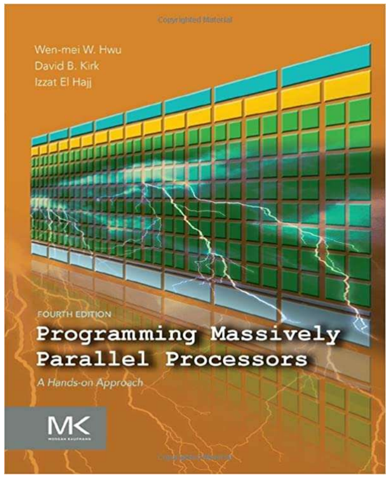
King Abdullah University of Science and Technology KAUST

CS 380 - GPU and GPGPU Programming
Lecture 6: GPU Architecture, Pt. 4

Markus Hadwiger, KAUST

Reading Assignment #3 (until Sep 22)

Read (required):



- Programming Massively Parallel Processors book, 4th edition, **Chapter 4** (*Compute architecture and scheduling*)

- NVIDIA CUDA C++ Programming Guide
  (current: v13.0.1, Sep 2, 2025)

`https://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf`

*Read **Chapter 5.6*** (Compute Capability);
*"Read" **Chapter 20.1** and **20.2*** (Compute Capabilities);
*Browse all of **Chapter 20*** (Compute Capabilities)
*Browse all of **Chapter 8.2*** (Maximize Utilization) and
Chapter 8.4 (Maximize Instruction Throughput)
CUDA C++ Programming Guide Chapter 8.4 now (since CUDA 13) actually refers to:
NVIDIA CUDA C++ Best Practices Guide, ***Chapter 12*** (Instruction Optimization)

`https://docs.nvidia.com/cuda/pdf/CUDA_C_Best_Practices_Guide.pdf`

GPU Architecture:
General Architecture

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

SIGGRAPH2009 NEW ORLEANS

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

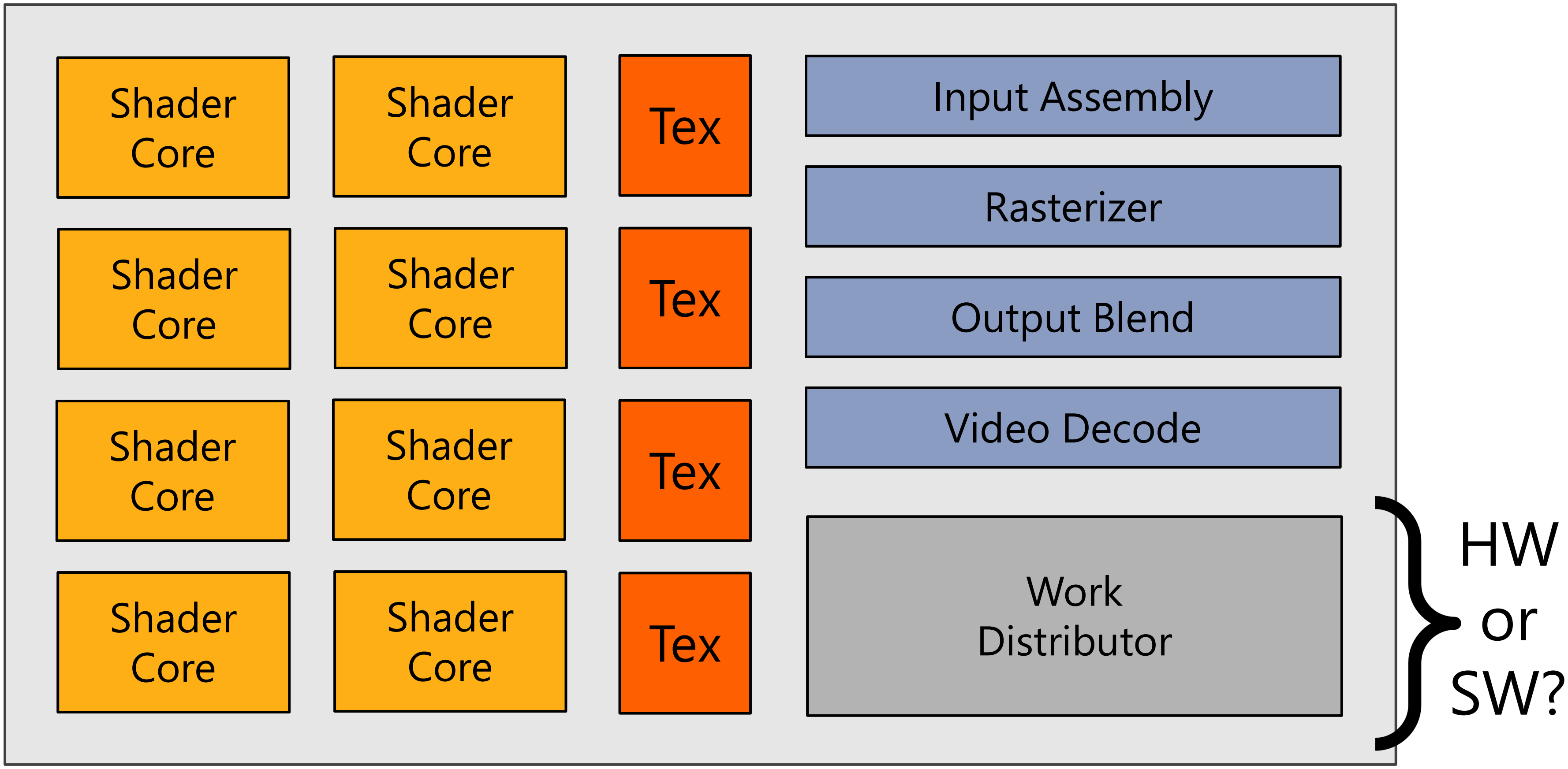
1. Use many "slimmed down cores," run them in parallel

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

GPUs are here! (usually)

3. Avoid latency stalls by interleaving execution of many groups of fragments
   - When one group stalls, work on another group

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

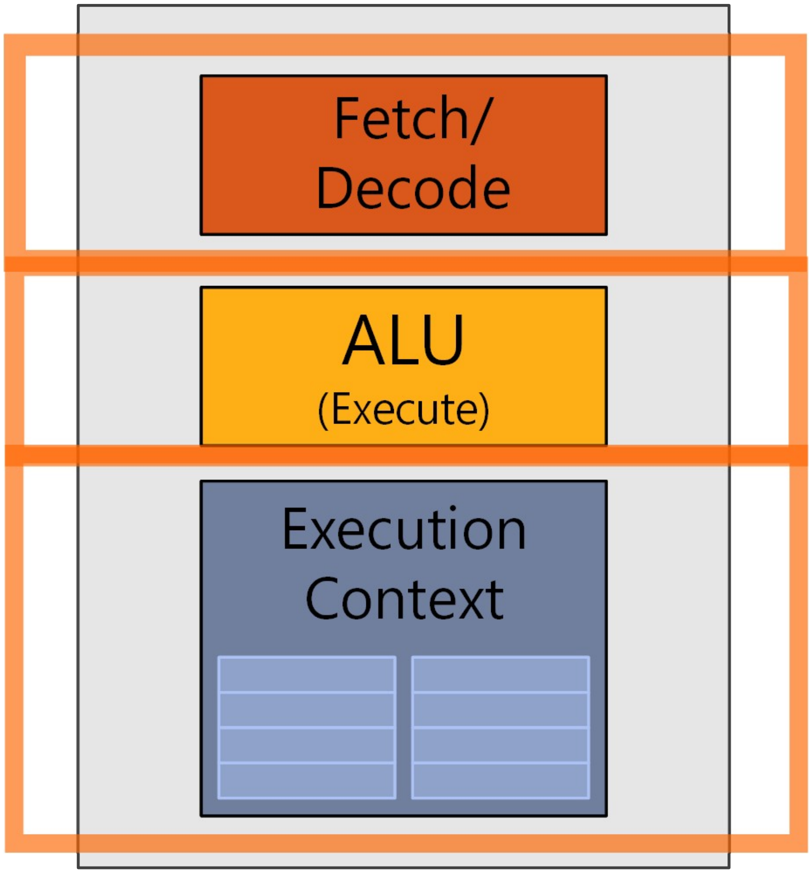
1 unshaded fragment input record

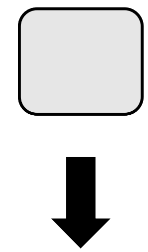
```
sampler mySamp;
Texture2D<float3> myTex;
float3 lightDir;

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

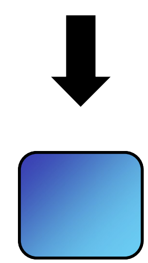
1 shaded fragment output record

Execute shader

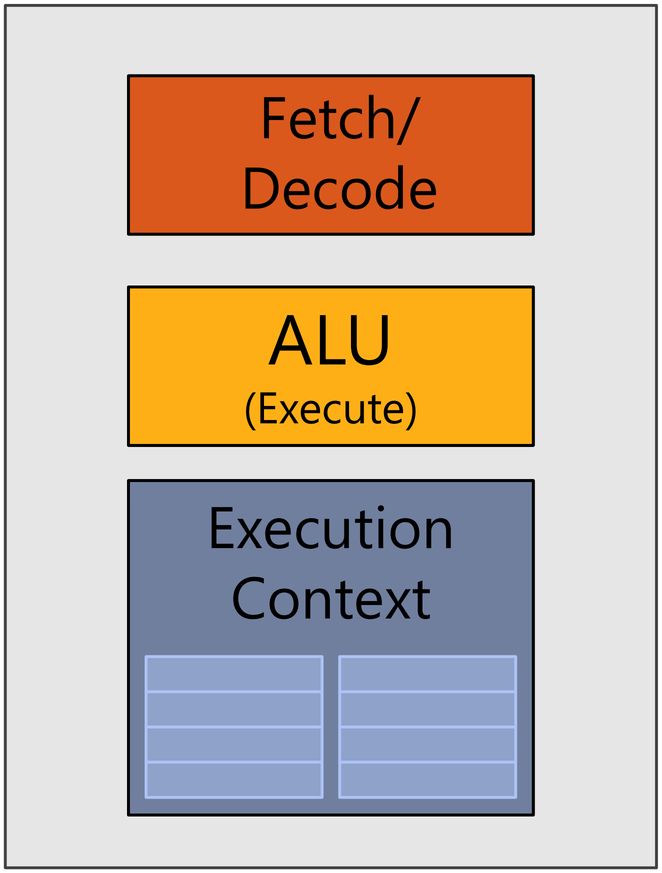


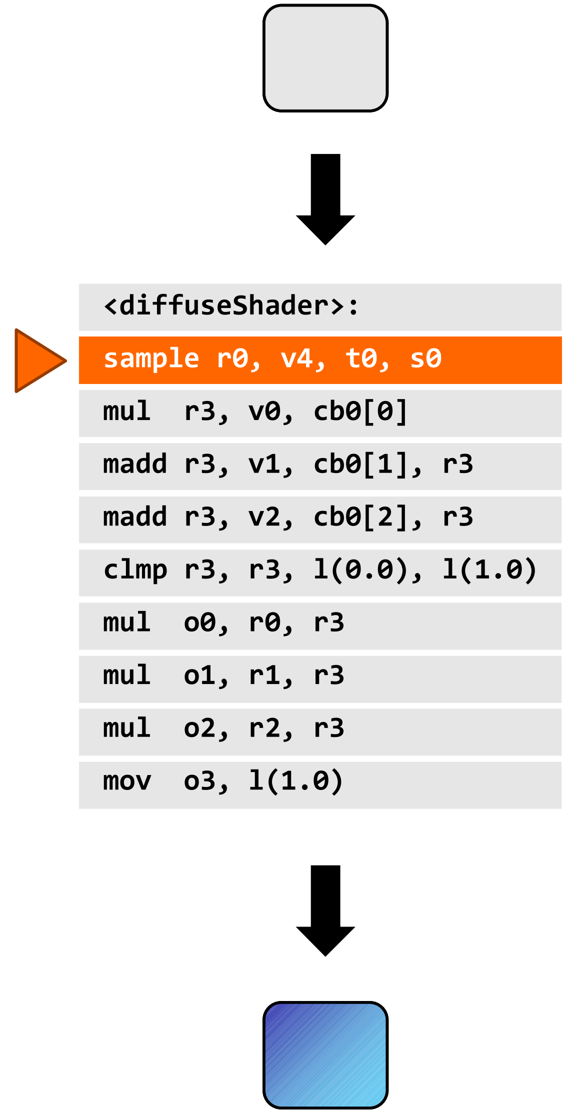


```
<diffuseShader>:
sample r0, v4, t0, s0
mul  r3, v0, cb0[0]
madd r3, v1, cb0[1], r3
madd r3, v2, cb0[2], r3
clmp r3, r3, l(0.0), l(1.0)
mul  o0, r0, r3
mul  o1, r1, r3
mul  o2, r2, r3
mov  o3, l(1.0)
```

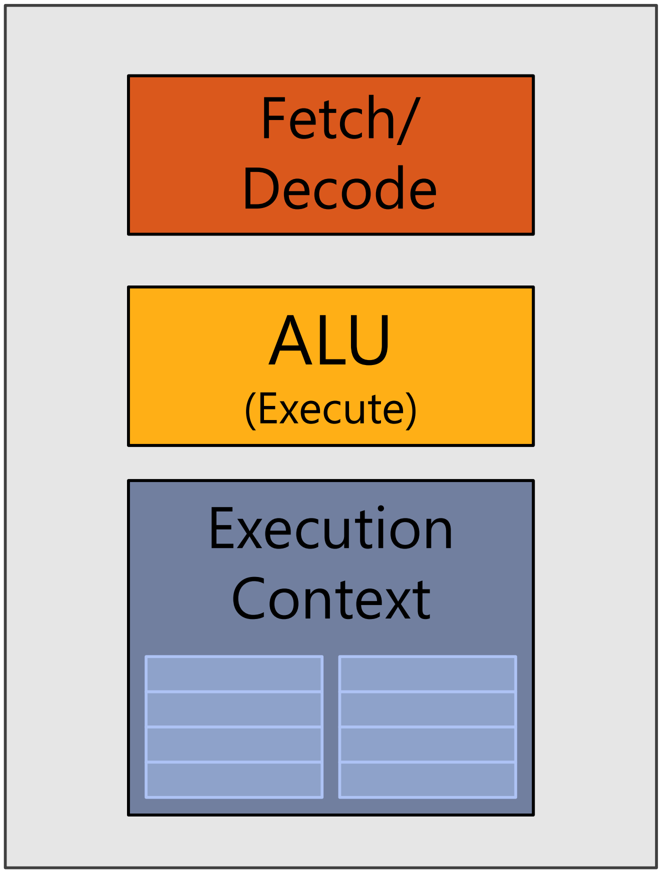


Execute shader



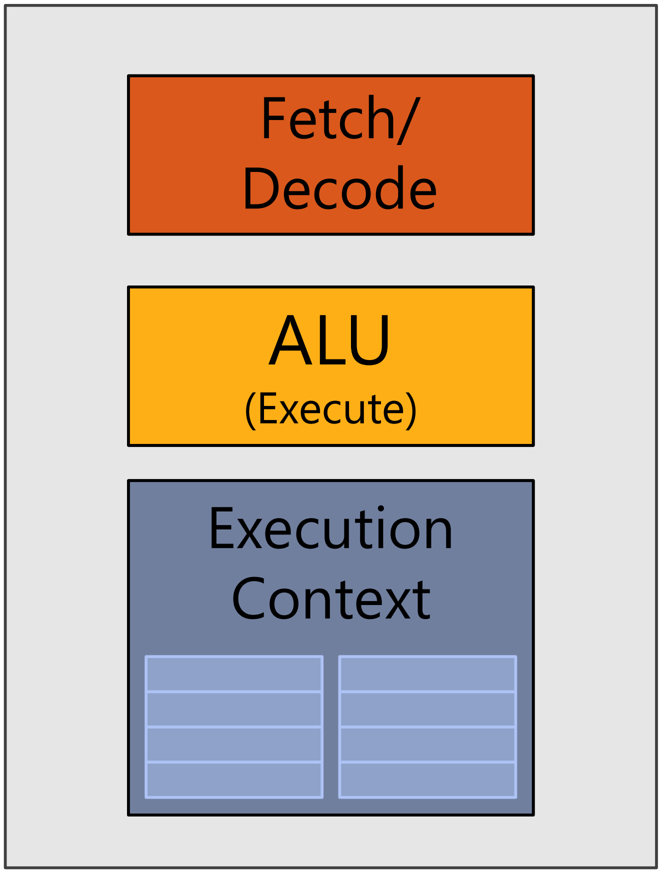


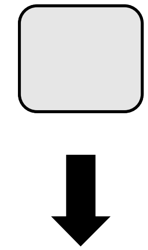
Execute shader



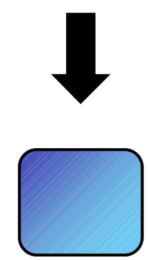

```
<diffuseShader>:
sample r0, v4, t0, s0
mul  r3, v0, cb0[0]
madd r3, v1, cb0[1], r3
madd r3, v2, cb0[2], r3
clmp r3, r3, l(0.0), l(1.0)
mul  o0, r0, r3
mul  o1, r1, r3
mul  o2, r2, r3
mov  o3, l(1.0)
```

Execute shader

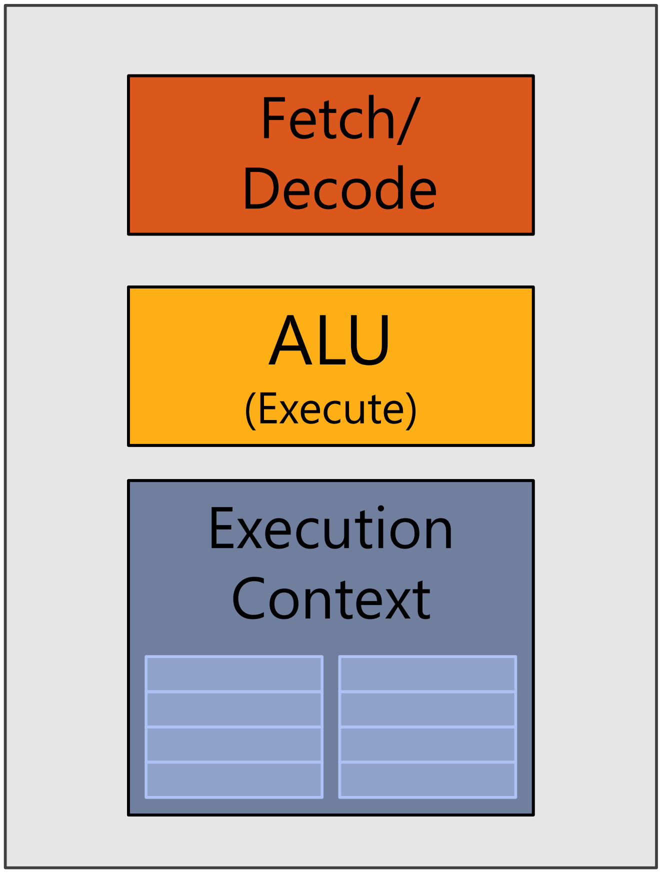




```
<diffuseShader>:
sample r0, v4, t0, s0
mul  r3, v0, cb0[0]
madd r3, v1, cb0[1], r3
madd r3, v2, cb0[2], r3
clmp r3, r3, l(0.0), l(1.0)
mul  o0, r0, r3
mul  o1, r1, r3
mul  o2, r2, r3
mov  o3, l(1.0)
```

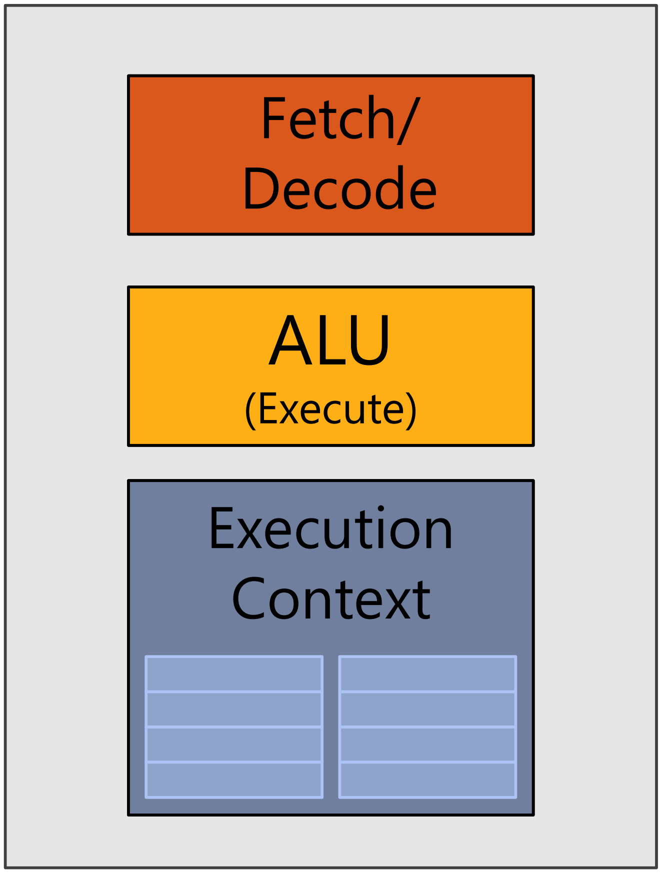


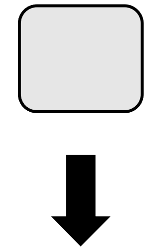
Execute shader



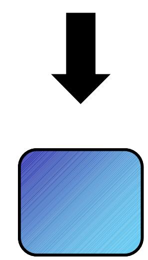

```
<diffuseShader>:
sample r0, v4, t0, s0
mul  r3, v0, cb0[0]
madd r3, v1, cb0[1], r3
madd r3, v2, cb0[2], r3
clmp r3, r3, l(0.0), l(1.0)
mul  o0, r0, r3
mul  o1, r1, r3
mul  o2, r2, r3
mov  o3, l(1.0)
```

Execute shader



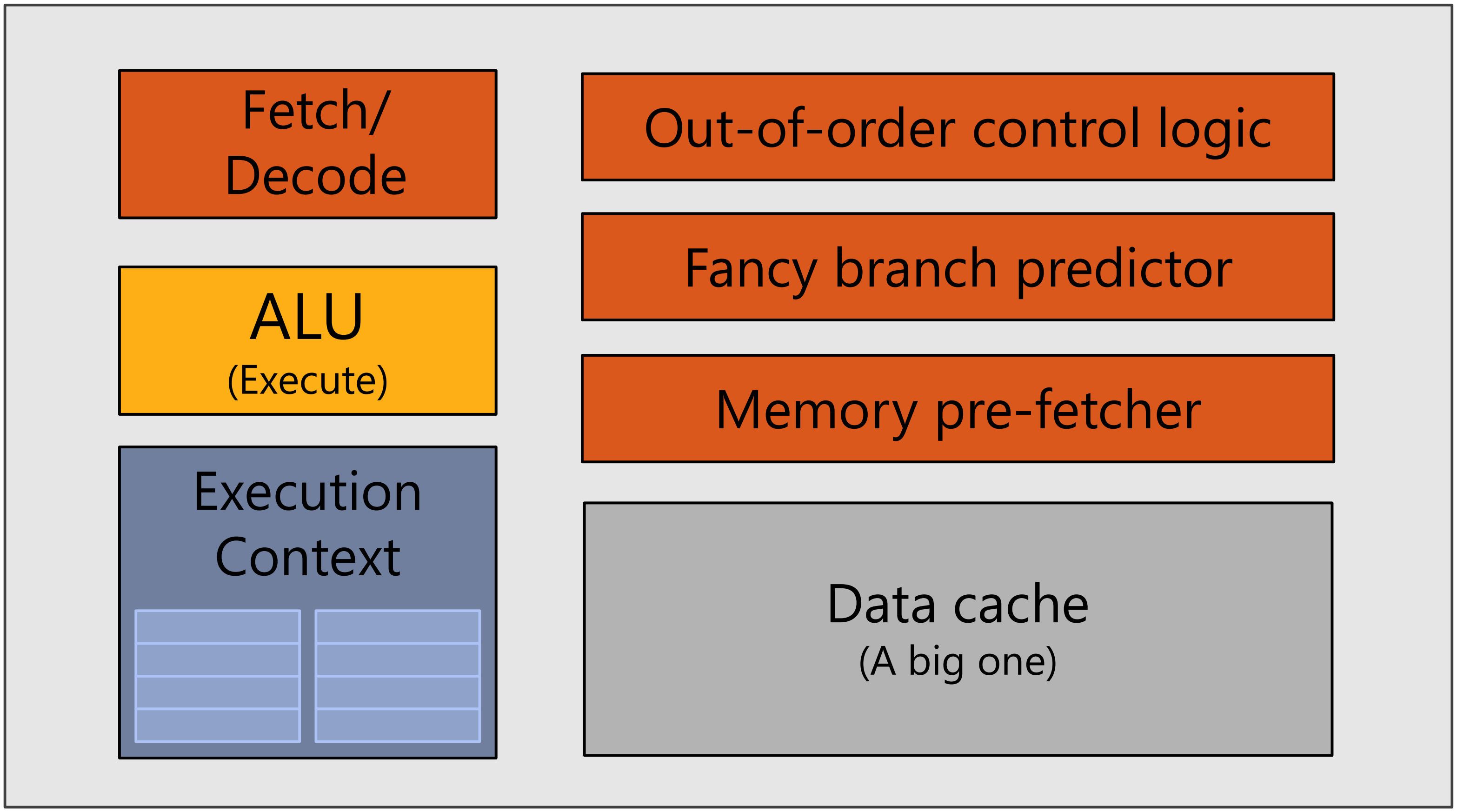


```
<diffuseShader>:
sample r0, v4, t0, s0
mul  r3, v0, cb0[0]
madd r3, v1, cb0[1], r3
madd r3, v2, cb0[2], r3
clmp r3, r3, l(0.0), l(1.0)
mul  o0, r0, r3
mul  o1, r1, r3
mul  o2, r2, r3
mov  o3, l(1.0)
```

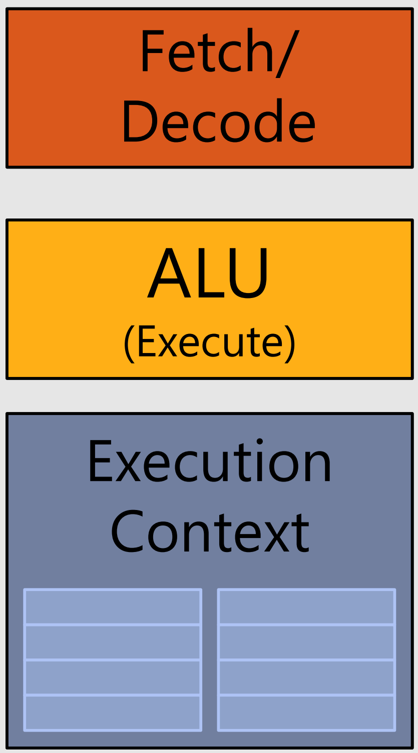


GPU Architecture
Big Idea #1

CPU-“style” cores



Idea #1: Slim down

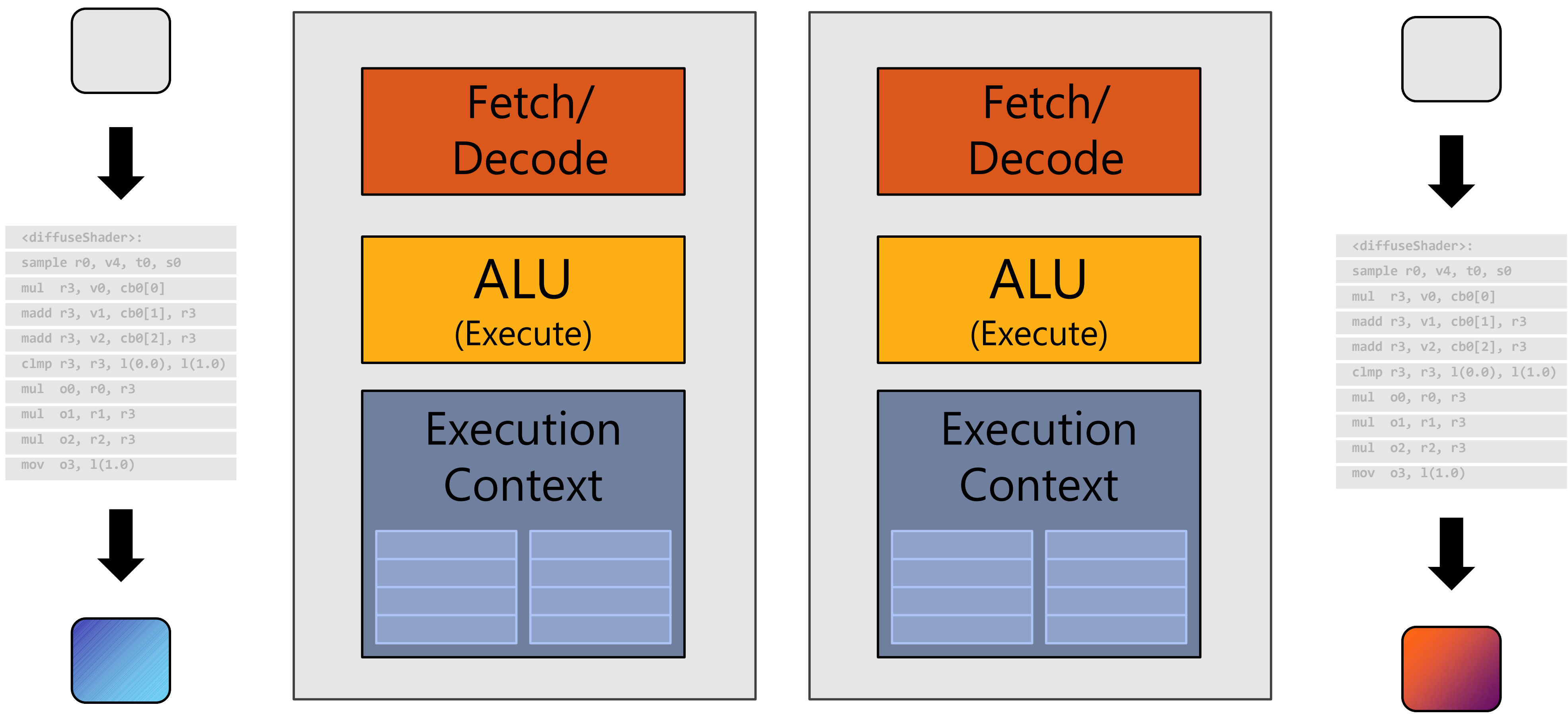


Idea #1:

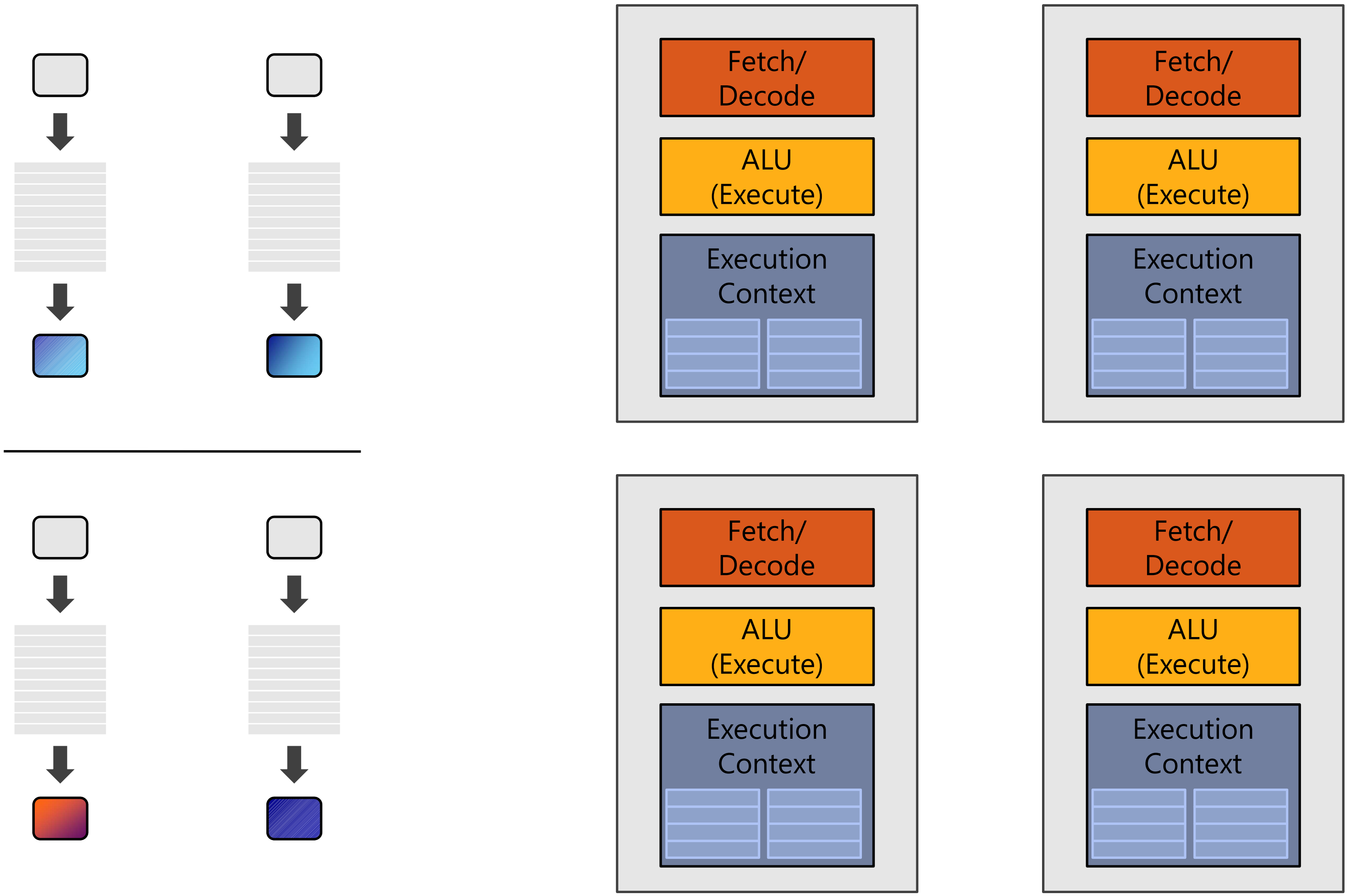
Remove components that help a single instruction stream run fast (i.e., that reduce latency of a single instruction stream)

Two cores (two fragments in parallel)

fragment 1 fragment 2



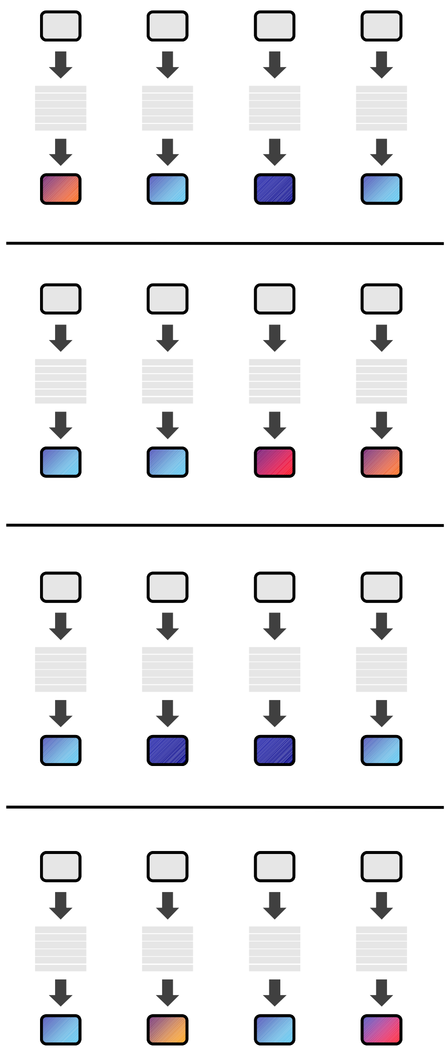
Four cores (four fragments in parallel)



Sixteen cores (sixteen fragments in parallel)

16 cores = 16 simultaneous instruction streams

Instruction stream sharing

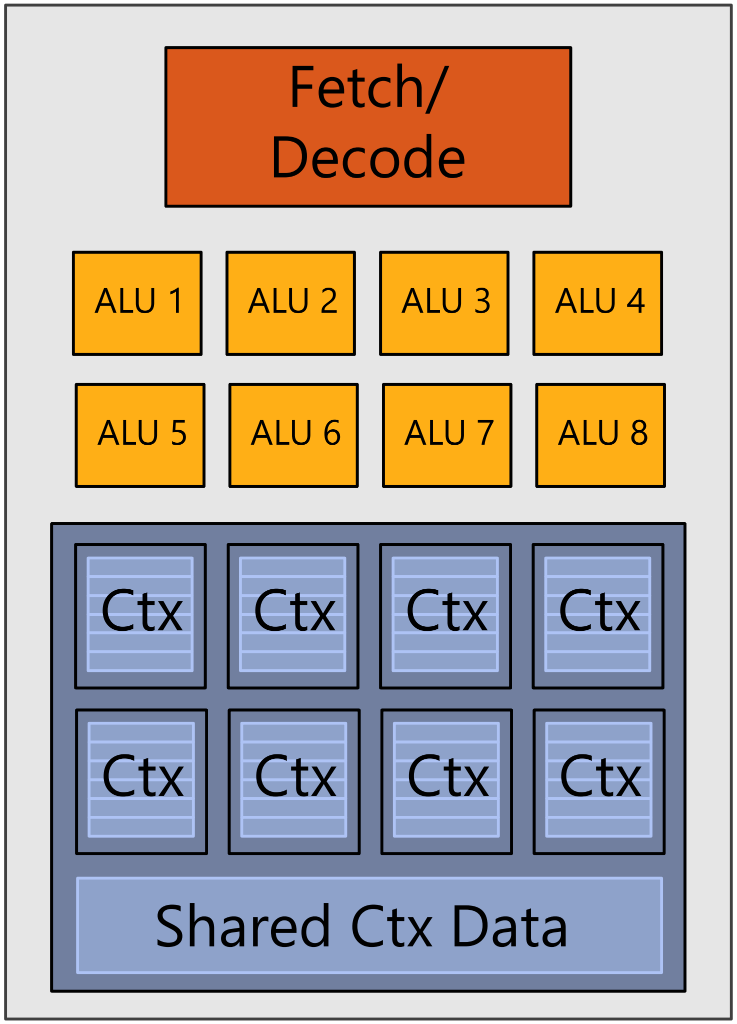


But… many fragments should be able to share an instruction stream! → **big idea #2 !**

```
<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



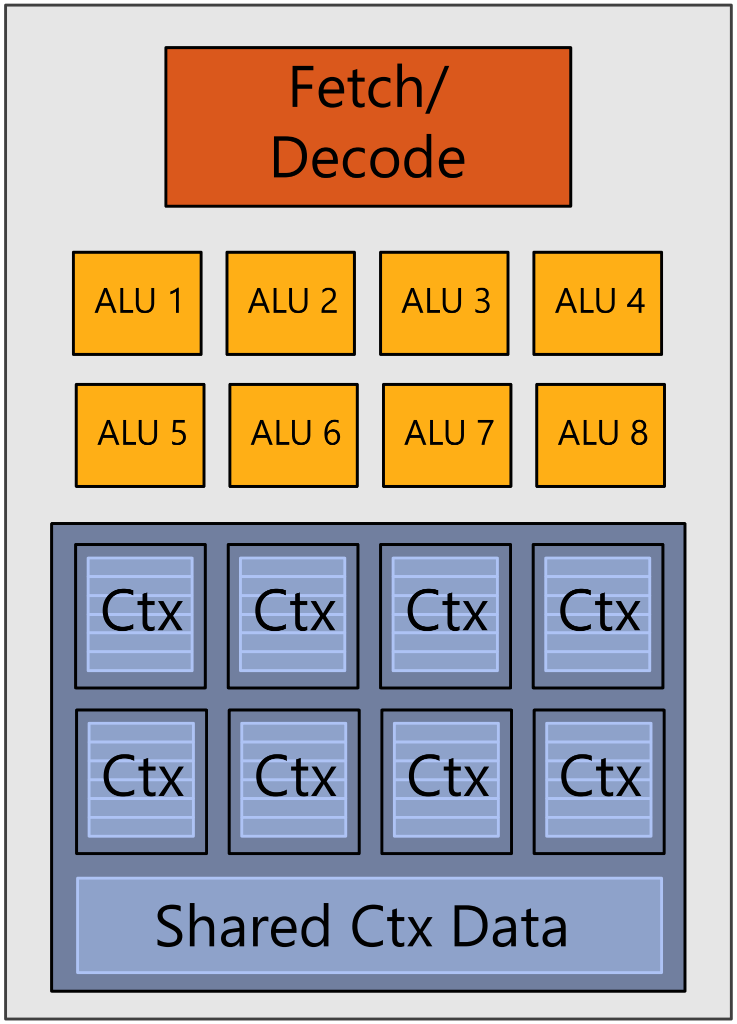
Idea #2:

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

SIMD processing

(or SIMT, SPMD)

How does shader execution behave?

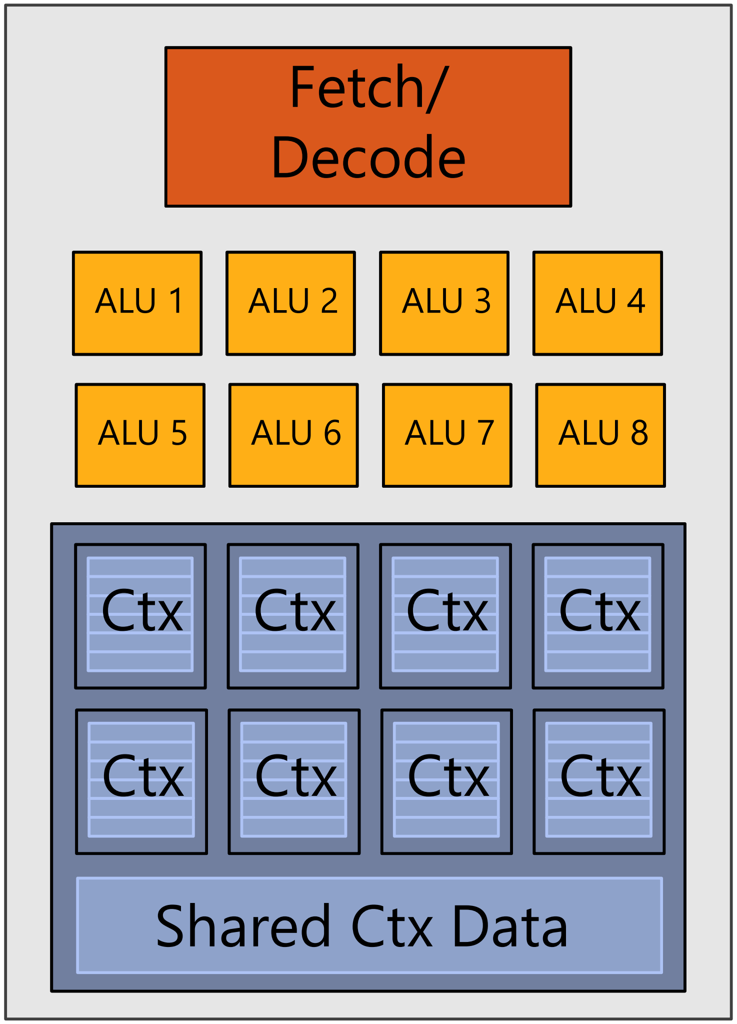



```
<diffuseShader>:
sample r0, v4, t0, s0
mul  r3, v0, cb0[0]
madd r3, v1, cb0[1], r3
madd r3, v2, cb0[2], r3
clmp r3, r3, l(0.0), l(1.0)
mul  o0, r0, r3
mul  o1, r1, r3
mul  o2, r2, r3
mov  o3, l(1.0)
```

Original compiled shader:

Processes one fragment
using scalar ops on scalar registers

How does shader execution behave?

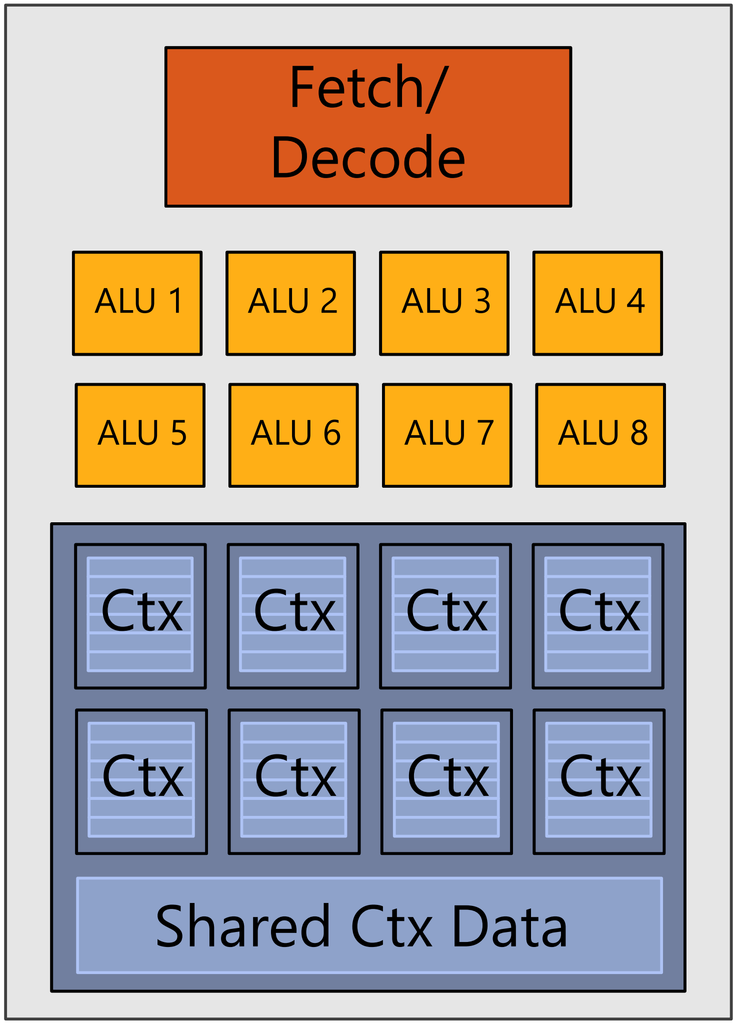


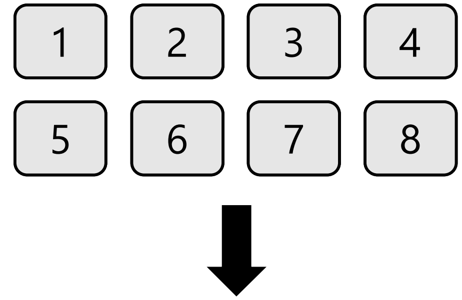

```
<VEC8_diffuseShader>:
VEC8_sample vec_r0, vec_v4, t0, vec_s0
VEC8_mul  vec_r3, vec_v0, cb0[0]
VEC8_madd vec_r3, vec_v1, cb0[1], vec_r3
VEC8_madd vec_r3, vec_v2, cb0[2], vec_r3
VEC8_clmp vec_r3, vec_r3, l(0.0), l(1.0)
VEC8_mul  vec_o0, vec_r0, vec_r3
VEC8_mul  vec_o1, vec_r1, vec_r3
VEC8_mul  vec_o2,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

Actually executed shader:

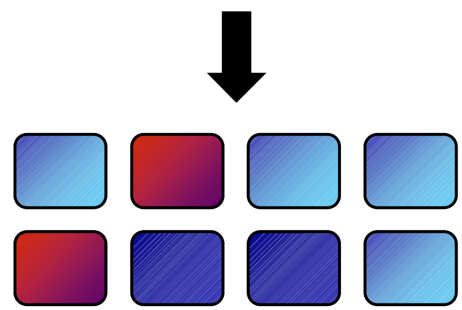
Processes 8 fragments
using “vector ops” on “vector registers”
(Caveat: This does NOT mean there are actual vector instructions/cores/regs! See later slide.)

How does shader execution behave?

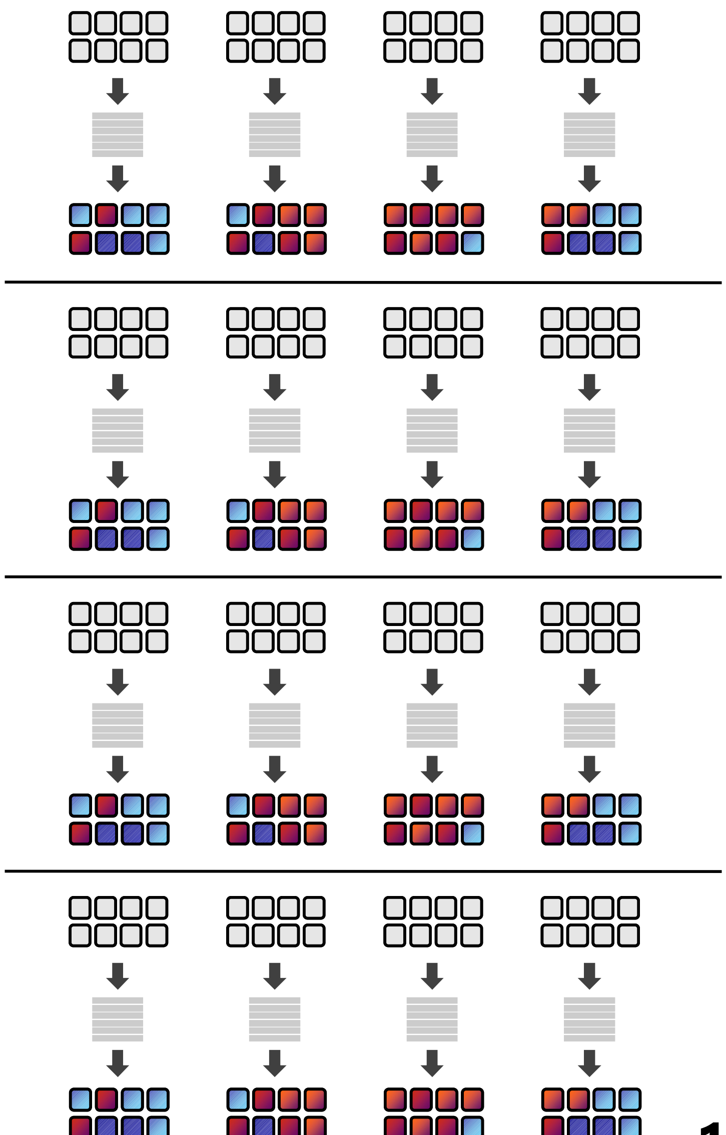


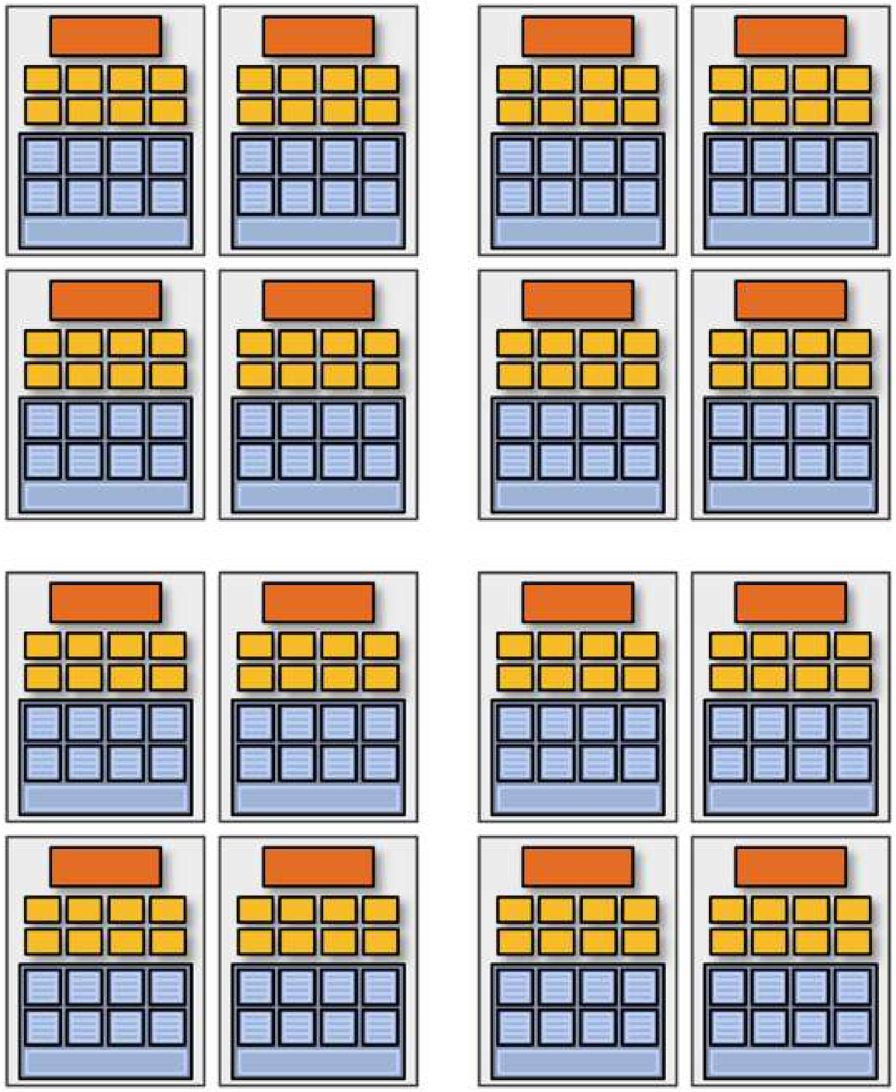



```
<VEC8_diffuseShader>:
VEC8_sample vec_r0, vec_v4, t0, vec_s0
VEC8_mul  vec_r3, vec_v0, cb0[0]
VEC8_madd vec_r3, vec_v1, cb0[1], vec_r3
VEC8_madd vec_r3, vec_v2, cb0[2], vec_r3
VEC8_clmp vec_r3, vec_r3, l(0.0), l(1.0)
VEC8_mul  vec_o0, vec_r0, vec_r3
VEC8_mul  vec_o1, vec_r1, vec_r3
VEC8_mul  vec_o2,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



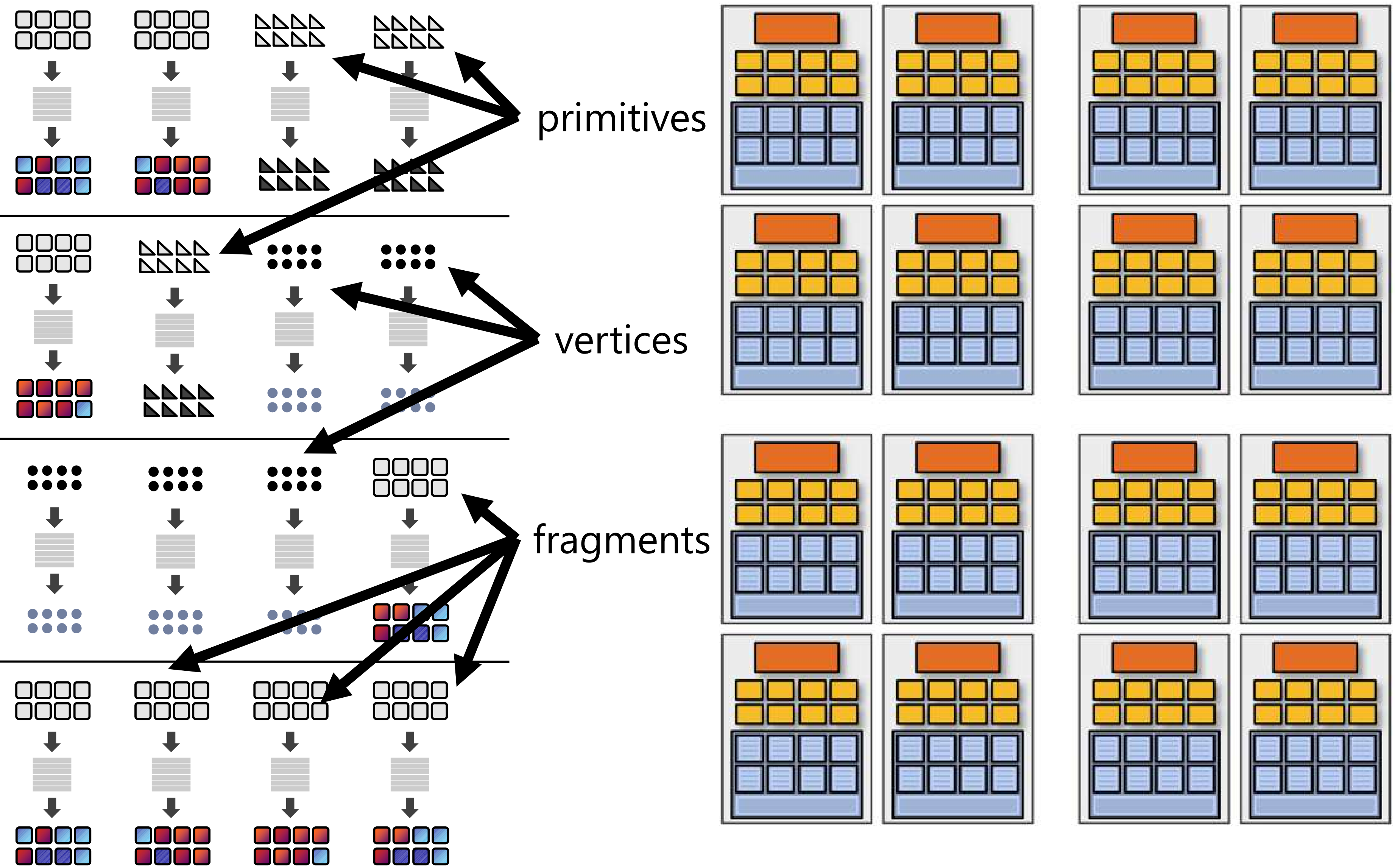
128 fragments in parallel





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

GPU Architecture Big Idea #3

(Teaser for next Lecture)

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.

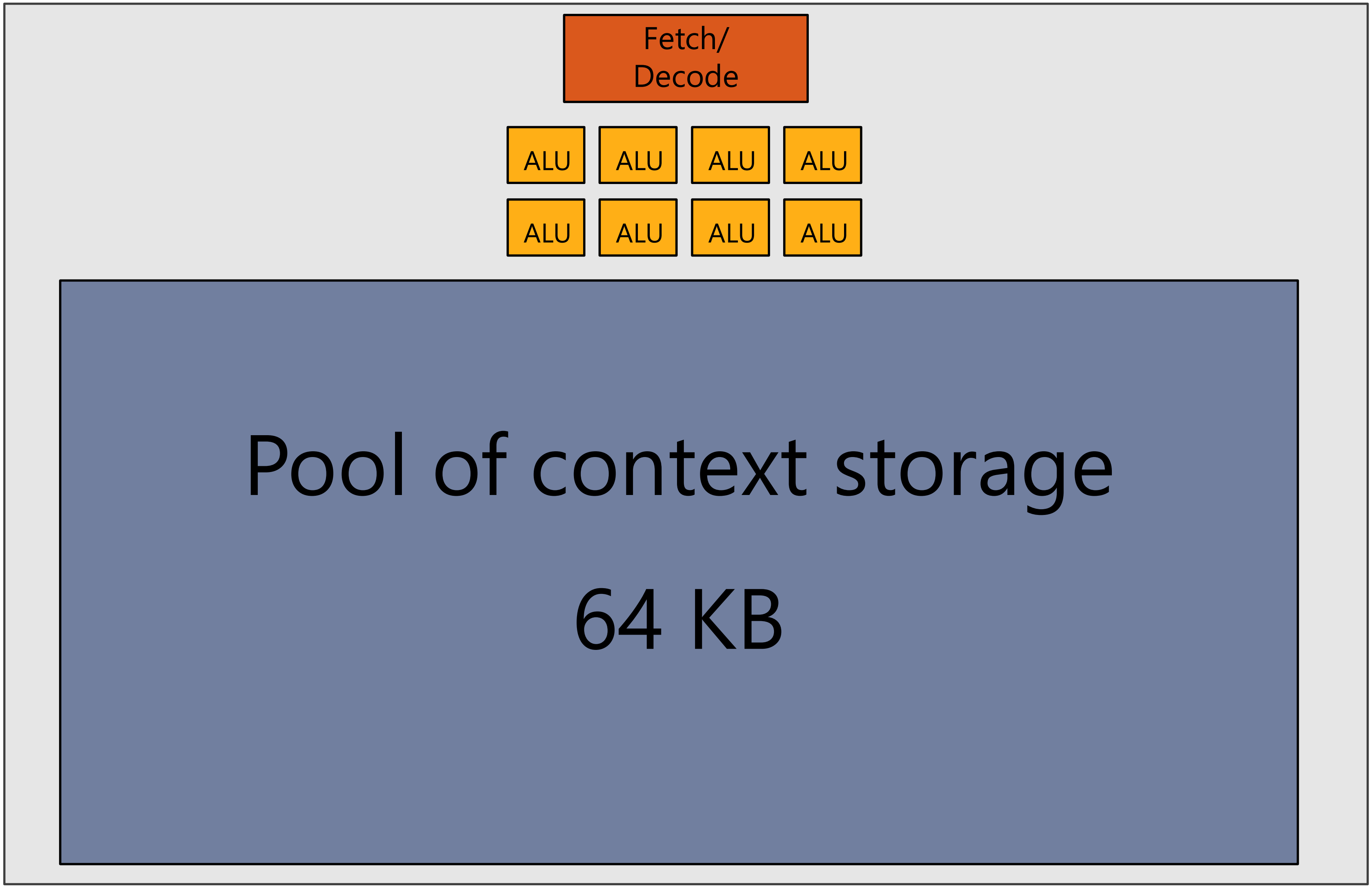
Idea #3: Interleave execution of groups

But we have LOTS of independent fragments.

Idea #3:

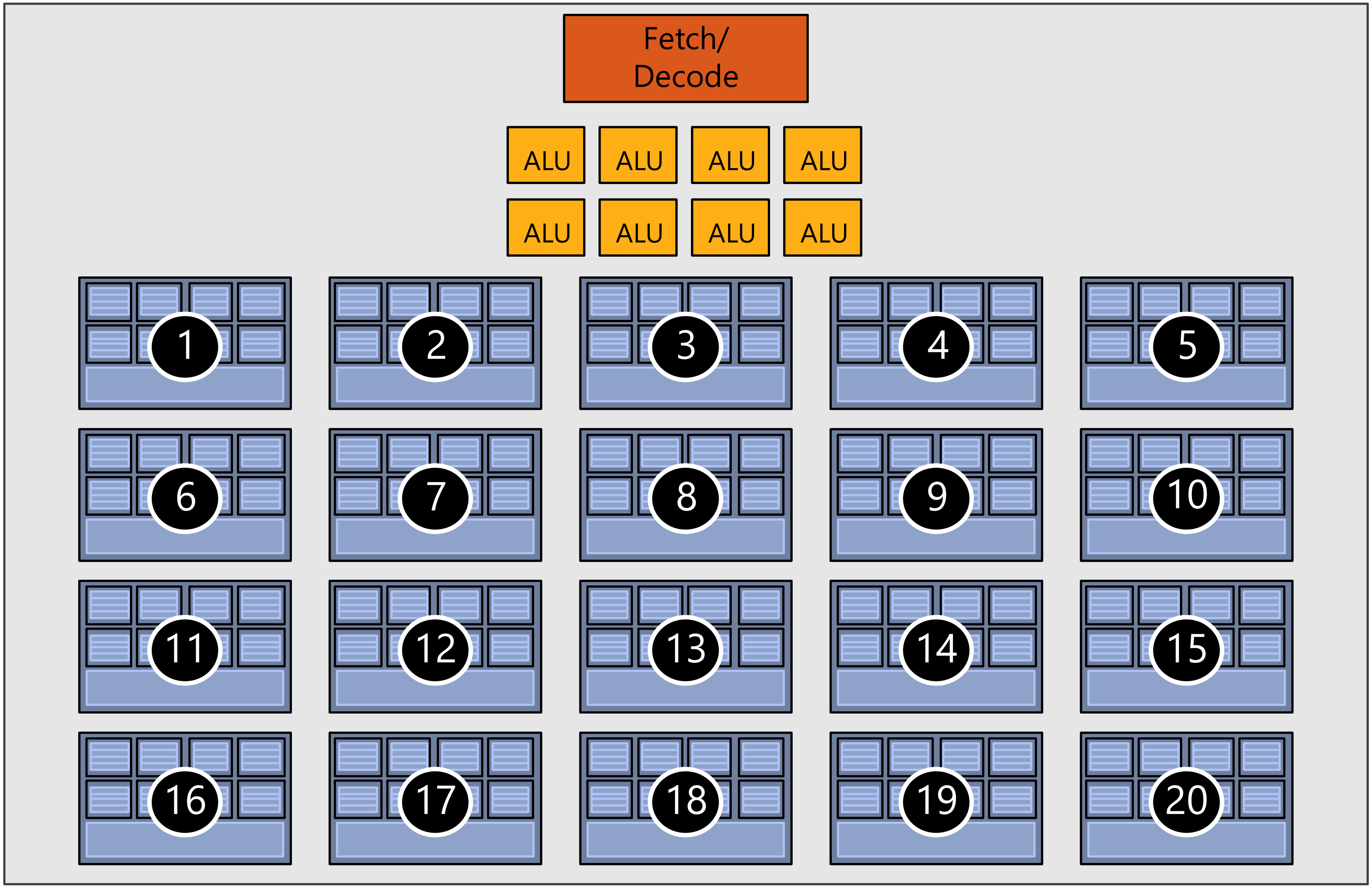
Interleave processing of many fragments on a single core to avoid stalls caused by high latency operations.

Idea #3: Store multiple group contexts

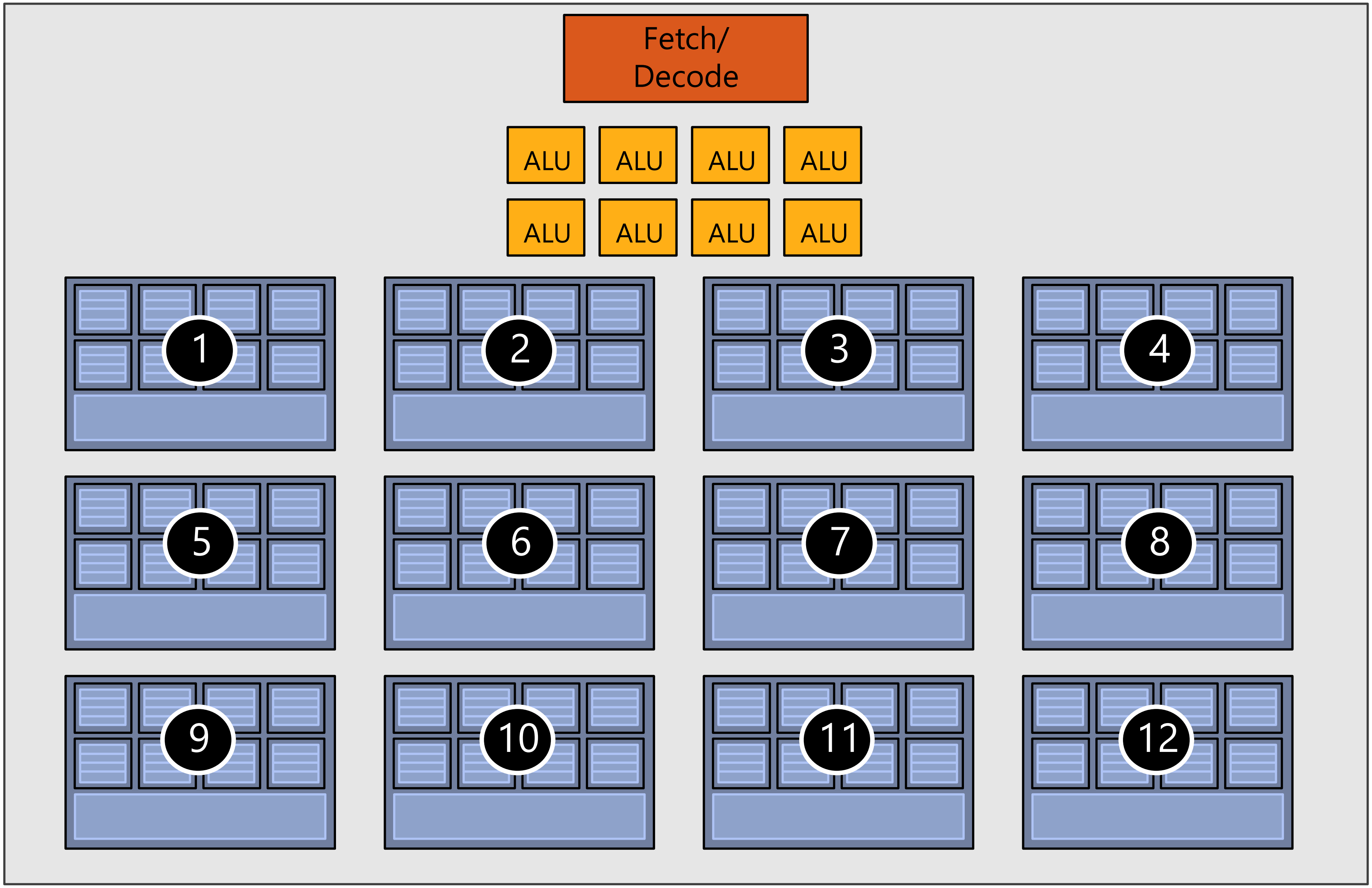


Twenty small contexts (few regs/thread)

(maximal latency hiding ability)

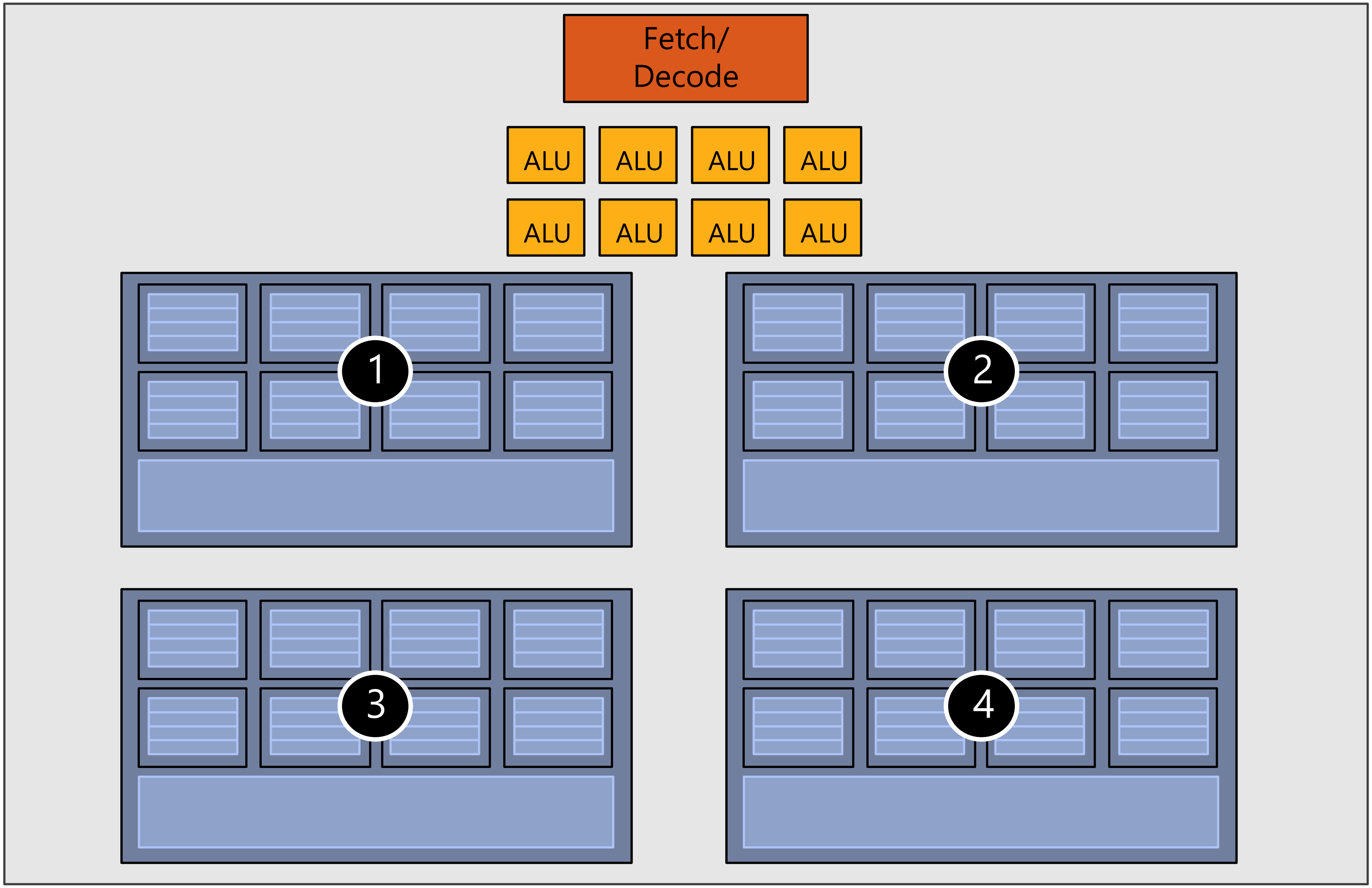


Twelve medium contexts (more regs/th.)



Four large contexts (many regs/thread)

(low latency hiding ability)



Thank you.