Lecture 1:

Course Introduction + Review of Throughput HW Architecture

Visual Computing Systems
Stanford CS348K, Spring 2022

Hello from the course staff

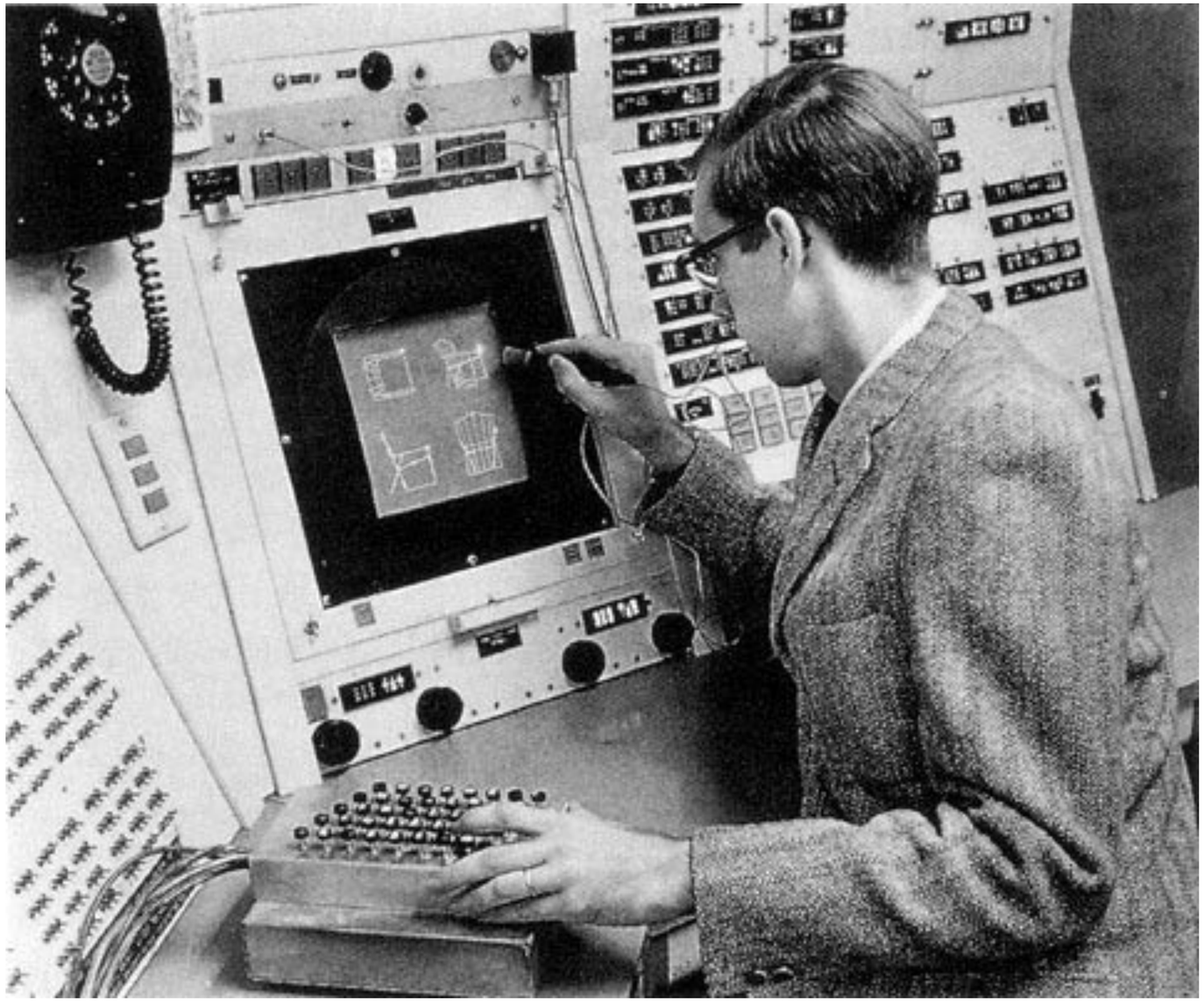
Your instructor (me)

Prof. Kayvon

Your CA

Brennan Shacklett

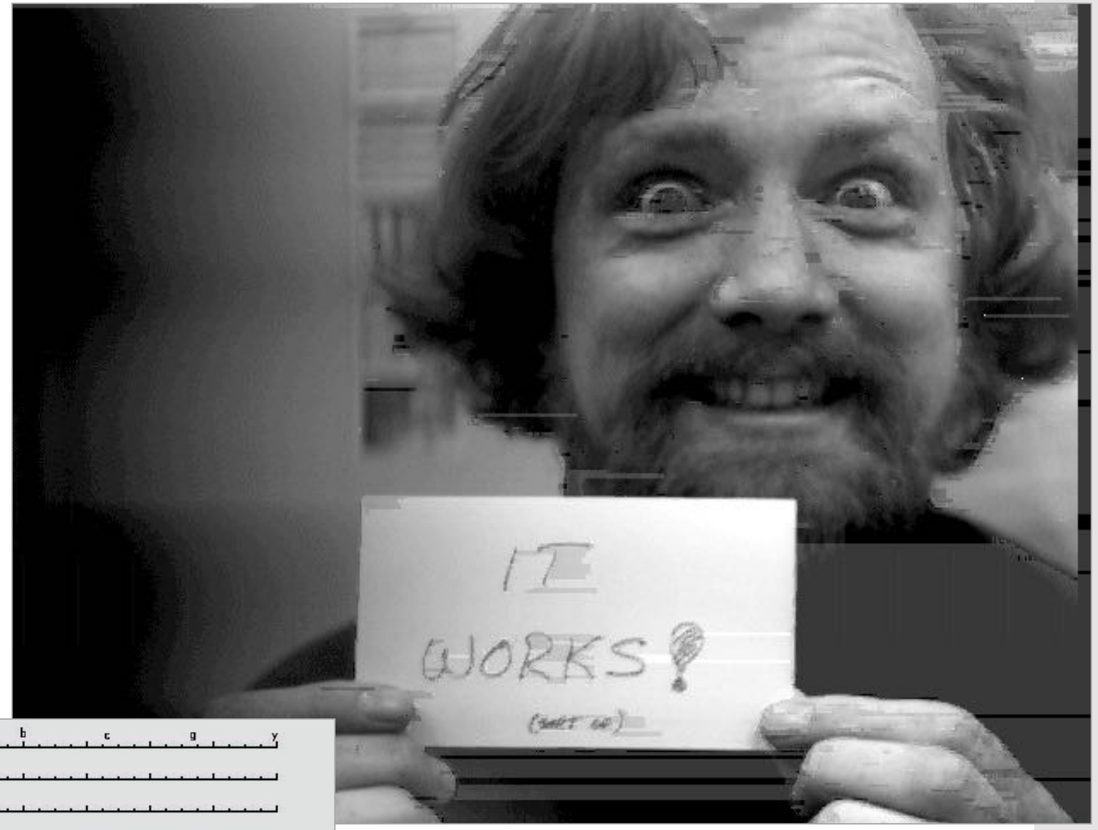
Visual computing applications
have always demanded some of the world's most advanced parallel computing systems

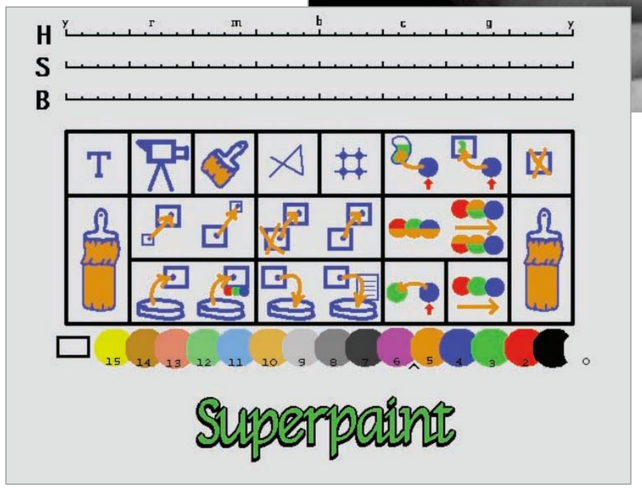


Ivan Sutherland's Sketchpad on MIT TX-2 (1962)

The frame buffer

Shoup's SuperPaint (PARC 1972-73)

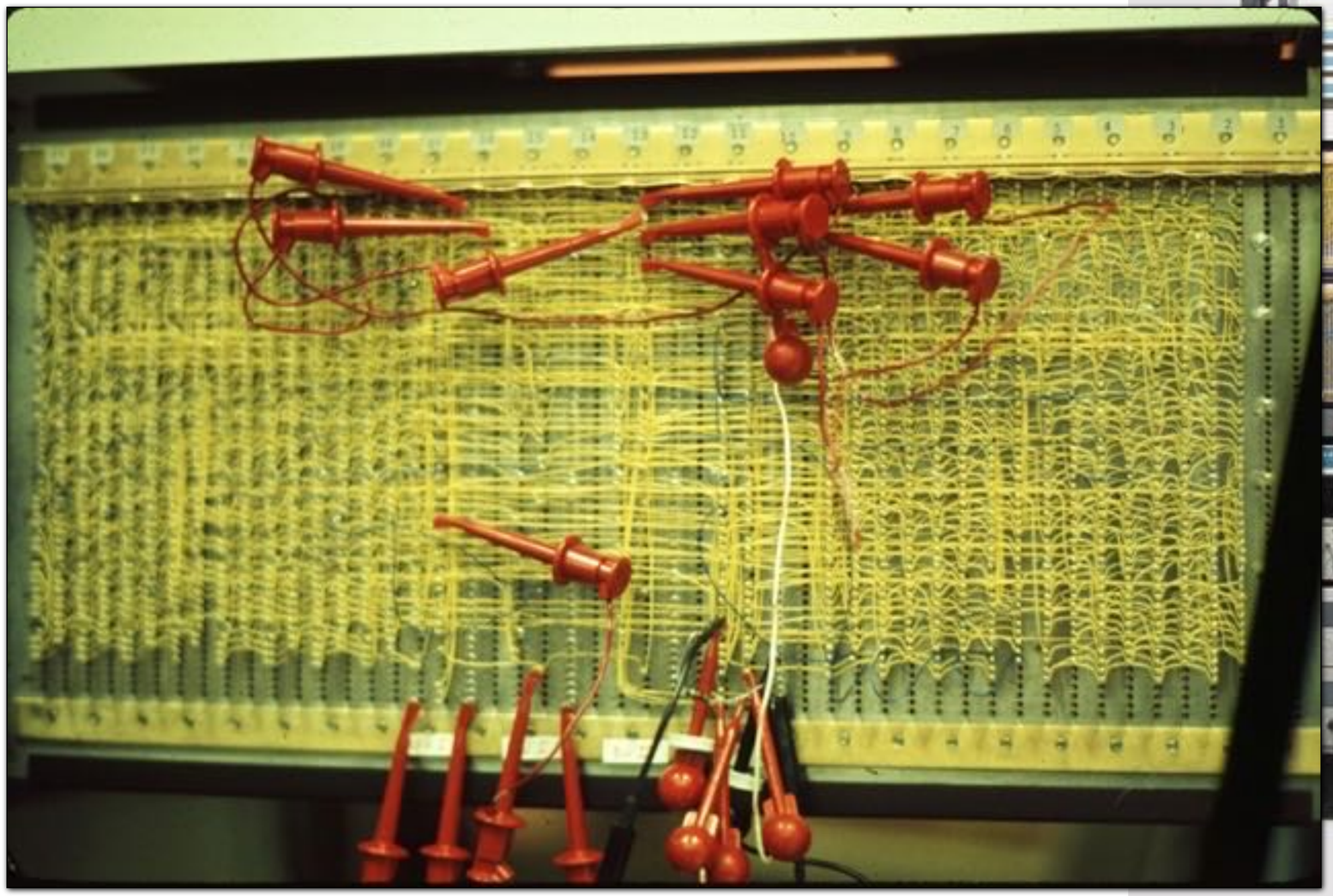




16 2K shift registers (640 x 486 x 8 bits)

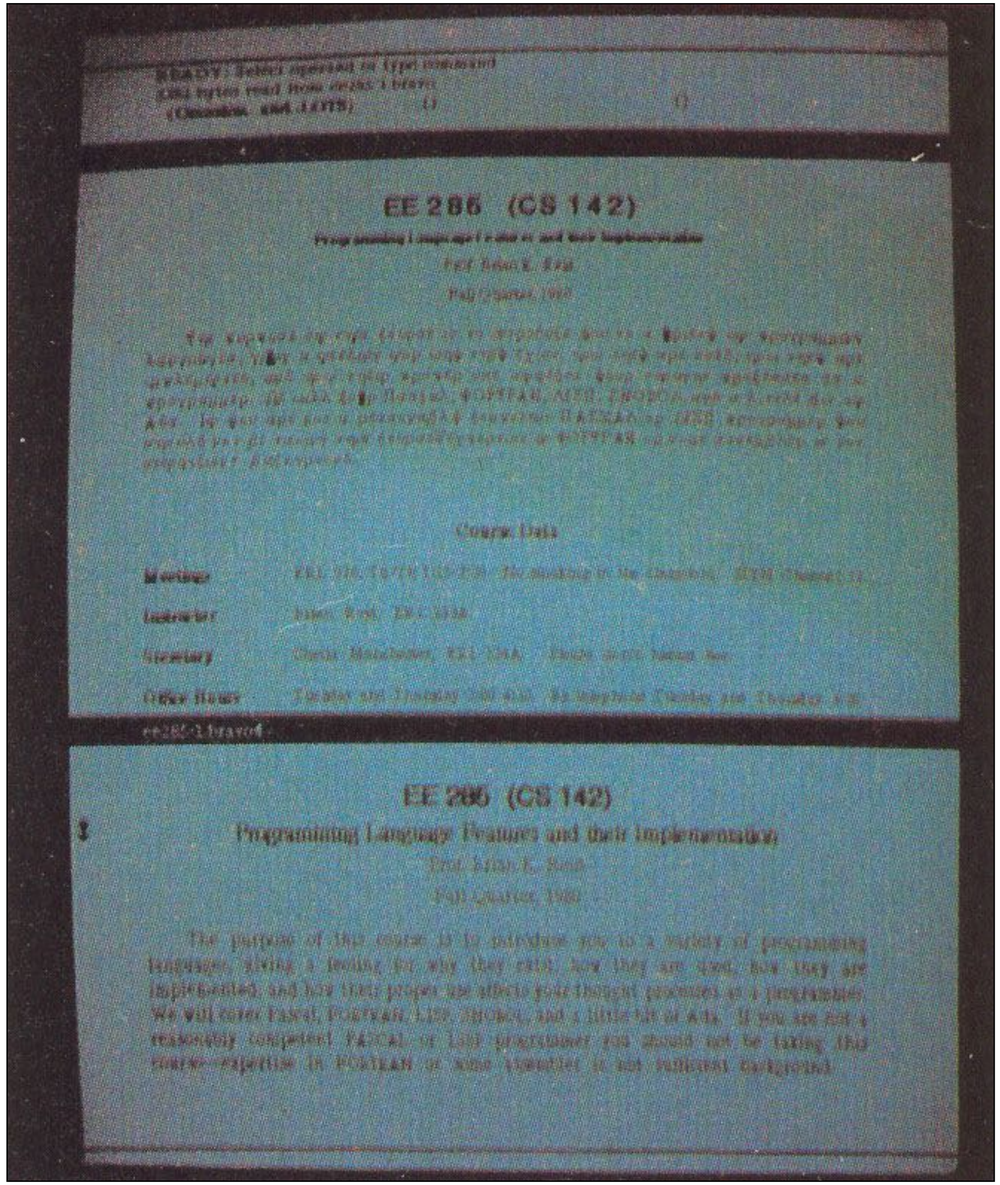
The frame buffer

Shoup's SuperPaint (PARC 1972-73)

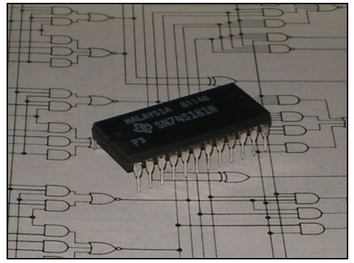


16 2K shift registers (640 x 486 x 8 bits)

Xerox Alto (1973)



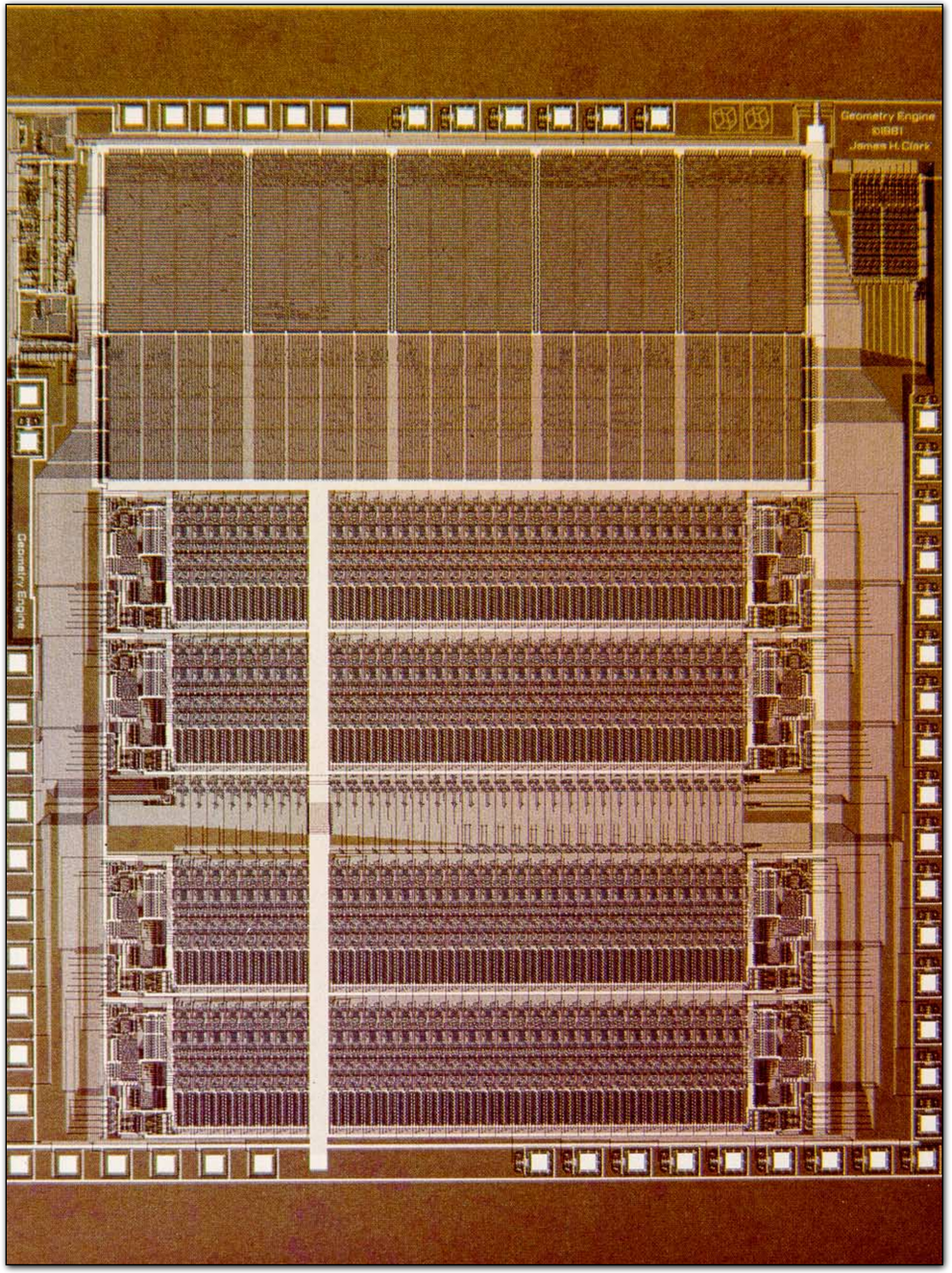
Bravo (WYSIWYG)



TI 74181 ALU

Clark's geometry engine (1982)

ASIC for geometric transforms used in real-time graphics



NVIDIA Titan RTX 3090 GPU

~ 40 TFLOPs fp32 *

4X flops of ASCI Q (top US supercomputer circa 2002) **

* doesn't about 70 TFLOPS of ray tracing compute + 320 TFLOPS of DNN compute

** not apples to Apples since ASCI Q is double precision flops

Cyberpunk 2077

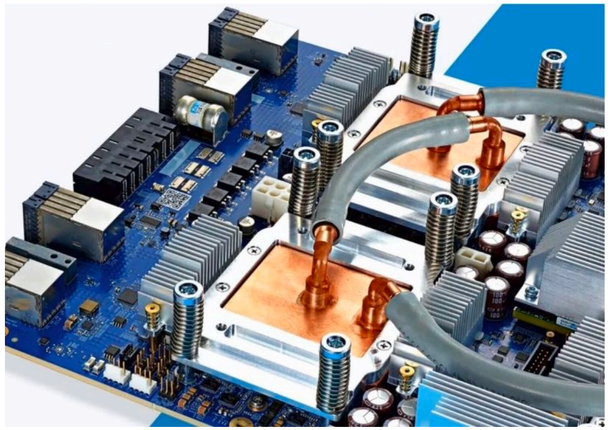
Forza Motorsport 7

Unreal 5 Demo (Nanite renderer)

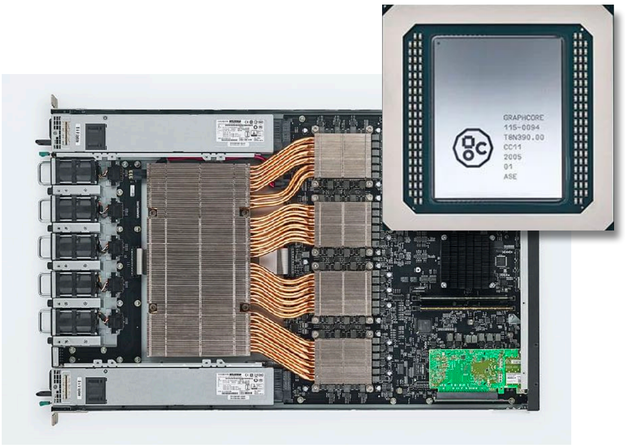
Image/video analysis via deep learning

https://medium.com/analytics-vidhya/introduction-to-computer-vision-with-opencv-part-1-3dc948521deb

Hardware acceleration of DNN inference/training



Google TPU3

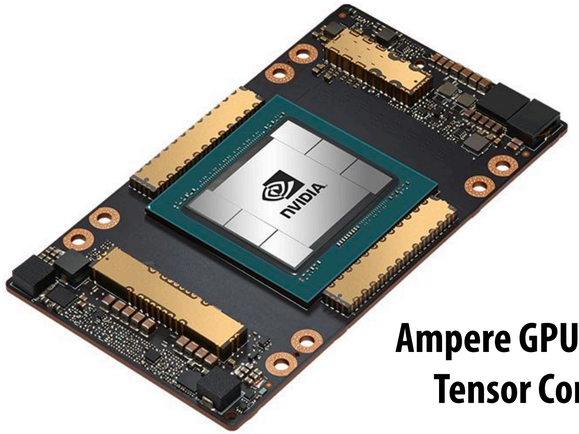


GraphCore IPU

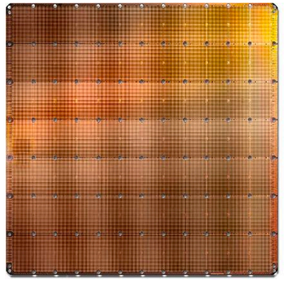
Apple Neural Engine

Intel Deep Learning
Inference Accelerator

SambaNova
Cardinal SN10

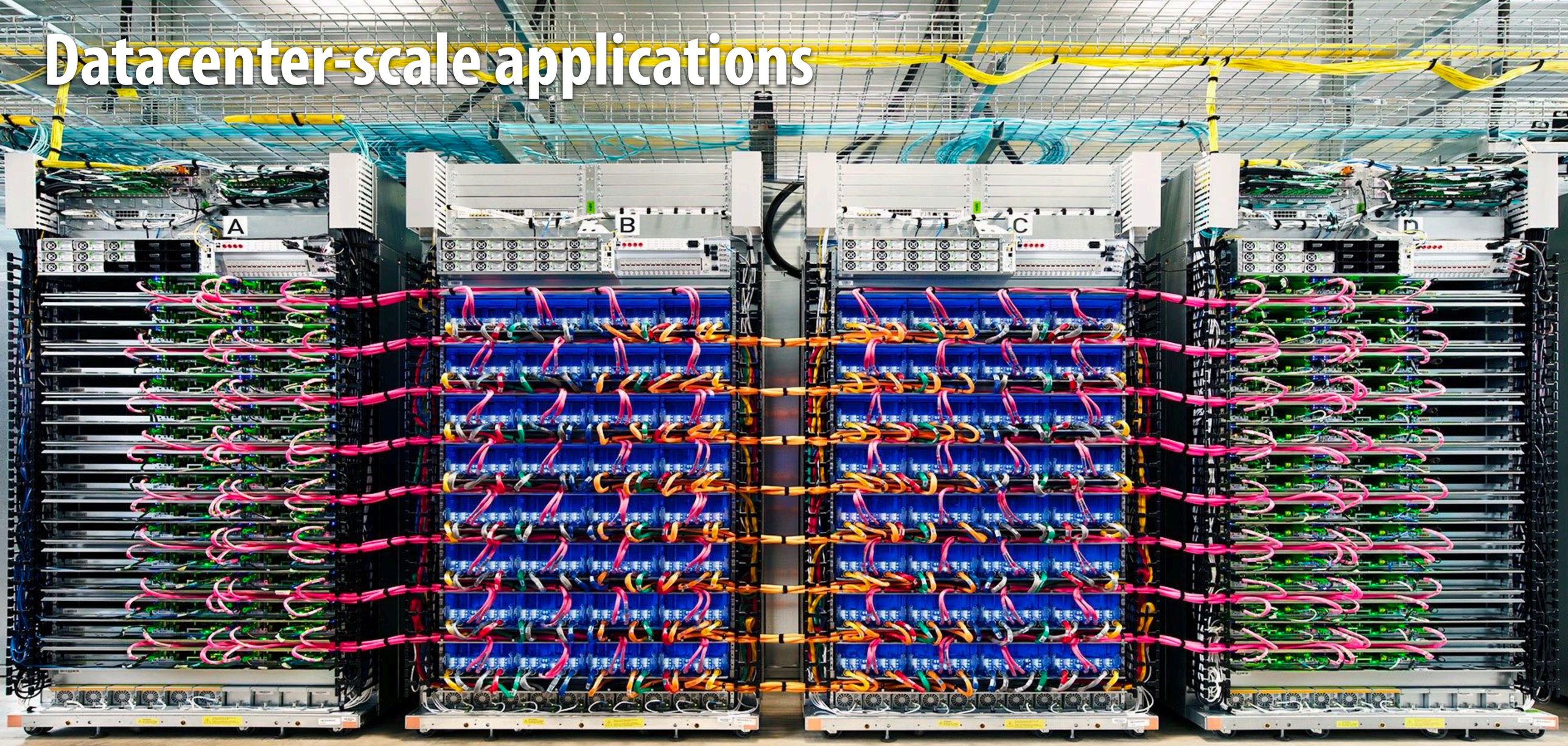


Ampere GPU with
Tensor Cores



Cerebras Wafer Scale Engine

Datacenter-scale applications

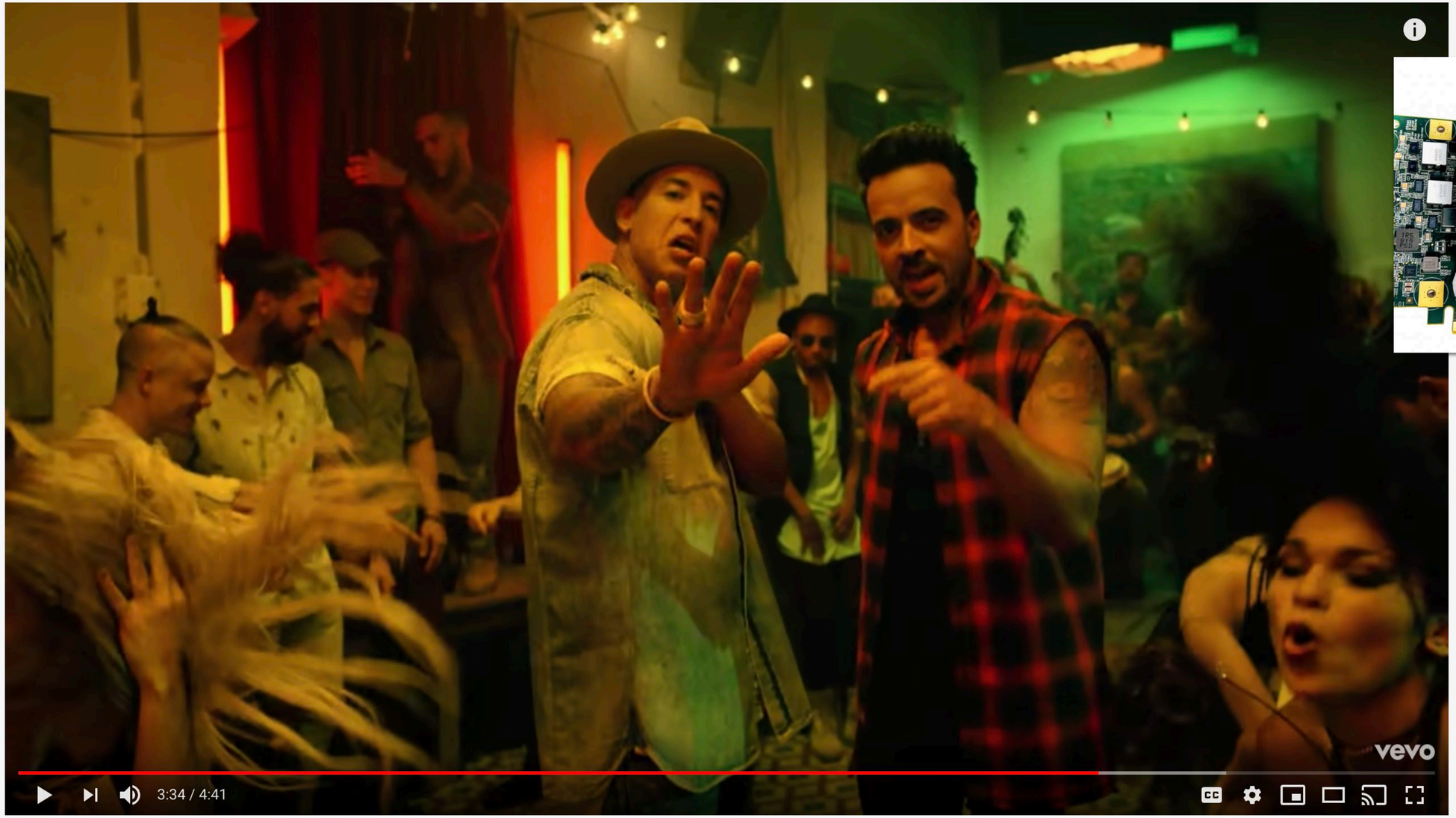


Google TPU pods

Image Credit: TechInsights Inc.

Youtube

Transcode, stream, analyze…

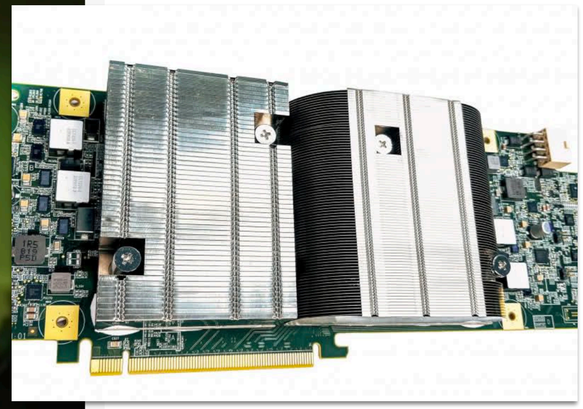


#LuisFonsi #Despacito #Imposible

Luis Fonsi - Despacito ft. Daddy Yankee

6,703,305,990 views • Jan 12, 2017

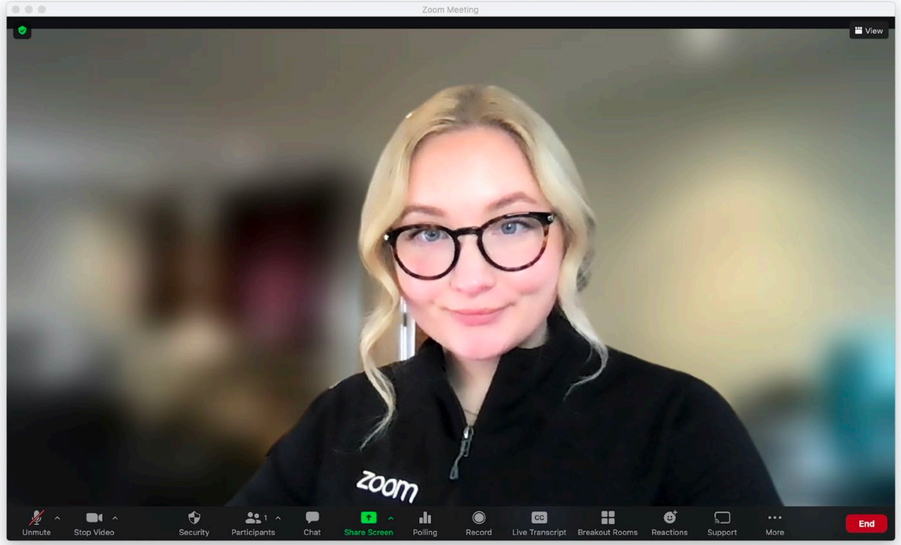
36M 4.4M SHARE SAVE •••

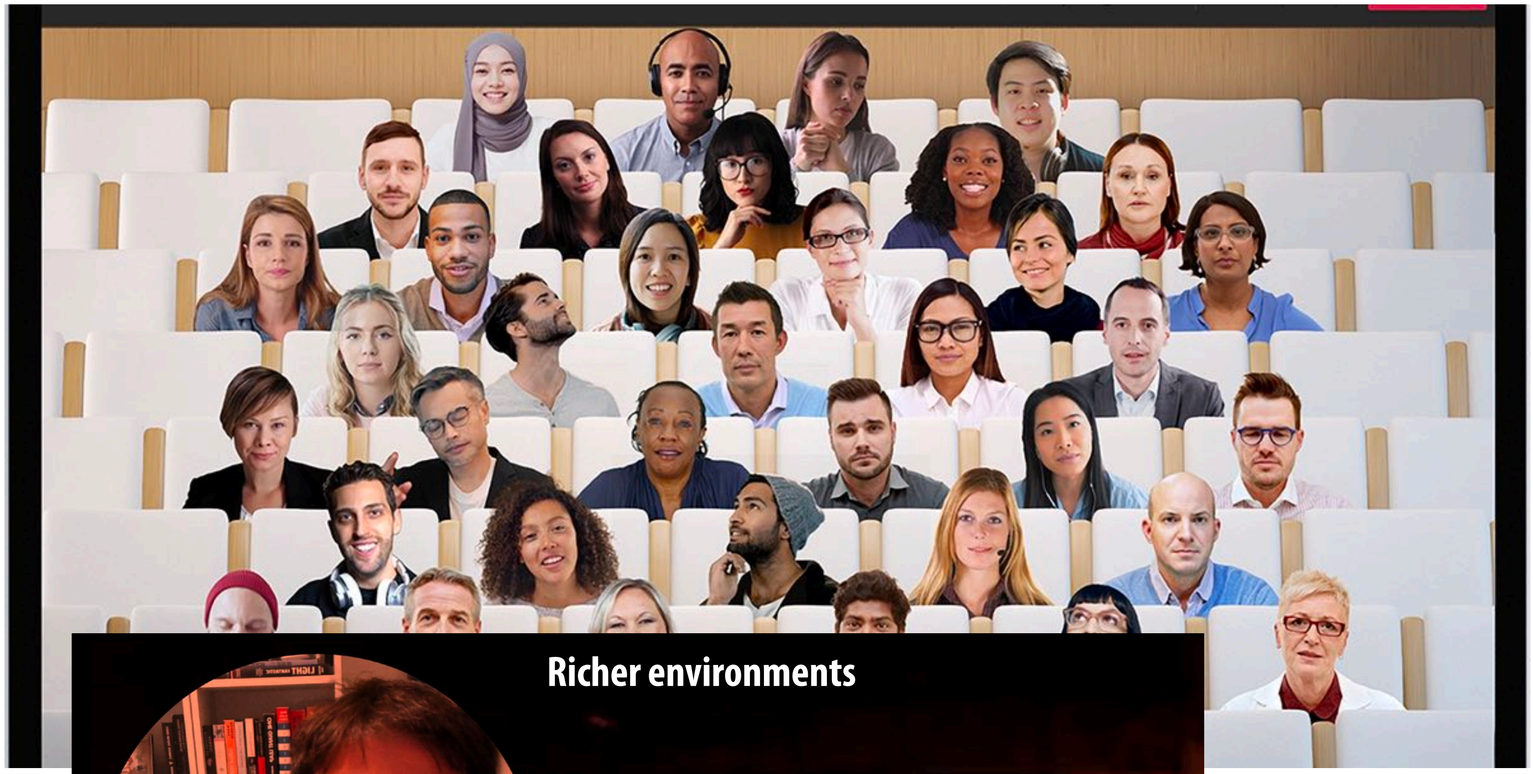


Google VPU transcoding HW

Video conferencing

Background blur





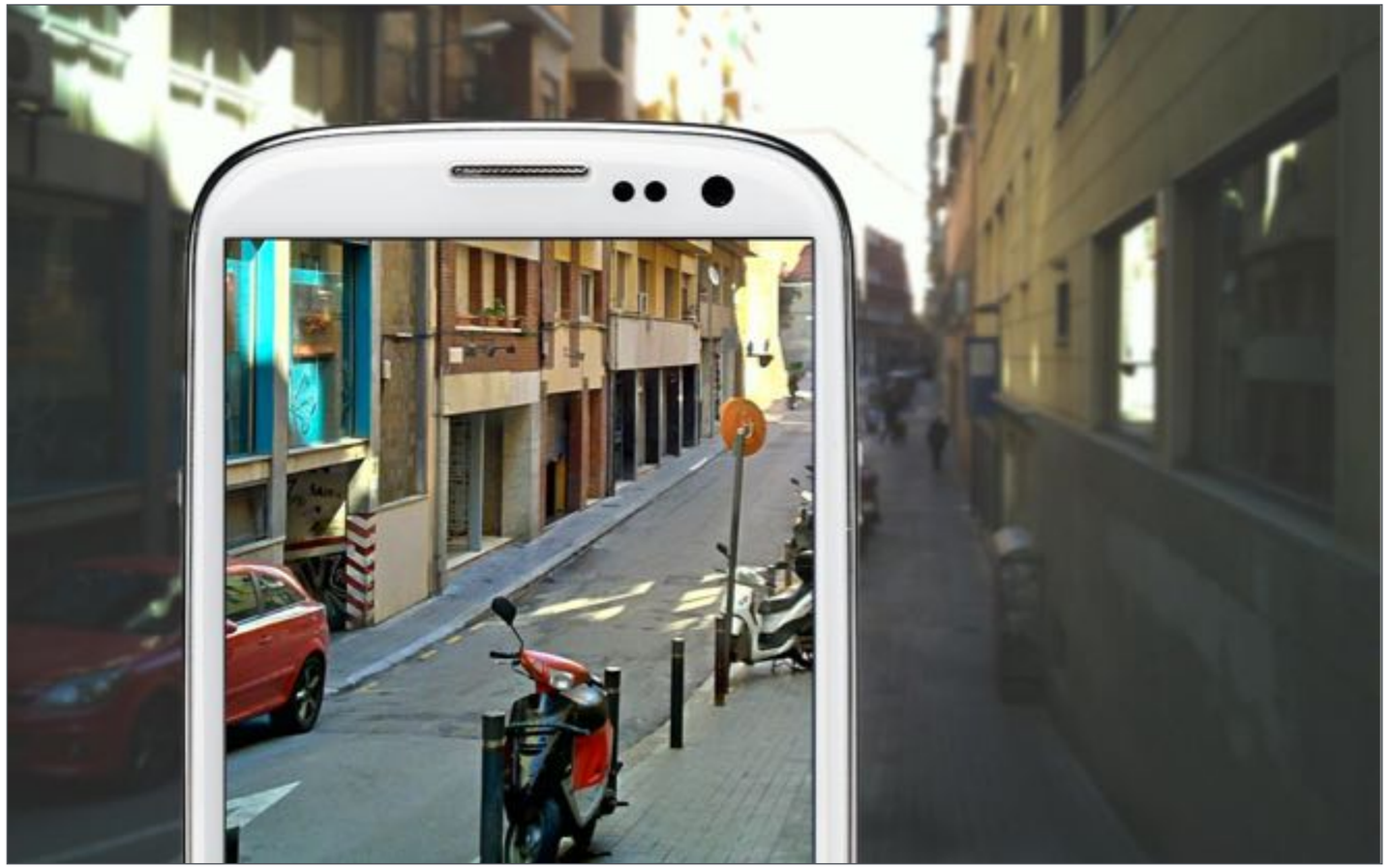
Richer environments

Add effects

Digital photography: major driver of compute capability of modern smartphones

Portrait mode
(simulate effects of large aperture DSLR lens)

High dynamic range (HDR) photography



Modern smartphones utilize multiple processing units to quickly generate high-quality images

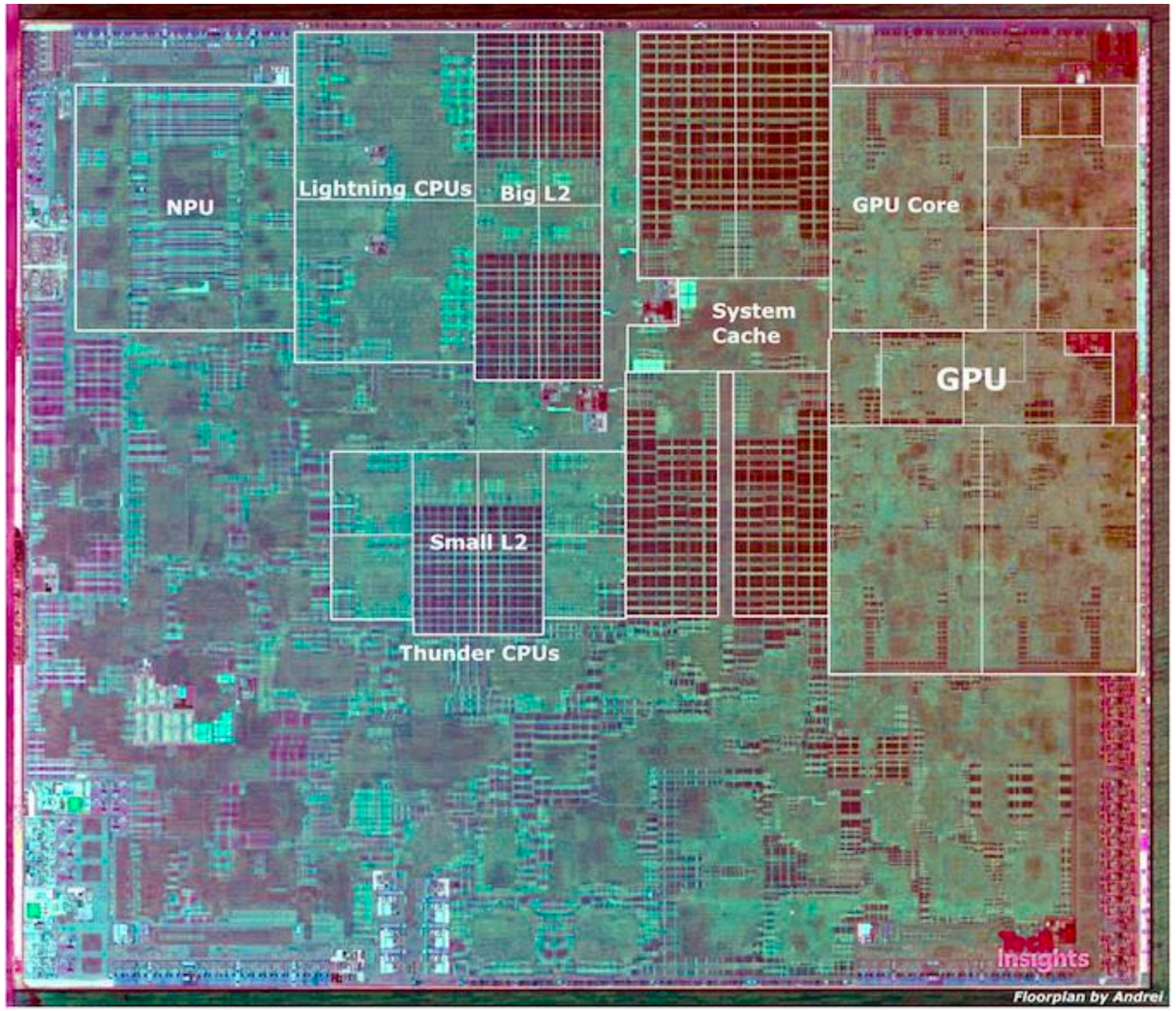


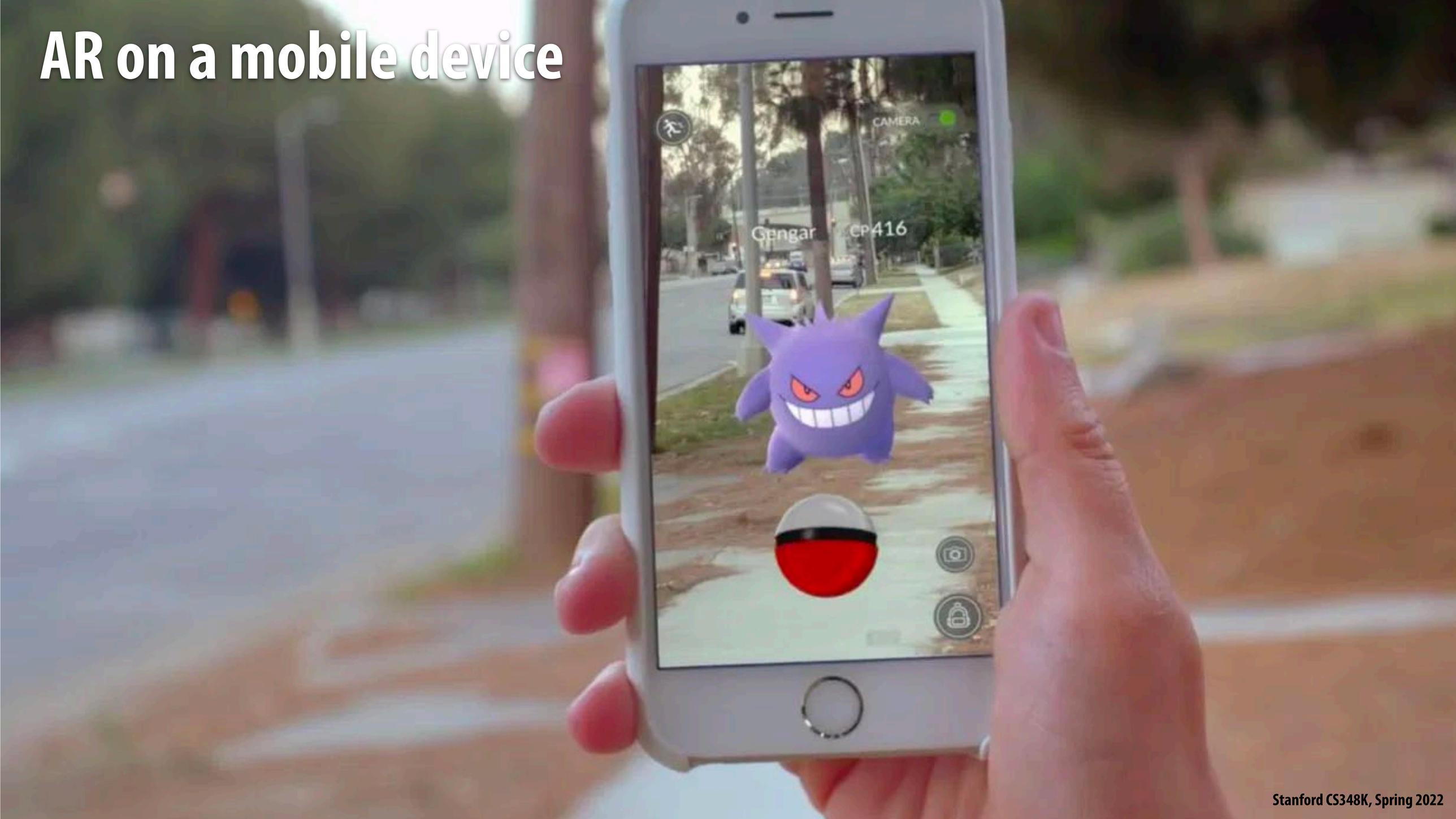
Image Credit: Anandtech / TechInsights Inc.

Apple A13 Bionic

Multi-core CPU (heterogeneous cores)
Multi-core GPU
Neural accelerator
Sensor processing accelerator
Video compression/decompression HW
Etc…

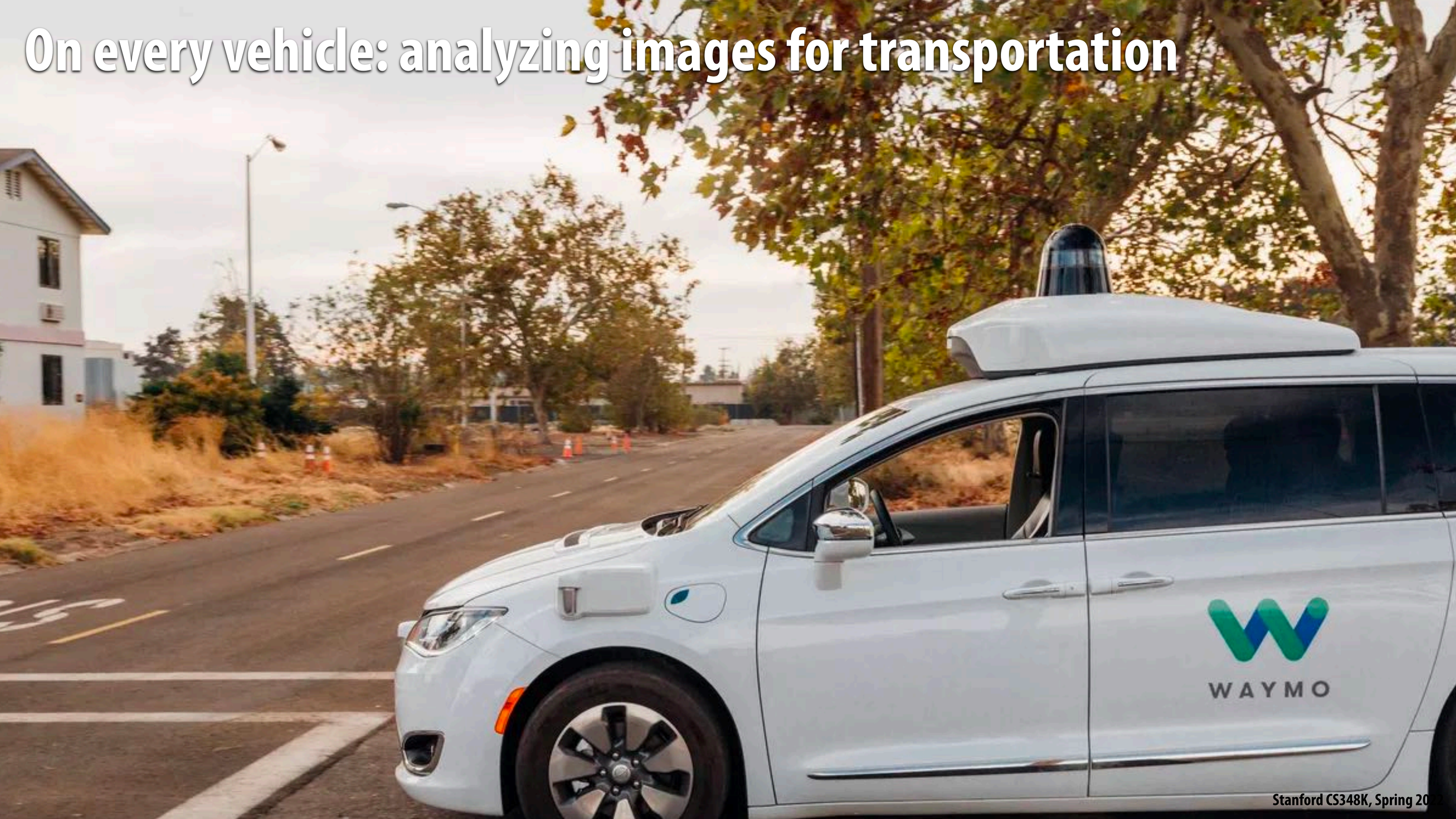
Oculus Quest 2 headset (2020)

AR on a mobile device



Snap AR Spectacles

On every vehicle: analyzing images for transportation



What is this course about?

Accelerator hardware architecture?

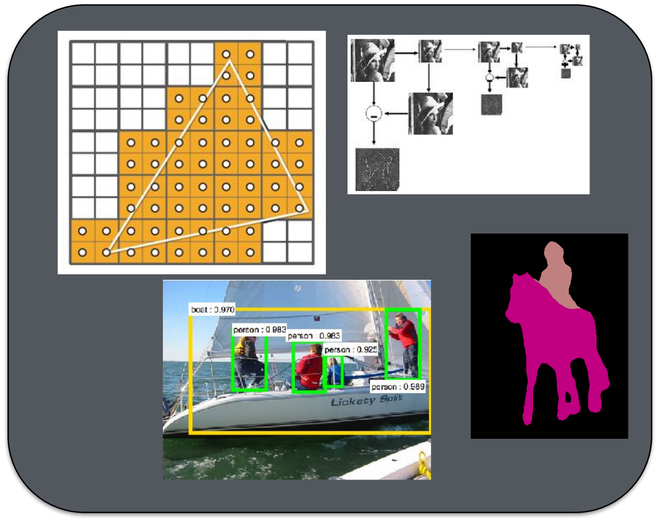
Graphics/vision/digital photography algorithms?

Programming systems?

What we will be learning about

Visual Computing Workloads

Algorithms for image/video processing, DNN evaluation, data compression, etc.



If you don't understand key workload characteristics, how can you design a "good" system?

What we will be learning about

Modern Hardware Organization

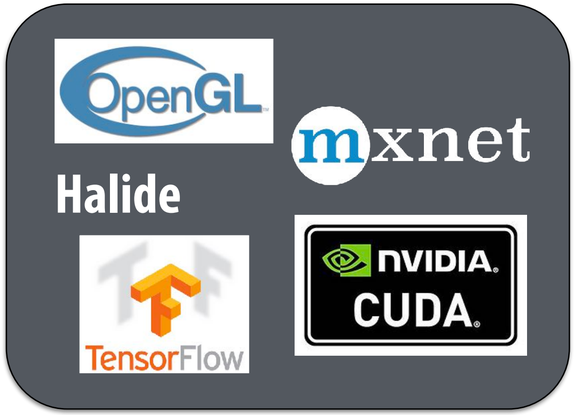
High-throughput hardware designs
(parallel, heterogeneous, and specialized)
fundamental constraints like area and power

If you don't understand key constraints of modern hardware, how can you design algorithms that are well suited to run on it efficiently?

What we will be learning about

Programming Model Design

Choice of programming abstractions,
level of abstraction issues,
domain-specific vs. general purpose, etc.



Good programming abstractions enable productive development of applications, while also providing system implementors flexibility to explore highly efficient implementations

This course is about architecting efficient and scalable systems…

It is about the process of understanding the fundamental structure of problems in the visual computing domain, and then leveraging that understanding to…

To design more efficient and more robust algorithms

To build the most efficient hardware to run these algorithms

To design programming systems to make developing new applications simpler, more productive, and highly performant

2022 course topics

The digital camera photo processing pipeline in modern smartphones

- Basic algorithms (the workload)
- Programming abstractions for writing image processing apps
- Mapping these algorithms to parallel hardware

Systems for creating fast and accurate deep learning models

- Designing efficient DNN topologies, and scheduling them on modern CPUs/GPUs
- Hardware for accelerating deep learning (why GPUs are not efficient enough!)
- Raising level of abstraction when designing models
- System support for automating data labeling

Processing and Transmitting Video

- Efficient DNN inference on video
- Trends in video compression (neural techniques)
- How modern video conferencing systems work, and what new experiences are on the horizon

Recent advances in real-time (hardware accelerated) rendering

- Advanced rasterization in energy-constrained mobile environments
- Recent API and hardware support for real-time ray tracing
- How deep learning, combined with RT hardware, is making real time ray tracing possible

+ a few assorted topics…

- Resigning renderers as simulation engines for ML
- A few guest speakers from industry

Logistics and Expectations

Logistics

- Course web site:
  - http://cs348k.stanford.edu
  - My goal is to post lecture slides the night before class

- All announcements will go out via Ed Discussion (not via Canvas)

My expectations of you

- 40% participation
  - There will be ~1 assigned technical paper reading per class
  - You will submit a response to each reading by 11am on class days via Gradescope
  - We will start most classes with a 30-45 minute discussion of the reading

This is so important. You've got to do the reasons and come to class to make the course tick.

- 20% two programming assignments (first 1/2 of course)
  - Implement and optimize a simple HDR photography processing pipeline
  - Understanding why "blocking" a conv layer in a DNN matters

- 40% self-selected term project
  - I suggest you start thinking about projects now

Review (or crash course):

key principles of modern throughput computing hardware

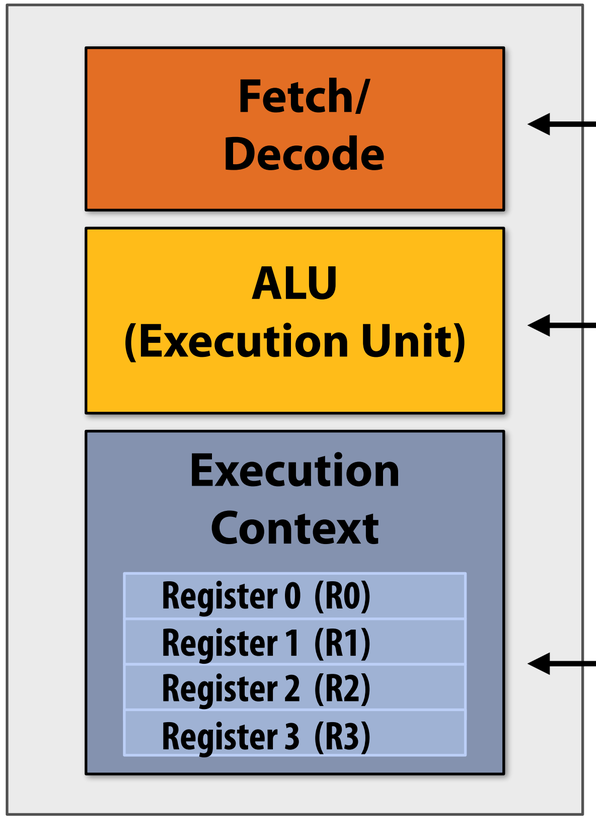
Concept #1:
The high cost of data communication

(Almost everything we talk about in this course starts from this concept)

A basic CPU that executes instructions

A processor executes instructions

Professor Kayvon's
Very Simple Processor

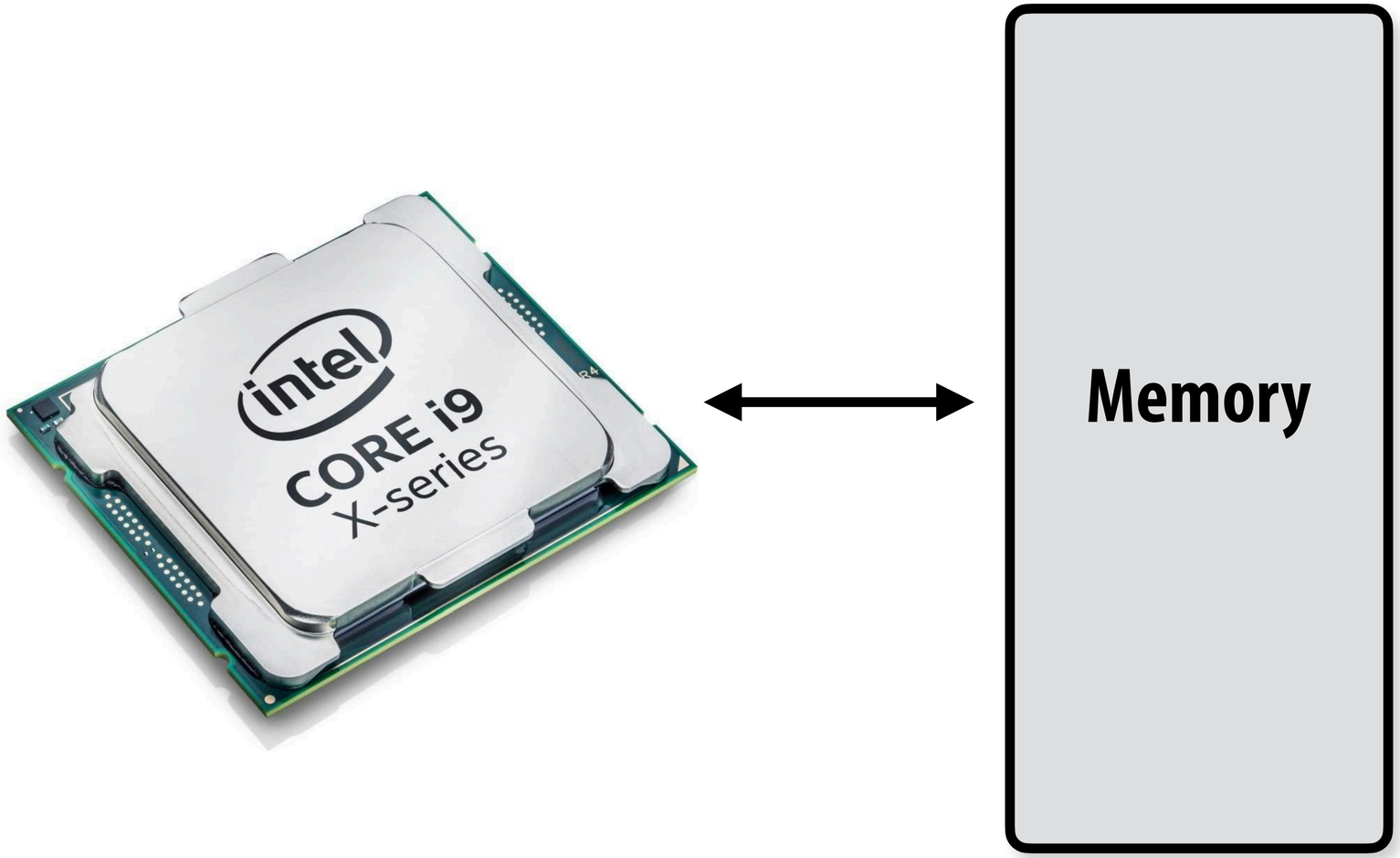


Determine what instruction to run next

Execution unit: performs the operation described by an instruction, which may modify values in the processor's registers or the computer's memory

Registers: maintain program state: store value of variables used as inputs and outputs to operations

But 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)

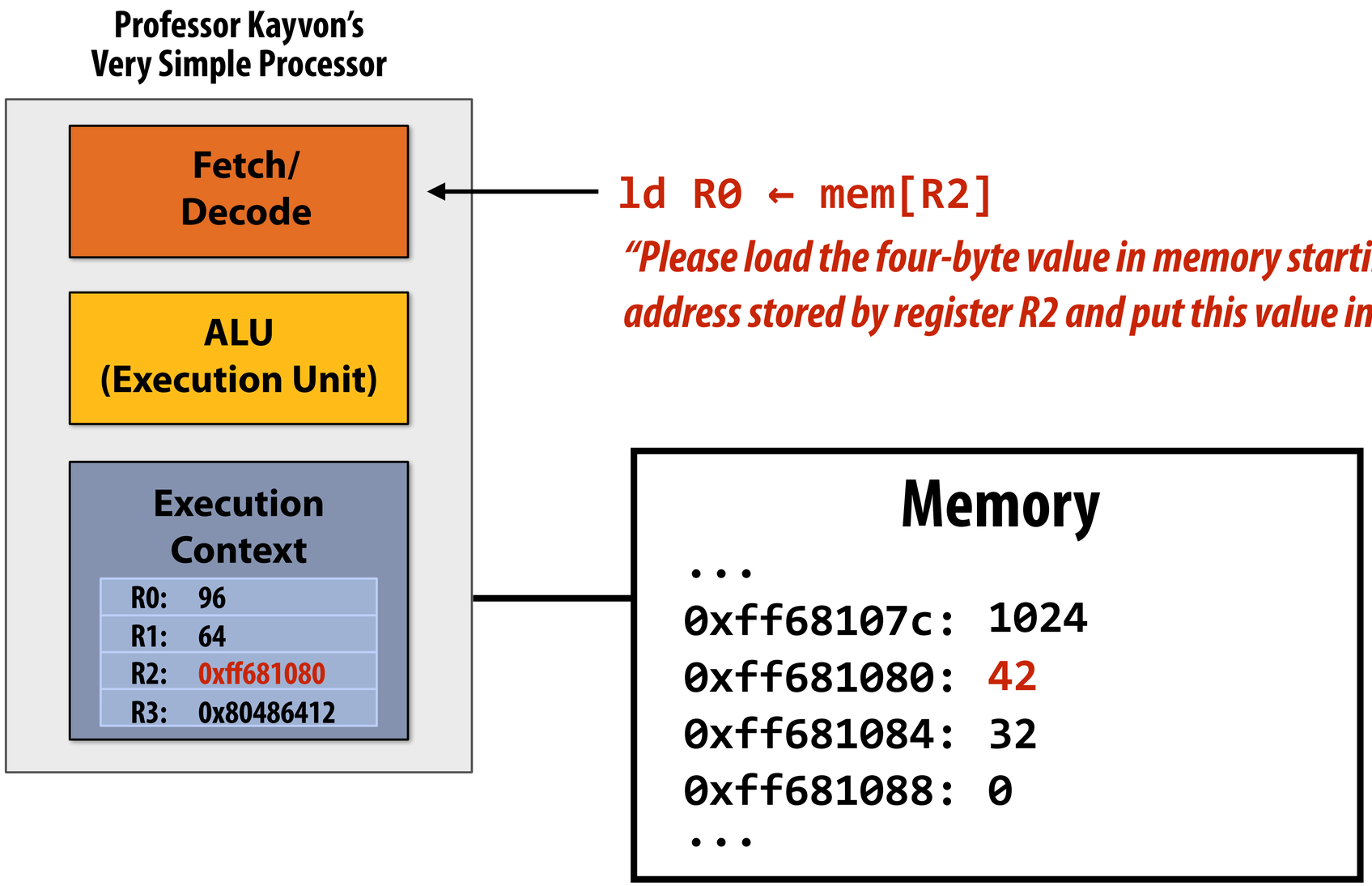
"The byte stored at address 0x8 has the value 32."

"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

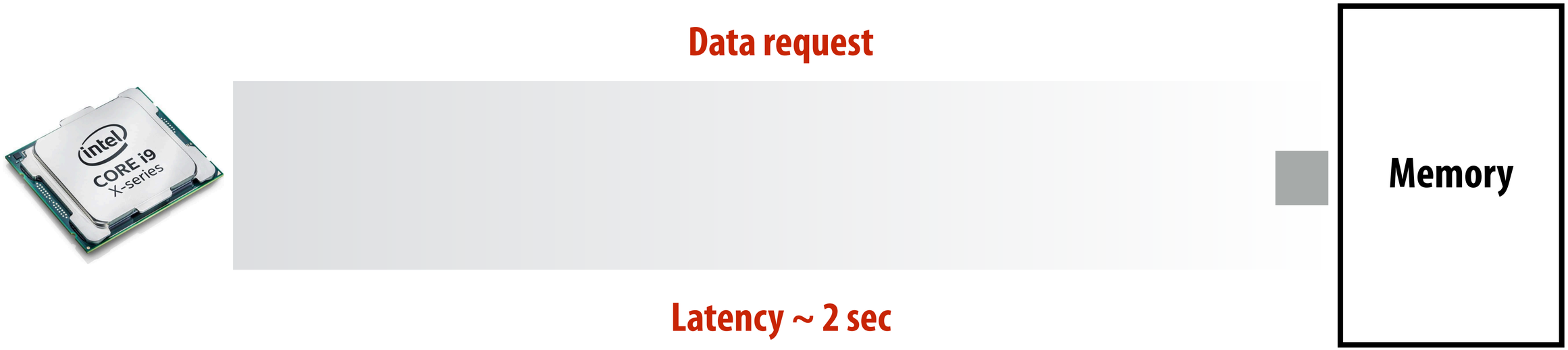


`ld R0 ← mem[R2]`

"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

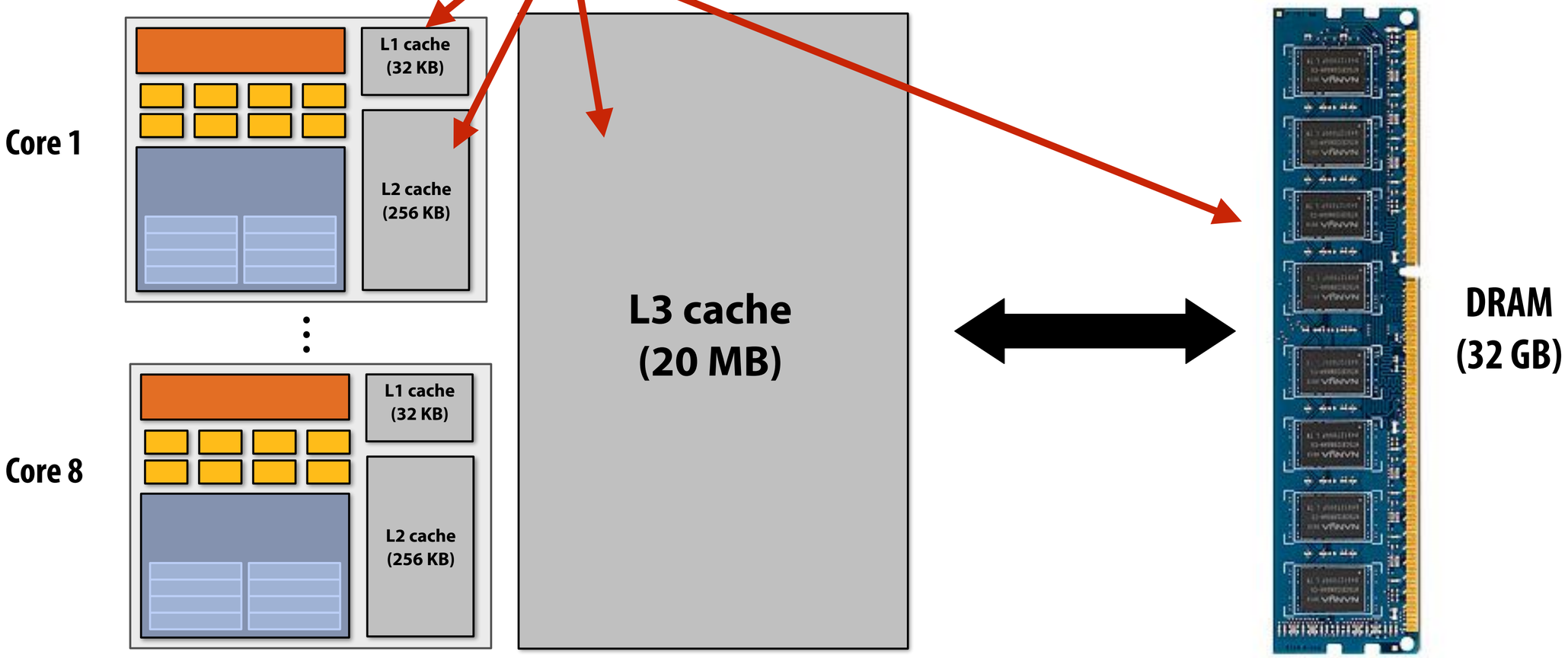
```
ld r0 mem[r2]

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

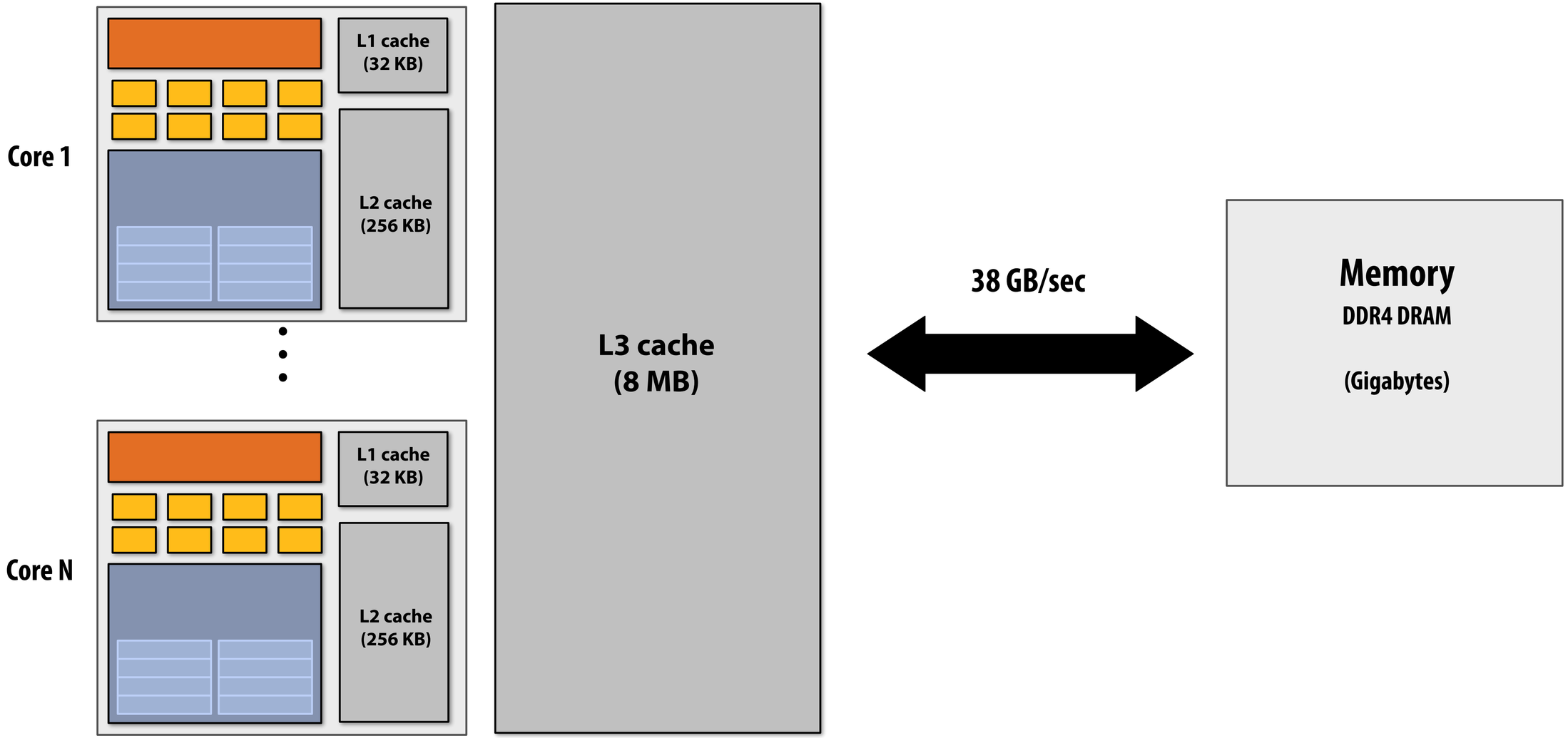
- Memory access times ~ 100's of cycles
  - Memory "access time" is a measure of latency

The implementation of the linear memory address space abstraction on a modern computer is complex

The instruction "load the value stored at address X into register R0" might involve a complex sequence of operations by multiple data caches and access to DRAM



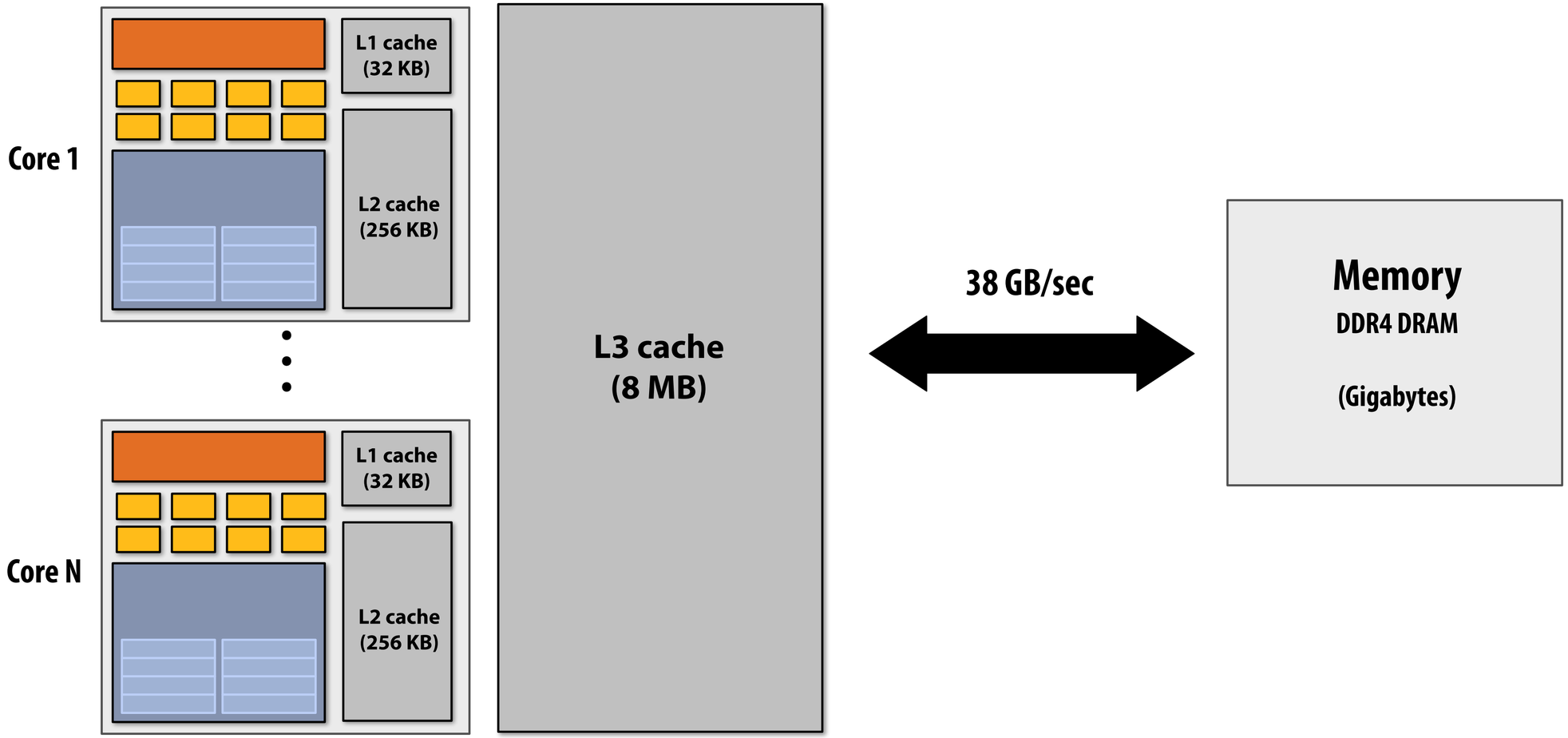
Why do modern processors have data caches?



Caches reduce length of stalls (reduce memory access latency)

Processors run efficiently when data is resident in caches

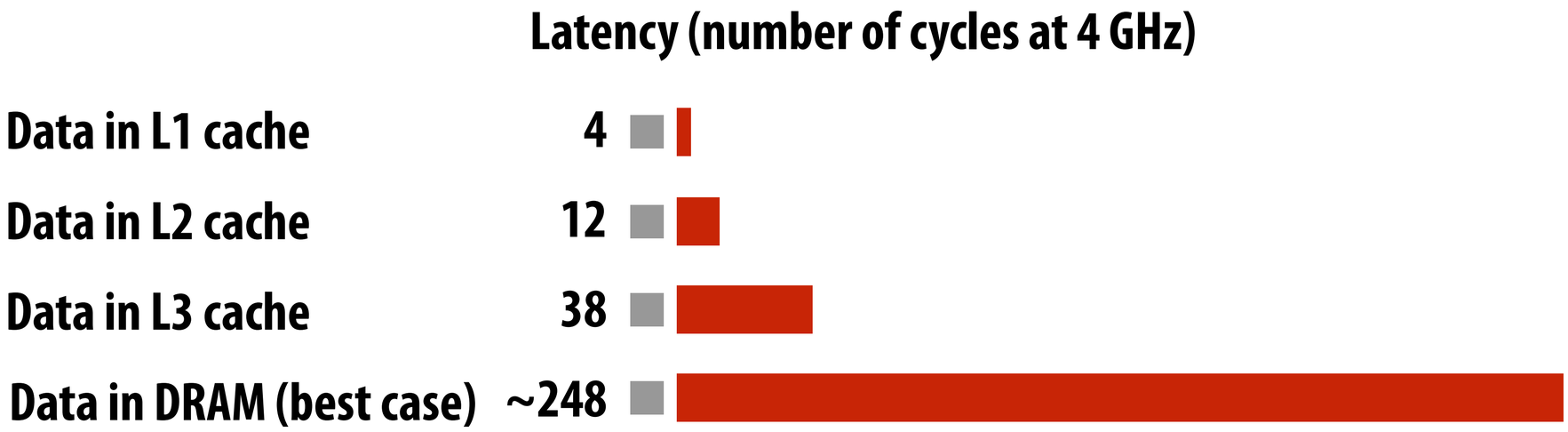
Caches reduce memory access latency *



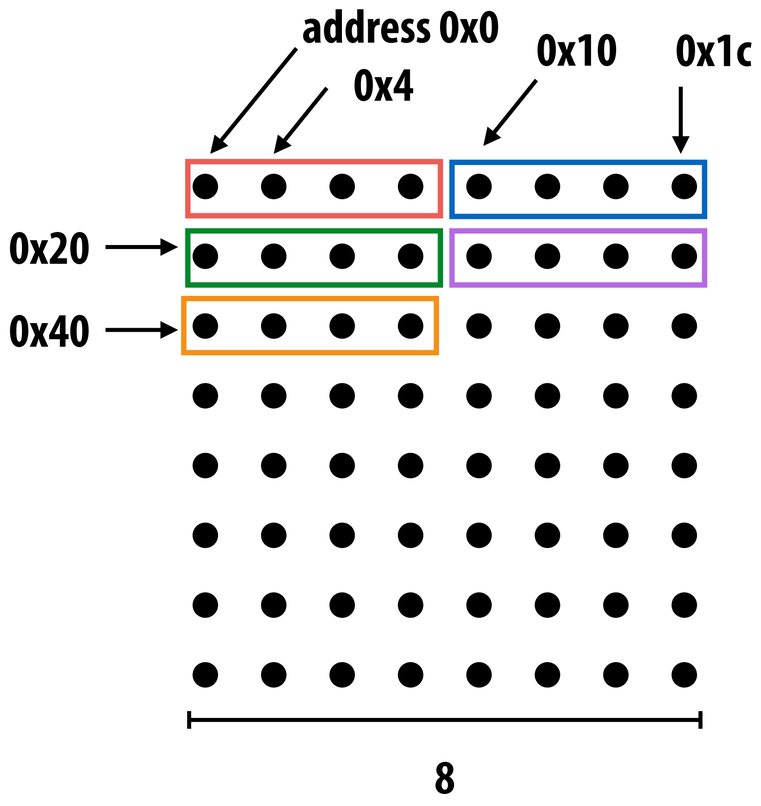
* Caches also provide high bandwidth data transfer to CPU

Data access times

(Kaby Lake CPU)



Cache review

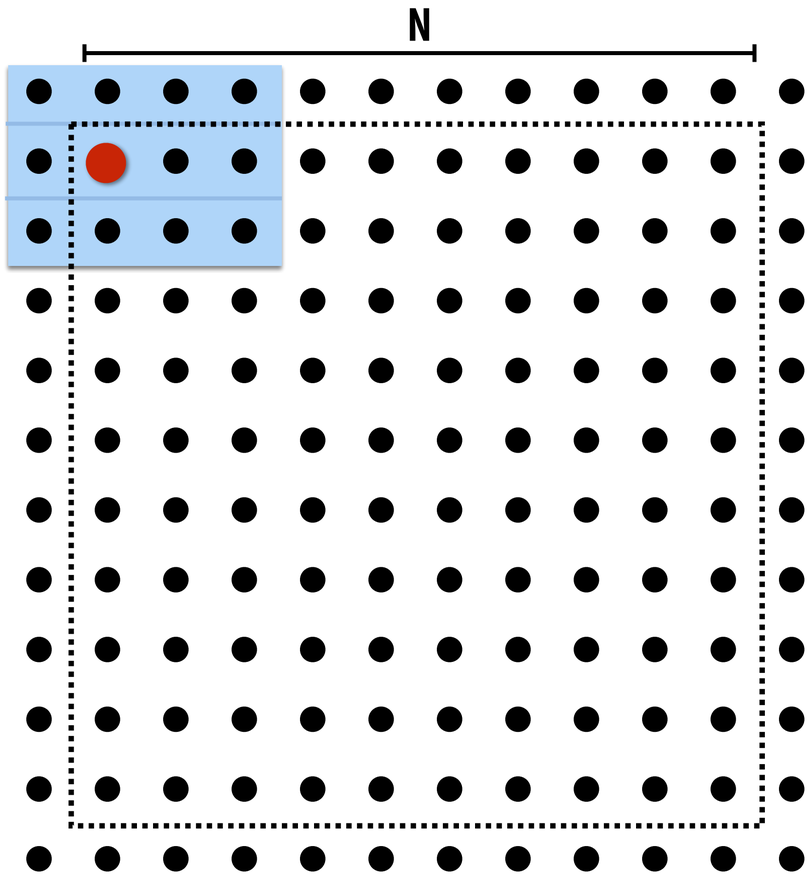


Address accessed	Cache state (after load is complete)		
0x0	0x0 ●●●●		"cold miss"
0x4	0x0 ●●●●		hit
0x8	0x0 ●●●●		hit
0xc	0x0 ●●●●		hit
0x10	0x0 ●●●●	0x10 ●●●●	cold miss
0x14	0x0 ●●●●	0x10 ●●●●	hit
0x18	0x0 ●●●●	0x10 ●●●●	hit
0x1c	0x0 ●●●●	0x10 ●●●●	hit
0x20	0x20 ●●●●	0x10 ●●●●	cold miss (evict 0x0)
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)

Consider 4-byte elements
Consider a cache with 16-byte cache lines and a total capacity of 32 bytes (2 lines fit in cache)
Least recently used (LRU) replacement policy

Data access in grid solver: row-major traversal

"Blocking": reorder computation to make working sets map well to system's memory hierarchy



Assume row-major grid layout.

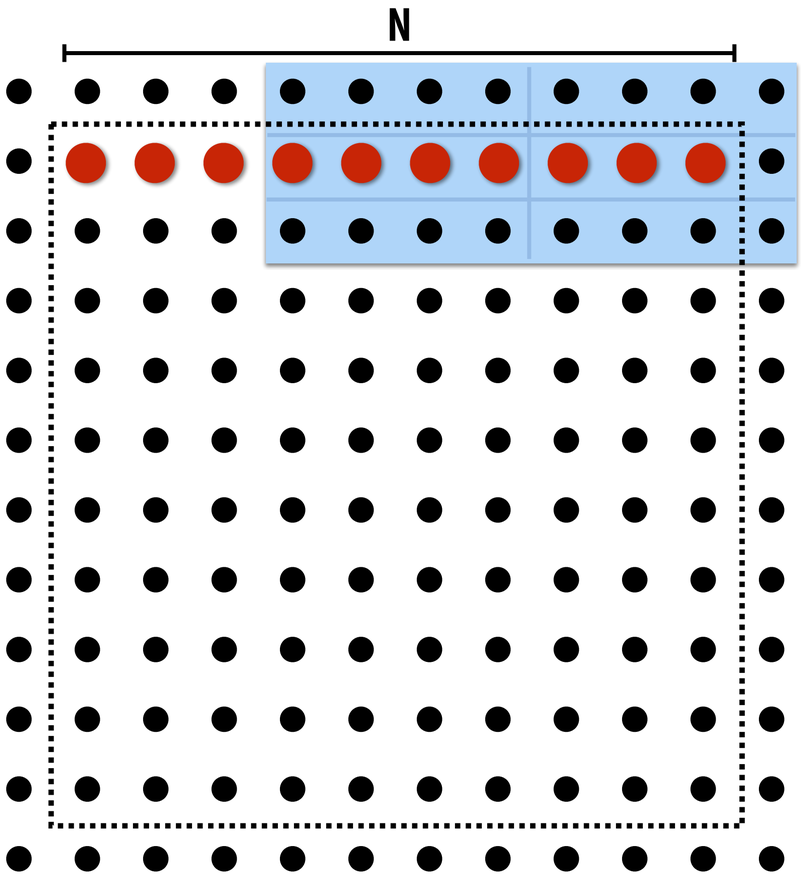
Assume cache line is 4 grid elements.

Cache capacity is 24 grid elements (6 lines)

Recall grid solver application.
Blue elements show data that is in cache after update to red element.

Data access in grid solver: row-major traversal

"Blocking": reorder computation to make working sets map well to system's memory hierarchy



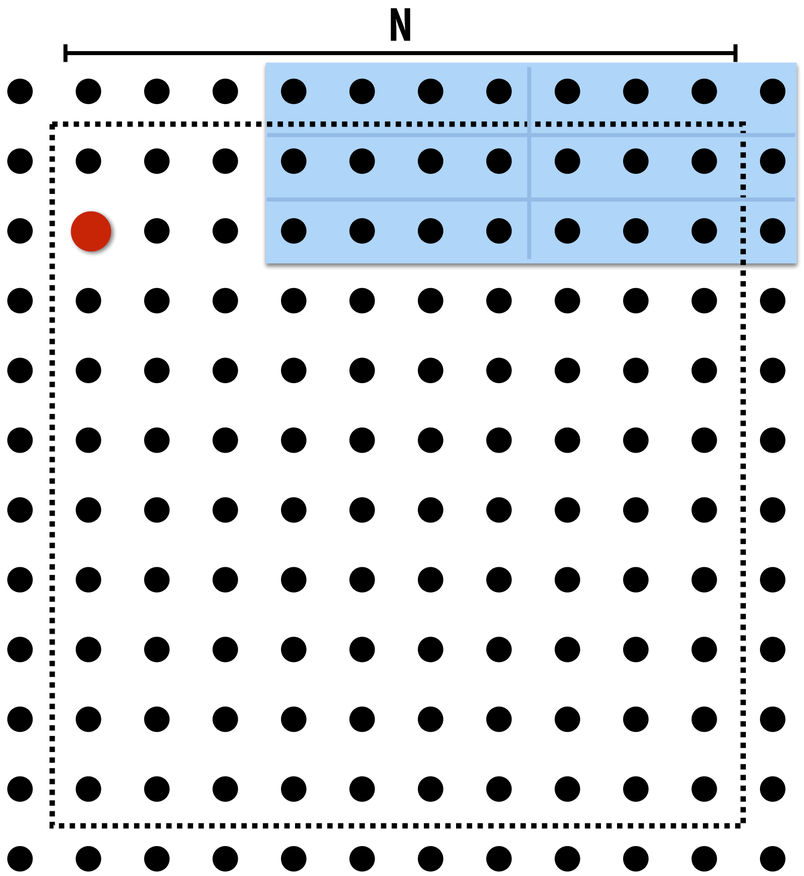
Assume row-major grid layout.

Assume cache line is 4 grid elements.

Cache capacity is 24 grid elements (6 lines)

Blue elements show data in cache at end of processing first row.

Problem with row-major traversal: long time between accesses to same data



Assume row-major grid layout.

Assume cache line is 4 grid elements.

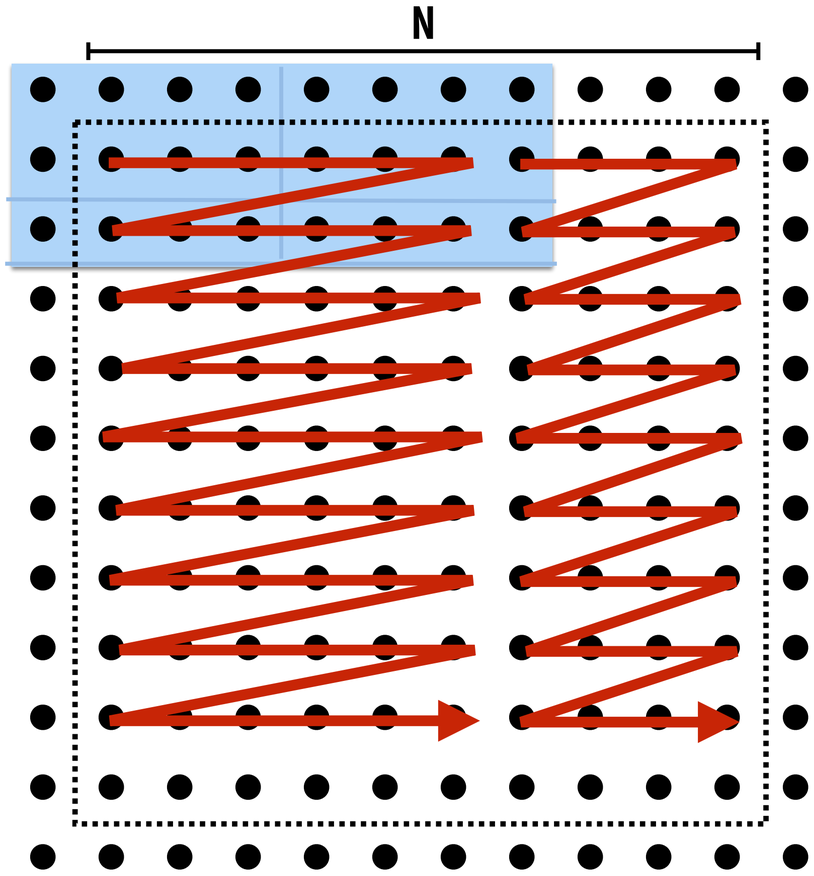
Cache capacity is 24 grid elements (6 lines)

Although elements (0,2) and (0,1) had been accessed previously, they are no longer present in cache at start of processing row 2.

This program loads three lines for every four elements of output.

Improving temporal locality by changing grid traversal order

"Blocking": reorder computation to make working sets map well to system's memory hierarchy



Assume row-major grid layout.

Assume cache line is 4 grid elements.

Cache capacity is 24 grid elements (6 lines)

"Blocked" iteration order

(diagram shows state of cache after finishing work from first row of first block)

Now load two cache lines for every six elements of output

Improving temporal locality by "fusing" loops

```
void add(int n, float* A, float* B, float* C) {
    for (int i=0; i<n; i++)
       C[i] = A[i] + B[i];
}

void mul(int n, float* A, float* B, float* C) {
    for (int i=0; i<n; i++)
       C[i] = A[i] * B[i];
}


float* A, *B, *C, *D, *E, *tmp1, *tmp2;

// assume arrays are allocated here

// compute E = D + ((A + B) * C)
add(n, A, B, tmp1);
mul(n, tmp1, C, tmp2);
add(n, tmp2, D, E);
```

Two loads, one store per math op (arithmetic intensity = 1/3)

Two loads, one store per math op (arithmetic intensity = 1/3)

Overall arithmetic intensity = 1/3

```
void fused(int n, float* A, float* B, float* C, float* D, float* E) {
    for (int i=0; i<n; i++)
       E[i] = D[i] + (A[i] + B[i]) * C[i];
}

// compute E = D + (A + B) * C
fused(n, A, B, C, D, E);
```

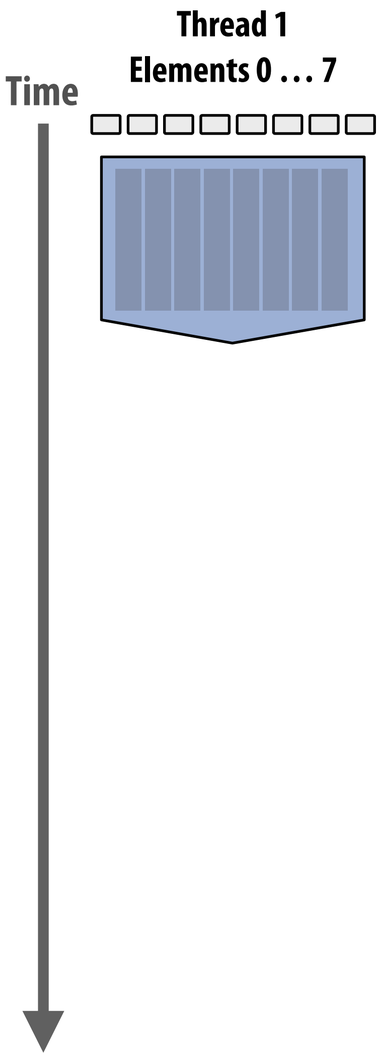
Four loads, one store per 3 math ops (arithmetic intensity = 3/5)

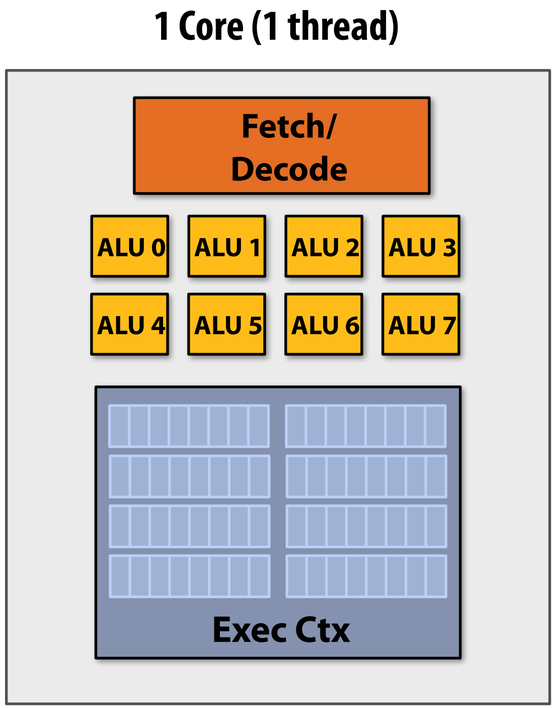
Code on top is more modular (e.g, array-based math library like numPy in Python)
Code on bottom performs much better. Why?

Multi-threading reduces stalls

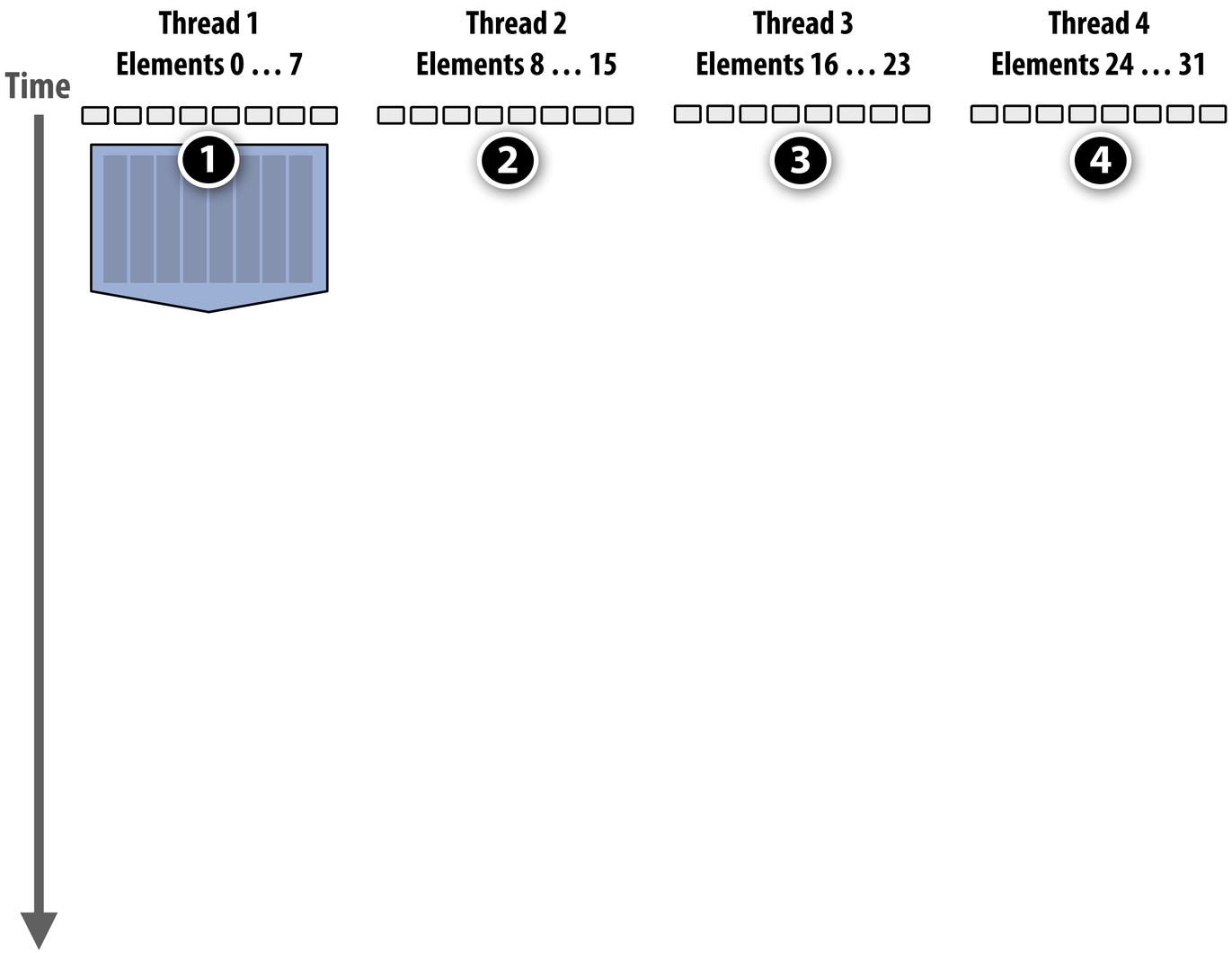
- Idea: <u>interleave</u> processing of multiple threads on the same core to hide stalls
  - If you can't make progress on the current thread… work on another one

Hiding stalls with multi-threading

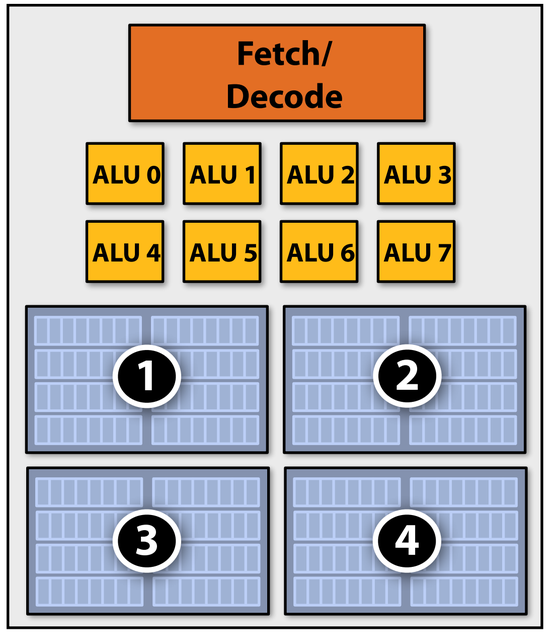




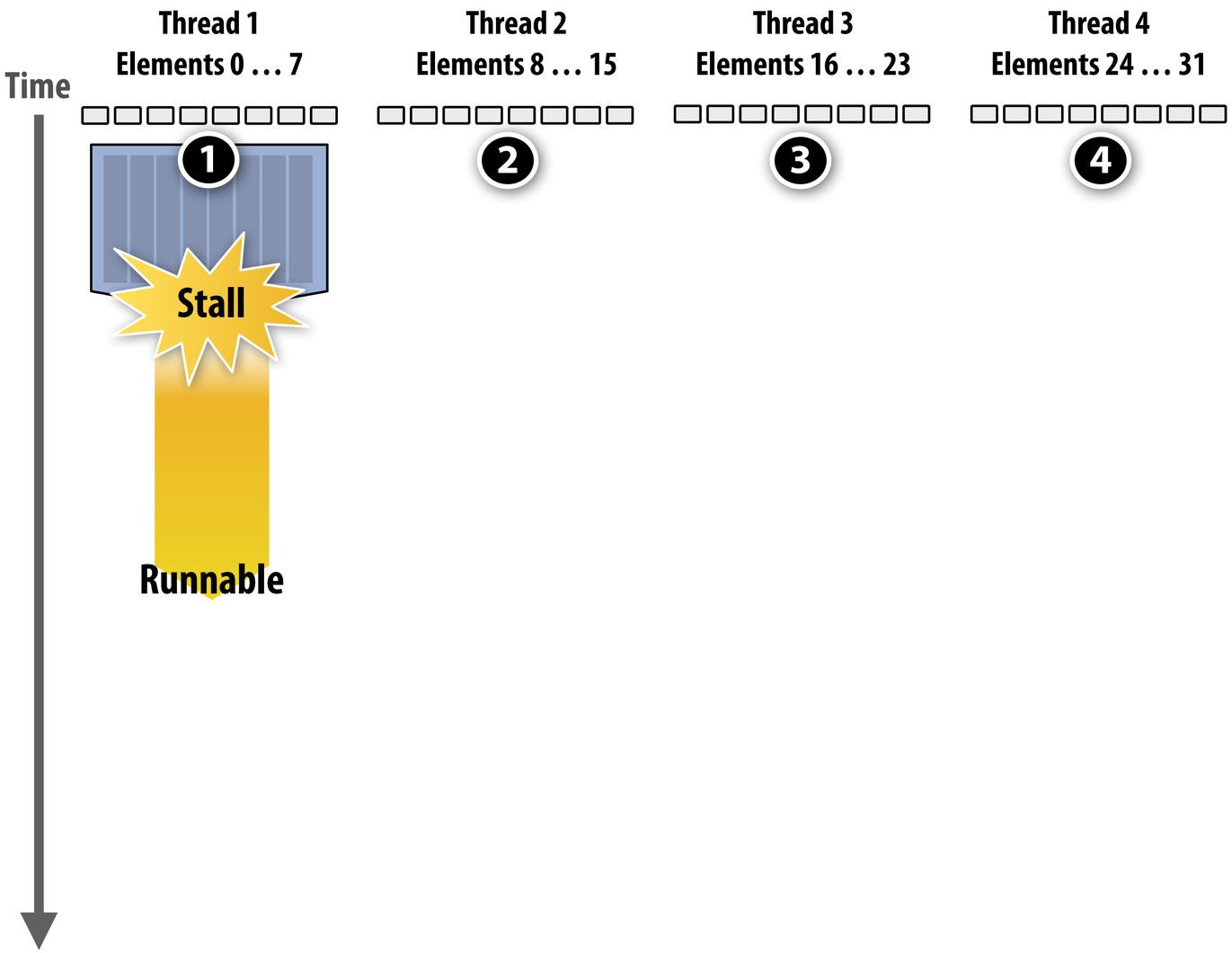
Hiding stalls with multi-threading



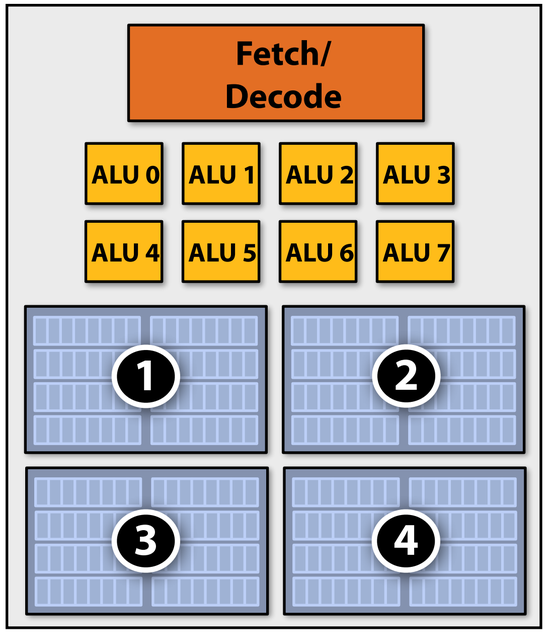
1 Core (4 hardware threads)



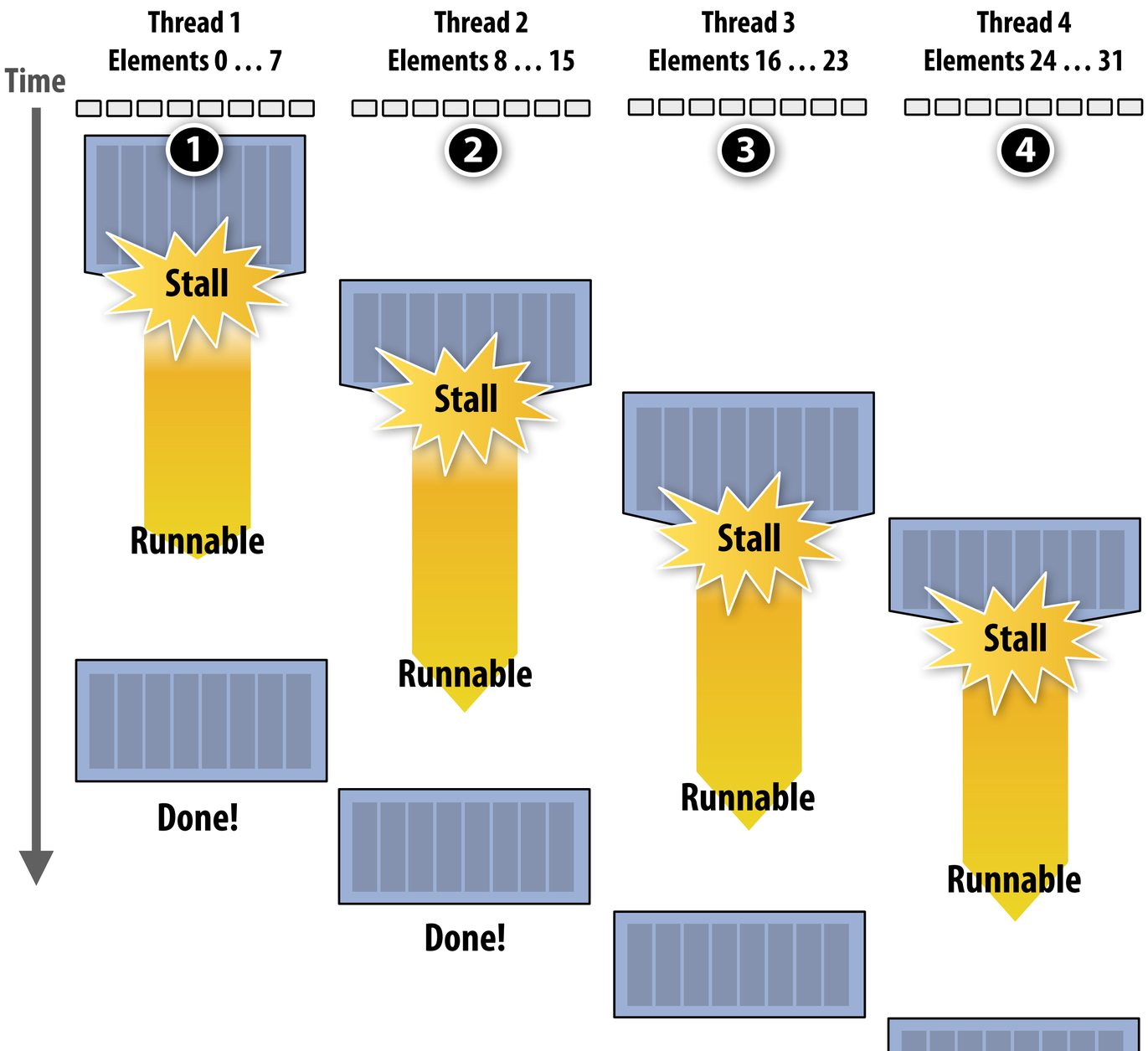
Hiding stalls with multi-threading



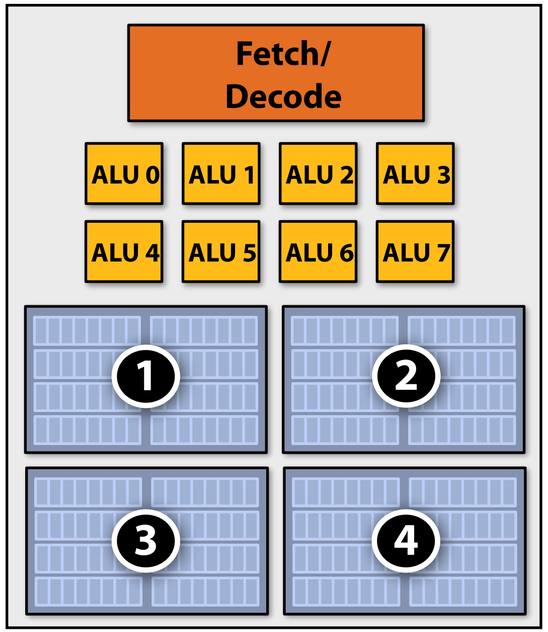
1 Core (4 hardware threads)



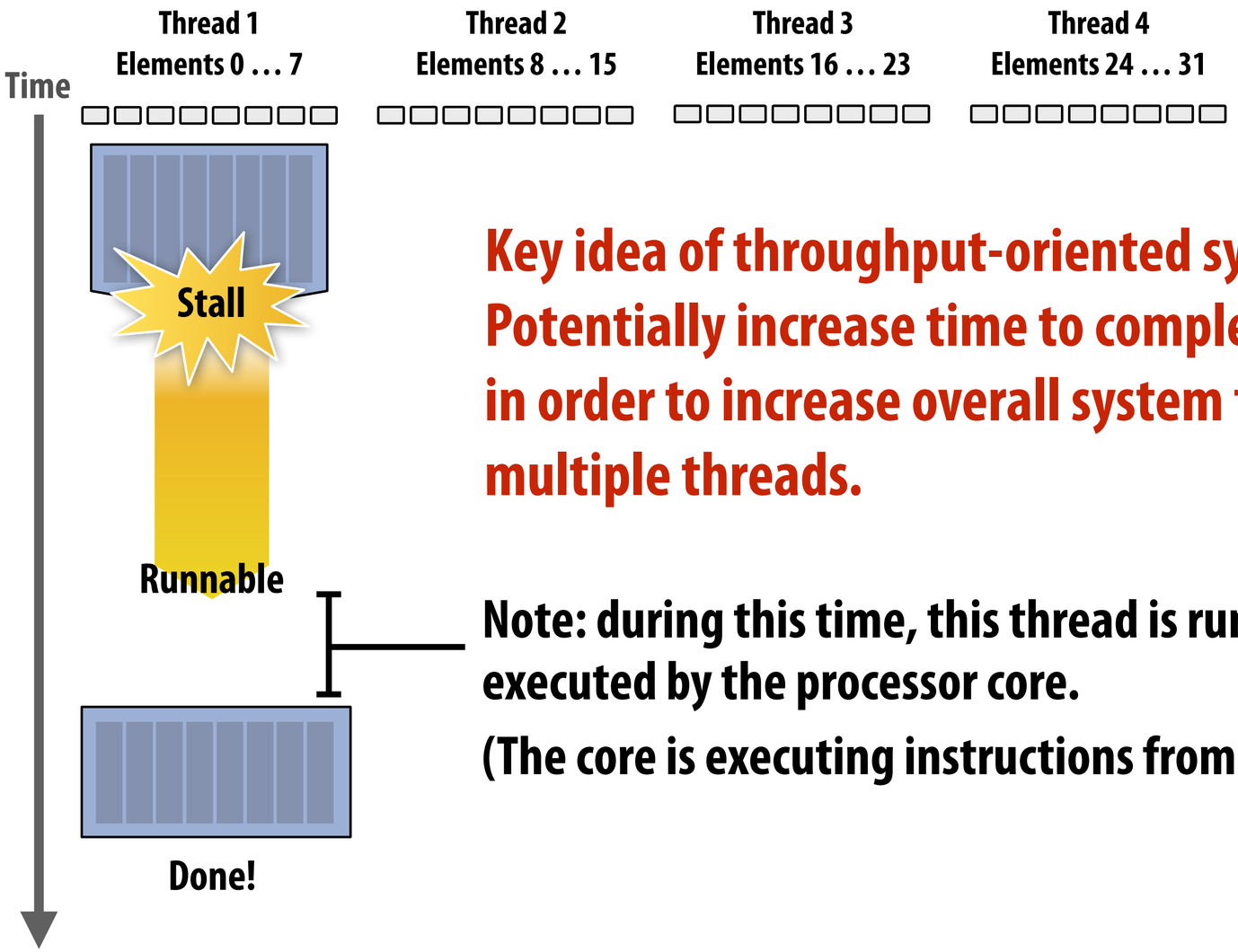
Hiding stalls with multi-threading



1 Core (4 hardware threads)



Throughput computing: a trade-off



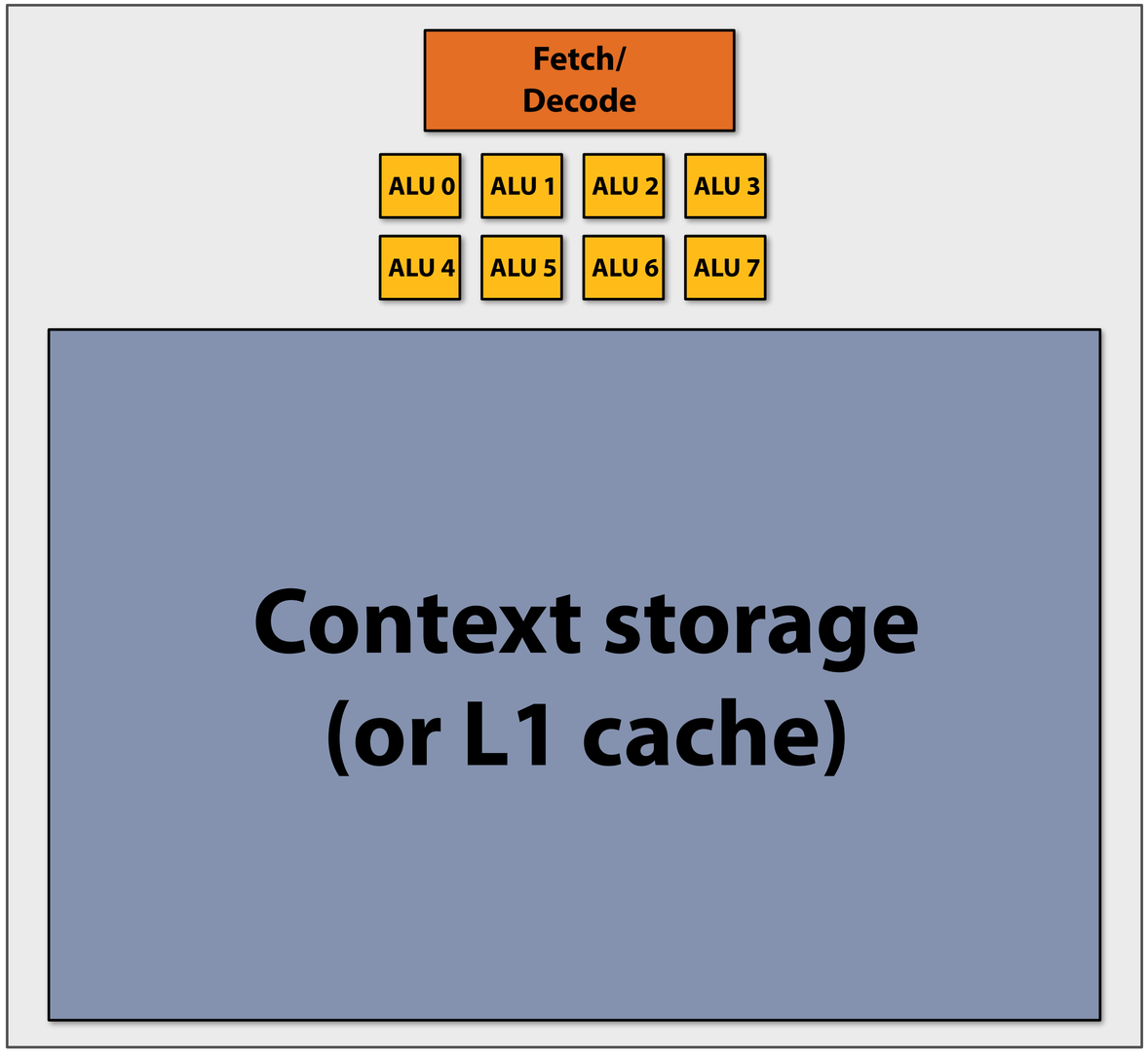
Key idea of throughput-oriented systems:
Potentially increase time to complete work by any one thread, in order to increase overall system throughput when running multiple threads.

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.)

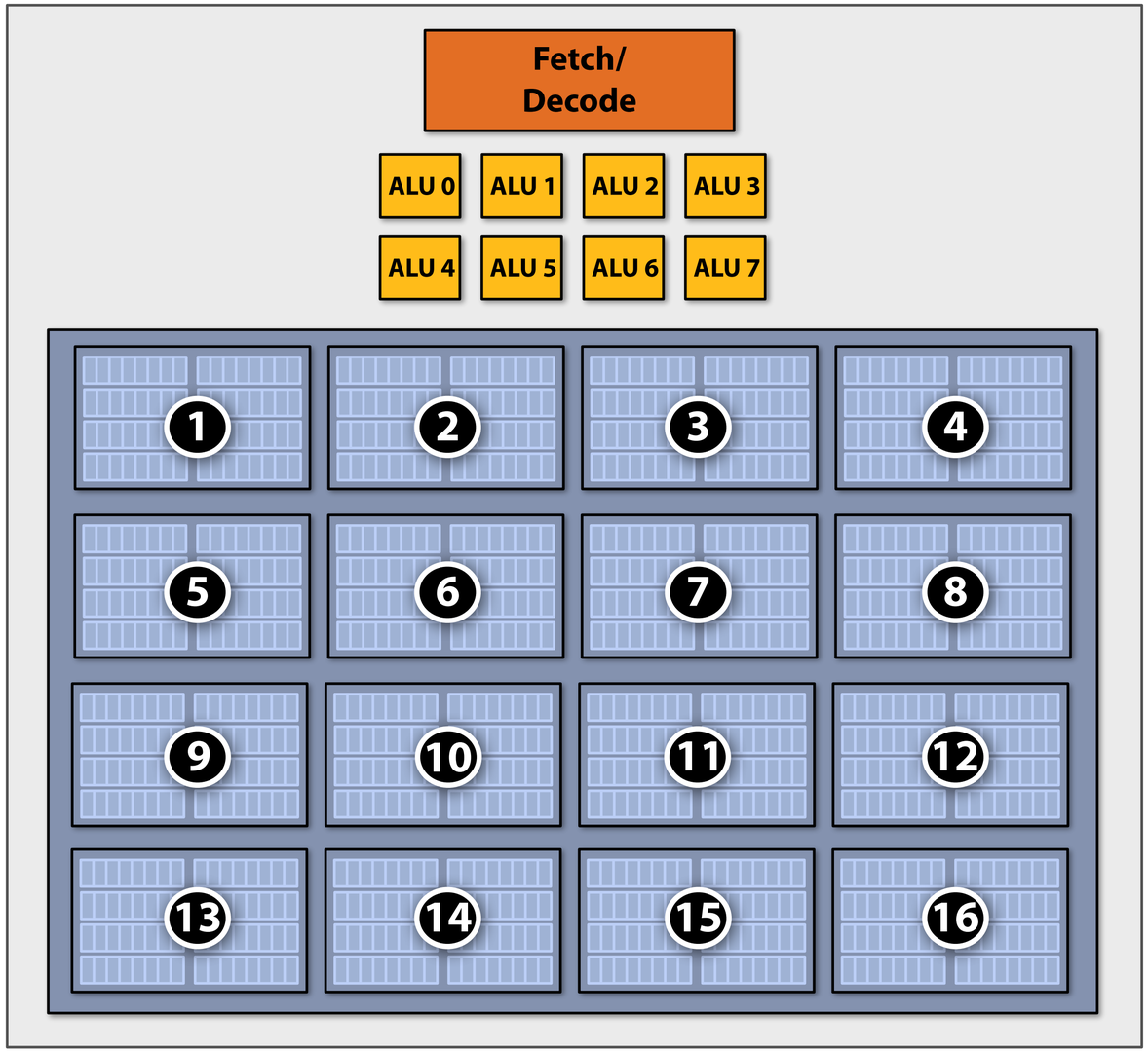
No free lunch: storing execution contexts

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



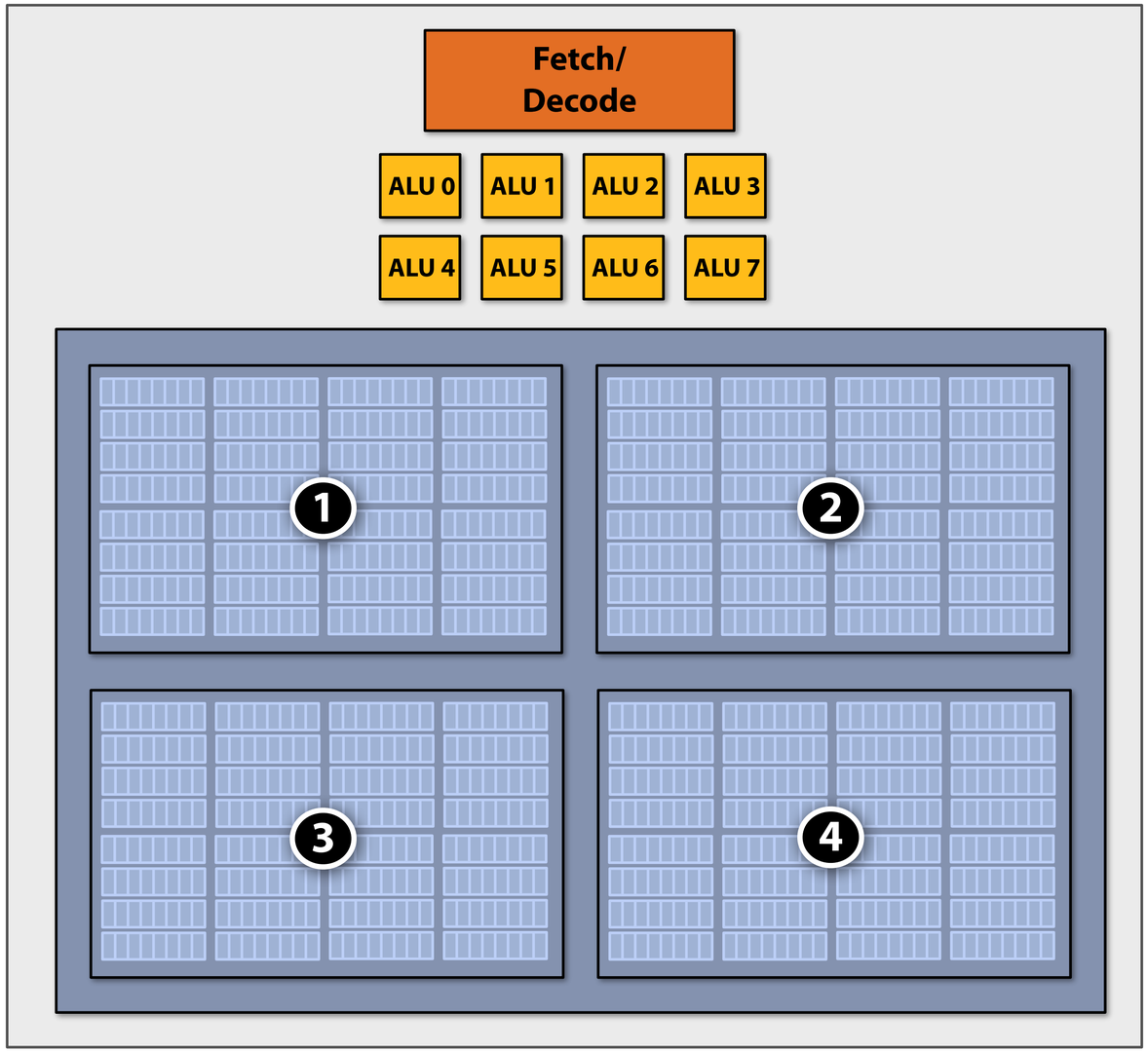
Many small contexts (high latency hiding ability)

16 hardware threads: storage for small working set per thread

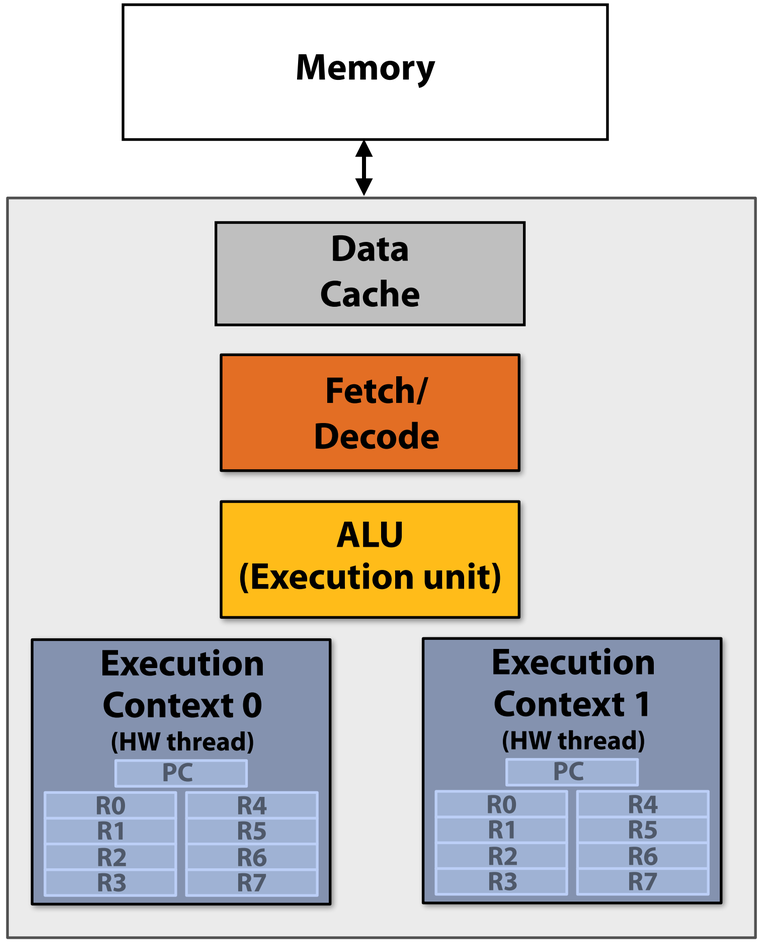


Four large contexts (low latency hiding ability)

4 hardware threads: storage for large working set per thread

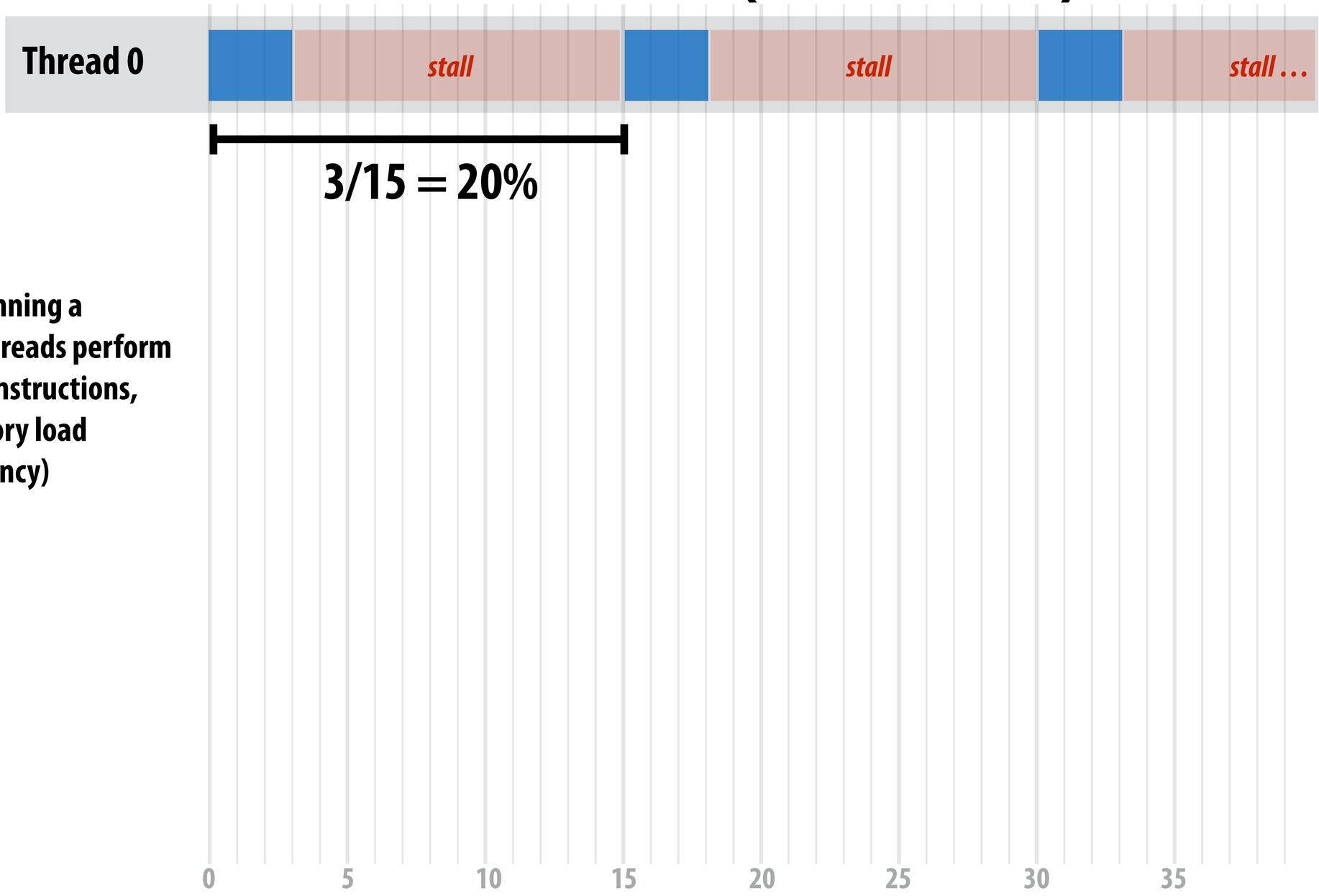


Exercise: consider a simple two threaded core



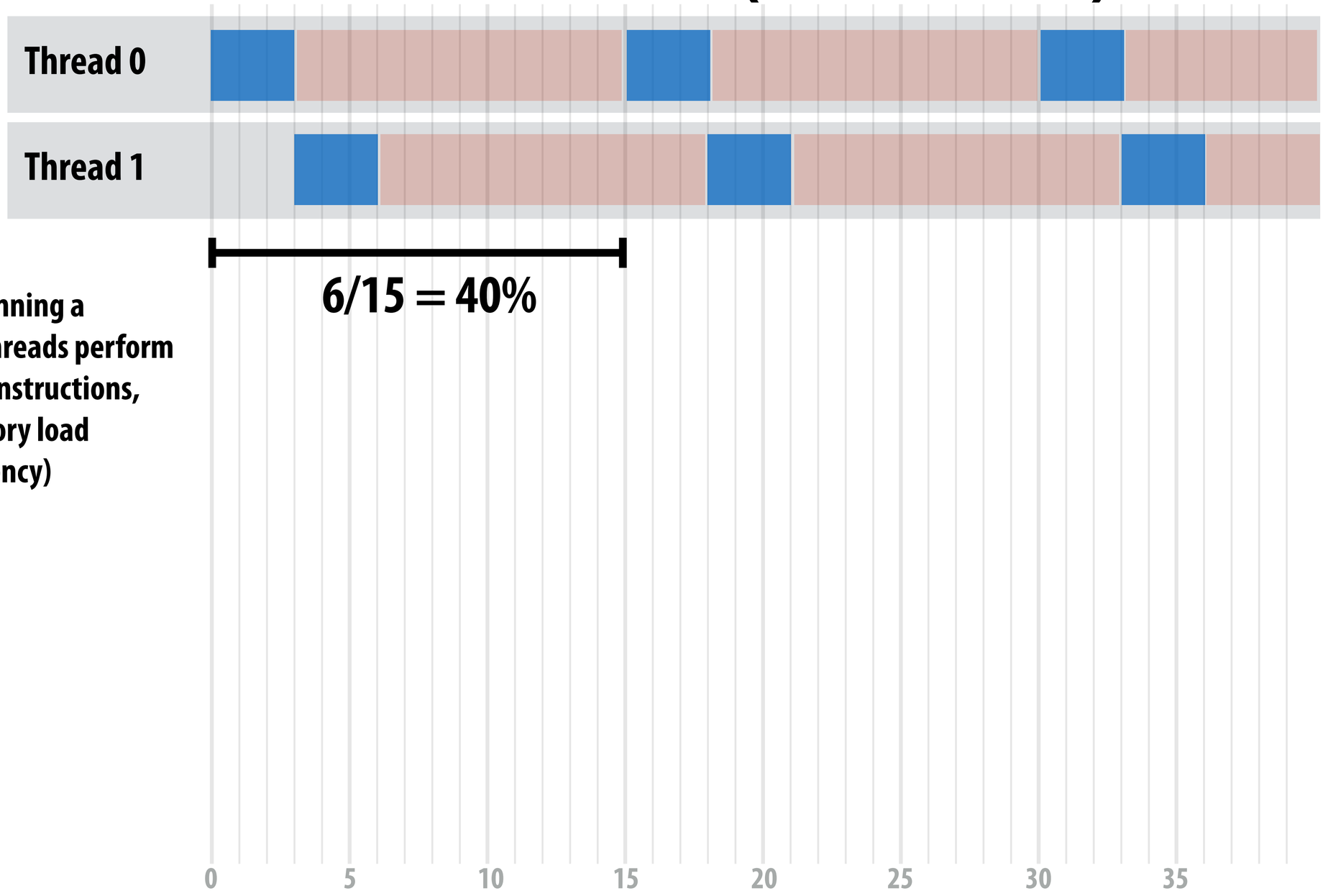
Single core processor, multi-threaded core (2 threads).
Can run one scalar instruction per clock from
one of the hardware threads

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



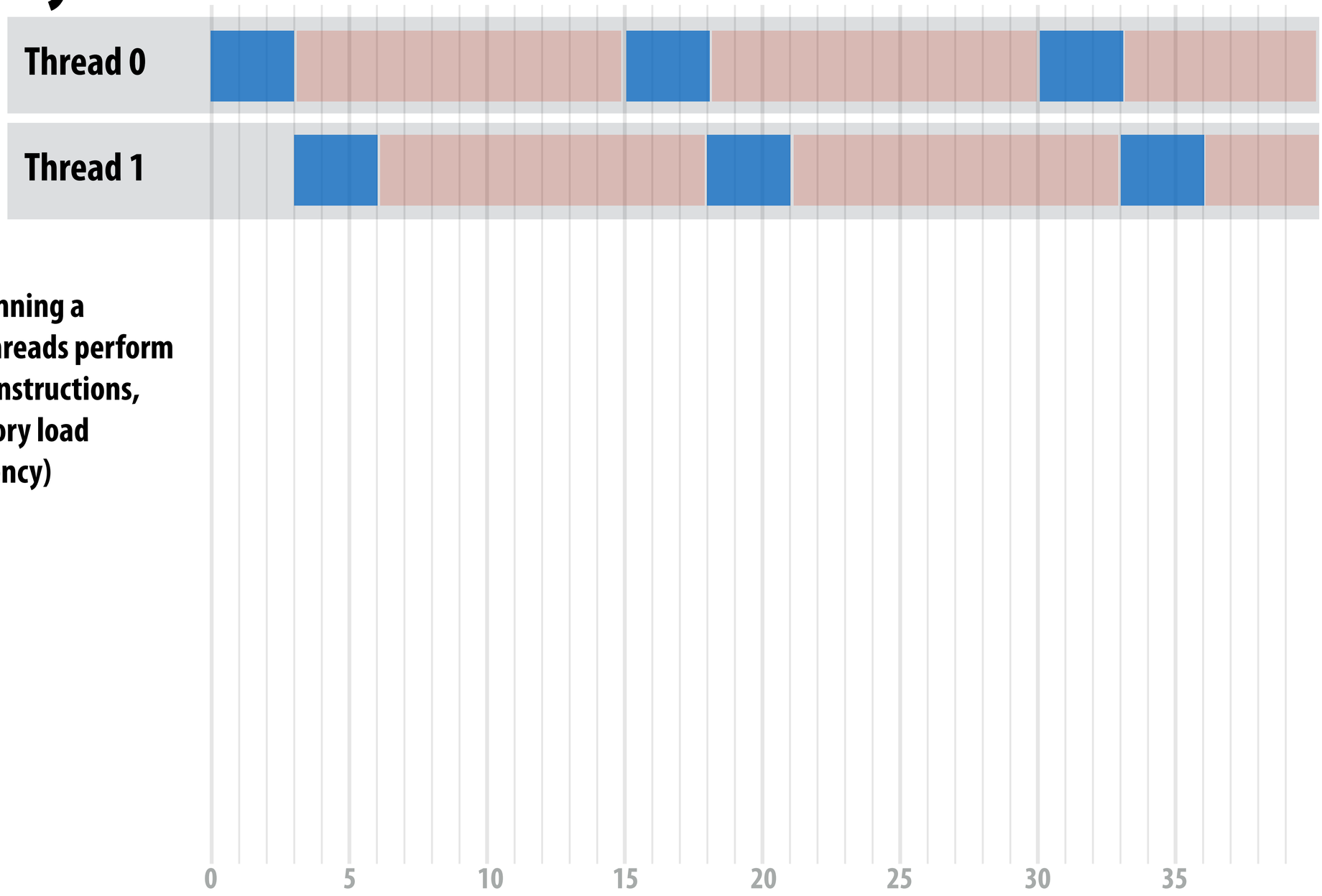
Assume we are running a program where threads perform three arithmetic instructions, followed by memory load (with 12 cycle latency)

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



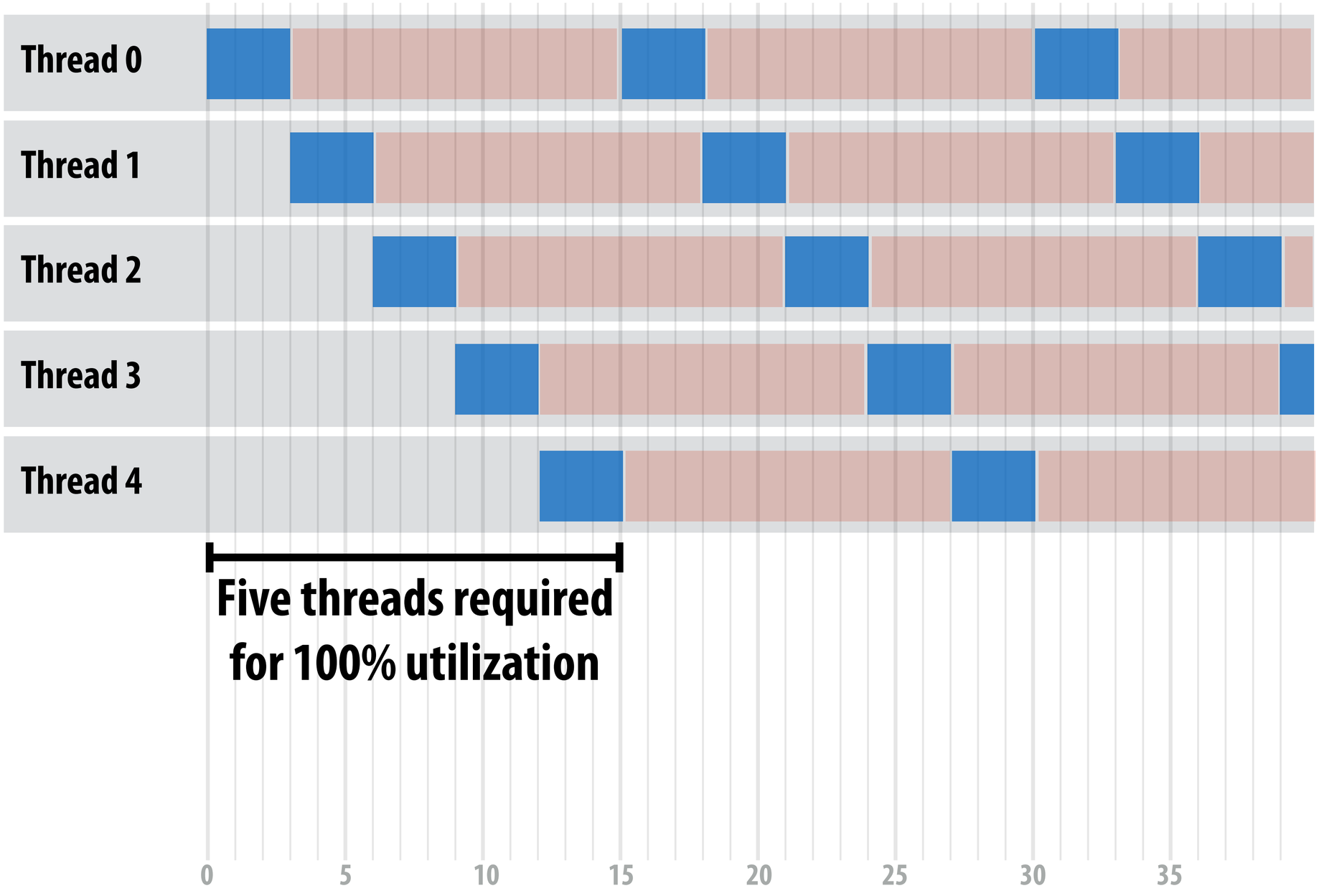
Assume we are running a program where threads perform three arithmetic instructions, followed by memory load (with 12 cycle latency)

How many threads are needed to achieve 100% utilization?

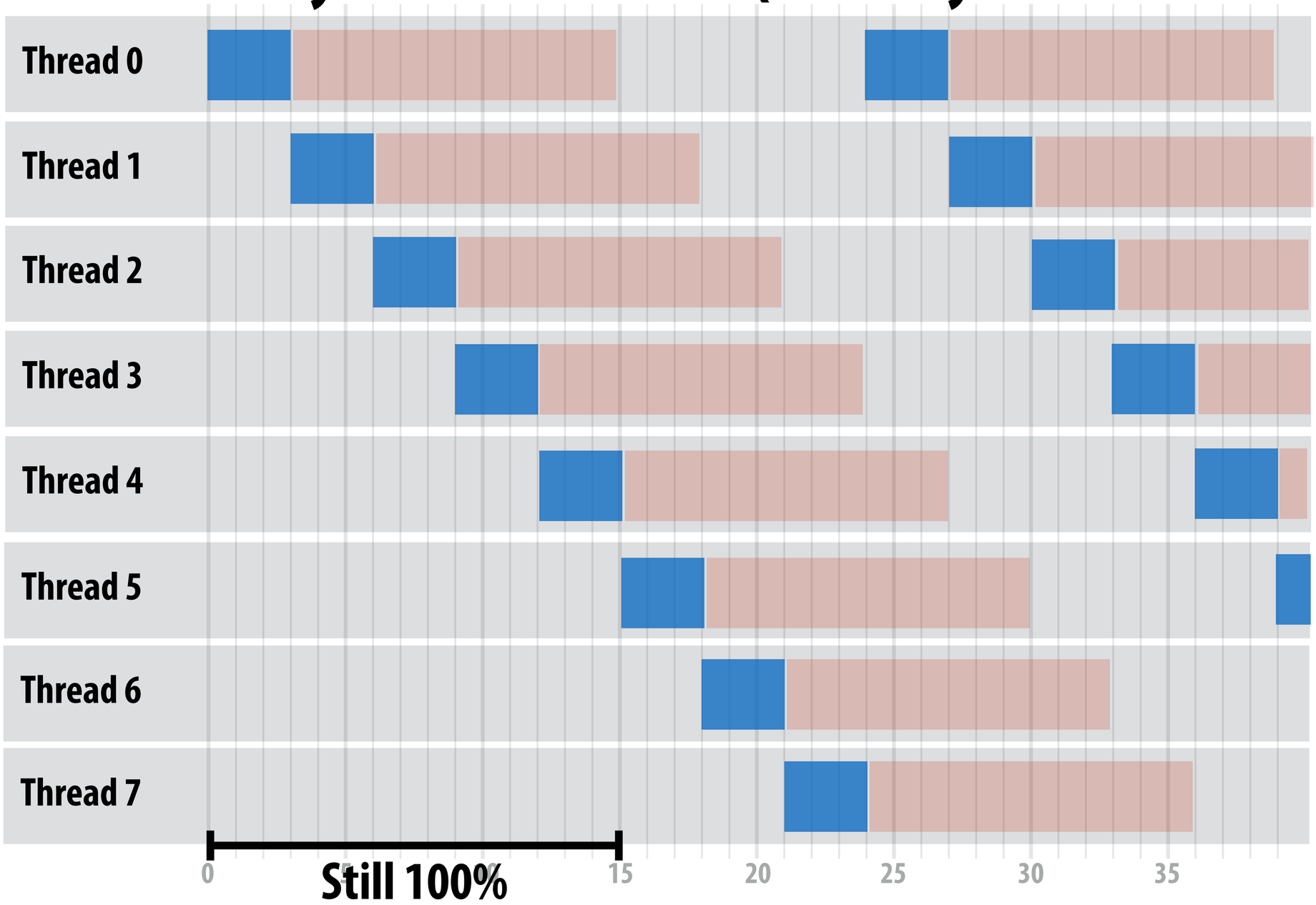


Assume we are running a program where threads perform three arithmetic instructions, followed by memory load (with 12 cycle latency)

Five threads needed to obtain 100% utilization

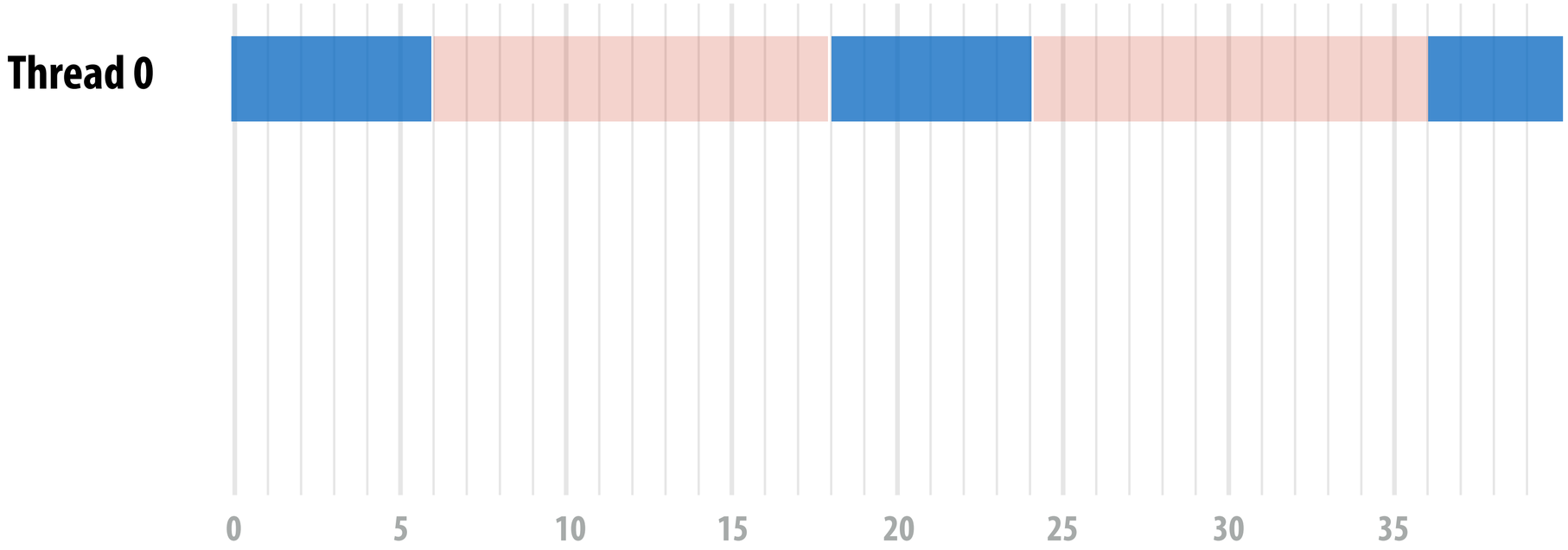


Additional threads yield no benefit (already 100% utilization)



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.

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

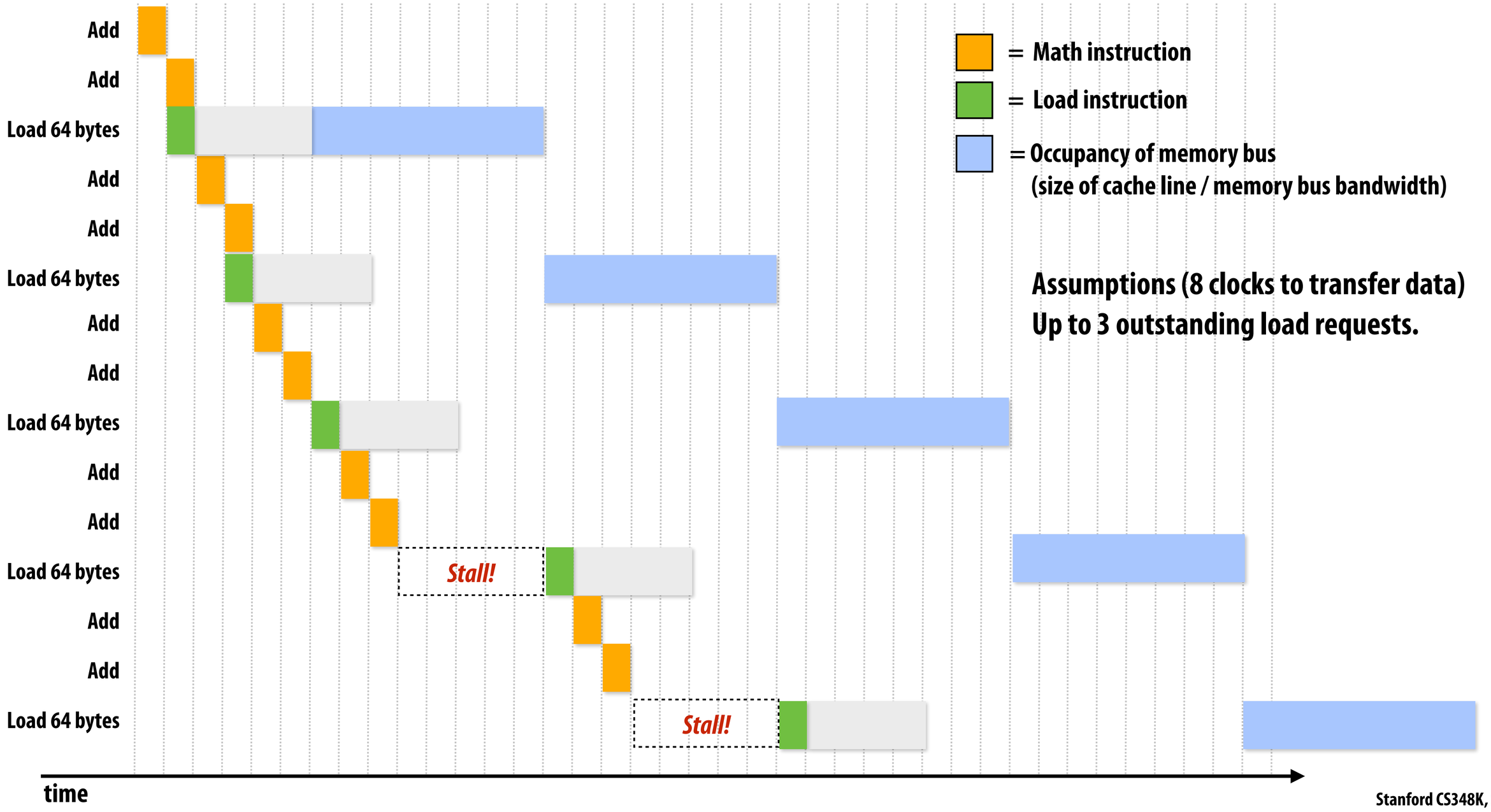
Terminology

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

Bandwidth: ~ 8 items/sec

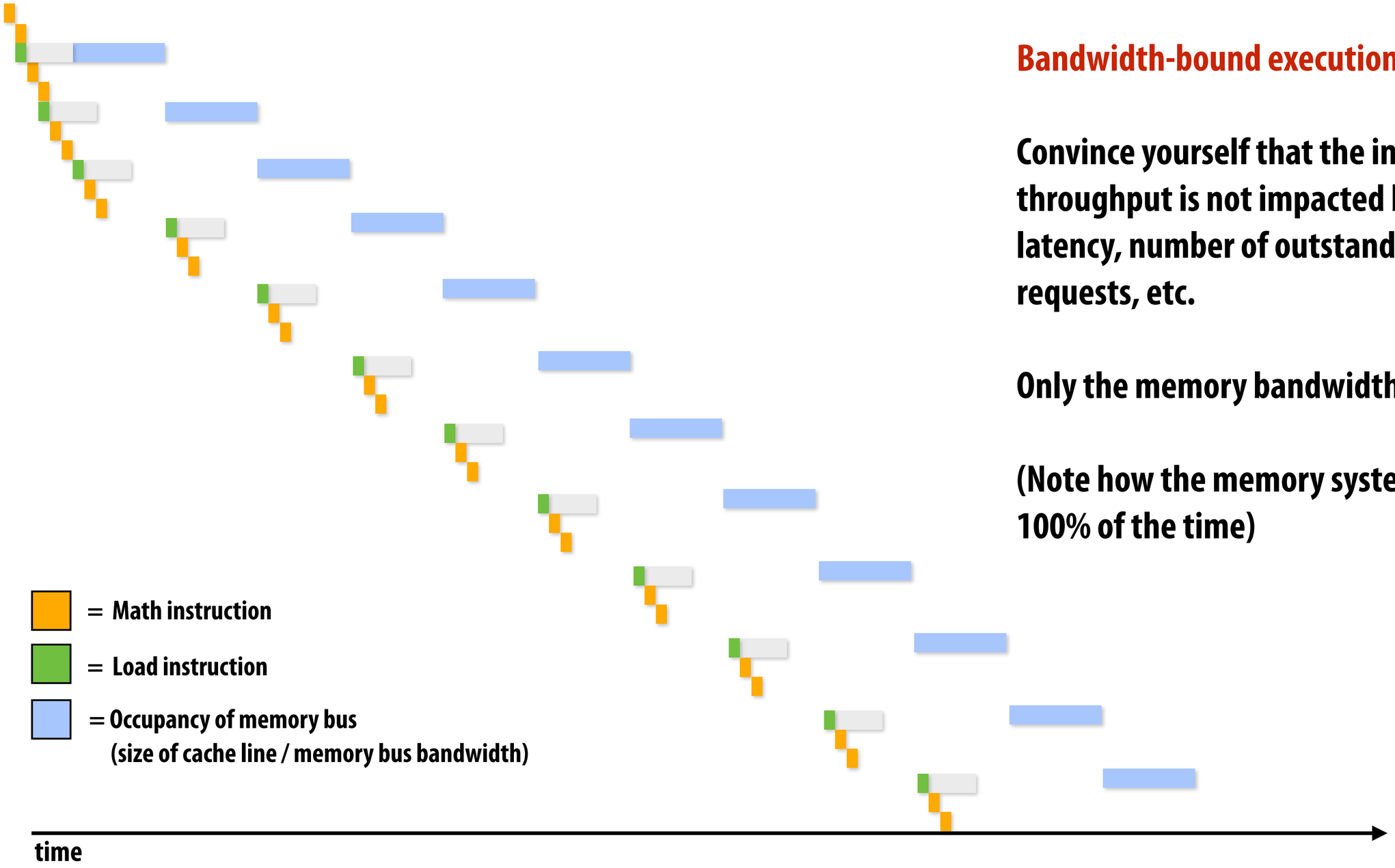
Latency of transferring any one item: ~2 sec

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



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

Rate of math instructions limited by available bandwidth



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!!!

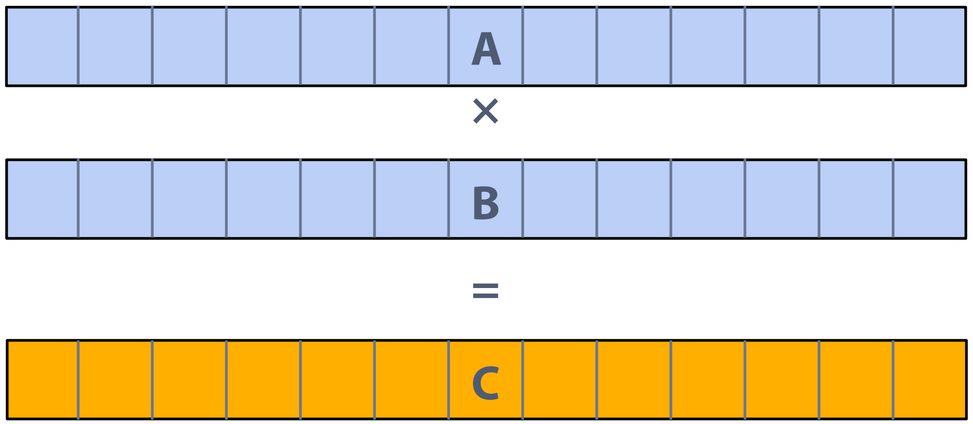
(Note how the memory system is occupied 100% of the time)

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)

Bandwidth limited!

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.

Data movement has high energy cost

- Rule of thumb in mobile system design: always seek to reduce amount of data transferred from memory
  - Earlier in class we discussed minimizing communication to reduce stalls (poor performance). Now, we wish to reduce communication to reduce energy consumption

- "Ballpark" numbers
  - Integer op: ~ 1 pJ *
  - Floating point op: ~20 pJ *
  - Reading 64 bits from small local SRAM (1mm away on chip): ~ 26 pJ
  - Reading 64 bits from low power mobile DRAM (LPDDR): ~1200 pJ

[Sources: Bill Dally (NVIDIA), Tom Olson (ARM)]

Suggests that recomputing values, rather than storing and reloading them, is a better answer when optimizing code for energy efficiency!

- Implications
  - Reading 10 GB/sec from memory: ~1.6 watts
  - Entire power budget for mobile GPU: ~1 watt
    (remember phone is also running CPU, display, radios, etc.)
  - iPhone 6 battery: ~7 watt-hours (note: my Macbook Pro laptop: 99 watt-hour battery)
  - Exploiting locality matters!!!

* Cost to just perform the logical operation, not counting overhead of instruction decode, load data from registers, etc.

In modern computing, bandwidth is the critical resource

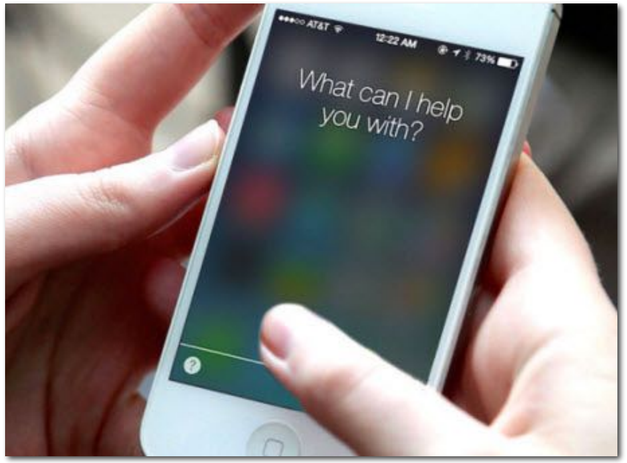
Performant parallel programs will:

- 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

Concept #2:
The value of specializing computation

Mobile: benefits of increasing efficiency

- **Run faster for a fixed period of time**
  - Run at higher clock, use more cores (reduce latency of critical task)
  - Do more at once
- **Run at a fixed level of performance for longer**
  - e.g., video playback, health apps
  - Achieve "always-on" functionality that was previously impossible



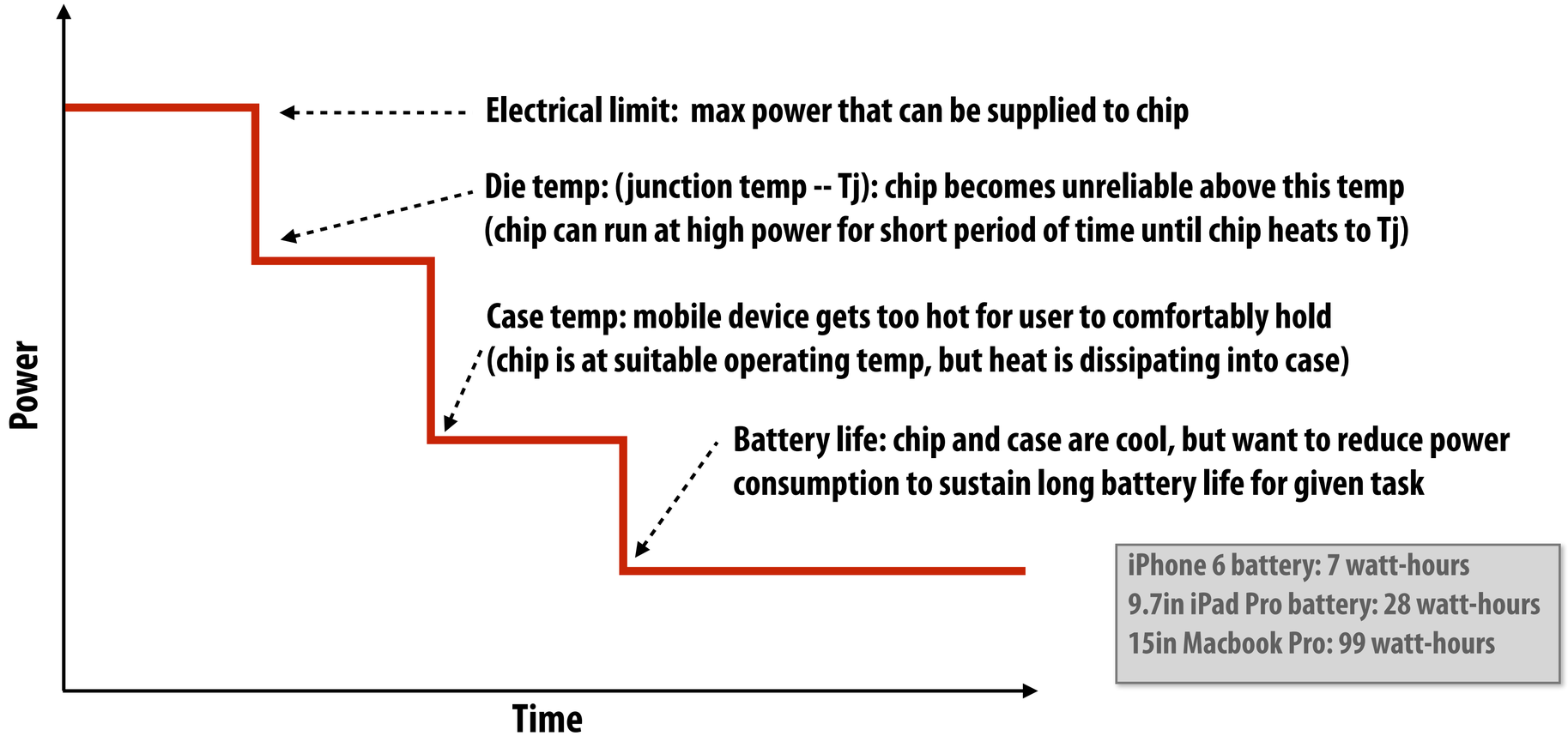
iPhone:
Siri activated by button press or holding phone up to ear

Amazon Echo / Google Home
Always listening

Google Glass: ~40 min recording per charge (nowhere near "always on")

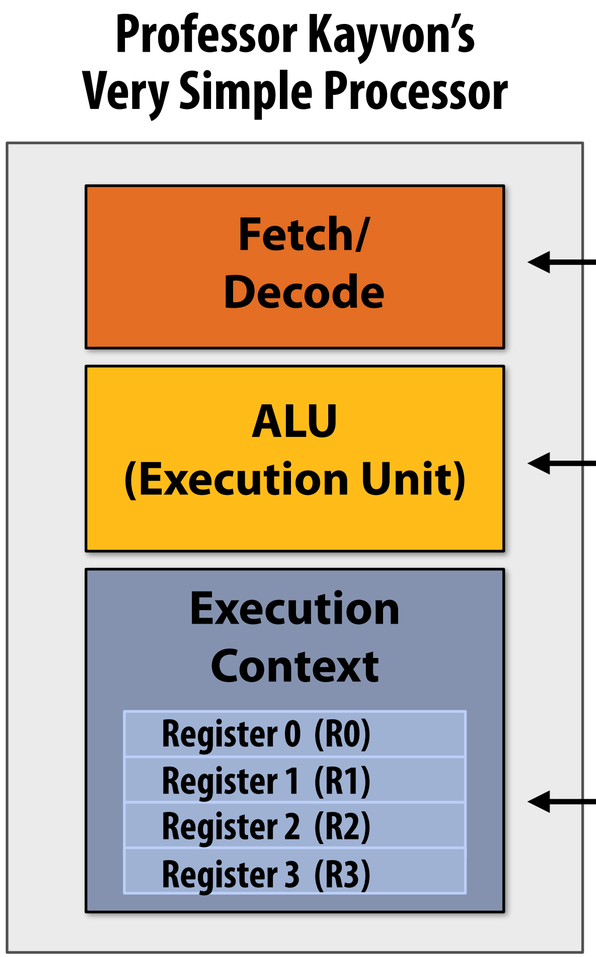
Limits on chip power consumption

- General mobile processing rule: the longer a task runs the less power it can use
  - Processor's power consumption is limited by heat generated (efficiency is required for more than just maximizing battery life)



Slide credit: adopted from original slide from M. Shebanow: HPG 2013 keynote

A basic CPU that executes instructions



Determine what instruction to run next

Execution unit: performs the operation described by an instruction, which may modify values in the processor's registers or the computer's memory

Registers: maintain program state: store value of variables used as inputs and outputs to operations

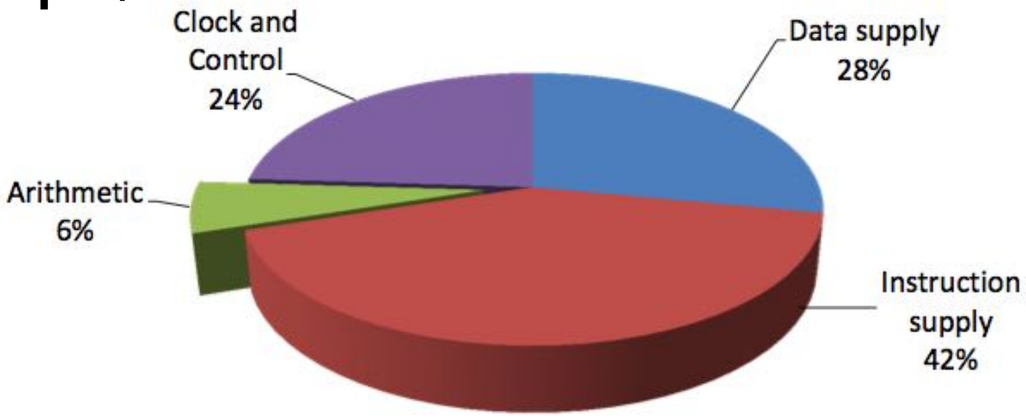
Efficiency benefits of compute specialization

- Rules of thumb: compared to high-quality C code on CPU...

- Throughput-maximized processor architectures: e.g., GPU cores
  - Approximately 10x improvement in perf / watt
  - Assuming code maps well to wide data-parallel execution and is compute bound

- Fixed-function ASIC ("application-specific integrated circuit")
  - Can approach 100-1000x or greater improvement in perf/watt
  - Assuming code is compute bound and

    and is not floating-point math

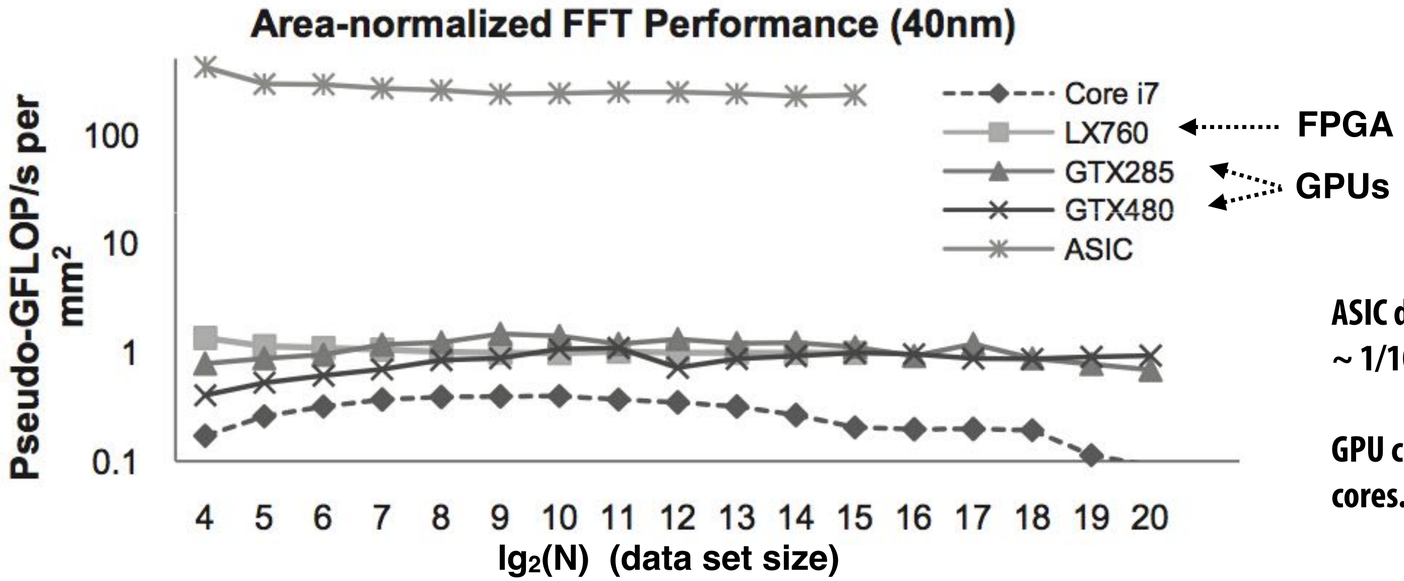


Efficient Embedded Computing [Dally et al. 08]

[Figure credit Eric Chung]

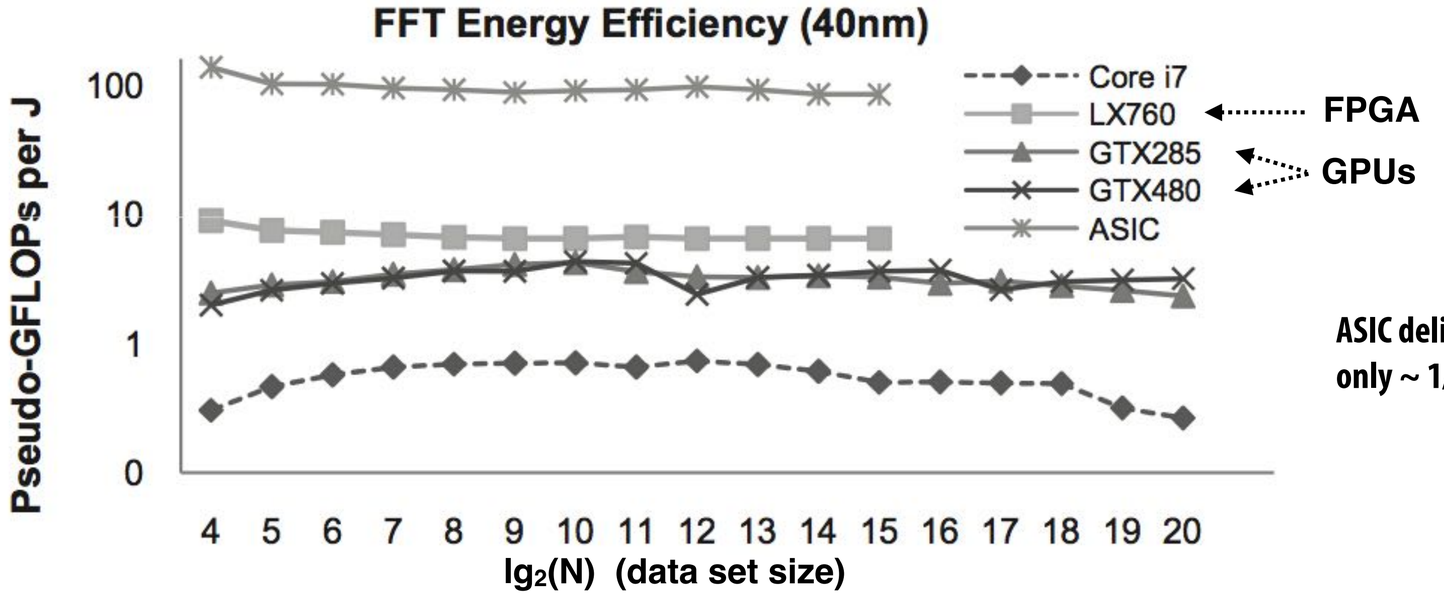
[Source: Chung et al. 2010 , Dally 08]

Hardware specialization increases efficiency



ASIC delivers same performance as one CPU core with ~ 1/1000th the chip area.

GPU cores: ~ 5-7 times more area efficient than CPU cores.



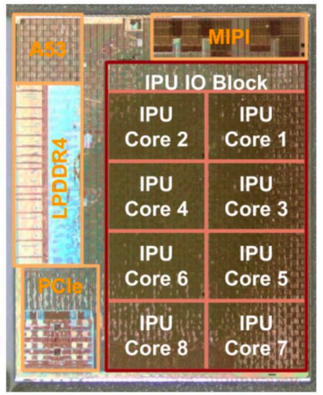
ASIC delivers same performance as one CPU core with only ~ 1/100th the power.

[Chung et al. MICRO 2010]

Let's crack open a recent smartphone

Google Pixel 2 Phone:
Qualcomm Snapdragon 835 SoC + Google Visual Pixel Core

Visual Pixel Core
Programmable image processor and DNN accelerator



"Hexagon" Programmable DSP
data-parallel multi-media processing

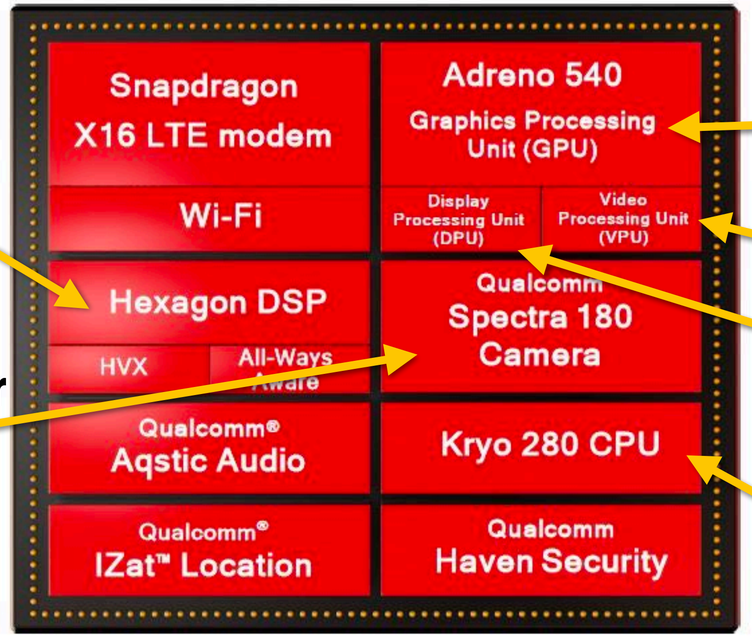


Image Signal Processor
ASIC for processing camera sensor pixels

Multi-core GPU
(3D graphics, OpenCL data-parallel compute)

Video encode/decode ASIC

Display engine
(compresses pixels for transfer to high-res screen)

Multi-core ARM CPU
4 "big cores" + 4 "little cores"

Modern smartphones utilize multiple processing units to quickly generate high-quality images

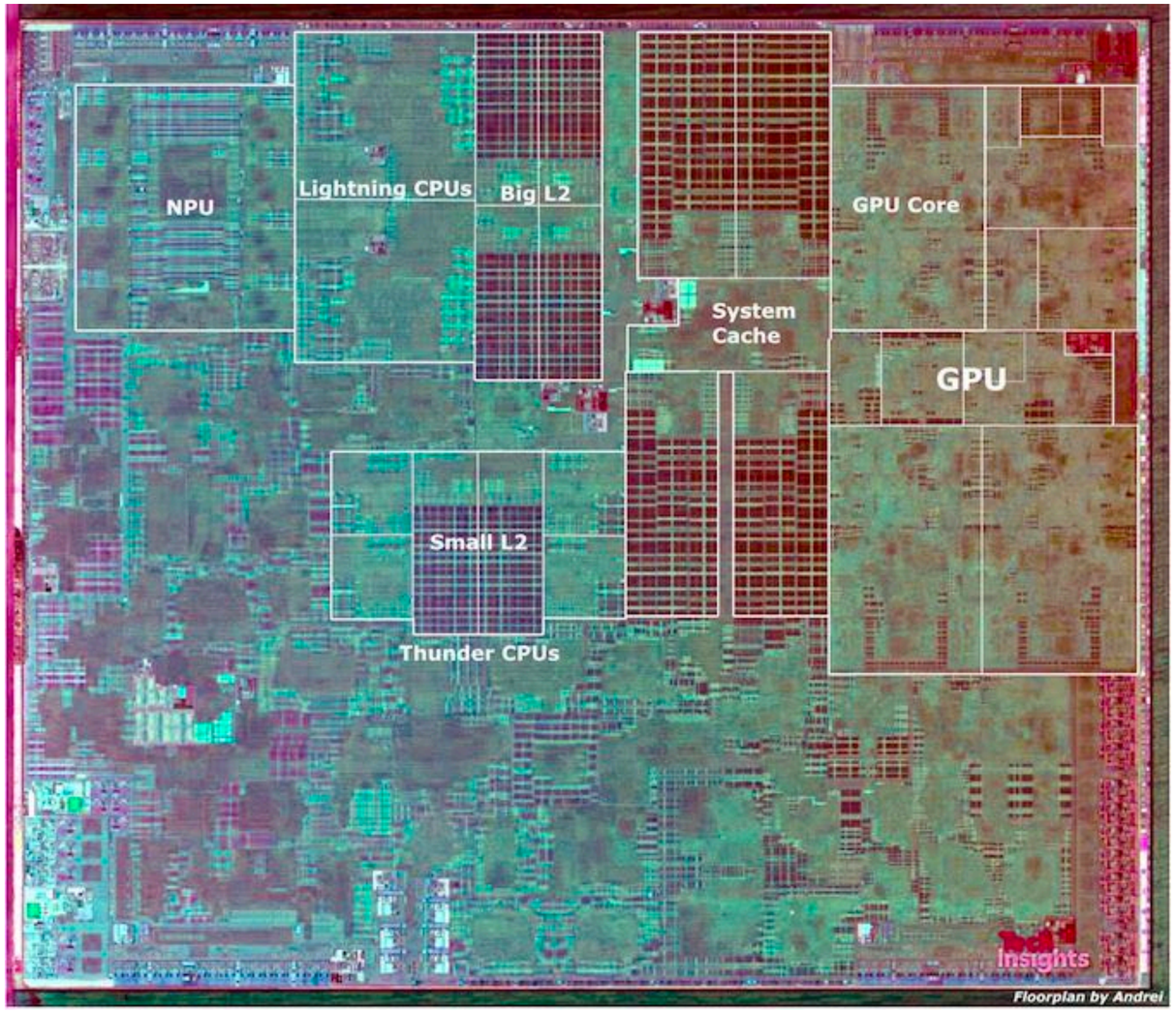


Image Credit: Anandtech / TechInsights Inc.

Apple A13 Bionic

Multi-core CPU (heterogeneous cores)
Multi-core GPU
Neural accelerator
Sensor processing accelerator
Video compression/decompression HW
Etc…

Modern systems use specialized HW for…

- Image/video encode/decode (e.g., H.264, JPG)
- Audio recording/playback
- Voice "wake up" (e.g., Ok Google)
- Camera "RAW" processing: processing data acquired by image sensor into images that are pleasing to humans
- Many 3D graphics tasks (rasterization, texture mapping, occlusion using the Z-buffer)
- Continuous sensing (health, fitness, GPS, etc)
- Deep network evaluation (Google's Tensor Processing Unit, Apple Neural engine, etc.)

Three things to know

1. What are these three hardware design strategies, and what problem/goals do they address? (See CS149 if you need a refresher)
   - Muti-core processing
   - SIMD processing
   - Hardware multi-threading

2. What is the motivation for specialization via…
   - Multiple types of processors (e.g., CPUs, GPUs)
   - Custom hardware units (ASIC)

3. Why is memory bandwidth a major constraint (often the most important constraint) when mapping applications to modern computer systems?

Welcome to CS348K!

- See website for tonight's reading