King Abdullah University of
Science and Technology

KAUST

CS 380 - GPU and GPGPU Programming
Lecture 6+7: GPU Architecture 5+6

Markus Hadwiger, KAUST

Reading Assignment #4 (until March 5)

Read (required):

- GLSL book, Chapter 7 (OpenGL Shading Language API)
- OpenGL Shading Language 4.3 specification: Chapter 2

`http://www.opengl.org/registry/doc/GLSLangSpec.4.30.6.pdf`

- Programming Massively Parallel Processors book, Chapter 2 (*History of GPU Computing*)

Read (optional):

- OpenGL 4.0 Shading Language Cookbook, Chapter 3
- Download OpenGL 4.3 specification

`http://www.opengl.org/registry/doc/glspec43.core.20120806.pdf`

See more at: **`http://www.opengl.org/documentation/specs/`**

Quiz #1: Mar. 5

Organization

- First 30 min of lecture
- No material (book, notes, ...) allowed

Content of questions

- Lectures (both actual lectures and slides)
- Reading assigments
- Programming assignments (algorithms, methods)
- Solve short practical examples

SIGGRAPH2009 NEW ORLEANS

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

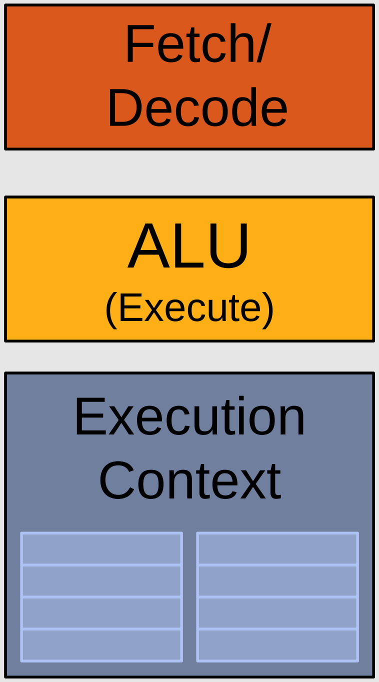
Kayvon Fatahalian
Stanford University

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 processing core) designs
  2. Optimize shaders/compute kernels
  3. Establish intuition: what workloads might benefit from the design of these architectures?

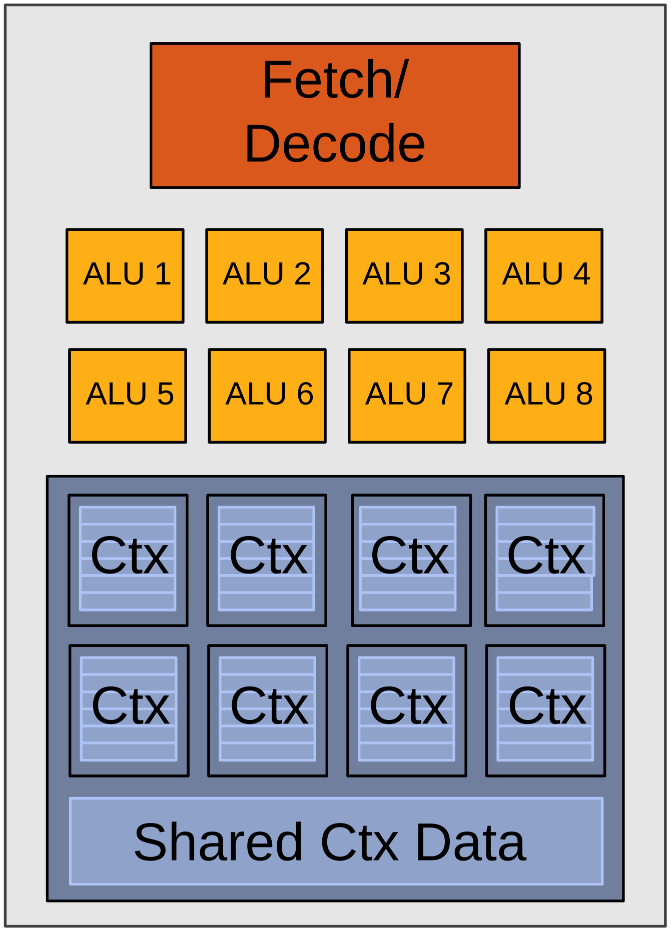
Slimming down



Idea #1:

Remove components that help a single instruction stream run fast

Add ALUs



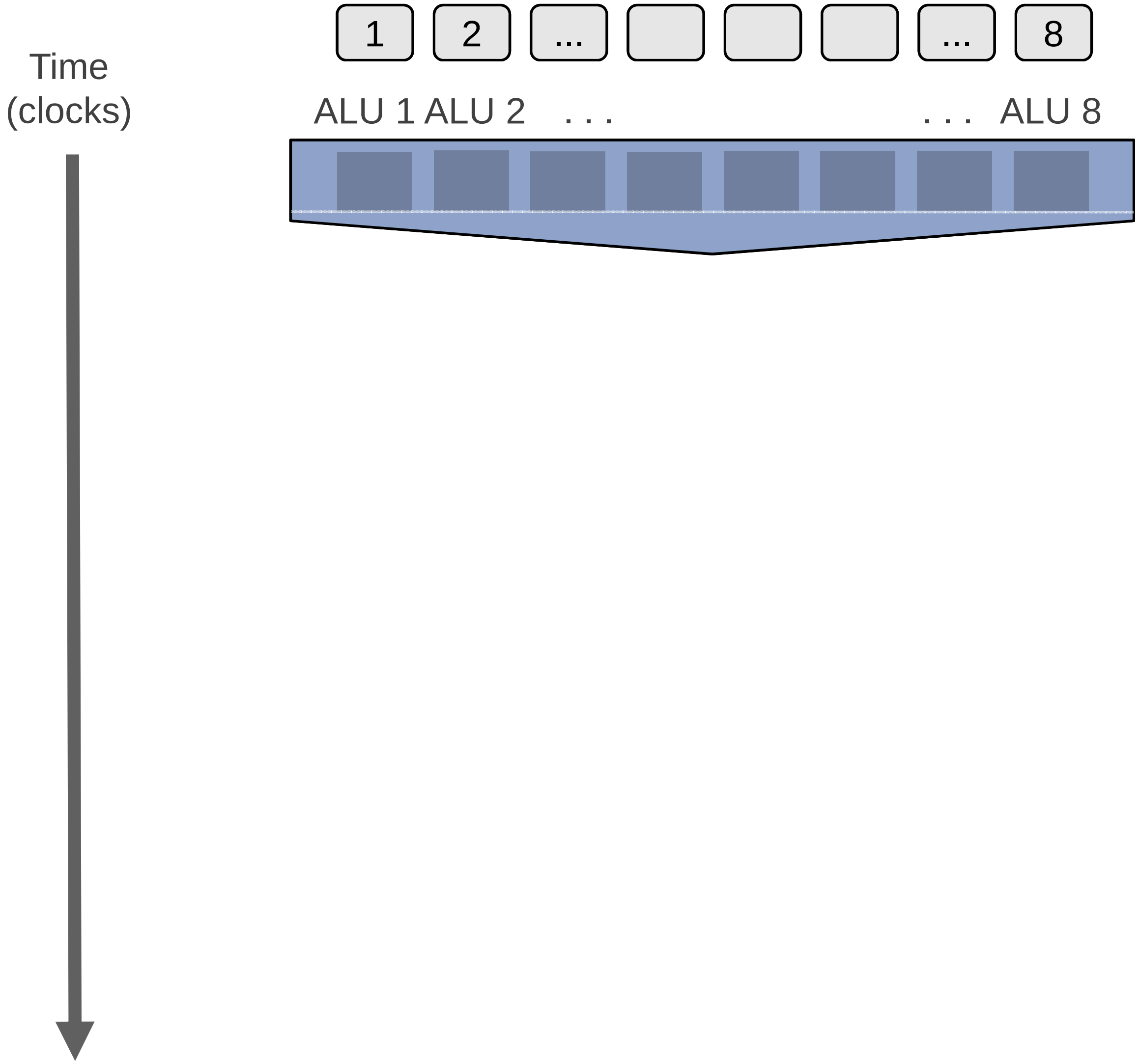
Idea #2:

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

SIMD processing

(or SIMT, SPMD)

But what about branches?

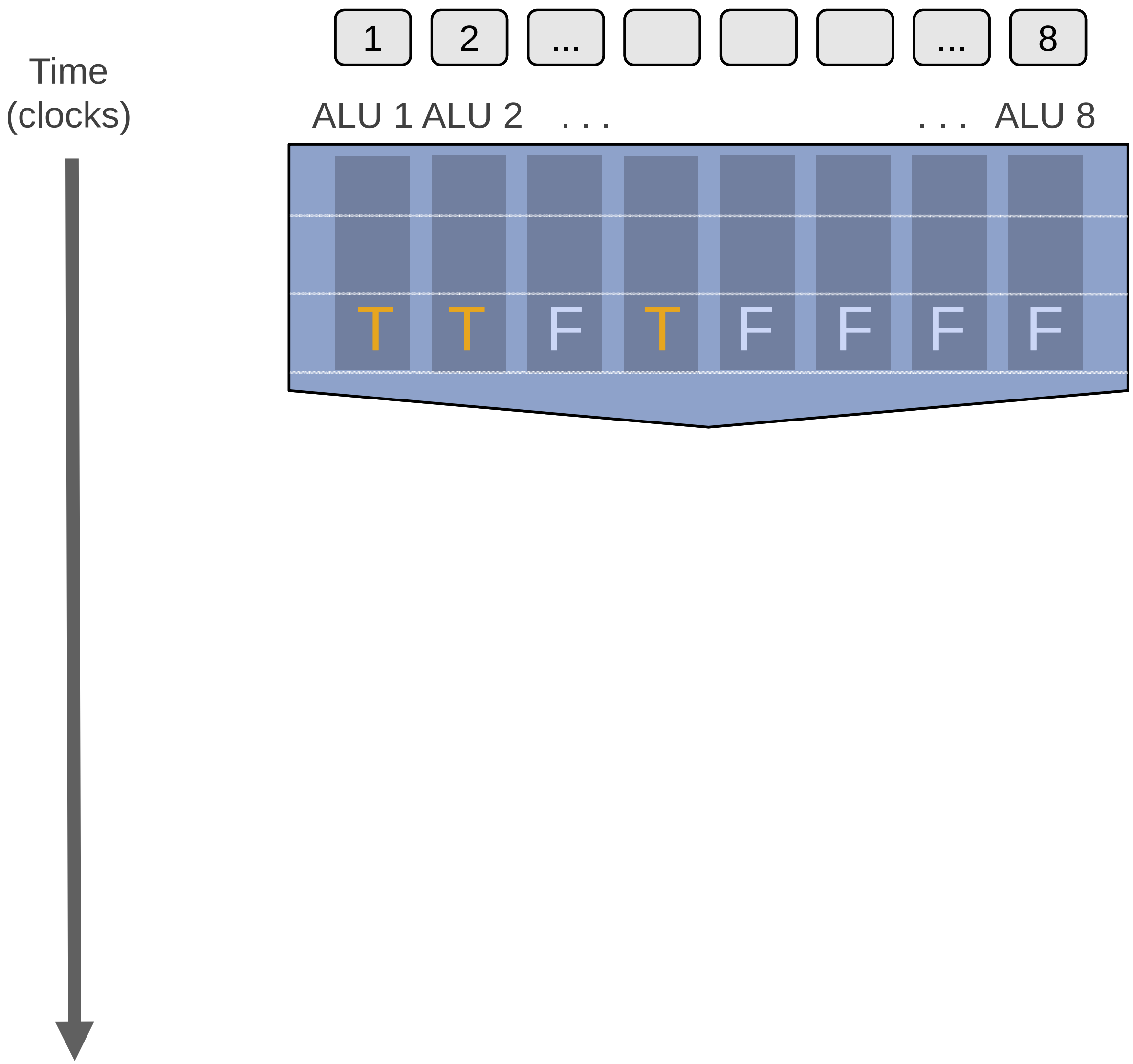


```
<unconditional
shader code>

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?

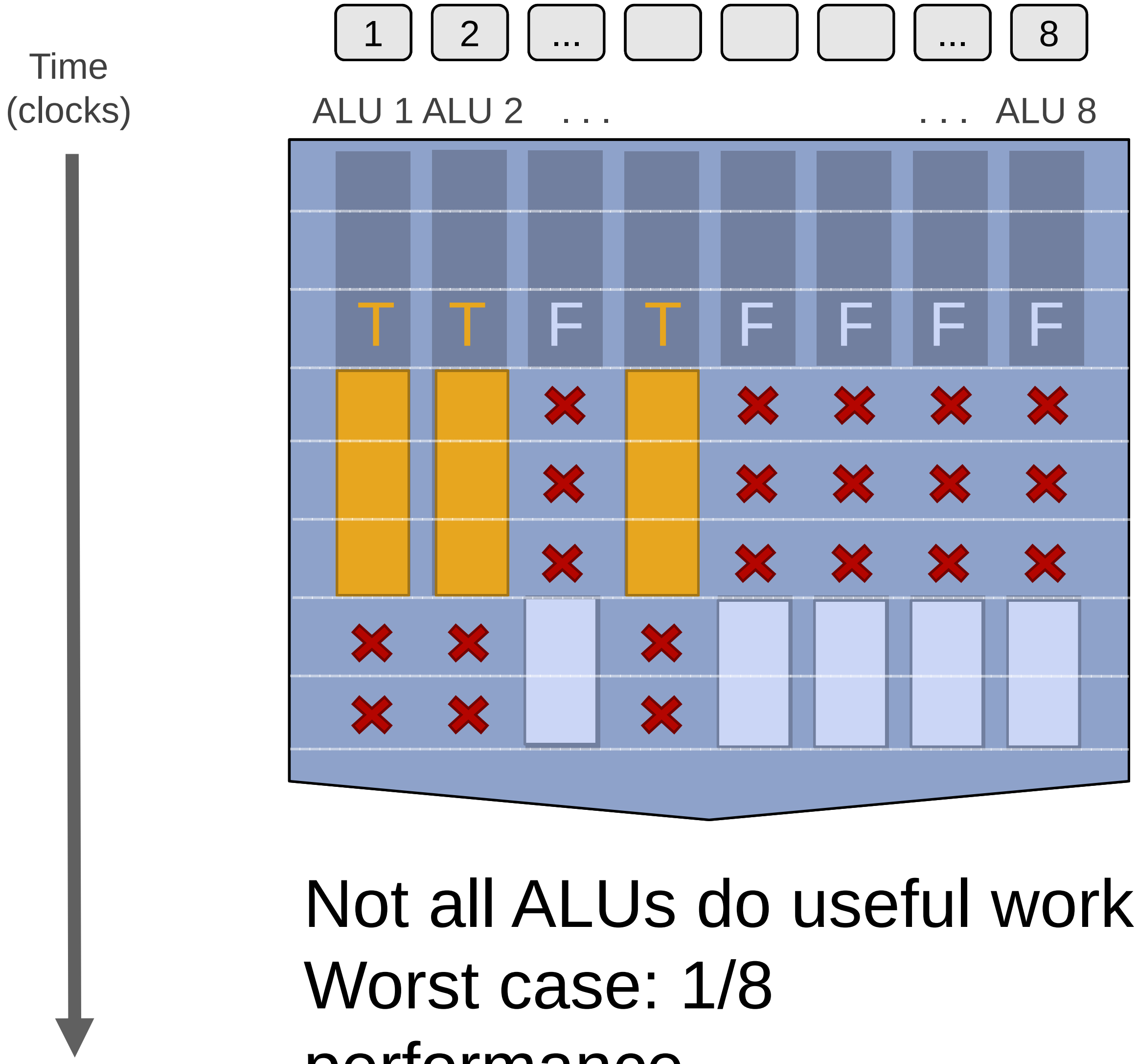


```
 <unconditional
 shader code>

 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?



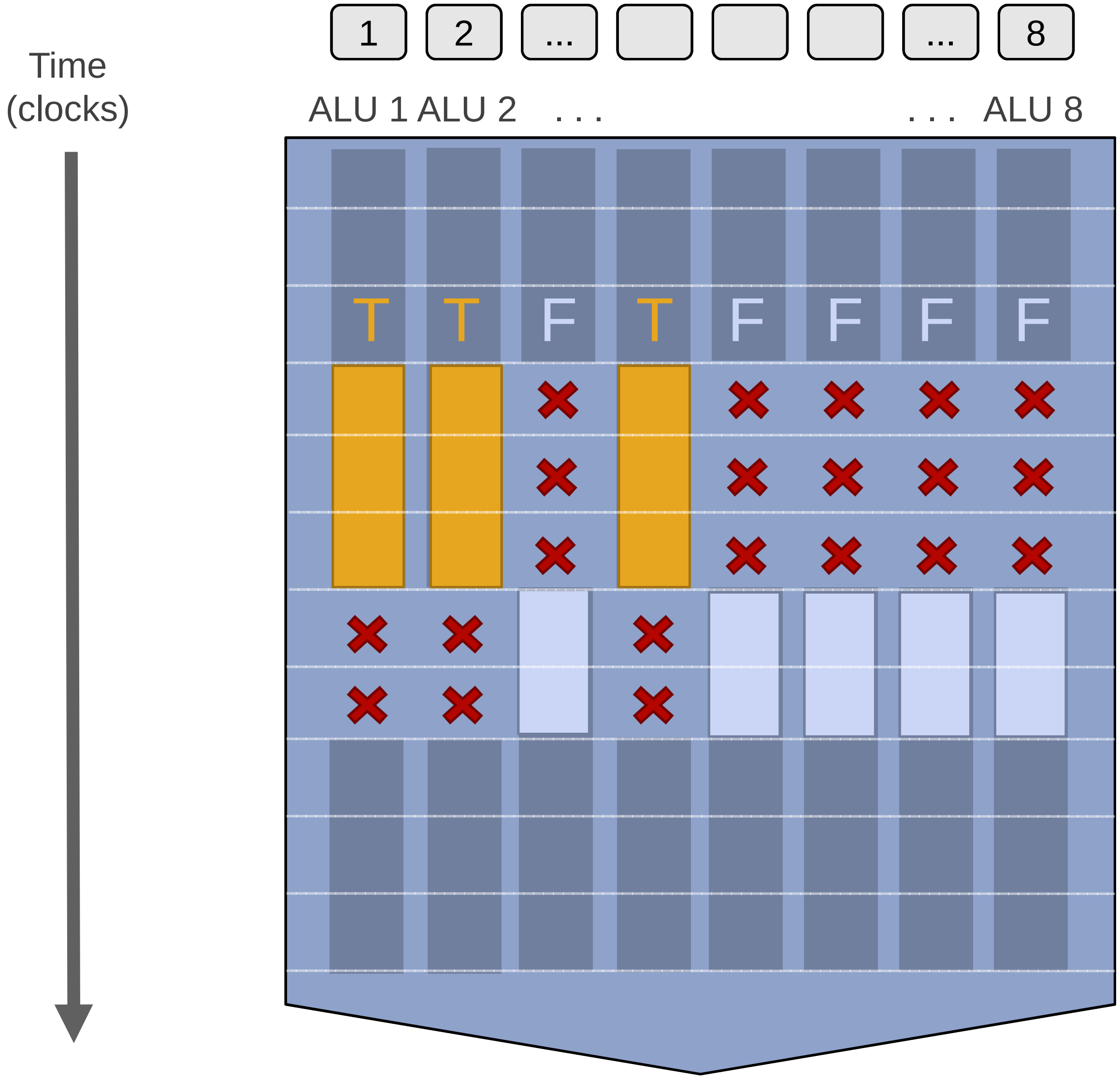
```
<unconditional
shader code>

if (x > 0) {
      y = pow(x, exp);
      y *= Ks;
      refl = y + Ka;
} else {
      x = 0;
      refl = Ka;
}

<resume unconditional
shader code>
```

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

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>
```

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

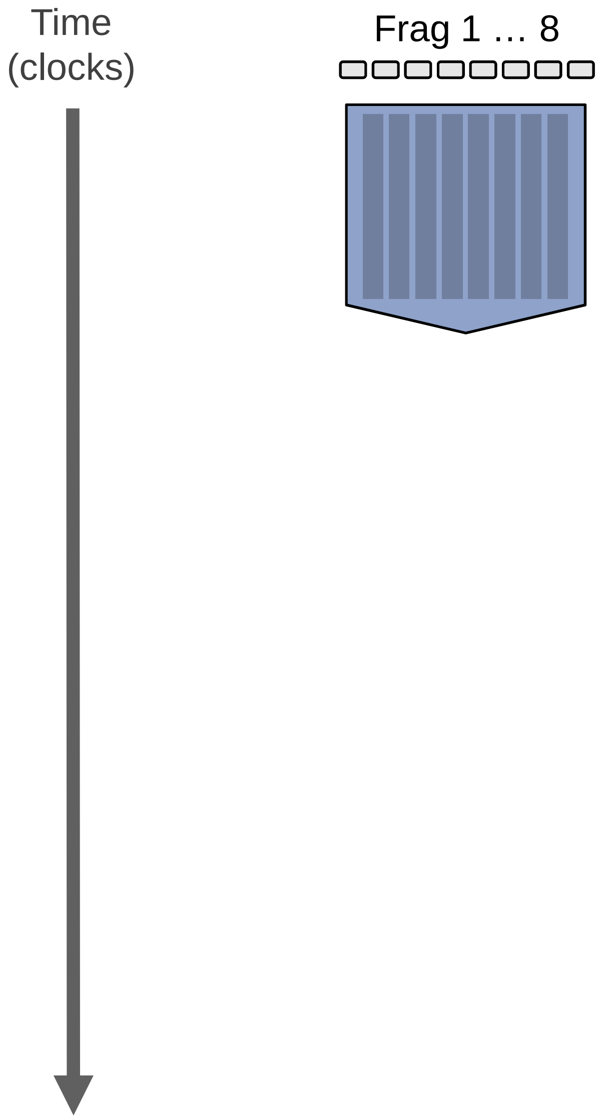
We’ve removed the fancy caches and logic that helps avoid stalls.

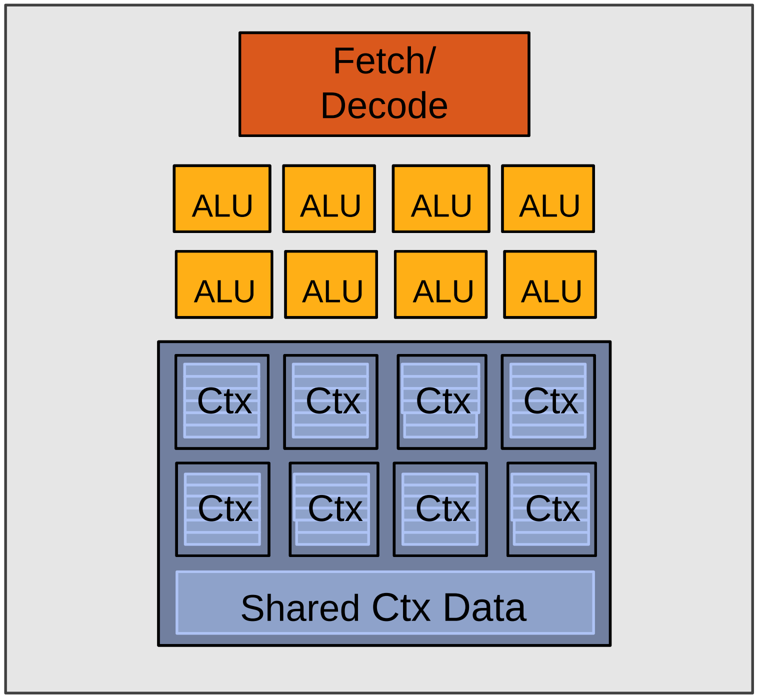
But we have LOTS of independent fragments.

Idea #3:

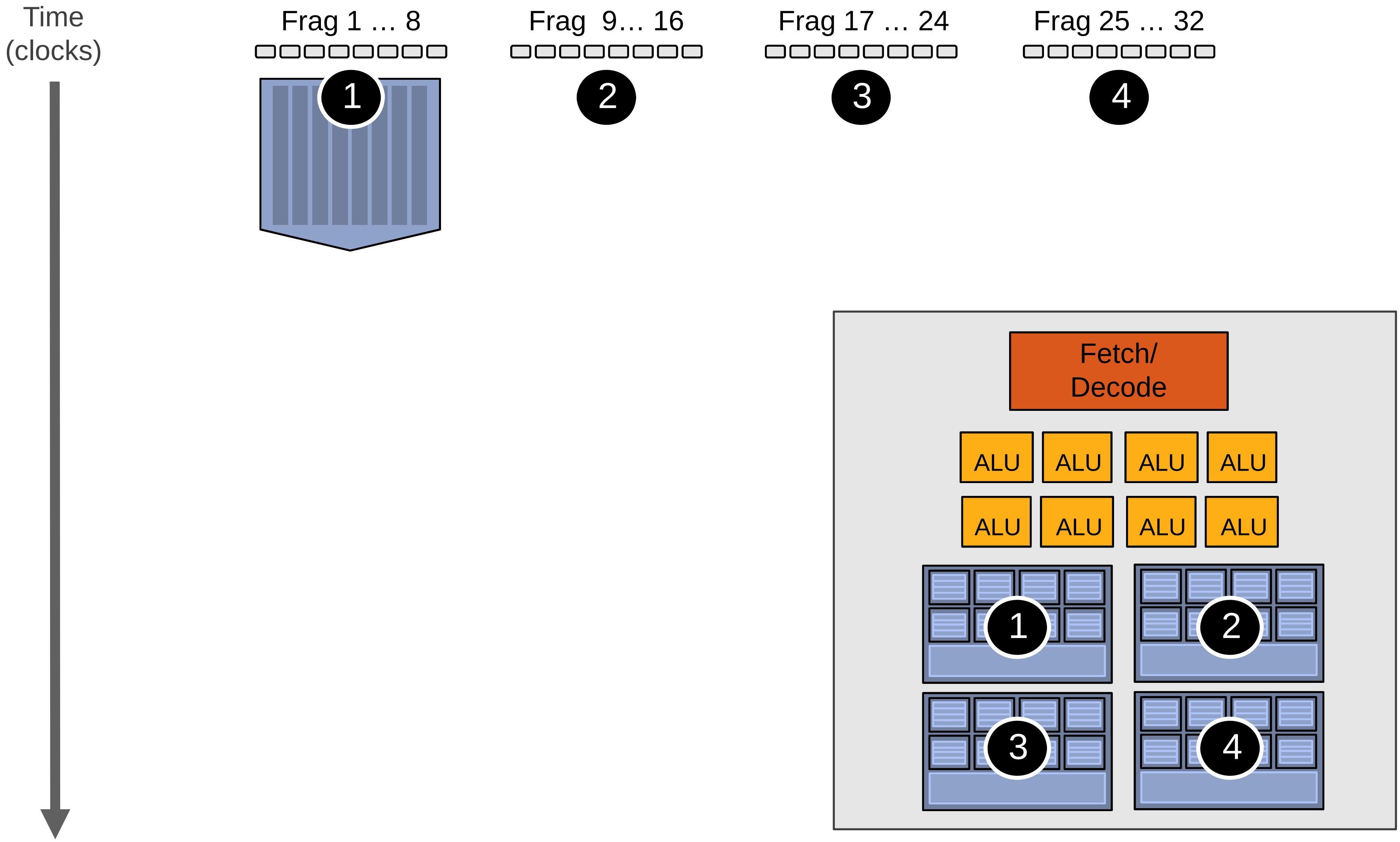
Interleave processing of many fragments on a single core to avoid stalls caused by high latency operations.

Hiding shader stalls

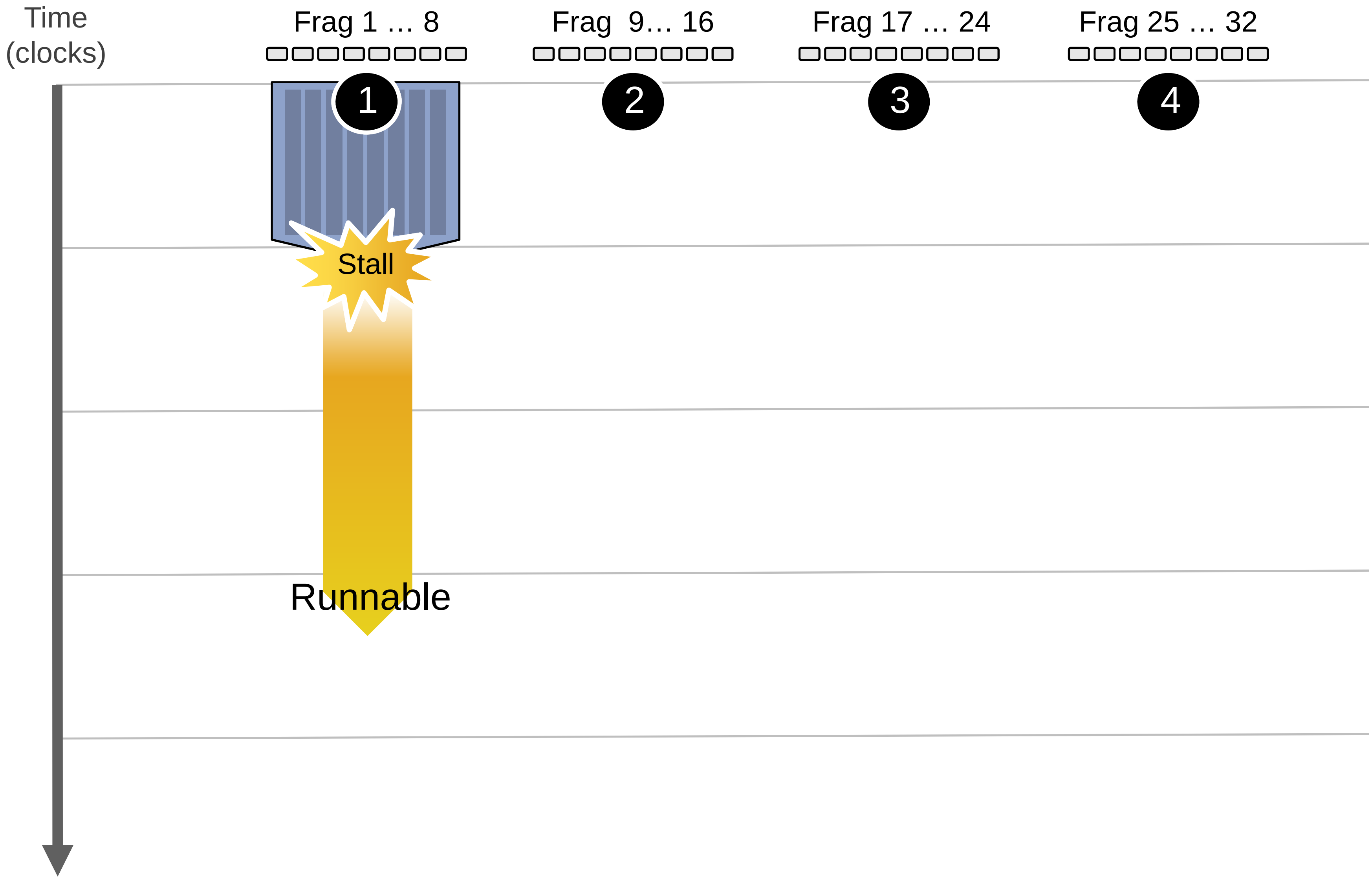




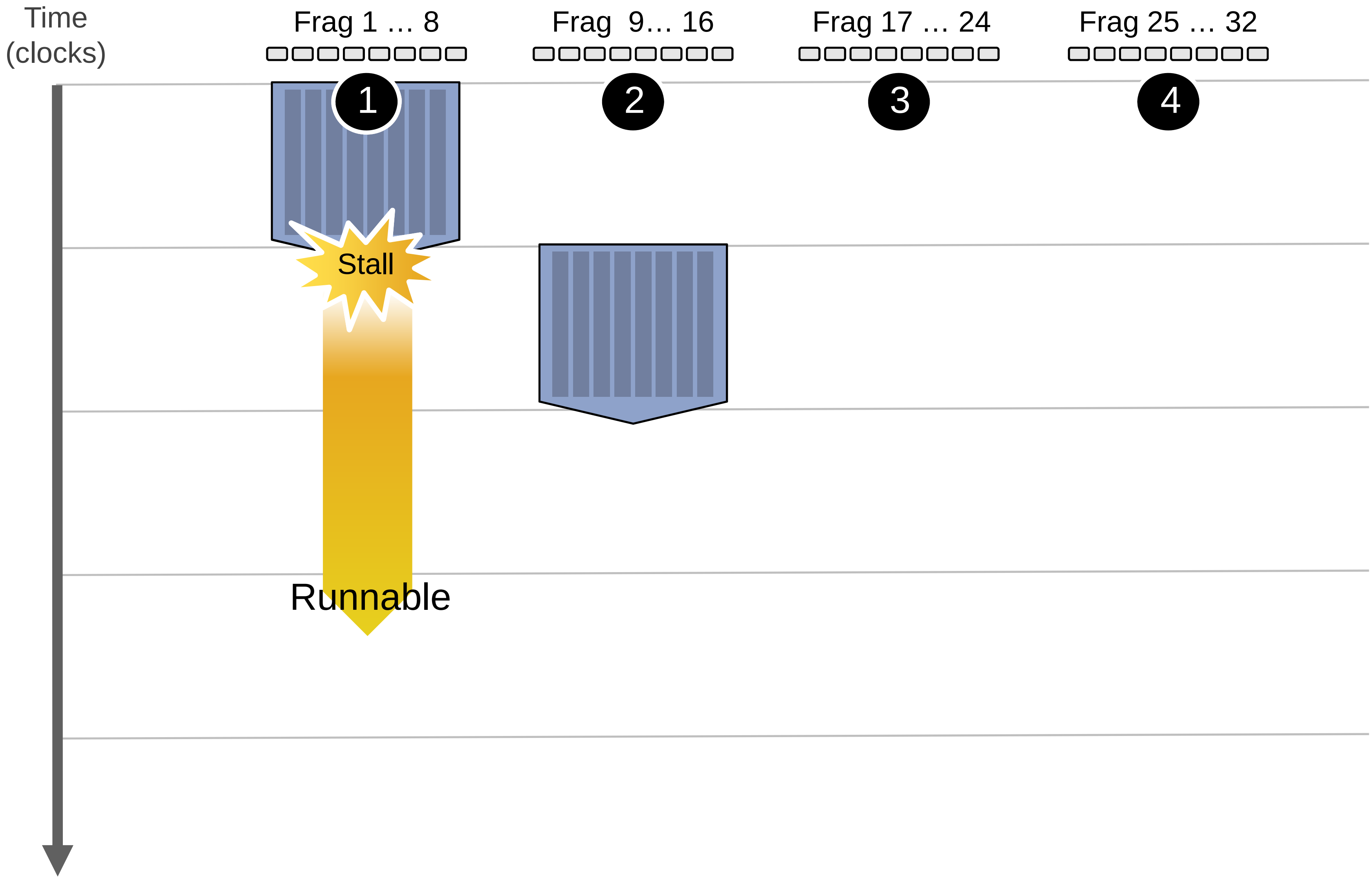
Hiding shader stalls



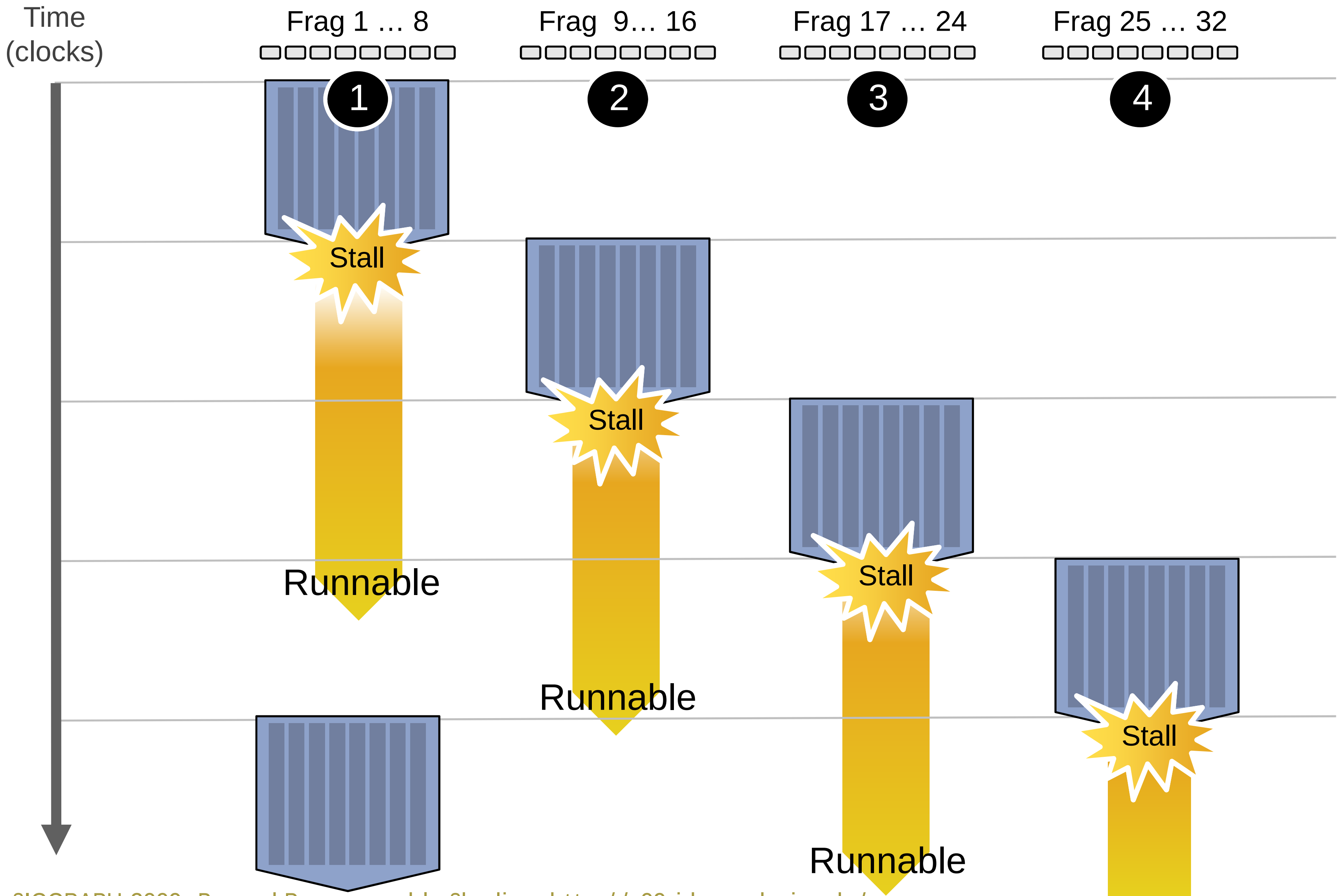
Hiding shader stalls



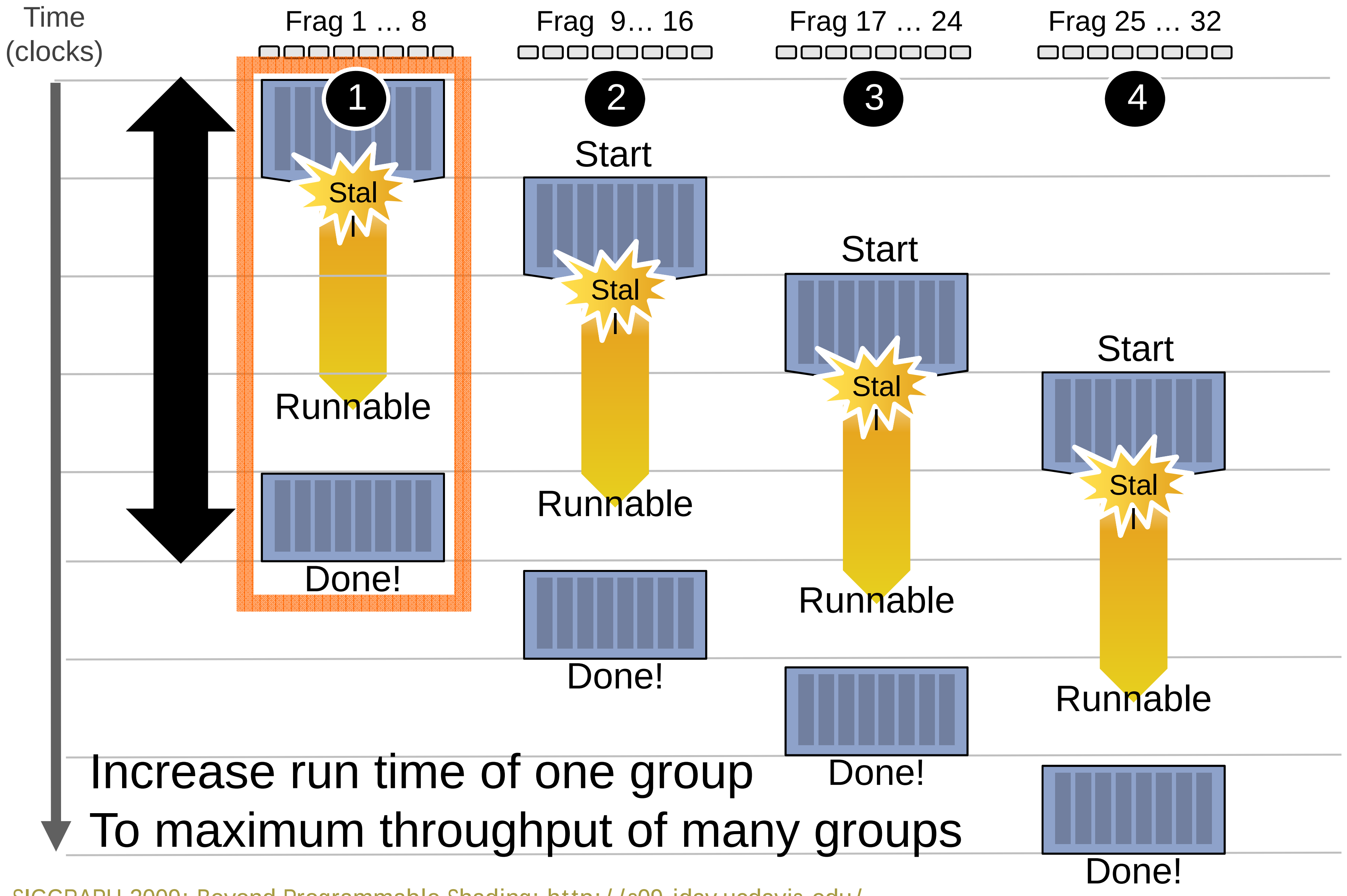
Hiding shader stalls



Hiding shader stalls

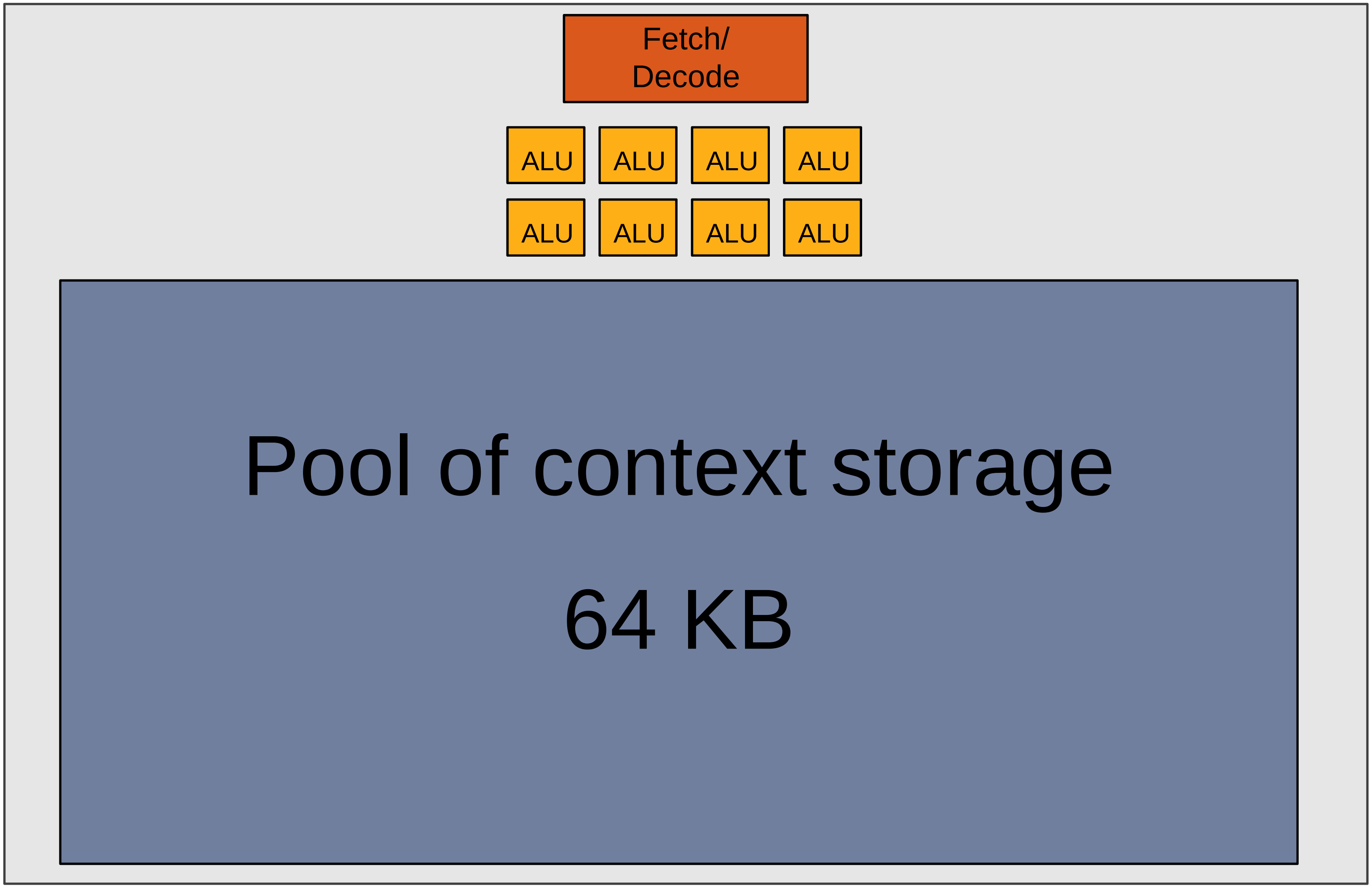


Throughput!



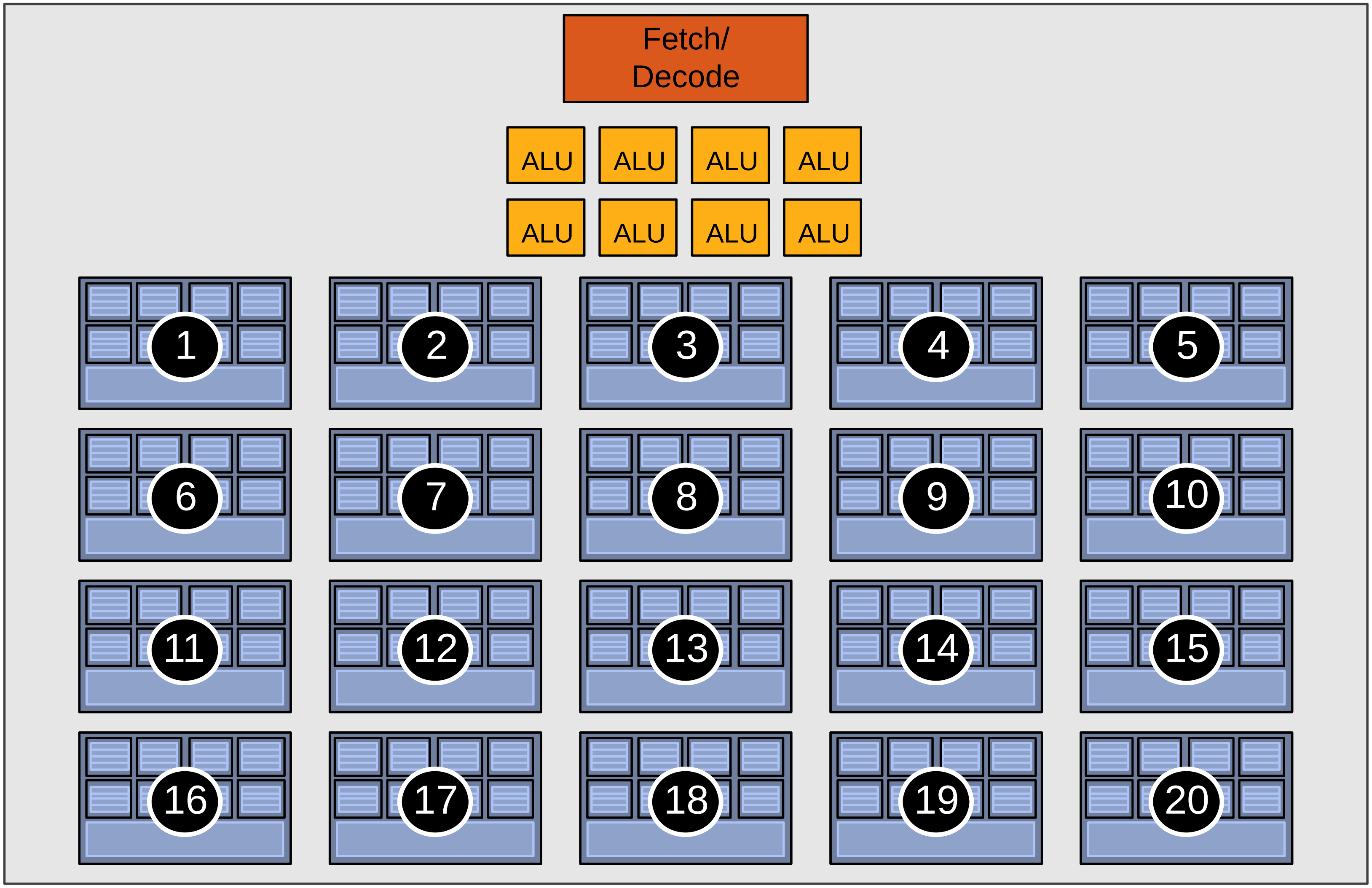
Increase run time of one group
To maximum throughput of many groups

Storing contexts

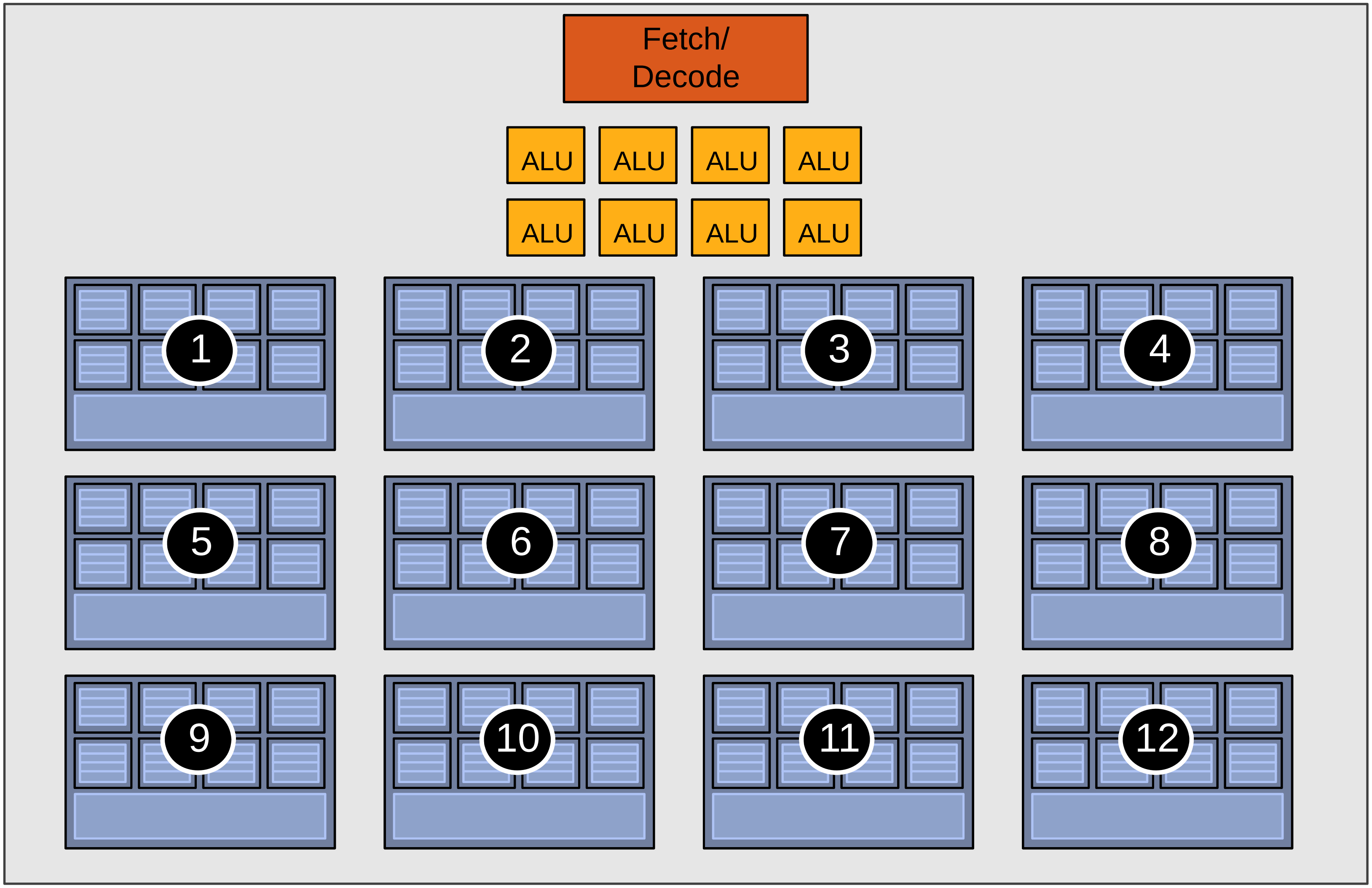


Twenty small contexts

(maximal latency hiding ability)

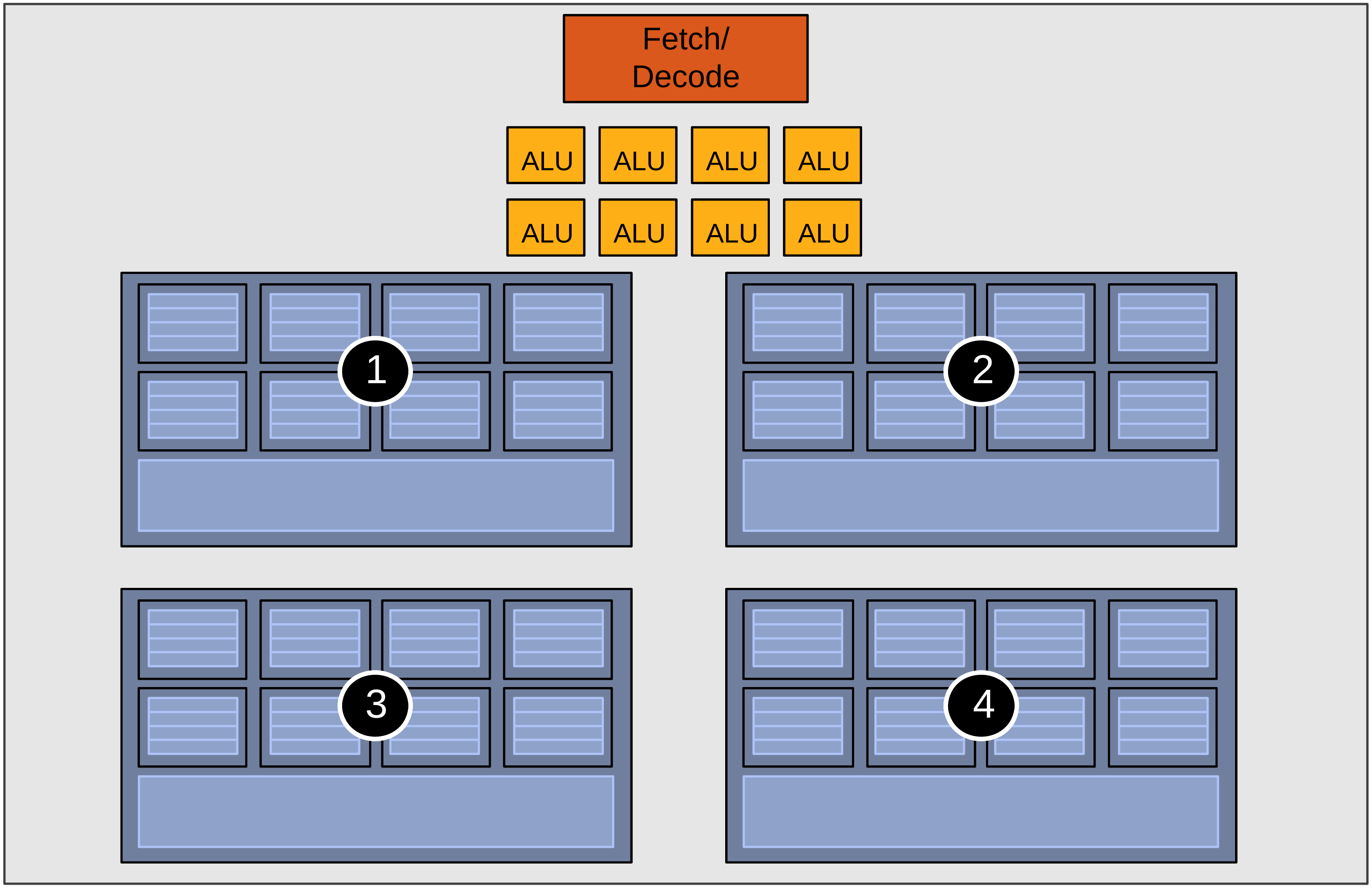


Twelve medium contexts



Four large contexts

(low latency hiding ability)



Clarification

Interleaving between contexts can be managed by HW or SW (or both!)

- NVIDIA / AMD Radeon GPUs
  - HW schedules / manages all contexts (lots of them)
  - Special on-chip storage holds fragment state
- Intel MIC/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

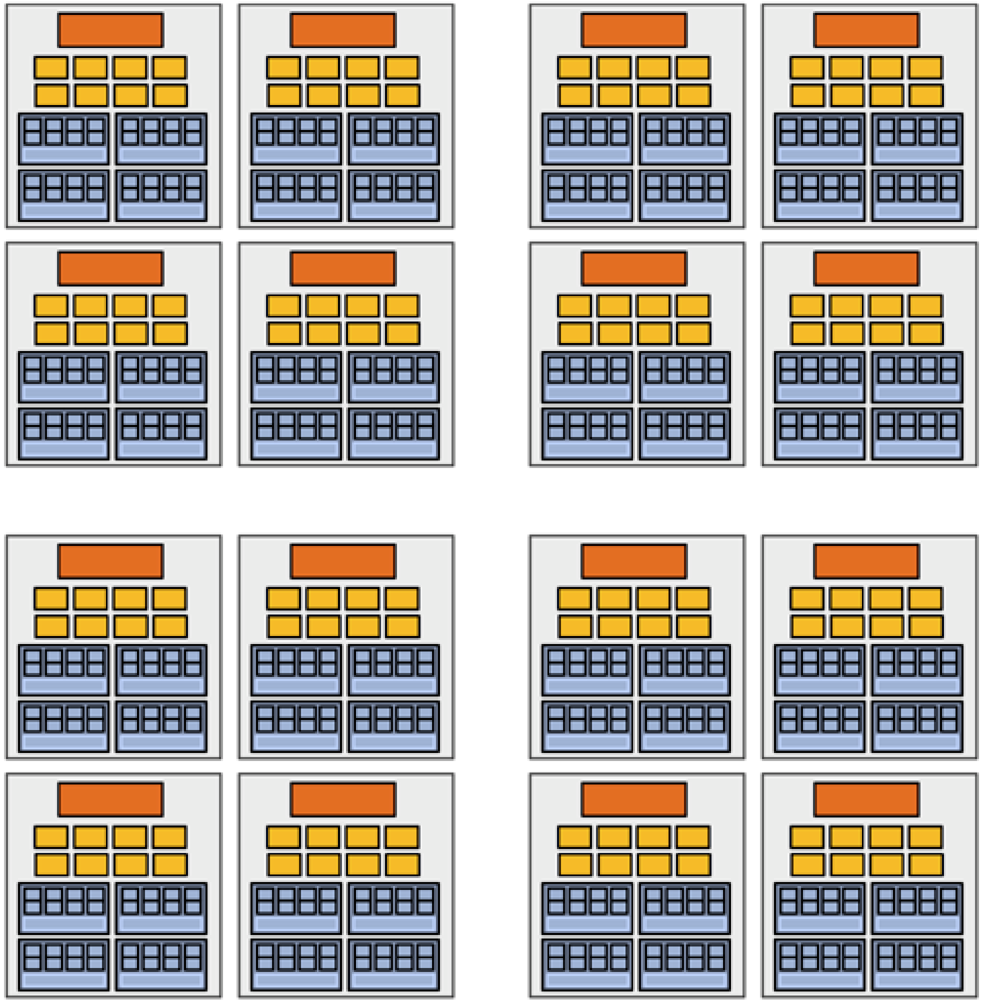
8 mul-add ALUs per core
(128 total)

16 simultaneous
instruction streams

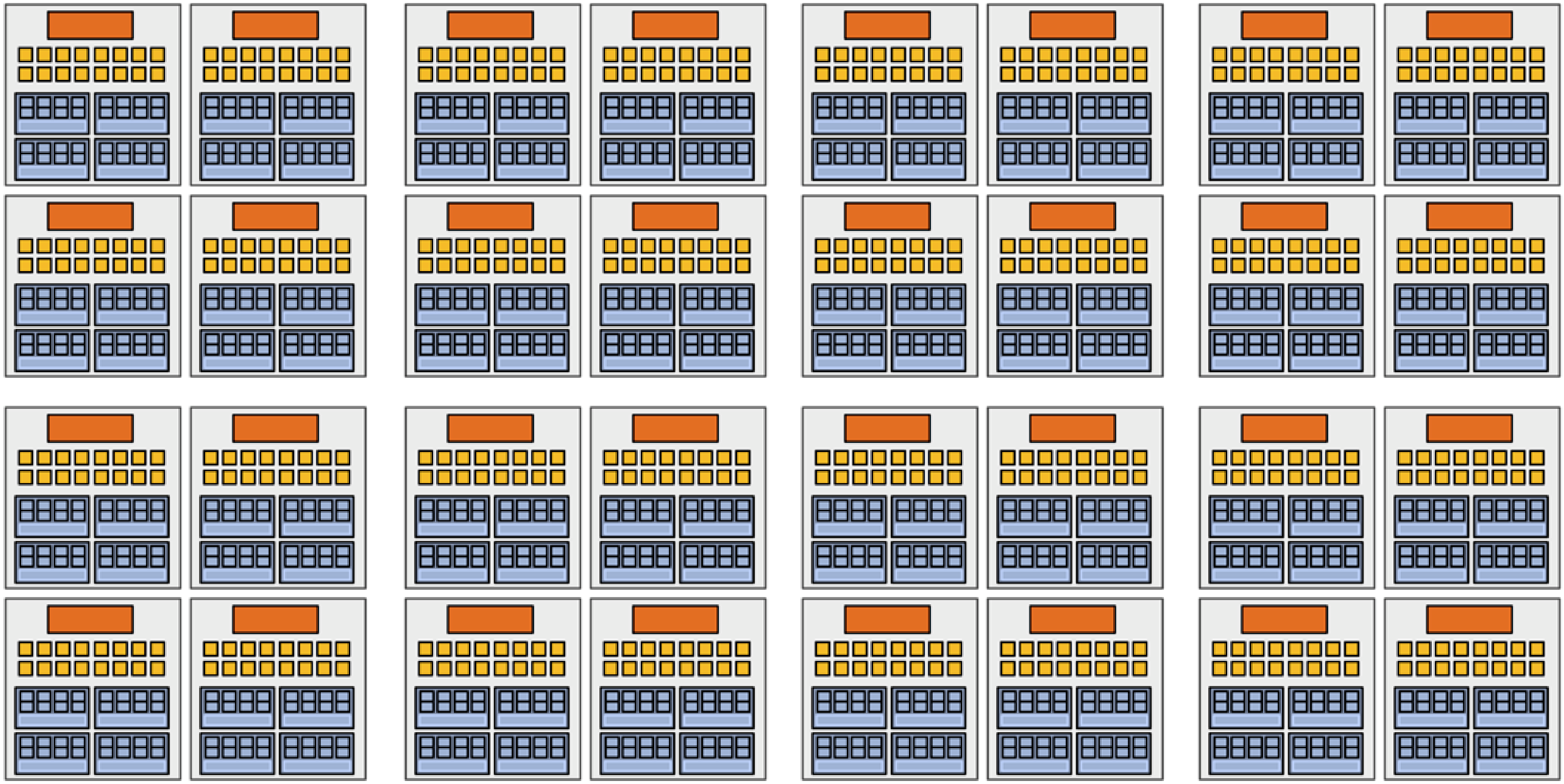
64 concurrent (but interleaved)
instruction streams

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 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

Thank you.