adapted from

Graphics and Imaging Architectures

Kayvon Fatahalian

http://www.cs.cmu.edu/afs/cs/academic/class/15869-f11/www/

also usefull as INTRODUCTION TO GPUs AND GPGPU

COMPUTING IS BECOMING HIGHLY VISUAL!

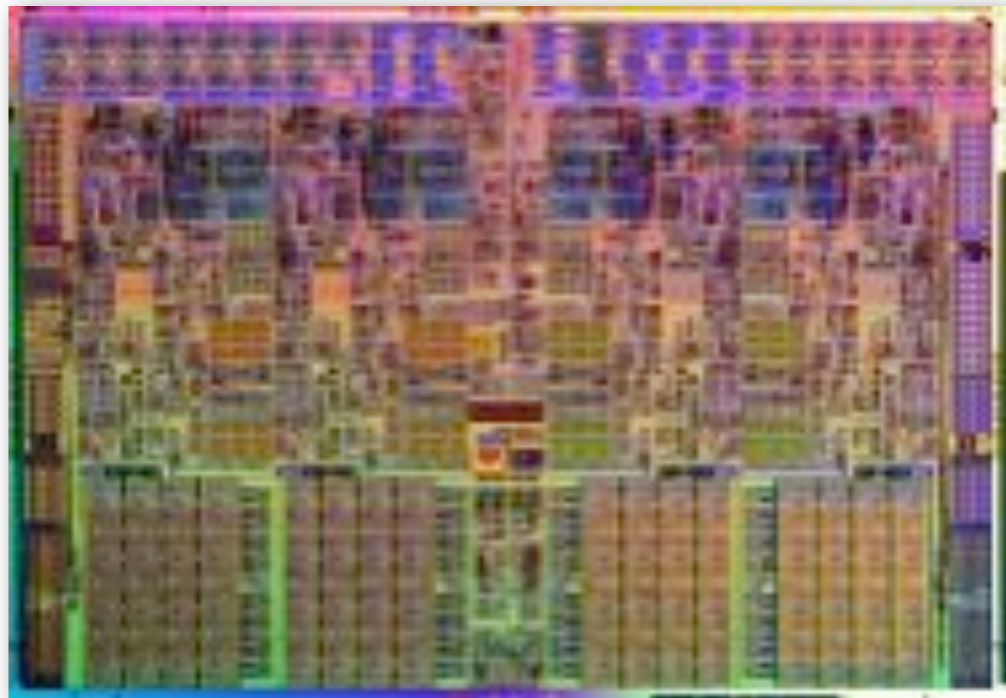
Visually rich user interfaces
(that require 3D graphics!)

Innovative new input modalities
(touch, gyro, cameras)

Ubiquitous, high-resolution cameras
(emergence of computational photography)

Games, entertainment
(continuous push towards higher visual realism)

COMPUTING EFFICIENTLY IS INCREASINGLY CRITICAL!


Core i7 (Nahalem)

Ubiquitous parallelism

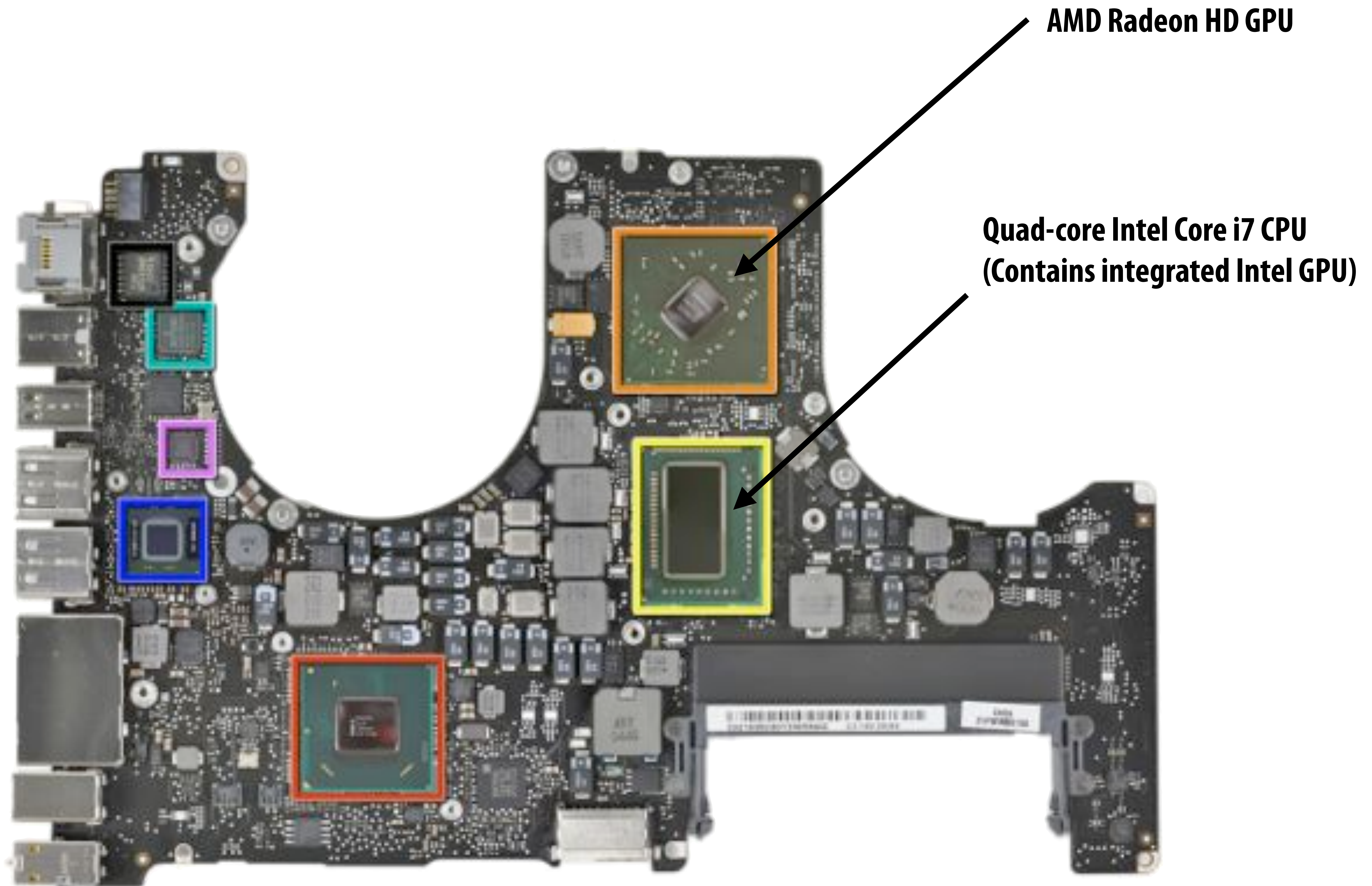
Heterogeneous parallelism

Power constraints

Mobile, mobile, mobile

Real-time 3D graphics has always found a way to consume all available compute: GPUs are efficient parallel, heterogeneous systems (that are relatively easy to program)

My Macbook Pro 2011 (two GPUs)

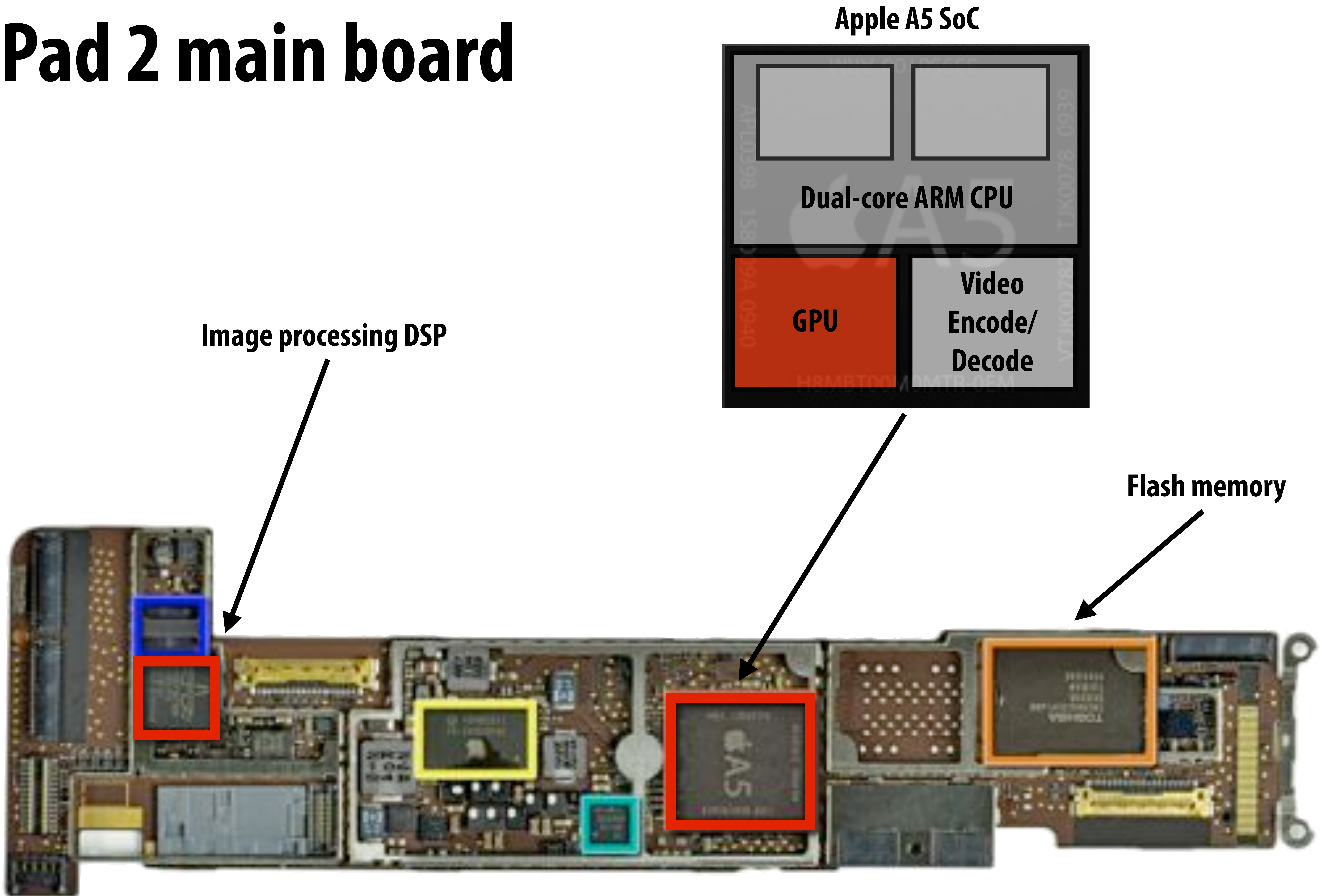


AMD Radeon HD GPU

Quad-core Intel Core i7 CPU
(Contains integrated Intel GPU)

From ifixit.com teardown

iPad 2 main board



From ifixit.com teardown

Touchscreen controller (integrated DSP) ~ $1

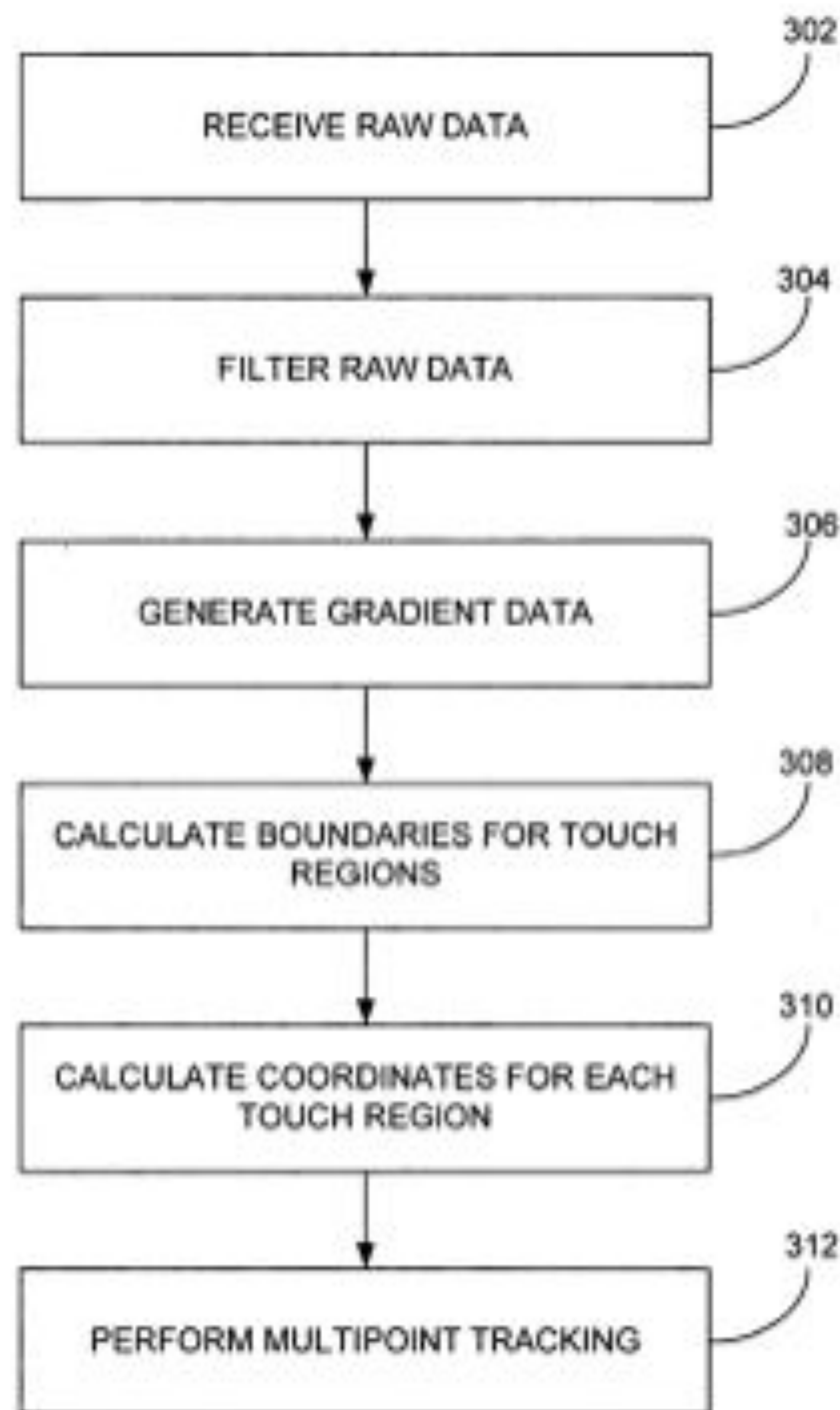


FIG. 16

RAW DATA INCLUDING NOISE

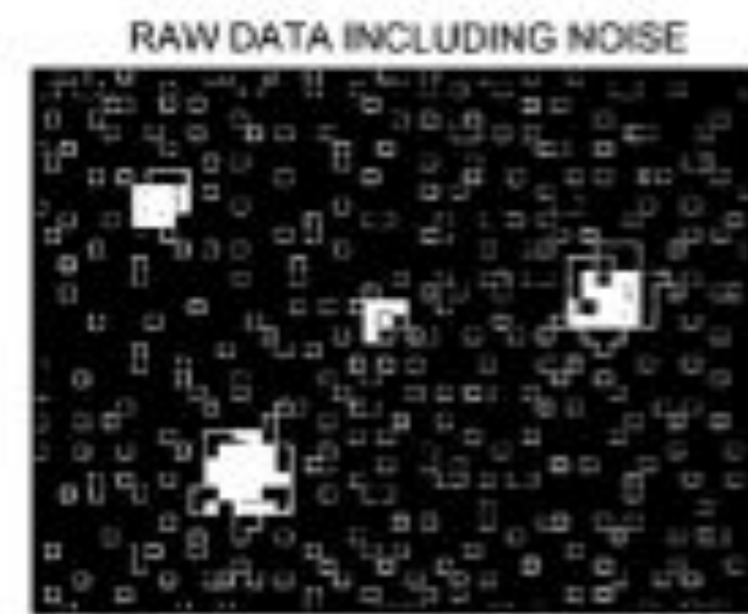


FIG. 17A

FILTERED DATA

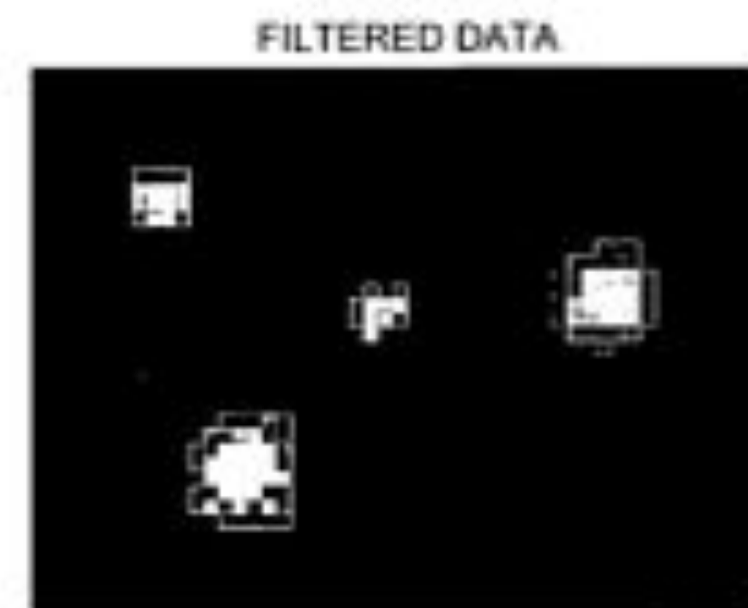


FIG. 17B

GRADIENT DATA

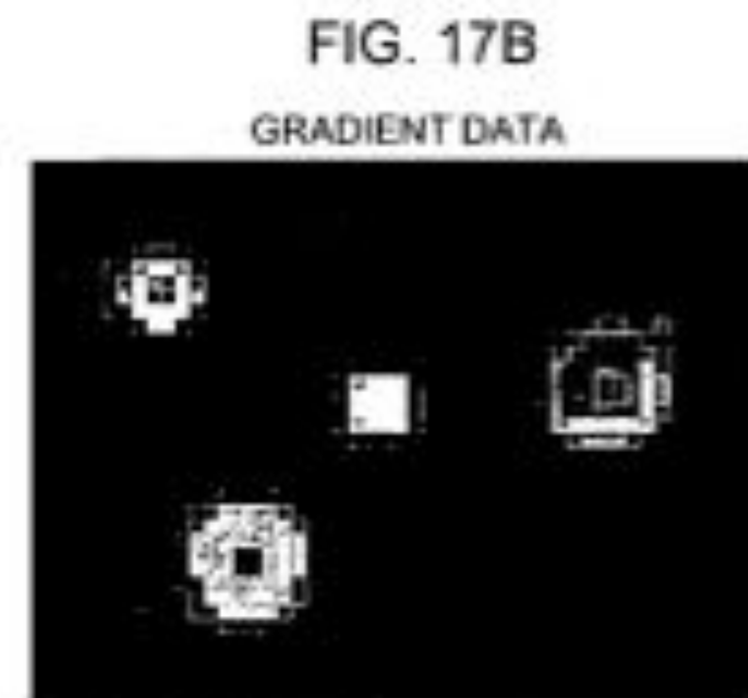


FIG. 17C

TOUCH REGIONS

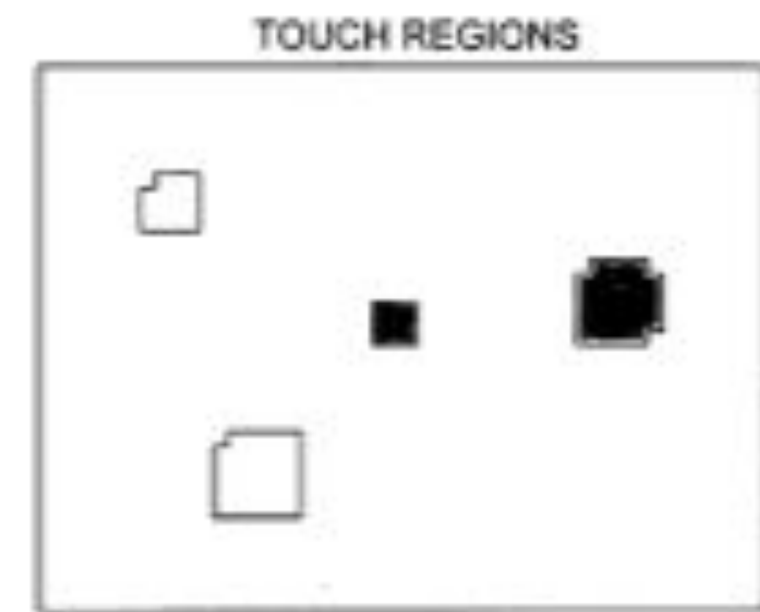


FIG. 17D

COORDINATES OF TOUCH REGIONS

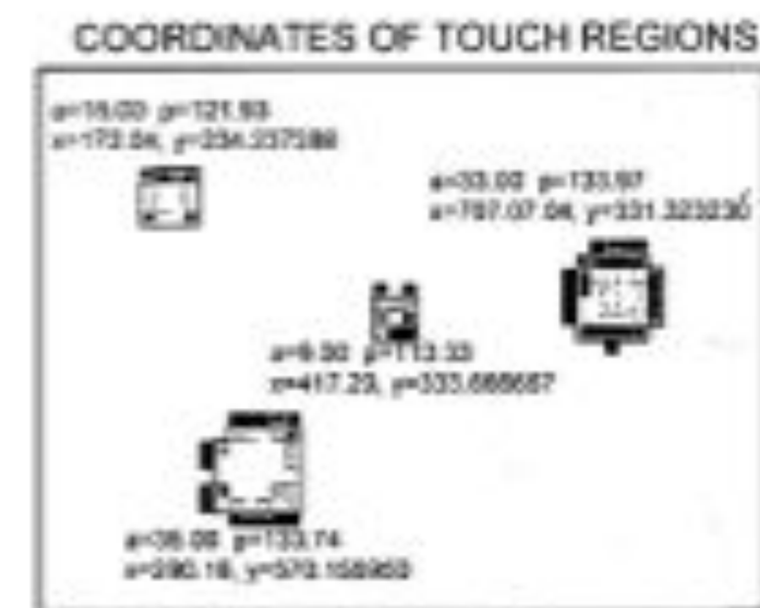


FIG. 17E

From US Patent Application 2006/0097991

My Nikon D7000 camera

16.2 MP sensor

14 bits per pixel

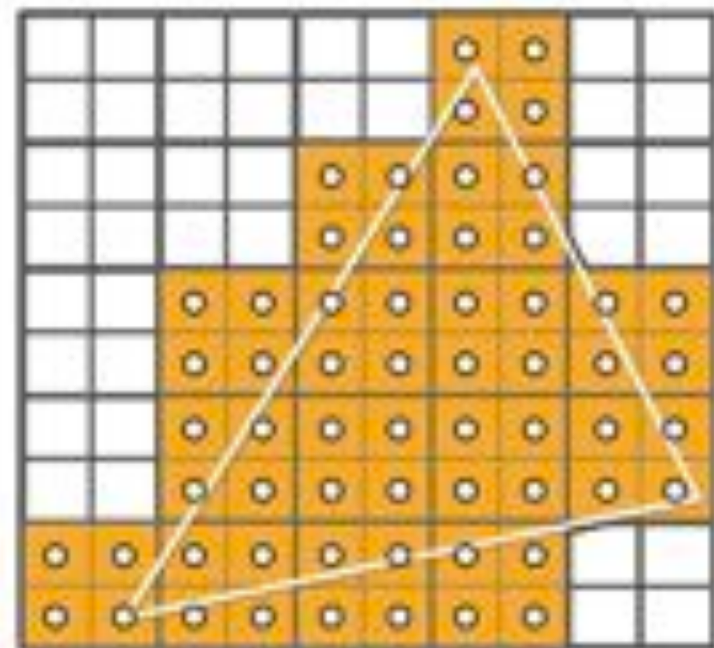
6 fps burst

What this course is

This is a course about how real-time graphics systems work

GRAPHICS ALGORITHMS

(the workload)



MACHINE ORGANIZATION

geometry processing
rasterization
texture mapping
anti-aliasing

Parallelism
Locality
Communication
The design of throughput processing cores
The role of fixed-function HW

What this course is

This is a course about how real-time graphics systems work

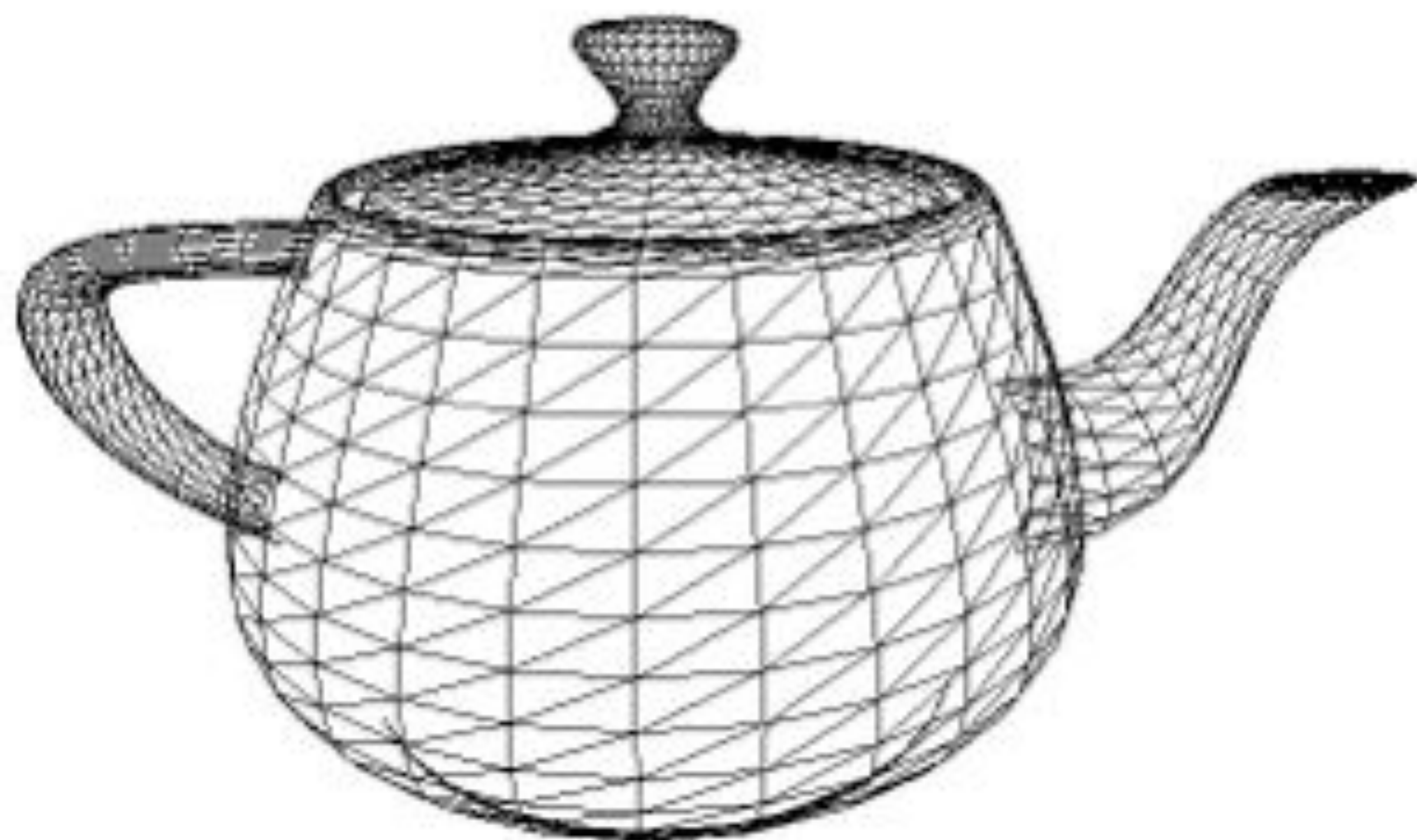
GRAPHICS ALGORITHMS
(the workload)

MACHINE ORGANIZATION

ABSTRACTIONS
(the real-time graphics pipeline)

choice of primitives
level of abstraction

3D rendering



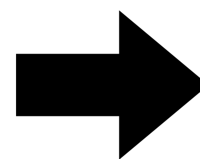


Image credit: Henrik Wann Jensen

Model of a scene:

3D surface geometry (e.g., triangle mesh)
surface materials
lights
camera

Image

How does each triangle contribute to each pixel in the image?

Real-time graphics pipeline (entities)

1
3
4
2

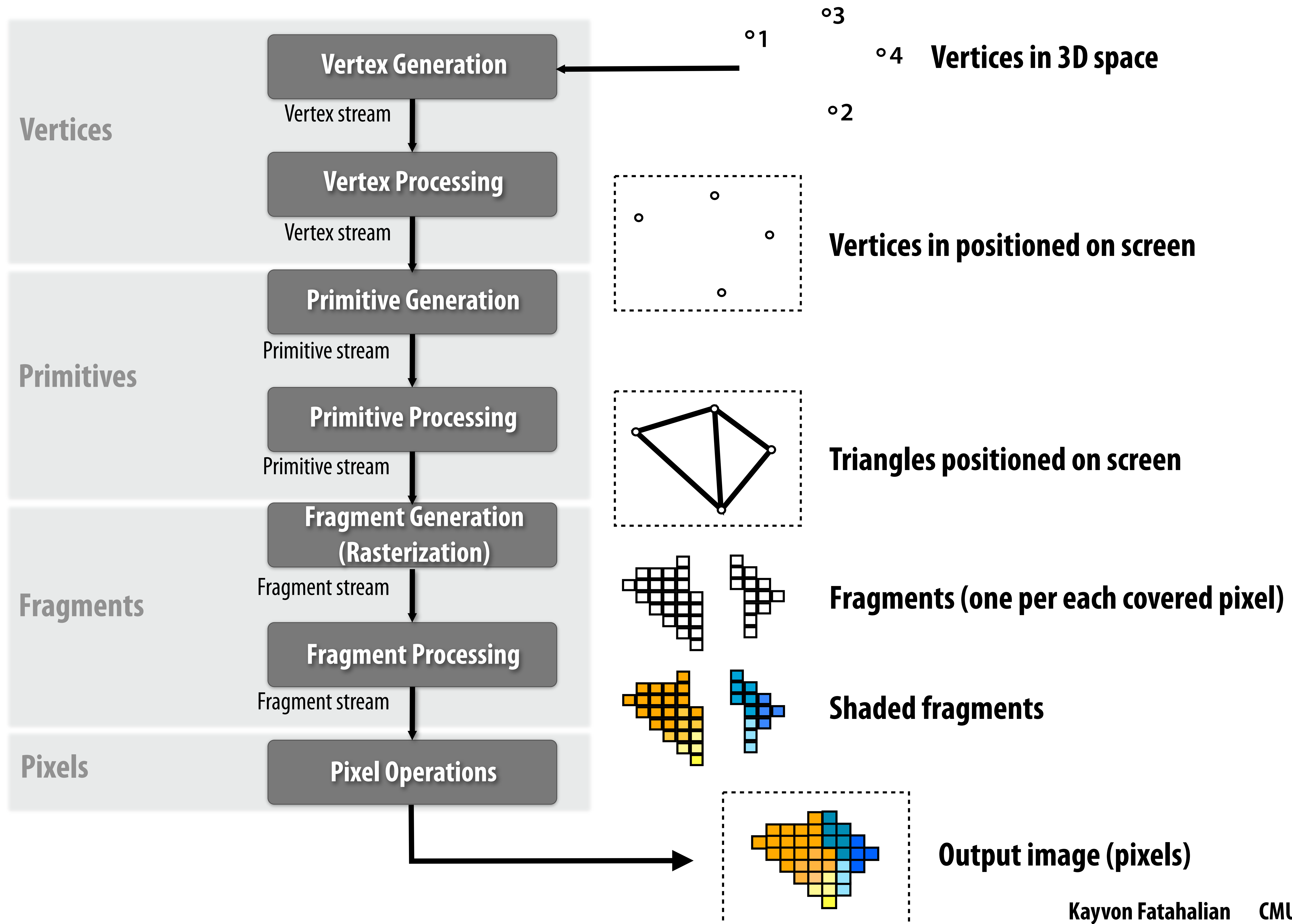
Vertices

Primitives
(triangles, points, lines)

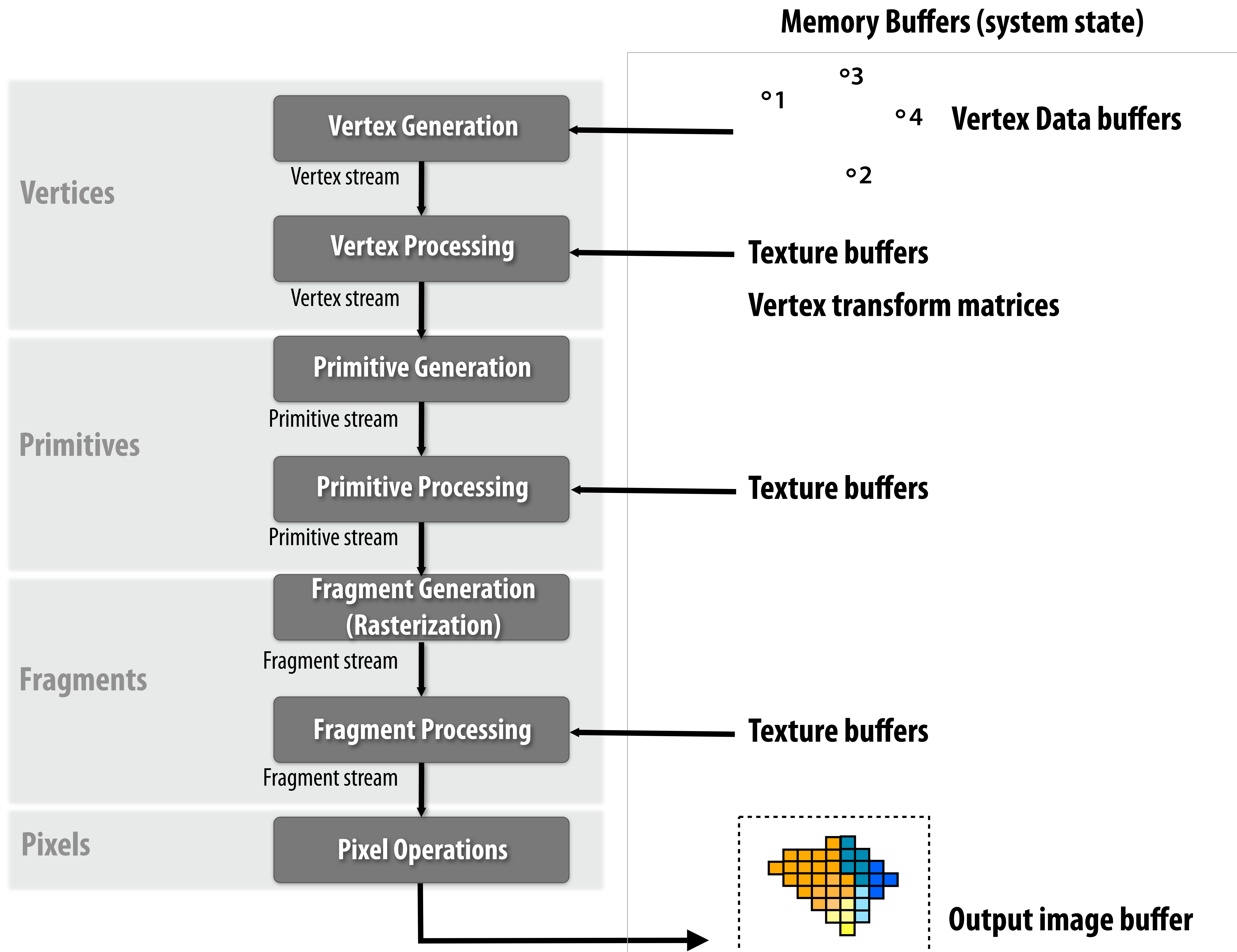
Fragments

Pixels

Real-time graphics pipeline (operations)



Real-time graphics pipeline (state)



Current generation

- **Some GPU computations now driven by alternative programming interface (not 3D graphics pipeline)**
  - to augment 3D rendering (game physics, global lighting)
  - for non-graphics applications (scientific computing)

Image credit: NVIDIA

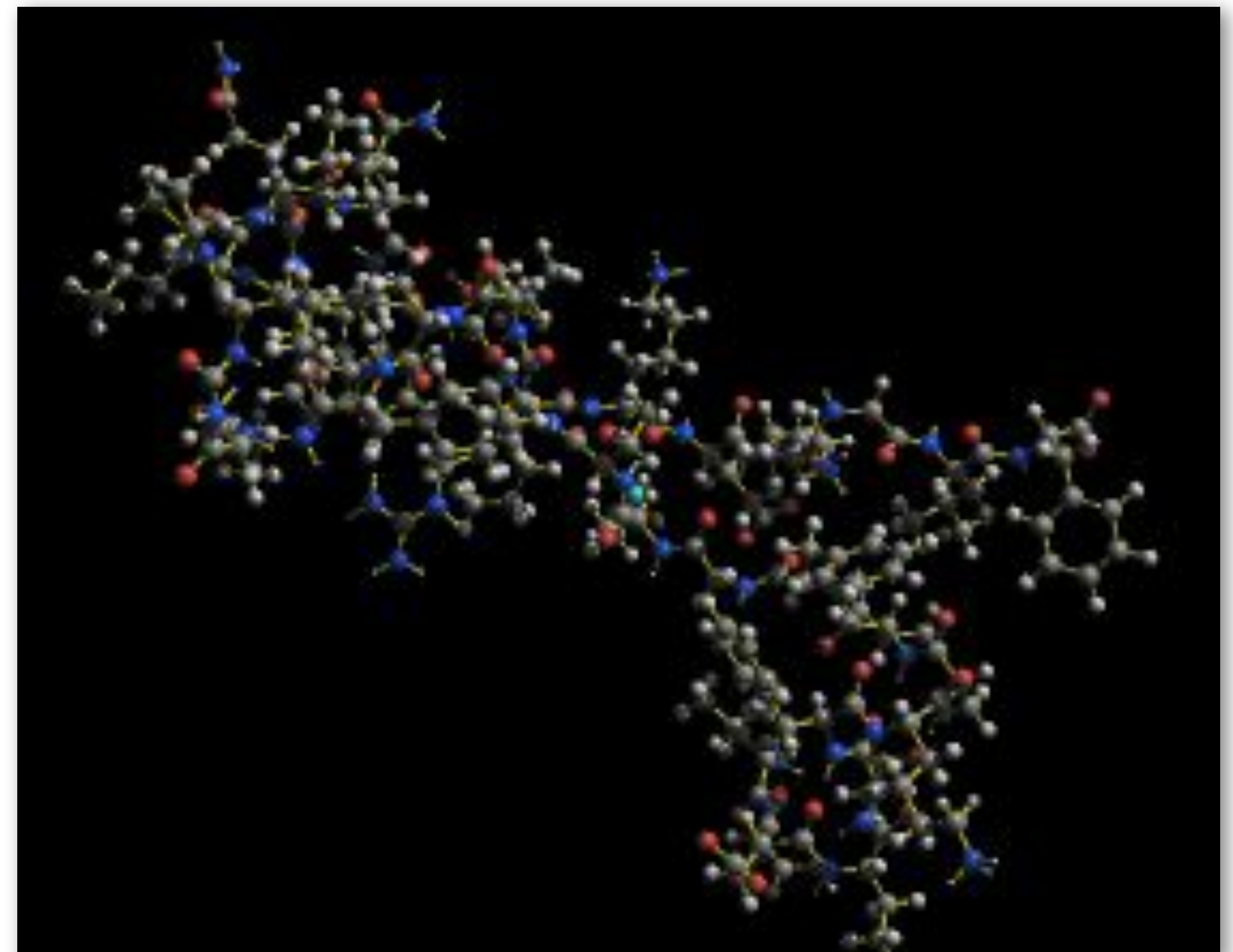


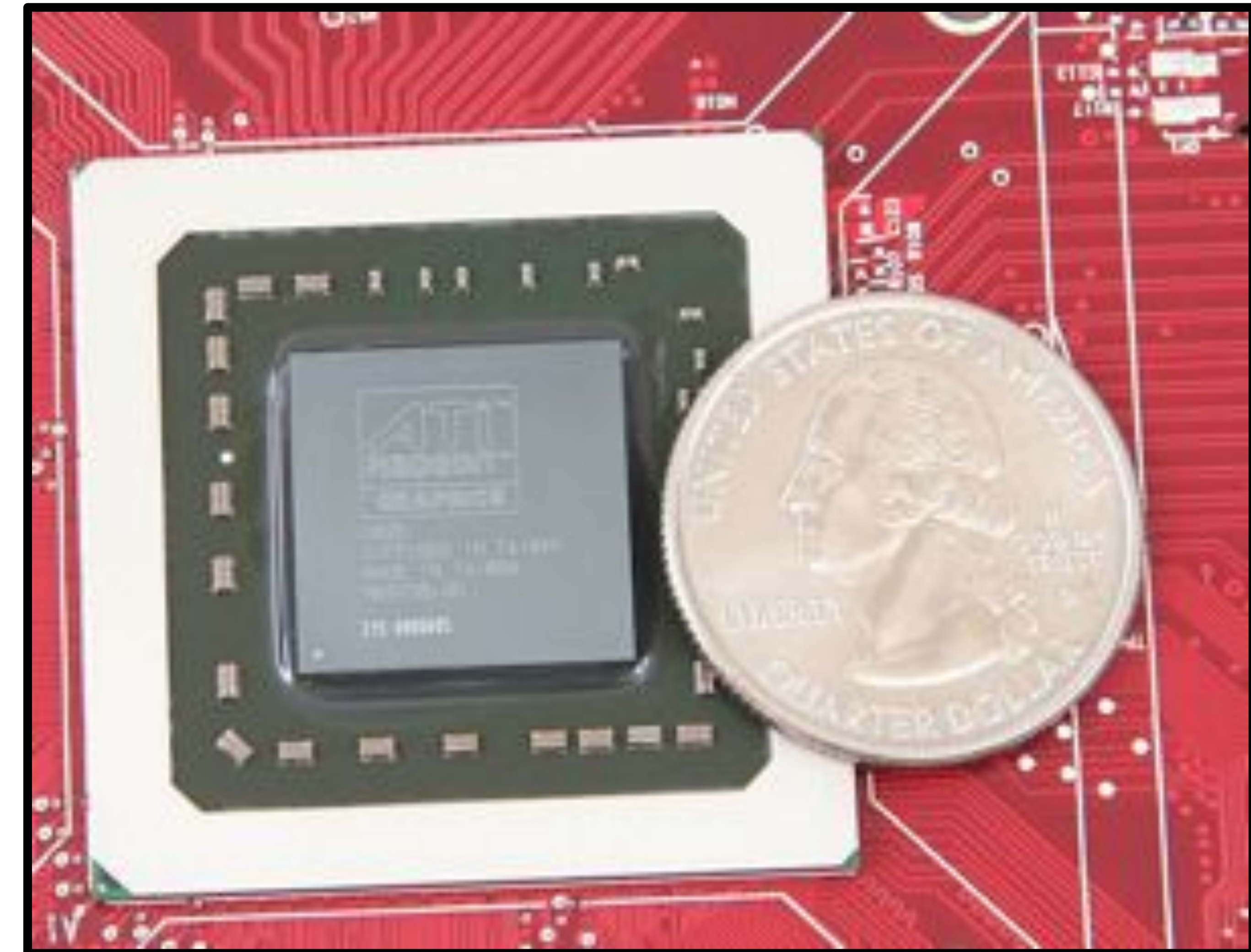
Image credit: Folding @ Home

Today: high versatility, high peak compute

Intel Core i7 (quad-core CPU)

~100 GFLOPS

730 million transistors



AMD Radeon HD 5870 GPU

~2.7 TFLOPS

2.2 billion transistors

More compute power than Pixar's entire render farm for Toy Story 1!

adapted from

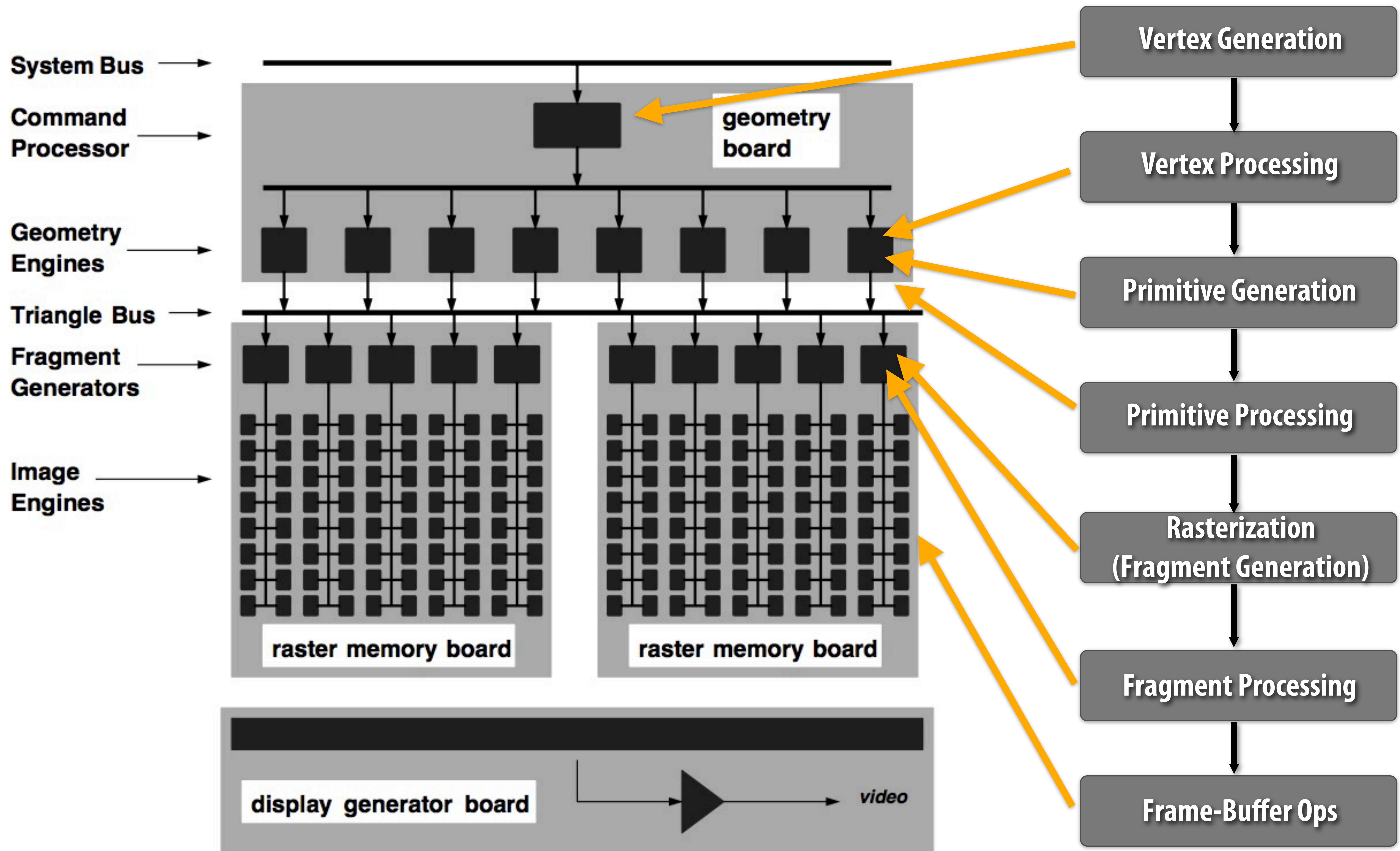
Lecture 7: The Programmable GPU Core

Kayvon Fatahalian

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

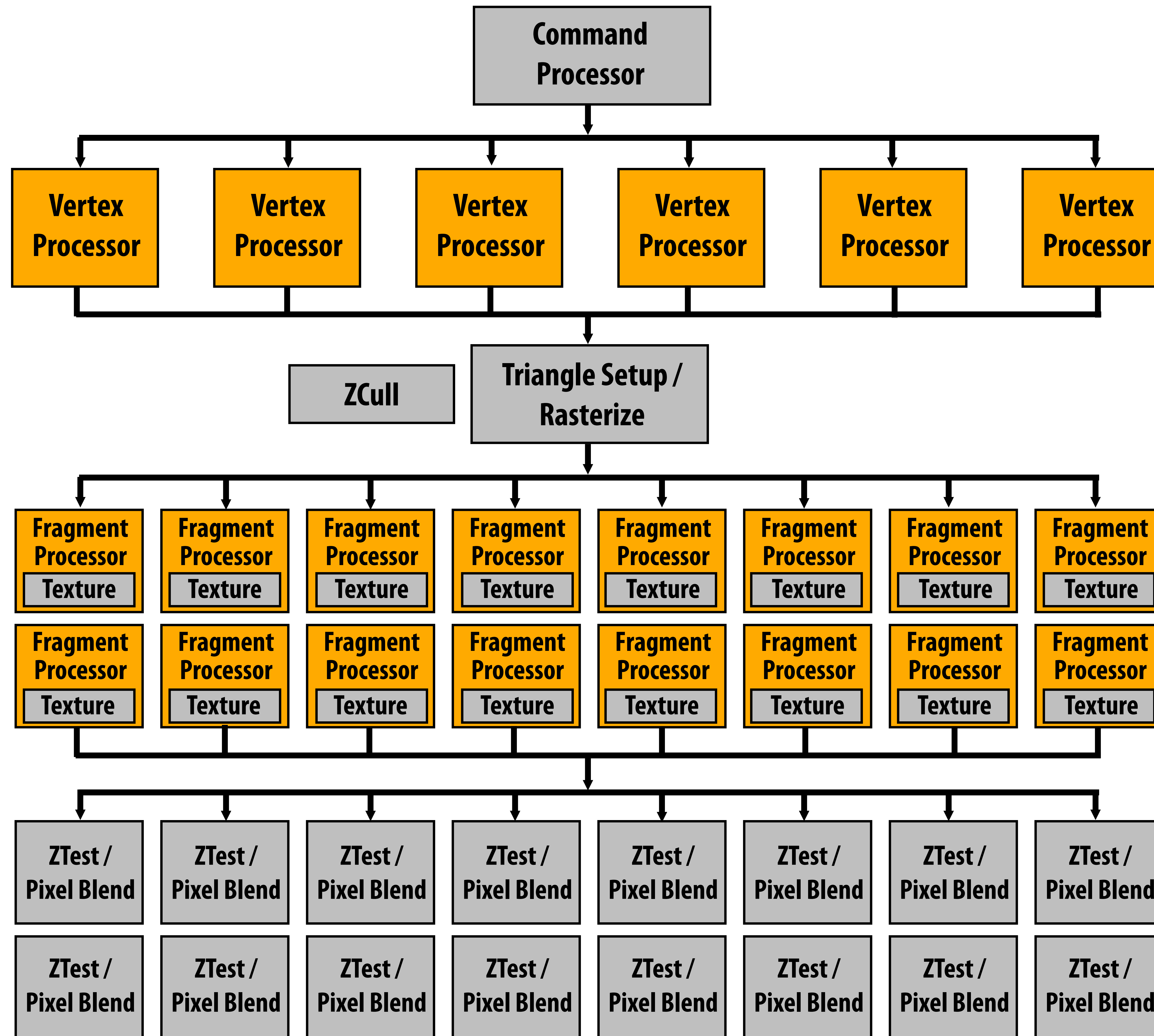
- **A brief history of GPU programmability**
- **Throughput processing core 101**
- **A detailed look at recent GPU designs**

SGI RealityEngine (1993)

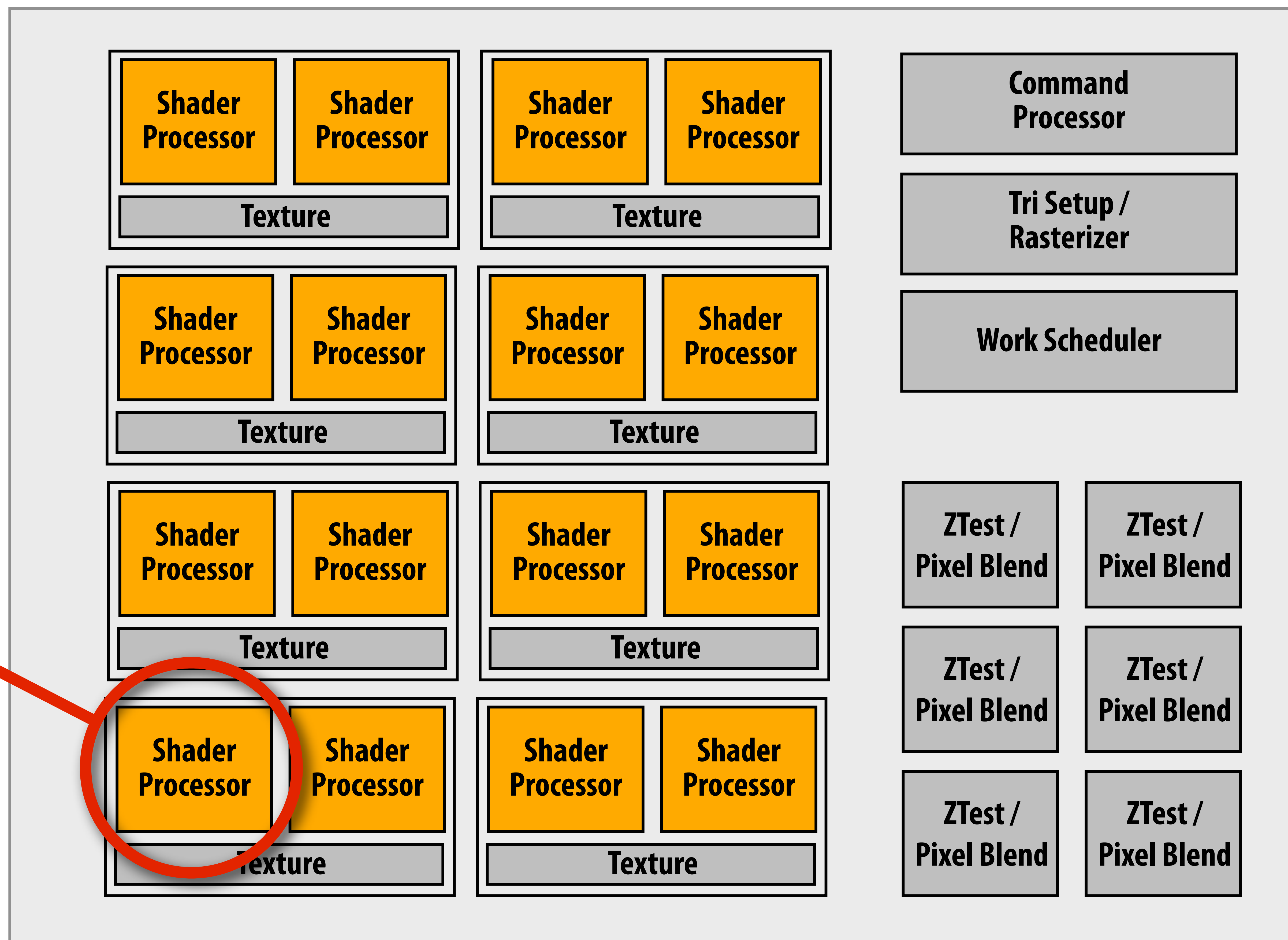


Programmable, just not by application:
(Geometry Engine contained Intel i860XP processor)

NVIDIA GeForce 6800 (2004)



NVIDIA GeForce 8800 (Tesla, 2006)



"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:

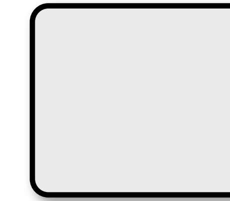
Fragments are processed *independently*, but there is no explicit parallel programming.

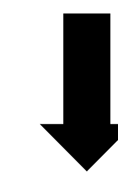
Independent logical sequence of control per *fragment*. ***

*** 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

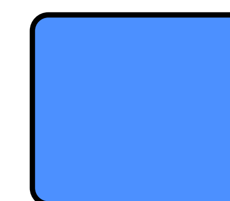
1 unshaded fragment input record





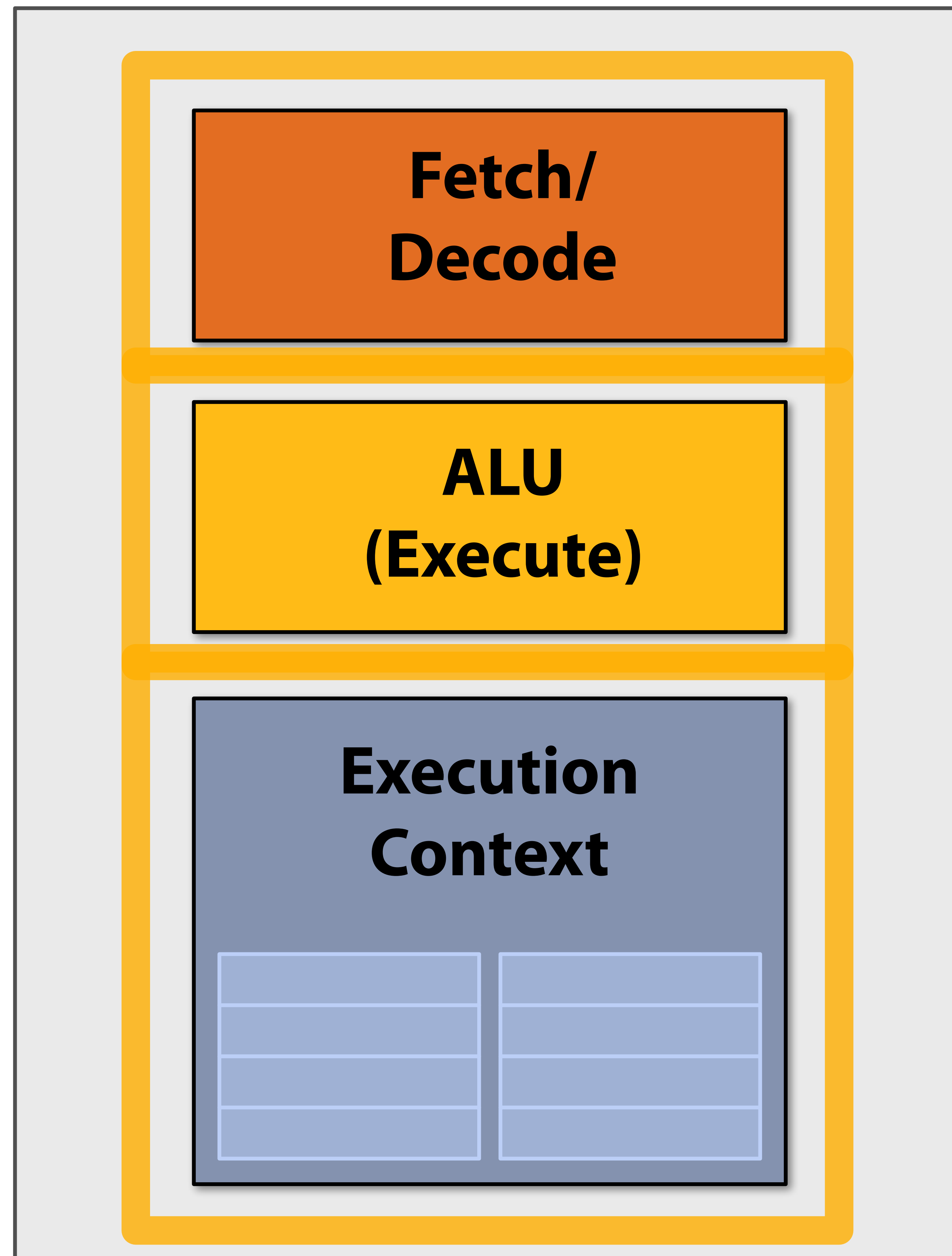
```
sampler mySamp;
Texture2D<float3> myTex;
float3 lightDir;

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);
}
```

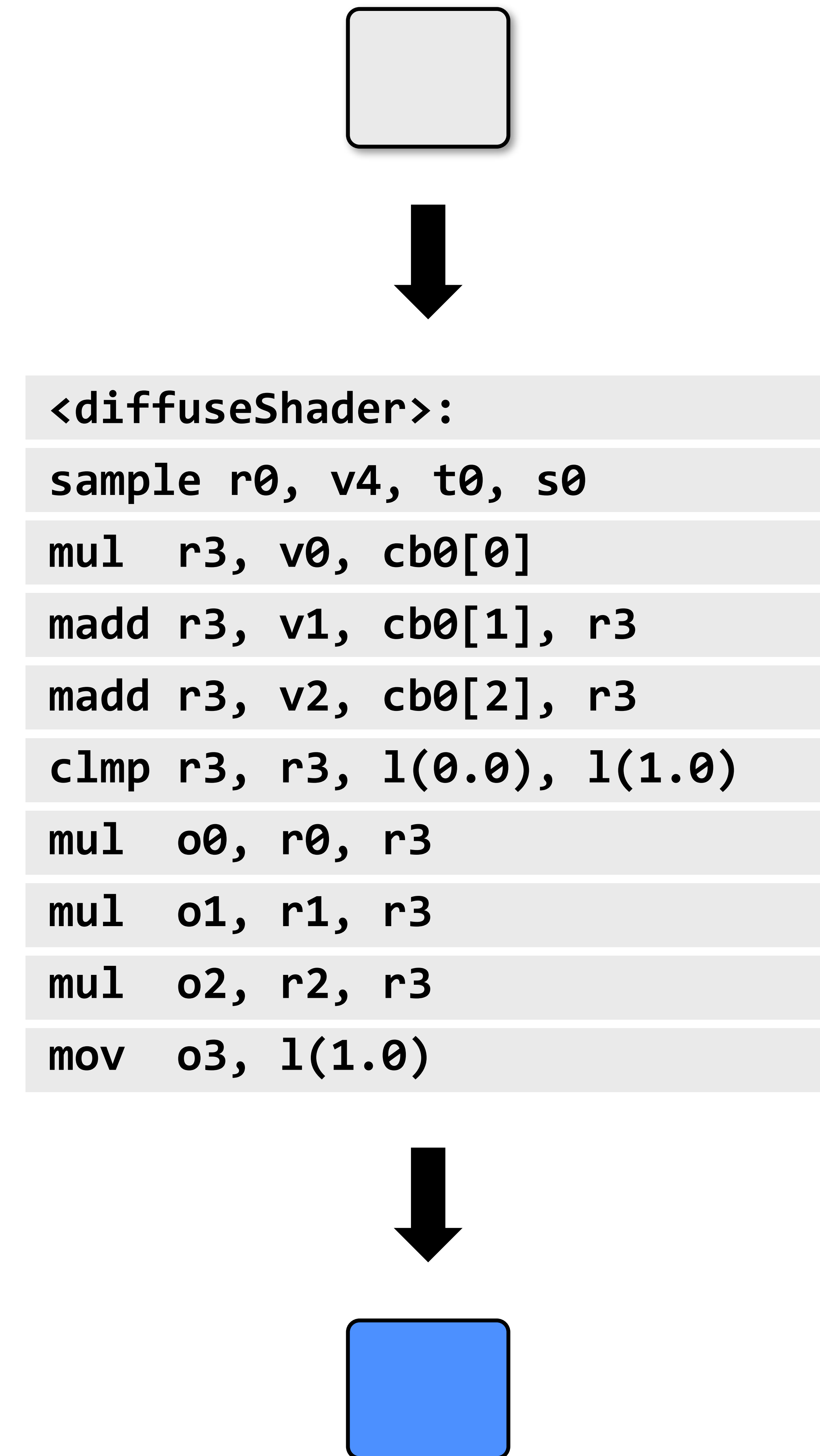
```
<diffuseShader>:
sample r0, v4, t0, s0
mul  r3, v0, cb0[0]
madd r3, v1, cb0[1], r3
madd r3, v2, cb0[2], r3
clmp r3, r3, l(0.0), l(1.0)
mul  o0, r0, r3
mul  o1, r1, r3
mul  o2, r2, r3
mov  o3, l(1.0)
```



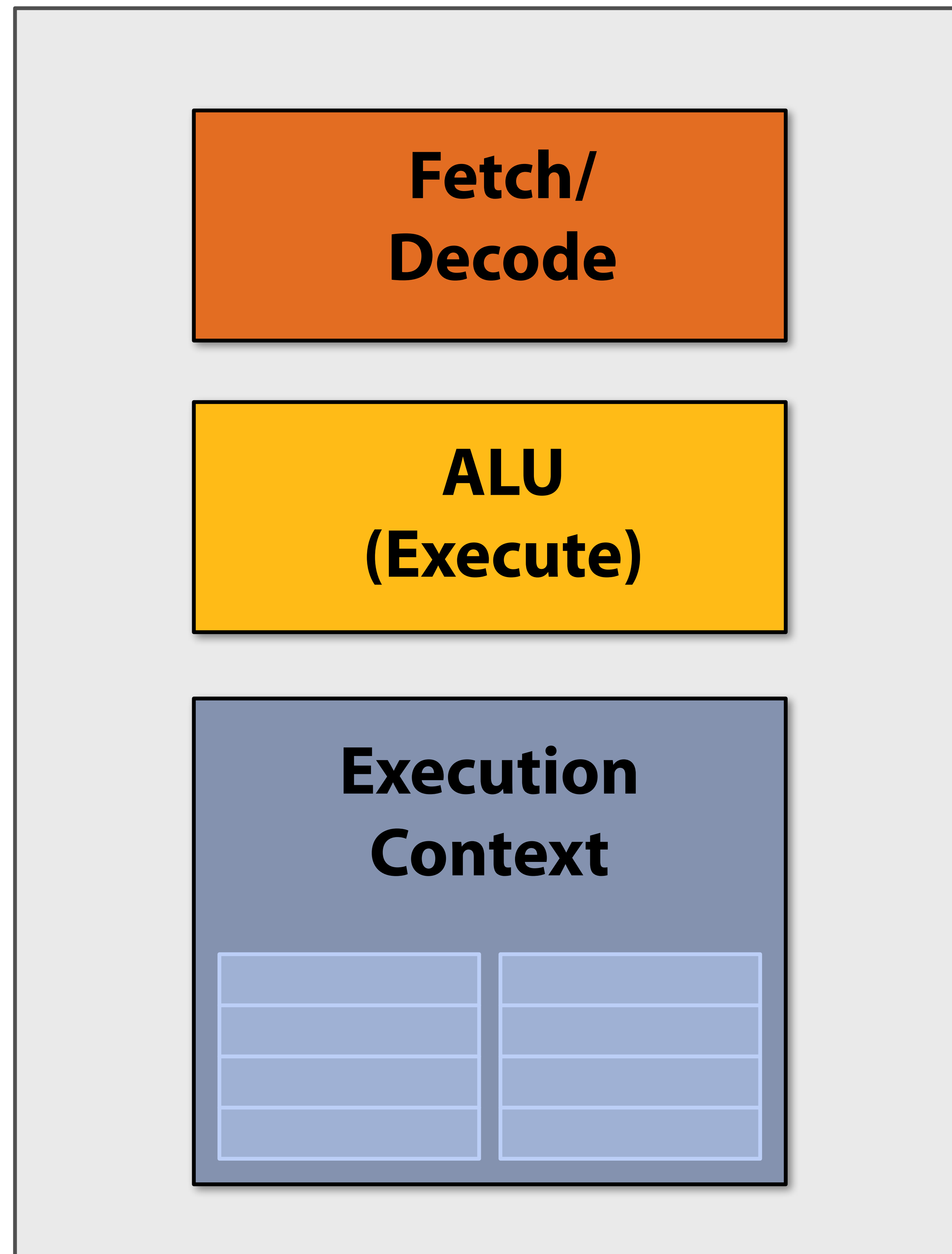
1 shaded fragment output record

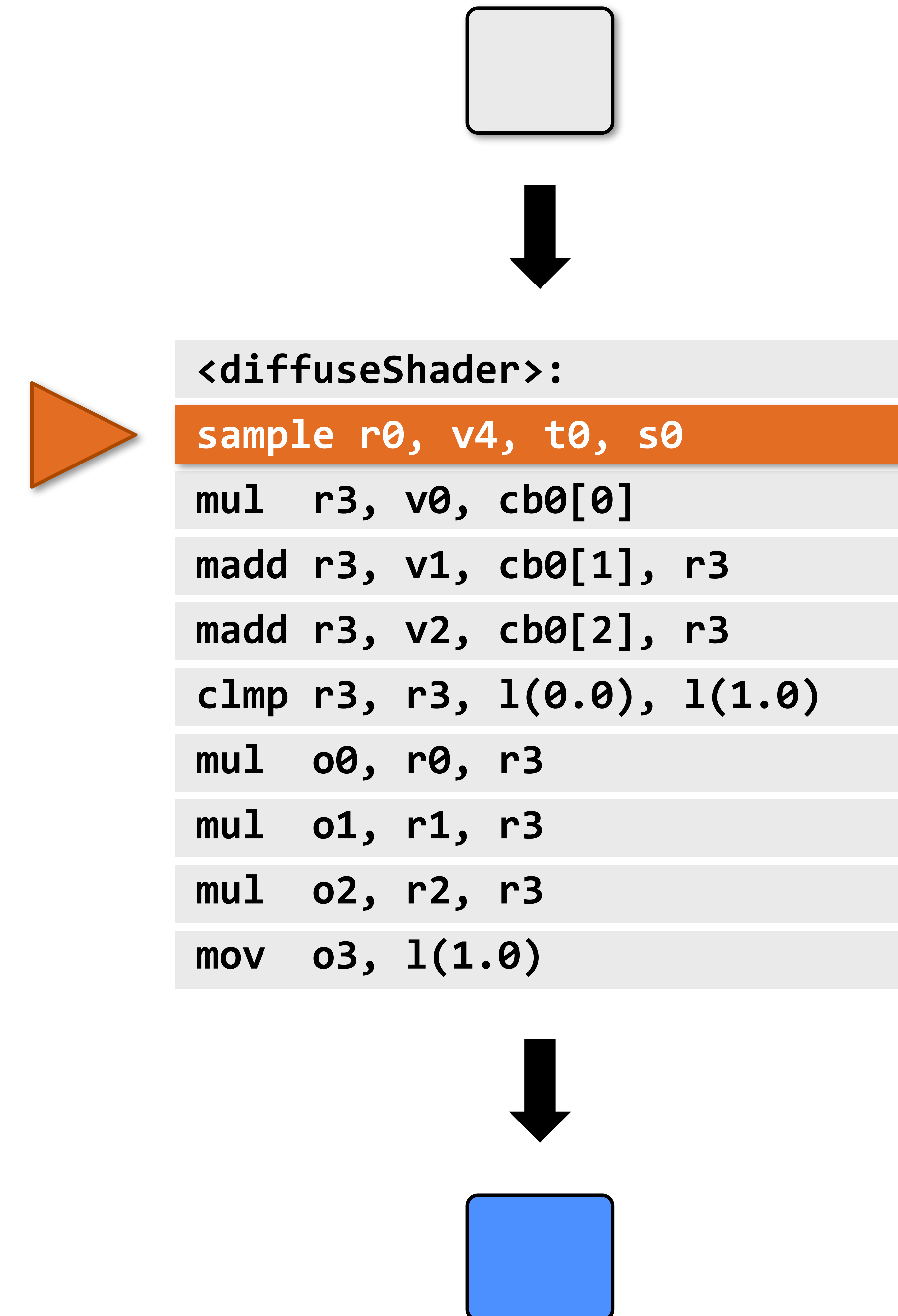
Execute shader



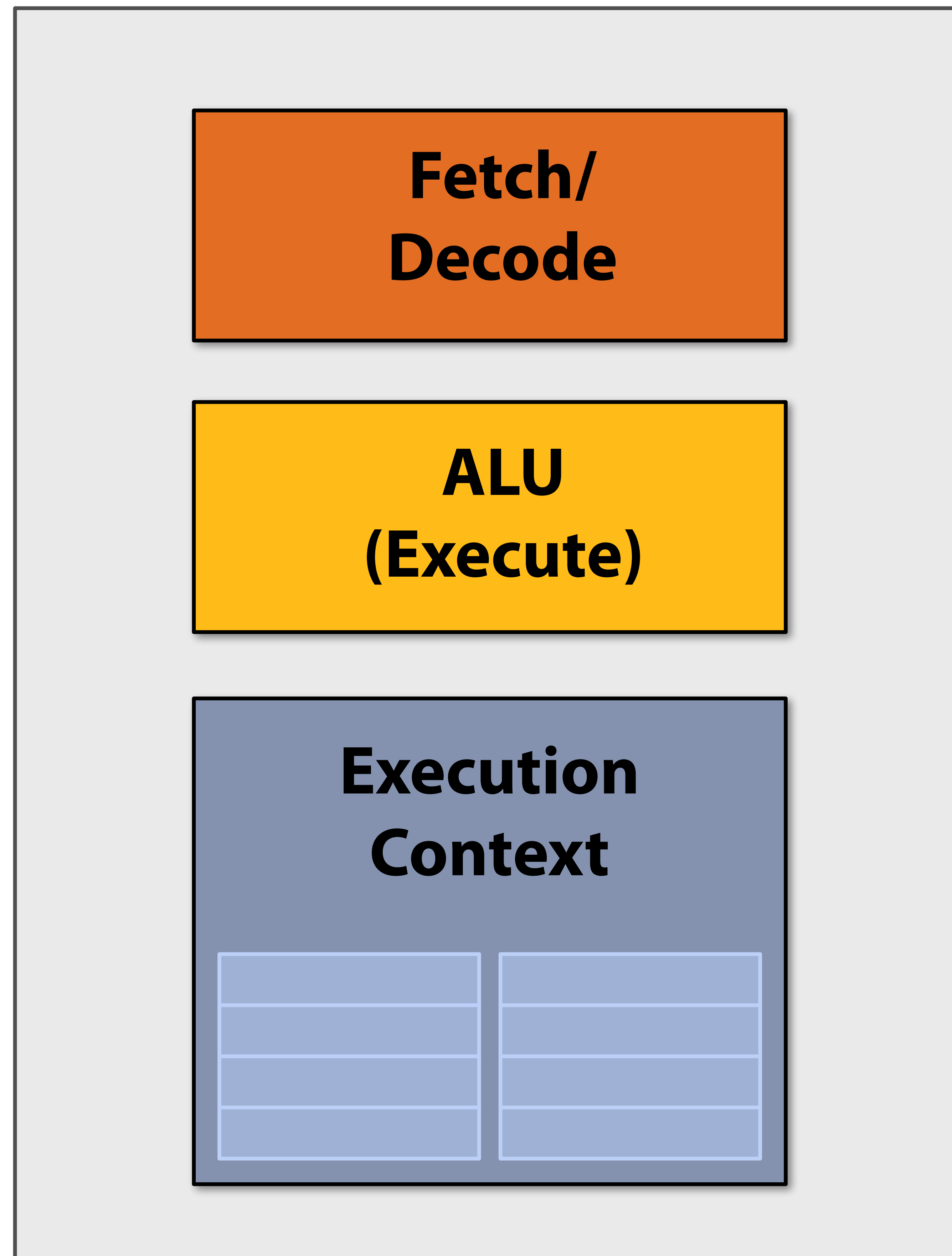


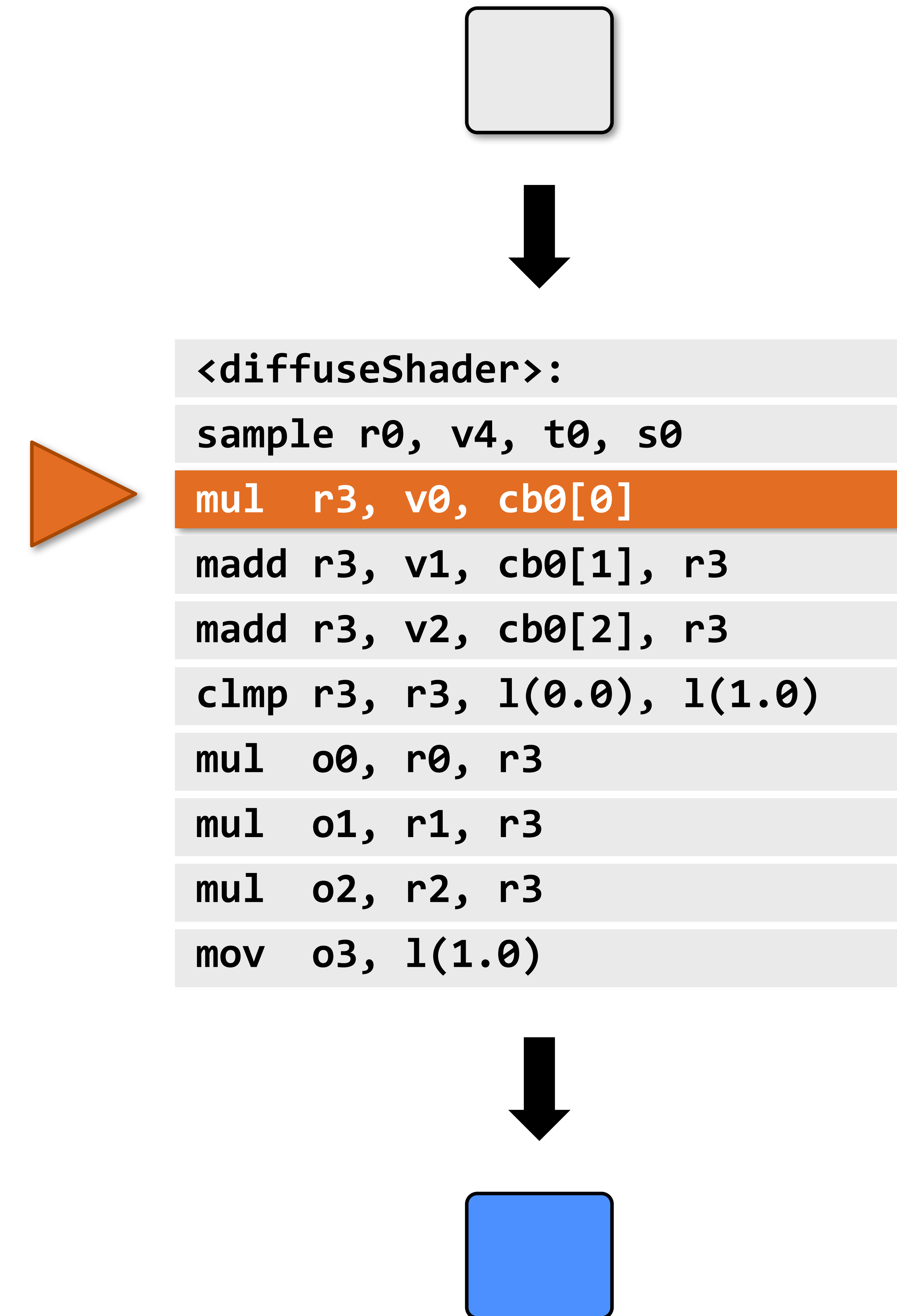
Execute shader



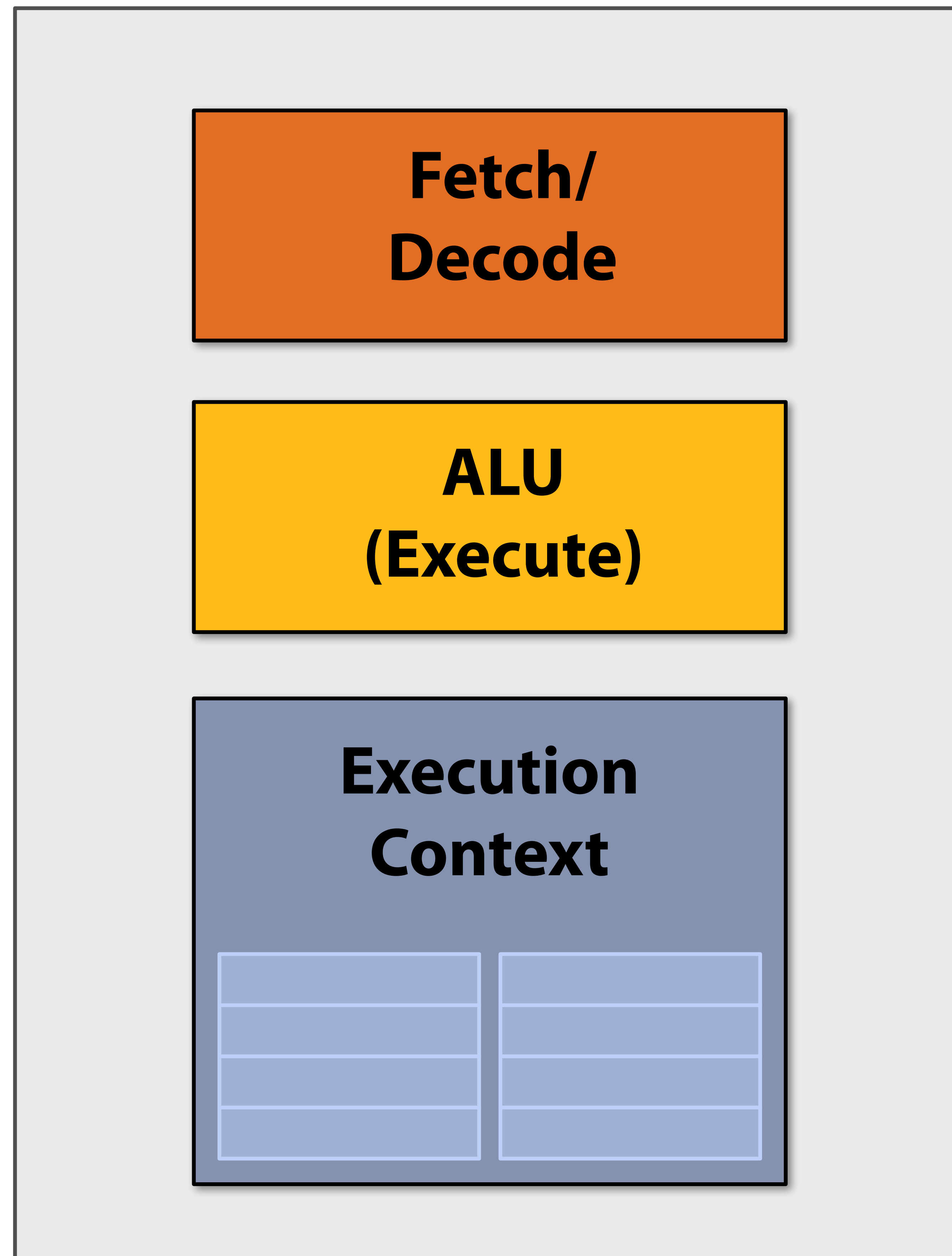


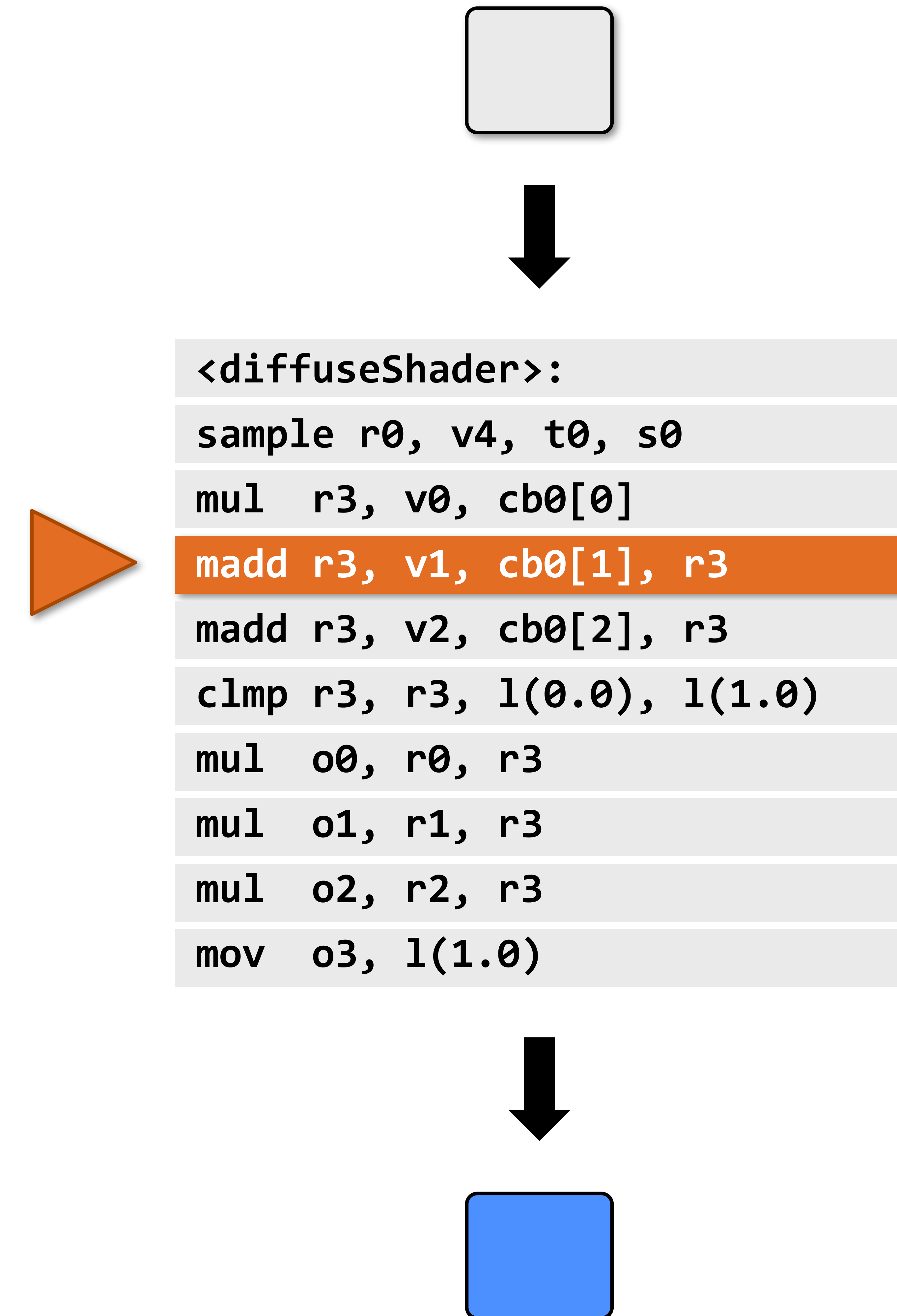
Execute shader



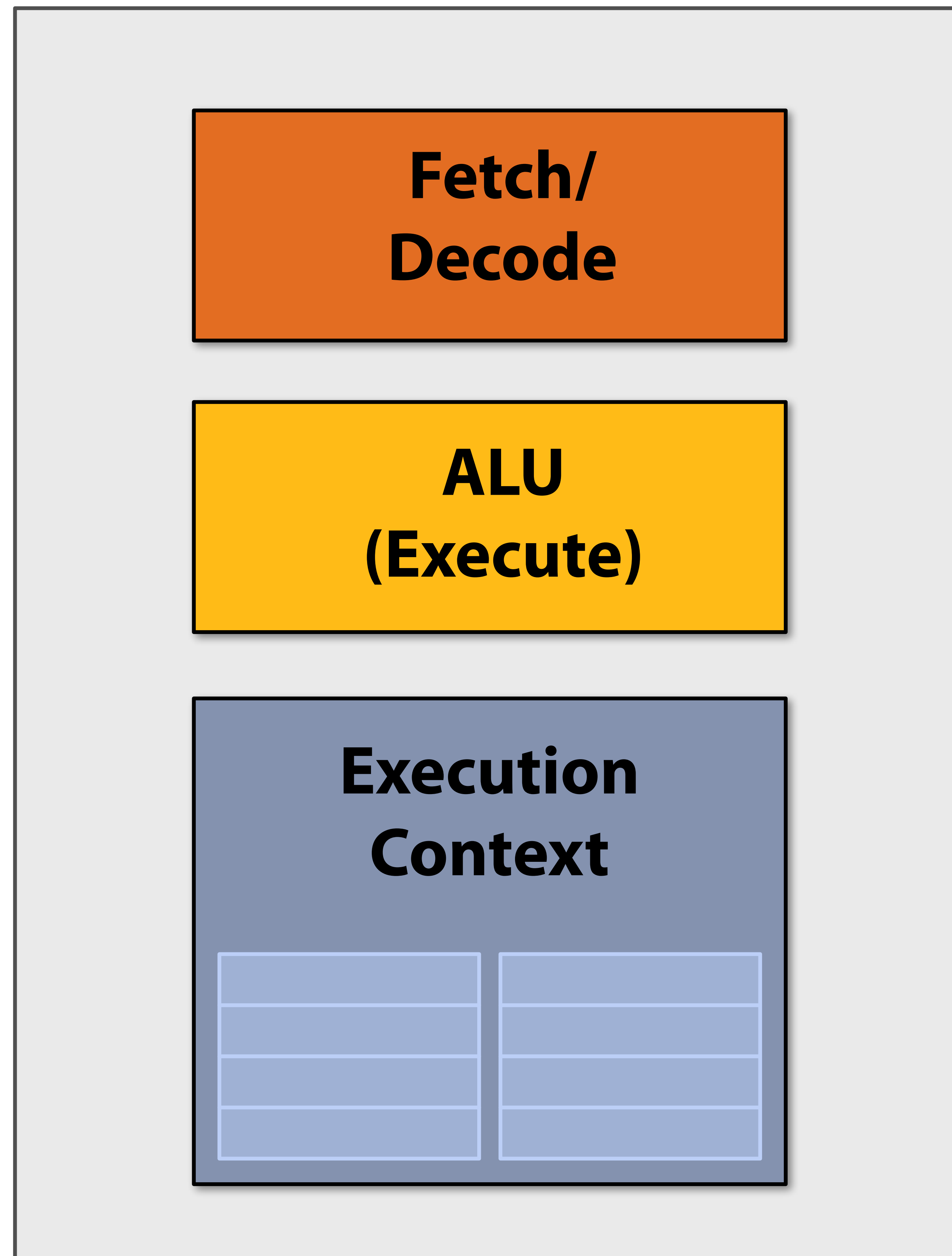


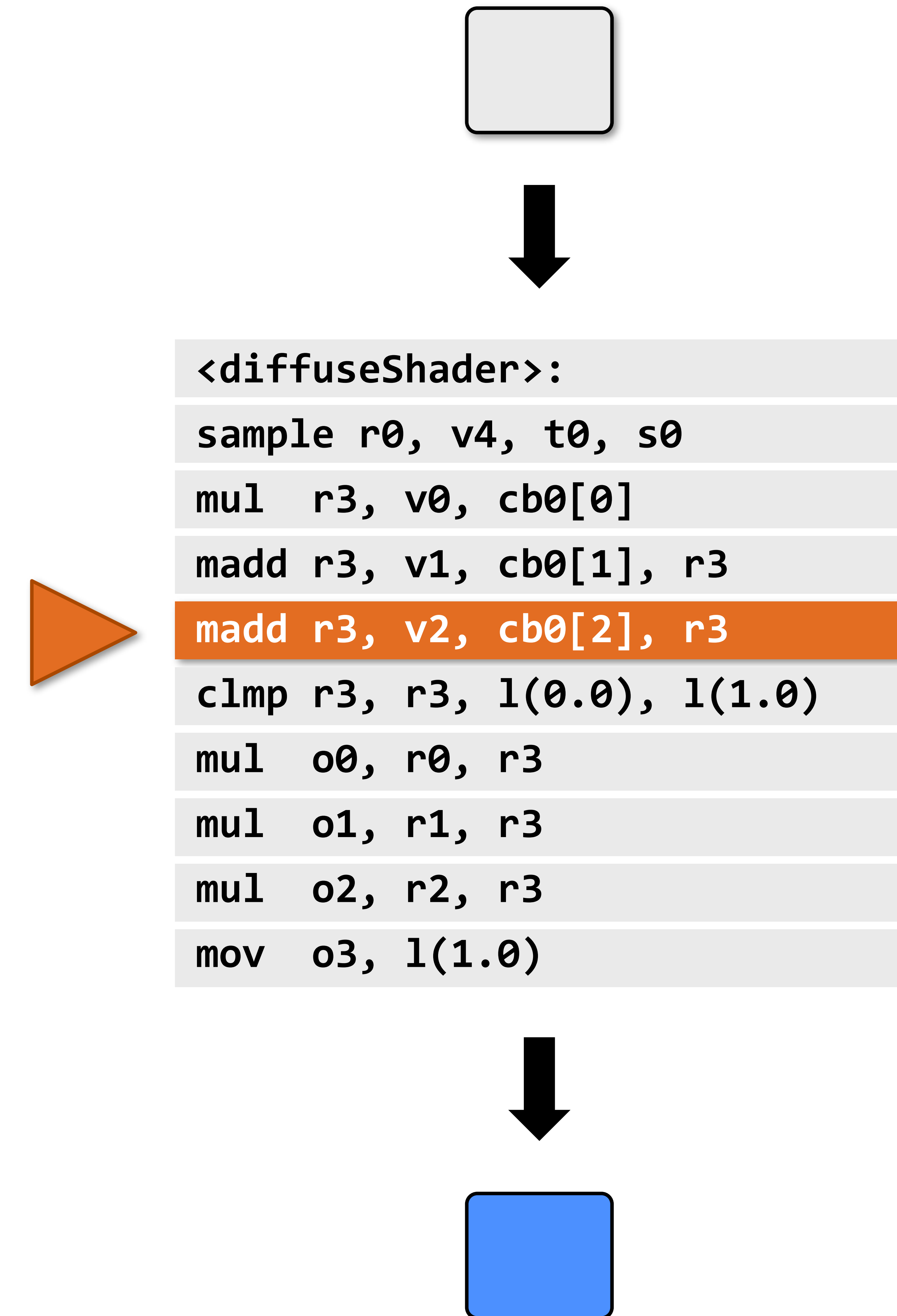
Execute shader



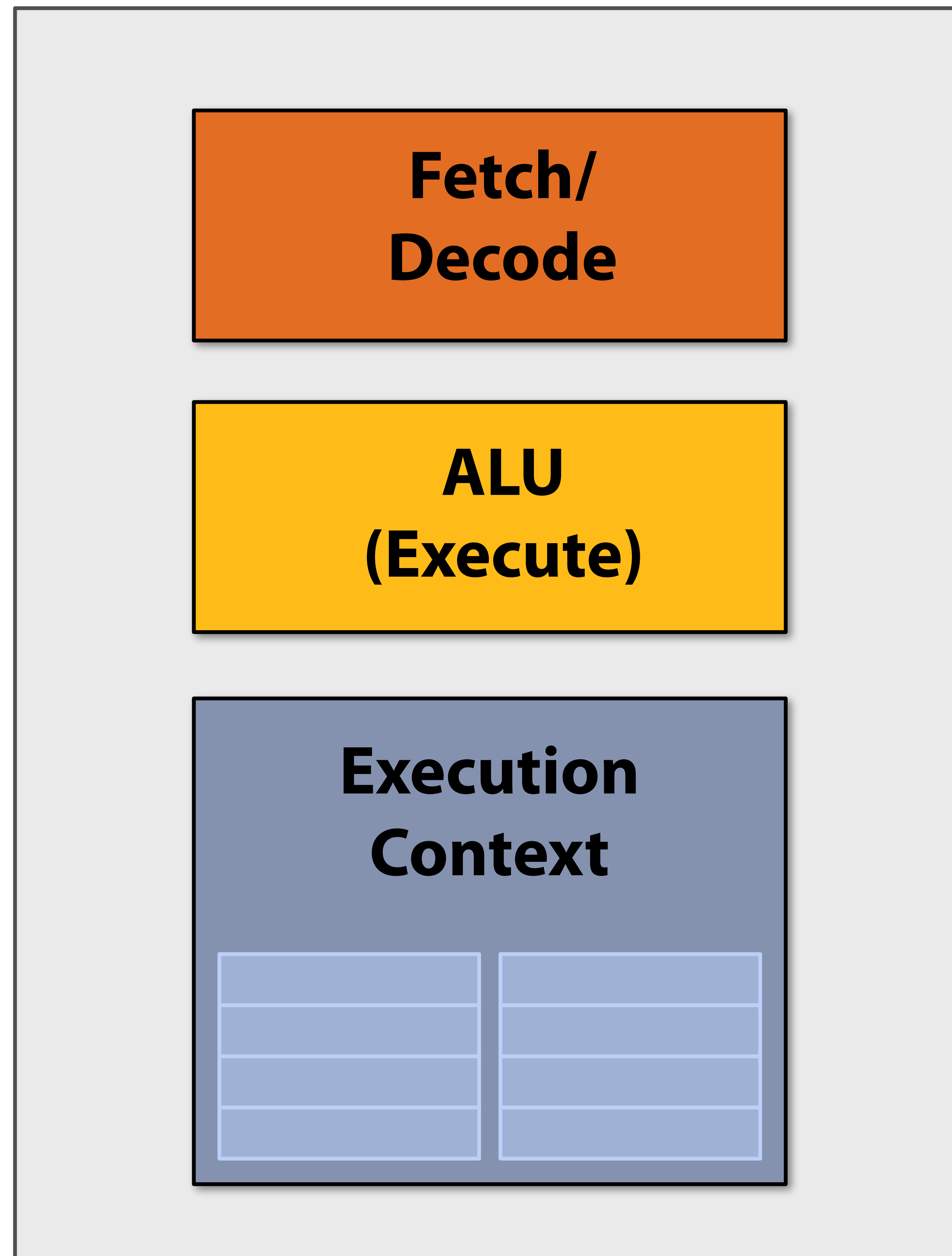


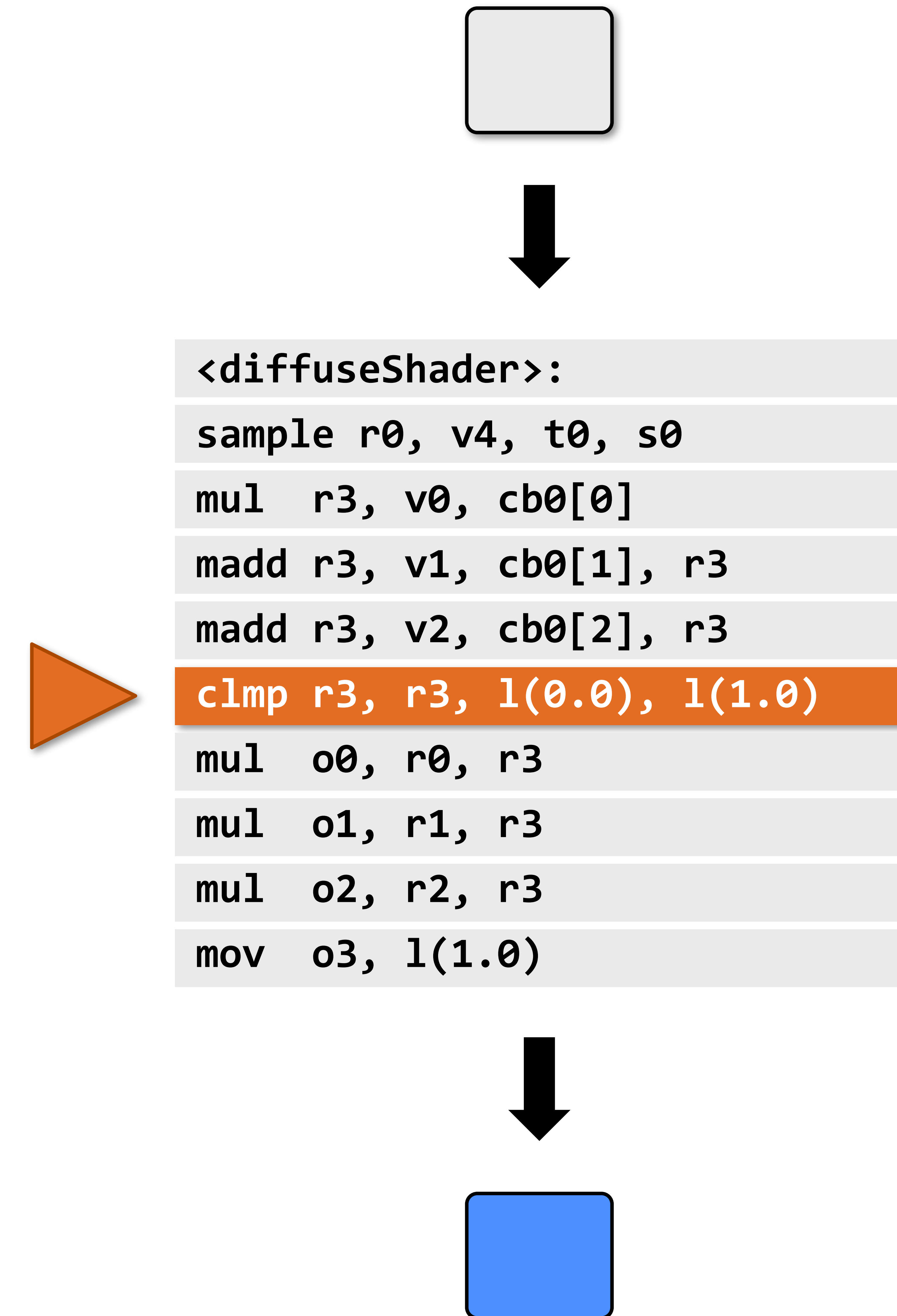
Execute shader



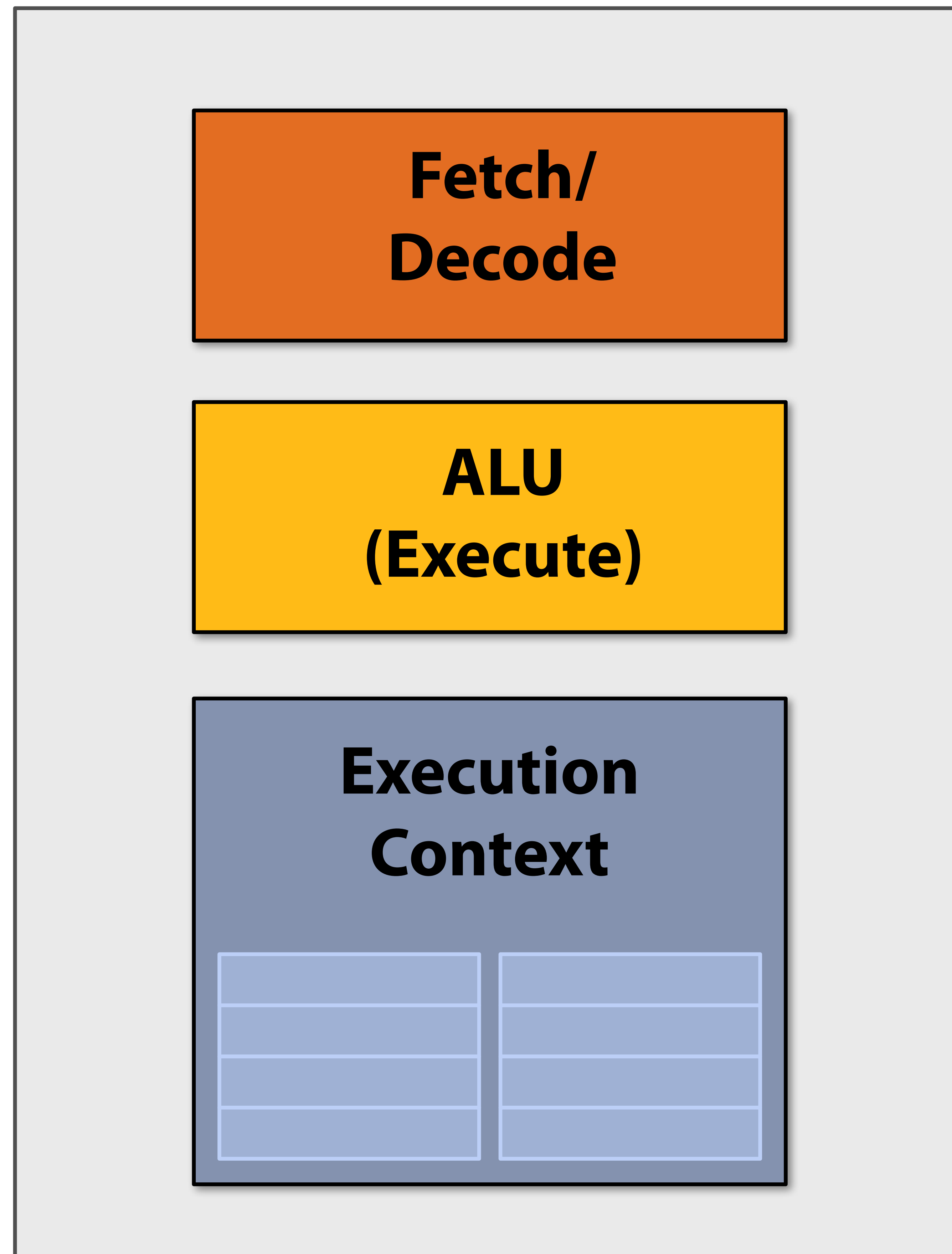


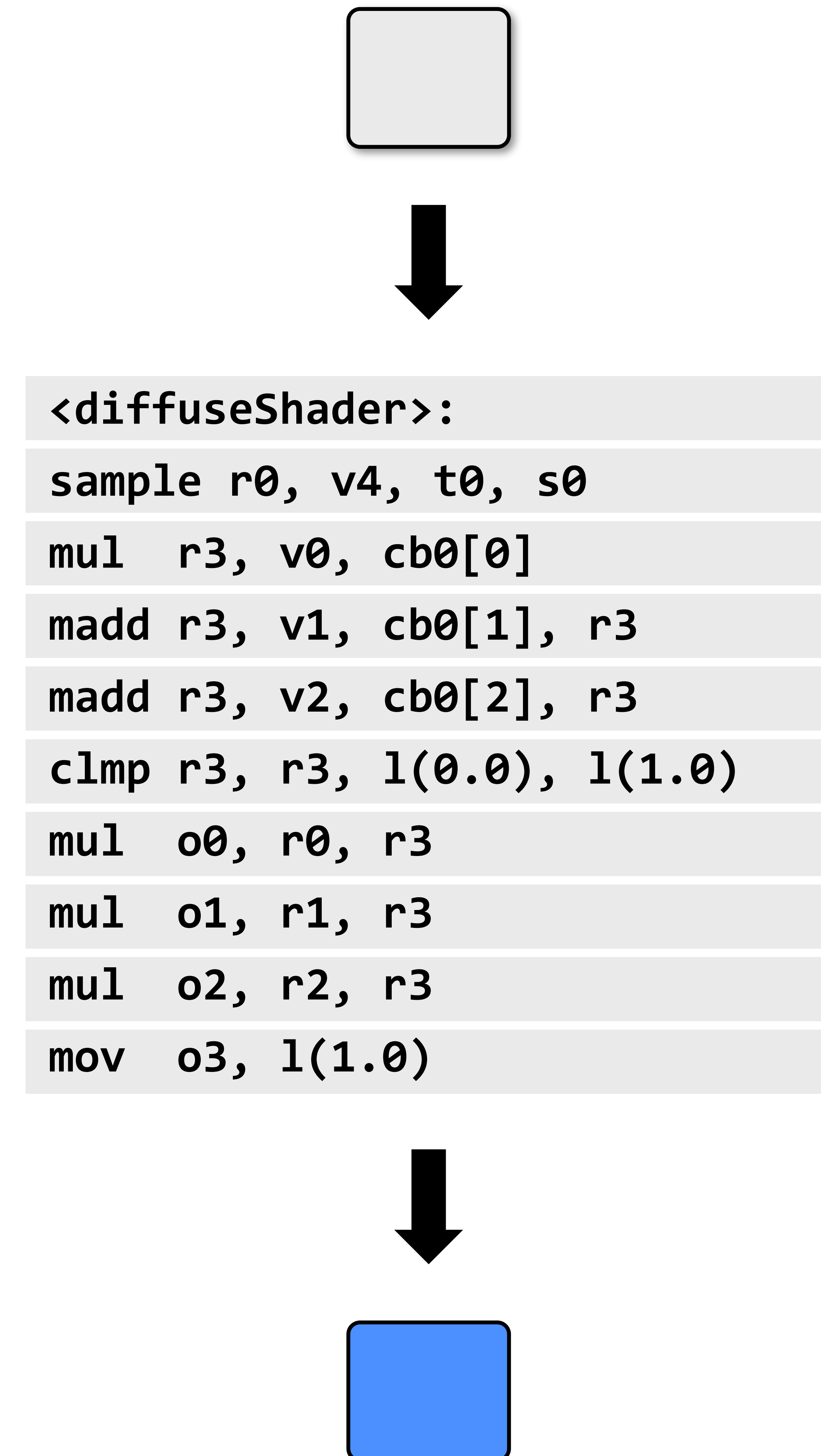
Execute shader



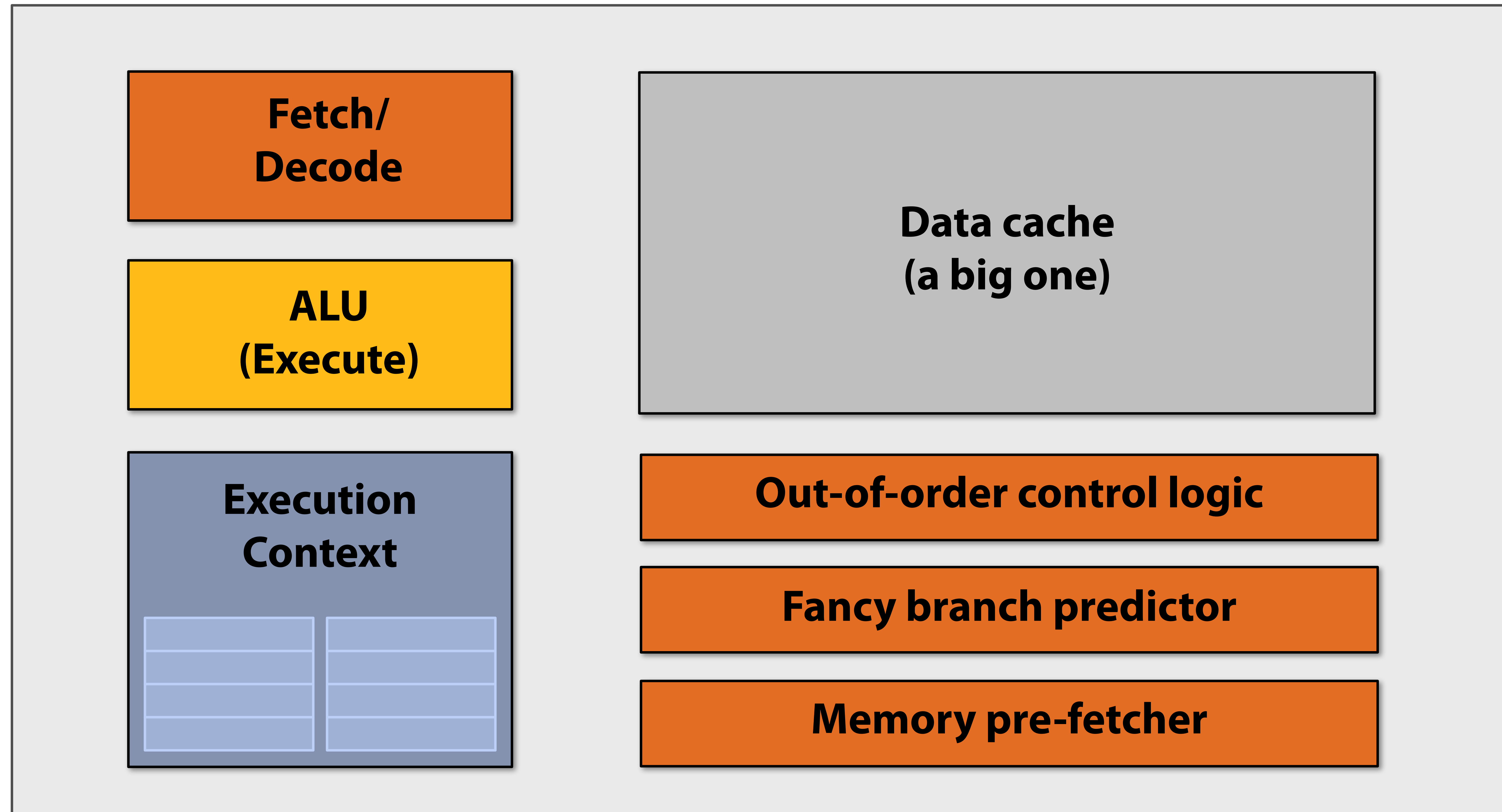


Execute shader

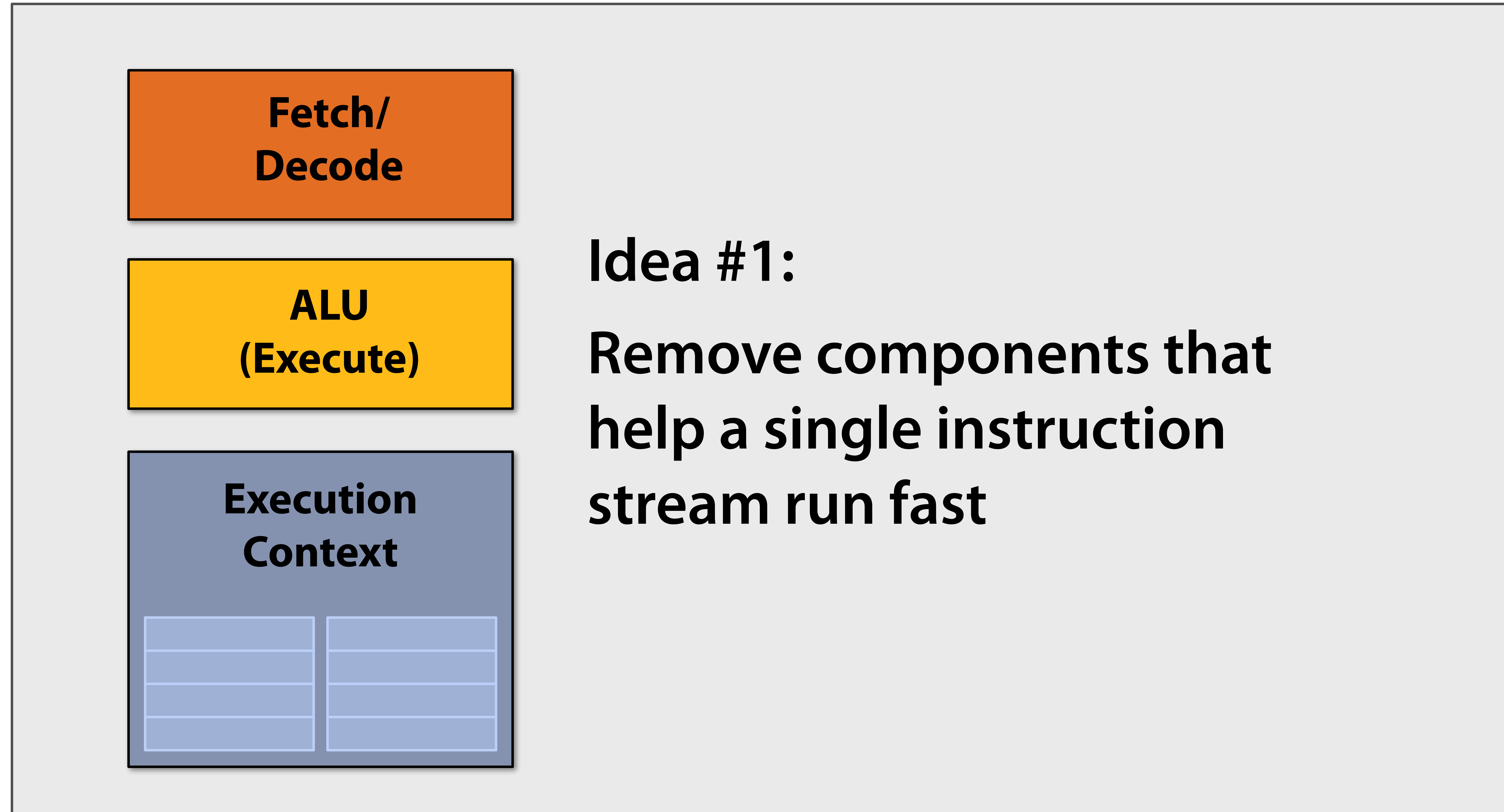




“CPU-style” cores

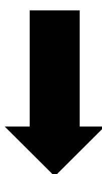


Slimming down

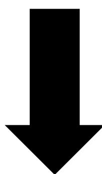


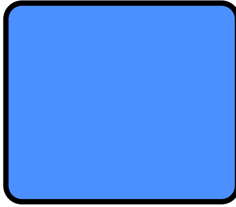
Two cores (two fragments in parallel)

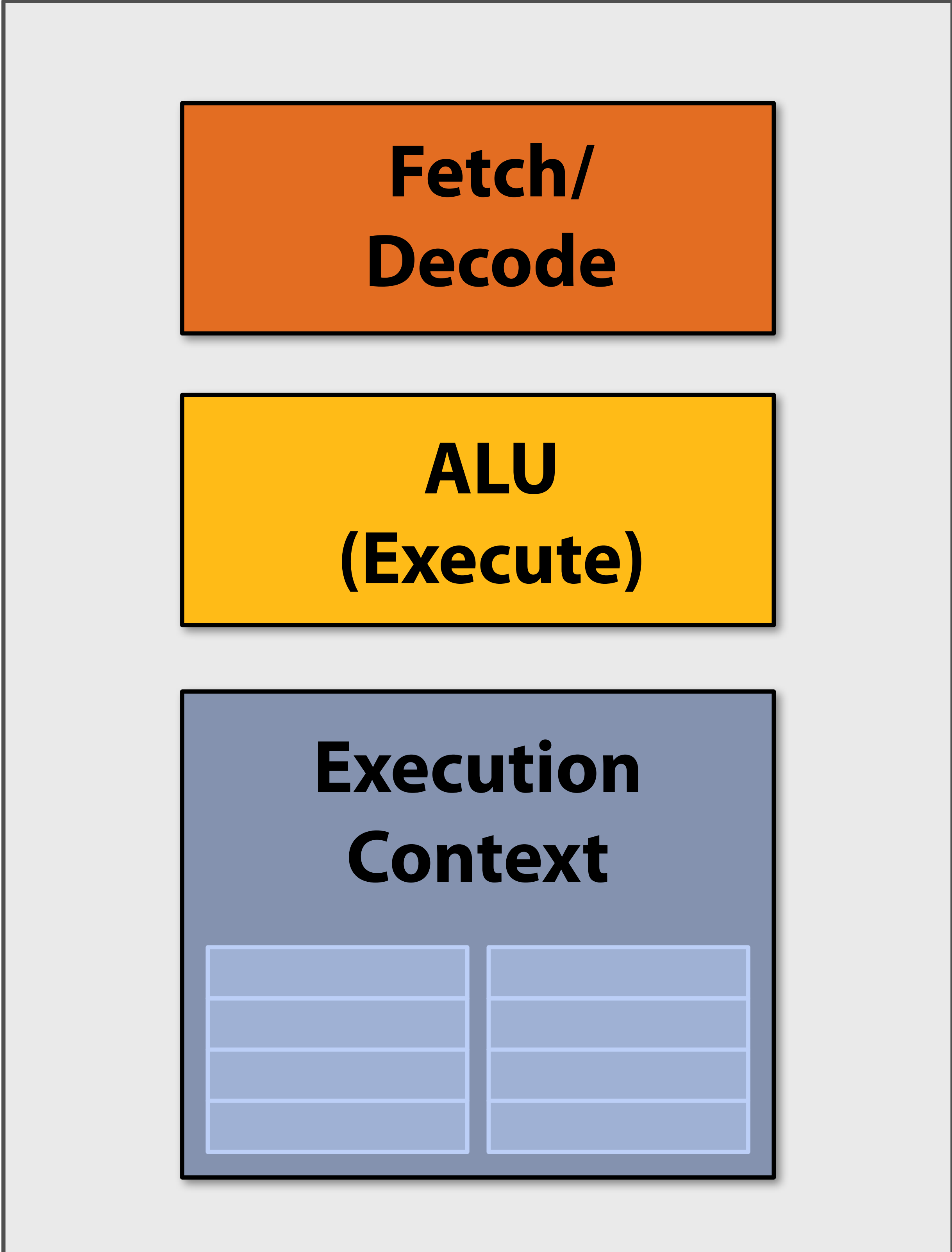
fragment 1

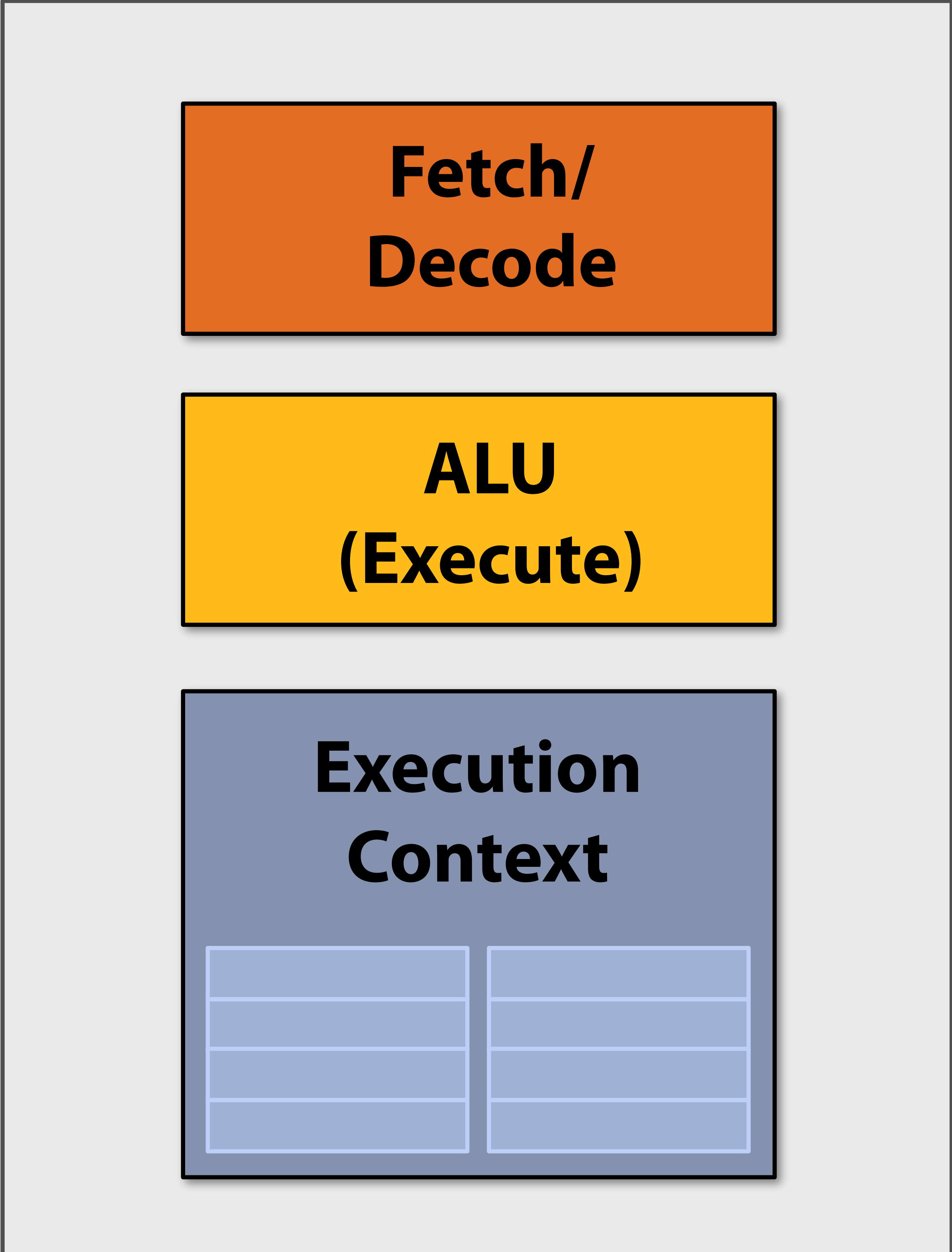


```
<diffuseShader>:
sample r0, v4, t0, s0
mul  r3, v0, cb0[0]
madd r3, v1, cb0[1], r3
madd r3, v2, cb0[2], r3
clmp r3, r3, l(0.0), l(1.0)
mul  o0, r0, r3
mul  o1, r1, r3
mul  o2, r2, r3
mov  o3, l(1.0)
```

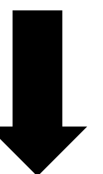




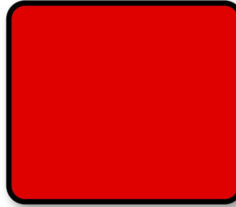




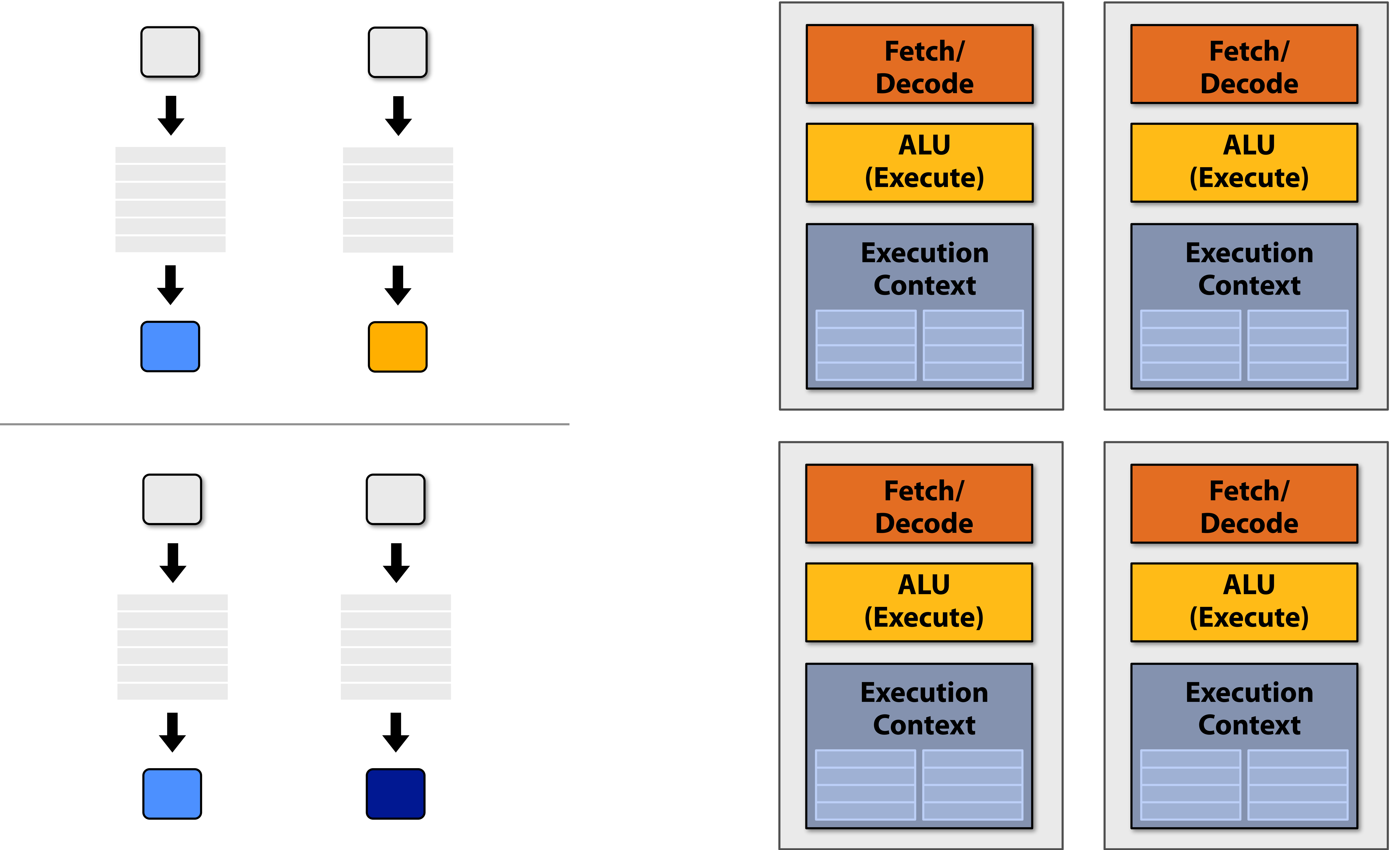
fragment 2



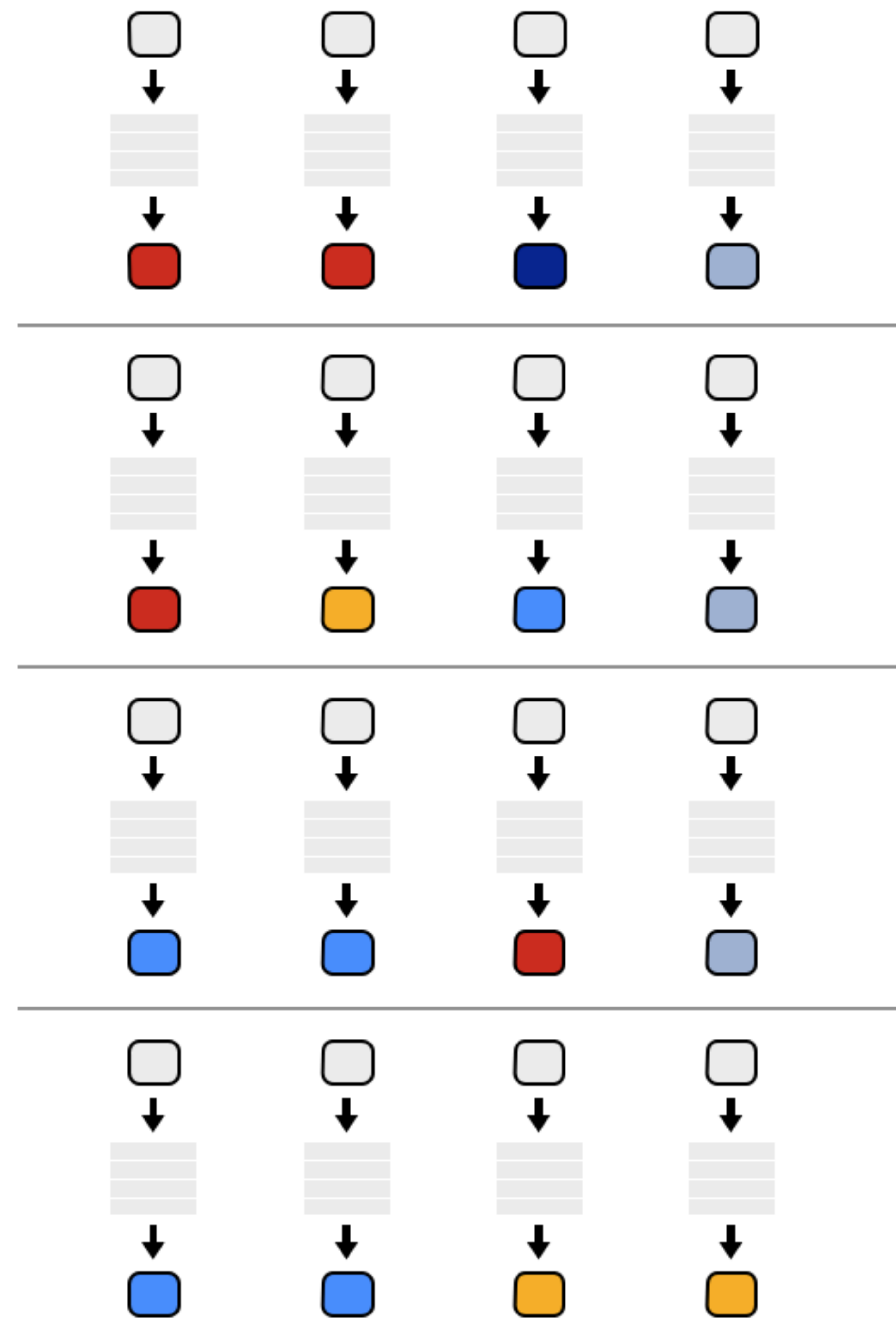

```
<diffuseShader>:
sample r0, v4, t0, s0
mul  r3, v0, cb0[0]
madd r3, v1, cb0[1], r3
madd r3, v2, cb0[2], r3
clmp r3, r3, l(0.0), l(1.0)
mul  o0, r0, r3
mul  o1, r1, r3
mul  o2, r2, r3
mov  o3, l(1.0)
```

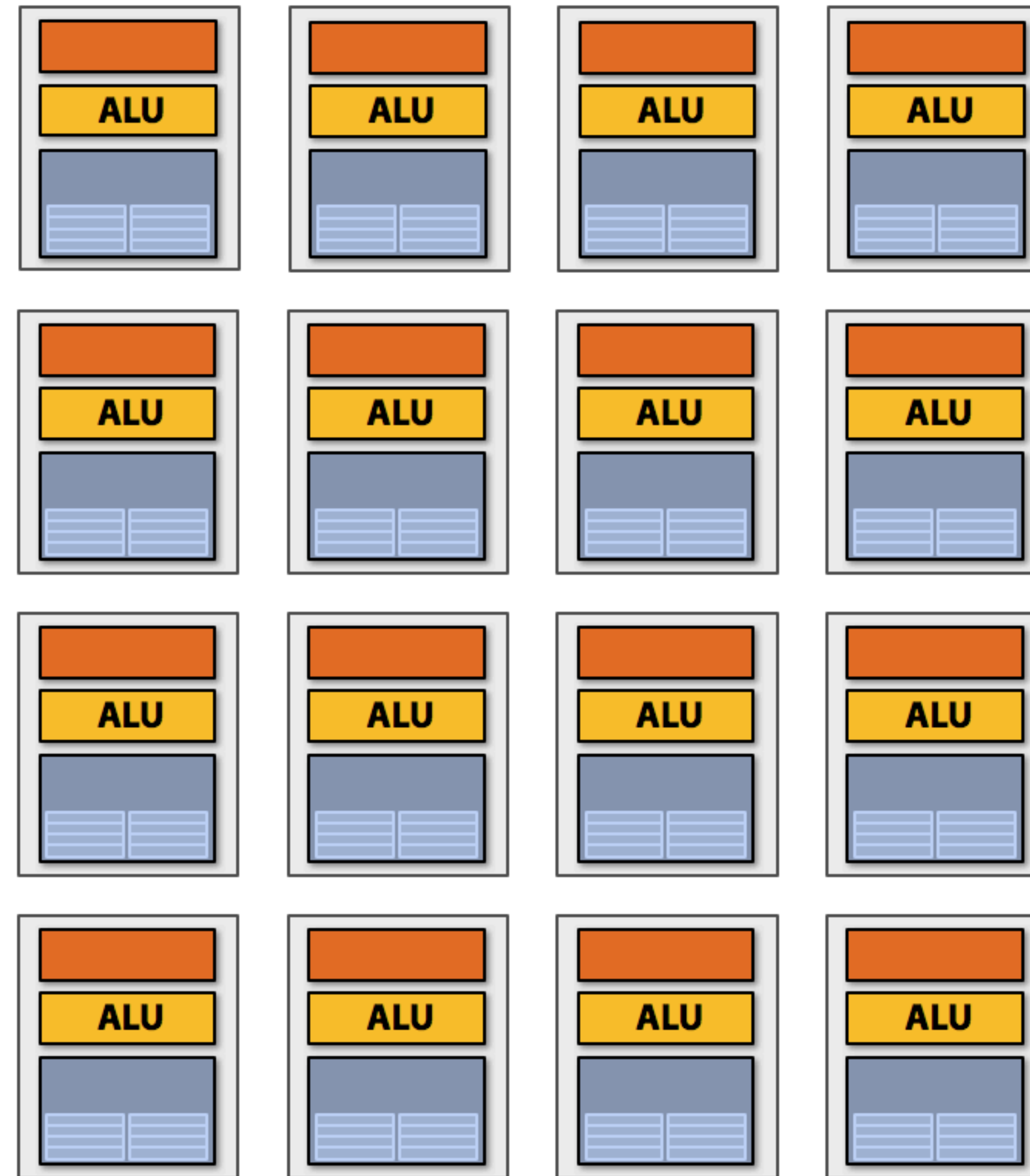


Four cores (four fragments in parallel)



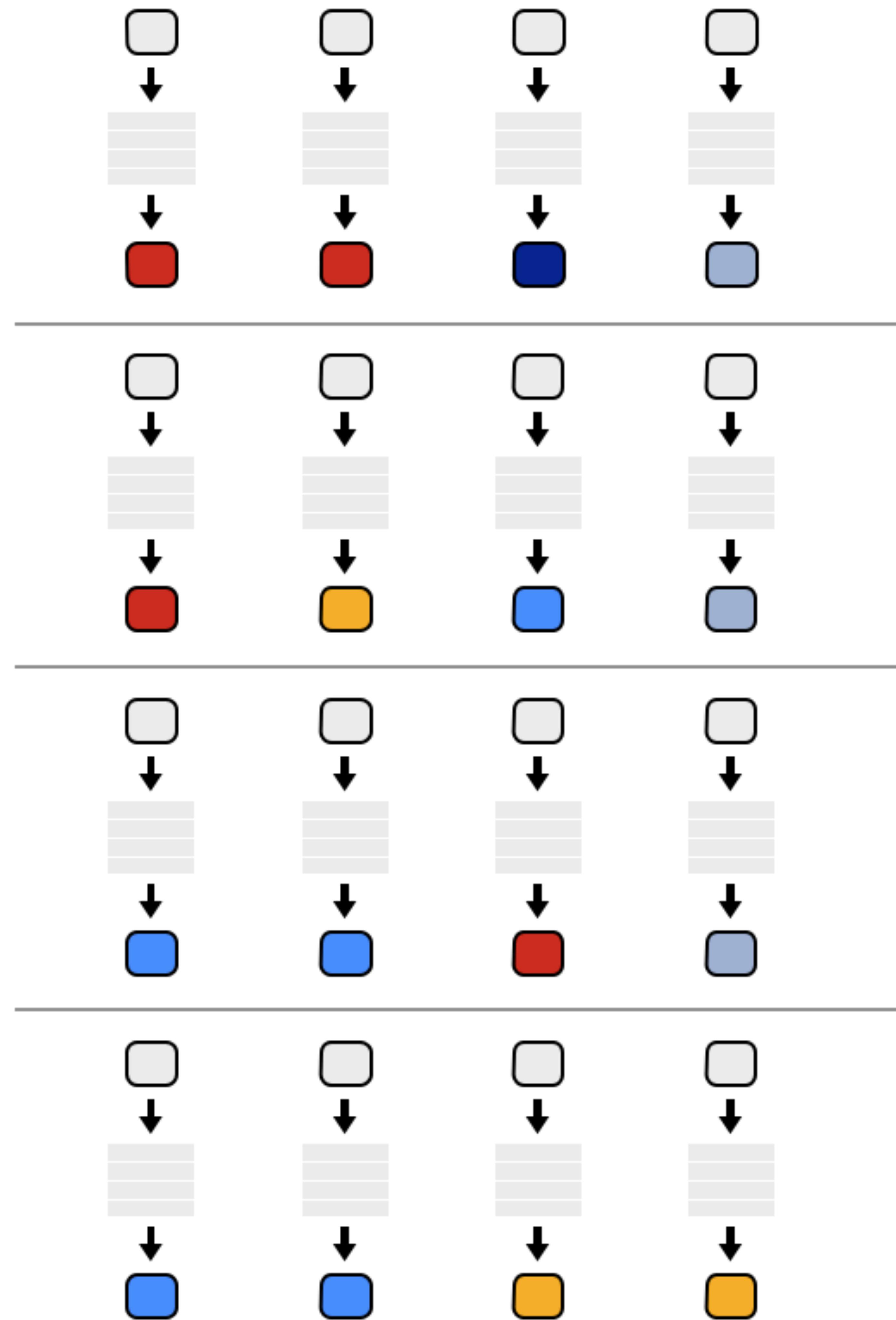
Sixteen cores (sixteen fragments in parallel)





16 cores = 16 simultaneous instruction streams

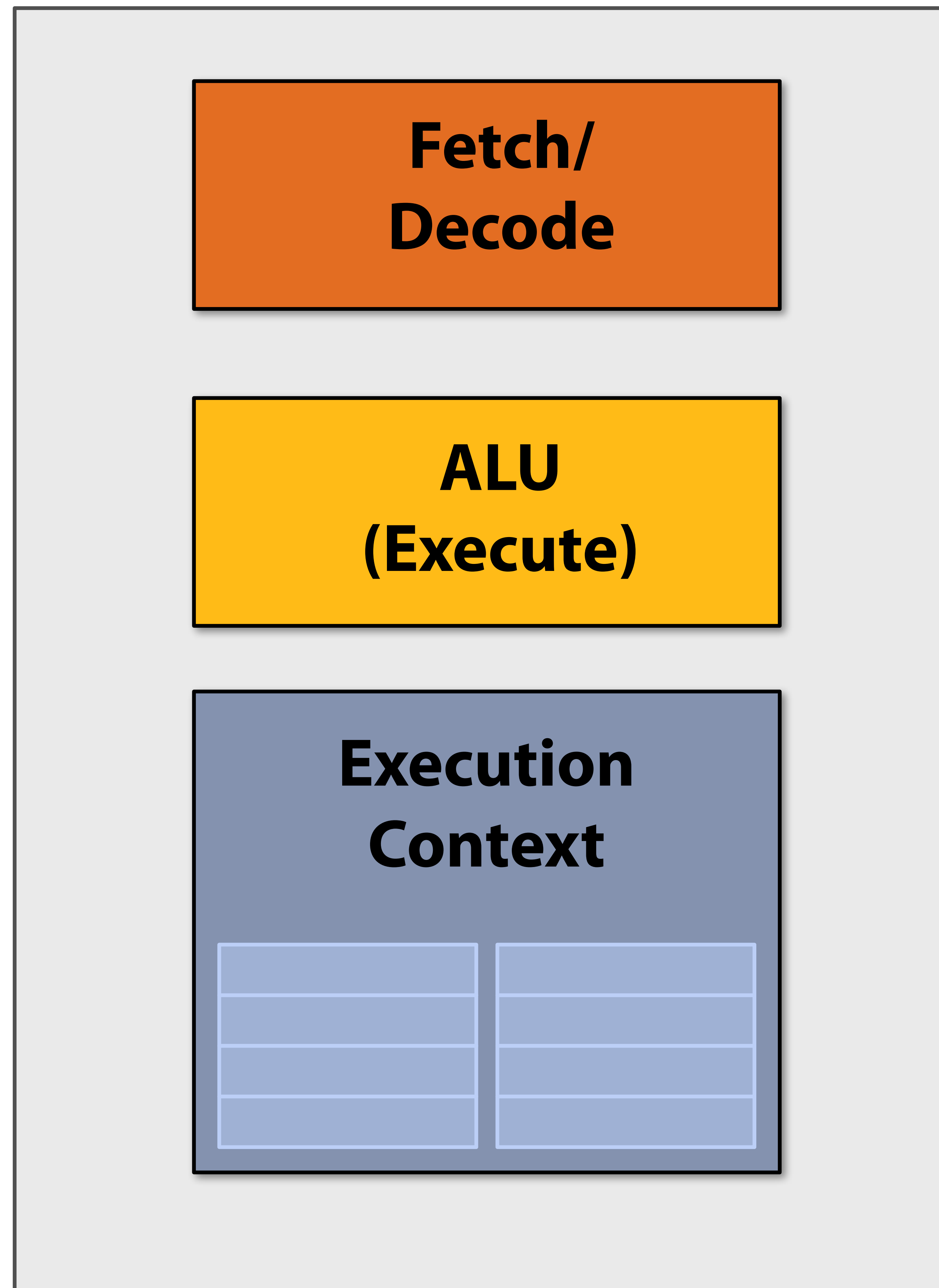
Instruction stream sharing



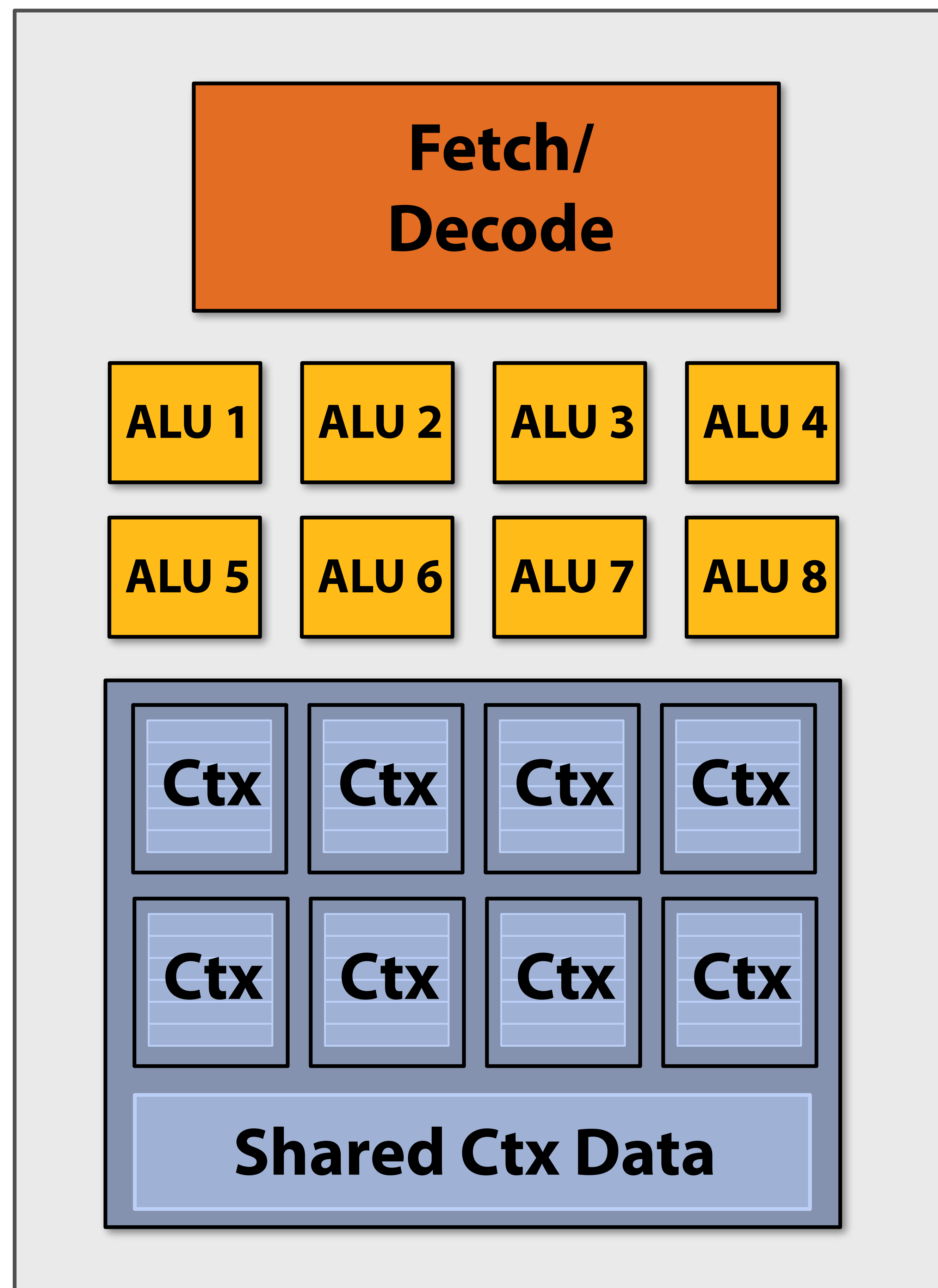
But ... many fragments should be able to share an instruction stream!

```
<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



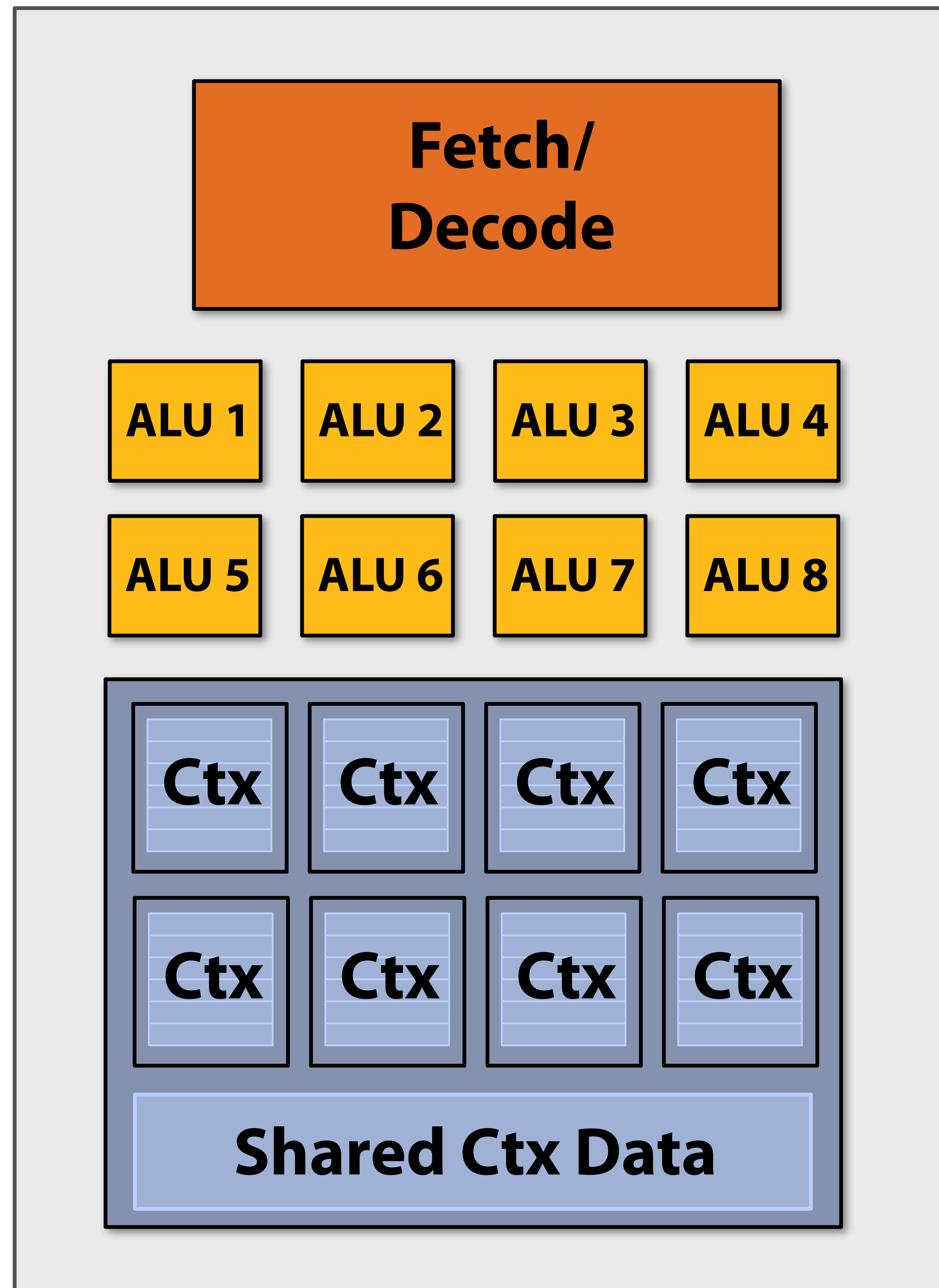
Add ALUs



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

SIMD processing

Modifying the shader

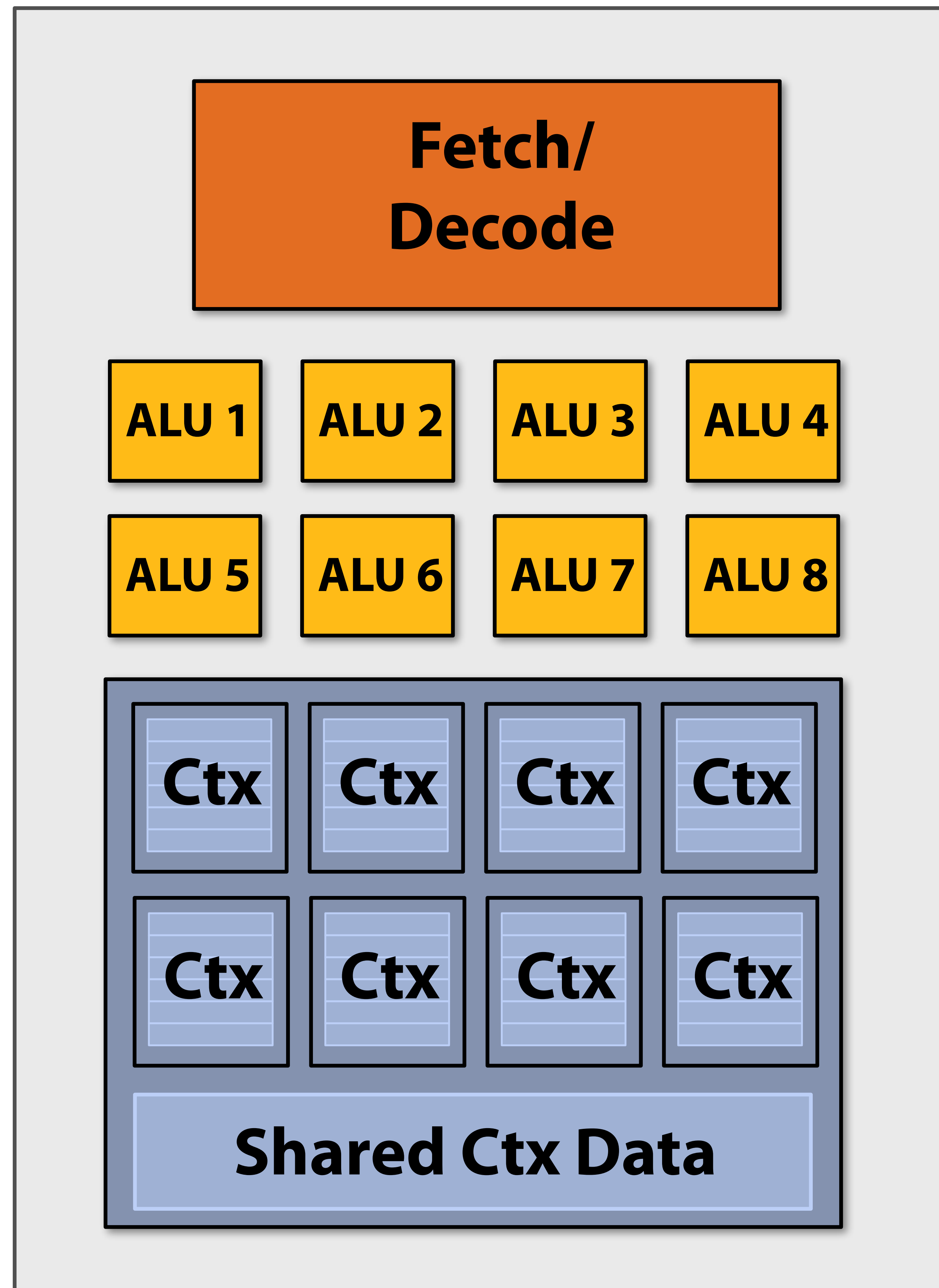



```
<diffuseShader>:
sample r0, v4, t0, s0
mul  r3, v0, cb0[0]
madd r3, v1, cb0[1], r3
madd r3, v2, cb0[2], r3
clmp r3, r3, l(0.0), l(1.0)
mul  o0, r0, r3
mul  o1, r1, r3
mul  o2, r2, r3
mov  o3, l(1.0)
```

Original compiled shader:

Processes one fragment using scalar ops on scalar registers

Modifying the shader

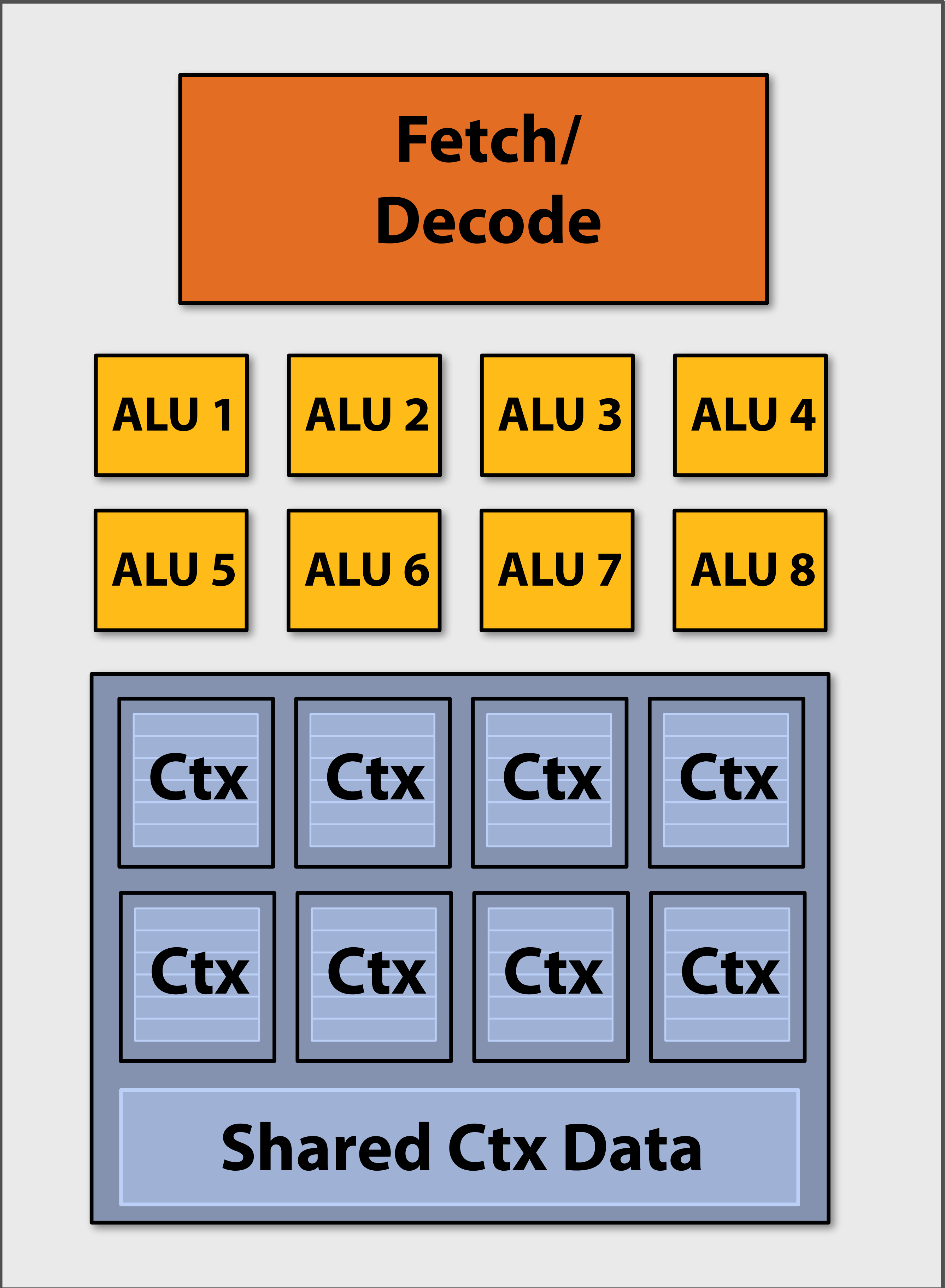


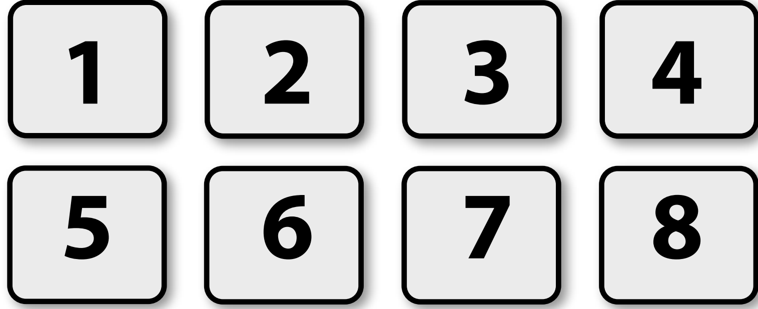

```
<VEC8_diffuseShader>:
VEC8_sample vec_r0, vec_v4, t0, vec_s0
VEC8_mul  vec_r3, vec_v0, cb0[0]
VEC8_madd vec_r3, vec_v1, cb0[1], vec_r3
VEC8_madd vec_r3, vec_v2, cb0[2], vec_r3
VEC8_clmp vec_r3, vec_r3, l(0.0), l(1.0)
VEC8_mul  vec_o0, vec_r0, vec_r3
VEC8_mul  vec_o1, vec_r1, vec_r3
VEC8_mul  vec_o2,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

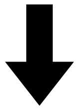
New compiled shader:

Processes eight fragments using vector ops on vector registers

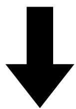
Modifying the shader

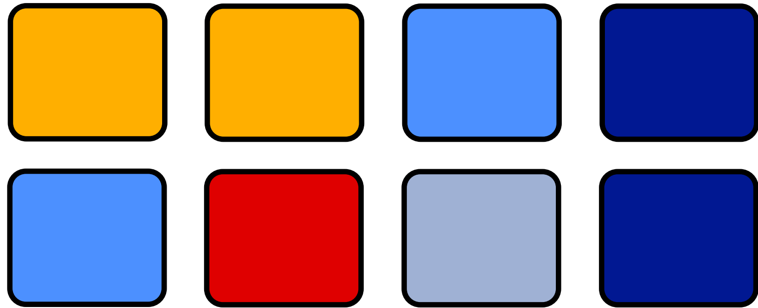




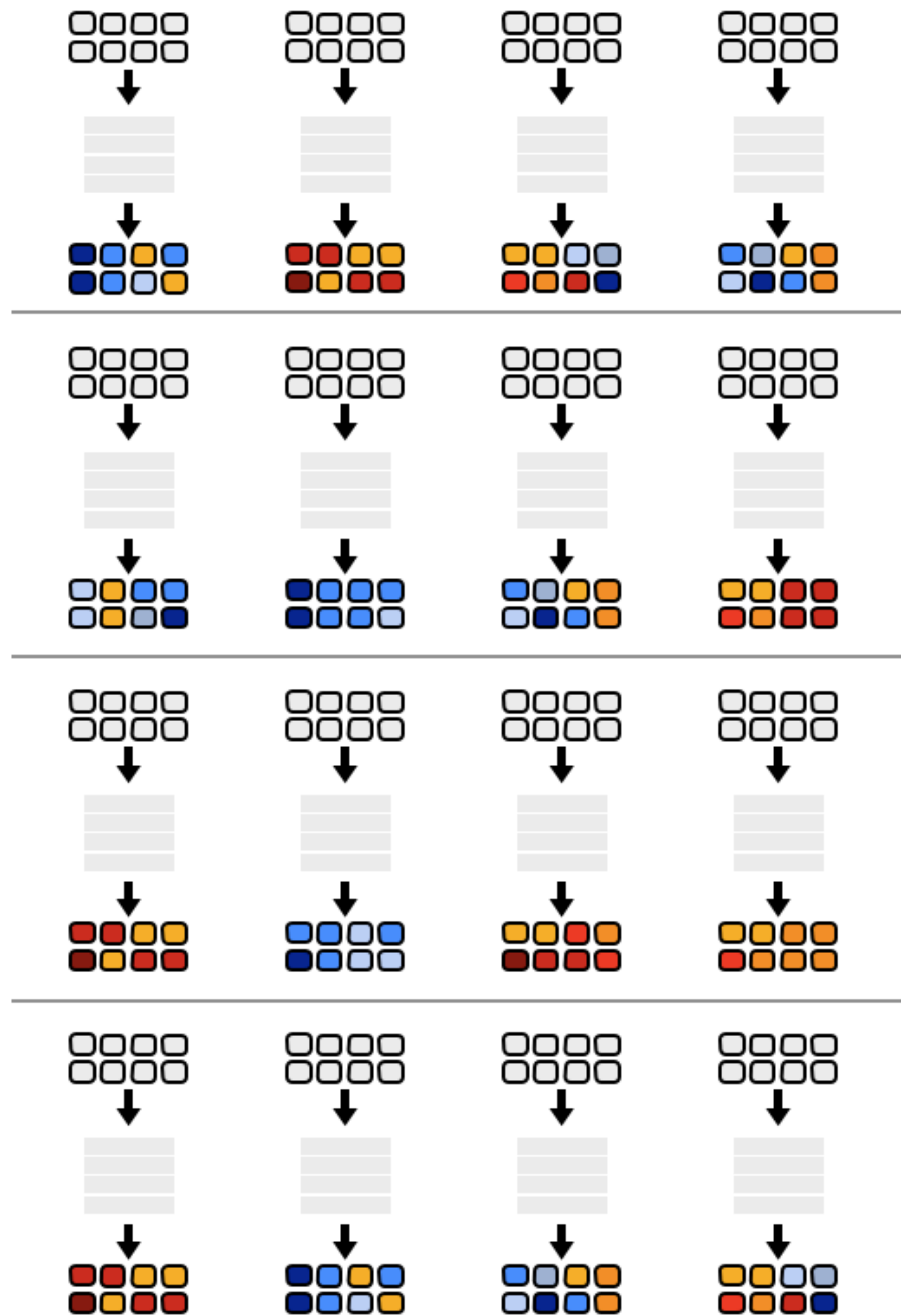


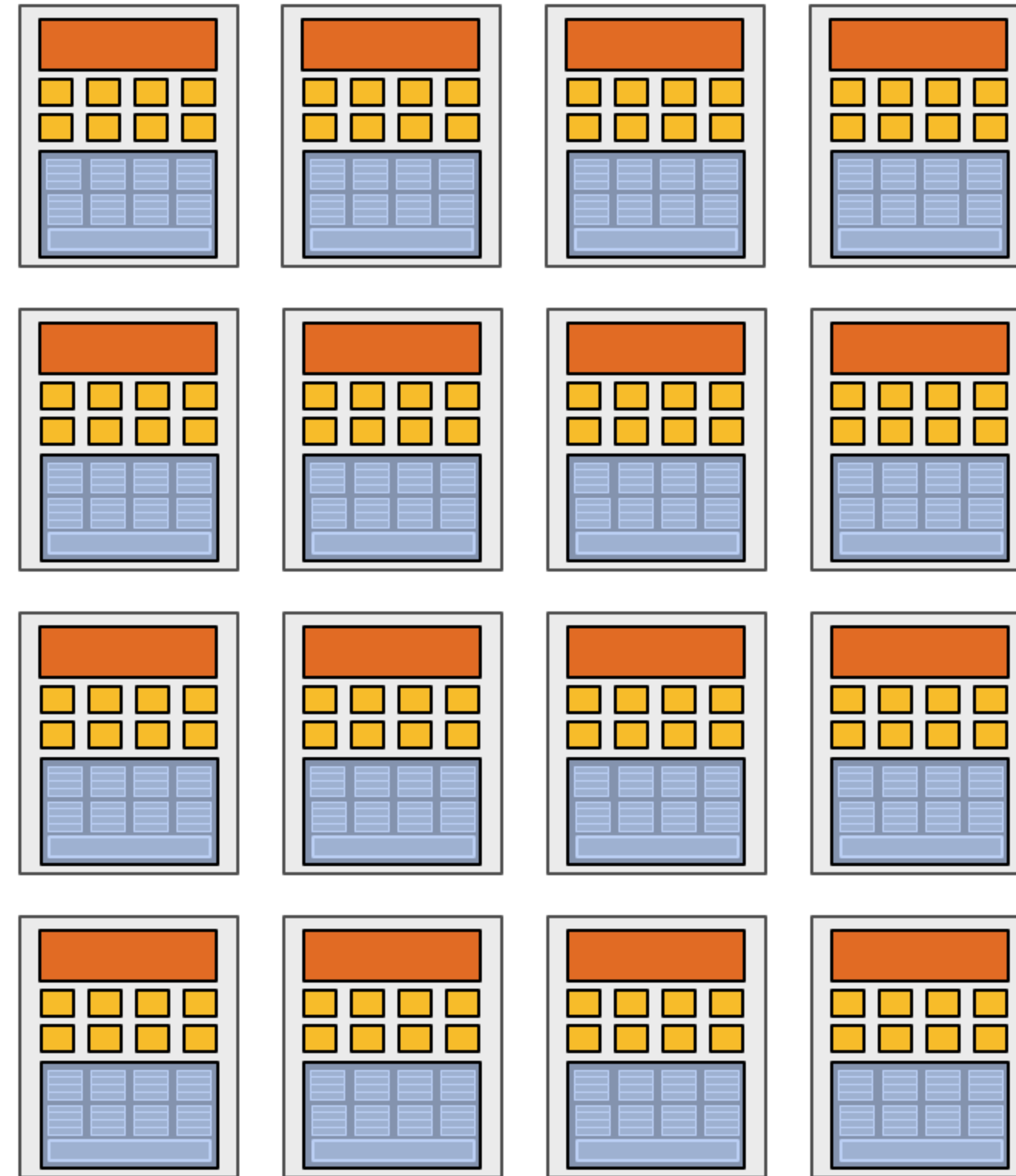
```
<VEC8_diffuseShader>:
VEC8_sample vec_r0, vec_v4, t0, vec_s0
VEC8_mul  vec_r3, vec_v0, cb0[0]
VEC8_madd vec_r3, vec_v1, cb0[1], vec_r3
VEC8_madd vec_r3, vec_v2, cb0[2], vec_r3
VEC8_clmp vec_r3, vec_r3, l(0.0), l(1.0)
VEC8_mul  vec_o0, vec_r0, vec_r3
VEC8_mul  vec_o1, vec_r1, vec_r3
VEC8_mul  vec_o2,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





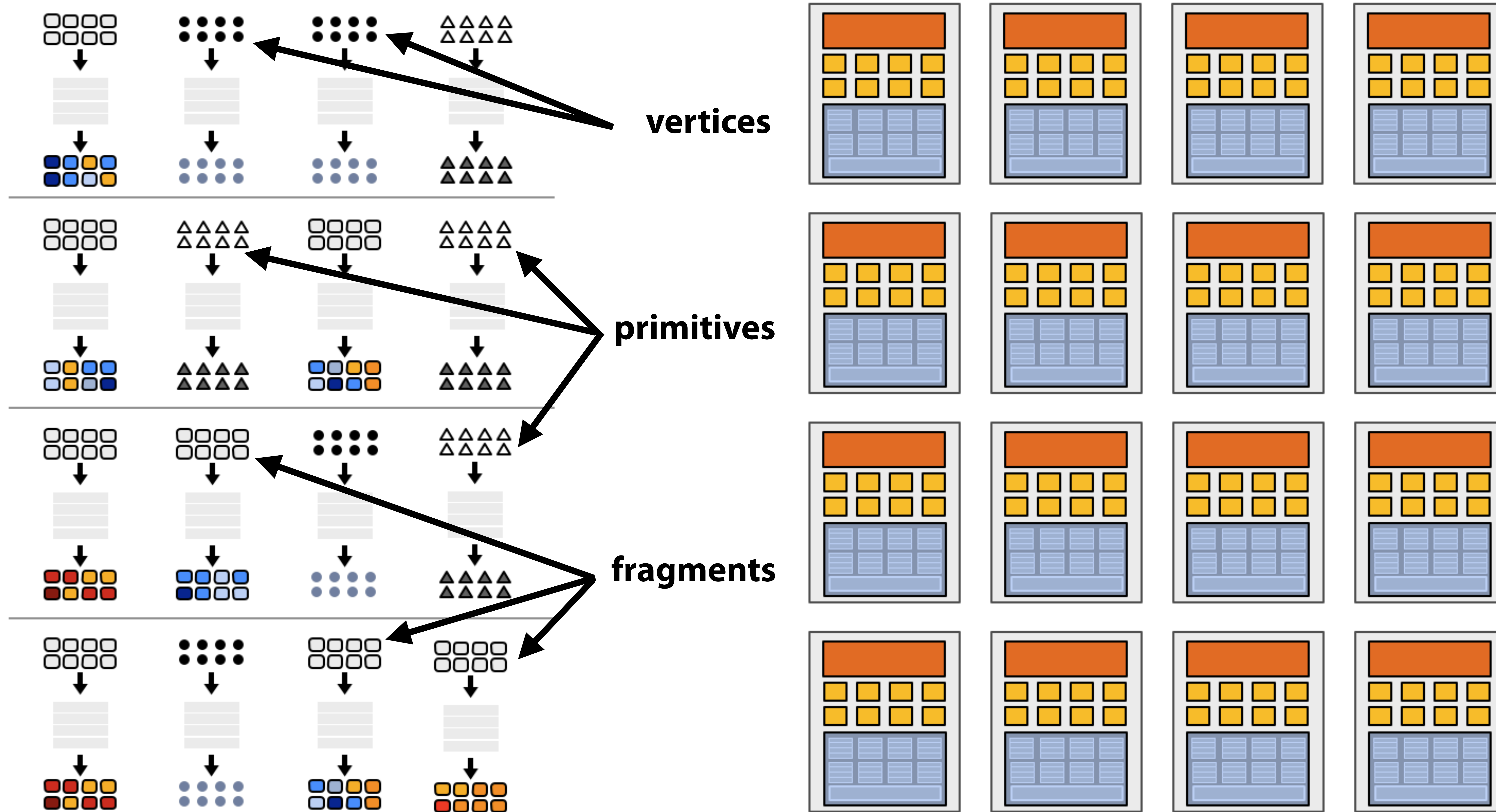
128 fragments in parallel



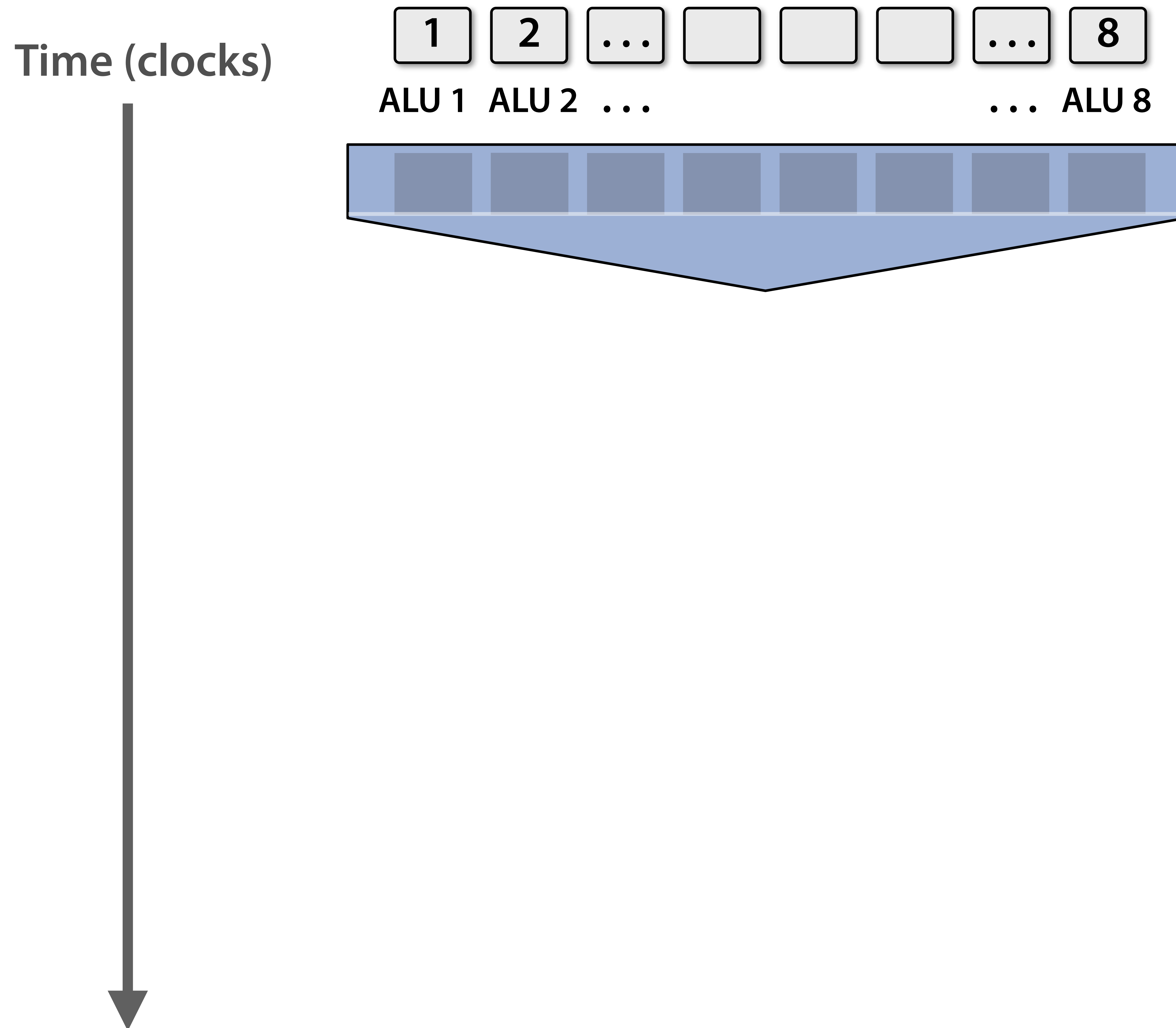


16 cores = 128 ALUs , 16 simultaneous instruction streams

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

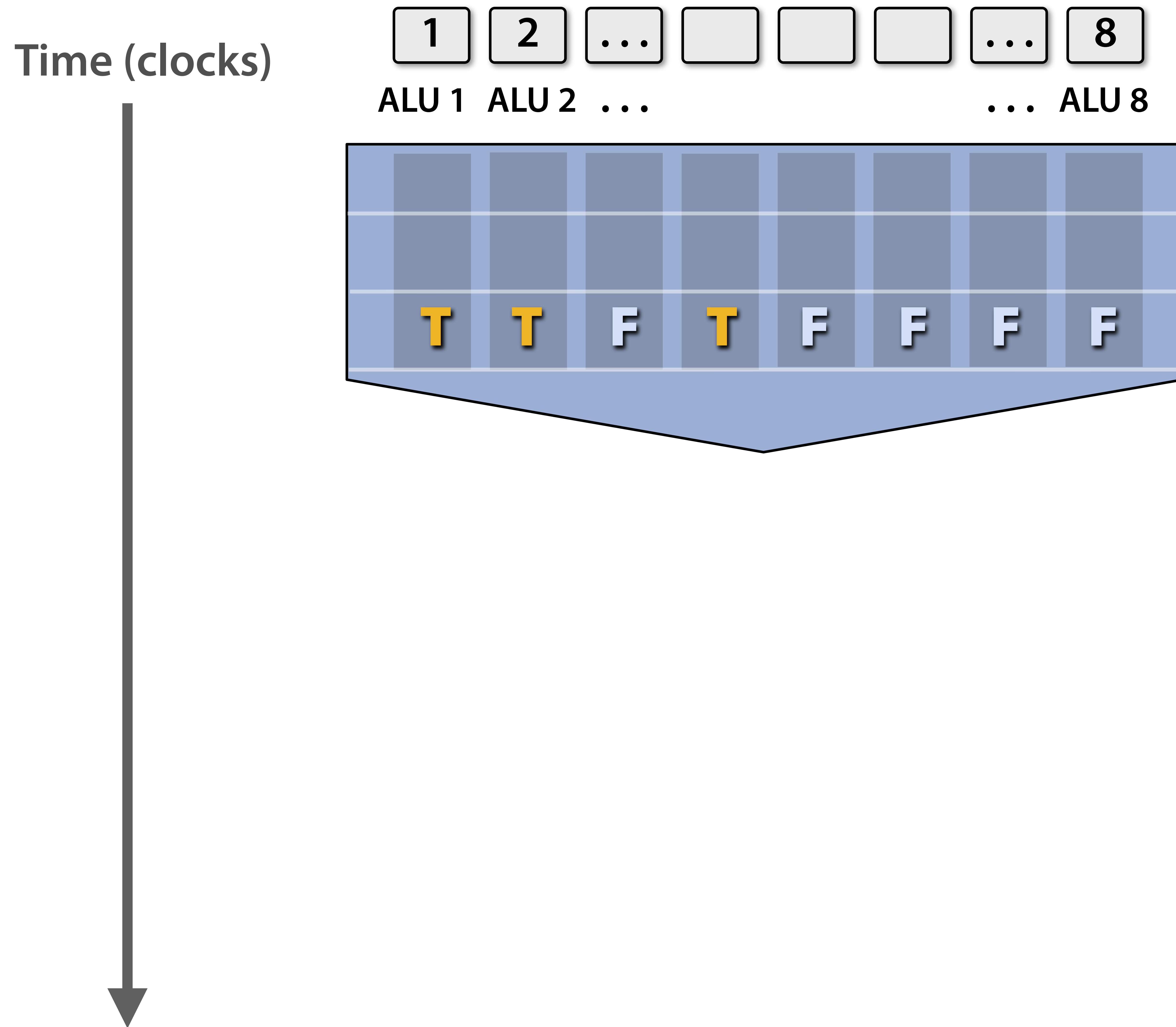


But what about branches?



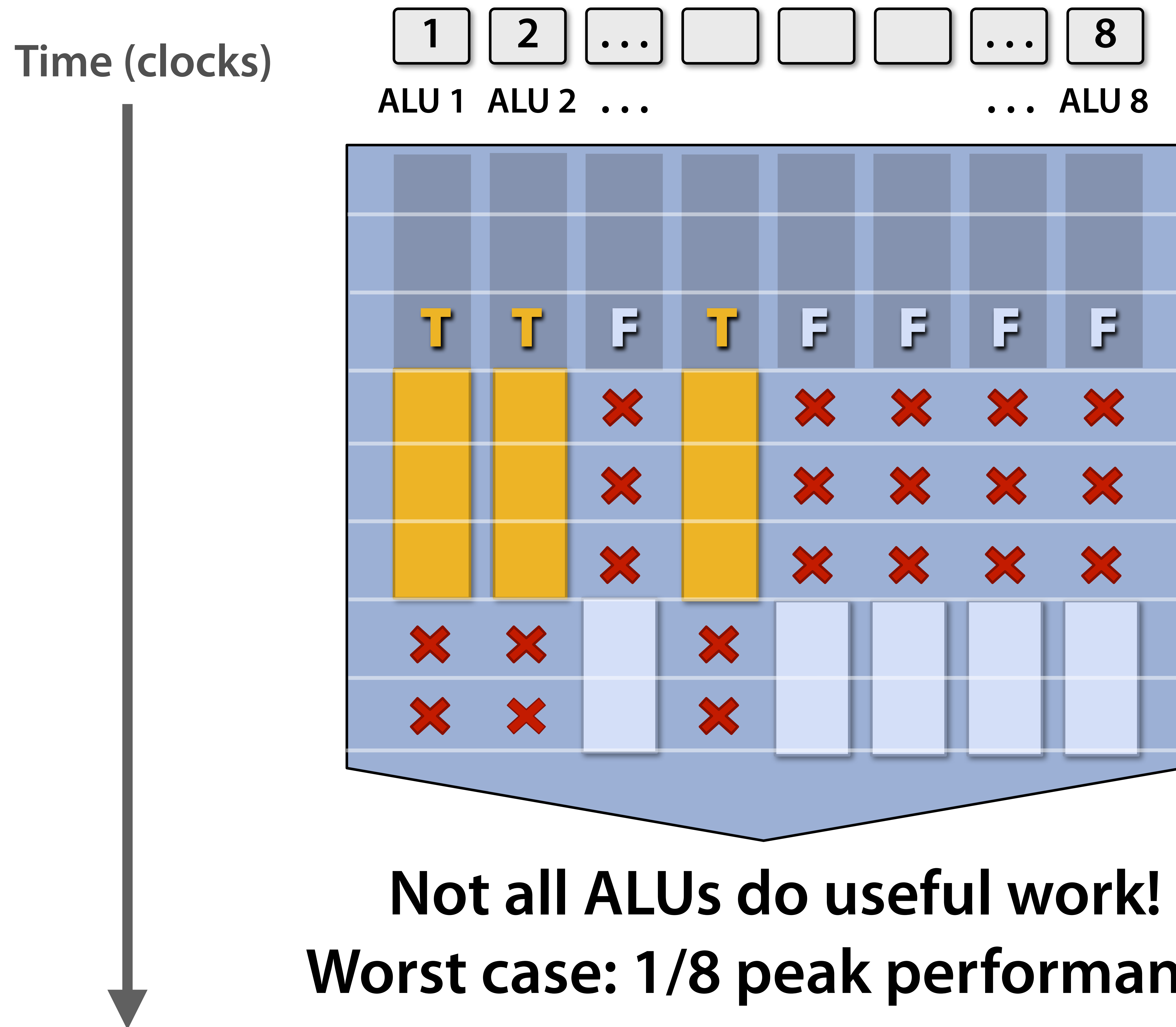

```
<unconditional
 shader code>

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?



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

```
<unconditional
 shader code>

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

<resume unconditional
 shader code>
```

But what about branches?

Time (clocks)

1 2 8

ALU 1 ALU 2 ... ... ALU 8

T T F T F F F F

```
<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)

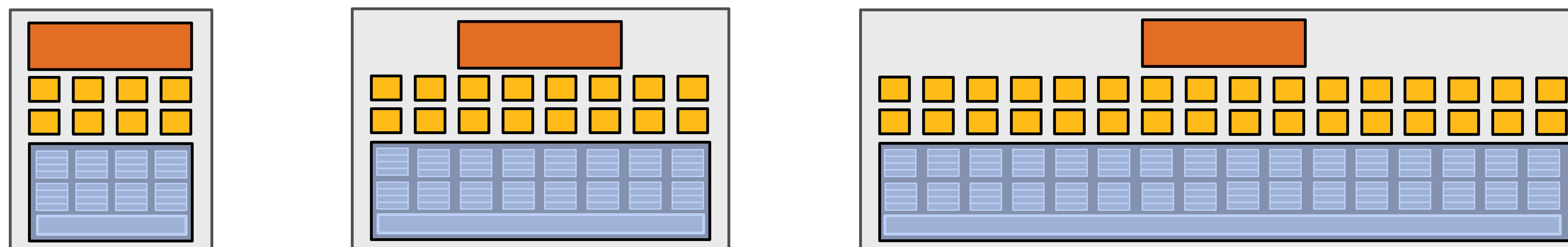
- "Divergence": lack of coherence
  - Usually used in reference to instruction streams (divergent execution does not make full use of SIMD processing)

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

Clarification

SIMD processing <u>does not</u> 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)

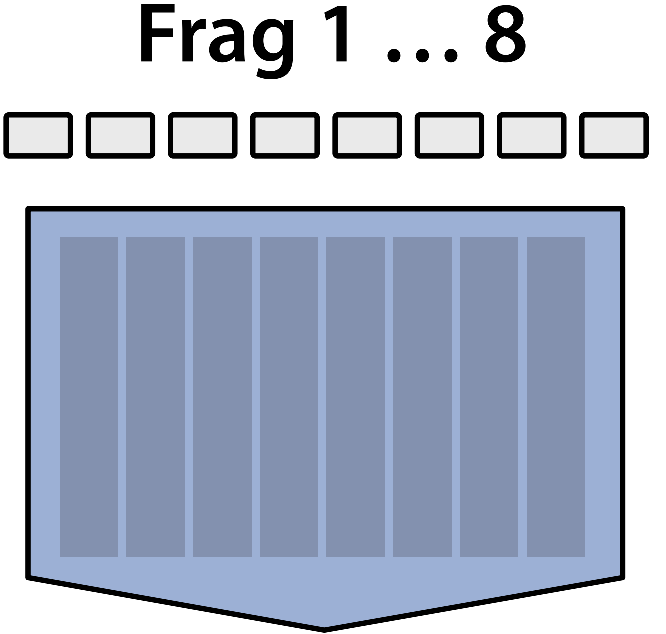
Idea #3:

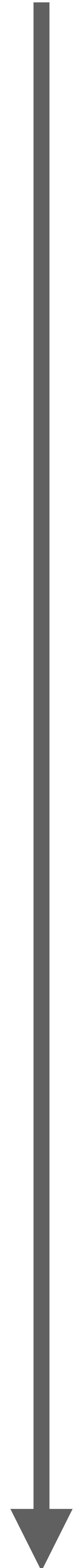
Interleave processing of many fragments on a single core to avoid stalls caused by high latency operations.

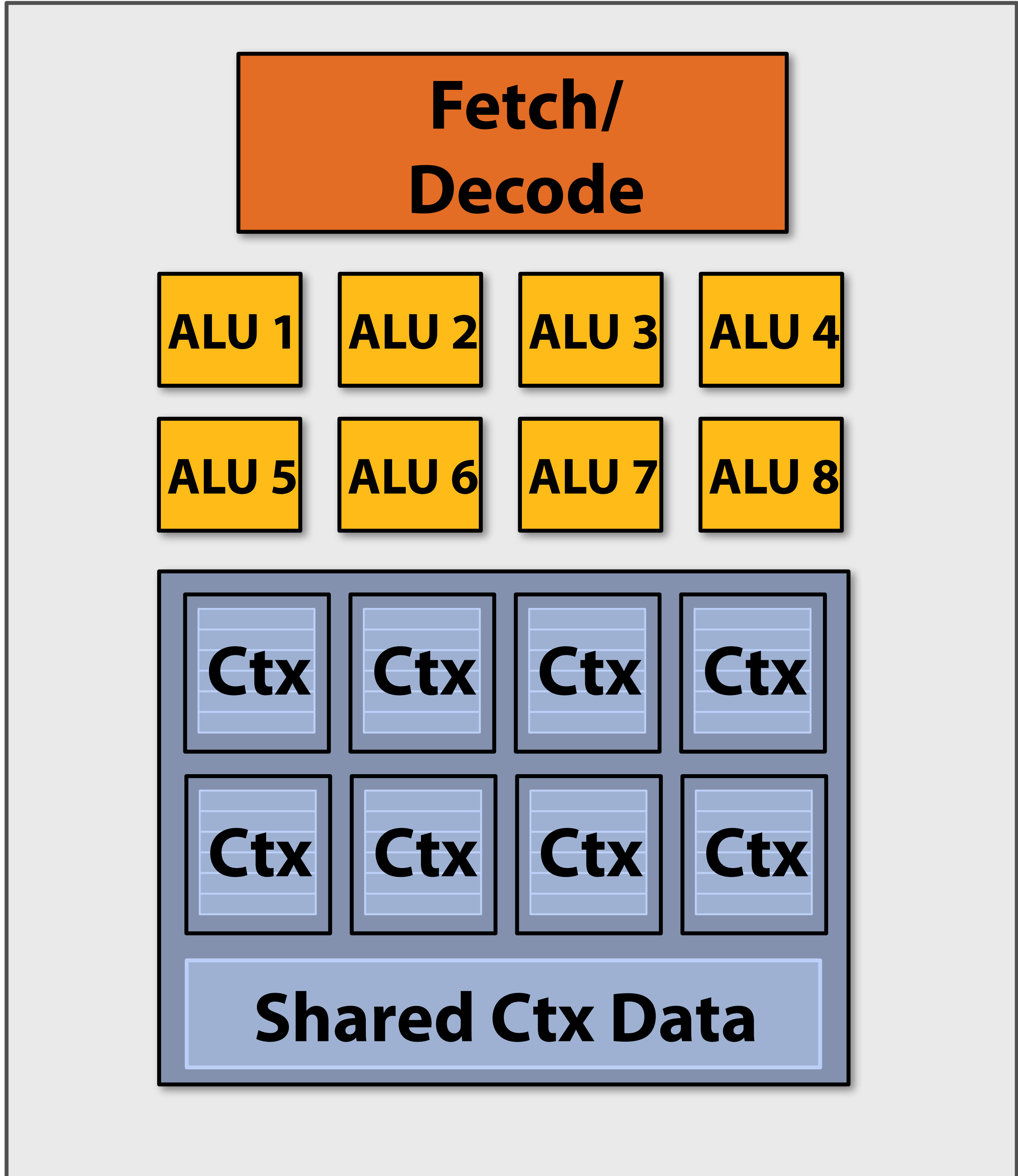
Hiding shader stalls

Time (clocks)

Frag 1 … 8

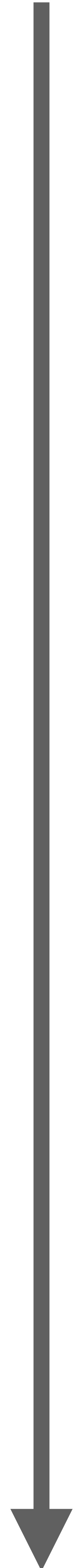




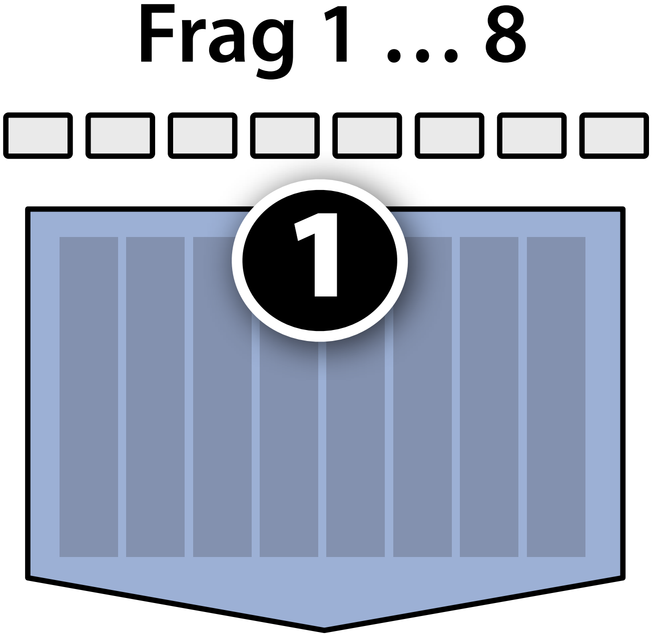


Hiding shader stalls

Time (clocks)



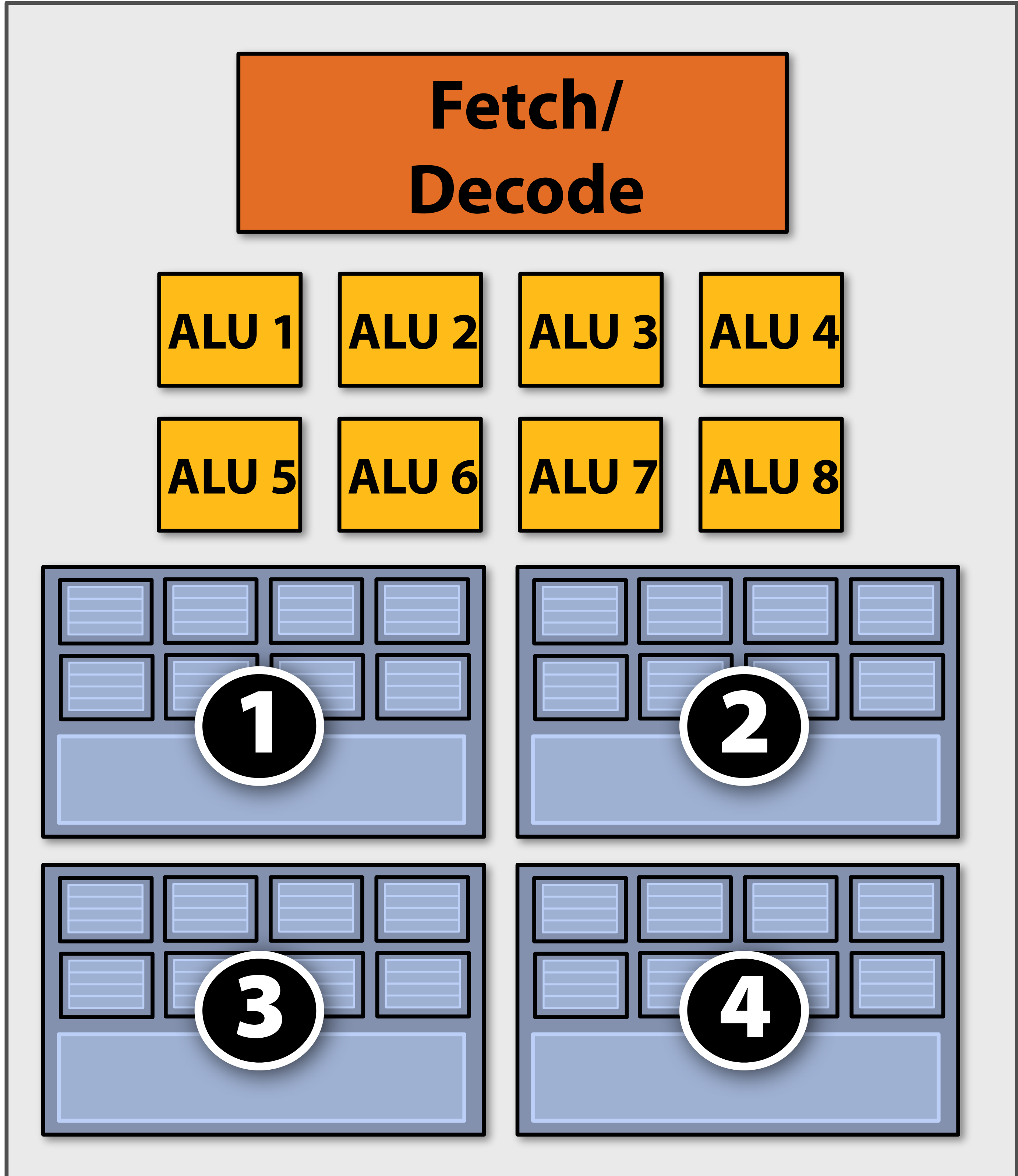
Frag 1 … 8



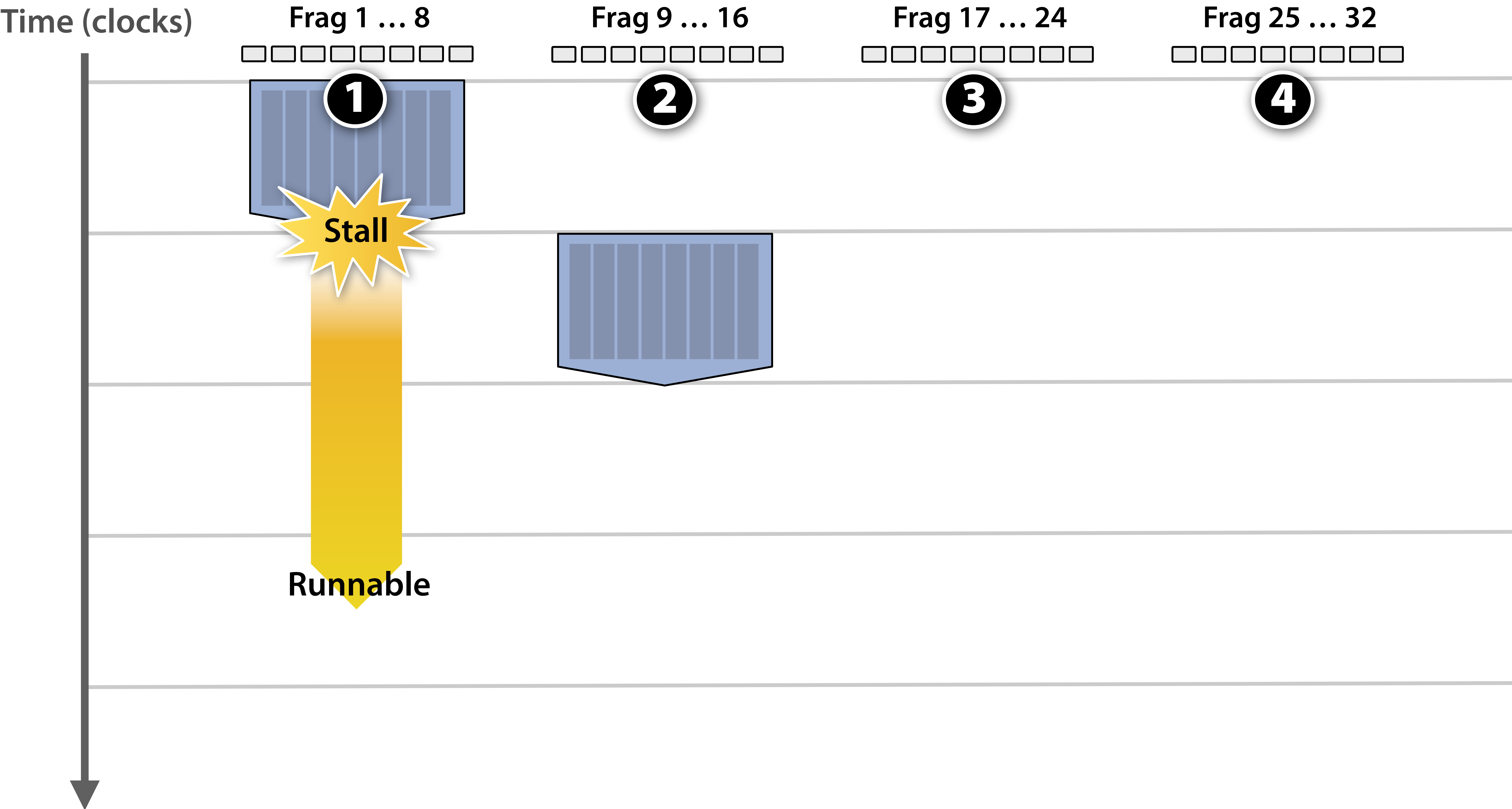
Frag 9 … 16

Frag 17 … 24

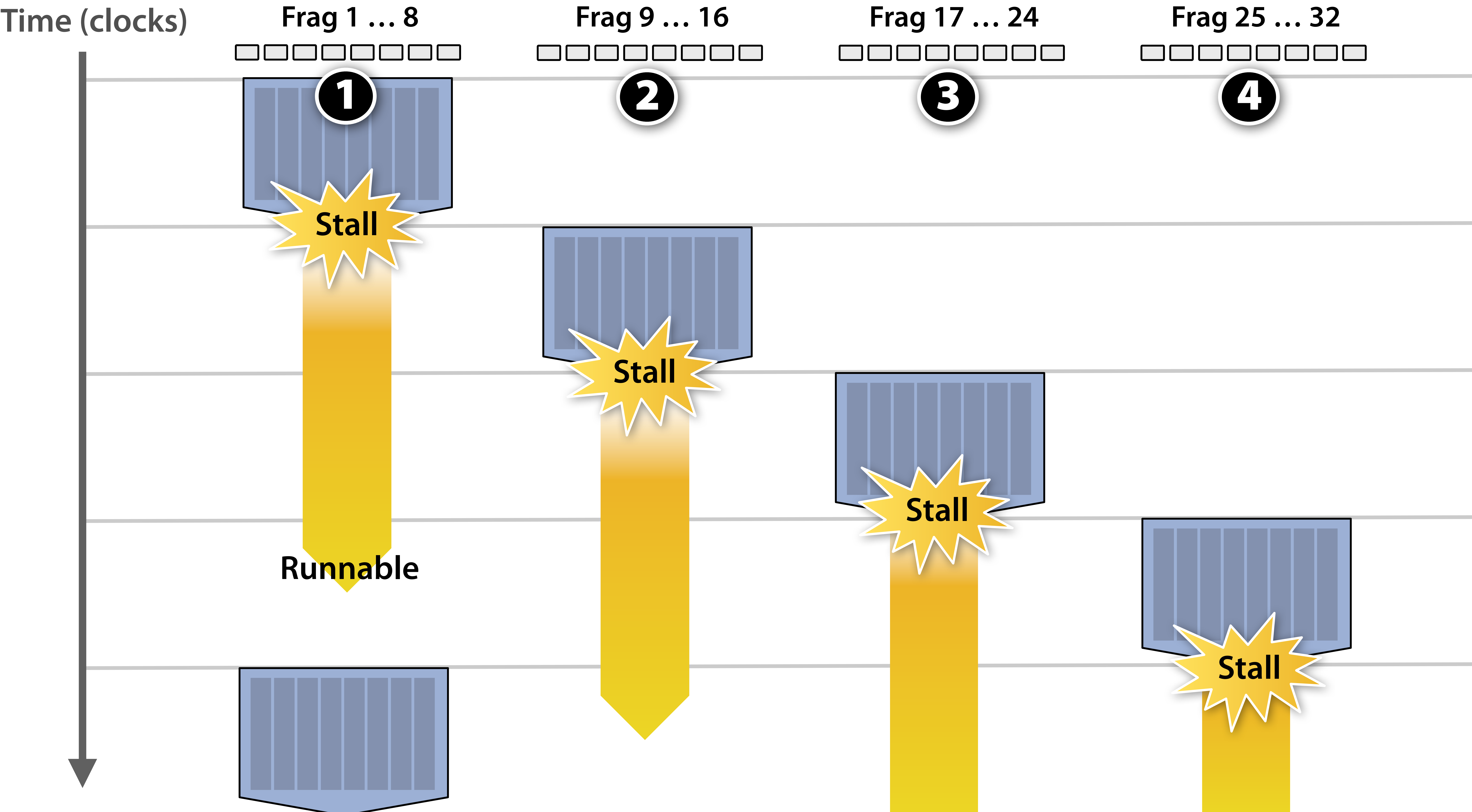
Frag 25 … 32



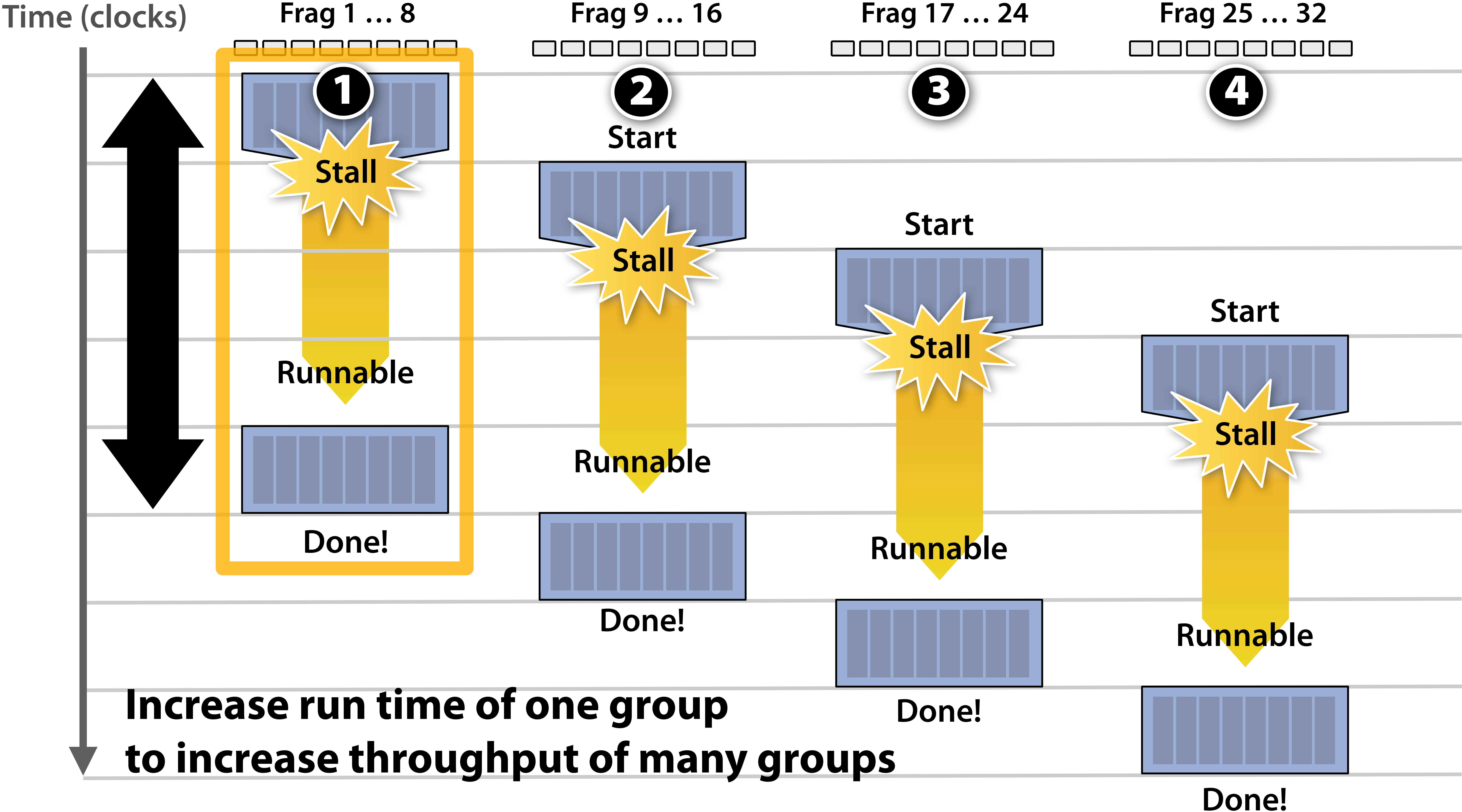
Hiding shader stalls



Hiding shader stalls

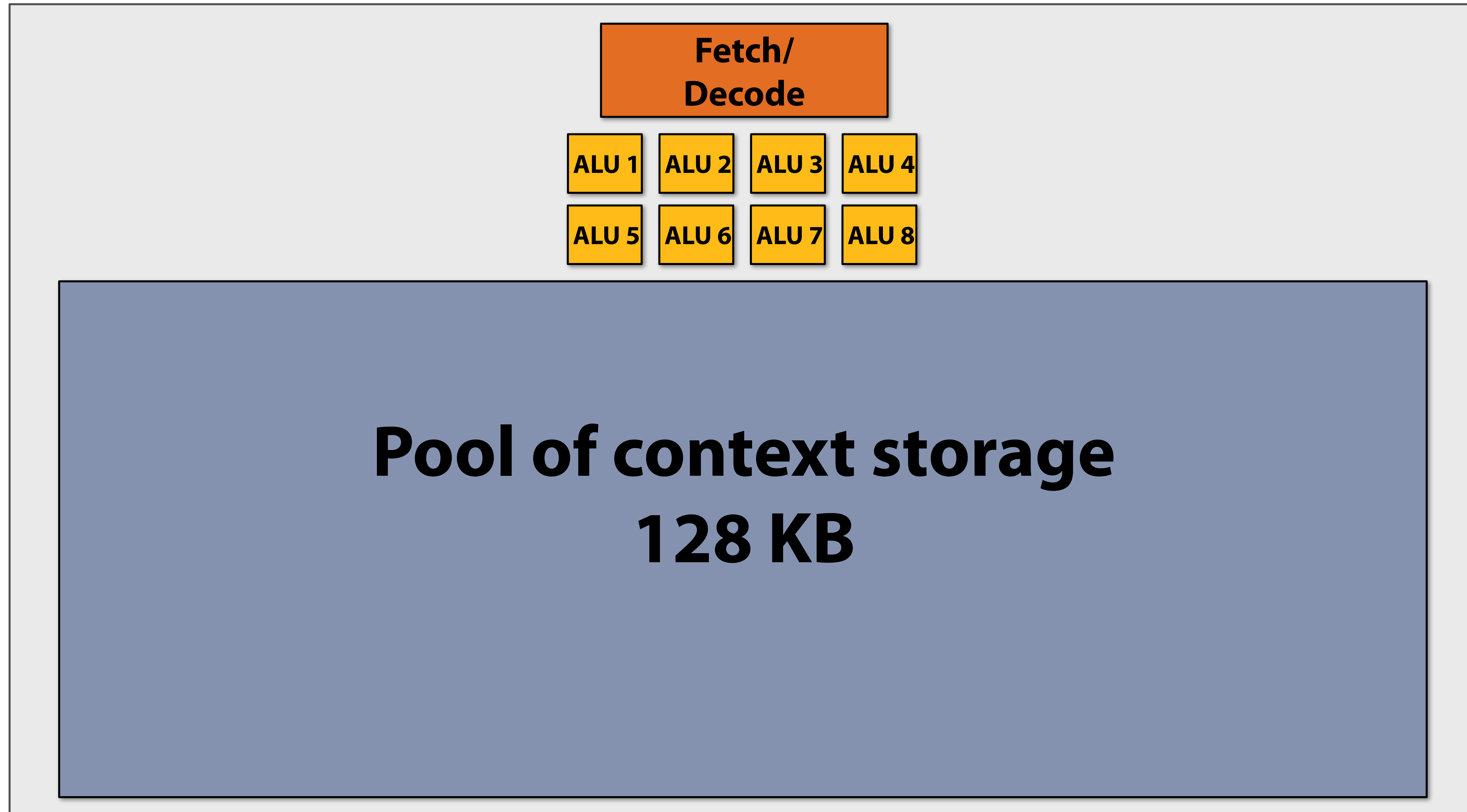


Throughput!



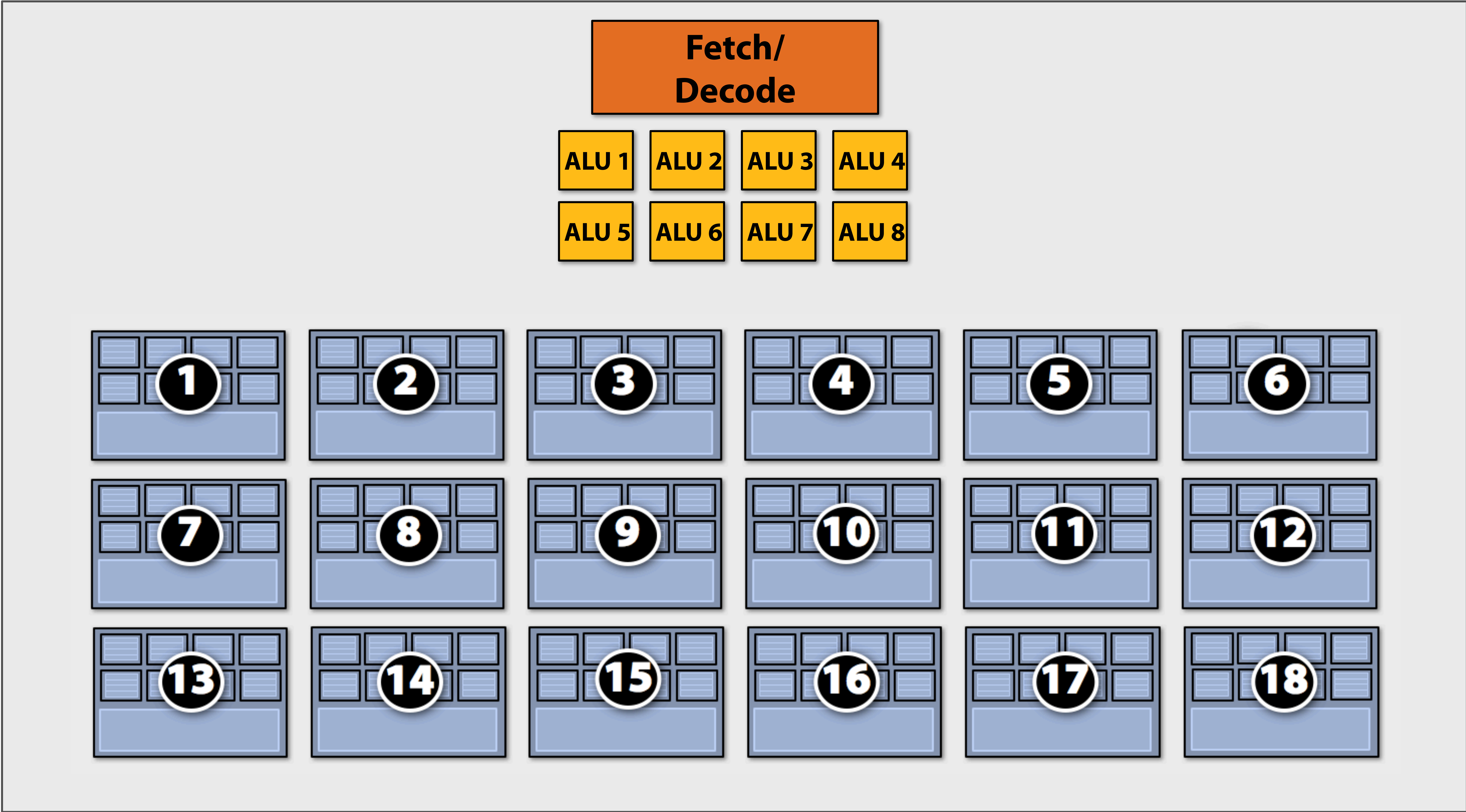
Increase run time of one group to increase throughput of many groups

Storing contexts

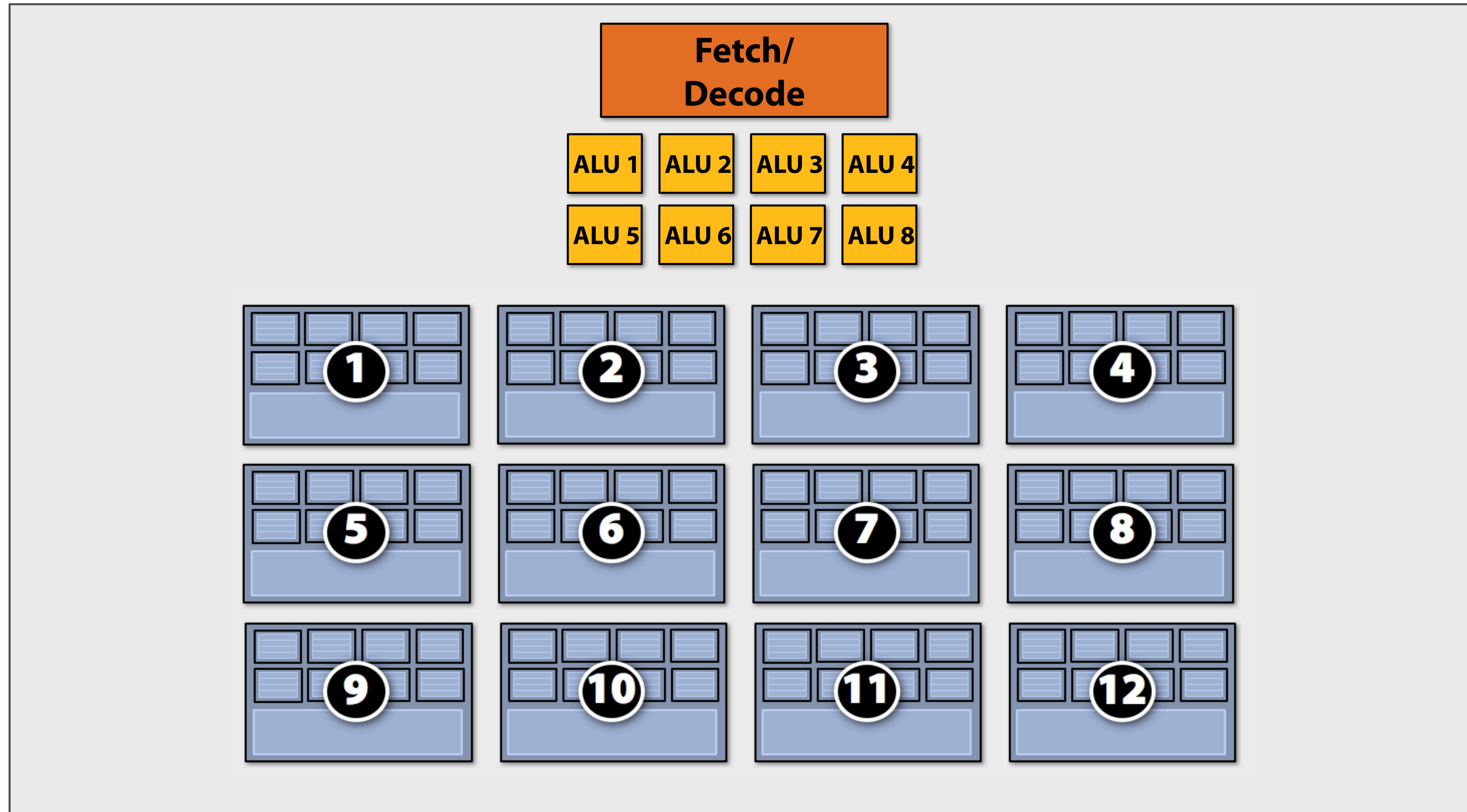


Eighteen small contexts

(maximal latency hiding)

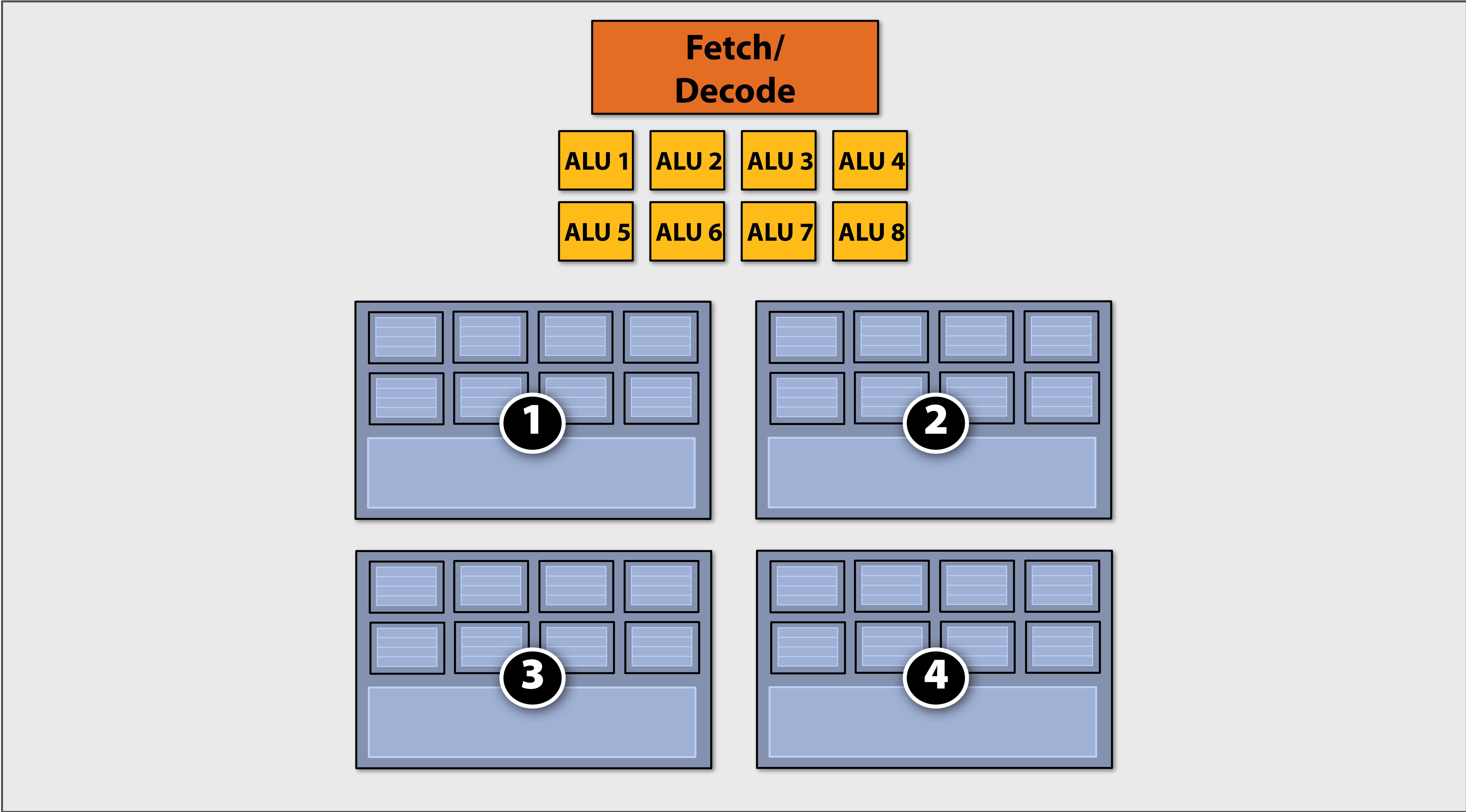


Twelve medium contexts



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)

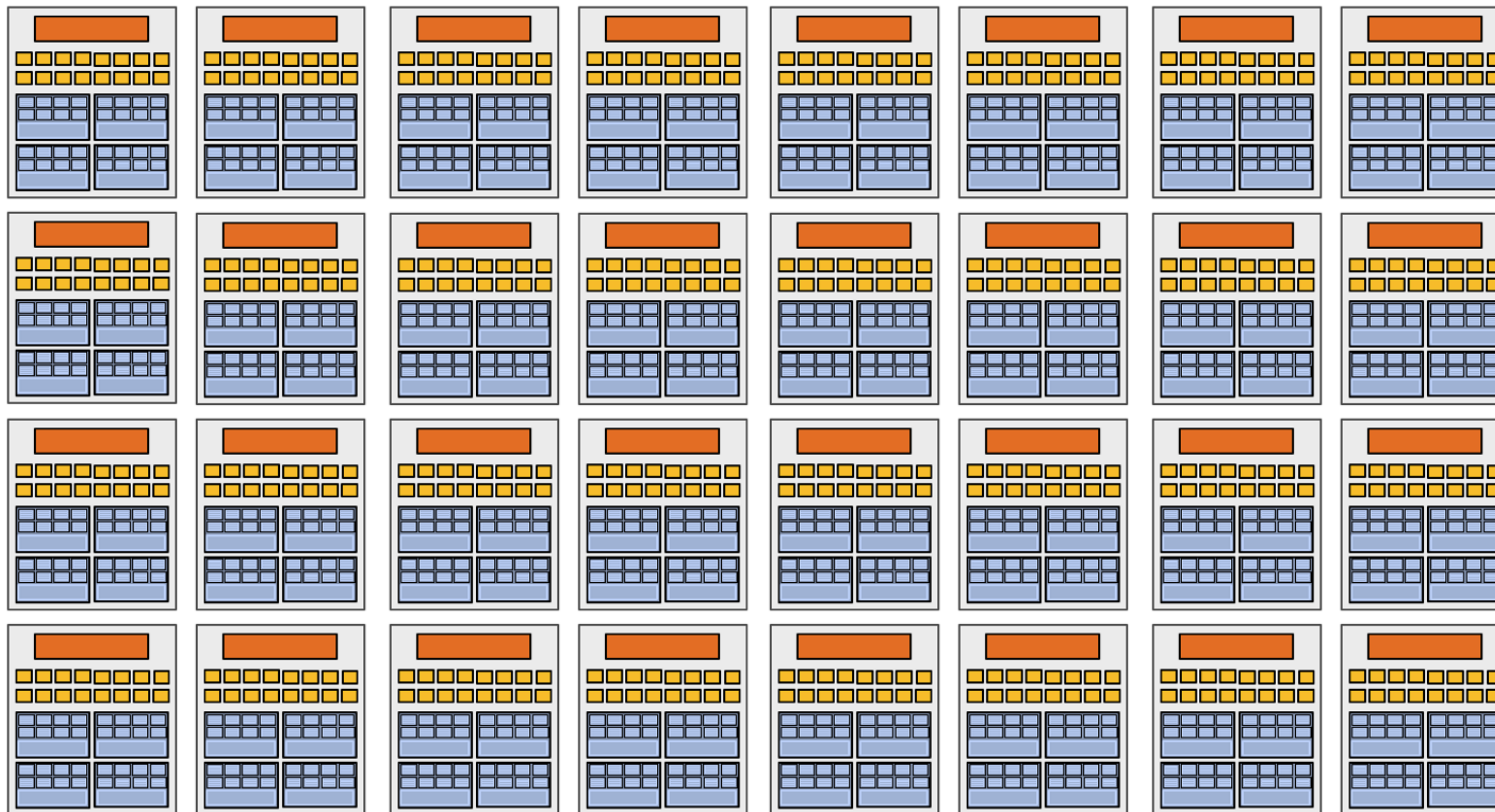
16 simultaneous
instruction streams

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

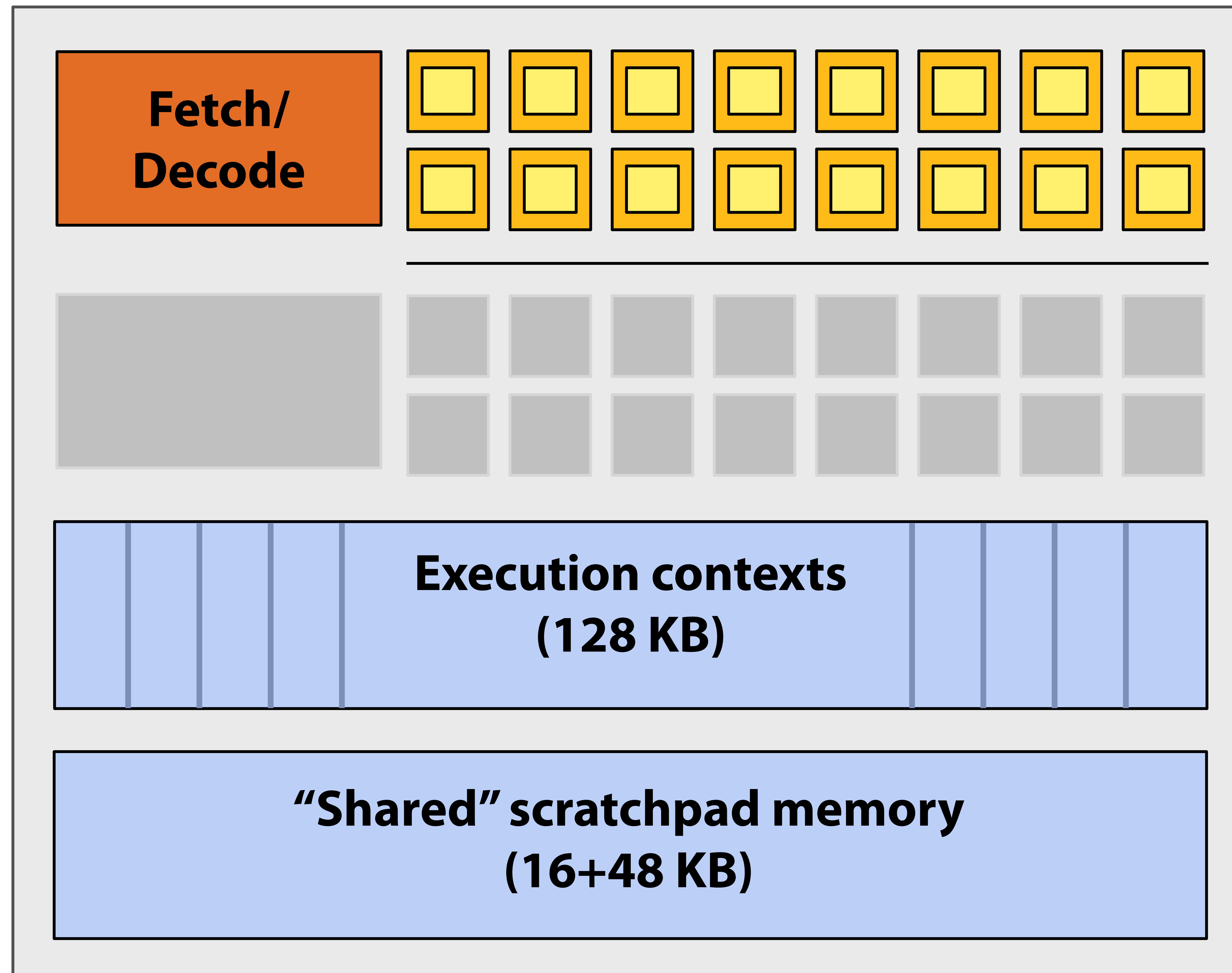
NVIDIA GeForce GTX 480
ATI Radeon HD 5870

NVIDIA GeForce GTX 480 (Fermi)

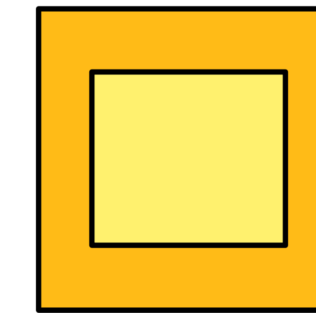
- **NVIDIA-speak:**
  - **480 stream processors ("CUDA cores")**
  - **"SIMT execution"**

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

NVIDIA GeForce GTX 480 “core”



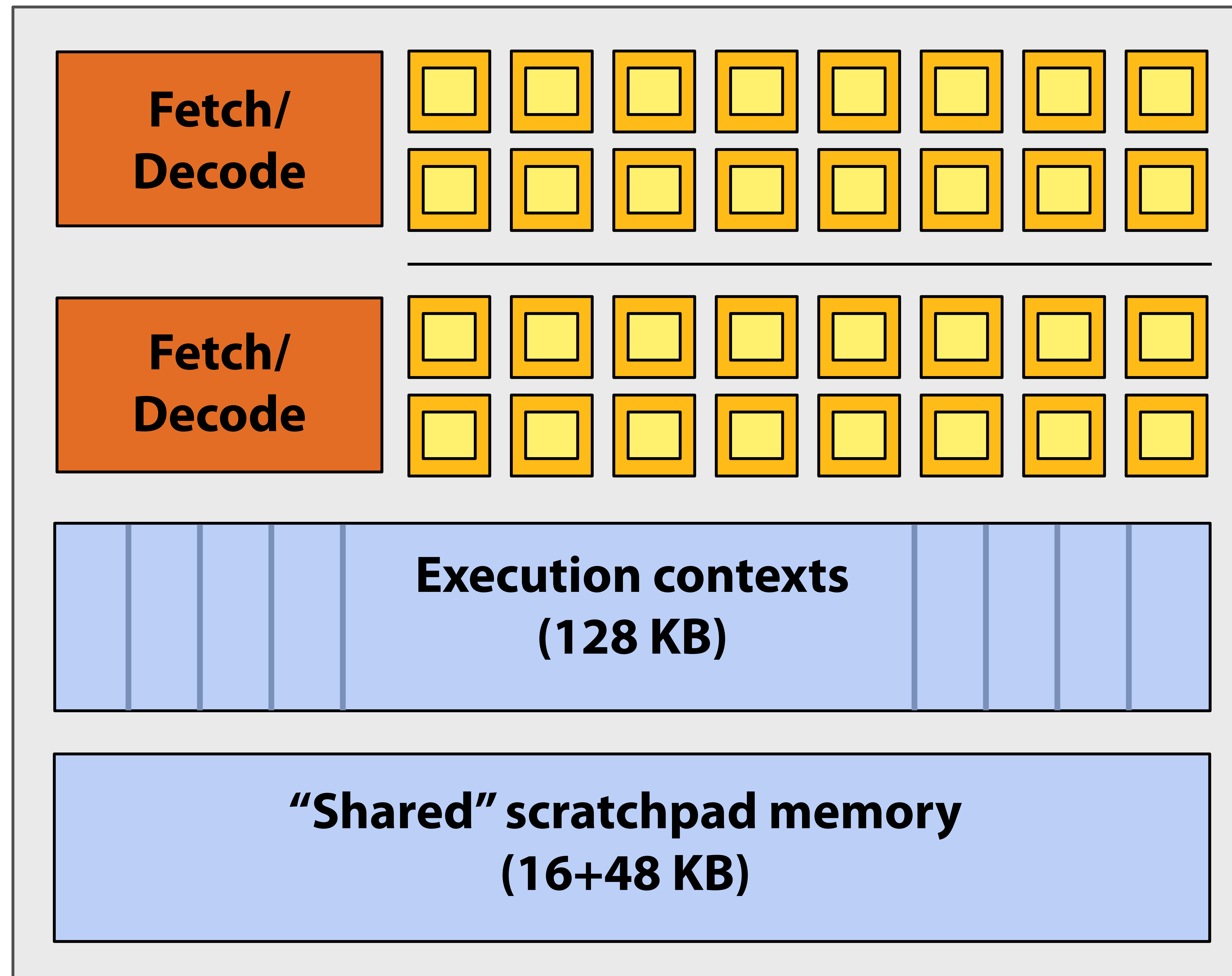
Source: Fermi Compute Architecture Whitepaper
CUDA Programming Guide 3.1, Appendix G



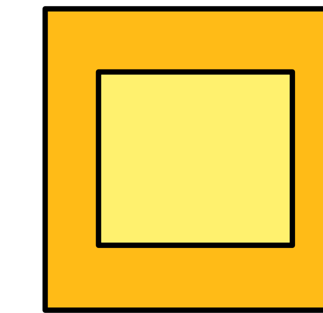
= SIMD function unit, control shared across 16 units (1 MUL-ADD per clock)

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

NVIDIA GeForce GTX 480 “core”



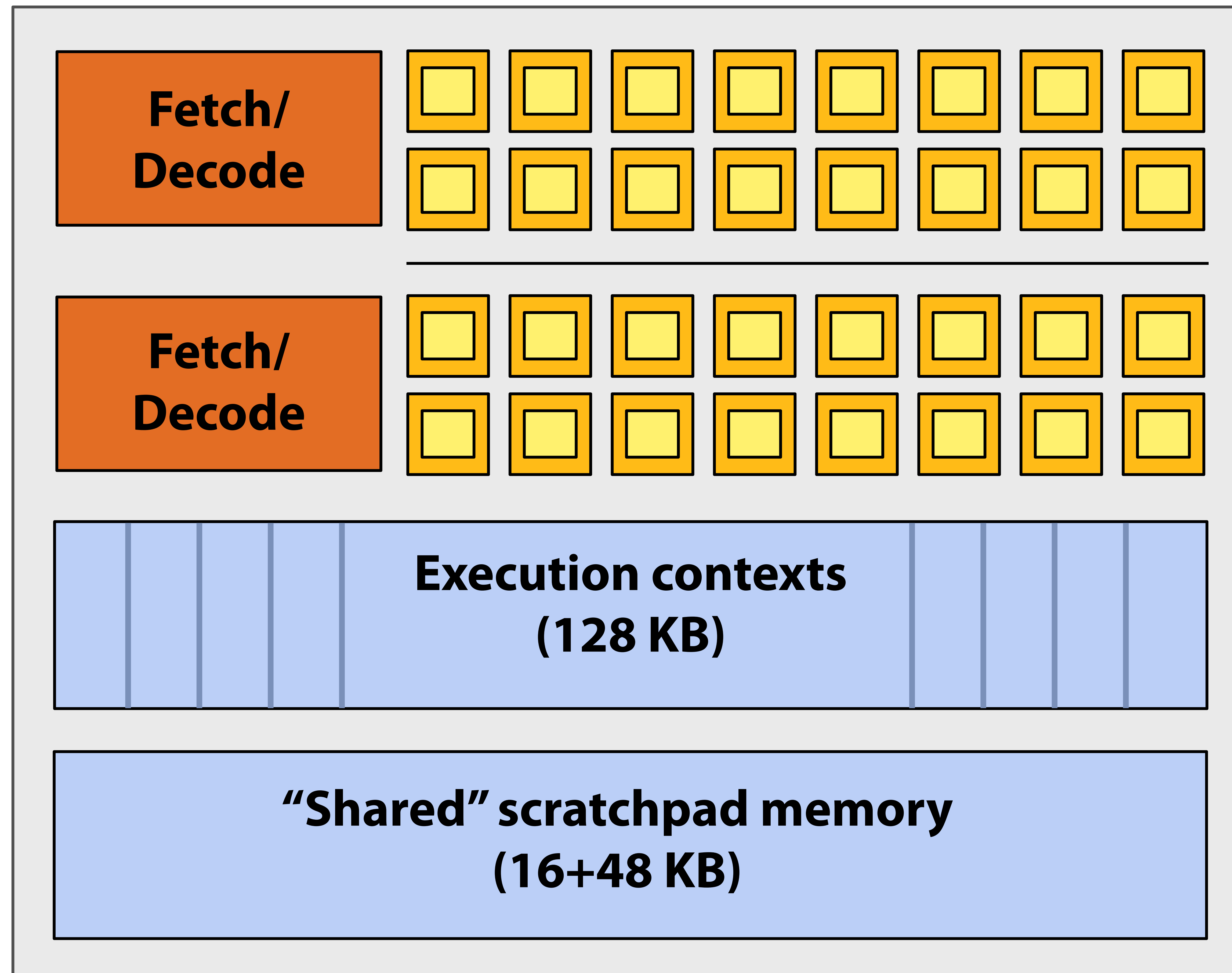
Source: Fermi Compute Architecture Whitepaper
CUDA Programming Guide 3.1, Appendix G



= 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)

NVIDIA GeForce GTX 480 "SM"

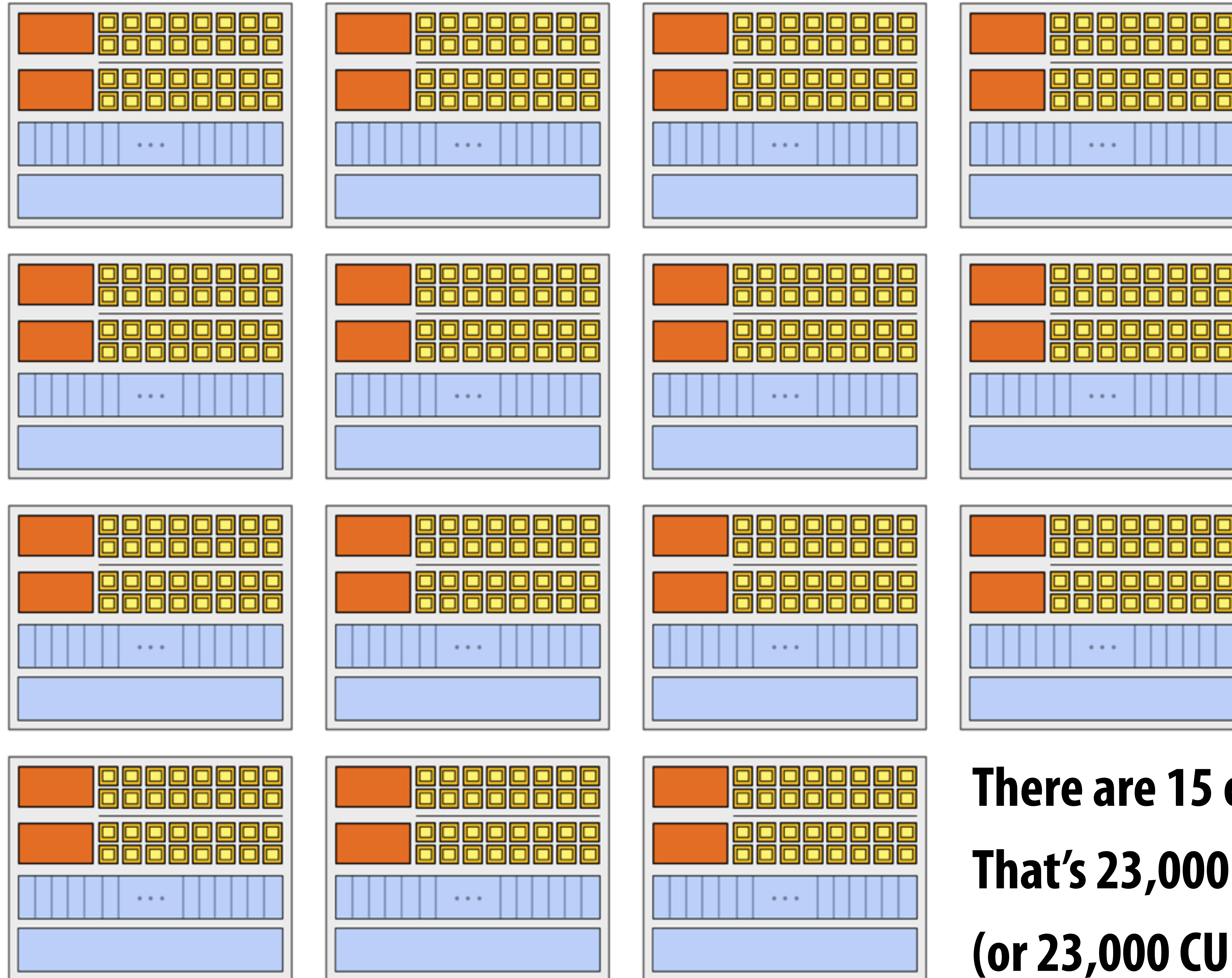


Source: Fermi Compute Architecture Whitepaper
CUDA Programming Guide 3.1, Appendix G

= 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

NVIDIA GeForce GTX 480



There are 15 of these things on the GTX 480:

That's 23,000 fragments!

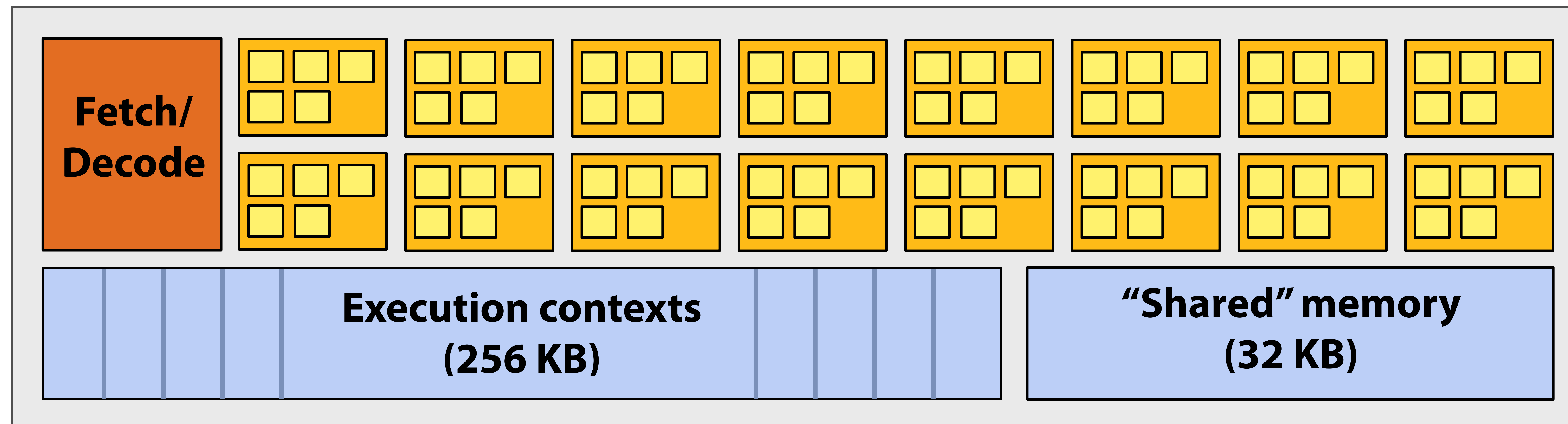
(or 23,000 CUDA threads!)

AMD Radeon HD 5870 (Cypress)

- **AMD-speak:**
  - 1600 stream processors

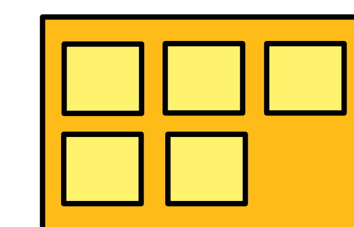
- **Generic speak:**
  - 20 cores
  - 16 "beefy" SIMD functional units per core
  - 5 multiply-adds per functional unit (VLIW processing)

AMD Radeon HD 5870 “core”



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

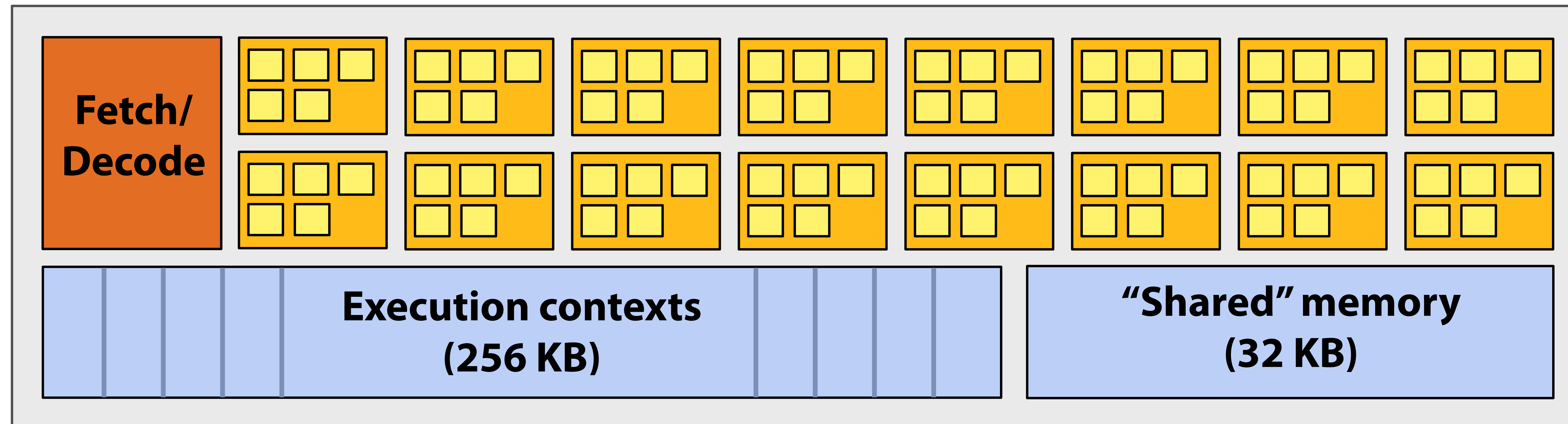
Four clocks to execute an instruction for all fragments in a group



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

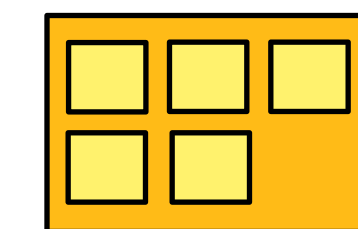
Source: ATI Radeon HD5000 Series: An Inside View (HPG 2010)

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

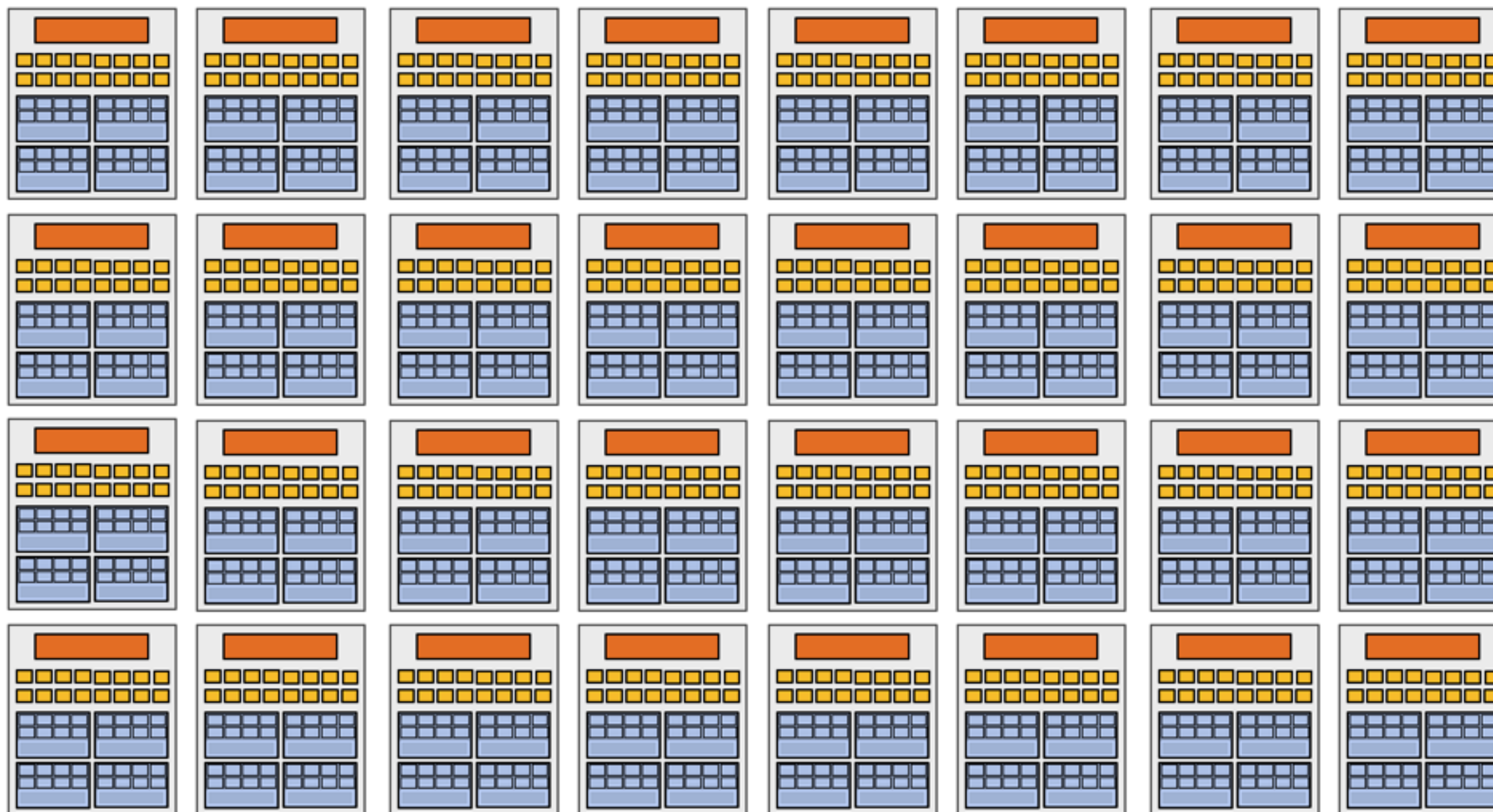
There are 20 of these "cores" on the 5870: that's about 31,000 fragments!

Lecture 8:
The GPU Memory Hierarchy

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

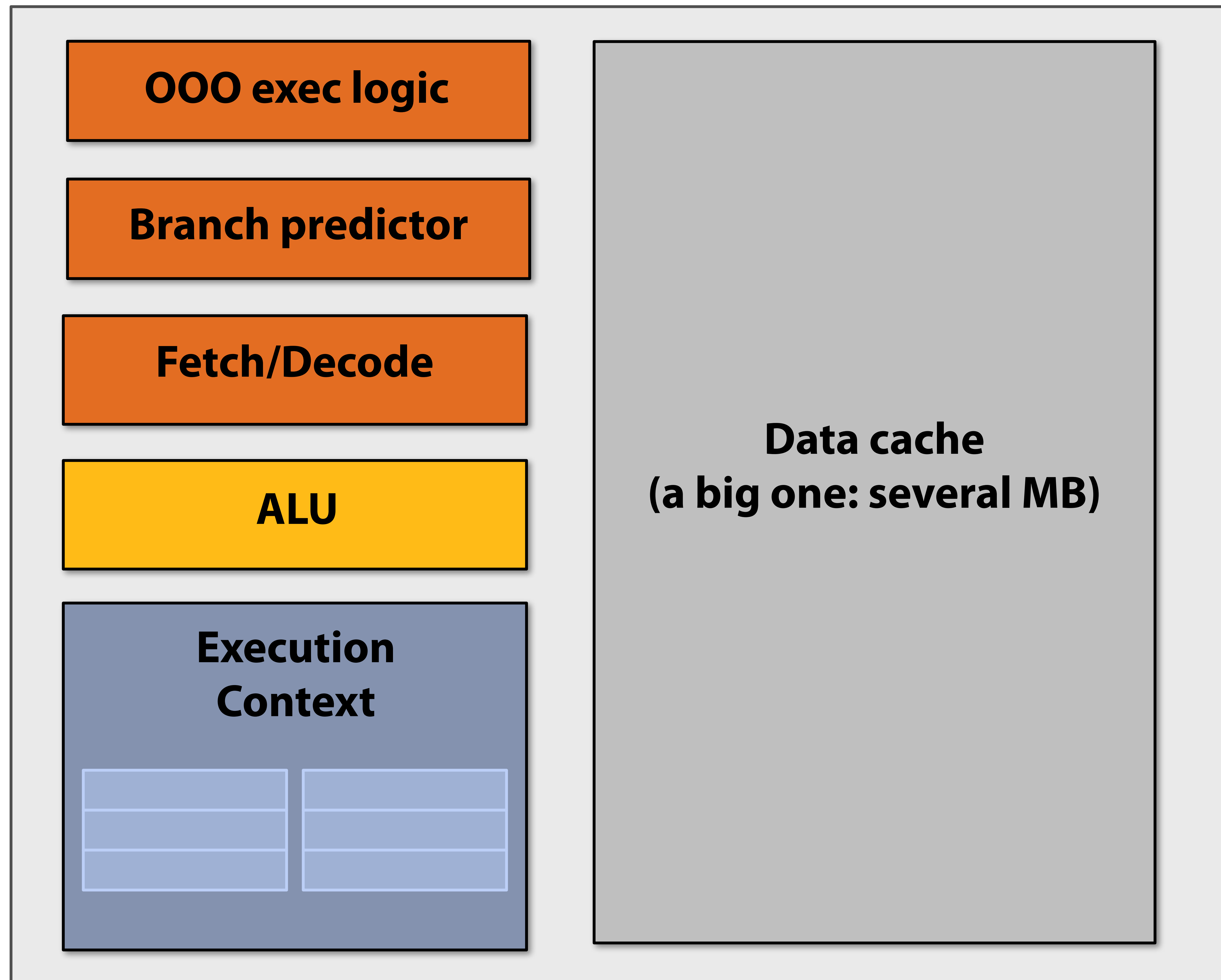
GPUs contain a collection of programmable processing cores: responsible for carrying out data-parallel stages of the graphics pipeline (vertex, fragment, primitive processing)

- **Many processing cores**
- **SIMD execution**
- **Hardware support for large-scale multi-threading**

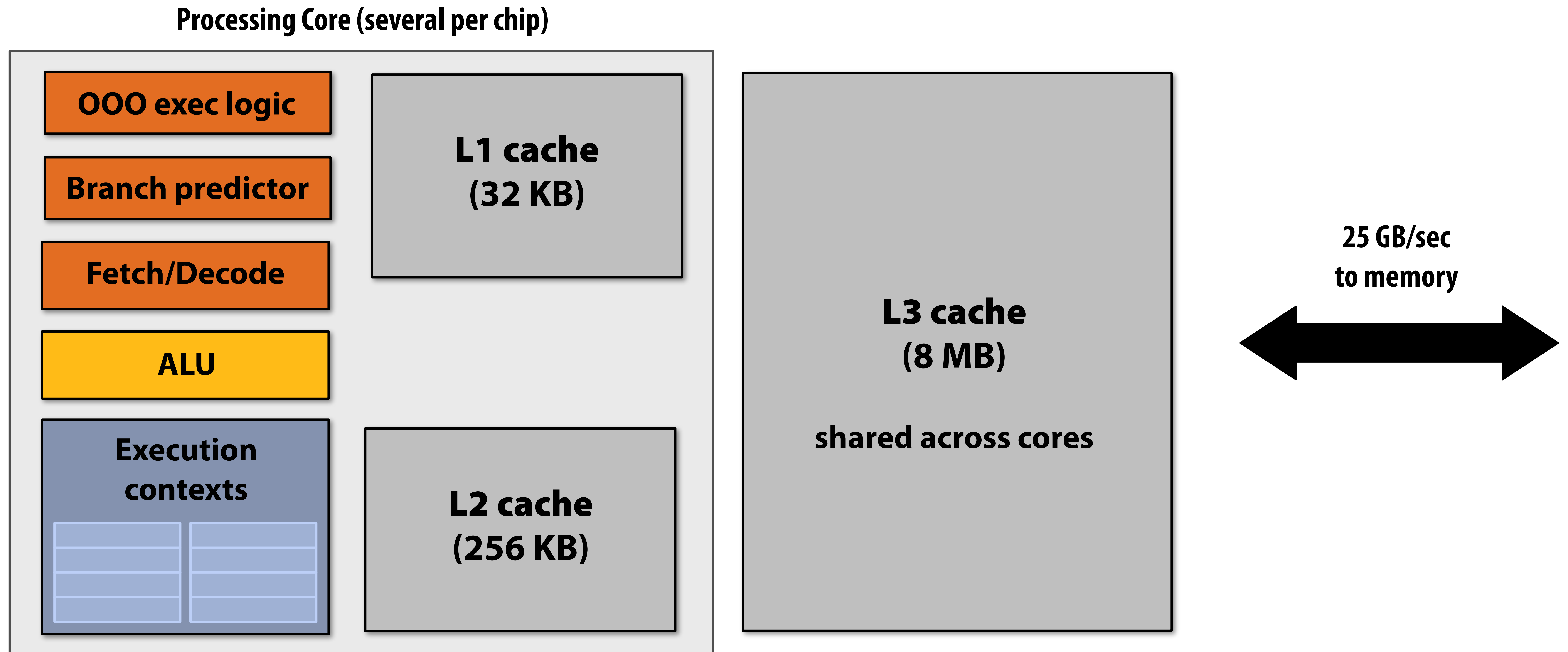


moving data to processors

Recall: "CPU-style" core

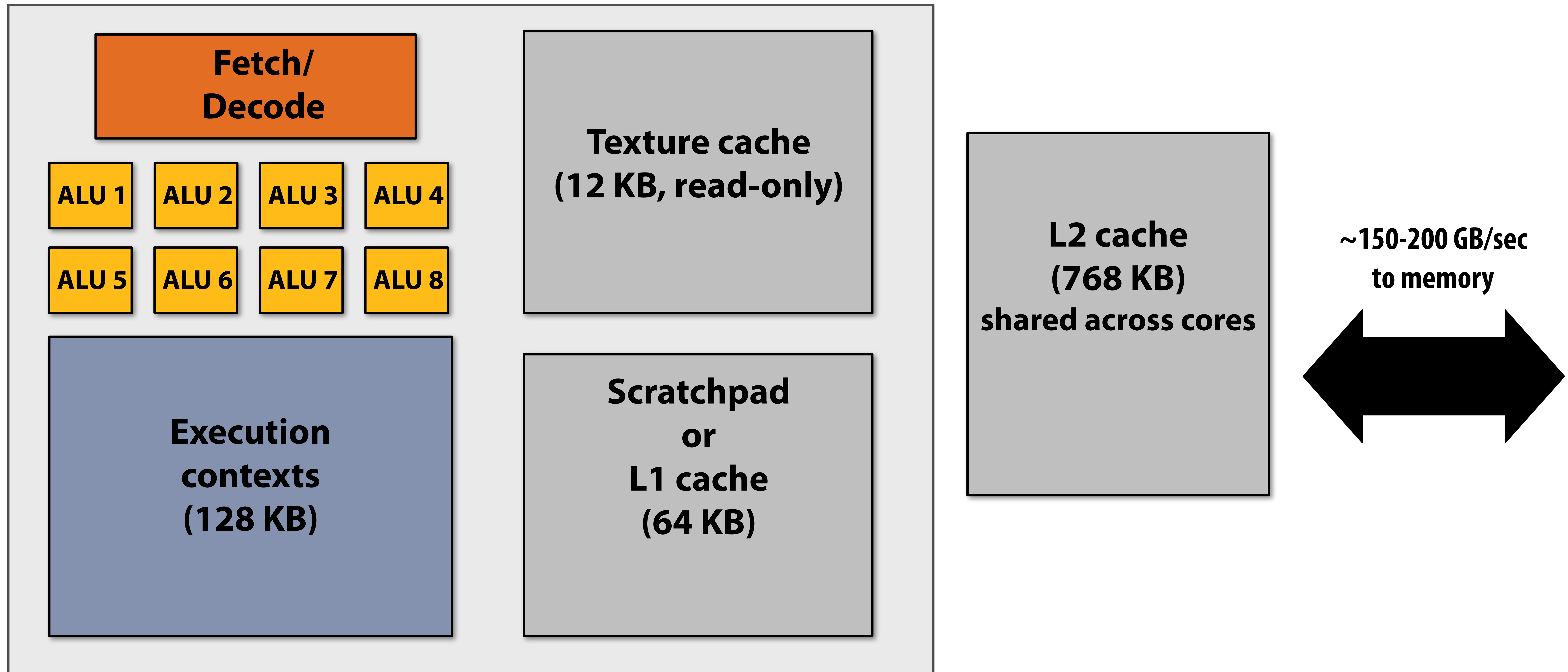


"CPU-style" memory hierarchy



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

"GPU-style" memory hierarchy (data from NVIDIA GF100: "Fermi")



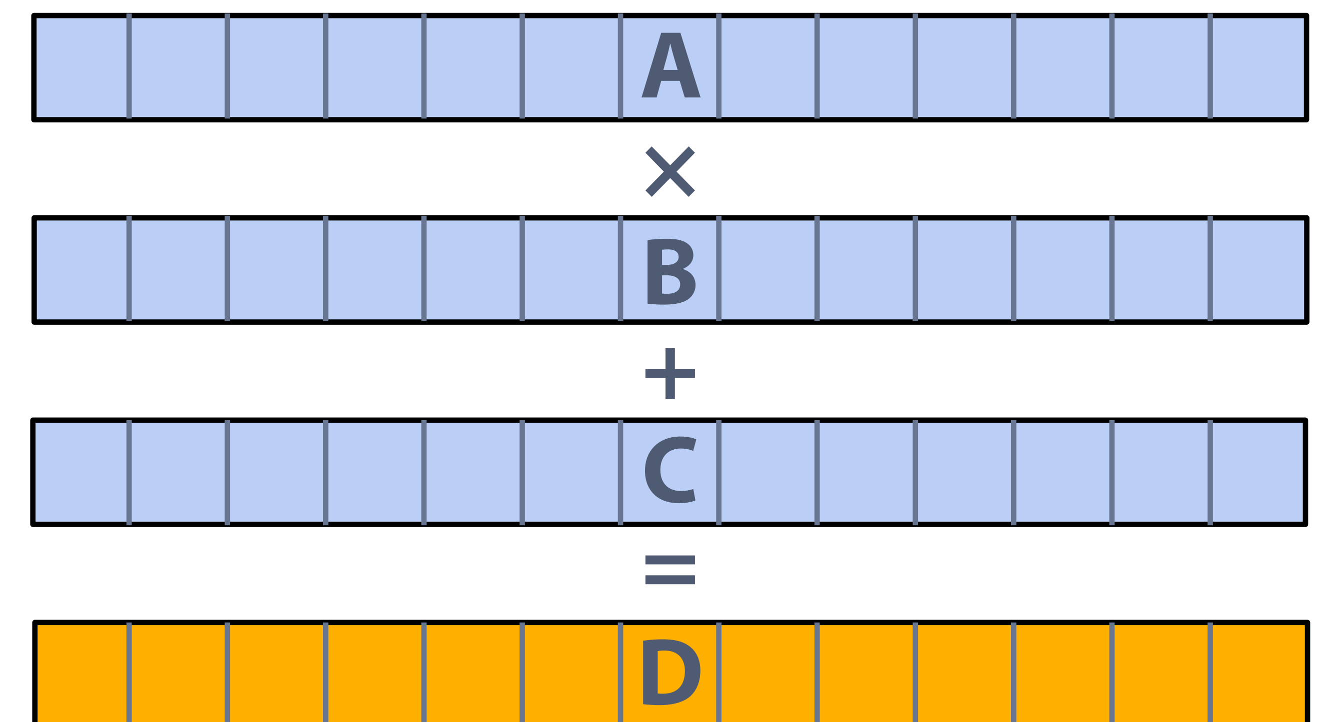
More cores, more ALUs, no large traditional cache hierarchy (use threads to tolerate latency)

Require high-bandwidth connection to memory

Thought experiment

Task: element-wise multiplication of two vectors A and B

1. Load input A[i]
2. Load input B[i]
3. Load input C[i]
4. Compute A[i] × B[i] + C[i]
5. Store result into D[i]



Four memory operations (16 bytes) for every MUL-ADD

Radeon HD 5870 can do 1600 MUL-ADDs per clock

Need ~20 TB/sec of bandwidth to keep functional units busy

Less than 1% efficiency… but 6x faster than 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 application developers on throughput-optimized systems.

Bandwidth is a critical resource

- A high-end GPU (e.g., Radeon HD 5870) has...
  - Over **twenty times** (2.7 TFLOPS) the compute performance of quad-core CPU
  - No large cache hierarchy to absorb memory requests

- GPU memory systems are designed for throughput
  - Wide memory bus (150-200 GB/sec)
  - Still, this is only **six-to-eight times** the bandwidth available to CPU

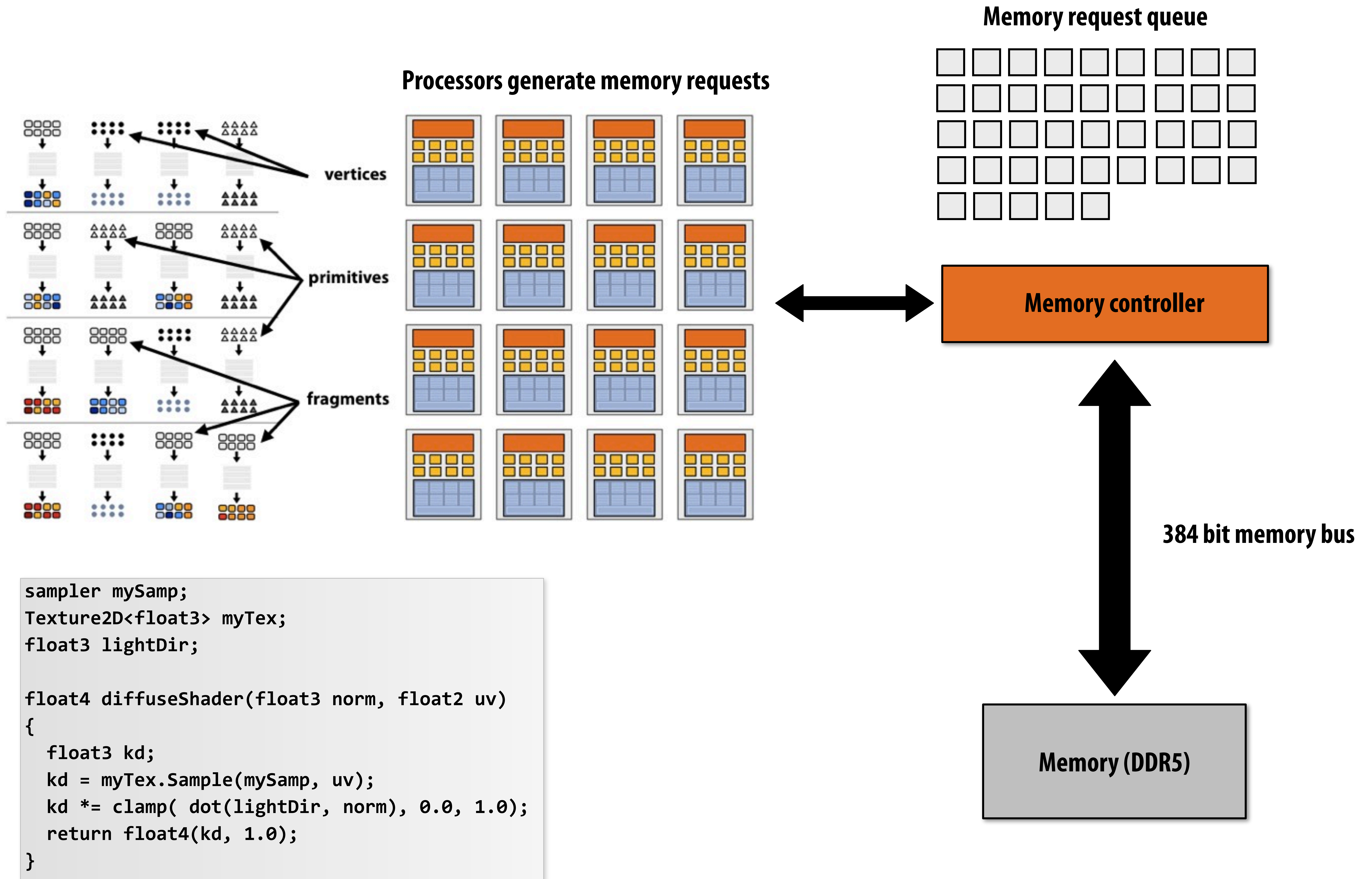
Bandwidth is a critical resource

- Use available bandwidth well

- Fetch data from memory less often (share/reuse data)

- Request data less often (instead, do more math: it's "free")
  - "arithmetic intensity" : ratio of math to data access

Using available bandwidth well




```
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);
}
```

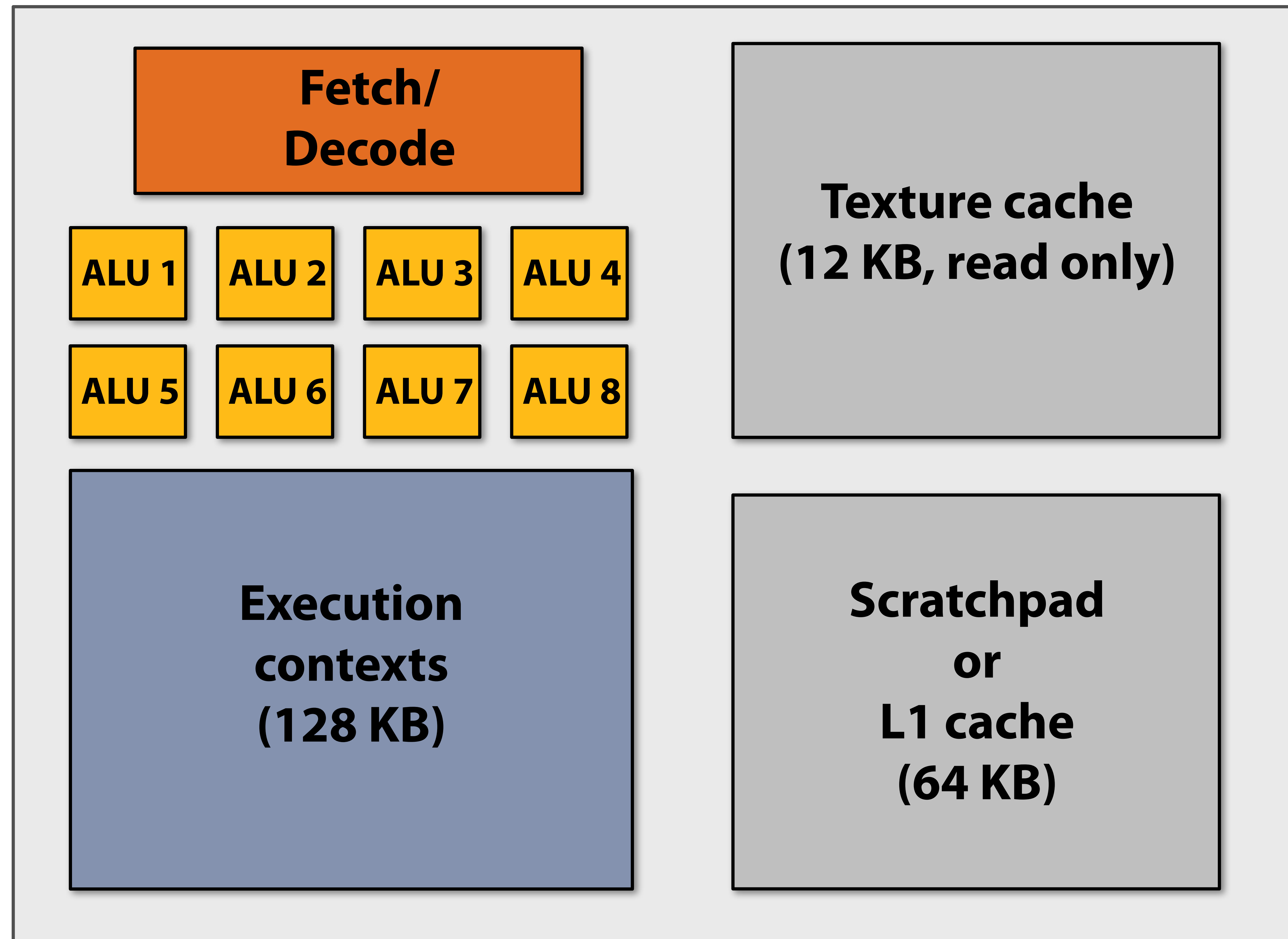
Bandwidth is a critical resource

- Use available bandwidth well
  - GPUs feature sophisticated memory request reordering logic
  - Repack/reorder/interleave many buffered memory requests to maximize memory utilization

- Fetch data from memory less often (share/reuse data)
  - Intra-fragment reuse
  - Cross-fragment reuse
  - Compression

- Request data less often (instead, do more math: it's "free")
  - "arithmetic intensity" : ratio of math to data access

Scratchpad for reuse known at compile-time



Load-data into scratchpad (LD addr -> scratchpad addr)

Many fragments reuse data loaded into scratchpad once ***

*** Not in OpenGL/Direct3D shader programming model (under the hood optimization)

Bandwidth is a critical resource

- Use available bandwidth well
  - GPUs feature sophisticated memory request reordering logic
  - Repack/reorder/interleave many buffered memory requests to maximize memory utilization

- Fetch data from memory less often (share/reuse data)
  - Intra-fragment reuse
  - Cross-fragment reuse
  - Compression

- Request data less often (instead, do more math: it's "free")
  - "arithmetic intensity" : ratio of math to data access

Shading often has high arithmetic intensity

Image credit: http://caig.cs.nctu.edu.tw/course/CG2007

```
sampler mySamp;
Texture2D<float3> myTex;
float3 ks;
float  shinyExp;
float3 lightDir;
float3 viewDir;

float4 phongShader(float3 norm, float2 uv)
{
  float result;
  float3 kd;
  kd = myTex.Sample(mySamp, uv);
  float spec = dot(viewDir, 2 * dot(-lightDir, norm) * norm + lightDir);
  result = kd * clamp(dot(lightDir, norm), 0.0, 1.0);
  result += ks * exp(spec, shinyExp);
  return float4(result, 1.0);
}
```

3 scalar float operations + 1 exp()
8 float3 operations + 1 clamp()
1 texture access (highlighted in red)

Vertex processing often has higher arithmetic intensity than fragment processing (less use of texturing)

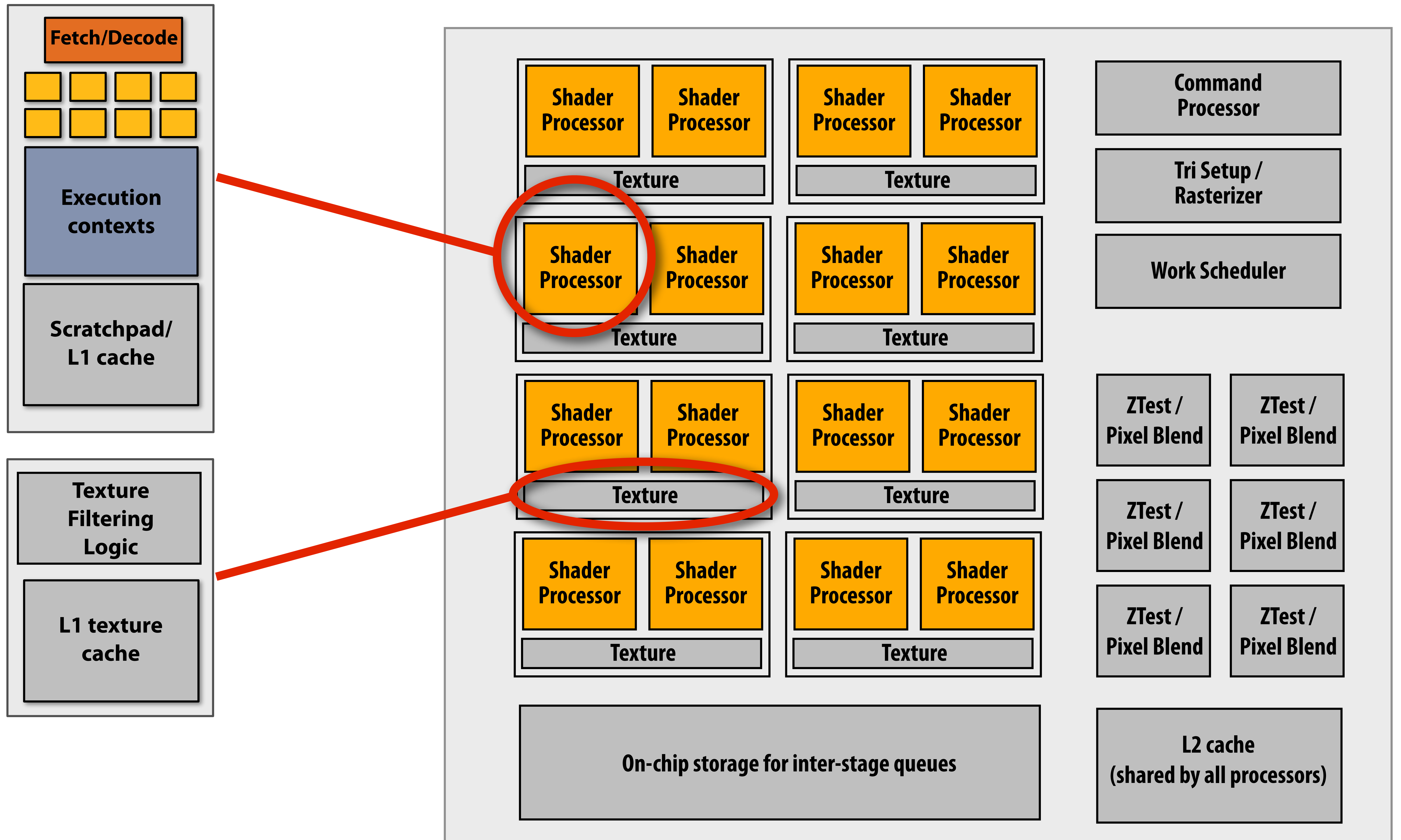
Summary: workloads that run efficiently on a GPU's programmable cores ...

- Have thousands of independent pieces of work
  - Utilizes many ALUs on many cores
  - Have <u>much more</u> parallel work than numbers of GPU ALUs, enabling large-scale interleaving as a mechanism to hide memory latency

- Are amenable to instruction stream sharing
  - Maps to SIMD execution well

- Are compute-heavy: the ratio of math operations to memory access is high
  - Not limited by memory bandwidth

Modern GPU: heterogeneous many-core



Homogeneous collection of throughput-optimized programmable processing cores

Augmented by fixed-function logic

adapted from

Lecture 11: “GPGPU” computing and the CUDA/OpenCL Programming Model

Kayvon Fatahalian

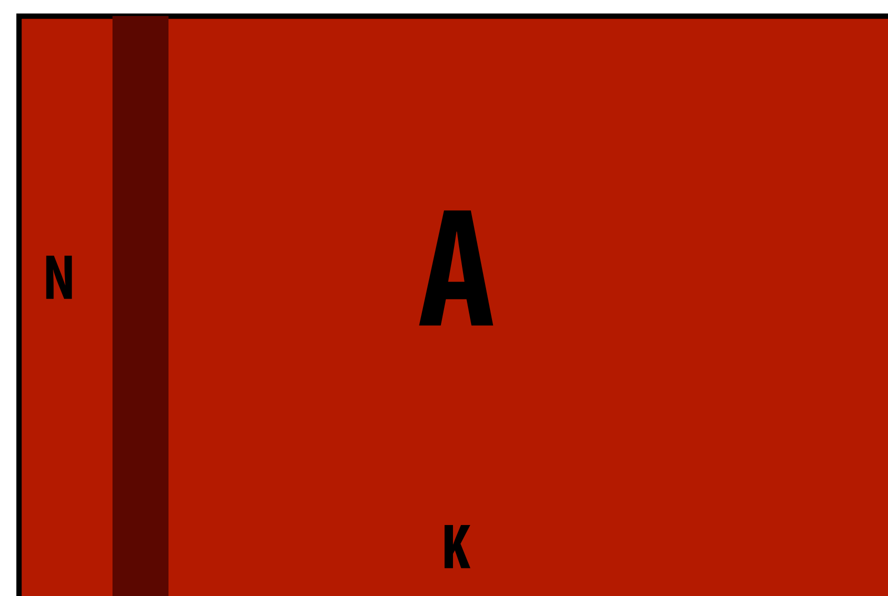
CMU 15-869: Graphics and Imaging Architectures (Fall 2011)

- **Some GPGPU history**
- **The CUDA (or OpenCL) programming model**

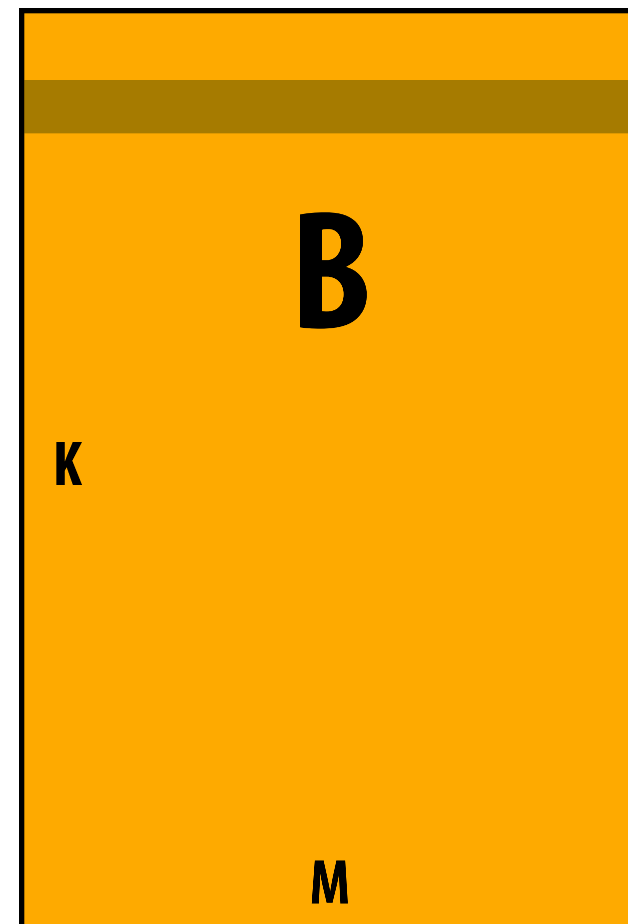
Early GPU-based scientific computation

Dense matrix-matrix multiplication

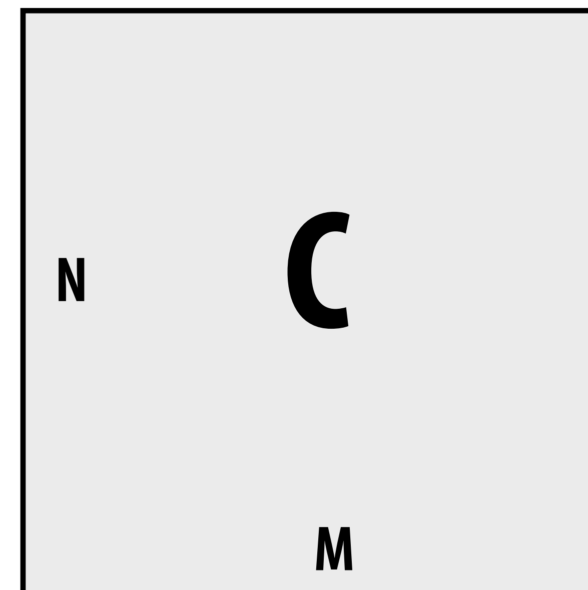
[Larson and McAllister, SC 2001]



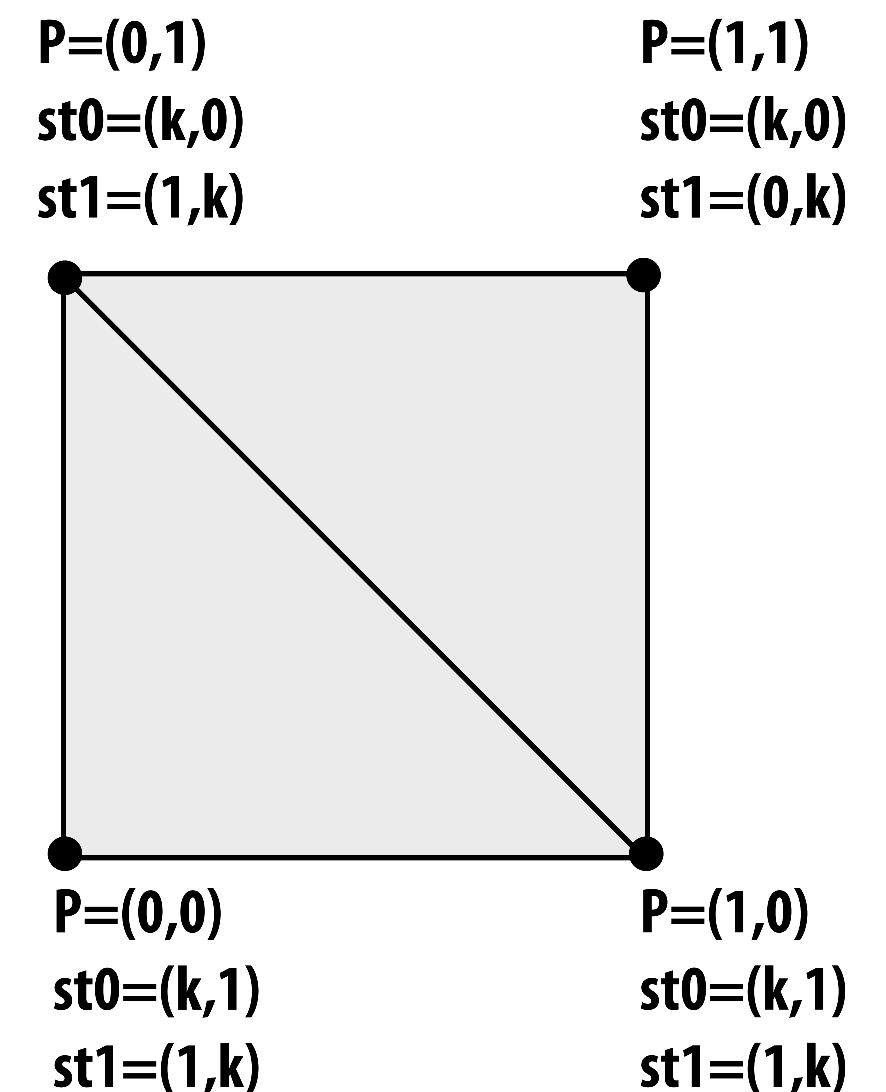
K x N texture 0



M x K texture 1

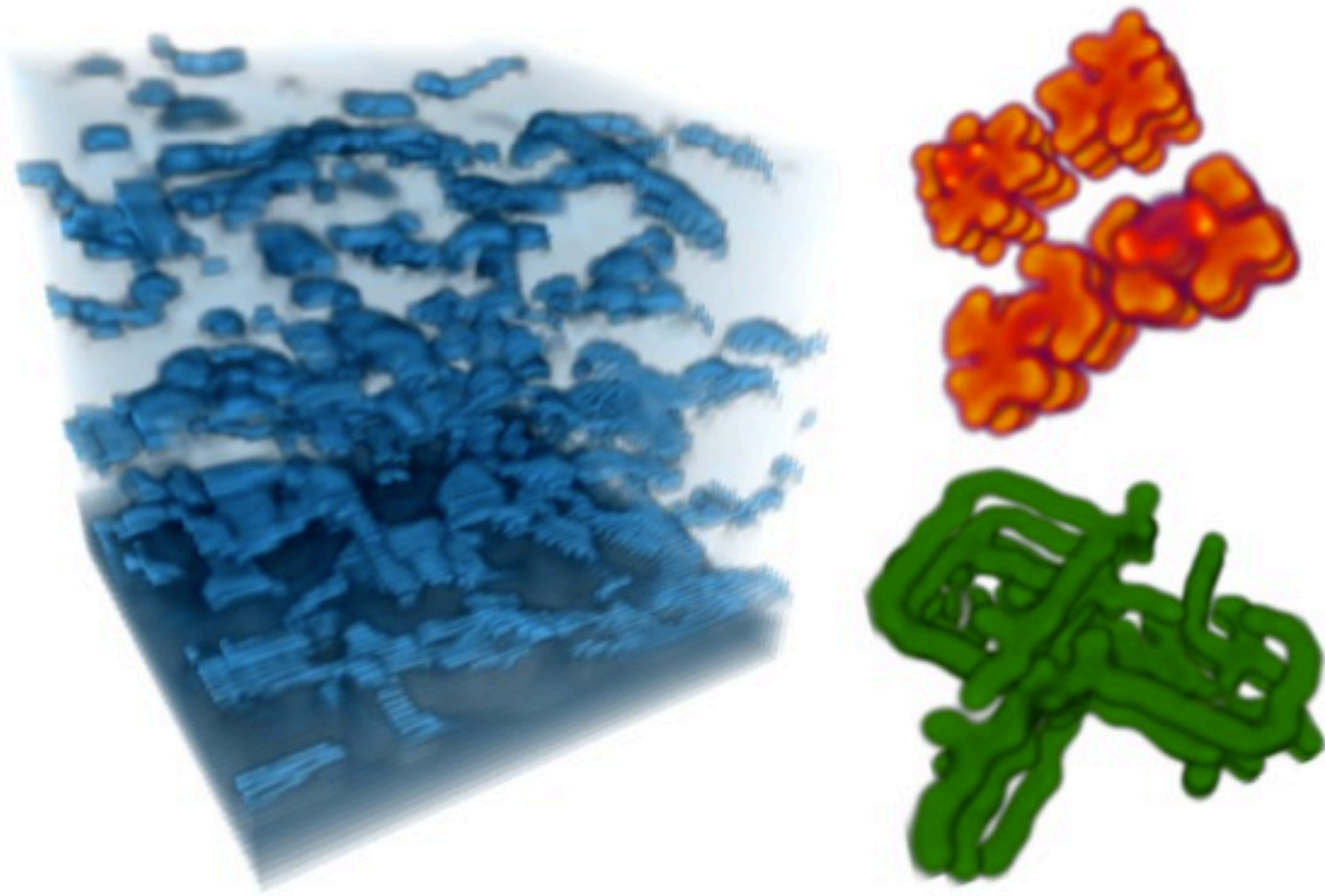


M x N frame buffer

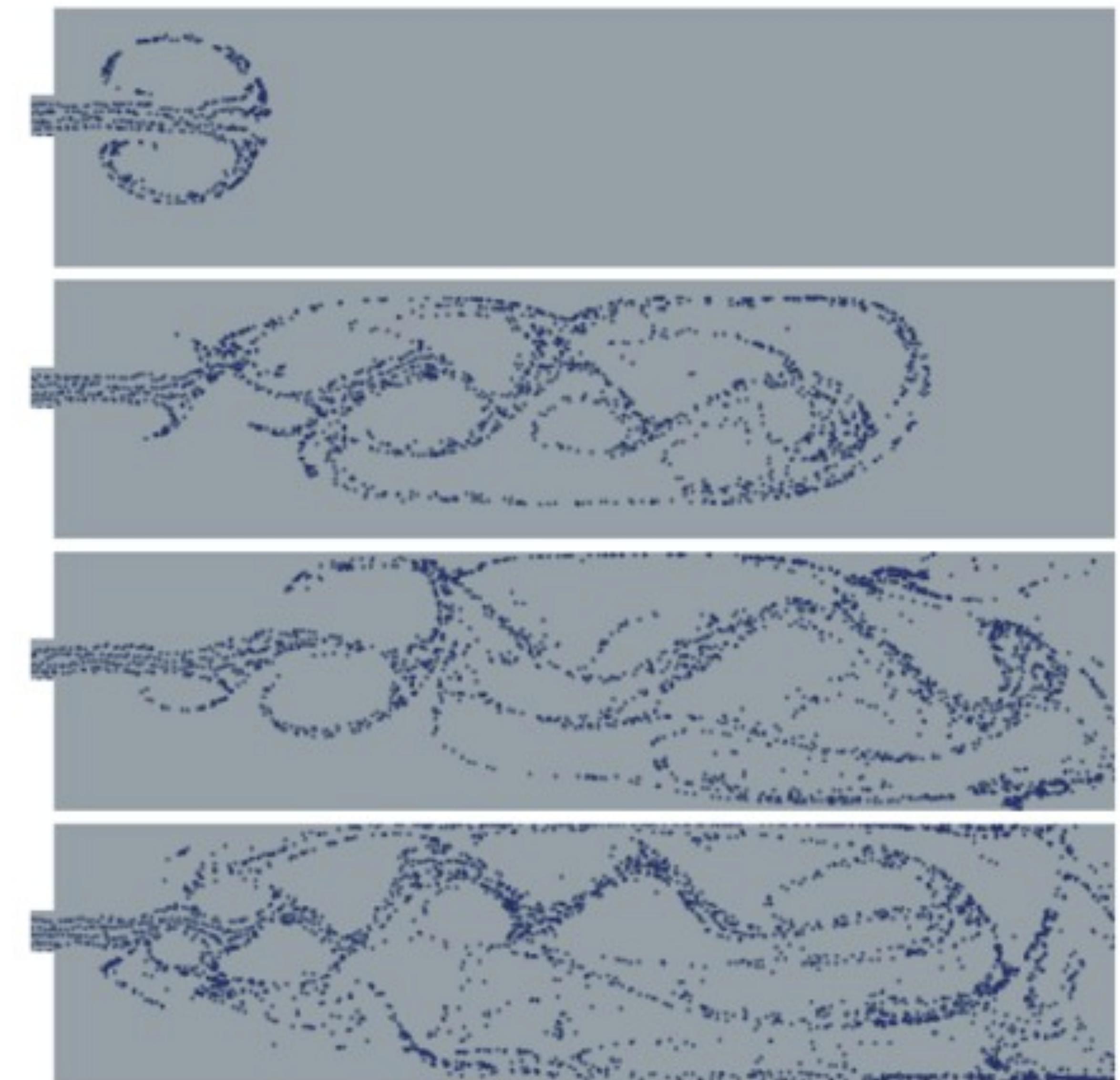


Set frame buffer blend mode to ADD
for k=0 to K
Set texture coords
Render 1 full-screen quadrilateral

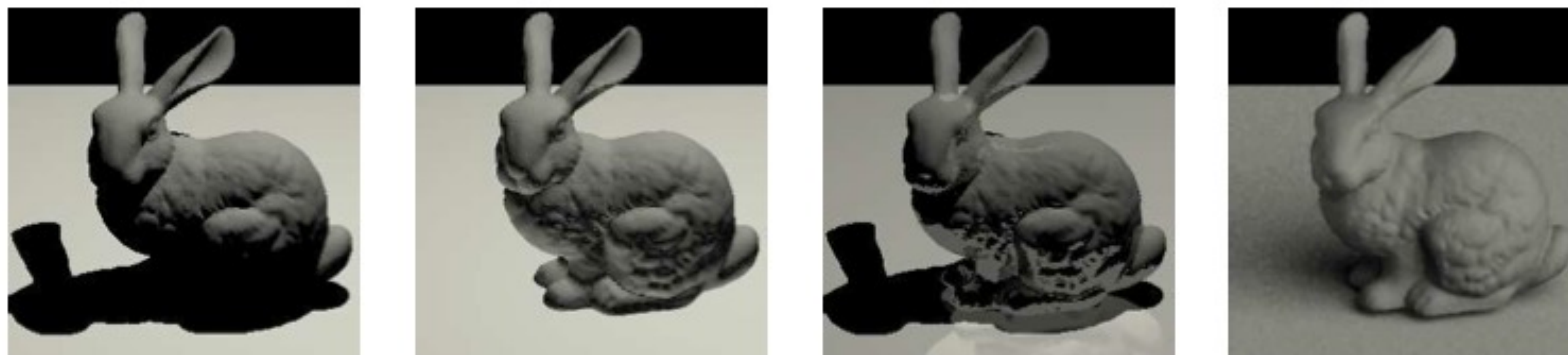
“GPGPU” 2002-2003



Coupled Map Lattice Simulation [Harris 02]



Sparse Matrix Solvers [Bolz 03]



Ray Tracing on Programmable Graphics Hardware [Purcell 02]

Brook for GPUs

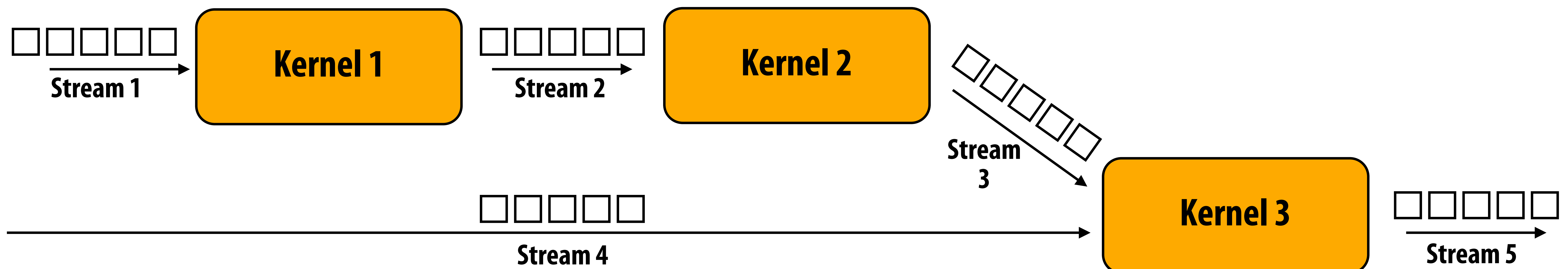
- **Abstract GPU as a generic stream processor (C extension)**
  - Streams: 1D, 2D arrays of data
  - Kernels: per-element processing of stream data **
  - Reductions: stream --> scalar
- **Influences**
  - Data-parallel programing: ZPL, Nesl
  - Stream programming: StreaMIT, StreamC/Kernel
- **Brook runtime generates appropriate OpenGL calls**

```
kernel void scale(float amount, float a<>, out float b<>)
{
   b = amount * a;
}

// note: omitting initialization
float scale_amount;
float input_stream<1000>;
float output_stream<1000>;

// map kernel onto streams
scale(scale_amount, input_stream, output_stream);
```

** Broke traditional stream processing model with in-kernel gather (more on this later)

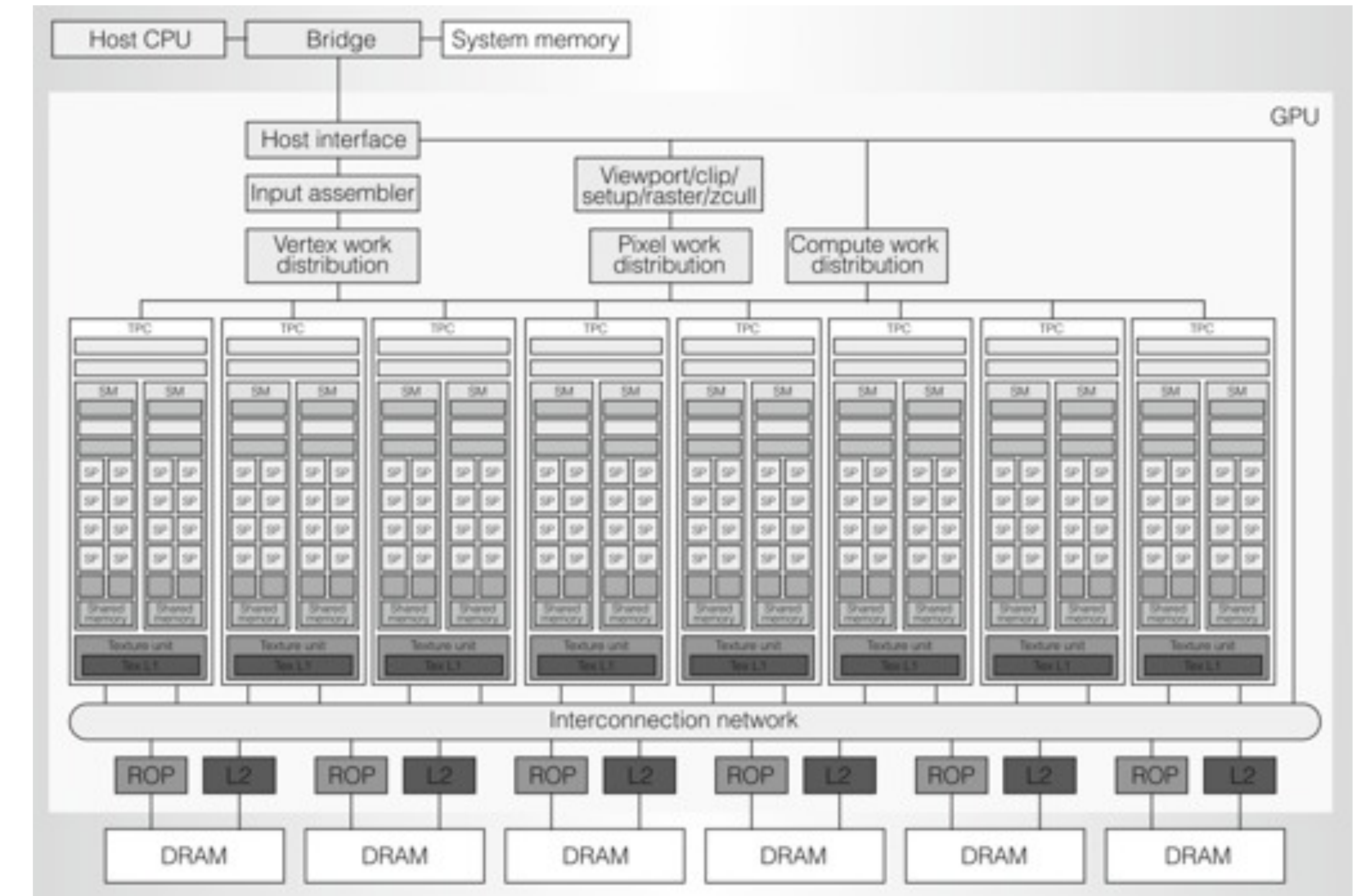
Stream programming ("pure")

- Streams
  - Encapsulate per-element parallelism
  - Encapsulate producer-consumer locality
- Kernels
  - Functions (side-effect-free)
  - Encapsulate locality (kernel's working set defined by inputs, outputs, and temporaries)
  - Encapsulate instruction-stream coherence (same kernel applied to each stream element)
- Modern implementations (e.g., StreaMIT, StreamC/KernelC) relied on static scheduling by compiler to achieve high performance



NVIDIA CUDA [Ian Buck at NVIDIA, 2007]

- **Alternative programming interface to Tesla-class GPUs**
  - Recall: Tesla was first "unified shading" GPU



- **Low level, reflects capabilities of hardware**
  - Recall arguments in Cg paper
  - Combines some elements of streaming, some of threading (like HW does)

- **Today: open standards embodiment of this programming model is OpenCL (Microsoft embodiment is Compute Shader)**

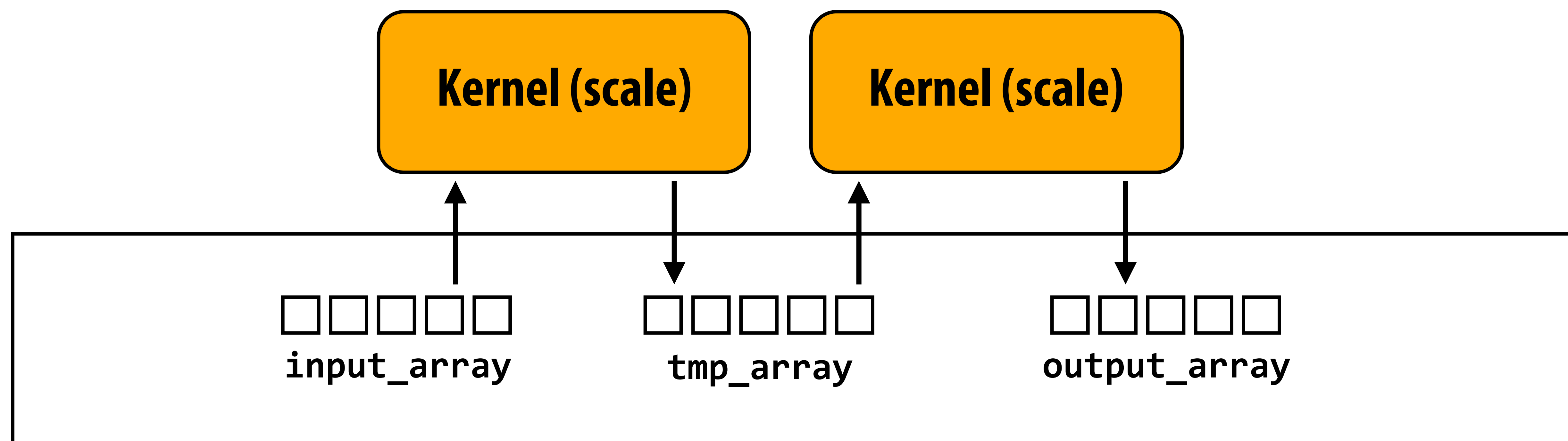
CUDA constructs (the kernel)

```
// CUDA kernel definition
__global__ void scale(float amount, float* a, float* b)
{
   int i = threadIdx.x;   // CUDA builtin: get thread id
   b[i] = amount * a[i];
}

// note: omitting initialization via cudaMalloc()
float scale_amount;
float* input_array;
float* output_array;

// launch N threads, each thread executes kernel 'scale'
scale<<1,N>>(scale_amount, input_array, output_array);
```

Bulk thread launch: logically spawns N threads

Can system find producer-consumer?

```
// CUDA kernel definition
__global__ void scale(float amount, float* a, float* b)
{
   int i = threadIdx.x;   // CUDA builtin: get thread id
   b[i] = amount * a[i];
}

// note: omitting initialization via cudaMalloc()
float scale_amount;
float* input_array;
float* output_array;
float* tmp_array;

scale<<1,N>>(scale_amount, input_array, tmp_array);
scale<<1,N>>(scale_amount, tmp_array, output_array);
```



CUDA constructs (the kernel)

```
// CUDA kernel definition
__global__ void scale(float amount, float* a, float* b)
{
   int i = threadIdx.x;   // CUDA builtin: get thread id
   b[i] = amount * a[i];
}

// note: omitting initialization via cudaMalloc()
float scale_amount;
float* input_array;
float* output_array;

// launch N threads, each thread executes kernel 'scale'
scale<<1,N>>(scale_amount, input_array, output_array);
```

Bulk thread launch: logically spawns N threads

Question: What should N be?

Question: Do you normally think of "threads" this way?

CUDA constructs (the kernel)

```
// CUDA kernel definition
__global__ void scale(float amount, float* a, float* b)
{
   int i = threadIdx.x;   // CUDA builtin: get thread id
   b[i] = amount * a[i];
}

// note: omitting initialization via cudaMalloc()
float scale_amount;
float* input_array;
float* output_array;

// launch N threads, each thread executes kernel 'scale'
scale<<1,N>>(scale_amount, input_array, output_array);
```

Given this implementation: each invocation of `scale` kernel is independent.

(bulk thread launch semantics no different than sequential semantics)

CUDA system has flexibility to parallelize any way it pleases.

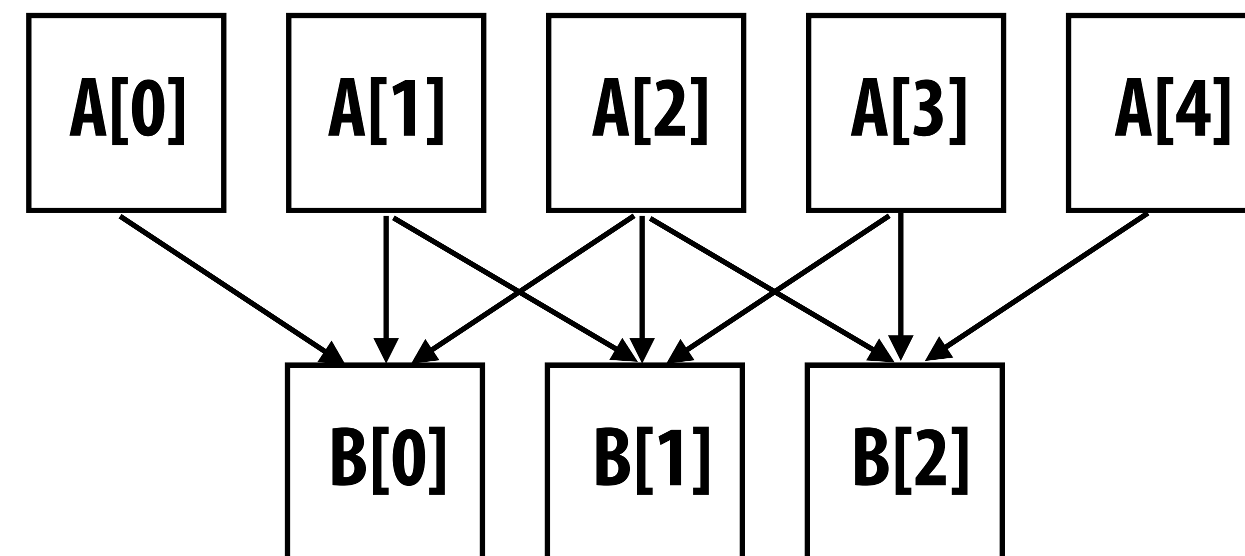
In many cases, thinking about a CUDA kernel as a stream processing kernel, and CUDA arrays as streams is perfectly reasonable.

(programmer just has to do a little indexing in the kernel to get a reference to stream inputs/outputs)

Convolution example

```
// assume len(A) = len(B) + 2
__global__ void convolve(float* a, float* b)
{
   // ignore
   int i = threadIdx.x;
   b[i] = a[i] + a[i+1] + a[i+2];
}
```

Note "adjacent" threads load same data.
Here: 3x input reuse (reuse increases with width of convolution filter)



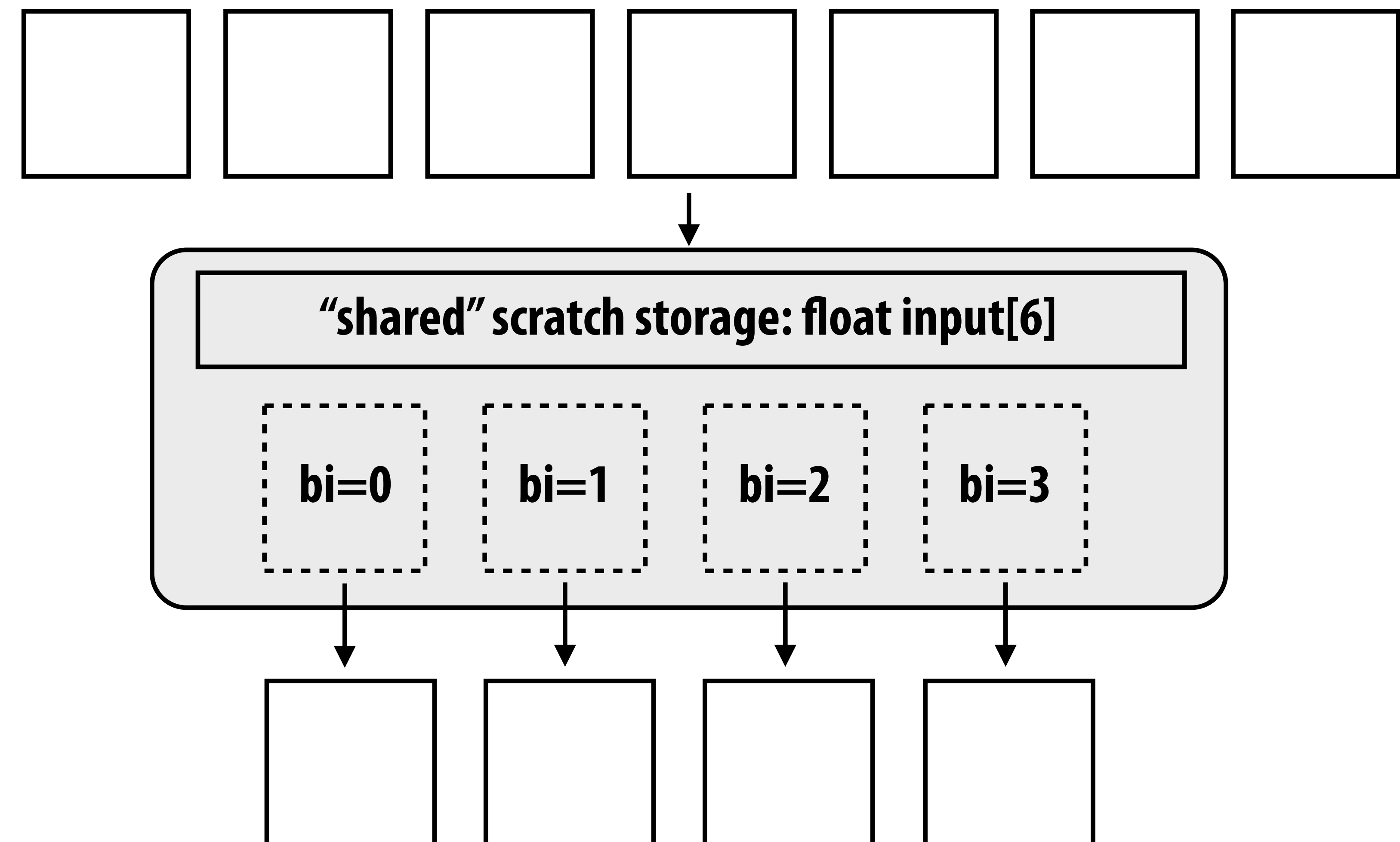
CUDA thread hierarchy

```
#define BLOCK_SIZE 4

__global__ void convolve(float* a, float* b)
{
   __shared__ float input[BLOCK_SIZE + 2];

   int bi = blockIdx.x;
   int ti = threadIdx.x;

   input[bi] = A[ti];
   if (bi < 2)
   {
      input[BLOCK_SIZE+bi] = A[ti+BLOCK_SIZE];
   }

   __syncthreads();   // barrier

   b[ti] = input[bi] + input[bi+1] + input[bi+2];
}

// allocation omitted
// assume len(A) = N+2, len(B)=N
float* A, *B;

convolve<<BLOCK_SIZE, N/BLOCK_SIZE>>(A, B);
```

CUDA threads are grouped into thread blocks

Threads in a block are <u>not</u> independent.
They can cooperate to process shared data.

1. **Threads communicate through `__shared__` variables**
2. **Threads barrier via `__syncthreads()`**



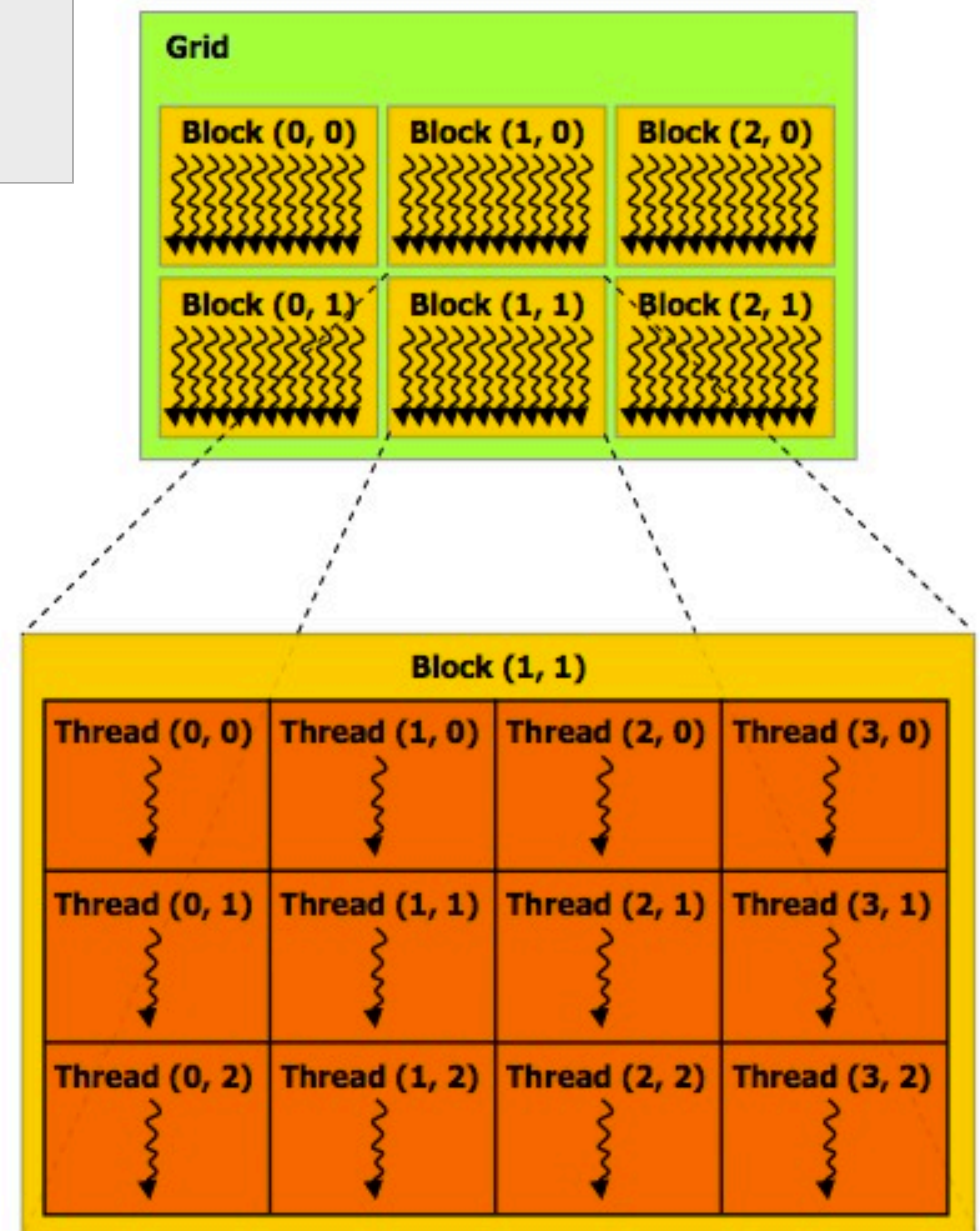
CUDA thread hierarchy

```
// this code will launch 96 threads
// 6 blocks of 16 threads each

dim2 threadsPerBlock(4,4);
dim2 blocks(3,2);
myKernel<<blocks, threadsPerBlock>>();
```

Thread blocks (and the overall "grid" of blocks) can be 1D, 2D, 3D
(Convenience: many CUDA programs operate on n-D grids)

Thread blocks represent independent execution

Threads in a thread block executed <u>simultaneously</u> on same GPU core
- Why on the same core?
- Why simultaneously?



Source: CUDA Programming Manual

The common way to think about CUDA

(thread centric)

- CUDA is a multi-threaded programming model
- Threads are logically grouped together into blocks and gang scheduled onto cores
- Threads in a block are allowed to synchronize and communicate through barriers and shared local memory
- Note: Lack of communication between threads in different blocks gives scheduler some flexibility (can "stream" blocks through the system)**

** Using global memory atomic operations provide a form of inter-thread block communication (more on this in a second)

Another way to think about CUDA

(like a streaming system: thread block centric)

- CUDA is a stream programming model (recall Brook)
  - Stream elements are now blocks of data
  - Kernels are thread blocks (larger working sets)
- Kernel invocations independent, but are multi-threaded
  - Achieves additional fine-grained parallelism
- Think: Implicitly parallel across thread blocks (kernels)
- Think: Explicitly parallel within a block

Canonical CUDA thread block program:

Threads cooperatively load block of data from input arrays into shared mem

```
__syncThreads(); // barrier
```

Threads perform computation, accessing shared mem

```
__syncThreads(); // barrier
```

Threads cooperatively write block of data to output arrays

Choosing thread-block sizes

Question: how many threads should be in a thread block?

Recall from GPU core lecture:

- How many threads per core?
- How much shared local memory per core?

“Persistent” threads

- No semblance of streaming at all any more
- Programmer is always thinking explicitly parallel
- Threads use atomic global memory operations to cooperate

```
// Persistent thread: Run until work is done, processing multiple work
// elements, rather than just one. Terminates when no more work is available
__global__ void persistent(int* ahead, int* bhead, int count, float* a, float* b)
{
    int in_index;
    while ( (in_index = read_and_increment(ahead)) < count)
    {
         // load a[in_index];

         // do work

         int out_index = read_and_increment(bhead);

         // write result to b[out_index]
    }
}

// launch exactly enough threads to fill up machine
// (to achieve sufficient parallelism and latency hiding)
persistent<<numBlocks,blockSize>>(ahead_addr, bhead_addr, total_count, A, B);
```

Quick aside: why was CUDA successful?

(Kayvon's personal opinion)

1. Provides access to a cheap, very fast machine

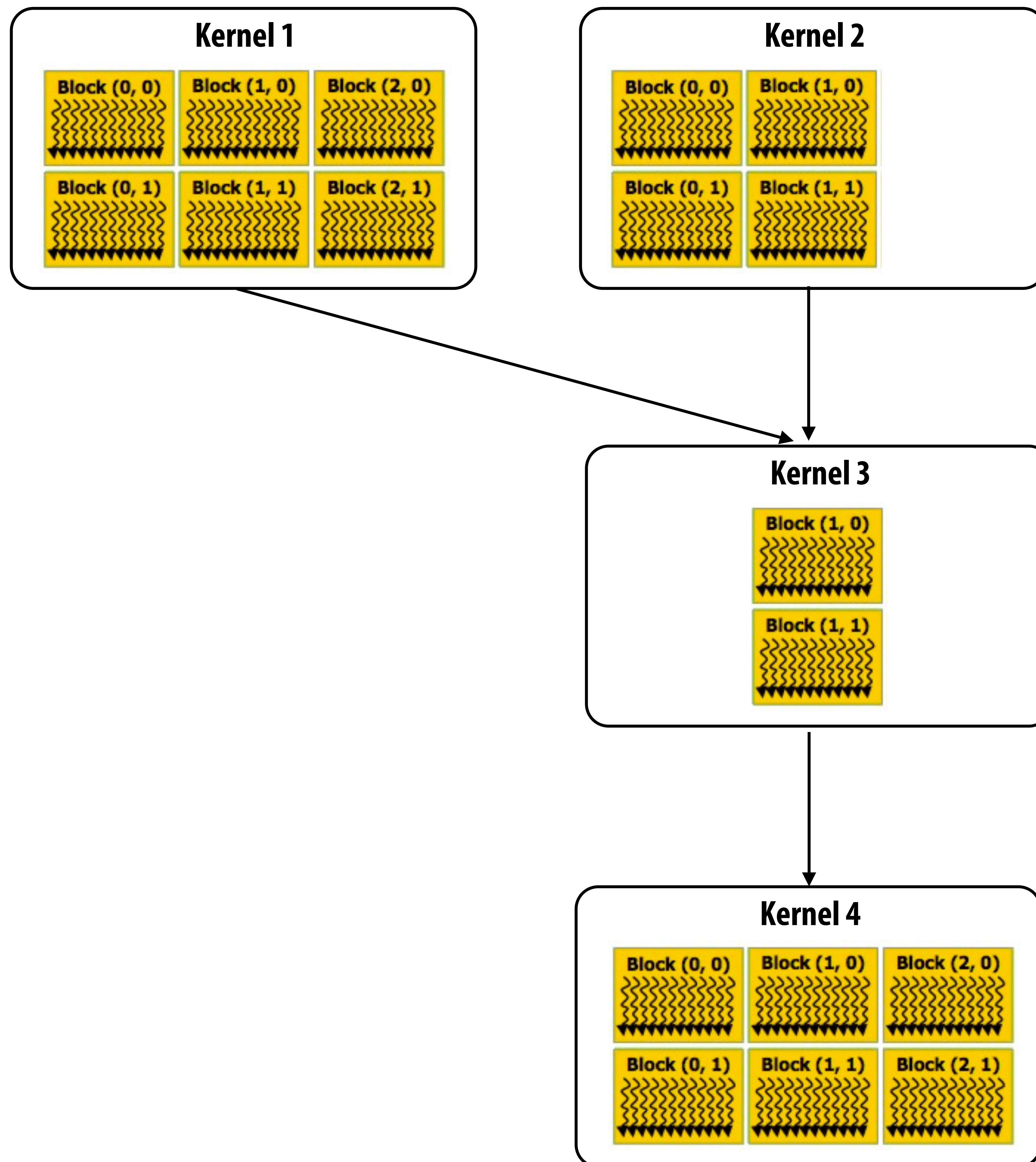
2. SPMD abstraction allows programmer to write scalar code, have it (almost trivially) mapped to vector hardware

Intel SPMD Program Compiler
An open-source compiler for high-performance SIMD programming on the CPU

Note: Five years later... one Intel employee (with LLVM and a graphics background)

3. More like thread programming than streaming: arbitrary in-kernel gather (+ GPU hardware multi-threading to hide memory latency)
   - More familiar, convenient, and flexible in comparison to more principled data-parallel or streaming systems
     [StreamC/KernelC, StreamMIT, ZPL, Nesl, synchronous data-flow, and many others]
   - The first program written is often pretty good
   - 1-to-1 with hardware behavior

Modern CUDA/OpenCL: DAGs of kernel launches



Note: arrows are specified dependencies between batch thread launches

Think of each launch like a draw() command in OpenGL (but application can turn off order, removing dependency on previous launch)

Graphics abstractions today

- Real-time graphics pipeline still hanging in there (Direct3D 11 / OpenGL 4)

- But lots of algorithm development in OpenCL/Direct3D compute shader/CUDA
  - Good: makes GPU compute power accessible. Triggering re-evaluation of best practices in field
  - Bad: community shifting too-far toward only thinking about current GPU-style data-parallelism

- CPU+GPU fusion will likely trigger emergence of alternative high-level frameworks for niches in interactive graphics
  - Example: NVIDIA Optix: new framework for ray tracing
    - Application provides key kernels, Optix compiler/runtimes schedules
    - Built on CUDA