Beyond Programmable Shading Course

ACM SIGGRAPH 2010

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

Kayvon Fatahalian

Stanford University

This talk

- Three major ideas that make GPU processing cores run fast
- Closer look at real GPU designs
  - NVIDIA GTX 480
  - ATI Radeon 5870
- The GPU memory hierarchy: moving data to processors

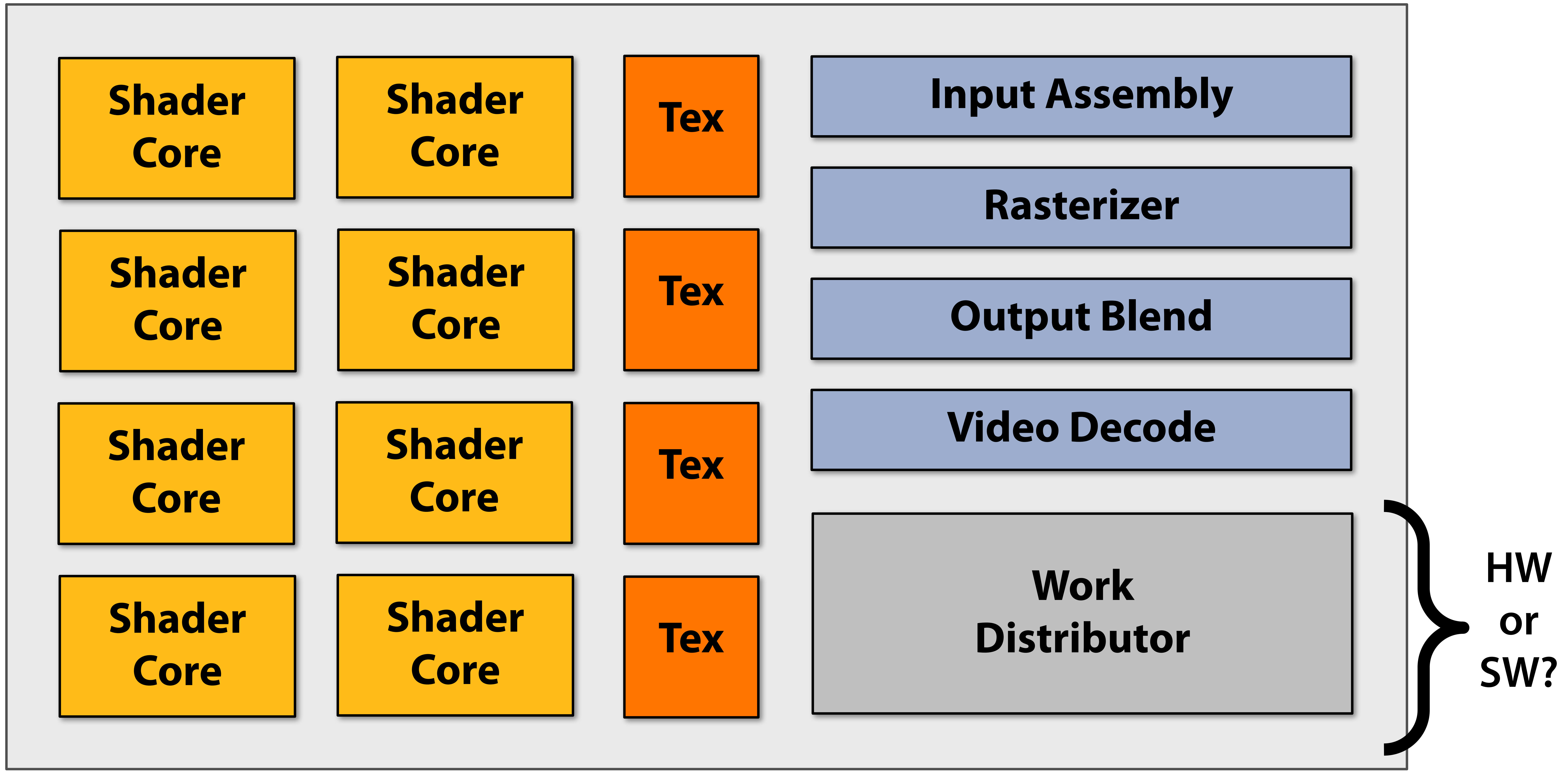
Part 1: throughput processing

- 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. Optimize shaders/compute kernels
  3. 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

Compile shader

1 unshaded fragment input record ⬜

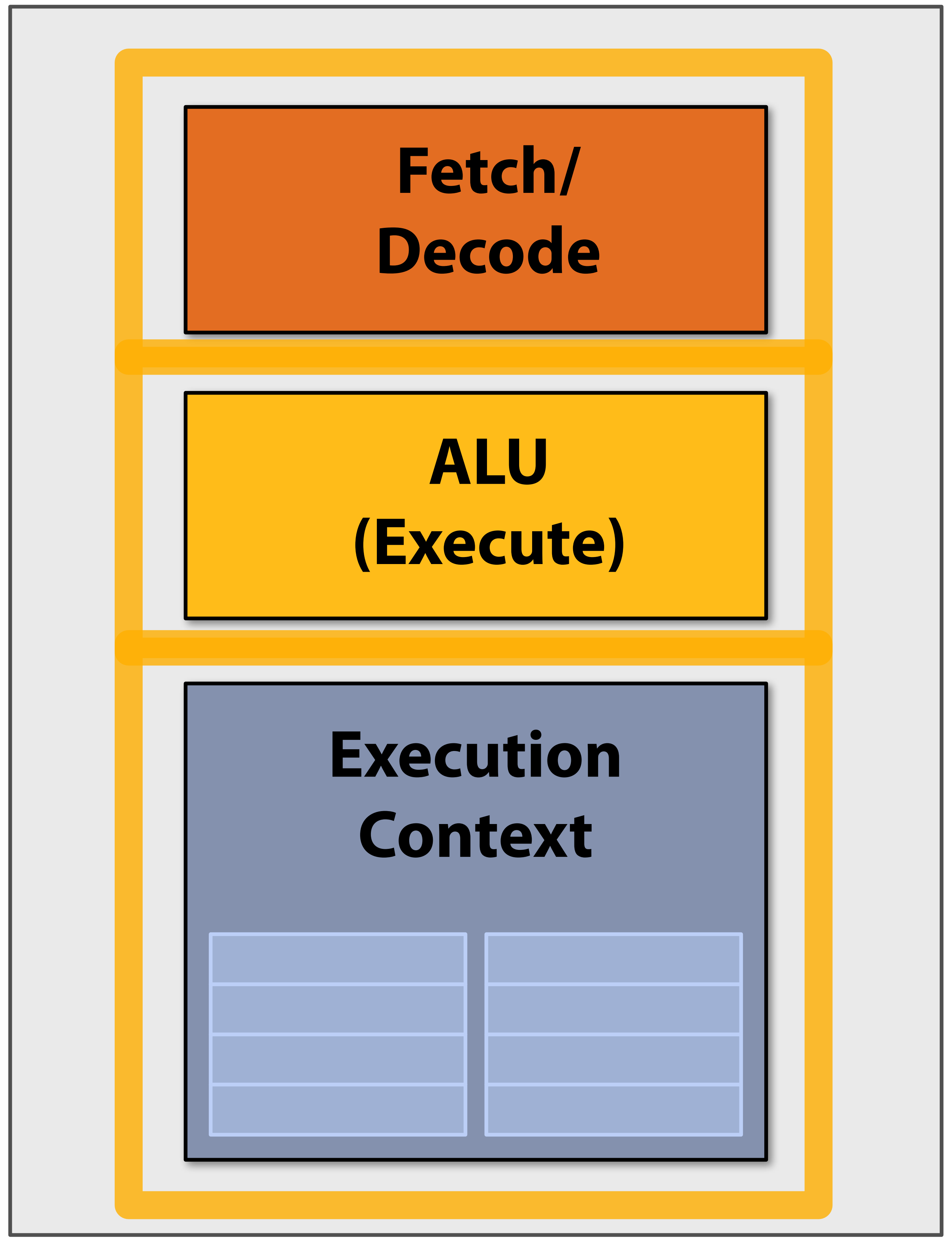
```
sampler mySamp;
Texture2D<float3> myTex;
float3 lightDir;

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);
}
```

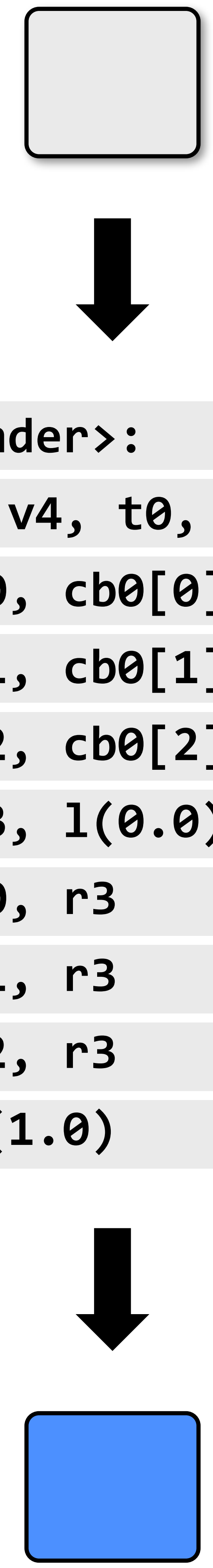
→

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

1 shaded fragment output record 🟦

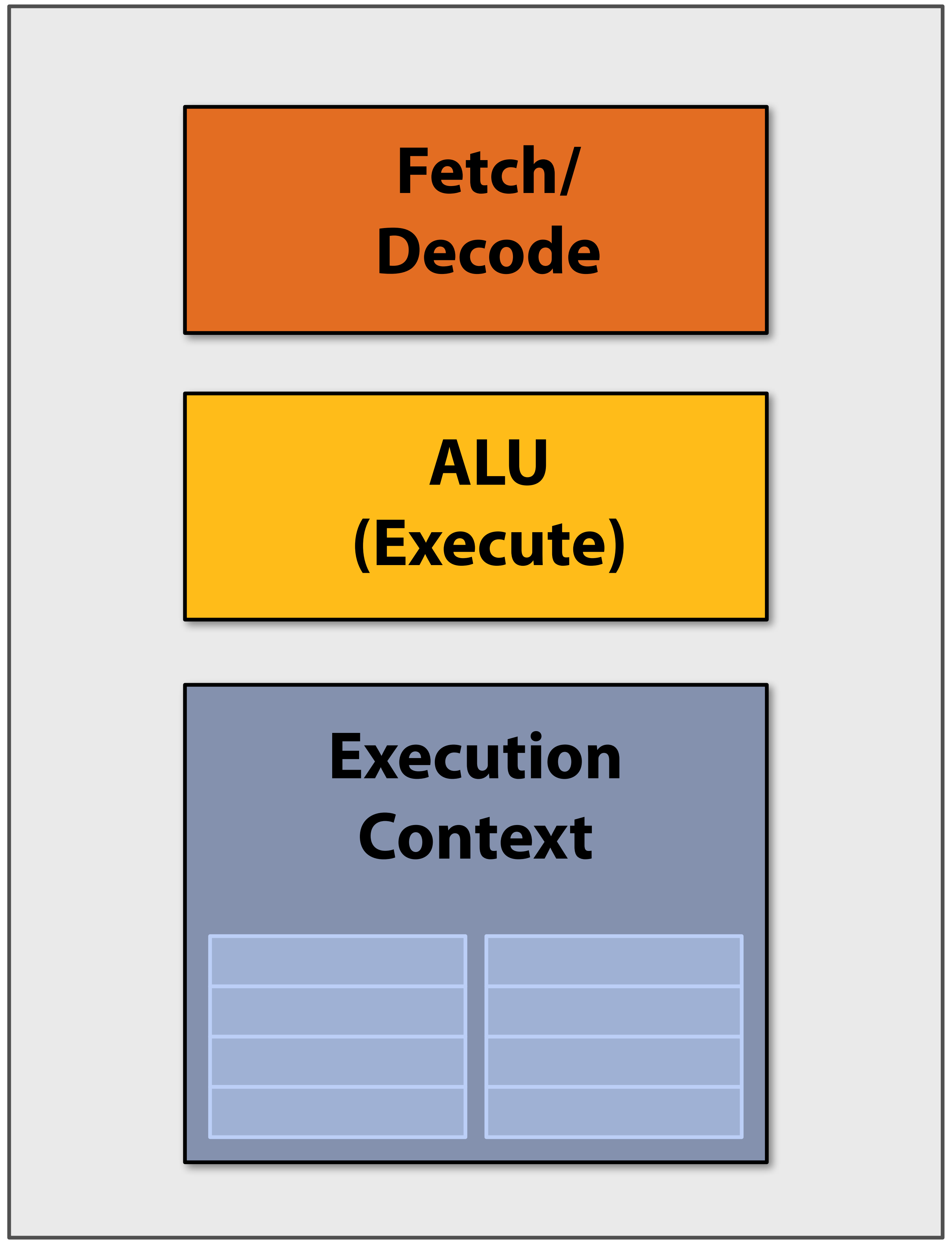
Execute shader

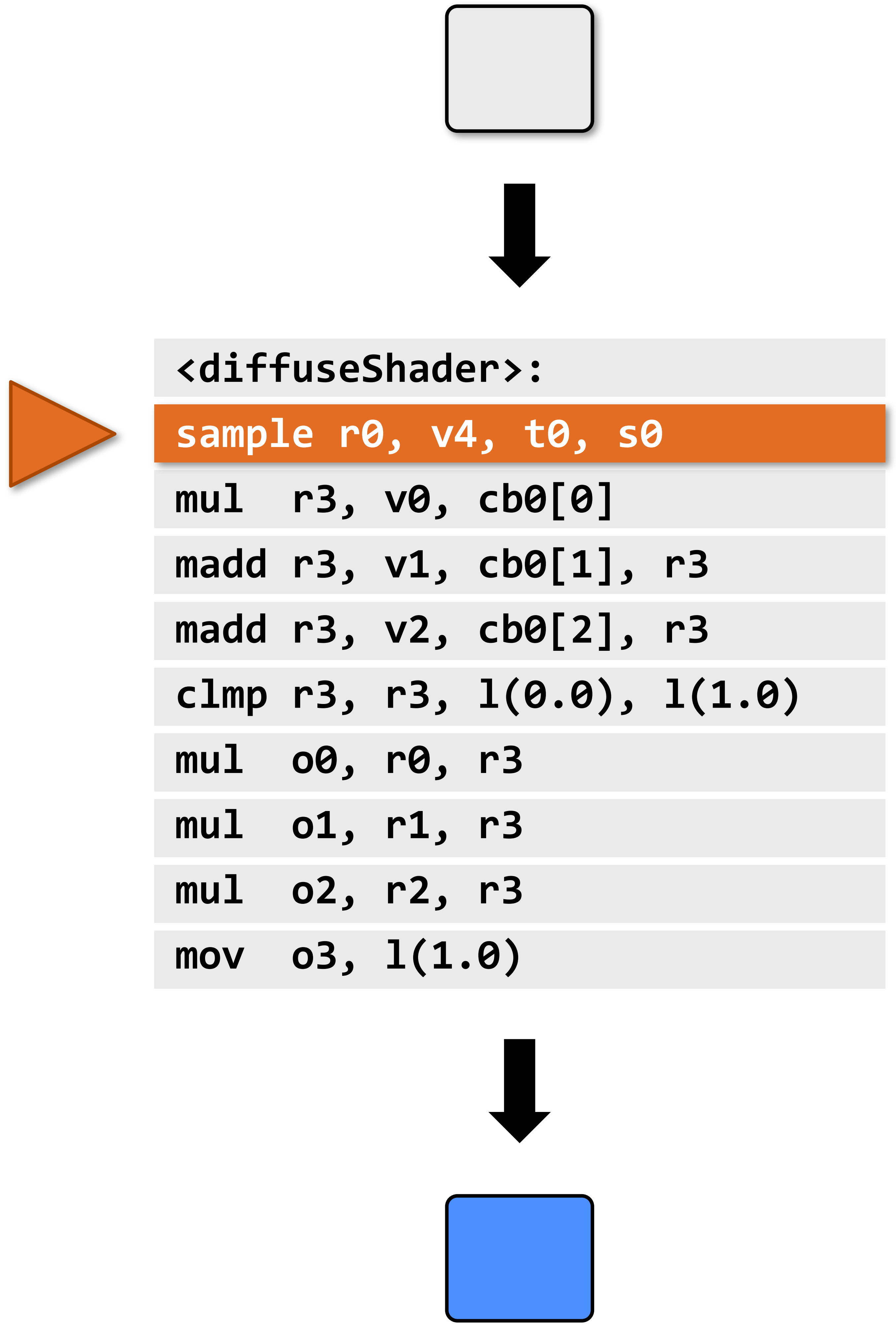




```
<diffuseShader>:
sample r0, v4, t0, s0
mul  r3, v0, cb0[0]
madd r3, v1, cb0[1], r3
madd r3, v2, cb0[2], r3
clmp r3, r3, l(0.0), l(1.0)
mul  o0, r0, r3
mul  o1, r1, r3
mul  o2, r2, r3
mov  o3, l(1.0)
```

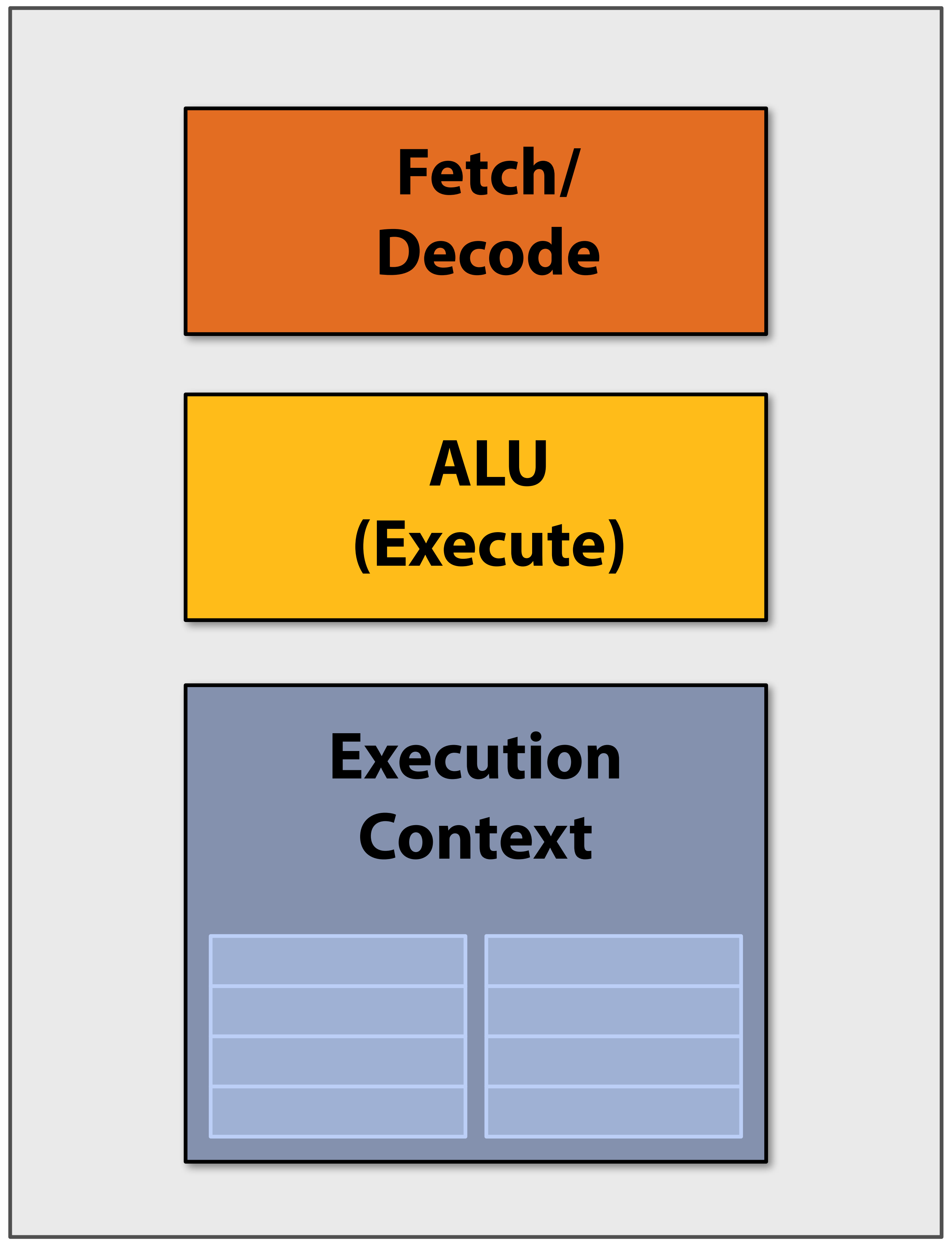
Execute shader

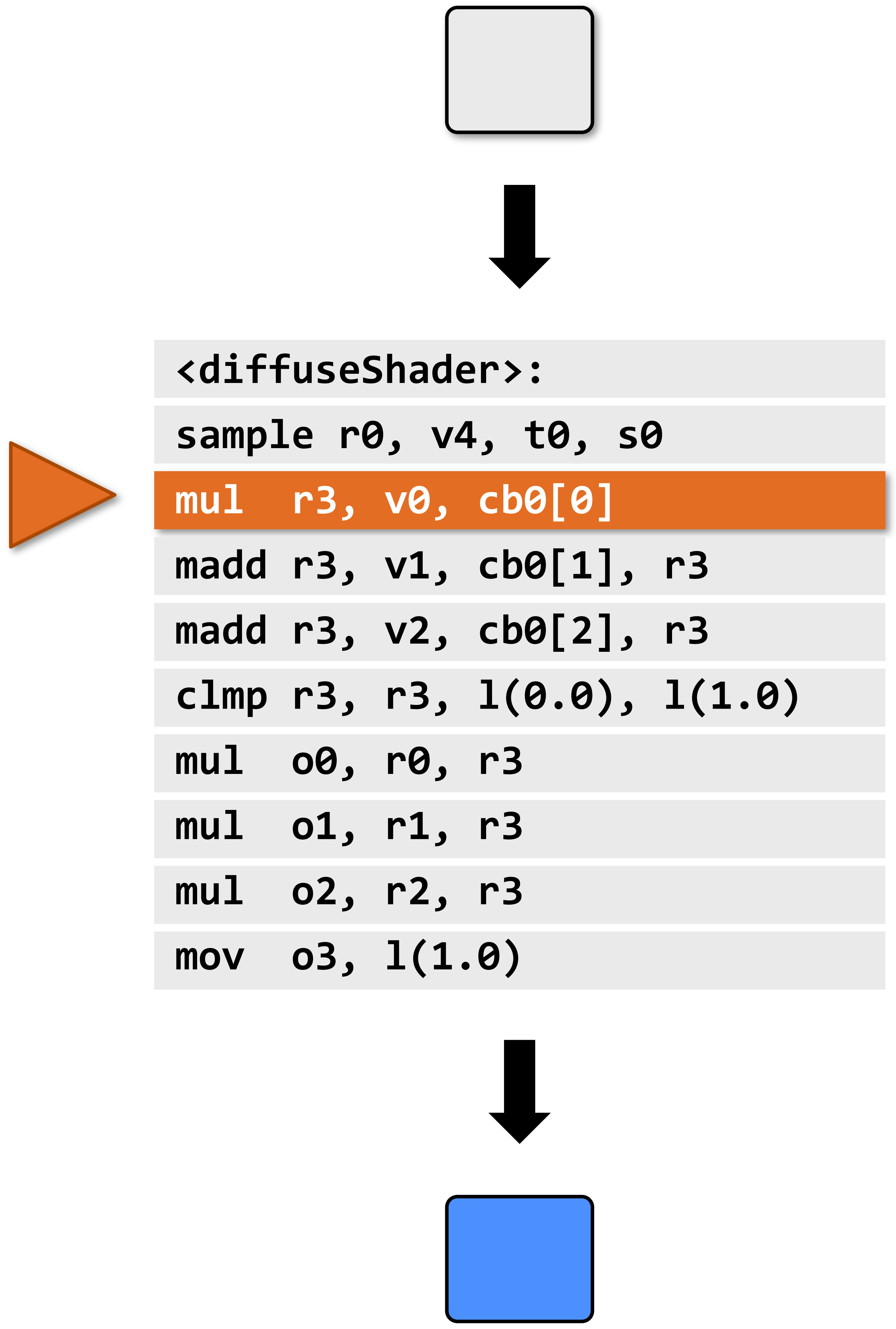




```
<diffuseShader>:
sample r0, v4, t0, s0
mul  r3, v0, cb0[0]
madd r3, v1, cb0[1], r3
madd r3, v2, cb0[2], r3
clmp r3, r3, l(0.0), l(1.0)
mul  o0, r0, r3
mul  o1, r1, r3
mul  o2, r2, r3
mov  o3, l(1.0)
```

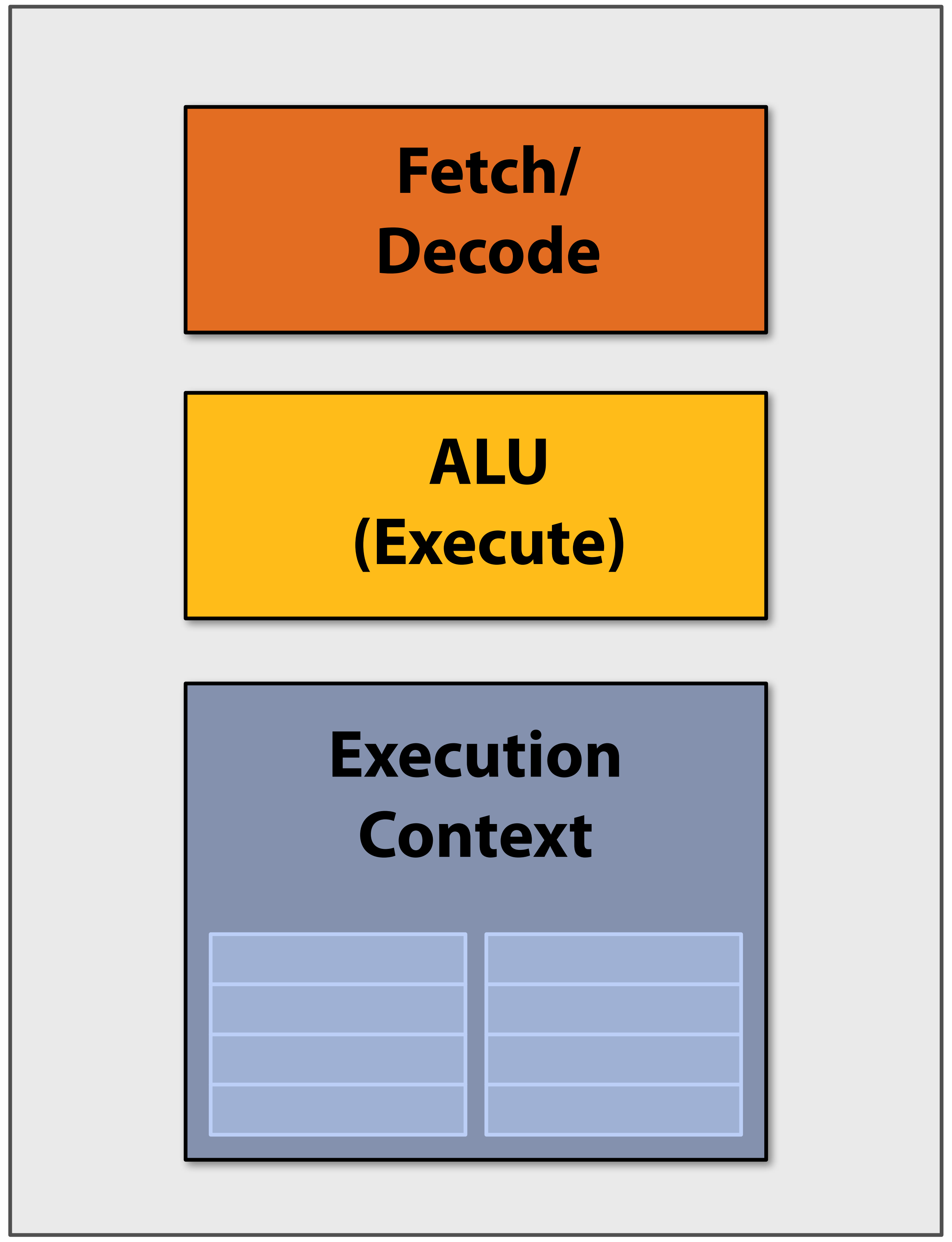
Execute shader

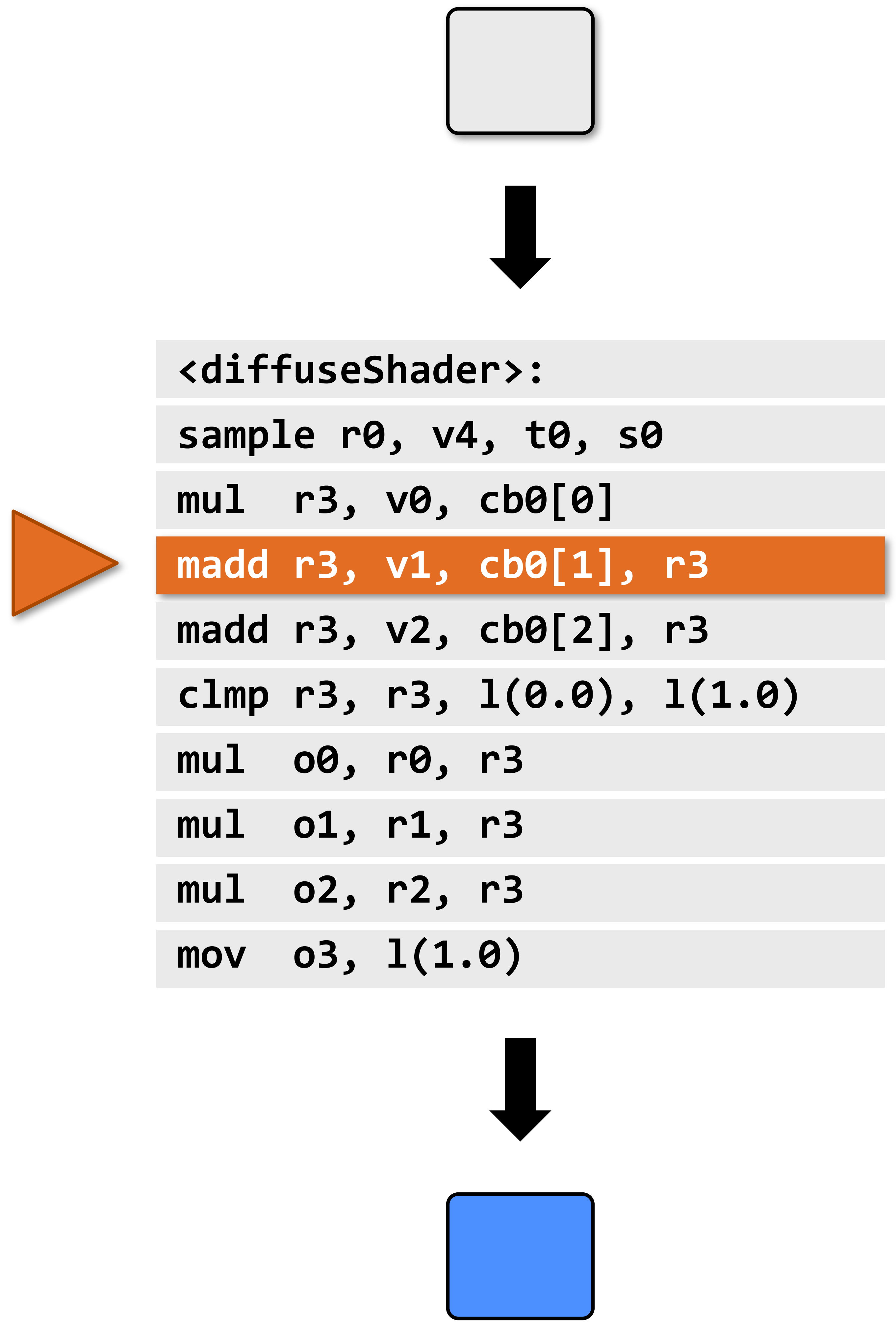




```
<diffuseShader>:
sample r0, v4, t0, s0
mul  r3, v0, cb0[0]
madd r3, v1, cb0[1], r3
madd r3, v2, cb0[2], r3
clmp r3, r3, l(0.0), l(1.0)
mul  o0, r0, r3
mul  o1, r1, r3
mul  o2, r2, r3
mov  o3, l(1.0)
```

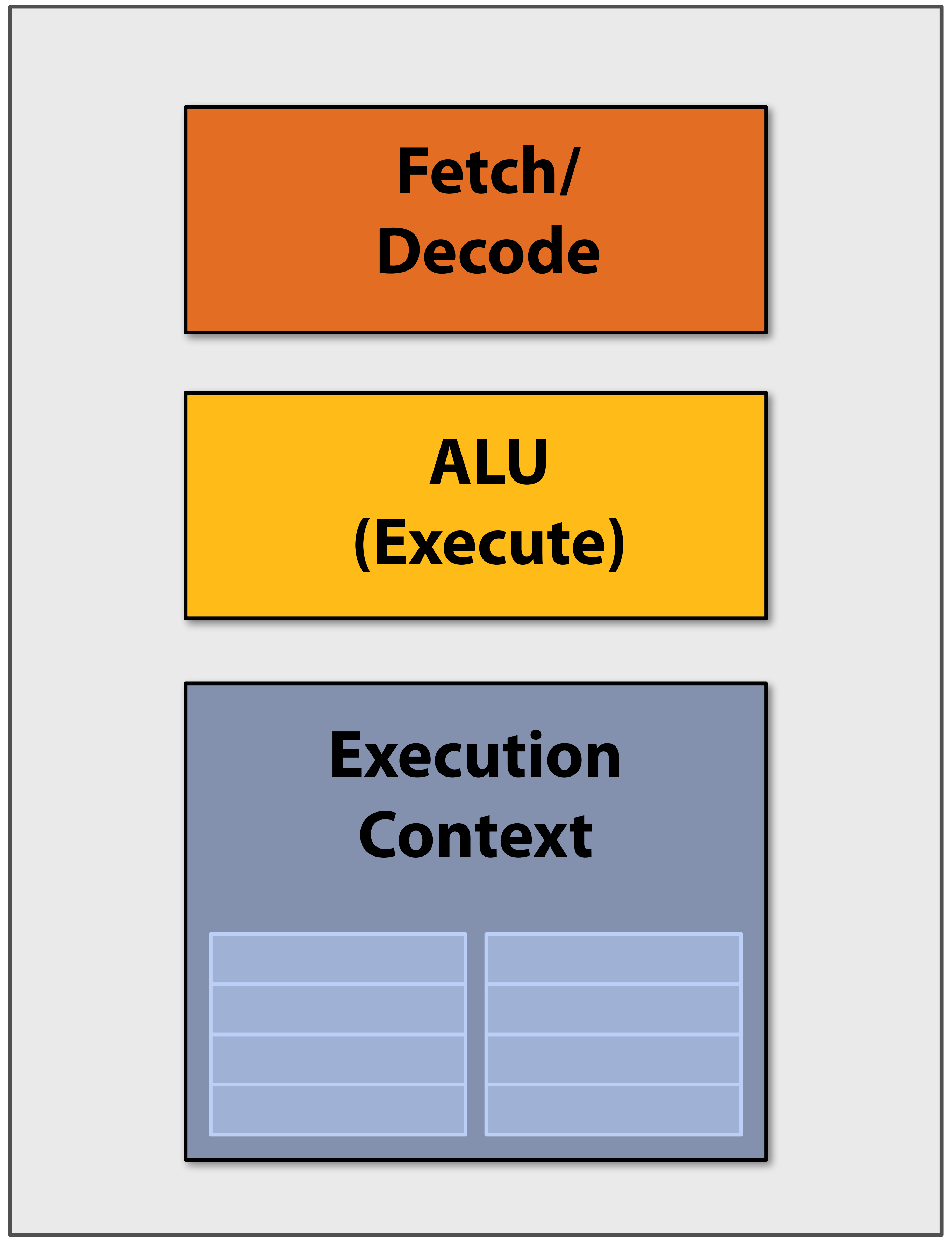
Execute shader

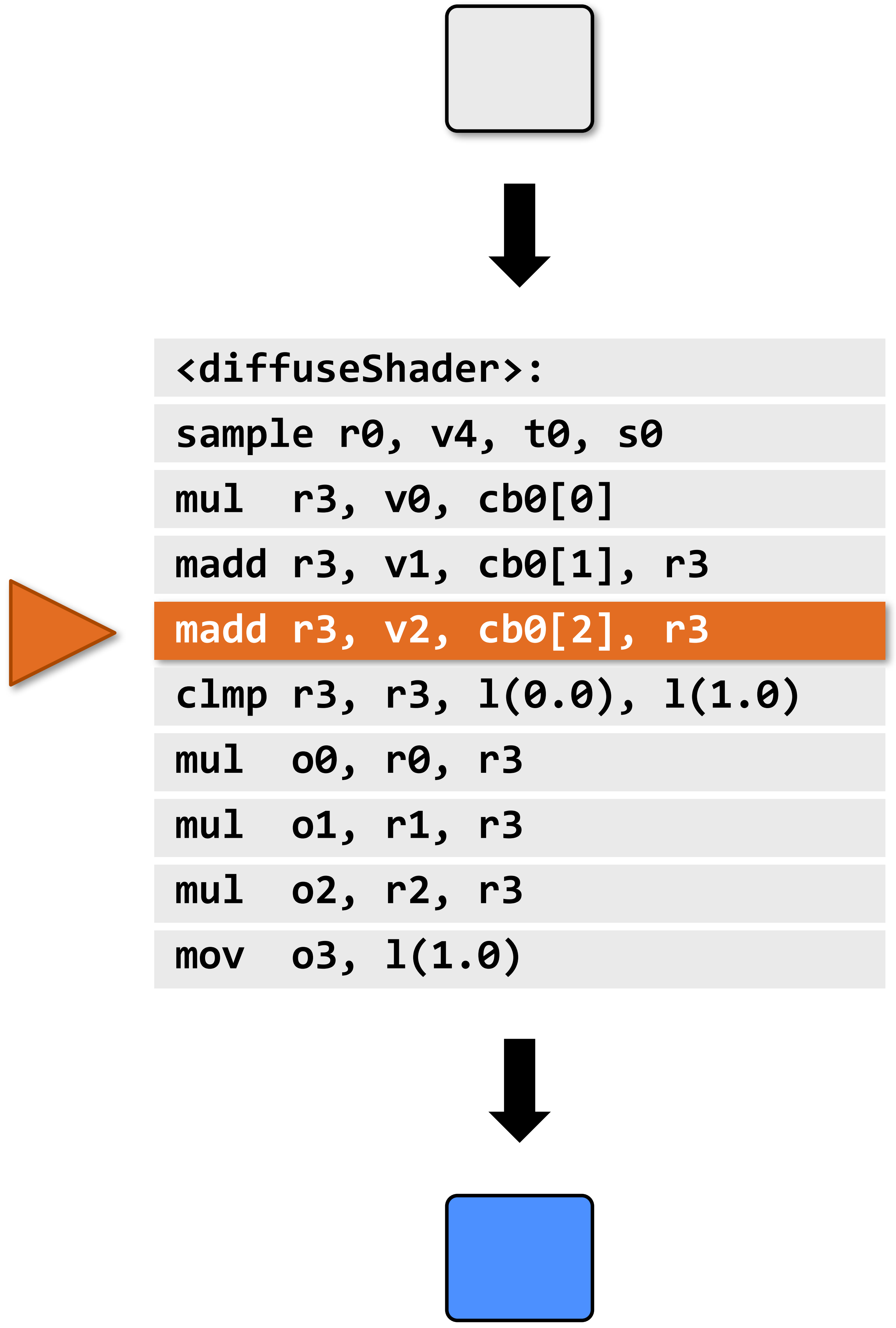




```
<diffuseShader>:
sample r0, v4, t0, s0
mul  r3, v0, cb0[0]
madd r3, v1, cb0[1], r3
madd r3, v2, cb0[2], r3
clmp r3, r3, l(0.0), l(1.0)
mul  o0, r0, r3
mul  o1, r1, r3
mul  o2, r2, r3
mov  o3, l(1.0)
```

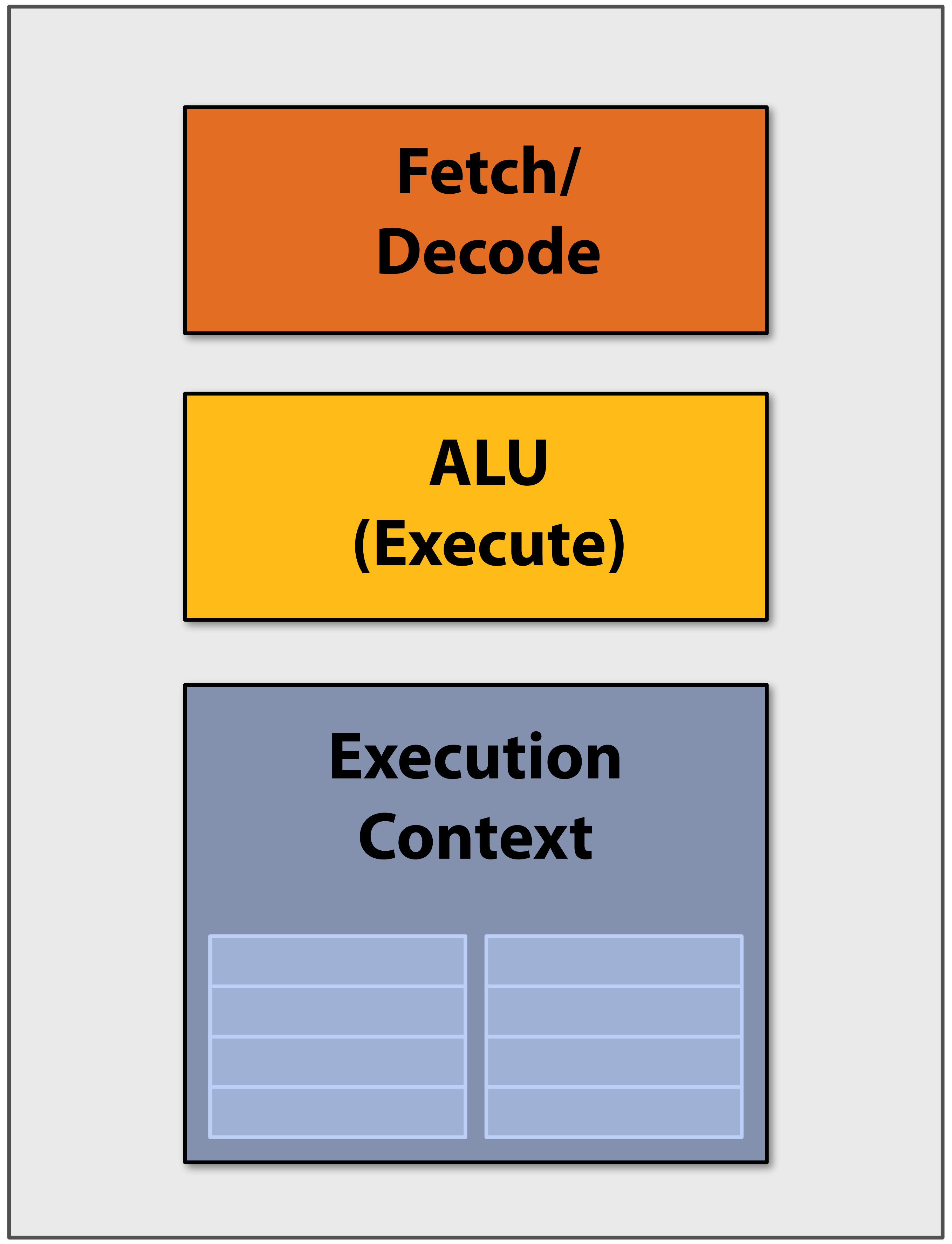
Execute shader

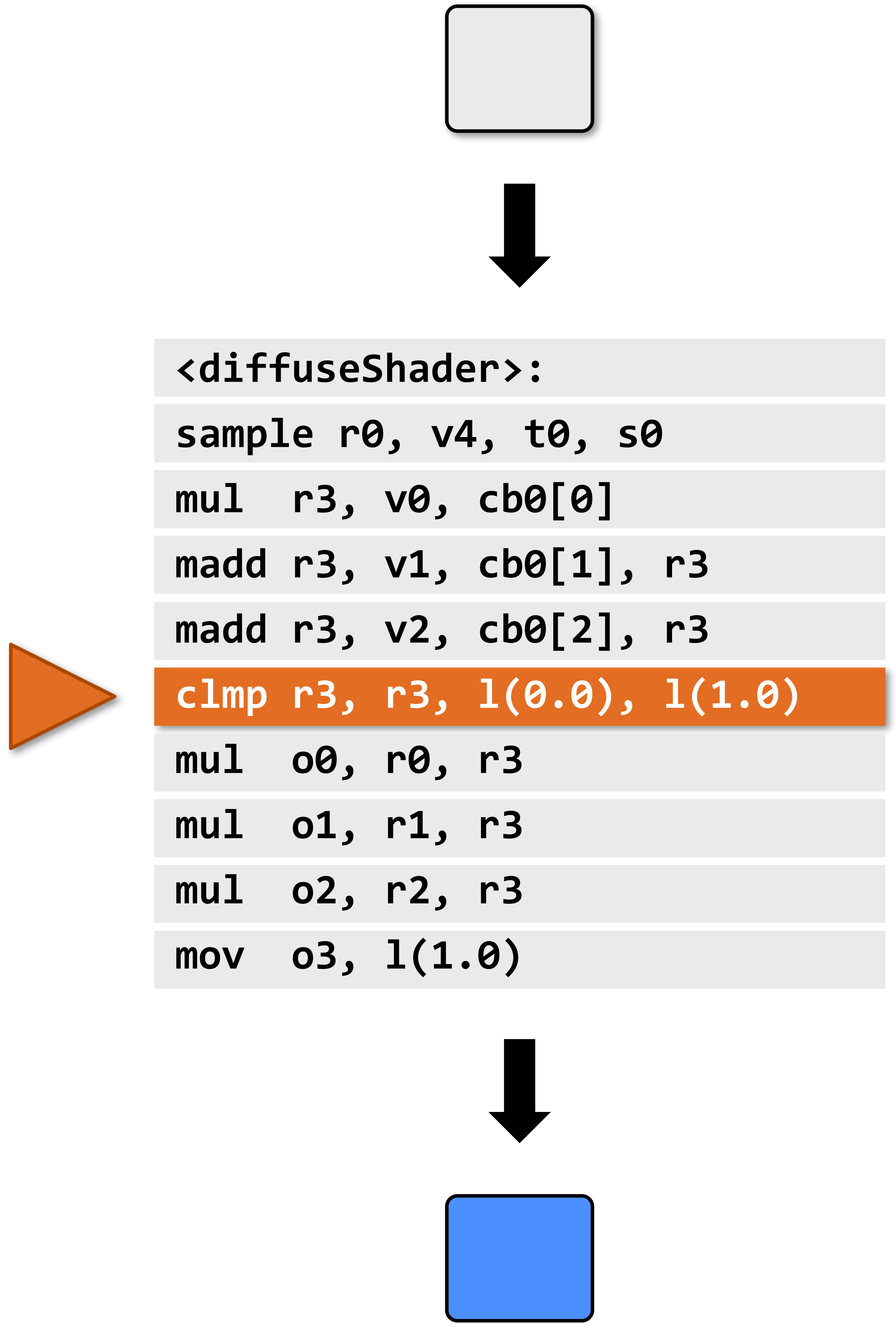




```
<diffuseShader>:
sample r0, v4, t0, s0
mul  r3, v0, cb0[0]
madd r3, v1, cb0[1], r3
madd r3, v2, cb0[2], r3
clmp r3, r3, l(0.0), l(1.0)
mul  o0, r0, r3
mul  o1, r1, r3
mul  o2, r2, r3
mov  o3, l(1.0)
```

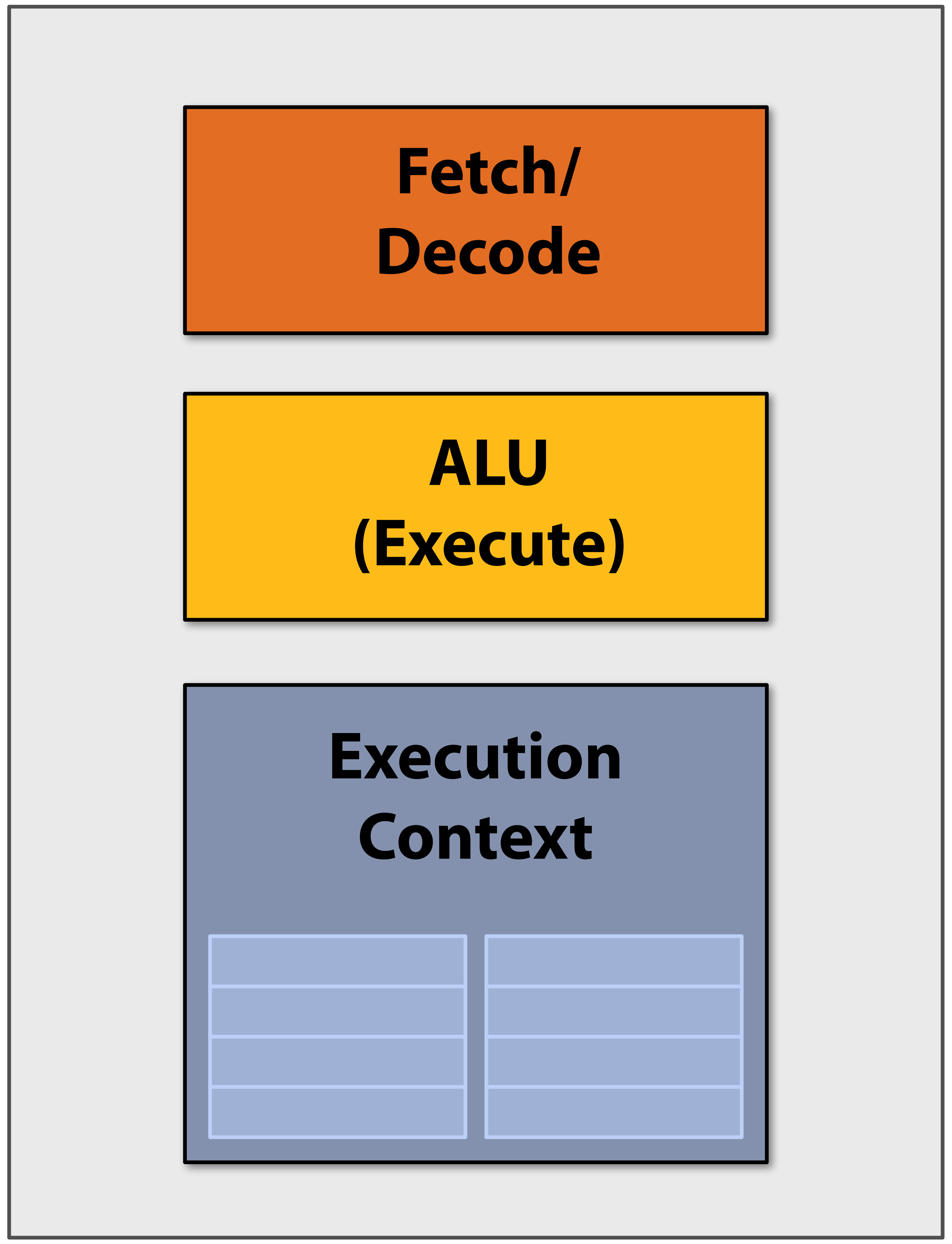
Execute shader

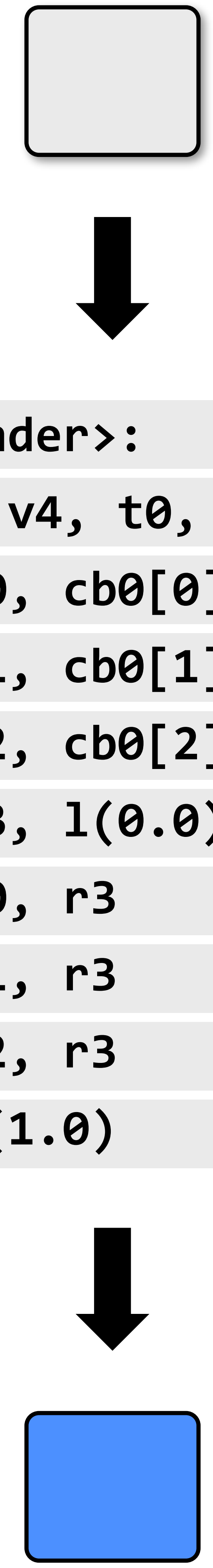




```
<diffuseShader>:
sample r0, v4, t0, s0
mul  r3, v0, cb0[0]
madd r3, v1, cb0[1], r3
madd r3, v2, cb0[2], r3
clmp r3, r3, l(0.0), l(1.0)
mul  o0, r0, r3
mul  o1, r1, r3
mul  o2, r2, r3
mov  o3, l(1.0)
```

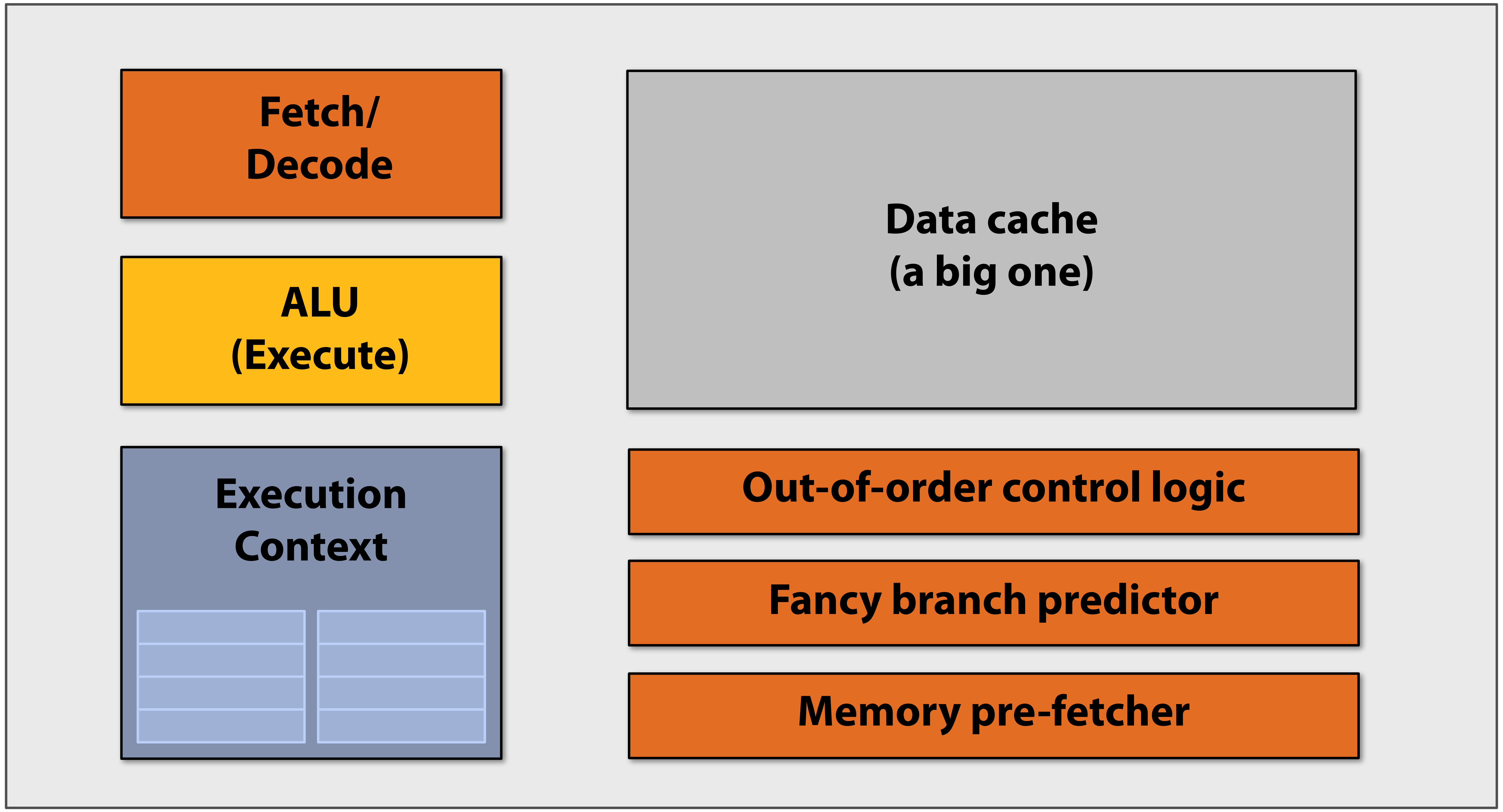
Execute shader



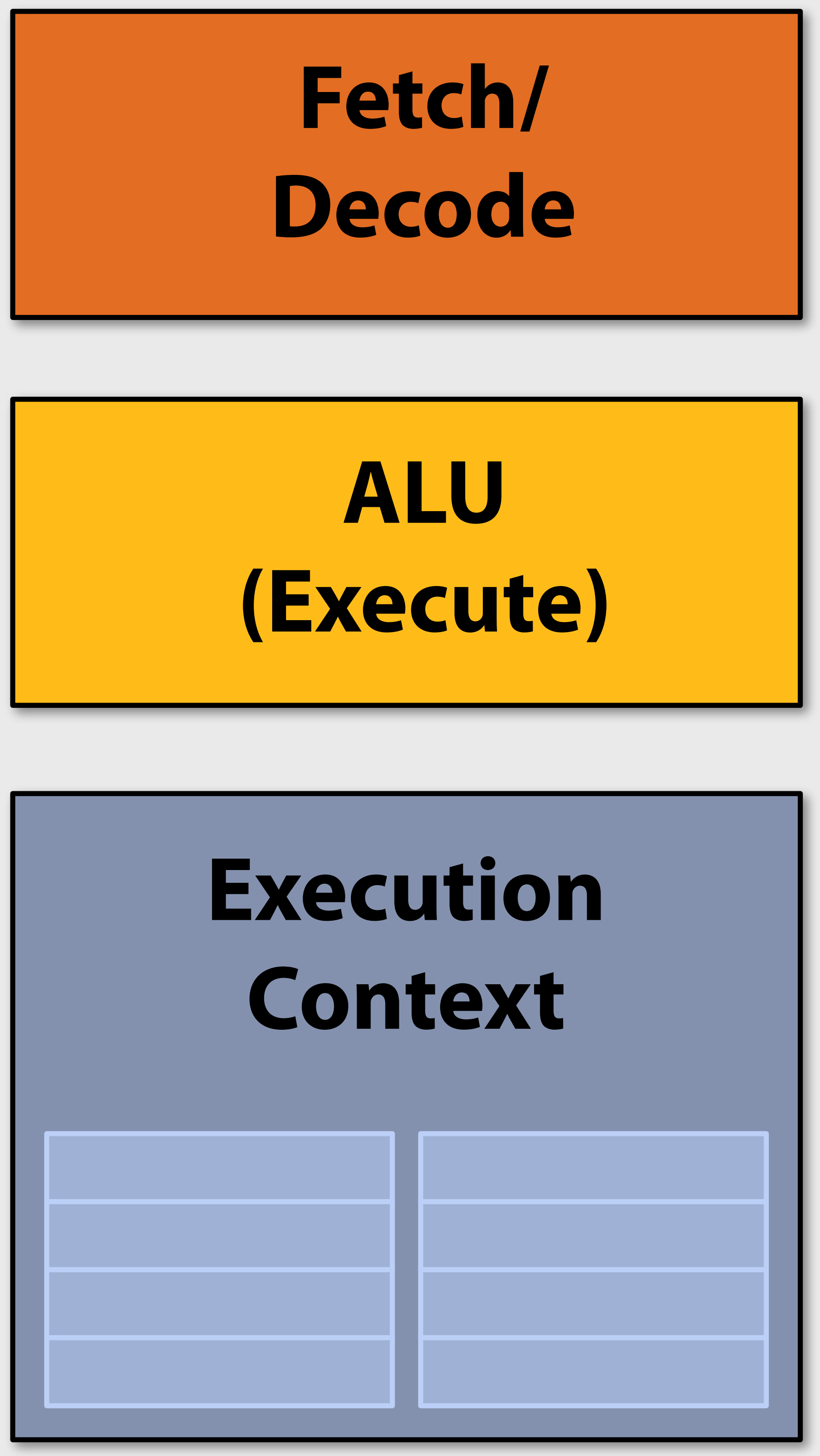


```
<diffuseShader>:
sample r0, v4, t0, s0
mul  r3, v0, cb0[0]
madd r3, v1, cb0[1], r3
madd r3, v2, cb0[2], r3
clmp r3, r3, l(0.0), l(1.0)
mul  o0, r0, r3
mul  o1, r1, r3
mul  o2, r2, r3
mov  o3, l(1.0)
```

“CPU-style” cores



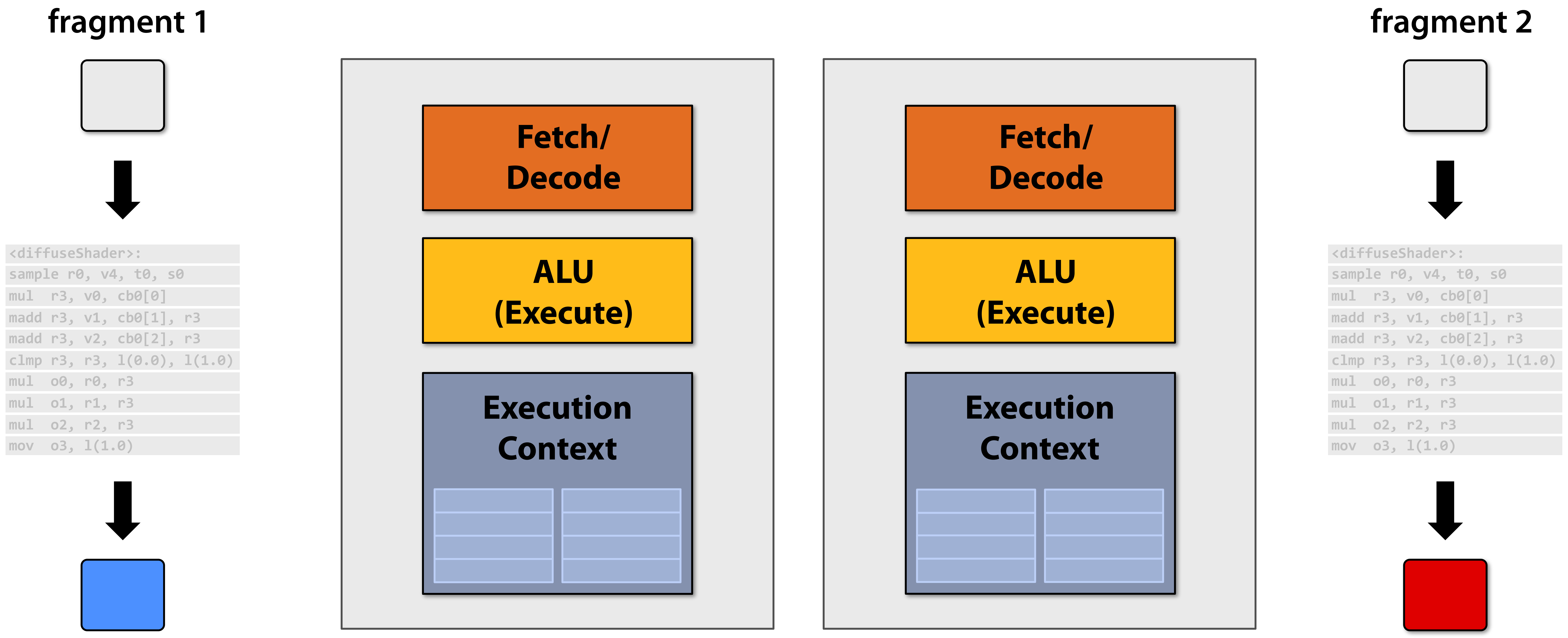
Slimming down



Idea #1:

Remove components that help a single instruction stream run fast

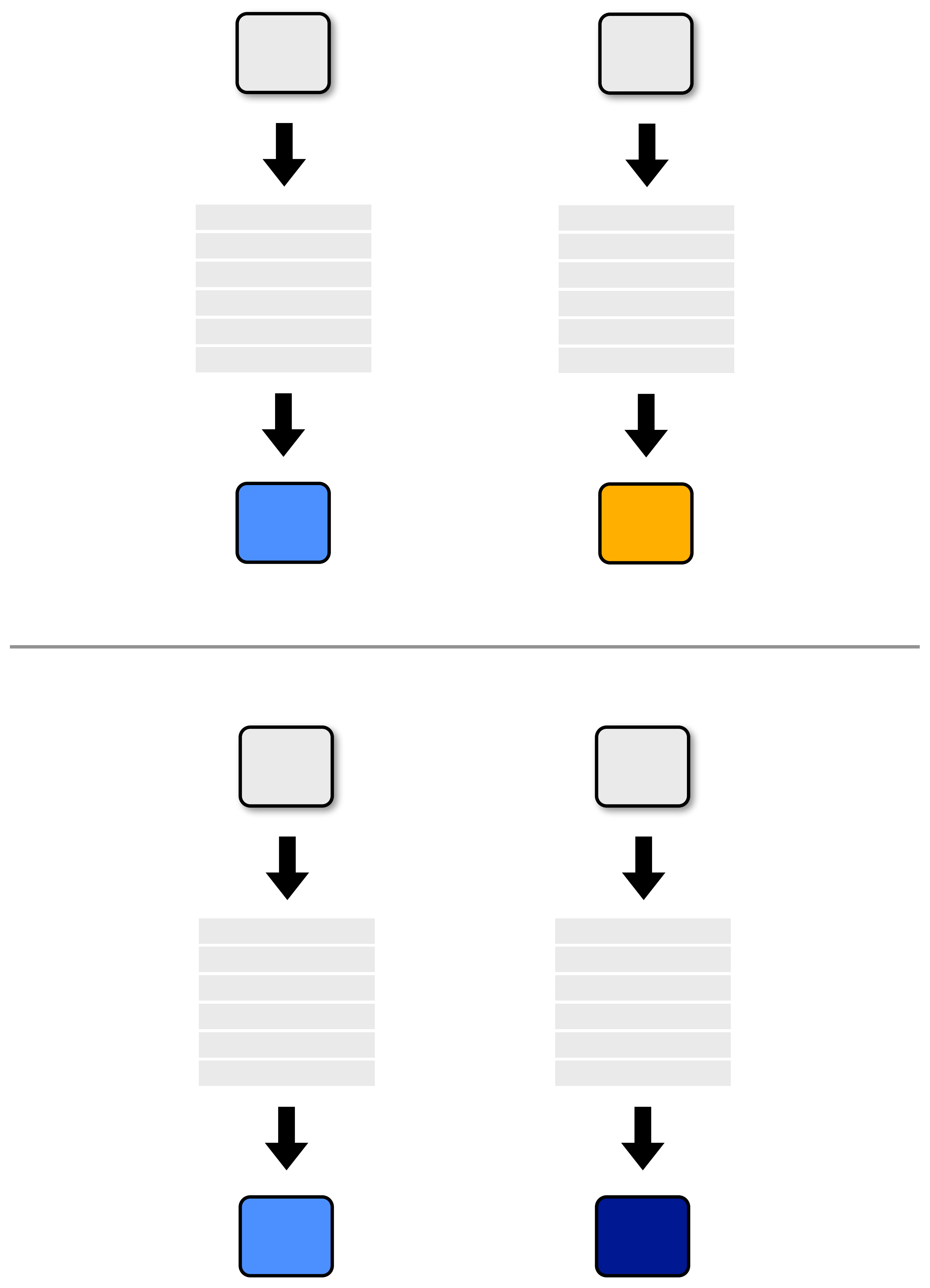
Two cores (two fragments in parallel)

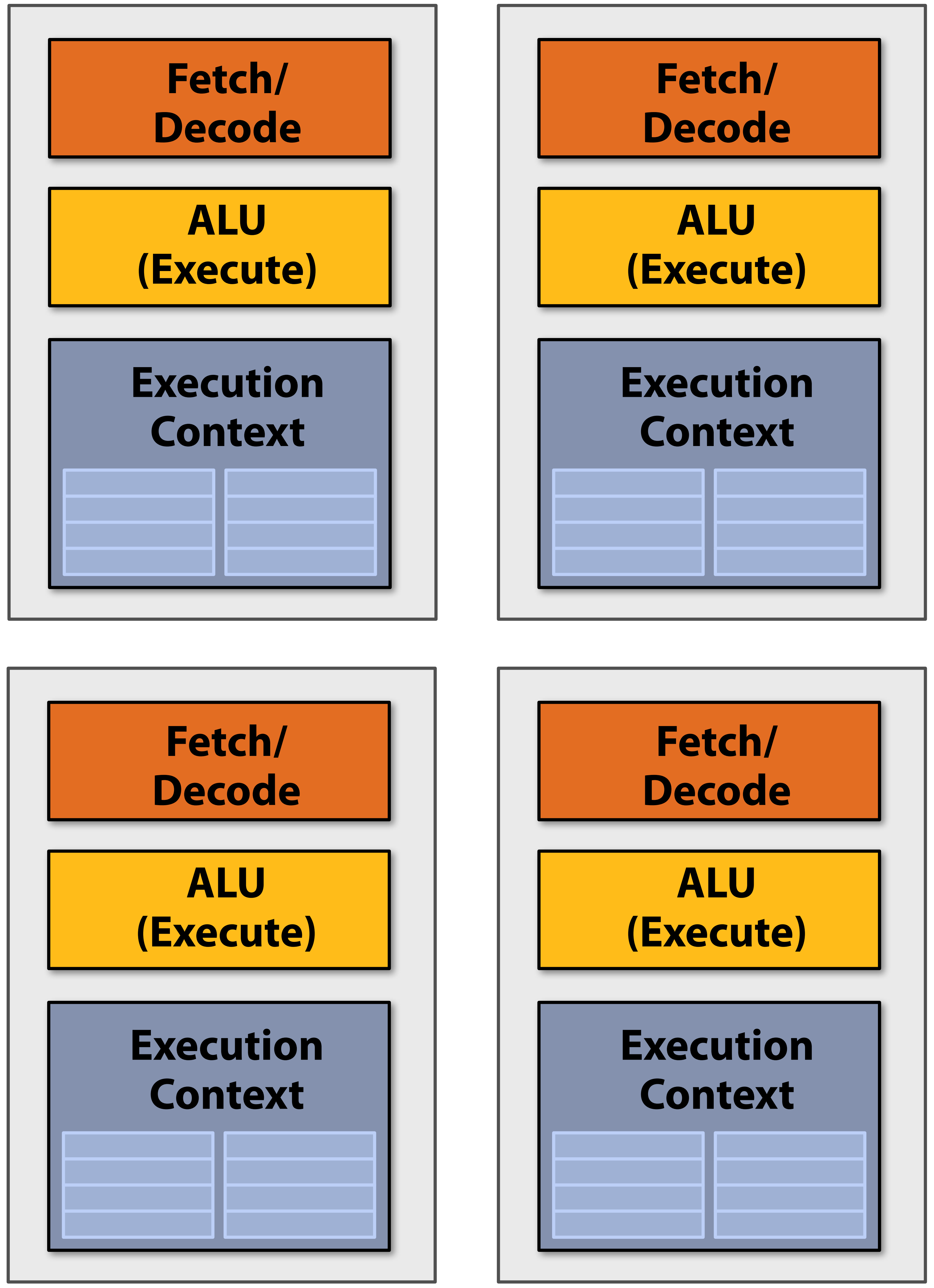


```
<diffuseShader>:
sample r0, v4, t0, s0
mul  r3, v0, cb0[0]
madd r3, v1, cb0[1], r3
madd r3, v2, cb0[2], r3
clmp r3, r3, l(0.0), l(1.0)
mul  o0, r0, r3
mul  o1, r1, r3
mul  o2, r2, r3
mov  o3, l(1.0)
```

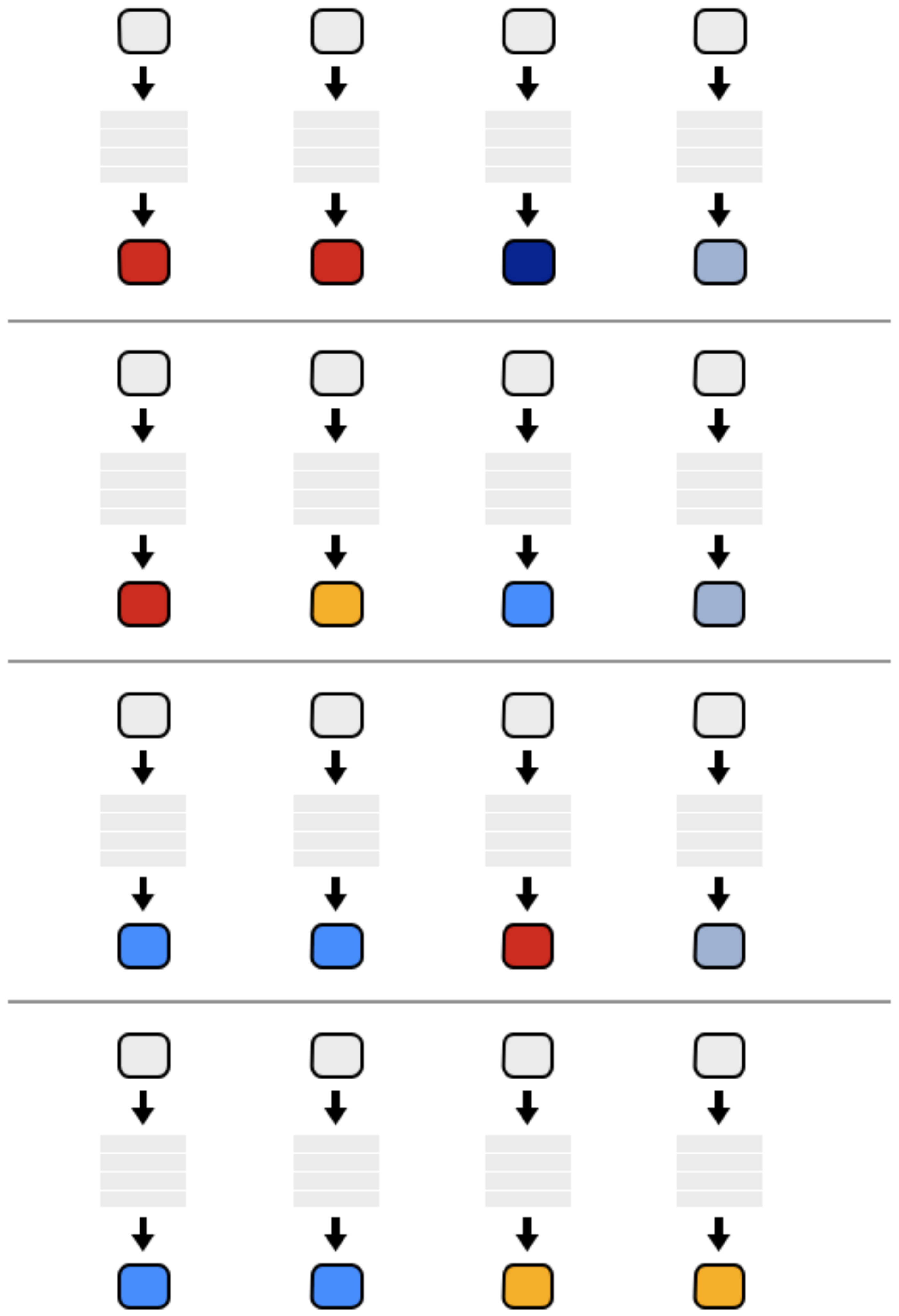
```
<diffuseShader>:
sample r0, v4, t0, s0
mul  r3, v0, cb0[0]
madd r3, v1, cb0[1], r3
madd r3, v2, cb0[2], r3
clmp r3, r3, l(0.0), l(1.0)
mul  o0, r0, r3
mul  o1, r1, r3
mul  o2, r2, r3
mov  o3, l(1.0)
```

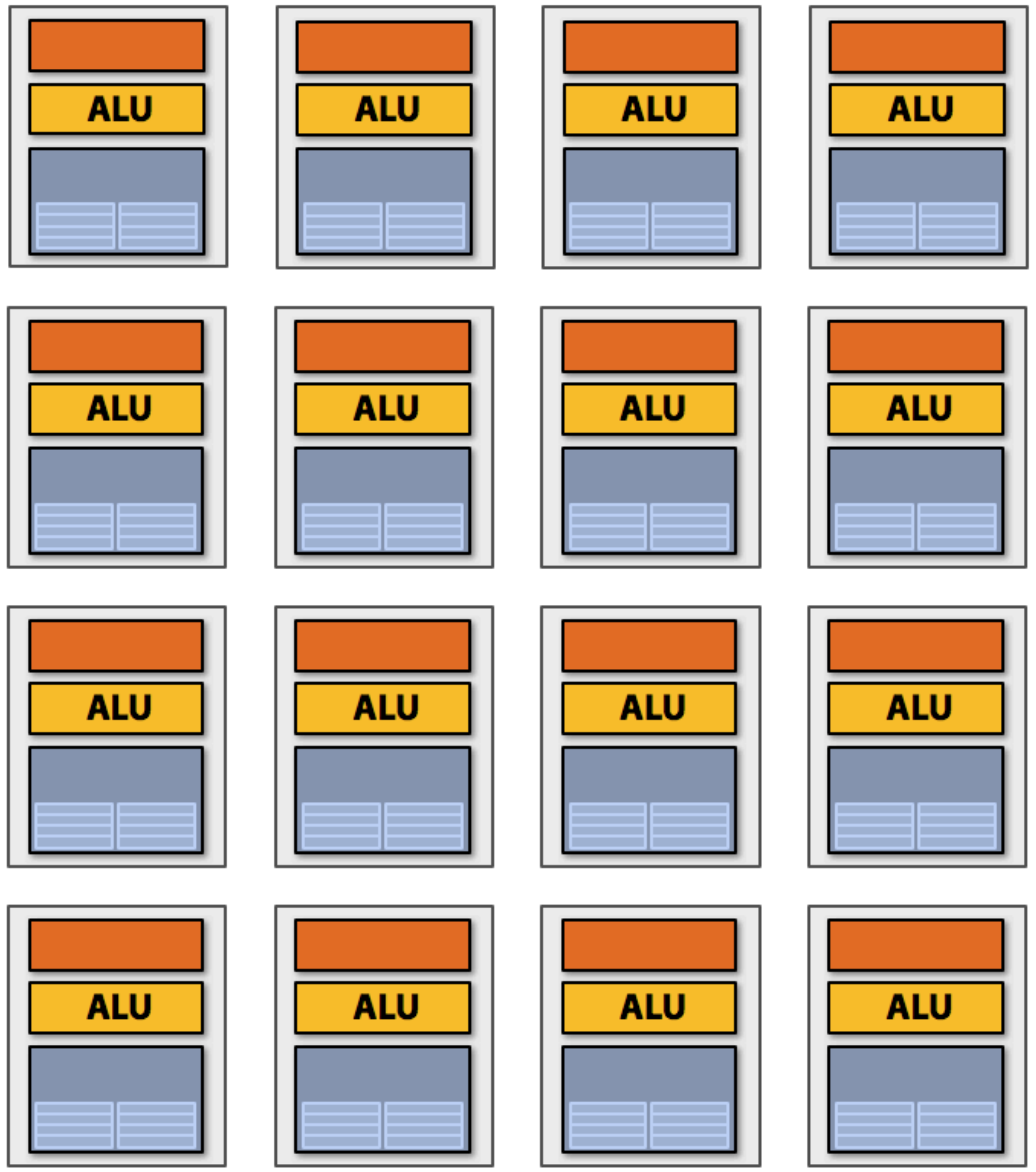
Four cores (four fragments in parallel)





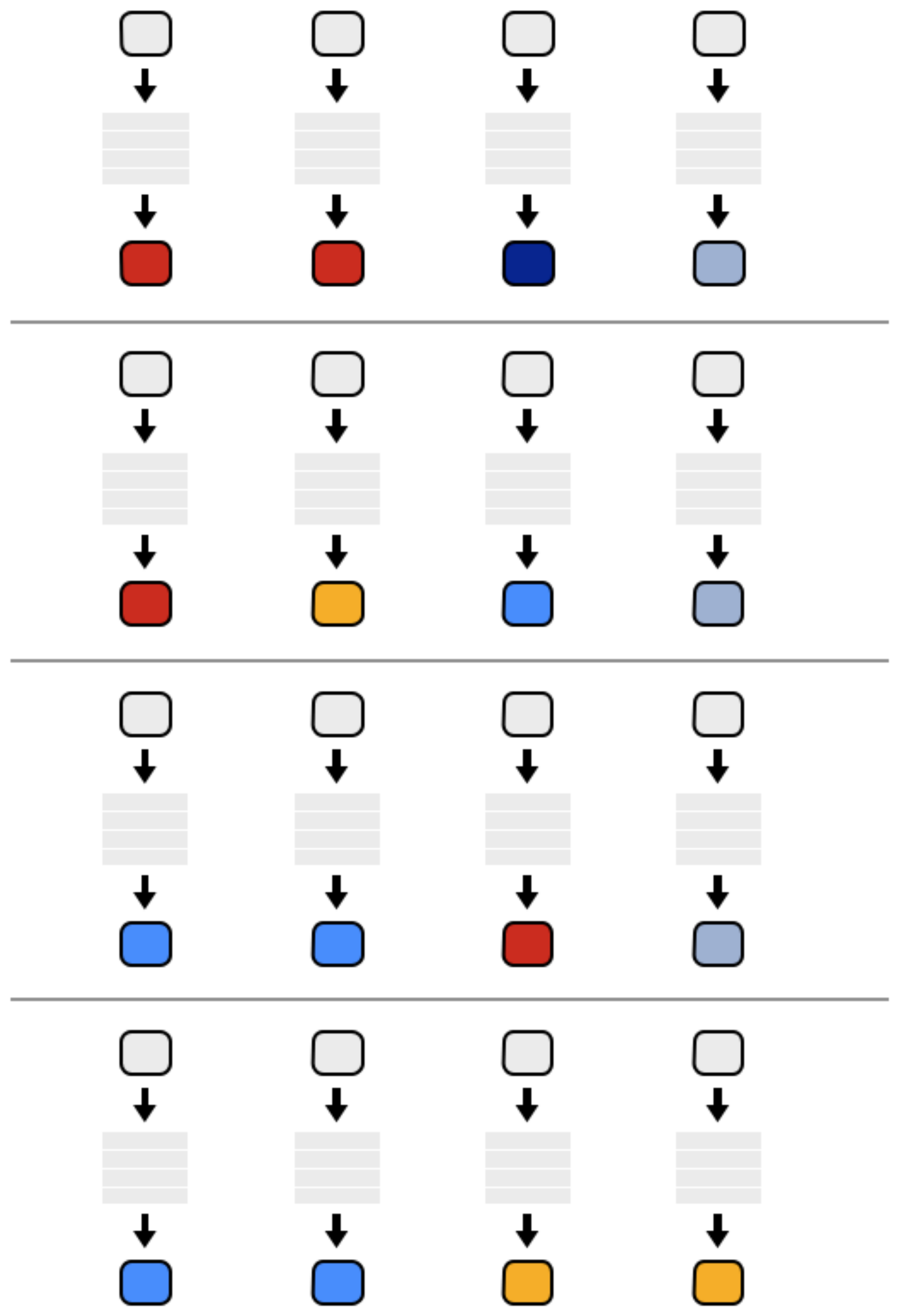
Sixteen cores (sixteen fragments in parallel)





16 cores = 16 simultaneous instruction streams

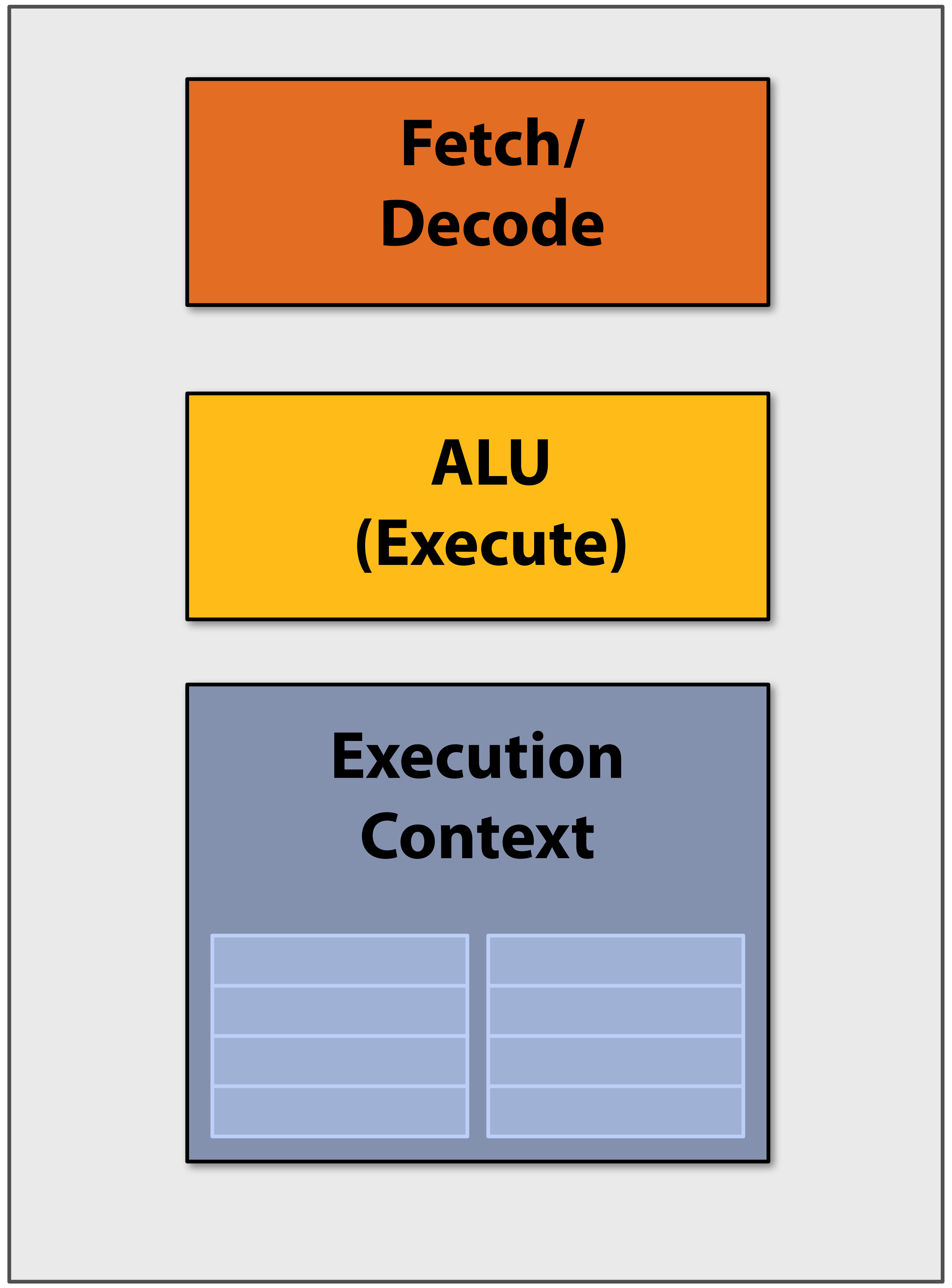
Instruction stream sharing



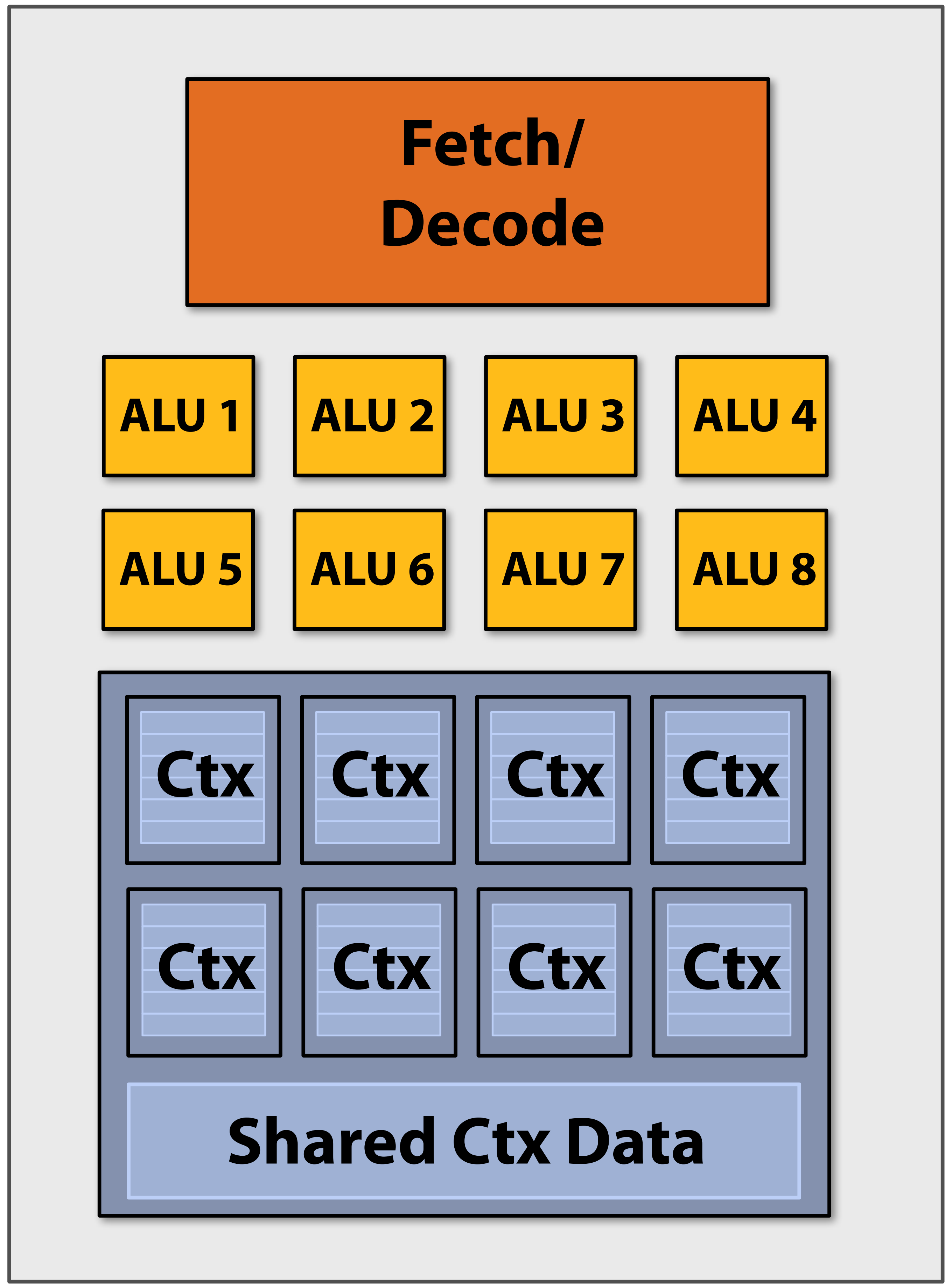
But ... many fragments should be able to share an instruction stream!

```
<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



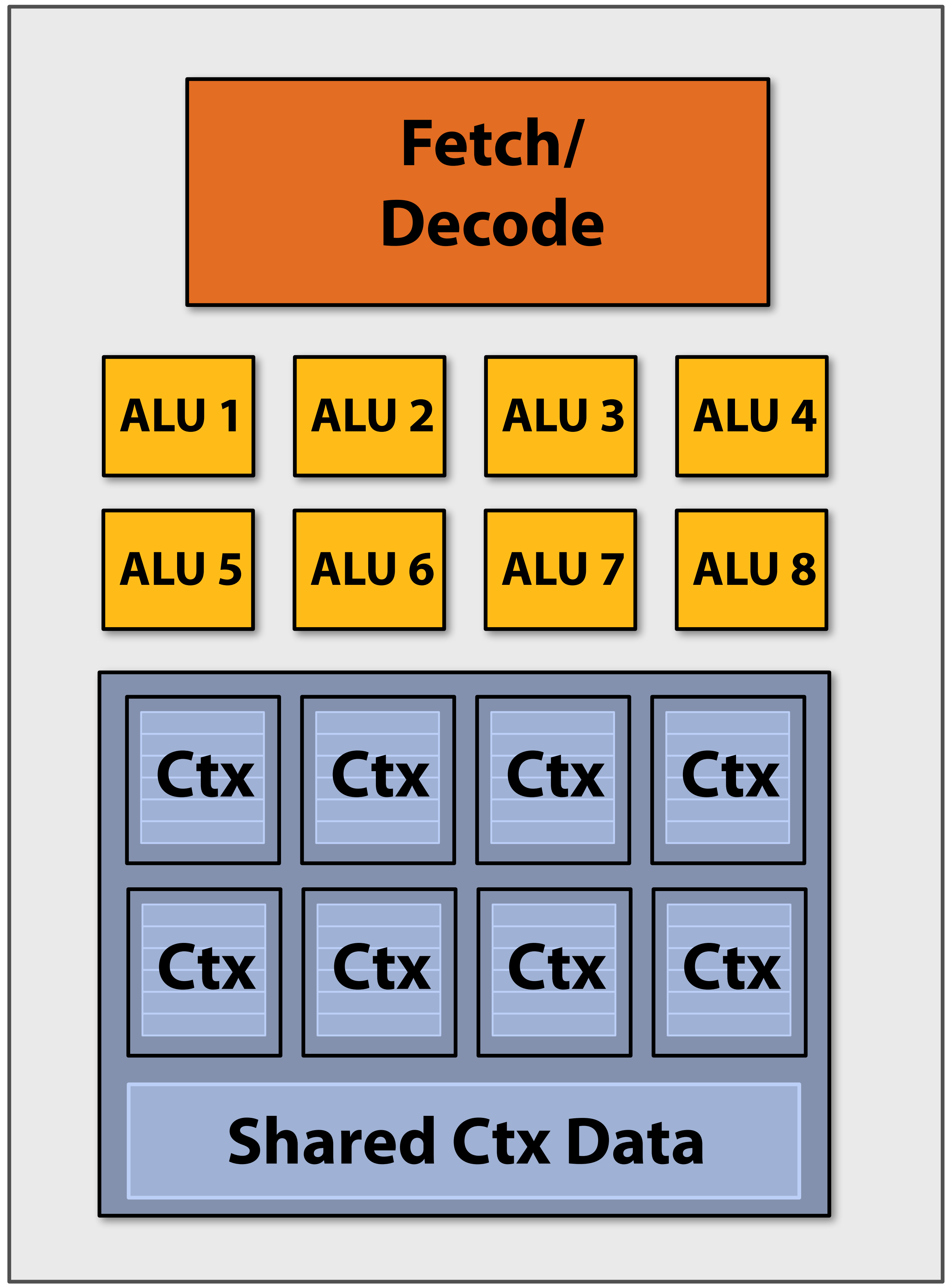
Add ALUs



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

SIMD processing

Modifying the shader

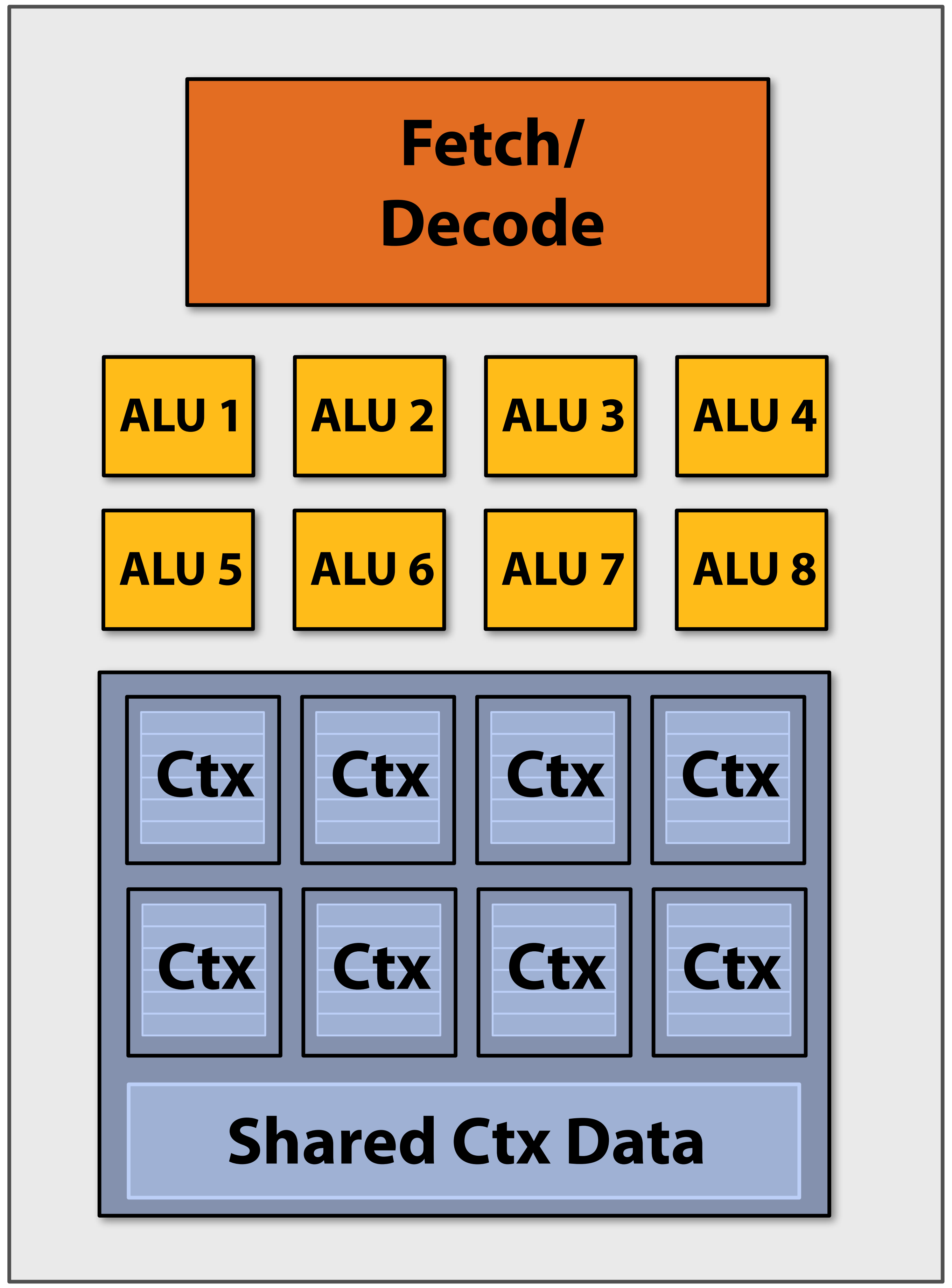


```
<diffuseShader>:
sample r0, v4, t0, s0
mul  r3, v0, cb0[0]
madd r3, v1, cb0[1], r3
madd r3, v2, cb0[2], r3
clmp r3, r3, l(0.0), l(1.0)
mul  o0, r0, r3
mul  o1, r1, r3
mul  o2, r2, r3
mov  o3, l(1.0)
```

Original compiled shader:

Processes one fragment using scalar ops on scalar registers

Modifying the shader

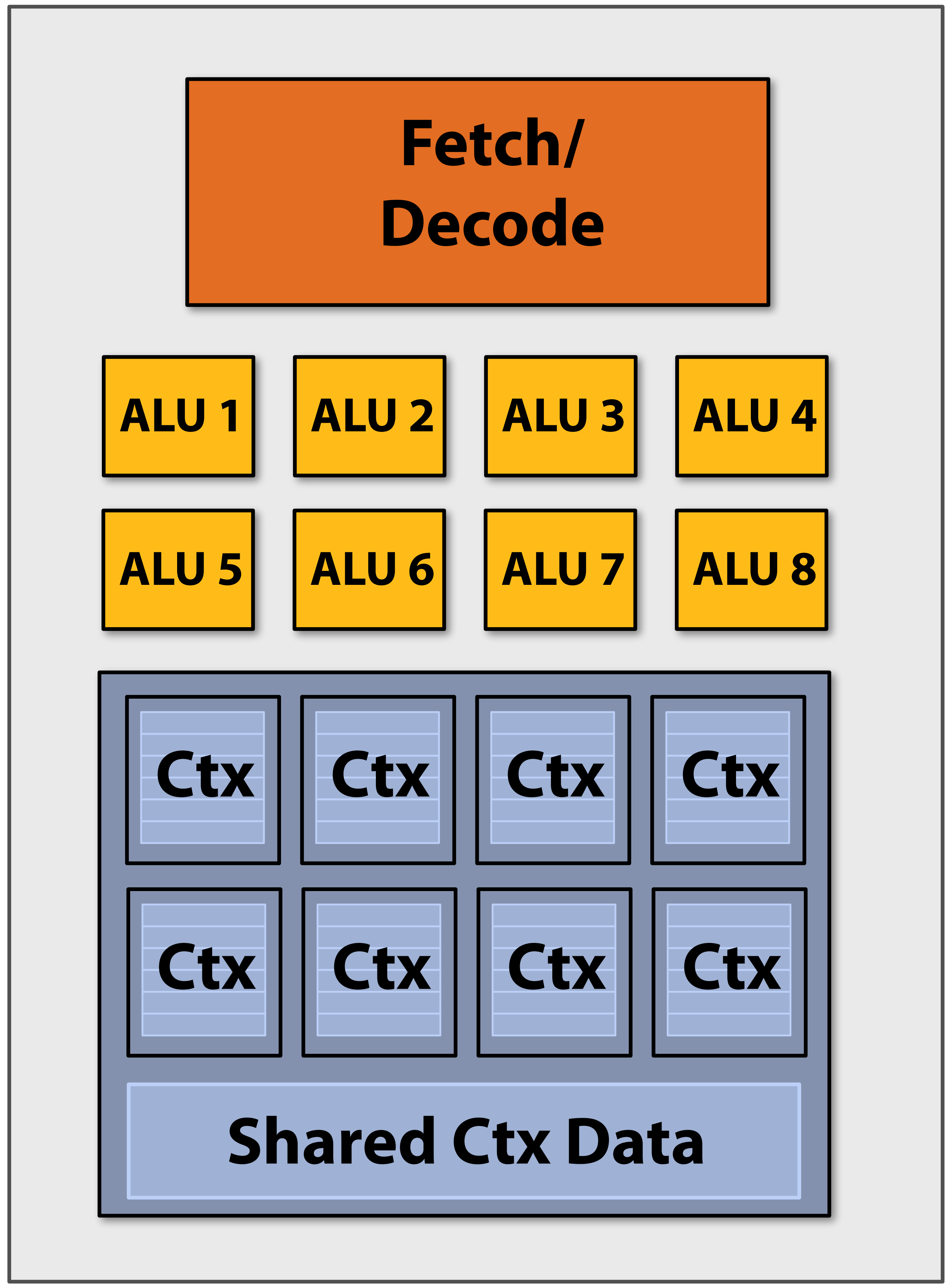


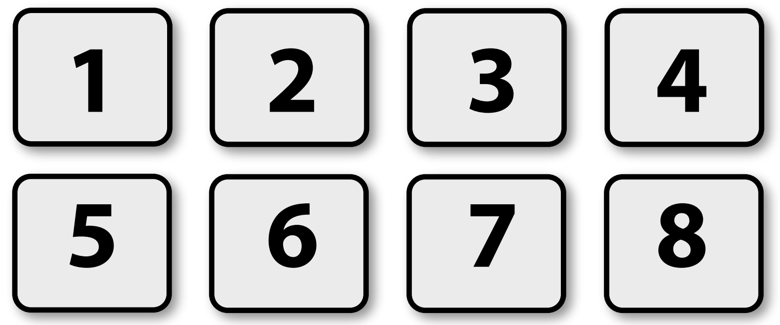
```
<VEC8_diffuseShader>:
VEC8_sample vec_r0, vec_v4, t0, vec_s0
VEC8_mul  vec_r3, vec_v0, cb0[0]
VEC8_madd vec_r3, vec_v1, cb0[1], vec_r3
VEC8_madd vec_r3, vec_v2, cb0[2], vec_r3
VEC8_clmp vec_r3, vec_r3, l(0.0), l(1.0)
VEC8_mul  vec_o0, vec_r0, vec_r3
VEC8_mul  vec_o1, vec_r1, vec_r3
VEC8_mul  vec_o2,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

New compiled shader:

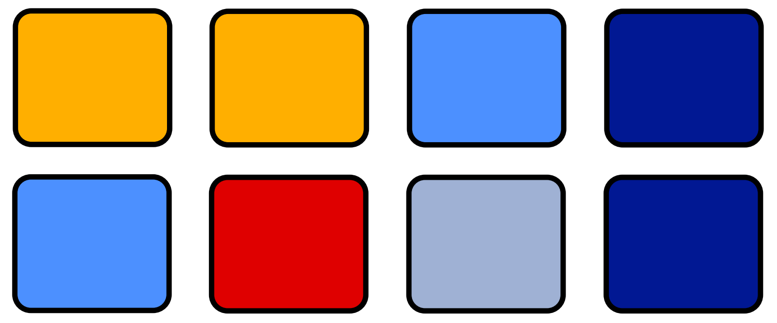
Processes eight fragments using vector ops on vector registers

Modifying the shader

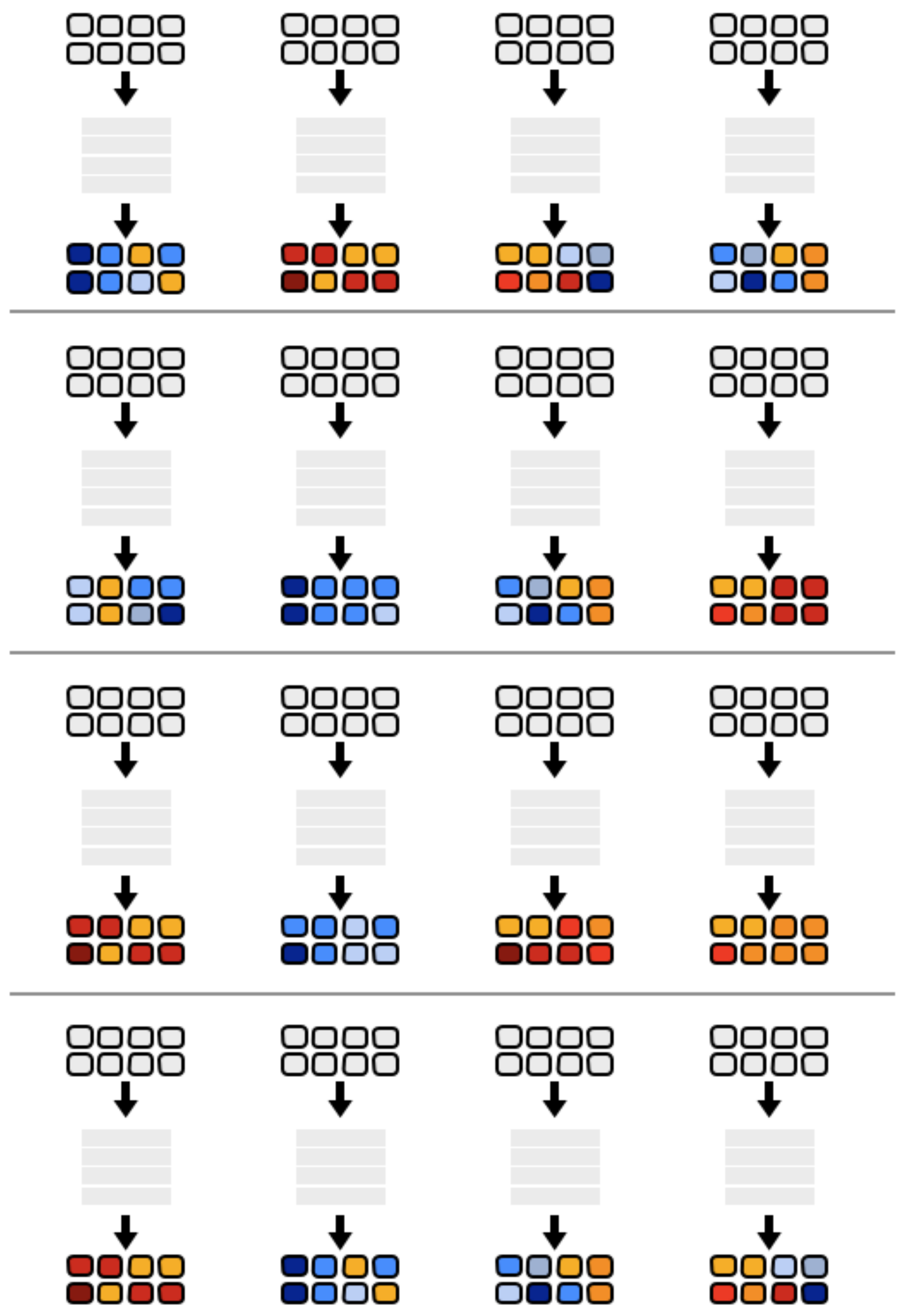


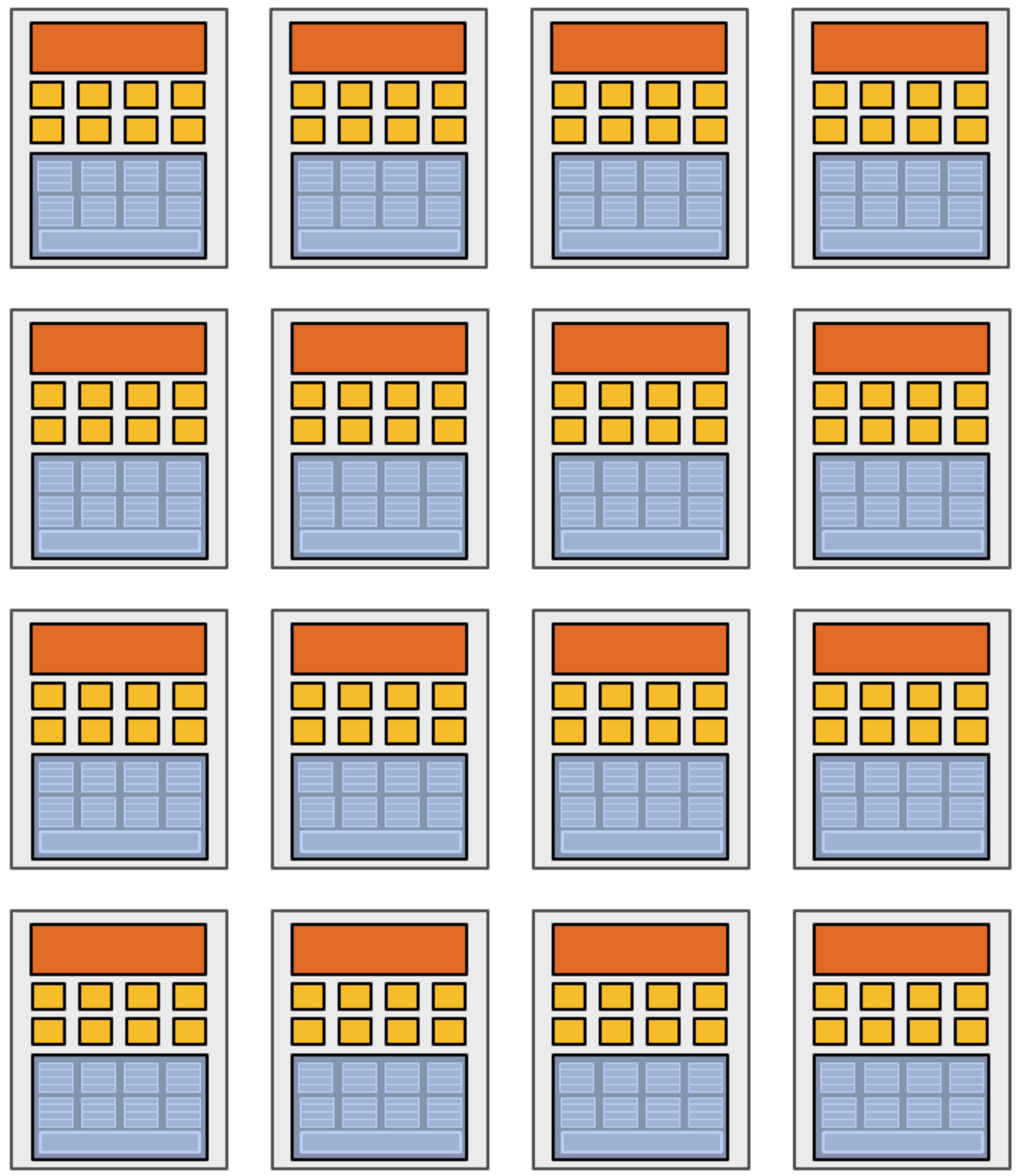


```
<VEC8_diffuseShader>:
VEC8_sample vec_r0, vec_v4, t0, vec_s0
VEC8_mul  vec_r3, vec_v0, cb0[0]
VEC8_madd vec_r3, vec_v1, cb0[1], vec_r3
VEC8_madd vec_r3, vec_v2, cb0[2], vec_r3
VEC8_clmp vec_r3, vec_r3, l(0.0), l(1.0)
VEC8_mul  vec_o0, vec_r0, vec_r3
VEC8_mul  vec_o1, vec_r1, vec_r3
VEC8_mul  vec_o2,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



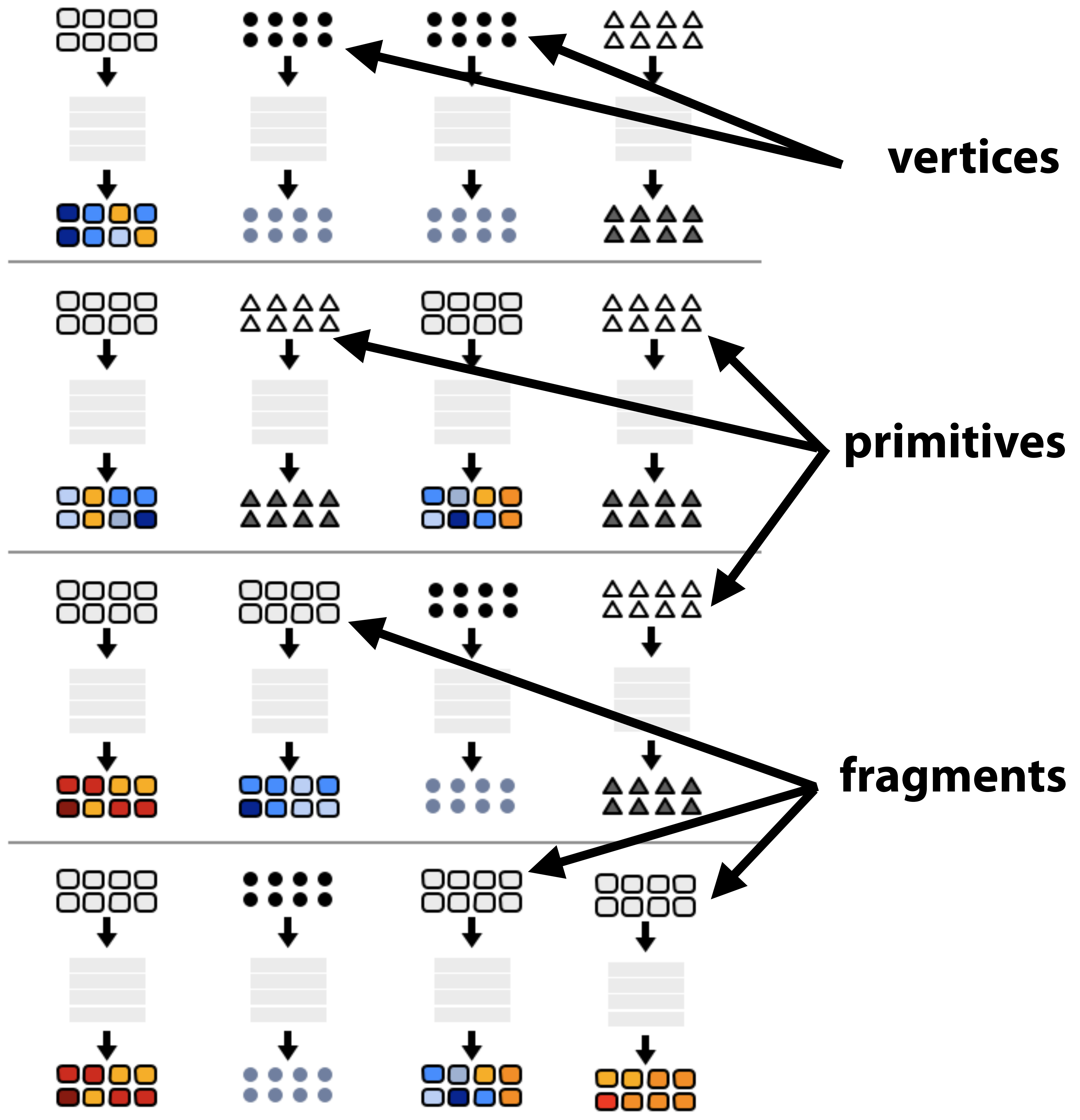
128 fragments in parallel

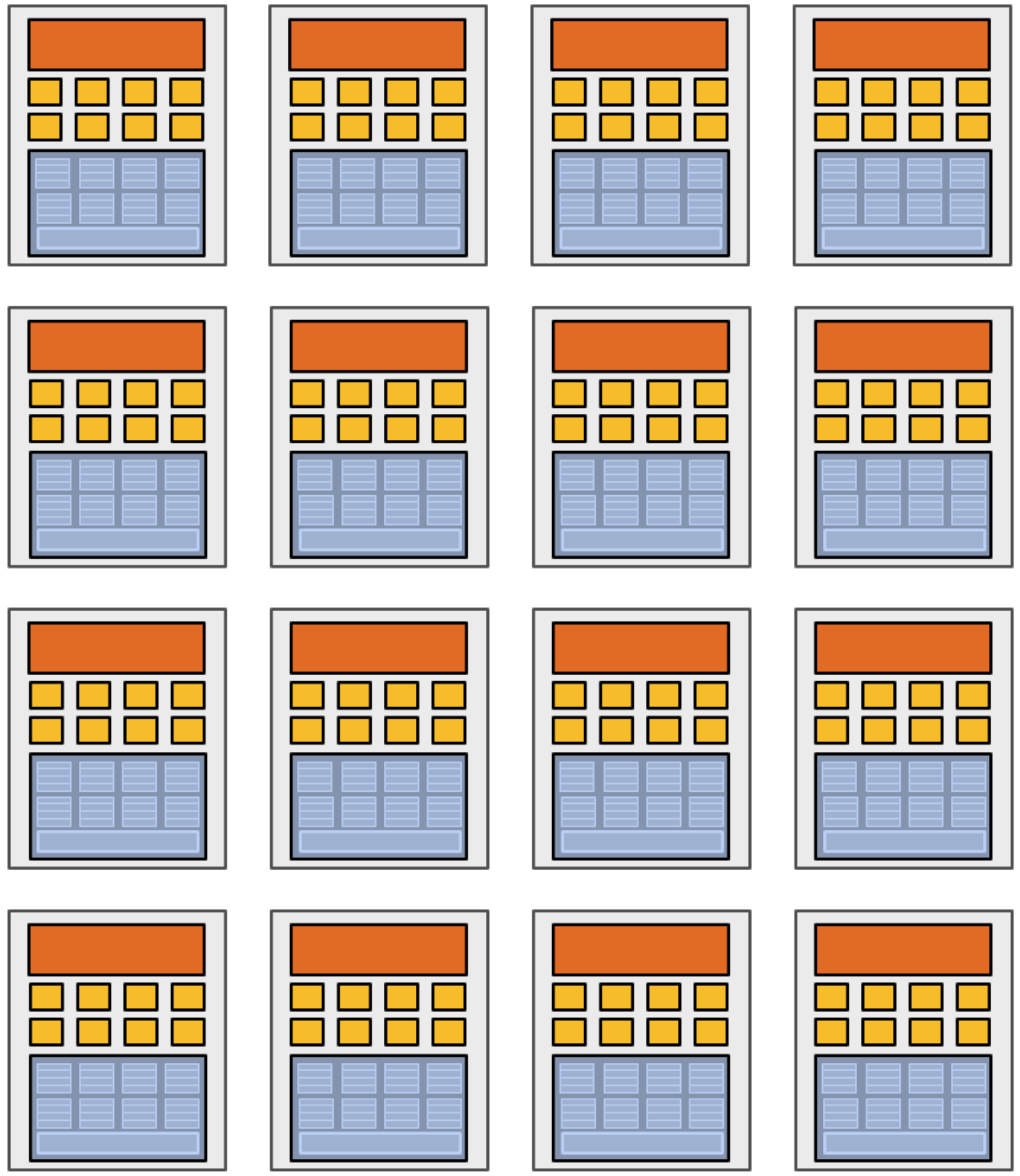




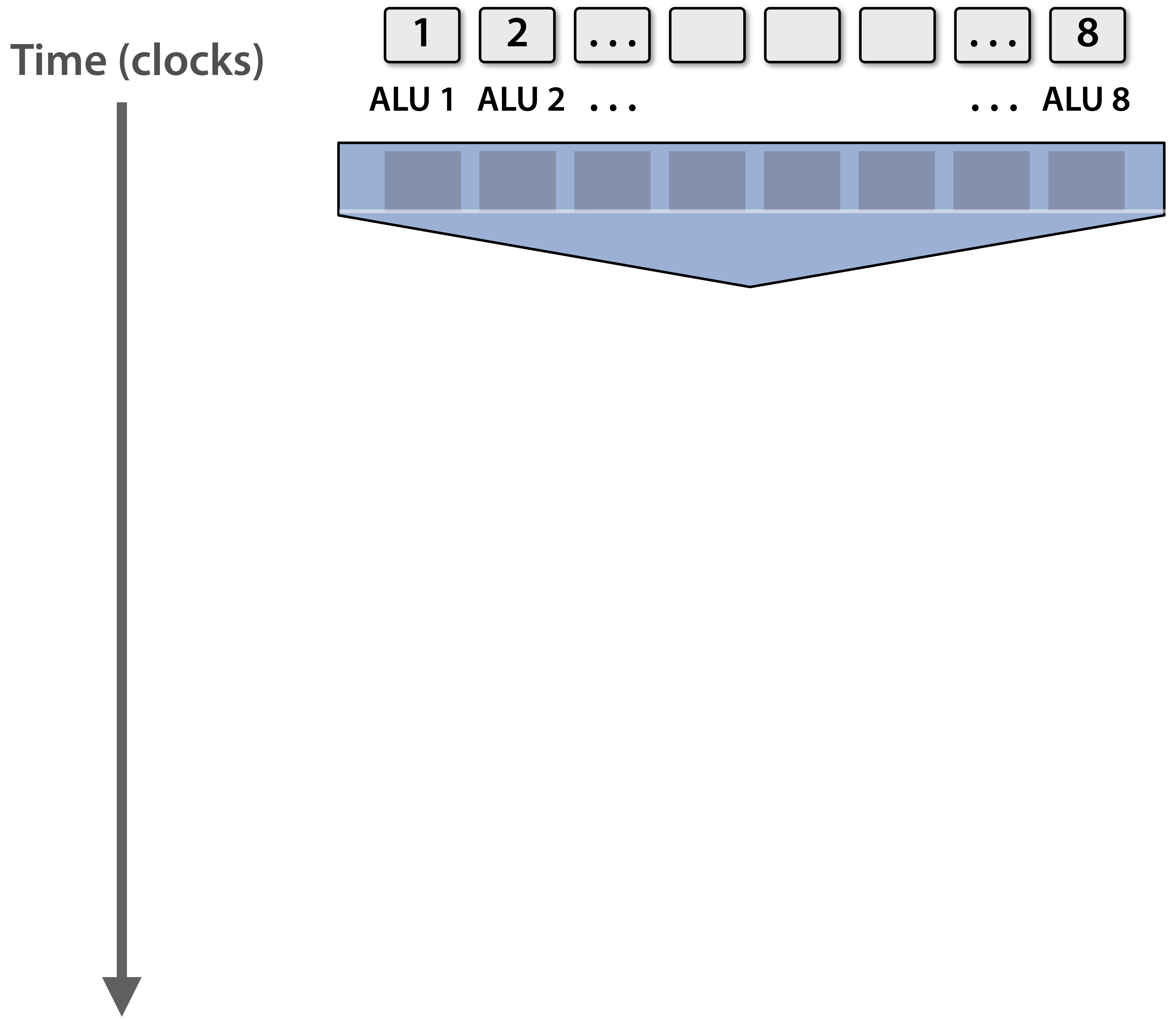
16 cores = 128 ALUs , 16 simultaneous instruction streams

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



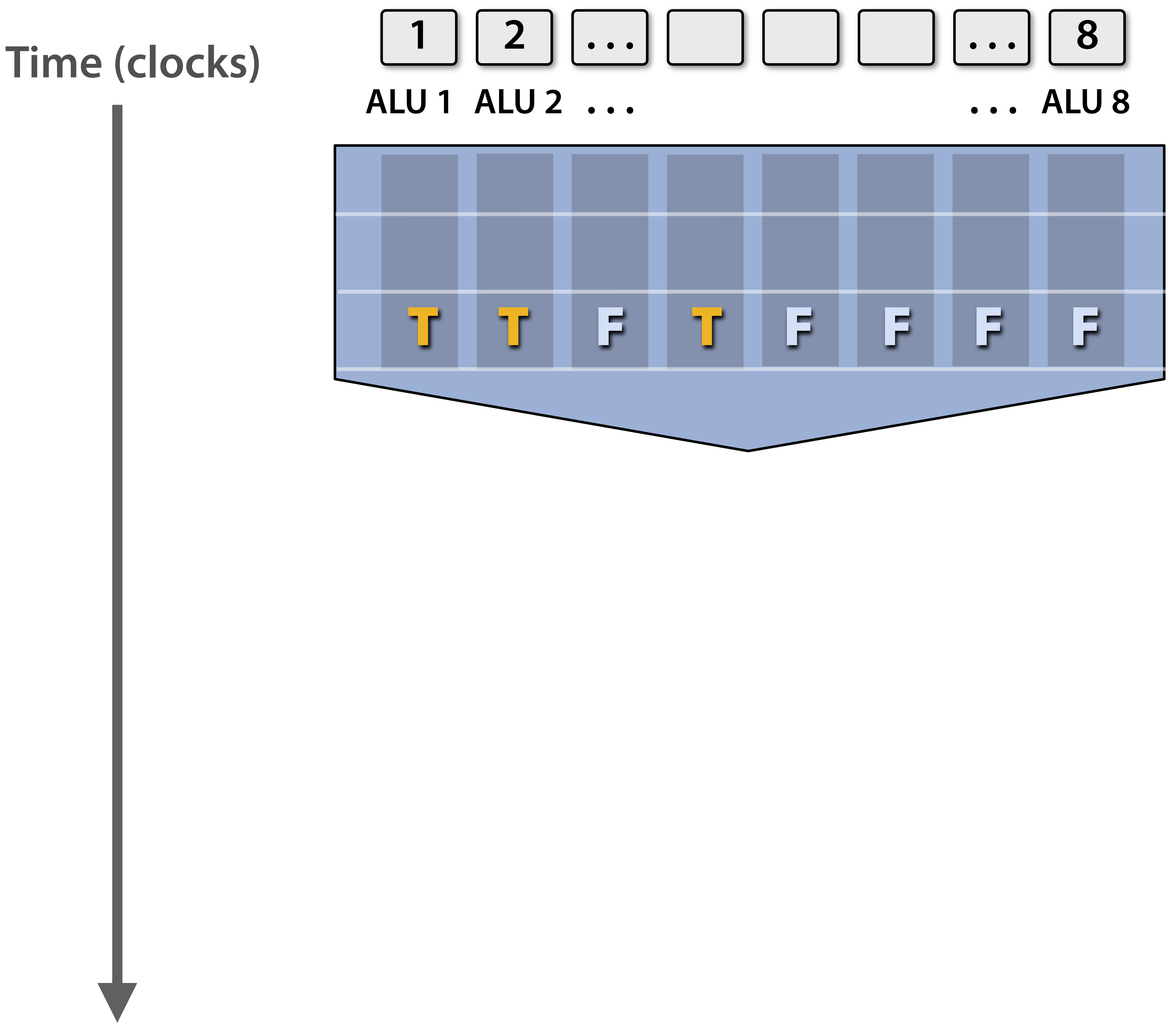


But what about branches?



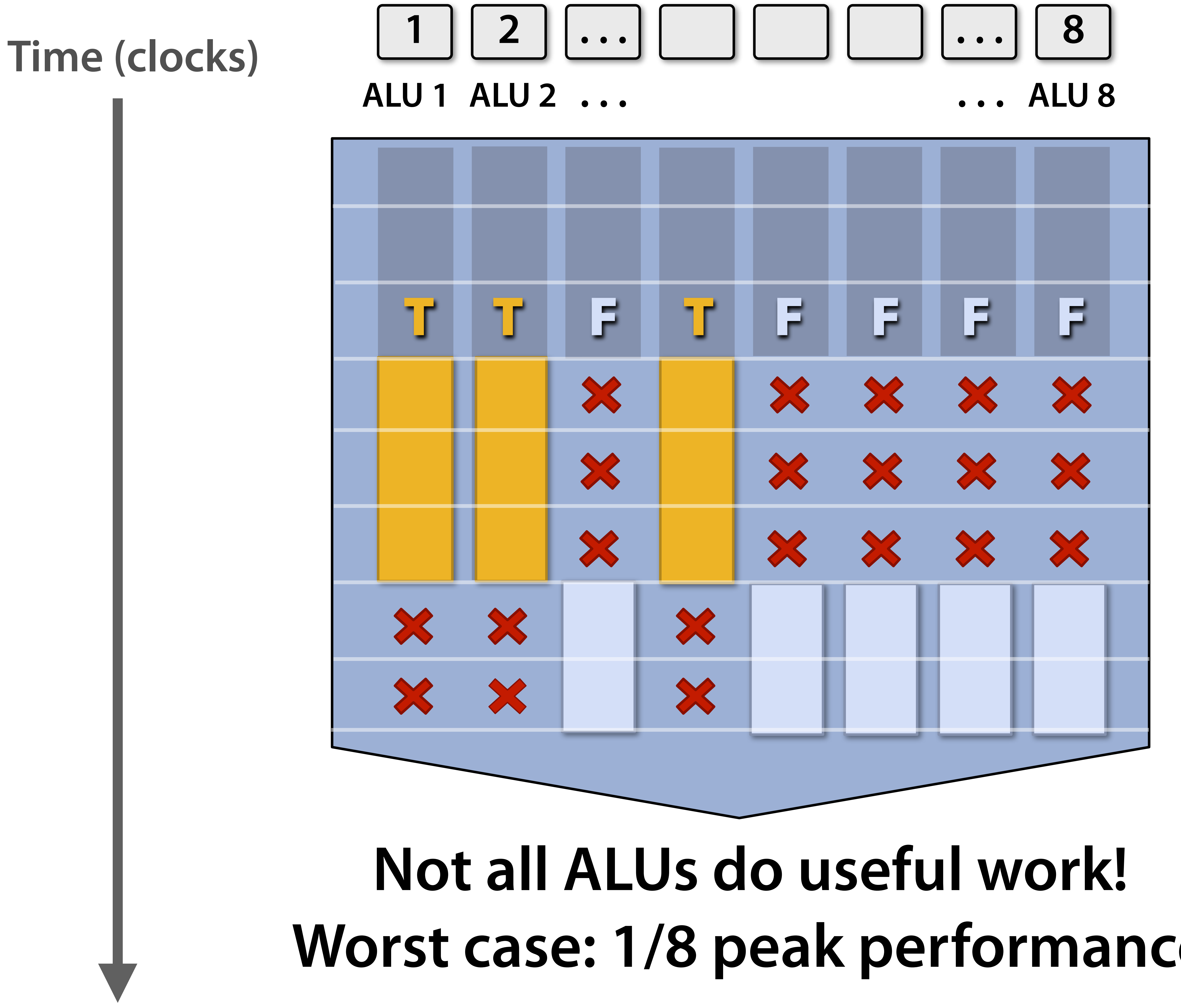
```
<unconditional
 shader code>

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>
```

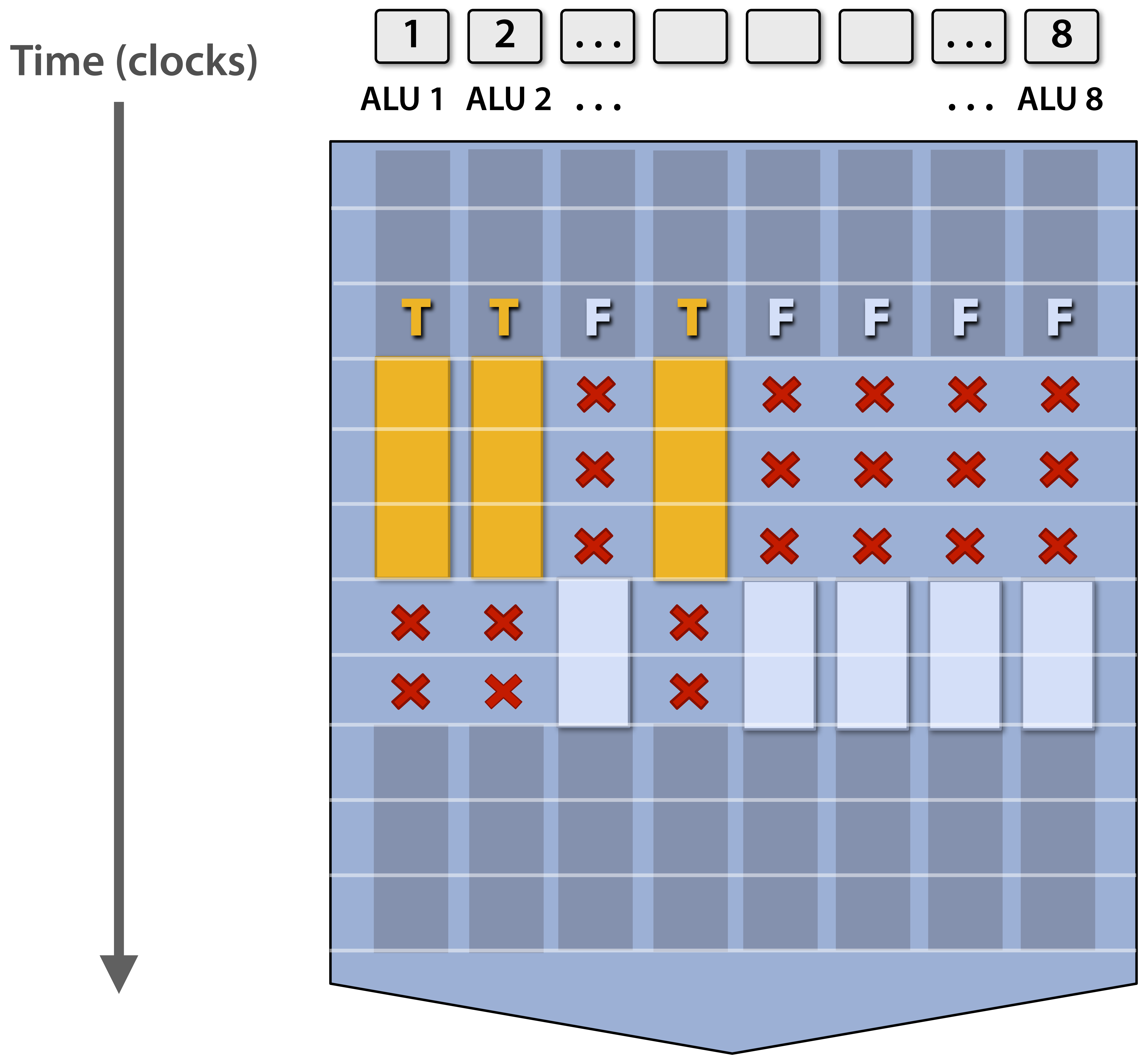
But what about branches?



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

```
<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?

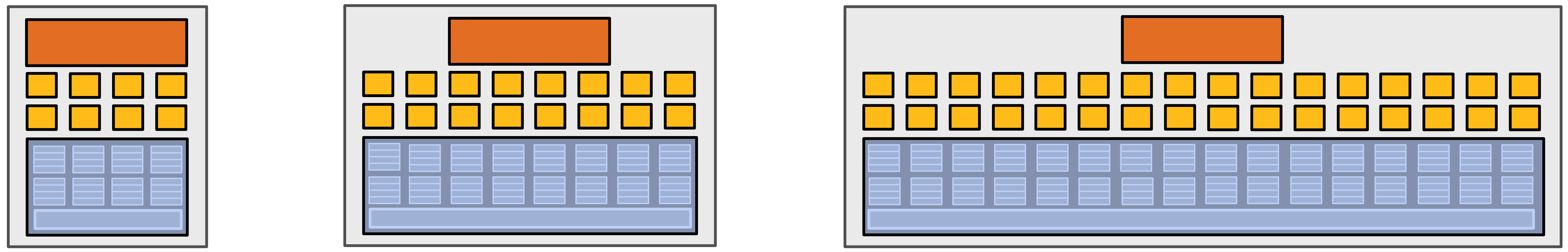


```
<unconditional
 shader code>

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

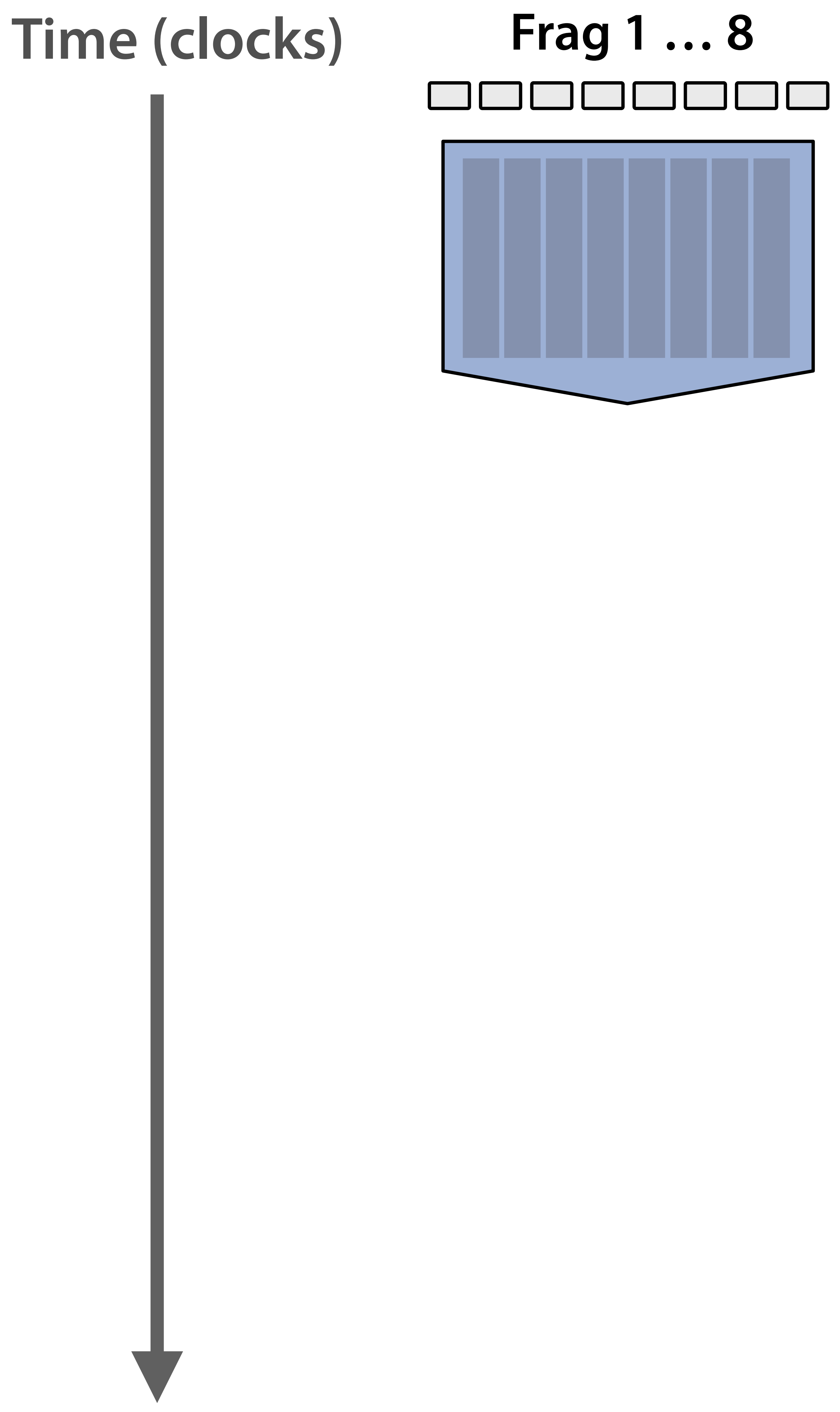
We’ve removed the fancy caches and logic that helps avoid stalls.

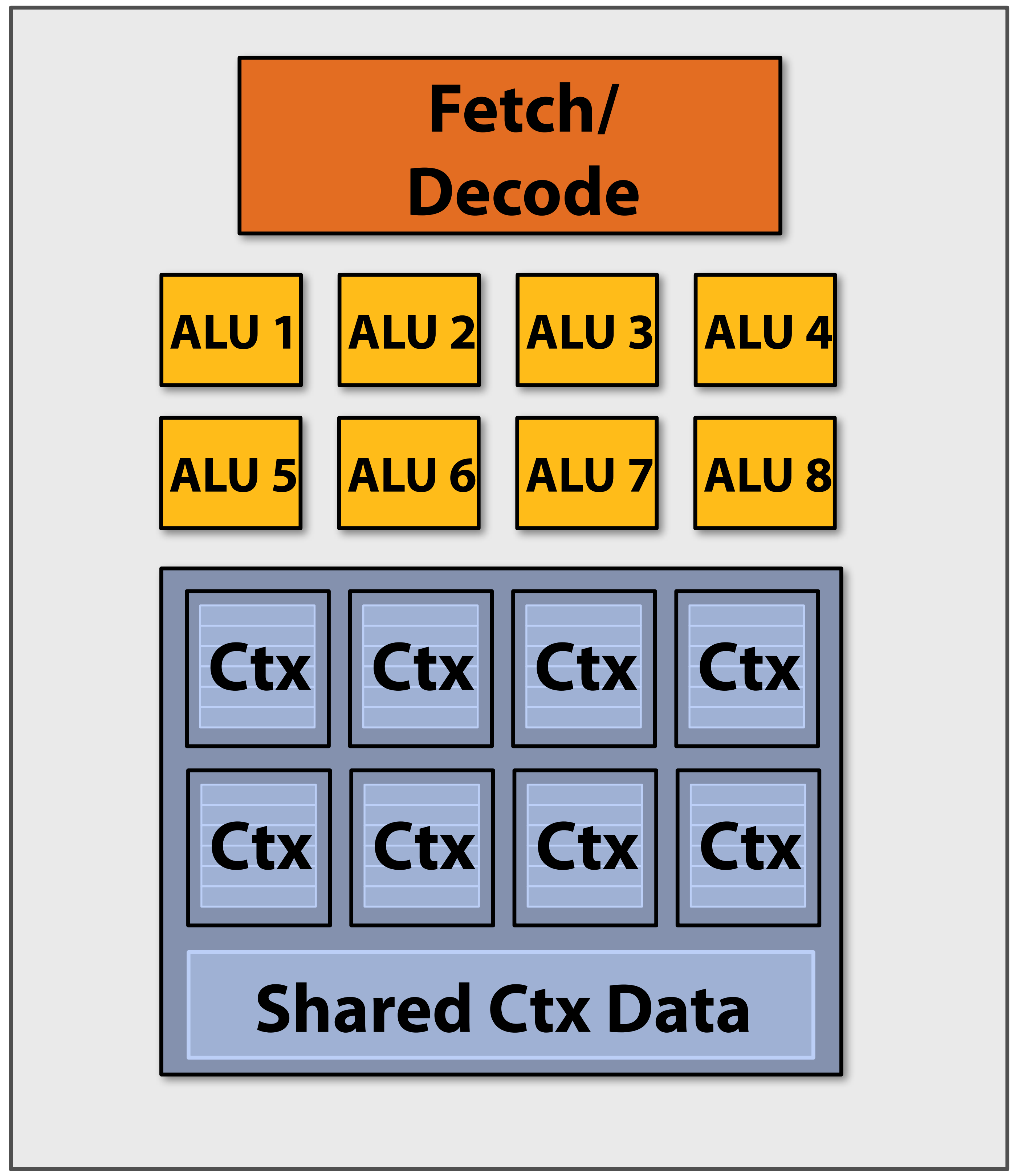
But we have **LOTS** of independent fragments.

Idea #3:

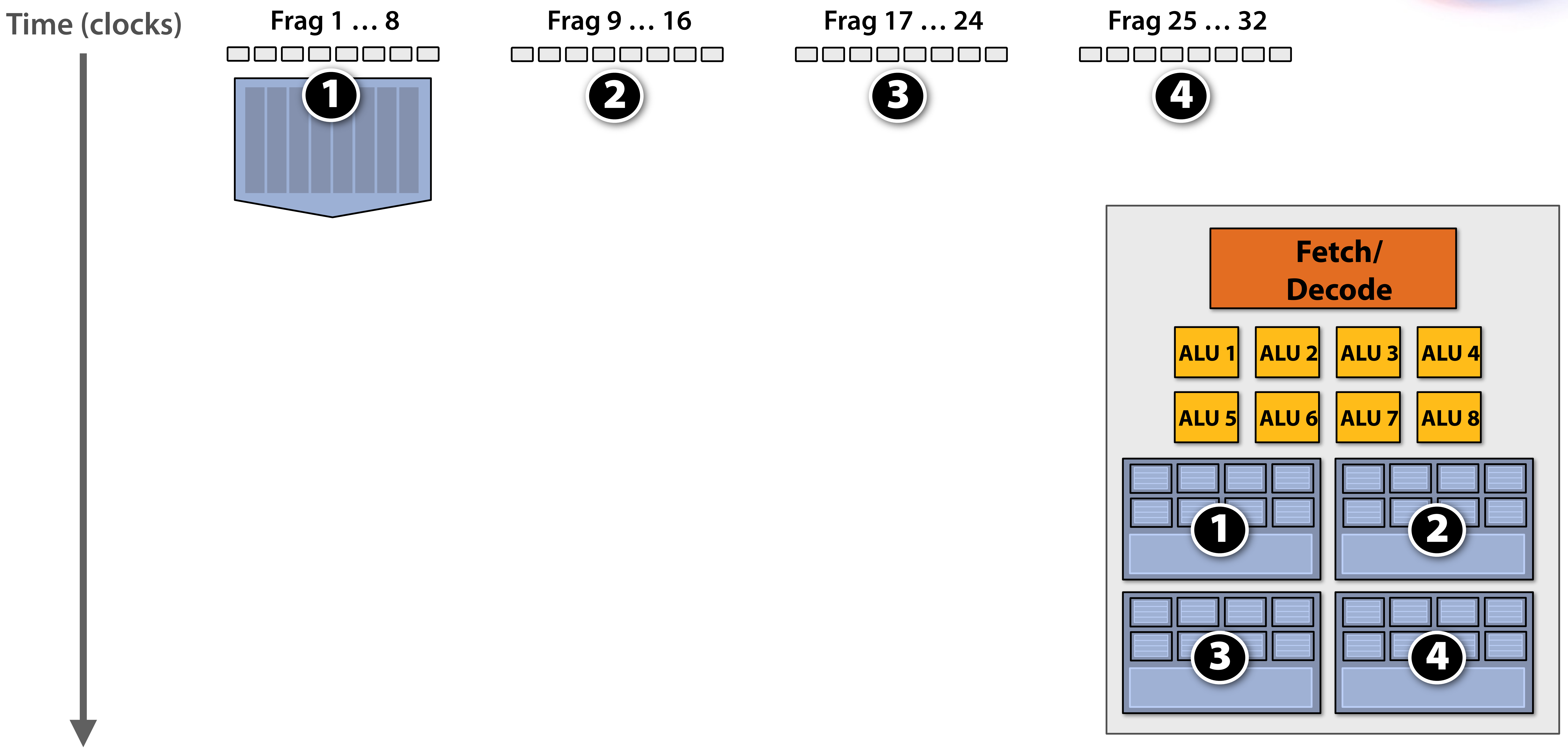
Interleave processing of many fragments on a single core to avoid stalls caused by high latency operations.

Hiding shader stalls

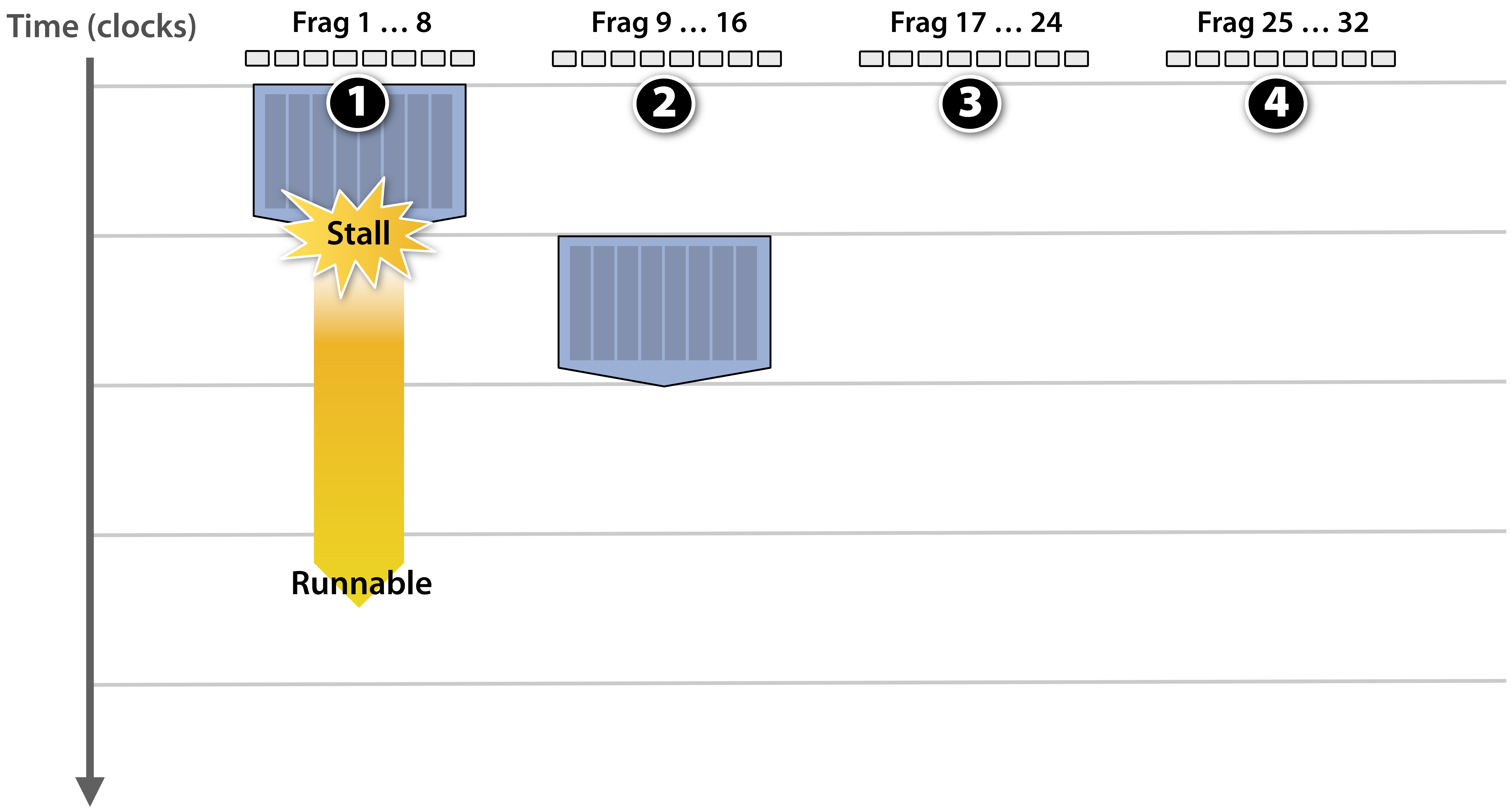




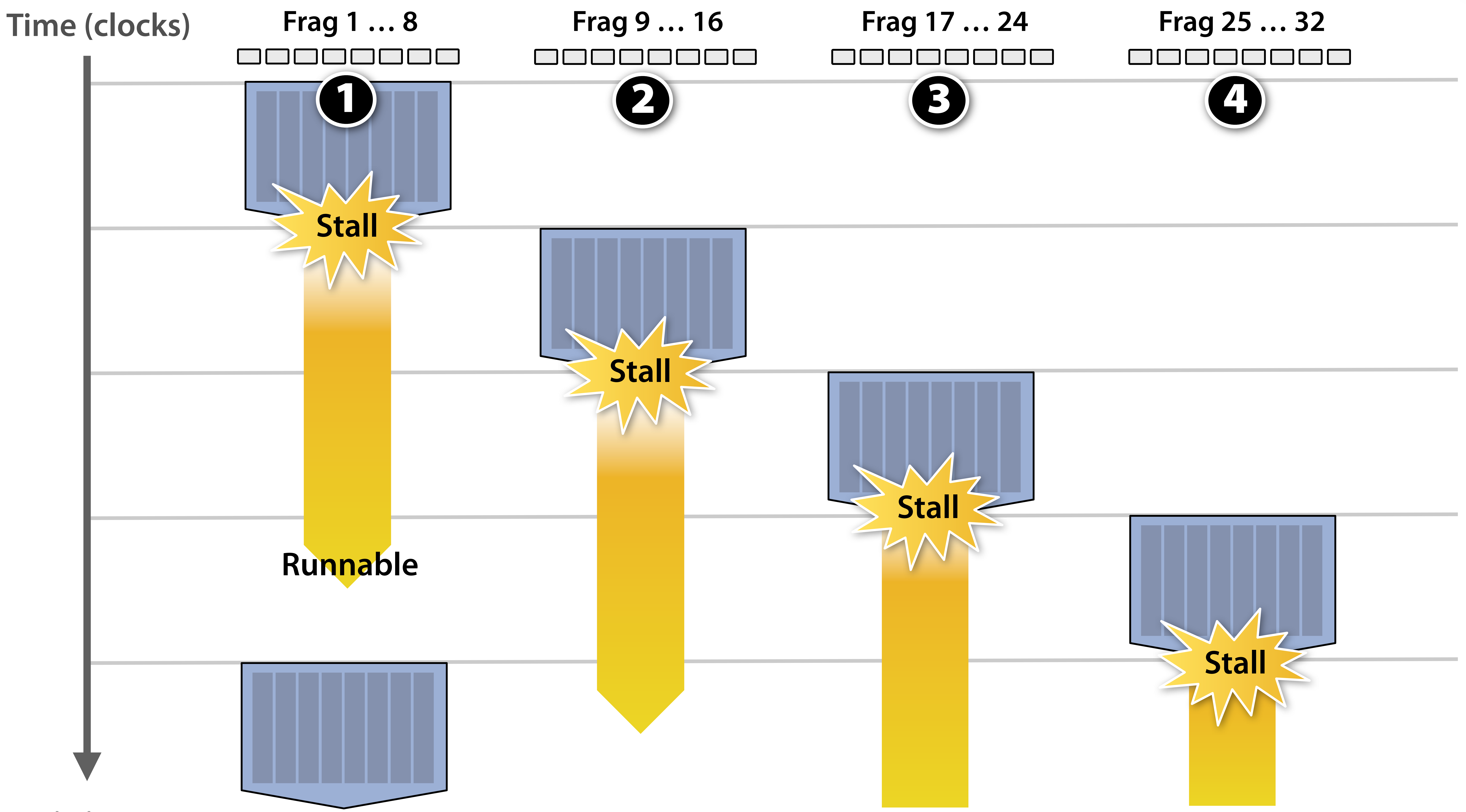
Hiding shader stalls



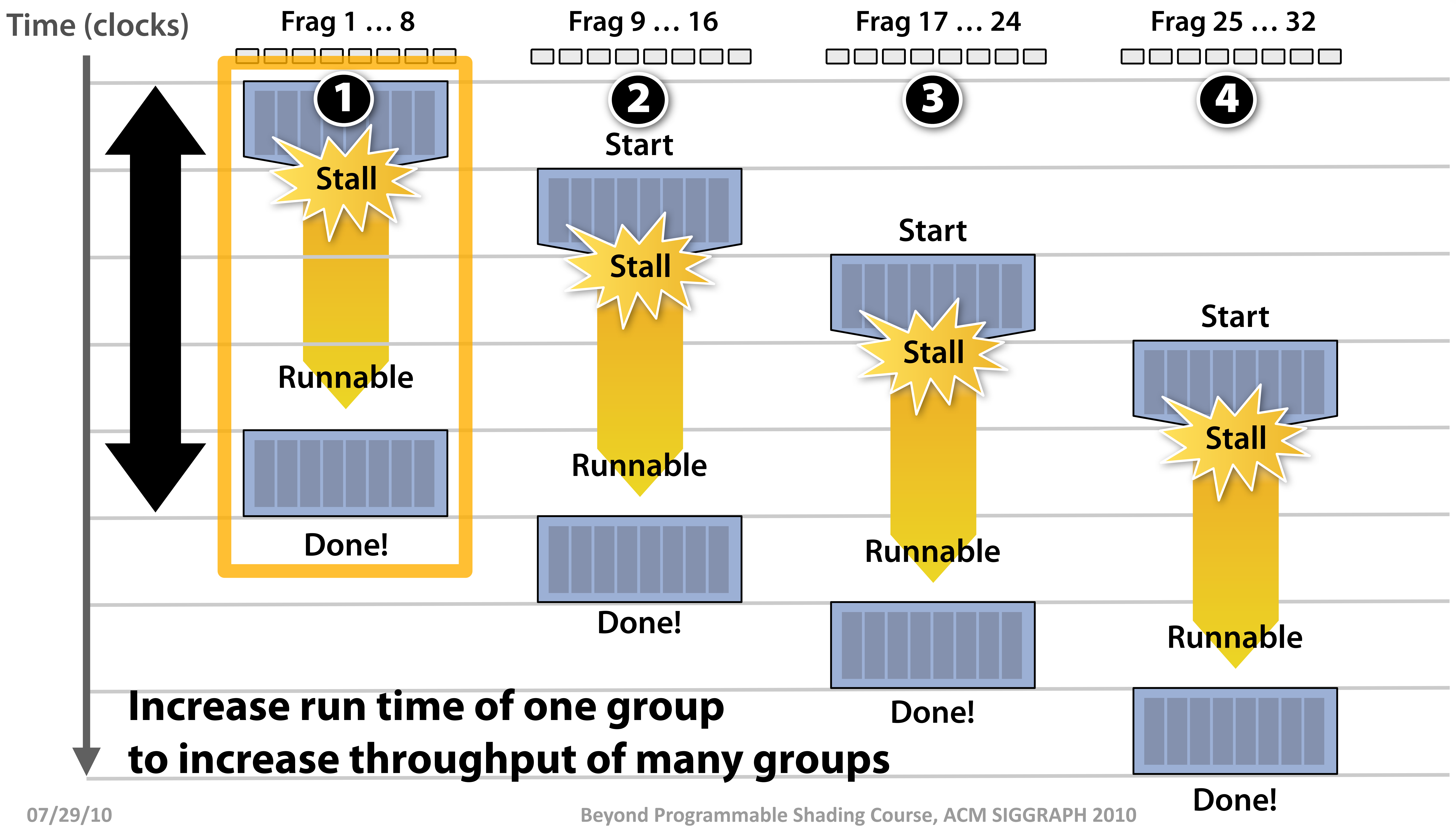
Hiding shader stalls



Hiding shader stalls

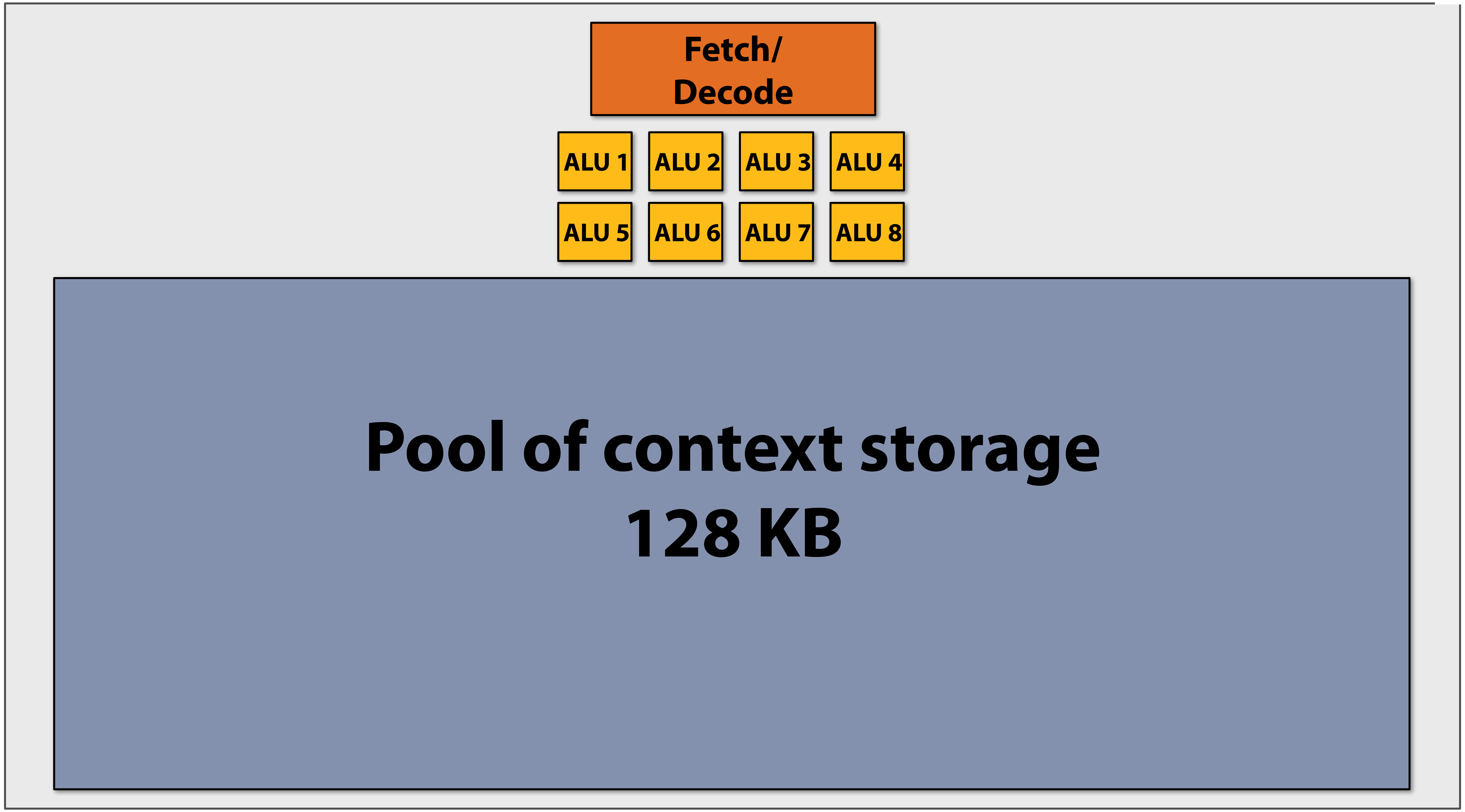


Throughput!

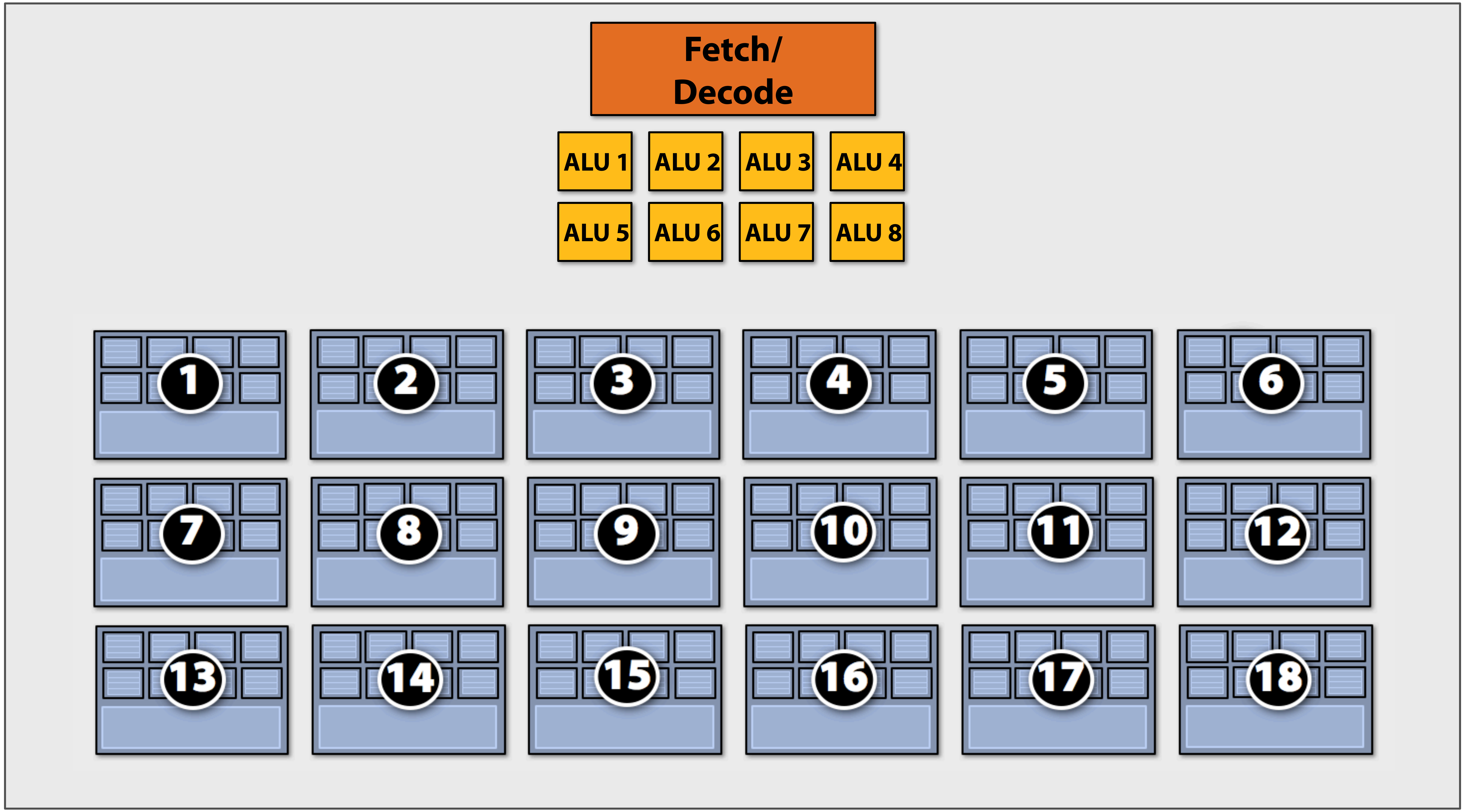


Increase run time of one group to increase throughput of many groups

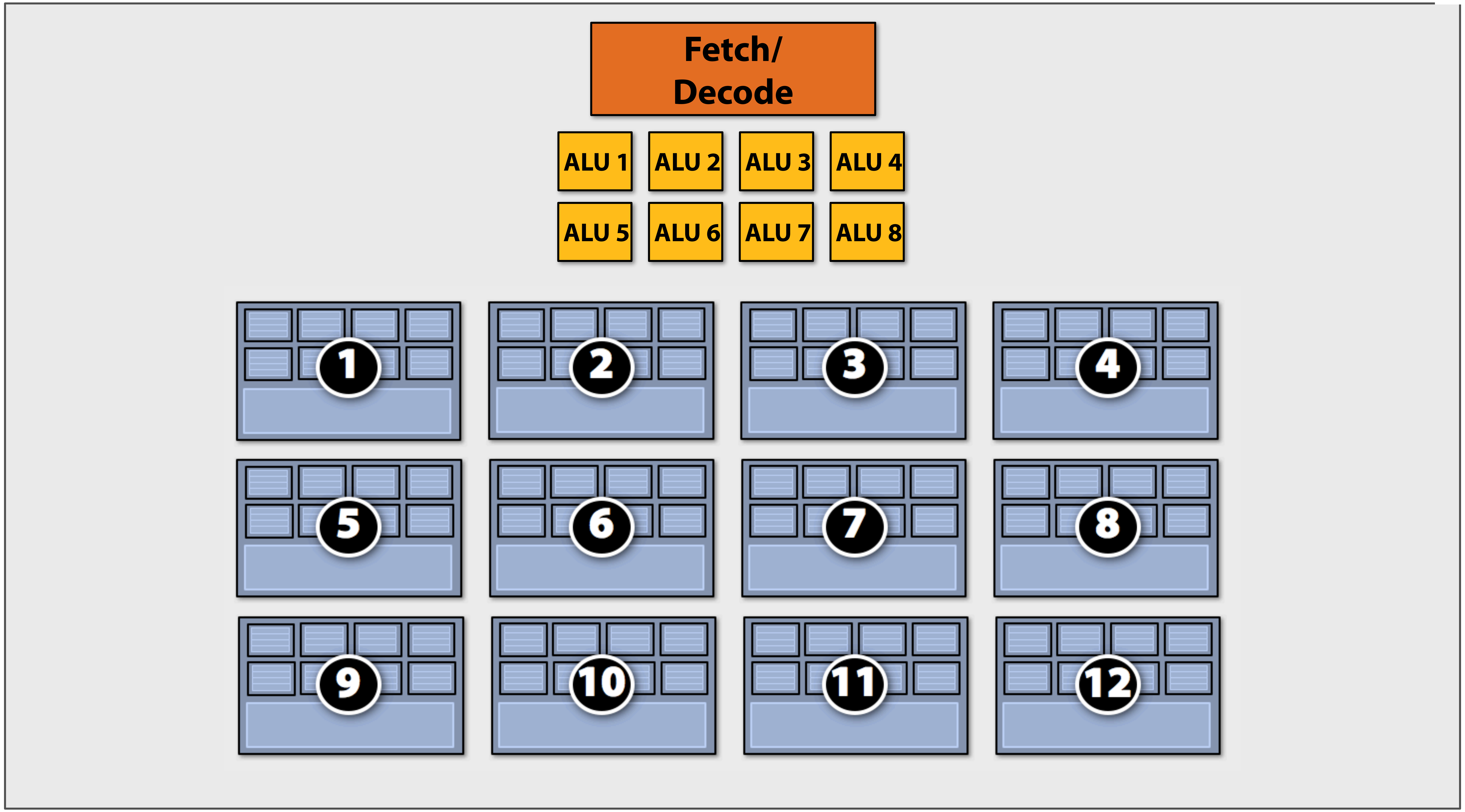
Storing contexts



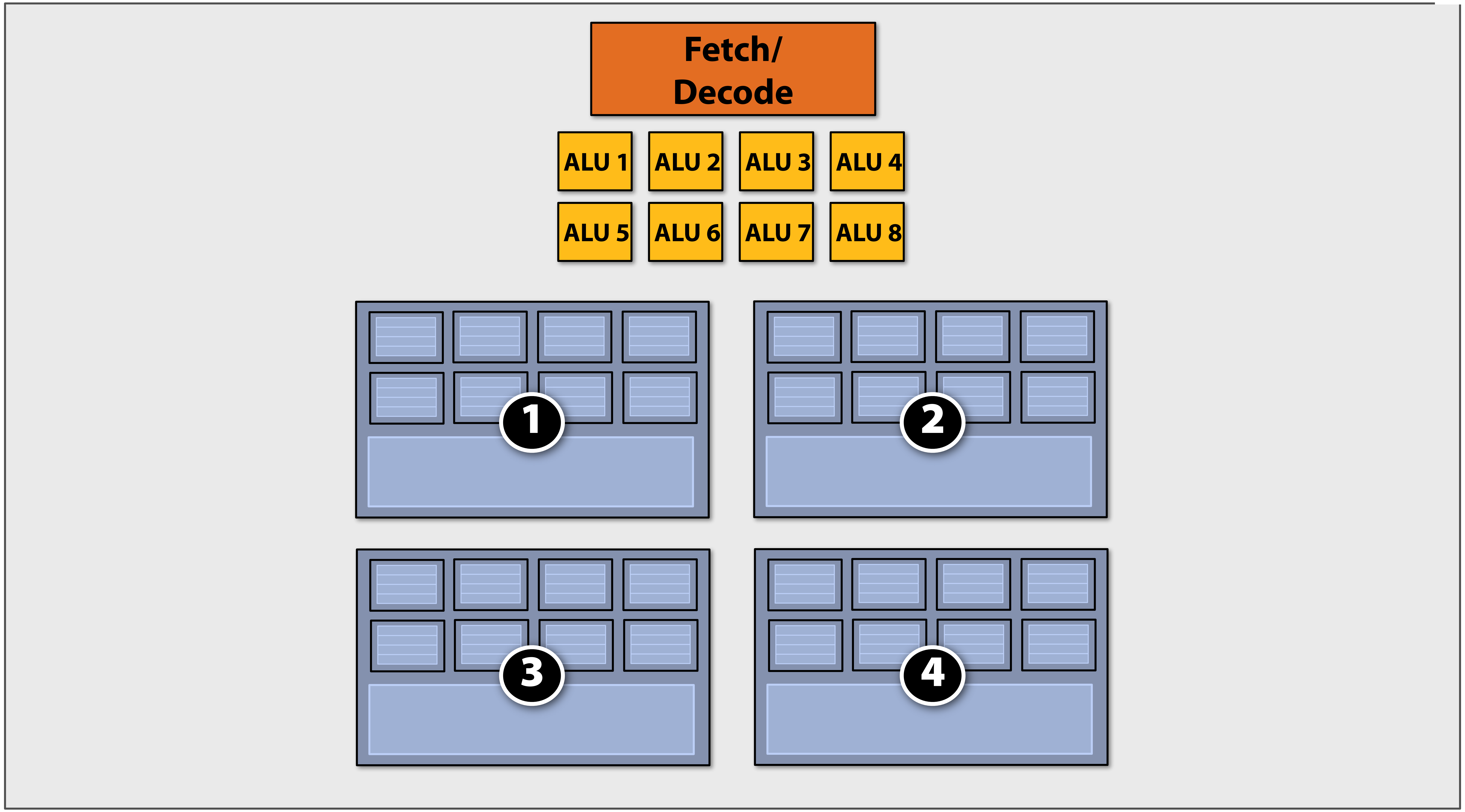
Eighteen small contexts (maximal latency hiding)



Twelve medium contexts



Four large contexts (low latency hiding ability)



Clarification

Interleaving between contexts can be managed by hardware or software (or both!)

- NVIDIA / ATI Radeon GPUs
  - HW schedules / manages all contexts (lots of them)
  - Special on-chip storage holds fragment state
- Intel Larrabee
  - HW manages four x86 (big) contexts at fine granularity
  - SW scheduling interleaves many groups of fragments on each HW context
  - L1-L2 cache holds fragment state (as determined by SW)

My chip!

16 cores

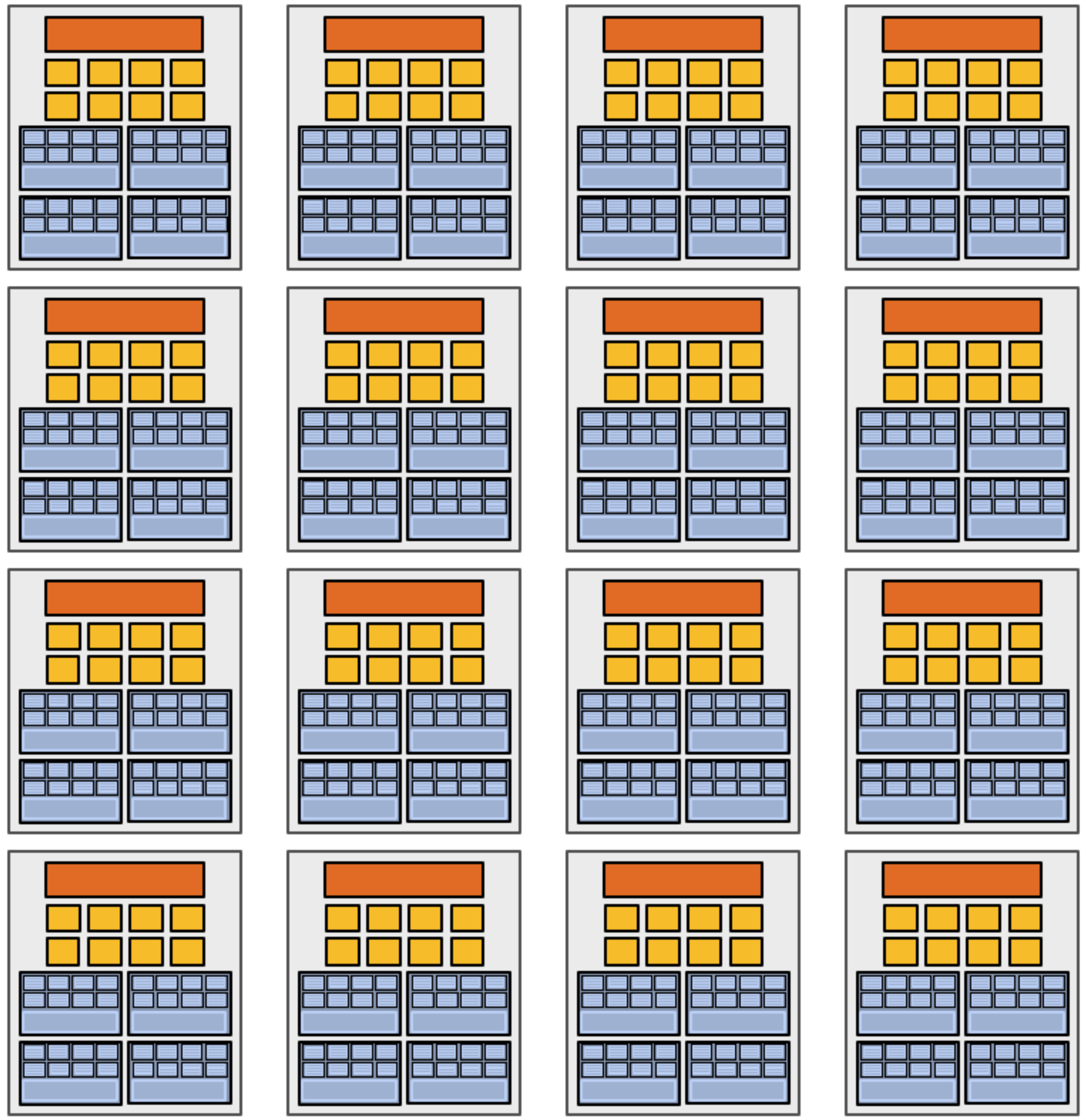
8 mul-add ALUs per core
(128 total)

16 simultaneous
instruction streams

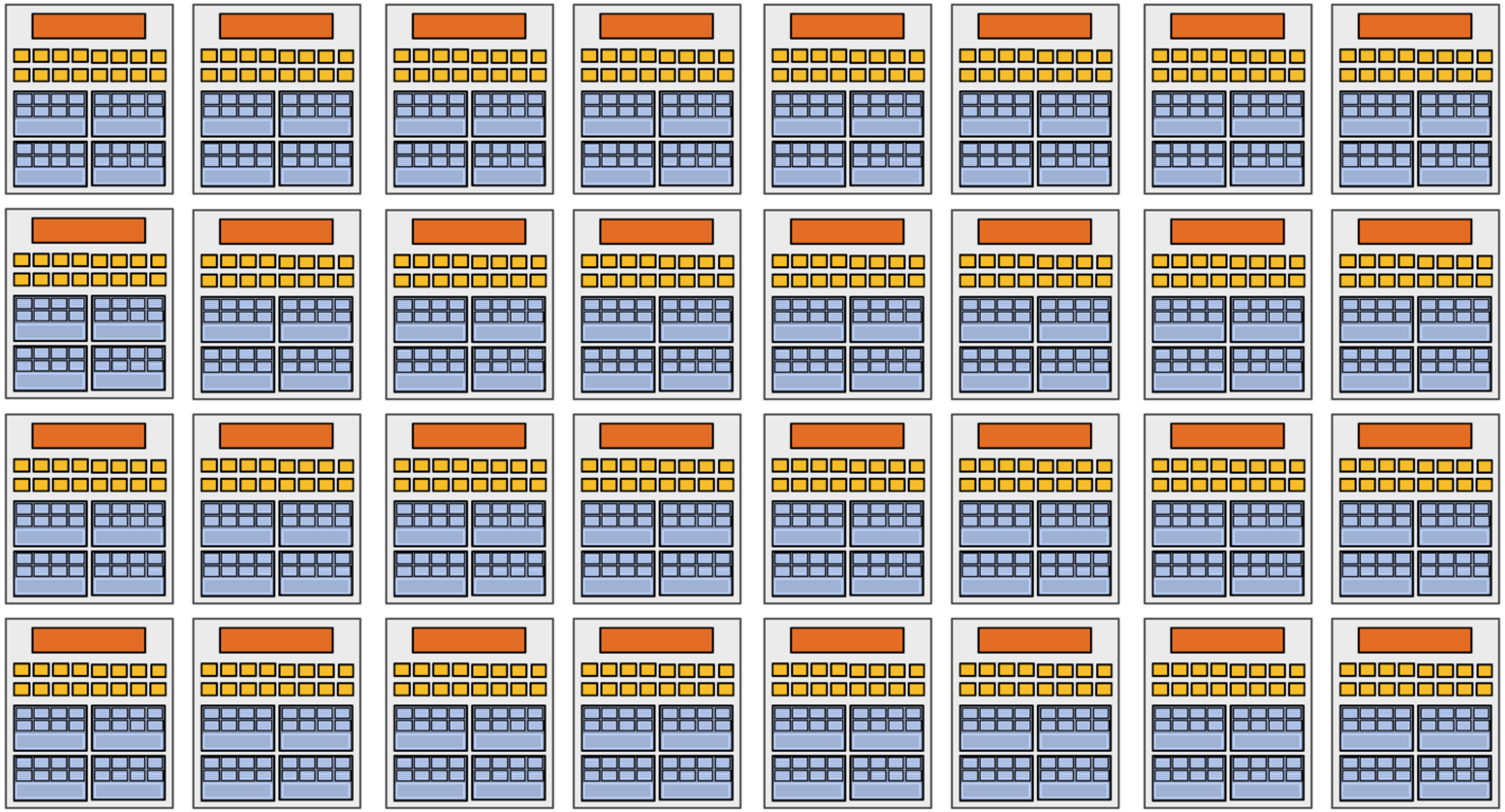
64 concurrent (but interleaved)
instruction streams

512 concurrent fragments

= 256 GFLOPs   (@ 1GHz)



My "enthusiast" chip!



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

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)
   - 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

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

NVIDIA GeForce GTX 480
ATI Radeon HD 5870

Disclaimer

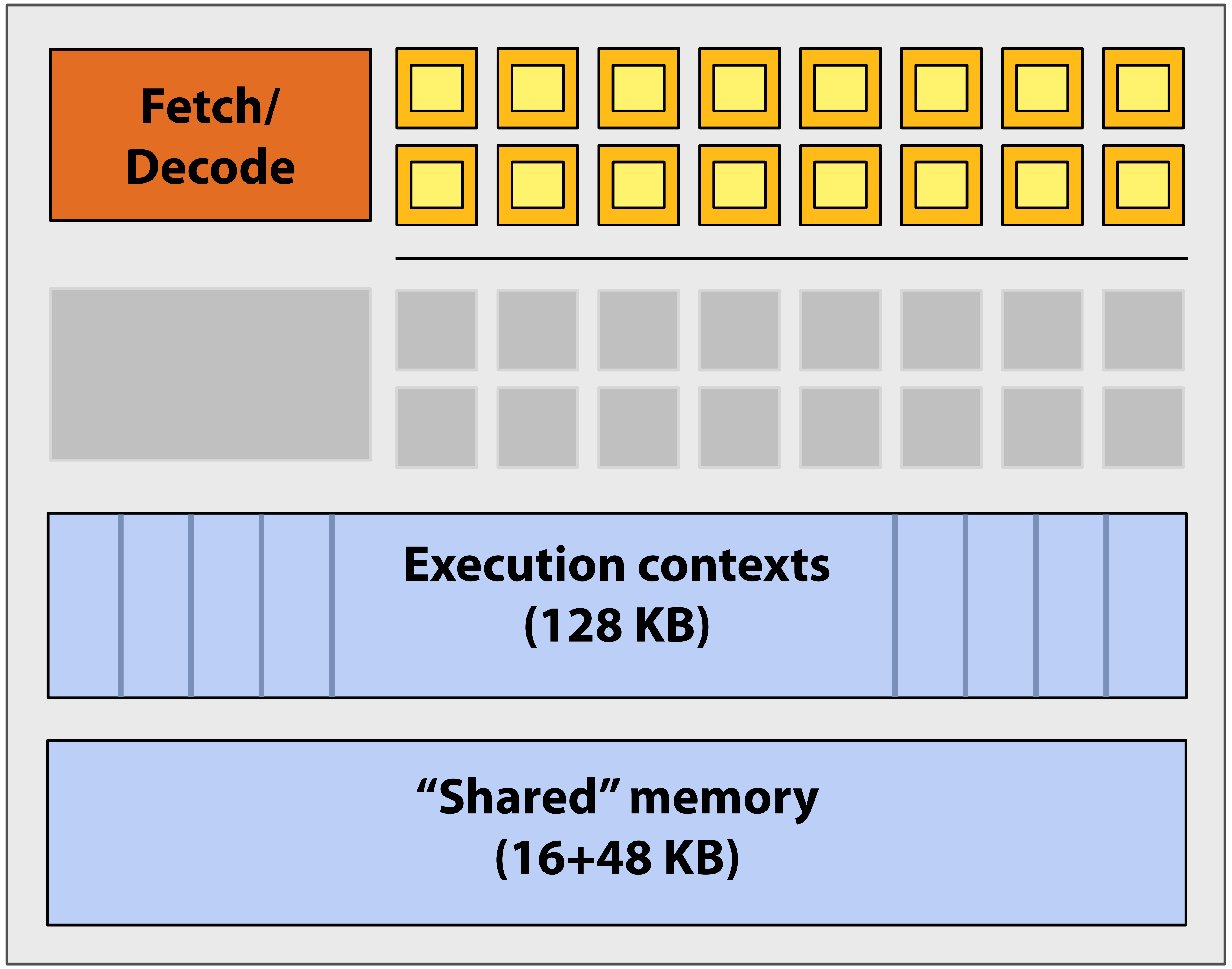
- The following slides describe "a reasonable way to think" about the architecture of commercial GPUs
- Many factors play a role in actual chip performance

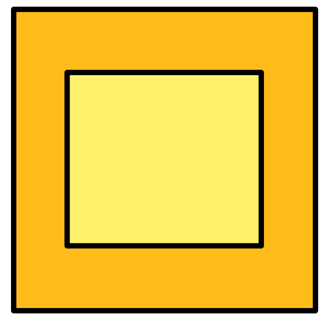
NVIDIA GeForce GTX 480 (Fermi)

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

- Generic speak:
  - 15 cores
  - 2 groups of 16 SIMD functional units per core

NVIDIA GeForce GTX 480 “core”

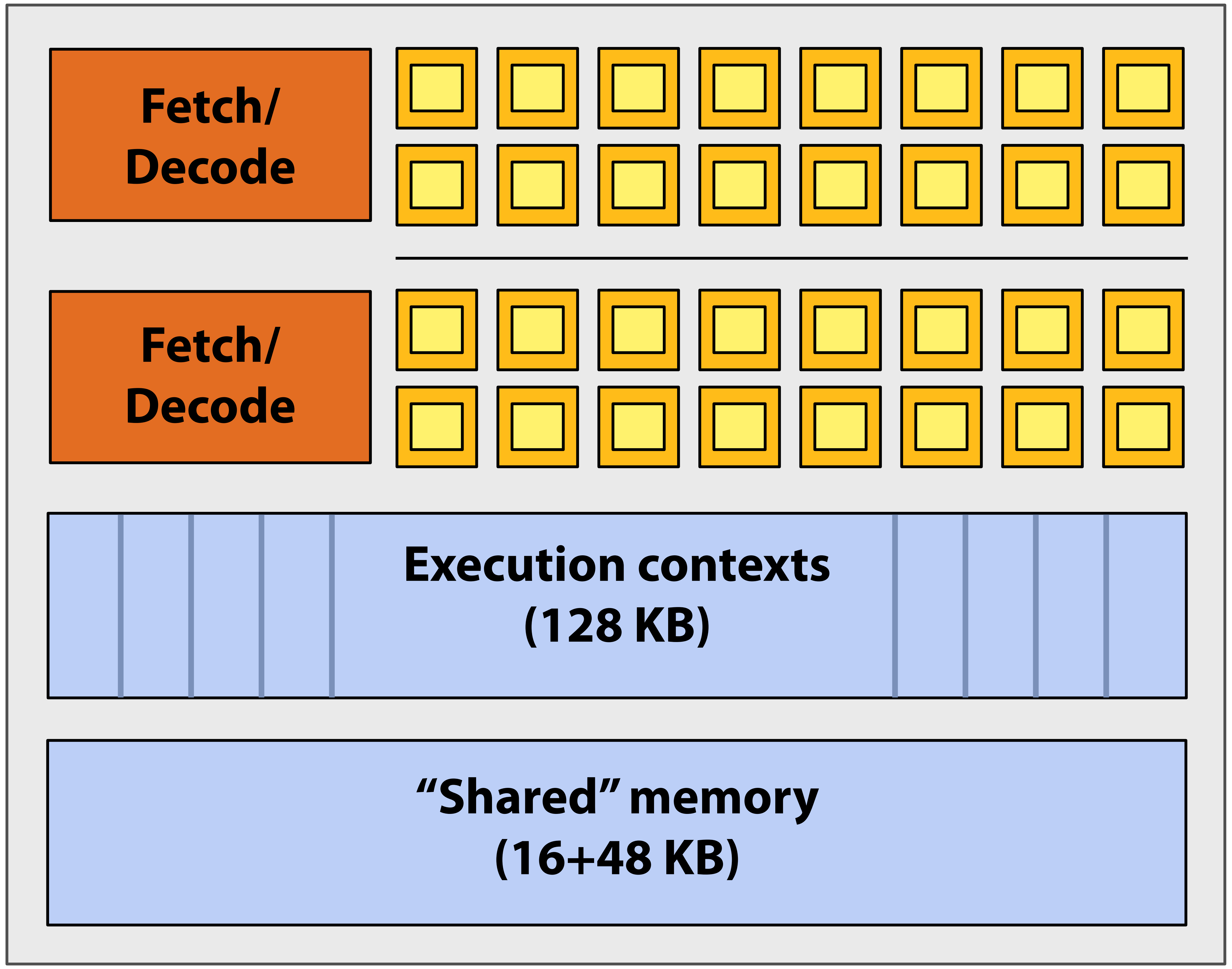


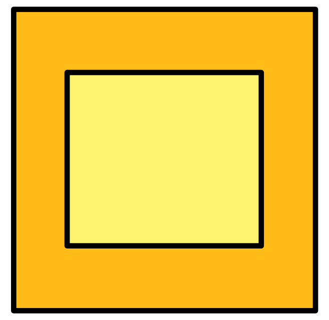


- Groups of 32 [fragments/vertices/CUDA threads] share an instruction stream
- Up to 48 groups are simultaneously interleaved
- Up to 1536 individual contexts can be stored

Source: Fermi Compute Architecture Whitepaper
CUDA Programming Guide 3.1, Appendix G

NVIDIA GeForce GTX 480 "core"

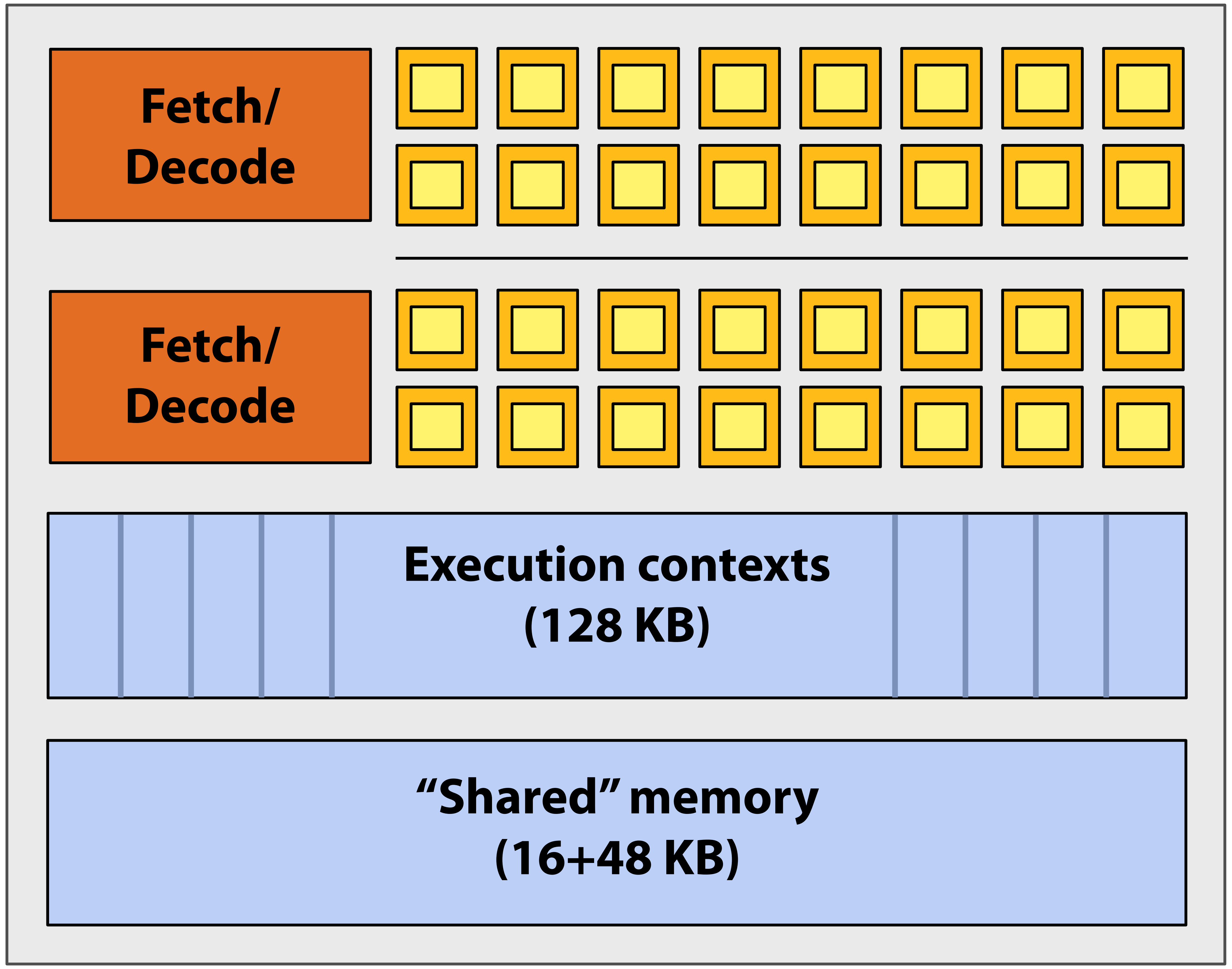


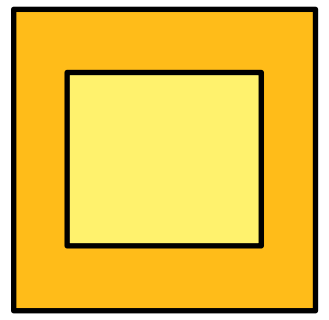

= SIMD function unit,
control shared across 16 units
(1 MUL-ADD per clock)

- The core contains 32 functional units
- Two groups are selected each clock (decode, fetch, and execute two instruction streams in parallel)

Source: Fermi Compute Architecture Whitepaper
CUDA Programming Guide 3.1, Appendix G

NVIDIA GeForce GTX 480 "SM"

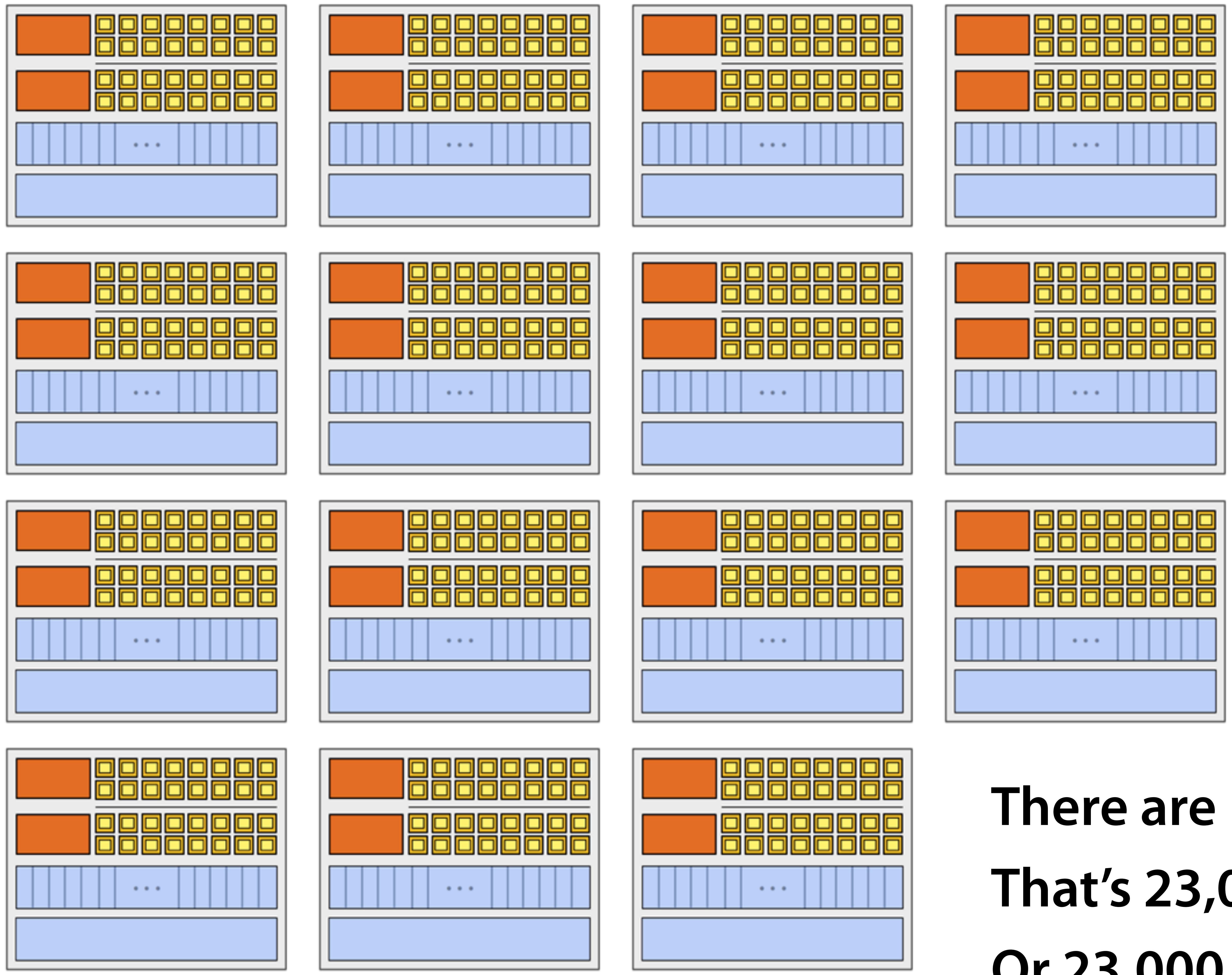




- The **SM** contains 32 **CUDA cores**
- Two **warps** are selected each clock (decode, fetch, and execute two **warps** in parallel)
- Up to 48 warps are interleaved, totaling 1536 **CUDA threads**

Source: Fermi Compute Architecture Whitepaper
CUDA Programming Guide 3.1, Appendix G

NVIDIA GeForce GTX 480



There are 15 of these things on the GTX 480:

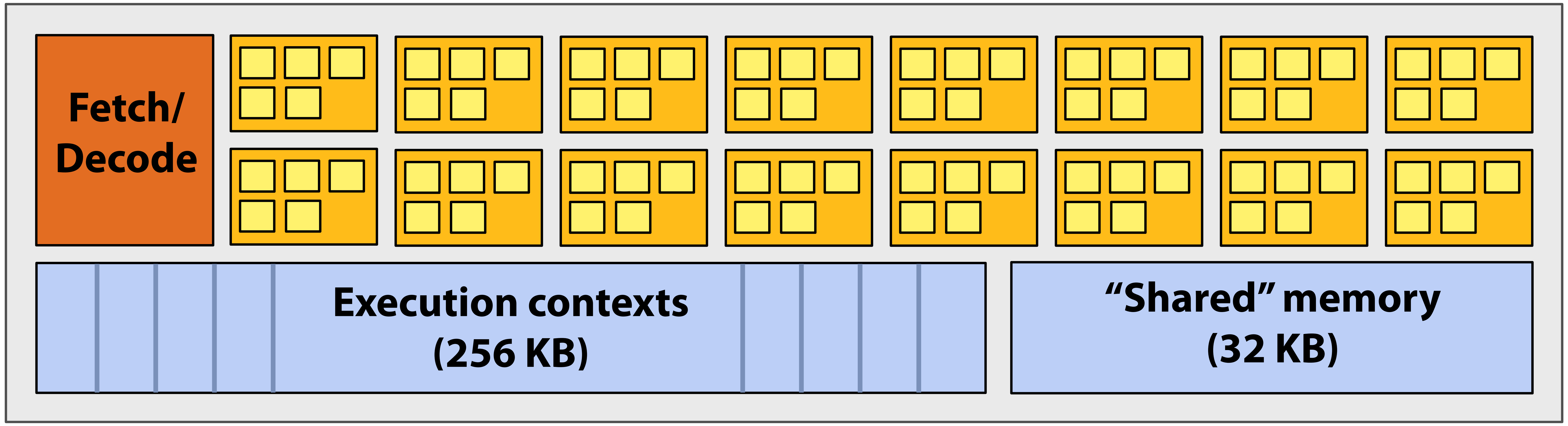
That's 23,000 fragments!

Or 23,000 CUDA threads!

ATI Radeon HD 5870 (Cypress)

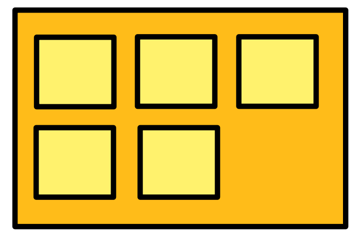
- AMD-speak:
  - 1600 stream processors
- Generic speak:
  - 20 cores
  - 16 “beefy” SIMD functional units per core
  - 5 multiply-adds per functional unit (VLIW processing)

ATI Radeon HD 5870 "core"



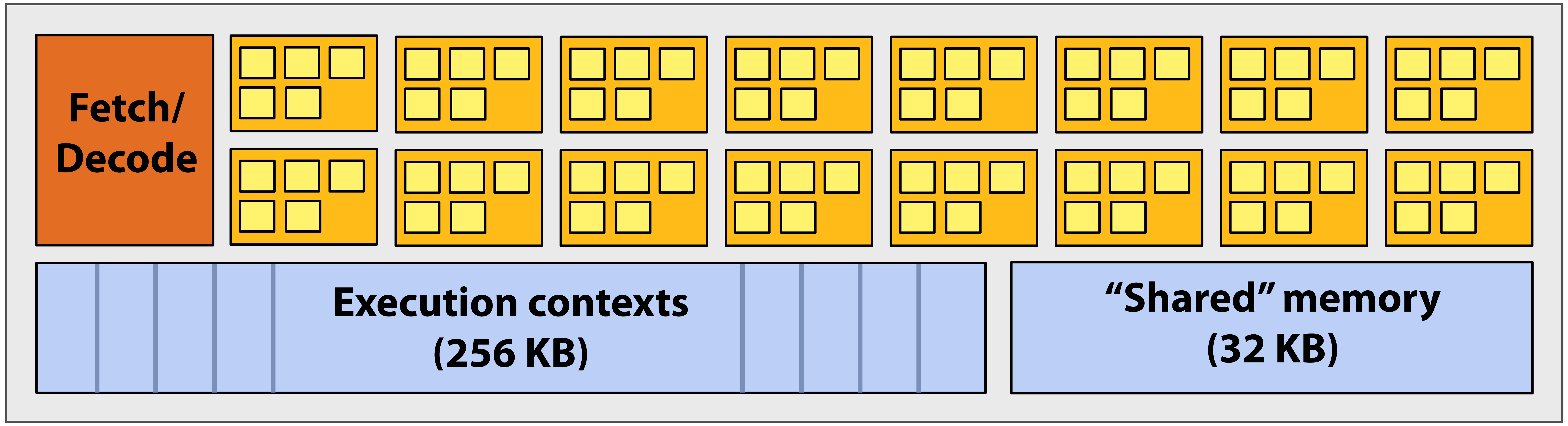
Groups of 64 [fragments/vertices/etc.] share instruction stream

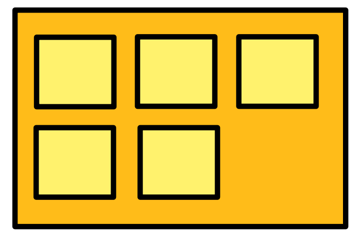
Four clocks to execute an instruction for all fragments in a group

 = SIMD function unit, control shared across 16 units (Up to 5 MUL-ADDs per clock)

Source: ATI Radeon HD5000 Series: An Inside View (HPG 2010)

ATI Radeon HD 5870 “SIMD-engine”





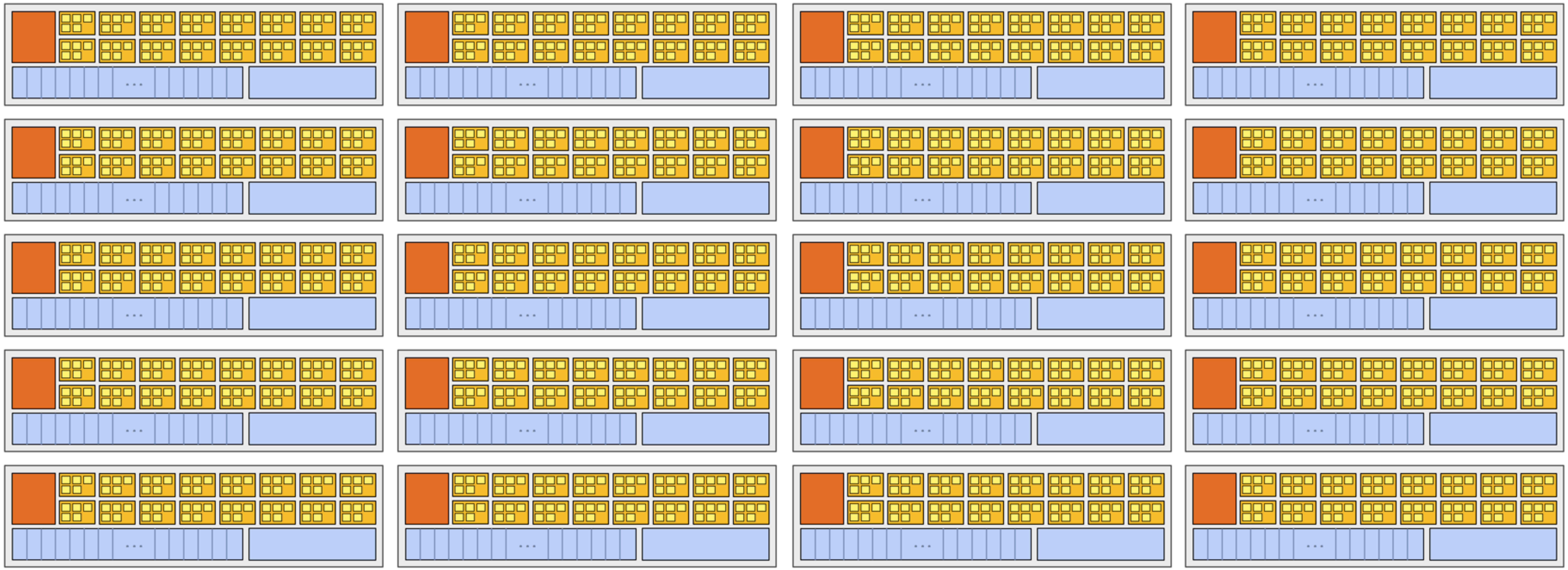
= **stream processor**, control shared across 16 units (Up to 5 MUL-ADDs per clock)

Groups of 64 [fragments/vertices/OpenCL work items] are in a **“wavefront”**.

Four clocks to execute an instruction for an entire **wavefront**

Source: ATI Radeon HD5000 Series: An Inside View (HPG 2010)

ATI Radeon HD 5870

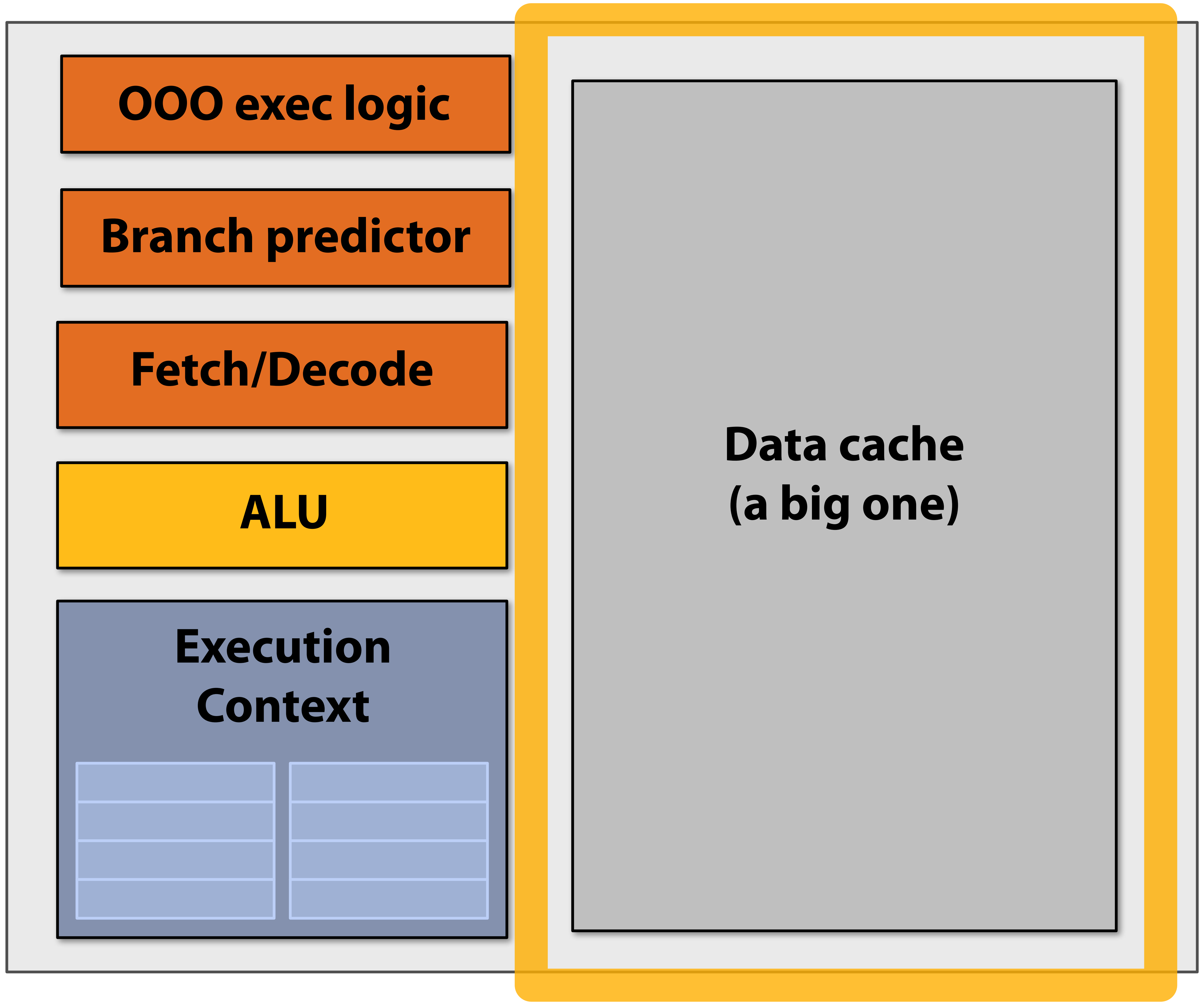


There are 20 of these "cores" on the 5870: that's about 31,000 fragments!

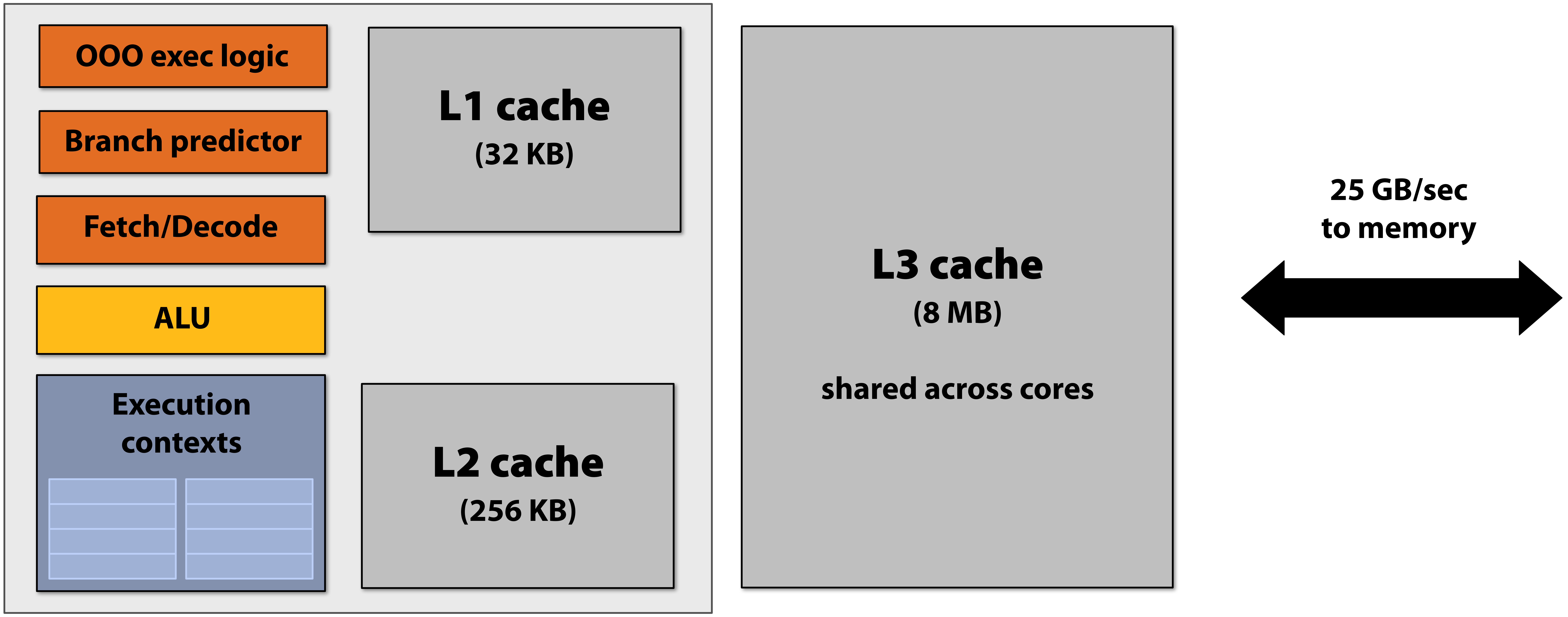
The talk thus far: processing data

Part 3: moving data to processors

Recall: “CPU-style” core

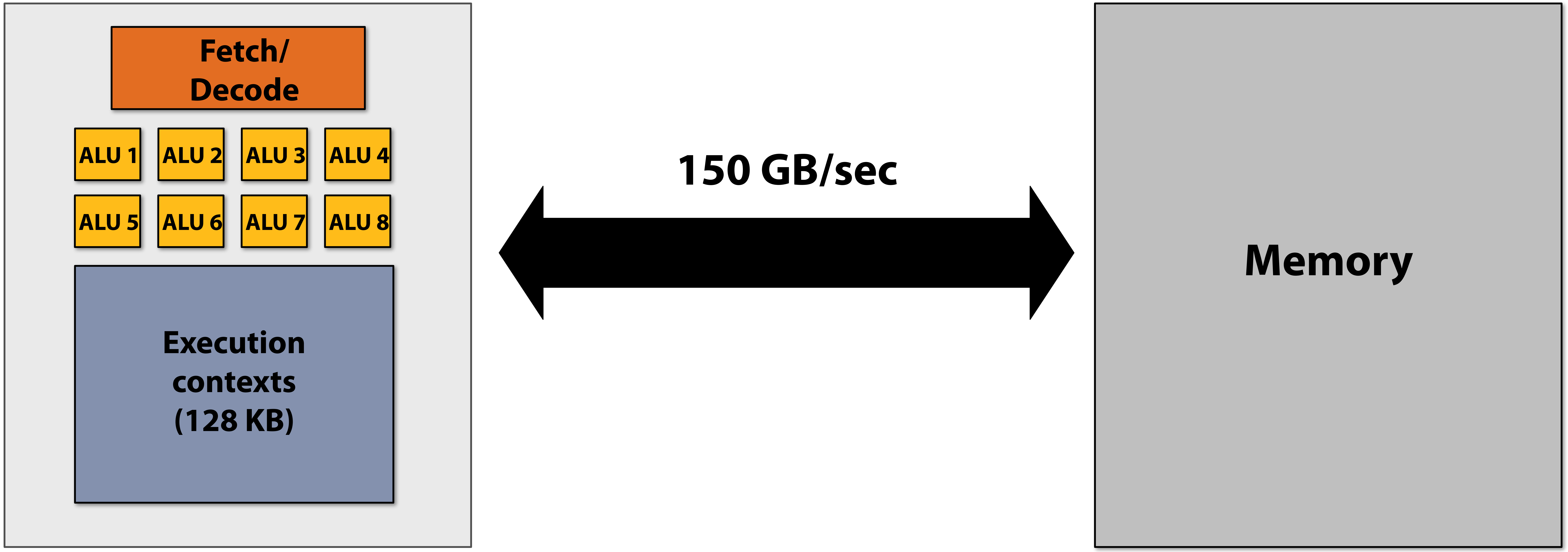


"CPU-style" memory hierarchy



CPU cores run efficiently when data is resident in cache
(caches reduce latency, provide high bandwidth)

Throughput core (GPU-style)



More ALUs, no large traditional cache hierarchy:
Need high-bandwidth connection to memory

Bandwidth is a critical resource

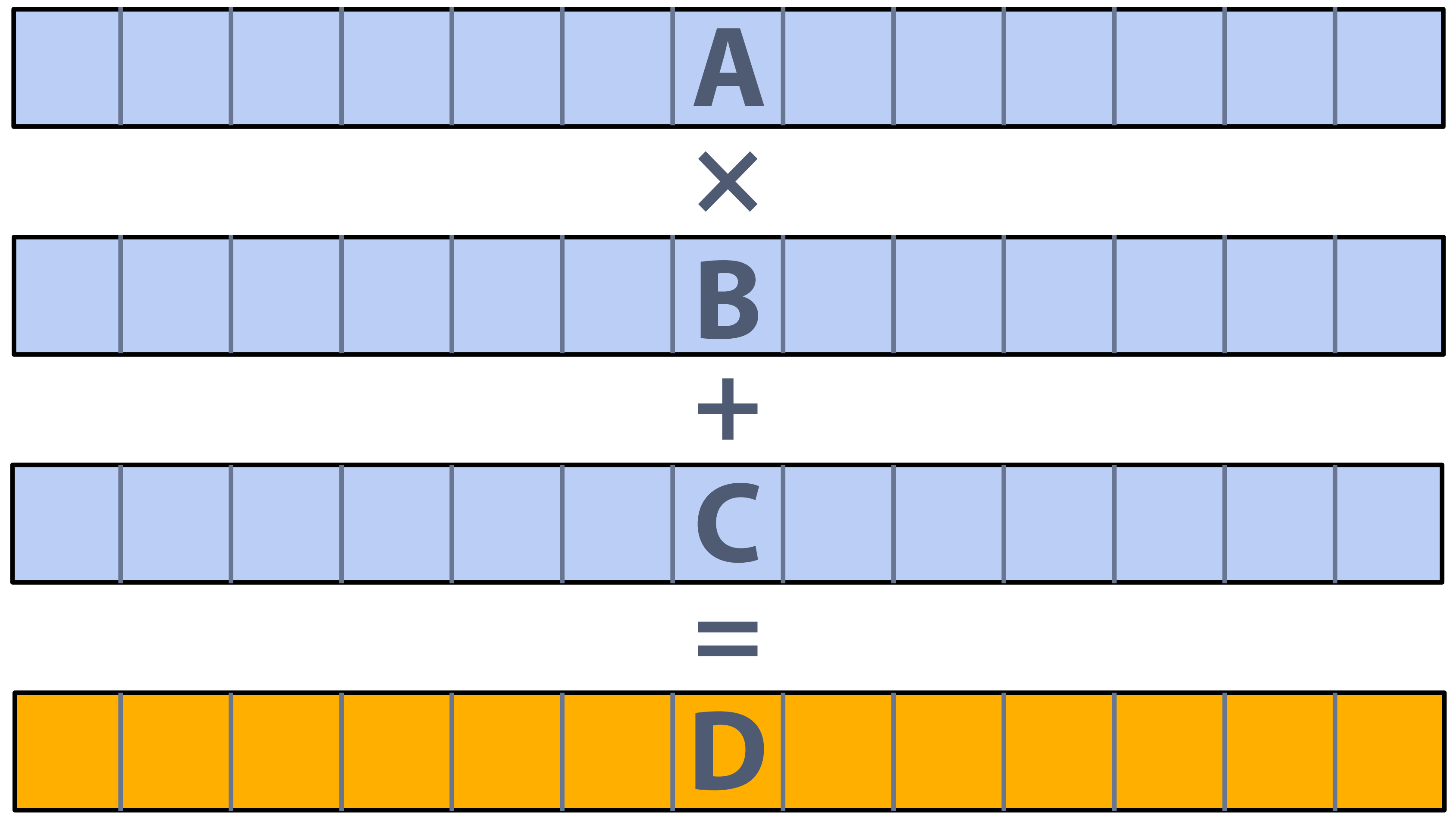
- A high-end GPU (e.g. Radeon HD 5870) has...
  - Over **twenty times** (2.7 TFLOPS) the compute performance of quad-core CPU
  - No large cache hierarchy to absorb memory requests

- GPU memory system is designed for throughput
  - Wide bus (150 GB/sec)
  - Repack/reorder/interleave memory requests to maximize use of memory bus
  - Still, this is only **six times** the bandwidth available to CPU

Bandwidth thought experiment

Task: element-wise multiply two long vectors A and B

1. Load input A[i]
2. Load input B[i]
3. Load input C[i]
4. Compute A[i] × B[i] + C[i]
5. Store result into D[i]



Four memory operations (16 bytes) for every MUL-ADD

Radeon HD 5870 can do 1600 MUL-ADDS per clock

Need ~20 TB/sec of bandwidth to keep functional units busy

Less than 1% efficiency… but 6x faster than CPU!

Bandwidth limited!

If processors request data at too high a rate,
the memory system cannot keep up.

No amount of latency hiding helps this.

Overcoming bandwidth limits are a common challenge
for GPU-compute application developers.

Reducing bandwidth requirements

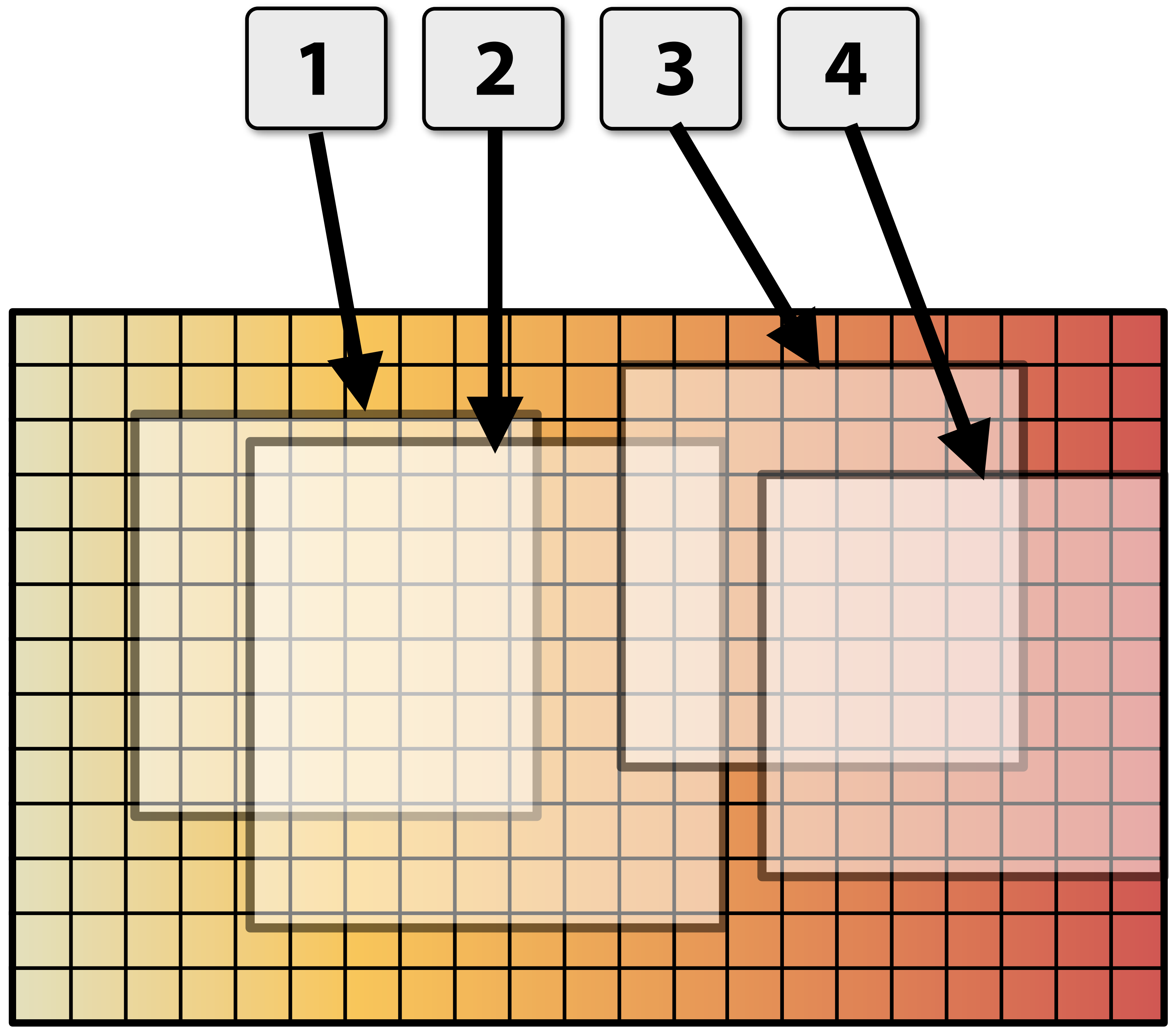
- Request data less often (instead, do more math)
  - "arithmetic intensity"

- Fetch data from memory less often (share/reuse data across fragments
  - on-chip communication or storage

Reducing bandwidth requirements

- Two examples of on-chip storage
  - Texture caches
  - OpenCL “local memory” (CUDA shared memory)

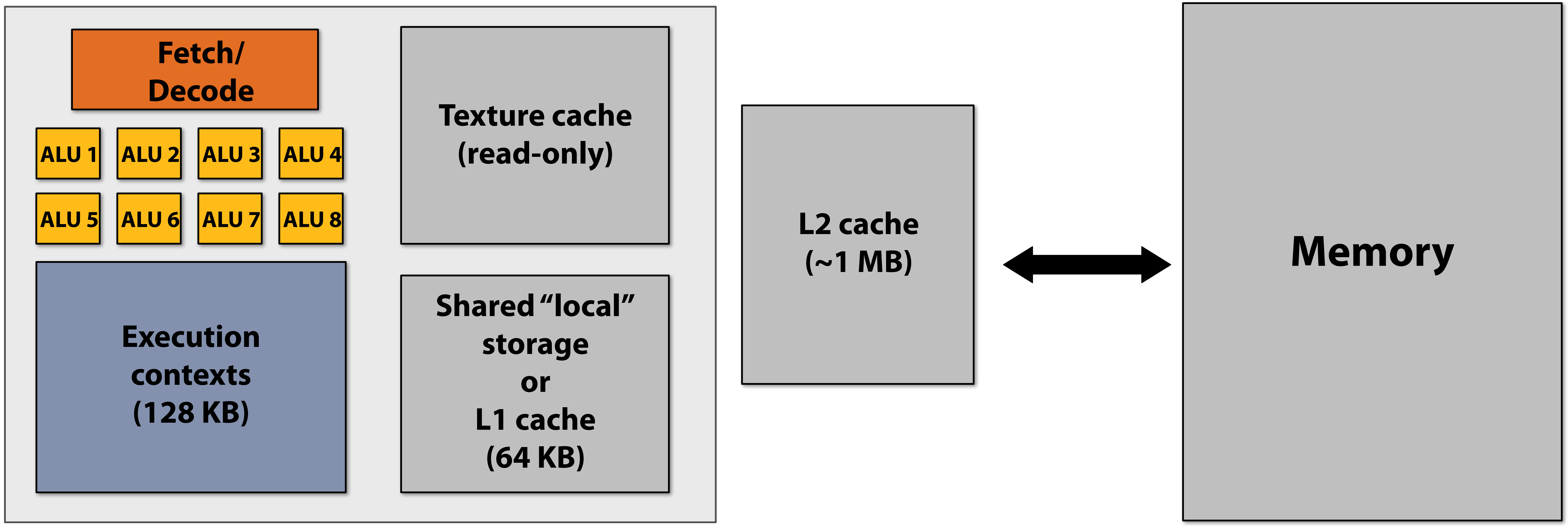
Texture data



Texture caches:

Capture reuse across fragments, not temporal reuse within a single shader program

Modern GPU memory hierarchy


Fetch/ Decode
ALU 1
ALU 2
ALU 3
ALU 4
ALU 5
ALU 6
ALU 7
ALU 8
Execution contexts (128 KB)
Texture cache (read-only)
Shared "local" storage or L1 cache (64 KB)
L2 cache (~1 MB)
Memory

On-chip storage takes load off memory system.
Many developers calling for more cache-like storage (particularly GPU-compute applications)

Summary

Think of a GPU as a heterogeneous multi-core processor optimized for maximum throughput.

(currently at the extreme end of the design space)

Think about how the various programming models map to these architectural characteristics.

(Aaron is going to talk about this next)

An efficient GPU workload ...

- Has thousands of independent pieces of work
  - Uses many ALUs on many cores
  - Supports massive interleaving for latency hiding

- Is amenable to instruction stream sharing
  - Maps to SIMD execution well

- Is compute-heavy: the ratio of math operations to memory access is high
  - Not limited by memory bandwidth

Thank you

Special thanks to the following contributors:

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