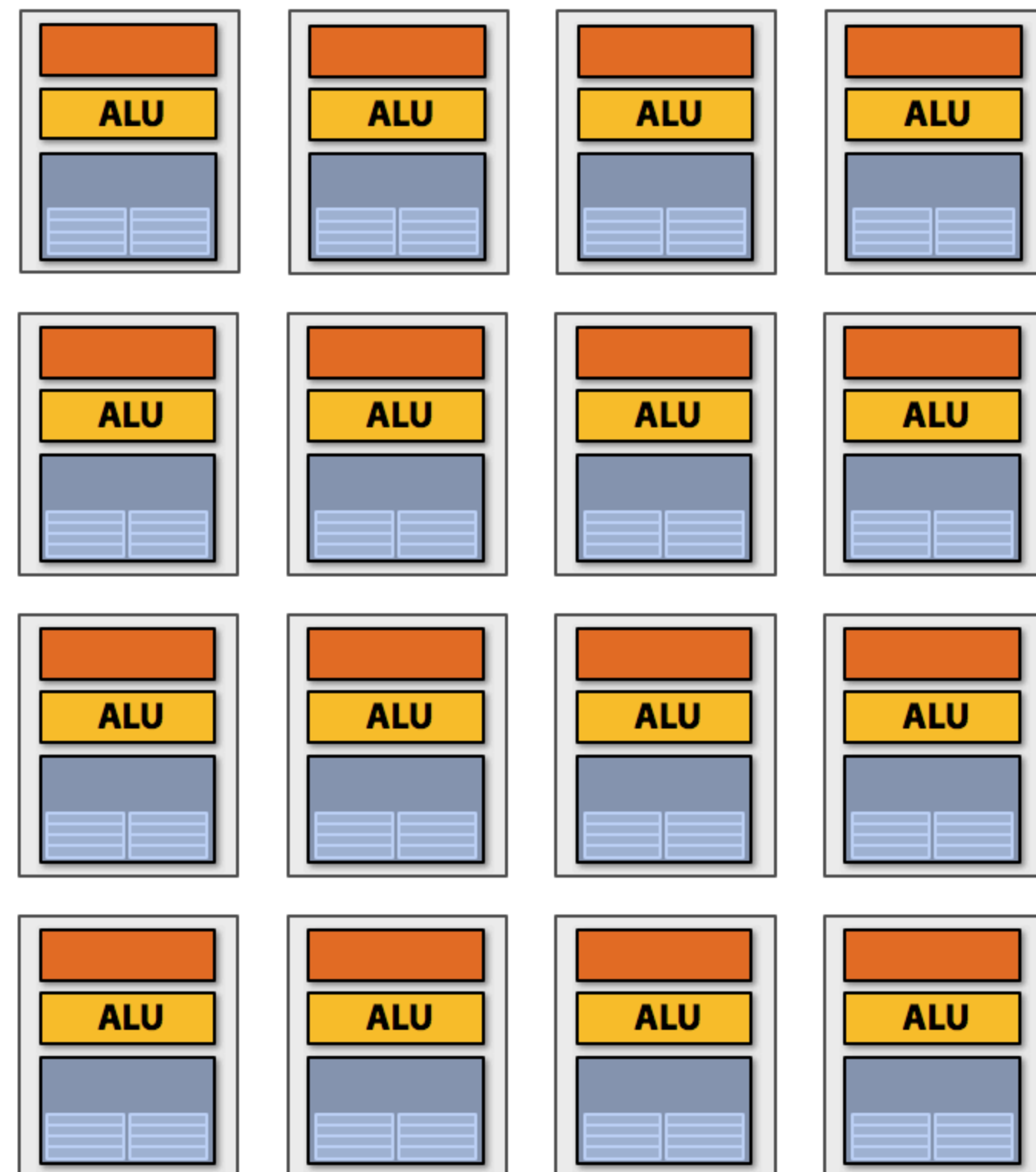
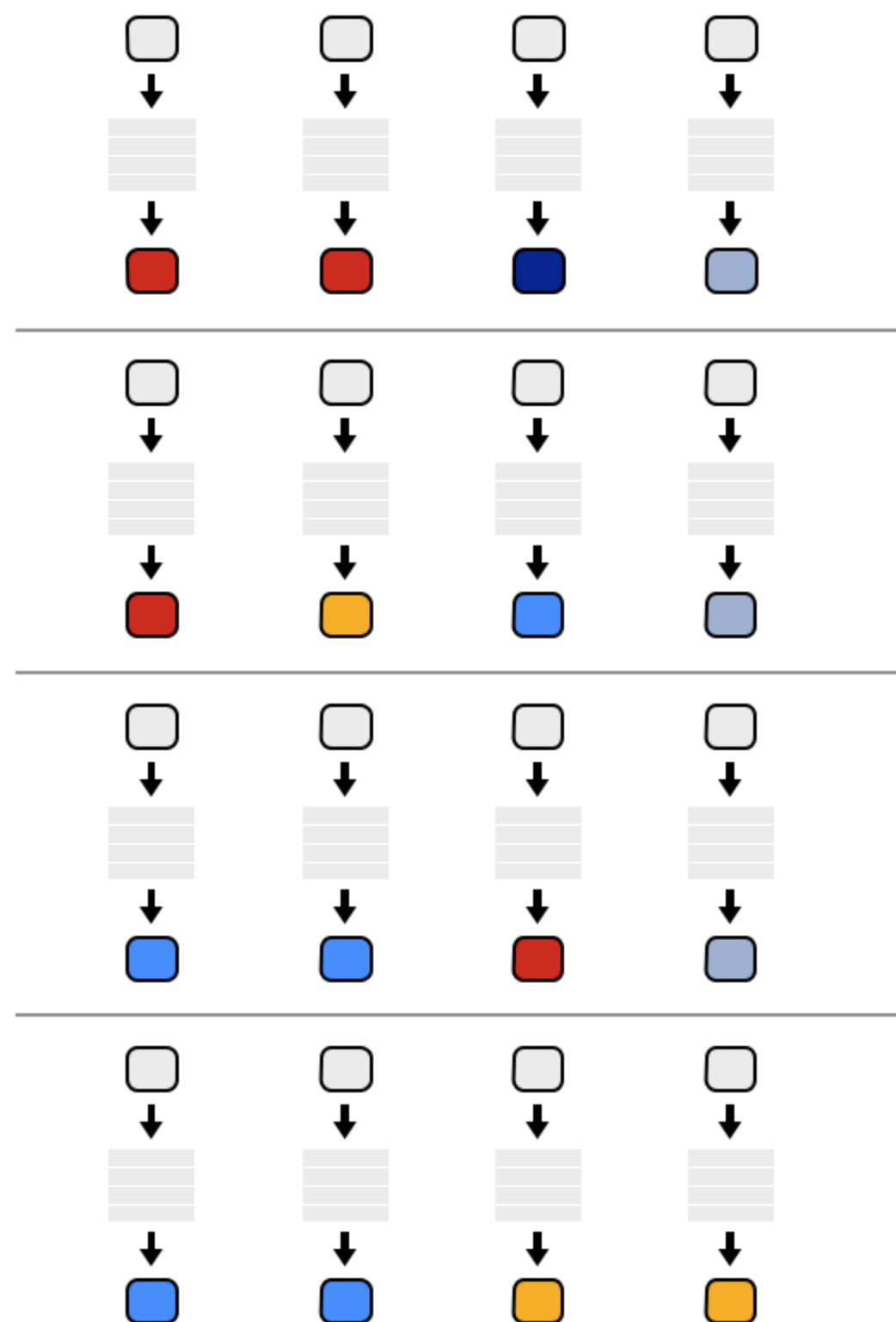
Two cores (two fragments in parallel)



Four cores (four fragments in parallel)

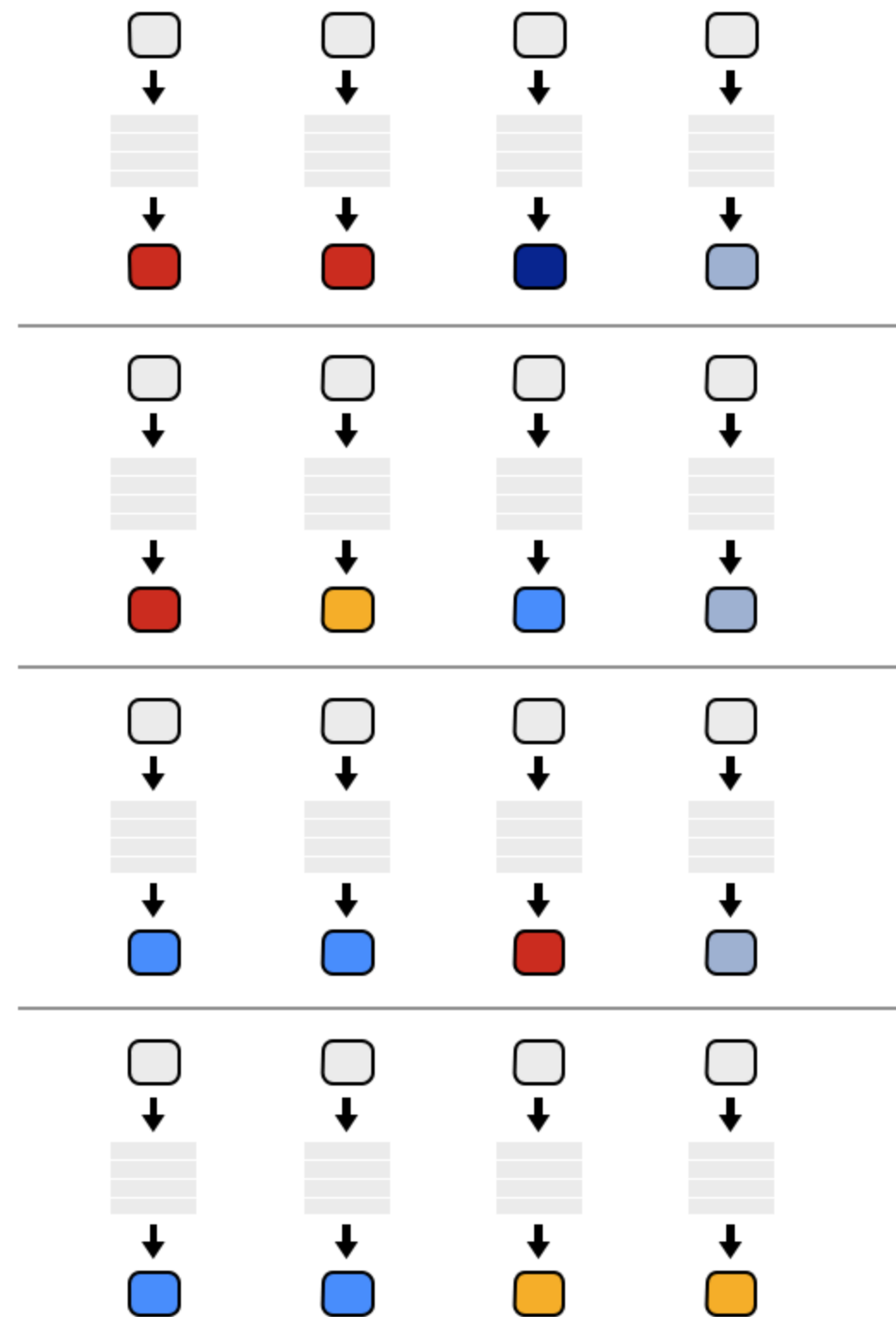


Sixteen cores (sixteen fragments in parallel)



16 cores = 16 simultaneous instruction streams

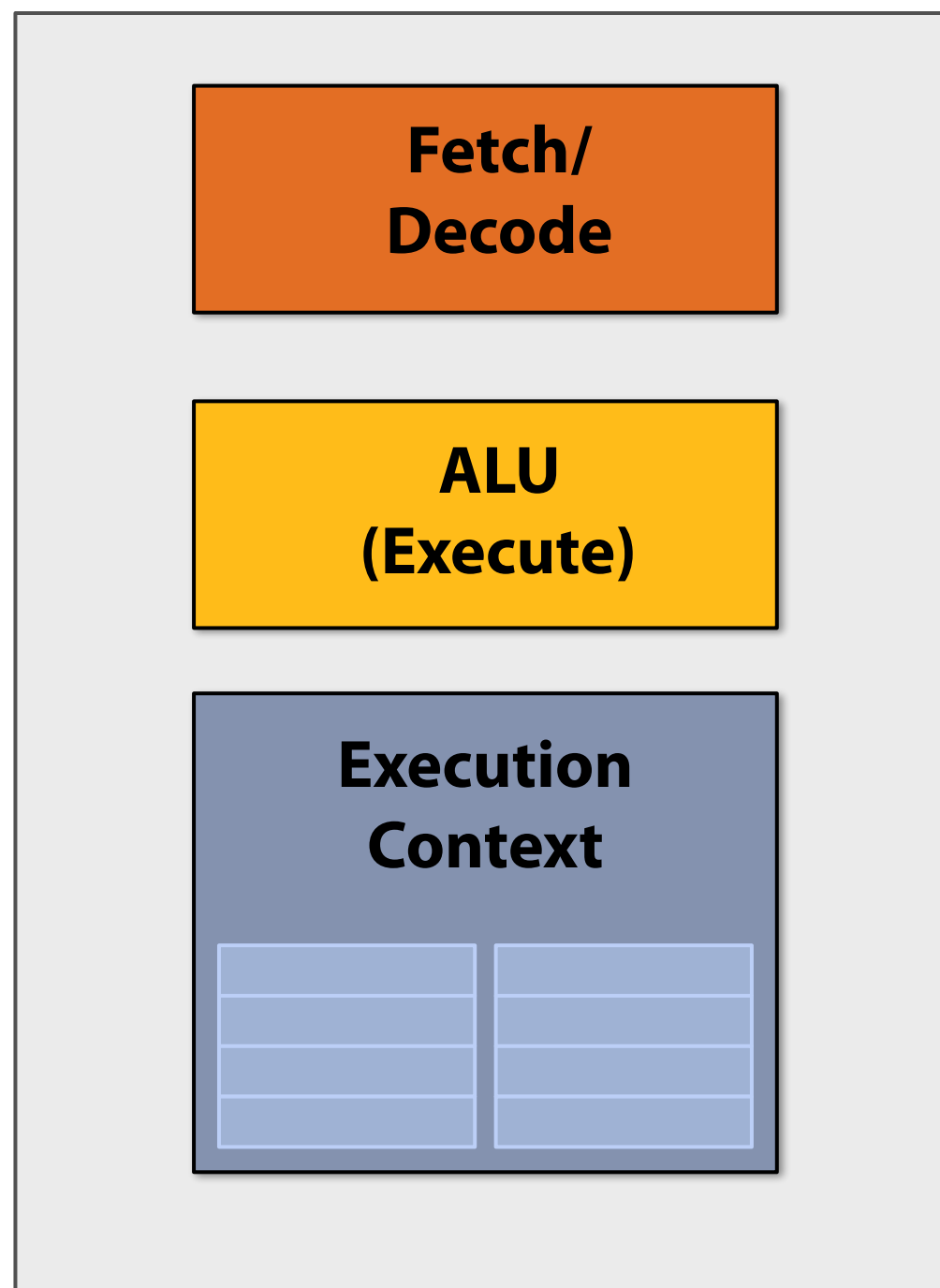
Instruction stream sharing



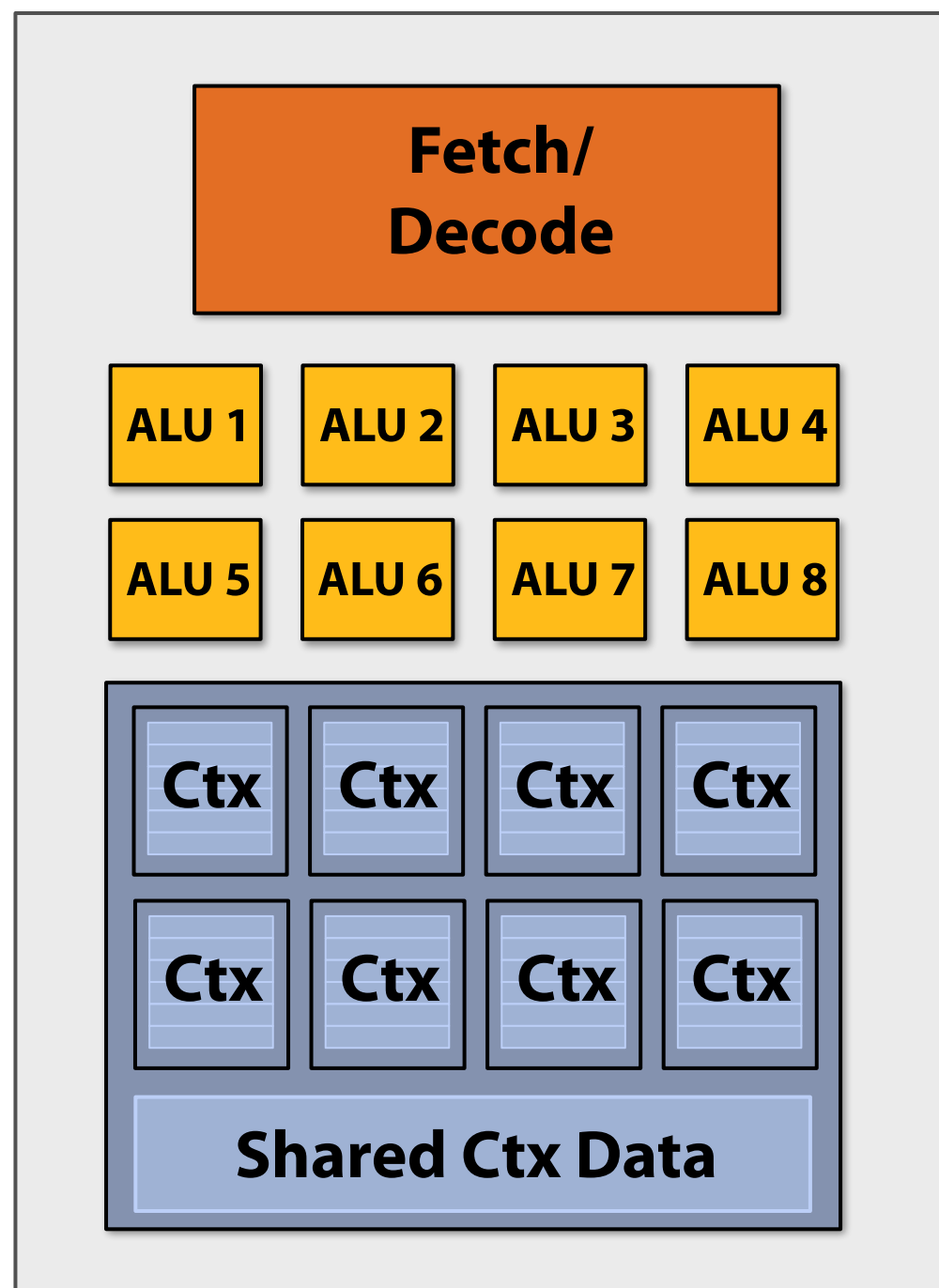
But,
many fragments should be able
to share an instruction stream!

```
<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, 1(0.0), 1(1.0)  
mul  o0, r0, r3  
mul  o1, r1, r3  
mul  o2, r2, r3  
mov  o3, 1(1.0)
```

Recall: simple processing core



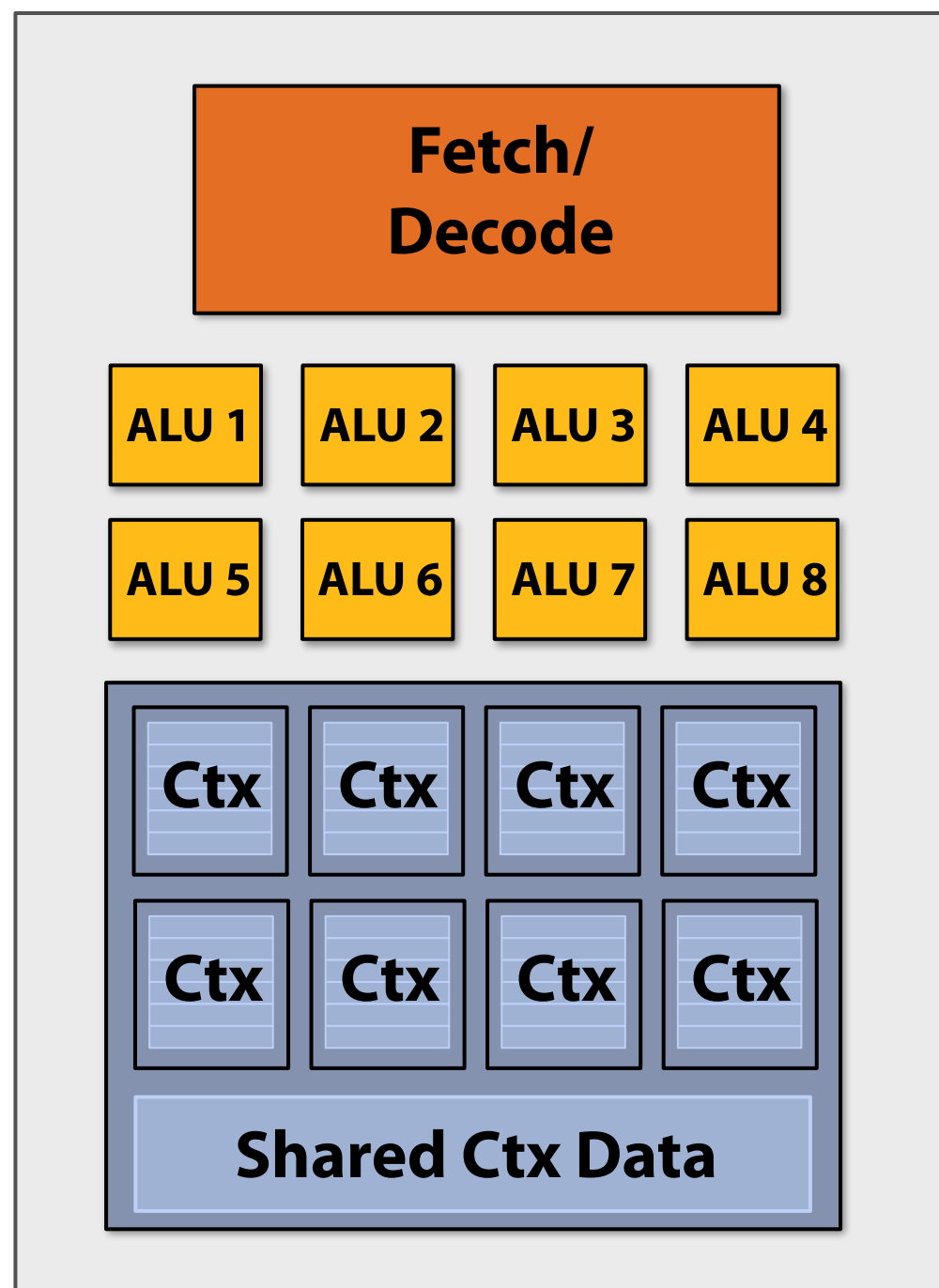
Add ALUs



Idea #2:
Amortize cost/complexity of
managing an instruction
stream across many ALUs

SIMD processing

Modifying the shader

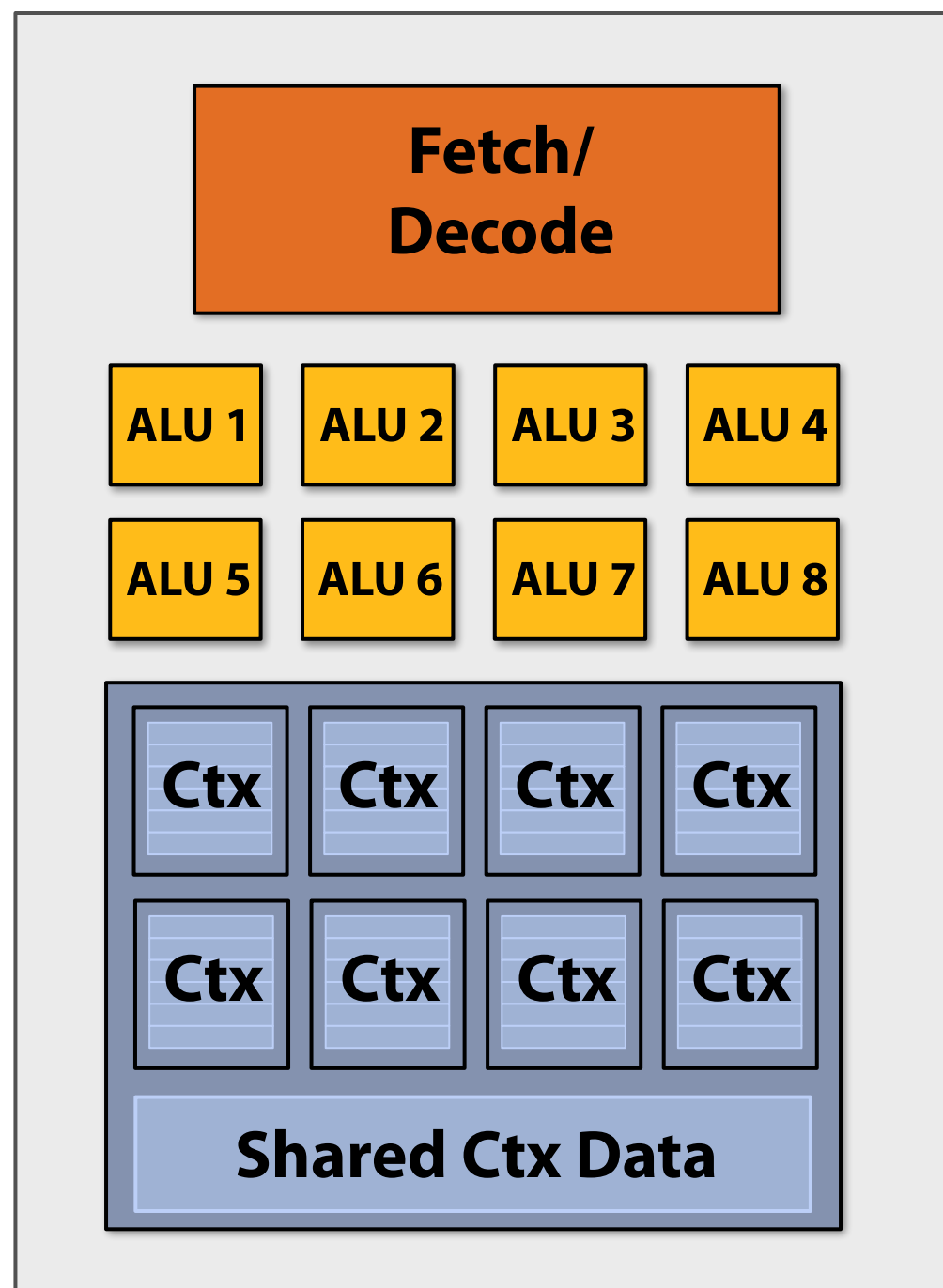


```
<diffuseShader>:  
sample r0, v4, t0, s0  
mul  r3, v0, cb0[0]  
madd r3, v1, cb0[1], r3  
madd r3, v2, cb0[2], r3  
clmp r3, r3, l(0.0), l(1.0)  
mul  o0, r0, r3  
mul  o1, r1, r3  
mul  o2, r2, r3  
mov  o3, l(1.0)
```

Original compiled shader:

Processes one fragment using
scalar ops on scalar registers

Modifying the shader



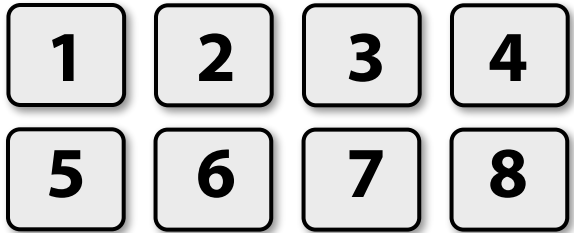
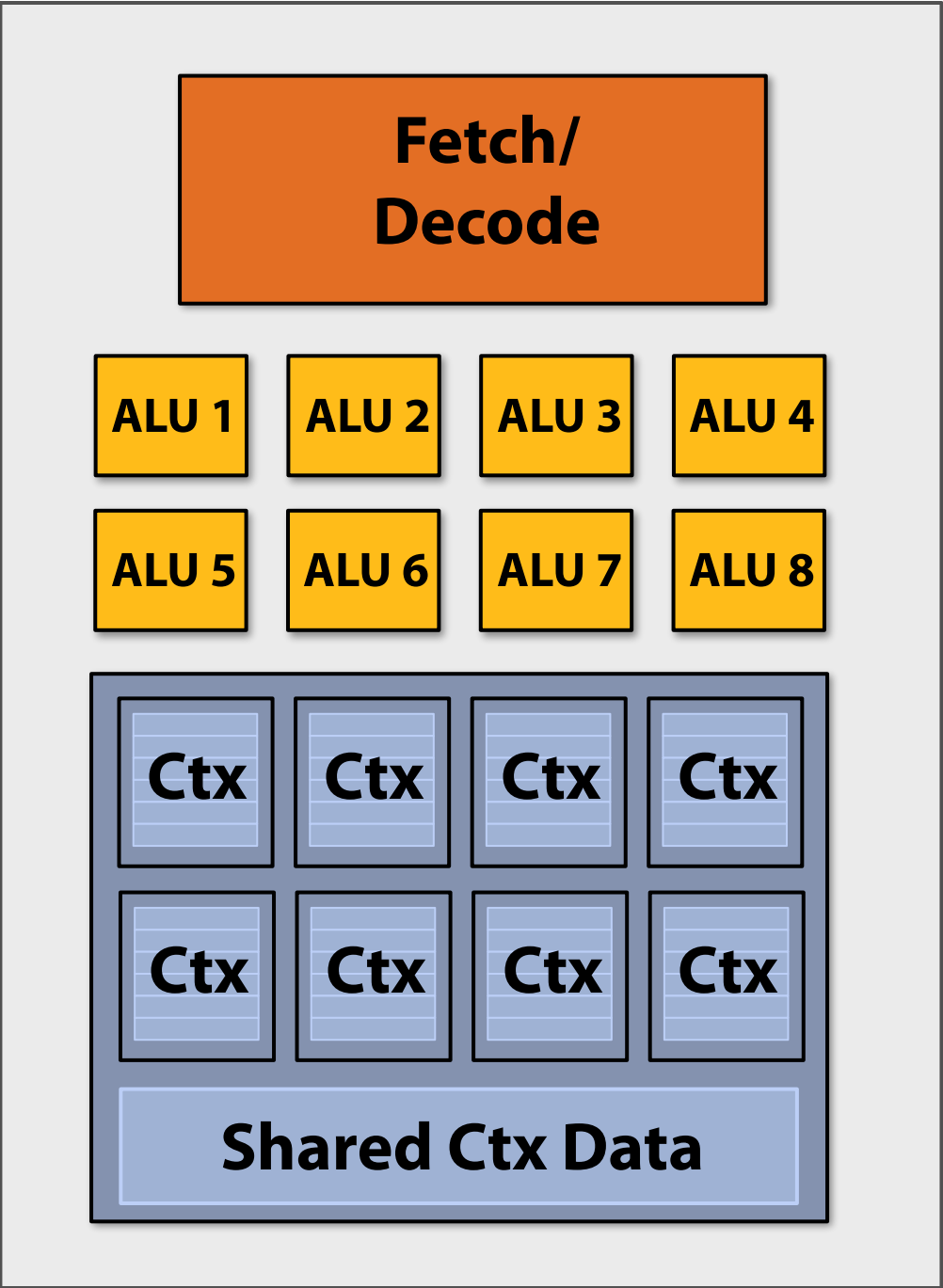
```
<VEC8_diffuseShader>:  
VEC8_sample vec_r0, vec_v4, t0, vec_s0  
VEC8_mul   vec_r3, vec_v0, cb0[0]  
VEC8_madd  vec_r3, vec_v1, cb0[1], vec_r3  
VEC8_madd  vec_r3, vec_v2, cb0[2], vec_r3  
VEC8_clmp  vec_r3, vec_r3, 1(0.0), 1(1.0)  
VEC8_mul   vec_o0, vec_r0, vec_r3  
VEC8_mul   vec_o1, vec_r1, vec_r3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

New compiled shader:

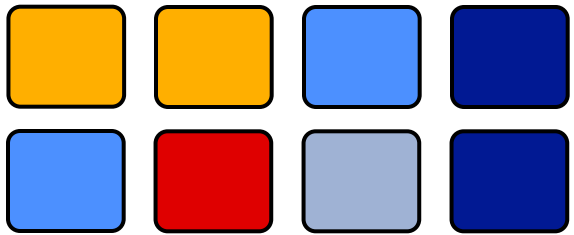
**Processes eight fragments using
vector ops on vector registers**

D

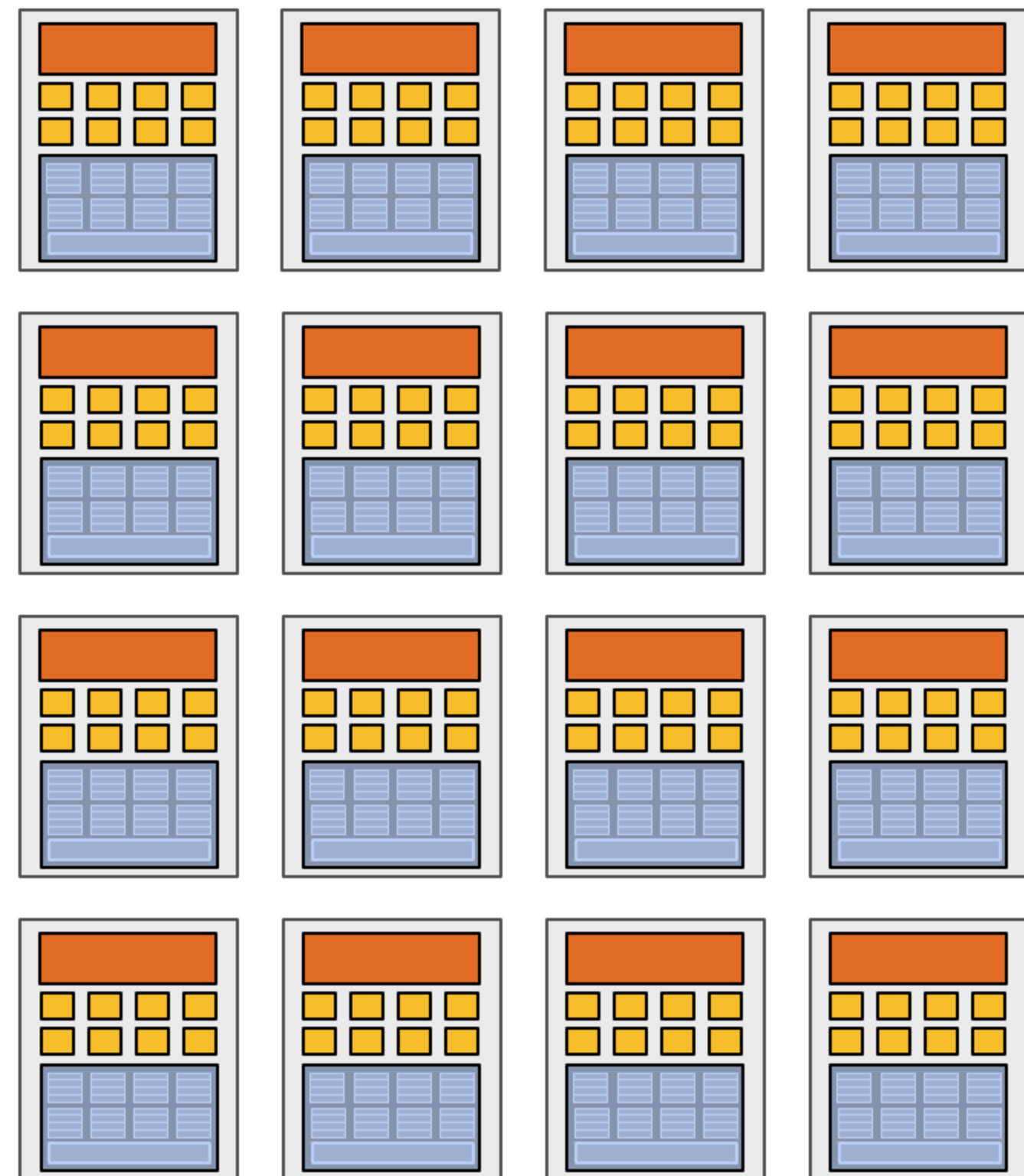
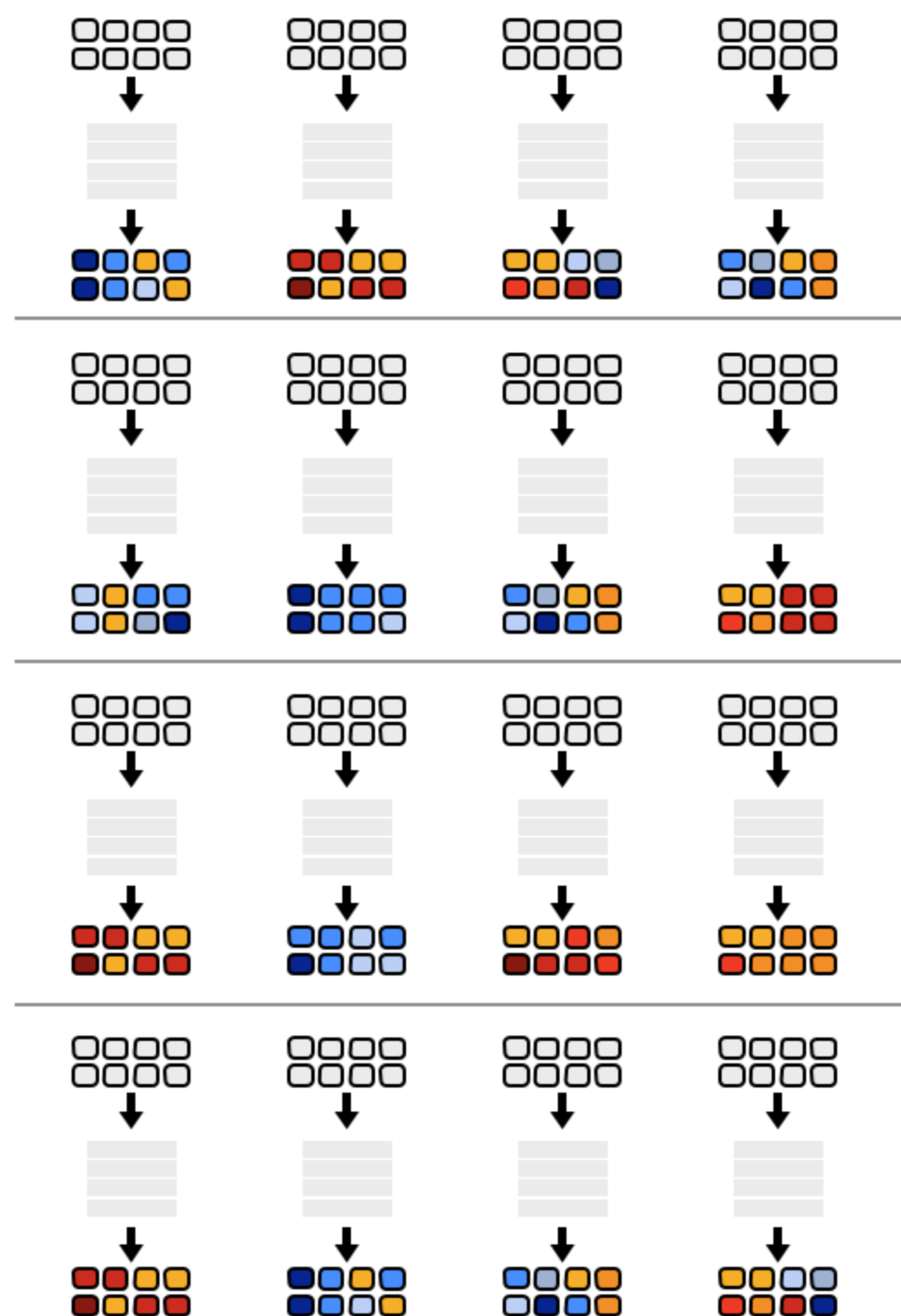
Modifying the shader



```
<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)
```

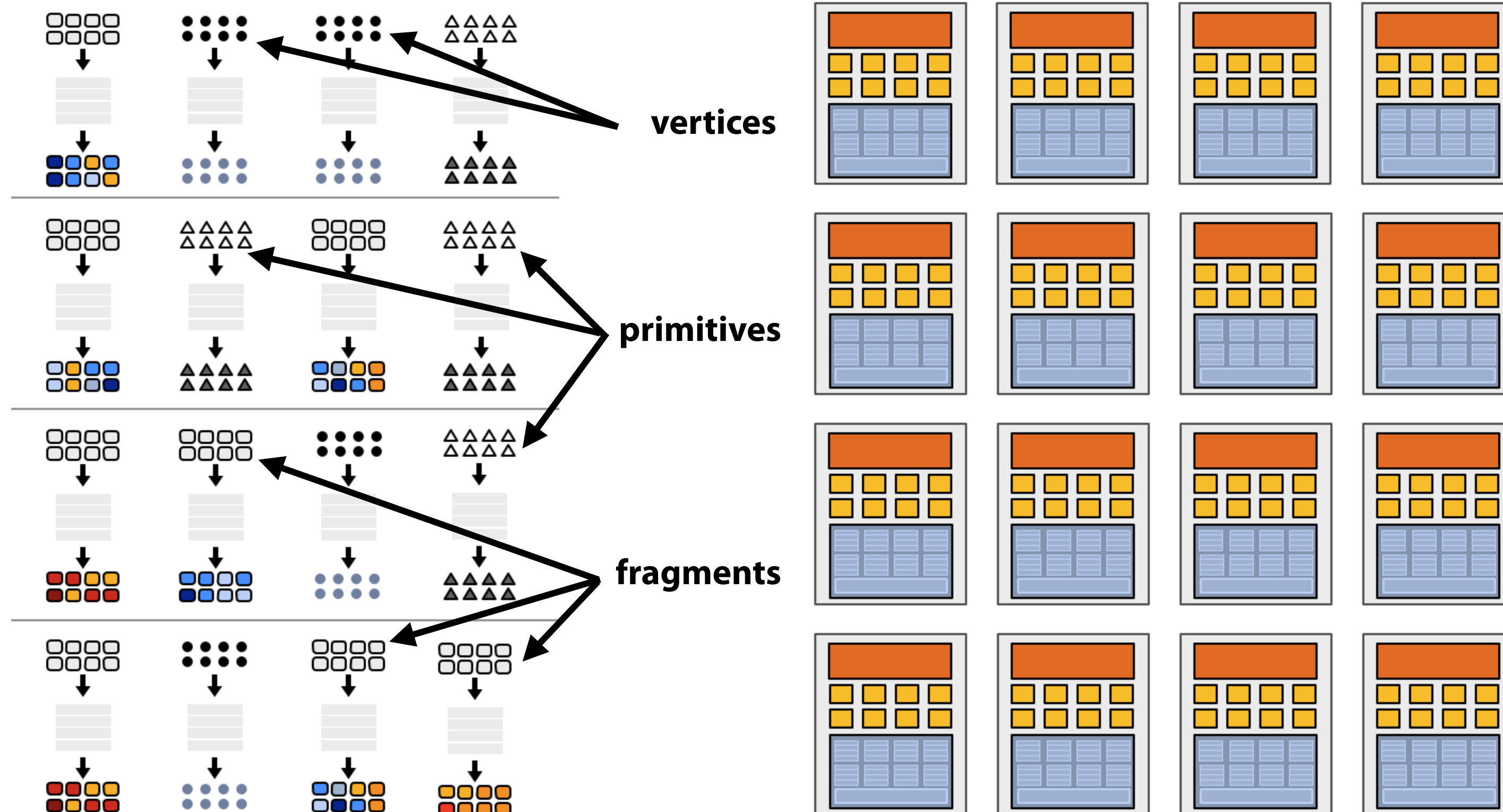


128 fragments in parallel

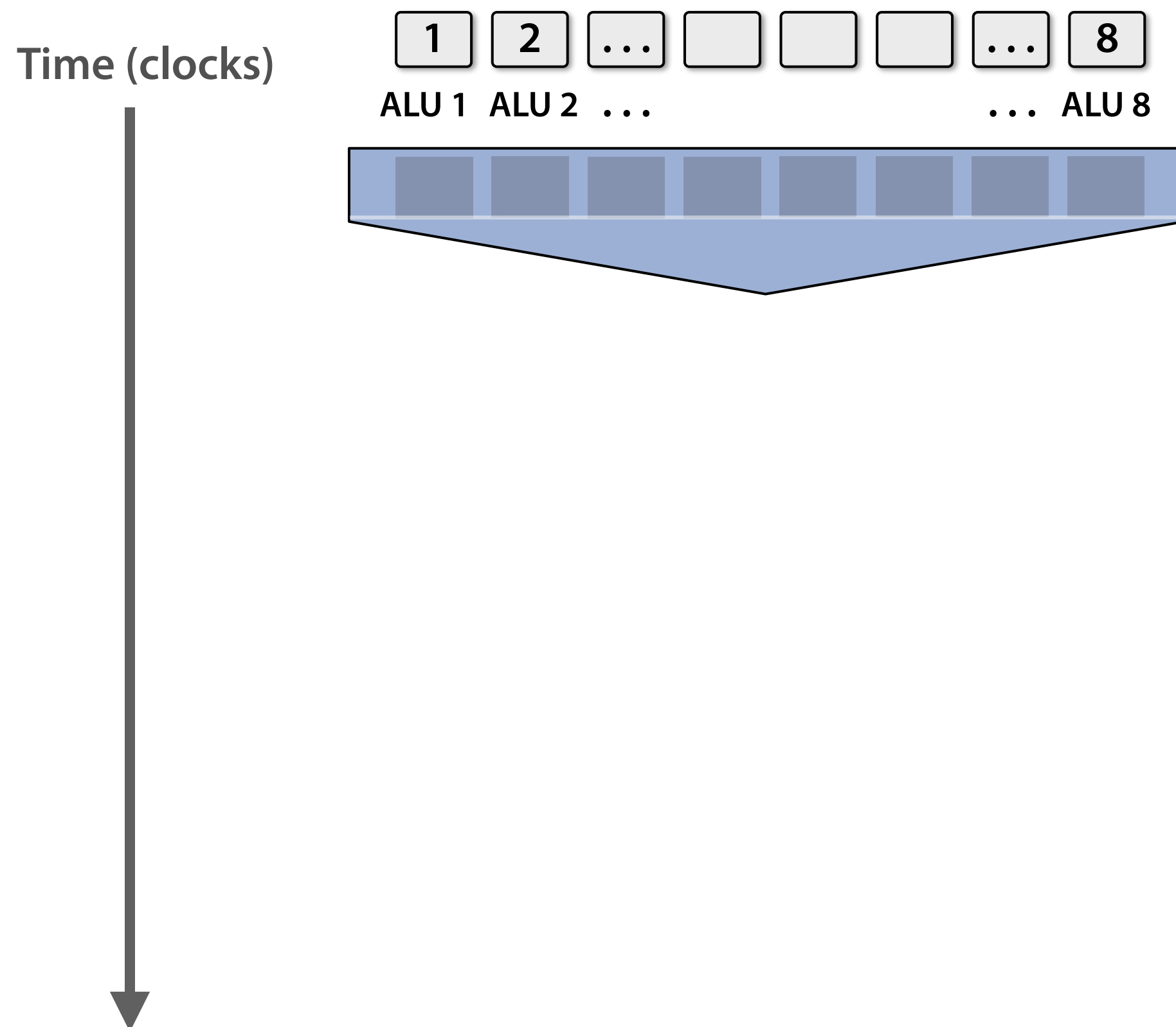


16 cores = 128 ALUs, 16 simultaneous instruction streams

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

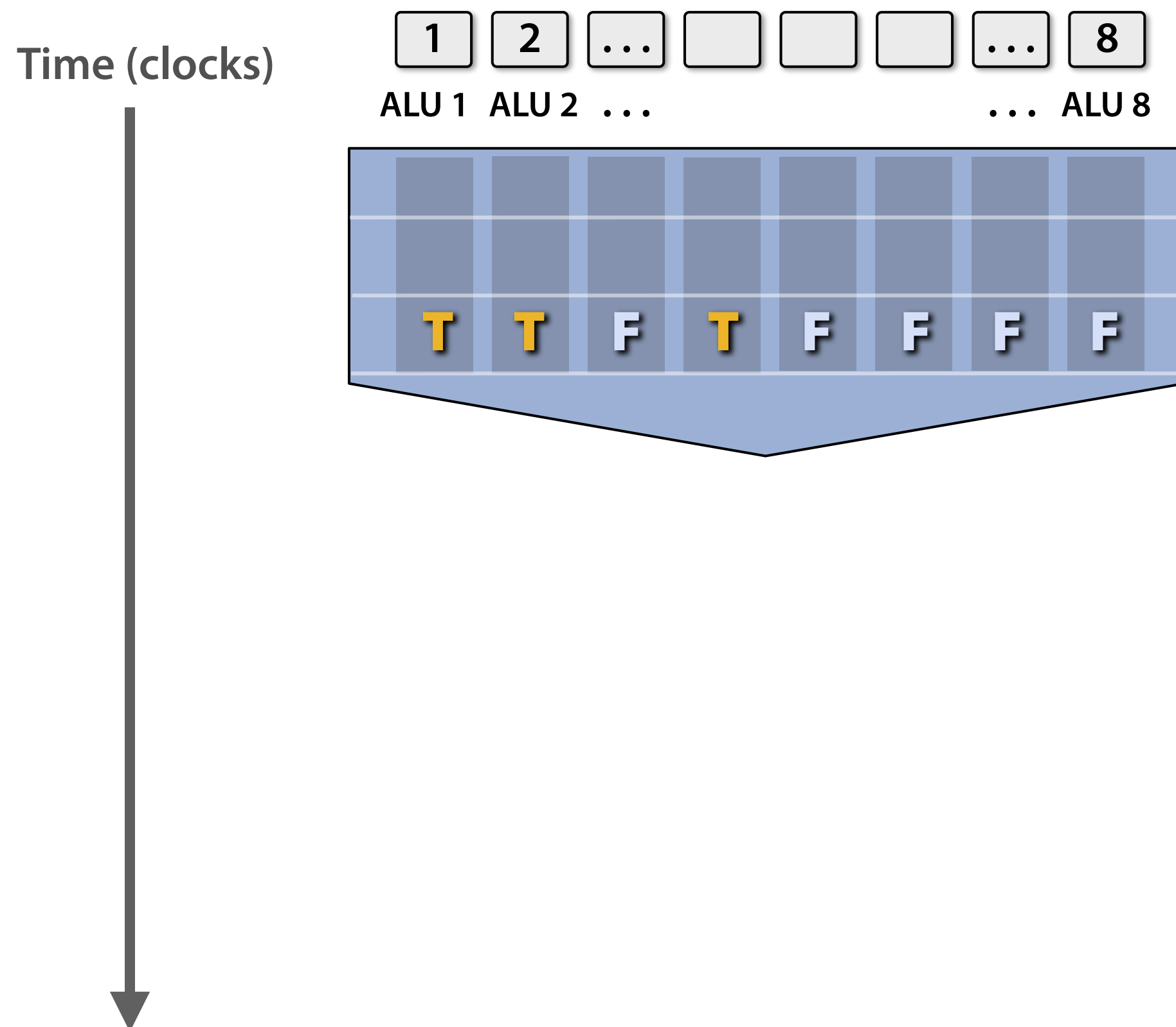


But what about branches?



```
<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?

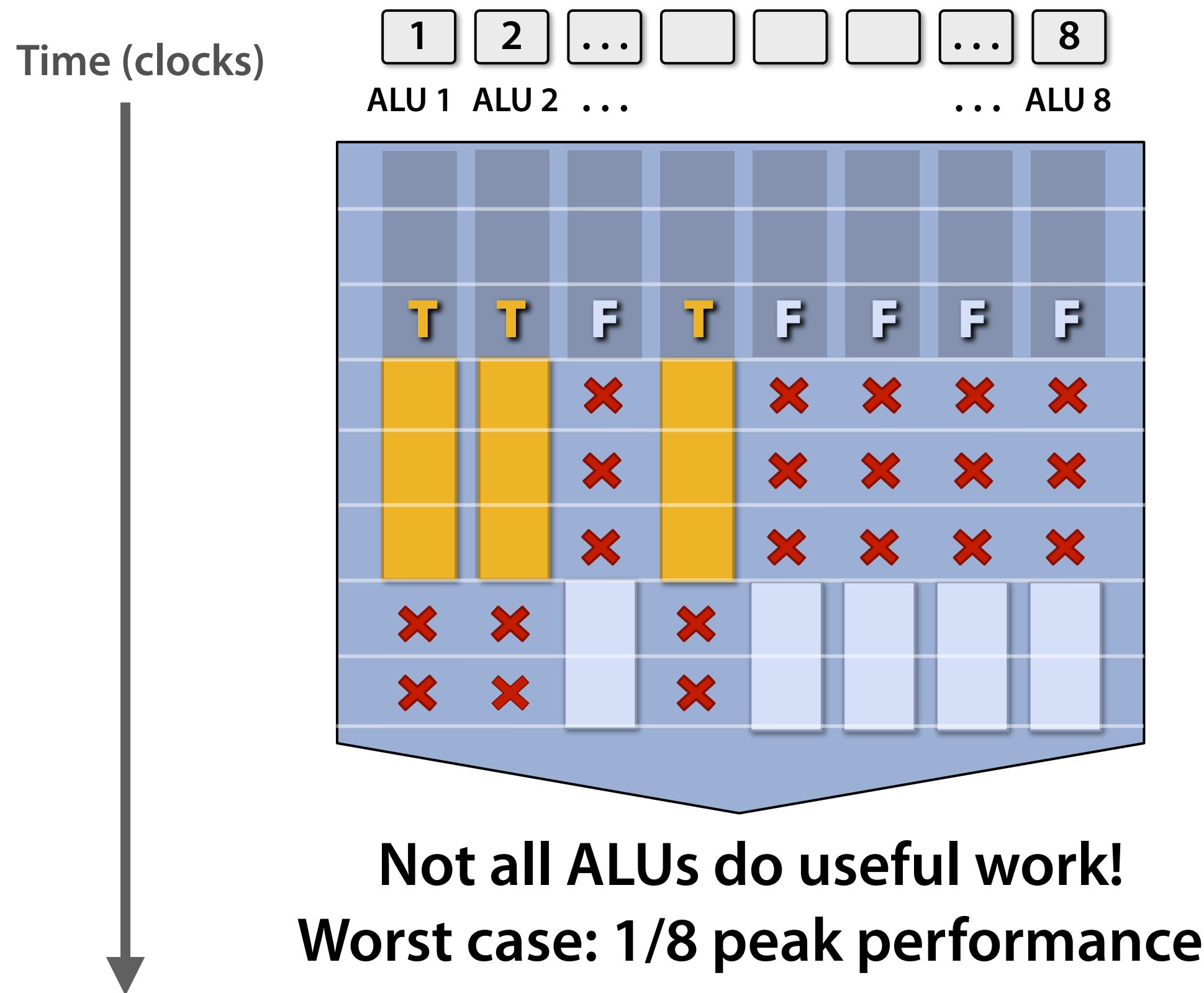


<unconditional
shader code>

```
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?



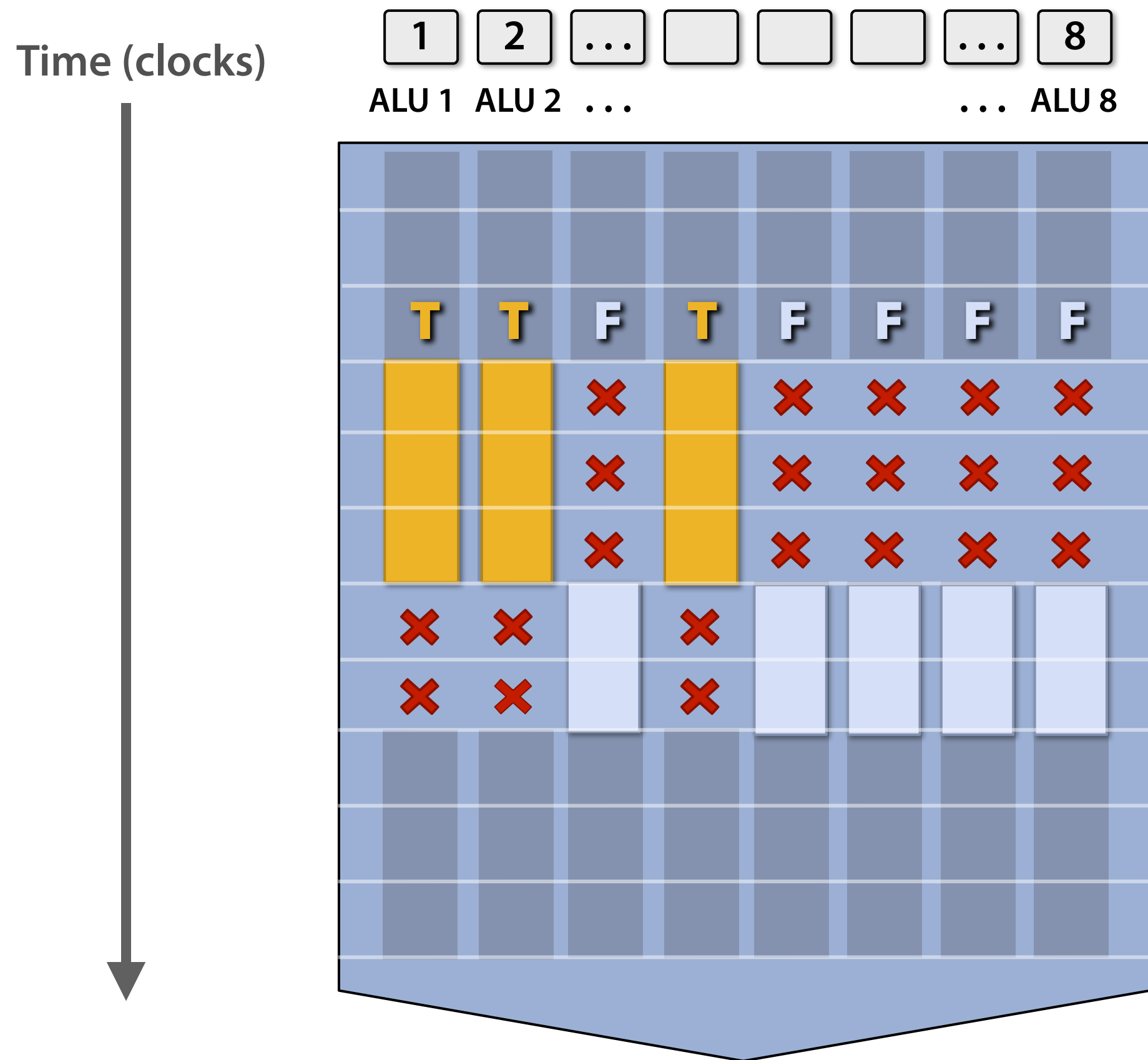
```

<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

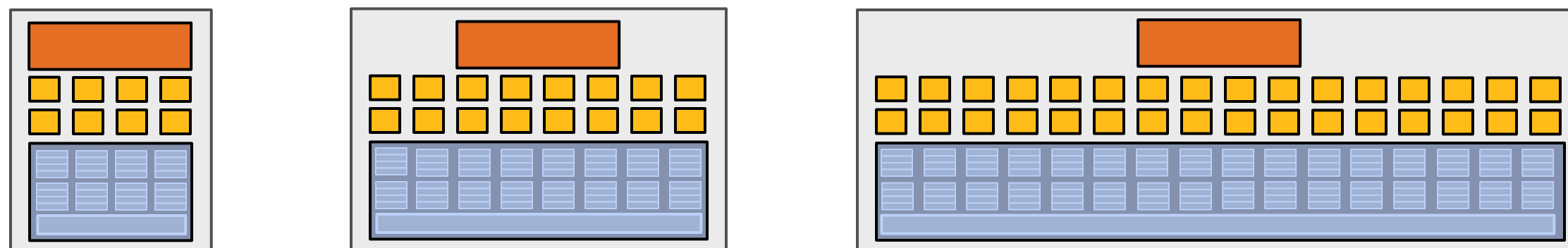
Option 1: explicit vector instructions

x86 SSE, 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

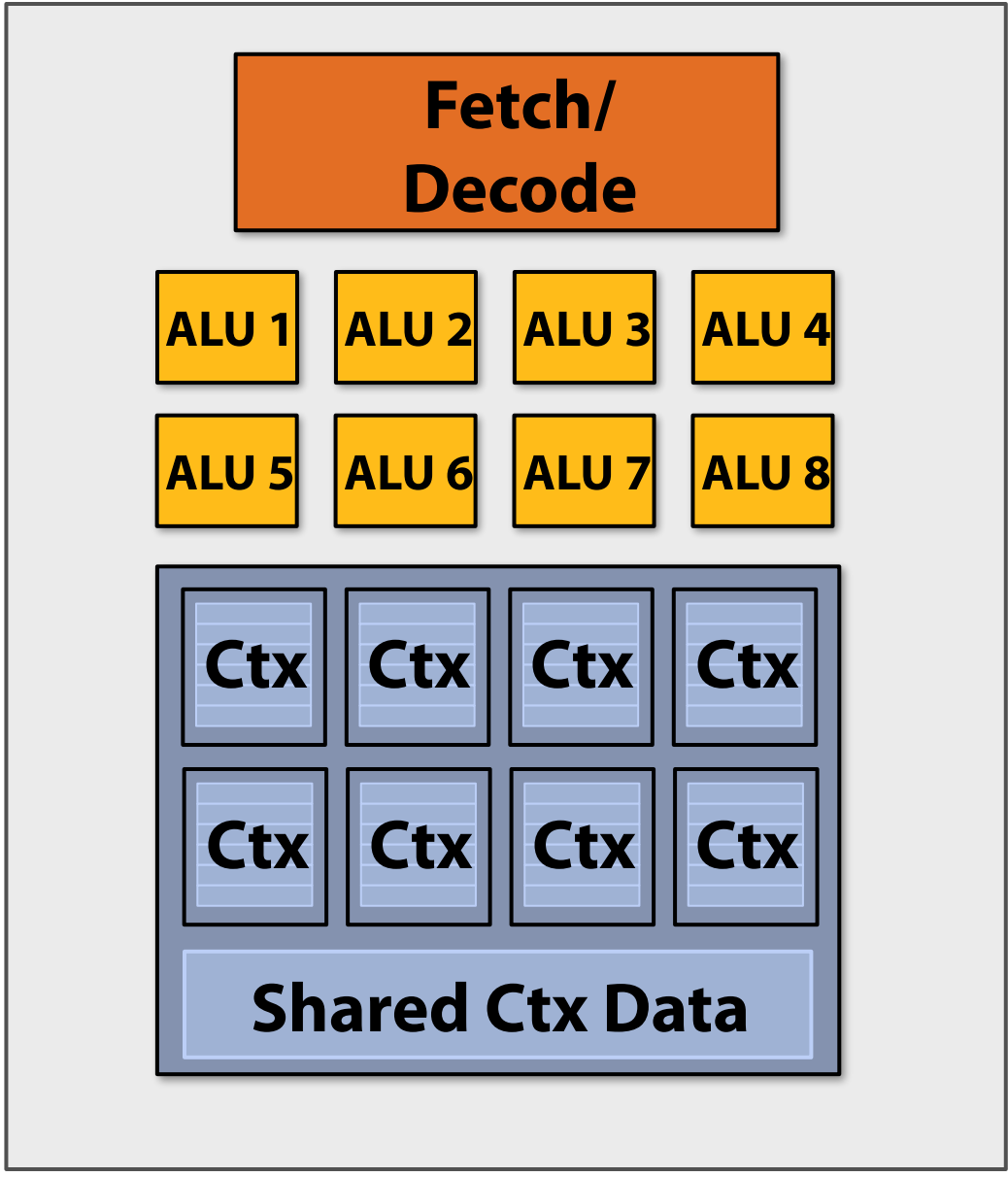
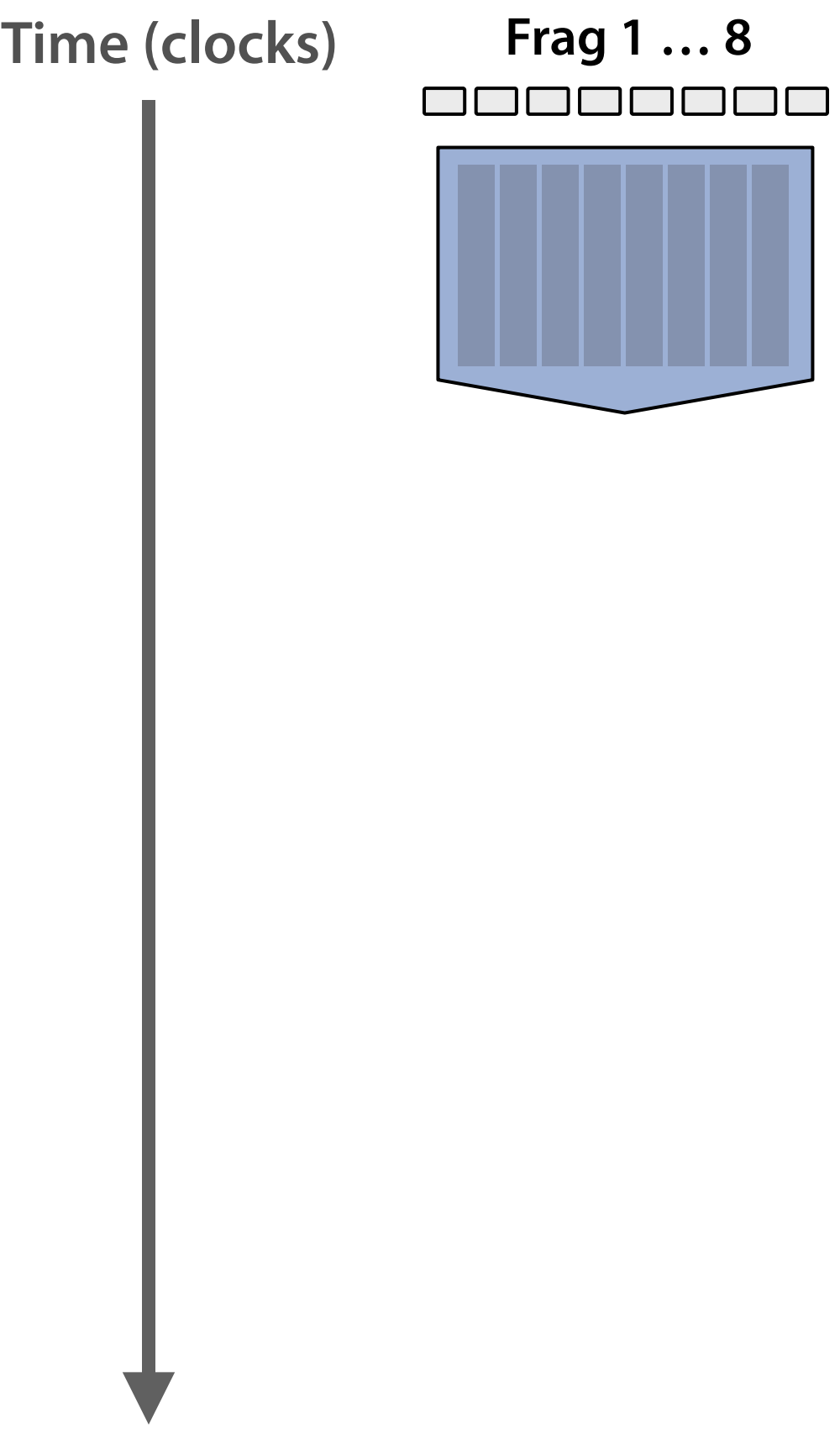
We've removed the fancy caches and logic that helps avoid stalls.

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

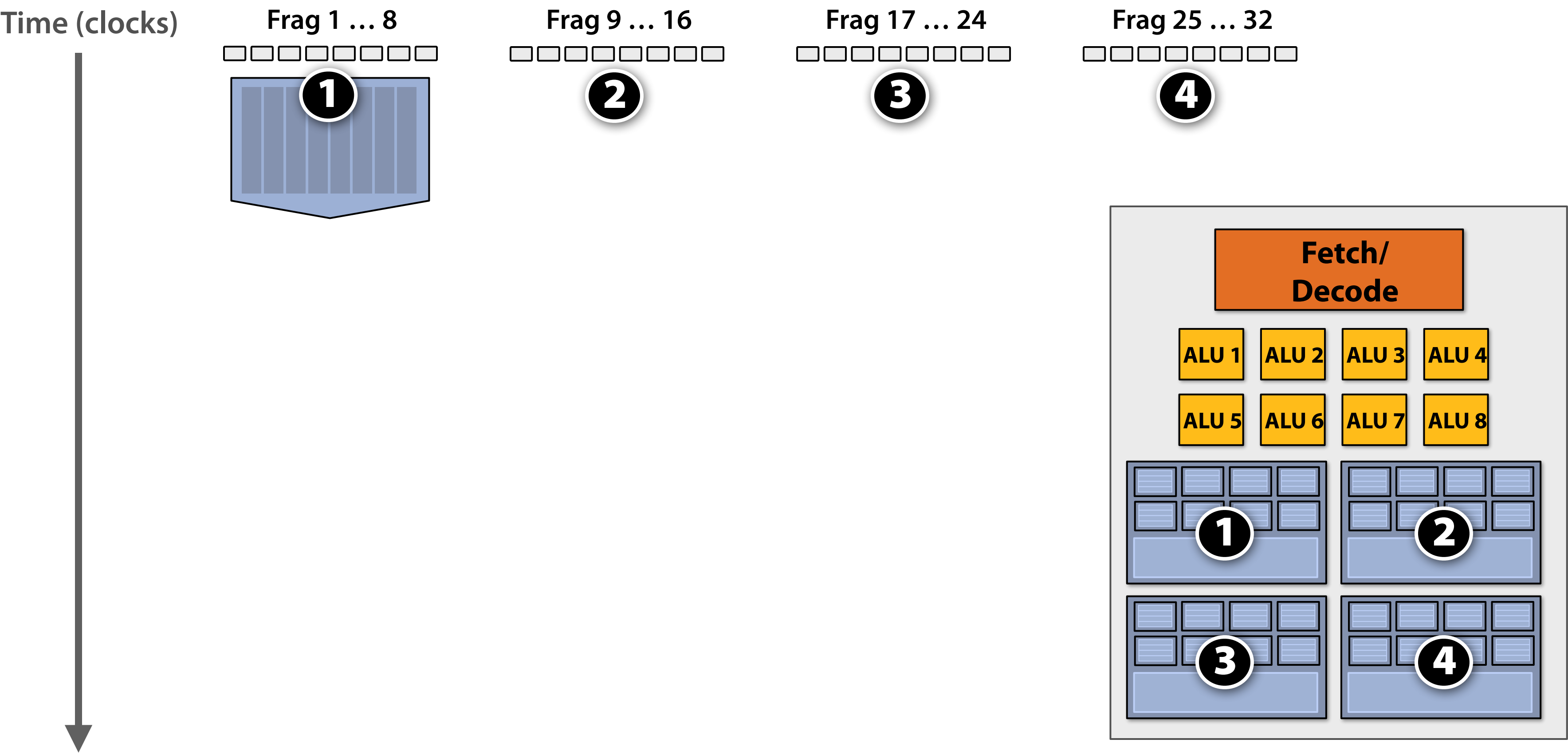
Idea #3:

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

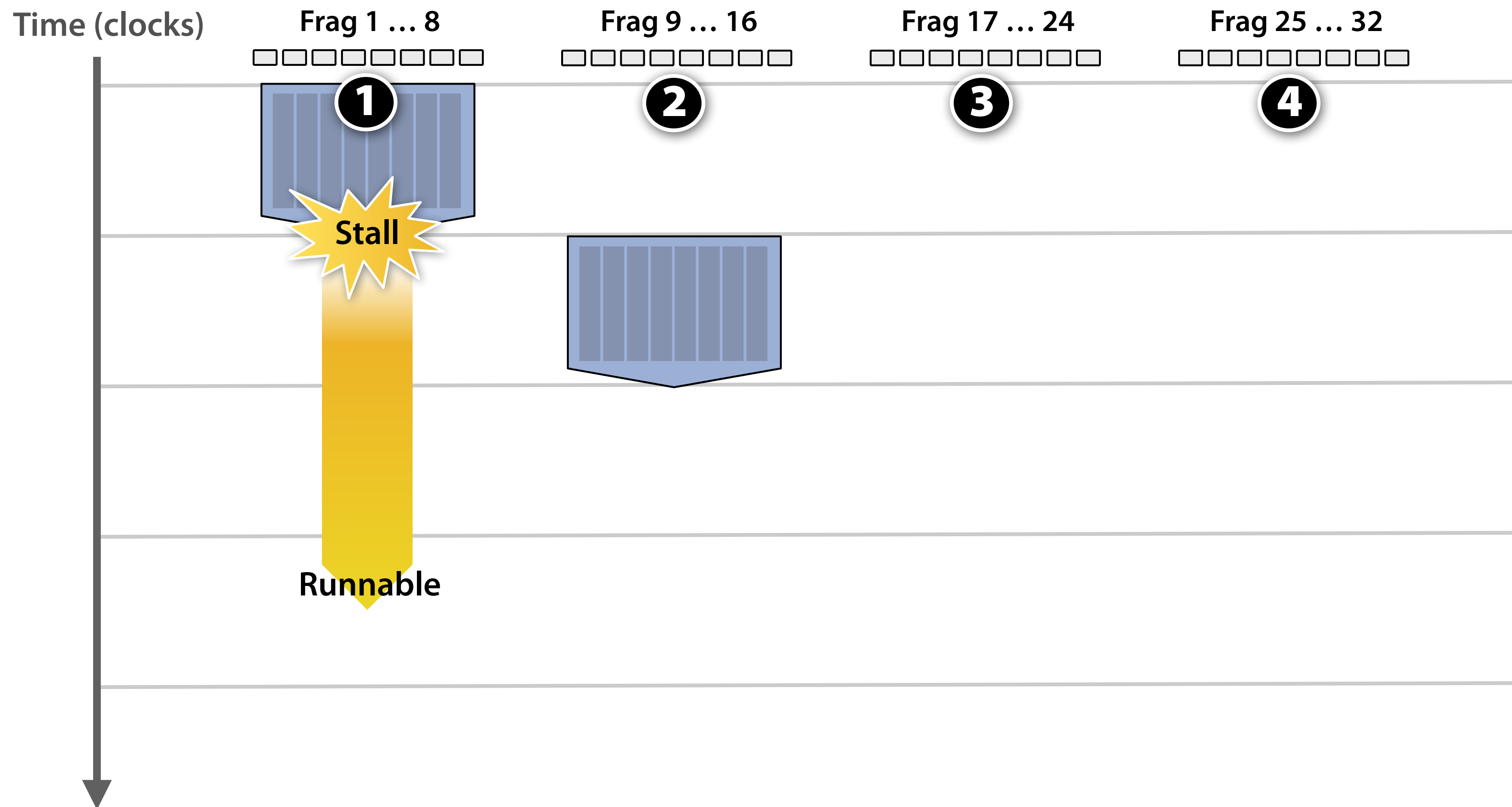
Hiding shader stalls



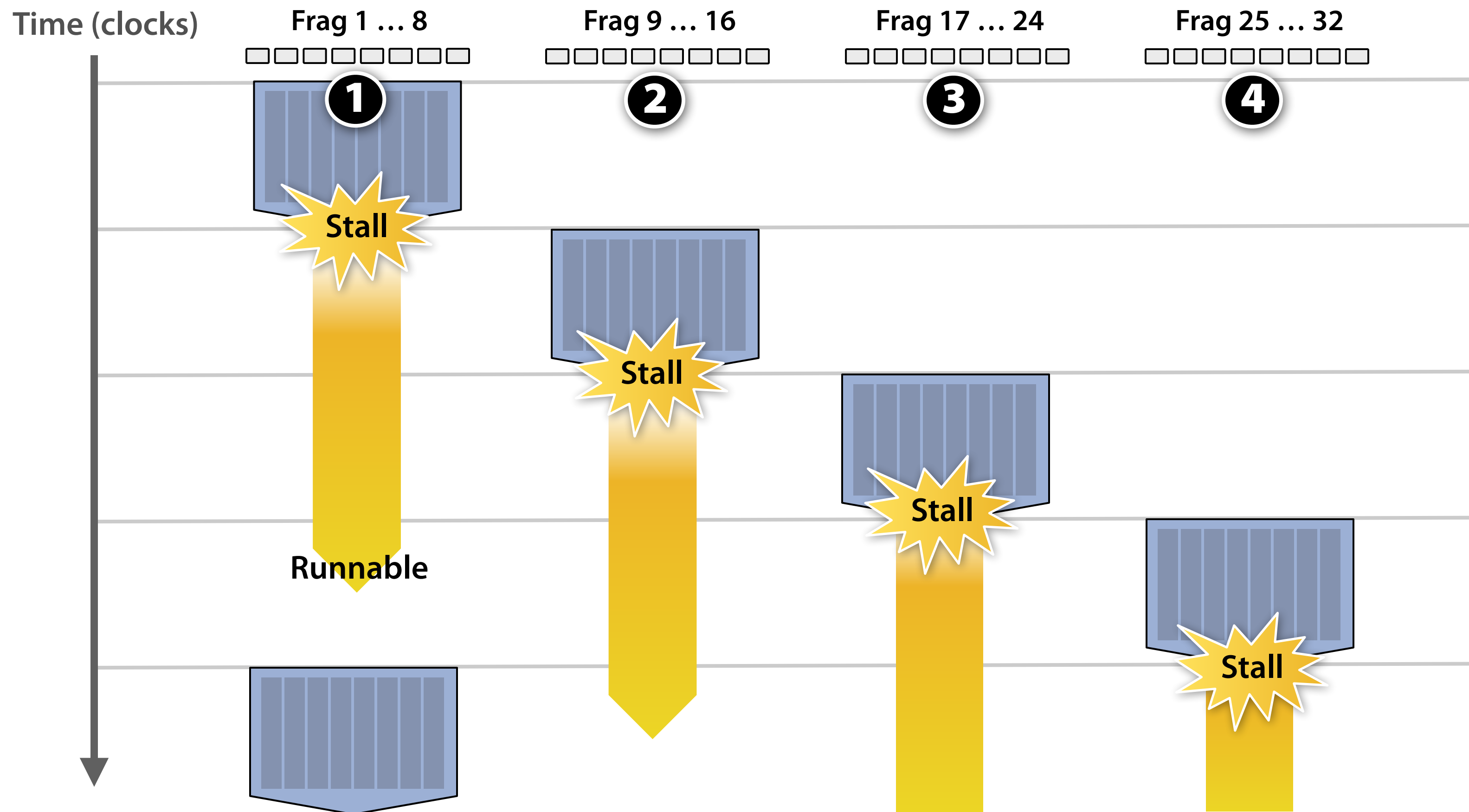
Hiding shader stalls



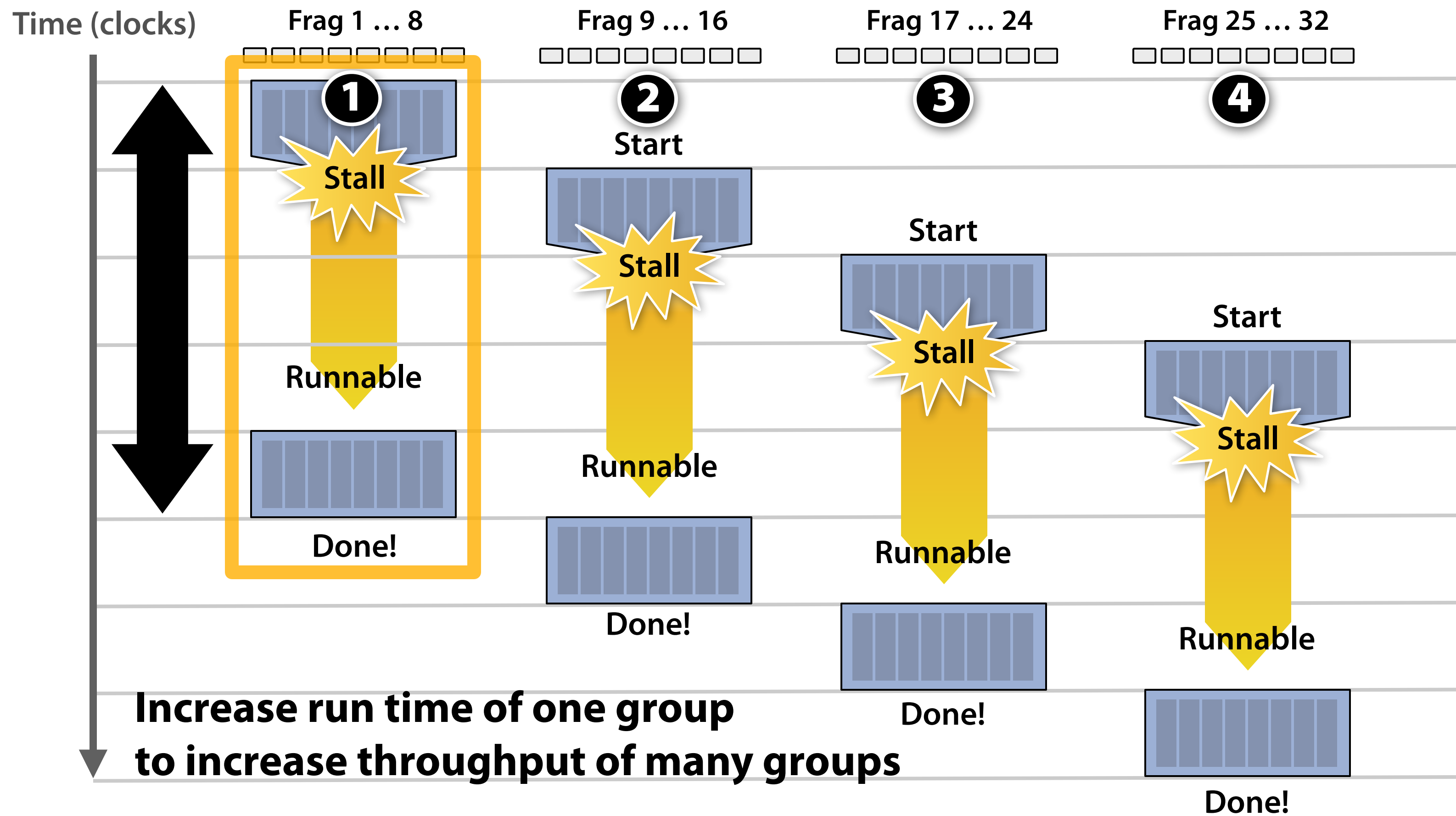
Hiding shader stalls



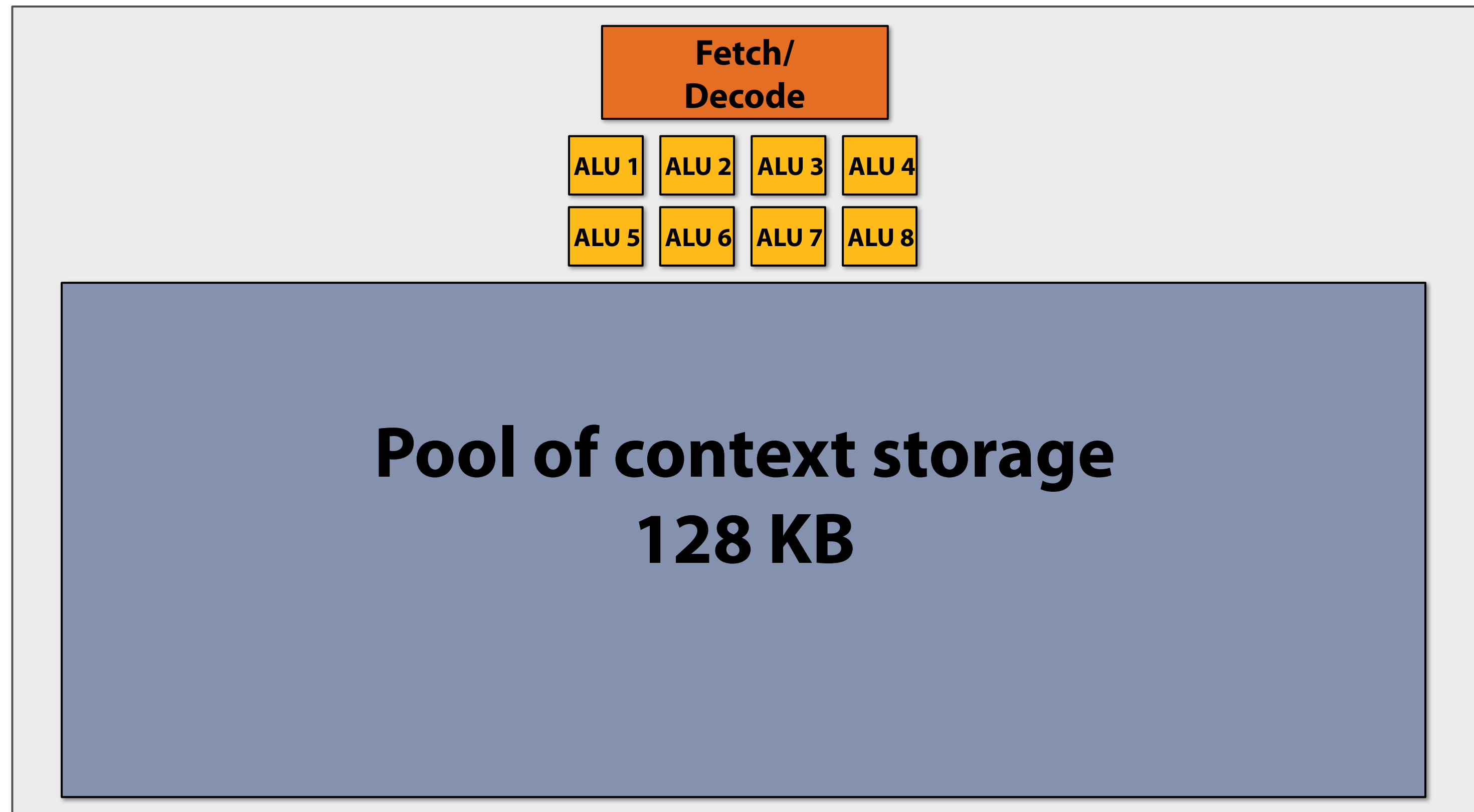
Hiding shader stalls



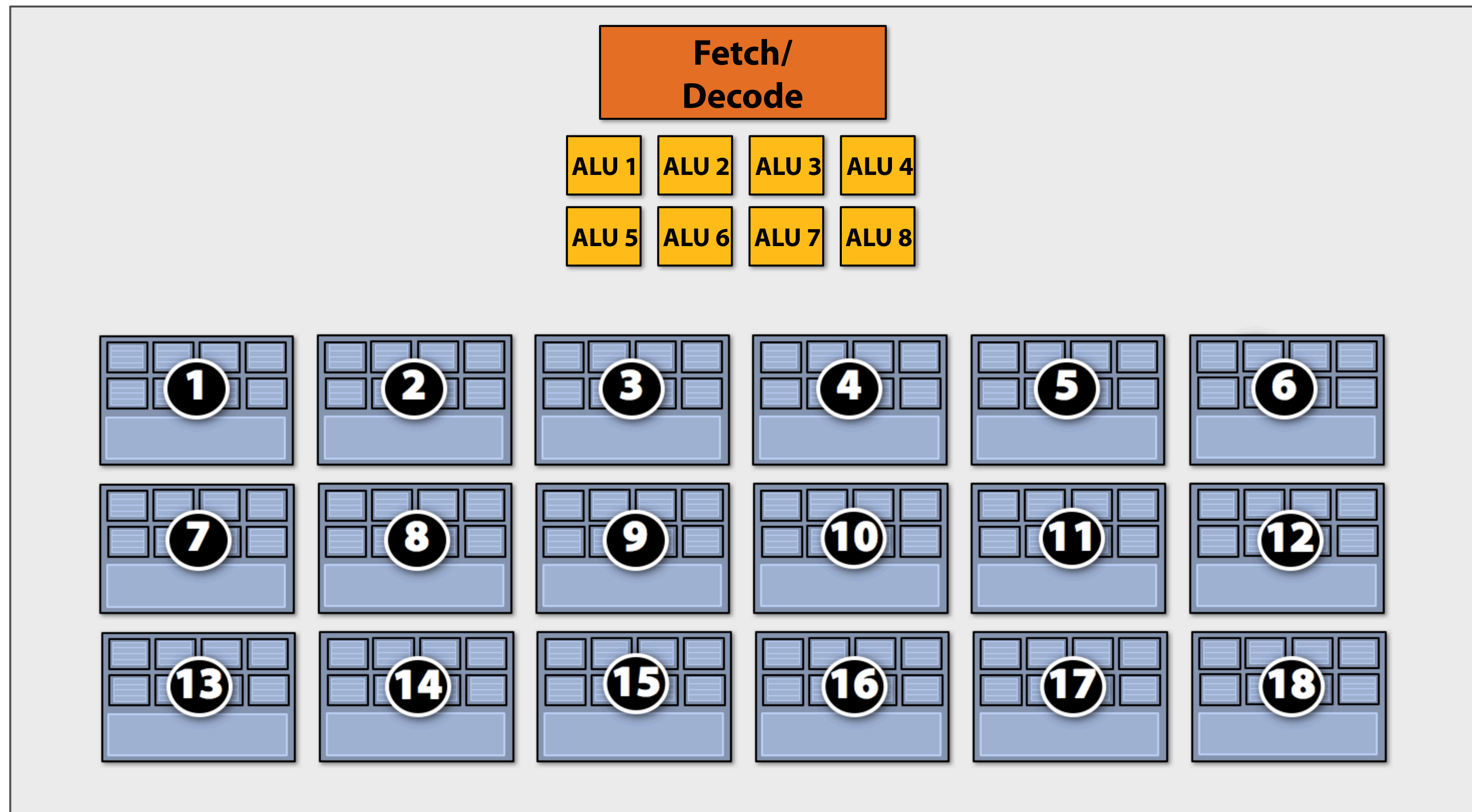
Throughput!



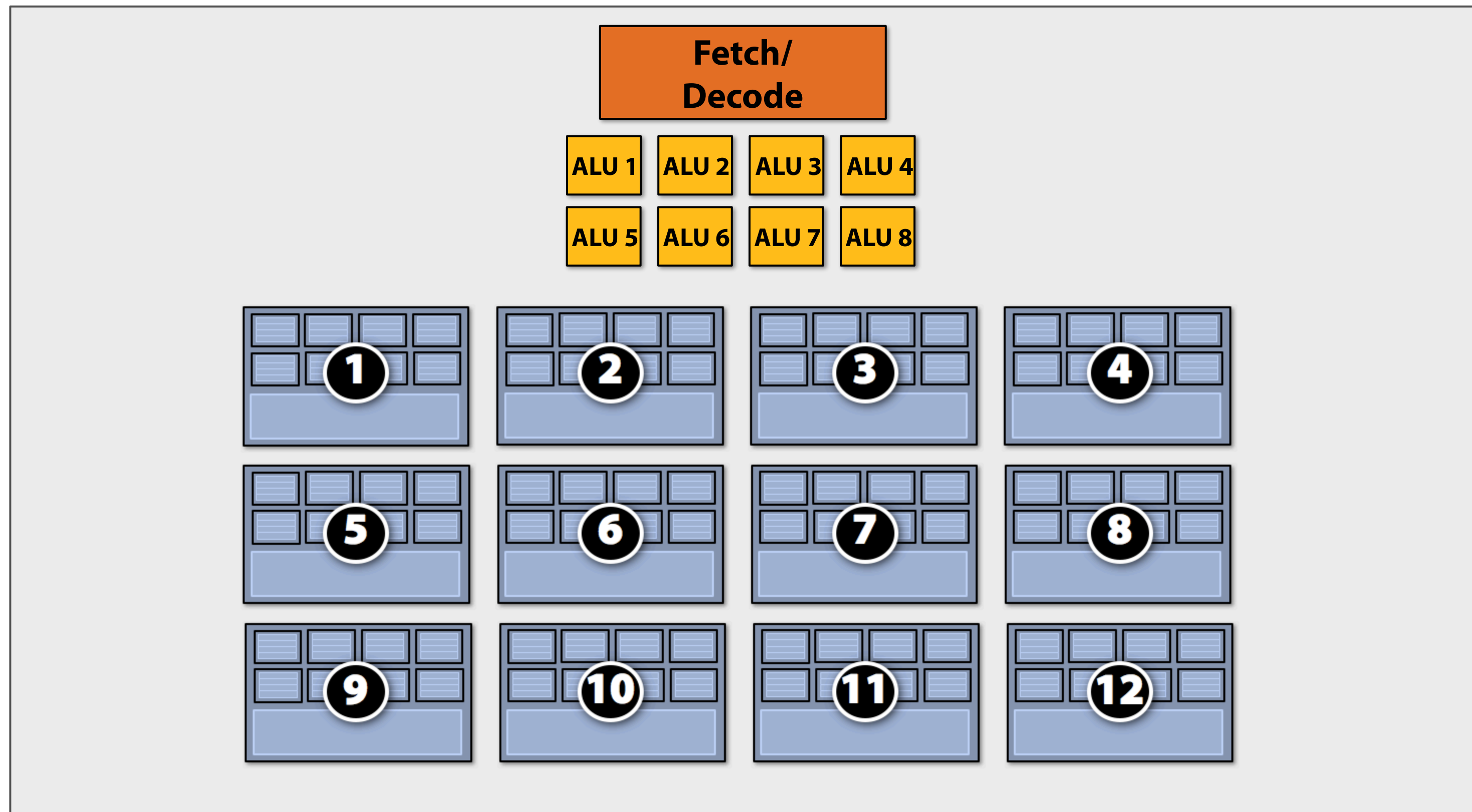
Storing contexts



Eighteen small contexts (maximal latency hiding)

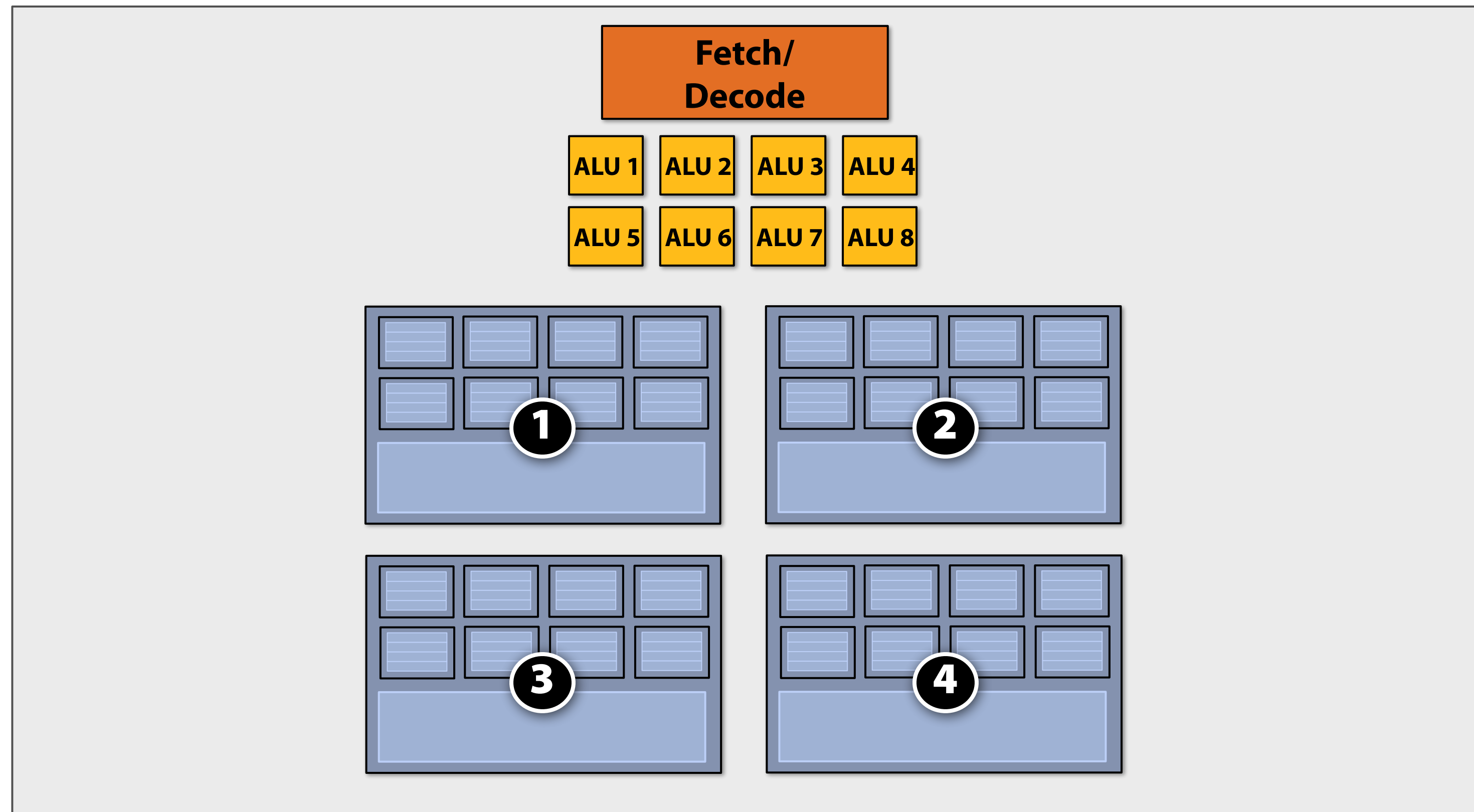


Twelve medium contexts



Four large contexts

(low latency hiding ability)



Our chip

16 cores

8 mul-add ALUs per core
(128 total)

16 simultaneous
instruction streams

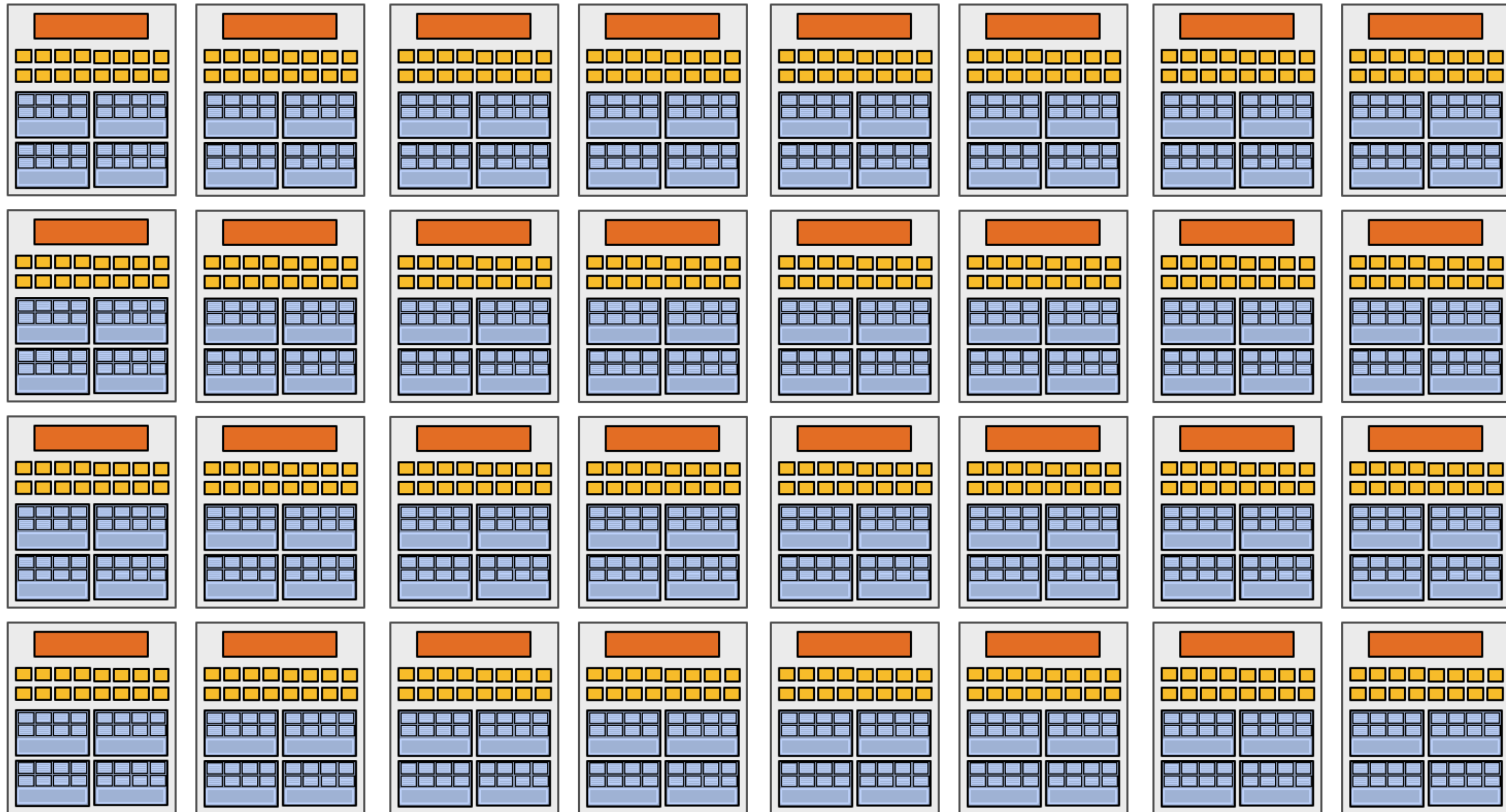
64 concurrent (but interleaved)
instruction streams

512 concurrent fragments

= 256 GFLOPs (@ 1GHz)



Our “enthusiast” chip



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

NVIDIA GeForce GTX 1080

NVIDIA-speak:

2560 stream processors (“CUDA cores”)

“SIMT execution”

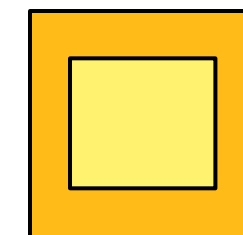
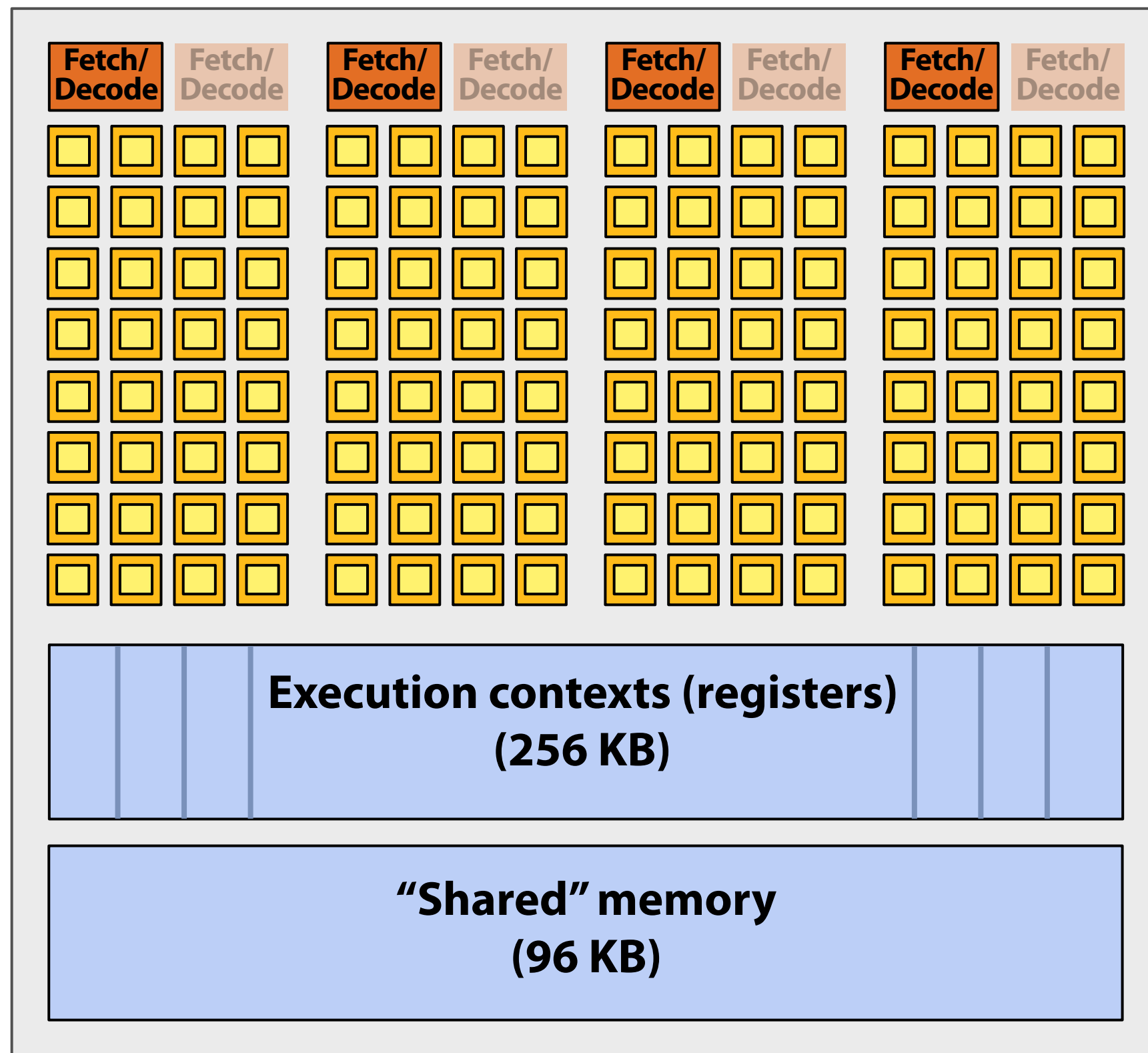
Generic speak:

20 cores

4 groups of 32 SIMD functional units per core



NVIDIA GeForce GTX 1080 “core”



= SIMD function unit,
control shared across 32 units
(1 MUL-ADD per clock)

- 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

NVIDIA GeForce GTX 1080



There are 20 of these things on the GTX 1080

That's 40,960 fragments!

(Or 40,960 "CUDA threads")

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



Pathtracing vs. Rasterization?

- Global Illumination and other rich visual effects are “free” with path tracing
 - Included by nature
- Many many years of graphics engineering work poured into techniques for simulating those effects
 - Reflections, shadows, etc.
 - “Baked” maps (light maps)
 - Screen-space effects
-

Why is pathtracing “hard” for GPUs?

- Recursion is hard (if not basically impossible)
 - Also, variable # of bounces is bad for SIMD
 - Could implement as iterative — fixed # of bounces
- Memory coherency is basically non-existent
 - You cast a ray into the scene —> it can hit basically anything, bounce basically anywhere —> it can hit basically anything, ...
- When rasterizing triangles you know exact information about the triangle you’re trying to shade
 - From one bounce to the next, no way to know which shader should be called next

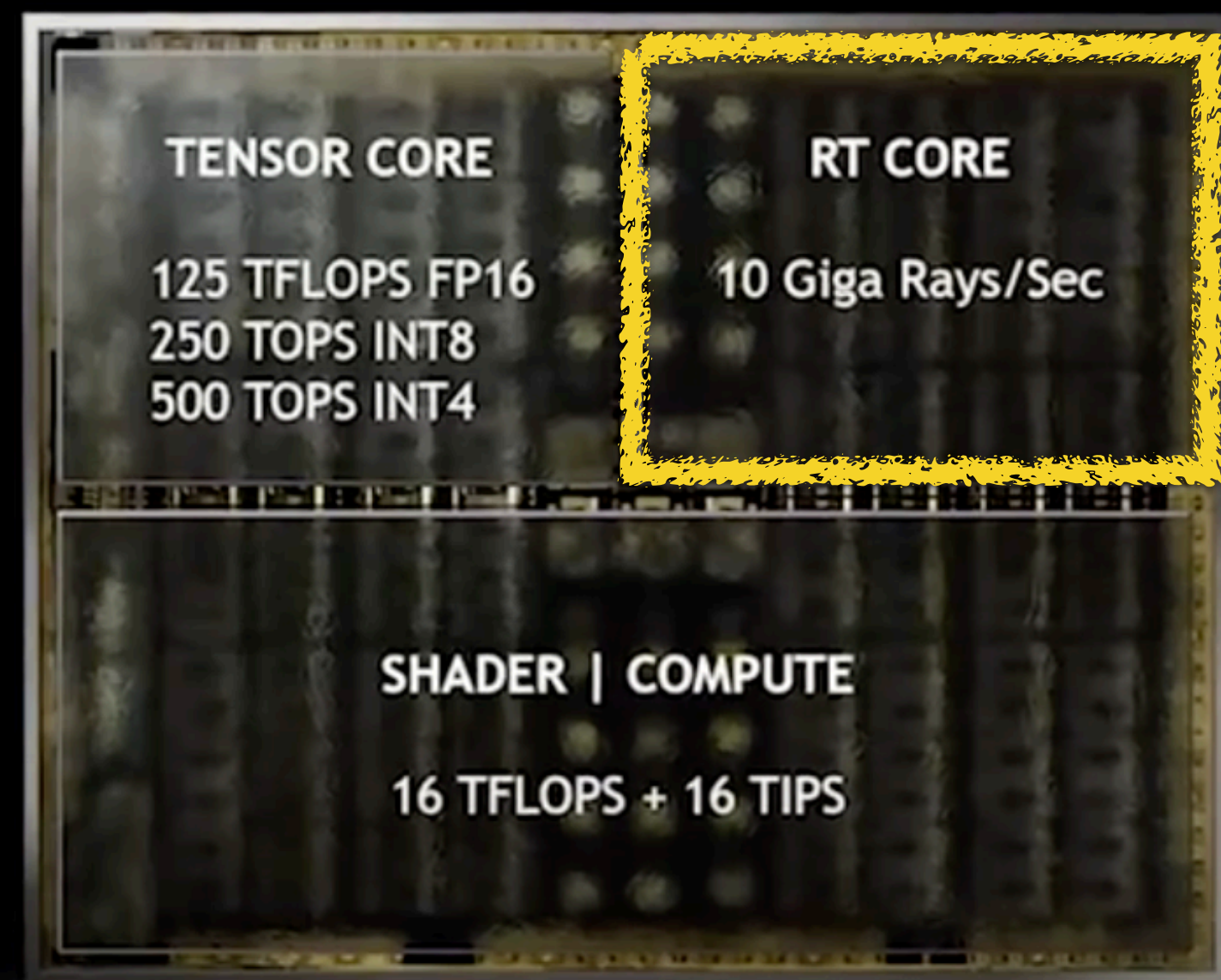
BVH Traversal

- Pathtracing without BVH is misery
 - As we know from Project 3
- But even with the BVH, computing ray intersections takes up primary time
 - BVH traversal with shader operations = ???
 - Thousands of instructions needed



PASCAL

11.8 Billion xtors | 471 mm² | 24 GB 10GHz



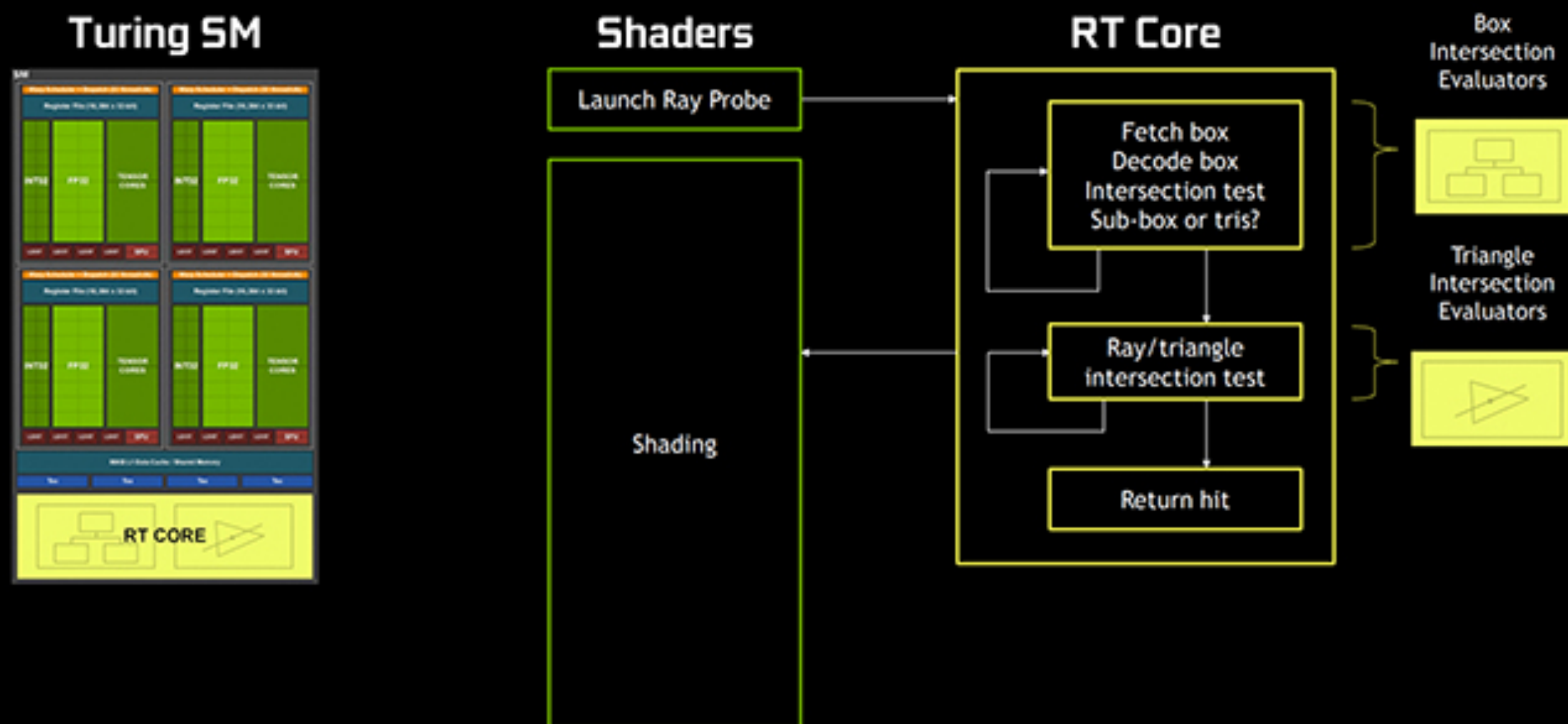
TURING

18.6 Billion xtors | 754 mm² | 48+48 GB 14GHz

Source: NVIDIA

TURING RAY TRACING WITH RT CORES

Hardware Acceleration Replaces Software Emulation



THIS PRESENTATION IS EMBARGOED UNTIL SEPTEMBER 14, 2018

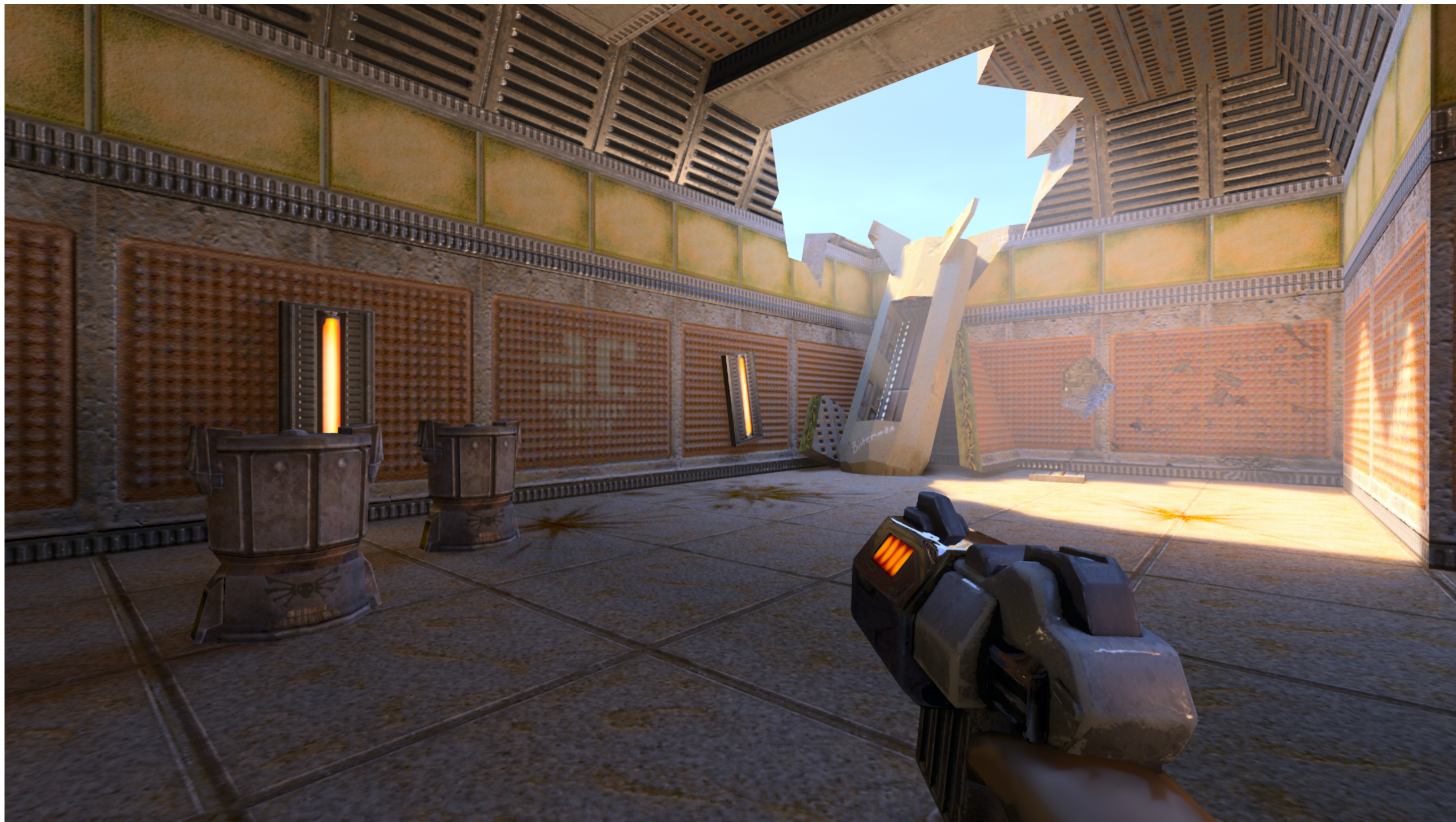


Source: NVIDIA

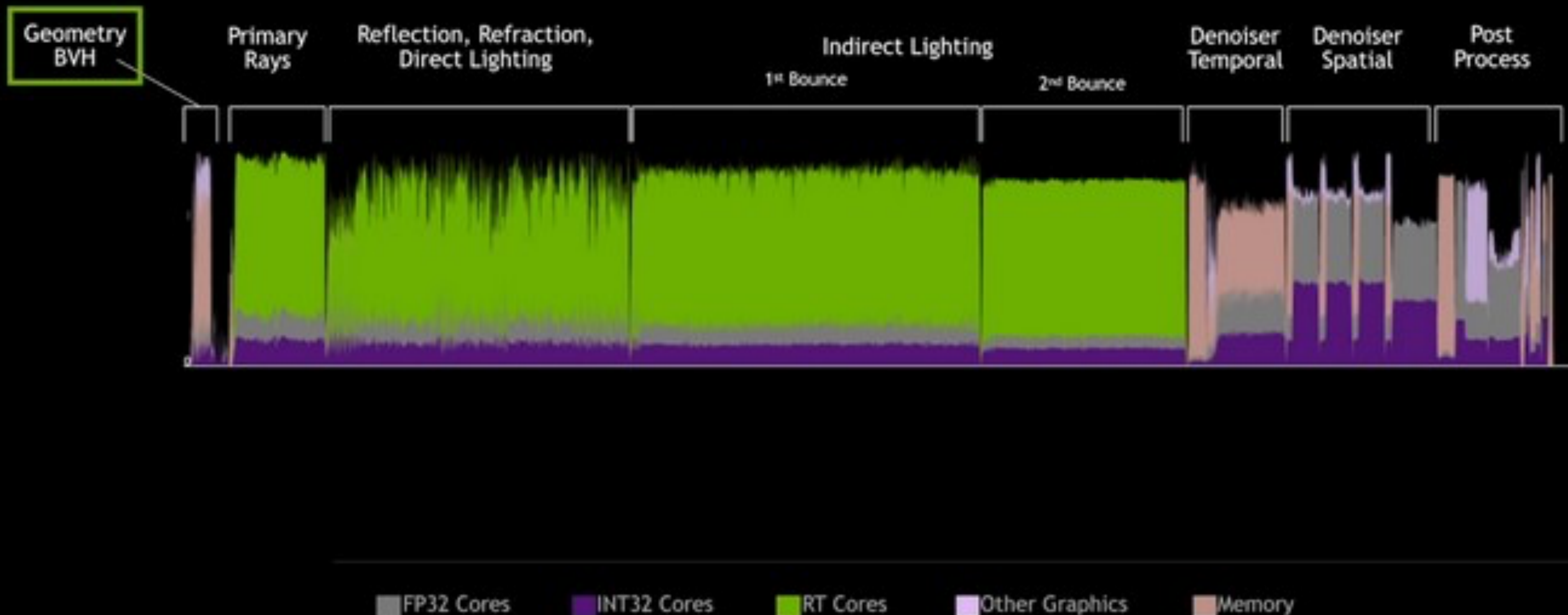
Quake II RTX

(All?) games marketed as supporting RTX actually use raytracing only for particular effects.

Exception: Quake II RTX (open source!)



Q2 RTX FRAME BREAKDOWN



Source: NVIDIA

Acknowledgments

Many thanks to Jonathan Ragan-Kelley for slide materials and lecture content.

Credit for slides and contributions to Kayvon Fatahalian, Kurt Akeley, Solomon Boulos, Mike Doggett, Pat Hanrahan, Mike Houston, Jeremy Sugerman.