King Abdullah University of Science and Technology

KAUST

CS 380 - GPU and GPGPU Programming
Lecture 5: GPU Architecture, Pt. 3

Markus Hadwiger, KAUST

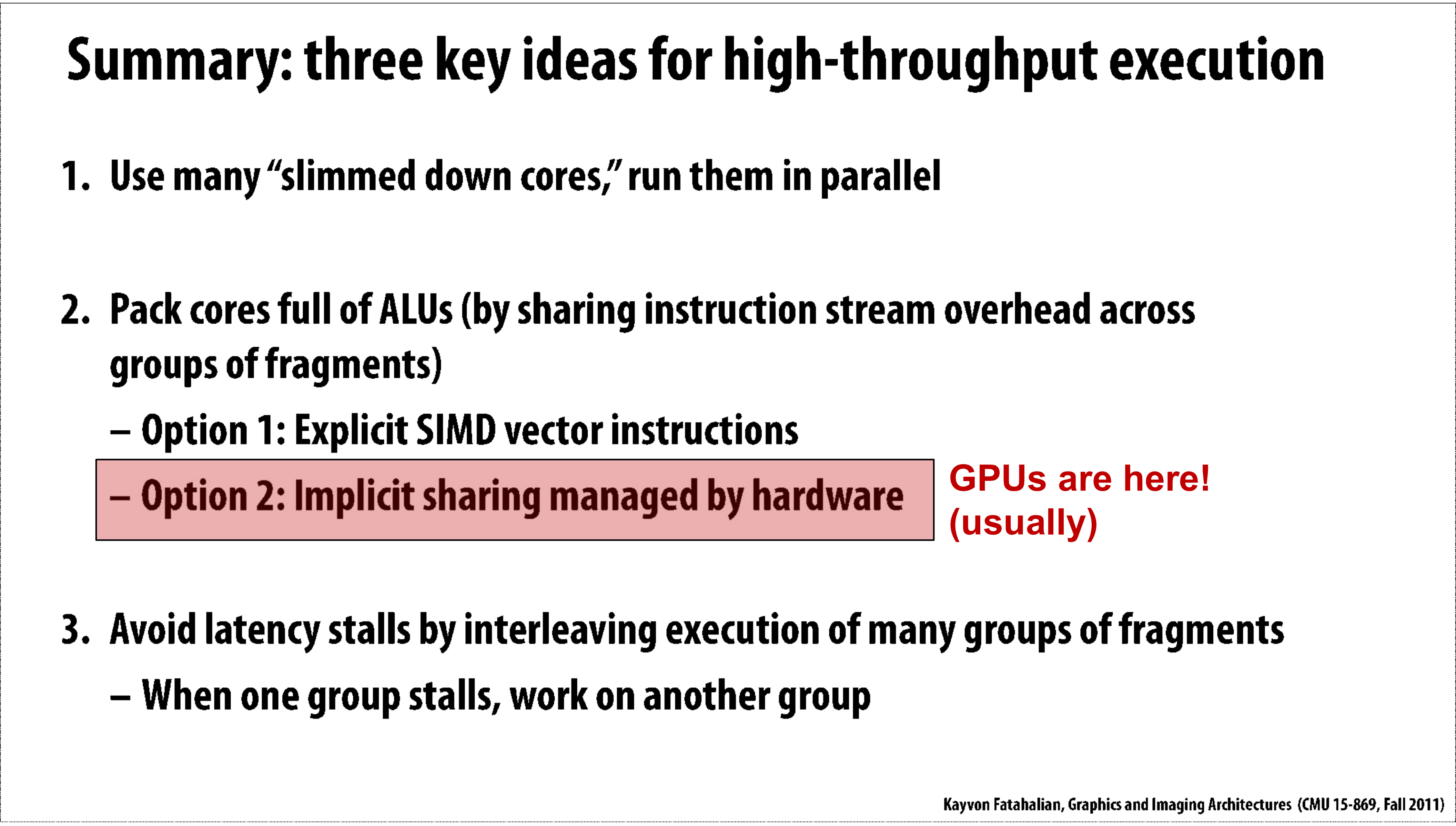
Reading Assignment #3 (until Sep 16)

Read (required):

- Programming Massively Parallel Processors book, 4th edition, **Chapter 4** (*Compute architecture and scheduling*)
- NVIDIA CUDA C++ Programming Guide (current: v12.6, Aug 29, 2024):
  Read ***Chapter 5.6*** (Compute Capability);
  Read ***Chapter 19.1*** (Compute Capabilities);
  Browse all of ***Chapter 19*** (Compute Capabilities)
  Browse all of ***Chapter 8.2*** (Maximize Utilization) and
  Chapter 8.4 (Maximize Instruction Throughput)

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

Where this is going...



Summary: three key ideas for high-throughput execution

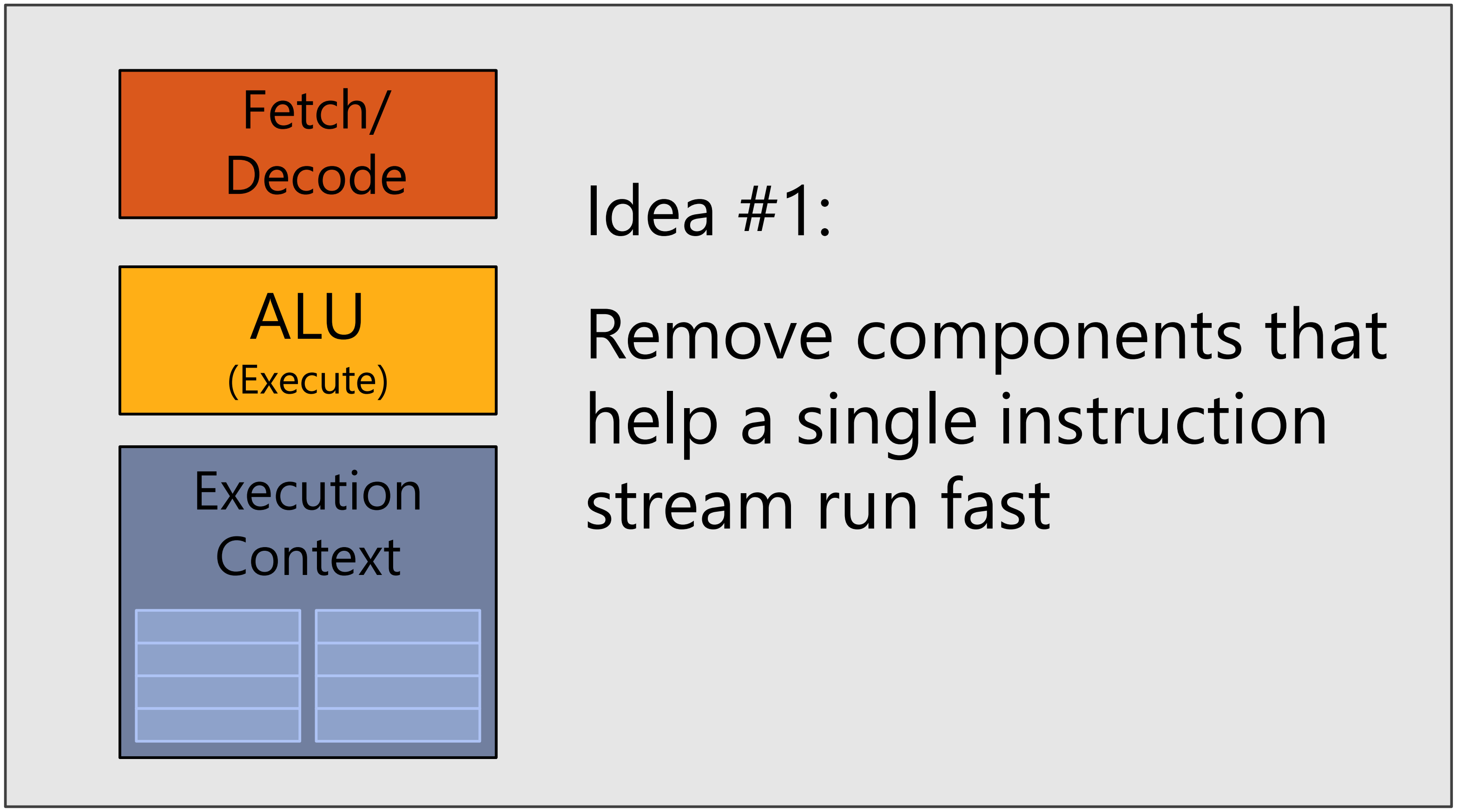
1. Use many "slimmed down cores," run them in parallel

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

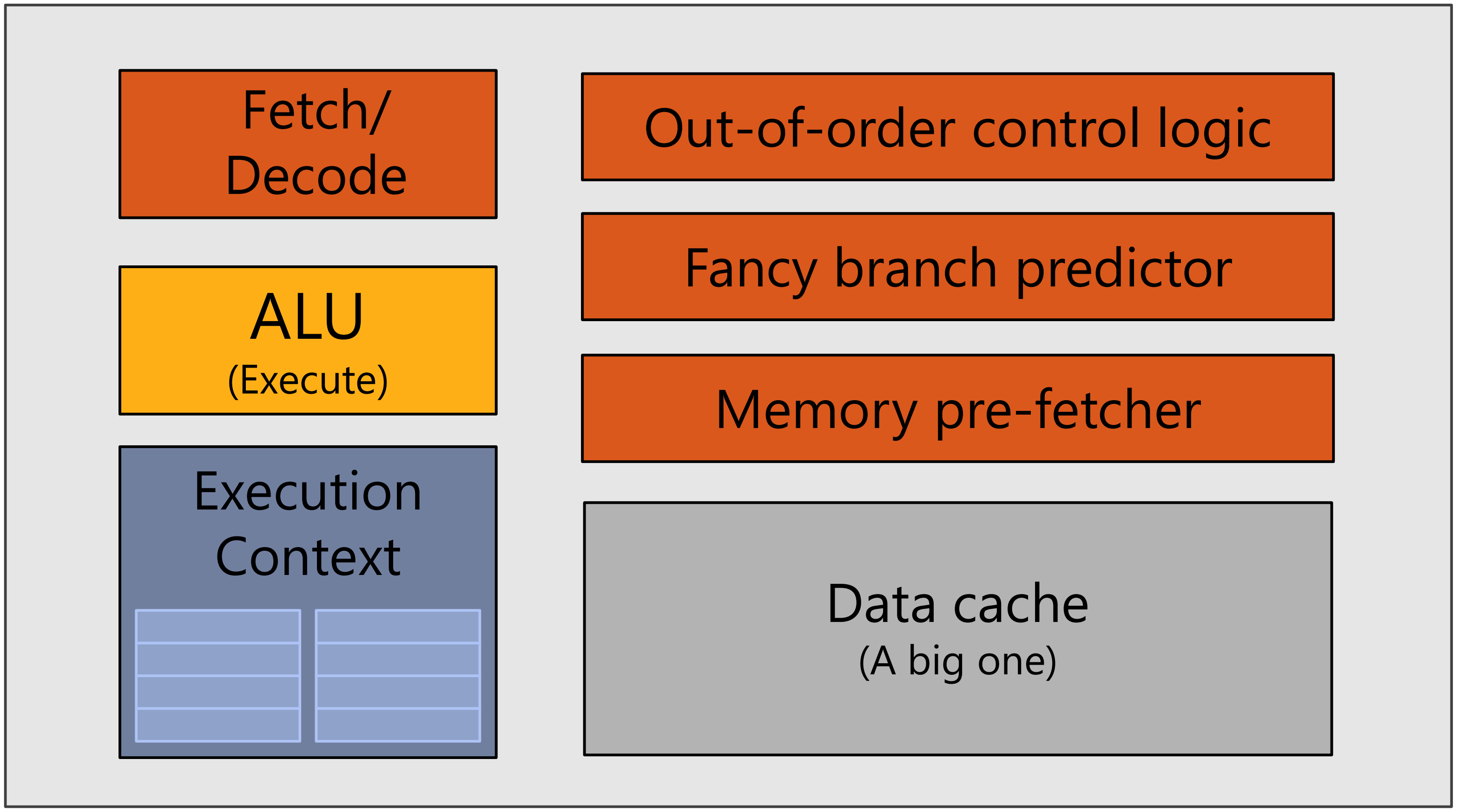
Idea #1: Slim down



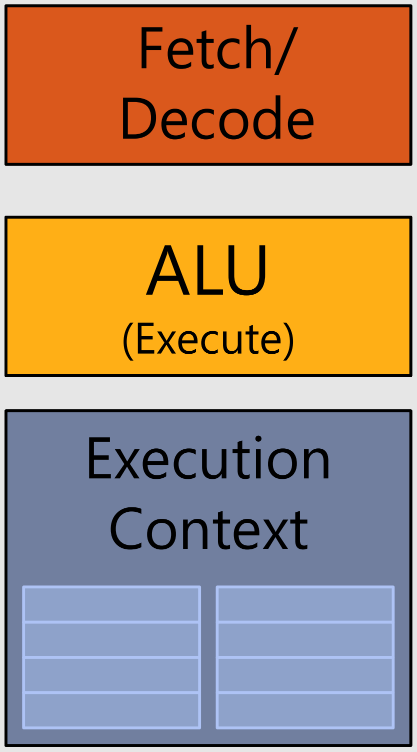
Idea #1:

Remove components that help a single instruction stream run fast

CPU-"style" cores



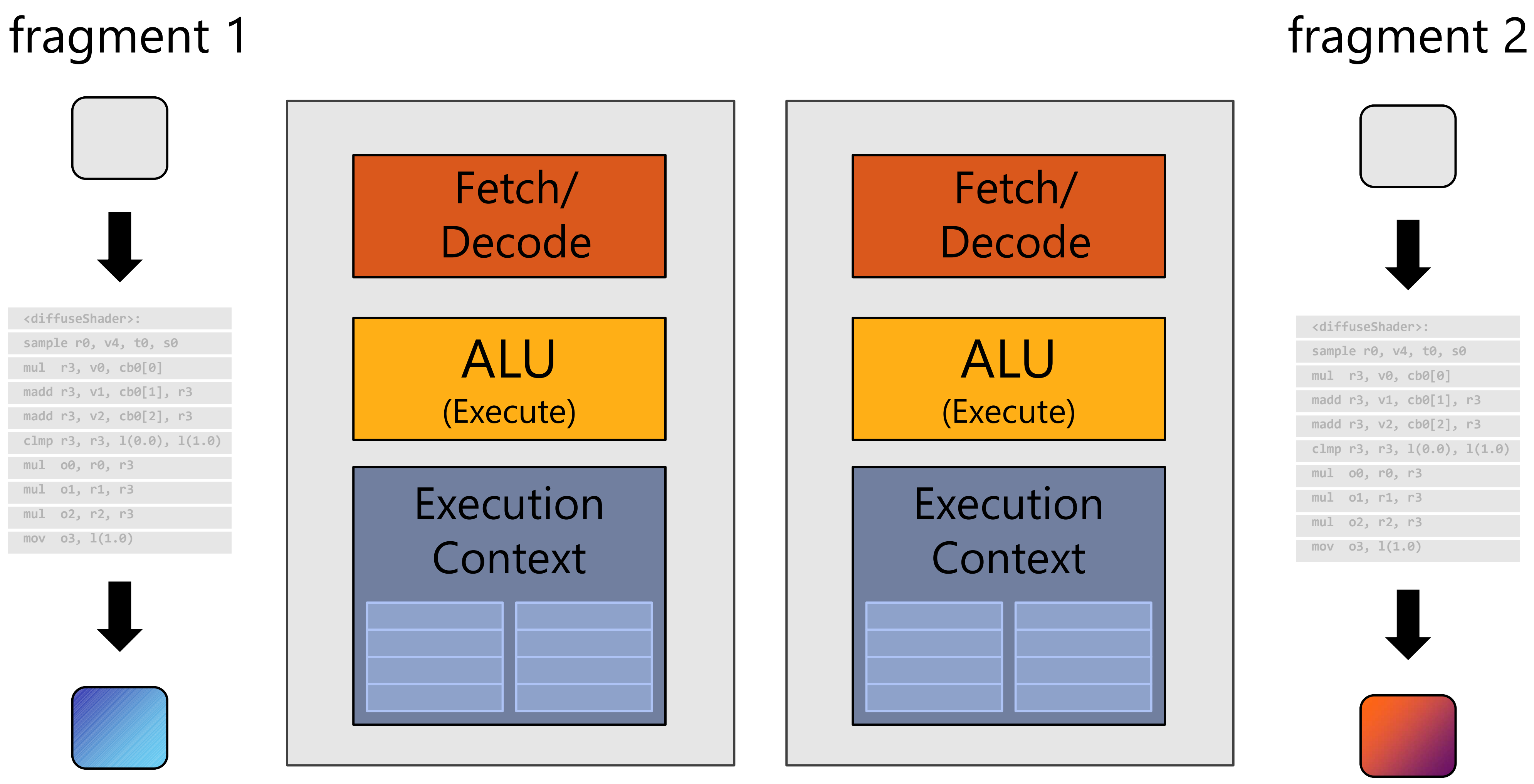
Idea #1: Slim down



Idea #1:

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

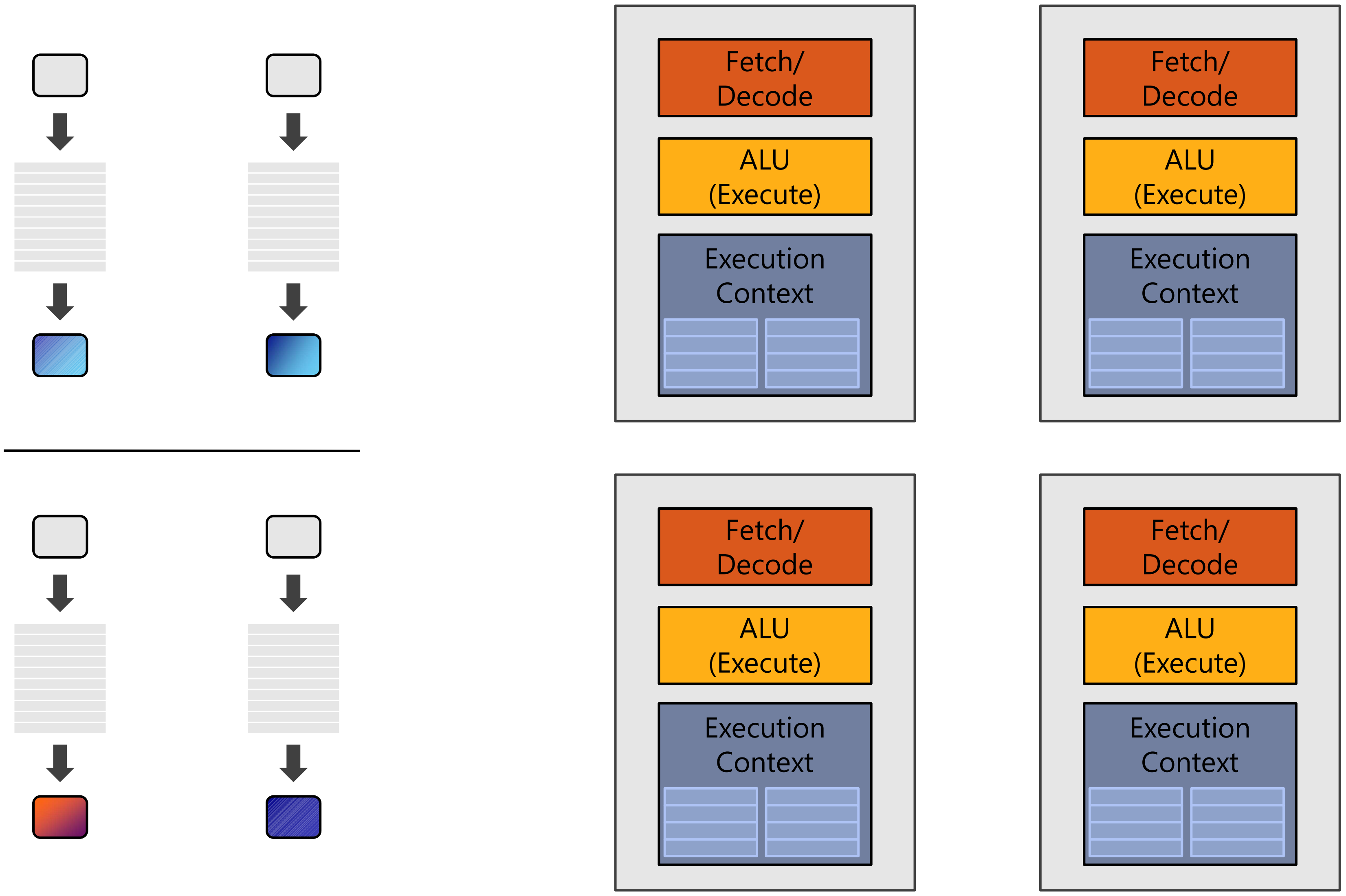
Two cores (two fragments in parallel)



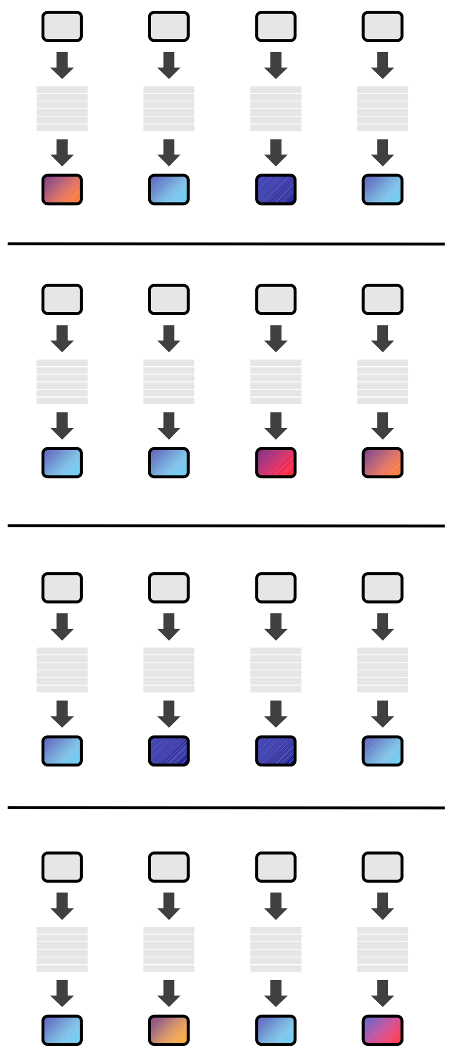
```
<diffuseShader>:
sample r0, v4, t0, s0
mul  r3, v0, cb0[0]
madd r3, v1, cb0[1], r3
madd r3, v2, cb0[2], r3
clmp r3, r3, l(0.0), l(1.0)
mul  o0, r0, r3
mul  o1, r1, r3
mul  o2, r2, r3
mov  o3, l(1.0)
```

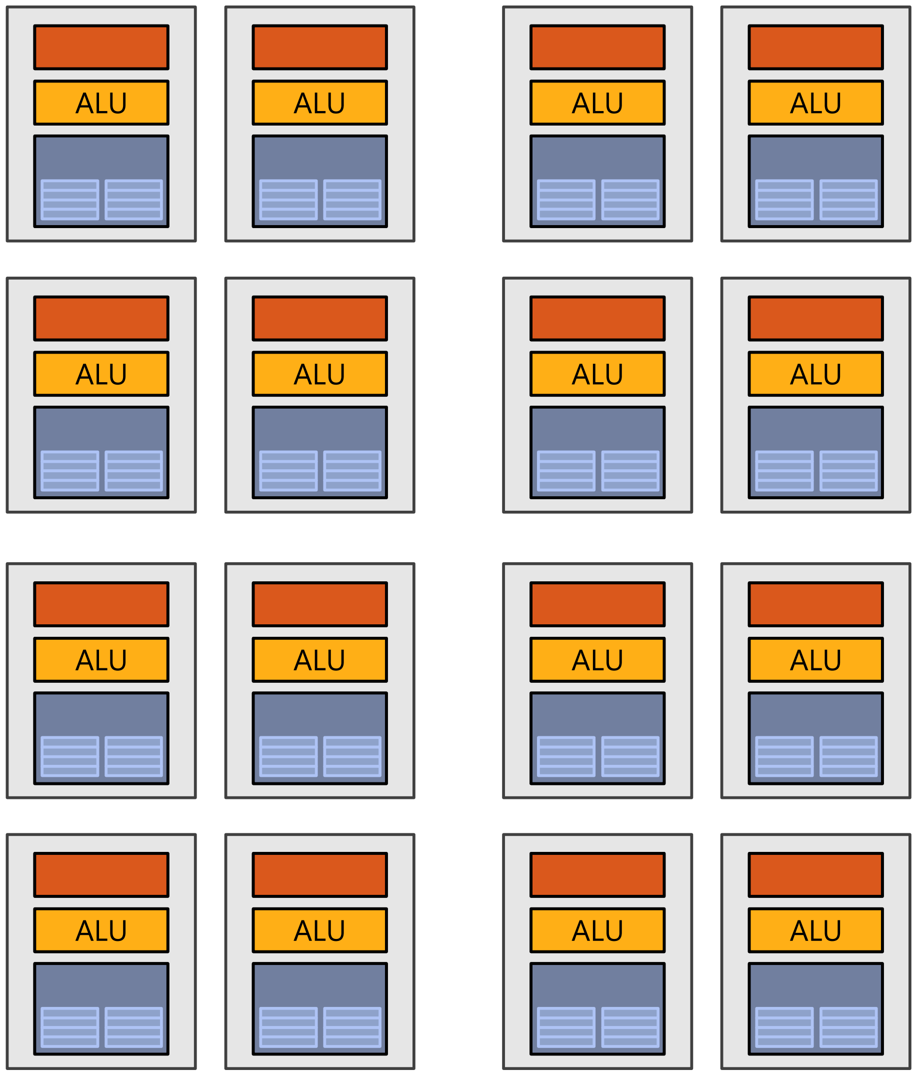
```
<diffuseShader>:
sample r0, v4, t0, s0
mul  r3, v0, cb0[0]
madd r3, v1, cb0[1], r3
madd r3, v2, cb0[2], r3
clmp r3, r3, l(0.0), l(1.0)
mul  o0, r0, r3
mul  o1, r1, r3
mul  o2, r2, r3
mov  o3, l(1.0)
```

Four cores (four fragments in parallel)



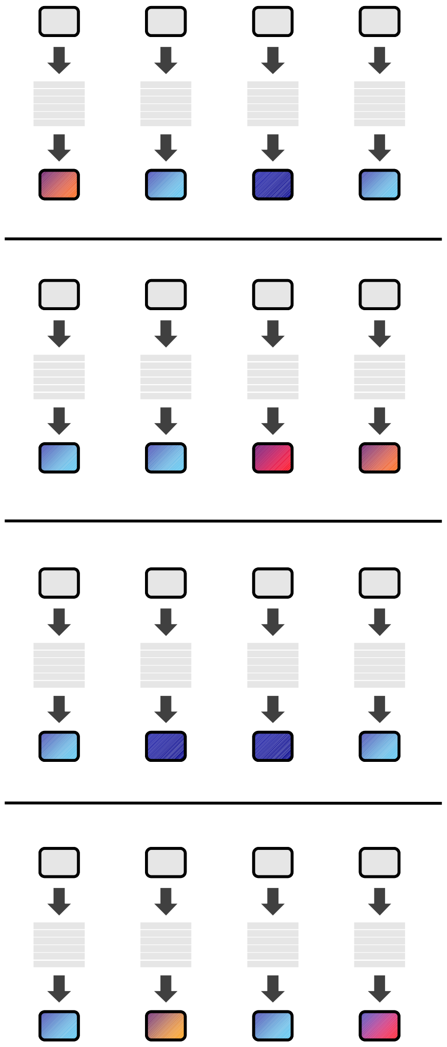
Sixteen cores (sixteen fragments in parallel)





16 cores = 16 simultaneous instruction streams

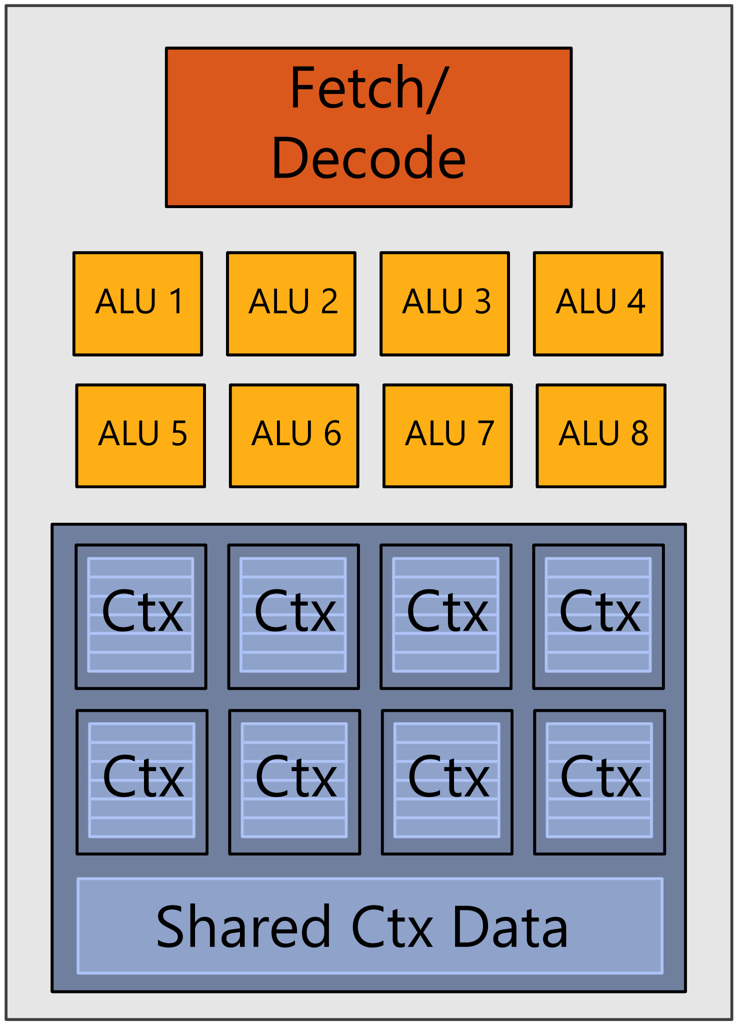
Instruction stream sharing



But… many fragments should be able to share an instruction stream! → **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)
```

Idea #2: Add ALUs



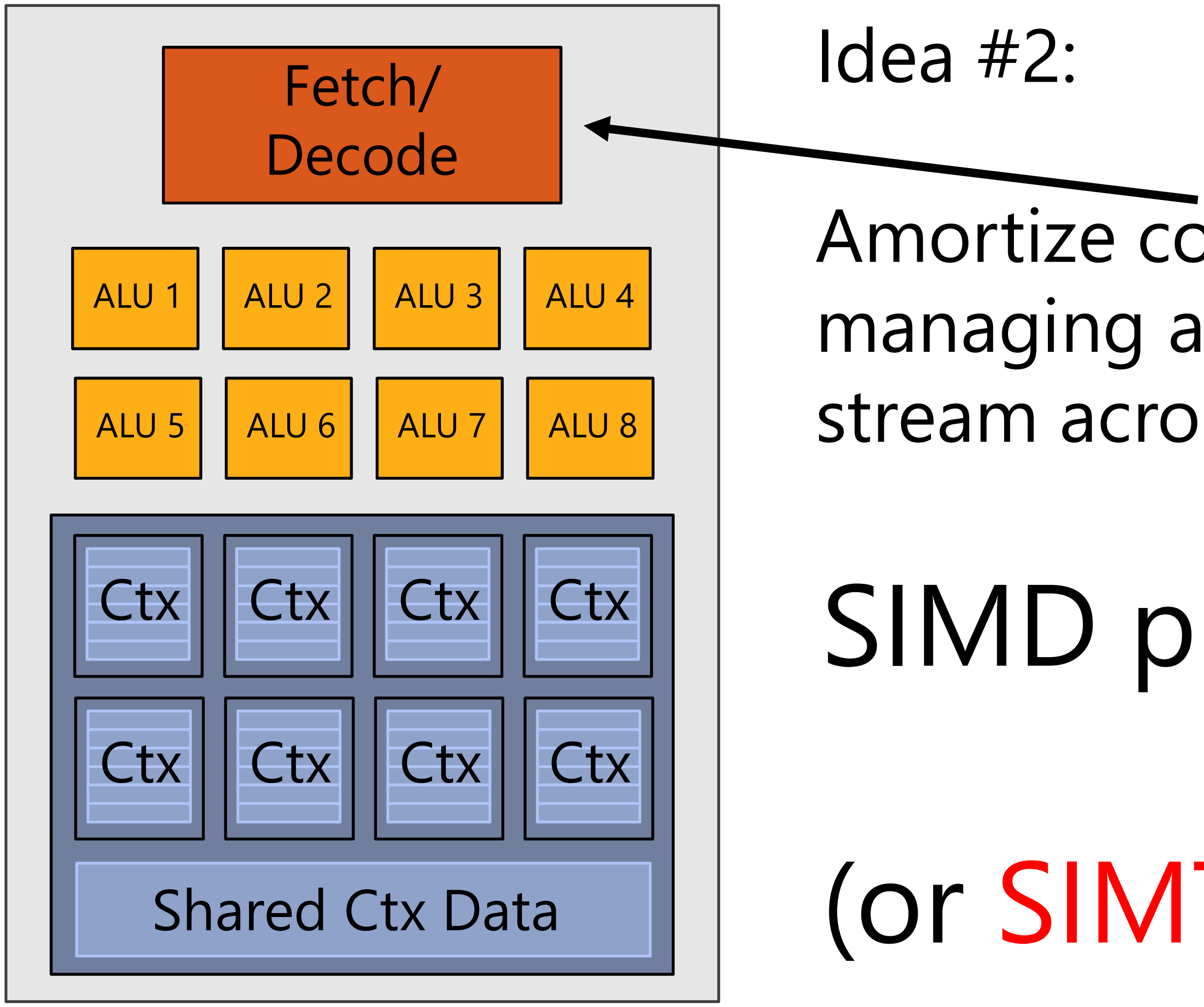
Idea #2:

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

SIMD processing

(or SIMT, SPMD)

Idea #2: Add ALUs



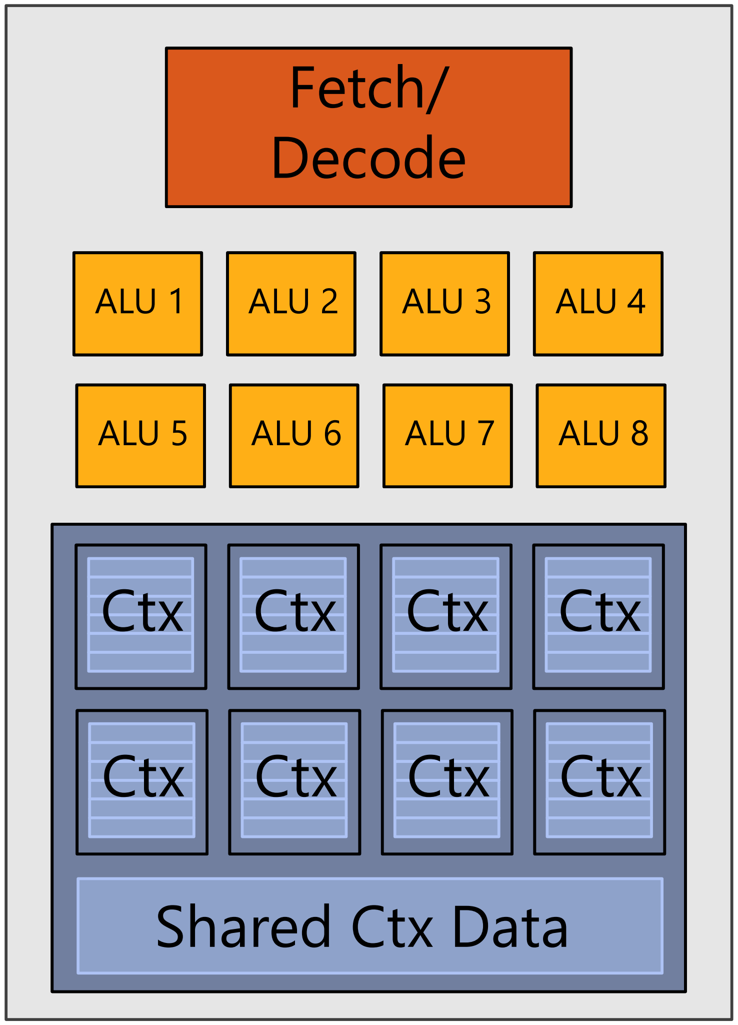
Idea #2:

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

SIMD processing

(or SIMT, SPMD)

How does shader execution behave?

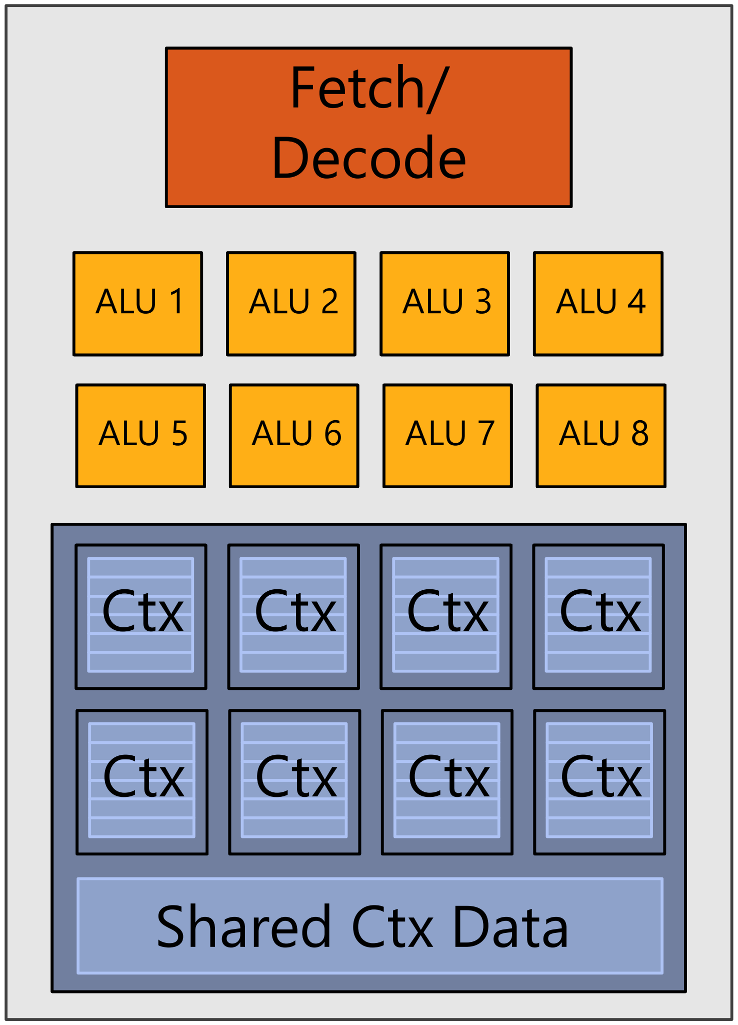


```
<diffuseShader>:
sample r0, v4, t0, s0
mul  r3, v0, cb0[0]
madd r3, v1, cb0[1], r3
madd r3, v2, cb0[2], r3
clmp r3, r3, l(0.0), l(1.0)
mul  o0, r0, r3
mul  o1, r1, r3
mul  o2, r2, r3
mov  o3, l(1.0)
```

Original compiled shader:

Processes one fragment
using scalar ops on scalar registers

How does shader execution behave?

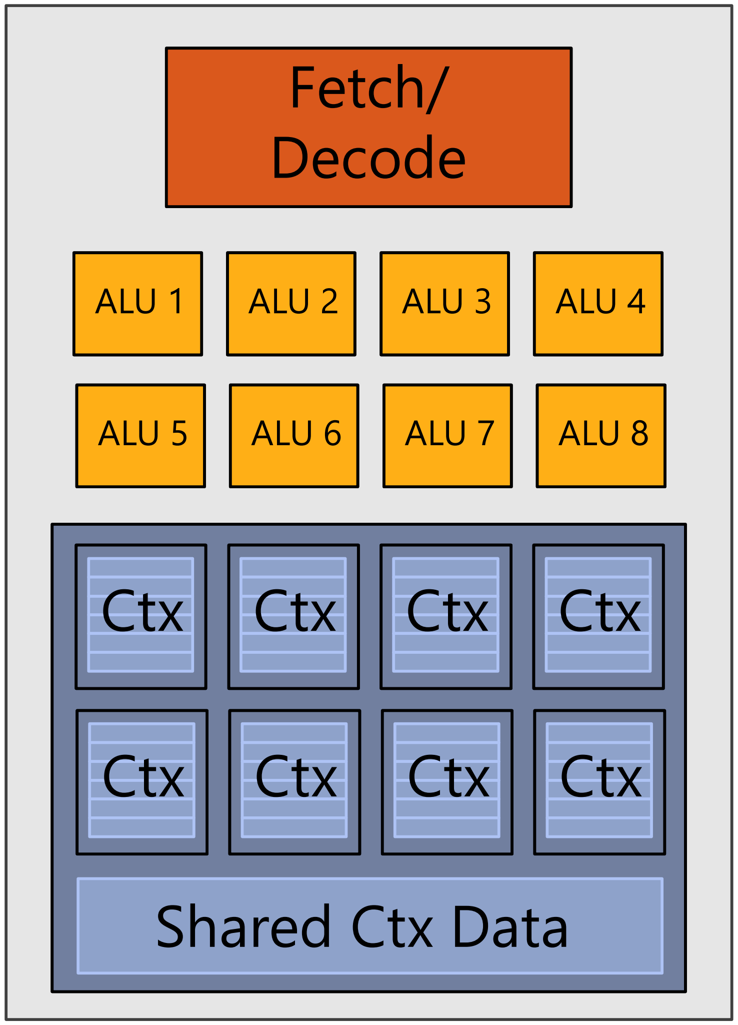


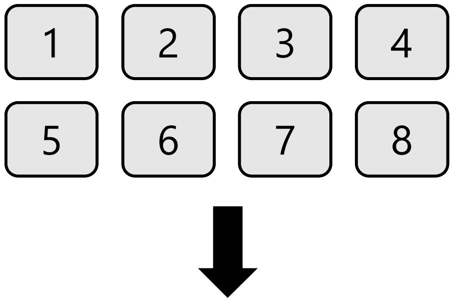
```
<VEC8_diffuseShader>:
VEC8_sample vec_r0, vec_v4, t0, vec_s0
VEC8_mul  vec_r3, vec_v0, cb0[0]
VEC8_madd vec_r3, vec_v1, cb0[1], vec_r3
VEC8_madd vec_r3, vec_v2, cb0[2], vec_r3
VEC8_clmp vec_r3, vec_r3, l(0.0), l(1.0)
VEC8_mul  vec_o0, vec_r0, vec_r3
VEC8_mul  vec_o1, vec_r1, vec_r3
VEC8_mul  vec_o2,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

Actually executed shader:

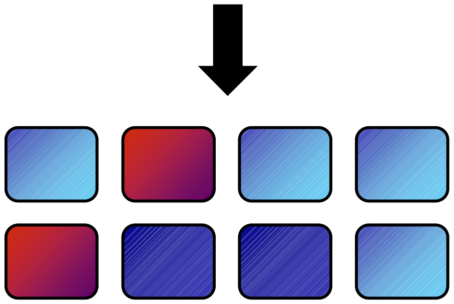
Processes 8 fragments
using "vector ops" on "vector registers"
(Caveat: This does NOT mean there are actual vector instructions/cores/regs! See later slide.)

How does shader execution behave?

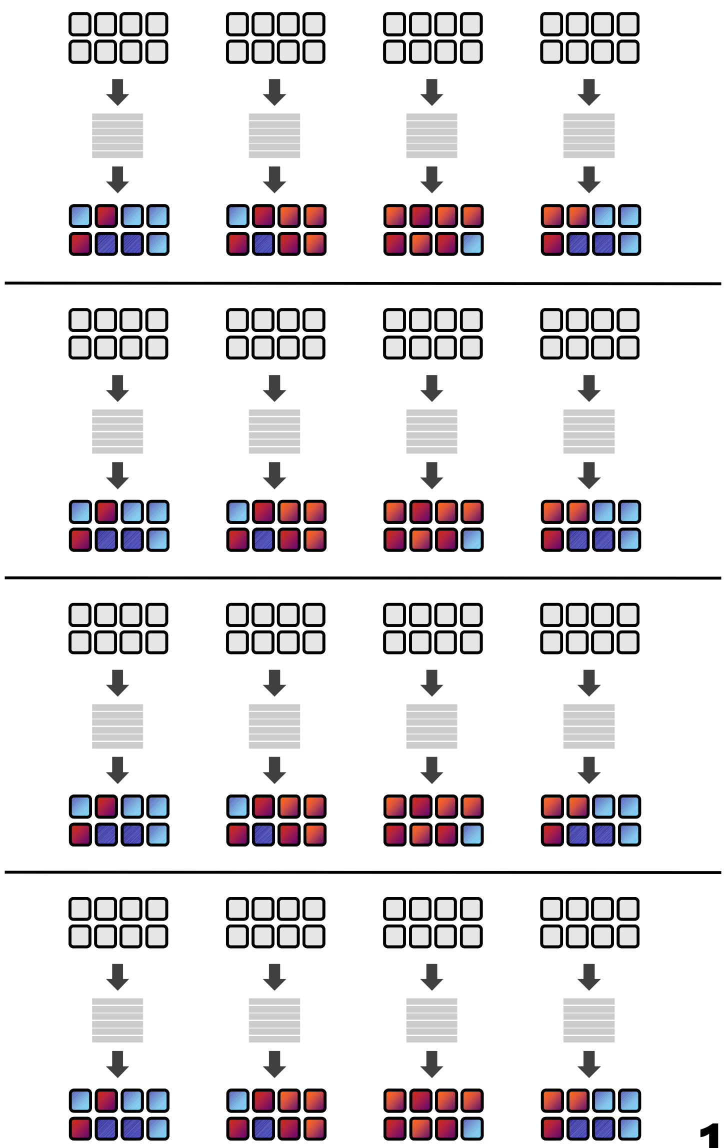


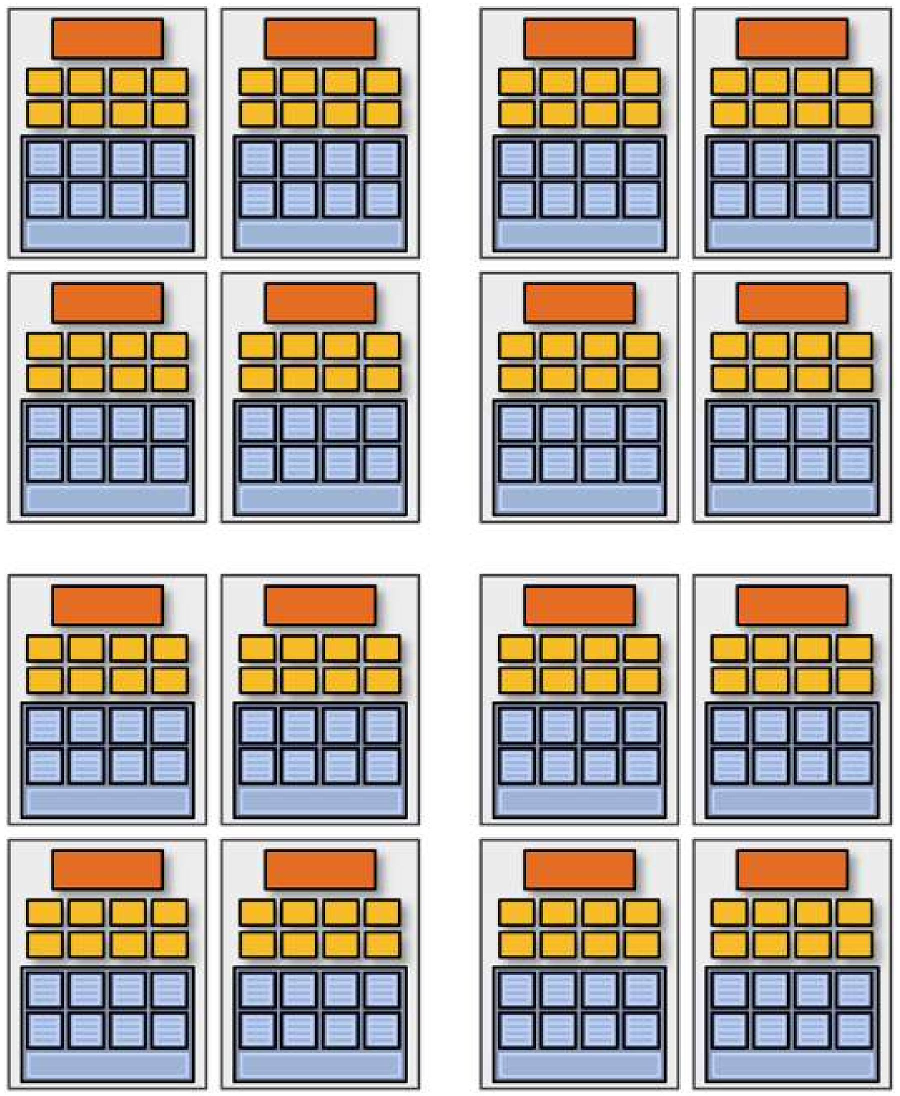


```
<VEC8_diffuseShader>:
VEC8_sample vec_r0, vec_v4, t0, vec_s0
VEC8_mul  vec_r3, vec_v0, cb0[0]
VEC8_madd vec_r3, vec_v1, cb0[1], vec_r3
VEC8_madd vec_r3, vec_v2, cb0[2], vec_r3
VEC8_clmp vec_r3, vec_r3, l(0.0), l(1.0)
VEC8_mul  vec_o0, vec_r0, vec_r3
VEC8_mul  vec_o1, vec_r1, vec_r3
VEC8_mul  vec_o2,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



128 fragments in parallel

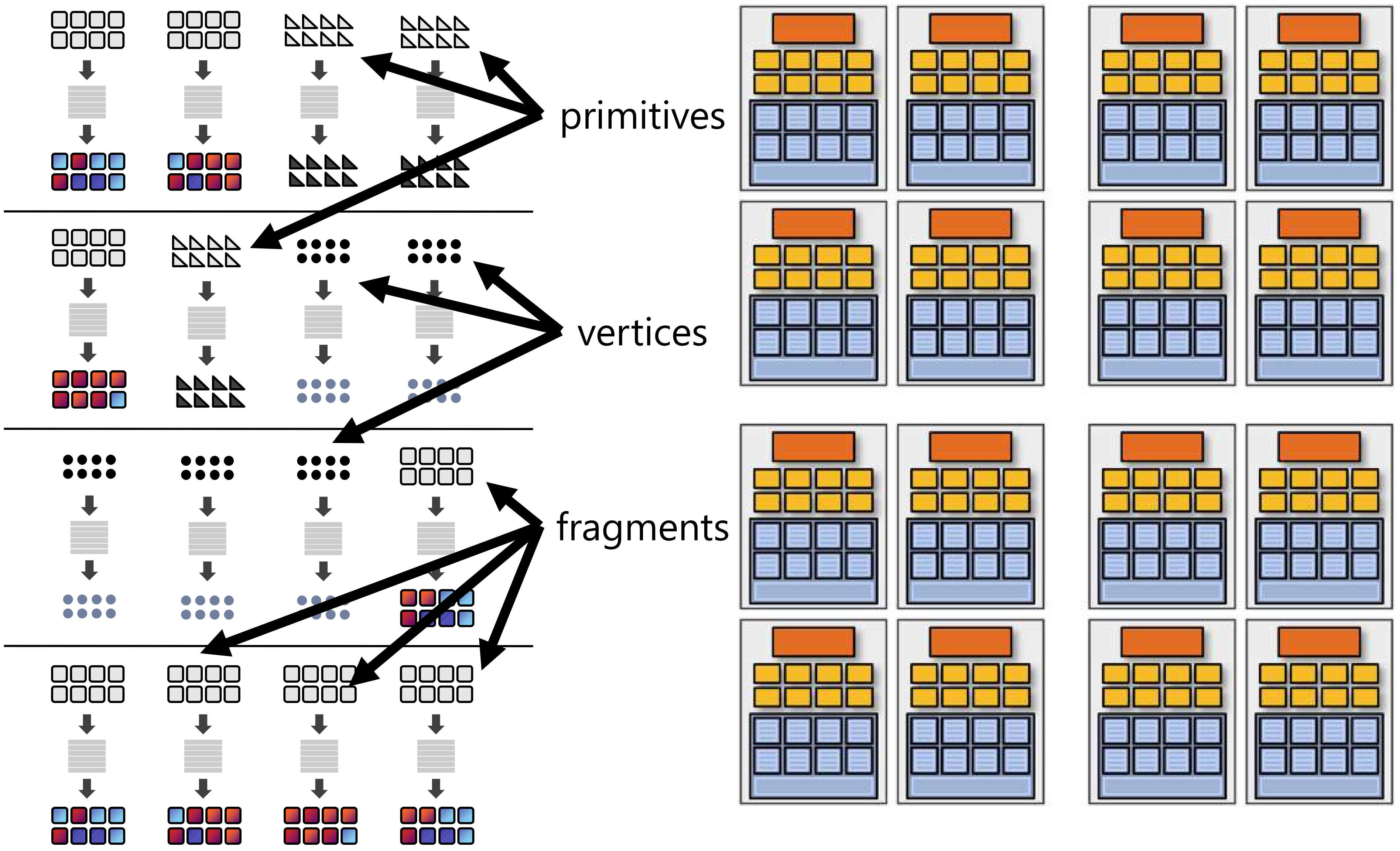




16 cores = **128** ALUs

= **16** simultaneous instruction streams

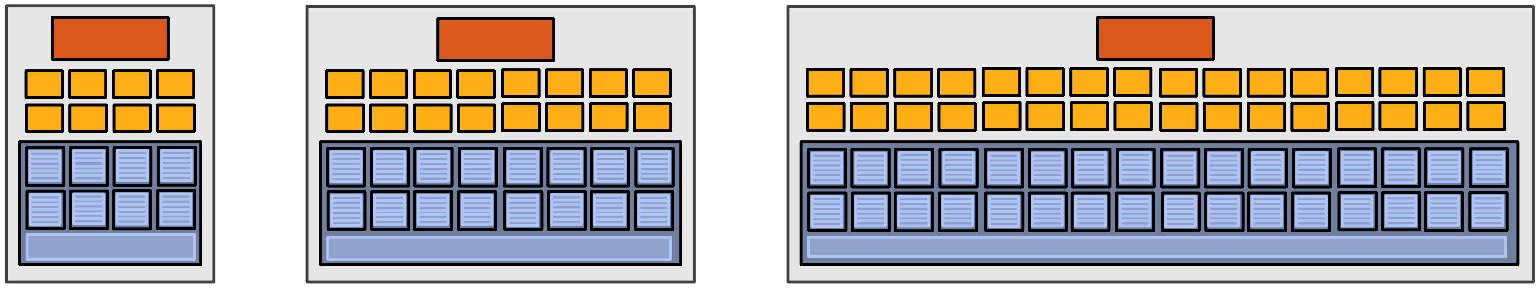
128 [vertices / pixels primitives CUDA threads OpenCL work items compute shader threads] in parallel



Clarification

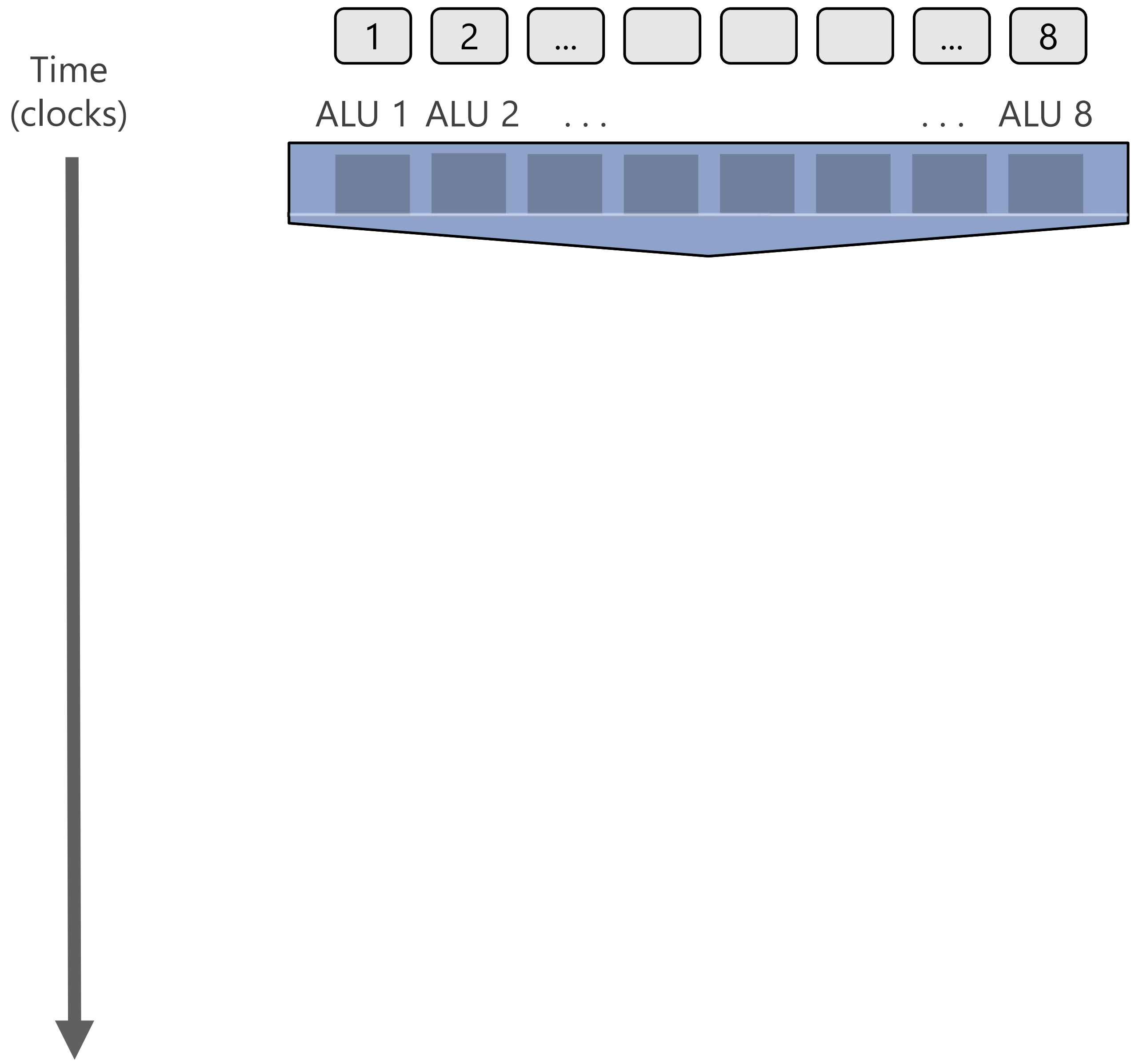
SIMD processing does not imply SIMD instructions

- 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)



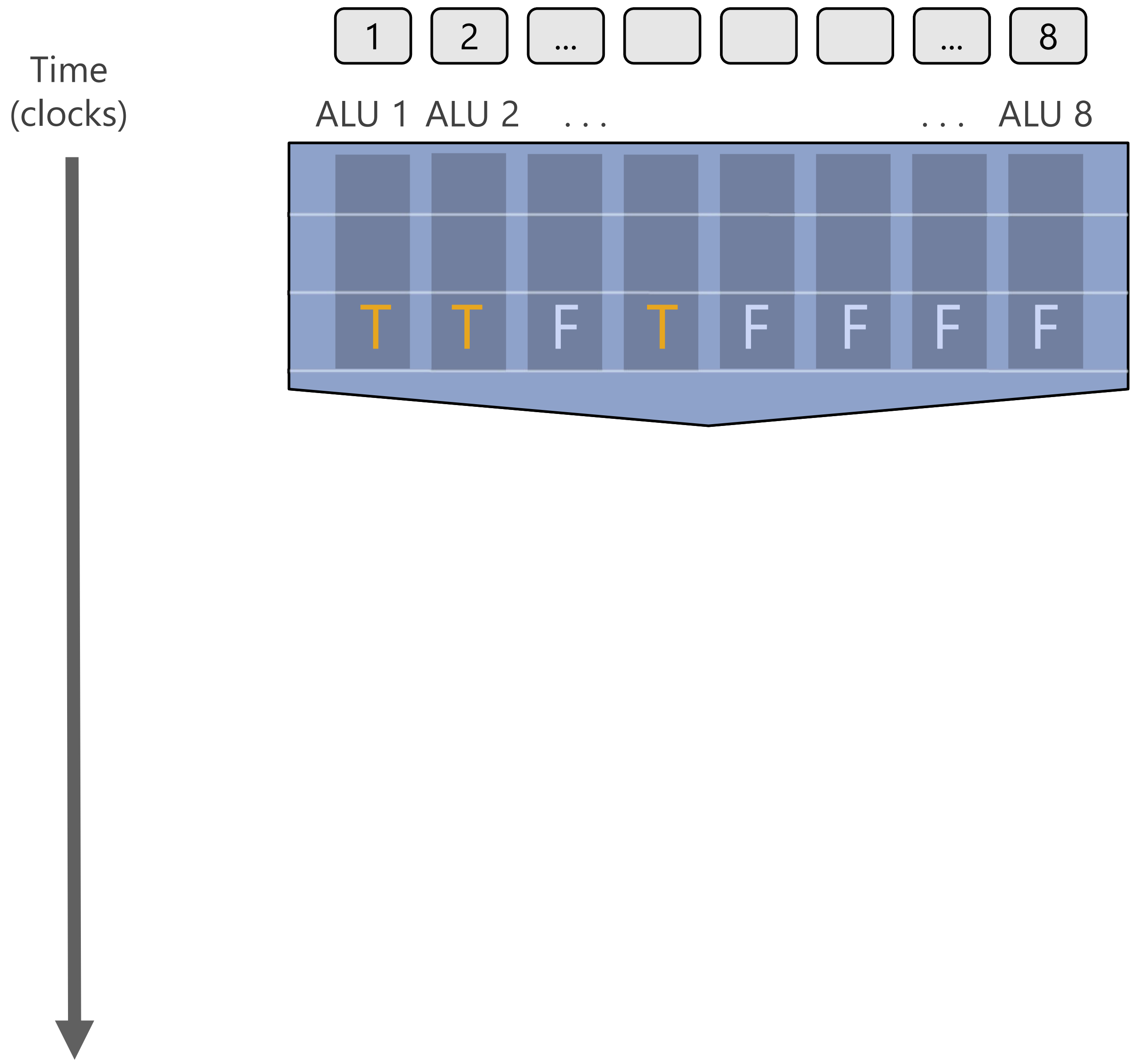
In practice: 16 to 64 fragments share an instruction stream

But what about branches?



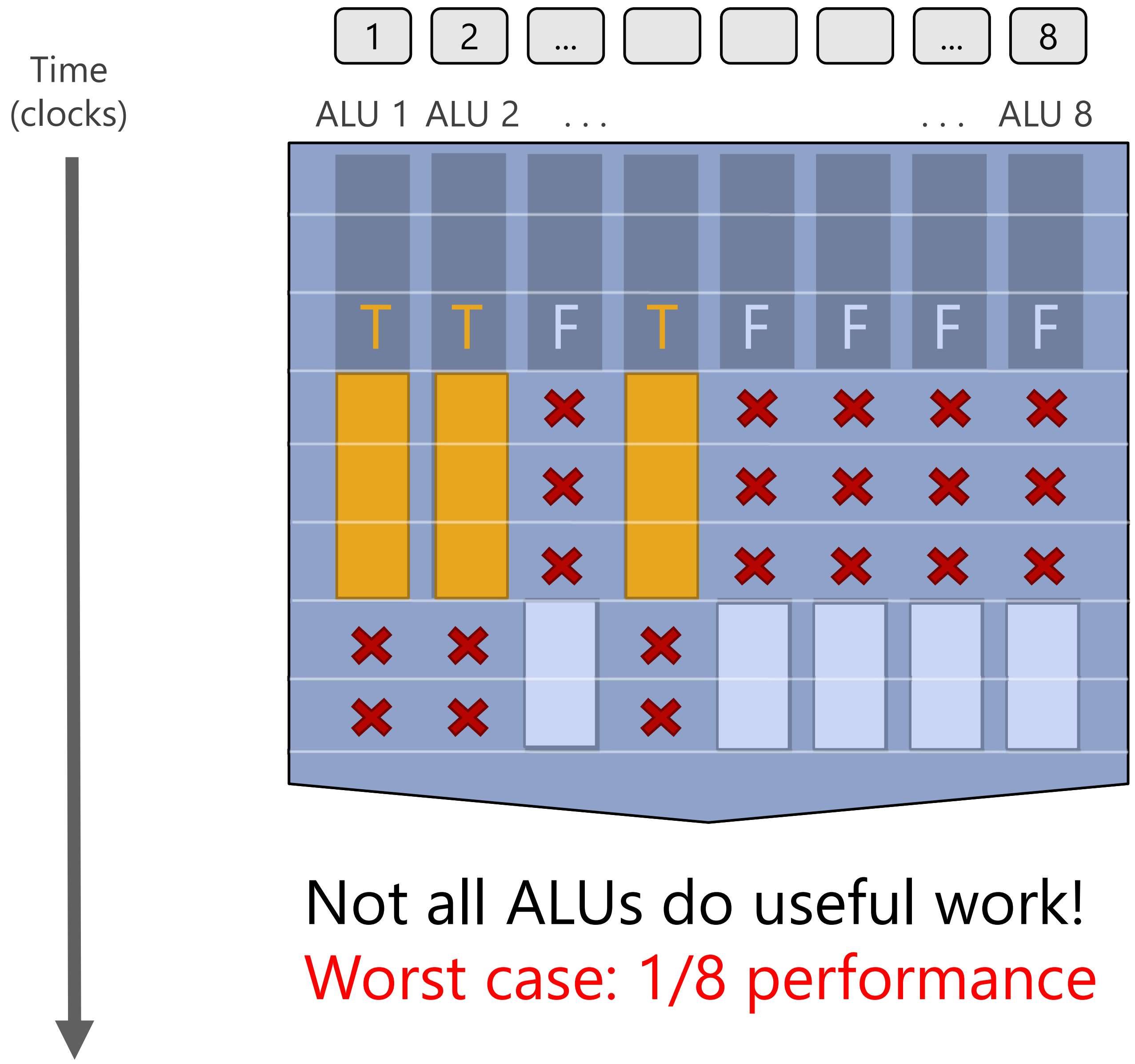
```
<unconditional
shader code>

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?



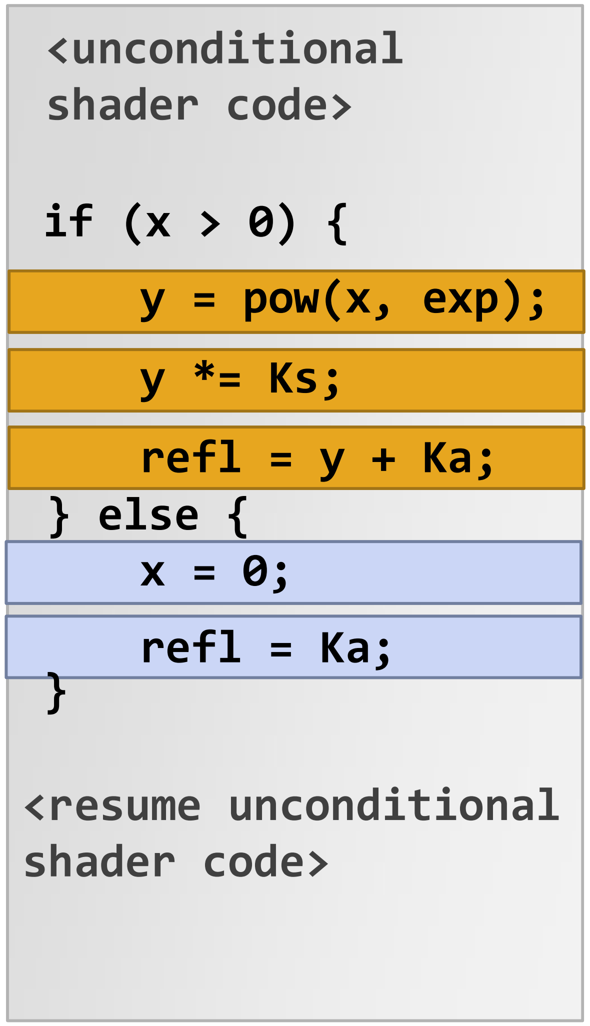
```
<unconditional
shader code>

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?



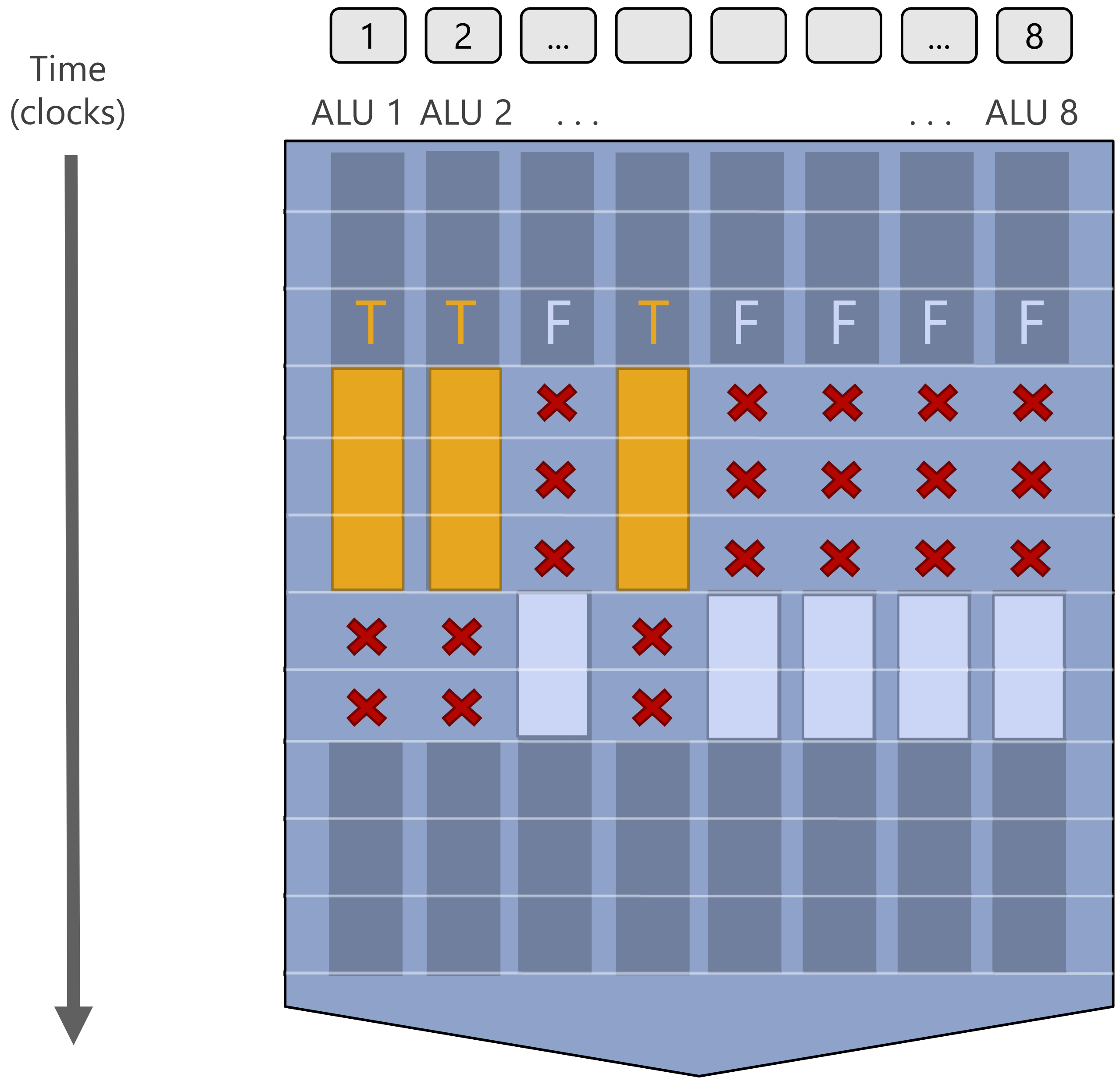
Not all ALUs do useful work!

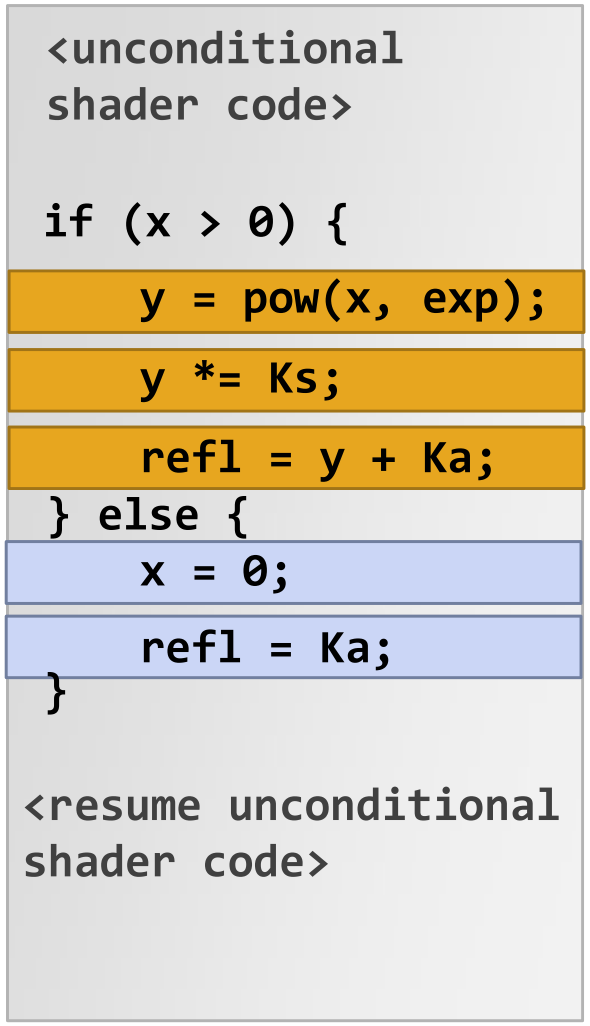
Worst case: 1/8 performance



```
<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>
```

Next Problem: Stalls!

Stalls occur when a core cannot run the next instruction because of a dependency on a previous operation.

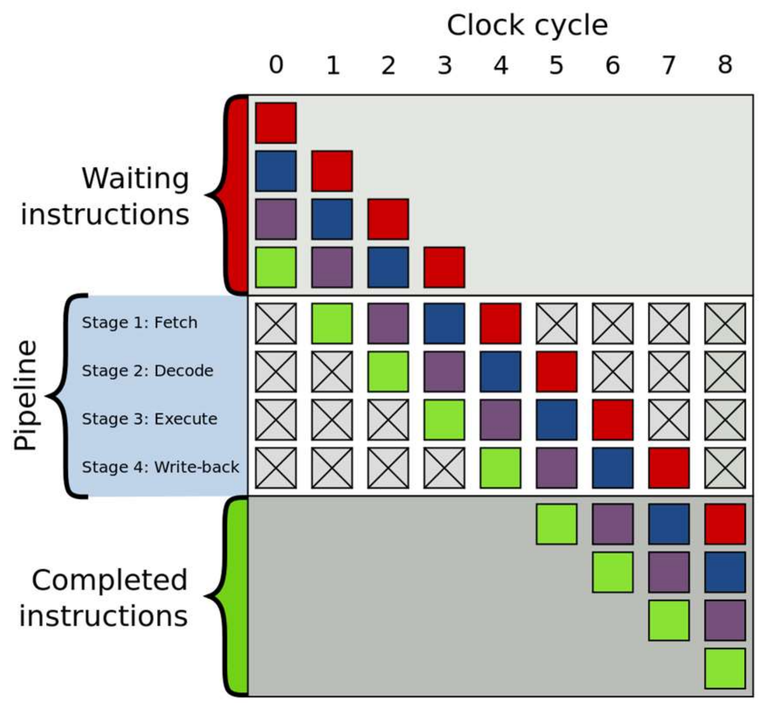
Texture access latency = 100’s to 1000’s of cycles
(also: instruction pipelining hazards, …)

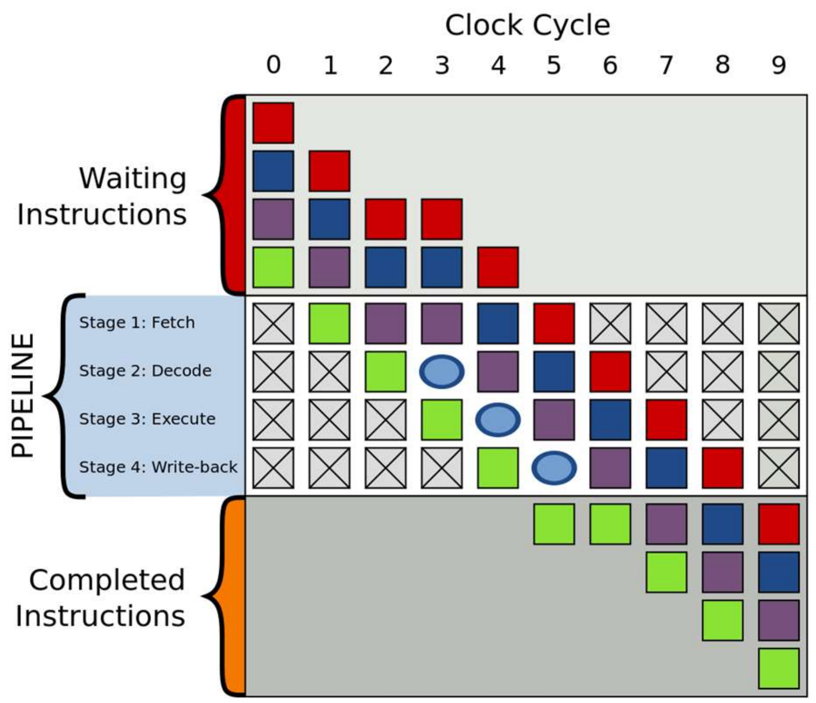
We’ve removed the fancy caches and logic that helps avoid stalls.

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
```

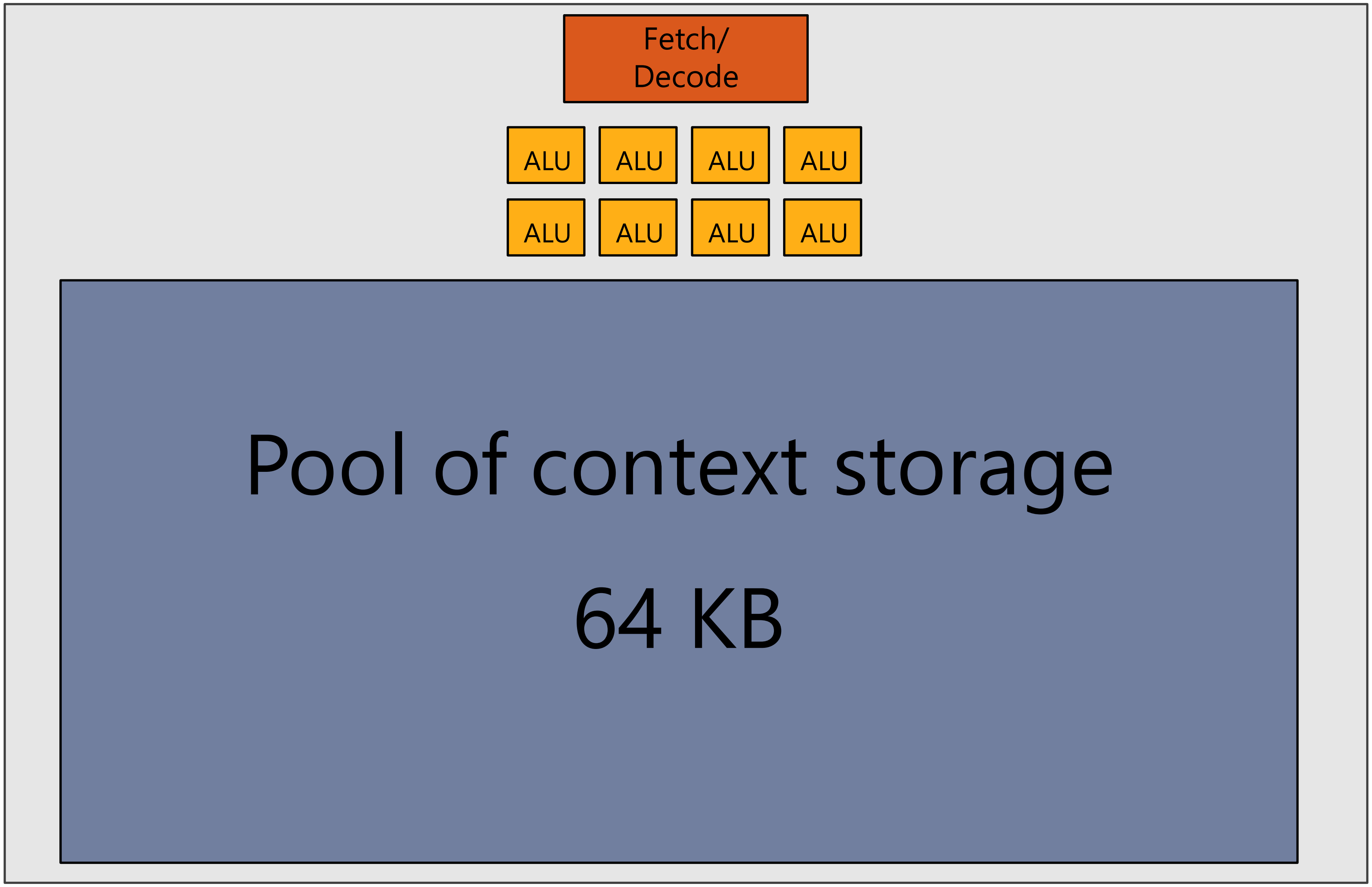
Idea #3: Interleave execution of groups

But we have LOTS of independent fragments.

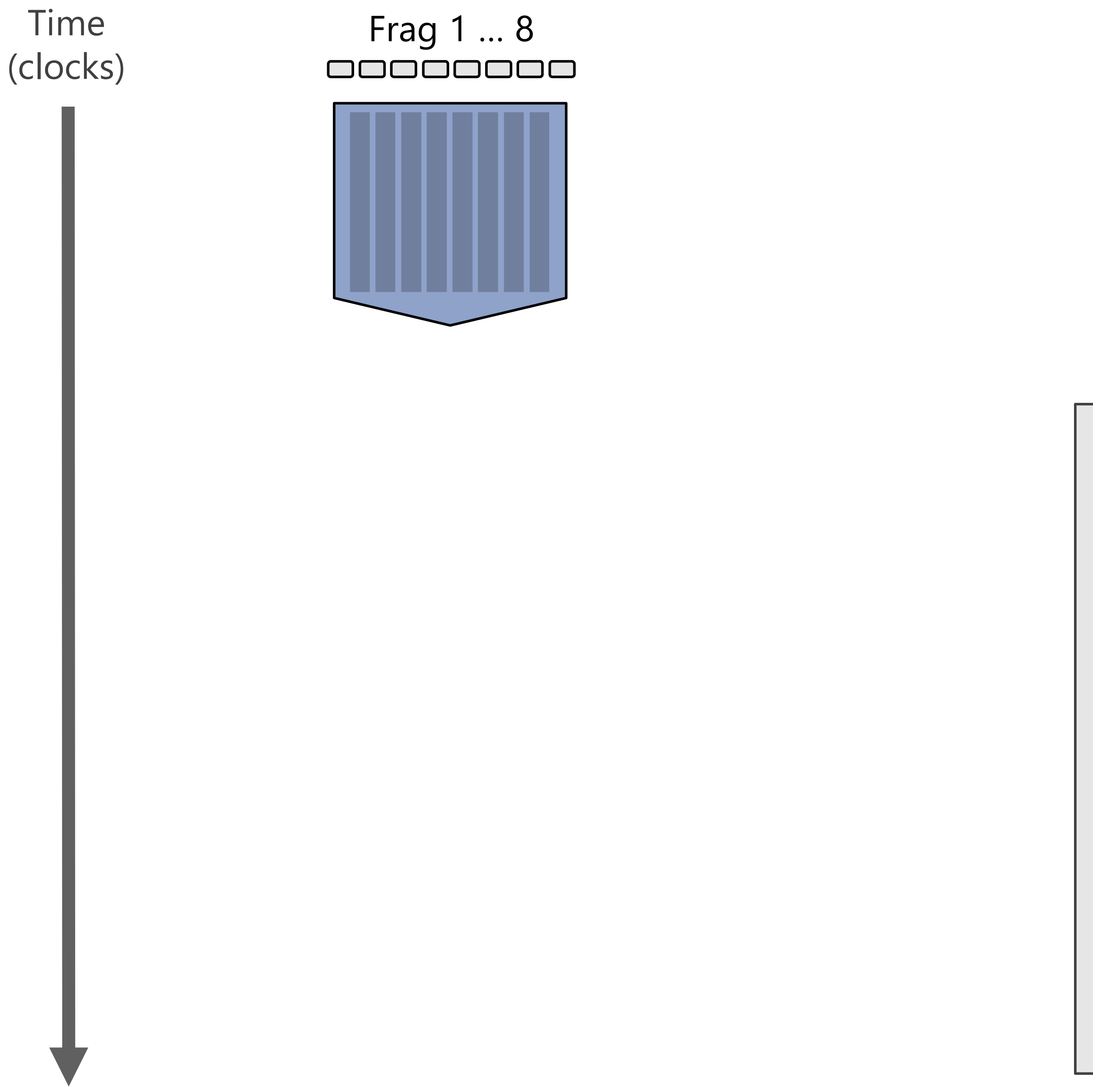
Idea #3:

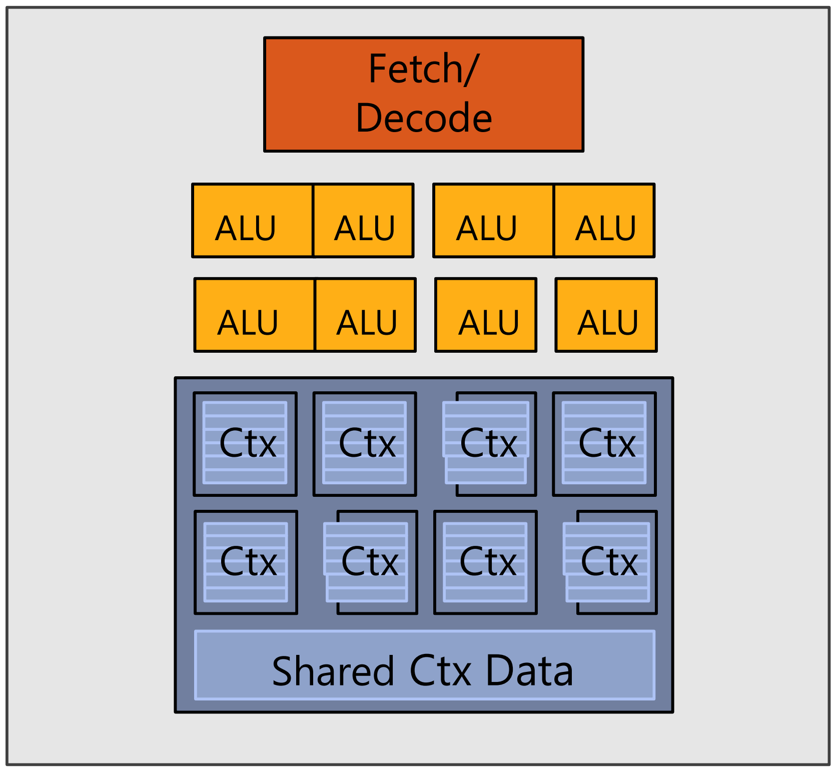
Interleave processing of many fragments on a single core to avoid stalls caused by high latency operations.

Idea #3: Store multiple group contexts

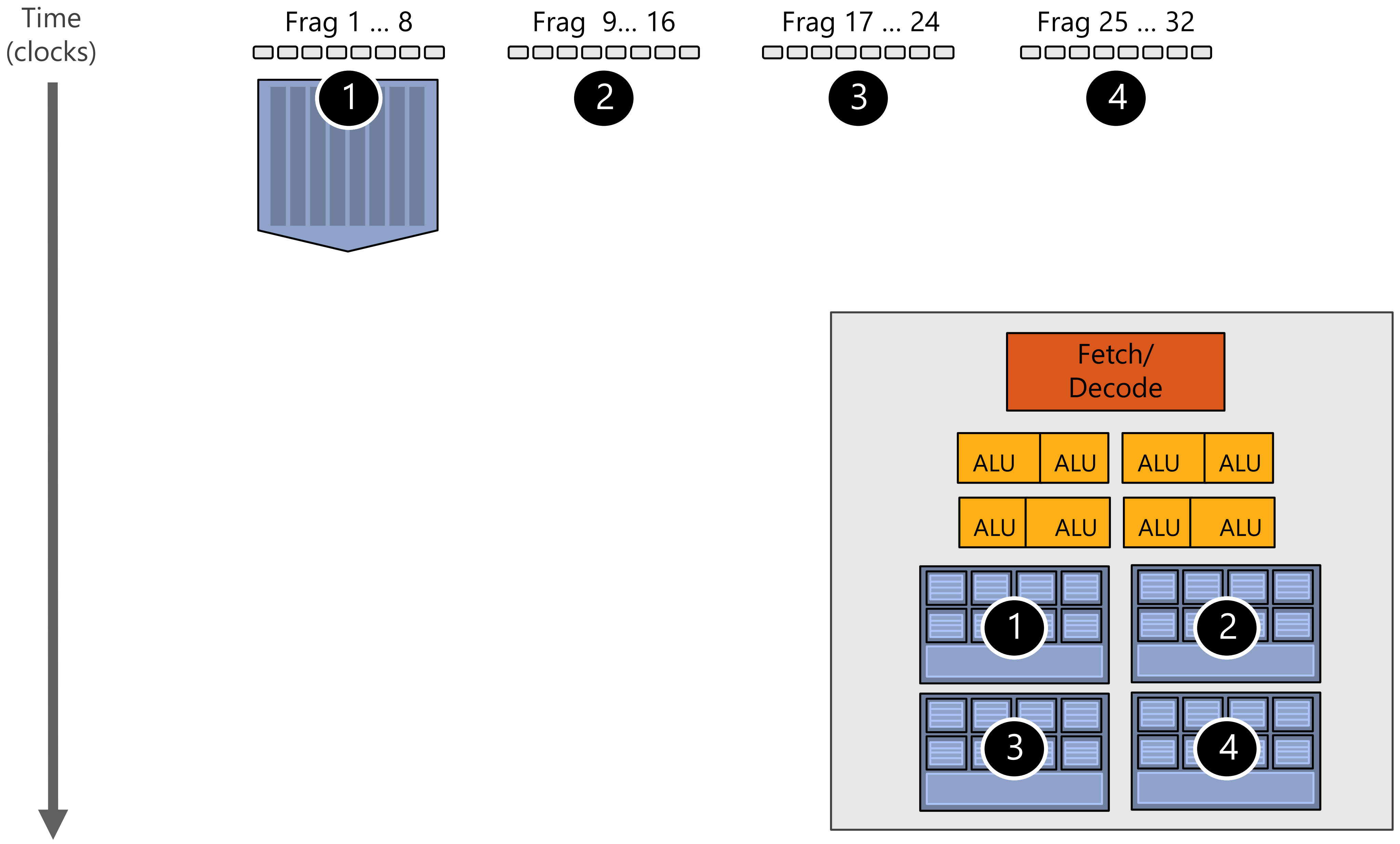


Hiding shader stalls

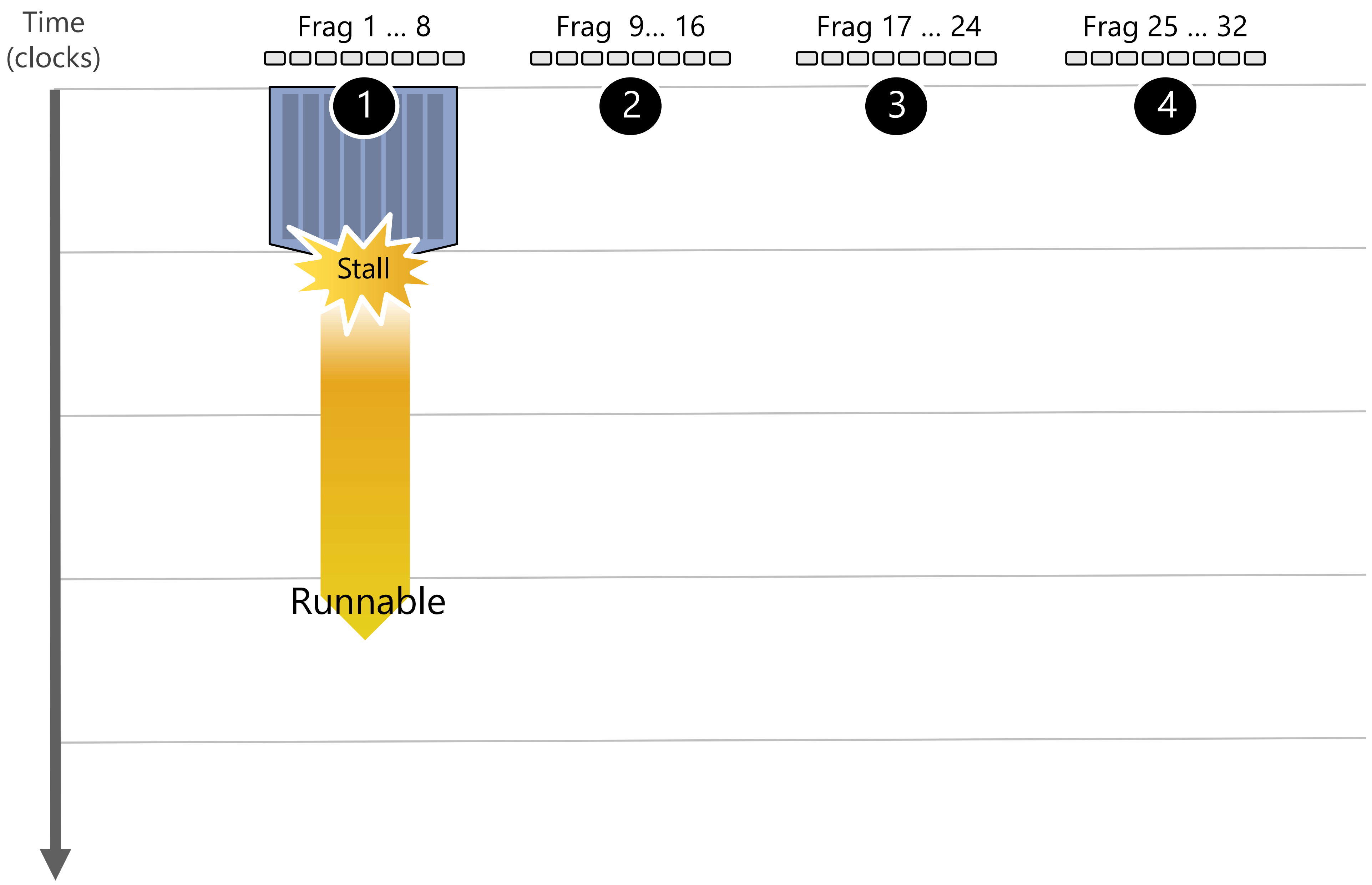




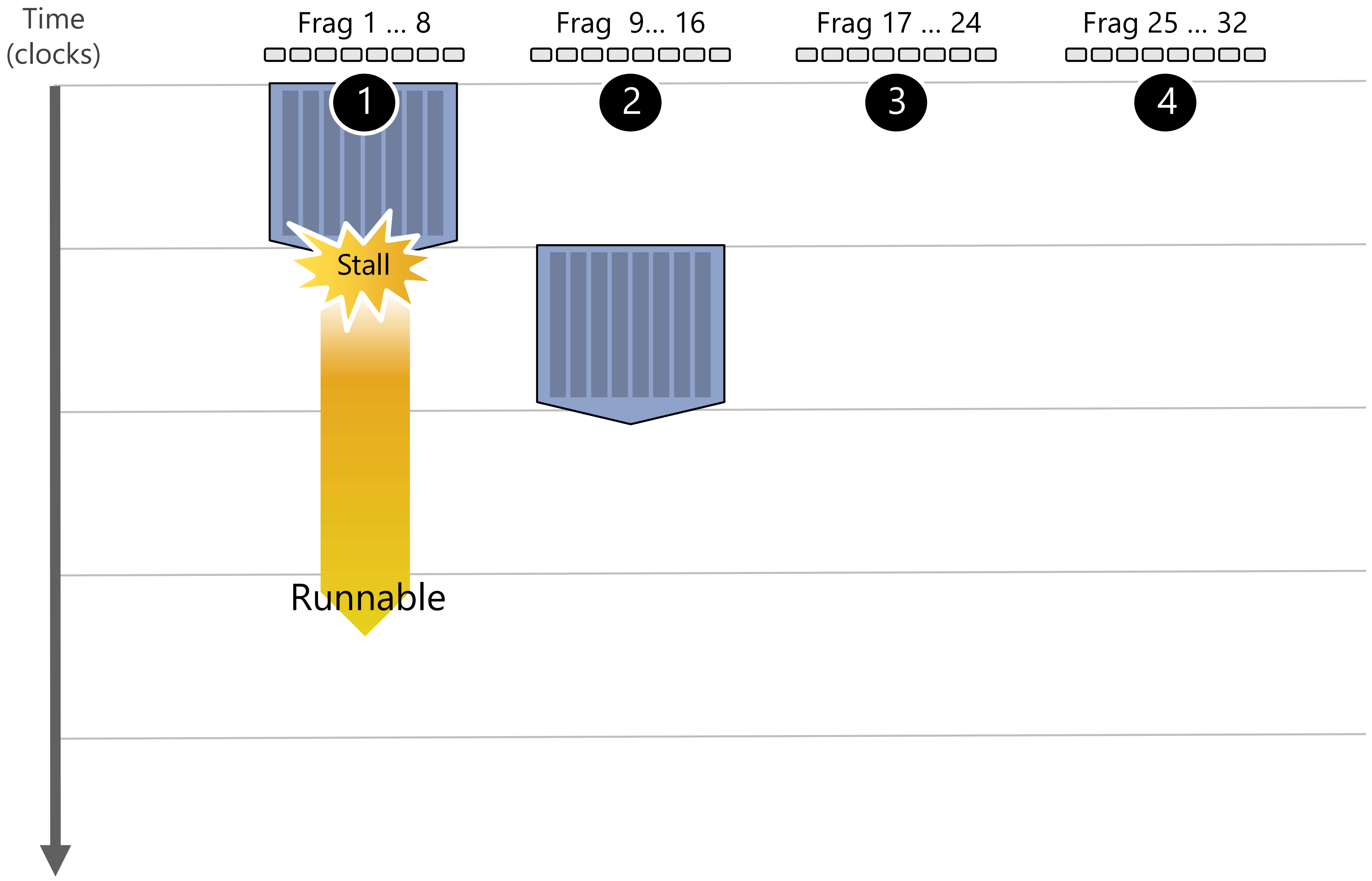
Hiding shader stalls



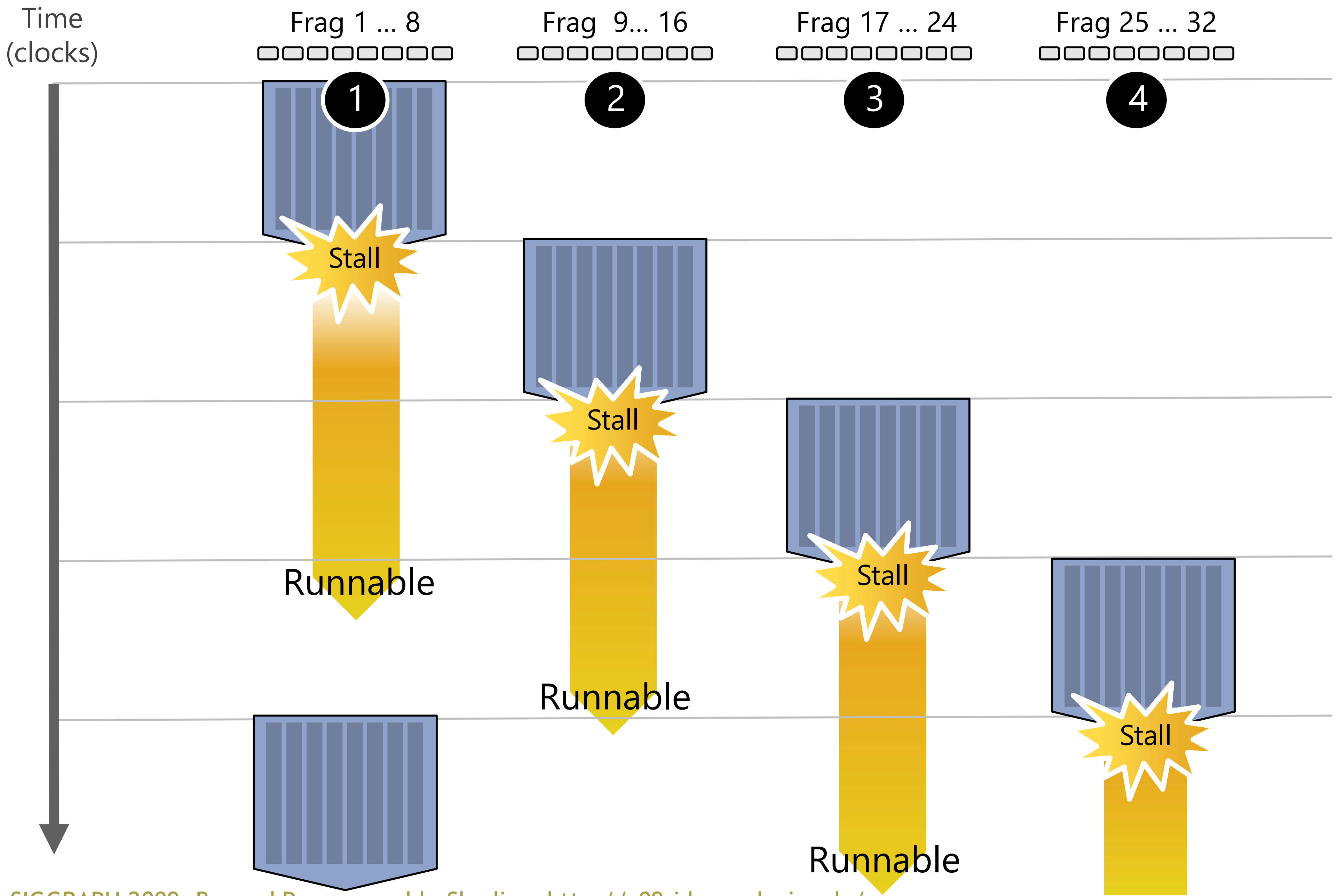
Hiding shader stalls



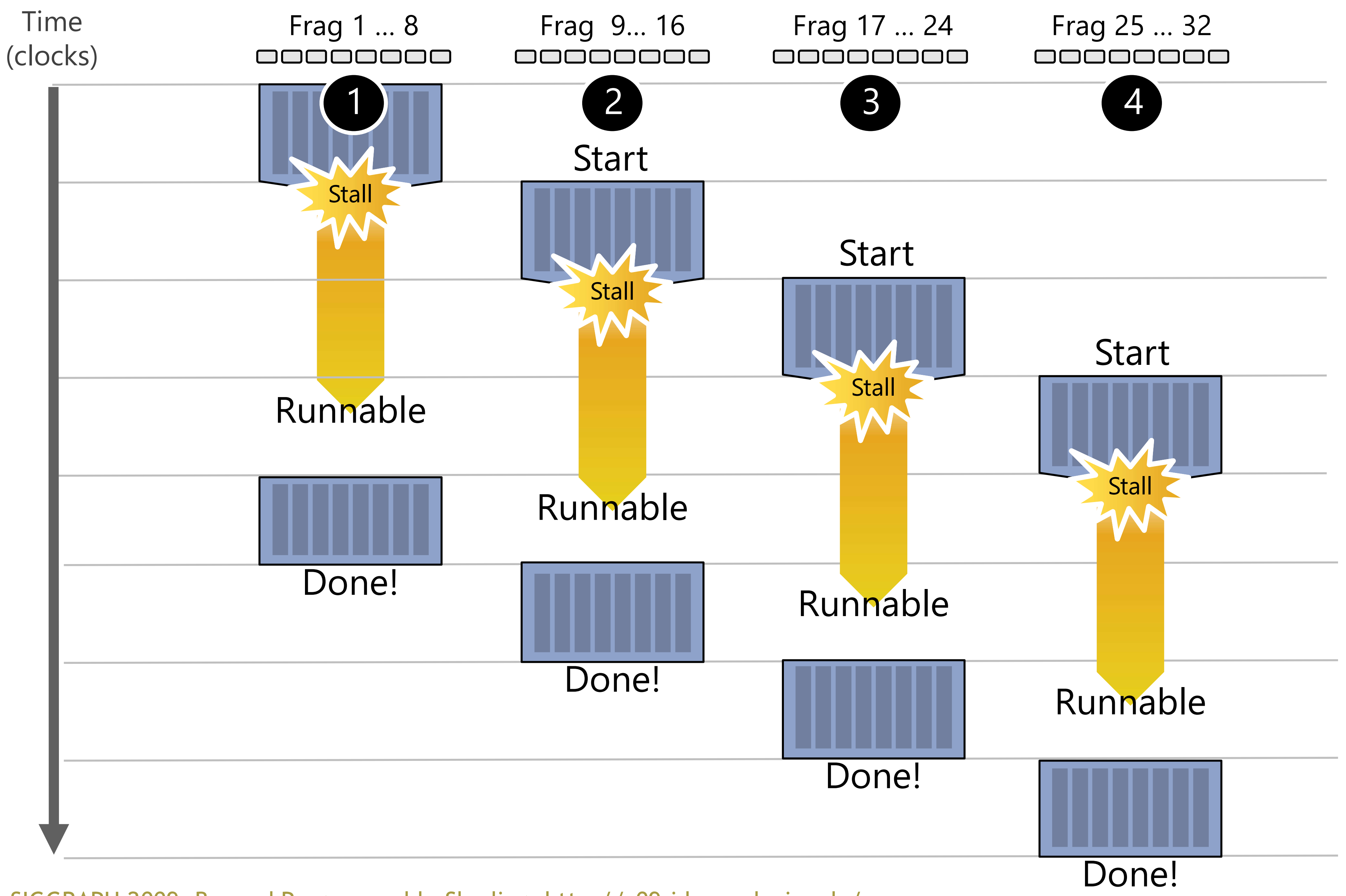
Hiding shader stalls



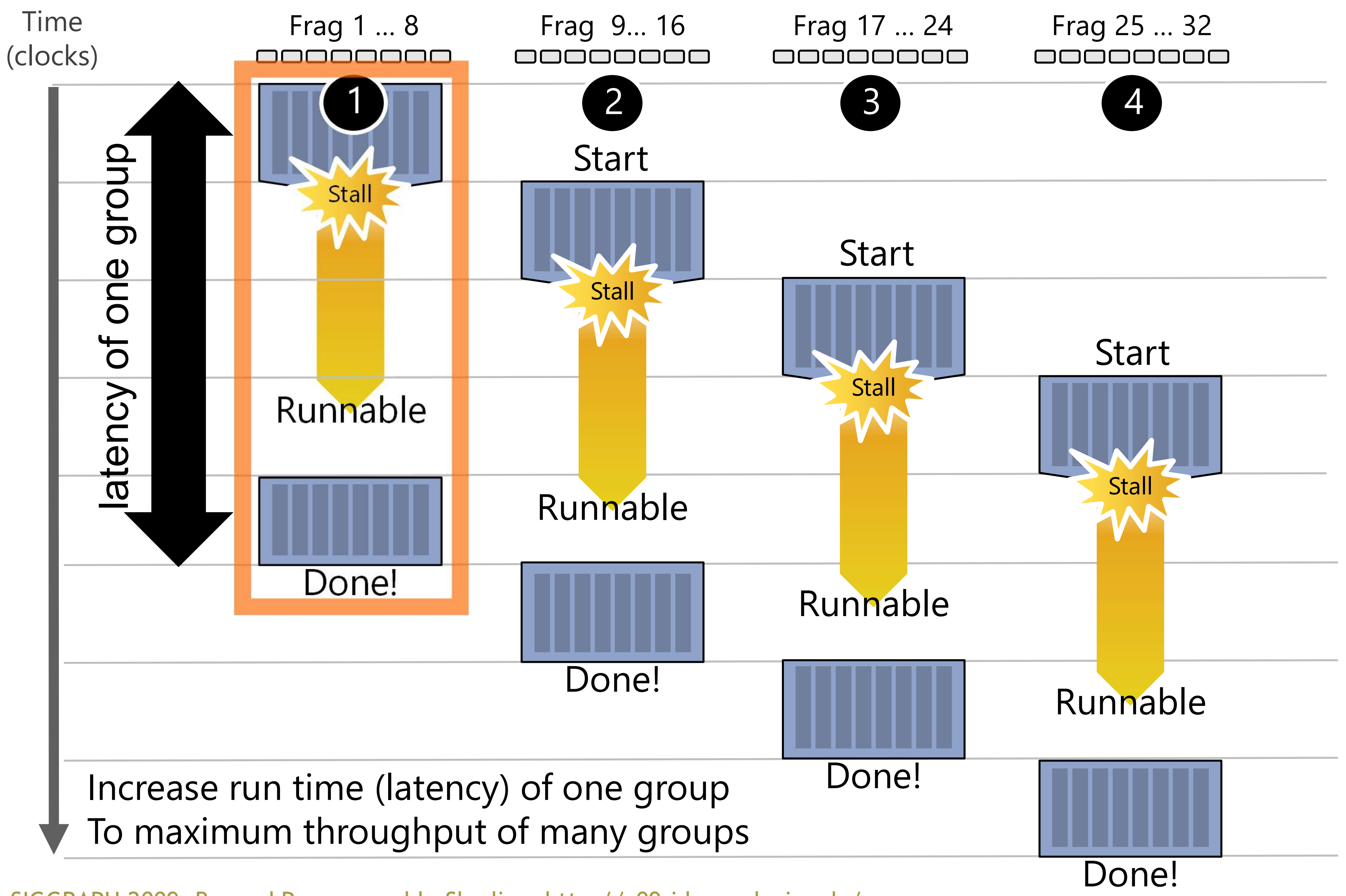
Hiding shader stalls



Hiding shader stalls

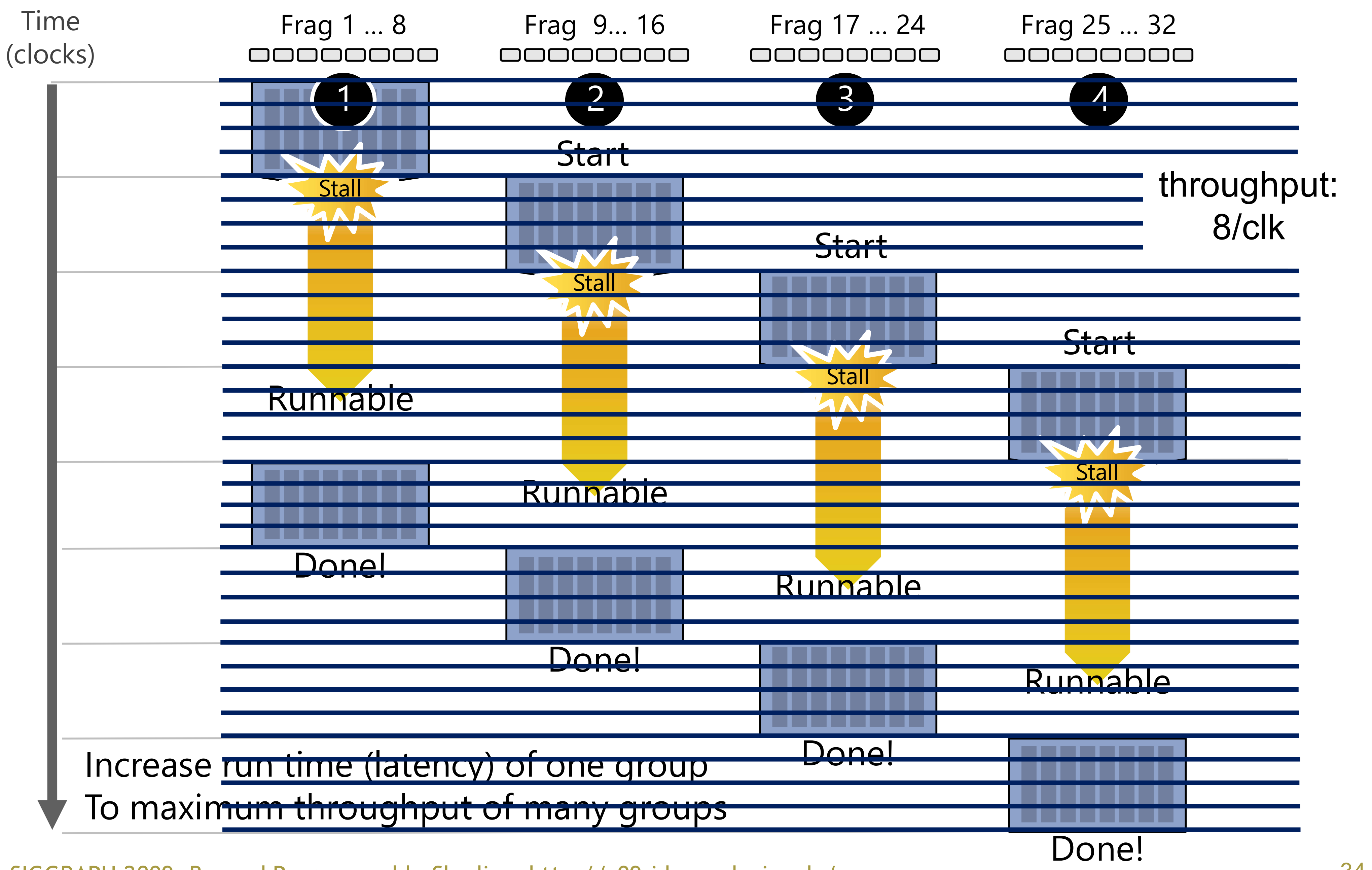


Hiding shader stalls



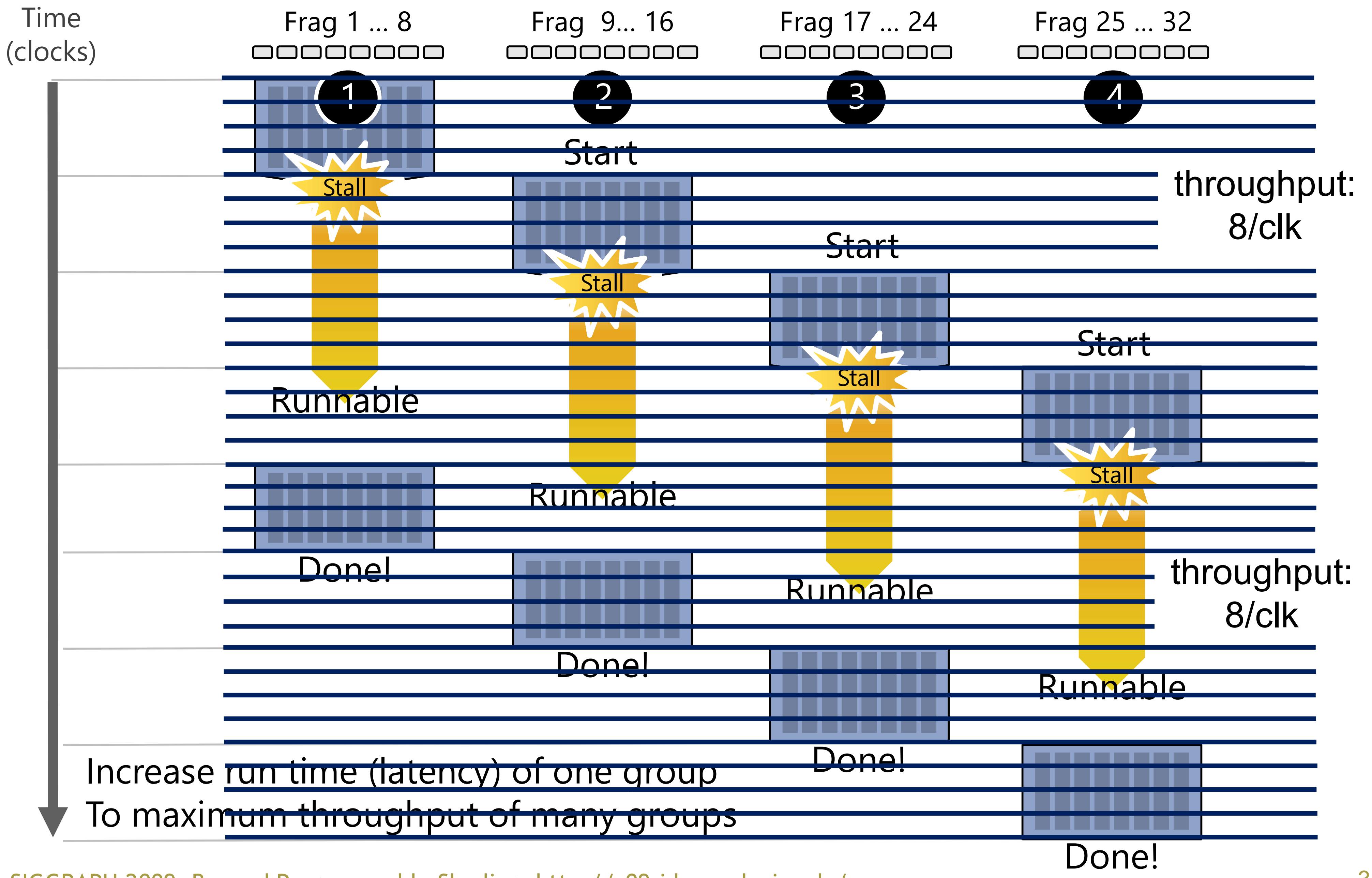
Increase run time (latency) of one group
To maximum throughput of many groups

Throughput! (4 groups of threads)



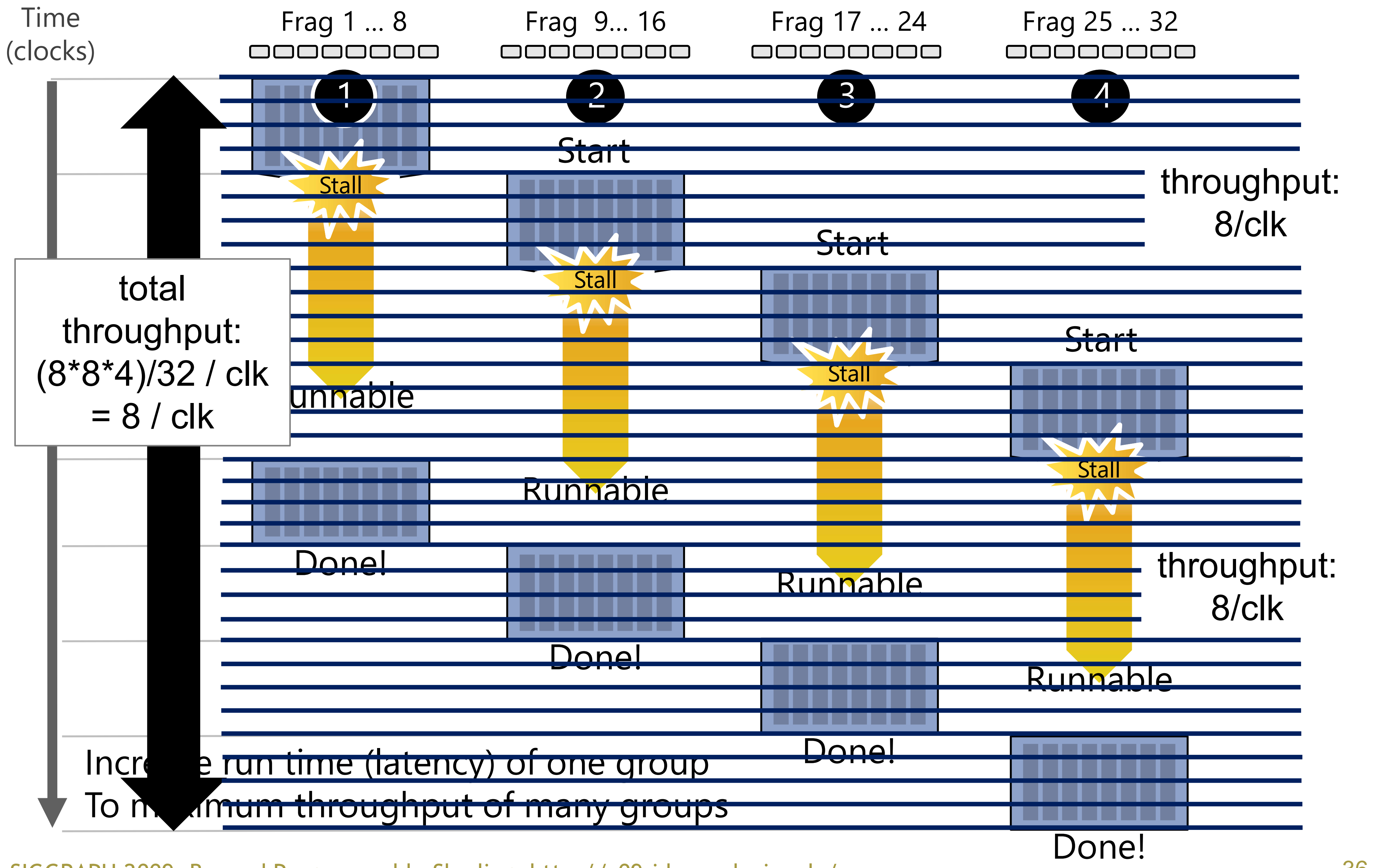
Increase run time (latency) of one group
To maximum throughput of many groups

Throughput! (4 groups of threads)



Increase run time (latency) of one group
To maximum throughput of many groups

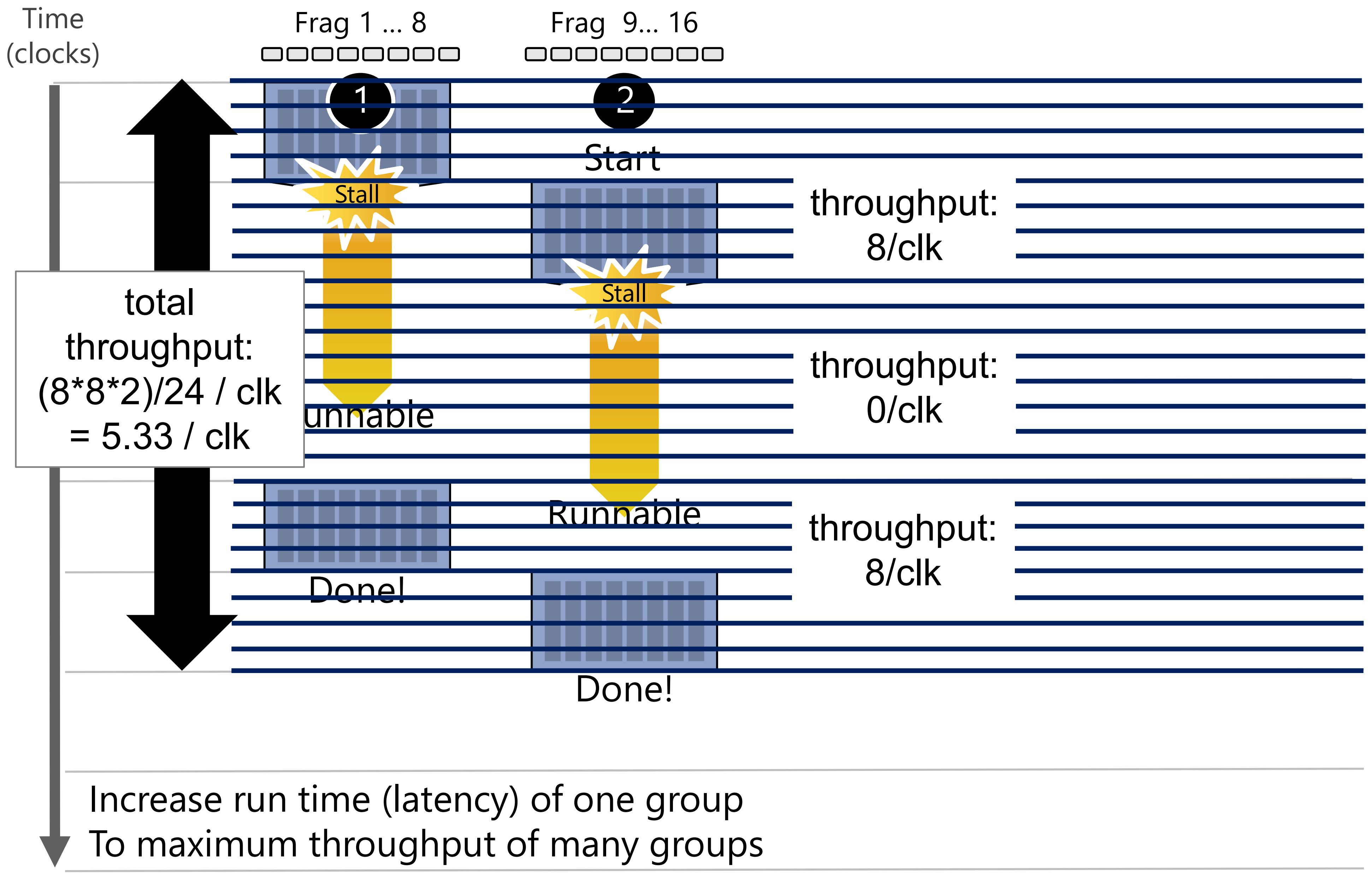
Throughput! (4 groups of threads)



total throughput: (8*8*4)/32 / clk = 8 / clk

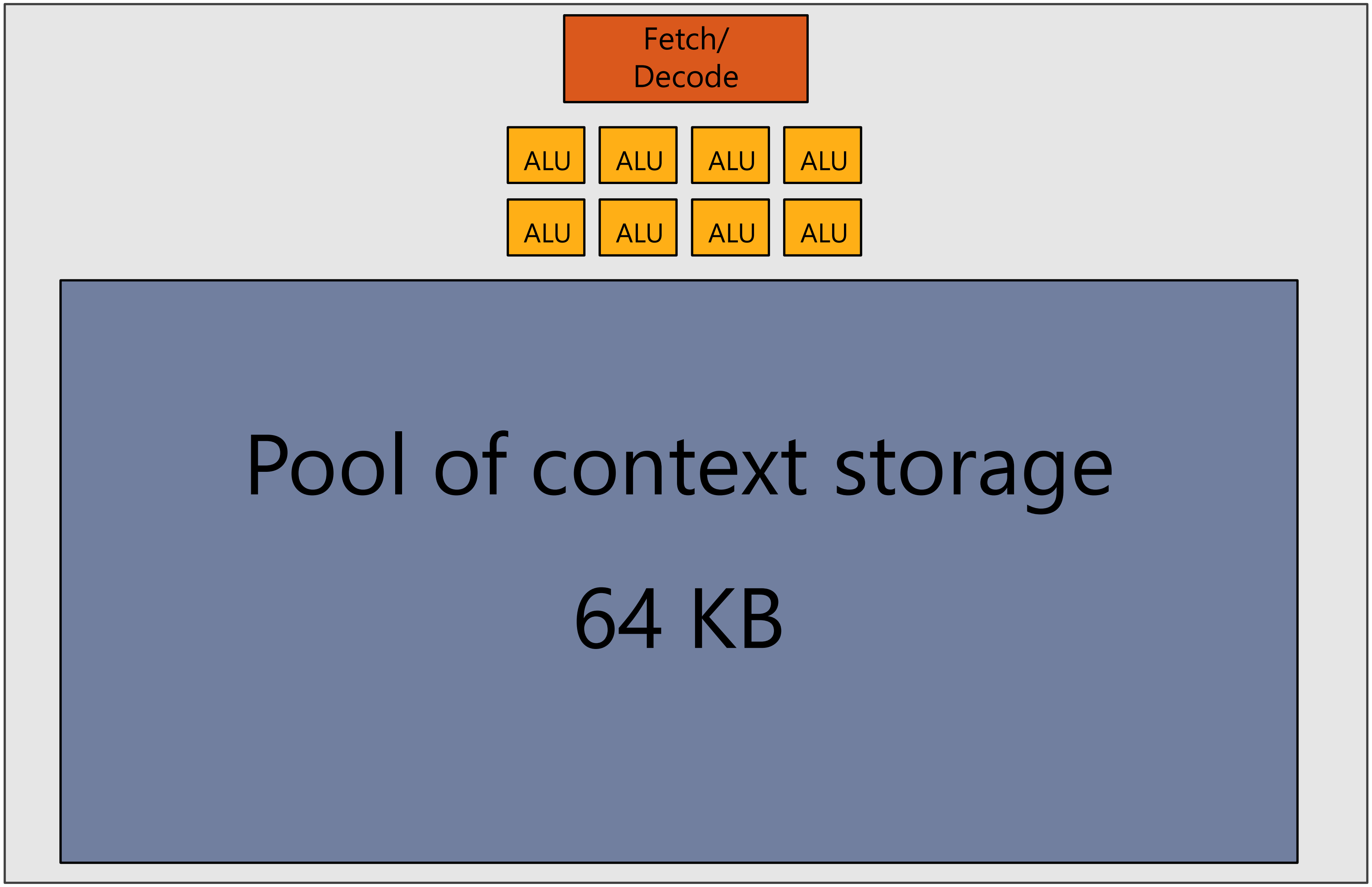
Increase run time (latency) of one group
To maximum throughput of many groups

Throughput! (2 groups of threads)



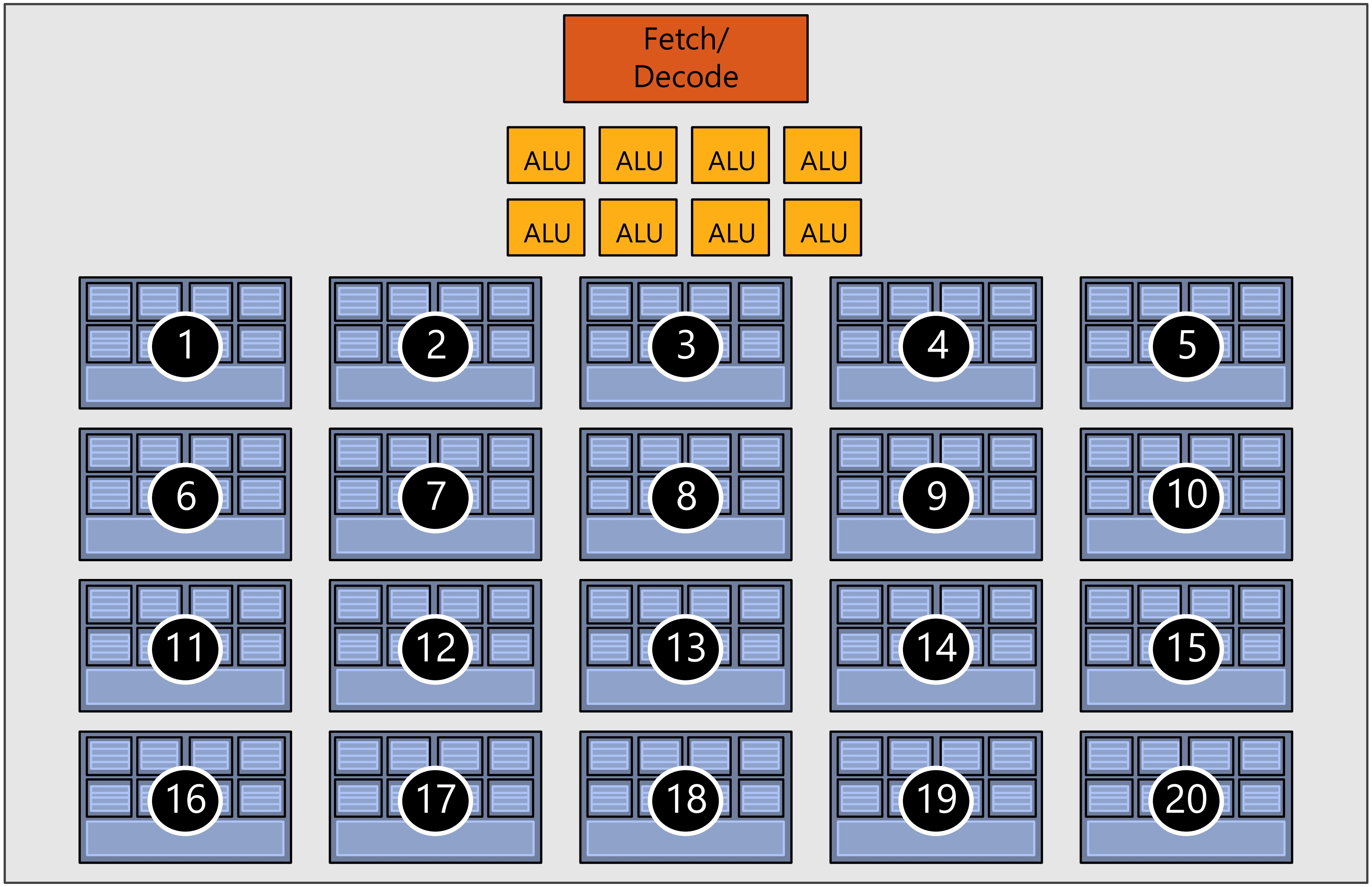
Increase run time (latency) of one group
To maximum throughput of many groups

Storing contexts

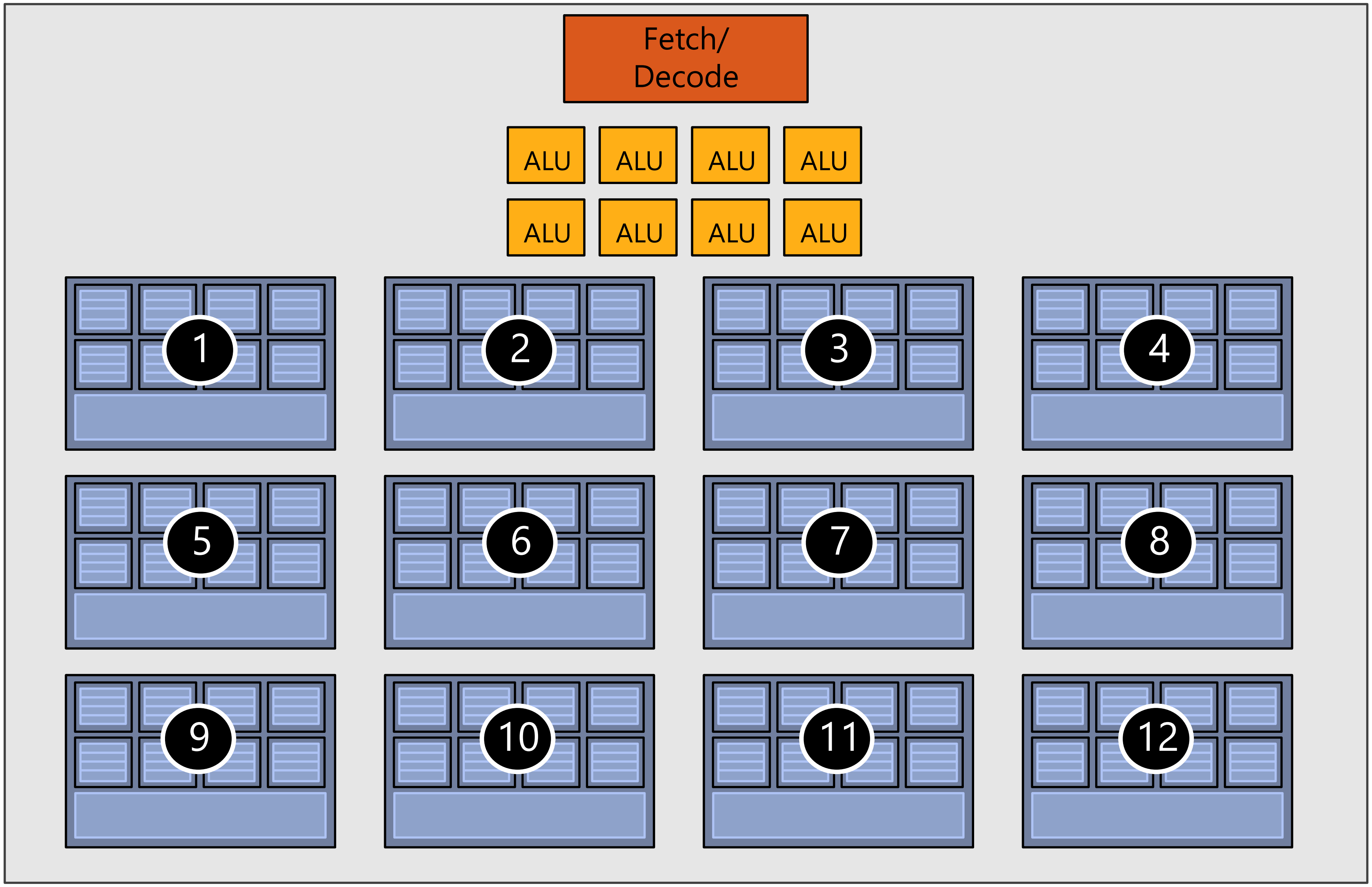


Twenty small contexts (few regs/thread)

(maximal latency hiding ability)

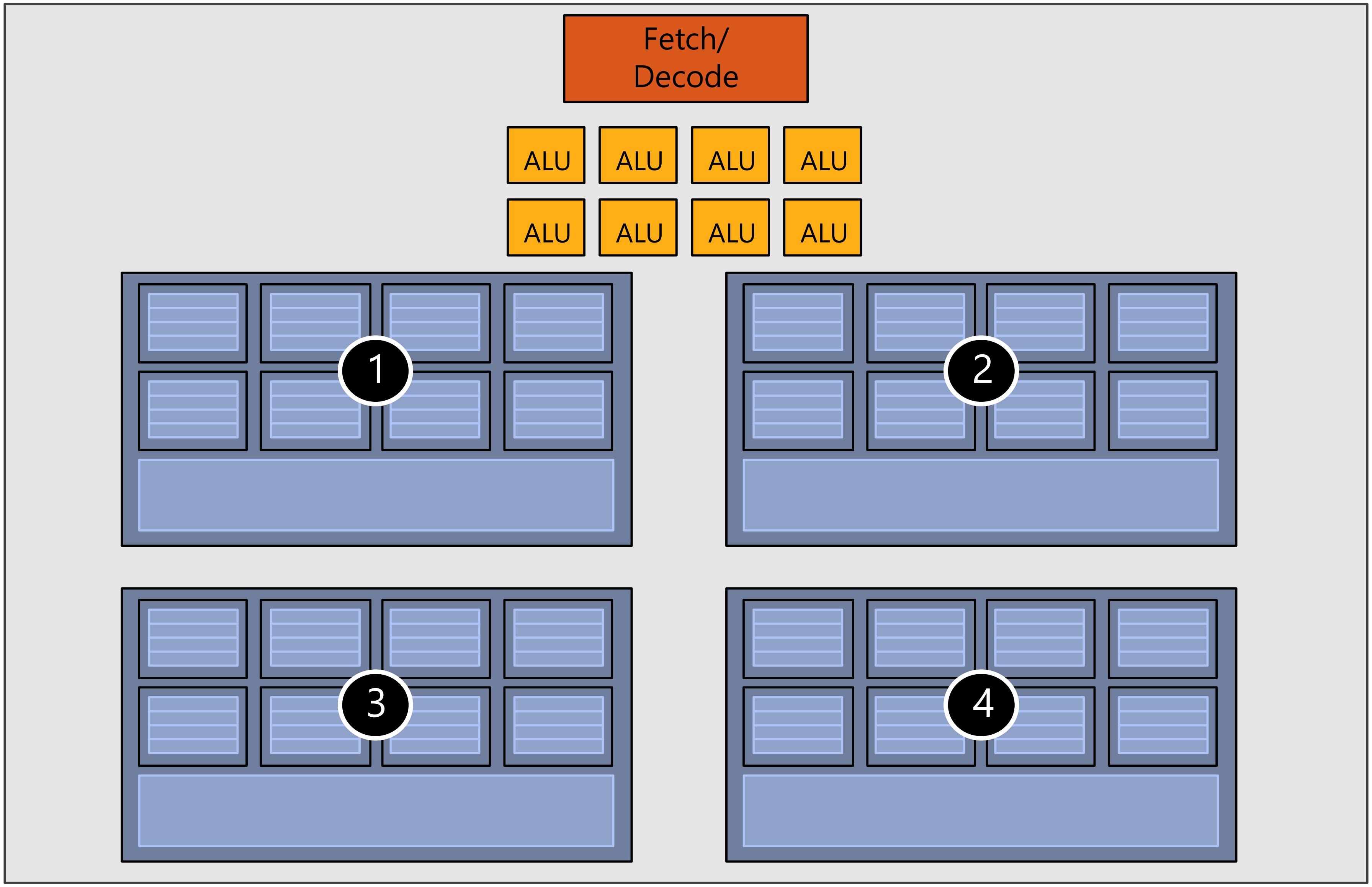


Twelve medium contexts (more regs/th.)



Four large contexts (many regs/thread)

(low latency hiding ability)



Concepts: SM Occupancy in CUDA (*TLP!*)

We need to hide latencies from

- Instruction pipelining hazards (RAW – read after write, etc.)
  (also: branches; behind branch, fetch instructions from different instruction stream)
- Memory access latency

First type of latency: Definitely need to hide! (it is always there)

Second type of latency: only need to hide if it does occur (of course not unusual)

Occupancy: How close are we to *maximum latency hiding ability?*
(how many threads are resident vs. how many could be)

See run time occupancy API, or Nsight Compute: `https://docs.nvidia.com/nsight-compute/NsightCompute/index.html#occupancy-calculator`

Complete GPU

16 cores

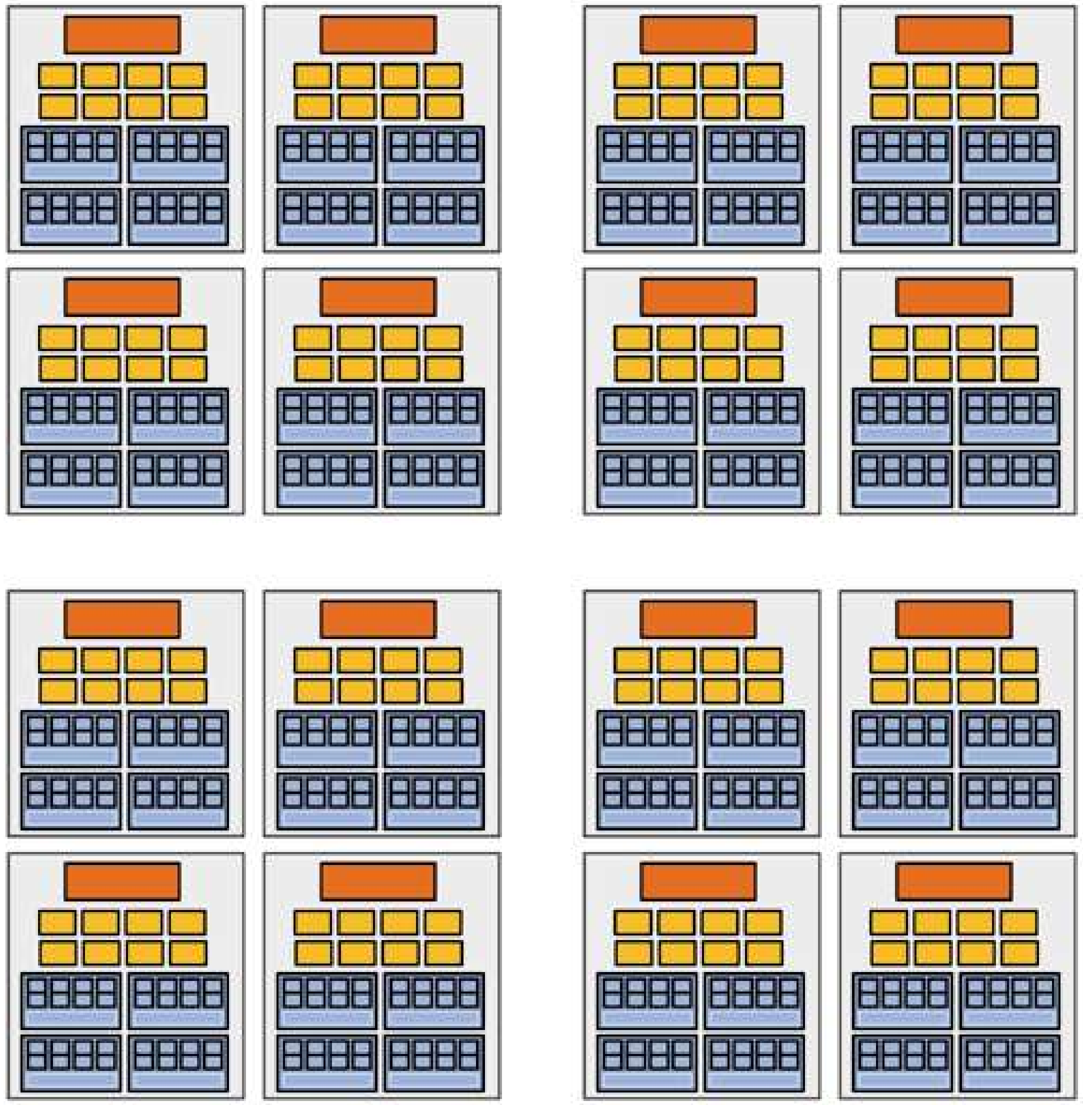
8 mul-add [mad] ALUs per core
(8*16 = **128** total)

16 simultaneous
instruction streams

64 (4*16) concurrent (but
interleaved) instruction streams

512 (8*4*16) concurrent
fragments (resident threads)

= **256 GFLOPs** (@ 1GHz)
(**128** * 2 [mad] * 1G)



Complete GPU

16 cores

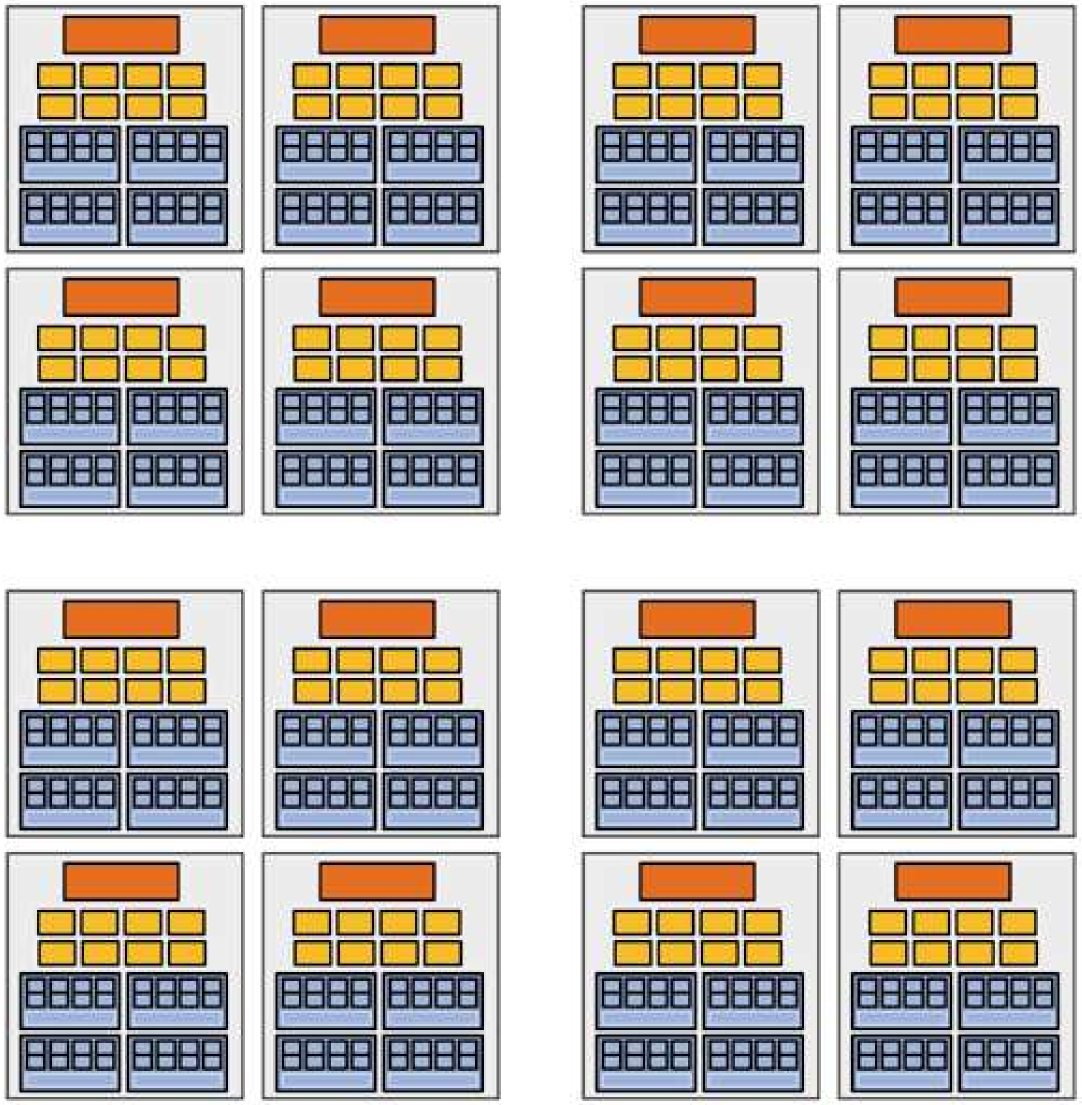
8 mul-add [mad] ALUs per core
(8*16 = **128** total)

16 simultaneous
instruction streams

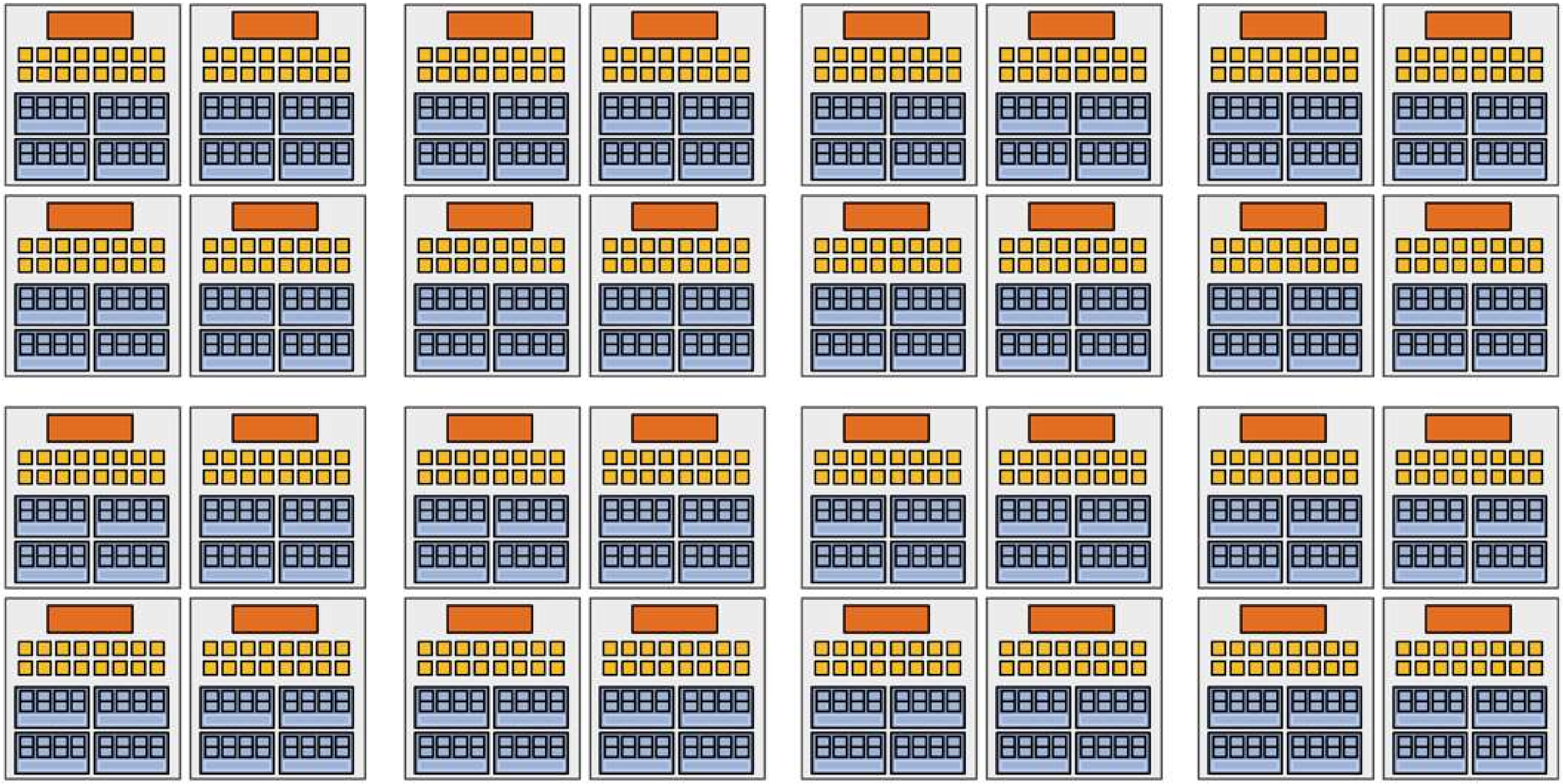
64 (4*16) concurrent (but
interleaved) instruction streams

512 (8*4*16) concurrent
fragments (resident threads)

= **256 GFLOPs** (@ 1GHz)
(**128** * 2 [mad] * 1G)

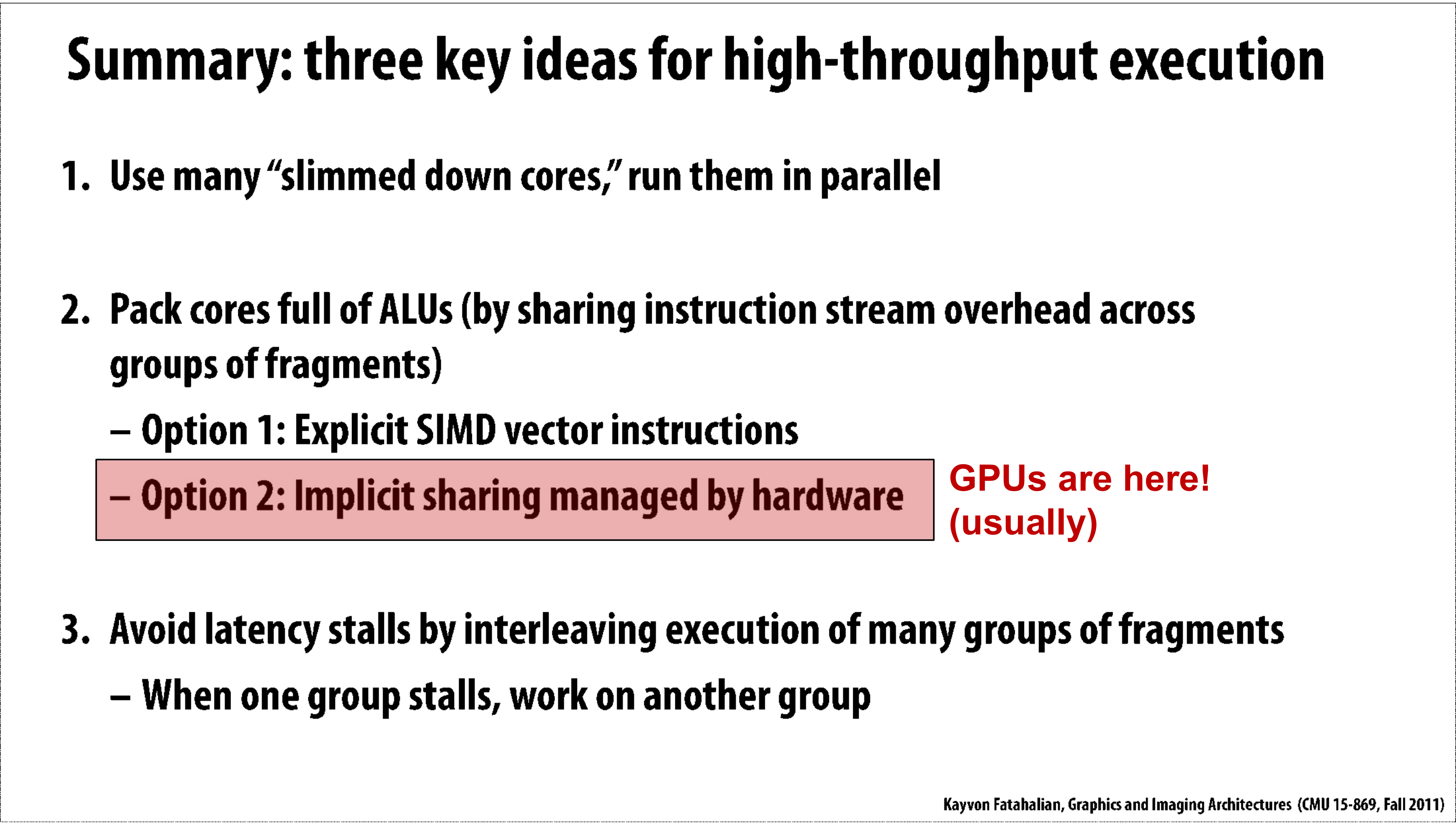


“Enthusiast” GPU (Some time ago :)



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

GPUs are here! (usually)

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

Thank you.