Lecture 3:

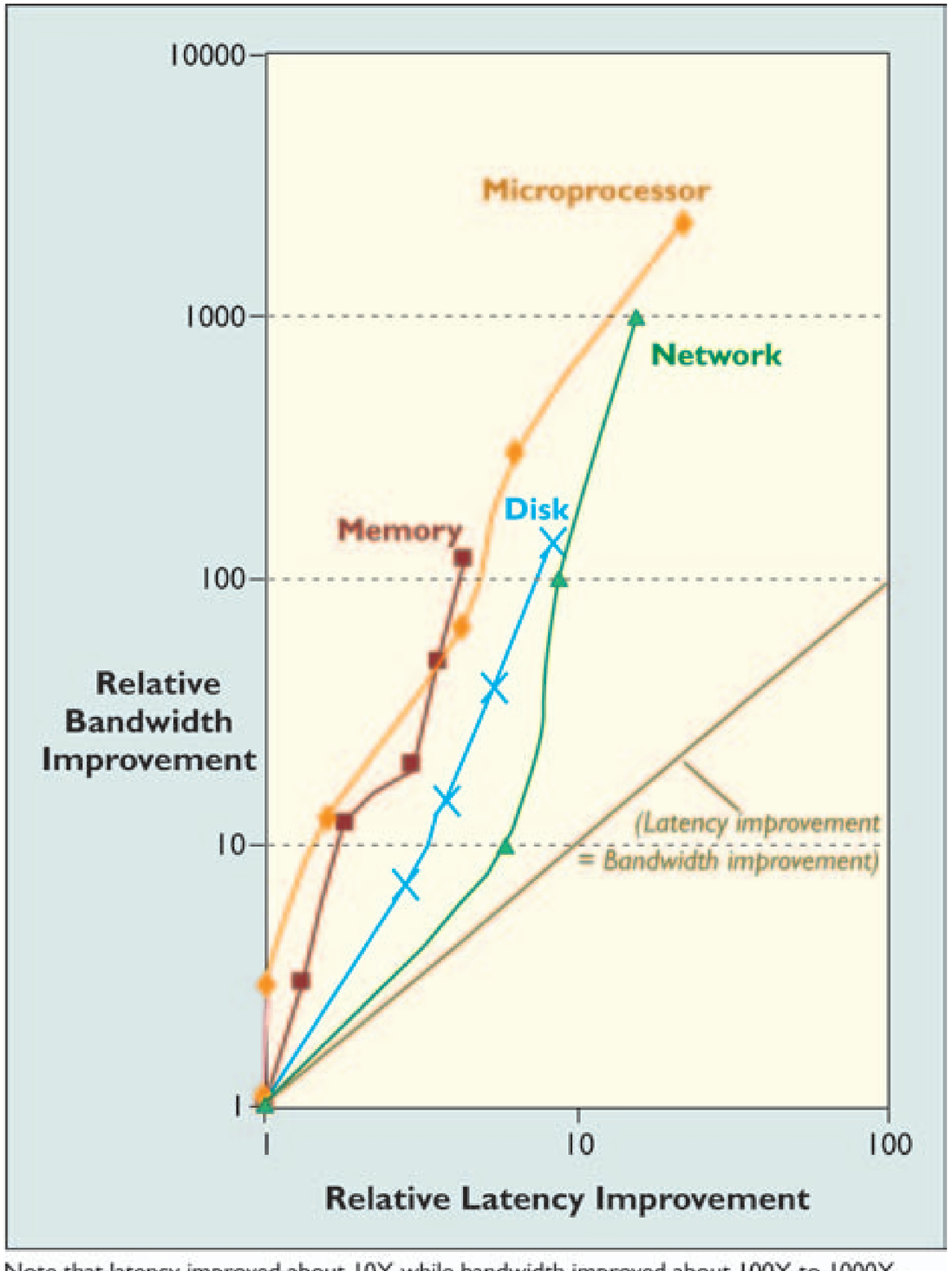
How Does a GPU Work?

EEC 277, Graphics Architecture
John Owens, UC Davis, Winter 2017

Announcements

- I'm still working on GLFW code for HW2, it's my big project for tomorrow

Latency Lags Bandwidth



Note that latency improved about 10X while bandwidth improved about 100X to 1000X.

D. Patterson, CACM
October 2004

From Bill Dally, 2015 talk …

Its not about the FLOPs

- DFMA 0.01mm^2 10pJ/OP – 2GFLOPs

16nm chip, 10mm on a side, 200W

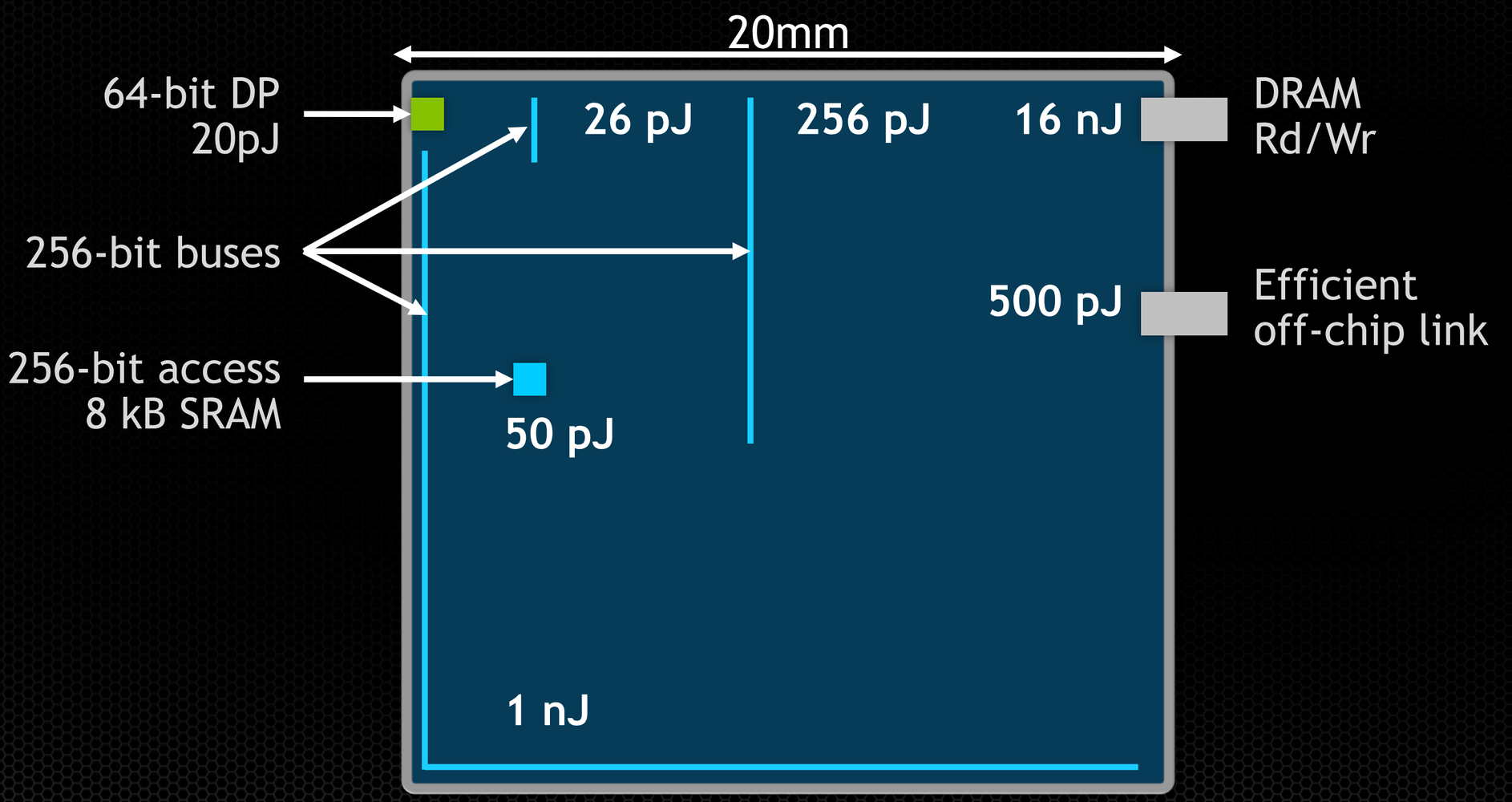
A chip with 10^4 FPUs:
100mm^2
200W
20TFLOPS

Pack 50,000 of these in racks
1EFLOPS
10MW

NVIDIA

From Bill Dally, 2015 talk …

Communication Dominates Arithmetic



NVIDIA

Challenges

- *Not* how to pack enough arithmetic on a chip but instead …
  - How to do so in an energy-efficient way
    - *"Perf/W is Perf"—Bill Dally*
  - How to design a programming model that allows you to use that arithmetic
- Communication, power, and latency
  - Dominant power cost
  - Must exploit locality (for power and latency)
  - Other side of chip is many cycles away
  - Off-chip is hundreds of cycles away

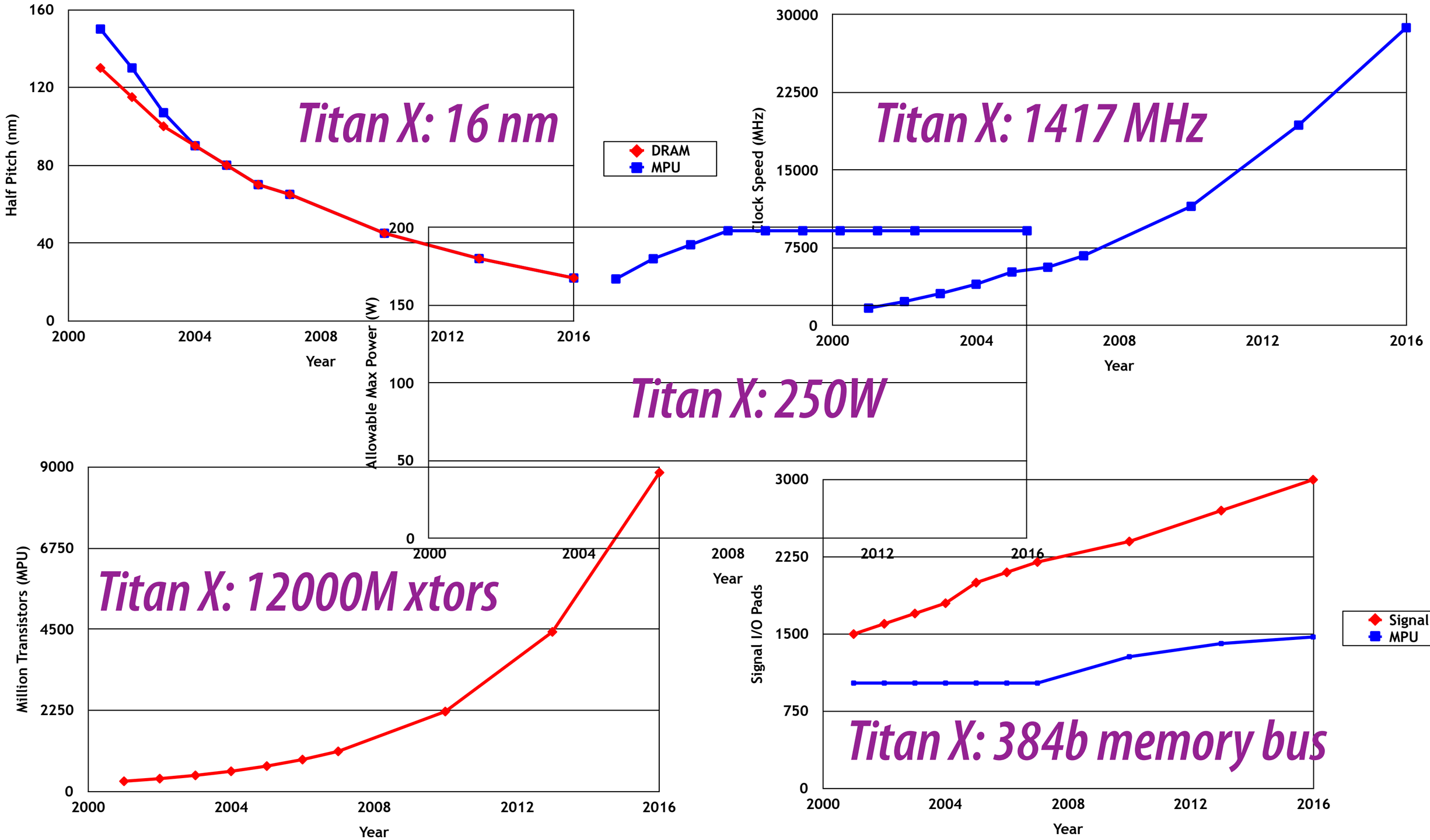
Major theme of this course: throughput-centered focus (as opposed to latency-centered)

Technology => dramatic change

- Processor
  - logic capacity: about 30% per year
  - clock rate: about 20% per year
  - capacity: about 70% per year
- Memory
  - DRAM capacity: about 60% per year (4x every 3 years)
  - DRAM bandwidth: about 25% per year
  - DRAM latency: about 5-10% per year
  - Cost per bit: improves about 25% per year
- Disk
  - capacity: about 60% per year
  - Total use of data: 100% per 9 months!
- Network bandwidth increasing more than 100% per year!

"Latency lags bandwidth" (bw improves by latency^2). D. Patterson, Oct 2004 CACM

International Technology Roadmap '01



10-Year Projection (2004–2014)

Transistors (NV40): 222M/2237M

Clock speed (NV40, MHz): 475/1890

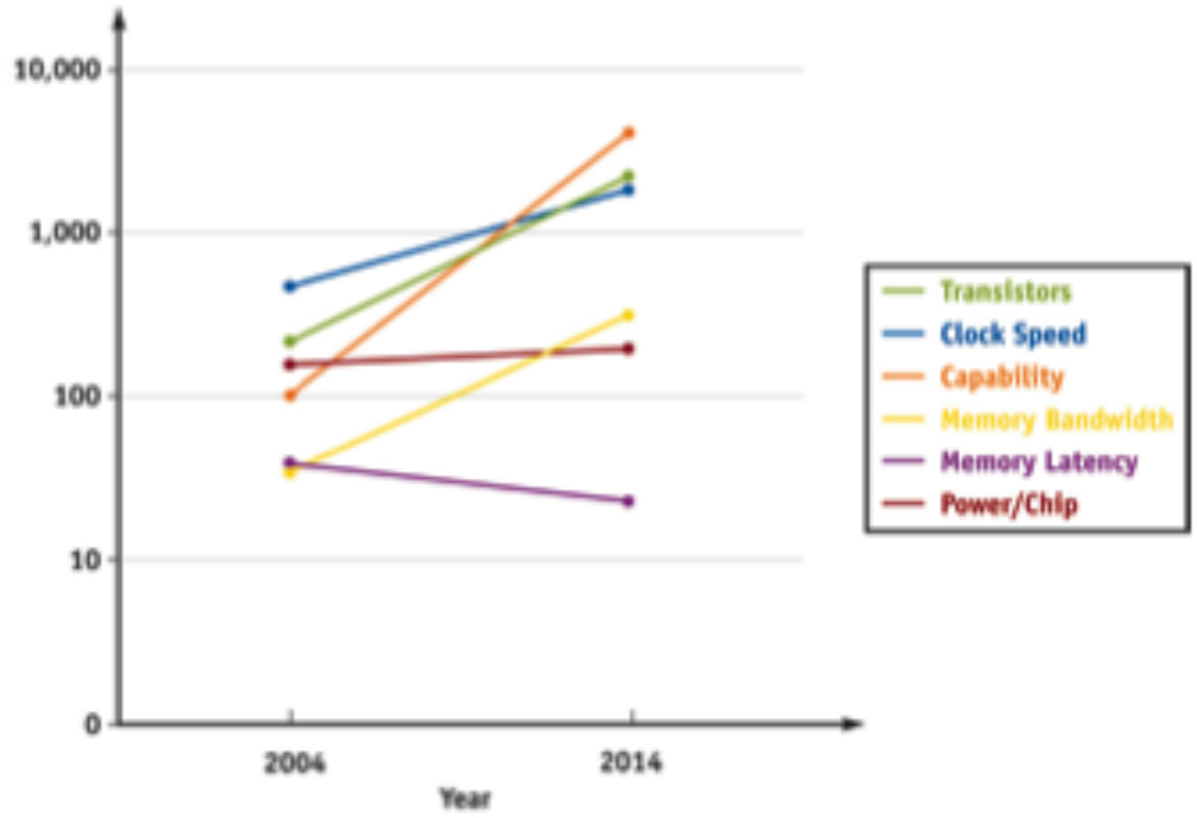
- **Capability: 105B/4228B**

Memory bandwidth (NV40, GB/s): 35/322

Memory latency (RAS, ns): 40/23

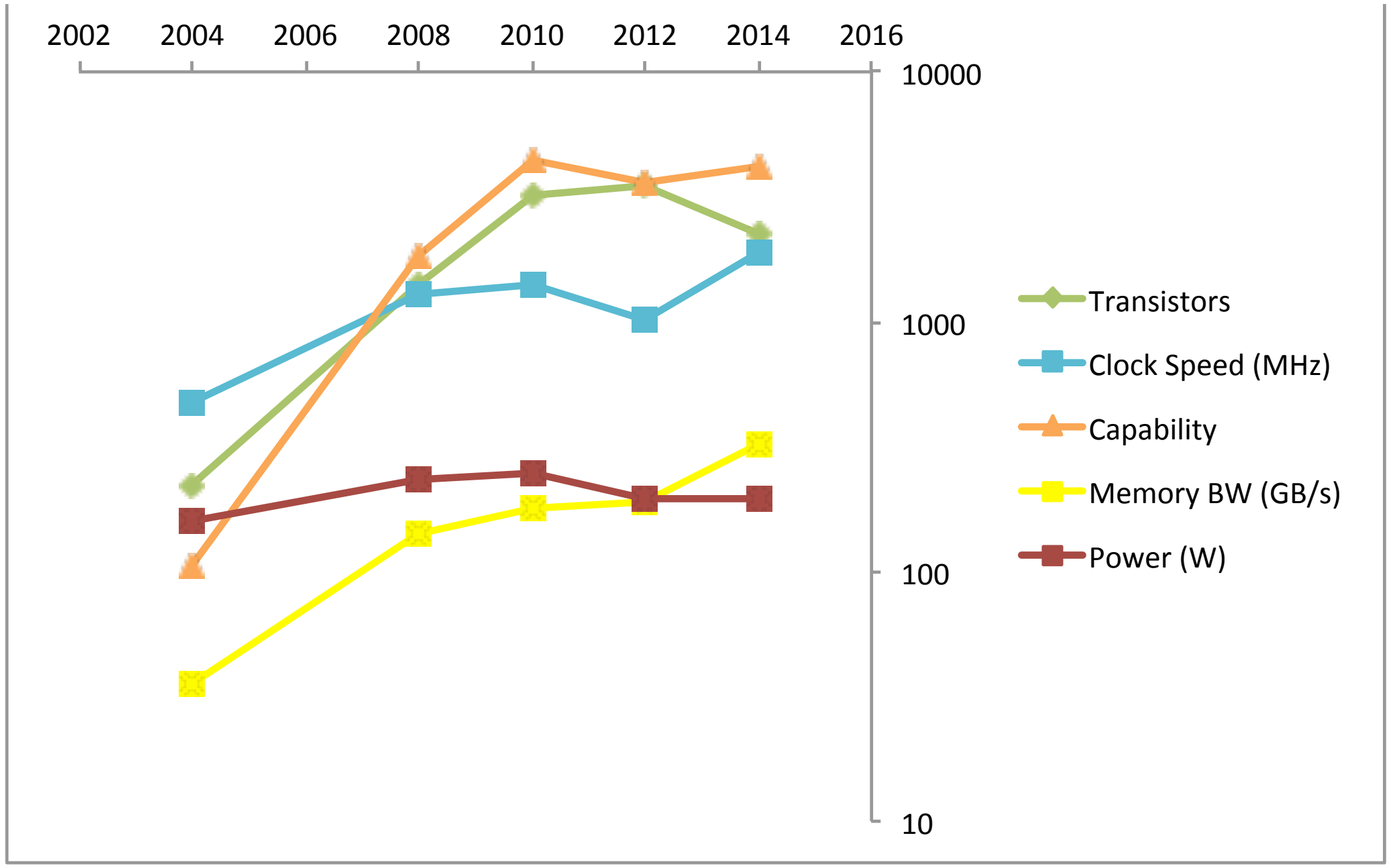
Power/chip (maximum, W): 158/198 (then flat)

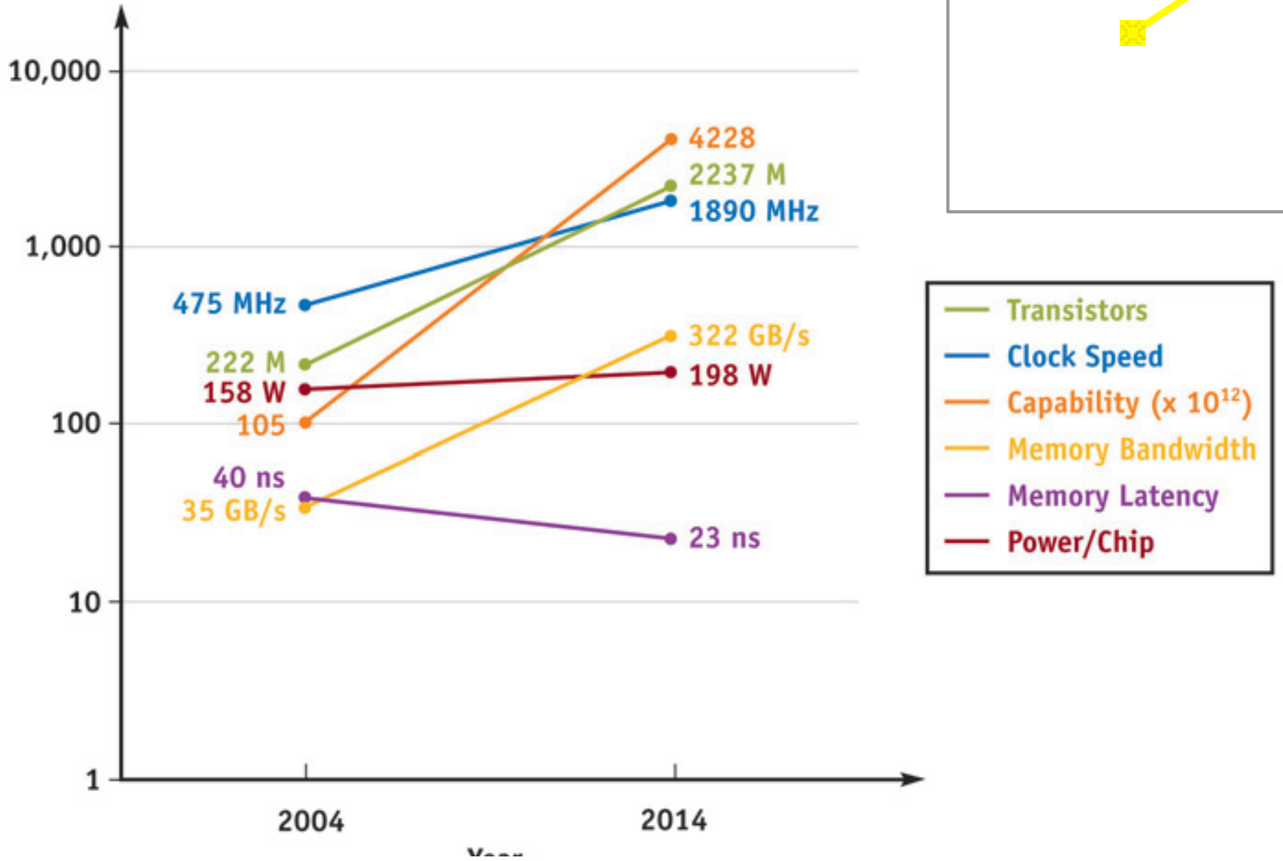
Take-home point: Capability > mem bw > mem latency



John Owens. Streaming Architectures and Technology Trends. In Matt Pharr, editor, GPU Gems 2, chapter 29, pages 457–470. Addison Wesley, March 2005.

Actual data from NVIDIA GPUs (GeForce 280, 480, 680)





10-Year Actual (2004–2014)

Transistors (NV40): 222M/2237M/ 7110M

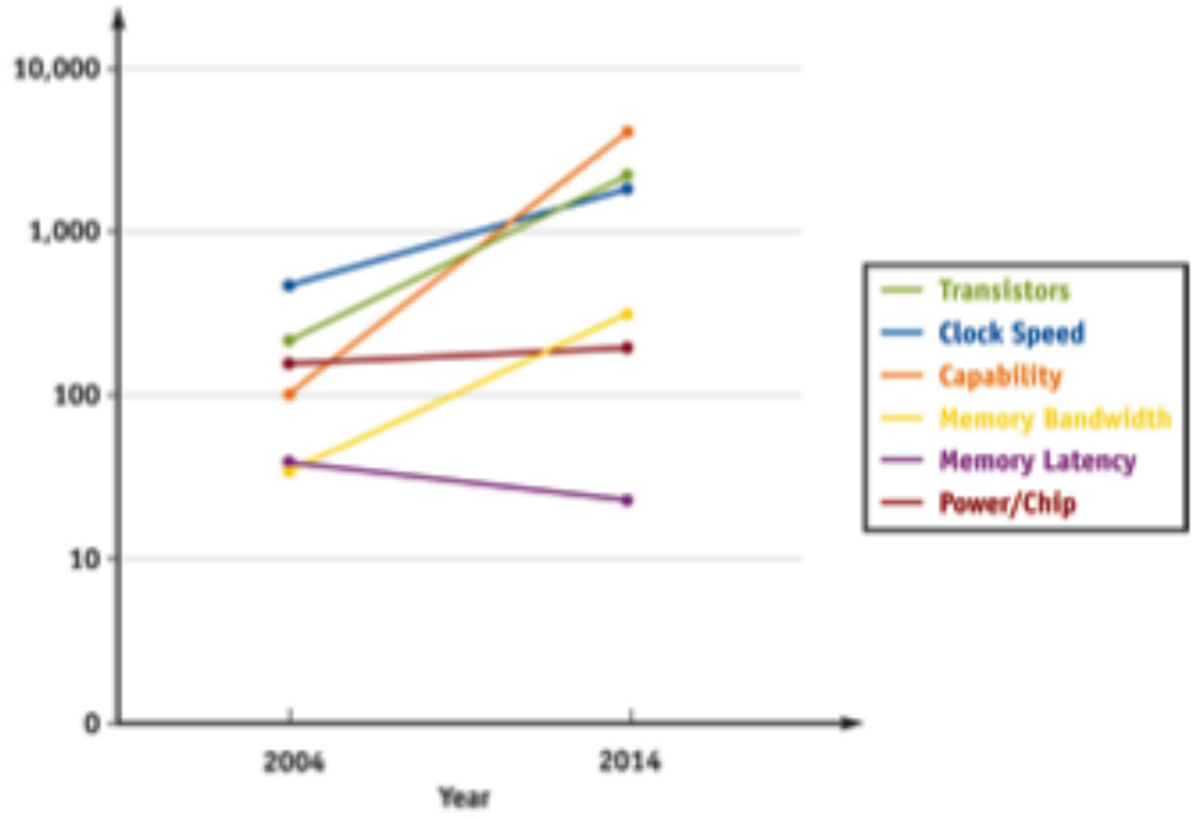
Clock speed (NV40, MHz): 475/1890/745

- **Capability: 105B/4228B/5296B**

Memory bandwidth (NV40, GB/s): 35/322/288

Memory latency (RAS, ns): 40/23/?

Power/chip (maximum, W): 158/198/235



Take-home point: Capability > mem bw > mem latency

ITRS 2.0 (2015)

- "As features approach the 10nm range and below it becomes clear that the semiconductor industry is running out of horizontal space."
- System integration goals: data centers, IoT, mobility
- "By 2020-25 device features will be reduces to a few nanometers and it will become practically impossible to reduce device dimensions any further." but "Increase in the number of transistors per unit area will eventually be accomplished by stacking multiple layers of transistors."
- "The number of cores in Graphics Processing Units (GPU) will increase by 50x in this time horizon (2015–2029). This increase in GPU processing capacity is necessary to keep up with the increasingly growing number of megapixels offered by the displays. The number of megapixels will in fact increase by 15x in the future. It is expected that the traffic between AP and memory will have to correspondingly increase more than 2x."

Trends / Responses

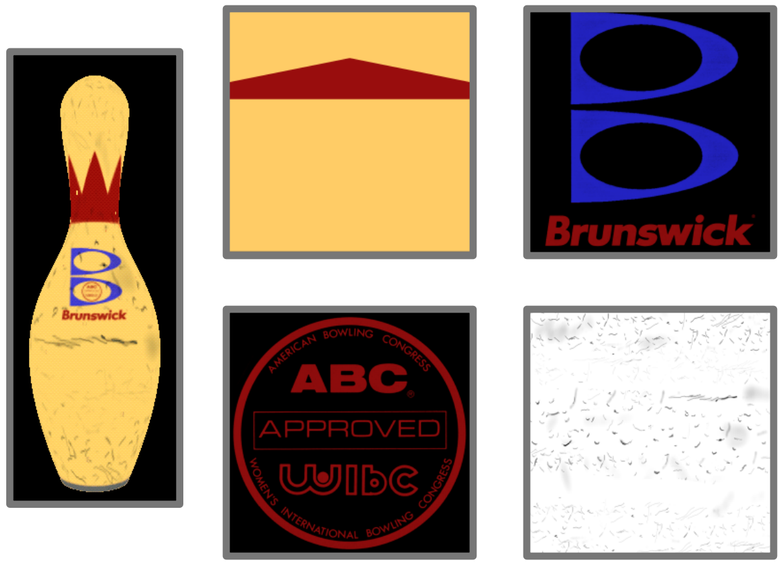
- More transistors / higher clock speeds
- Ratio between off-chip communication and computation changing
- On-chip communication speeds standing still

Trends / Responses

- More transistors / higher clock speeds
  - Faster processing
  - More features
    - New graphics features
    - Features to aid optimization of computation
    - Features easing burden on programmer
    - Higher quality
    - Programmability
- Ratio between off-chip communication and computation changing
- On-chip communication getting more costly

Trends / Responses

- More transistors / higher clock speeds
- Ratio between off-chip communication and computation changing
  - Reduce bandwidth from off-chip
    - Vertex (input) bandwidth
    - Texture bandwidth
    - Bandwidth to framebuffer
  - Compute rather than communicate
  - Patterson: caching, replication, prediction
- On-chip communication getting more costly



Trends / Responses

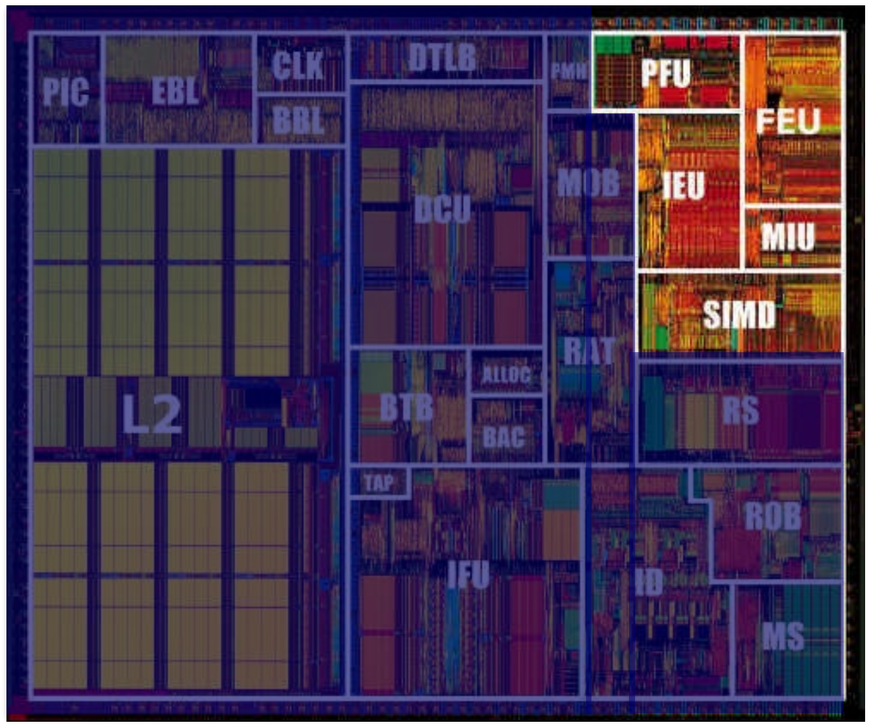
- More transistors / higher clock speeds
- Ratio between off-chip communication and computation changing
- On-chip communication getting more costly
  - Locality is king
  - Reduce global on-chip communication
  - Replicate computation rather than communicate
  - Power considerations are paramount
    - Figure of merit is performance per watt, not performance

Characteristics of Graphics

- Define problem
  - Large computational requirements
  - Massive parallelism
  - Long latencies tolerable
  - Deep, feed-forward pipelines
  - Hacks are OK—can tolerate lack of accuracy

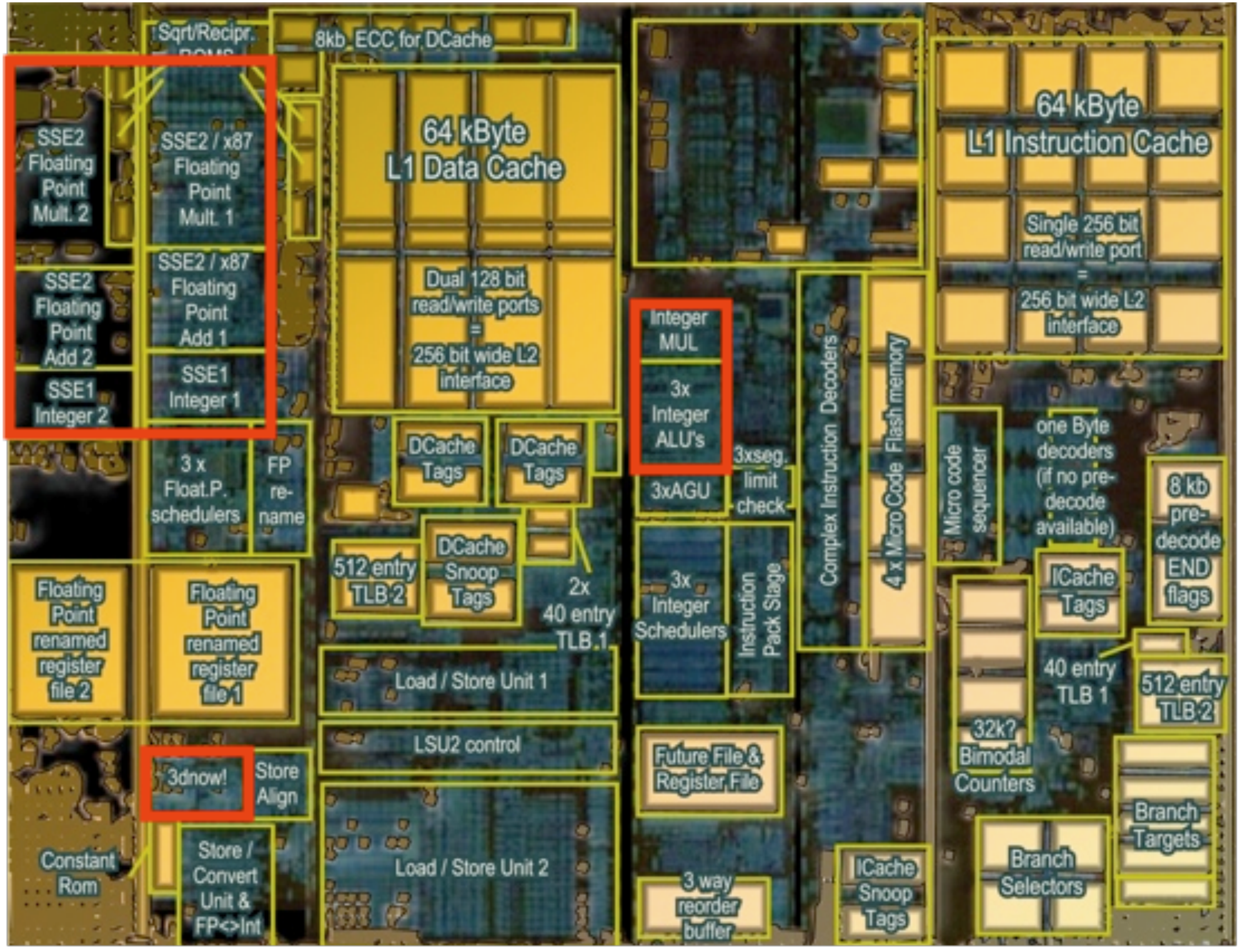
Why Not Microprocessors?

- Microprocessors address a different application space
  - Scalar programming model with no native data parallelism
  - Few arithmetic units—little area
  - Optimized for complex control
  - Optimized for low latency not high bandwidth
  - Processor scaling is slowing—difficult to maintain gates/clock, IPC increases

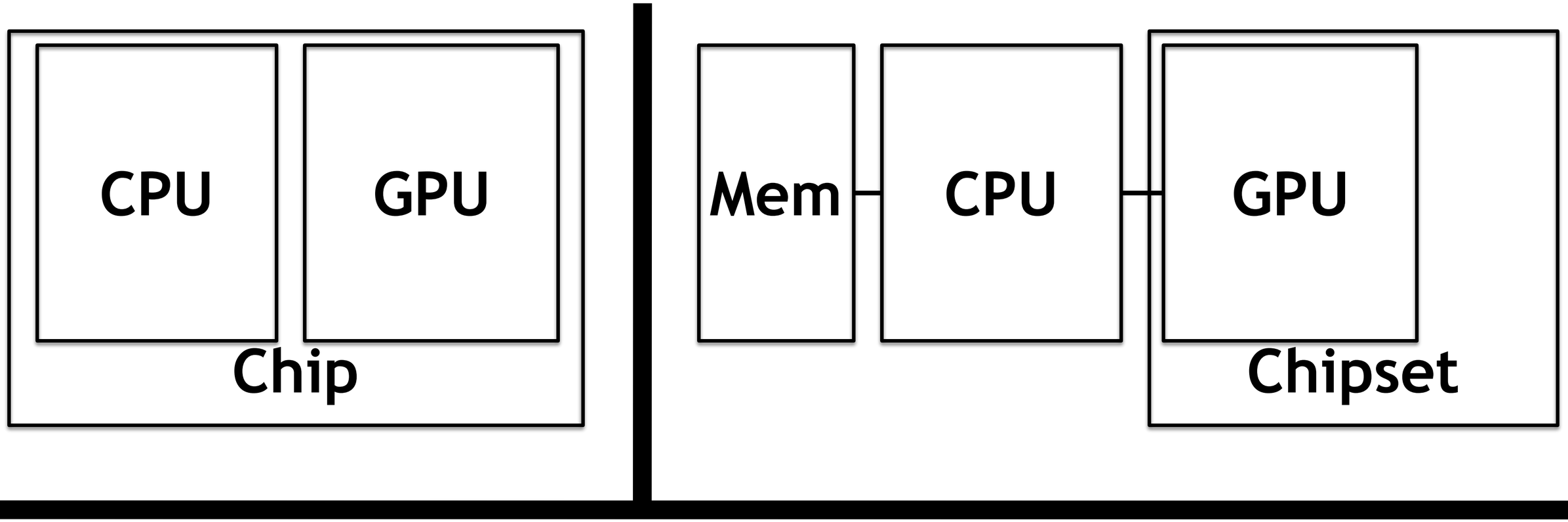


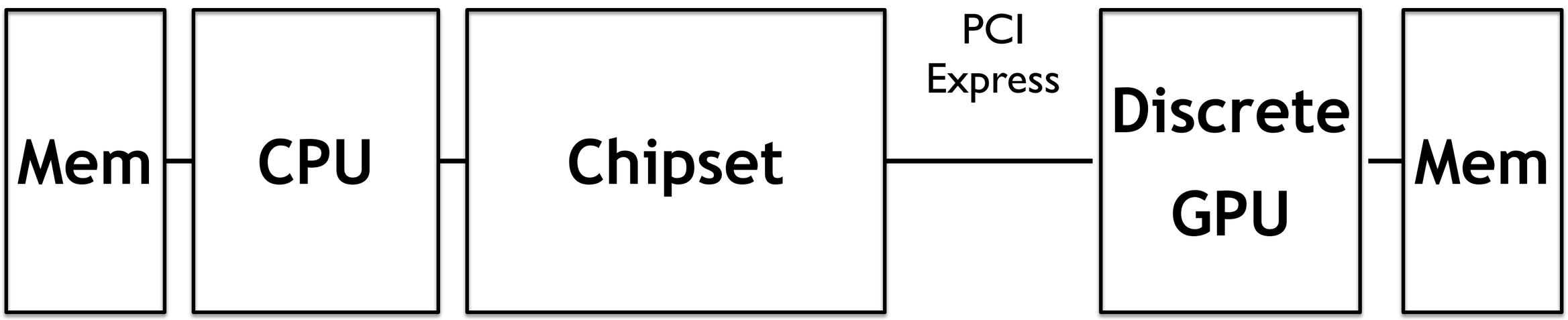
Pentium III – 28.1M T

AMD K7 “Deerhound”



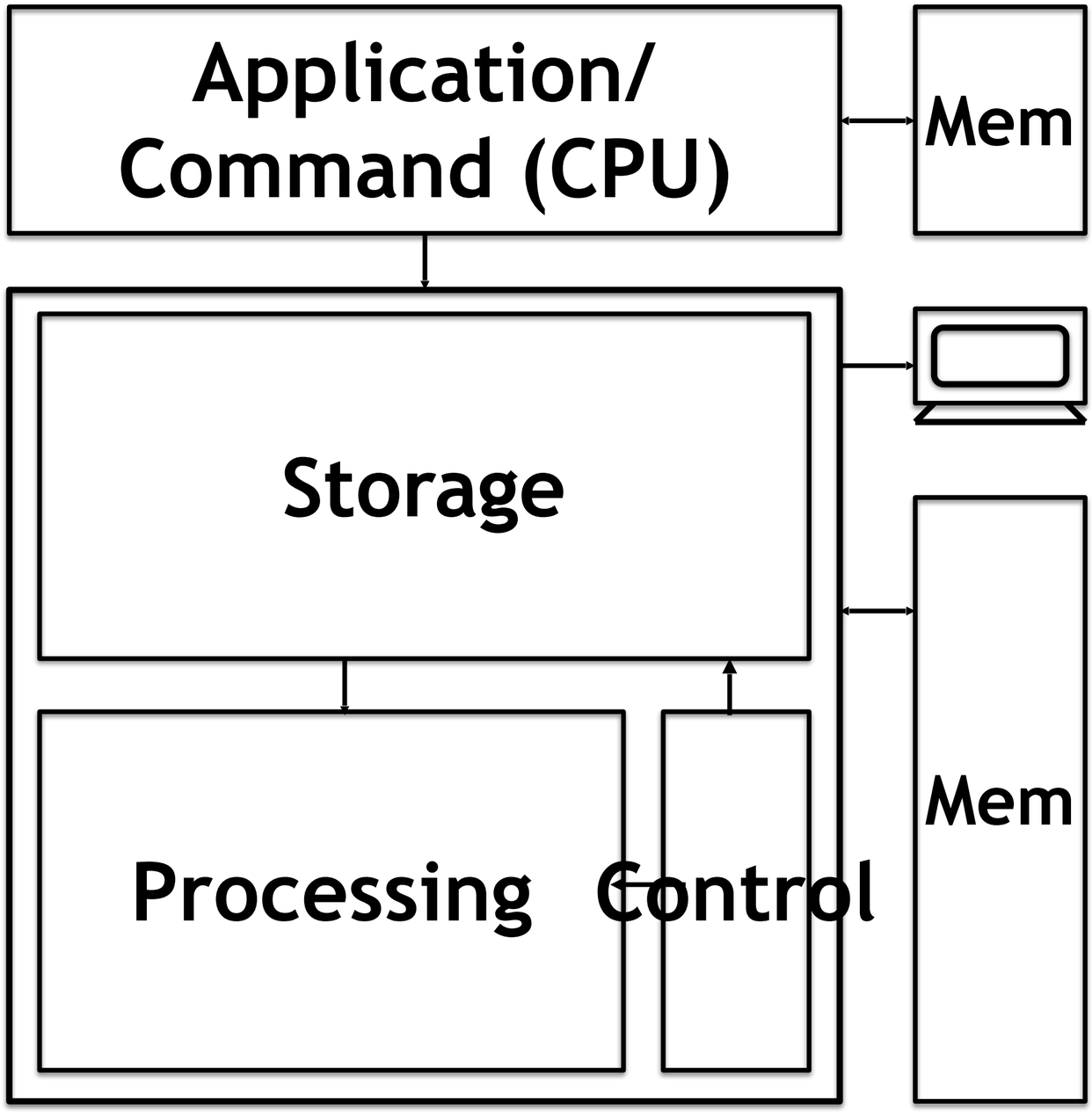
GPU in system (3 alternatives)



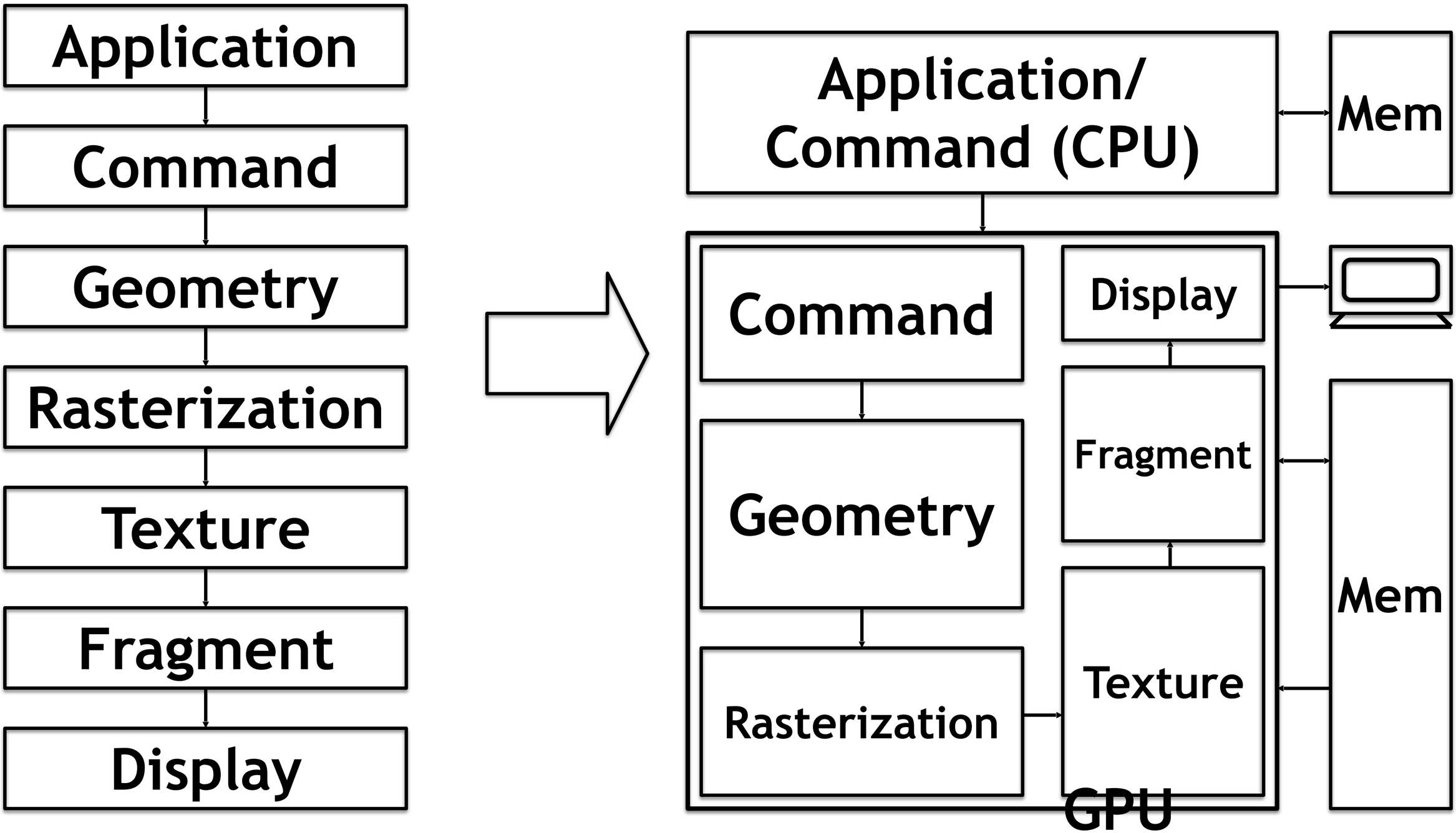


Time Multiplexed

- Processing divided in *time* not *space*
- Desirable storage:
  - Is cache useful?
  - Producer-consumer
- Kernel processing: must handle large arithmetic demand



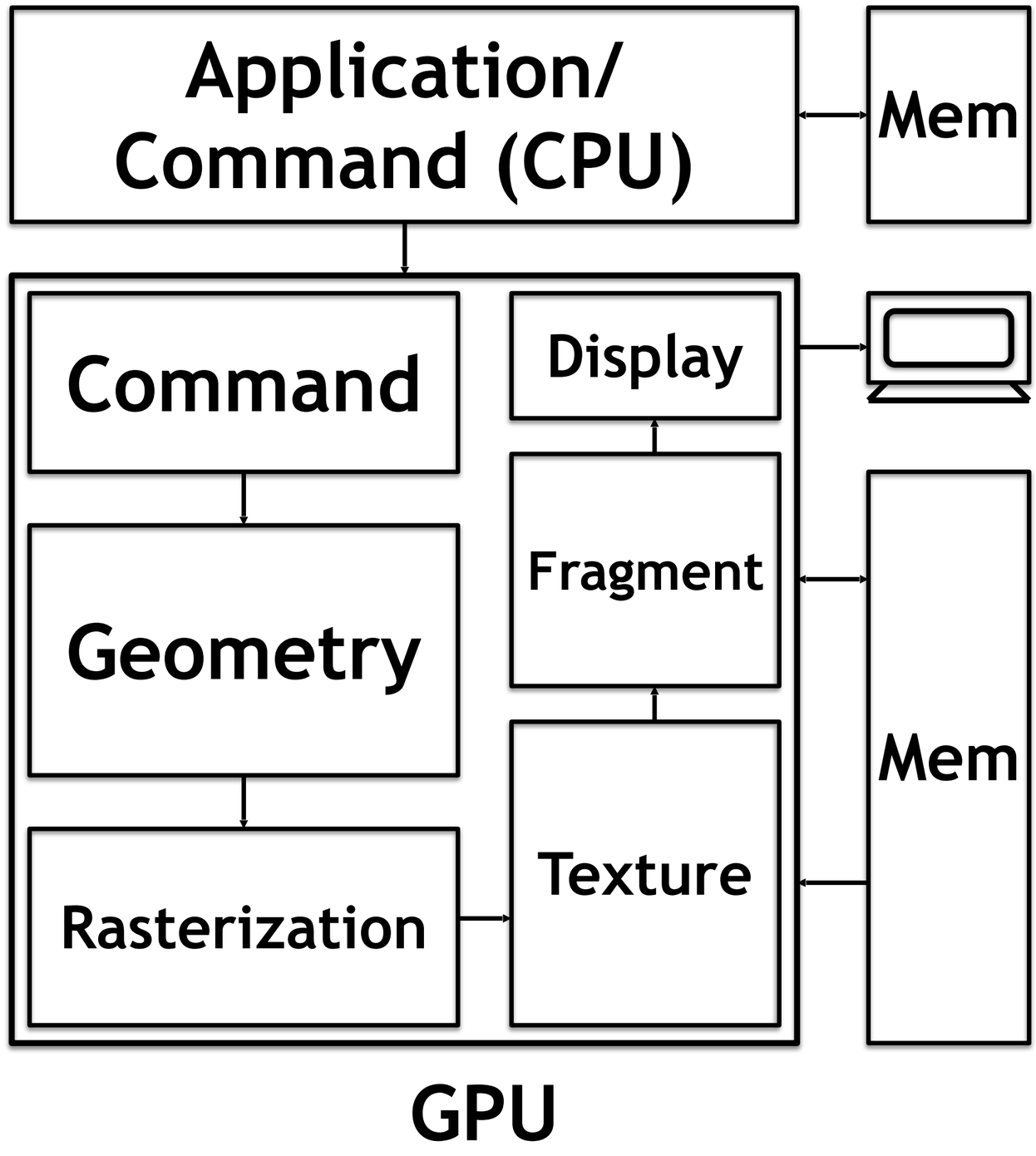
Graphics Hardware—Task Parallel



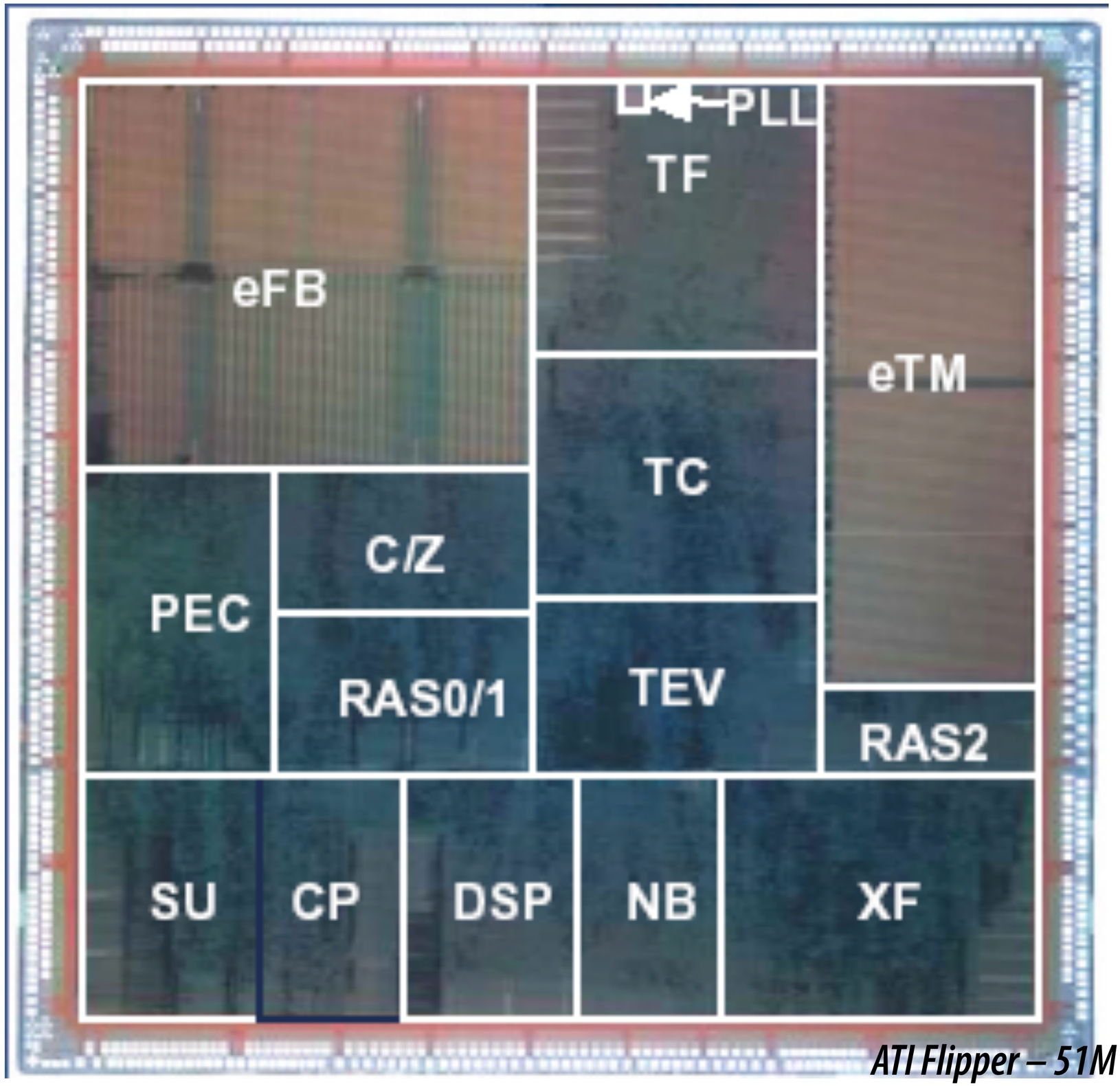
Why is Graphics Hardware Fast?

- Large computational requirements
- Massive parallelism
- Long latencies tolerable
- Deep, feed-forward pipelines

- Problems?



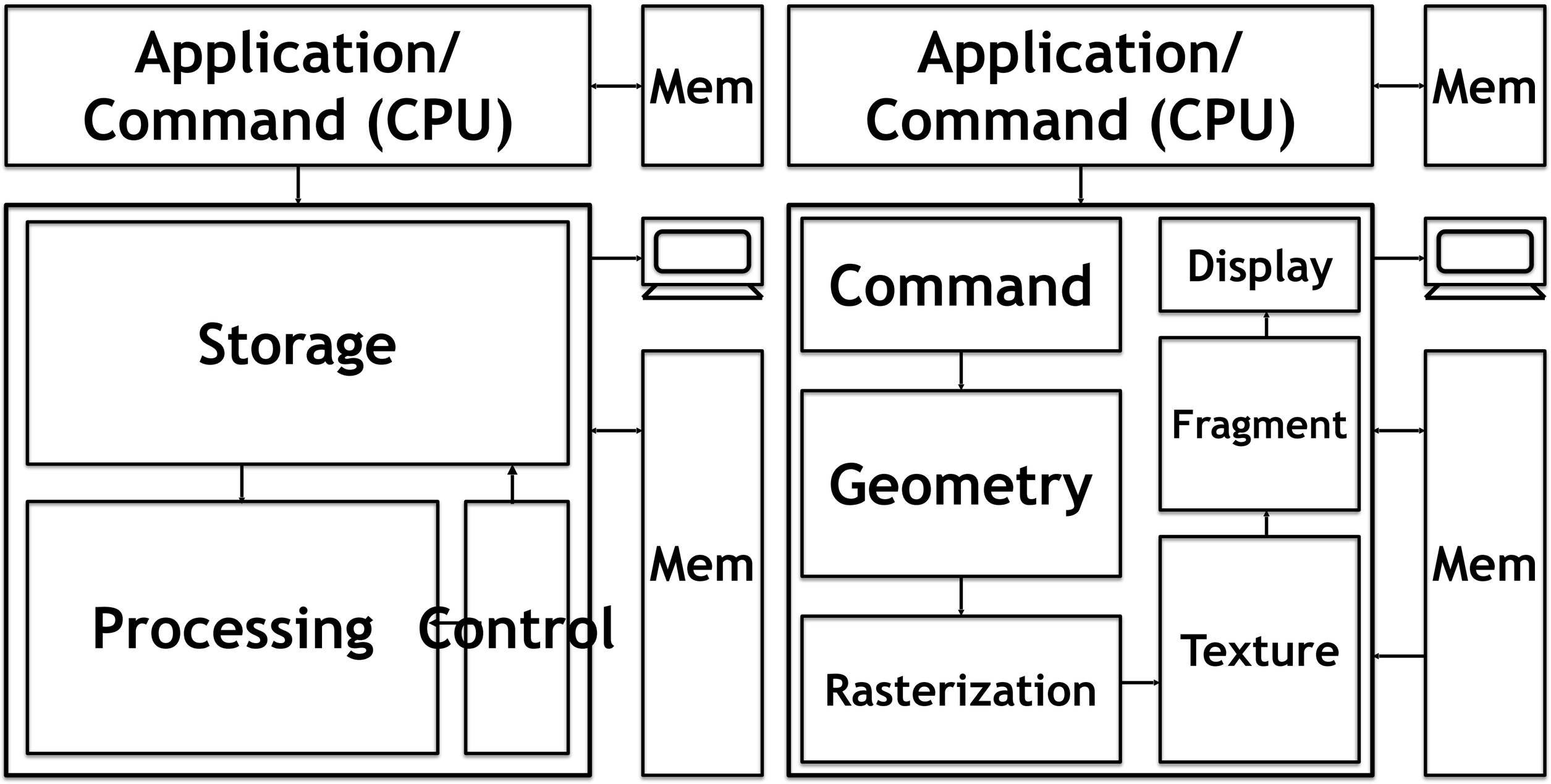
Why is Graphics Hardware Fast?



ATI Flipper – 51M T

- Task-parallel organization
- Each module hardwired
- Provides ample computation resources
- Exploits producer-consumer locality
- Efficient communication patterns

Comparison


Application/
Command (CPU)
Mem
Storage
Processing
Control
Mem
Application/
Command (CPU)
Mem
Command
Display
Geometry
Fragment
Rasterization
Texture
Mem

Task-parallel vs. Time-multiplexed

- (Usually hardwired)
- Concurrent
  - multiple stages in parallel
- Specialized
  - each stage: special hw
- Overspecialization
  - much of hardware unused for any specific scene
- Poor load balance btwn stages

- (Programmability inherent)
- One stage at a time
  - no task-level parallelism
- Specialization less useful
  - unused for most stages
- Efficient computation
  - spends no effort (time or silicon) on unused capabilities
- Good load balance btwn stages, but possible imbalance within stage

. . . what if every stage was programmable?

Slides from Fatahalian and Houston

- K. Fatahalian and M. Houston, "A closer look at GPUs," *Communications of the ACM*, vol. 51, no. 10, pp. 50–57, Oct. 2008.

This talk

- Three major ideas that make GPU processing cores run fast

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

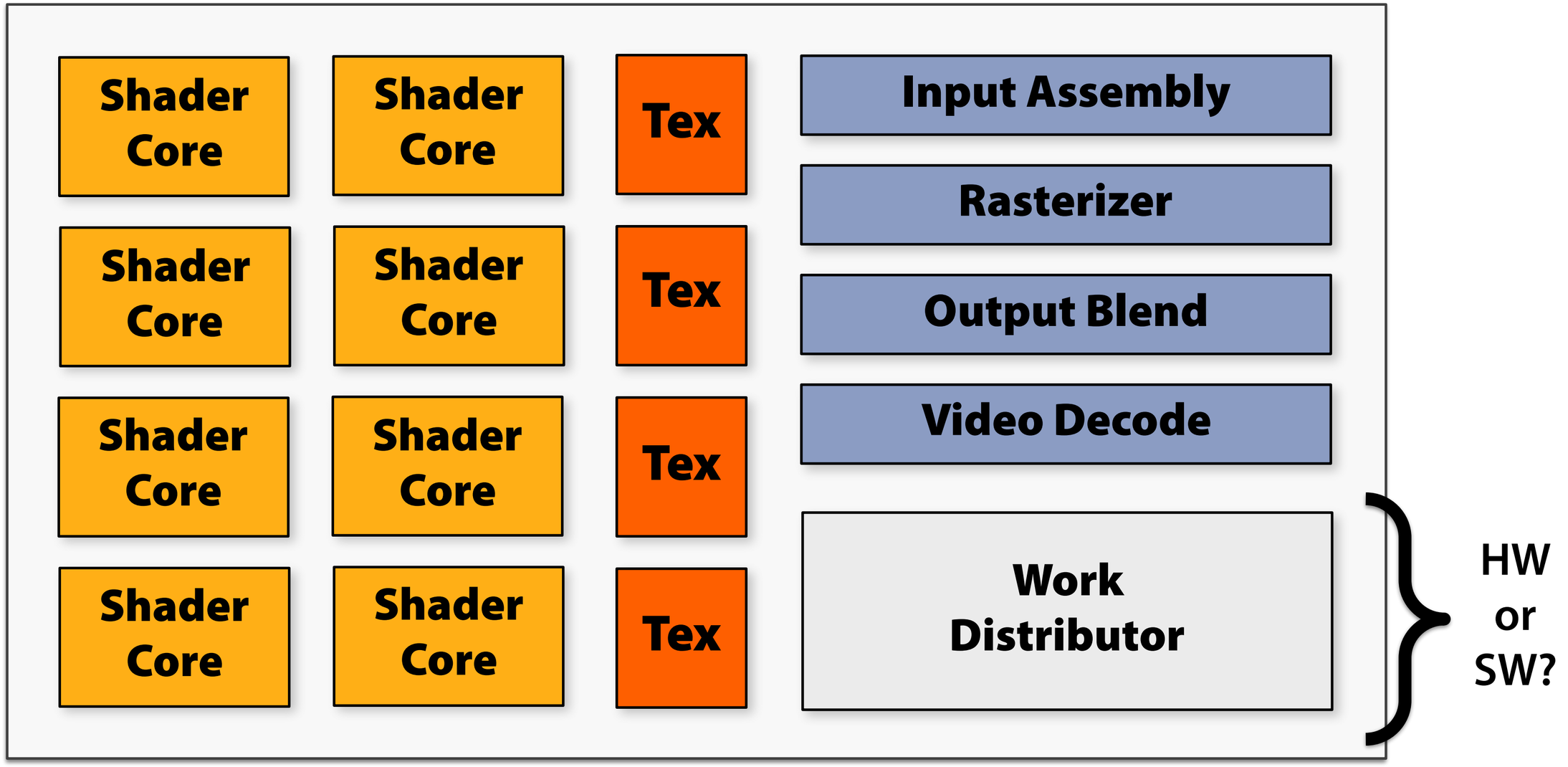
- Memory hierarchy: moving data to processors

Part 1: throughput processing

- Three key concepts behind how modern GPU processing cores run code

- Knowing these concepts will help you:
  - Understand space of GPU core (and throughput CPU processing core) designs
  - Optimize shaders/compute kernels
  - 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

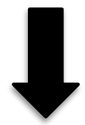
```
sampler mySamp;
Texture2D<float3> myTex;
float3 lightDir;

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

```
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);
}
```

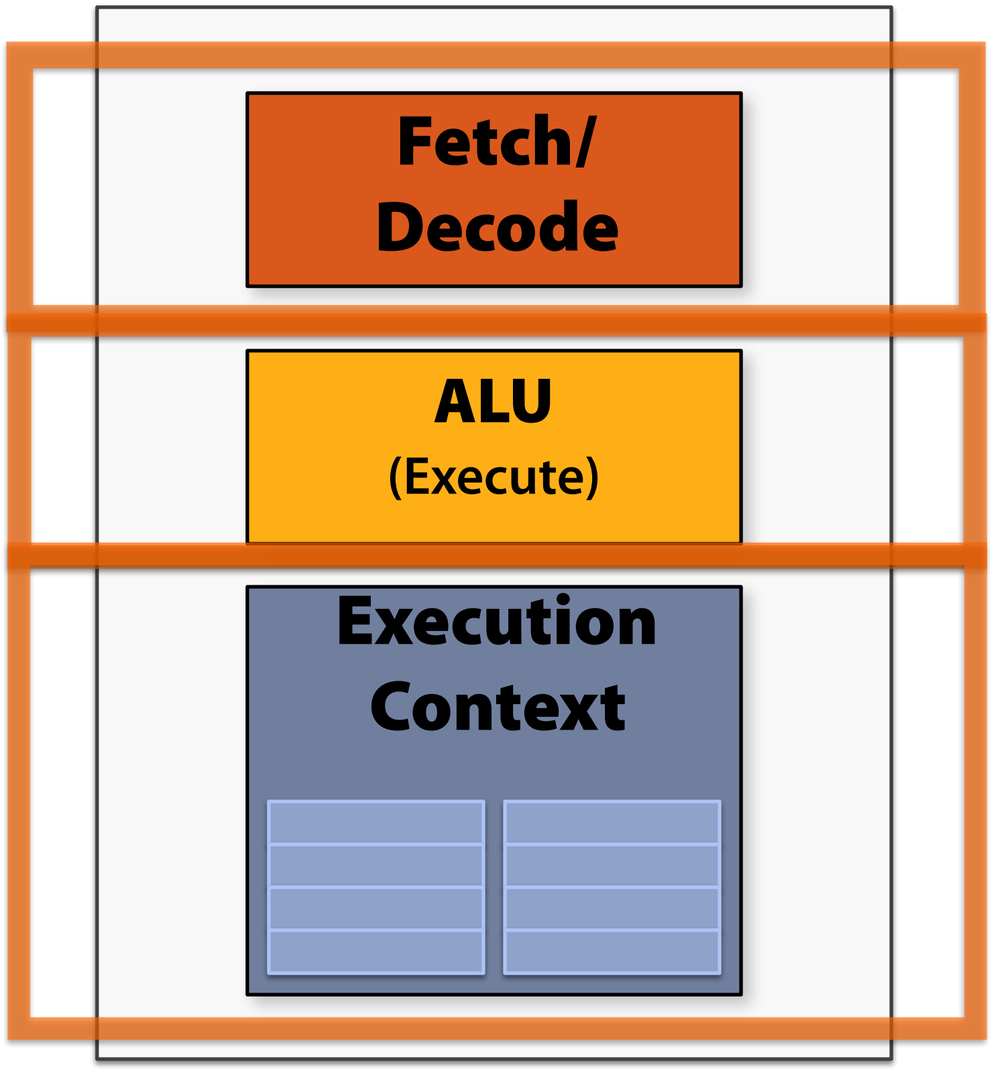

1 unshaded fragment input record

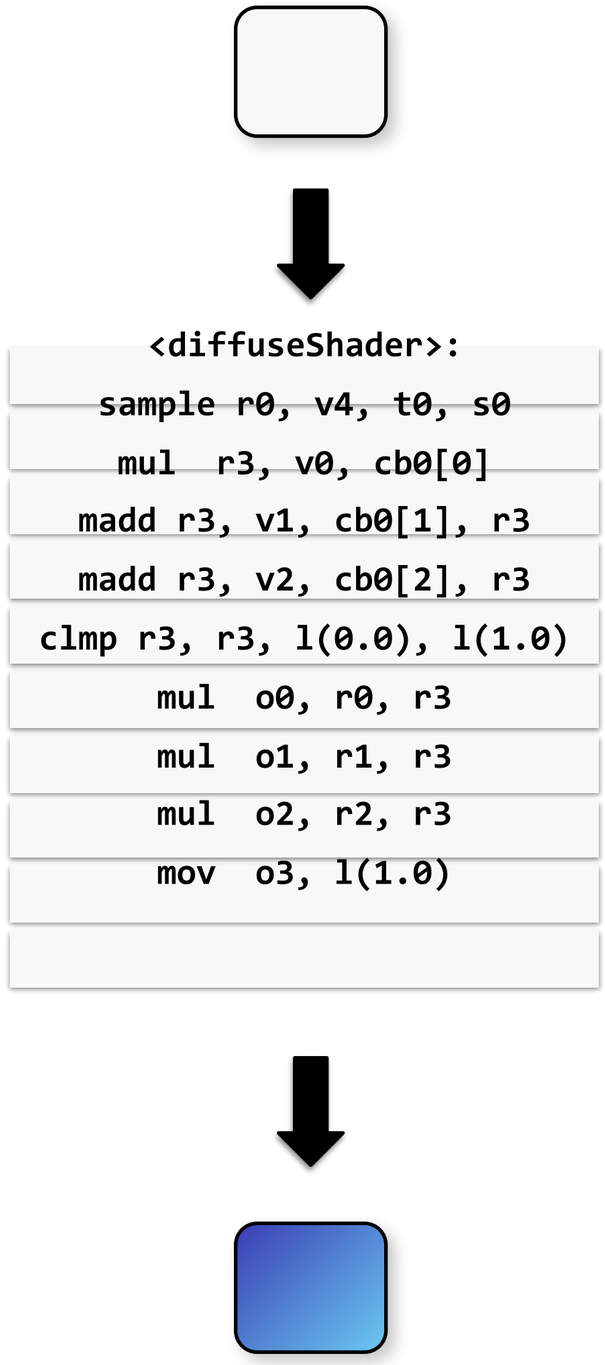


```
<diffuseShader>:
sample r0, v4, t0, s0
mul  r3, v0, cb0[0]
madd r3, v1, cb0[1], r3
madd r3, v2, cb0[2], r3
clmp r3, r3, l(0.0), l(1.0)
mul  o0, r0, r3
mul  o1, r1, r3
mul  o2, r2, r3
mov  o3, l(1.0)
```

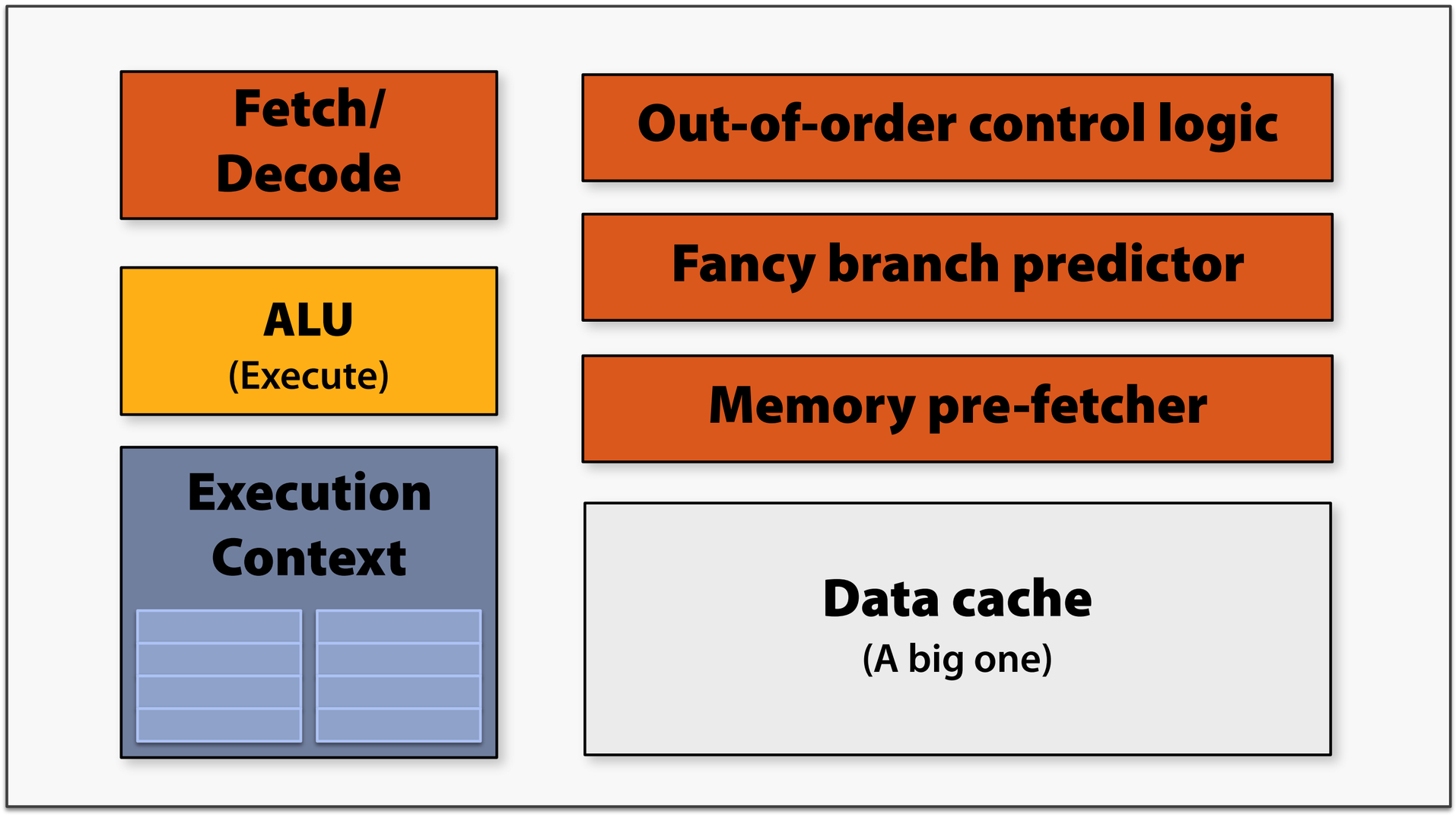
1 shaded fragment output record

Execute shader

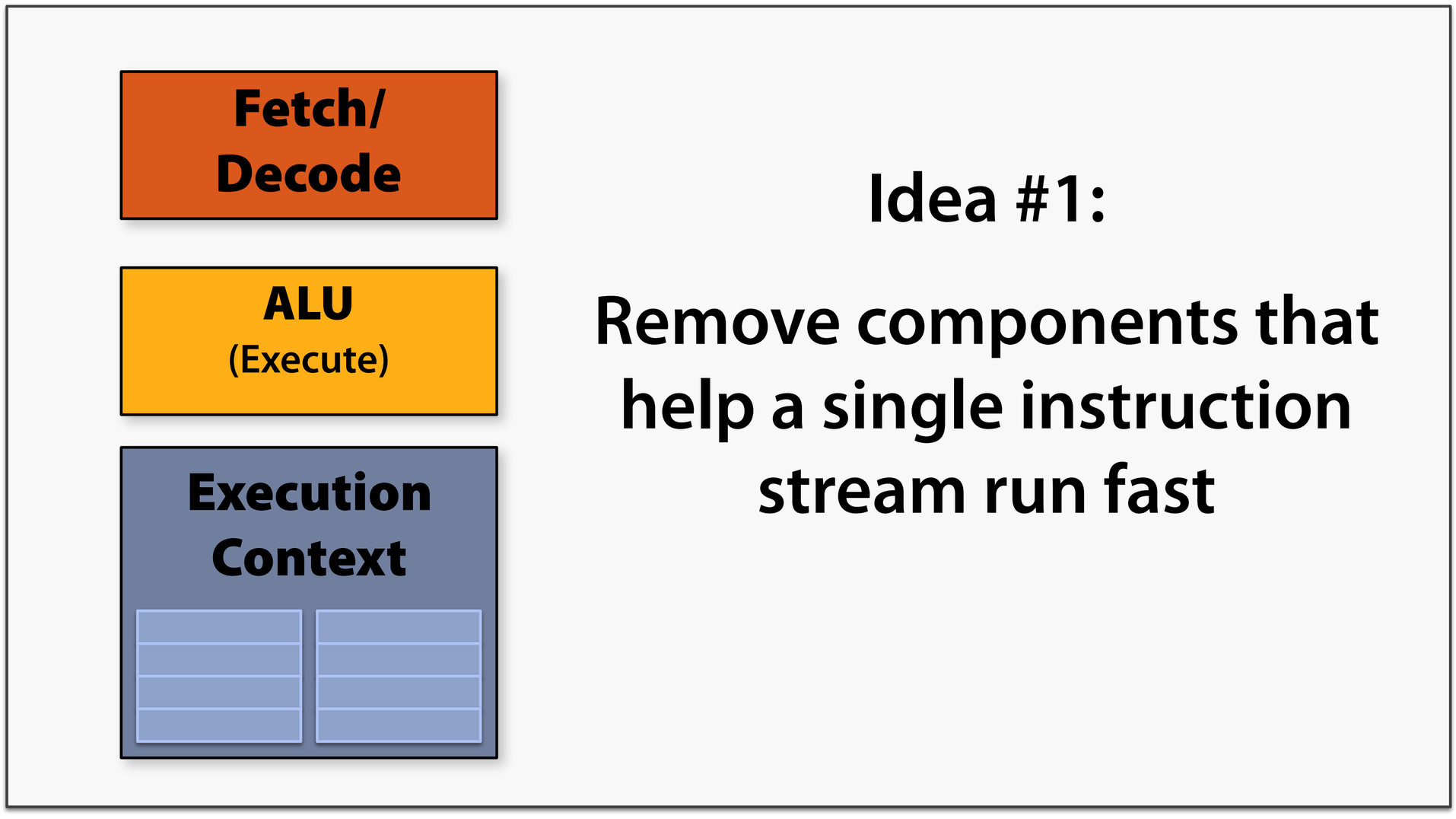




CPU-"style" cores

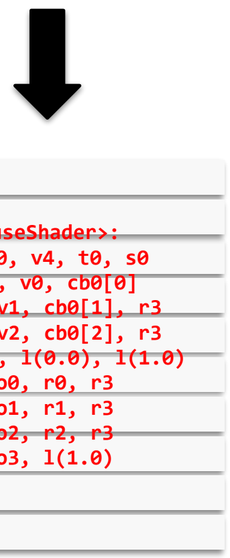


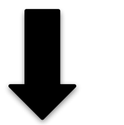
Slimming down

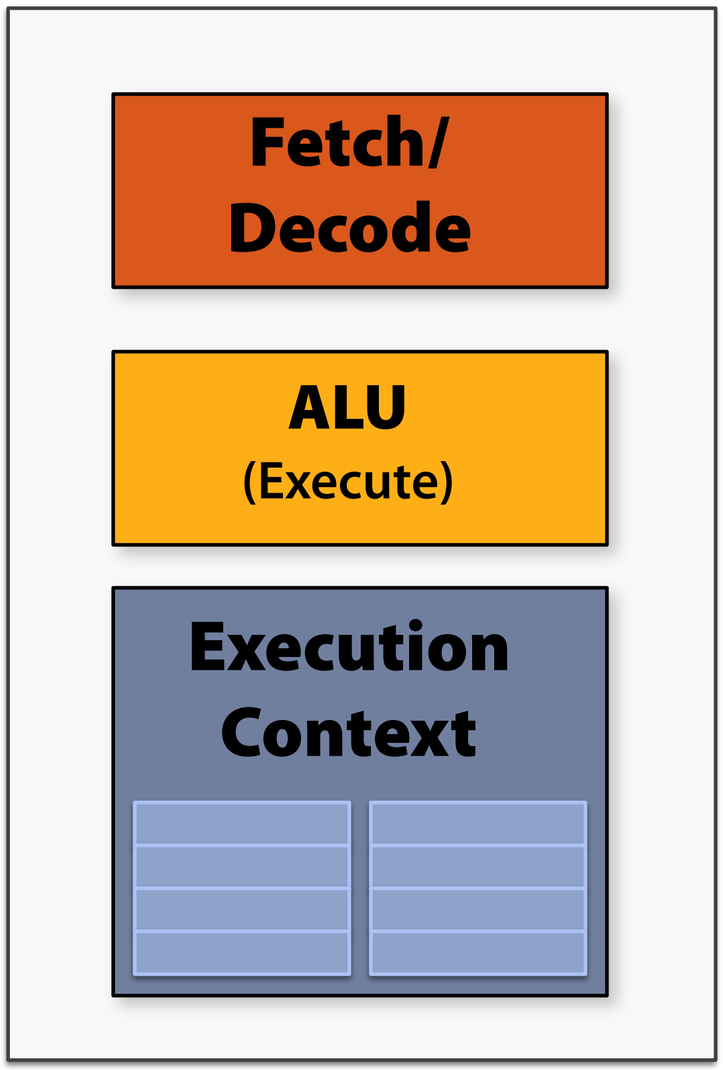


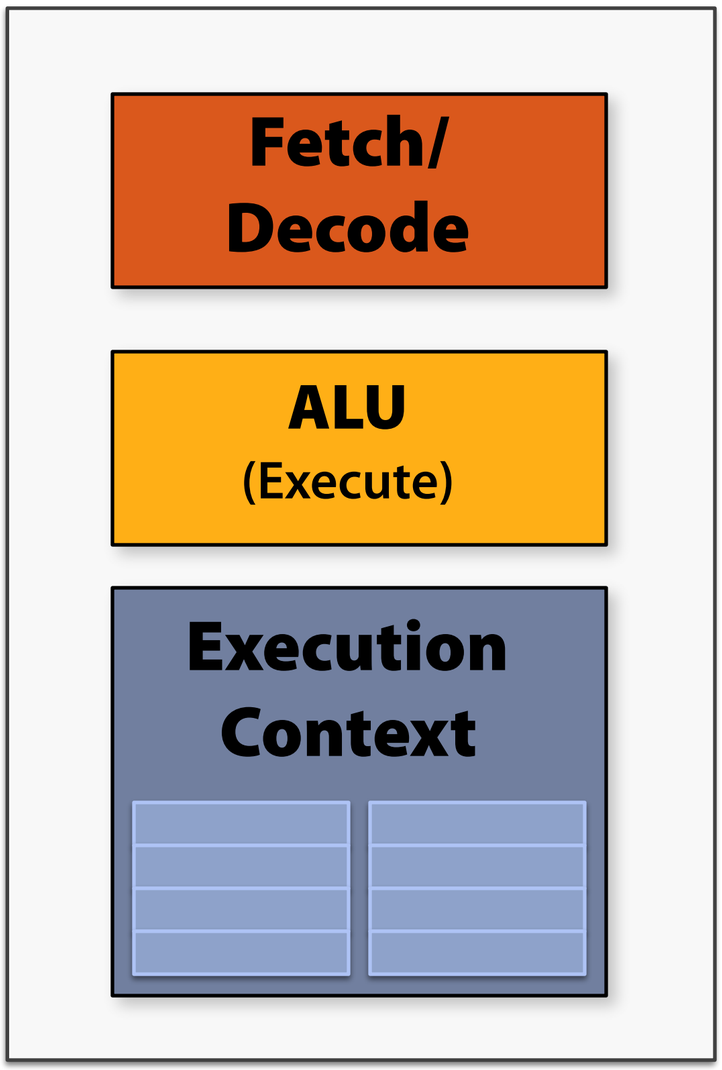
Two cores (two fragments in parallel)

fragment 1

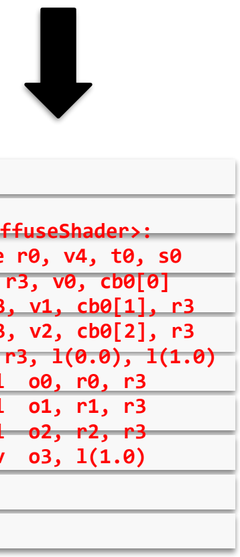


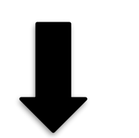




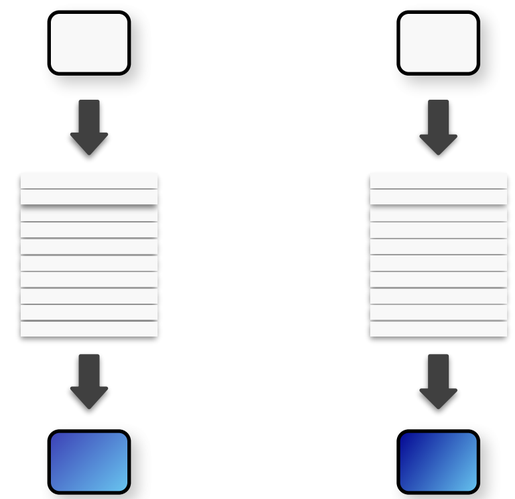


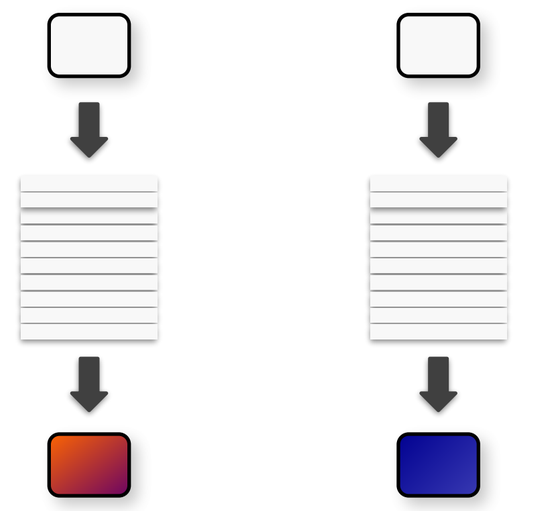
fragment 2

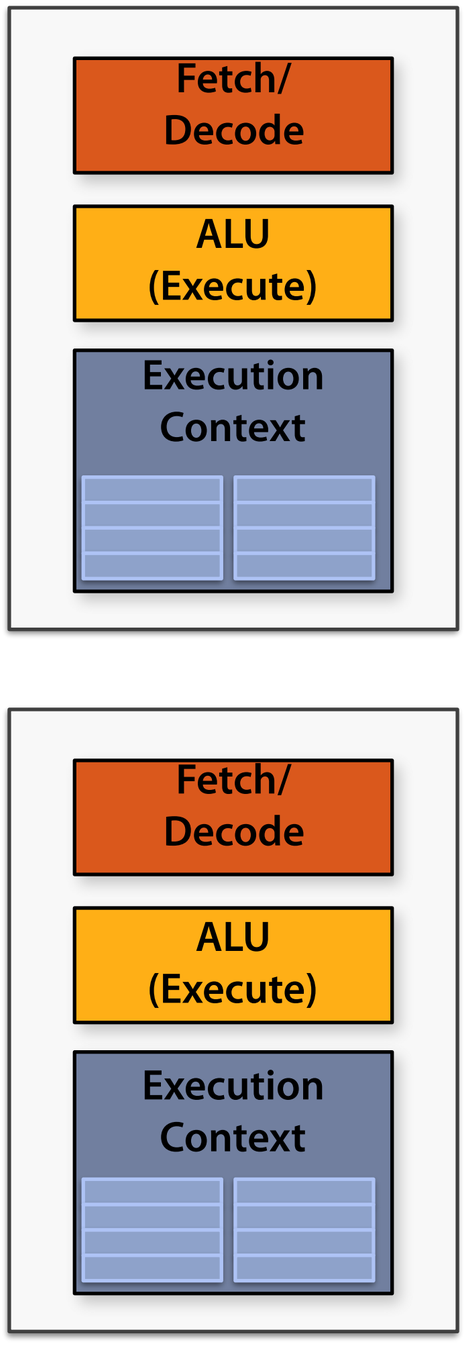


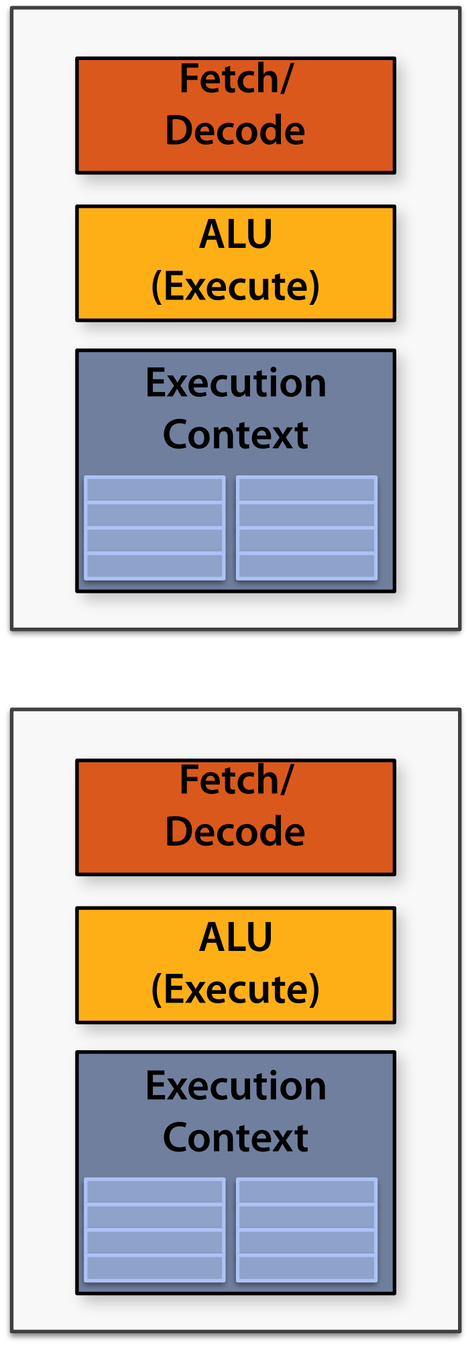


Four cores (four fragments in parallel)

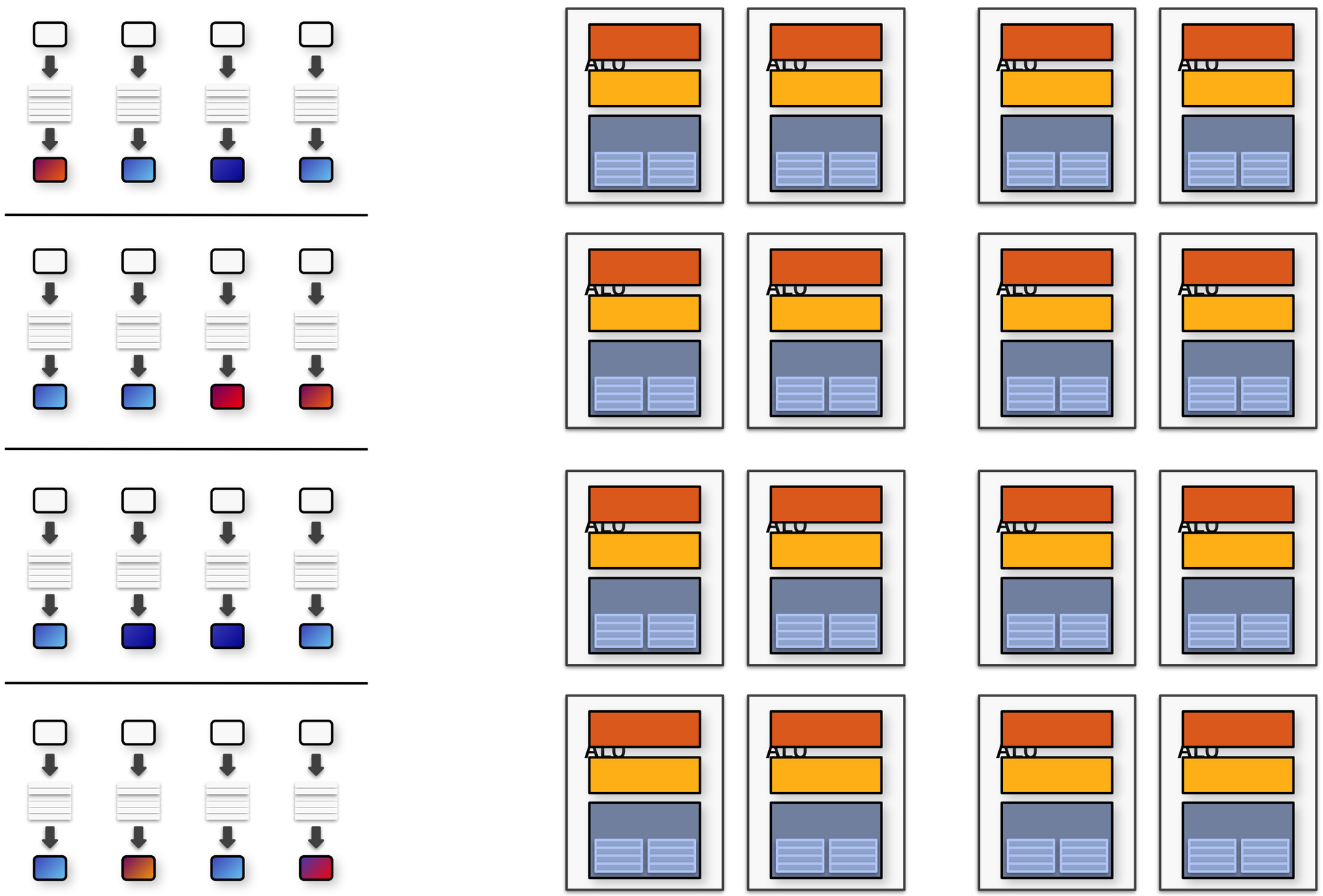






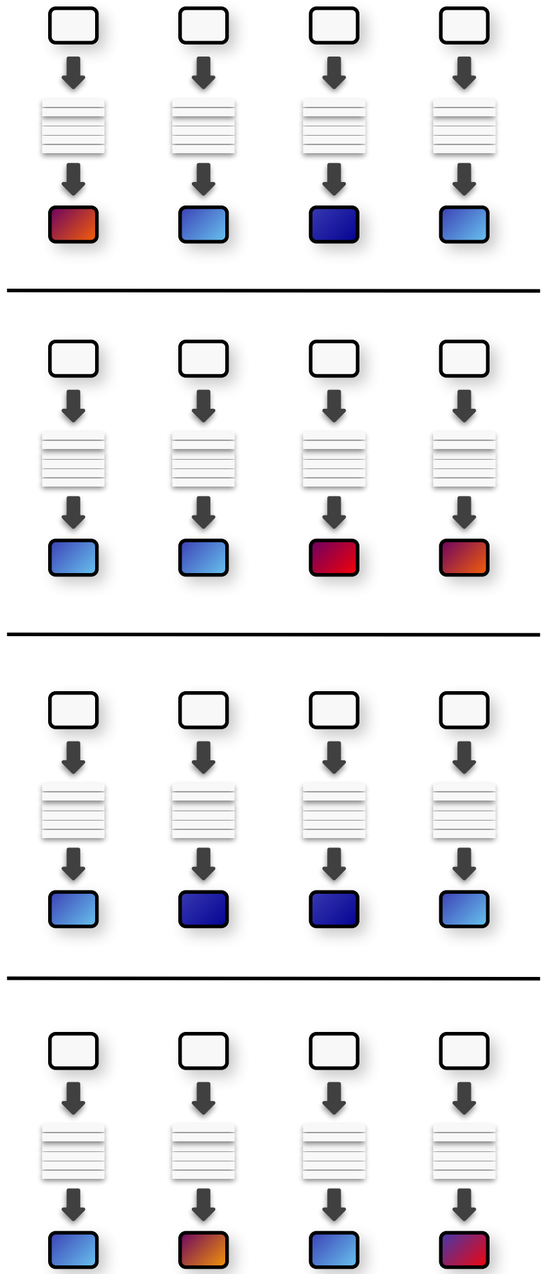


Sixteen cores (sixteen fragments in parallel)



16 cores = 16 simultaneous instruction streams

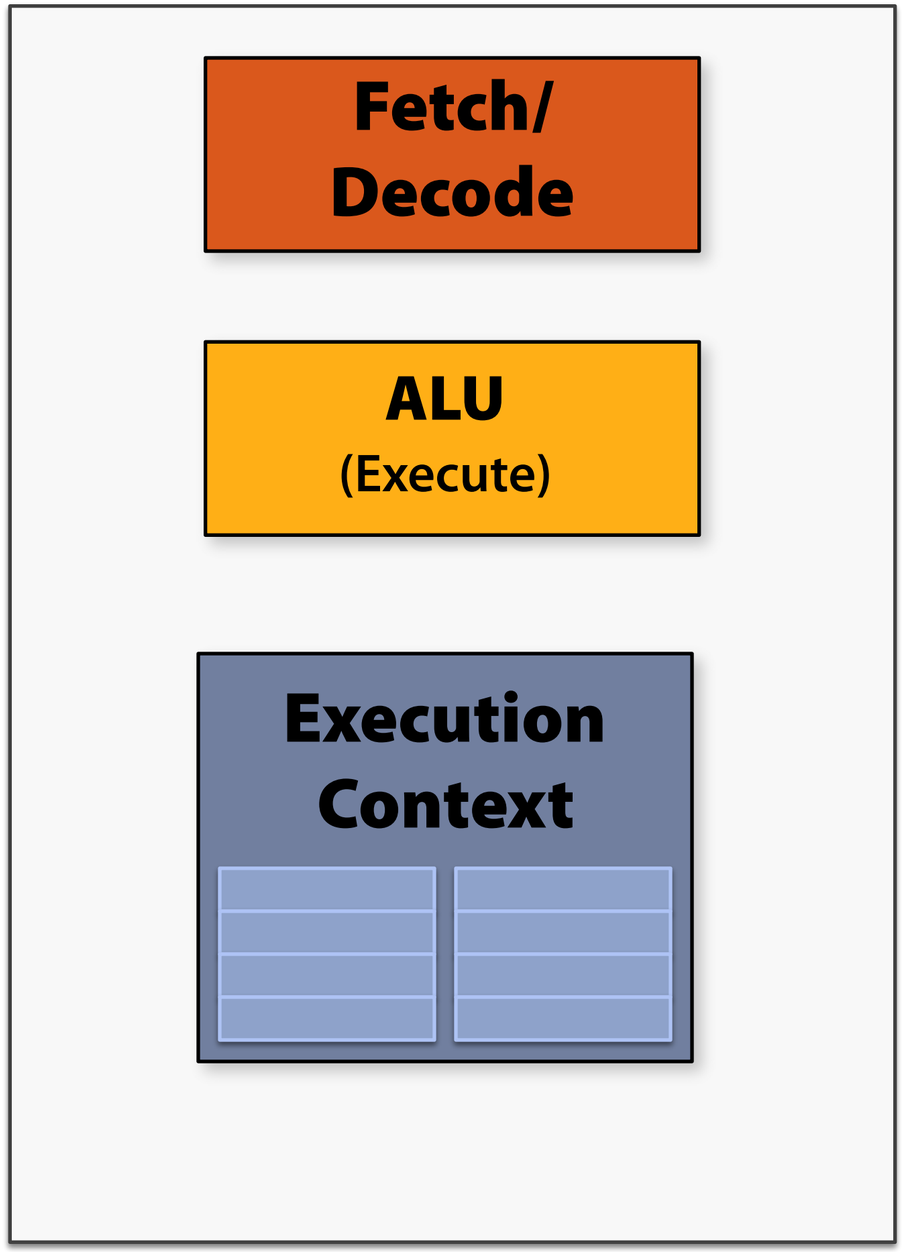
Instruction stream sharing



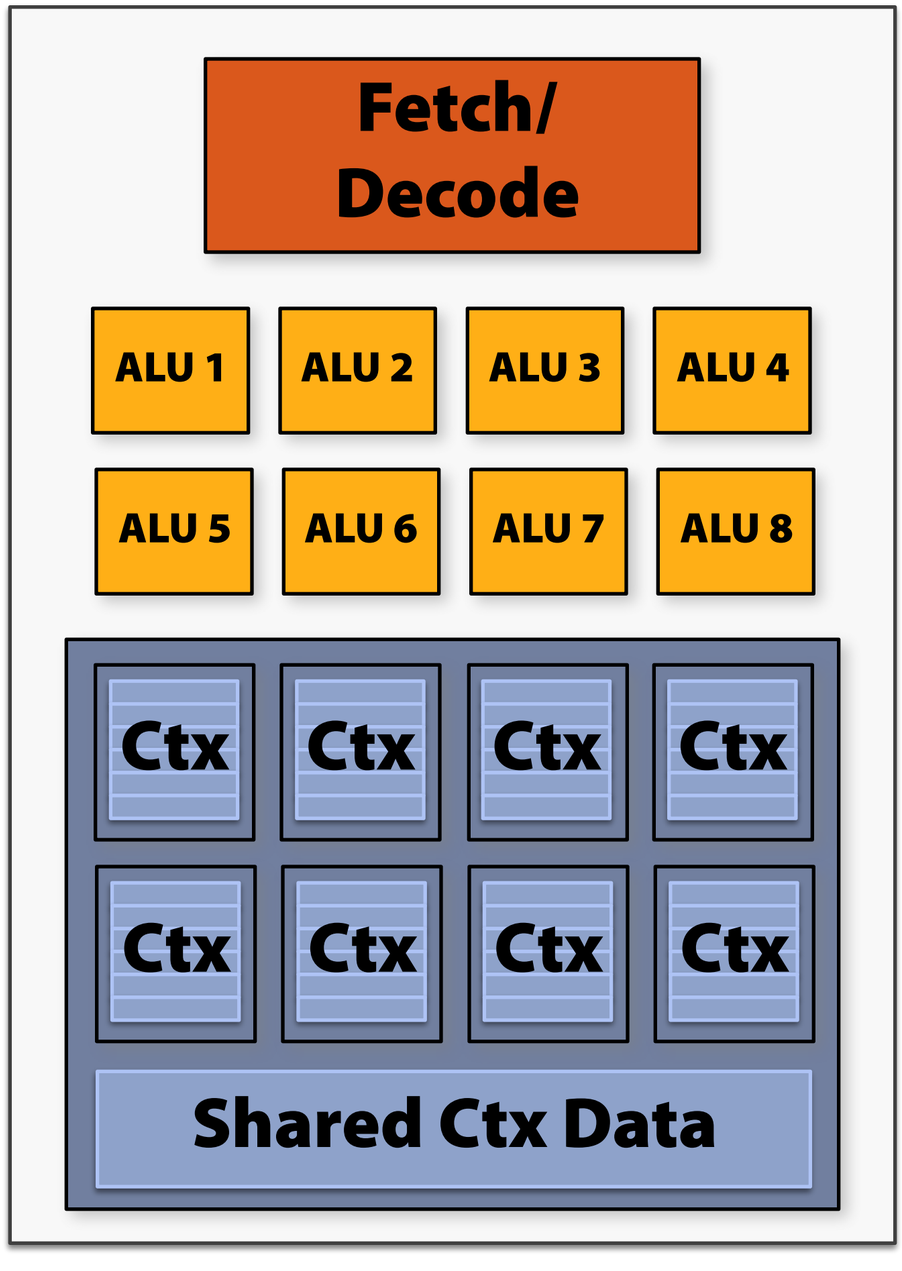
But… many fragments *should* be able to share an instruction stream!

```
<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

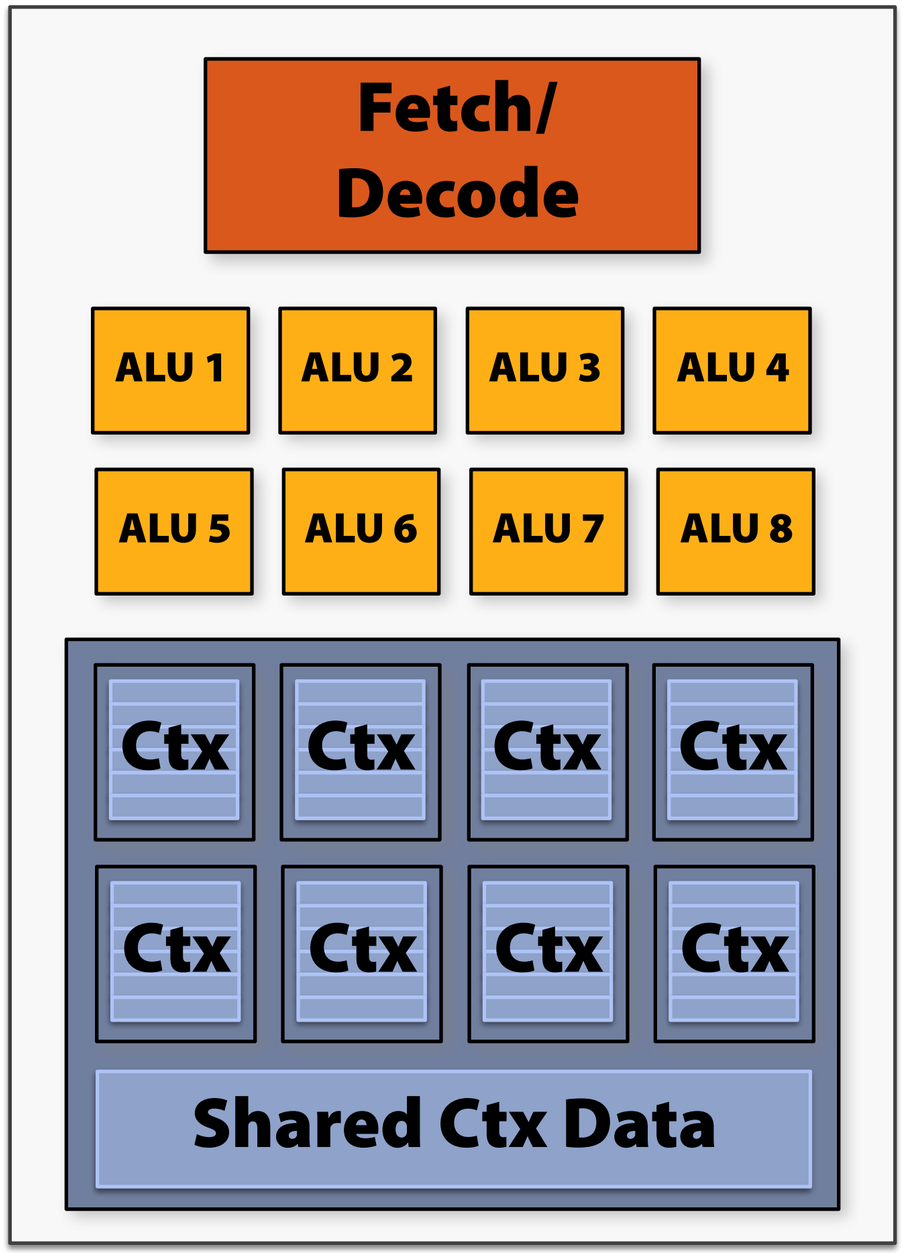


Idea #2:

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

SIMD processing

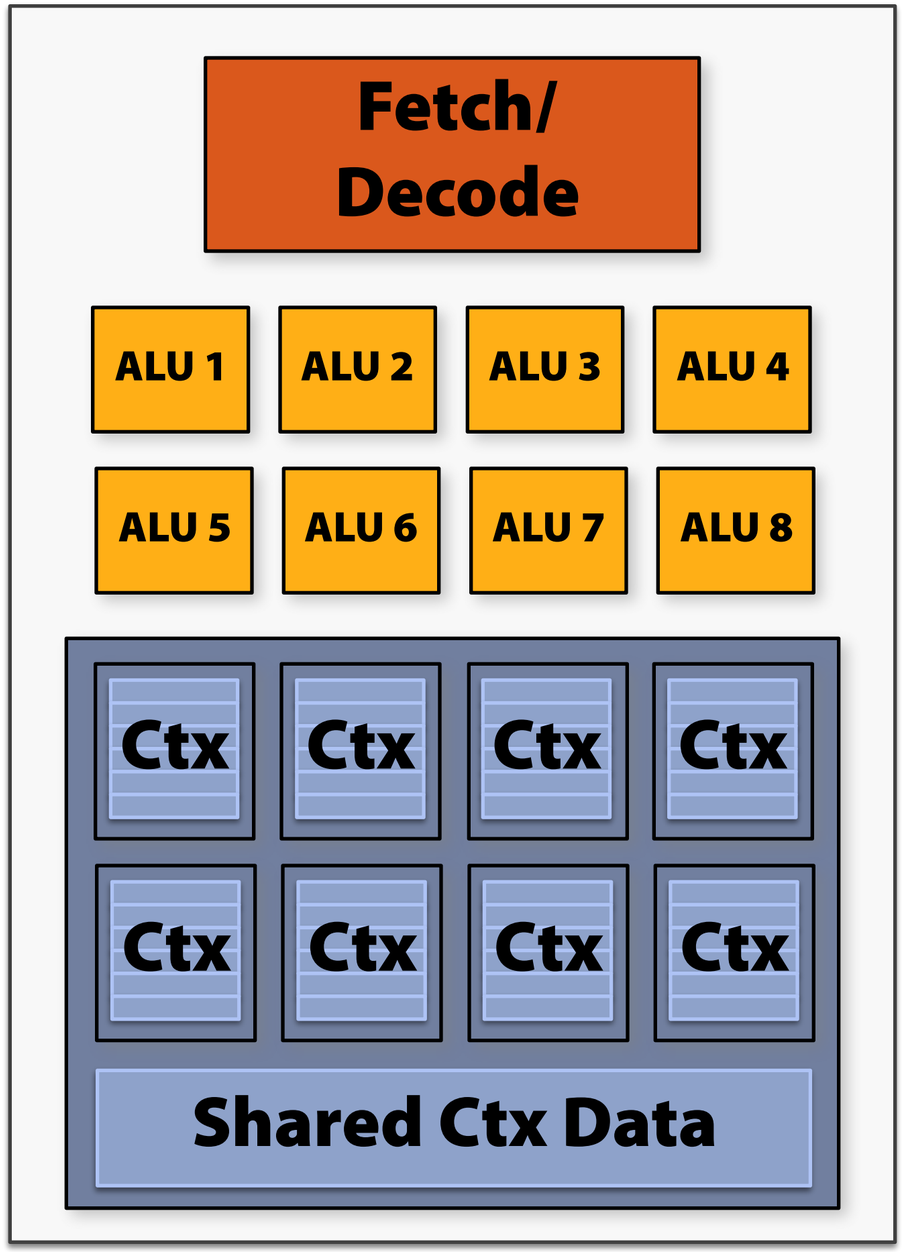
Modifying the shader




```
<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

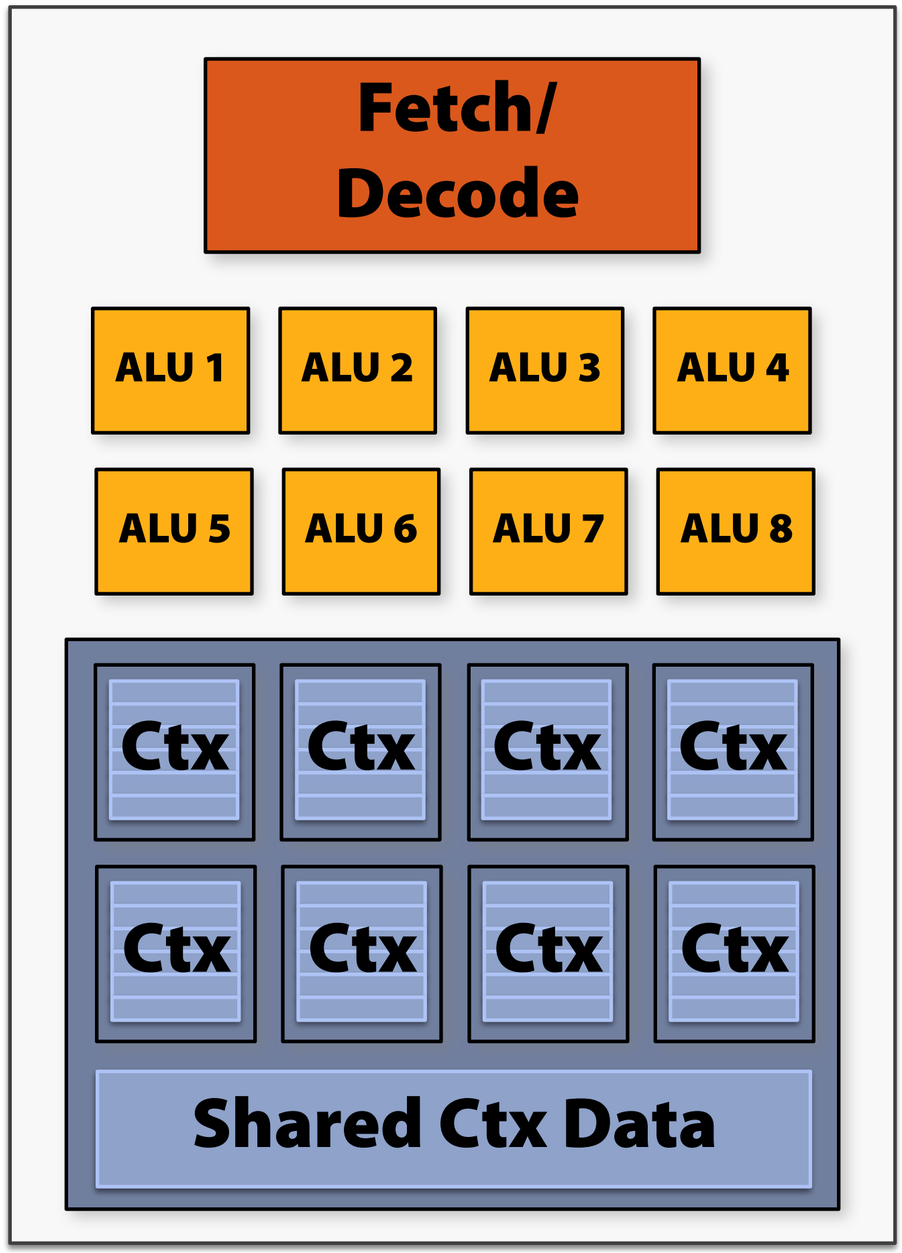
Modifying the shader

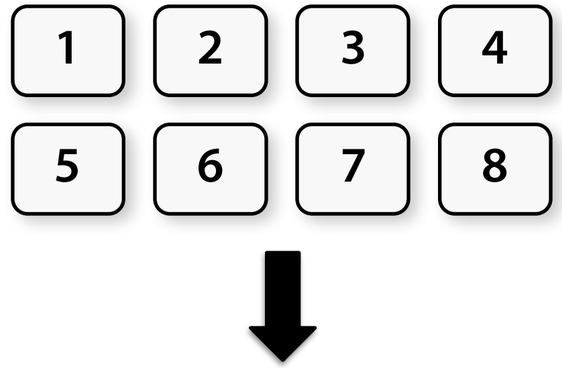



```
<VEC8_diffuseShader>:
VEC8_sample vec_r0, vec_v4, t0, vec_s0
VEC8_mul  vec_r3, vec_v0, cb0[0]
VEC8_madd vec_r3, vec_v1, cb0[1], vec_r3
VEC8_madd vec_r3, vec_v2, cb0[2], vec_r3
VEC8_clmp vec_r3, vec_r3, l(0.0), l(1.0)
VEC8_mul  vec_o0, vec_r0, vec_r3
VEC8_mul  vec_o1, vec_r1, vec_r3
VEC8_mul  vec_o2,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

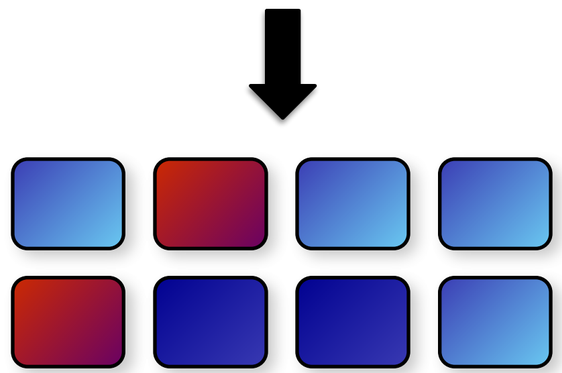
New compiled shader: Processes 8 fragments using vector ops on vector registers

Modifying the shader

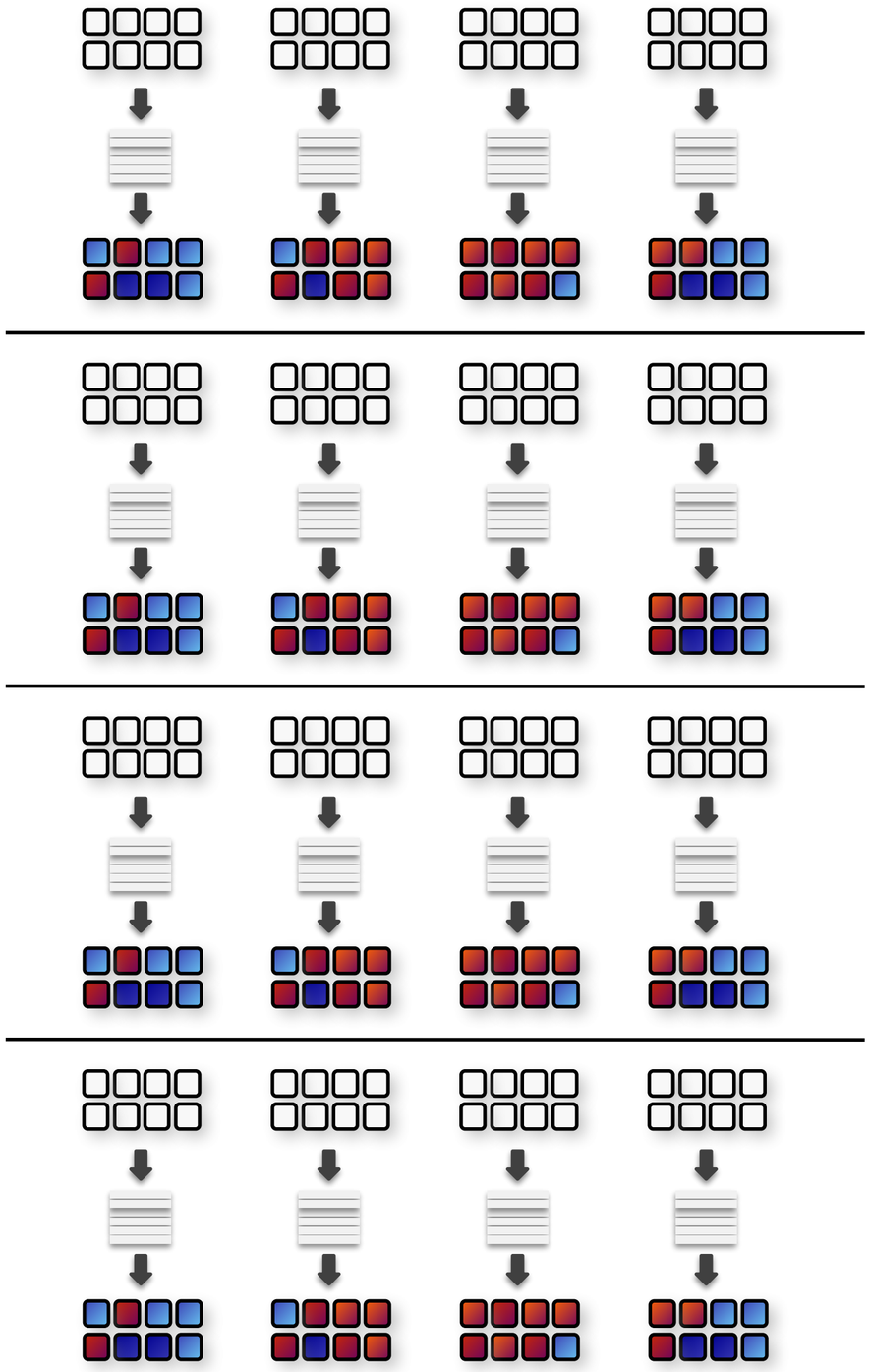


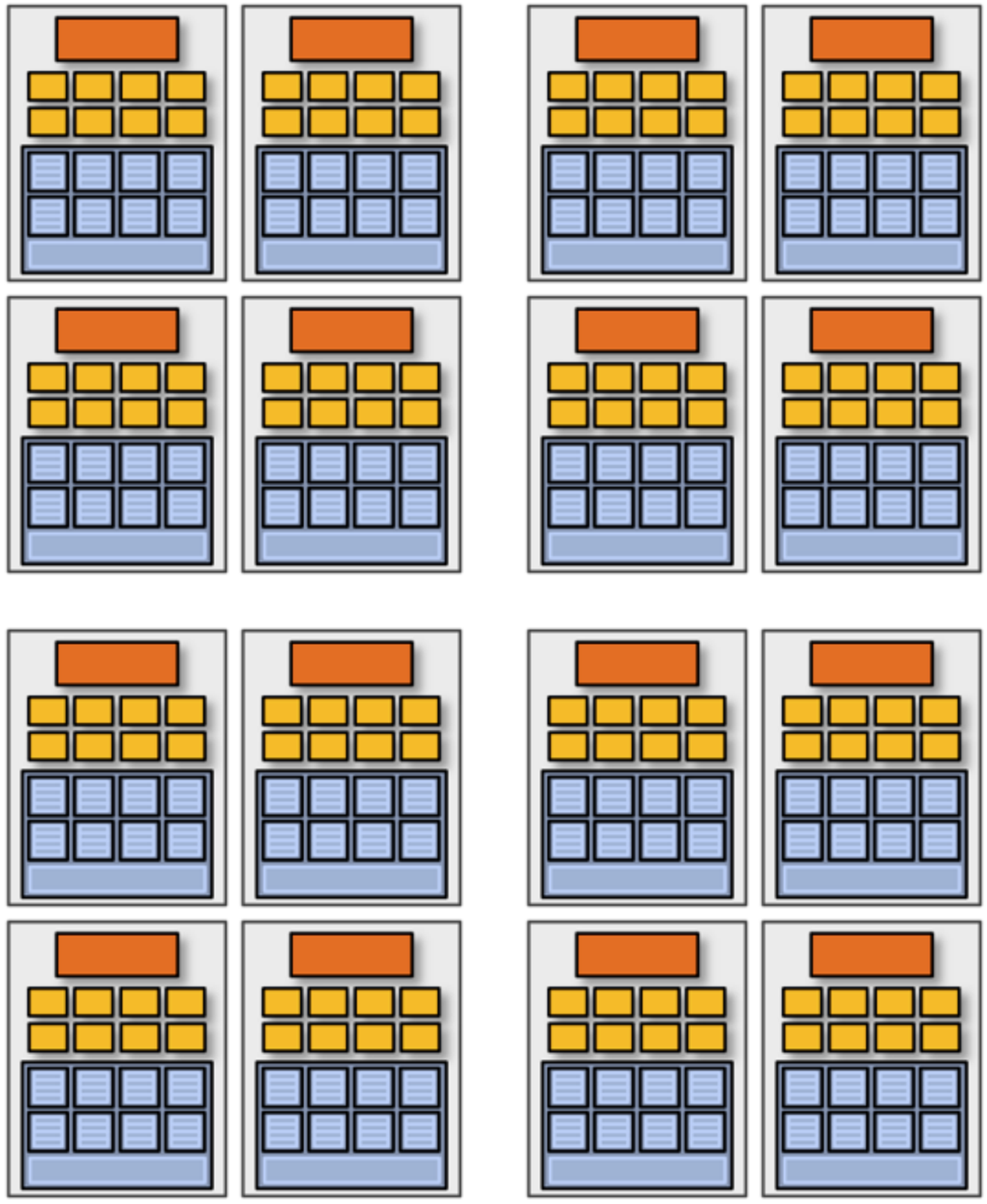



```
<VEC8_diffuseShader>:
VEC8_sample vec_r0, vec_v4, t0, vec_s0
VEC8_mul  vec_r3, vec_v0, cb0[0]
VEC8_madd vec_r3, vec_v1, cb0[1], vec_r3
VEC8_madd vec_r3, vec_v2, cb0[2], vec_r3
VEC8_clmp vec_r3, vec_r3, l(0.0), l(1.0)
VEC8_mul  vec_o0, vec_r0, vec_r3
VEC8_mul  vec_o1, vec_r1, vec_r3
VEC8_mul  vec_o2,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



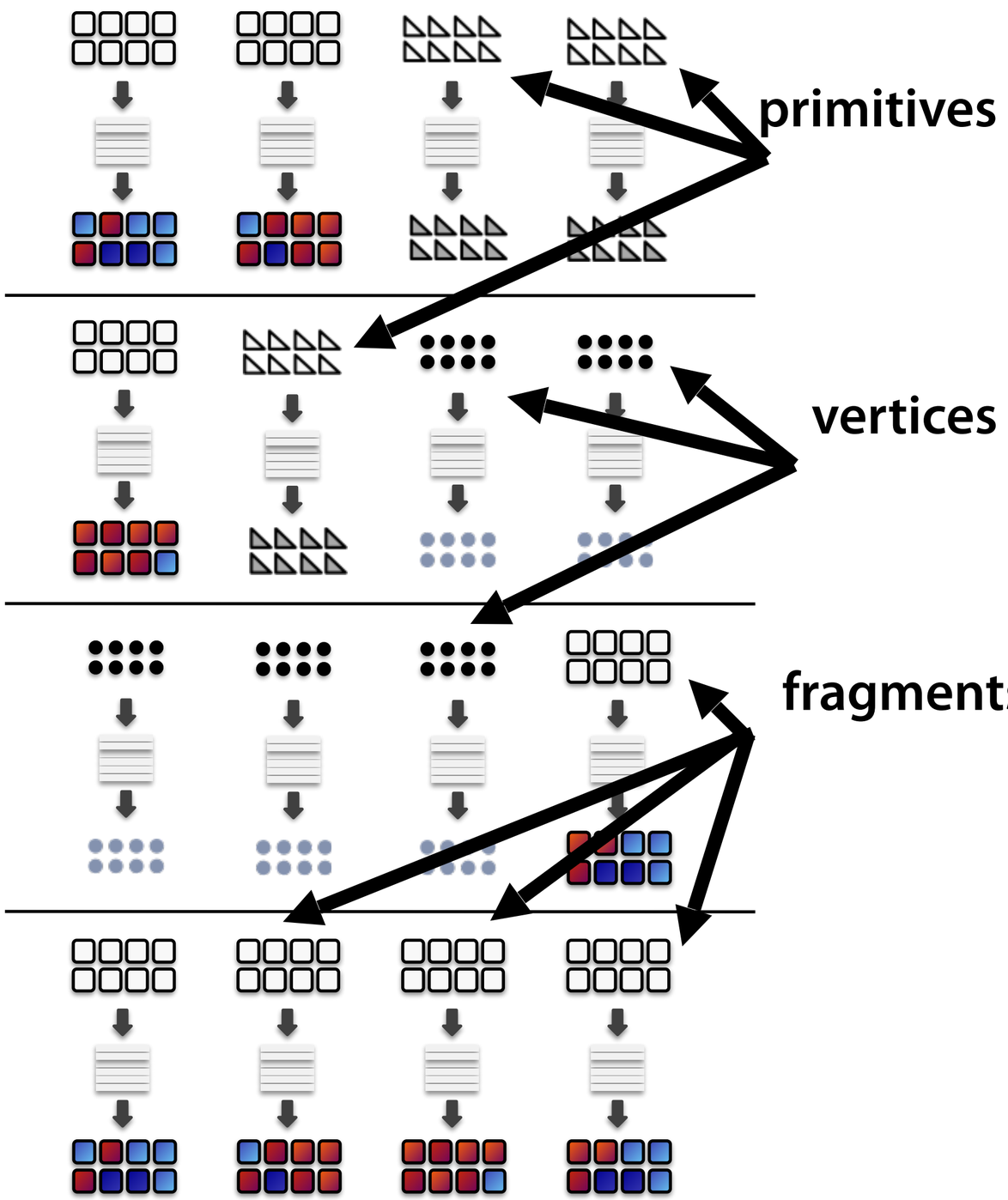
128 fragments in parallel

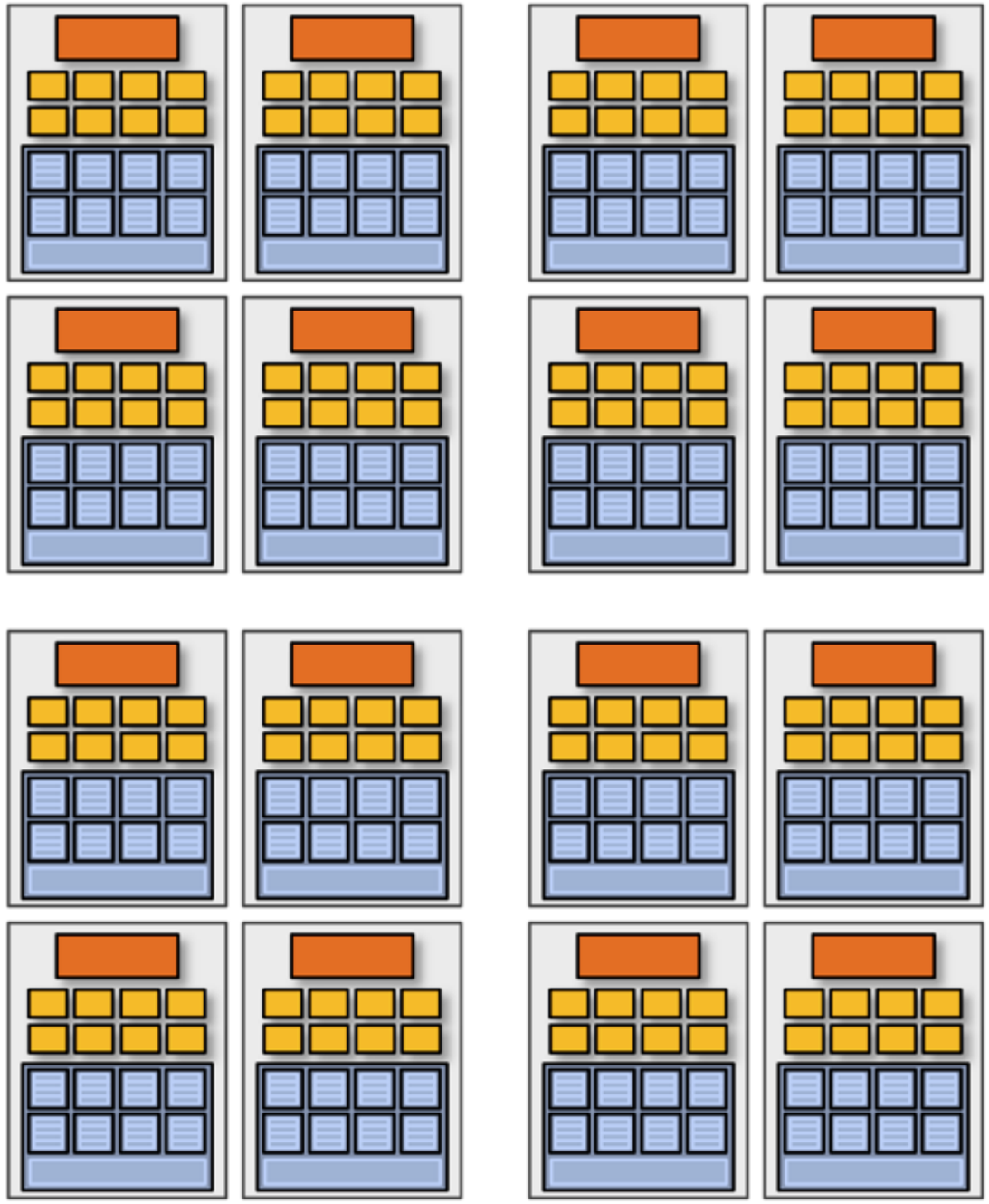




16 cores = 128 ALUs = 16 simultaneous instruction streams

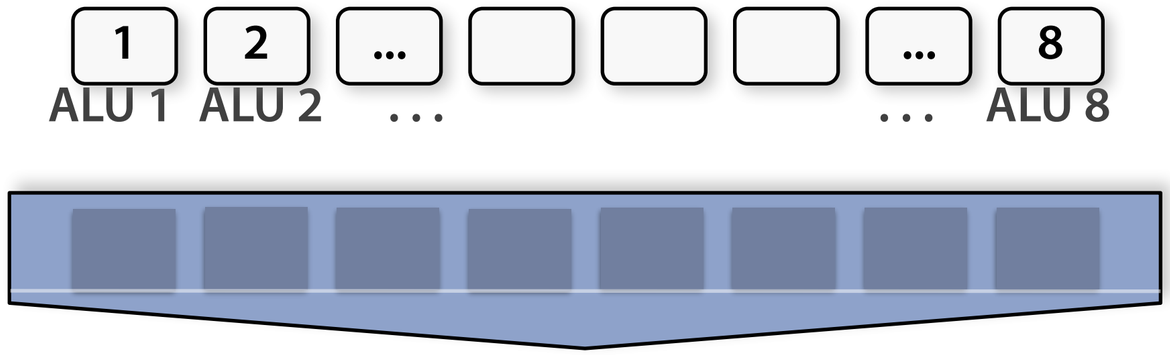
128 [vertices / fragments
primitives
CUDA threads
OpenCL work items
compute shader threads] in parallel





But what about branches?

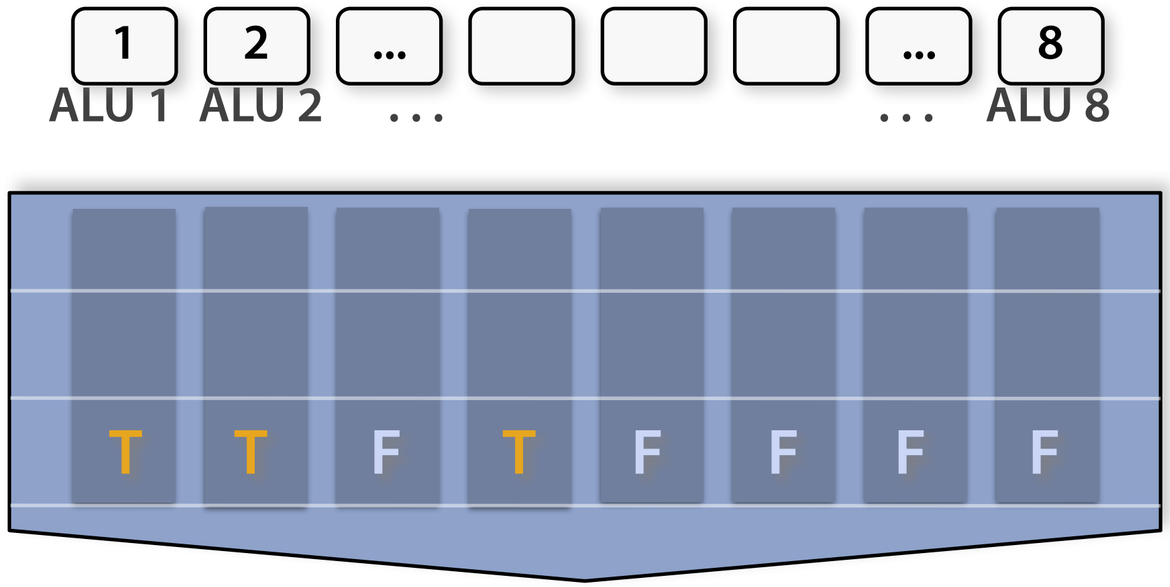
Time (clocks)




```
  <unconditional
   shader code>

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

    <resume
  unconditional
   shader code>
```

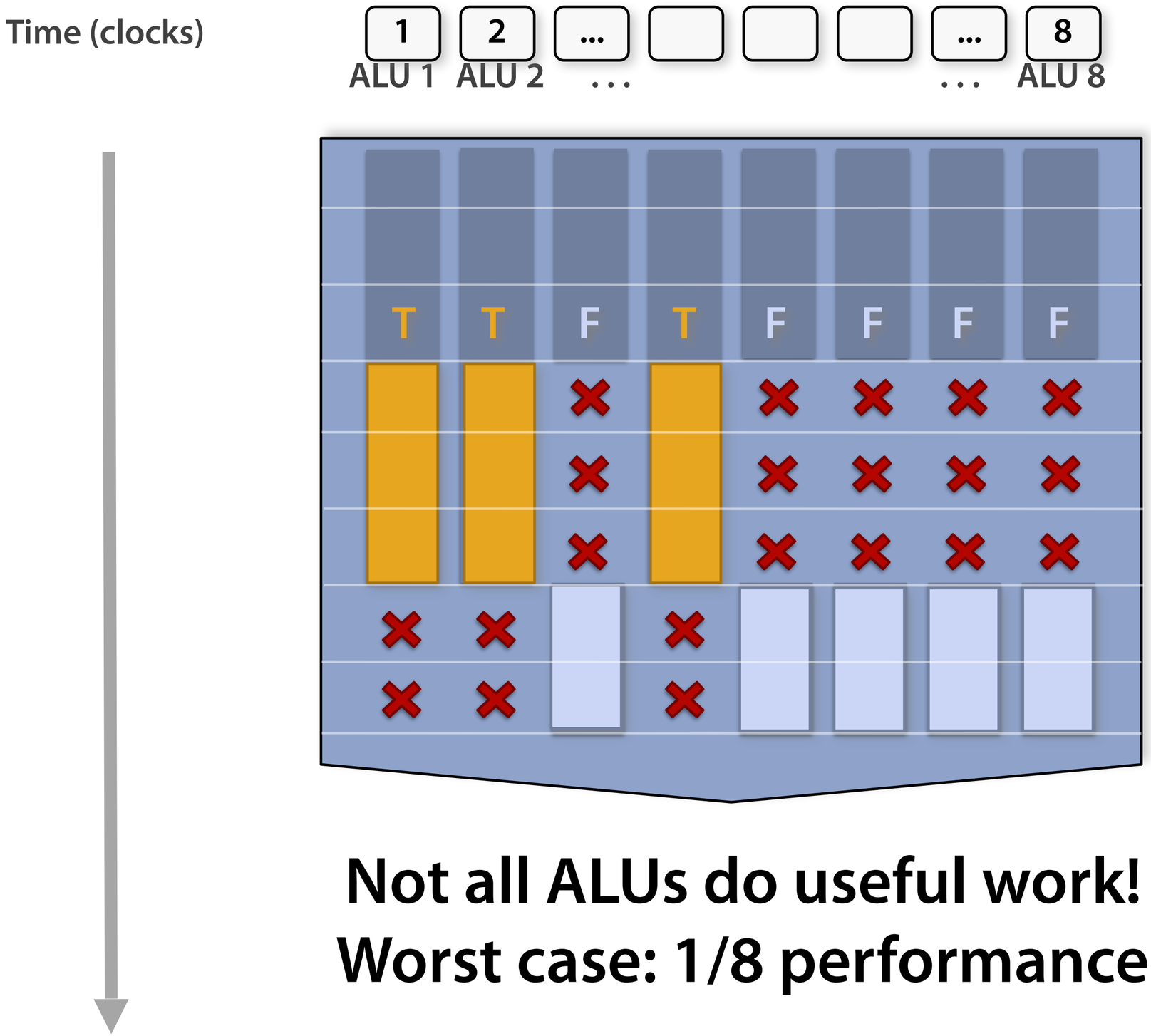
But what about branches?

Time (clocks)




```
<unconditional
 shader code>

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

   <resume
 unconditional
 shader code>
```

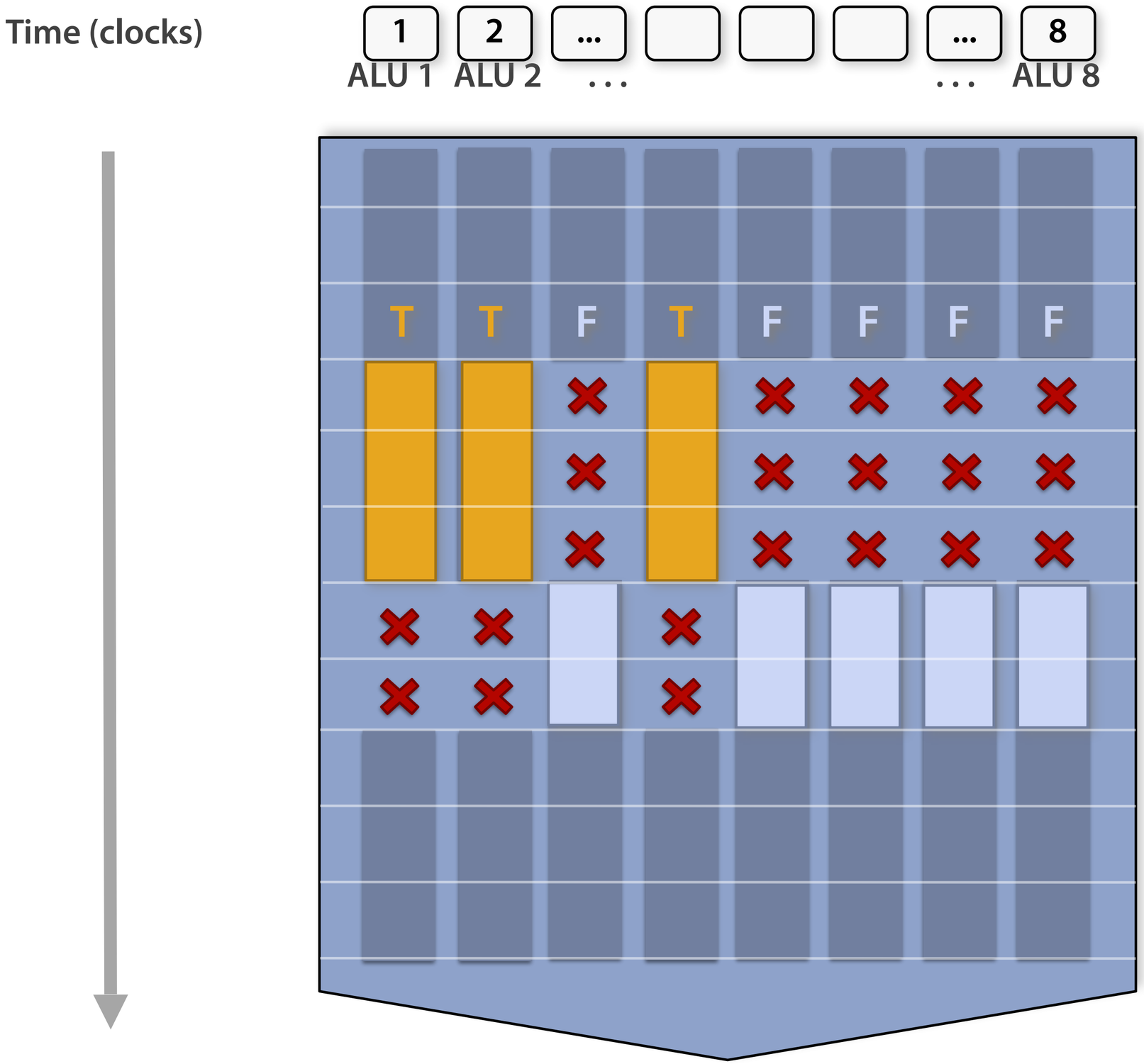
But what about branches?




```
<unconditional
 shader code>

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

<resume
unconditional
shader code>
```

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

But what about branches?



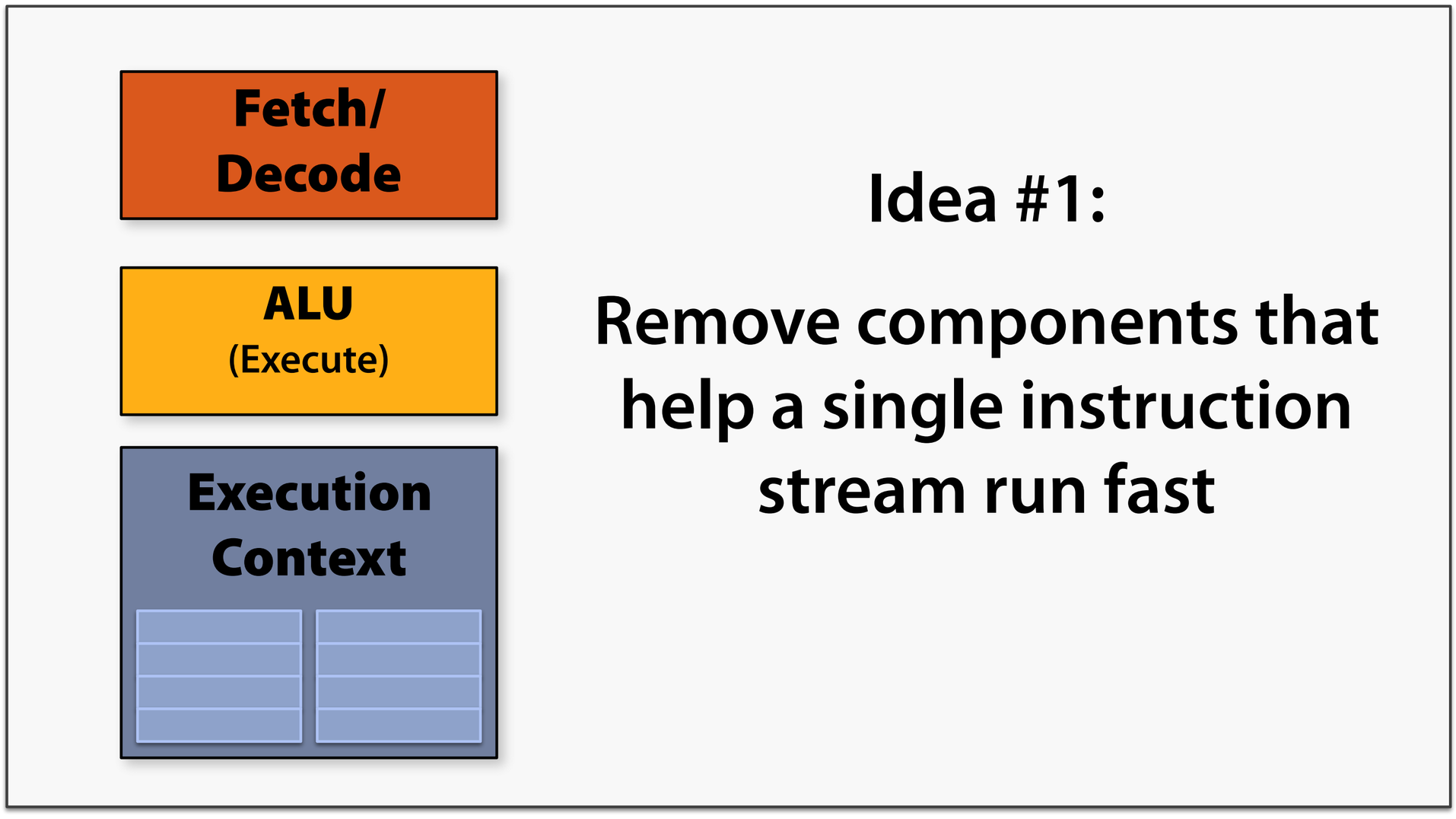

```
<unconditional
 shader code>

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

<resume
unconditional
shader code>
```

Clarification

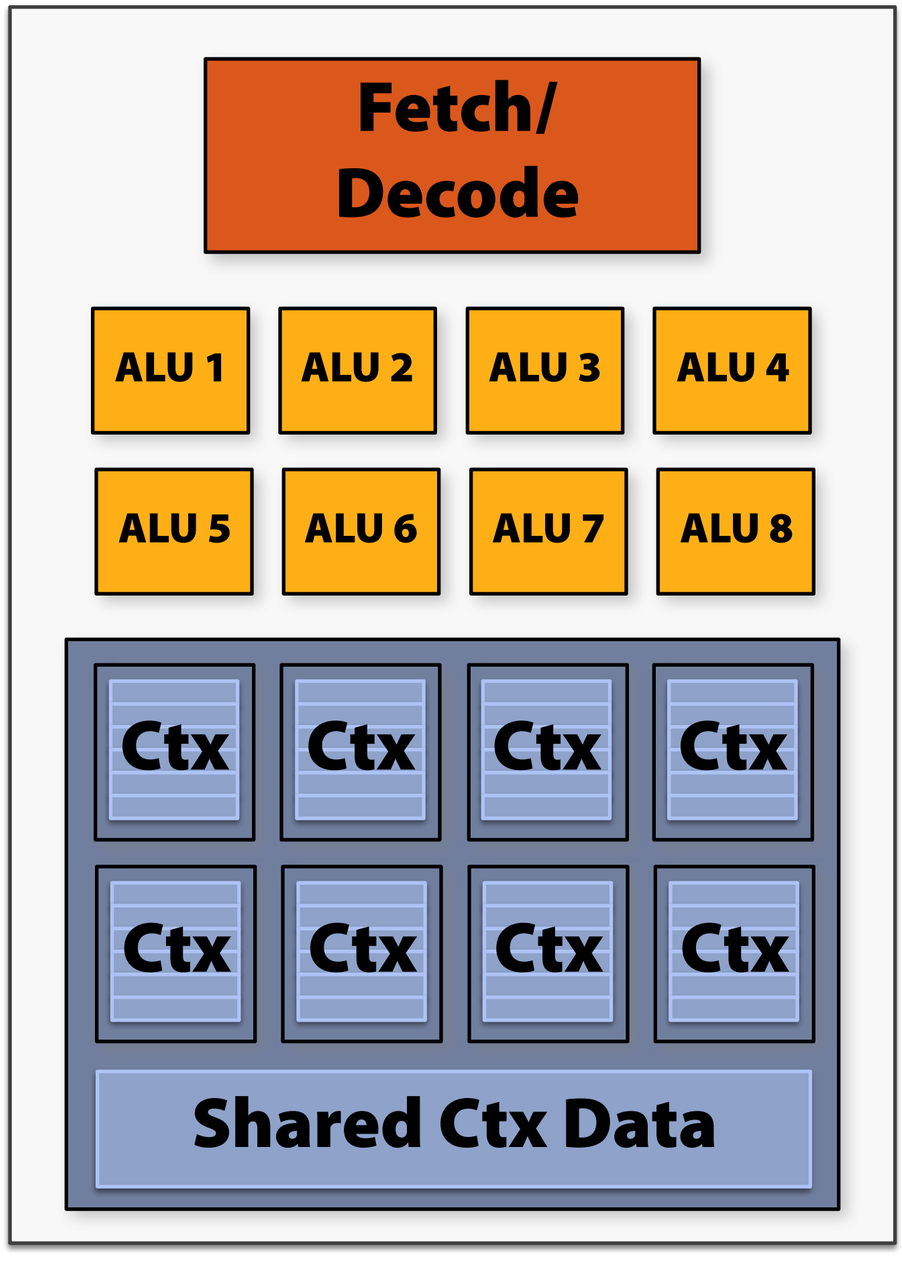
- SIMD processing does not imply SIMD instructions
- Option 1: Explicit vector instructions
  - Intel/AMD x86 SSE, Intel Larrabee
- Option 2: Scalar instructions, implicit HW vectorization
  - HW determines instruction stream sharing across ALUs (amount of sharing hidden from software)
  - NVIDIA GeForce ("SIMT" warps), AMD Radeon architectures

In practice: 16 to 64 fragments share an instruction stream

Slimming down



Add ALUs



Idea #2:

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

SIMD processing

Stalls!

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

Texture (more generally, memory) 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.

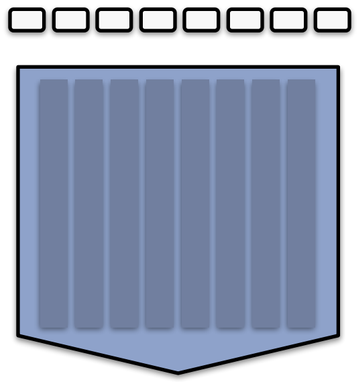
Idea #3:

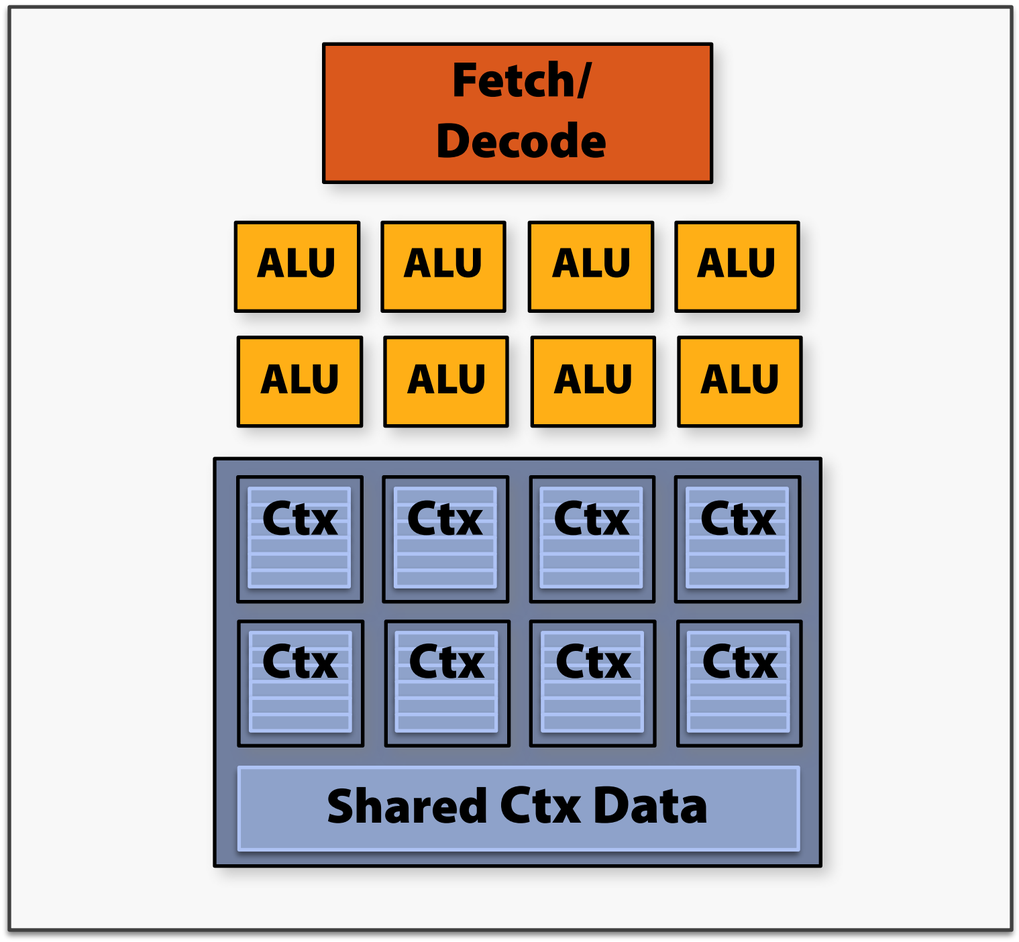
Interleave processing of many fragments on a single core to avoid stalls caused by high latency operations.

Hiding shader stalls

Time (clocks)

Frag 1 … 8





Hiding shader stalls

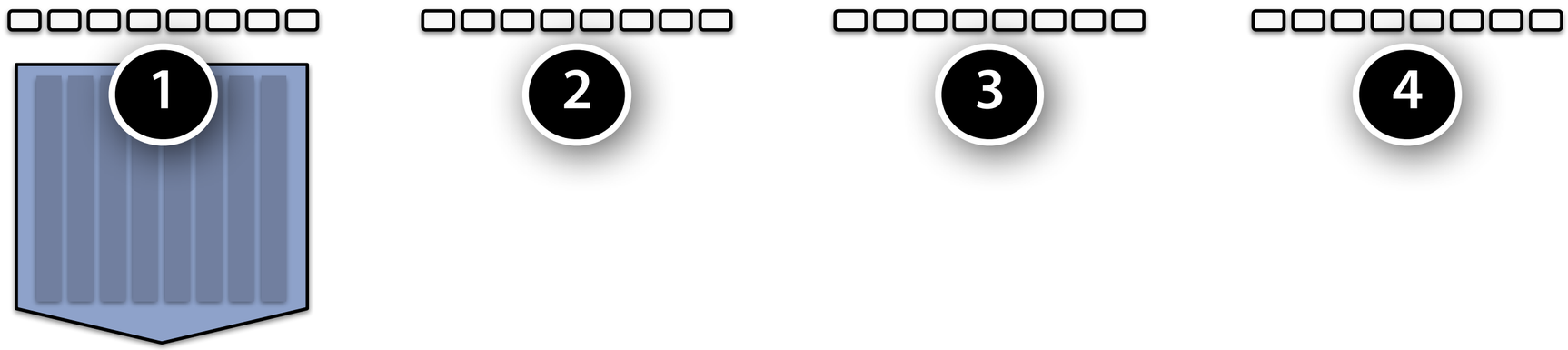
Time (clocks)

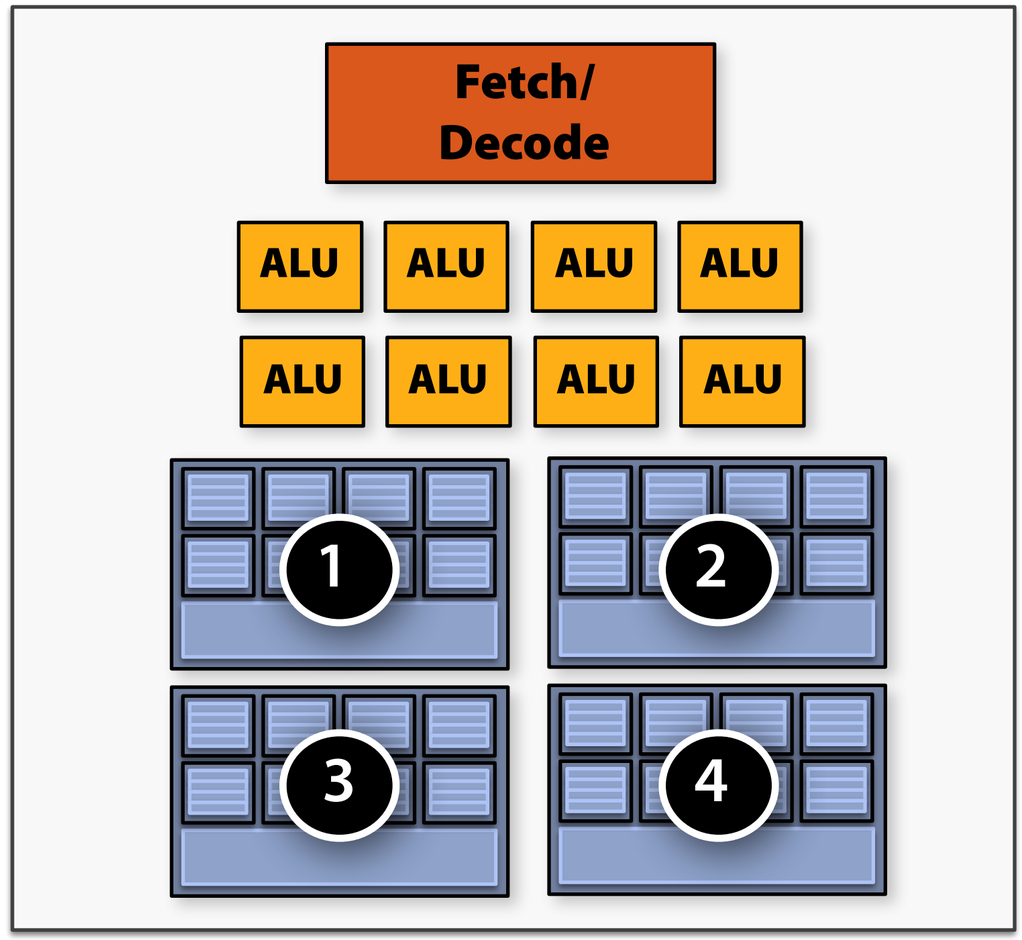
Frag 1 … 8

Frag 9… 16

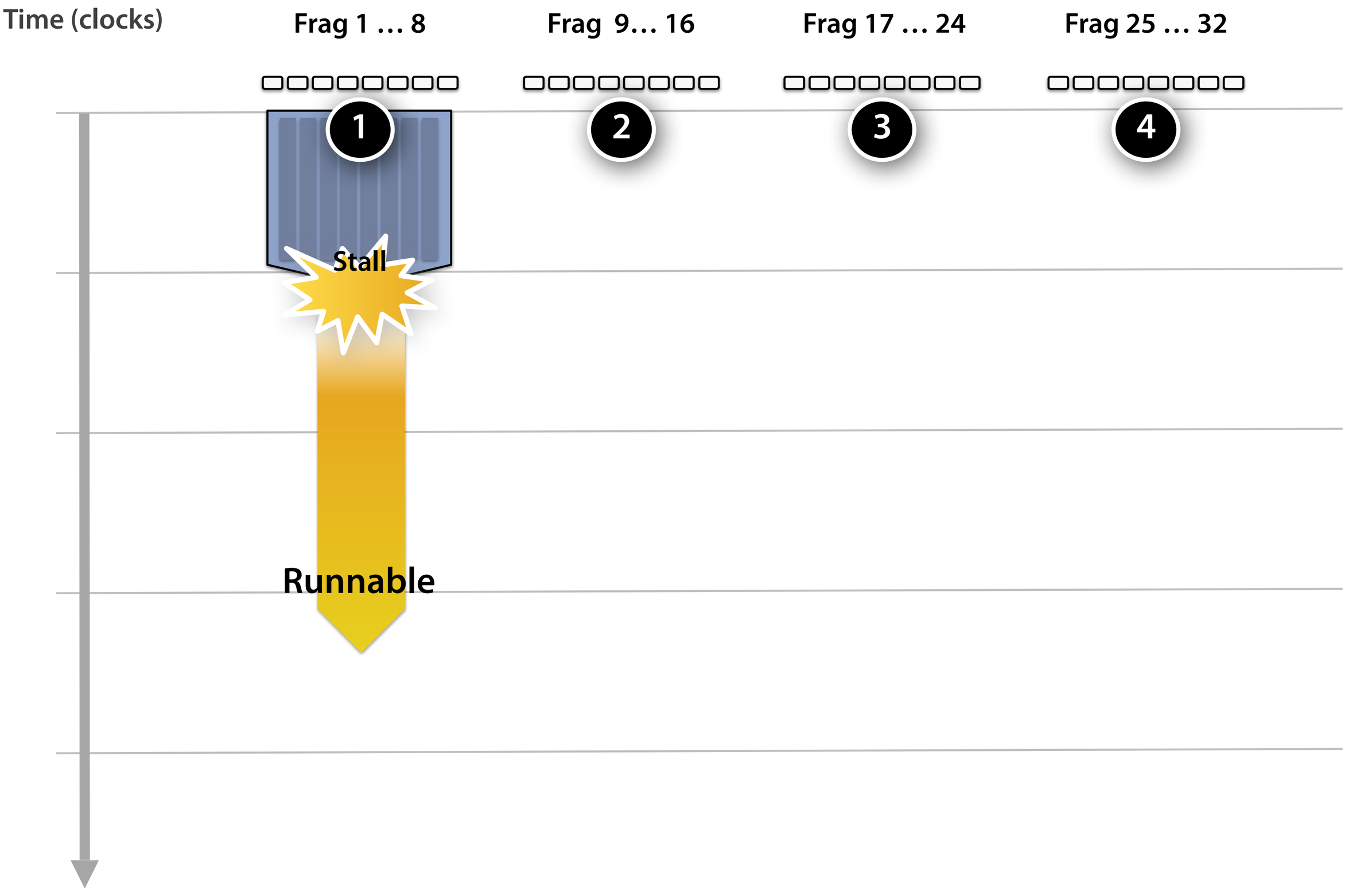
Frag 17 … 24

Frag 25 … 32

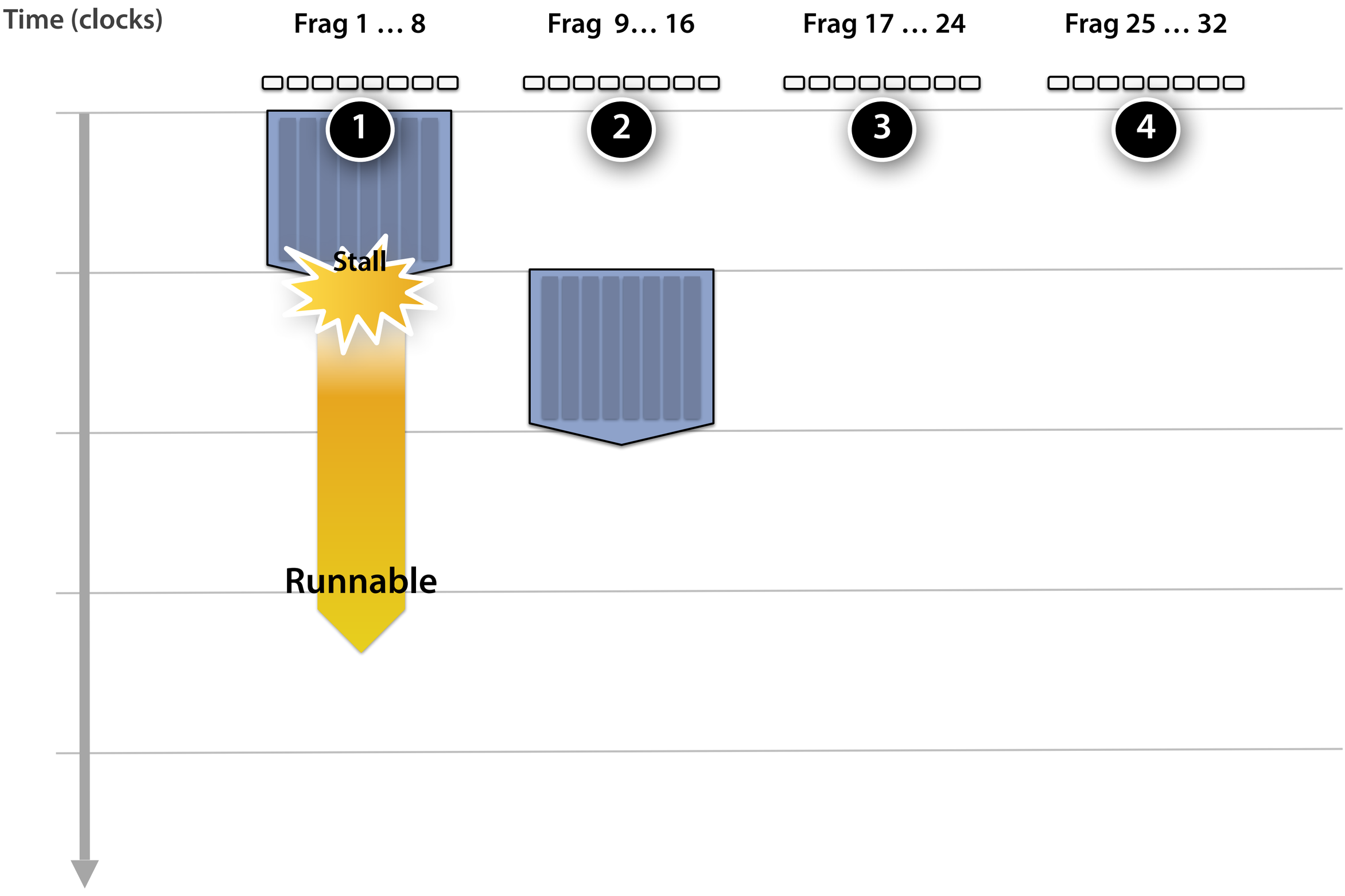




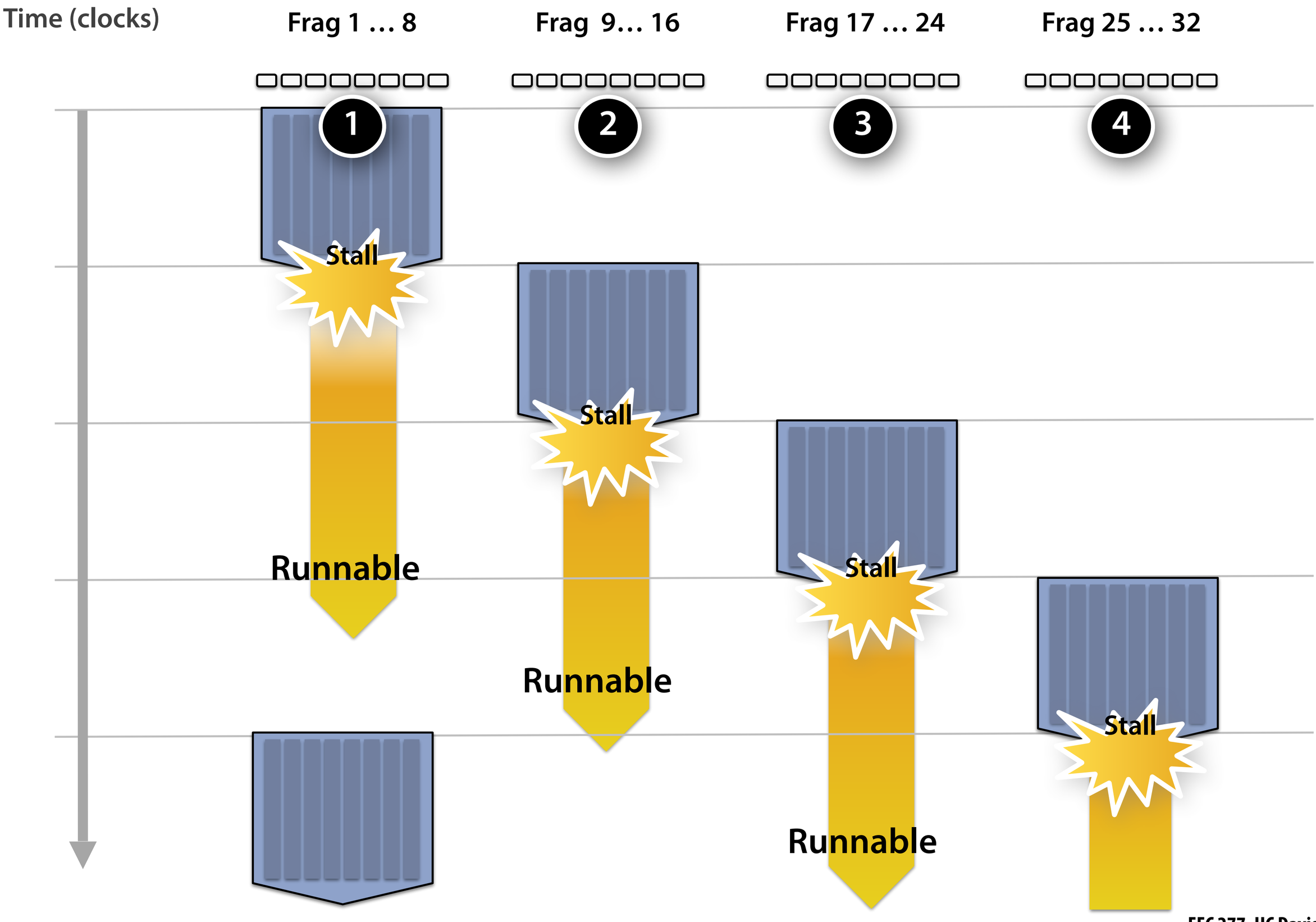
Hiding shader stalls



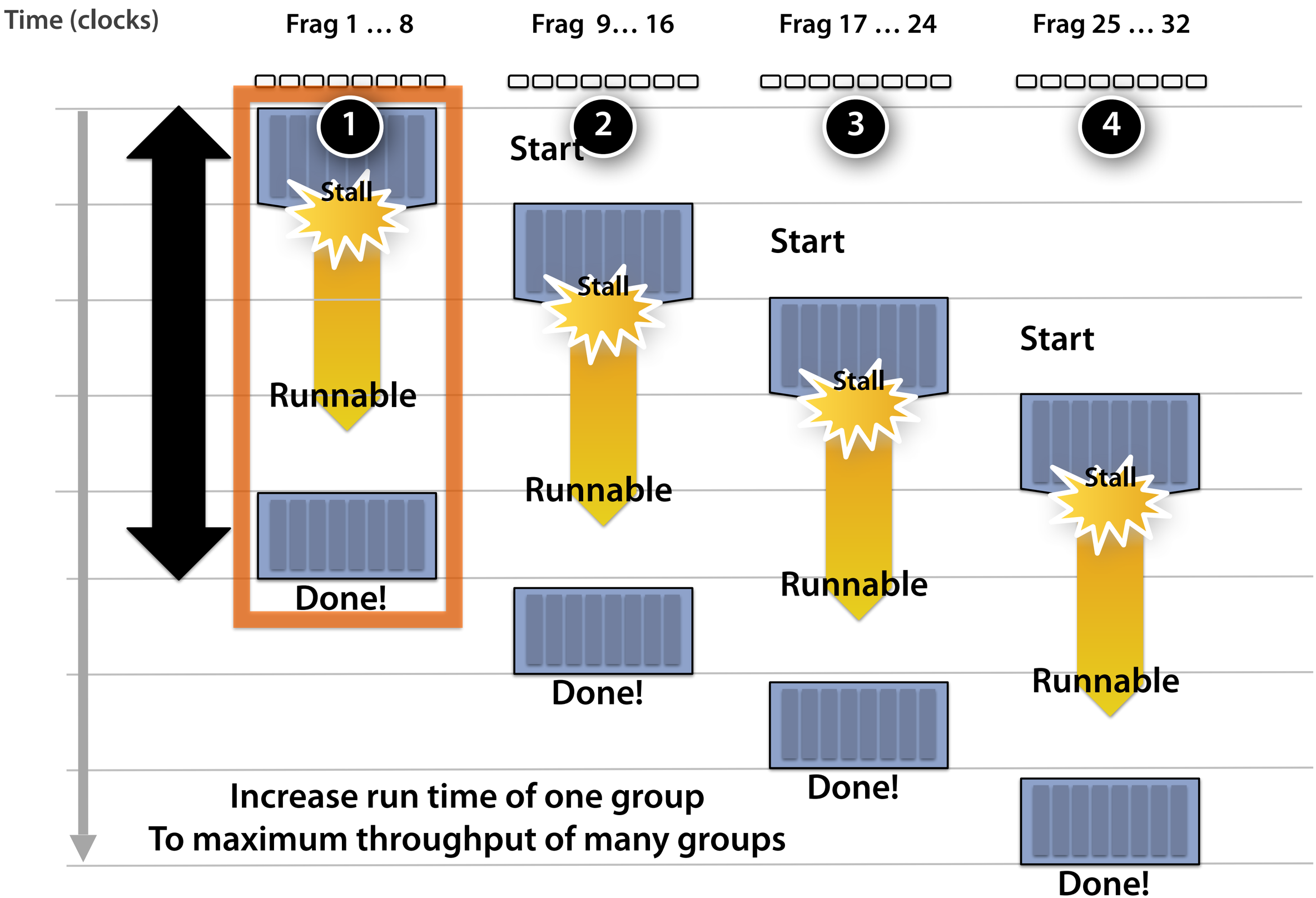
Hiding shader stalls



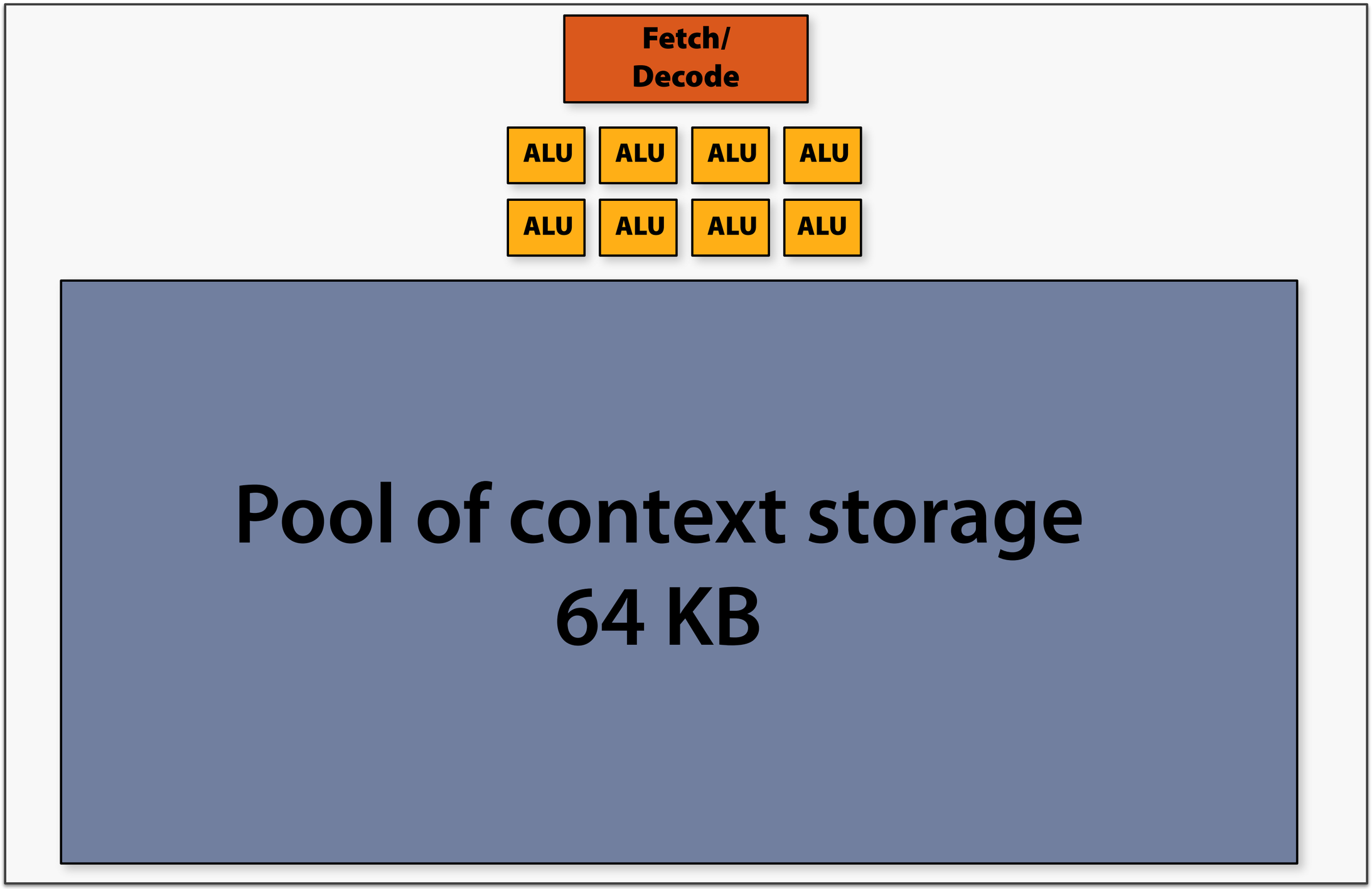
Hiding shader stalls



Throughput!

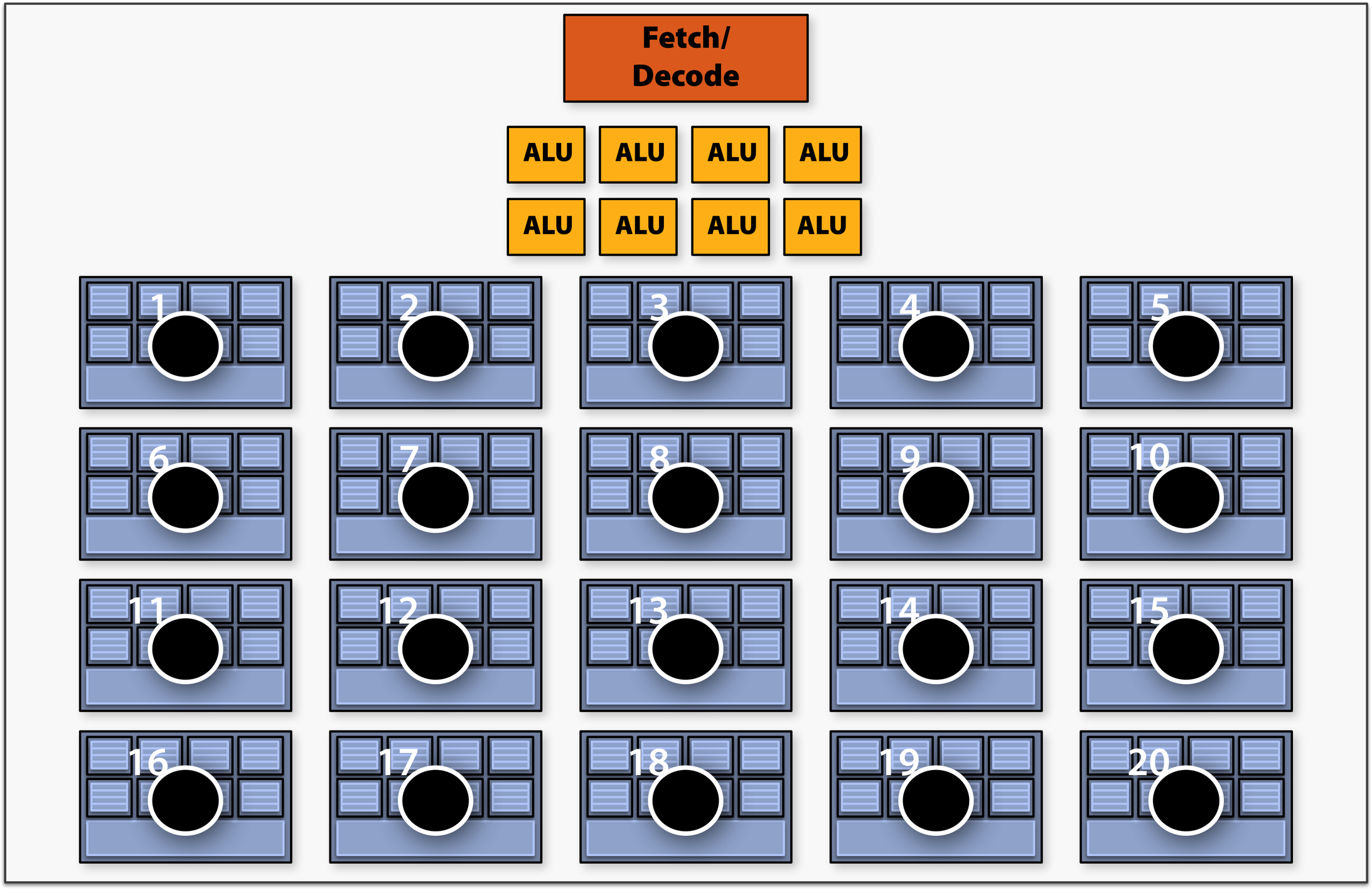

Time (clocks)
Frag 1 … 8
Frag 9… 16
Frag 17 … 24
Frag 25 … 32
1
2
3
4
Start
Stall
Runnable
Done!
Start
Stall
Runnable
Done!
Start
Stall
Runnable
Done!
Start
Stall
Runnable
Done!
Increase run time of one group
To maximum throughput of many groups

Storing contexts

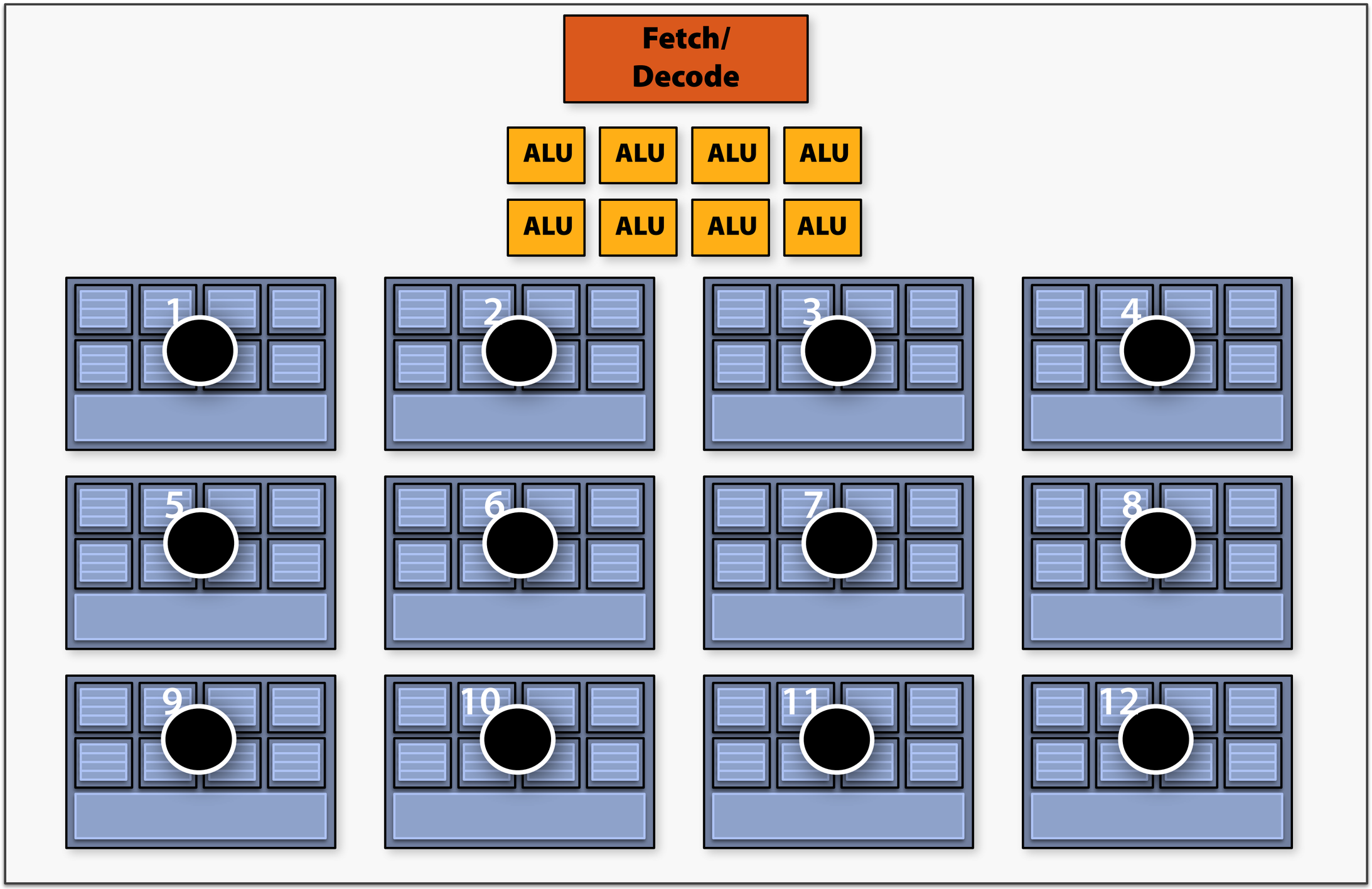


Twenty small contexts

(maximal latency hiding ability)

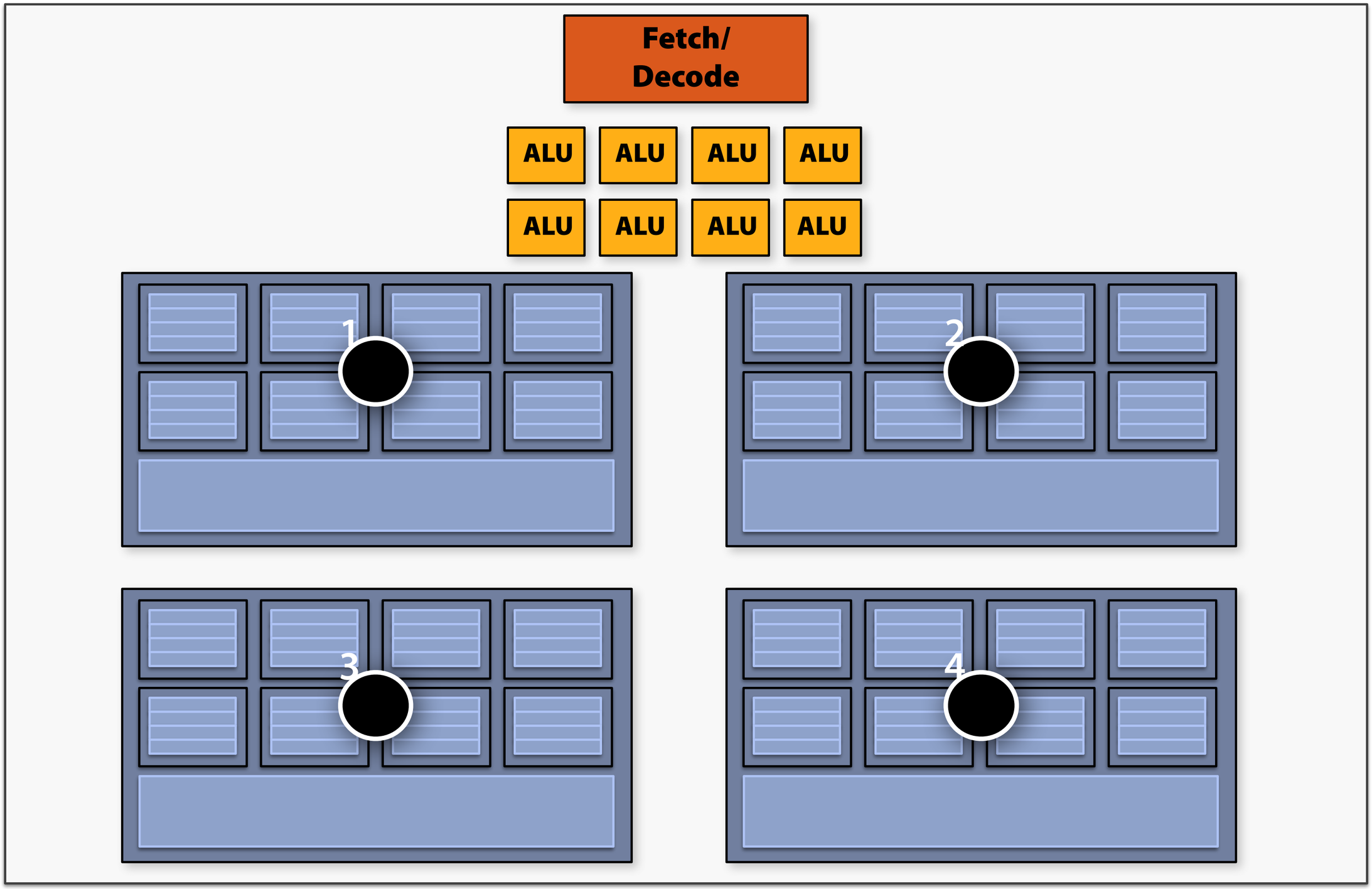


Twelve medium contexts



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

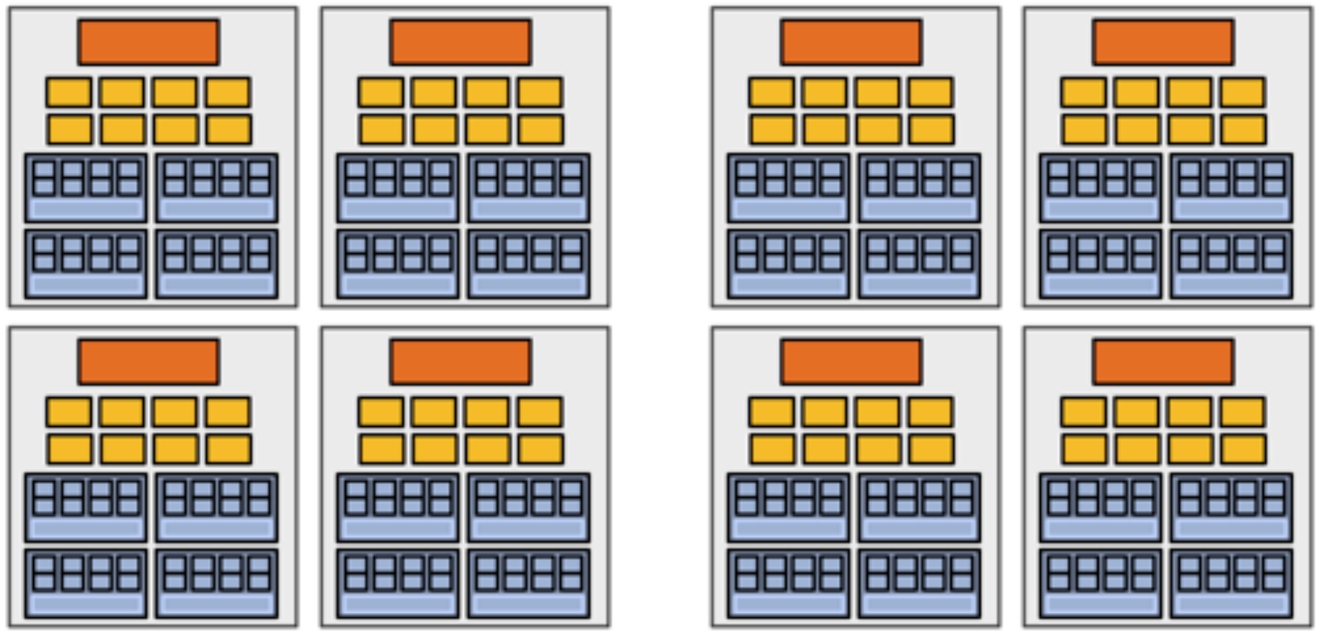
8 mul-add ALUs per core
(128 total)

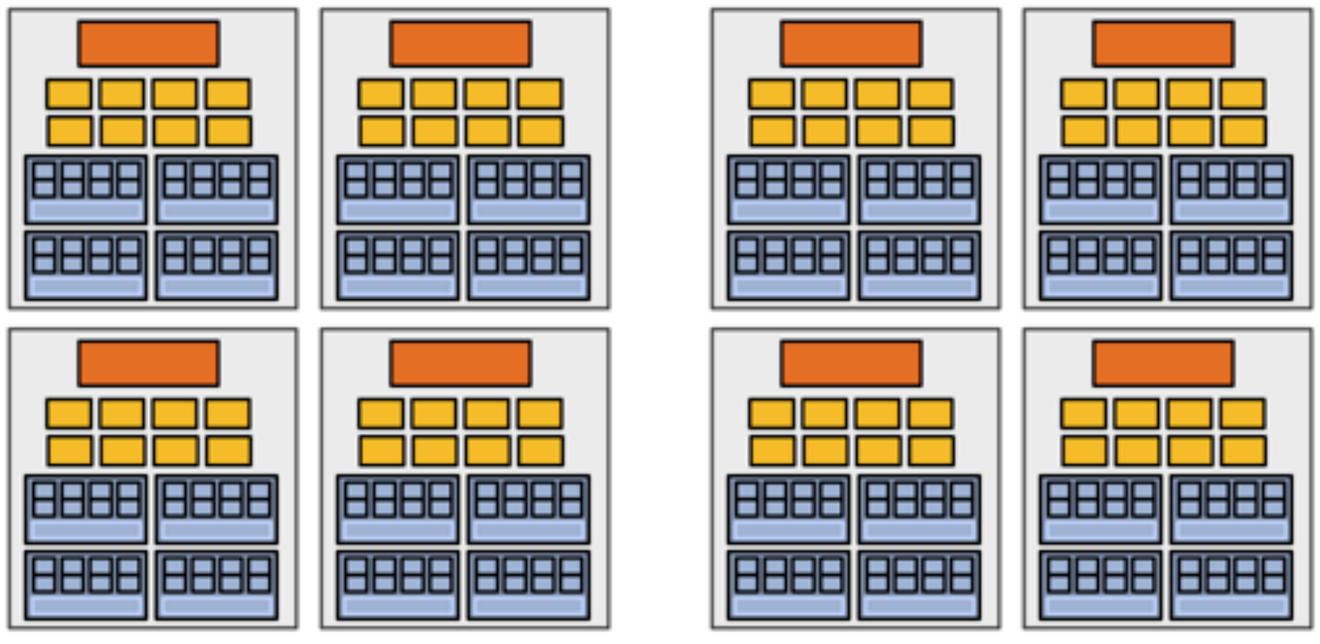
16 simultaneous
instruction streams

64 concurrent (but interleaved)
instruction streams

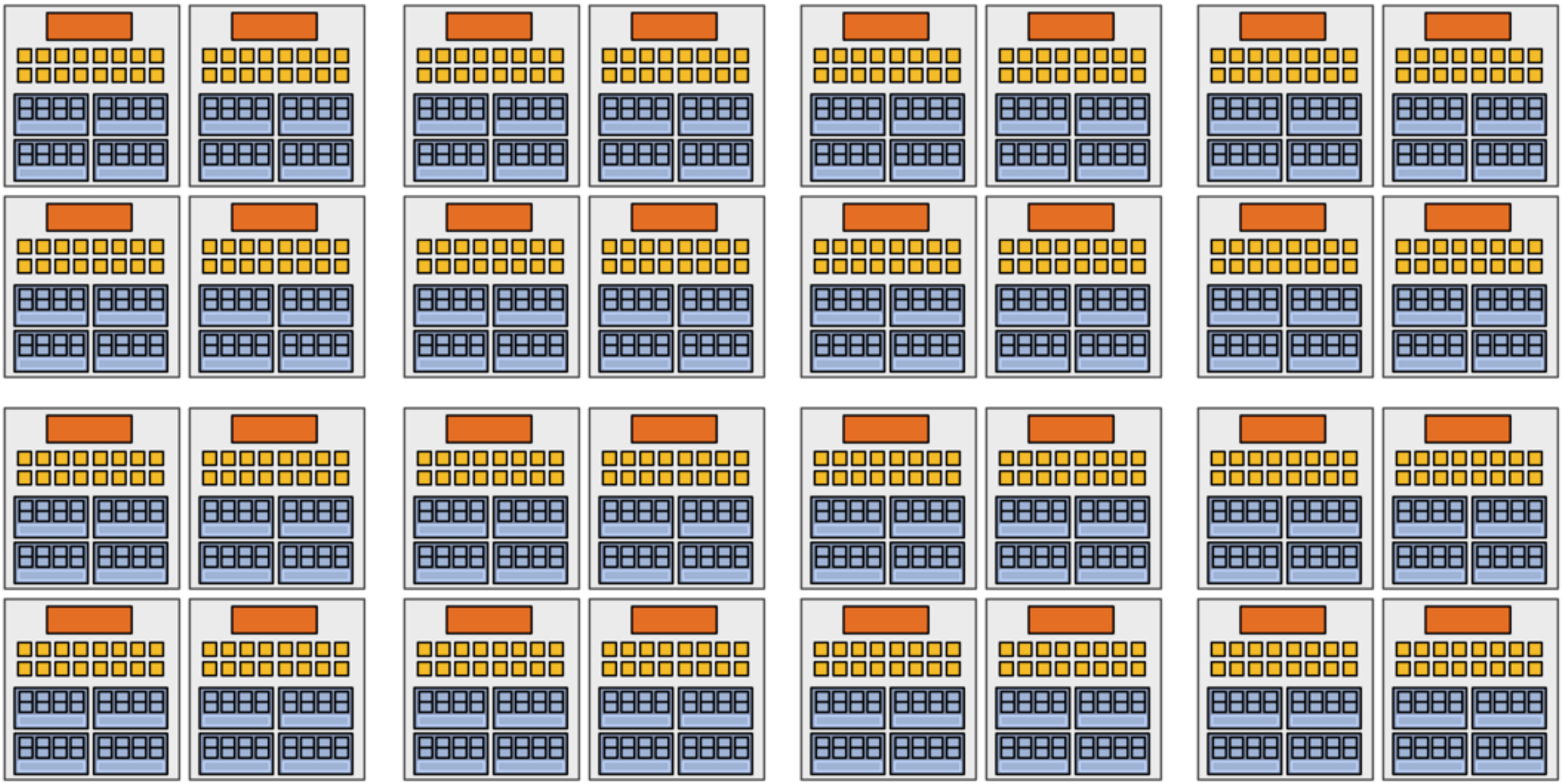
512 concurrent fragments

= 256 GFLOPs (@ 1GHz)



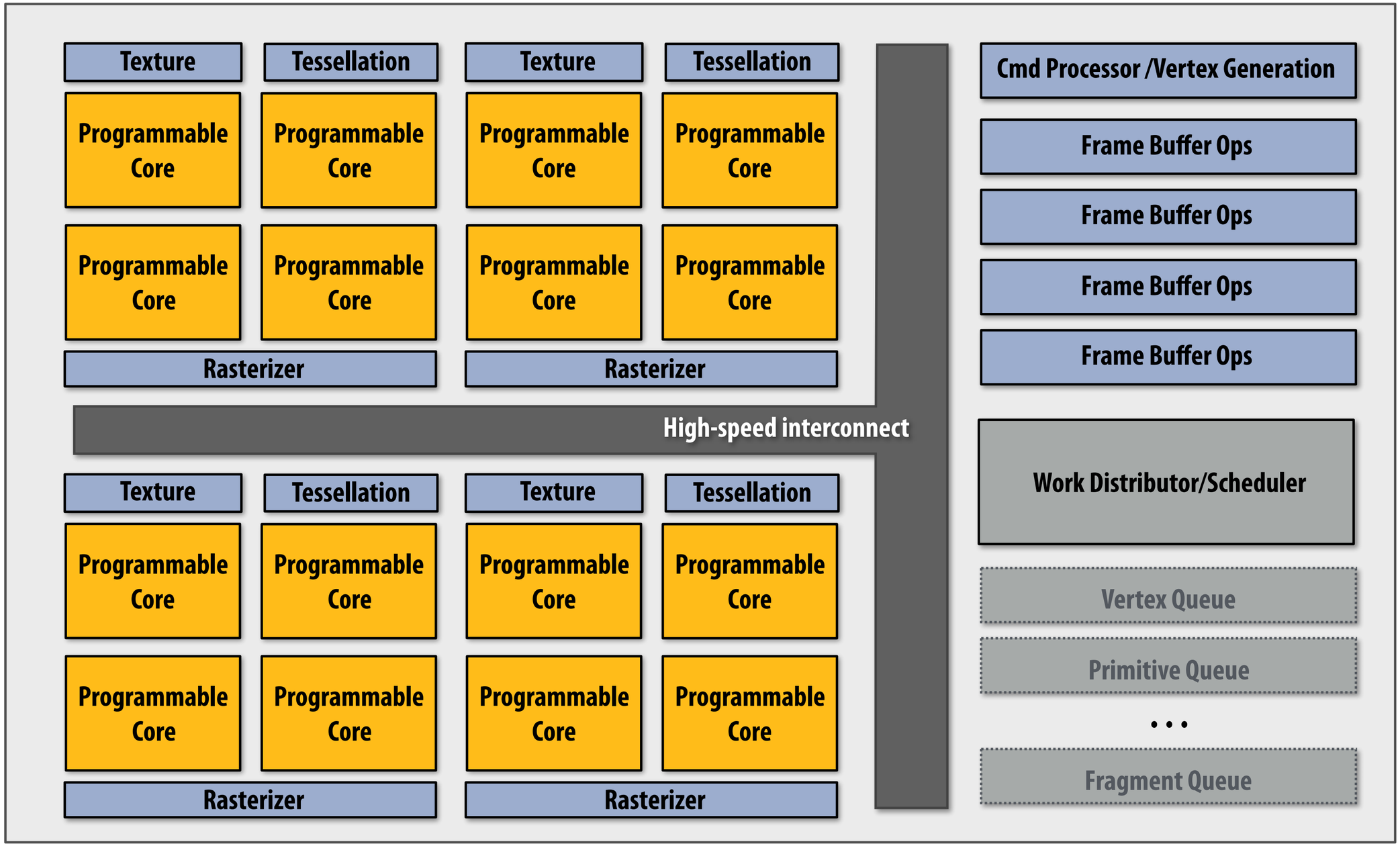


My “enthusiast” chip!



32 cores, 16 ALUs per core (512 total) = 1 TFLOP (@ 1 GHz)

Modern GPU: programmable parts of pipeline virtualized on pool of programmable cores



Hardware is a heterogeneous collection of resources (programmable and non-programmable)
Programmable resources are time-shared by vertex/primitive/fragment processing work
Hardware work distributor assigns work to cores (based on contents of inter-stage queues)
What is our communication strategy between functional units? Future lecture.

Summary: three key ideas

- Use many "slimmed down cores" to run in parallel

- 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

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

Real GPU Hardware

- 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

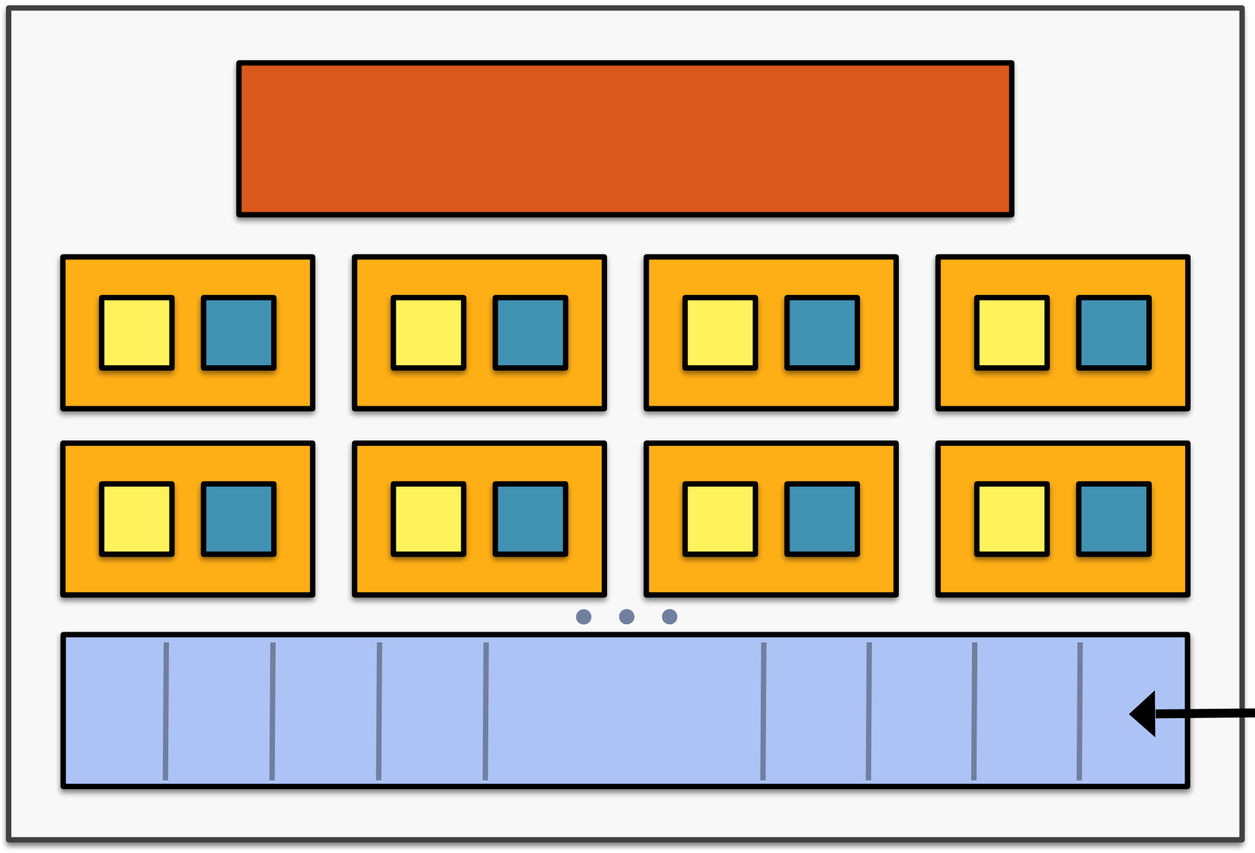
- Many factors play a role in actual chip performance

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

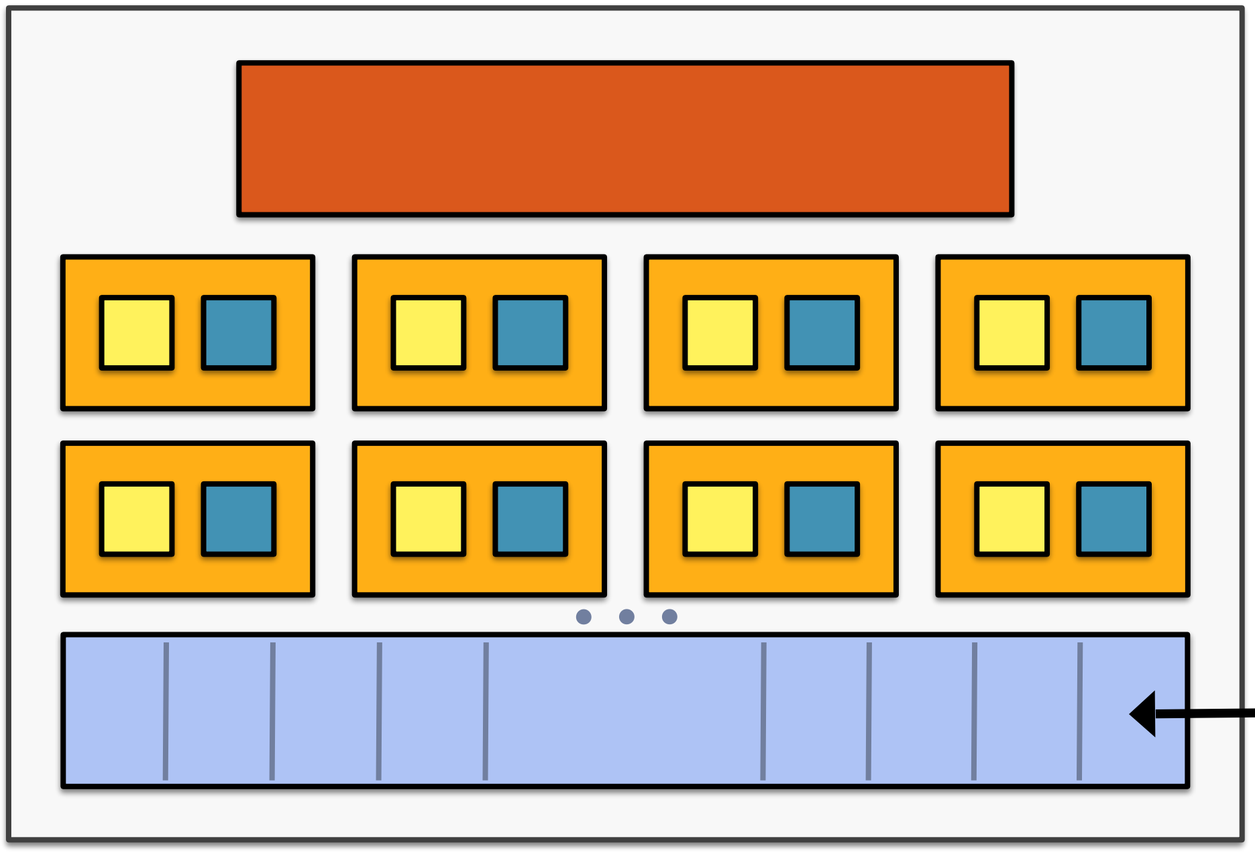
= multiply-add

= multiply

= instruction stream decode

= execution context storage

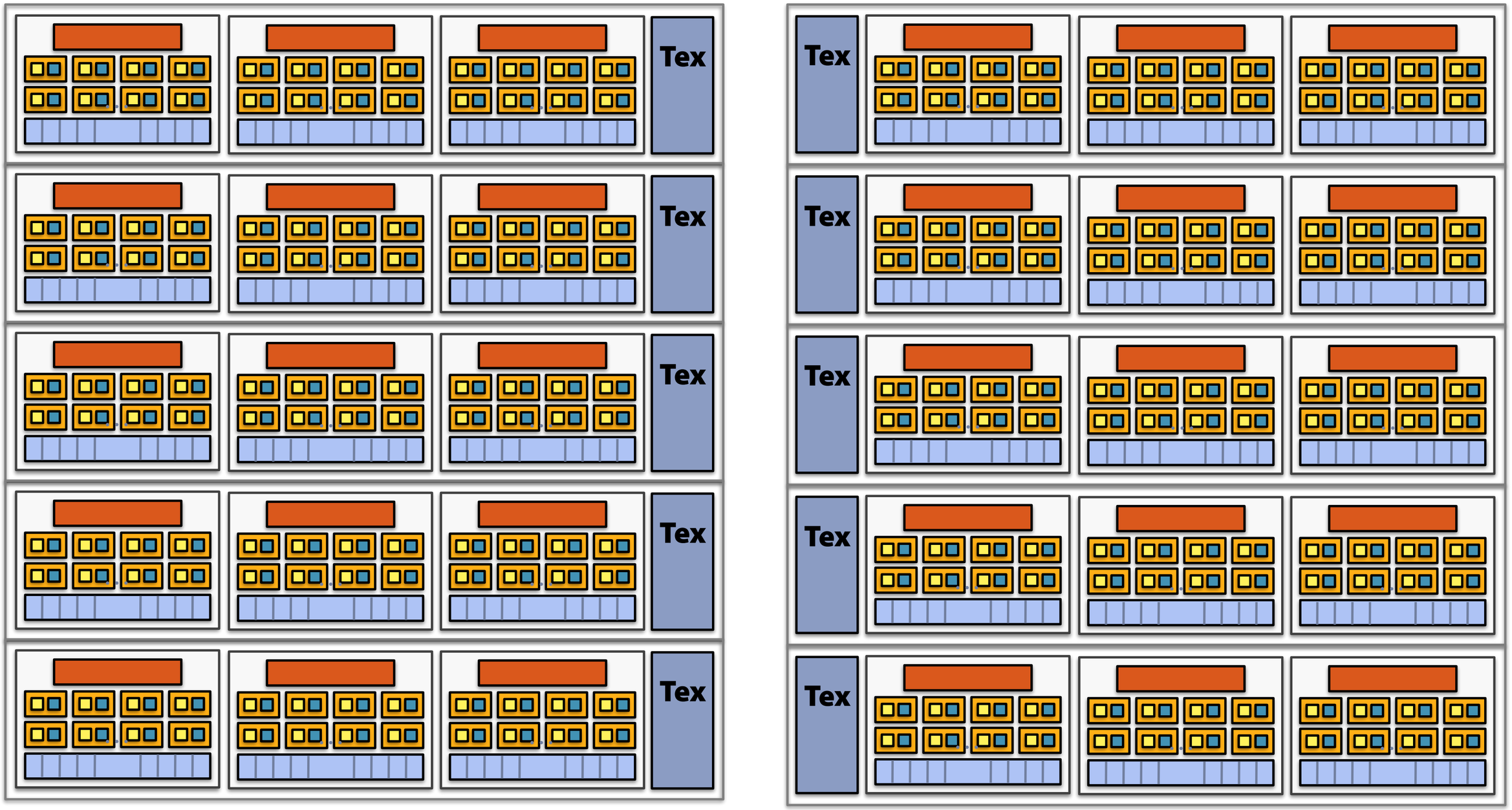
NVIDIA GeForce GTX 285 “core”



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



NVIDIA GeForce GTX 285

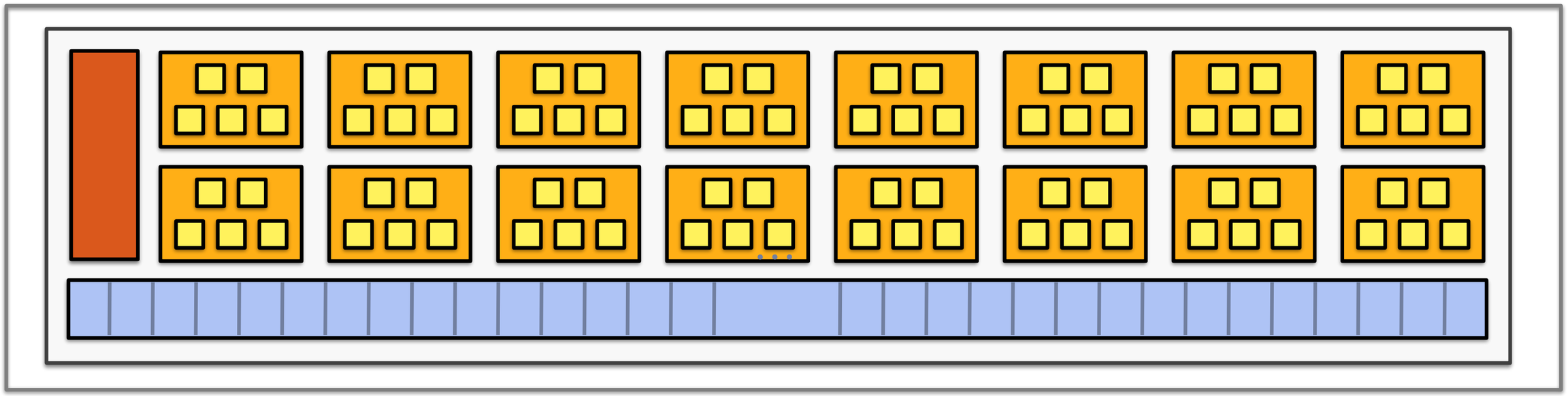
- Generic speak:
  - 30 processing cores
  - 8 SIMD functional units per core
  - Best case: 240 mul-adds + 240 muls per clock

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”



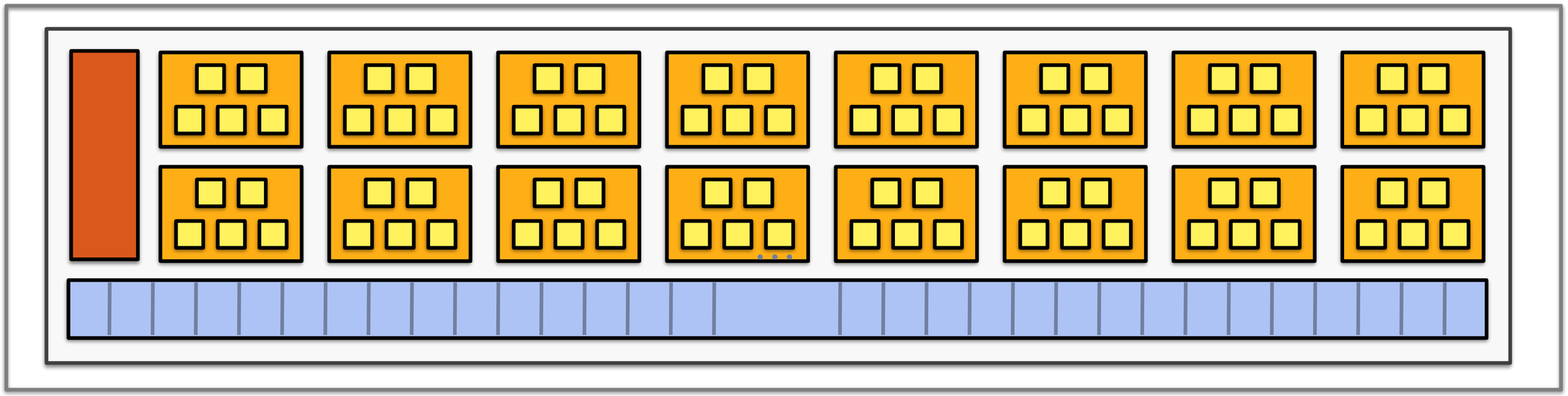
= SIMD functional unit, control shared across 16 units

= multiply-add

= instruction stream decode

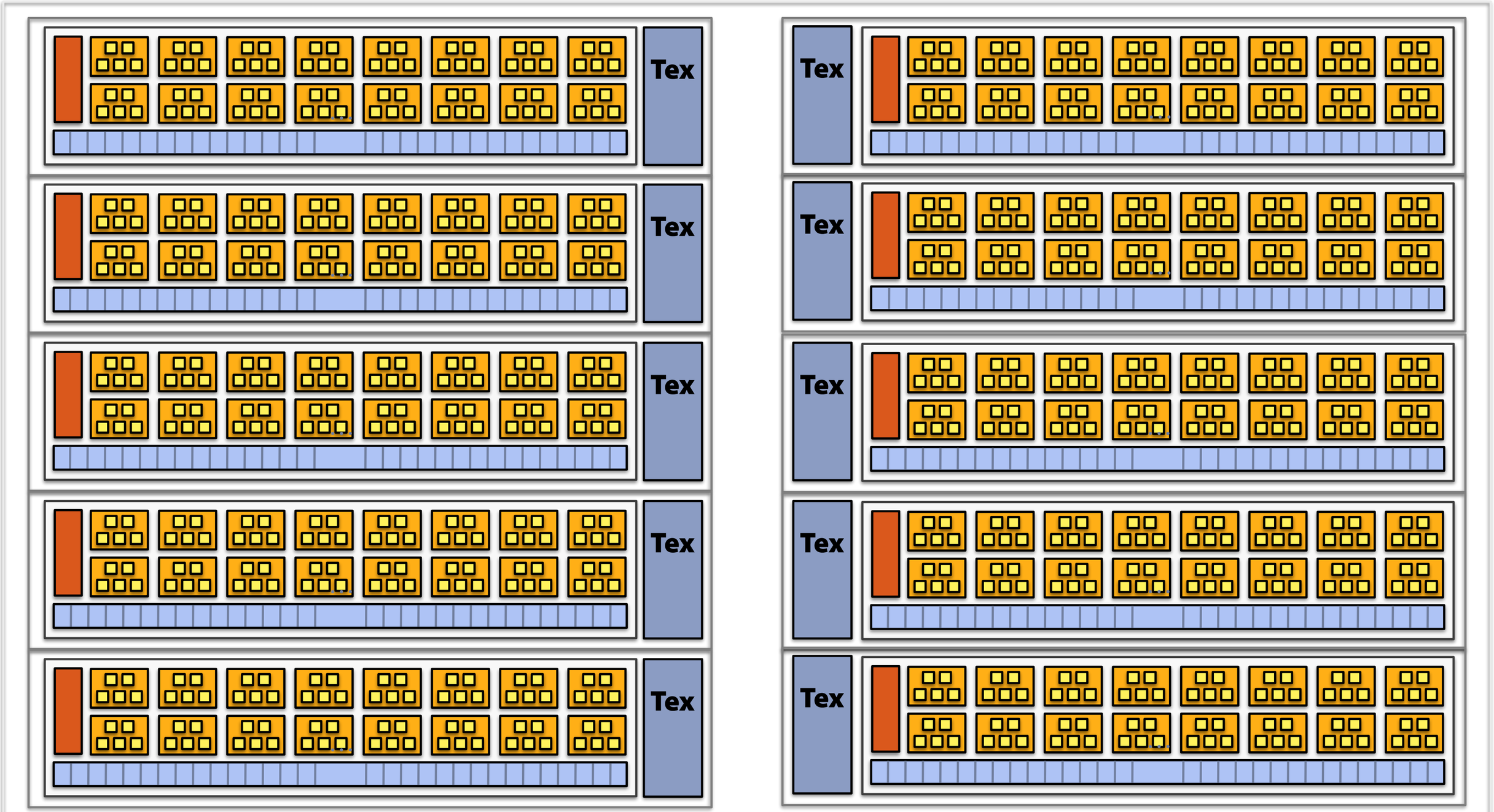
= execution context storage

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

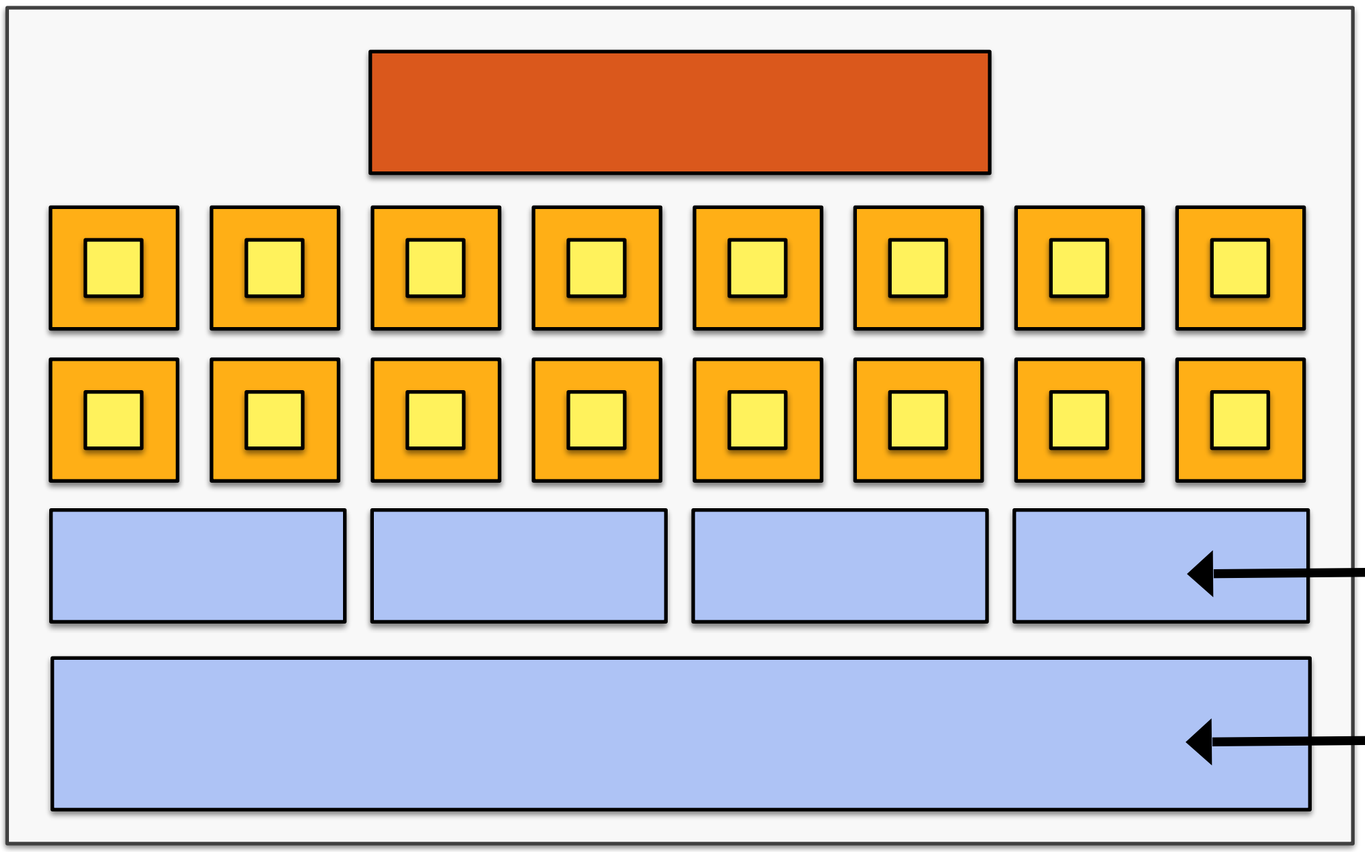
- Scale of interleaving similar to NVIDIA GPUs

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

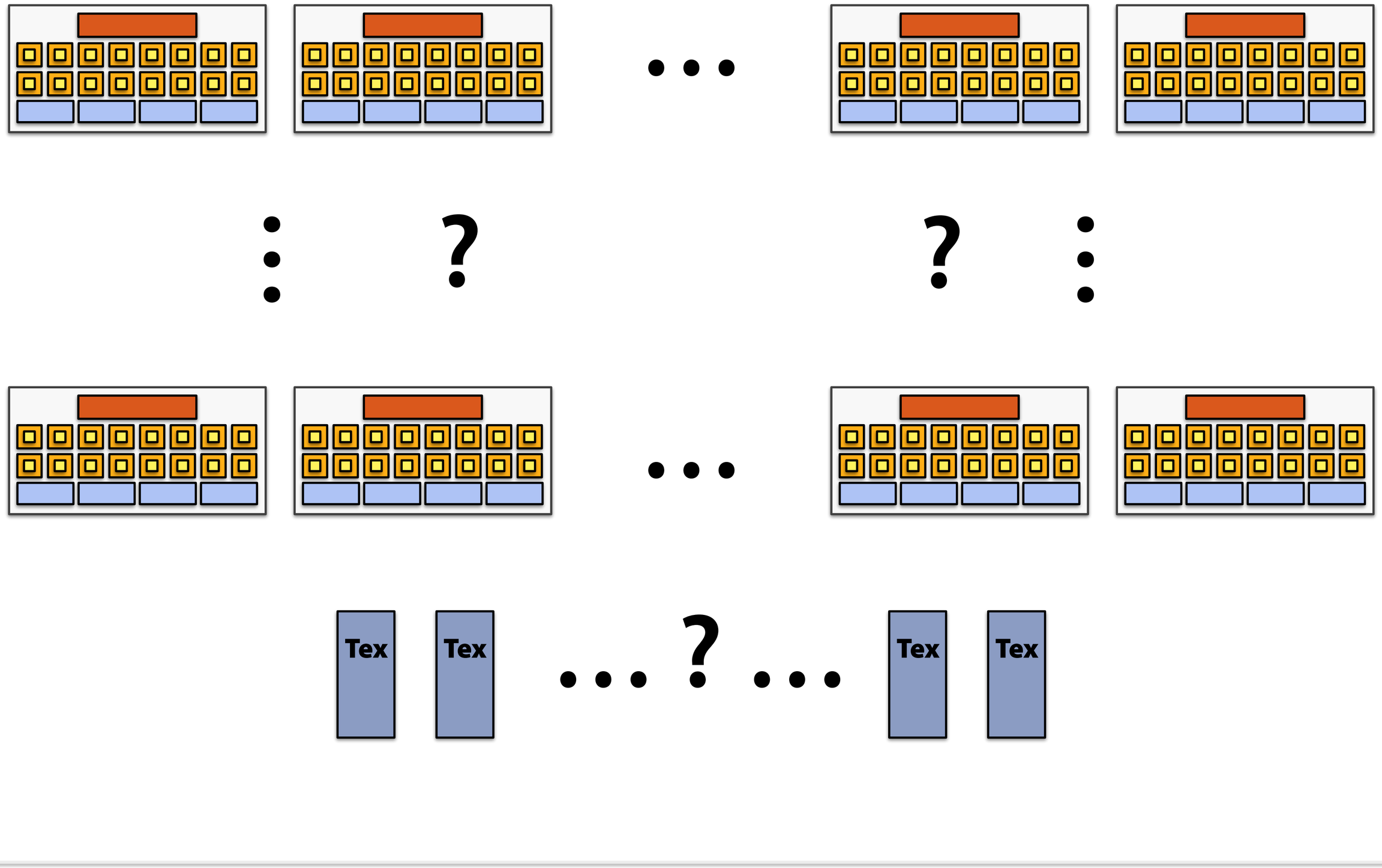
= SIMD functional unit, control shared across 16 units

= mul-add

= instruction stream decode

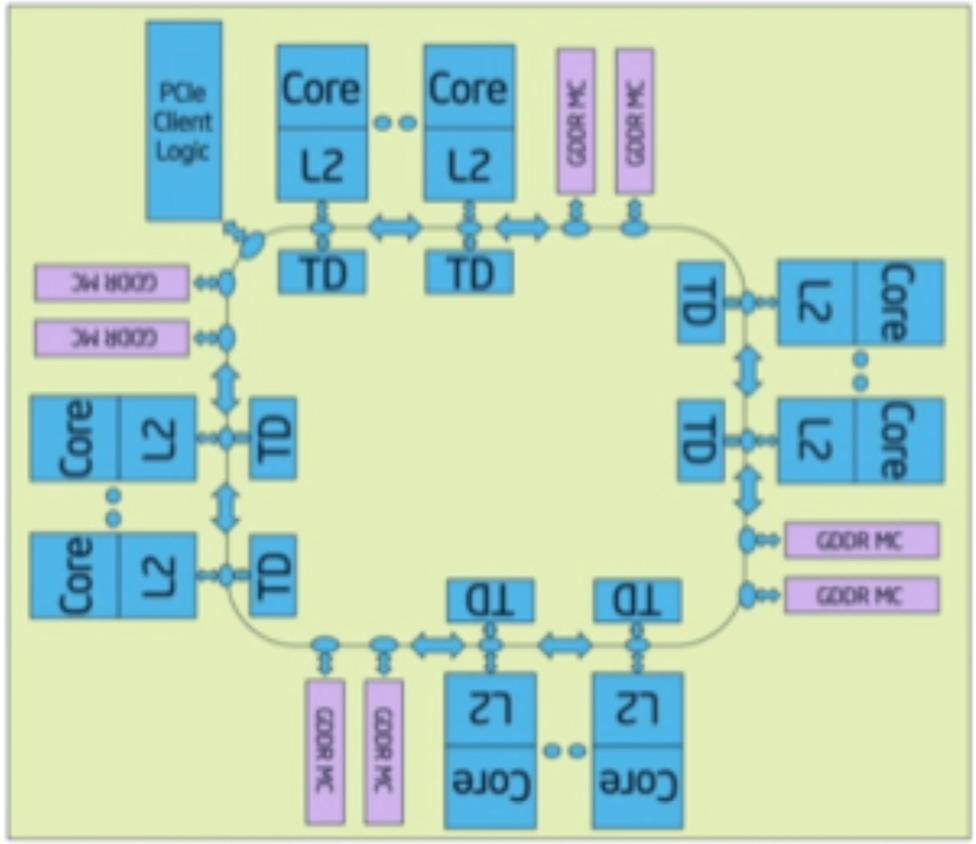
= execution context storage/ HW registers

Intel Larrabee



Intel Xeon Phi

- 60 cores
- Each core: 512b-wide (16 x 32b) SSE-style vector unit, plus scalar
  - 2147 GFLOPS SP, 1074 DP
  - Compare to NVIDIA K20X: 3950/1310
- 8 GB DRAM, 352 GB/s theoretical peak, 200 GB/s achievable

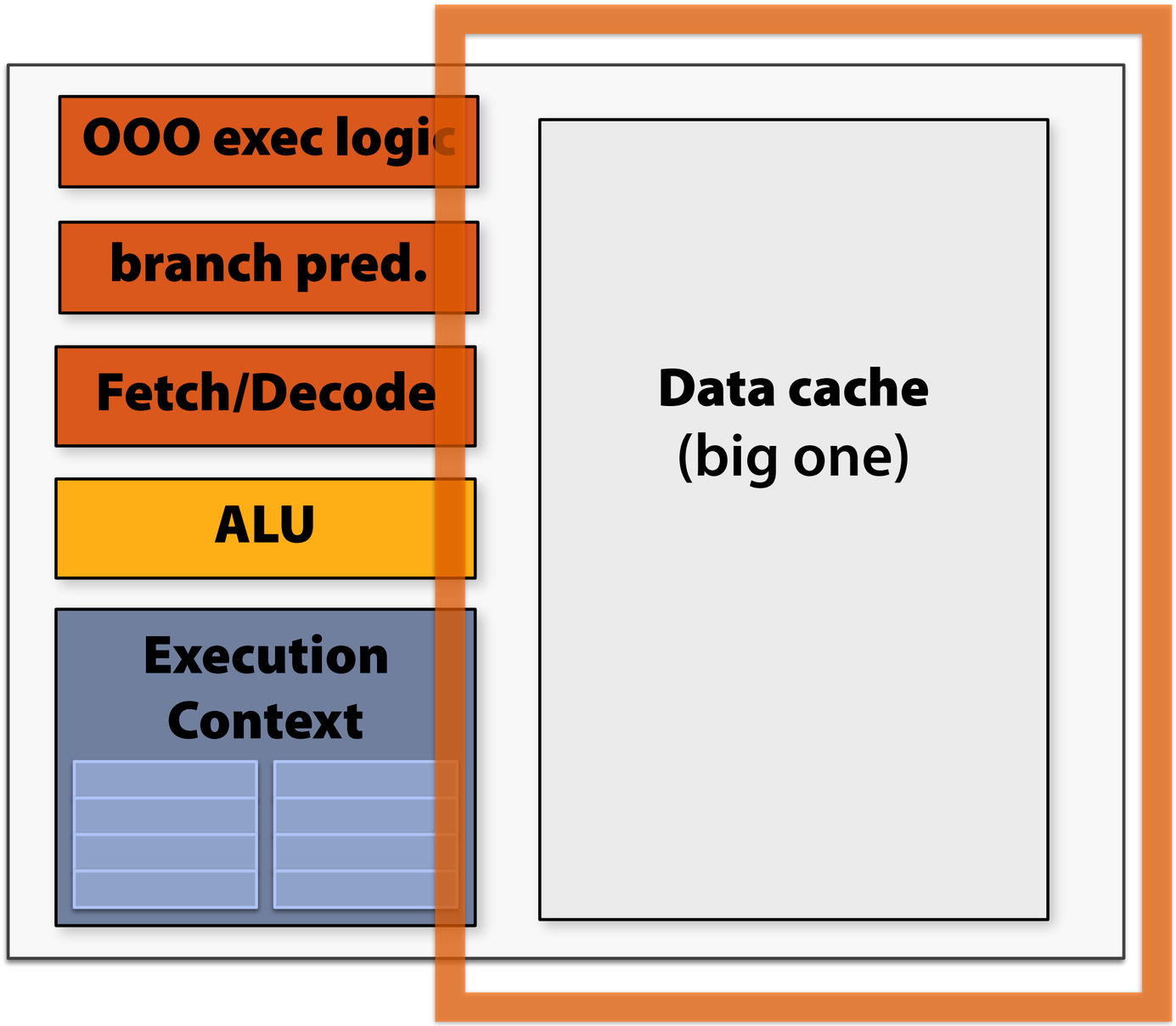


http://www.drdobbs.com/parallel/programming-intels-xeon-phi-a-jumpstart/240144160

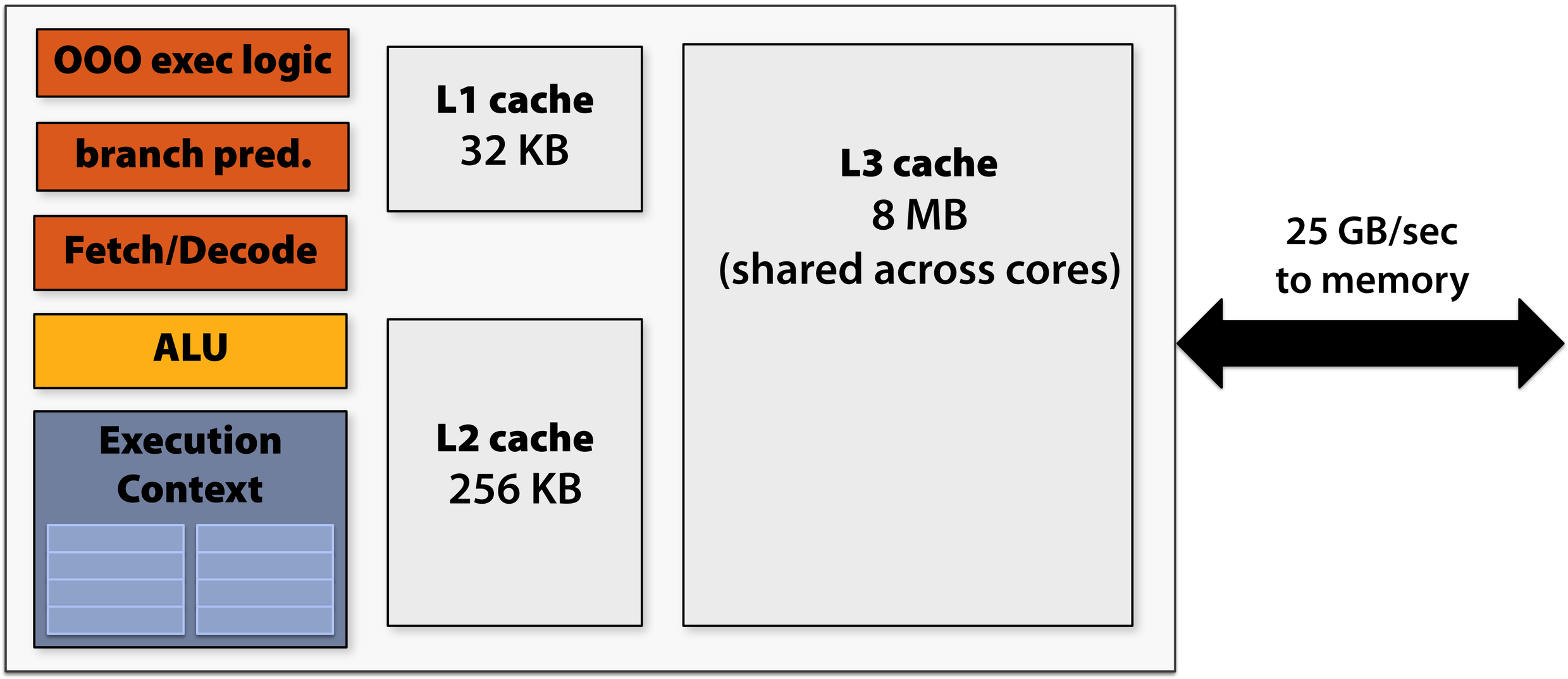
The talk thus far: processing data

- Part 3: moving data to processors

Recall: CPU-"style" core

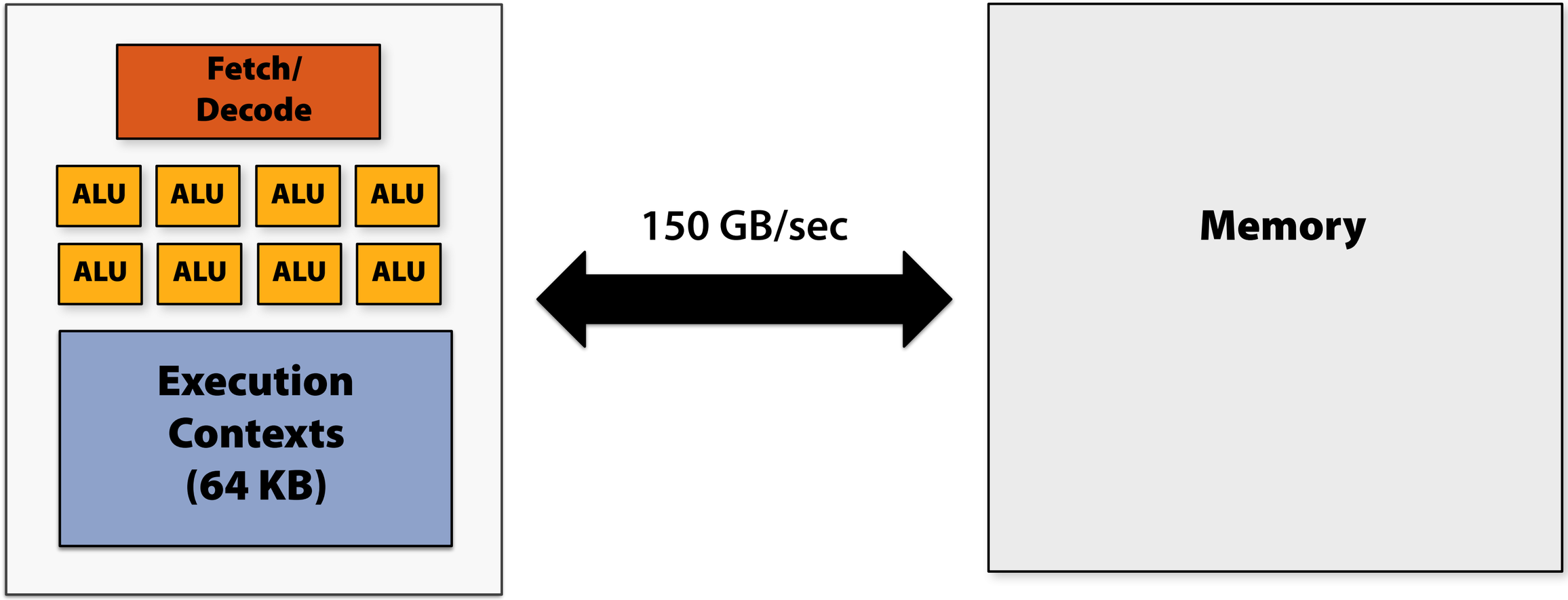


CPU-"style" memory hierarchy



- 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:
  - ~6x 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, fastest DRAM in 2017: 512 GB/s, with faster stacked memory)
  - 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.

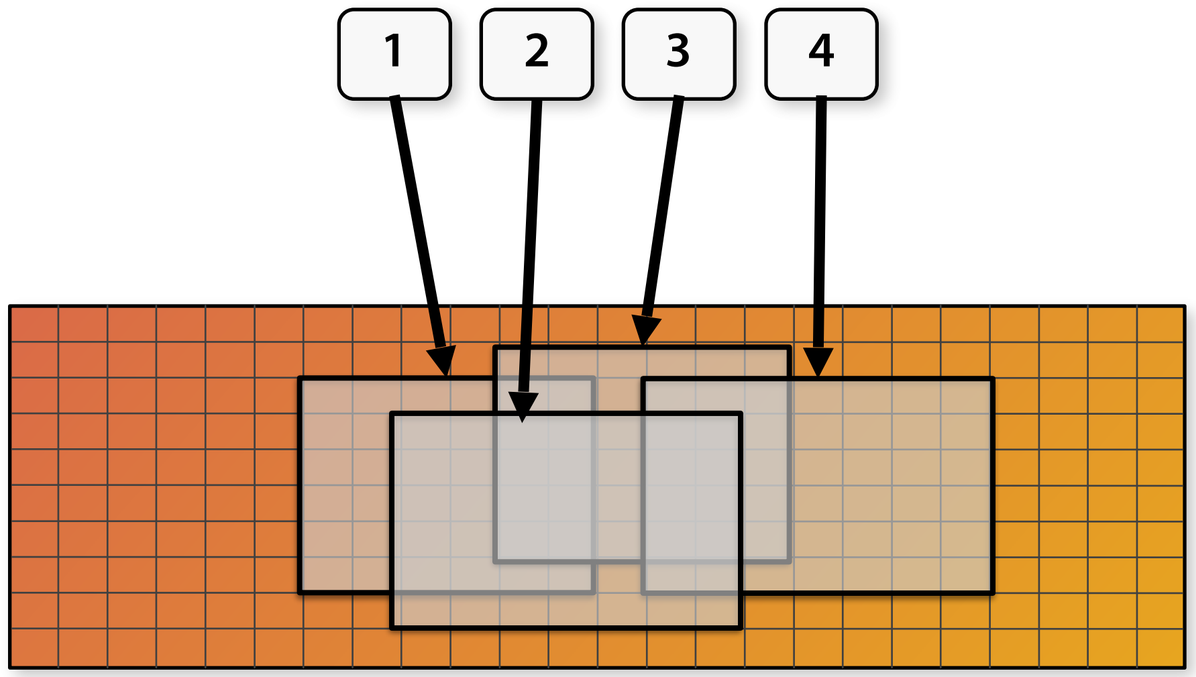
Reducing required bandwidth

- Request data less often
  - (do more math)

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

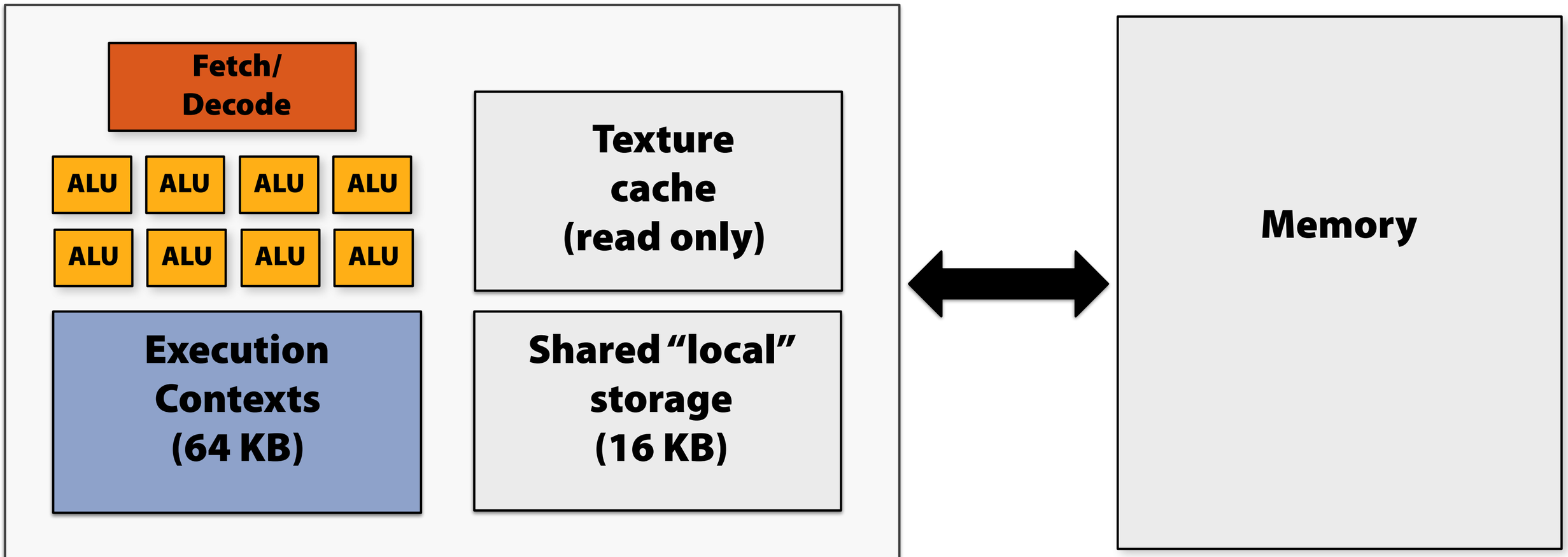
Reducing required bandwidth

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



GPU memory hierarchy

- On-chip storage takes load off memory system
- Many developers calling for larger, more cache-like storage (and most recent GPUs now have it)



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 independent 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