



# How a GPU Works

Kayvon Fatahalian

November 2012



---

# Today

---

- 1. Review: the graphics pipeline**
- 2. History of GPUs**
- 3. How a modern GPU works (and why it is so fast!)**
- 4. Closer look at a real GPU design**
  - NVIDIA GTX 480 (the GPUs in the teaching lab)

---

# **Part 1:**

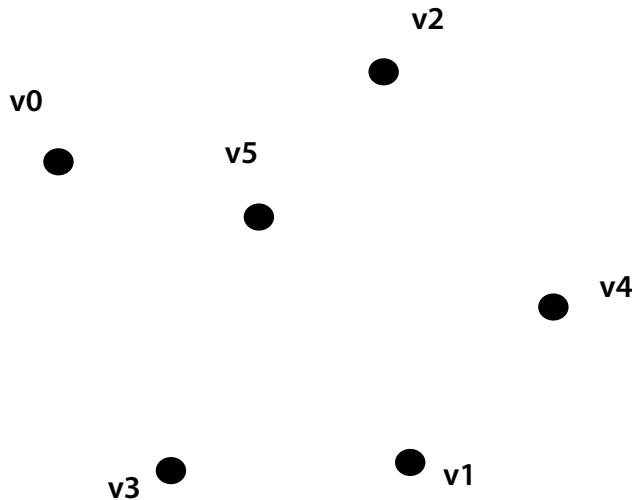
# **The graphics pipeline**

**(a programming abstraction)**

# Vertex processing

---

Vertices are transformed into “screen space”

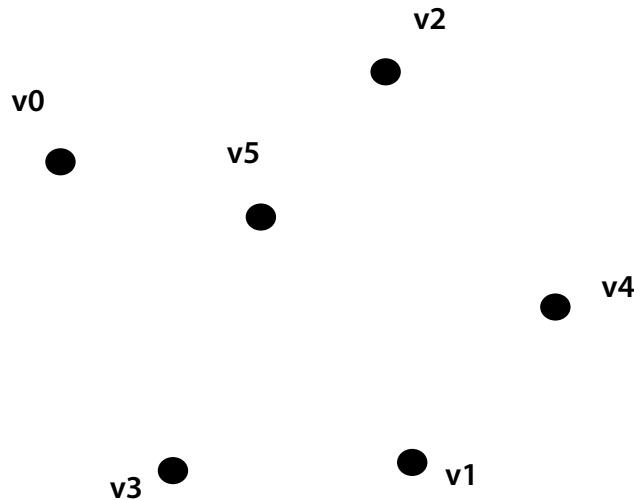


**Vertices**

# Vertex processing

---

Vertices are transformed into “screen space”

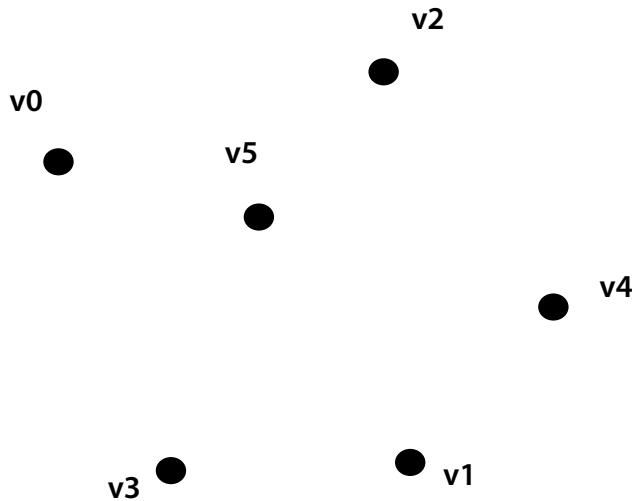


**Vertices**

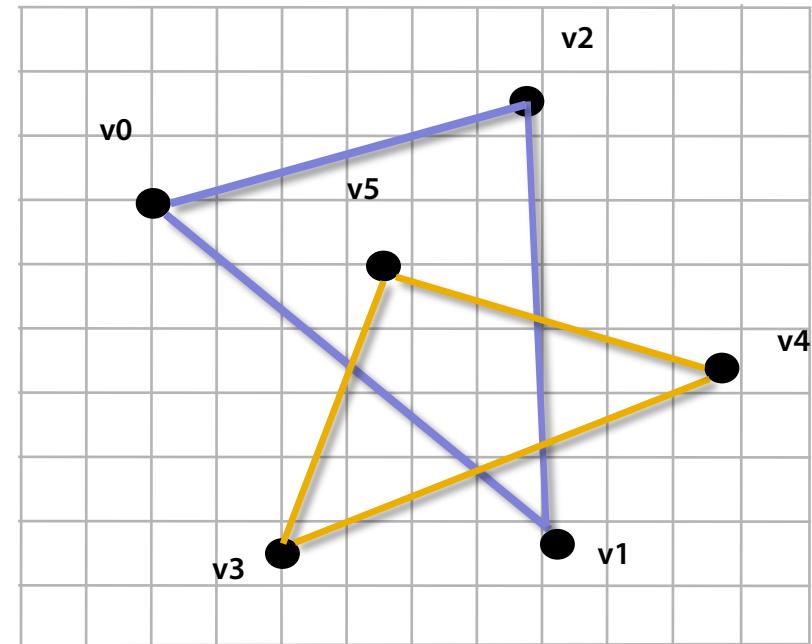
**EACH VERTEX IS  
TRANSFORMED  
INDEPENDENTLY**

# Primitive processing

Then organized into primitives that are clipped and culled...



**Vertices**

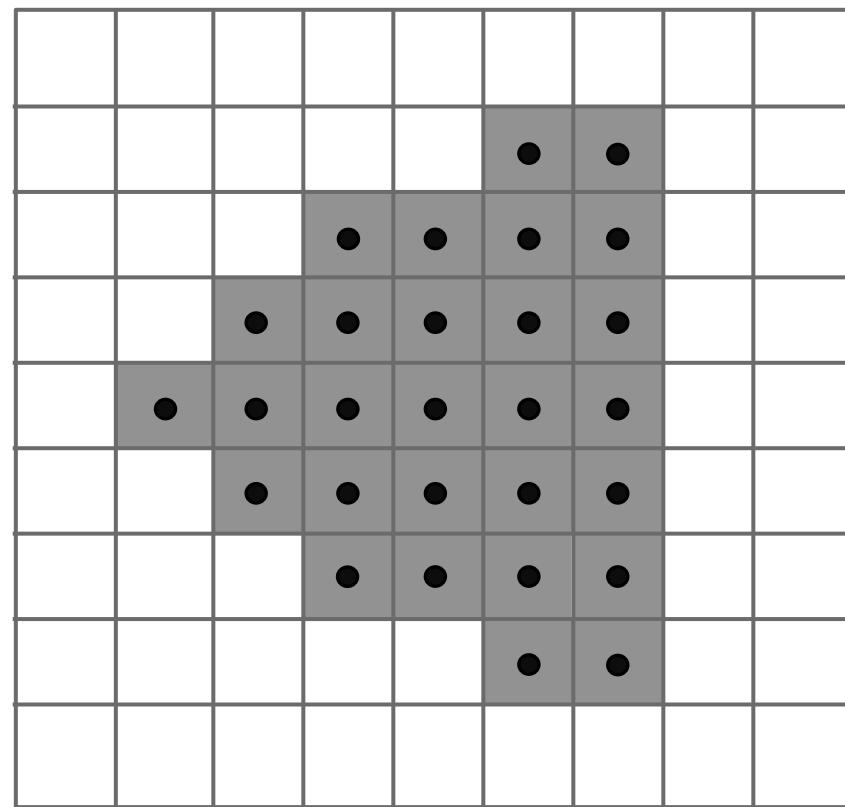
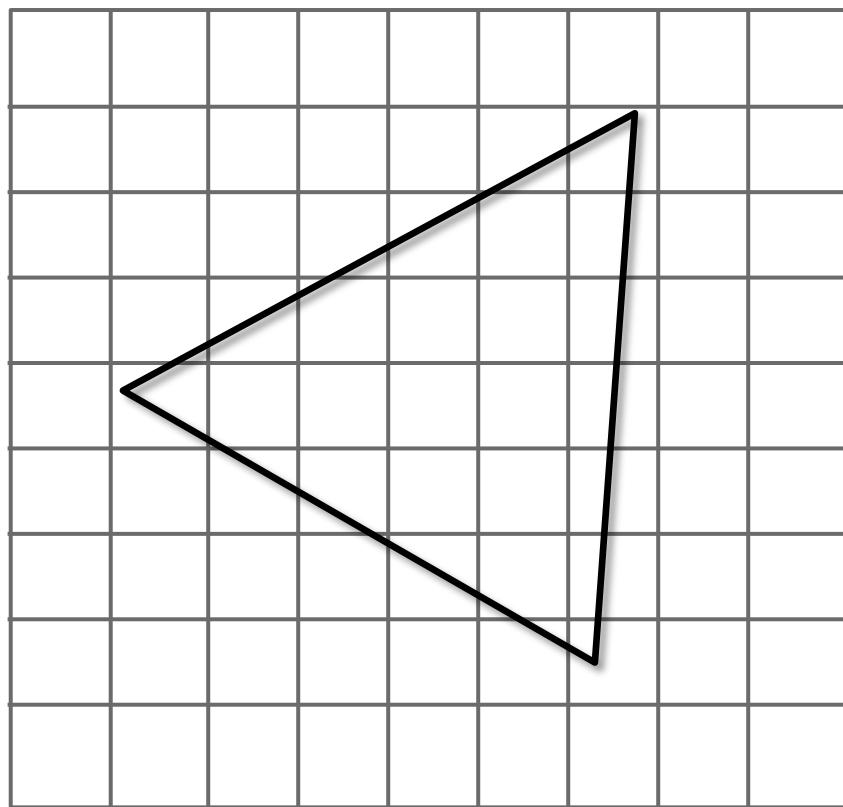


**Primitives  
(triangles)**

# Rasterization

---

Primitives are rasterized into “pixel fragments”

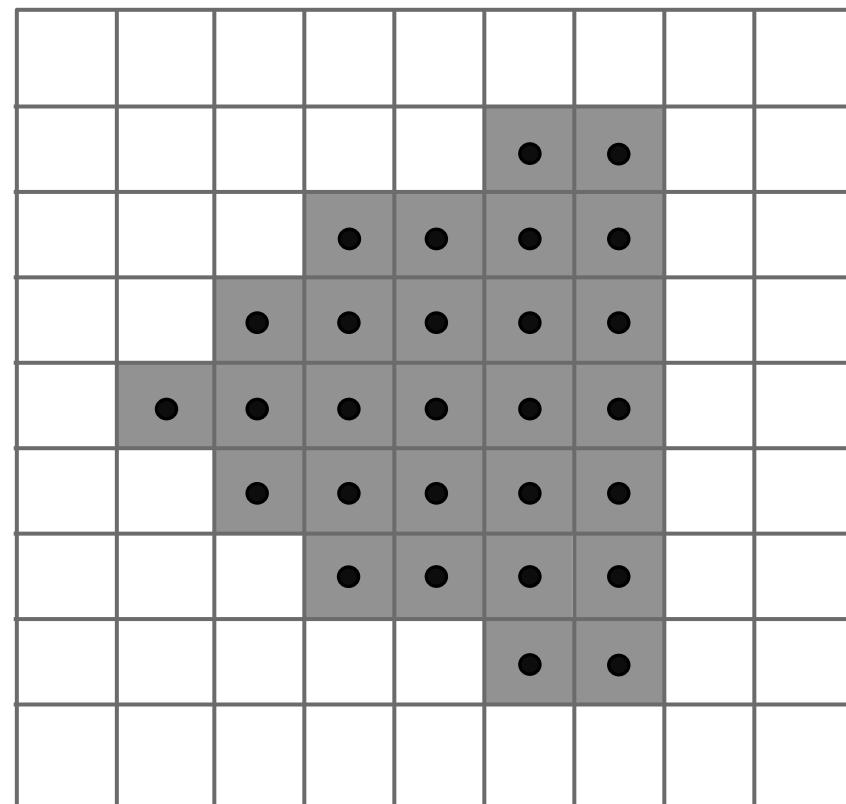
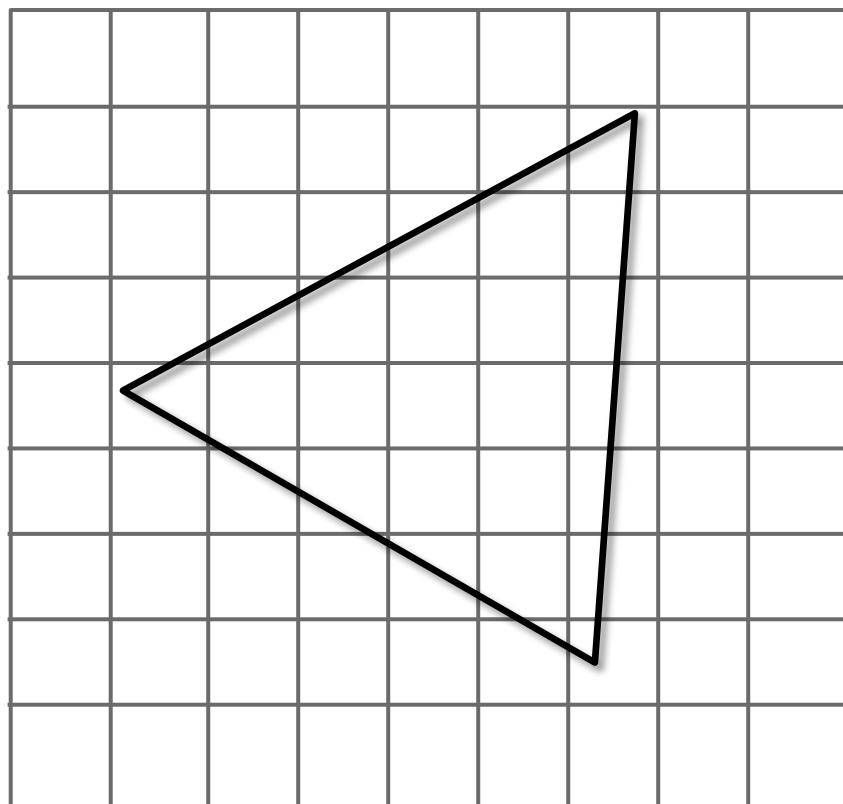


**Fragments**

# Rasterization

---

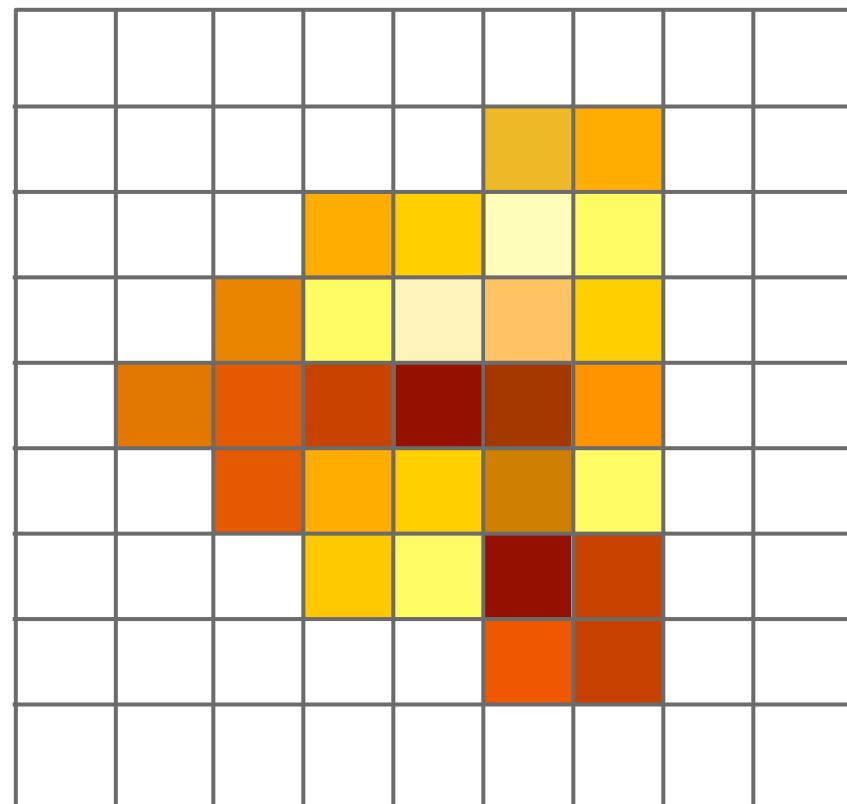
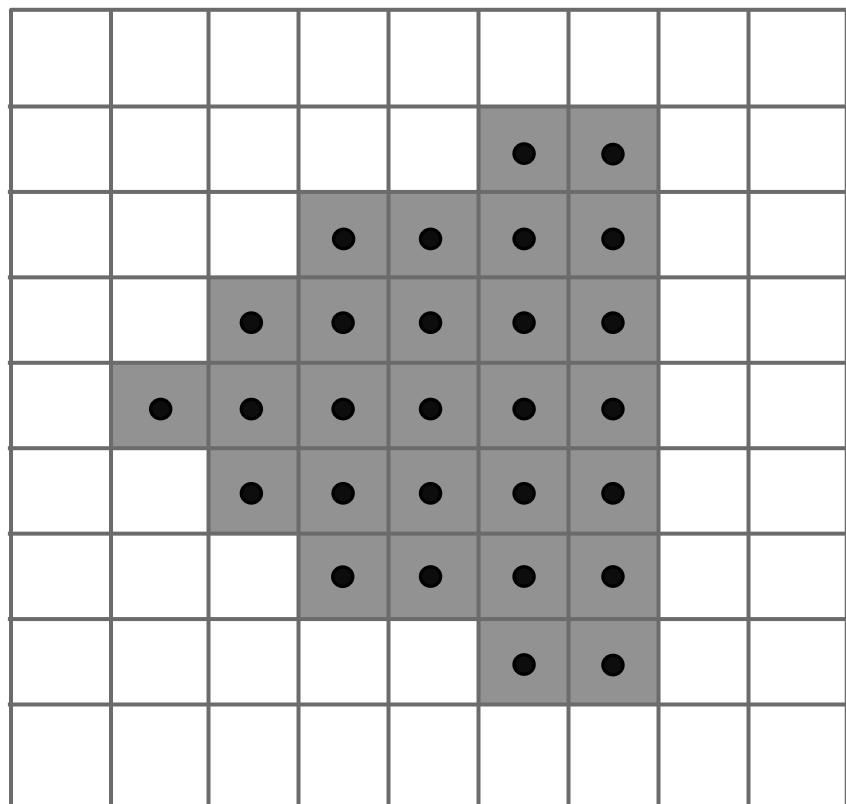
Primitives are rasterized into “pixel fragments”



**EACH PRIMITIVE IS RASTERIZED  
INDEPENDENTLY**

# Fragment processing

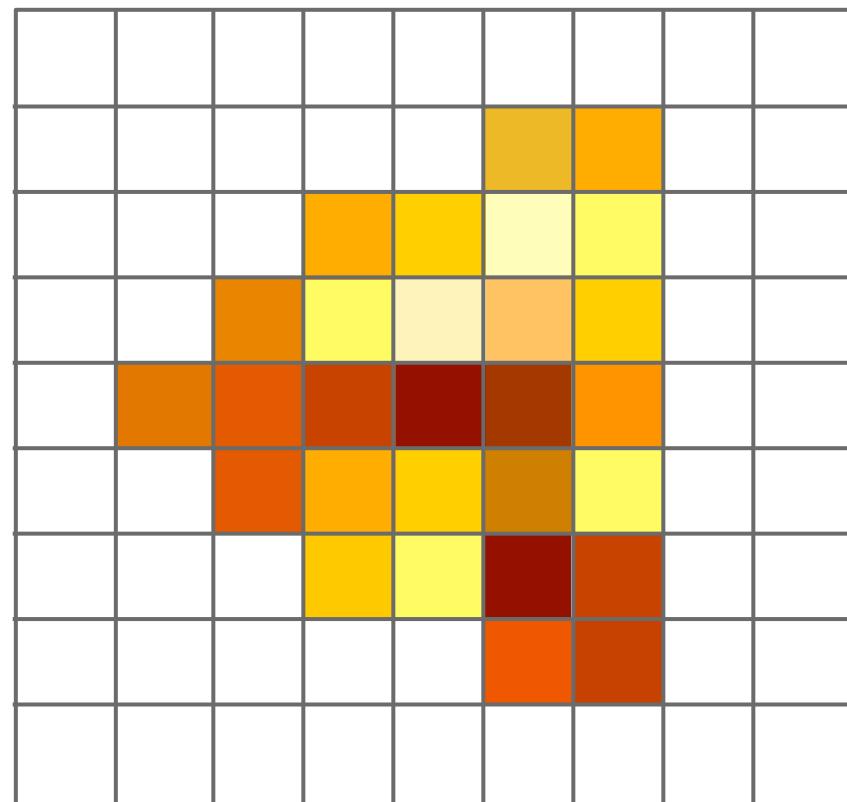
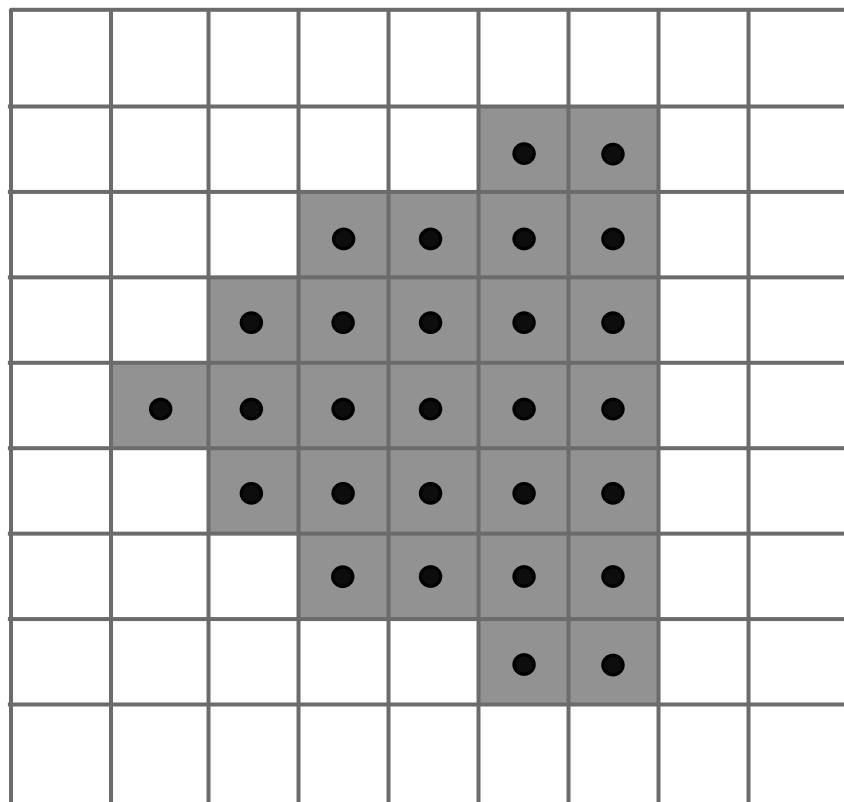
Fragments are shaded to compute a color at each pixel



Shaded fragments

# Fragment processing

Fragments are shaded to compute a color at each pixel

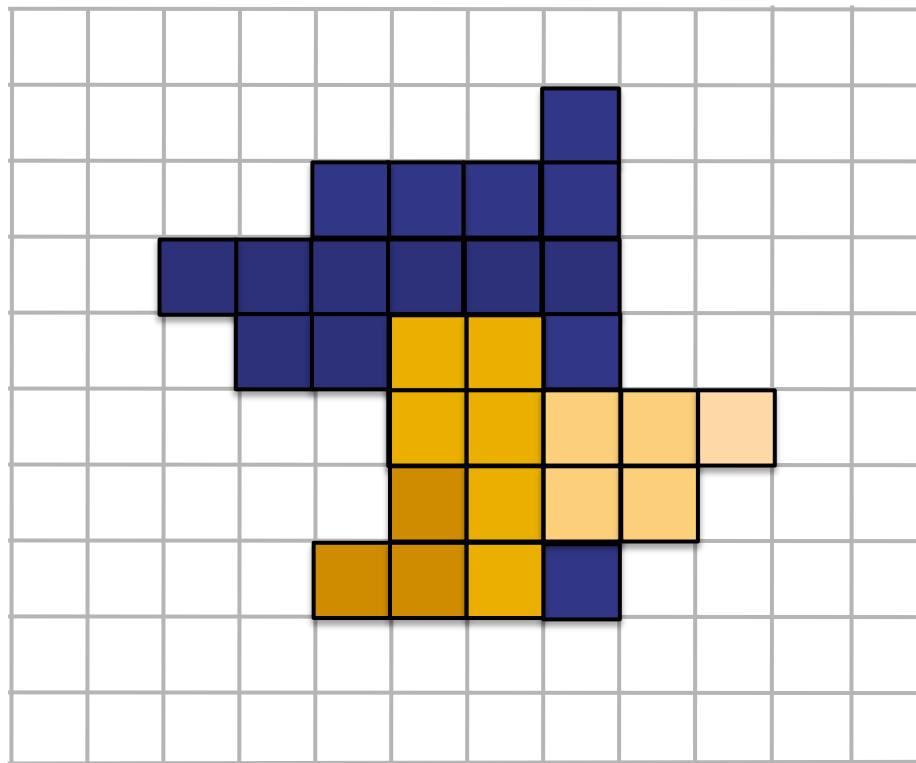


**EACH FRAGMENT IS PROCESSED  
INDEPENDENTLY**

# Pixel operations

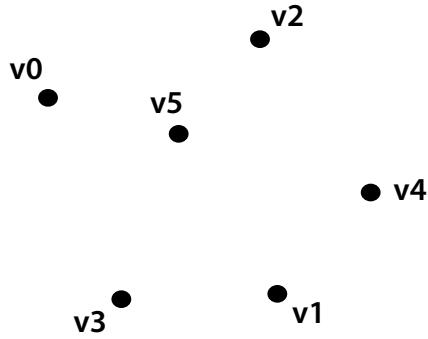
---

Fragments are blended into the frame buffer at their pixel locations (z-buffer determines visibility)

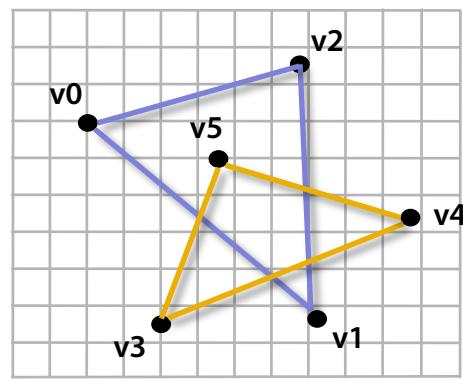


Pixels

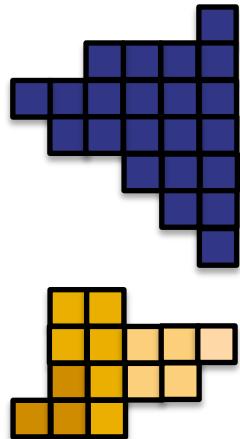
# Pipeline entities



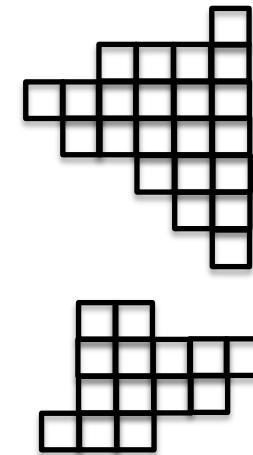
**Vertices**



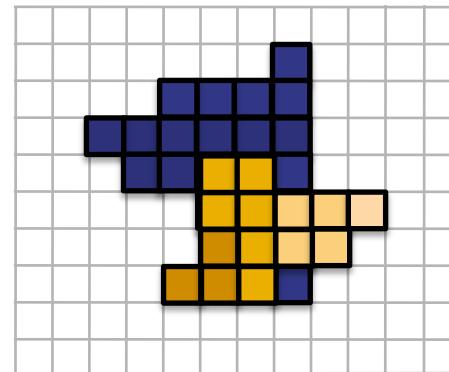
**Primitives**



**Fragments (shaded)**

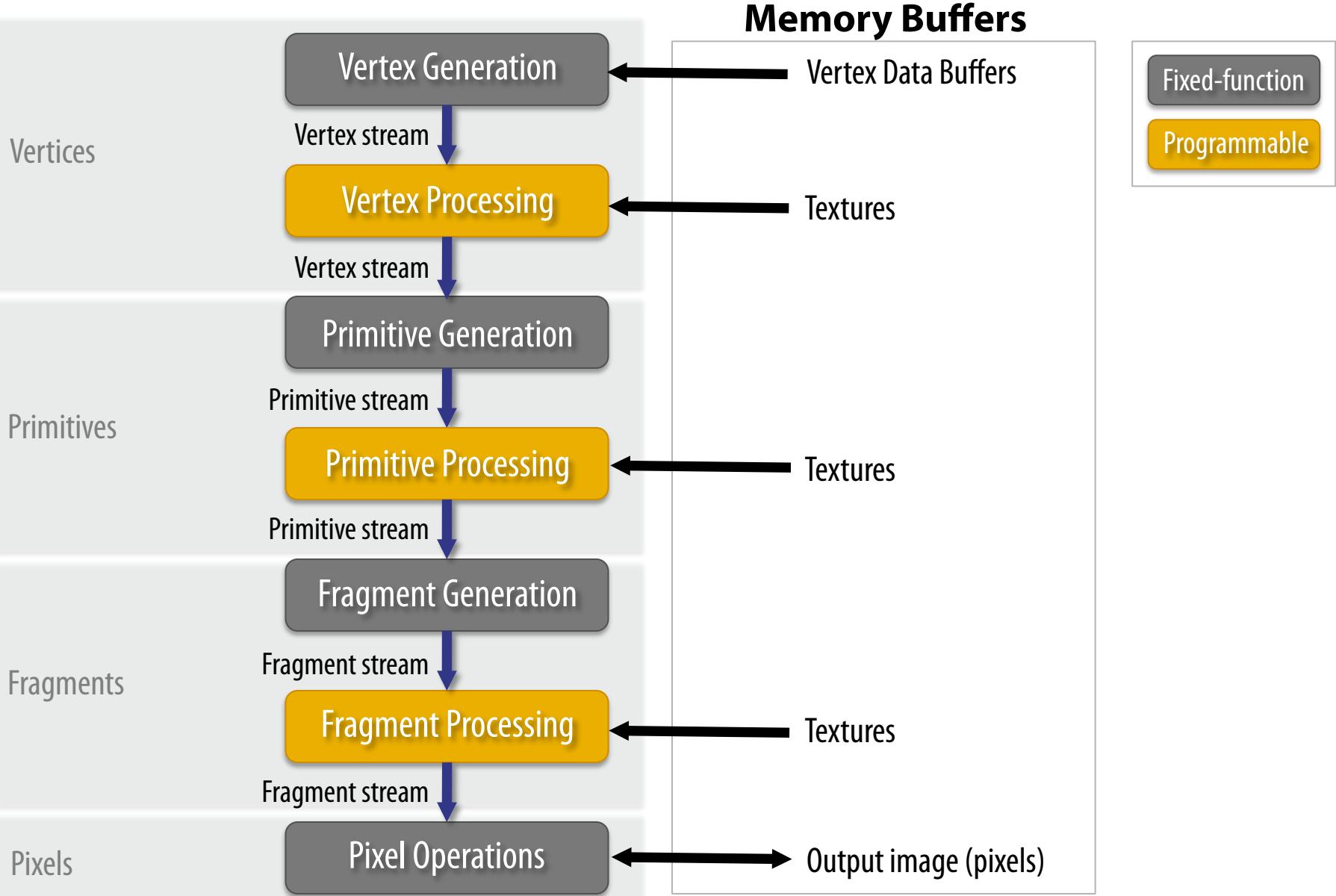


**Fragments  
(not yet shaded)**



**Pixels**

# Graphics pipeline

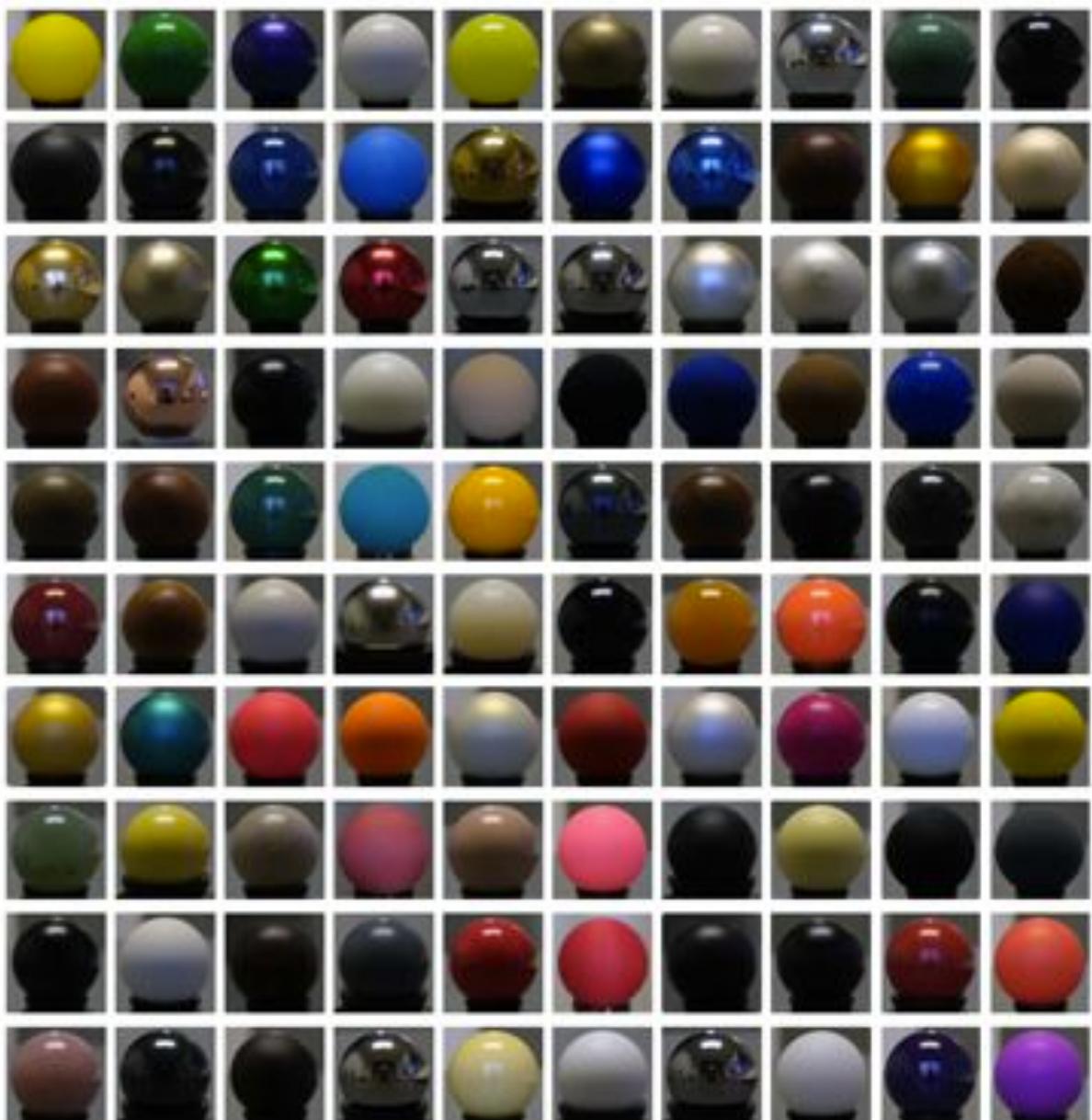


# Diversity in materials, surfaces, and lights



# More materials

---



Slide credit Pat Hanrahan

Images from Matusik et al. SIGGRAPH 2003

---

Part 2:  
**Graphics architectures**

**(implementations of the graphics pipeline)**

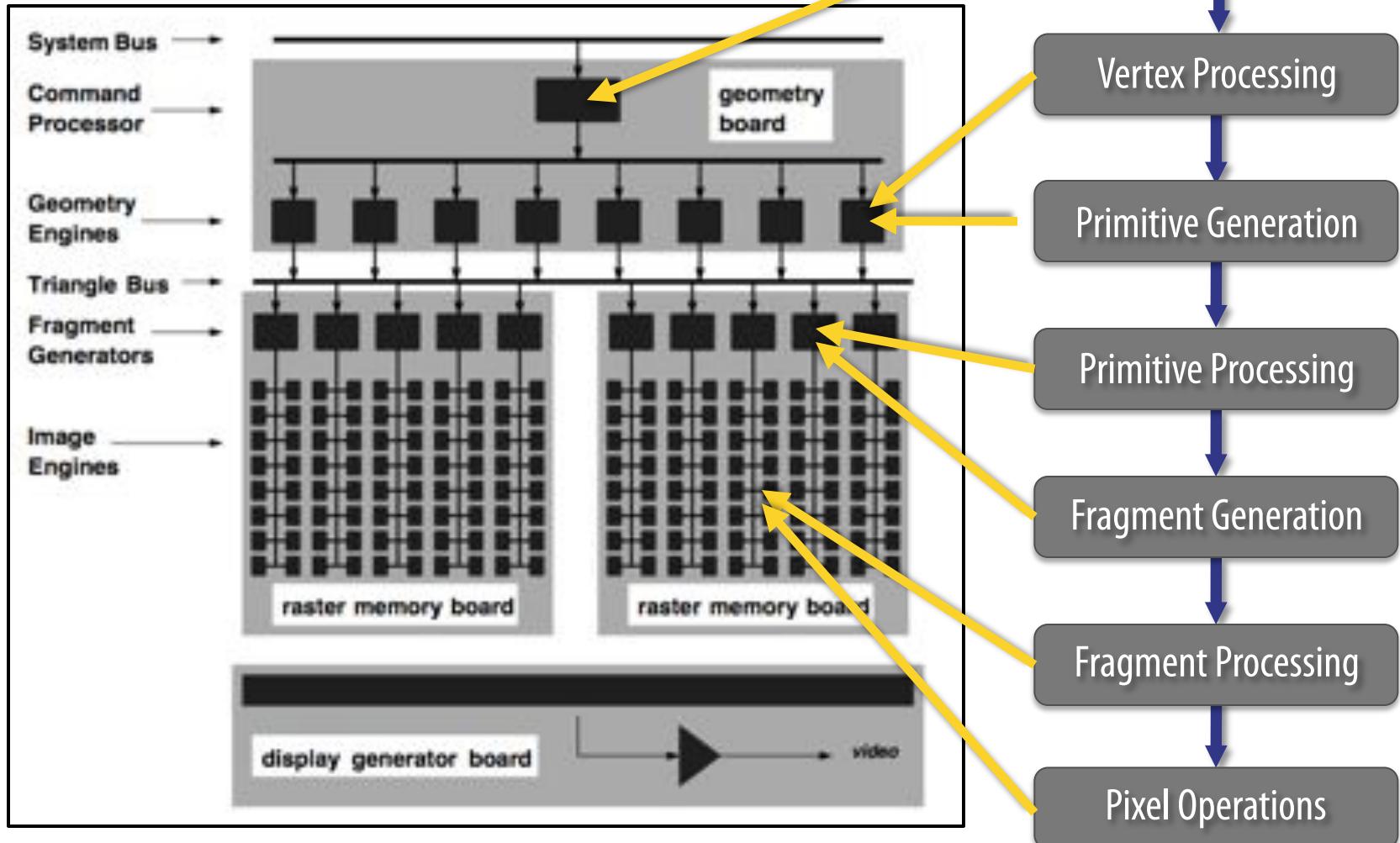
# Independent

---

- What's so important about “independent” computations?

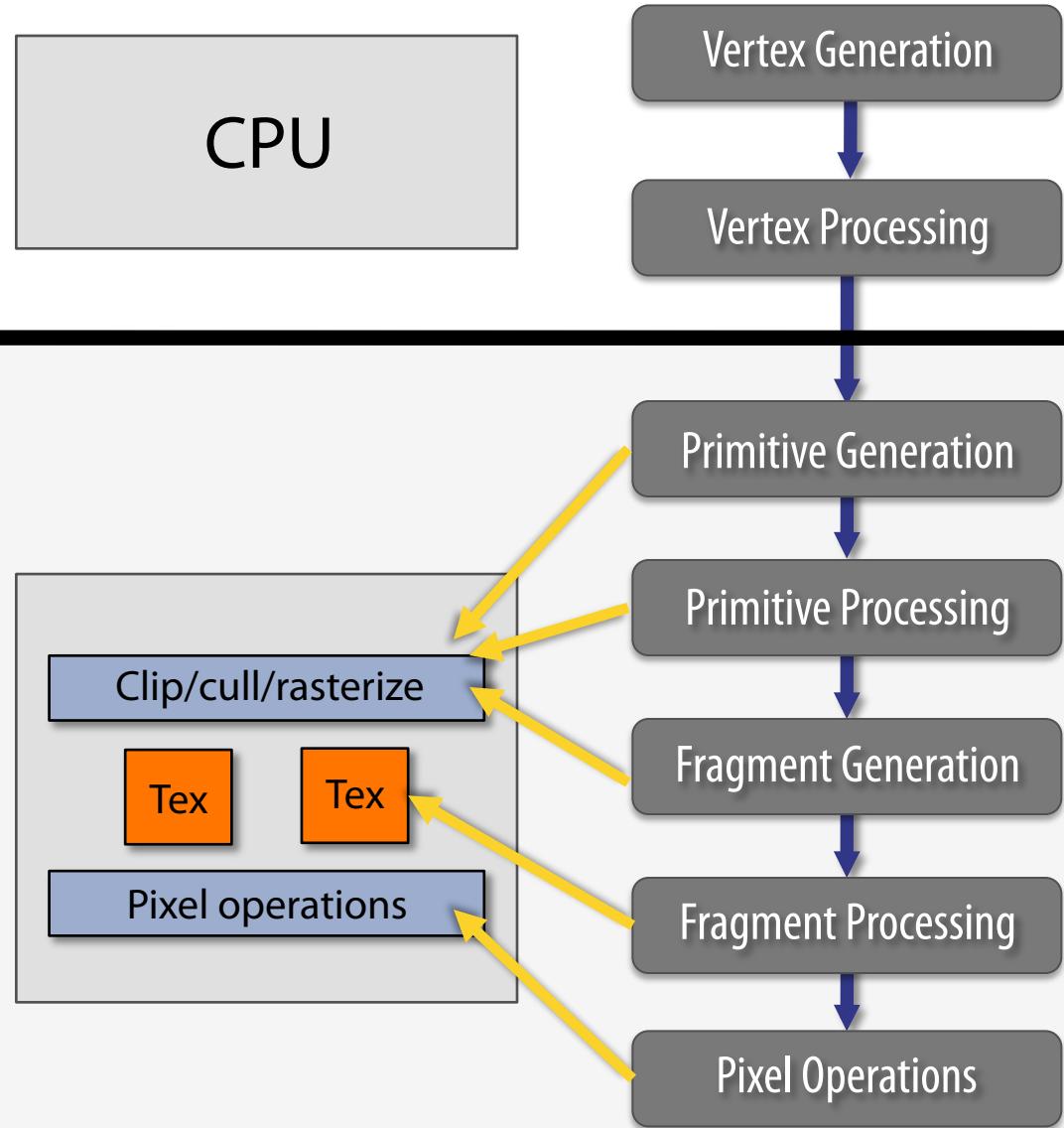
# Silicon Graphics RealityEngine (1993)

“graphics supercomputer”

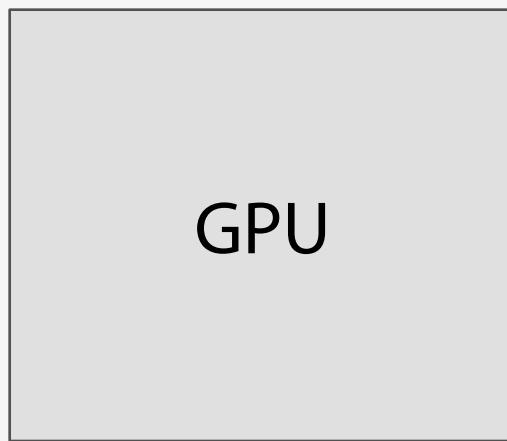


# Pre-1999 PC 3D graphics accelerator

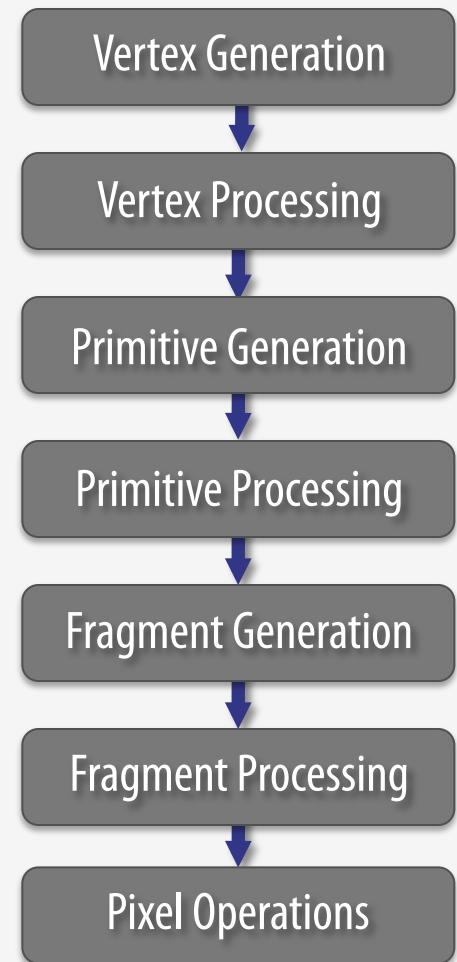
**3dfx Voodoo**  
**NVIDIA RIVA TNT**



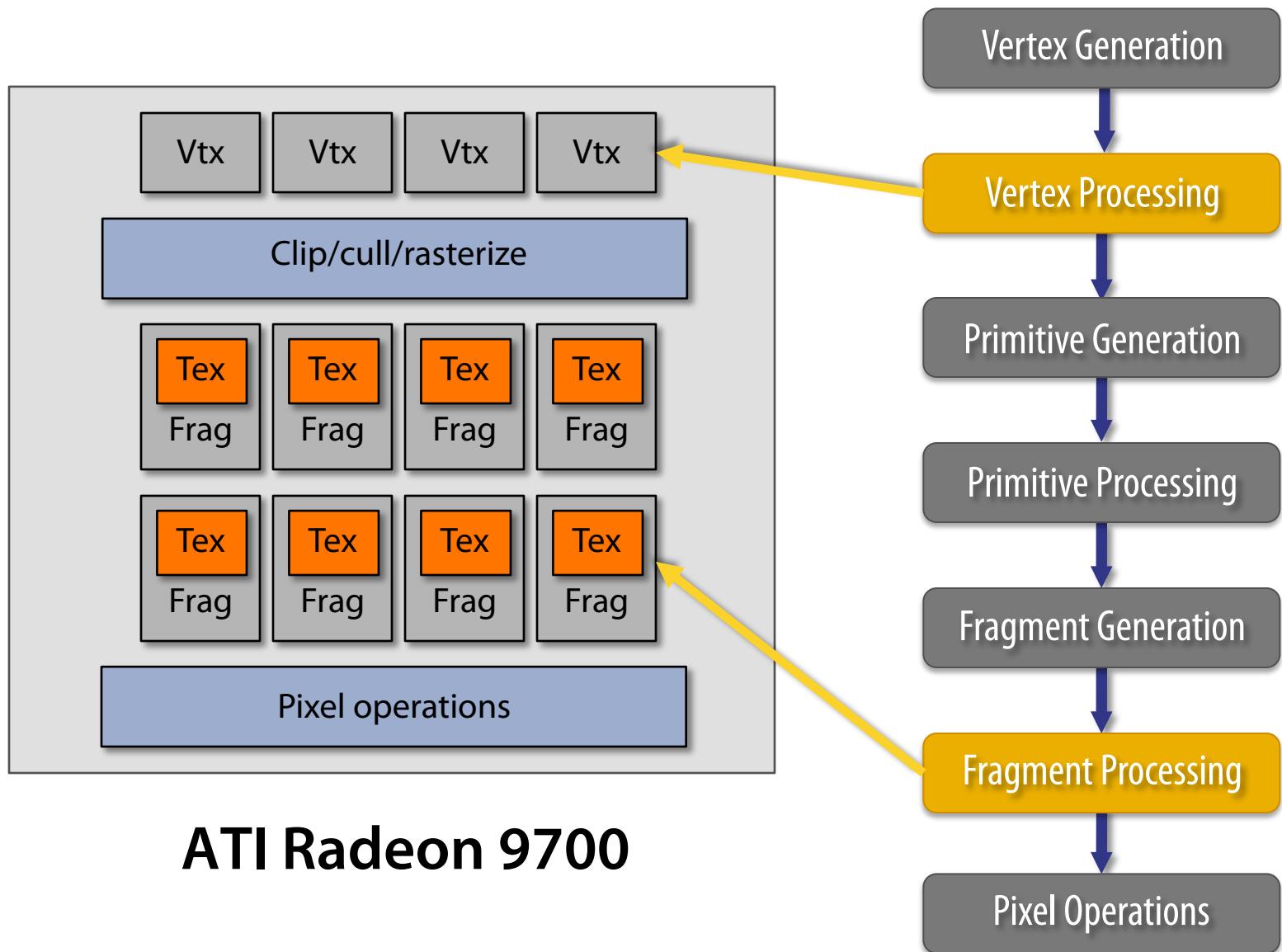
# GPU\* circa 1999



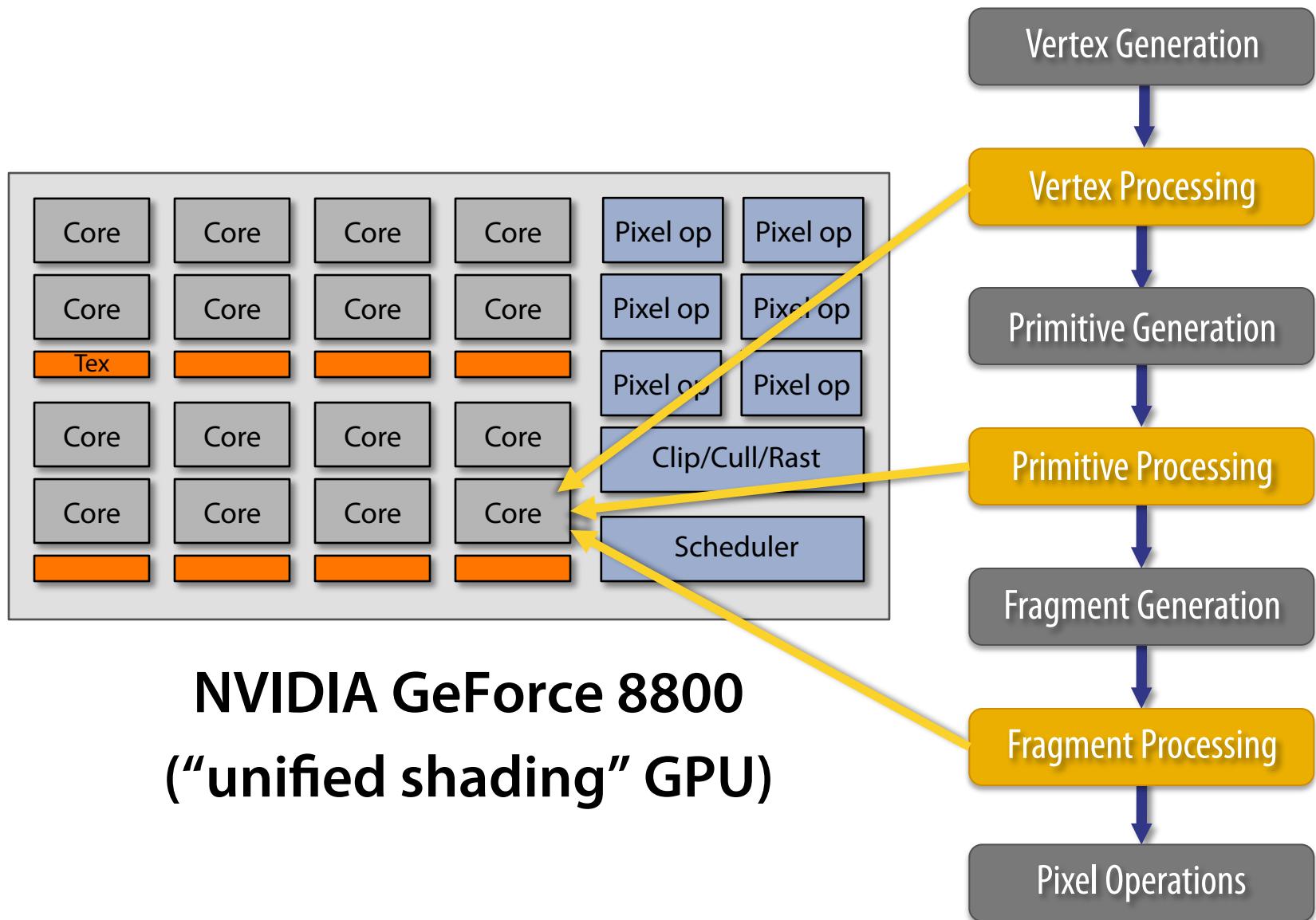
**NVIDIA GeForce 256**



# Direct3D 9 programmability: 2002

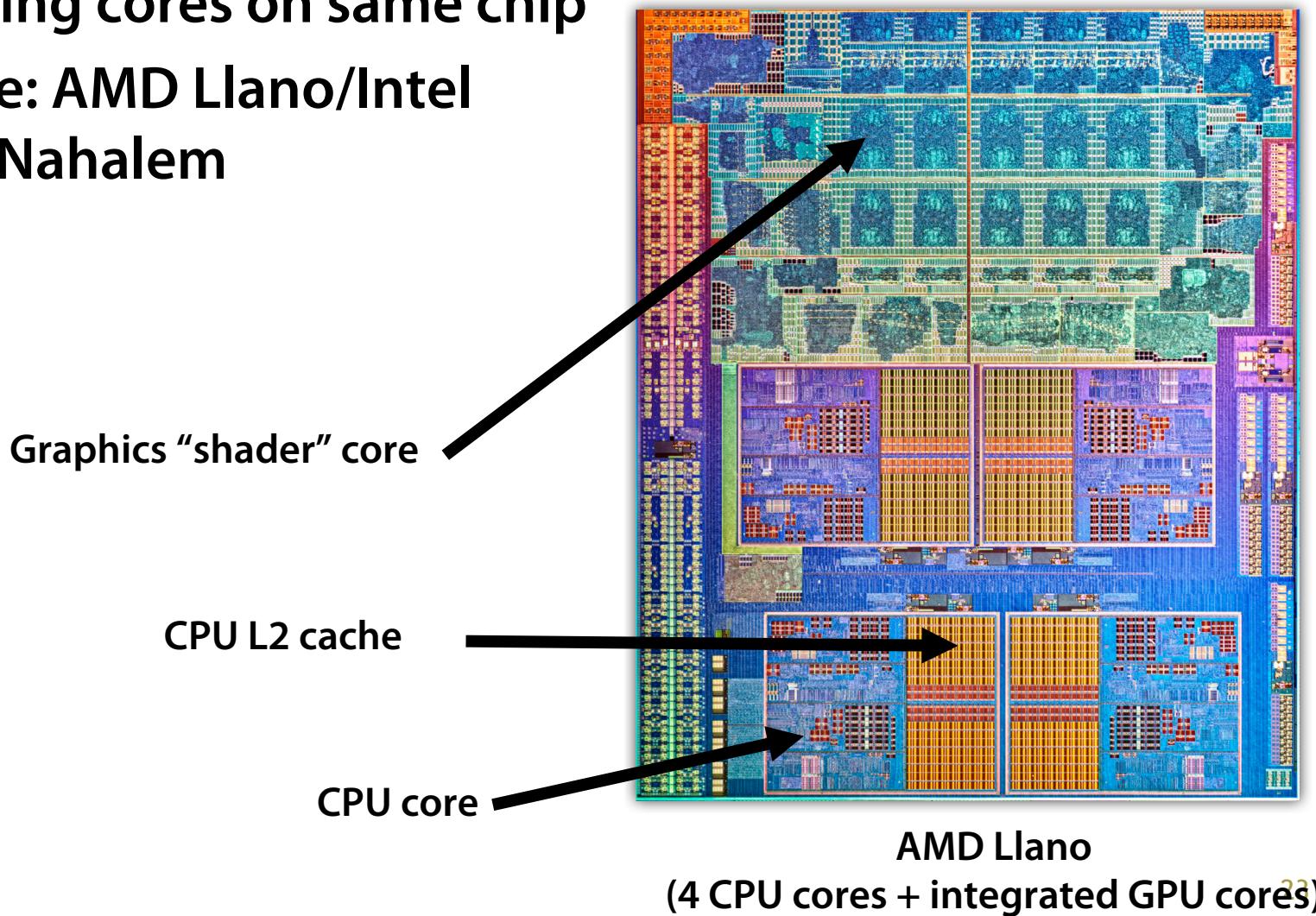


# Direct3D 10 programmability: 2006



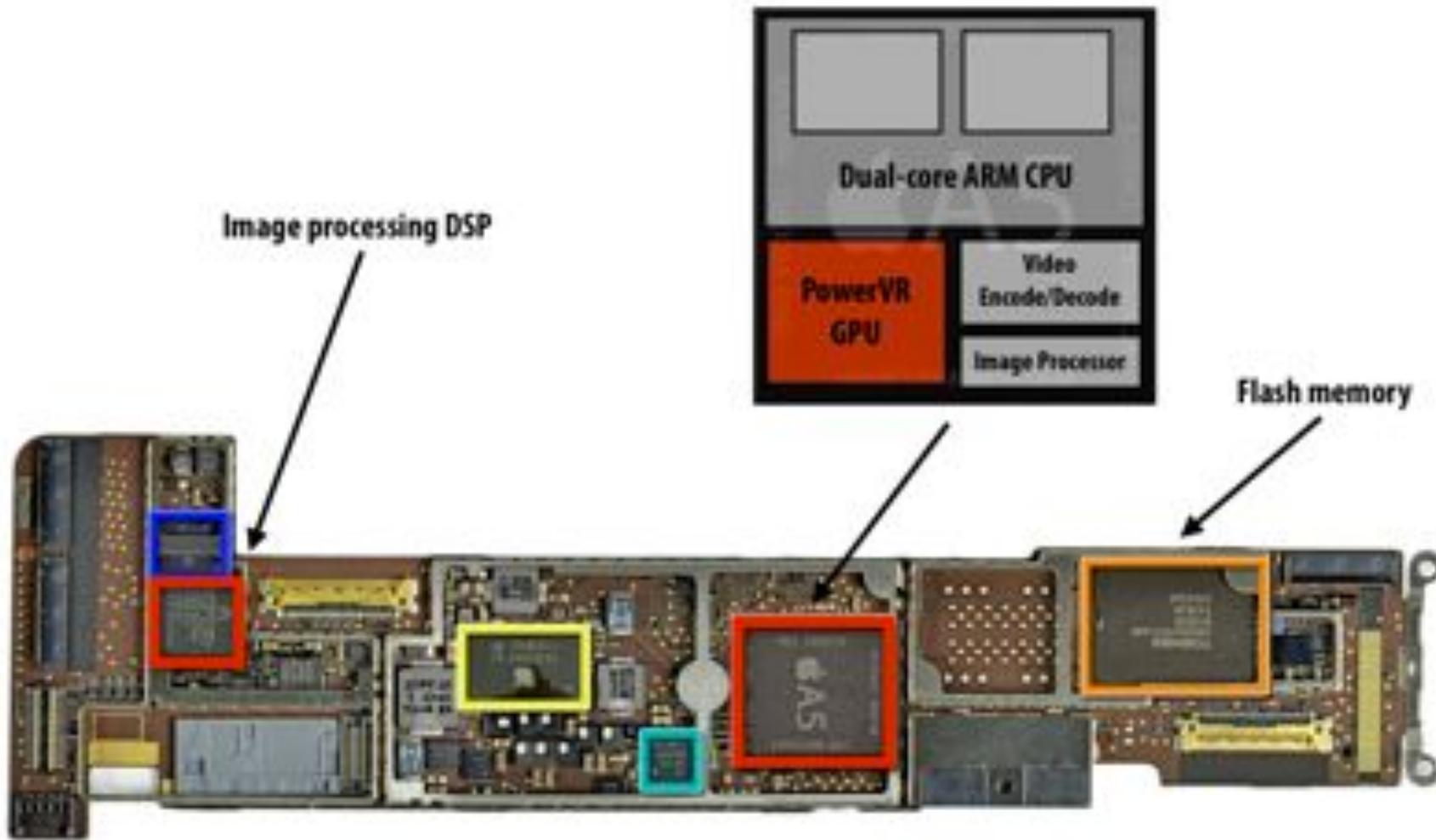
# Modern heterogeneous processor

- Combine CPUs and GPU processing cores on same chip
- Example: AMD Llano/Intel Core i7 Nahalem



# Heterogeneity is status quo in mobile

## Example: iPad 2



---

**Part 3:**  
How the GPU's shader processing cores work  
(how GPUs execute vertex, geometry, and  
fragment shader programs)

**(three key ideas)**

# A diffuse reflectance shader

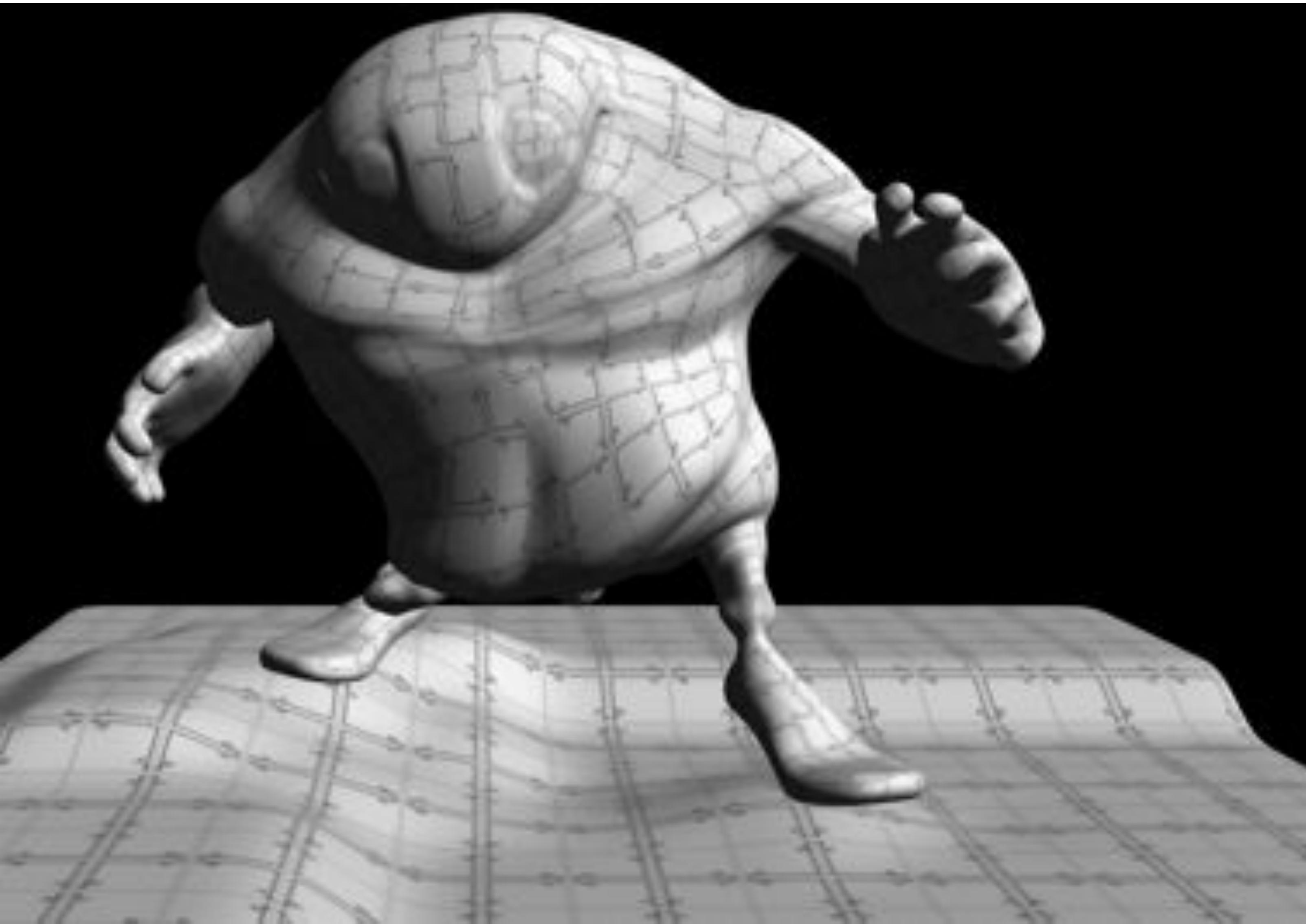
```
sampler mySamp;  
Texture2D<float3> myTex;  
float3 lightDir;  
  
float4 diffuseShader(float3 norm, float2 uv)  
{  
    float3 kd;  
    kd = myTex.Sample(mySamp, uv);  
    kd *= clamp( dot(lightDir, norm), 0.0, 1.0);  
    return float4(kd, 1.0);  
}
```

## Shader programming model:

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

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

# Big Guy, lookin' diffuse



# Compile shader

```
sampler mySamp;  
Texture2D<float3> myTex;  
float3 lightDir;  
  
float4 diffuseShader(float3 norm, float2 uv)  
{  
    float3 kd;  
    kd = myTex.Sample(mySamp, uv);  
    kd *= clamp( dot(lightDir, norm), 0.0, 1.0);  
    return float4(kd, 1.0);  
}
```

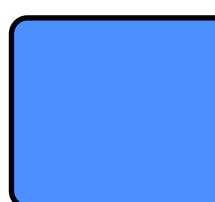
1 unshaded fragment input record



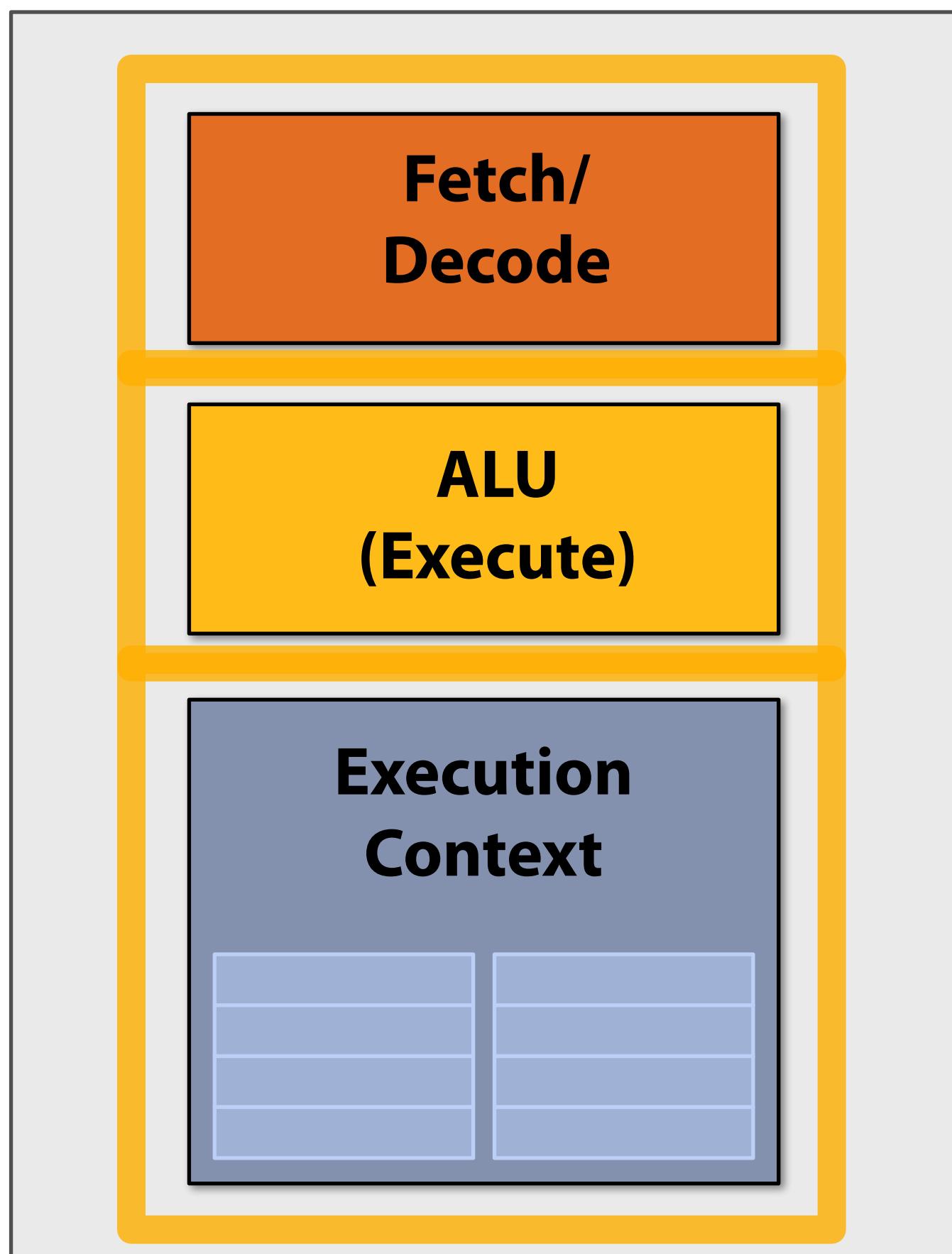
```
<diffuseShader>:  
sample r0, v4, t0, s0  
mul r3, v0, cb0[0]  
madd r3, v1, cb0[1], r3  
madd r3, v2, cb0[2], r3  
clmp r3, r3, l(0.0), l(1.0)  
mul o0, r0, r3  
mul o1, r1, r3  
mul o2, r2, r3  
mov o3, l(1.0)
```



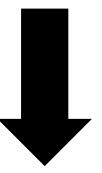
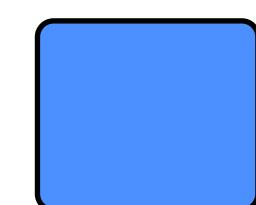
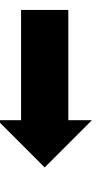
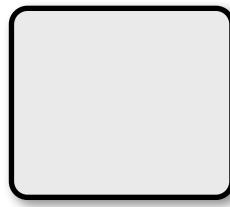
1 shaded fragment output record



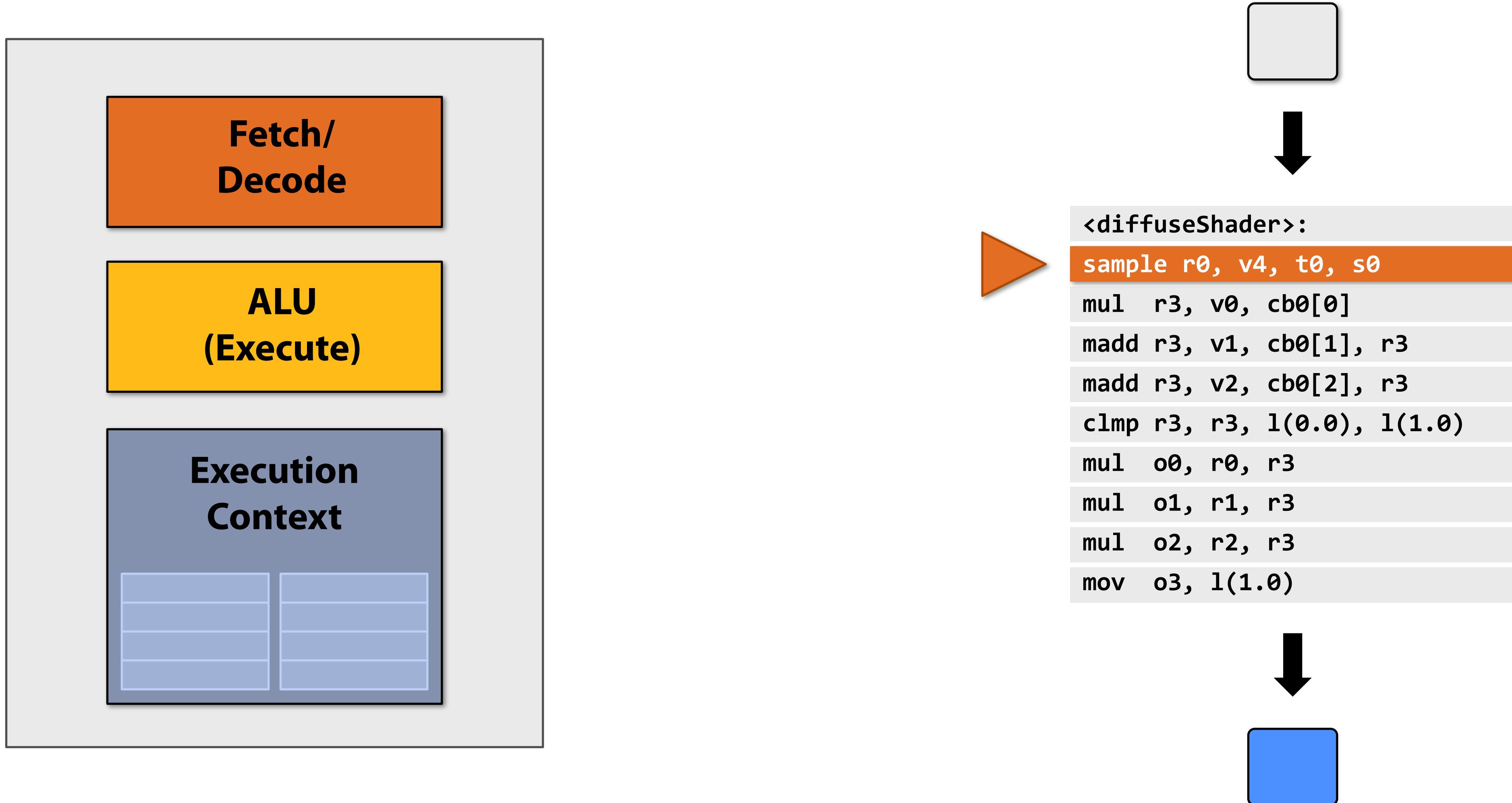
# Execute shader



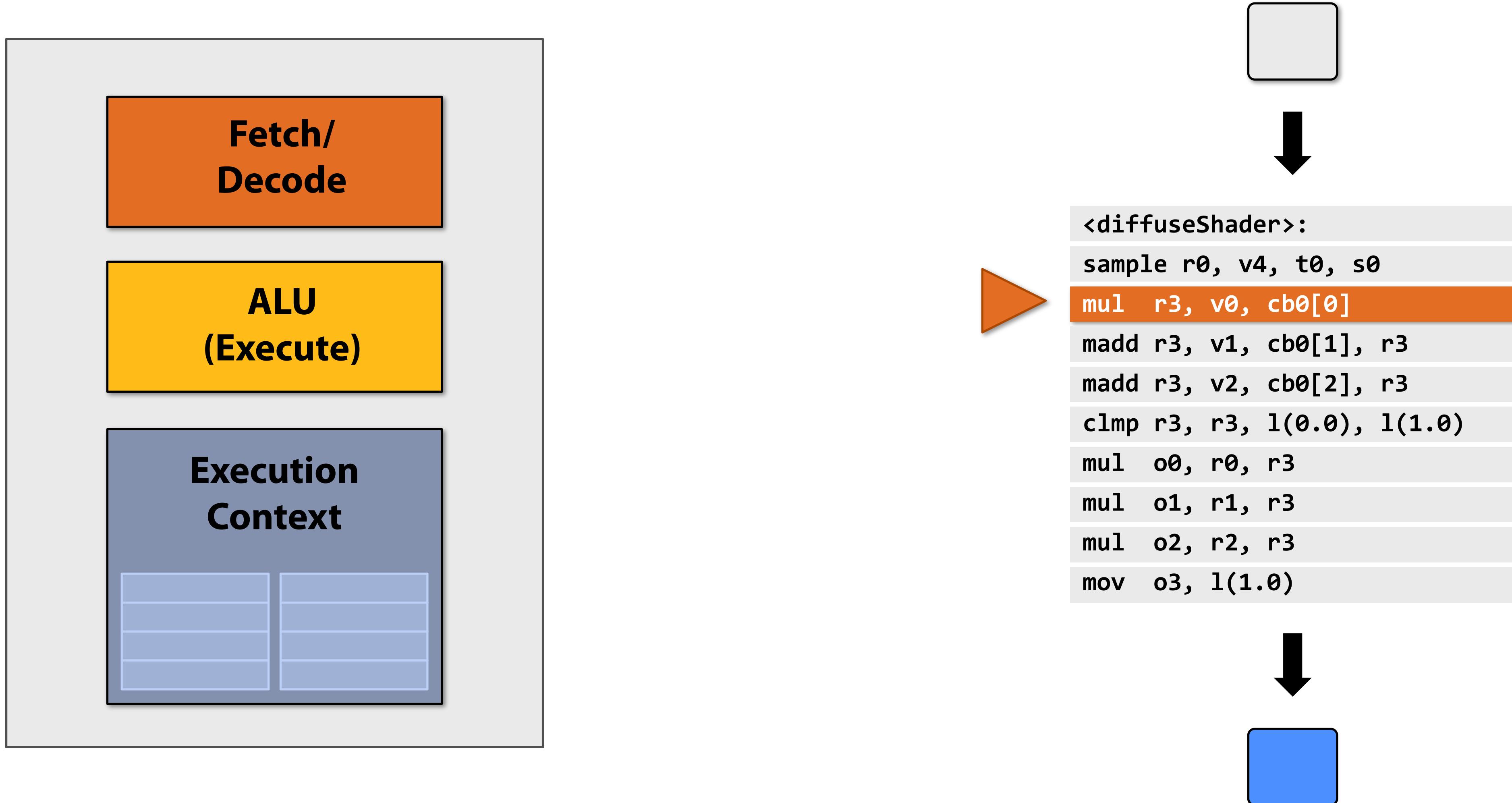
```
<diffuseShader>:  
sample r0, v4, t0, s0  
mul r3, v0, cb0[0]  
madd r3, v1, cb0[1], r3  
madd r3, v2, cb0[2], r3  
clmp r3, r3, l(0.0), l(1.0)  
mul o0, r0, r3  
mul o1, r1, r3  
mul o2, r2, r3  
mov o3, l(1.0)
```



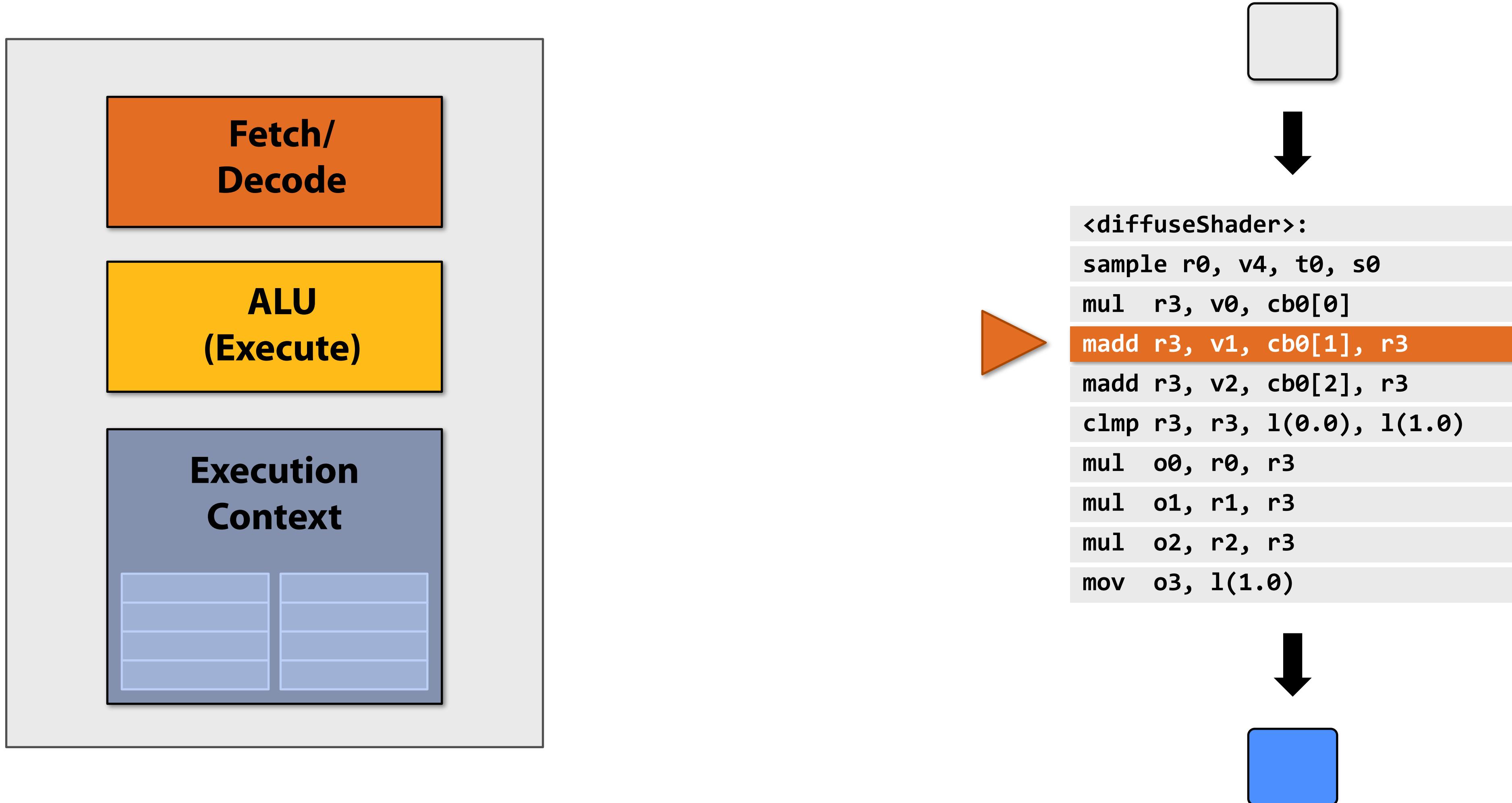
# Execute shader



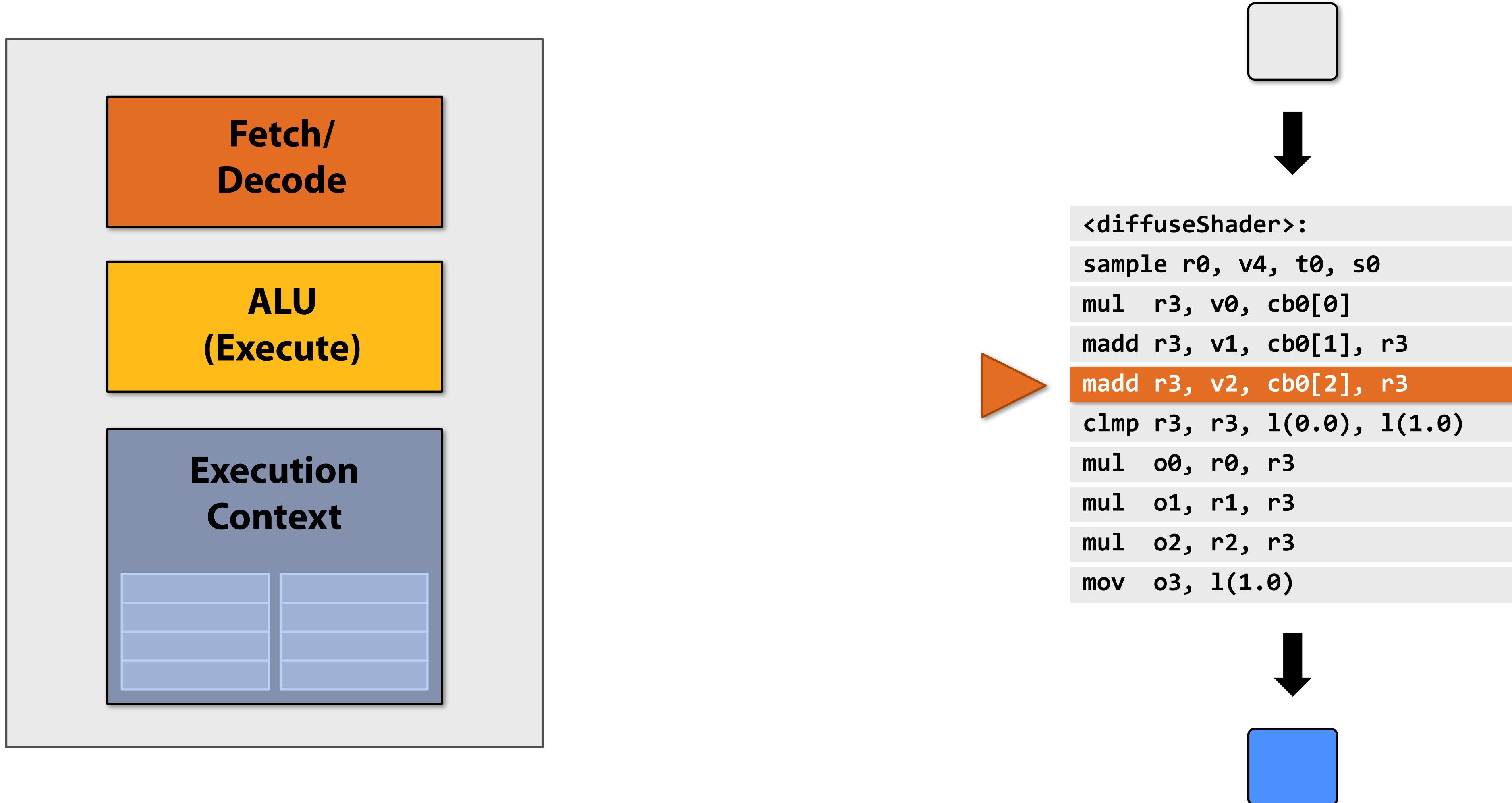
# Execute shader



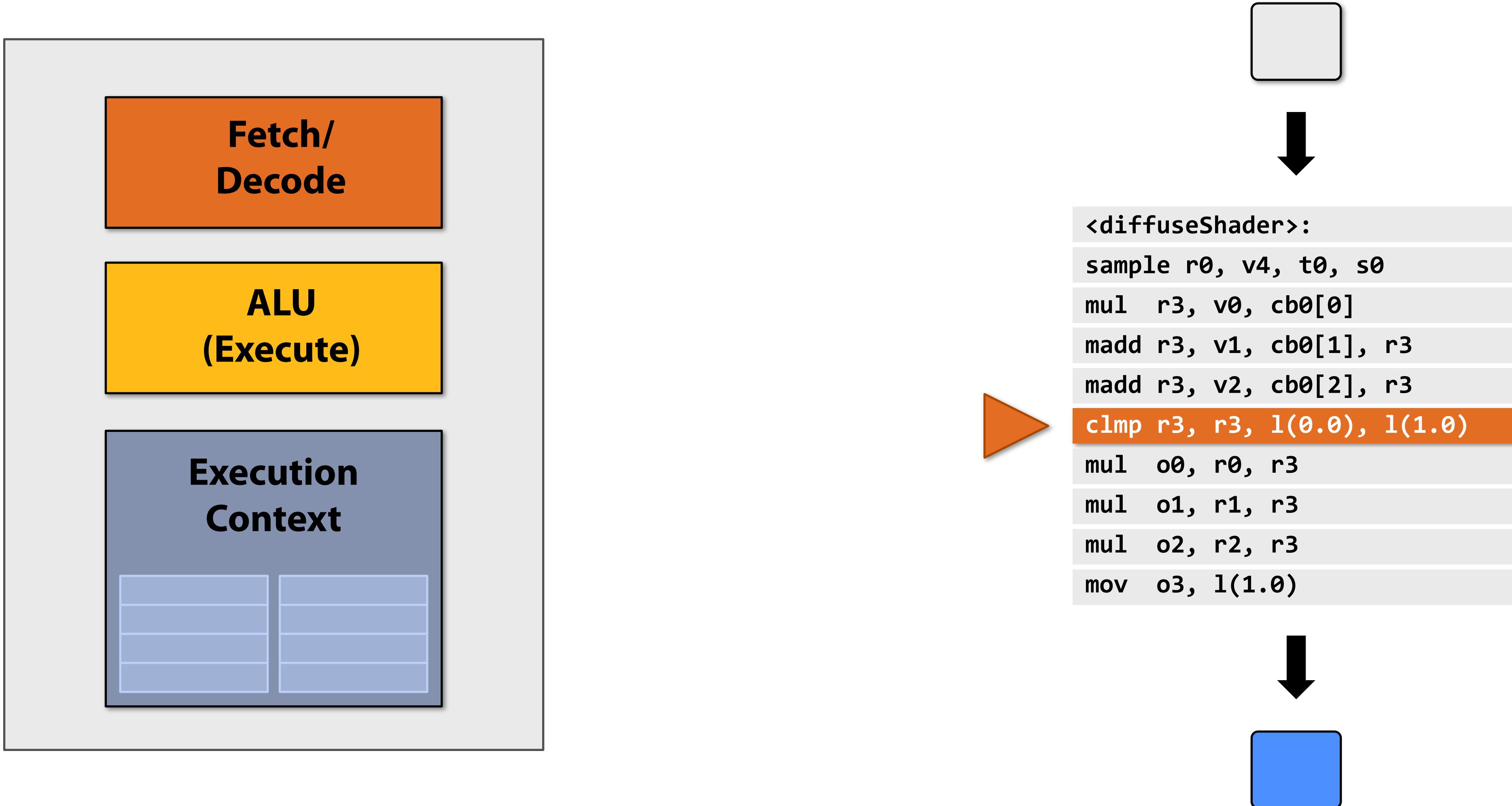
# Execute shader



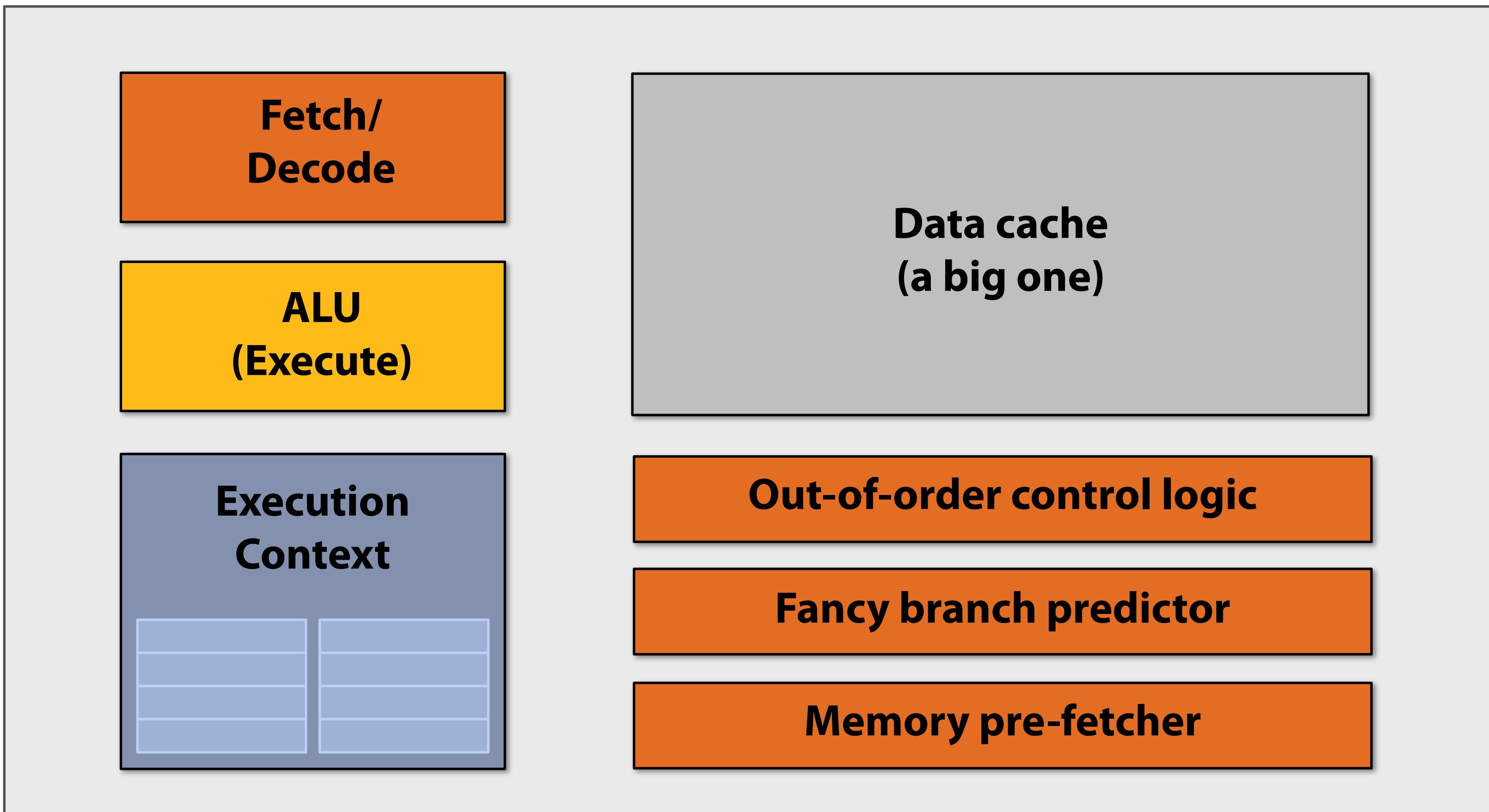
# Execute shader



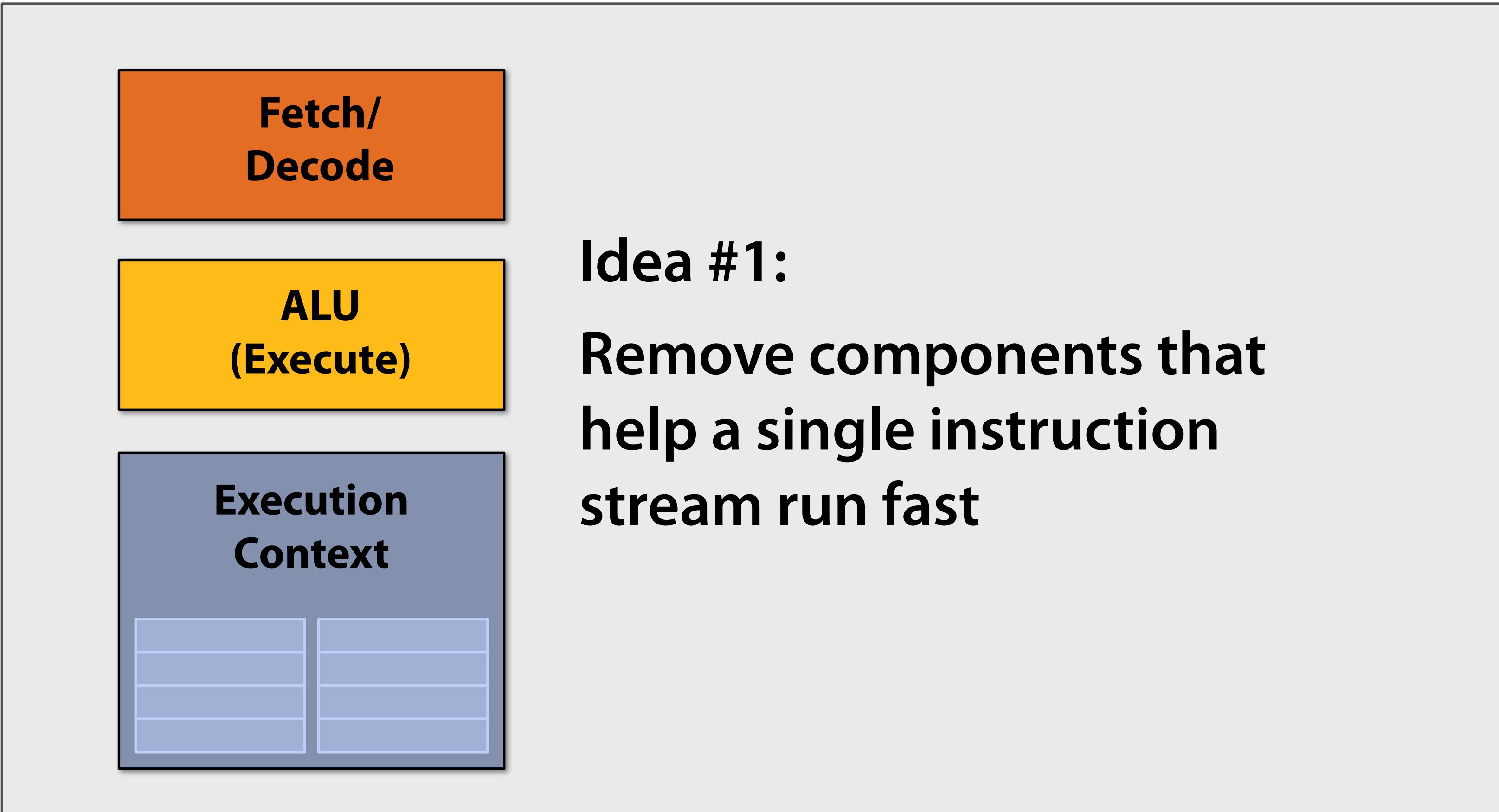
# Execute shader



# “CPU-style” cores

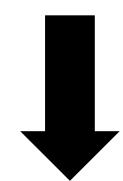
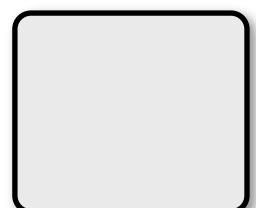


# **Slimming down**

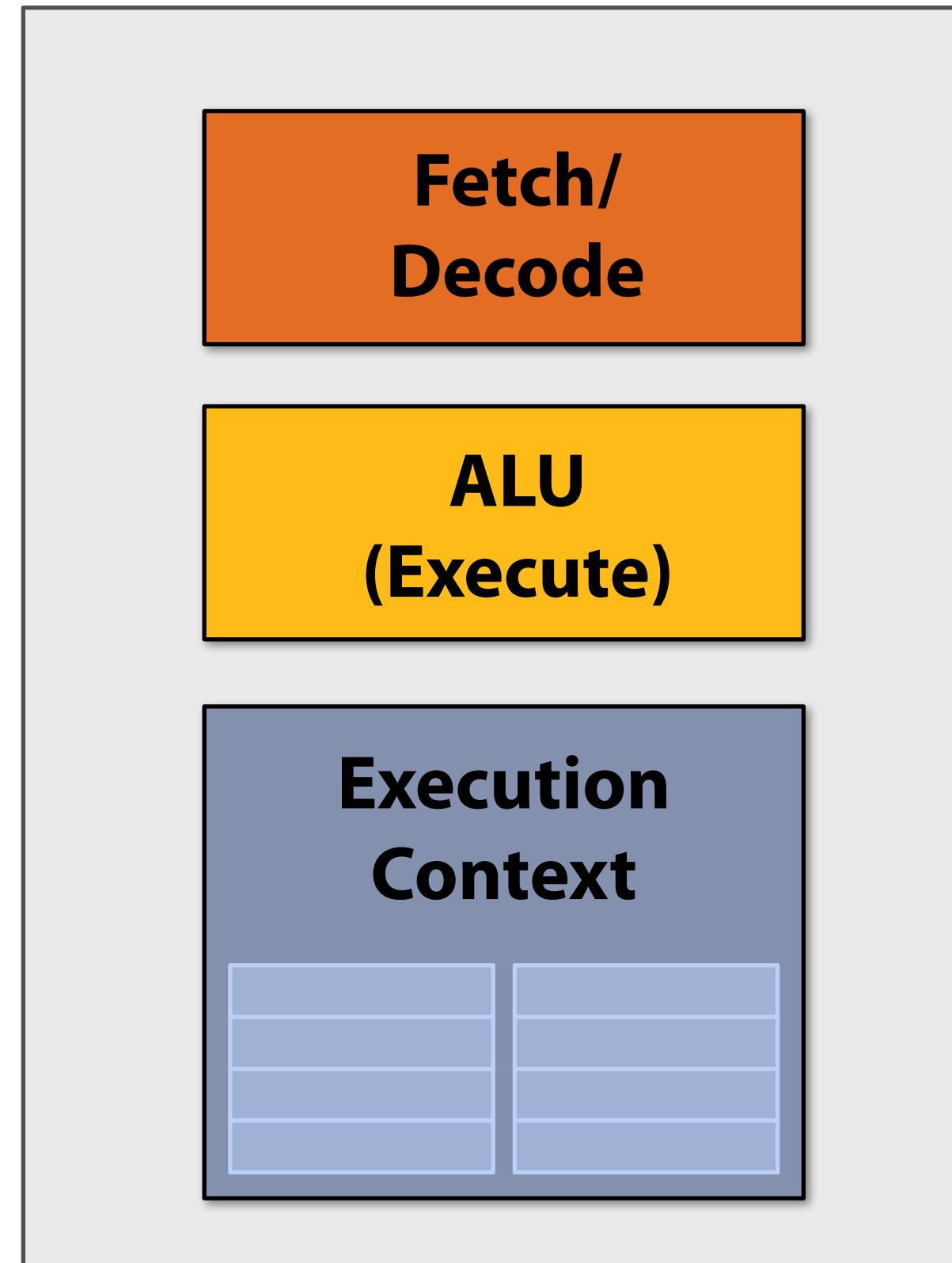


# Two cores (two fragments in parallel)

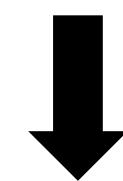
fragment 1



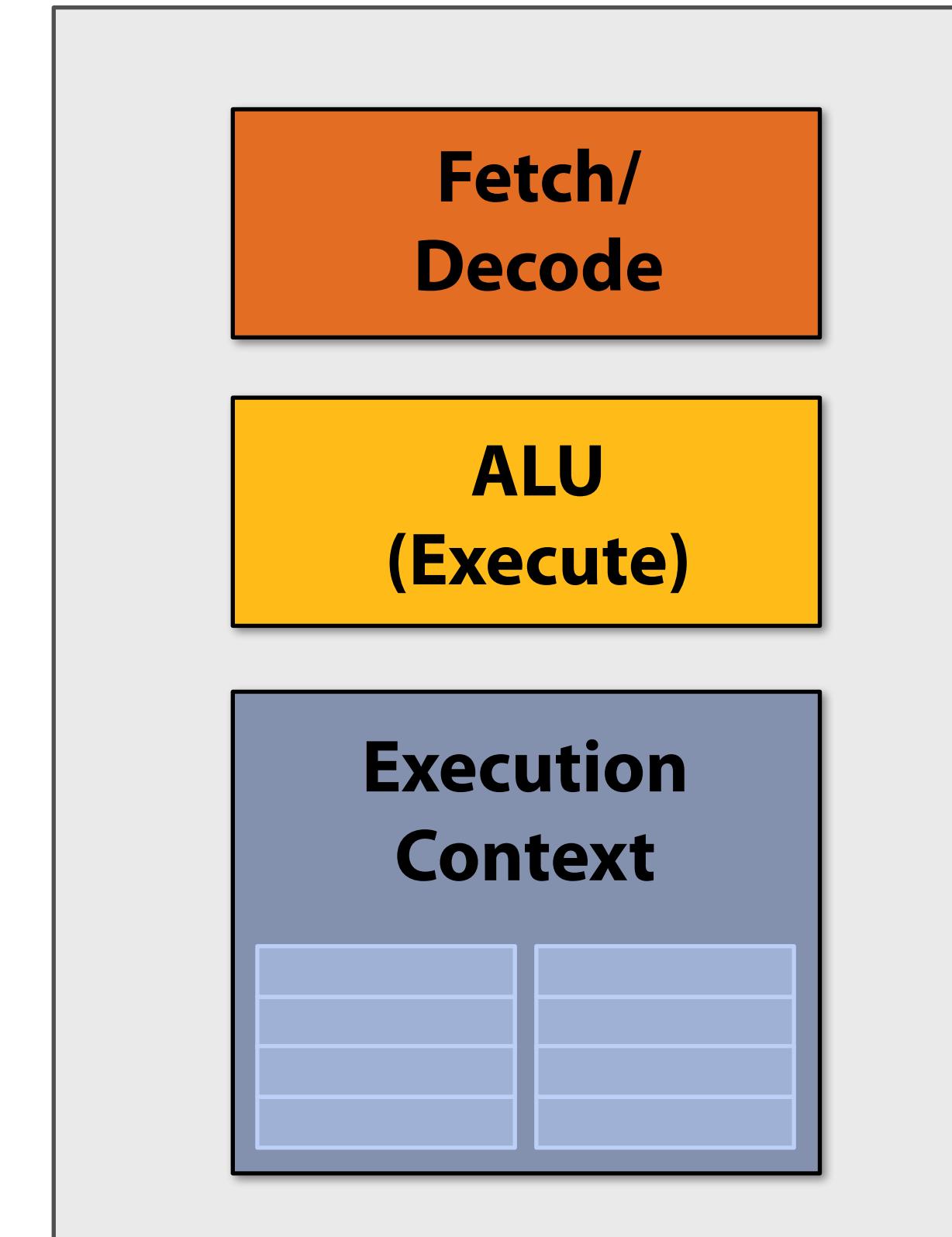
```
<diffuseShader>:  
sample r0, v4, t0, s0  
mul r3, v0, cb0[0]  
madd r3, v1, cb0[1], r3  
madd r3, v2, cb0[2], r3  
clmp r3, r3, 1(0.0), 1(1.0)  
mul o0, r0, r3  
mul o1, r1, r3  
mul o2, r2, r3  
mov o3, 1(1.0)
```



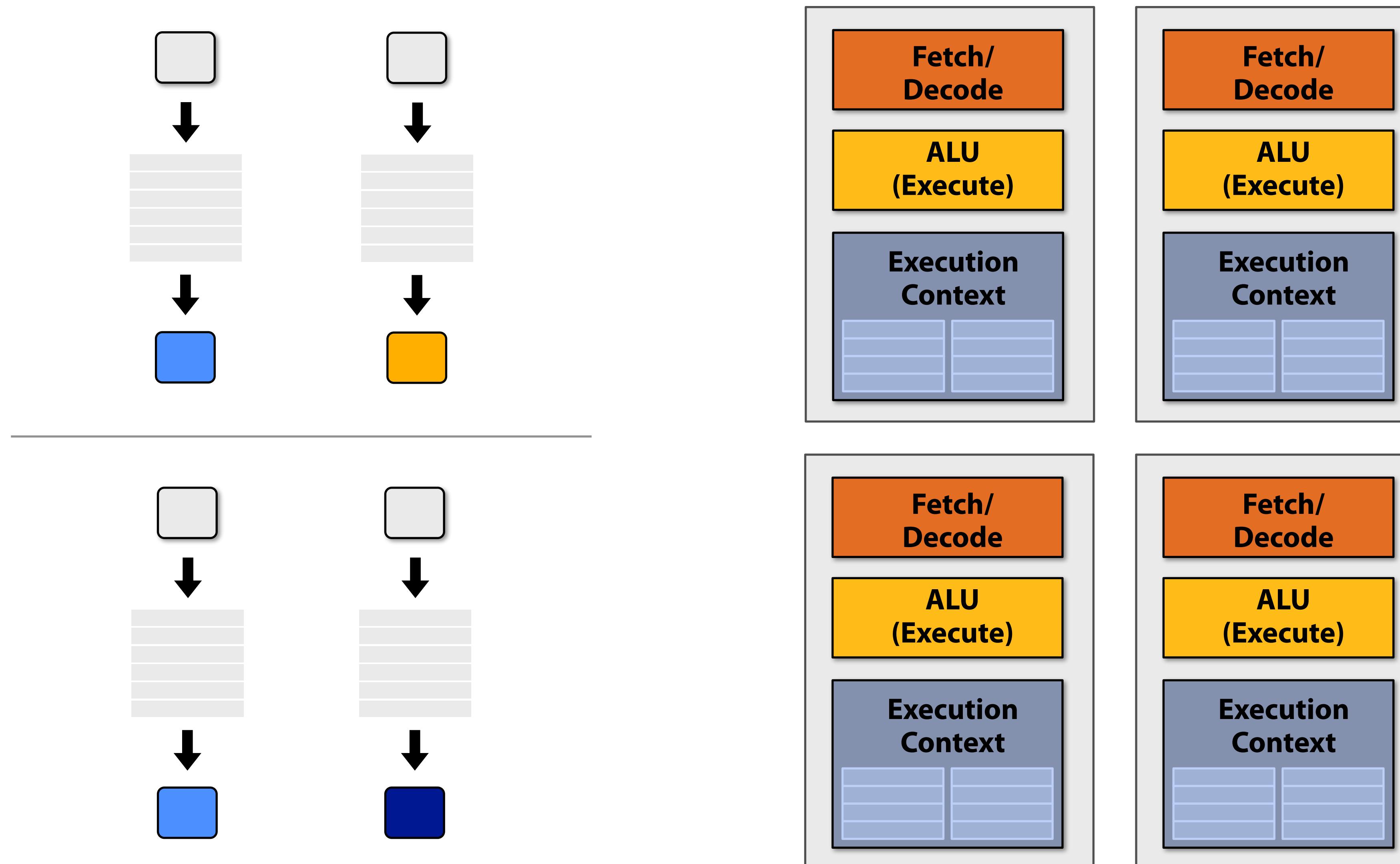
fragment 2



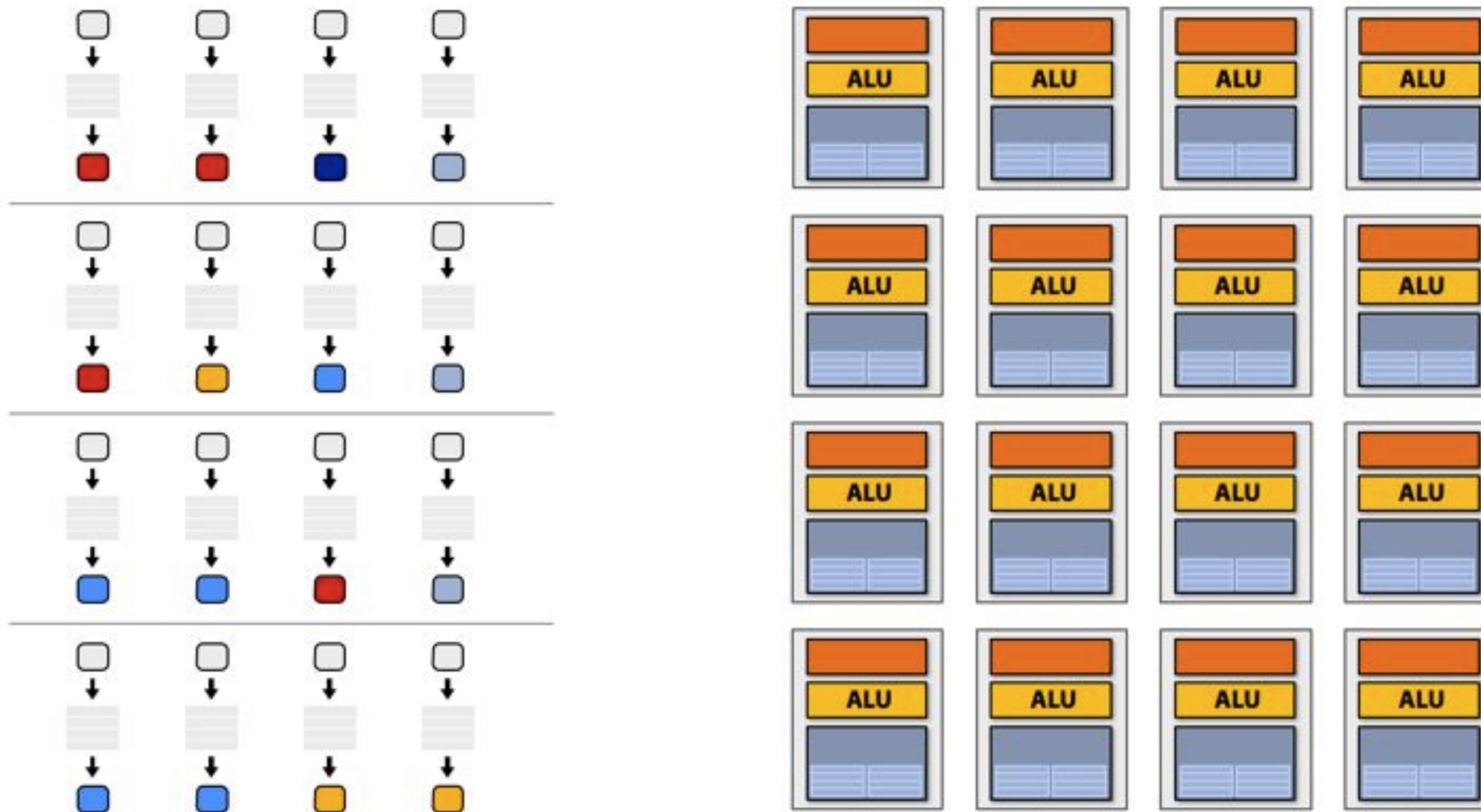
```
<diffuseShader>:  
sample r0, v4, t0, s0  
mul r3, v0, cb0[0]  
madd r3, v1, cb0[1], r3  
madd r3, v2, cb0[2], r3  
clmp r3, r3, 1(0.0), 1(1.0)  
mul o0, r0, r3  
mul o1, r1, r3  
mul o2, r2, r3  
mov o3, 1(1.0)
```



# Four cores (four fragments in parallel)

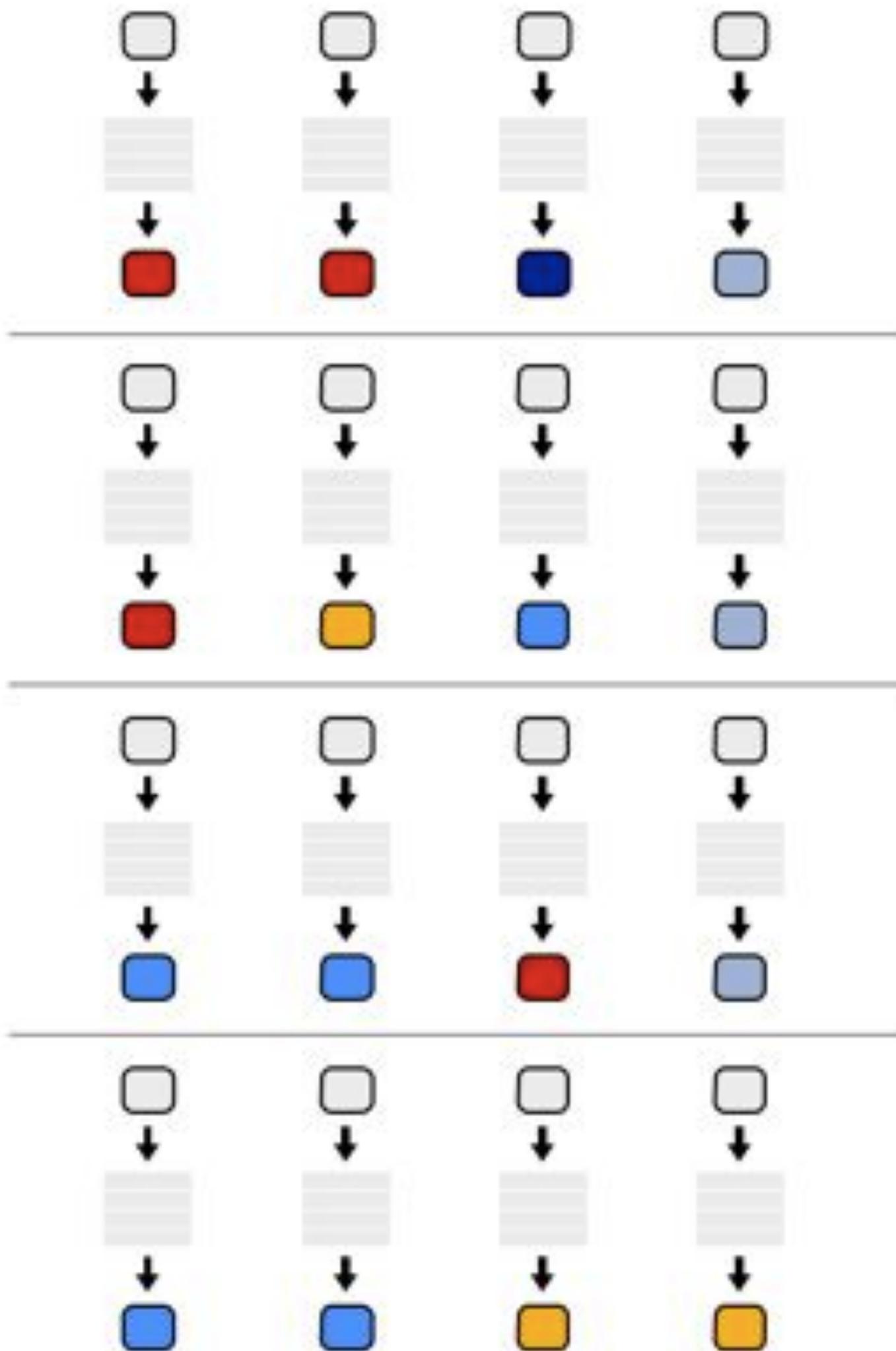


# Sixteen cores (sixteen fragments in parallel)



16 cores = 16 simultaneous instruction streams

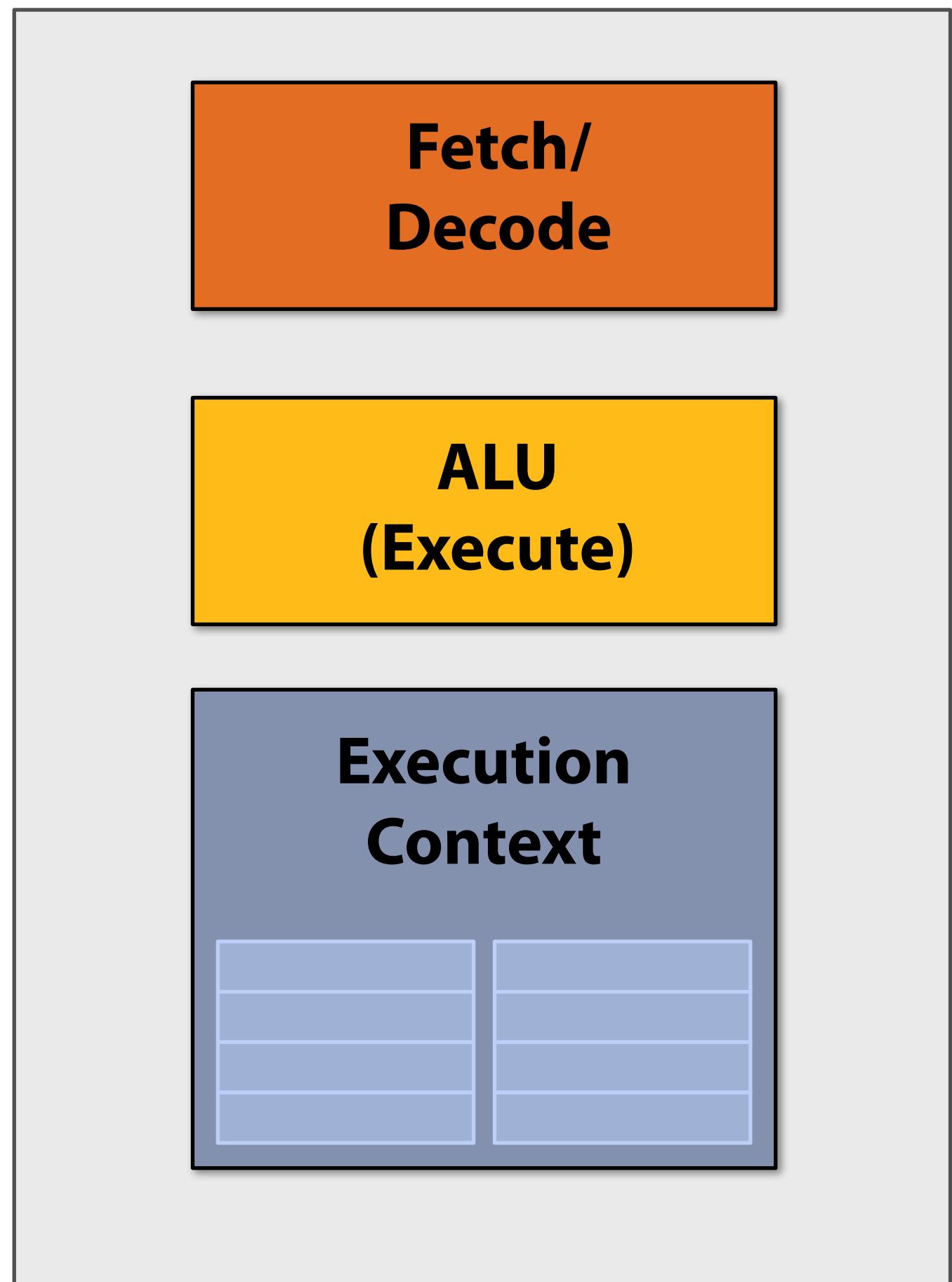
# Instruction stream sharing



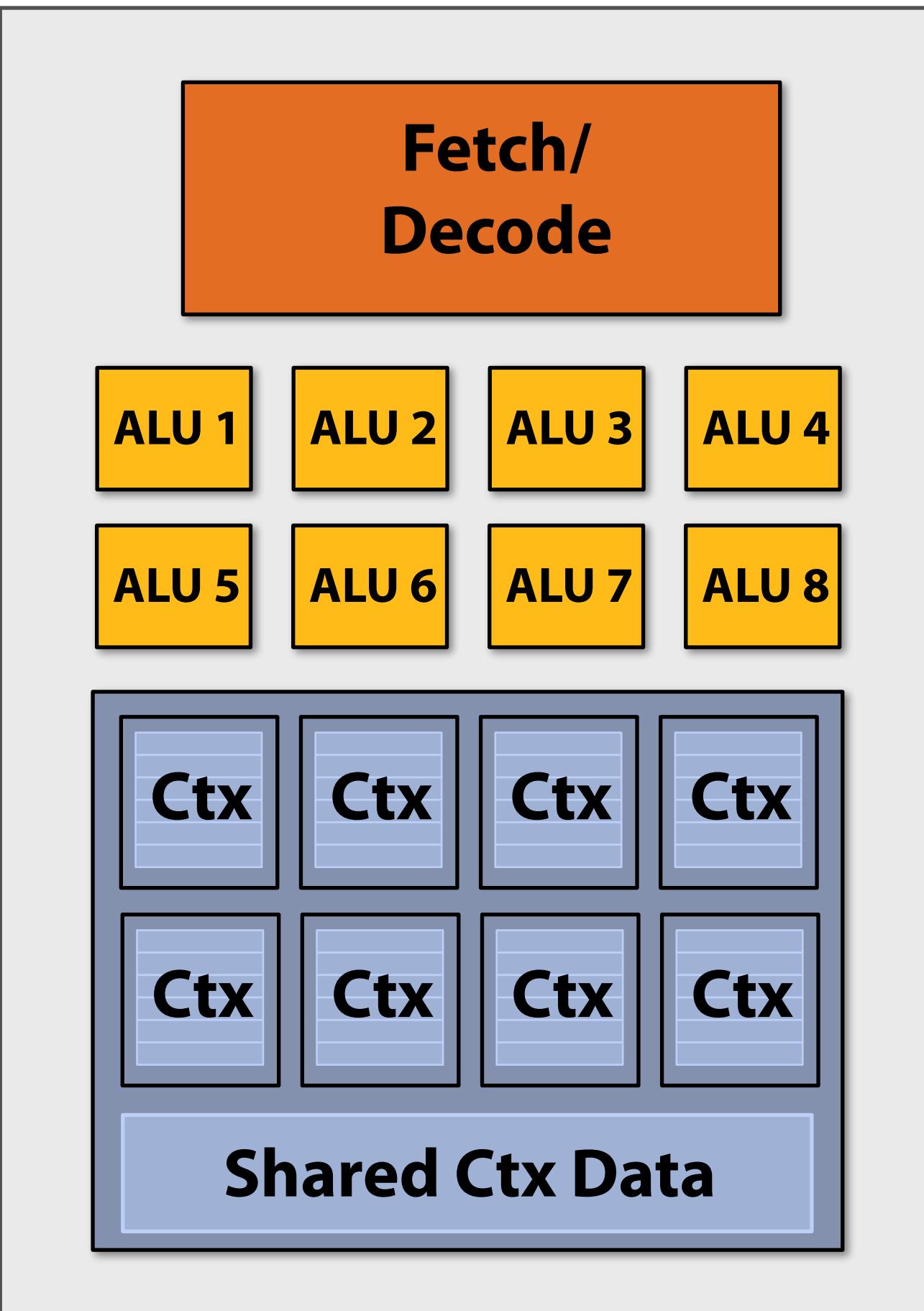
But ... many fragments  
should be able to share an  
instruction stream!

```
<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



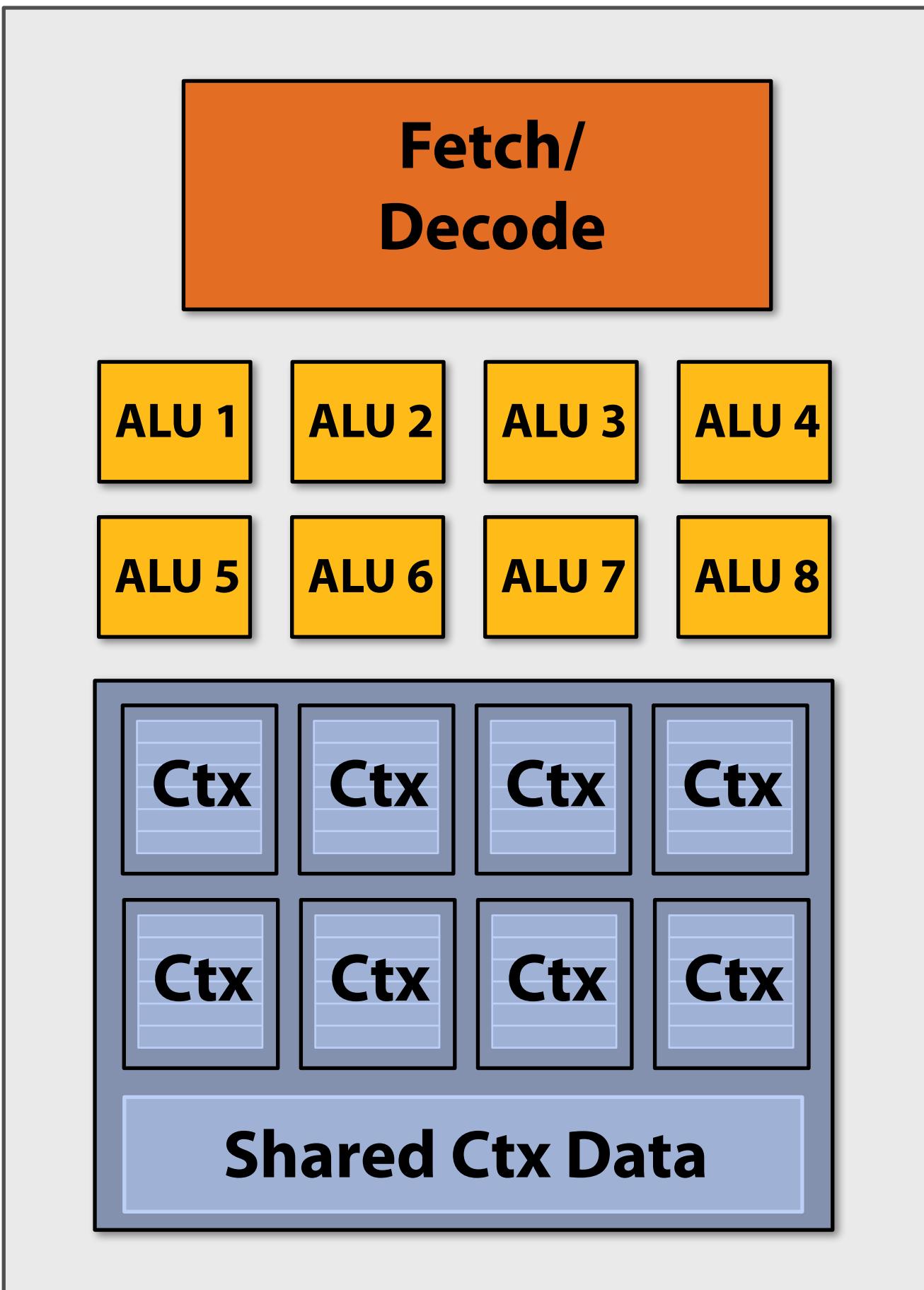
# Add ALUs



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

**SIMD processing**

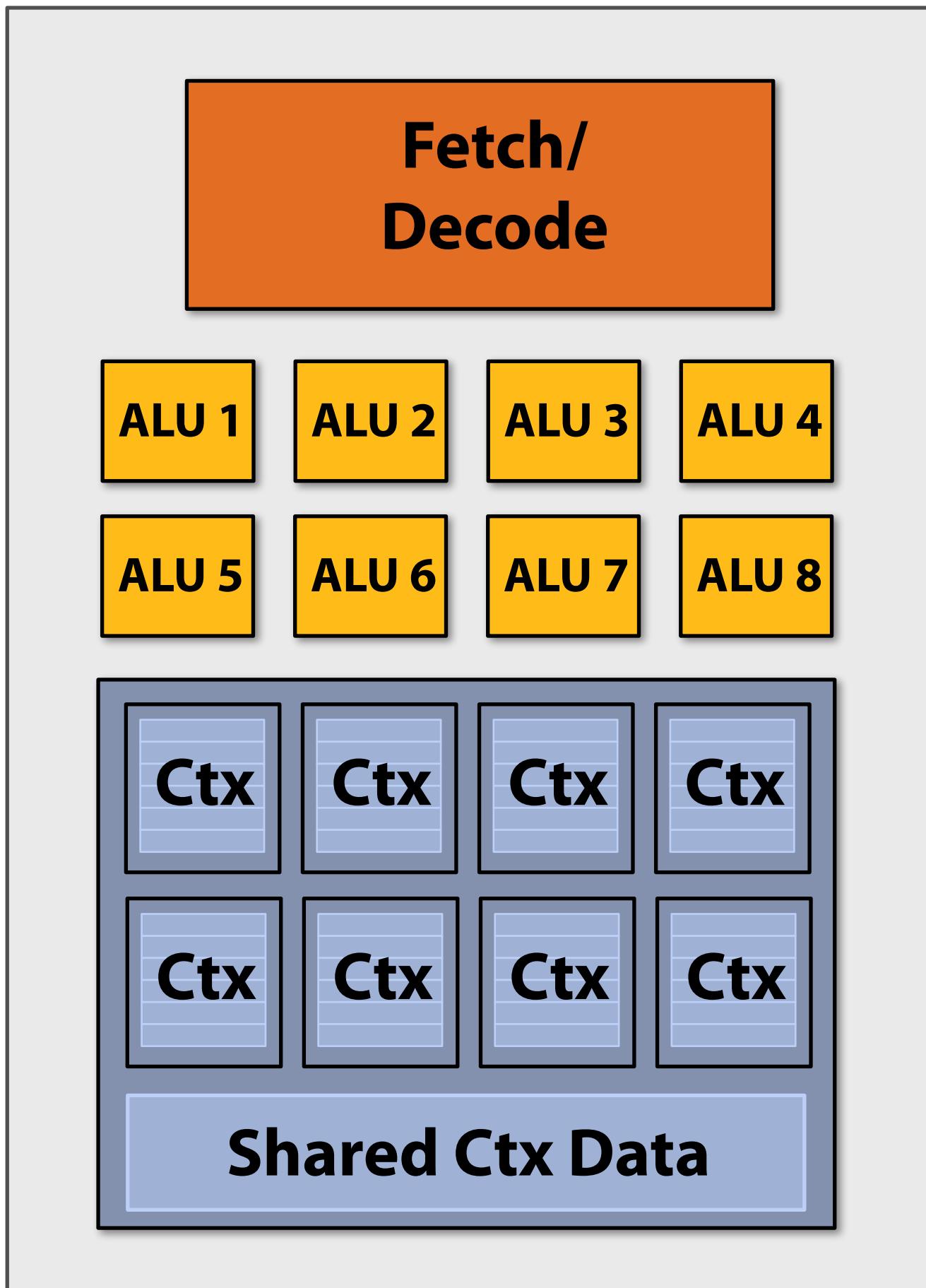
# Modifying the shader



```
<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, 1(0.0), 1(1.0)  
mul o0, r0, r3  
mul o1, r1, r3  
mul o2, r2, r3  
mov o3, 1(1.0)
```

**Original compiled shader:**  
**Processes one fragment using scalar ops on scalar registers**

# Modifying the shader

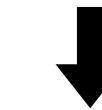
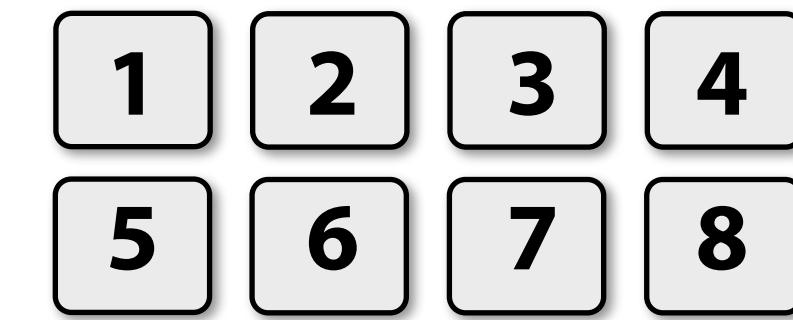
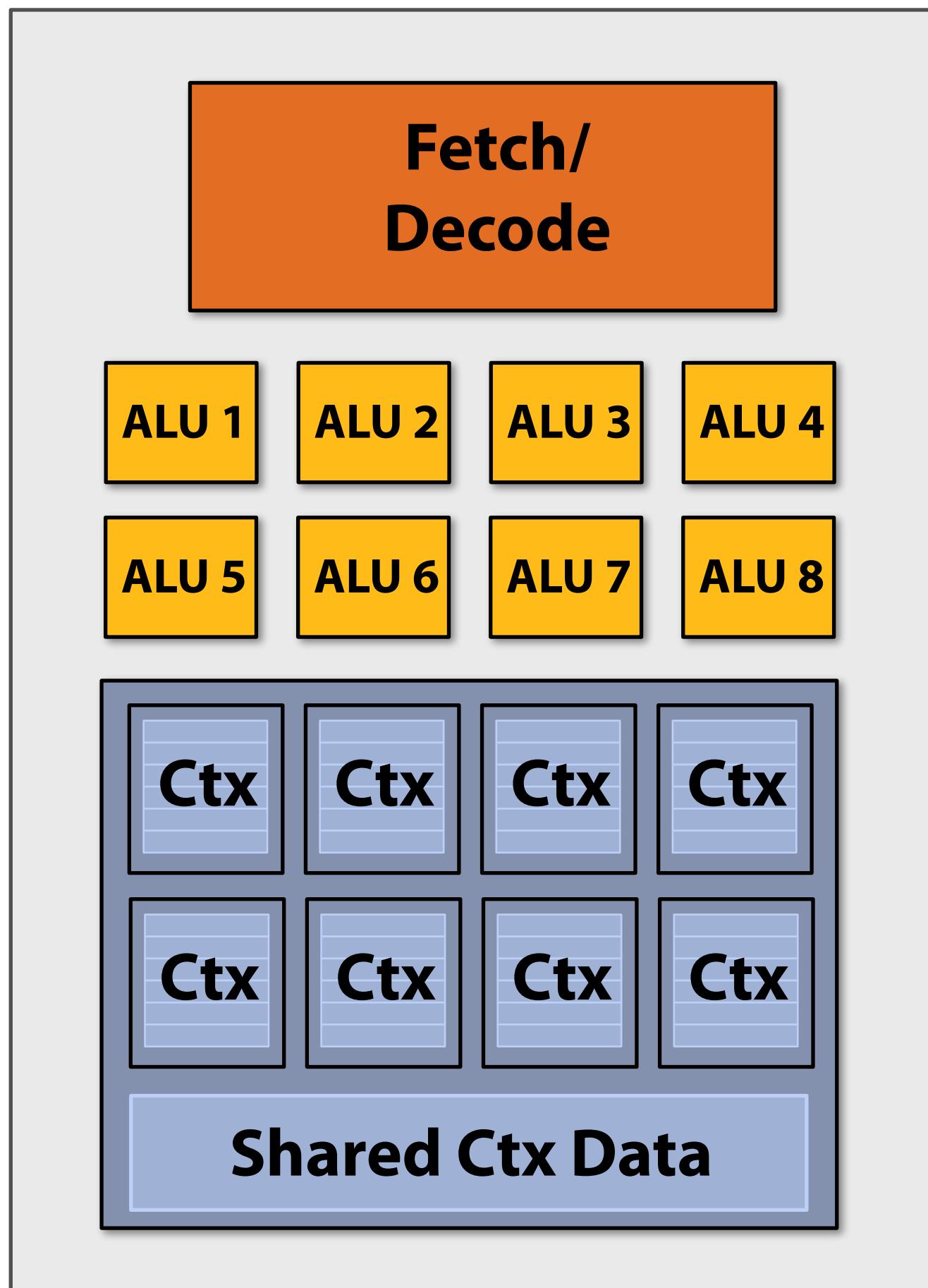


```
<VEC8_diffuseShader>:  
    VEC8_sample vec_r0, vec_v4, t0, vec_s0  
    VEC8_mul   vec_r3, vec_v0, cb0[0]  
    VEC8_madd  vec_r3, vec_v1, cb0[1], vec_r3  
    VEC8_madd  vec_r3, vec_v2, cb0[2], vec_r3  
    VEC8_clmp  vec_r3, vec_r3, l(0.0), l(1.0)  
    VEC8_mul   vec_o0, vec_r0, vec_r3  
   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

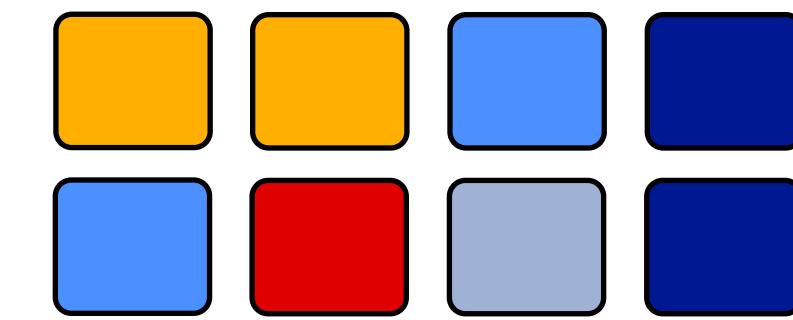
New compiled shader:

Processes eight fragments using  
vector ops on vector registers

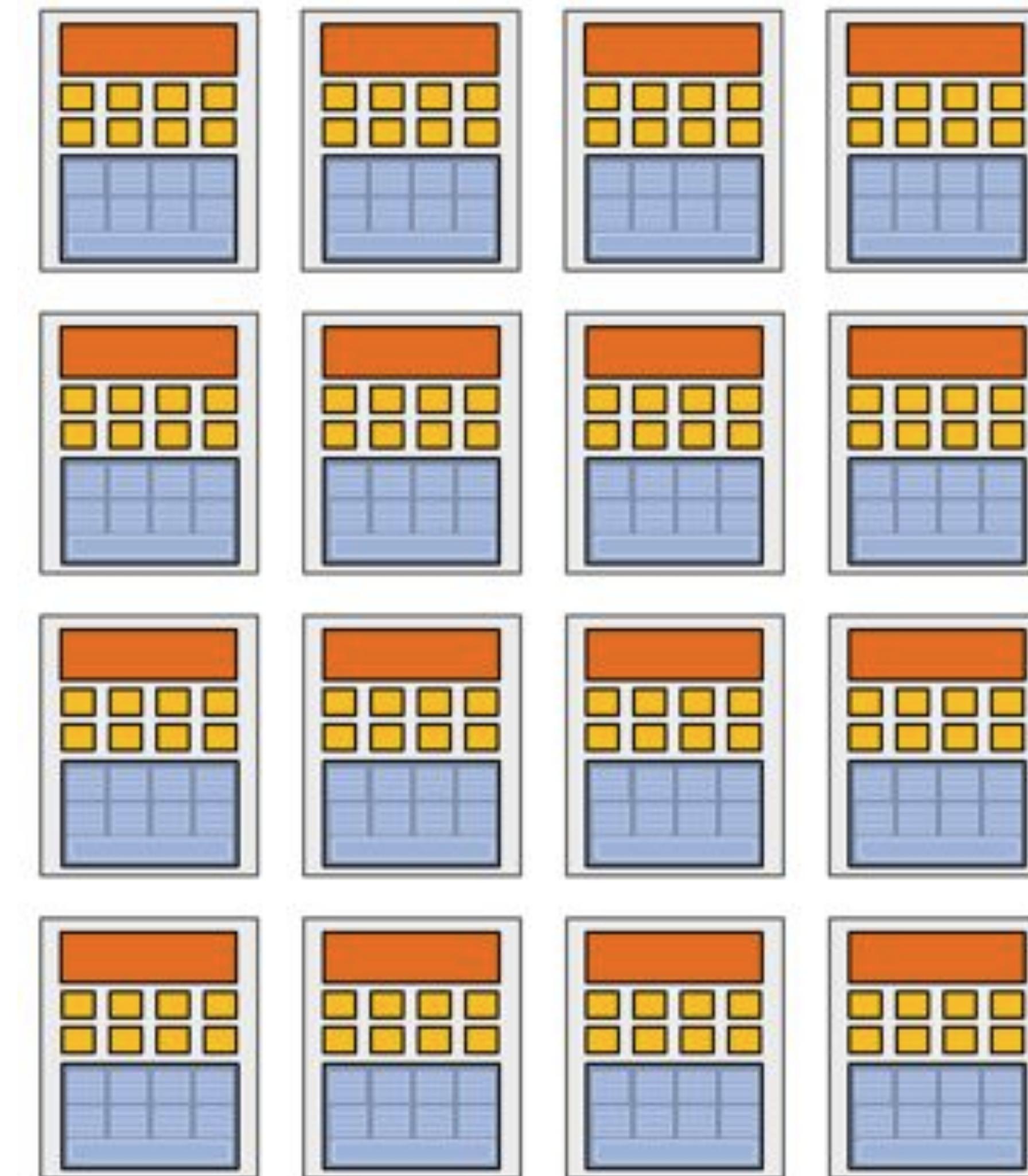
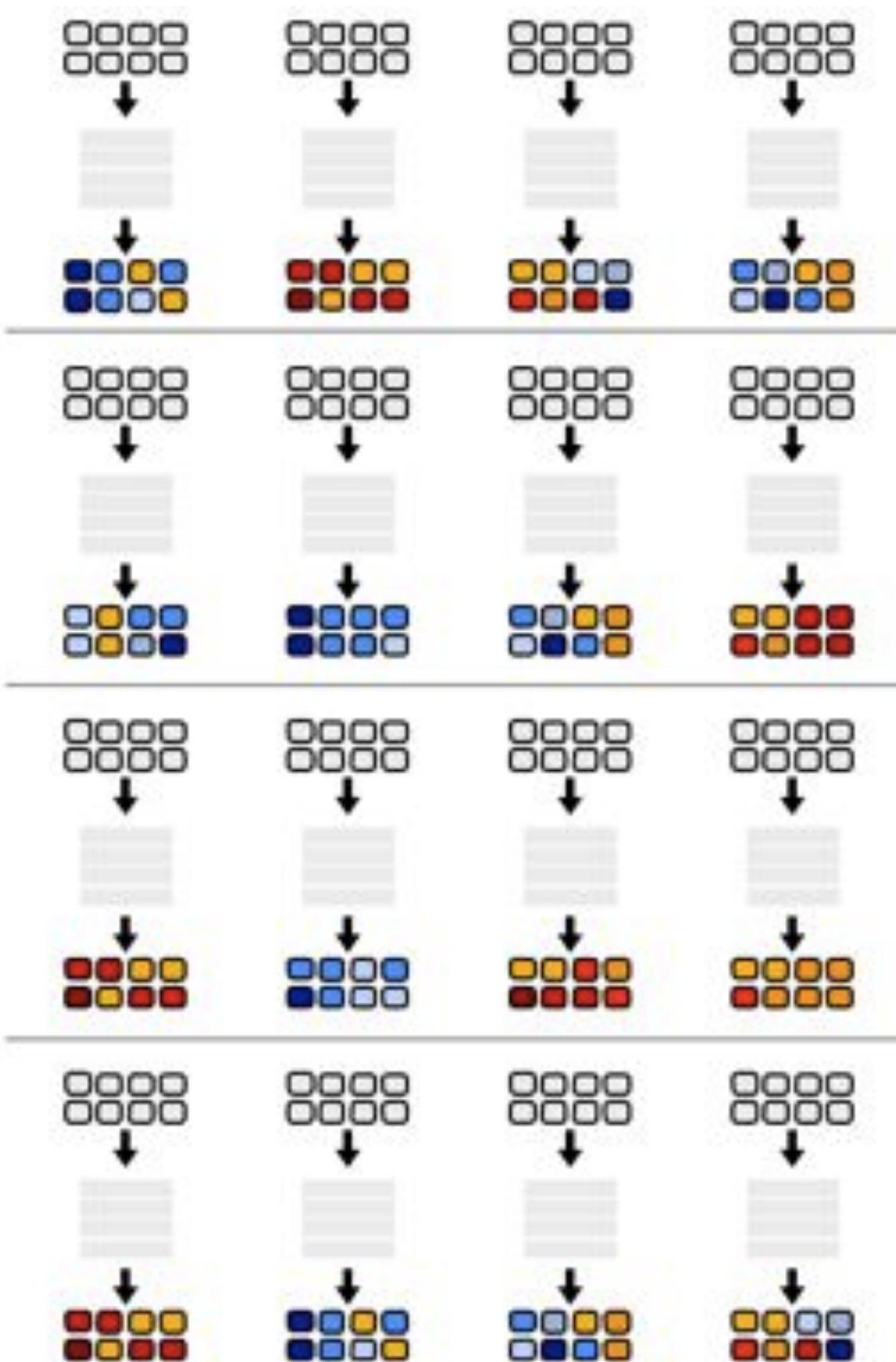
# Modifying the shader



```
<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)
```



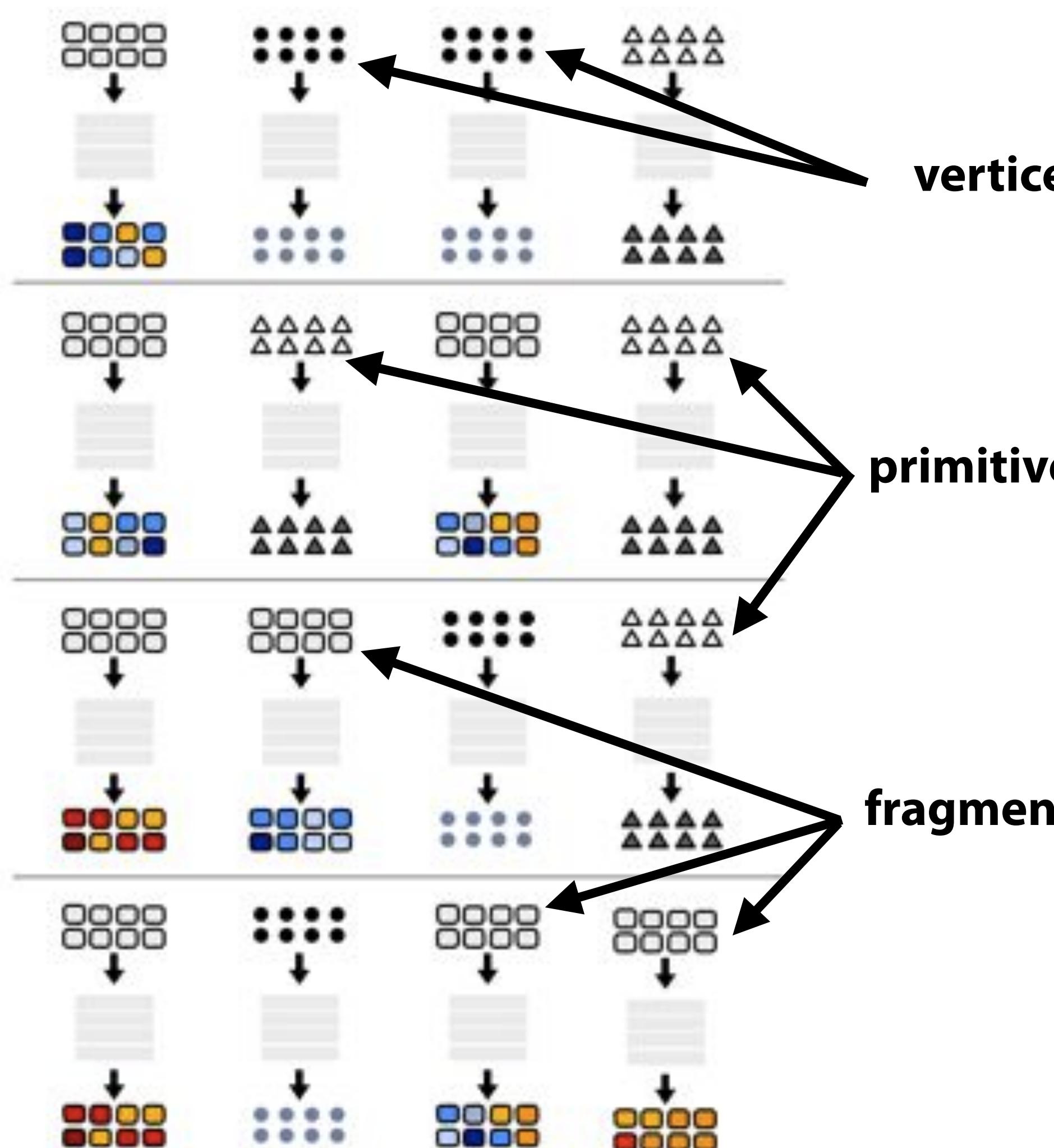
# 128 fragments in parallel



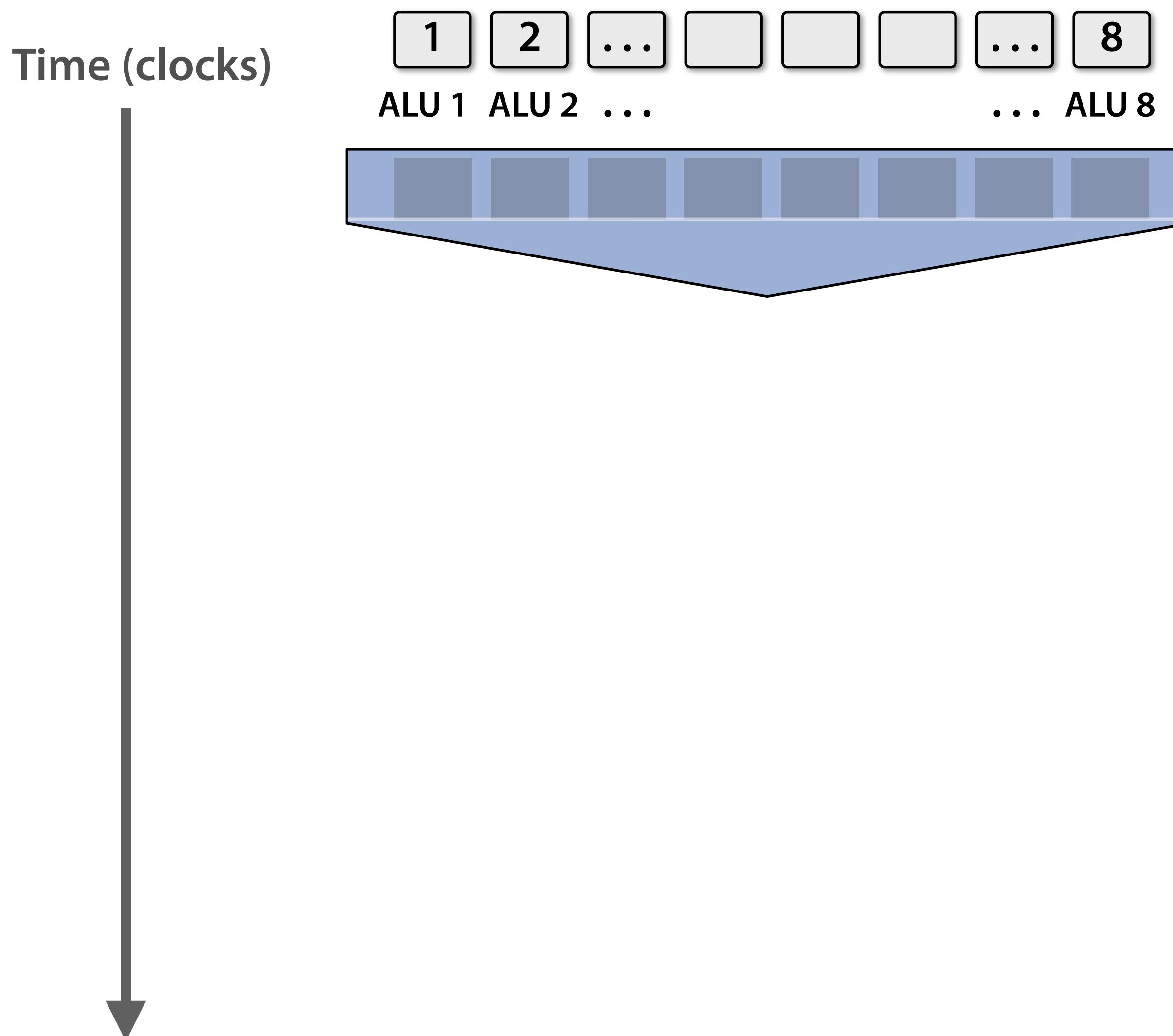
16 cores = 128 ALUs, 16 simultaneous instruction streams

# 128 [ ] in parallel

vertices/fragments  
primitives  
OpenCL work items  
CUDA threads



# But what about branches?

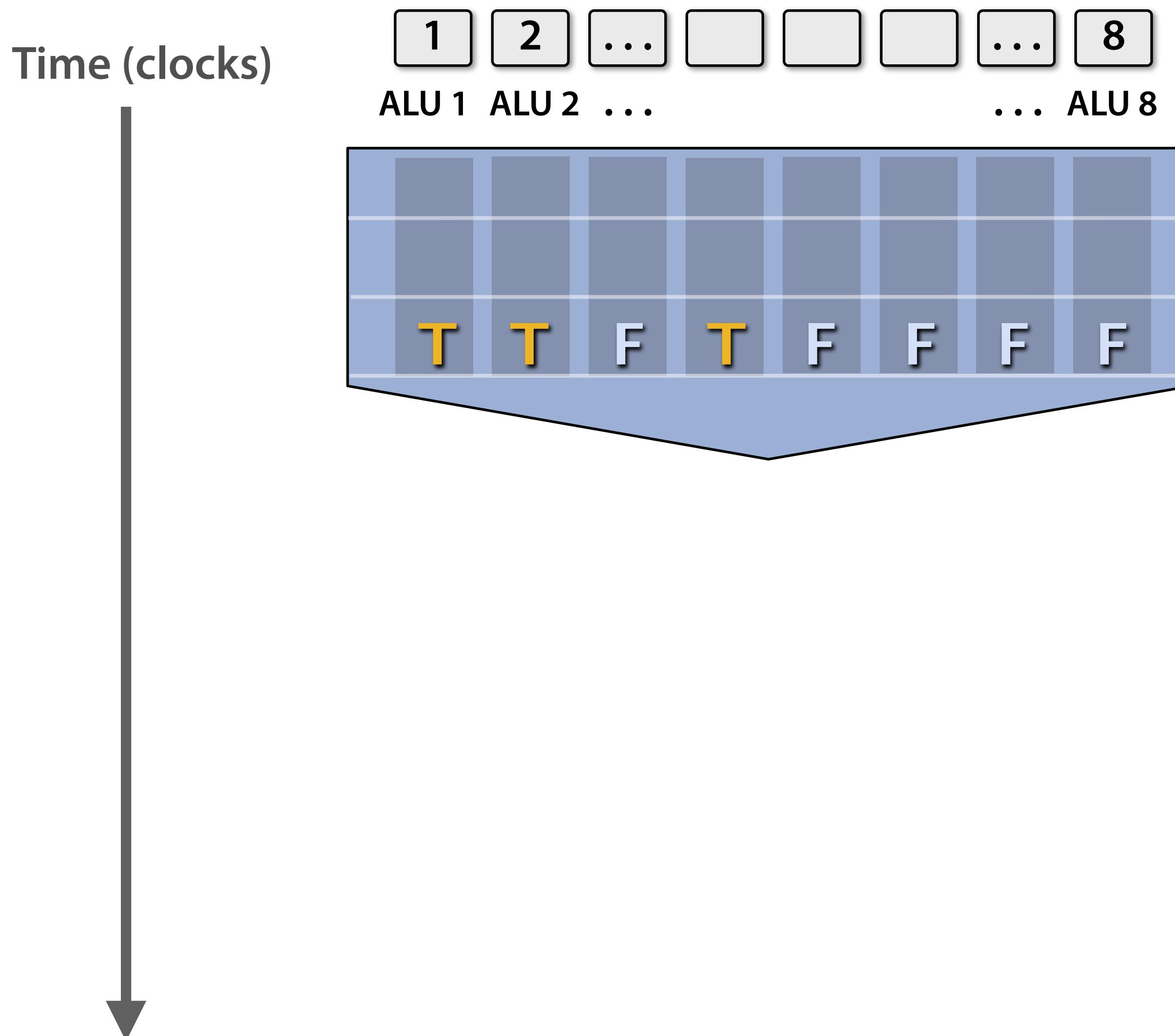


<unconditional  
shader code>

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

<resume unconditional  
shader code>

# But what about branches?

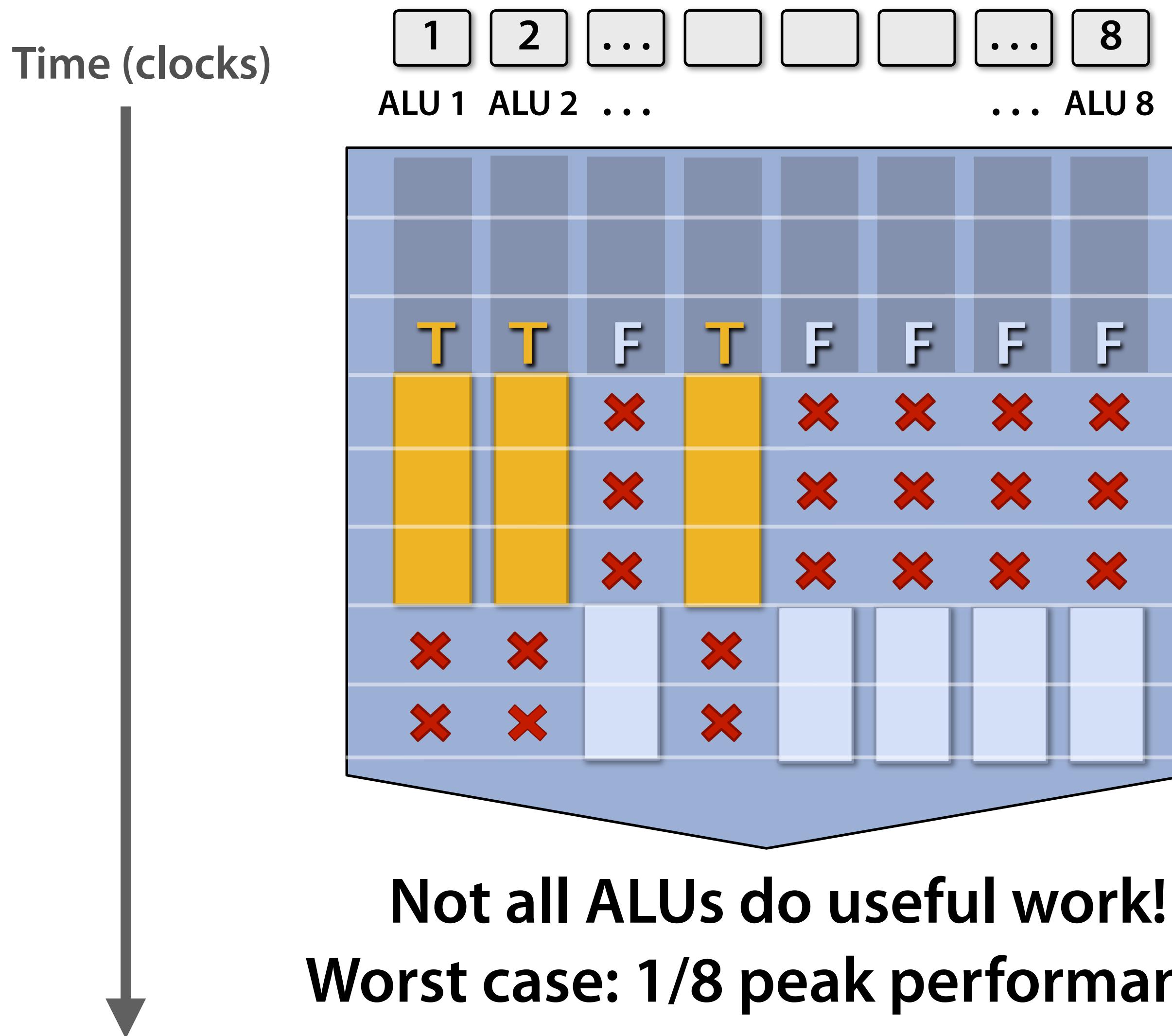


<unconditional  
shader code>

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

<resume unconditional  
shader code>

# But what about branches?

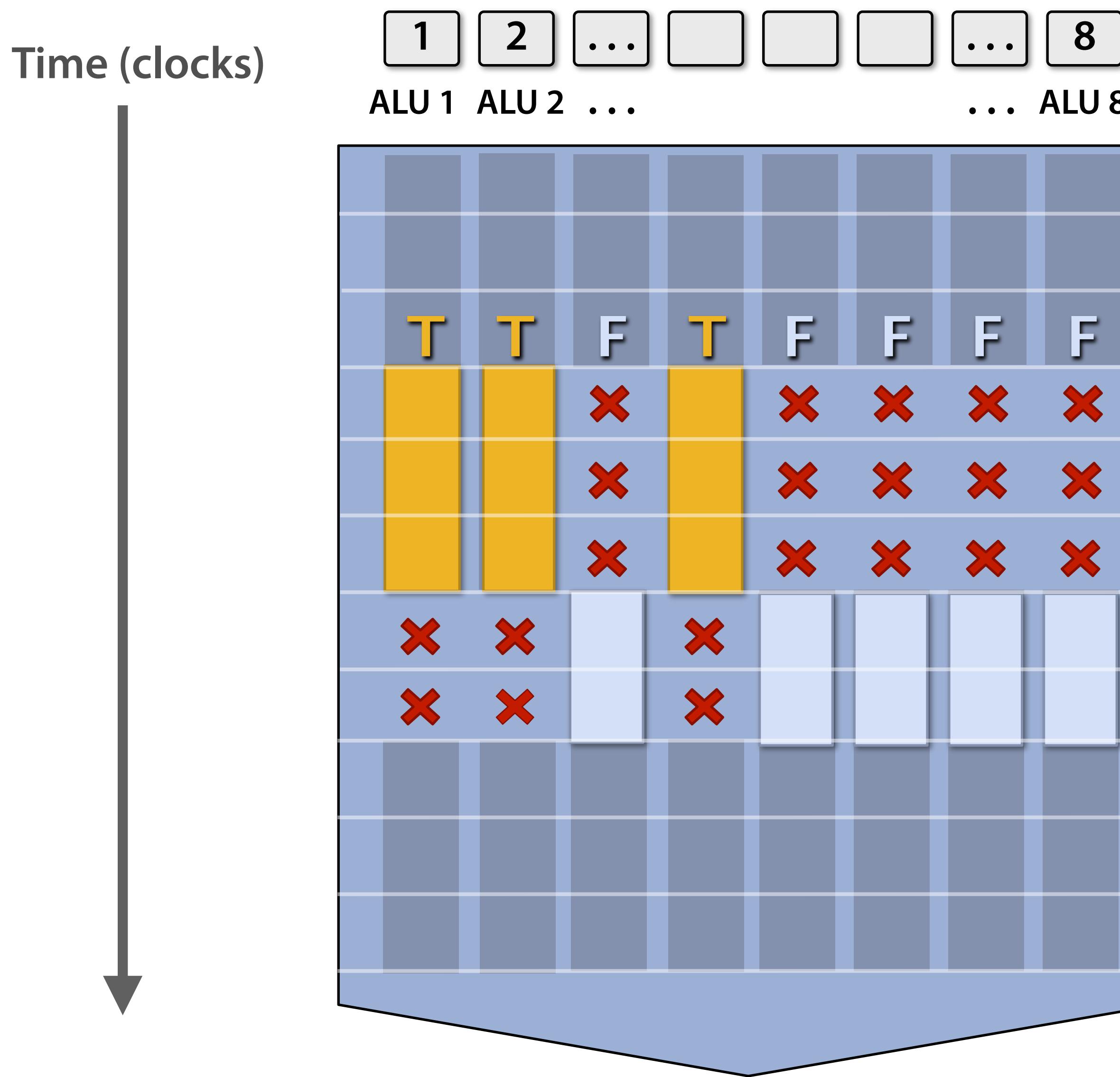


<unconditional  
shader code>

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

<resume unconditional  
shader code>

# But what about branches?



<unconditional  
shader code>

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

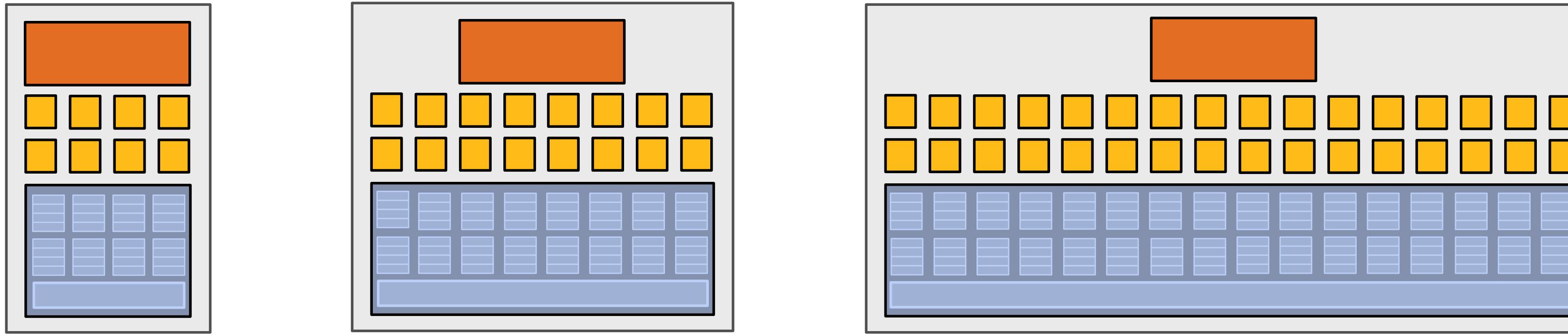
<resume unconditional  
shader code>

# Terminology

- “Coherent” execution\*\*\* (admittedly fuzzy definition): when processing of different entities is similar, and thus can share resources for efficient execution
  - Instruction stream coherence: different fragments follow same sequence of logic
  - Memory access coherence:
    - Different fragments access similar data (avoid memory transactions by reusing data in cache)
    - Different fragments simultaneously access contiguous data (enables efficient, bulk granularity memory transactions)
- “Divergence”: lack of coherence
  - Usually used in reference to instruction streams (divergent execution does not make full use of SIMD processing)

\*\*\* Do not confuse this use of term “coherence” with cache coherence protocols

# GPUs share instruction streams across many fragments



**In modern GPUs: 16 to 64 fragments share an instruction stream.**

# **Stalls!**

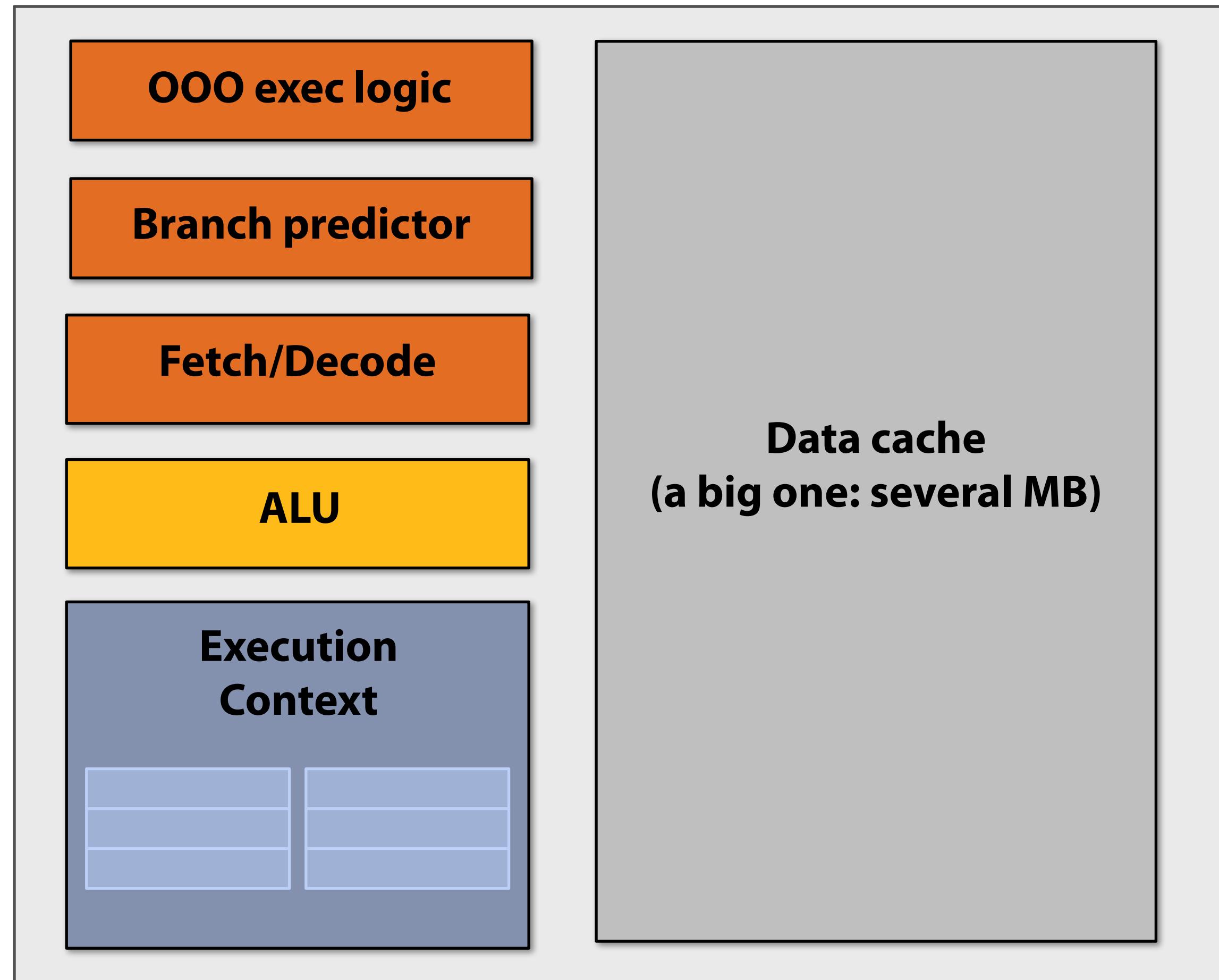
**Stalls occur when a core cannot run the next instruction because of a dependency on a previous operation.**

# Recall: diffuse reflectance shader

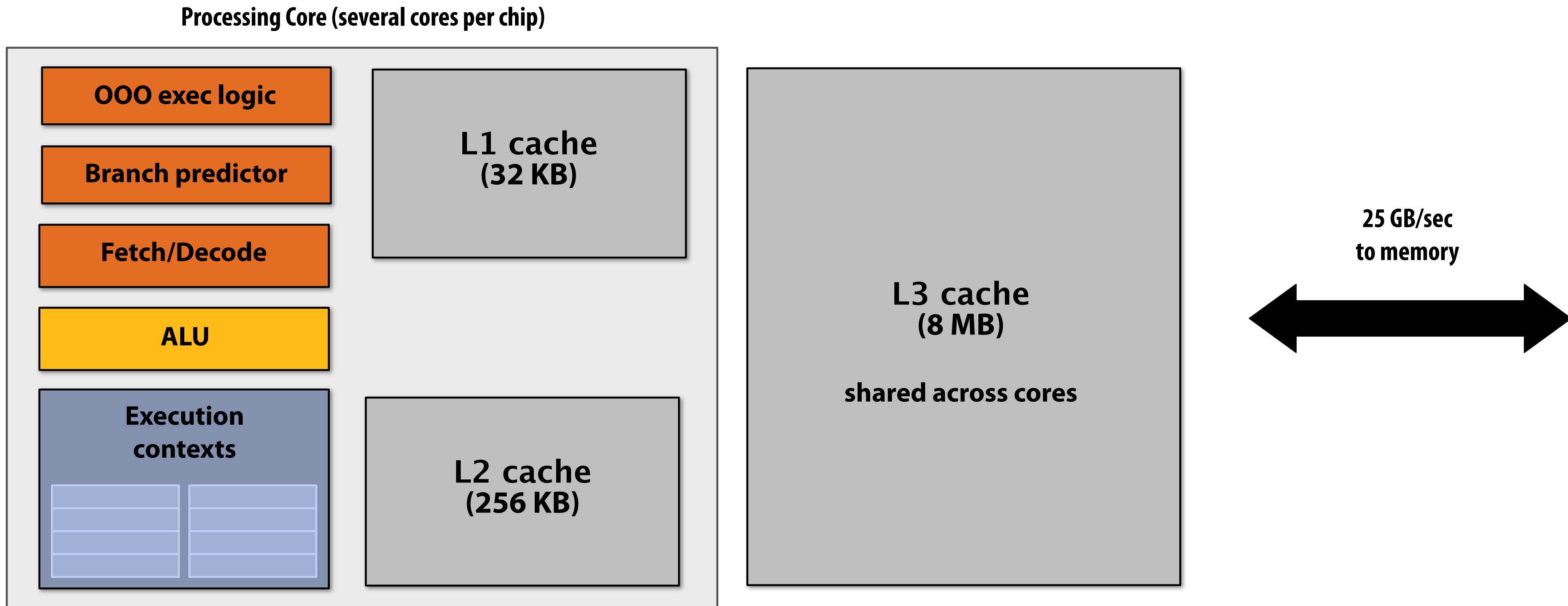
```
sampler mySamp;  
Texture2D<float3> myTex;  
float3 lightDir;  
  
float4 diffuseShader(float3 norm, float2 uv)  
{  
    float3 kd;  
    kd = myTex.Sample(mySamp, uv);  
    kd *= clamp( dot(lightDir, norm), 0.0, 1.0);  
    return float4(kd, 1.0);  
}
```

Texture access:  
Latency of 100's of cycles

# Recall: CPU-style core



# CPU-style memory hierarchy



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

# **Stalls!**

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

**Texture access latency = 100's to 1000's of cycles**

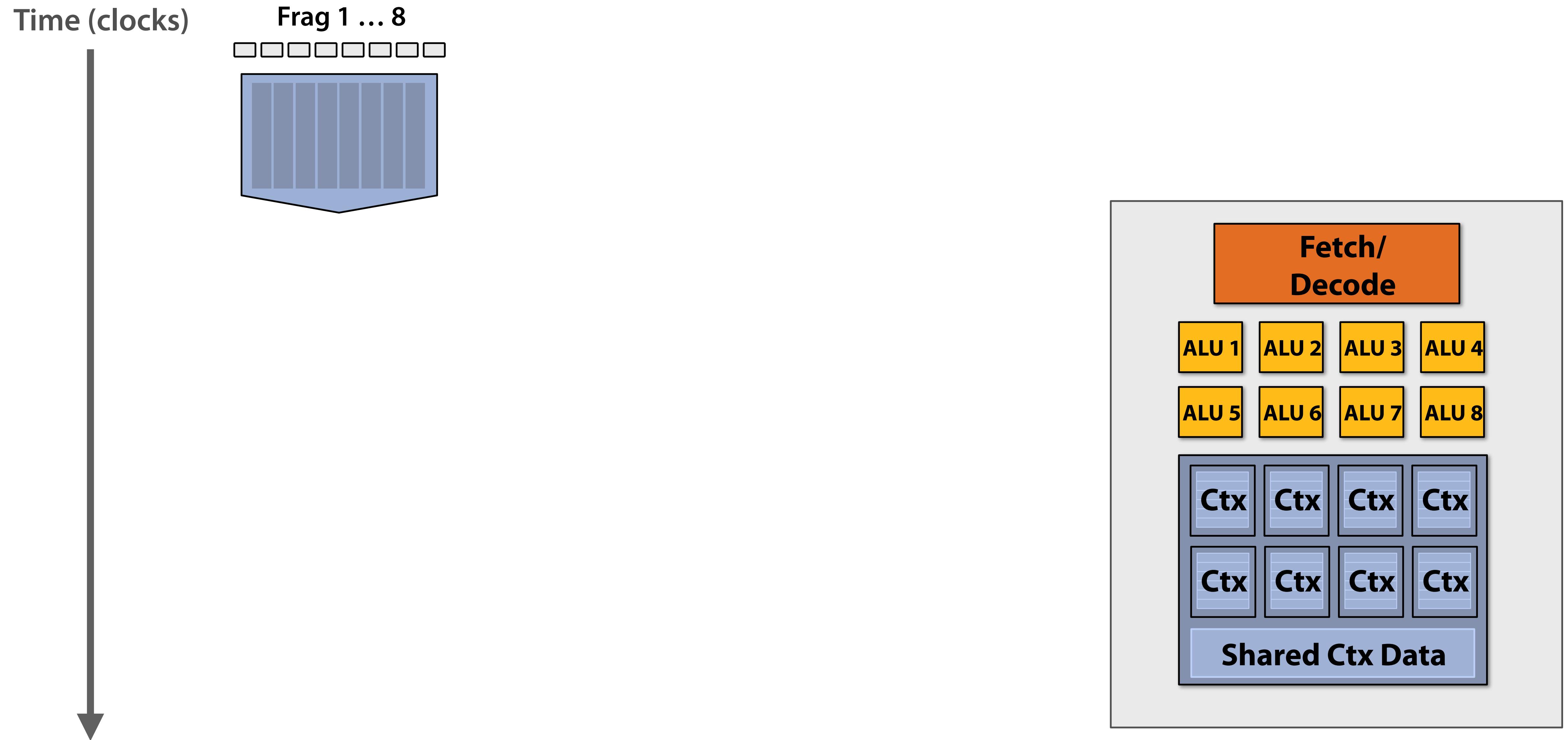
**We've removed the fancy caches and logic that helps avoid stalls.**

**But we have LOTS of independent fragments.**  
**(Way more fragments to process than ALUs)**

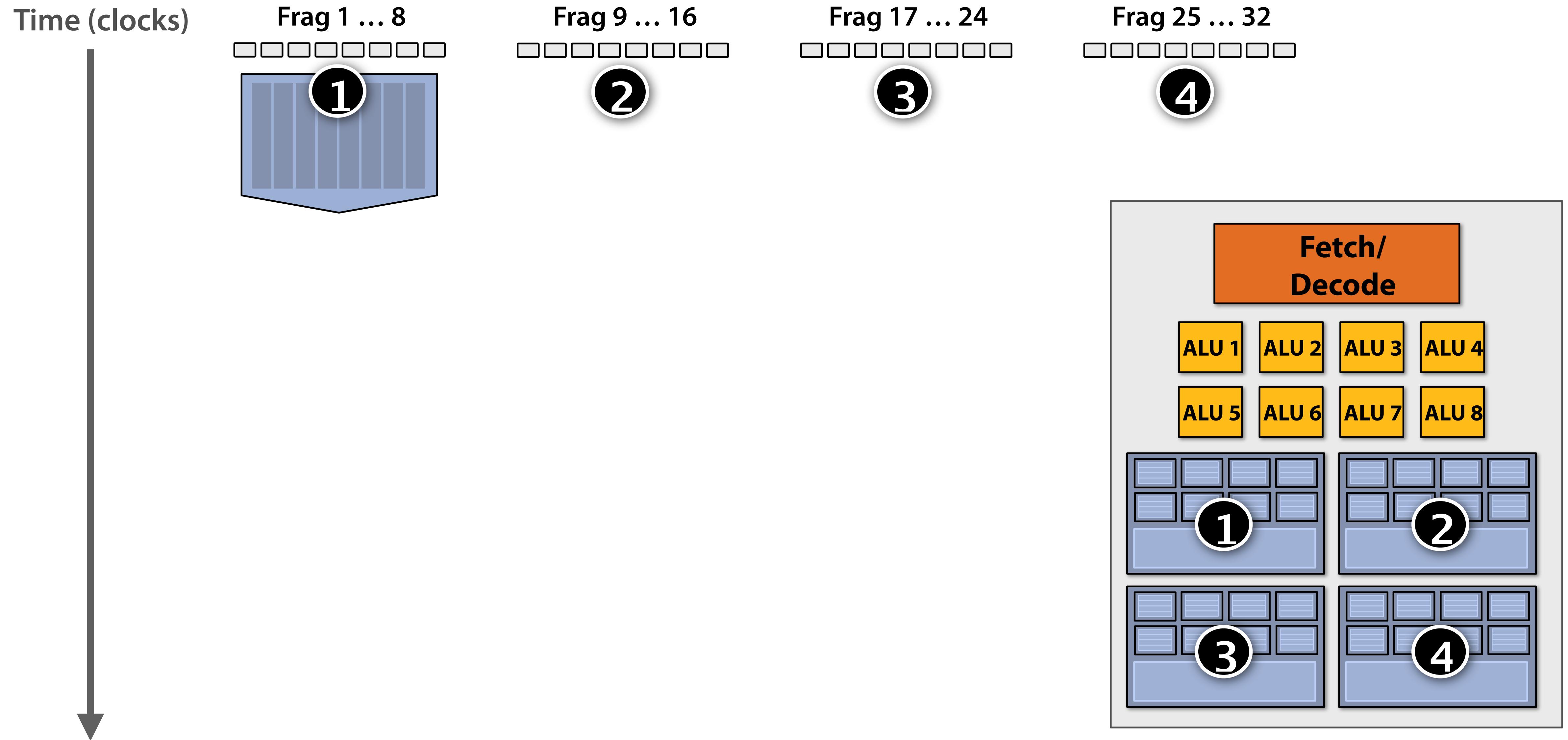
### **Idea #3:**

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

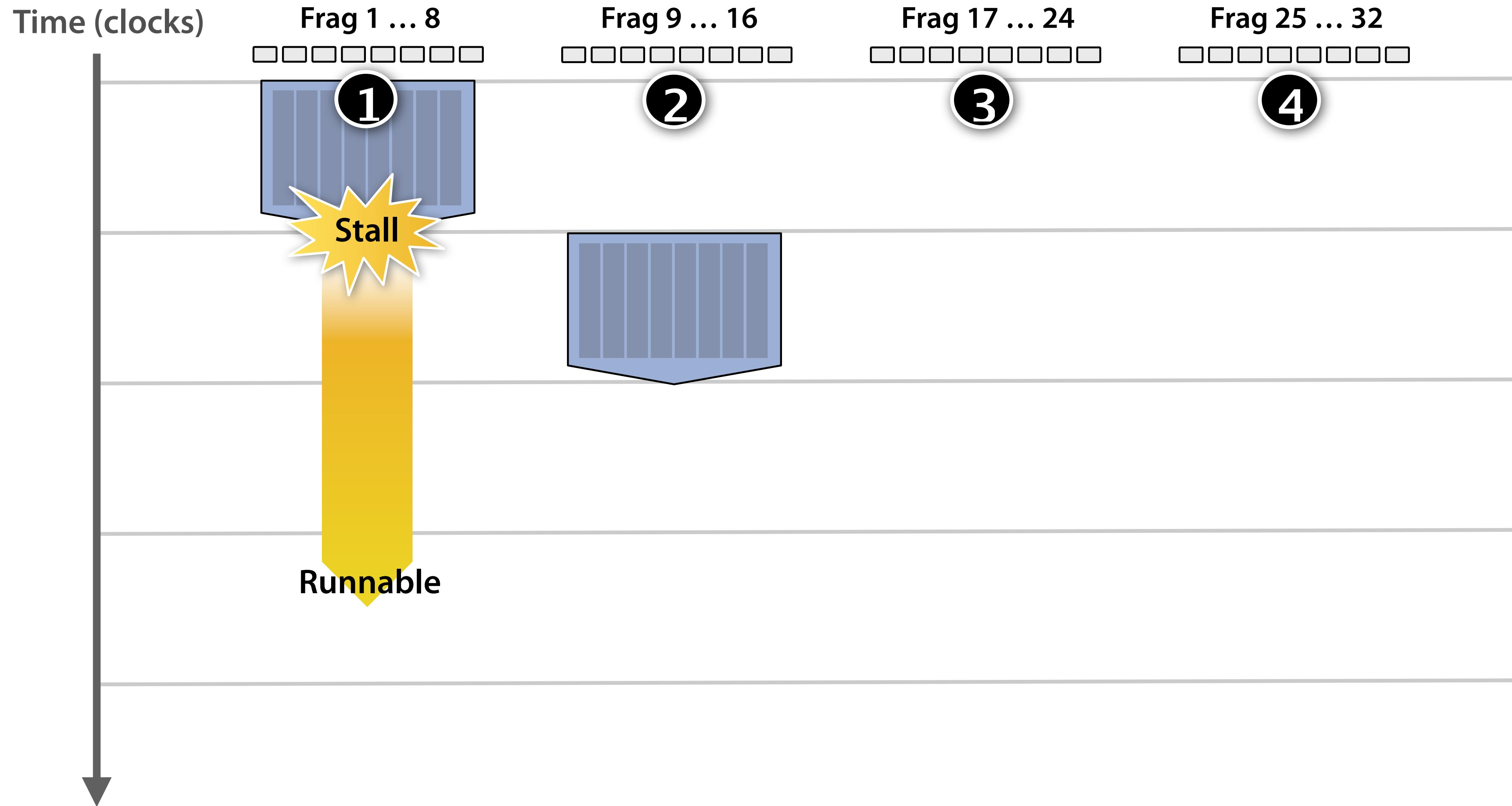
# Hiding shader stalls



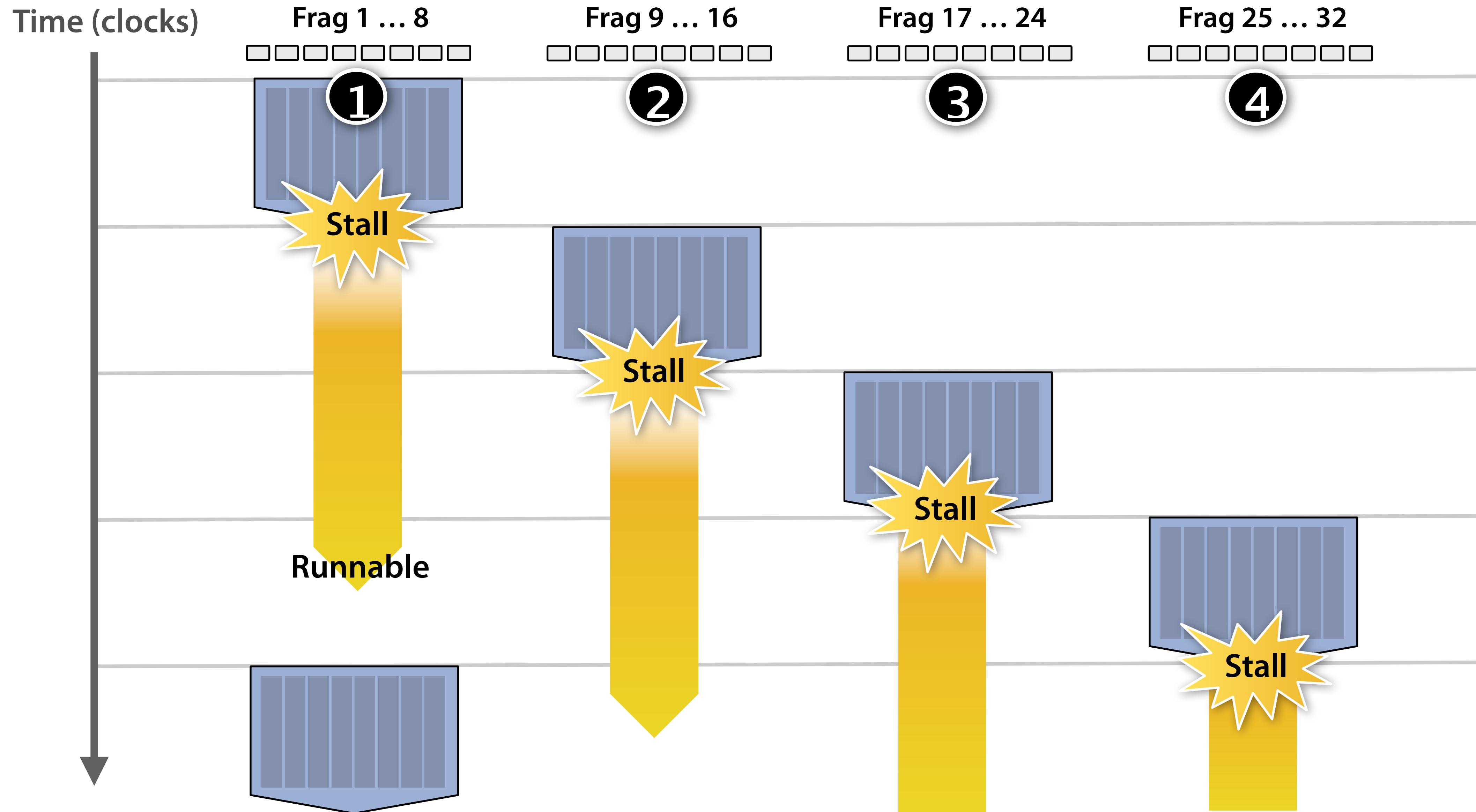
# Hiding shader stalls



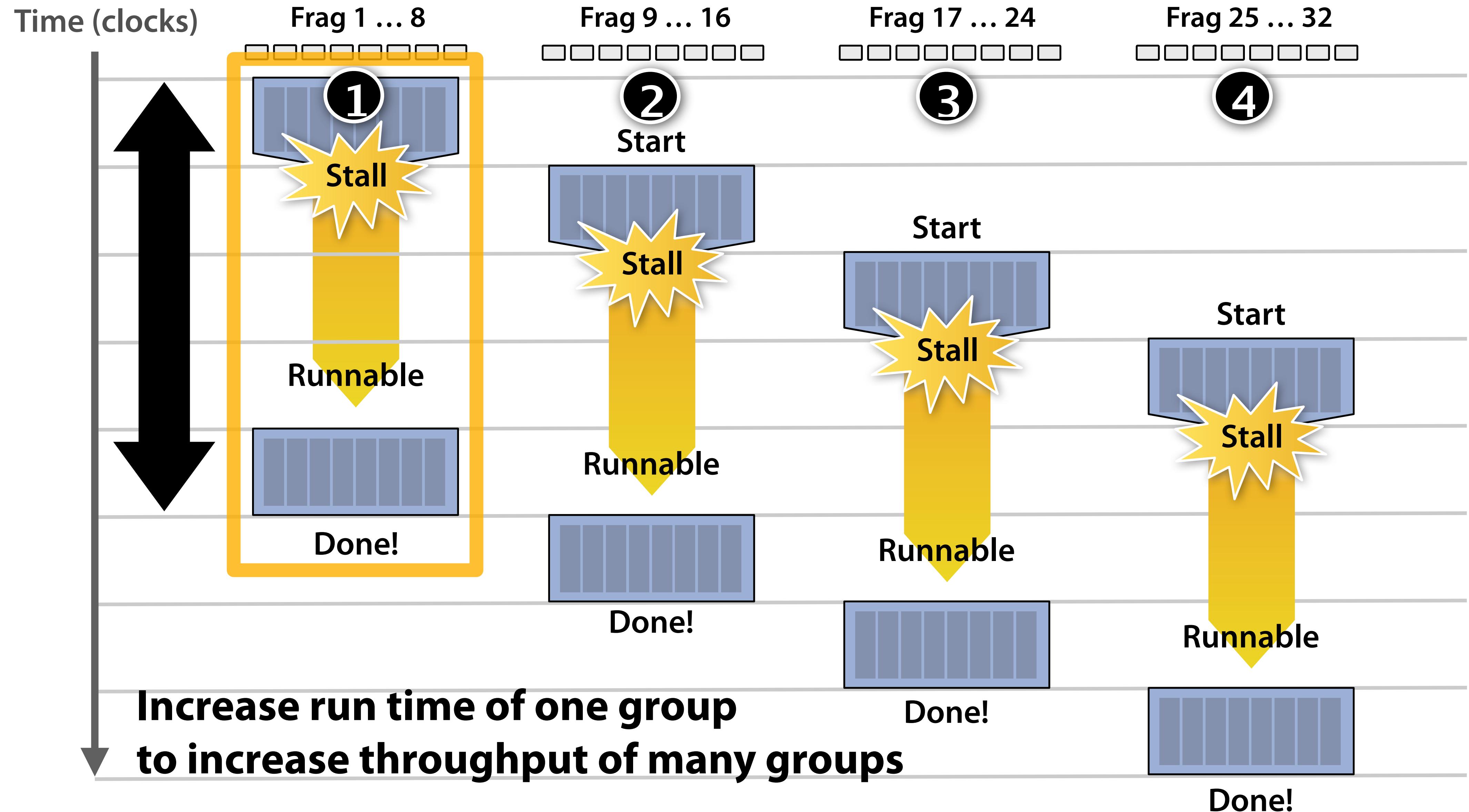
# Hiding shader stalls



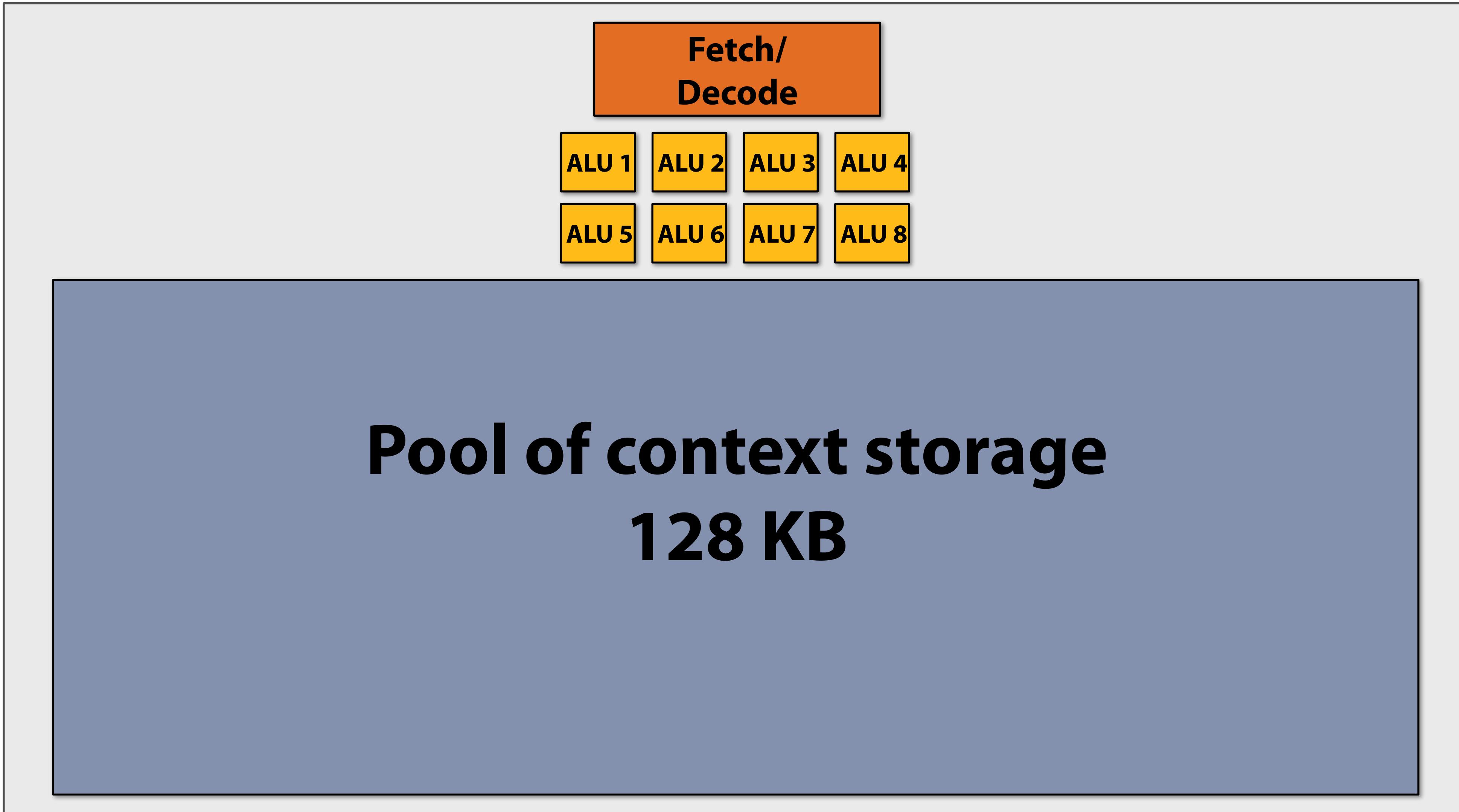
# Hiding shader stalls



# Throughput!

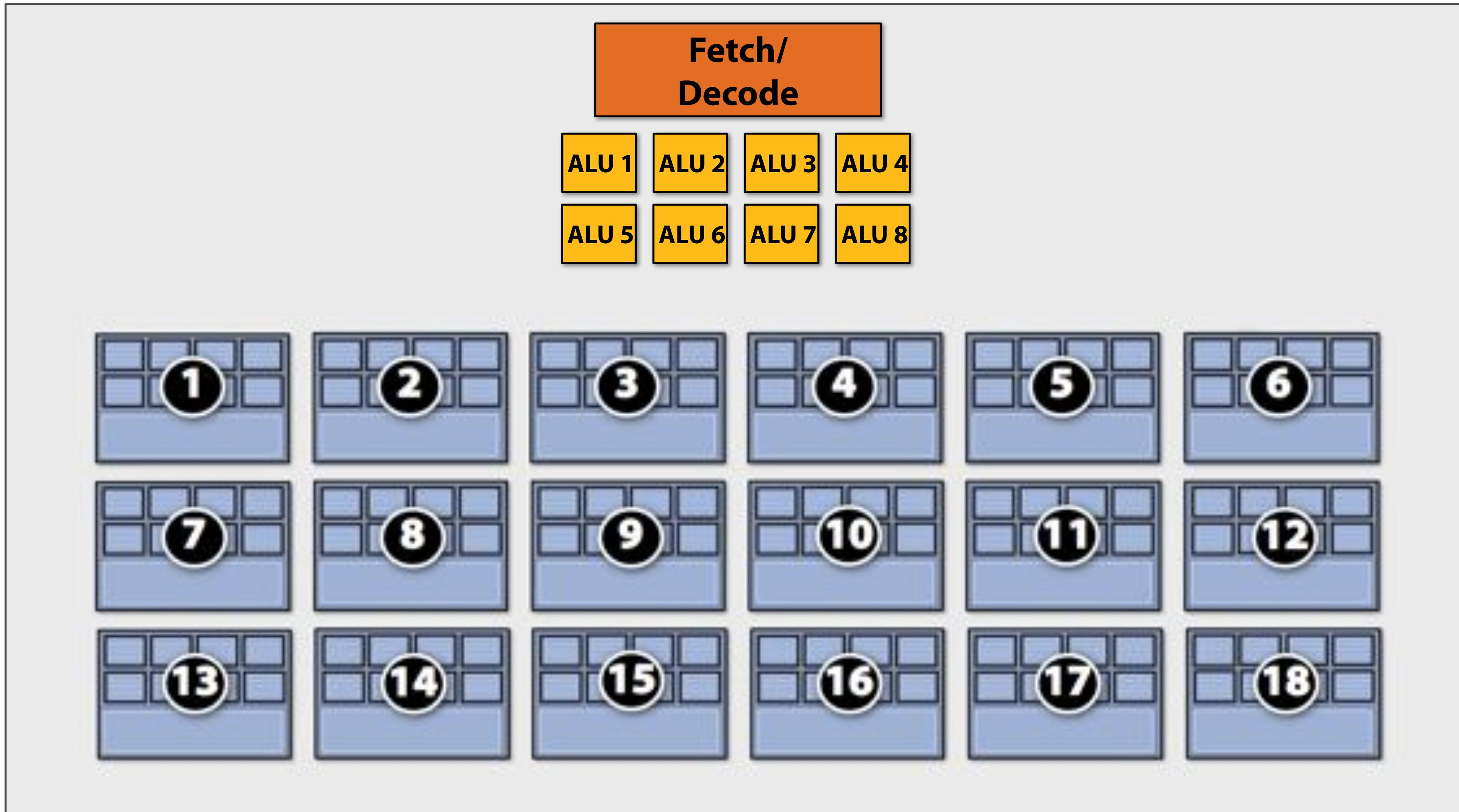


# Storing contexts

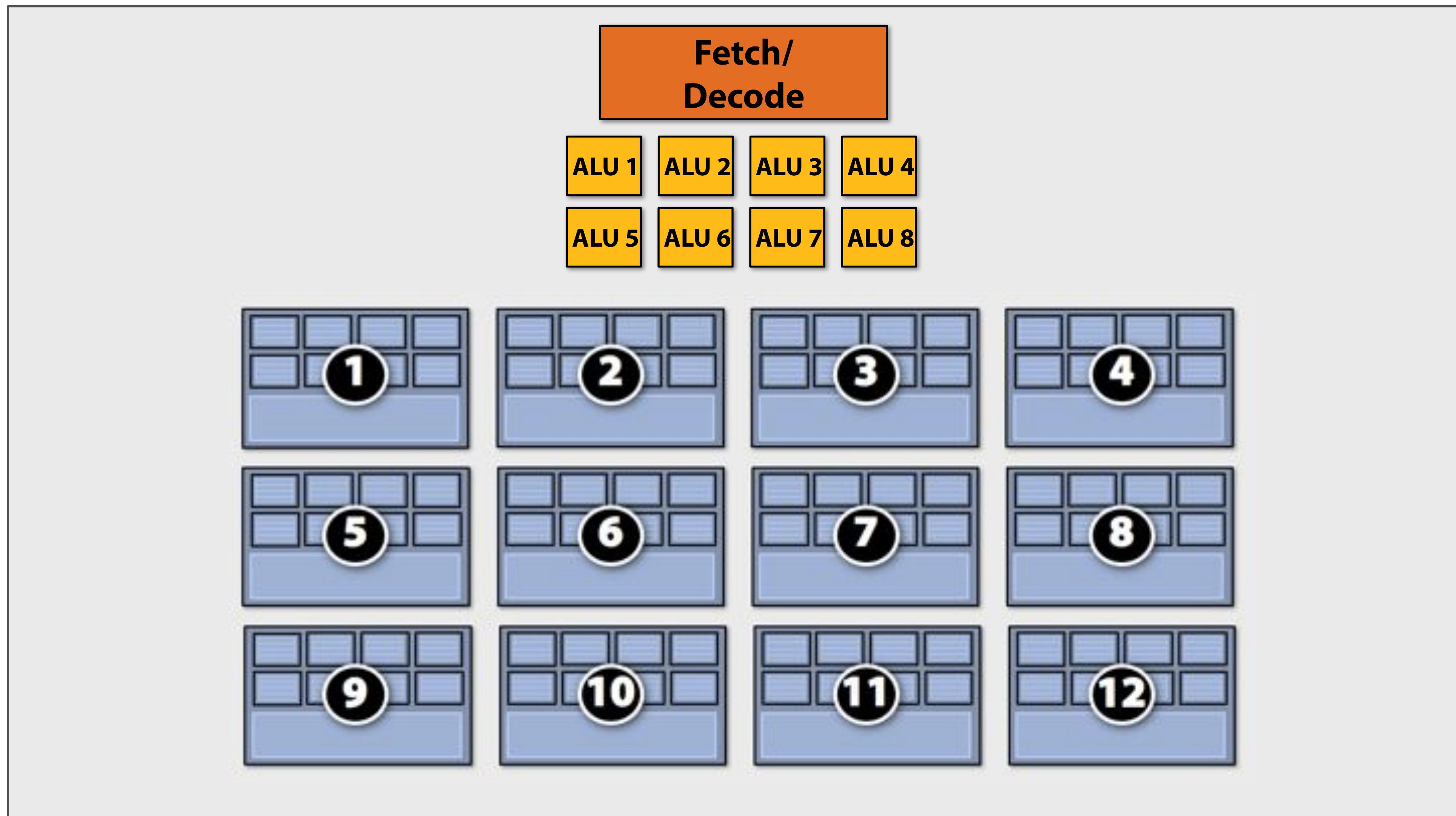


# Eighteen small contexts

(maximal latency hiding)

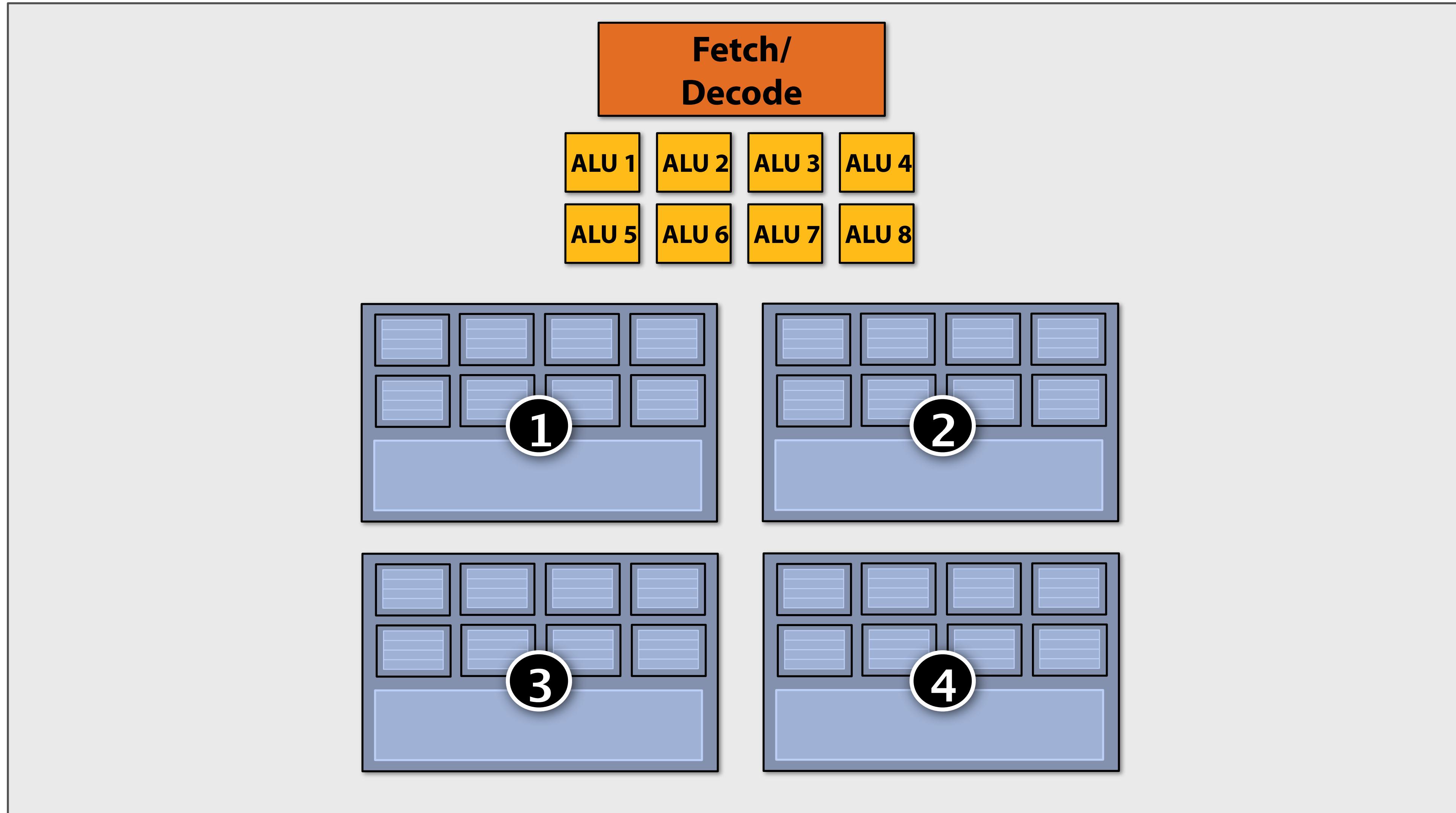


# Twelve medium contexts



# Four large contexts

(low latency hiding ability)



# My chip!

16 cores

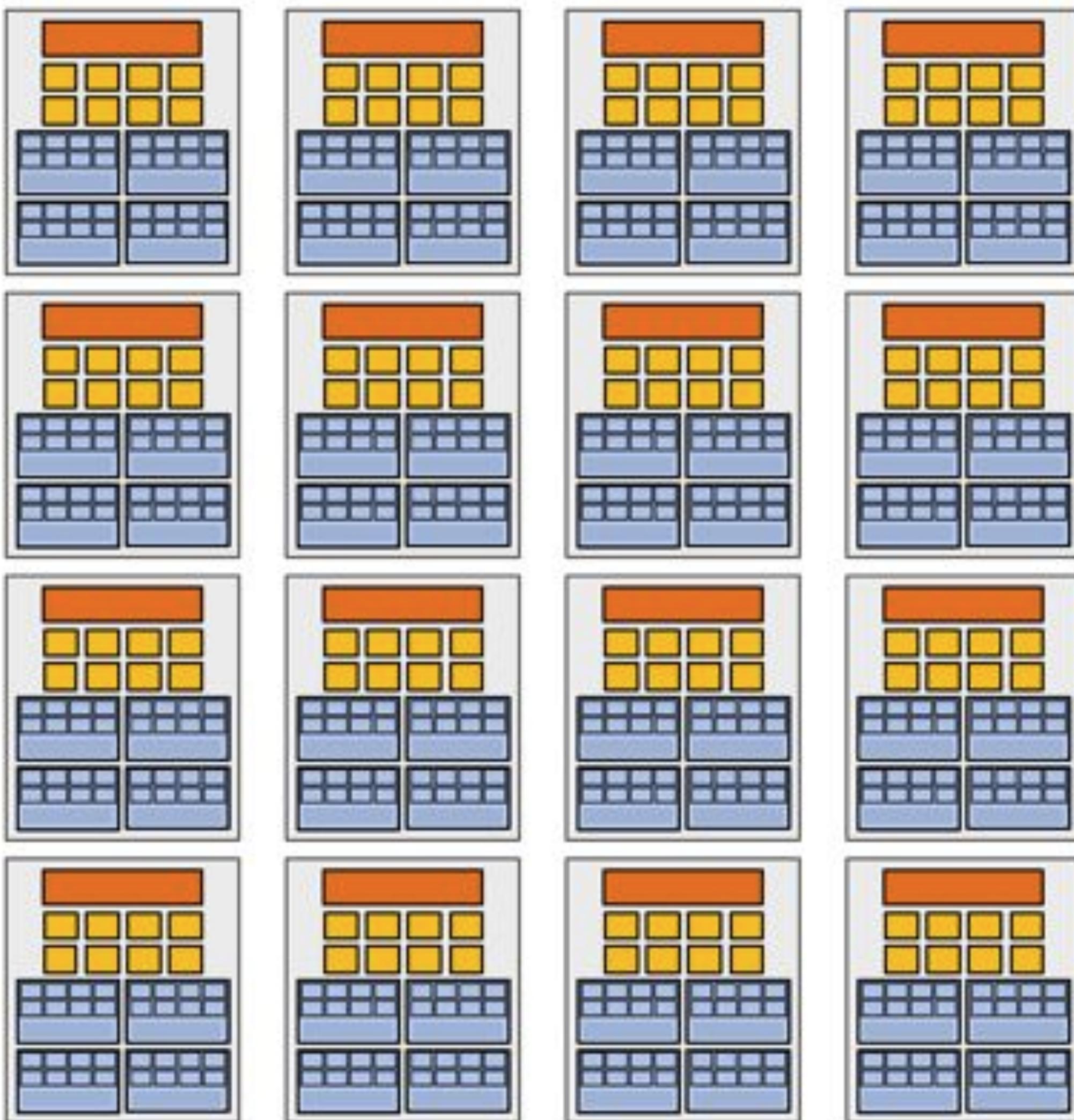
8 mul-add ALUs per core  
(128 total)

16 simultaneous  
instruction streams

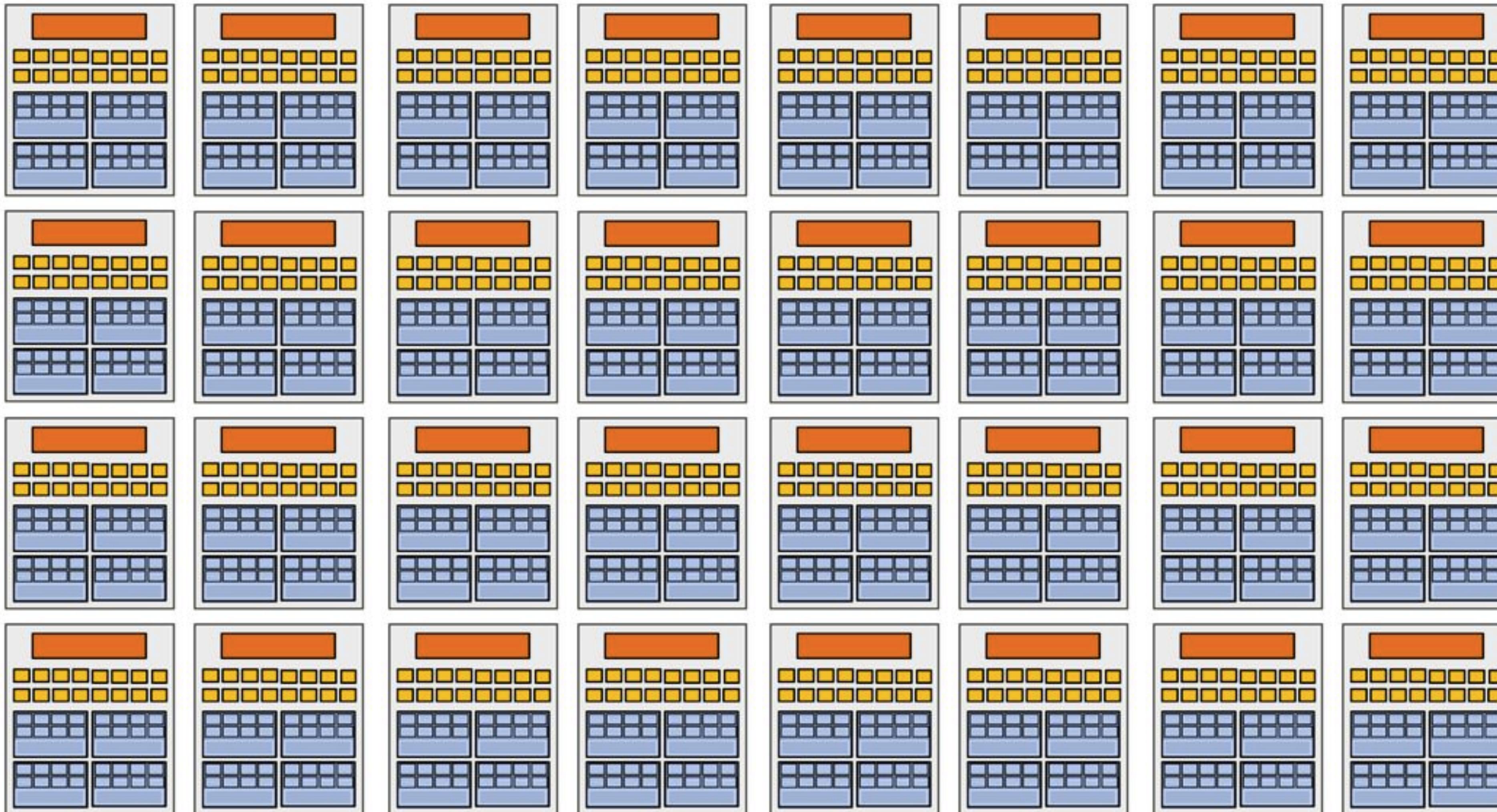
64 concurrent (but interleaved)  
instruction streams

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 a real GPU**

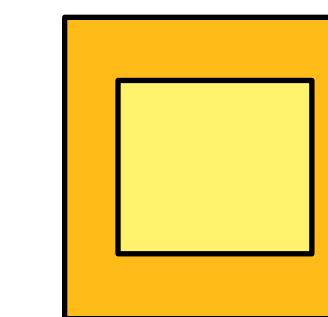
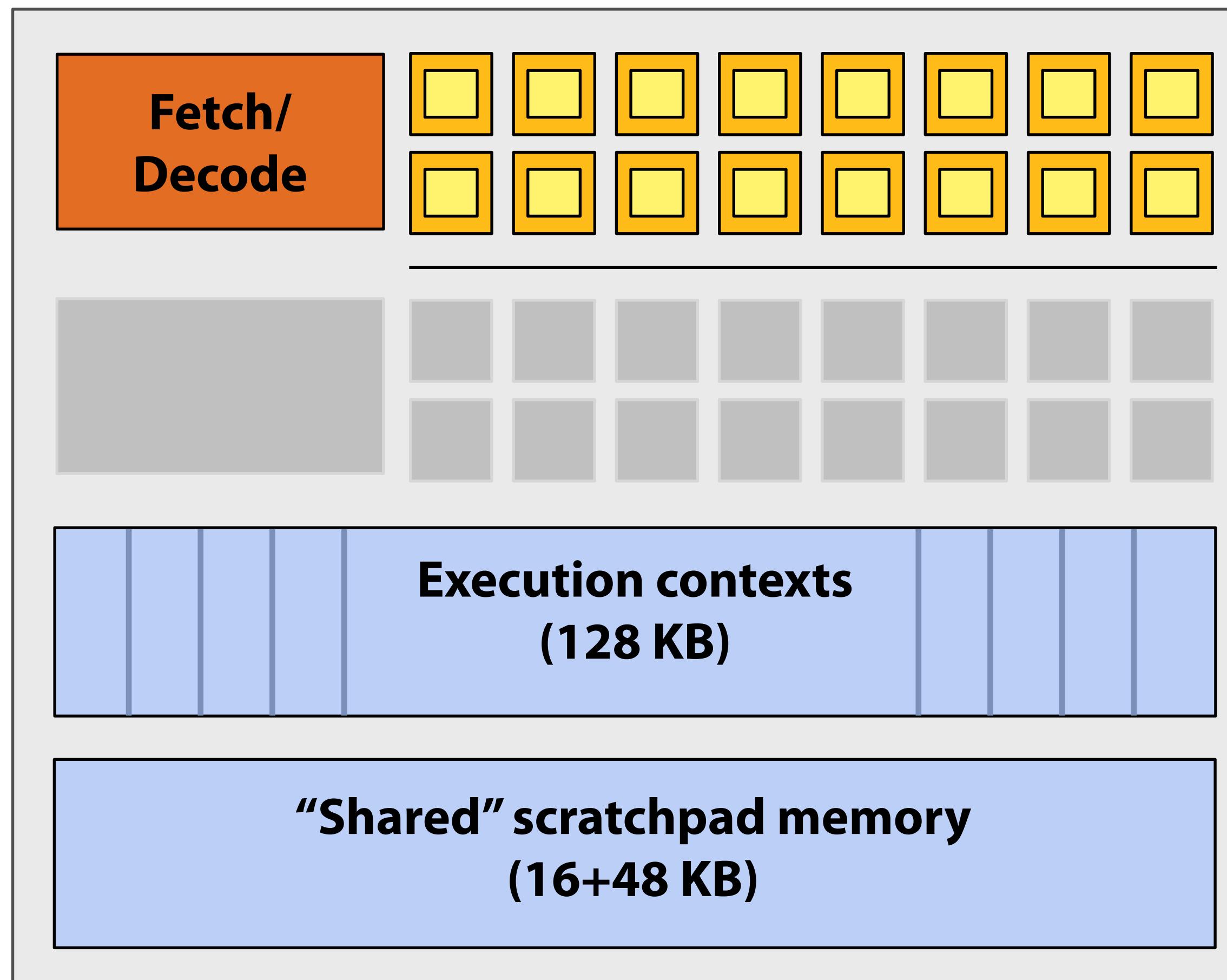
**NVIDIA GeForce GTX 480**

# NVIDIA GeForce GTX 480 (Fermi)

- NVIDIA-speak:
  - **480 stream processors (“CUDA cores”)**
  - **“SIMT execution”**
  
- Generic speak:
  - **15 cores**
  - **2 groups of 16 SIMD functional units per core**



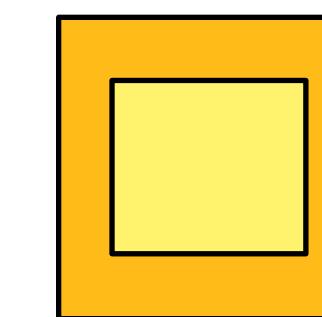
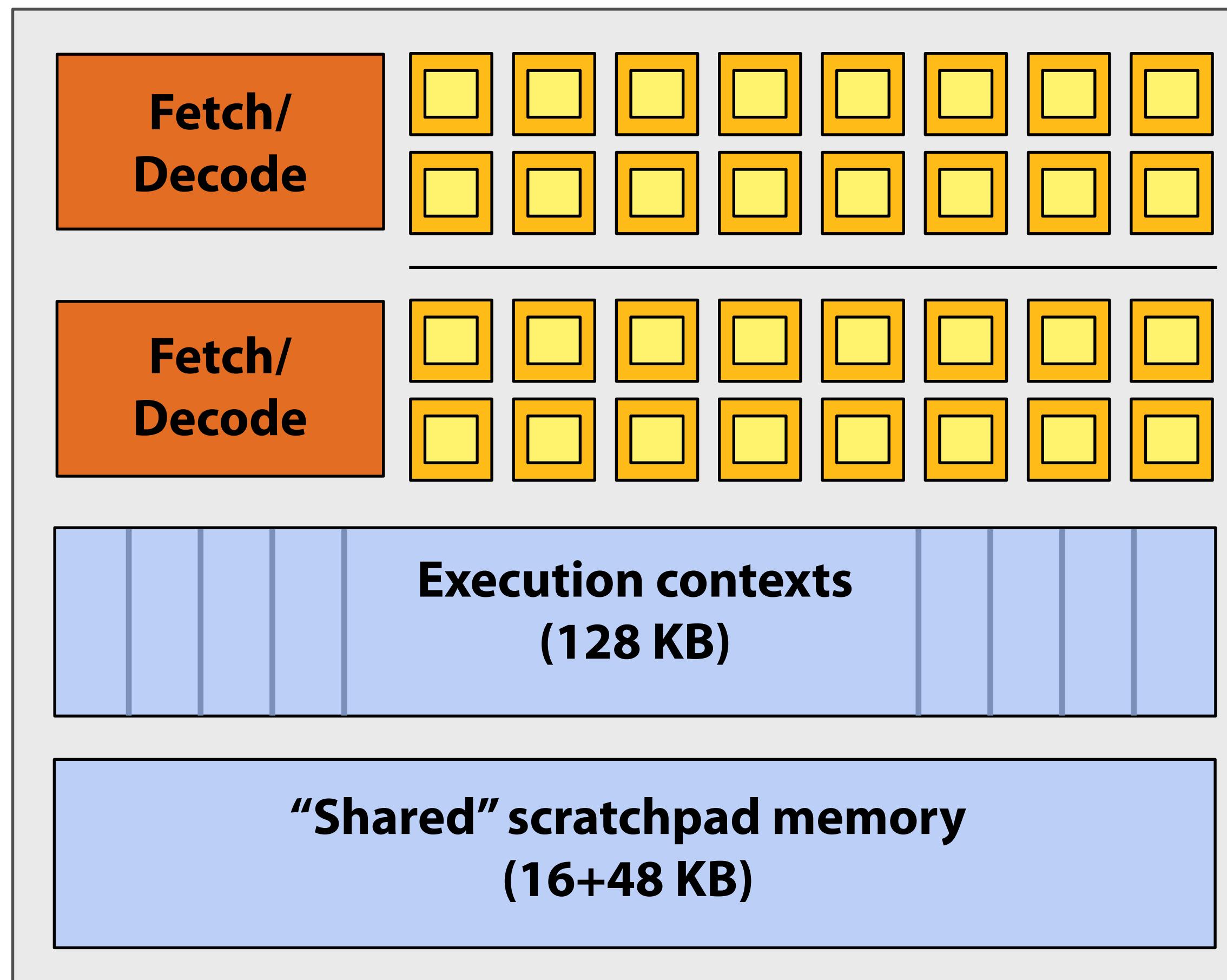
# NVIDIA GeForce GTX 480 “core”



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

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

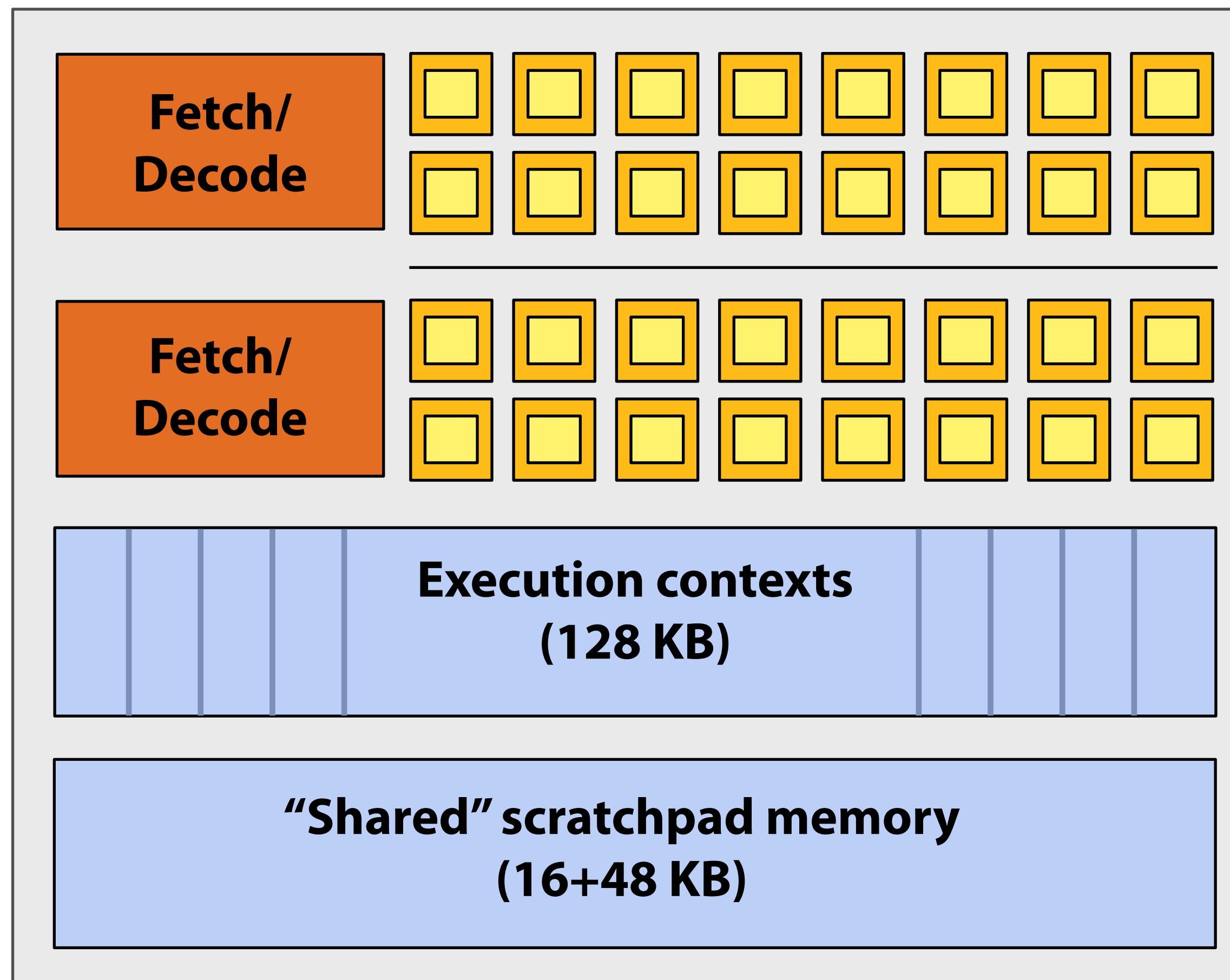
# NVIDIA GeForce GTX 480 “core”

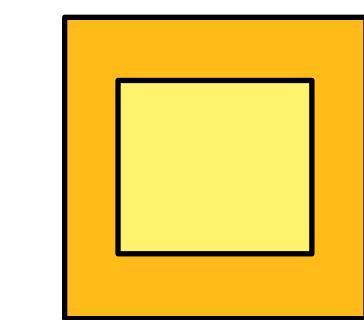


= SIMD function unit,  
control shared across 16 units  
(1 MUL-ADD per clock)

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

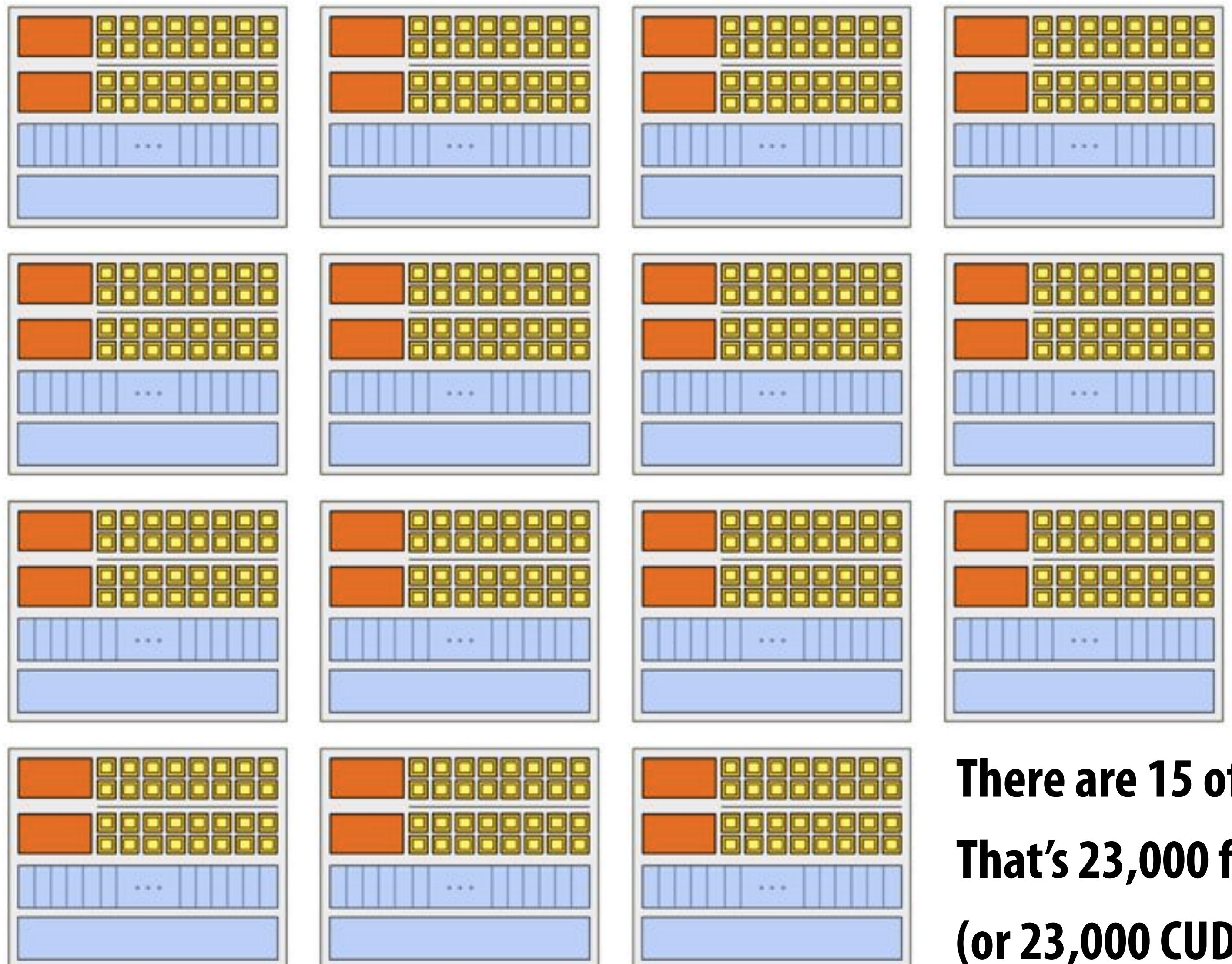
# NVIDIA GeForce GTX 480 "SM"



 = **CUDA core**  
(1 MUL-ADD per clock)

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

# NVIDIA GeForce GTX 480



**There are 15 of these things on the GTX 480:  
That's 23,000 fragments!  
(or 23,000 CUDA threads!)**

# **Looking Forward**

# Recent trends

- **Support for alternative programming interfaces**
  - Accelerate non-graphics applications using GPU (CUDA, OpenCL)
- **How does graphics pipeline abstraction change to enable more advanced real-time graphics?**
  - Direct3D 11 adds three new pipeline stages

# Global illumination algorithms



Credit: NVIDIA

**Ray tracing:**  
**for accurate reflections, shadows**



Credit: Bratincevic

# Alternative shading structures (e.g., deferred shading)



For more efficient scaling to many lights (1000 lights, [Andersson 09])

# Simulation



# Cinematic scene complexity



Image credit: Pixar (Toy Story 3, 2010)



# Motion blur



# **Thanks!**

**Relevant CMU Courses for students interested in high performance graphics:**

**15-418: Parallel Architecture and Programming (spring 2013 semester)**

**15-869: Graphics and Imaging Architectures (offered next fall)**