

CS 380 - GPU and GPGPU Programming

Lecture 7: GPU Architecture, Pt. 5

Markus Hadwiger, KAUST

Reading Assignment #4 (until Sep 28)



Read (required):

- Get an overview of NVIDIA Hopper (H100) Tensor Core GPU white paper:

<https://resources.nvidia.com/en-us-hopper-architecture/nvidia-h100-tensor-c>

- Get an overview of NVIDIA Blackwell (GB202) GPU white papers:

<https://images.nvidia.com/aem-dam/Solutions/geforce/blackwell/nvidia-rtx-blackwell-gpu-architecture.pdf>

<https://www.nvidia.com/content/dam/en-zz/Solutions/design-visualization/quadro-product-literature/NVIDIA-RTX-Blackwell-PRO-GPU-Architecture-v1.0.pdf>

Read (optional):

- Look at the “Tuning Guides” for different architectures in the CUDA SDK
- PTX Instruction Set Architecture (9.4): https://docs.nvidia.com/cuda/pdf/ptx_isa_9.4.pdf
Read Chapters 1 – 3; get an overview of Chapter 9;
browse through the other chapters to get a feeling for what PTX looks like
- CUDA SASS ISA (13.4), Chap. 6: https://docs.nvidia.com/cuda/pdf/CUDA_Binary_Uutilities.pdf

Concepts: Latency vs. Throughput



Latency

- What is the time *between start and finish* of an operation/computation?
- How long does it take between starting to execute an instruction until the execution is actually finished / its results are available?
- Examples: 1 FP32 MUL instruction; 1 vertex computation, ...

Throughput

- How many computations (operations/instructions) *finish per time unit*?
- How many instructions of a certain type (e.g., FP32 MUL) finish per time unit (per clock cycle, per second)?

GPUs: ***High-throughput execution*** (at the expense of latency)
(but: *hide* latencies to avoid throughput going down)

Concepts: Types of Parallelism



Instruction level parallelism (ILP)

- In single instruction stream: Can consecutive instructions/operations be executed in parallel? (Because they don't have a dependency)
- Exploit ILP: Execute independent instructions (1) via pipelined execution (instr. pipe), or even (2) in multiple parallel instruction pipelines (superscalar processors)
- On GPUs: also important, but much less than TLP (compare, e.g., Kepler with current GPUs)

Thread level parallelism (TLP)

- Exploit that by definition operations in different threads are independent (if no explicit communication/synchronization is used, which should be minimized)
- Exploit TLP: Execute operations/instructions from multiple threads in parallel (which also needs multiple parallel instruction pipelines)
- **On GPUs: main type of parallelism**

more types:

- Bit-level parallelism (processor word size: 64 bits instead of 32, etc.)
- Data parallelism (SIMD/vector instructions), task parallelism, ...

Concepts: Latency Hiding



**Not about latency of single operation or group of operations:
It's about avoiding that the *throughput* goes below peak**

Hide latency that *does* occur for one instruction (group) by
executing a different instruction (group) as soon as current one stalls:

→ *Total throughput does not go down*

In GPUs, hide latencies via:

- **TLP: pull independent, not-stalling instruction from other thread group**
- ILP: pull independent instruction from down the inst. stream in same thread group
- Depending on GPU: TLP often sufficient, but sometimes also need ILP
- However: If in one cycle TLP doesn't work, ILP can jump in or vice versa

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

Kayvon Fatahalian
Stanford University

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...

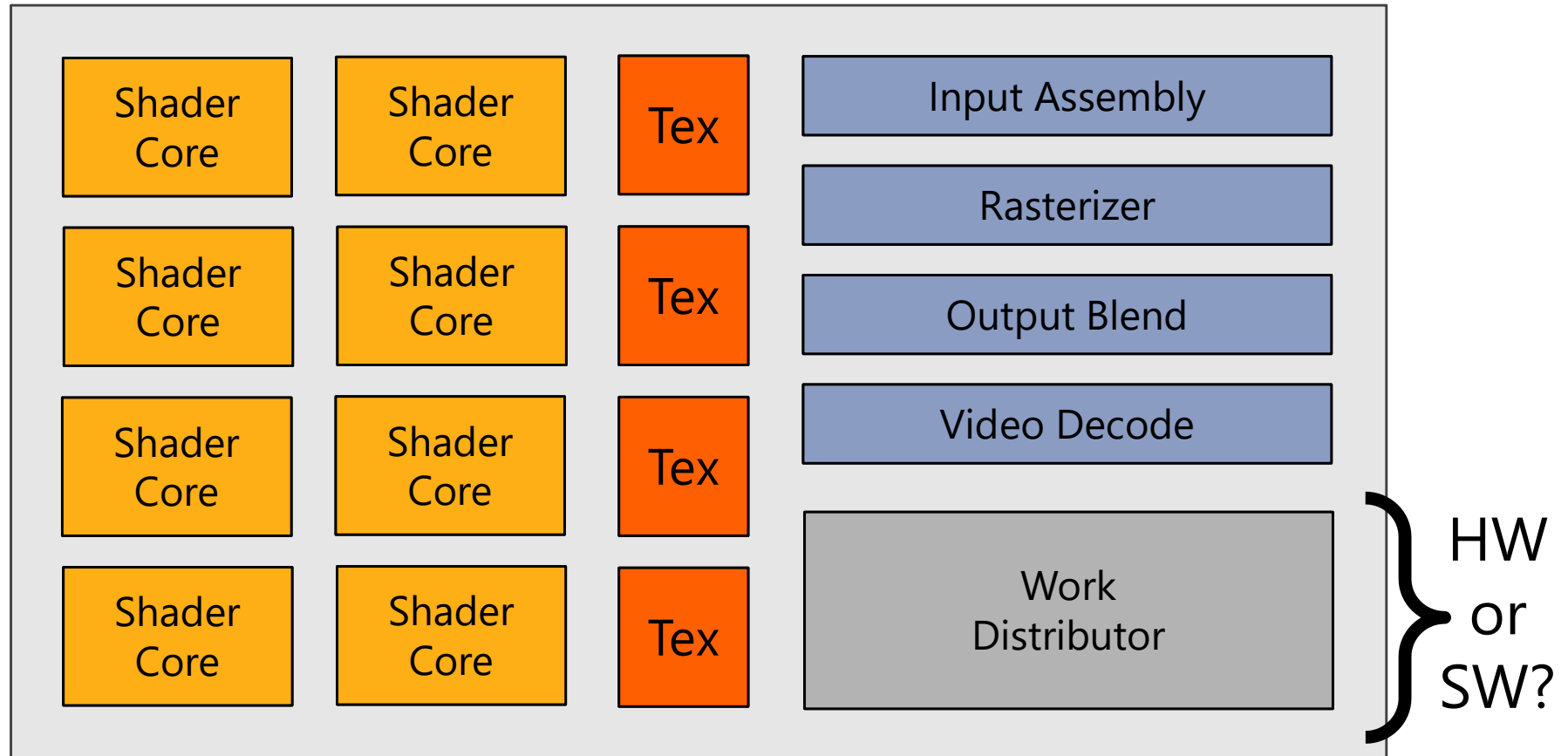


Summary: three key ideas for high-throughput execution

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

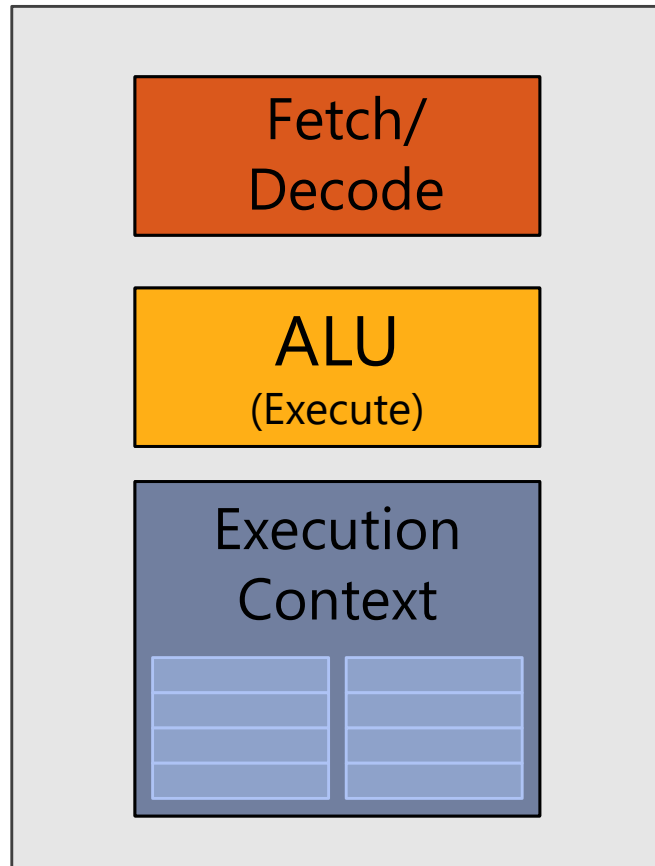
**GPUs are here!
(usually)**

What's in a GPU?



Heterogeneous chip multi-processor (highly tuned for graphics)

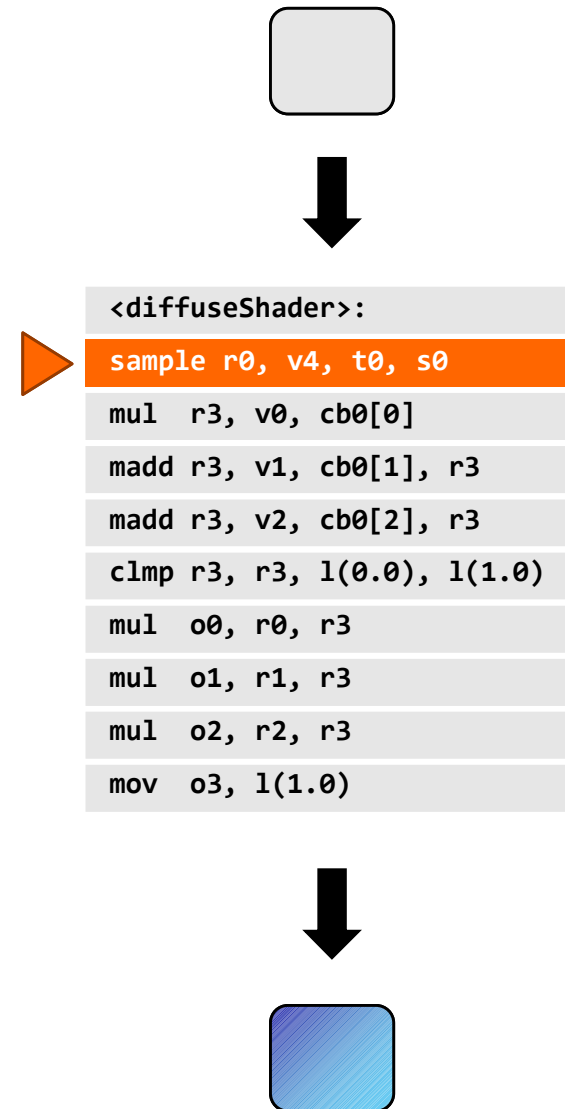
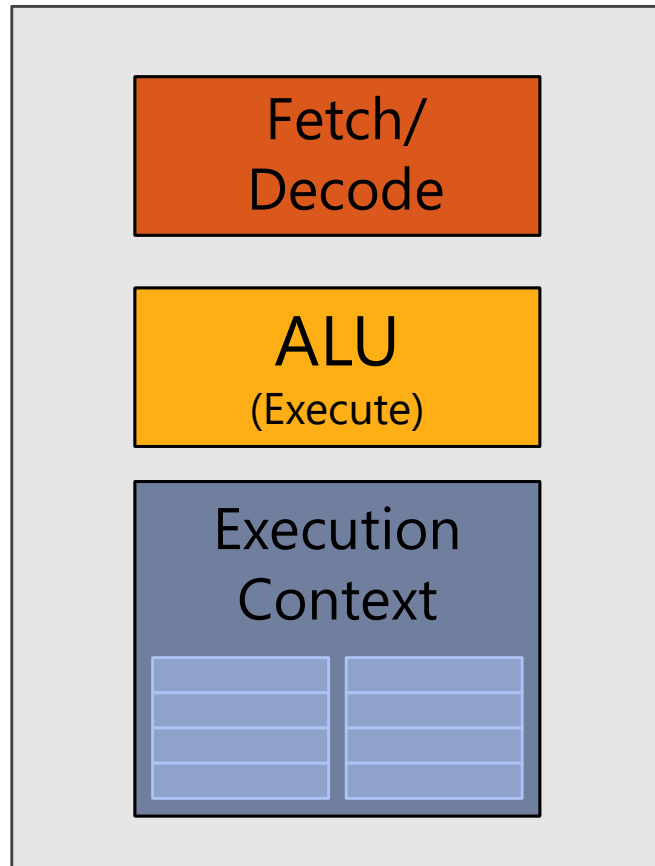
Execute shader



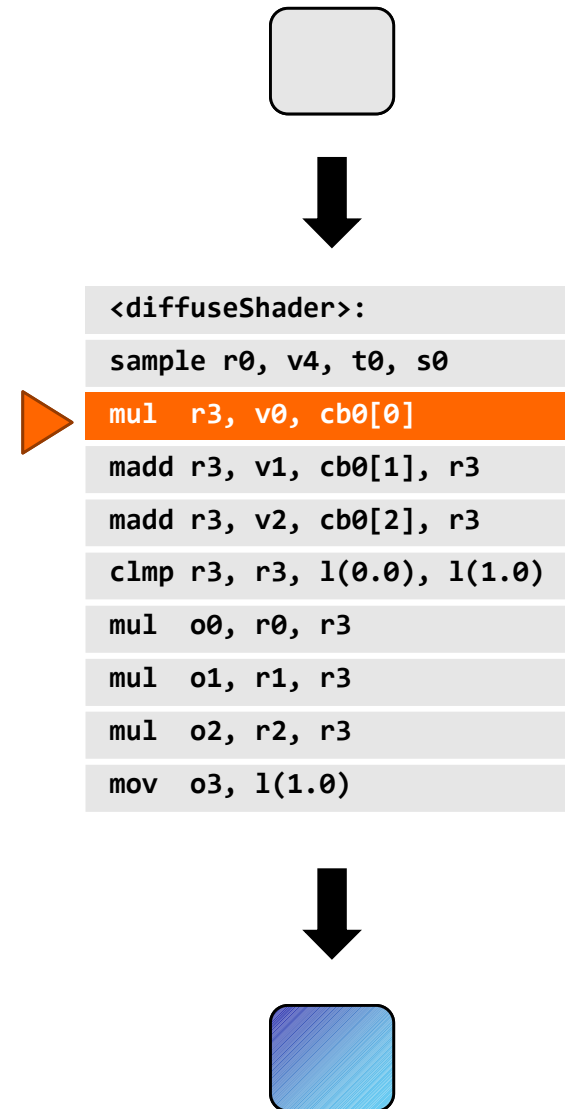
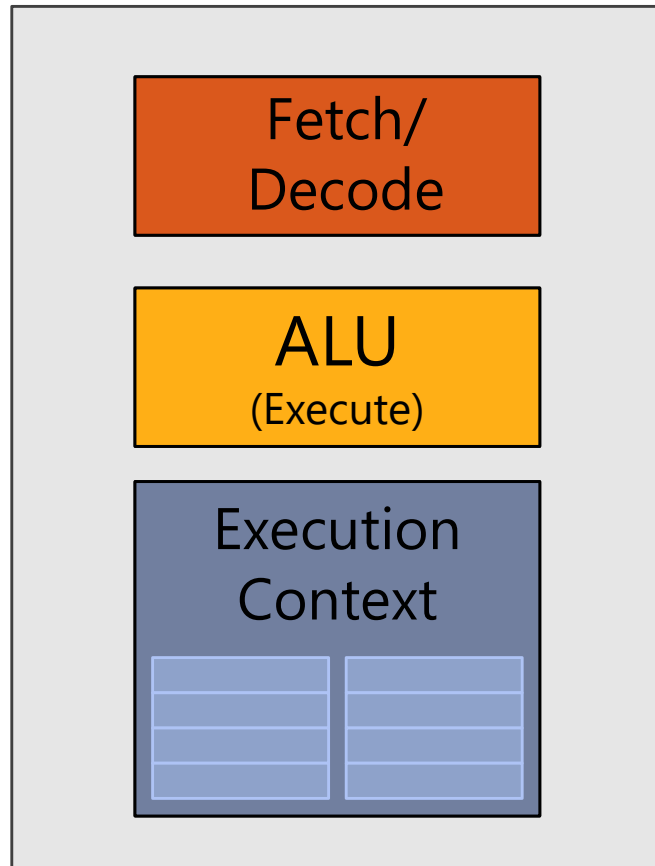
```
<diffuseShader>:  
sample r0, v4, t0, s0  
mul  r3, v0, cb0[0]  
madd r3, v1, cb0[1], r3  
madd r3, v2, cb0[2], r3  
clmp r3, r3, 1(0.0), 1(1.0)  
mul  o0, r0, r3  
mul  o1, r1, r3  
mul  o2, r2, r3  
mov  o3, 1(1.0)
```



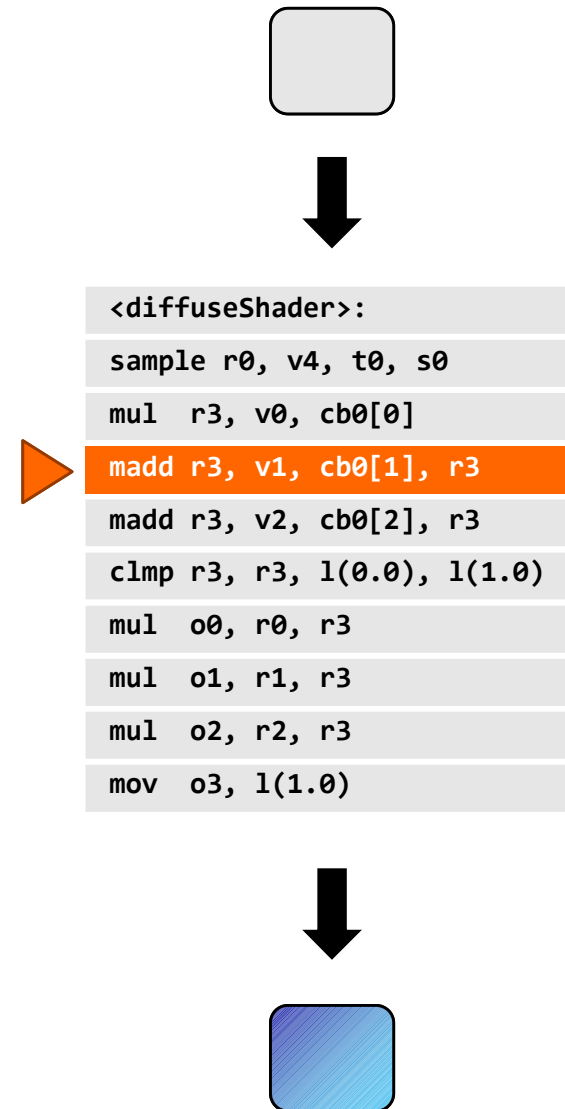
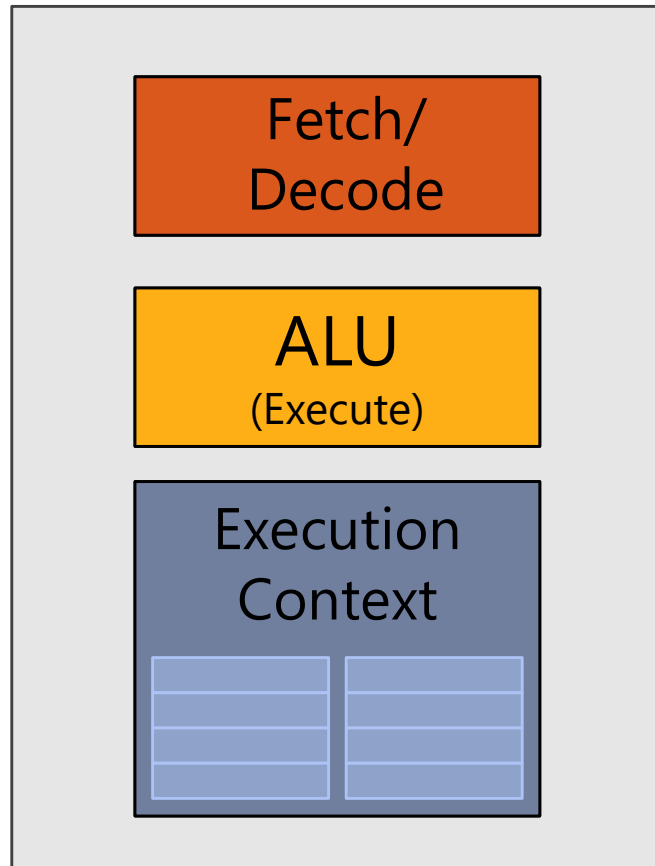
Execute shader



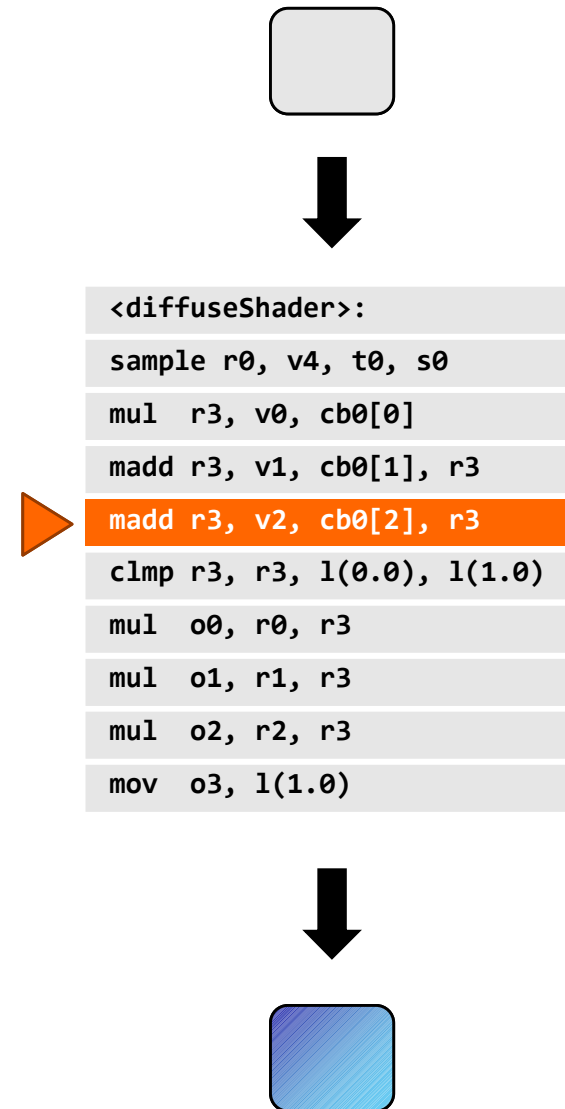
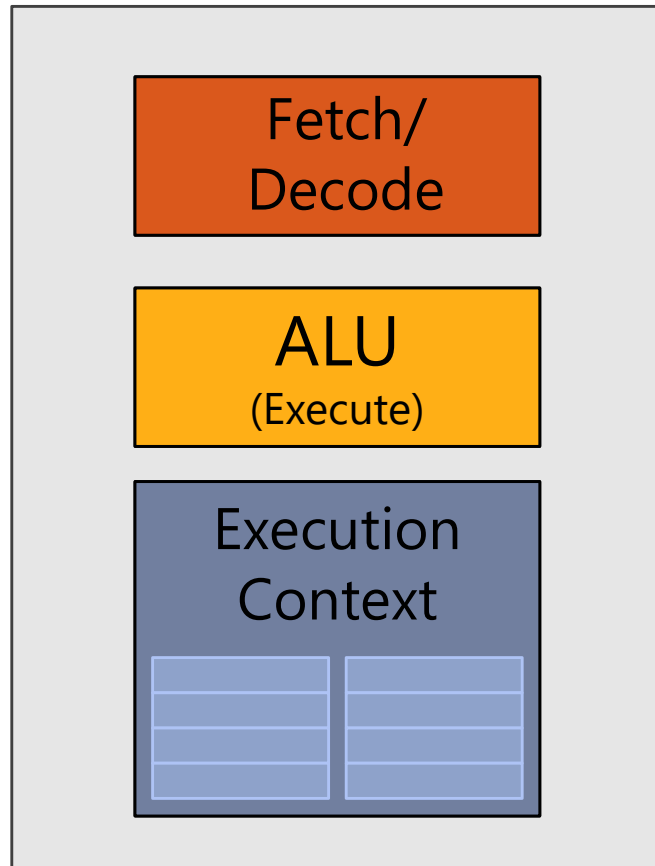
Execute shader



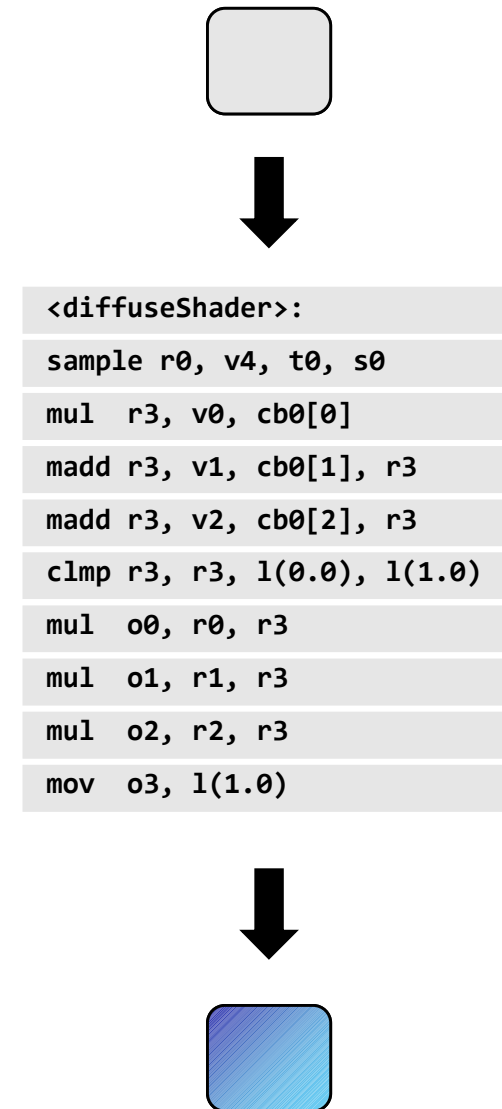
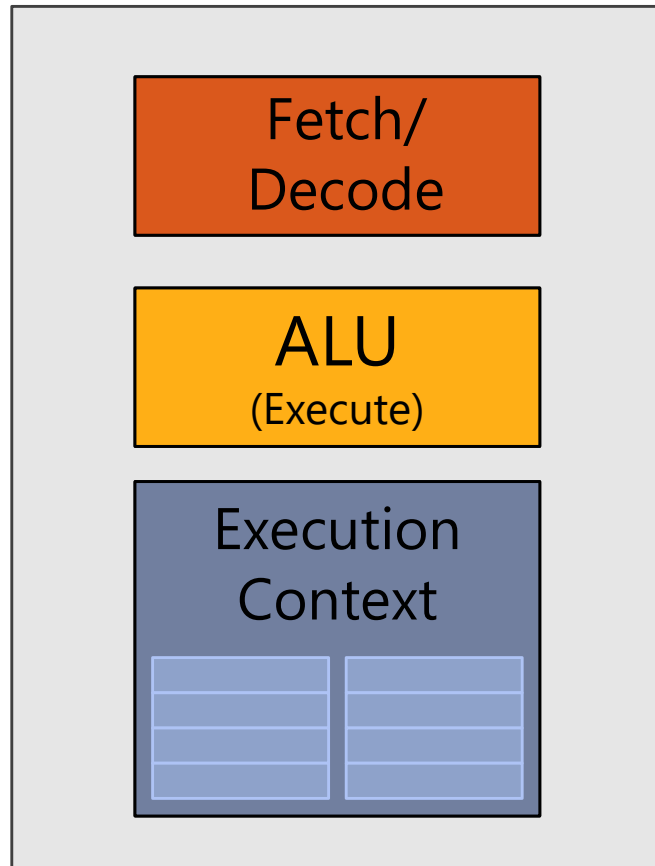
Execute shader



Execute shader



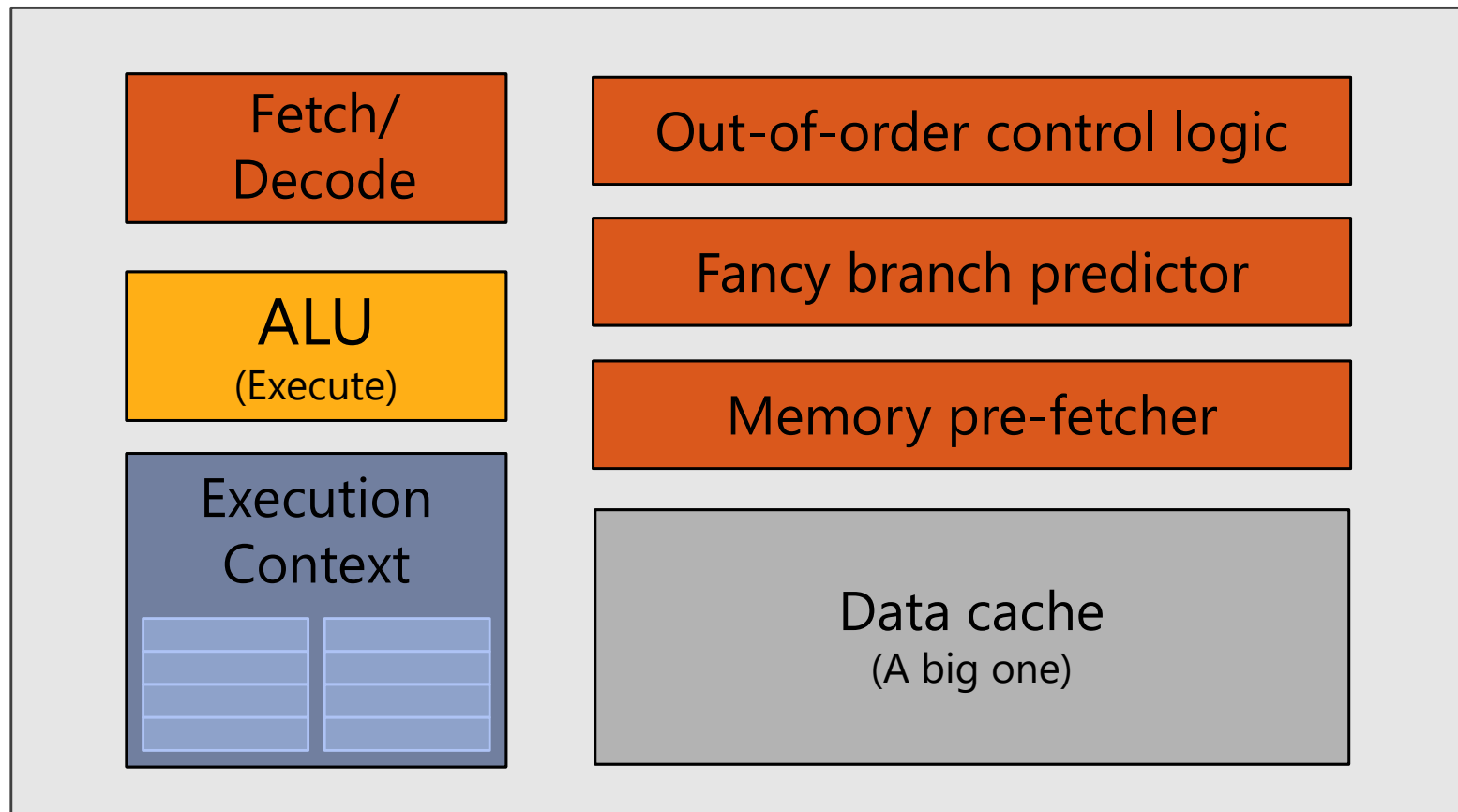
Execute shader



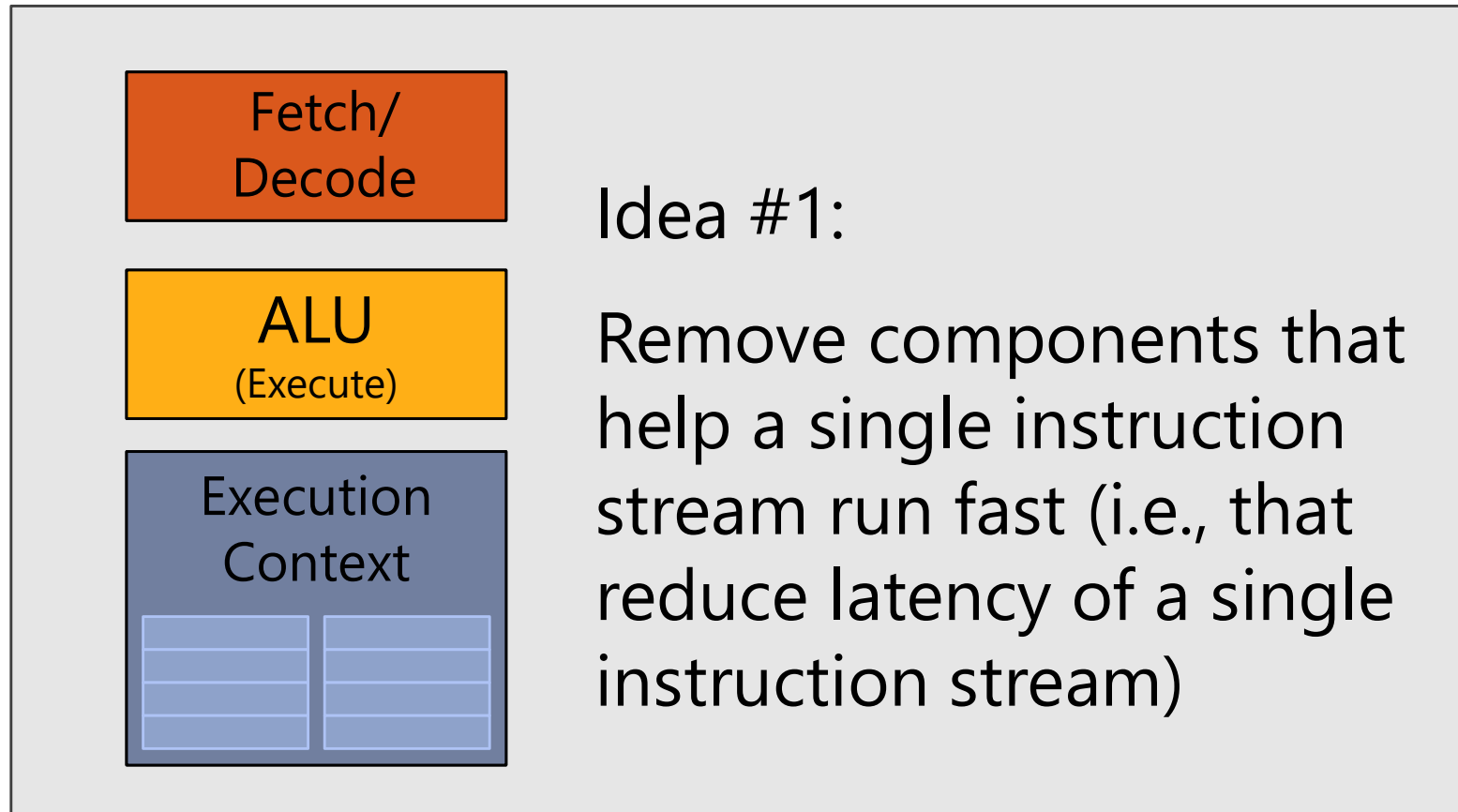
GPU Architecture

Big Idea #1

CPU-“style” cores

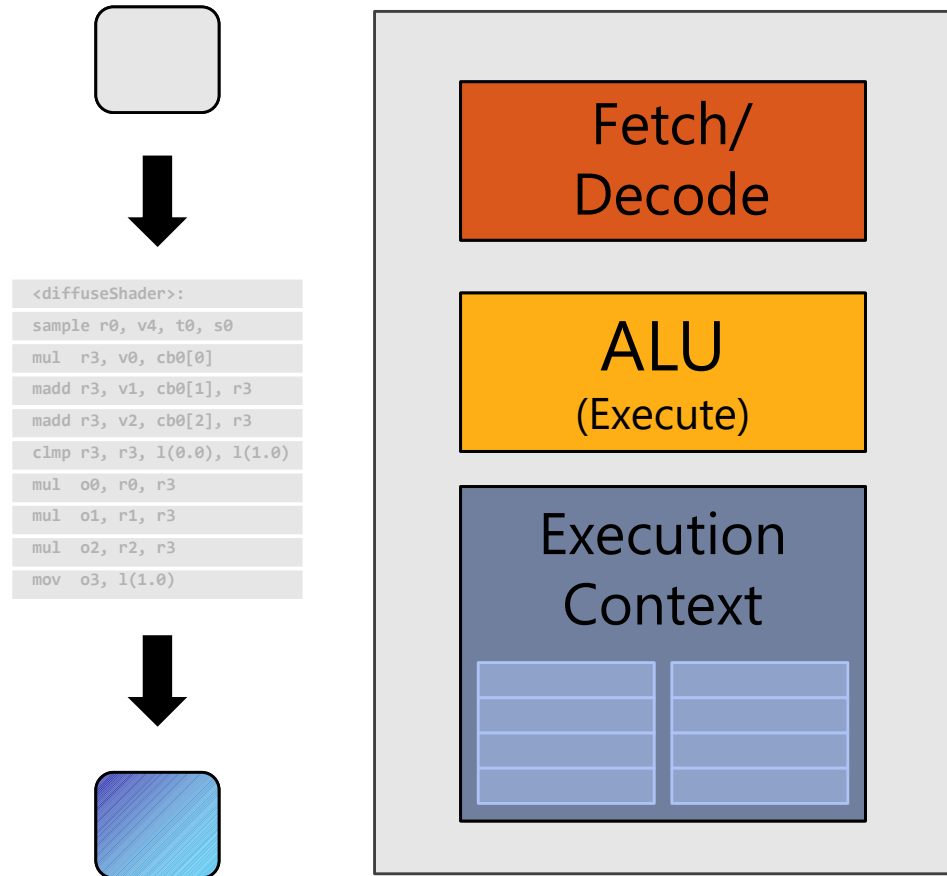


Idea #1: Slim down

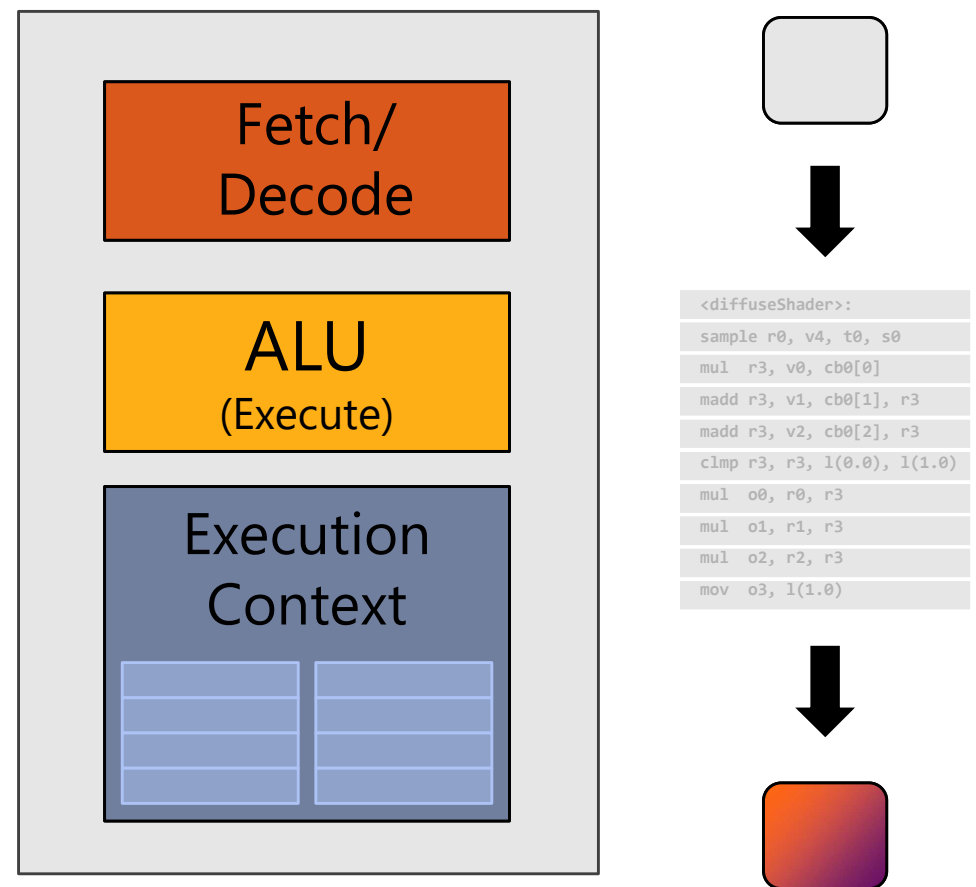


Two cores (two fragments in parallel)

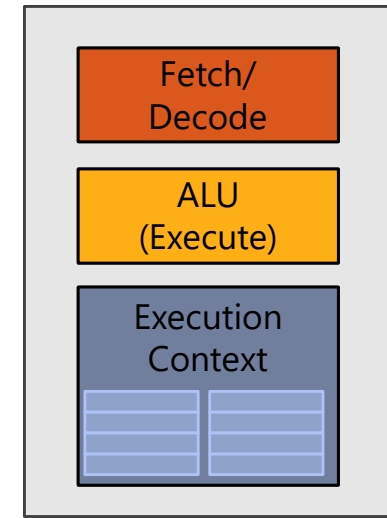
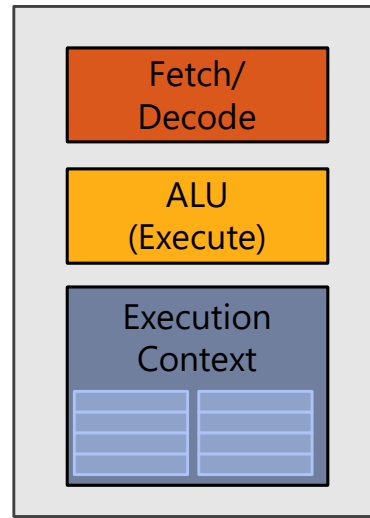
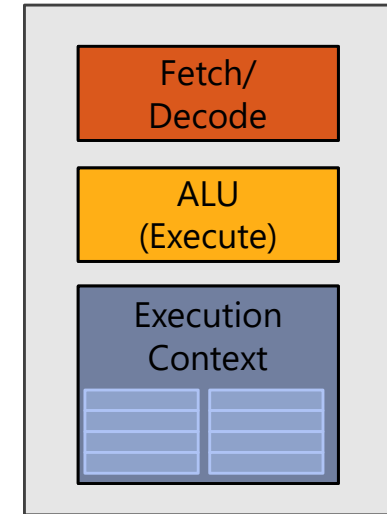
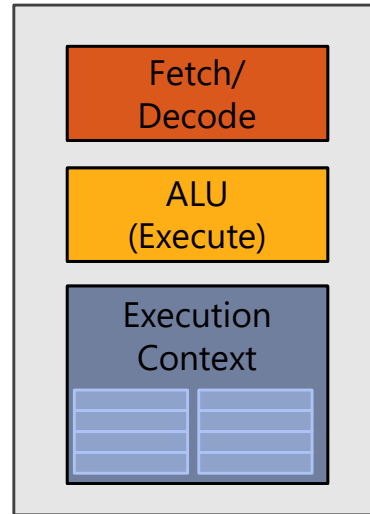
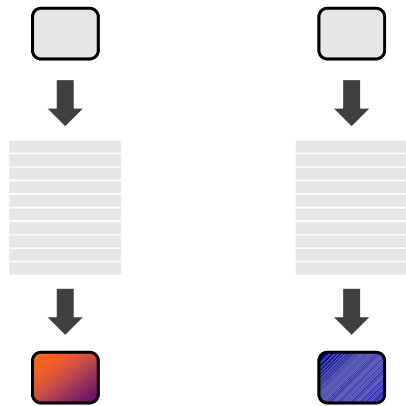
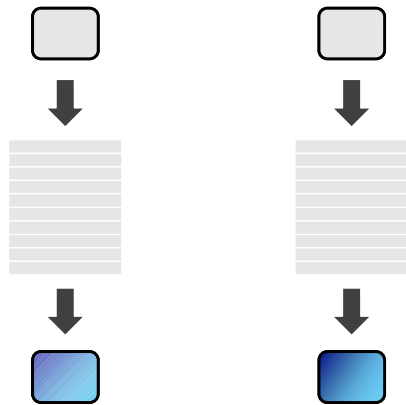
fragment 1



fragment 2



Four cores (four fragments in parallel)

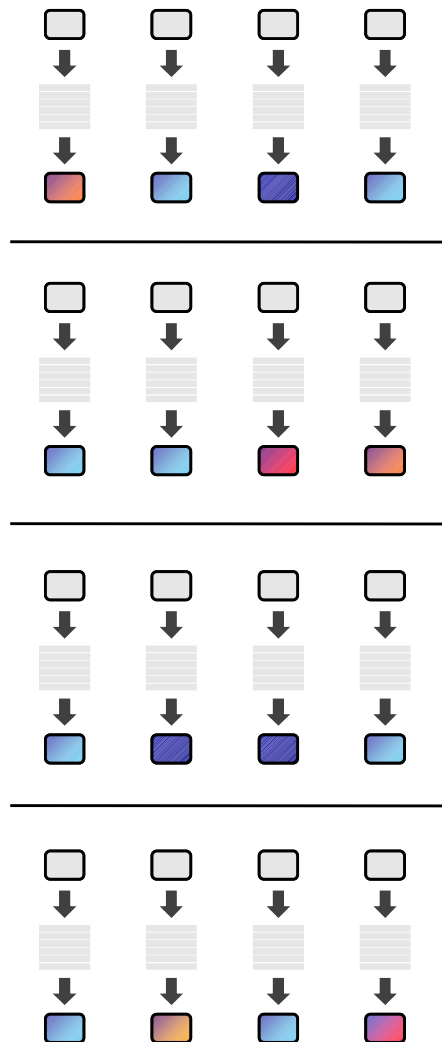


Sixteen cores (sixteen fragments in parallel)



16 cores = 16 simultaneous instruction streams

Instruction stream sharing



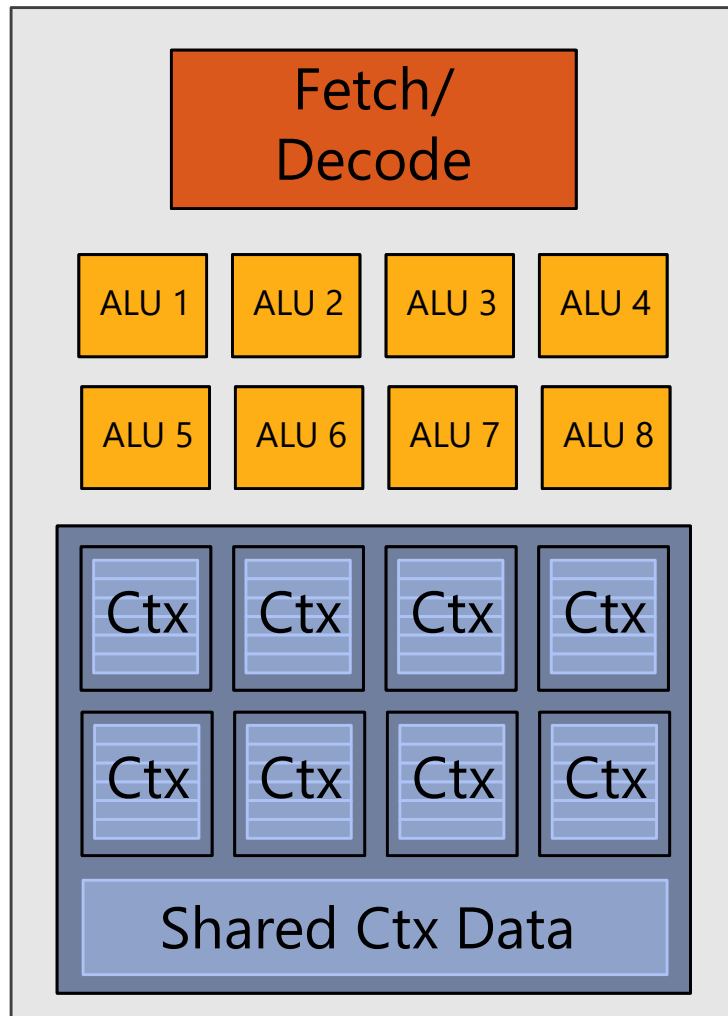
But... many fragments should be able to share an instruction stream! → **big idea #2 !**

```
<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)
```

GPU Architecture

Big Idea #2

Idea #2: Add ALUs



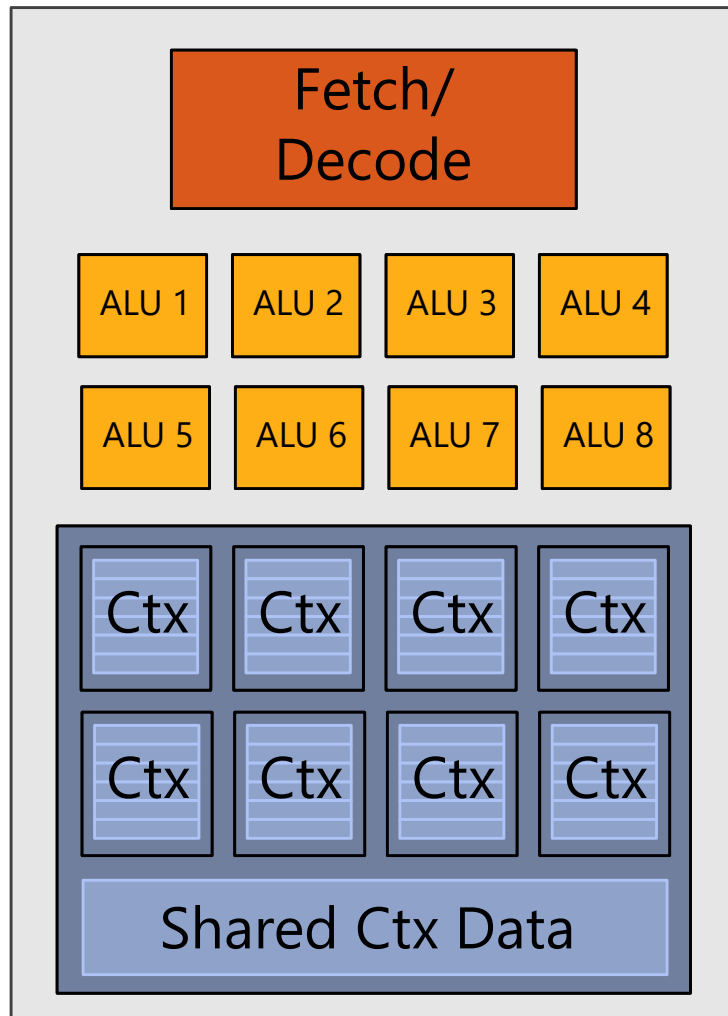
Idea #2:

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

SIMD processing

(or **SIMT**, SPMD)

How does shader execution behave?

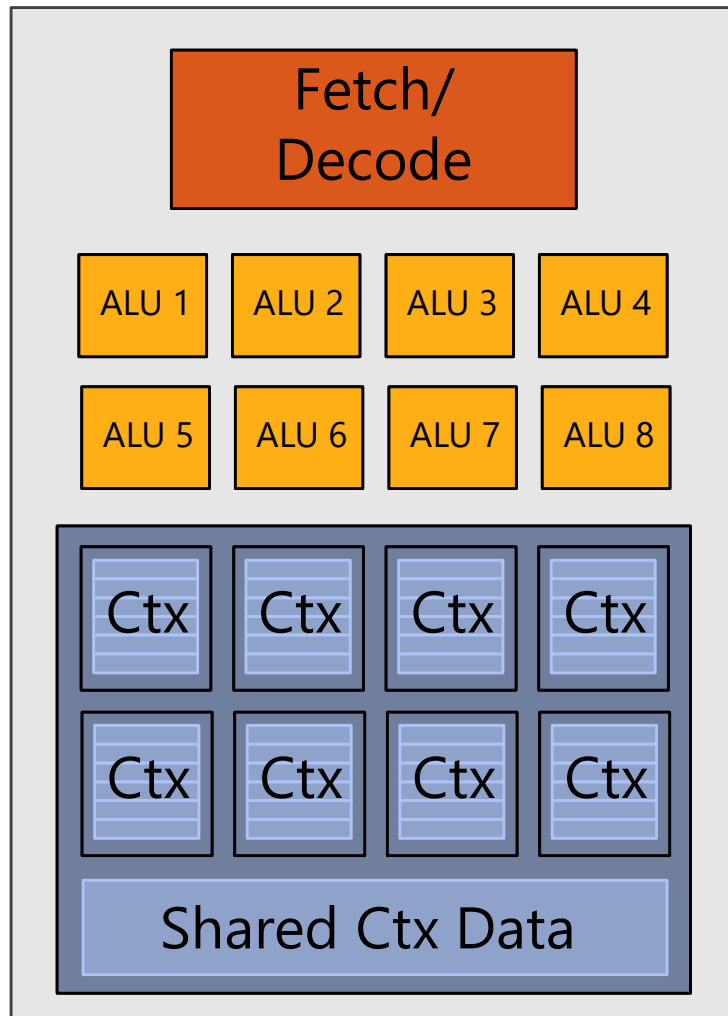


```
<diffuseShader>:  
sample r0, v4, t0, s0  
mul  r3, v0, cb0[0]  
madd r3, v1, cb0[1], r3  
madd r3, v2, cb0[2], r3  
clmp r3, r3, l(0.0), l(1.0)  
mul  o0, r0, r3  
mul  o1, r1, r3  
mul  o2, r2, r3  
mov  o3, l(1.0)
```

Original compiled shader:

Processes one fragment
using scalar ops on scalar registers

How does shader execution behave?

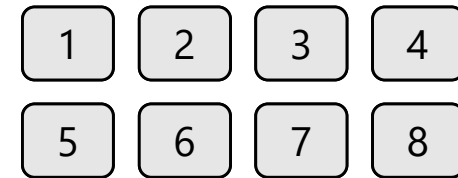
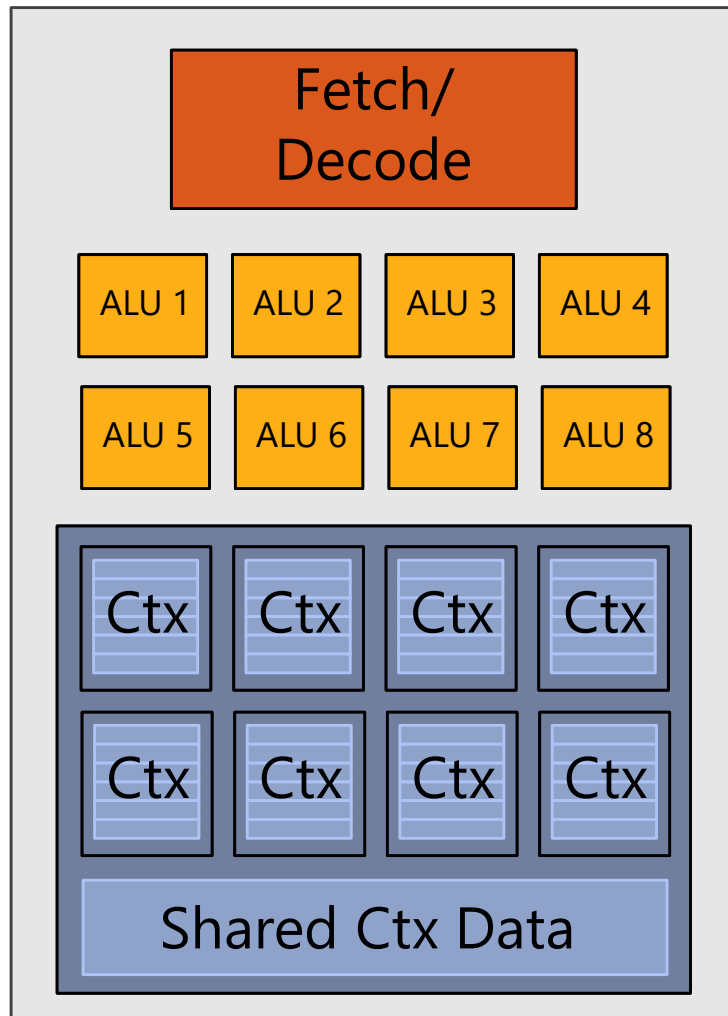


```
<VEC8_diffuseShader>:  
VEC8_sample vec_r0, vec_v4, t0, vec_s0  
VEC8_mul   vec_r3, vec_v0, cb0[0]  
VEC8_madd  vec_r3, vec_v1, cb0[1], vec_r3  
VEC8_madd  vec_r3, vec_v2, cb0[2], vec_r3  
VEC8_clmp  vec_r3, vec_r3, 1(0.0), 1(1.0)  
VEC8_mul   vec_o0, vec_r0, vec_r3  
VEC8_mul   vec_o1, vec_r1, vec_r3  
VEC8_mul   vec_o2, vec_r2, vec_r3  
VEC8_mov   vec_o3, 1(1.0)
```

Actually executed shader:

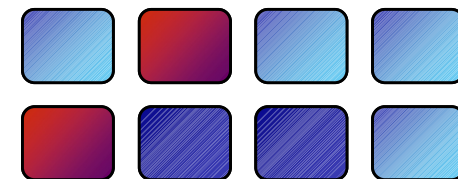
Processes 8 fragments
using "vector ops" on "vector registers"
**(Caveat: This does NOT mean there are actual
vector instructions/cores/regs! See later slide.)**

How does shader execution behave?

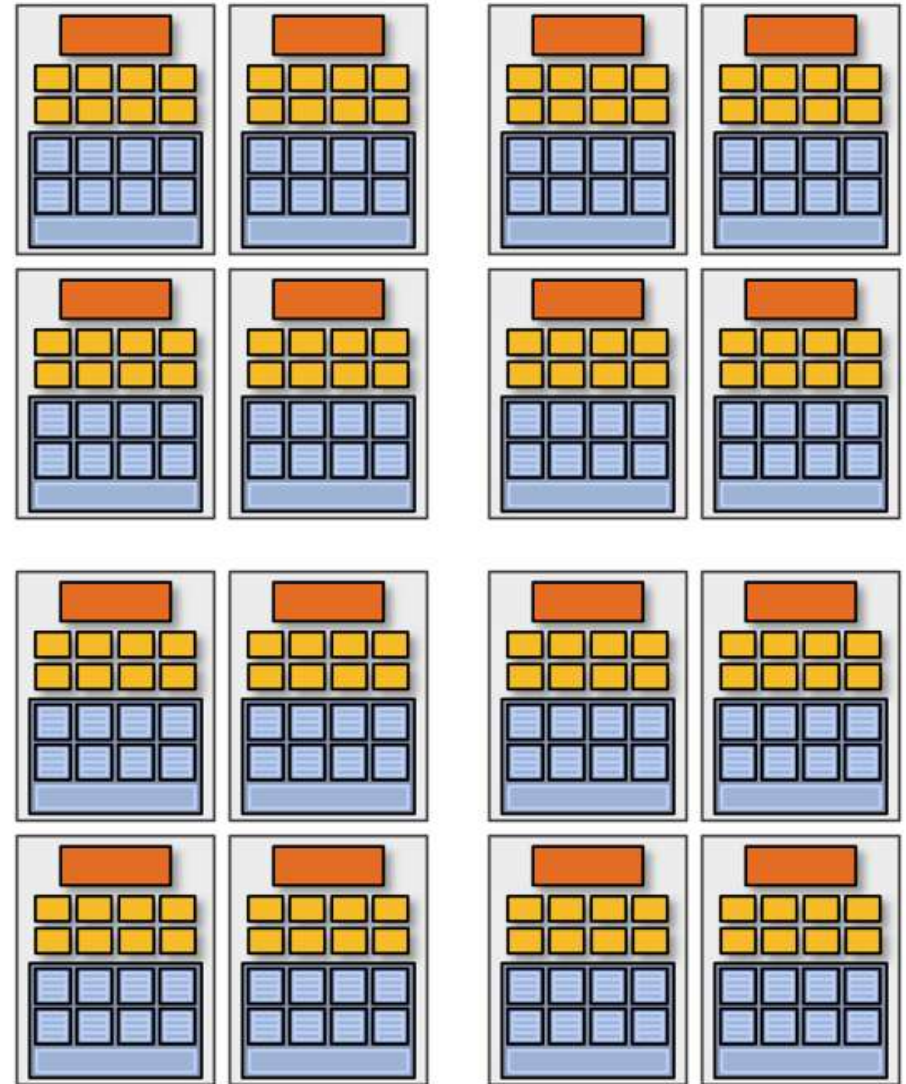
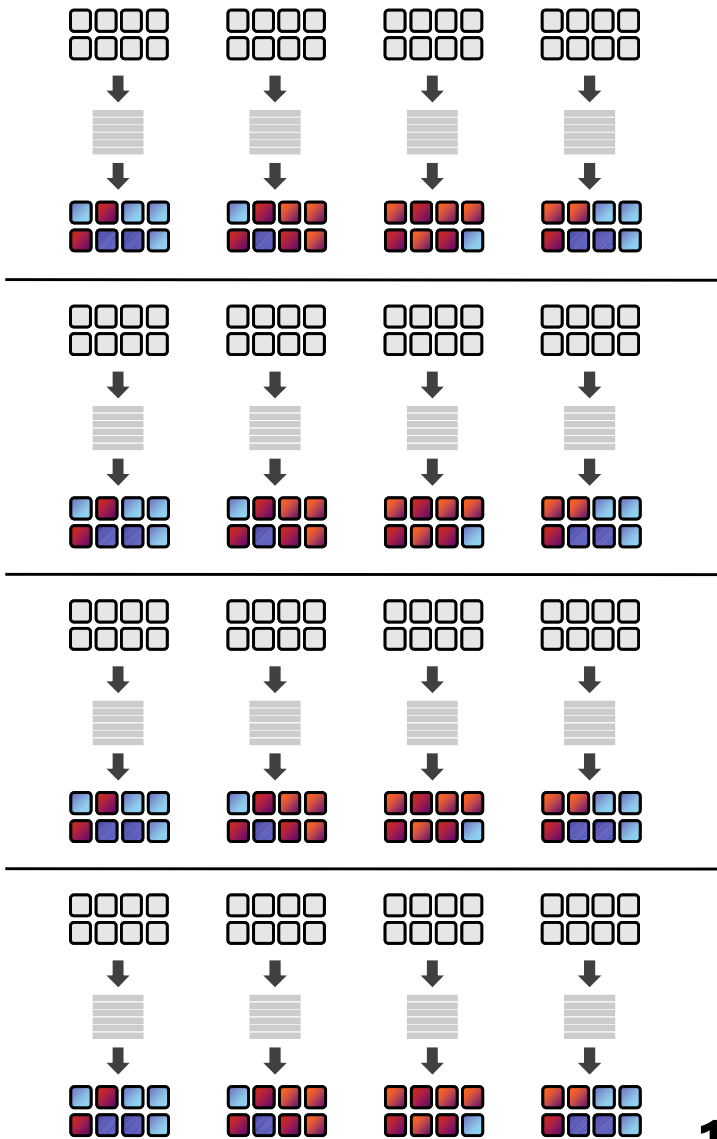


<VEC8_diffuseShader>:

```
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)
```

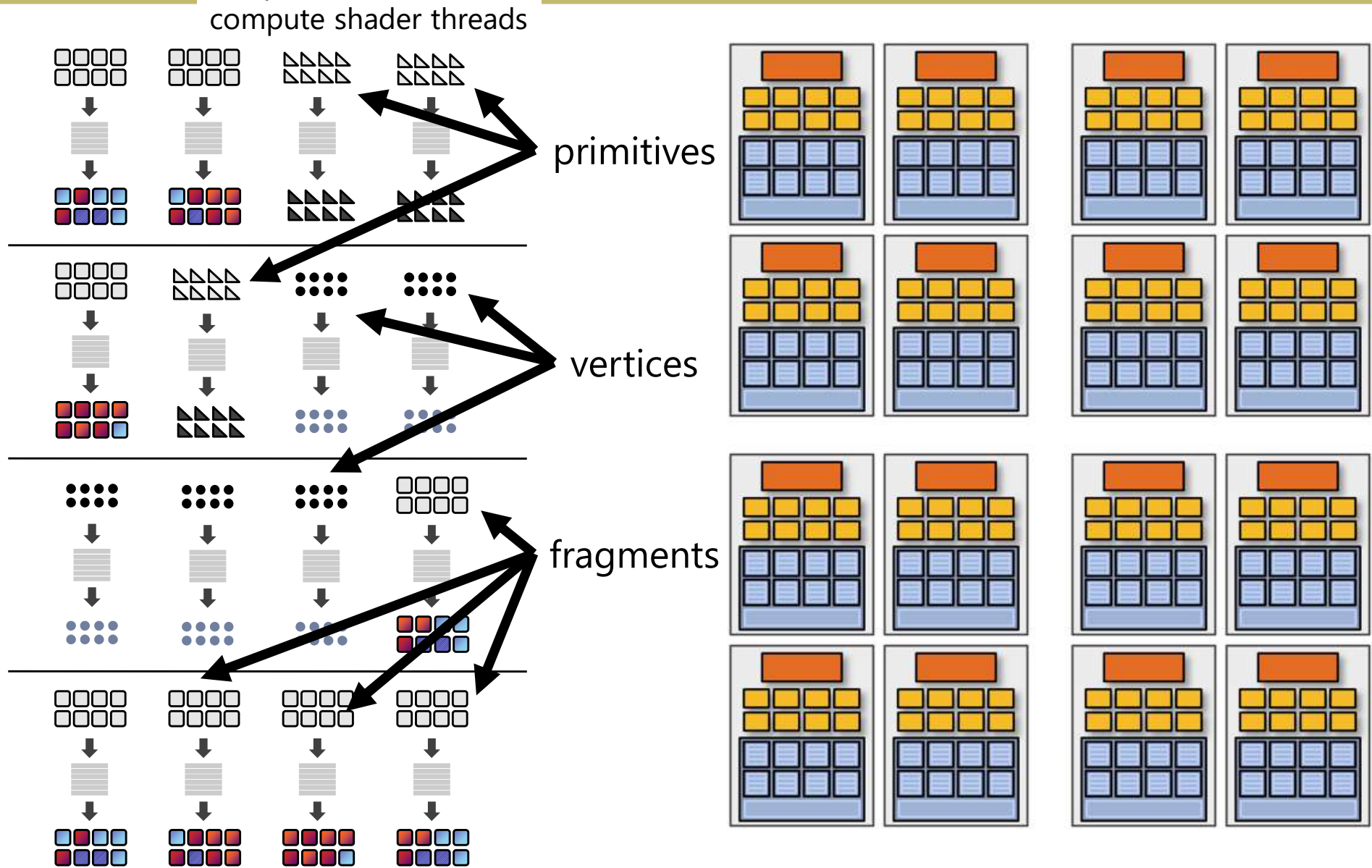


128 fragments in parallel



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

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



Clarification

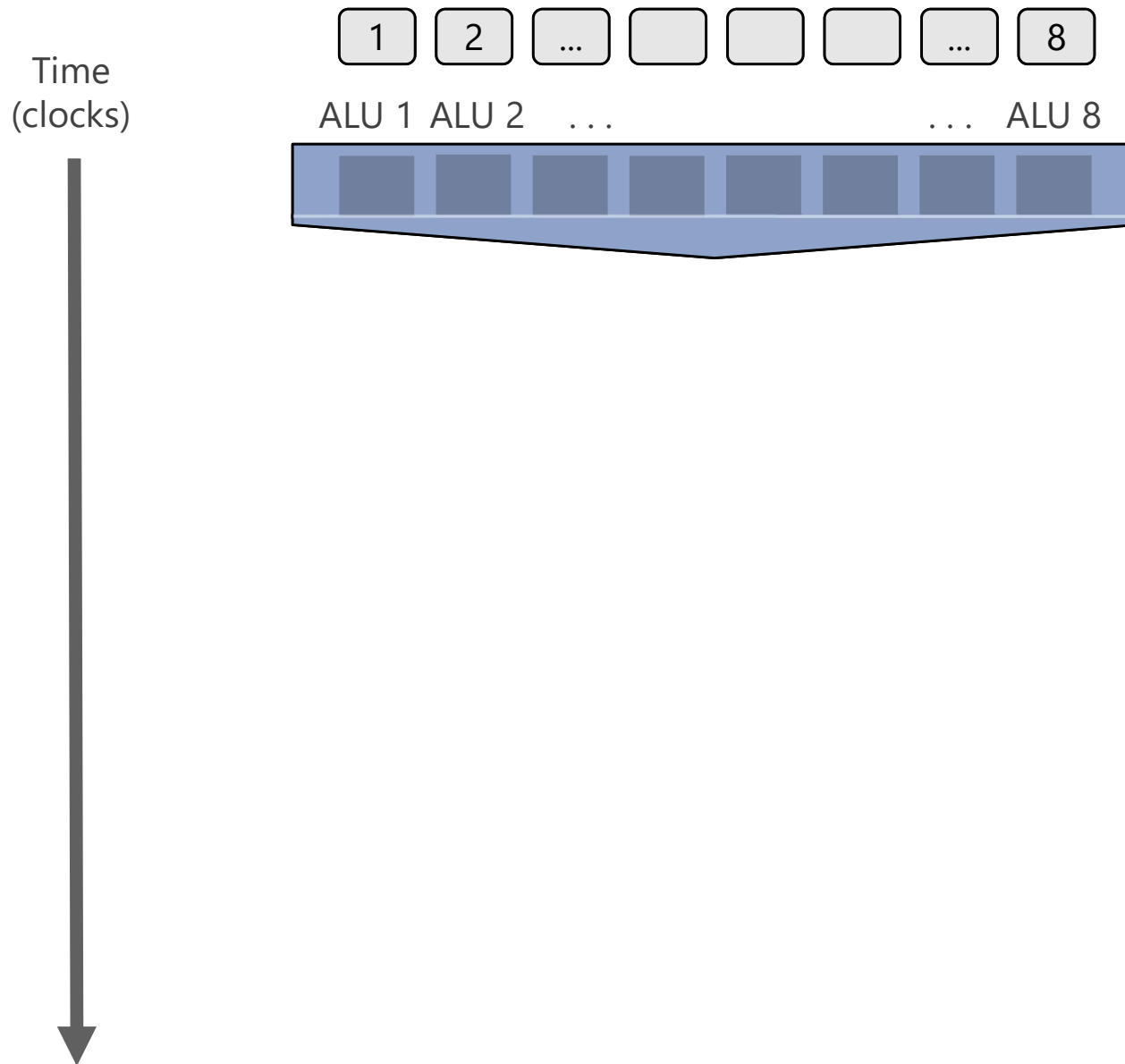
SIMD processing does not imply SIMD instructions

- 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?

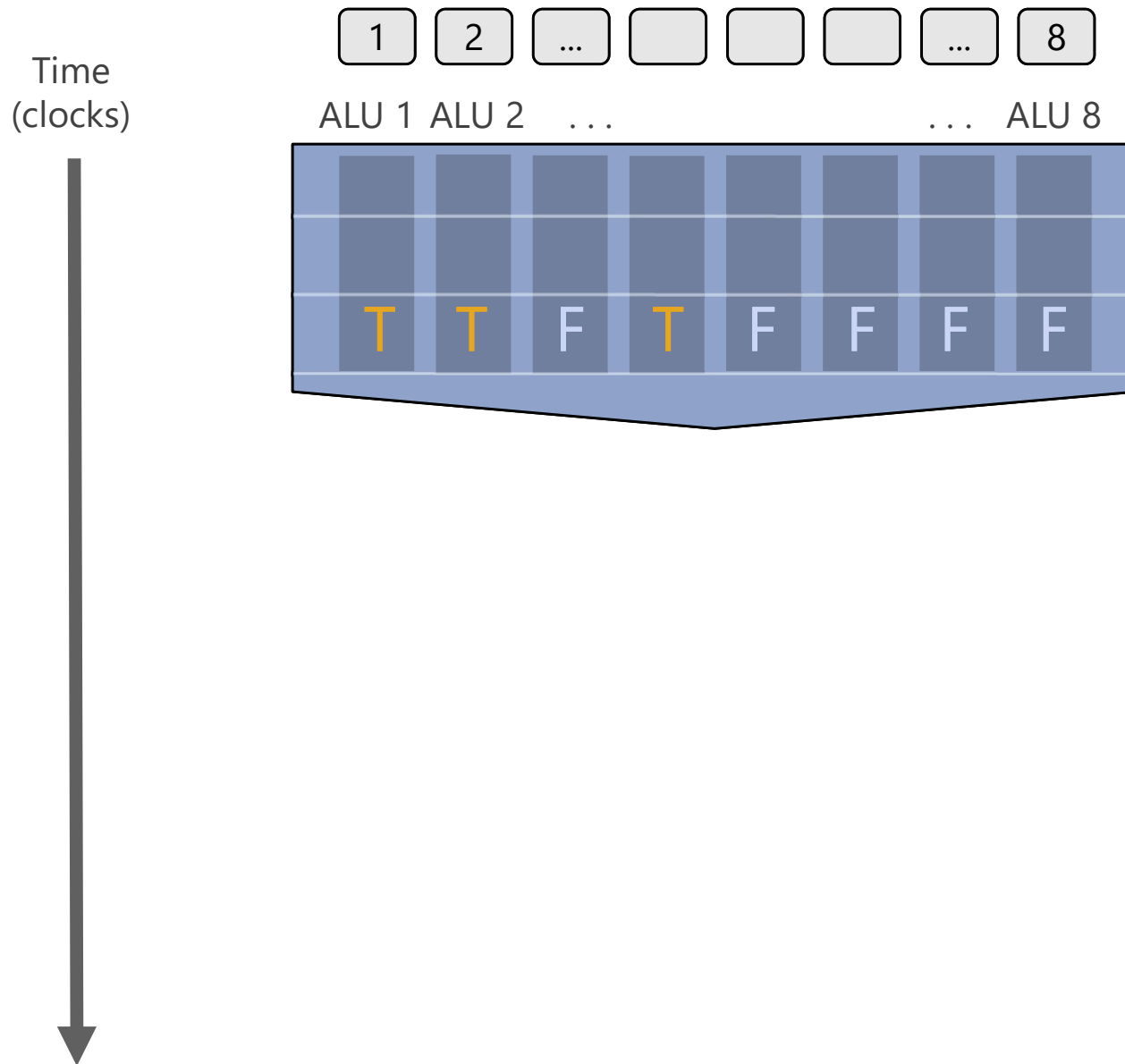


<unconditional
shader code>

```
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?

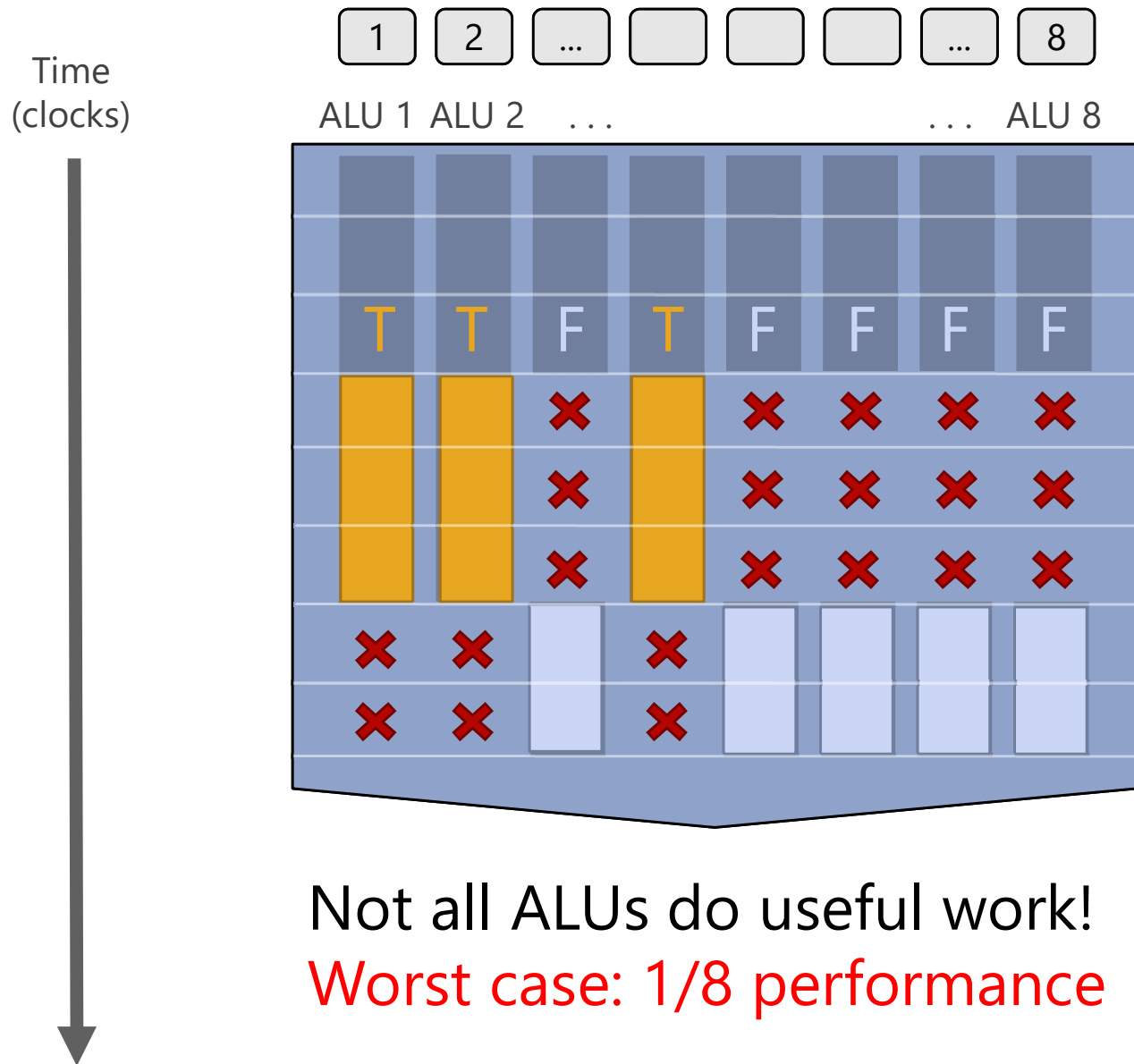


<unconditional
shader code>

```
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 {
```

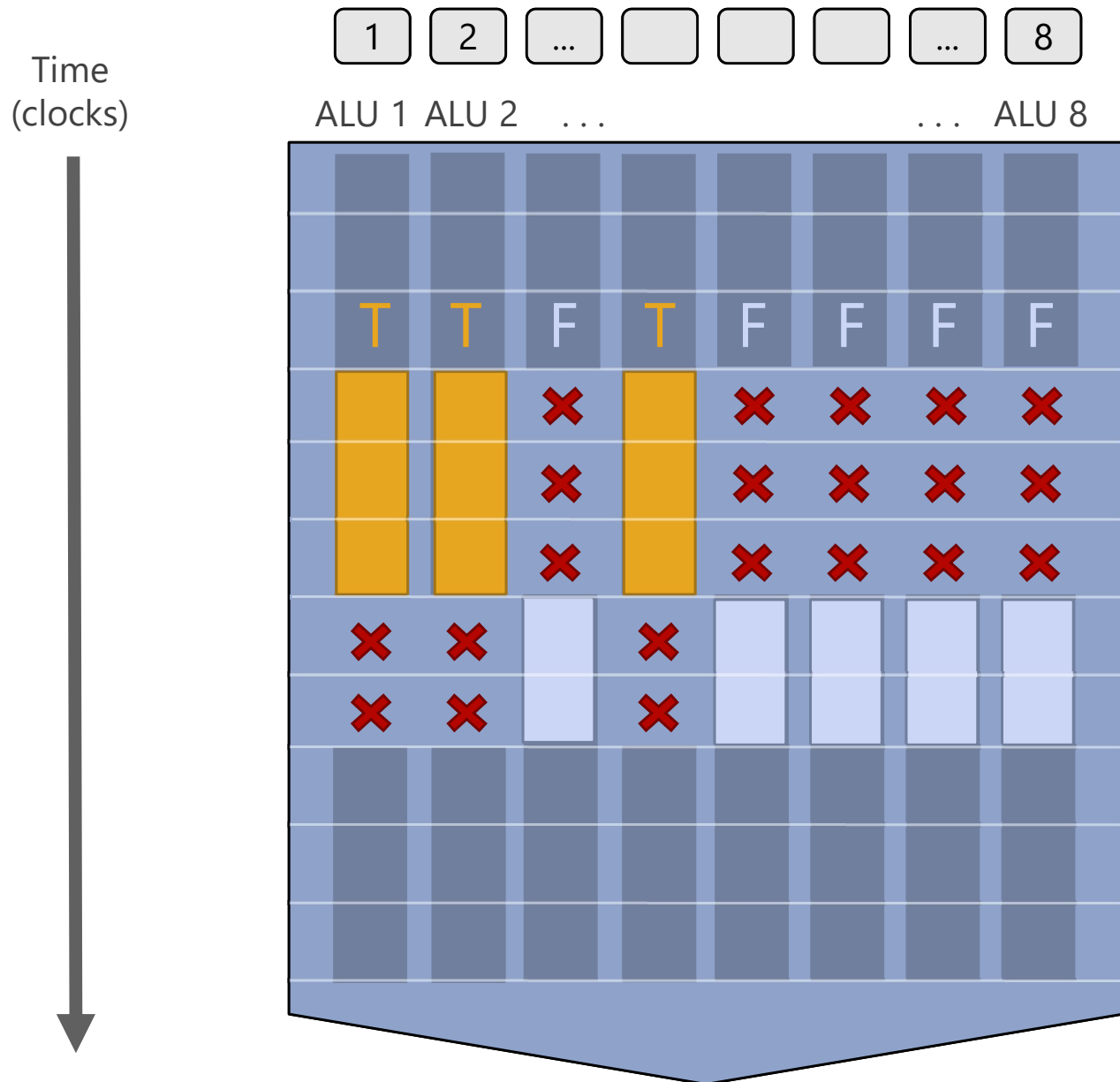
```
    x = 0;
```

```
    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>

GPU Architecture

Big Idea #3

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, ...)

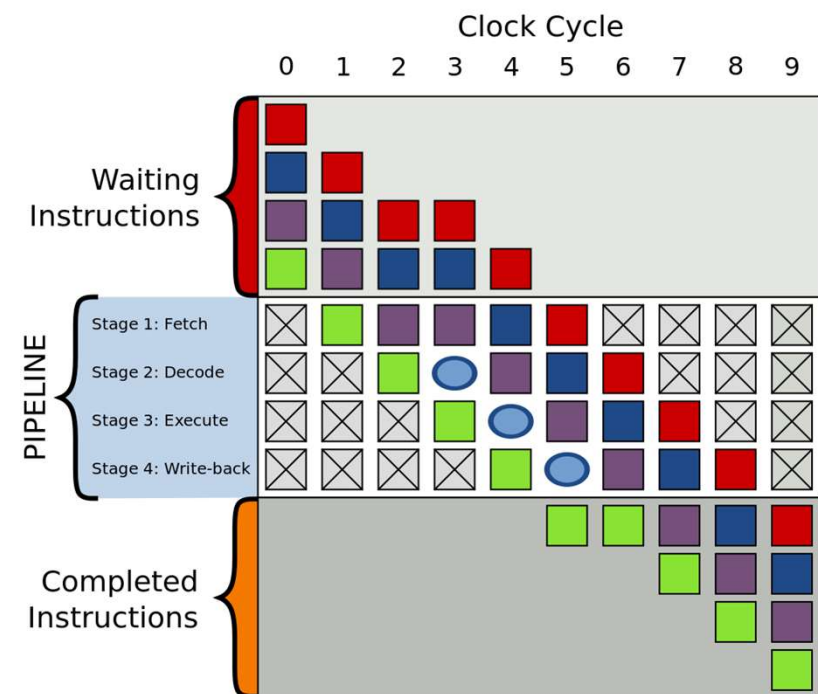
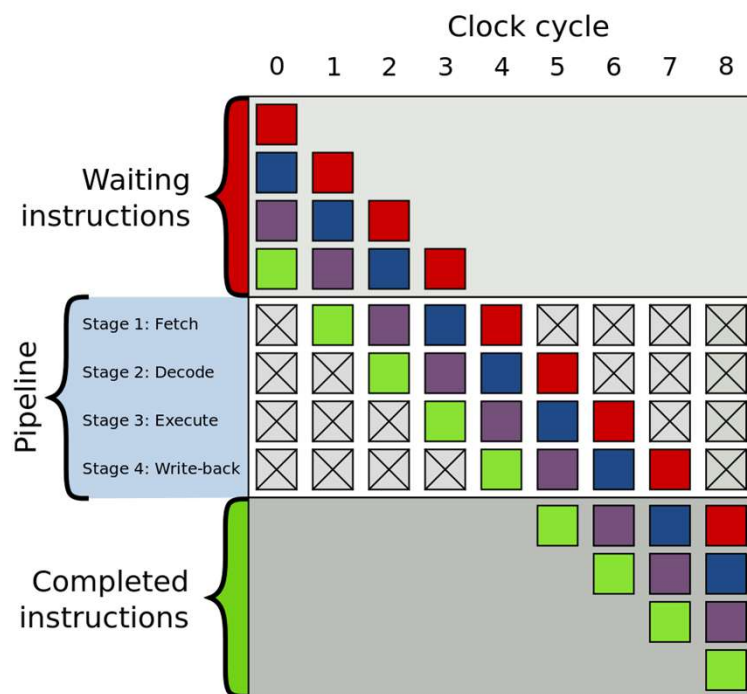
We've removed the fancy caches and logic that helps avoid stalls.

Interlude: Instruction Pipelining



Most common way to exploit *instruction-level parallelism* (ILP)

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



wikipedia

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

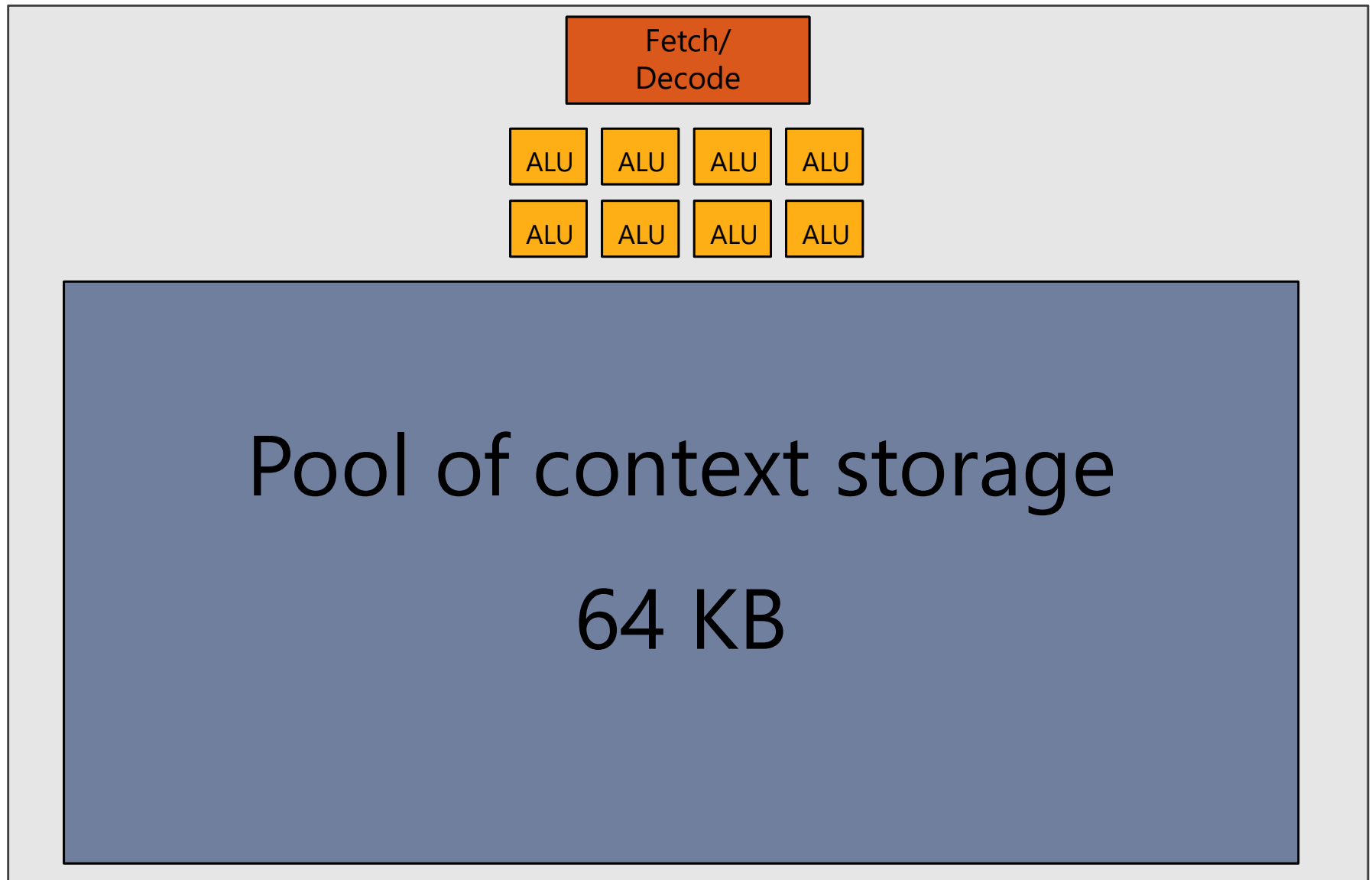
Idea #3: Interleave execution of groups

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

Idea #3:

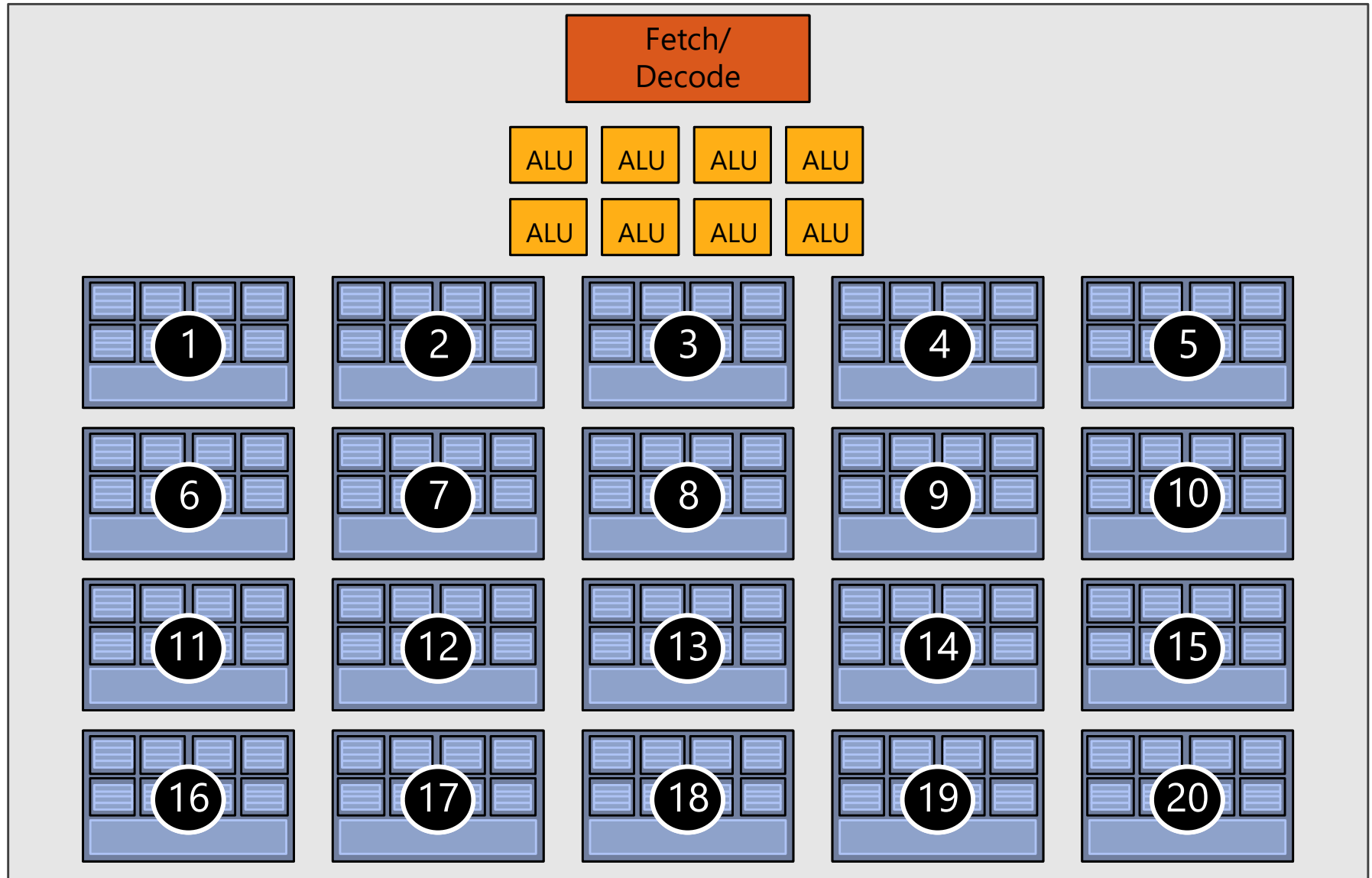
Interleave processing of many fragments on a single core to avoid stalls caused by high latency operations.

Idea #3: Store multiple group contexts

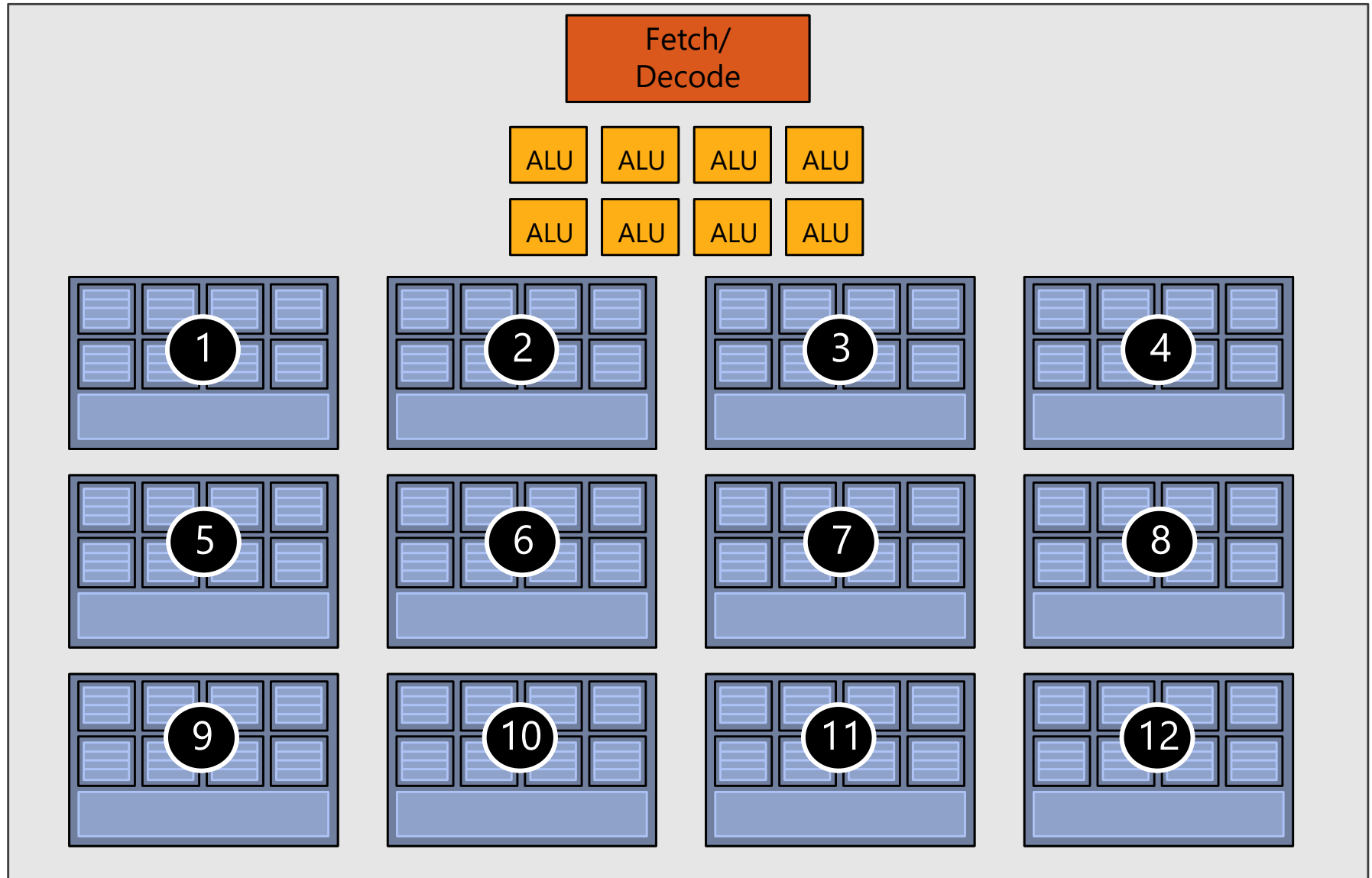


Twenty small contexts (few regs/thread)

(maximal latency hiding ability)

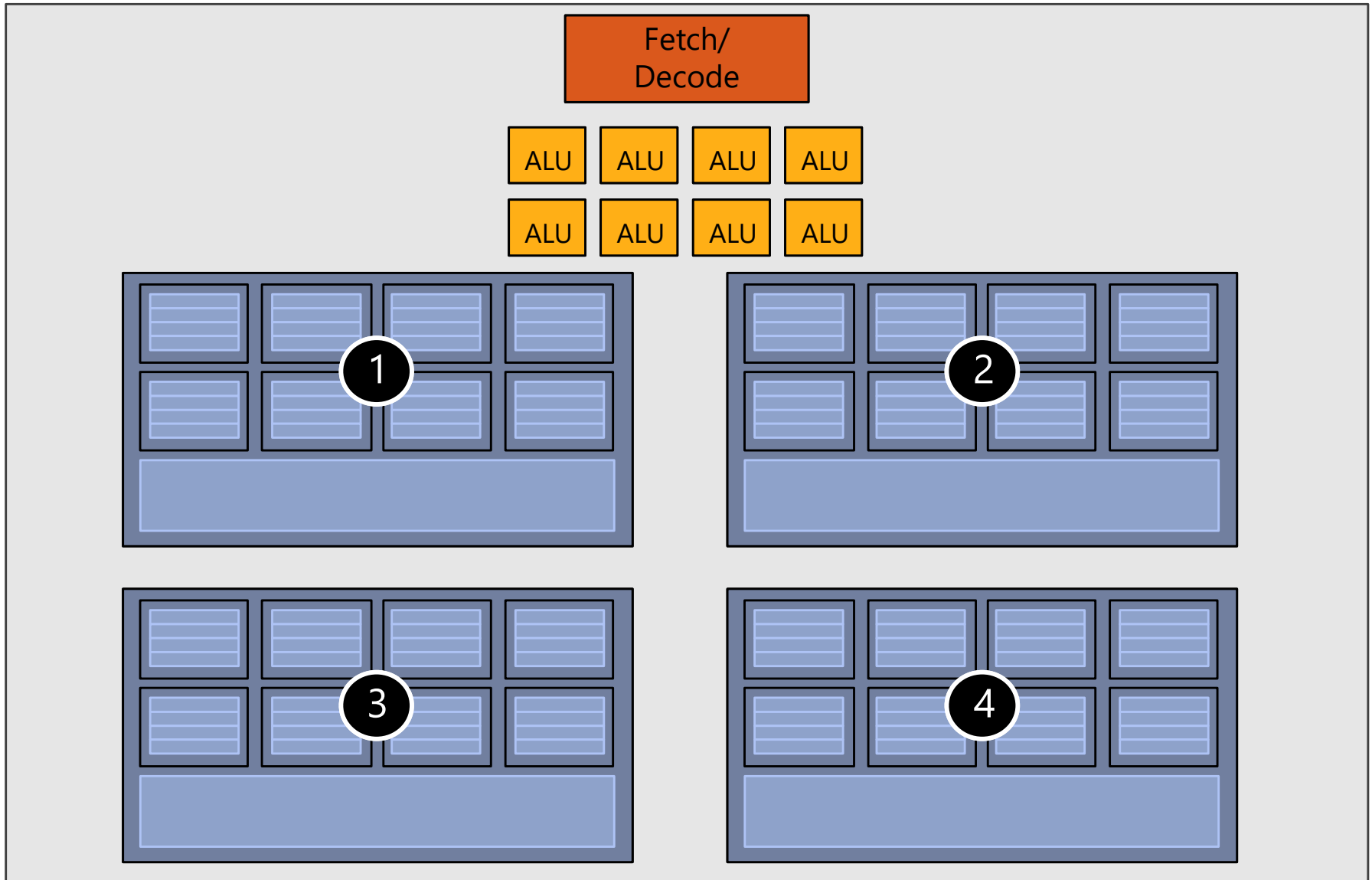


Twelve medium contexts (more regs/th.)



Four large contexts (many regs/thread)

(low latency hiding ability)



Thank you.