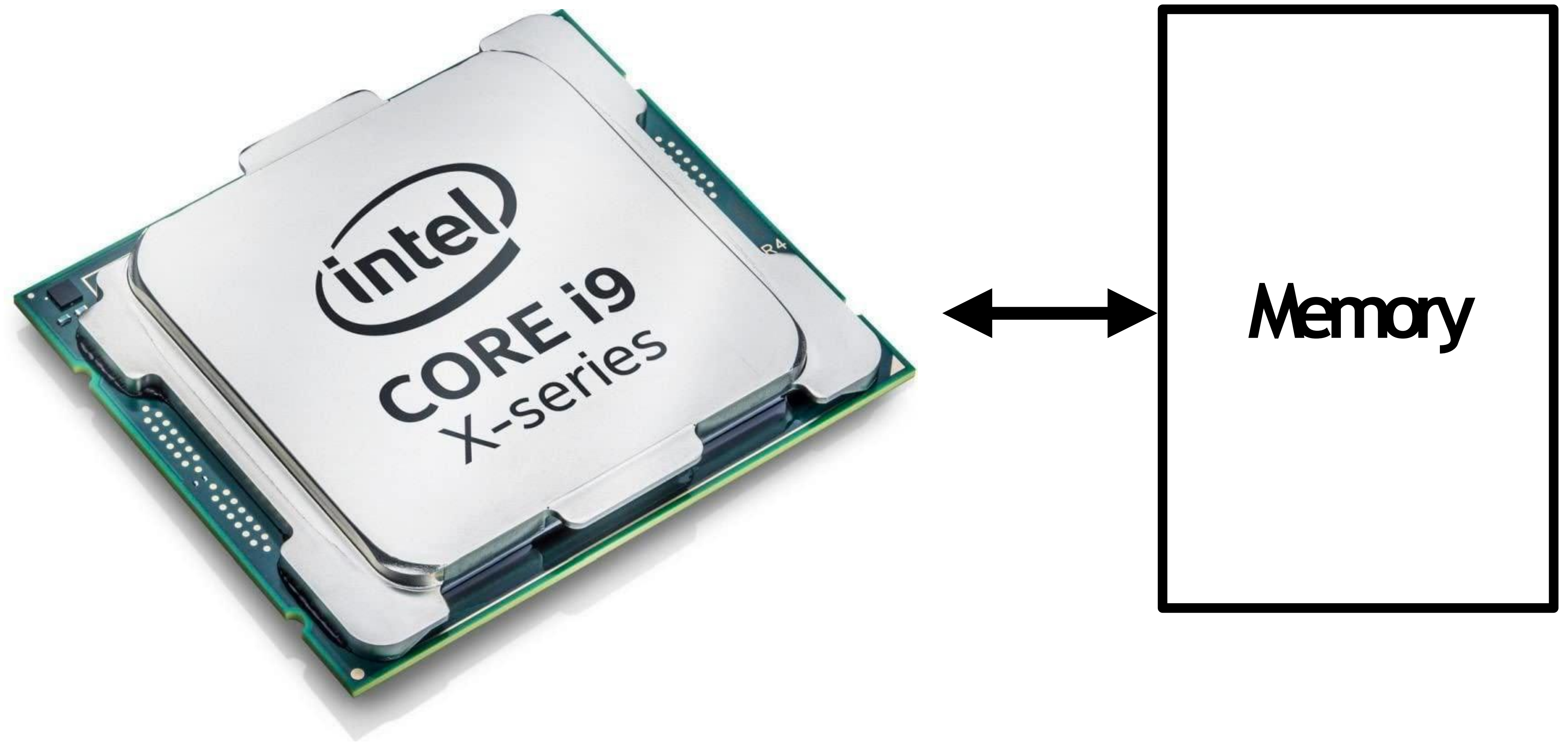
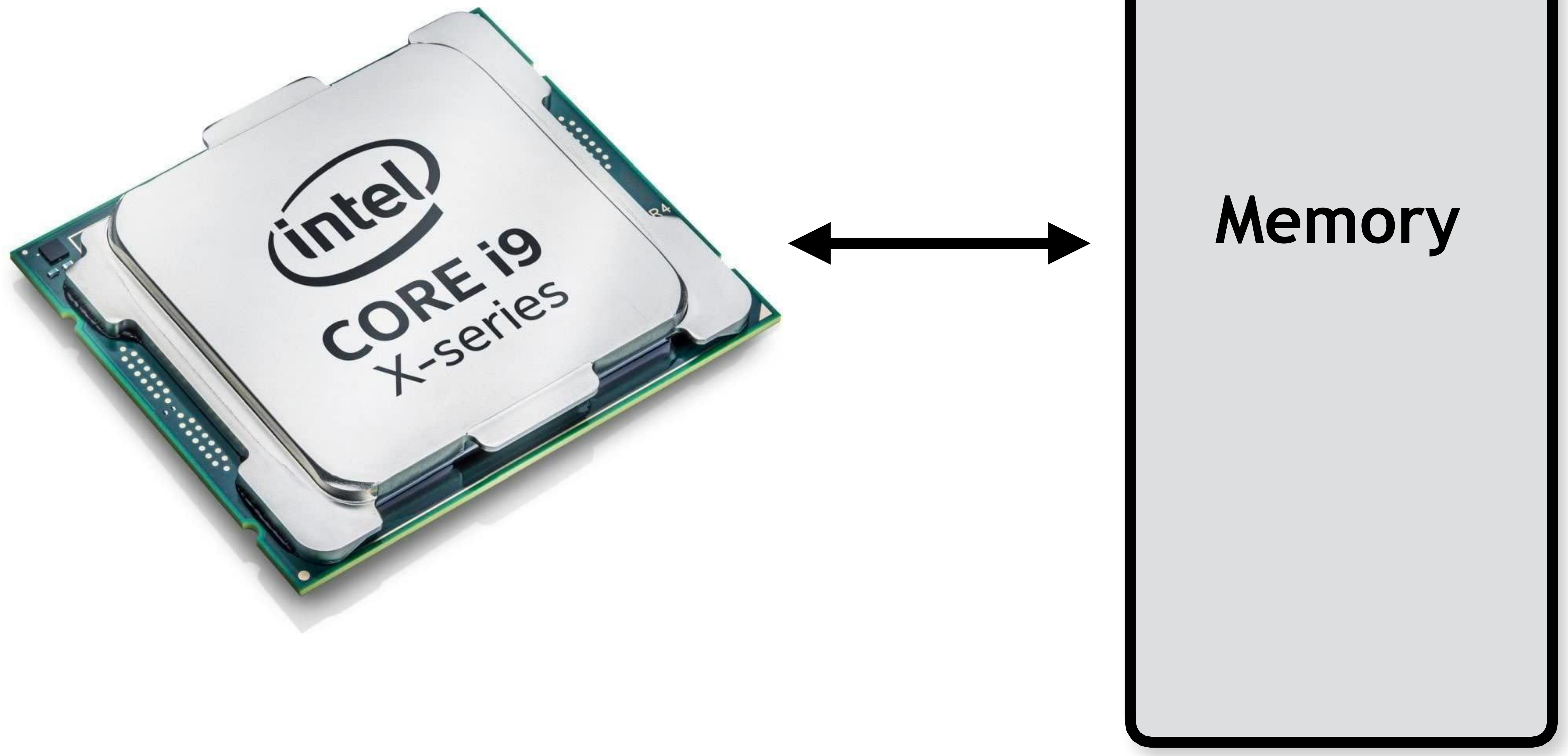
Part 2: accessing memory



What is memory?



A program's memory address space

- A computer's memory is organized as a array of bytes

- Each byte is identified by its "address" in memory (its position in this array)

  (Today we'll assume memory is byte-addressable)

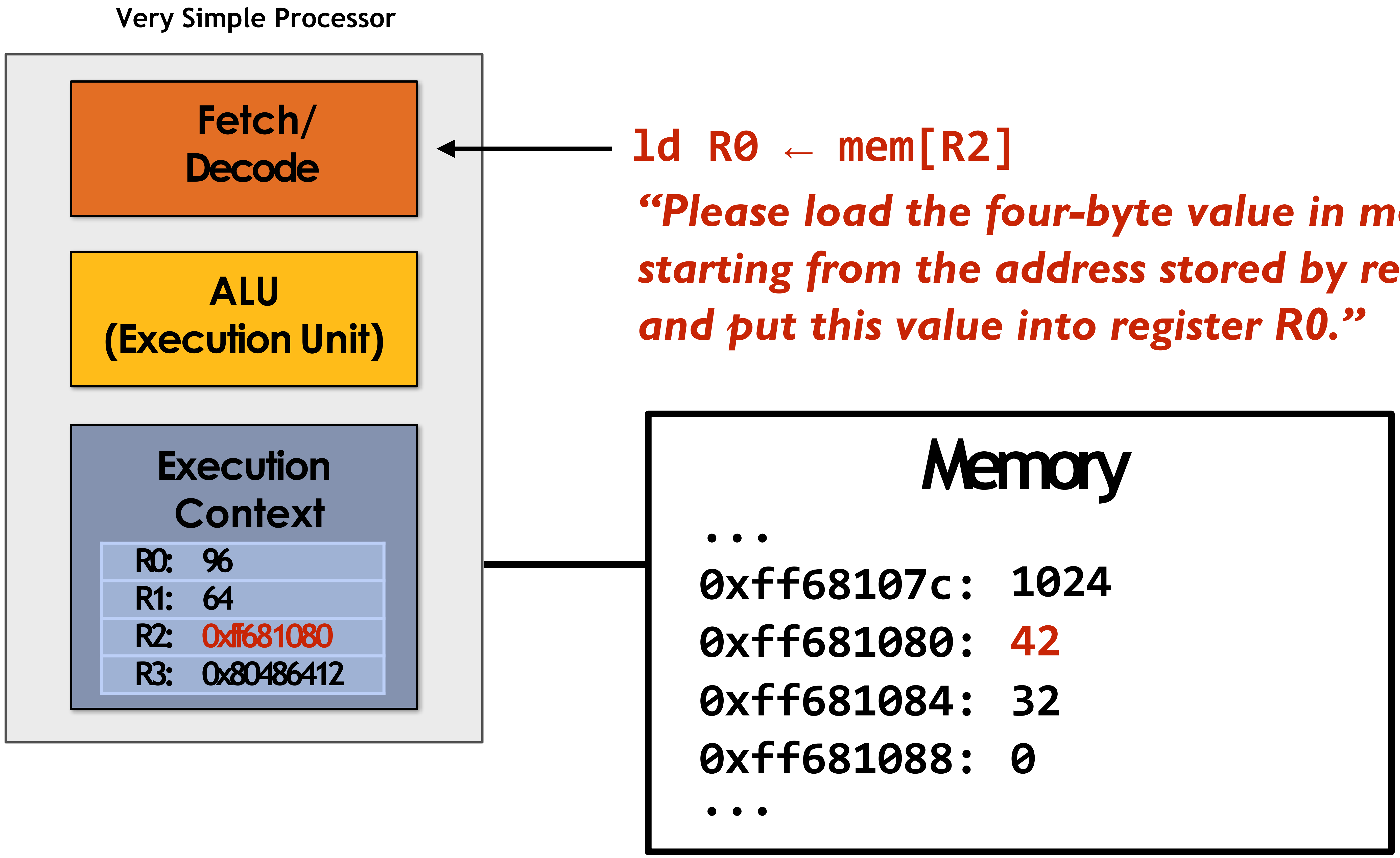
"The byte stored at address 0x8 has the value 32."

"The byte stored at address 0x10 (16) has the value 128."

In the illustration on the right, the program's memory address space is 32 bytes in size
(so valid addresses range from 0x0 to 0x1F)

Address	Value
0x0	16
0x1	255
0x2	14
0x3	0
0x4	0
0x5	0
0x6	6
0x7	0
0x8	32
0x9	48
0xA	255
0xB	255
0xC	255
0xD	0
0xE	0
0xF	0
0x10	128
⋮	⋮
0x1F	0

Load: an instruction for accessing the contents of memory

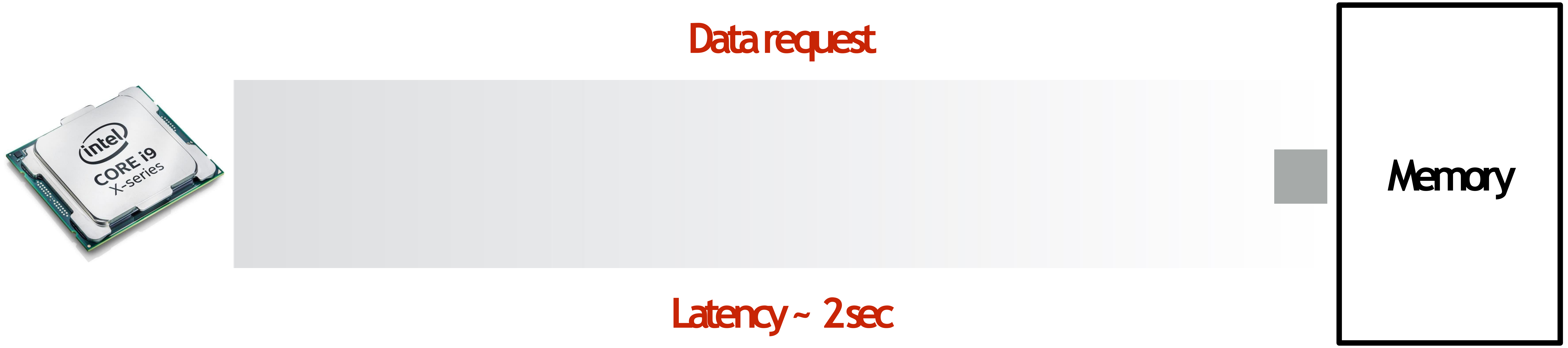


`ld R0 ← mem[R2]`

"Please load the four-byte value in memory starting from the address stored by register R2 and put this value into register R0."

Terminology

- Memory access latency
  - The amount of time it takes the memory system to provide data to the processor
  - Example: 100 clock cycles, 100 nsec



Stalls

- A processor "stalls" when it cannot run the next instruction in an instruction stream because of a dependency on a previous instruction that is not yet complete.

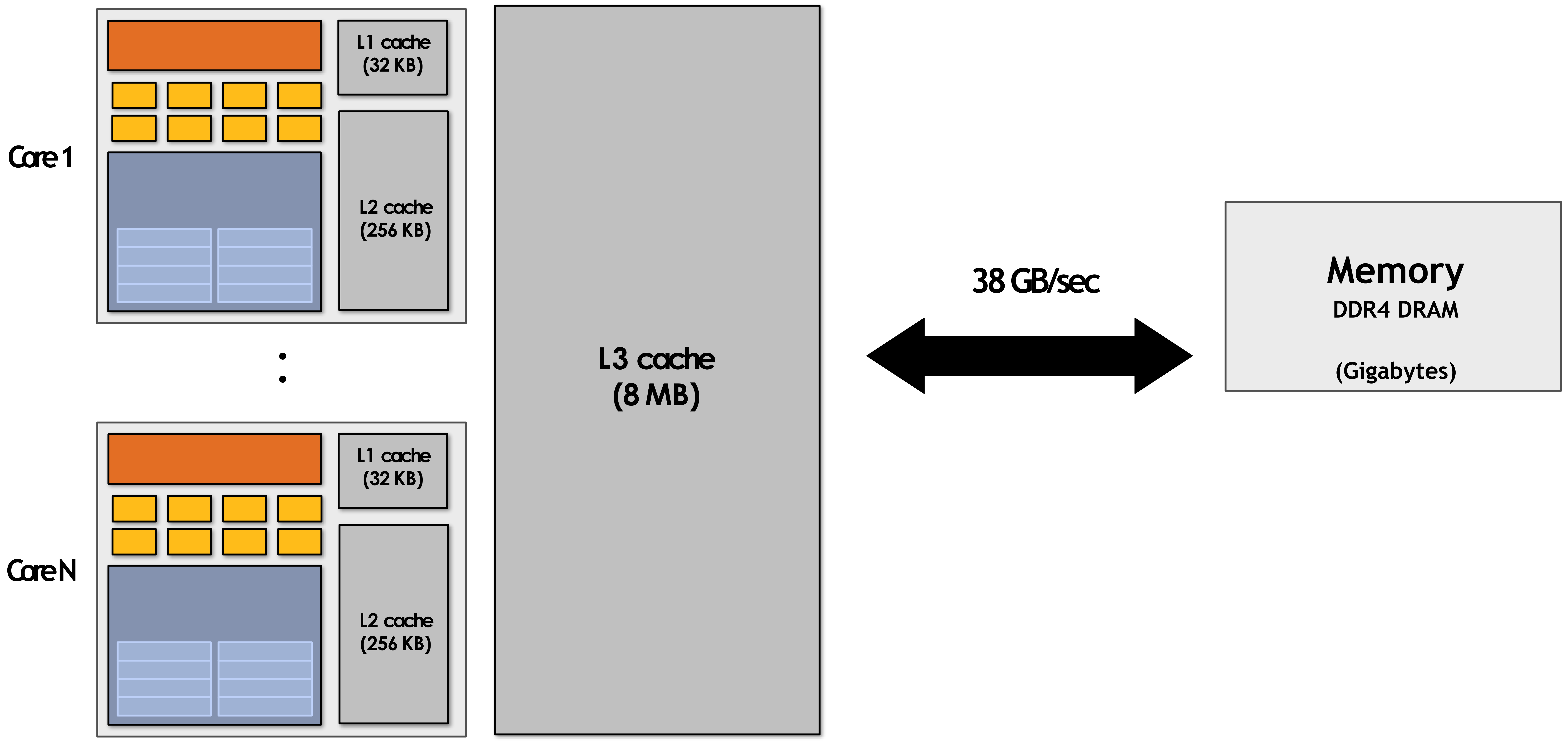
- Accessing memory is a major source of stalls

```
ld  r0  mem[r2]
ld  r1  mem[r3]
add r0, r0, r1
```

Dependency: cannot execute 'add' instruction until data from mem[r2] and mem[r3] have been loaded from memory

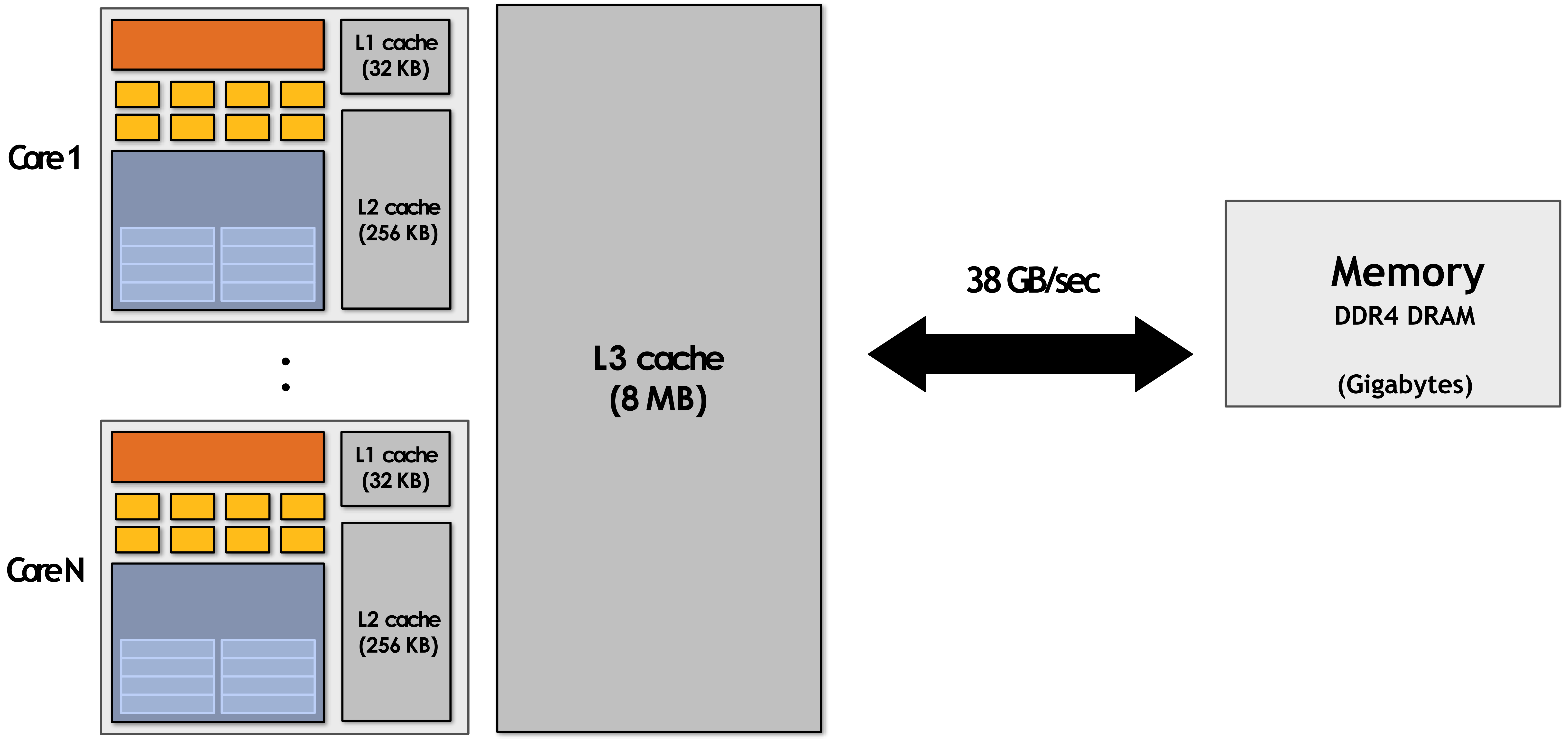
- Memory access times ~ 100's of cycles
  - Memory "access time" is a measure of latency

Why do modern processors have data caches?



Caches reduce length of stalls (reduce memory access latency)

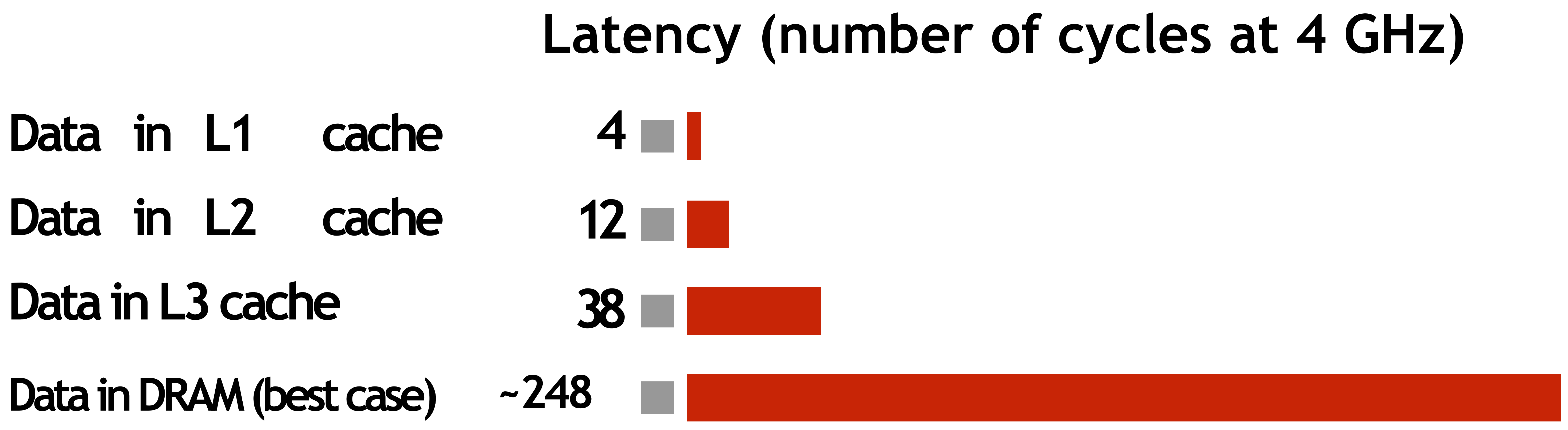
Processors run efficiently when data is resident in caches Caches reduce memory access latency *



* Caches also provide high bandwidth data transfer to CPU

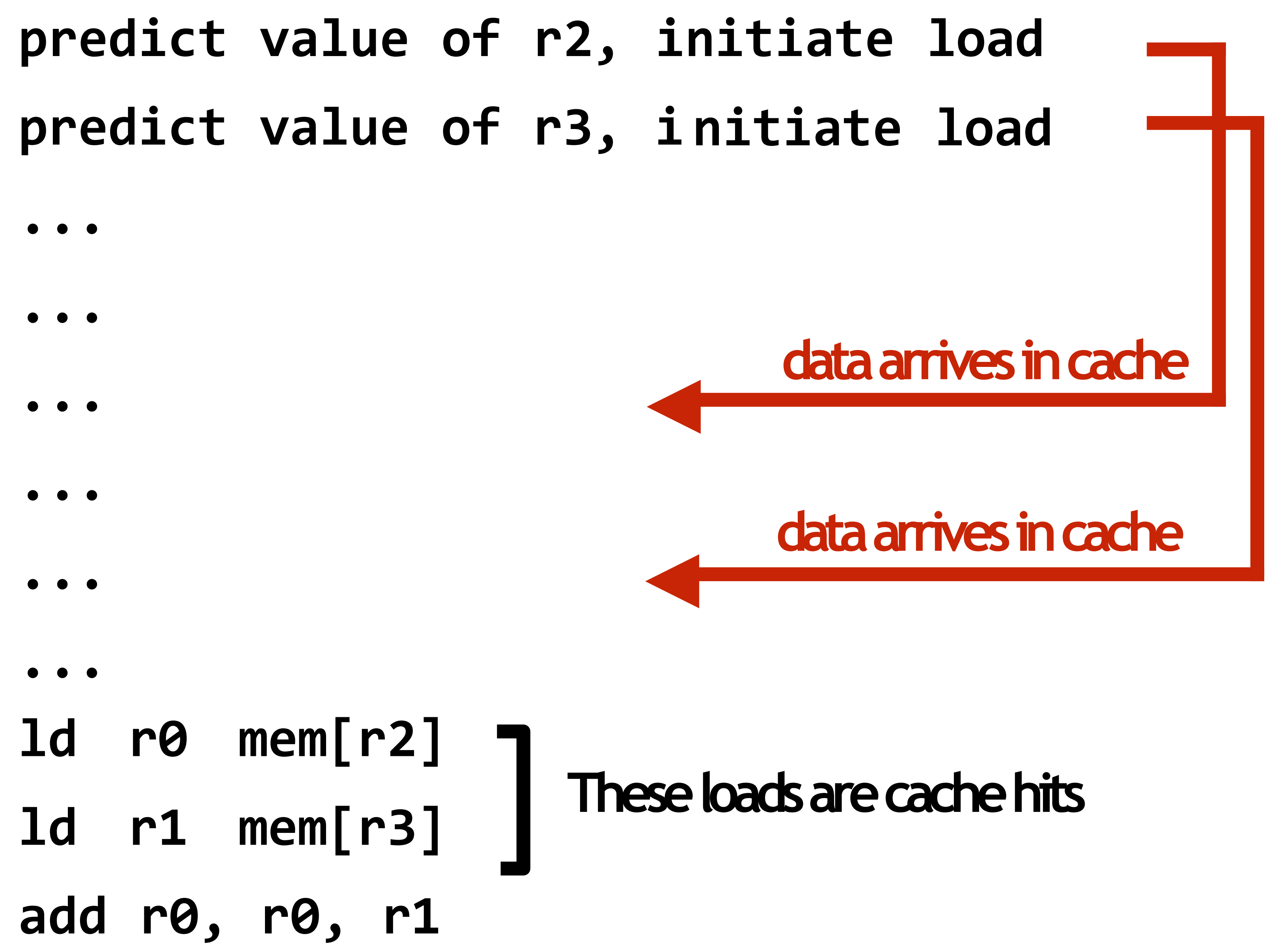
Data access times

(Kaby Lake CPU)



Prefetching reduces stalls (<u>hides</u> latency)

- Many modern CPUs have logic for predicting what data will be accessed in the future and “pre-fetching” this data into caches
  - Dynamically analyze program’s memory access patterns to make predictions
- Prefetching reduces stalls since data is resident in cache when accessed



```
predict value of r2, initiate load
predict value of r3, initiate load
...
...
...
...
...
...
ld  r0  mem[r2]
ld  r1  mem[r3]
add r0, r0, r1
```

Note: Prefetching can also reduce performance if the guess is wrong (consumes bandwidth, pollutes caches)

Cooking your favor meal…

Image credit: https://www.escoffier.edu/blog/food-entrepreneurship/culinary-side-hustles/

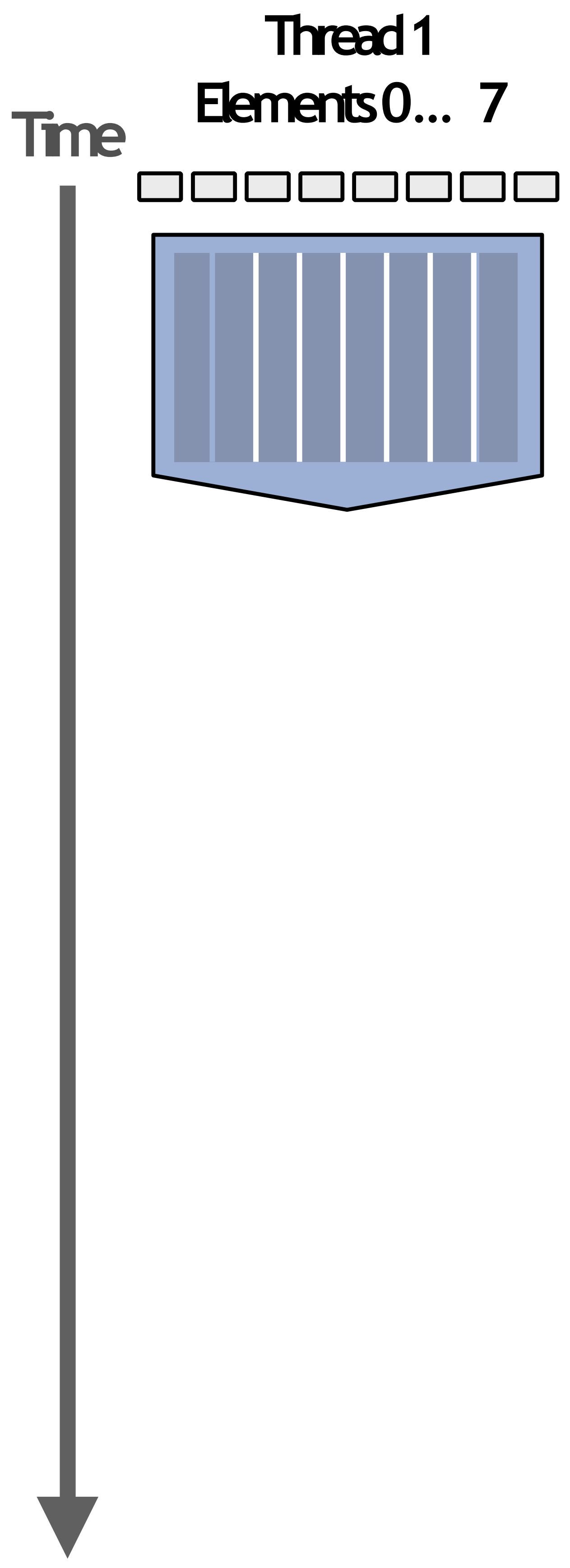
Doing your laundry…

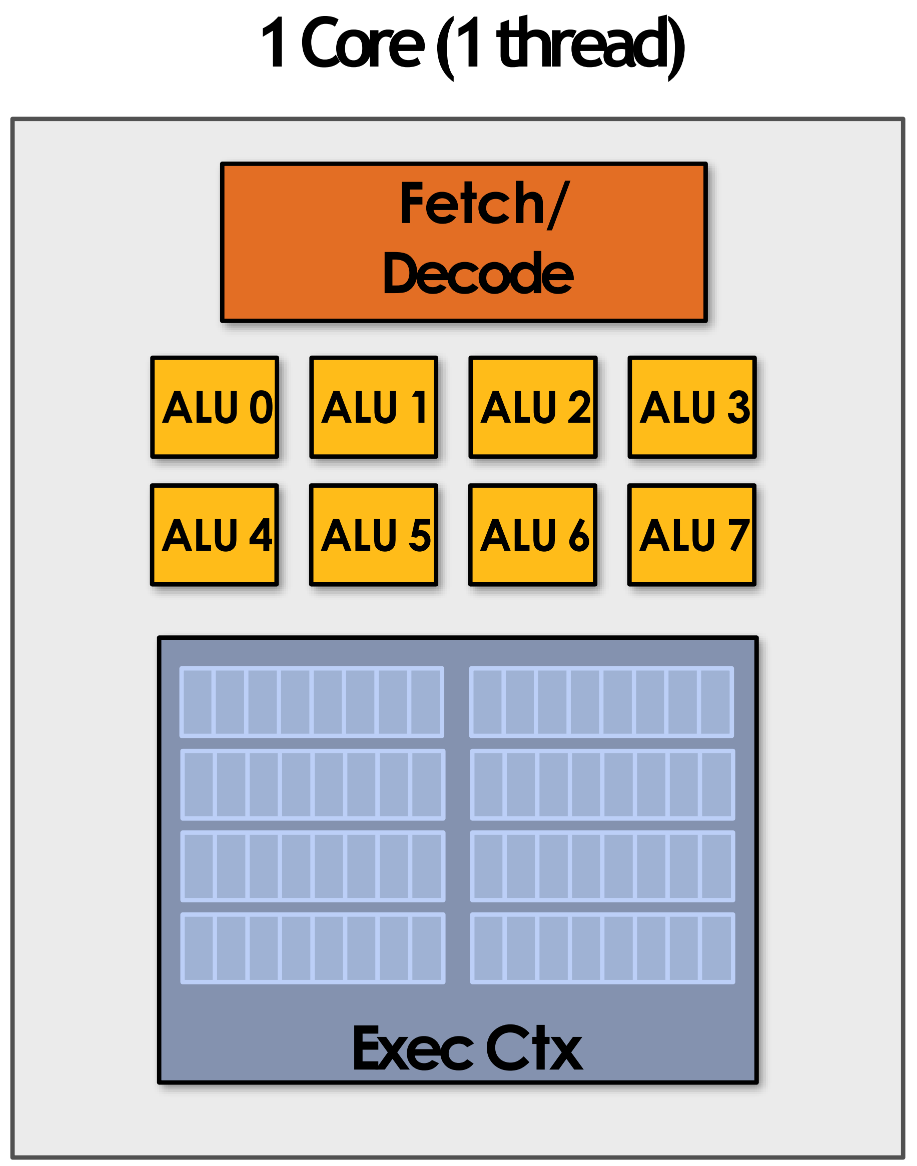
Credit: https://www.theodysseyonline.com/the-dos-and-donts-of-dorm-laundry

Multi-threading reduces stalls

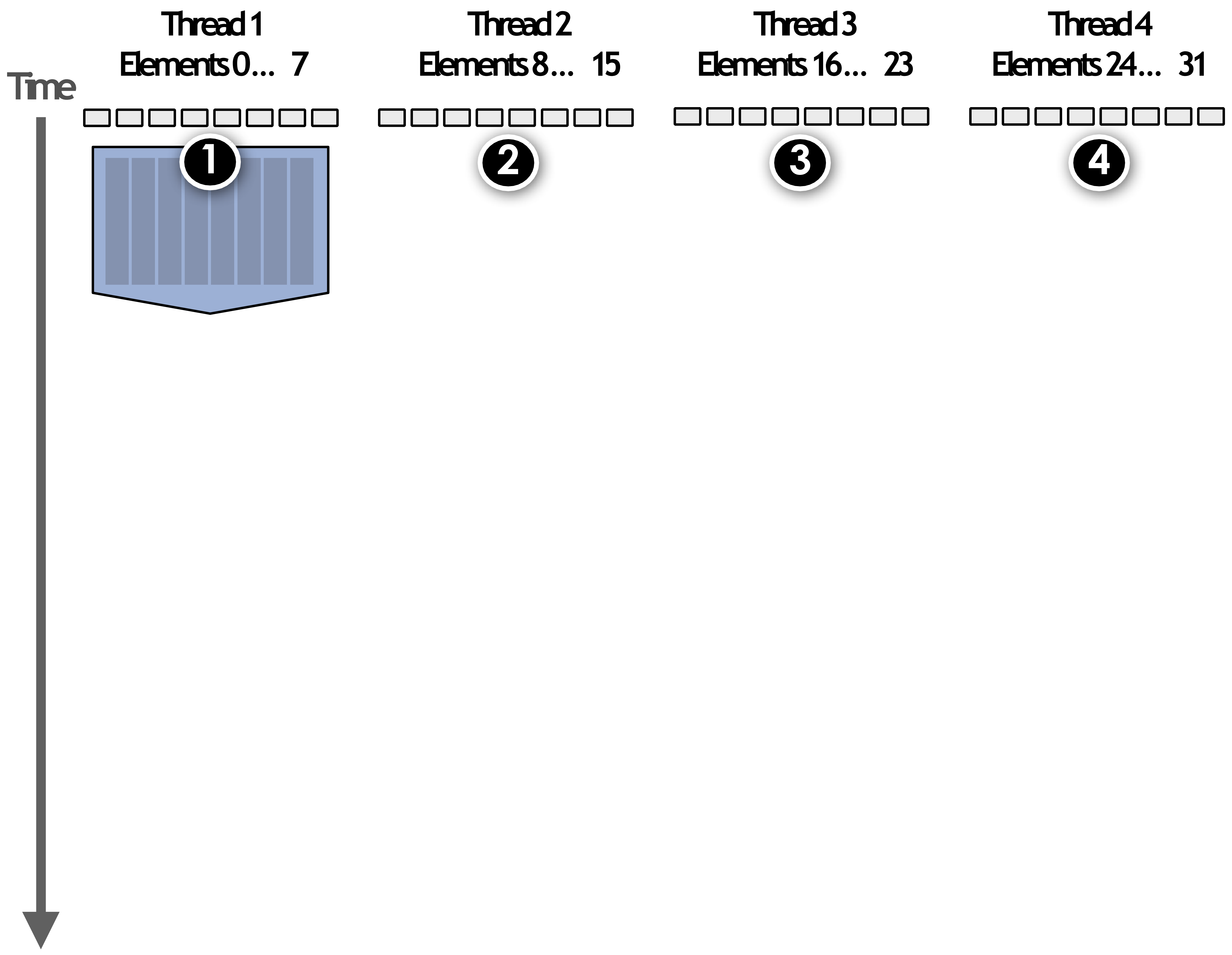
- Idea #3: <u>interleave</u> processing of multiple threads on the same core to hide stalls
  - If you can’t make progress on the current thread… work on another one

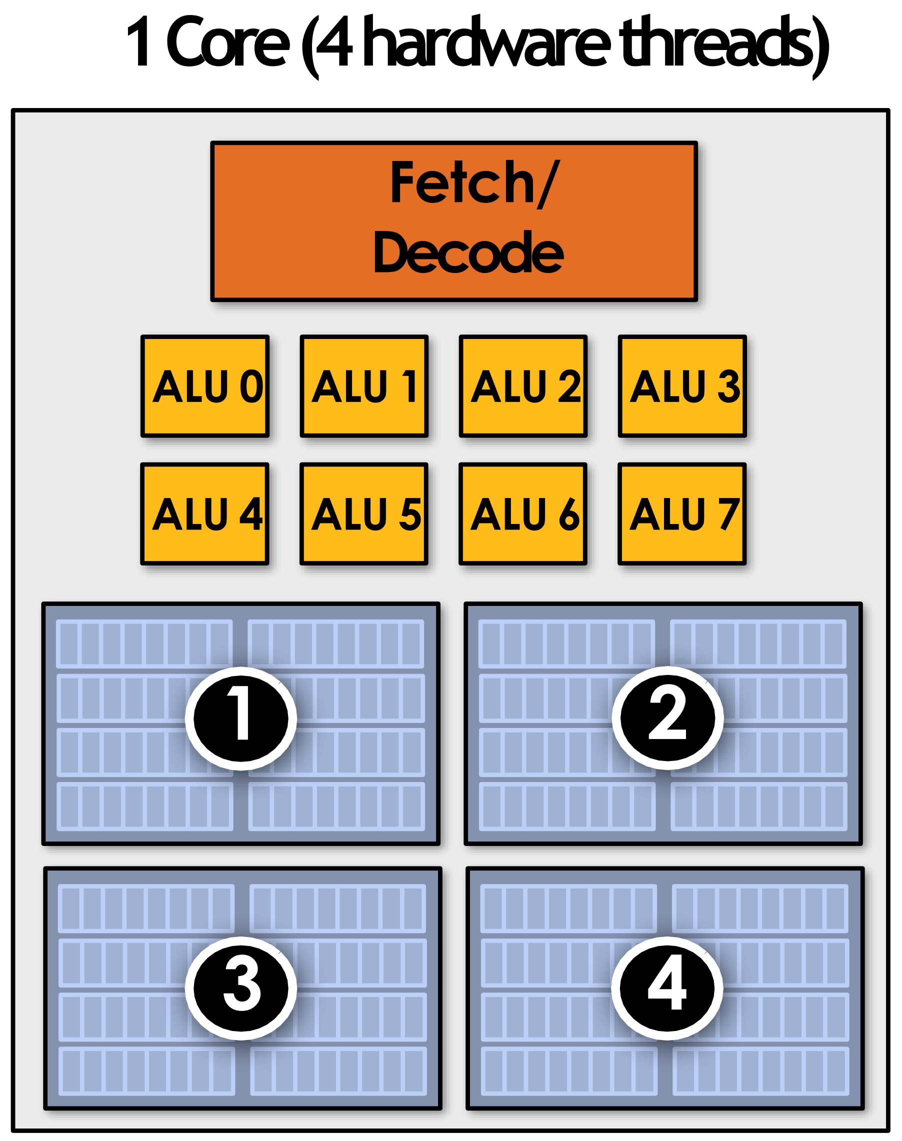
Hiding stalls with multi-threading



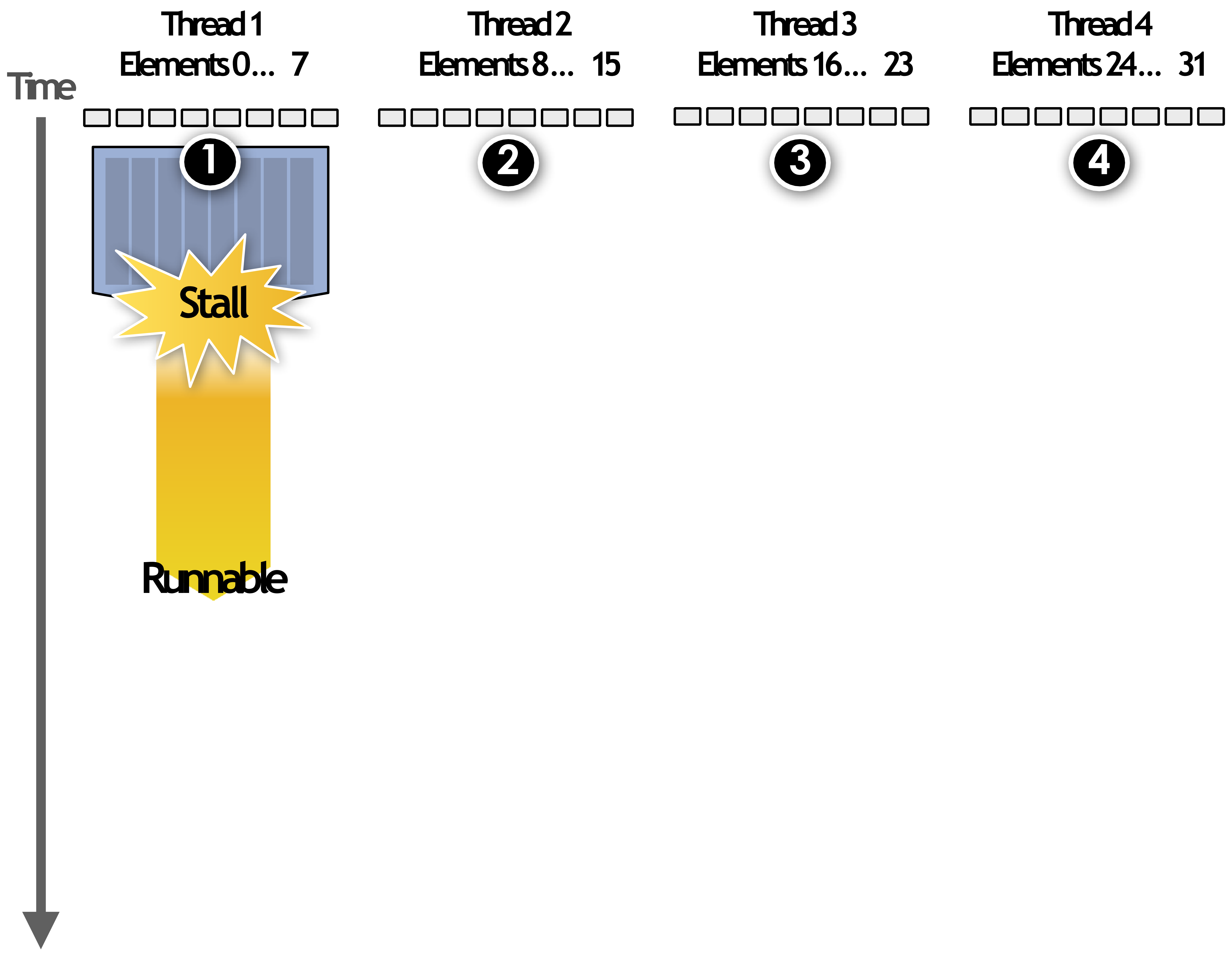


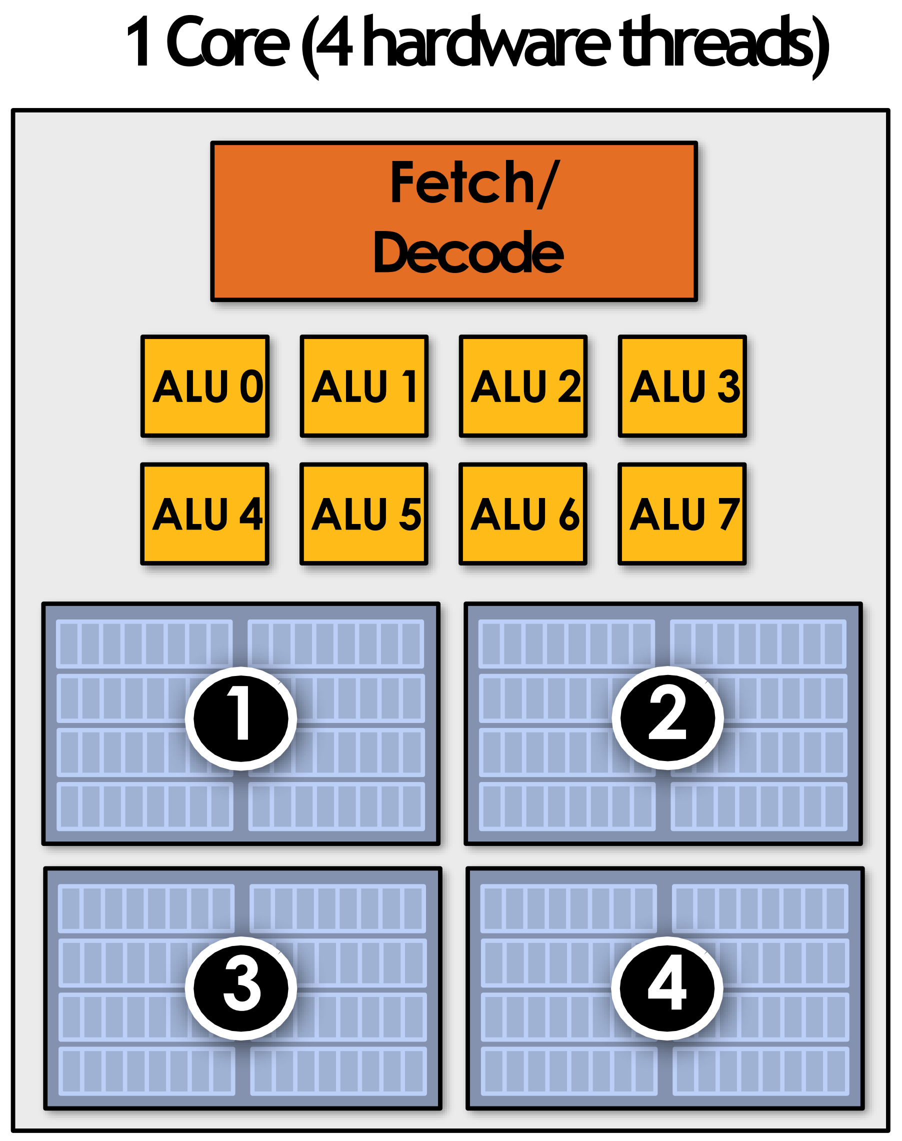
Hiding stalls with multi-threading



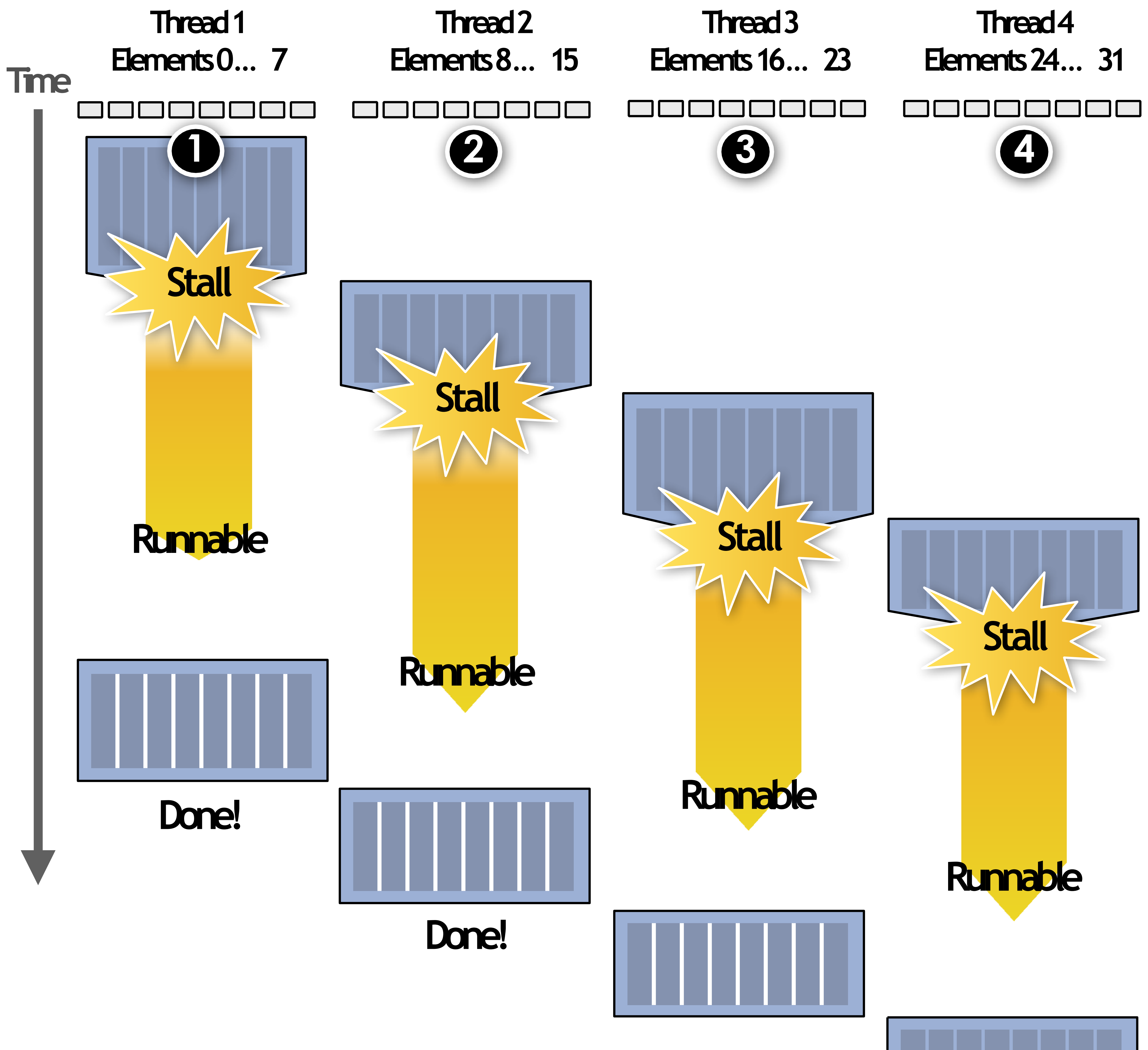


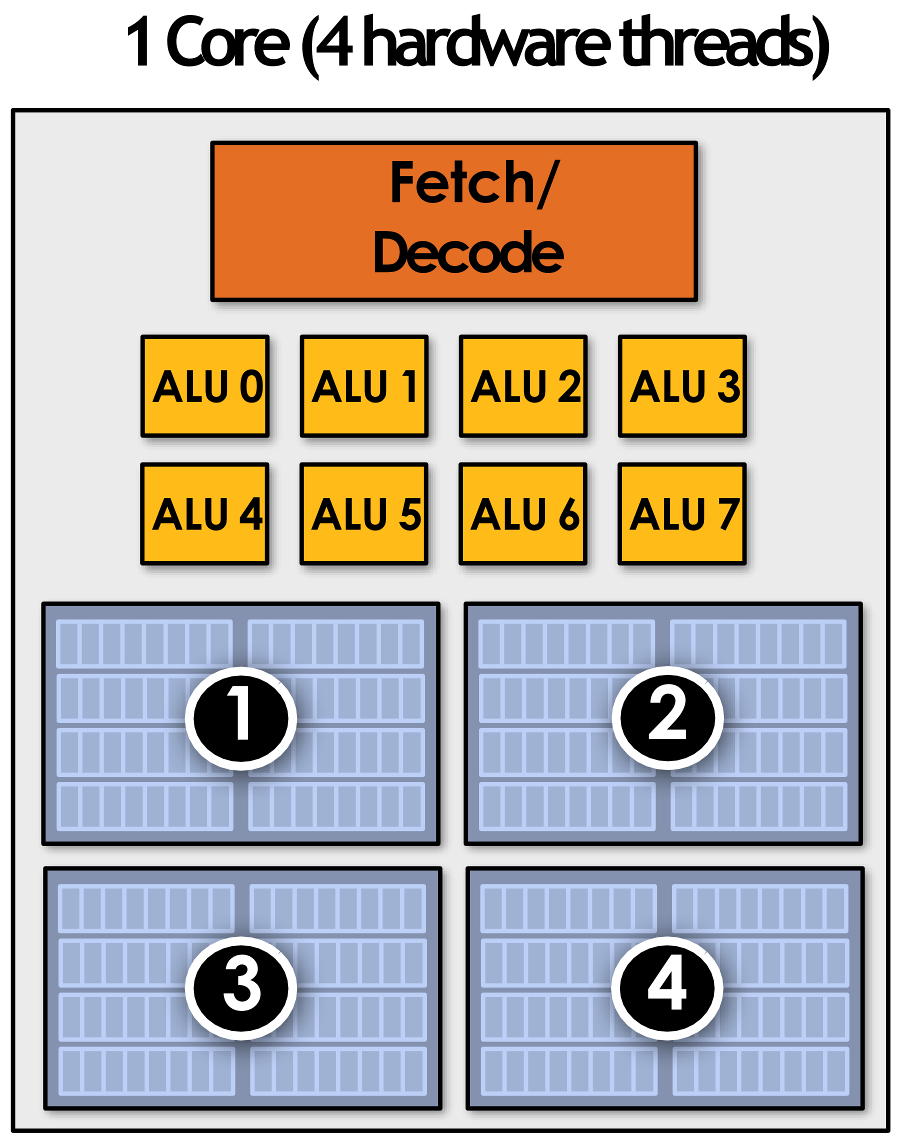
Hiding stalls with multi-threading



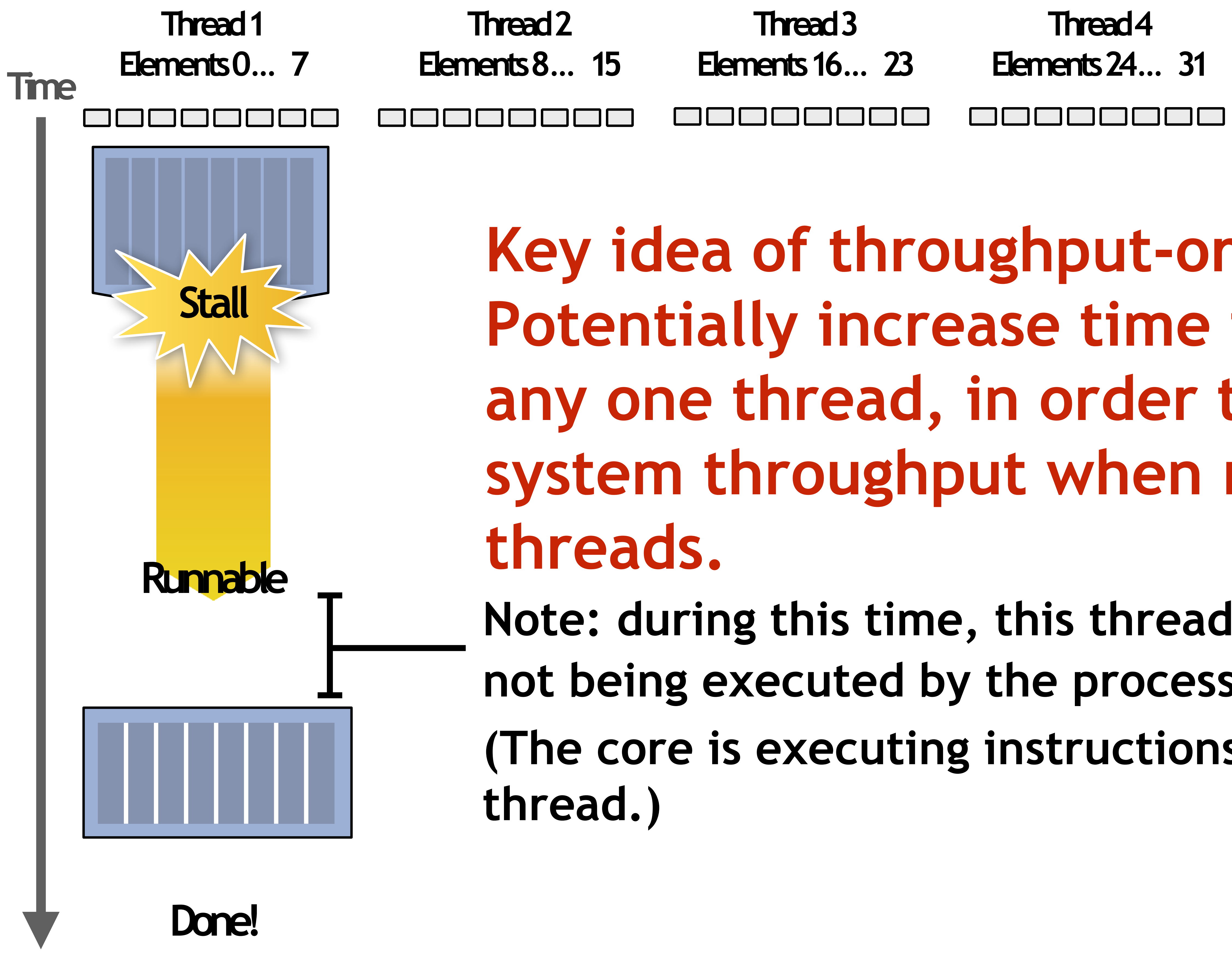


Hiding stalls with multi-threading





Throughput computing: a trade-off



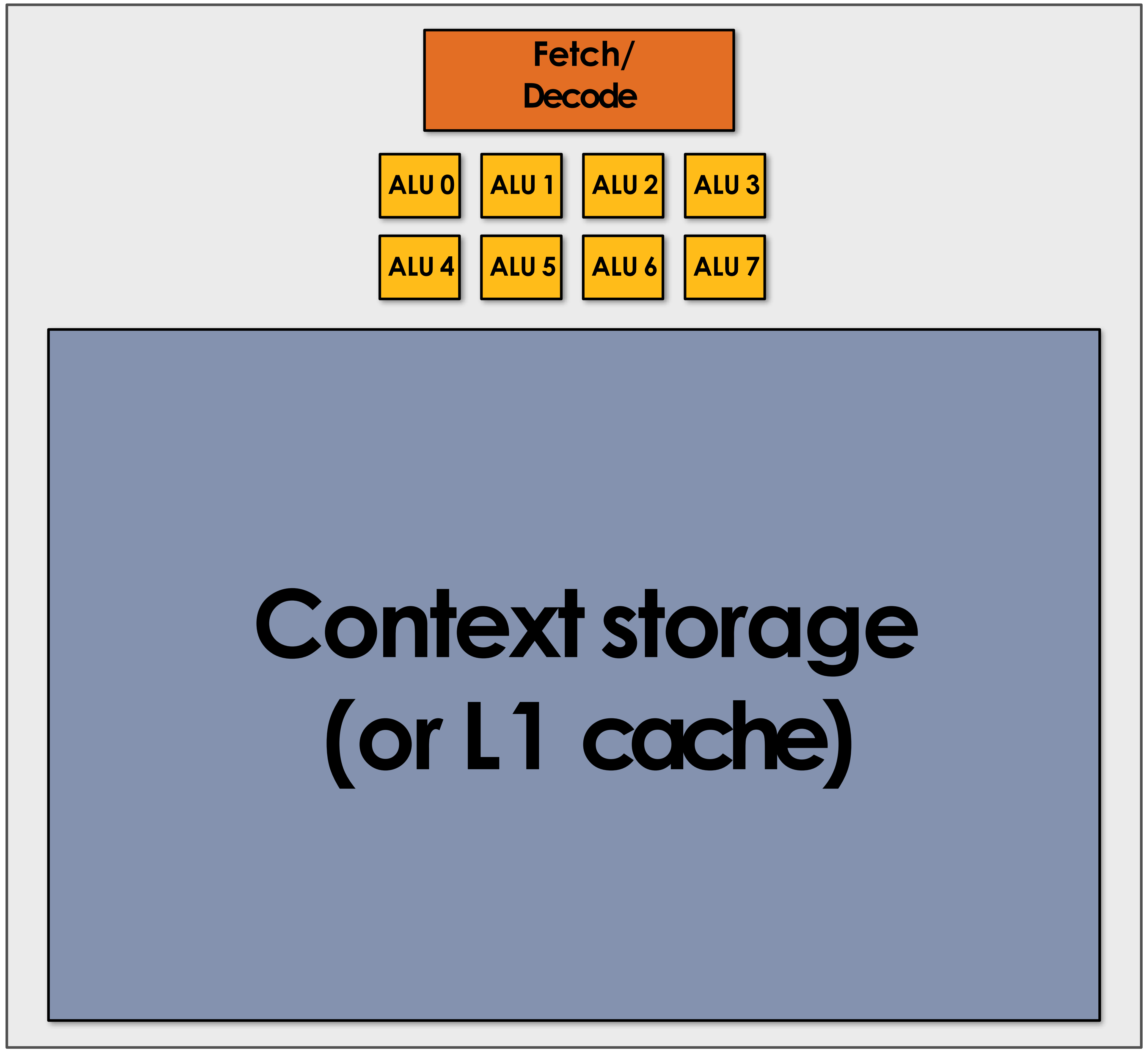
Key idea of throughput-oriented systems: Potentially increase time to complete work by any one thread, in order to increase overall system throughput when running multiple threads.

Note: during this time, this thread is runnable, but it is not being executed by the processor core.

(The core is executing instructions from another thread.)

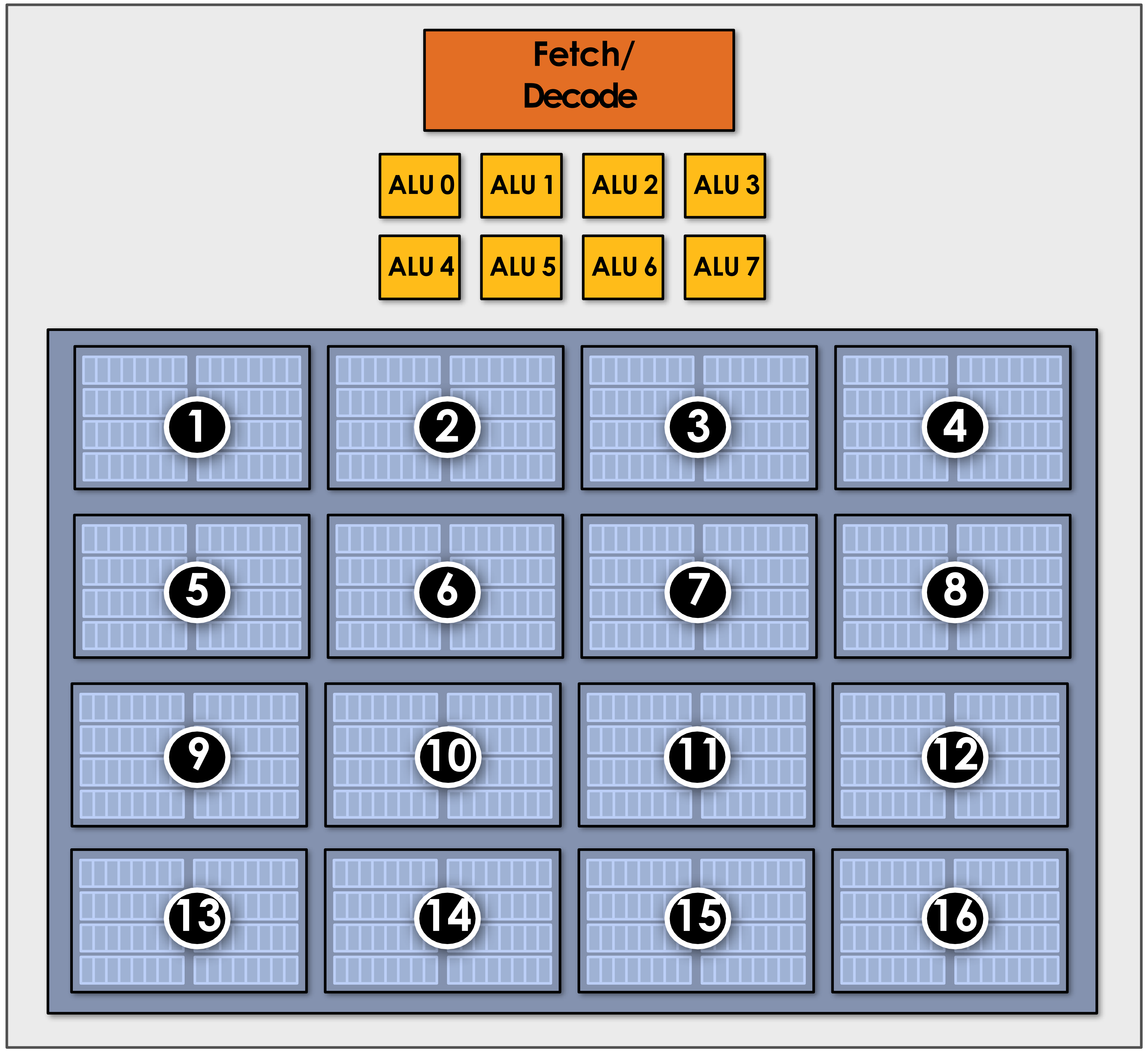
No free lunch: storing execution contexts

Consider on-chip storage of execution contexts as a finite resource



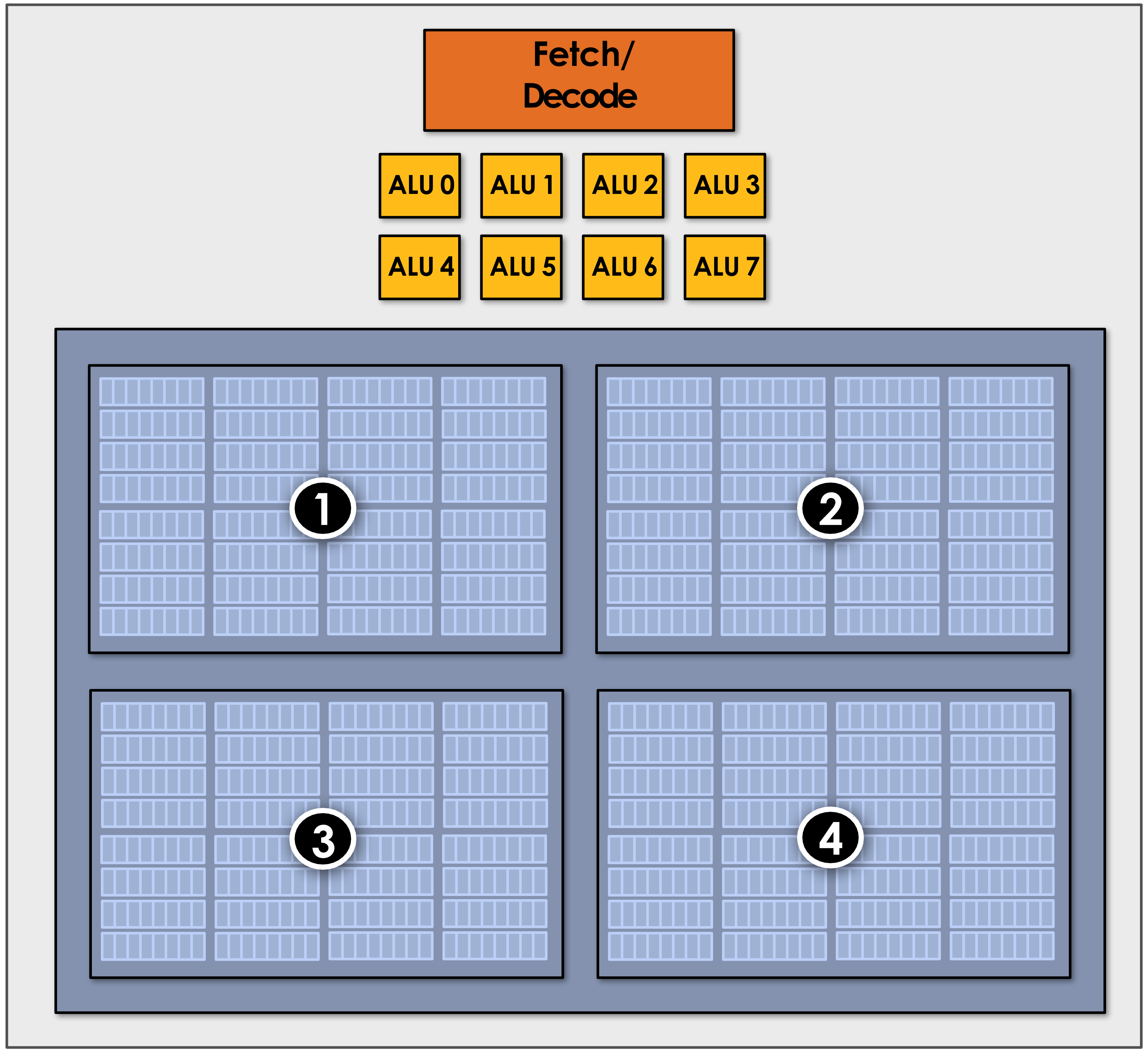
Many small contexts (high latency hiding ability)

16 hardware threads: storage for small working set per thread

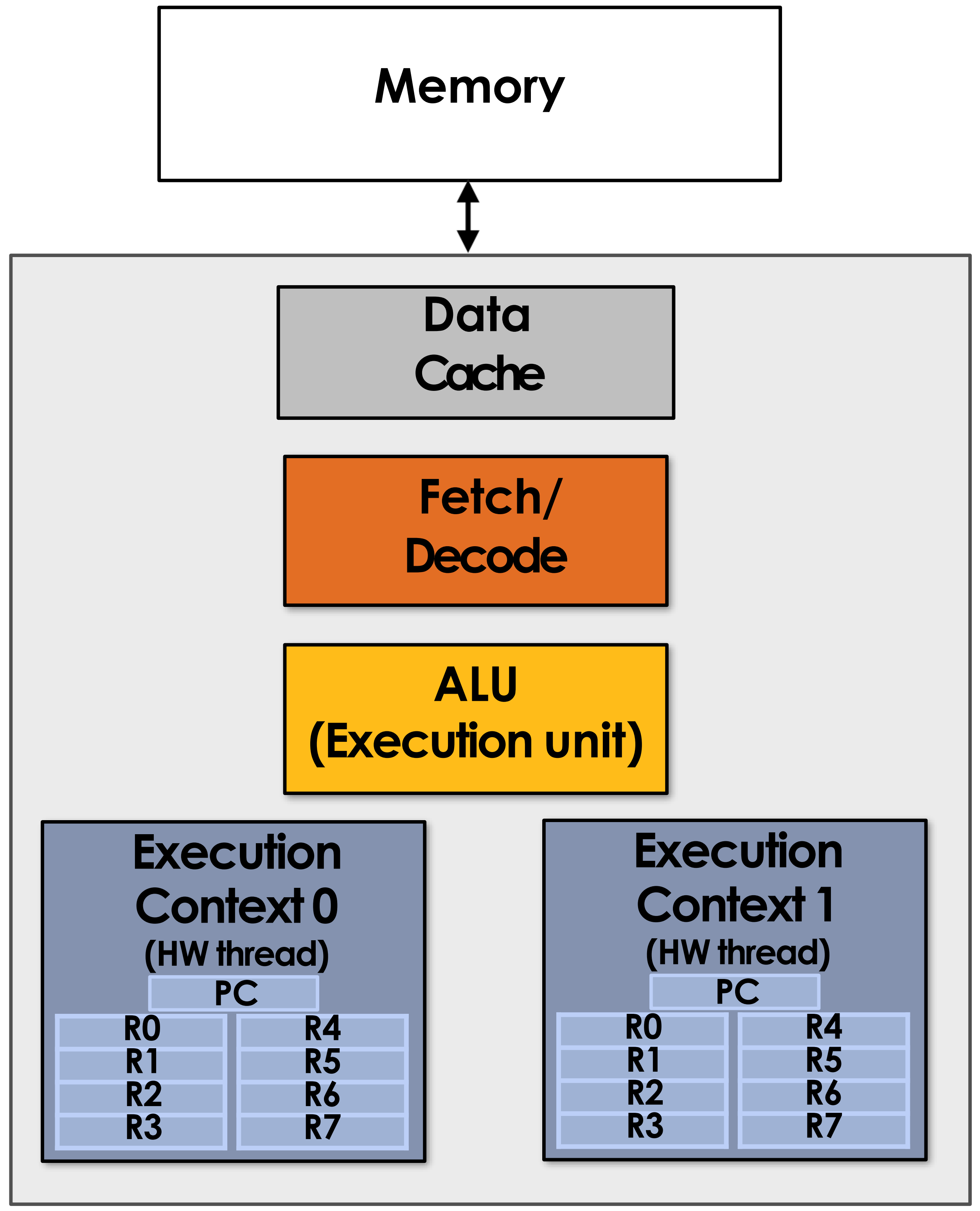


Four large contexts (low latency hiding ability)

4 hardware threads: storage for large working set per thread



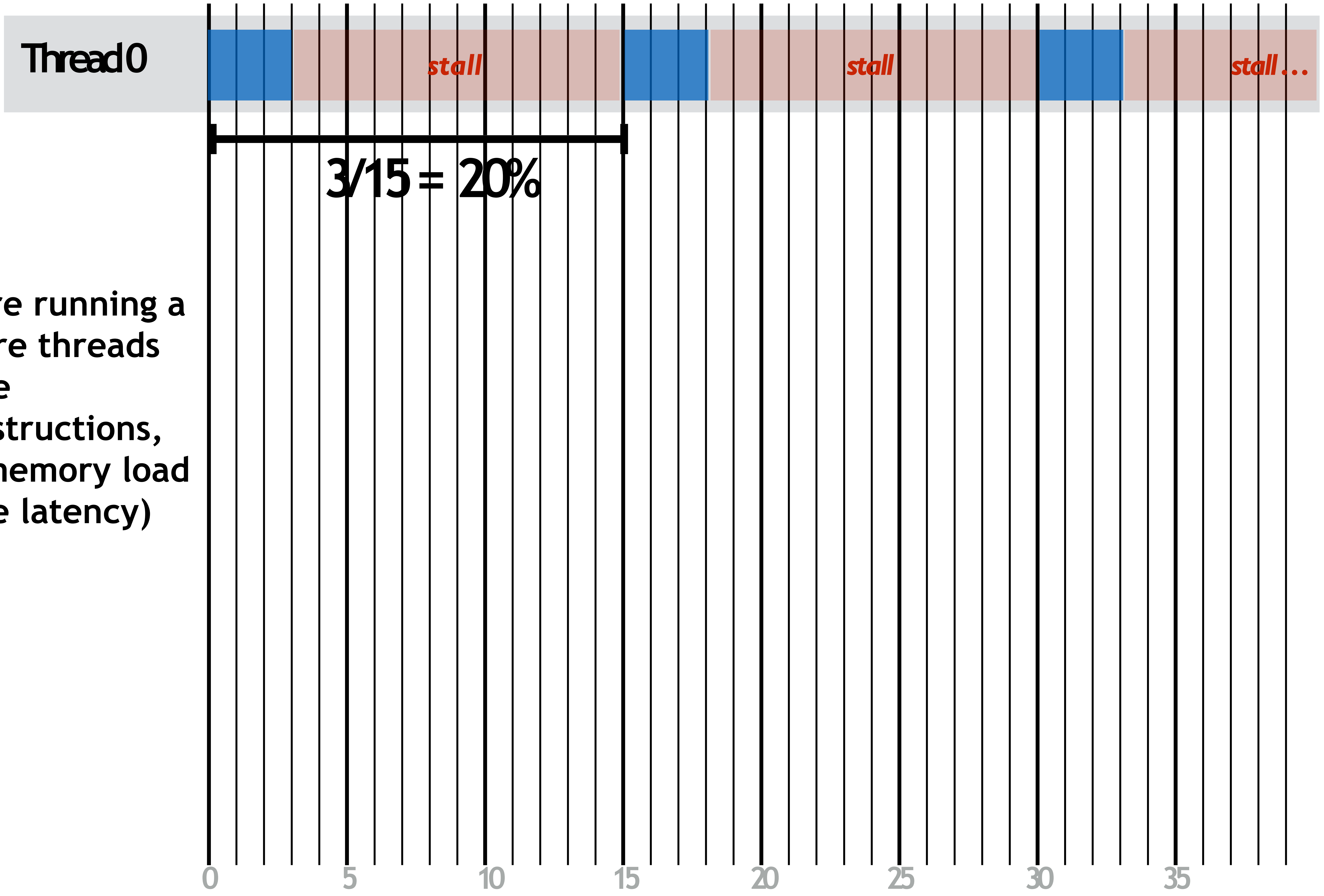
Exercise: consider a simple two threaded core



Single core processor, multi-threaded core (2 threads).

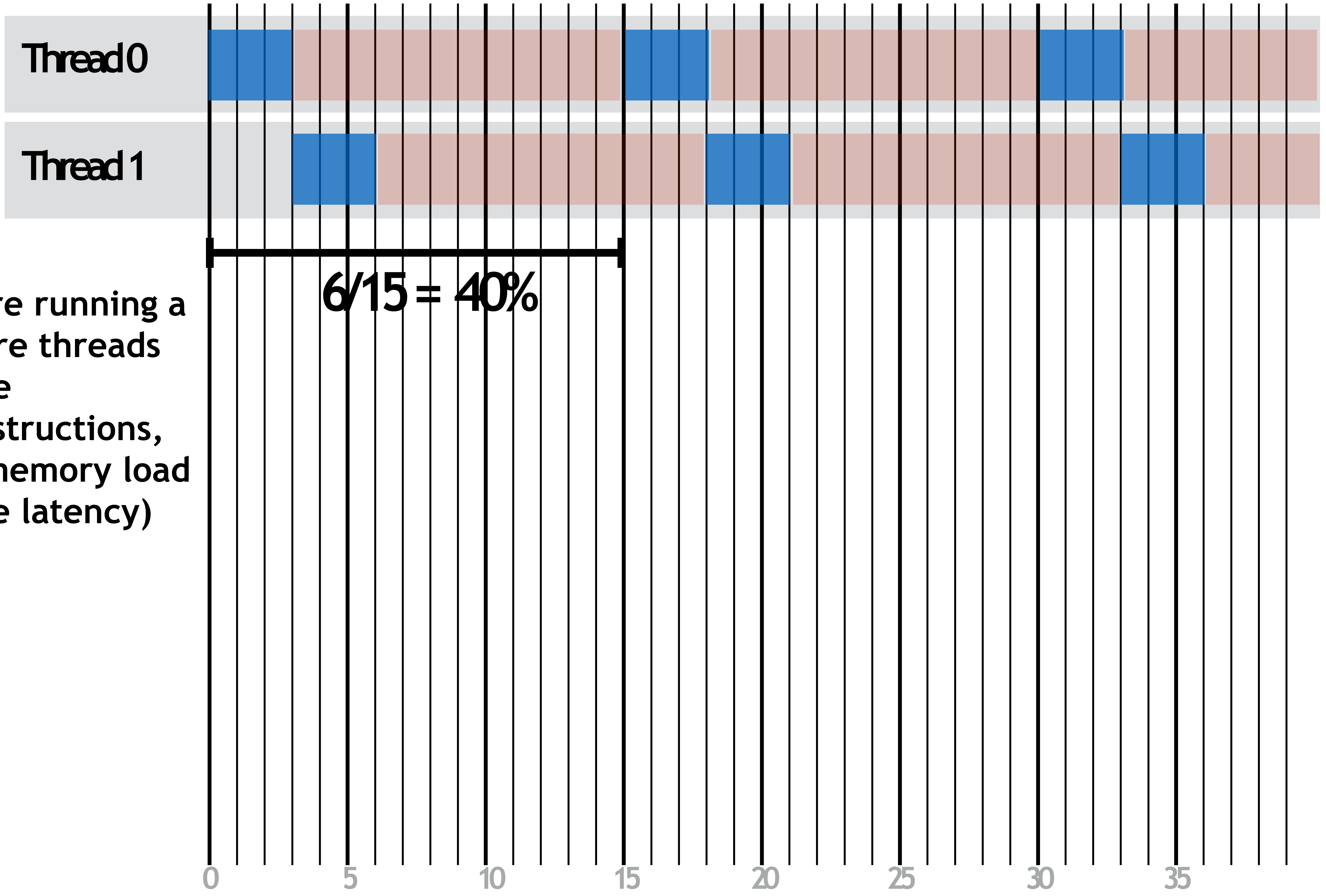
Can run one scalar instruction per clock from one of the hardware threads

What is the utilization of the core? (one thread)



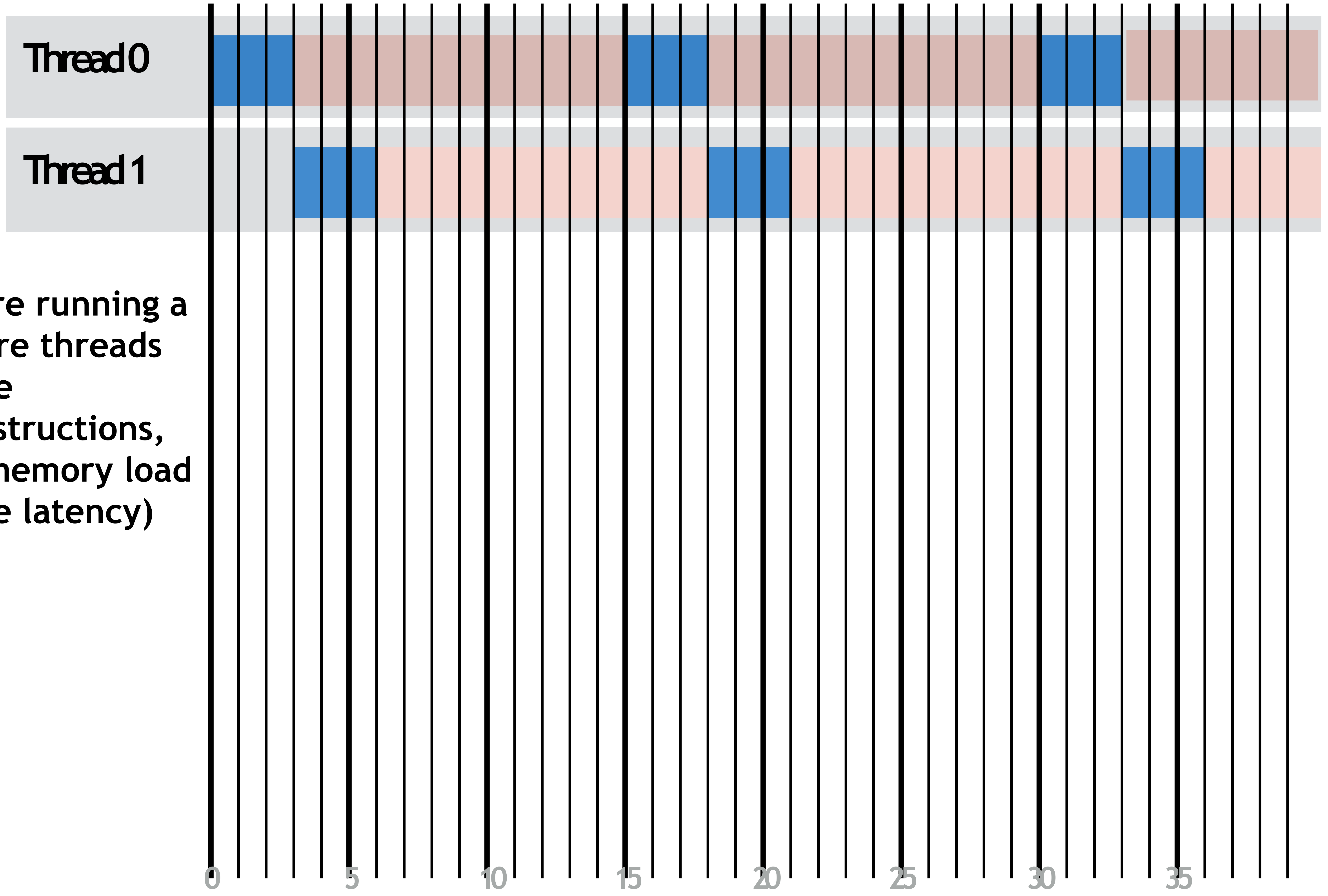
Assume we are running a program where threads perform three arithmetic instructions, followed by memory load (with 12 cycle latency)

What is the utilization of the core? (two threads)



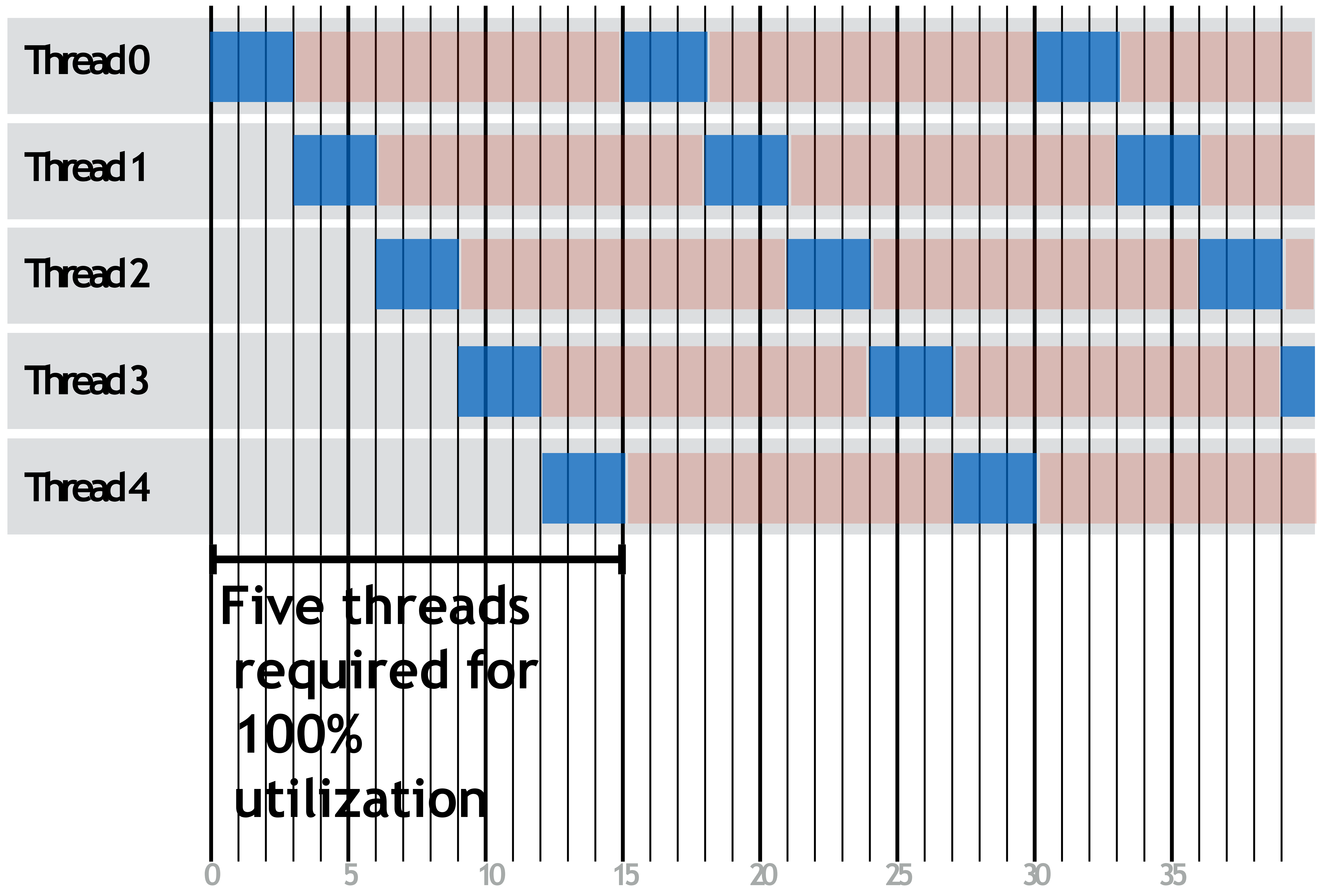
Assume we are running a program where threads perform three arithmetic instructions, followed by memory load (with 12 cycle latency)

How many threads are needed to achieve 100% utilization?

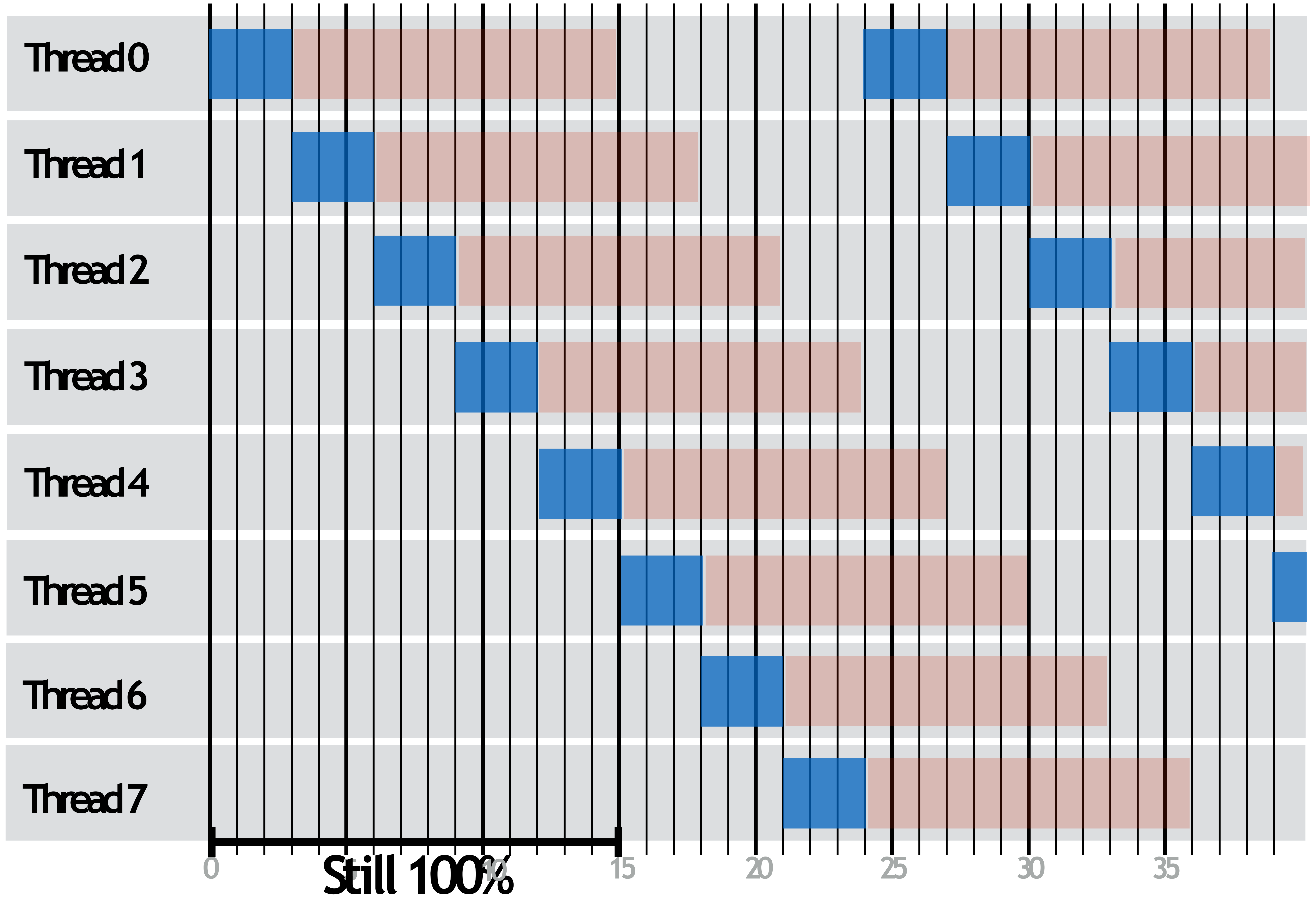


Assume we are running a program where threads perform three arithmetic instructions, followed by memory load (with 12 cycle latency)

Five threads needed to obtain 100% utilization

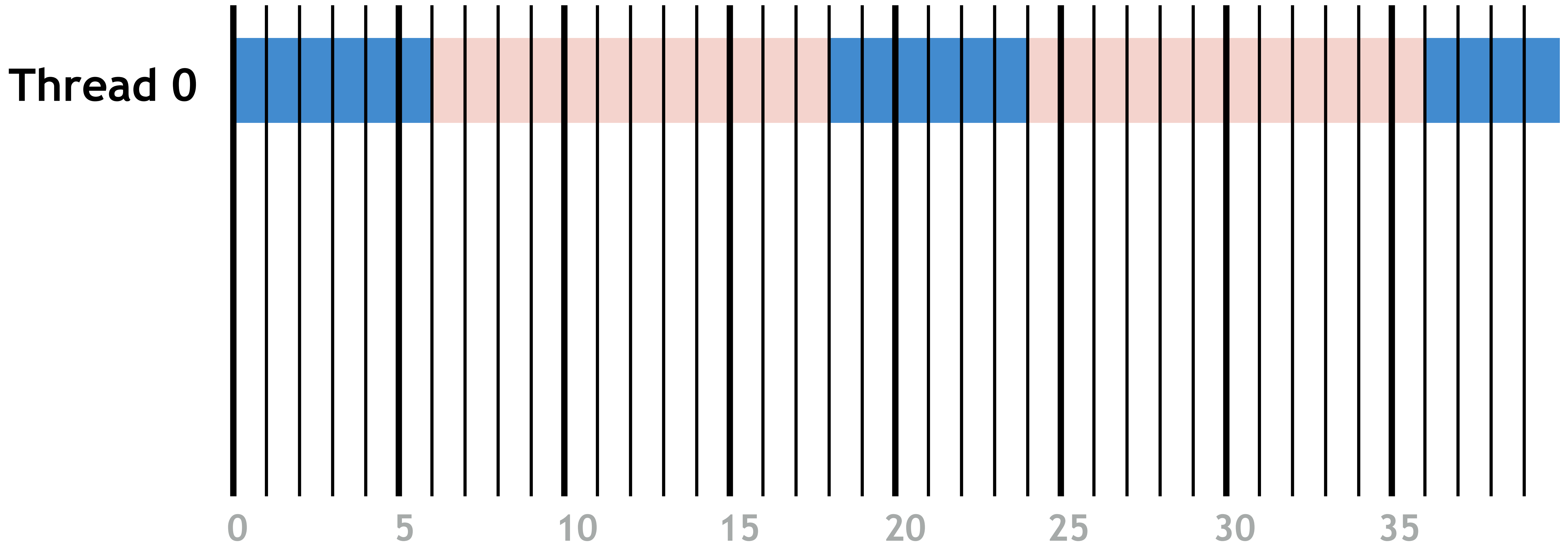


Additional threads yield no benefit (already 100% utilization)



Breakout: How many threads are needed to achieve 100% utilization?

Threads now perform *six arithmetic instructions*, followed by memory load (with 12 cycle latency)



How does a higher ratio of math instructions to memory latency affect the number of threads needed for latency hiding?

Takeaway (point 1):

A processor with multiple hardware threads has the ability to ***avoid stalls*** by performing instructions from other threads when one thread must wait for a long latency operation to complete.

Note: the latency of the memory operation is not changed by multi- threading, it just no longer causes reduced processor utilization.

Takeaway (point 2):

A multi-threaded processor hides memory latency by performing arithmetic from other threads.

Programs that feature more arithmetic per memory access need fewer threads to hide memory stalls.

Hardware-supported multi-threading

- Core manages execution contexts for multiple threads
  - Core still has the same number of ALU resources: multi-threading only helps use them more efficiently in the face of high-latency operations like memory access
  - Processor makes decision about which thread to run each clock

- Interleaved multi-threading (a.k.a. temporal multi-threading)
  - What I described on the previous slides: each clock, the core chooses a thread, and runs an instruction from the thread on the core's ALUs

- Simultaneous multi-threading (SMT)
  - Each clock, core chooses instructions from multiple threads to run on ALUs
  - Example: Intel Hyper-threading (2 threads per core)

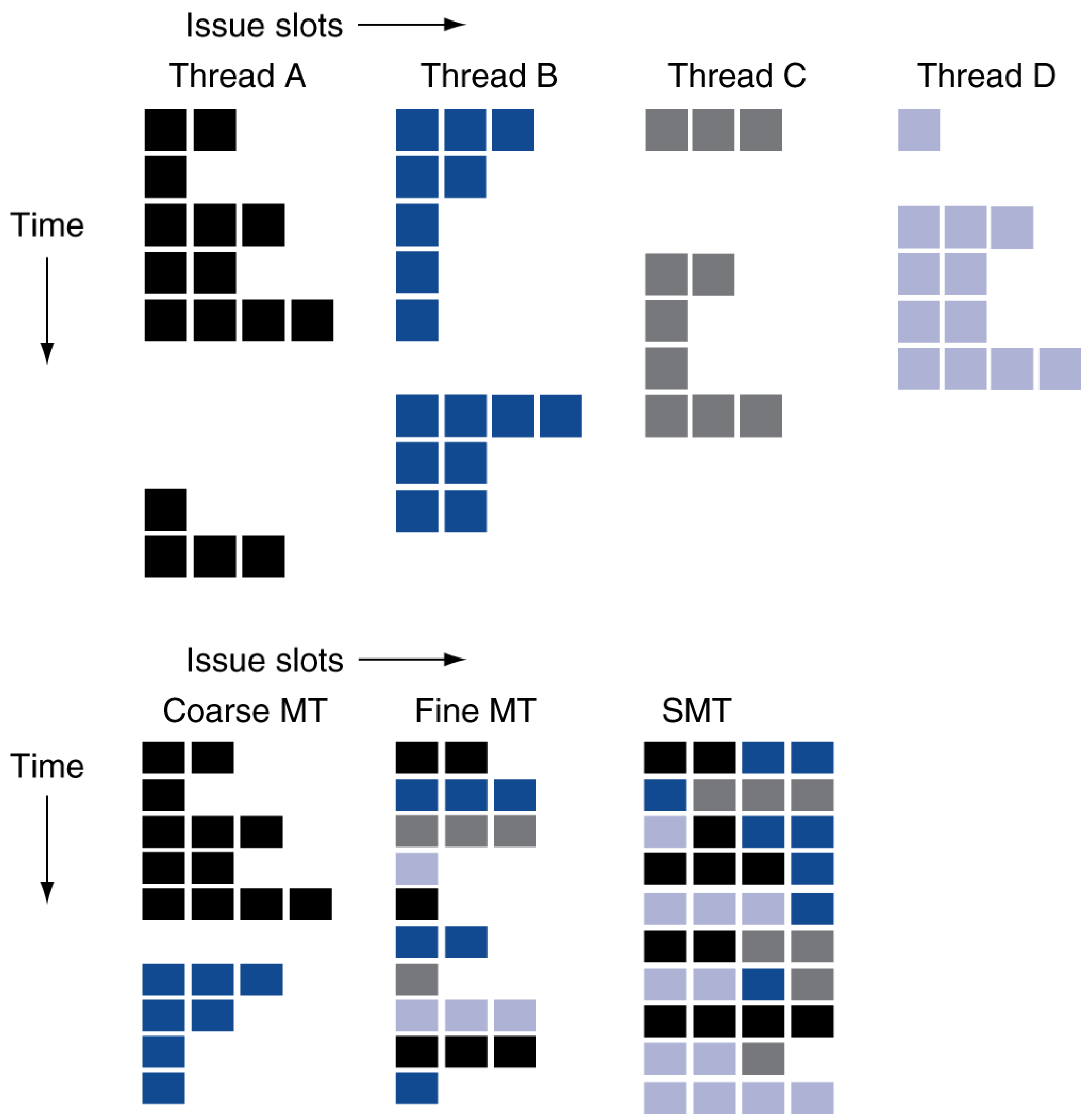
Multithreading (interleaved)

- Performing multiple threads of execution in parallel
  - Replicate registers, PC, etc.
  - Fast switching between threads
- Fine-grain multithreading
  - Switch threads after each cycle
  - Interleave instruction execution
  - If one thread stalls, others are executed
- Coarse-grain multithreading
  - Only switch on long stall (e.g., L2-cache miss)
  - Simplifies hardware, but doesn’t hide short stalls (eg, data hazards)

Simultaneous Multithreading

- In multiple-issue dynamically scheduled processor
  - Schedule instructions from multiple threads
  - Instructions from independent threads execute when function units are available
  - Within threads, dependencies handled by scheduling and register renaming
- Example: Intel Pentium-4 HT
  - Two threads: duplicated registers, shared function units and caches

Multithreading Example



Example of multi-core chip

16 cores

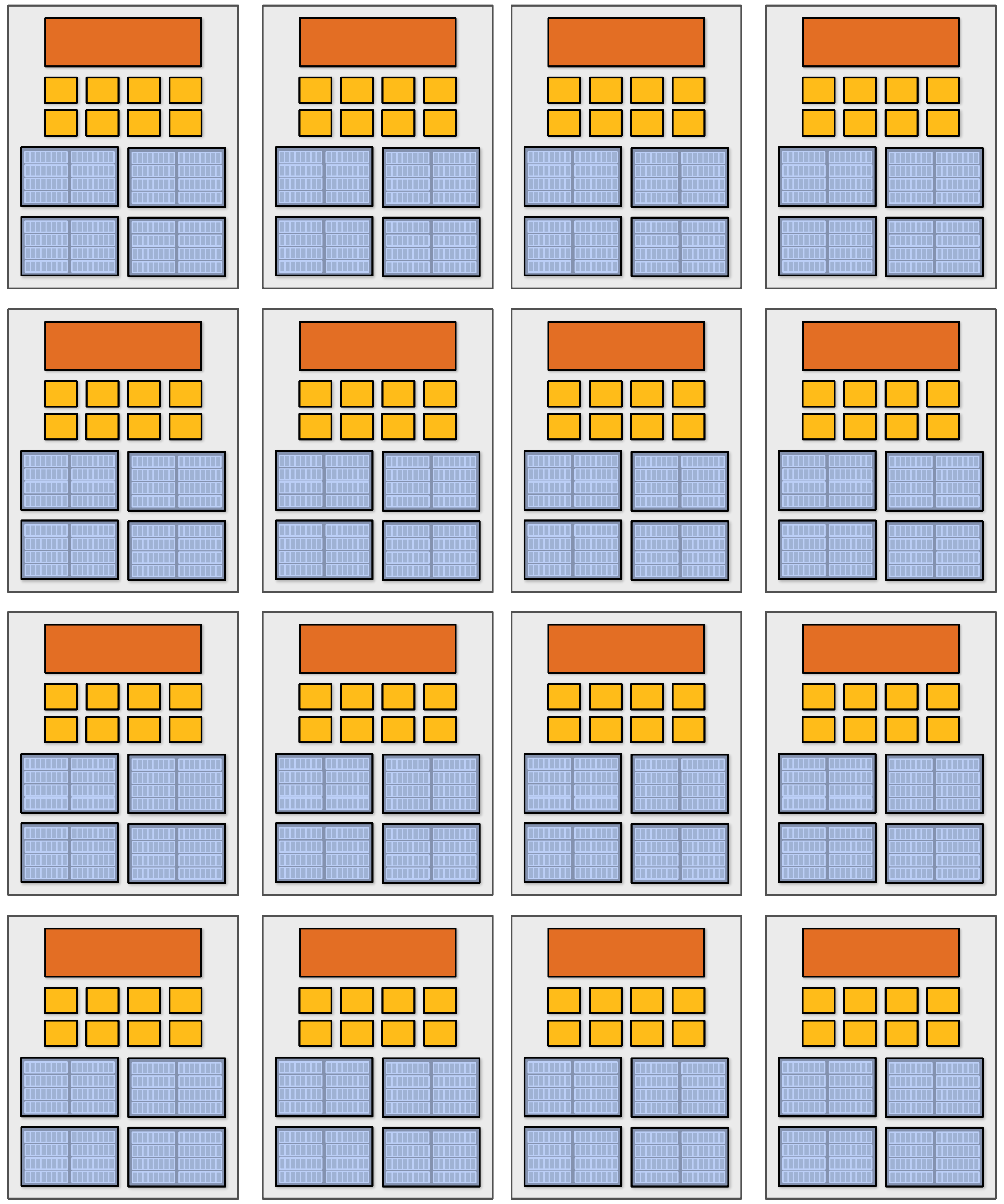
8 SIMD ALUs per core (128 total)

4 threads per core

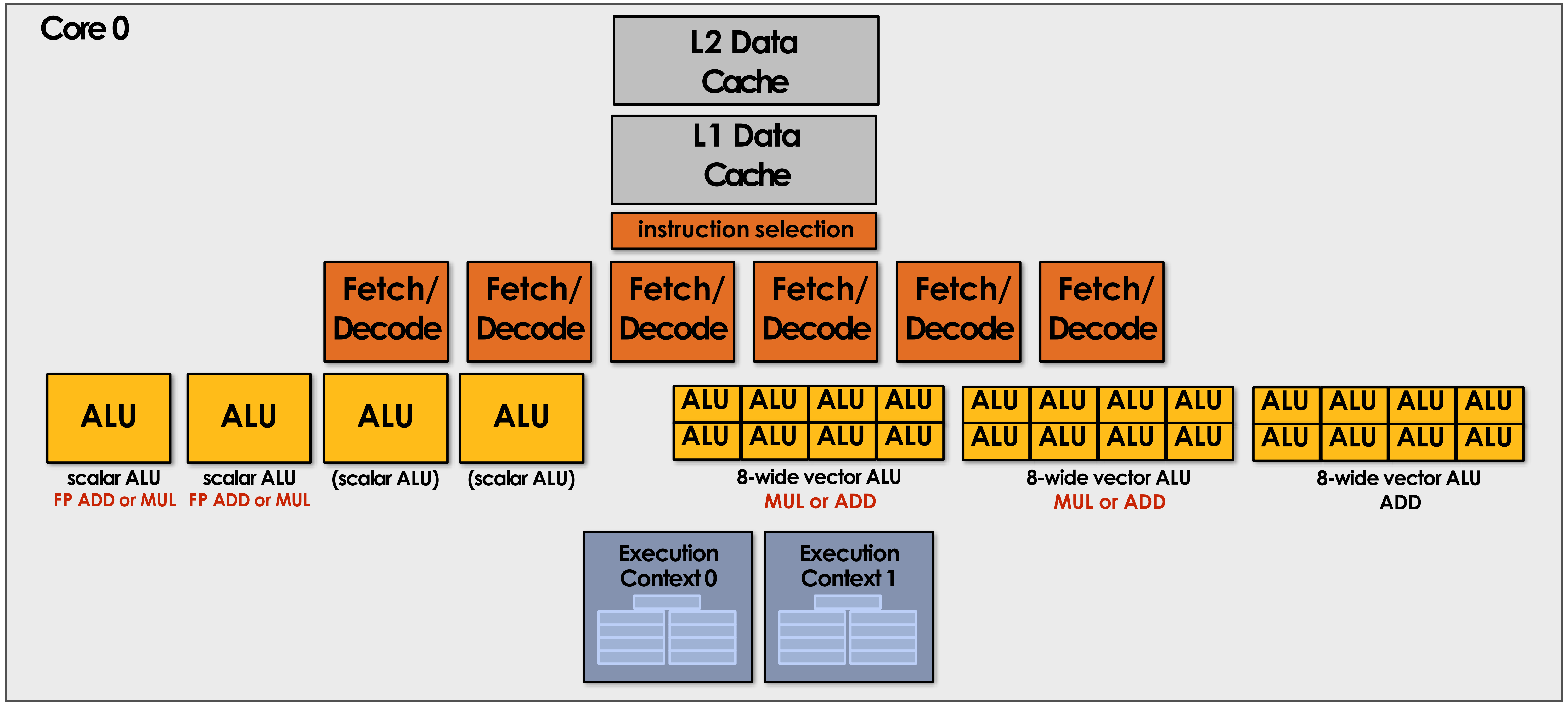
16 simultaneous instruction streams

64 total concurrent instruction streams

512 independent pieces of work are needed to run chip with maximal latency hiding ability



Example: Intel Skylake/Kaby Lake core

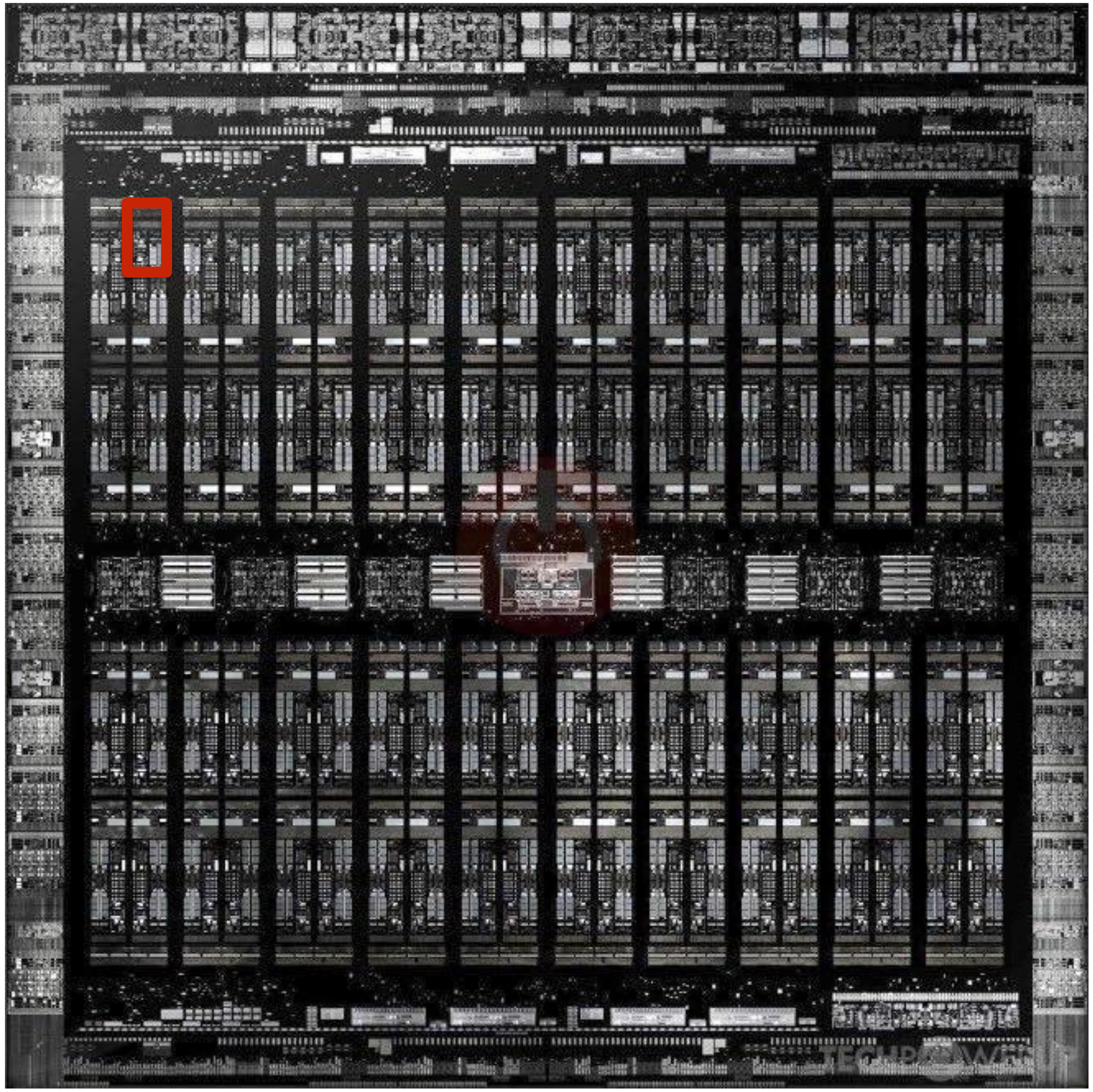


Two-way multi-threaded cores (2 threads).
Each core can run up to four independent scalar instructions and up to three 8-wide vector instructions
(up to 2 vector mul or 3 vector add)

Not shown on this diagram: units for LD/ST operations

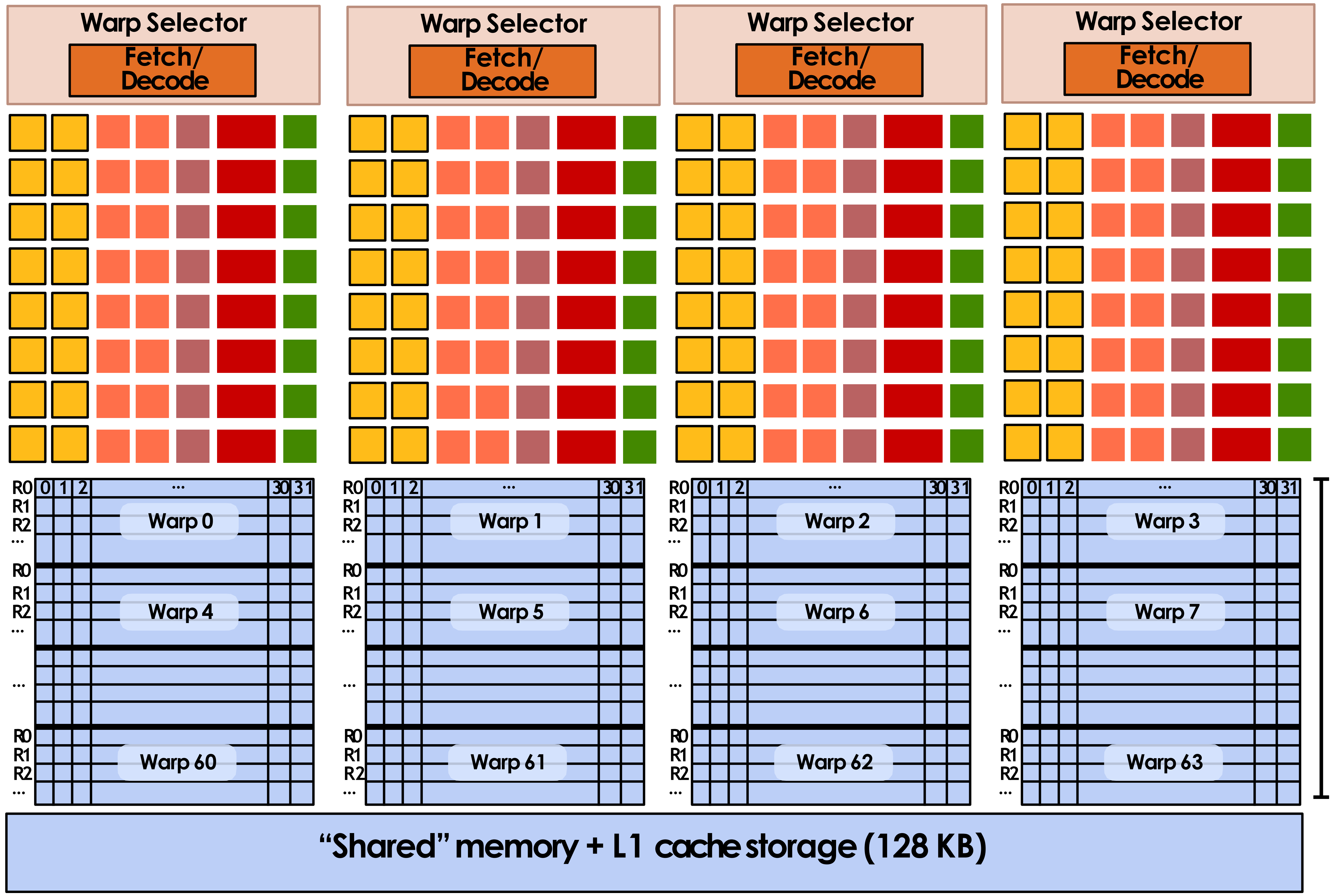
NVIDIA V100

- SM = "Streaming Multi-processor"



GPUs: extreme throughput-oriented processors

This is one NVIDIA V100 streaming multi-processor (SM) unit



64 "warp" execution contexts per SM

Wide SIMD: 16-wide SIMD ALUs (carry out 32-wide SIMD execute over 2 clocks)

64 x 32 = up to 2048 data items processed concurrently per "SM" core

64 KB registers per sub-core

256 KB registers in total per SM

Registers divided among (up to) 64 "warps" per SM

🟨 = SIMD fp32 functional unit, control shared across 16 units (16 x MUL-ADD per clock*)

🟧 = SIMD int functional unit, control shared across 16 units (16 x MUL/ADD per clock*)

🟫 = SIMD fp64 functional unit, control shared across 8 units (8 x MUL/ADD per clock**)

🟥 = Tensor core unit

🟩 = Load/store unit

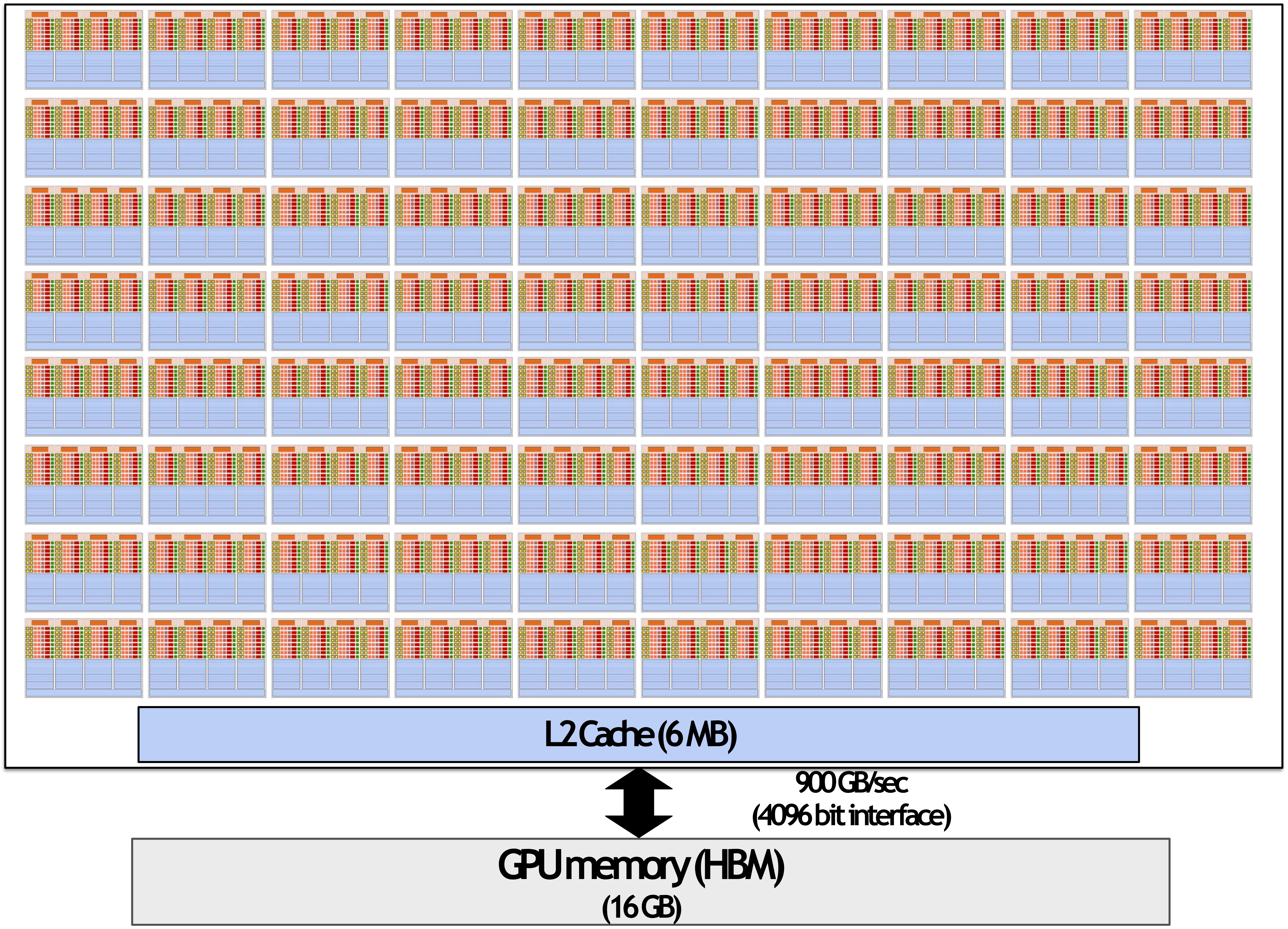
* one 32-wide SIMD operation every 2 clocks

** one 32-wide SIMD operation every 4 clocks

NVIDIA V100

There are 80 SM cores on the V100:

That’s 163,840 pieces of data being processed concurrently to get maximal latency hiding!



The story so far...

To utilize modern parallel processors efficiently, an application must:

1. Have sufficient parallel work to utilize all available execution units (across many cores and many execution units per core)

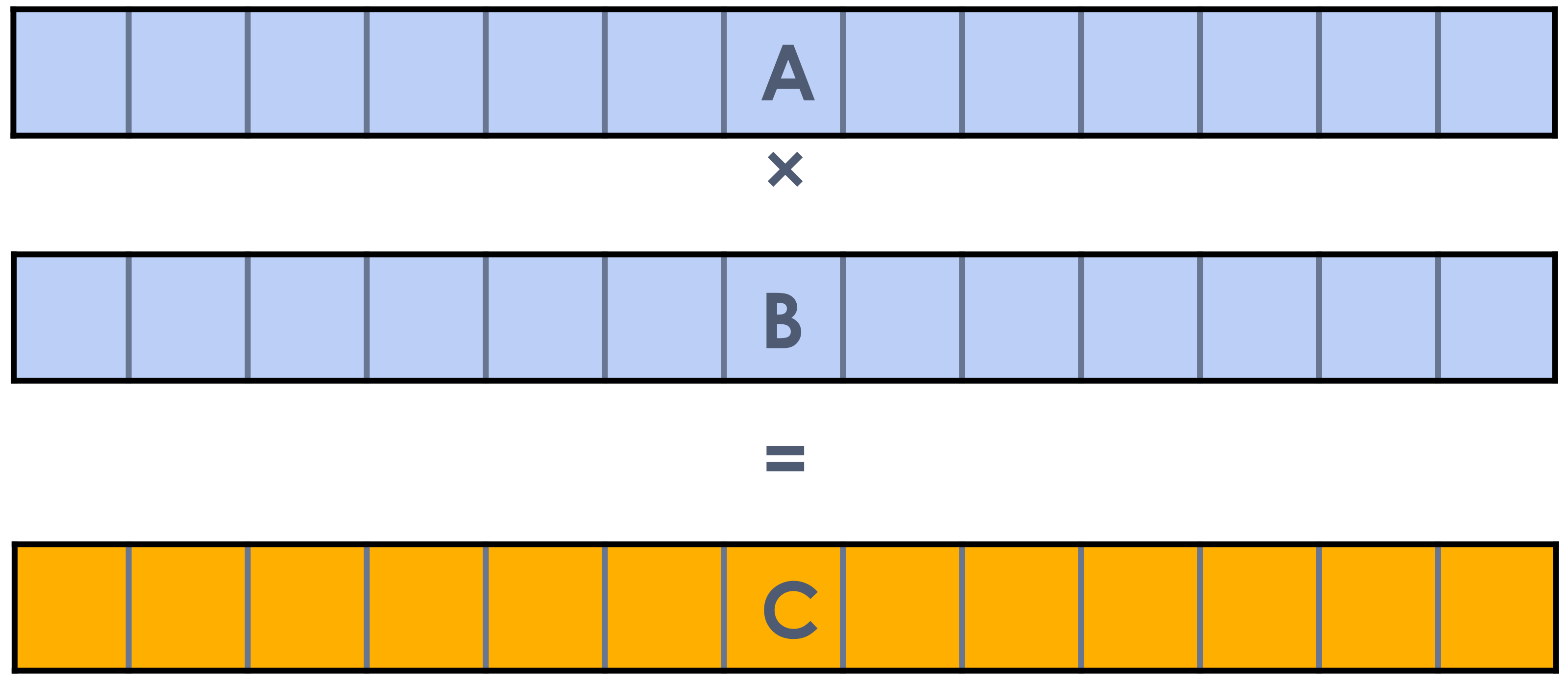
2. Groups of parallel work items must require the same sequences of instructions (to utilize SIMD execution)

3. Expose more parallel work than processor ALUs to enable interleaving of work to hide memory stalls

Thought experiment

Task: 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] × B[i]
- Store result into C[i]



Is this a good application to run on a modern throughput-oriented parallel processor?

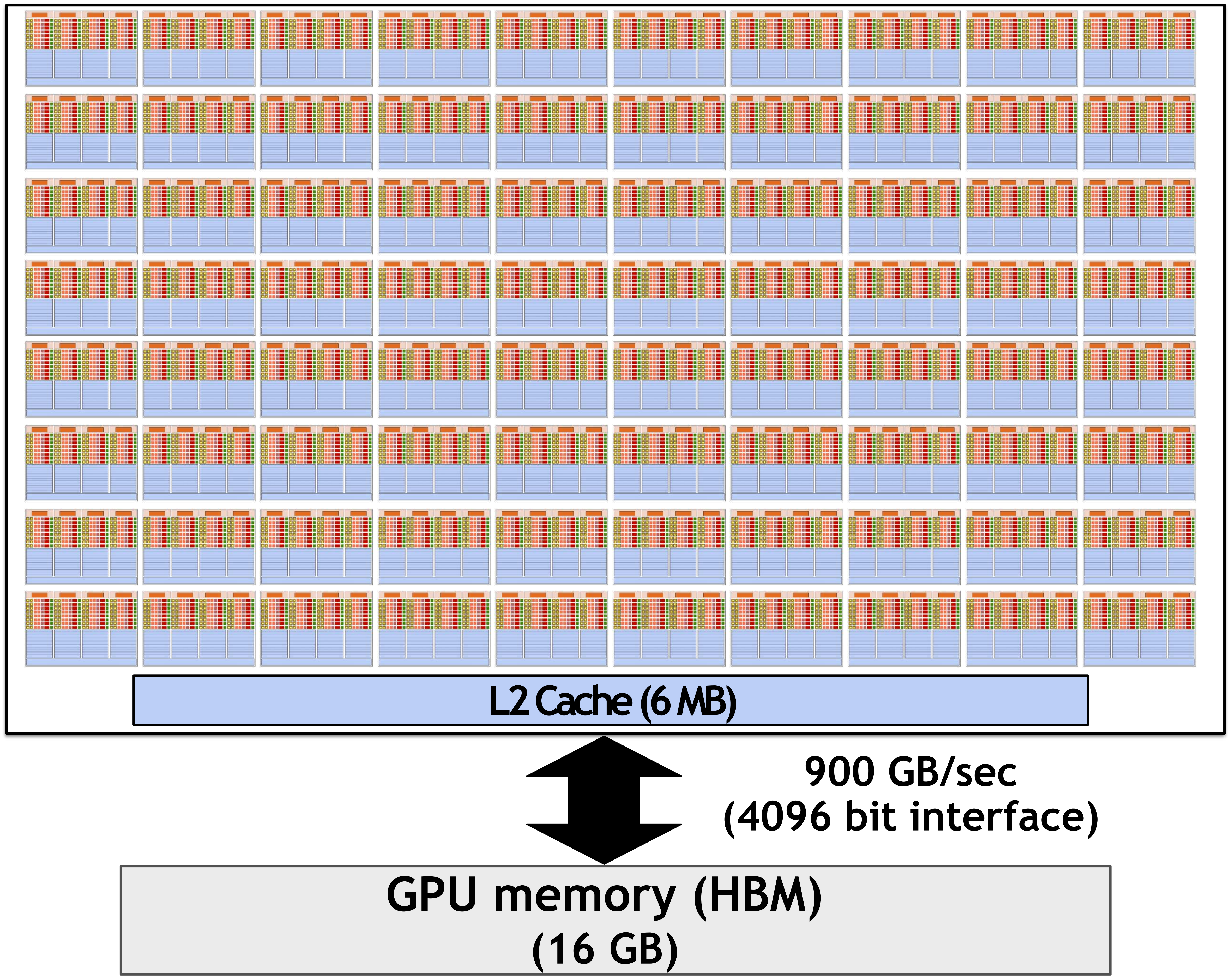
Oh, one more thing… (if time)

NVIDIA V100

There are 80 SM cores on the V100:

80 SM x 64 fp32 ALUs per SM = 5120 ALUs

Think about supplying all those ALUs with data each clock. 🙀

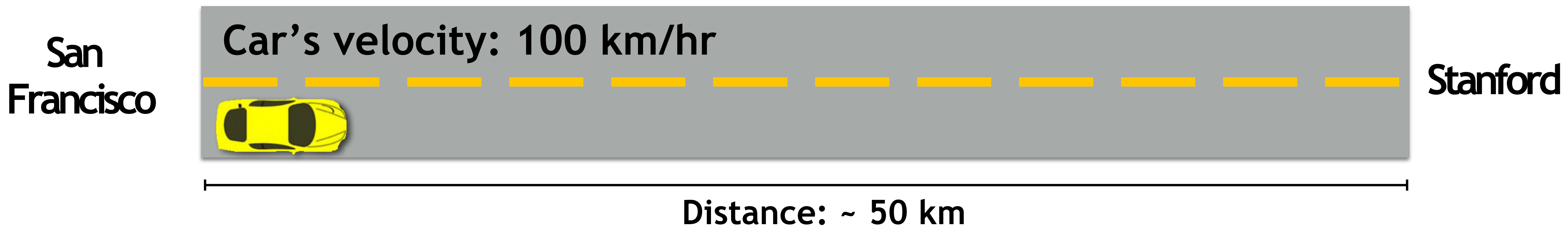


Understanding latency and bandwidth

Understanding traffic jam

Everyone wants to get to back to the same place!

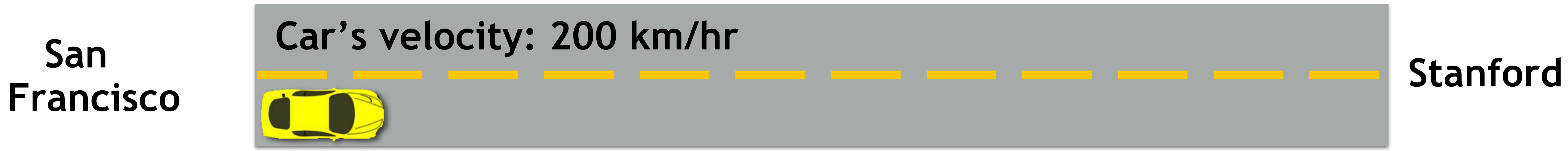
Assume only one car in a lane of the highway at once.
When car on highway reaches Stanford, the next car
leaves San Francisco.



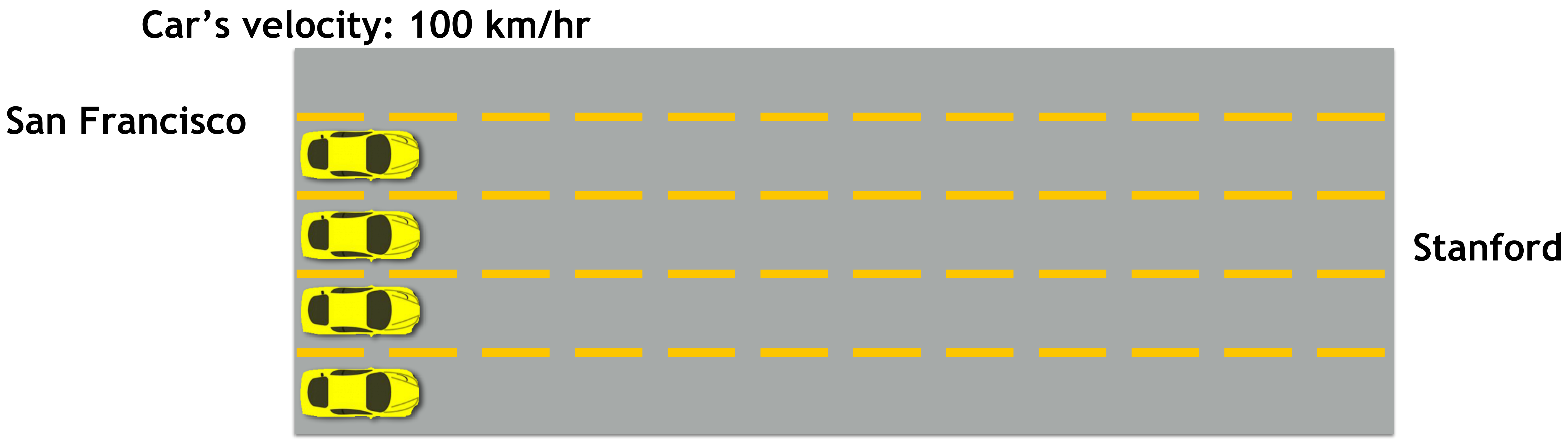
Latency of driving from San Francisco to Stanford: 0.5

hours Throughput: 2 cars per hour

Improving throughput

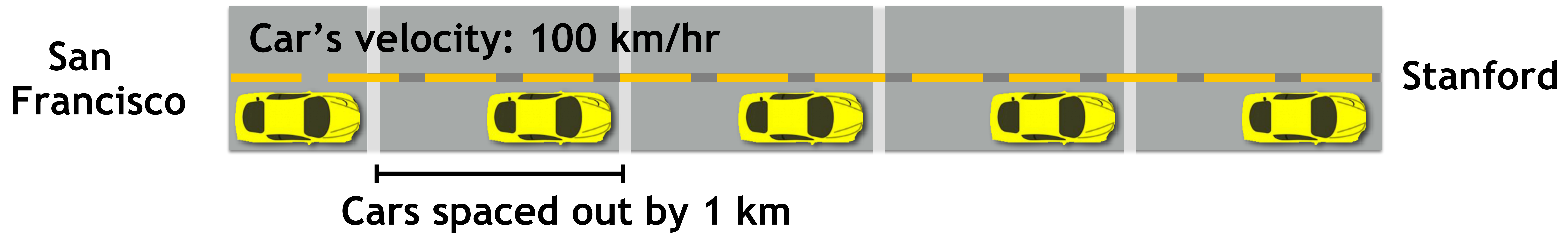


Approach 1: drive faster!
Throughput = 4 cars per hour

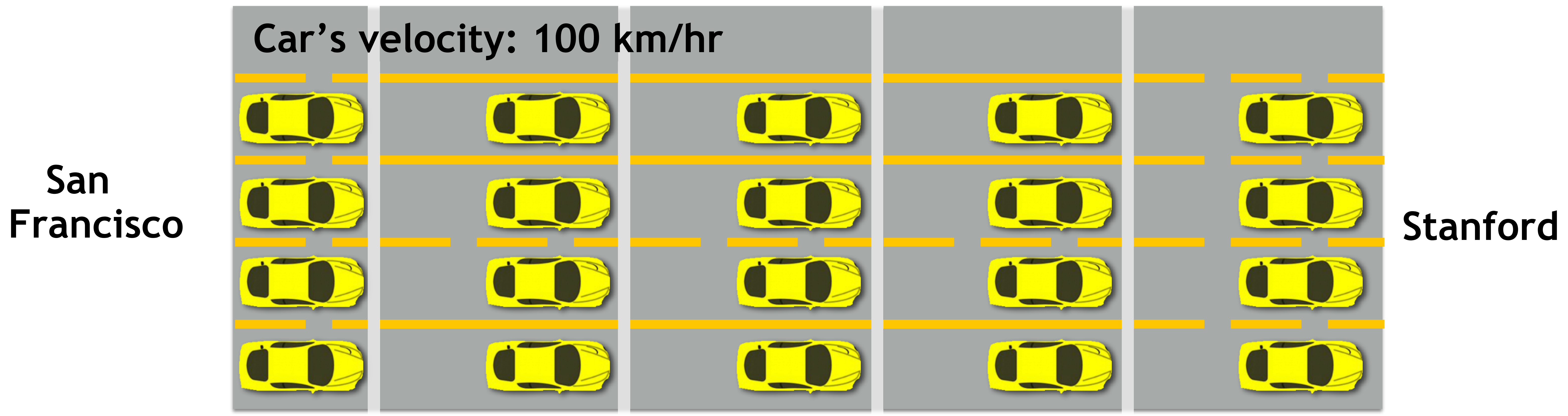


Approach 2: build more lanes!
Throughput = 8 cars per hour (2 cars per hour per lane)

Using the highway more efficiently



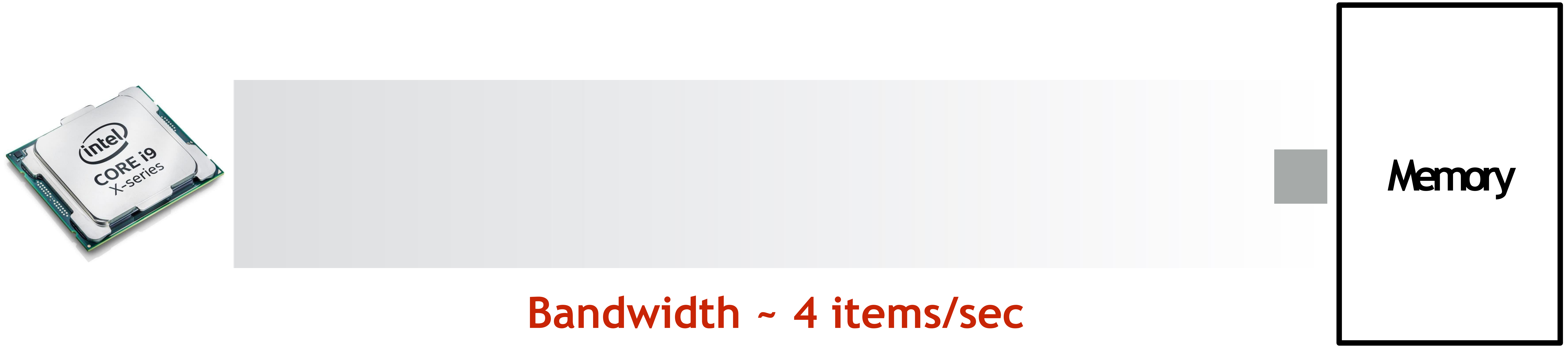
Throughput: 100 cars/hr (1 car every 1/100th of hour)



Throughput: 400 cars/hr (4 cars every 1/100th of hour)

Terminology

- Memory bandwidth
  - The rate at which the memory system can provide data to a processor
  - Example: 20 GB/s

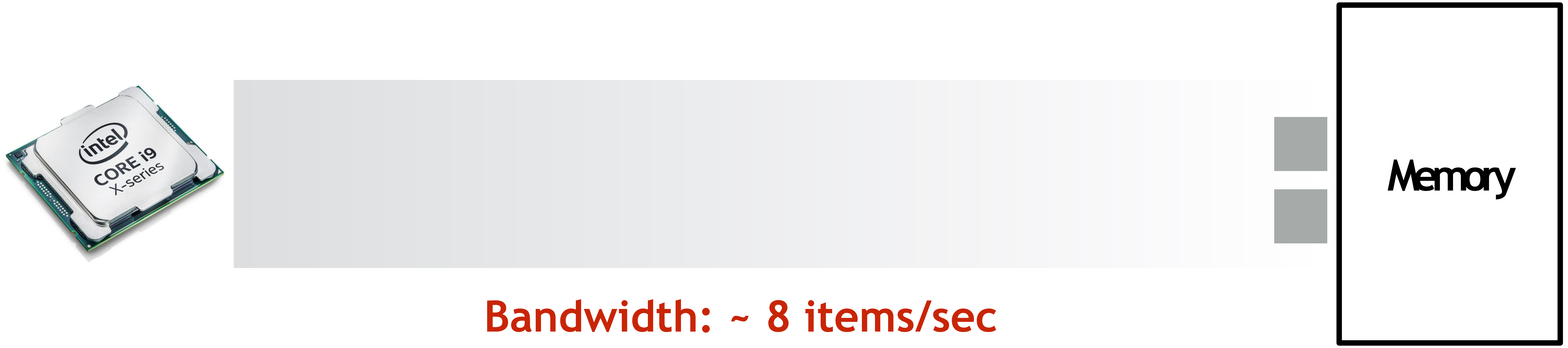


Bandwidth ~ 4 items/sec

Latency of transferring any one item: ~2 sec

Terminology

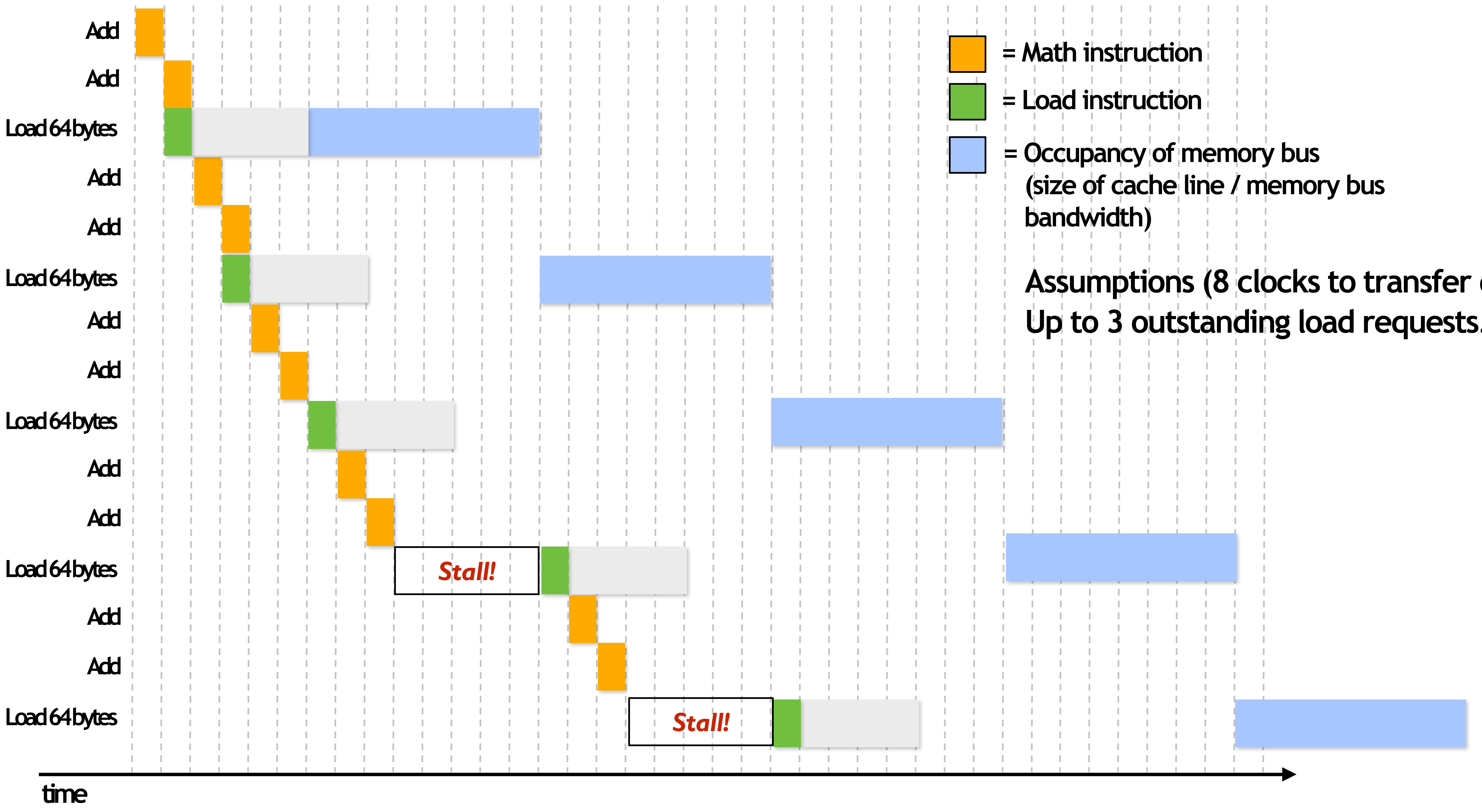
- Memory bandwidth
  - The rate at which the memory system can provide data to a processor
  - Example: 20 GB/s



Bandwidth: ~ 8 items/sec

Latency of transferring any one item: ~2 sec

Consider a processor that can do one add per clock (+ can co-issue LD)



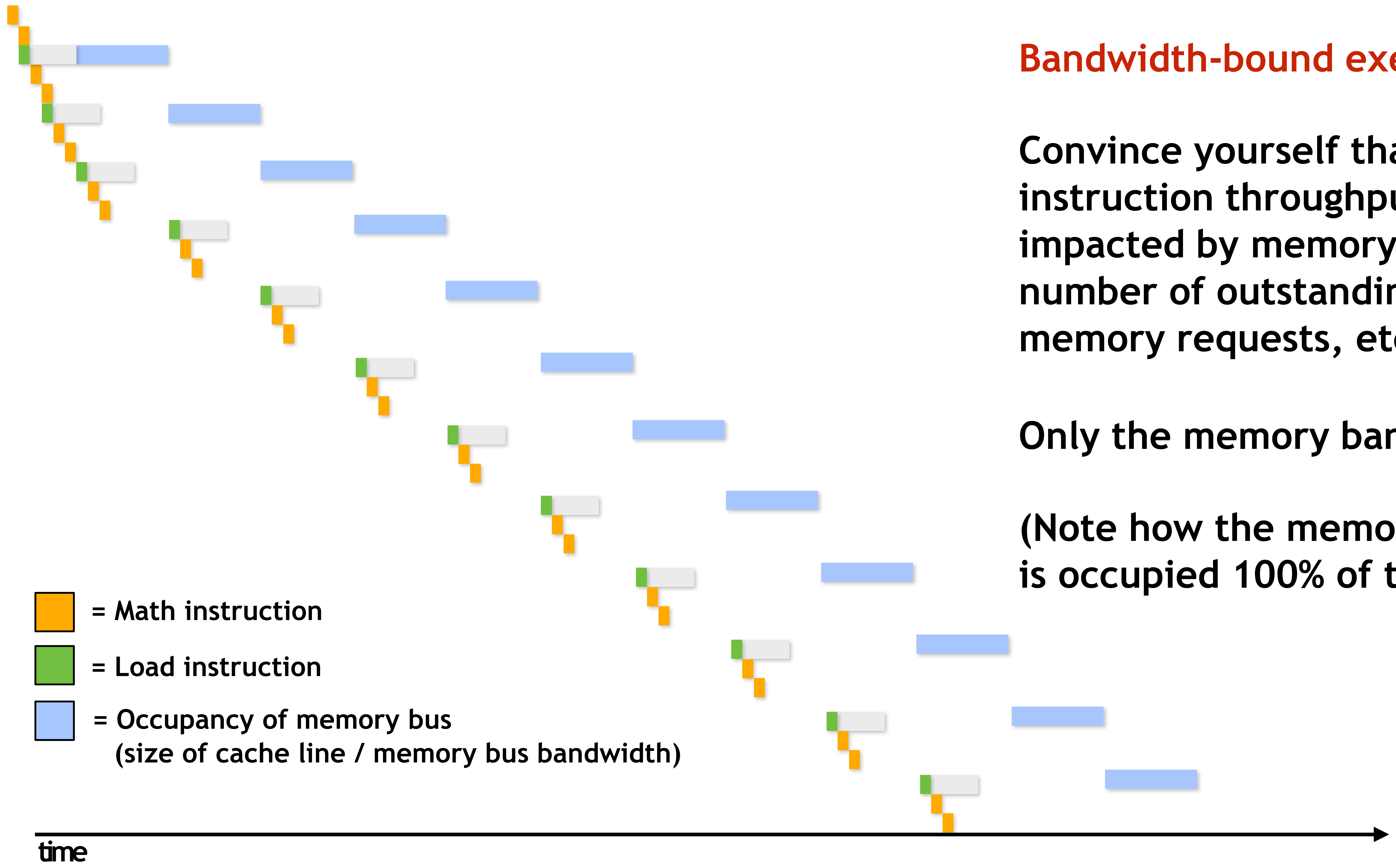
🟧 = Math instruction

🟩 = Load instruction

🟦 = Occupancy of memory bus (size of cache line / memory bus bandwidth)

Assumptions (8 clocks to transfer data)
Up to 3 outstanding load requests.

Rate of math instructions limited by available bandwidth



🟧 = Math instruction
🟩 = Load instruction
🟦 = Occupancy of memory bus (size of cache line / memory bus bandwidth)
time

Bandwidth-bound execution!

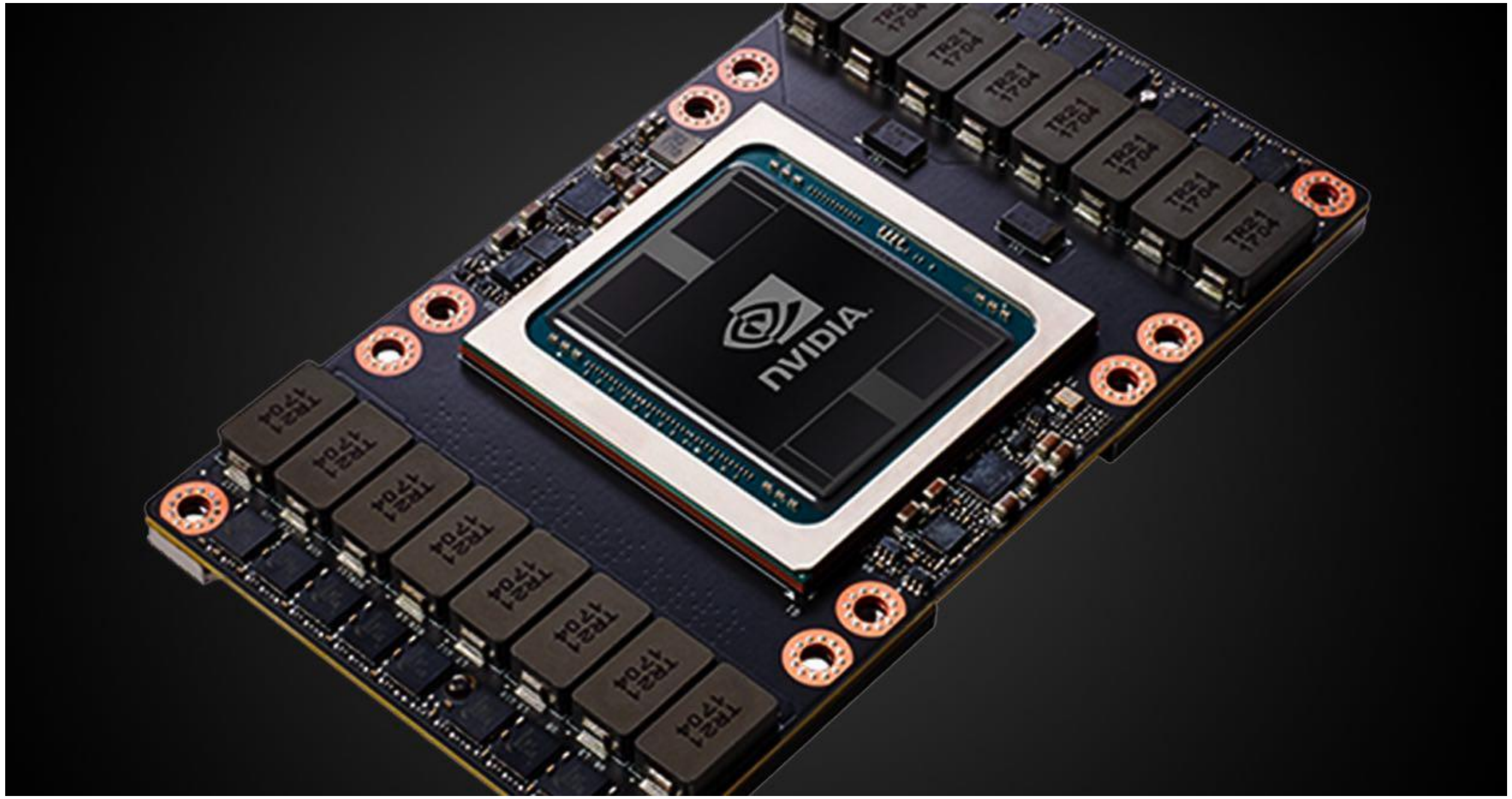
Convince yourself that the instruction throughput is not impacted by memory latency, number of outstanding memory requests, etc.

Only the memory bandwidth!!!

(Note how the memory system is occupied 100% of the time)

High bandwidth memories

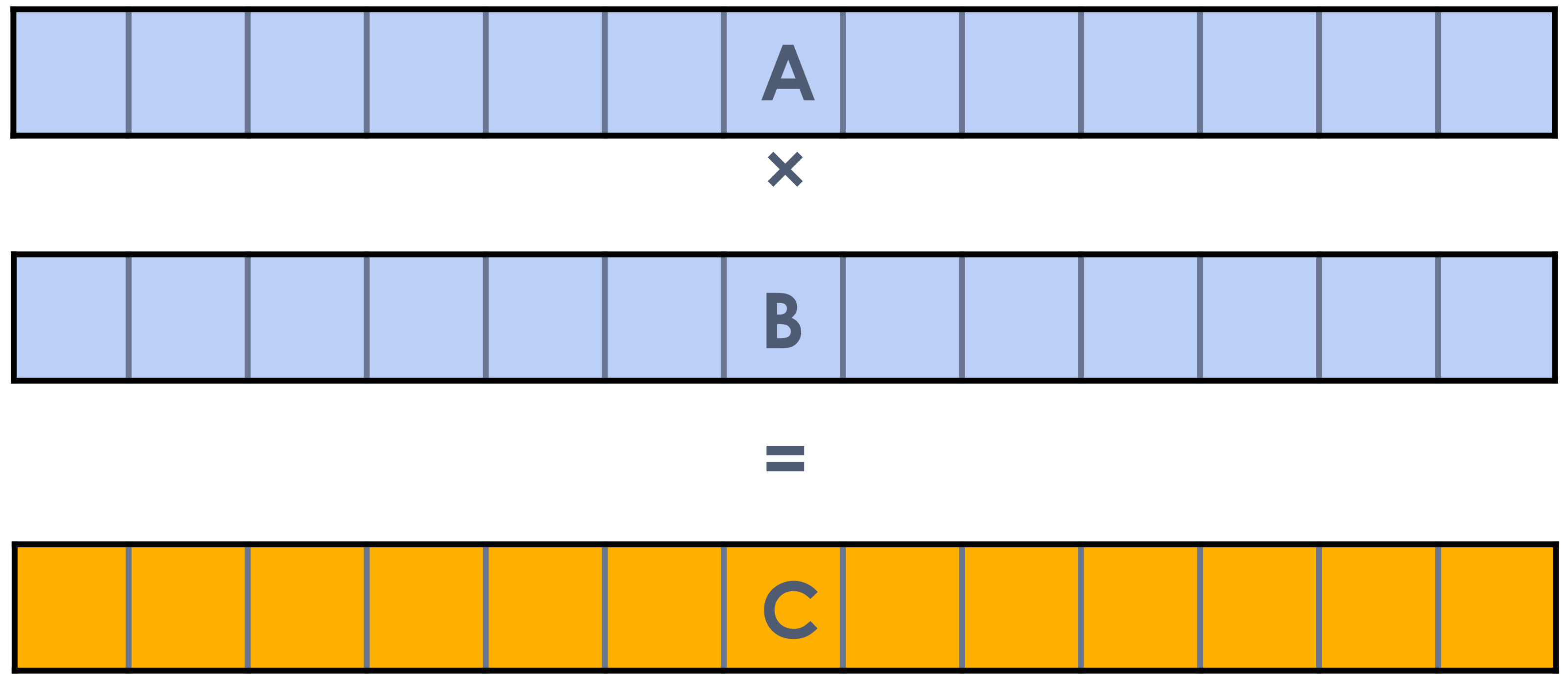
- Modern GPUs leverage high bandwidth memories located near processor
- Example:
  - V100 uses HBM2
  - 900 GB/s



Thought experiment

Task: 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] × B[i]
- Store result into C[i]



Three memory operations (12 bytes) for every MUL

NVIDIA V100 GPU can do 5120 fp32 MULs per clock

(@ 1.6 GHz) Need ~98 TB/sec of bandwidth to keep functional units busy

<1% GPU efficiency… but still 12x faster than eight-core CPU!

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

This computation is bandwidth limited!

If processors request data at too high a rate, the memory system cannot keep up.

Overcoming bandwidth limits is often the most important challenge facing software developers targeting modern throughput-optimized systems.

In modern computing, bandwidth is the critical resource

Performant parallel programs will:

- Organize computation to fetch data from memory less often
  - Reuse data previously loaded by the same thread (temporal locality optimizations)
  - Share data across threads (inter-thread cooperation)
- Favor performing additional arithmetic to storing/reloading values (the math is "free")
- Main point: programs must access memory infrequently to utilize modern processors efficiently

What we learned

- Modern parallel processors employ the following throughput computing ideas
  - Use multiple processing cores
    - Simpler cores (embrace parallelism across different threads)
  - Amortize instruction stream processing over many ALUs (SIMD)
    - Increase compute capability with little extra cost
  - Use multi-threading to increase utilization of processing resources

- GPU architectures use the same throughput computing ideas as CPUs
  - GPUs just push these concepts to extreme scales

- Due to high arithmetic capability on modern chips, many parallel applications are bandwidth bound (on both CPUs and GPUs)

Know these terms

- Instruction stream
- Multi-core processor
- SIMD execution
- Coherent control flow
- Hardware multi-threading
  - Interleaved multi-threading
  - Simultaneous multi-threading
- Memory latency
- Memory bandwidth
- Bandwidth bound application

REVIEW

HOW IT ALL FITS TOGETHER:

superscalar execution, SIMD execution, multi-core execution, and hardware multi-threading

Running code on a simple processor

C program source

```c
void sinx(int N, int terms, float* x, float* y)
{
   for (int i=0; i<N; i++)
   {
      float value = x[i];
      float numer = x[i] * x[i] * x[i];
      int denom = 6;  // 3!
      int sign = -1;

      for (int j=1; j<=terms; j++)
      {
         value += sign * numer / denom;
         numer *= x[i] * x[i];
         denom *= (2*j+2) * (2*j+3);
         sign *= -1;
      }

      y[i] = value;
   }
}
```

compiler

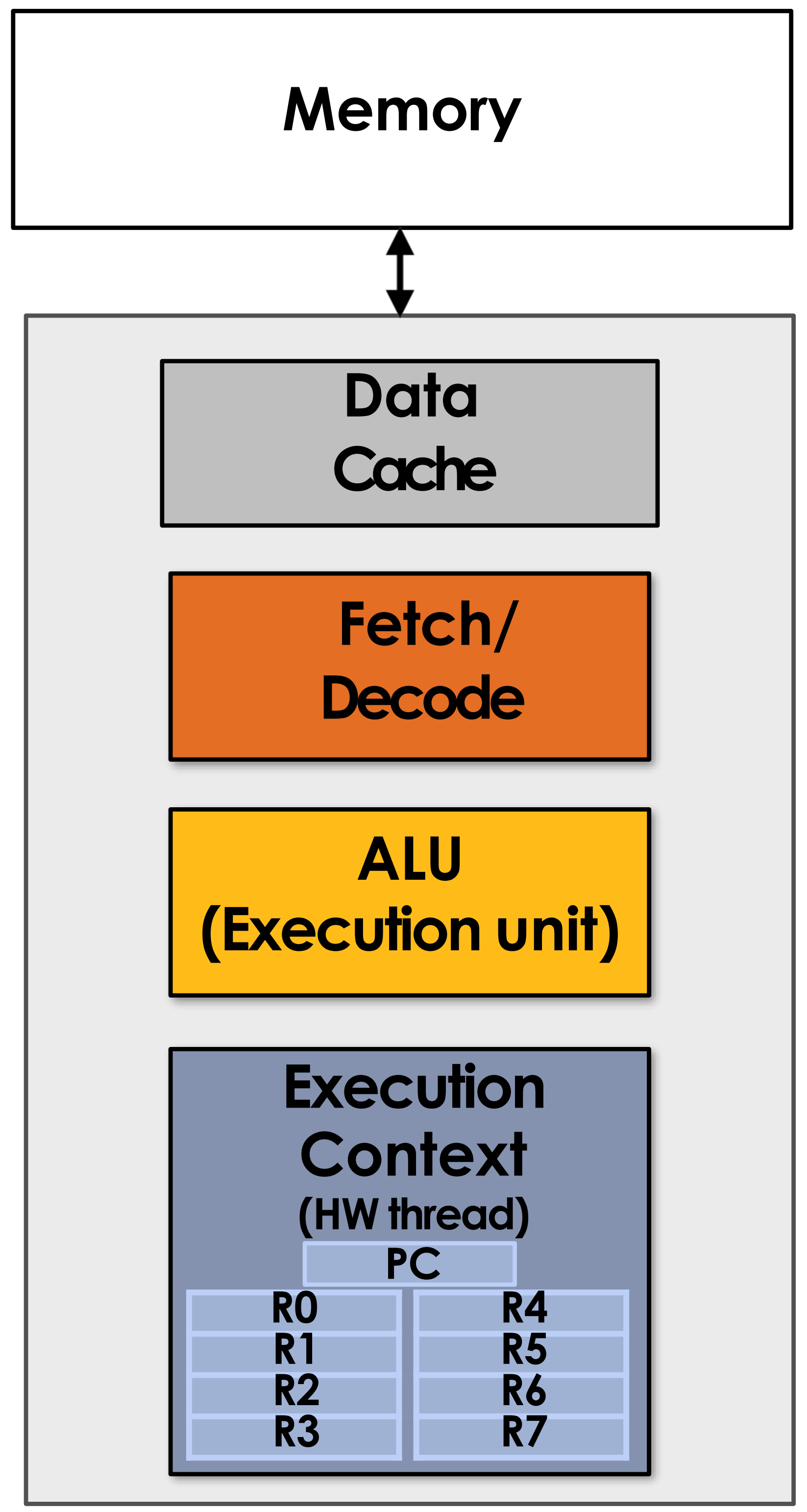
Compiled instruction stream (scalar instructions)

```
ld    r0, addr[r1]
mul   r1, r0, r0
add   r2, r0, r0
mul   r3, r1, r2
...
...
...
...
...
st    addr[r2], r0
```

Running code on a simple processor

Instruction stream

```
ld   r0, addr[r1]
mul  r1, r0, r0
add  r2, r0, r0
mul  r3, r1, r2
...
...
...
...
...
st   addr[r2], r0
```

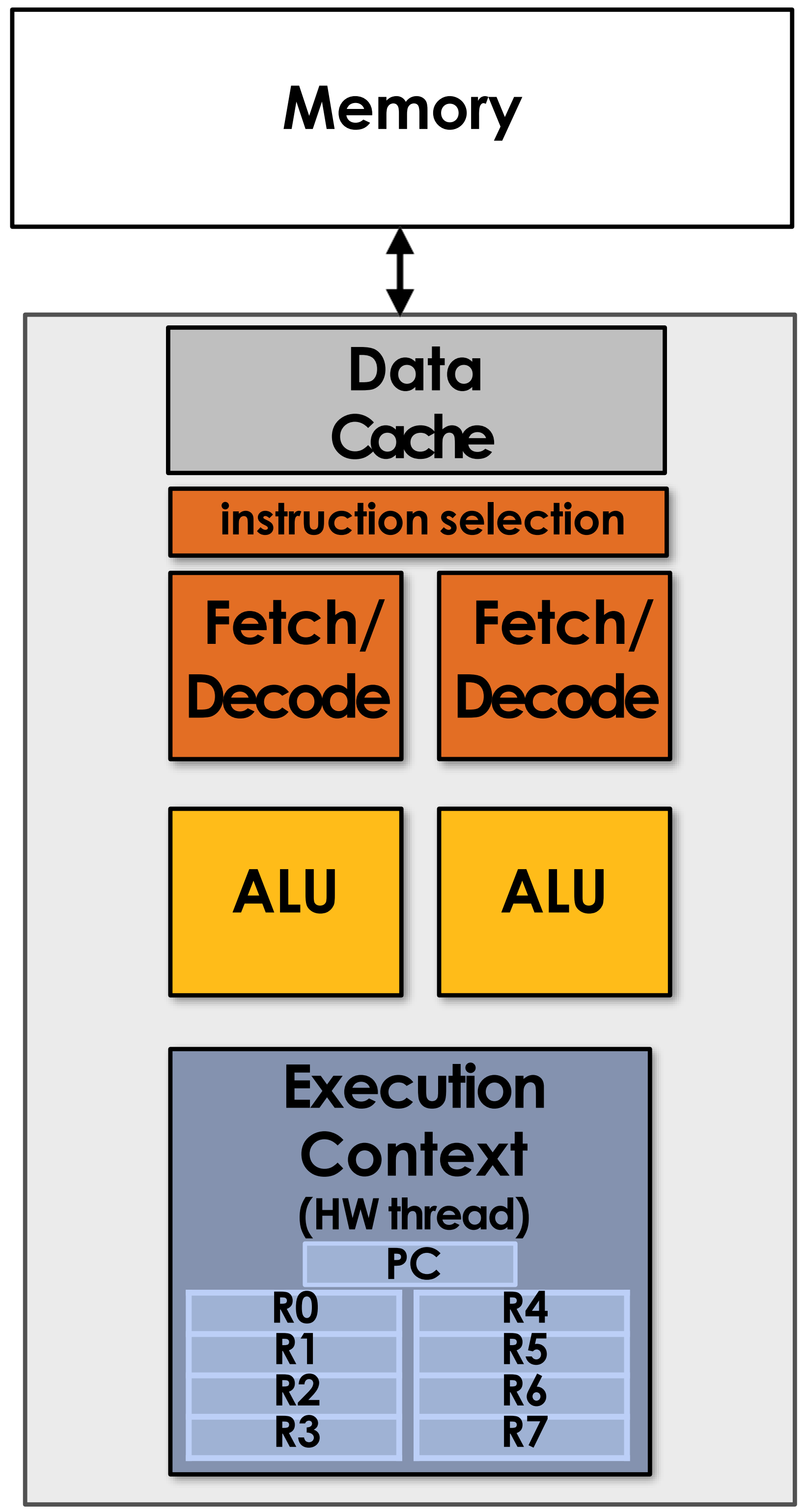


Single core processor, single-threaded core.
Can run one scalar instruction per clock

Superscalar core

Instruction stream

```
ld   r0, addr[r1]
mul  r1, r0, r0
add  r2, r0, r0
mul  r3, r1, r2
...
...
...
...
...
st   addr[r2], r0
```

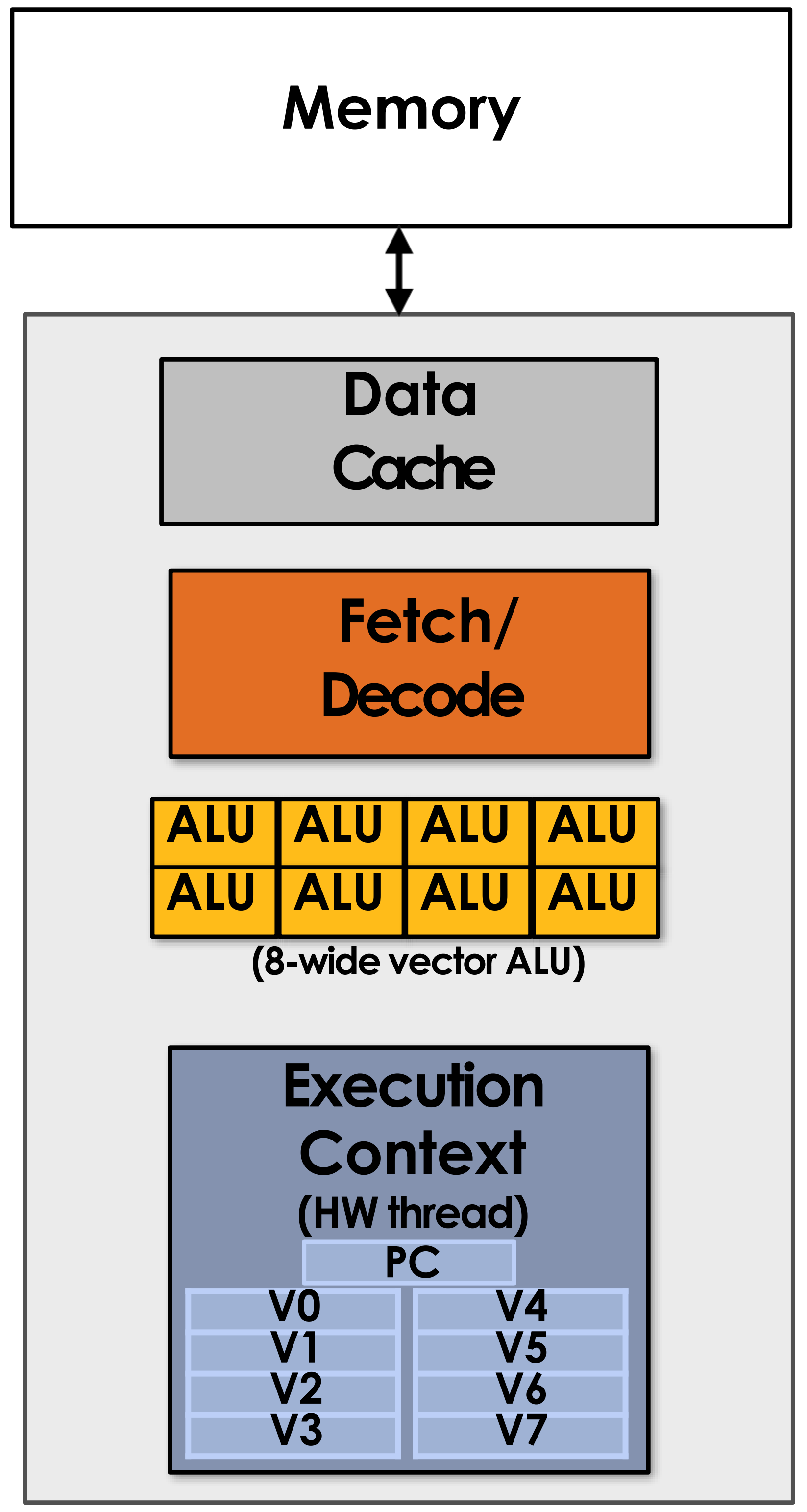


Single core processor, single-threaded core.
Two-way superscalar core:
can run up to two independent scalar instructions
per clock from one instruction stream (one hardware thread)

SIMD execution capability

Instruction stream
(now with vector instructions)

```
vector_ld    v0, vector_addr[r1]
vector_mul   v1, v0, v0
vector_add   v2, v0, v0
vector_mul   v3, v1, v2
...
...
...
...
...
vector_st    addr[r2], v0
```

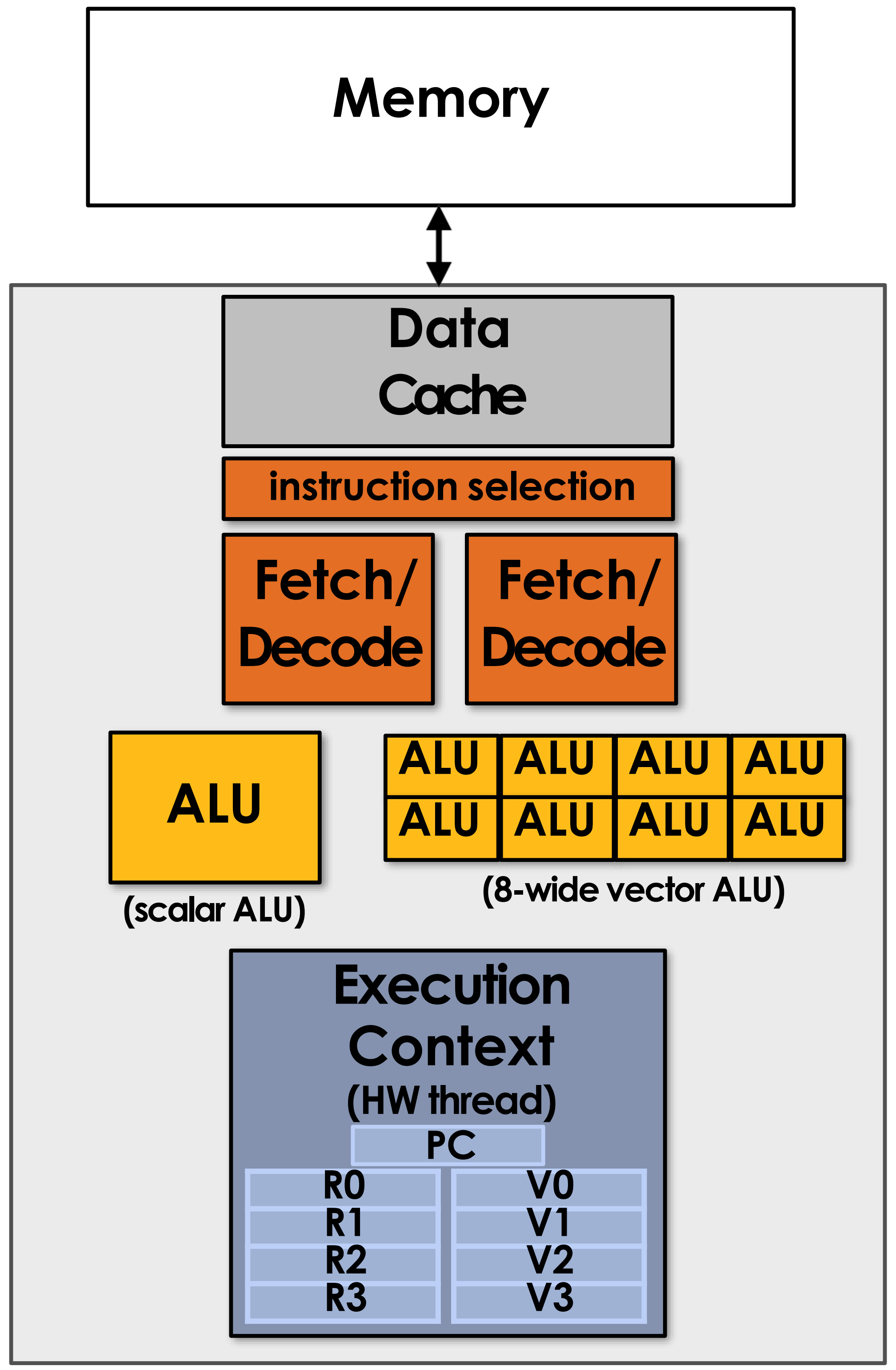


Single core processor, single-threaded core. can run one 8-wide SIMD vector instruction from one instruction stream

Heterogeneous superscalar (scalar + SIMD)

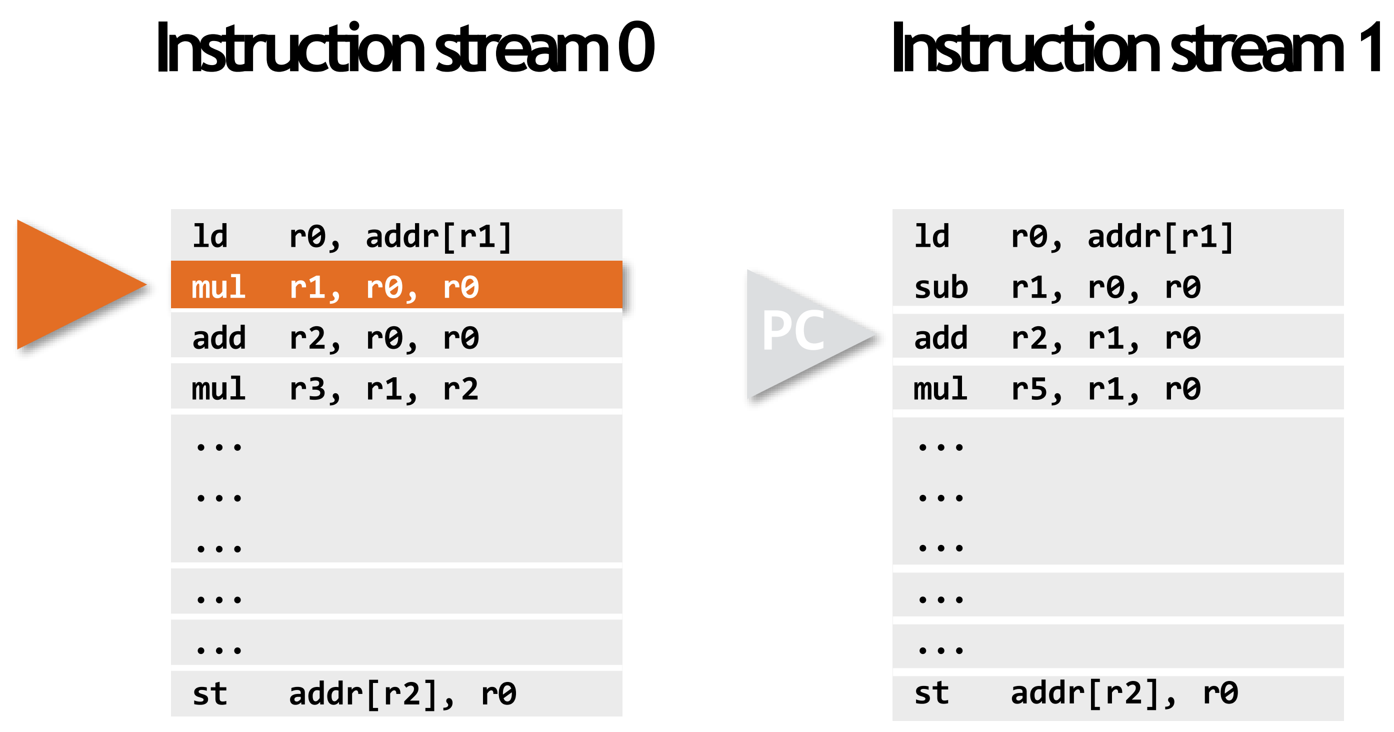
Instruction stream

```
vector_ld    v0, vector_addr[r1]
vector_mul   v1, v0, v0
add          r2, r1, r0
vector_add   v2, v0, v0
vector_mul   v3, v1, v2...
...
...
...
...
vector_st    addr[r2], v0
```



Single core processor, single-threaded core.
Two-way superscalar core:
can run up to two independent instructions per clock from one instruction stream, provided one is scalar and the other is vector

Multi-threaded core



Instruction stream 0

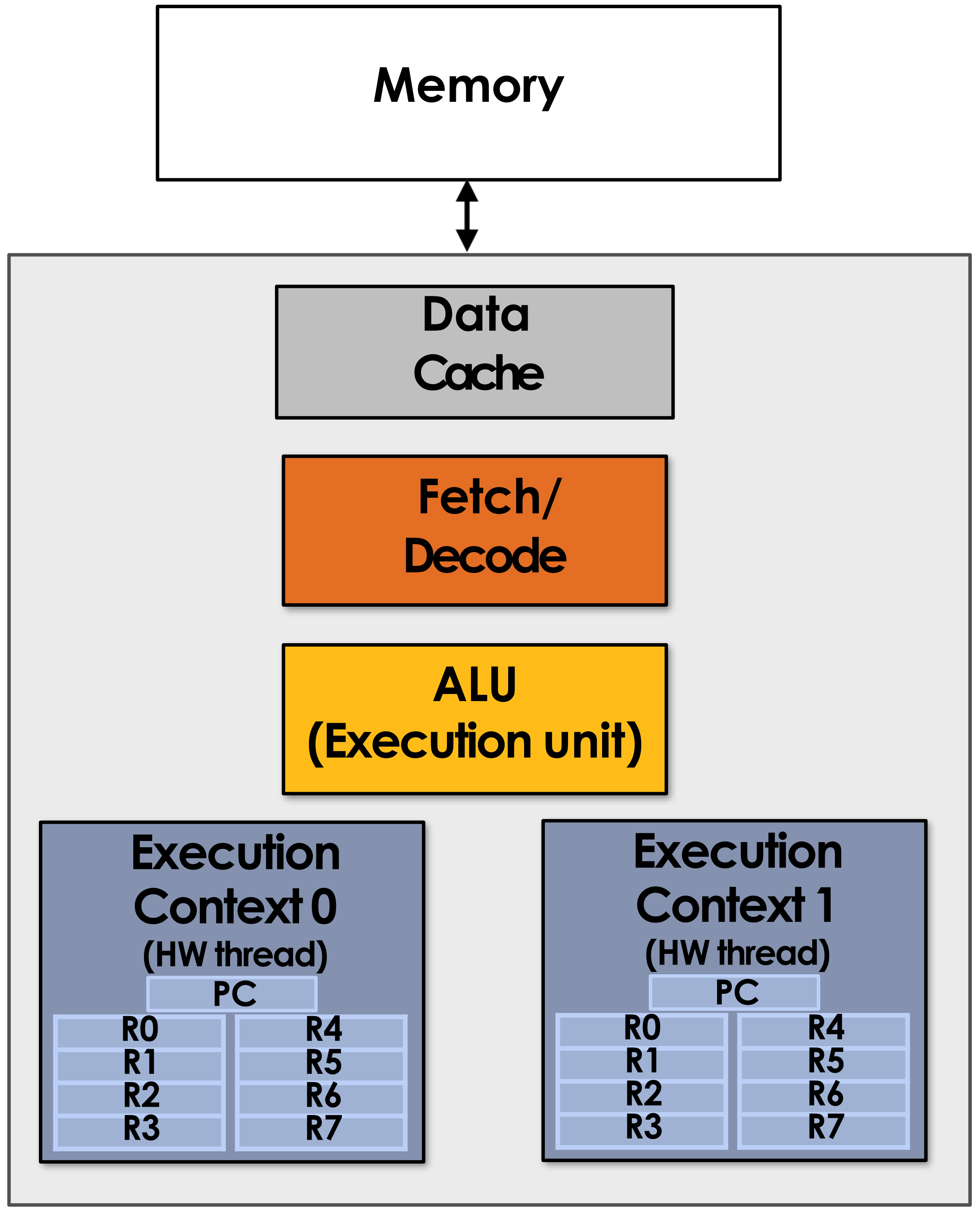
```
ld   r0, addr[r1]
mul  r1, r0, r0
add  r2, r0, r0
mul  r3, r1, r2
...
...
...
...
...
st   addr[r2], r0
```

Instruction stream 1

```
ld   r0, addr[r1]
sub  r1, r0, r0
add  r2, r1, r0
mul  r5, r1, r0
...
...
...
...
...
st   addr[r2], r0
```

Note: threads can be running completely different instruction streams (and be at different points in these streams)

Execution of hardware threads is interleaved in time.



Single core processor, multi-threaded core (2 threads). Can run one scalar instruction per clock from one of the instruction streams (hardware threads)

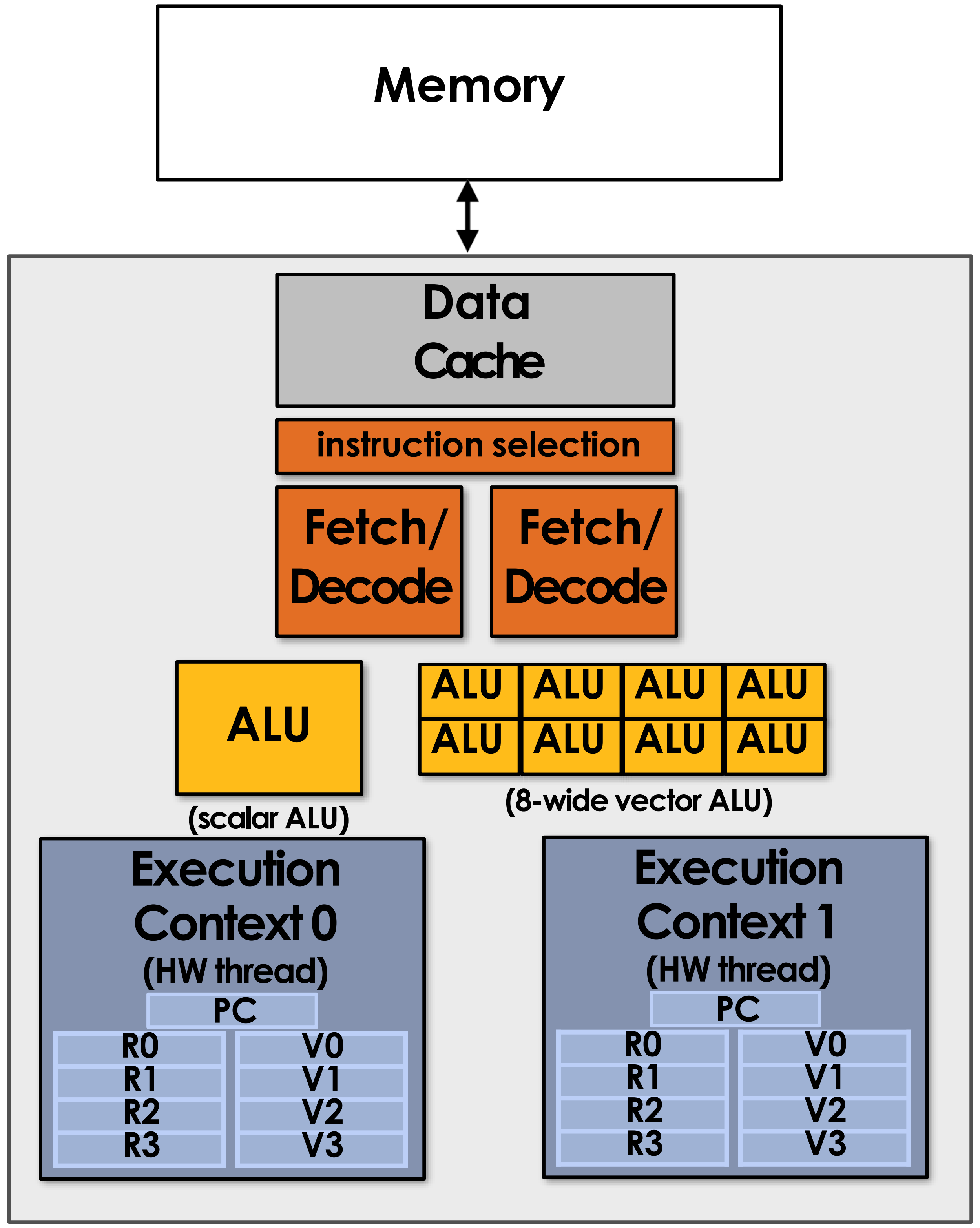
Multi-threaded, superscalar core

Instruction stream 0

```
vector_ld     v0, addr[r1]
vector_mul    v1, v0, v0
vector_add    v2, v1, v1
mul           r2, r1, r1
...
...
...
...
...
vector_st     addr[r2], v0
```

Instruction stream 1

```
vector_ld     v0, addr[r1]
sub           r1, r0, r0
vector_add    v2, v0, v0
mul           r5, r1, r0
...
...
...
...
...
rect          addr[r2], v0
```



Note: threads can be running completely different instruction streams (and be at different points in these streams)

Execution of hardware threads is interleaved in time.

Single core processor, multi-threaded core (2 threads).

Two-way superscalar core: in this example I defined my core as being capable of running up to two independent instructions per clock from a single instruction stream*, provided one is scalar and the other is vector

* This detail was an arbitrary decision on this slide: a different implementation of "instruction selection" might run two instructions where one is drawn from each thread, see next slide.

Multi-threaded, superscalar core

(that combines interleaved and simultaneous execution of multiple hardware threads)

Instruction stream 0

```
vector_ld      v0, addr[r1]
vector_mul     v1, v0, v0
vector_add     v2, v1, v1
mul            r2, r1, r1
...
...
...
...
...
vector_st      addr[r2], v0
```

Instruction stream 1

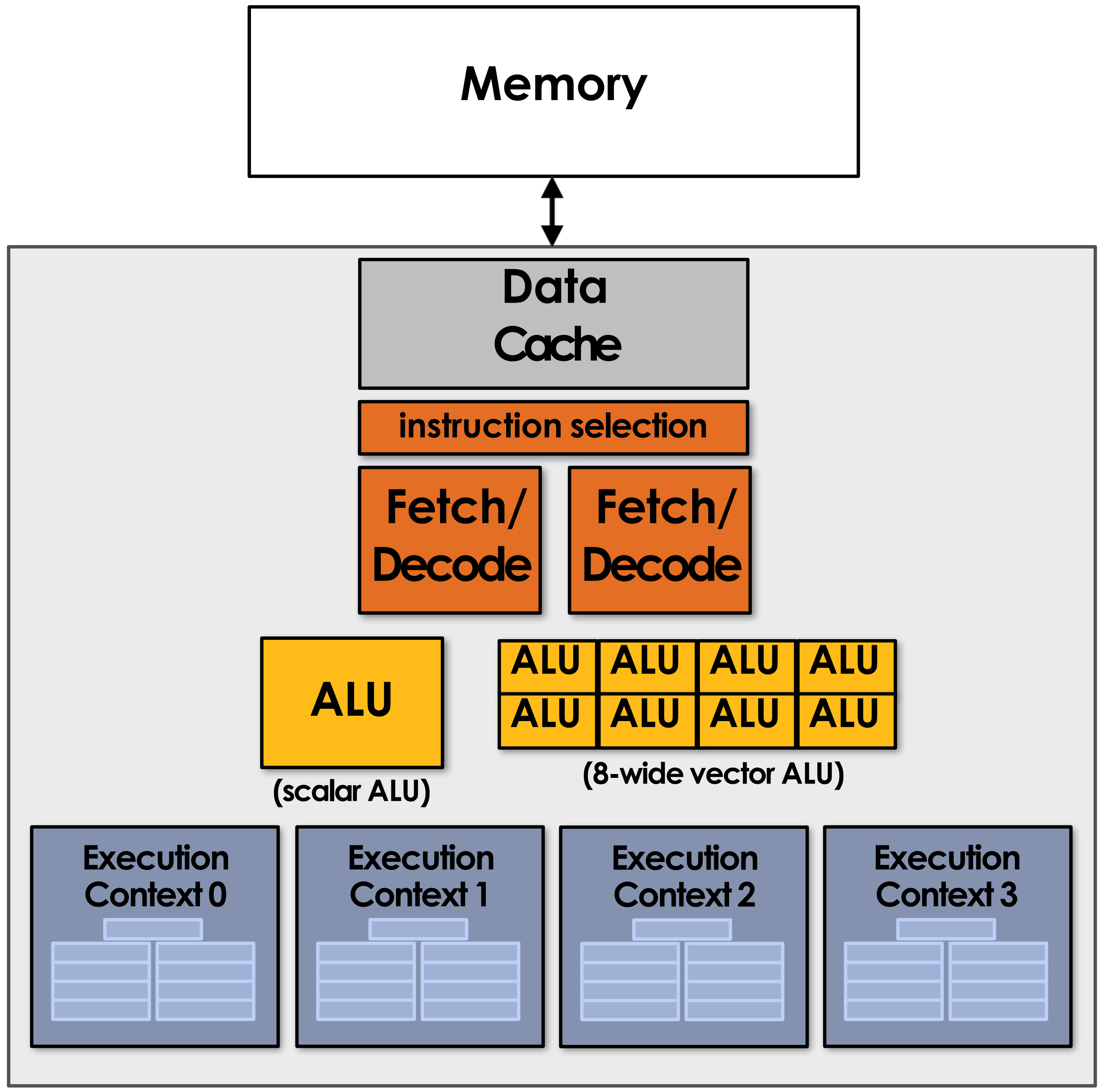
```
vector_ld      v0, addr[r1]
sub            r1, r0, r0
vector_add     v2, v0, v0
mul            r5, r1, r0
...
...
...
...
...
rect           addr[r2], v0
```

Instruction stream 2

```
vector_ld      v0, addr[r1]
vector_mul     v2, v0, v0
mul            r3, r0, r0
sub            r1, r0, r3 ...
...
...
...
...
rect           addr[r2], v0
```

Instruction stream 3

```
vector_ld      v0, addr[r1]
sub            r1, r0, r0
vector_add     v1, v0, v0
vector_add     v2, v0, v1
mul            r2, r1, r1
...
...
...
...
rect           addr[r2], v0
```



Execution of hardware threads may or may not be interleaved in time

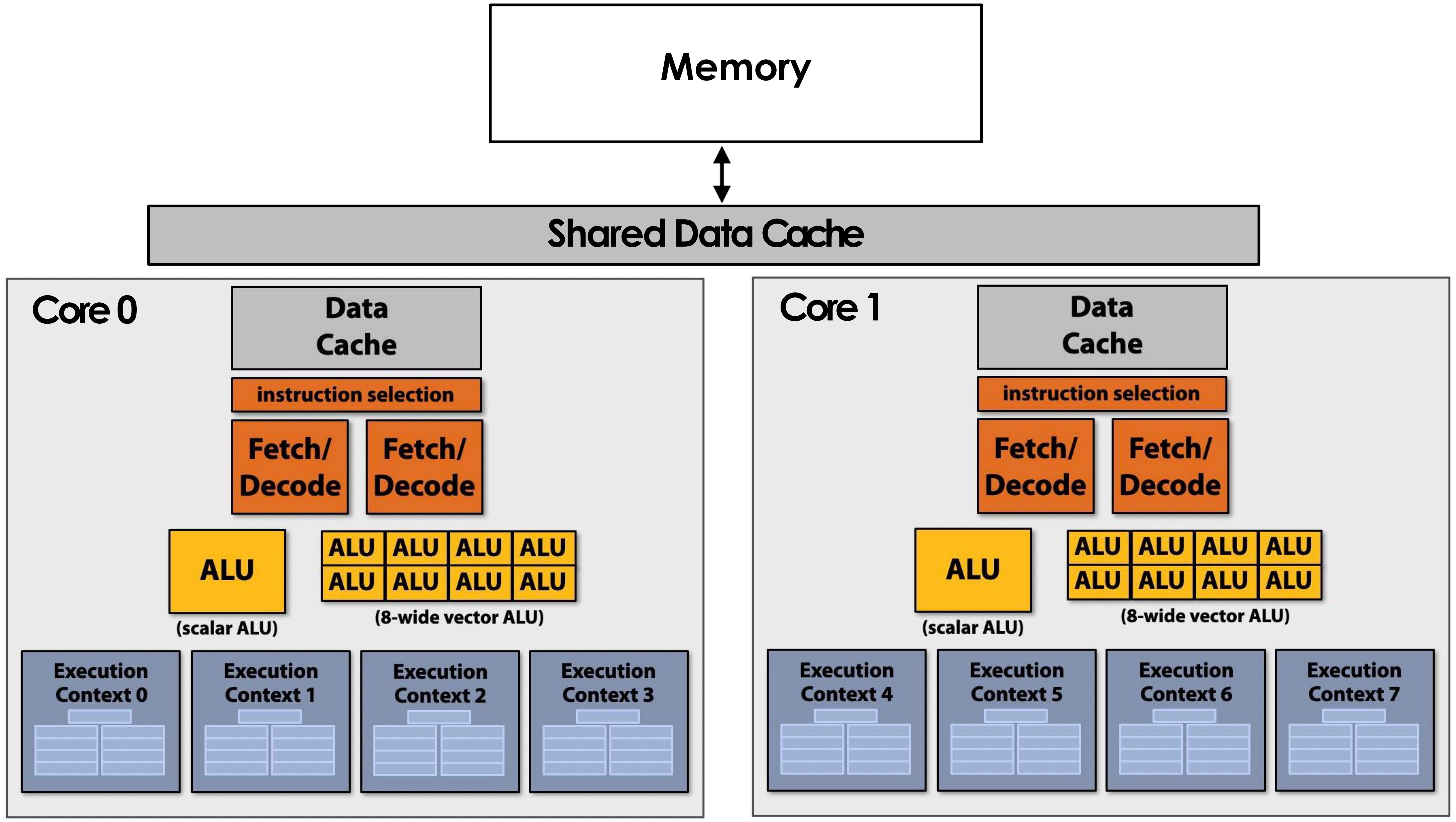
(instructions from different threads may be running simultaneously)

Single core processor, multi-threaded core (4 threads).

Two-way superscalar core:

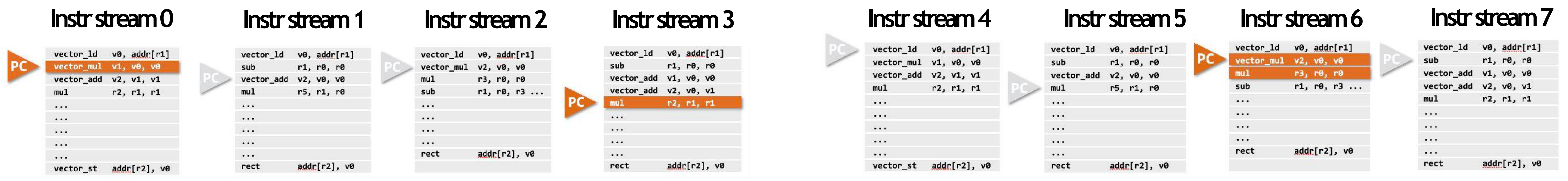
can run up to two independent instructions per clock from <u>any of the threads</u>, <u>provided one is scalar and the other is vector</u>

Multi-core, with multi-threaded, superscalar cores



Dual-core processor, multi-threaded cores (4 threads/core).

Two-way superscalar cores: each core can run up to two independent instructions per clock from any of its threads, provided one is scalar and the other is vector



Instr stream 0

```
vector_ld    v0, addr[r1]
vector_mul   v1, v0, v0
vector_add   v2, v1, v1
mul          r2, r1, r1
...
...
...
...
...
vector_st    addr[r2], v0
```

Instr stream 1

```
vector_ld    v0, addr[r1]
sub          r1, r0, r0
vector_add   v2, v0, v0
mul          r5, r1, r0
...
...
...
...
...
rect         addr[r2], v0
```

Instr stream 2

```
vector_ld    v0, addr[r1]
vector_mul   v2, v0, v0
mul          r3, r0, r0
sub          r1, r0, r3 ...
...
...
...
...
rect         addr[r2], v0
```

Instr stream 3

```
vector_ld    v0, addr[r1]
sub          r1, r0, r0
vector_add   v1, v0, v0
vector_add   v2, v0, v1
mul          r2, r1, r1
...
...
...
...
rect         addr[r2], v0
```

Instr stream 4

```
vector_ld    v0, addr[r1]
vector_mul   v1, v0, v0
vector_add   v2, v1, v1
mul          r2, r1, r1
...
...
...
...
...
vector_st    addr[r2], v0
```

Instr stream 5

```
vector_ld    v0, addr[r1]
sub          r1, r0, r0
vector_add   v2, v0, v0
mul          r5, r1, r0
...
...
...
...
...
rect         addr[r2], v0
```

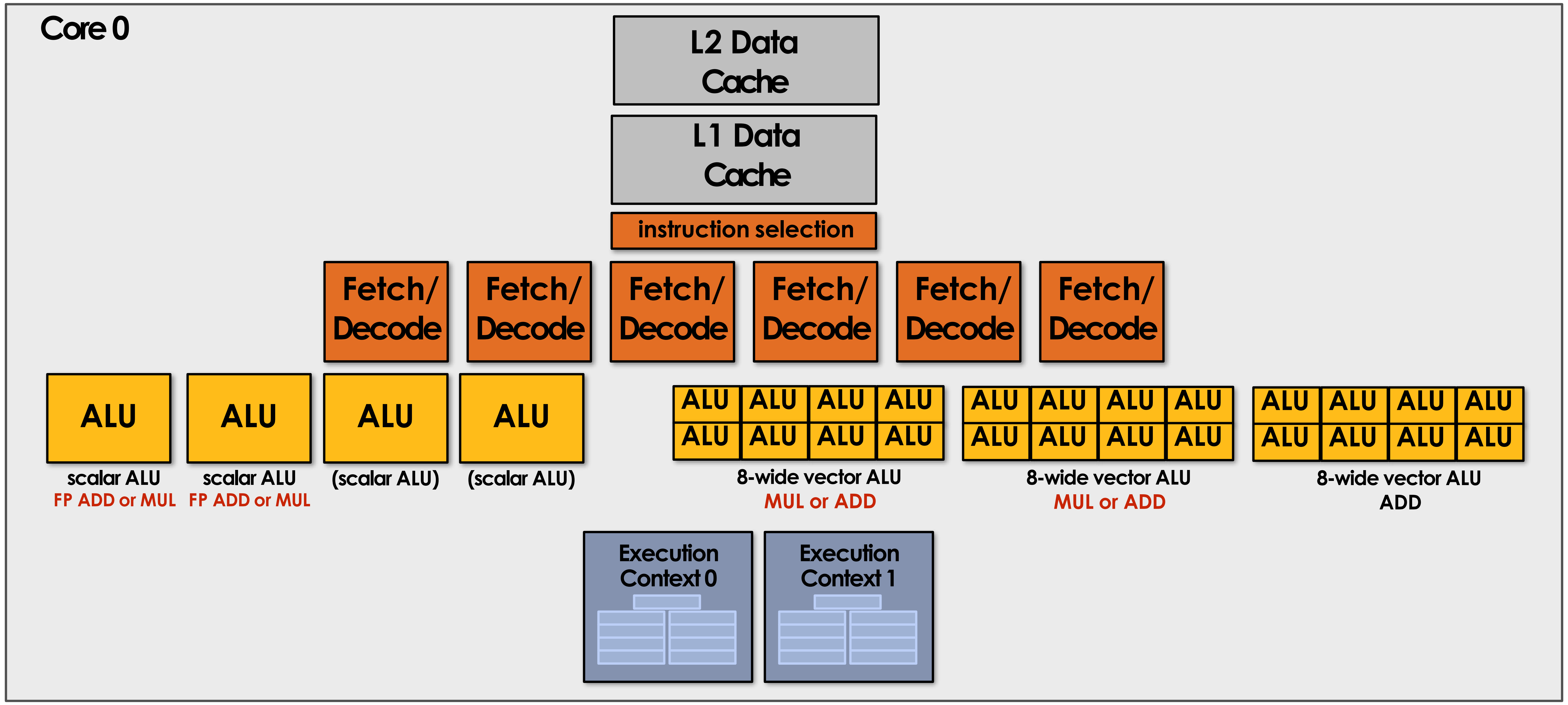
Instr stream 6

```
vector_ld    v0, addr[r1]
vector_mul   v2, v0, v0
mul          r3, r0, r0
sub          r1, r0, r3 ...
...
...
...
...
rect         addr[r2], v0
```

Instr stream 7

```
vector_ld    v0, addr[r1]
sub          r1, r0, r0
vector_add   v1, v0, v0
vector_add   v2, v0, v1
mul          r2, r1, r1
...
...
...
...
rect         addr[r2], v0
```

Example: Intel Skylake/Kaby Lake core

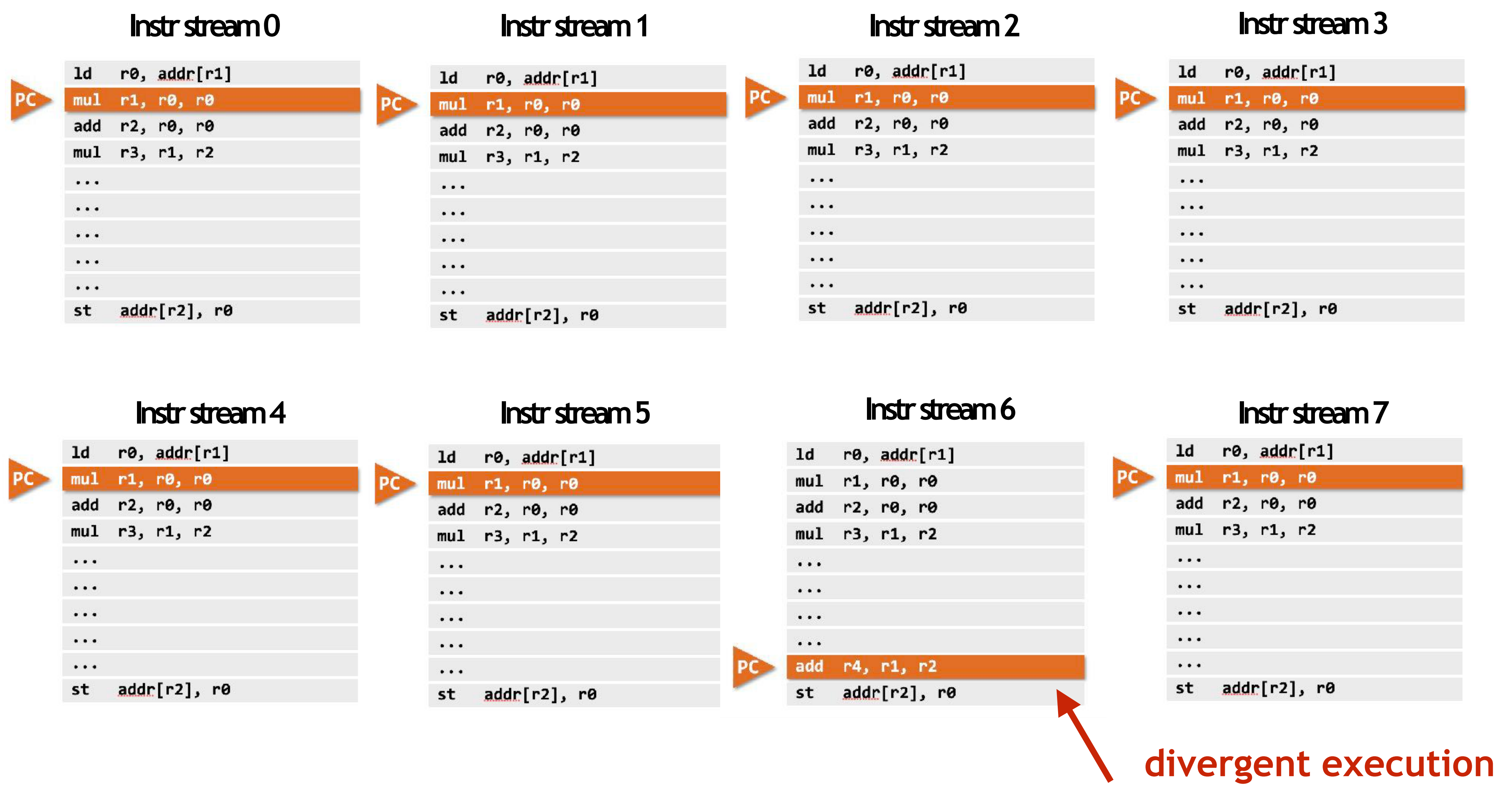


Two-way multi-threaded cores (2 threads).

Each core can run up to four independent scalar instructions and up to three 8-wide vector instructions (up to 2 vector mul or 3 vector add)

Not shown on this diagram: units for LD/ST operations

GPU “SIMT” (single instruction multiple thread)



Instr stream 0

```
ld    r0, addr[r1]
mul   r1, r0, r0
add   r2, r0, r0
mul   r3, r1, r2
...
...
...
...
...
st    addr[r2], r0
```

Instr stream 1

```
ld    r0, addr[r1]
mul   r1, r0, r0
add   r2, r0, r0
mul   r3, r1, r2
...
...
...
...
...
st    addr[r2], r0
```

Instr stream 2

```
ld    r0, addr[r1]
mul   r1, r0, r0
add   r2, r0, r0
mul   r3, r1, r2
...
...
...
...
...
st    addr[r2], r0
```

Instr stream 3

```
ld    r0, addr[r1]
mul   r1, r0, r0
add   r2, r0, r0
mul   r3, r1, r2
...
...
...
...
...
st    addr[r2], r0
```

Instr stream 4

```
ld    r0, addr[r1]
mul   r1, r0, r0
add   r2, r0, r0
mul   r3, r1, r2
...
...
...
...
...
st    addr[r2], r0
```

Instr stream 5

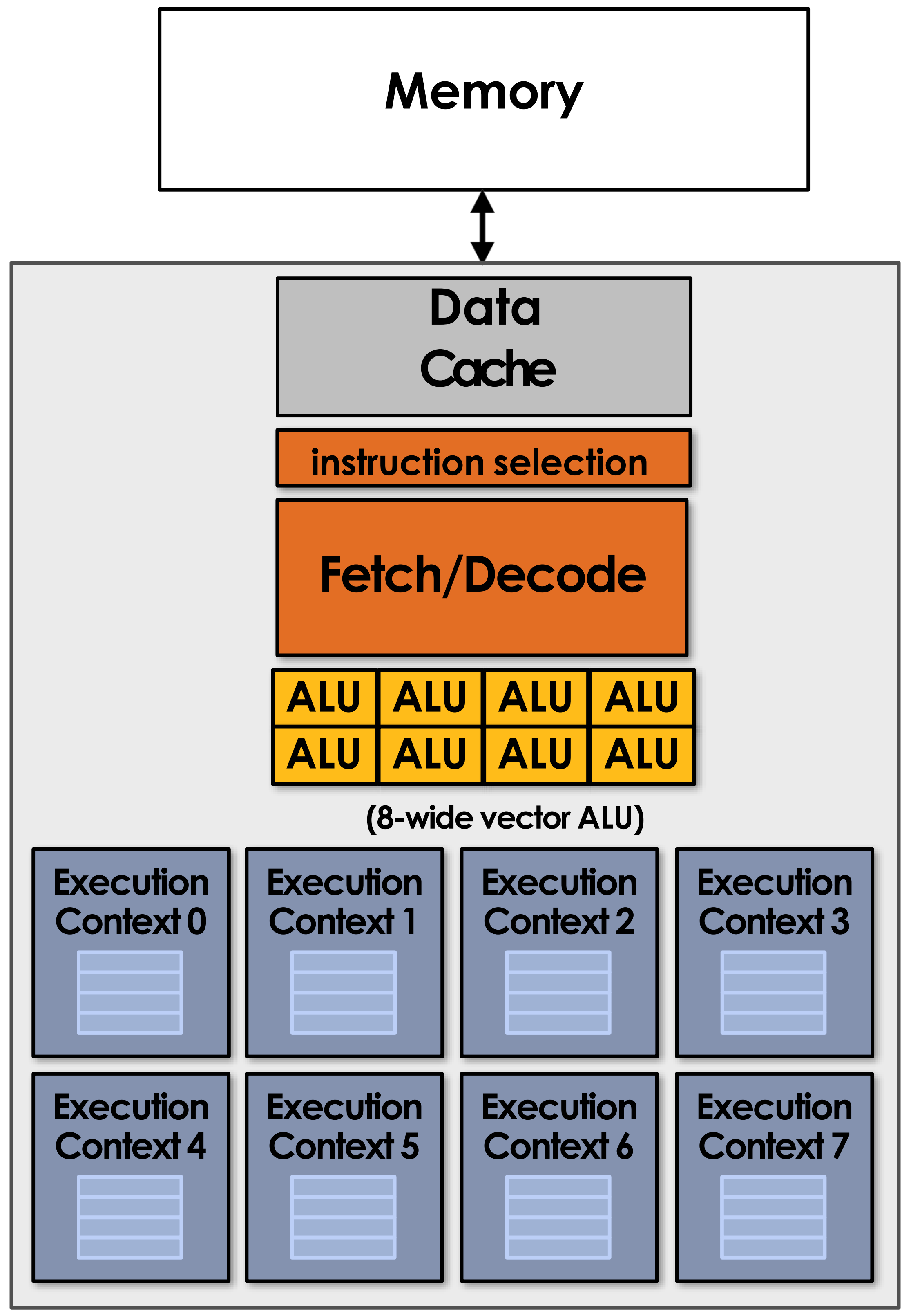
```
ld    r0, addr[r1]
mul   r1, r0, r0
add   r2, r0, r0
mul   r3, r1, r2
...
...
...
...
...
st    addr[r2], r0
```

Instr stream 6

```
ld    r0, addr[r1]
mul   r1, r0, r0
add   r2, r0, r0
mul   r3, r1, r2
...
...
...
...
add   r4, r1, r2
st    addr[r2], r0
```

Instr stream 7

```
ld    r0, addr[r1]
mul   r1, r0, r0
add   r2, r0, r0
mul   r3, r1, r2
...
...
...
...
...
st    addr[r2], r0
```



Many modern GPUs execute hardware threads that run instruction streams with only scalar instructions.

GPU cores detect when different hardware threads are executing the same instruction, and implement simultaneous execution of up to SIMD-width threads using SIMD ALUs.

Here ALU 6 would be “masked off” since thread 6 is not executing the same instruction as the other hardware threads.