Lecture 1:

Course Introduction + Review of Throughput Hardware Concepts

Visual Computing Systems
CMU 15-769, Fall 2016

Visual computing

2D/3D graphics

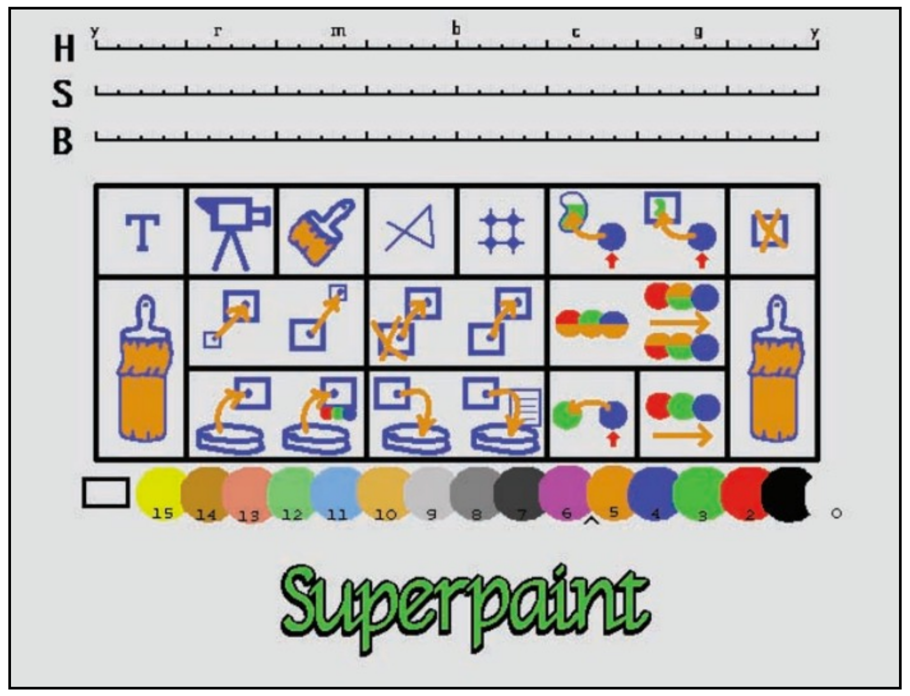
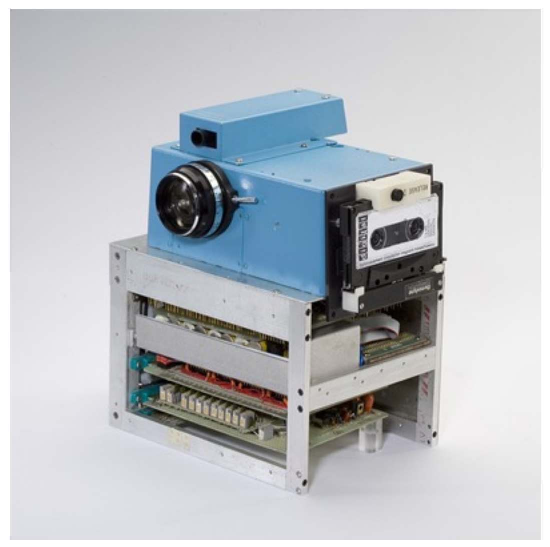
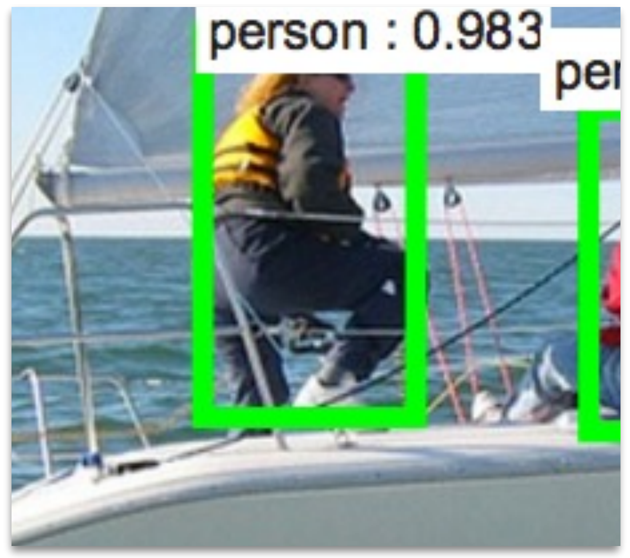
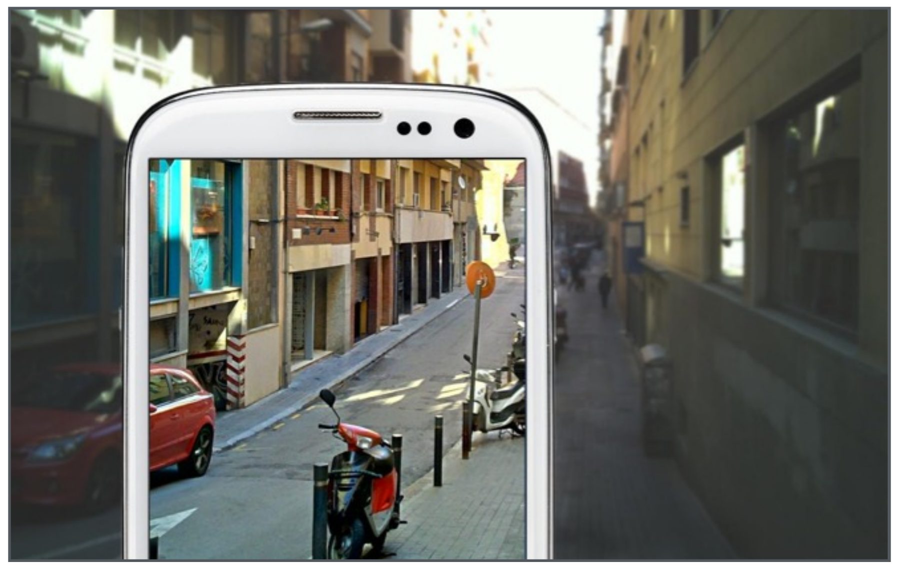
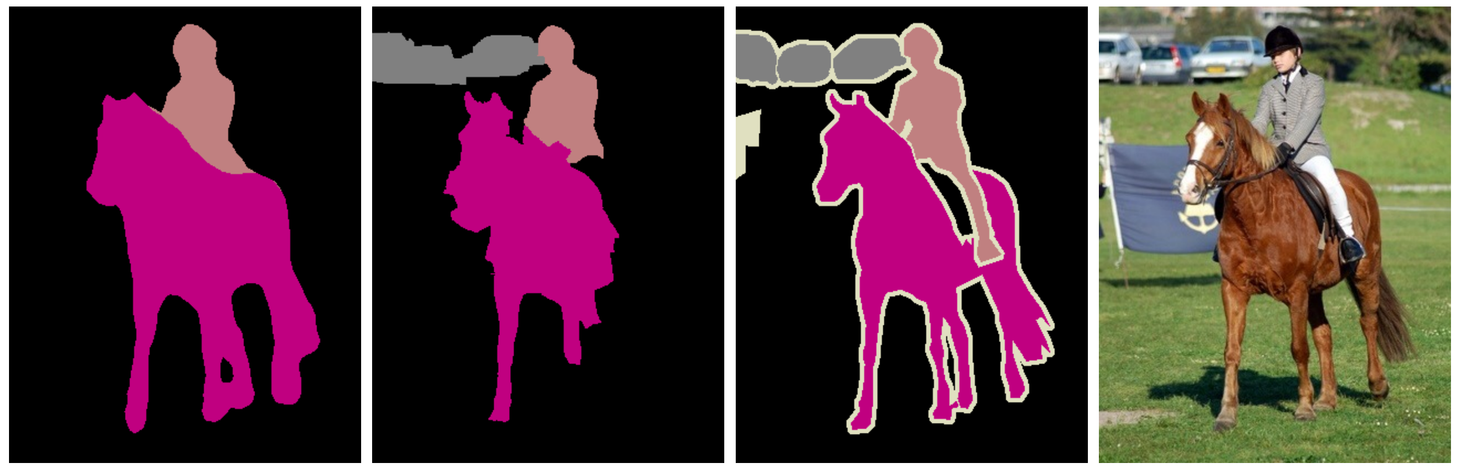


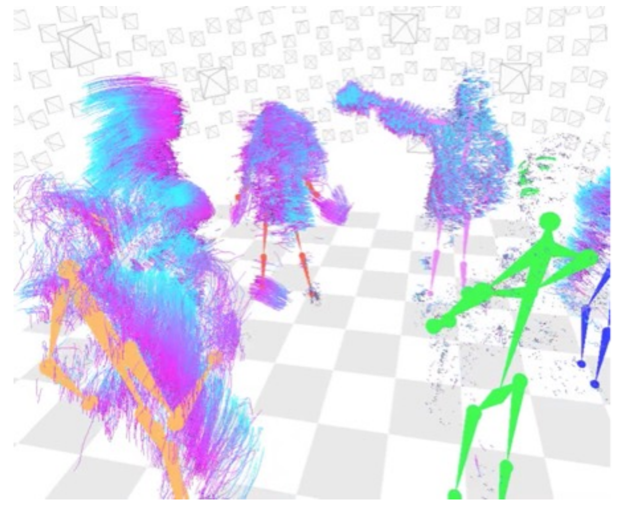
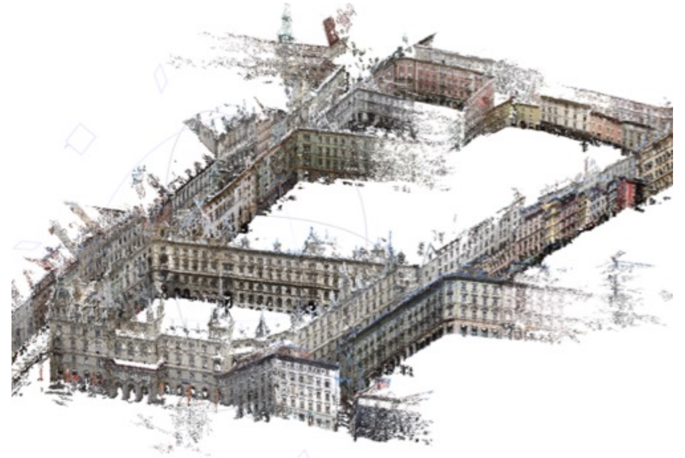
Image processing / computational photography



Computer vision (visual scene understanding)



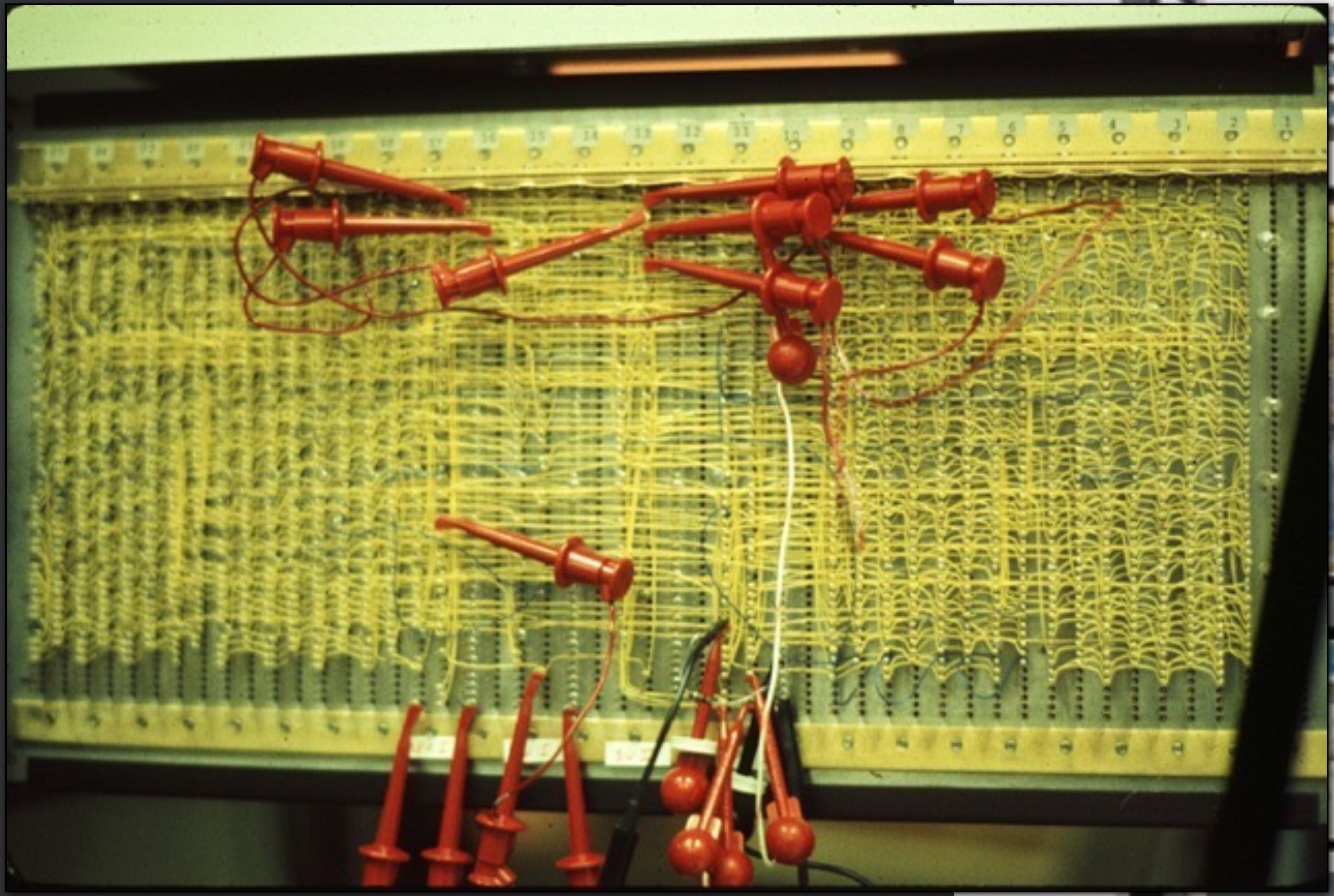
Visual Computing Systems — Some History

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

The frame buffer

Shoup's SuperPaint (PARC 1972-73)

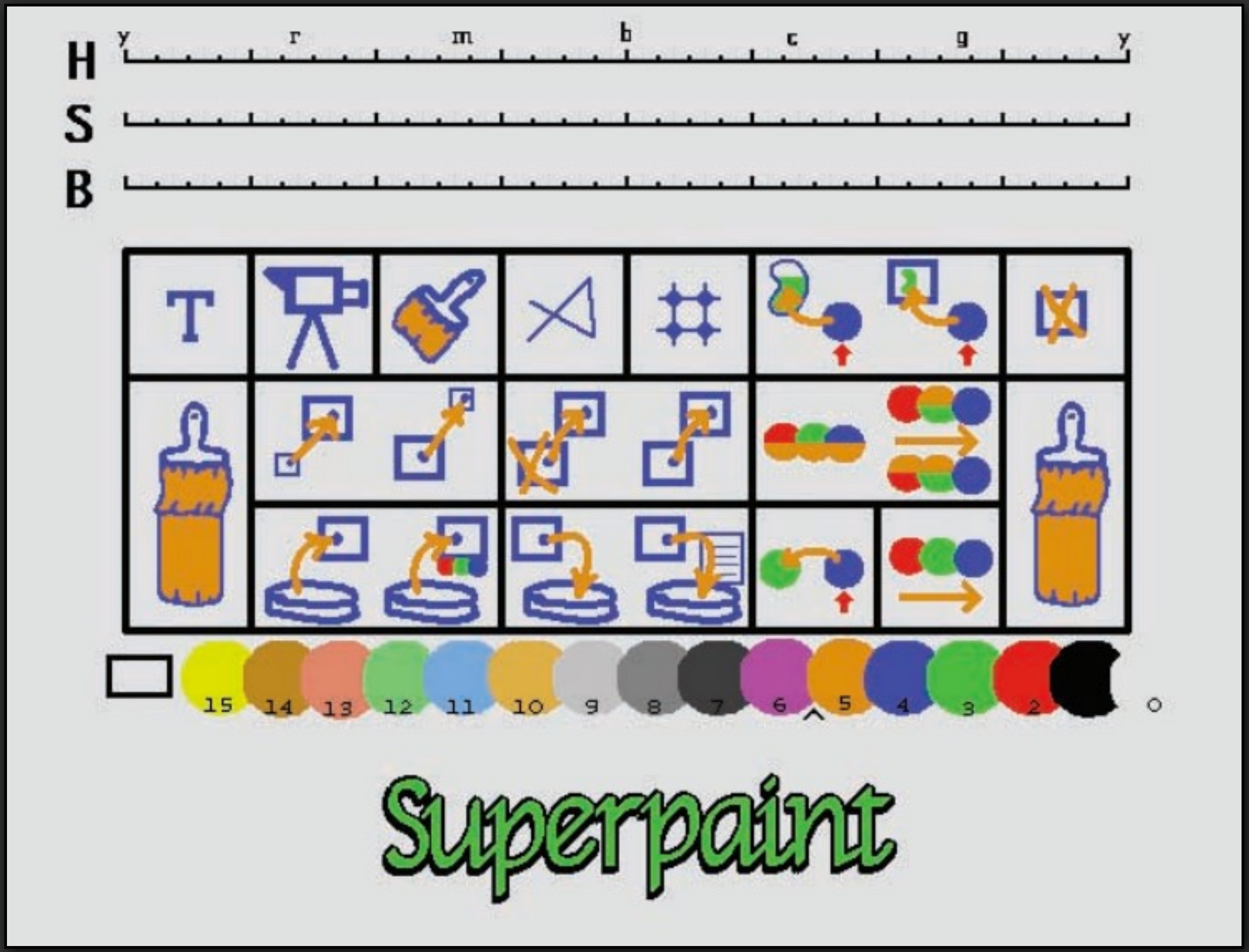
16 2K shift registers (640 x 486 x 8 bits)



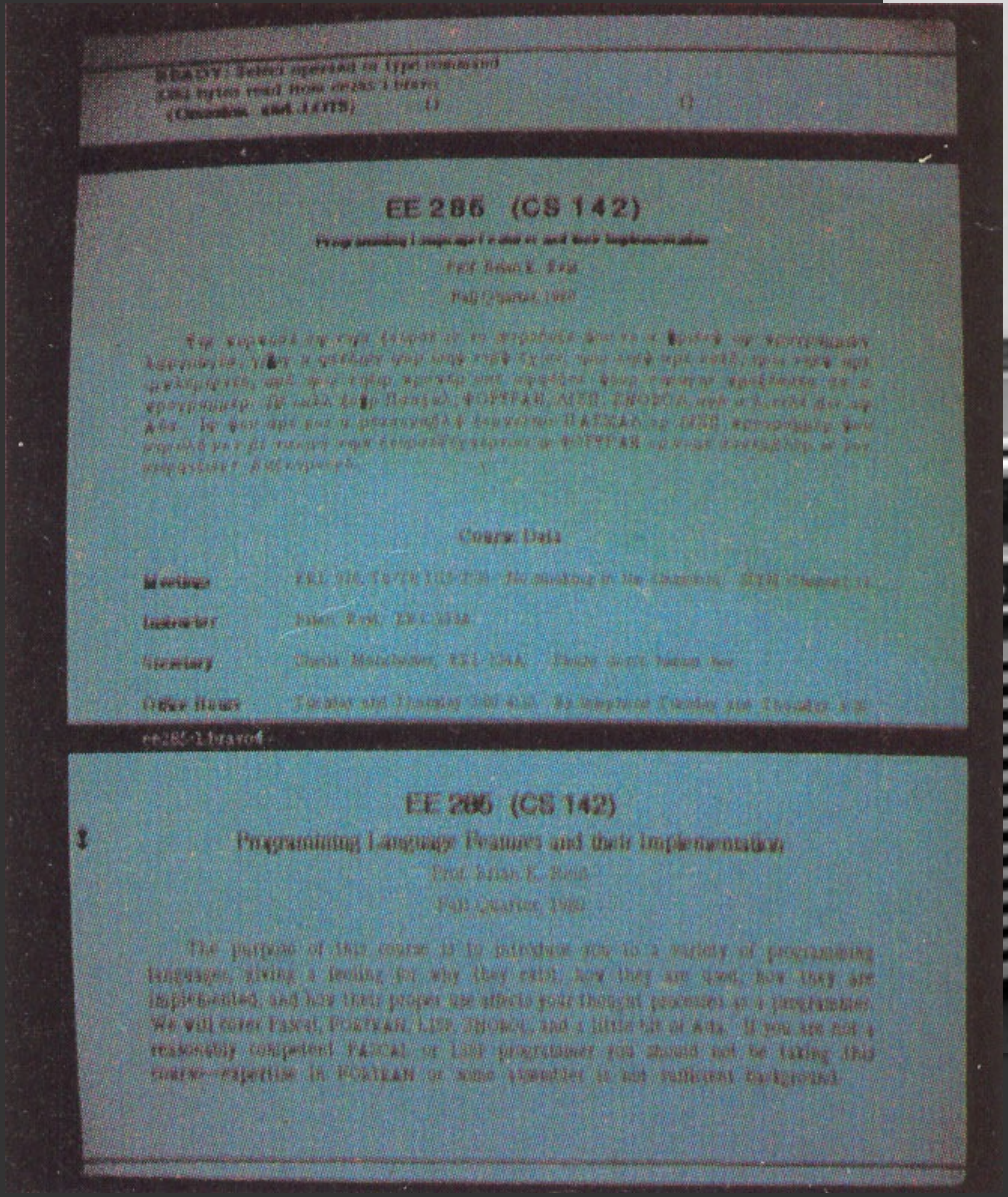
The frame buffer

Shoup's SuperPaint (PARC 1972-73)

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



Xerox Alto (1973)



READY: Select operand or type command
[illegible] bytes read from ee285-1.bravo
(Character [illegible])

EE 286 (CS 142)

Programming Language Features and their Implementation

Prof. [illegible]

Fall Quarter, [illegible]

[illegible]

Course Data

Meetings	[illegible]
Instructor	[illegible]
Secretary	[illegible]
Office Hours	[illegible]

ee285-1.bravo

EE 286 (CS 142)

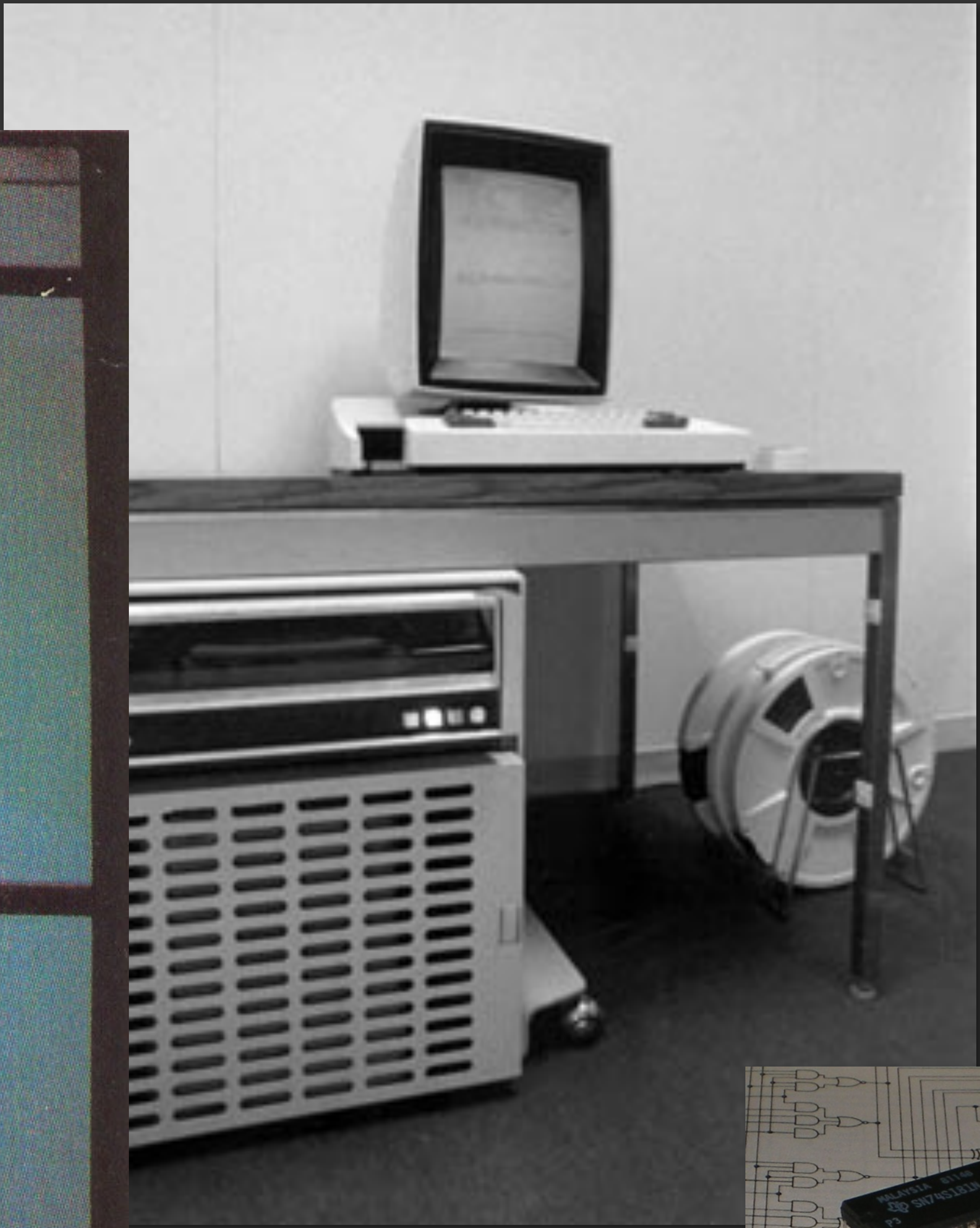
Programming Language Features and their Implementation

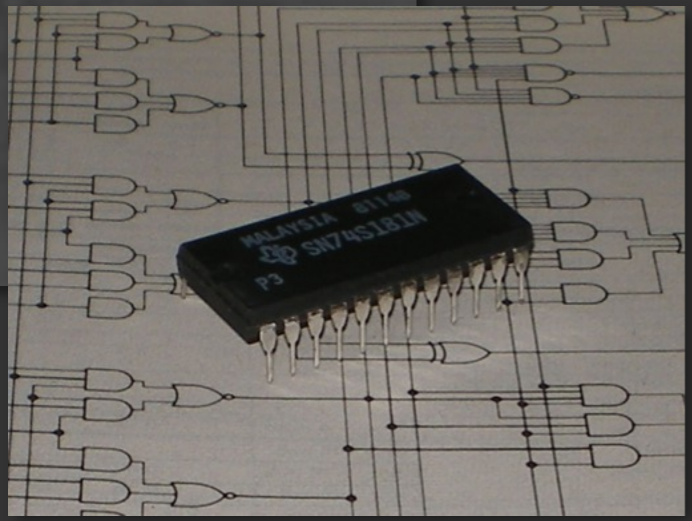
Prof. [illegible]

Fall Quarter, [illegible]

The purpose of this course is to introduce you to a variety of programming languages, giving a feeling for why they exist, how they are used, how they are implemented, and how their proper use affects your thought processes as a programmer. We will cover Pascal, FORTRAN, LISP, SNOBOL, and a little bit of Ada. If you are not a reasonably competent PASCAL or LISP programmer you should not be taking this course—expertise in FORTRAN or some assembler is not sufficient background.

Bravo (WYSIWYG)

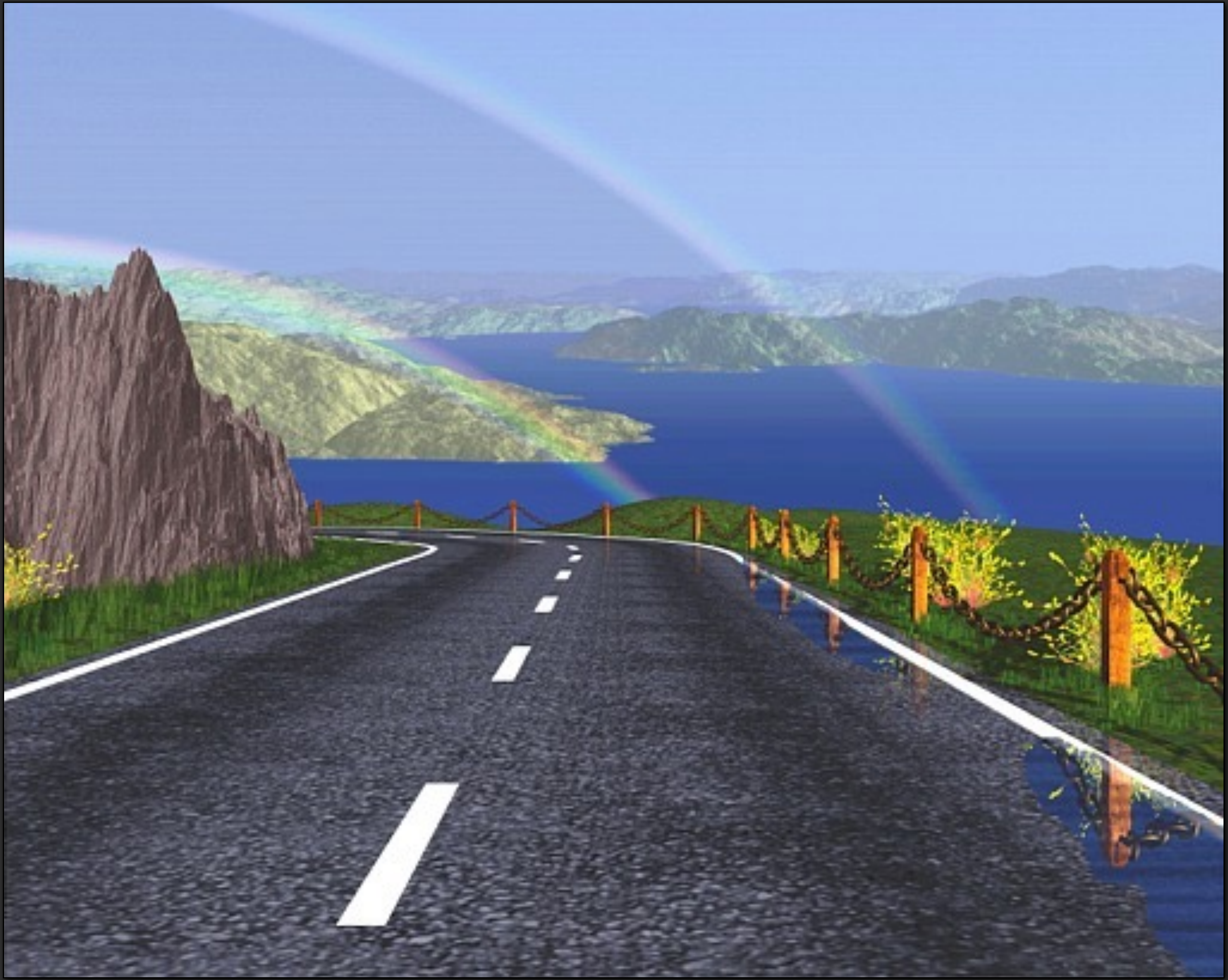




TI 74181 ALU

Goal: <u>r</u>ender <u>e</u>verything <u>y</u>ou've <u>e</u>ver <u>s</u>een

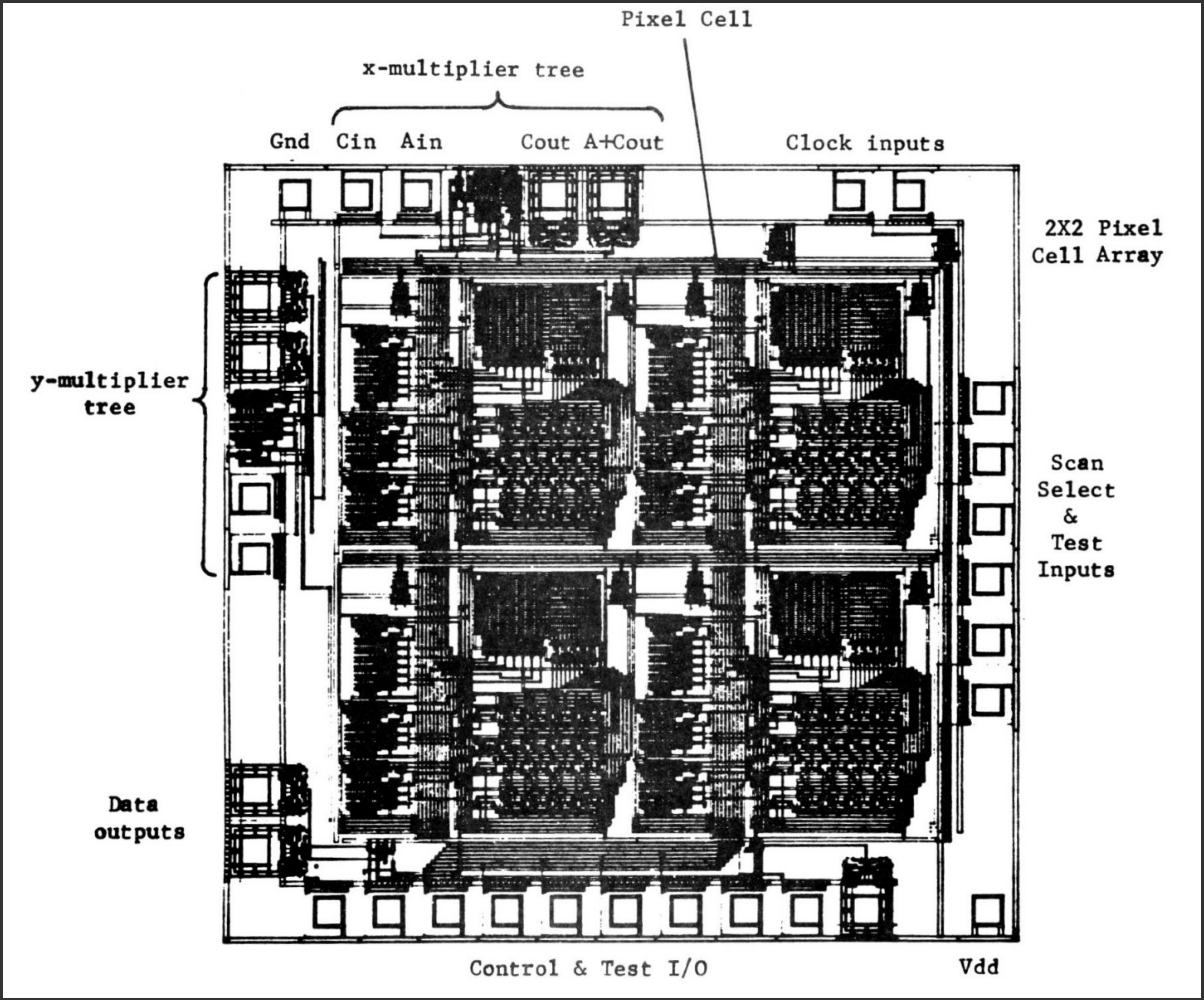
"Road to Pt. Reyes"
LucasFilm (1983)



Pixar's Toy Story (1995)

BUZZ LIGHTYEAR
SPACE

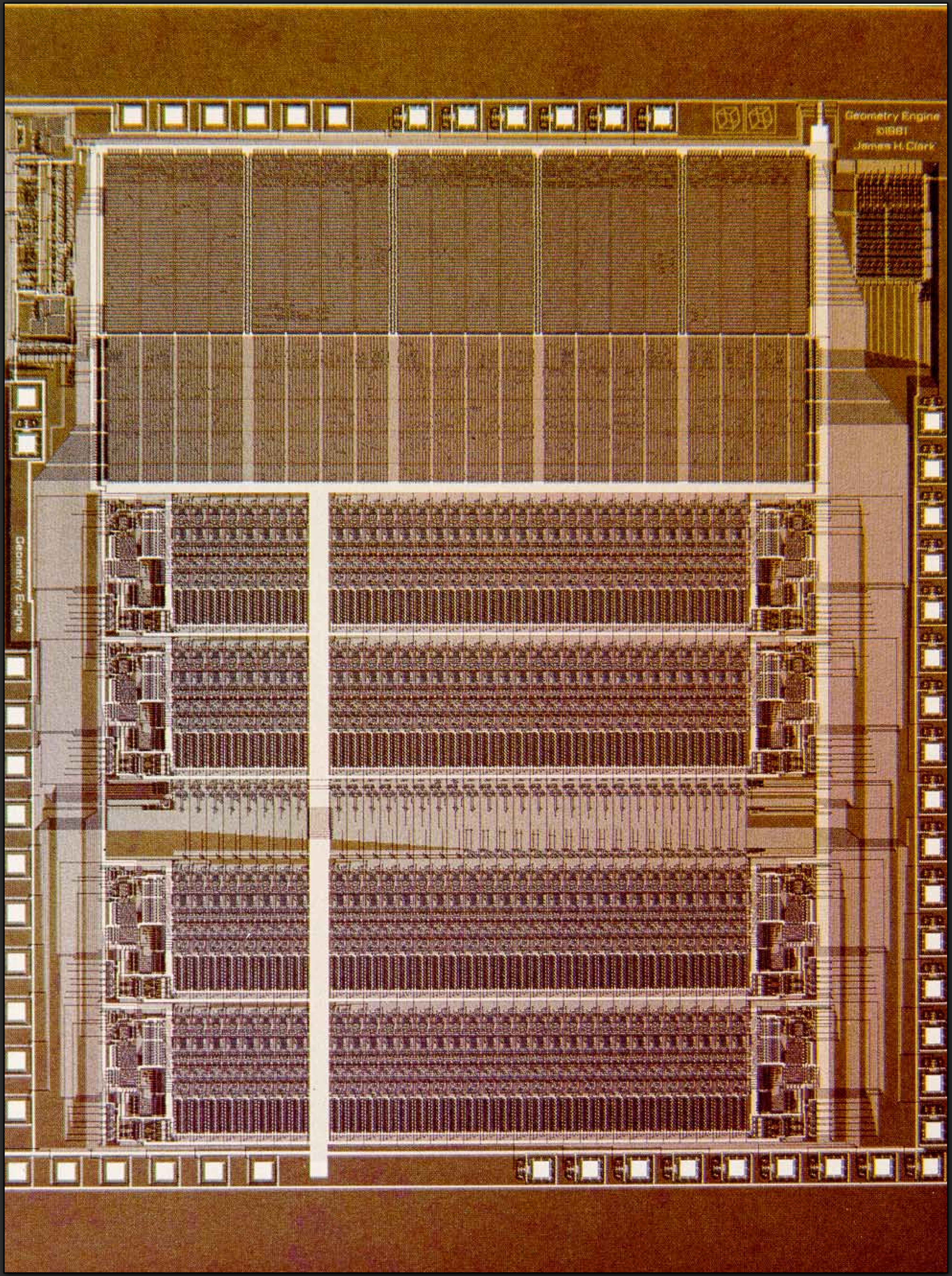
"We take an average of three hours to draw a single frame on the fastest computer money can buy."
 - Steve Jobs

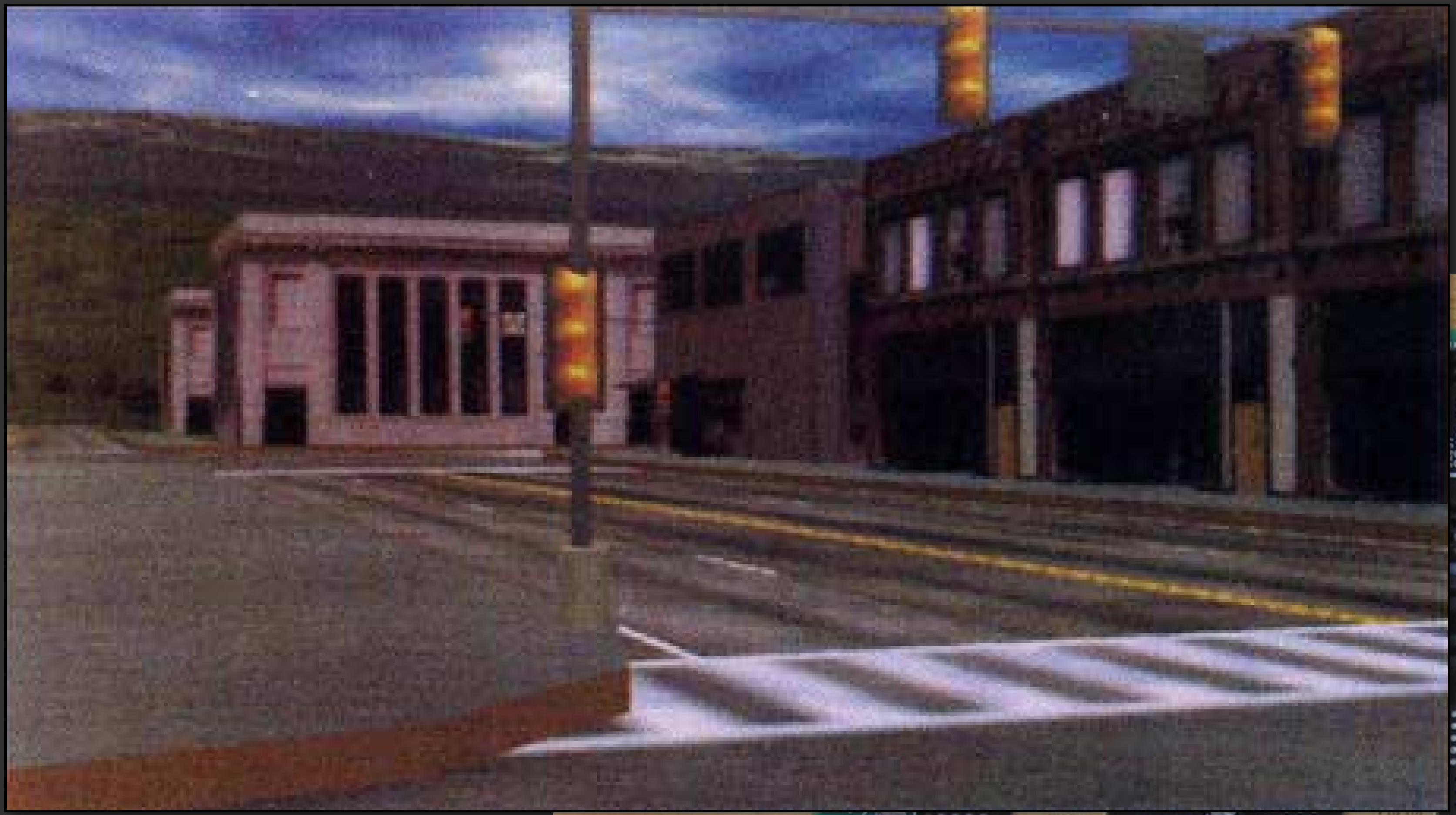


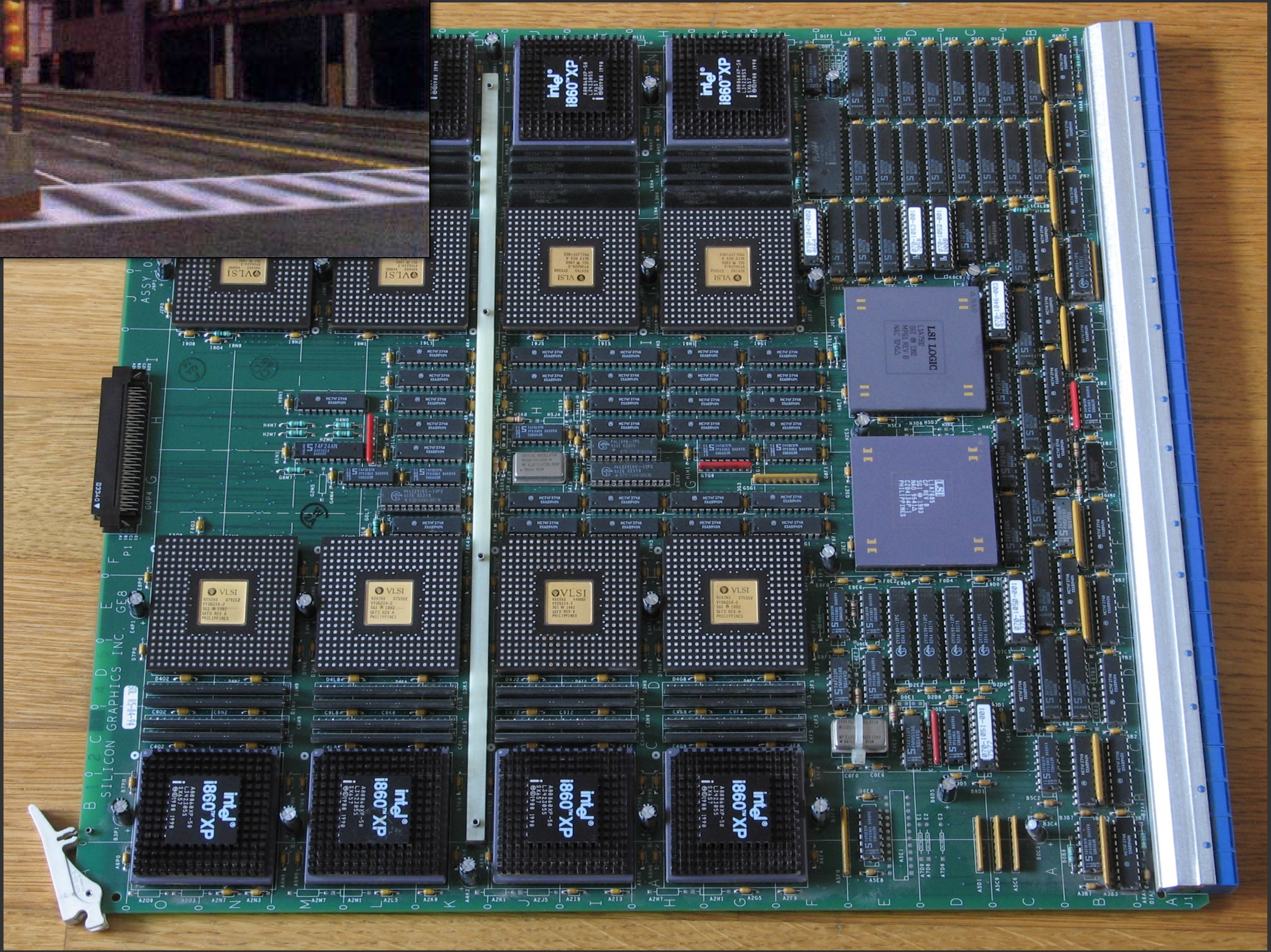
UNC Pixel Planes (1981), computation-enhanced frame buffer

Ed Clark's Geometry Engine (1982)

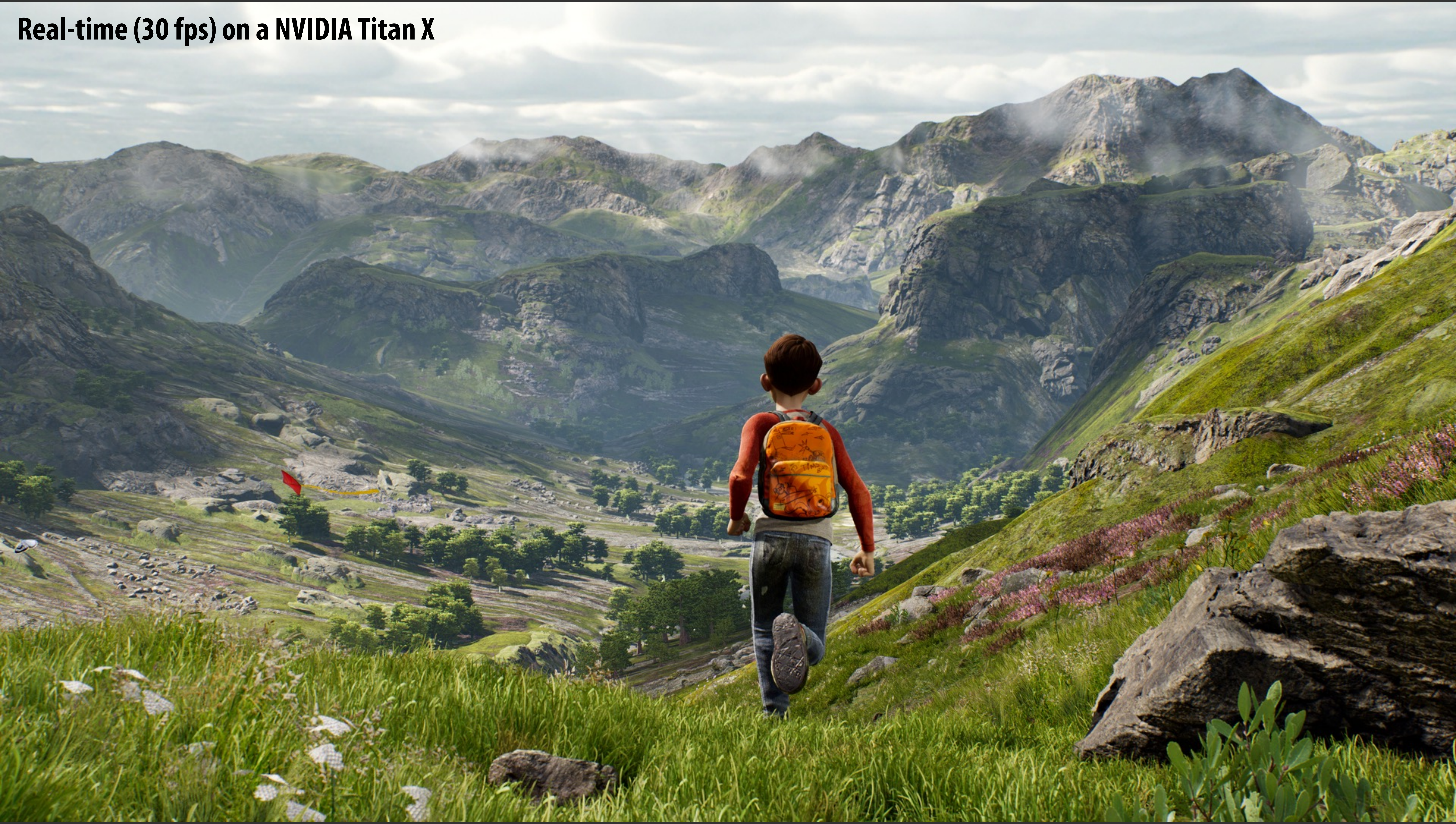
ASIC for geometric transforms used in real-time graphics.



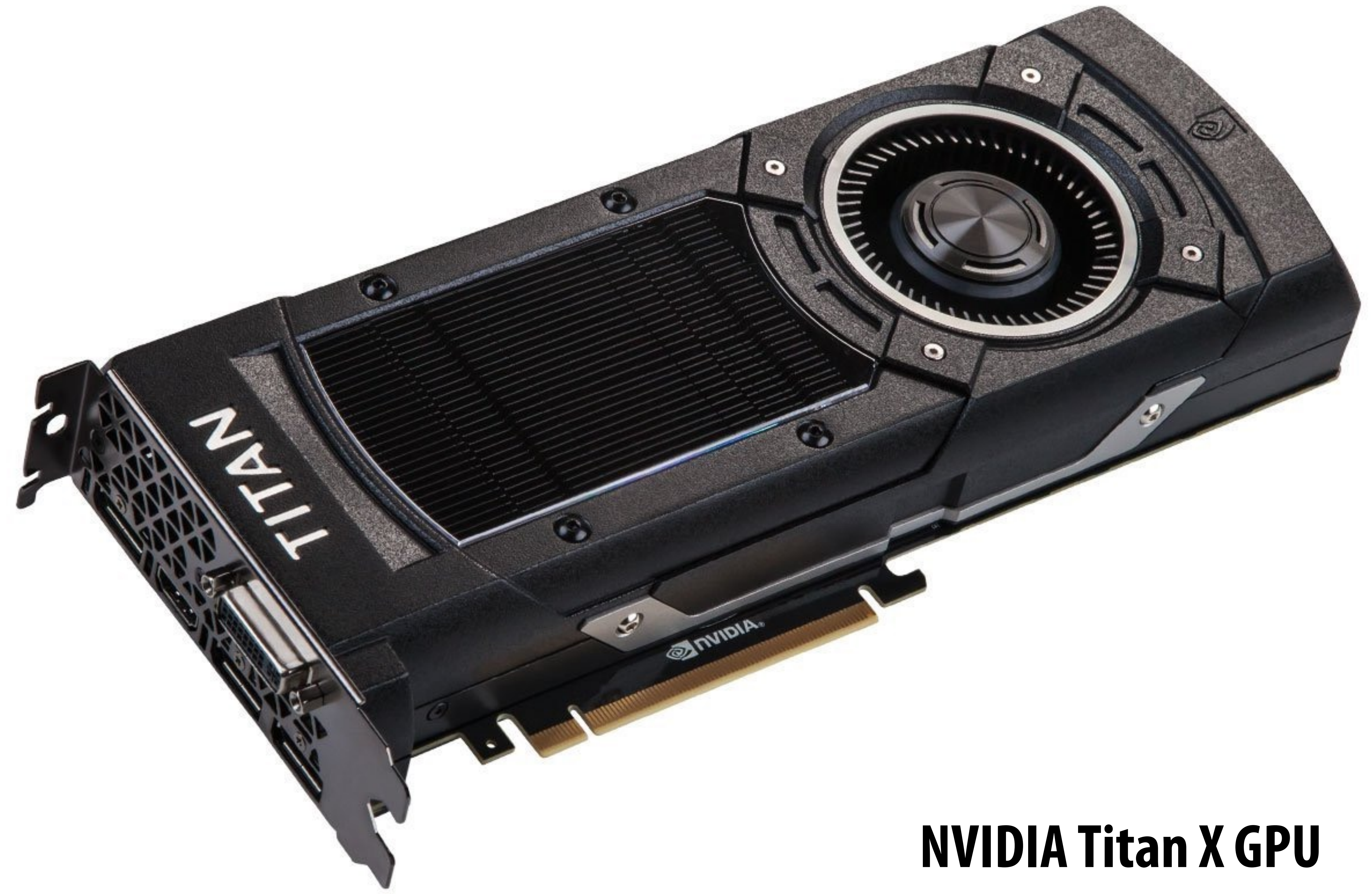




SGI RealityEngine GE8 board (1993)



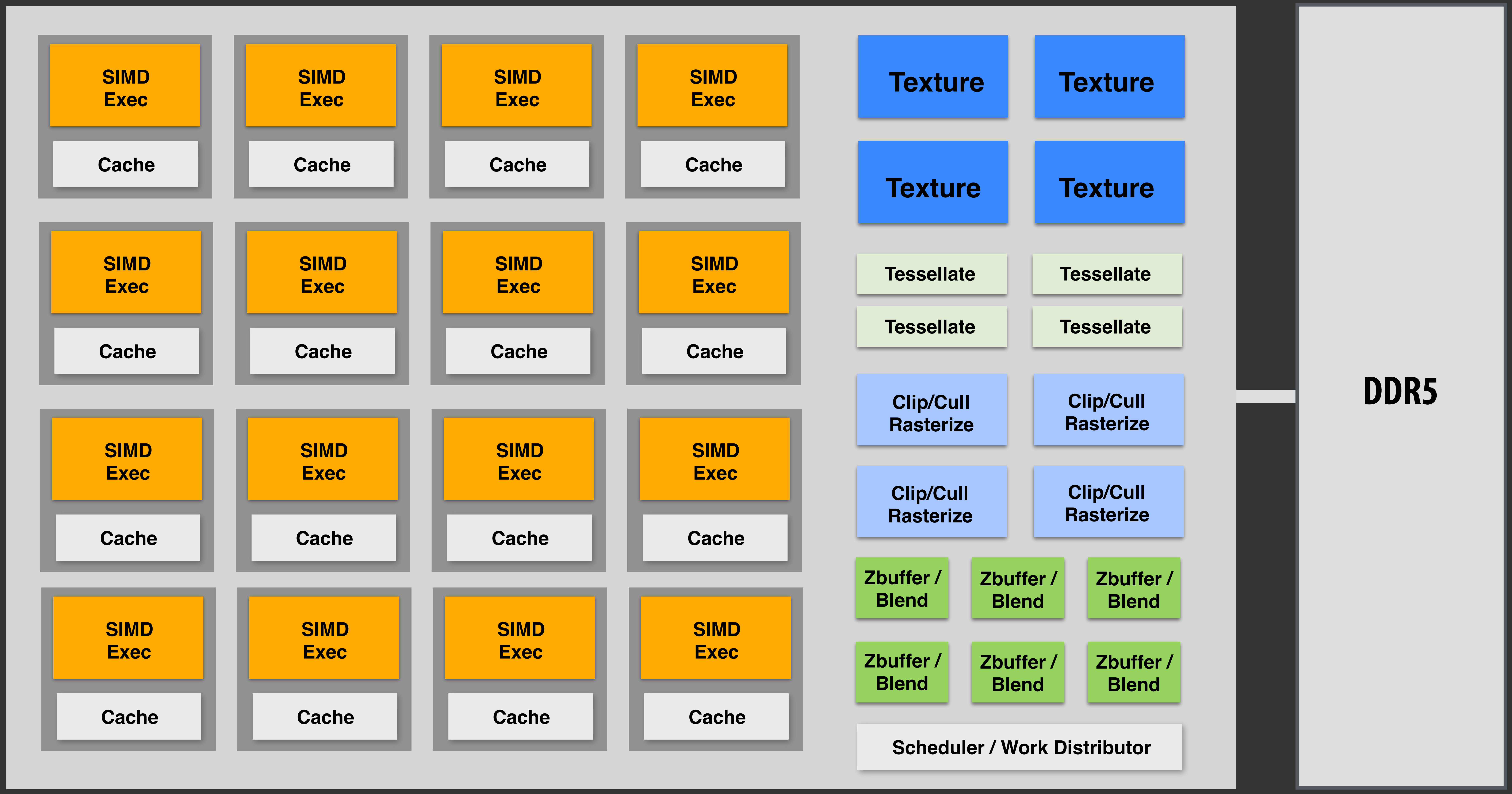
Unreal Engine Kite Demo (Epic Games 2015)



NVIDIA Titan X GPU
(~ 7 TFLOPs fp32)

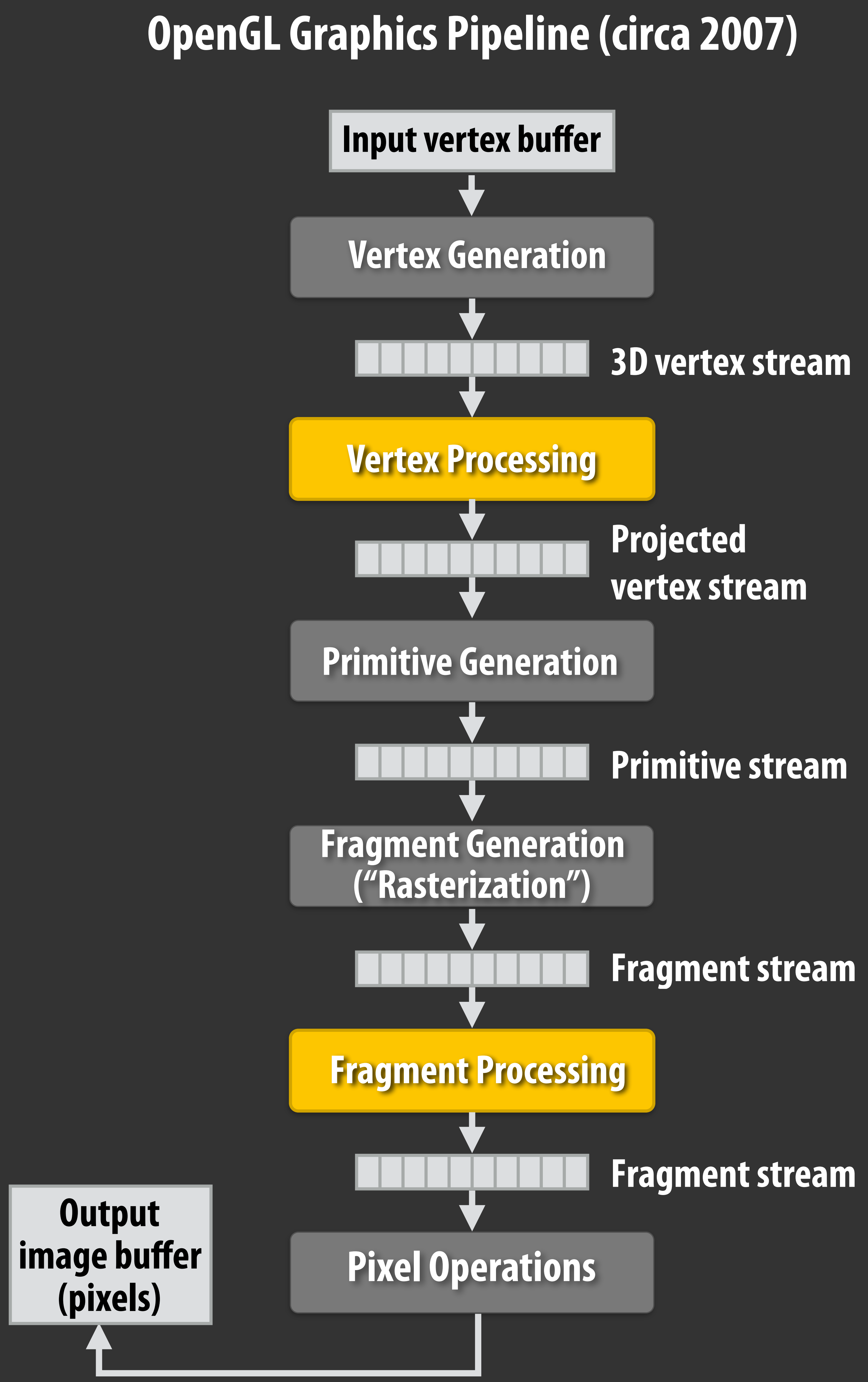
~ ASCI Red (top US supercomputer circa 2000)

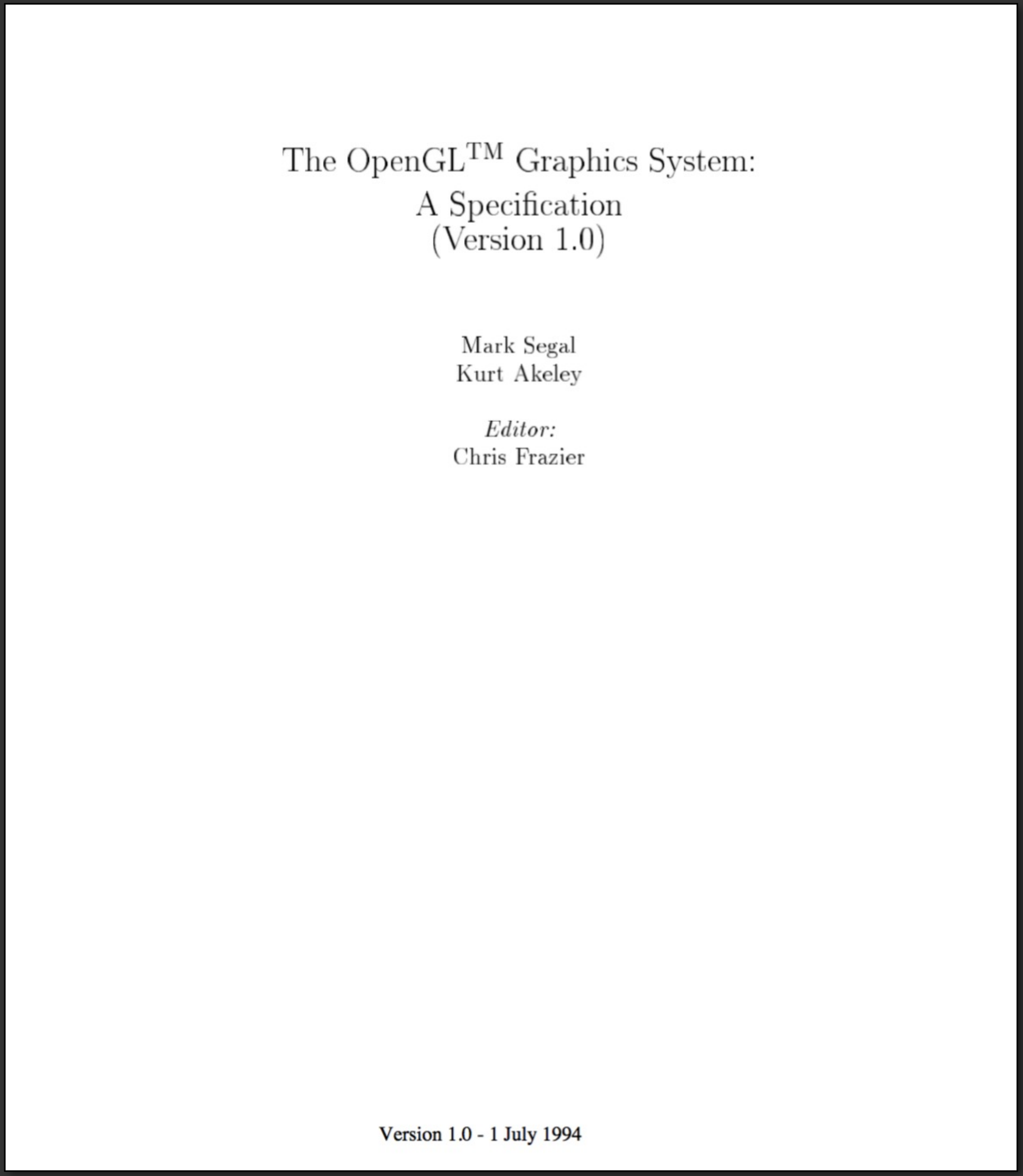
Modern GPU: heterogeneous multi-core



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




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

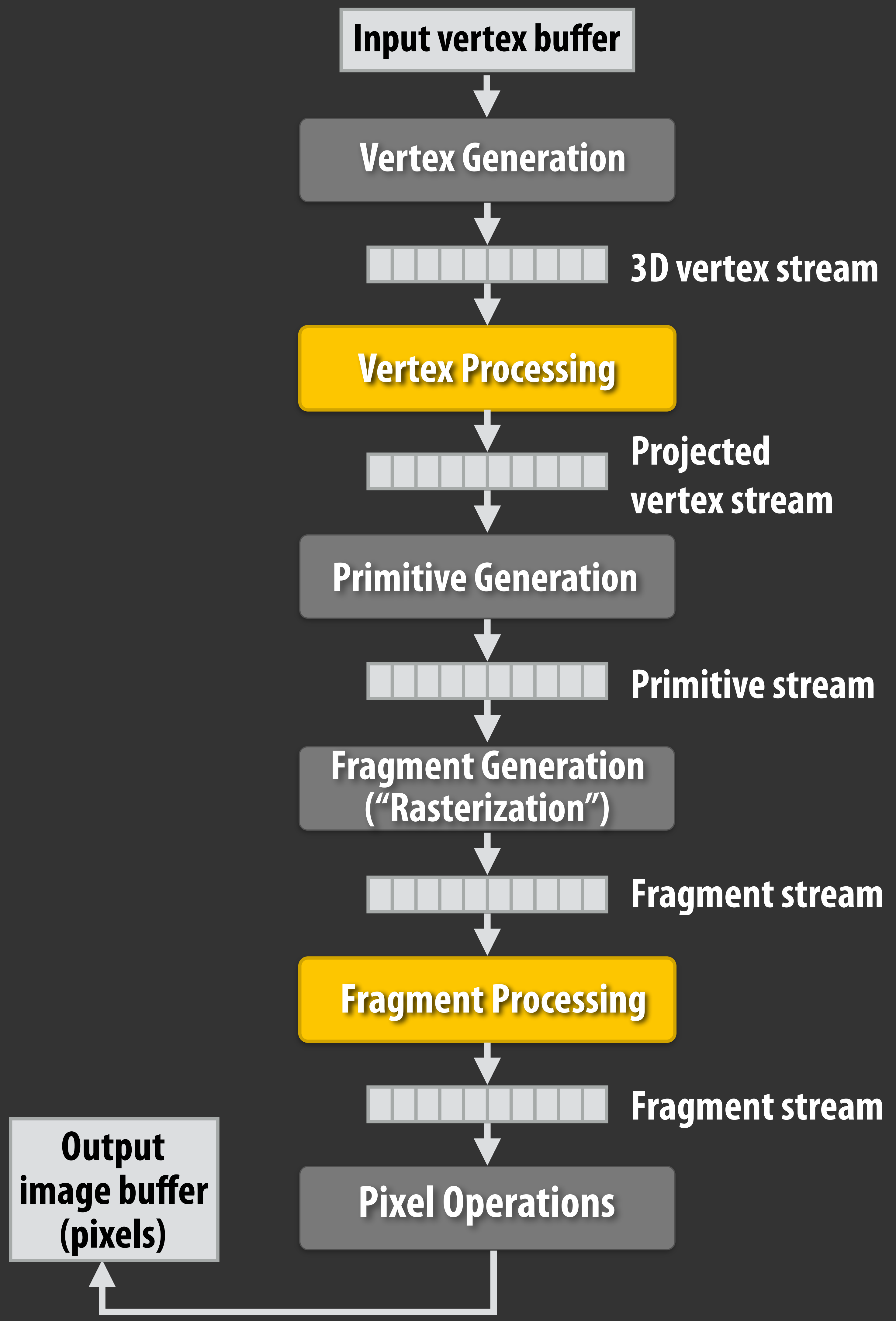
Mark Segal
Kurt Akeley

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

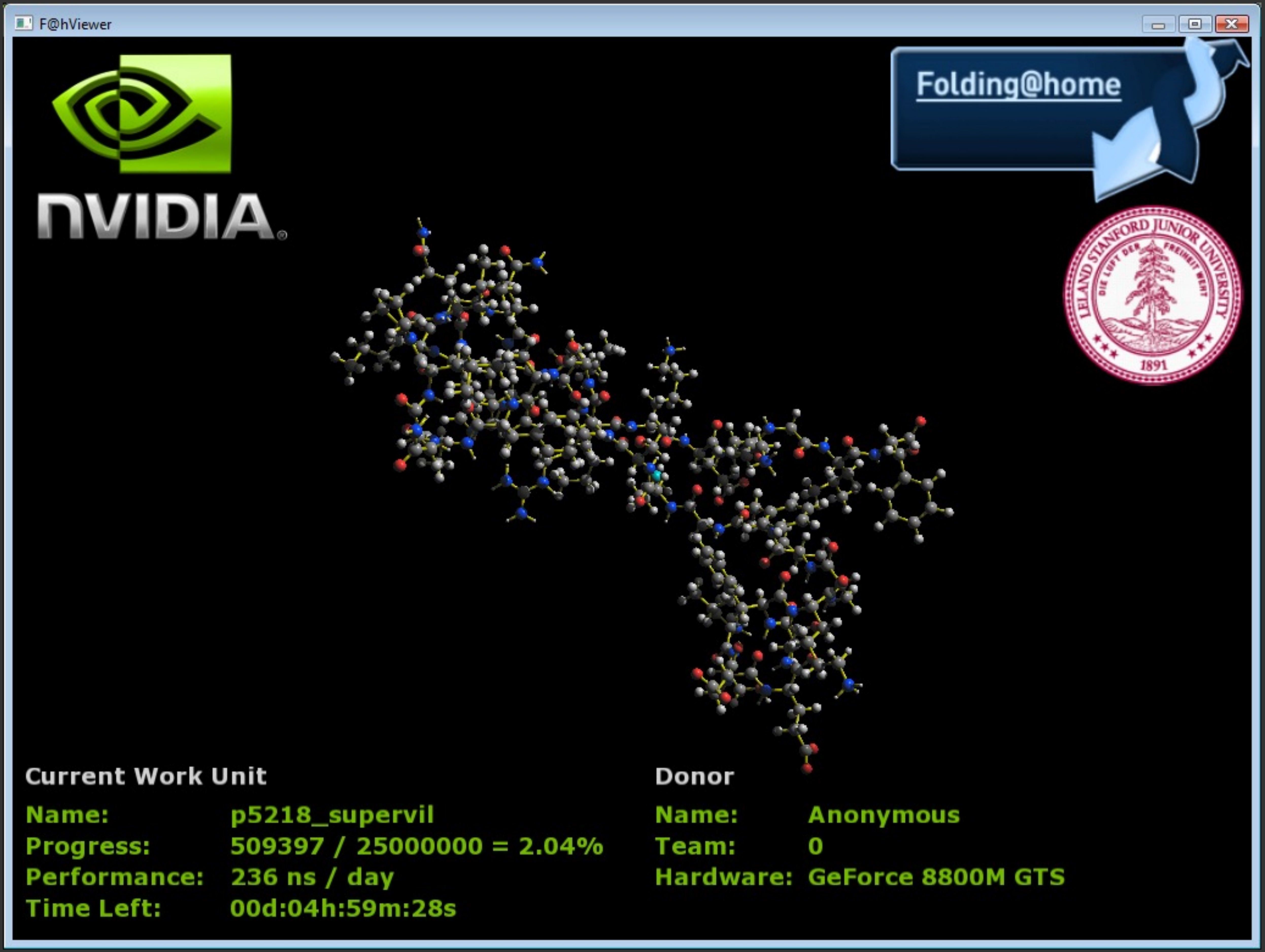
per-element output: RGBA surface color at pixel

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

Generalization beyond graphics: commodity parallel computing

Brook for GPUs (Buck 2004)

NVIDIA CUDA (2007)

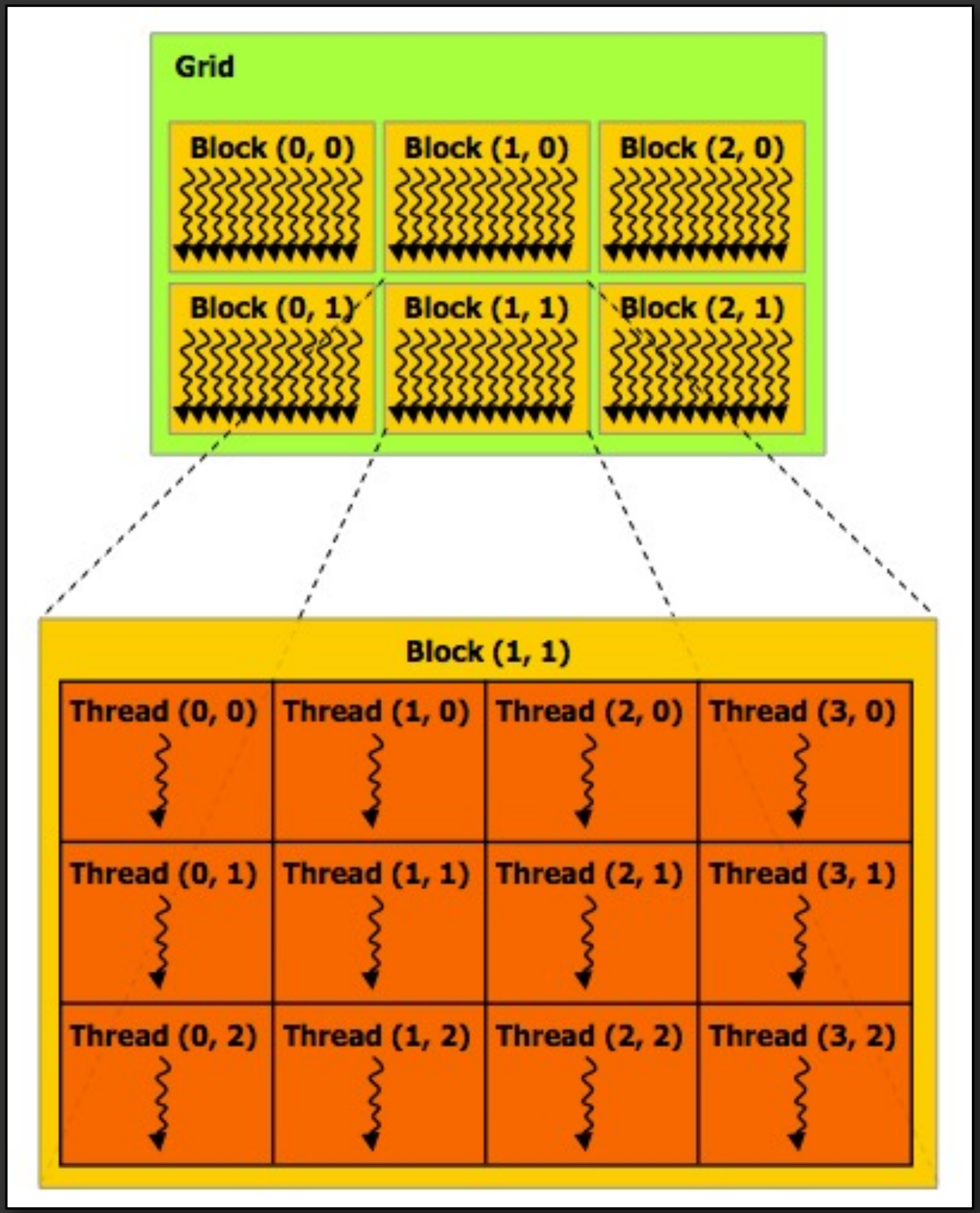


Current Work Unit

Name: p5218_supervil
Progress: 509397 / 25000000 = 2.04%
Performance: 236 ns / day
Time Left: 00d:04h:59m:28s

Donor

Name: Anonymous
Team: 0
Hardware: GeForce 8800M GTS



Goals of visual computing (until recently)

Modeling the real-world in increasingly rich detail: so we can simulate it (“render everything you’ve ever seen”)

Depict and organize information to augment human thought: enable humans to effectively use computing to create/analyze/interpret/communicate

Key characteristics of visual computing

Requires exceptional levels of efficiency

- Applications turn more ops/watt into new value
- Pack chips full of ALUs (parallel, heterogeneity/specialization are fundamental)
- Applications utilize hardware pipelines very well

Embrace domain-specific programming frameworks

- Achieve high efficiency/productivity
- Today: OpenGL, Halide, game engine frameworks, deep learning frameworks

Aspects of computation are fundamentally approximate

- Manifests as willingness to change algorithms (not approximate HW)

Visual computing — what’s next?

Goals of visual computing (present — future)

To capture everything that can be seen

To enable humans to communicate more effectively

To record and analyze the world's visual information so that computers can understand and reason about it

The immediate future: capturing rich visual information to enhance communication

Capturing pixels to communicate

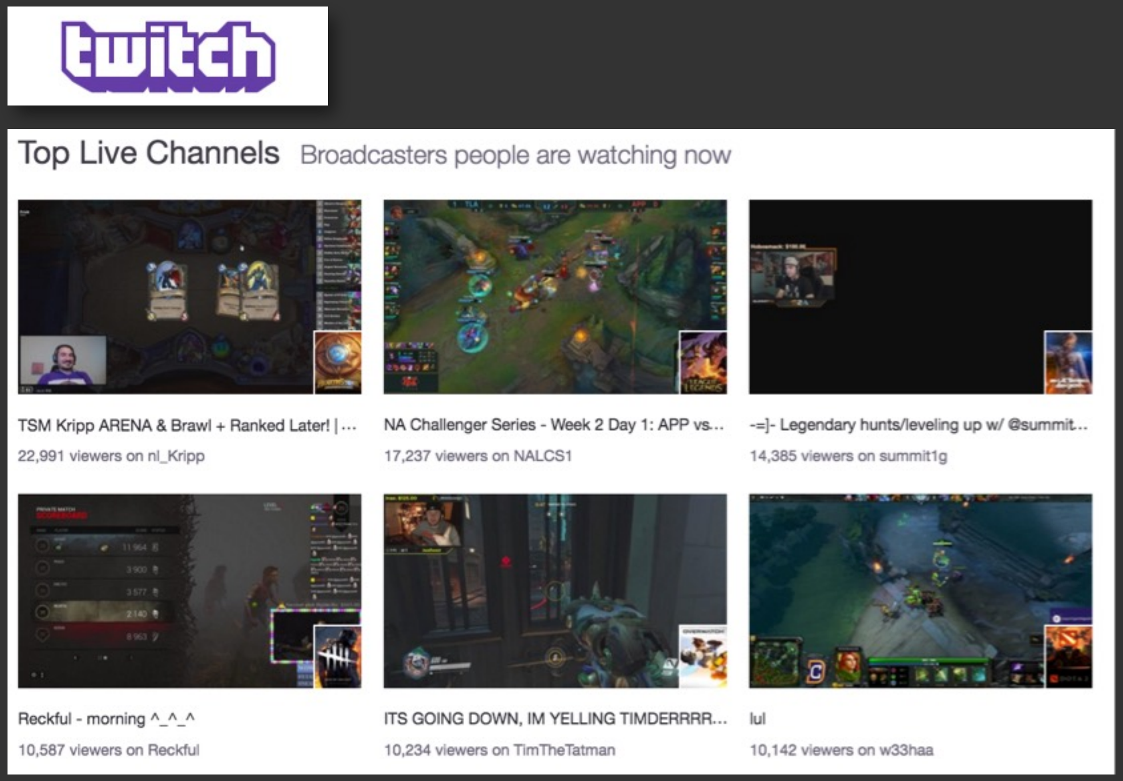
Ingesting/serving
the world's photos

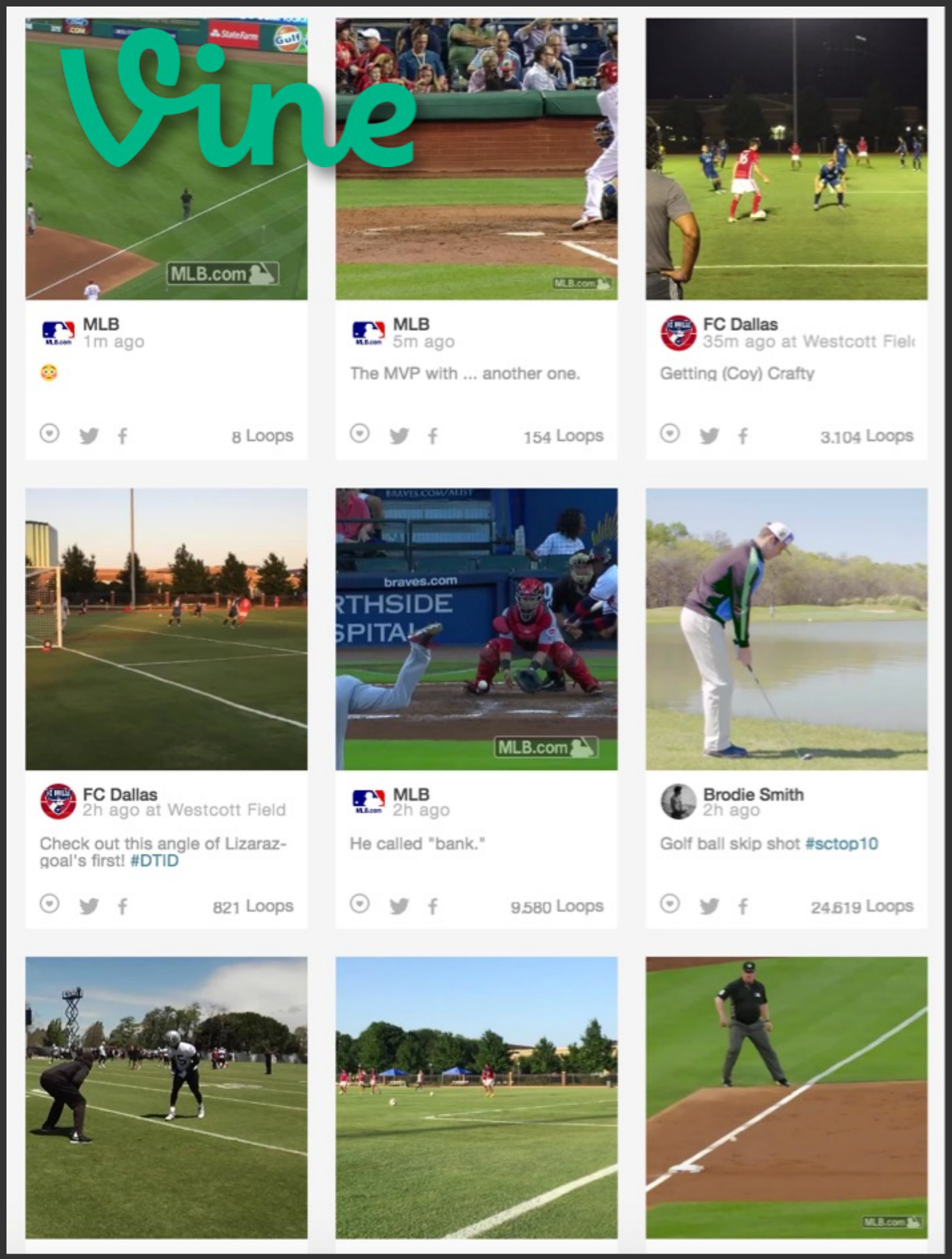
2B photo uploads and shares per day across Facebook sites (incl. Instagram+WhatsApp) [FB2015]

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)

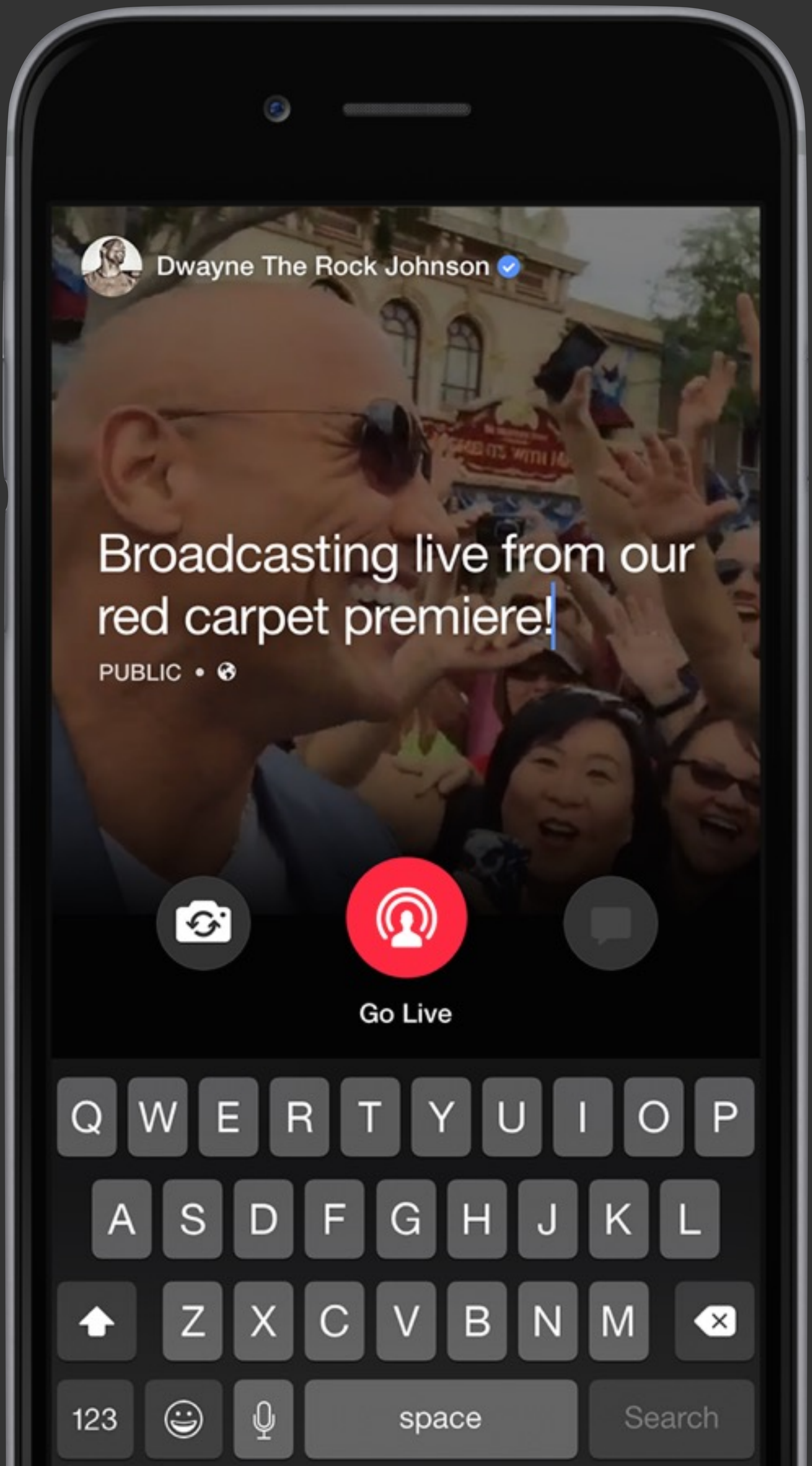

twitch
Top Live Channels Broadcasters people are watching now
TSM Kripp ARENA & Brawl + Ranked Later! | ...
22,991 viewers on nl_Kripp
NA Challenger Series - Week 2 Day 1: APP vs...
17,237 viewers on NALCS1
-=]- Legendary hunts/leveling up w/ @summit...
14,385 viewers on summit1g
Reckful - morning ^_^_^
10,587 viewers on Reckful
ITS GOING DOWN, IM YELLING TIMDERRRR...
10,234 viewers on TimTheTatman
lul
10,142 viewers on w33haa

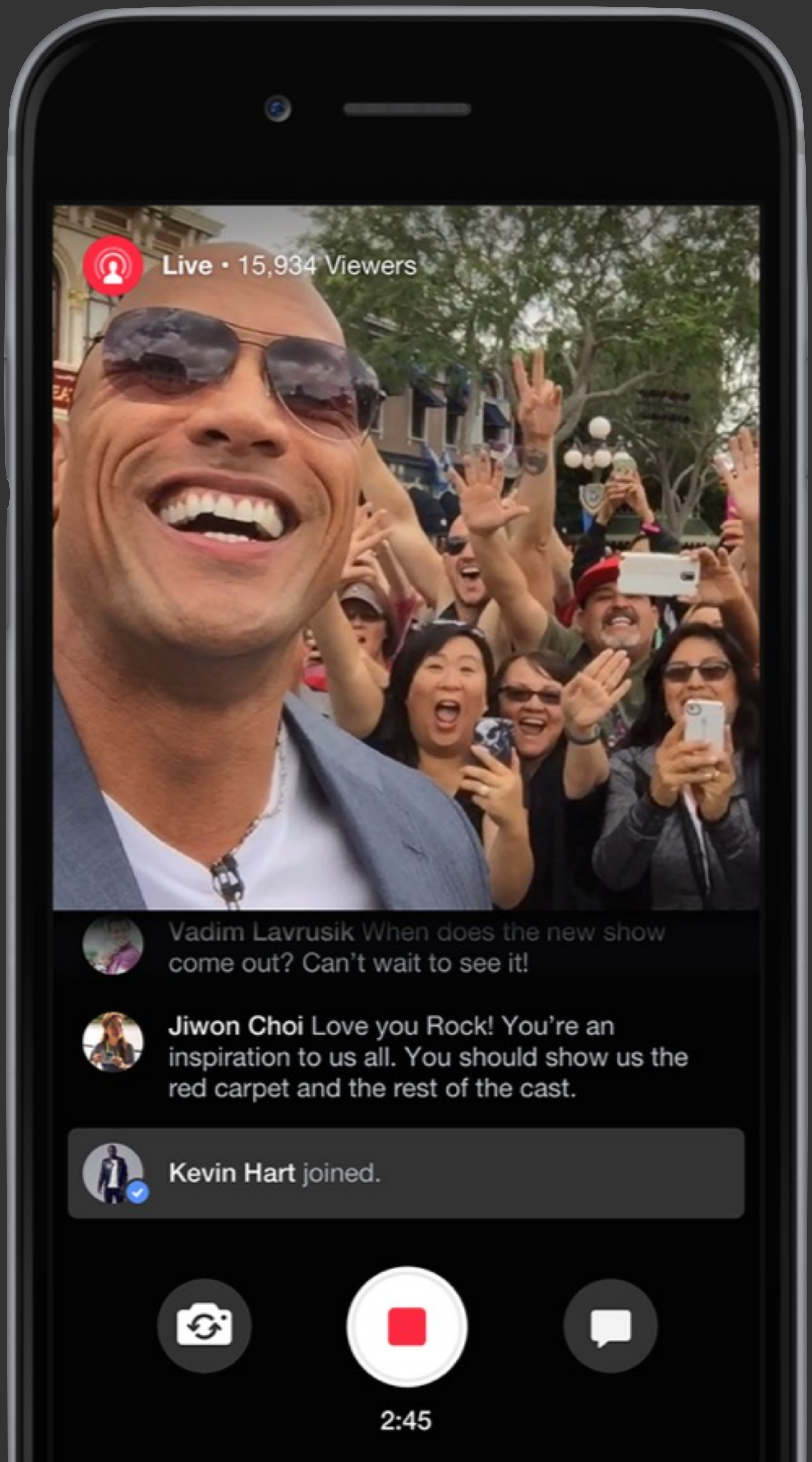


Richer content: beyond a single image

- 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



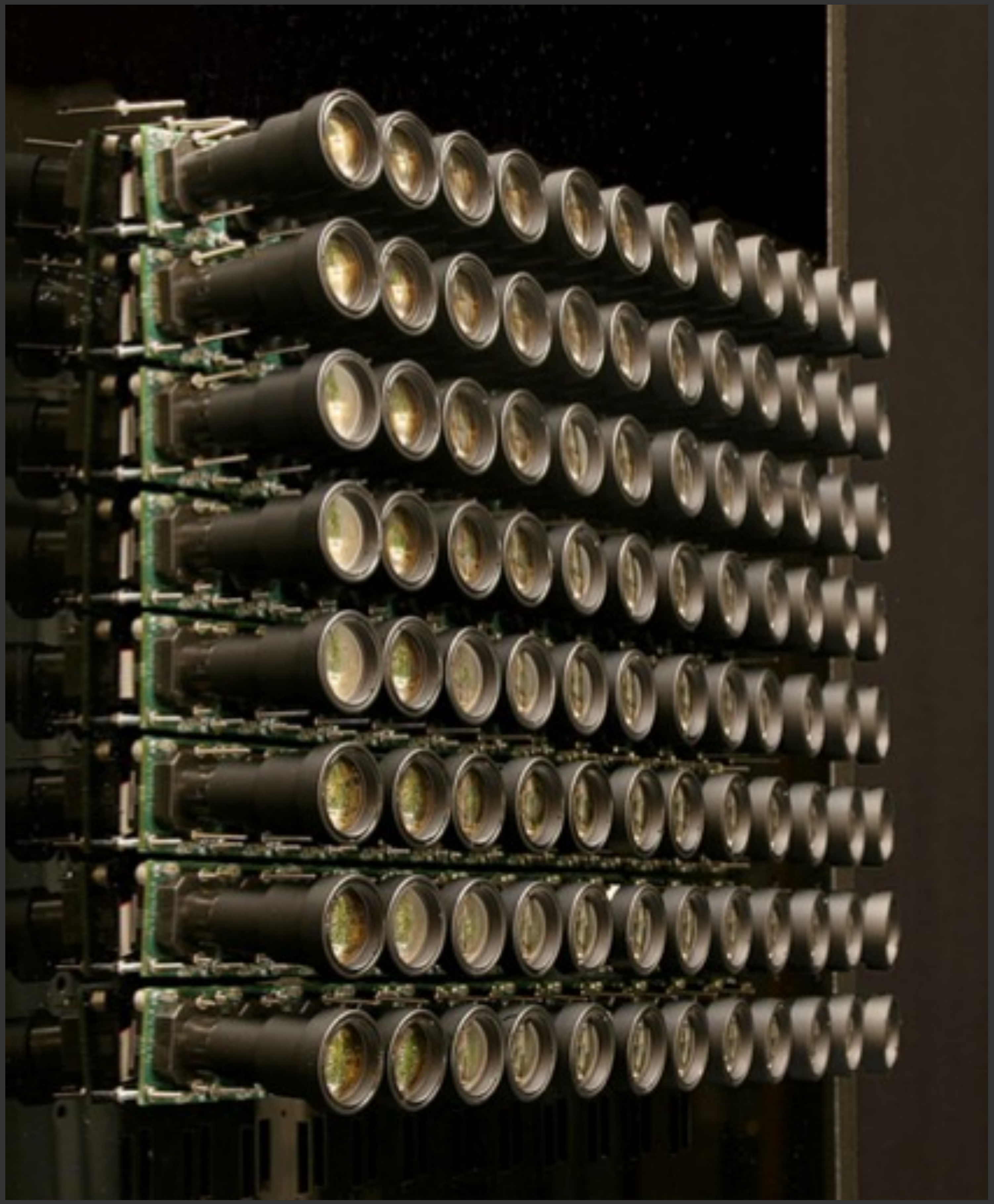


Vadim Lavrusik When does the new show come out? Can't wait to see it!

Jiwon Choi Love you Rock! You're an inspiration to us all. You should show us the red carpet and the rest of the cast.

Kevin Hart joined.

Acquiring richer content: light fields



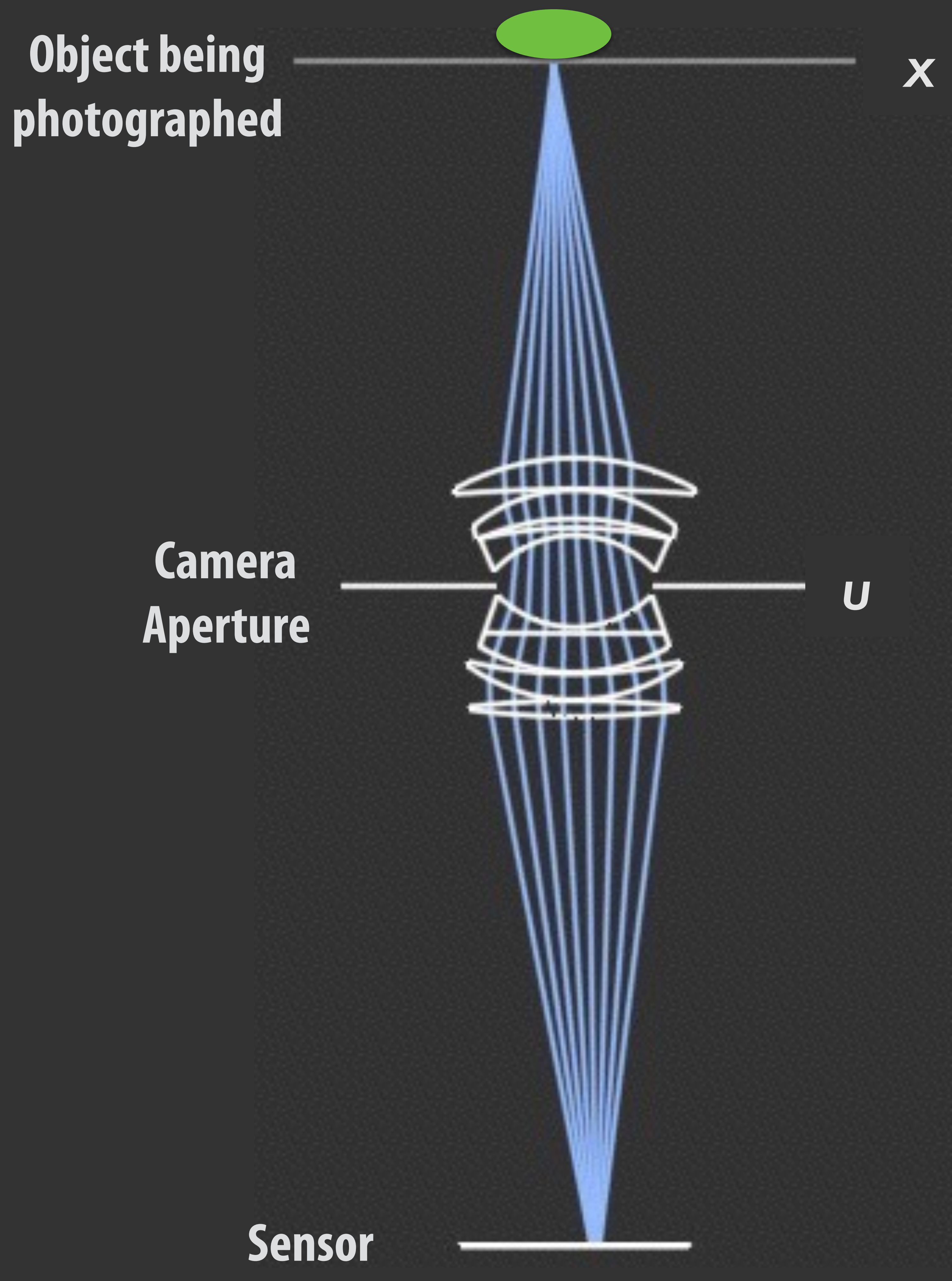
Stanford camera array
Wilburn [2005]

Richer content: light fields

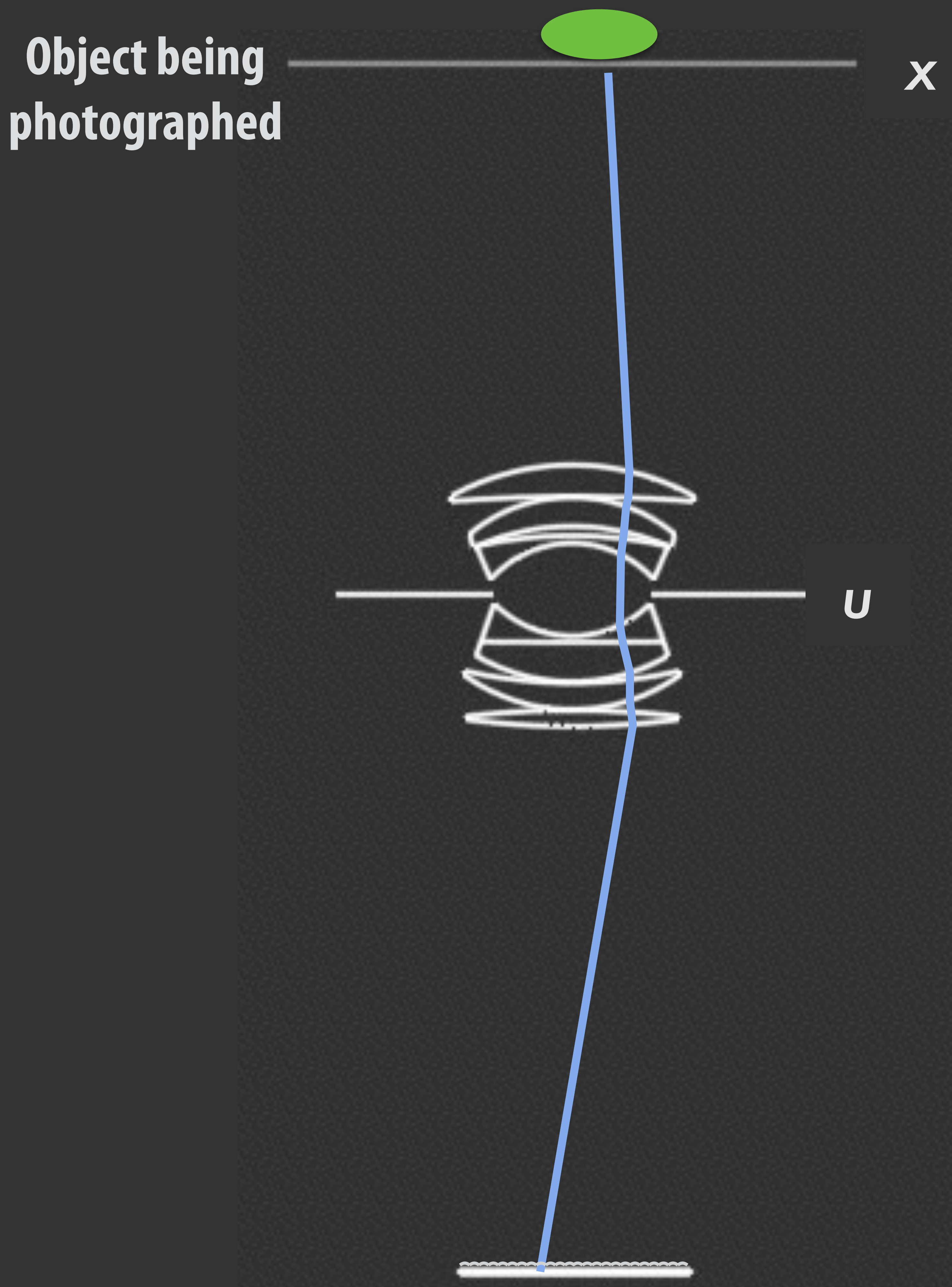
Light L16

Lytro Illum

Light field camera: capturing a light field



2D traditional camera:
measures how much light hits a point on sensor



"4D" light field camera:
measures how much light hits point on sensor **from a particular direction**

[Slide courtesy Ren Ng]

[Slide courtesy Ren Ng]

[Slide courtesy Ren Ng]

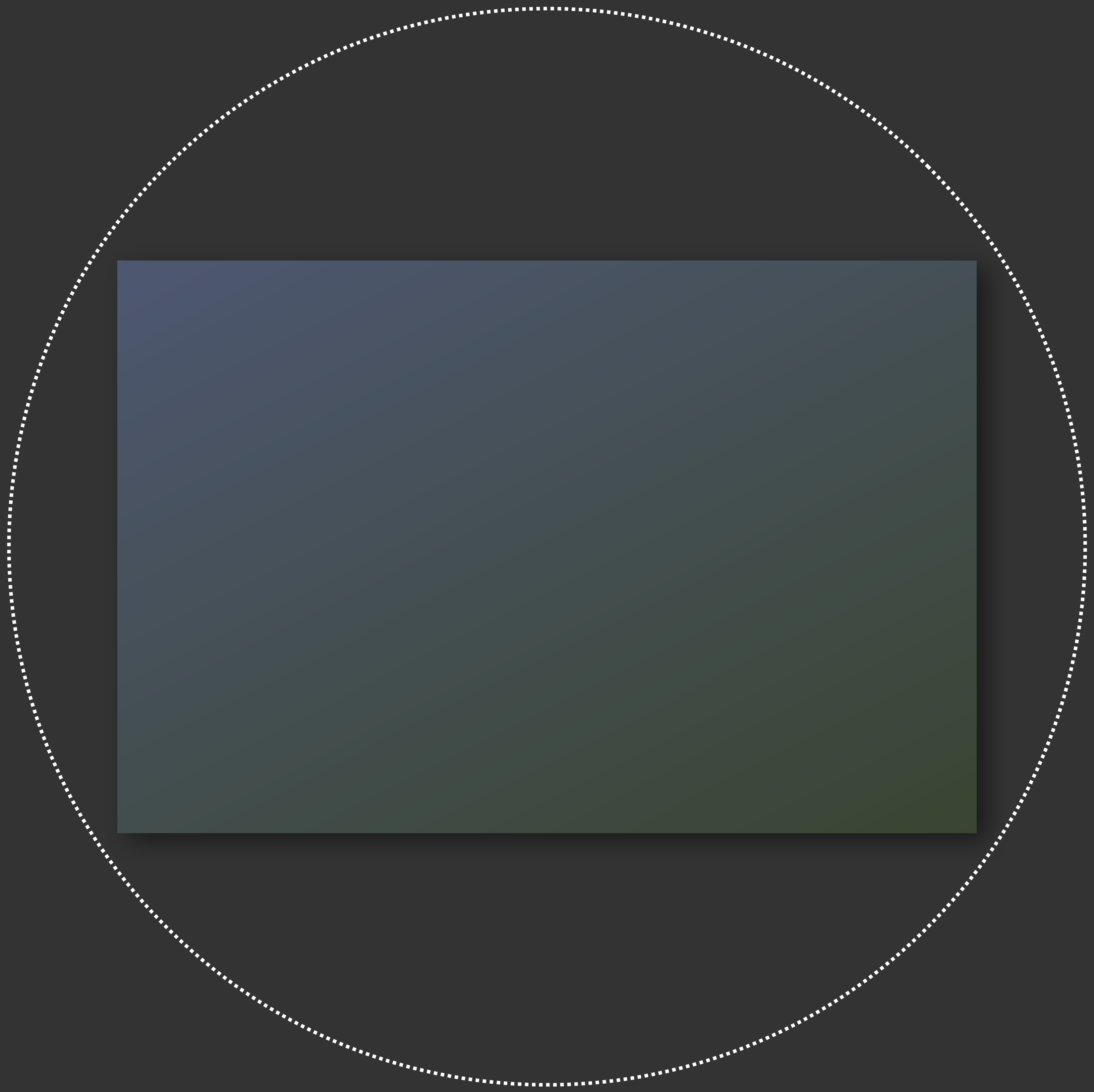
[Slide courtesy Ren Ng]

[Slide courtesy Ren Ng]

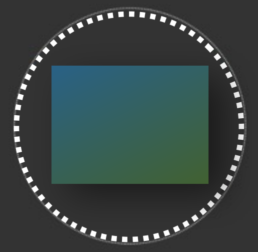
[Slide courtesy Ren Ng]

[Slide courtesy Ren Ng]

Sensor industry has large untapped resolution



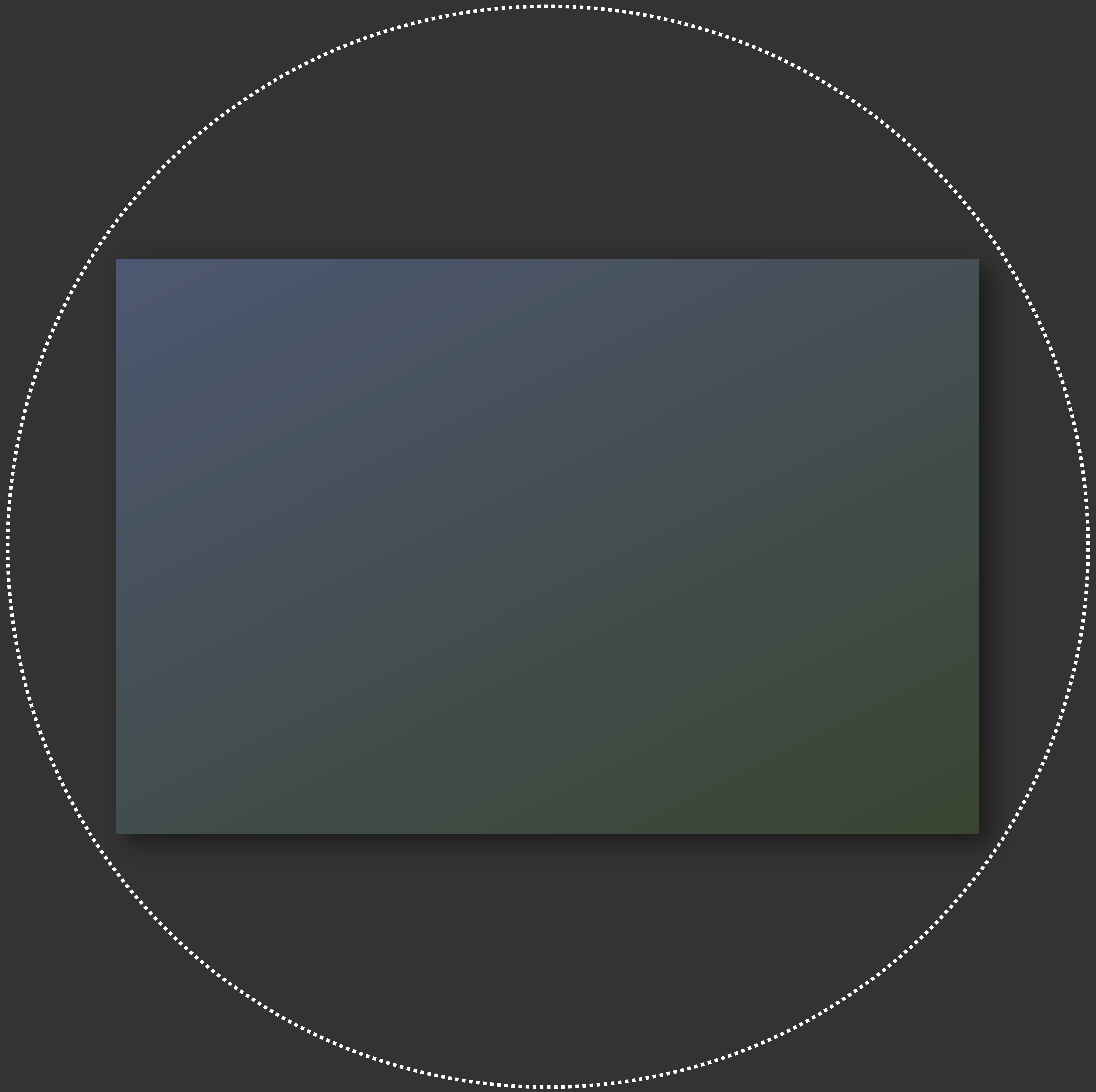
Full-Frame Sensor
36 x 24 mm
Up to 36 MP
4.9 micron pixel

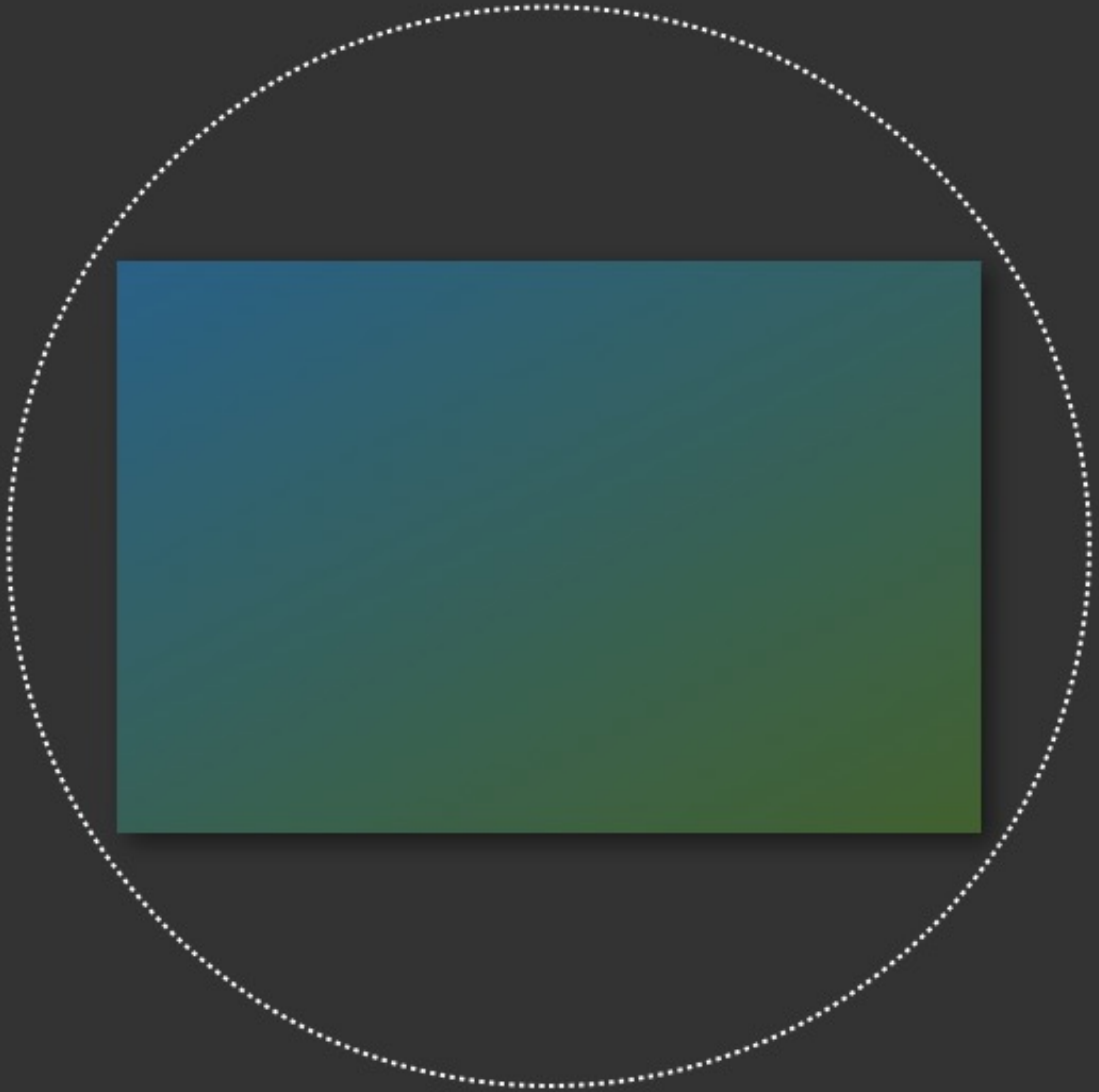


1/3" Sensor
4.8 x 3.6 mm
Up to 13 MP
1.12 micron pixel

[Slide courtesy Ren Ng]

Sensor industry has large untapped resolution


Full-Frame Sensor
36 x 24 mm
Up to 36 MP
4.9 micron pixel


Full-Frame Sensor
36 x 24 mm
688 MP
1.12 micron pixel

[Slide courtesy Ren Ng]

Lytro Cinema

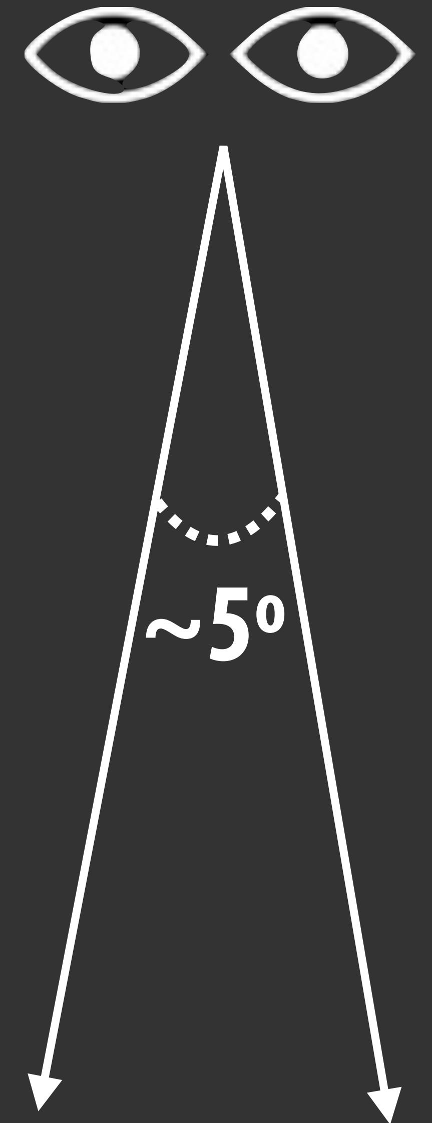
755 Mpixel camera

VR output

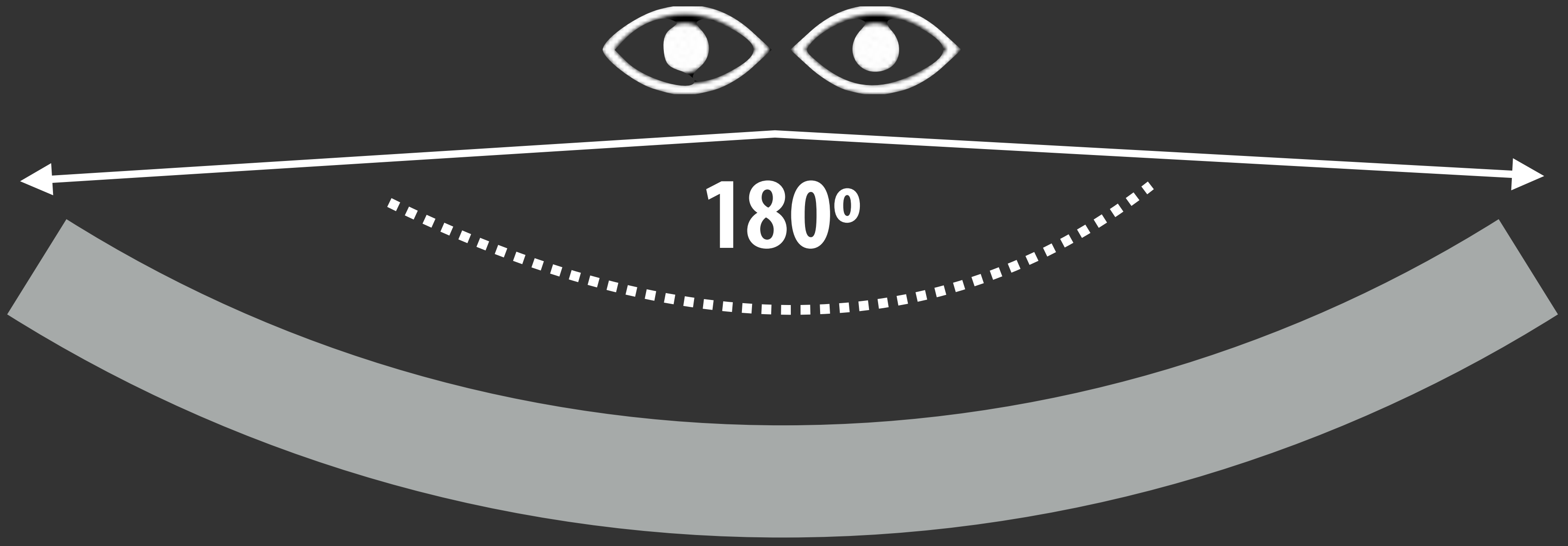
Example: Google's JumpVR video
Input stream: 16 4K GoPro cameras

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

VR creates high resolution requirements



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

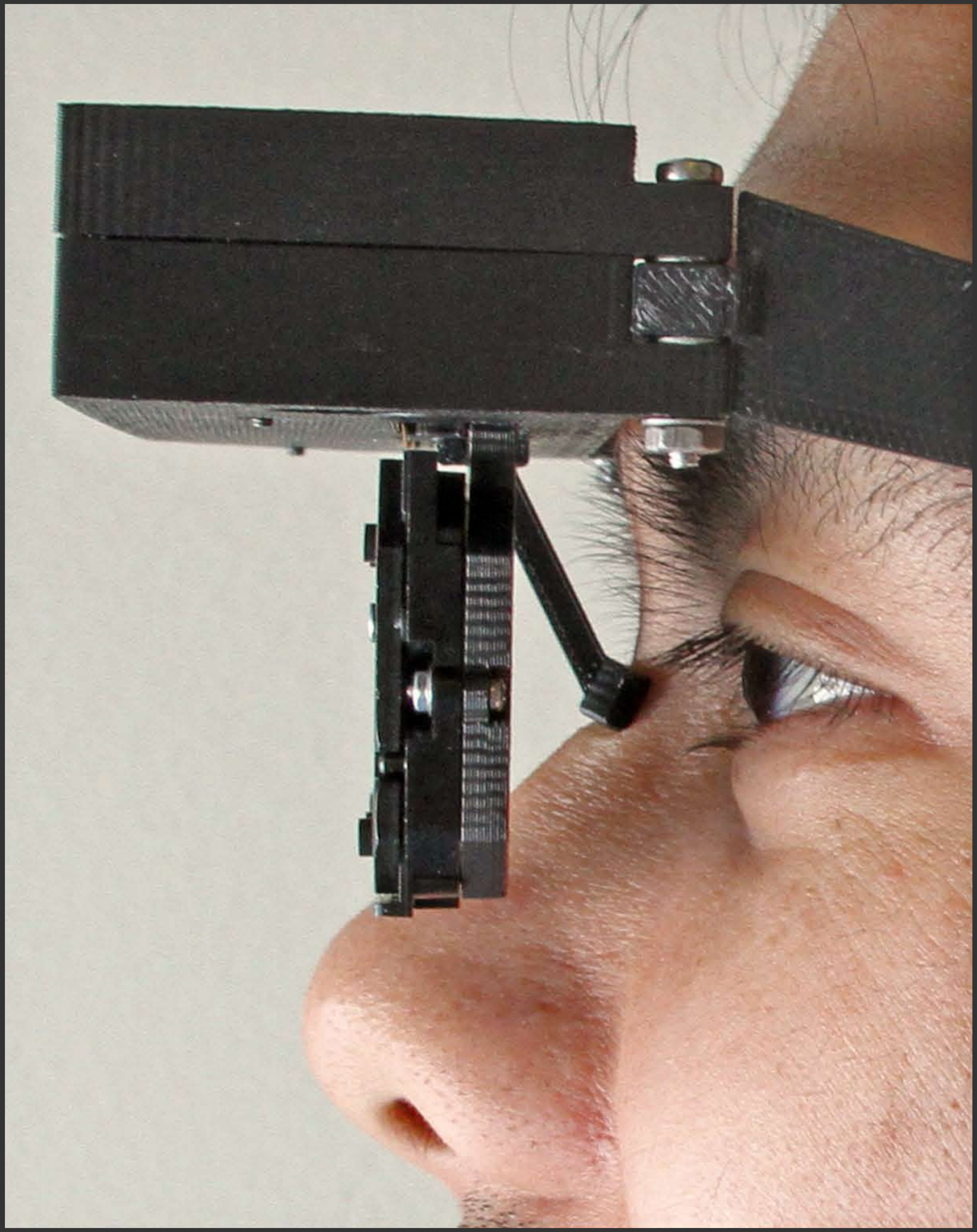


Future "retina" VR display:
57 ppd covering 180°
= 10K x 10K display per eye
= 200 MPixel

RAW data rate @ 120Hz ≈ 72 GB/sec

[Lanman 2013]

VR: Light field display

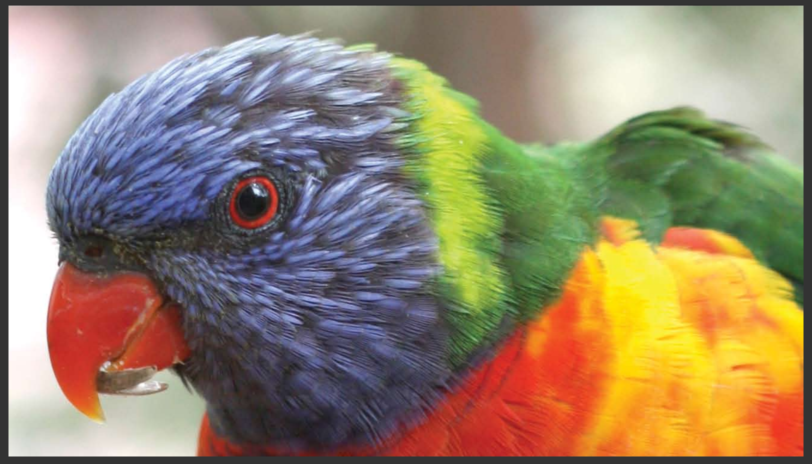


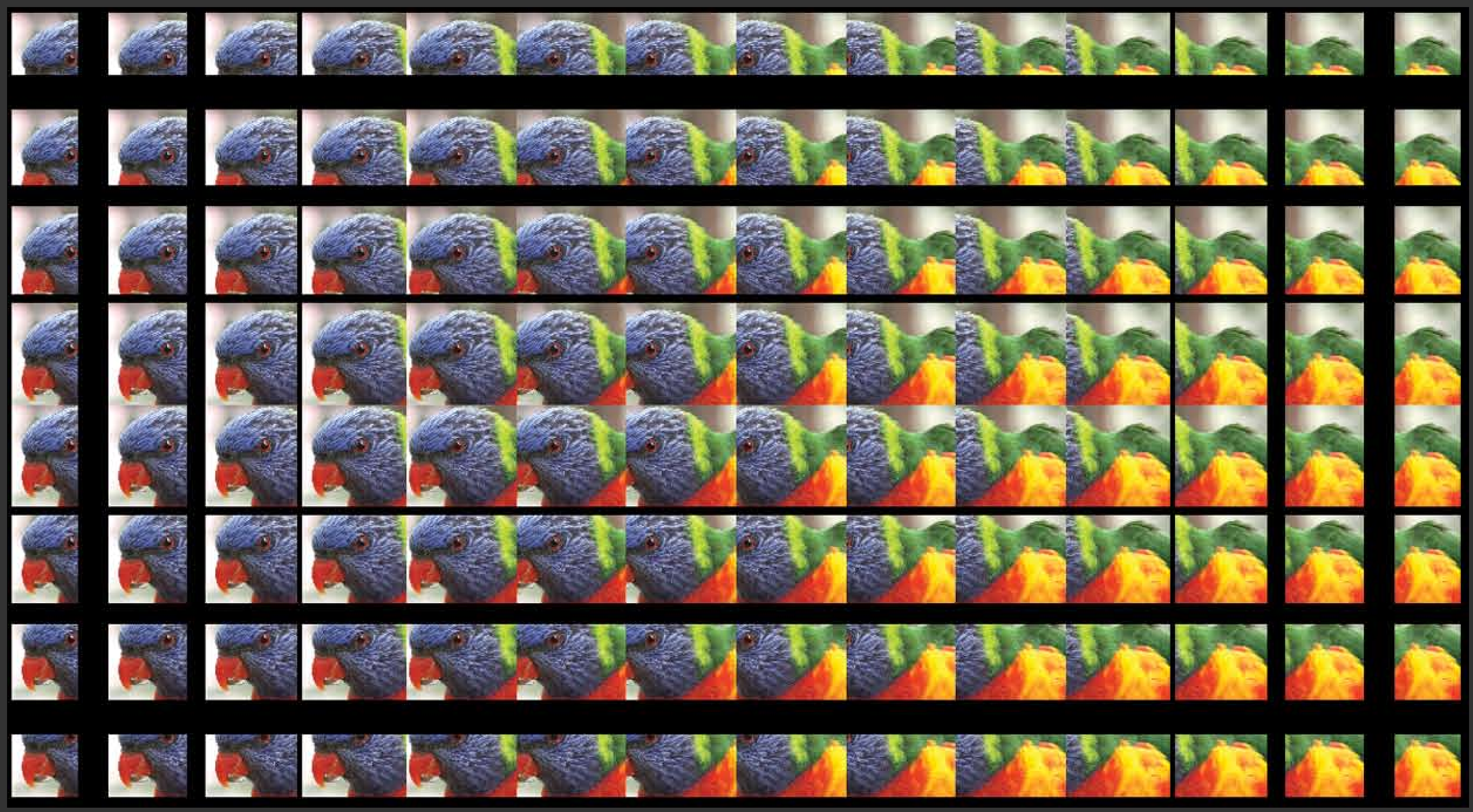
146 x 78 spatial resolution

Using 1MP microdisplay

Simple idea:

Recreate the same light field that was present in the scene when it was captured



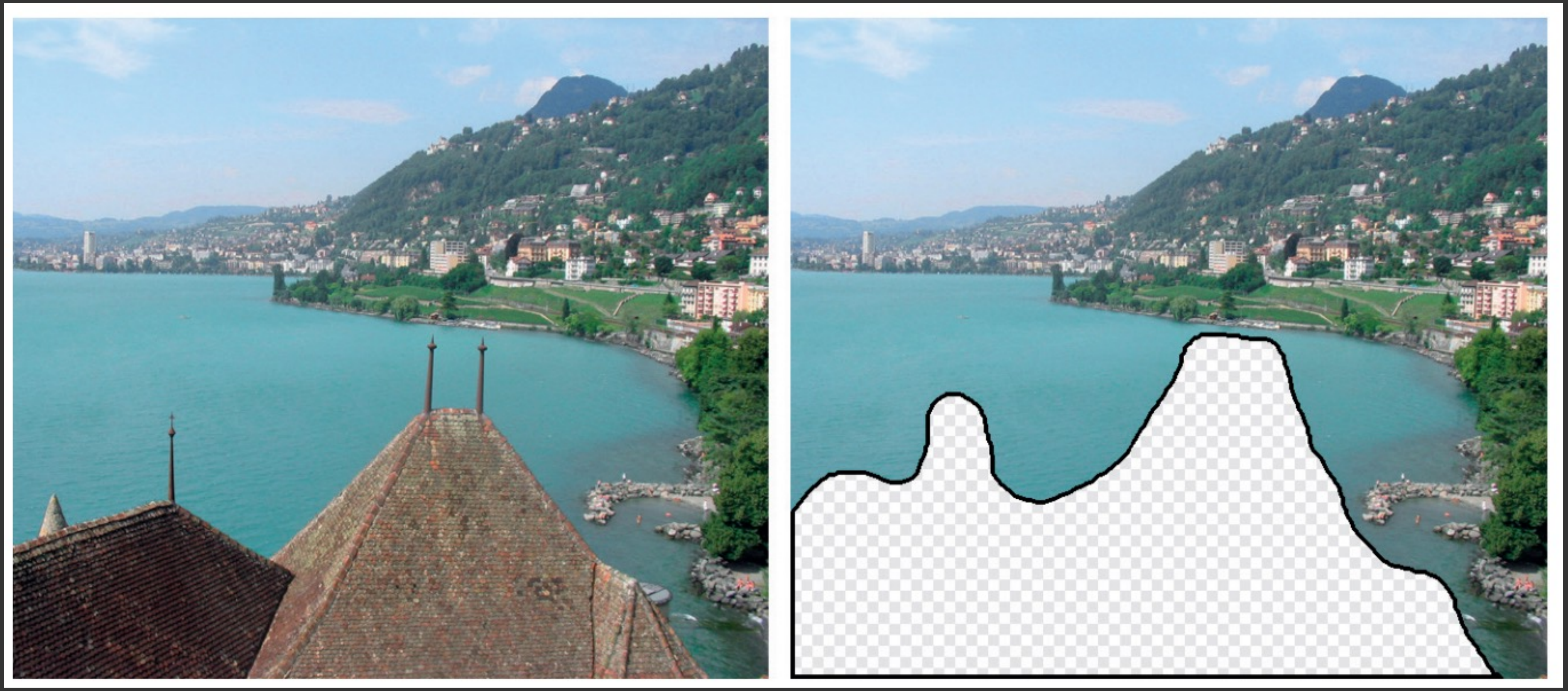


Output of display (prior to optics)

Enhancing communication: understanding images to improve acquired content

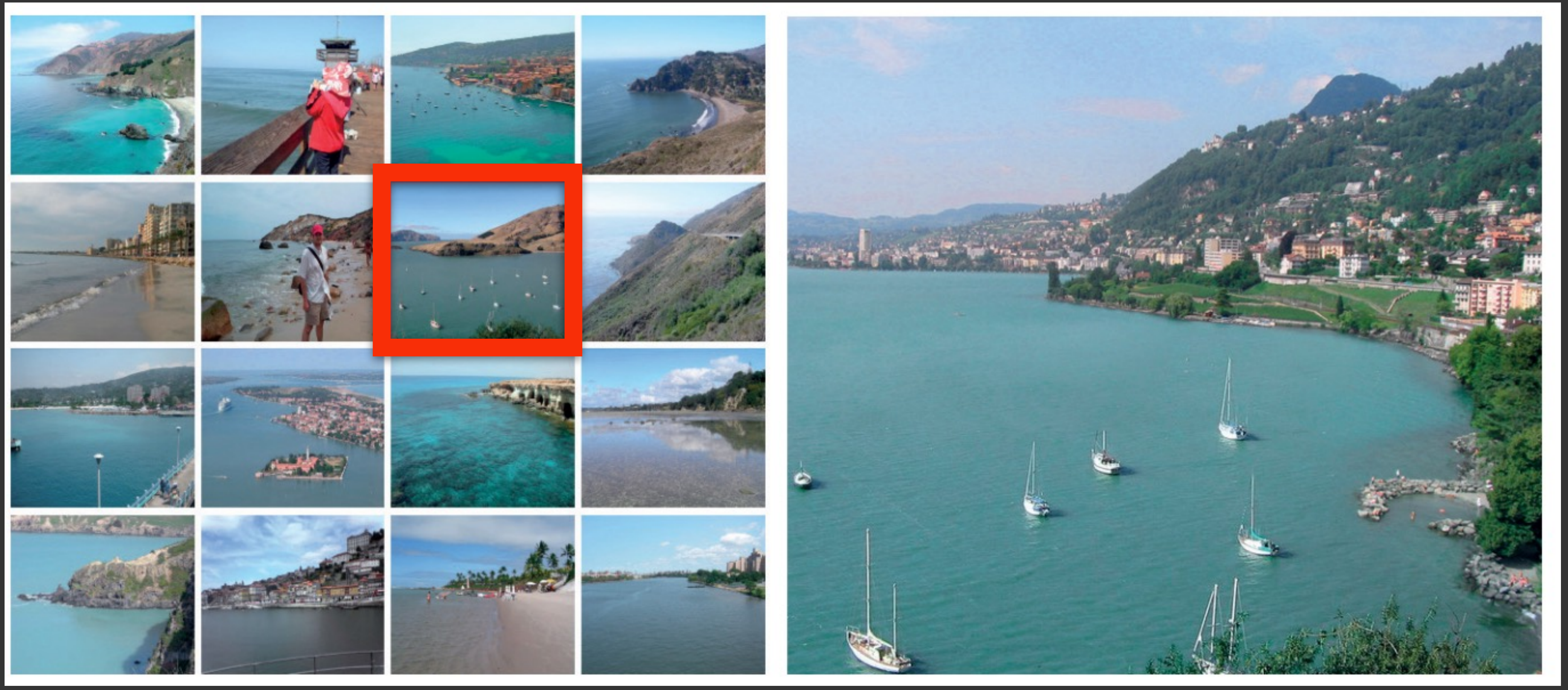
AutoEnhance:

Photo "fix up" [Hayes 2007]



My bad vacation photo

Part to fix



Similar photos others have taken

Fixed!

Summary

We are observing rapid growth in the richness of visual communication

Sensing the world with higher fidelity to deliver improved content to humans

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

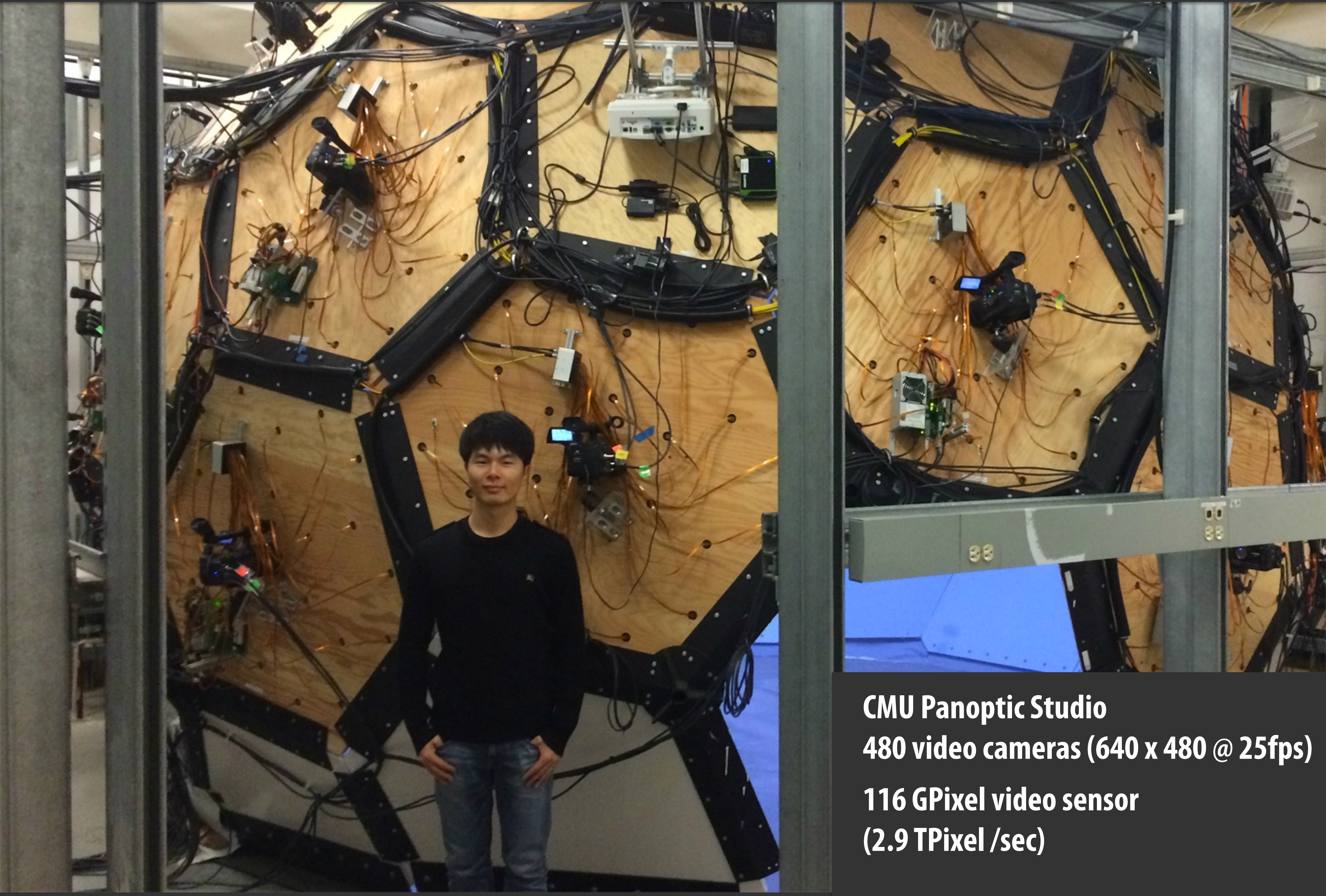
Continuous (always on)

Exceptionally high resolution

Capture for computers to analyze, not humans to watch

Sensing human social interactions

[Joo 2015]

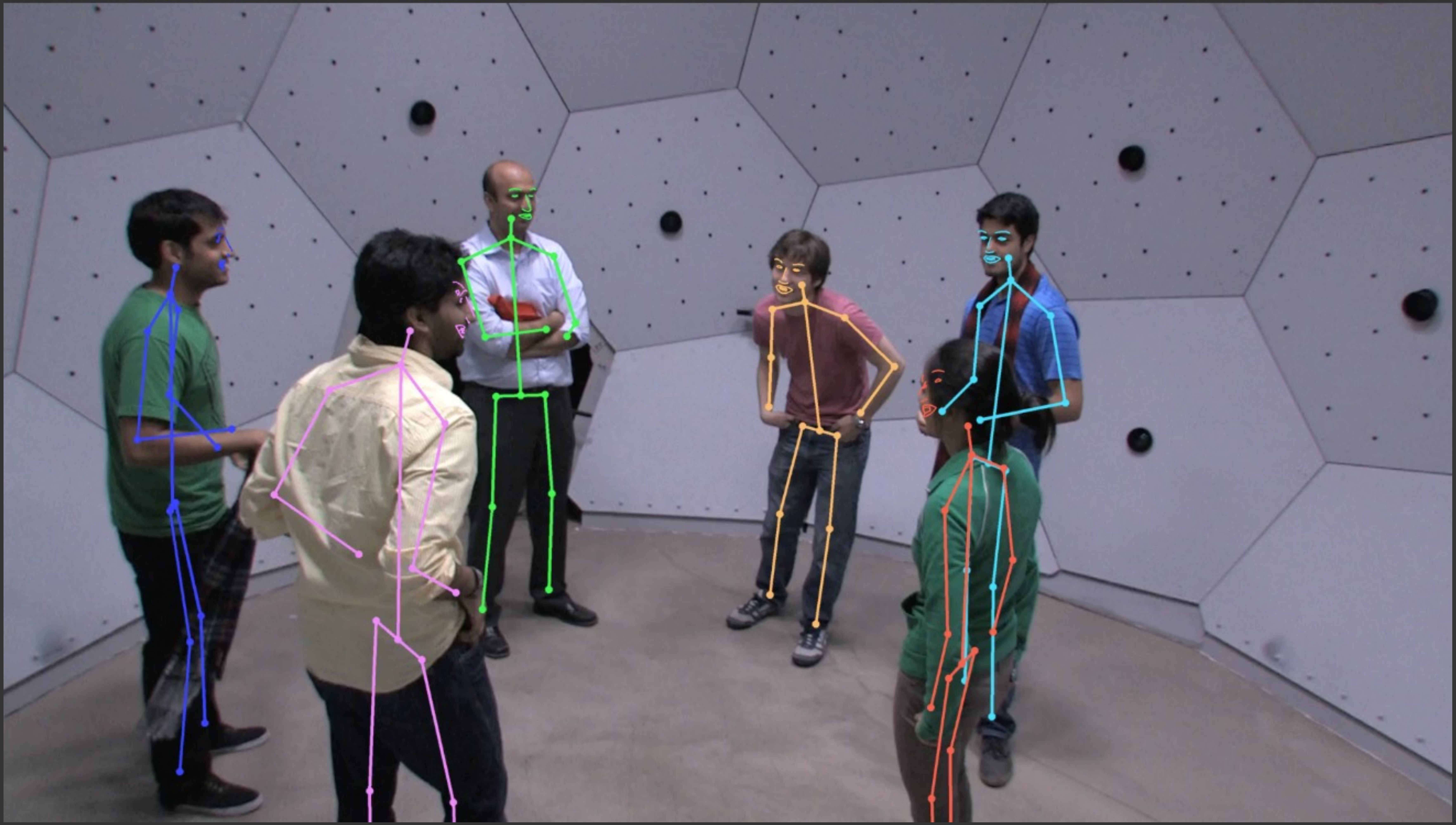


CMU Panoptic Studio

480 video cameras (640 x 480 @ 25fps)

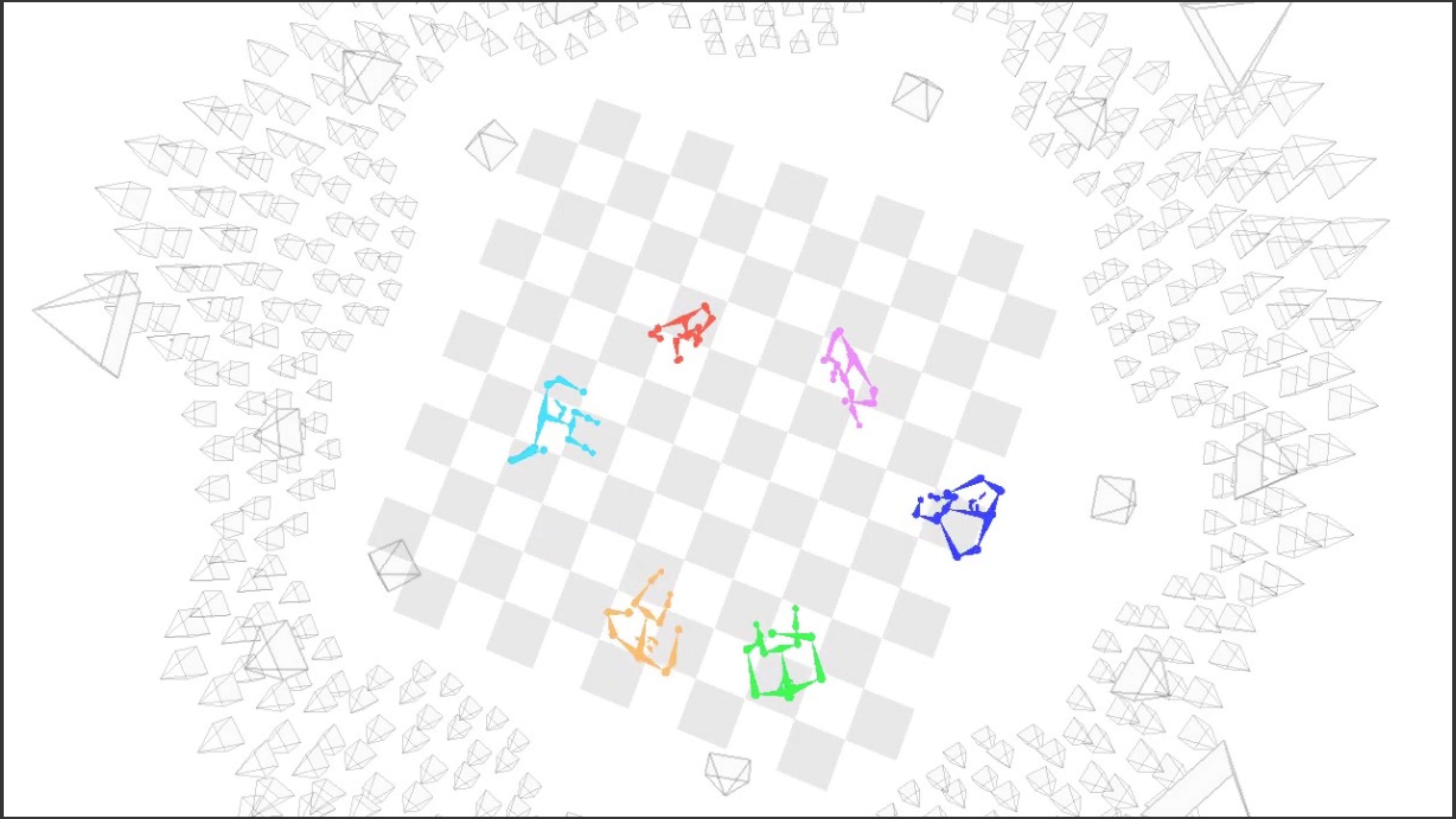
116 GPixel video sensor
(2.9 TPixel /sec)

Capturing social interactions



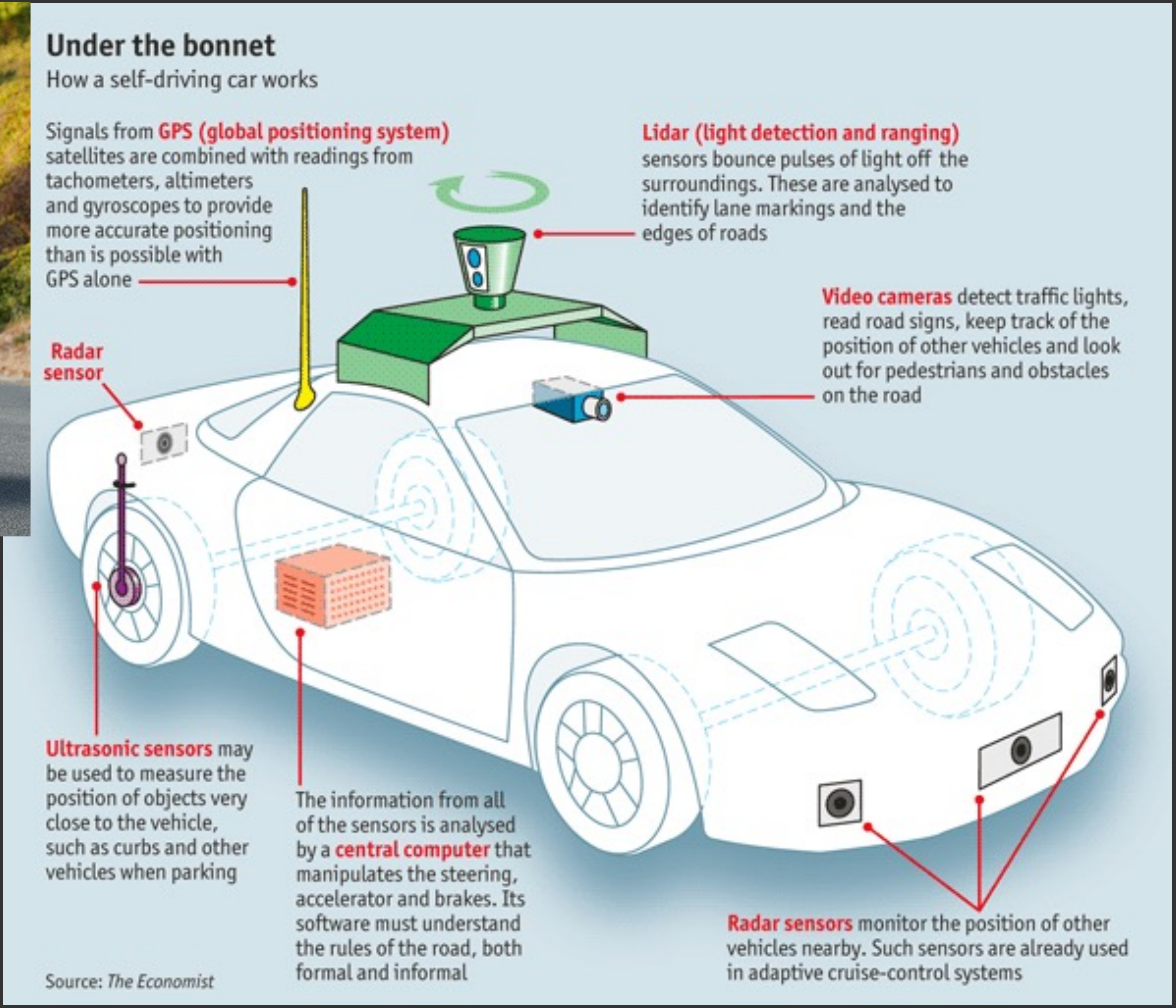
[Courtesy Yaser Sheikh, Tomas Simon, Hanbyul Joo]

Capturing social interactions



[Courtesy Yaser Sheikh, Tomas Simon, Hanbyul Joo]

Robot navigation depends on low-latency localization and surrounding object recognition



Under the bonnet

How a self-driving car works

Signals from **GPS (global positioning system)** satellites are combined with readings from tachometers, altimeters and gyroscopes to provide more accurate positioning than is possible with GPS alone

Lidar (light detection and ranging) sensors bounce pulses of light off the surroundings. These are analysed to identify lane markings and the edges of roads

Video cameras detect traffic lights, read road signs, keep track of the position of other vehicles and look out for pedestrians and obstacles on the road

Radar sensor

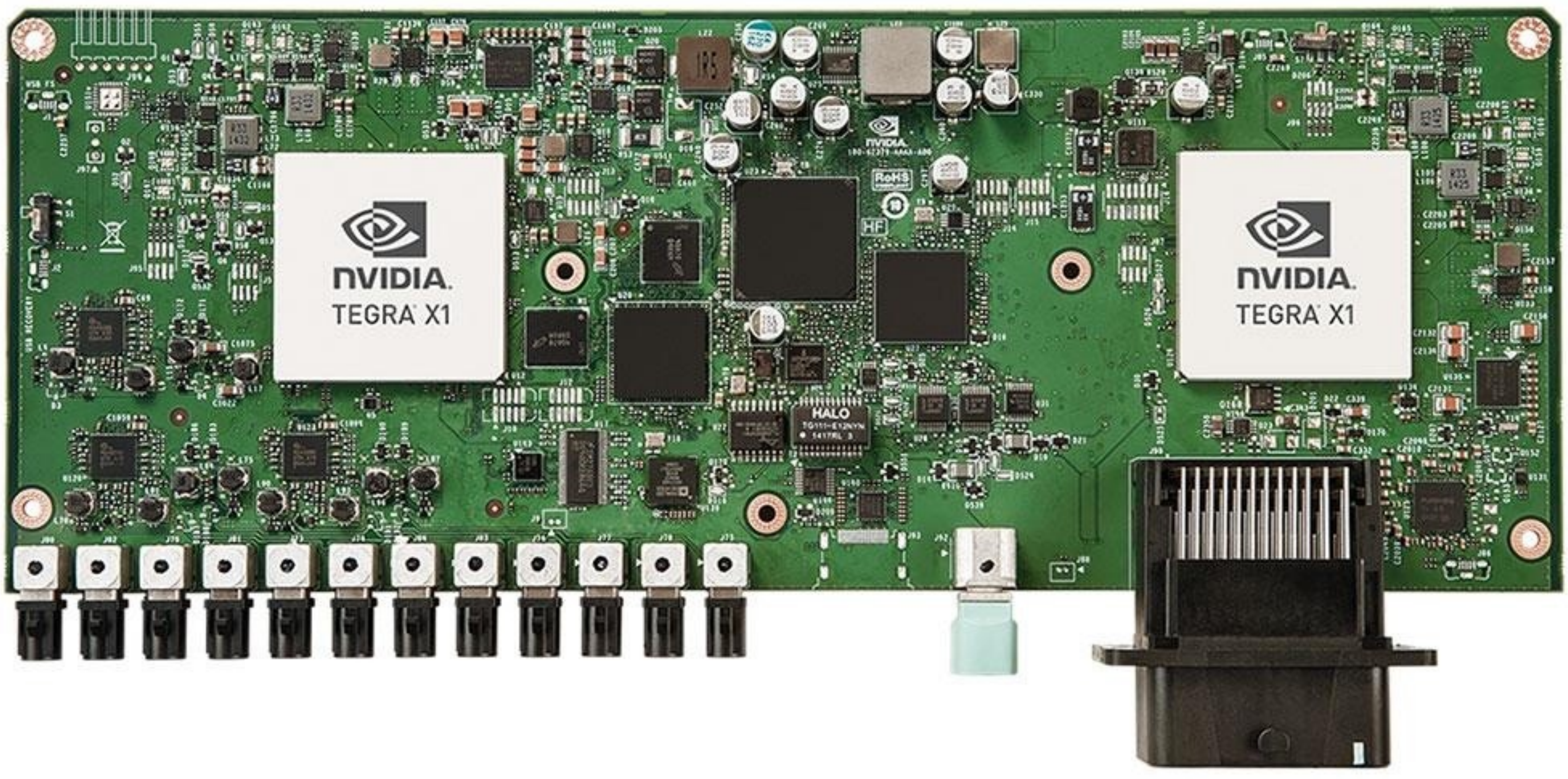
Ultrasonic sensors may be used to measure the position of objects very close to the vehicle, such as curbs and other vehicles when parking

The information from all of the sensors is analysed by a **central computer** that manipulates the steering, accelerator and brakes. Its software must understand the rules of the road, both formal and informal

Radar sensors monitor the position of other vehicles nearby. Such sensors are already used in adaptive cruise-control systems

Source: *The Economist*

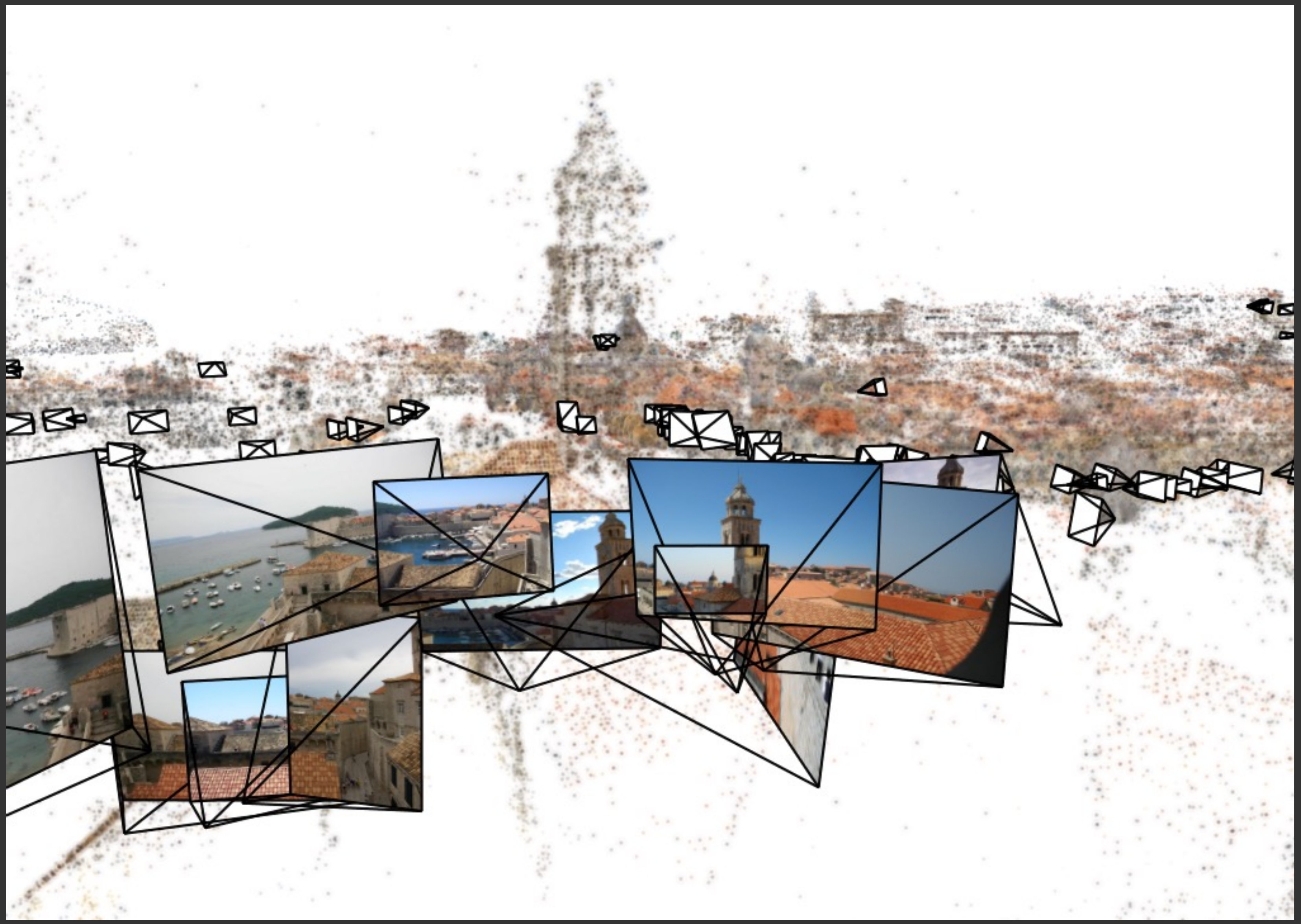
NVIDIA Drive PX



Tegra X1 (1 TFlop fp16 at 1GHz)

AR requires low-latency localization and scene object recognition

Making "maps": pervasive 3D construction

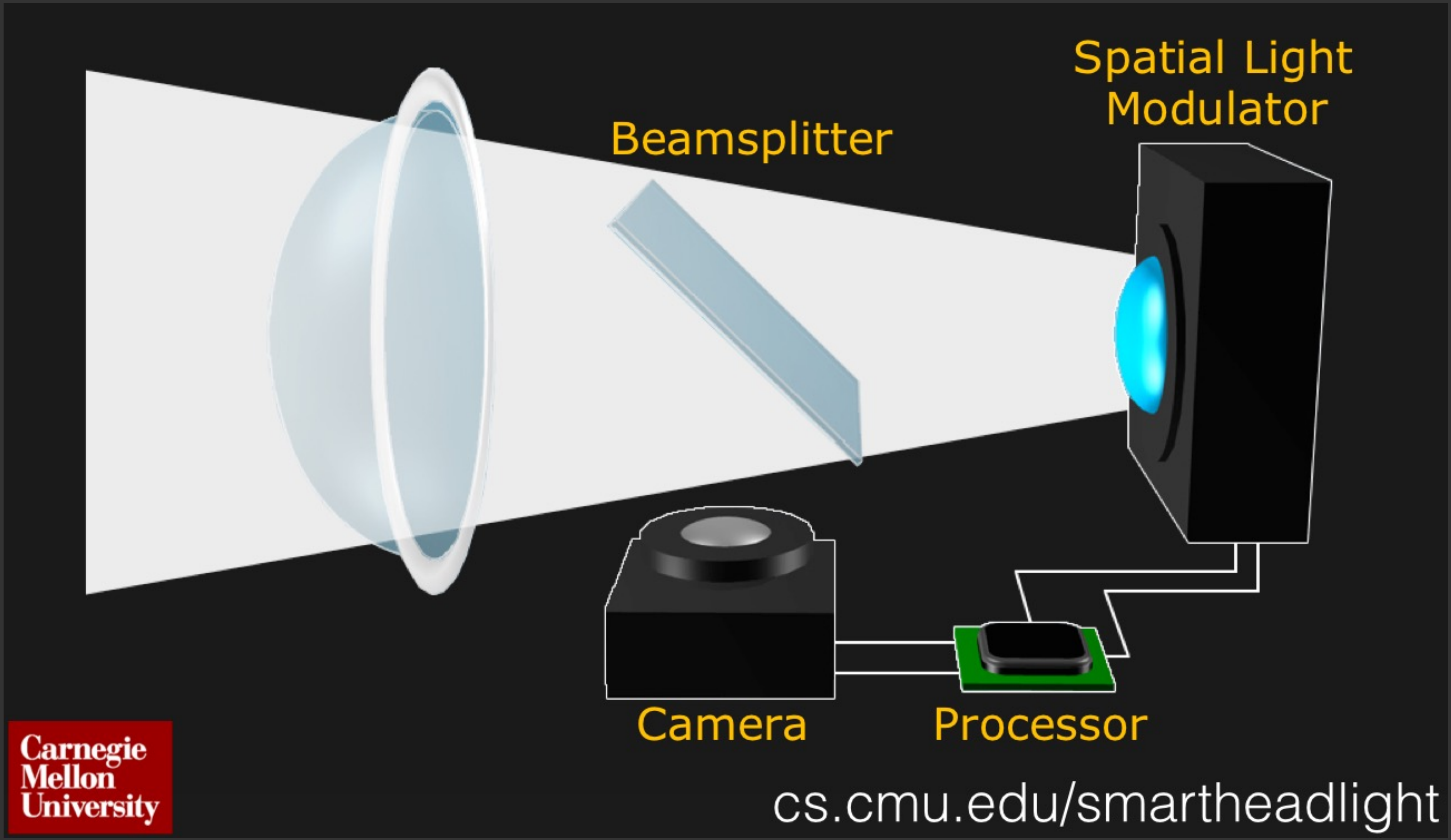


Carnegie Mellon University

cs.cmu.edu/smartheadlight

[Tamburo 2016]

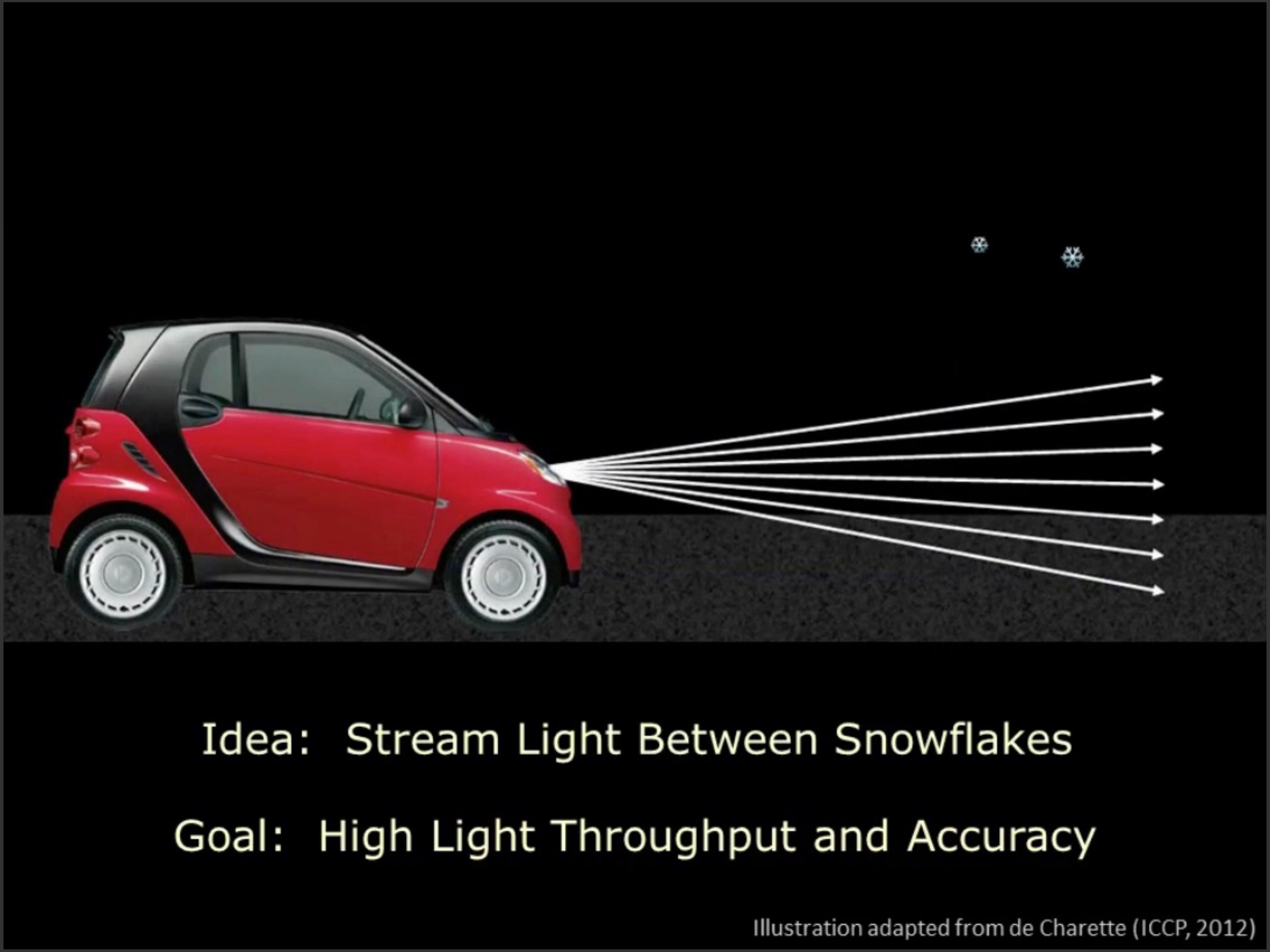
Smart headlight system



~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)

Urban video command center

(Centro de Operações Preifetura do Rio de Janeiro)

Overview summary

- Visual computing has always involved a healthy interaction between architecture, programming systems, and algorithms
  - Domain focus has been exceptionally useful for vertical thought
  - Willing to throw out old and re-engineer software (new hardware enables programs that haven't been written yet!)
  - Architects should know the algorithms well, and influence them!

- Visual computing has always challenged computer systems by its desire to simulate/synthesize complex visual information

- Next 1-2 decades: interpreting the worldwide visual signal
  - Acquiring and modeling everything humans would see, to enable computers to interpret and analyze
  - We will continue to take every op (op/Watt) you can give us

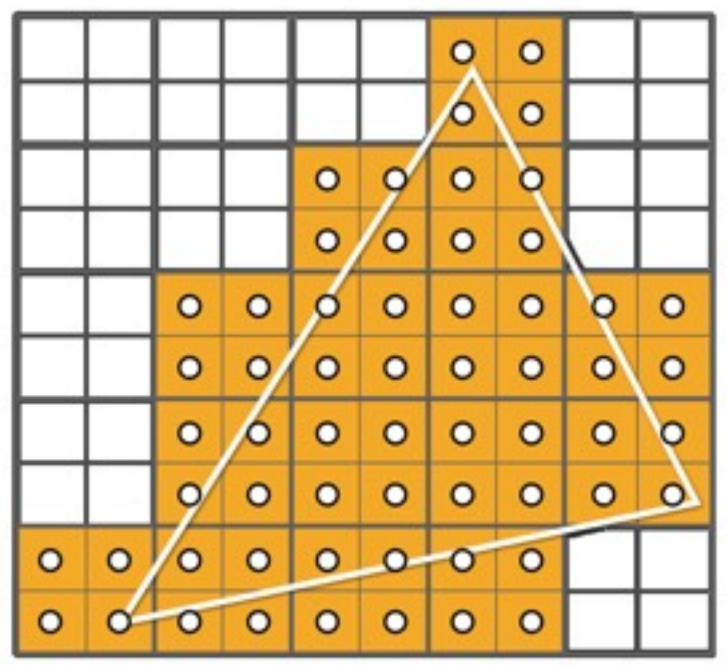
Course Logistics

What this course is about

1. The characteristics/requirements of important visual computing workloads
2. Techniques used to achieve efficient system implementations

VISUAL COMPUTING WORKLOADS

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

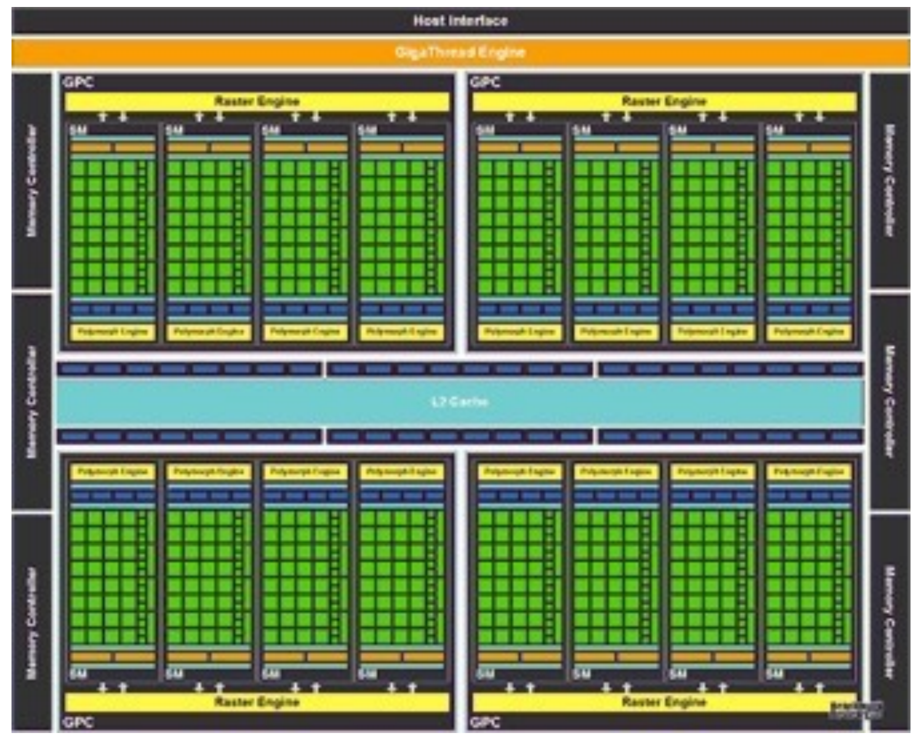


mapping/scheduling

$\longleftrightarrow$

Parallelism
Exploiting locality
Minimizing communication

MACHINE ORGANIZATION



High-throughput hardware designs: Parallel and heterogeneous

DESIGN OF GOOD ABSTRACTIONS FOR VISUAL COMPUTING

choice of programming primitives
level of abstraction

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 better algorithms

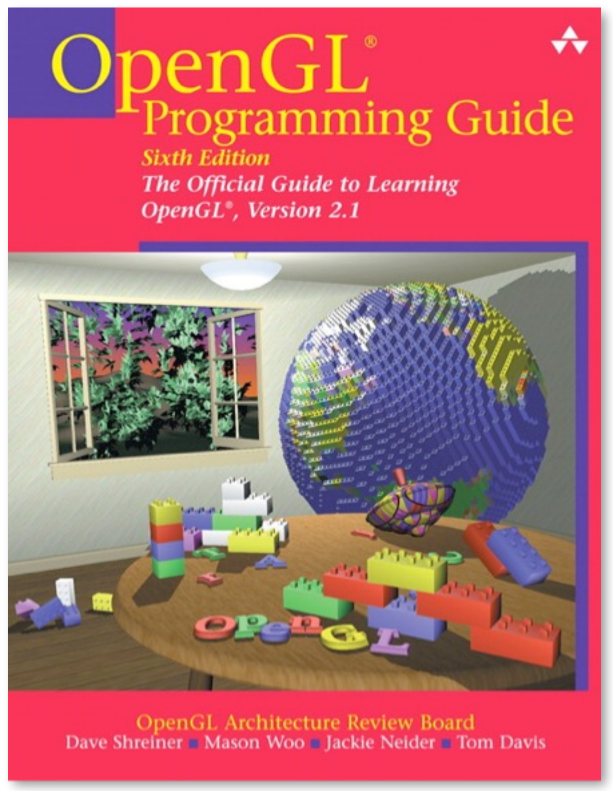
To build the most efficient hardware to run these applications

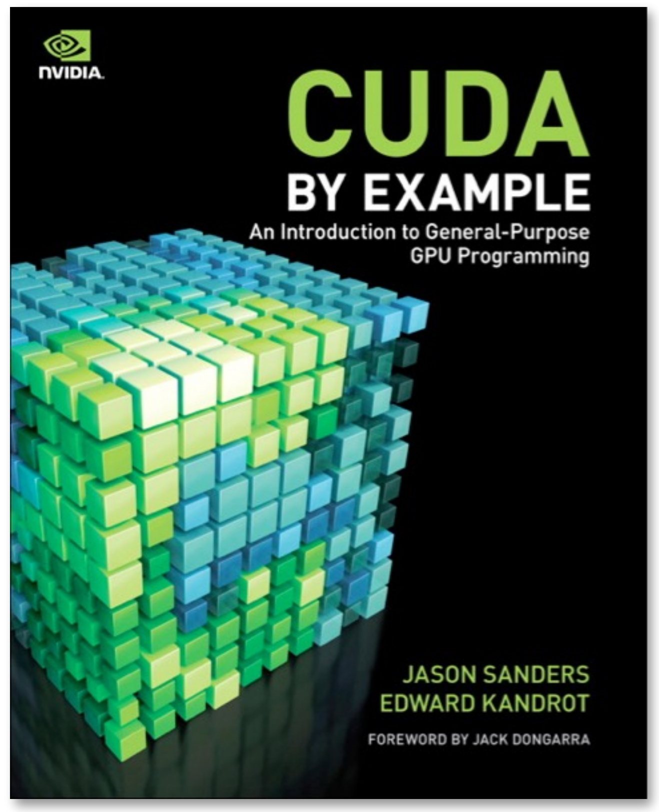
To design the right programming systems to make developing new applications simpler and also highly performant.

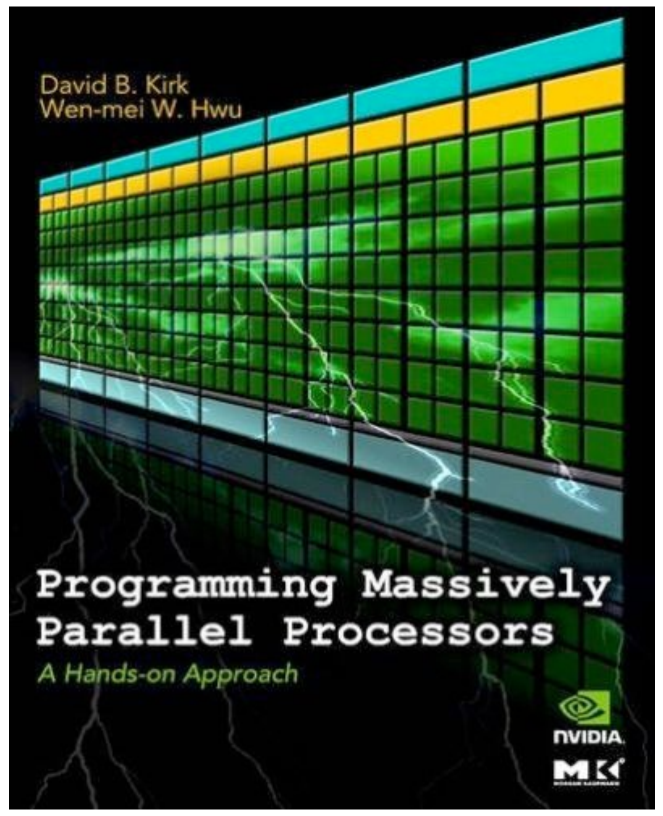
What this course is NOT about

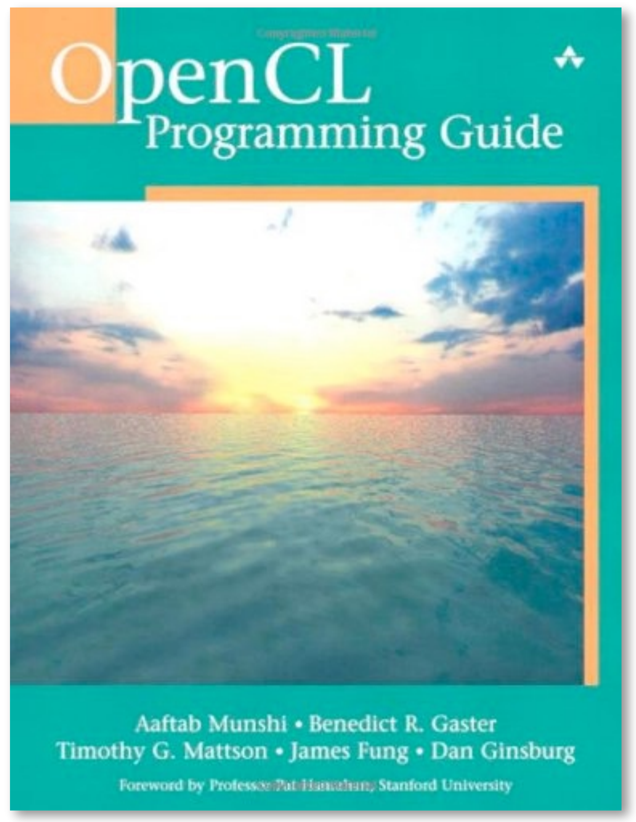
- This is not an [OpenGL, CUDA, OpenCL] programming course
  - But we will be analyzing and critiquing the design of these systems in detail
  - I expect students to pick up familiarity with relevant systems as we go

Many excellent references...

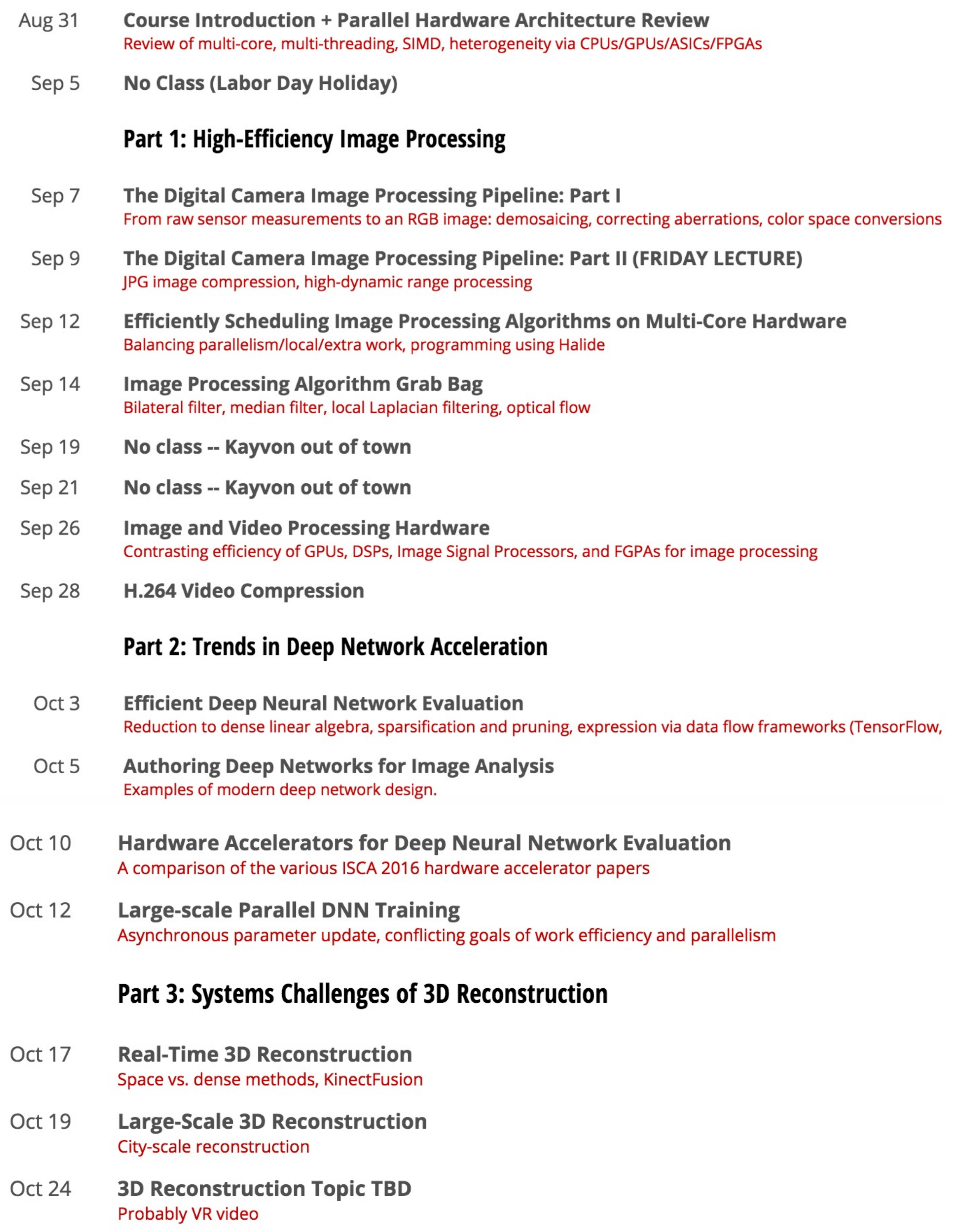

OpenGL Programming Guide
Sixth Edition
The Official Guide to Learning OpenGL, Version 2.1
OpenGL Architecture Review Board
Dave Shreiner Mason Woo Jackie Neider Tom Davis


CUDA
BY EXAMPLE
An Introduction to General-Purpose GPU Programming
JASON SANDERS
EDWARD KANDROT
FOREWORD BY JACK DONGARRA




OpenCL Programming Guide
Aaftab Munshi • Benedict R. Gaster
Timothy G. Mattson • James Fung • Dan Ginsburg

Major course themes/topics



Aug 31	**Course Introduction + Parallel Hardware Architecture Review** Review of multi-core, multi-threading, SIMD, heterogeneity via CPUs/GPUs/ASICs/FPGAs
Sep 5	**No Class (Labor Day Holiday)**
	Part 1: High-Efficiency Image Processing
Sep 7	**The Digital Camera Image Processing Pipeline: Part I** From raw sensor measurements to an RGB image: demosaicing, correcting aberrations, color space conversions
Sep 9	**The Digital Camera Image Processing Pipeline: Part II (FRIDAY LECTURE)** JPG image compression, high-dynamic range processing
Sep 12	**Efficiently Scheduling Image Processing Algorithms on Multi-Core Hardware** Balancing parallelism/local/extra work, programming using Halide
Sep 14	**Image Processing Algorithm Grab Bag** Bilateral filter, median filter, local Laplacian filtering, optical flow
Sep 19	**No class -- Kayvon out of town**
Sep 21	**No class -- Kayvon out of town**
Sep 26	**Image and Video Processing Hardware** Contrasting efficiency of GPUs, DSPs, Image Signal Processors, and FGPAs for image processing
Sep 28	**H.264 Video Compression**
	Part 2: Trends in Deep Network Acceleration
Oct 3	**Efficient Deep Neural Network Evaluation** Reduction to dense linear algebra, sparsification and pruning, expression via data flow frameworks (TensorFlow,
Oct 5	**Authoring Deep Networks for Image Analysis** Examples of modern deep network design.
Oct 10	**Hardware Accelerators for Deep Neural Network Evaluation** A comparison of the various ISCA 2016 hardware accelerator papers
Oct 12	**Large-scale Parallel DNN Training** Asynchronous parameter update, conflicting goals of work efficiency and parallelism
	Part 3: Systems Challenges of 3D Reconstruction
Oct 17	**Real-Time 3D Reconstruction** Space vs. dense methods, KinectFusion
Oct 19	**Large-Scale 3D Reconstruction** City-scale reconstruction
Oct 24	**3D Reconstruction Topic TBD** Probably VR video

High-performance image processing

Algorithms for processing images/video in a modern digital camera

Image processing hardware components

Image/video compression

Expressing and accelerating deep learning for computer vision

Large-scale 3D reconstruction

Major course themes/topics

Part 4: The Design and Implementation of 3D Graphics Systems

Oct 26	**Architecture of the GPU-Accelerated Real-Time 3D Graphics Pipeline** Graphics pipeline abstractions, scheduling challenges
Oct 31	**Rasterization and Occlusion** Hardware acceleration, depth and color compression algorithms
Nov 2	**Texture Mapping** Texture sampling and prefiltering, texture compression, data layout optimizations
Nov 7	**Parallel Scheduling of the Graphics Pipeline** Molnar taxonomy, scheduling under data amplification, tiled rendering
Nov 9	**Deferred Shading and Image-Space Rendering Techniques** Deferred shading as a scheduling decision, image-space anti-aliasing
Nov 14	**Hardware-Accelerated Ray Tracing** Ray-tracing as an alternative to rasterization, what does modern ray tracing HW do?
Nov 16	**Shading Language Design** Contrasting different shading languages, is CUDA a DSL?
Nov 21	**Case Study: The Spire Shading Language** Discussion of relationship to other recent DSLs

The GPU-accelerated 3D graphics pipelines
(high-performance rendering for real time applications)

Logistics

- **Course web site:**
  - **http://graphics.cs.cmu.edu/courses/15769/fall2016/**
- **All announcements will go out via Piazza**
  - **http://www.piazza.com/cmu/fall2016/15769**
- **Kayvon's office hours: drop in or by appointment (EDSH 225)**

Expectations of you

- **30% participation**
  - There will be ~1-2 assigned paper readings 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.

- **15% mini-assignments (2-3 short programming assignments)**
  - Assignment 1: implement and optimize a basic RAW image processing pipeline

- **20% 2 take-home "exams"**
  - Exam 1: covers course parts 1 and 2
  - Exam 2: covers course parts 3 and 4

- **35% self-selected final project**
  - I suggest you start talking to me now (can be teams of up to two)

Review: throughput computing hardware

Review concepts

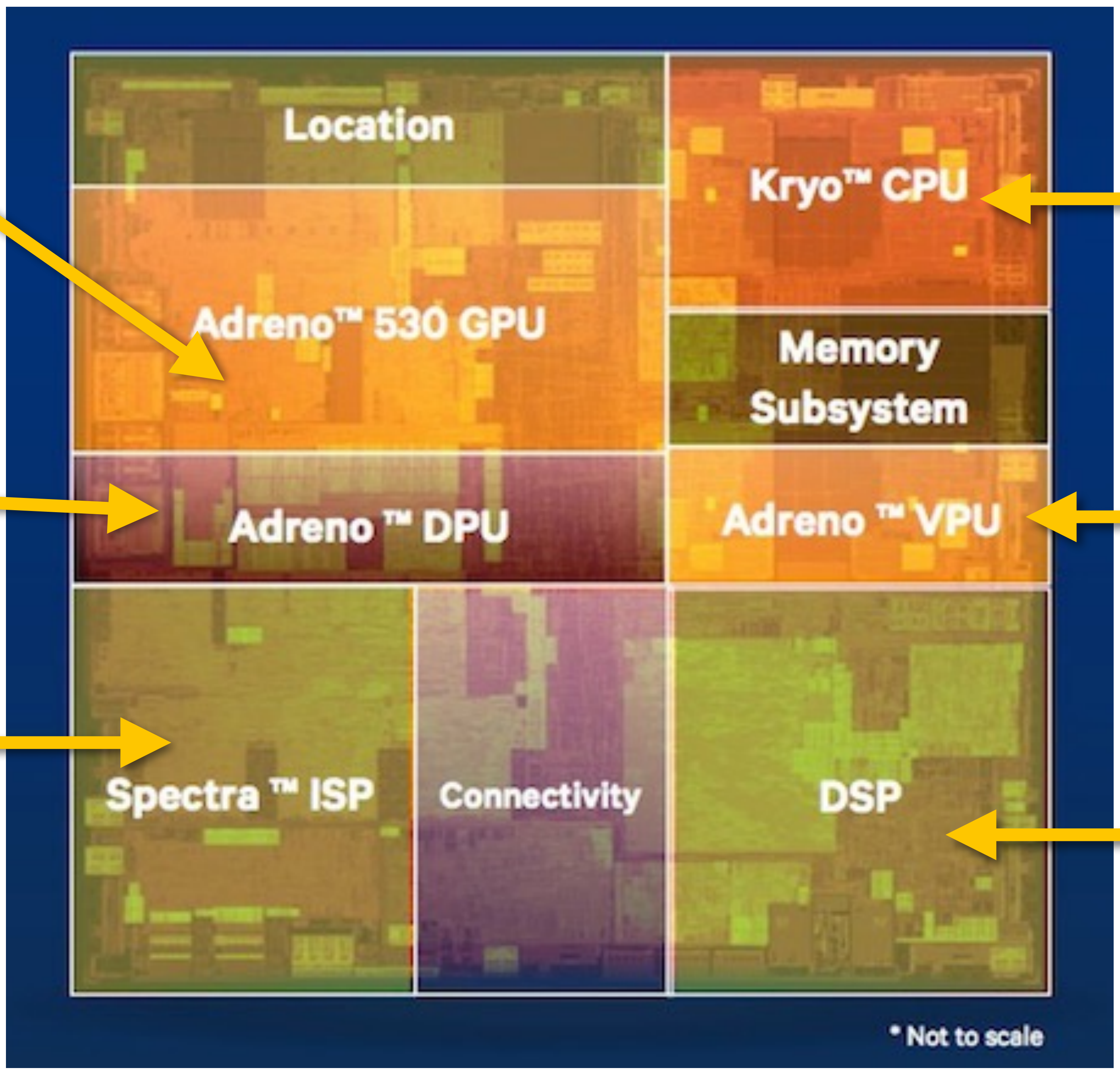
- What are these design concepts, and what problem/goals do they address?
  - Muti-core processing
  - SIMD processing
  - Hardware multi-threading

- 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 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)

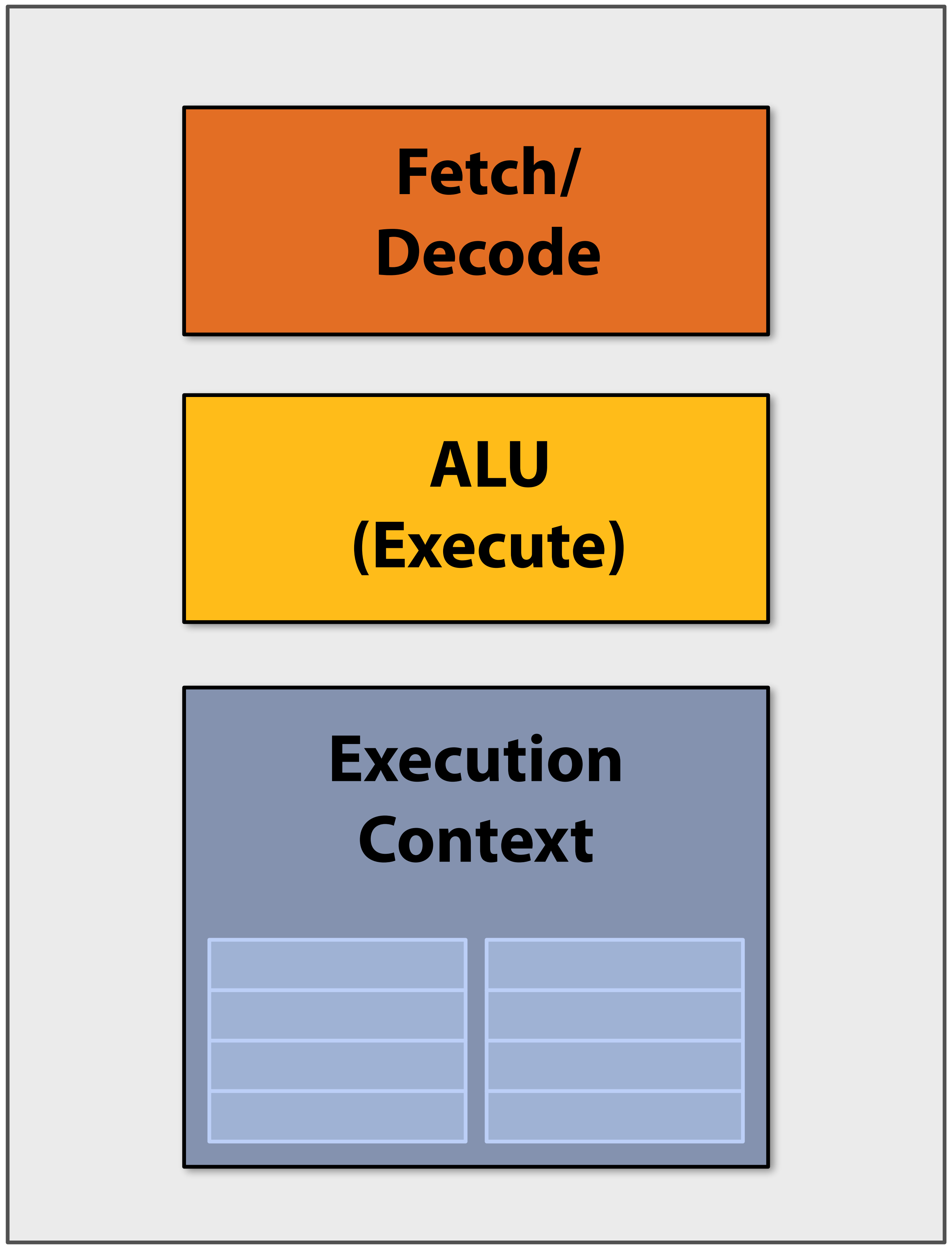
Multi-core ARM CPU

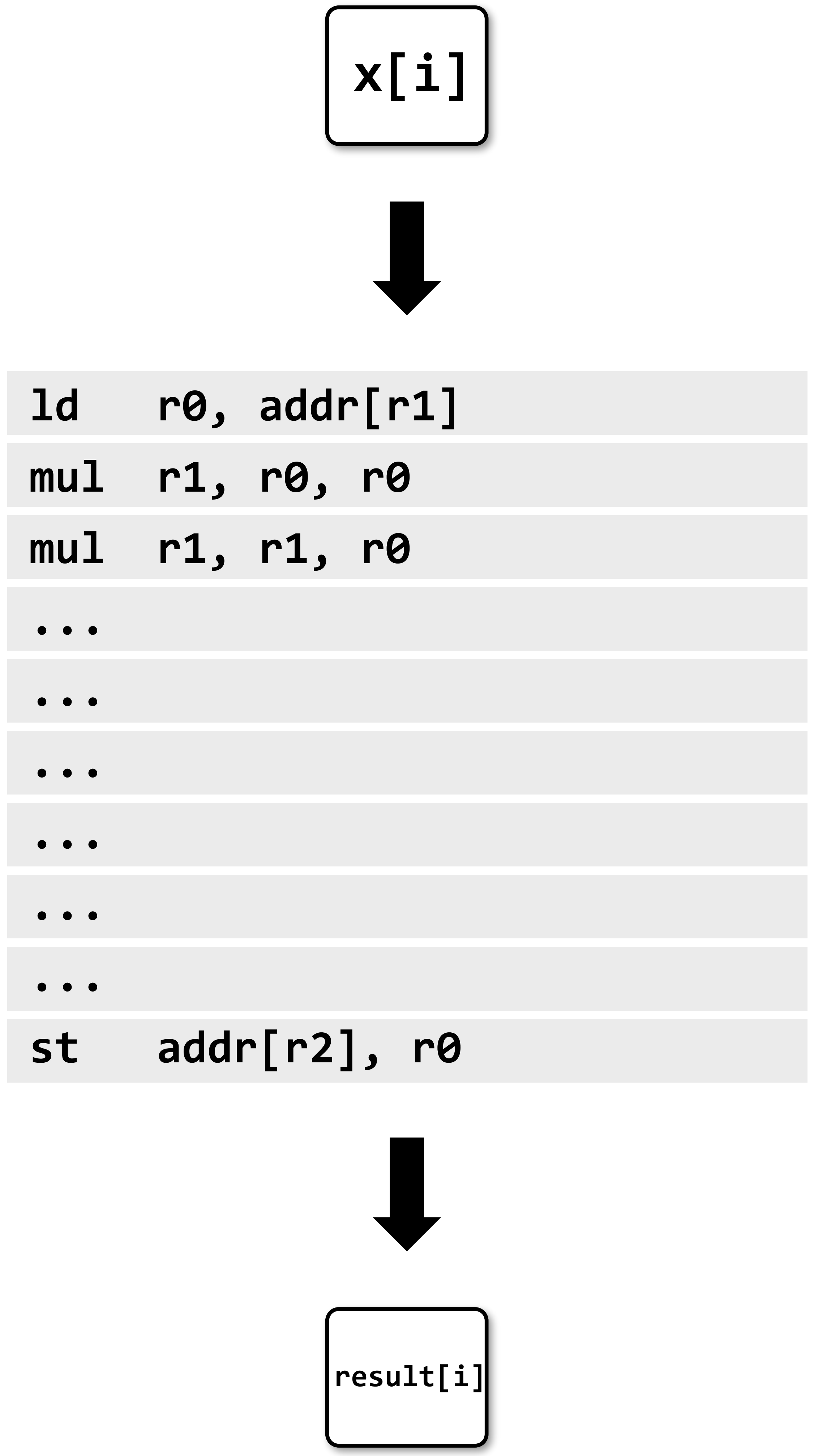
Video encode/decode ASIC (H.265 @ 4K)

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

Multi-core processing

Executing an instruction stream

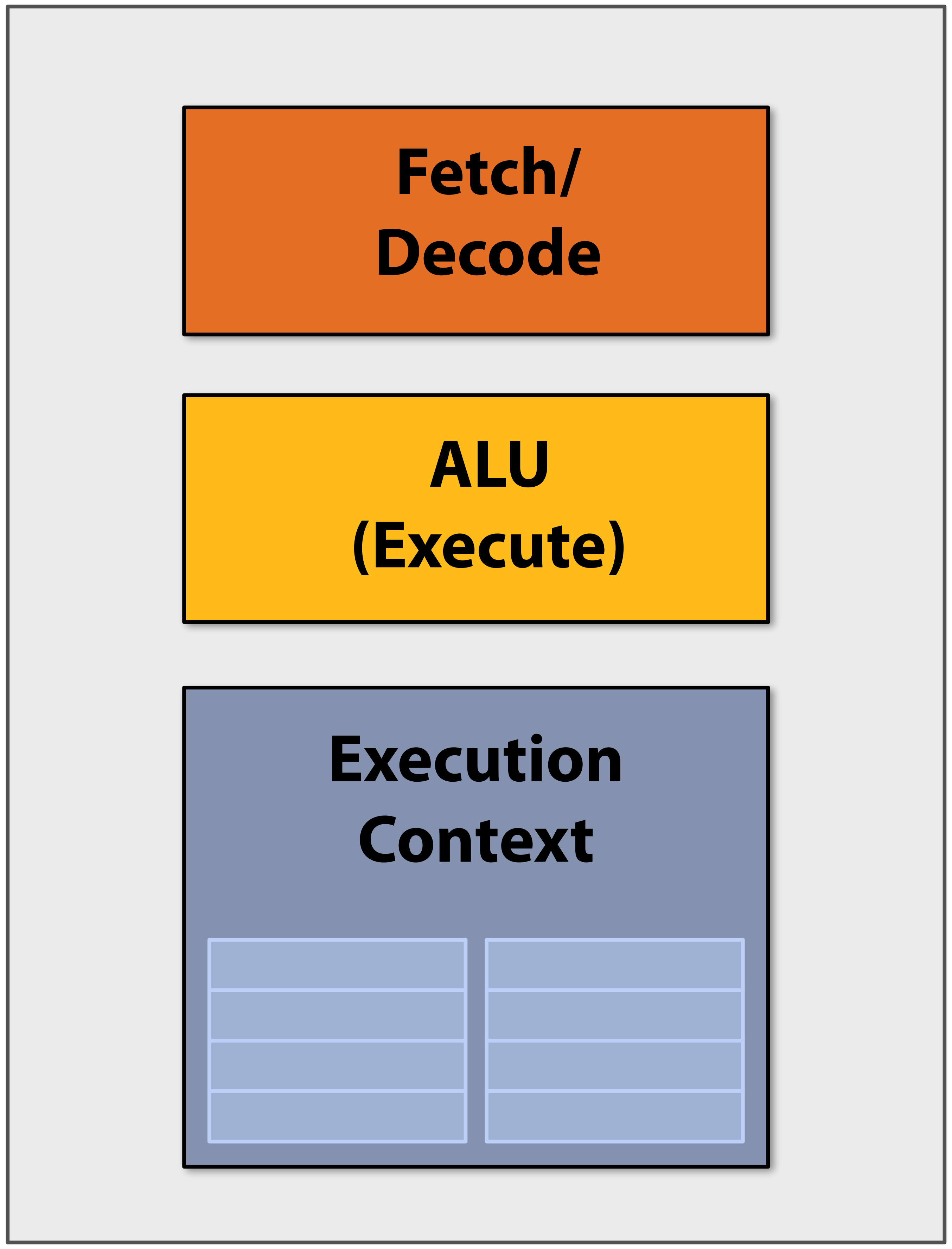


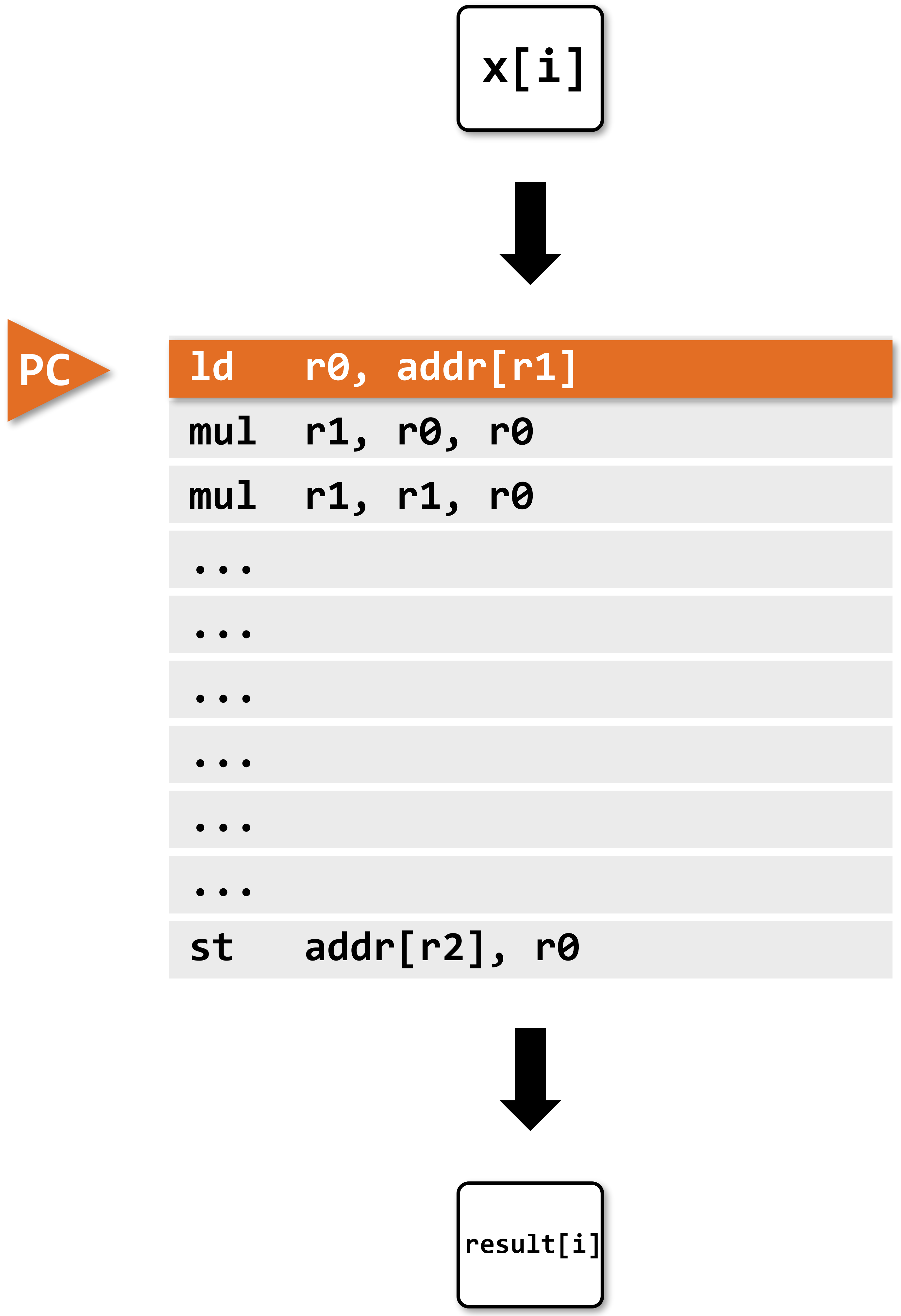


```
ld   r0, addr[r1]
mul  r1, r0, r0
mul  r1, r1, r0
...
...
...
...
...
...
st   addr[r2], r0
```

Executing an instruction stream

My very simple processor: executes one instruction per clock

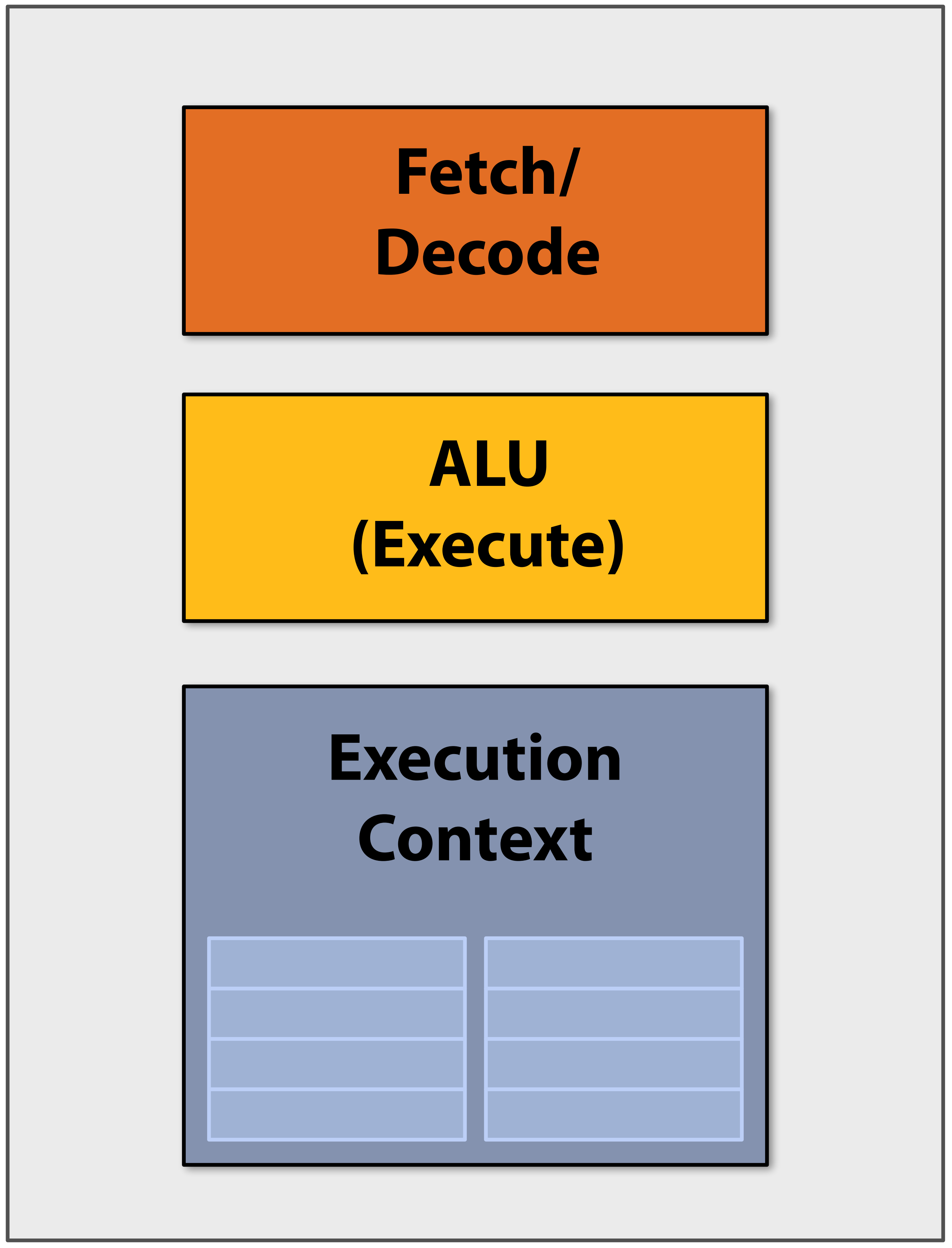


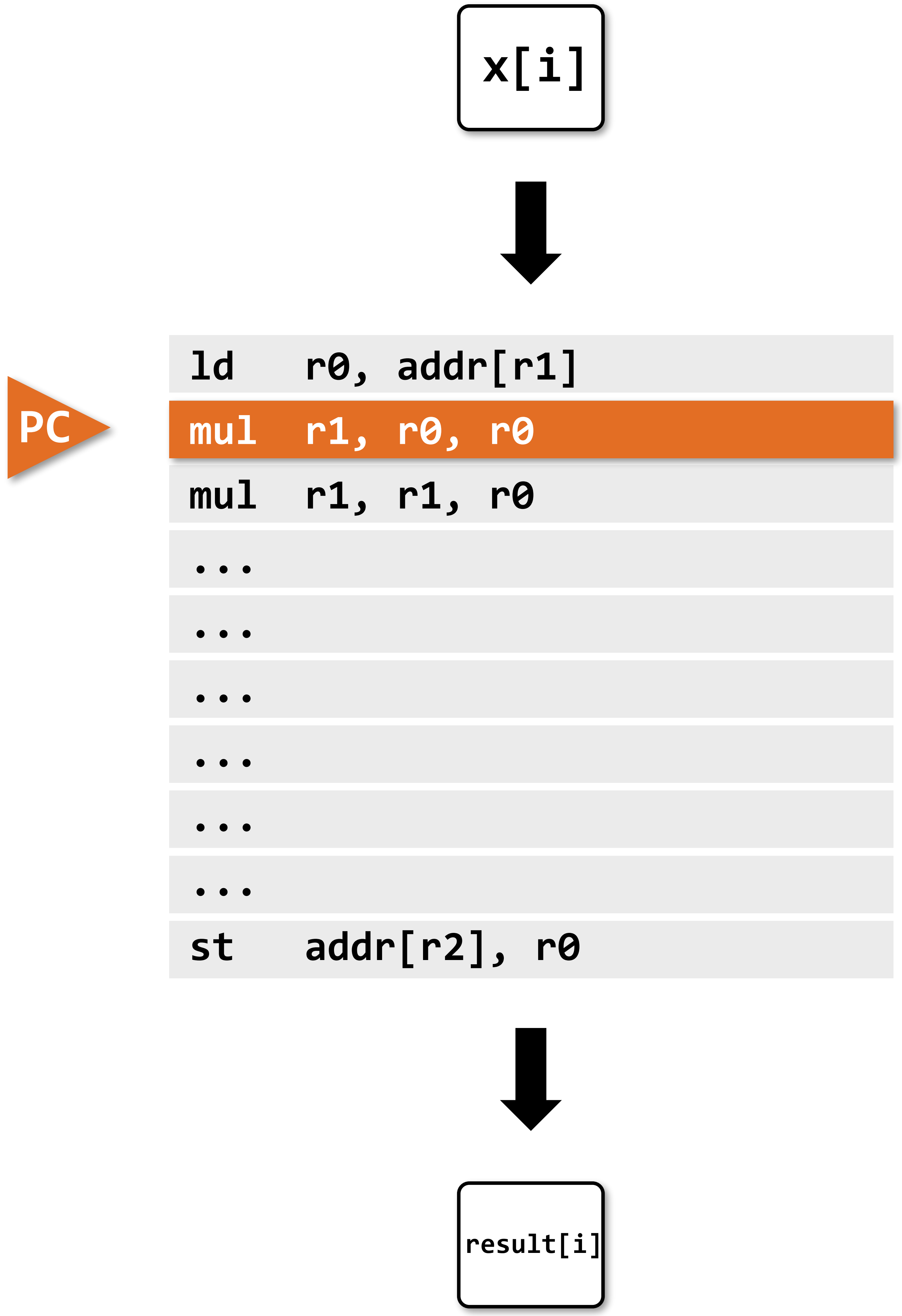


```
ld   r0, addr[r1]
mul  r1, r0, r0
mul  r1, r1, r0
...
...
...
...
...
...
st   addr[r2], r0
```

Executing an instruction stream

My very simple processor: executes one instruction per clock

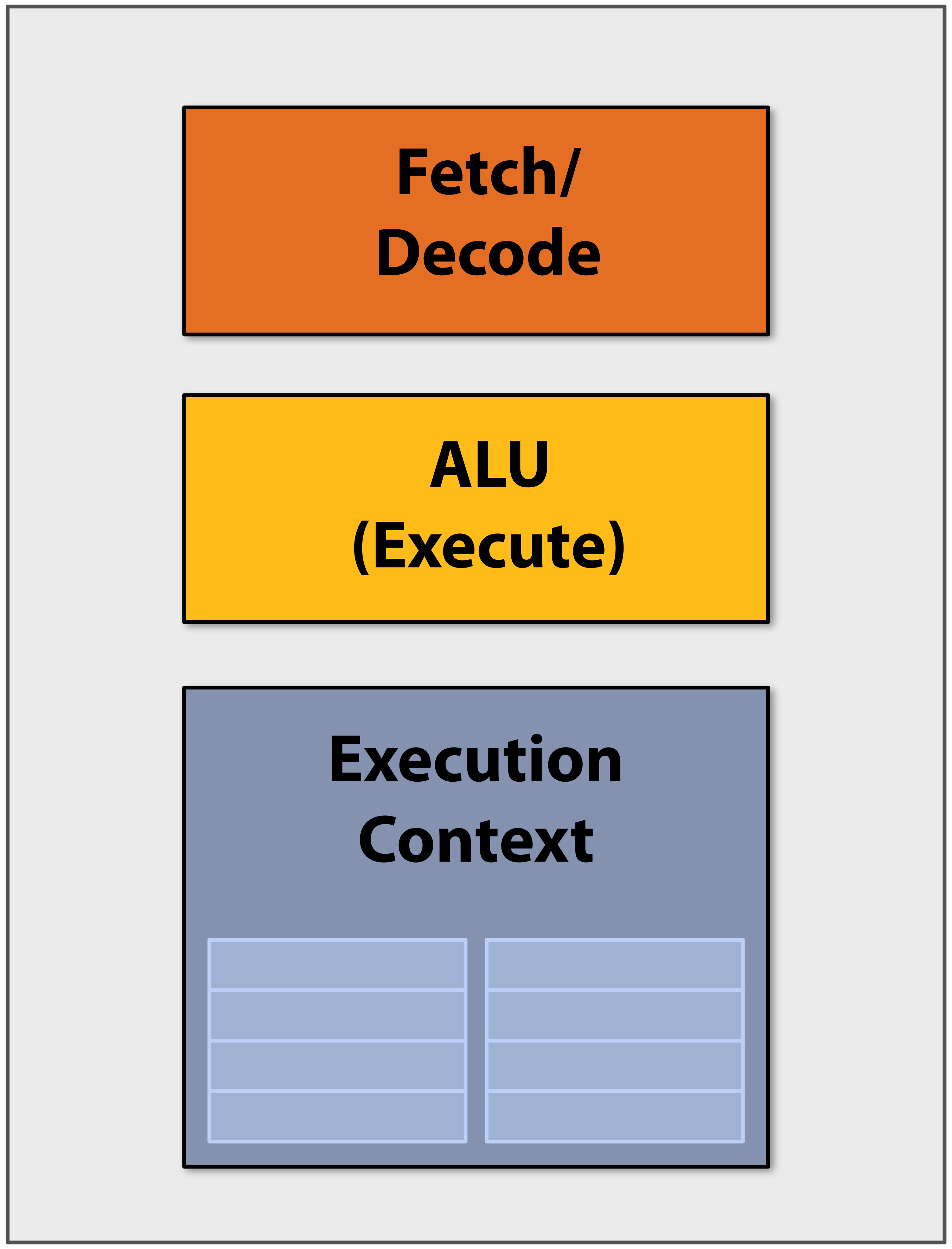


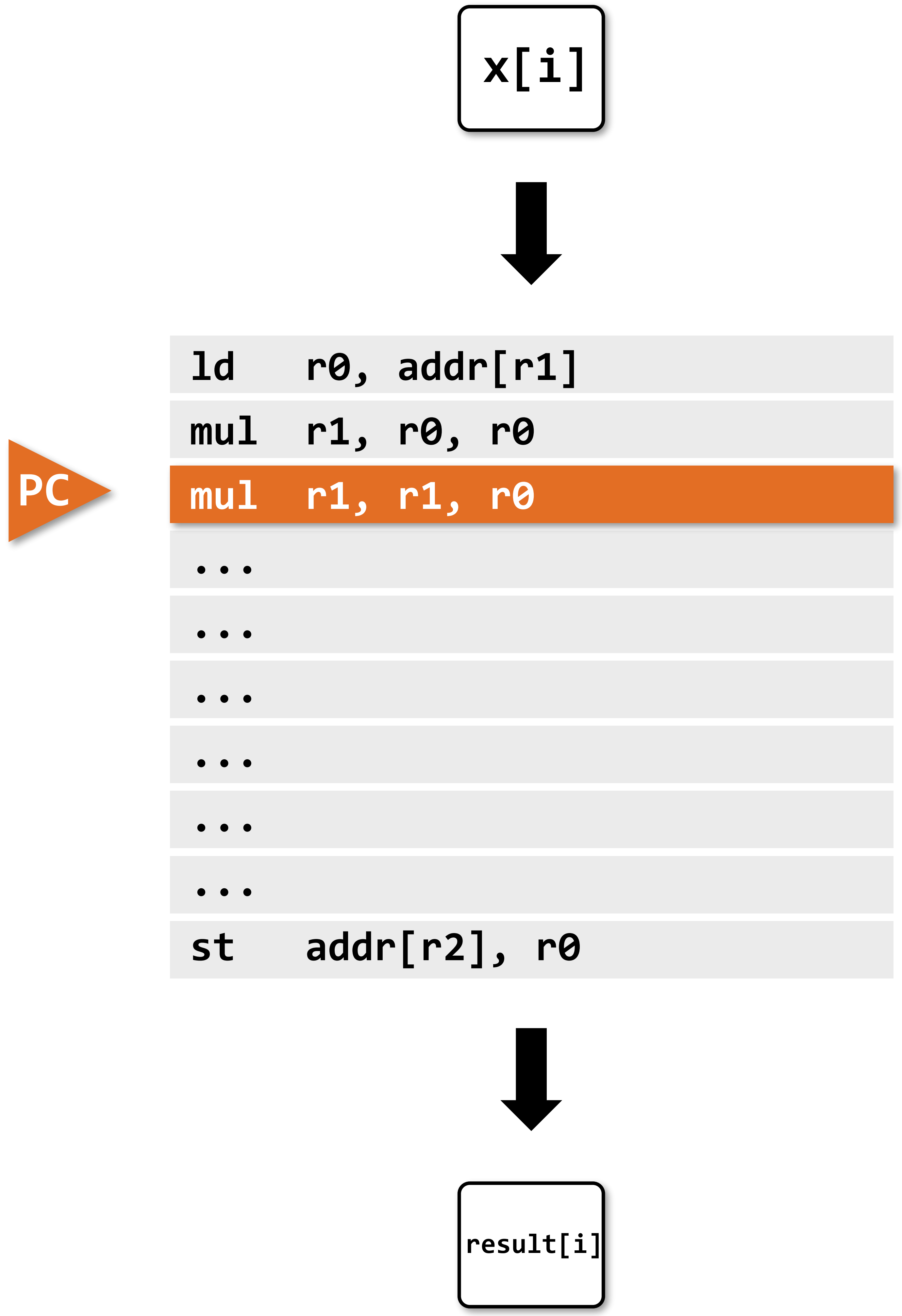


```
ld   r0, addr[r1]
mul  r1, r0, r0
mul  r1, r1, r0
...
...
...
...
...
...
st   addr[r2], r0
```

Executing an instruction stream

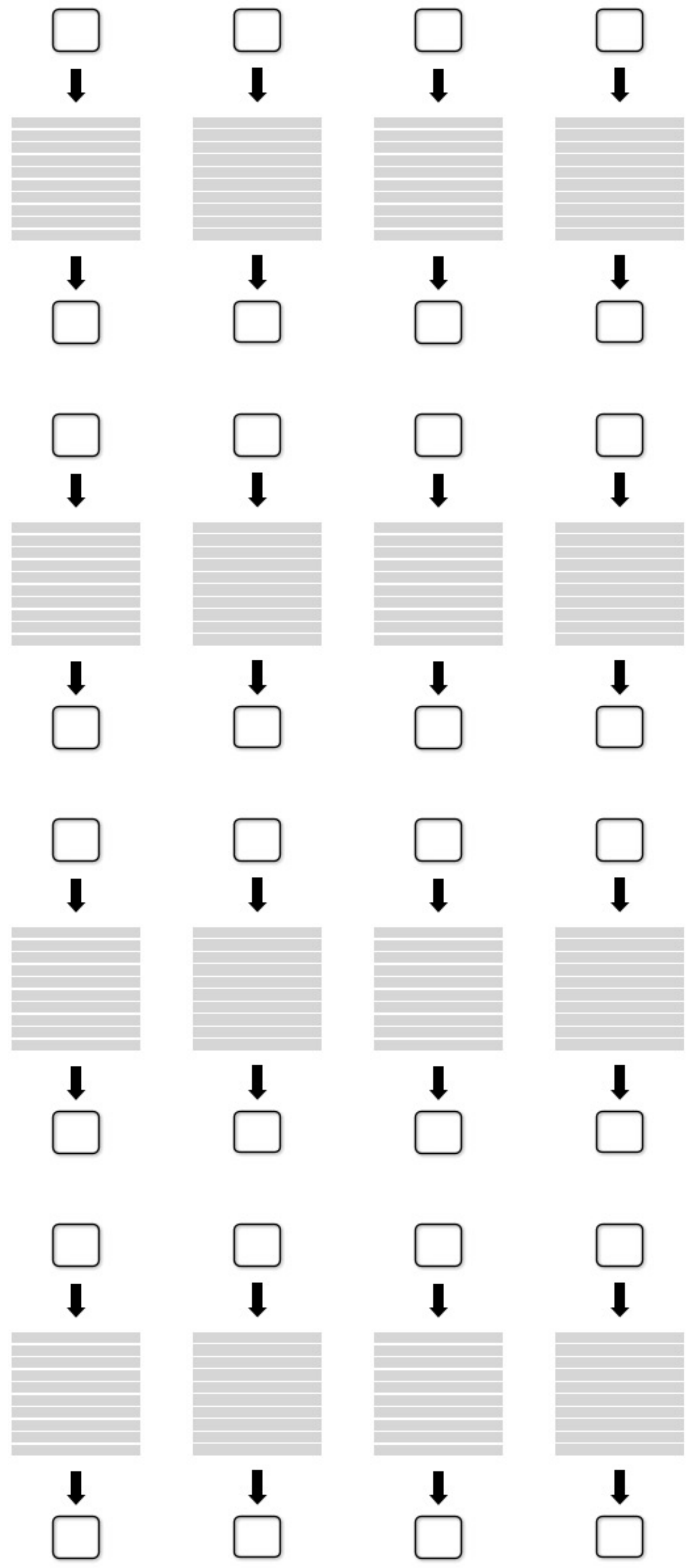
My very simple processor: executes one instruction per clock

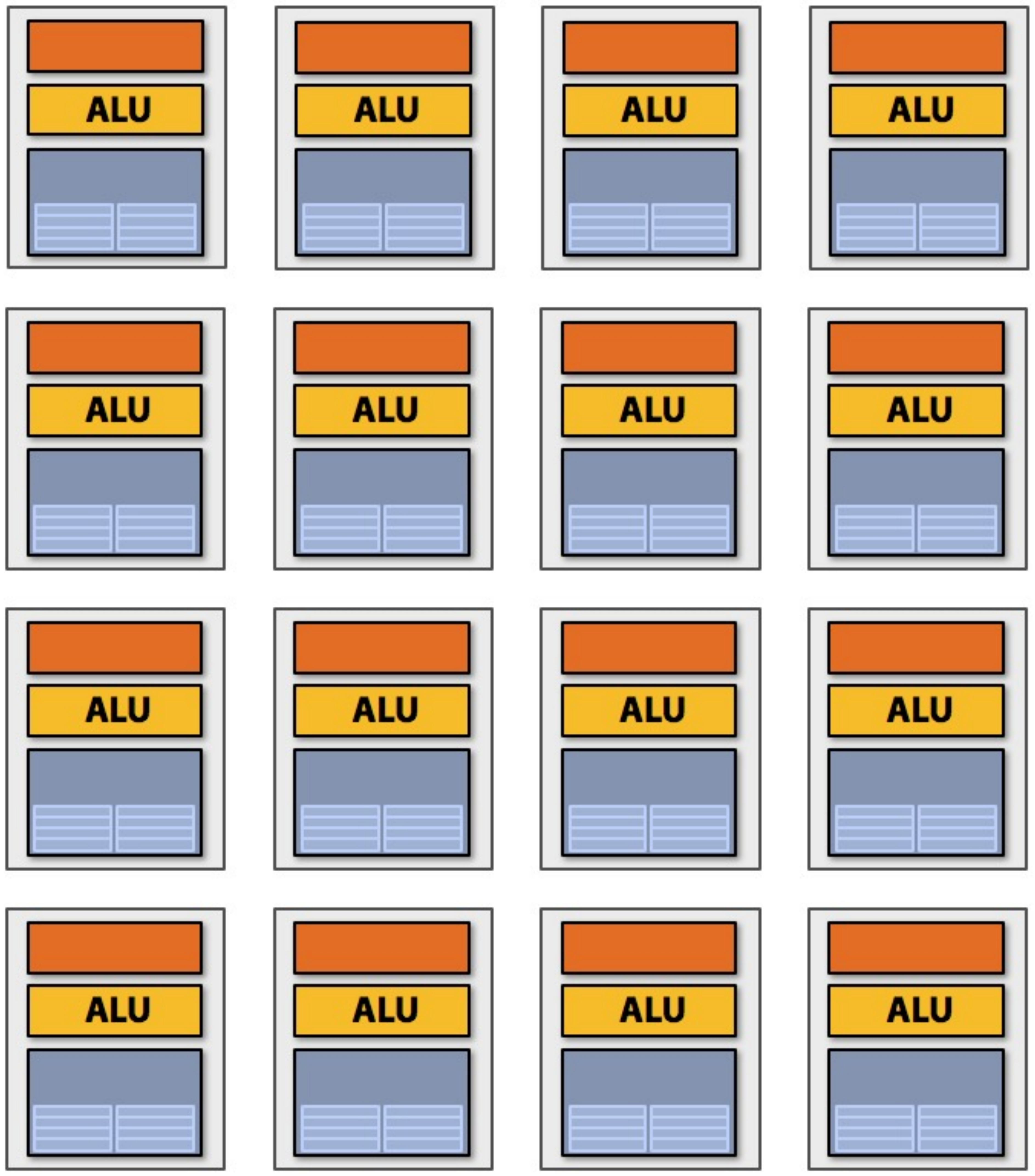




```
ld   r0, addr[r1]
mul  r1, r0, r0
mul  r1, r1, r0
...
...
...
...
...
...
st   addr[r2], r0
```

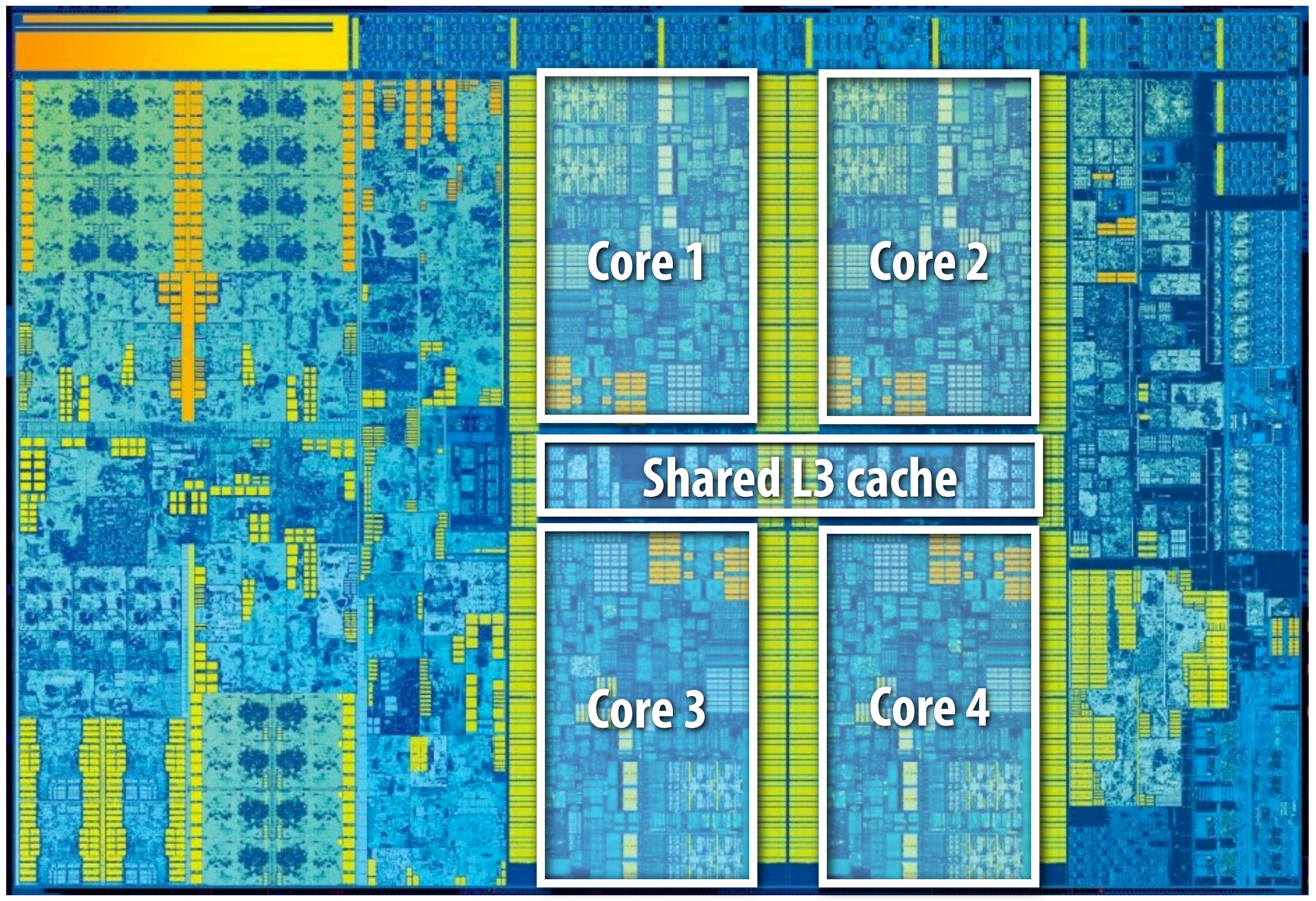
Multi-core: process multiple instruction streams in parallel



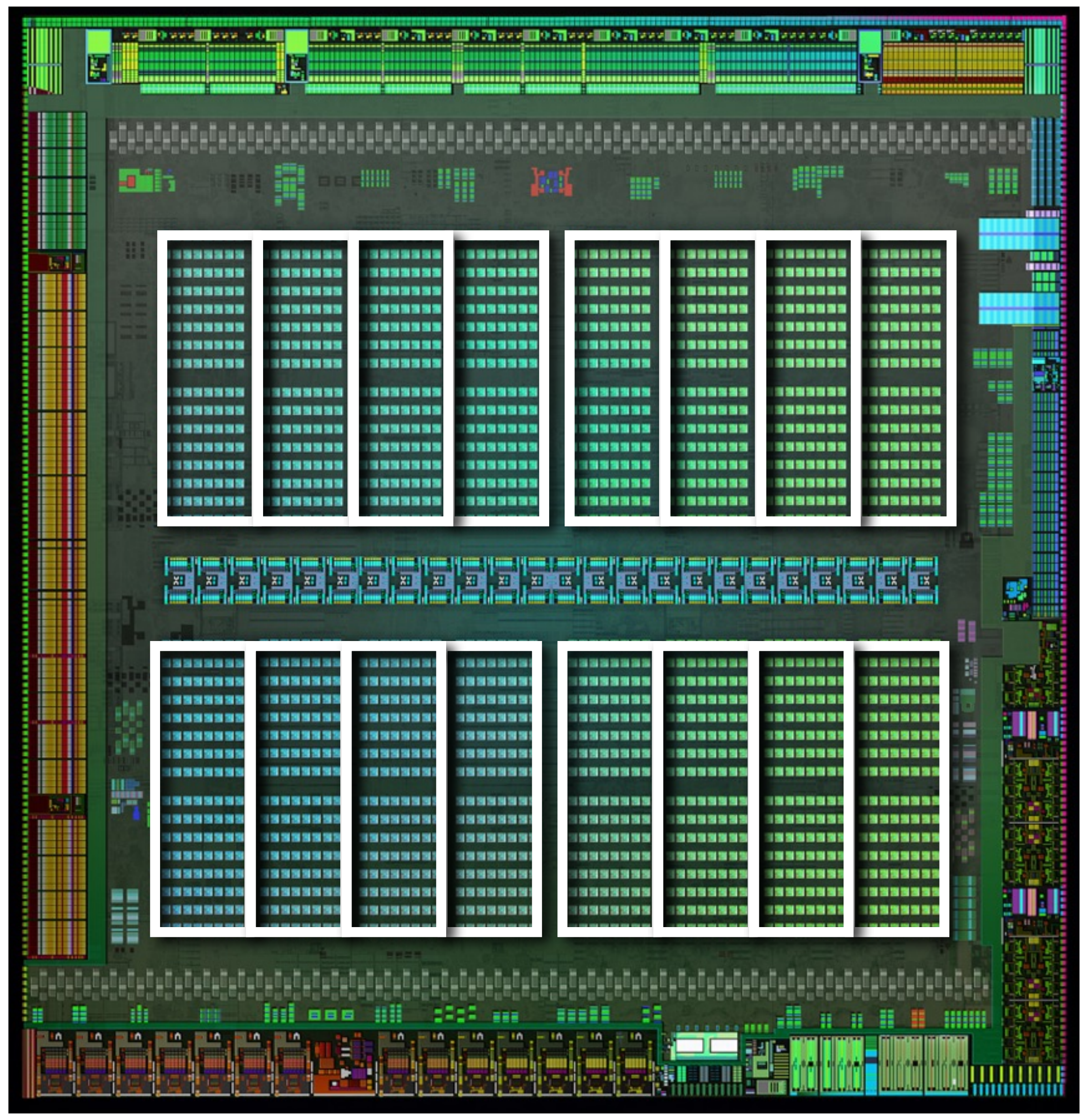


Sixteen cores, sixteen simultaneous instruction streams

Multi-core examples

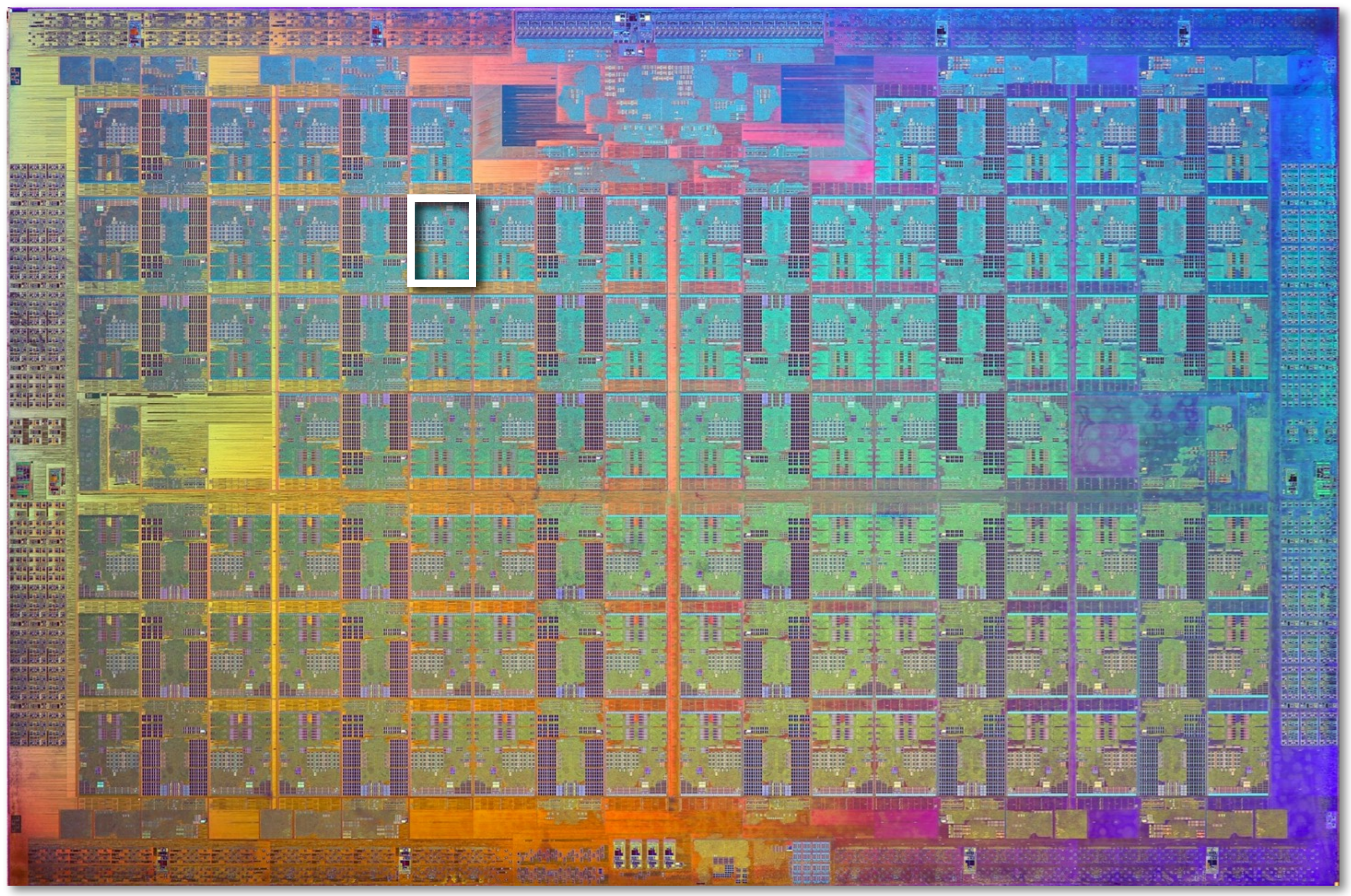


Intel "Skylake" Core i7 quad-core CPU
(2015)

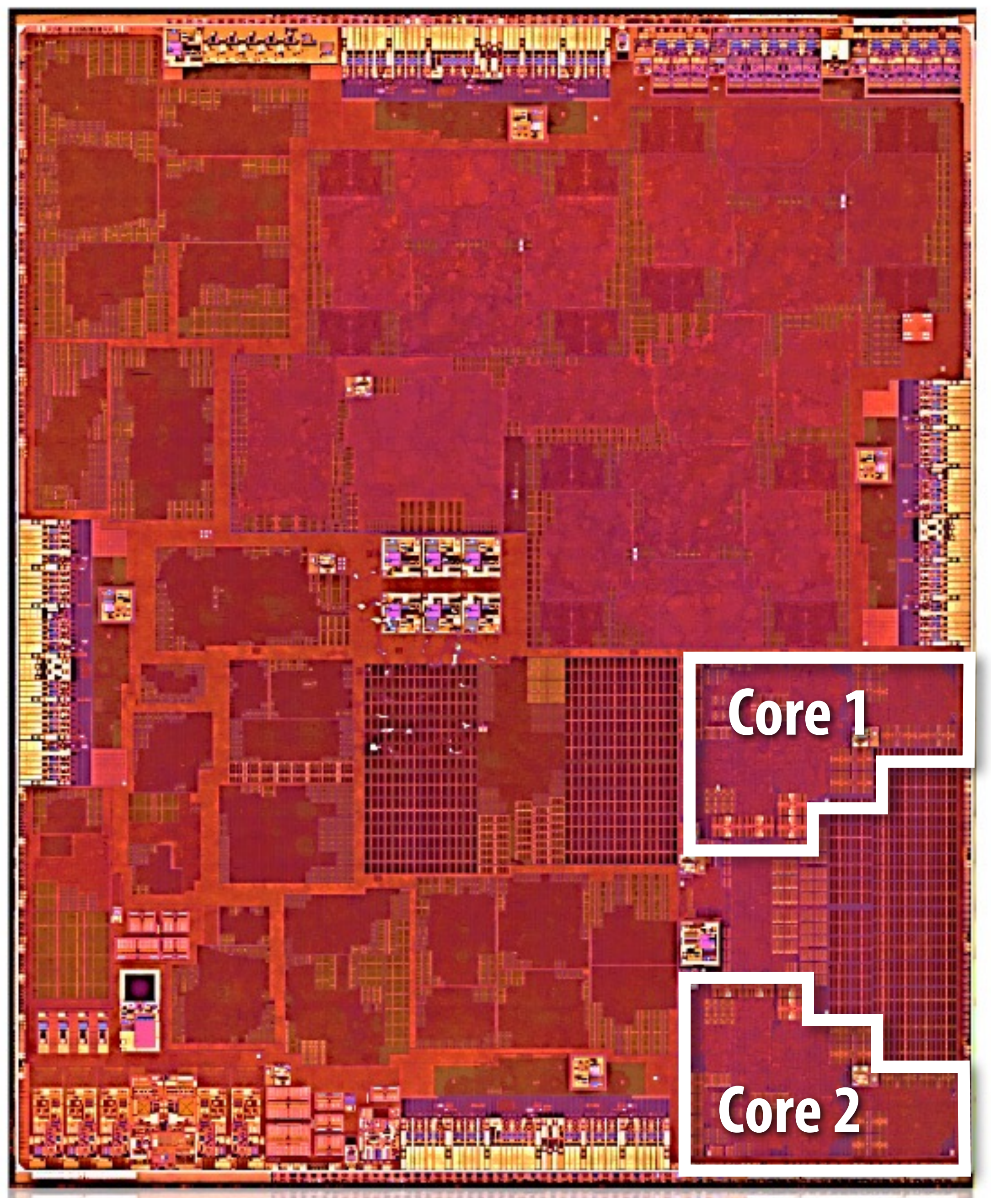


NVIDIA GTX 980 GPU
16 replicated processing cores ("SM")
(2014)

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

Superscalar execution

Superscalar execution

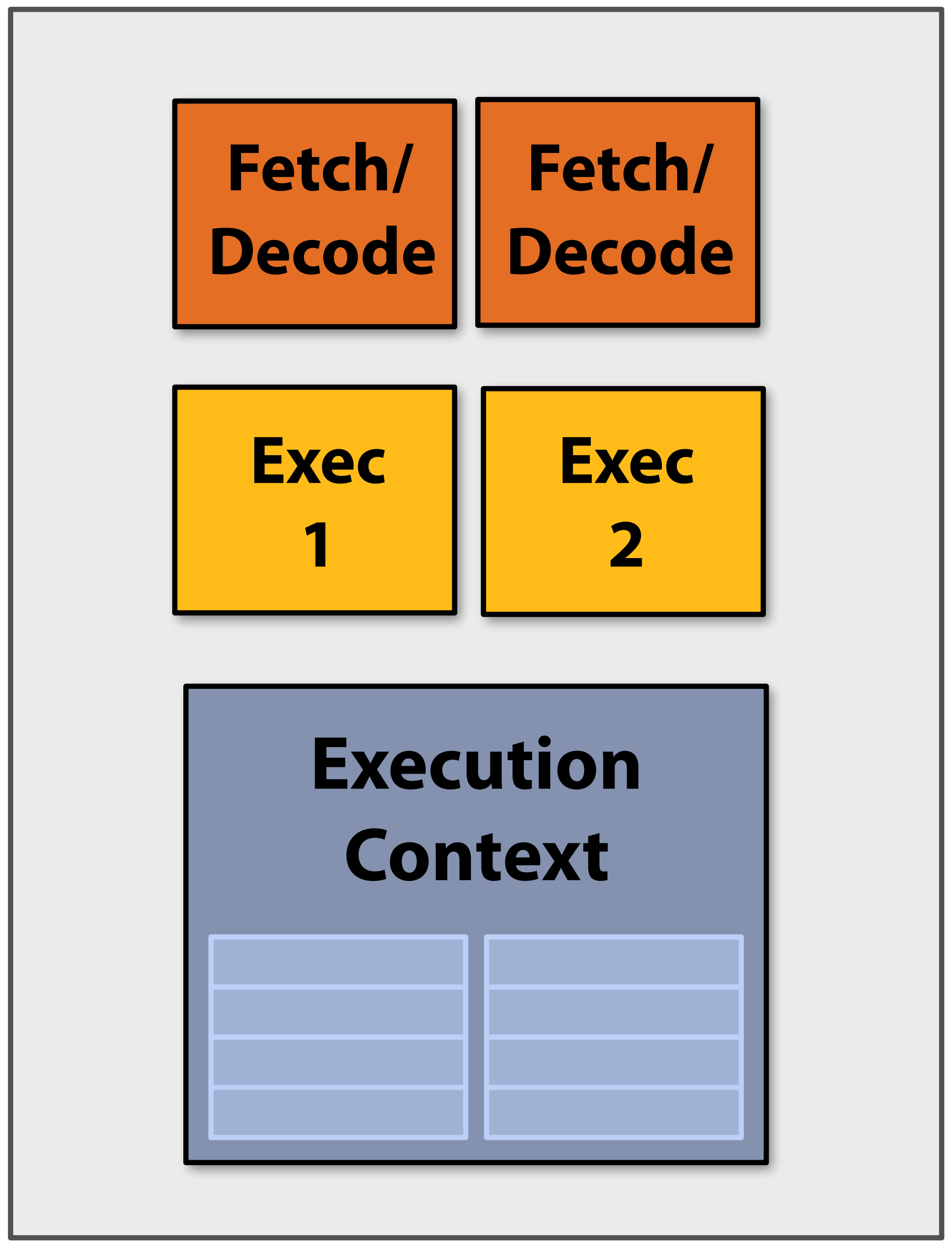
Program: computes sin of input x via Taylor expansion

```
result 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

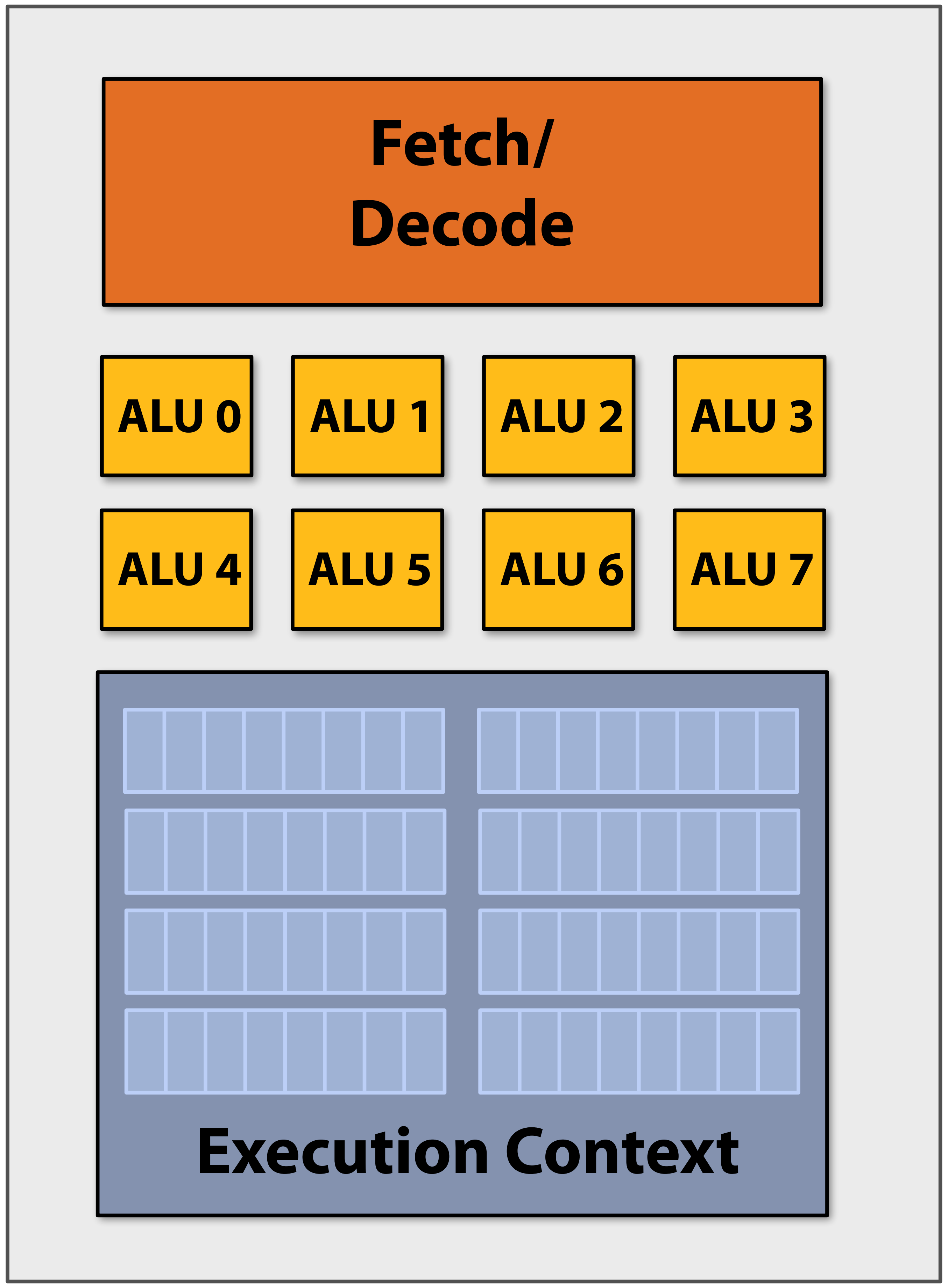
(They are detected by the processor at run-time and may be executed in parallel on execution units 1 and 2)

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



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

Vector program (using AVX intrinsics)

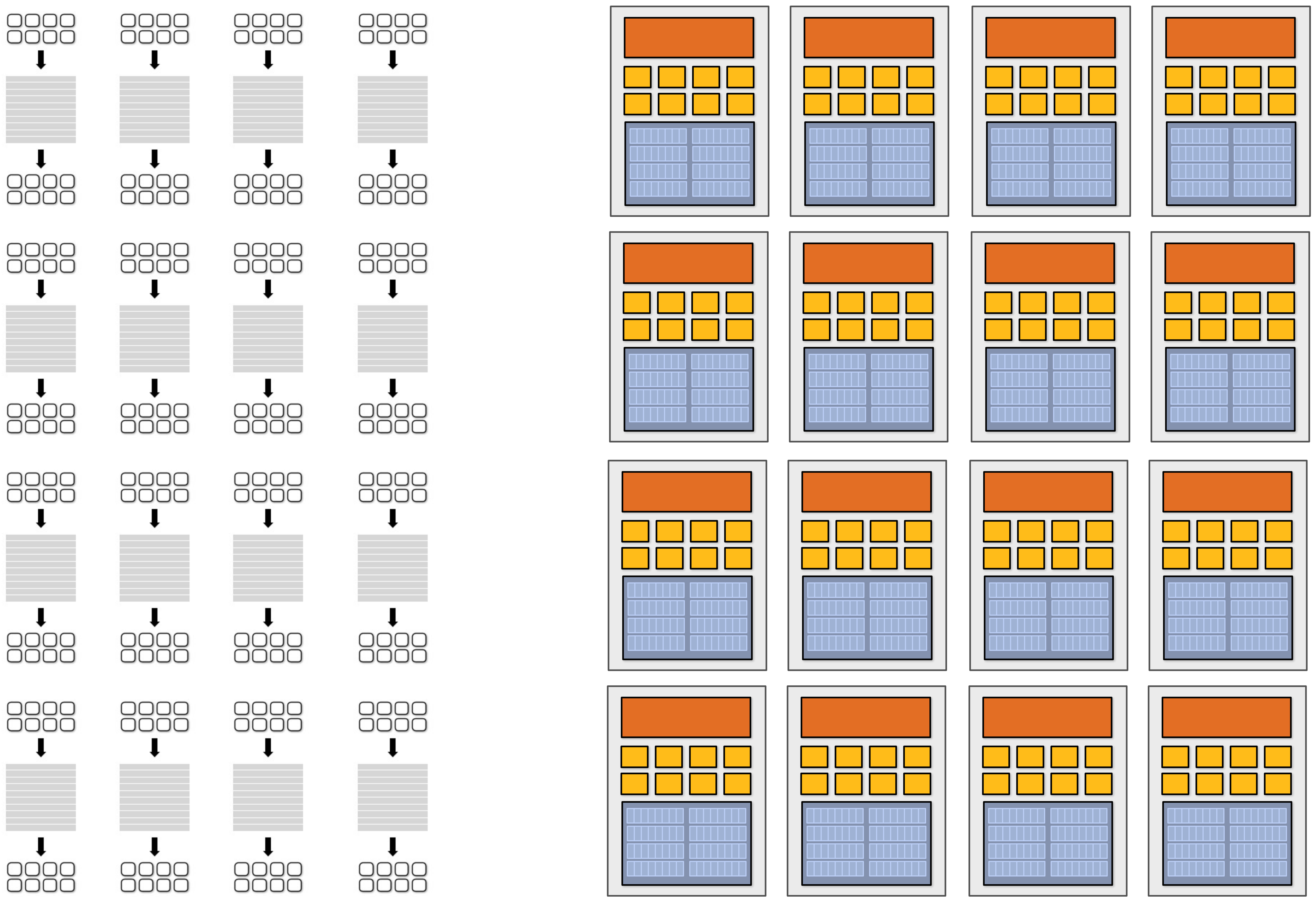
```
#include <immintrin.h>
void sinx(int N, int terms, float* x, float* sinx)
{
   float three_fact = 6;  // 3!
   for (int i=0; i<N; i+=8)
   {
       __m256 origx = _mm256_load_ps(&x[i]);
       __m256 value = origx;
       __m256 numer = _mm256_mul_ps(origx, _mm256_mul_ps(origx, origx));
       __m256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

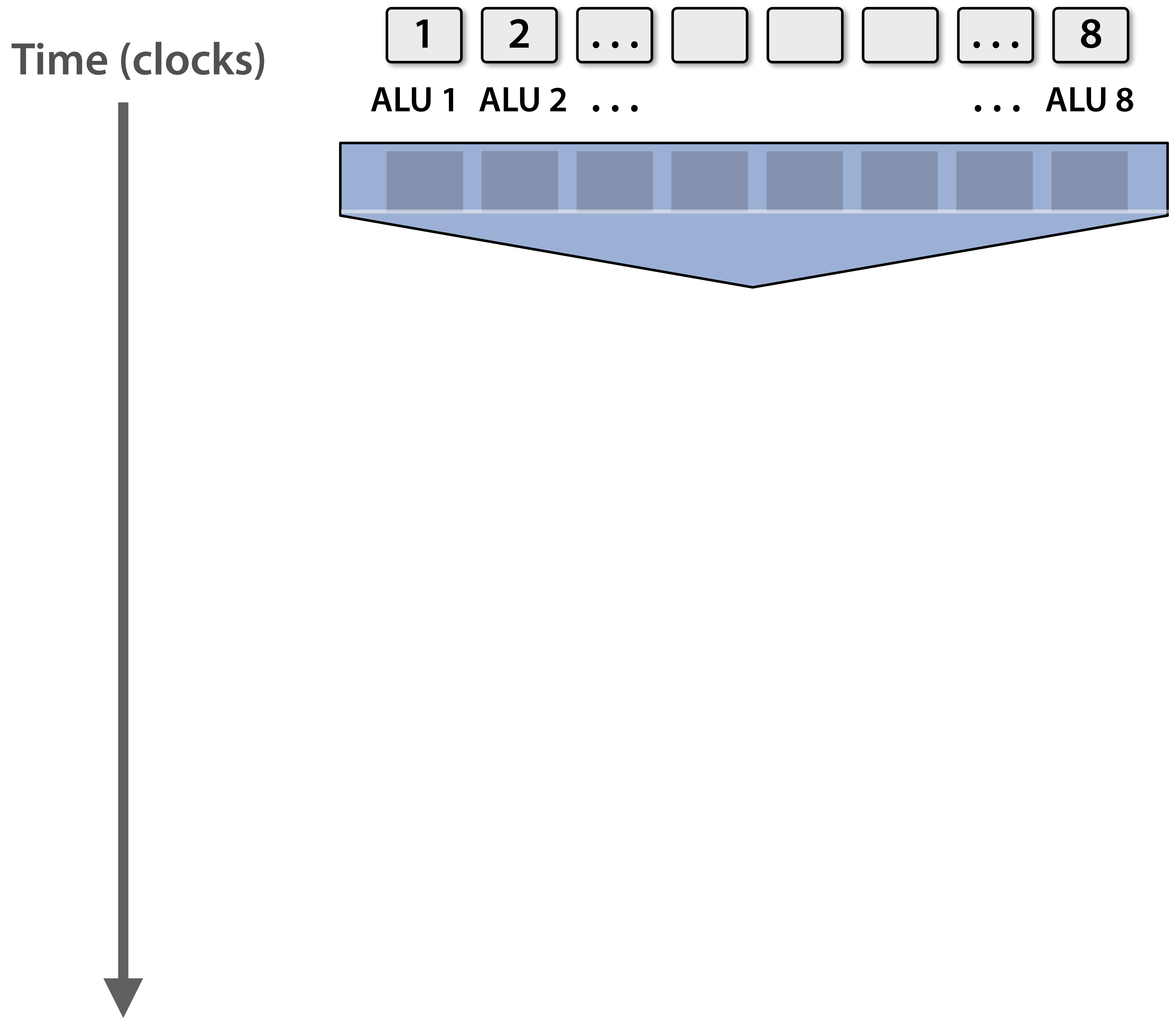
(in Kayvon's fictitious data-parallel language)

```
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.

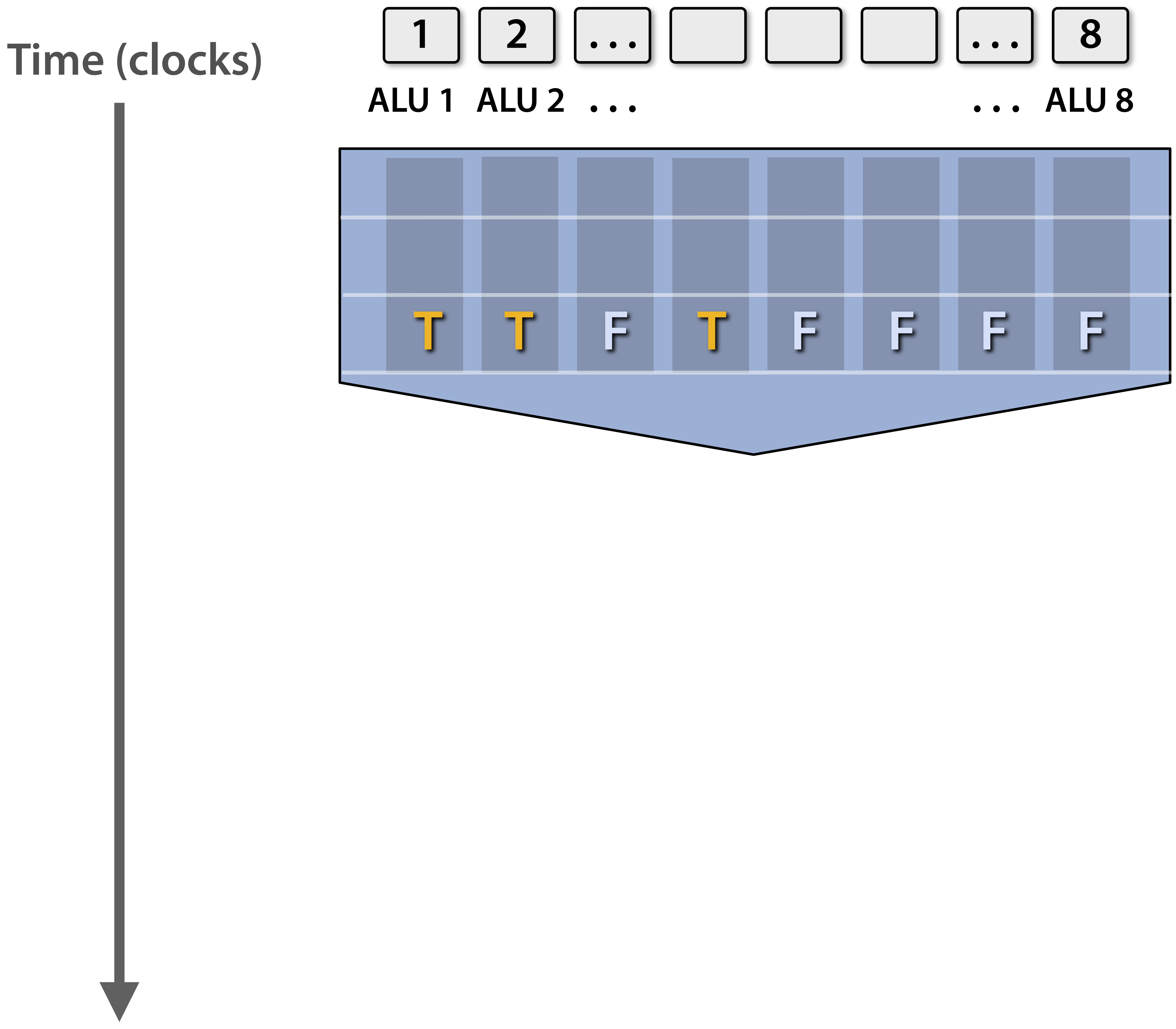
What about conditional execution?



(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;
```

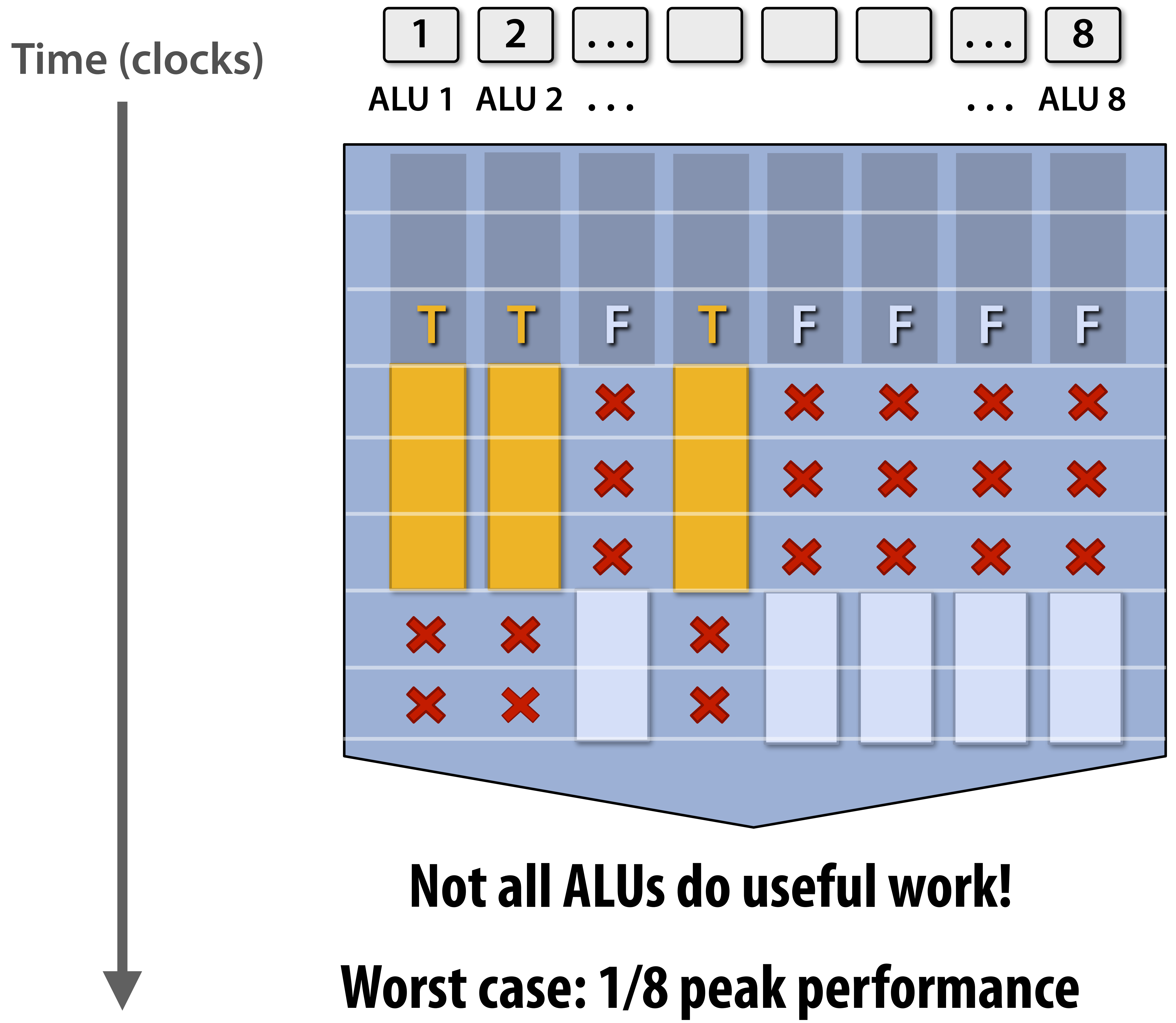
What about conditional execution?



(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



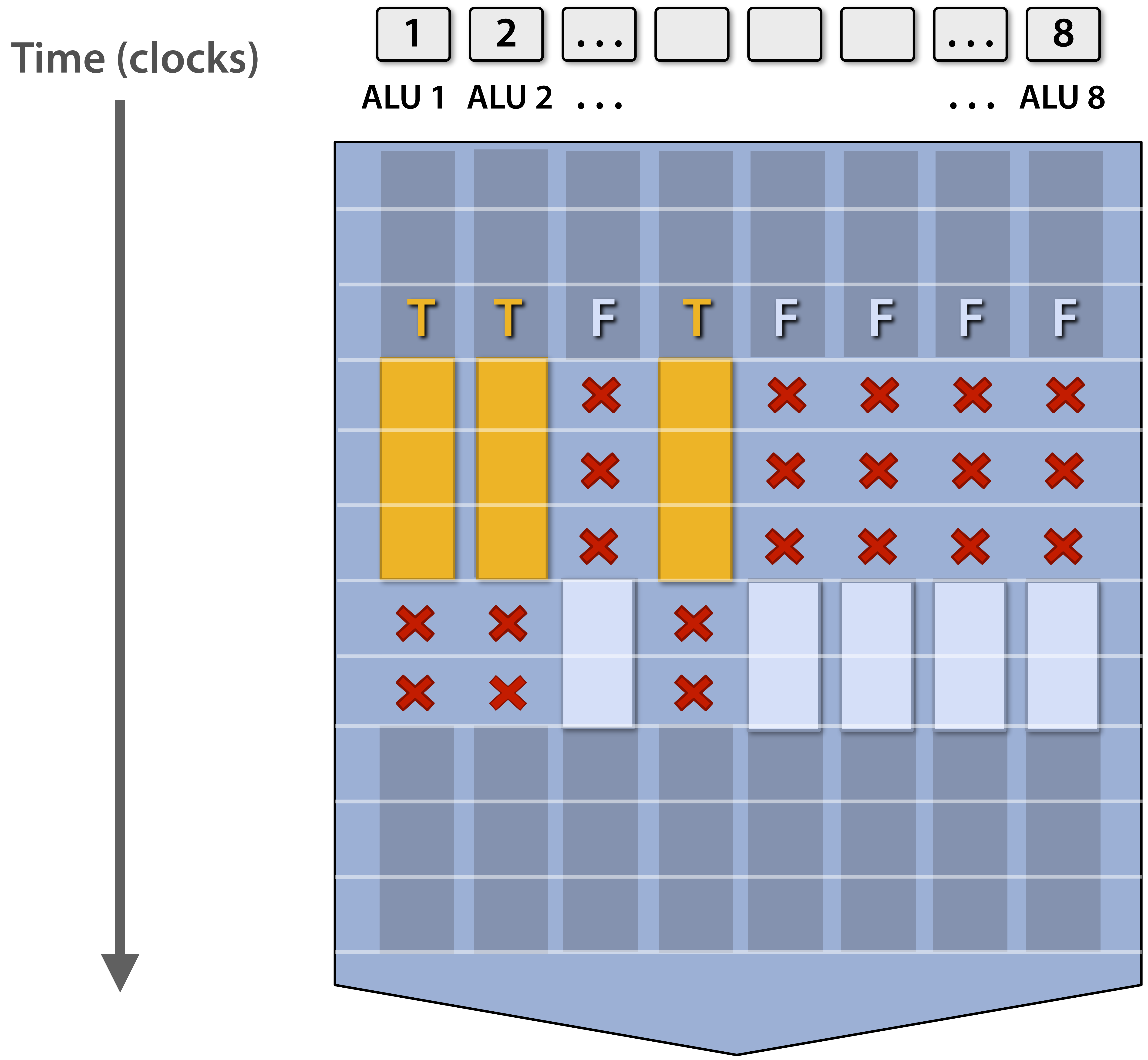
(assume logic below is to be executed for each element in input array 'A', producing output into the array 'result')

```
<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;
```

Not all ALUs do useful work!

Worst case: 1/8 peak performance

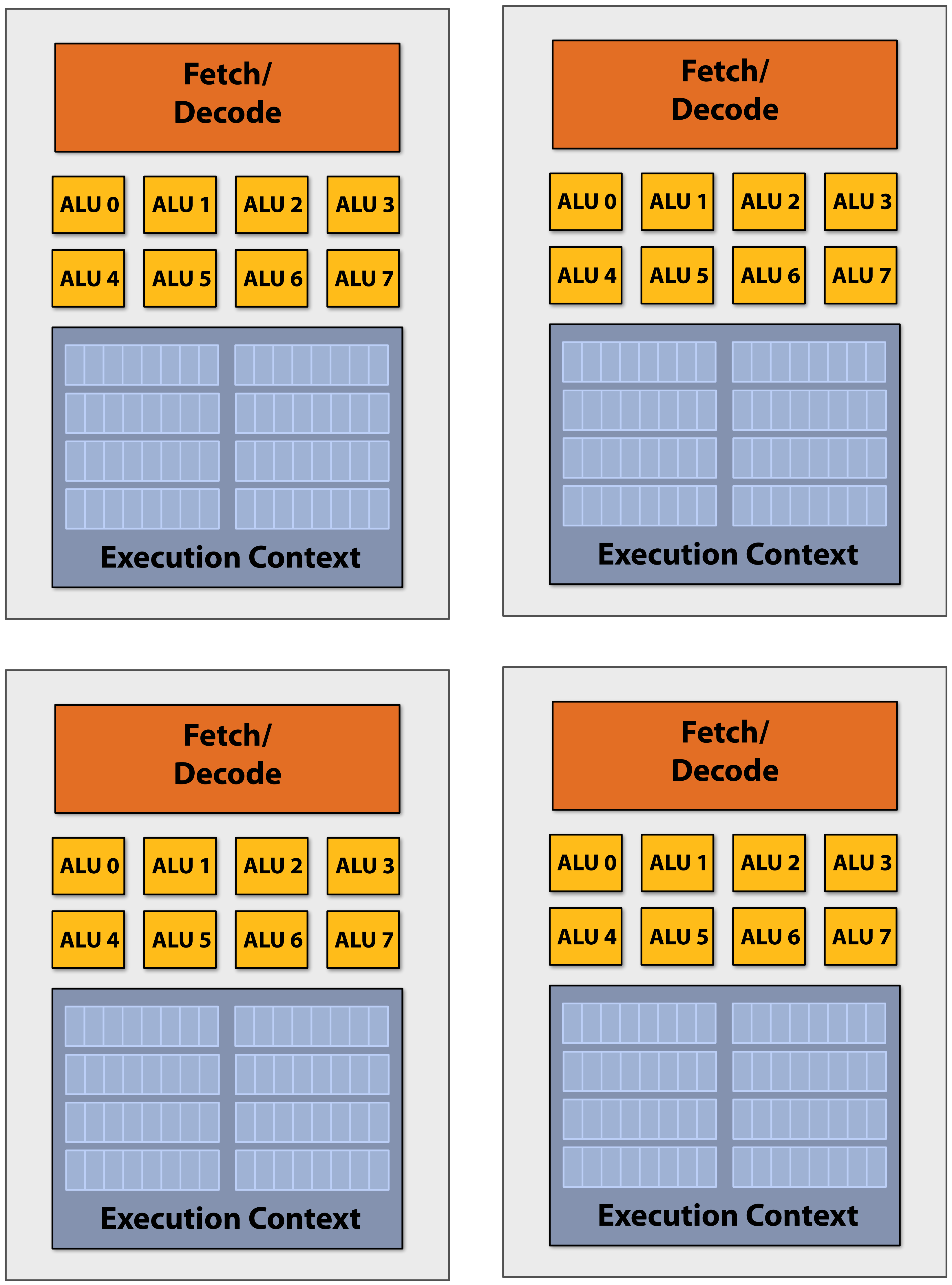
After branch: continue at full performance



(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: Intel Core i7



4 cores

8 SIMD ALUs per core

(AVX instructions)

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.

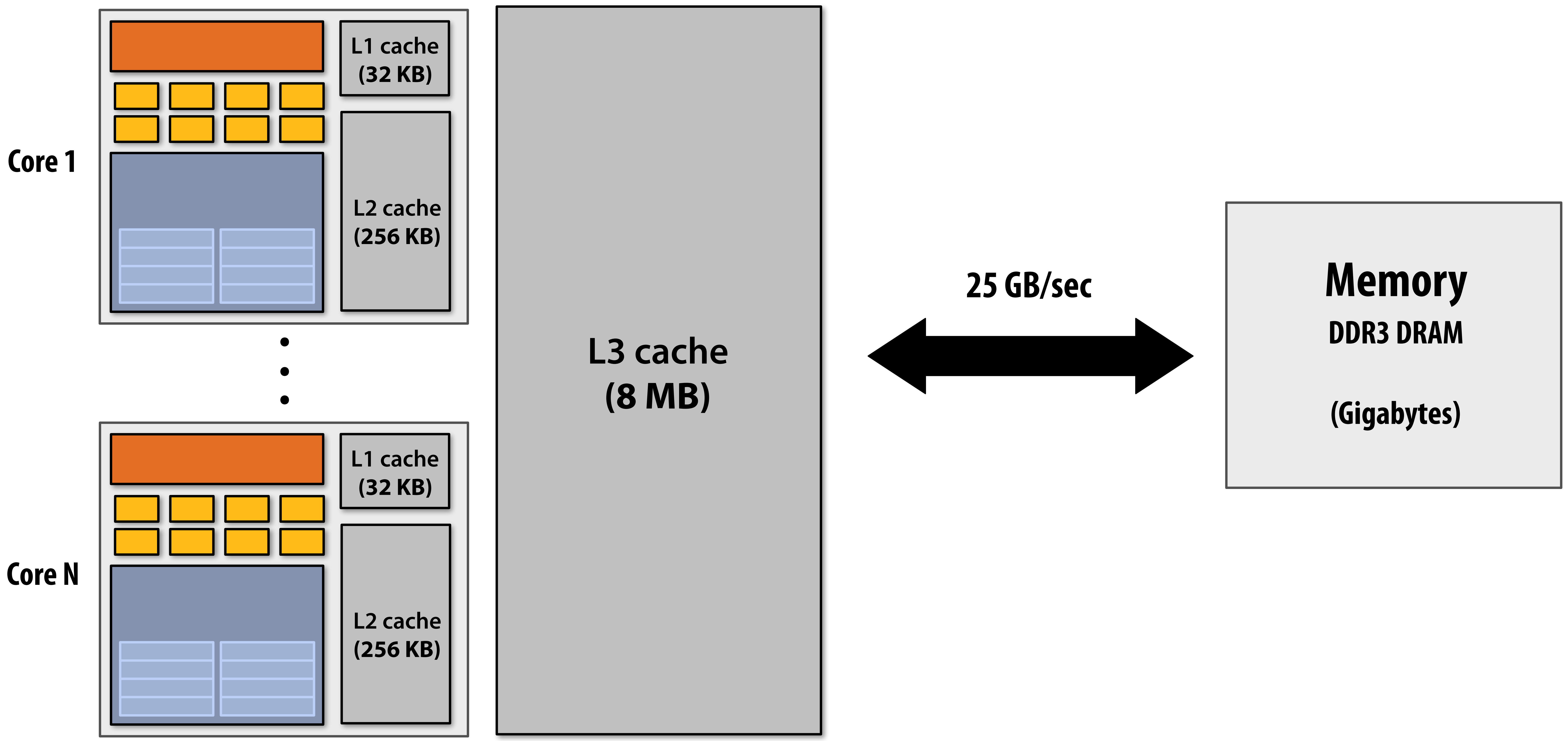
- Accessing memory is a major source of stalls

```
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

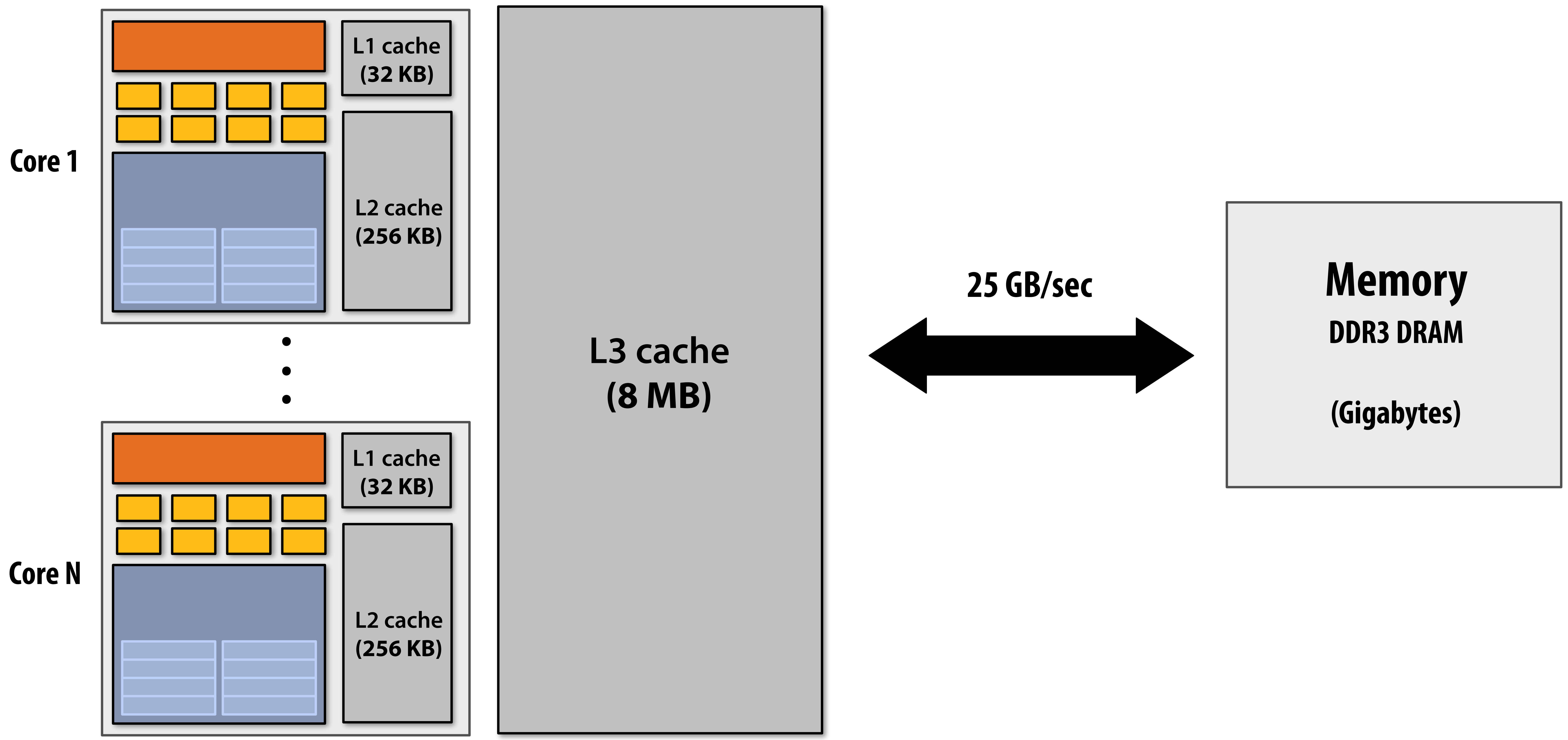
Review: why do modern processors have caches?



Caches reduce length of stalls (reduce latency)

Processors run efficiently when data is resident in caches

Caches reduce memory access latency *



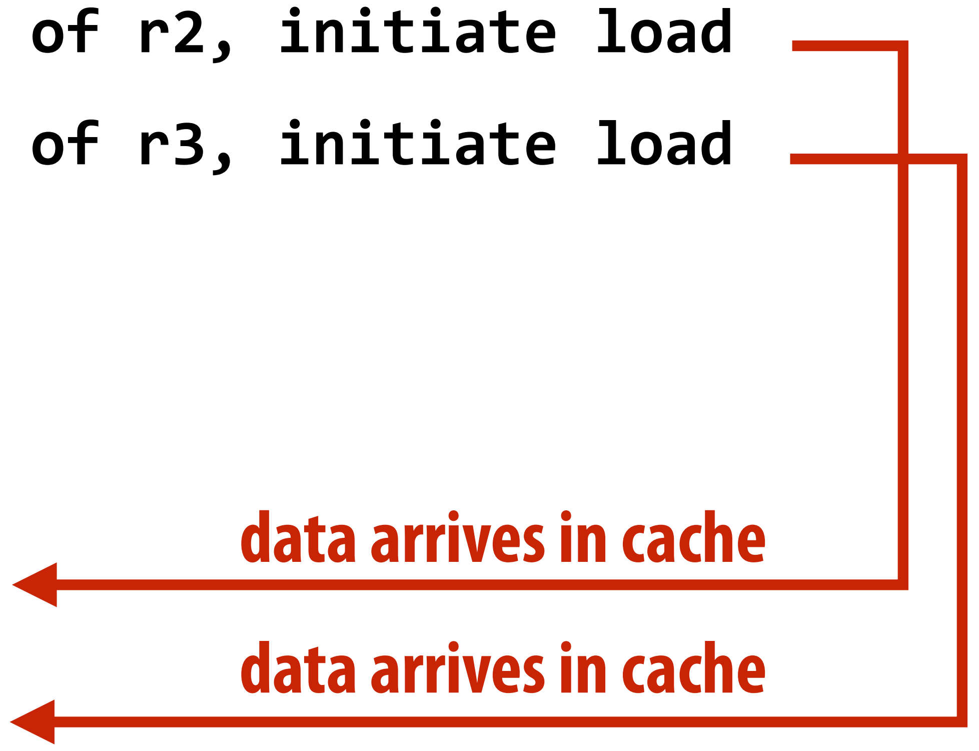
*** Caches also provide high bandwidth data transfer to CPU**

Prefetching reduces stalls (<u>hides</u> 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
```



These loads are cache hits

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

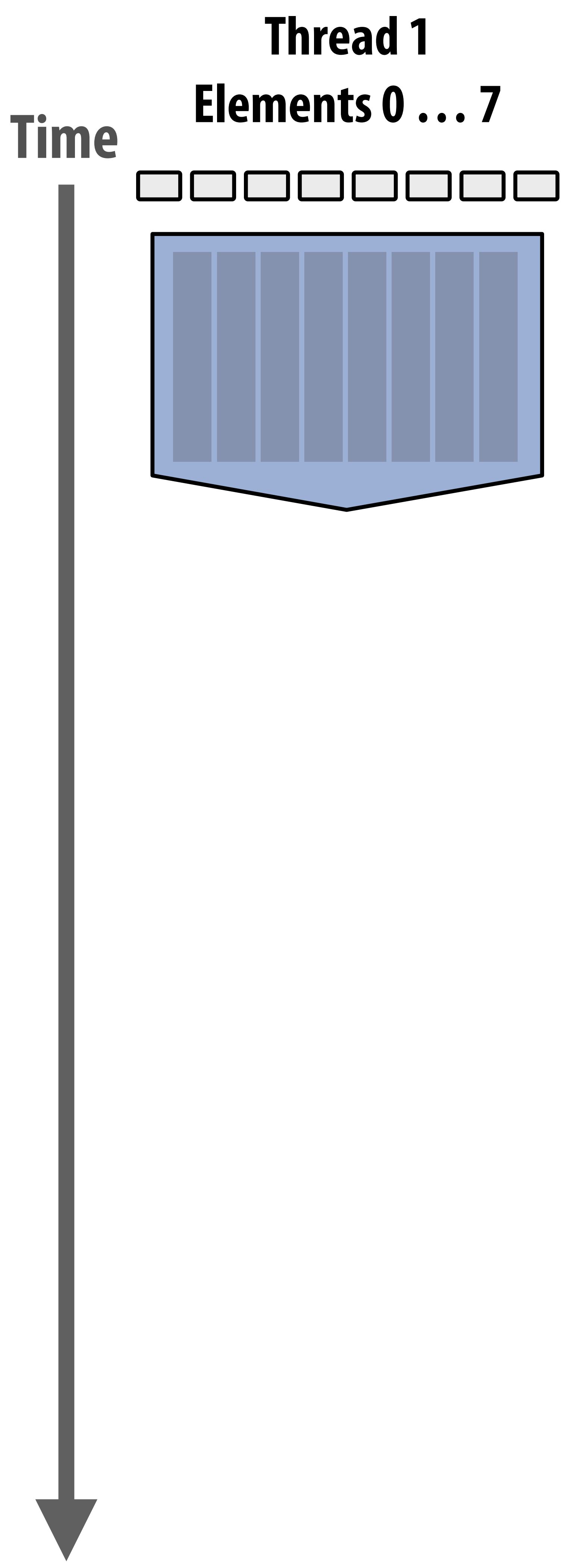
(more detail later in course)

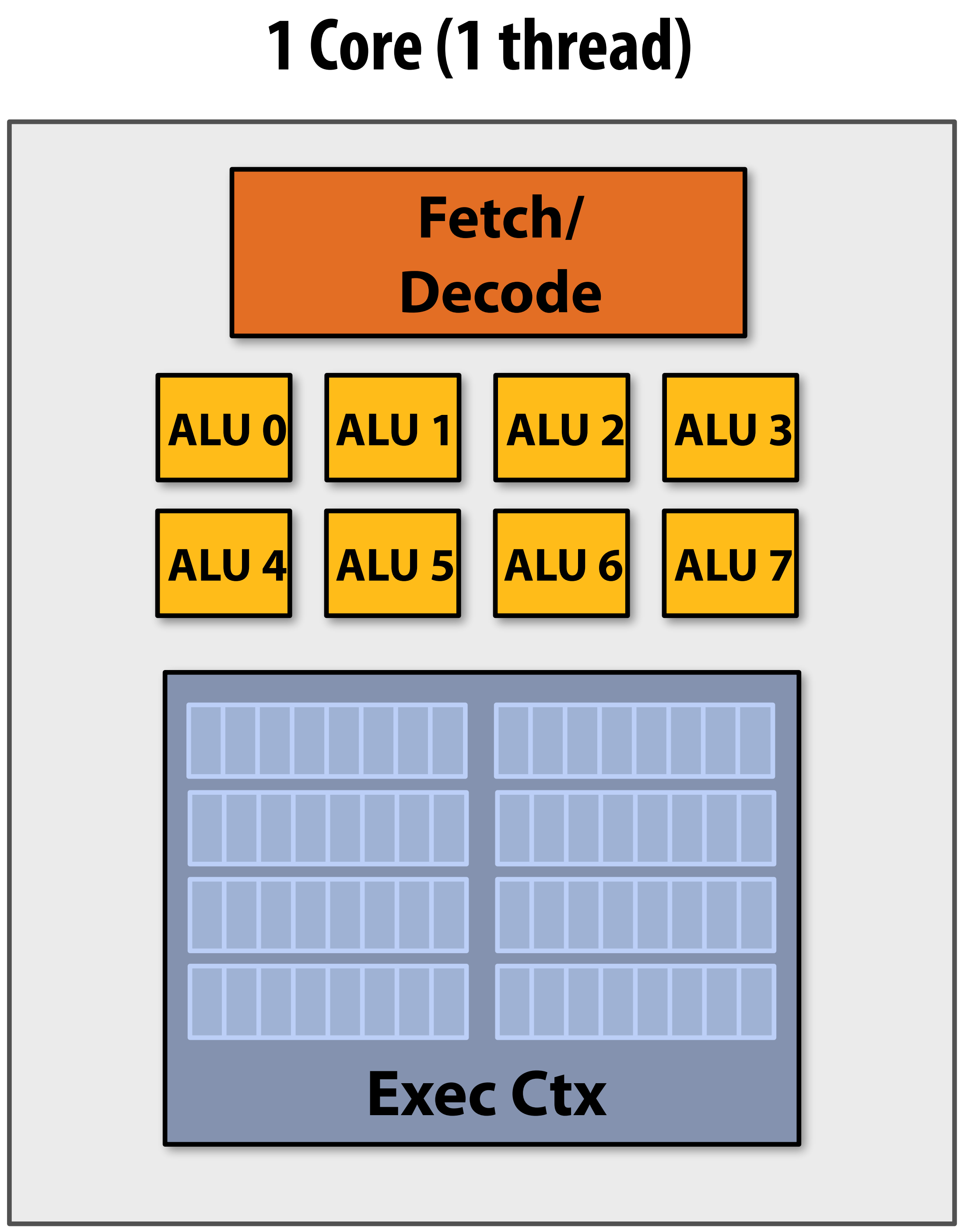
Multi-threading reduces stalls

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

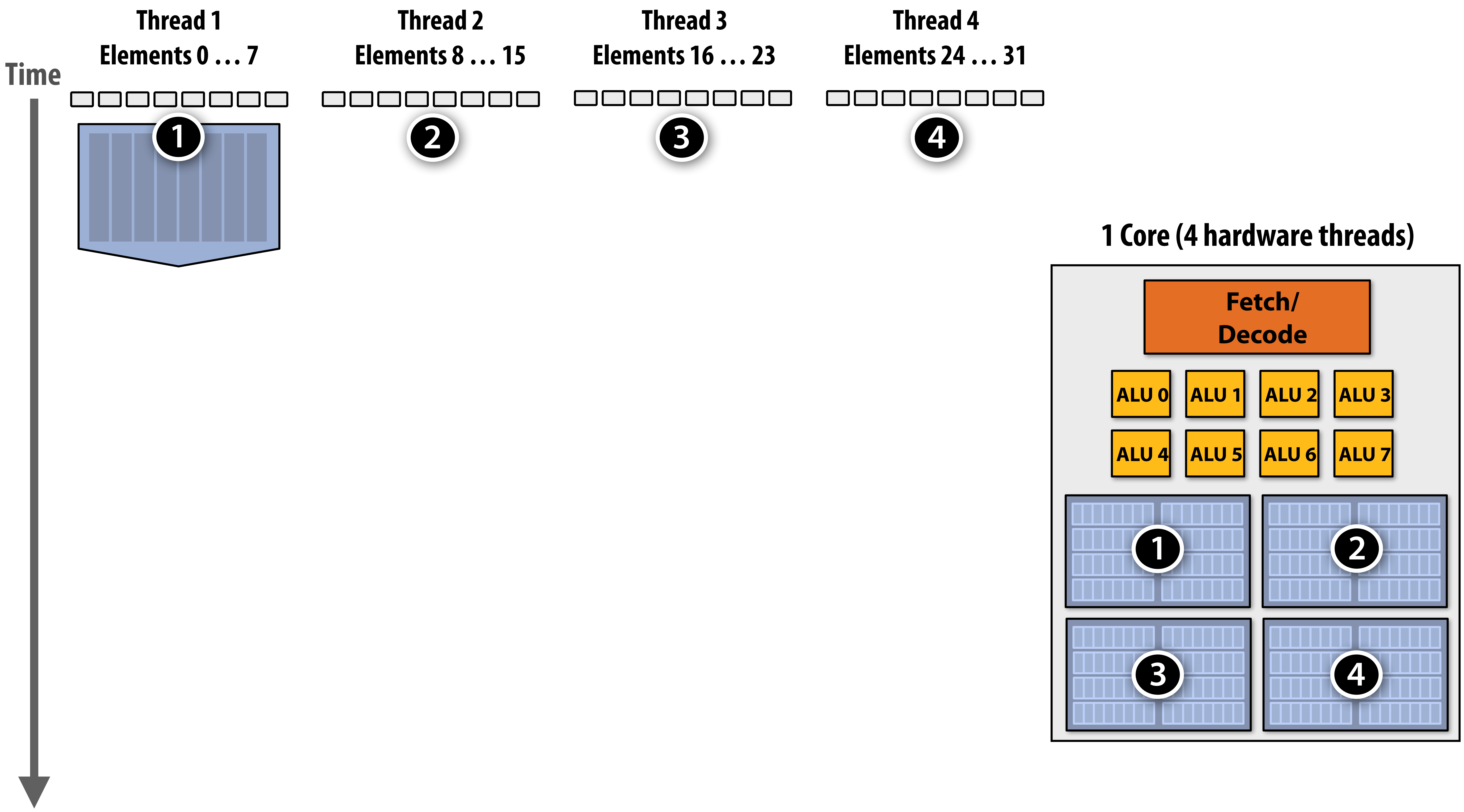
- Like prefetching, multi-threading is a latency hiding, not a latency reducing technique

Hiding stalls with multi-threading

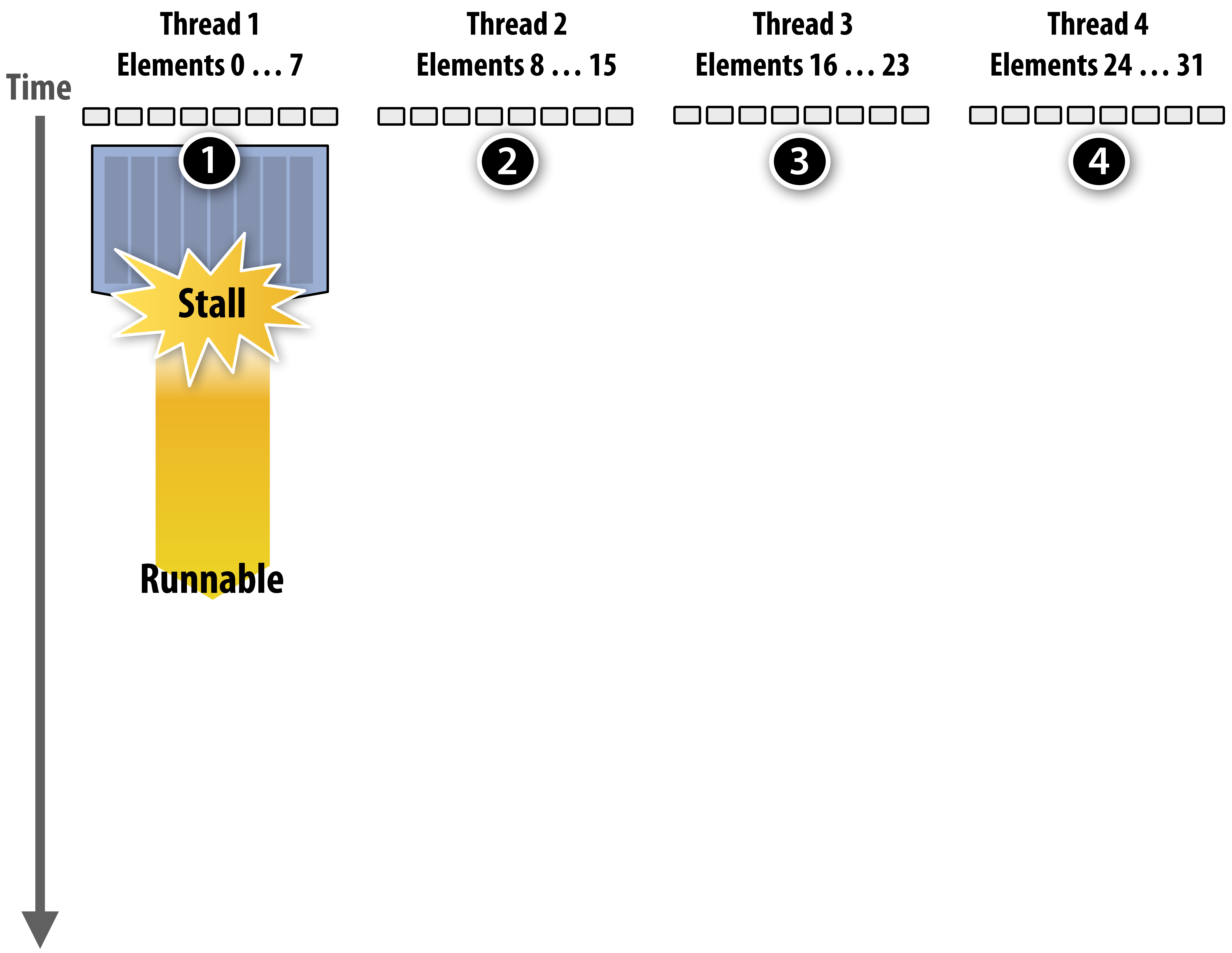


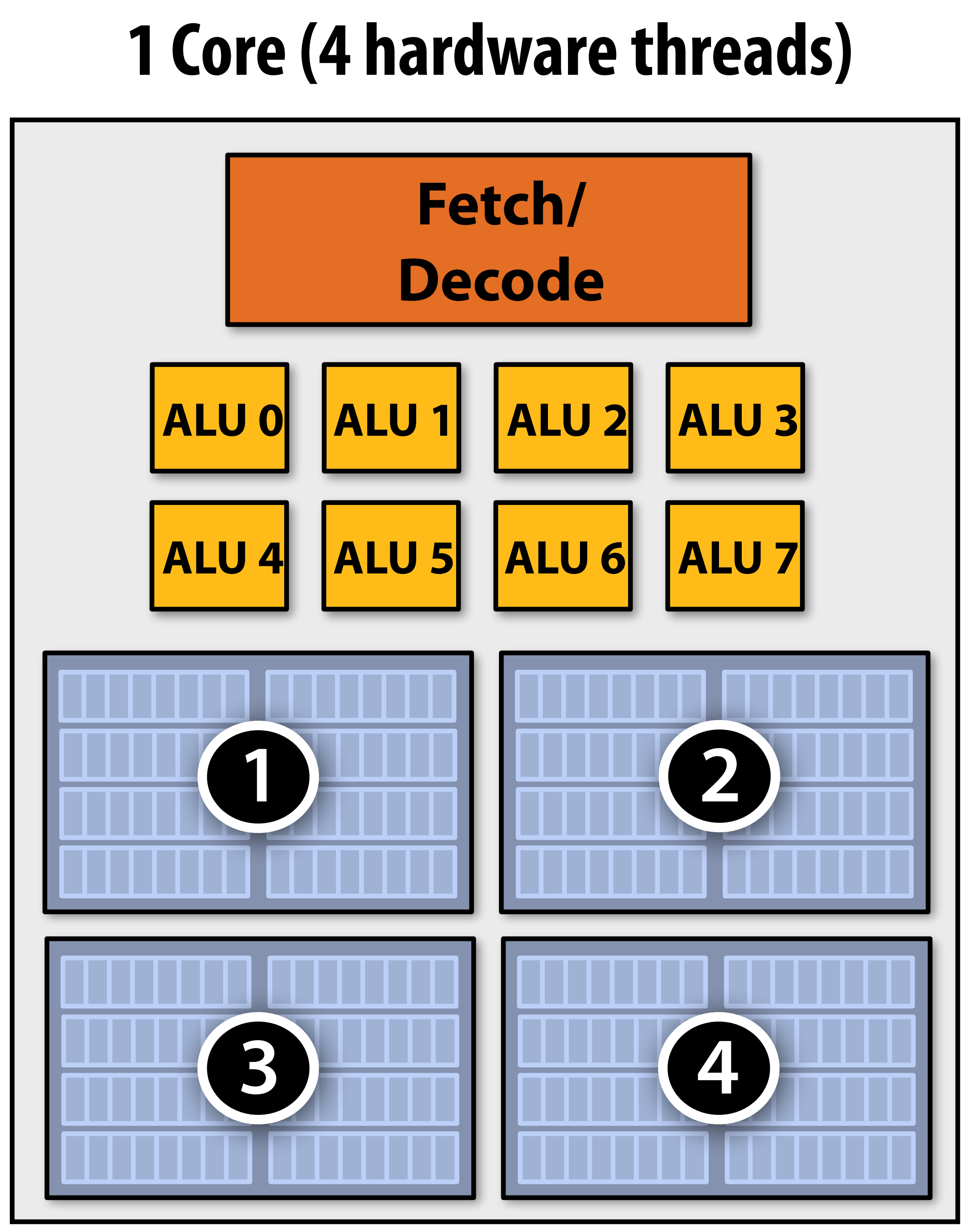


Hiding stalls with multi-threading

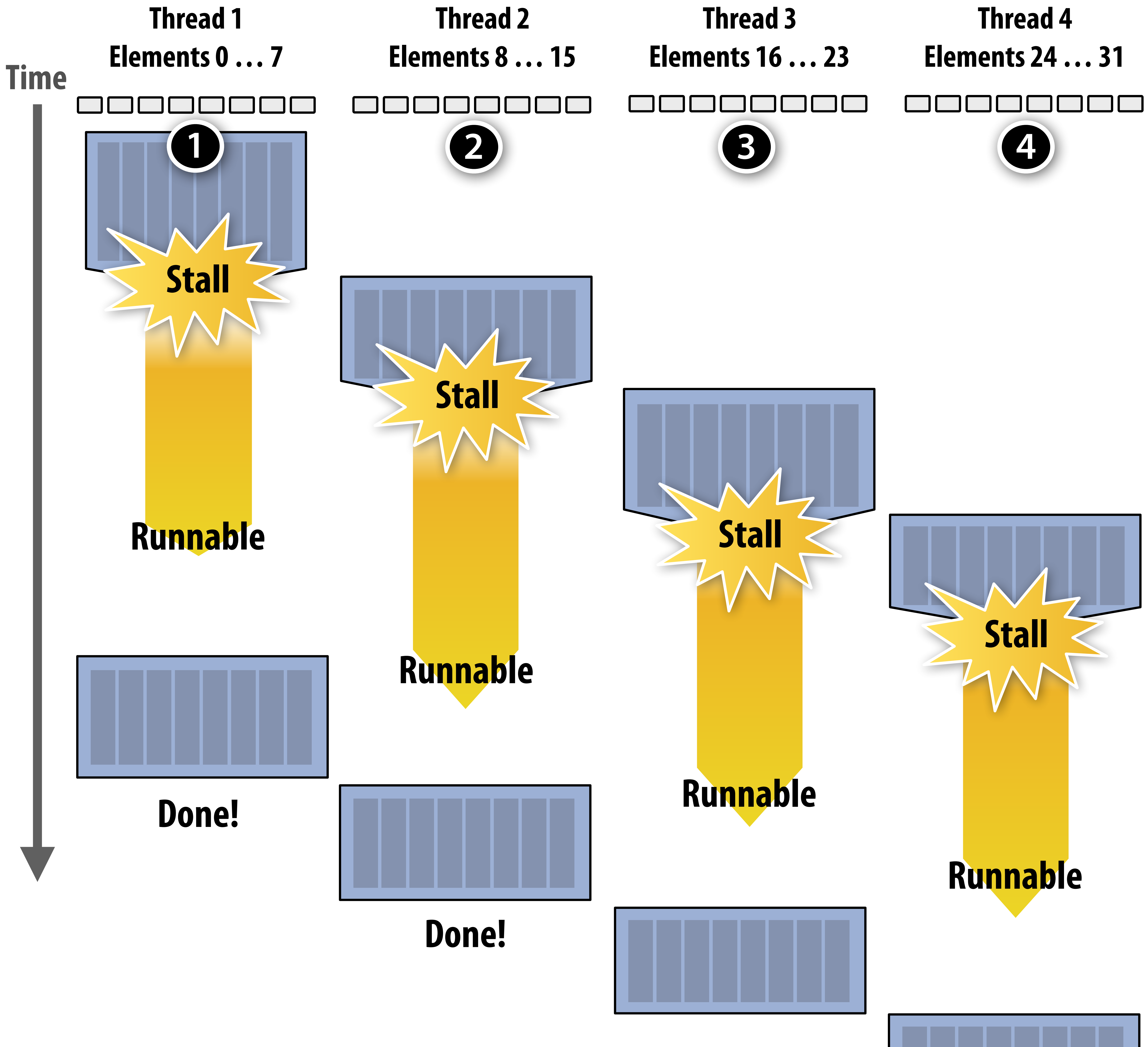


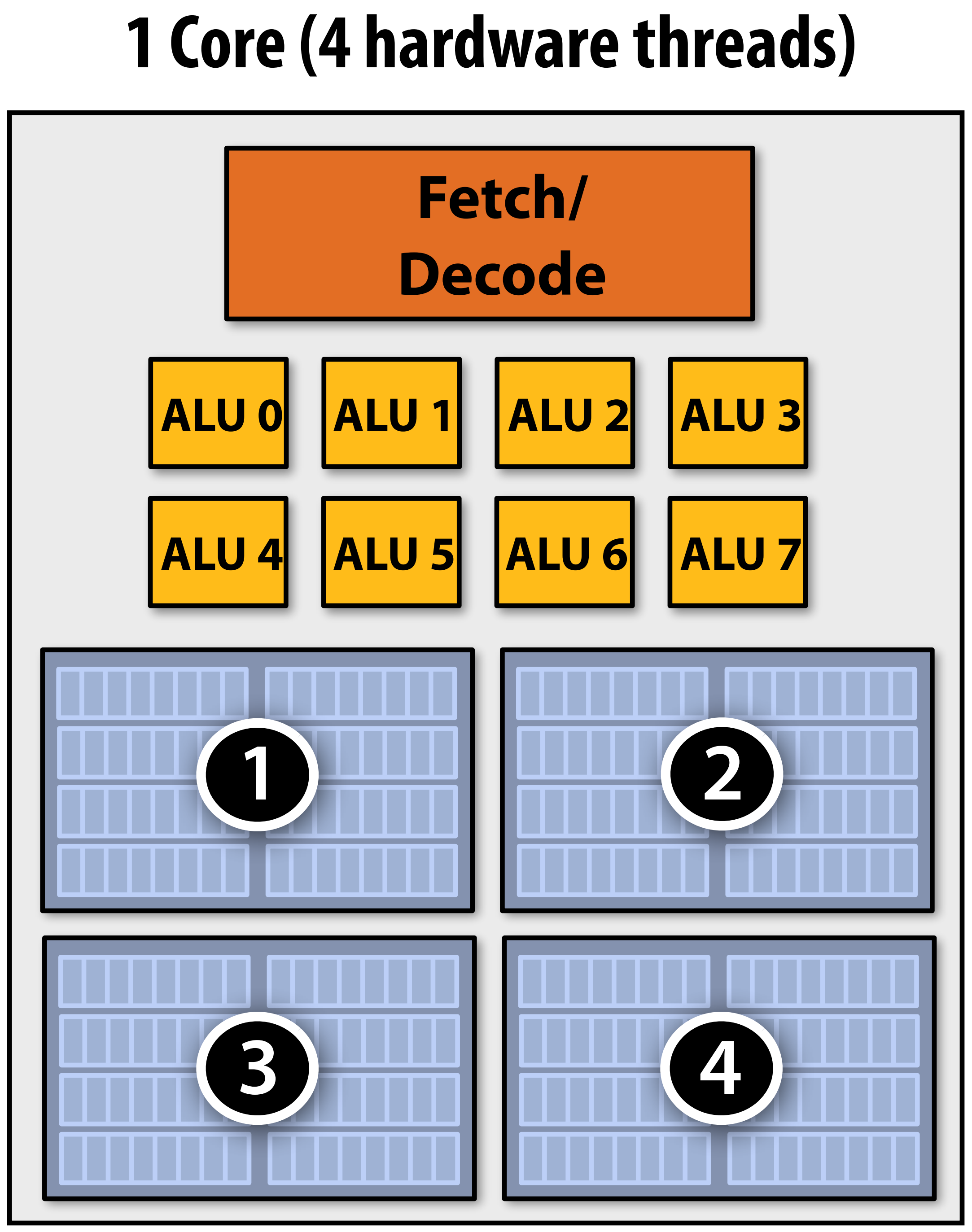
Hiding stalls with multi-threading



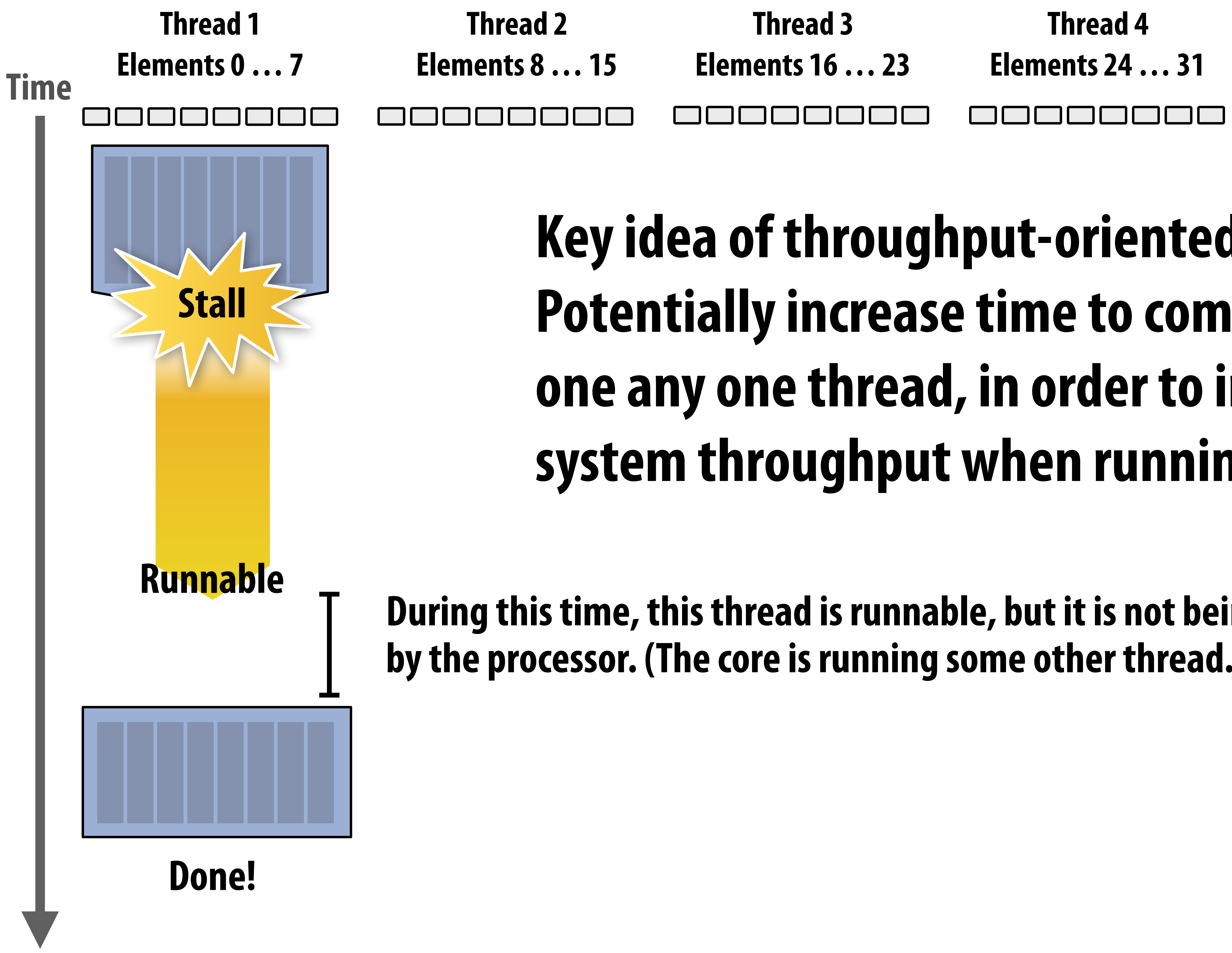


Hiding stalls with multi-threading





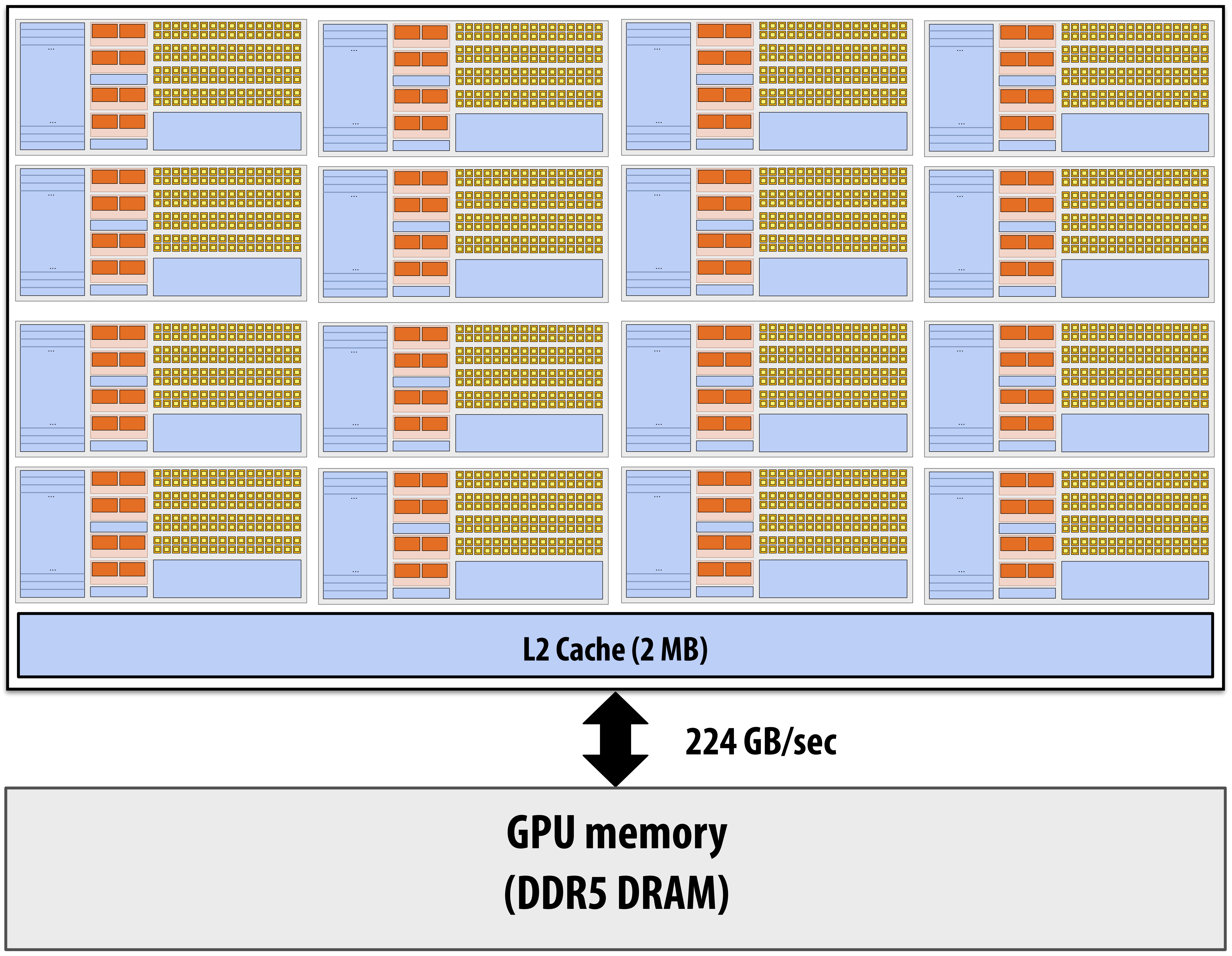
Throughput computing trade-off



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.)

NVIDIA GTX 980 (2014)



1.1 GHz clock

16 SMM "cores" per chip

16 x 128 = 2,048 SIMD mul-add ALUs
= 4.6 TFLOPs

64-way multi-threading per SMM

32-SIMD execution

Translating to CUDA/OpenCL-speak for those familiar with programming GPUs:
1 warp = 32 CUDA threads (1 warp ~ hardware thread, 1 CUDA thread ~ 1 SIMD vector lane)
64 warps per SMM
16 x 64 = 1024 interleaved warps per chip (32,768 CUDA threads/chip, a.k.a. "32K pixels at once")

Another example:
for review and to check your understanding

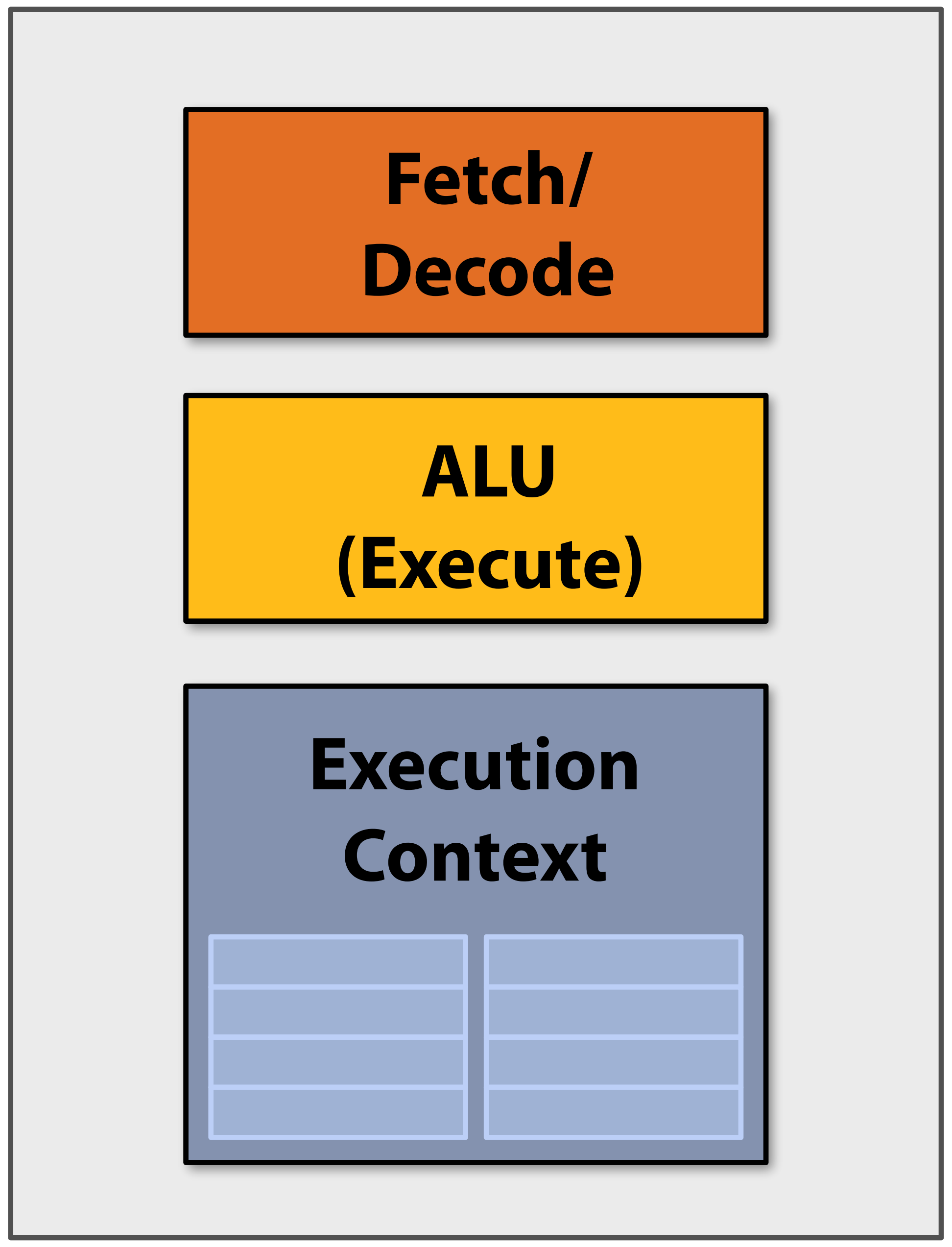
(if you understand the following sequence you understand this lecture)

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

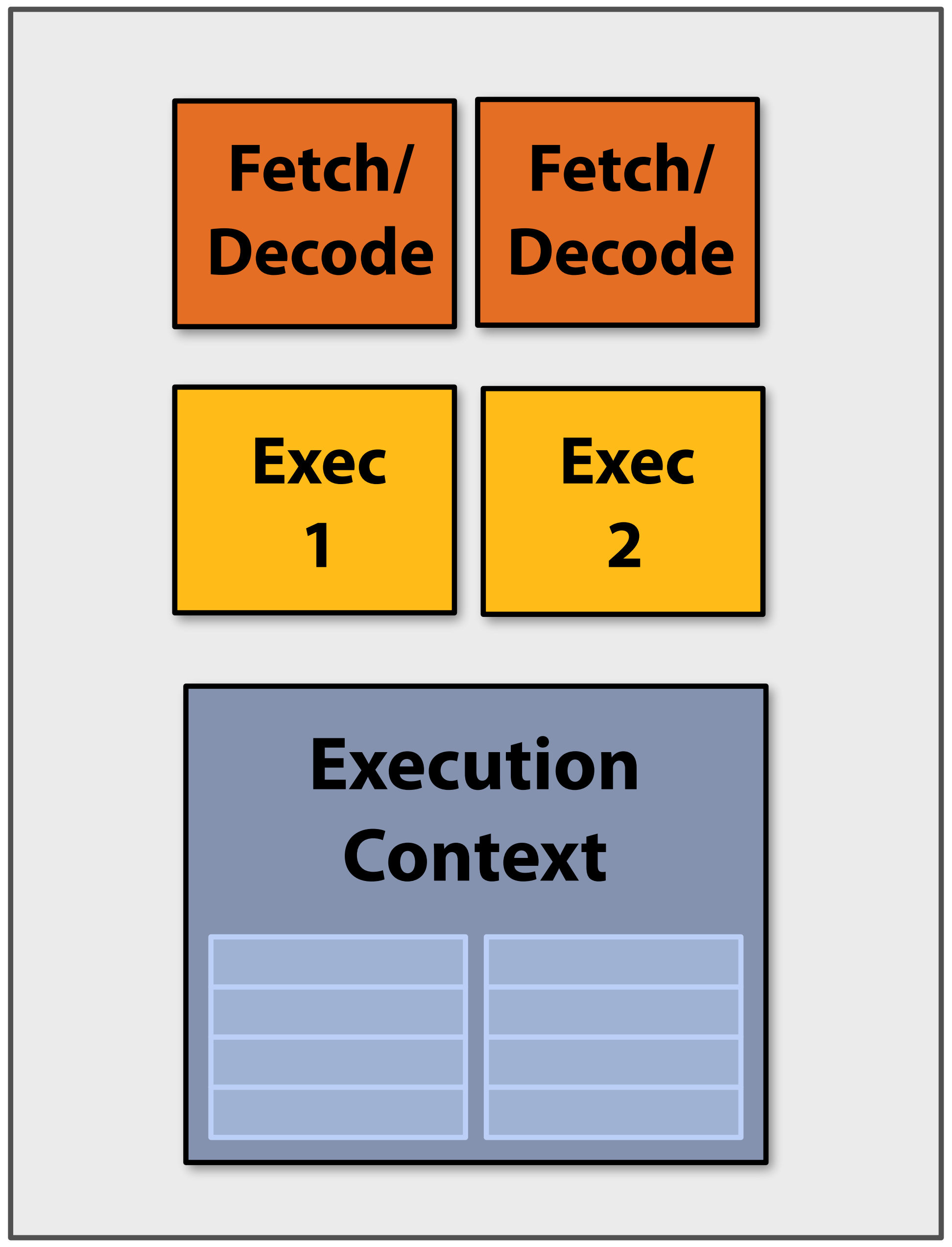
Unmodified program

```
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.

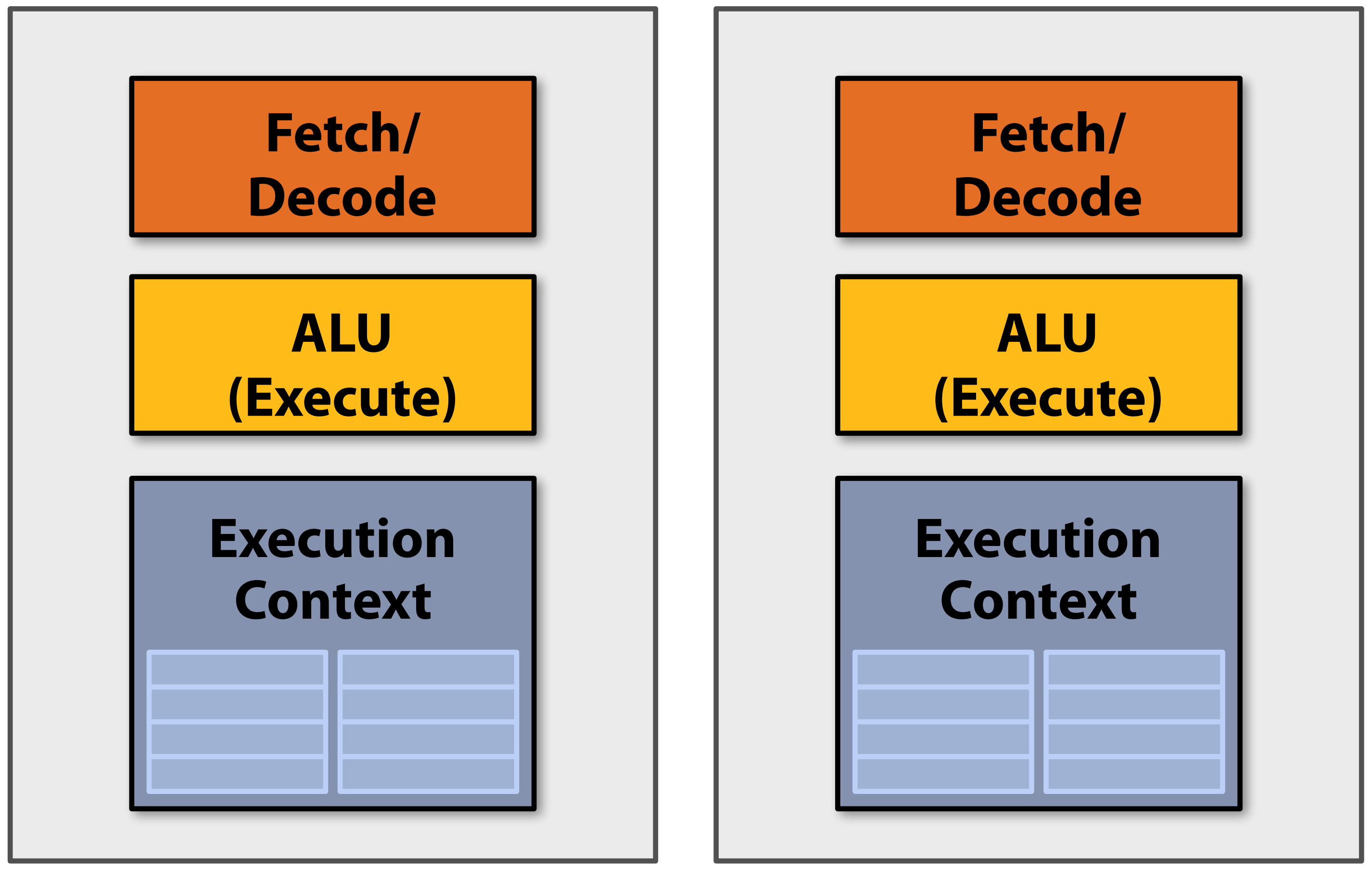


Review: multi-core execution (two cores)

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;

    pthread_create(&thread_id, NULL, my_thread_start, &args); // launch thread
    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.

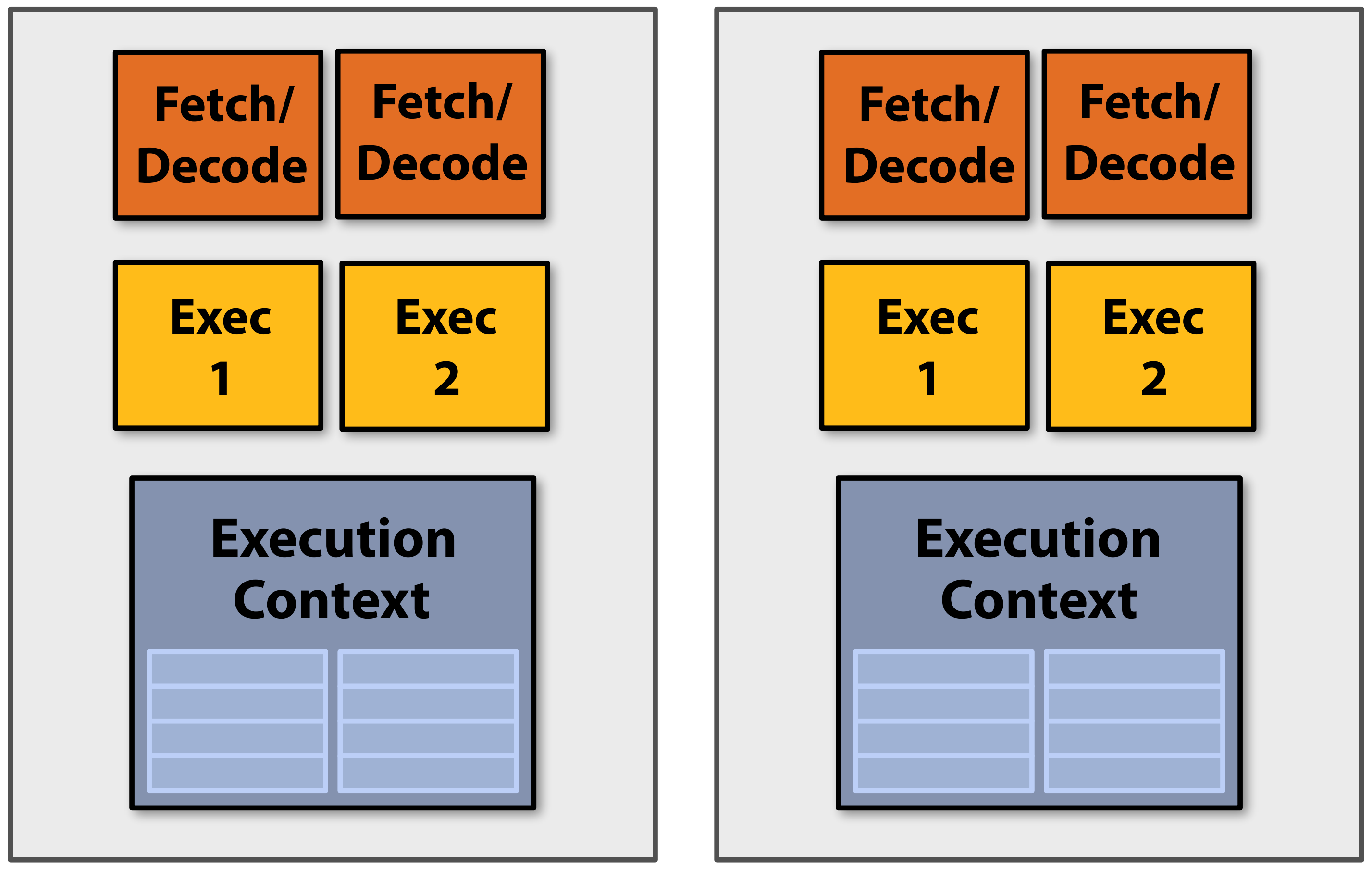


Review: multi-core + superscalar execution

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;

    pthread_create(&thread_id, NULL, my_thread_start, &args); // launch thread
    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.

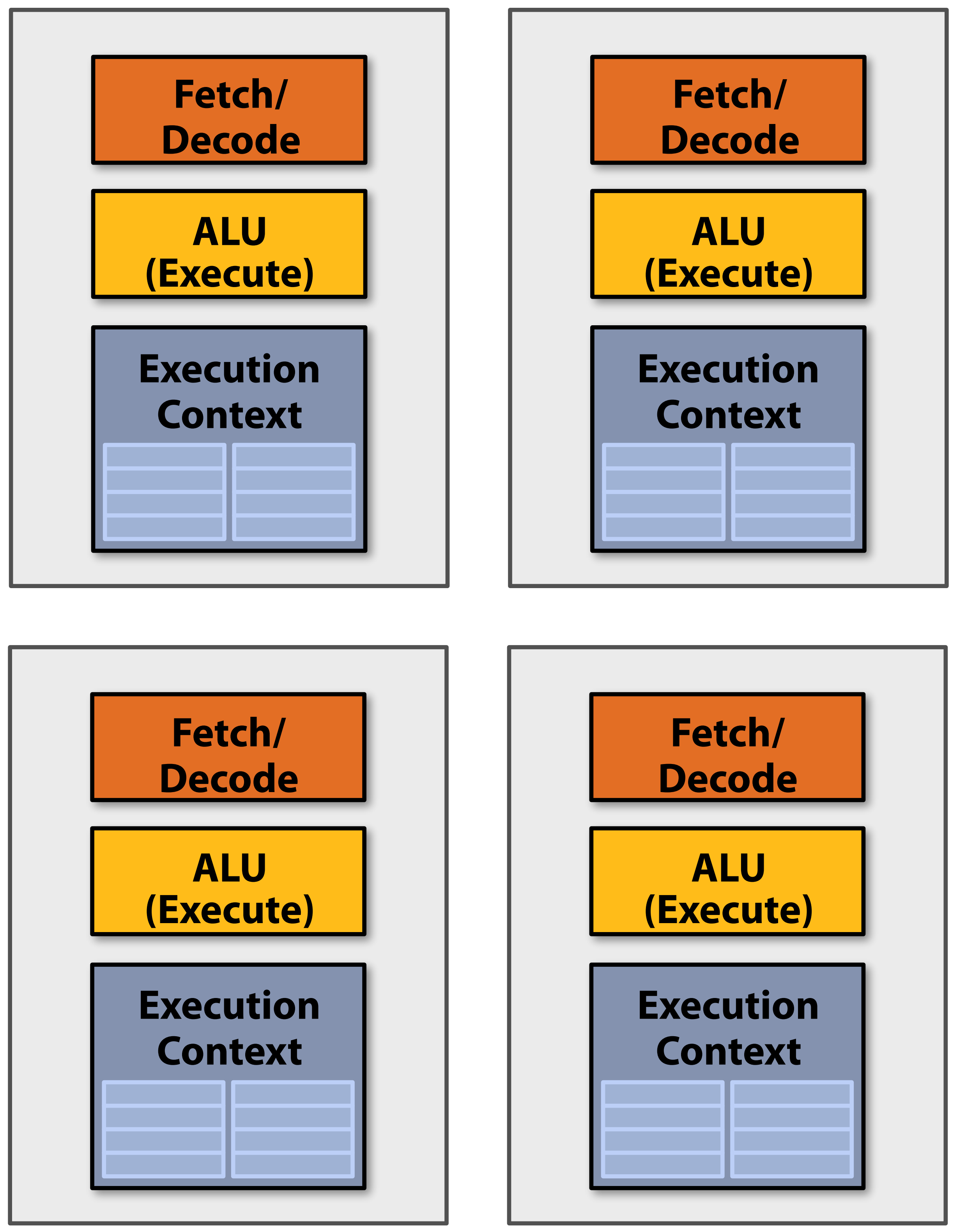


Review: multi-core (four cores)

Modify program to create many threads of control: recall Kayvon's fictitious 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 each core.



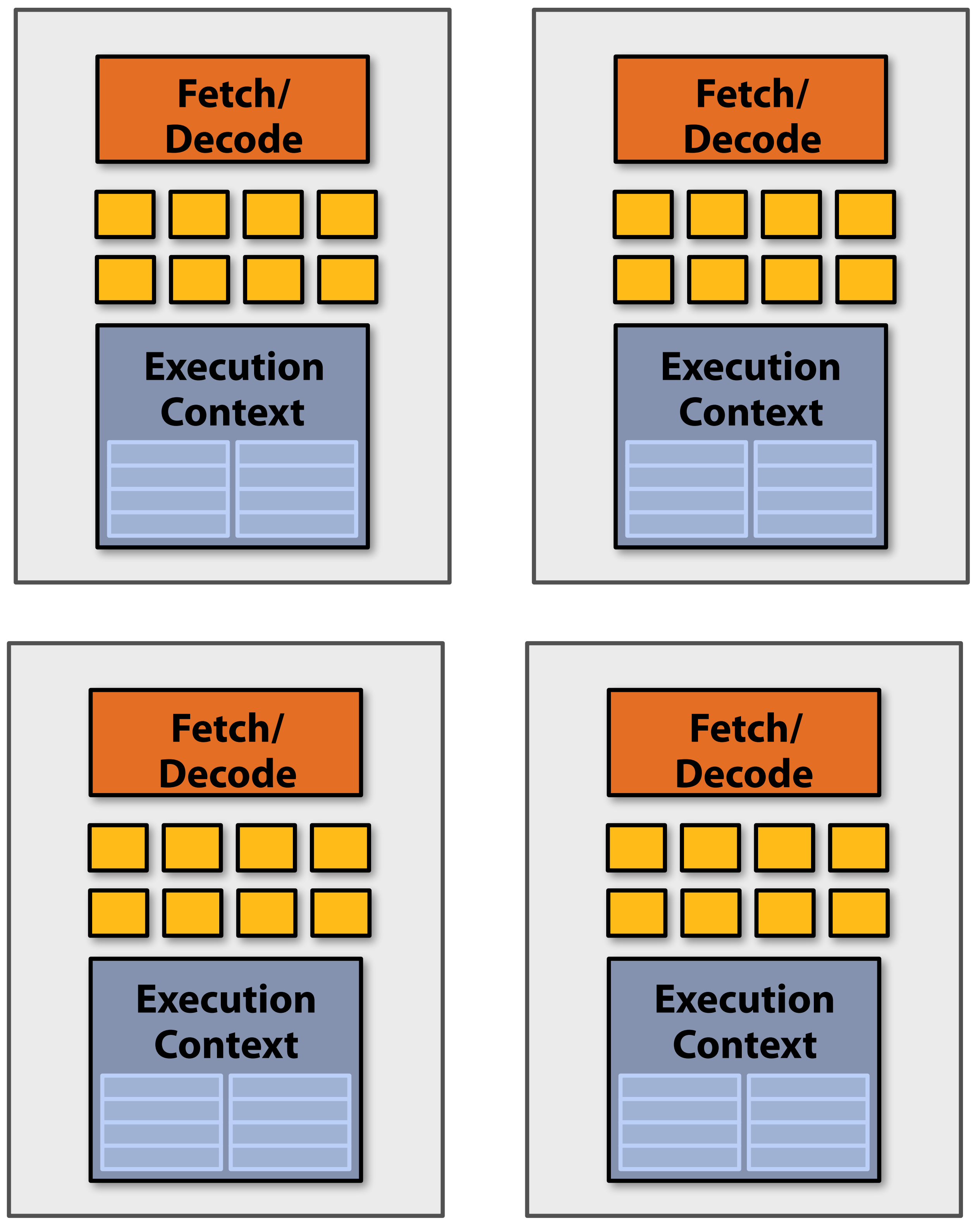
Review: four, 8-wide SIMD cores

Observation: program must execute many iterations of the <u>same</u> 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 <u>each</u> core.



Review: four SIMD, multi-threaded cores

Observation: memory operations have very long latency

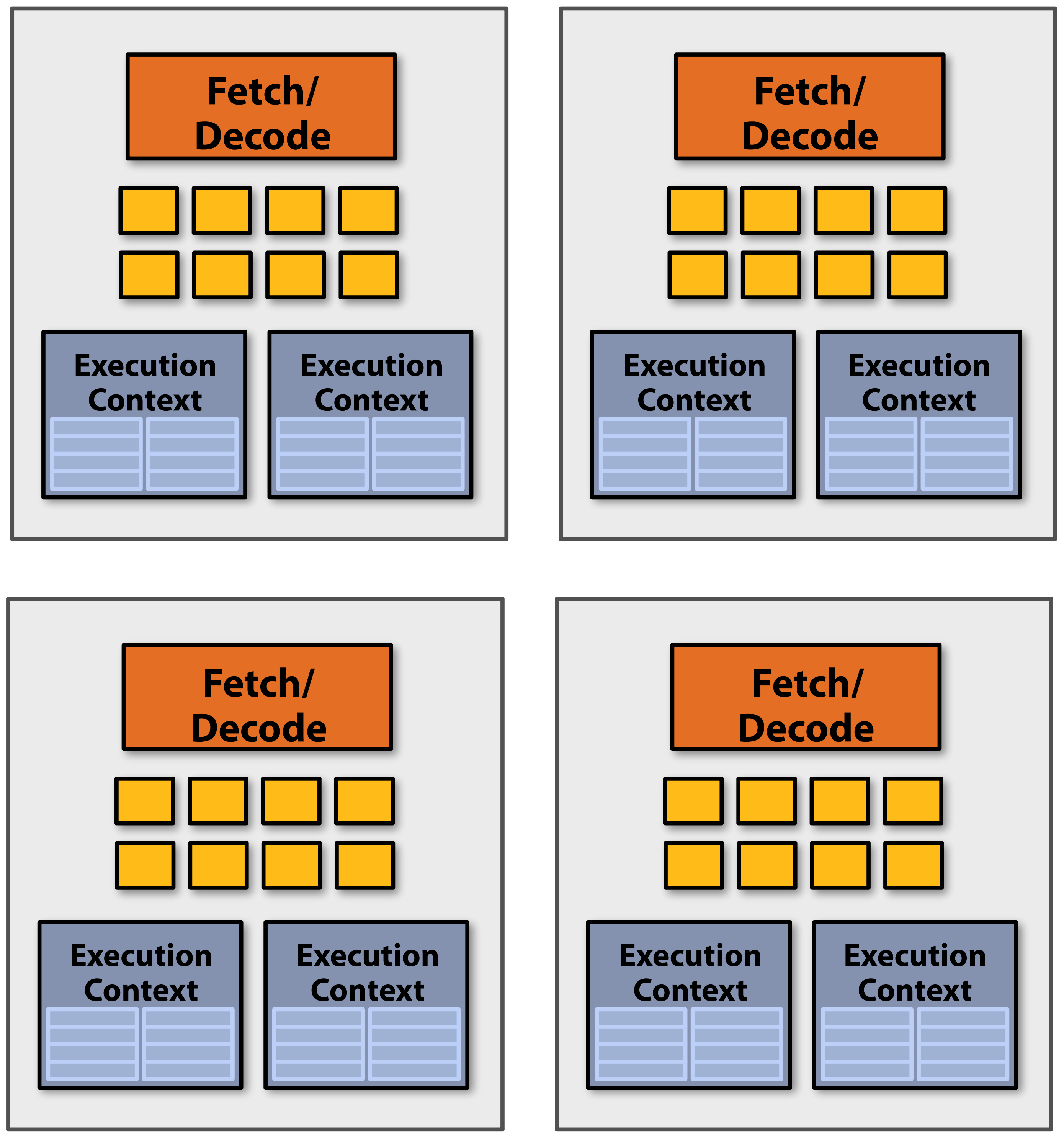
Solution: hide latency of loading data for one iteration by executing arithmetic instructions from other iterations

```
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.

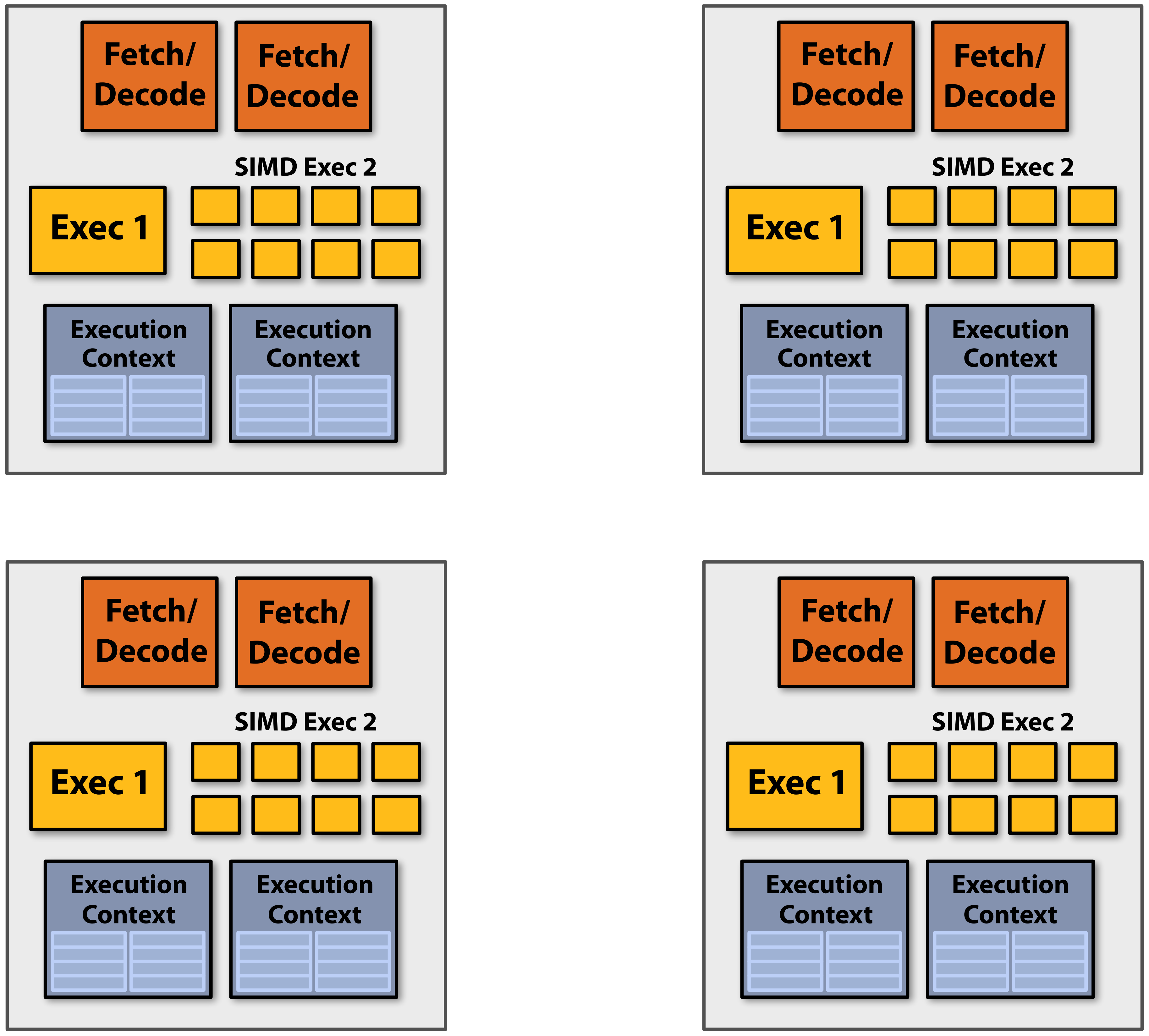


Summary: four superscalar, SIMD, multi-threaded cores

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).

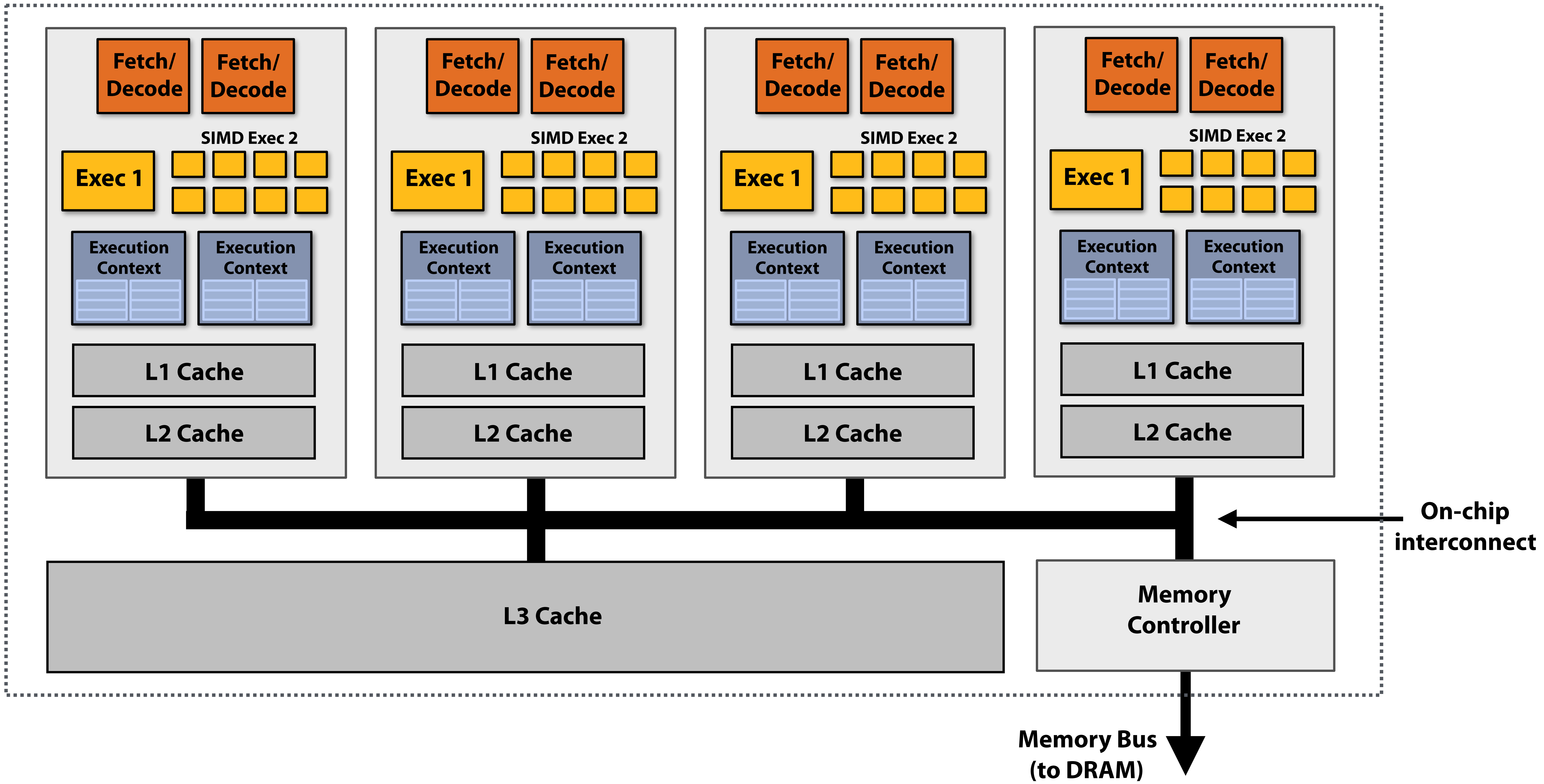
Processor can switch to execute the other instruction stream when faced with stall.



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

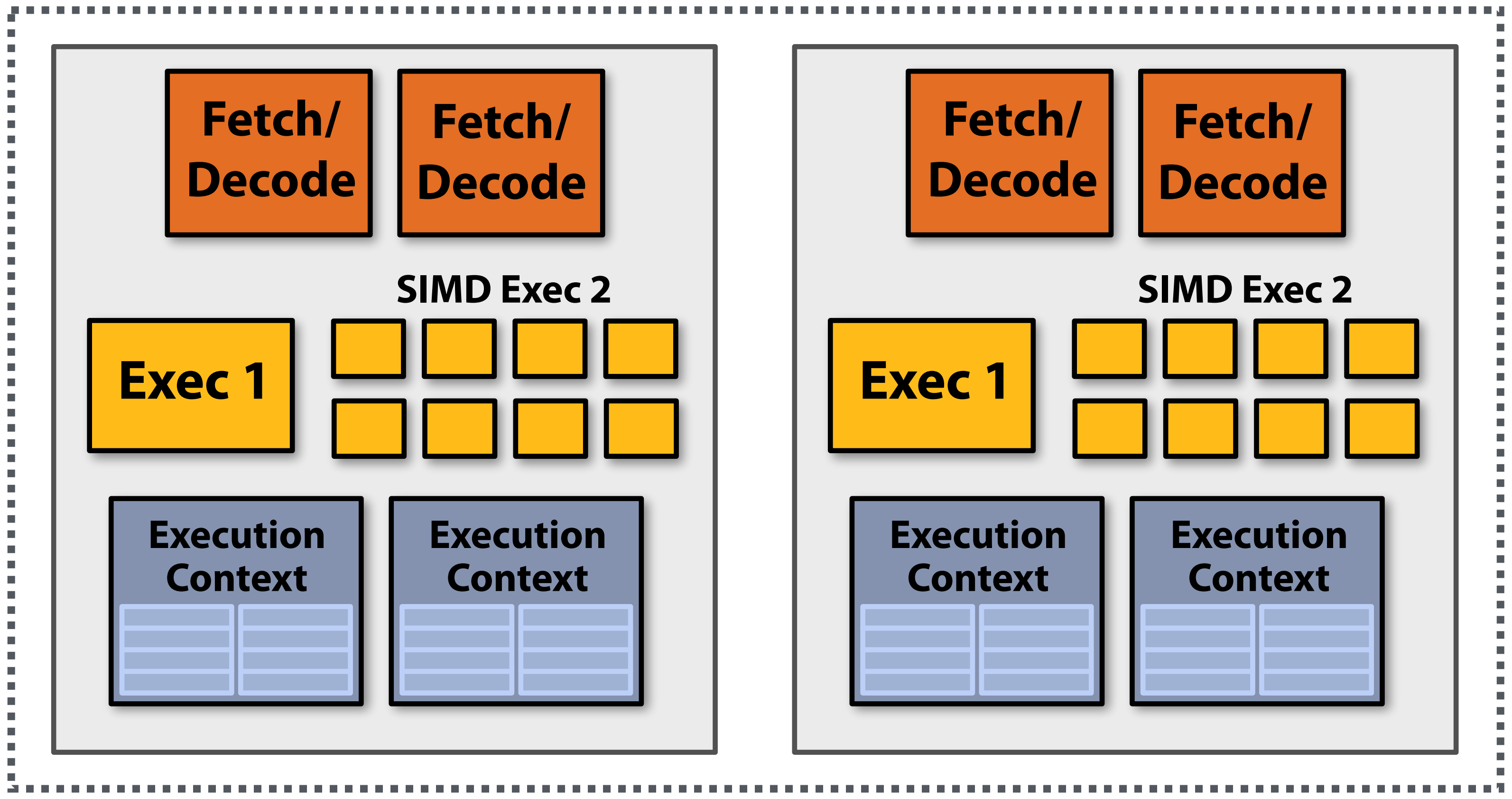
- You write a C application that spawns <u>two</u> pthreads
- The application runs on the processor shown below
  - Two cores, two-execution contexts per core, up to instructions per clock, one instruction is an 8-wide SIMD instruction.

- 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?

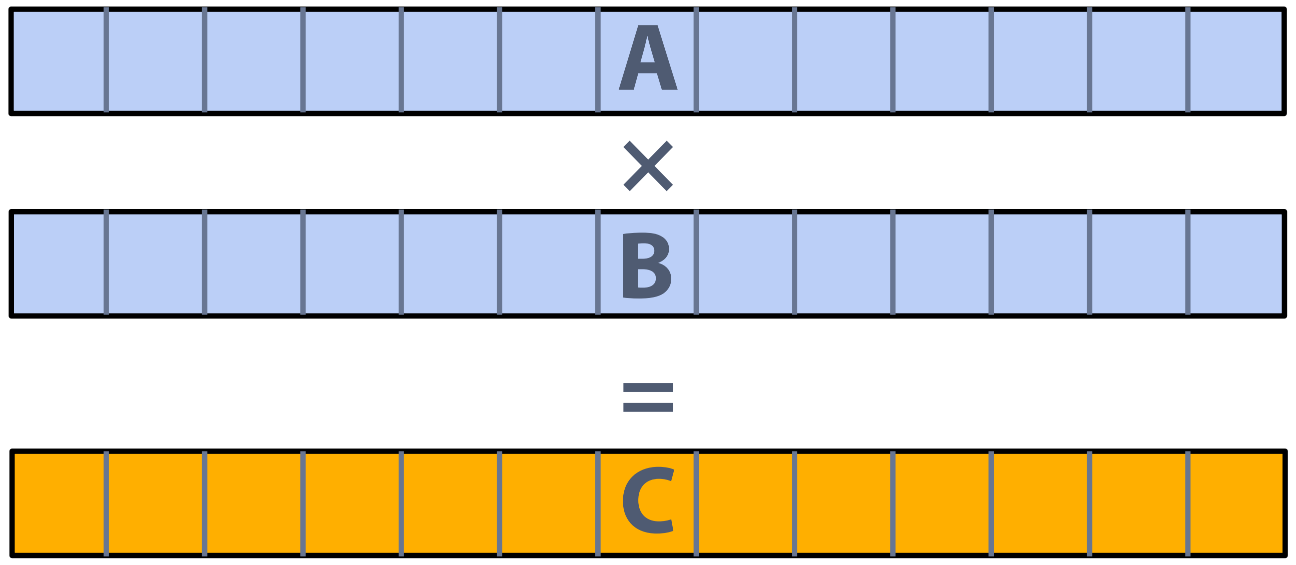


Another thought experiment

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% efficiency… but 10x faster than quad-core CPU!

(4 GHz Core i7 Gen 6 quad-core CPU connected to 34 GB/sec memory bus)

Bandwidth limited!

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

No amount of latency hiding helps this.

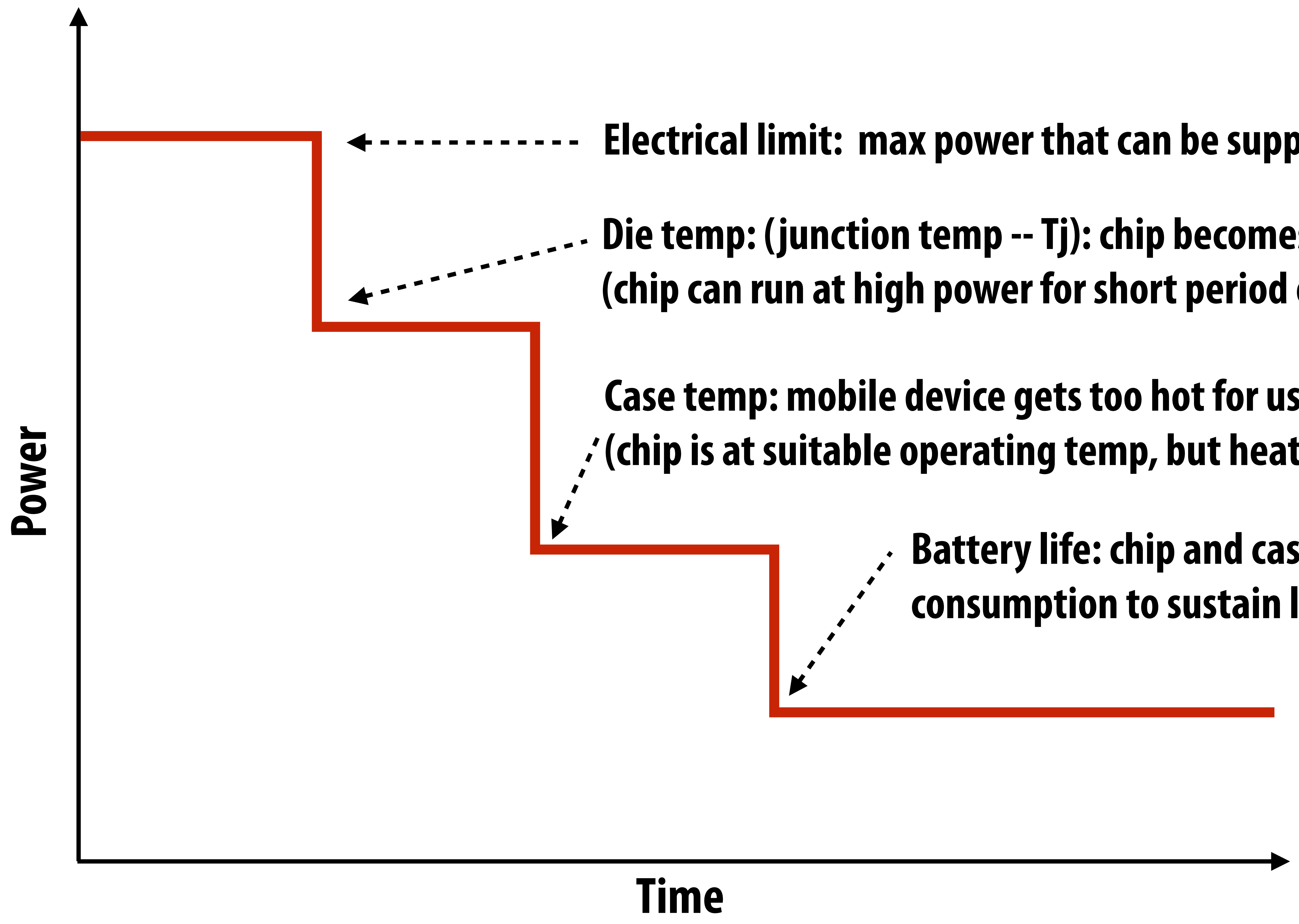
Bandwidth is a critical resource

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

Hardware specialization

Why does energy efficiency matter?

- General in 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)



Electrical limit: max power that can be supplied to chip

Die temp: (junction temp -- Tj): chip becomes unreliable above this temp (chip can run at high power for short period of time until chip heats to Tj)

Case temp: mobile device gets too hot for user to comfortably hold (chip is at suitable operating temp, but heat is dissipating into case)

Battery life: chip and case are cool, but want to reduce power consumption to sustain long battery life for given task

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

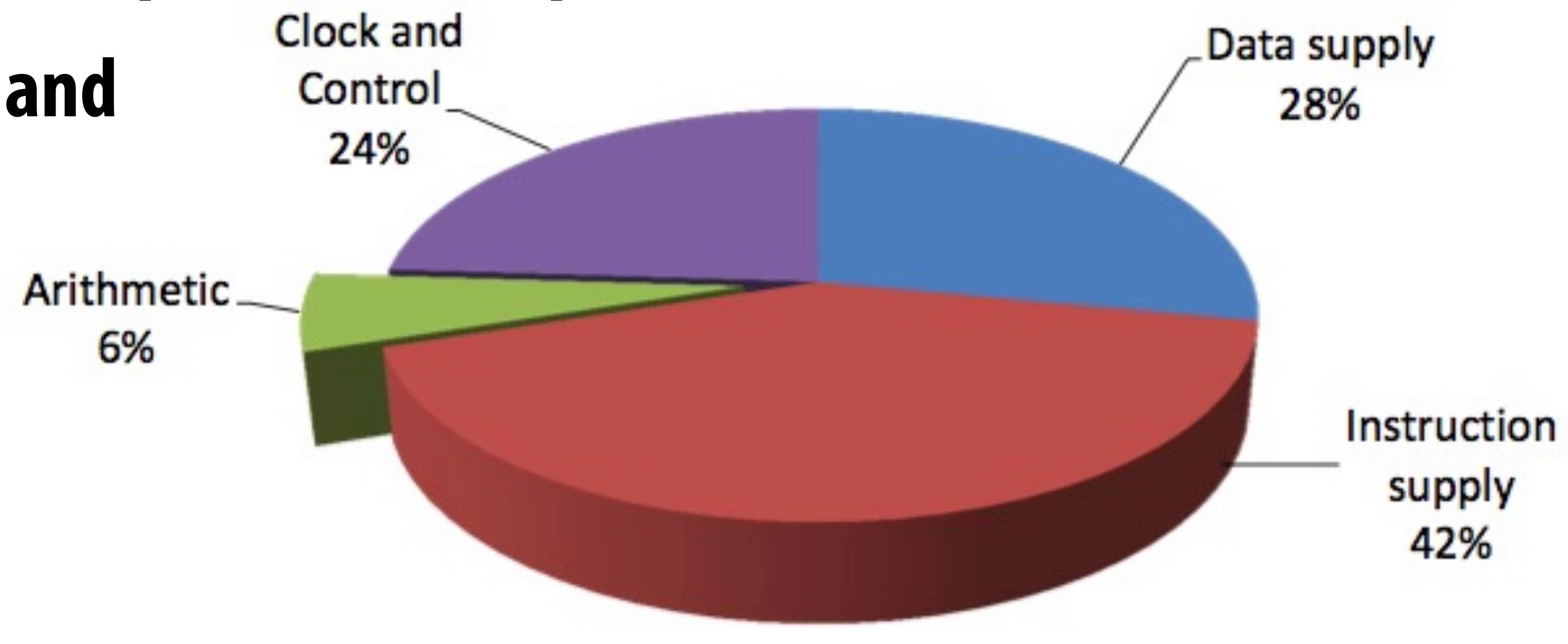
Slide credit: adopted from original slide from M. Shebanow: HPG 2013 keynote

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**

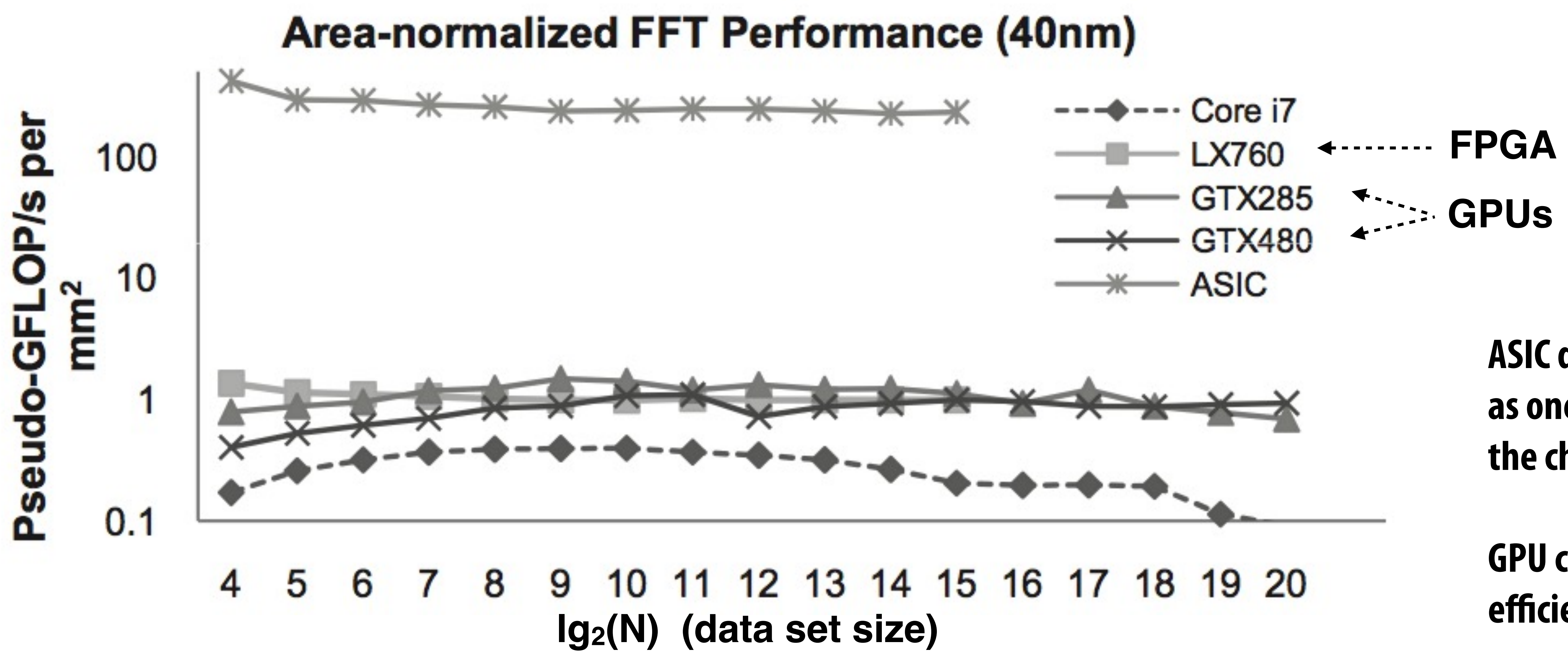


Efficient Embedded Computing [Dally et al. 08]

[Figure credit Eric Chung]

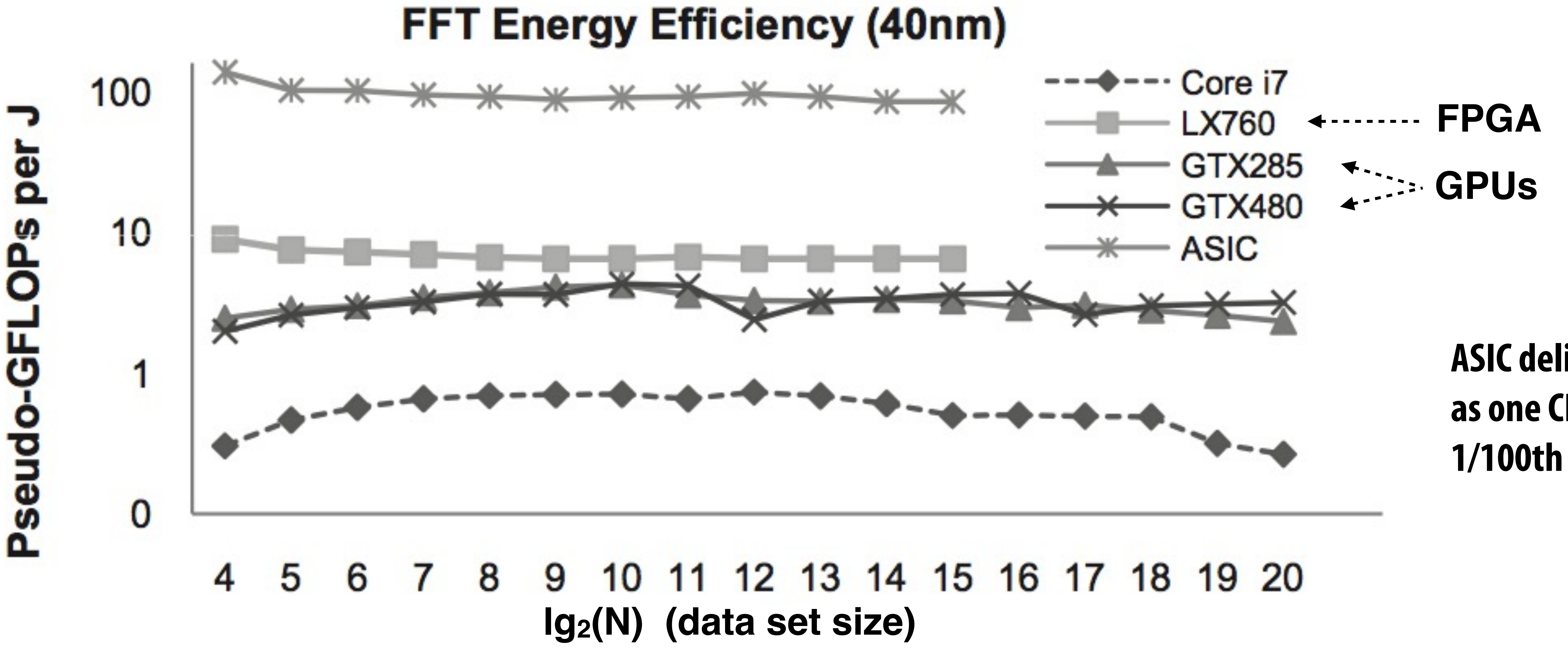
[Source: Chung et al. 2010 , Dally 08]

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]

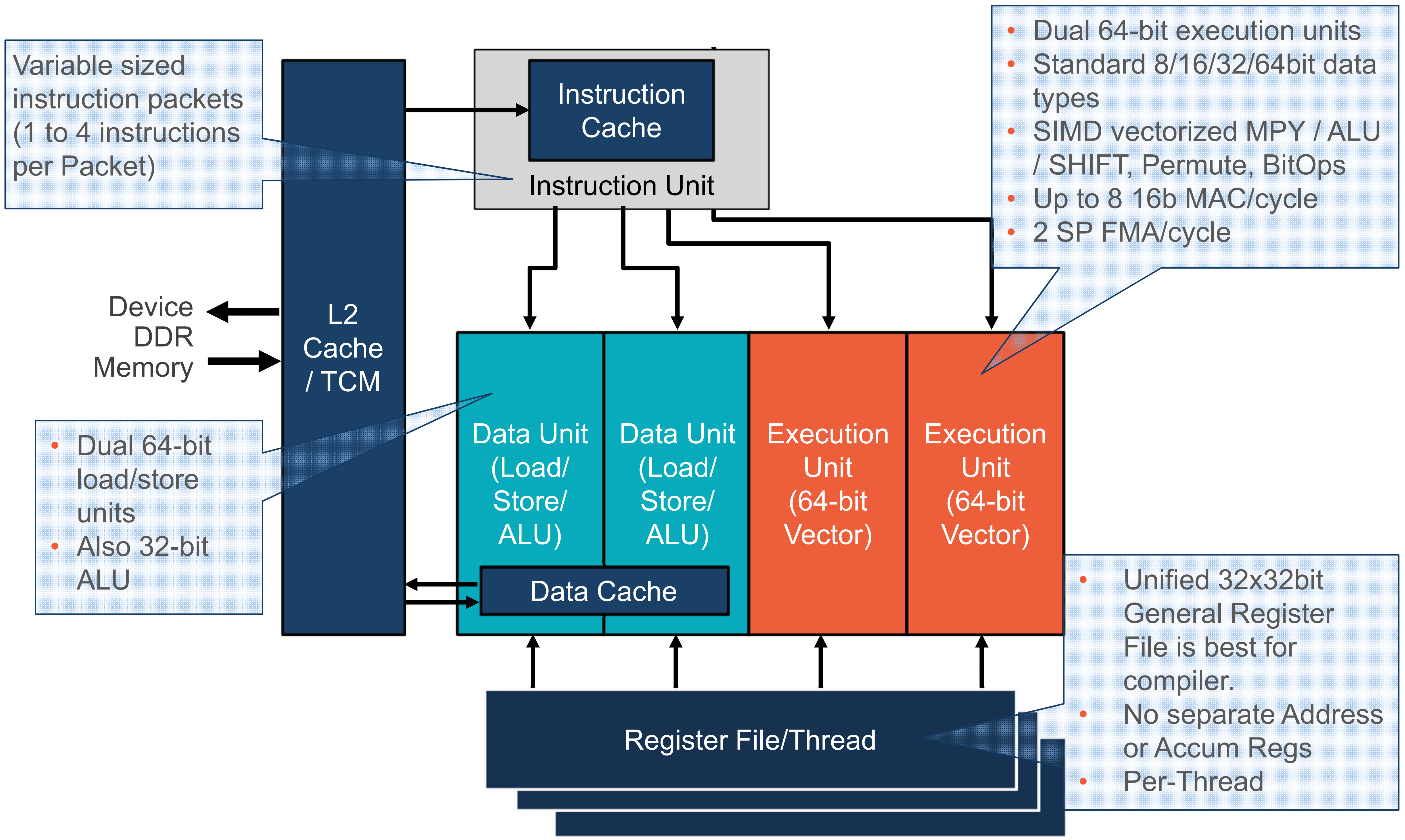
Modern systems use ASICs for…

- 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)

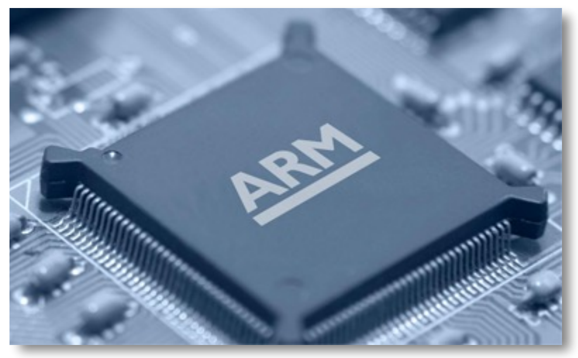
Qualcomm Hexagon DSP

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

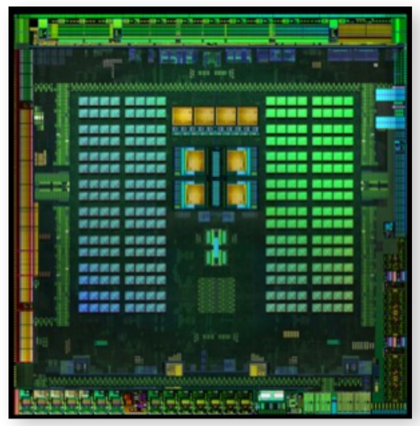

Variable sized instruction packets (1 to 4 instructions per Packet)
Instruction Cache
Instruction Unit
Dual 64-bit execution units
Standard 8/16/32/64bit data types
SIMD vectorized MPY / ALU / SHIFT, Permute, BitOps
Up to 8 16b MAC/cycle
2 SP FMA/cycle
Device DDR Memory
L2 Cache / TCM
Data Unit (Load/ Store/ ALU)
Data Unit (Load/ Store/ ALU)
Execution Unit (64-bit Vector)
Execution Unit (64-bit Vector)
Dual 64-bit load/store units
Also 32-bit ALU
Data Cache
Register File/Thread
Unified 32x32bit General Register File is best for compiler.
No separate Address or Accum Regs
Per-Thread

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 15-769!

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

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

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?