Software Performance Engineering

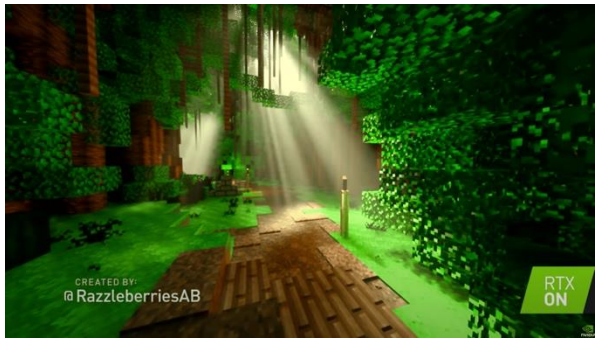
LECTURE 11
GPU ARCHITECTURE

Xuhao Chen

October 30, 2025

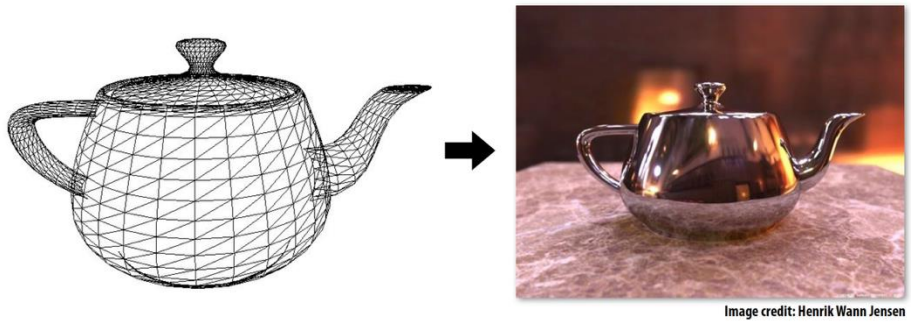
What is a GPU?

- Graphics Processing Units



ray tracing

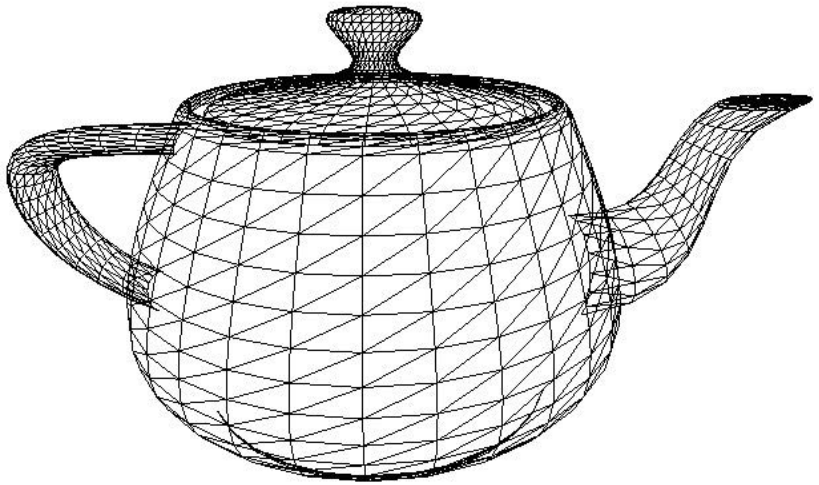
gaming



3D rendering

What GPUs were originally designed to do

- Simple definition of rendering task: computing how each triangle in 3D mesh contributes to appearance of each pixel in the image?



Input: description of a scene:
3D surface geometry (e.g., triangle mesh)
surface materials, lights, camera, etc.

Output: image of the scene

3D rendering

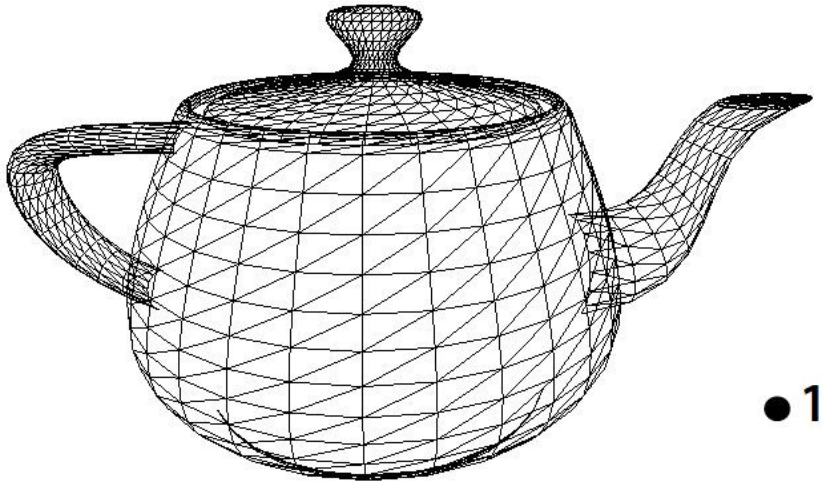
Render Highly Complex 3D Scenes in Realtime

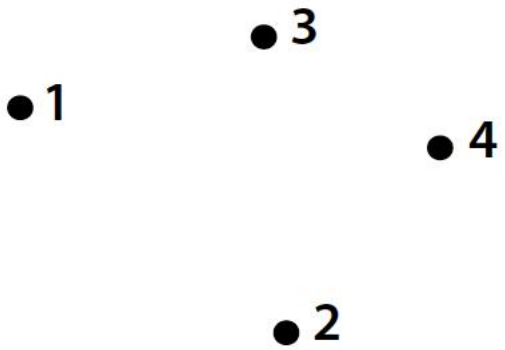
- 30-60 frames per second (even higher for VR)
- High resolution (2-4 megapixel)

Render Highly Complex 3D Scenes in Realtime

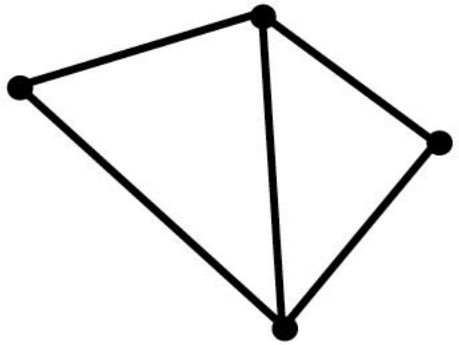
Real-time graphics primitives (entities)

- Represent surfaces as 3D **triangle meshes**





Vertices
(points in space)



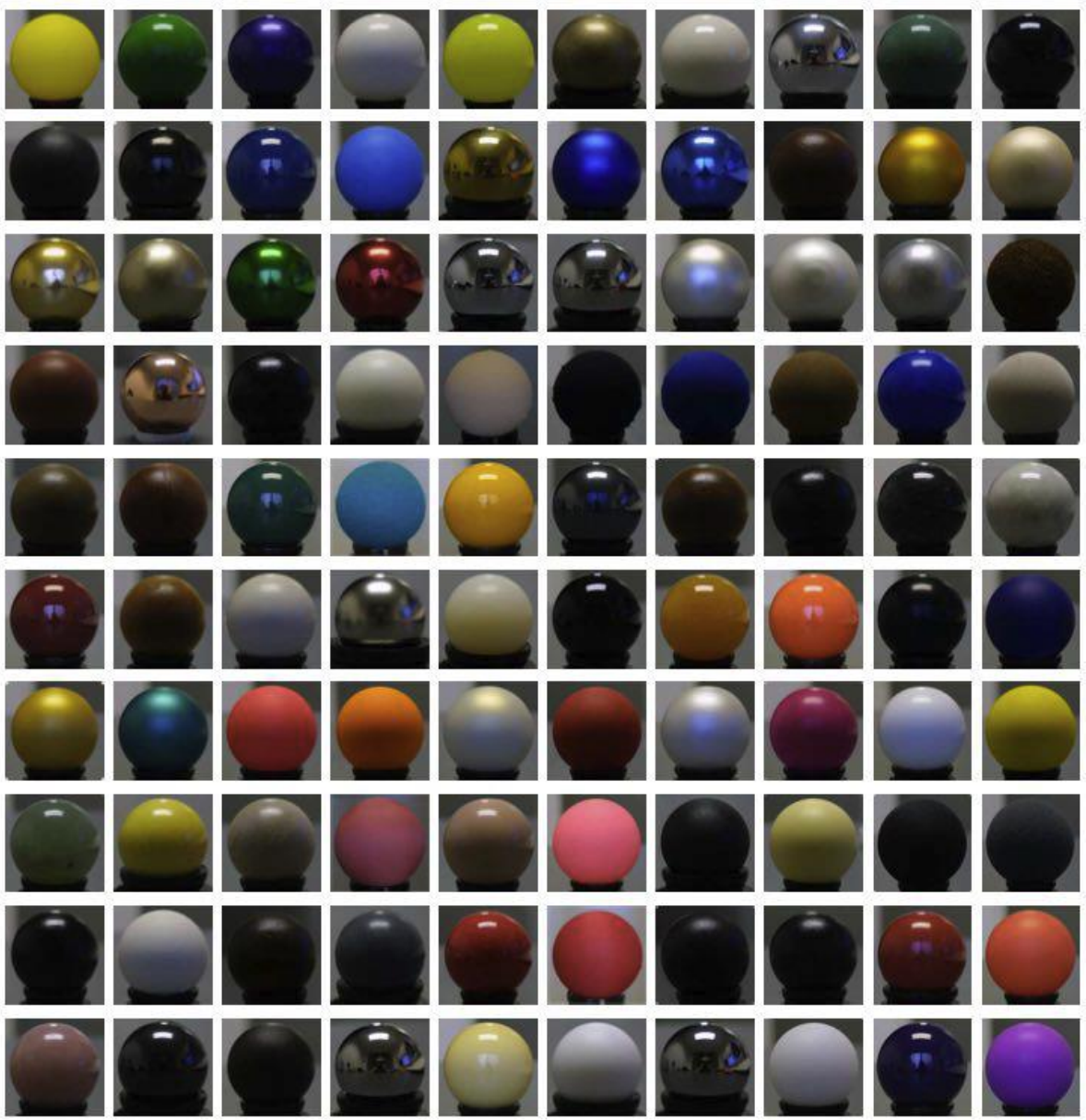
Primitives
(e.g., triangles, points, lines)

Workload in one slide

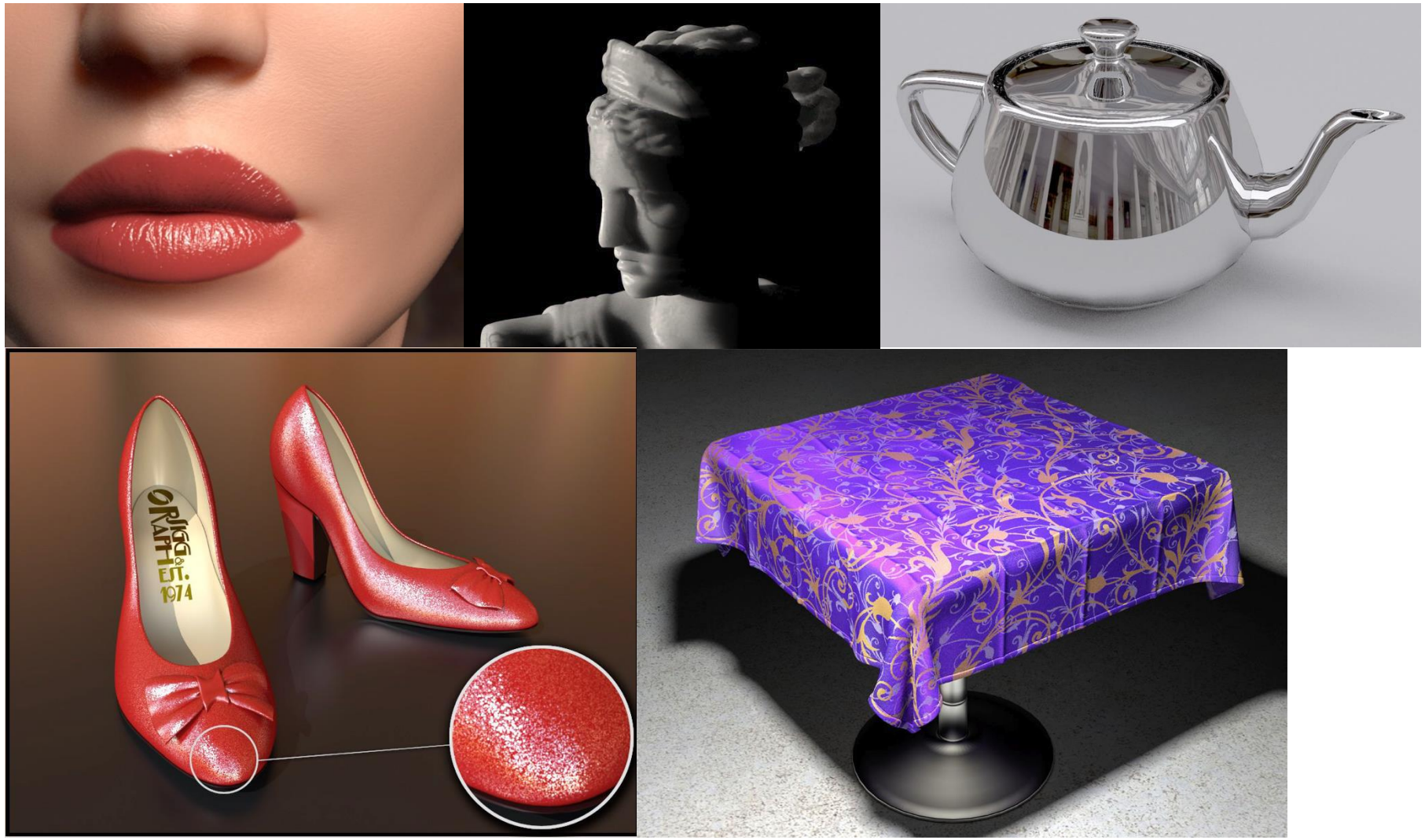
- Given a triangle, **determine where it lies on screen** given the position of a virtual camera

- For all output image pixels covered by the triangle, **compute the color** of the surface at that pixel.

What does the surface look like at a point?



Great diversity of materials and lights in the world!



Example “shader program”

OpenGL shading language (GLSL) shader program: defines behavior of fragment processing stage

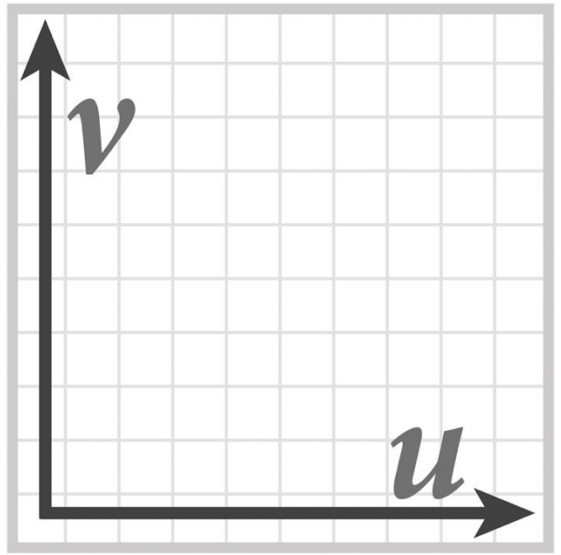
```
sampler mySamp;
Texture2D<float3> myTex;
float3 lightDir;

float4 myShader(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);
}
```

read-only global variables

“Shader” function
(the function invoked to compute the color of the pixel)

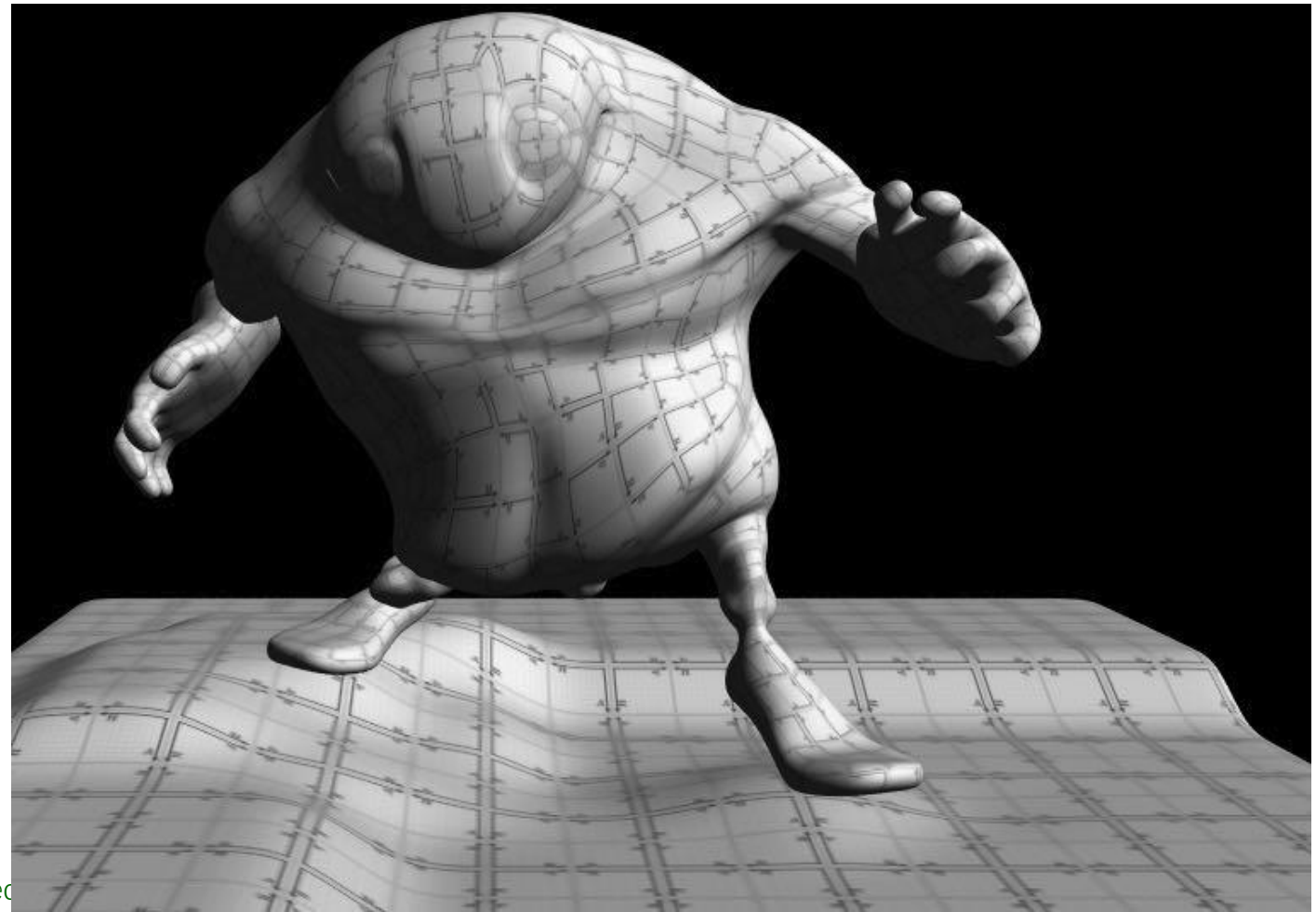
per-pixel output: RGBA surface color at pixel

myTexture is a texture map



Shaded result

- Image contains output of `myShader()` for each pixel covered by surface
  - pixels covered by multiple surfaces contain output from surface closest to camera



Example “shader program”

OpenGL shading language (GLSL) shader program: defines behavior of fragment processing stage

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

float4 myShader(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);
}
```

- How much compute is this? 4k @ 60fps

4 multiply-adds & 1 texture fetch

4K = 8 MPixels

x 5x Overdraw = 40 MPixels / frame

x 60hz = 2.4 GPixels/sec

~10 GFLOPS

A real game is 10s to 100s of times more!

Outline

- GPU Architecture
  - Three major ideas that make GPU processing cores run fast
  - Closer look at real GPU designs
  - The GPU memory hierarchy: moving data to processors
- General Purpose GPU (GPGPU)
- GPU Programming Model
- Program a GPU with CUDA

GPU Architecture: Three Big Ideas

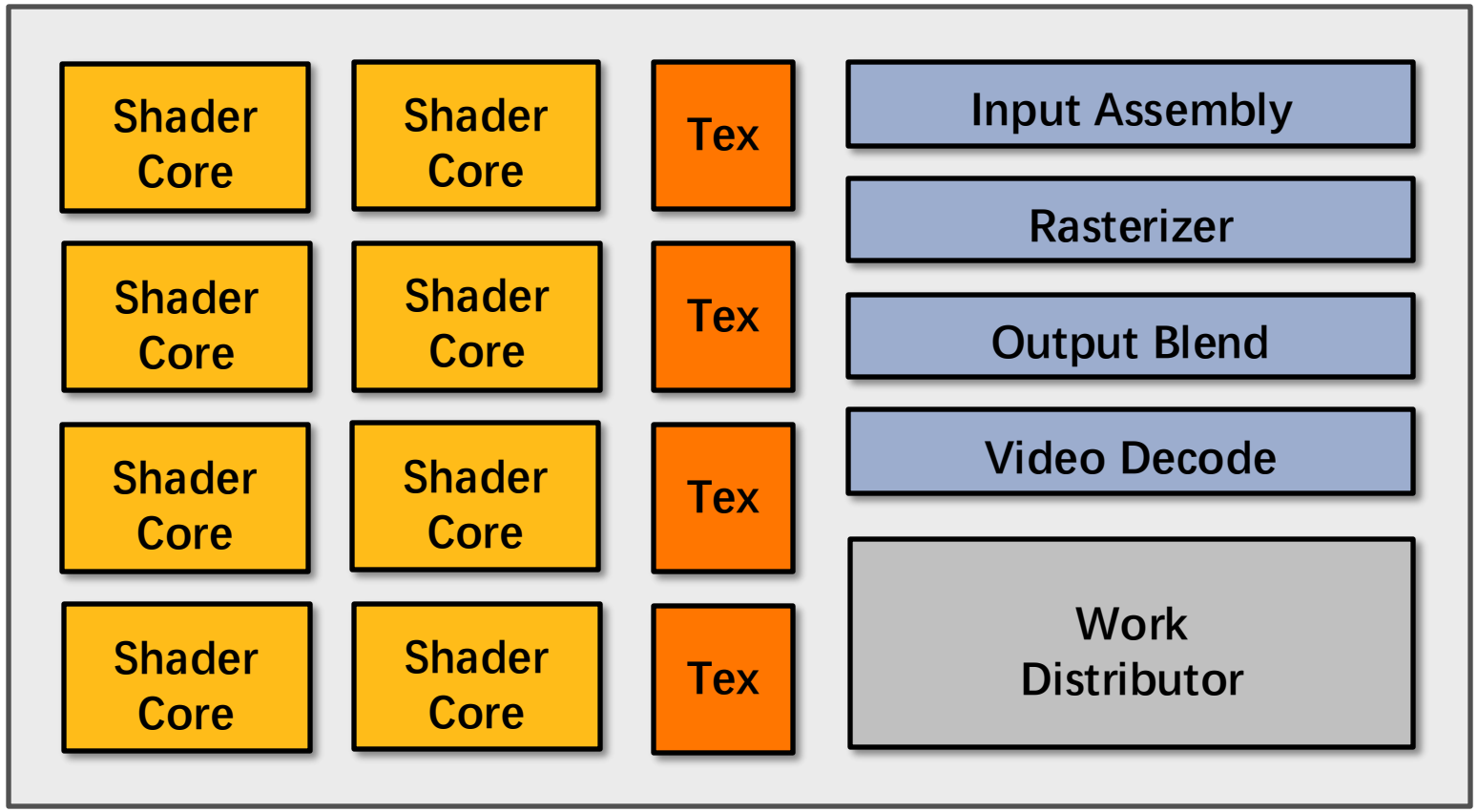
Throughput processing

- Three key concepts behind how modern GPUs run code

- Knowing these concepts will help you:
  - Understand space of GPU core designs
  - Understand how “GPU” cores do (and don’t!) differ from “CPU” cores
  - Optimize shaders/compute kernels
  - Establish intuition: what workloads might benefit from the design of these architectures?

What's in a GPU?

- A GPU is a heterogeneous chip multi-processor (highly tuned for graphics)



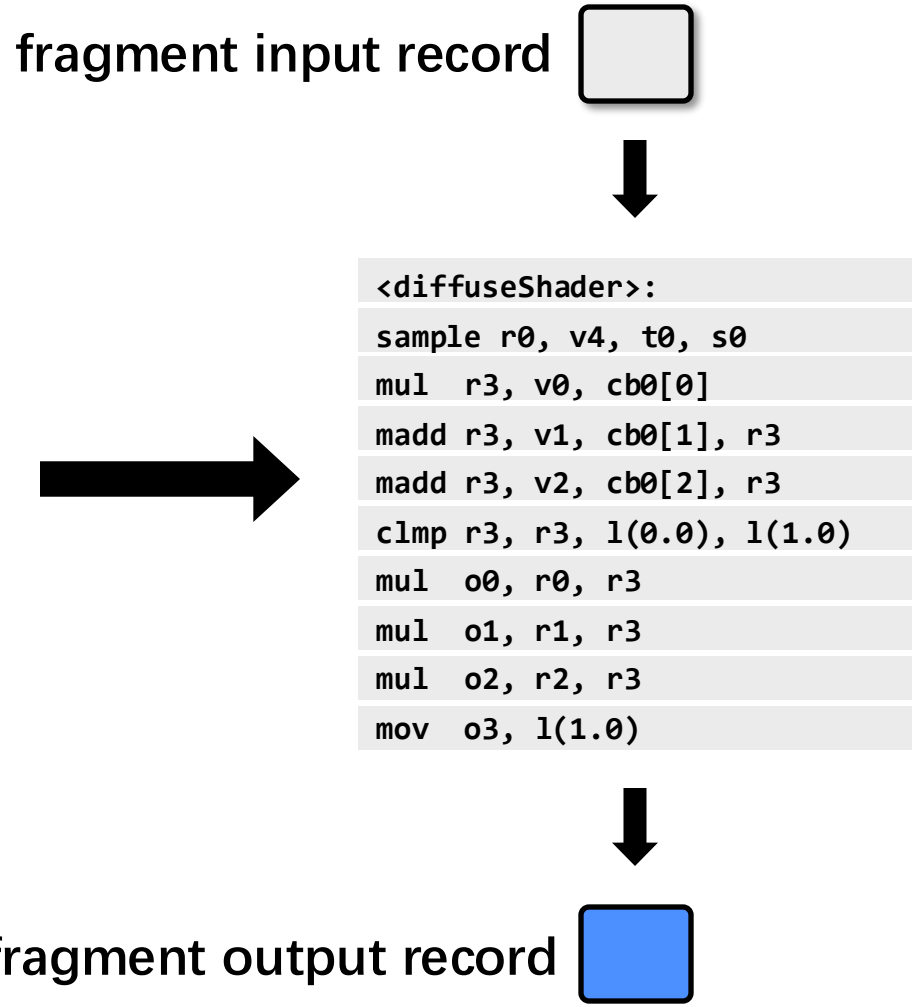
A diffuse reflectance shader

```
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);
}
```

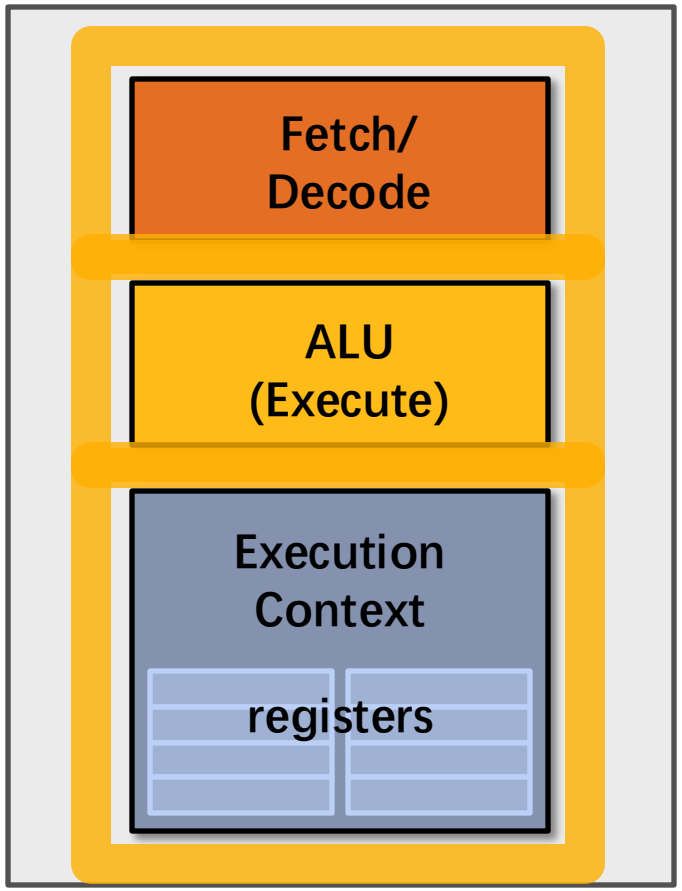
- Shader programming model
  - Fragments are processed **independently**,
  - but there is no explicit parallel programming

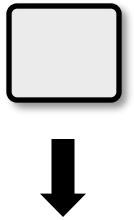
Compile shader

```
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);
}
```

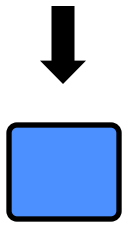


Execute shader

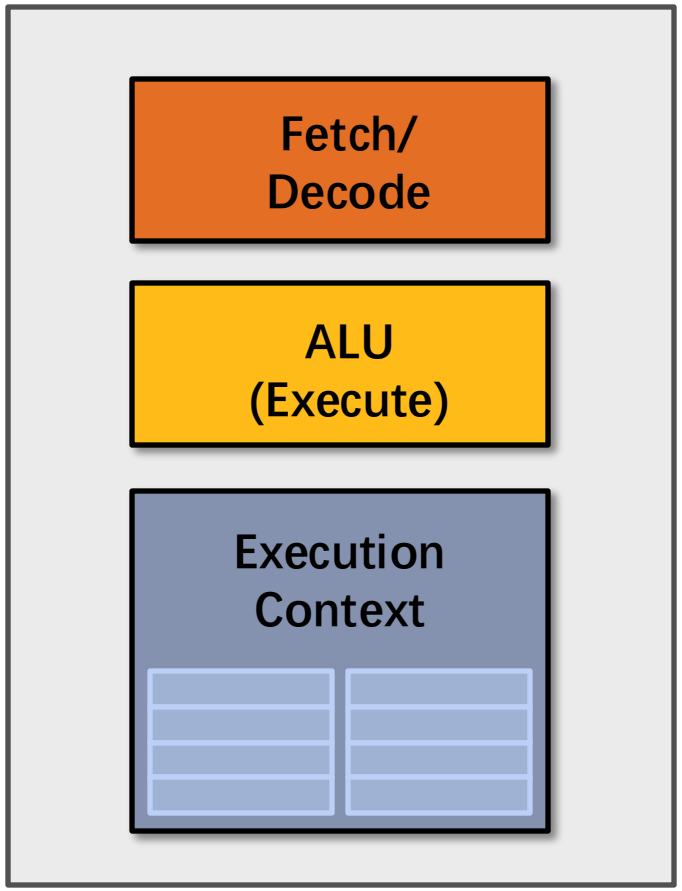


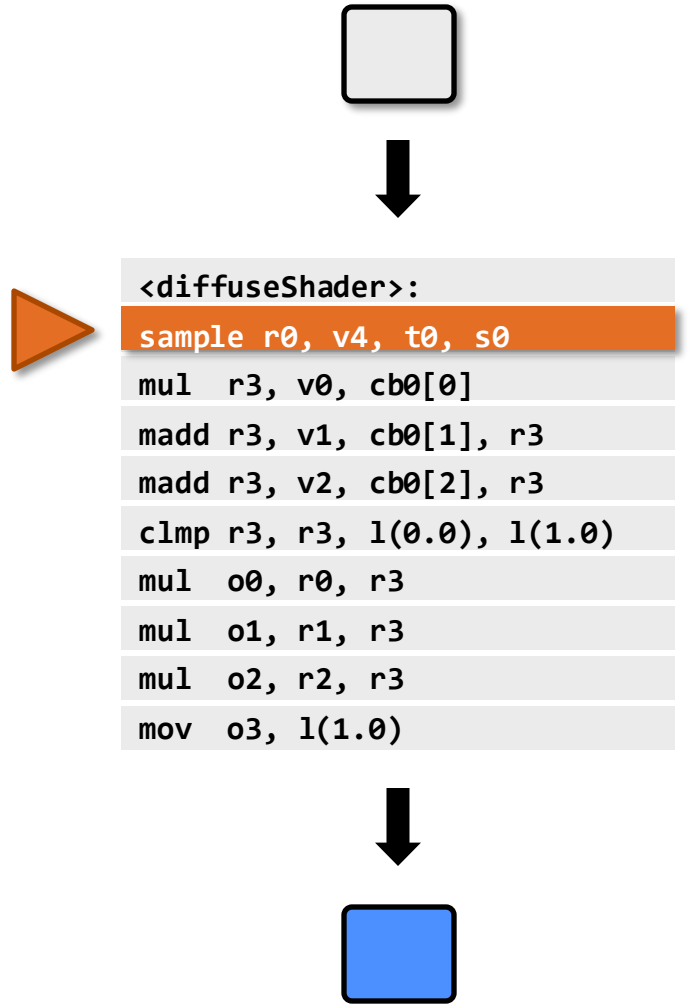


```
<diffuseShader>:
sample r0, v4, t0, s0
mul  r3, v0, cb0[0]
madd r3, v1, cb0[1], r3
madd r3, v2, cb0[2], r3
clmp r3, r3, l(0.0), l(1.0)
mul  o0, r0, r3
mul  o1, r1, r3
mul  o2, r2, r3
mov  o3, l(1.0)
```

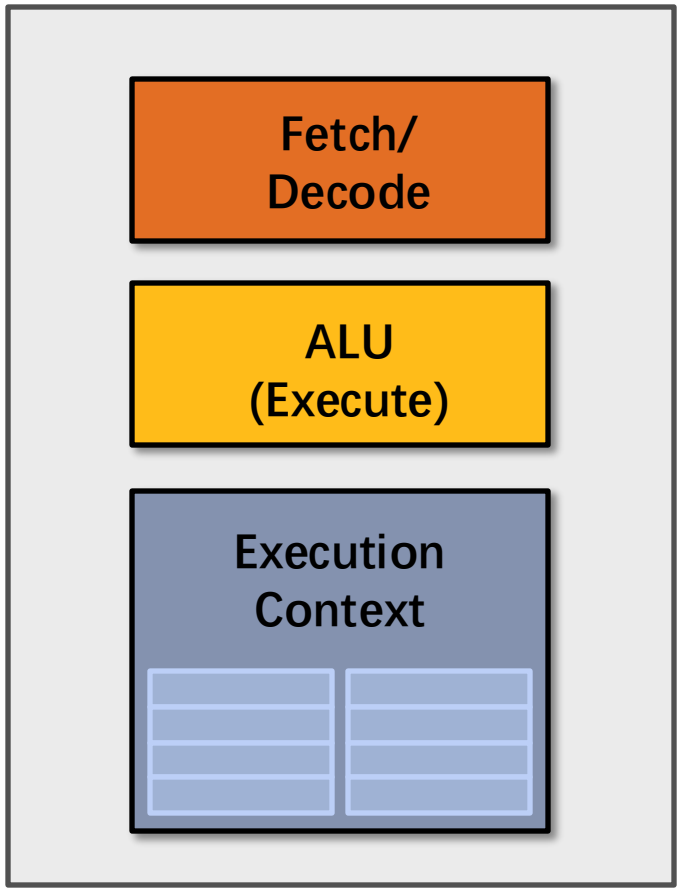


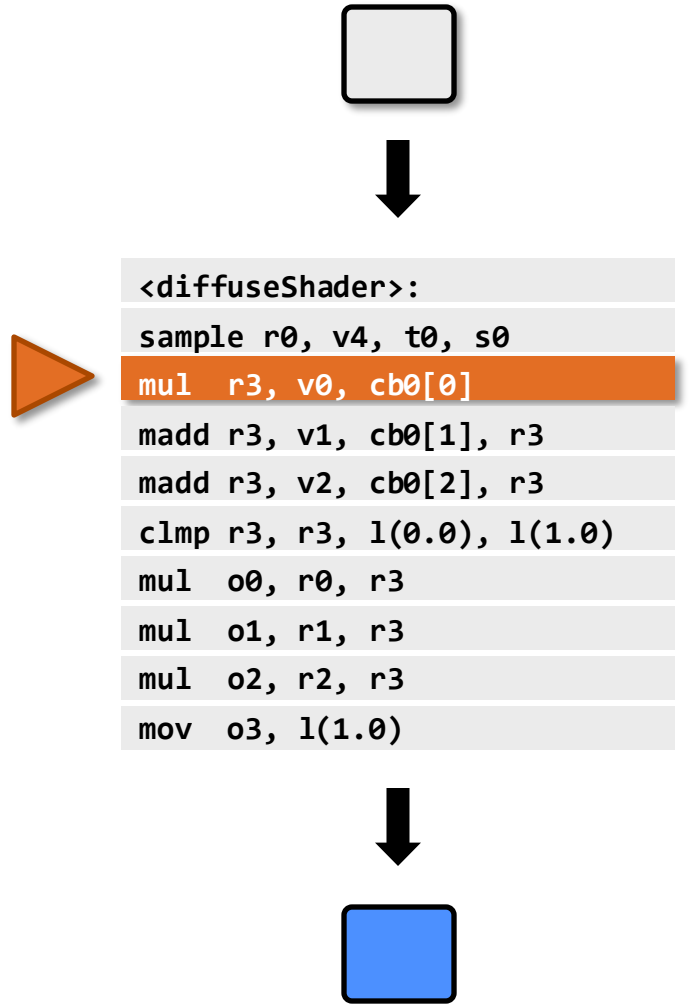
Execute shader



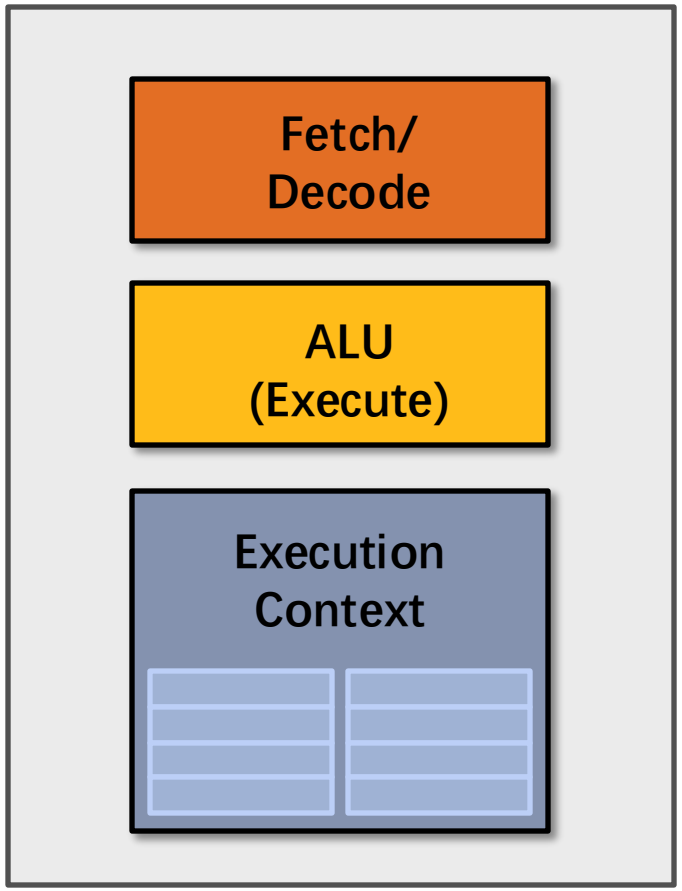


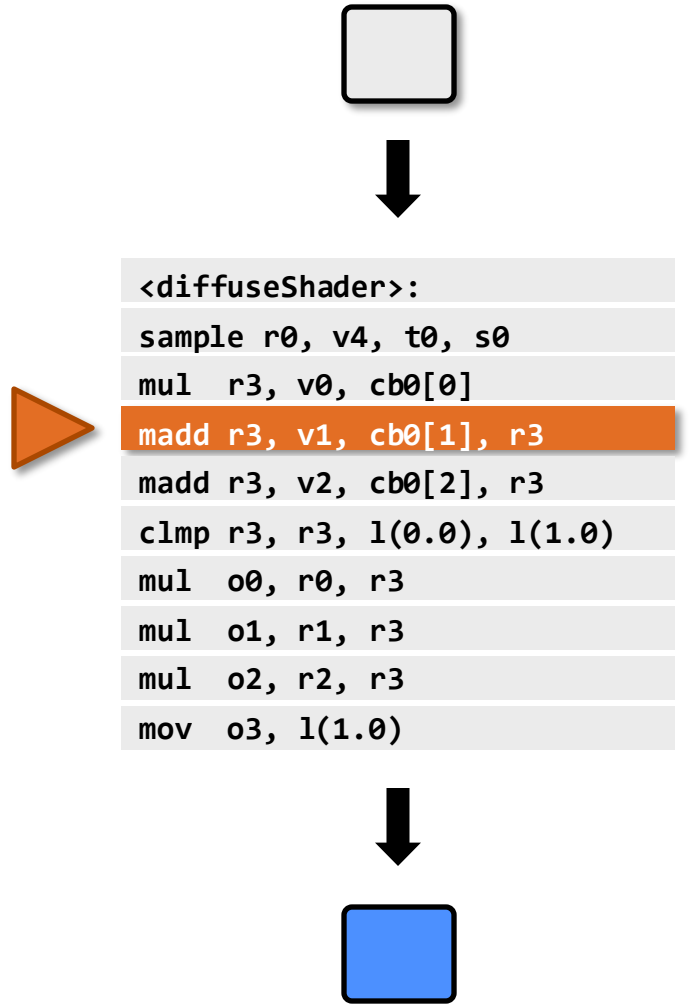
Execute shader



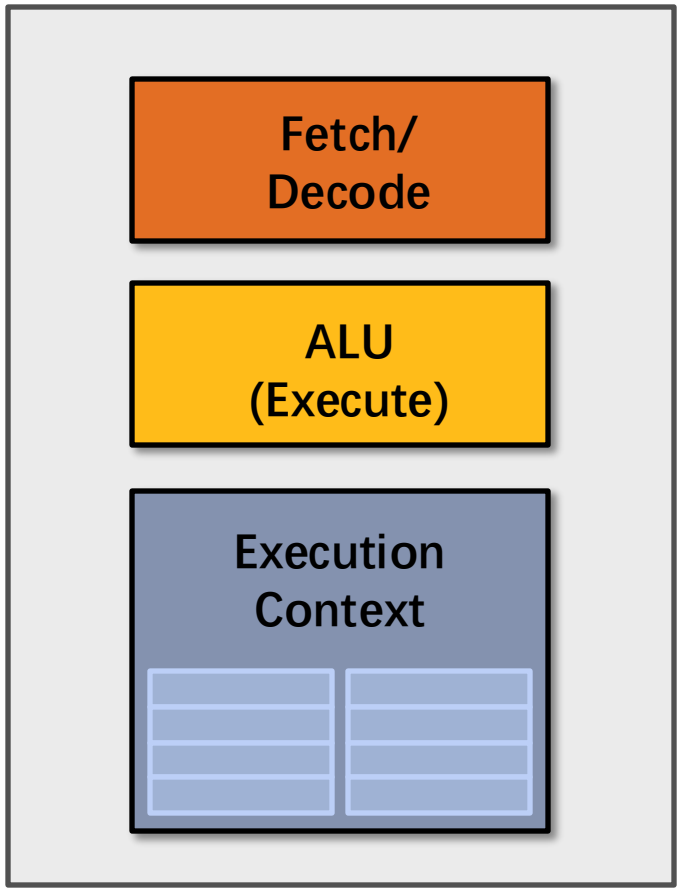


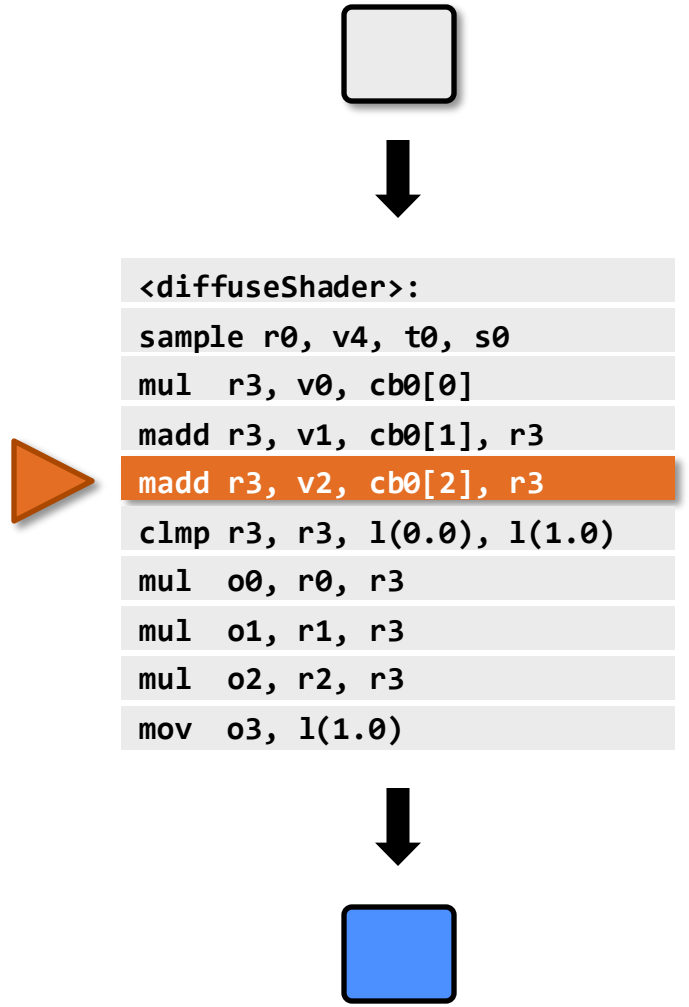
Execute shader



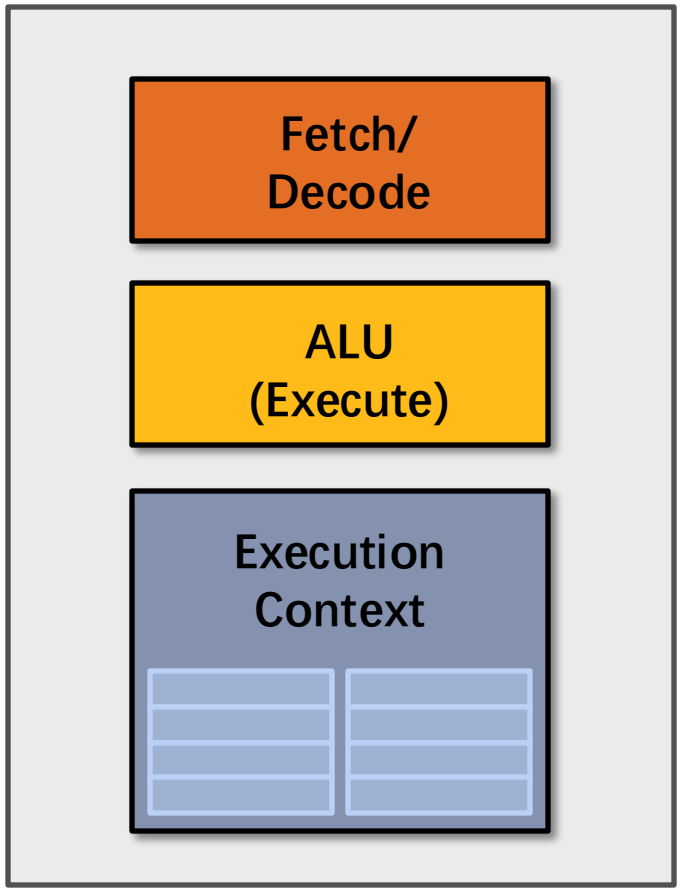


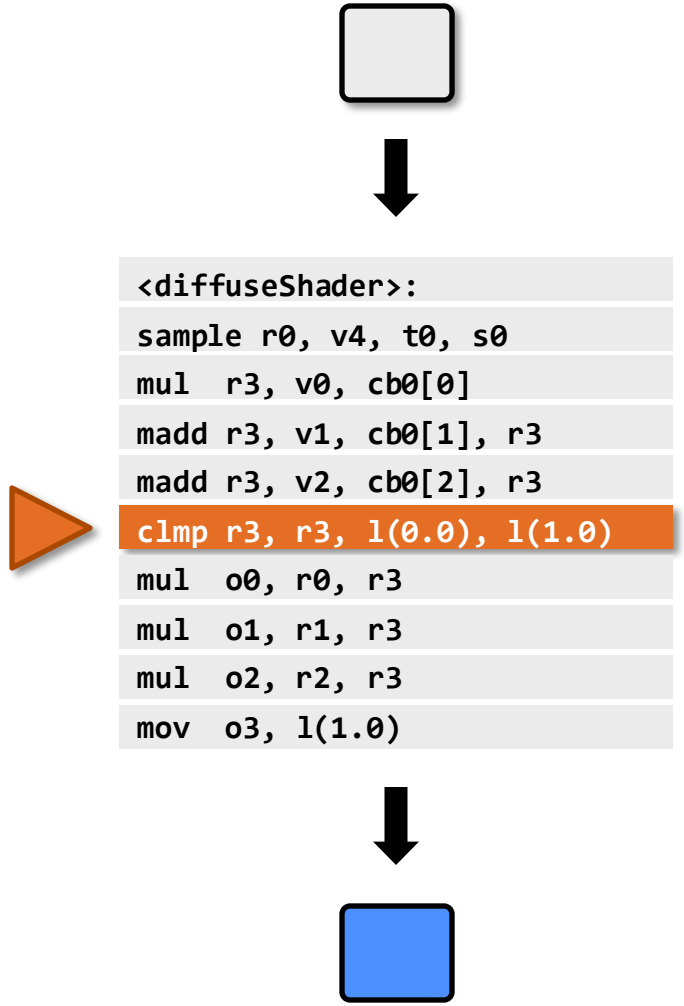
Execute shader



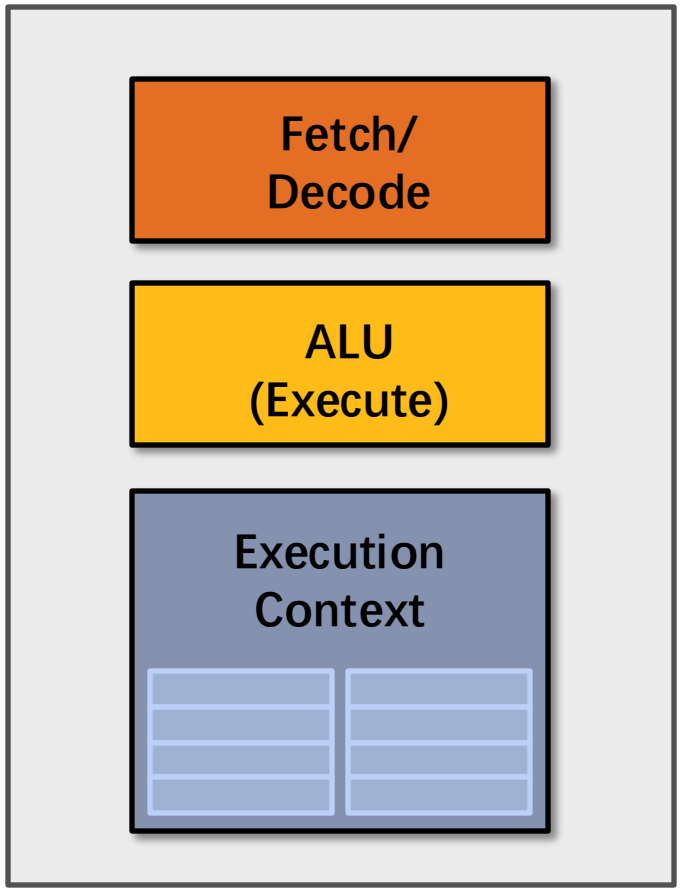


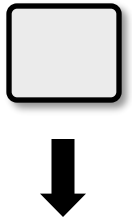
Execute shader



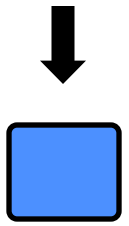


Execute shader

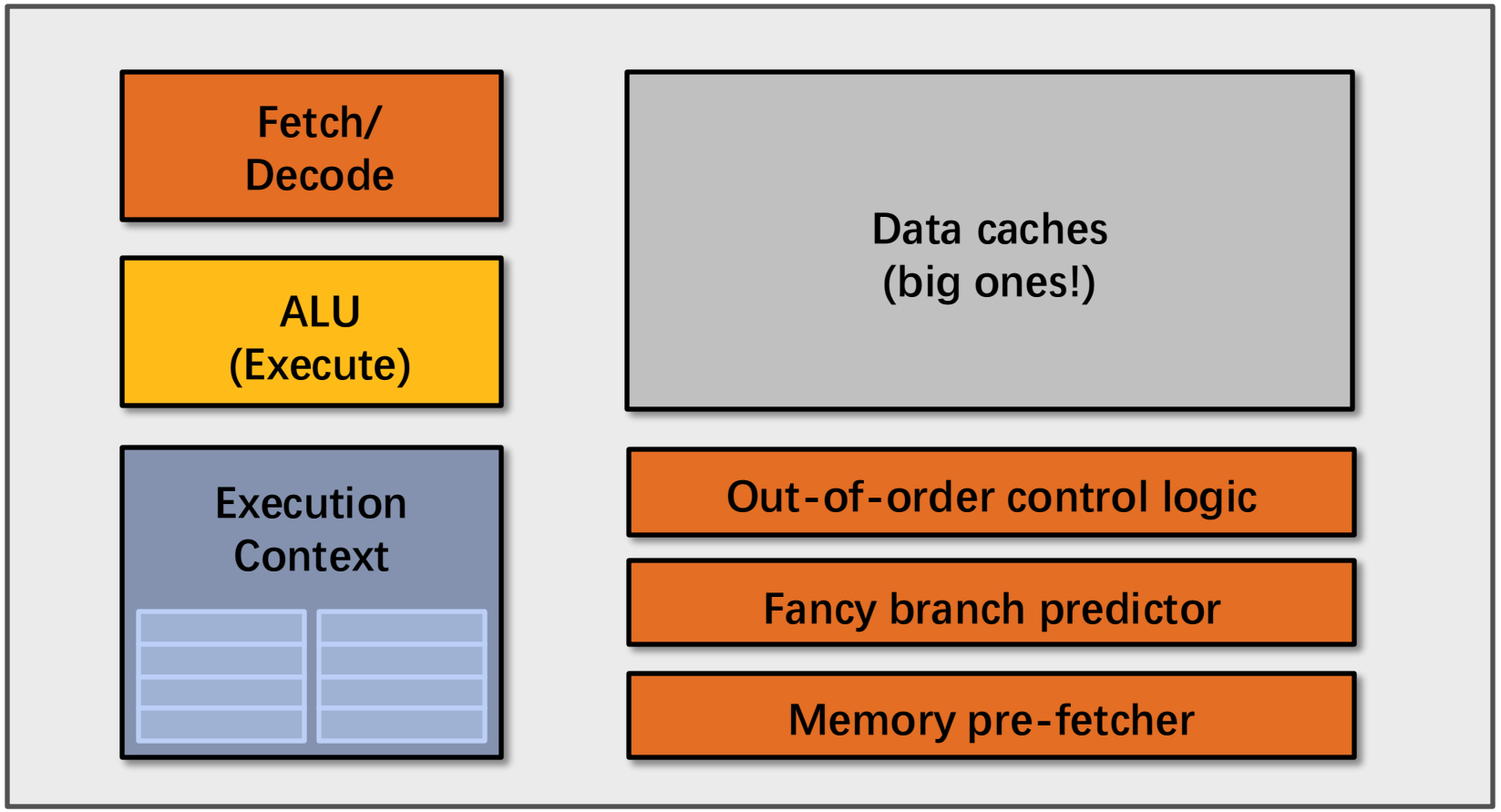




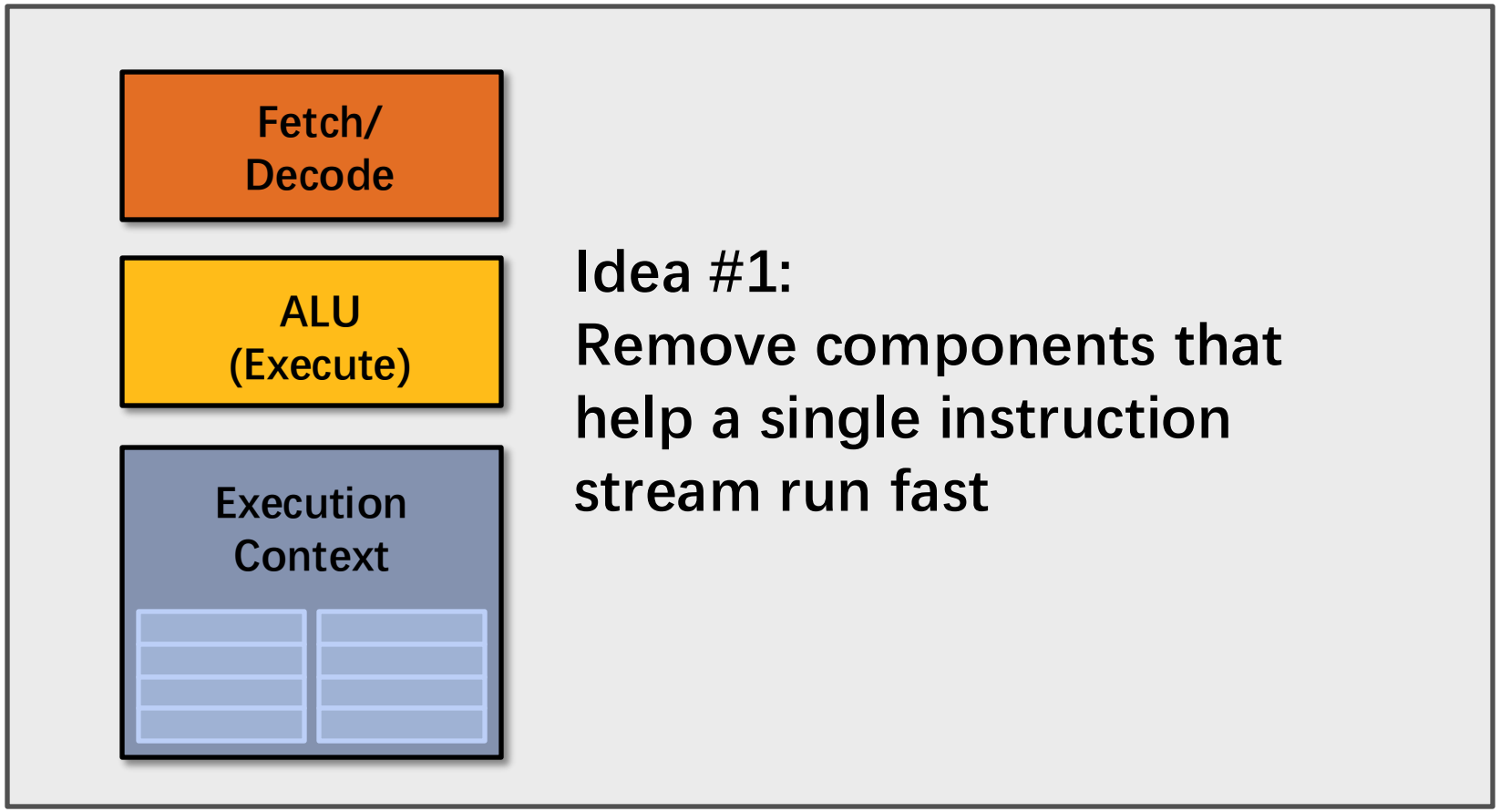
```
<diffuseShader>:
sample r0, v4, t0, s0
mul  r3, v0, cb0[0]
madd r3, v1, cb0[1], r3
madd r3, v2, cb0[2], r3
clmp r3, r3, l(0.0), l(1.0)
mul  o0, r0, r3
mul  o1, r1, r3
mul  o2, r2, r3
mov  o3, l(1.0)
```



“CPU-style” cores

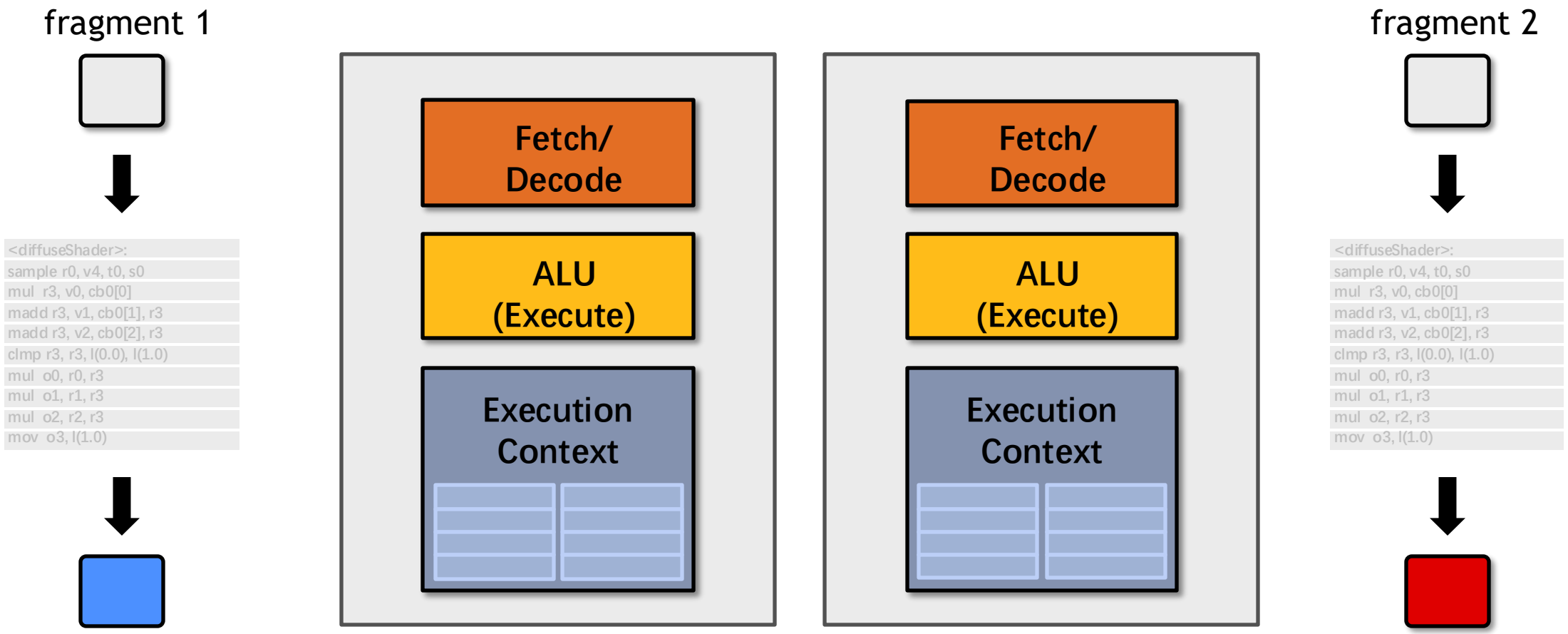


Slimming down

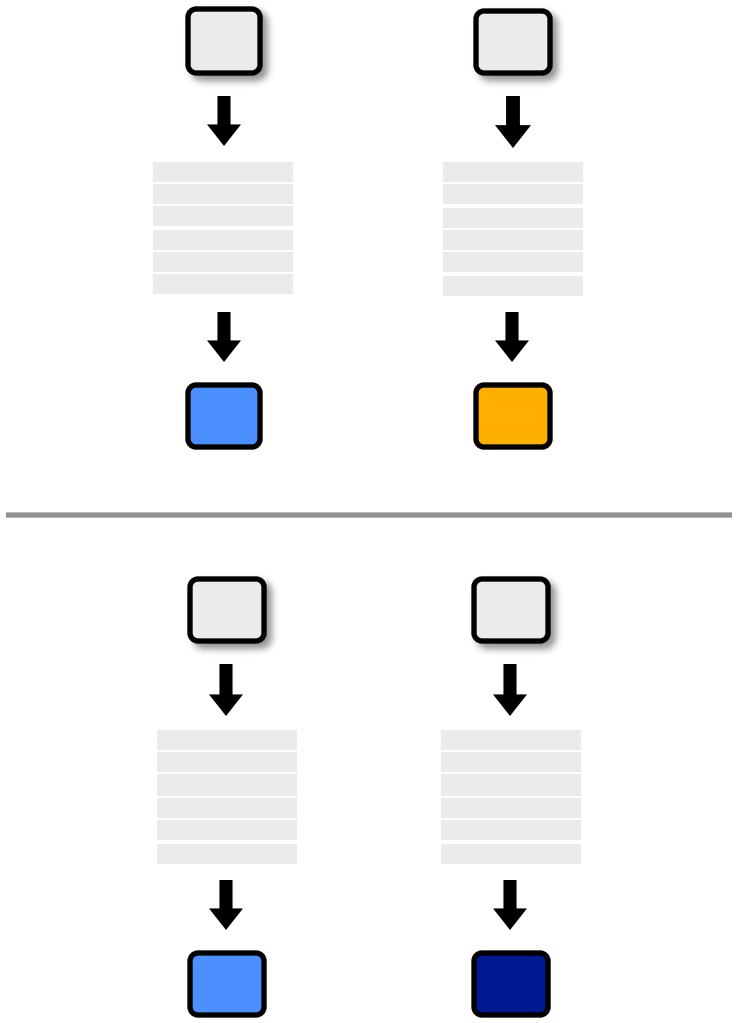


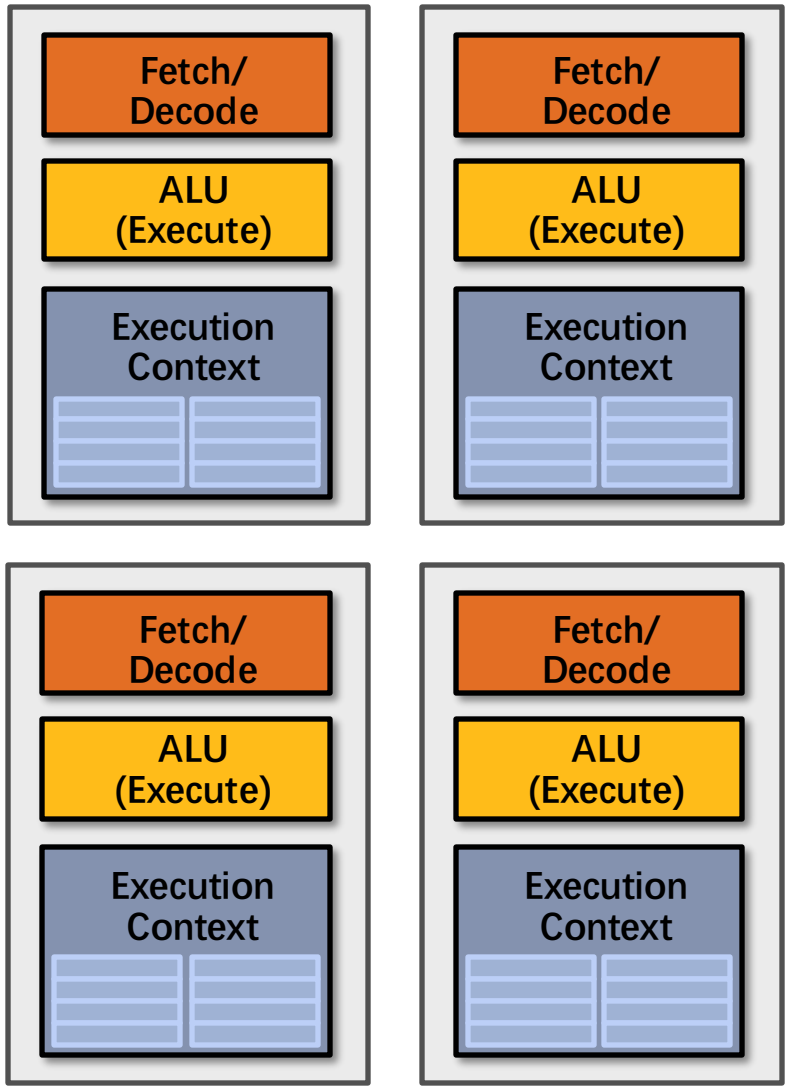
Idea #1:
Remove components that help a single instruction stream run fast

Two cores (two fragments in parallel)

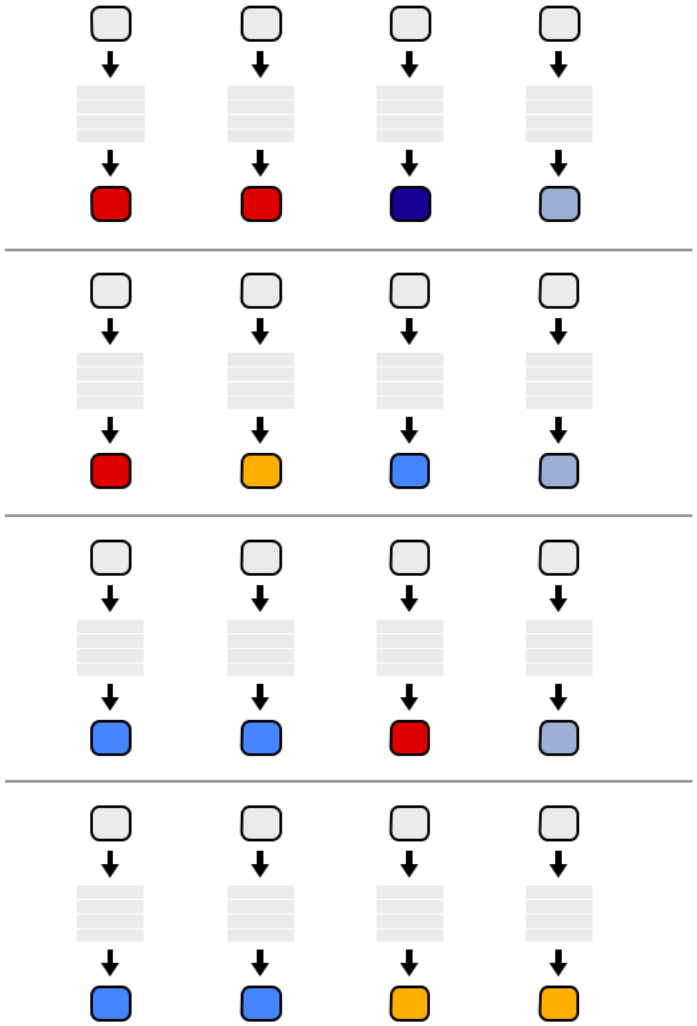


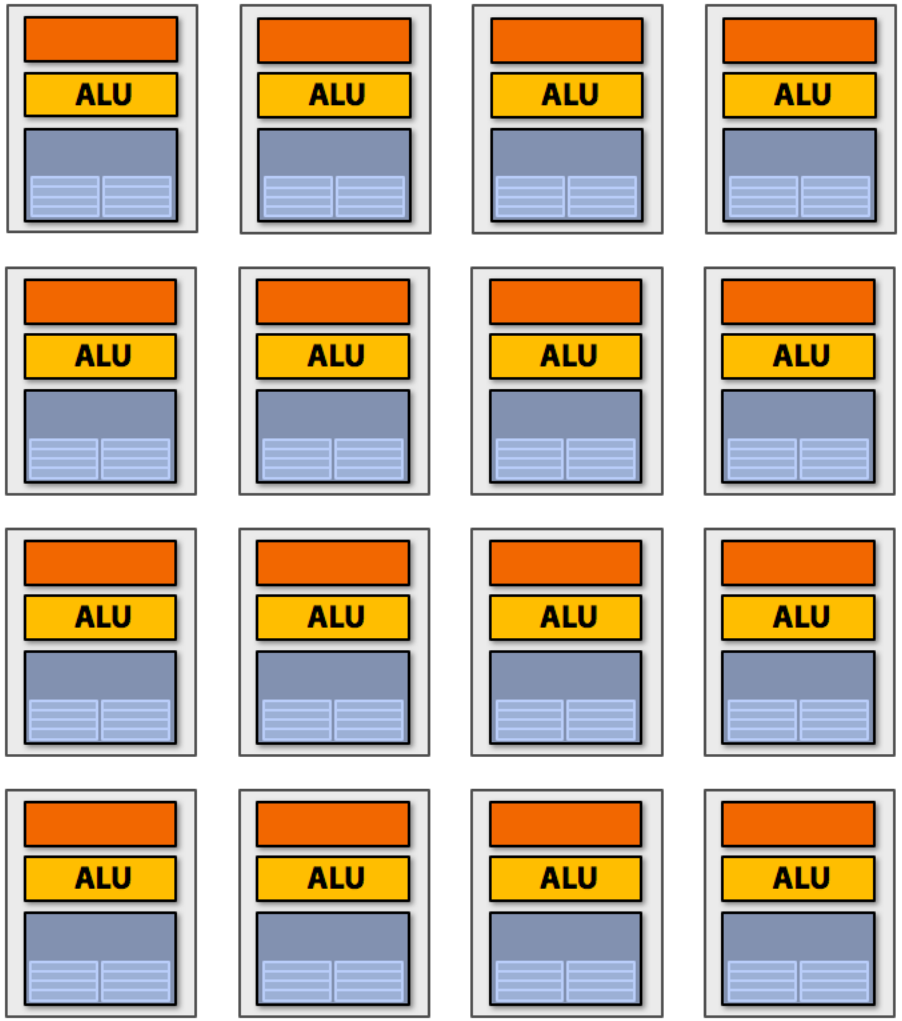
Four cores (four fragments in parallel)





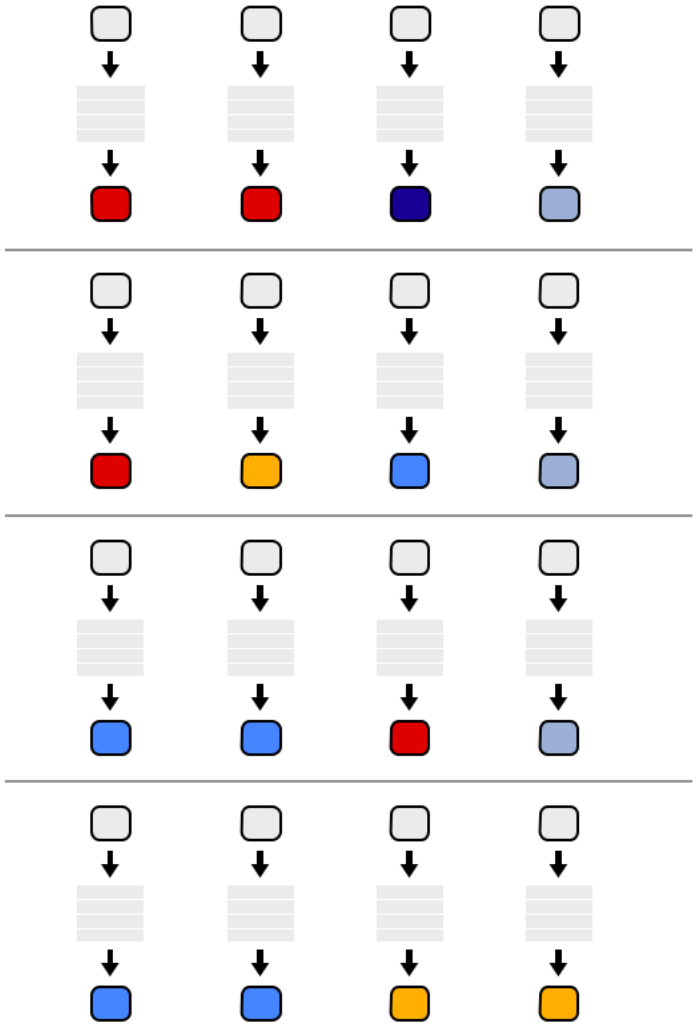
Sixteen cores (sixteen fragments in parallel)





16 cores = 16 simultaneous instruction streams

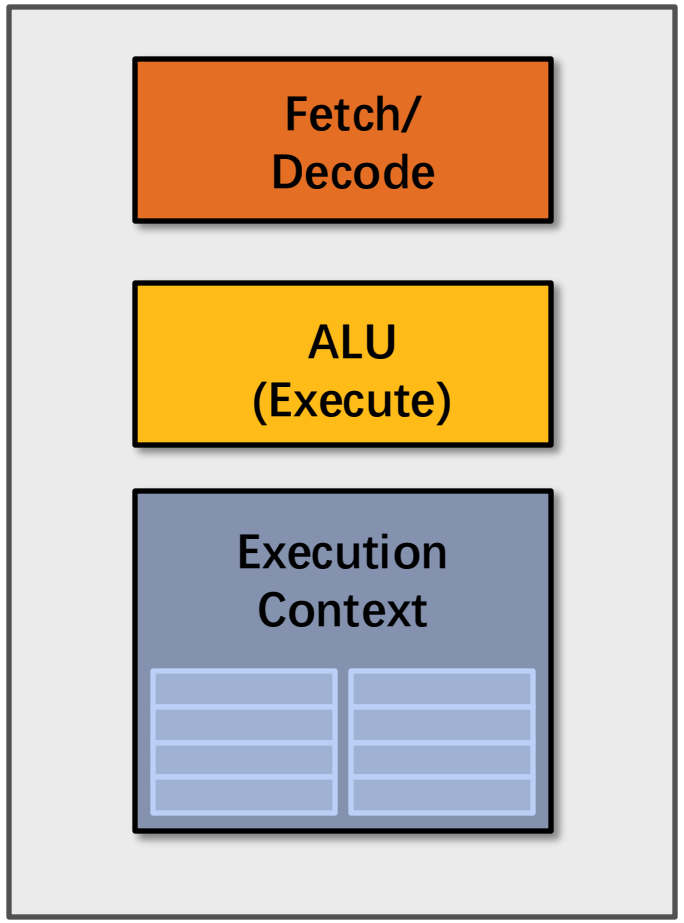
Instruction stream sharing



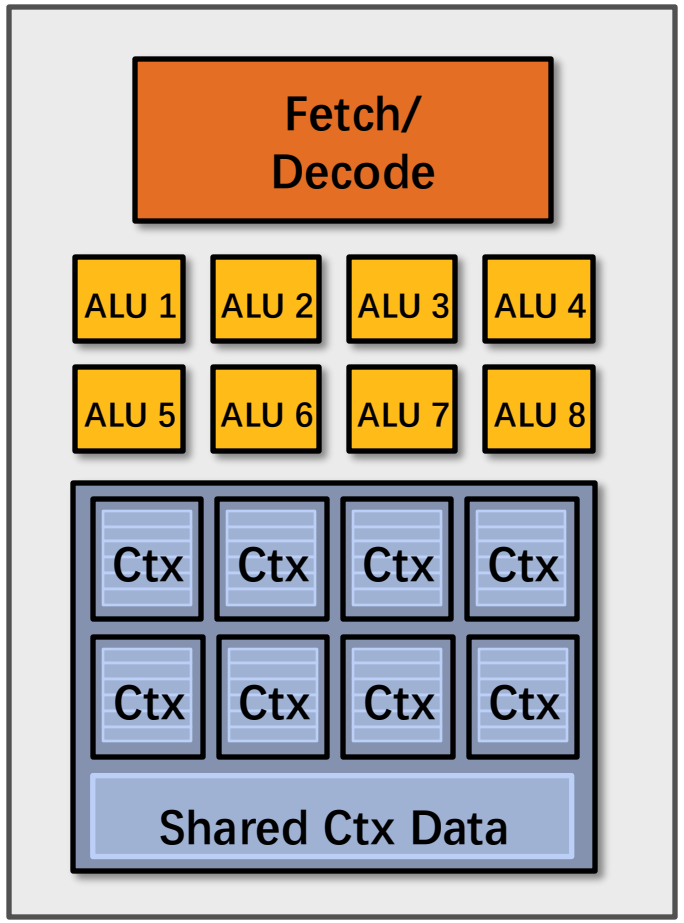
But … many fragments should be able to share an instruction stream!

```
<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

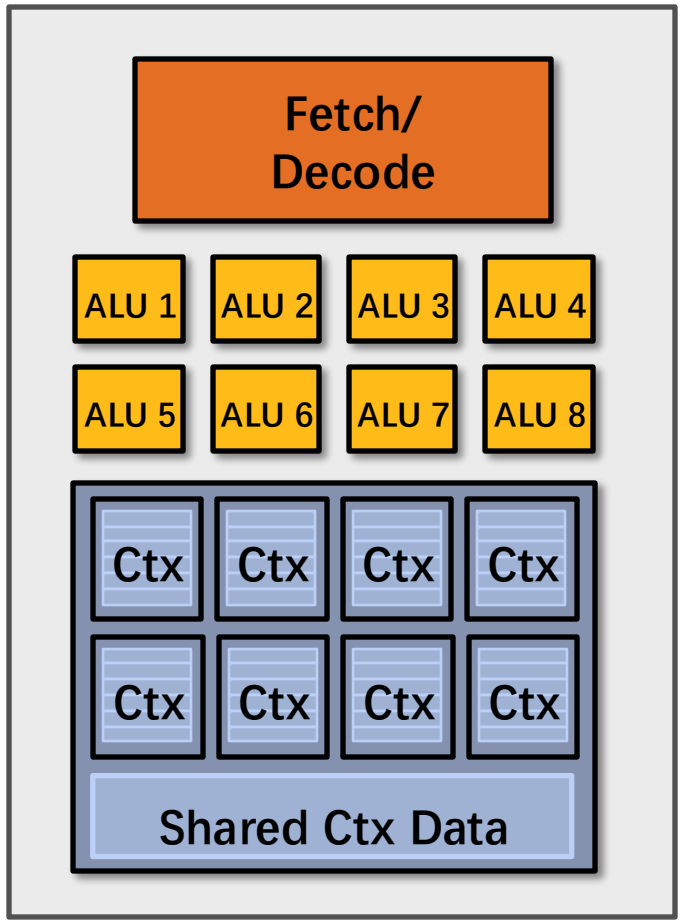


Idea #2:

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

SIMD processing

Modifying the shader

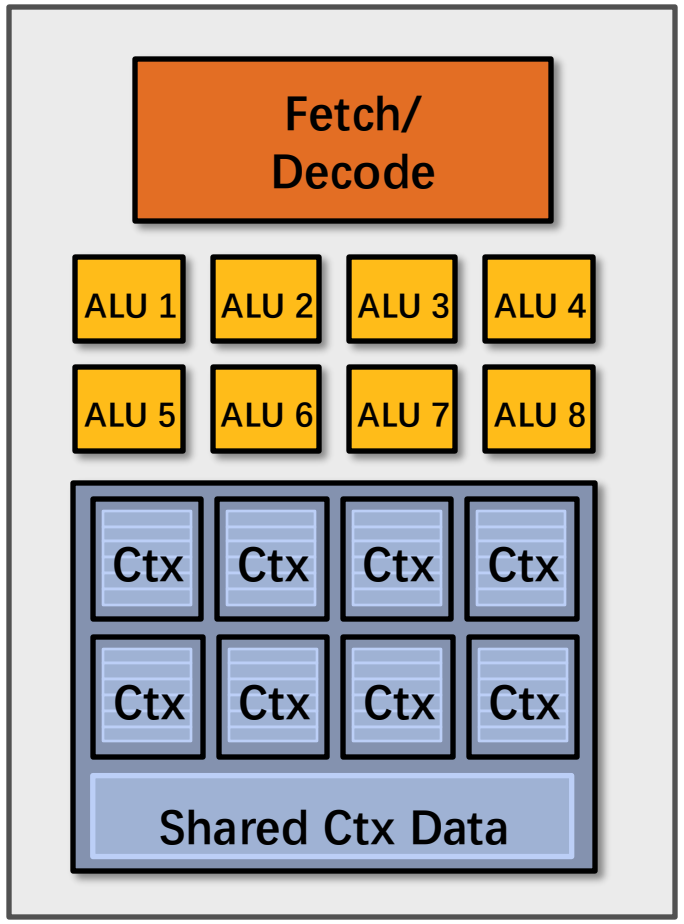



```
<diffuseShader>:
sample r0, v4, t0, s0
mul  r3, v0, cb0[0]
madd r3, v1, cb0[1], r3
madd r3, v2, cb0[2], r3
clmp r3, r3, l(0.0), l(1.0)
mul  o0, r0, r3
mul  o1, r1, r3
mul  o2, r2, r3
mov  o3, l(1.0)
```

Original compiled shader:

Processes one fragment using scalar ops on scalar registers

Modifying the shader

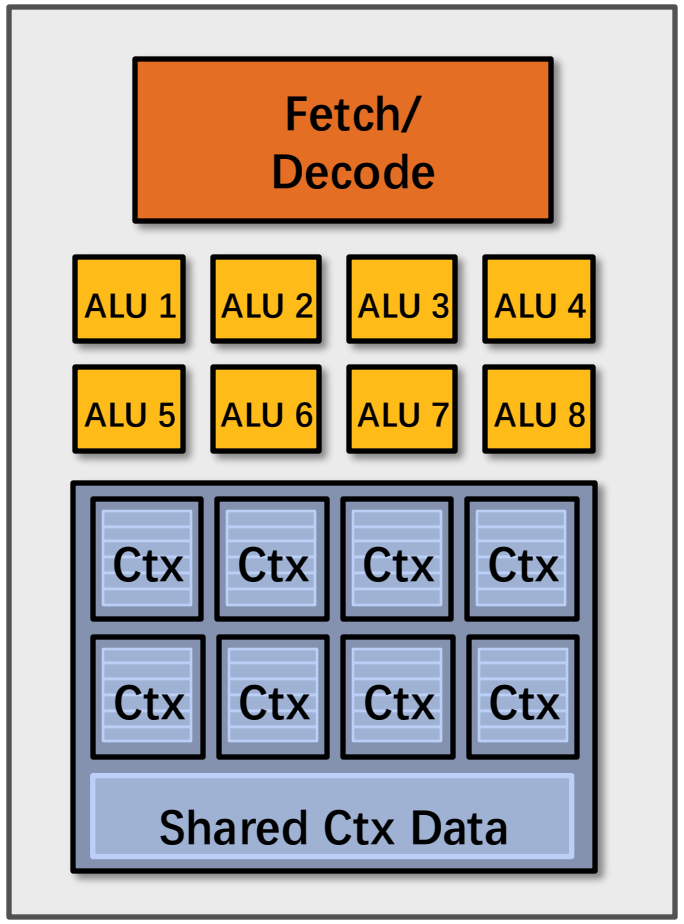


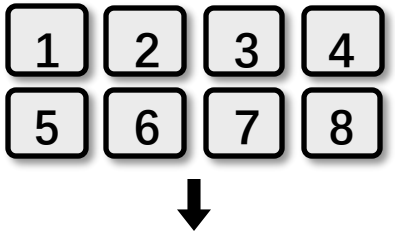

```
<VEC8_diffuseShader>:
VEC8_sample vec_r0, vec_v4, t0, vec_s0
VEC8_mul    vec_r3, vec_v0, cb0[0]
VEC8_madd   vec_r3, vec_v1, cb0[1], vec_r3
VEC8_madd   vec_r3, vec_v2, cb0[2], vec_r3
VEC8_clmp   vec_r3, vec_r3, l(0.0), l(1.0)
VEC8_mul    vec_o0, vec_r0, vec_r3
VEC8_mul    vec_o1, vec_r1, vec_r3
VEC8_mul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

New compiled shader:

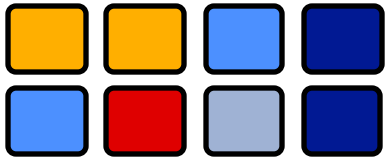
Processes eight fragments using vector ops on vector registers

Modifying the shader

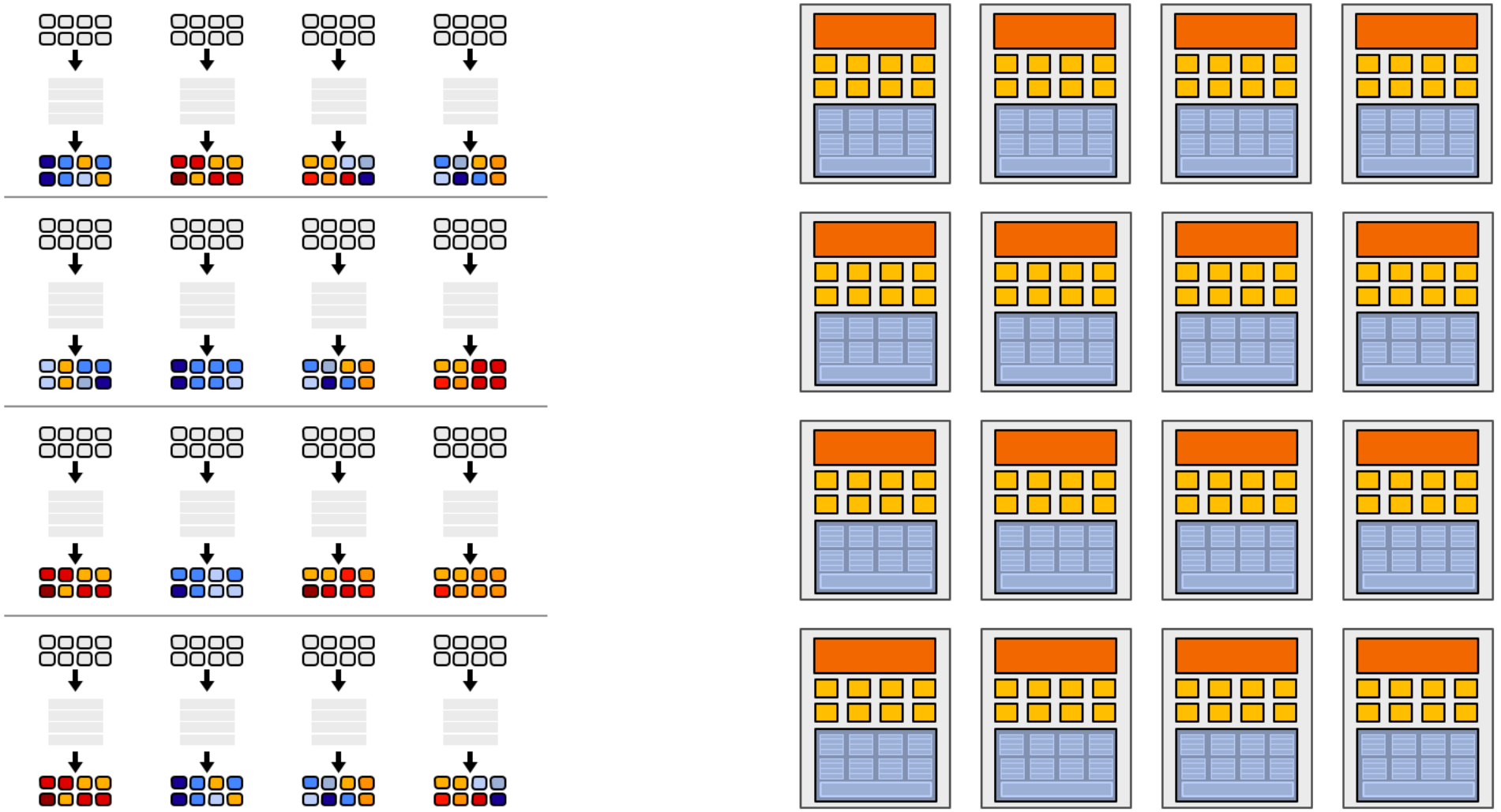





```
<VEC8_diffuseShader>:
VEC8_sample vec_r0, vec_v4, t0, vec_s0
VEC8_mul  vec_r3, vec_v0, cb0[0]
VEC8_madd vec_r3, vec_v1, cb0[1], vec_r3
VEC8_madd vec_r3, vec_v2, cb0[2], vec_r3
VEC8_clmp vec_r3, vec_r3, l(0.0), l(1.0)
VEC8_mul  vec_o0, vec_r0, vec_r3
VEC8_mul  vec_o1, vec_r1, vec_r3
VEC8_mul  vec_o2,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

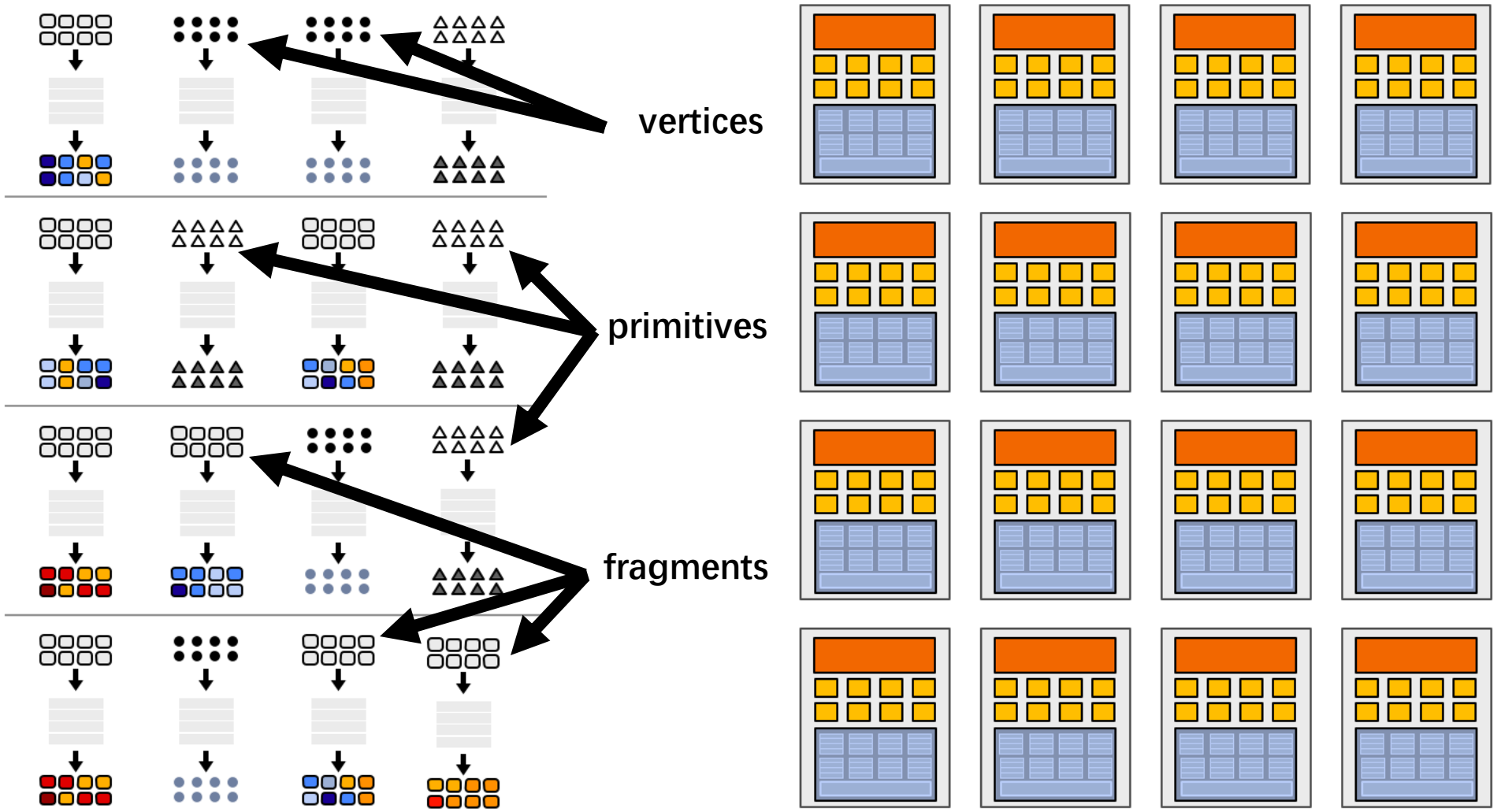


128 fragments in parallel



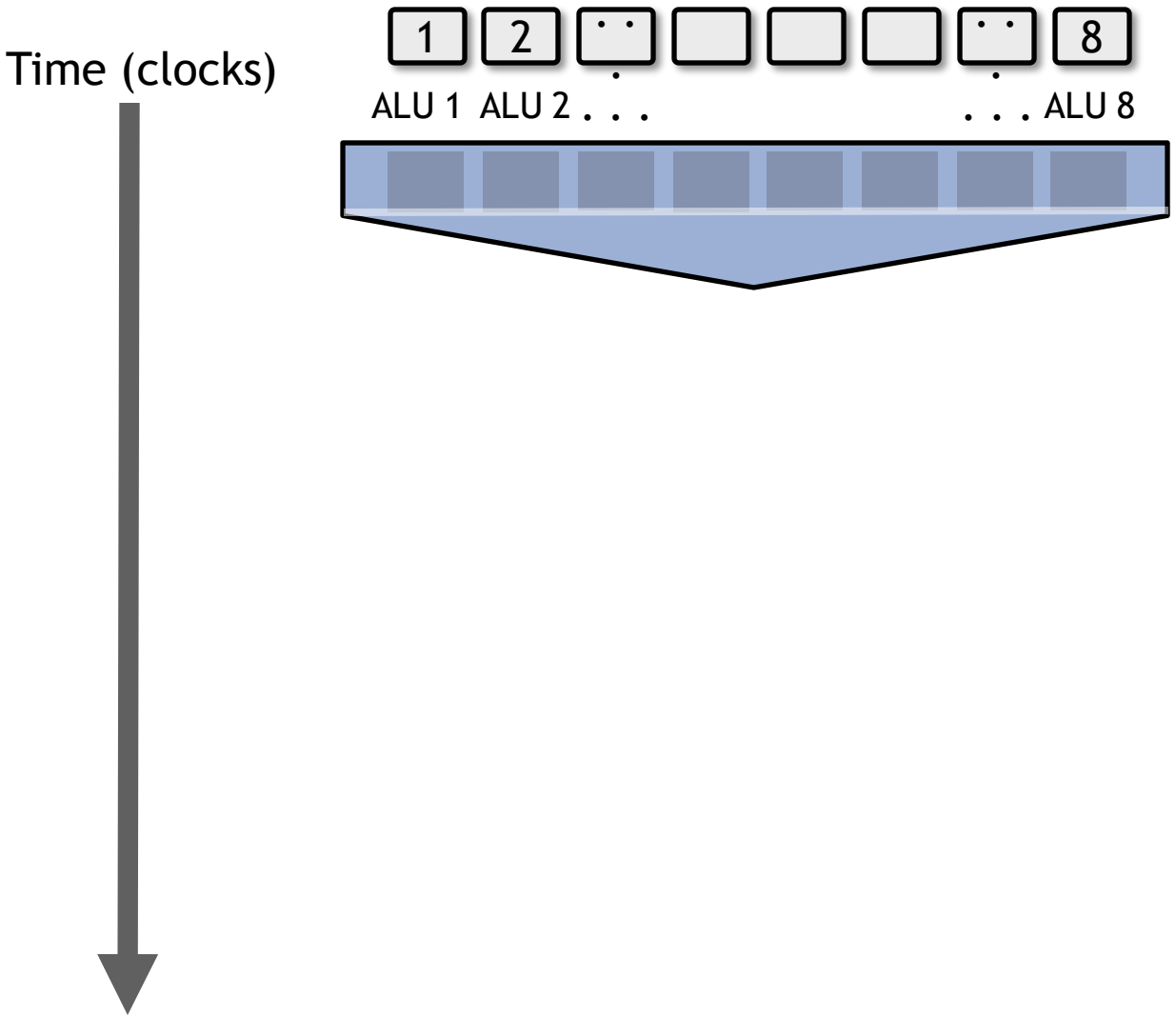
16 cores × 8 = 128 ALUs, 16 simultaneous instruction streams

128 fragments in parallel



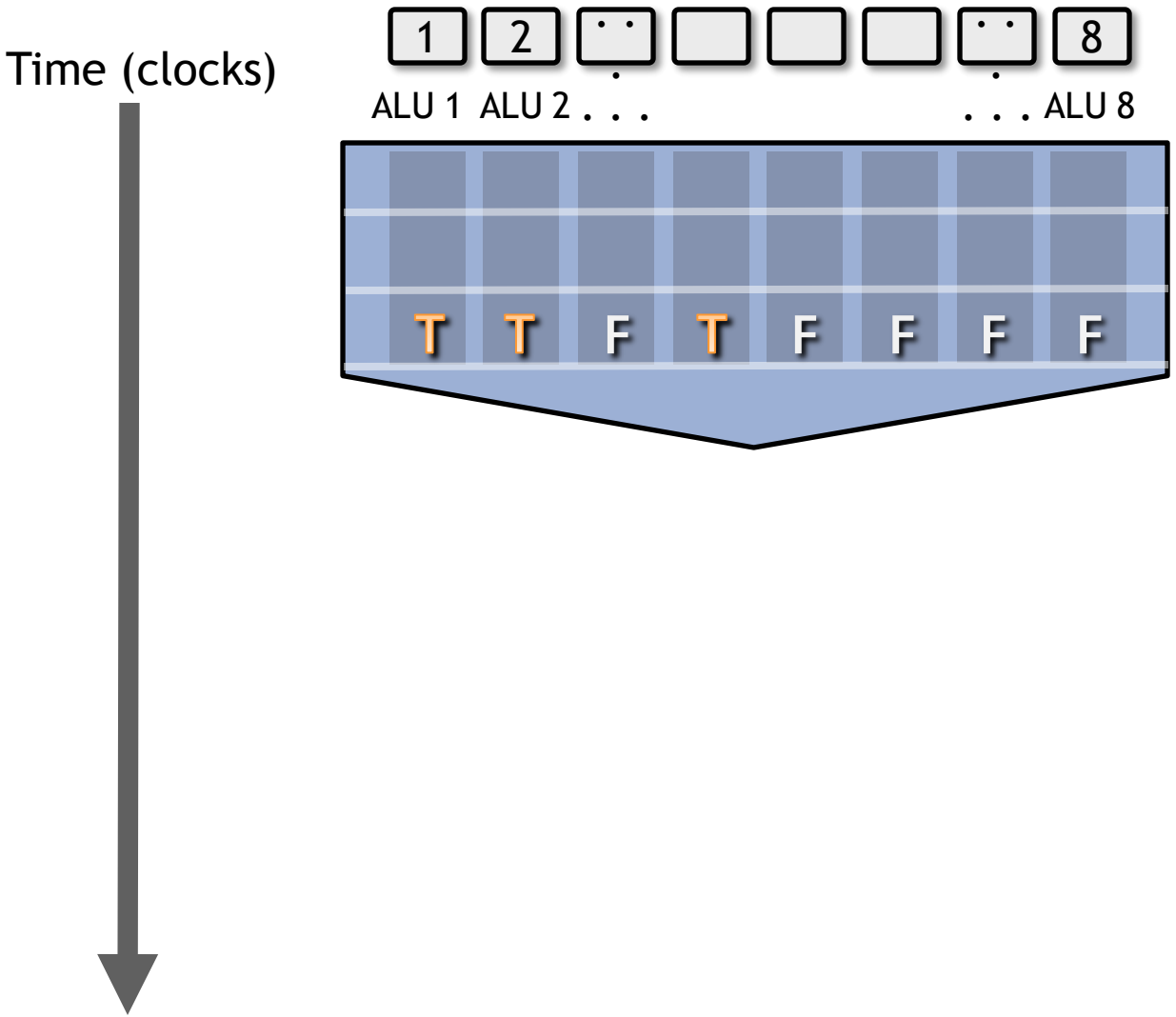
OpenCL work items / CUDA threads

But what about branches?



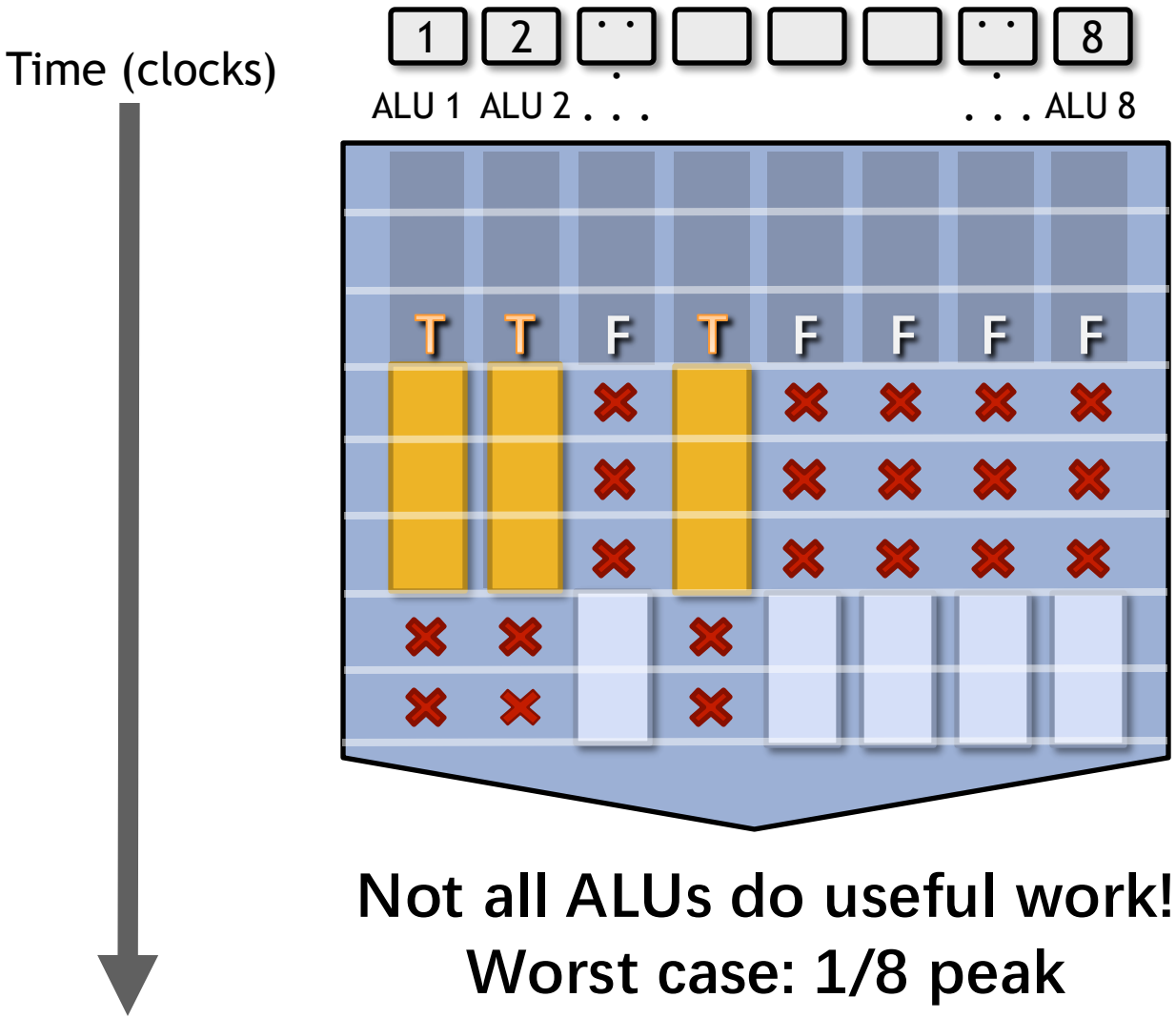

```
<unconditional
 shader code>

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>
```

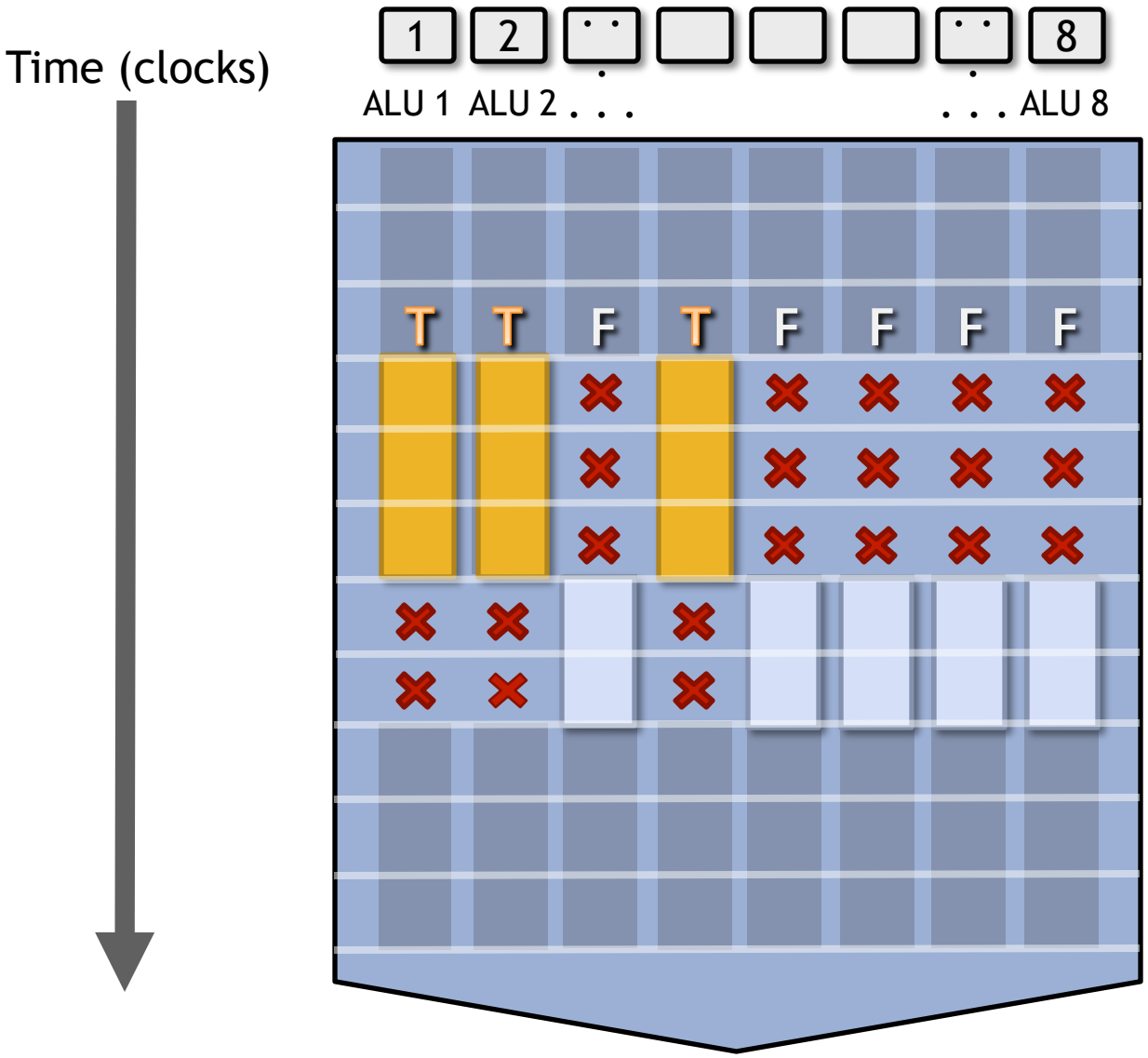
But what about branches?




```
<unconditional
 shader code>

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

<resume unconditional
shader code>
```

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

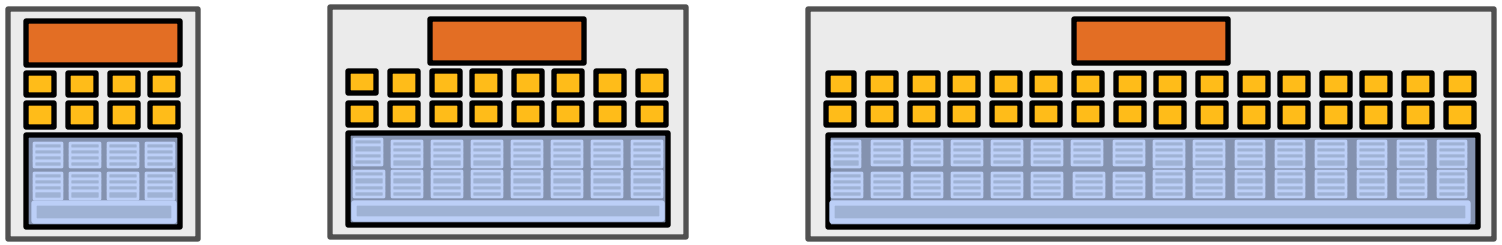
But what about branches?




```
<unconditional
 shader code>

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

<resume unconditional
shader code>
```

Clarification

- SIMD processing does not imply SIMD instructions
- Option 1: explicit vector instructions
  - x86 AVX, ARM NEON, RISC V Vectors
- 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 practice: 16 to 64 fragments share an instruction stream.

What about *stalls*?

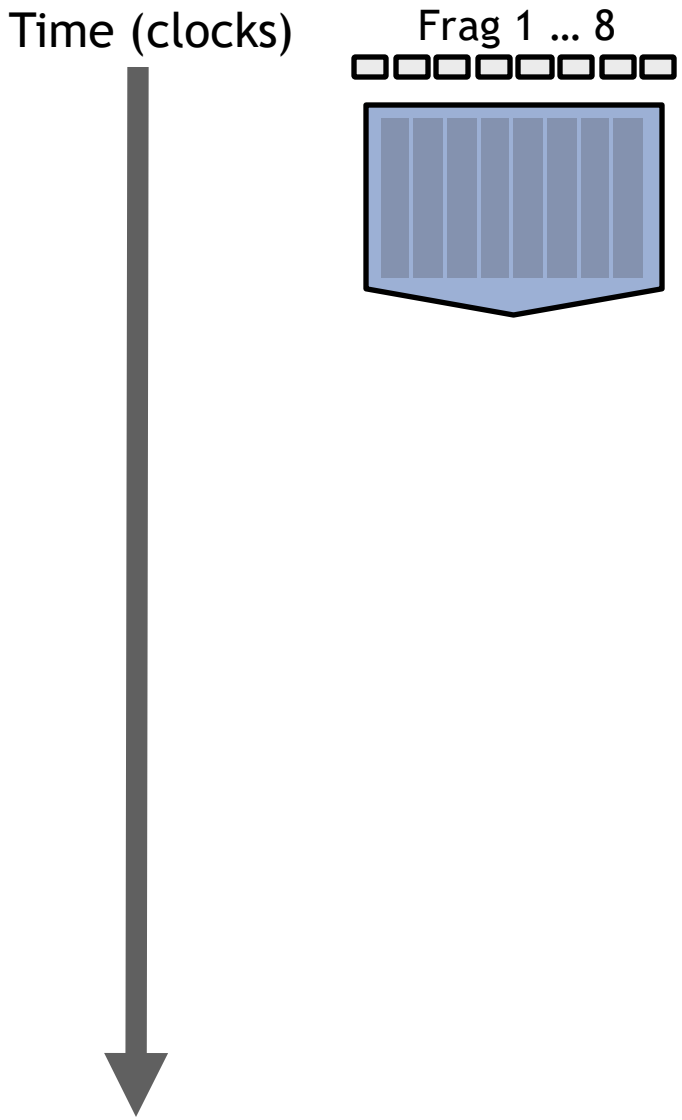
- Stalls occur when a core cannot run the next instruction because of a dependency on a previous operation.
- Texture access latency = 100s of cycles
- We've removed the fancy caches and logic that helps avoid stalls

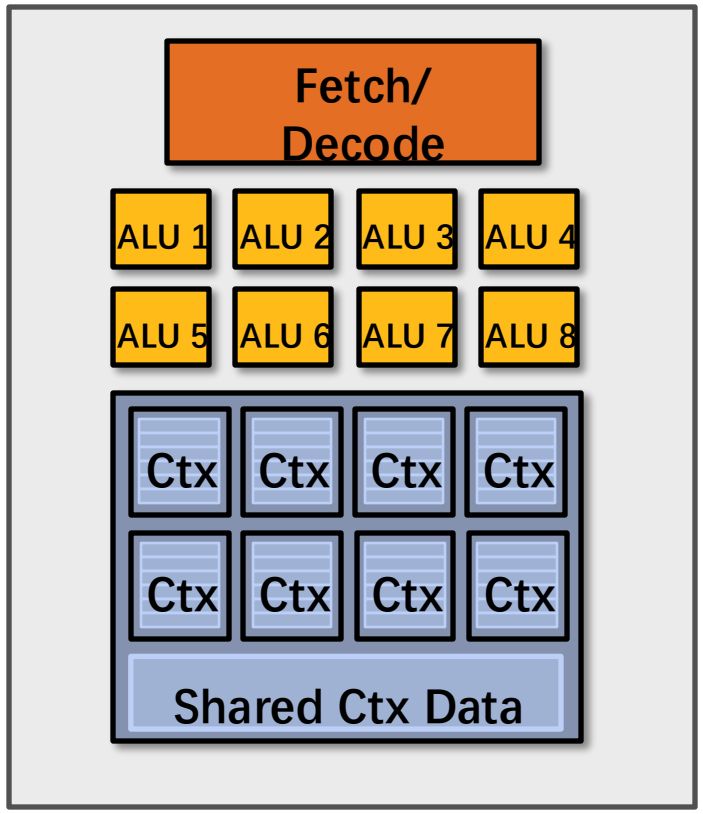
But we have LOTS of independent fragments!

Idea #3:

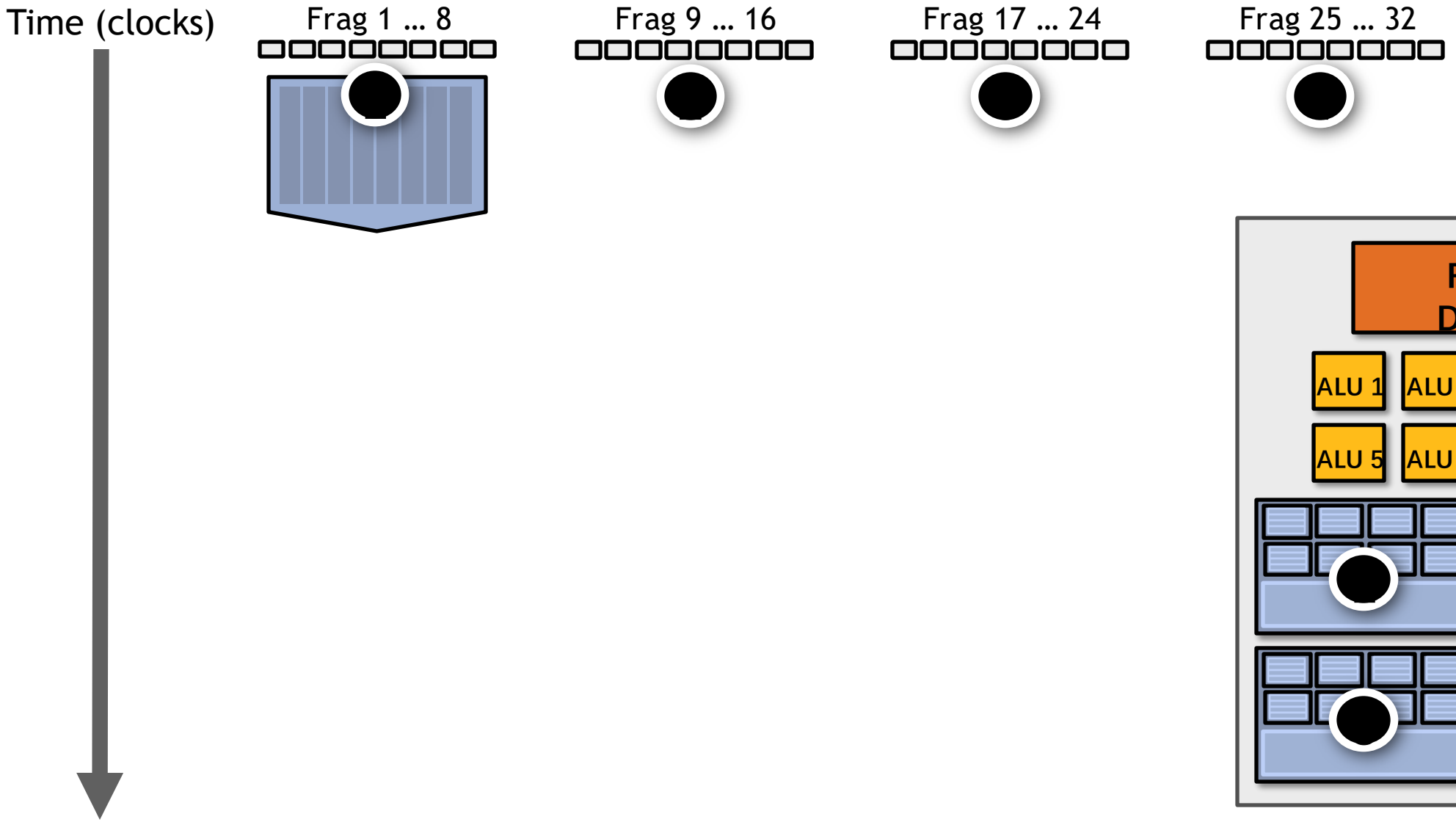
Interleave processing of many fragments on a single core to avoid stalls caused by high latency operations.

Hiding shader stalls

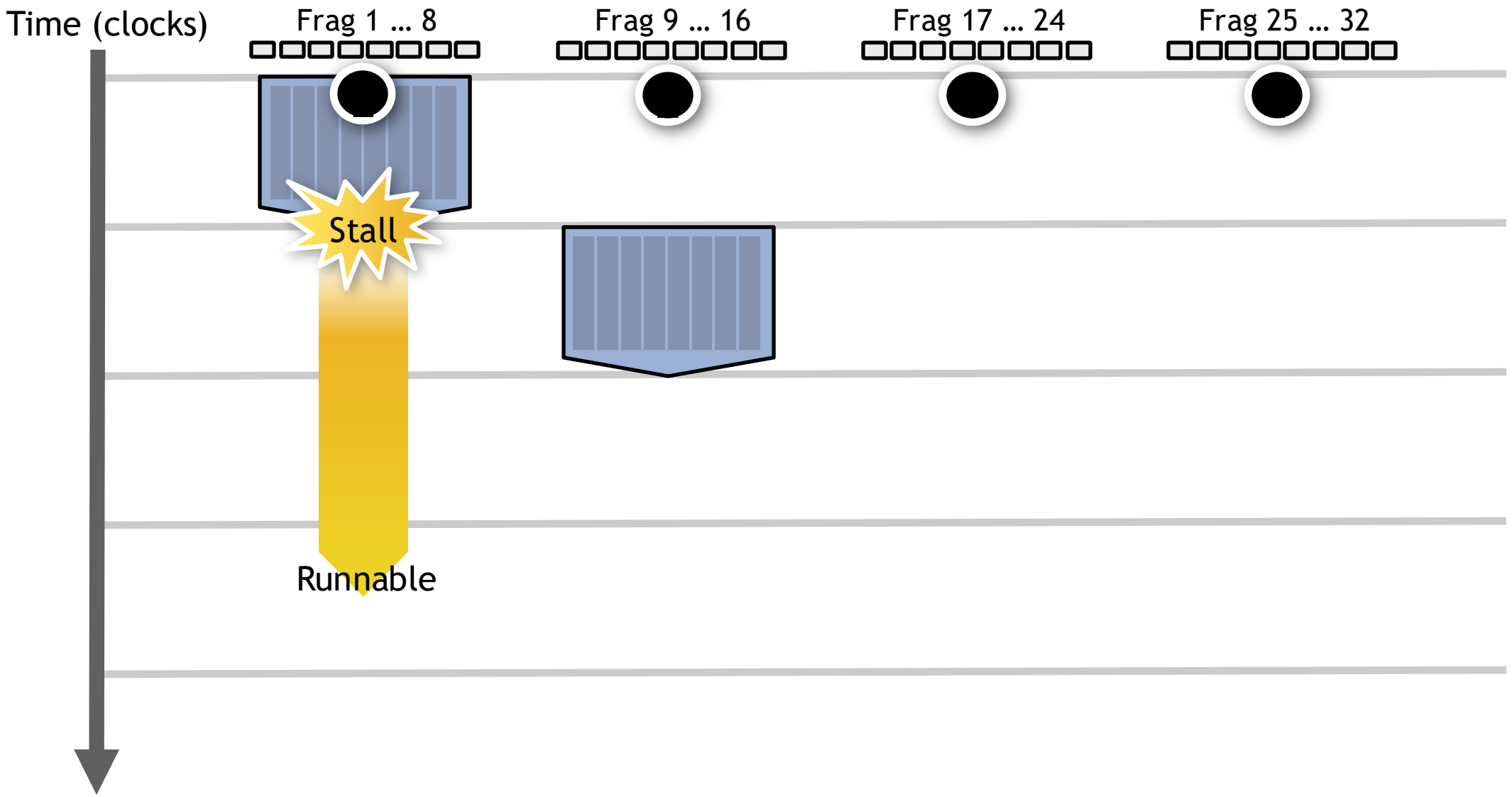




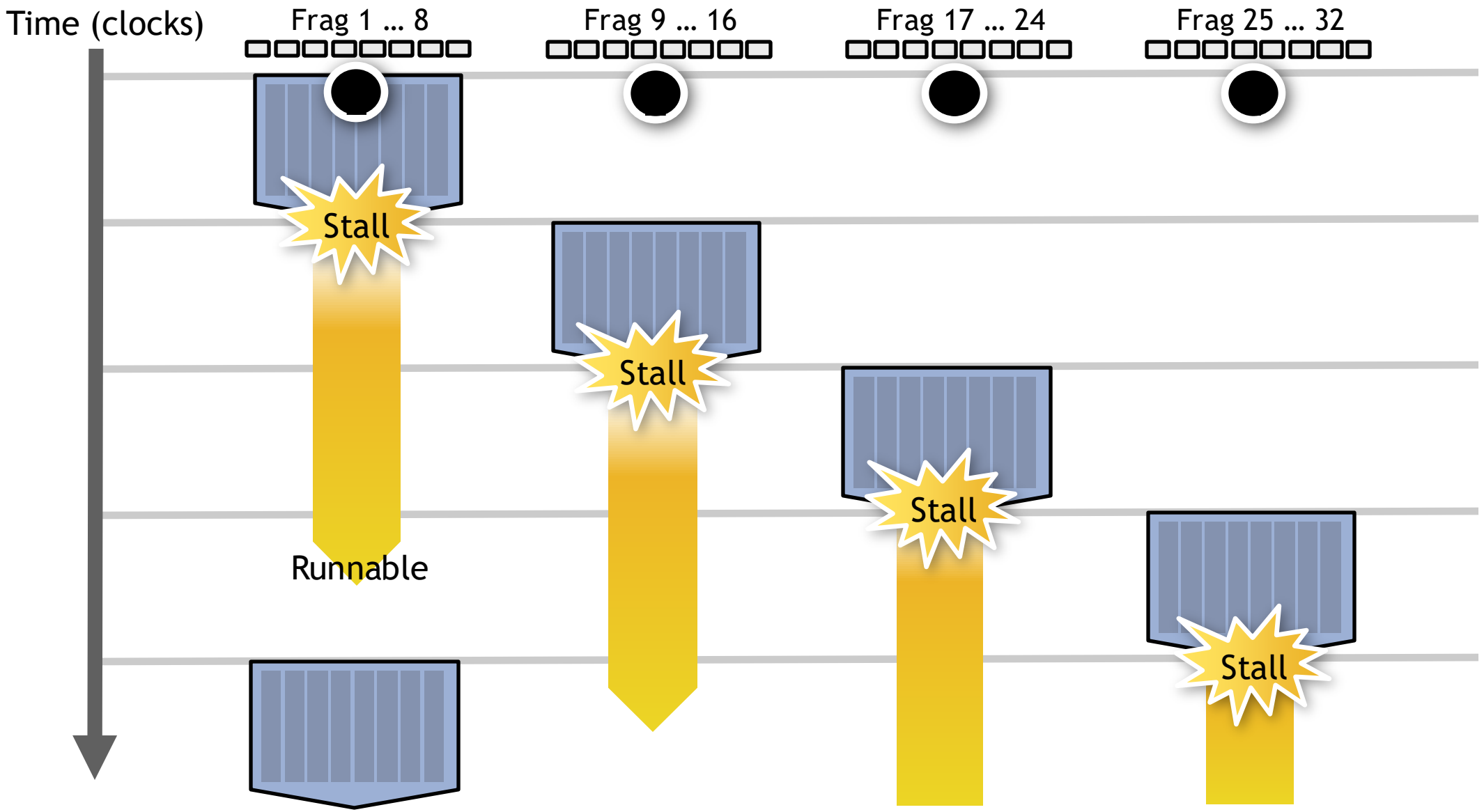
Hiding shader stalls



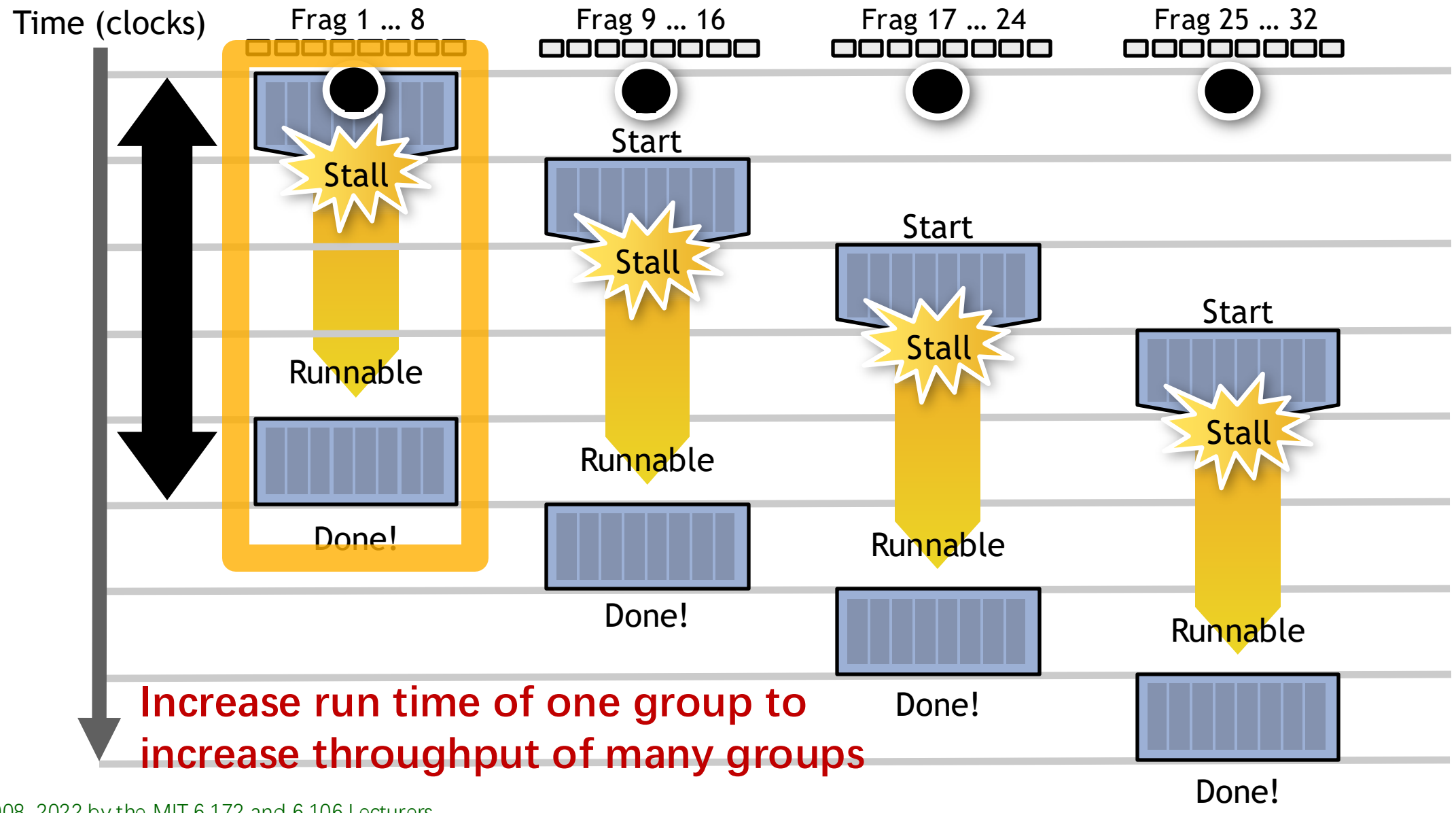
Hiding shader stalls



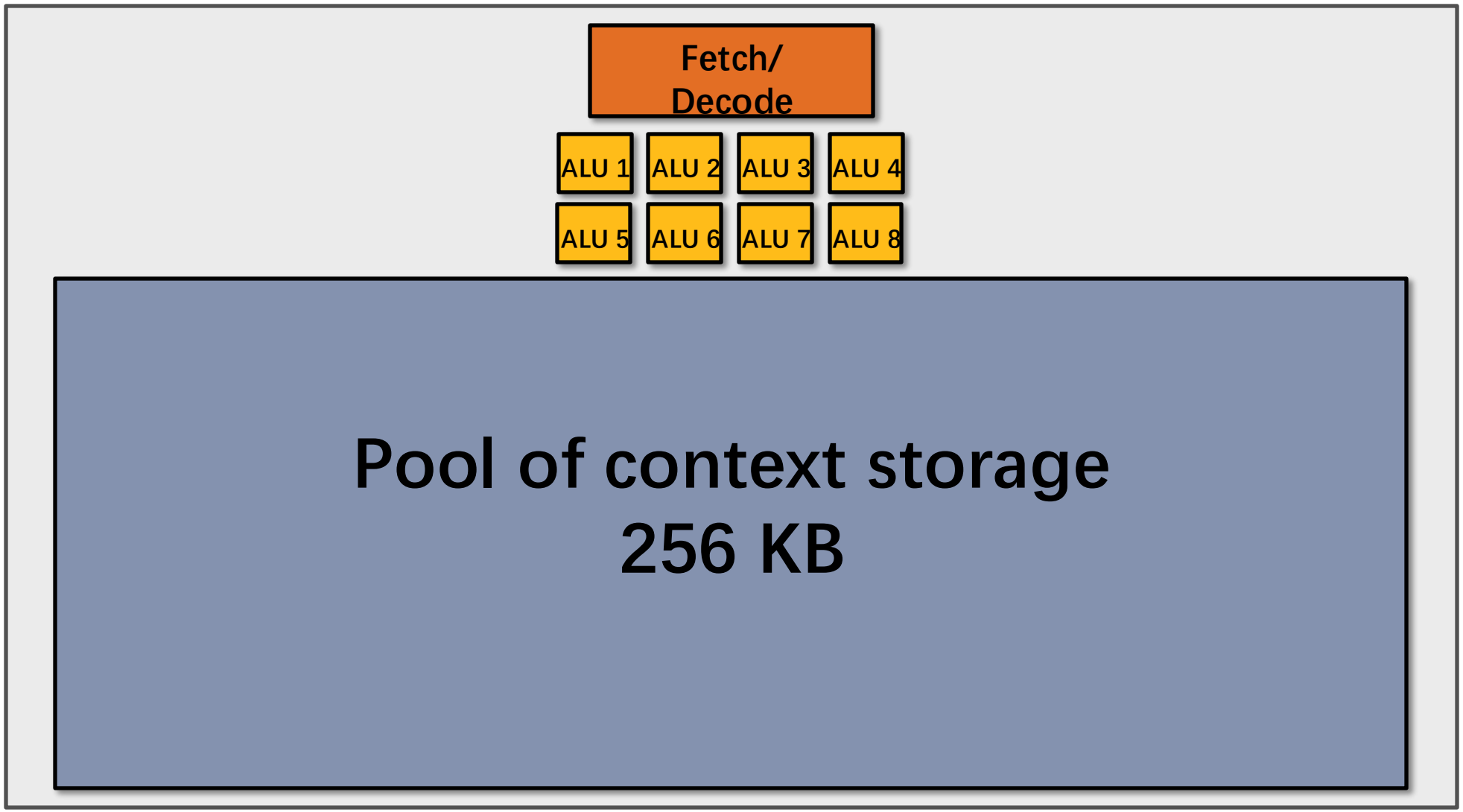
Hiding shader stalls



Throughput!

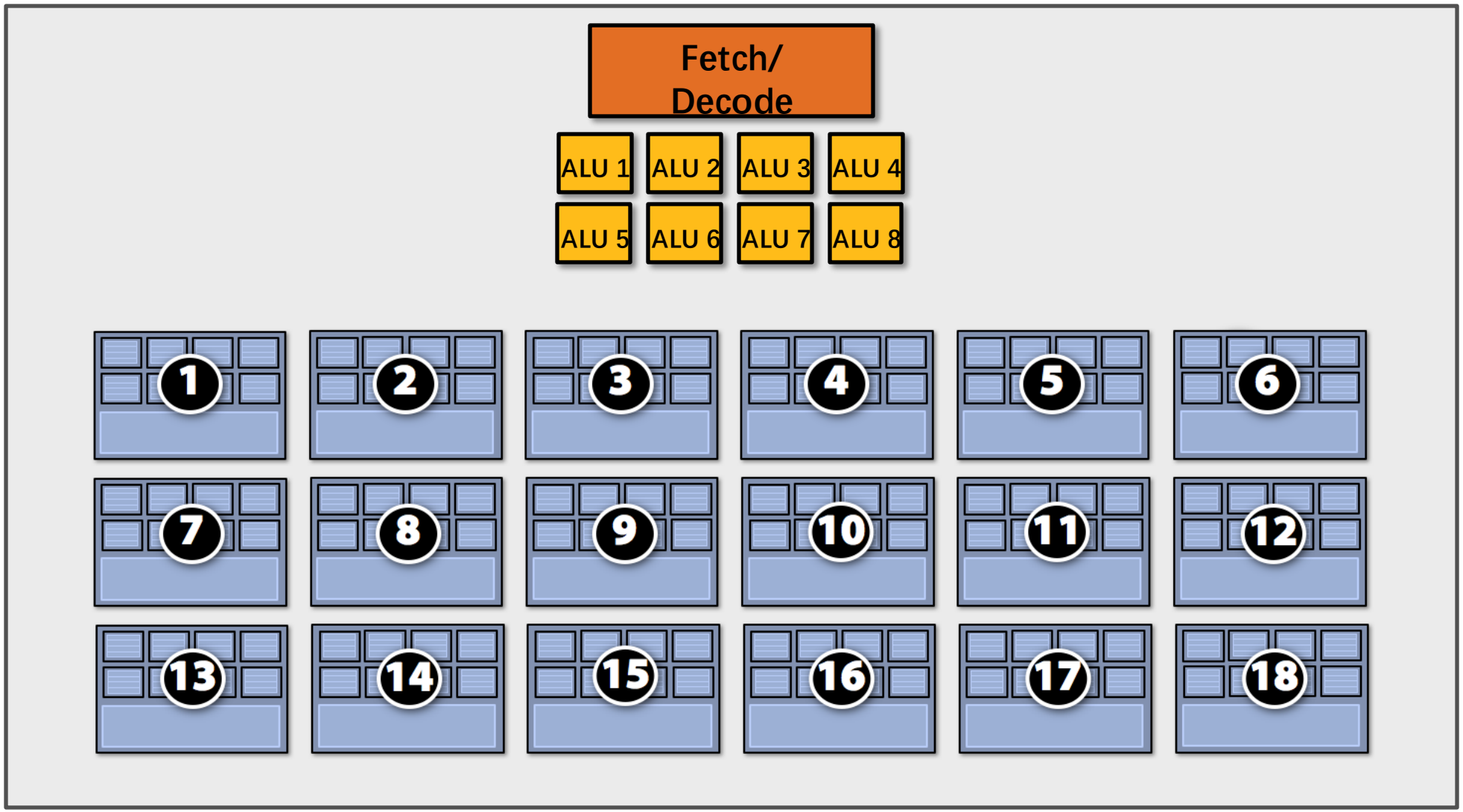


Storing contexts

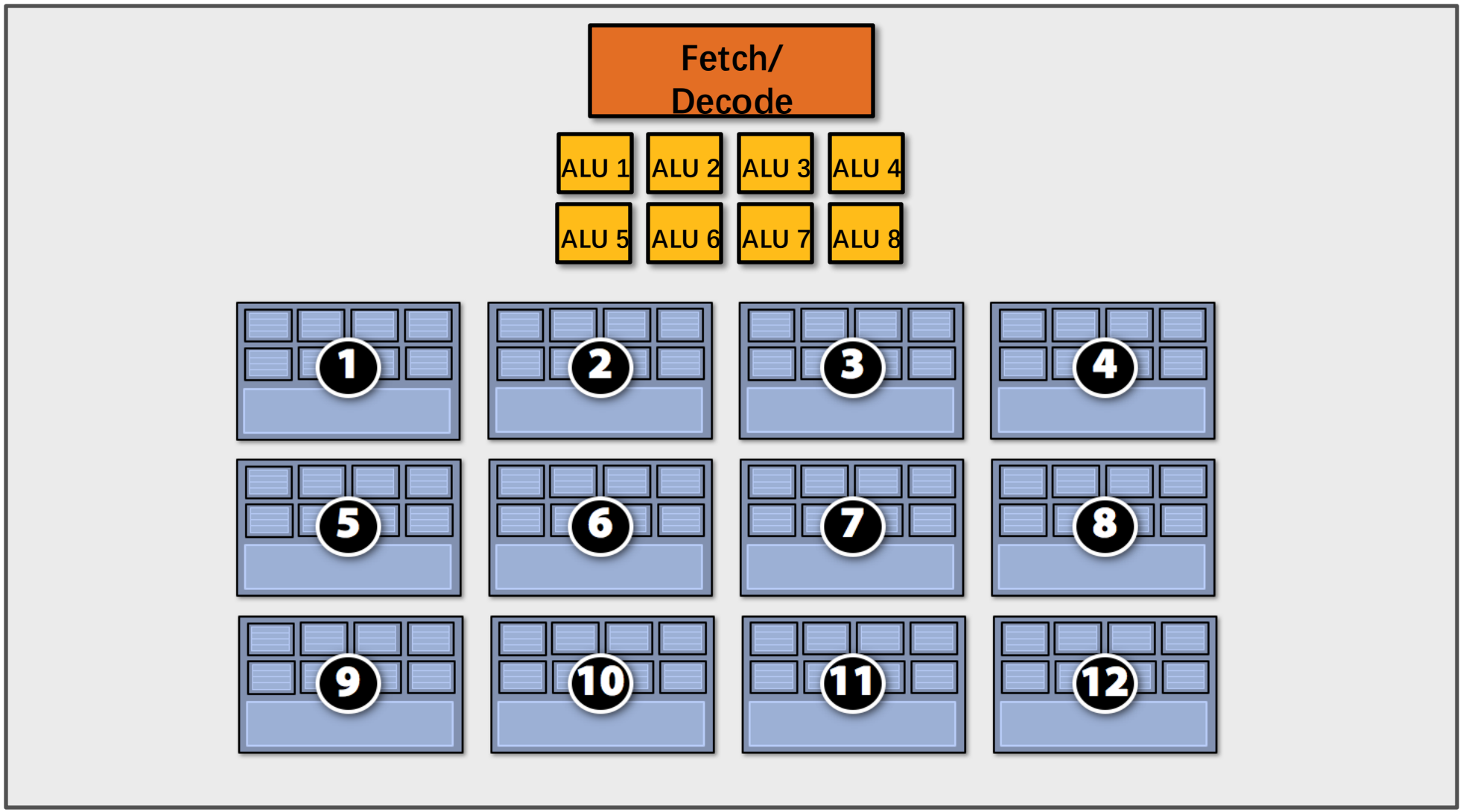


Eighteen small contexts

(maximal latency hiding)

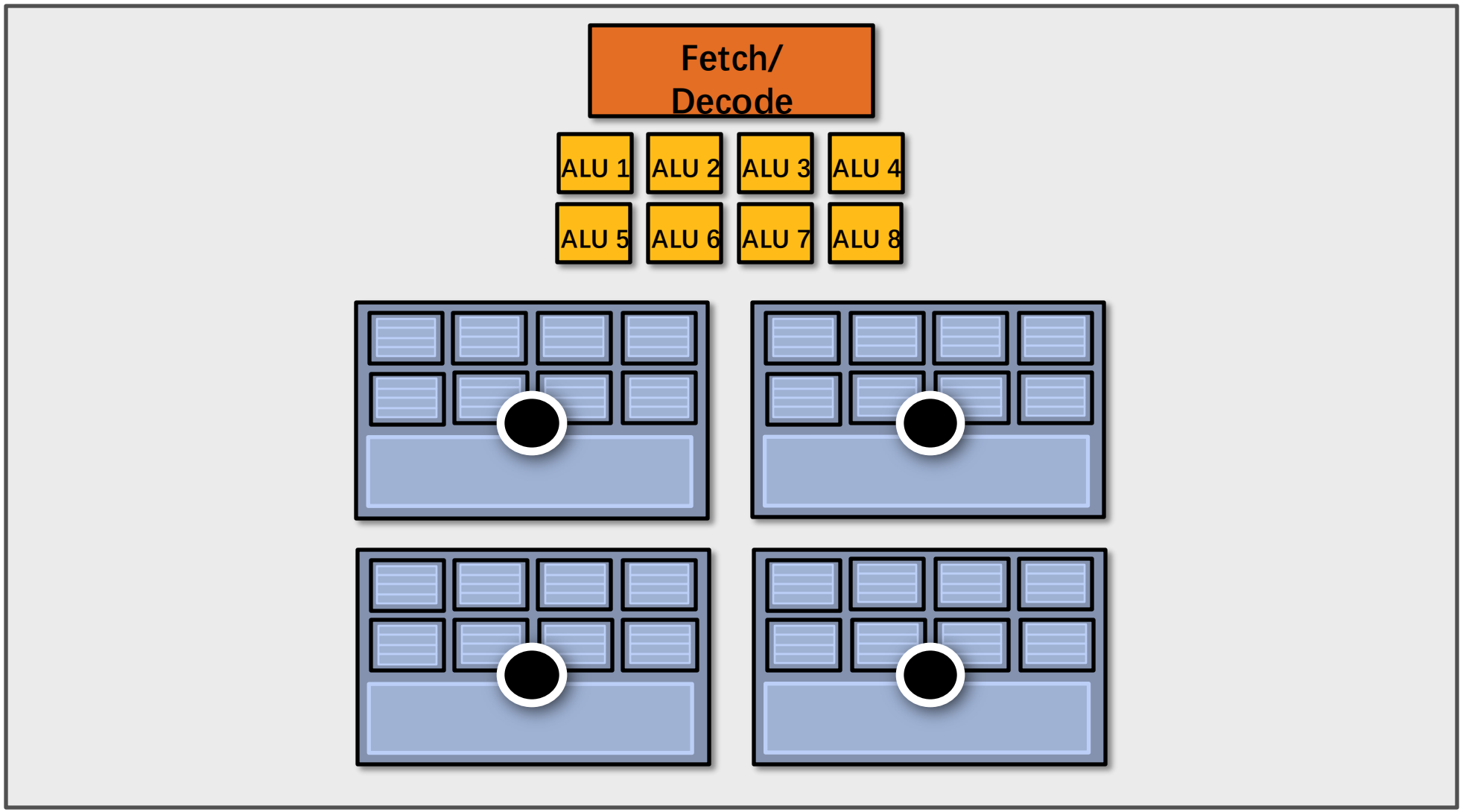


Twelve medium contexts



Four large contexts

(low latency hiding ability)



Clarification

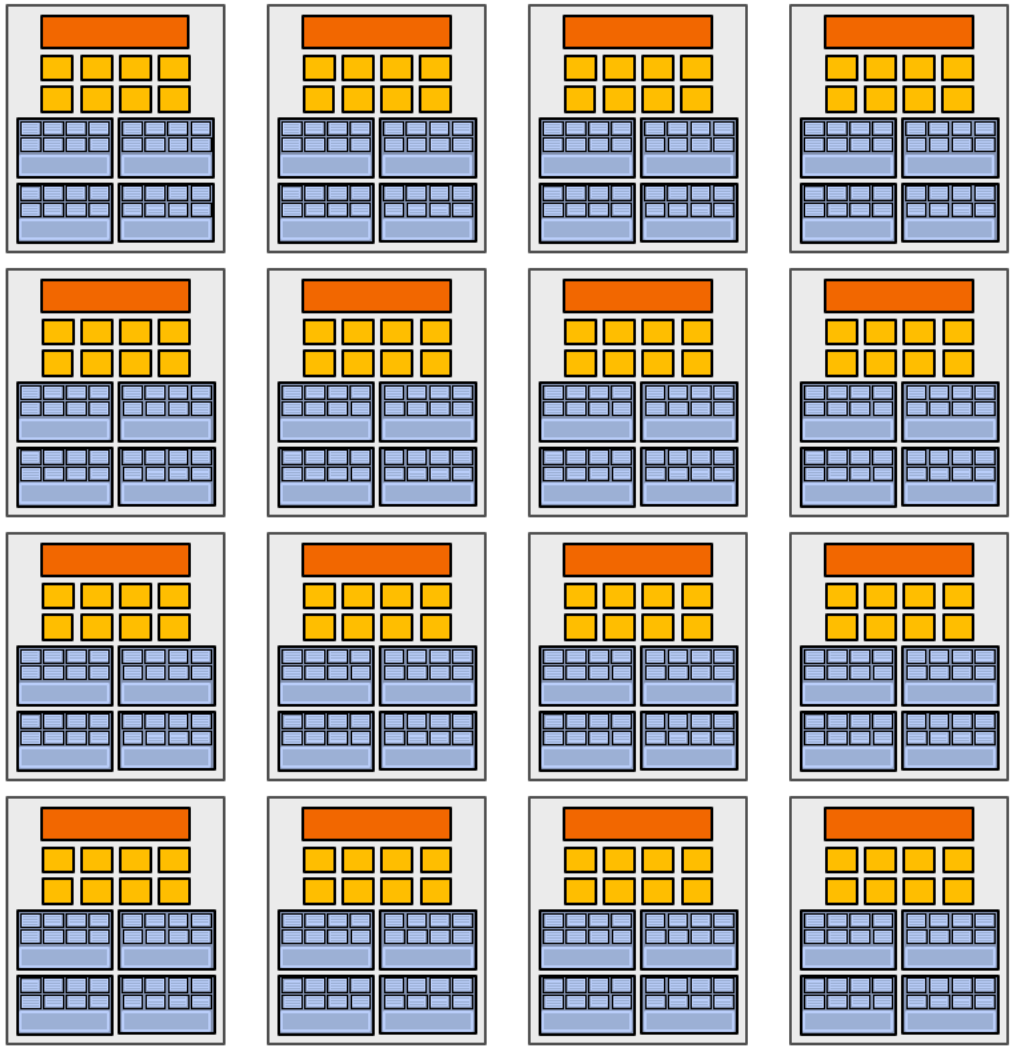
- Interleaving between contexts can be managed by hardware or software (or both!)

- 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 GPU!

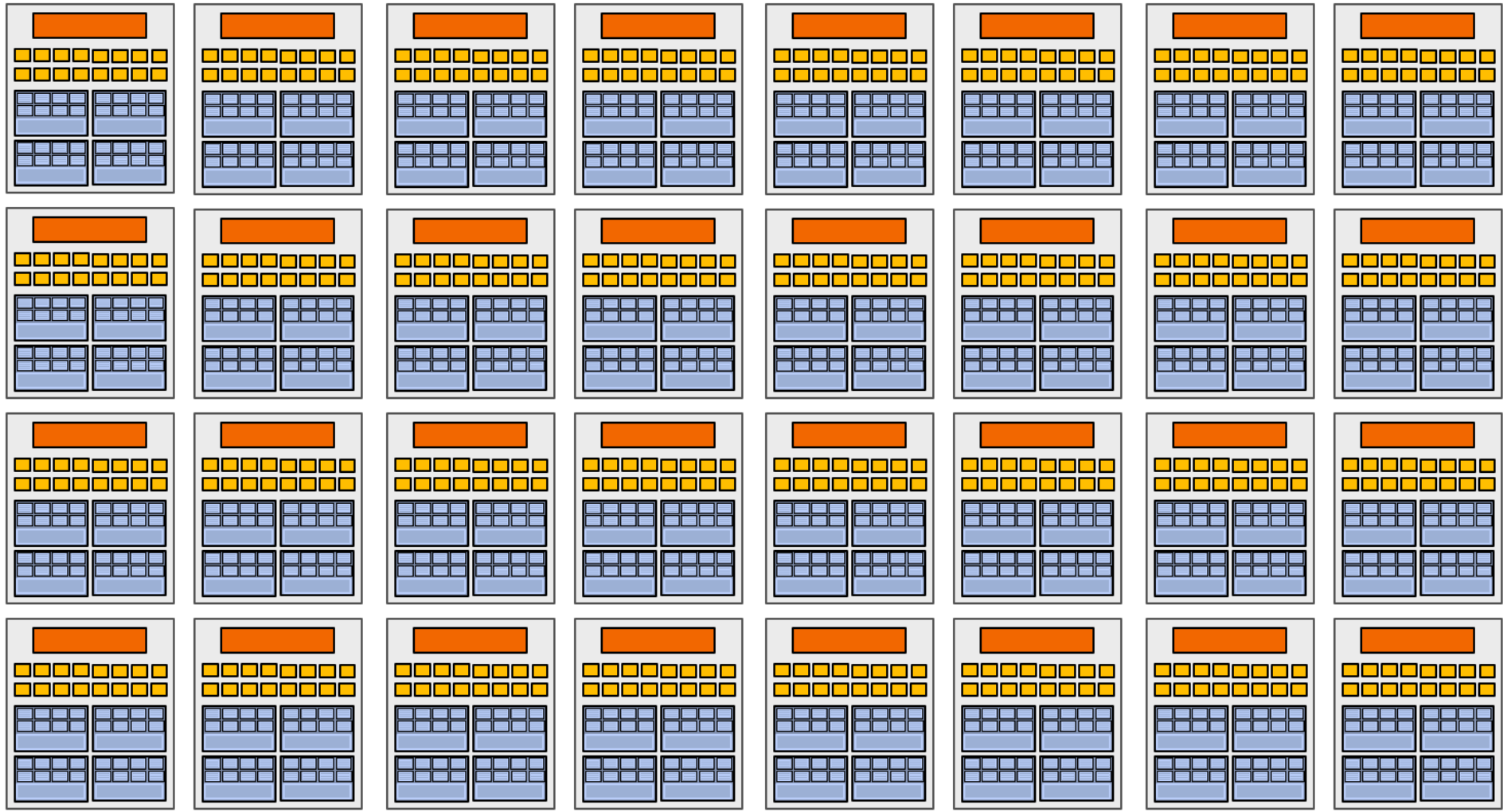
- 16 cores
- 16 simultaneous inst. streams
- 8 mul-add ALUs per core (128 total)
- 64 concurrent (but interleaved) instruction streams
- 512 concurrent fragments



8 ALUs × 2 FLOPs = 16 FLOPs / cycle / core

16 cores × 16 FLOPs
= 256 GFLOPs (@ 1GHz)

My bigger GPU!

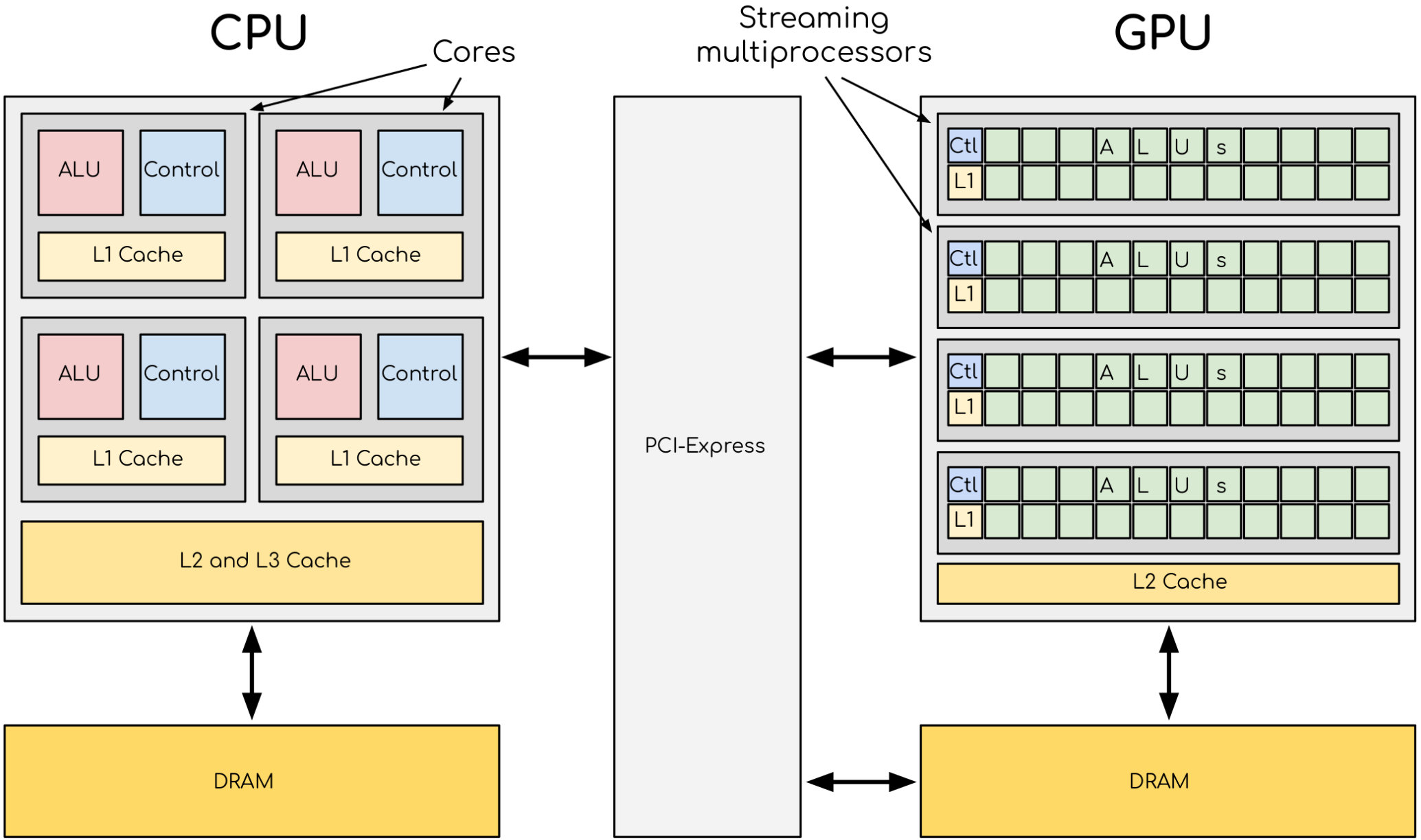


32 cores, 16 ALUs per core (512 total) = 1 TFLOP (@ 1 GHz)

Summary: three key ideas

- **Massive Parallelism**: use many "slimmed down cores" to run in parallel
- **SIMD→SIMT**: 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
- **Warp Interleaving**: avoid latency stalls by interleaving execution of many groups of fragments
  - When one group stalls, work on another group

A comparison of the CPU and GPU architecture



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

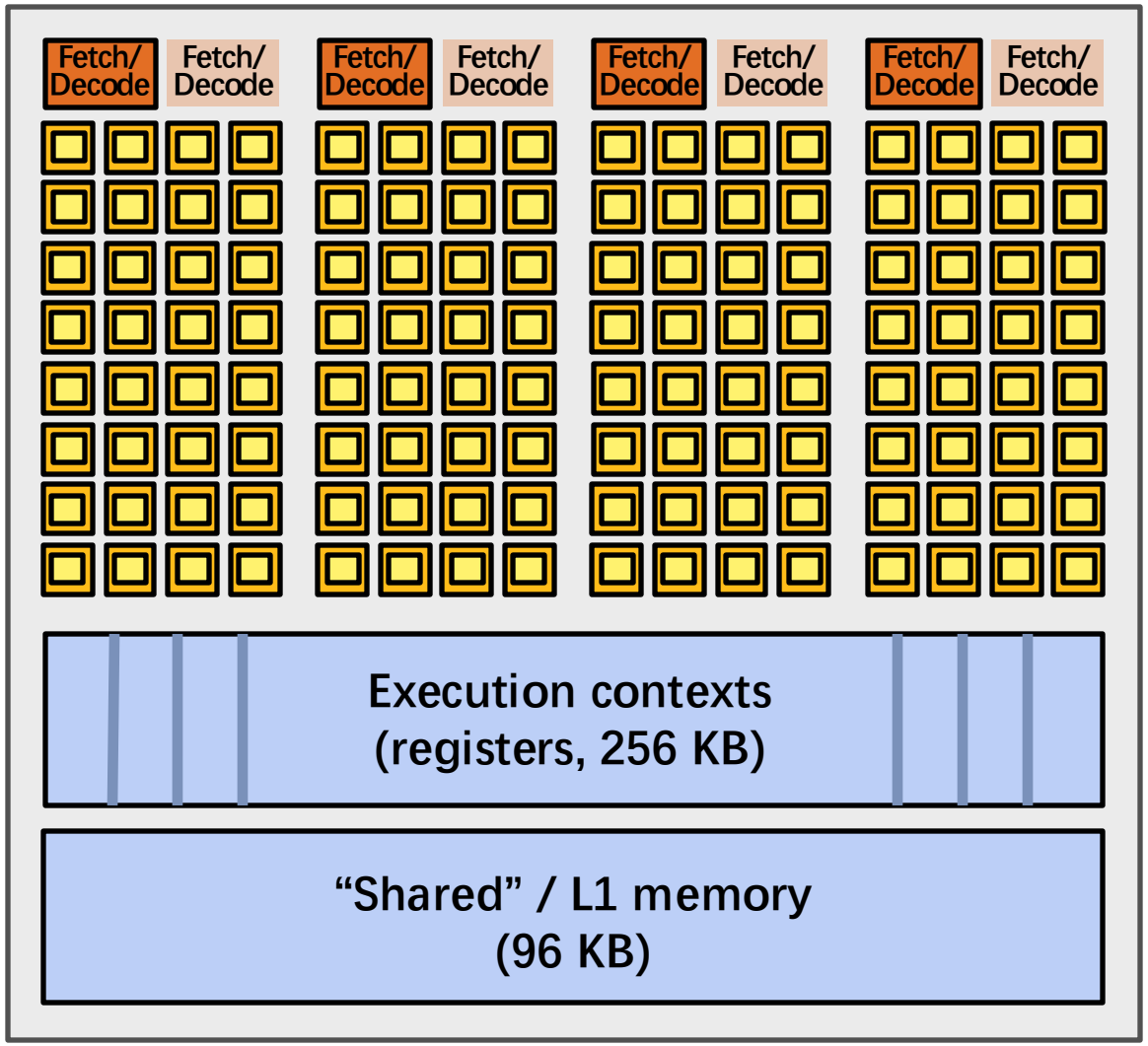
NVIDIA GeForce GTX 1080
NVIDIA GeForce RTX 2080 ~ 5080

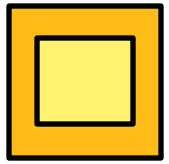
NVIDIA GeForce GTX 1080

- NVIDIA-speak:
  - 2560 stream processors (“CUDA cores”)
  - “SIMT execution”

- Generic speak:
  - 20 cores
  - 4 groups of 32 SIMD functional units per core

NVIDIA GeForce GTX 1080 “core”





= SIMD function unit,
control shared across 32 units
(1 MUL-ADD per clock)

- Groups of 32 [fragments/vertices/CUDA threads] share an instruction stream
- Up to 64 groups are simultaneously interleaved
- Up to 2,048 individual contexts can be stored

Source: NVIDIA Pascal tuning guide

NVIDIA GeForce GTX 1080

There are 20 of these things on the GTX 1080

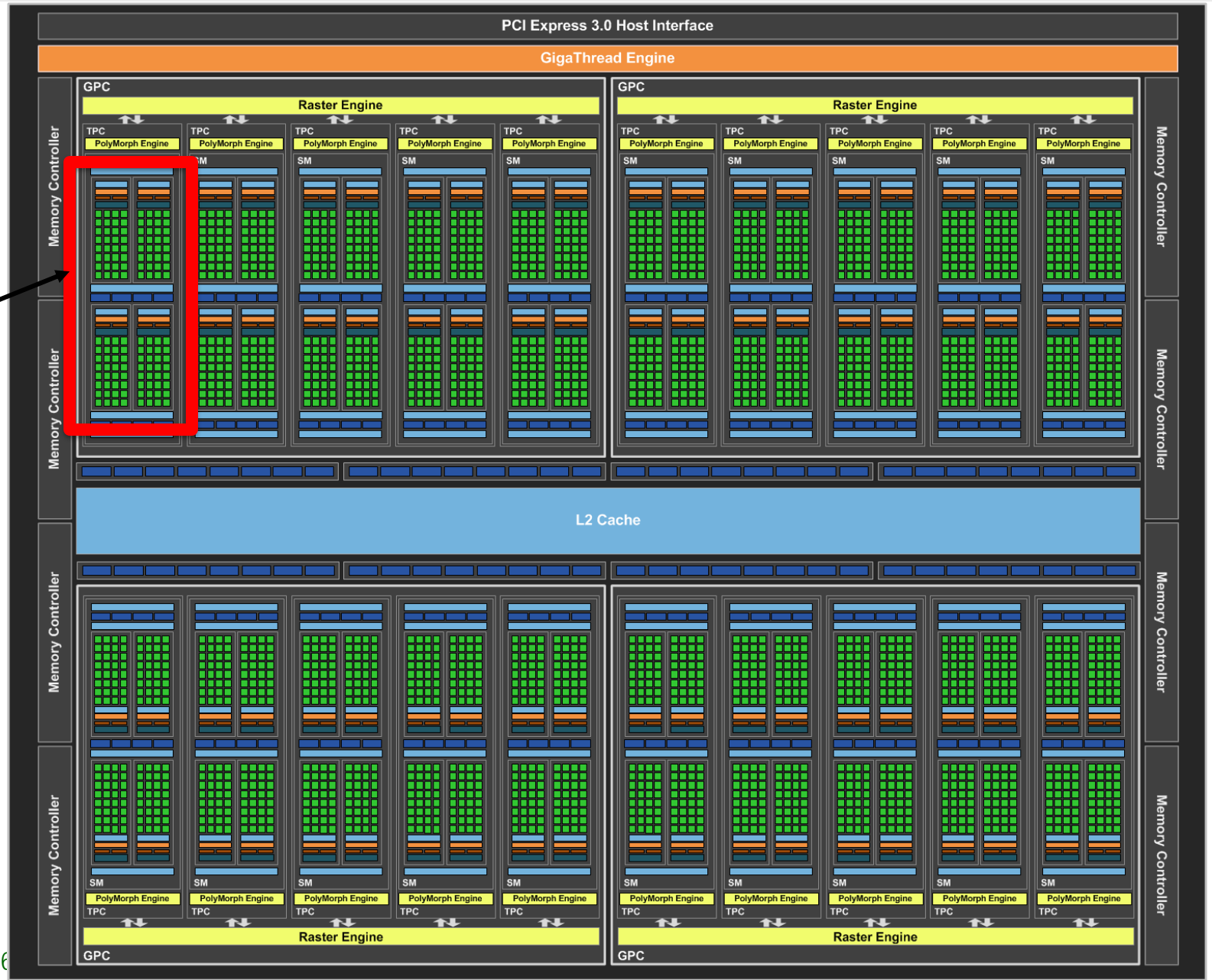
That's 40,960 fragments!

(Or 40,960 "CUDA threads")

Block Diagram of the GP104 GPU

- Pascal

Simultaneous Multiprocessor



NVIDIA GeForce GTX 1080 “core”

- Pascal

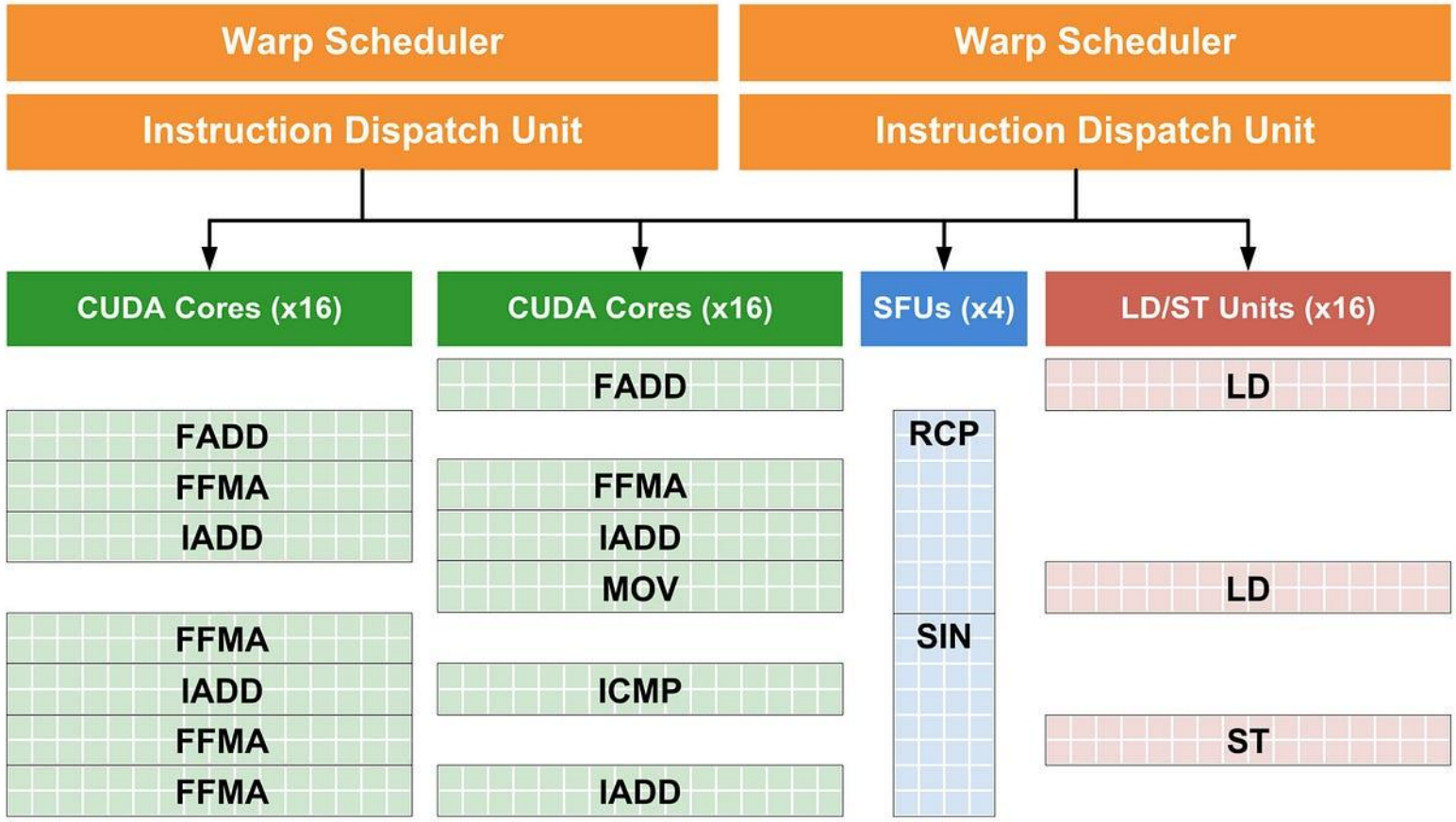
32 CUDA cores

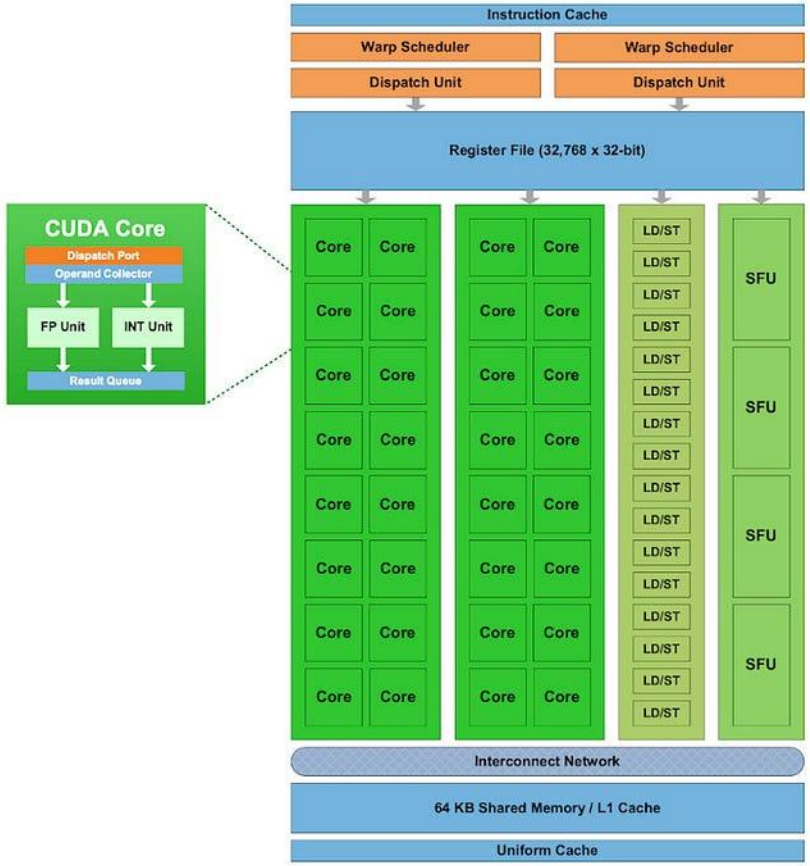
LD/ST Units
Special Function Units

Warp schedulers
Dispatch units
Registers

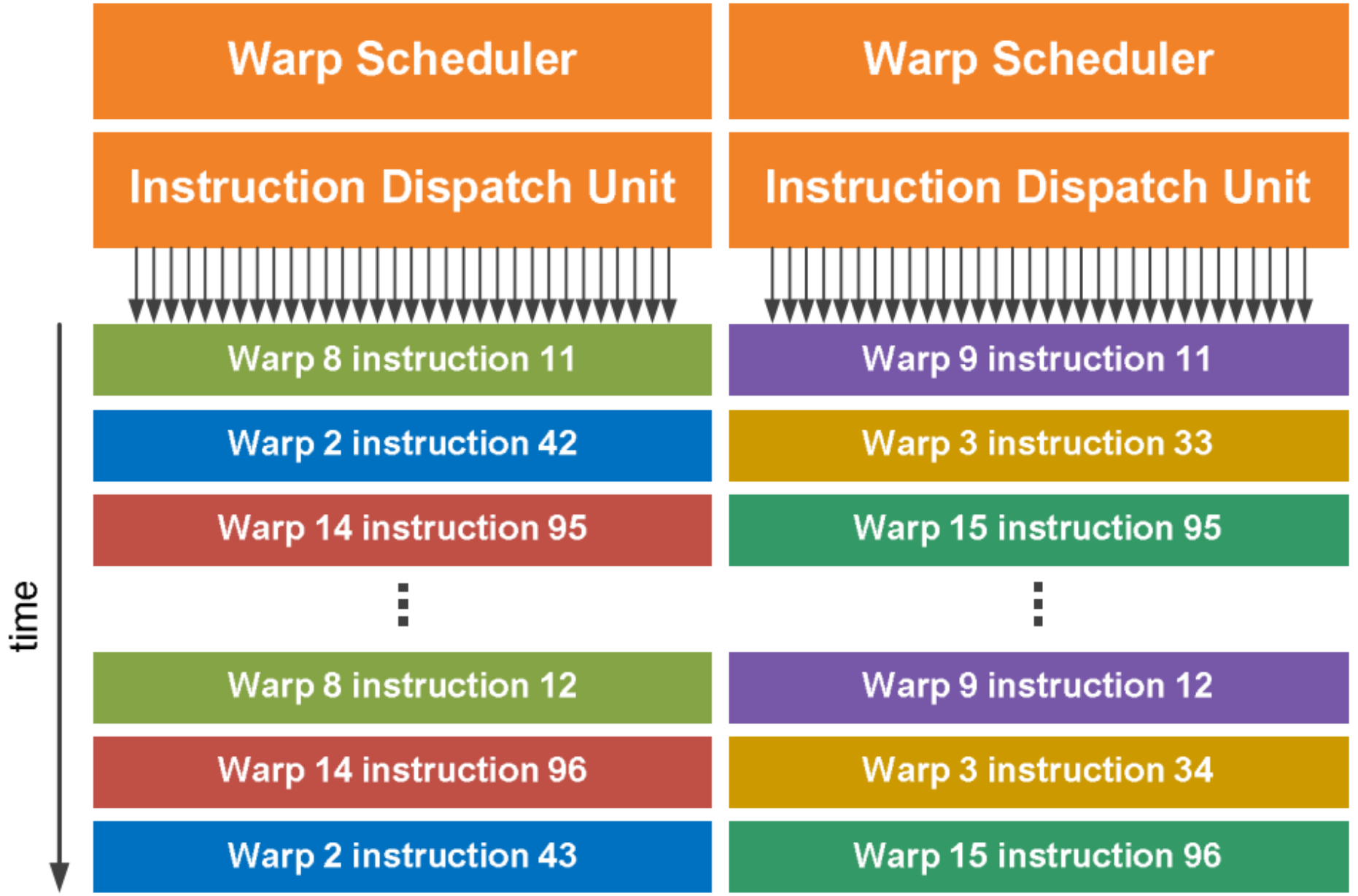
L1 Caches
Scratchpad

Warp Scheduler





Warp Scheduler

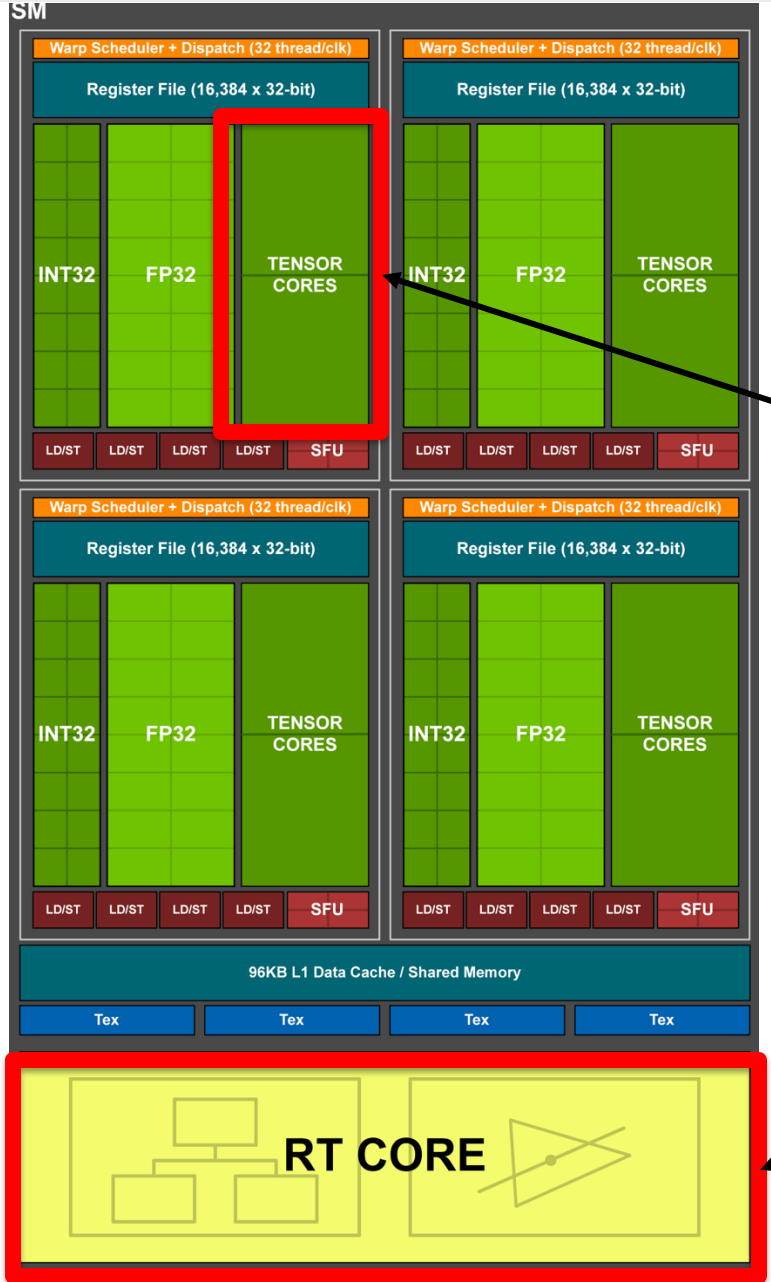


NVIDIA GeForce RTX 2080 “core”

- Turing

https://developer.nvidia.com/blog/nvidia-turing-architecture-in-depth/

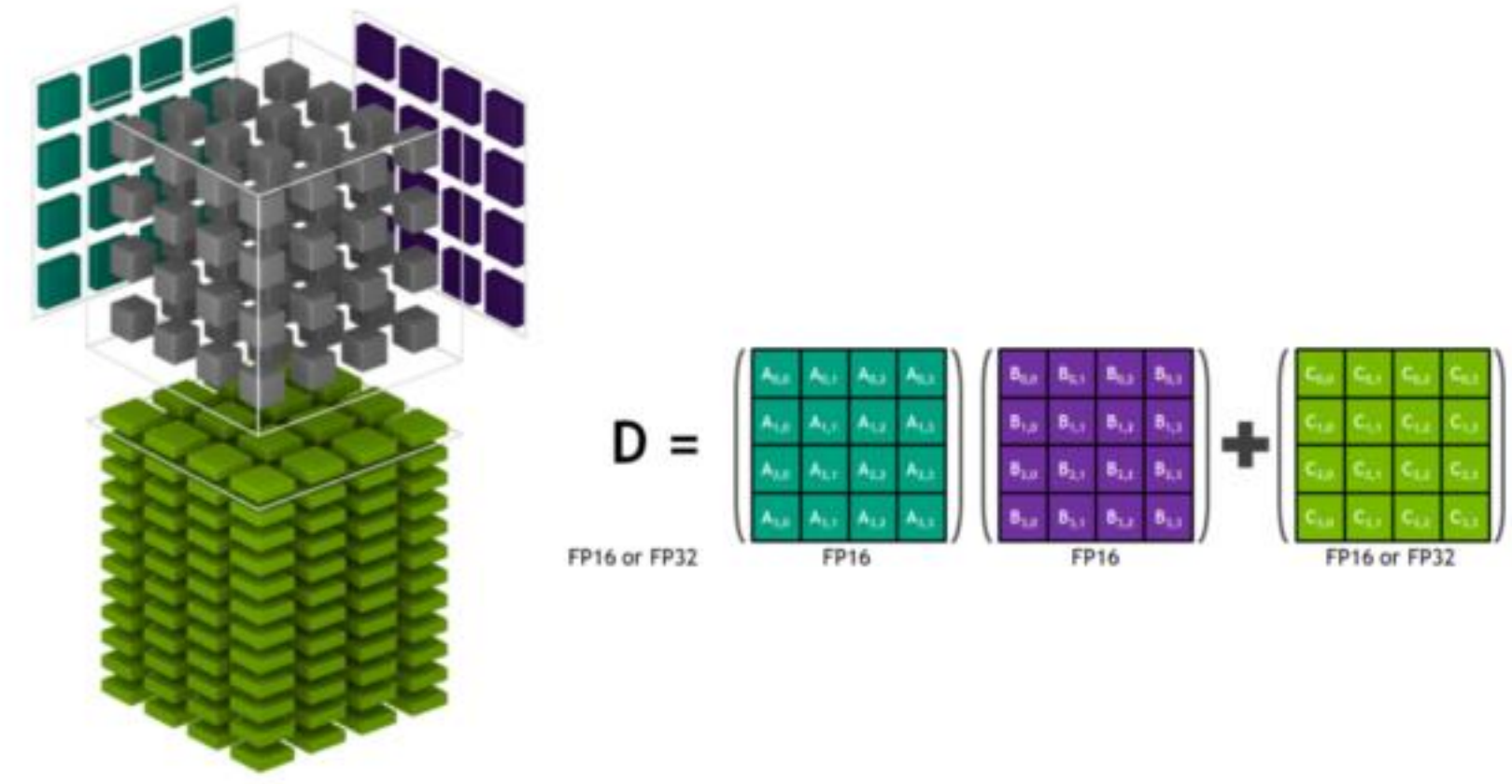
NVIDIA Tesla V100 GPU Architecture Whitepaper



Tensor Core?

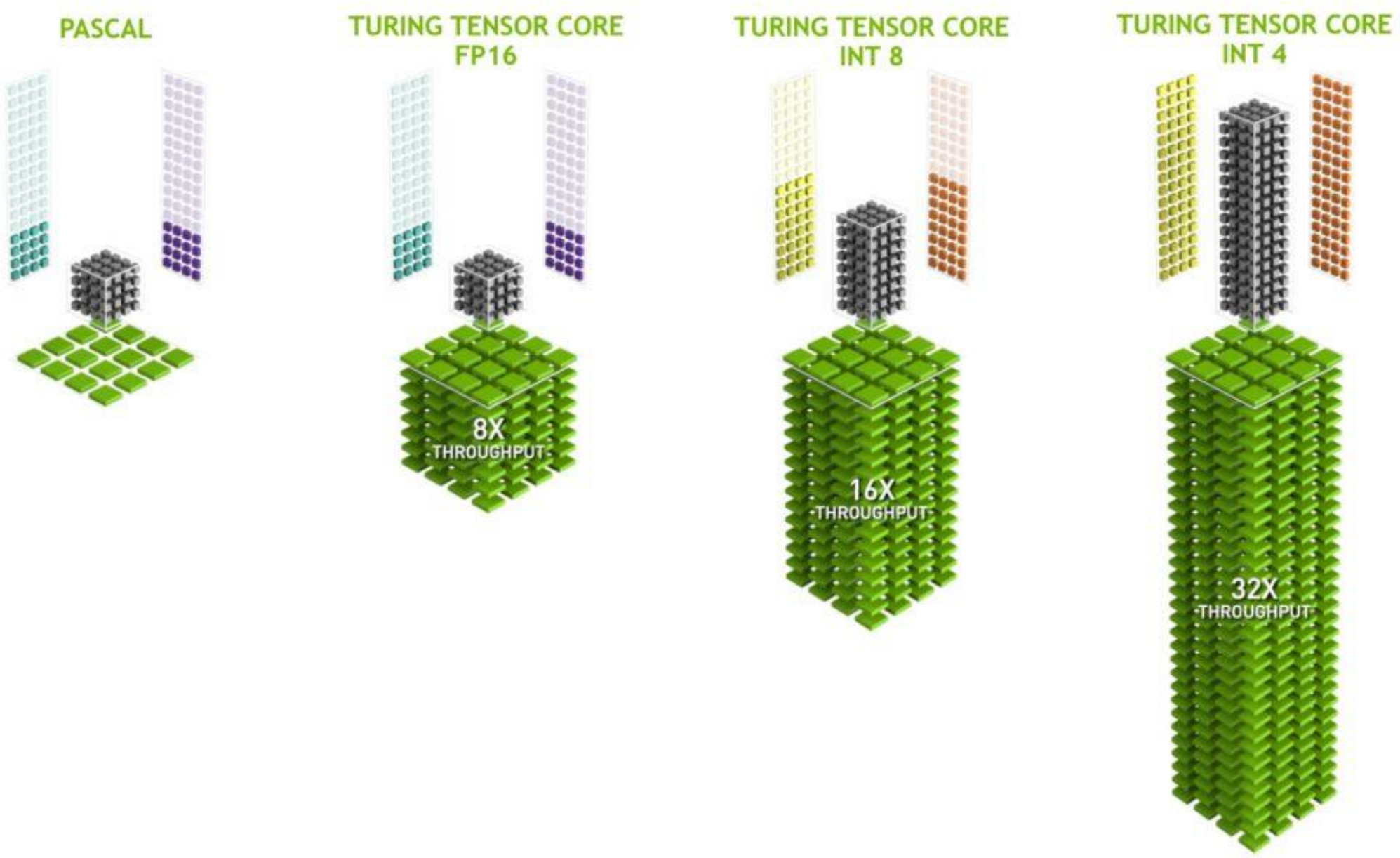
RT Core?

Tensor Cores

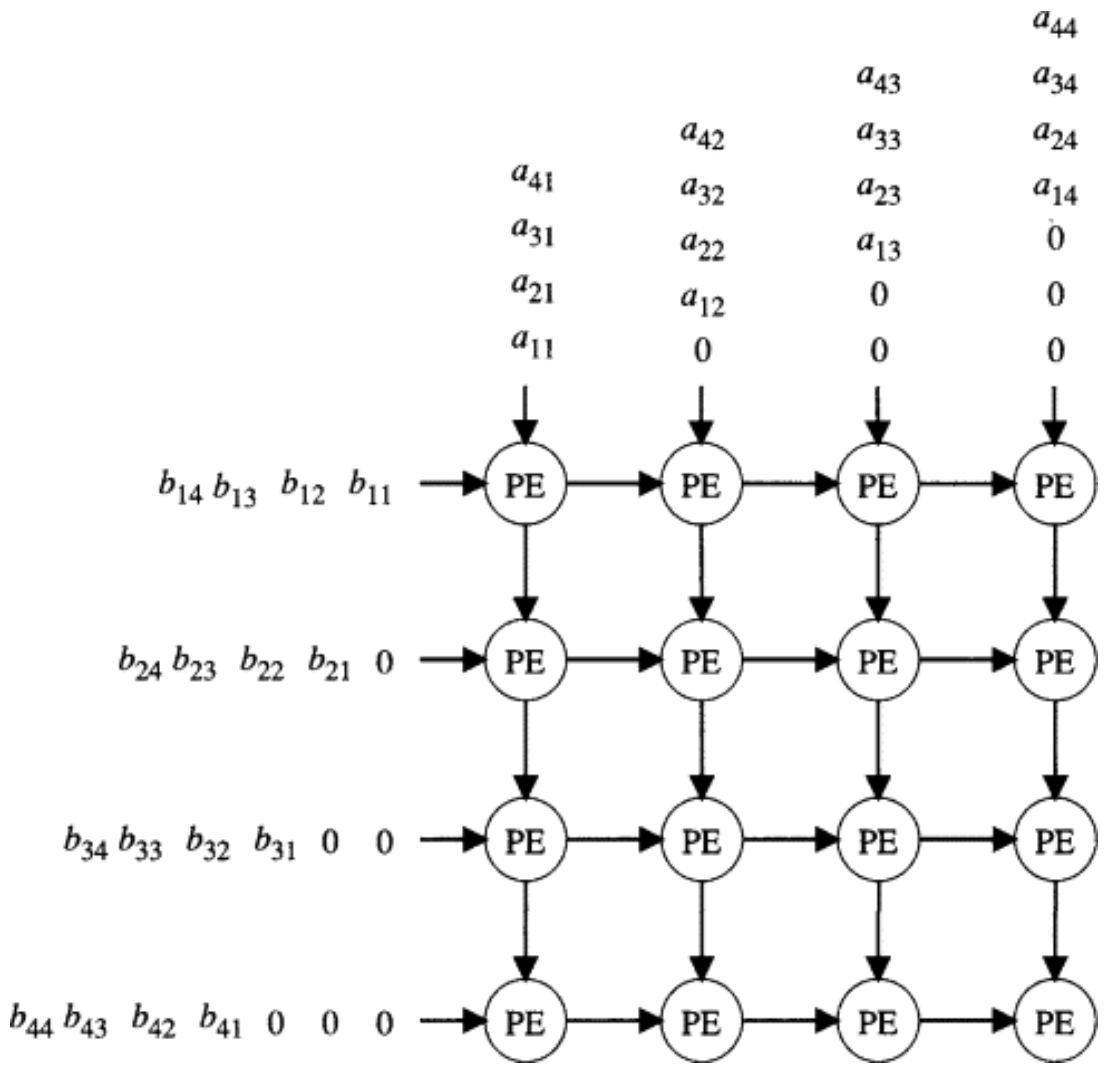


https://developer.nvidia.com/blog/tensor-core-ai-performance-milestones/

Tensor Cores

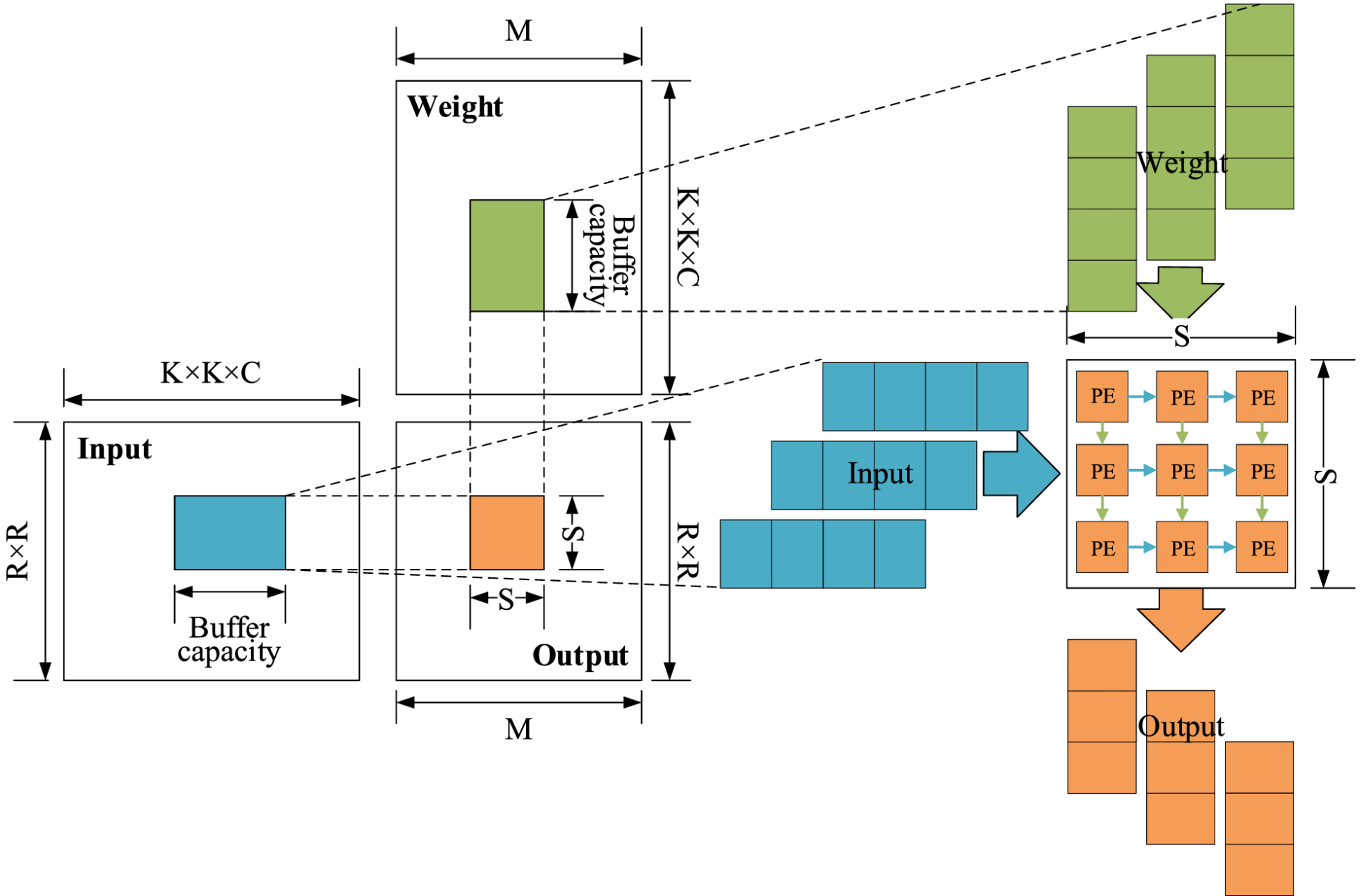


HW Structure: Systolic Arrays

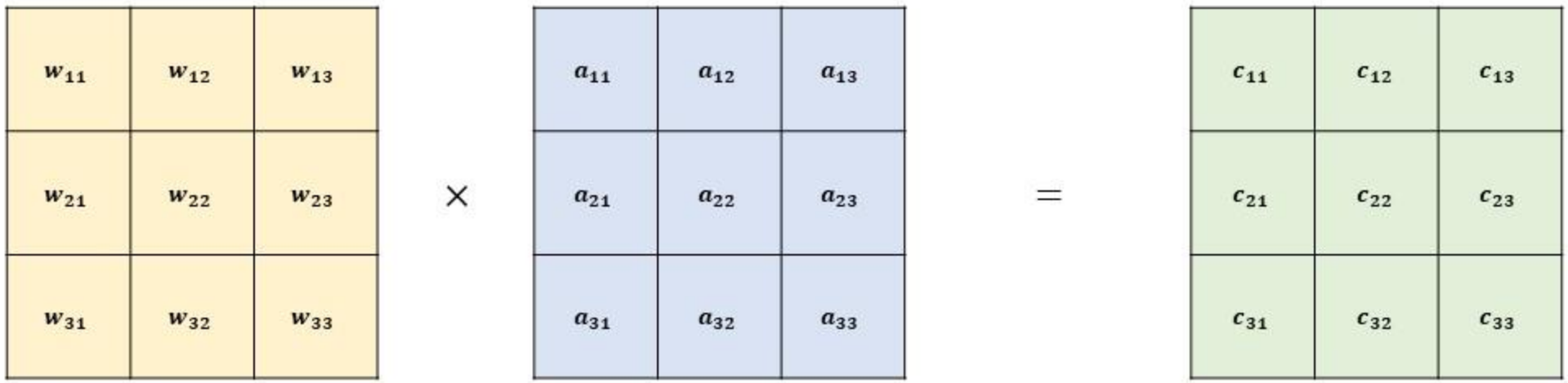


[1] HT Kung, CE Leiserson, Systolic arrays (for VLSI), Sparse Matrix Proceedings 1978 1, 256-282

MM on Systolic Arrays



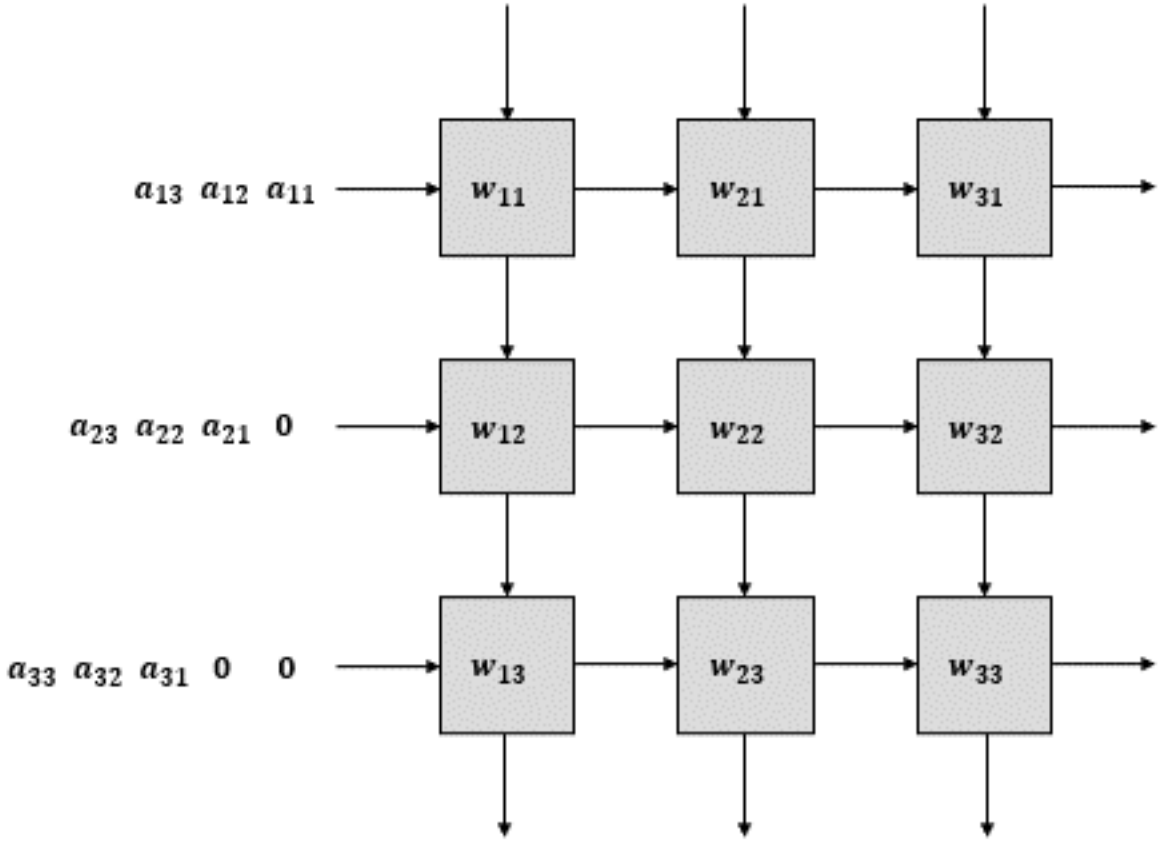
Matrix Multiplication
A matrix: 3x3
W matrix: 3x3
C = W x A



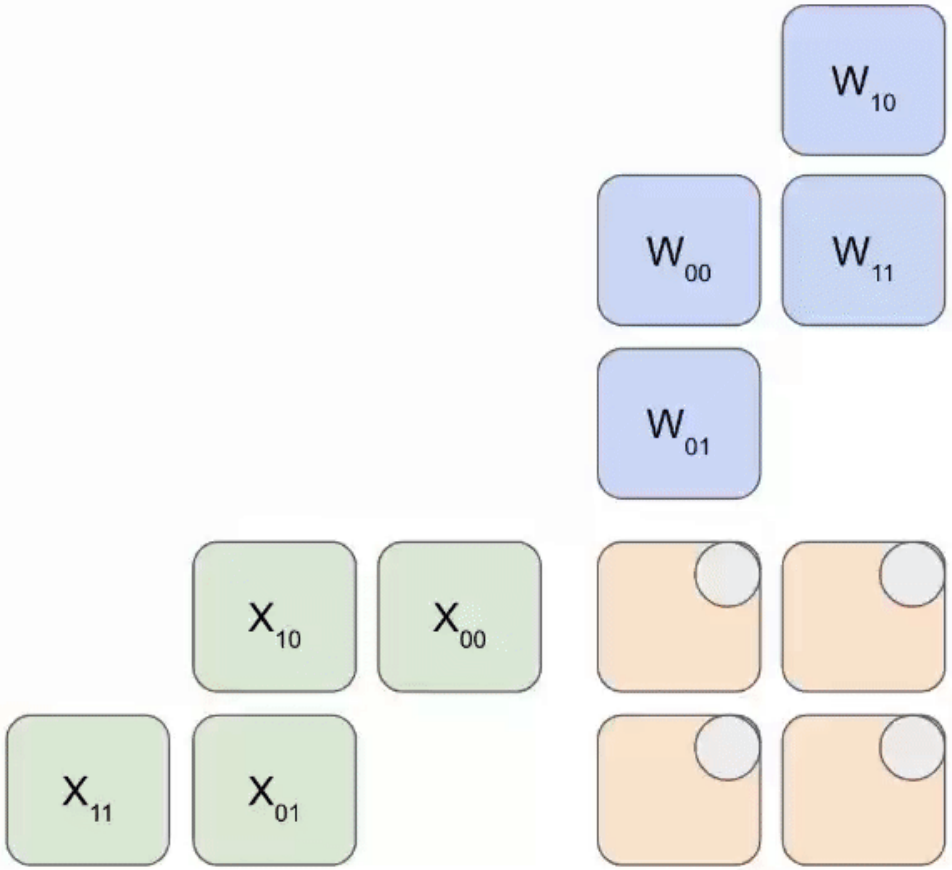
$$c_{11} = w_{11} \times a_{11} + w_{12} \times a_{21} + w_{13} \times a_{31}$$
$$c_{12} = w_{11} \times a_{12} + w_{12} \times a_{22} + w_{13} \times a_{32}$$
$$c_{13} = w_{11} \times a_{13} + w_{12} \times a_{23} + w_{13} \times a_{33}$$
$$\vdots$$
$$c_{33} = w_{31} \times a_{13} + w_{32} \times a_{23} + w_{33} \times a_{33}$$

Compute

a_{11}	a_{12}	a_{13}
a_{21}	a_{22}	a_{23}
a_{31}	a_{32}	a_{33}

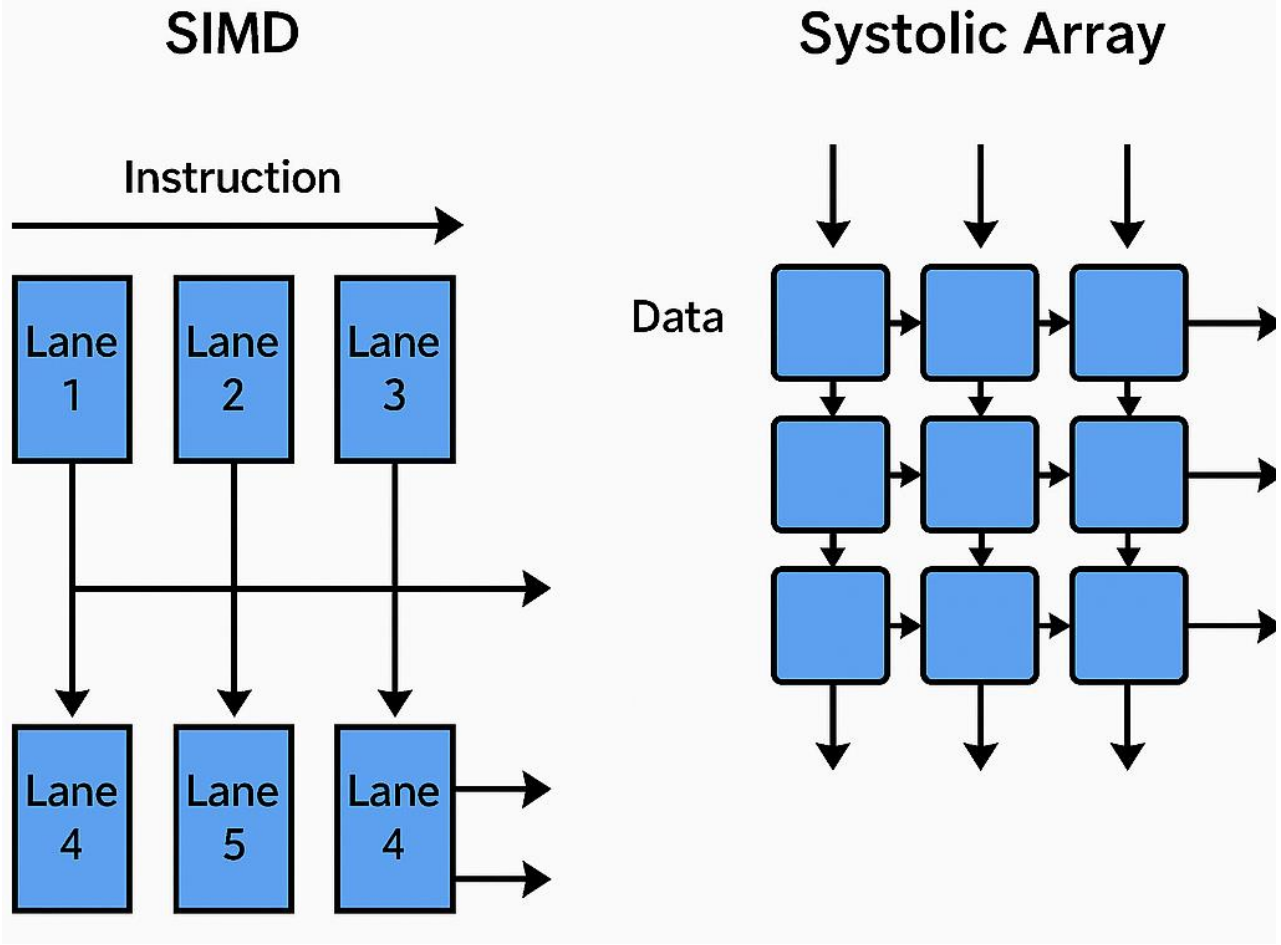


https://deep-math.tistory.com/29



SIMD vs. Systolic Array

- Trade off general-purpose flexibility for specialization



SIMD vs. Systolic Array

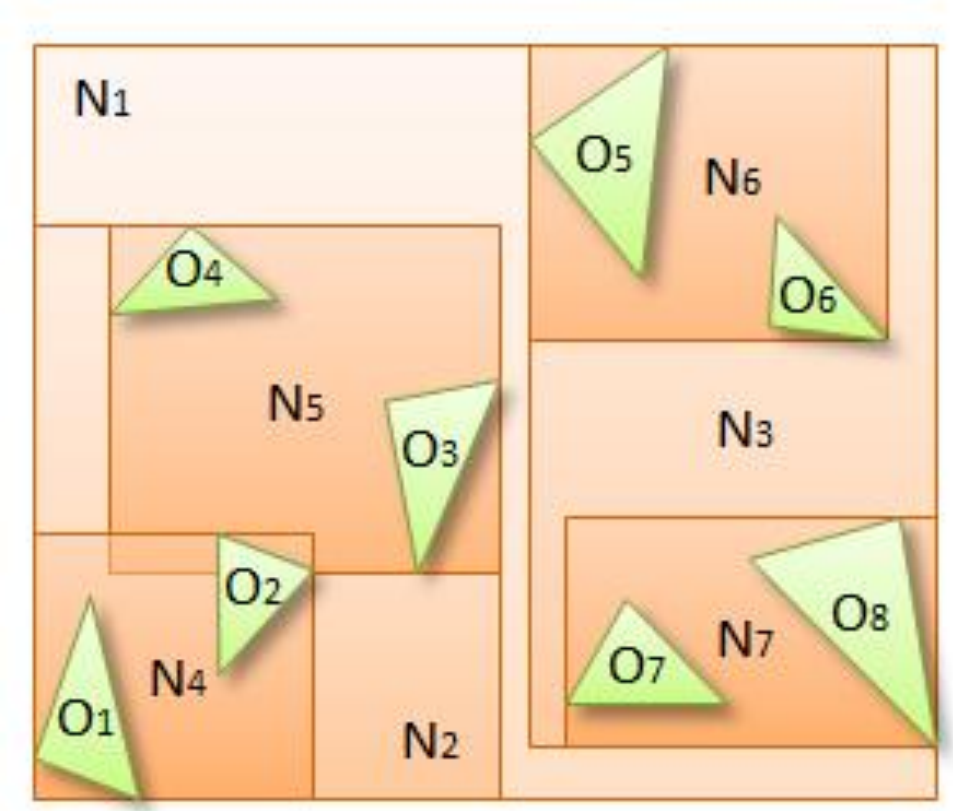
- Trade off general-purpose flexibility for specialization

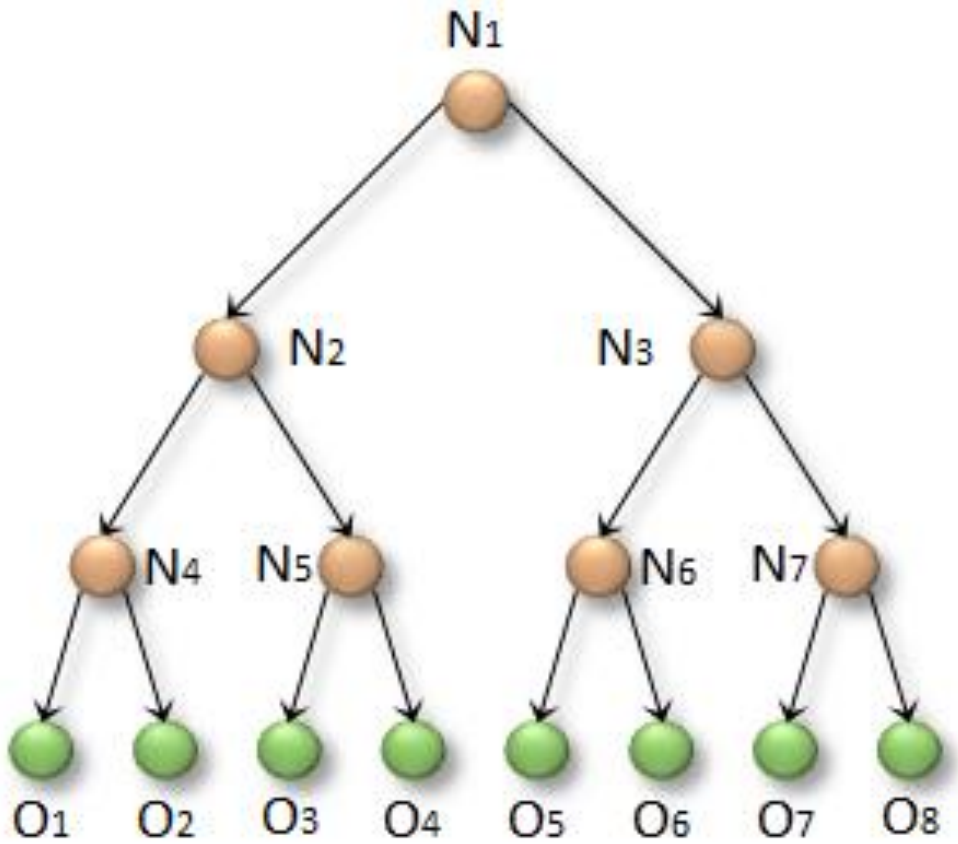
Feature	SIMD (CUDA Cores)	Systolic Array (Tensor Cores)
Model	General-purpose vector processing	Specialized dataflow architecture
Execution	Same instruction over multiple data lanes	Pipelined wave of data through fixed-function units
Communication	Registers/shared memory	Local neighbor communication in hardware
Control Logic	More flexible but control-heavy	Very low control overhead, data flows autonomously
Target Workloads	General parallel computing	Repeated matrix multiplies (e.g., DNNs, GEMMs)

Real-Time Ray Tracing

RTX Mega Geometry enables hundreds of millions of animated triangles through real-time subdivision surfaces

Bounding Volume Hierarchy



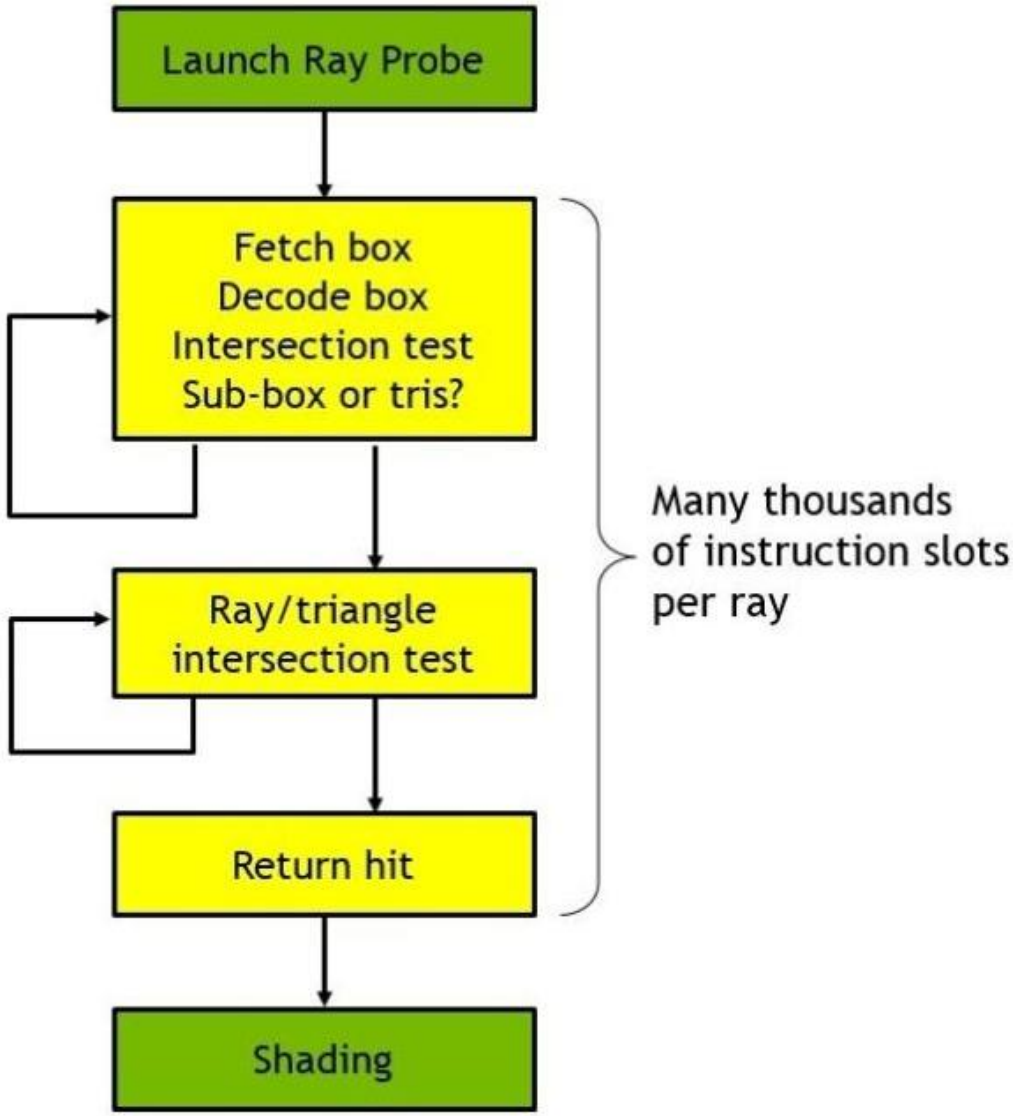


https://developer.nvidia.com/blog/thinking-parallel-part-ii-tree-traversal-gpu/

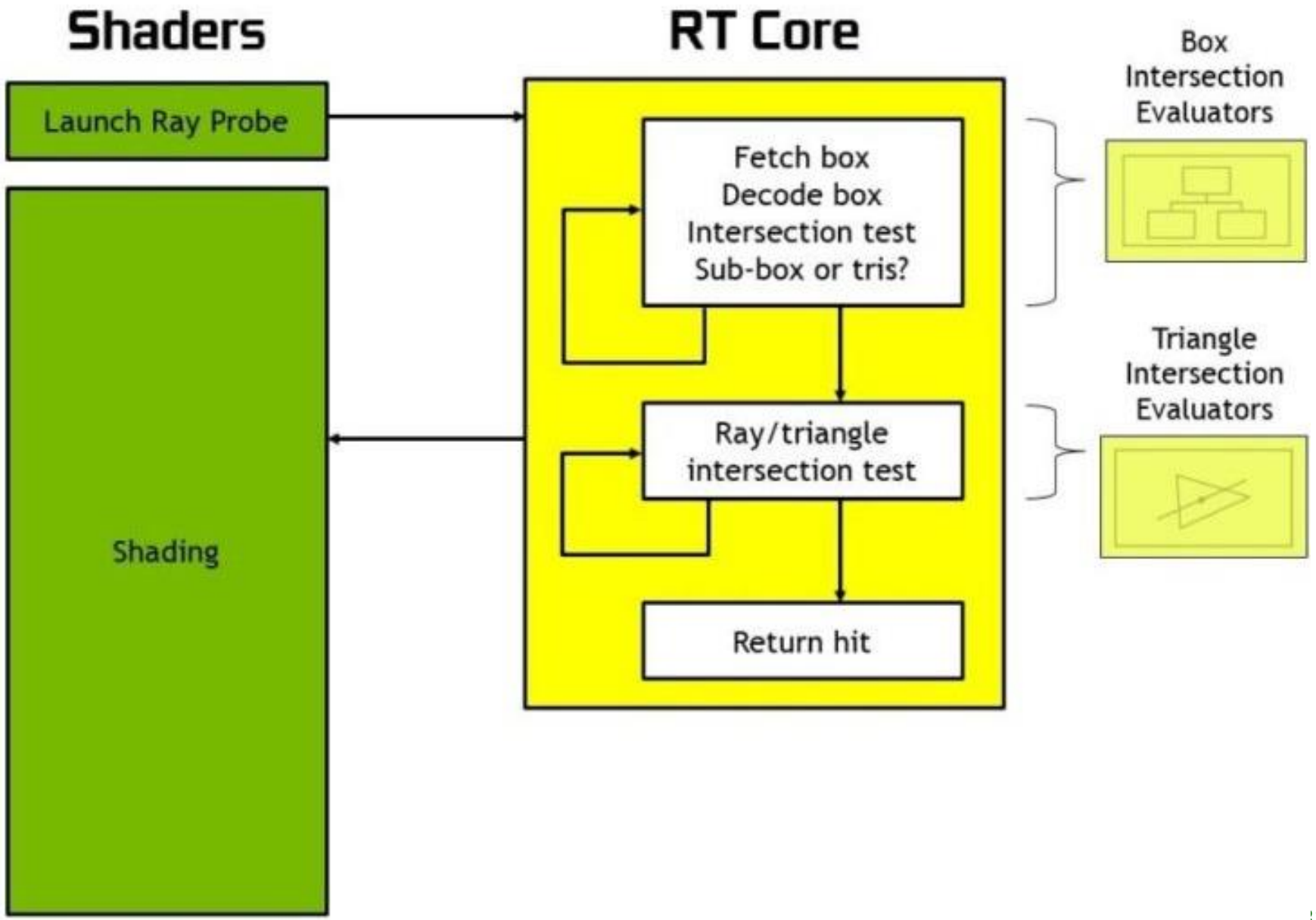
Software Emulation for BVH Search

Pascal SM

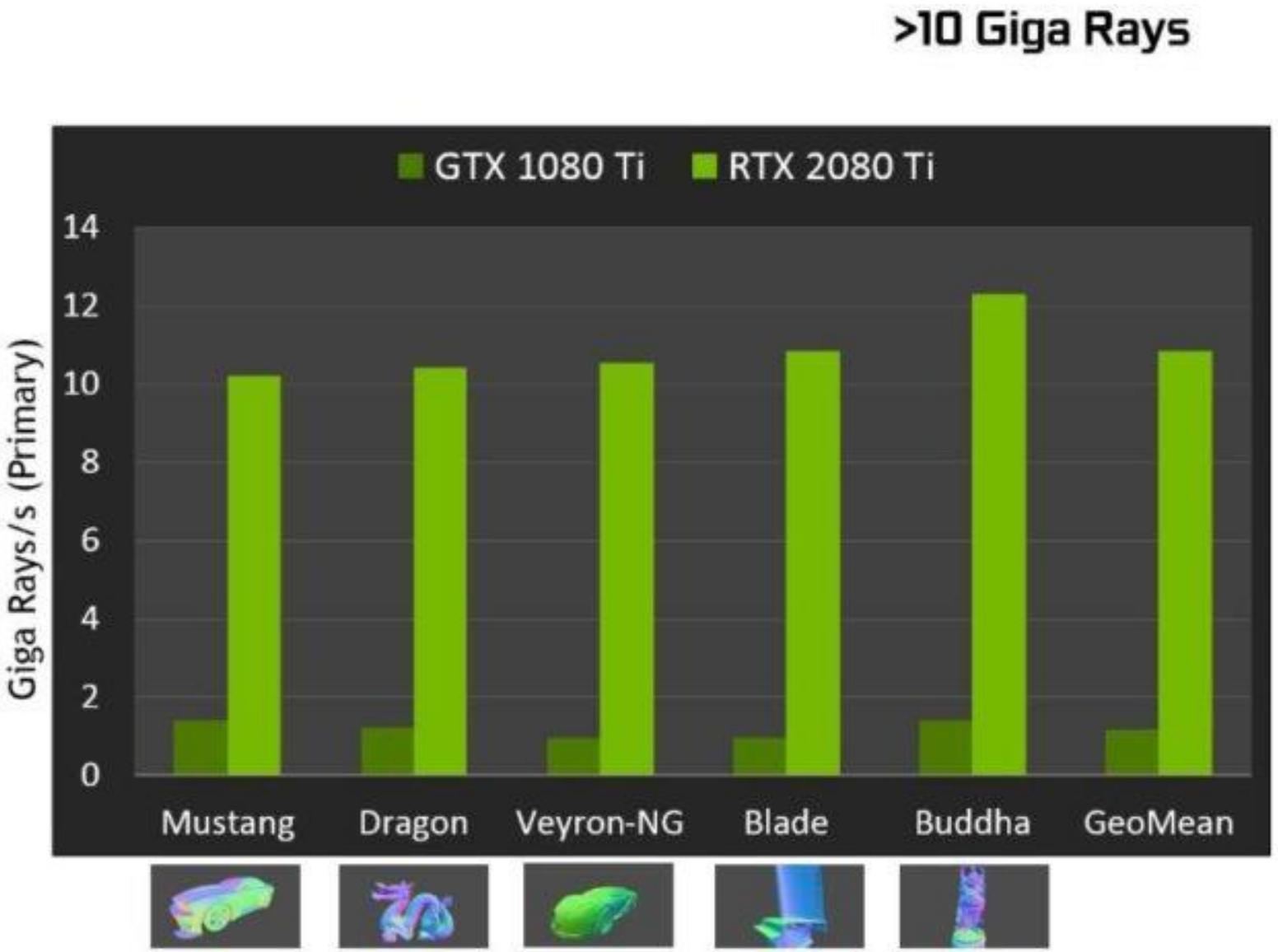
Shaders



Ray Tracing with RT Cores



Turing Ray Tracing Performance



GTX 1080 Ti	RTX 2080 Ti
11.3 TFLOPS	68 RT Cores
1.1 Giga Rays	10+ Giga Rays
10 TFLOPS / Giga Ray	~10X faster than 1080 Ti

NVIDIA GeForce RTX 3080 “core”

- Ampere

Streaming Multiprocessor (SM)

- Each SM has:
  - 64 FP32 cores
  - 32 FP64 cores
  - 64 KB registers
  - 192 KB L1 + shared mem

GA100

NVIDIA GeForce RTX 3080

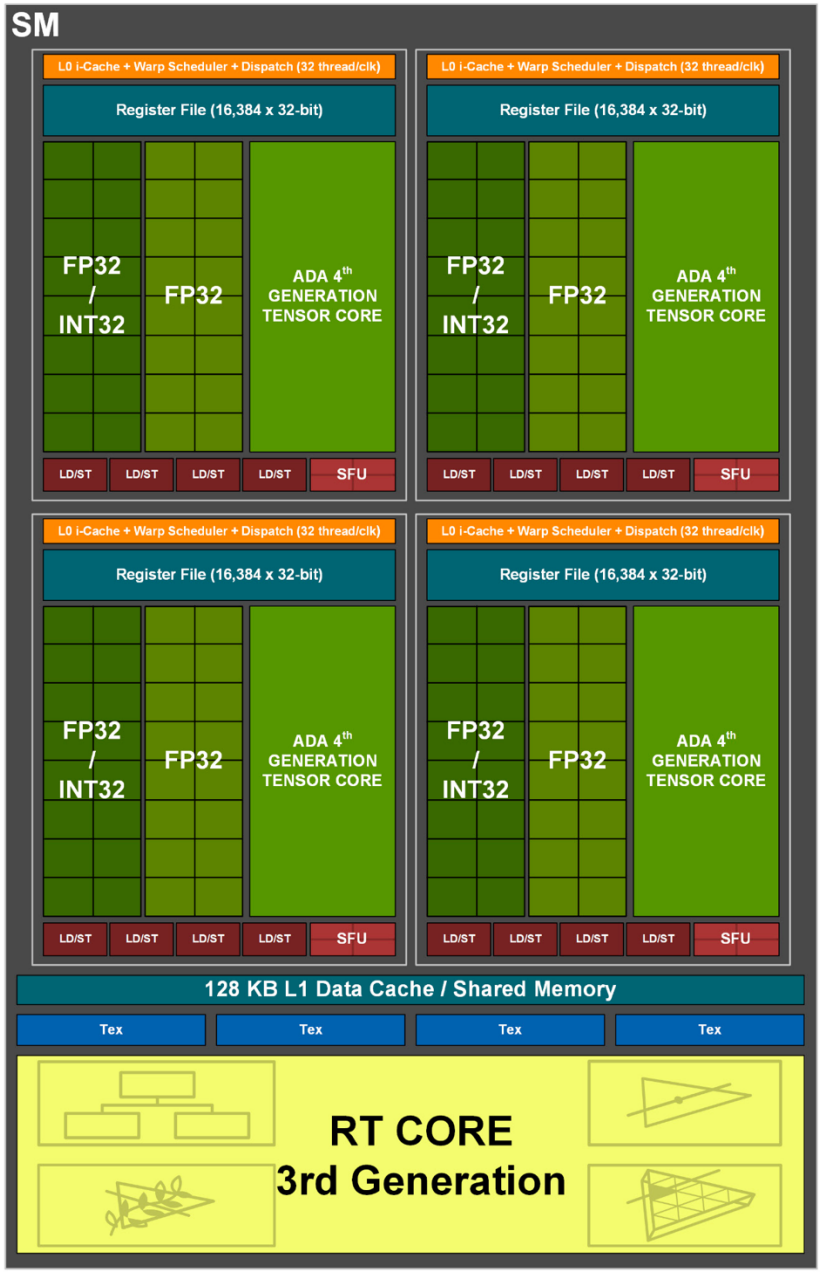
There are 68 SMs on the RTX 3080

That's 139,264 fragments!

(Or 139,264 "CUDA threads")

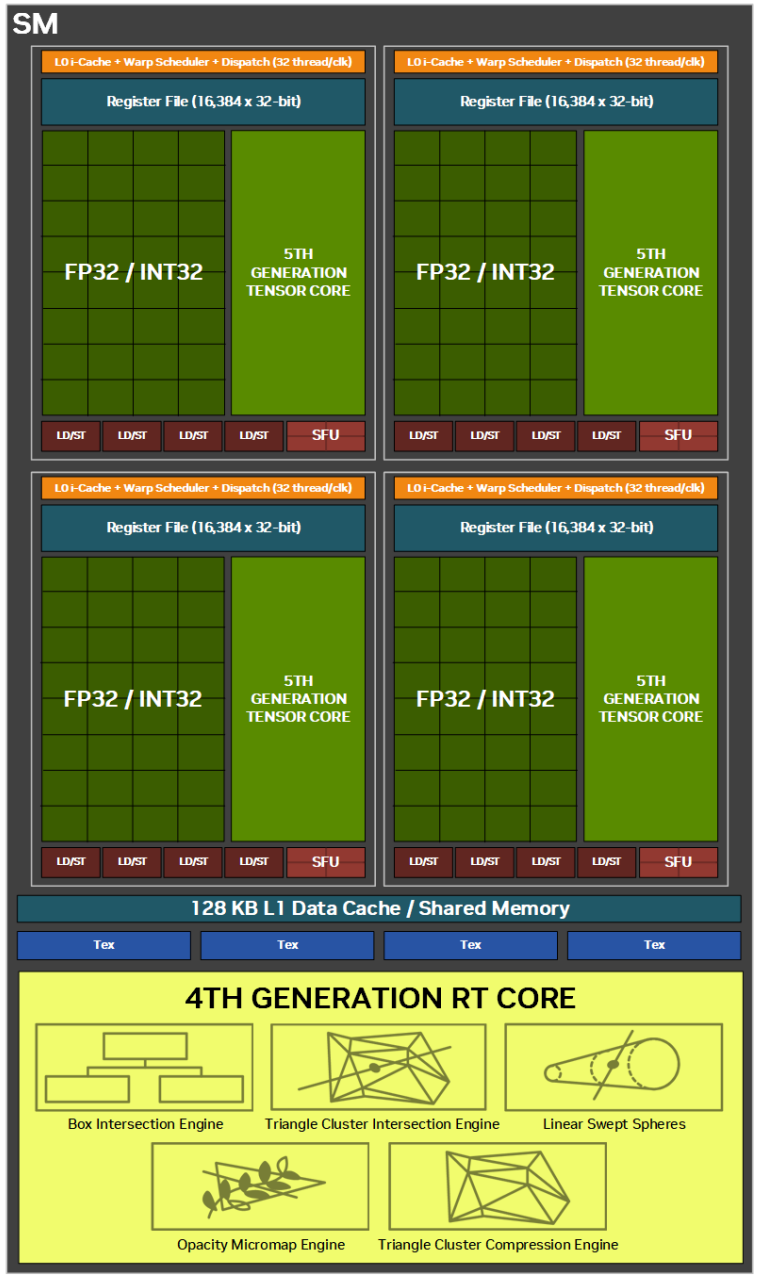
NVIDIA GeForce RTX 4080 “core”

- Ada

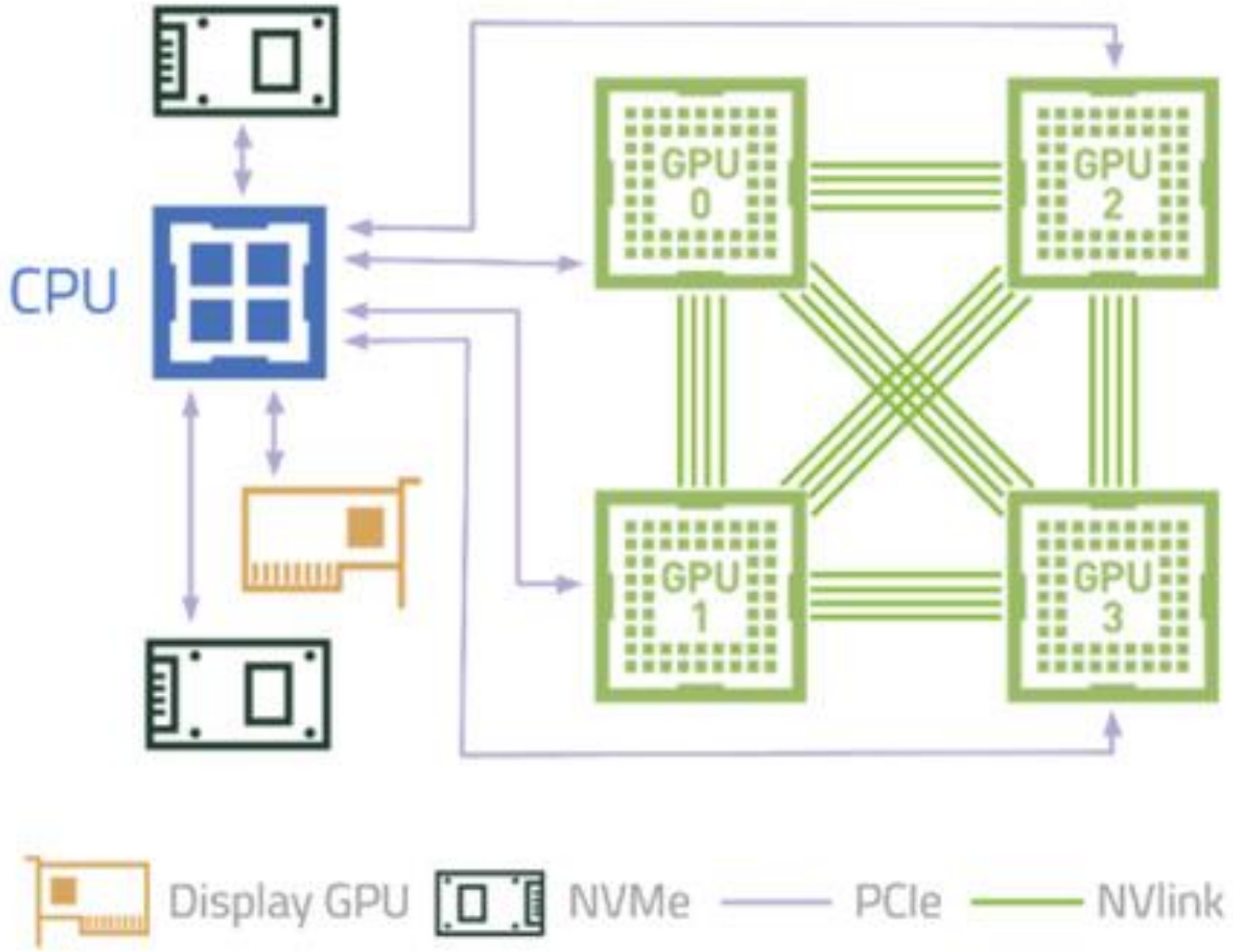


NVIDIA GeForce RTX 5080 “core”

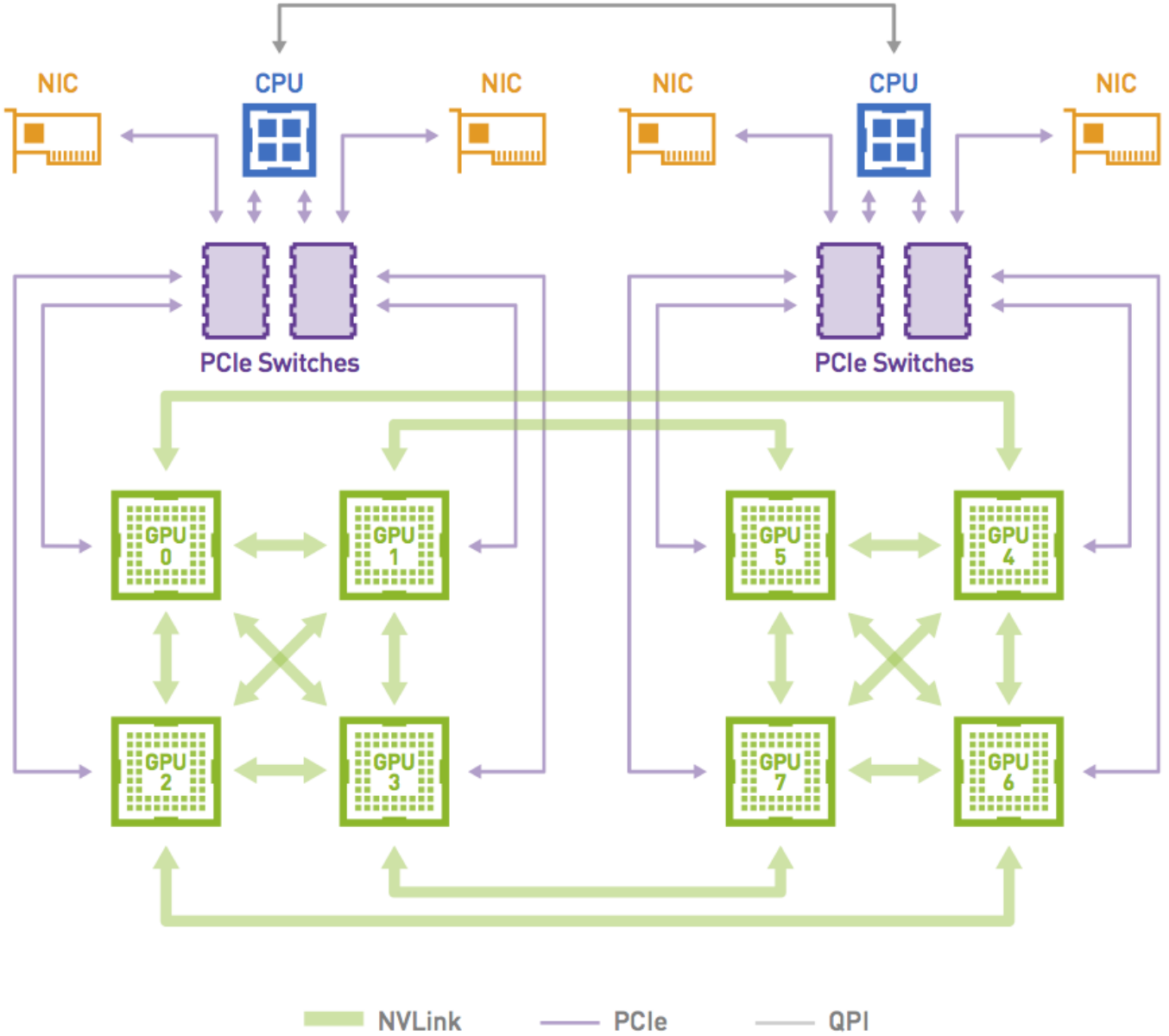
- Blackwell



Multi-GPU System

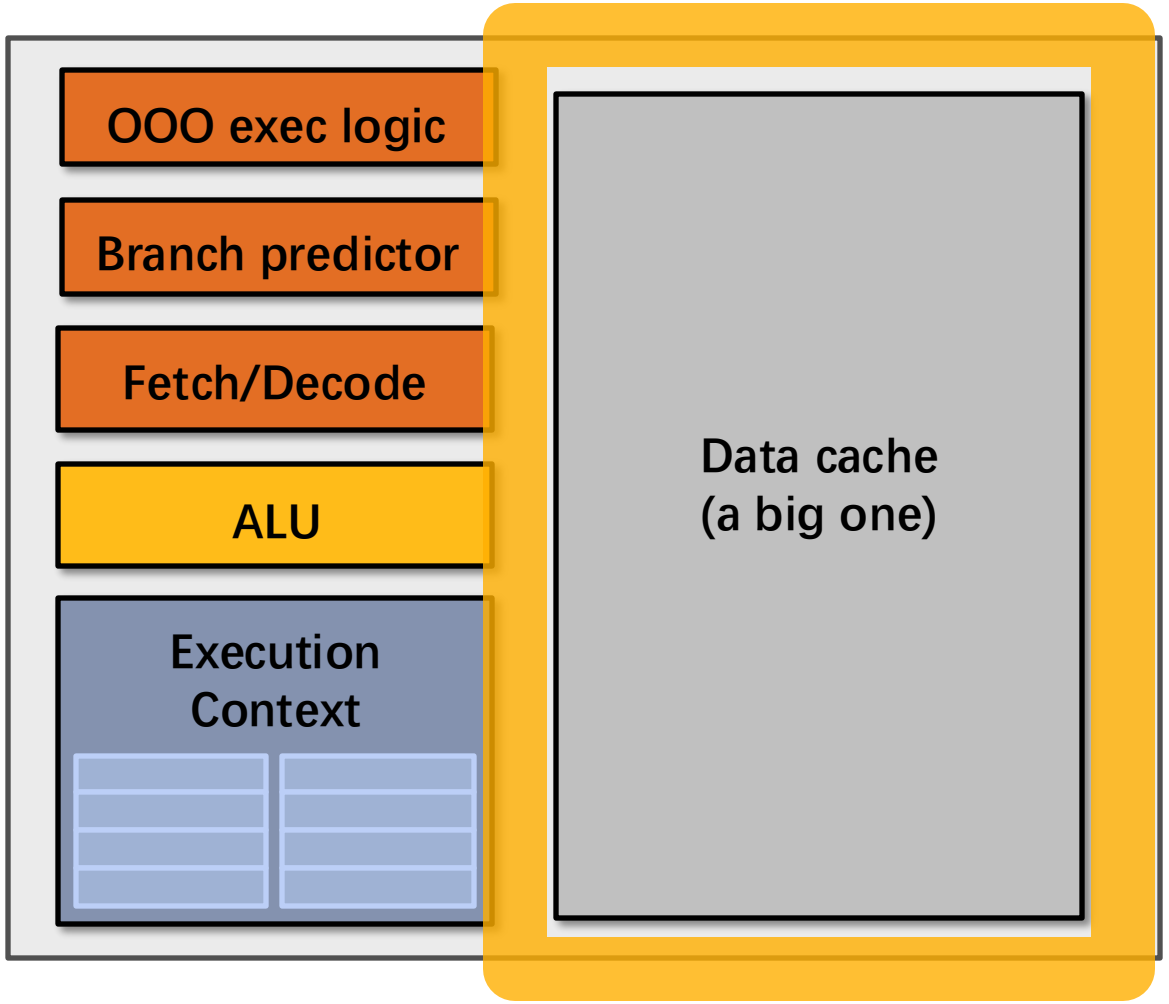


Multi-GPU System

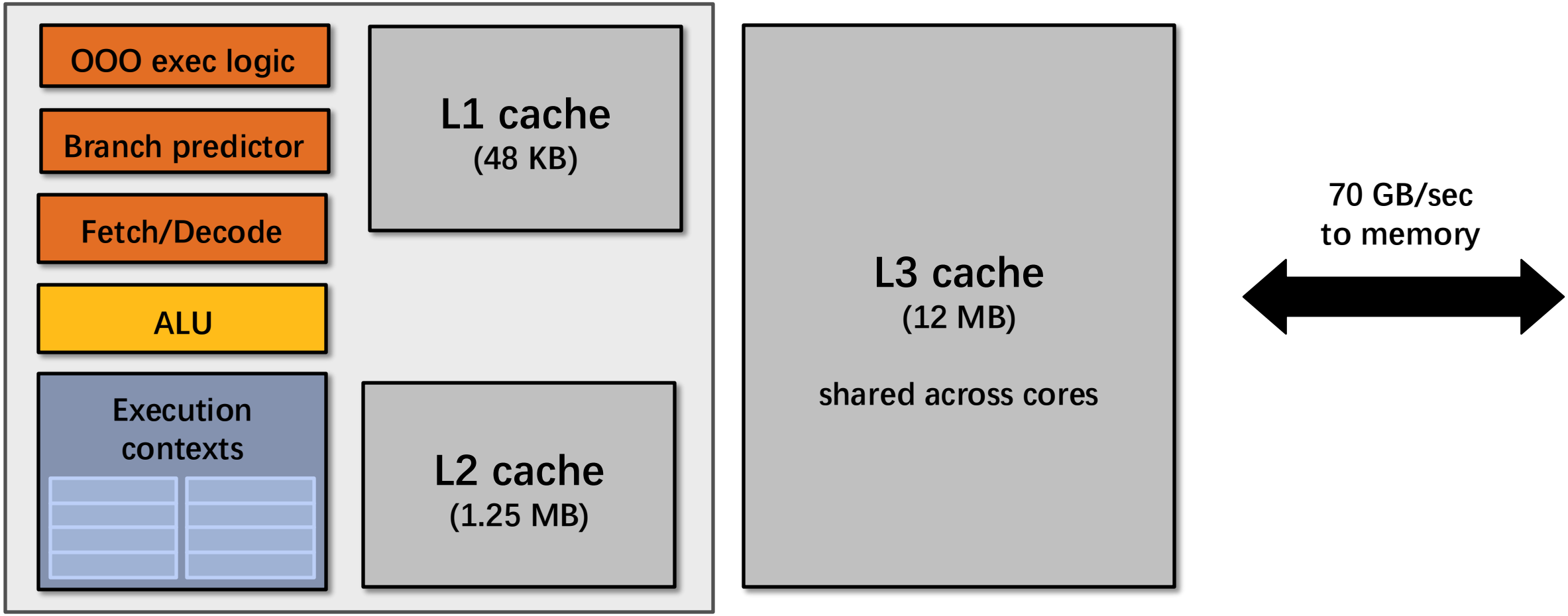


Moving Data to Processors

Recall: “CPU-style” core

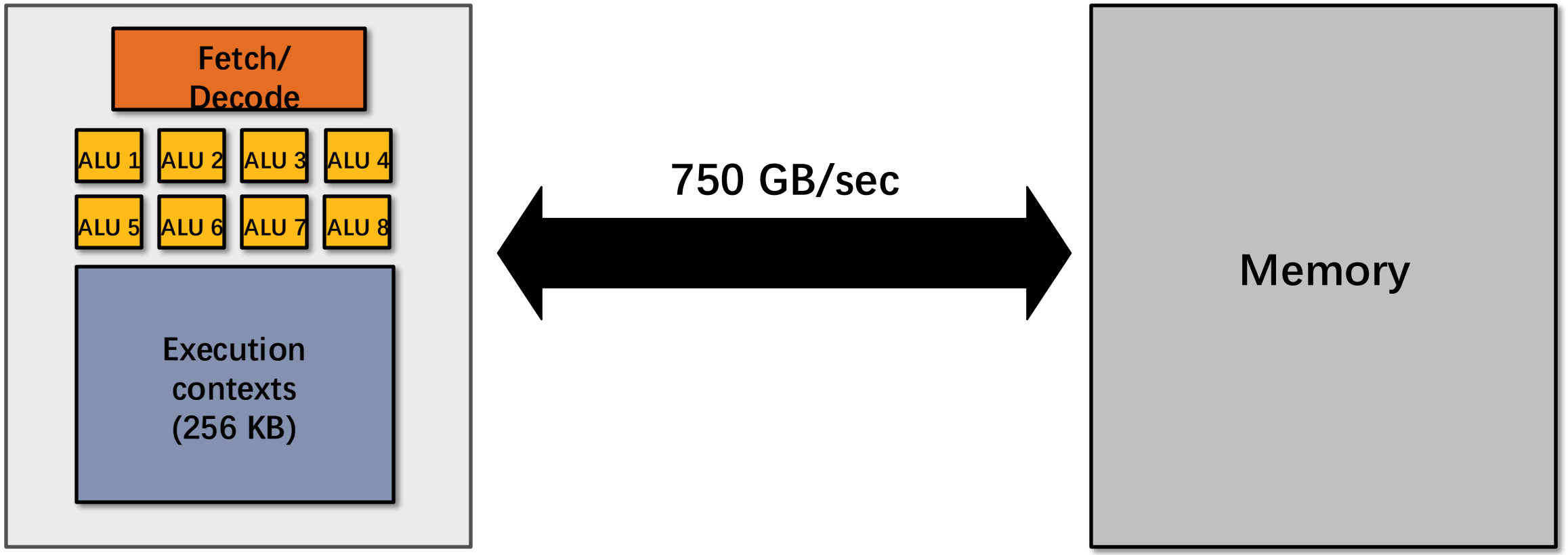


“CPU-style” memory hierarchy



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

Throughput core (GPU-style)



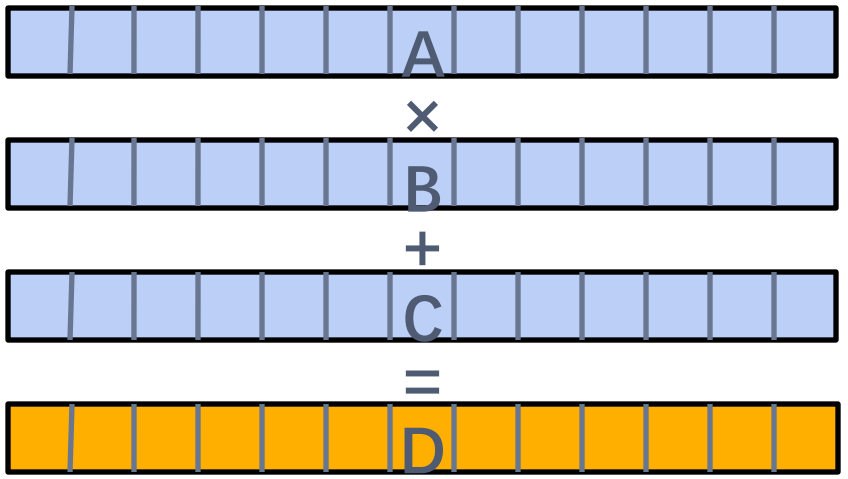
More ALUs, no large traditional cache hierarchy:
Need high-bandwidth connection to memory

Bandwidth is a critical resource

- A high-end GPU (e.g. RTX 3080) has…
  - About **sixty** times (30 TFLOPS) the compute performance of a 4-core CPU
  - No large cache hierarchy to absorb memory requests
- GPU memory system is designed for throughput
  - Wide bus (750 GB/sec)
  - Repack/reorder/interleave memory requests to maximize use of mem. bus
  - Still, this is only **ten** times the bandwidth available to CPU

Bandwidth thought experiment

- Task: element-wise multiply two long vectors A and B
  - Load input A[i]
  - Load input B[i]
  - Load input C[i]
  - Compute A[i] × B[i] + C[i]
  - Store result into D[i]



- Four memory operations (16 bytes) for every MUL-ADD
- RTX 3080 can do 8704 MUL-ADDS per clock
- Need ~240 TB/sec of bandwidth to keep functional units busy
- Roughly 0.33% efficiency… but 10x faster than CPU!

Bandwidth limited!

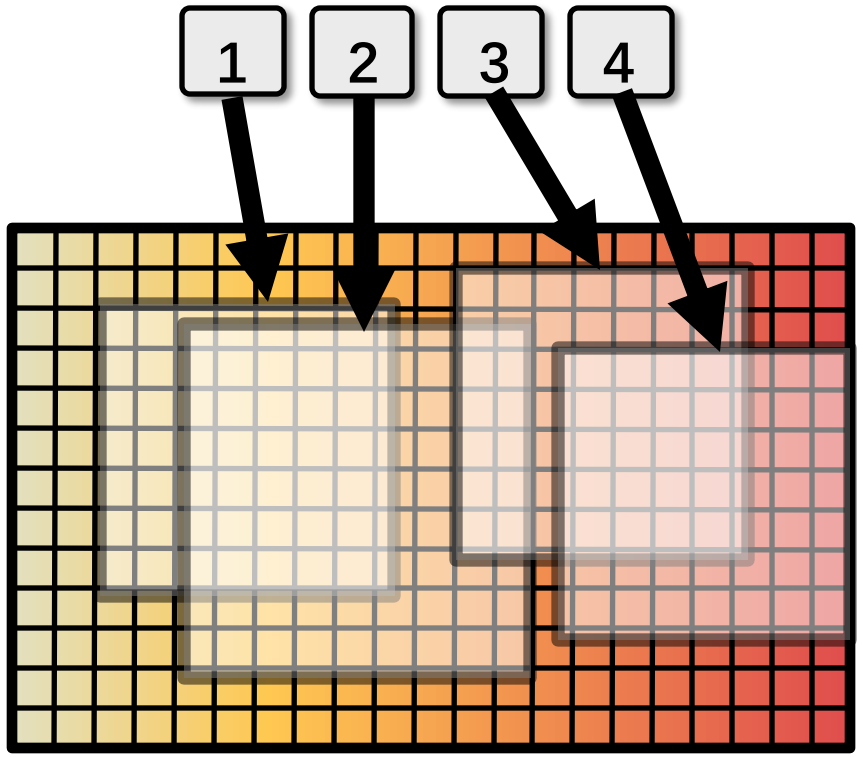
- If processors request data at too high a rate, the memory system cannot keep up
- No amount of latency hiding helps this
- Overcoming bandwidth limits are a common challenge for GPU-compute application developers

Reducing bandwidth requirements

- Request data less often (instead, do more math)
  - "arithmetic intensity"
- Fetch data from mem. less often (share/reuse data across fragments
  - on-chip communication or storage

Reducing bandwidth requirements

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

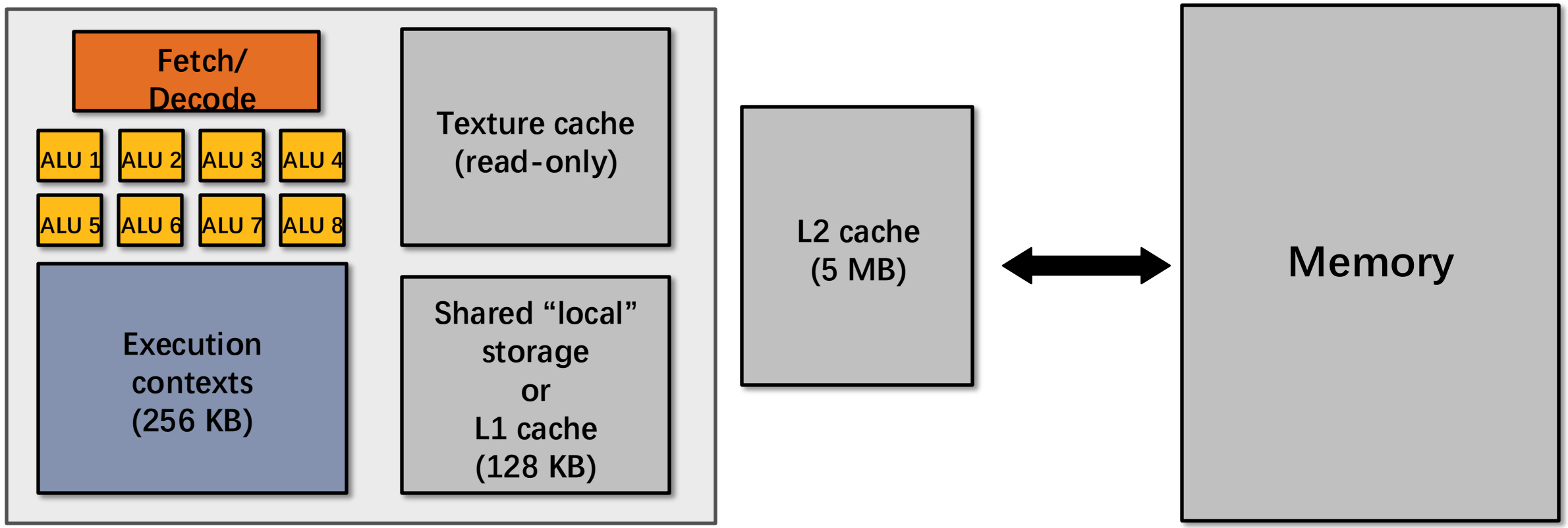


Texture data

Texture caches:

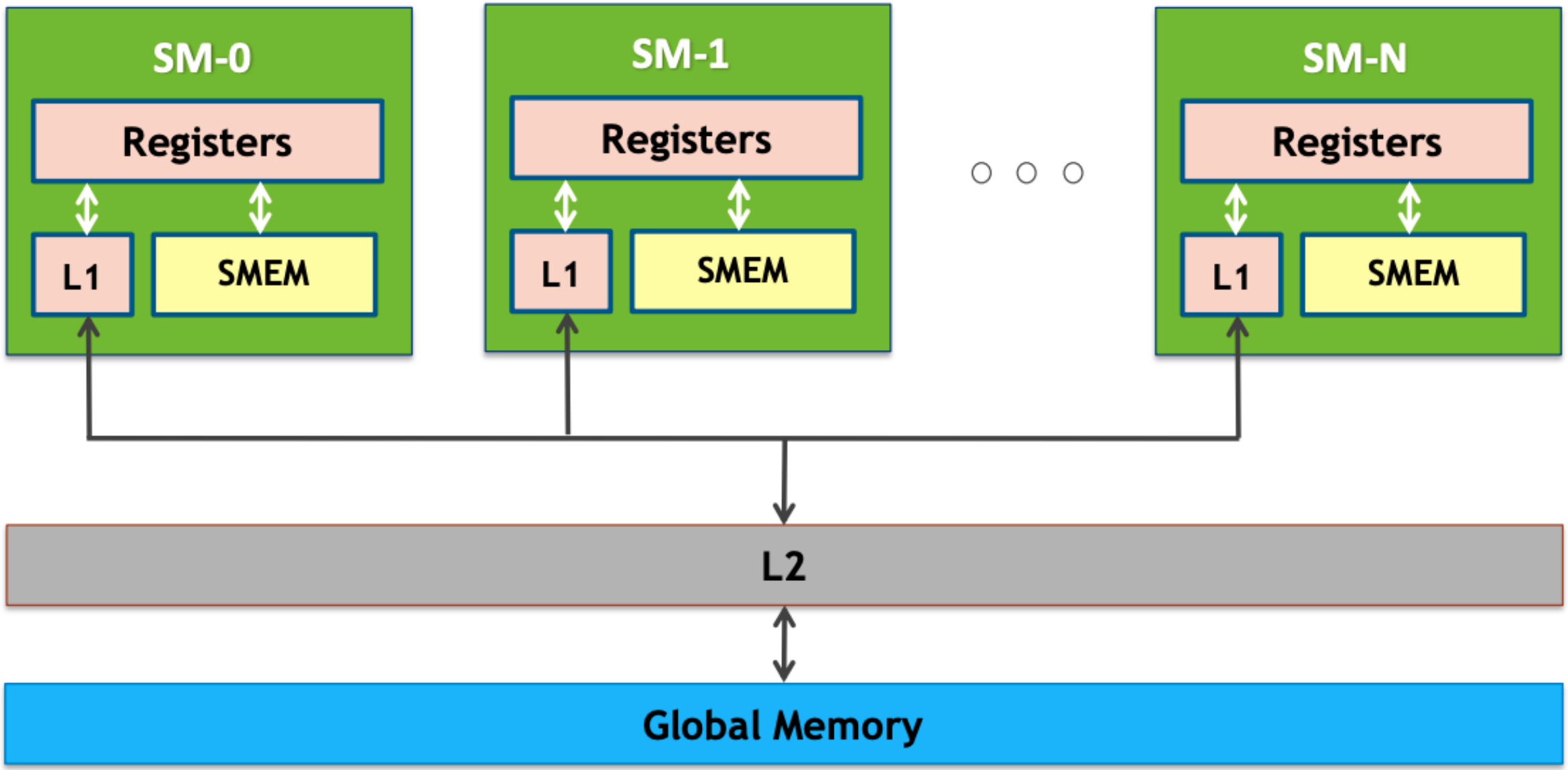
Capture reuse across fragments, not temporal reuse within a single shader program

Modern GPU memory hierarchy



- On-chip storage takes load off memory system
- Many developers calling for more cache-like storage
- (particularly GPU-compute applications)

GPU Memory Hierarchy



Summary: GPU Architecture

- Think of a GPU as a **vectorized many-core** processor optimized for maximum throughput
  - **Massive Parallelism**
  - **SIMD processing**
  - **Warp Interleaving**
  - **Specialized Units:** Tensor cores, RT cores

- Think about how the various programming models map to these architectural characteristics

Summary: GPU Architecture

- 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 memory bandwidth

- Next lecture: GPU Programming