Lecture 7:
The Programmable GPU Core

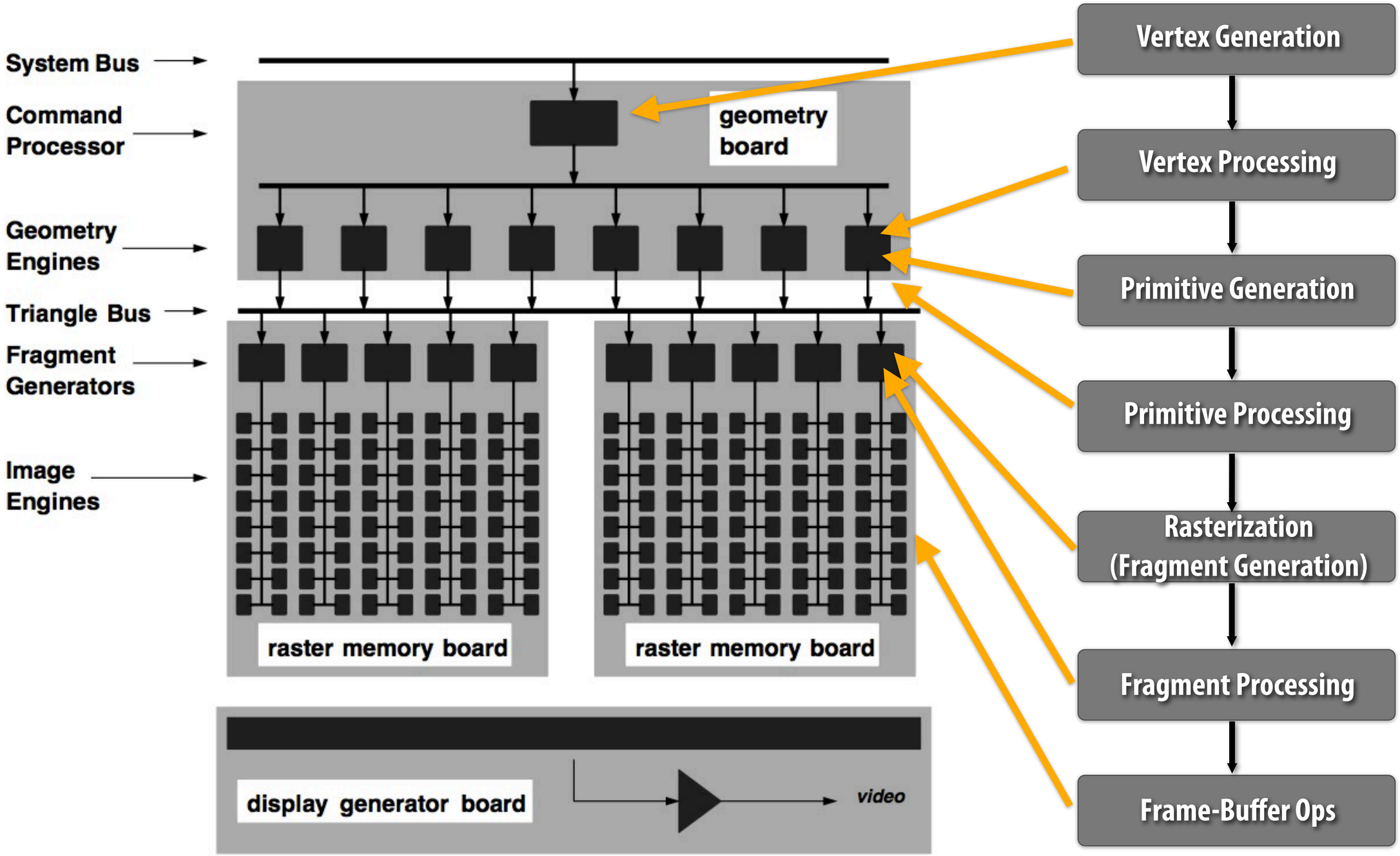
Kayvon Fatahalian

CMU 15-869: Graphics and Imaging Architectures (Fall 2011)

Today

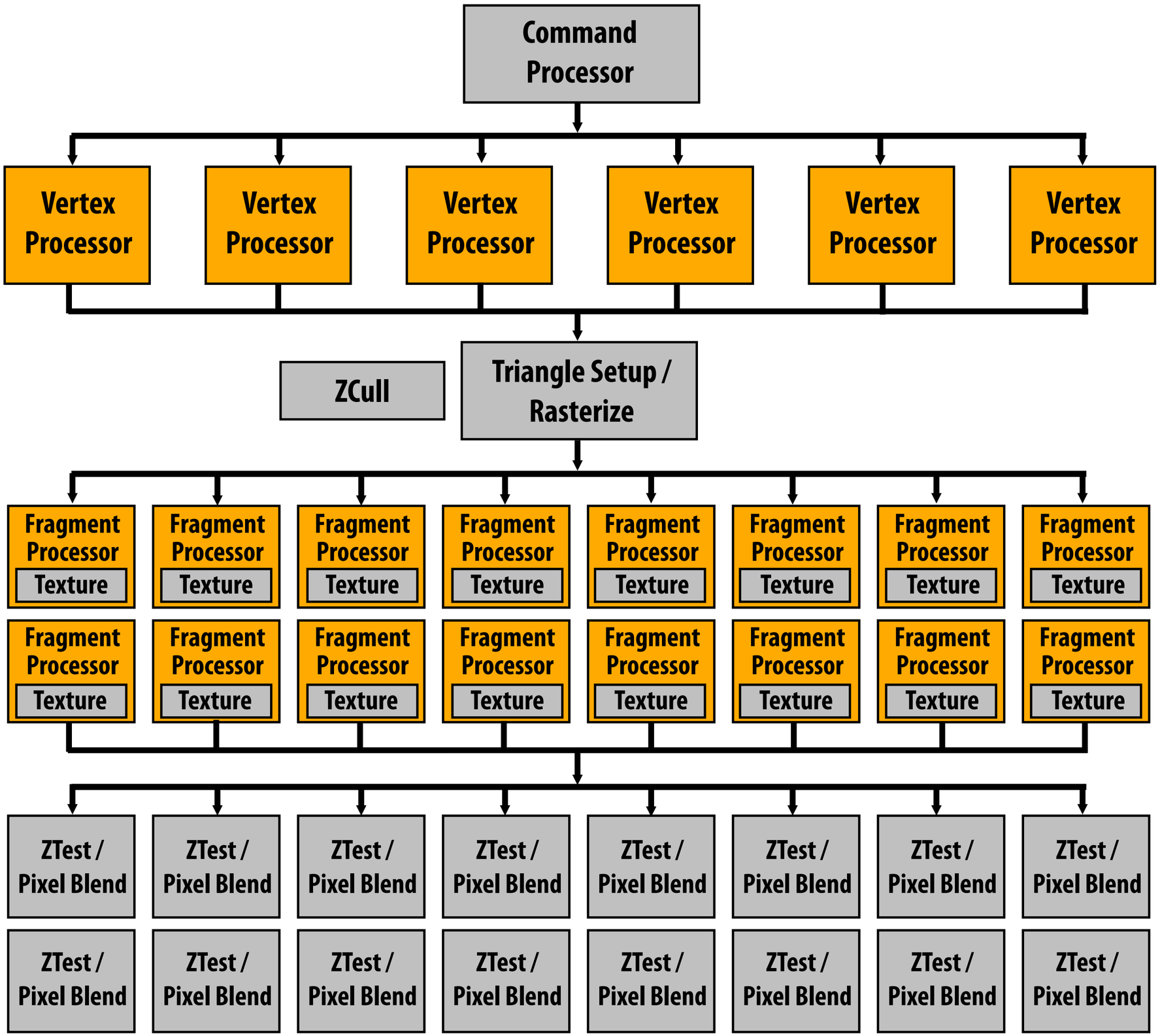
- A brief history of GPU programmability
- Throughput processing core 101
- A detailed look at recent GPU designs

SGI RealityEngine (1993)

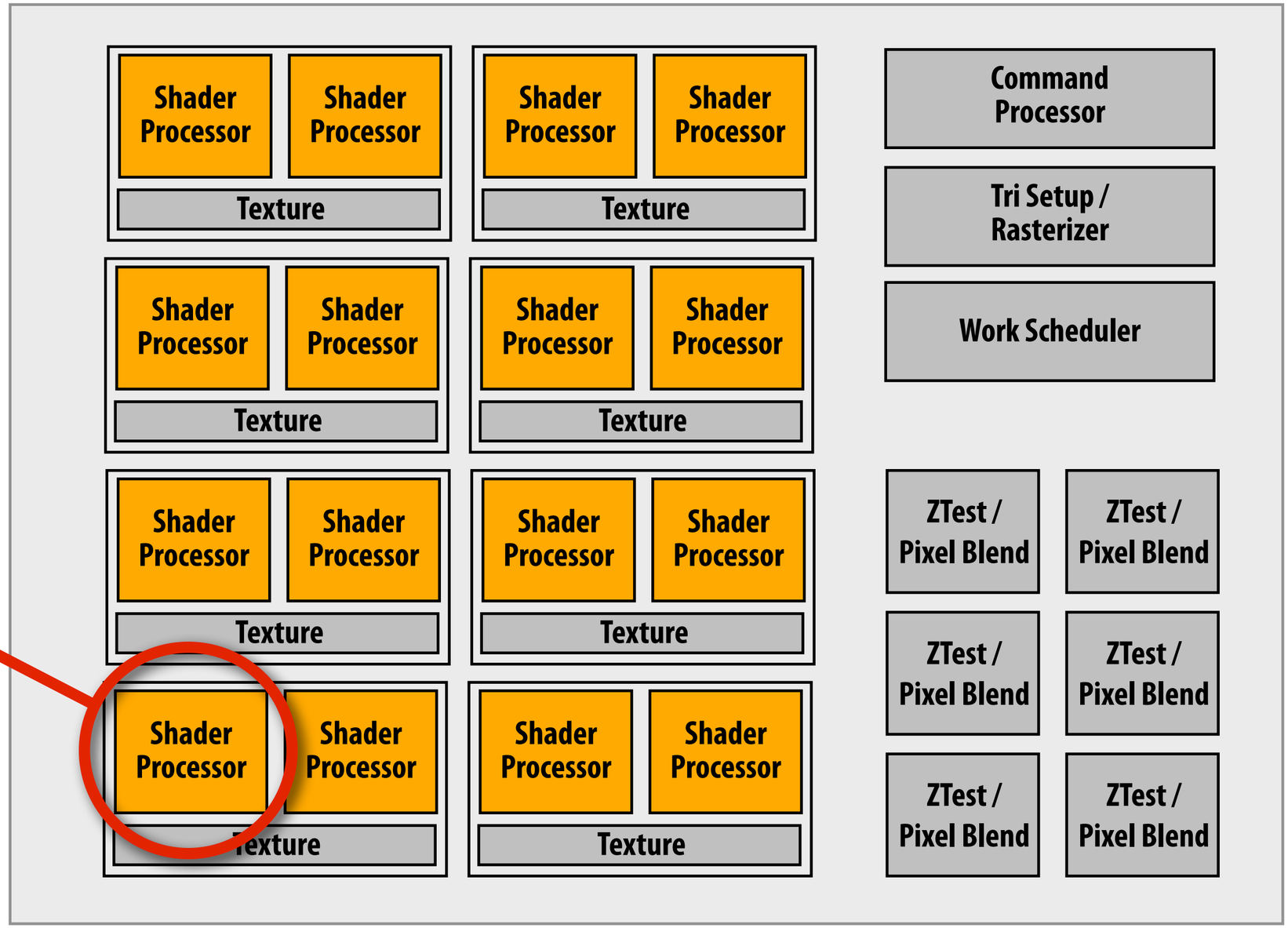


Programmable, just not by application:
(Geometry Engine contained Intel i860XP processor)

NVIDIA GeForce 6800 (2004)



NVIDIA GeForce 8800 (Tesla, 2006)



Today's topic

"Unified shading" : pool of programmable processors execute vertex and fragment processing

Cores also exposed via alternative abstraction (CUDA, 2007)

(Note: Programmable resources exposed by CUDA as homogeneous multi-core)

The GPU programmable processing core

- Three major ideas that make GPU processing cores run fast
- Closer look at two real GPU designs
  - NVIDIA GTX 480
  - AMD Radeon 5870

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);
}
```

Shader programming model:

Fragments are processed *independently*, but there is no explicit parallel programming.

Independent logical sequence of control per fragment. ***

***** In this talk, references to "fragment" can be replaced with "vertex" (in the vertex shader), "primitive" (in the geometry or hull shaders), or "thread" (in OpenCL or CUDA)

Compile shader

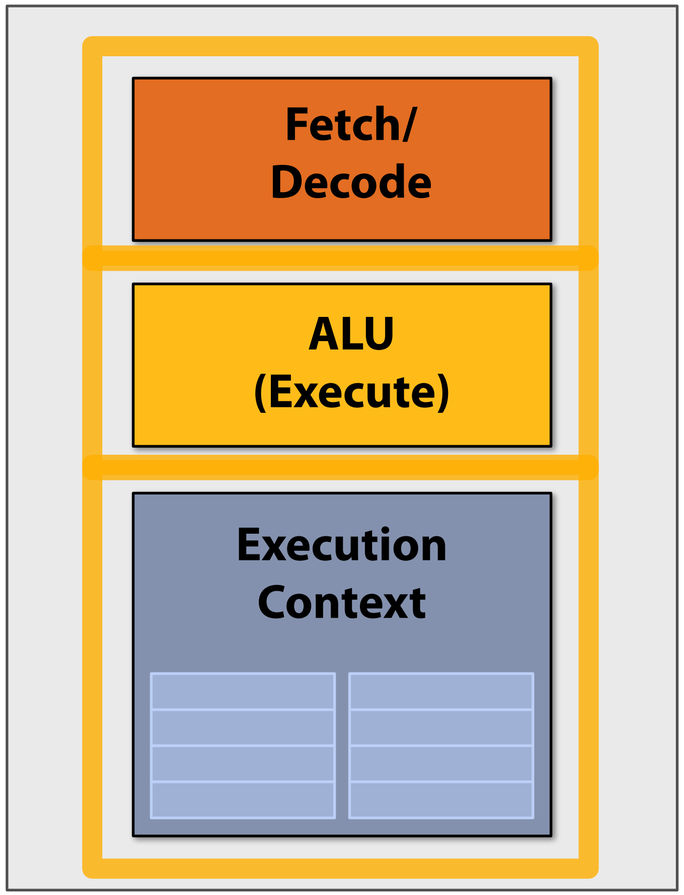
```
sampler mySamp;
Texture2D<float3> myTex;
float3 lightDir;

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);
}
```

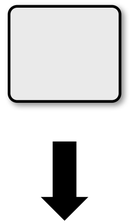
1 unshaded fragment input record

```
<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)
```

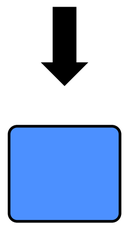
1 shaded fragment output record

Execute shader

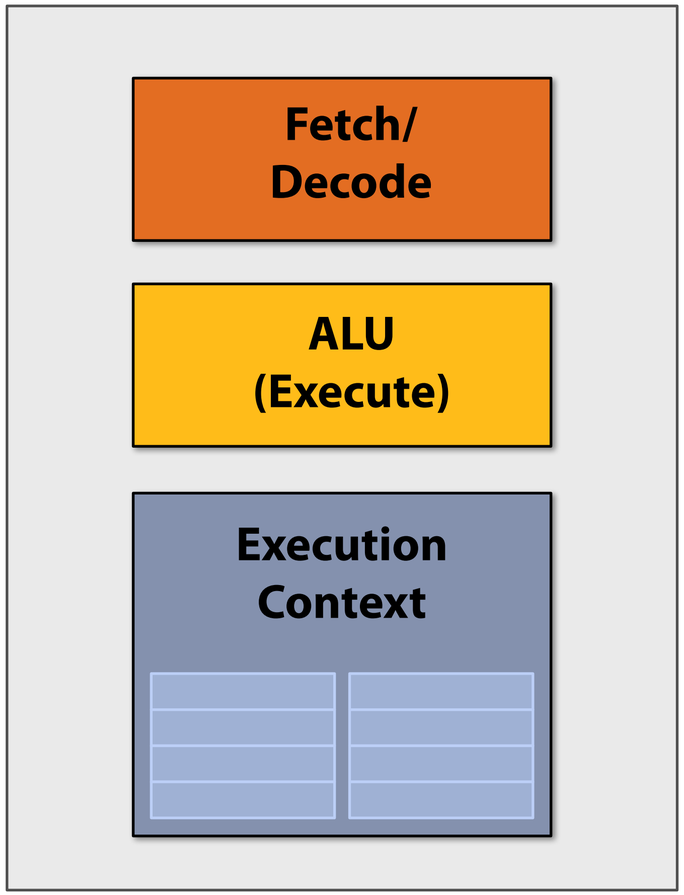


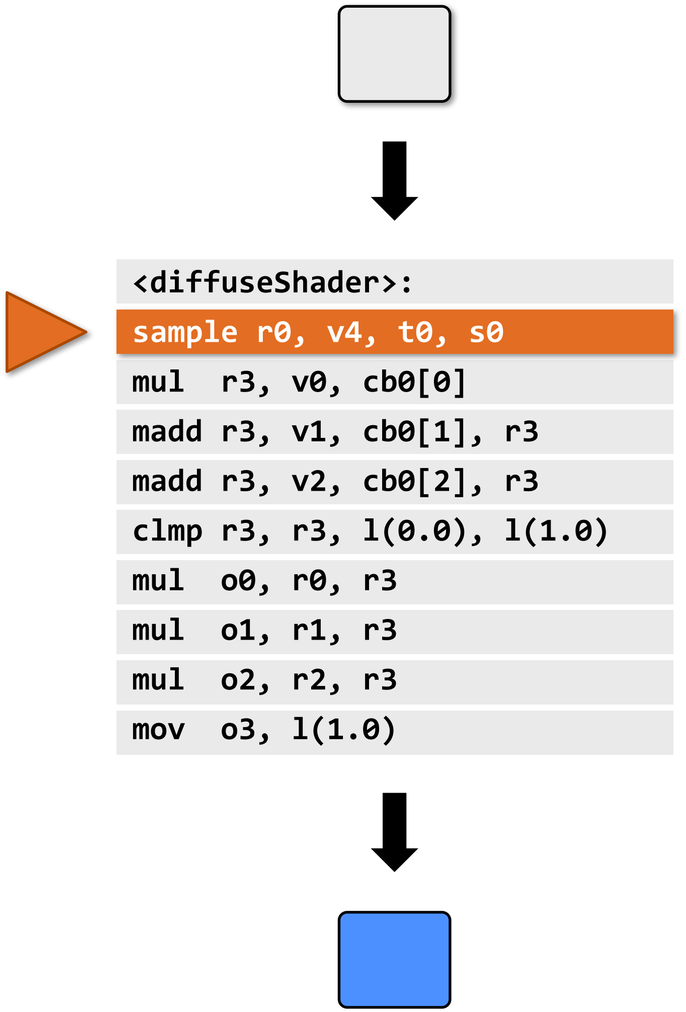


```
<diffuseShader>:
sample r0, v4, t0, s0
mul  r3, v0, cb0[0]
madd r3, v1, cb0[1], r3
madd r3, v2, cb0[2], r3
clmp r3, r3, l(0.0), l(1.0)
mul  o0, r0, r3
mul  o1, r1, r3
mul  o2, r2, r3
mov  o3, l(1.0)
```

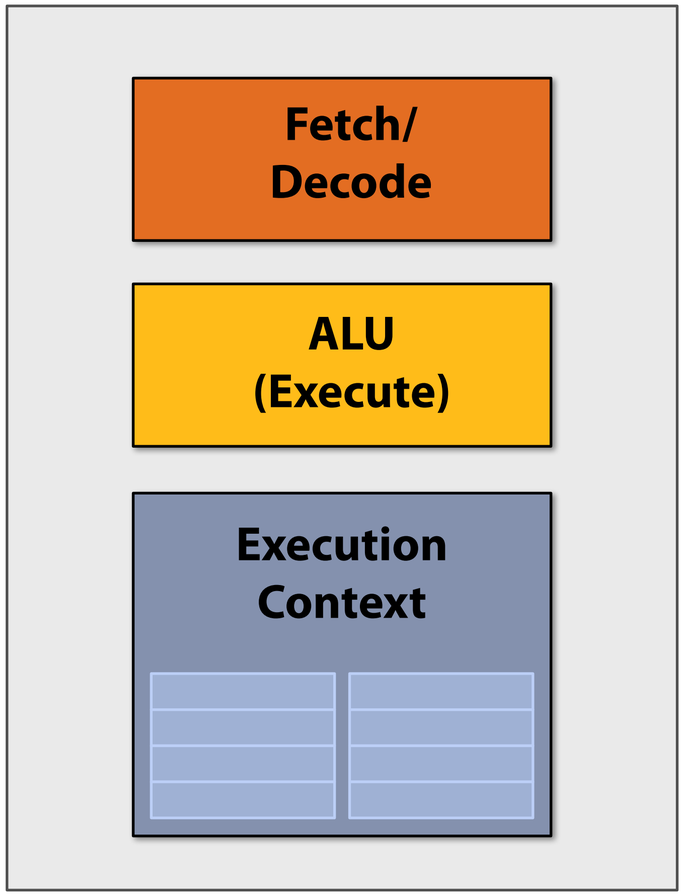


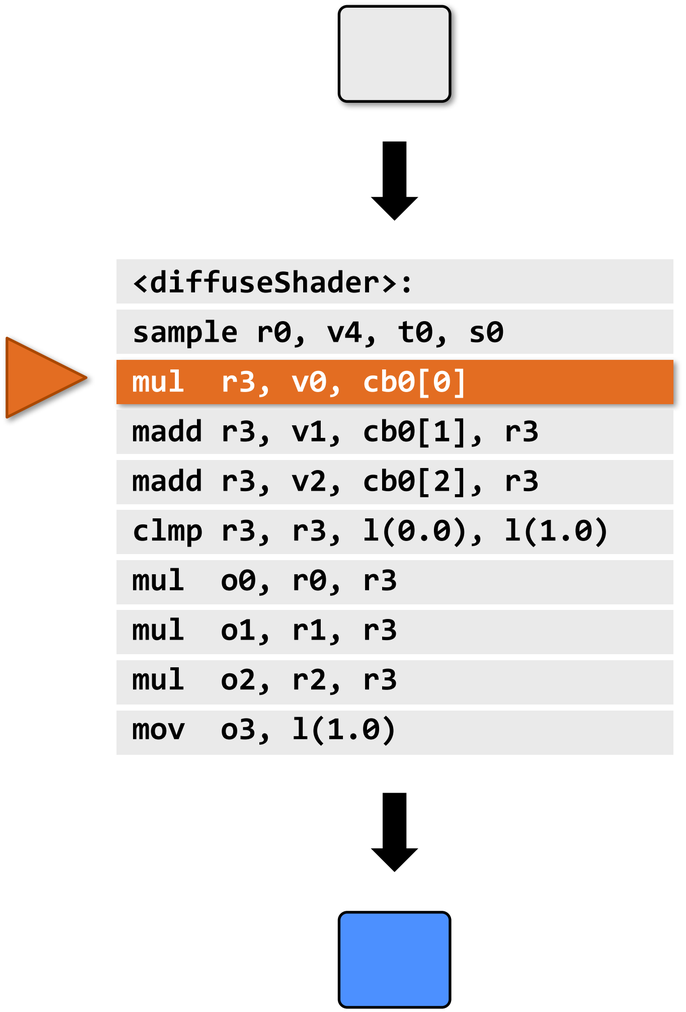
Execute shader



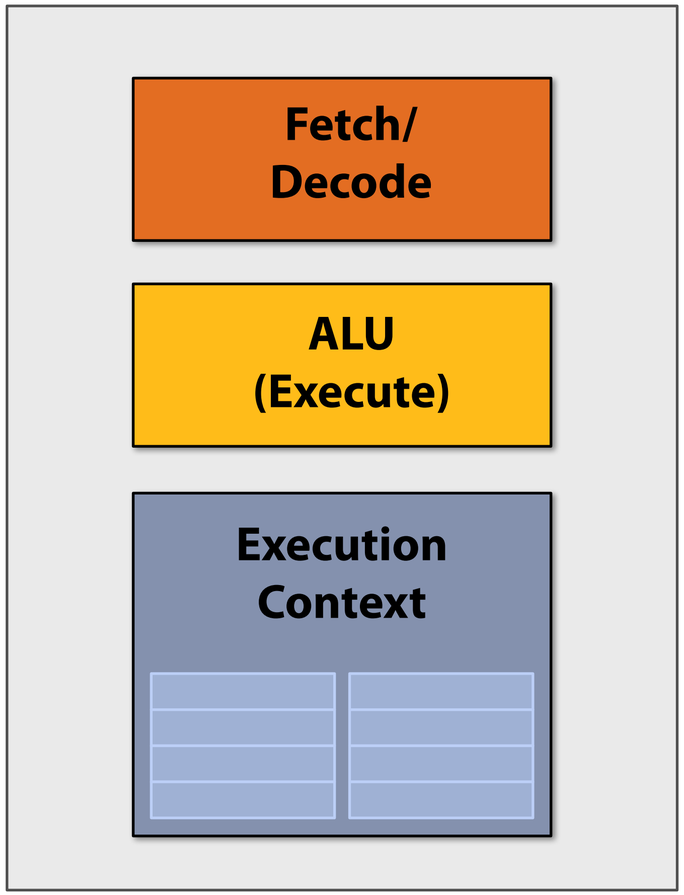


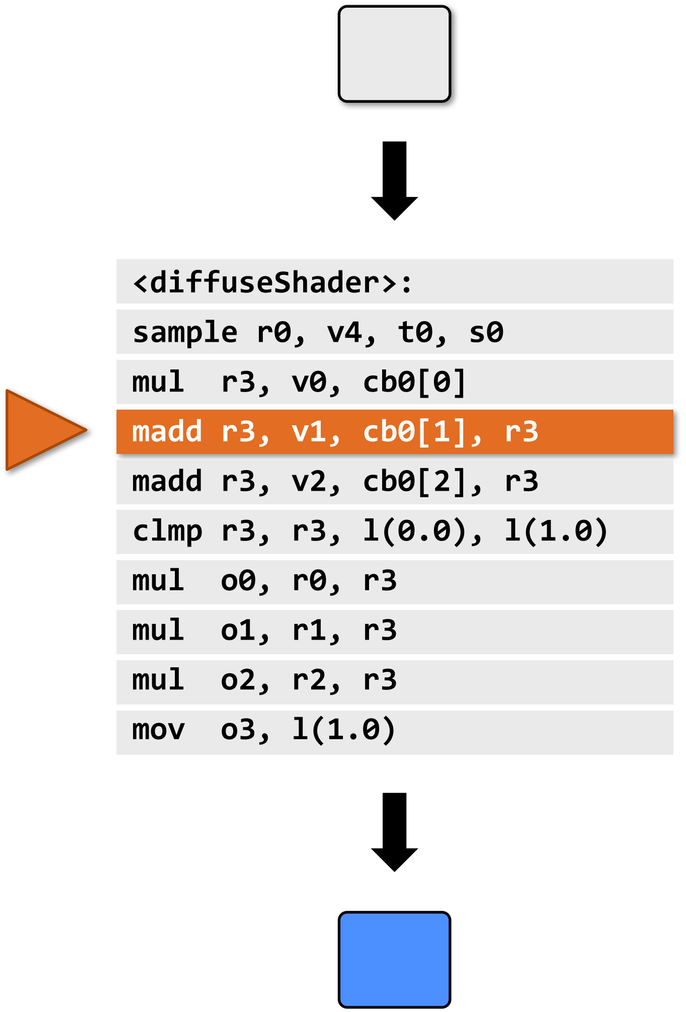
Execute shader



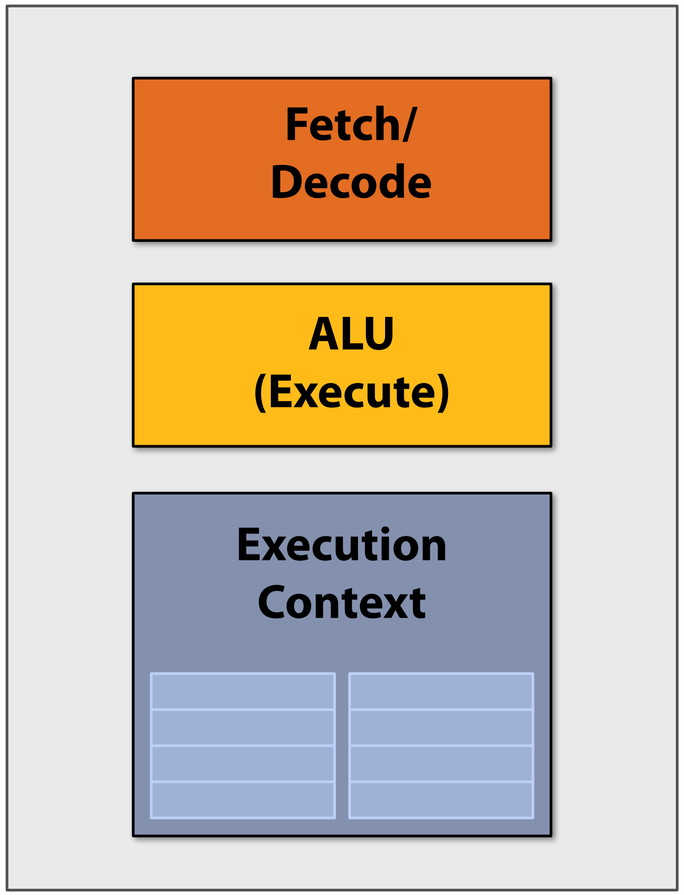


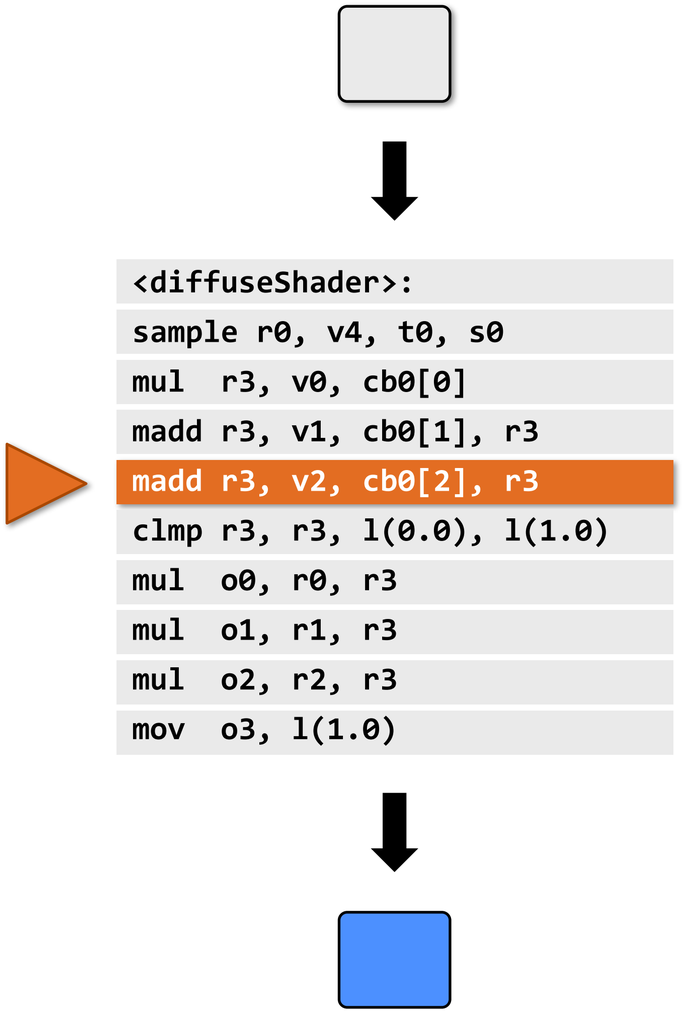
Execute shader



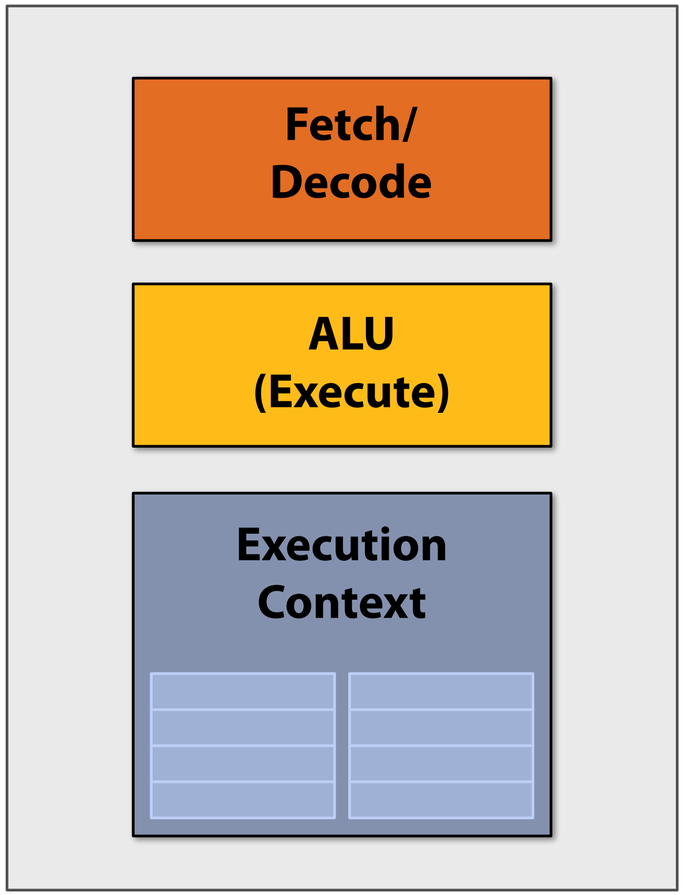


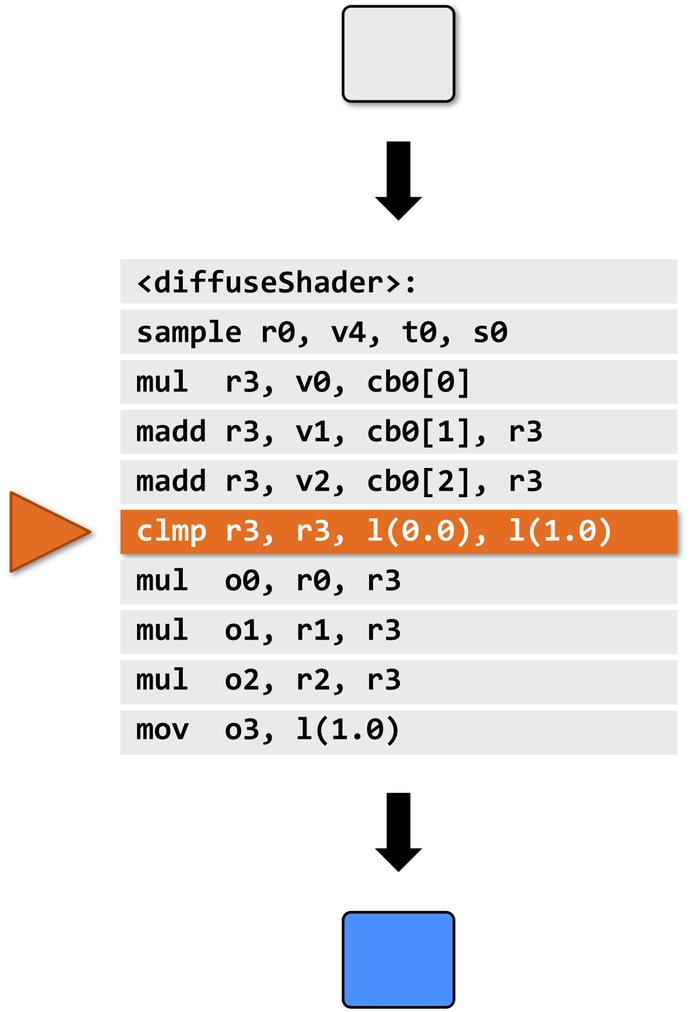
Execute shader



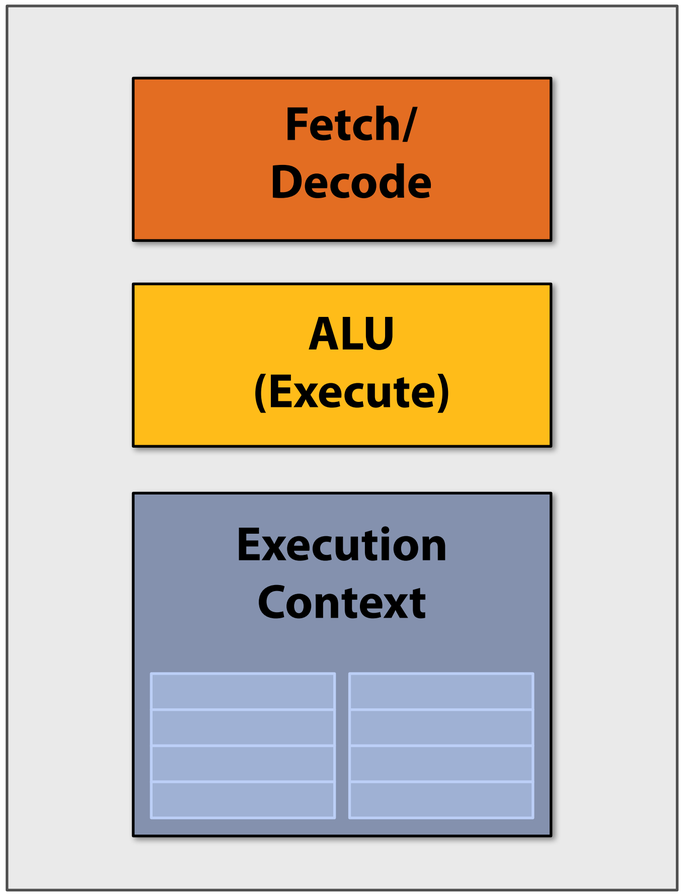


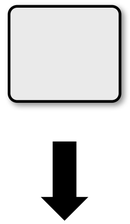
Execute shader



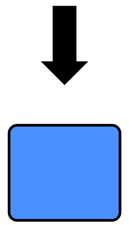


Execute shader

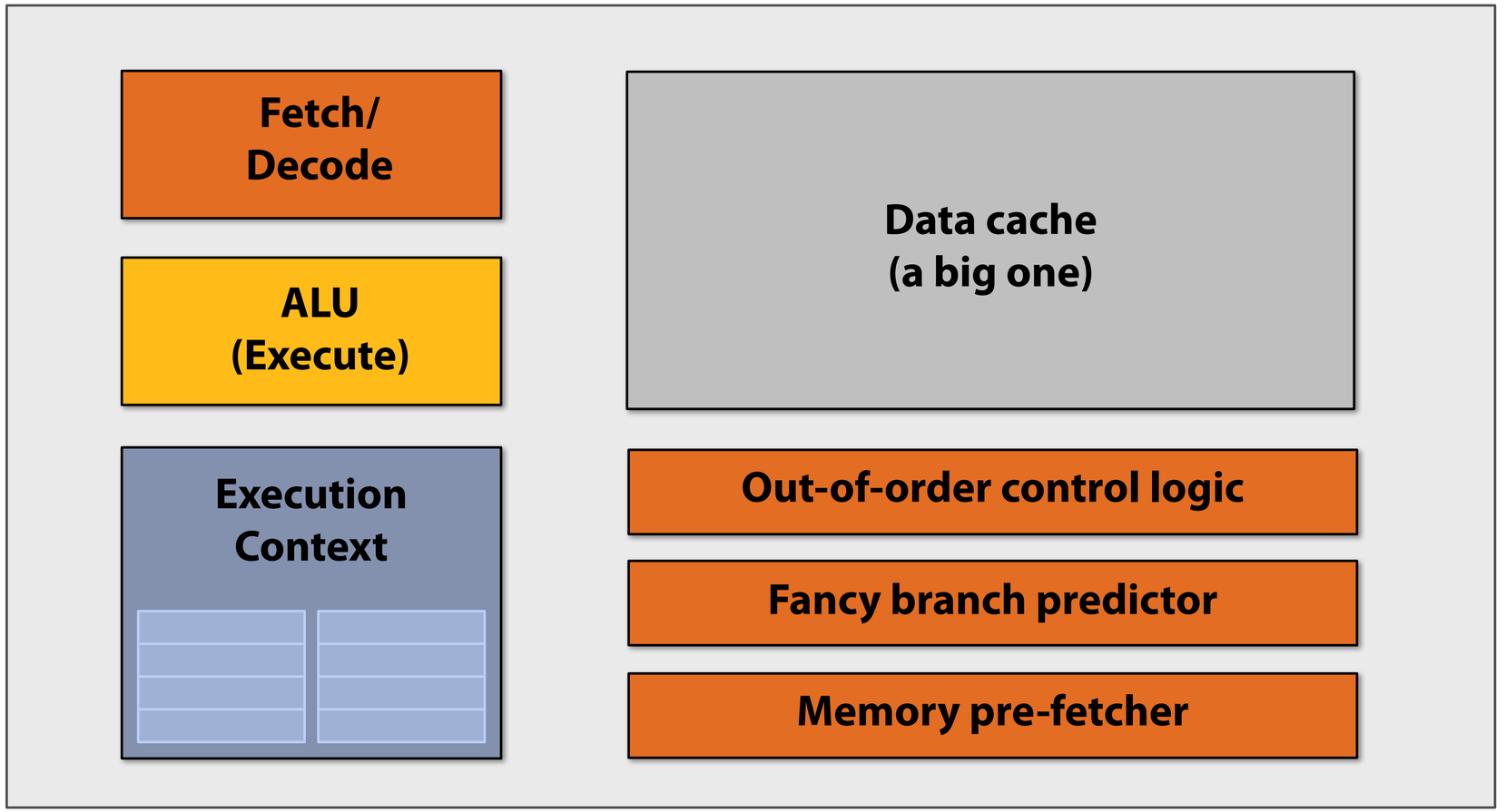




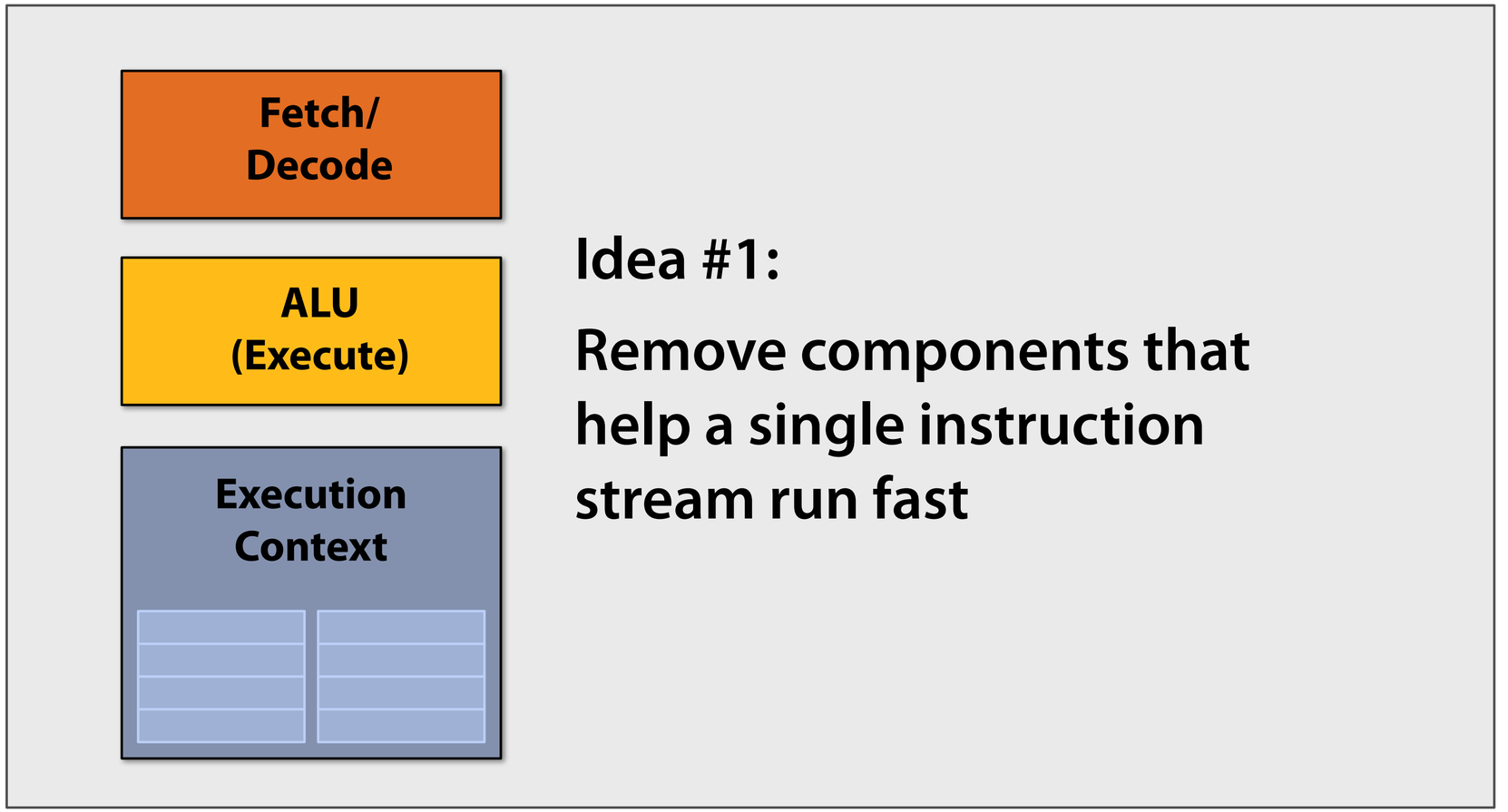
```
<diffuseShader>:
sample r0, v4, t0, s0
mul  r3, v0, cb0[0]
madd r3, v1, cb0[1], r3
madd r3, v2, cb0[2], r3
clmp r3, r3, l(0.0), l(1.0)
mul  o0, r0, r3
mul  o1, r1, r3
mul  o2, r2, r3
mov  o3, l(1.0)
```



“CPU-style” cores



Slimming down



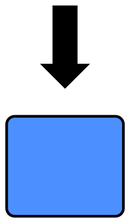
Idea #1:

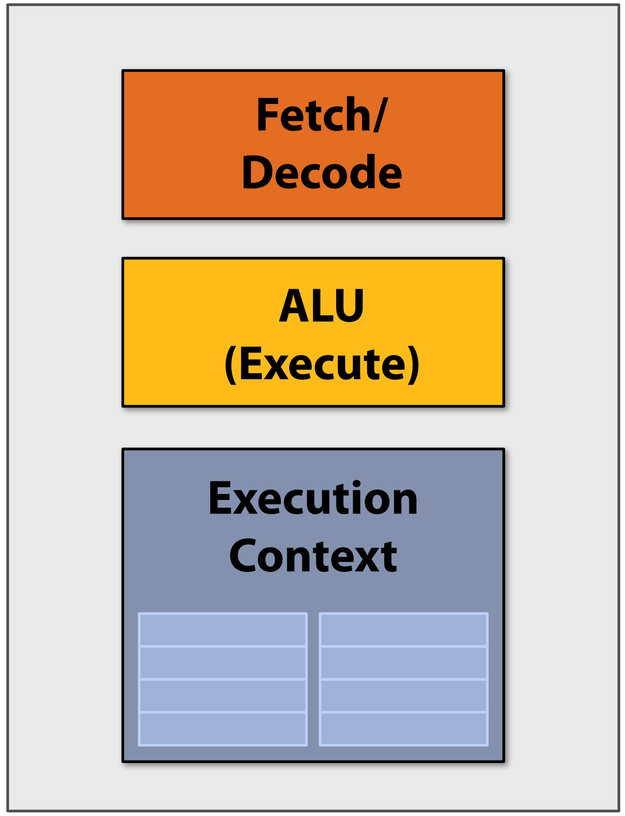
Remove components that help a single instruction stream run fast

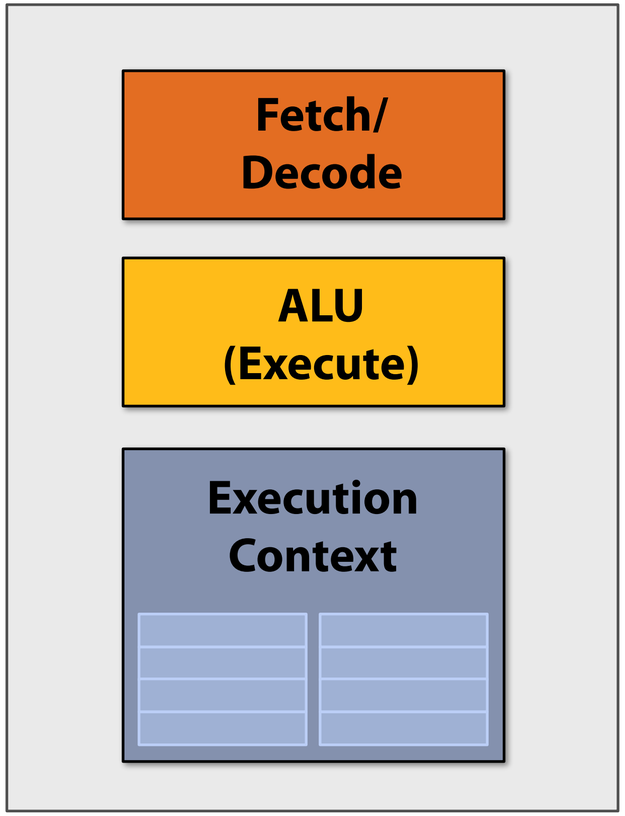
Two cores (two fragments in parallel)

fragment 1

```
<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)
```

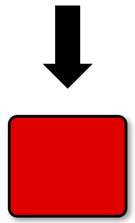




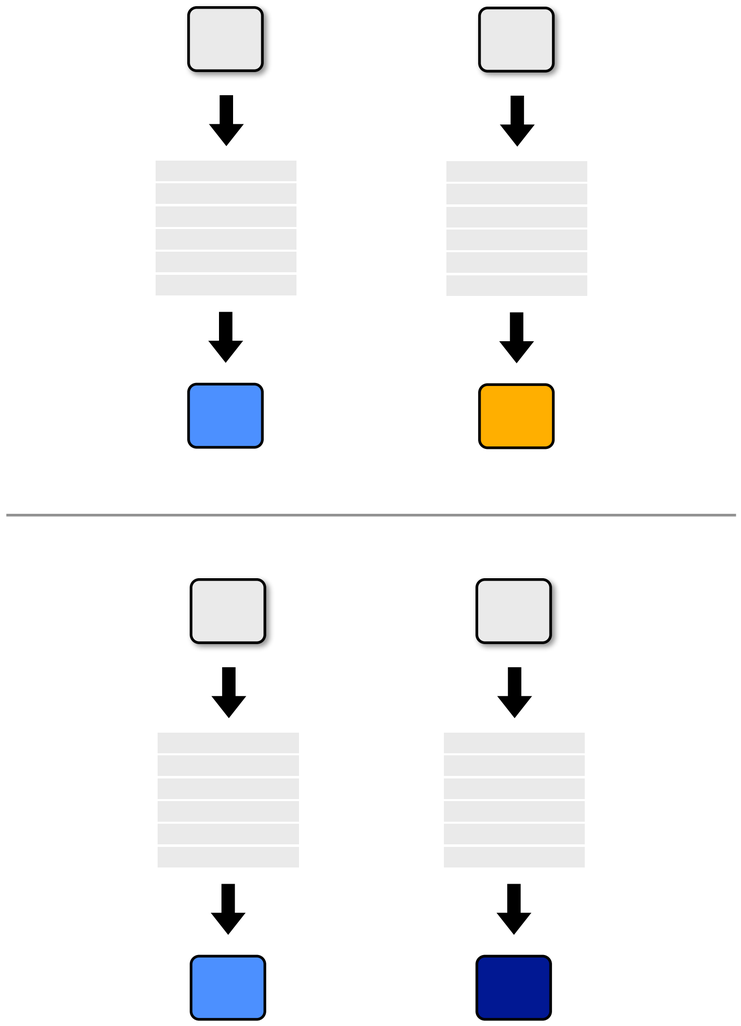


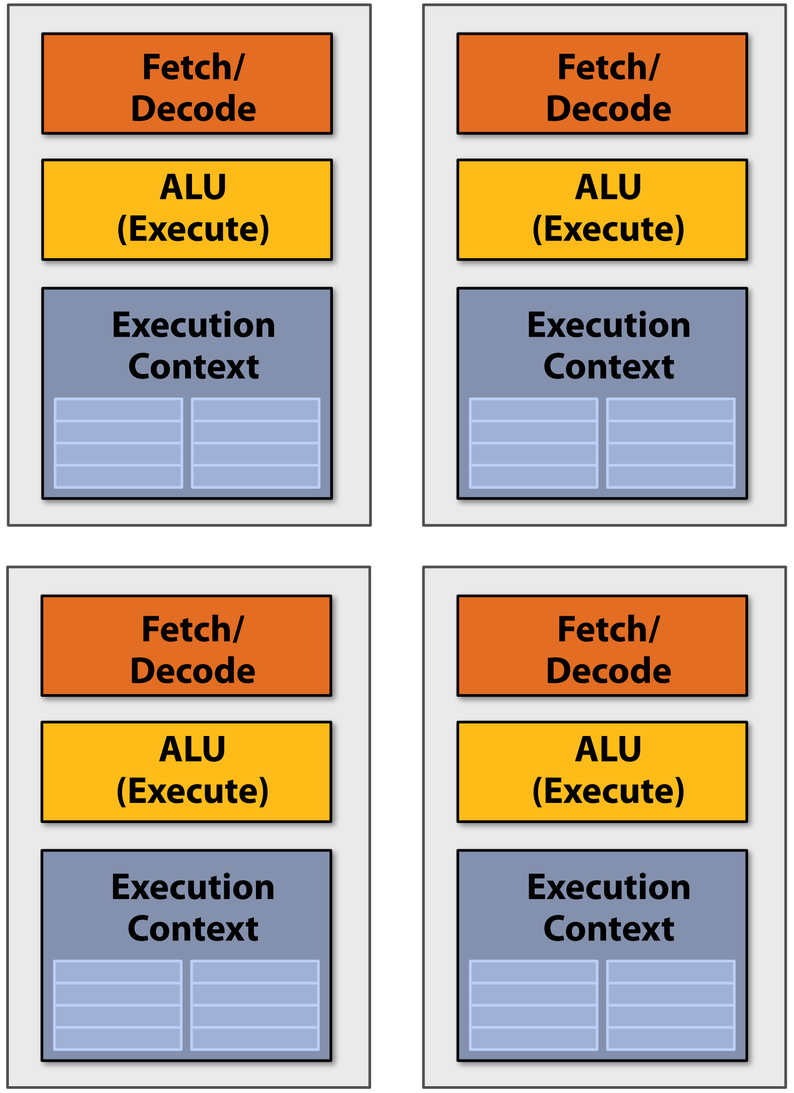
fragment 2

```
<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)
```

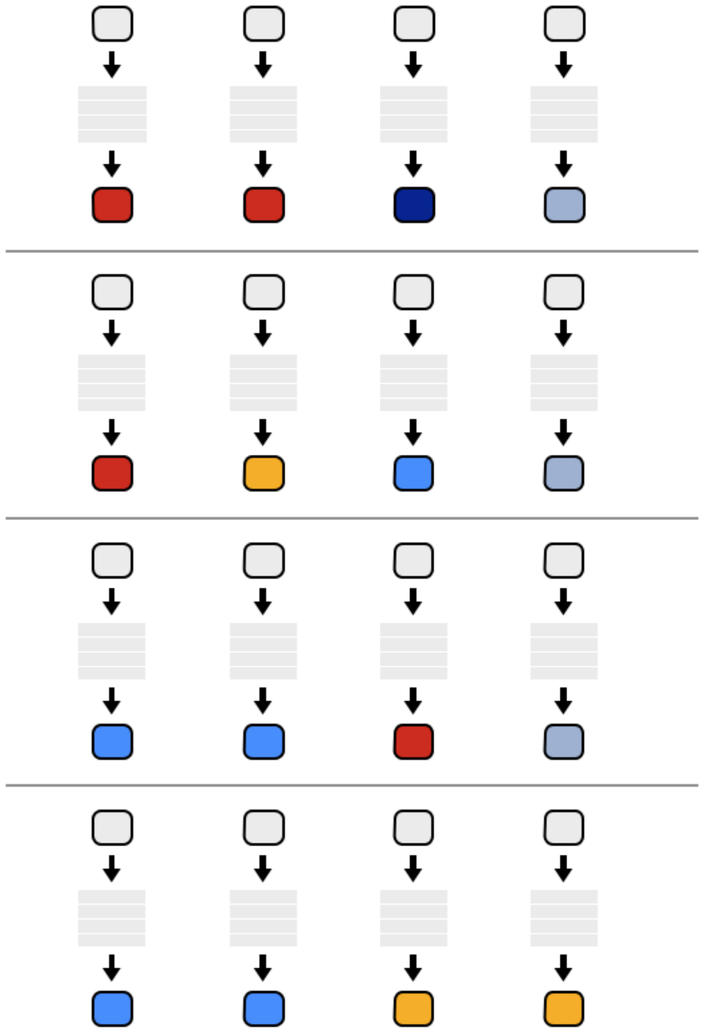


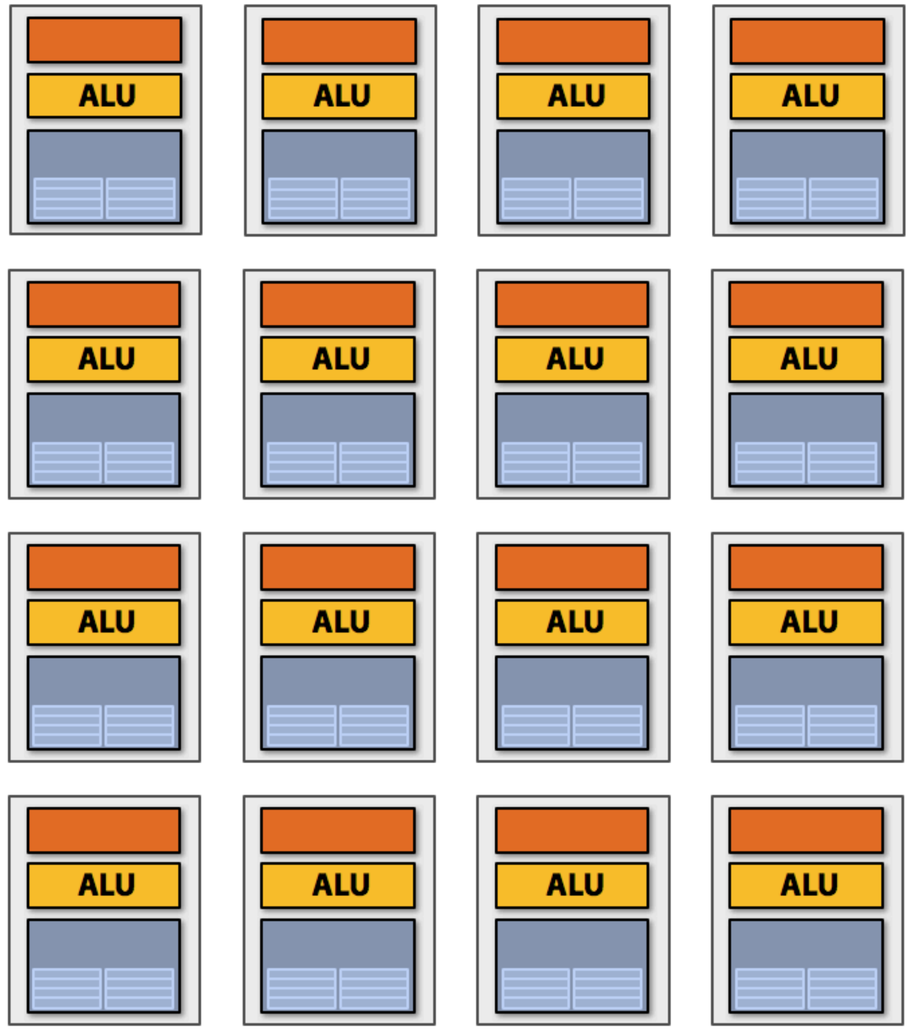
Four cores (four fragments in parallel)





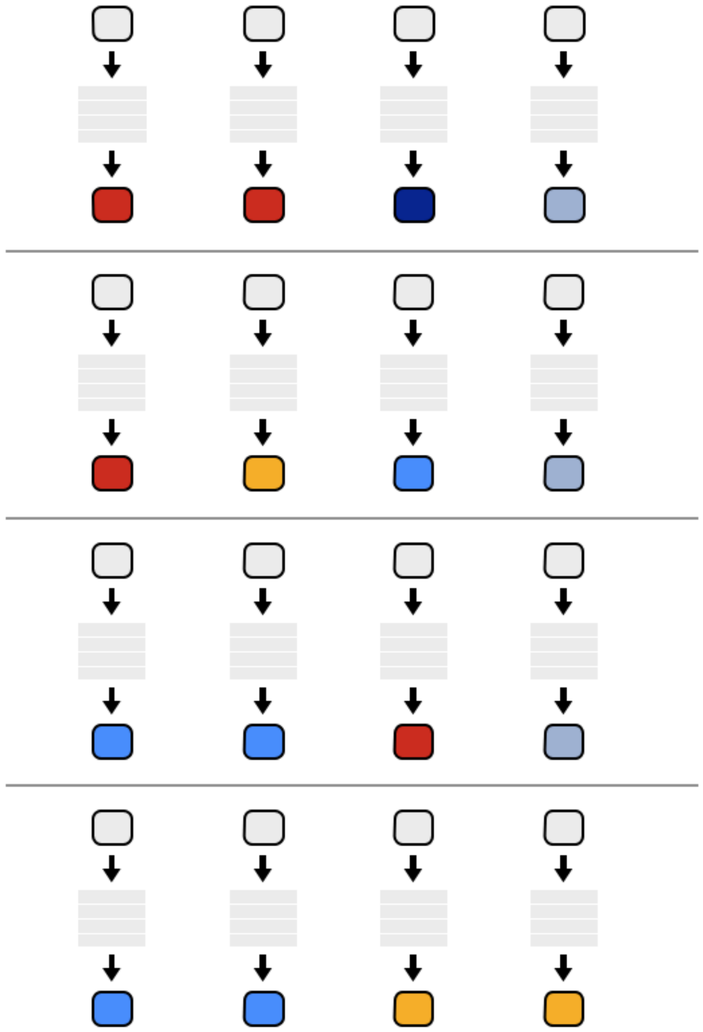
Sixteen cores (sixteen fragments in parallel)





16 cores = 16 simultaneous instruction streams

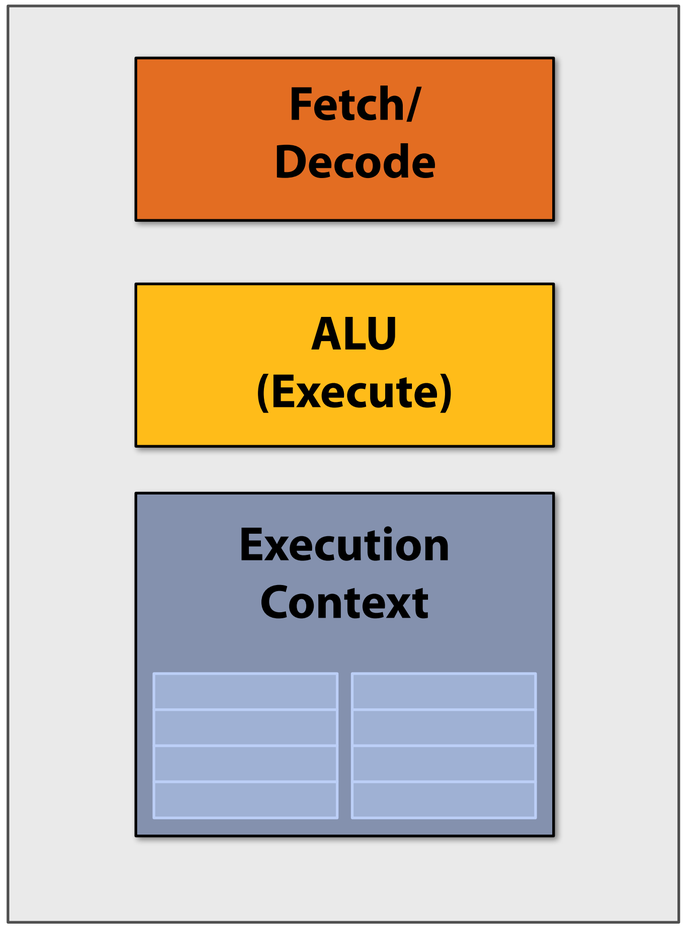
Instruction stream sharing



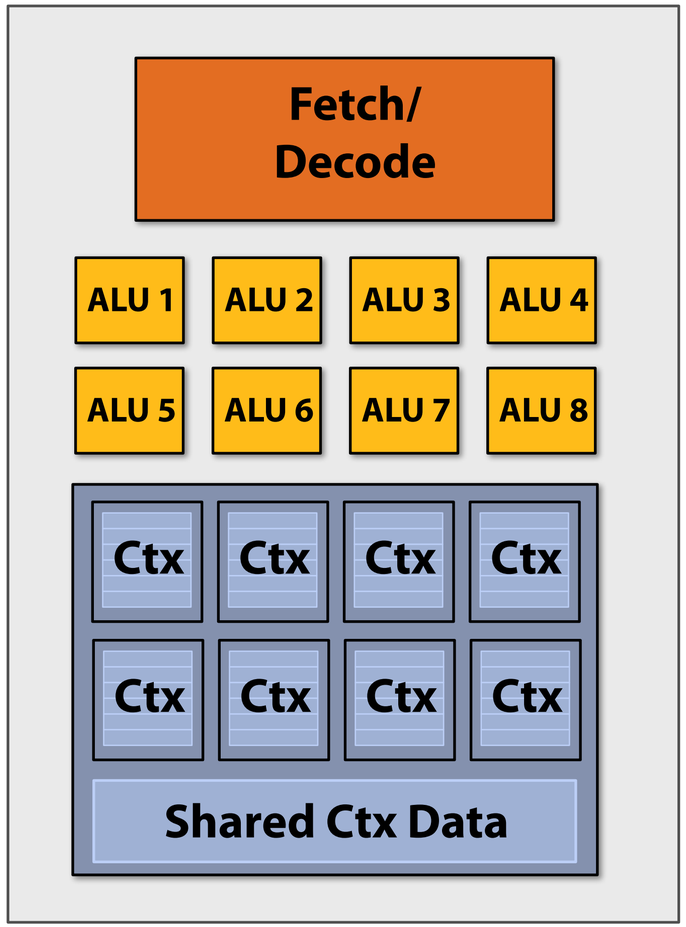
But ... many fragments should be able to share an instruction stream!

```
<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



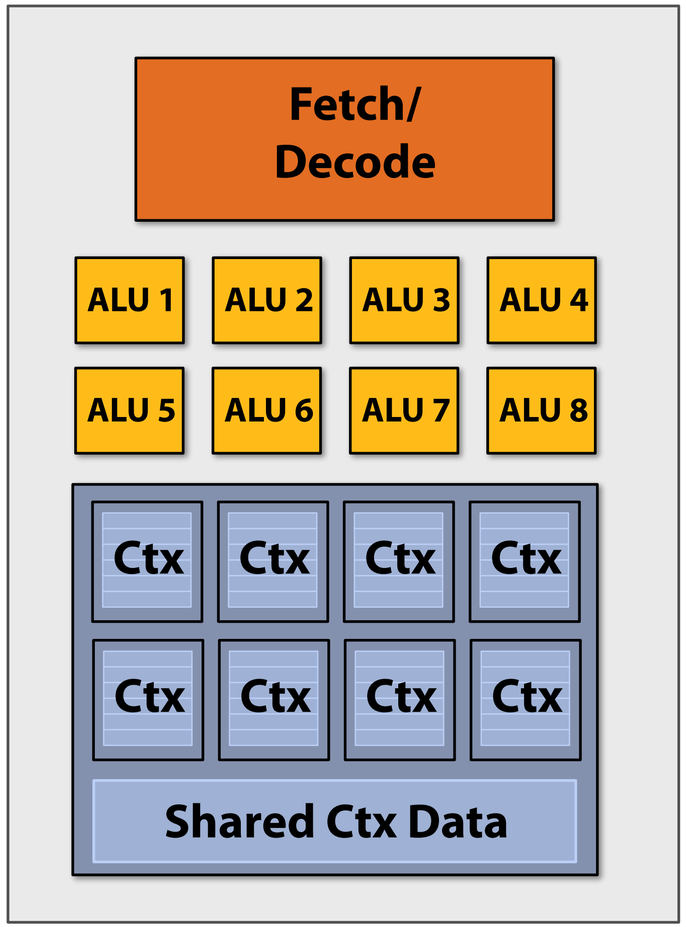
Add ALUs



Idea #2:
Amortize cost/complexity of managing an instruction stream across many ALUs

SIMD processing

Modifying the shader

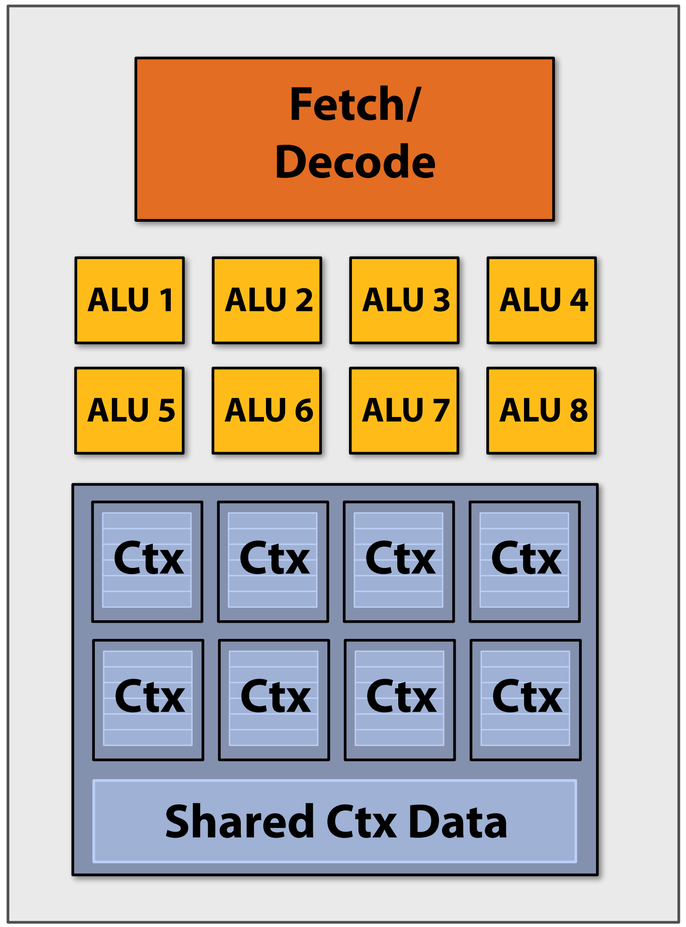



```
<diffuseShader>:
sample r0, v4, t0, s0
mul  r3, v0, cb0[0]
madd r3, v1, cb0[1], r3
madd r3, v2, cb0[2], r3
clmp r3, r3, l(0.0), l(1.0)
mul  o0, r0, r3
mul  o1, r1, r3
mul  o2, r2, r3
mov  o3, l(1.0)
```

Original compiled shader:

Processes one fragment using scalar ops on scalar registers

Modifying the shader

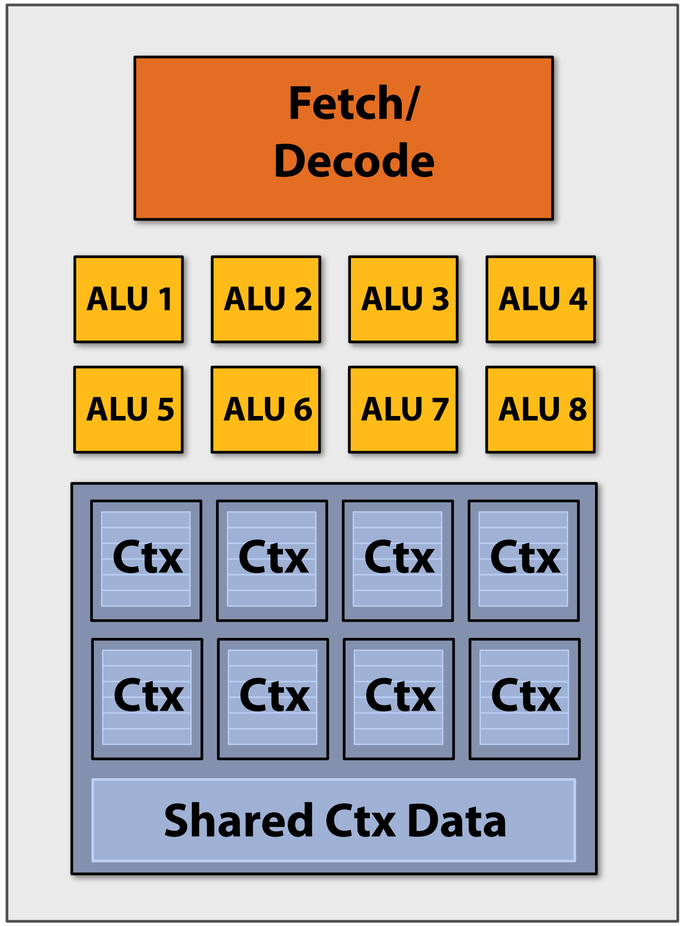


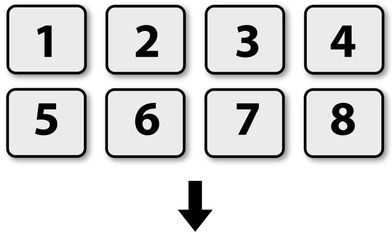

```
<VEC8_diffuseShader>:
VEC8_sample vec_r0, vec_v4, t0, vec_s0
VEC8_mul    vec_r3, vec_v0, cb0[0]
VEC8_madd   vec_r3, vec_v1, cb0[1], vec_r3
VEC8_madd   vec_r3, vec_v2, cb0[2], vec_r3
VEC8_clmp   vec_r3, vec_r3, l(0.0), l(1.0)
VEC8_mul    vec_o0, vec_r0, vec_r3
VEC8_mul    vec_o1, vec_r1, vec_r3
VEC8_mul    vec_o2, vec_r2, vec_r3
VEC8_mov    o3, l(1.0)
```

New compiled shader:

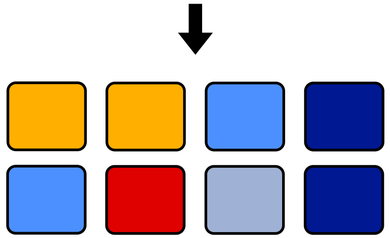
Processes eight fragments using vector ops on vector registers

Modifying the shader

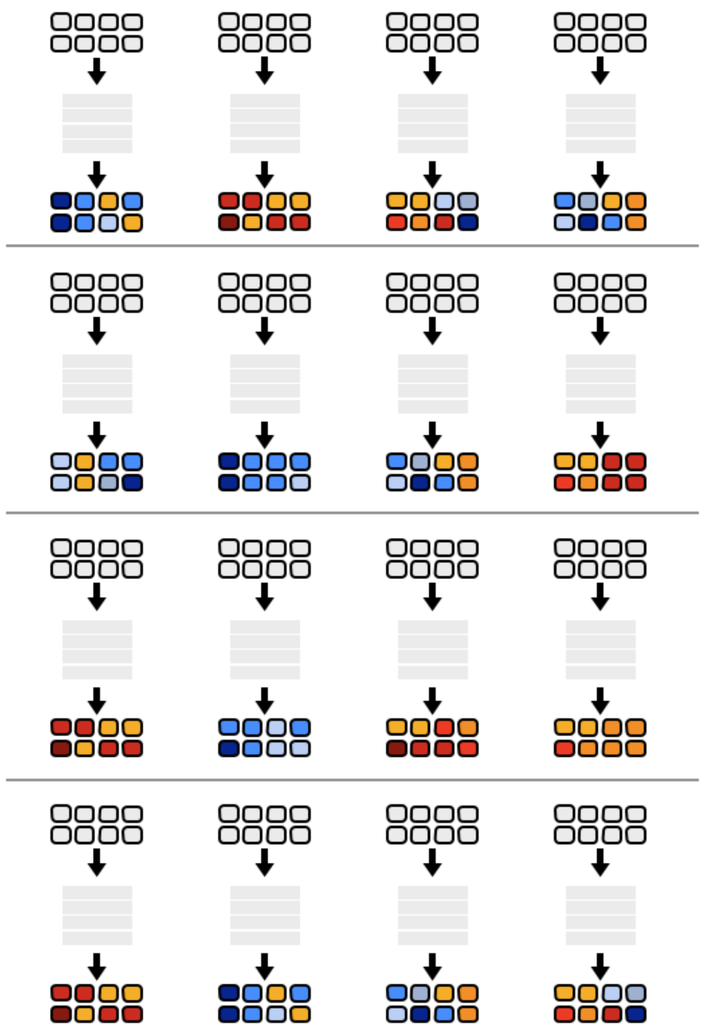


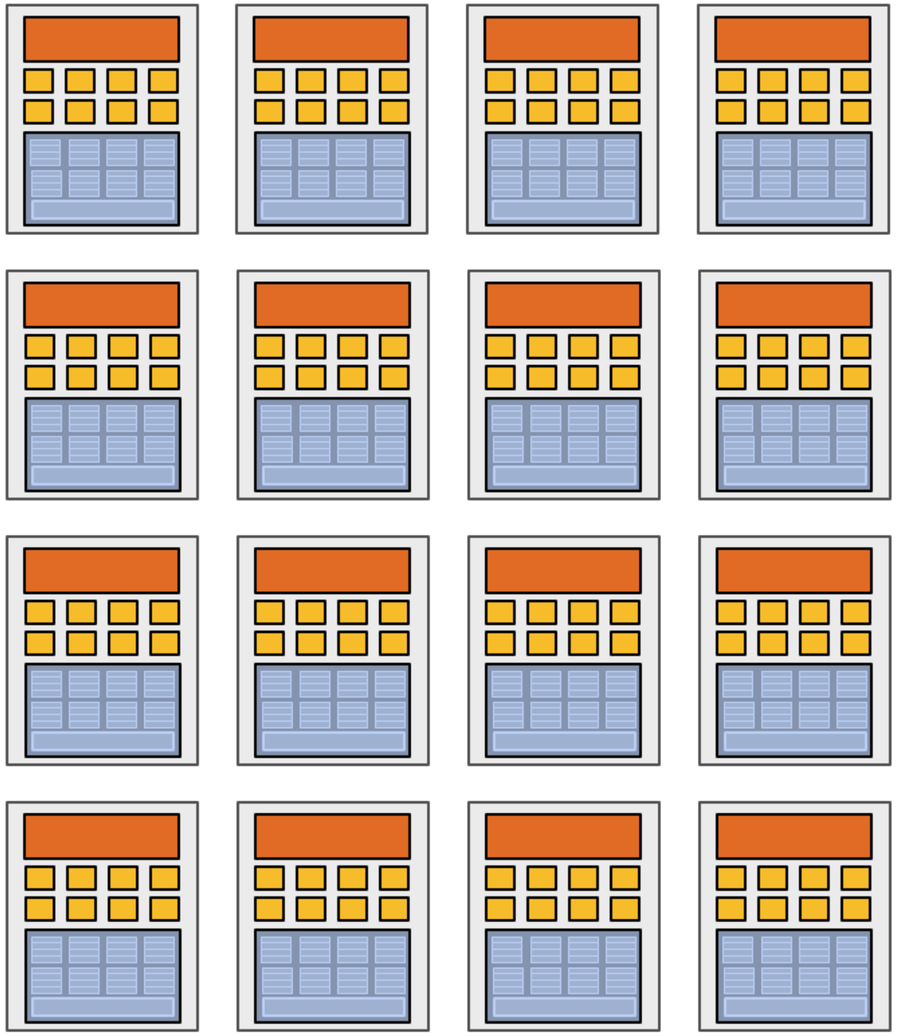



```
<VEC8_diffuseShader>:
VEC8_sample vec_r0, vec_v4, t0, vec_s0
VEC8_mul  vec_r3, vec_v0, cb0[0]
VEC8_madd vec_r3, vec_v1, cb0[1], vec_r3
VEC8_madd vec_r3, vec_v2, cb0[2], vec_r3
VEC8_clmp vec_r3, vec_r3, l(0.0), l(1.0)
VEC8_mul  vec_o0, vec_r0, vec_r3
VEC8_mul  vec_o1, vec_r1, vec_r3
VEC8_mul  vec_o2, vec_r2, vec_r3
VEC8_mov  o3, l(1.0)
```



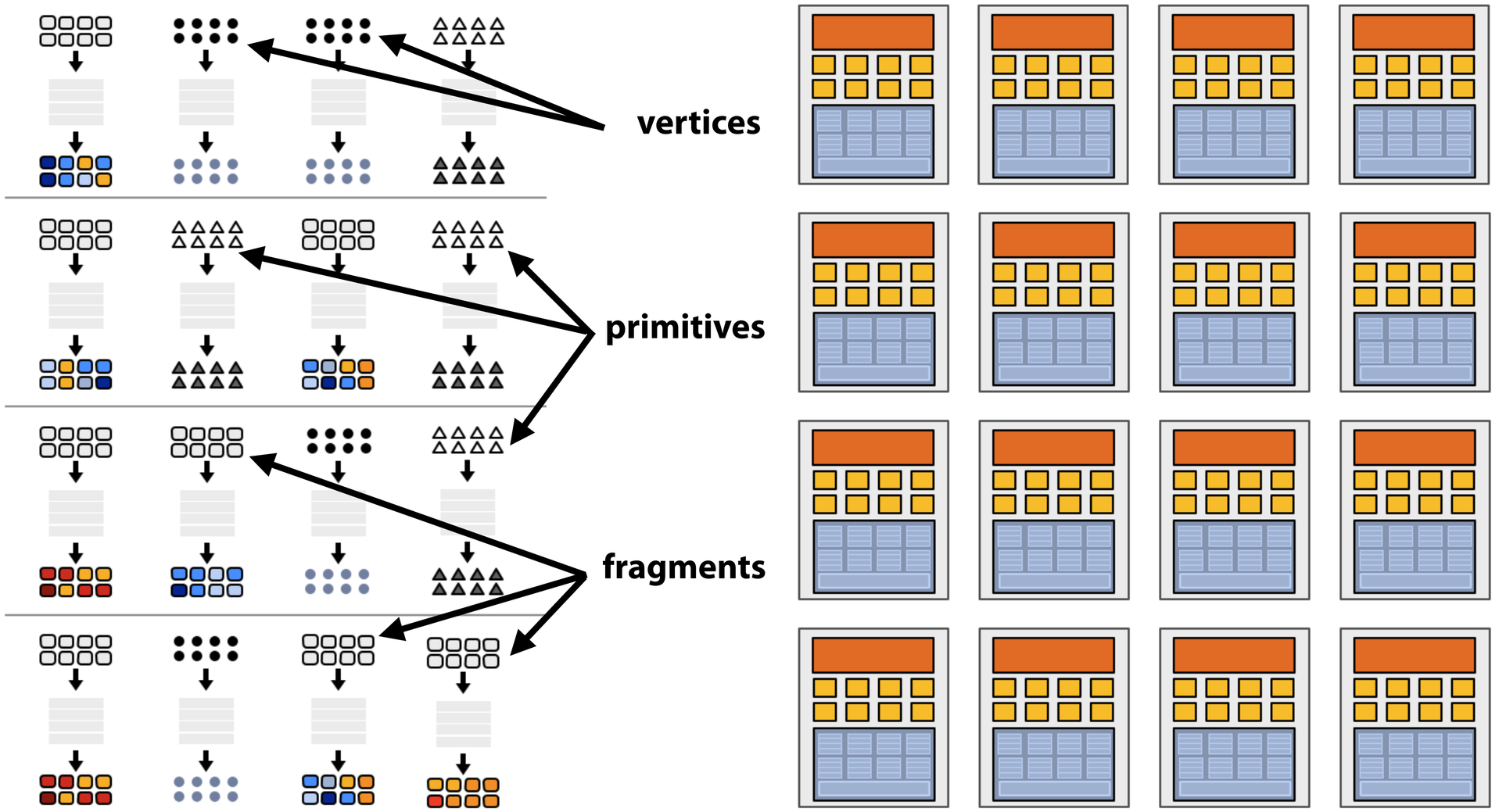
128 fragments in parallel



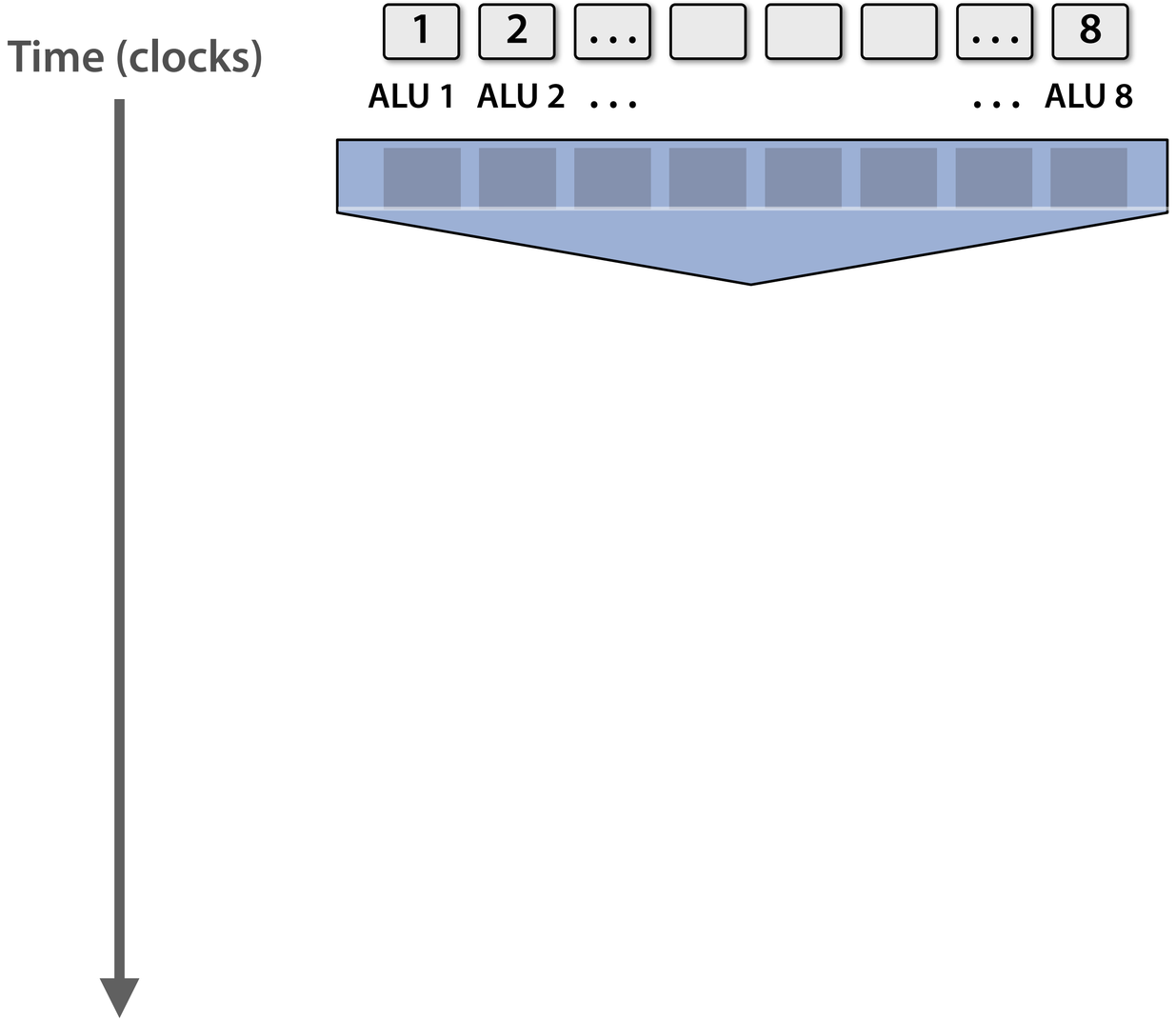


16 cores = 128 ALUs, 16 simultaneous instruction streams

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

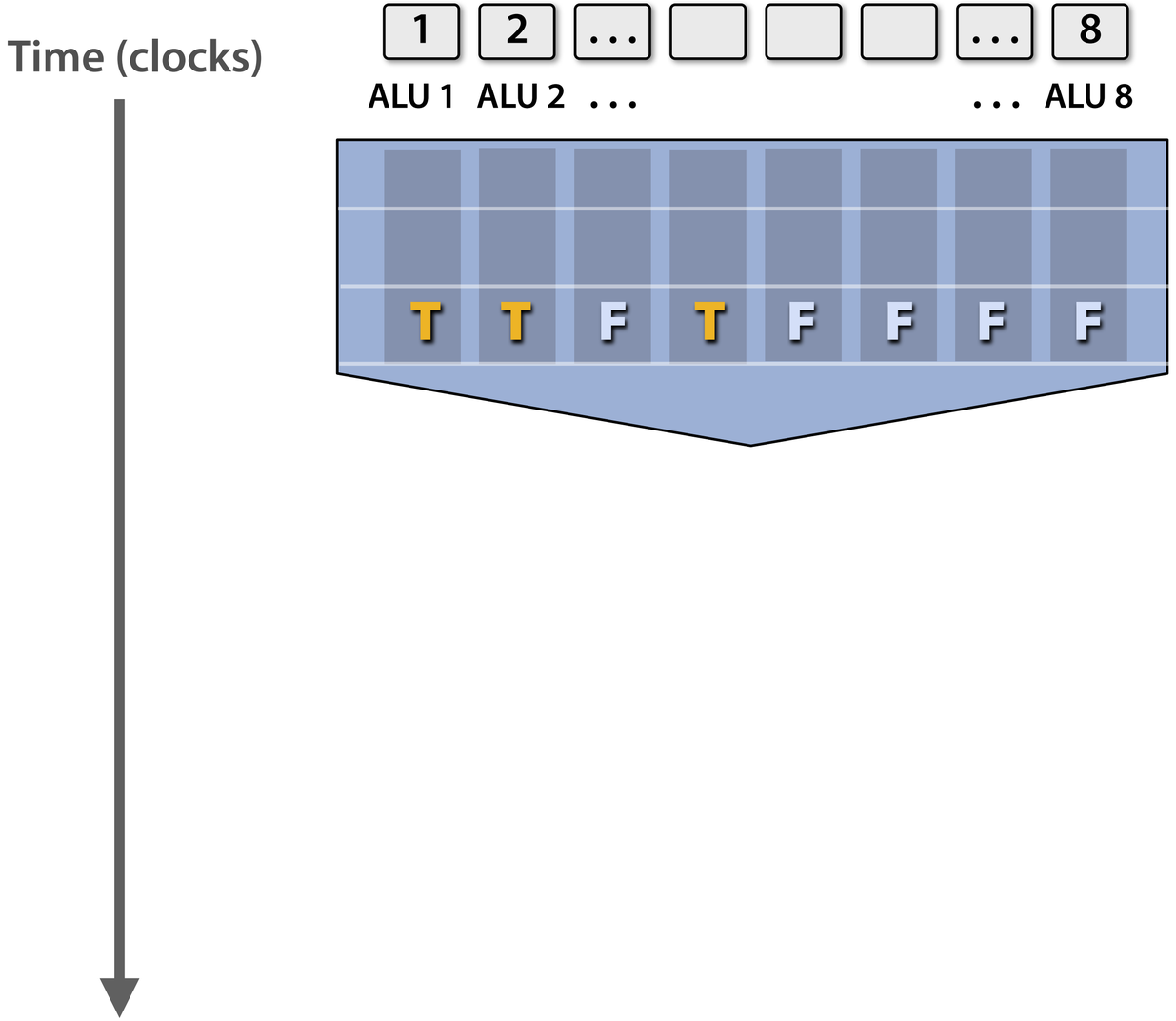


But what about branches?



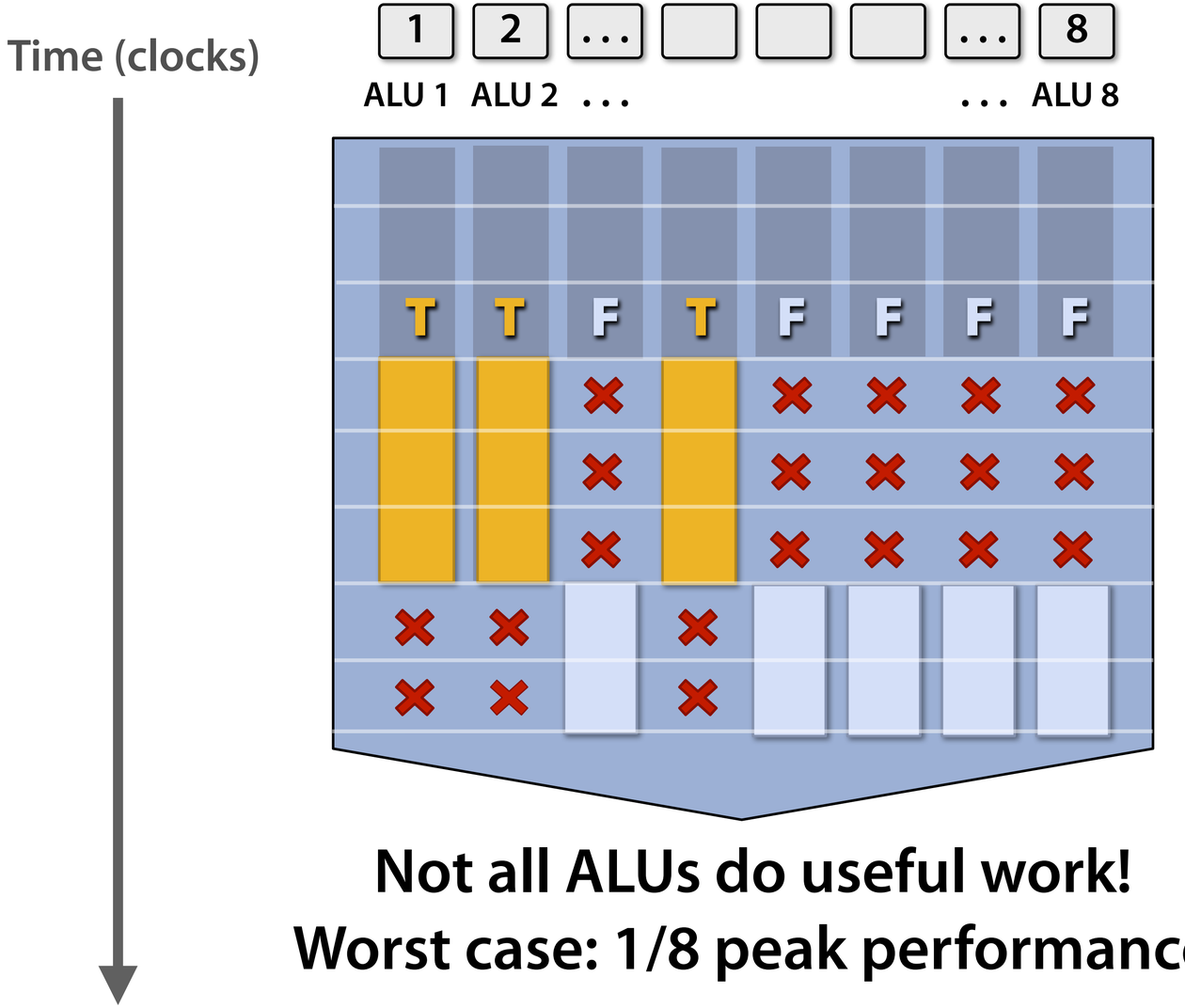

```
<unconditional
 shader code>

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?

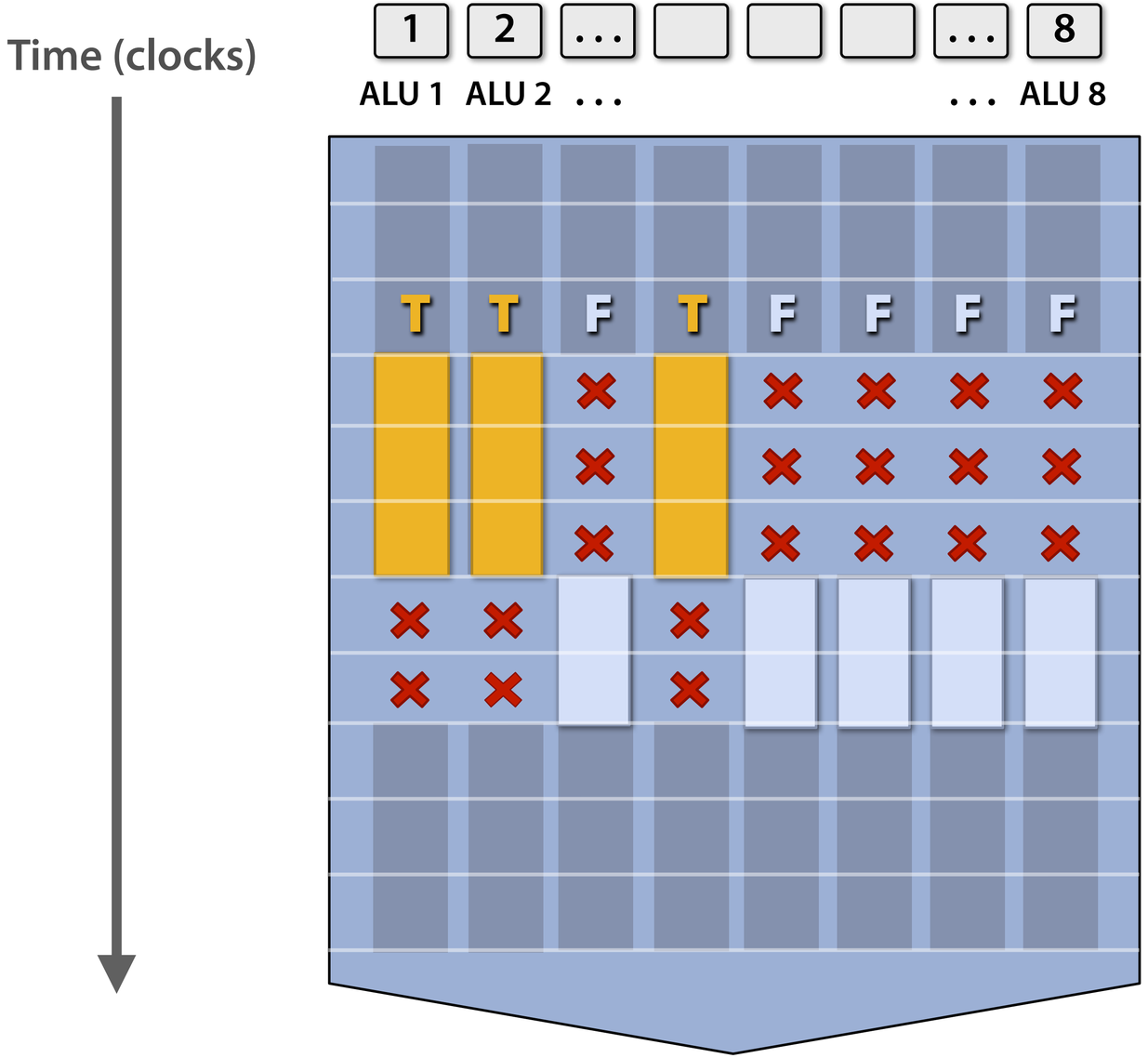



```
<unconditional
 shader code>

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 peak 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>
```

Terminology

- **"Coherent" execution*** (admittedly fuzzy definition): when processing of different entities is similar, and thus can share resources for efficient execution**
  - Instruction stream coherence: different fragments follow same sequence of logic
  - Memory access coherence:
    - Different fragments access similar data (avoid memory transactions by reusing data in cache)
    - Different fragments simultaneously access contiguous data (enables efficient, bulk granularity memory transactions)

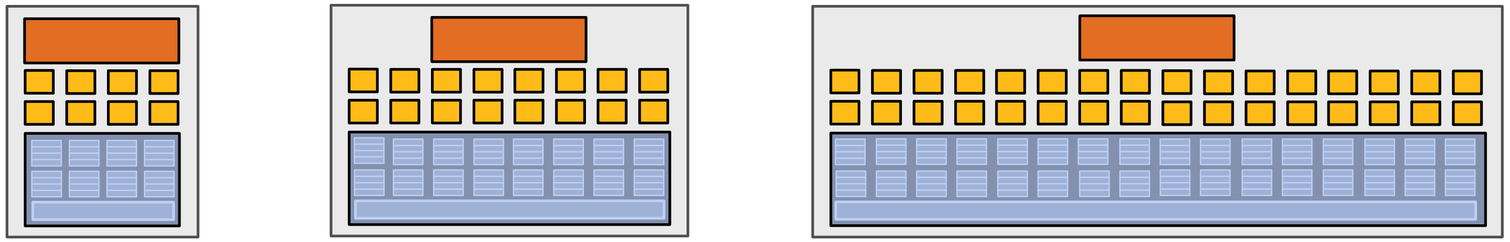
- **"Divergence": lack of coherence**
  - Usually used in reference to instruction streams (divergent execution does not make full use of SIMD processing)

*** Do not confuse this use of term "coherence" with cache coherence protocols

Clarification

SIMD processing does not imply SIMD instructions in an ISA

- Option 1: explicit vector instructions
  - x86 SSE (4-wide), Intel AVX (8-wide), Intel Larrabee (16-wide)
- Option 2: scalar instructions, implicit HW vectorization
  - HW determines instruction stream sharing across ALUs (amount of sharing hidden from software)
  - NVIDIA GeForce ("SIMT" warps), ATI Radeon architectures ("wavefronts")



In modern GPUs: 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.

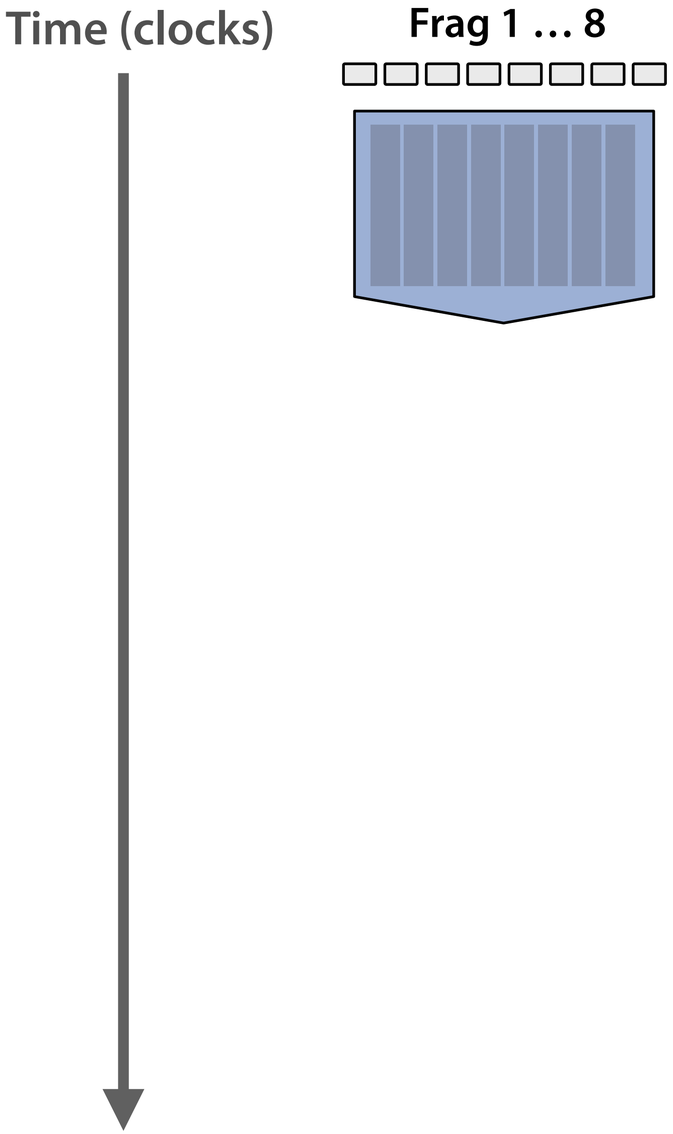
But we have LOTS of independent fragments.

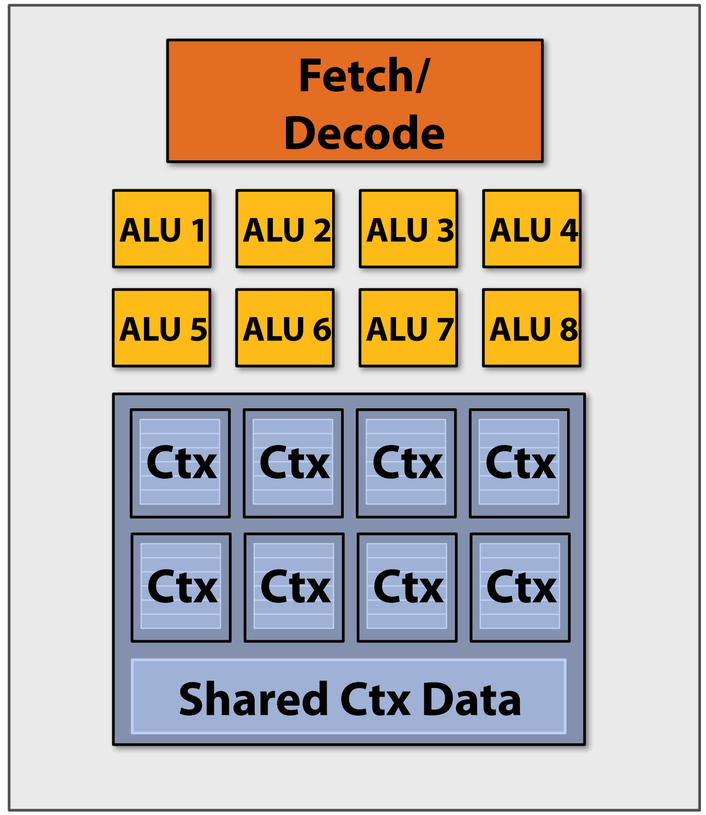
(Way more fragments to process than ALUs)

Idea #3:

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

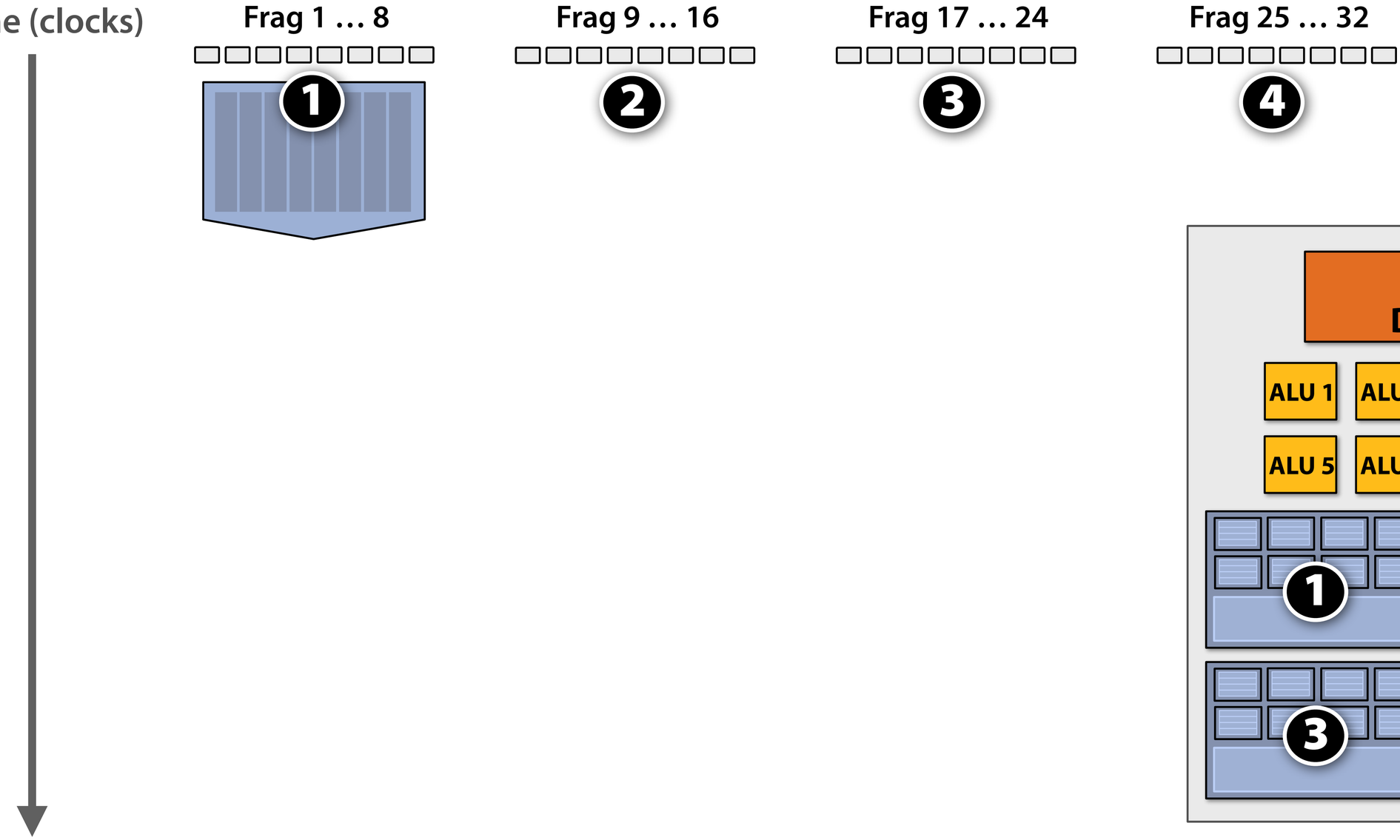
Hiding shader stalls

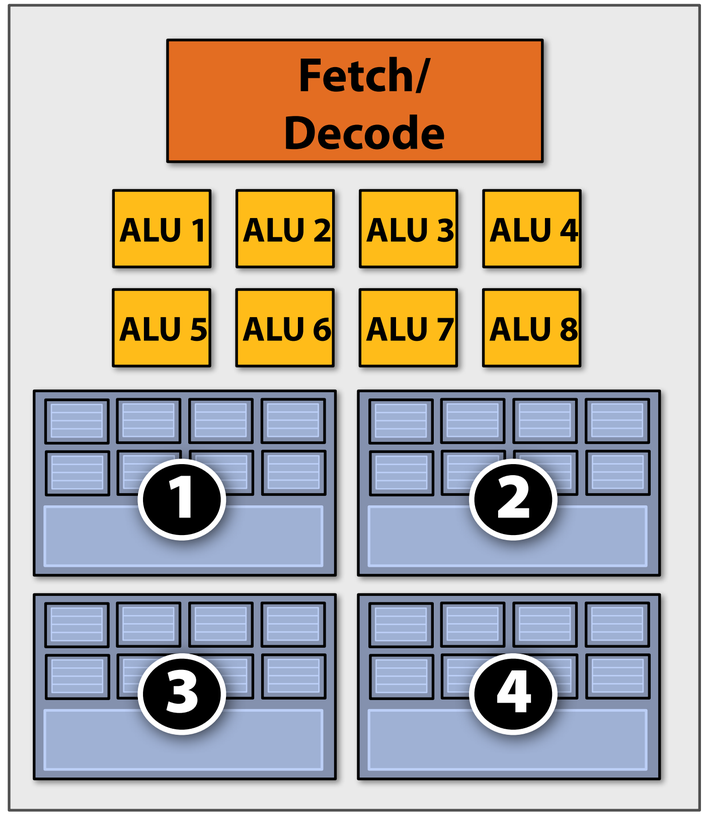




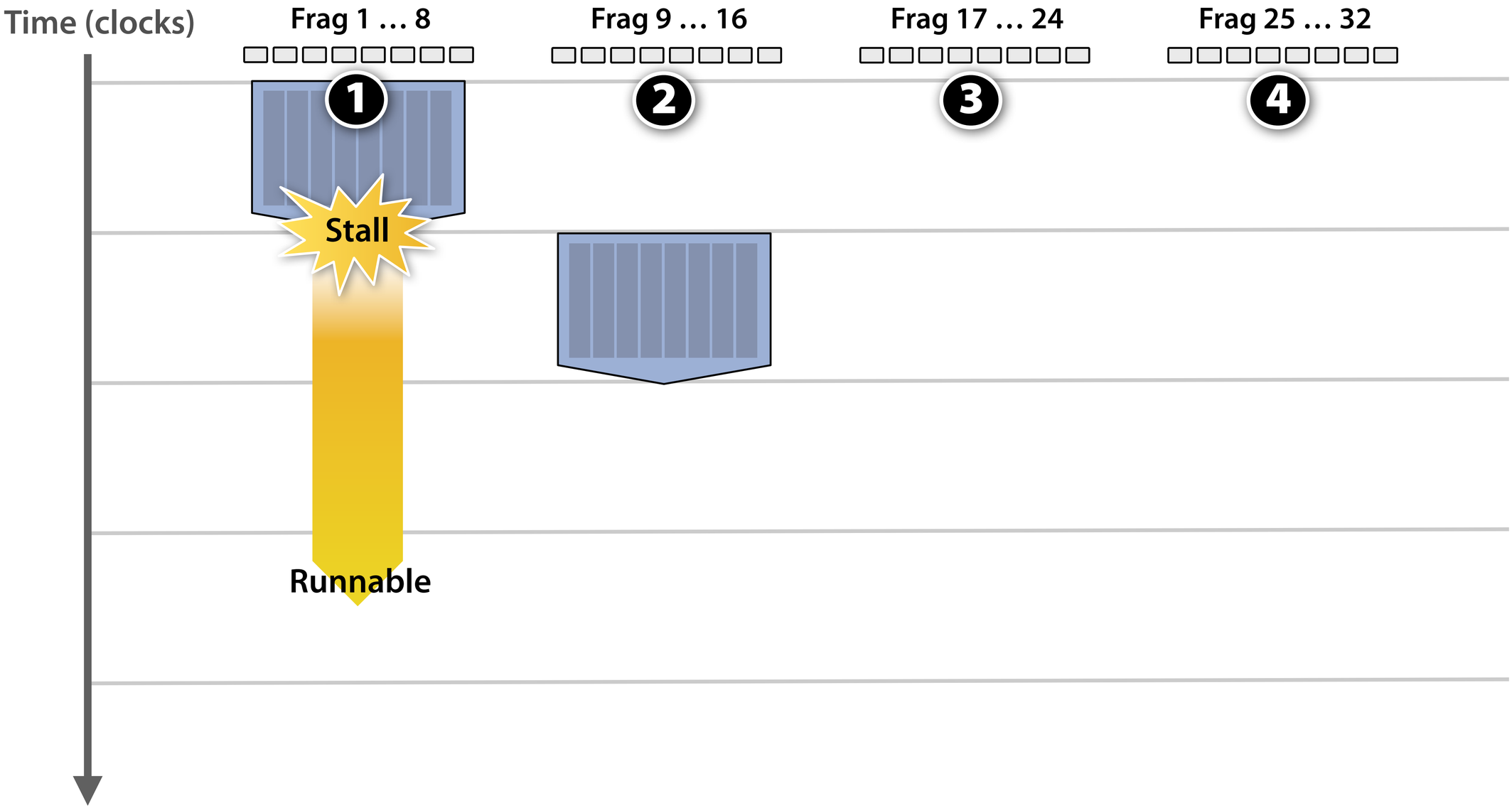
Hiding shader stalls

Time (clocks)

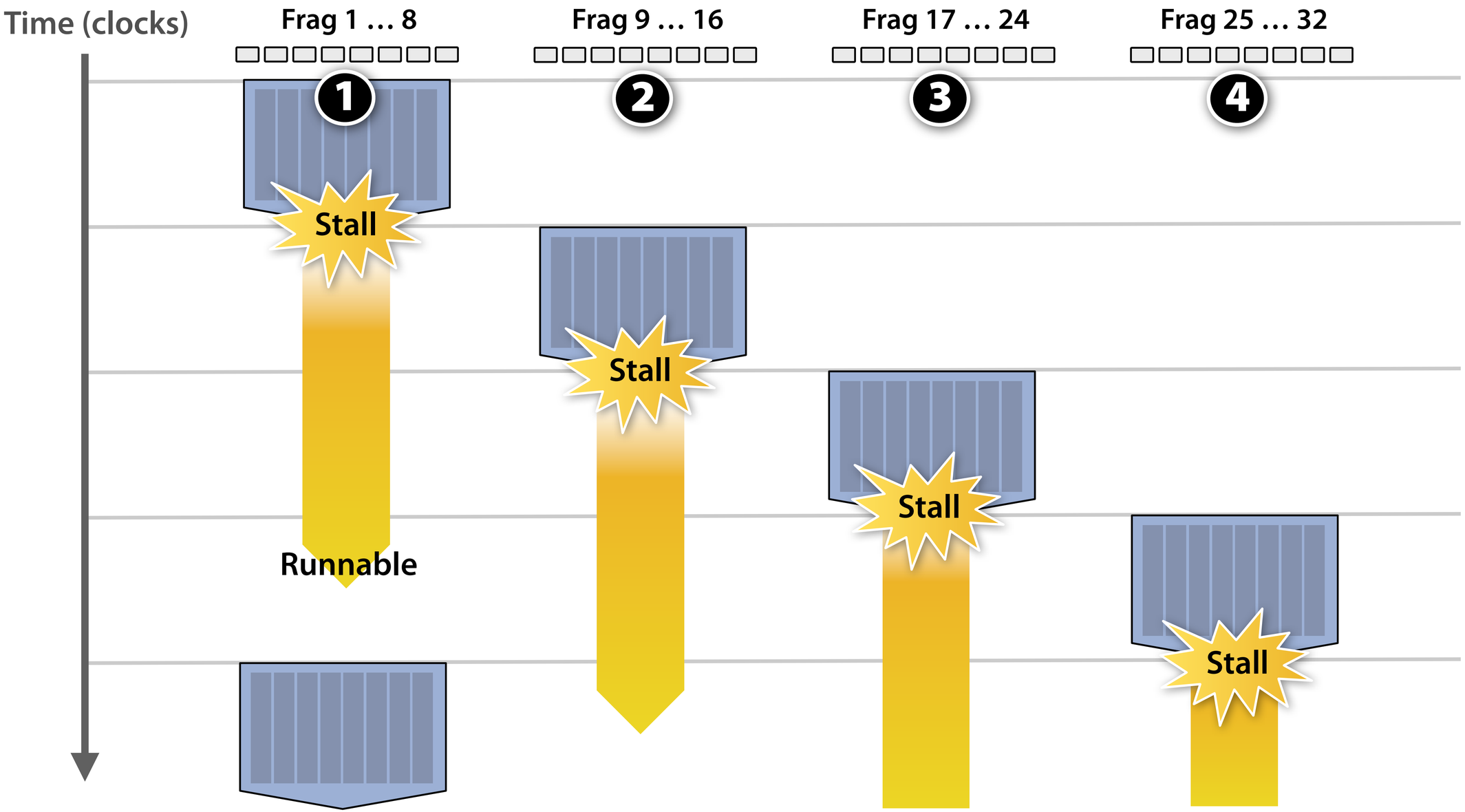




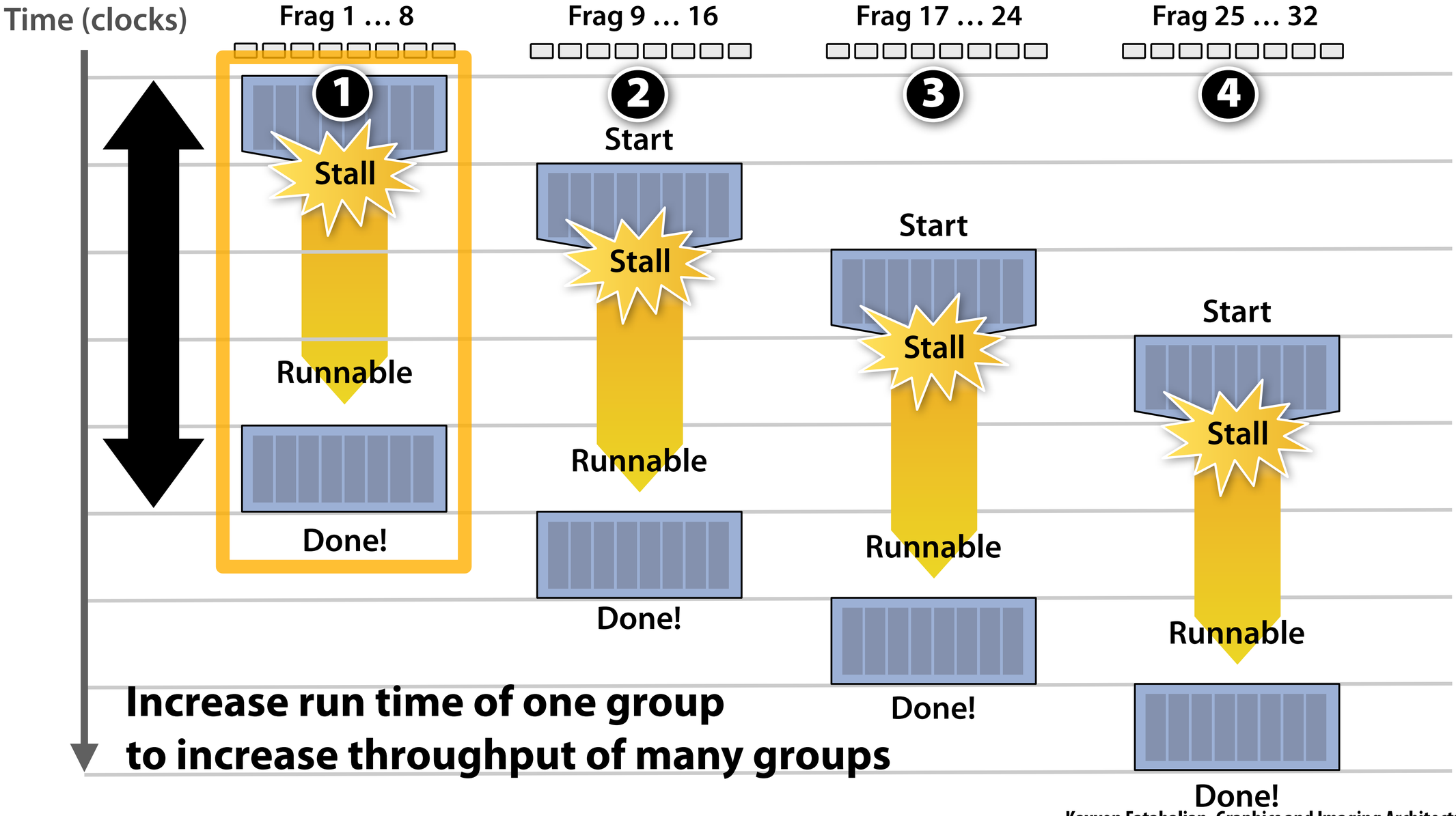
Hiding shader stalls



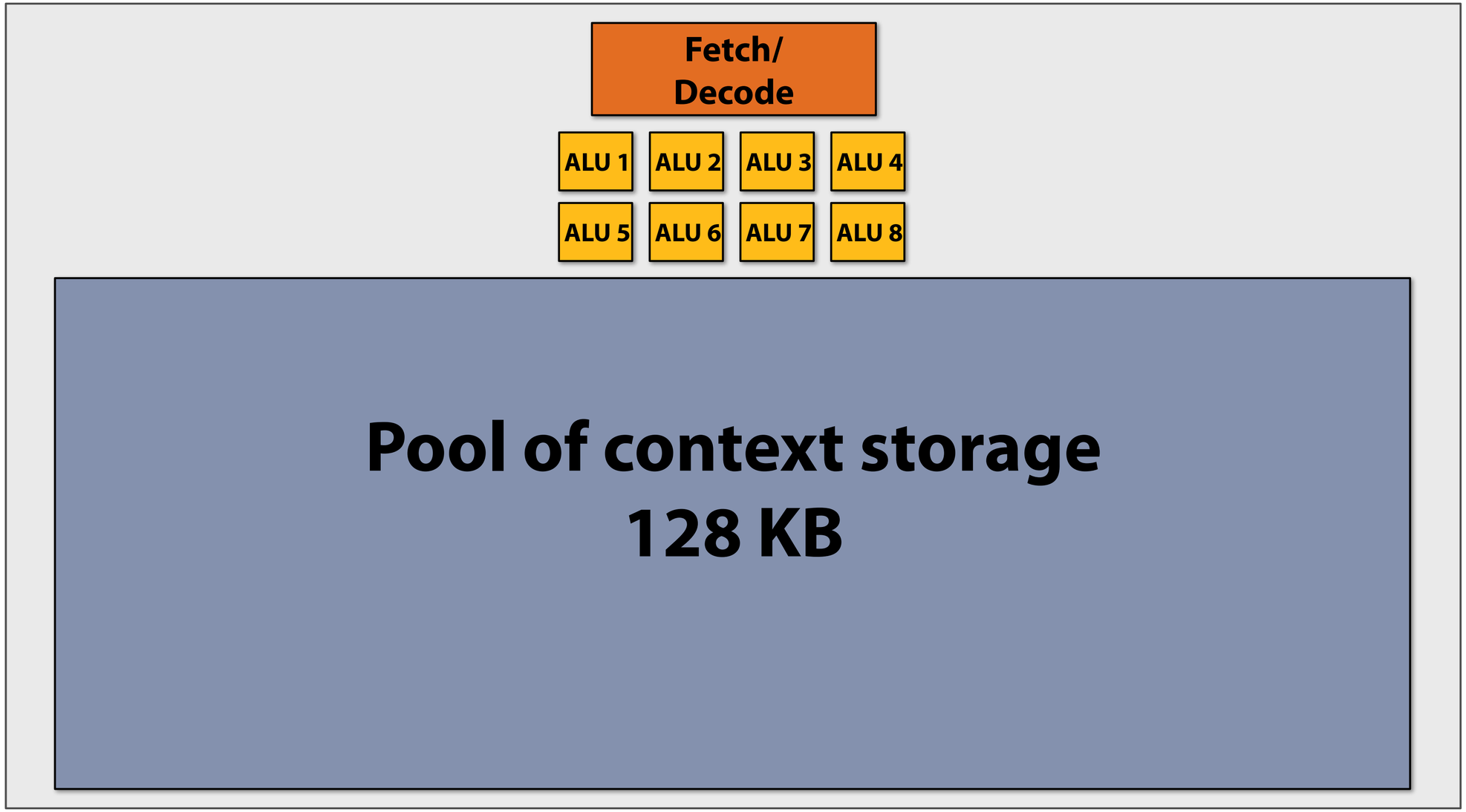
Hiding shader stalls



Throughput!

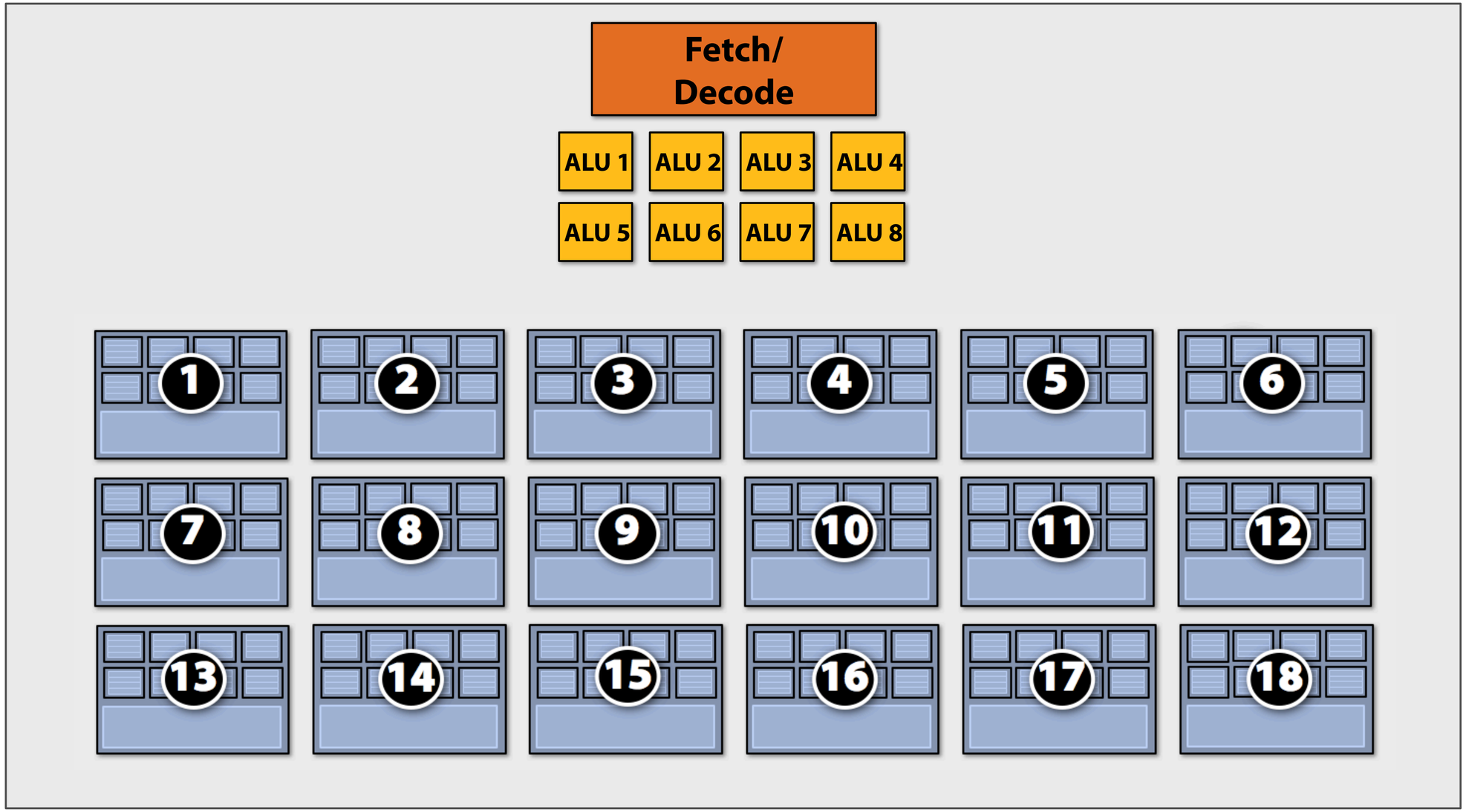

Time (clocks)
Frag 1 … 8
Frag 9 … 16
Frag 17 … 24
Frag 25 … 32
1
2
3
4
Stall
Start
Runnable
Done!
Increase run time of one group
to increase throughput of many groups

Storing contexts

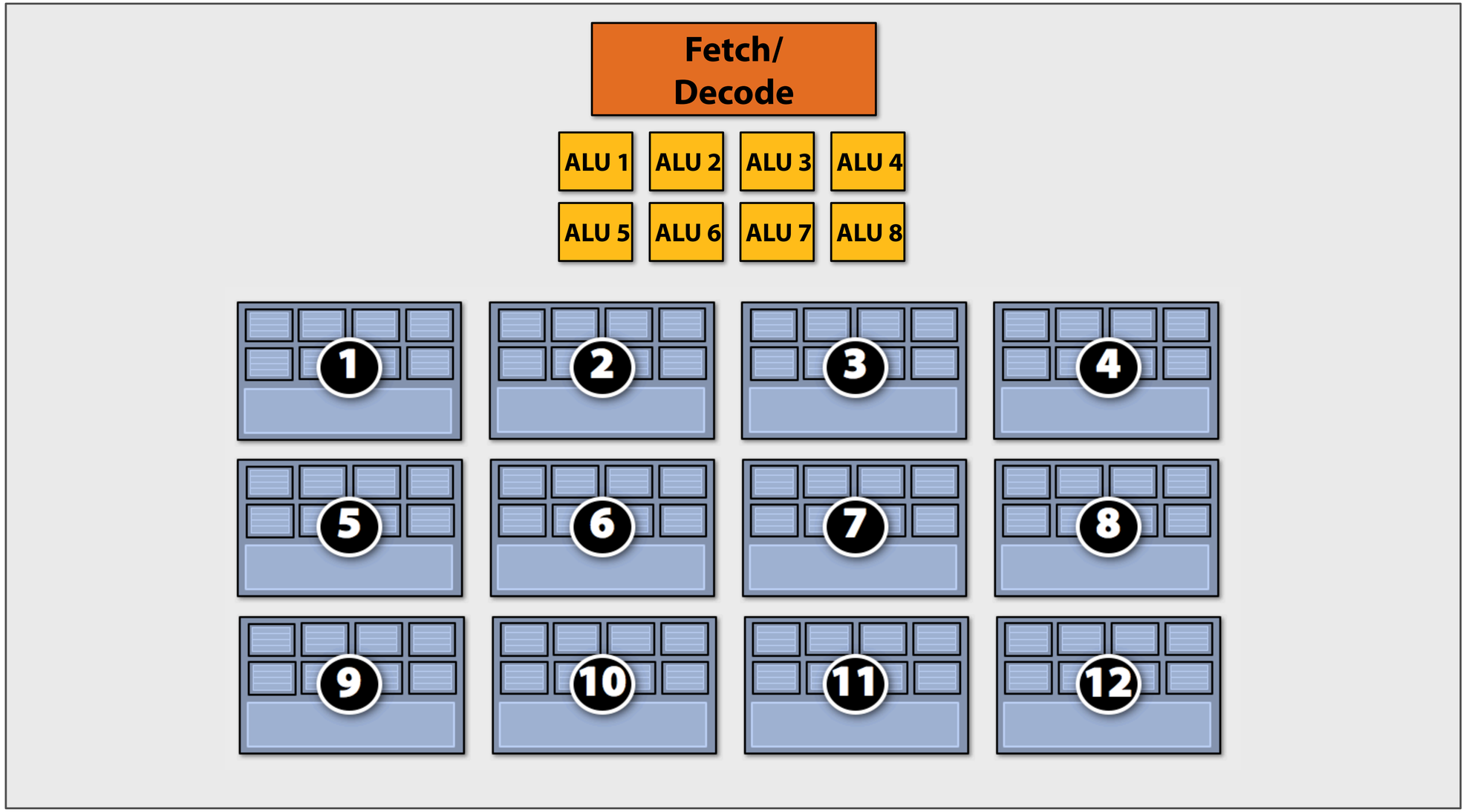


Eighteen small contexts

(maximal latency hiding)

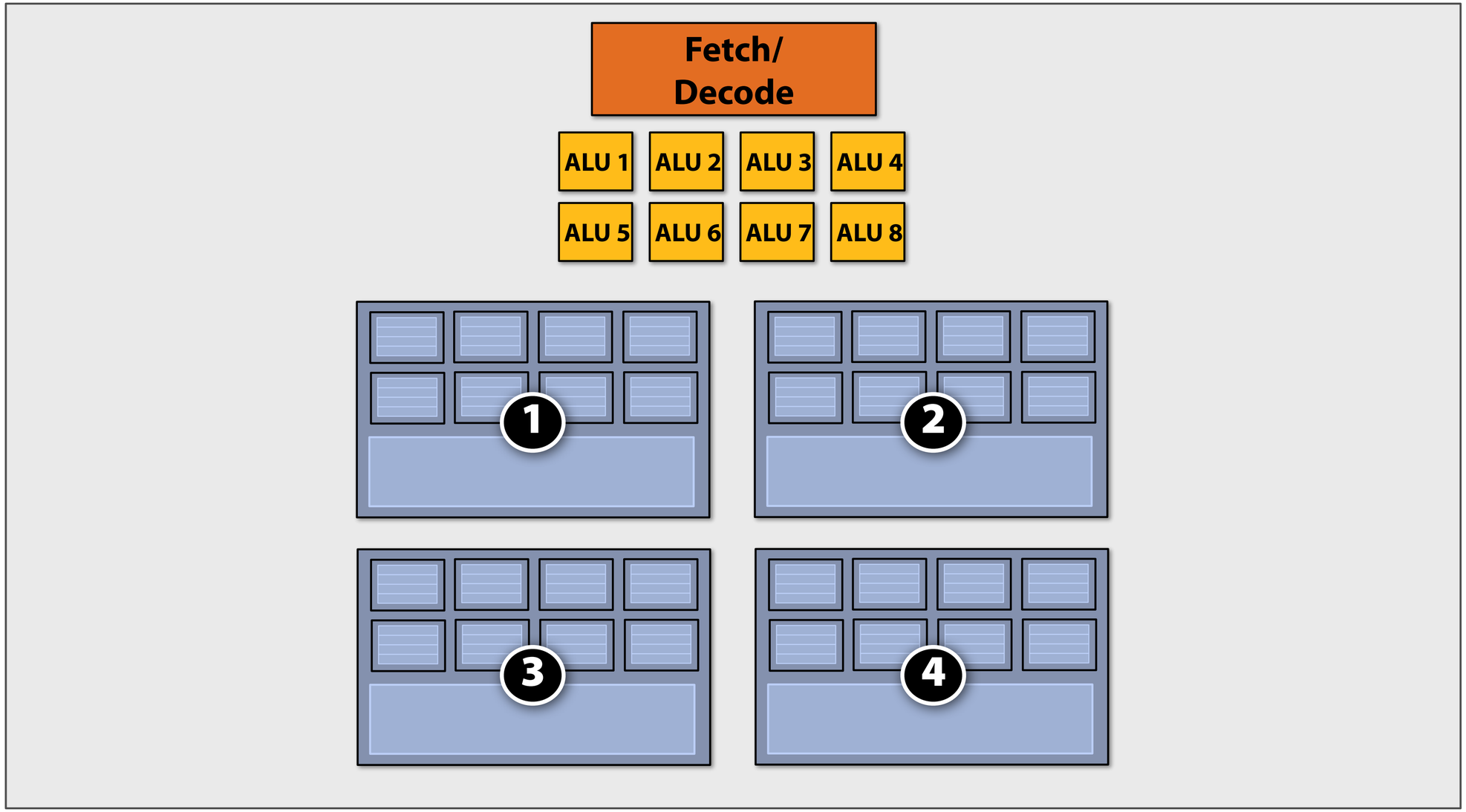


Twelve medium contexts



Four large contexts

(low latency hiding ability)



Clarification

Interleaving between contexts can be managed by hardware or software (or both!)

- NVIDIA / ATI 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

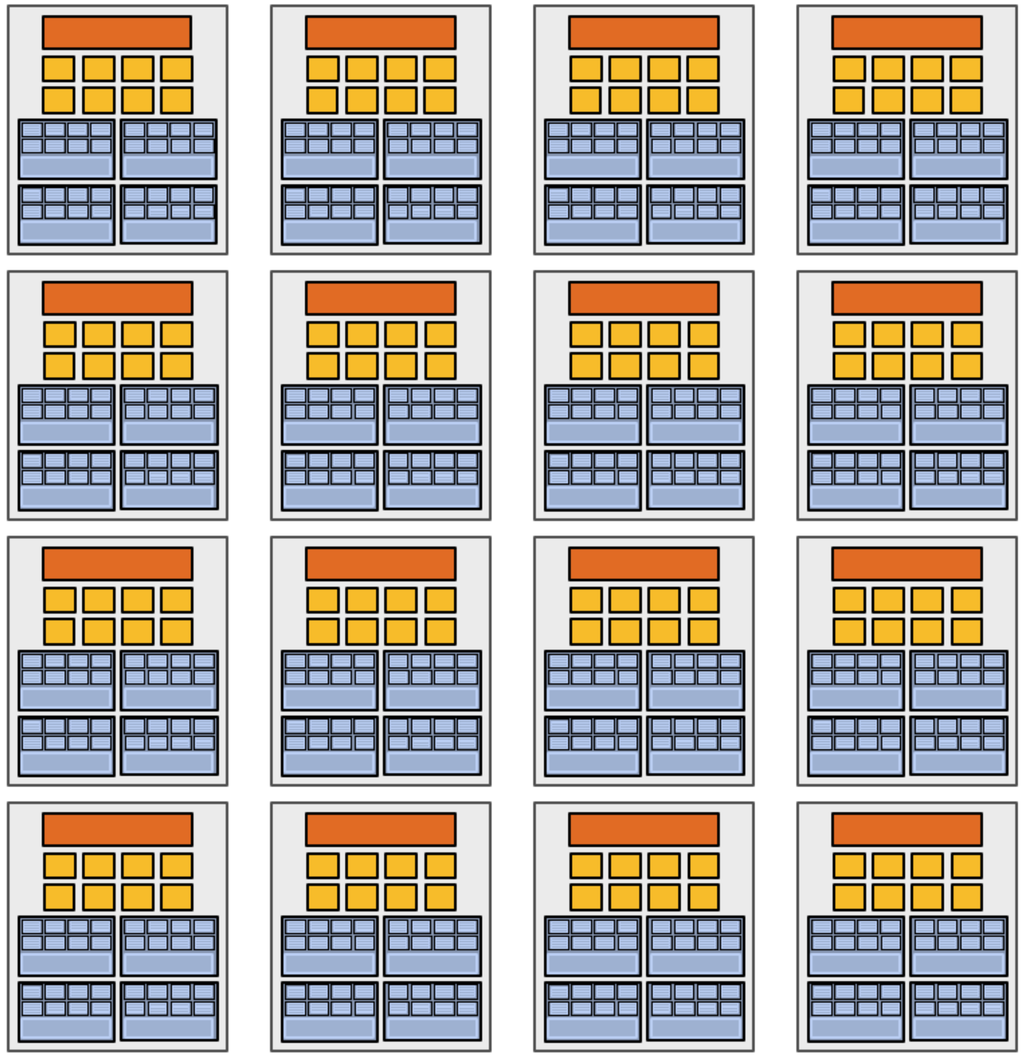
8 mul-add ALUs per core
(128 total)

16 simultaneous
instruction streams

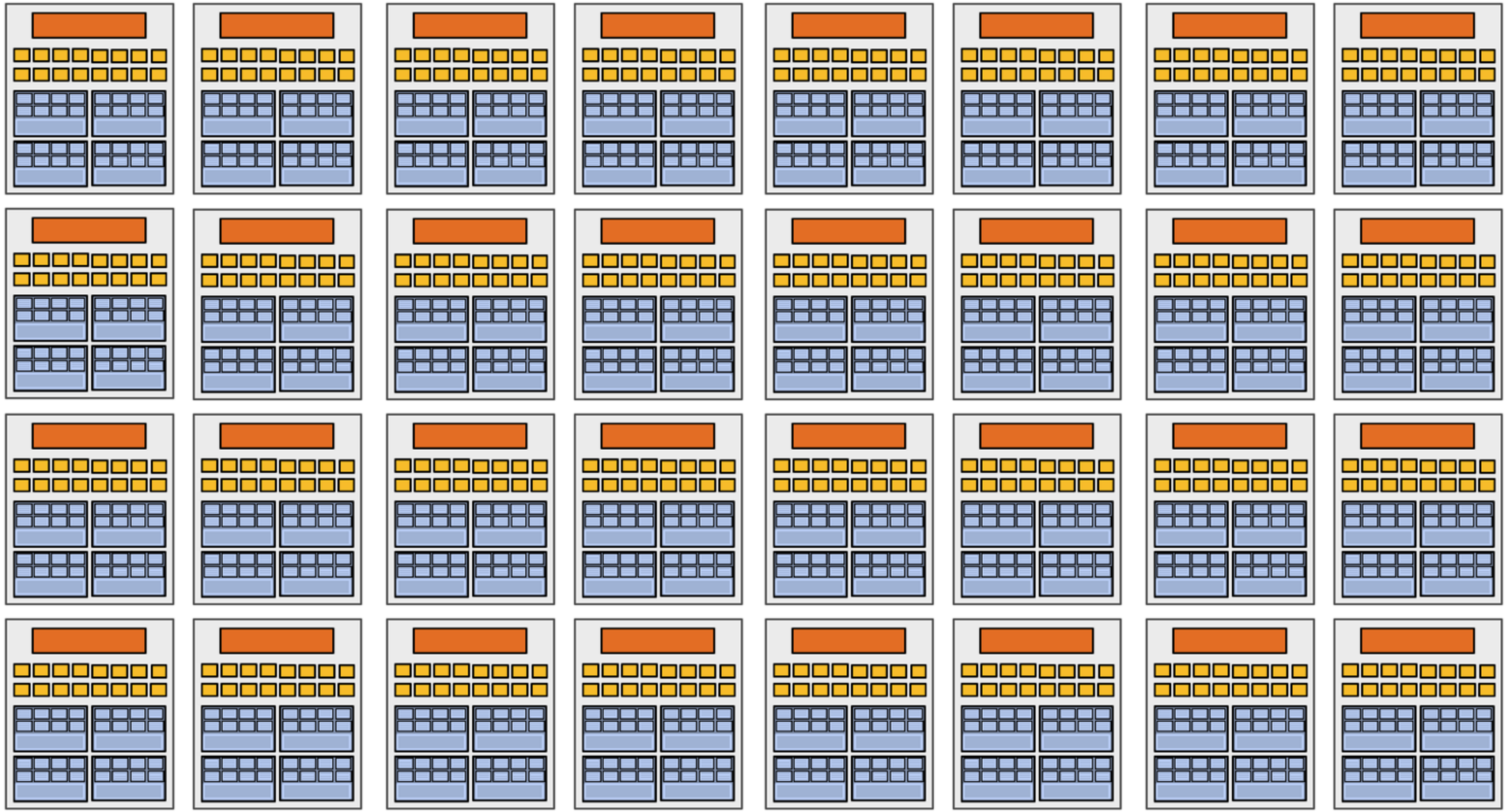
64 concurrent (but interleaved)
instruction streams

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 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

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

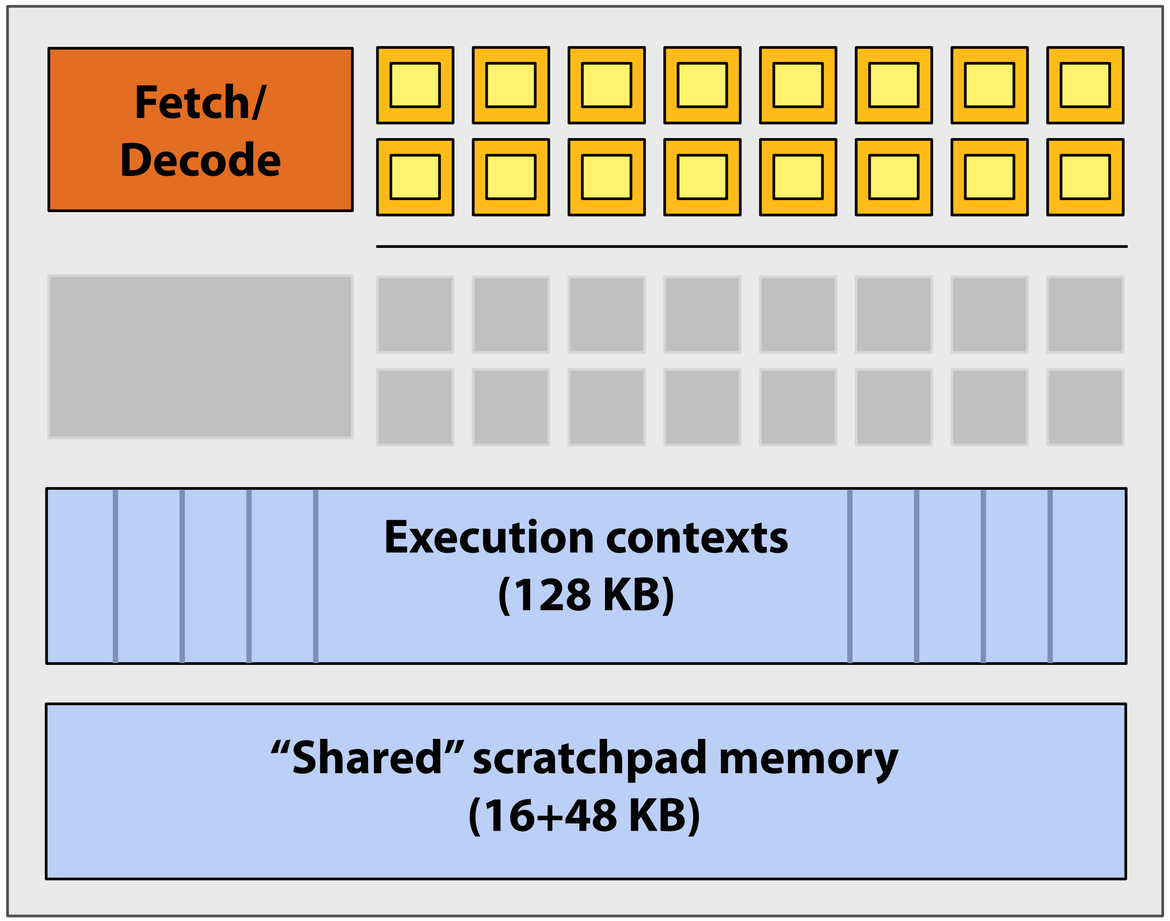
NVIDIA GeForce GTX 480
ATI Radeon HD 5870

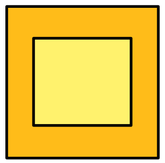
NVIDIA GeForce GTX 480 (Fermi)

- NVIDIA-speak:
  - 480 stream processors (“CUDA cores”)
  - “SIMT execution”

- Generic speak:
  - 15 cores
  - 2 groups of 16 SIMD functional units per core

NVIDIA GeForce GTX 480 “core”



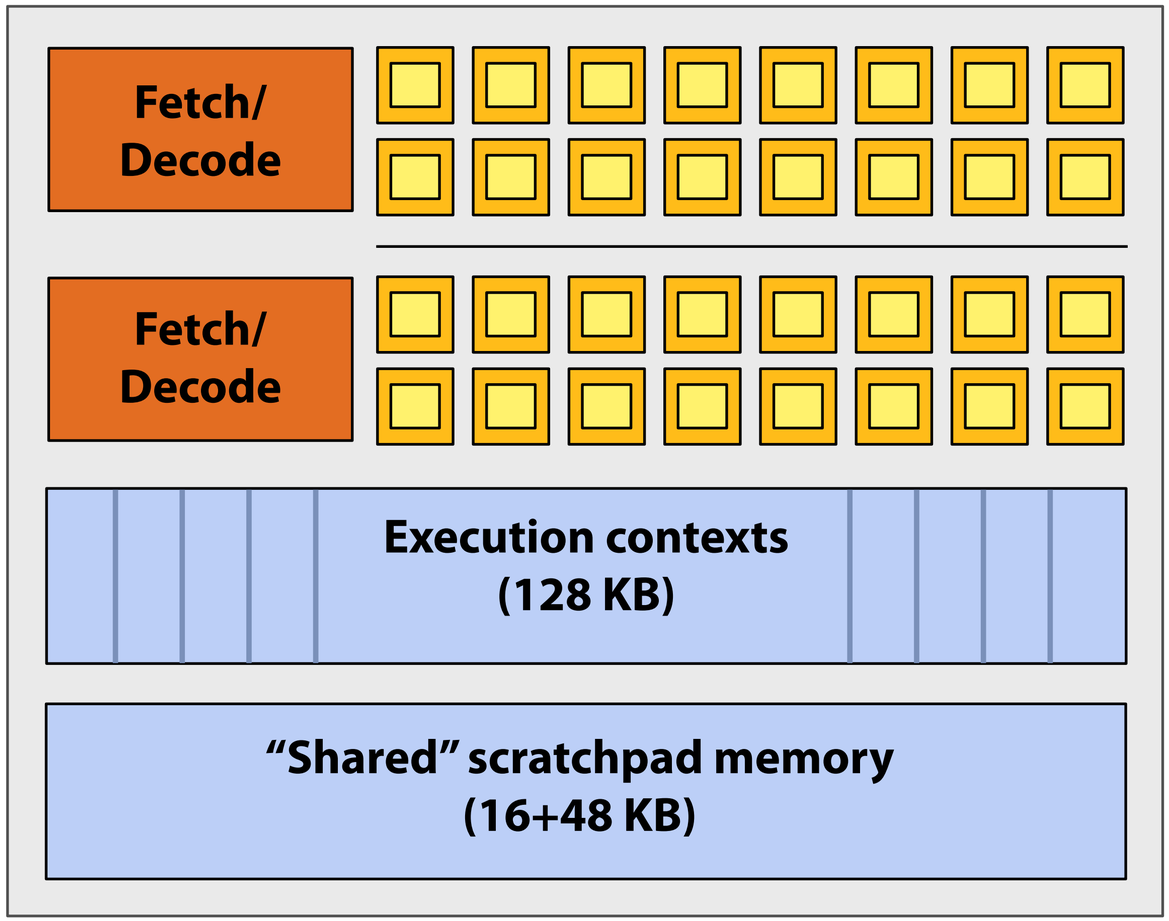


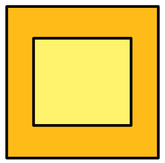
= SIMD function unit,
control shared across 16 units
(1 MUL-ADD per clock)

- Groups of 32 fragments share an instruction stream
- Up to 48 groups are simultaneously interleaved
- Up to 1536 individual contexts can be stored

Source: Fermi Compute Architecture Whitepaper
CUDA Programming Guide 3.1, Appendix G

NVIDIA GeForce GTX 480 “core”



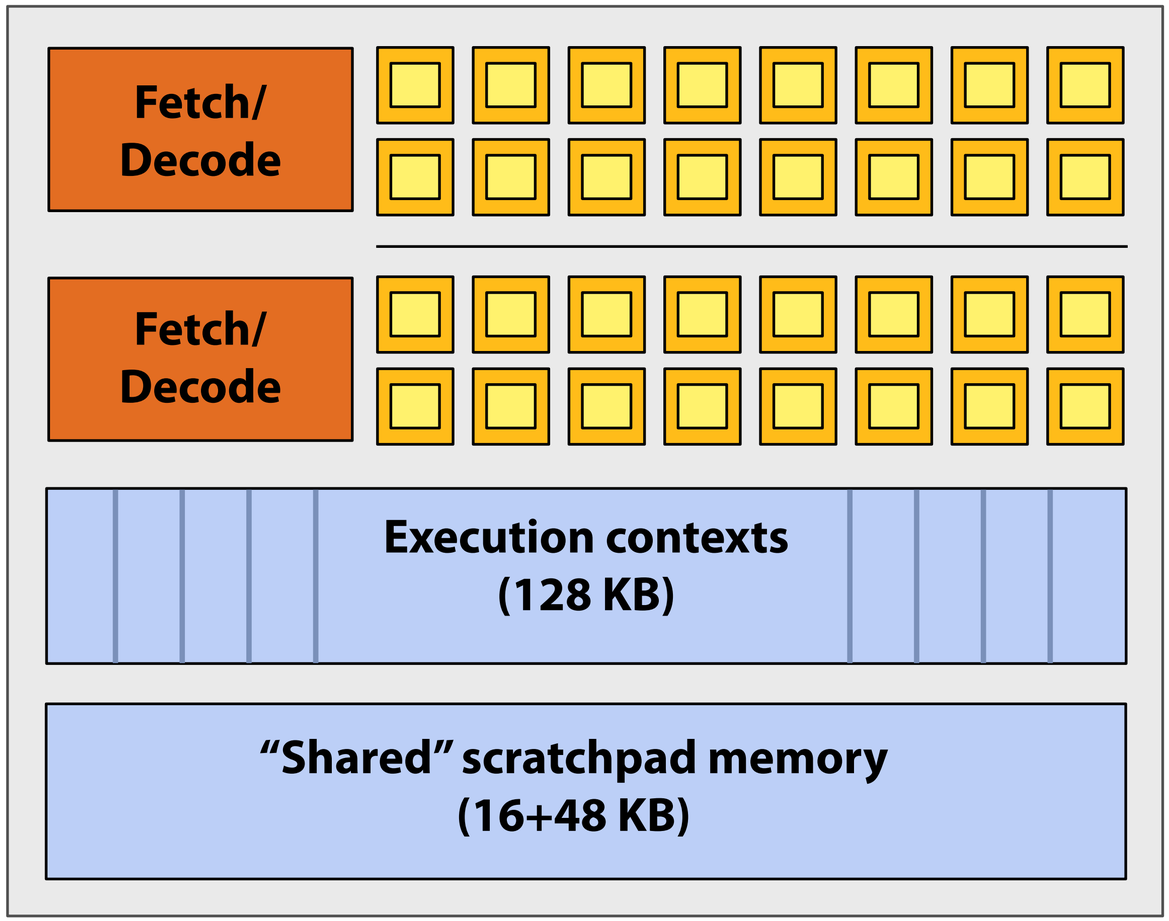


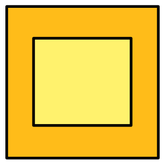
= SIMD function unit,
control shared across 16 units
(1 MUL-ADD per clock)

- The core contains 32 functional units
- Two groups are selected each clock (decode, fetch, and execute two instruction streams in parallel)

Source: Fermi Compute Architecture Whitepaper
CUDA Programming Guide 3.1, Appendix G

NVIDIA GeForce GTX 480 "SM"



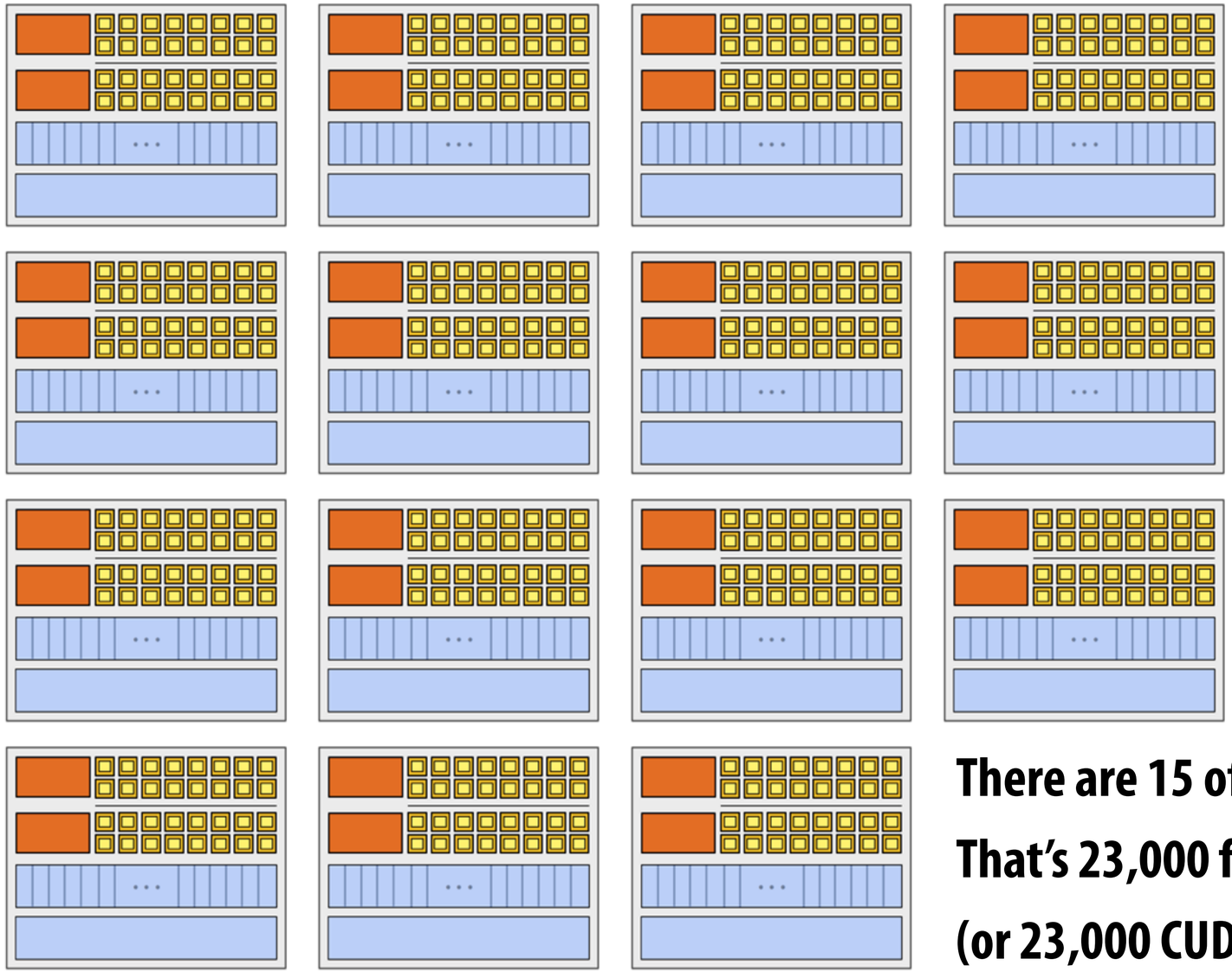


= CUDA core
(1 MUL-ADD per clock)

- The SM contains 32 CUDA cores
- Two warps are selected each clock (decode, fetch, and execute two warps in parallel)
- Up to 48 warps are interleaved, totaling 1536 CUDA threads

Source: Fermi Compute Architecture Whitepaper
CUDA Programming Guide 3.1, Appendix G

NVIDIA GeForce GTX 480



There are 15 of these things on the GTX 480:

That's 23,000 fragments!

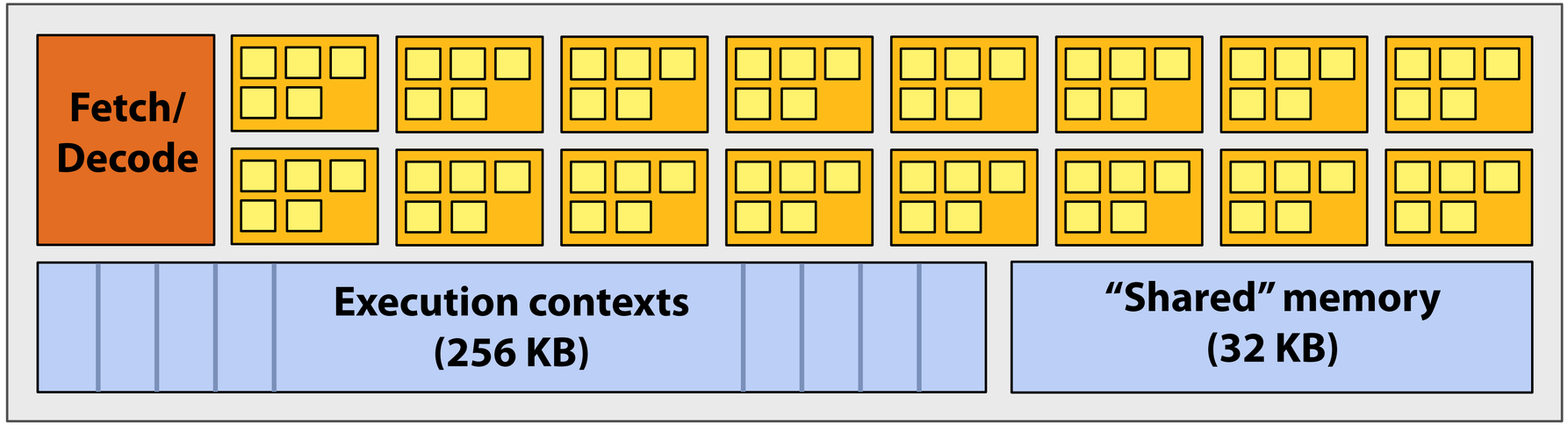
(or 23,000 CUDA threads!)

AMD Radeon HD 5870 (Cypress)

- AMD-speak:
  - 1600 stream processors

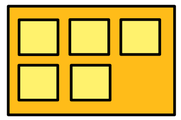
- Generic speak:
  - 20 cores
  - 16 "beefy" SIMD functional units per core
  - 5 multiply-adds per functional unit (VLIW processing)

AMD Radeon HD 5870 "core"



Groups of 64 [fragments/vertices/etc.] share instruction stream

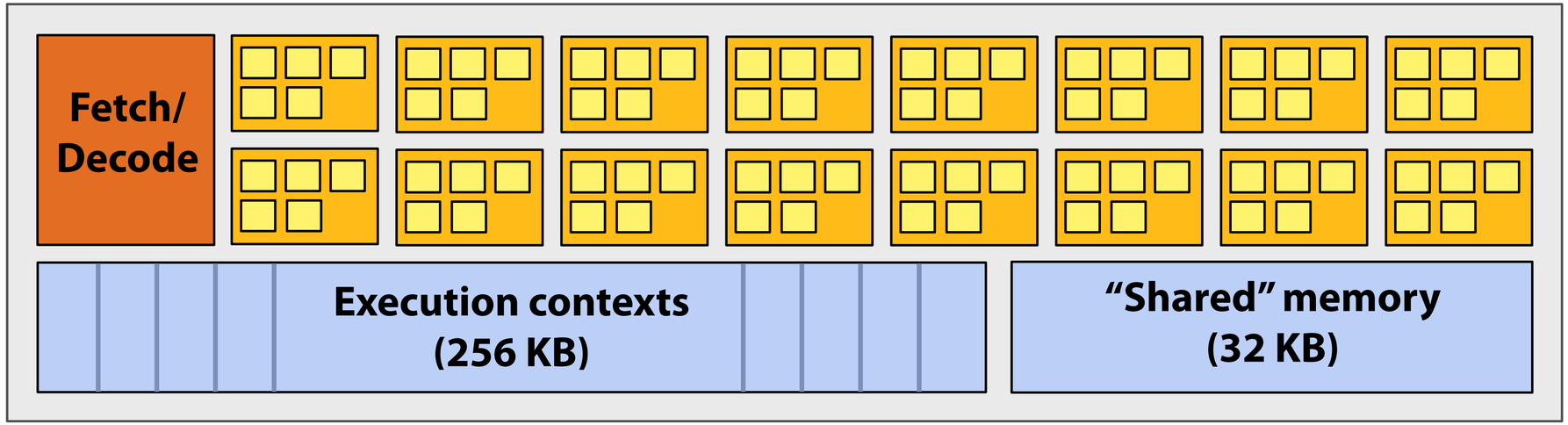
Four clocks to execute an instruction for all fragments in a group



= SIMD function unit,
control shared across 16 units
(Up to 5 MUL-ADDs per clock)

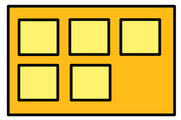
Source: ATI Radeon HD5000 Series: An Inside View (HPG 2010)

AMD Radeon HD 5870 "SIMD-engine"



Groups of 64 [fragments/vertices/OpenCL work items] are in a "wavefront".

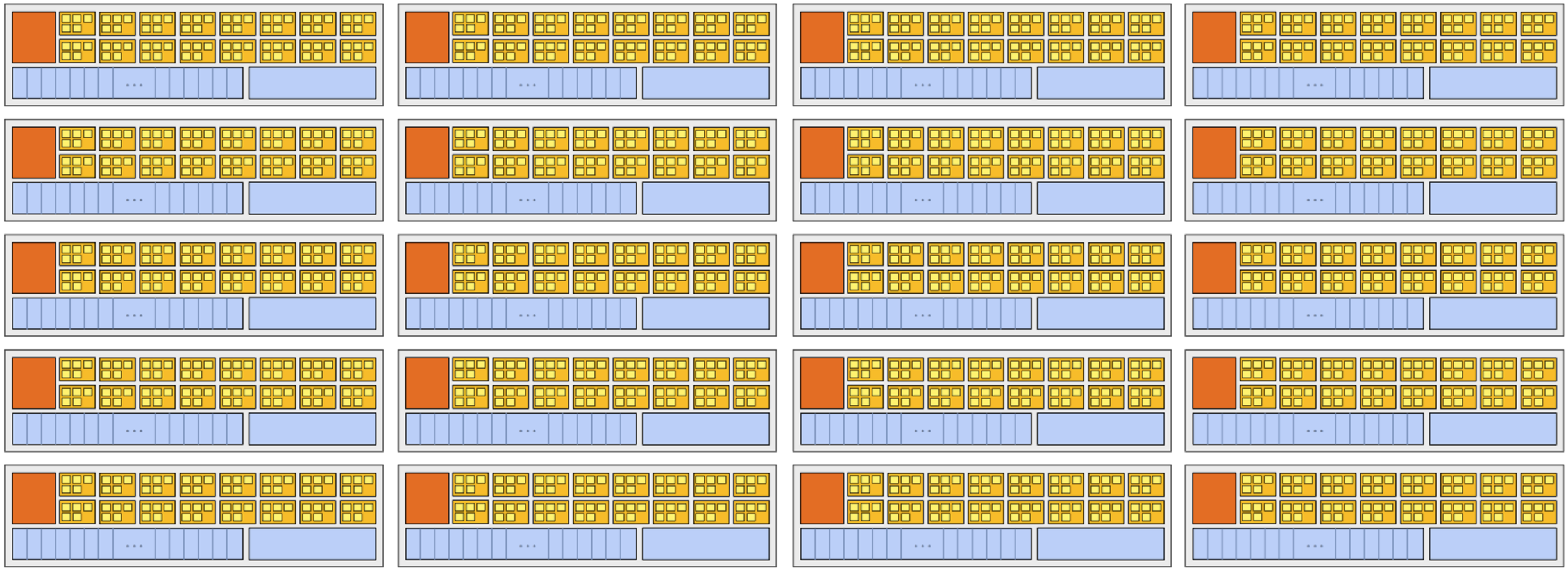
Four clocks to execute an instruction for an entire wavefront



= **stream processor**,
control shared across 16 units
(Up to 5 MUL-ADDs per clock)

Source: ATI Radeon HD5000 Series: An Inside View (HPG 2010)

AMD Radeon HD 5870



There are 20 of these "cores" on the 5870: that's about 31,000 fragments!