Lecture 1:

Course Introduction + Review of Throughput Hardware Concepts

Visual Computing Systems
Stanford CS348V : Winter 2018

Hi!

Me:

Your TA!

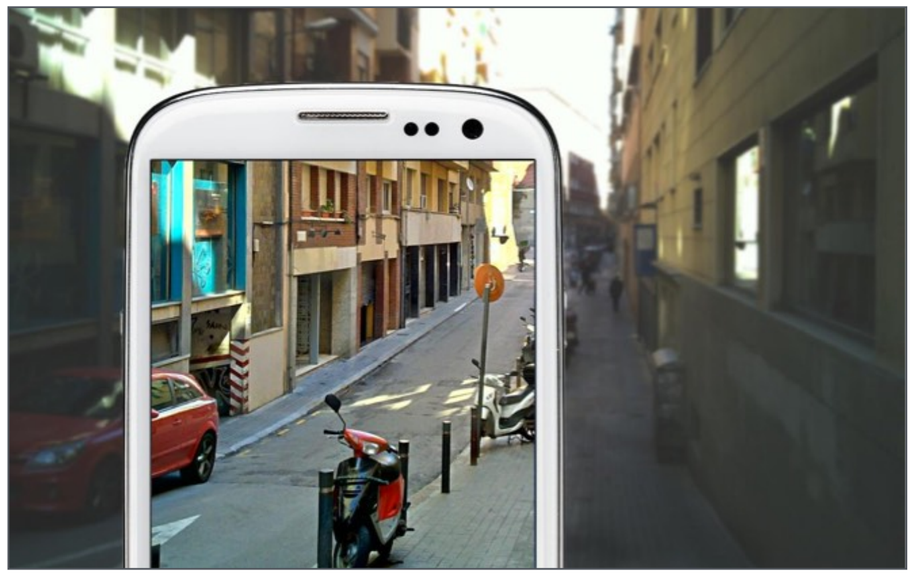
Prof. Kayvon

Raj Setaluri

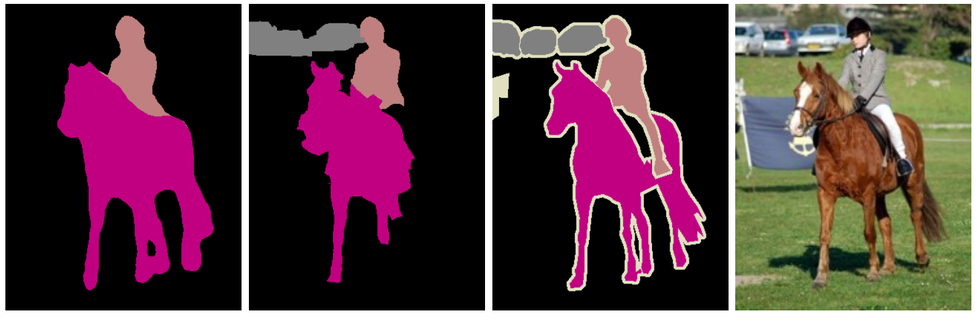
Visual computing applications

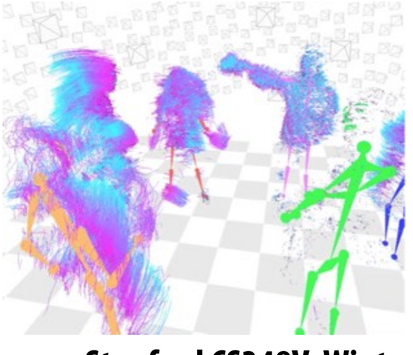
2D/3D graphics

Computational photography and image processing



Understanding the contents of images and videos





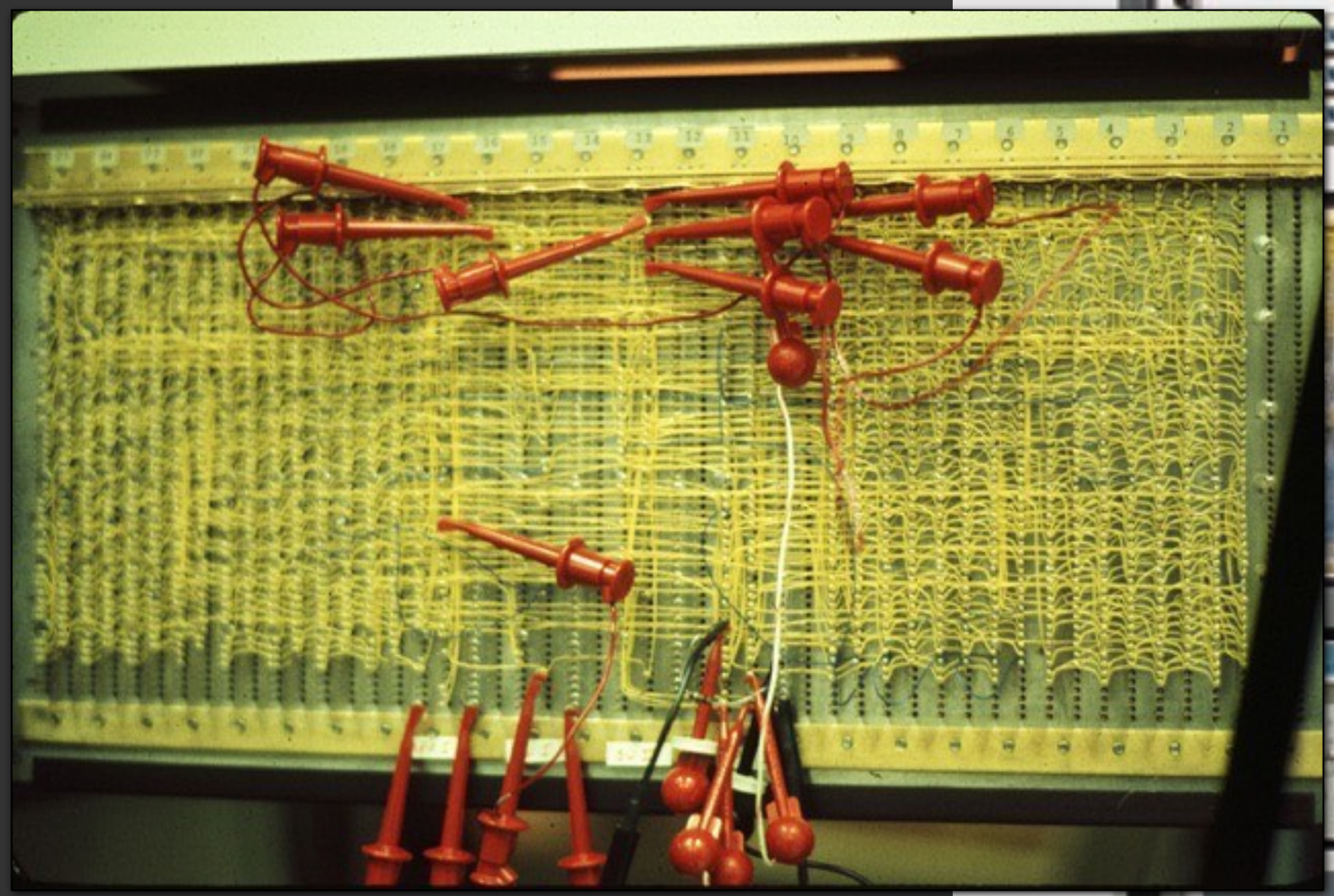
Visual Computing Systems — Some History

(why I get so excited about this topic)

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

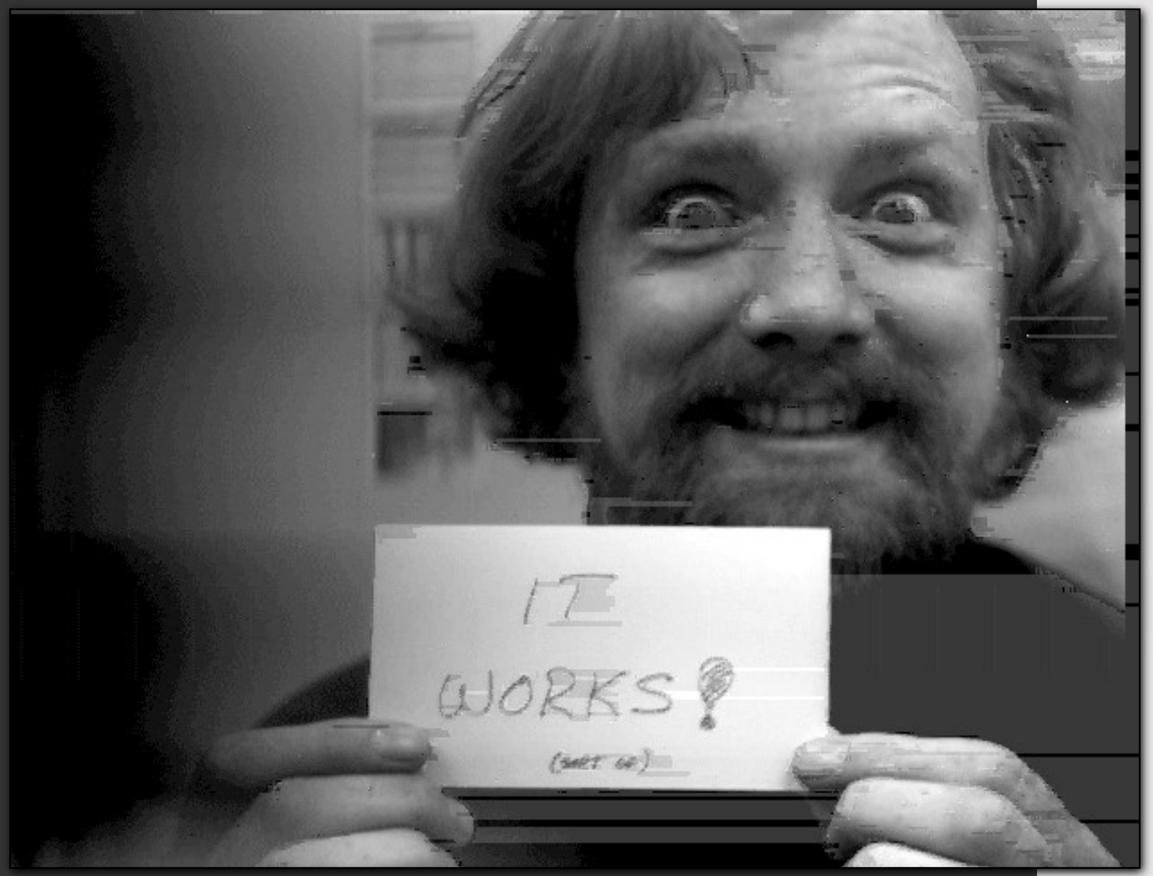
The frame buffer

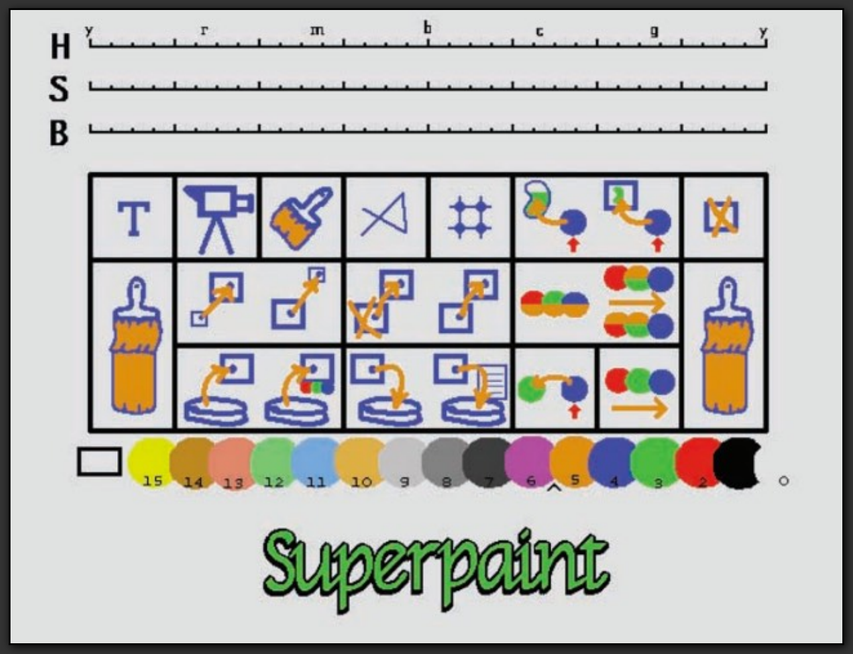
Shoup's SuperPaint (PARC 1972-73)



The frame buffer

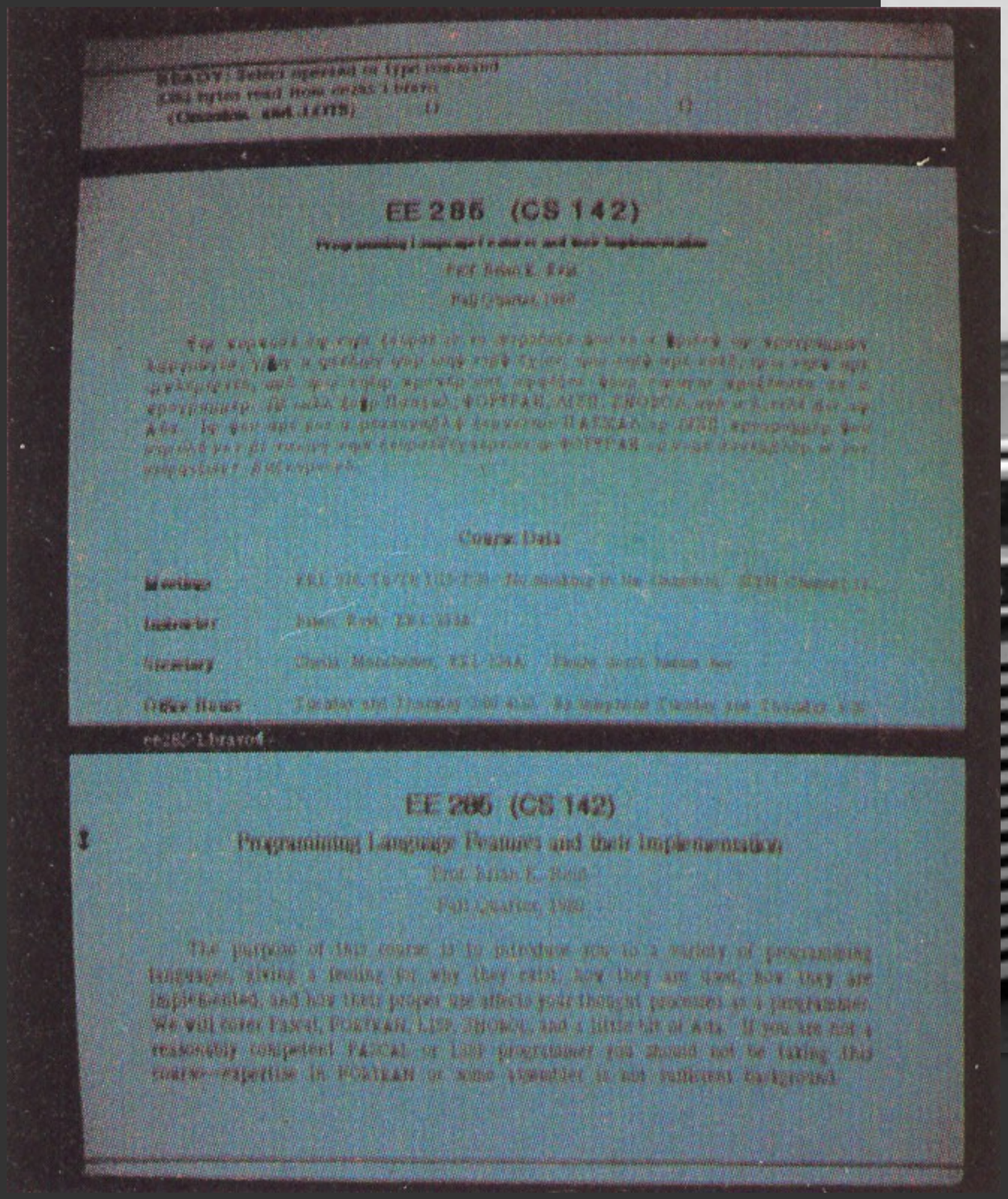
Shoup's SuperPaint (PARC 1972-73)





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

Xerox Alto (1973)

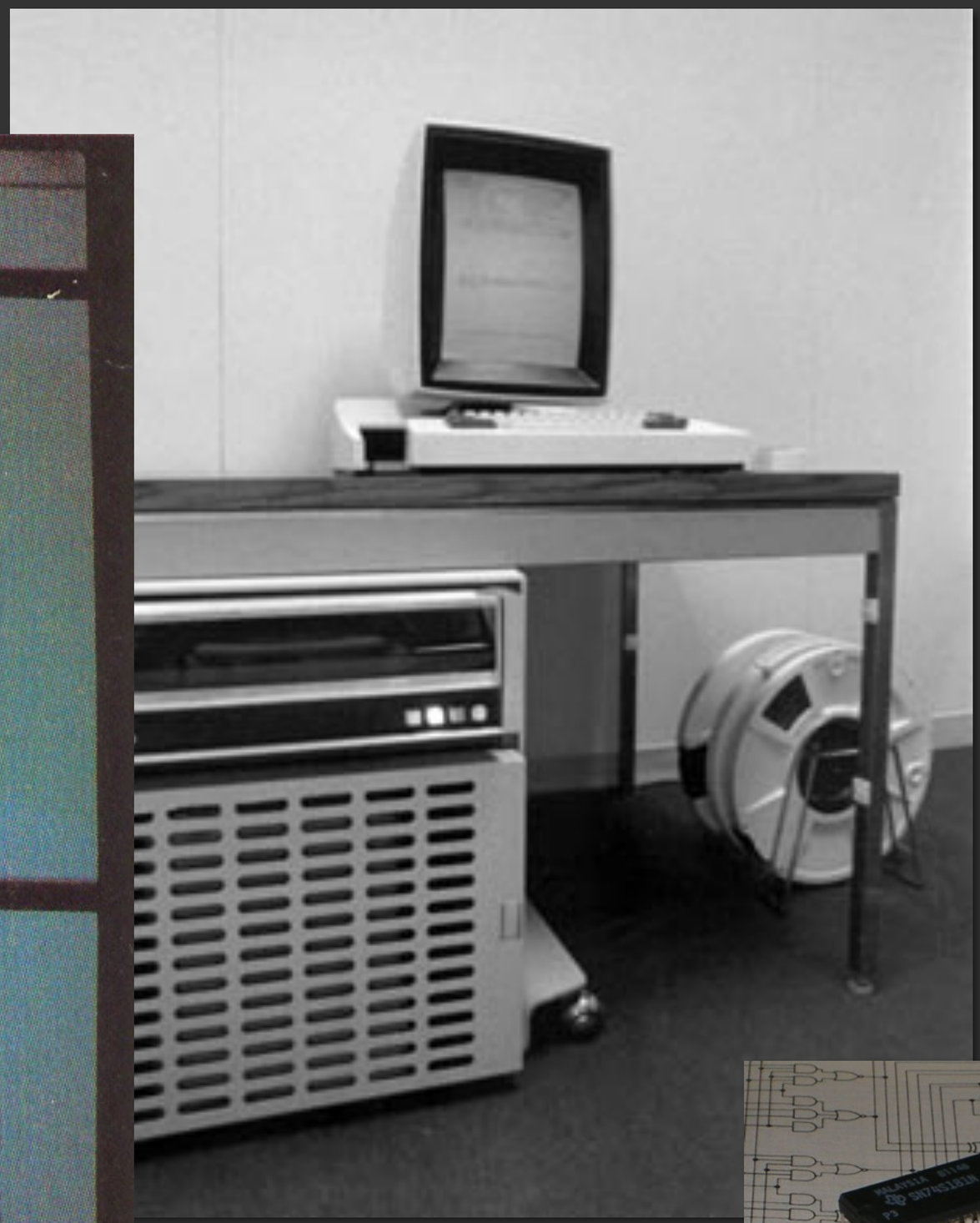

EE 286 (CS 142)

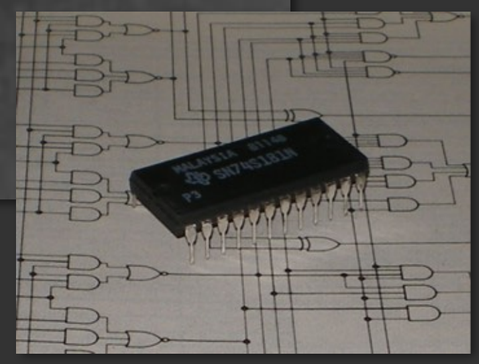
Course Data

EE 286 (CS 142)

Programming Language Features and their Implementation

Bravo (WYSIWYG)





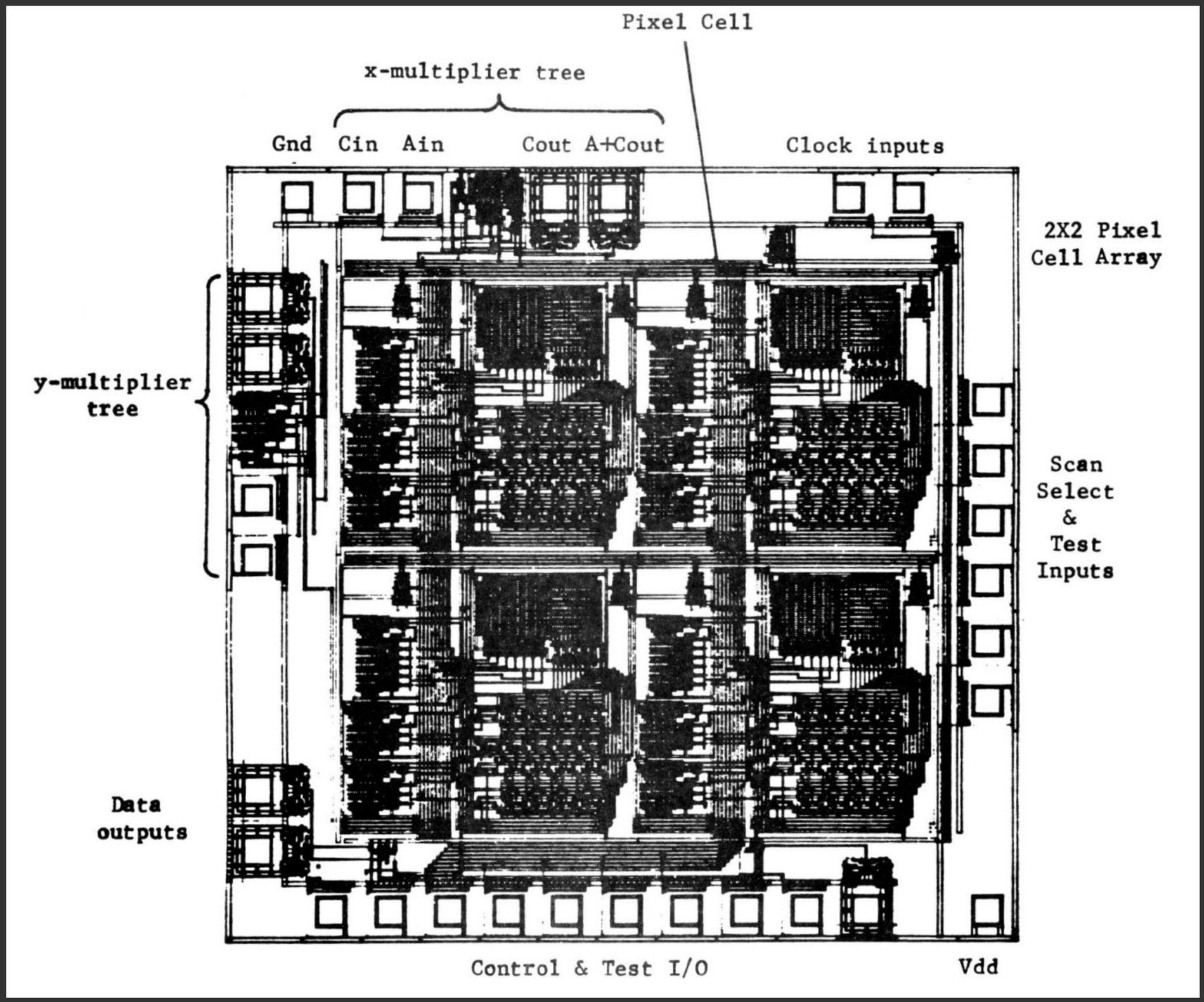
TI 74181 ALU

Goal: render everything you've ever seen

"Road to Pt. Reyes"
LucasFilm (1983)

Pixar's Toy Story (1995)

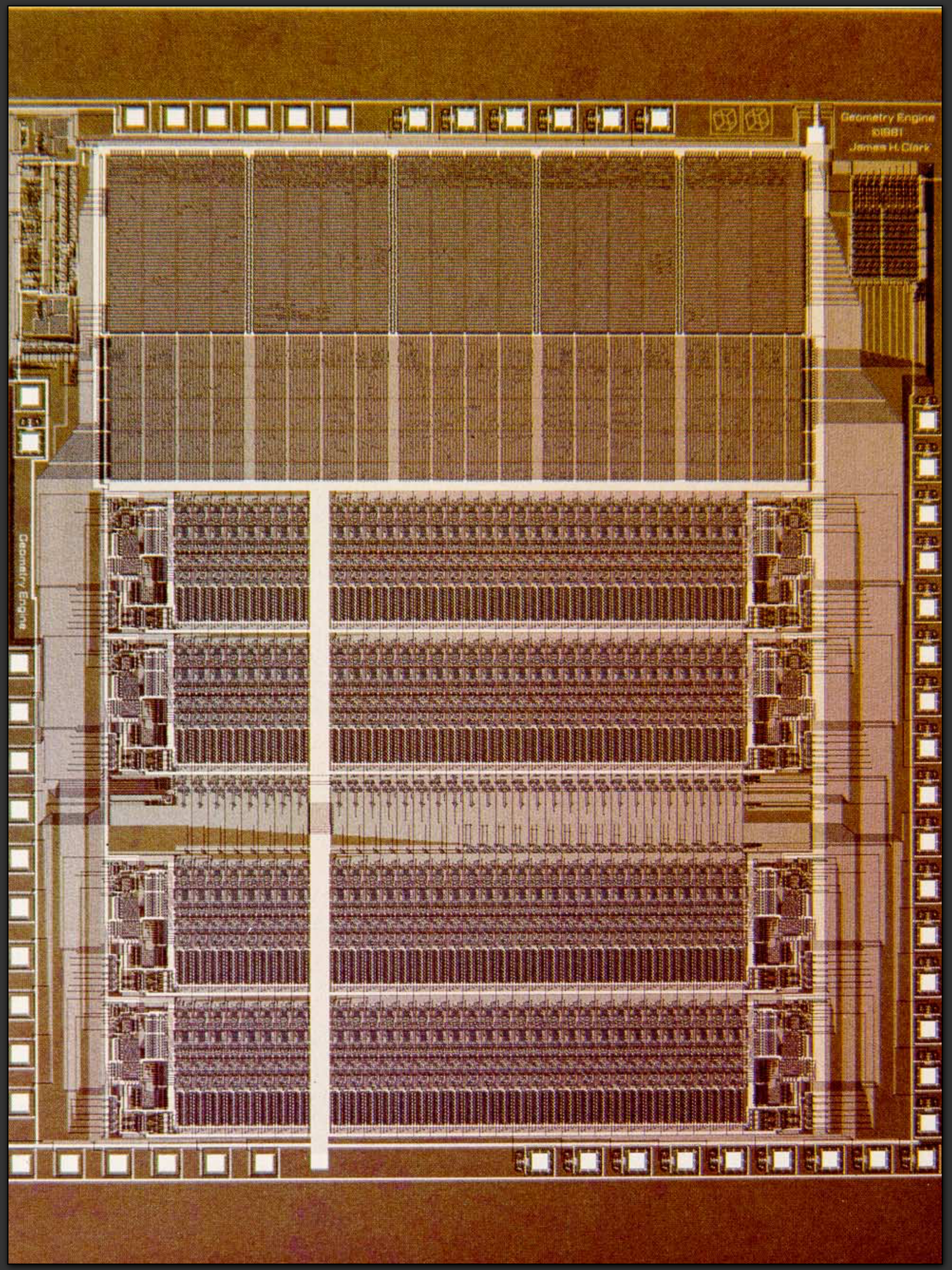
"We take an average of three hours to draw a single frame on the fastest computer money can buy."
- Steve Jobs



UNC Pixel Planes (1981), computation-enhanced frame buffer

Ed Clark's Geometry Engine (1982)

ASIC for geometric transforms used in real-time graphics.



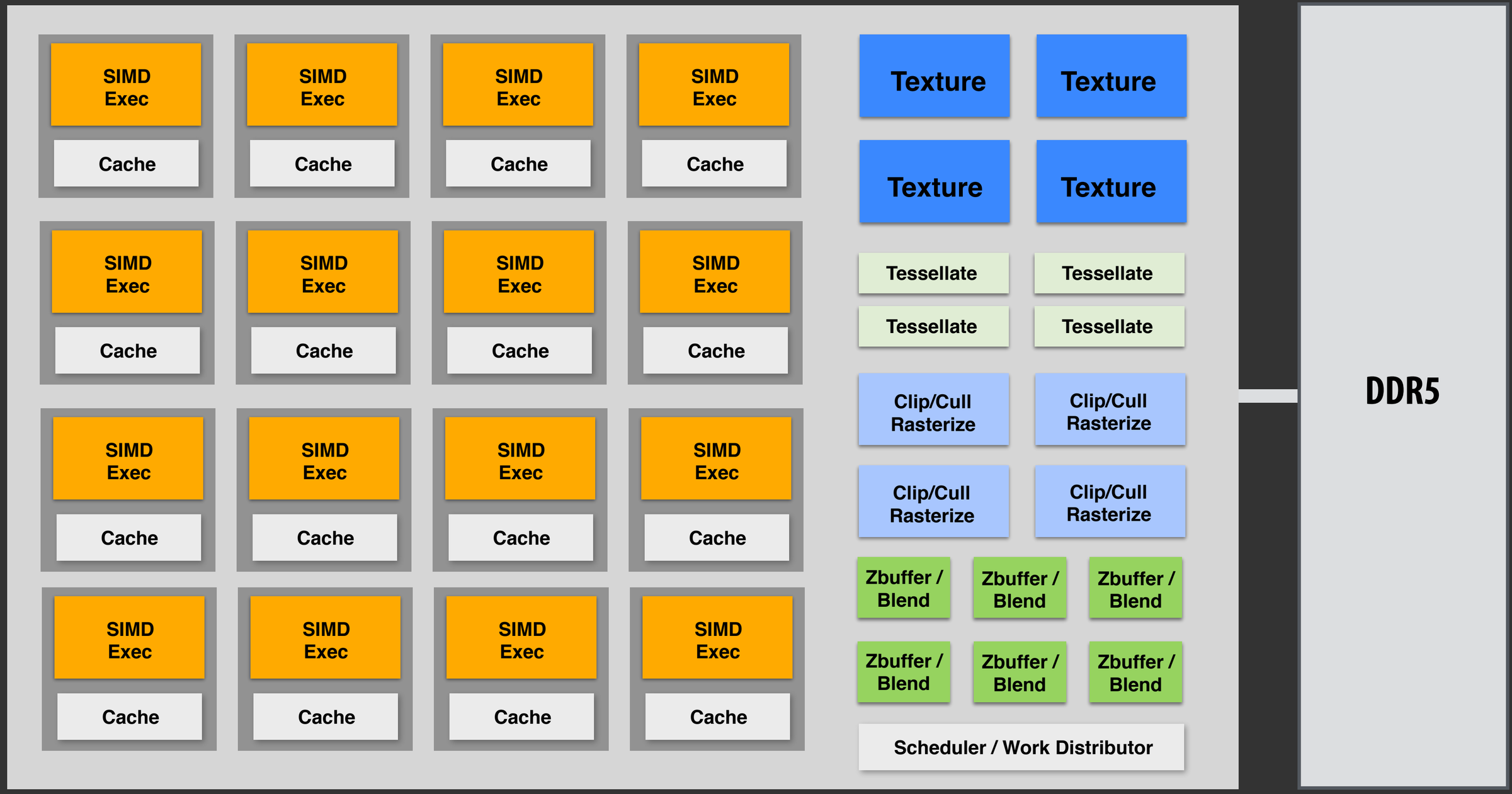
Real-time (30 fps) on a NVIDIA Titan X

Unreal Engine Kite Demo (Epic Games 2015)

NVIDIA Titan X Pascal GPU (2017)
(~ 12 TFLOPs fp32)

~ ASCI Q (top US supercomputer circa 2002)

Modern GPU: heterogeneous multi-core



Multi-threaded, SIMD cores

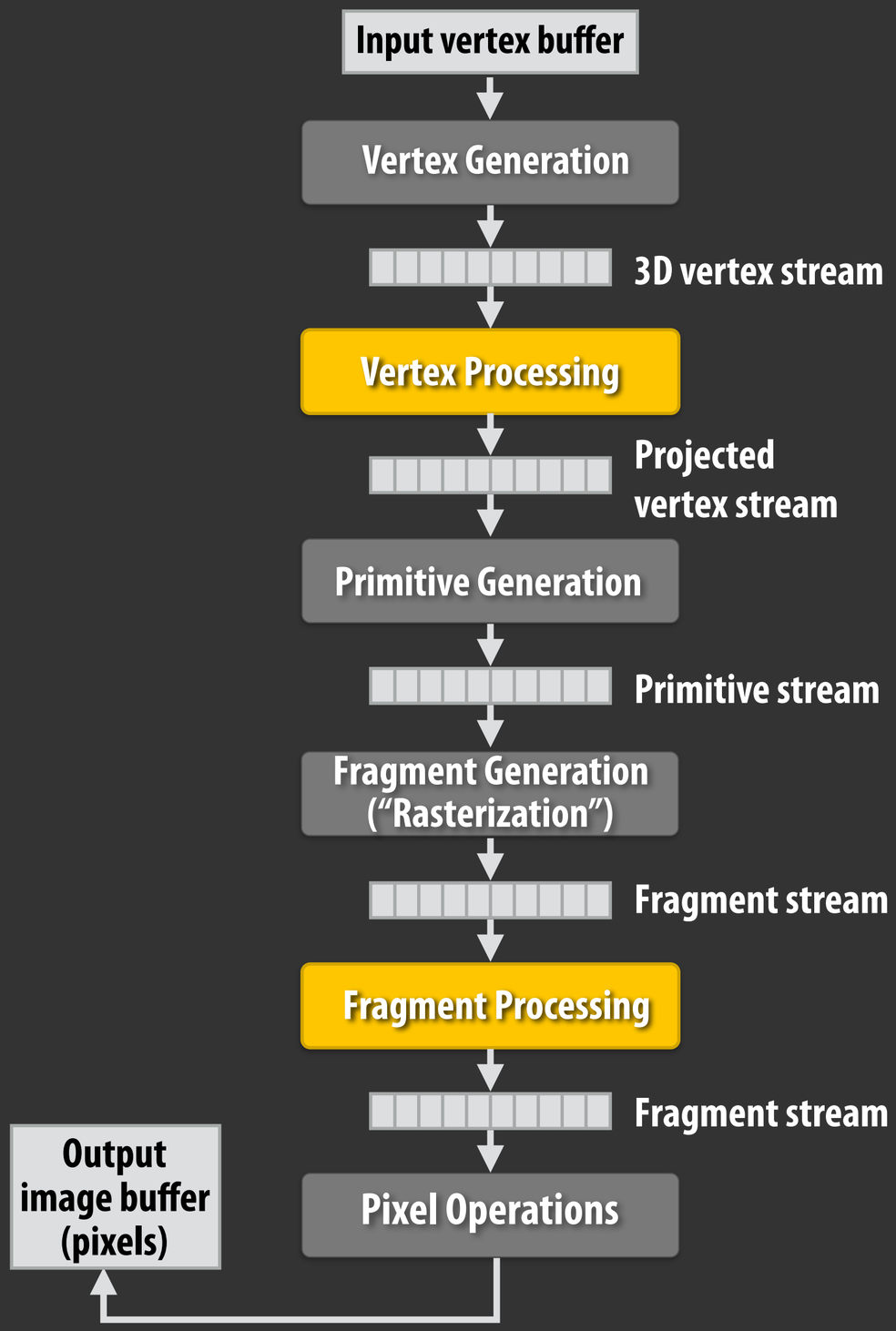
Custom circuits for key graphics arithmetic

Custom circuits for HW-assisted graphics-specific DRAM compression

HW logic for scheduling work onto these resources

Domain-specific languages for heterogeneous computing

OpenGL Graphics Pipeline (circa 2007)



The OpenGL™ Graphics System:
A Specification
(Version 1.0)

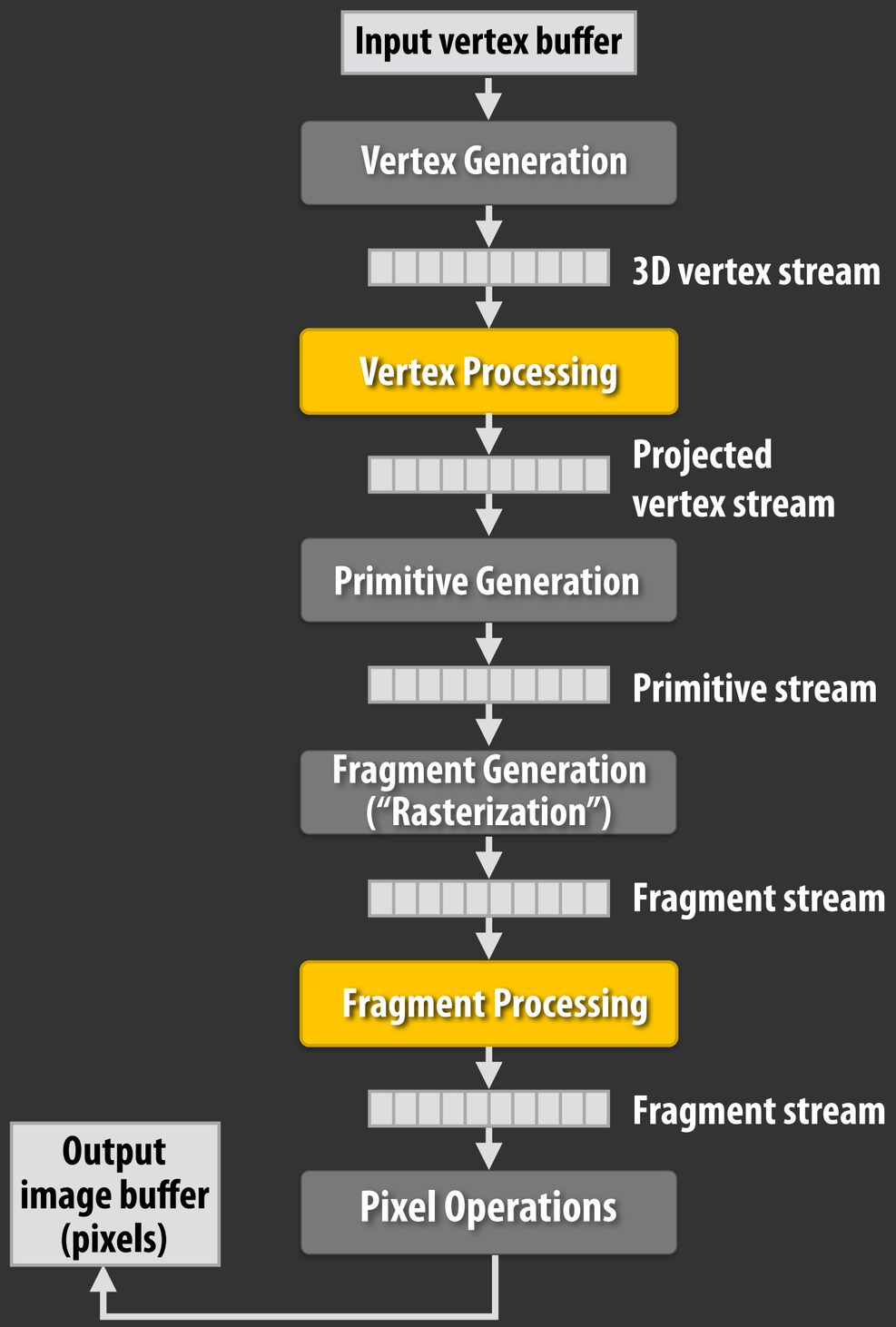
Mark Segal
Kurt Akeley

Editor:
Chris Frazier

Version 1.0 - 1 July 1994

Domain-specific languages for heterogeneous computing

OpenGL Graphics Pipeline (circa 2007)




```
uniform sampler2D myTexture;
uniform float3 lightDir;
varying vec3 norm;
varying vec2 uv;

void myFragmentShader()
{
  vec3 kd = texture2D(myTexture, uv);
  kd *= clamp(dot(lightDir, norm), 0.0,
1.0);
  return vec4(kd, 1.0);
}
```

read-only global variables

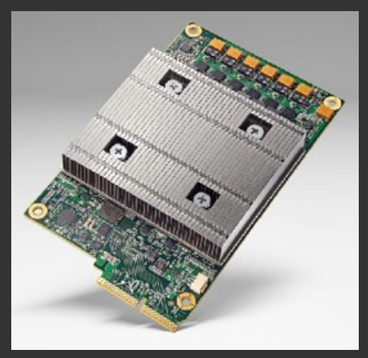
"per-element" inputs

per-element output: RGBA surface color at pixel

"fragment shader" (a.k.a kernel function mapped onto input fragment stream)

Emerging state-of-the-art visual computing systems today…

- Intelligent cameras in smartphones
- Cloud servers ("infinite" computing and storage at your disposal as a service)



- Proliferation of specialized compute accelerators
  - For image processing, machine learning
- Proliferation of high-resolution image sensors…

Capturing pixels to communicate

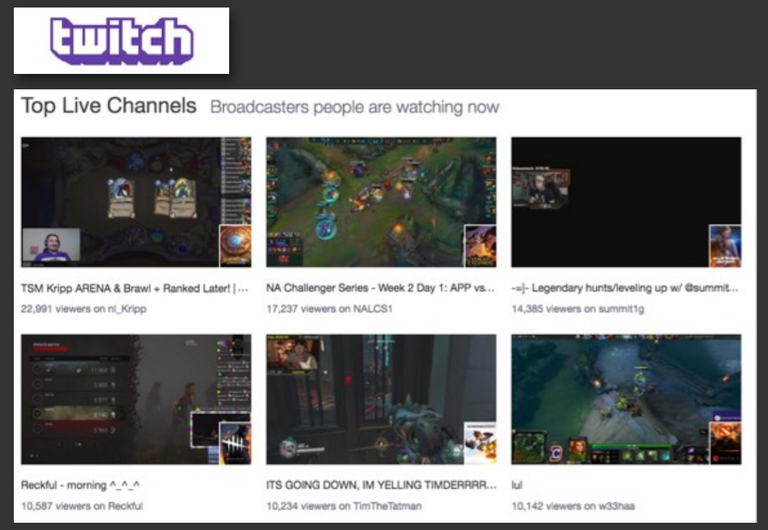
Ingesting/serving the world's photos

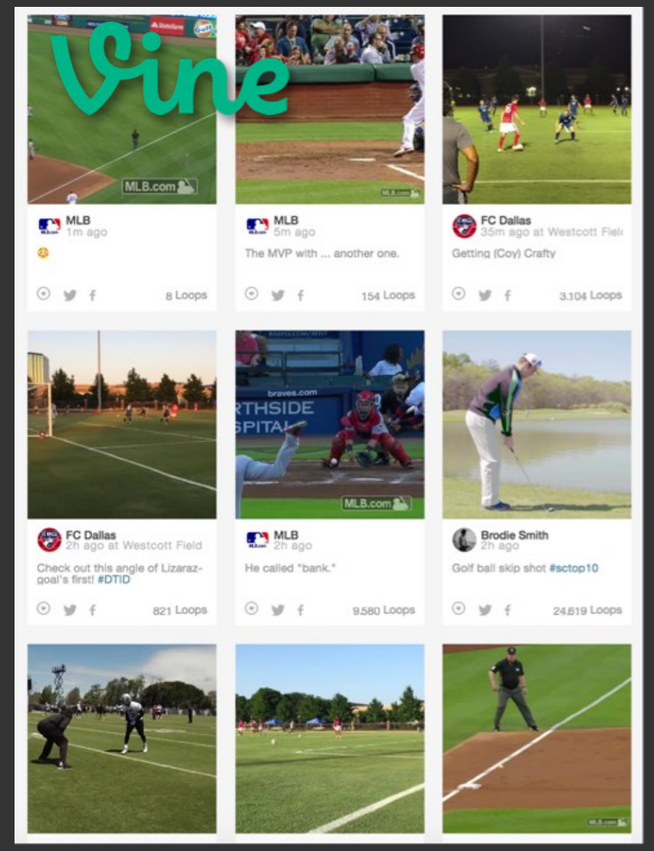
2B photo uploads and shares per day across Facebook sites (incl. Instagram+WhatsApp) [FB2015]

Ingesting/streaming world's video

Youtube 2015: 300 hours uploaded per minute [Youtube]

Cisco VNI projection: 80-90% of 2019 internet traffic will be video. (64% in 2014)

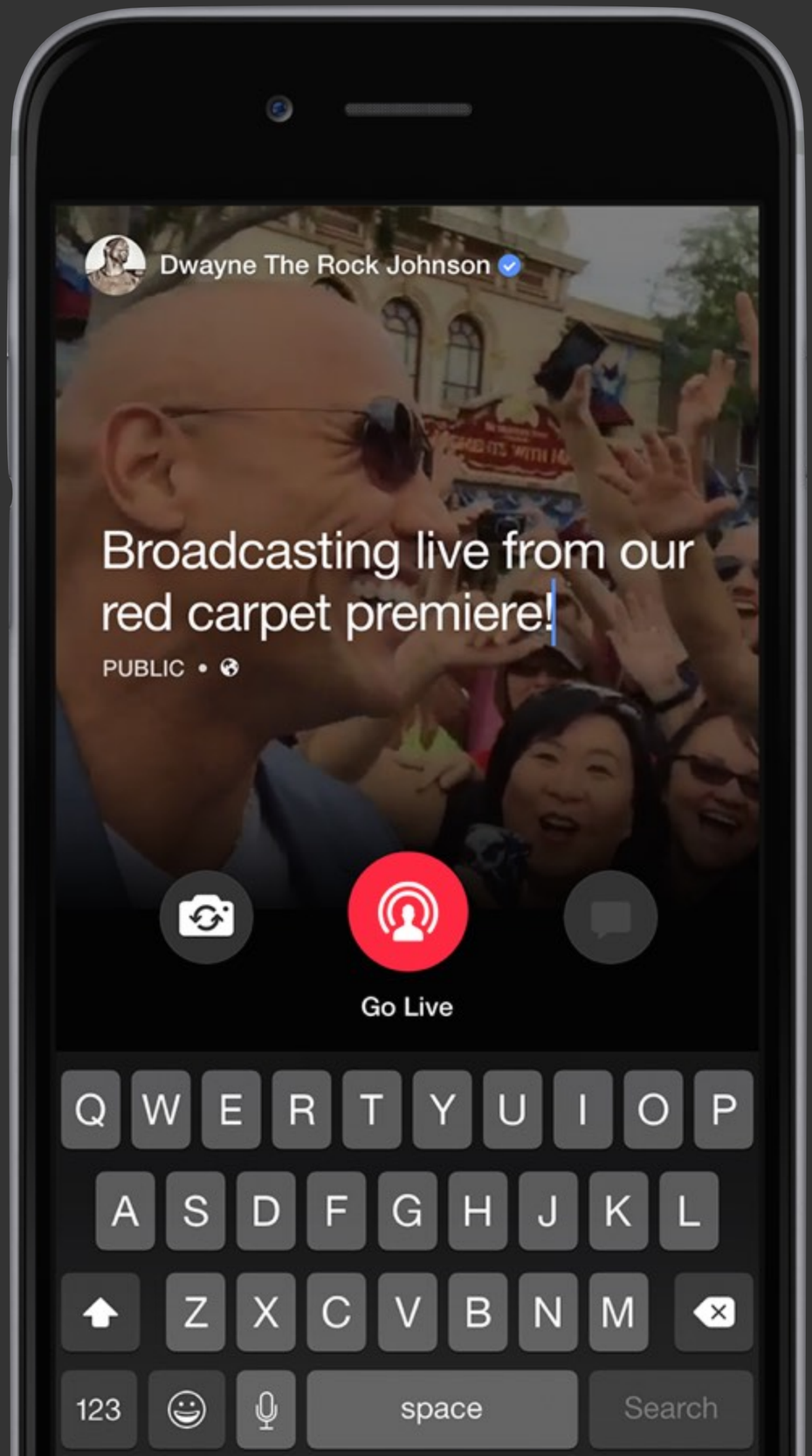




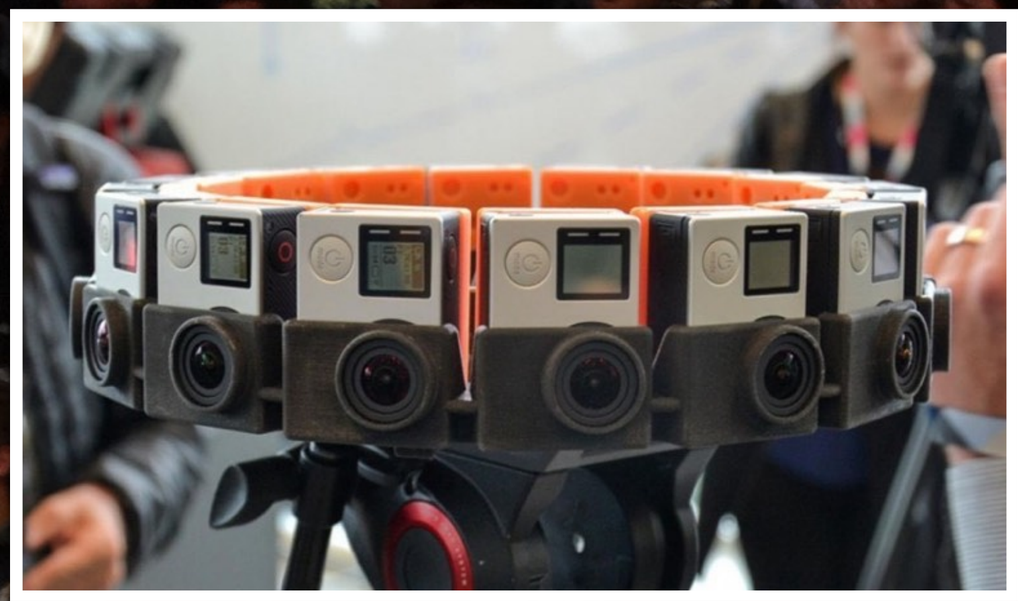
Richer content: beyond a single image

- Example: Apple's "Live Photos"
- Each photo is not only a single frame, but a few seconds of video before and after the shutter is clicked

Facebook Live



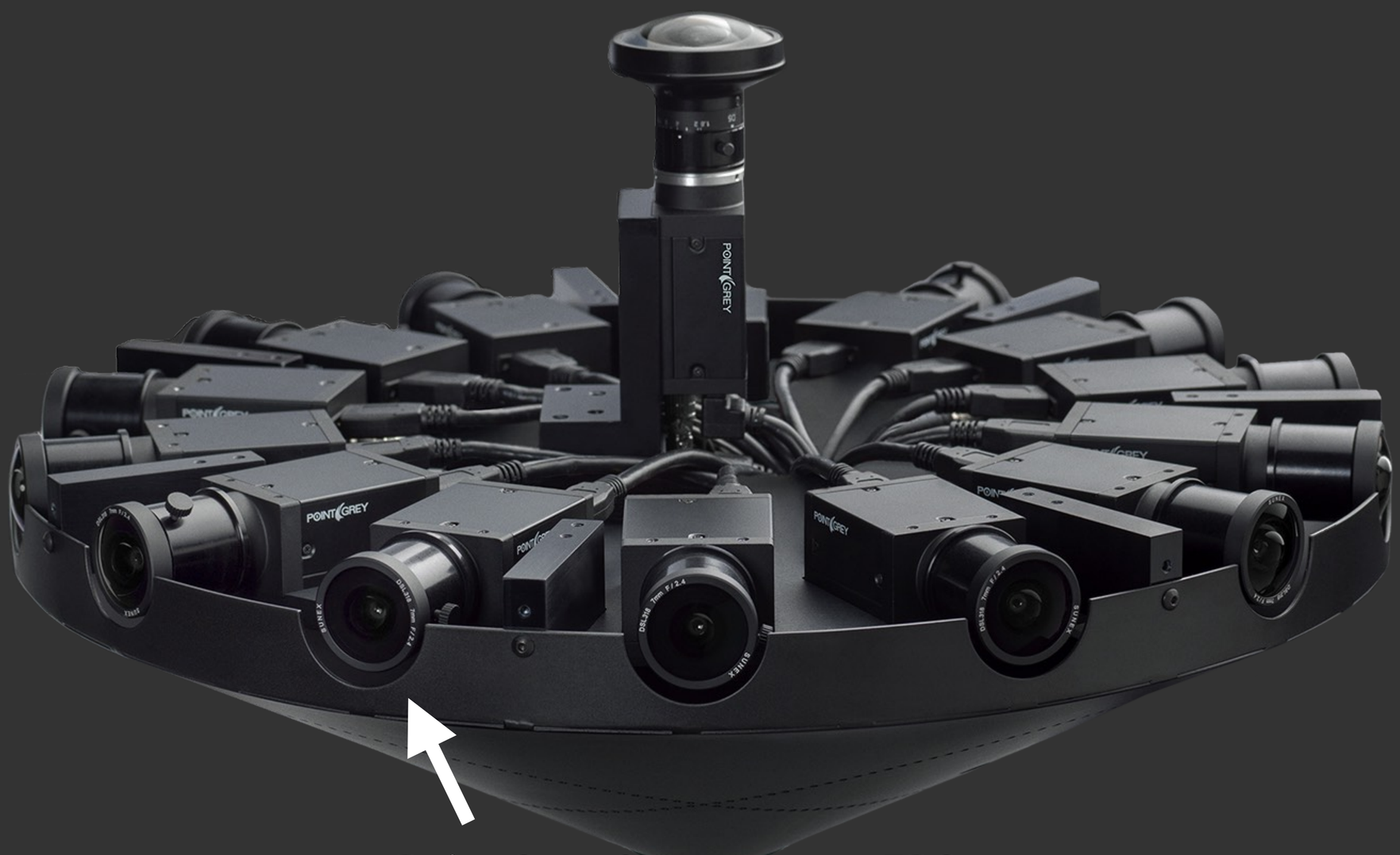
VR output



Example: Google's JumpVR video
Input stream: 16 4K GoPro cameras

Register + 3D align video stream (on edge device)
Broadcast encoded video stream across the country to millions of viewers

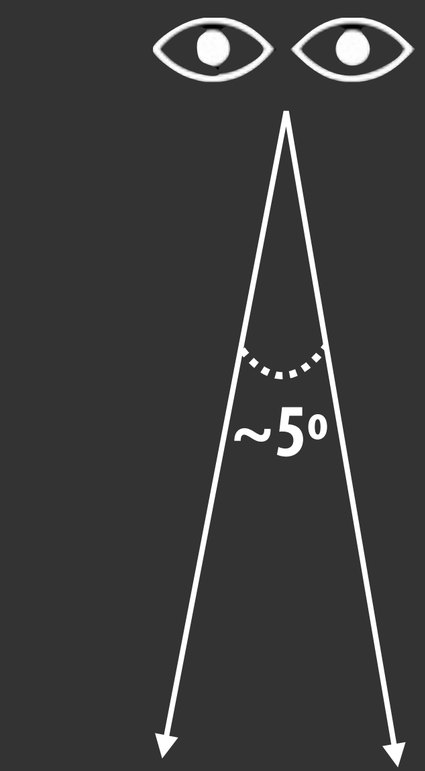
High resolution, multi-camera
Facebook Surround 360 VR video



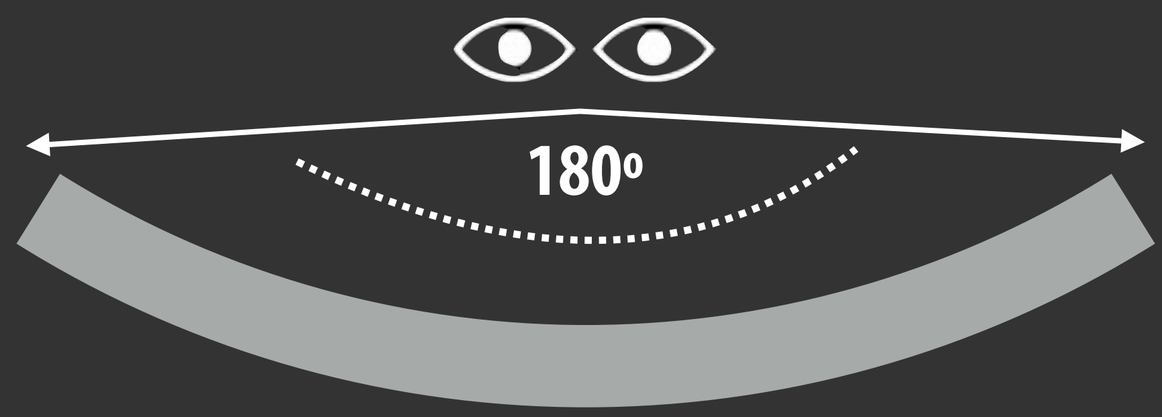
14 cameras

8K x 8K stereo panorama output

VR: high resolution requirements



iPhone 6: 4.7 in "retina" display:
1.3 MPixel
326 ppi → 57 ppd

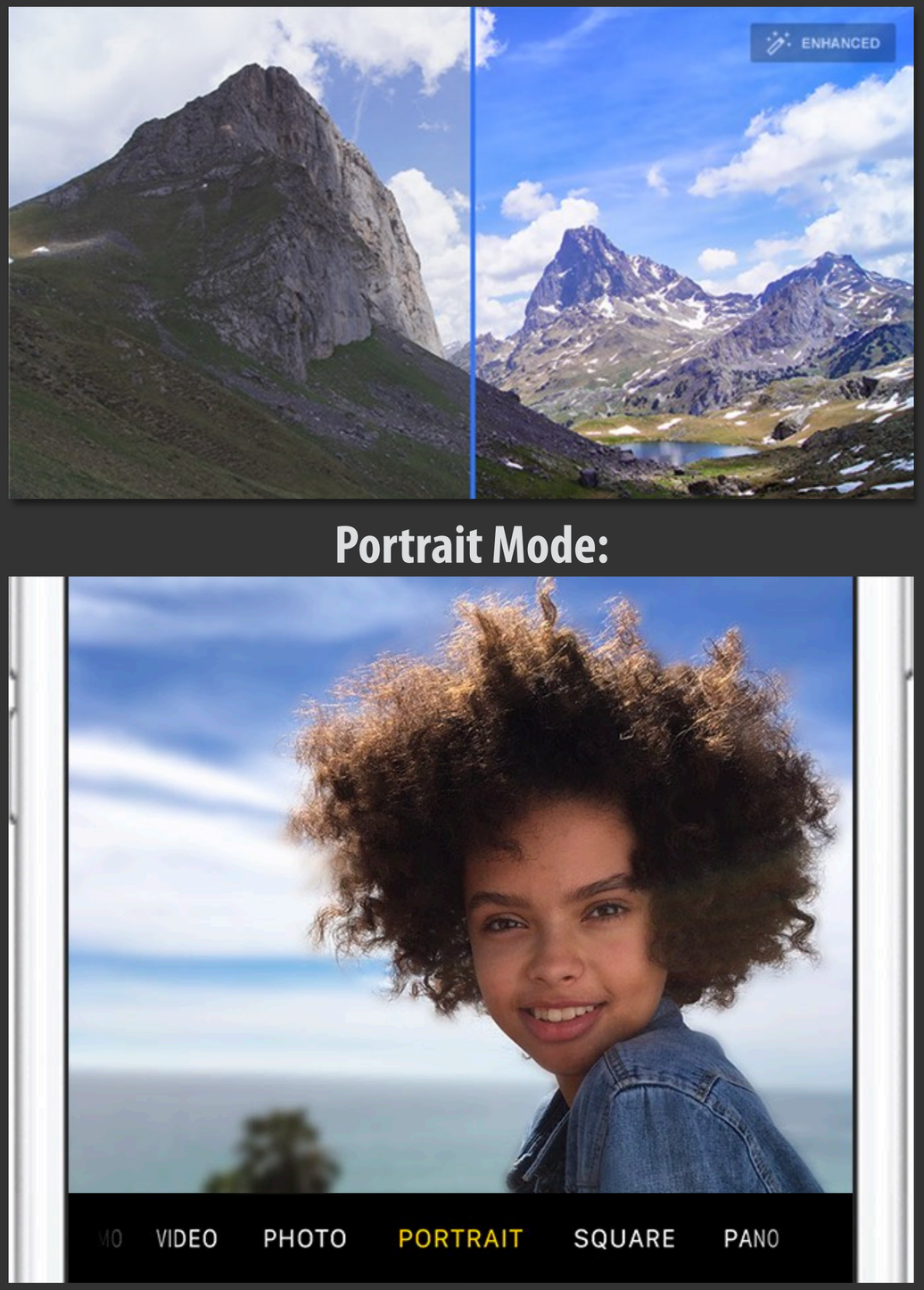


Future "retina" VR display:
57 ppd covering 180°
= 10K x 10K display per eye
= 200 MPixel

RAW data rate @ 120Hz ≈ 72 GB/sec

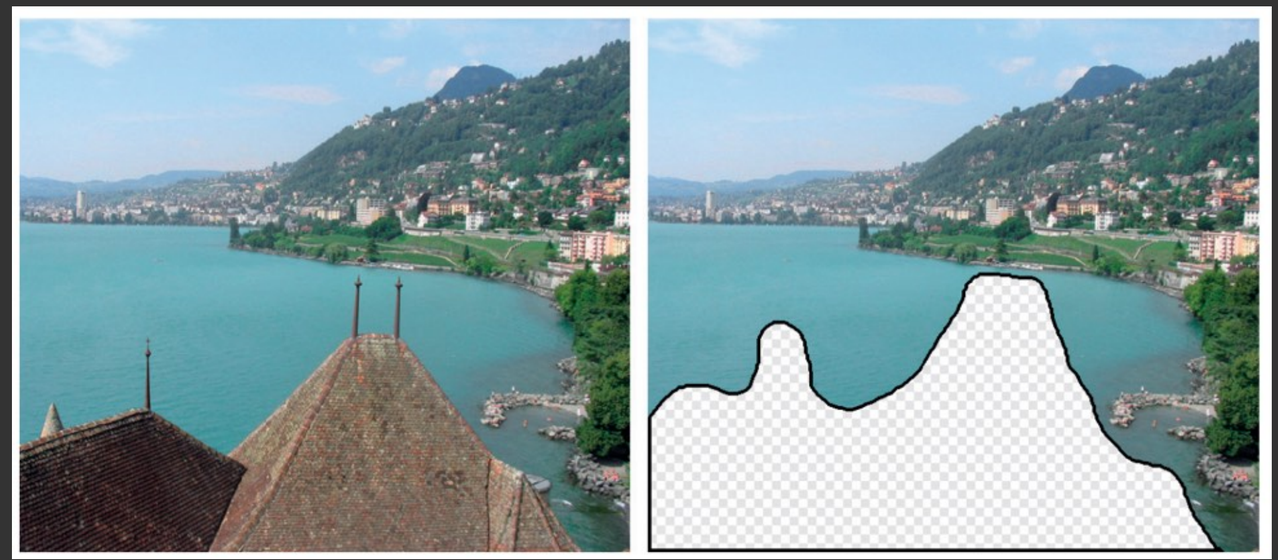
Enhancing communication: understanding images to improve acquired content

AutoEnhance:

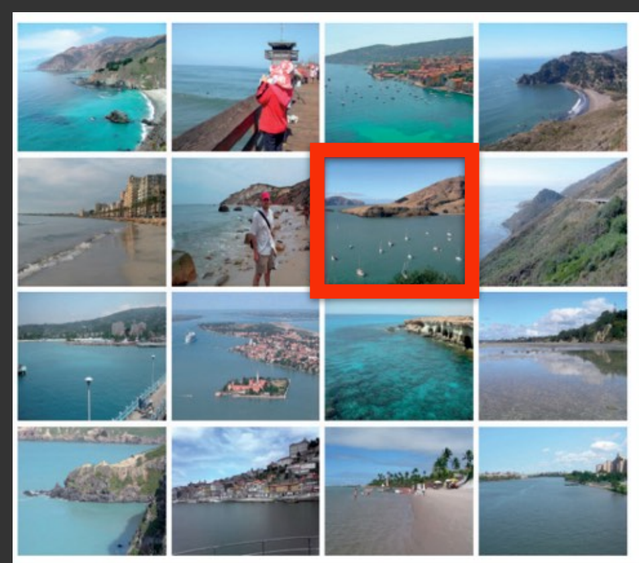


Portrait Mode:

Photo “fix up” [Hayes 2007]



My bad vacation photo

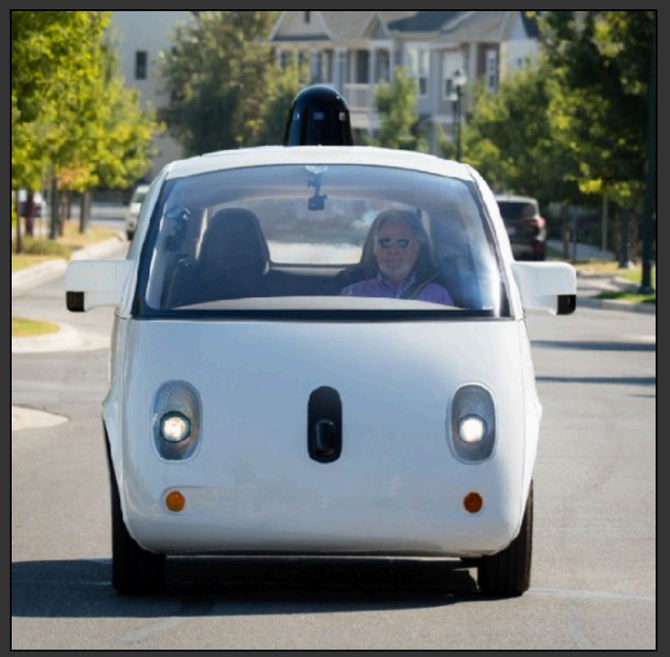


Similar photos others have taken

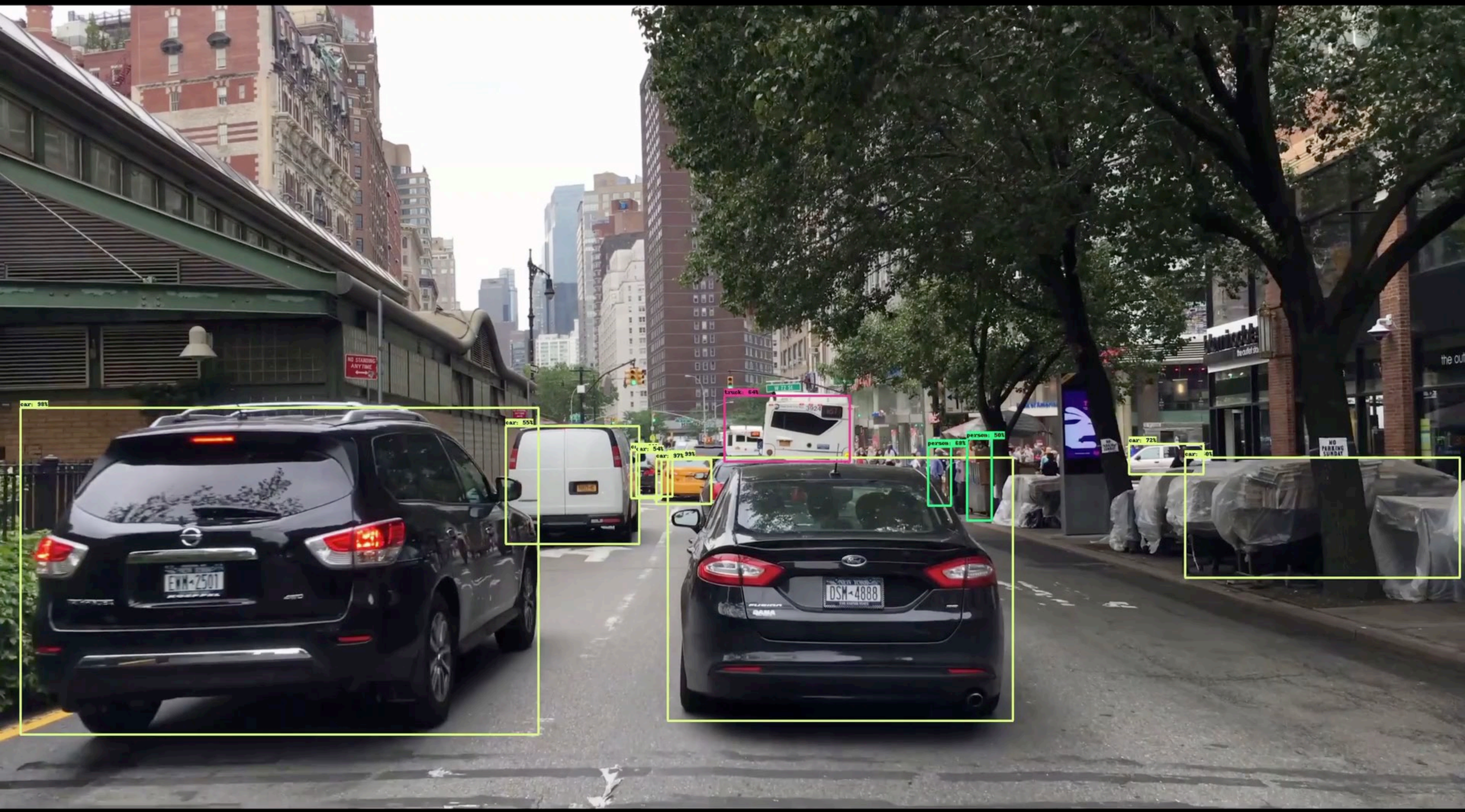
Part to fix

Fixed!

On every vehicle: analyzing images for robot navigation

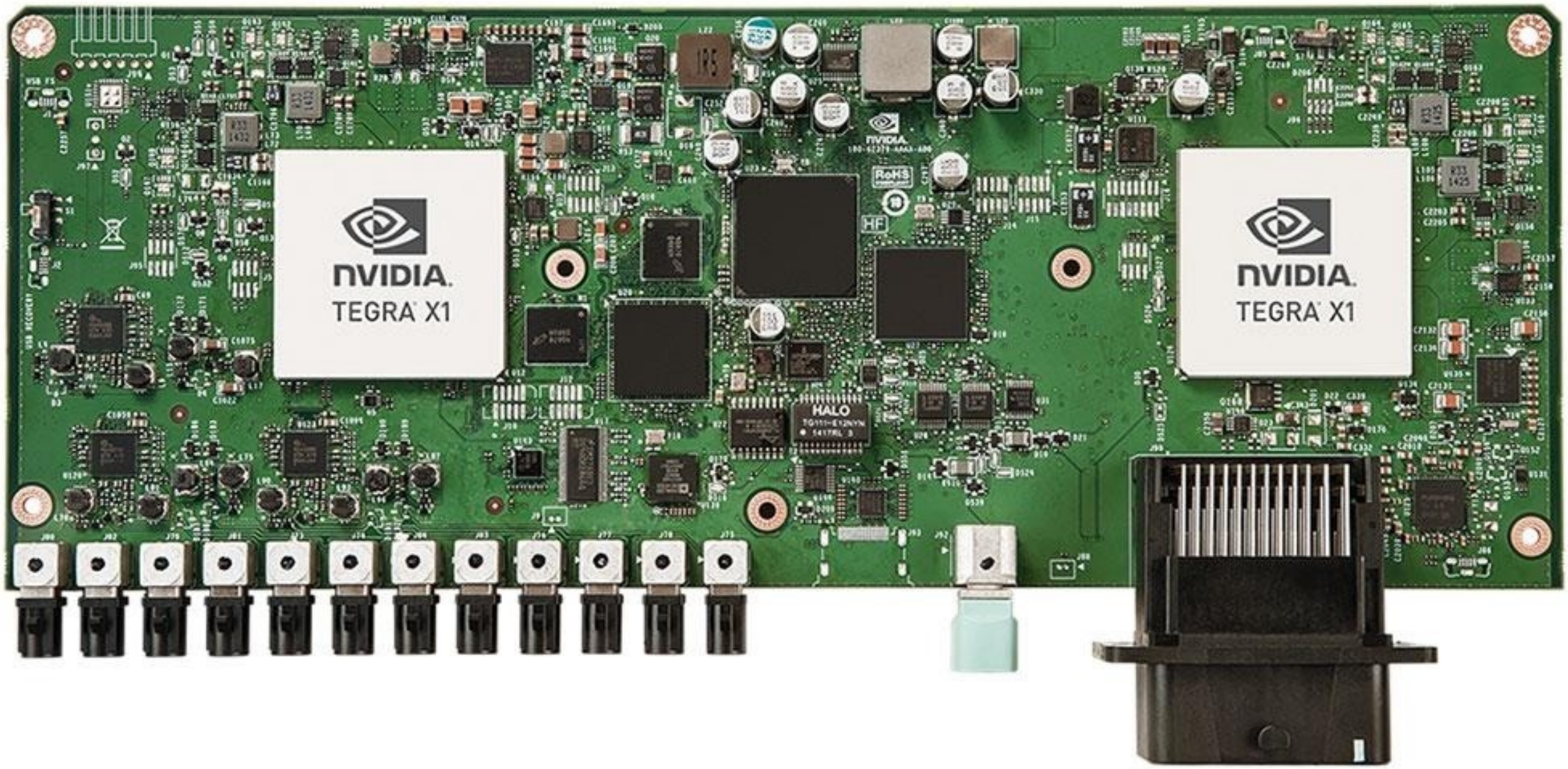


High-resolution video (moving camera)



[Image Credit: Kundu et al. 2016]

NVIDIA Drive PX



Tegra X1 (1 TFlop fp16 at 1GHz)

On every corner: analyzing images for urban efficiency

"Managing urban areas has become one of the most important development challenges of the 21st century. Our success or failure in building sustainable cities will be a major factor in the success of the post-2015 UN development agenda."

- UN Dept. of Economic and Social Affairs

High resolution (static camera)

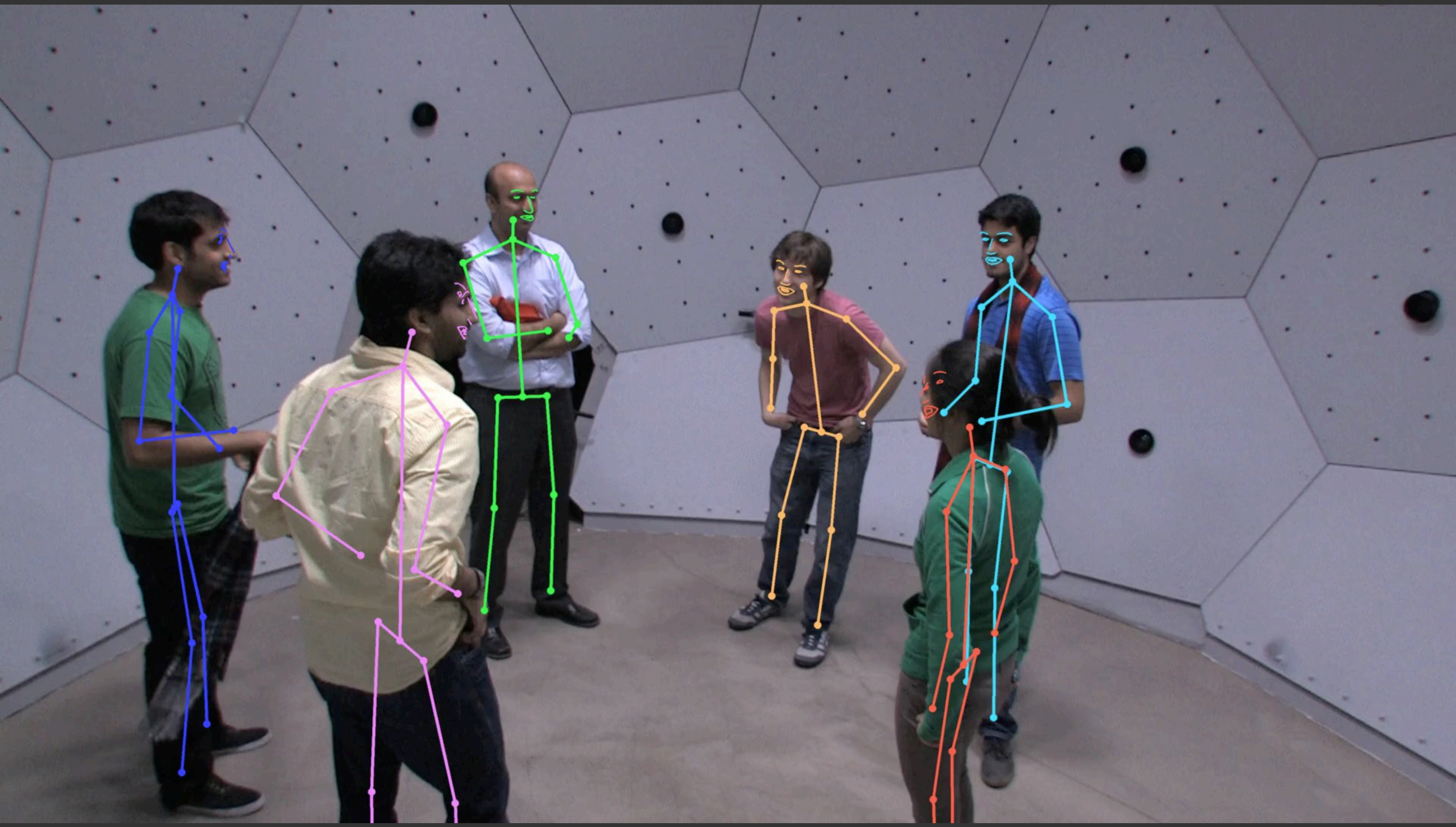
Sensing human social interactions

[Joo 2015]

CMU Panoptic Studio
480 video cameras (640 x 480 @ 25fps)

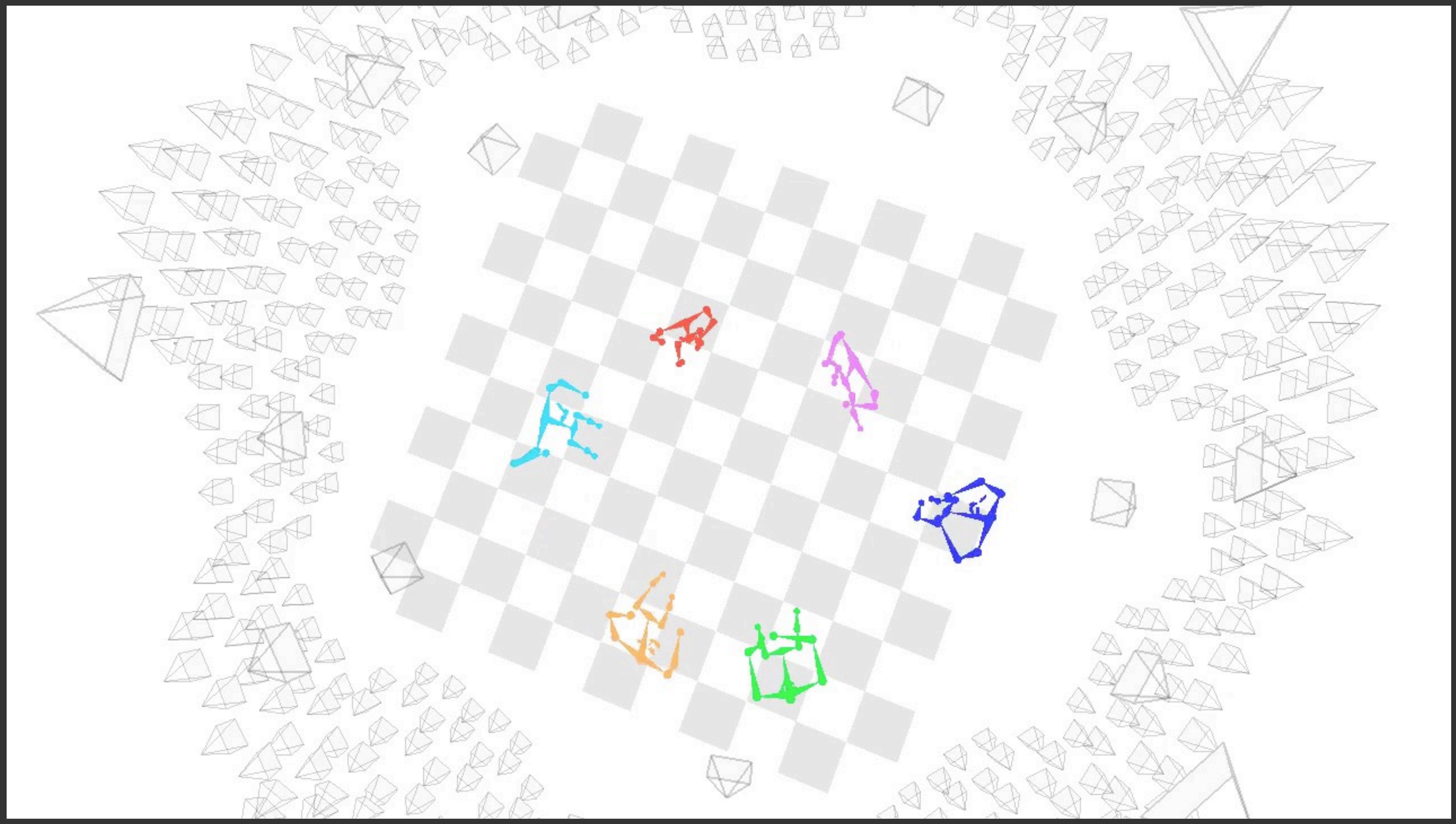
116 GPixel video sensor
(2.9 TPixel /sec)

Capturing social interactions



[Courtesy Yaser Sheikh, Tomas Simon, Hanbyul Joo]

Capturing social interactions



[Courtesy Yaser Sheikh, Tomas Simon, Hanbyul Joo]

On every human: analyzing egocentric images to augment humans

Gwangjiang Market (Seoul)

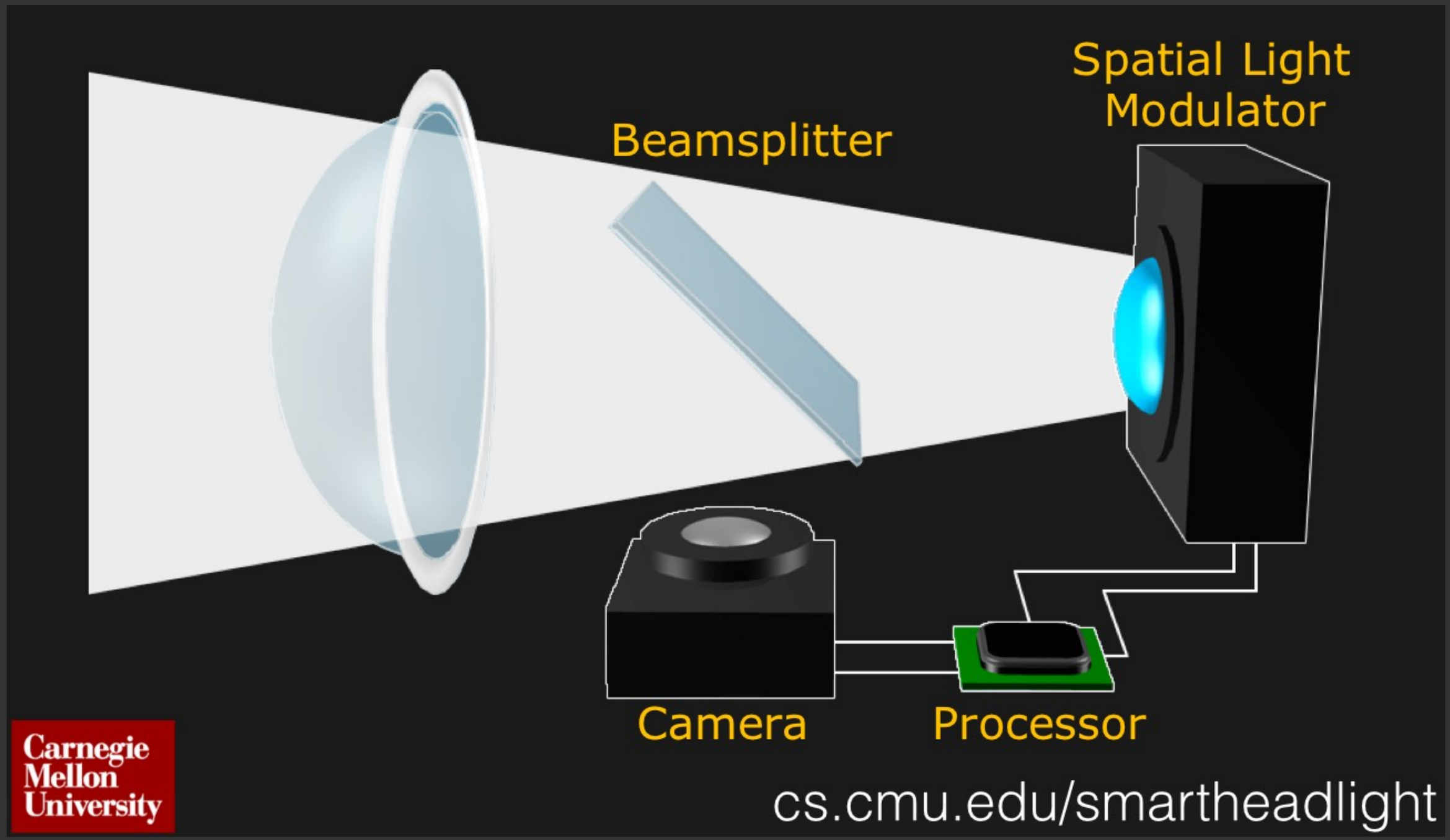
AR requires low-latency localization and scene object recognition

Carnegie Mellon University

cs.cmu.edu/smartheadlight

[Tamburo 2016]

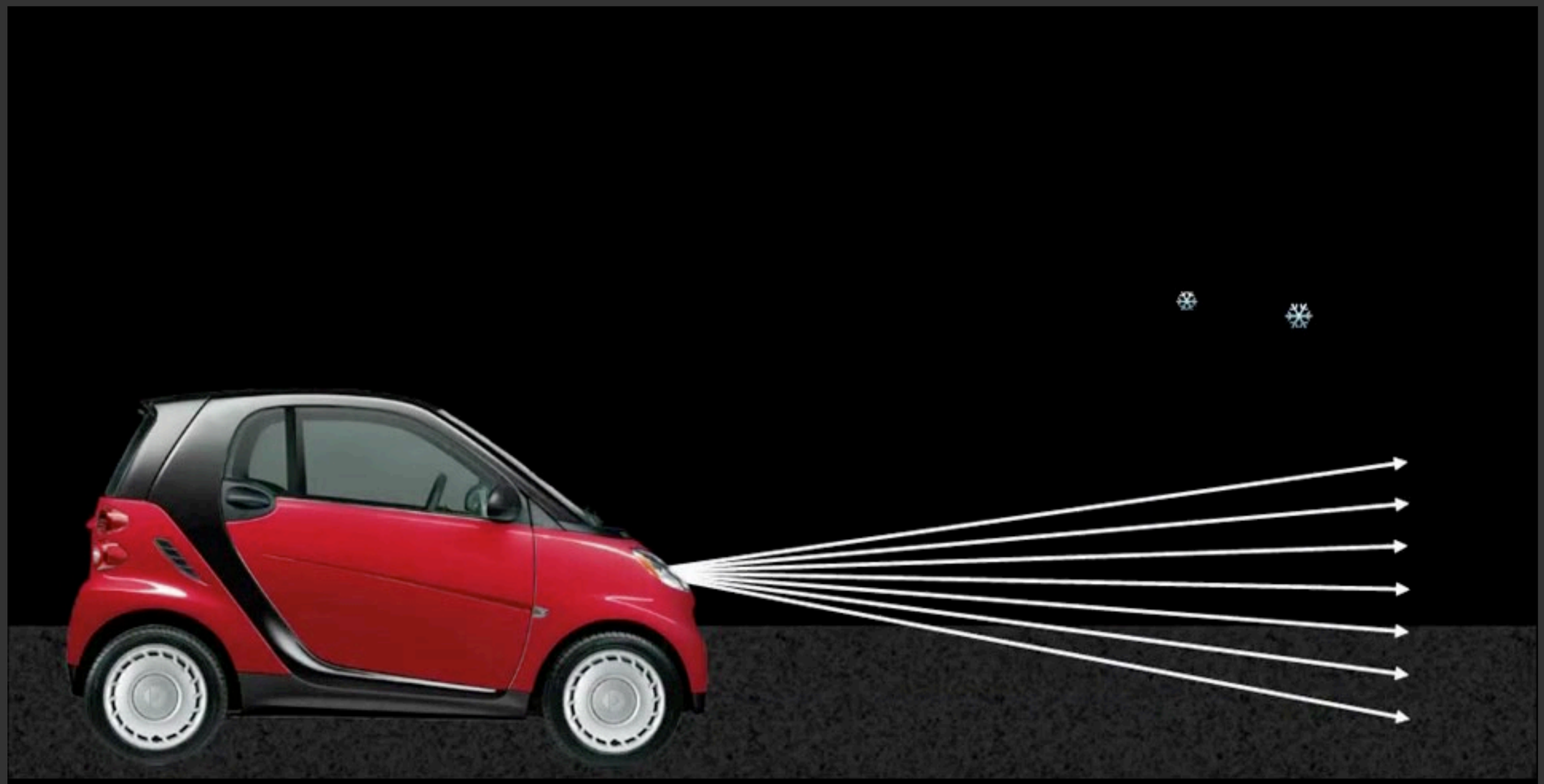
Smart headlight system



~1000 Hz (1 - 1.5 ms latency)

[Tamburo 2016]

Seeing clearly through precipitation



Idea: Stream Light Between Snowflakes

Goal: High Light Throughput and Accuracy

Illustration adapted from de Charette (ICCP, 2012)

Future challenge: recording and analyzing the world's visual information, so computers can understand and reason about it

Capturing everything about the visual world

To understand people

To understand the world around vehicles/drones

To understand cities

Mobile

Continuous (always on)

Exceptionally high resolution

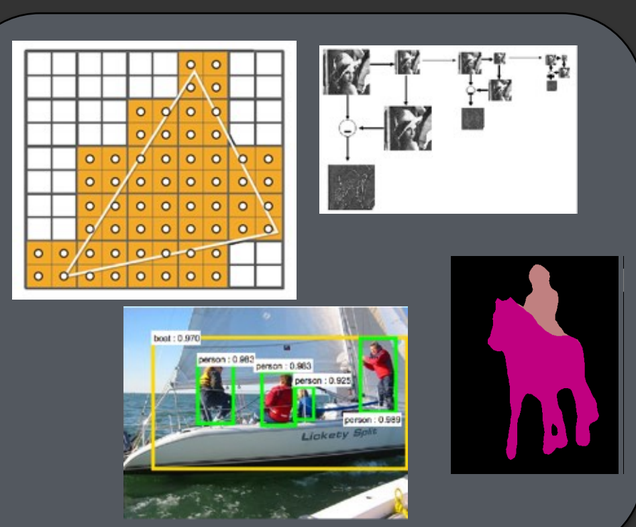
Capture for computers to analyze, not humans to watch

What is this course about?

1. Understanding the characteristics of important visual computing workloads
2. Understanding techniques used to achieve efficient system implementations

VISUAL COMPUTING WORKLOADS

Algorithms for 3D graphics, image processing, compression, etc.



Parallelism
Exploiting locality
Minimizing communication

MACHINE ORGANIZATION

High-throughput hardware designs: Parallel, heterogeneous, specialized

DESIGN OF PROGRAMMING ABSTRACTIONS FOR VISUAL COMPUTING

choice of programming primitives
level of abstraction

OpenGL

TensorFlow

mxnet

In other words

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

To design more efficient algorithms

To build the most efficient hardware to run these applications

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

Course Logistics

Logistics

- Course web site:
  - http://graphics.stanford.edu/courses/cs348v-18-winter
- All announcements will go out via Piazza
  - https://piazza.com/class/jc1f626cfne6r6
- Kayvon's office hours: Tuesday after class, or by appt.

Expectations of you

- 20% participation
  - There will be ~1 assigned paper reading per class
  - Everyone is expected to come to class and participate in discussions based on readings
  - You are encouraged discuss papers and or my lectures on the course discussion board.
  - If you form a weekly course reading/study group, I will buy Pizza for said group.

- 30% mini-assignments (3 short programming assignments)
  - Assignment 1: analyze parallel program performance on a multi-core CPU
  - Assignment 2: implement and optimize a basic RAW image processing pipeline
  - Assignment 3: optimize performance of a modern DNN module

- 20% 1 take-home "exam"

- 30% self-selected final project
  - I suggest you start thinking about projects now (can be teams of up to two)

Major course themes/topics

Part 1: High Efficiency Image and Video Processing

Overview of a Modern Digital Camera Processing Pipeline
Image Processing Algorithms You Should Know
Efficiently Scheduling Image Processing Algorithms on Parallel Hardware
Specialized Hardware for Image Processing
Lossy Image (JPG) and Video (H.264) Video Compression
Video Processing/Synthesis for Virtual Reality Display

Part 2: Accelerating Deep Learning for Computer Vision (from a systems perspective)

Workload Characteristics of DNN Inference for Image Analysis
Scheduling and Algorithms for Parallel DNN Training at Scale
A Case Study of Algorithmic Optimizations for Object Detection
Leveraging Task-Specific DNN Structure for Improving Performance and Accuracy
Hardware Accelerators for DNN Inference
Design Space of Dataflow Programming Abstractions for Deep Learning
Enhancing Efficiency Through Model Specialization (in particular for video)
Efficient Inference at Datacenter Scale

Algorithmic innovation in image classification

Improving *accuracy-per-unit cost* using better DNN designs?

2014 → 2017 ~ 25x improvement in cost at similar accuracy

	ImageNet Top-1 Accuracy	Num Params	Cost/image (MADDs)	
VGG-16	71.5%	138M	15B	[2014]
GoogleNet	70%	6.8M	1.5B	[2015]
ResNet-18	73% *	11.7M	1.8B	[2016]
MobileNet-224	70.5%	4.2M	0.6B	[2017]

* 10-crop results (ResNet 1-crop results are similar to other DNNs in this table)

Major course themes/topics

Part 3: The GPU Accelerated 3D Graphics Pipeline

Real-Time 3D Graphics Pipeline Architecture
Hardware Acceleration of Z-Buffering and Texturing
Scheduling the Graphics Pipeline onto a GPU
Domain Specific Languages for Shading

Review: key principles of modern throughput computing hardware

Review concepts

- What are these design concepts, and what problem/goals do they address?
  - Muti-core processing
  - SIMD processing
  - Hardware multi-threading

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

- What is memory bandwidth a major constraint when mapping applications to modern computer systems?

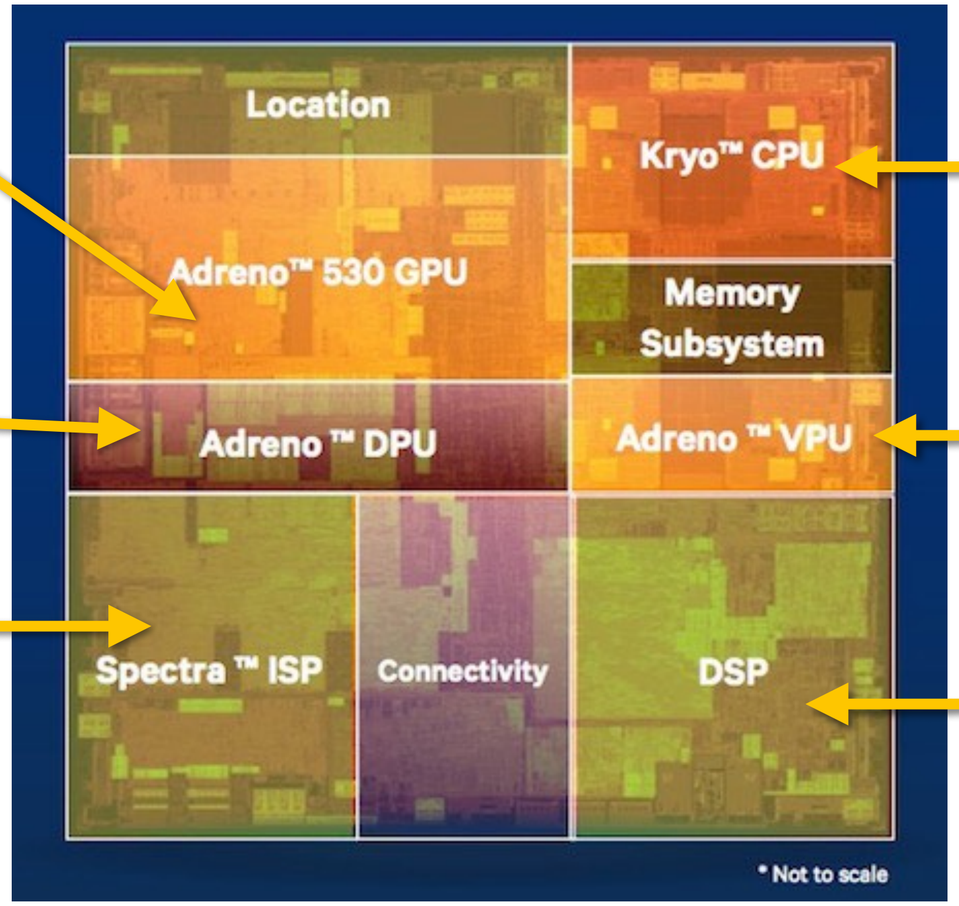
Let's crack open a modern smartphone

Samsung Galaxy S7 phone with Qualcomm Snapdragon 820 processor

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

Display engine (compresses pixels for transfer to 4K screen)

Image Signal Processor (ISP): ASIC for processing pixels off camera (25MP at 30Hz)



Multi-core ARM CPU

Video encode/decode ASIC (H.265 @ 4K)

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

Multi-core processing

Review: what does a processor do?

It runs programs!

Processor executes instruction referenced by the program counter (PC)

(executing the instruction will modify machine state: contents of registers, memory, CPU state, etc.)

Move to next instruction …

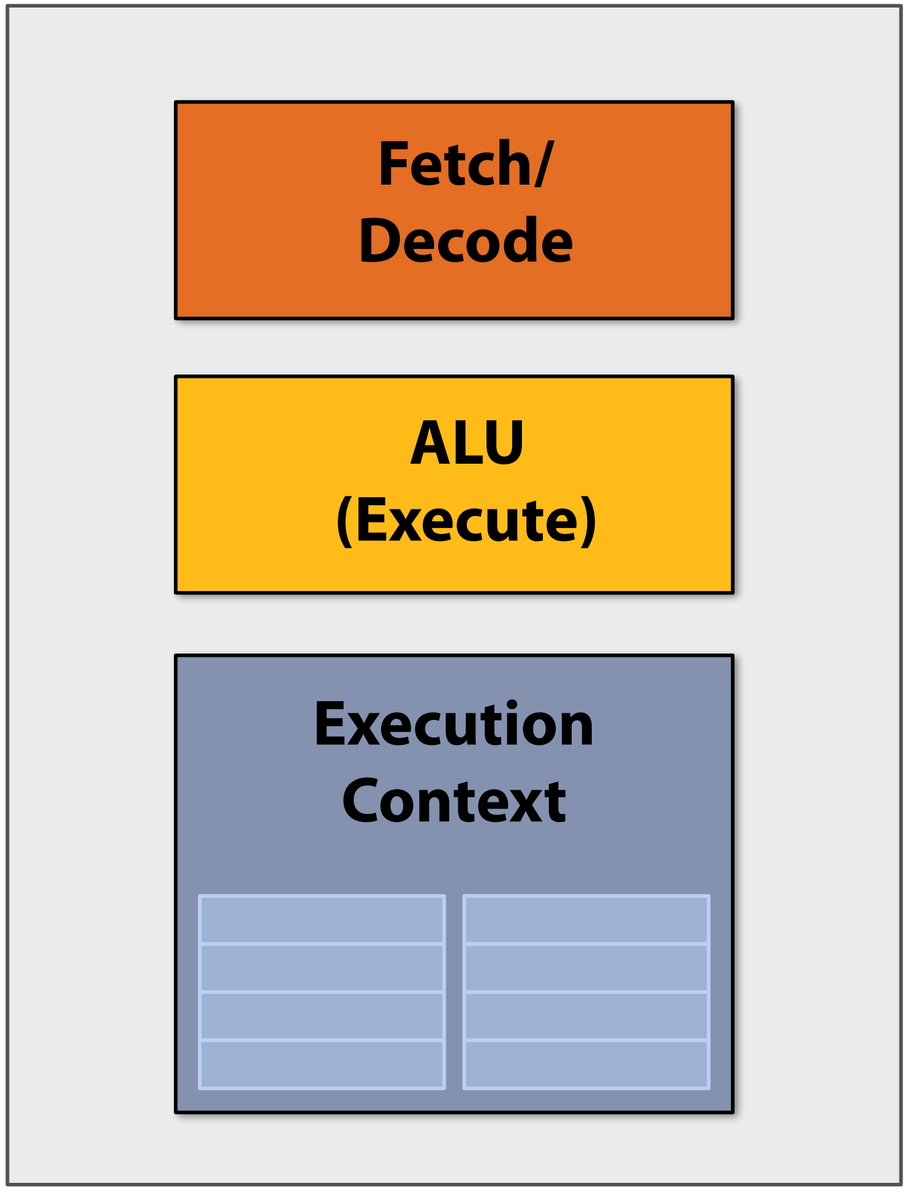
Then execute it…

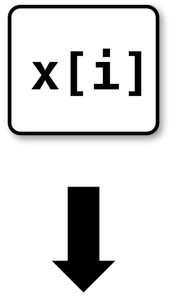
And so on…

PC

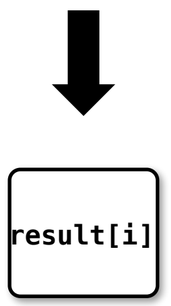
```
_main:
100000f10:   pushq   %rbp
100000f11:   movq%rsp, %rbp
100000f14:   subq$32, %rsp
100000f18:   movl$0, -4(%rbp)
100000f1f:   movl%edi, -8(%rbp)
100000f22:   movq%rsi, -16(%rbp)
100000f26:   movl$1, -20(%rbp)
100000f2d:   movl$0, -24(%rbp)
100000f34:   cmpl$10, -24(%rbp)
100000f38:   jge 23 <_main+0x45>
100000f3e:   movl-20(%rbp), %eax
100000f41:   addl-20(%rbp), %eax
100000f44:   movl%eax, -20(%rbp)
100000f47:   movl-24(%rbp), %eax
100000f4a:   addl$1, %eax
100000f4d:   movl%eax, -24(%rbp)
100000f50:   jmp -33 <_main+0x24>
100000f55:   leaq58(%rip), %rdi
100000f5c:   movl-20(%rbp), %esi
100000f5f:   movb$0, %al
100000f61:   callq   14
100000f66:   xorl%esi, %esi
100000f68:   movl%eax, -28(%rbp)
100000f6b:   movl%esi, %eax
100000f6d:   addq$32, %rsp
100000f71:   popq%rbp
100000f72:   retq
```

Executing an instruction stream



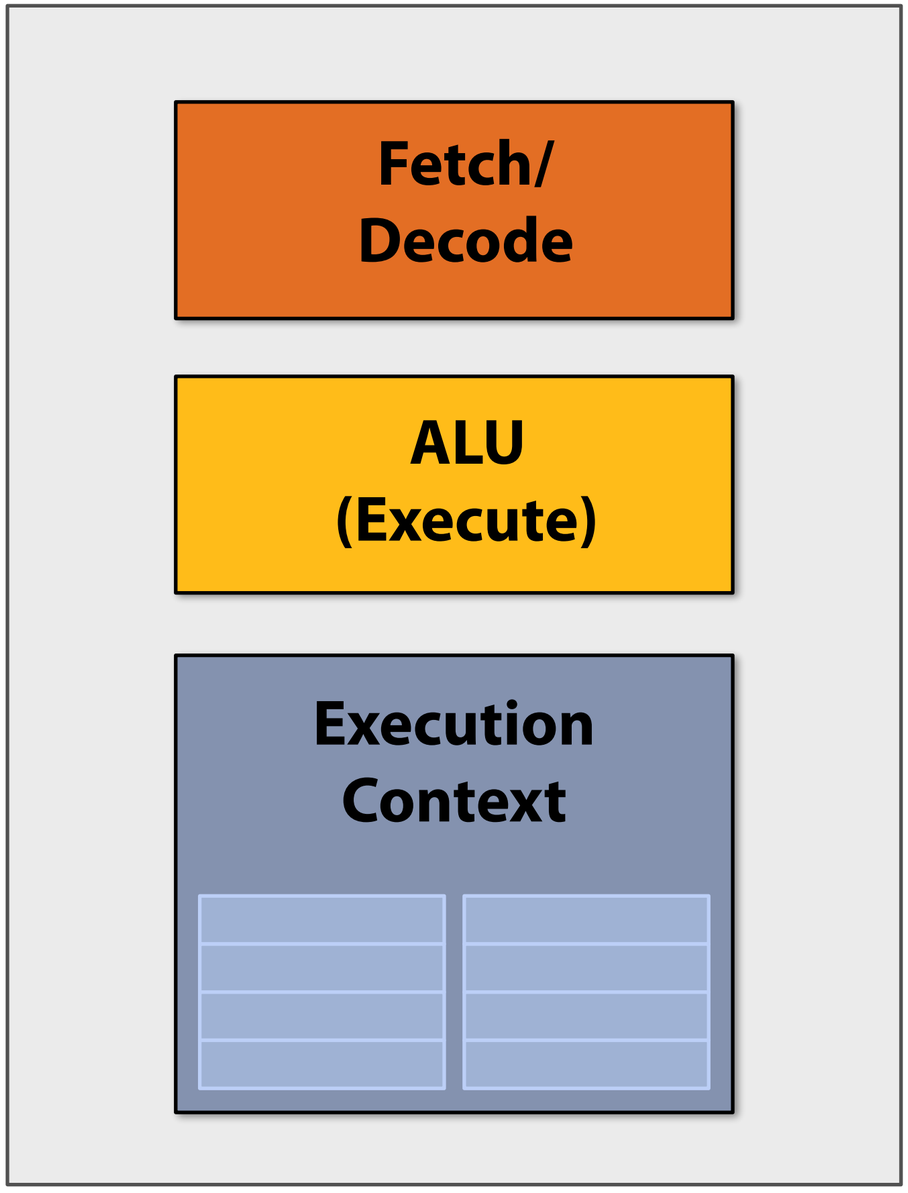


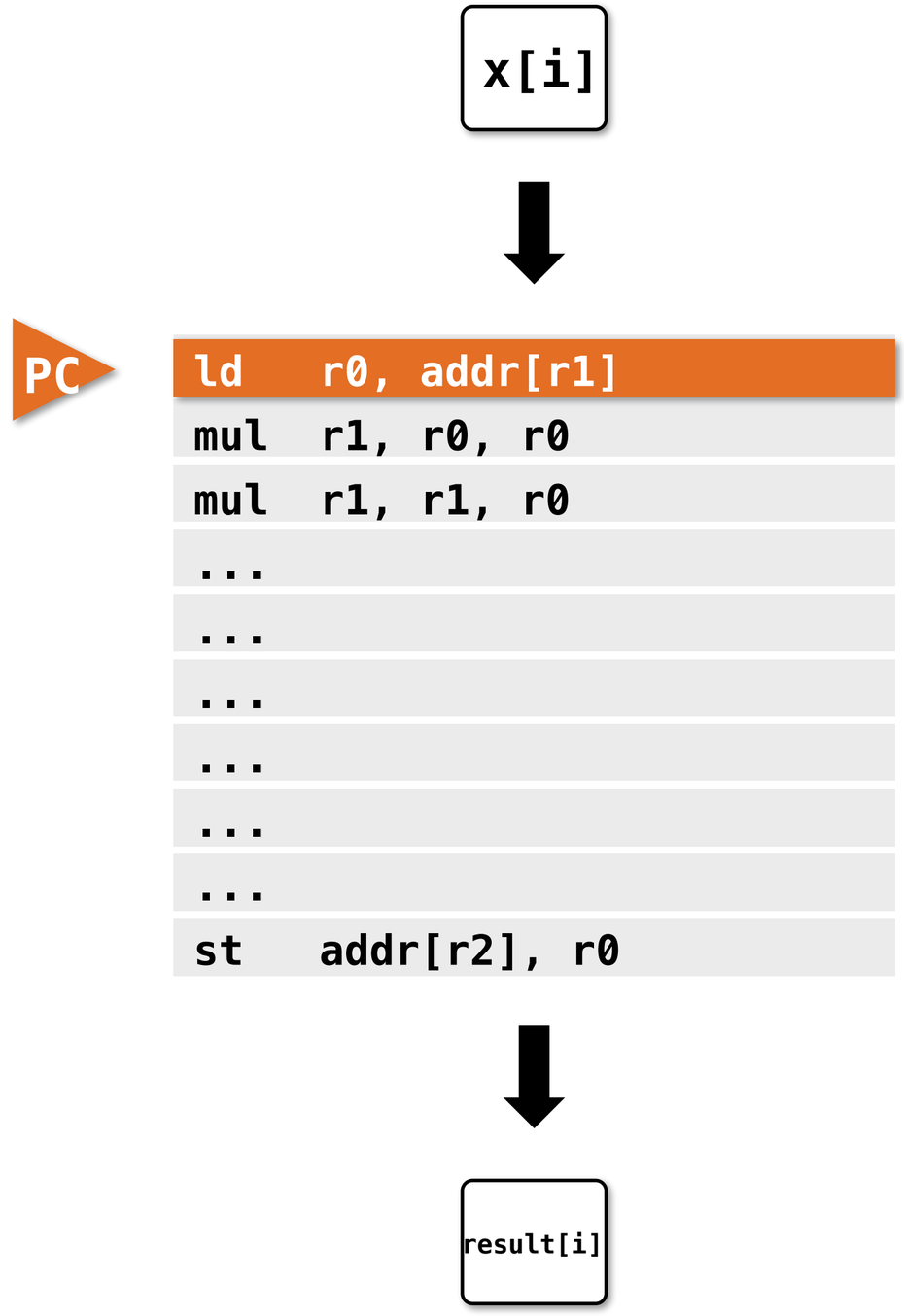

```
ld    r0, addr[r1]
mul   r1, r0, r0
mul   r1, r1, r0
...
...
...
...
...
...
st    addr[r2], r0
```



Executing an instruction stream

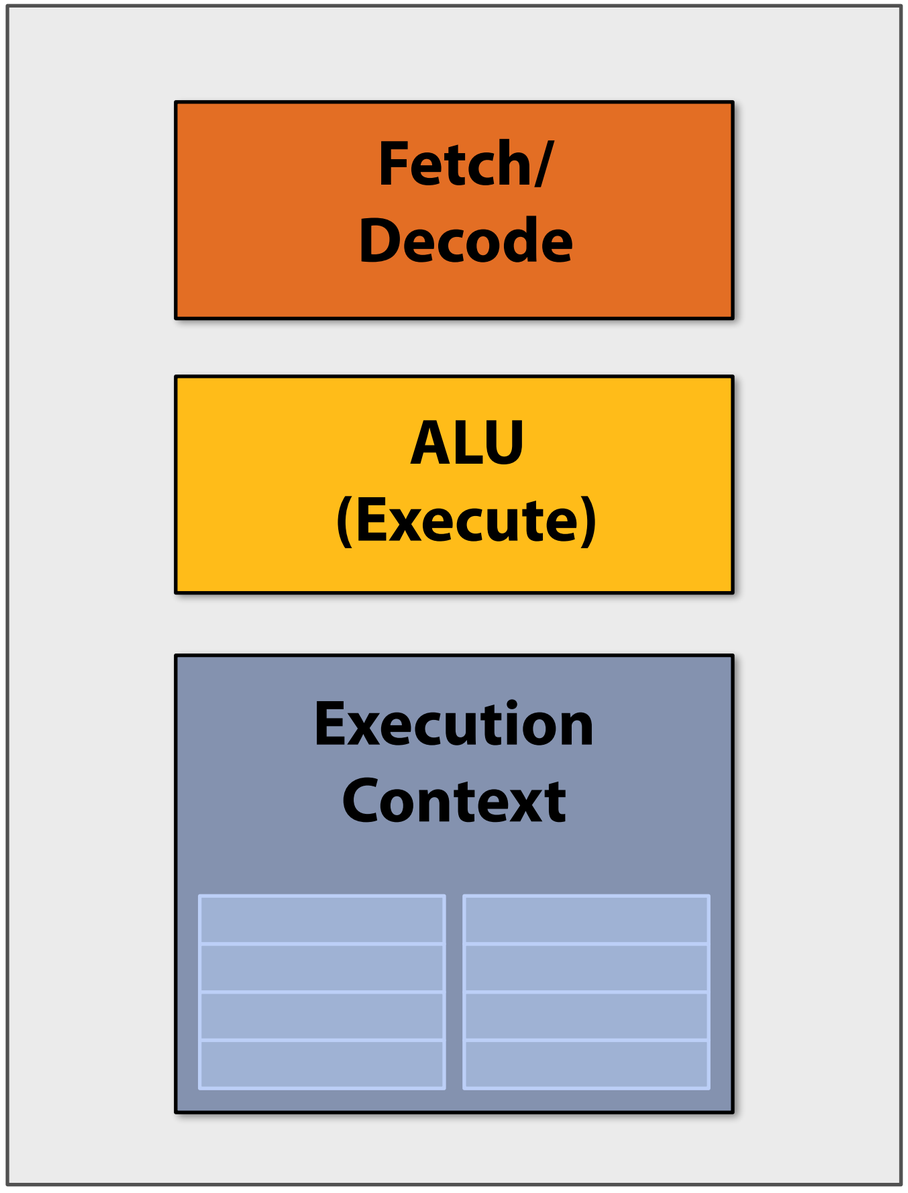
My very simple processor: executes one instruction per clock

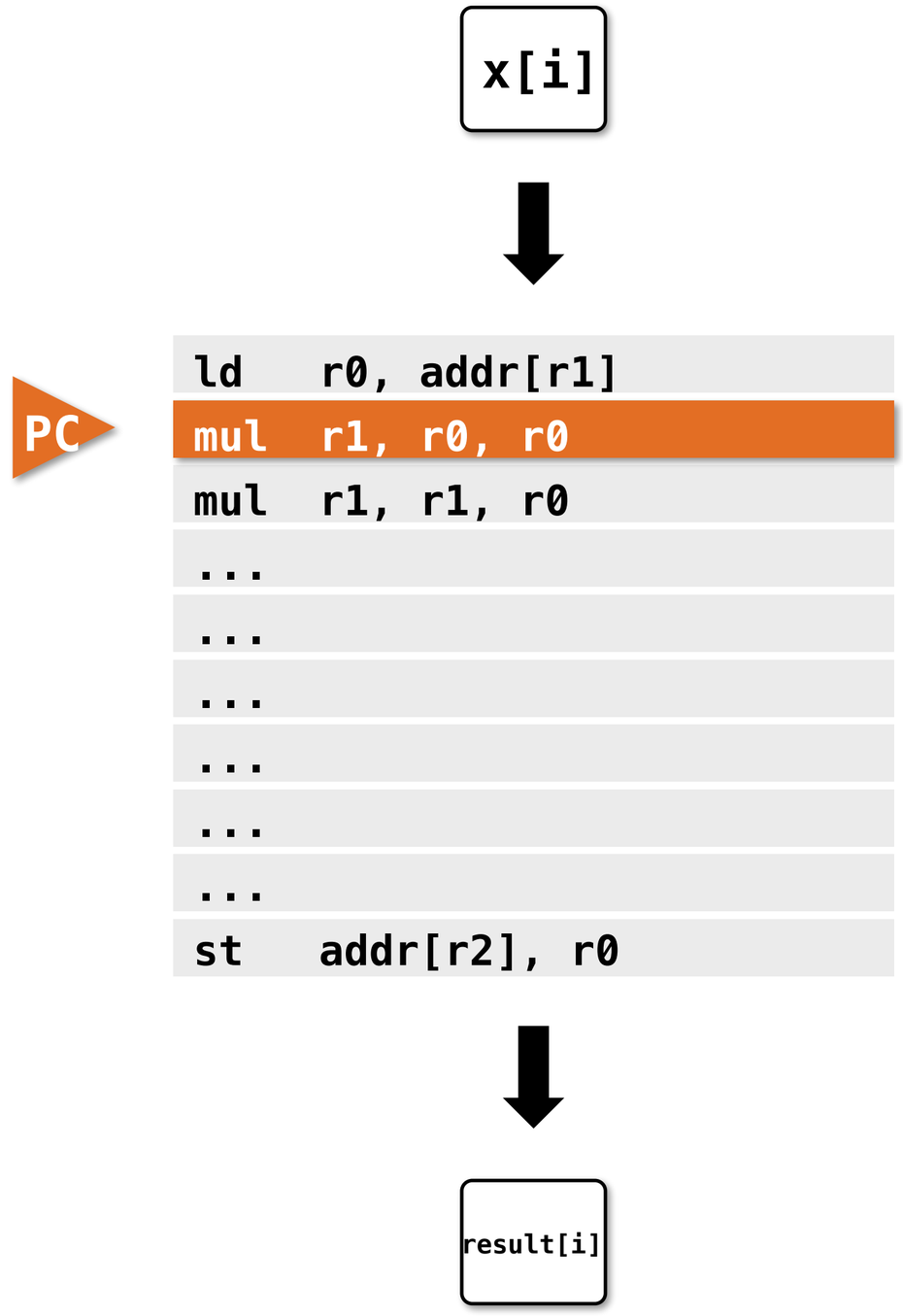




Executing an instruction stream

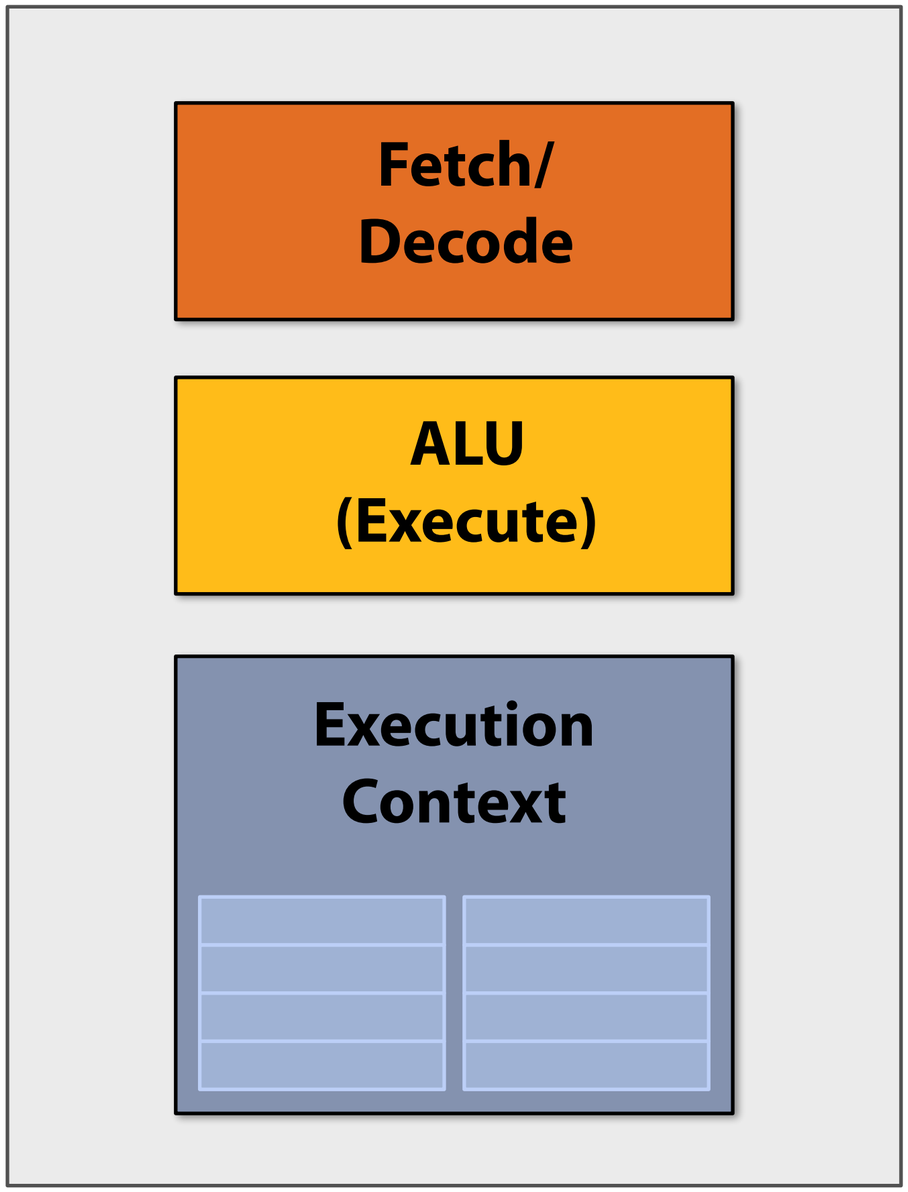
My very simple processor: executes one instruction per clock

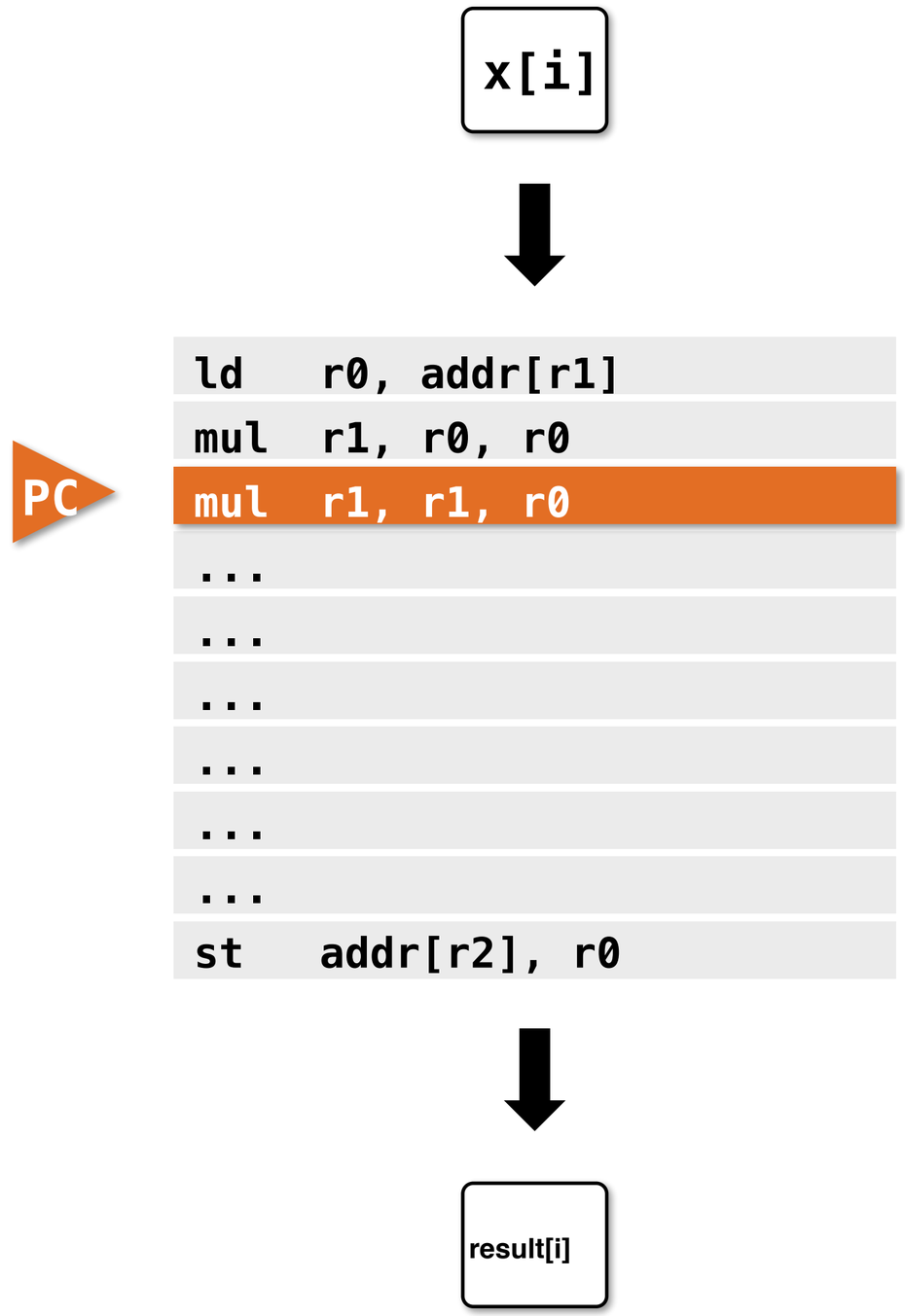




Executing an instruction stream

My very simple processor: executes one instruction per clock





Quick aside:
Instruction-level parallelism and superscalar execution

Instruction level parallelism (ILP) example

$$a = x*x + y*y + z*z$$

Consider the following program:

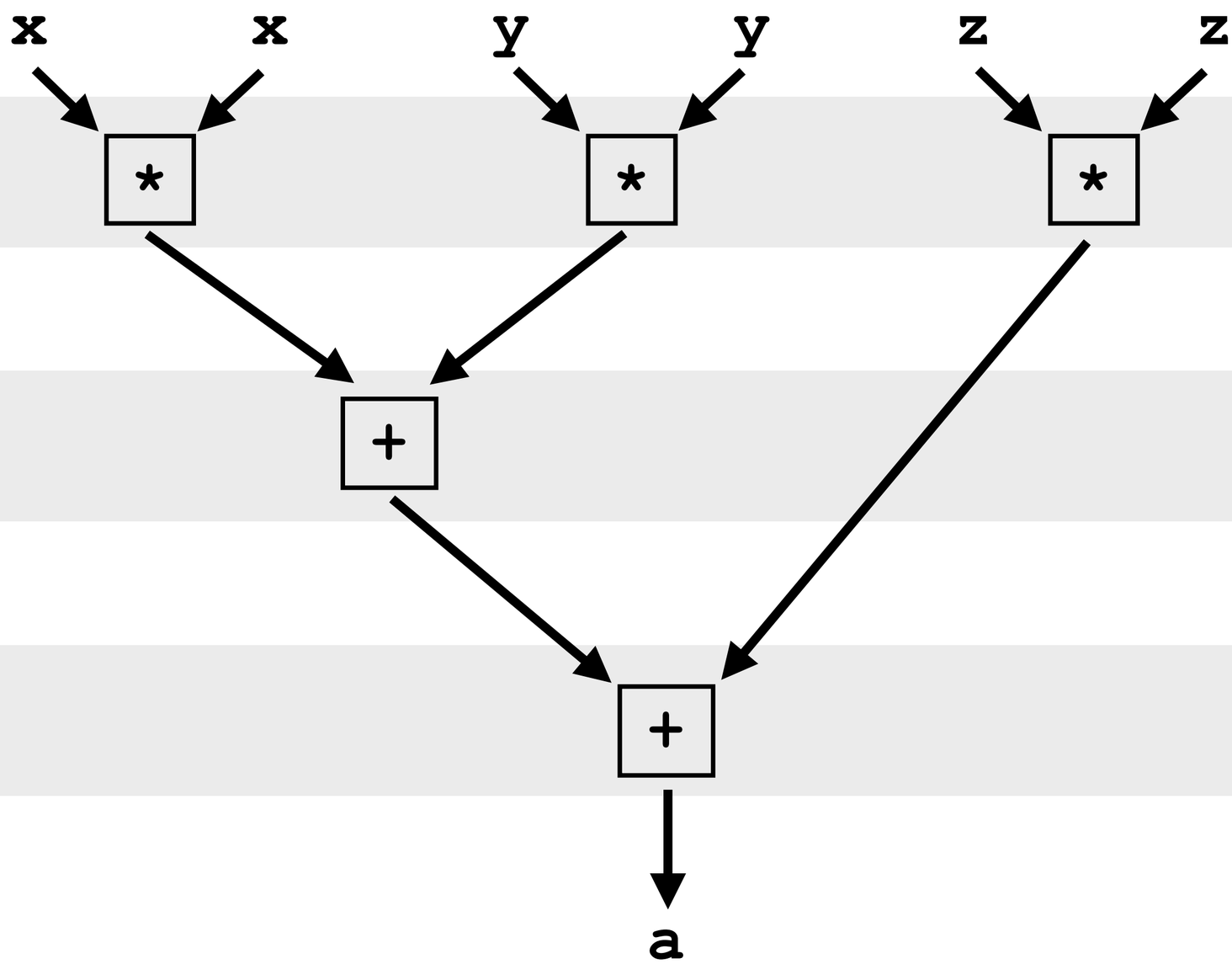
```
// assume r0=x, r1=y, r2=z

mul r0, r0, r0
mul r1, r1, r1
mul r2, r2, r2
add r0, r0, r1
add r3, r0, r2

// now r3 stores value of program variable ‘a’
```

This program has five instructions, so it will take five clocks to execute, correct?

Can we do better?

ILP example

```
a = x*x + y*y + z*z
```



ILP = 3

ILP = 1

ILP = 1

Superscalar execution

$$a = x*x + y*y + z*z$$

```
   // assume r0=x, r1=y, r2=z

1. mul r0, r0, r0
2. mul r1, r1, r1
3. mul r2, r2, r2
4. add r0, r0, r1
5. add r3, r0, r2

   // r3 stores value of variable ‘a’
```

Superscalar execution: processor automatically finds independent instructions in an instruction sequence and executes them in parallel on multiple execution units!

In this example: instructions 1, 2, and 3 can be executed in parallel (on a superscalar processor that determines that the lack of dependencies exists)

But instruction 4 must come after instructions 1 and 2

And instruction 5 must come after instruction 4

Superscalar execution

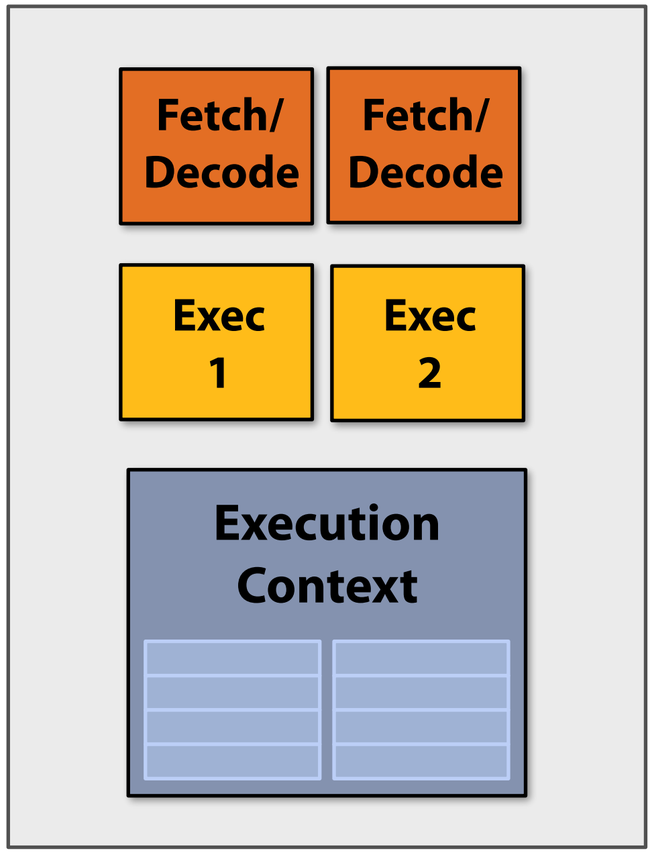
Program: computes sin of input *x* via Taylor expansion

```
void sinx(int N, int terms, float x)
{
      float value = x;
      float numer = x * x * x;
      int denom = 6;  // 3!
      int sign = -1;

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

      return value;
}
```

Independent operations in instruction stream

(They are detected by the processor at run-time and may be executed in parallel on execution units 1 and 2)

My single core, superscalar processor: executes up to two instructions per clock from a single instruction stream.



Now consider a program that computes the sine of many numbers…

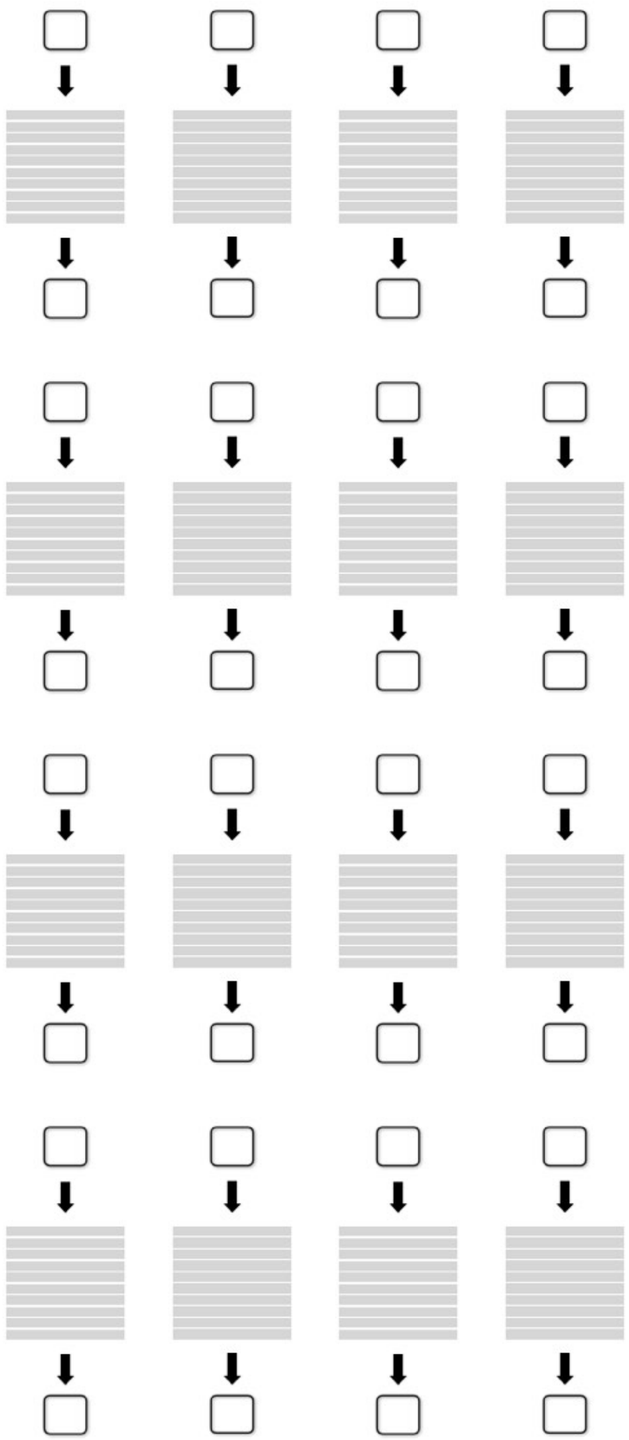
Example program

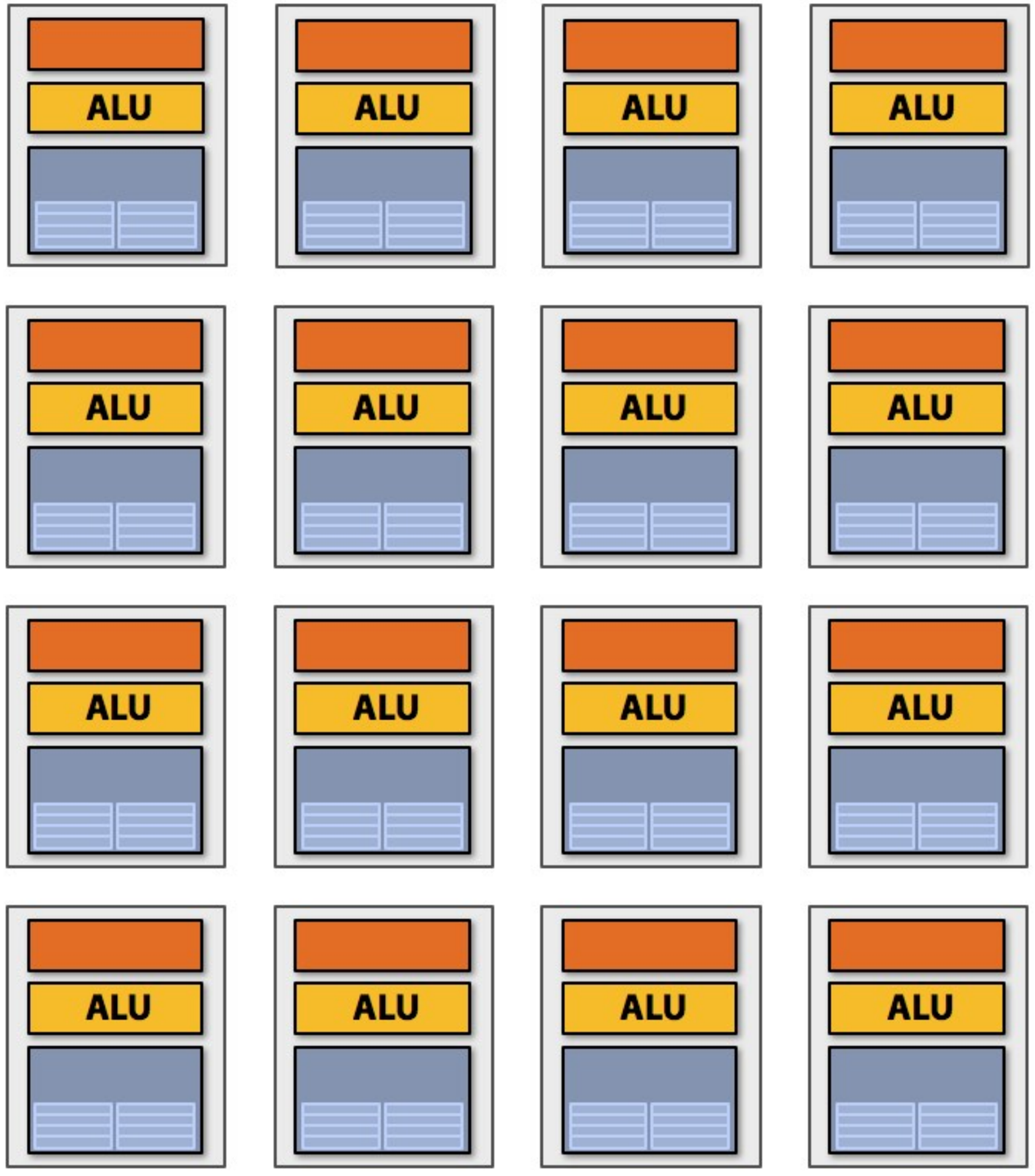
Compute $\sin(x)$ **using Taylor expansion:** $\sin(x) = x - x^3/3! + x^5/5! - x^7/7! + \ldots$

for each element of an array of N floating-point numbers

```
void sinx(int N, int terms, float* x, float* result)
{
   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;
      }

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

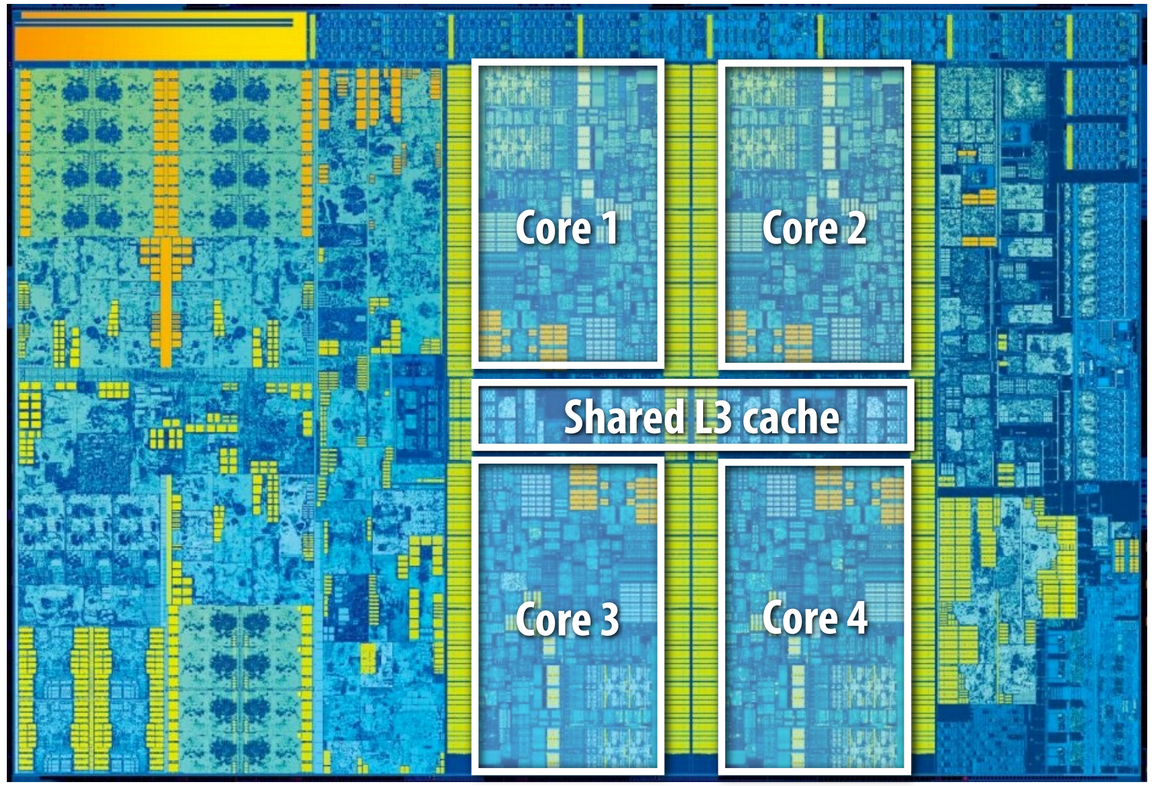
Multi-core: process multiple instruction streams in parallel



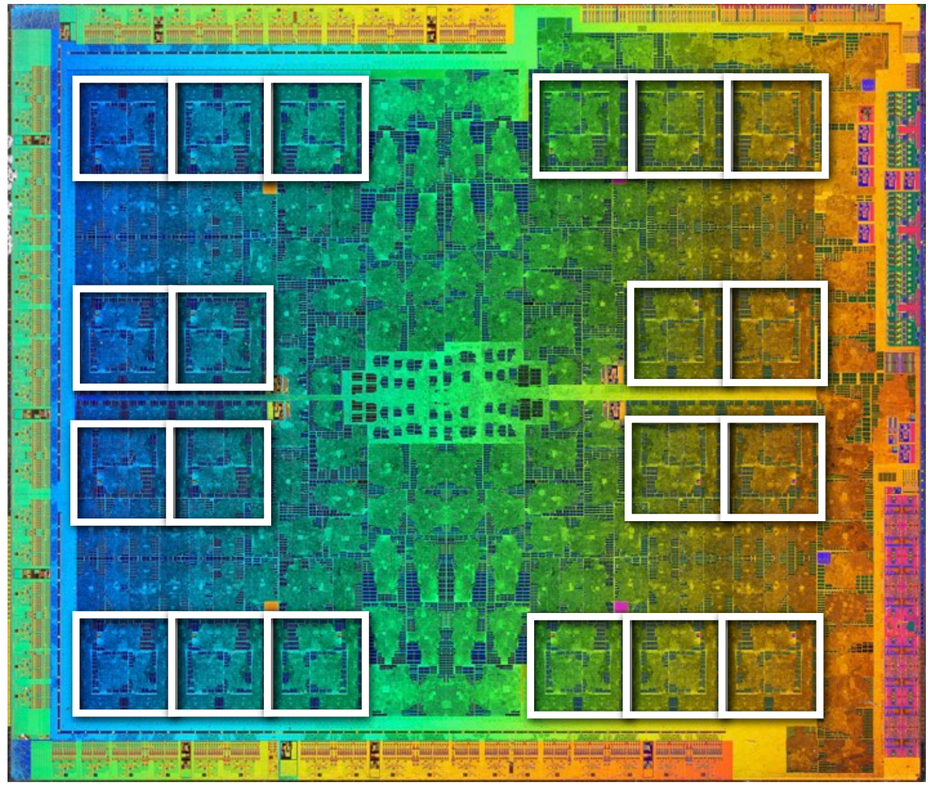


Sixteen cores, sixteen simultaneous instruction streams

Multi-core examples

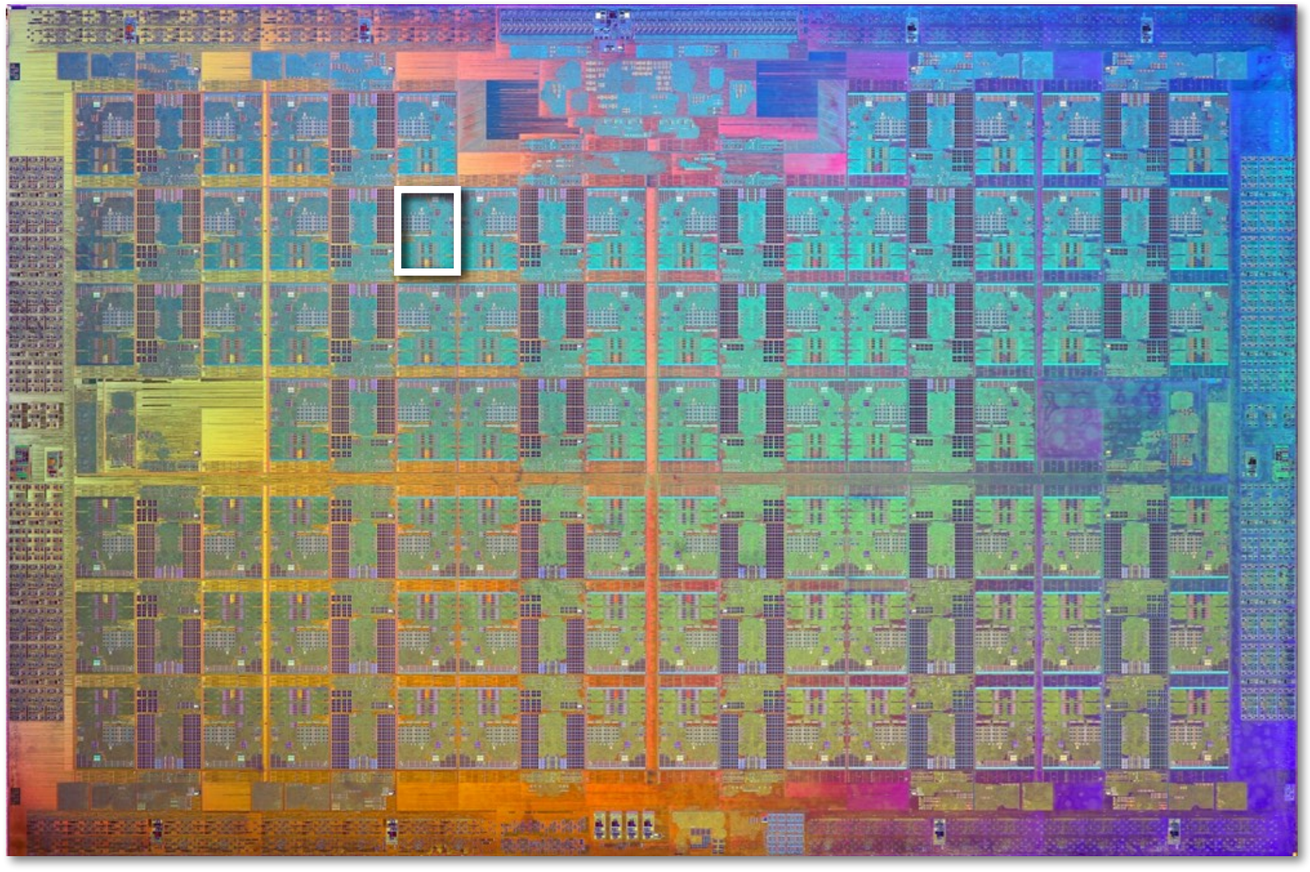


Intel "Skylake" Core i7 quad-core CPU
(2015)

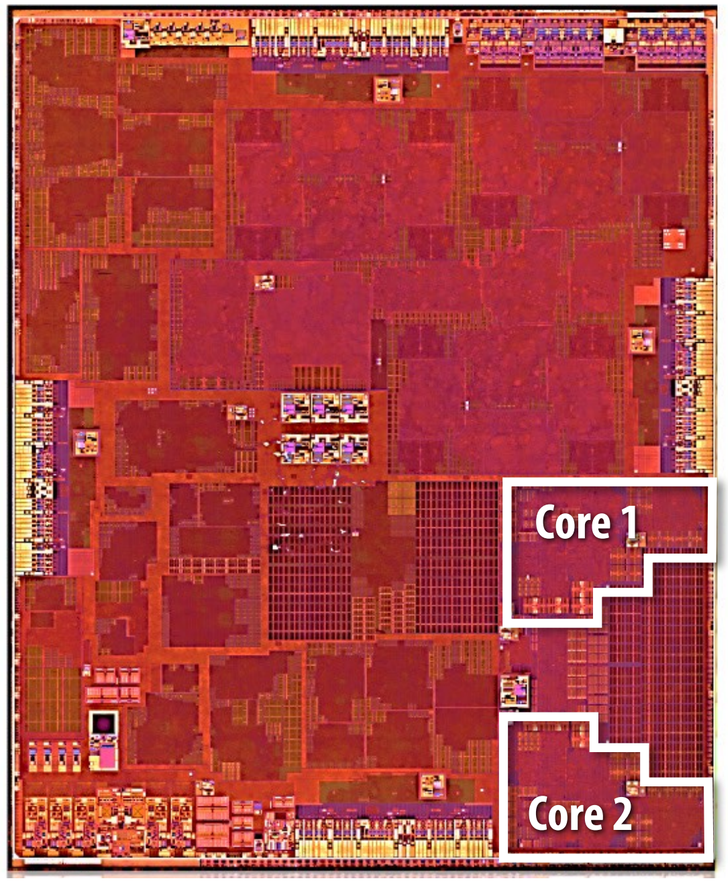


NVIDIA GP104 (GTX 1080) GPU
20 replicated ("SM") cores
(2016)

More multi-core examples



Intel Xeon Phi "Knights Landing " 76-core CPU
(2015)

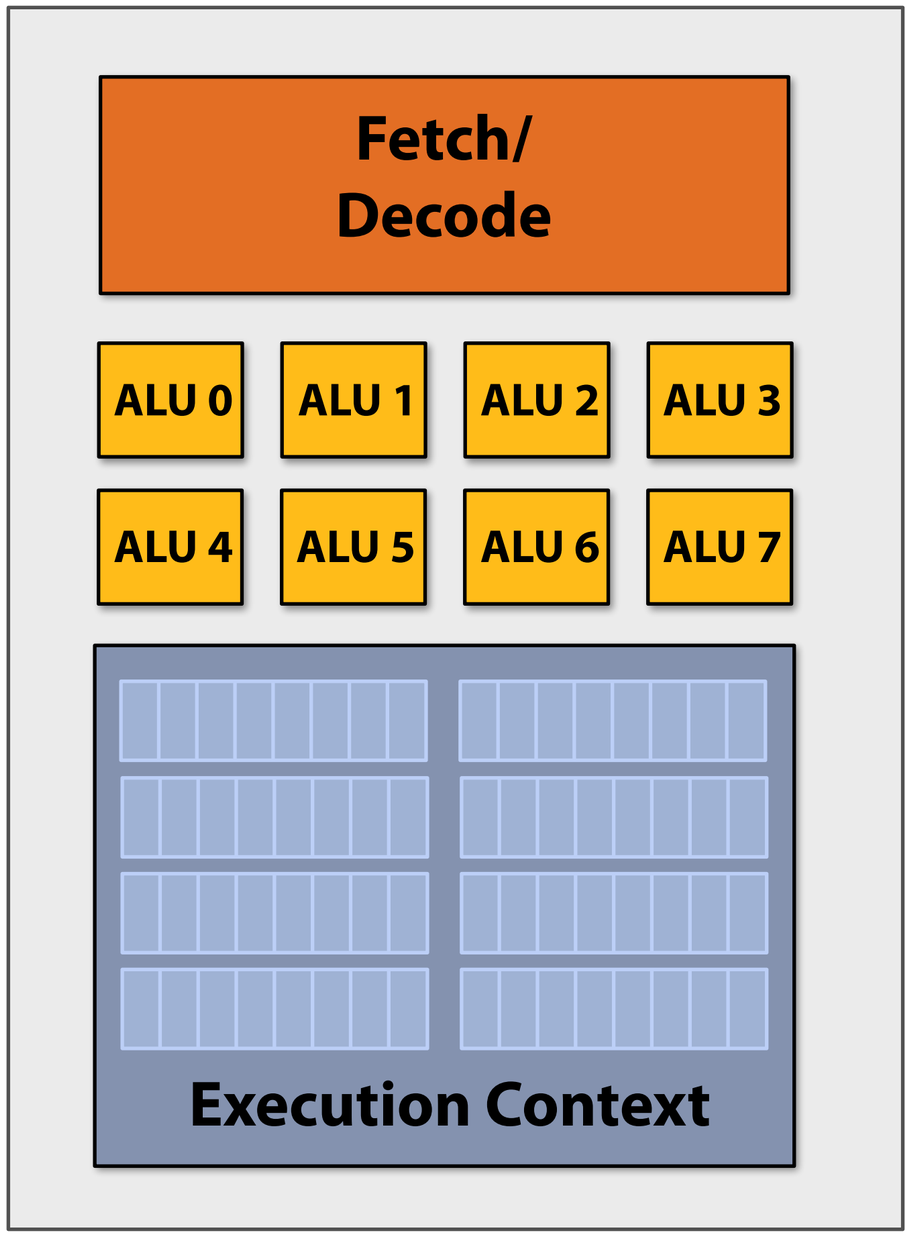


Apple A9 dual-core CPU
(2015)

A9 image credit: Chipworks (obtained via Anandtech)
http://www.anandtech.com/show/9686/the-apple-iphone-6s-and-iphone-6s-plus-review/3

SIMD processing

Add ALUs to increase compute capability



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

SIMD processing

Single instruction, multiple data

Same instruction broadcast to all ALUs
Executed in parallel on all ALUs

Scalar program

```
void sinx(int N, int terms, float* x, float* result)
{
   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;
      }

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

Original compiled program:

Processes one array element using scalar instructions on scalar registers (e.g., 32-bit floats)

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

Vector program (using AVX intrinsics)

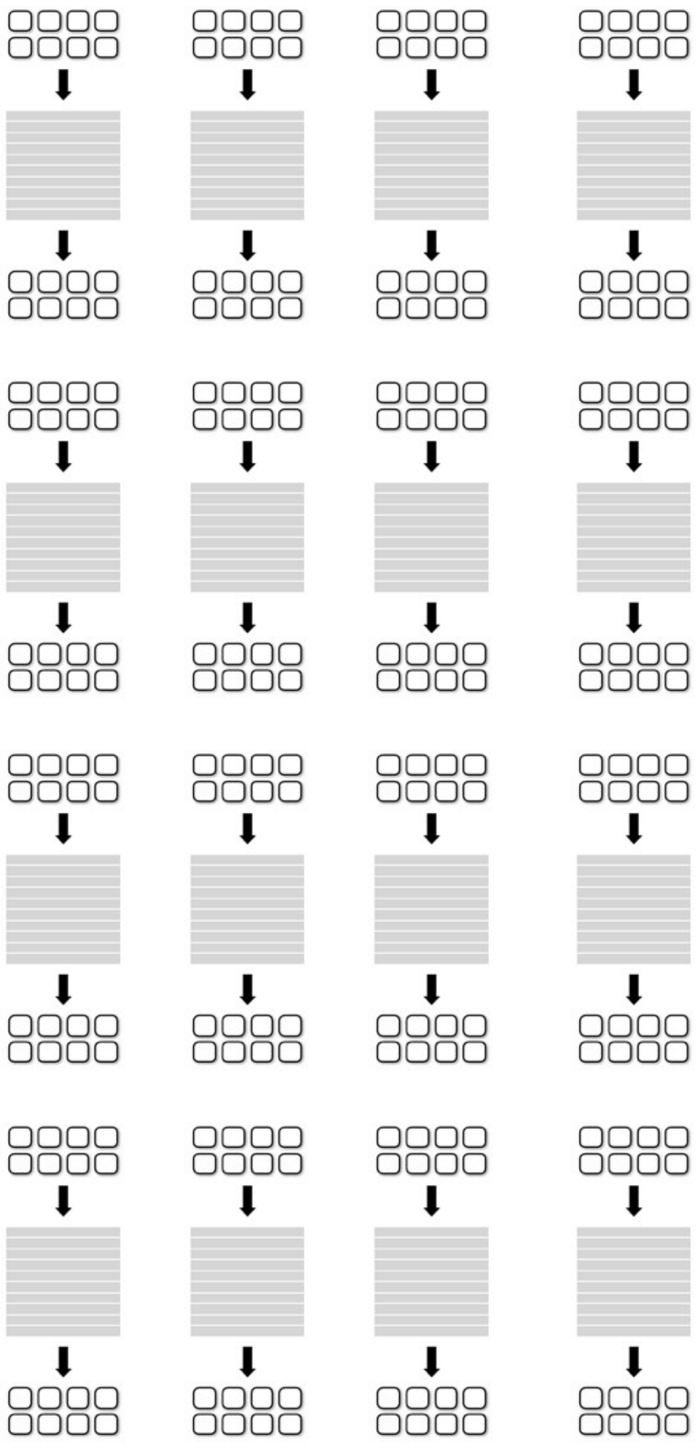
```
#include <immintrin.h>
void sinx(int N, int terms, float* x, float* sinx)
{
   float three_fact = 6;  // 3!
   for (int i=0; i<N; i+=8)
   {
       __m256 origx = _mm256_load_ps(&x[i]);
      __m256 value = origx;
      __m256 numer = _mm256_mul_ps(origx, _mm256_mul_ps(origx, origx));
      __m256 denom = _mm256_broadcast_ss(&three_fact);
      int sign = -1;

      for (int j=1; j<=terms; j++)
      {
         // value += sign * numer / denom
         __m256 tmp =
                _mm256_div_ps(_mm256_mul_ps(_mm256_broadcast_ss(sign),numer),denom);
         value = _mm256_add_ps(value, tmp);

         numer = _mm256_mul_ps(numer, _mm256_mul_ps(origx, origx));
         denom = _mm256_mul_ps(denom, _mm256_broadcast_ss((2*j+2) * (2*j+3)));
         sign *= -1;
      }
      _mm256_store_ps(&sinx[i], value);
   }
}
```

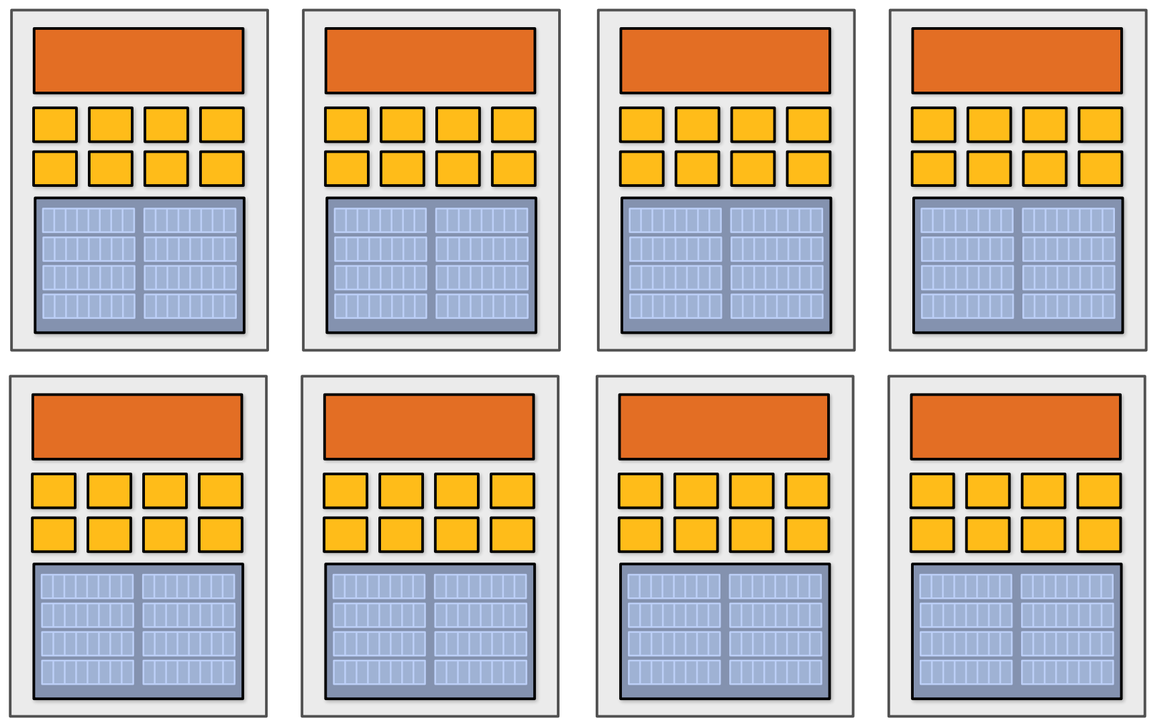
```
vloadps  xmm0, addr[r1]
vmulps   xmm1, xmm0, xmm0
vmulps   xmm1, xmm1, xmm0
...
...
...
...
...
...
vstoreps  addr[xmm2], xmm0
```

Compiled program:

Processes eight array elements simultaneously using vector instructions on 256-bit vector registers

16 SIMD cores: 128 elements in parallel





16 cores, 128 ALUs, 16 simultaneous instruction streams

Data-parallel expression

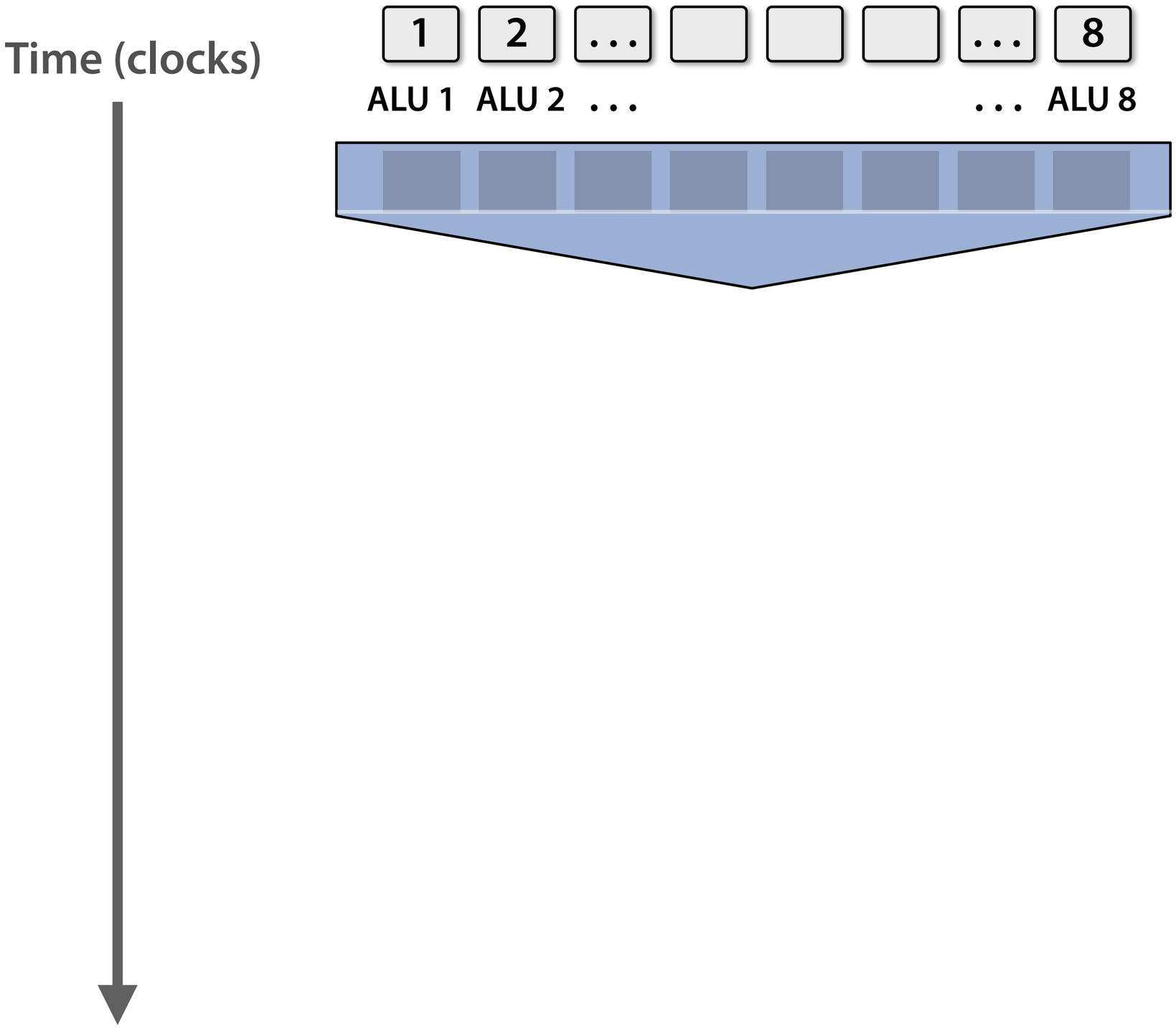
(in Kayvon's fictitious data-parallel language)

```
void sinx(int N, int terms, float* x, float* result)
{
   // declare independent loop iterations
   forall (int i from 0 to N-1)
   {
      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;
      }

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

Compiler understands loop iterations are independent, and that same loop body will be executed on a large number of data elements.

Abstraction facilitates automatic generation of both multi-core parallel code, and vector instructions to make use of SIMD processing capabilities within a core.

What about conditional execution?

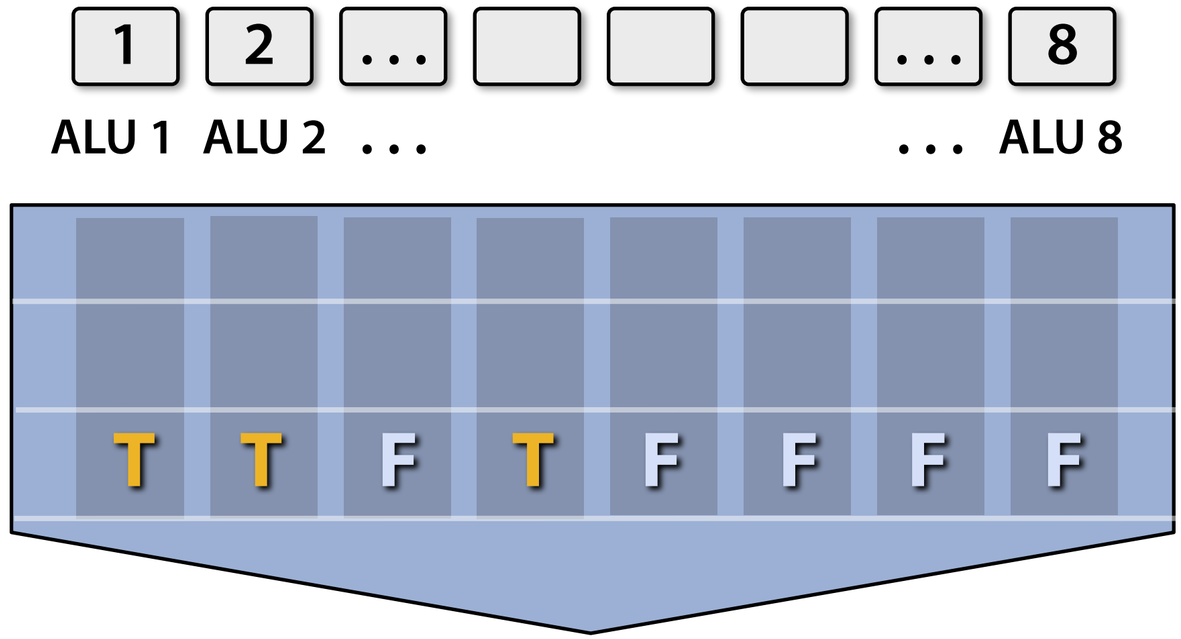


(assume logic below is to be executed for each element in input array 'A', producing output into the array 'result')

```
<unconditional code>

float x = A[i];

if (x > 0) {
   float tmp = exp(x,5.f);
   tmp *= kMyConst1;
   x = tmp + kMyConst2;
} else {
   float tmp = kMyConst1;
   x = 2.f * tmp;
}

<resume unconditional code>


result[i] = x;
```

What about conditional execution?

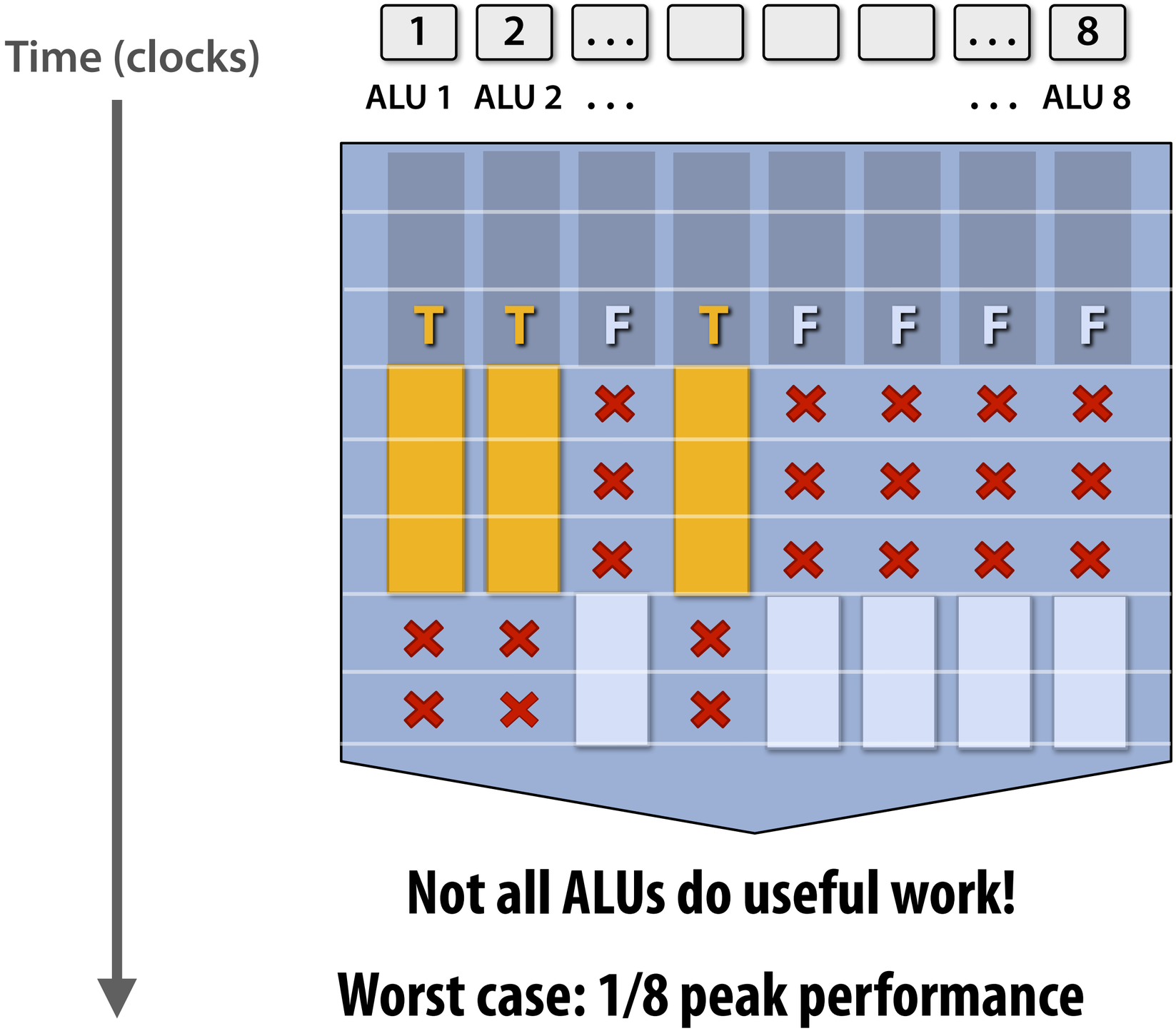
Time (clocks)



(assume logic below is to be executed for each element in input array 'A', producing output into the array 'result')

```
<unconditional code>

float x = A[i];

if (x > 0) {
   float tmp = exp(x,5.f);
   tmp *= kMyConst1;
   x = tmp + kMyConst2;
} else {
   float tmp = kMyConst1;
   x = 2.f * tmp;
}

<resume unconditional code>


result[i] = x;
```

Mask (discard) output of ALU



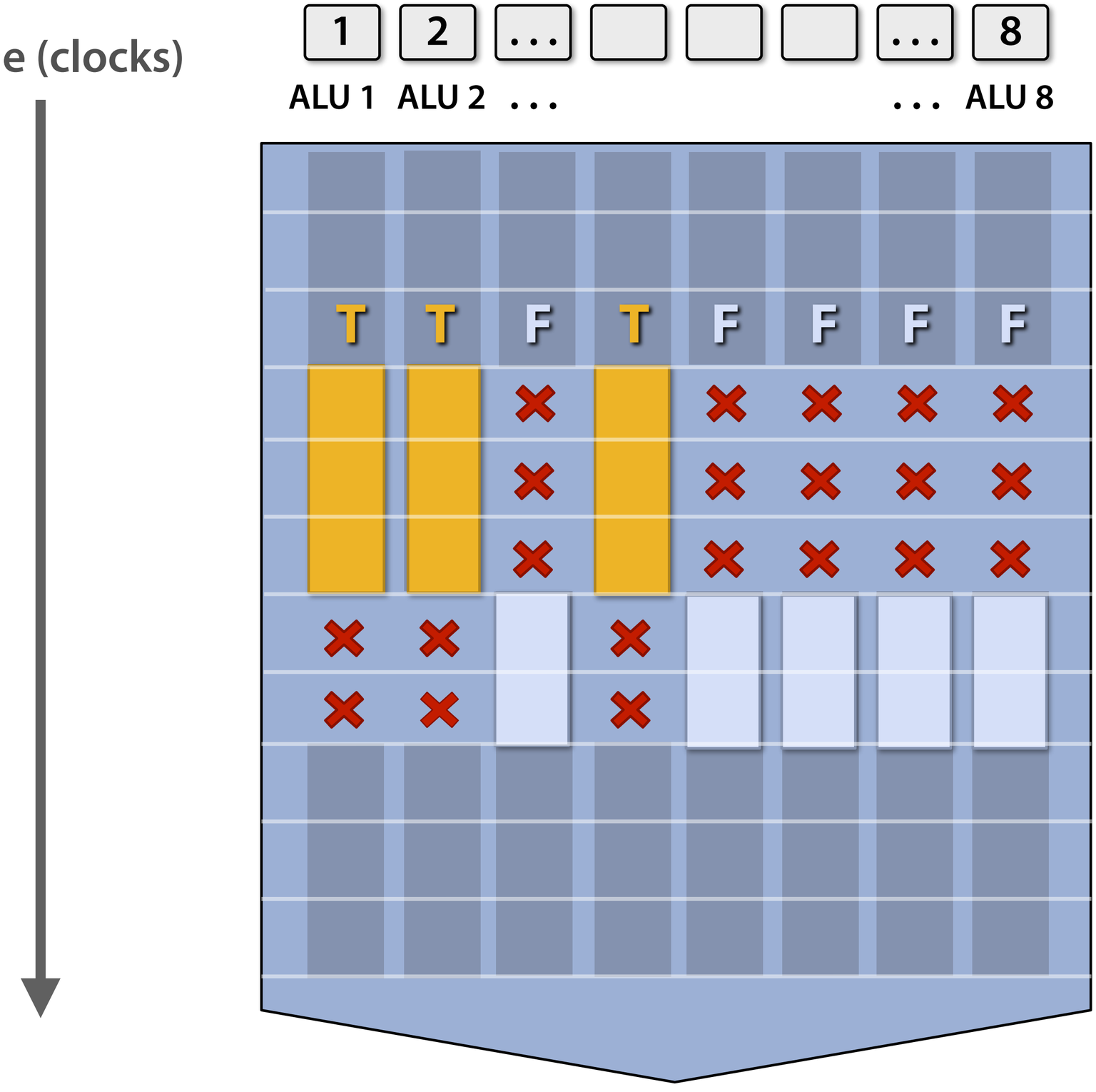
Not all ALUs do useful work!

Worst case: 1/8 peak performance

(assume logic below is to be executed for each element in input array 'A', producing output into the array 'result')

```
<unconditional code>

float x = A[i];

if (x > 0) {
   float tmp = exp(x,5.f);
   tmp *= kMyConst1;
   x = tmp + kMyConst2;
} else {
   float tmp = kMyConst1;
   x = 2.f * tmp;
}

<resume unconditional
code>

result[i] = x;
```

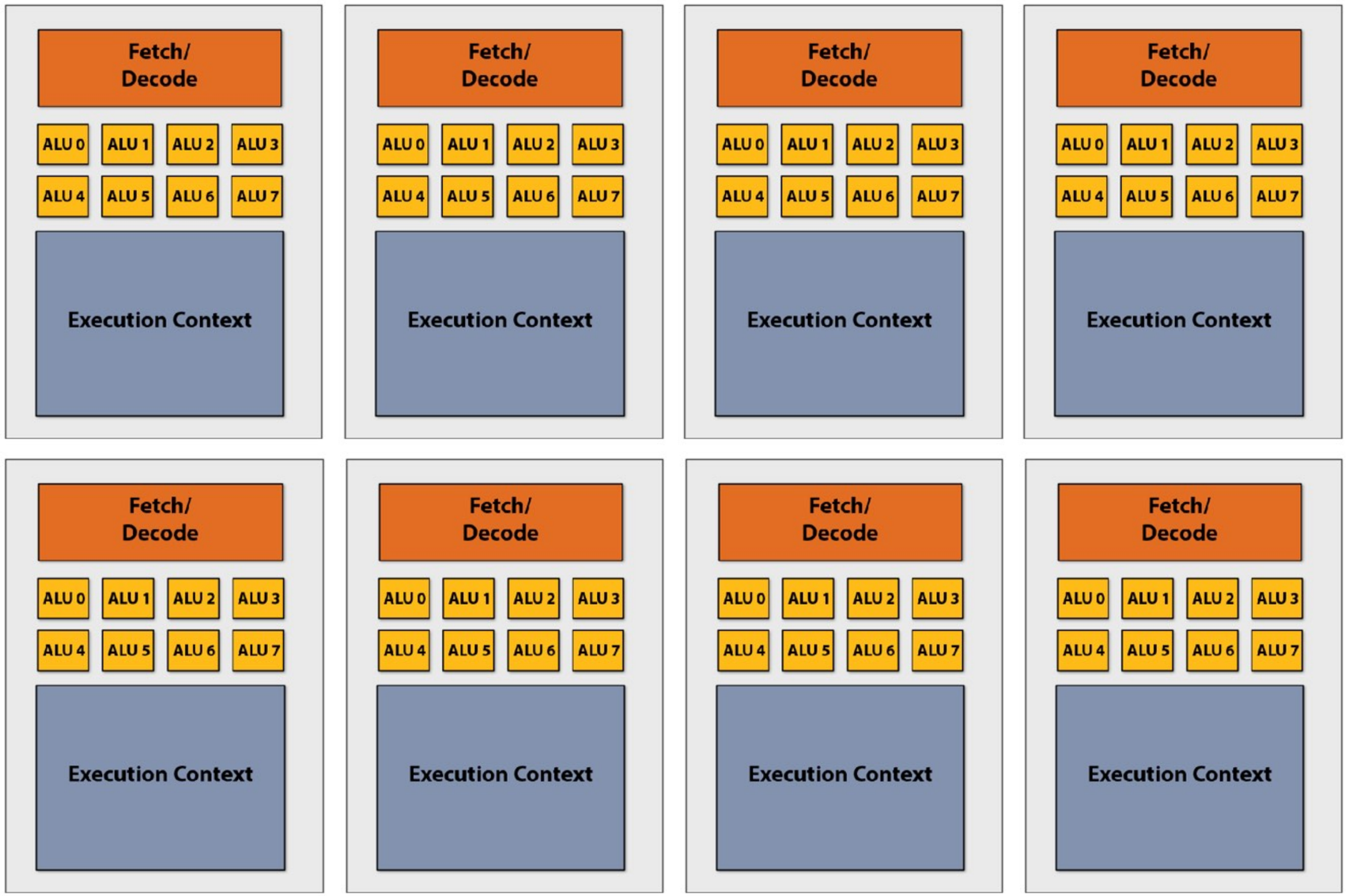
After branch: continue at full performance



(assume logic below is to be executed for each element in input array 'A', producing output into the array 'result')

```
<unconditional code>

float x = A[i];

if (x > 0) {
   float tmp = exp(x,5.f);
   tmp *= kMyConst1;
   x = tmp + kMyConst2;
} else {
   float tmp = kMyConst1;
   x = 2.f * tmp;
}

<resume unconditional code>

result[i] = x;
```

Example: eight-core Intel Xeon E5-1660 v4



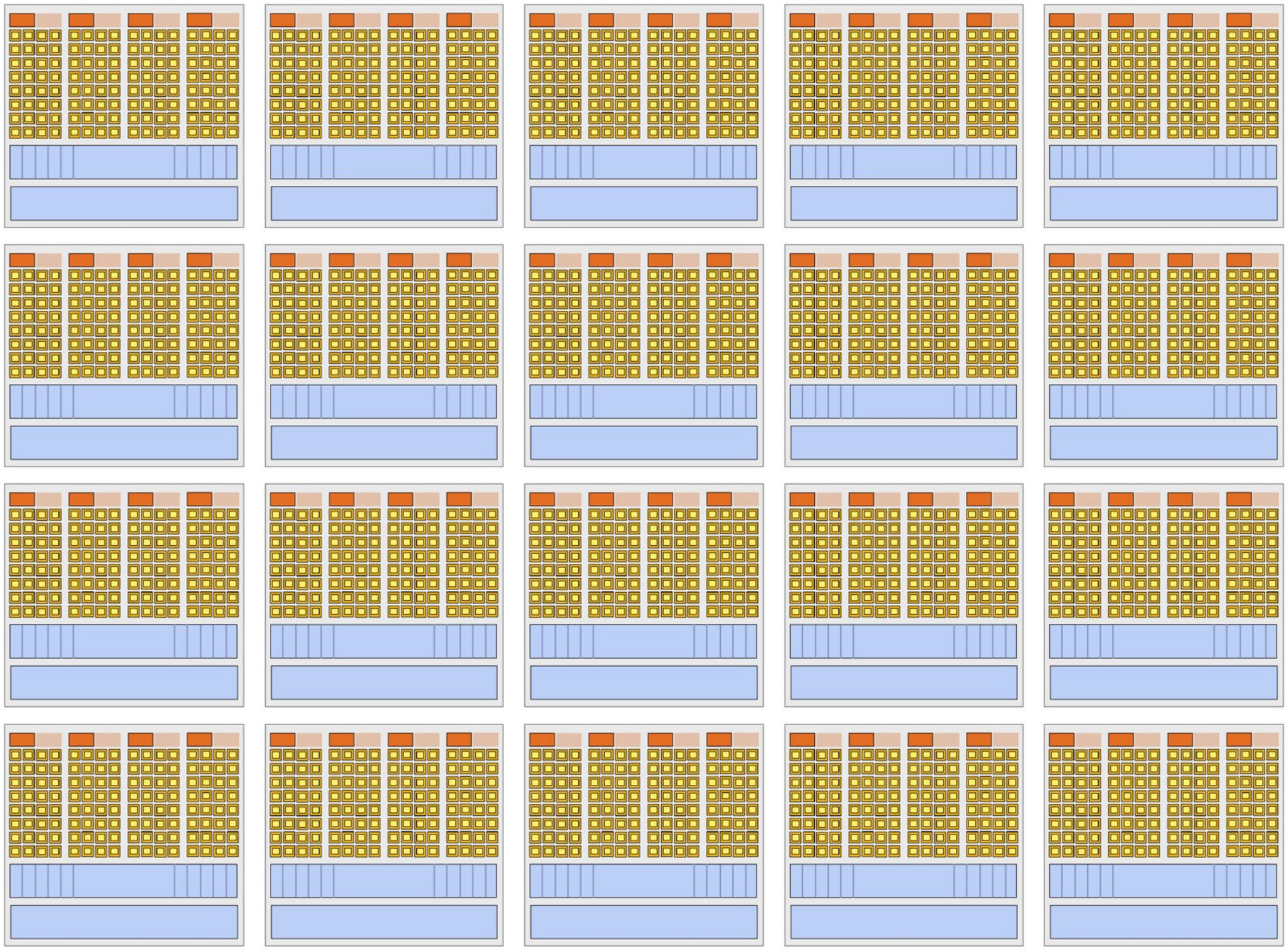
8 cores

8 SIMD ALUs per core (AVX2 instructions)

490 GFLOPs (@3.2 GHz) (140 Watts)

* Showing only AVX math units, and fetch/decode unit for AVX (additional capability for integer math)

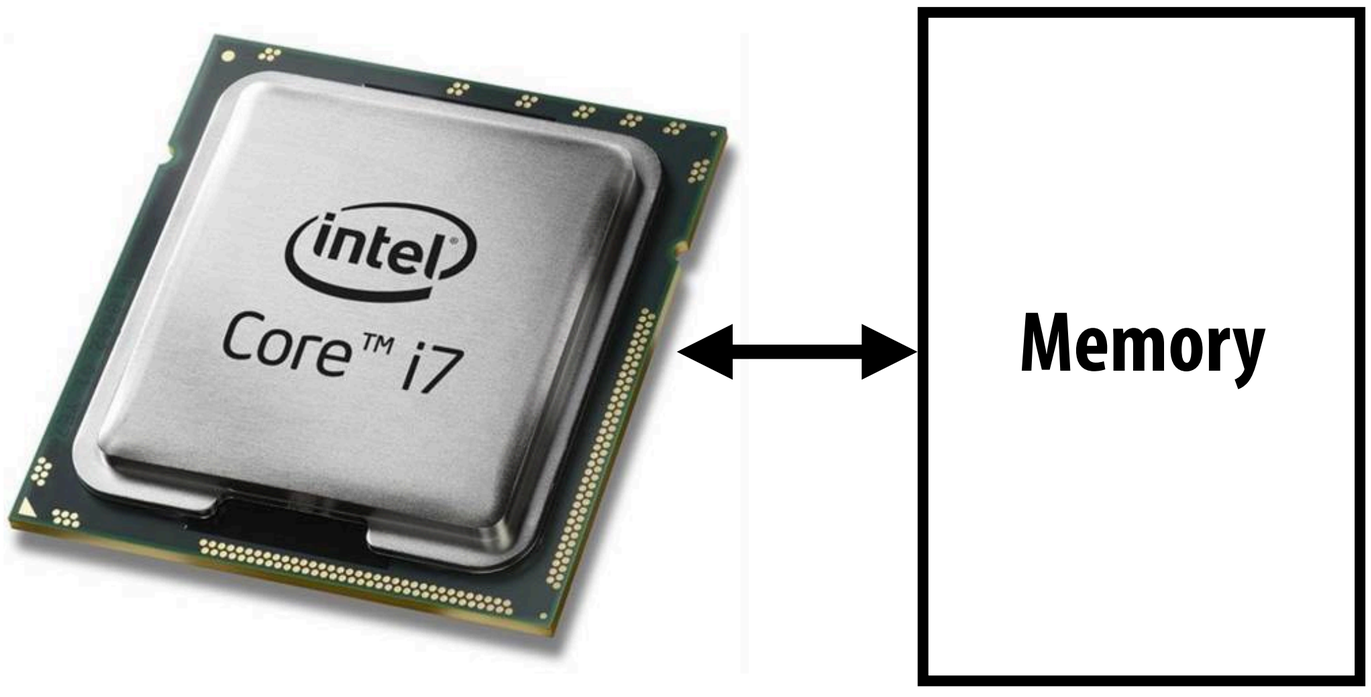
Example: NVIDIA GTX 1080 GPU



20 cores ("SMs")

128 SIMD ALUs per core (@1.6 GHz) = 8.1 TFLOPs (180 Watts)

Part 2:
accessing memory



Hardware multi-threading

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

Stalls

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

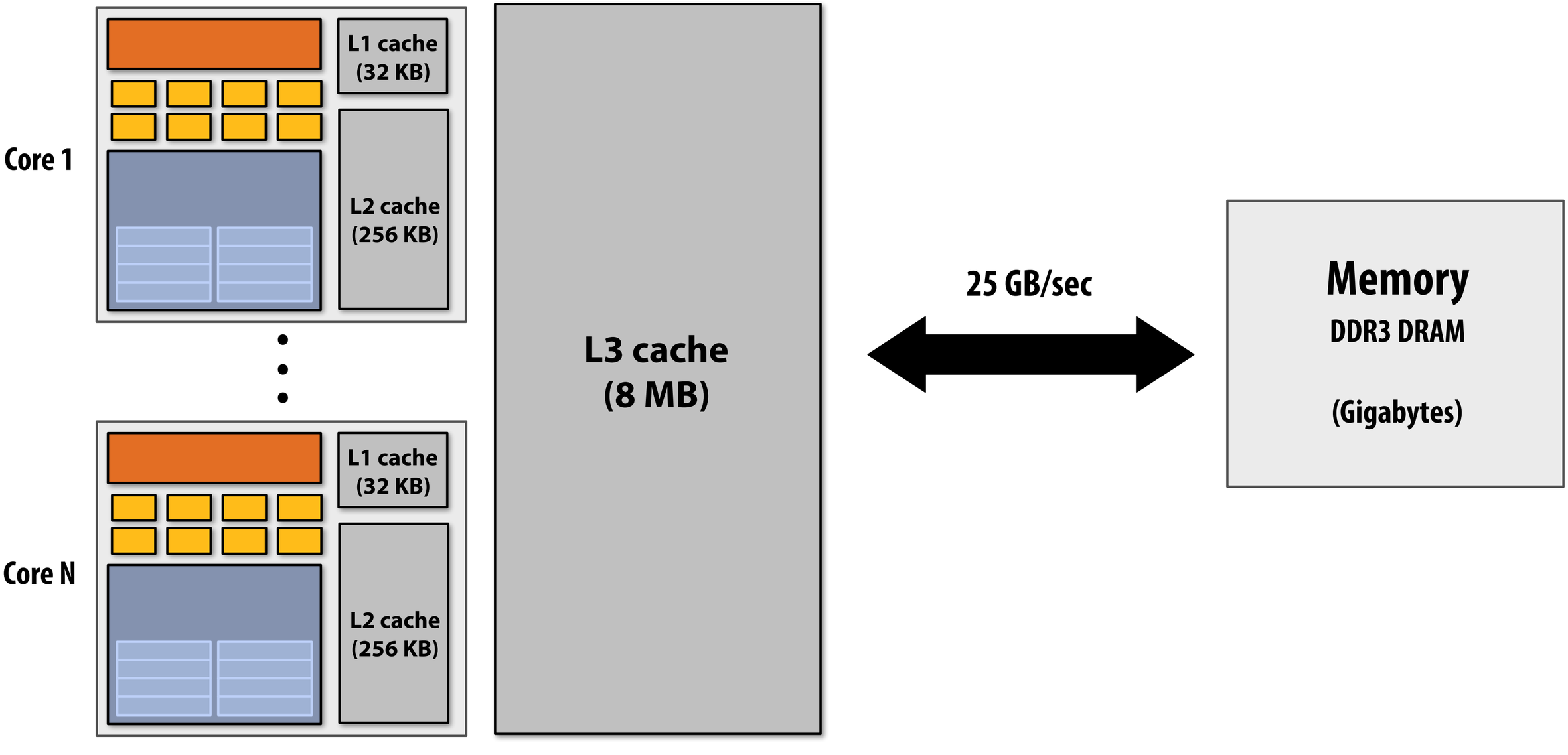
- Accessing memory is a major source of stalls

```
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

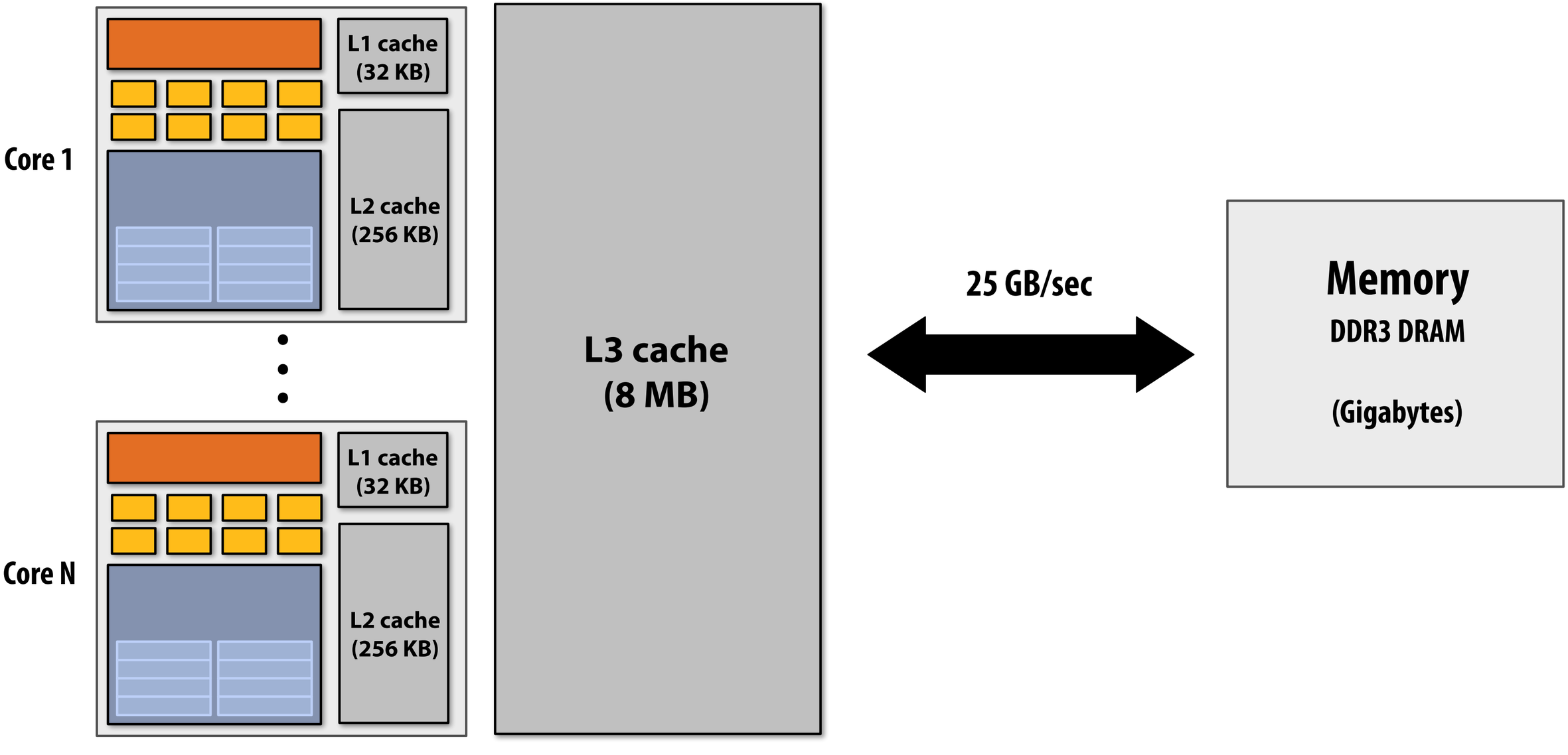
- Memory access times ~ 100's of cycles
  - Memory "access time" is a measure of latency

Review: why do modern processors have caches?



Caches reduce length of stalls (reduce latency)

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



*** Caches also provide high bandwidth data transfer to CPU**

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
```

data arrives in cache

data arrives in cache

These loads are cache hits

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

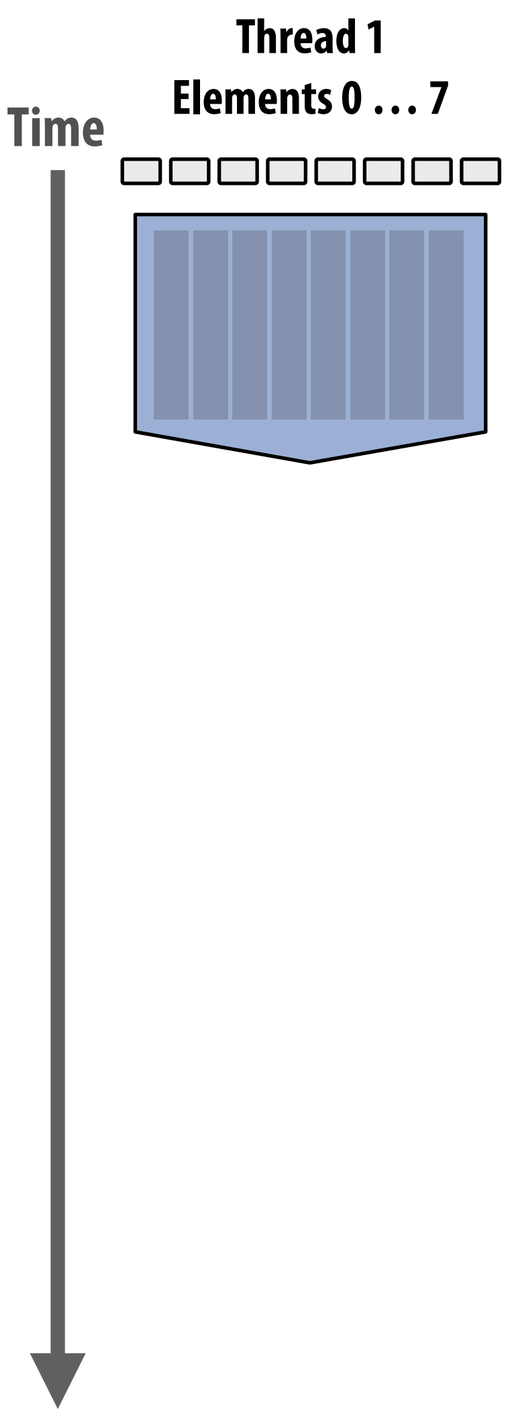
(more detail later in course)

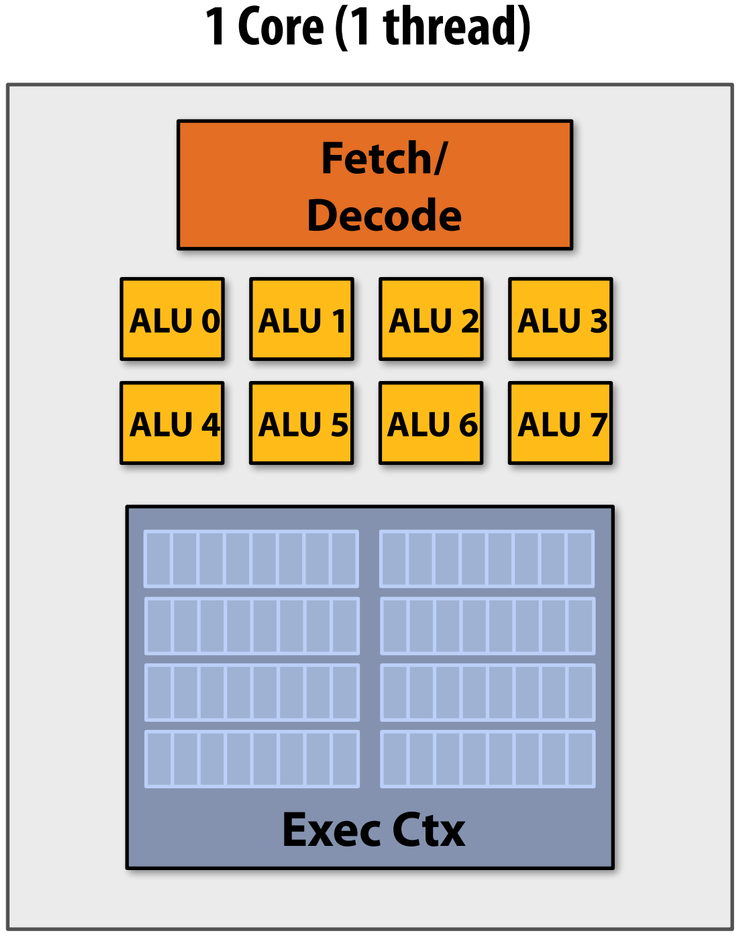
Multi-threading reduces stalls

- Idea: interleave processing of multiple threads on the same core to hide stalls

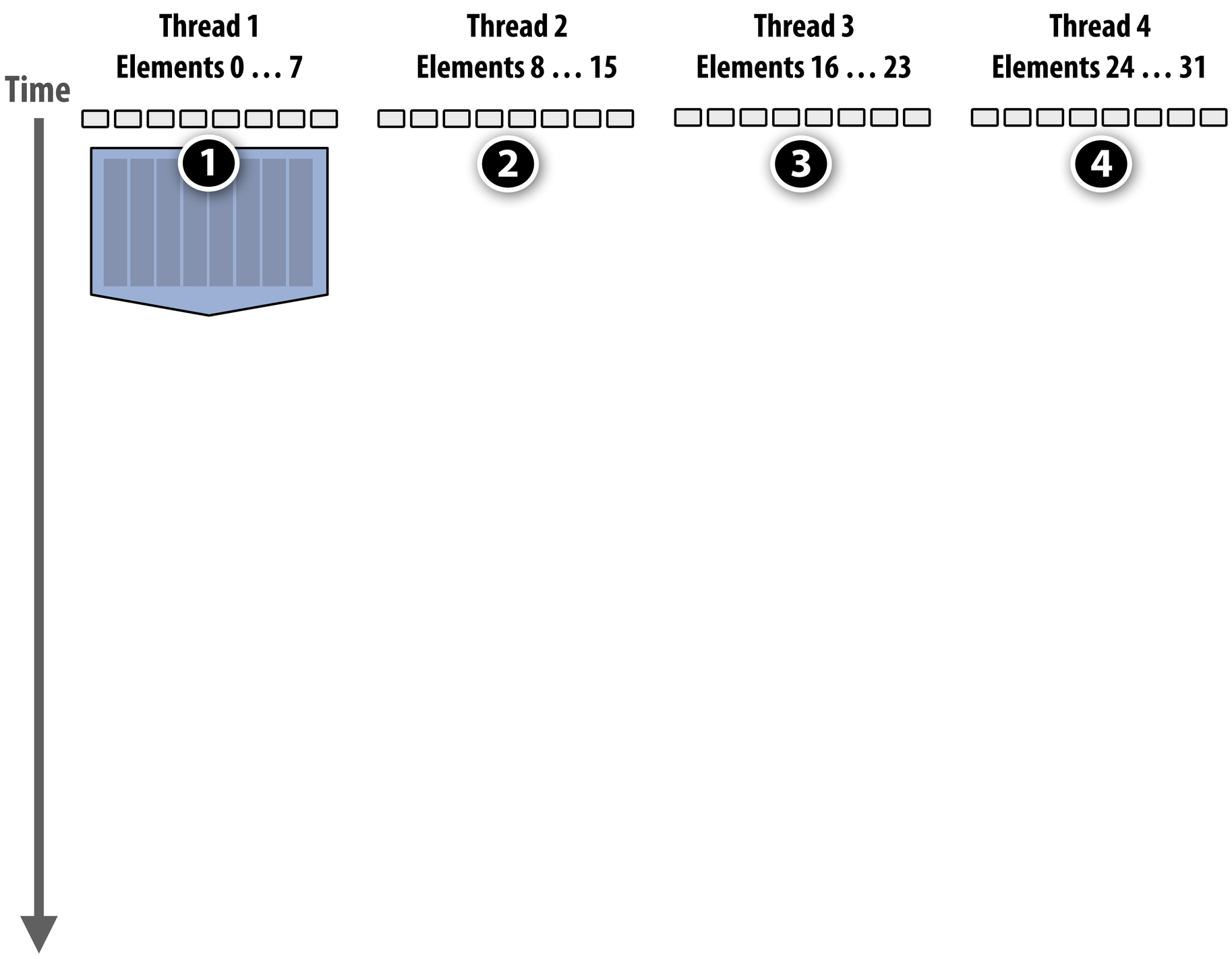
- Like prefetching, multi-threading is a latency hiding, not a latency reducing technique

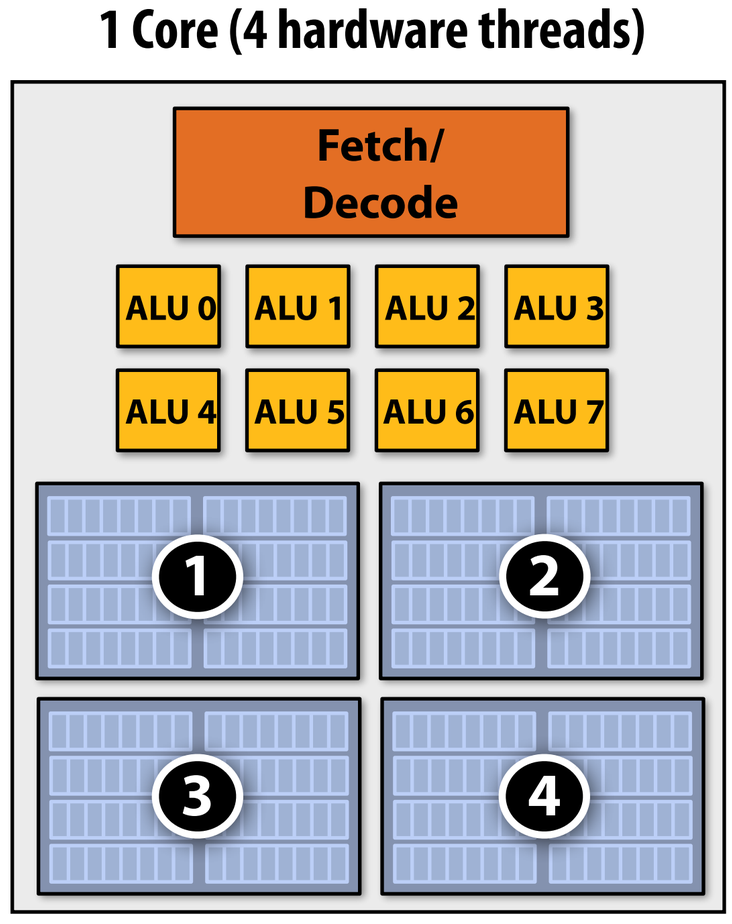
Hiding stalls with multi-threading



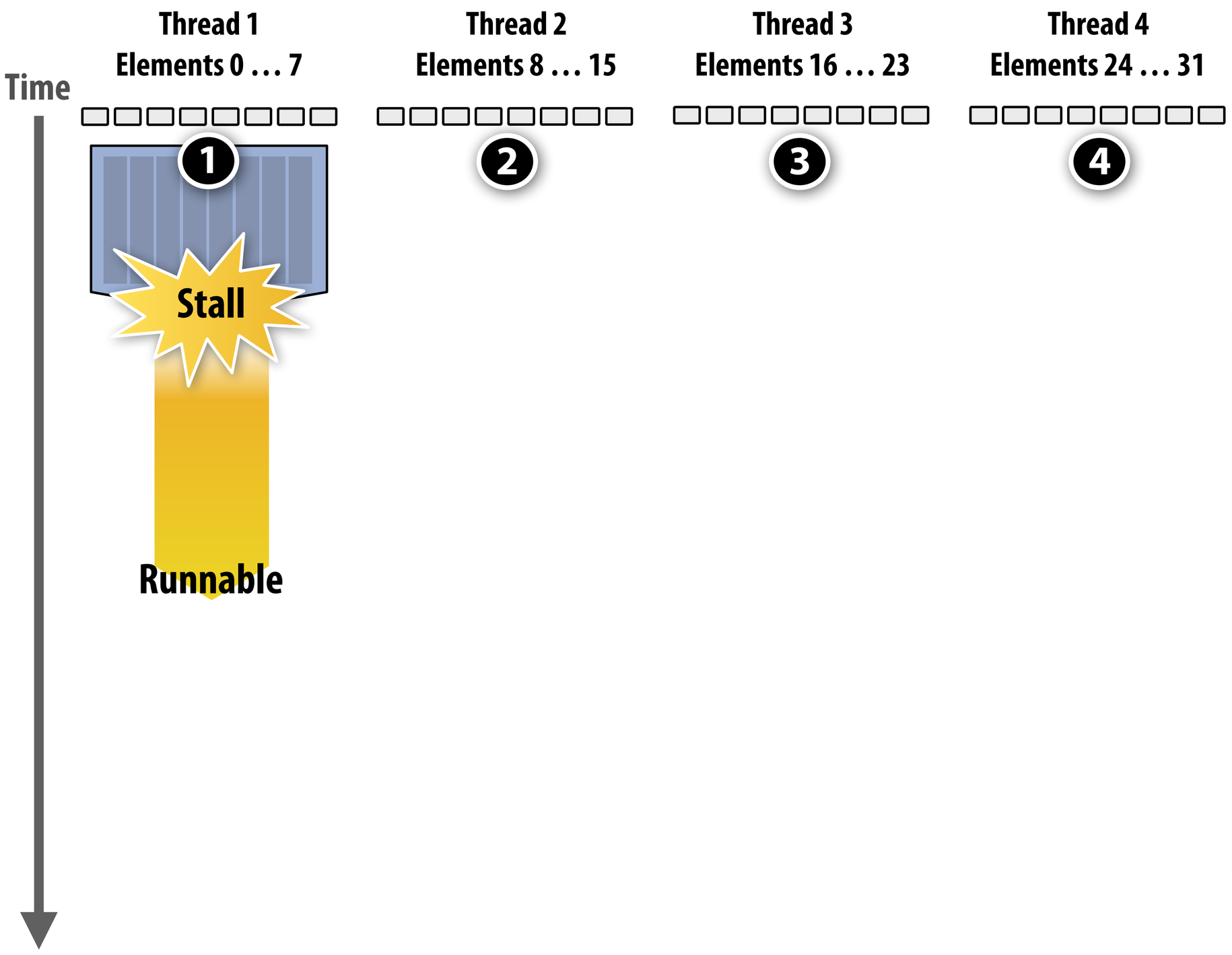


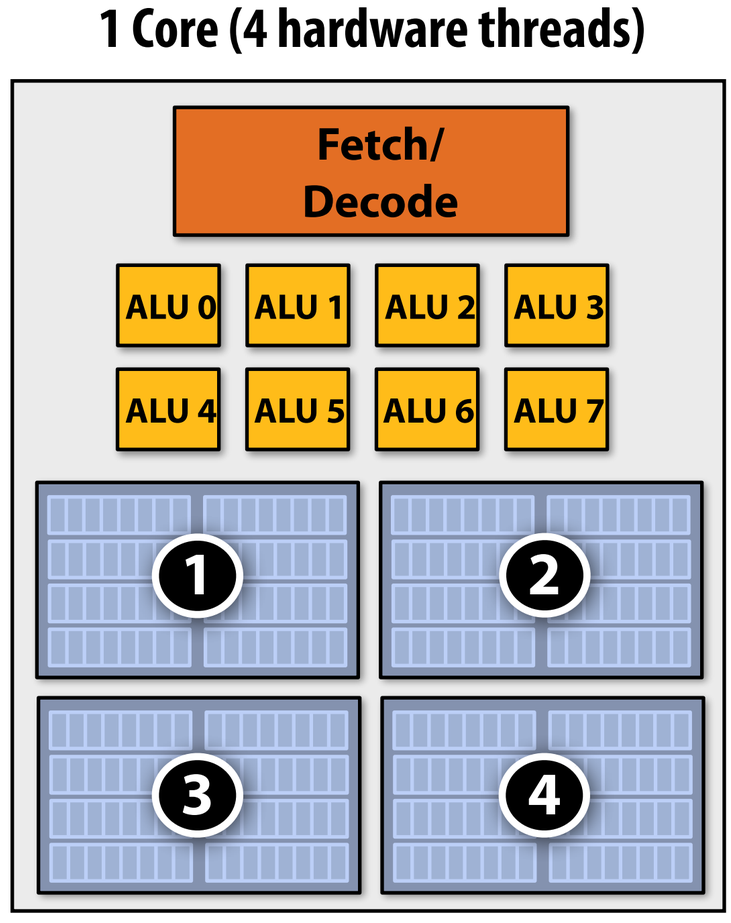
Hiding stalls with multi-threading





Hiding stalls with multi-threading





Hiding stalls with multi-threading

Thread 1
Elements 0 . . . 7

Thread 2
Elements 8 . . . 15

Thread 3
Elements 16 . . . 23

Thread 4
Elements 24 . . . 31

Time

1

Stall

Runnable

Done!

2

Stall

Runnable

Done!

3

Stall

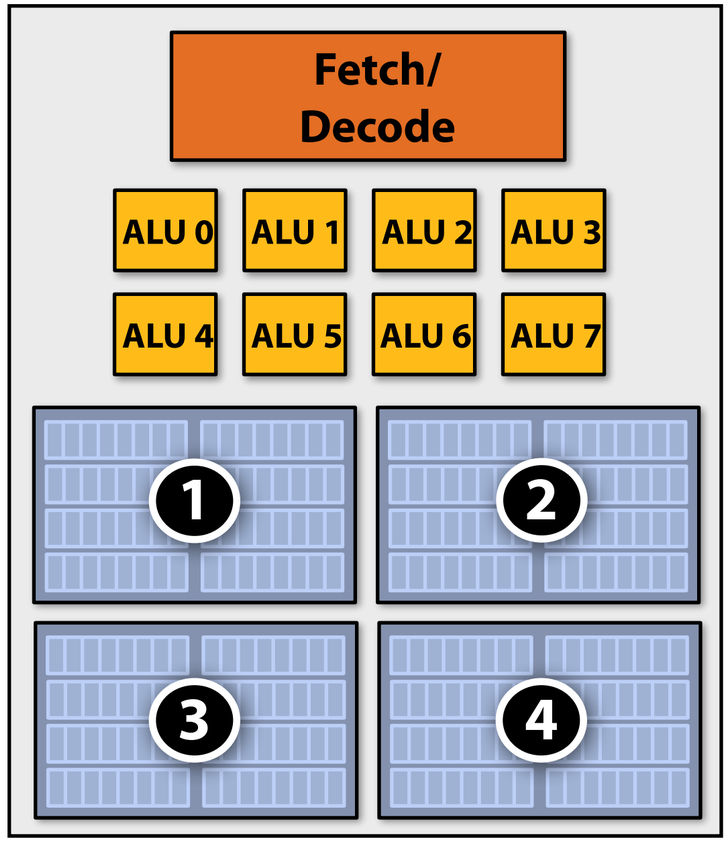
Runnable

4

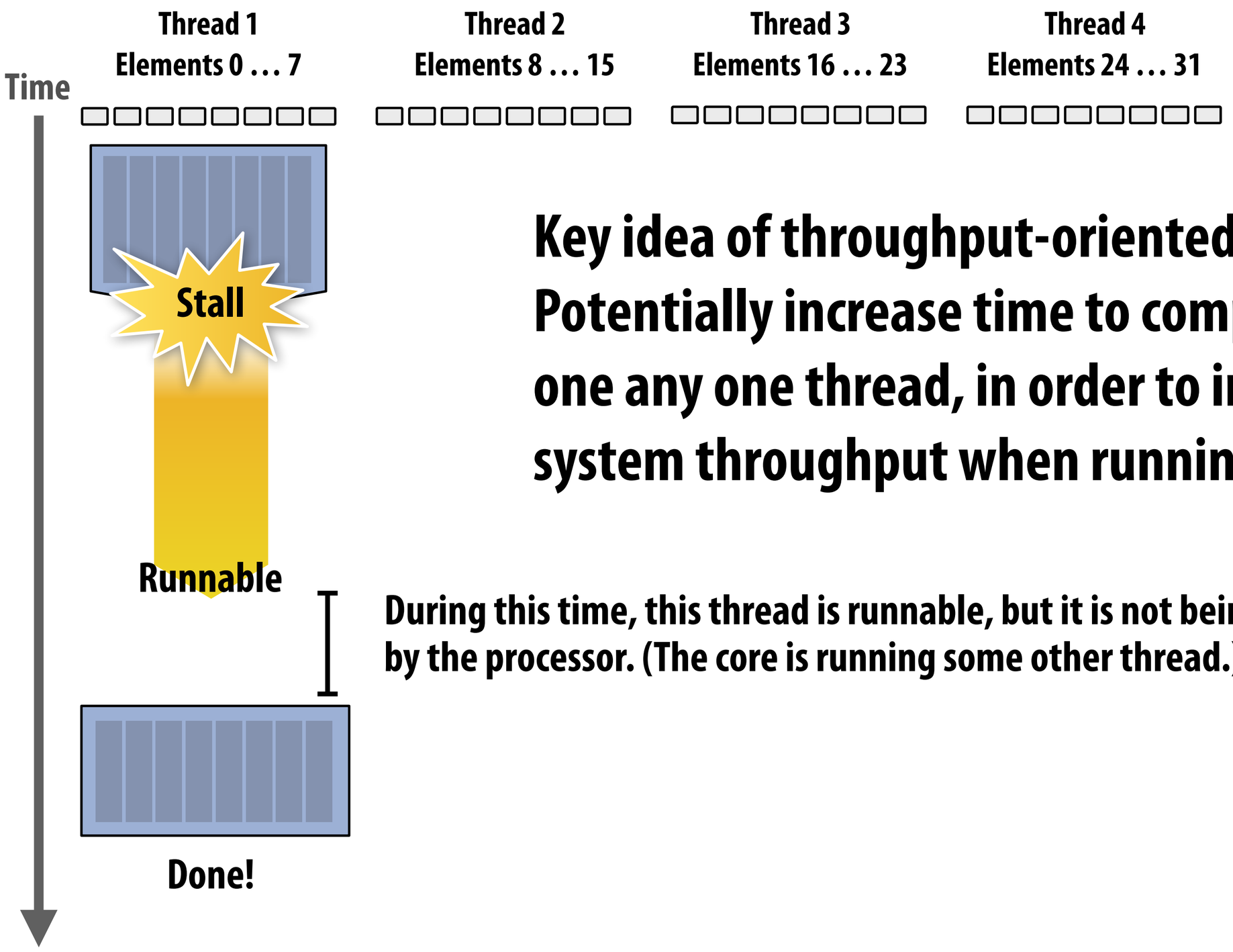
Stall

Runnable

1 Core (4 hardware threads)



Throughput computing trade-off



Key idea of throughput-oriented systems: Potentially increase time to complete work by any one 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.)

Kayvon's fictitious multi-core chip

16 cores

8 SIMD ALUs per core

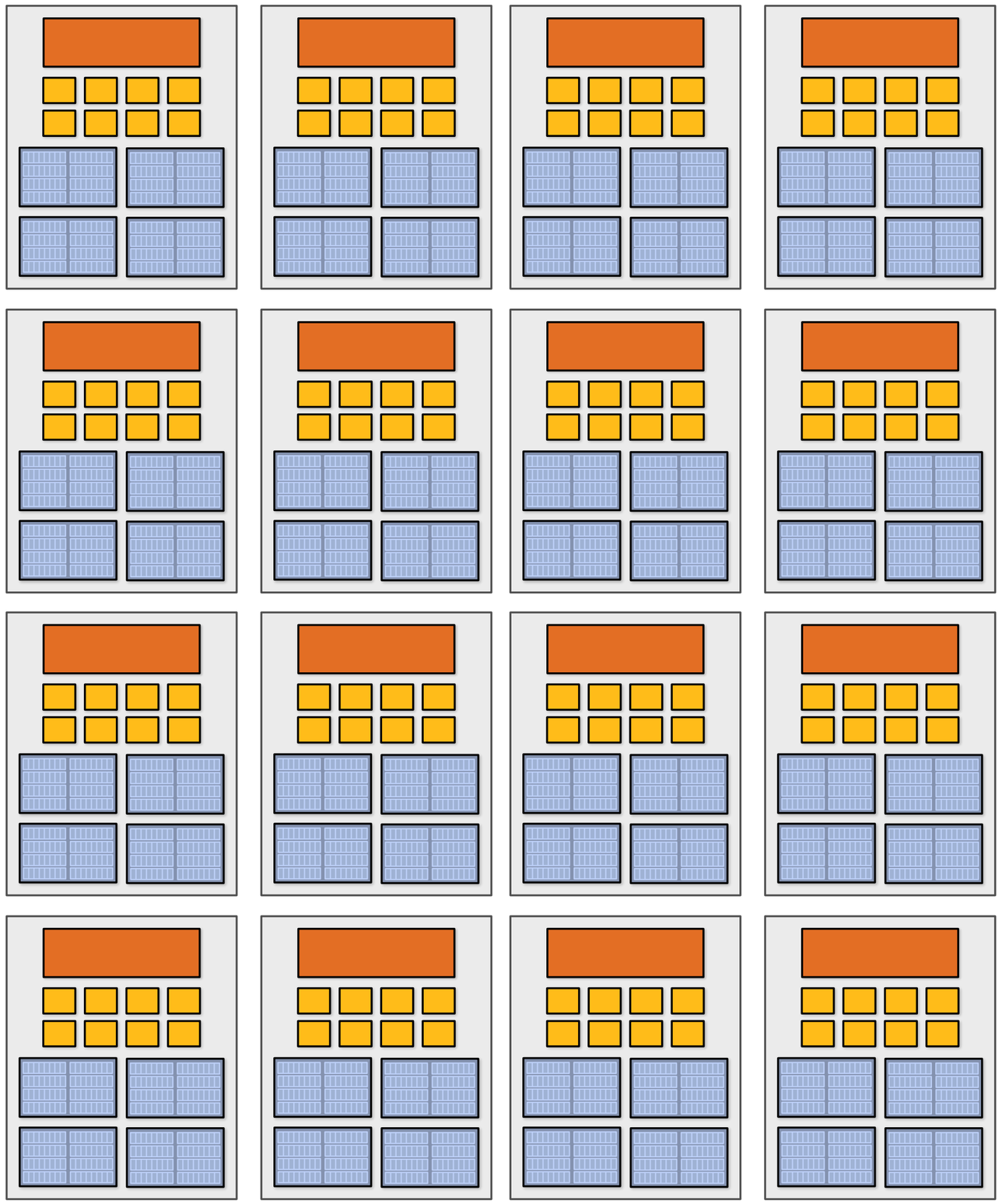
(128 total)

4 threads per core

16 simultaneous instruction streams

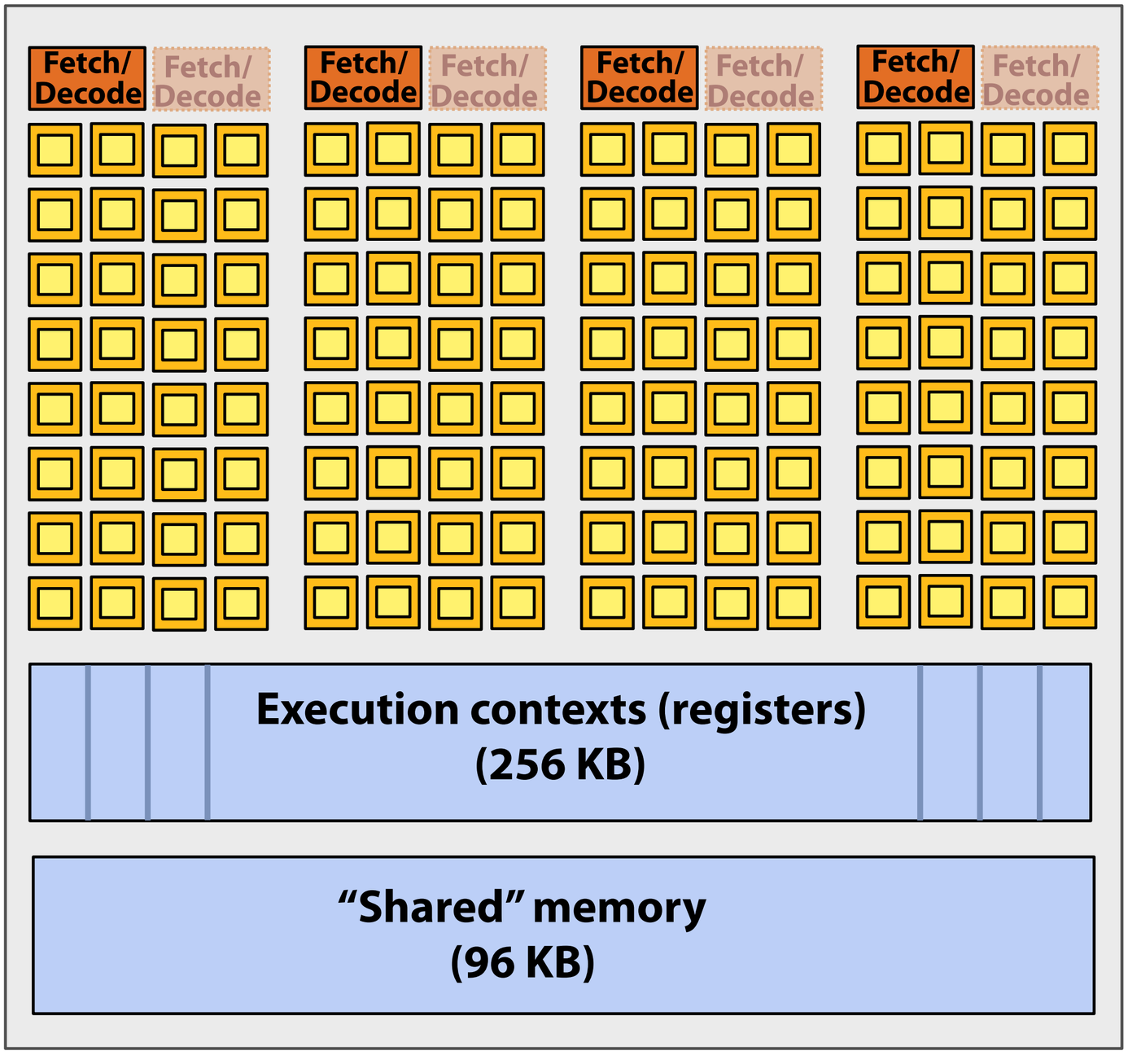
64 total concurrent instruction streams

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")

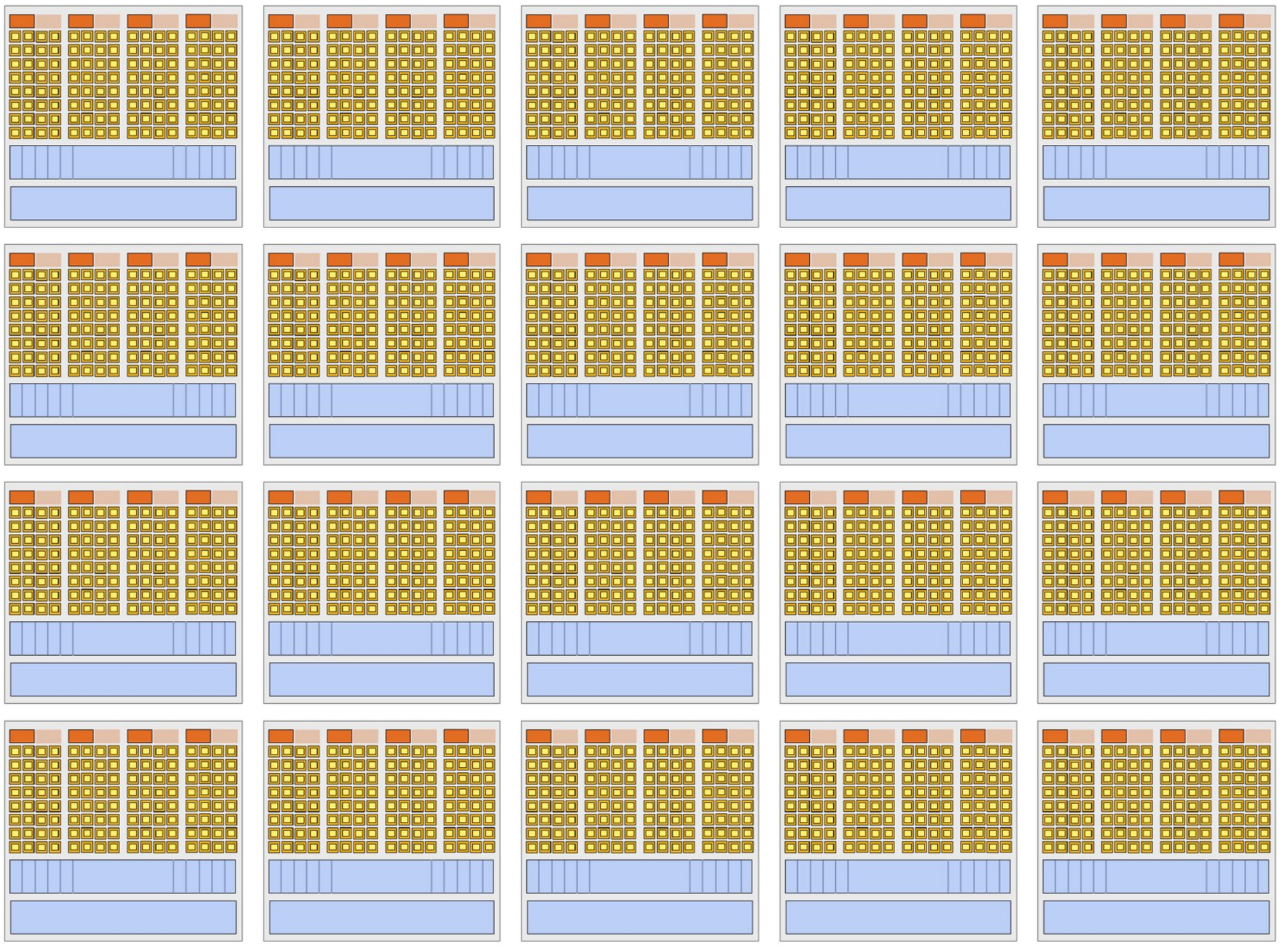


= SIMD function unit,
control shared across 32 units
(1 MUL-ADD per clock)

- 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!

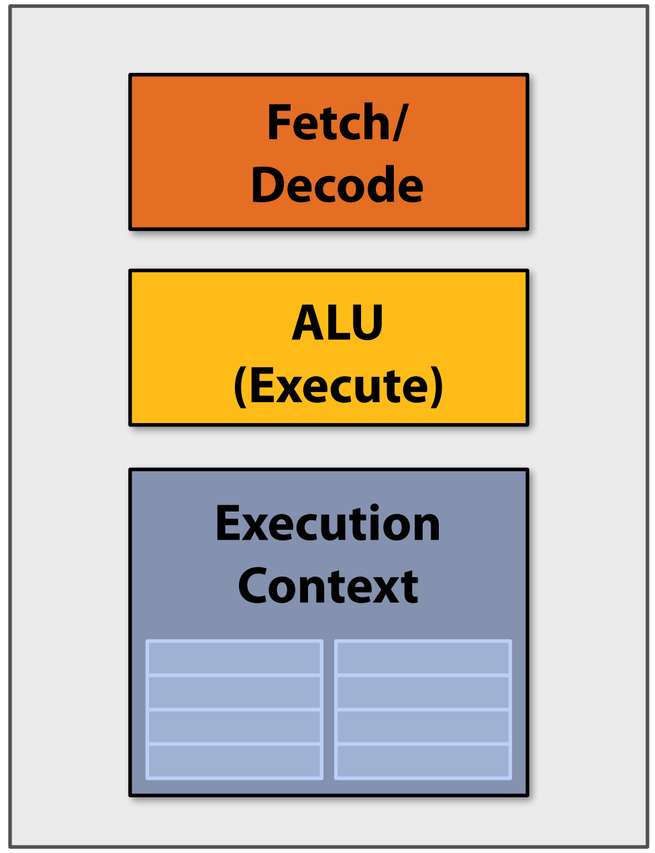
Another example:
for review and to check your understanding
(if you understand the following sequence you understand this lecture)

Running code on a simple processor

My very simple program: compute $\sin(x)$ using Taylor expansion

```
void sinx(int N, int terms, float* x, float* result)
{
   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;
      }

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

My very simple processor: completes one instruction per clock



Review: superscalar execution

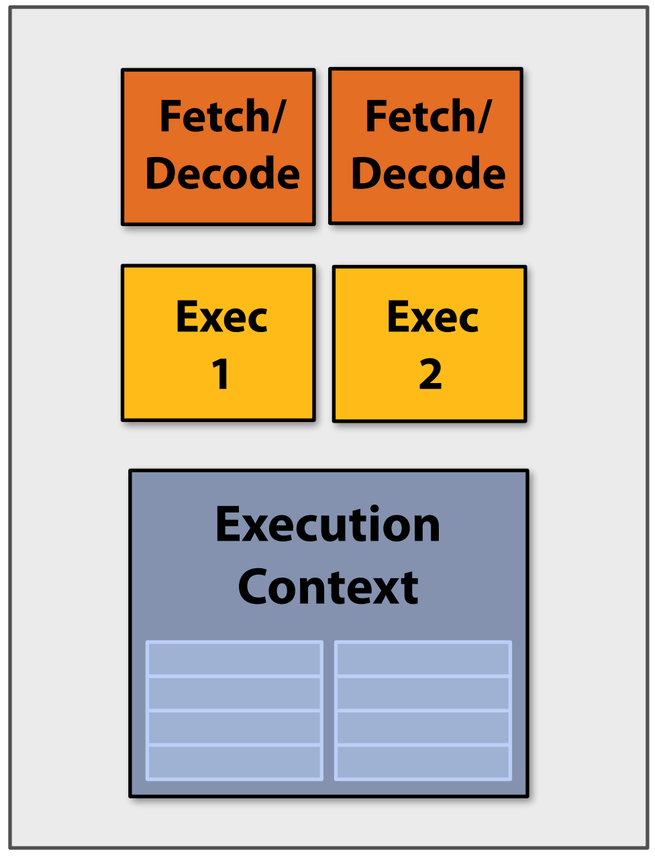
Unmodified program

```
void sinx(int N, int terms, float* x, float* result)
{
   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;
      }

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

Independent operations in instruction stream

(They are detected by the processor at run-time and may be executed in parallel on execution units 1 and 2)

My single core, superscalar processor: executes up to two instructions per clock from a single instruction stream.



Review: multi-core execution (two cores)

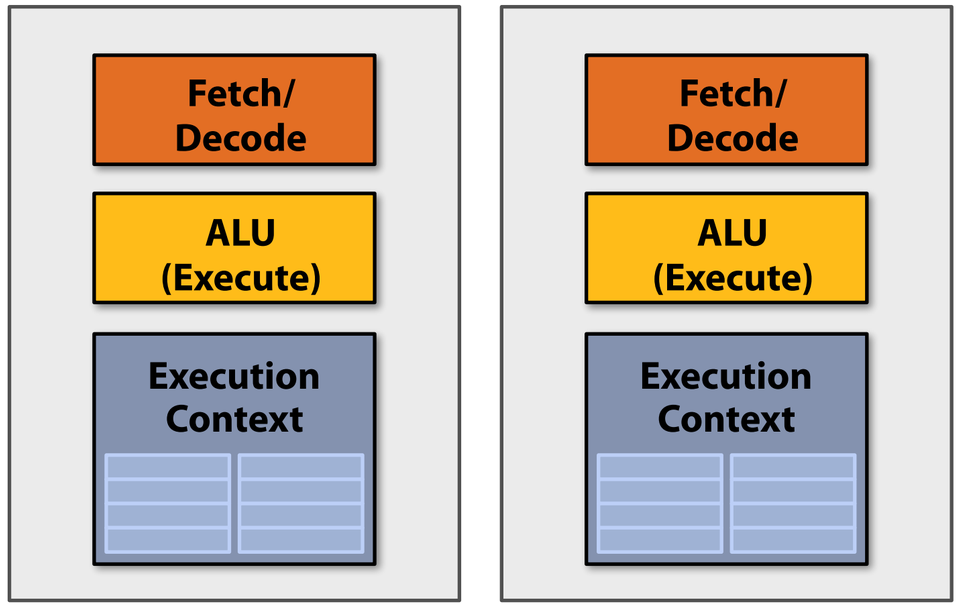
Modify program to create two threads of control (two instruction streams)

```
typedef struct {
   int N;
   int terms;
   float* x;
   float* result;
} my_args;

void parallel_sinx(int N, int terms, float* x, float* result) {
    pthread_t thread_id;
    my_args args;

    args.N = N/2;
    args.terms = terms;
    args.x = x;
    args.result = result;

    // launch thread
    pthread_create(&thread_id, NULL, my_thread_start, &args);
    sinx(N - args.N, terms, x + args.N, result + args.N); // do work
    pthread_join(thread_id, NULL);
}

void my_thread_start(void* thread_arg) {
   my_args* thread_args = (my_args*)thread_arg;
   sinx(args->N, args->terms, args->x, args->result); // do work
}
```

My dual-core processor:

executes one instruction per clock

from an instruction stream on <u>each</u> core.

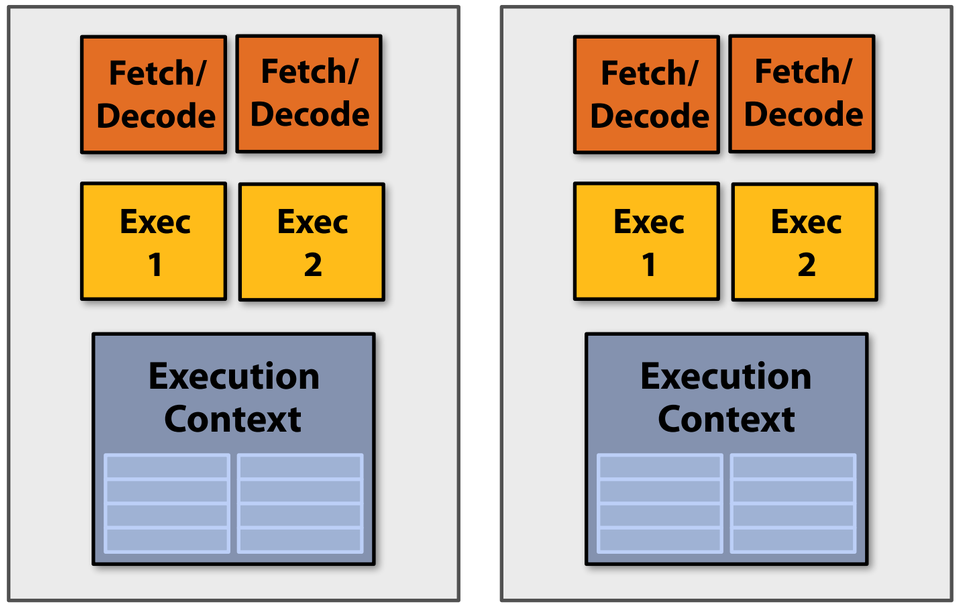


Review: multi-core + superscalar execution

Modify program to create two threads of control (two instruction streams)

```
typedef struct {
   int N;
   int terms;
   float* x;
   float* result;
} my_args;

void parallel_sinx(int N, int terms, float* x, float* result) {
    pthread_t thread_id;
    my_args args;

    args.N = N/2;
    args.terms = terms;
    args.x = x;
    args.result = result;

    // launch thread
    pthread_create(&thread_id, NULL, my_thread_start, &args);
    sinx(N - args.N, terms, x + args.N, result + args.N); // do work
    pthread_join(thread_id, NULL);
}

void my_thread_start(void* thread_arg) {
   my_args* thread_args = (my_args*)thread_arg;
   sinx(args->N, args->terms, args->x, args->result); // do work
}
```

My superscalar dual-core processor: executes up to two instructions per clock from an instruction stream on each core.

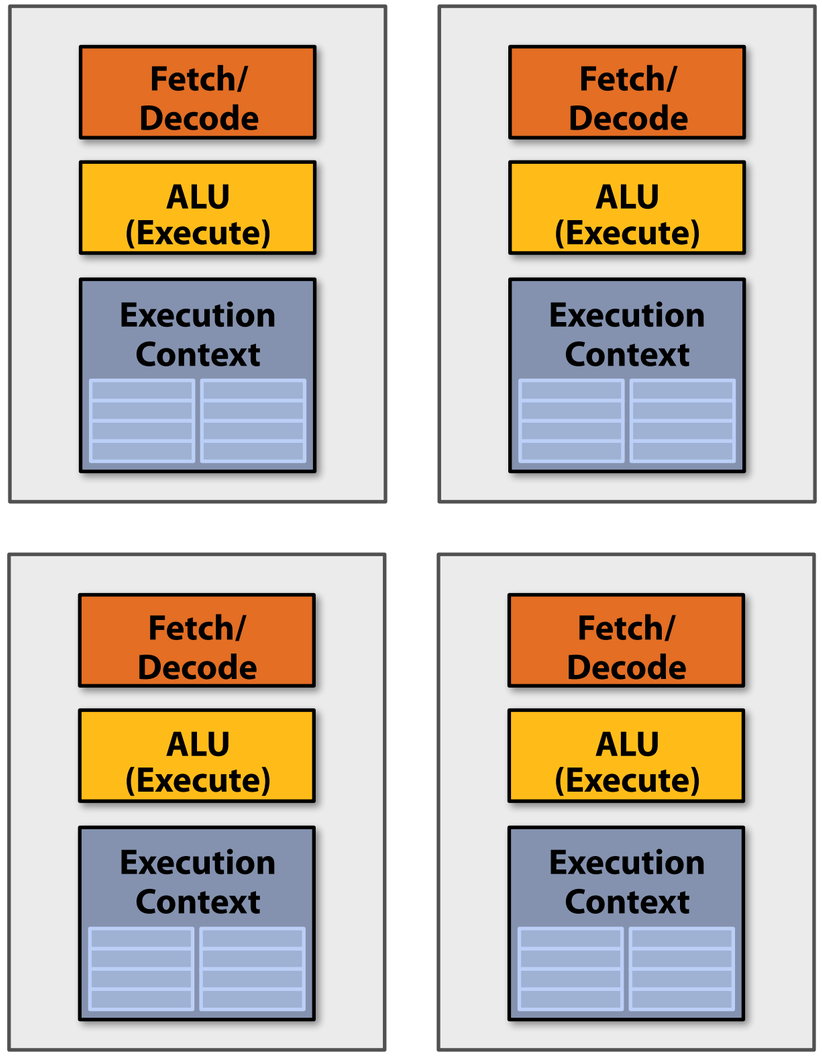


Review: multi-core (four cores)

Modify program to create many threads of control:
(code written in Kayvon's fictitious data-parallel language)

```
void sinx(int N, int terms, float* x, float* result)
{
   // declare independent loop iterations
   forall (int i from 0 to N-1)
   {
      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;
      }

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

My quad-core processor:
executes one instruction per clock
from an instruction stream on <u>each</u> core.



Review: four, 8-wide SIMD cores

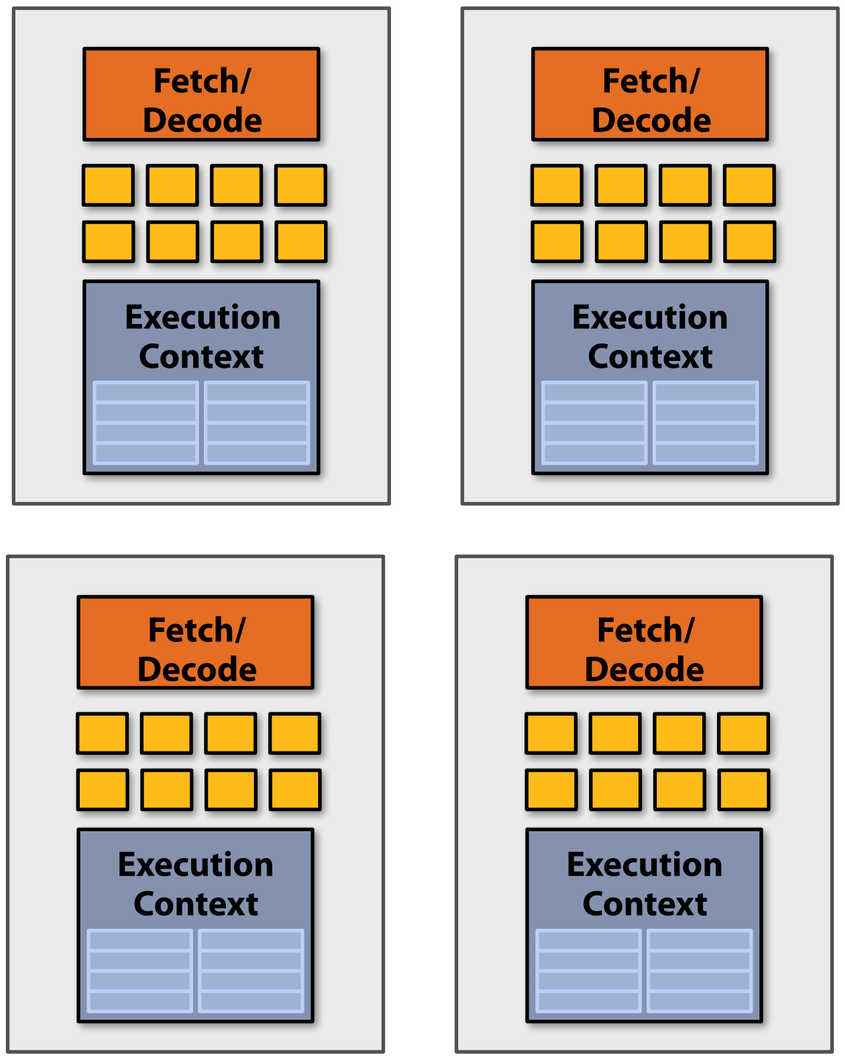
Observation: program must execute many iterations of the same loop body.

Optimization: share instruction stream across execution of multiple iterations (single instruction multiple data = SIMD)

```
void sinx(int N, int terms, float* x, float* result)
{
   // declare independent loop iterations
   forall (int i from 0 to N-1)
   {
      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;
      }

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

My SIMD quad-core processor:

executes one 8-wide SIMD instruction per clock from an instruction stream on each core.



Review: four SIMD, multi-threaded cores

Observation: memory operations have very long latency

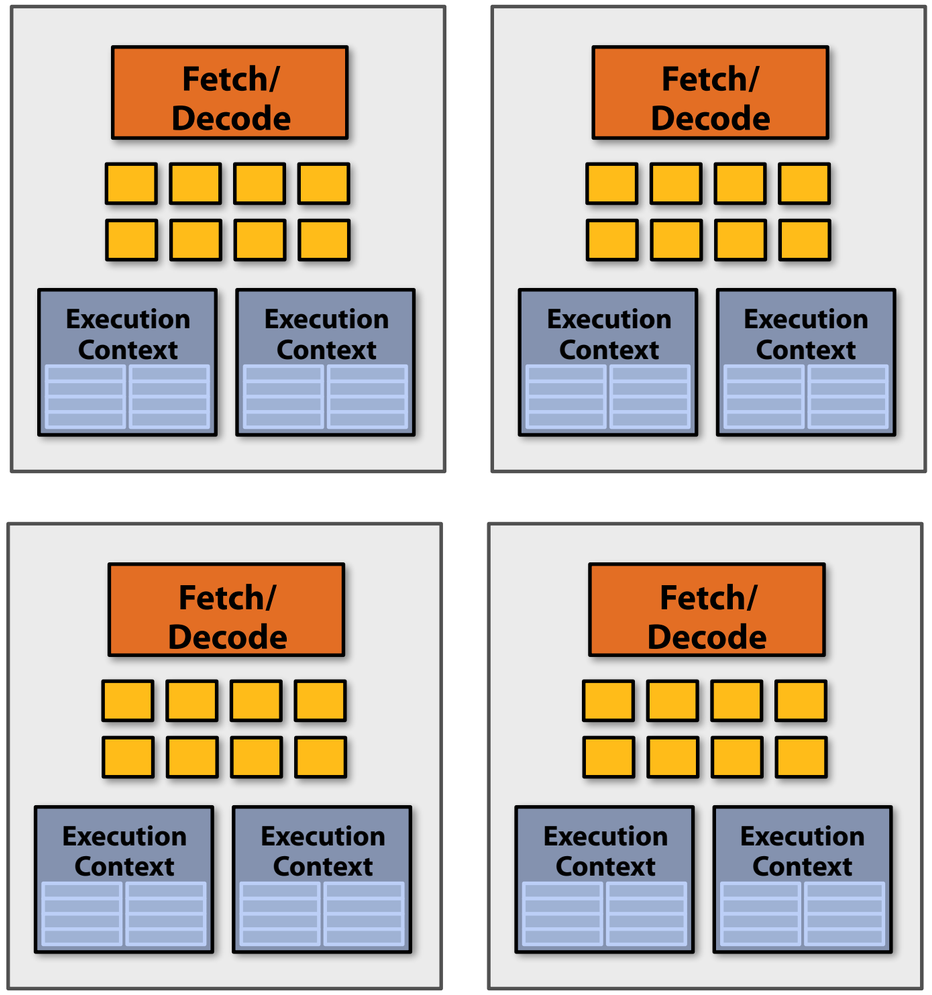
Solution: hide latency of loading data for one iteration by executing arithmetic instructions from other iterations

```
void sinx(int N, int terms, float* x, float* result)
{
   // declare independent loop iterations
   forall (int i from 0 to N-1)
   {
      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;
      }

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

Memory load

Memory store

My multi-threaded, SIMD quad-core processor: executes one SIMD instruction per clock from one instruction stream on each core. But can switch to processing the other instruction stream when faced with a stall.

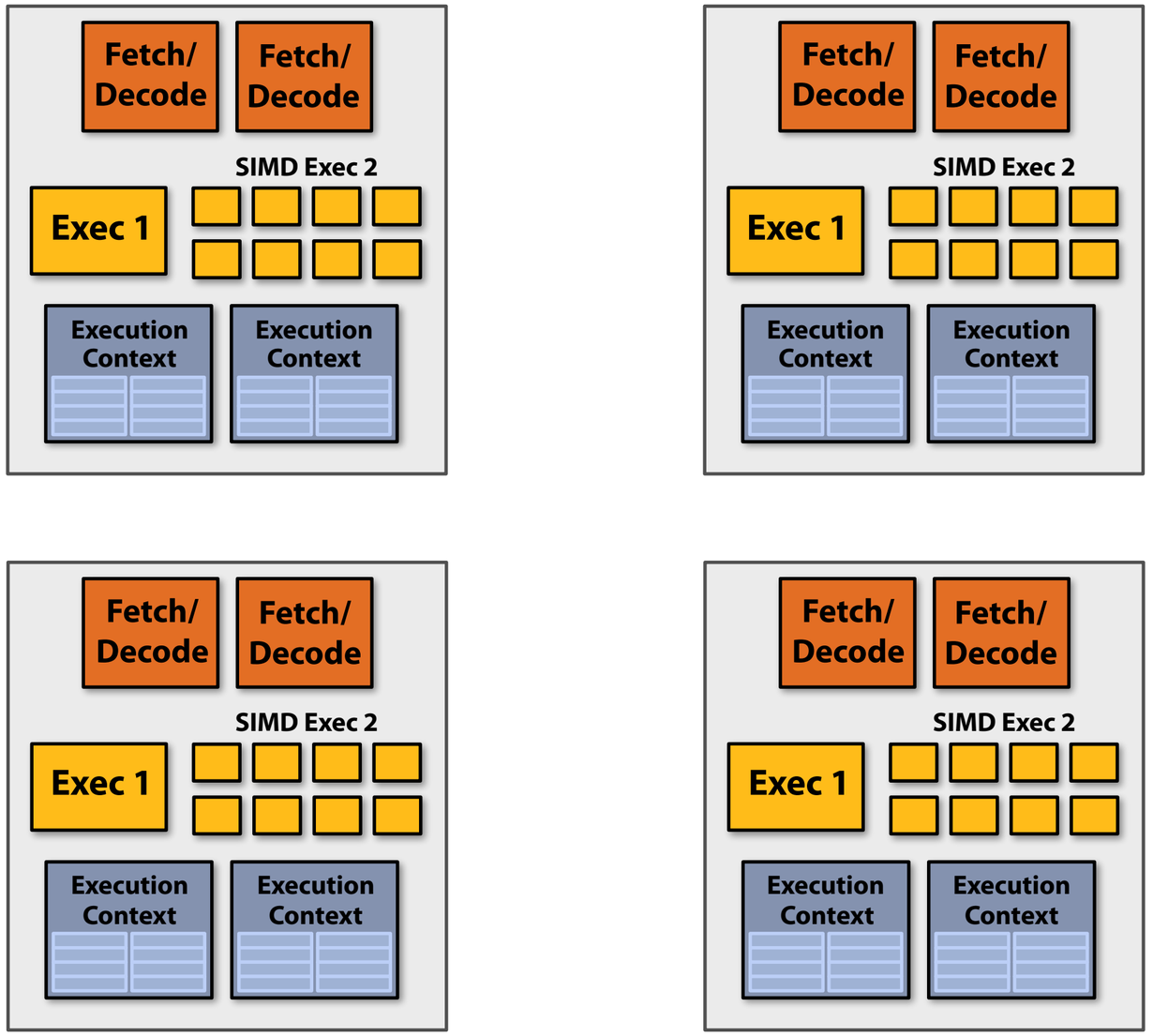


Summary: four superscalar, SIMD, multi-threaded cores

My multi-threaded, superscalar, SIMD quad-core processor:

executes up to two instructions per clock from one instruction stream on each core (in this example: one SIMD instruction + one scalar instruction).

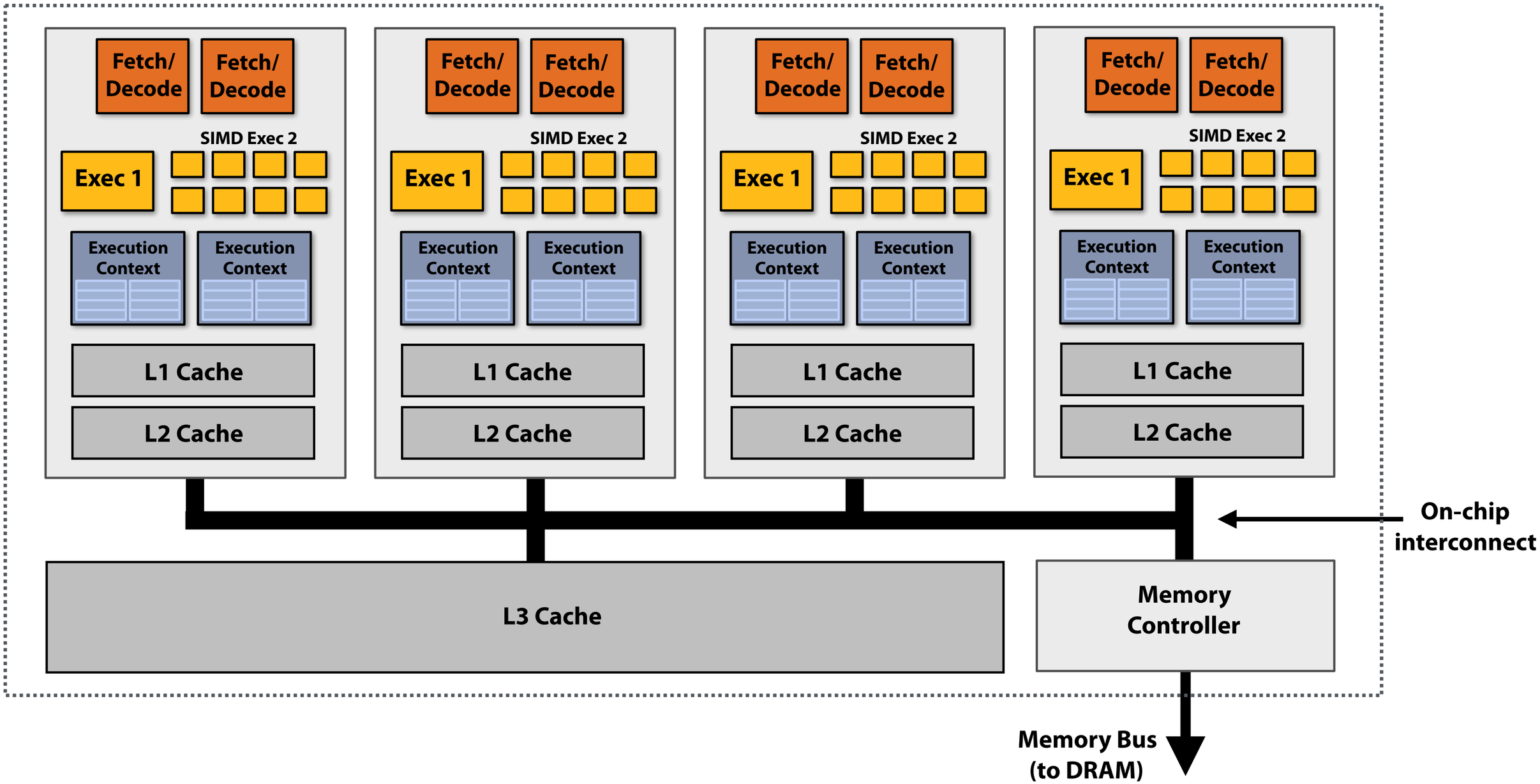
Processor can switch to execute the other instruction stream when faced with stall.



Connecting it all together

Kayvon's simple quad-core processor:

Four cores, two-way multi-threading per core (max eight threads active on chip at once), up to two instructions per clock per core (one of those instructions is 8-wide SIMD)

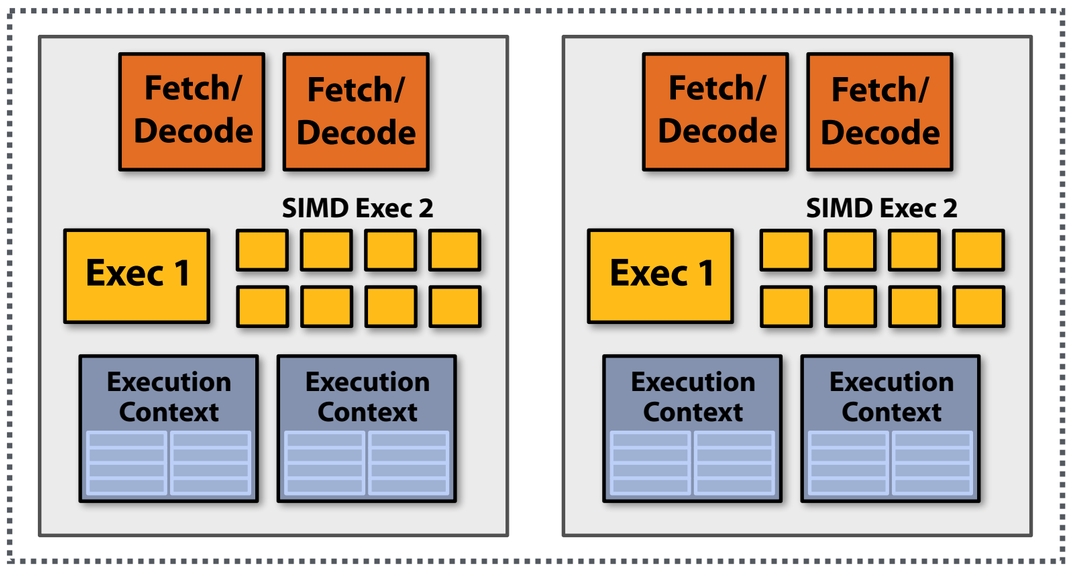


Thought experiment

- You write a C application that spawns <u>two</u> pthreads
- The application runs on the processor shown below
  - Two cores, two-execution contexts per core, up to instructions per clock, one instruction is an 8-wide SIMD instruction.
- Question: "who" is responsible for mapping your pthreads to the processor's thread execution contexts?

Answer: the operating system

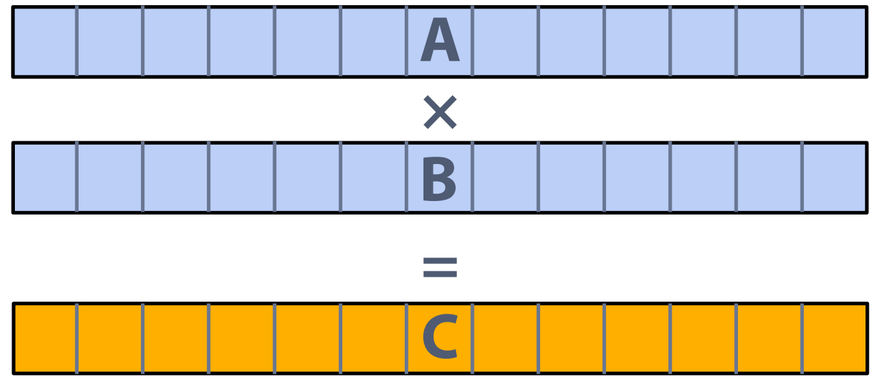
- Question: If you were the OS, how would to assign the two threads to the four available execution contexts?
- Another question: How would you assign threads to execution contexts if your C program spawned <u>five</u> pthreads?



Another 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 ~50 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!

Bandwidth limited!

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

No amount of latency hiding helps this.

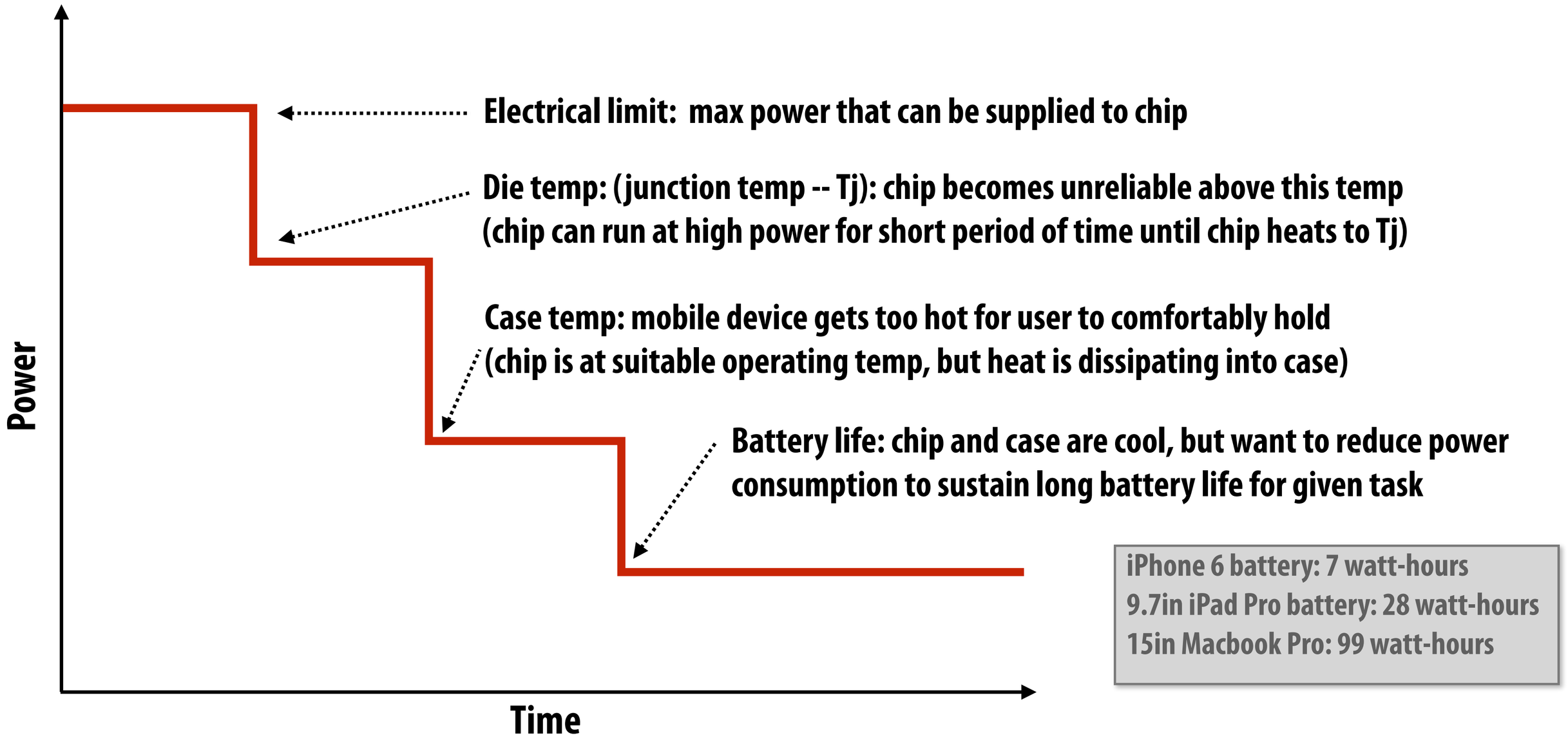
Bandwidth is a critical resource

Overcoming bandwidth limits are a common challenge for application developers on throughput-optimized systems.

Hardware specialization

Why does energy efficiency matter?

- 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)



iPhone 6 battery: 7 watt-hours
9.7in iPad Pro battery: 28 watt-hours
15in Macbook Pro: 99 watt-hours

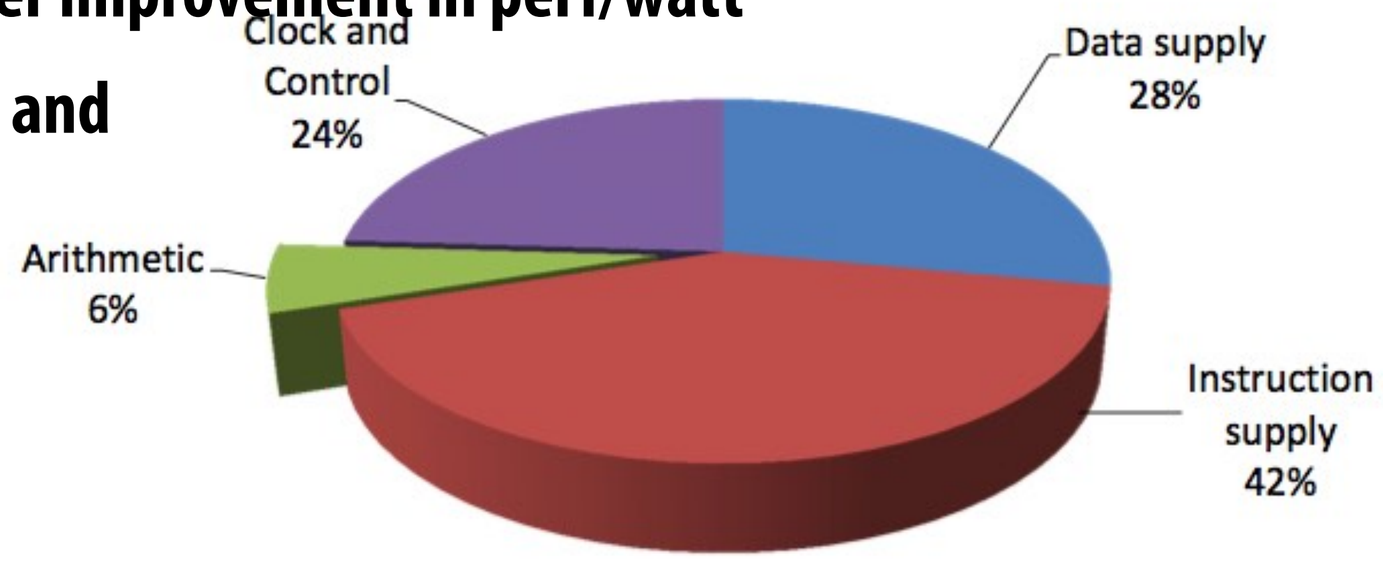
Slide credit: adopted from original slide from M. Shebanow: HPG 2013 keynote

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

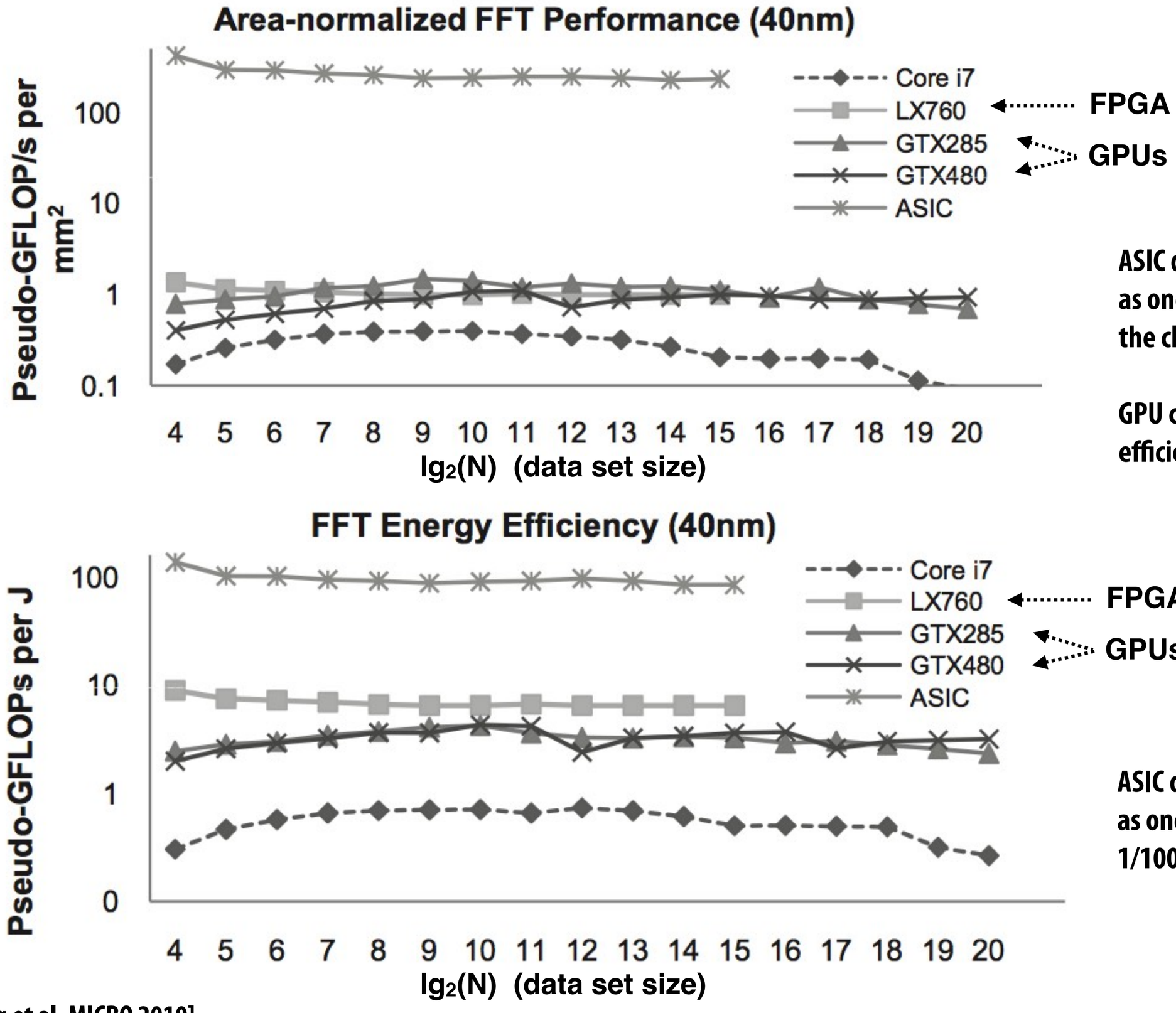


Efficient Embedded Computing [Dally et al. 08]

[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.

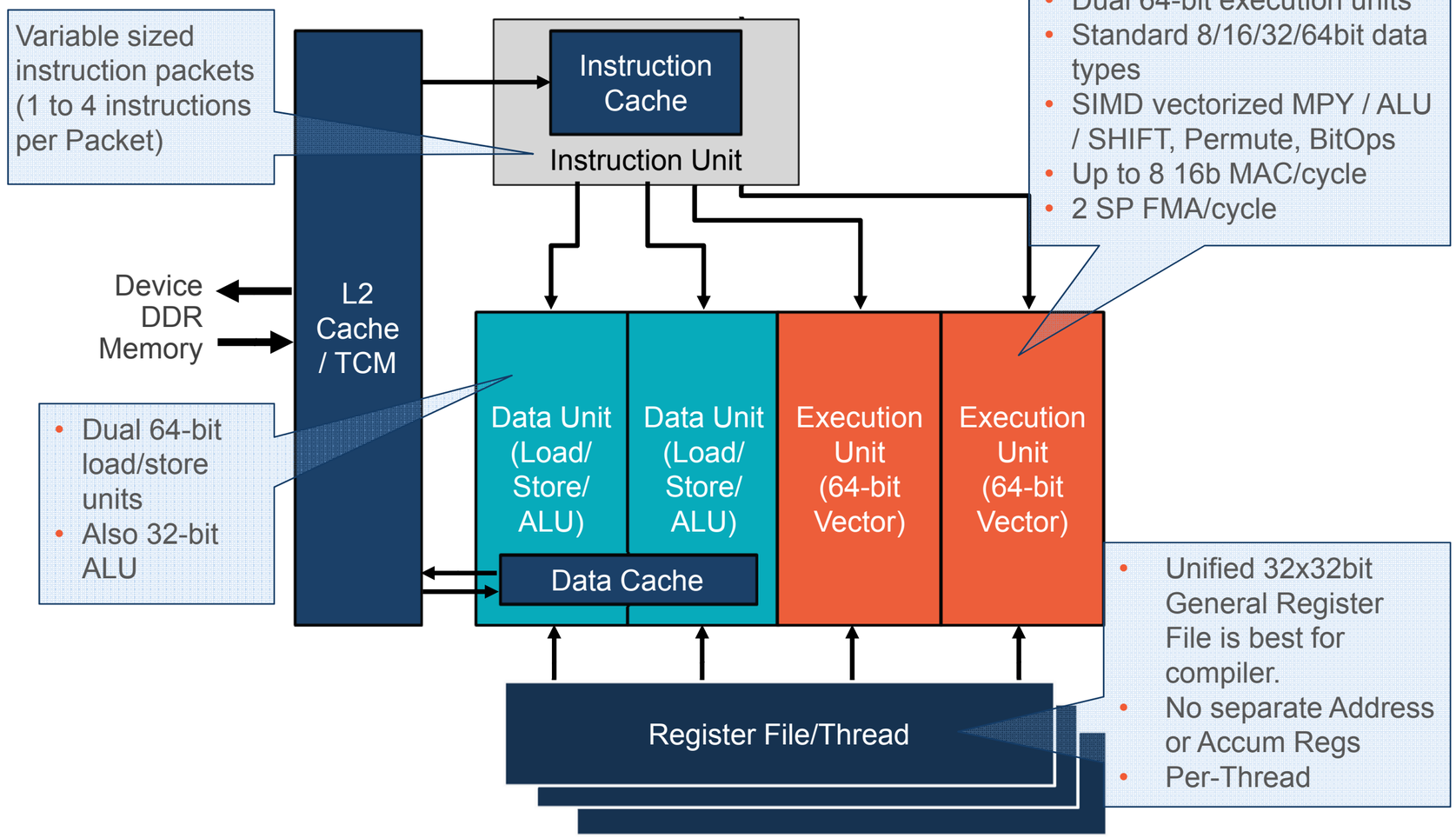
[Chung et al. MICRO 2010]

Modern systems use ASICs 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)
- Significant modern interest in ASICs for deep network evaluation (e.g., Google's Tensor Processing Unit)

Qualcomm Hexagon DSP

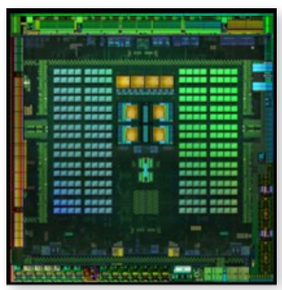
- Originally used for audio/LTE support on Qualcomm SoC's
- Multi-threaded, VLIW DSP
- Third major programmable unit on modern Qualcomm SoCs
  - Multi-core CPU
  - Multi-core GPU (Adreno)
  - Hexagon DSP



Summary: choosing the right tool for the job

Energy-optimized CPU

Throughput-oriented processor (GPU)



Programmable DSP

HEXAGON™

FPGA/Future reconfigurable logic

ASIC

Video encode/decode, Audio playback, Camera RAW processing, neural nets (future?)

~10X more efficient

~100X??? (jury still out)

~100-1000X more efficient

Easiest to program

Difficult to program (making it easier is active area of research)

Not programmable + costs 10-100's millions of dollars to design / verify / create

Credit Pat Hanrahan for this taxonomy

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

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.

Welcome to cs348v!

- Make sure you are signed up on Piazza so you get announcements

- Tonight's reading:
  - "The Compute Architecture of Intel Processor Graphics Gen9" - Intel Technical Report, 2015
  - "The Rise of Mobile Visual Computing Systems", Fatahalian, IEEE Mobile Computing 2016

More review

For the rest of this class, know these 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

Which program performs better?

Program 1

```
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);
```

Program 2

```
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);
```

(Note: an answer probably needs to state its assumptions.)

More thought questions

Program 1

```
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);
```

Program 2

```
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);
```

Which code structuring style would you rather write?

Consider running either of these programs: would CPU support for hardware-multi-threading help performance?

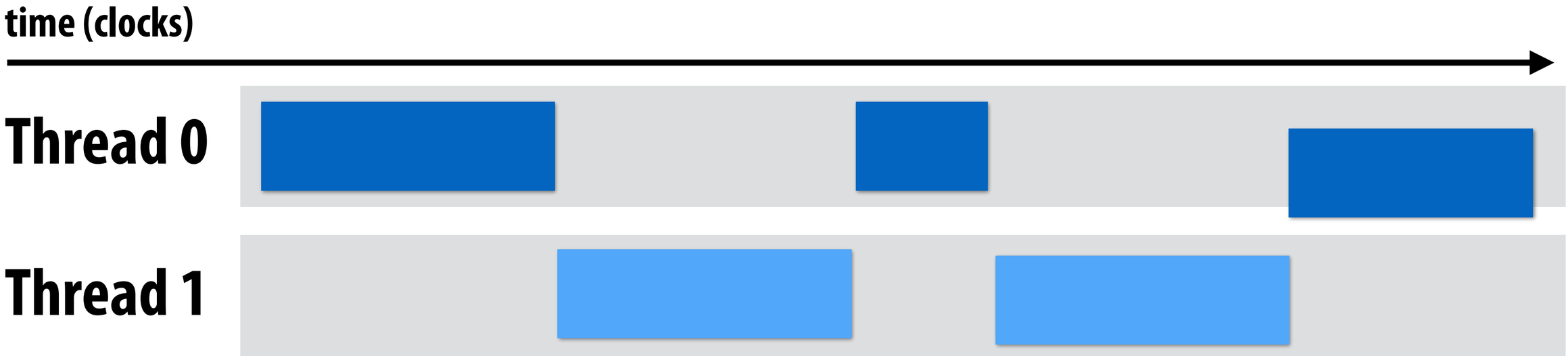
Visualizing interleaved and simultaneous multi-threading

(and combinations thereof)

Interleaved multi-threading

Consider a processor with:

- **Two execution contexts**
- **One fetch and decode unit (one instruction per clock)**
- **One ALU (to execute the instruction)**



= ALU executing T0 at this time

= ALU executing T1 at this time

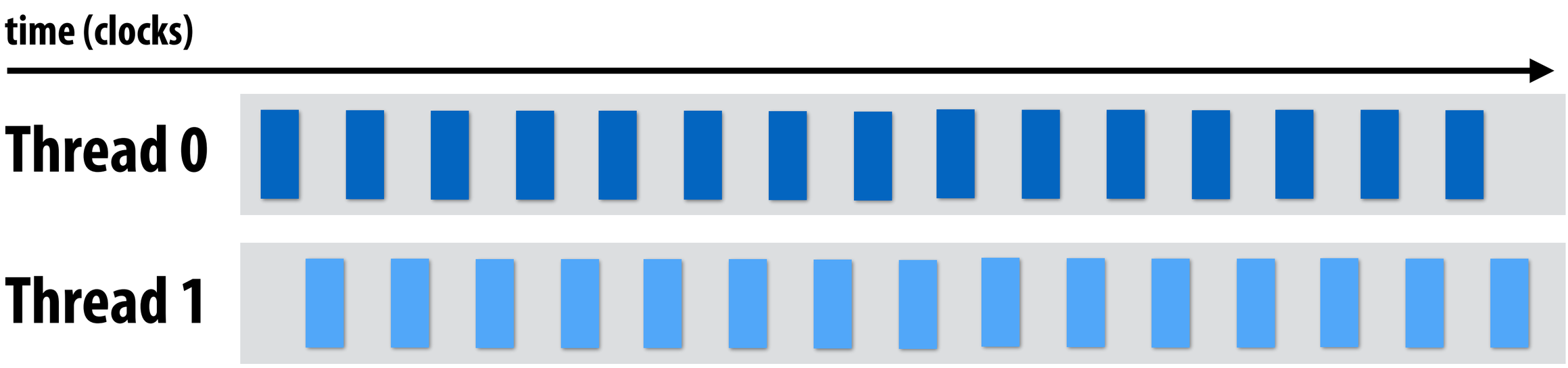
In an interleaved multi-threading scenario, the threads share the processor.

(This is a visualization of when threads are having their instructions executed by the ALU.)

Interleaved multi-threading

Consider a processor with:

- **Two execution contexts**
- **One fetch and decode unit (one instruction per clock)**
- **One ALU (to execute the instruction)**



= ALU executing T0 at this time

= ALU executing T1 at this time

Same as previous slide, but now just a different scheduling order of the threads (fine-grained interleaving)

Simultaneous multi-threading

Consider a processor with:

- **Two execution contexts**
- **Two fetch and decode units (two instructions per clock)**
- **Two ALUs (to execute the two instructions)**

= ALU executing T0 at this time

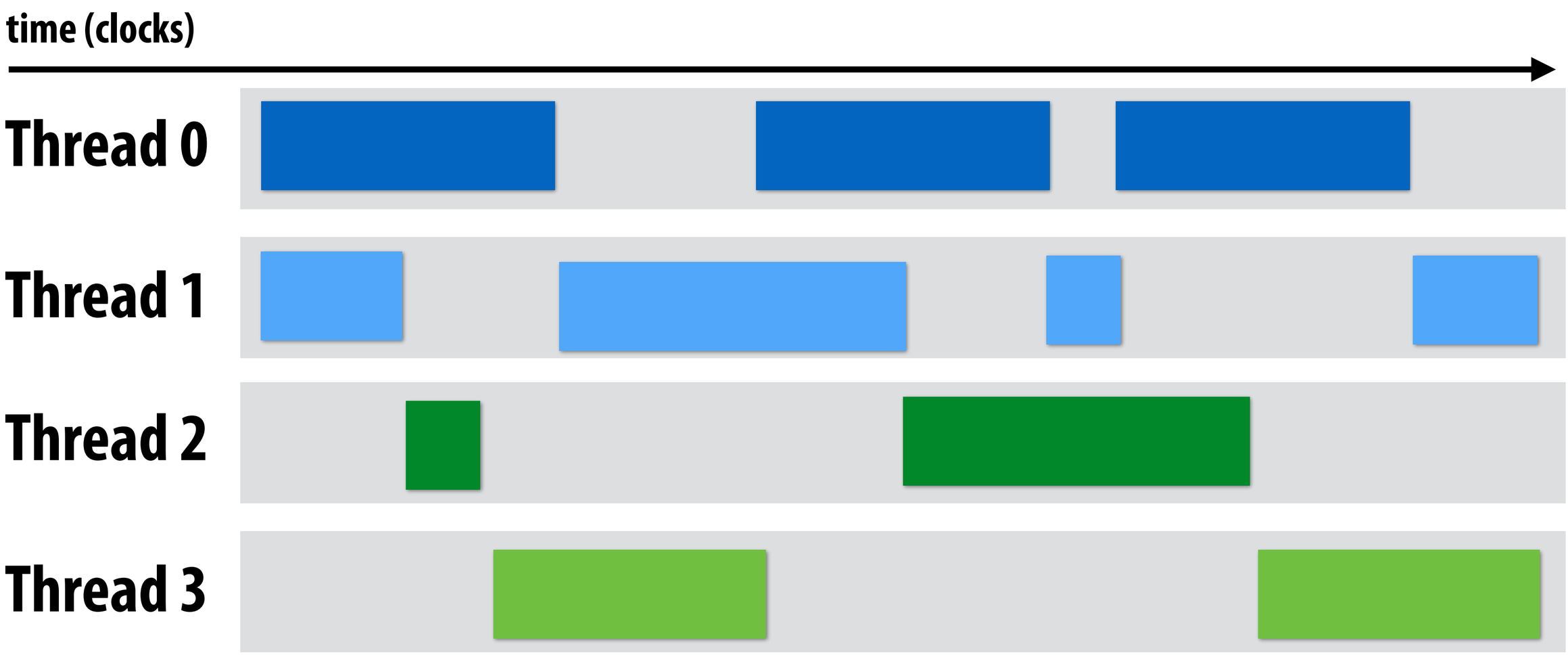
= ALU executing T1 at this time

In an simultaneous multi-threading scenario, the threads execute simultaneously on the two ALUs. (note, no ILP in a thread since each thread is run sequentially on one ALU)

Combining simultaneous and interleaved multi-threading

Consider a processor with:

- **Four execution contexts**
- **Two fetch and decode units (two instructions per clock, choose two of four threads)**
- **Two ALUs (to execute the two instructions)**

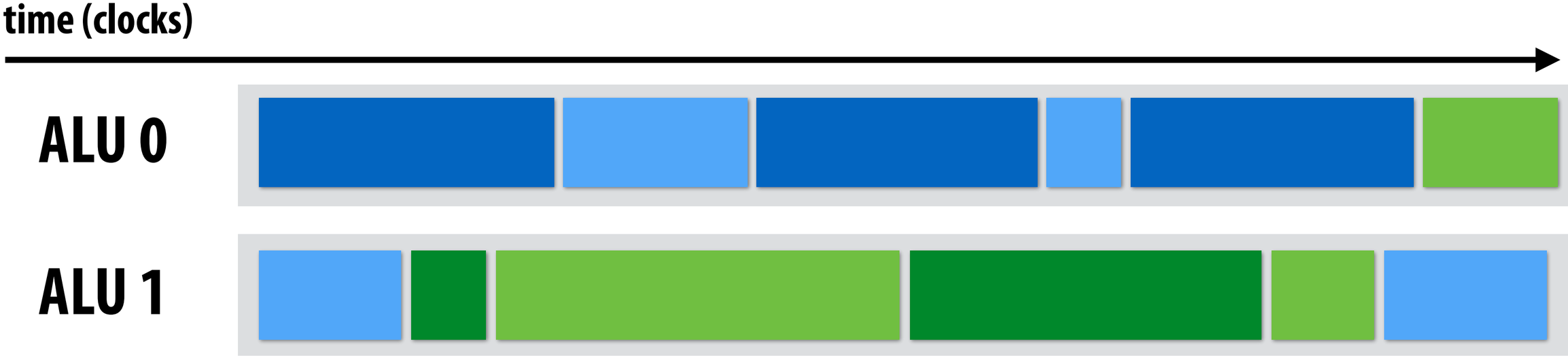


Another way to visualize execution (ALU-centric view)

Consider a processor with:

- **Four execution contexts**
- **Two fetch and decode units (two instructions per clock, choose two of four threads)**
- **Two ALUs (to execute the two instructions)**

Now the graph is visualizing what each ALU is doing each clock:



Instructions can be drawn from same thread (ILP)

Consider a processor with:

- **Four execution contexts**
- **Two fetch and decode units (two instructions per clock, choose any two independent instructions from the four threads)**
- **Two ALUs (to execute the two instructions)**