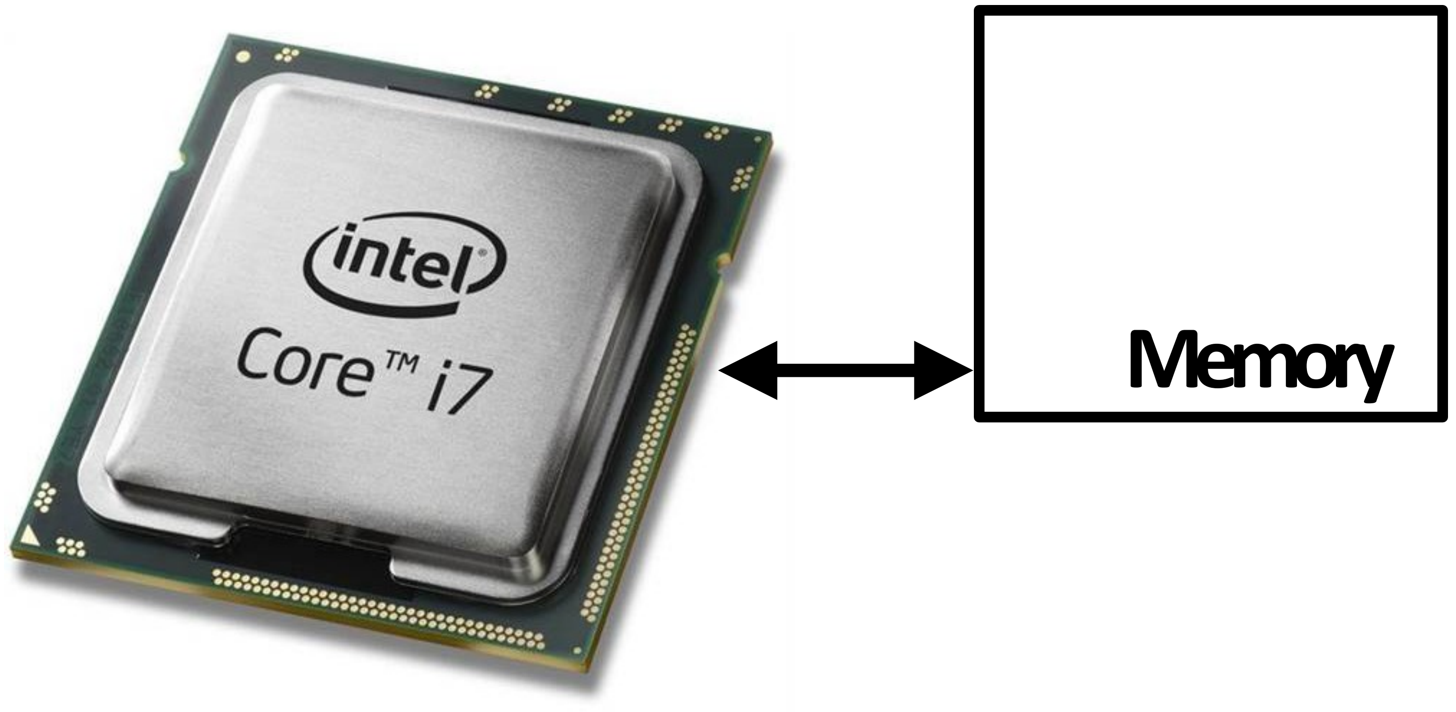
Memory



Terminology

- Memory latency
  - The amount of time for a memory request (e.g., load, store) from a processor to be serviced by the memory system
  - Example: 100 cycles, 100 nsec

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

Memory Technologies

- Cost (what can 200$ buy today 2010?)
  - SRAM 16MB
  - DRAM 4,000MB (8GB), 500x cheaper than SRAM
  - Flash 64,000 (128GB) - 16x cheaper than DRAM
  - Disk 2,000,000 (2TB) - 16x cheaper than Flash
- Latency
  - SRAM <1 to 2ns (on-chip)
  - DRAM ~50ns - 100x or more slower than SRAM
  - Flash 75,000 ns (75 µs) - 1500x vs DRAM
  - Disk 10,000,000 (10ms) - 133x vs FLash
- Bandwidth
  - SRAM 300 GB/s (12 port, 8 byte at 3 Ghz)
  - DRAM 25 GB/s ;
  - Flash 0.25 GB/s ; Disk 100MB/s

$\downarrow$

Ideally, one would desire an infinitely large memory capacity such that any particular word would be immediately available ... We are forced to recognize the possibility of constructing a hierarchy of memories, each of which has a greater capacity than the preceding but which is less quickly accessible.”

Burks, Goldstine, VonNeumann
“Preliminary discussion of the logical design of an electronic computing instrument”
IAS memo 1946

Exploiting Locality

- Locality of memory references
  - interesting property of real programs; few exceptions
- Temporal Locality
  - recently referenced data likely to be used again
  - keep data in small and fast storage (**reactive**)
- Spatial Locality
  - Likely to access data near each other
  - fetch data in chunks (**Proactive**)

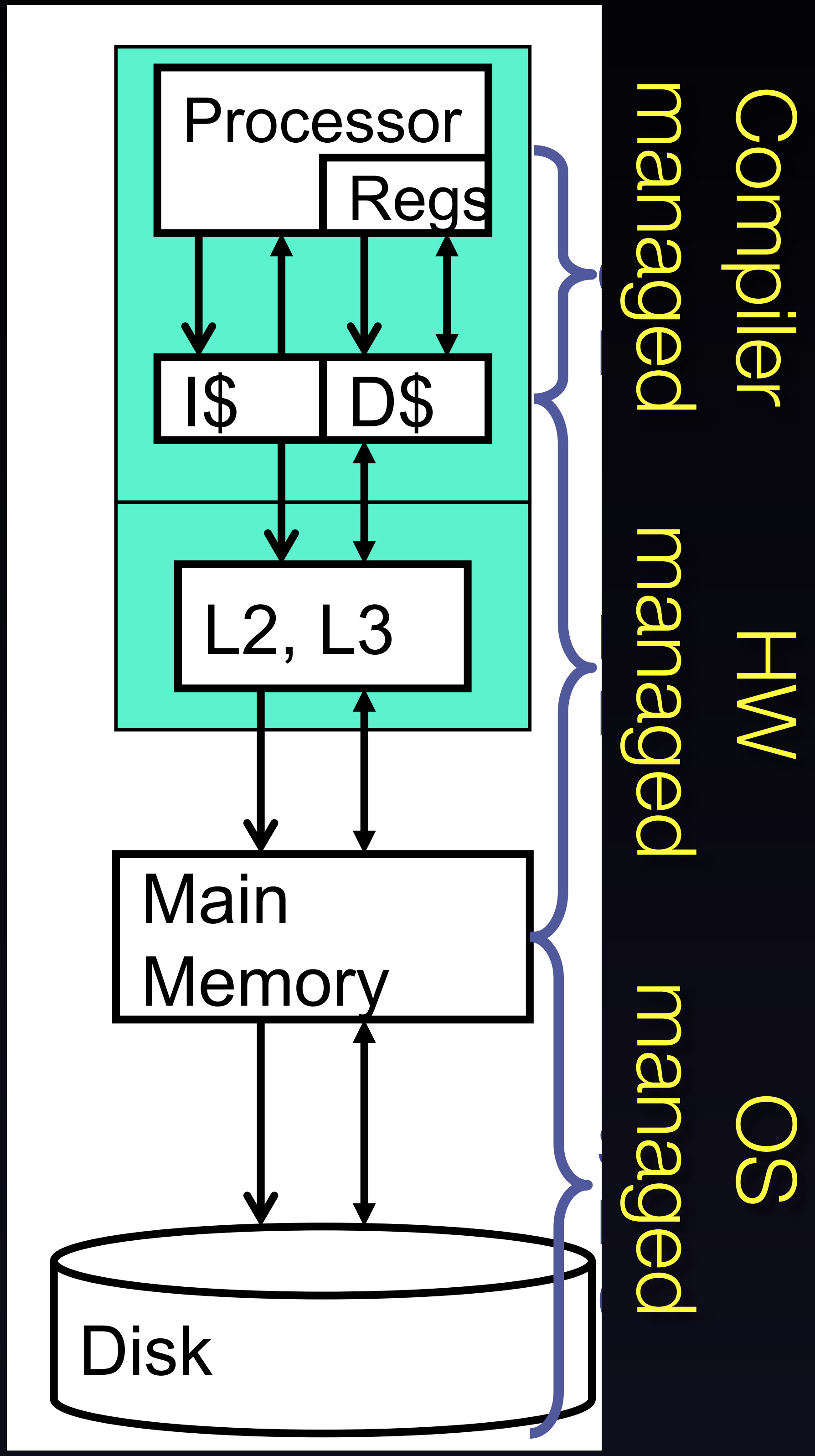
Library Analogy

- Consider books in library
  - library has lots of books, but slow
  - far away (time to walk to library)
  - big (time to walk within library)
- How can you avoid latencies
  - check out books and put them on desk (limited capacity)
  - keep recently used books around (**Temporal locality**)
  - keep books on related topic together (**Spatial locality)**
  - Guess what books will be needed in the future (prefetching)

Memory Hierarchy: Exploiting Locality

- Hierarchy of memory components
  - Upper components; Fast, Small, expensive
  - Lower components; Slow, Big, Cheap
- Most frequently seen data in M1
  - move data up and down the hierarchy
- Optimize
  - Avg. Latency = $\text{Latency}_{\text{hit}}$ + %miss * $\text{Latency}_{\text{miss}}$

Memory Hierarchy

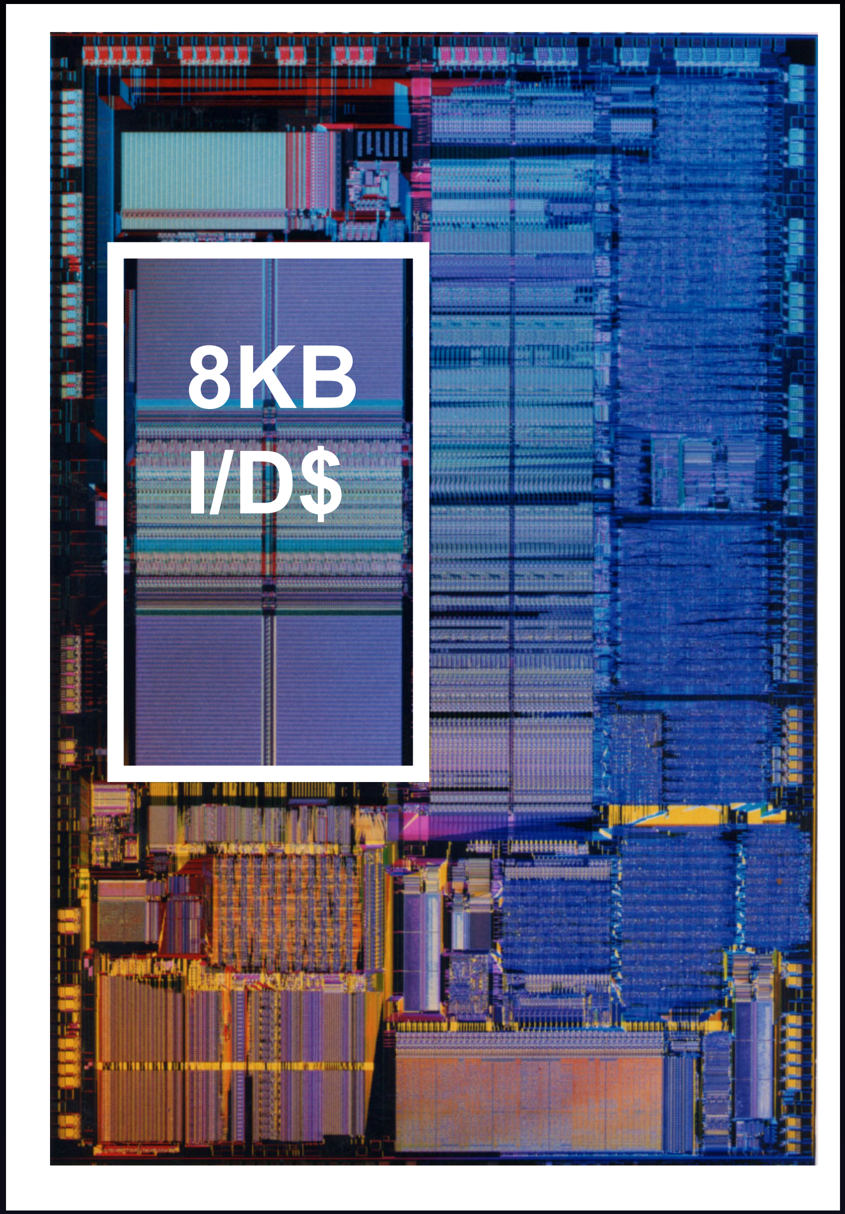


- Level 0 : Registers
- Level 1 : Split Ins. and Data cache
  - typically 8-64KB
  - inside core
- Level 2 and 3 (SRAM)
  - shared by cores
  - 2nd level typically 256-512KB
  - last-level (LLC) typicall 4-16MB
- Level 4 : Main Memory DRAM
  - Desk (4GB), Servers (100s GB)

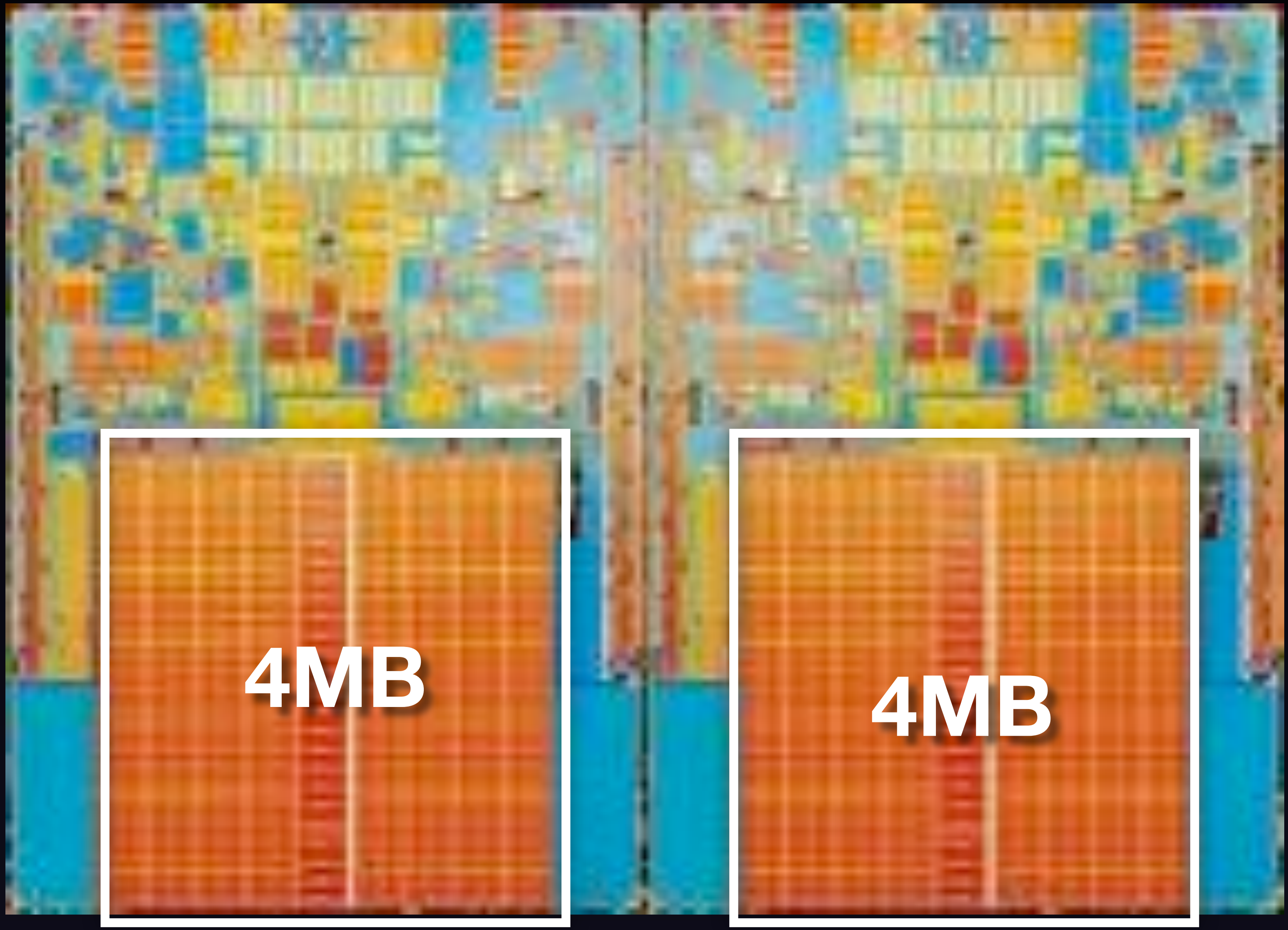
Library Analogy

- Registers = Books on desk
  - actively used, small capacity
- Caches = bookshelves
  - moderate capacity, pretty fast to access
- Main Memory = Library
  - Big; holds almost all data; but slow

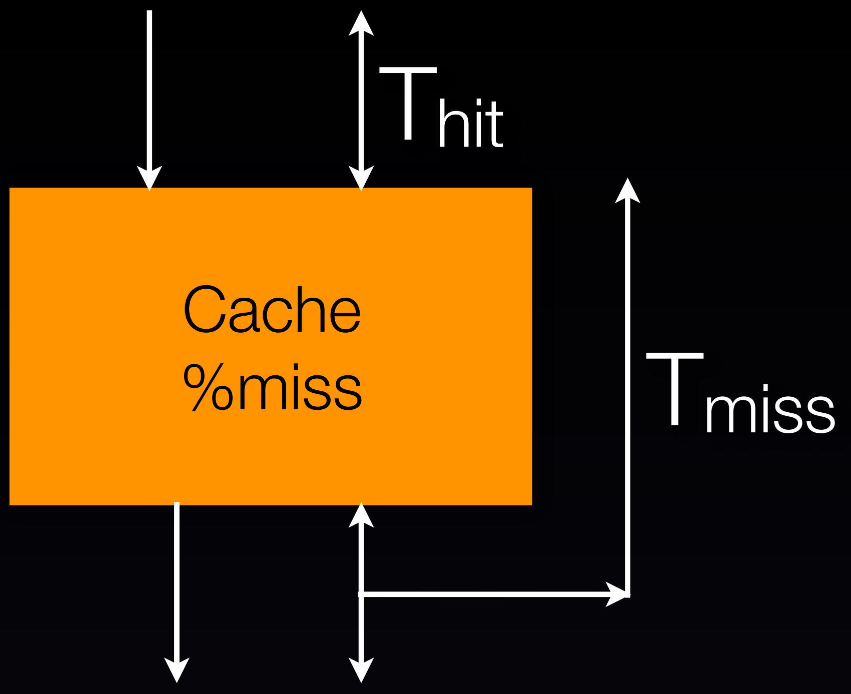
Intel 486



Intel Penryn



Cache Terminology



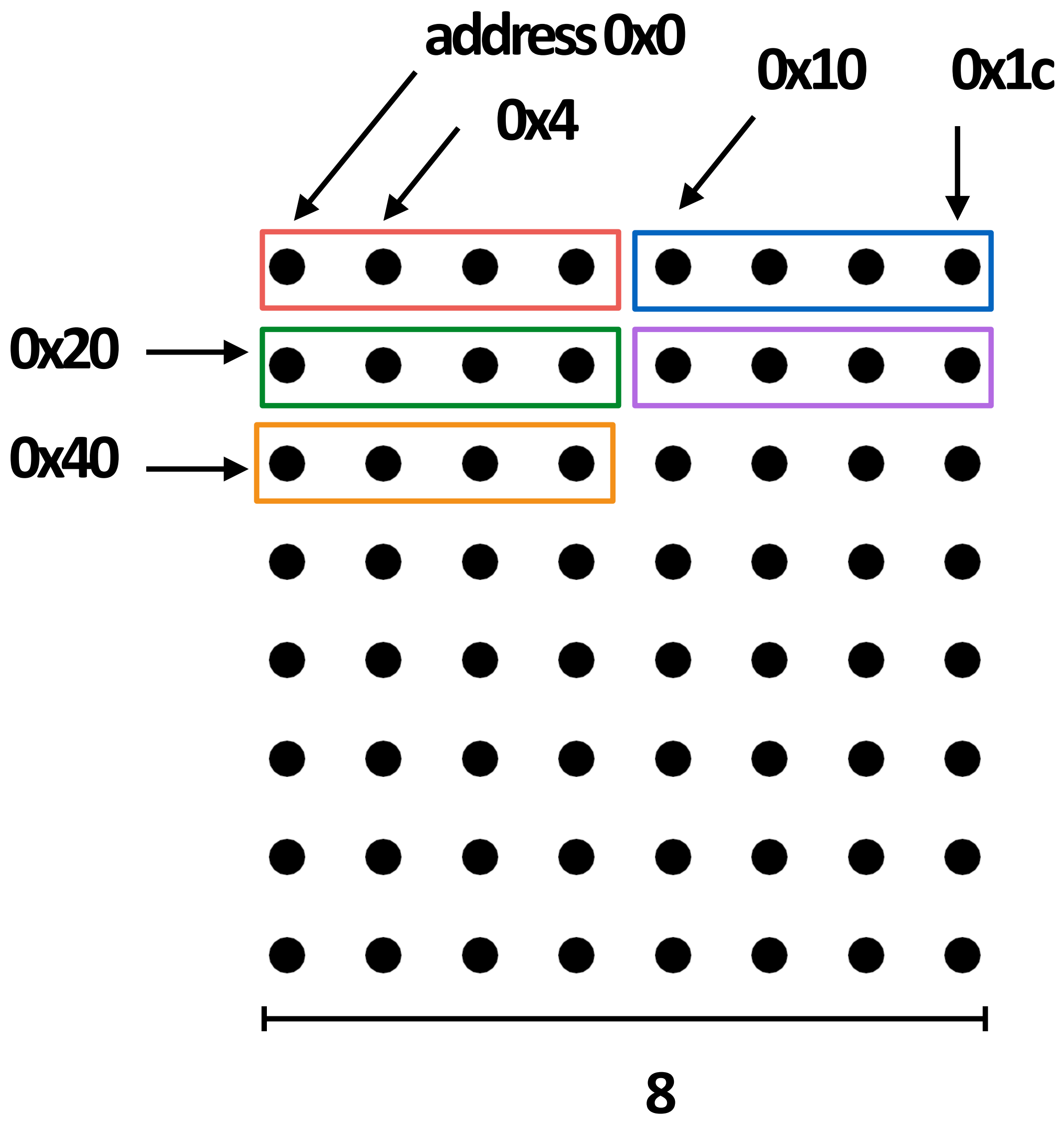
Access : Read/Write to cache

Hit : Desired data in cache
Miss : Desired data not in cache
Fill : Data placed in cache

% miss = #misses/ #accesses
MPKI = # misses/ 1000 inst.
T_{hit} = Time to read (write) data from cache
T_{miss} = Time to read data into cache

$T_{avg} = T_{hit} + \%miss * T_{miss}$

Cache review



Consider 4-byte elements

Consider a cache with 16-byte cache lines

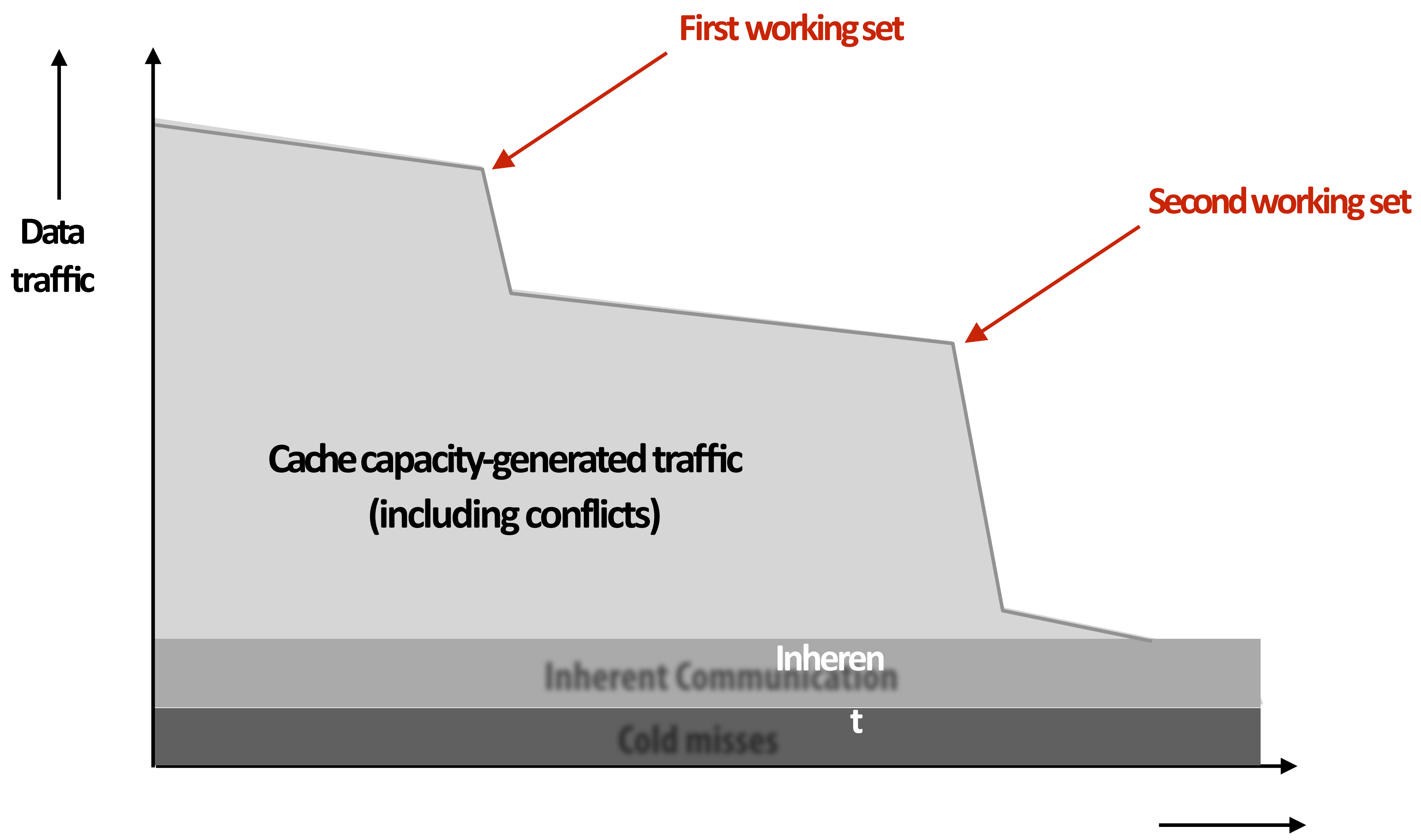
and a capacity of 32 bytes

(2 lines fit in cache)

Least recently used (LRU) replacement

Address accessed	Cache state (after load is complete)		
0x0	0x0 ●●●●		"cold miss"
0x4	0x0 ●●●●		hit
0x8	0x0 ●●●●		hit
0xc	0x0 ●●●●		hit
0x10	0x0 ●●●●	0x10 ●●●●	cold miss
0x14	0x0 ●●●●	0x10 ●●●●	hit
0x18	0x0 ●●●●	0x10 ●●●●	hit
0x1c	0x0 ●●●●	0x10 ●●●●	hit
0x20	0x20 ●●●●	0x10 ●●●●	cold miss (evict 0x0)
0x24	0x20 ●●●●	0x10 ●●●●	hit
0x28	0x20 ●●●●	0x10 ●●●●	hit
0x2c	0x20 ●●●●	0x10 ●●●●	hit
0x30	0x20 ●●●●	0x30 ●●●●	cold miss (evict 0x10)
0x34	0x20 ●●●●	0x30 ●●●●	hit
0x38	0x20 ●●●●	0x30 ●●●●	hit
0x3c	0x20 ●●●●	0x30 ●●●●	hit
0x40	0x40 ●●●●	0x30 ●●●●	cold miss (evict 0x20)

Communication: working set perspective



Stalls

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

- Accessing memory is a major source of stalls

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

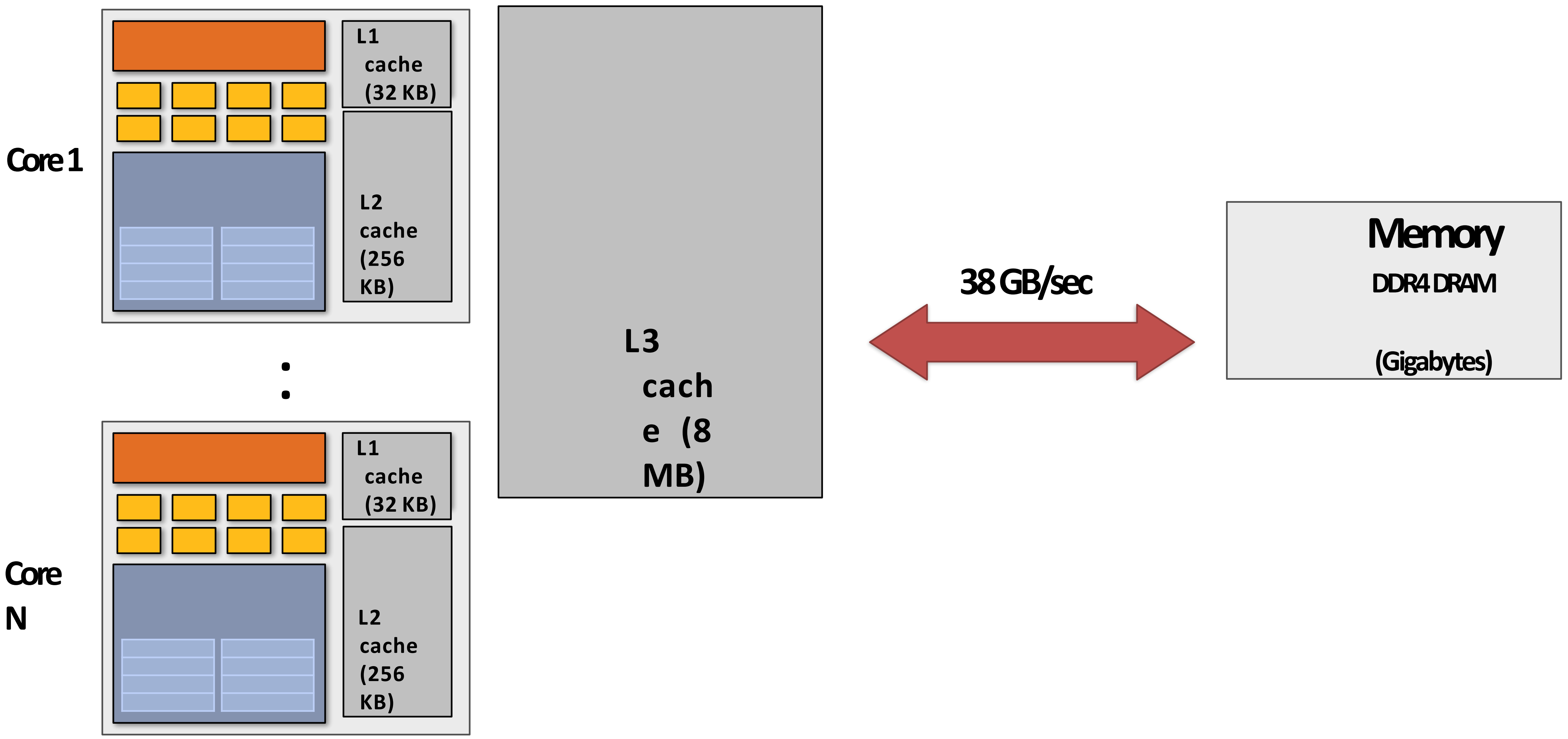
Dependency: cannot execute 'add' instruction until data at mem[r2] and mem[r3] have been loaded from memory

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

Review: why do modern processors have caches?

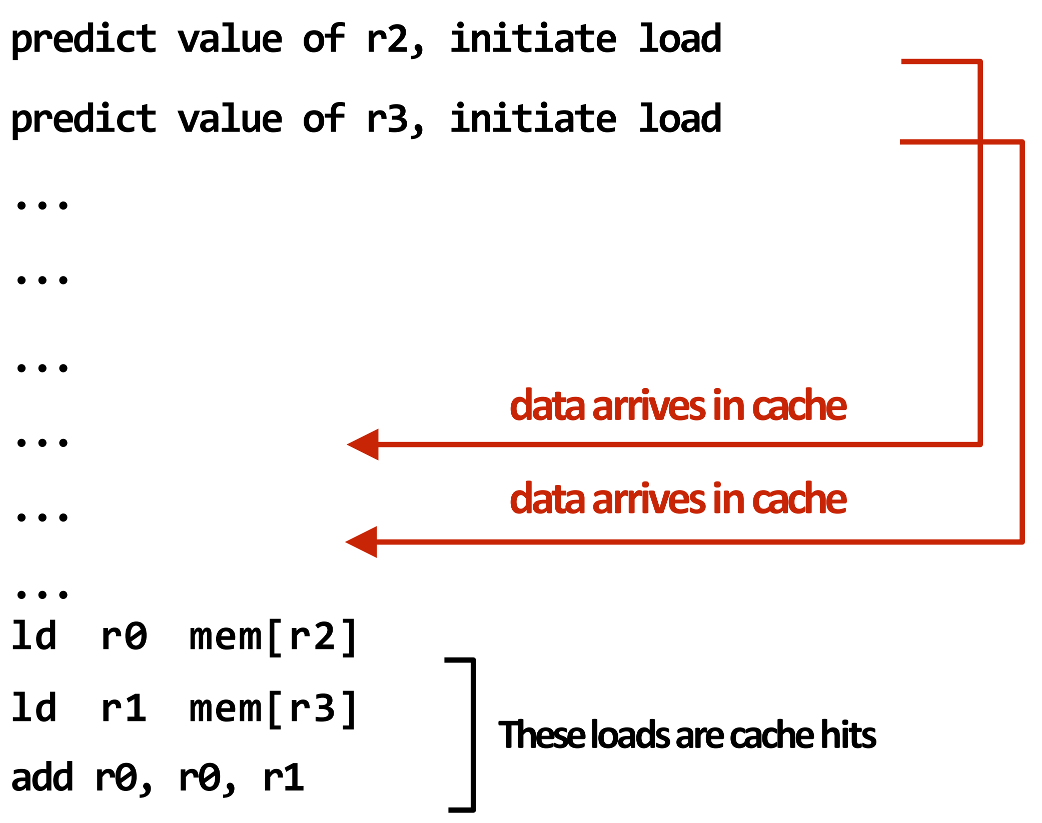
Processors run efficiently when data is resident in caches

Caches reduce memory access latency *



Prefetching reduces stalls (hides latency)

- All modern CPUs have logic for prefetching data into caches
  - Dynamically analyze program's access patterns, predict what it will access soon
- 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
```

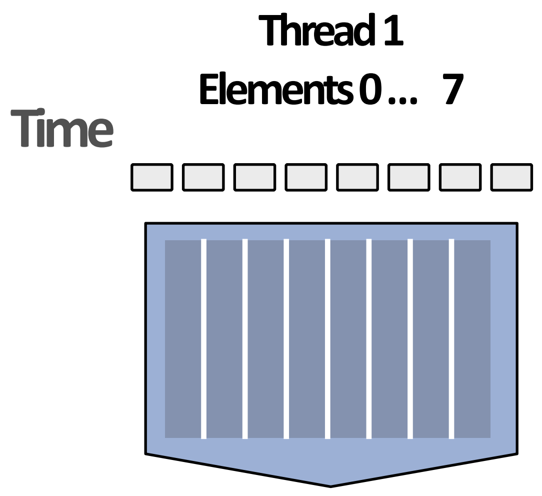
Note: Prefetching can also reduce performance if the guess is wrong (hogs bandwidth, pollutes caches)

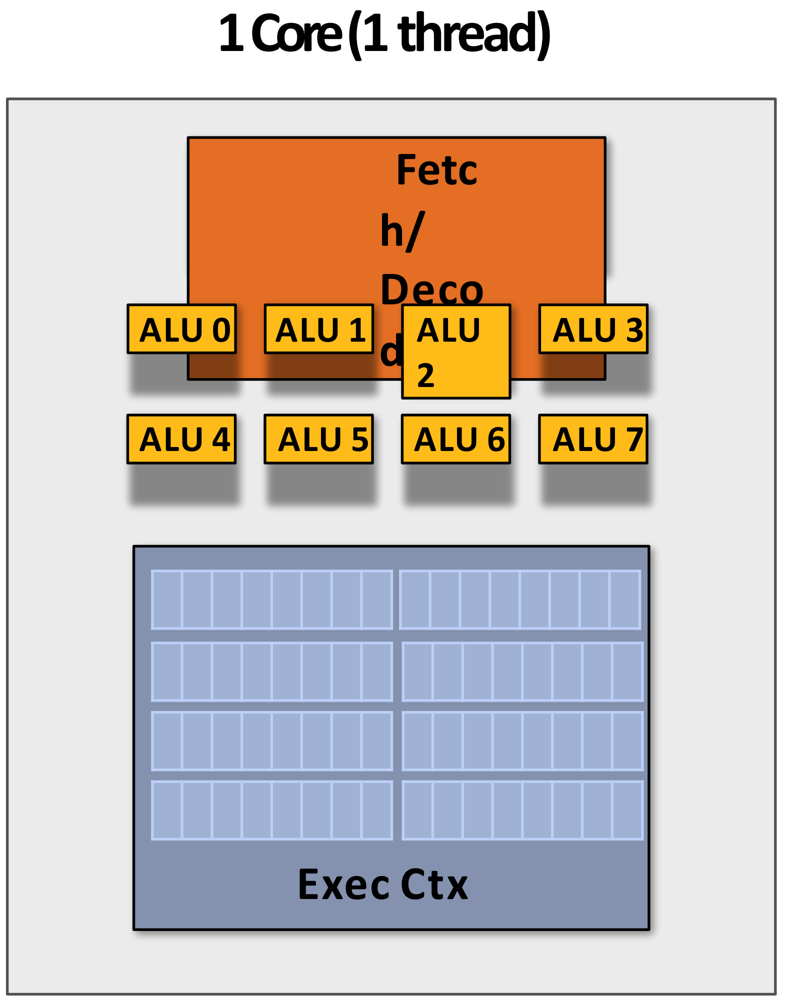
Multi-threading reduces stalls

- Idea: <u>interleave</u> processing of multiple threads on the same core to hide stalls

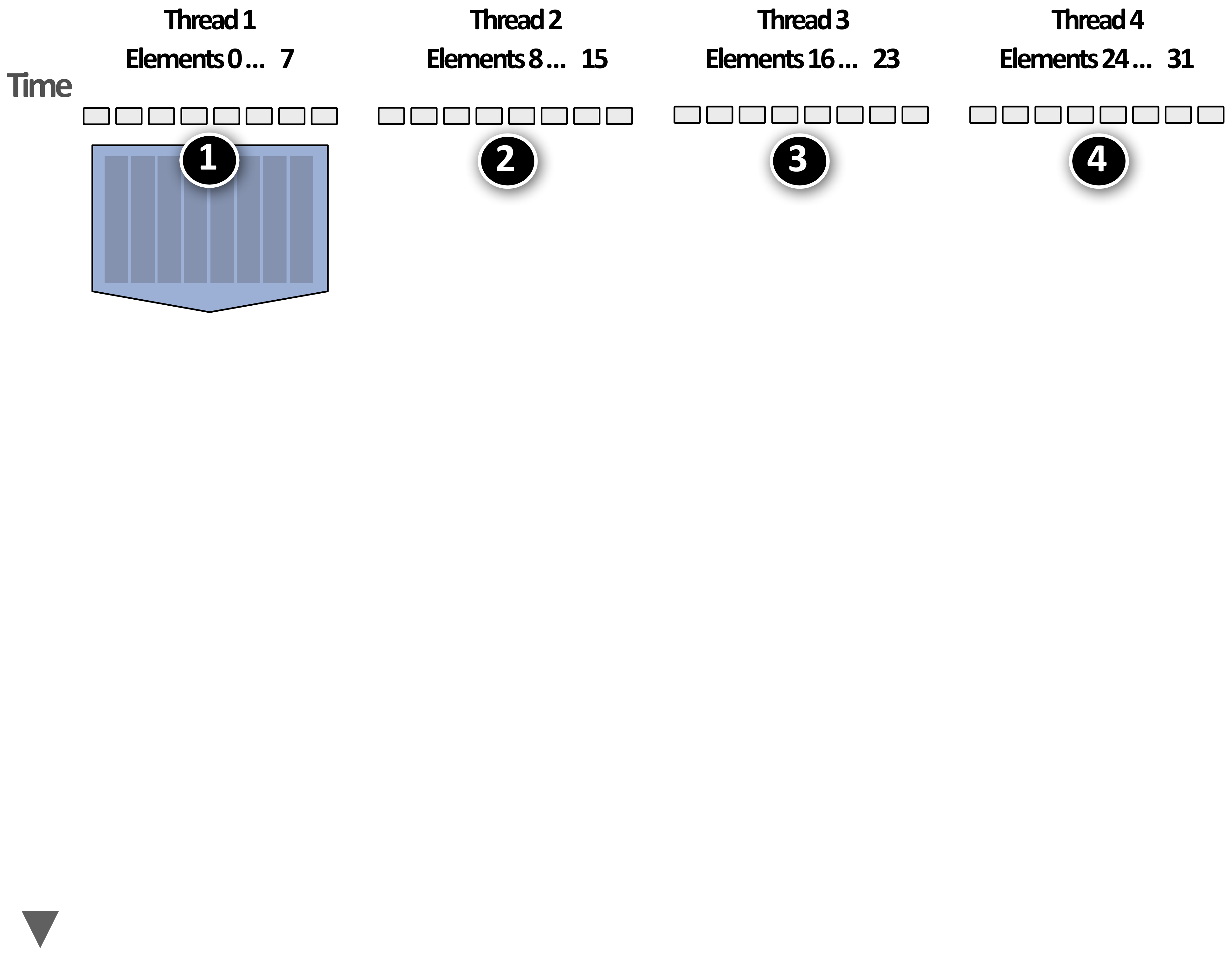
- Like prefetching, multi-threading is a latency <u>hiding</u>, not a latency <u>reducing</u> technique

Hiding stalls with multi-threading

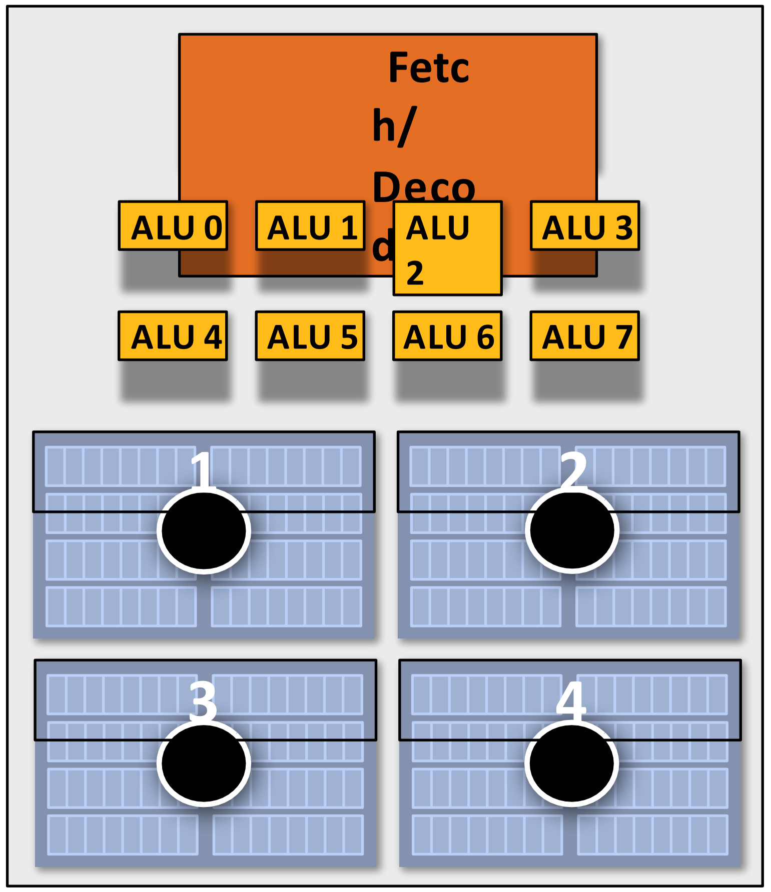




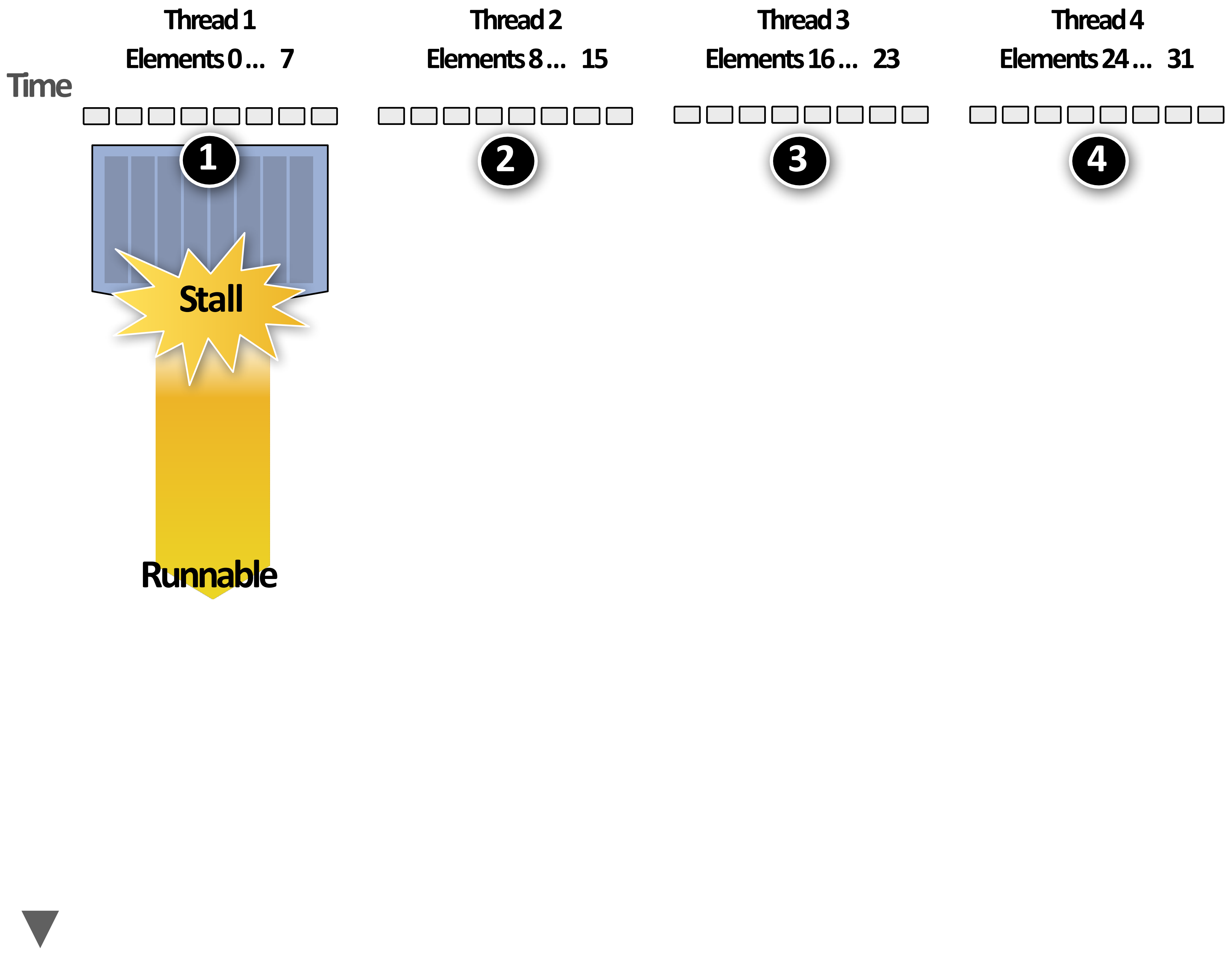
Hiding stalls with multi-threading



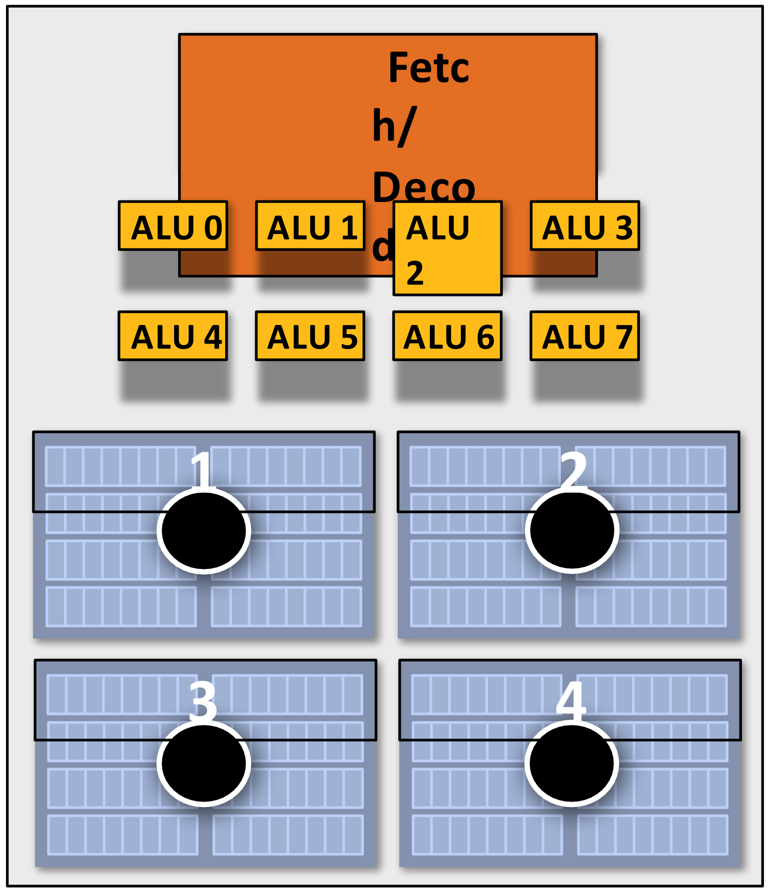
1 Core (4 hardware threads)



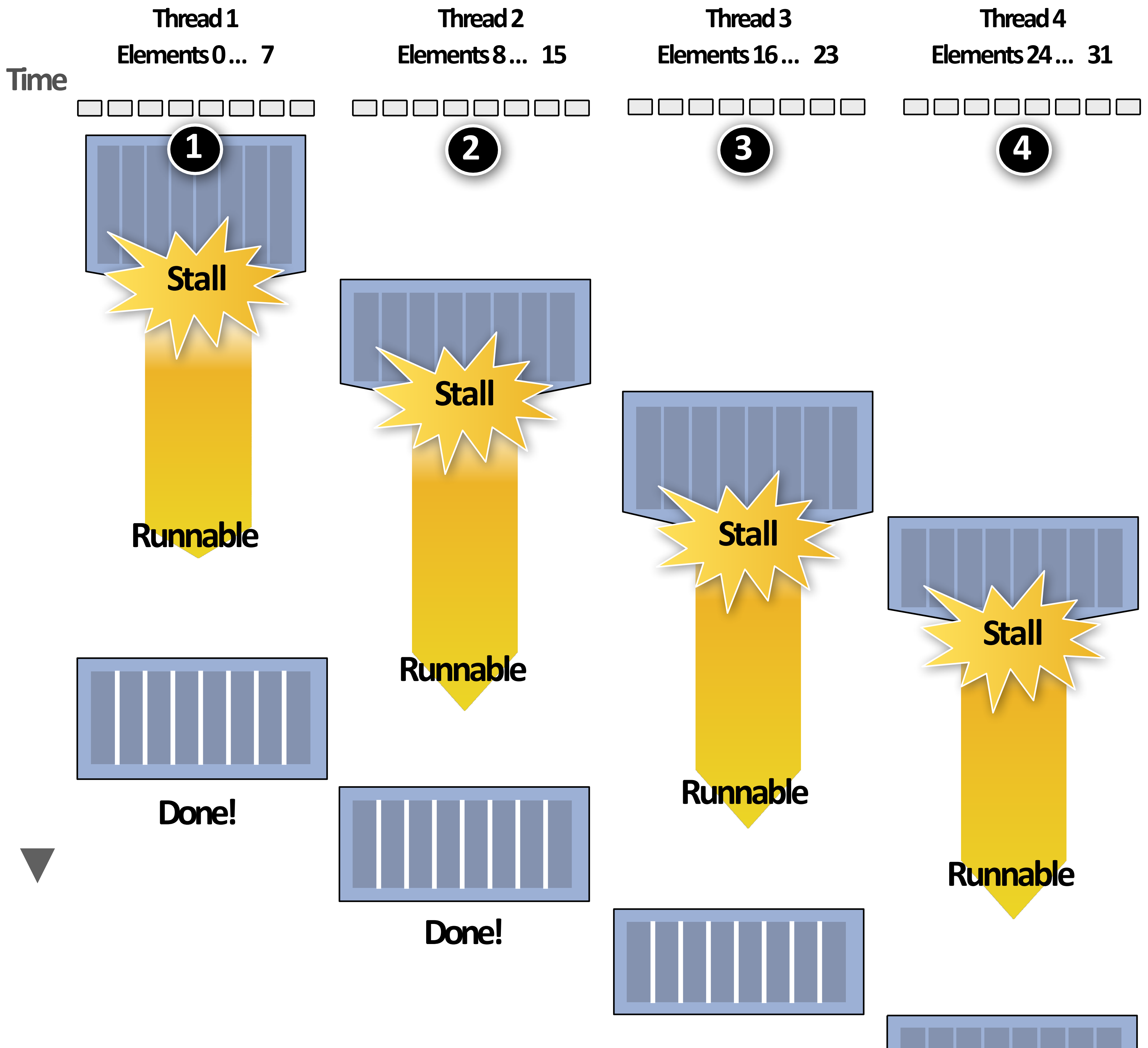
Hiding stalls with multi-threading

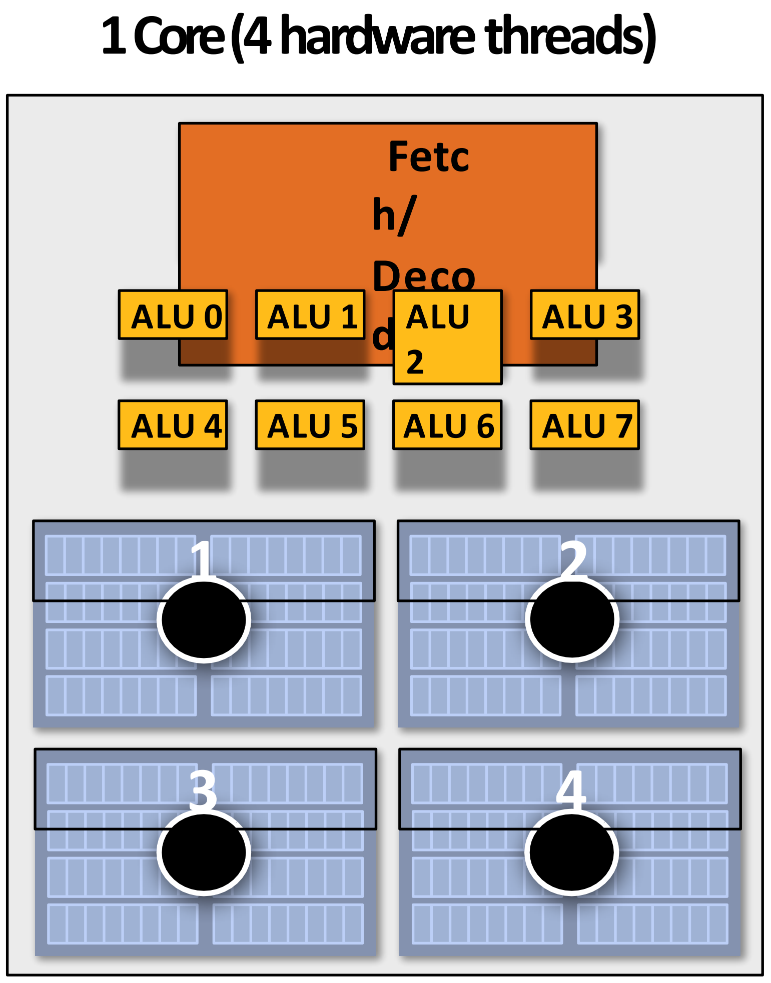


1 Core (4 hardware threads)

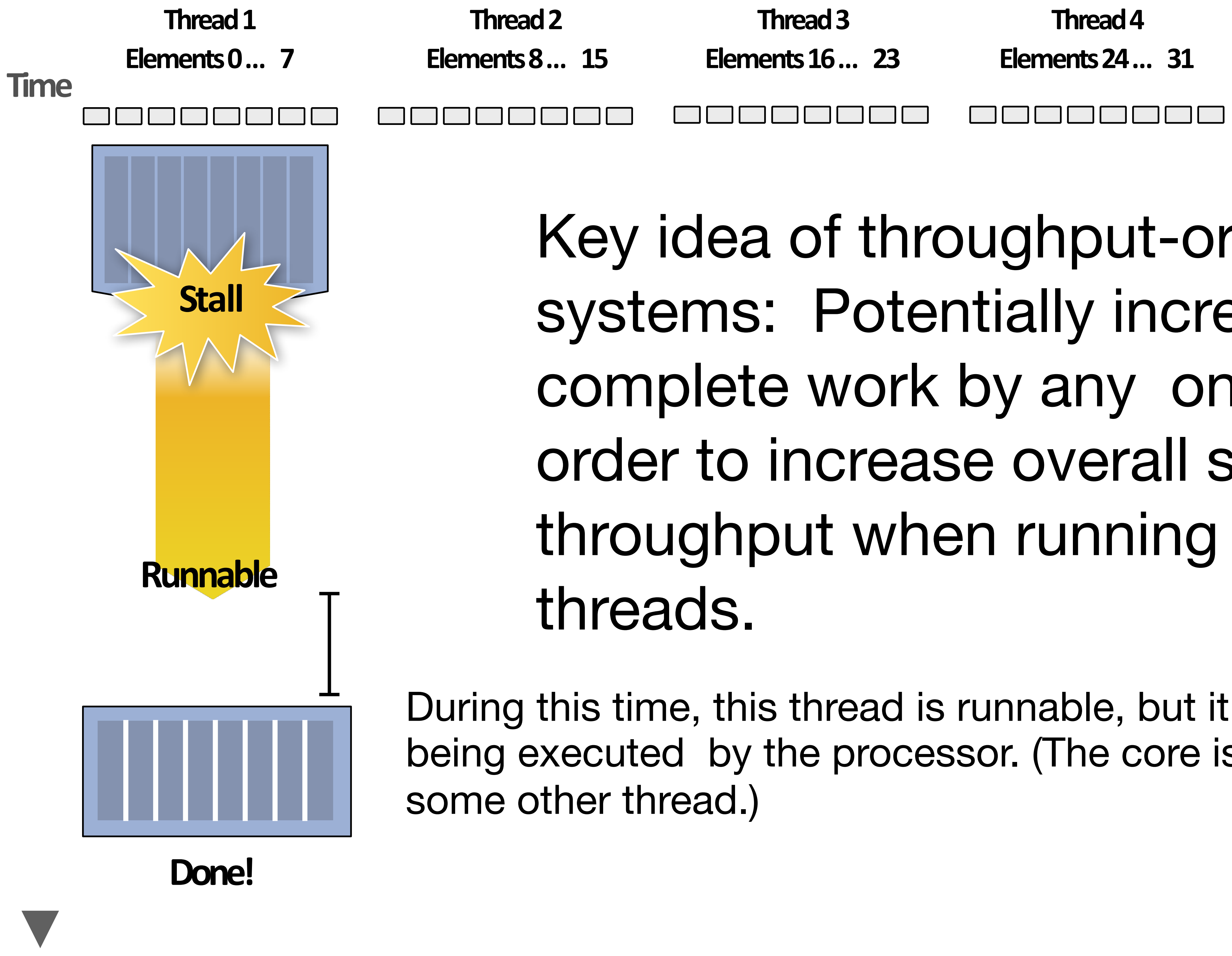


Hiding stalls with multi-threading





Throughput computing trade-off

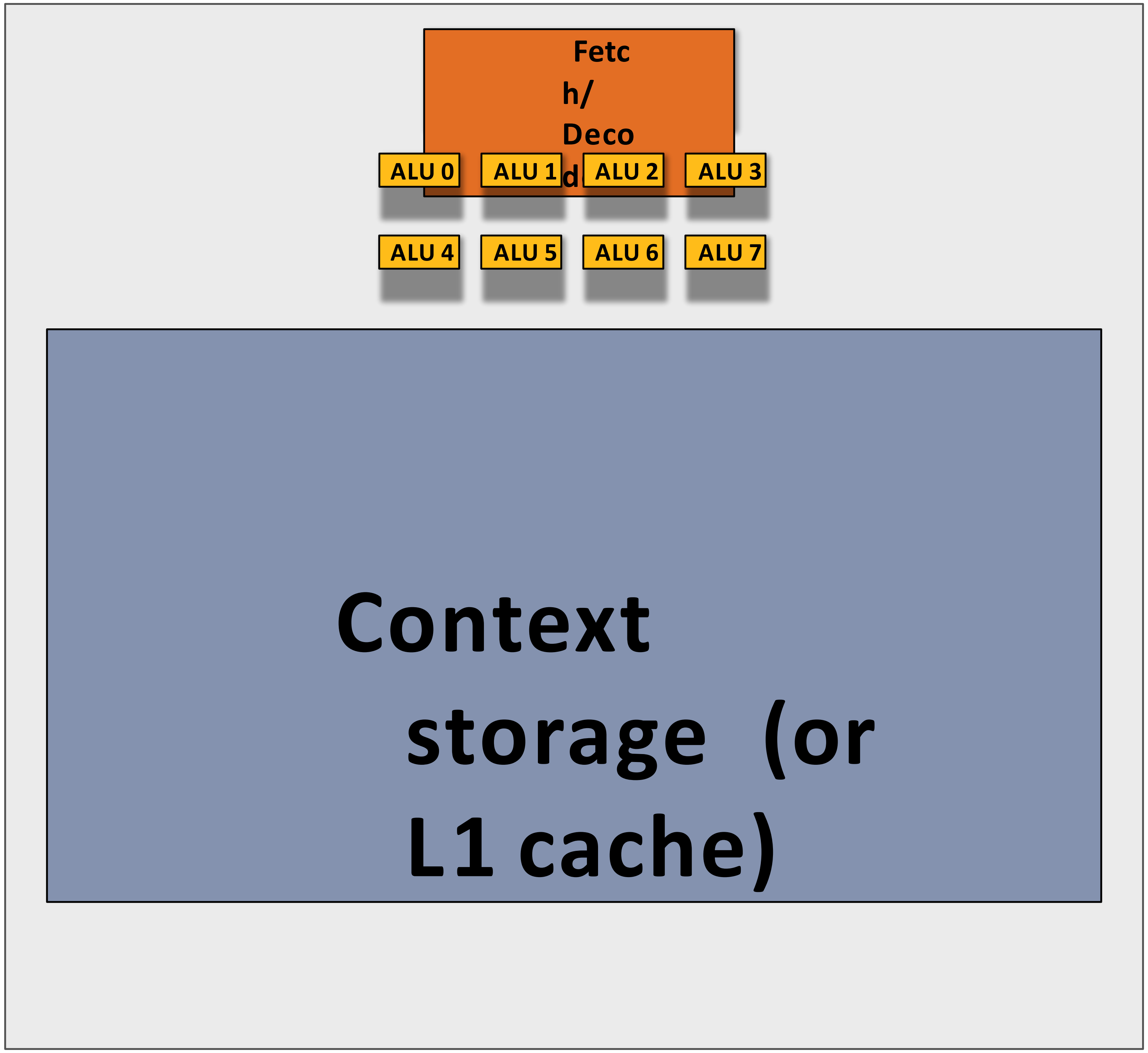


Key idea of throughput-oriented systems: Potentially increase time to complete work by any one thread, in order to increase overall system throughput when running multiple threads.

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

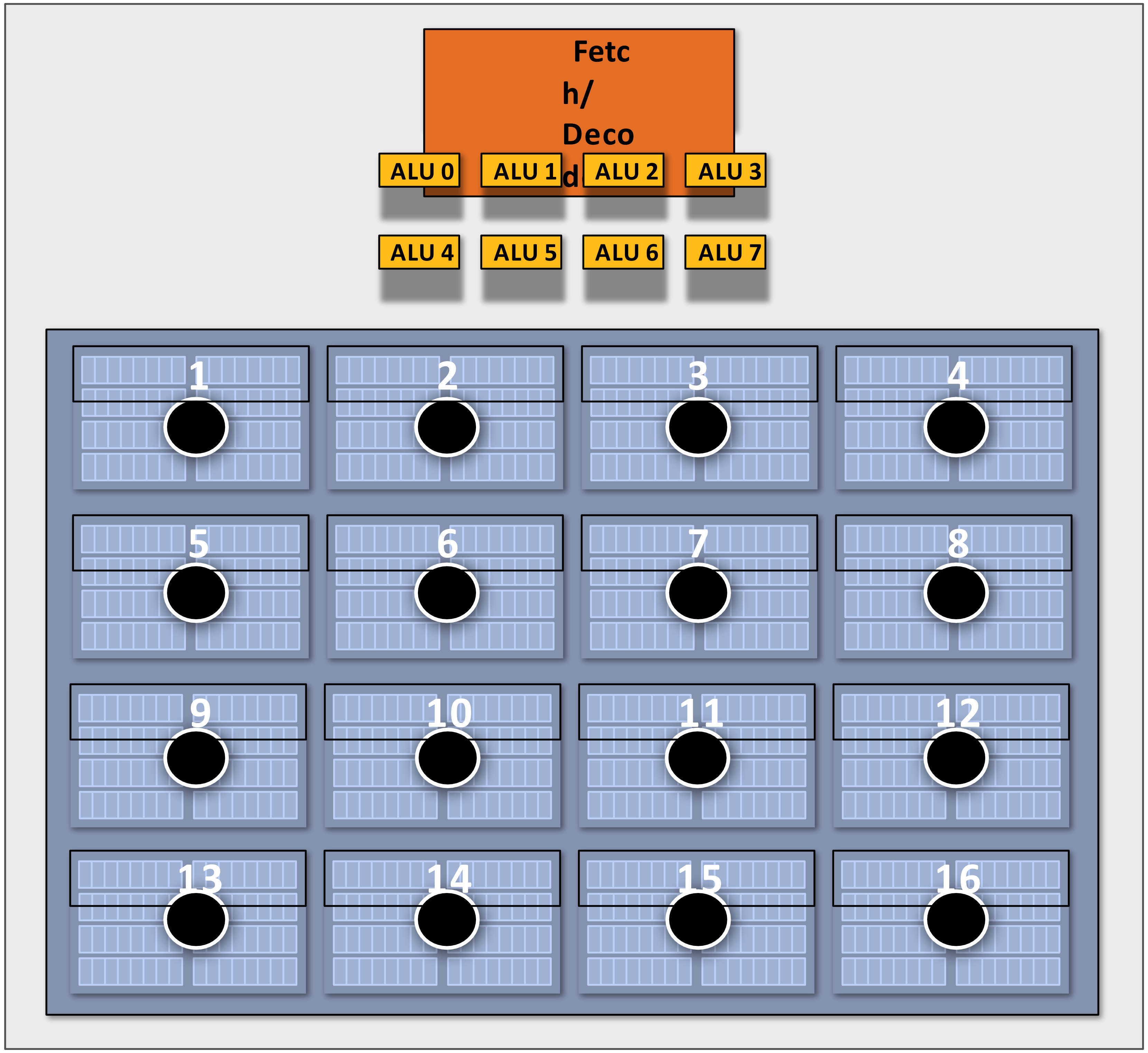
Storing execution contexts

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



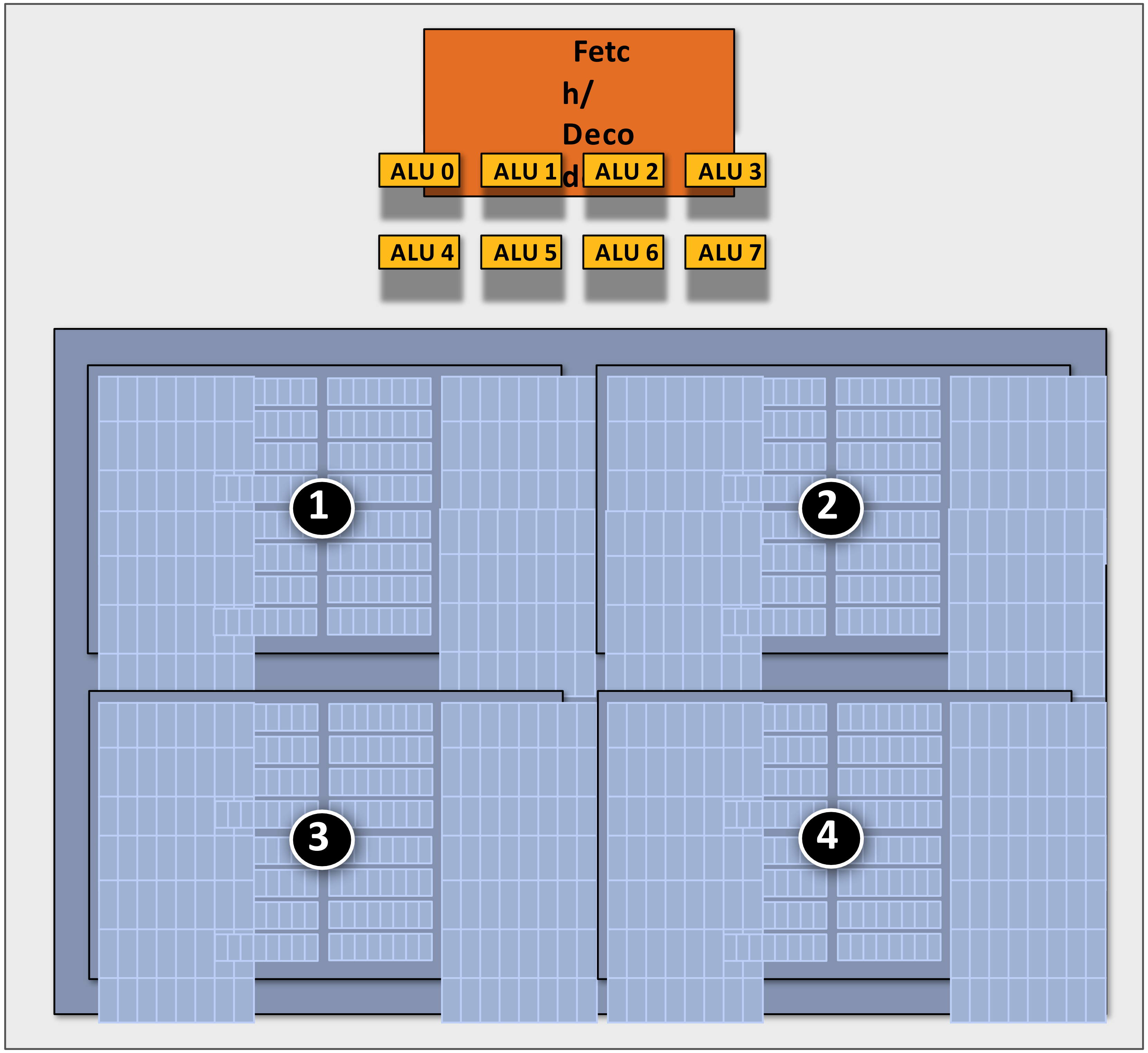
Many small contexts (high latency hiding ability)

1 core
(16 hardware threads, storage for small working set per thread)



Four large contexts (low latency hiding ability)

1 core
(4 hardware threads, storage for larger working set per thread)



Hardware-supported multi-threading

- Core manages execution contexts for multiple threads
  - Runs instructions from runnable threads (processor makes decision about which thread to run each clock, not the operating system)
  - 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

- 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 ALUs

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

Multi-threading summary

- Benefit: use a core's execution resources (ALUs) more efficiently
  - Hide memory latency
  - Fill multiple functional units of superscalar architecture (when one thread has insufficient ILP)

- Costs
  - Requires additional storage for thread contexts
  - Increases run time of any single thread
    (often not a problem, we usually care about throughput in parallel apps)
  - Requires additional independent work in a program (more independent work than ALUs!)
  - Relies heavily on memory bandwidth
    - More threads → larger working set → less cache space per thread
    - May go to memory more often, but can hide the latency

Kayvon's fictitious multi-core chip

16 cores

8 SIMD ALUs per core

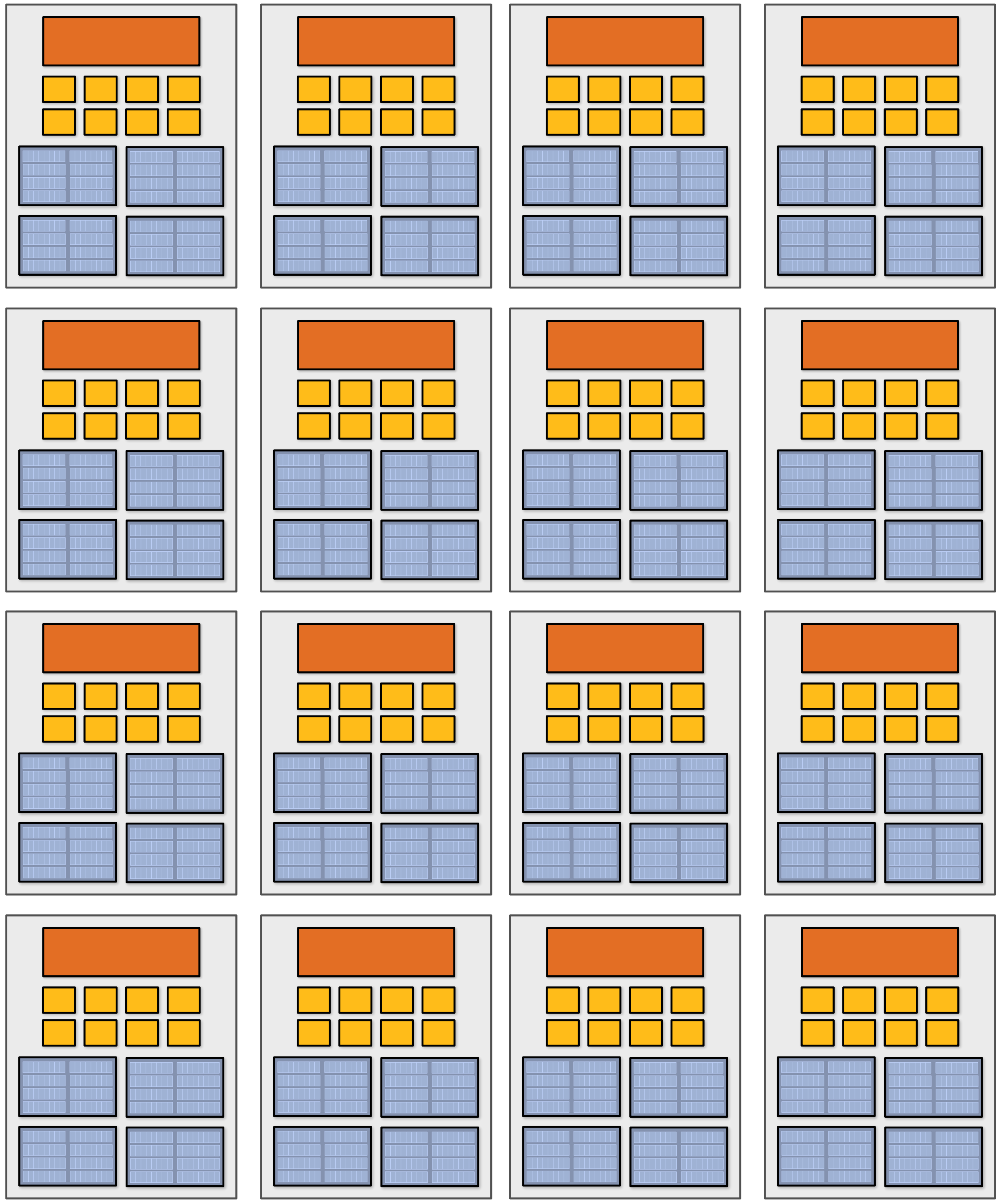
(128 total)

4 threads per core

16 simultaneous instruction streams

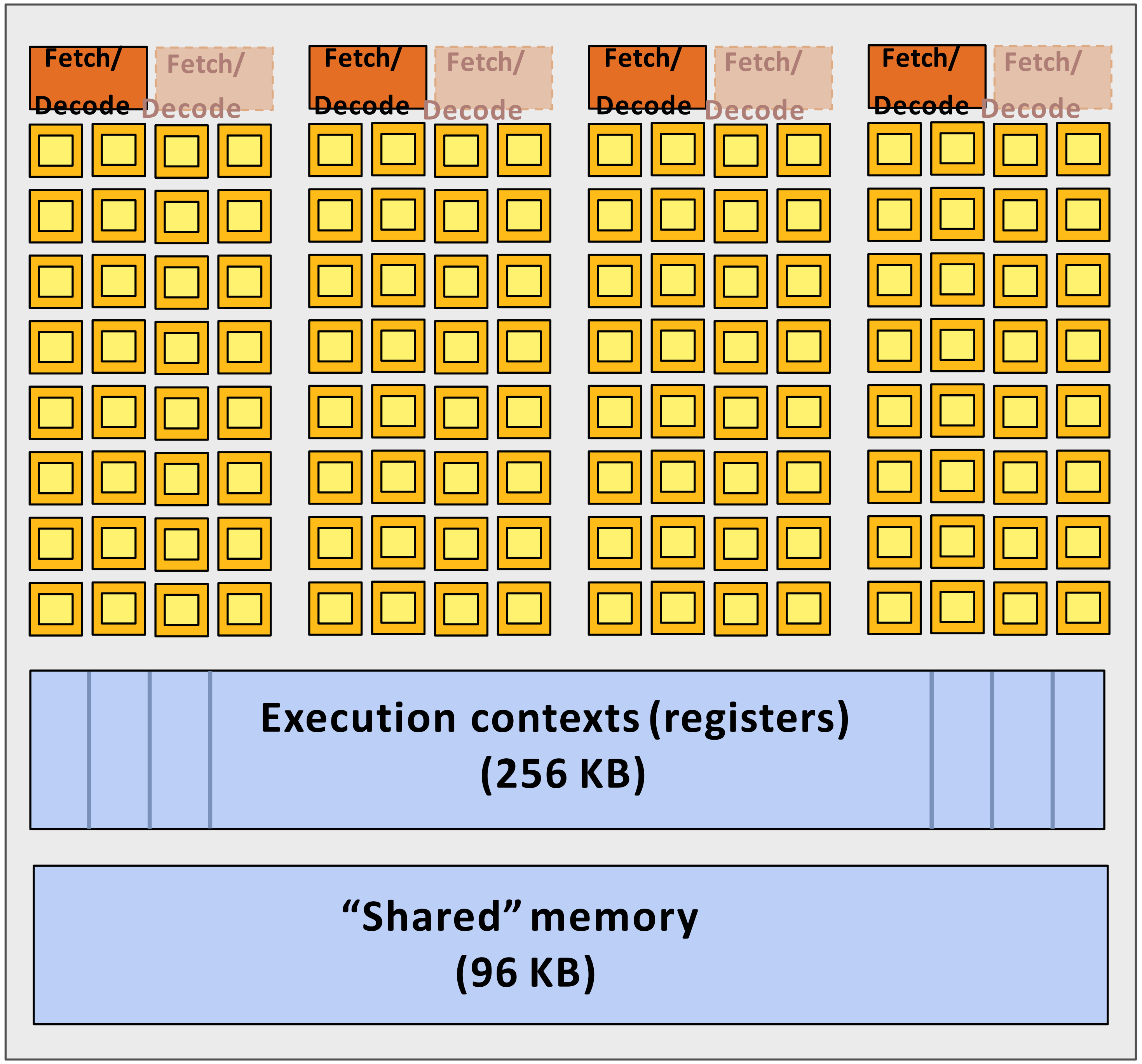
64 total concurrent instruction streams

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



GPUs: extreme throughput-oriented processors

NVIDIA GTX 1080 core ("SM")

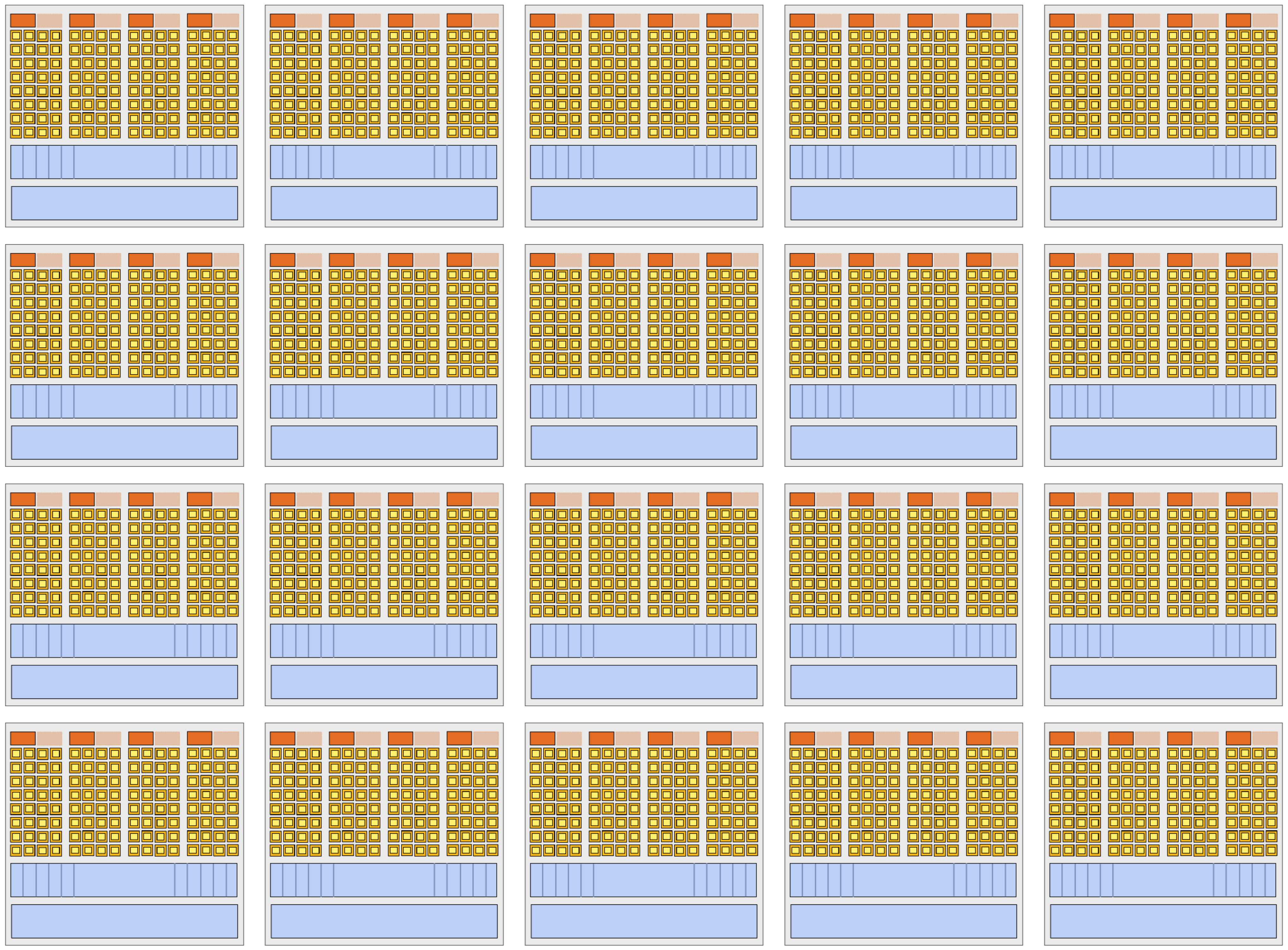


🟨 = SIMD function unit, control shared across 32 units (1 MUL-ADD per clock)

- Instructions operate on 32 pieces of data at a time (instruction streams called "warps").
- Think: warp = thread issuing 32-wide vector instructions
- Different instructions from up to four warps can be executed simultaneously (simultaneous multi-threading)
- Up to 64 warps are interleaved on the SM (interleaved multi-threading)
- Over 2,048 elements can be processed concurrently by a core

Source: NVIDIA Pascal Tuning Guide

NVIDIA GTX 1080



There are 20 SM cores on the GTX 1080:

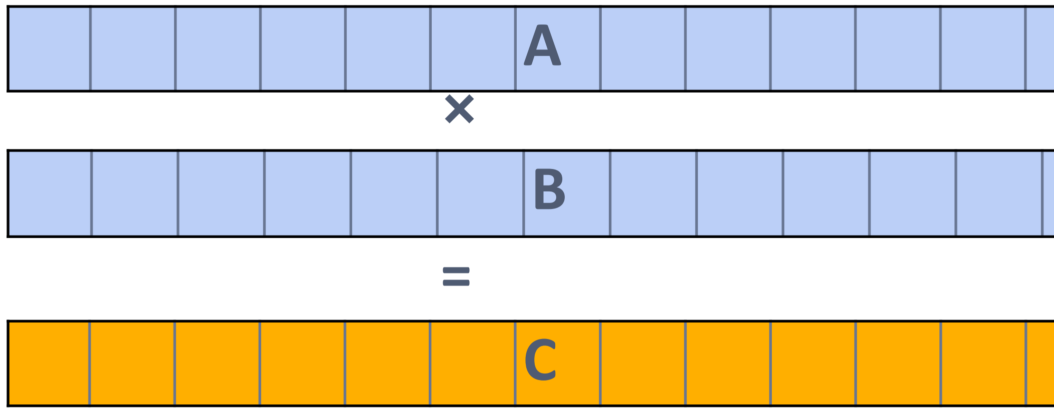
That’s 40,960 pieces of data being processed concurrently to get maximal latency hiding!

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 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!

(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 (traditional intra-thread temporal locality optimizations)
  - Share data across threads (inter-thread cooperation)

- 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

Summary

- Three major ideas that all modern processors employ to varying degrees
  - Provide multiple processing cores
    - Simpler cores (embrace thread-level parallelism over instruction-level parallelism)
  - Amortize instruction stream processing over many ALUs (SIMD)
    - Increase compute capability with little extra cost
  - Use multi-threading to make more efficient use of processing resources (hide latencies, fill all available resources)

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

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

Terms

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