CSC 2224: Parallel Computer Architecture and Programming GPU Architecture: Introduction

Prof. Gennady Pekhimenko

University of Toronto

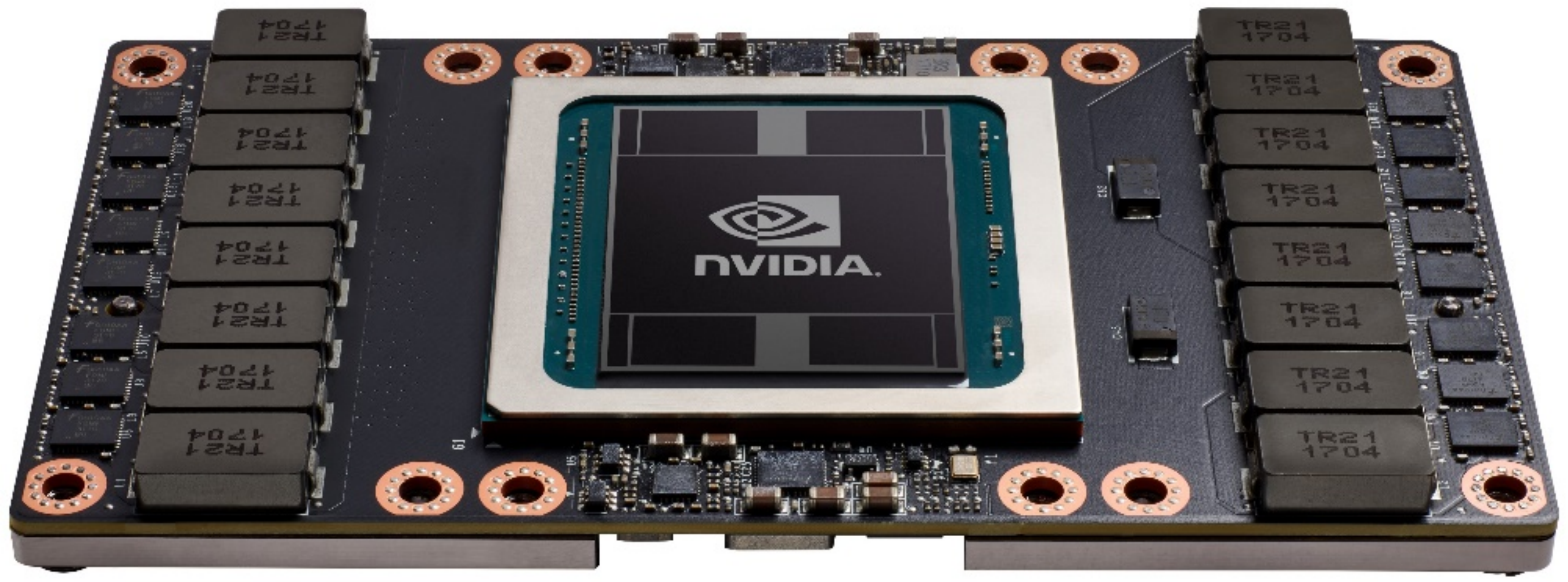
Fall 2019

The content of this lecture is adapted from the slides of Kayvon Fatahalian (Stanford), Olivier Giroux and Luke Durant (Nvidia), Tor Aamodt (UBC) and Edited by: Serina Tan

https://www.youtube.com/watch?v=-P28LKWTzrI

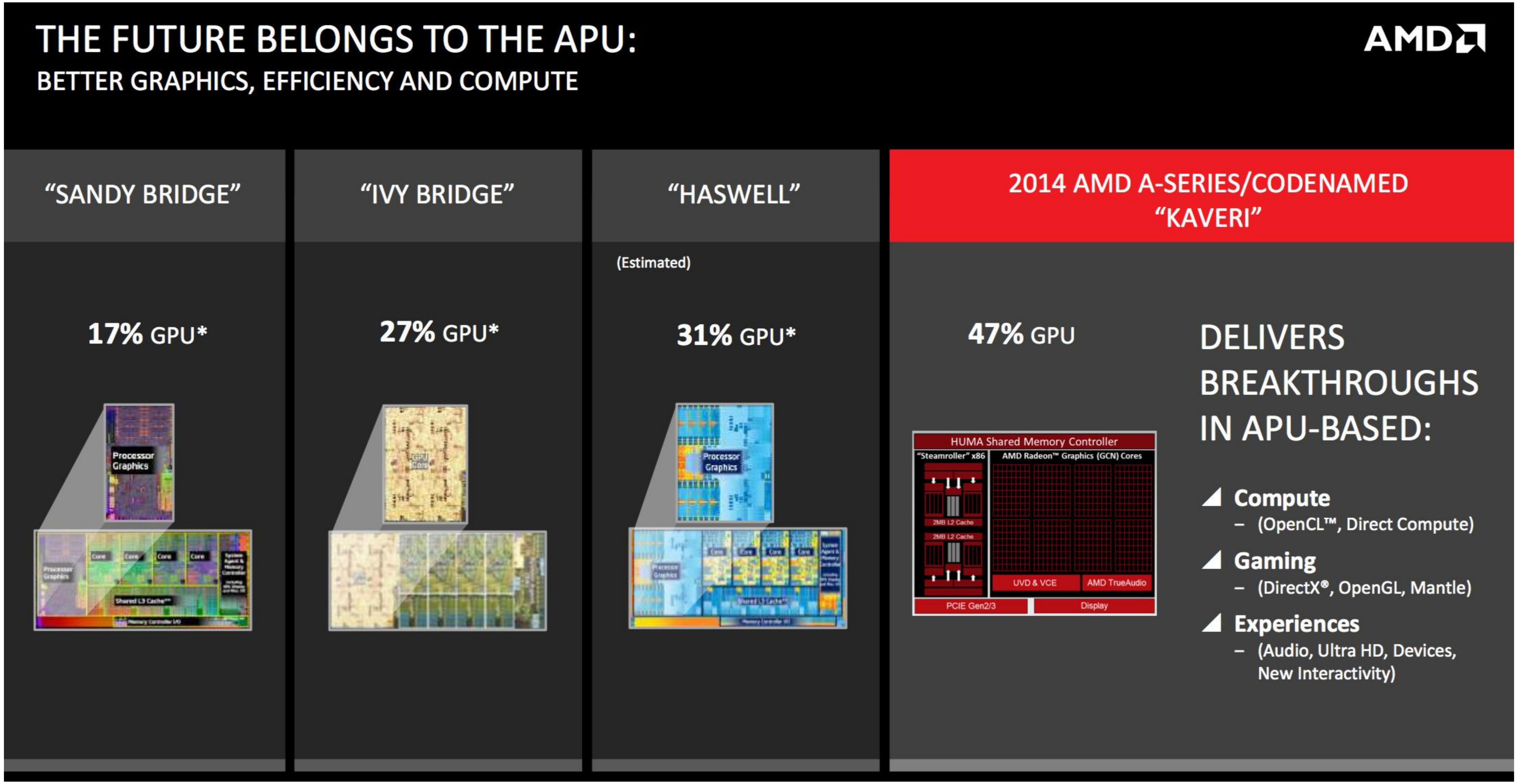
What is a GPU?

- GPU = Graphics Processing Unit
  - Accelerator for raster based graphics (OpenGL, DirectX)
  - Highly programmable (Turing complete)
  - Commodity hardware
  - 100’s of ALUs; 10’s of 1000s of concurrent threads



NVIDIA Volta: V100

The GPU is Ubiquitous



THE FUTURE BELONGS TO THE APU:
BETTER GRAPHICS, EFFICIENCY AND COMPUTE

"SANDY BRIDGE"	"IVY BRIDGE"	"HASWELL"	2014 AMD A-SERIES/CODENAMED "KAVERI"
		(Estimated)	
17% GPU*	27% GPU*	31% GPU*	47% GPU

DELIVERS BREAKTHROUGHS IN APU-BASED:

- Compute
  - (OpenCL™, Direct Compute)
- Gaming
  - (DirectX®, OpenGL, Mantle)
- Experiences
  - (Audio, Ultra HD, Devices, New Interactivity)

AMD
Processor Graphics
Core
Core
Core
Core
System Agent & Memory Controller
Shared L3 Cache**
Memory Controller I/O
GPU Core
HUMA Shared Memory Controller
"Steamroller" x86
AMD Radeon™ Graphics (GCN) Cores
2MB L2 Cache
2MB L2 Cache
UVD & VCE
AMD TrueAudio
PCIE Gen2/3
Display

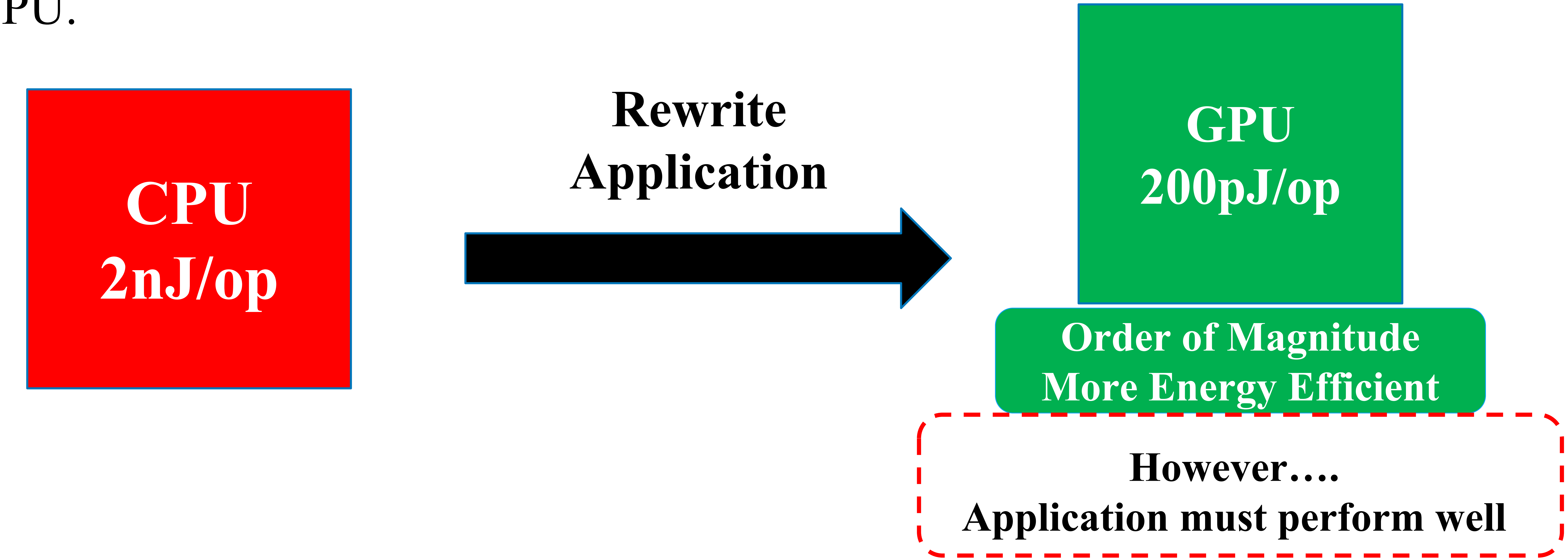
[APU13 keynote]

“Early” GPU History

- – 1981: IBM PC Monochrome Display Adapter (2D)
- – 1996: 3D graphics (e.g., 3dfx Voodoo)
- – 1999: register combiner (NVIDIA GeForce 256)
- – 2001: programmable shaders (NVIDIA GeForce 3)
- – 2002: floating-point (ATI Radeon 9700)
- – 2005: unified shaders (ATI R520 in Xbox 360)
- – 2006: compute (NVIDIA GeForce 8800)

Why use a GPU for computing?

- GPU uses larger fraction of silicon for computation than CPU.
- At peak performance GPU uses order of magnitude less energy per operation than CPU.



CPU
2nJ/op

Rewrite
Application

GPU
200pJ/op

Order of Magnitude
More Energy Efficient

However….
Application must perform well

Agenda

- Three key ideas that make GPUs run fast
- GPU memory hierarchy
- Closer look at a modern GPU architecture (Nvidia’s Volta)
  - Memory: higher bandwidth, larger capacity
  - Compute: application-specific hardware

Why GPUs Run Fast?

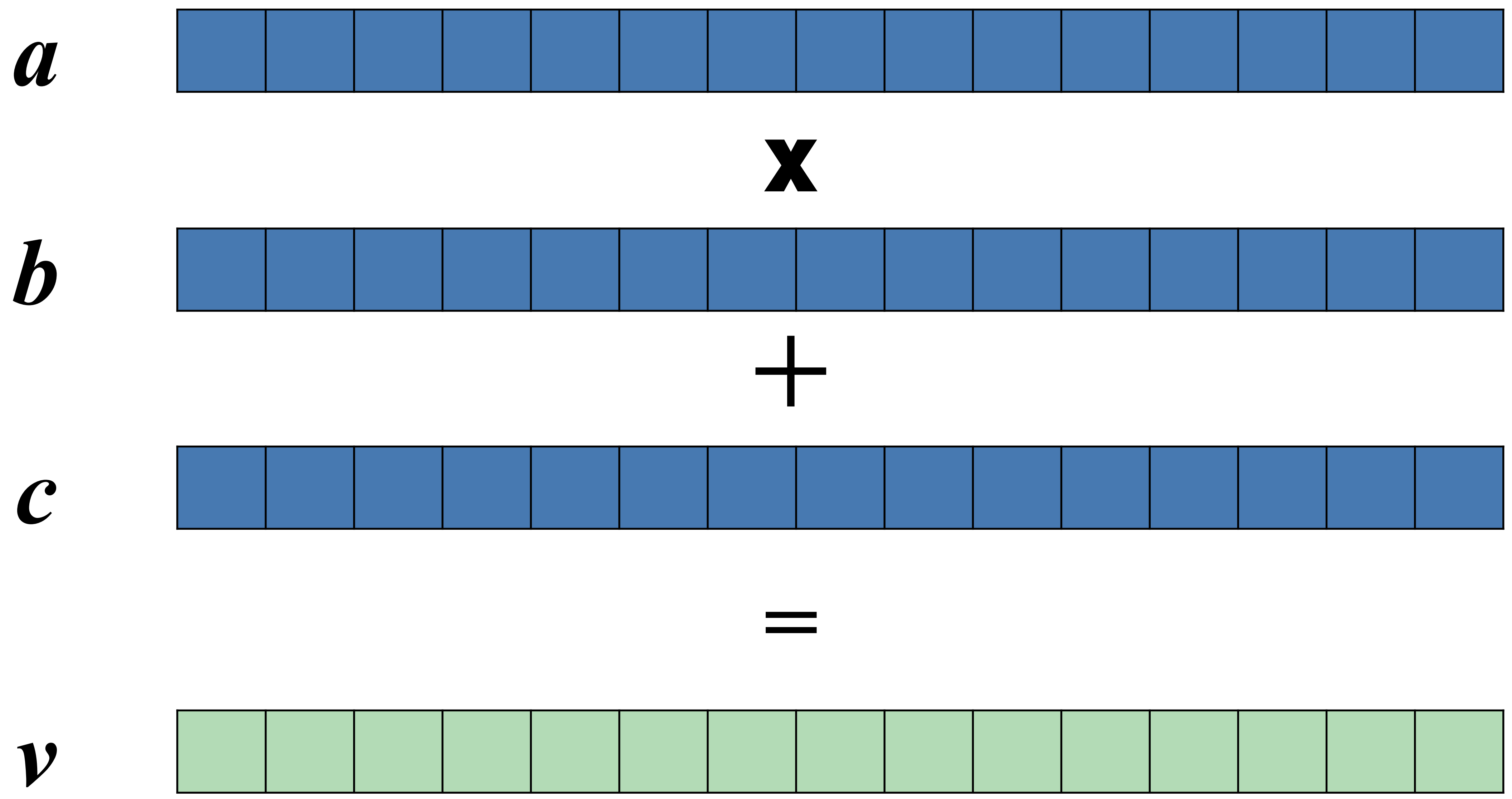
- Three key ideas behind how modern GPU processing cores run code

- Knowing these concepts will help you:

  1. Understand GPU core designs
  2. Optimize performance of your parallel programs
  3. Gain intuition about what workloads might benefit from such a parallel architecture

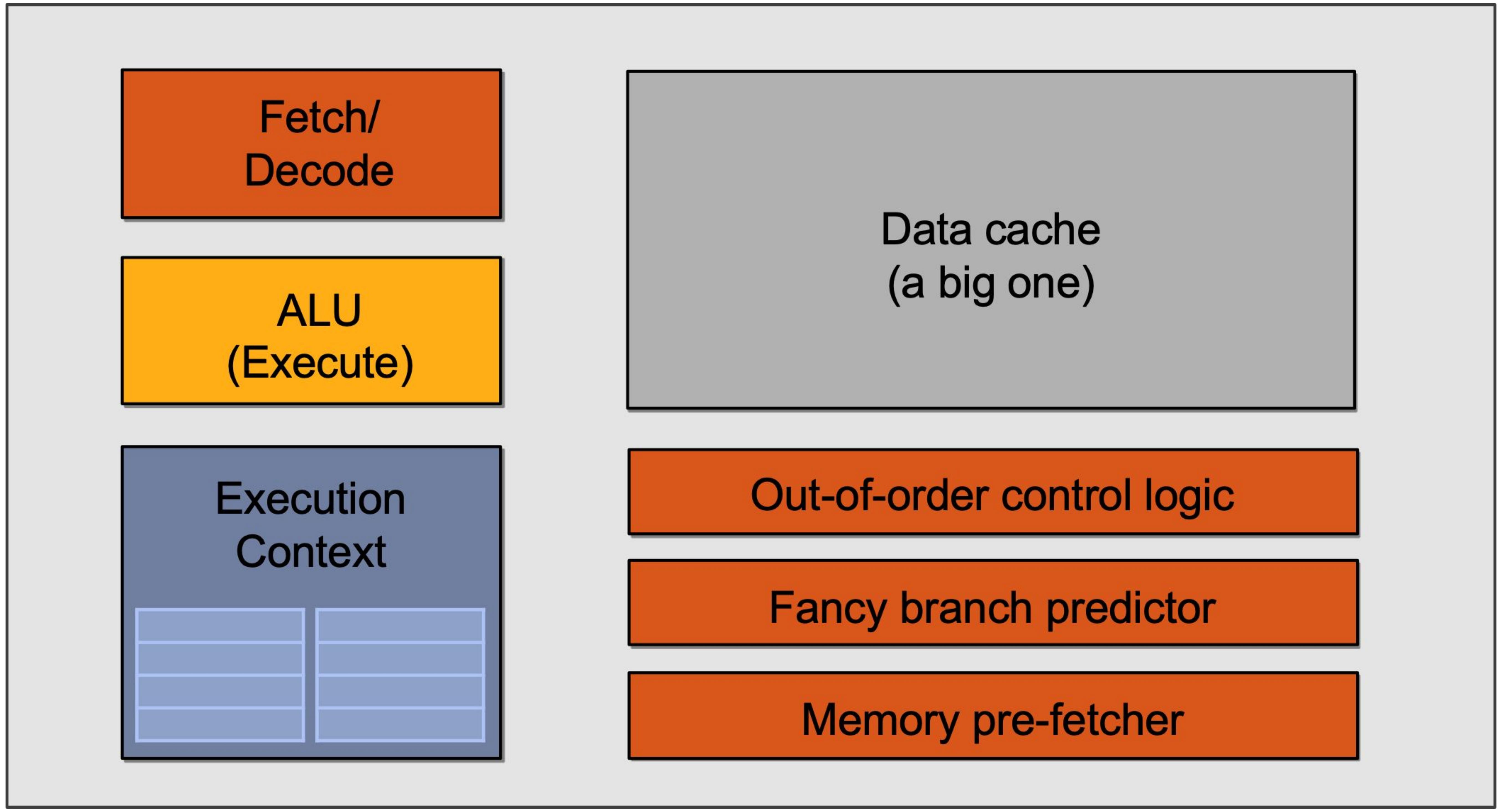
Example Program: Vector Multiply-Add

- Compute ***v*** = ***a*** · ***b*** + ***c*** *(**a**, **b**, **c** and **v** are vectors with a length of N)*



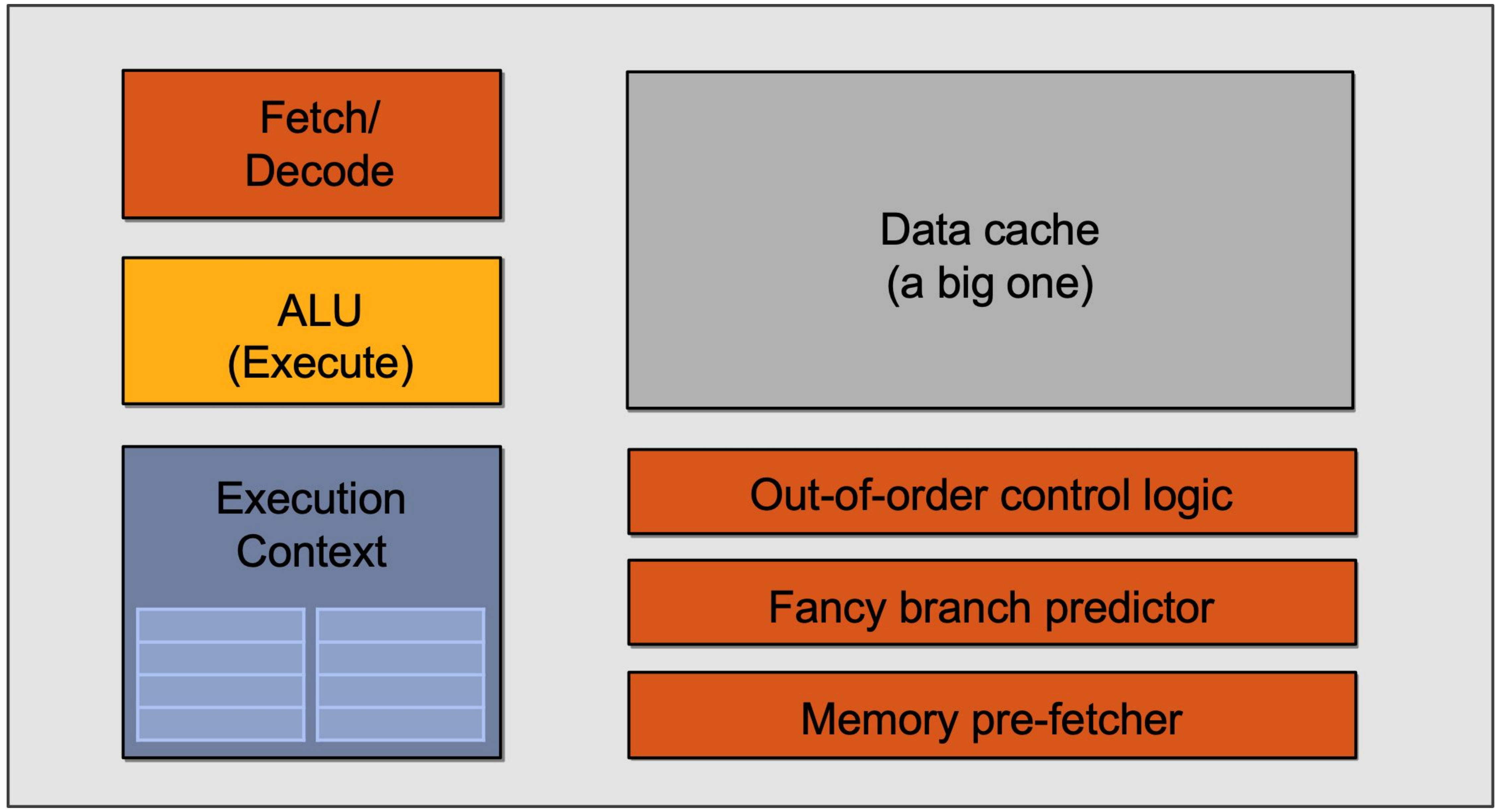
```
void mul_add (int N, float* a, float* b, float* c, float* v) {
        for (int i = 0; i < N; i++) {
                v[i] = a[i] * b[i] + c[i]
        }
}
```

Single-core CPU Execution



```
 mov R1, 0
START:
 ld R2, a[R1]
 ld R3, b[R1]
 ld R4, c[R1]
 madd R5, R2, R3, R4
 st R5, v[R1]
 add R1, R1, 1
 bra START if R1 < N
```

Single-core CPU Execution



madd stalled,
jump to the next
independent instruction

Can also be executed
out-of-order
through register renaming

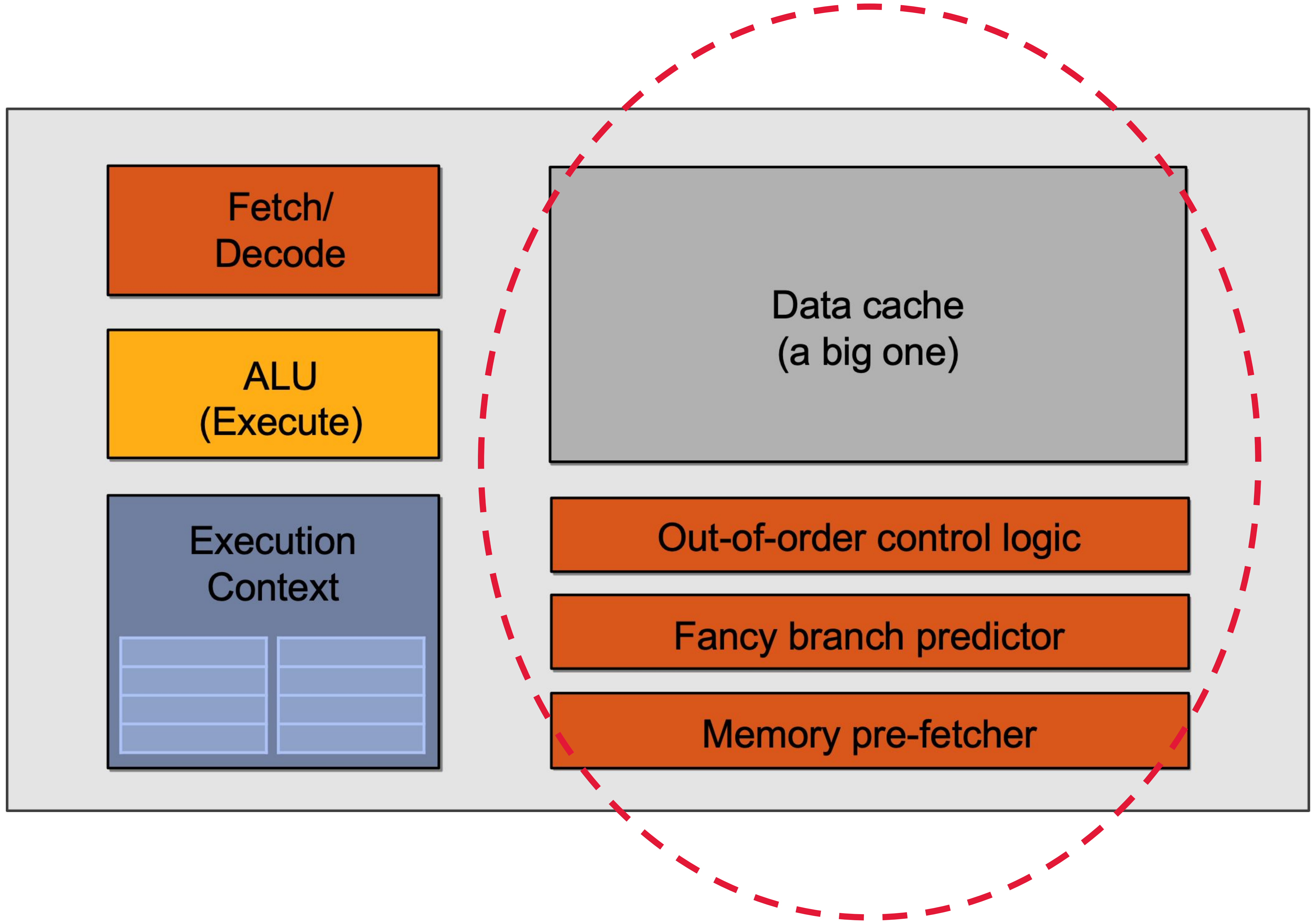
```
 mov R1, 0
START:
 ld R2, a[R1]
 ld R3, b[R1]
 ld R4, c[R1]
 madd R5, R2, R3, R4
 st R5, v[R1]
 add R1, R1, 1
 bra START if R1 < N

START:
 ld R2, a[R1]
 ld R3, b[R1]
 ld R4, c[R1]
 madd R5, R2, R3, R4
 st R5, v[R1]
 add R1, R1, 1
 bra START if R1 < N

        …
```

Instruction
Flow

Single-core CPU Execution



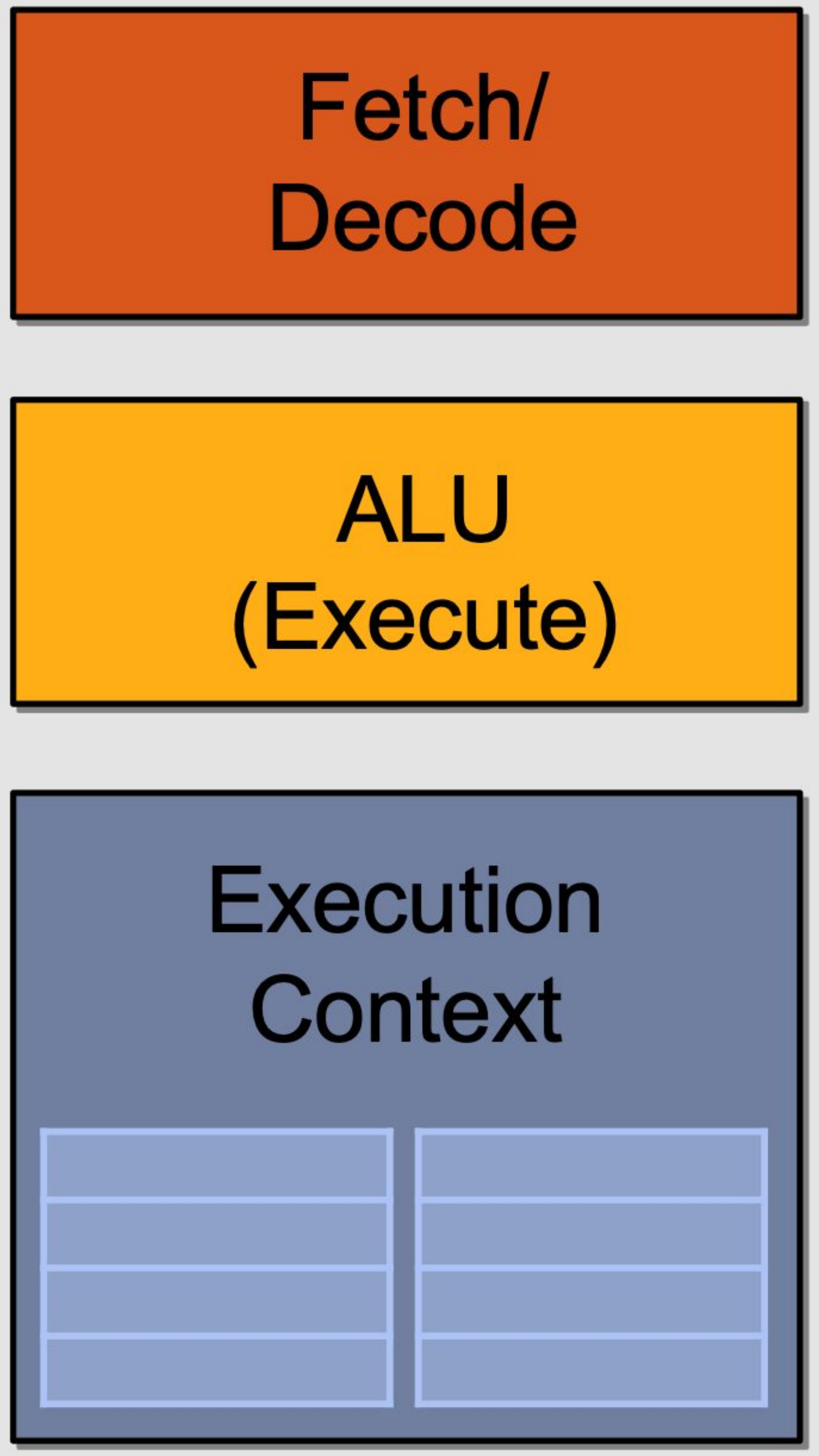
But what if we tell the hardware these two blocks can be executed in parallel to begin with?

```
  mov R1, 0
START:
  ld R2, a[R1]
  ld R3, b[R1]
  ld R4, c[R1]
  madd R5, R2, R3, R4
  st R5, v[R1]
  add R1, R1, 1
  bra START if R1 < N
START:
  ld R2, a[R1]
  ld R3, b[R1]
  ld R4, c[R1]
  madd R5, R2, R3, R4
  st R5, v[R1]
  add R1, R1, 1
  bra START if R1 < N
```

…

Instruction Flow

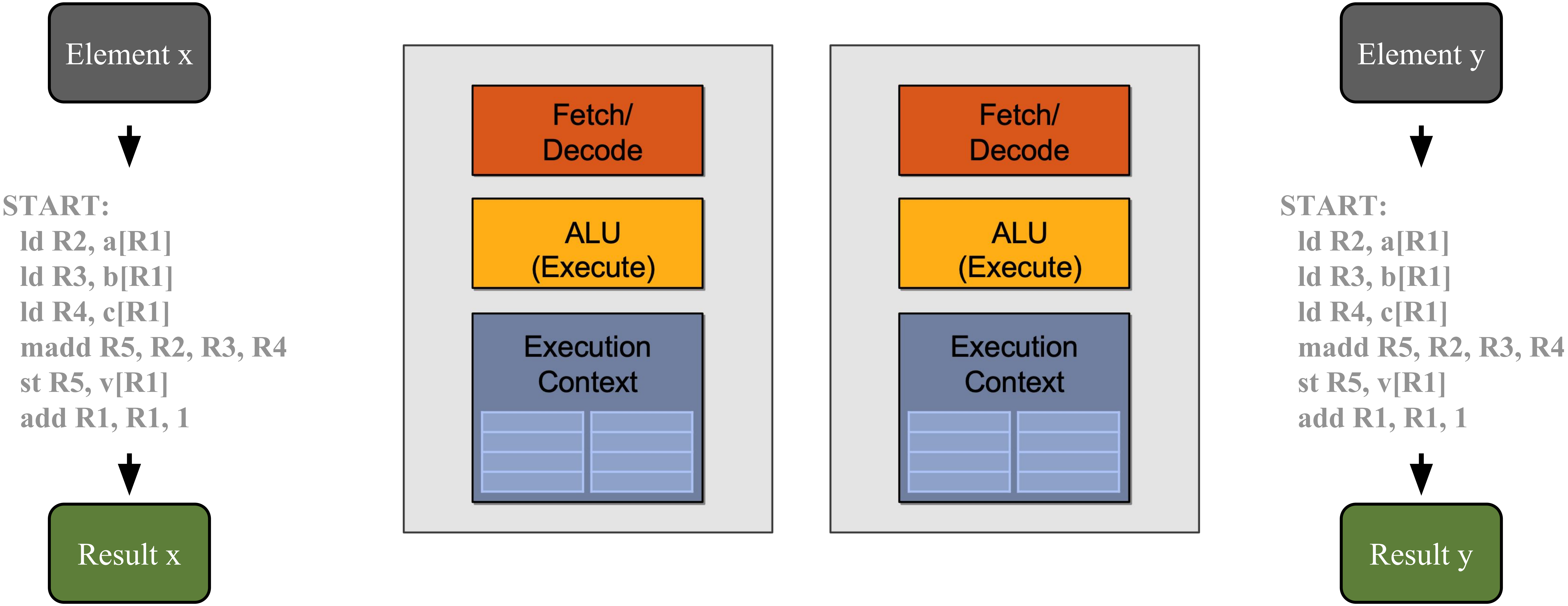
Slimming Down



Idea #1:
Use increasing transistor count to add more cores to the processor

… rather than use transistors to increase sophistication of processor logic that accelerates a single instruction stream (e.g., out-of-order and speculative operations)

Two cores (Two Elements in Parallel)



Element x

```
START:
  ld R2, a[R1]
  ld R3, b[R1]
  ld R4, c[R1]
  madd R5, R2, R3, R4
  st R5, v[R1]
  add R1, R1, 1
```

Result x

Fetch/ Decode

ALU (Execute)

Execution Context

Fetch/ Decode

ALU (Execute)

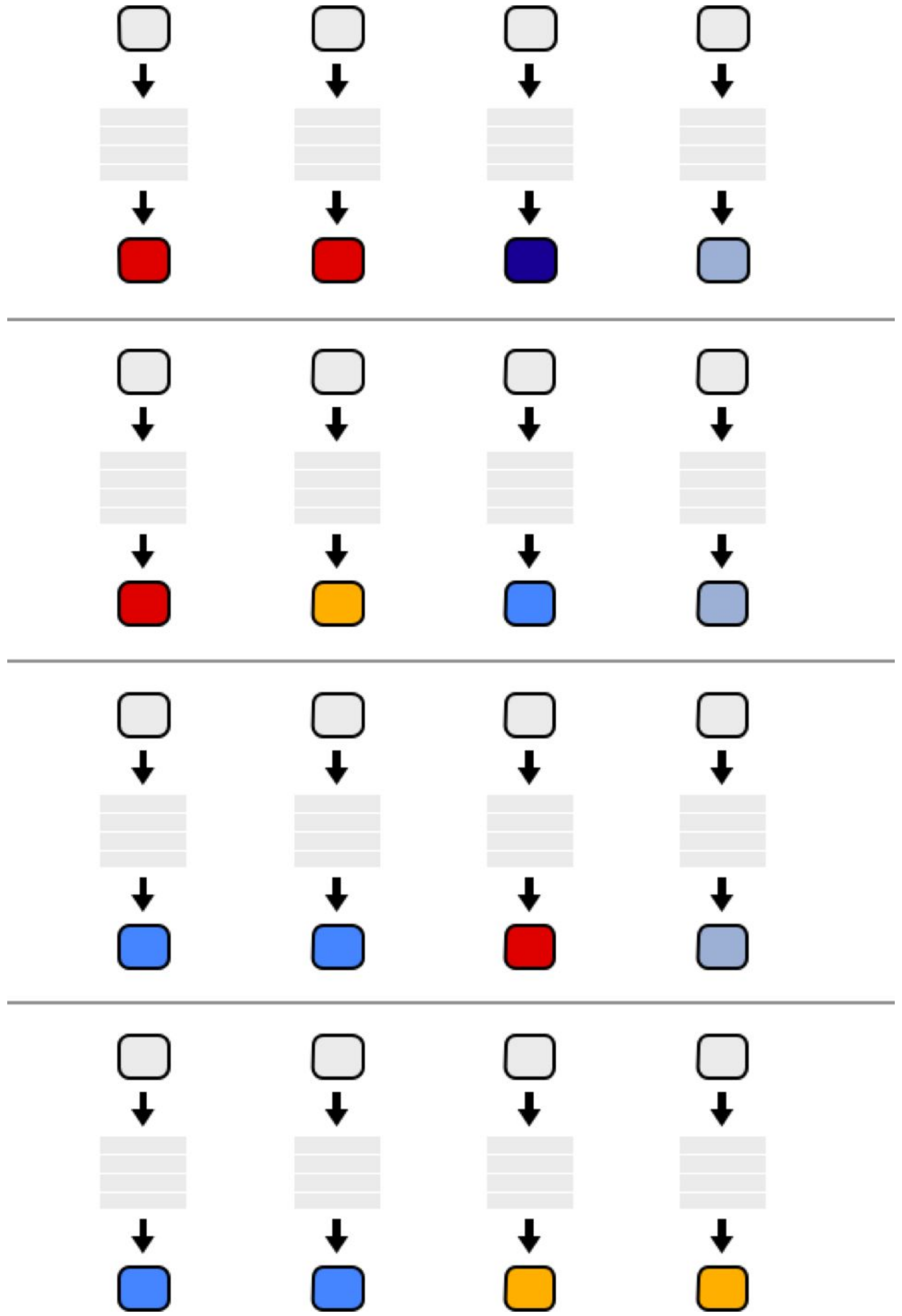
Execution Context

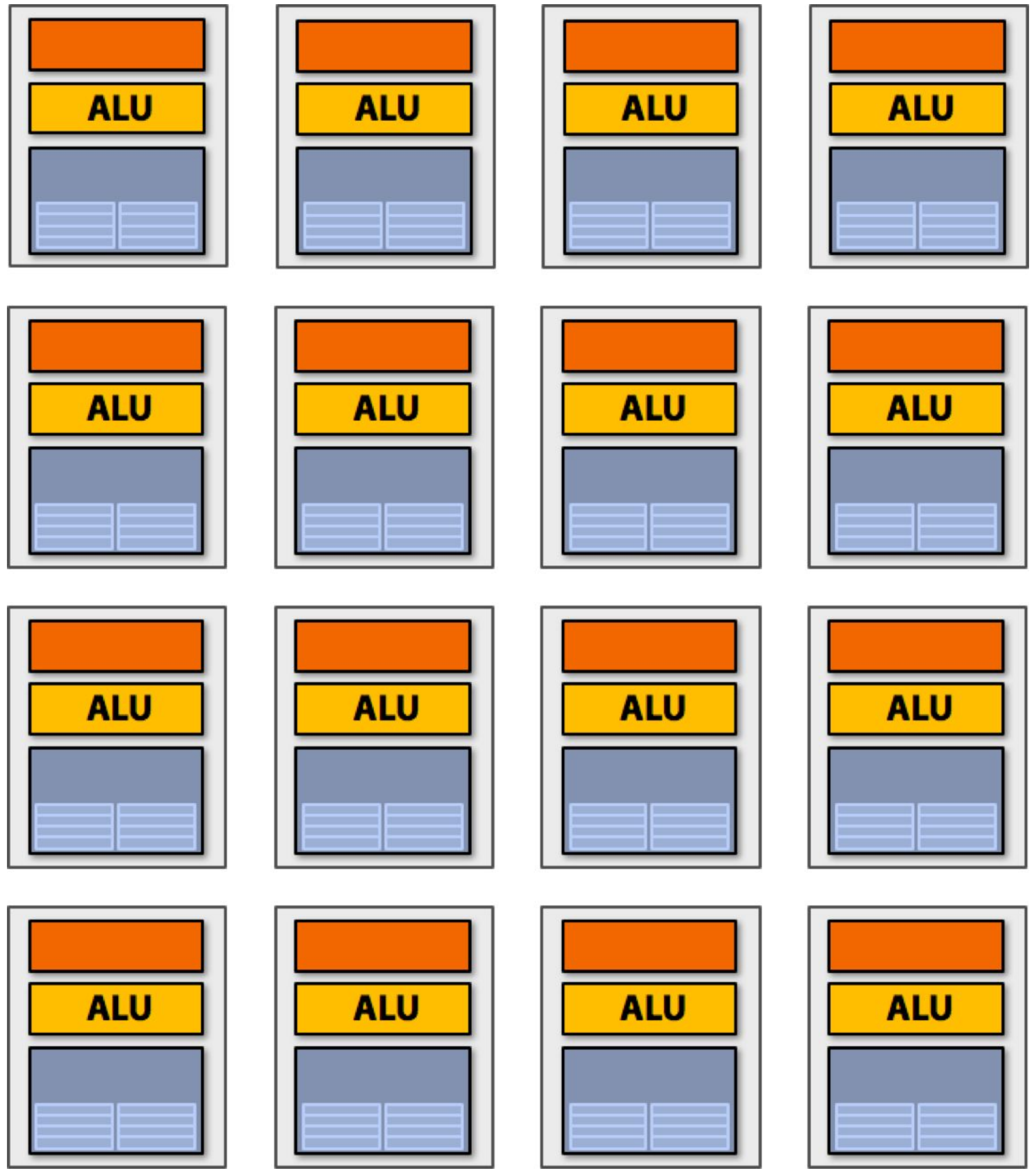
Element y

```
START:
  ld R2, a[R1]
  ld R3, b[R1]
  ld R4, c[R1]
  madd R5, R2, R3, R4
  st R5, v[R1]
  add R1, R1, 1
```

Result y

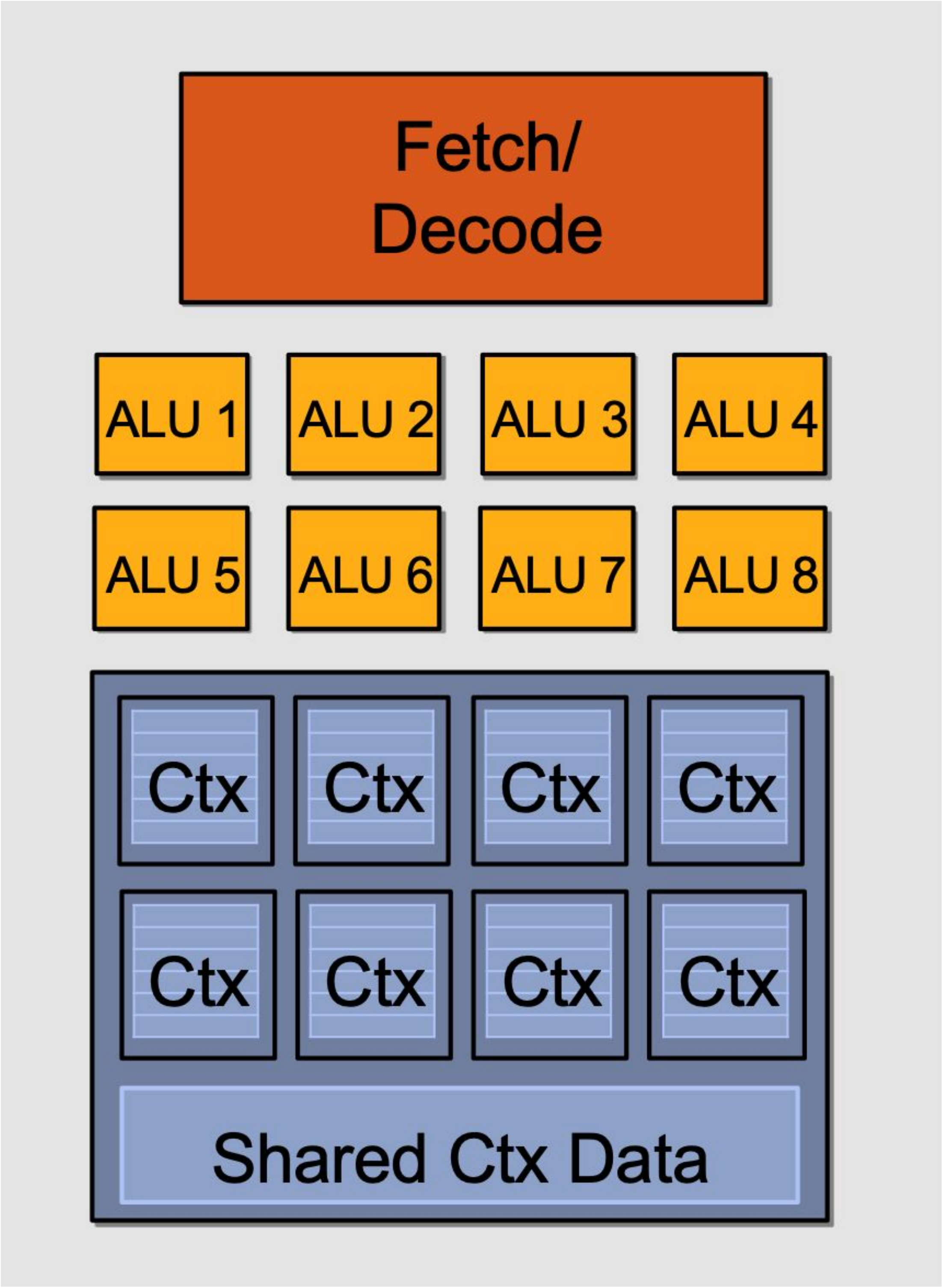
Sixteen Cores





16 cores = 16 simultaneous instruction streams

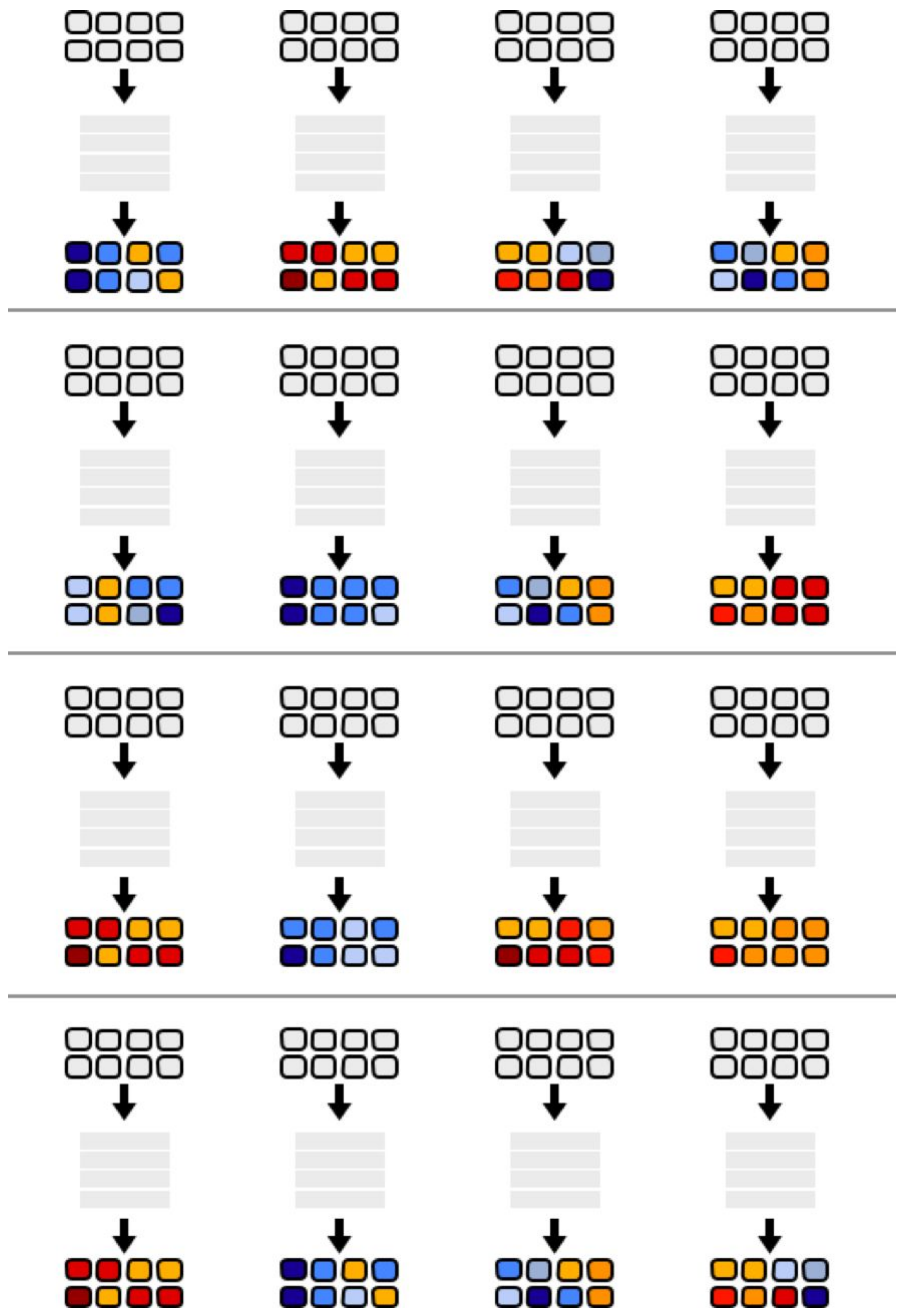
Instruction Stream Sharing



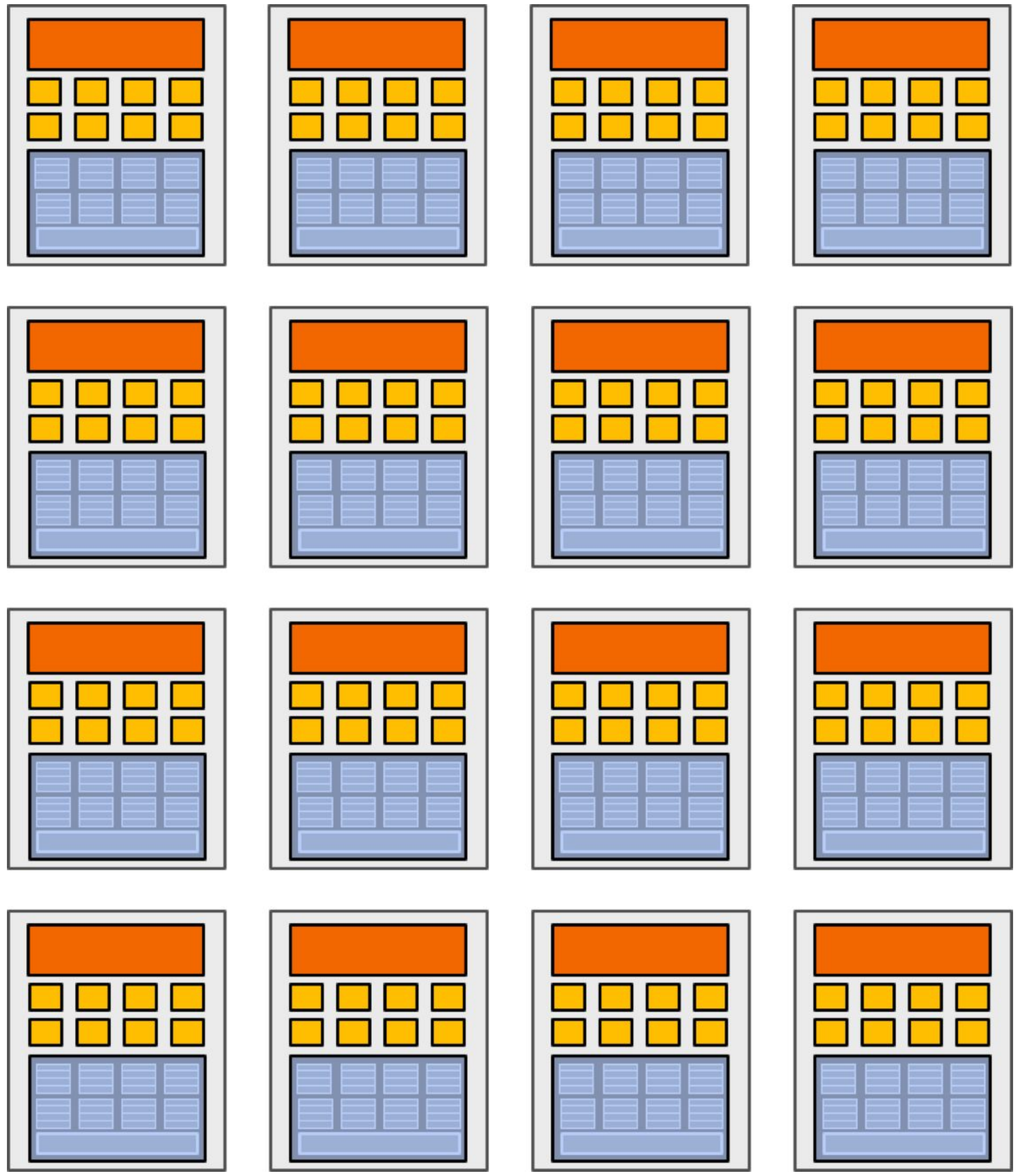
Idea #2:
Amortize cost/complexity of managing an instruction stream across many ALUs

SIMD processing!

128 Elements in Parallel

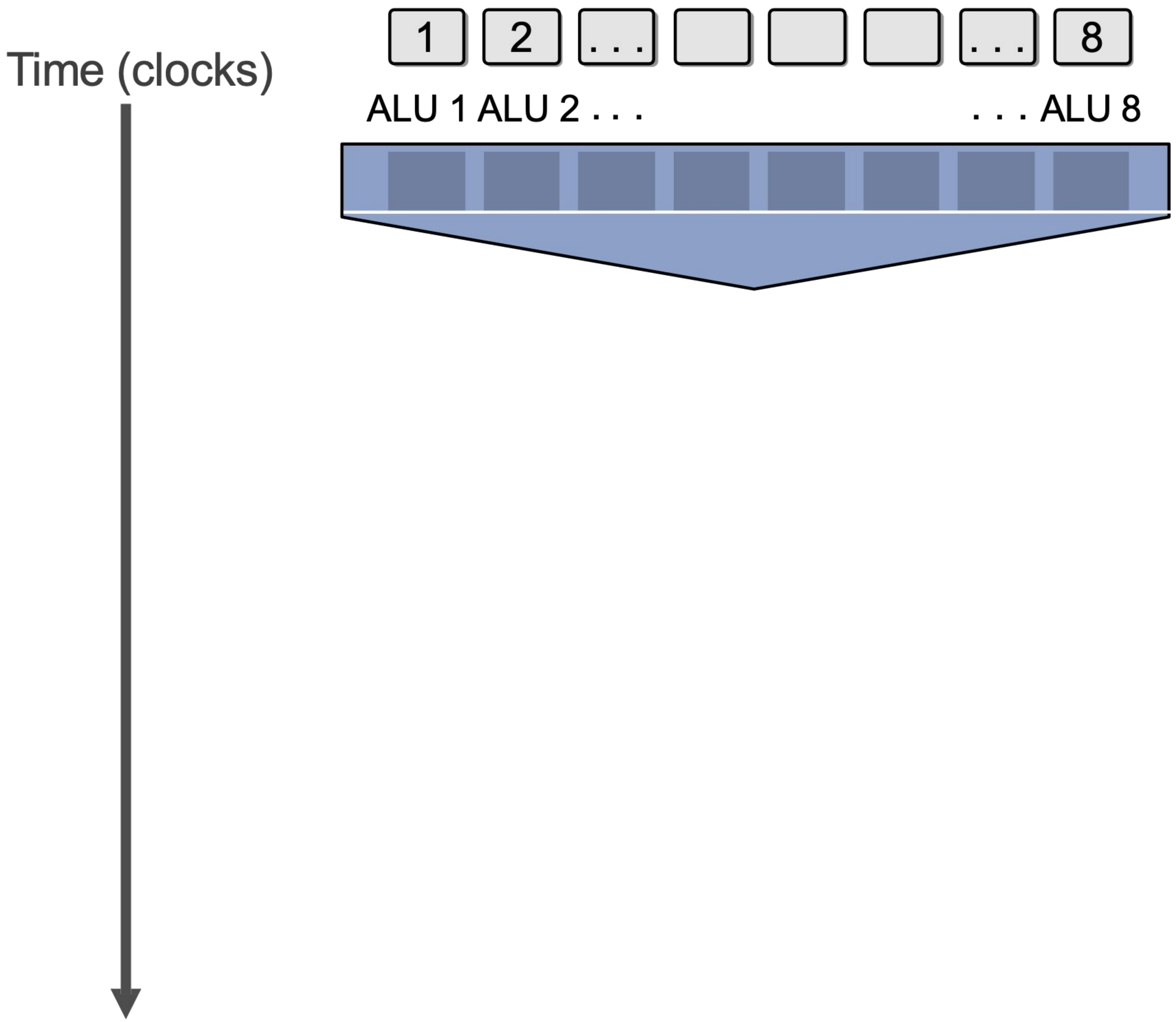


16 cores x 8 ALUs/core = 128 ALUs



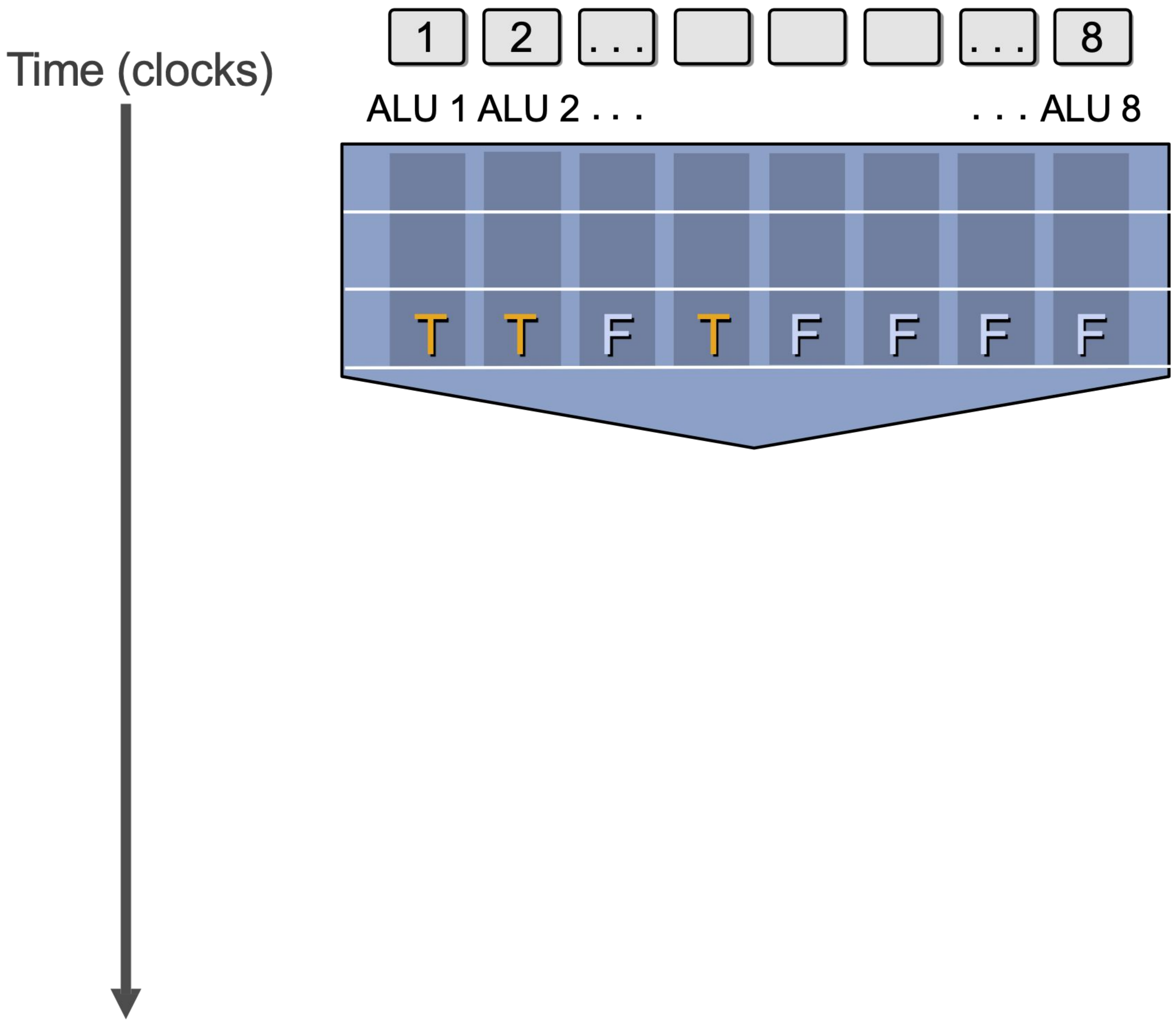
16 cores = 16 simultaneous instruction streams

What about Branches?



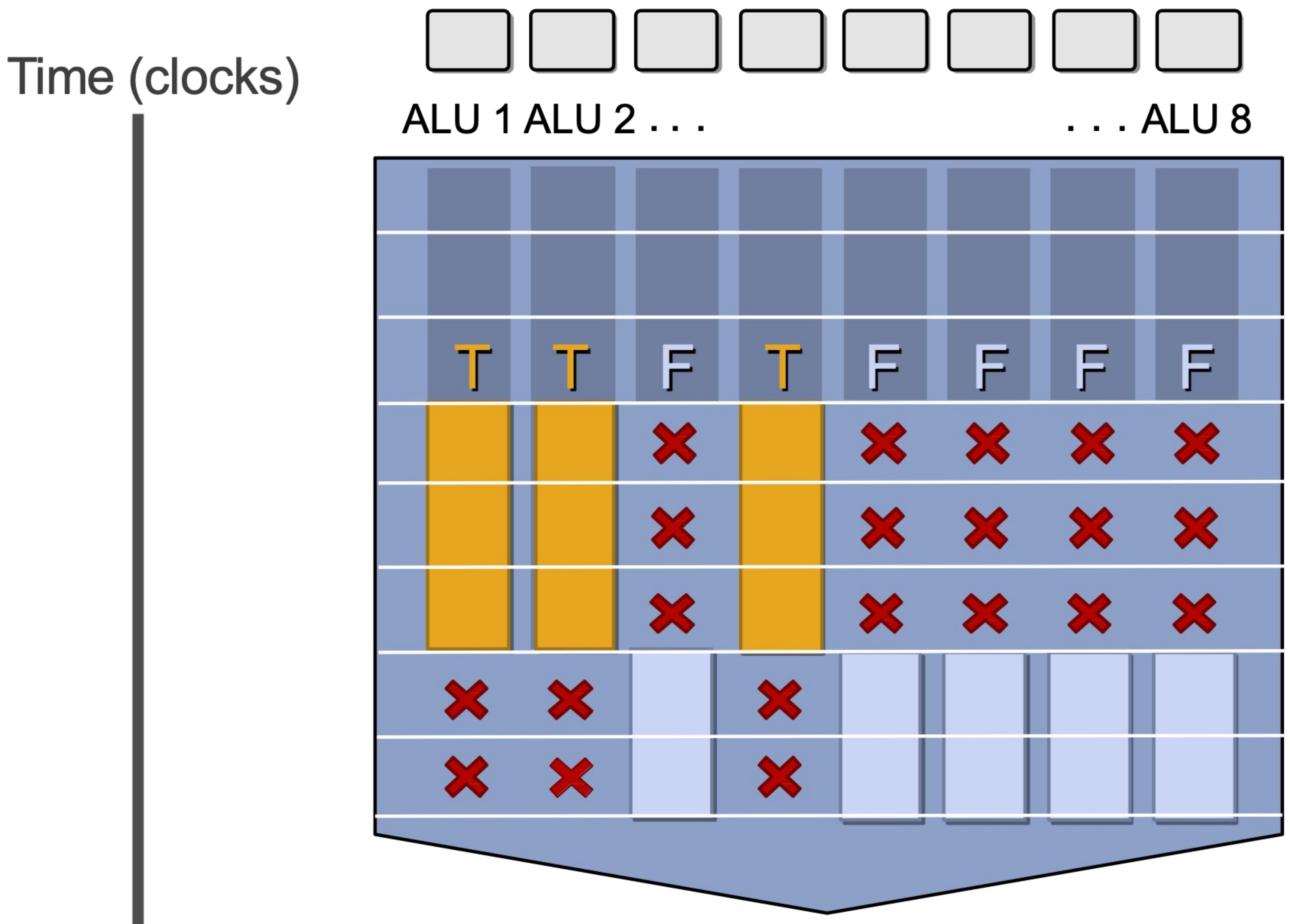
```
<unconditional shader code>
if (x > 0) {
      y = pow(x, exp);
      y *= Ks;
      refl = y + Ka;
} else {
      x = 0;
      refl = Ka;
}
<resume unconditional shader code>
```

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>
```

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

Terminology

- Instruction stream coherence (“coherent execution”)
  - Same instruction sequence applies to all elements operated upon simultaneously
  - Coherent execution is necessary for efficient use of SIMD processing resources
  - Coherent execution IS NOT necessary for efficient parallelization across cores, since each core has the capability to fetch/decode a different instruction stream
- “Divergent” execution
  - A lack of instruction stream coherence

SIMD Execution on Modern GPUs

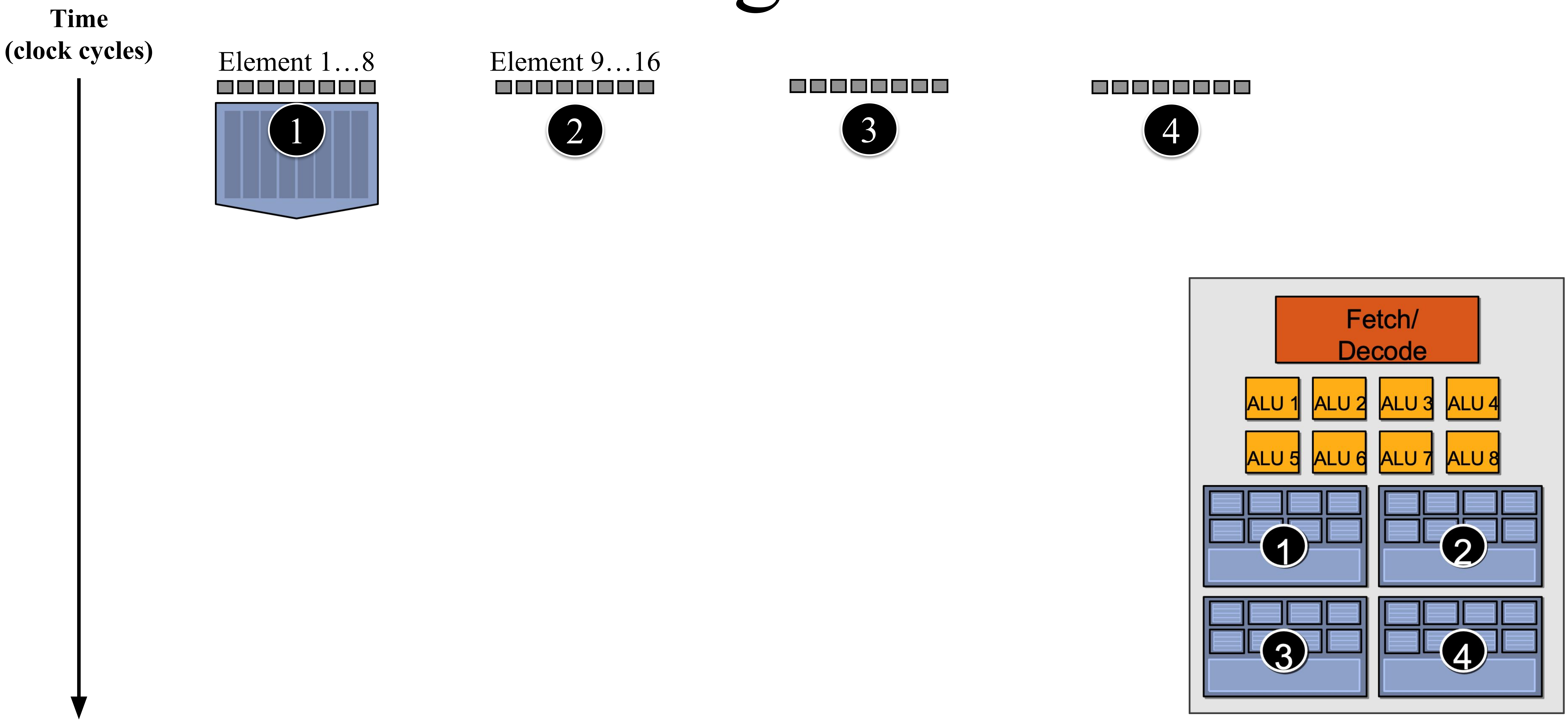
- “Implicit SIMD”
  - Compiler generates a scalar binary (scalar as opposed to vector instructions)
  - But N instances of the program are *always running* together on the processor
    i.e., execute(my_function, N) // execute my_function N times
  - Hardware (not compiler) is responsible for simultaneously executing the same instruction on different data in SIMD ALUs
- SIMD width in practice
  - *32* on NVIDIA GPUs (a warp of threads) and *64* on AMD GPUs (wavefront)
  - Divergence can be a big issue (poorly written code might execute at 1/32 the peak capability of the machine!)

Dealing with Stalls on In-order Cores

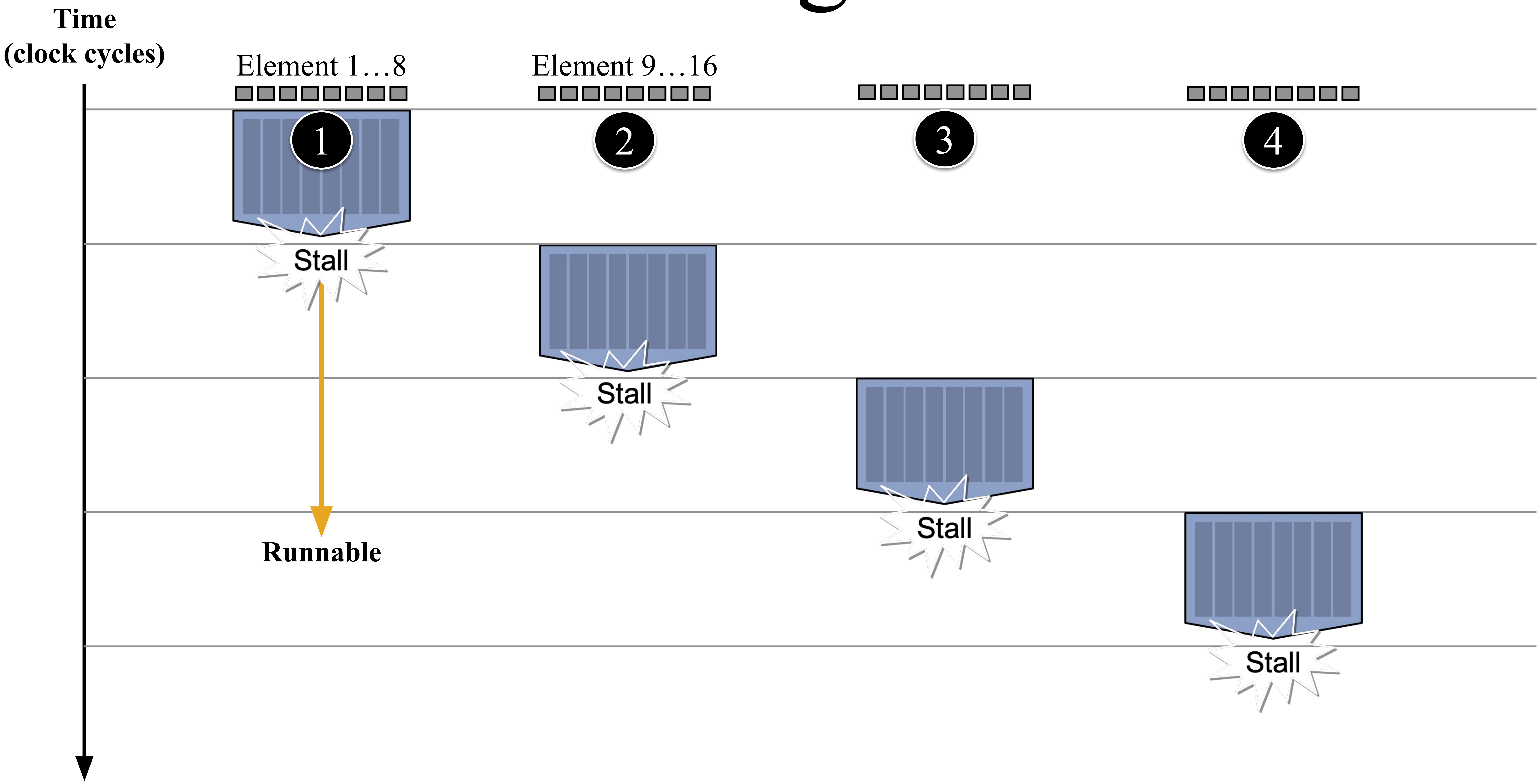
- Stalls occur when a core cannot run the next instruction because of a dependency on a previous long-latency operation
- We've removed fancy logic that helps avoid stalls
  - No more out-of-order execution to exploit instruction-level parallelism (ILP)
  - Traditional cache doesn't always help since a lot of workloads are streaming data
- But, we have a LOT of parallel work…

Idea #3: Interleave processing of many warps on a single core to avoid stalls caused by high-latency operations

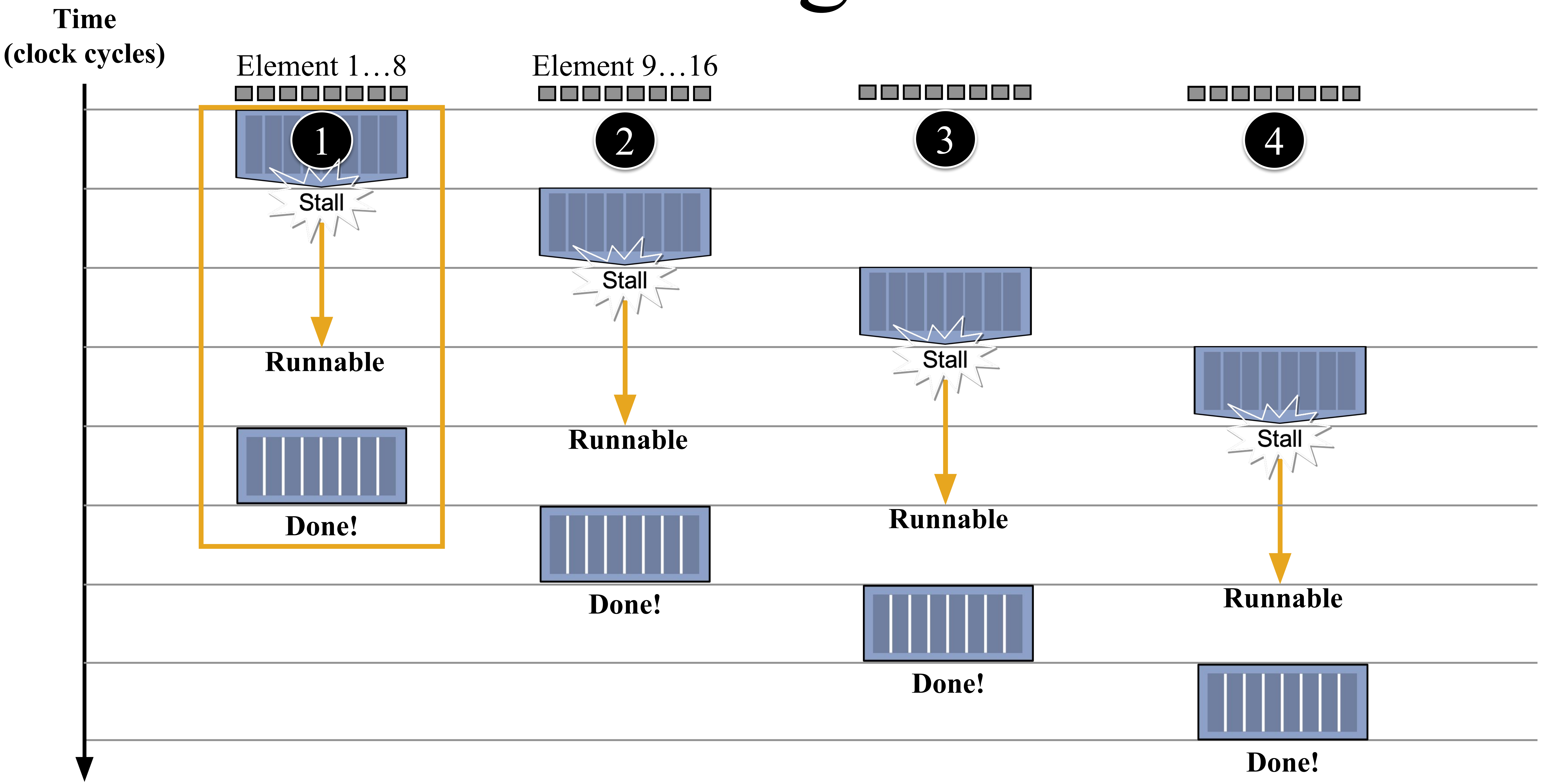
Hiding Stalls



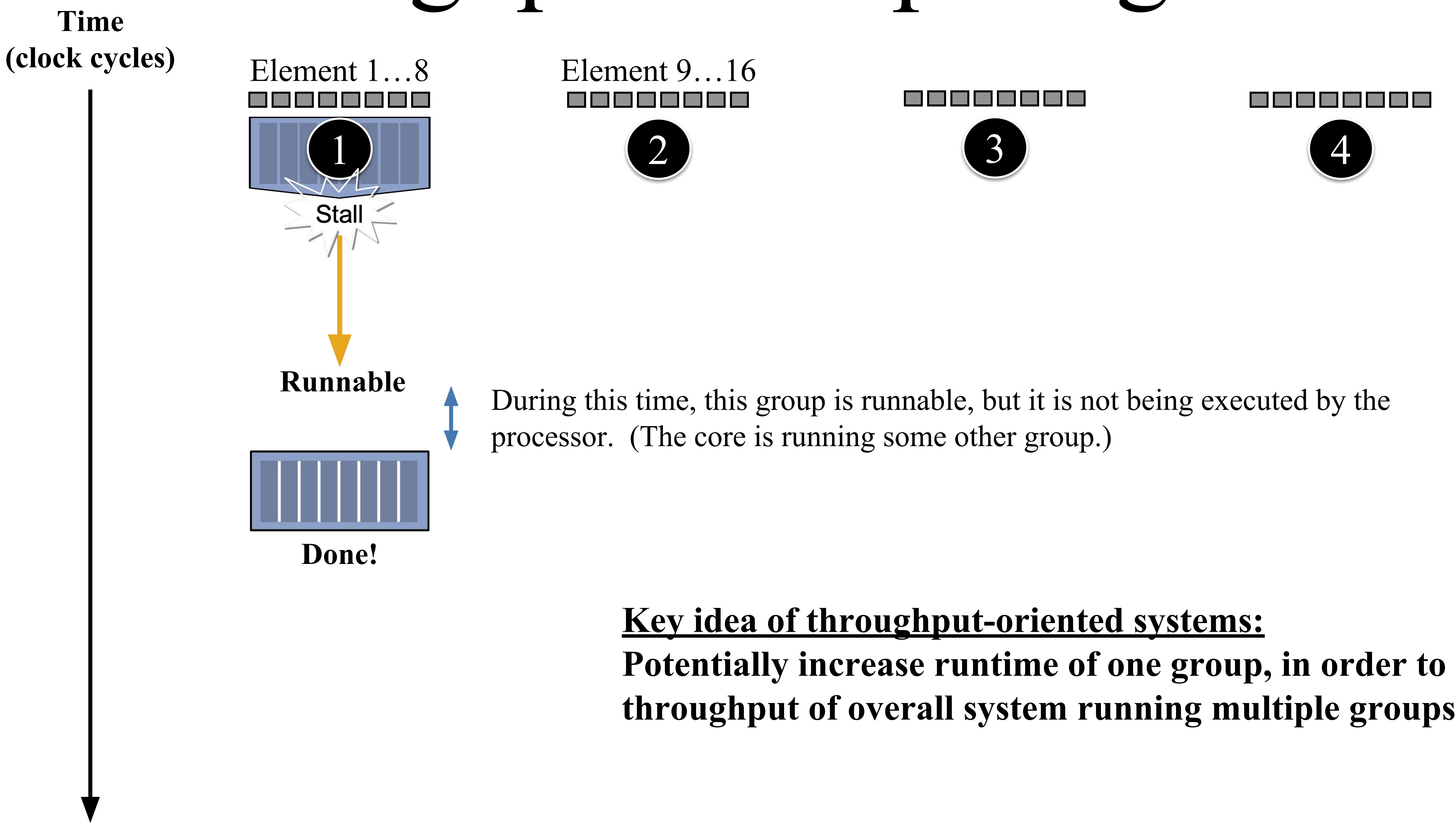
Hiding Stalls



Hiding Stalls



Throughput Computing Trade-off

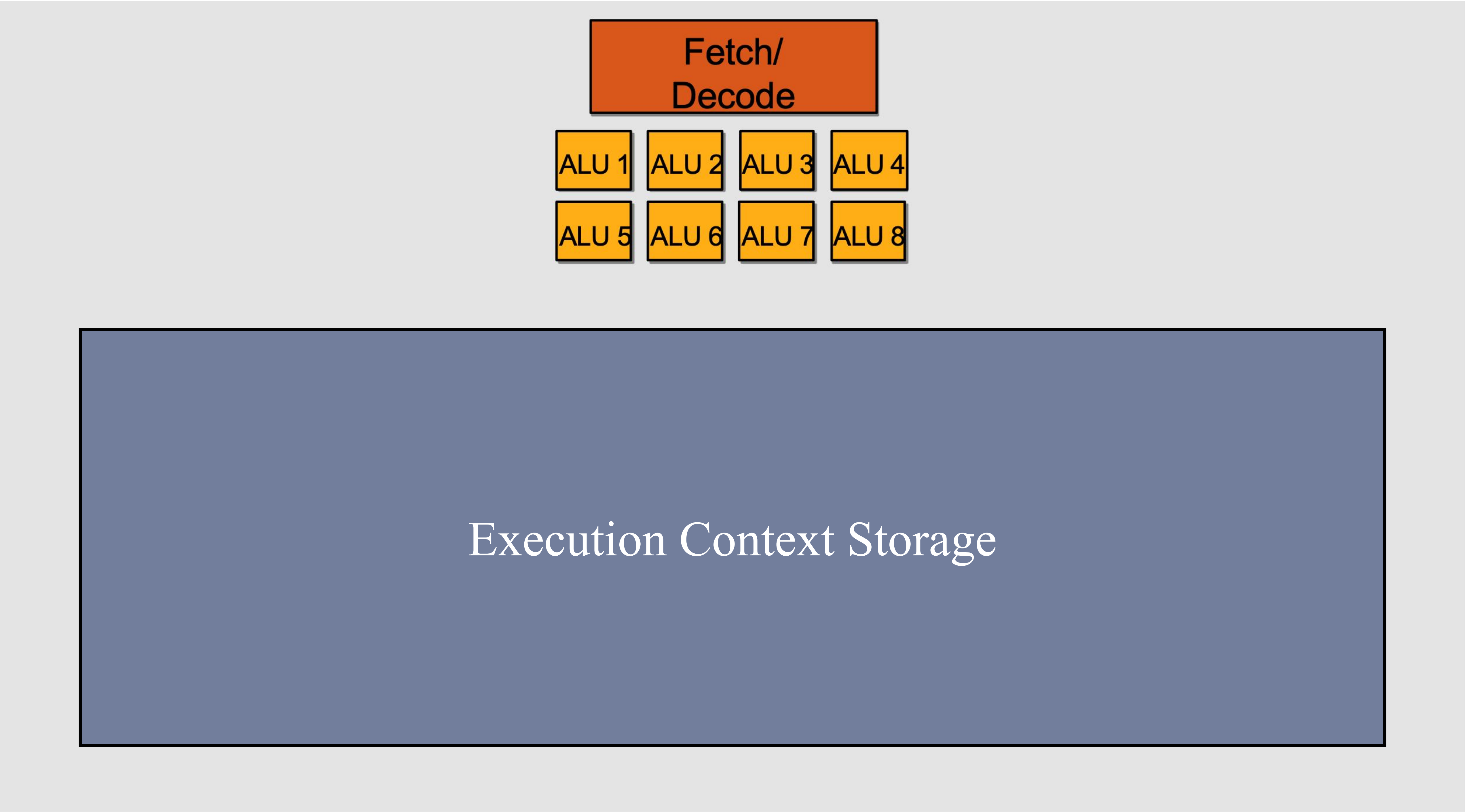


During this time, this group is runnable, but it is not being executed by the processor. (The core is running some other group.)

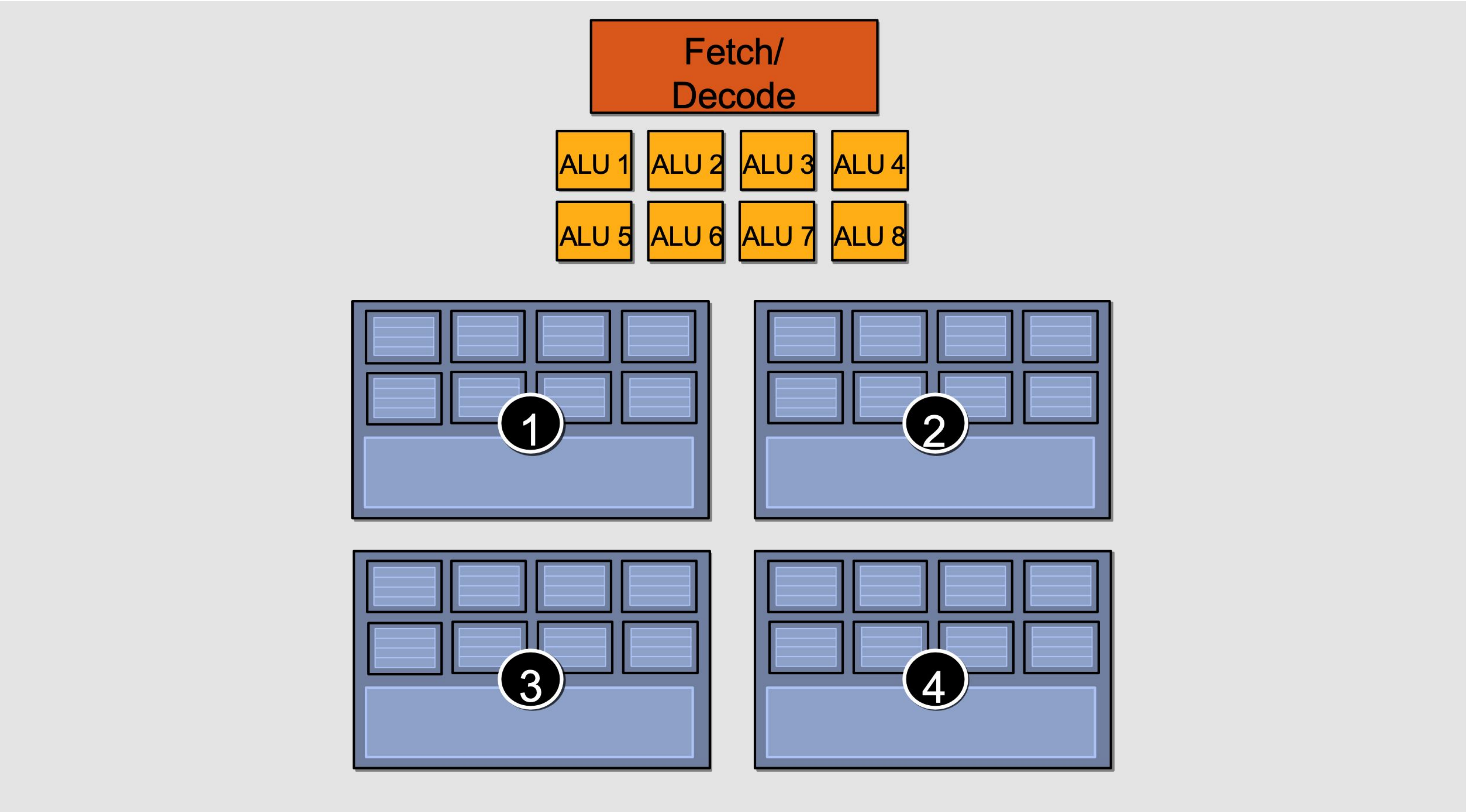
<u>**Key idea of throughput-oriented systems:**</u>
Potentially increase runtime of one group, in order to increase throughput of overall system running multiple groups.

Storing Execution Contexts

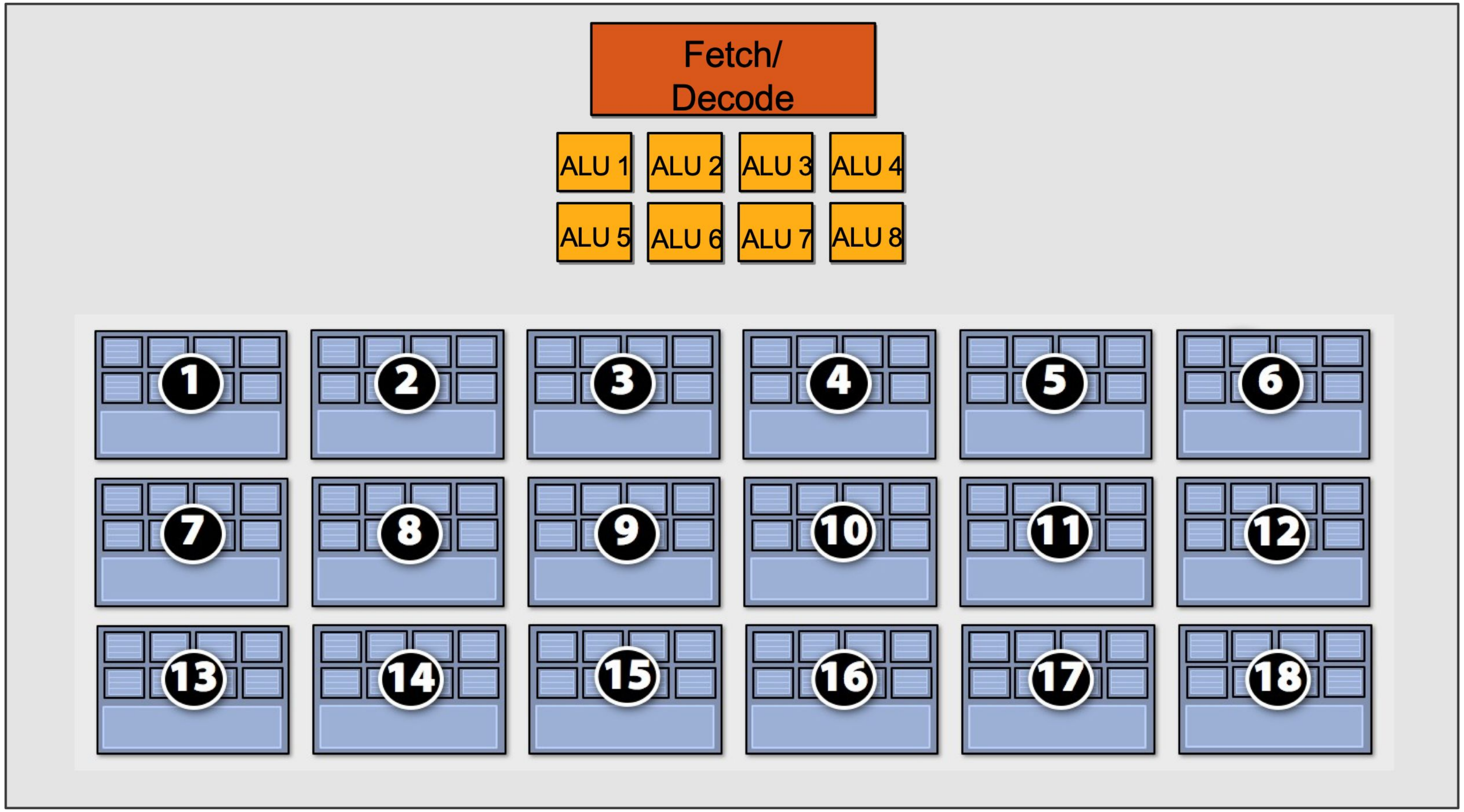
- Consider on-chip storage of execution contexts <u>a finite resource</u>
- Resource consumption of each thread group is <u>program-dependent</u>



Four Large Contexts (*Low* Latency Hiding)



Eighteen Small Contexts (*High* Latency Hiding)



Summary: Three Key Ideas

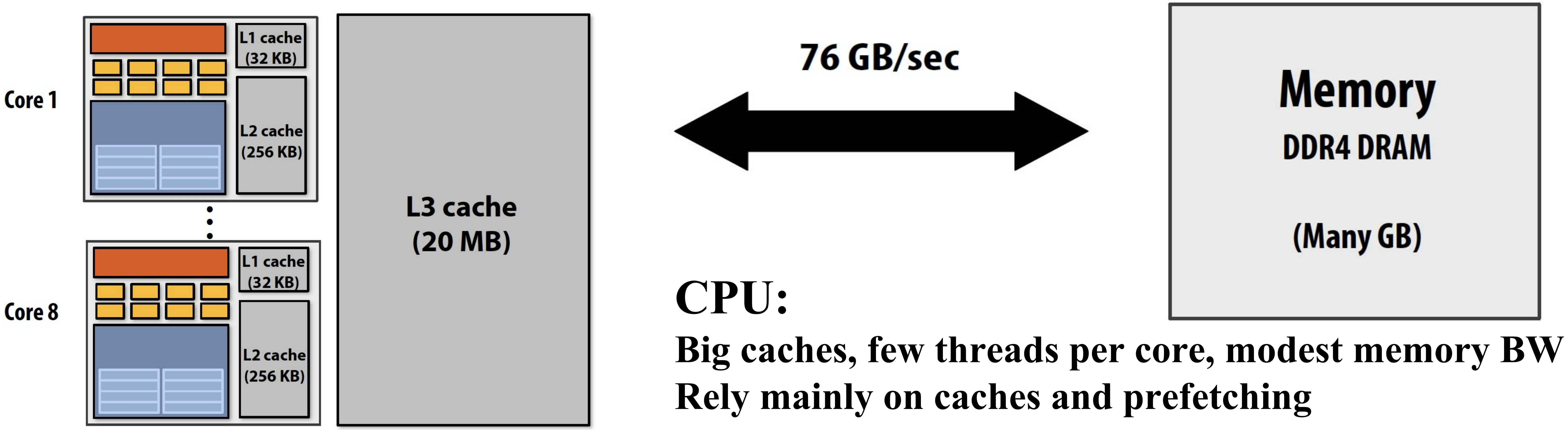
1. Use many “slimmed down cores” to run in parallel

2. Pack cores full of ALUs (by sharing instruction stream on multiple data)

3. Avoid latency stalls by interleaving execution of many groups of threads

    - When one group stalls, work on another group

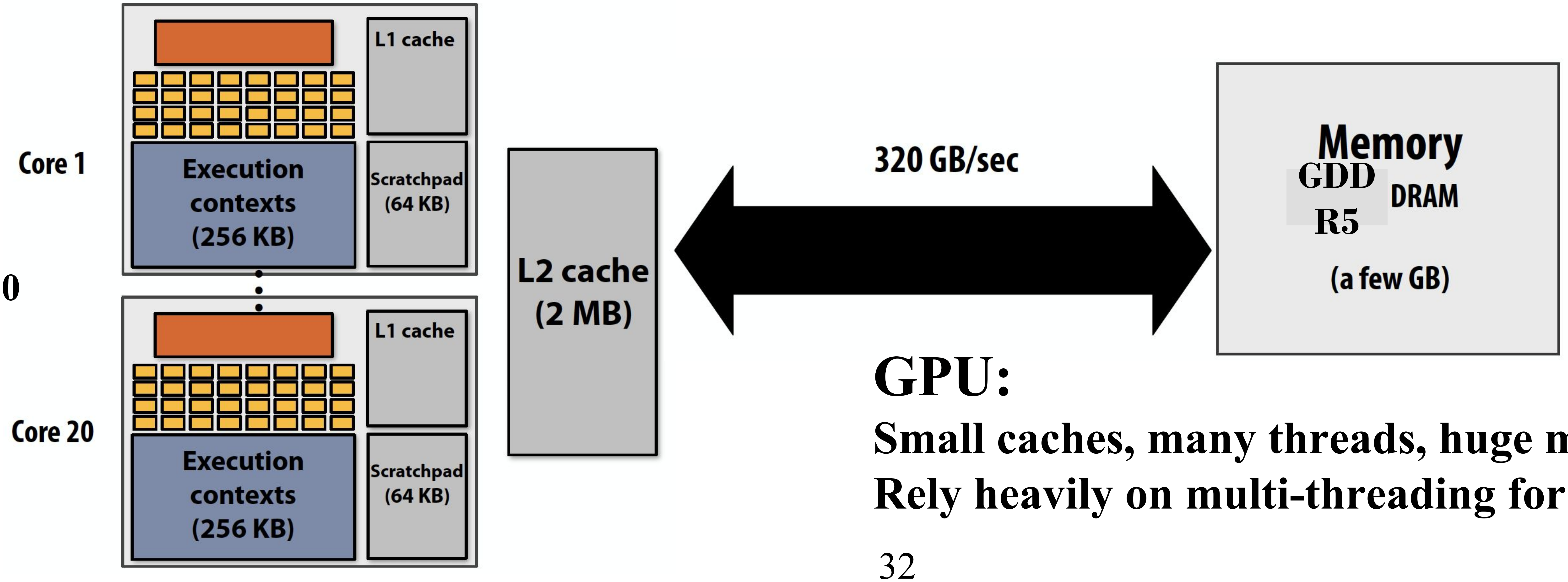
CPU v.s. GPU Memory Hierarchies



CPU:

Big caches, few threads per core, modest memory BW

Rely mainly on caches and prefetching



NVIDIA GTX 1080
(Pascal)

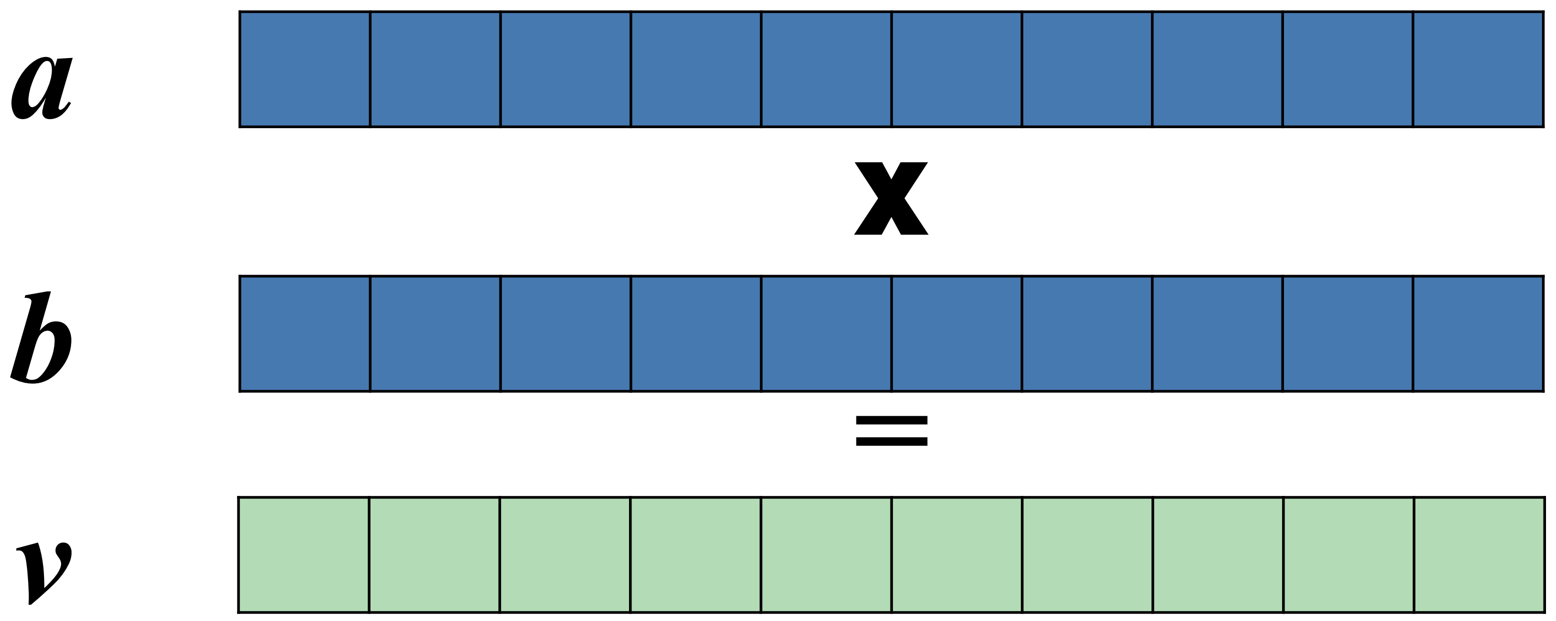
GPU:

Small caches, many threads, huge memory BW

Rely heavily on multi-threading for performance

Thought Experiment

- Consider element-wise multiplication of two vectors ***a*** and ***b***
- Assume vectors contain millions of elements
  - Load input ***a***[i]
  - Load input ***b***[i]
  - Compute ***a***[i] **x** ***b***[i]
  - Store result into ***v***[i]



- Three memory operations (12 bytes) for every MUL
- NVIDIA GTX 1080 GPU can do 2560 MULs per clock (@ 1.6 GHz)
- Need ~45 TB/sec of bandwidth to keep functional units busy (only have 320 GB/sec)

<1% GPU efficiency… but 4.2x faster than eight-core CPU in lab!

(3.2 GHz Xeon E5v4 eight-core CPU connected to 76 GB/sec memory bus will exhibit ~3% efficiency on this computation)

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

Performant parallel programs will:

- Organize computation to fetch data from memory less often
  - Reuse data previously loaded by the same thread
  - Share data across threads through scratchpad (inter-thread cooperation)
  - Access contiguous memory within the same warp (hardware managed memory coalescing)
- Request data less often (instead, do more arithmetic: it’s “free”)
  - Useful term: “arithmetic intensity” — ratio of math operations to data access operations in an instruction stream
  - Main point: programs must have high arithmetic intensity to utilize modern processors efficiently

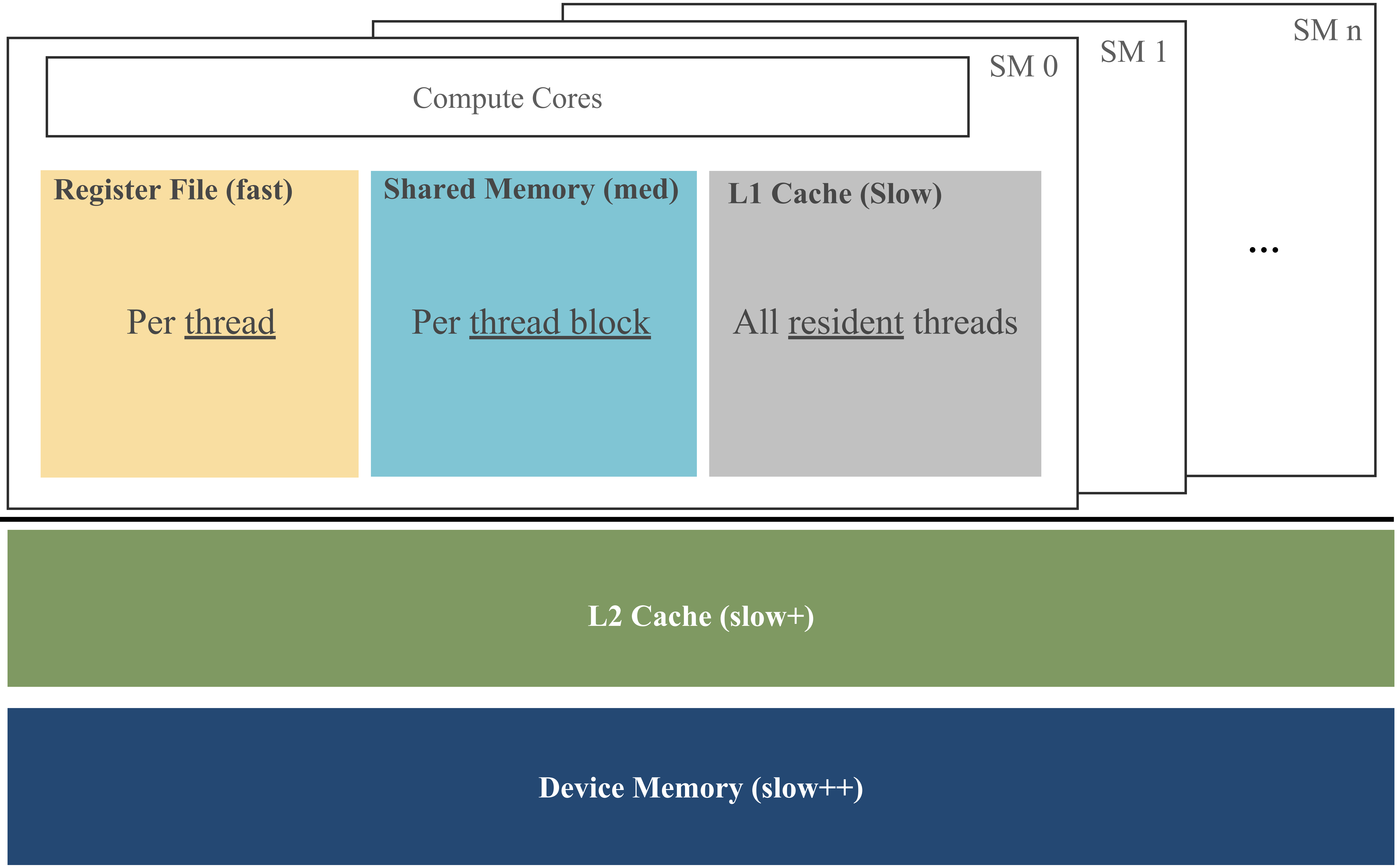
Memory Spaces in GPU

On-chip:

- Register file
  - Usage determined by compiler
  - Spills go to local memory
- Shared memory, i.e. scratchpad
  - Programmer managed
  - Bank conflicts
- L1 cache

Off-chip:

- L2 cache
  - Bandwidth filter for DRAM rather than reducing latency as in CPUs
- Device memory (DRAM)
  - Several spaces: global memory, texture memory, local memory
  - Different spaces have different caching policies



Modern GPU Architecture (Volta 2017)

21B transistors

815 mm^2

80 SM

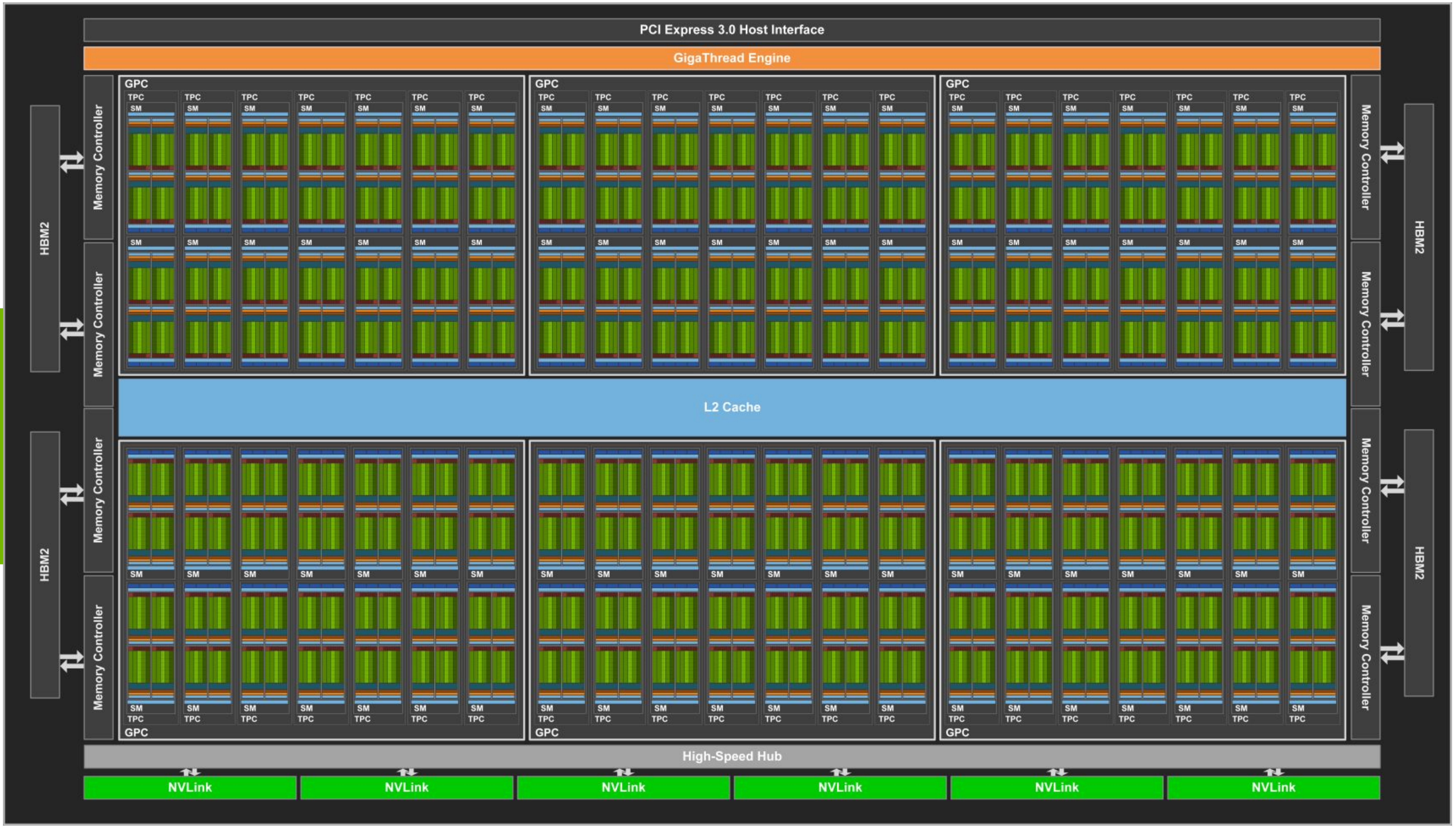
5120 CUDA Cores

640 Tensor Cores

16/32 GB HBM2

900 GB/s HBM2

300 GB/s NVLink



*full GV100 chip contains 84 SMs

NVIDIA

Review #7

GPUs and the Future of Parallel Computing

Steve Keckler et al., *IEEE Micro 2011*

Due Nov. 11

CSC 2224: Parallel Computer Architecture and Programming GPU Architecture: Introduction

Prof. Gennady Pekhimenko

University of Toronto

Fall 2019

The content of this lecture is adapted from the slides of Kayvon Fatahalian (Stanford), Olivier Giroux and Luke Durant (Nvidia), Tor Aamodt (UBC) and Edited by: Serina Tan