SIGGRAPH2009 NEW ORLEANS

From Shader Code to a Teraflop: How Shader Cores Work

Kayvon Fatahalian

Stanford University

This talk

1. Three major ideas that make GPU processing cores run fast

2. Closer look at real GPU designs
   - NVIDIA GTX 285
   - AMD Radeon 4890
   - Intel Larrabee

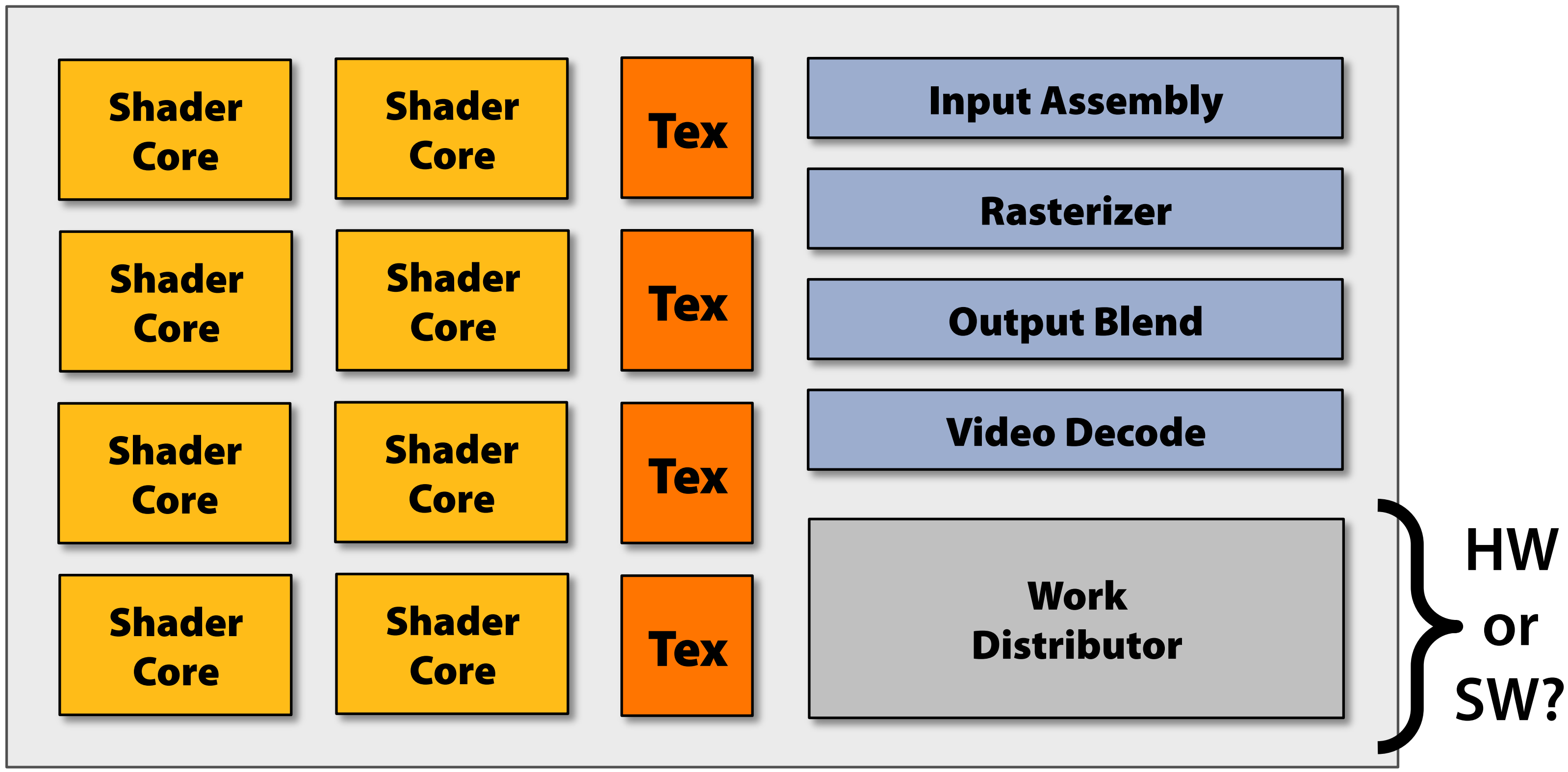
3. Memory hierarchy: moving data to processors

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 processing 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?



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

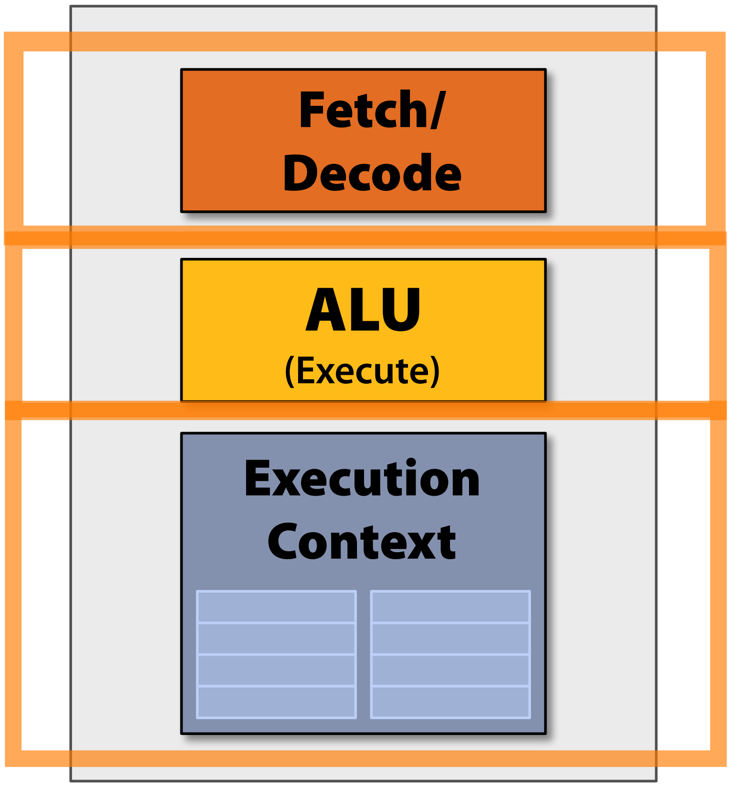
```
sampler mySamp;
Texture2D<float3> myTex;
float3 lightDir;

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)
```

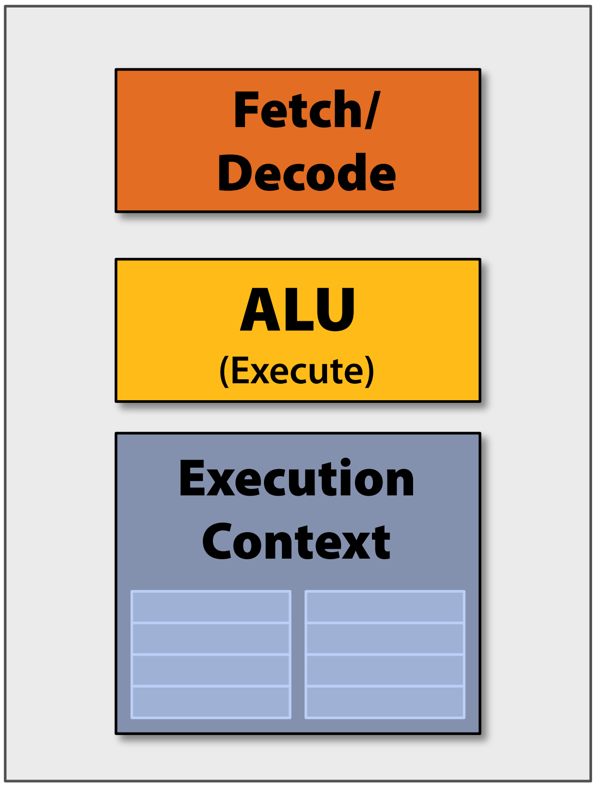
1 shaded fragment output record

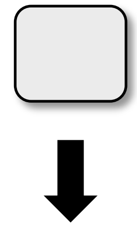
Execute shader



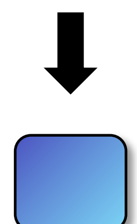

```
<diffuseShader>:
sample r0, v4, t0, s0
mul  r3, v0, cb0[0]
madd r3, v1, cb0[1], r3
madd r3, v2, cb0[2], r3
clmp r3, r3, l(0.0), l(1.0)
mul  o0, r0, r3
mul  o1, r1, r3
mul  o2, r2, r3
mov  o3, l(1.0)
```

Execute shader

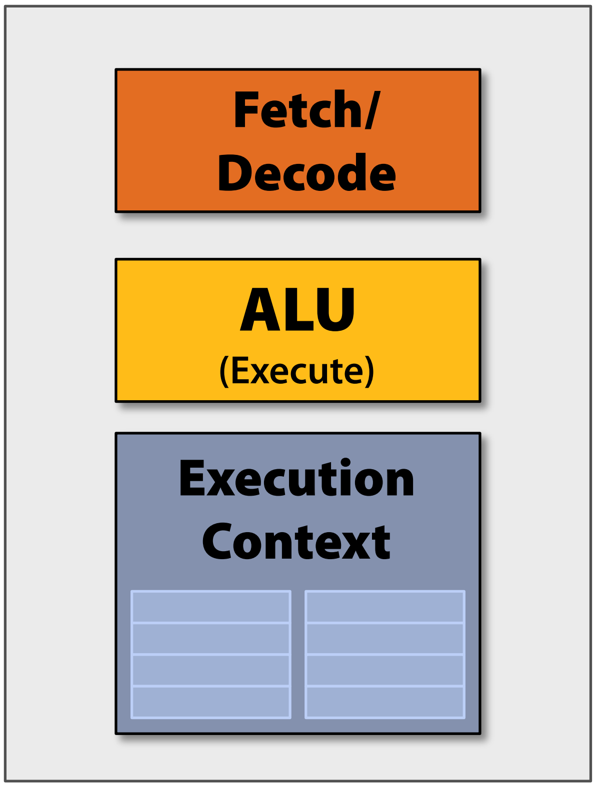


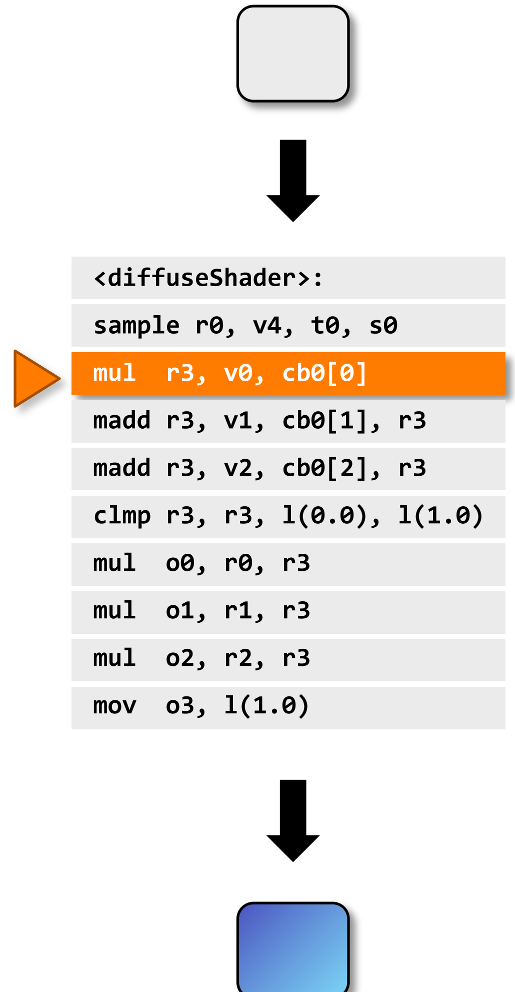


```
<diffuseShader>:
sample r0, v4, t0, s0
mul  r3, v0, cb0[0]
madd r3, v1, cb0[1], r3
madd r3, v2, cb0[2], r3
clmp r3, r3, l(0.0), l(1.0)
mul  o0, r0, r3
mul  o1, r1, r3
mul  o2, r2, r3
mov  o3, l(1.0)
```

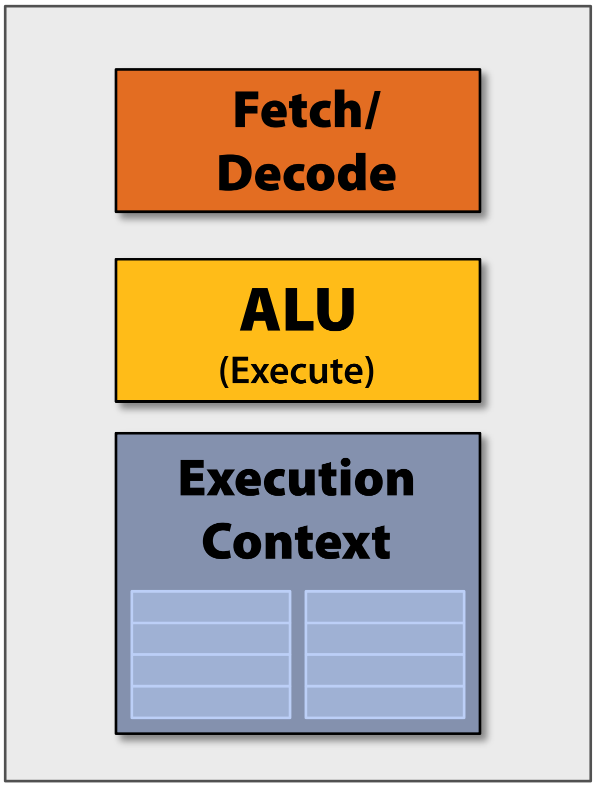


Execute shader



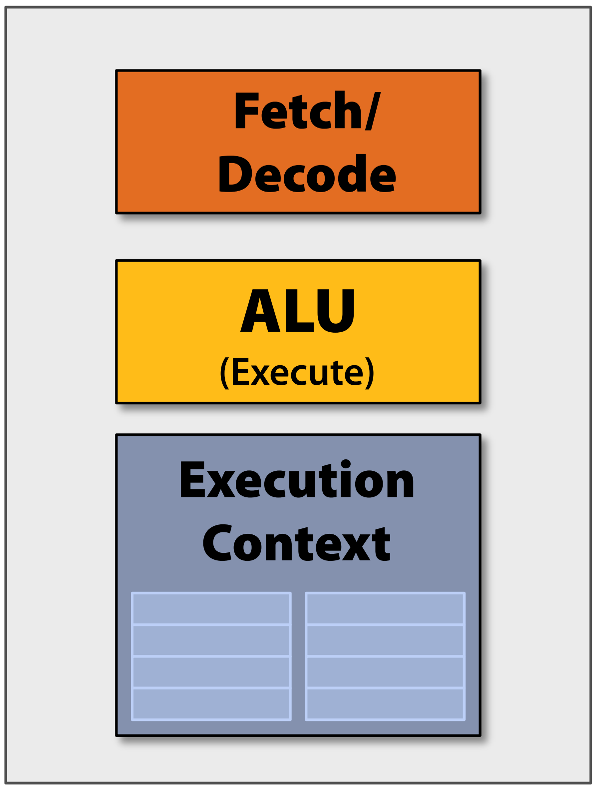


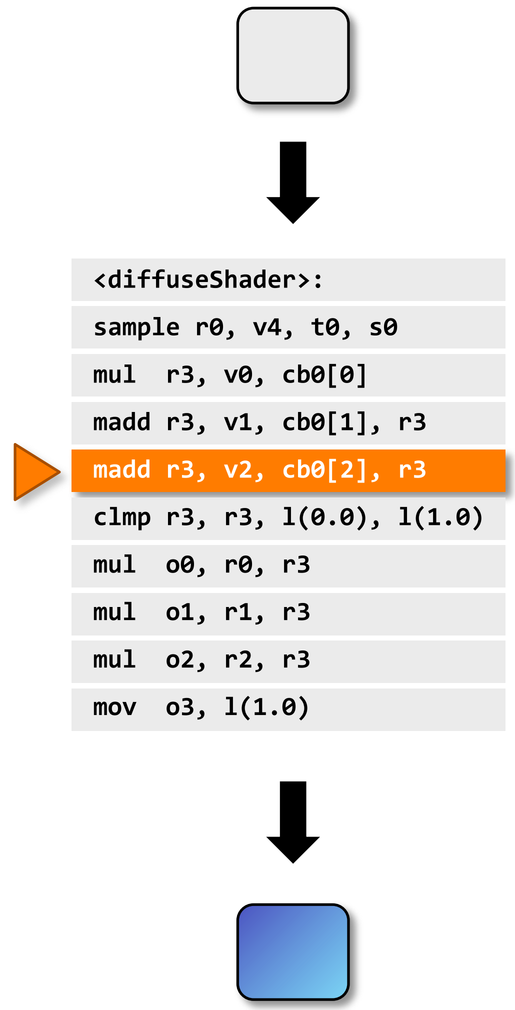
Execute shader




```
<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)
```

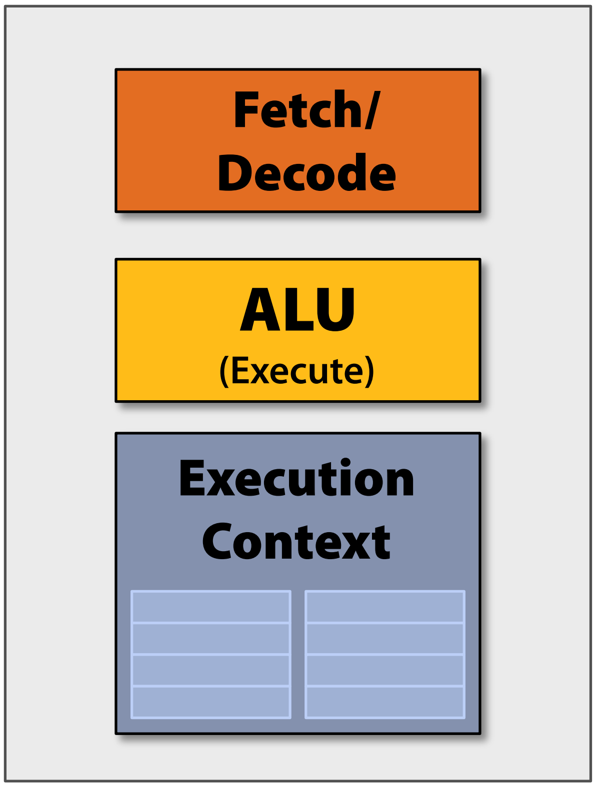
Execute shader

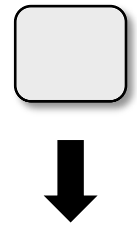




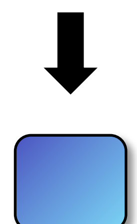
```
<diffuseShader>:
sample r0, v4, t0, s0
mul  r3, v0, cb0[0]
madd r3, v1, cb0[1], r3
madd r3, v2, cb0[2], r3
clmp r3, r3, l(0.0), l(1.0)
mul  o0, r0, r3
mul  o1, r1, r3
mul  o2, r2, r3
mov  o3, l(1.0)
```

Execute shader

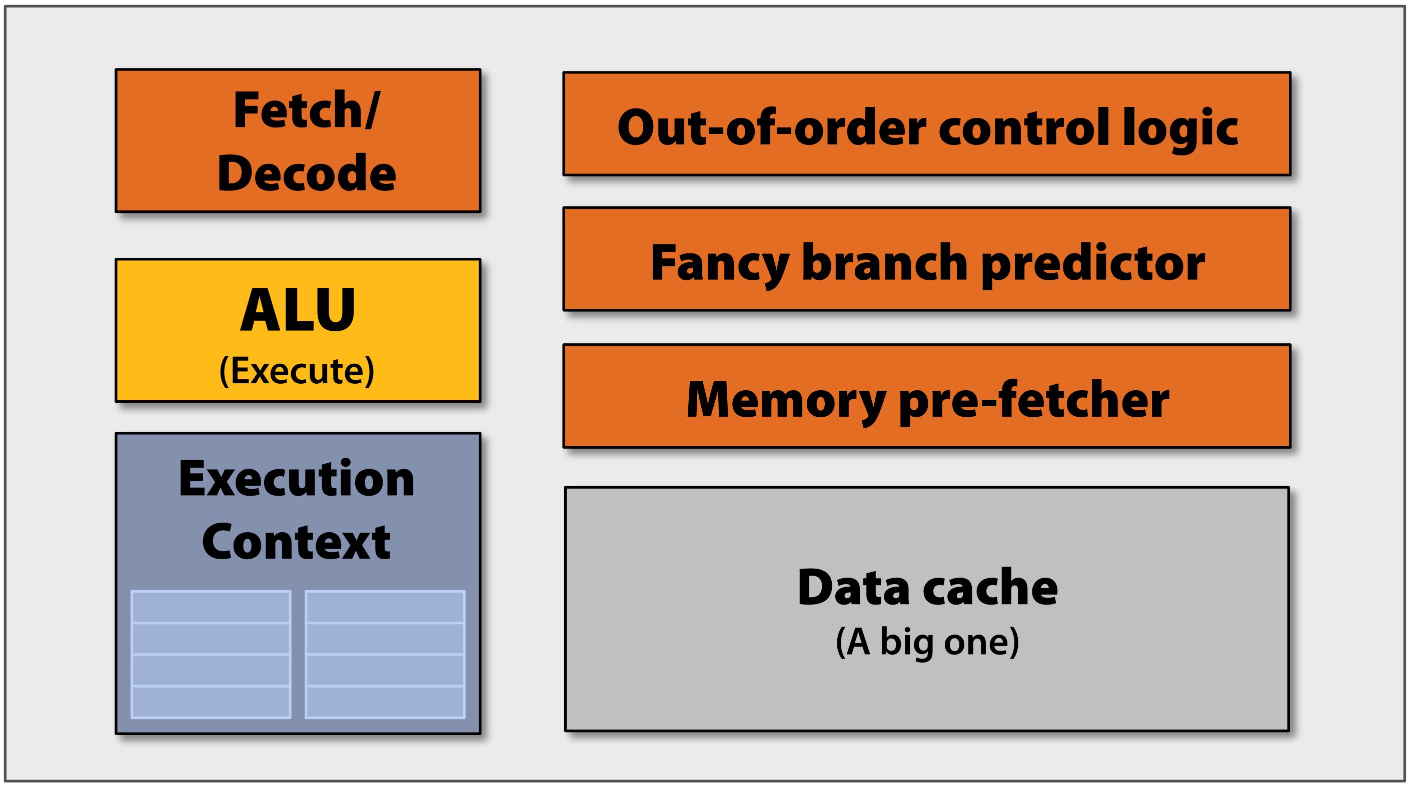




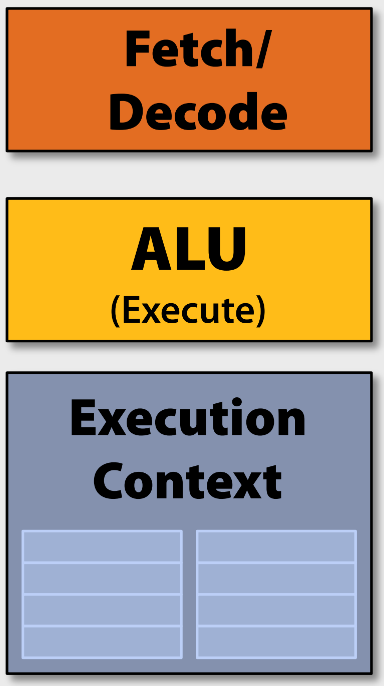
```
<diffuseShader>:
sample r0, v4, t0, s0
mul  r3, v0, cb0[0]
madd r3, v1, cb0[1], r3
madd r3, v2, cb0[2], r3
clmp r3, r3, l(0.0), l(1.0)
mul  o0, r0, r3
mul  o1, r1, r3
mul  o2, r2, r3
mov  o3, l(1.0)
```



CPU-"style" cores



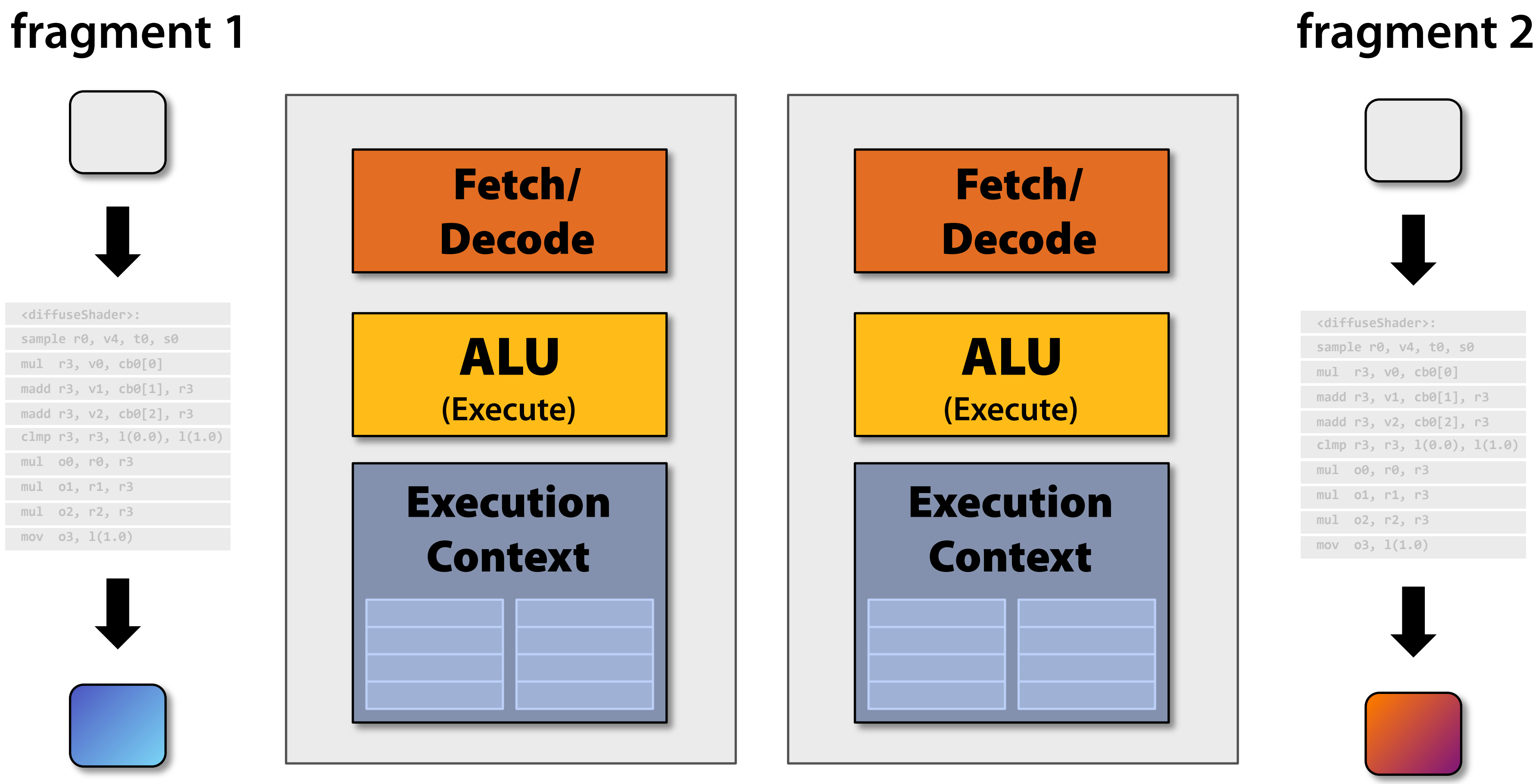
Slimming down



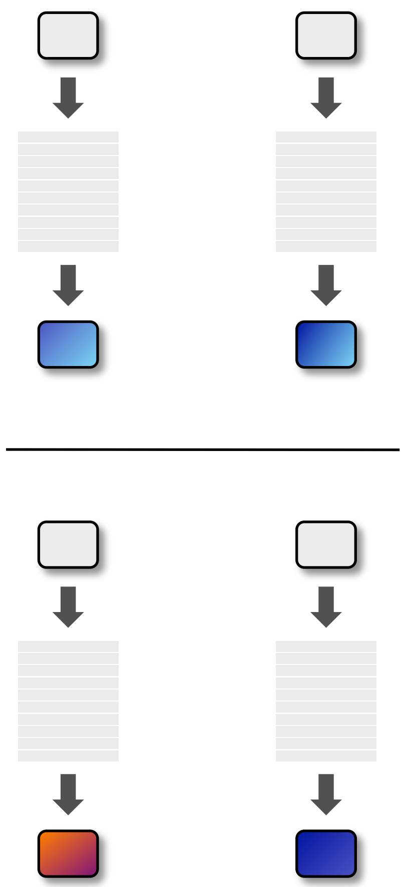
Idea #1:

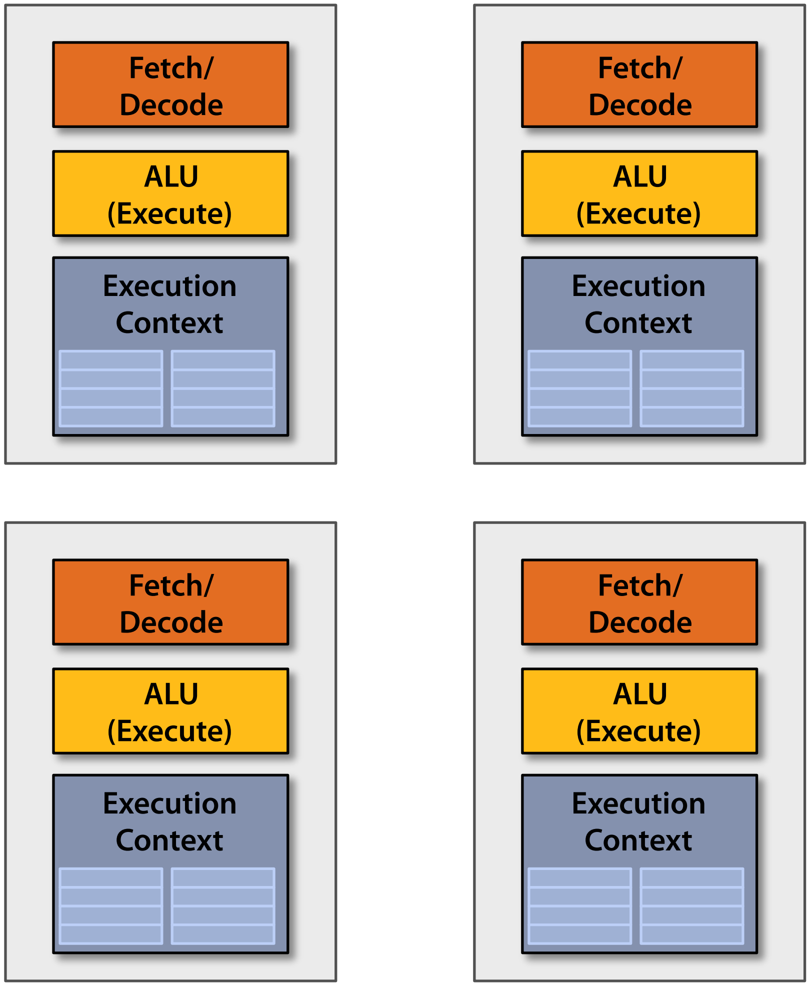
Remove components that help a single instruction stream run fast

Two cores (two fragments in parallel)

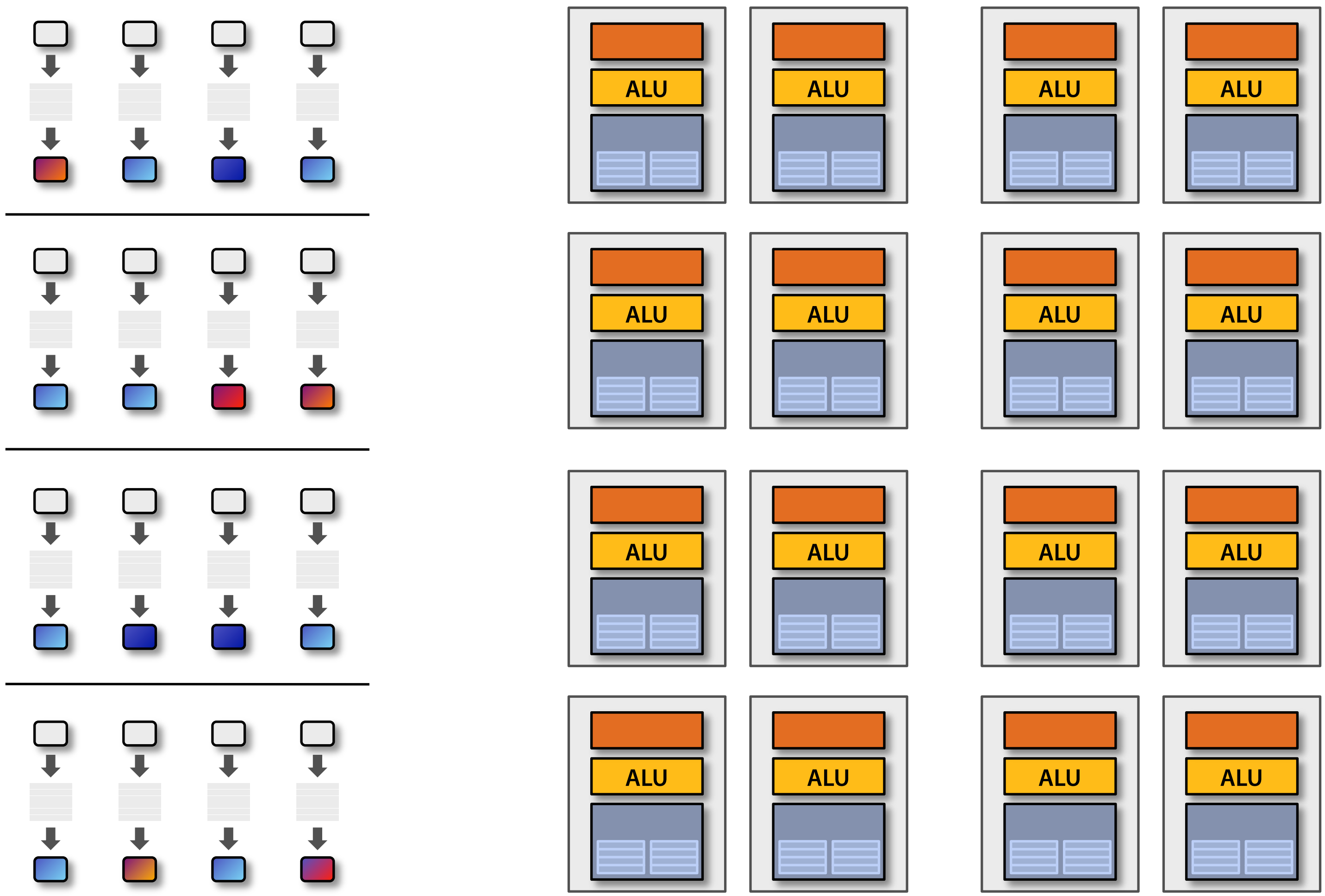


Four cores (four fragments in parallel)



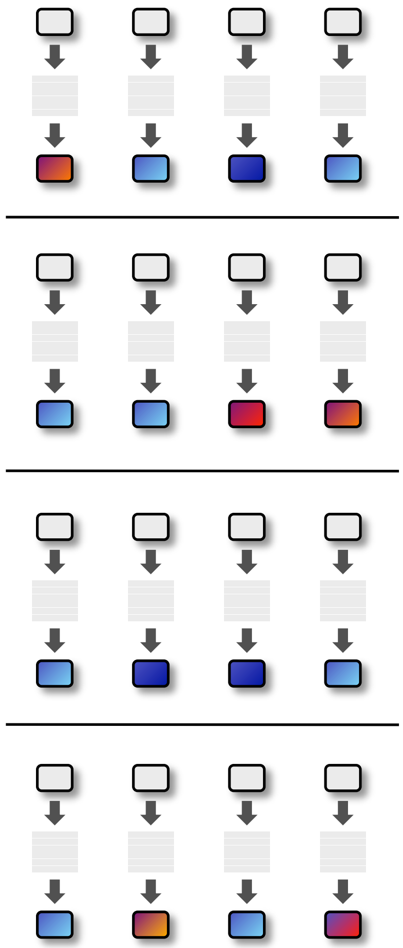


Sixteen cores (sixteen fragments in parallel)



16 cores = 16 simultaneous instruction streams

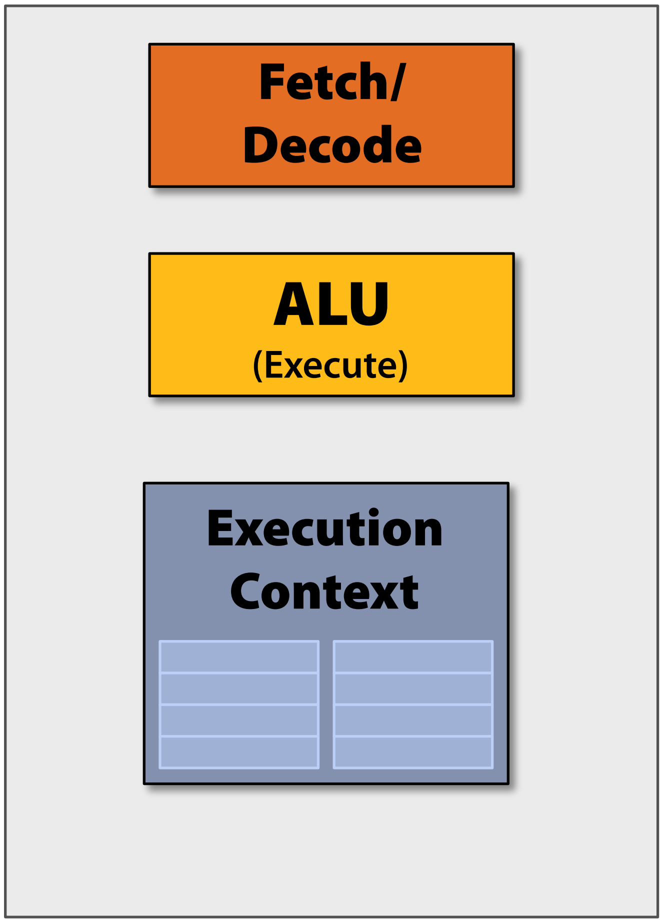
Instruction stream sharing



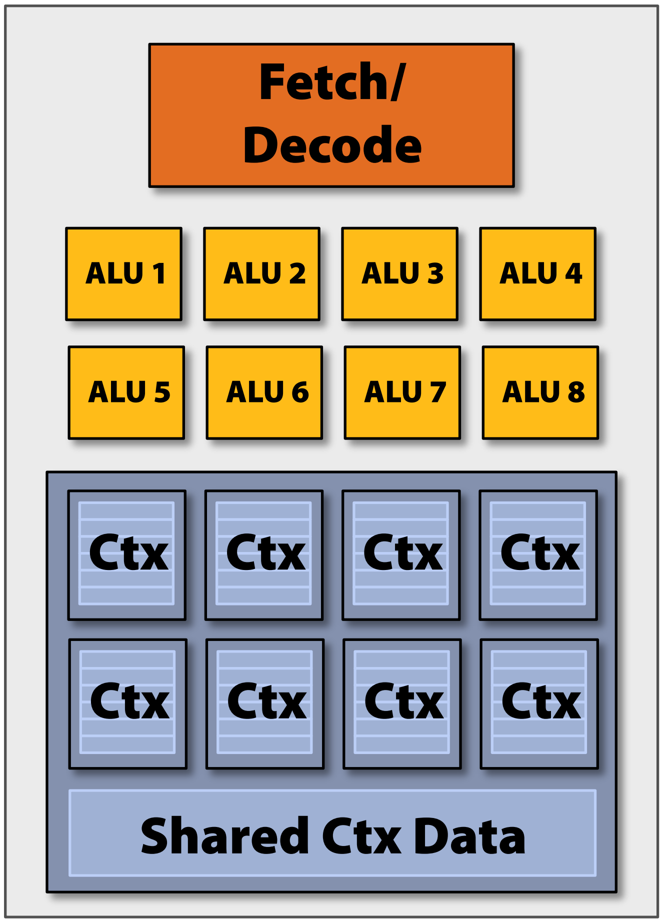
But… many fragments *should* be able to share an instruction stream!

```
<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

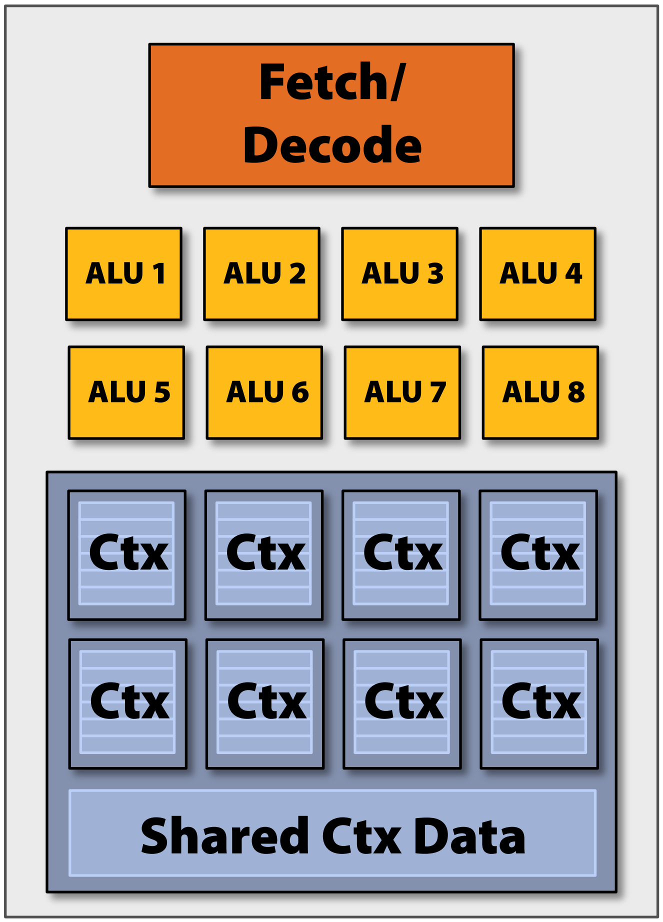


Idea #2:

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

SIMD processing

Modifying the shader

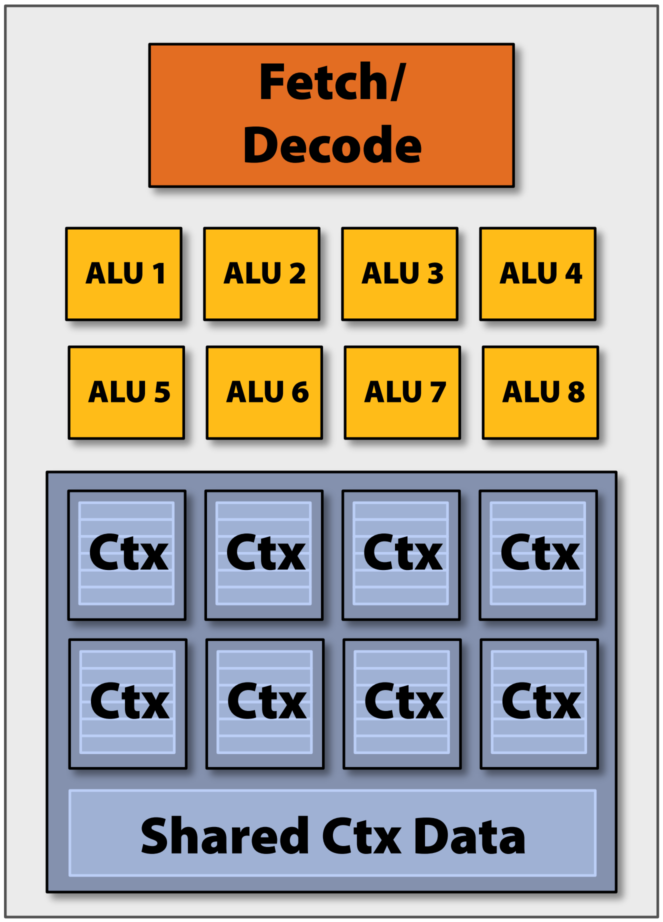



```
<diffuseShader>:
sample r0, v4, t0, s0
mul  r3, v0, cb0[0]
madd r3, v1, cb0[1], r3
madd r3, v2, cb0[2], r3
clmp r3, r3, l(0.0), l(1.0)
mul  o0, r0, r3
mul  o1, r1, r3
mul  o2, r2, r3
mov  o3, l(1.0)
```

Original compiled shader:

Processes one fragment using scalar ops on scalar registers

Modifying the shader

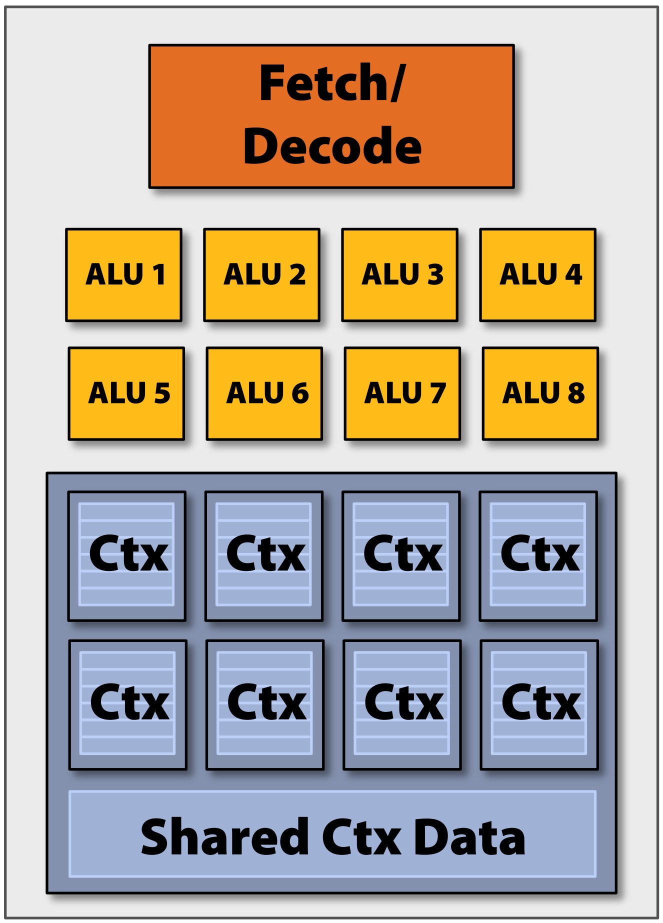


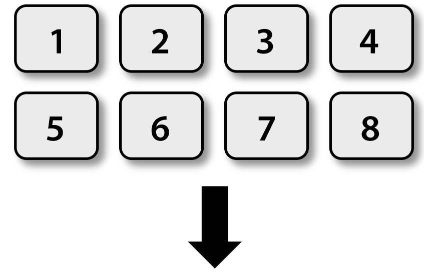

```
<VEC8_diffuseShader>:
VEC8_sample vec_r0, vec_v4, t0, vec_s0
VEC8_mul  vec_r3, vec_v0, cb0[0]
VEC8_madd vec_r3, vec_v1, cb0[1], vec_r3
VEC8_madd vec_r3, vec_v2, cb0[2], vec_r3
VEC8_clmp vec_r3, vec_r3, l(0.0), l(1.0)
VEC8_mul  vec_o0, vec_r0, vec_r3
VEC8_mul  vec_o1, vec_r1, vec_r3
VEC8_mul  vec_o2,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

New compiled shader:

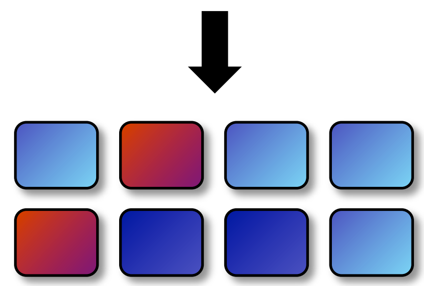
Processes 8 fragments using vector ops on vector registers

Modifying the shader

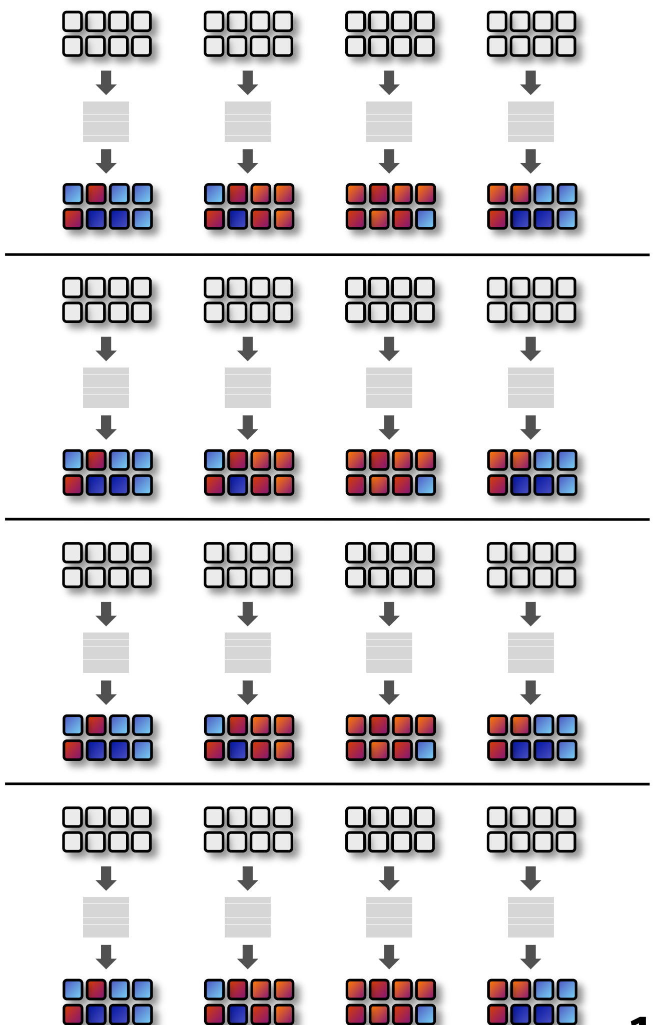


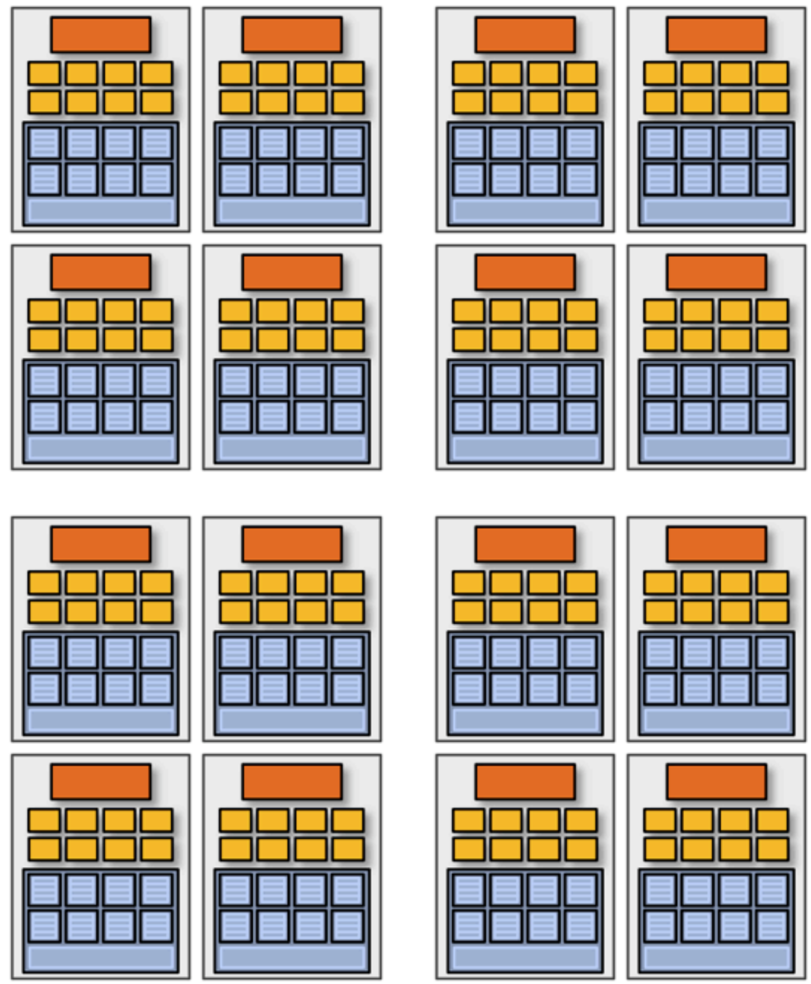



```
<VEC8_diffuseShader>:
VEC8_sample vec_r0, vec_v4, t0, vec_s0
VEC8_mul  vec_r3, vec_v0, cb0[0]
VEC8_madd vec_r3, vec_v1, cb0[1], vec_r3
VEC8_madd vec_r3, vec_v2, cb0[2], vec_r3
VEC8_clmp vec_r3, vec_r3, l(0.0), l(1.0)
VEC8_mul  vec_o0, vec_r0, vec_r3
VEC8_mul  vec_o1, vec_r1, vec_r3
VEC8_mul  vec_o2,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



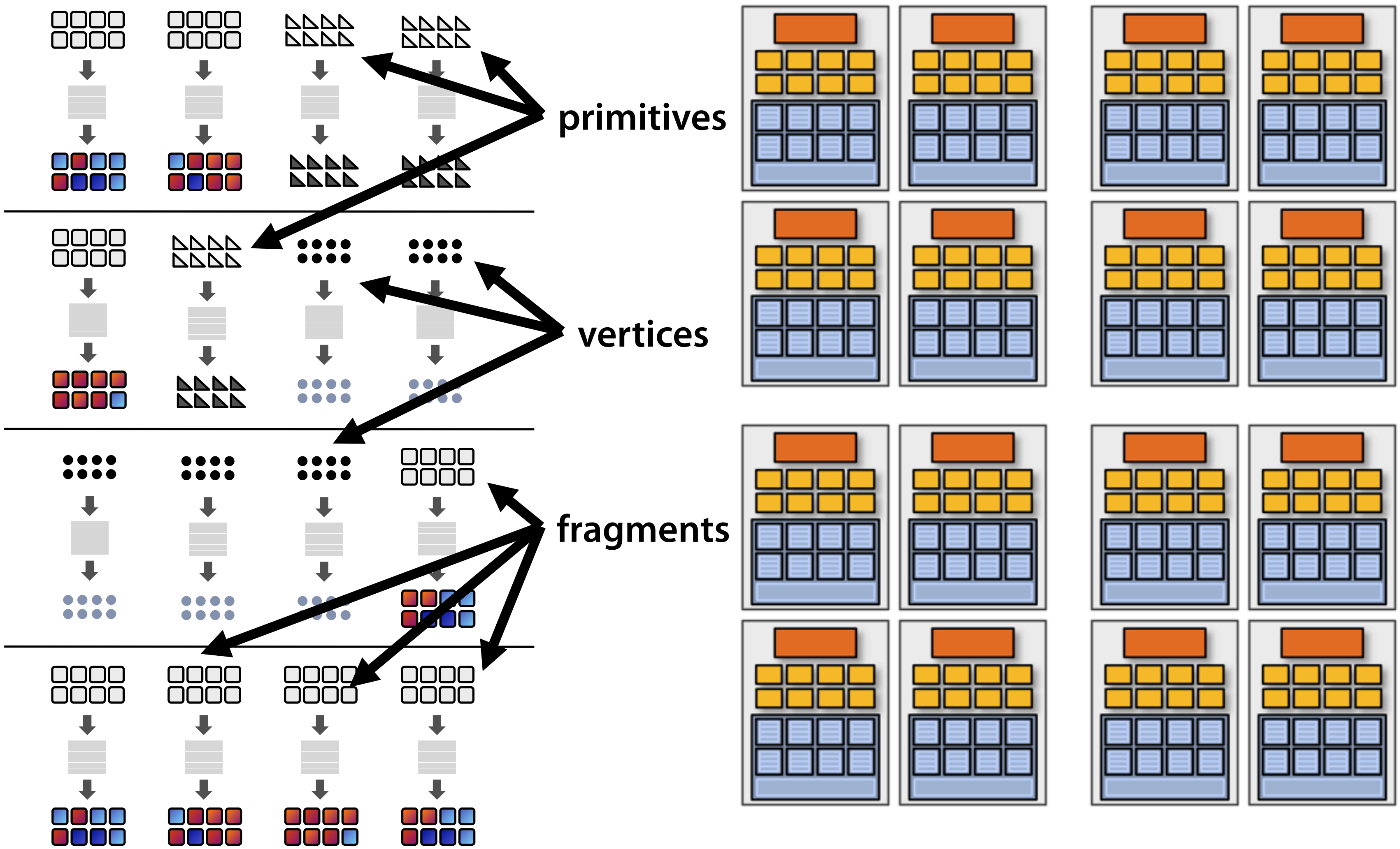
128 fragments in parallel



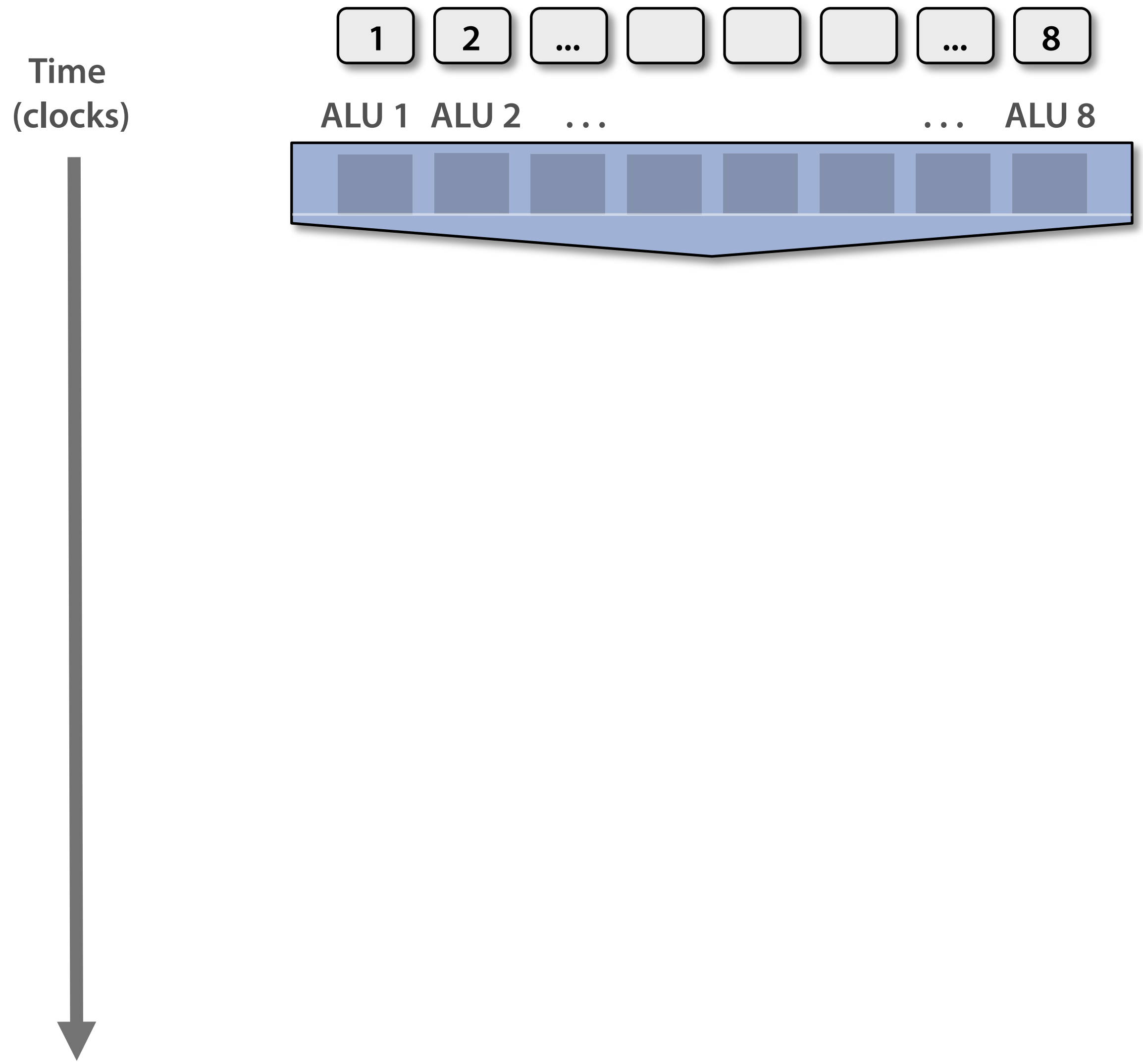


16 cores = 128 ALUs
= 16 simultaneous instruction streams

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



But what about branches?

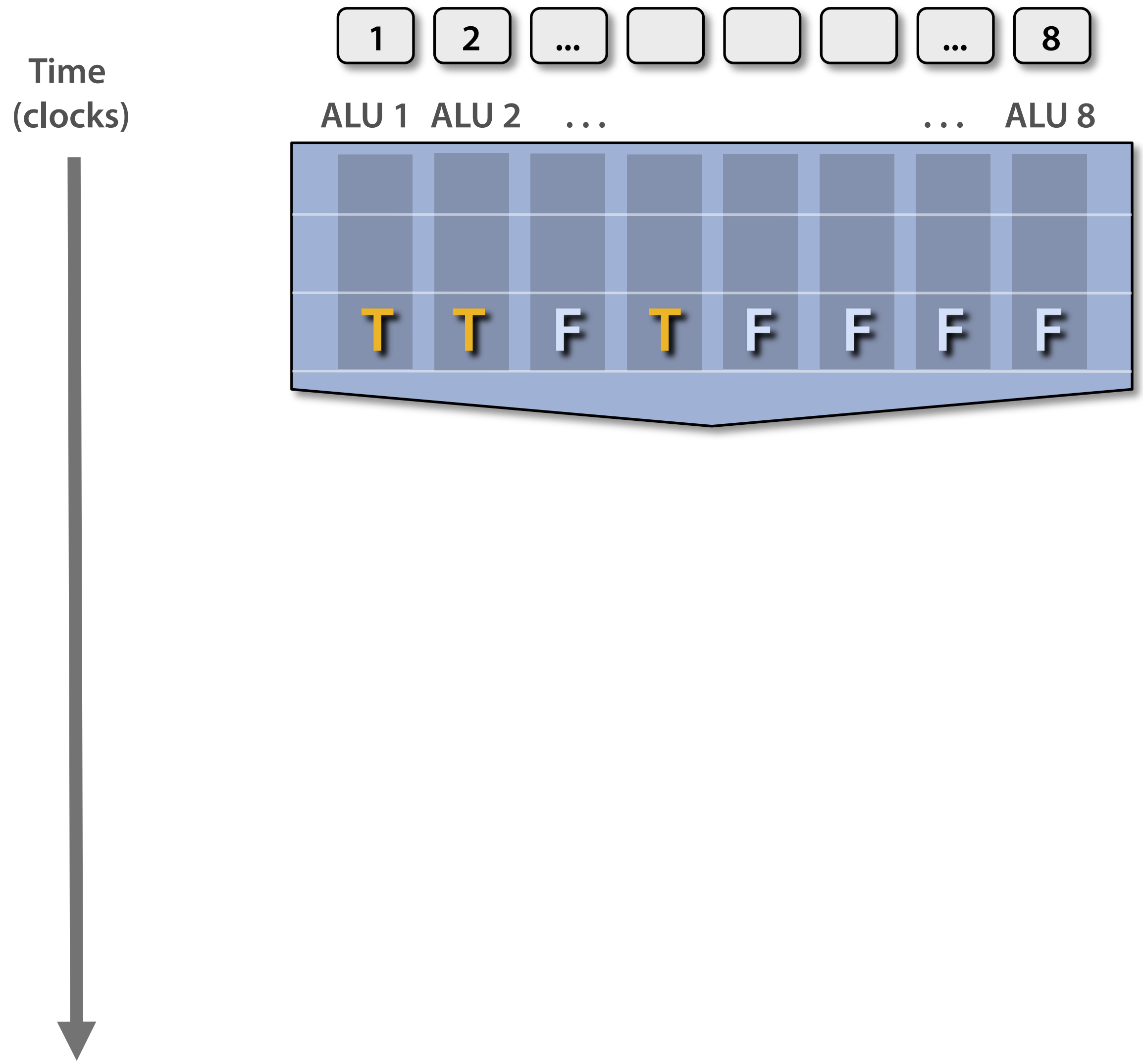



```
<unconditional
shader code>

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?

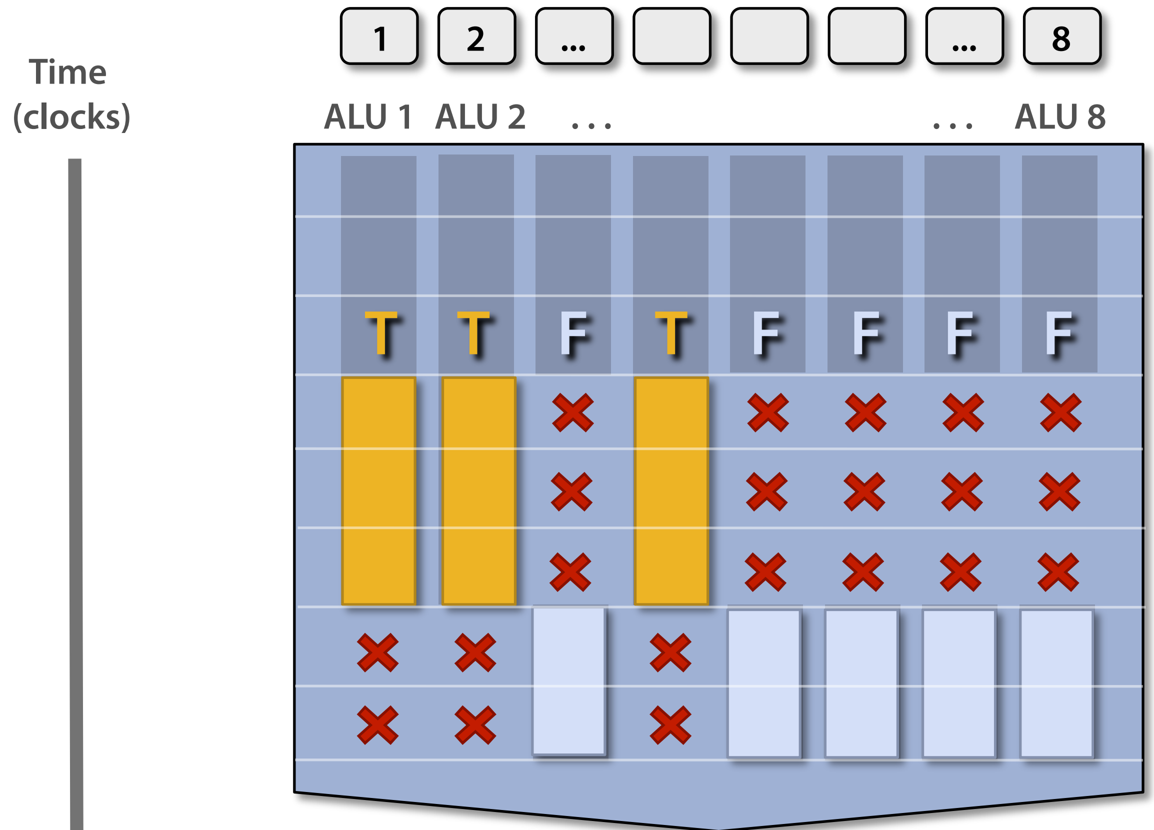



```
<unconditional
shader code>

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?



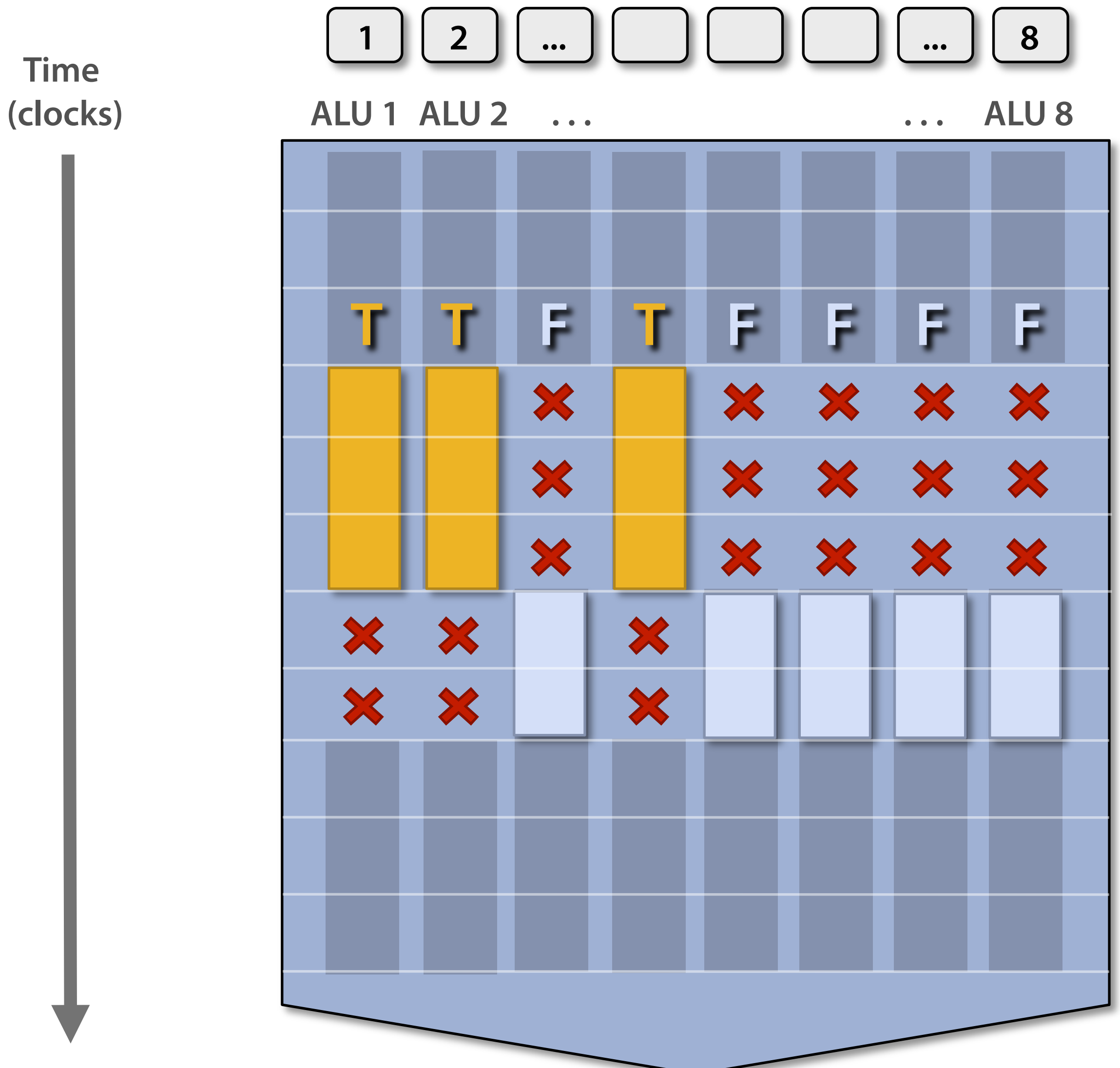

```
<unconditional
shader code>

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>
```

Clarification

SIMD processing does not imply SIMD instructions

- Option 1: Explicit vector instructions
  - Intel/AMD 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), AMD Radeon architectures

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

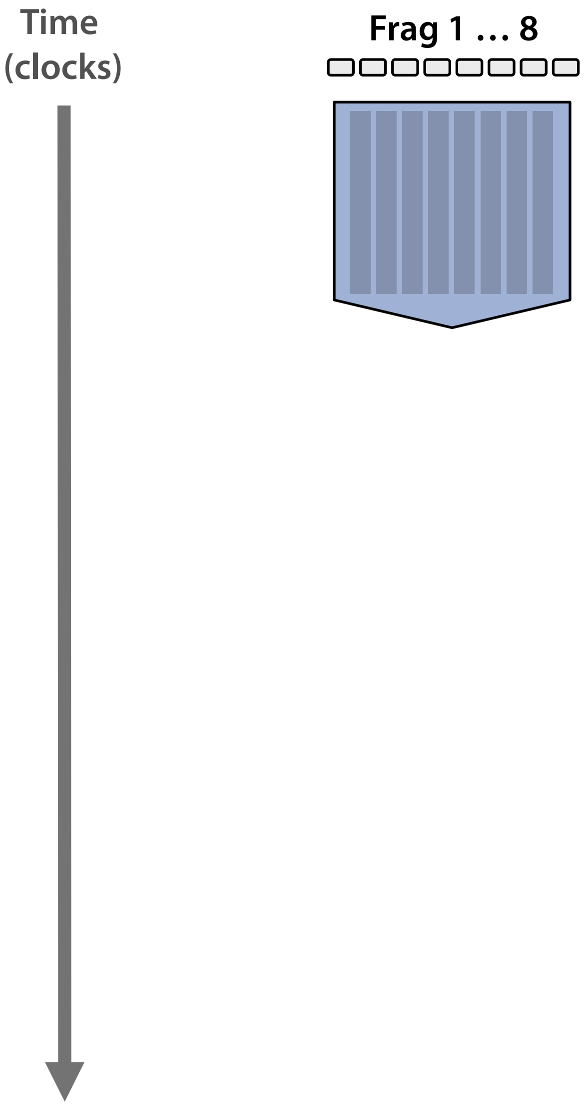
We've removed the fancy caches and logic that helps avoid stalls.

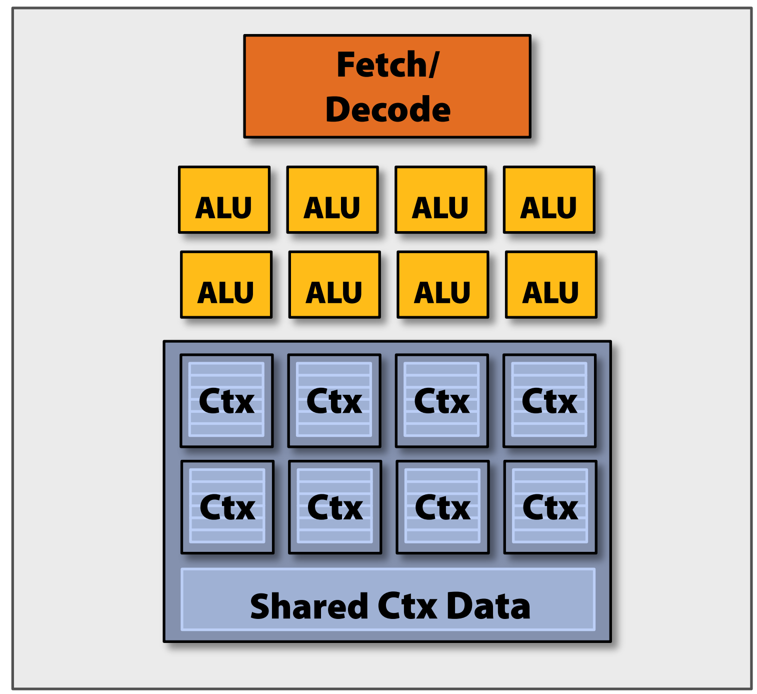
But we have **LOTS** of independent fragments.

Idea #3:

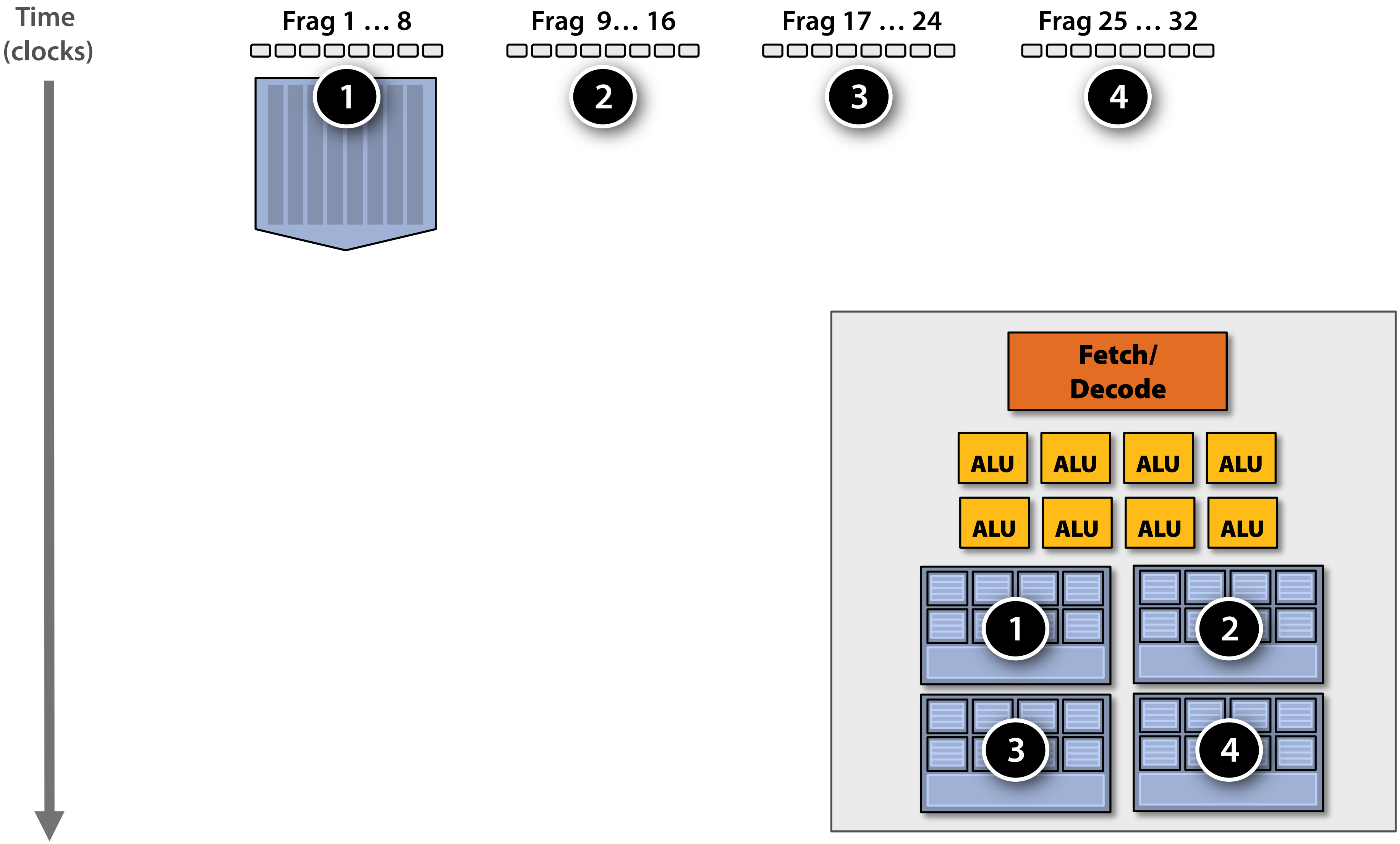
Interleave processing of many fragments on a single core to avoid stalls caused by high latency operations.

Hiding shader stalls

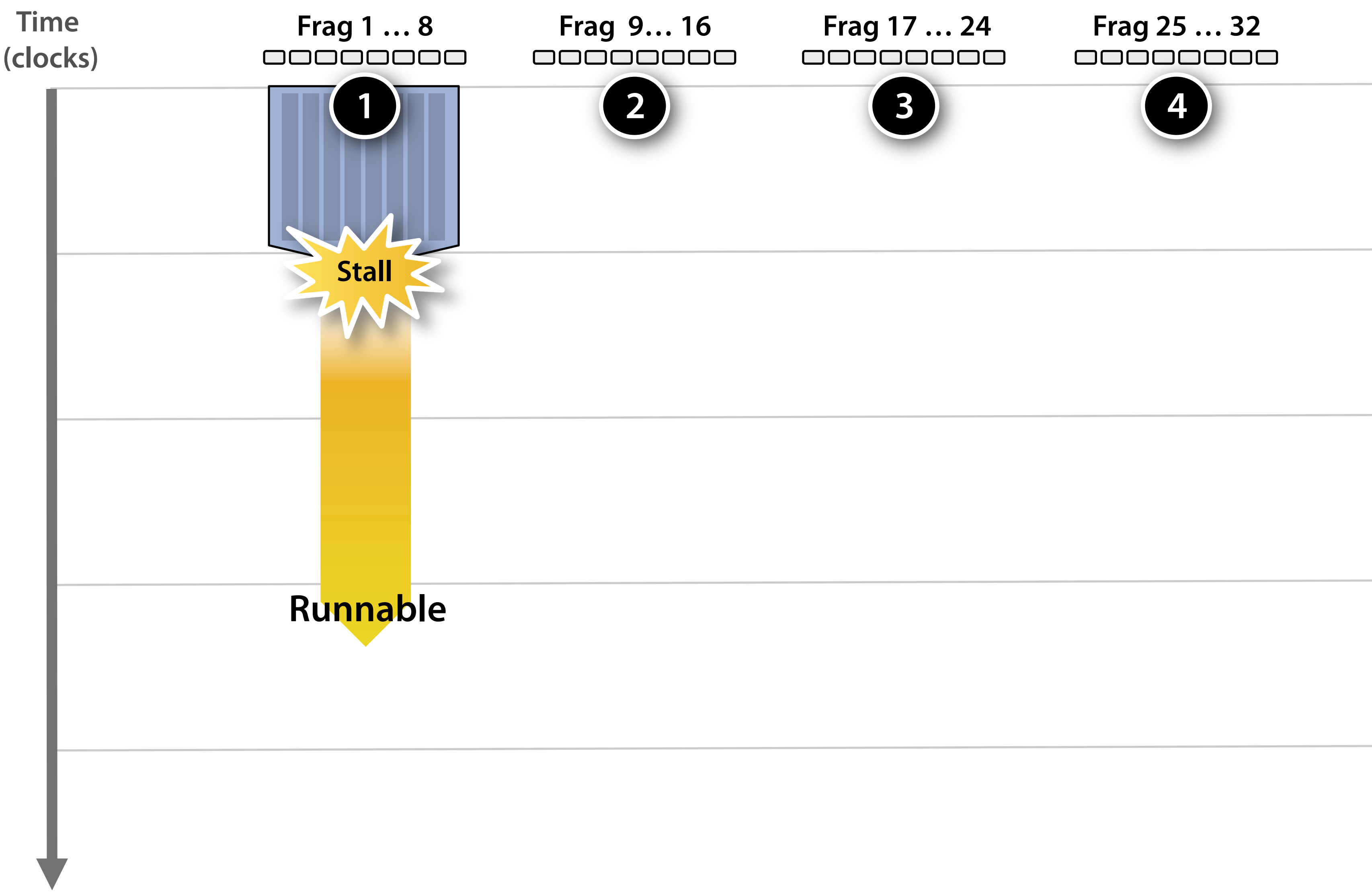




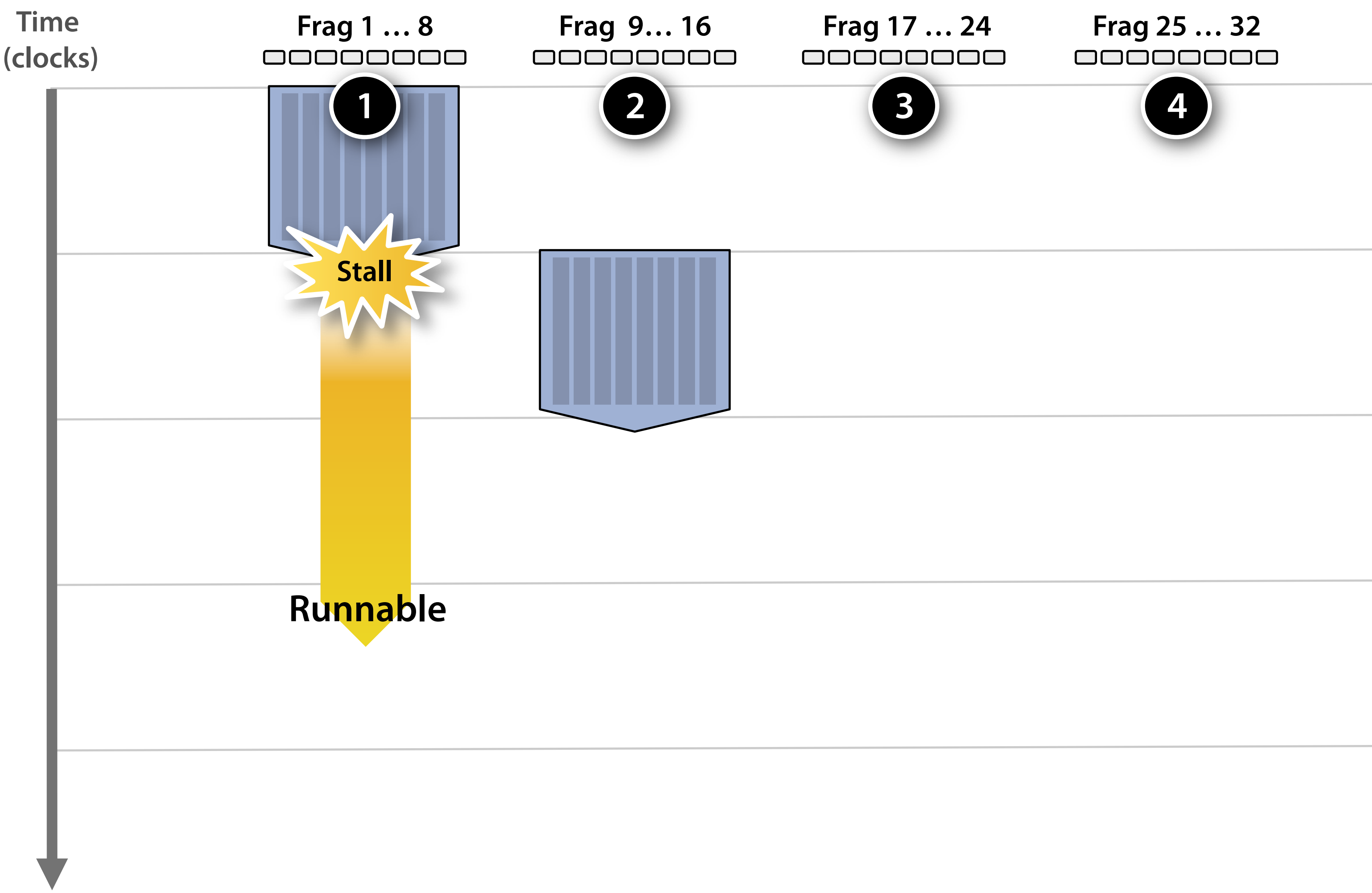
Hiding shader stalls



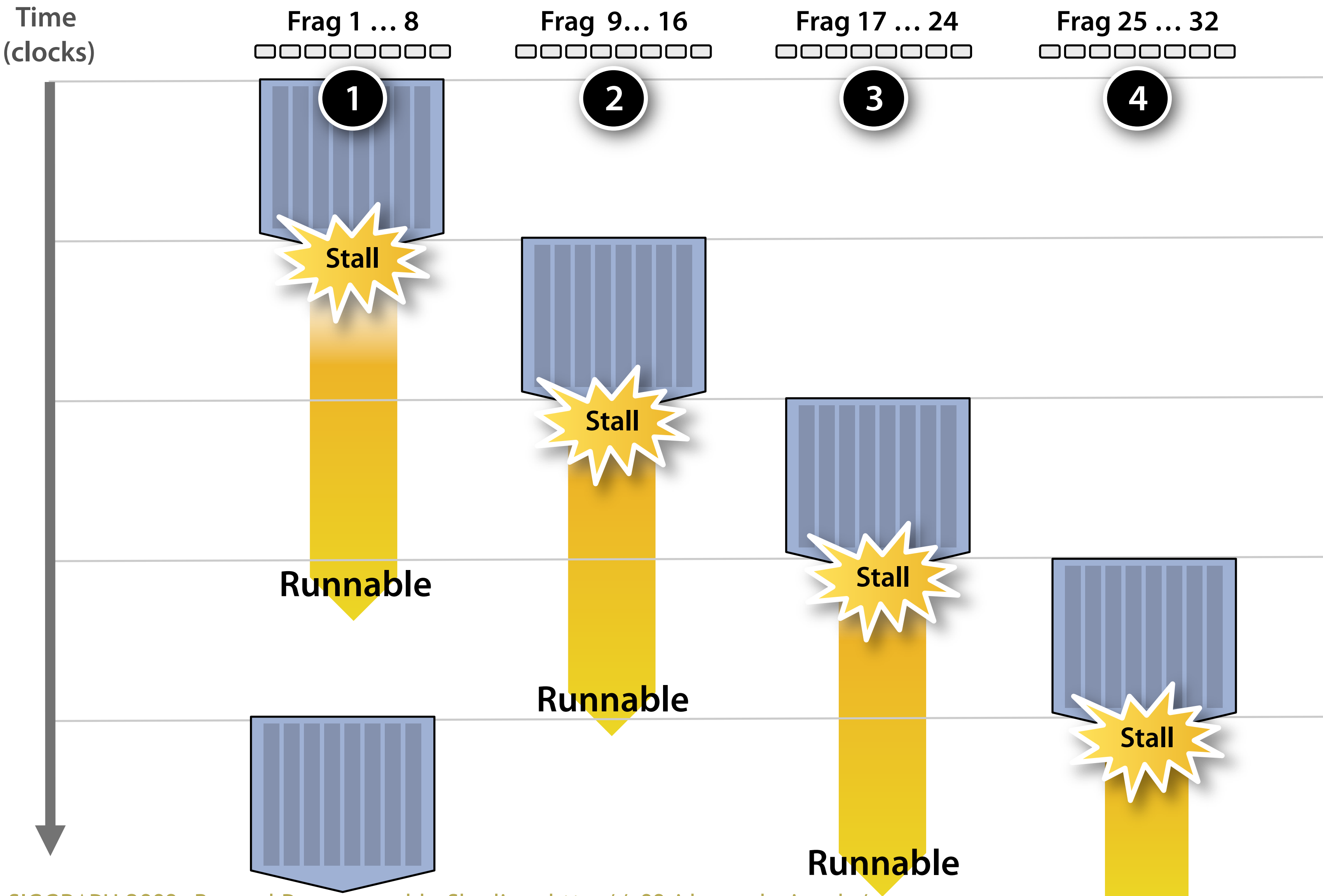
Hiding shader stalls



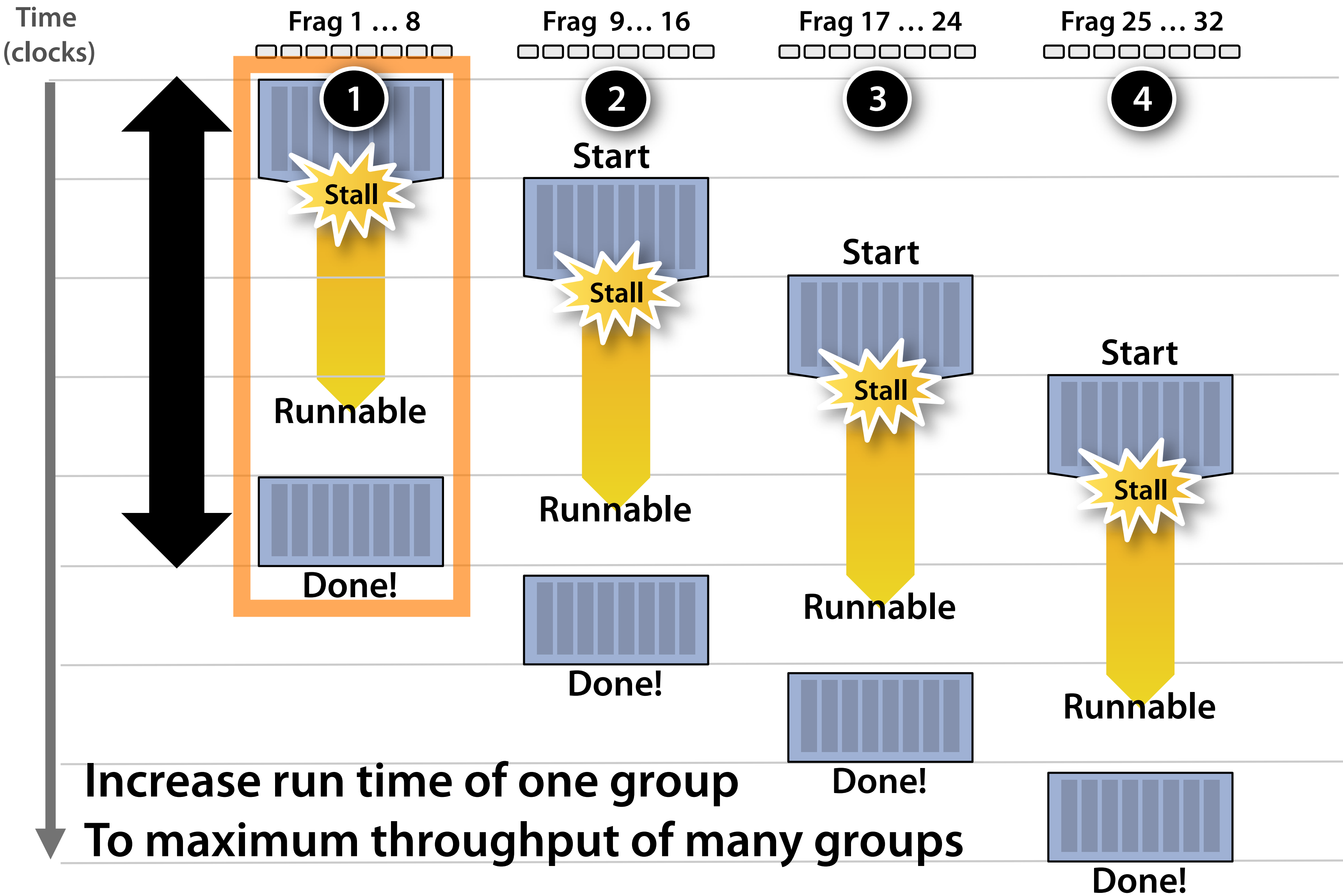
Hiding shader stalls



Hiding shader stalls

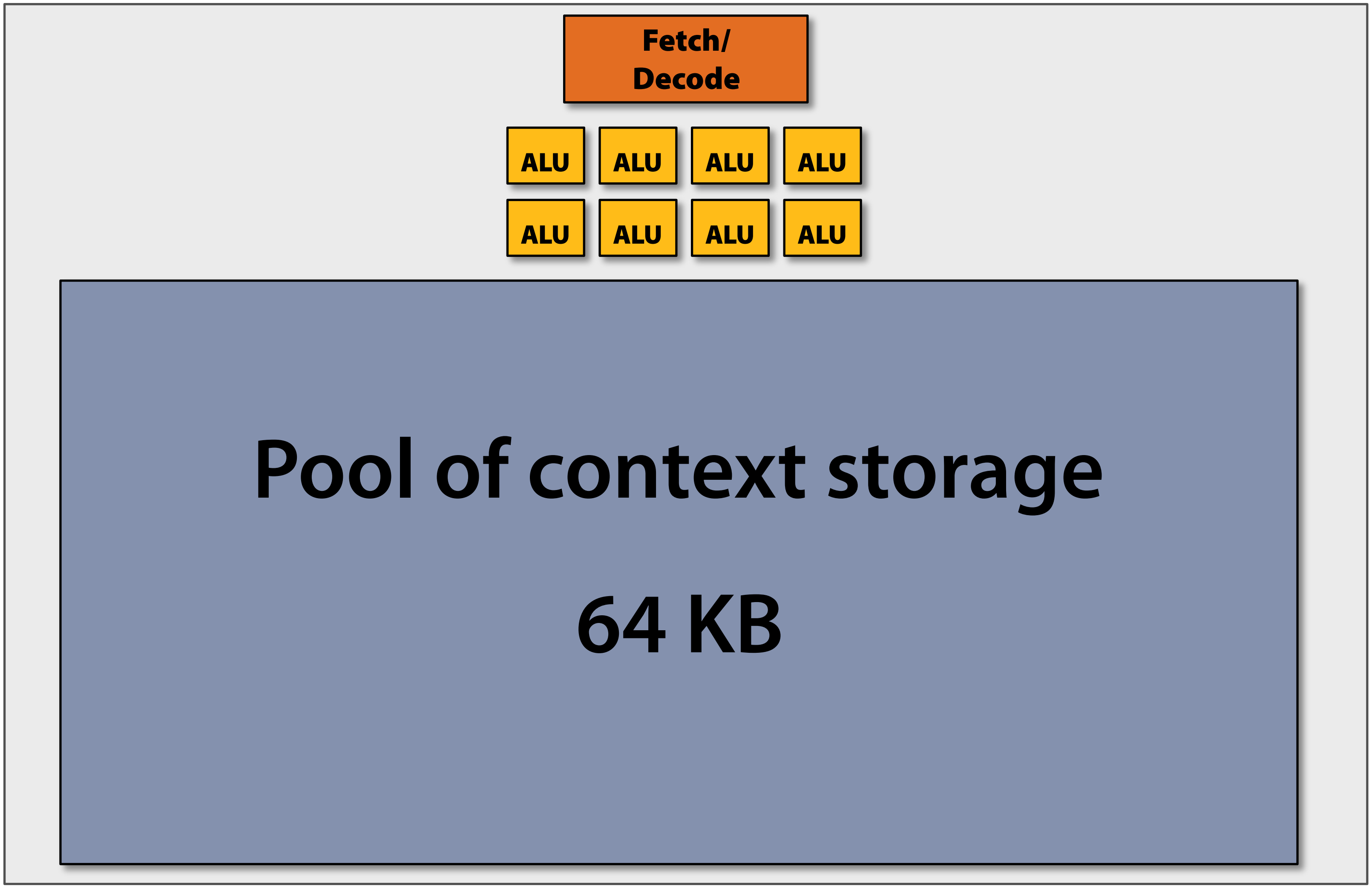


Throughput!



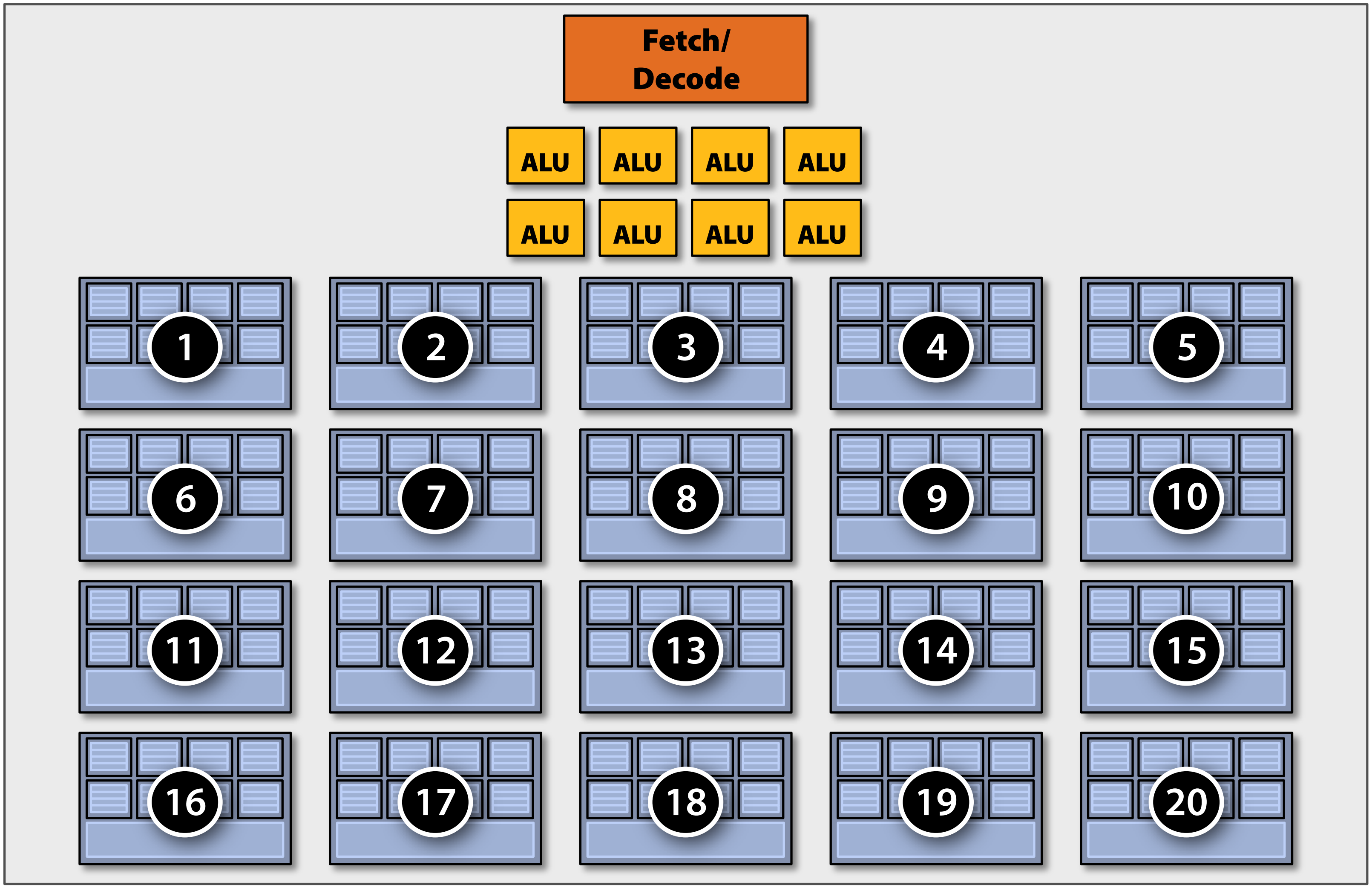
Increase run time of one group
To maximum throughput of many groups

Storing contexts

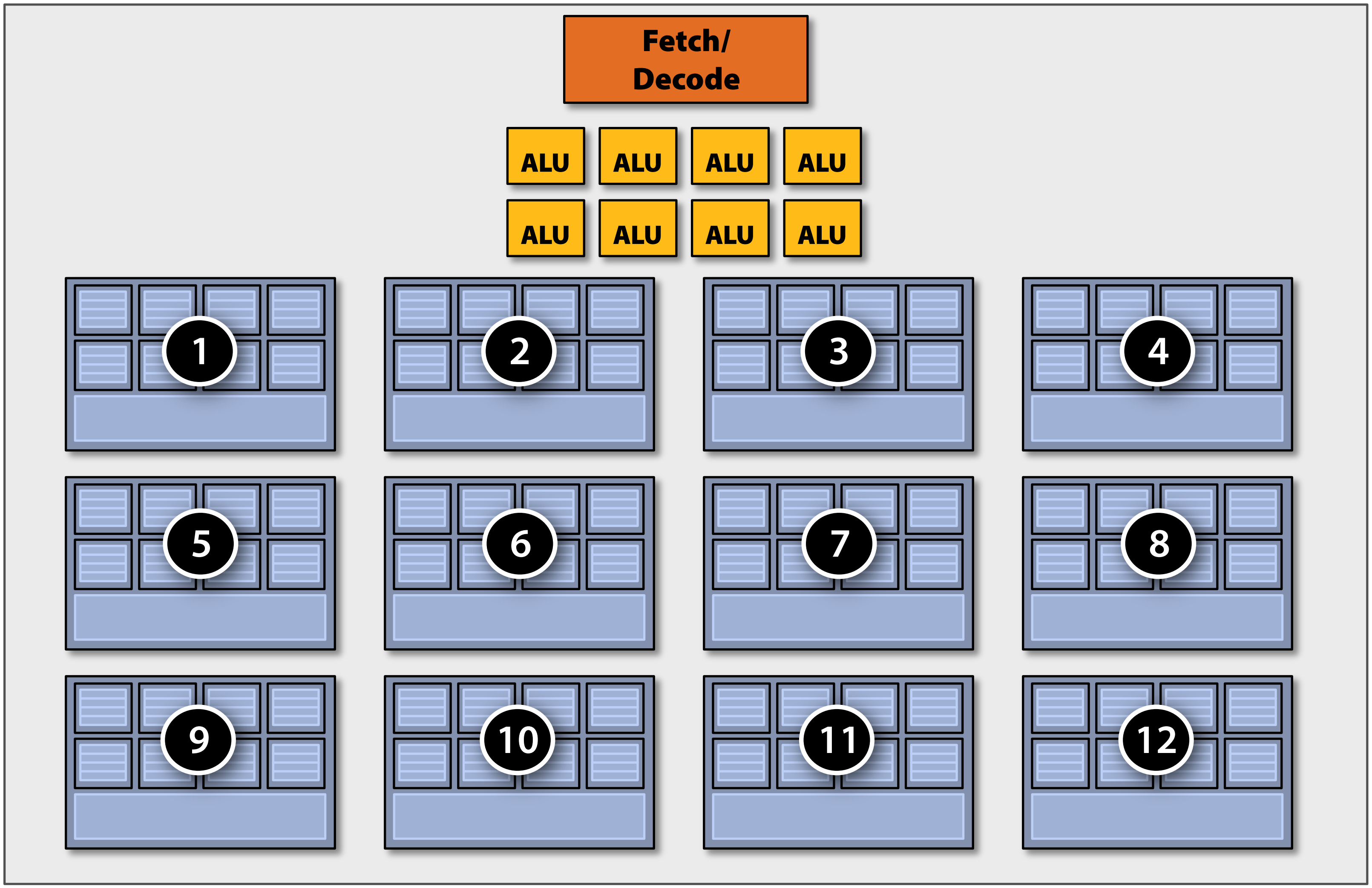


Twenty small contexts

(maximal latency hiding ability)

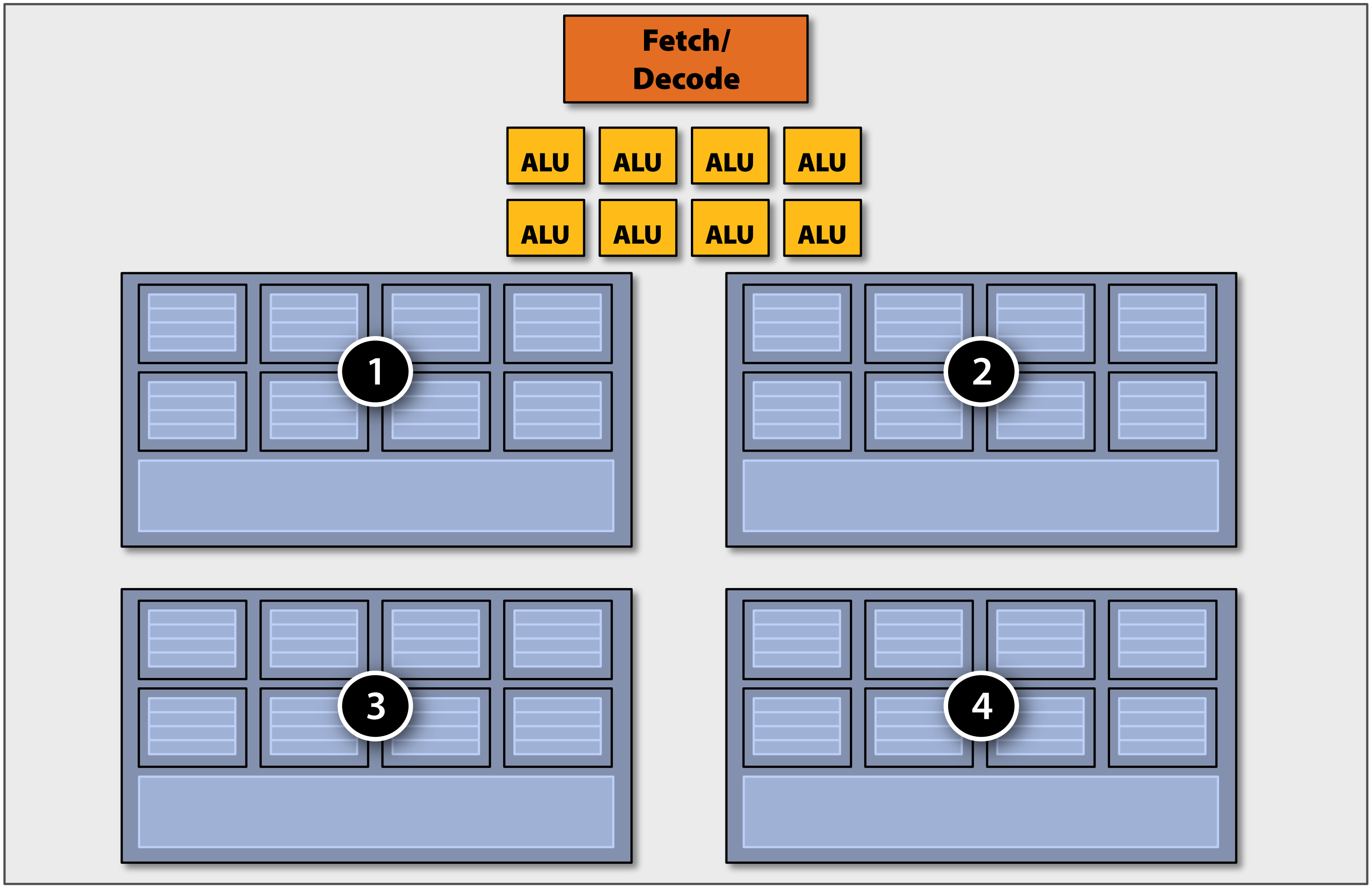


Twelve medium contexts



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 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

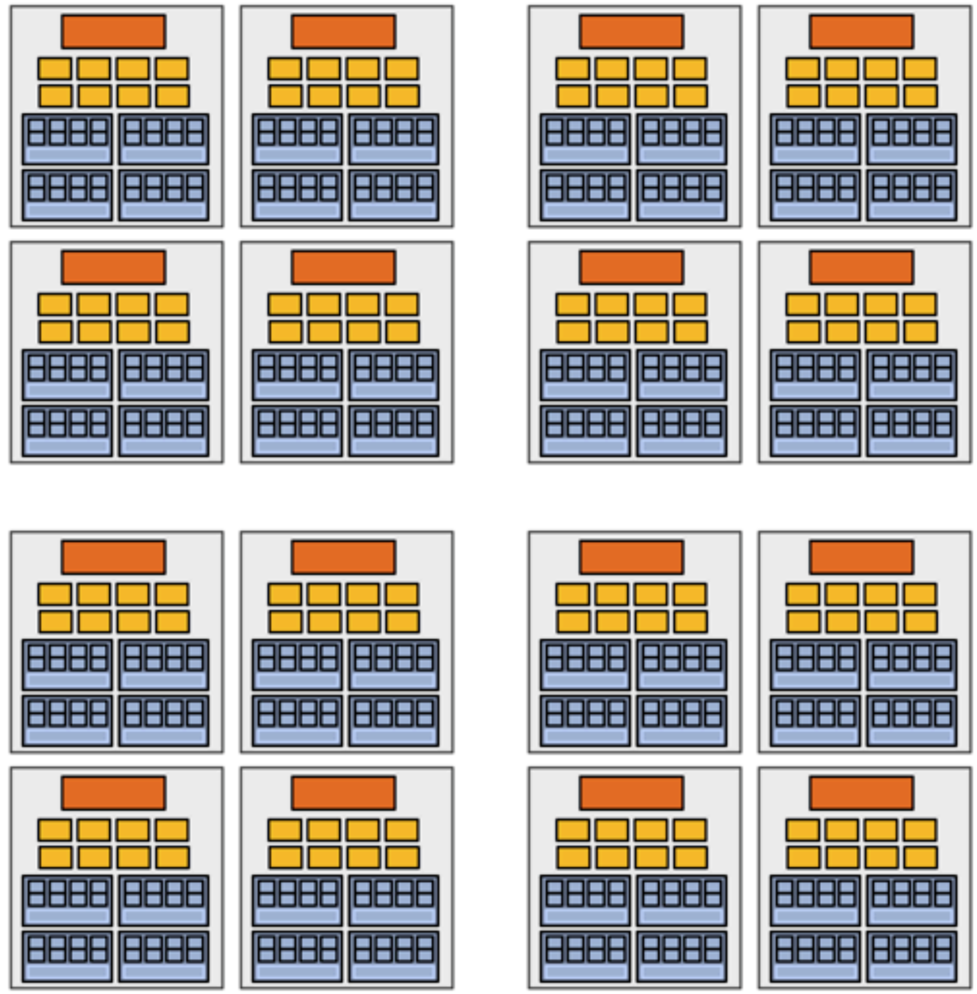
8 mul-add ALUs per core
(128 total)

16 simultaneous
instruction streams

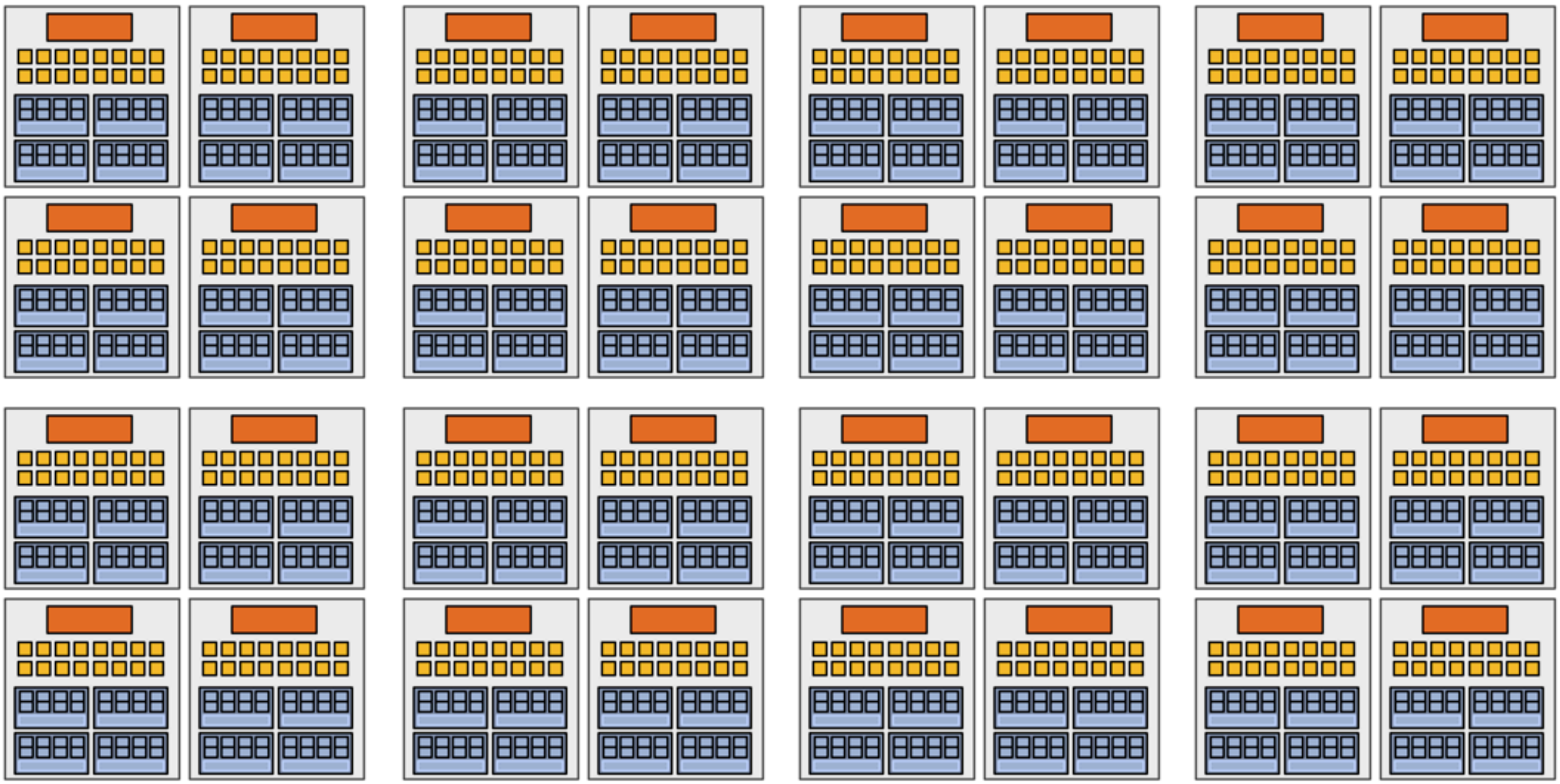
64 concurrent (but interleaved)
instruction streams

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
   – When one group stalls, work on another group

Part 2:

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

NVIDIA GeForce GTX 285
AMD Radeon HD 4890
Intel Larrabee (as proposed)

Disclaimer

- The following slides describe “how one can think” about the architecture of NVIDIA, AMD, and Intel GPUs

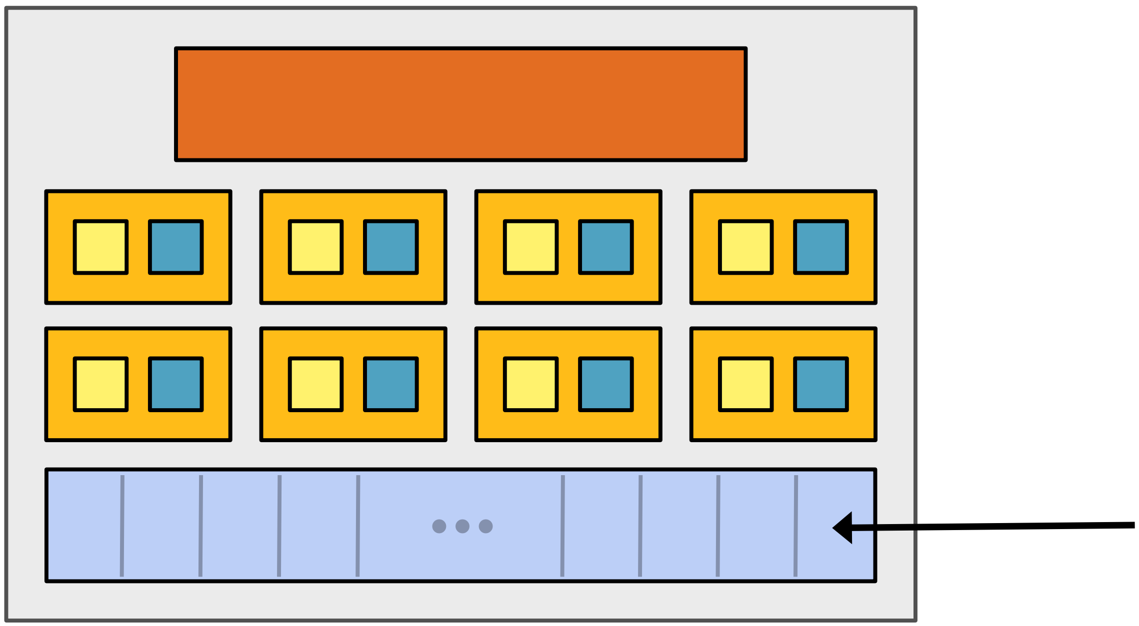
- Many factors play a role in actual chip performance

NVIDIA GeForce GTX 285

- NVIDIA-speak:
  - 240 stream processors
  - "SIMT execution"

- Generic speak:
  - 30 cores
  - 8 SIMD functional units per core

NVIDIA GeForce GTX 285 “core”



64 KB of storage
for fragment
contexts (registers)

= SIMD functional unit, control shared across 8 units

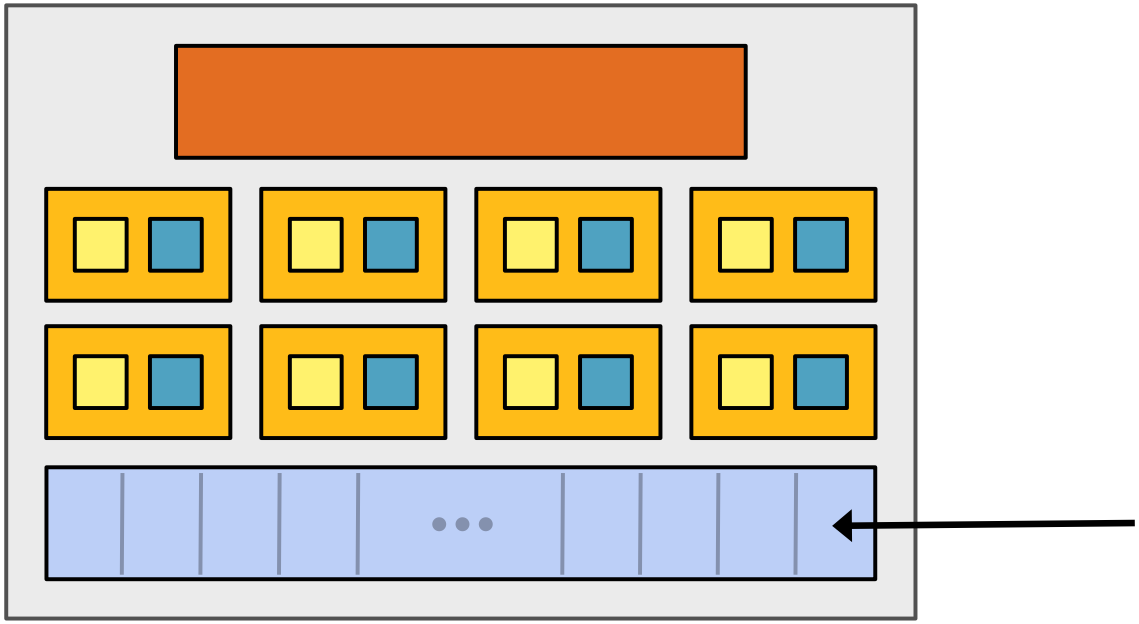
= multiply-add

= multiply

= instruction stream decode

= execution context storage

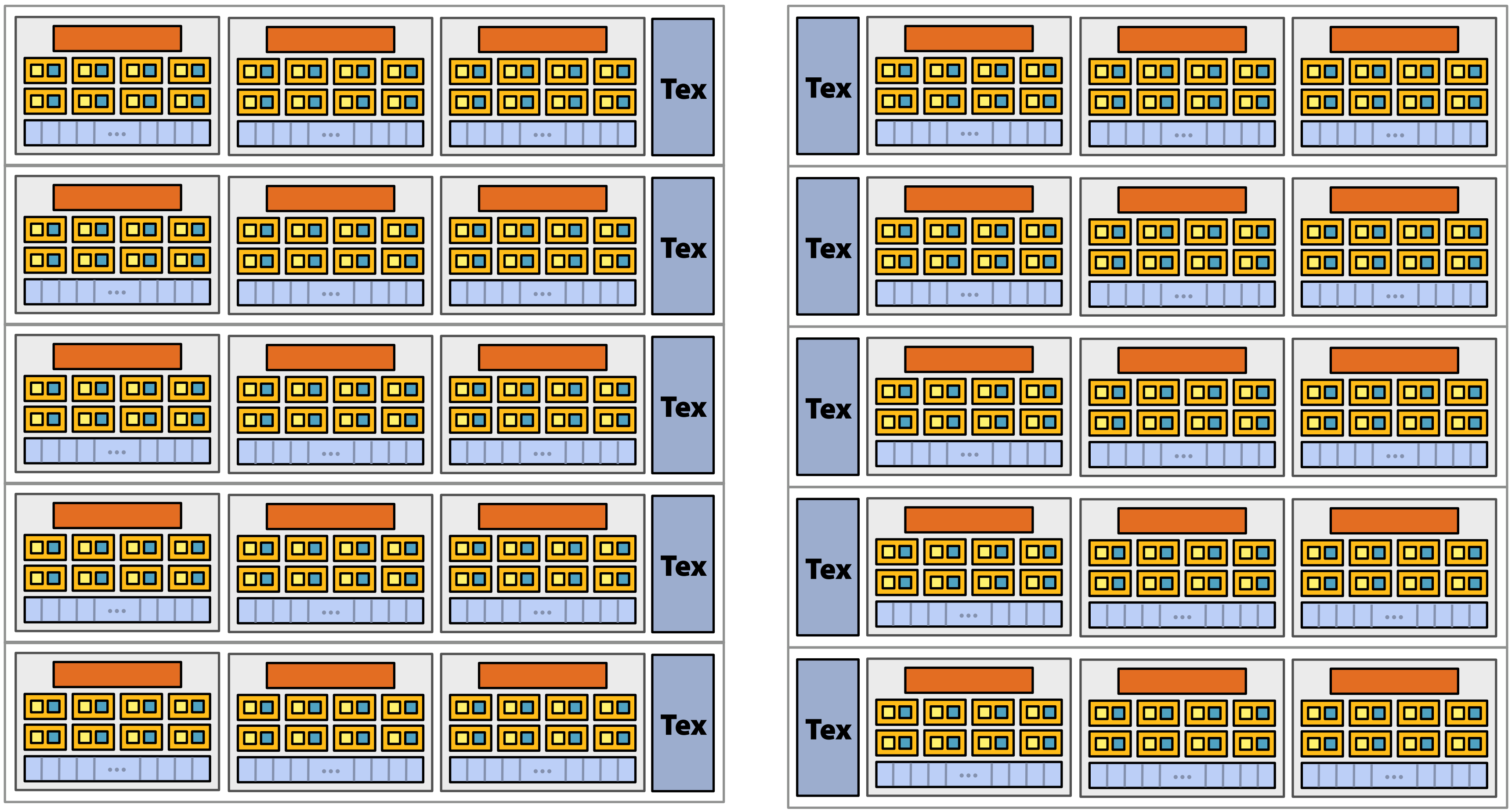
NVIDIA GeForce GTX 285 “core”



64 KB of storage for fragment contexts (registers)

- Groups of 32 [fragments/vertices/threads/etc.] share instruction stream (they are called “WARPS”)
- Up to 32 groups are simultaneously interleaved
- Up to 1024 fragment contexts can be stored

NVIDIA GeForce GTX 285



There are 30 of these things on the GTX 285: 30,000 fragments!

NVIDIA GeForce GTX 285

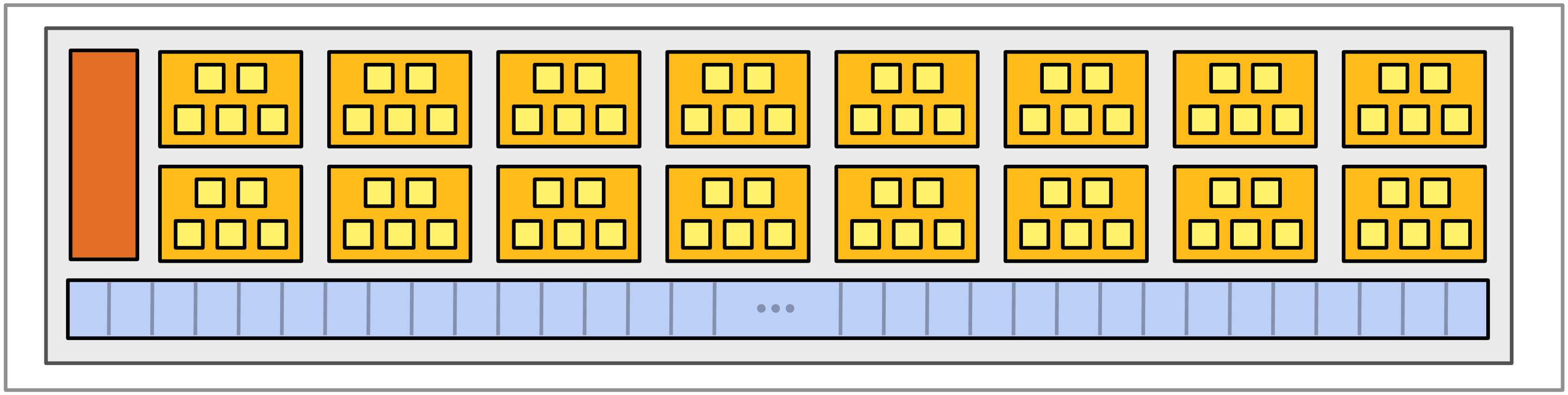
- Generic speak:
  - 30 processing cores
  - 8 SIMD functional units per core
  - Best case: 240 mul-adds + 240 muls per clock

AMD Radeon HD 4890

- AMD-speak:
  - 800 stream processors
  - HW-managed instruction stream sharing (like "SIMT")

- Generic speak:
  - 10 cores
  - 16 "beefy" SIMD functional units per core
  - 5 multiply-adds per functional unit

AMD Radeon HD 4890 “core”



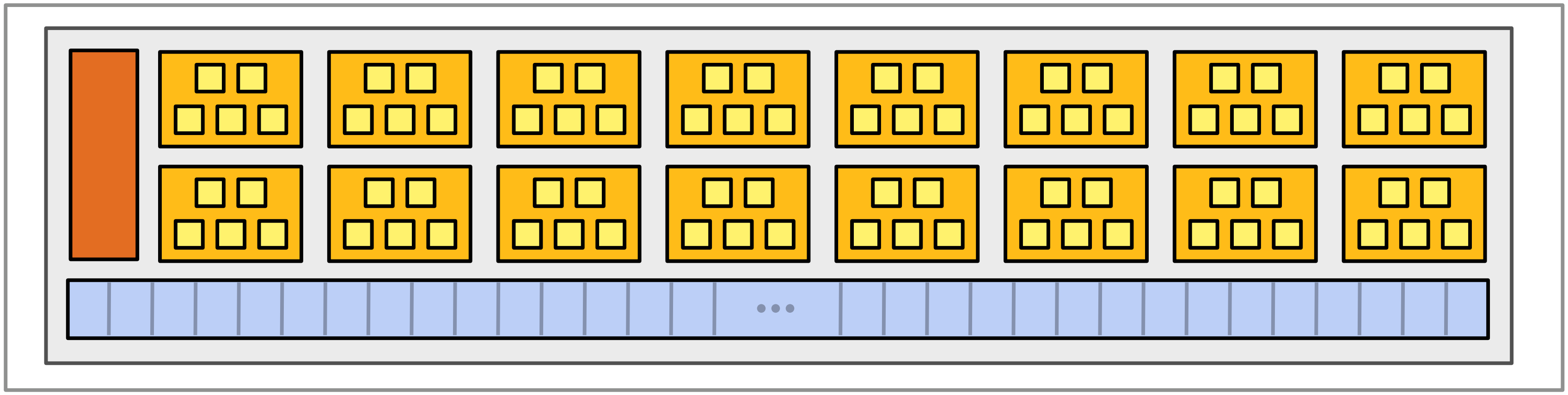
= SIMD functional unit, control shared across 16 units

= multiply-add

= instruction stream decode

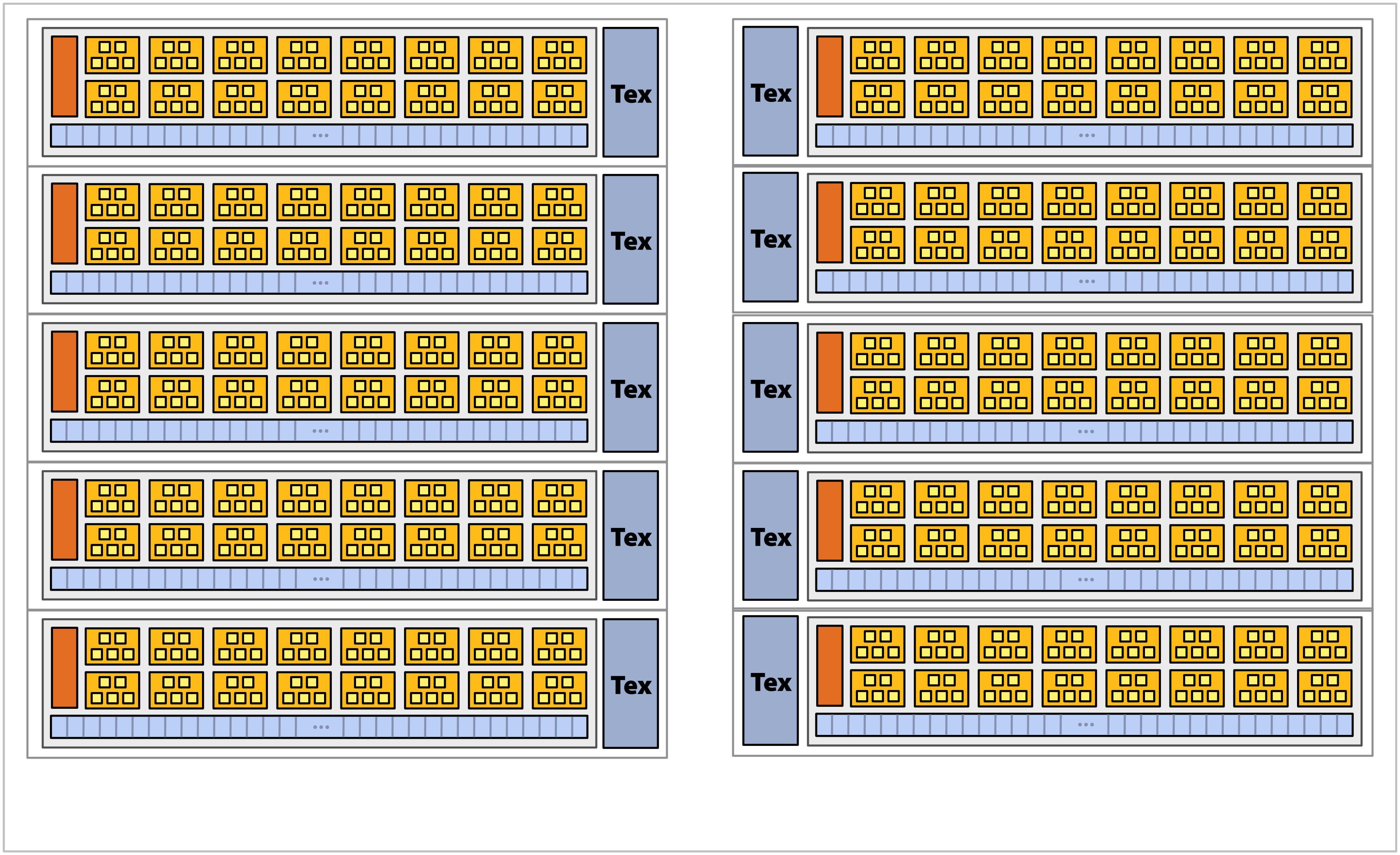
= execution context storage

AMD Radeon HD 4890 "core"



- Groups of 64 [fragments/vertices/etc.] share instruction stream (AMD doesn't have a fancy name like "WARP")
  - One fragment processed by each of the 16 SIMD units
  - Repeat for four clocks

AMD Radeon HD 4890



AMD Radeon HD 4890

- Generic speak:
  - 10 processing “cores”
  - 16 “beefy” SIMD functional units per core
  - 5 multiply-adds per functional unit
  - Best case: 800 multiply-adds per clock

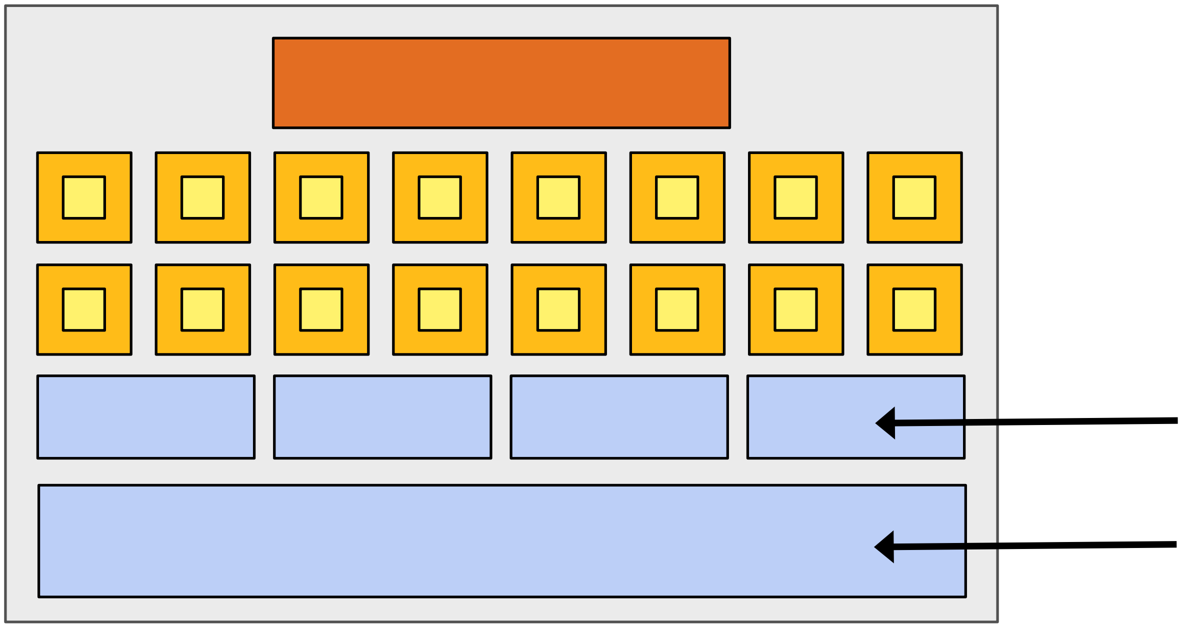
- Scale of interleaving similar to NVIDIA GPUs

Intel Larrabee

- Intel speak:
  - We won't say anything about core count or clock rate
  - Explicit 16-wide vector ISA
  - Each core interleaves four x86 instruction streams
  - Software implements additional interleaving

- Generic speak:
  - That was the generic speak

Intel Larrabee "core"



Each HW context:
32 vector registers

32 KB of L1 cache
256 KB of L2 cache

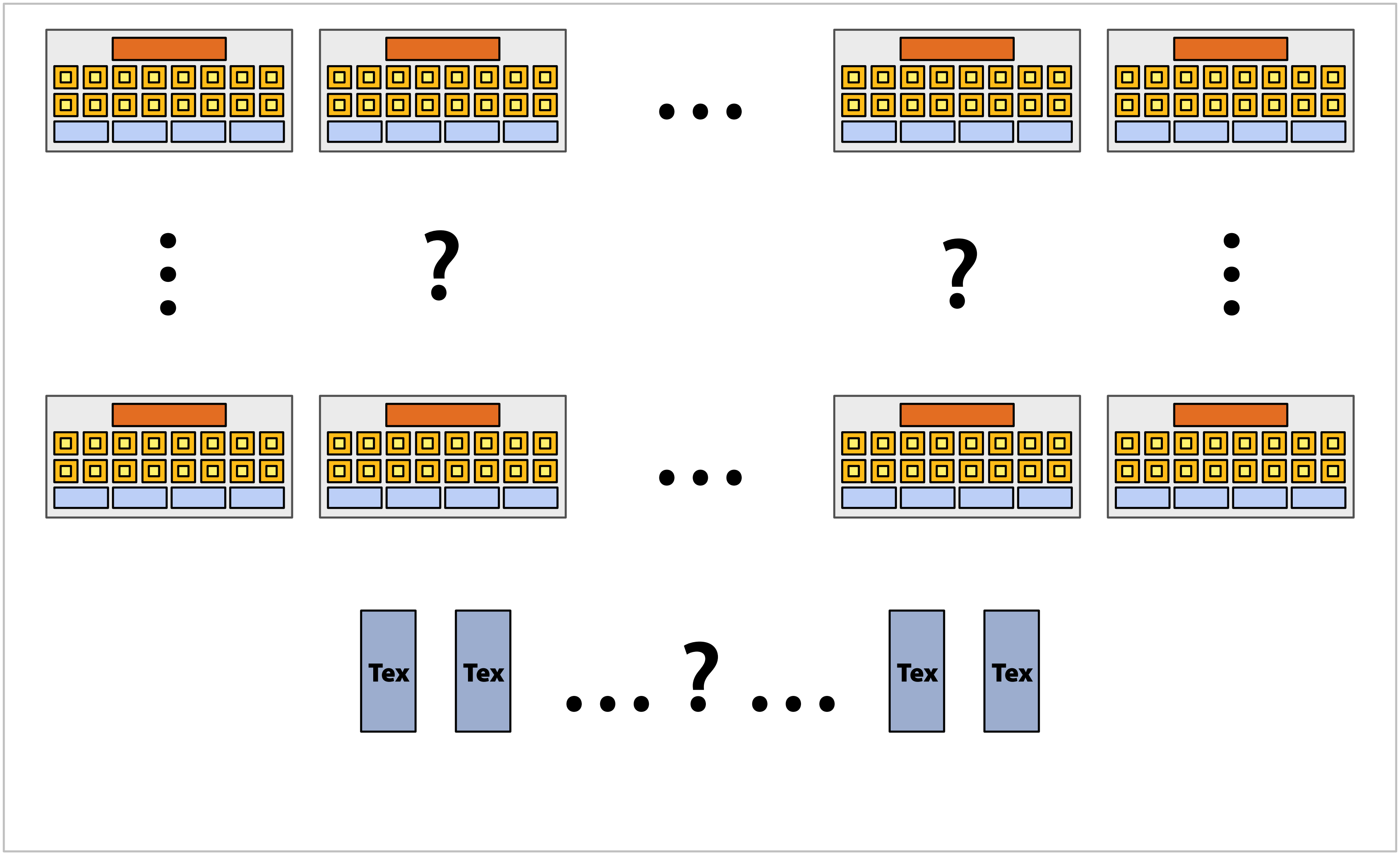
= SIMD functional unit, control shared across 16 units

= mul-add

= instruction stream decode

= execution context storage/ HW registers

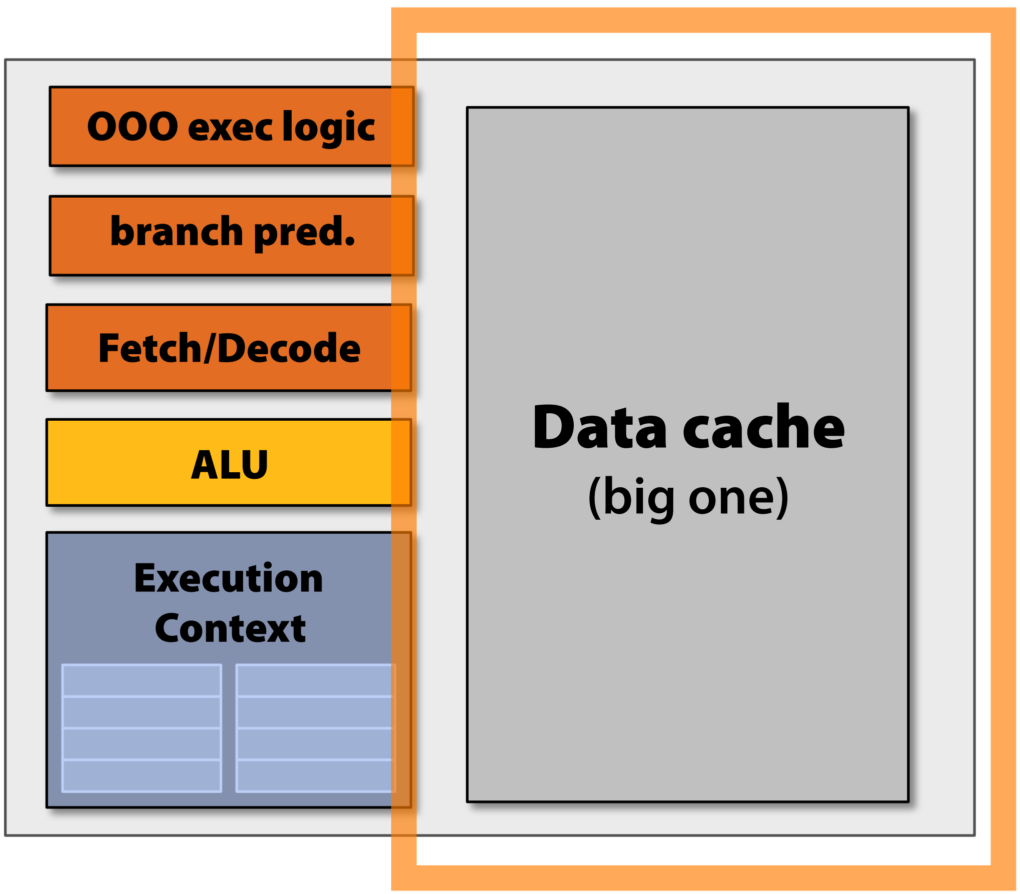
Intel Larrabee



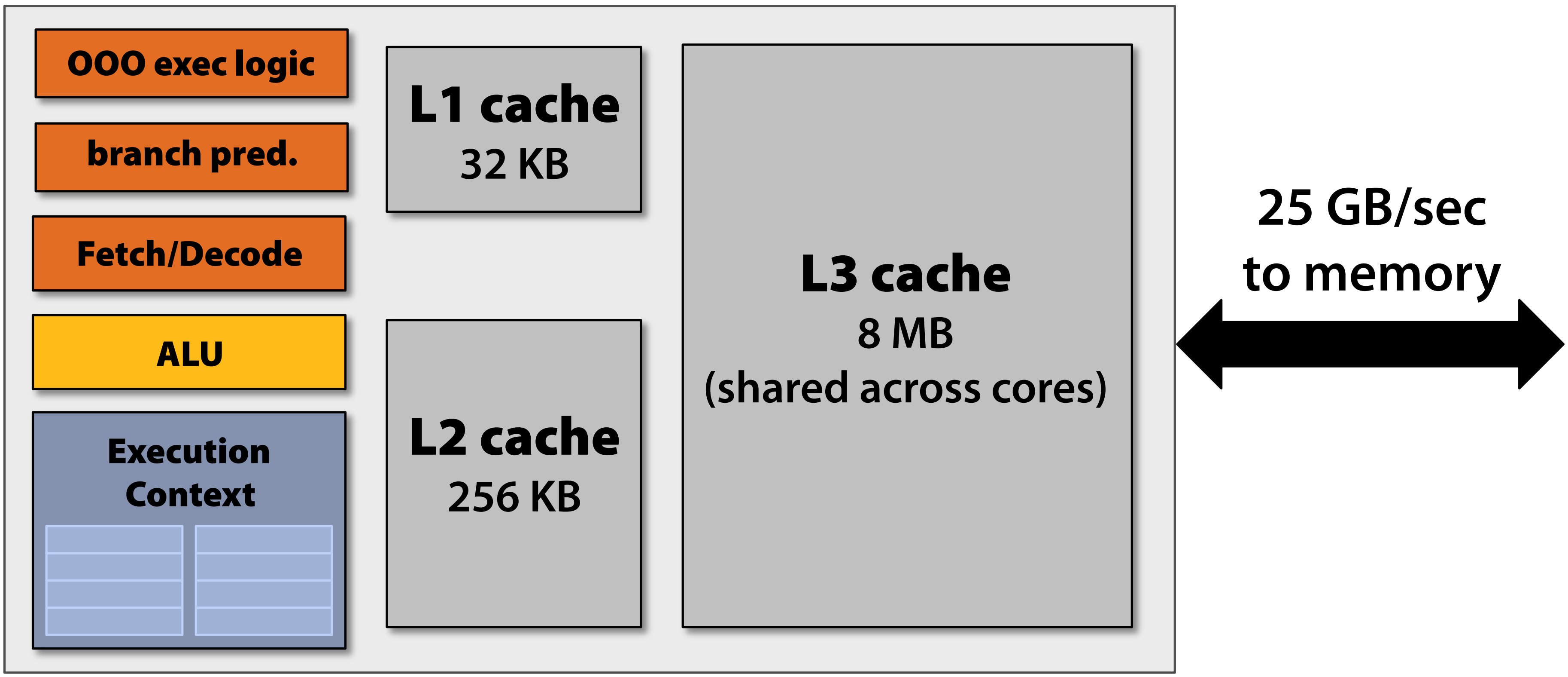
The talk thus far: processing data

Part 3: moving data to processors

Recall: CPU-"style" core

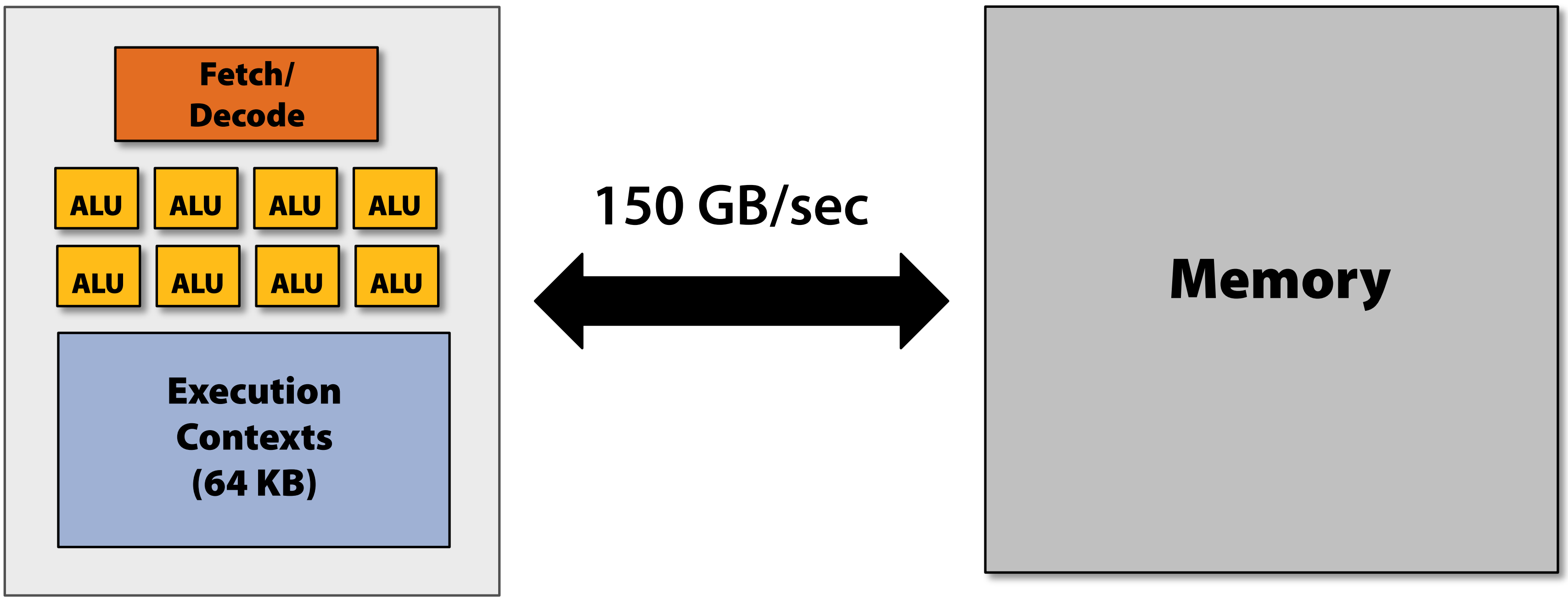


CPU-“style” memory hierarchy



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

Throughput core (GPU-style)



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

Bandwidth is critical

- On a high-end GPU:
  - 11x compute performance of high-end CPU
  - 6x bandwidth to feed it
  - No complicated cache hierarchy

- GPU memory system is designed for throughput
  - Wide bus (150 GB/sec)
  - Repack/reorder/interleave memory requests to maximize use of memory bus

Bandwidth thought experiment

- Element-wise multiply two long vectors A and B
  - Load input A[i]
  - Load input B[i]
  - Multiply
  - Store result C[i]

- 3 memory operations every 4 cycles (12 bytes)
- Needs ~1 TB/sec of bandwidth on a high-end GPU
- 7x available bandwidth

15% efficiency… but 6x faster than high-end 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.

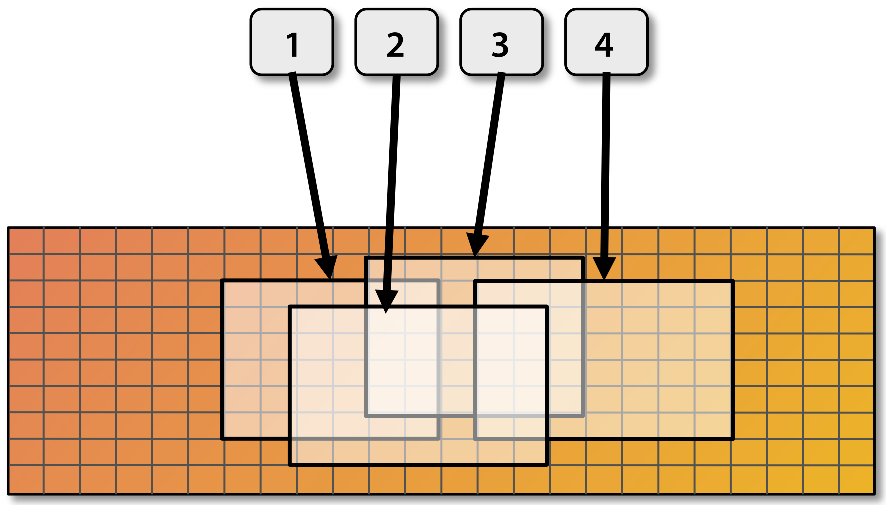
Reducing required bandwidth

Request data less often
(do more math)

Share/reuse data across fragments
(increase on-chip storage)

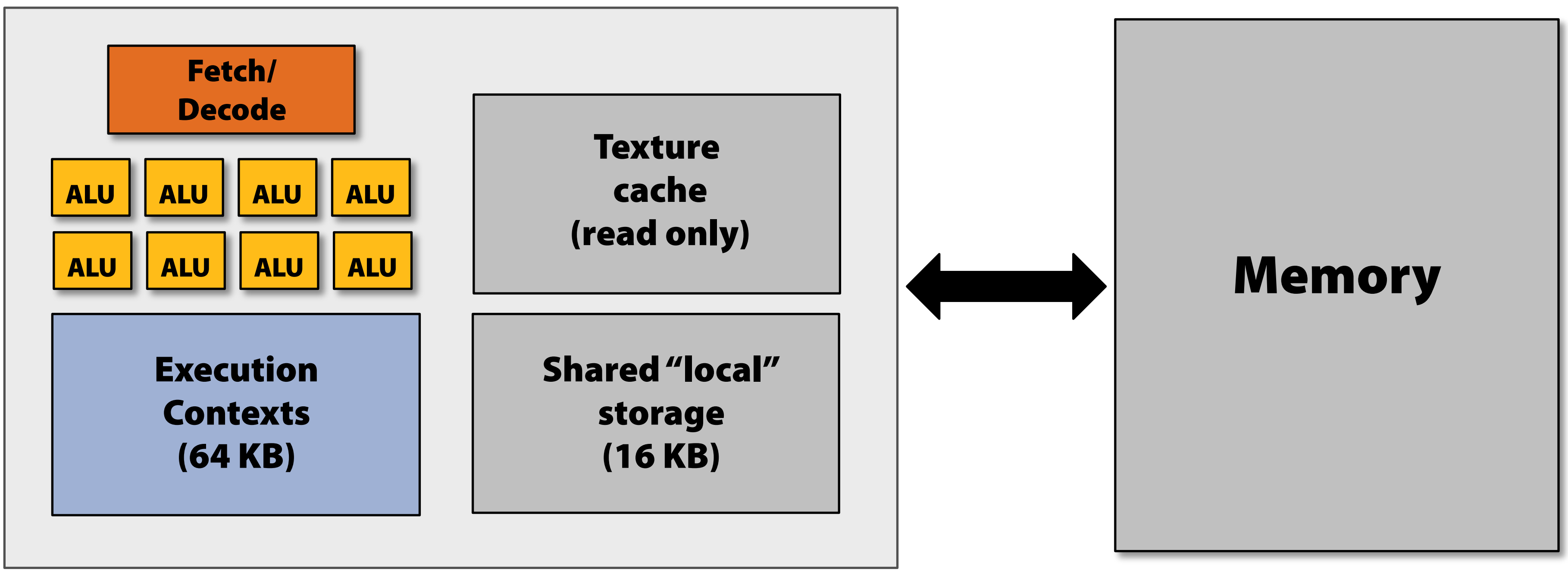
Reducing required bandwidth

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



Texture data

GPU memory hierarchy



On-chip storage takes load off memory system
Many developers calling for larger, more cache-like storage

Blocks and warps on NVidia

- The programmer groups threads into *blocks* which are assigned to a stream processor
- These in turn are grouped into *warps*, which are groups of threads that execute together. The number of threads in a warp is a function of ALUs
- Warps are a scheduling unit
- Coalescing is attempted on memory accesses by a warp
  - Attempt to group a set of accesses into as few accesses as possible
  - Reduces the number of DMA operations, increases bandwidth

Summary

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

(currently at extreme end of design space)

An efficient GPU workload…

- Has thousands of <u>independent</u> 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 bandwidth

Acknowledgements

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

Thank you

http://gates381.blogspot.com