

Lecture 15:

Hardware Specialization + Domain Specific Programming Languages

**Parallel Computing
Stanford CS149, Fall 2022**

Today

- **Part 1: motivation for heterogeneous (specialized) hardware designs**
 - **Good: high efficiency computing**
 - **Bad: challenging to map application software to these machines**

- **Part 2: raising the level of abstraction using domain specific languages**
 - **Obtaining high performance and high productivity**

I want to begin this lecture by reminding you...

In assignment 1 we observed that a well-optimized parallel implementation of a compute-bound application is about 40 times faster on a quad-core CPU than the output of single-threaded C code compiled with gcc -O3.

(In other words, a lot of software makes inefficient use of modern CPUs.)

Today we're going to talk about how inefficient the CPU is, even if you are using it as efficiently as possible.

Heterogeneous processing

Observation: most “real world” applications have complex workload characteristics

They have components that can be widely parallelized.

And components that are difficult to parallelize.

They have components that are amenable to wide SIMD execution.

And components that are not. (divergent control flow)

They have components with predictable data access

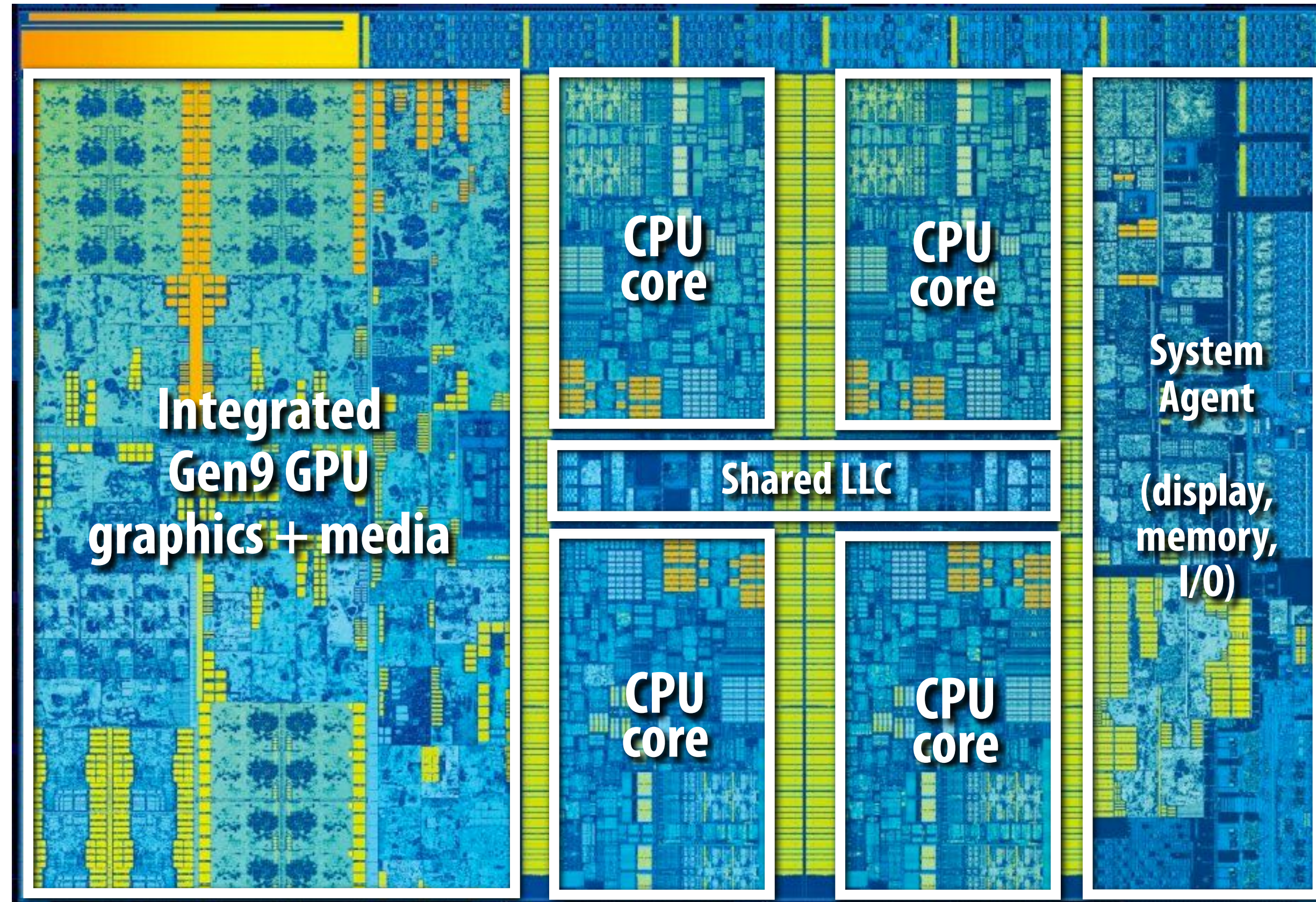
And components with unpredictable access, but those accesses might cache well.

Idea: the most efficient processor is a heterogeneous mixture of resources (“use the most efficient tool for the job”)

Examples of heterogeneity

Example: Intel "Skylake" (2015)

(6th Generation Core i7 architecture)

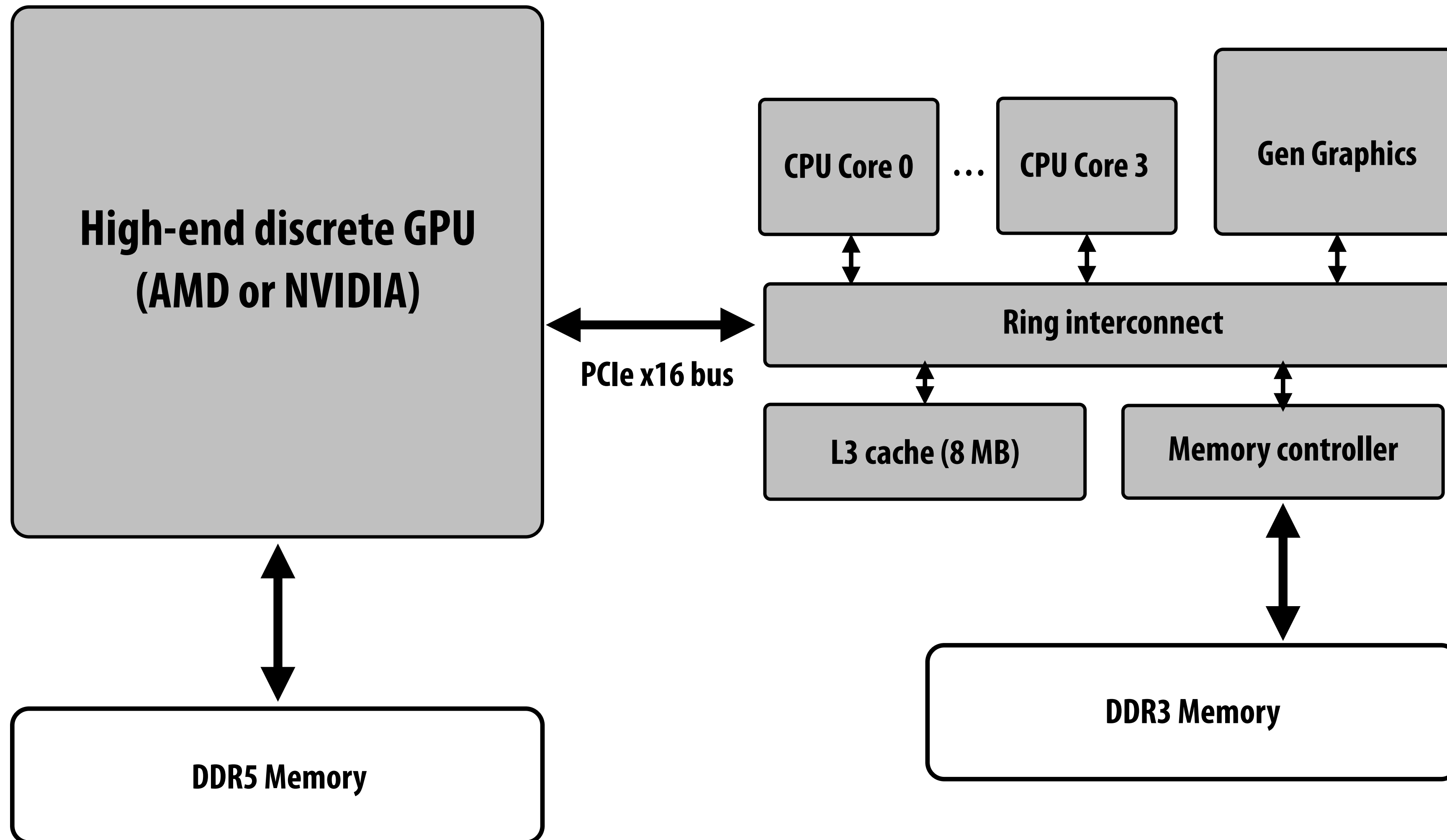


- CPU cores + GPU (which itself has multiple cores)
- CPU cores and graphics cores share same memory system
- Also share L3 cache (LLC="last level cache")
 - Enables, low-latency, high-bandwidth communication between CPU and integrated GPU
- Graphics cores are cache coherent with CPU cores

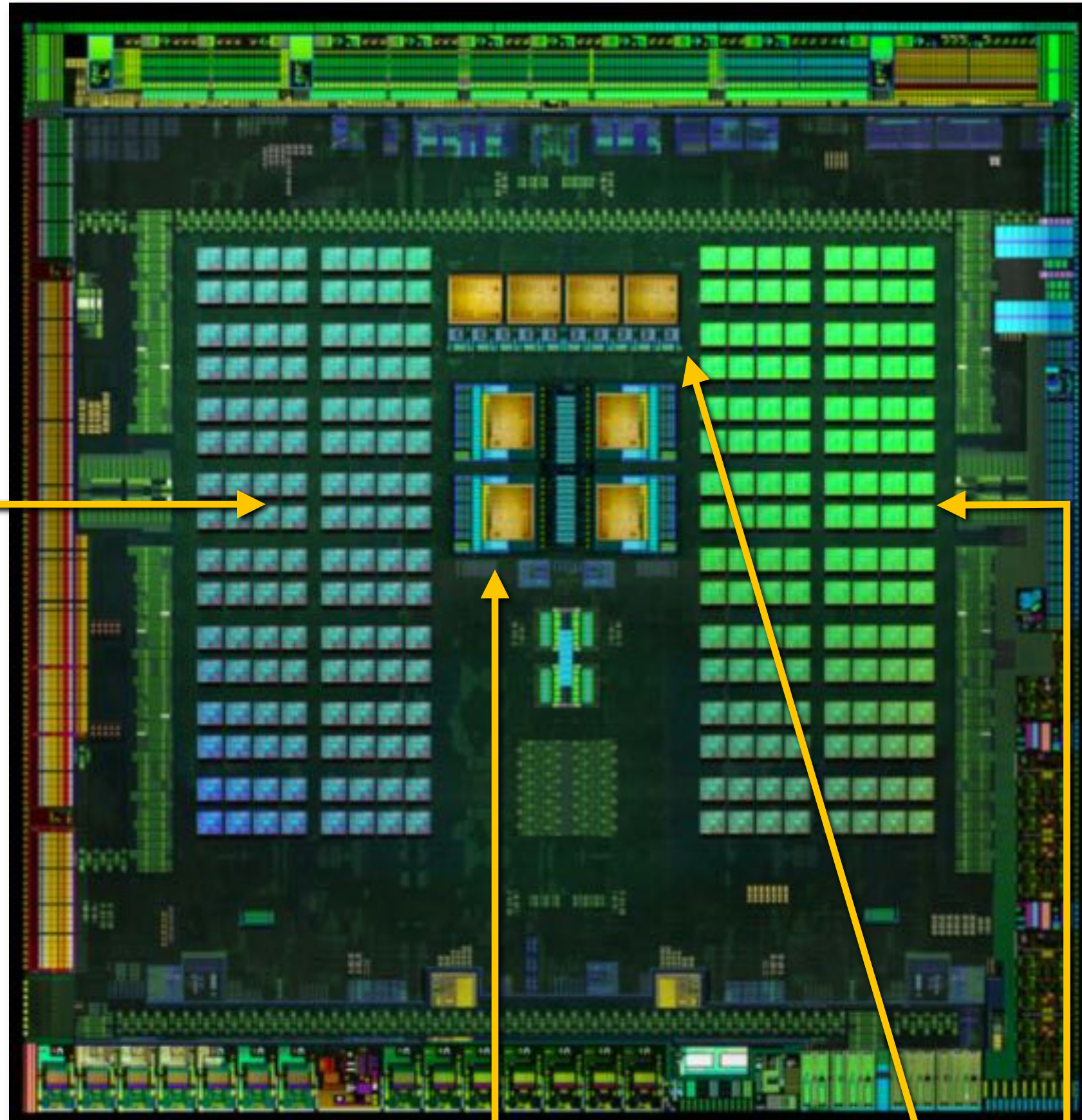
More heterogeneity: add discrete GPU

Keep discrete (power hungry) GPU unless needed for graphics-intensive applications

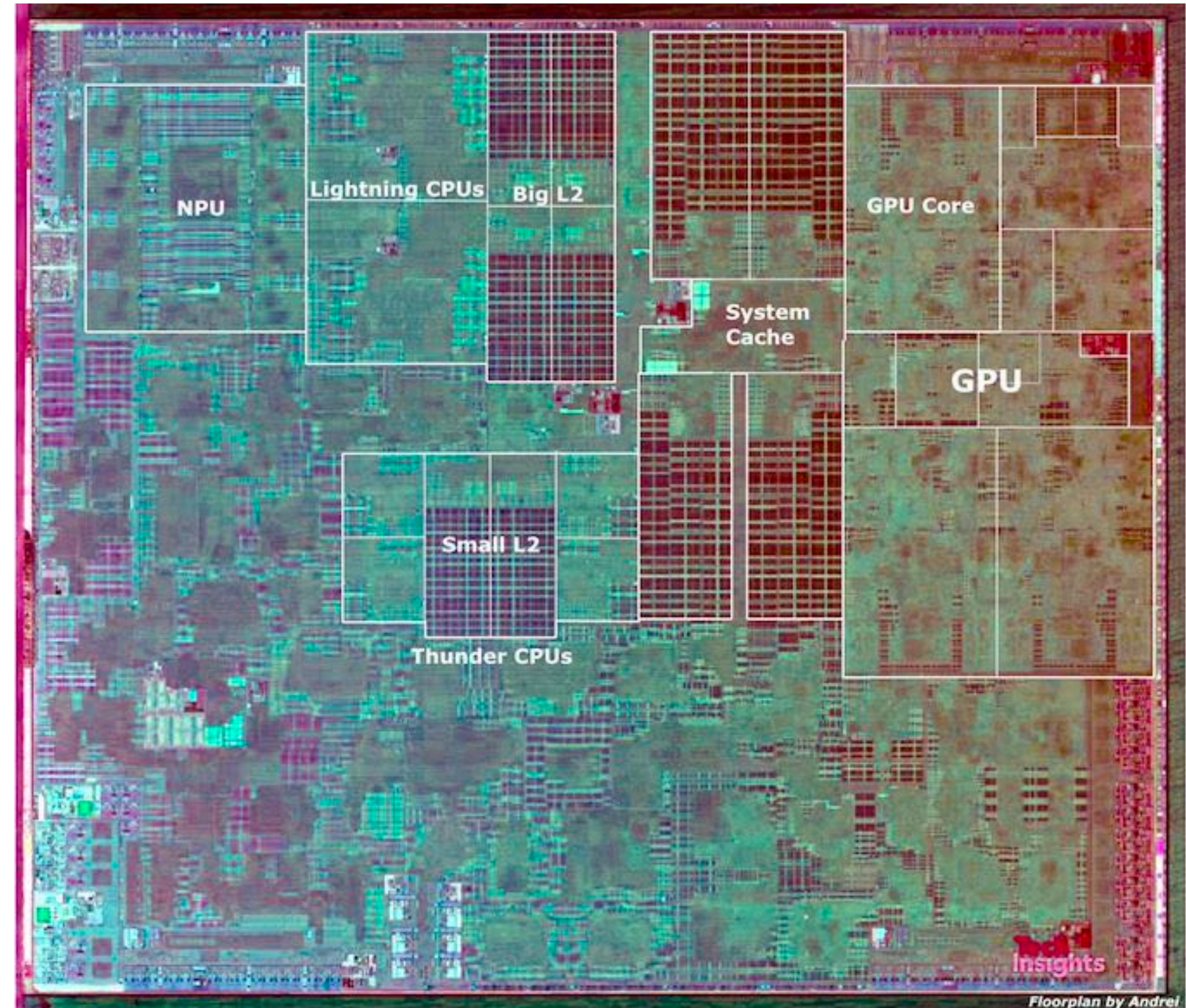
Use integrated, low power graphics for basic graphics/window manager/UI



Mobile heterogeneous processors



NVIDIA Tegra X1
Four ARM Cortex A57 CPU cores for applications
Four low performance (low power) ARM A53 CPU cores
One Maxwell SMM (256 "CUDA" cores)



Apple A13 Bionic Multi-core CPU (heterogeneous cores)
Multi-core GPU
Neural accelerator
Sensor processing accelerator
Video compression/decompression HW
Etc...

GPU-accelerated supercomputing



**Frontier (at Oak Ridge National Lab)
(world's #1 in Fall 2022)**

9472 x 64 core AMD CPUs (606,208 CPU cores)

37,888 Radeon GPUs

21 Megawatts

Heterogeneous architectures for supercomputing

Source: Top500.org November 2022 rankings

Rank	System	Cores	Rmax (PFlop/s)	Rpeak (PFlop/s)	Power (kW)
1	Frontier - HPE Cray EX235a, AMD Optimized 3rd Generation EPYC 64C 2GHz, AMD Instinct MI250X, Slingshot-11, HPE DOE/SC/Oak Ridge National Laboratory United States	8,730,112	1,102.00	1,685.65	21,100
2	Supercomputer Fugaku - Supercomputer Fugaku, A64FX 48C 2.2GHz, Tofu interconnect D, Fujitsu RIKEN Center for Computational Science Japan	7,630,848	442.01	537.21	29,899
3	LUMI - HPE Cray EX235a, AMD Optimized 3rd Generation EPYC 64C 2GHz, AMD Instinct MI250X, Slingshot-11, HPE EuroHPC/CSC Finland	2,220,288	309.10	428.70	6,016
4	Leonardo - BullSequana XH2000, Xeon Platinum 8358 32C 2.6GHz, NVIDIA A100 SXM4 64 GB, Quad-rail NVIDIA HDR100 Infiniband, Atos EuroHPC/CINECA Italy	1,463,616	174.70	255.75	5,610
5	Summit - IBM Power System AC922, IBM POWER9 22C 3.07GHz, NVIDIA Volta GV100, Dual-rail Mellanox EDR Infiniband, IBM DOE/SC/Oak Ridge National Laboratory United States	2,414,592	148.60	200.79	10,096
6	Sierra - IBM Power System AC922, IBM POWER9 22C 3.1GHz, NVIDIA Volta GV100, Dual-rail Mellanox EDR Infiniband, IBM / NVIDIA / Mellanox DOE/NNSA/LLNL United States	1,572,480	94.64	125.71	7,438
7	Sunway TaihuLight - Sunway MPP, Sunway SW26010 260C 1.45GHz, Sunway, NRCPC National Supercomputing Center in Wuxi China	10,649,600	93.01	125.44	15,371

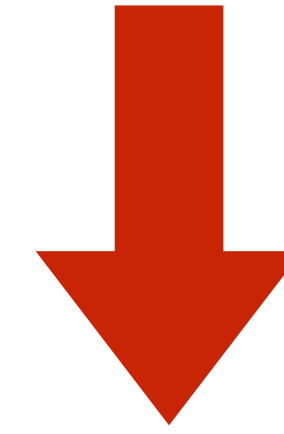
1.1 exaflops (observed running LINPACK)
21.1 MWatt
(52.2 GFLOPS/W)

Energy-Constrained Computing

Performance and power

$$\text{POWER} = \frac{\text{Performance}}{\text{Energy Efficiency}} = \frac{\text{Ops}}{\text{Sec}} \times \frac{\text{Joules}}{\text{Op}}$$

Fixed!



Specialization (fixed function) → better energy efficiency

What is the magnitude of improvement from specialization?

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 is not floating-point math**

Why is a “general-purpose processor” so inefficient?

Wait... this entire class we've been talking about making efficient use out of multi-core CPUs and GPUs... and now you're telling me these platforms are “inefficient”?

Consider the complexity of executing an instruction on a modern processor...

Read instruction ———— | Address translation, communicate with icache, access icache, etc.

Decode instruction ———— | Translate op to uops, access uop cache, etc.

Check for dependencies/pipeline hazards

Identify available execution resource

Use decoded operands to control register file SRAM (retrieve data)

Move data from register file to selected execution resource

Perform arithmetic operation

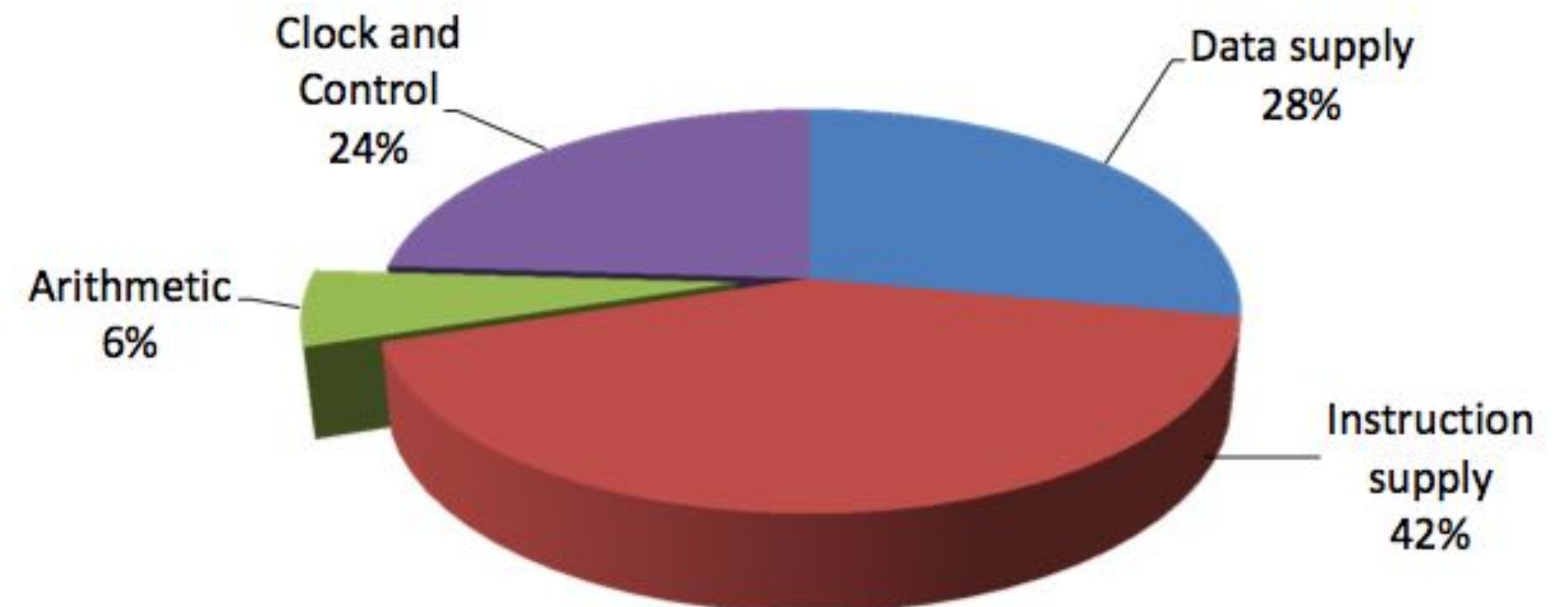
Move data from execution resource to register file

Use decoded operands to control write to register file SRAM

Review question:

How does SIMD execution reduce overhead of certain types of computations?

What properties must these computations have?

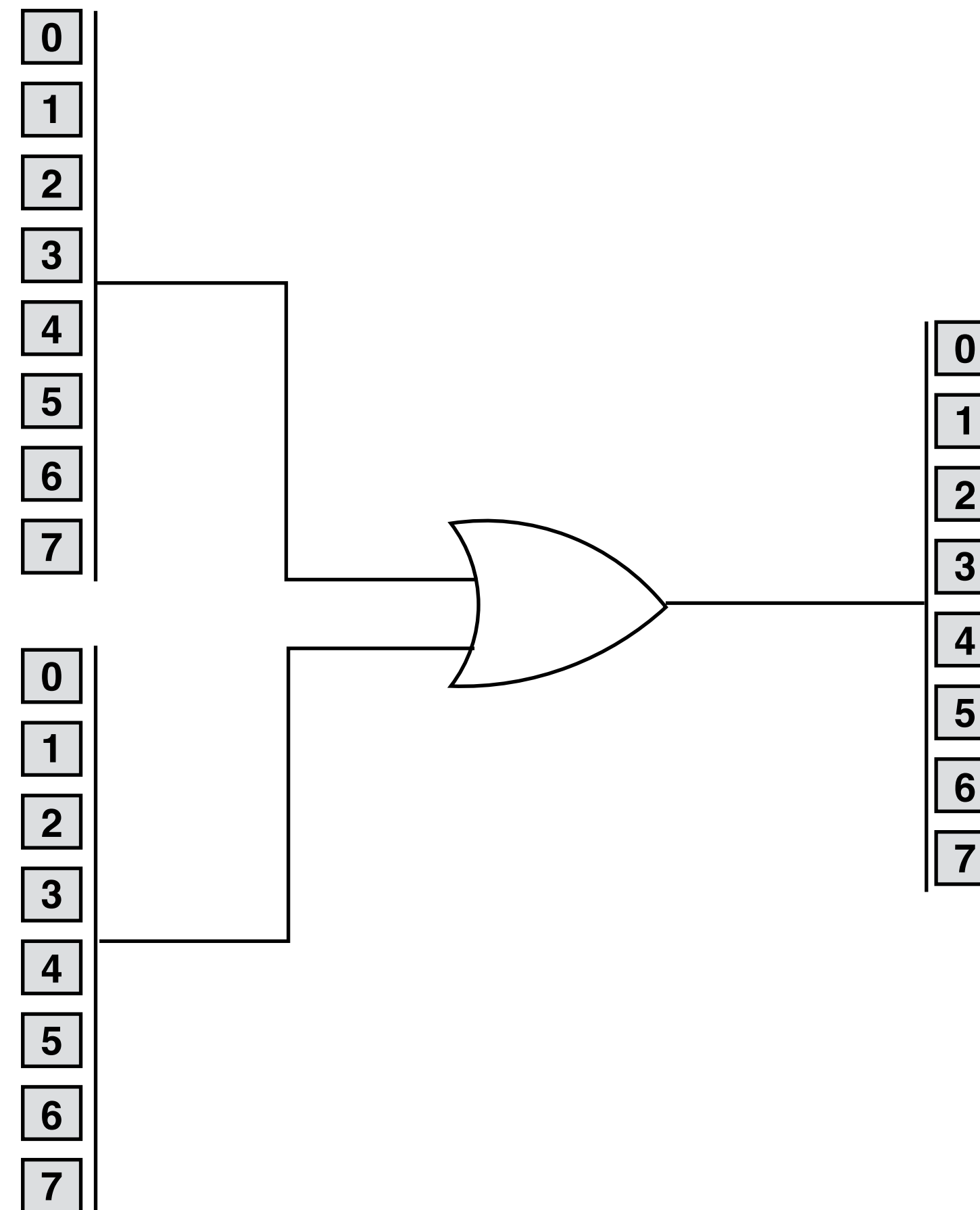


Efficient Embedded Computing [Dally et al. 08]

[Figure credit Eric Chung]

Contrast that complexity to the circuit required to actually perform the operation

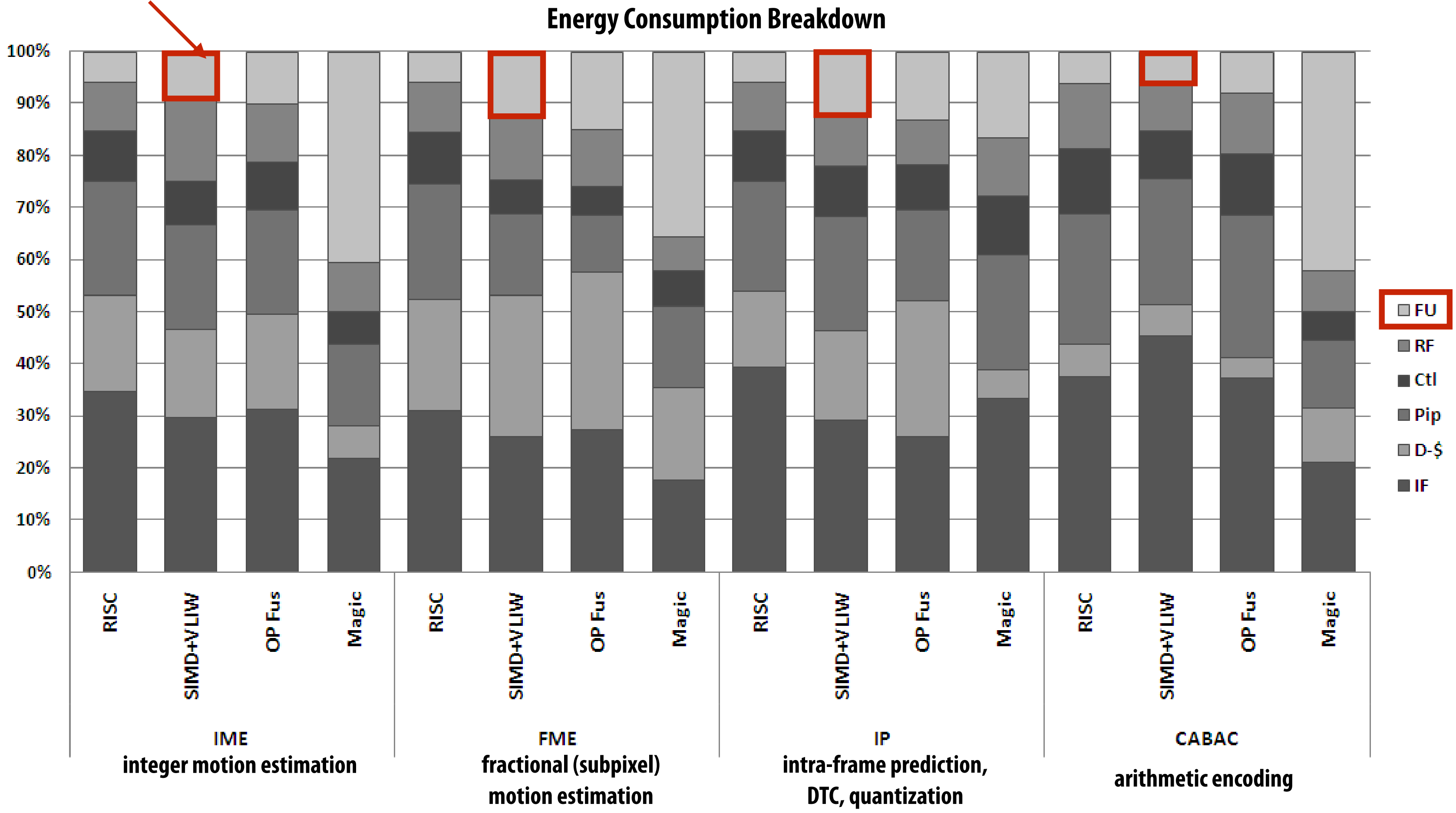
Example: 8-bit logical OR



H.264 video encoding: fraction of energy consumed by functional units is small (even when using SIMD)

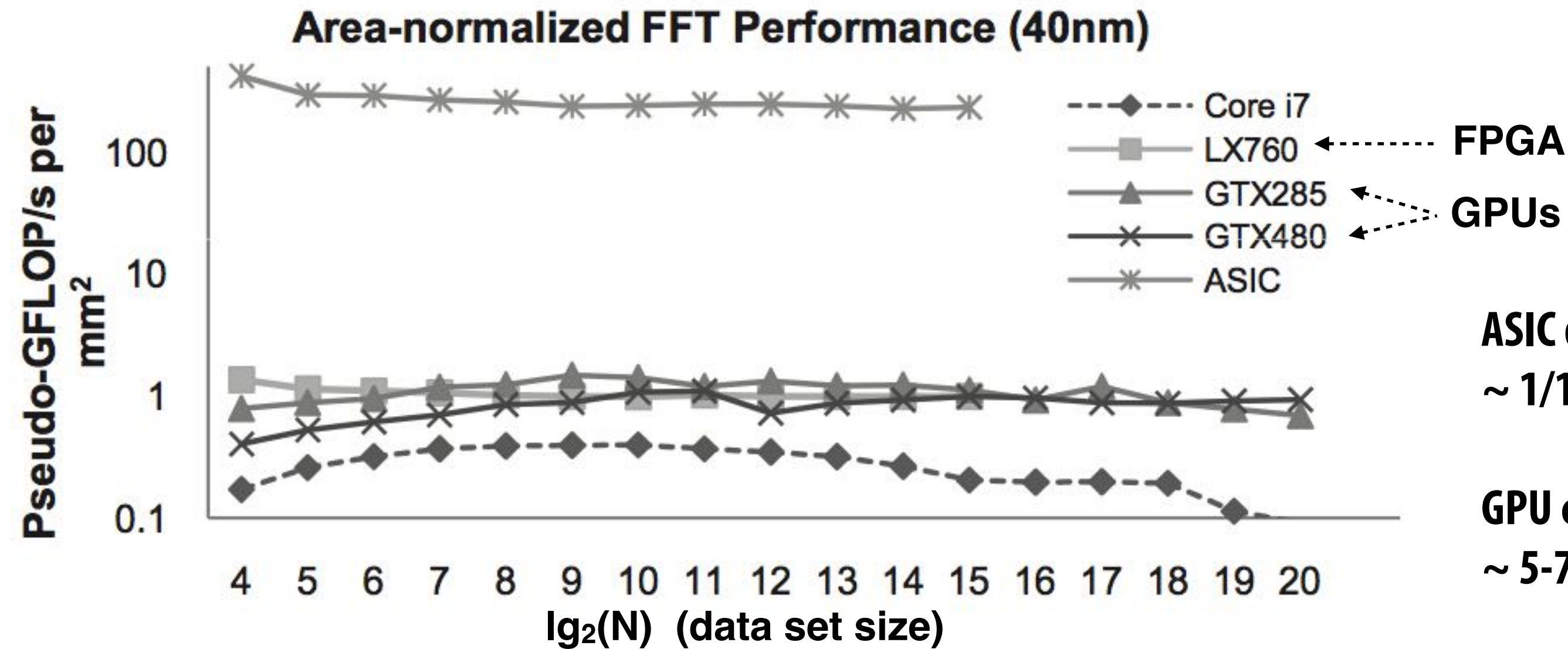
Even after encoding implemented with SIMD instruction

[Hameed et al. ISCA 2010]



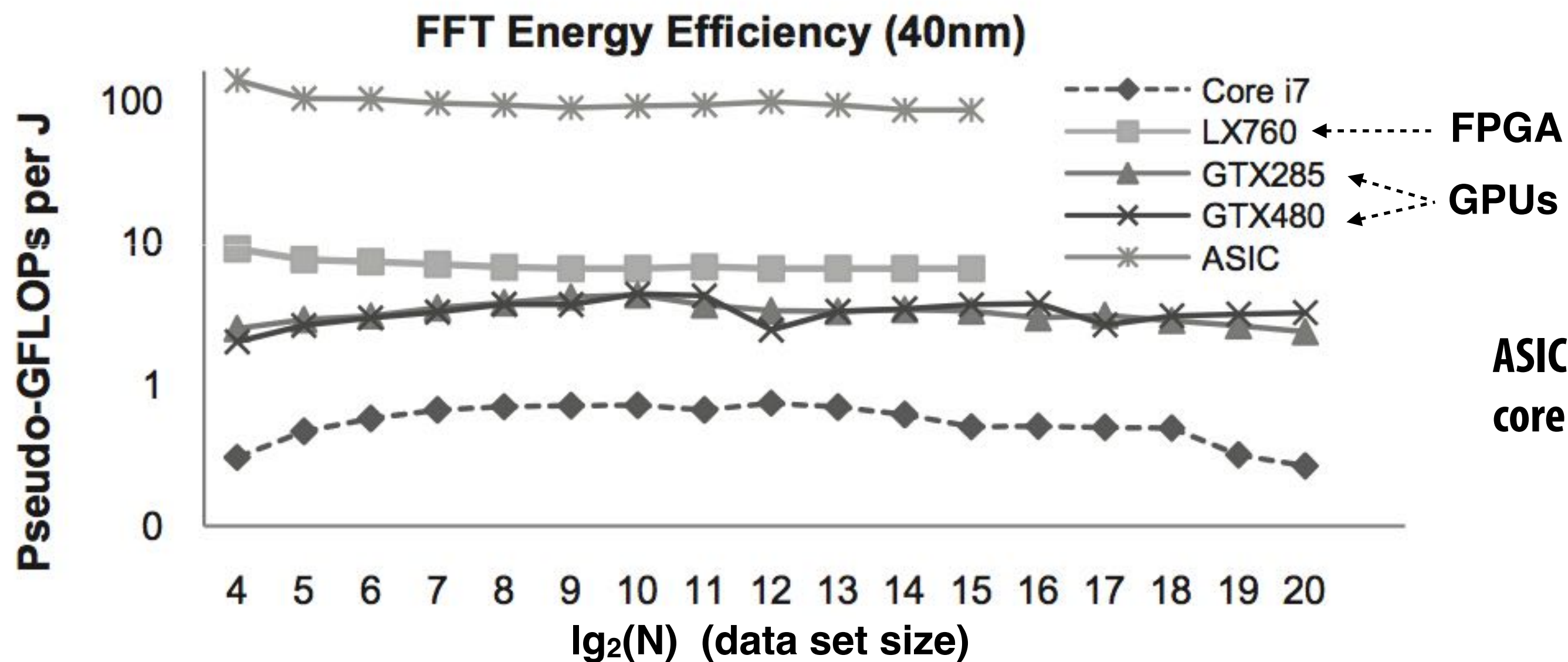
FU = functional units
RF = register fetch
Ctrl = misc pipeline control
Pip = pipeline registers (interstage)
D-\$ = data cache
IF = instruction fetch + instruction cache

Fast Fourier transform (FFT): throughput and energy benefits of specialization



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

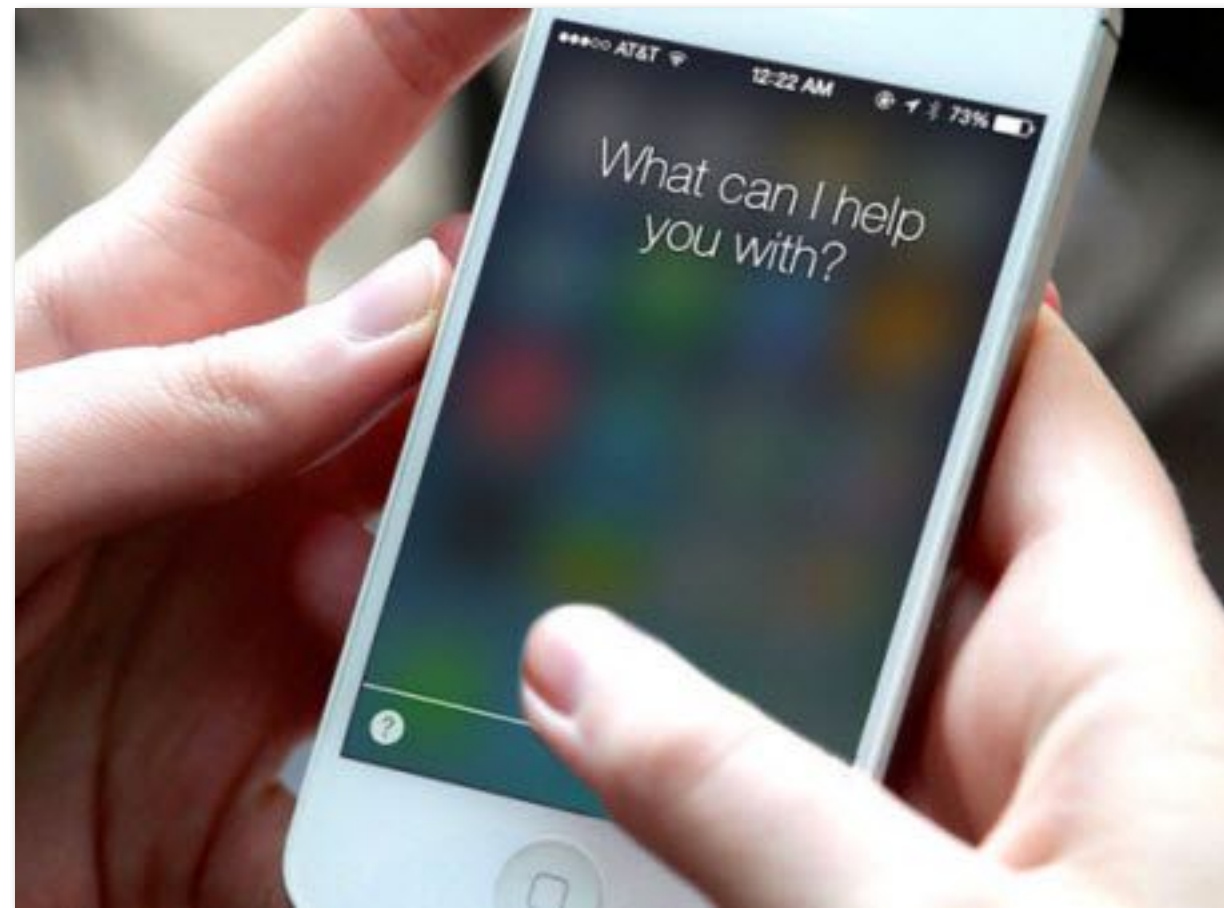
GPU cores:
 $\sim 5-7$ times more area efficient than CPU cores.



ASIC delivers same performance as one CPU core using only $\sim 1/100$ th the power

Mobile: benefits of increasing efficiency

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



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

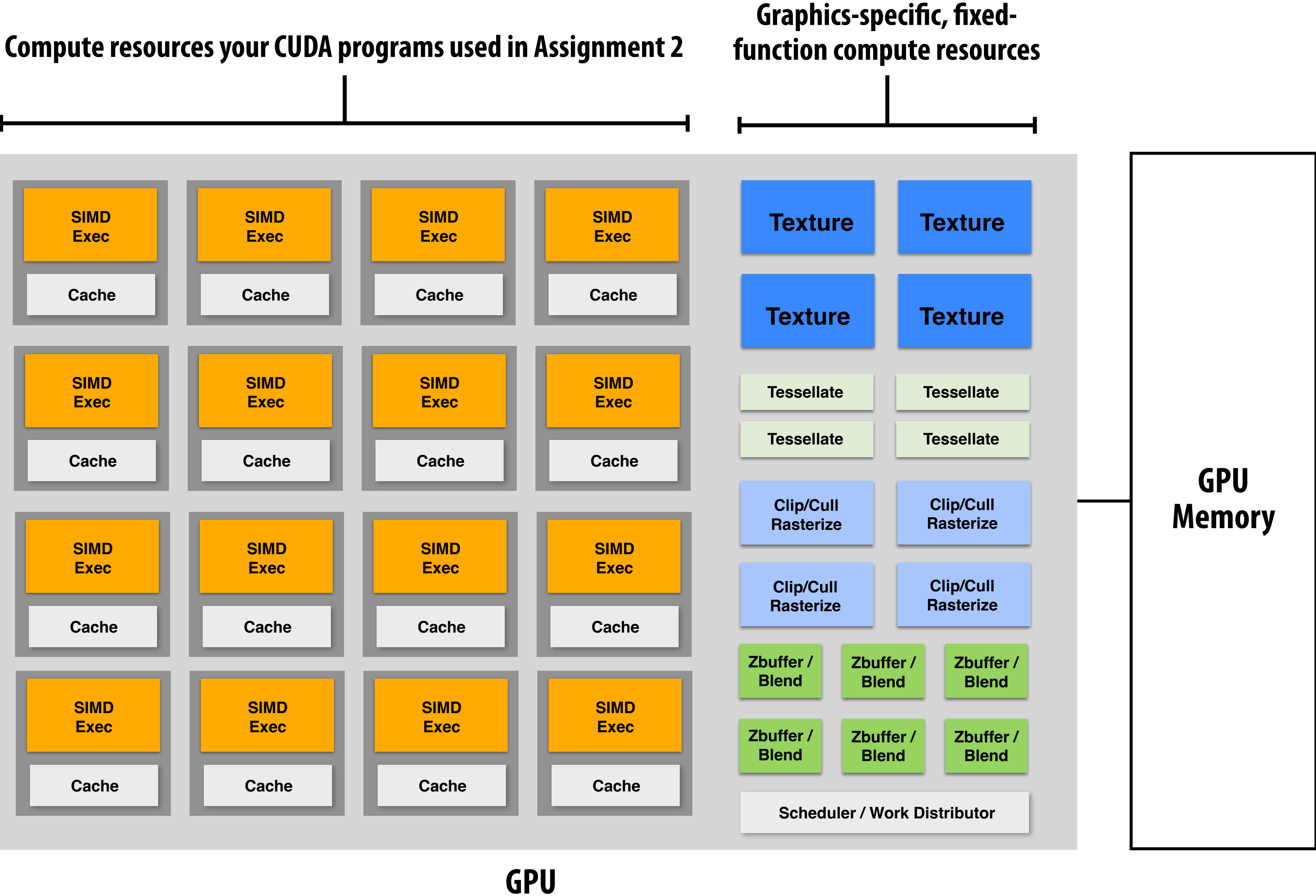


Amazon Echo / Google Home
Always listening



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

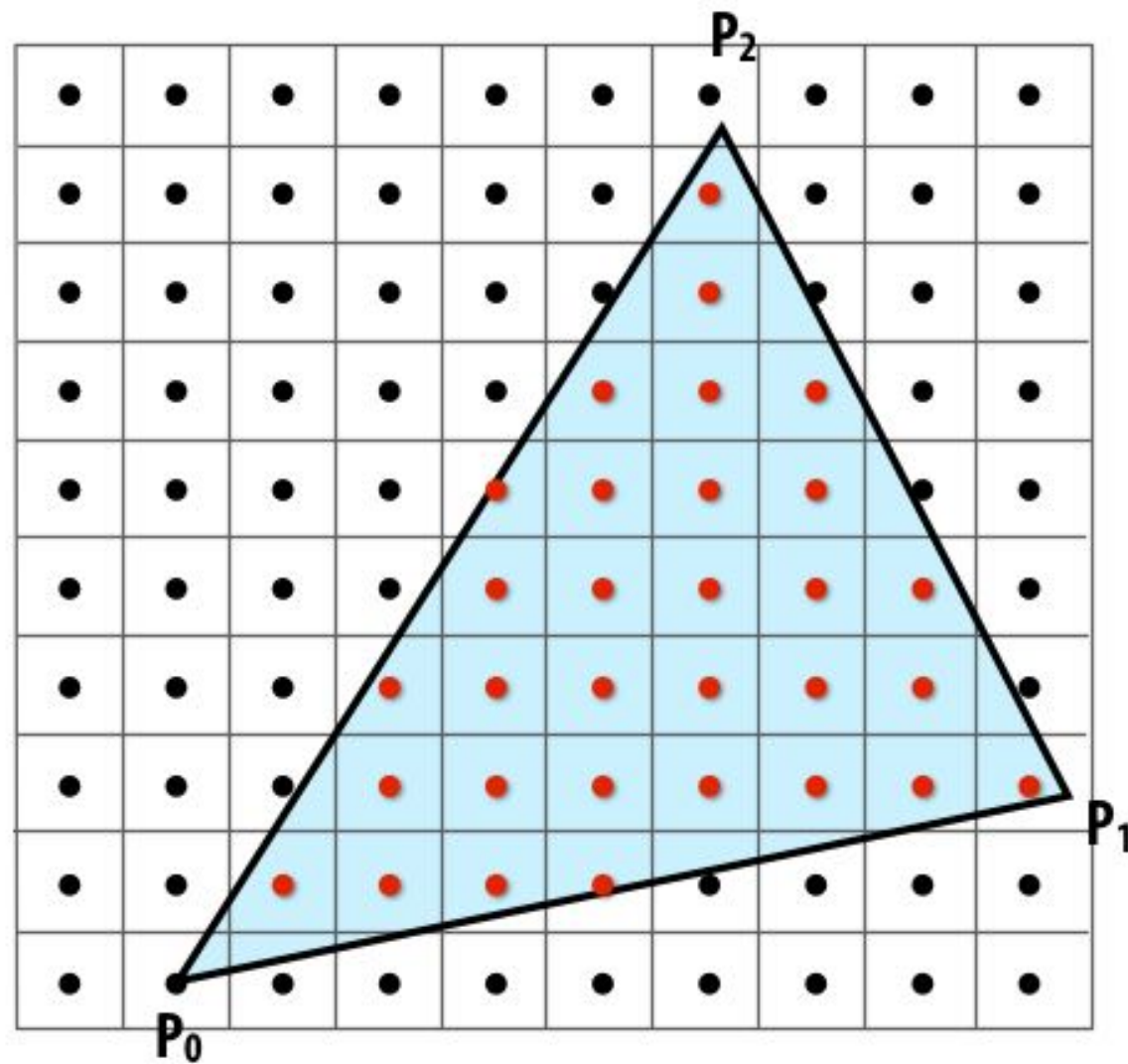
GPU's are themselves heterogeneous multi-core processors



Example graphics tasks performed in fixed-function HW

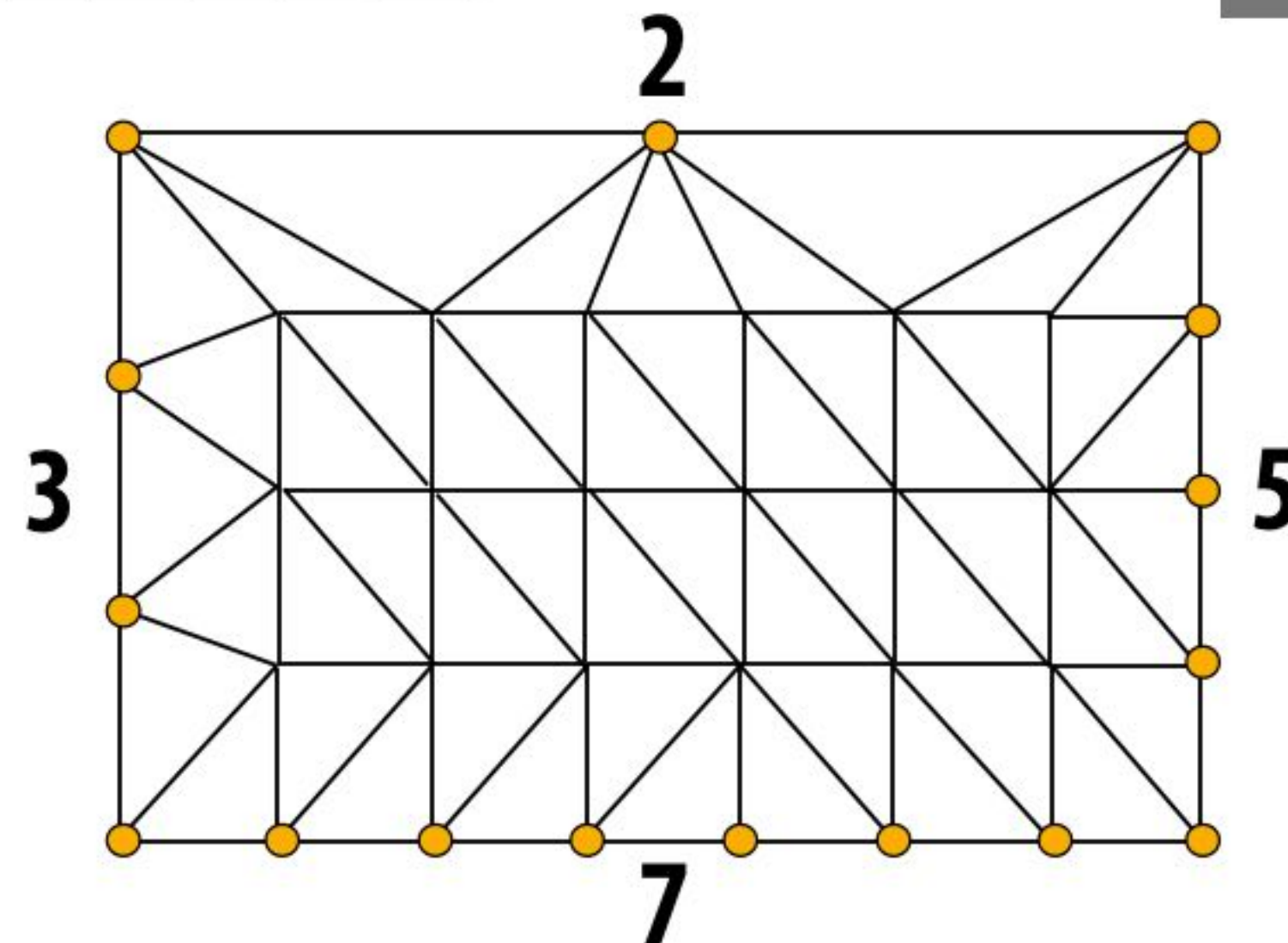
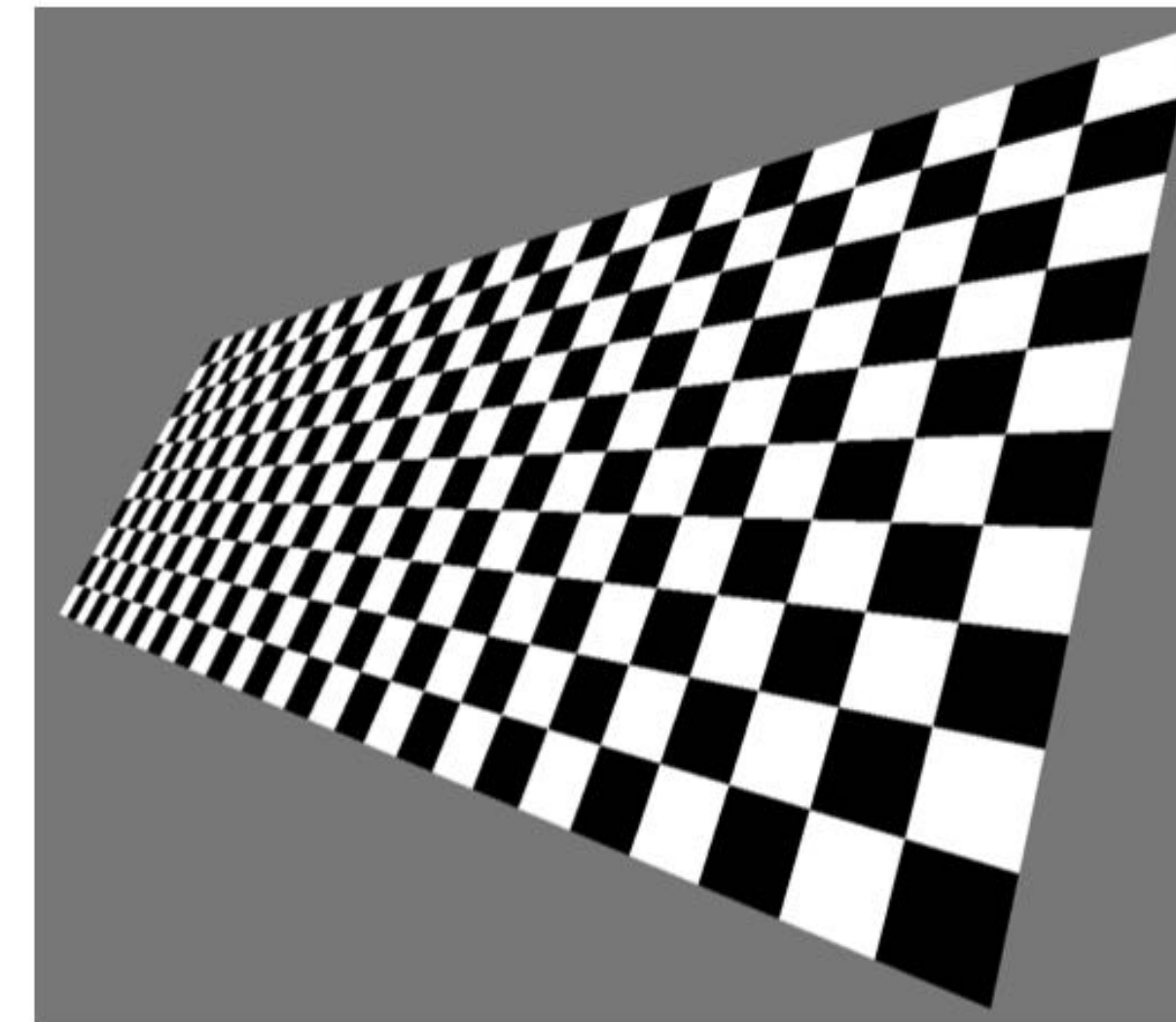
Rasterization:

Determining what pixels a triangle overlaps



Texture mapping:

Warping/filtering images to apply detail to surfaces



Geometric tessellation:
computing fine-scale geometry
from coarse geometry

Digital signal processors (DSPs)

Programmable processors, but simpler instruction stream control paths

Complex instructions (e.g., SIMD/VLIW): perform many operations per instruction (amortize cost of control)

Example: Qualcomm Hexagon DSP

Used for modem, audio, and (increasingly) image processing on Qualcomm Snapdragon SoC processors

VLIW: "very-long instruction word"

Single instruction specifies multiple different operations to do at once (contrast to SIMD)

Below: innermost loop of FFT

Hexagon DSP performs 29 "RISC" ops per cycle

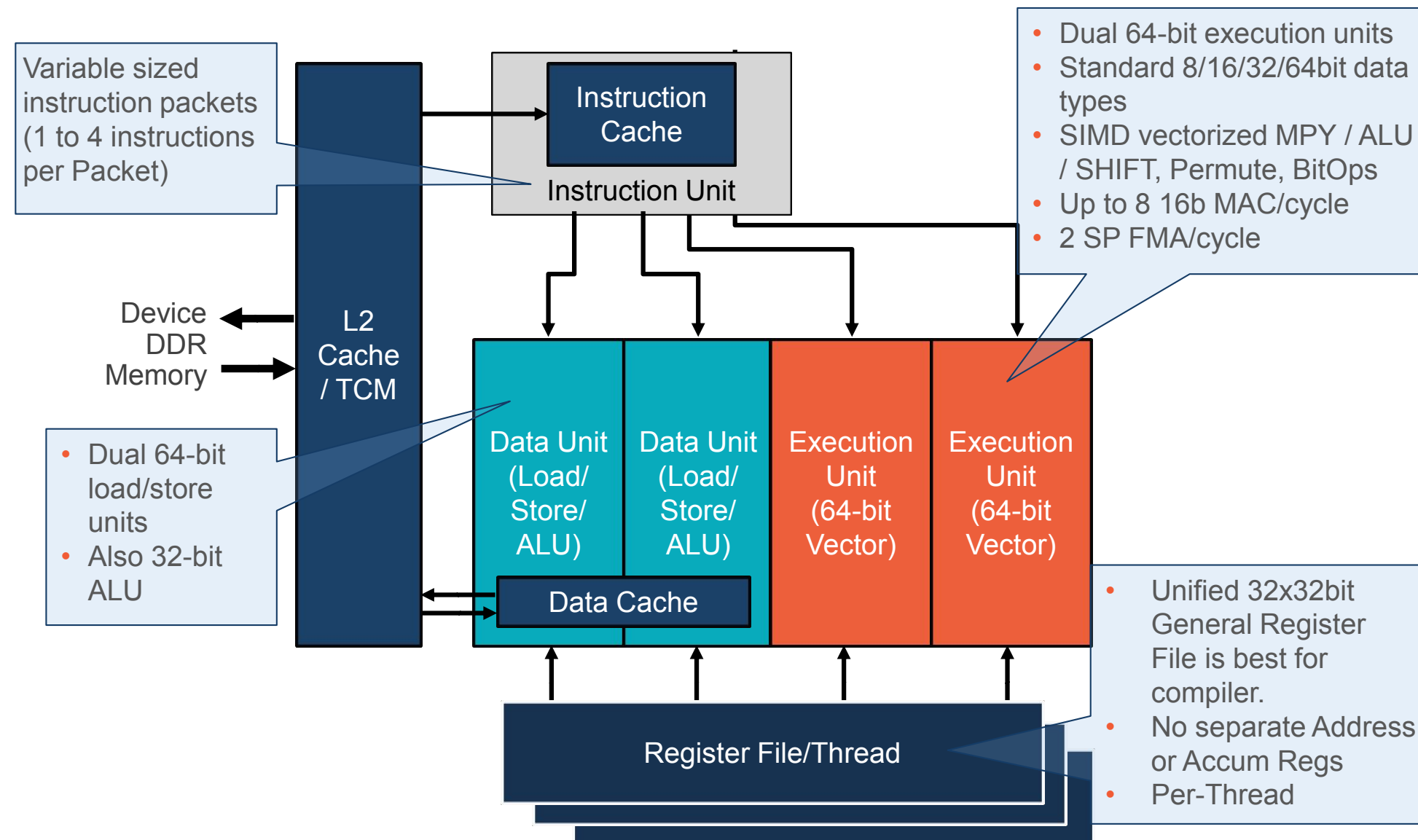
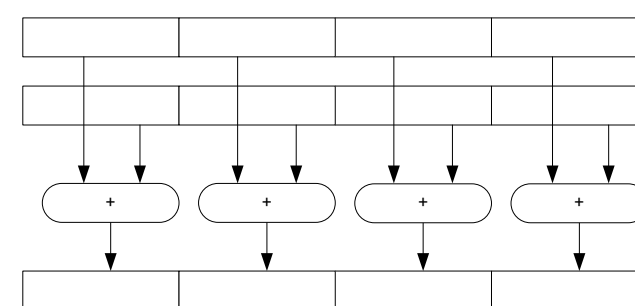
64-bit Load and
64-bit Store with
post-update
addressing

```
{ R17:16 = MEMD(R0++M1)
  MEMD(R6++M1) = R25:24
  R20 = CMPY(R20, R8):<<1:rnd:sat
  R11:10 = VADDH(R11:10, R13:12)
}:endloop0
```

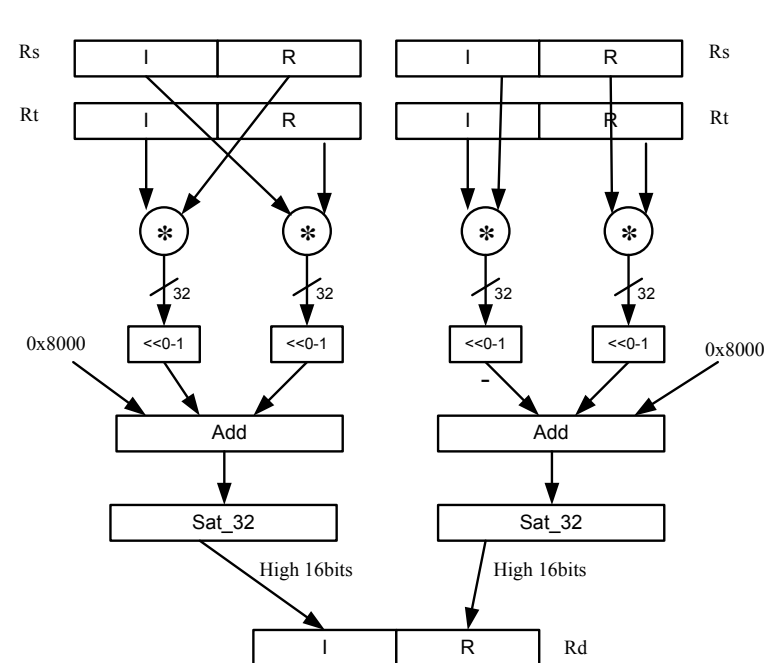
Zero-overhead loops

- Dec count
- Compare
- Jump top

Vector 4x16-bit Add



Complex multiply with
round and saturation

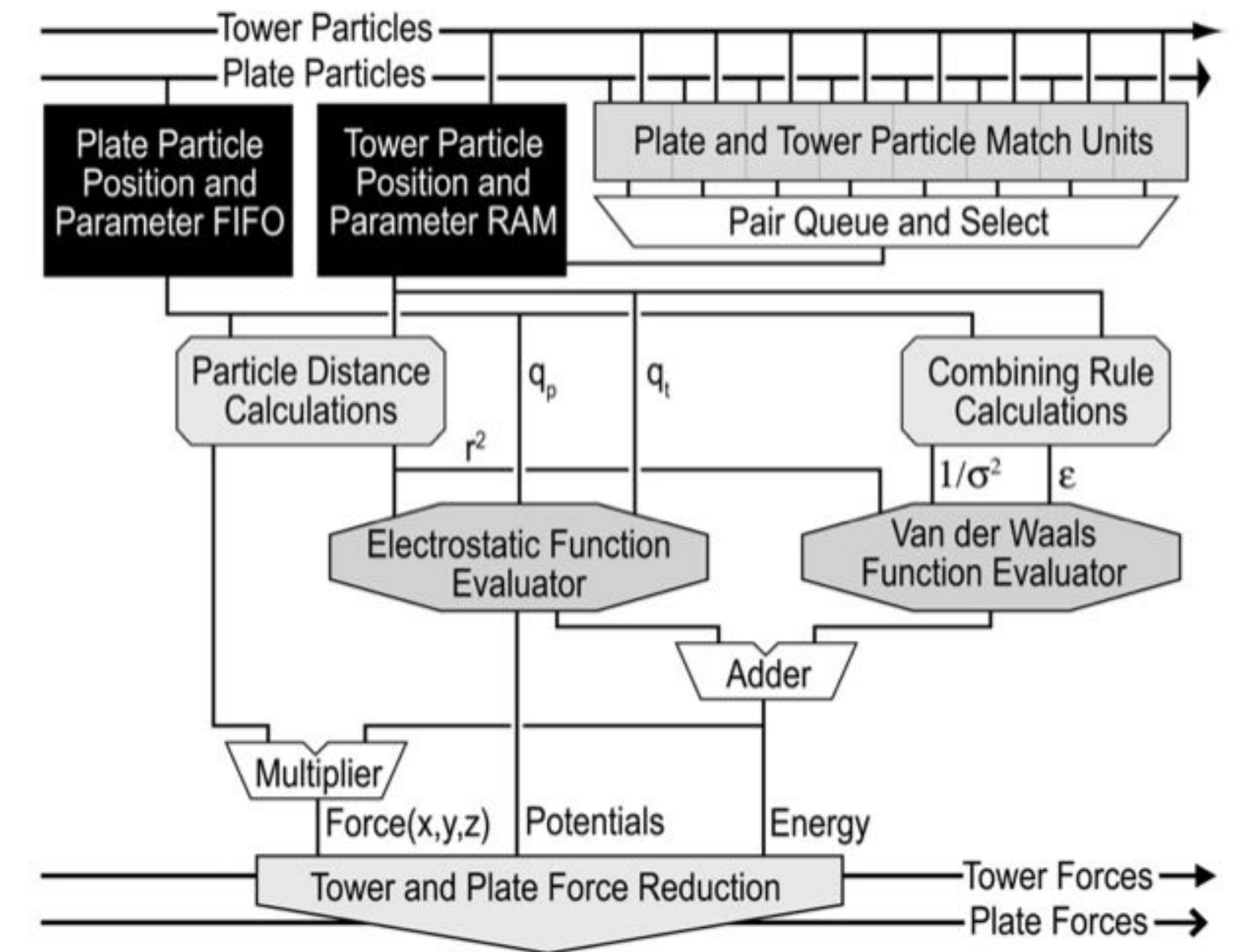
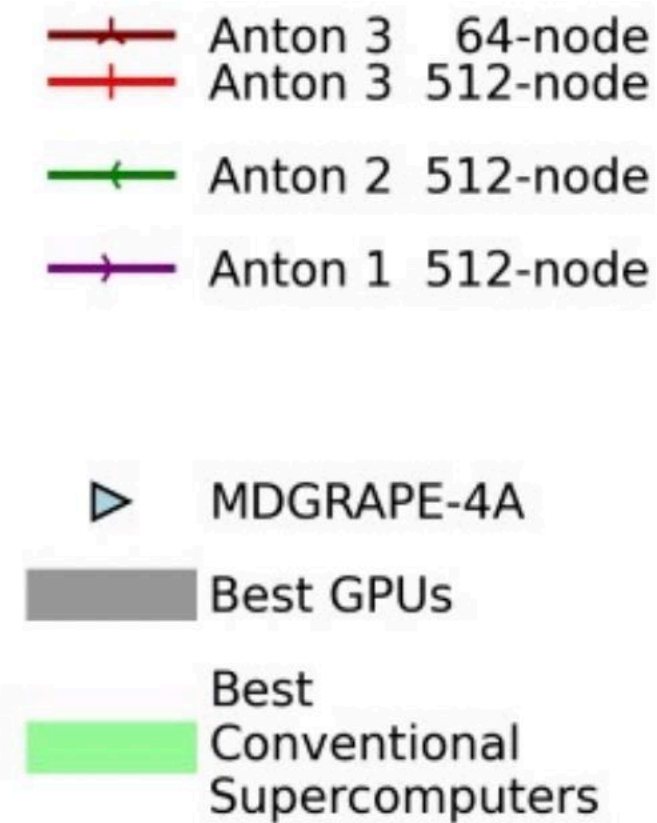
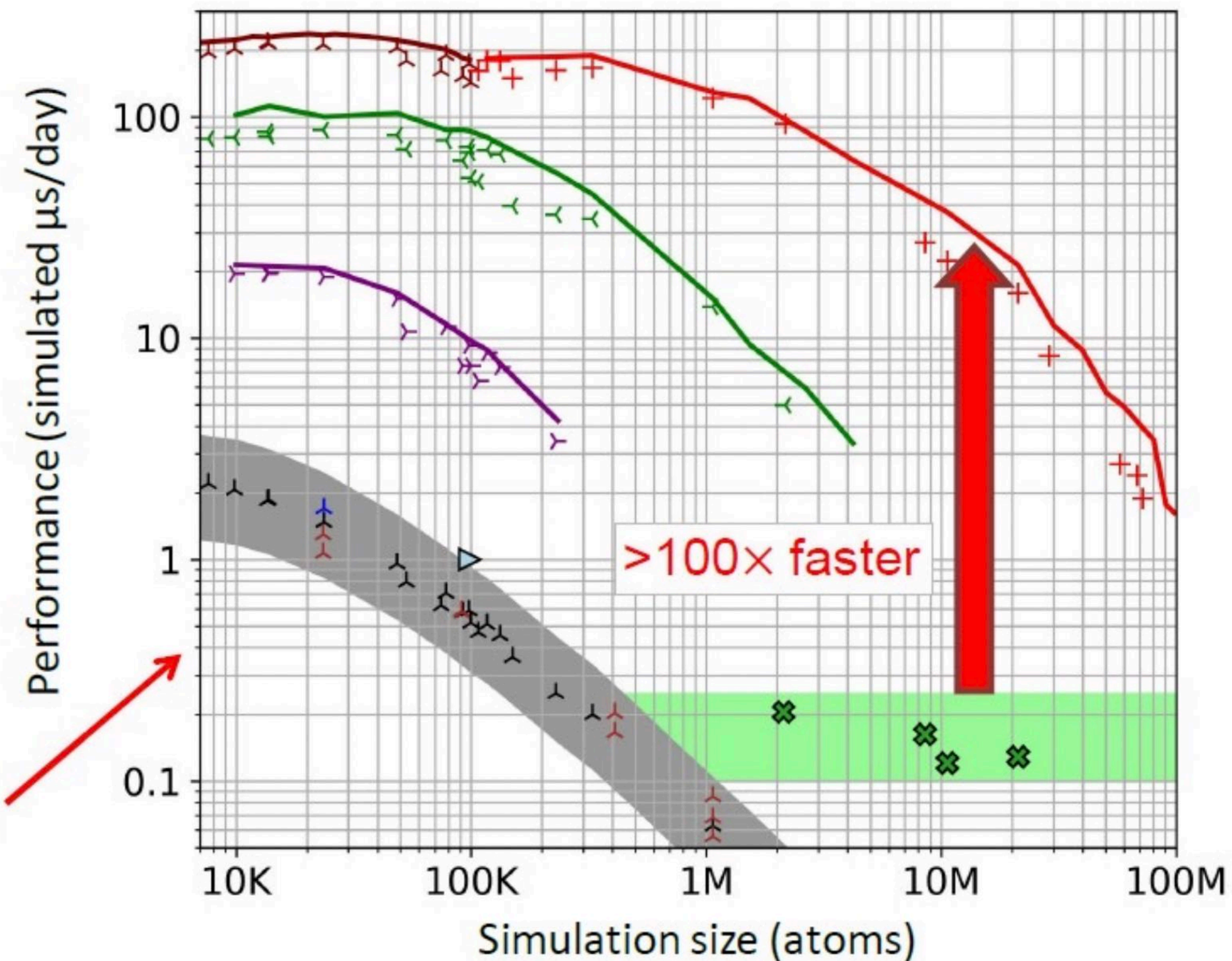


Hexagon DSP is in Google Pixel phone

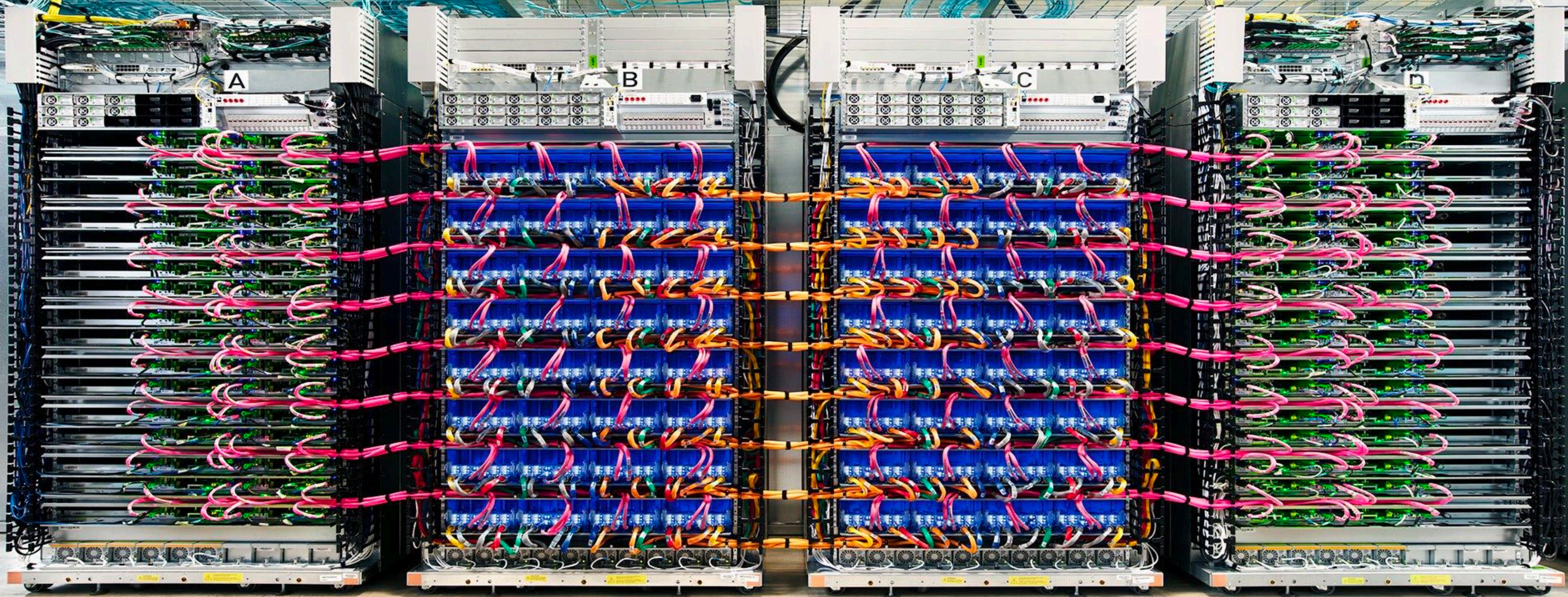
Anton supercomputer for molecular dynamics

[Developed by DE Shaw Research]

- Simulates time evolution of proteins
- ASIC for computing particle-particle interactions (512 of them in machine)
- Throughput-oriented subsystem for efficient fast-fourier transforms
- Custom, low-latency communication network designed for communication patterns of N-body simulations



Specialized processors for evaluating deep networks



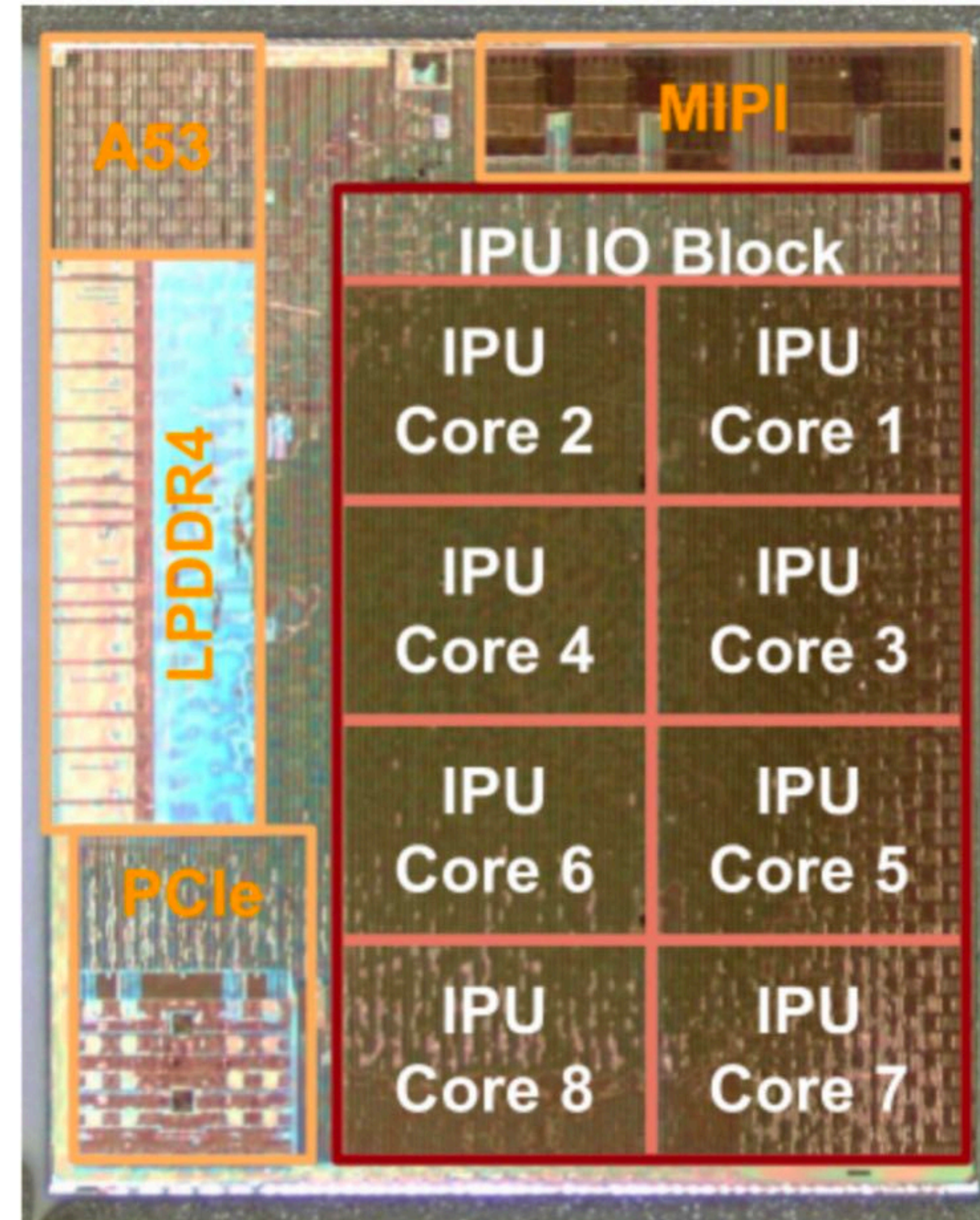
Google TPU pods

Image Credit: TechInsights Inc.

Example: Google's Pixel Visual Core

Programmable "image processing unit" (IPU)

- Each core = 16x16 grid of 16 bit multiply-add ALUs
- ~10-20x more efficient than GPU at image processing tasks (Google's claims at HotChips '18)



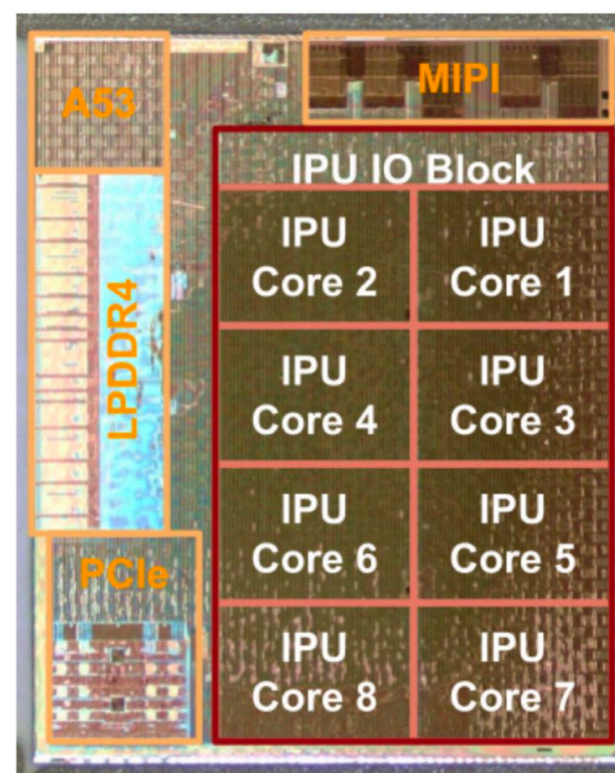
Let's crack open a smartphone

Google Pixel 2 Phone:

Qualcomm Snapdragon 835 SoC + Google Visual Pixel Core

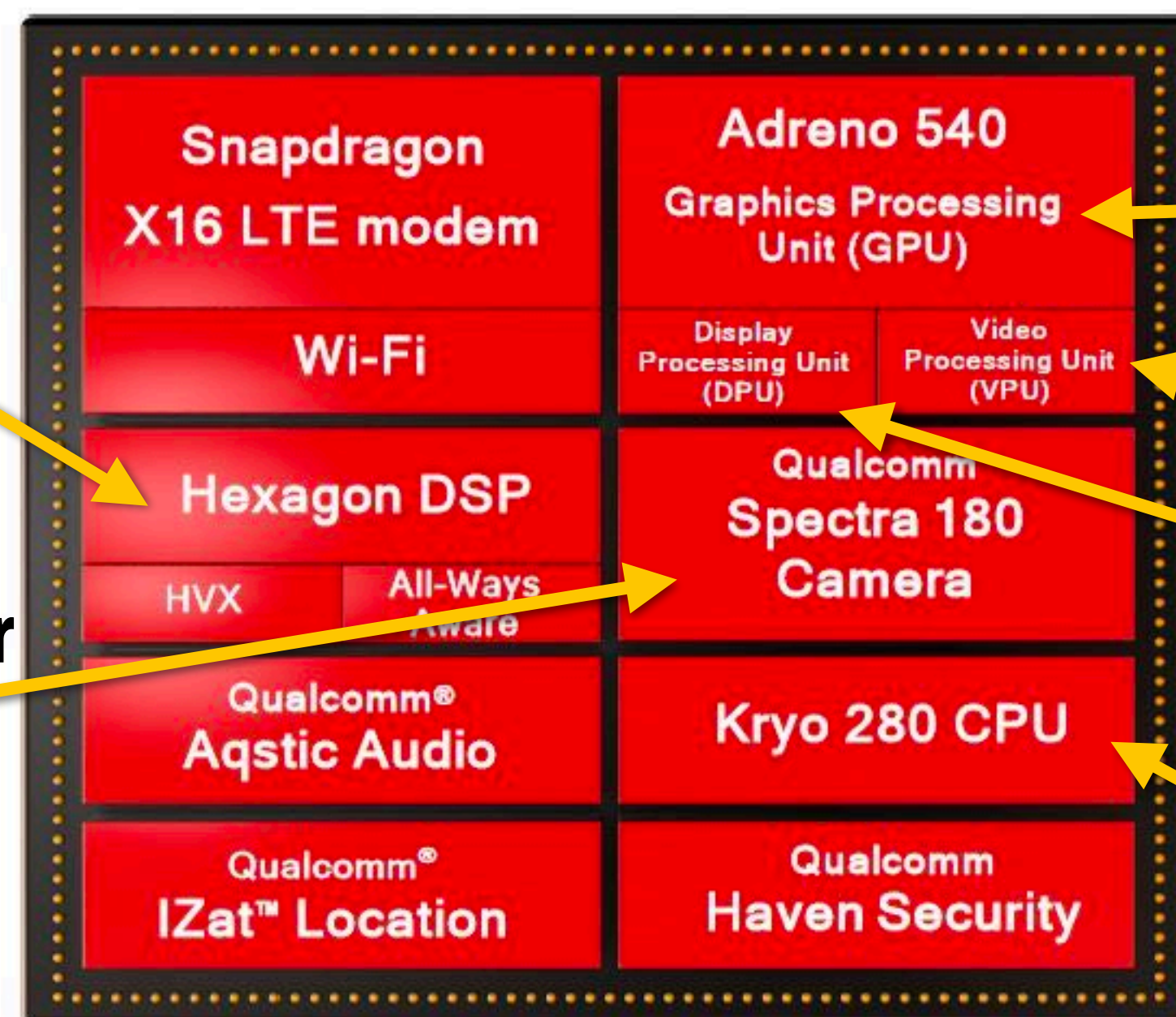


Visual Pixel Core
Programmable image processor and DNN accelerator



“Hexagon”
Programmable DSP
data-parallel multi-media processing

Image Signal Processor
ASIC for processing camera sensor pixels



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

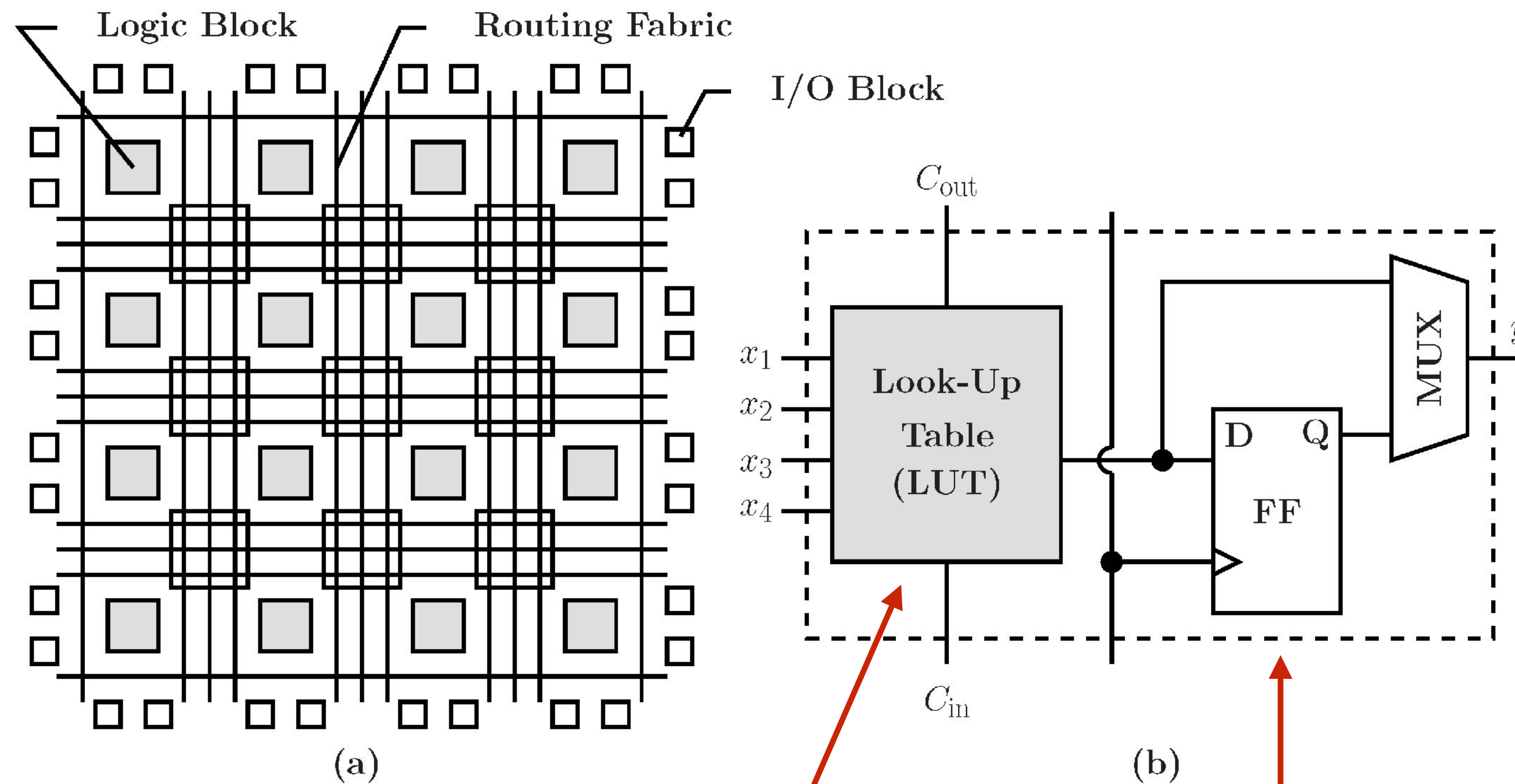
Video encode/decode ASIC

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

Multi-core ARM CPU
4 “big cores” + 4 “little cores”

FPGAs (Field Programmable Gate Arrays)

- Middle ground between an ASIC and a processor
- FPGA chip provides array of logic blocks, connected by interconnect
- Programmer-defined logic implemented directly by FPGA

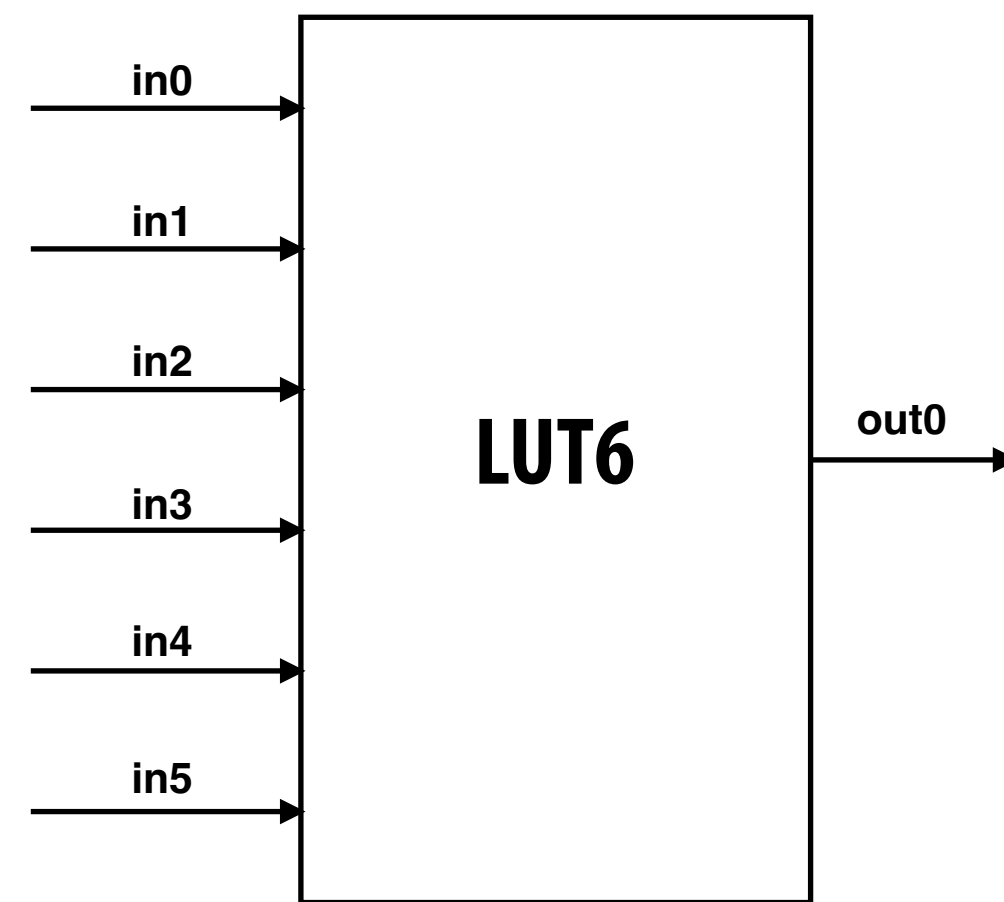


Programmable lookup table (LUT)

Flip flop (a register)

Specifying combinatorial logic as a LUT

- Example: 6-input, 1 output LUT in Xilinx Virtex-7 FPGAs
 - Think of a LUT6 as a 64 element table



Example:
6-input AND

In	Out
0	0
1	0
2	0
3	0
⋮	⋮
63	1

40-input AND constructed by chaining outputs of eight LUT6's (delay = 3)

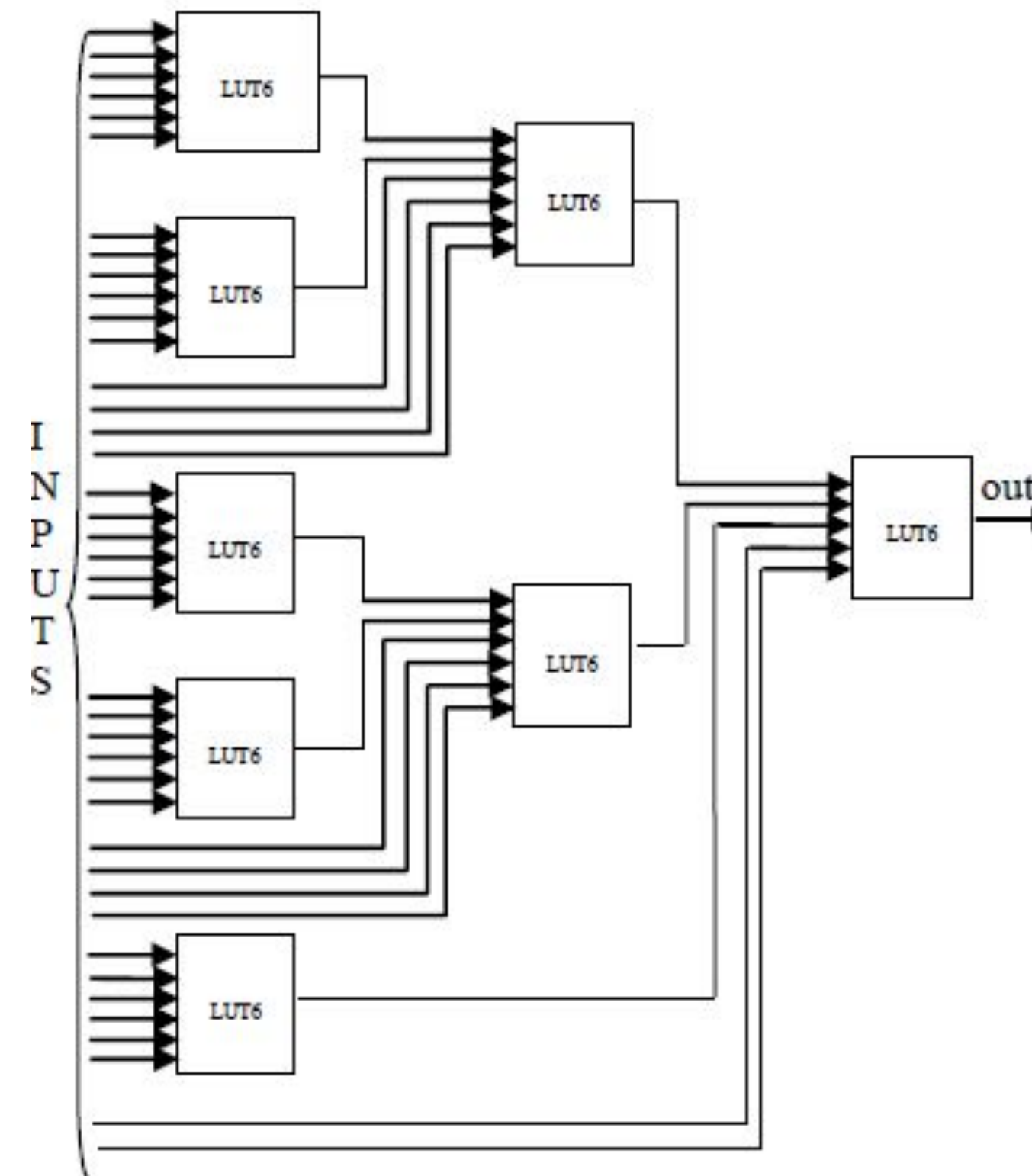
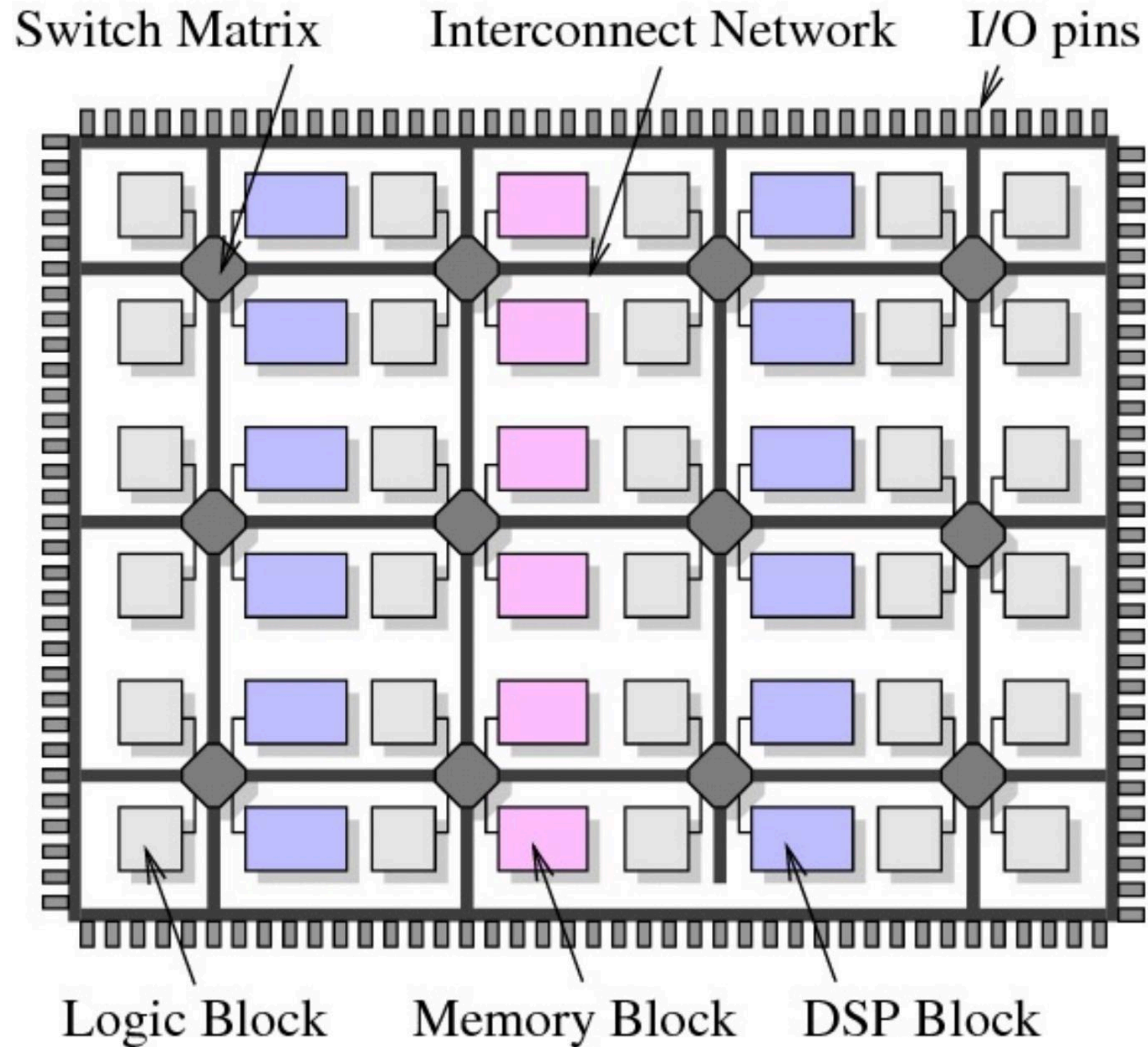


Image credit: [Zia 2013]

Modern FPGAs



- **A lot of area dedicated to hard gates**
 - **Memory blocks (SRAM)**
 - **DSP blocks (multipliers)**

Project Catapult

[Putnam et al. ISCA 2014]

- Microsoft Research investigation of use of FPGAs to accelerate datacenter workloads
- Demonstrated offload of part of Bing search's document ranking logic

FPGA board

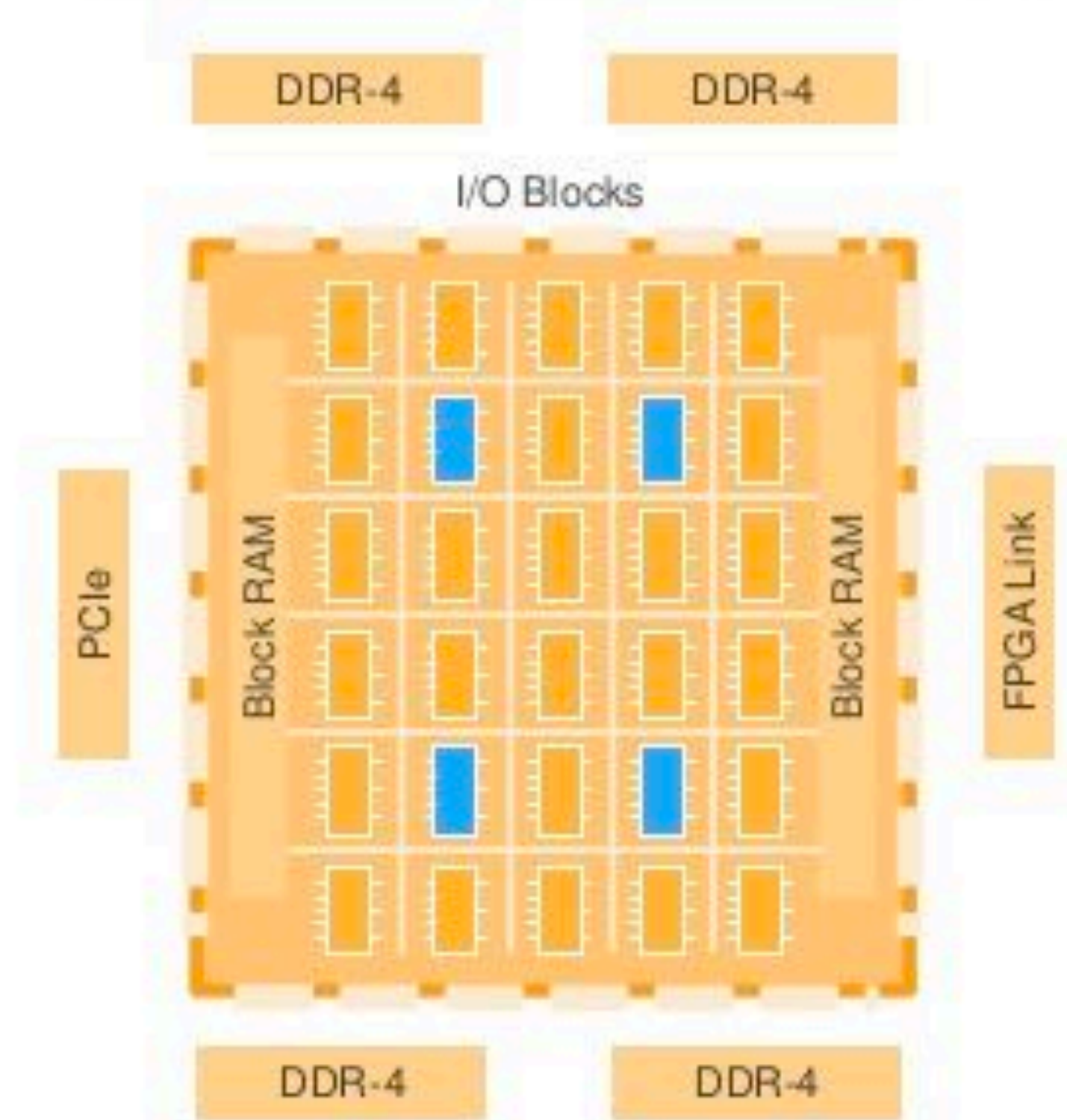


1U server (Dual socket CPU + FPGA connected via PCIe bus)

Amazon F1

- **FPGA's are now available on Amazon cloud services**

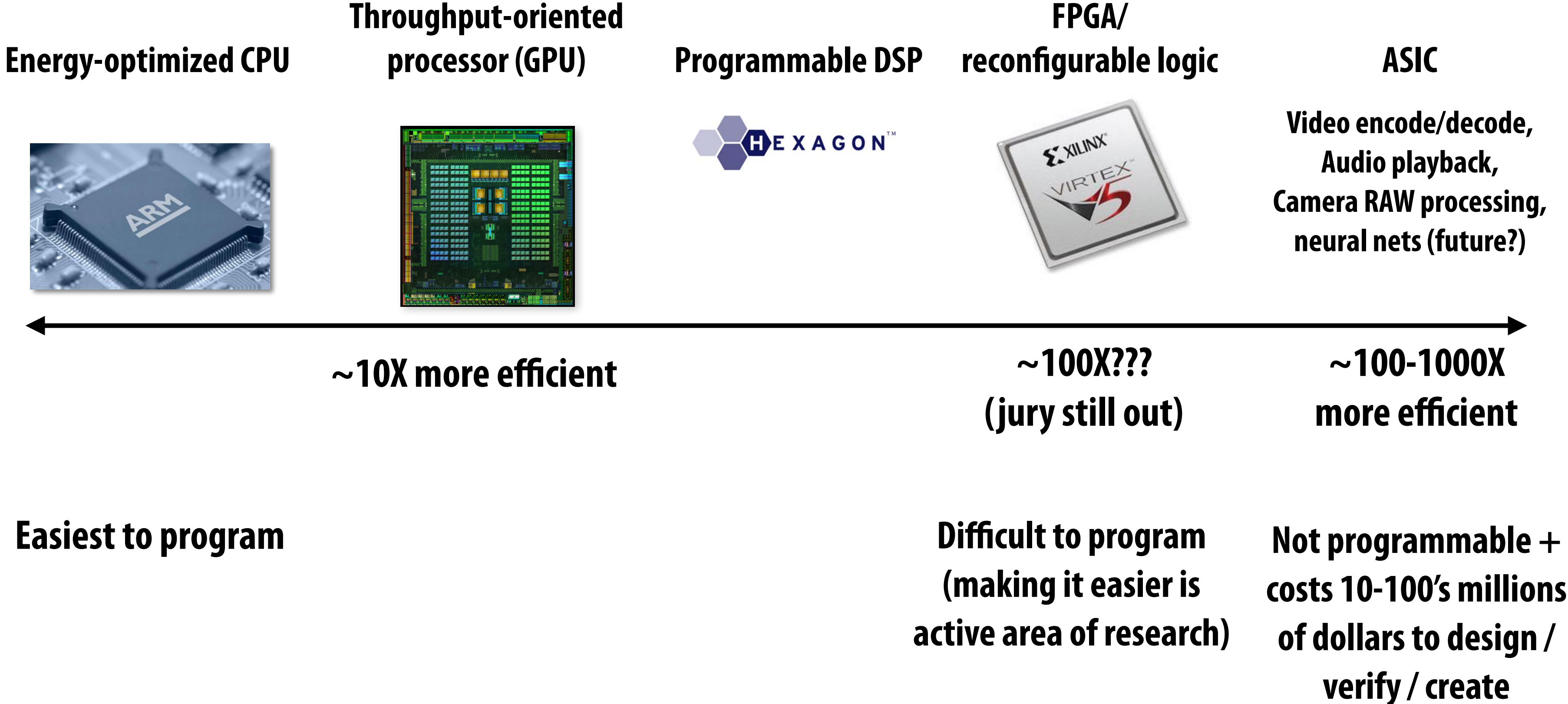
What's Inside the F1 FPGA?



- System Logic Block:**
Each FPGA in F1 provides over 2M of these logic blocks
- DSP (Math) Block:**
Each FPGA in F1 has more than 5000 of these blocks
- I/O Blocks:**
Used to communicate externally, for example to DDR-4, PCIe, or ring
- Block RAM:**
Each FPGA in F1 has over 60Mb of internal Block RAM, and over 230Mb of embedded UltraRAM



Summary: choosing the right tool for the job



Credit: Pat Hanrahan for this slide design

Energy-constrained computing

- **Supercomputers are energy constrained**
 - **Due to sheer scale**
 - **Overall cost to operate (power for machine and for cooling)**
- **Datacenters are energy constrained**
 - **Reduce cost of cooling**
 - **Reduce physical space requirements**
- **Mobile devices are energy constrained**
 - **Limited battery life**
 - **Heat dissipation**

Challenges of heterogeneous designs

(it's not easy to realize the potential of specialized, heterogeneous processing)

Challenges of heterogeneity

- **Heterogeneous system: preferred processor for each task**
- **Challenge to software developer: how to map application onto a heterogeneous collection of resources?**
 - **Challenge: “Pick the right tool for the job”**: design algorithms that decompose into components that each map well to different processing components of the machine
 - **The scheduling problem is more complex on a heterogeneous system**
- **Challenge for hardware designer: what is the right mixture of resources?**
 - **Too few throughput oriented resources (lower peak throughput for parallel workloads)**
 - **Too few sequential processing resources (limited by sequential part of workload)**
 - **How much chip area should be dedicated to a specific function, like video?**

**Reducing energy consumption idea 1:
use specialized processing**
(use the right processor for the job)

**Reducing energy consumption idea 2:
move less data**

Data movement has high energy cost

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

- **Rule of thumb in mobile system design: always seek to reduce amount of data transferred from memory**
 - **Earlier in class we discussed minimizing communication to reduce stalls (poor performance). Now, we wish to reduce communication to reduce energy consumption**
- **“Ballpark” numbers**
 - Integer op: ~ 1 pJ *
 - Floating point op: ~20 pJ *
 - Reading 64 bits from small local SRAM (1mm away on chip): ~ 26 pJ
 - Reading 64 bits from low power mobile DRAM (LPDDR): ~1200 pJ ← **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 SoC running graphics: ~6-10 watts
 - iPhone 12 battery: ~10 watt-hours (Macbook Pro M1 laptop: 58 watt-hour battery)
 - Exploiting locality matters!!!

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

Three trends in energy-optimized computing

■ Compute less!

- **Computing costs energy: parallel algorithms that do more work than sequential counterparts may not be desirable even if they run faster**

■ Specialize compute units:

- **Heterogeneous processors: CPU-like cores + throughput-optimized cores (GPU-like cores)**
- **Fixed-function units: audio processing, “movement sensor processing” video decode/encode, image processing/computer vision?**
- **Specialized instructions: expanding set of AVX vector instructions, instructions for accelerating AES encryption (AES-NI)**
- **Programmable soft logic: FPGAs**

■ Reduce bandwidth requirements

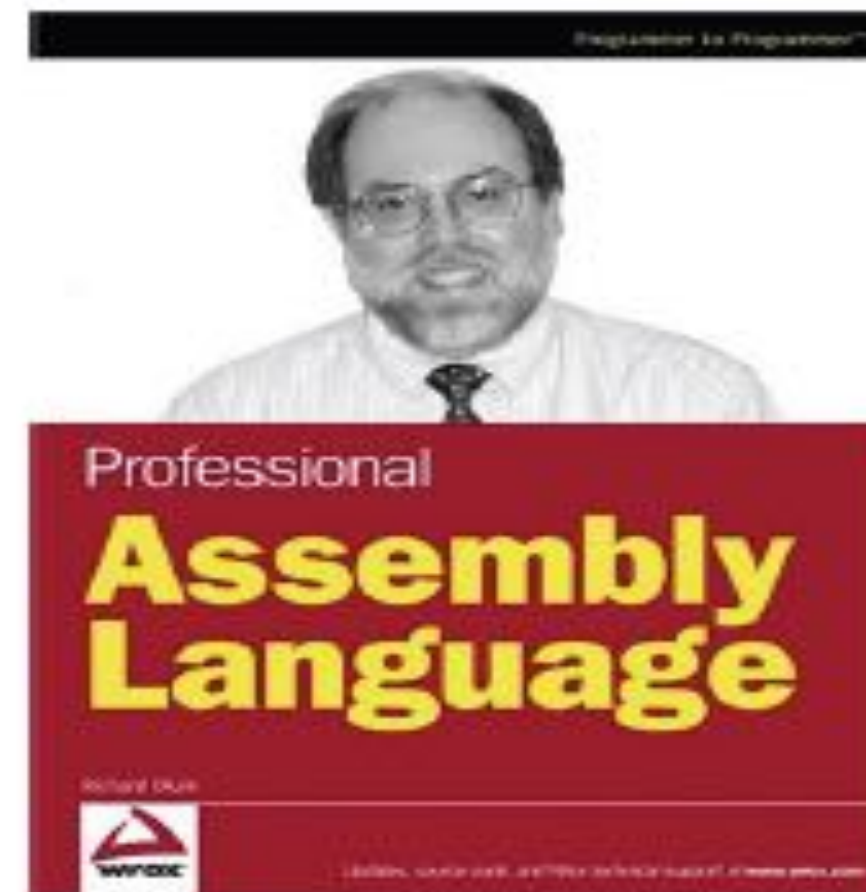
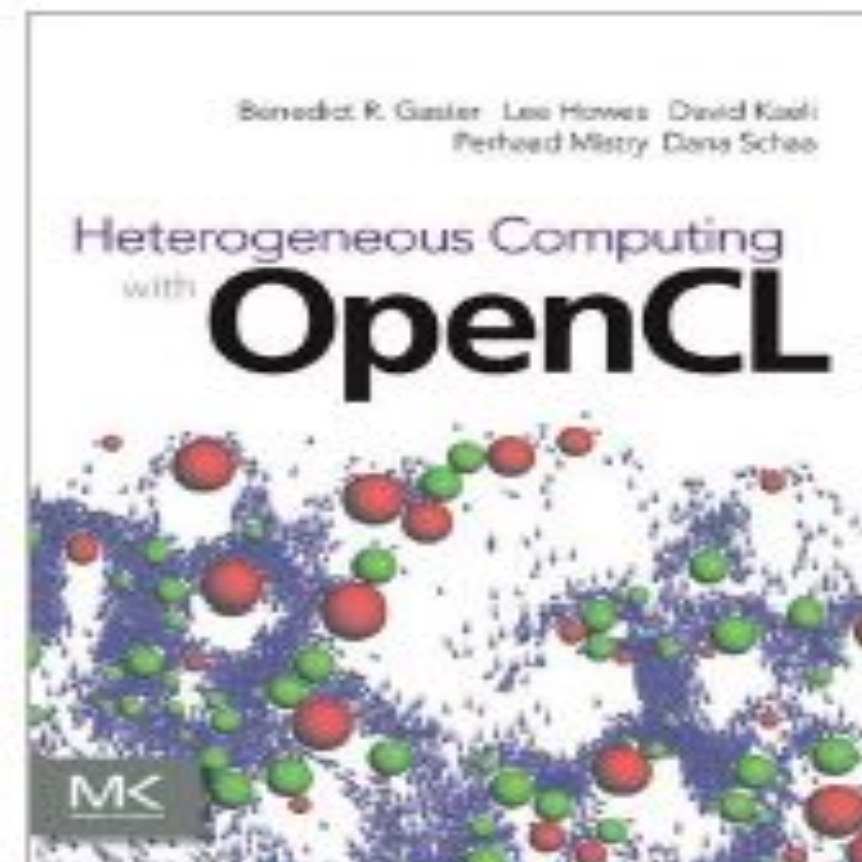
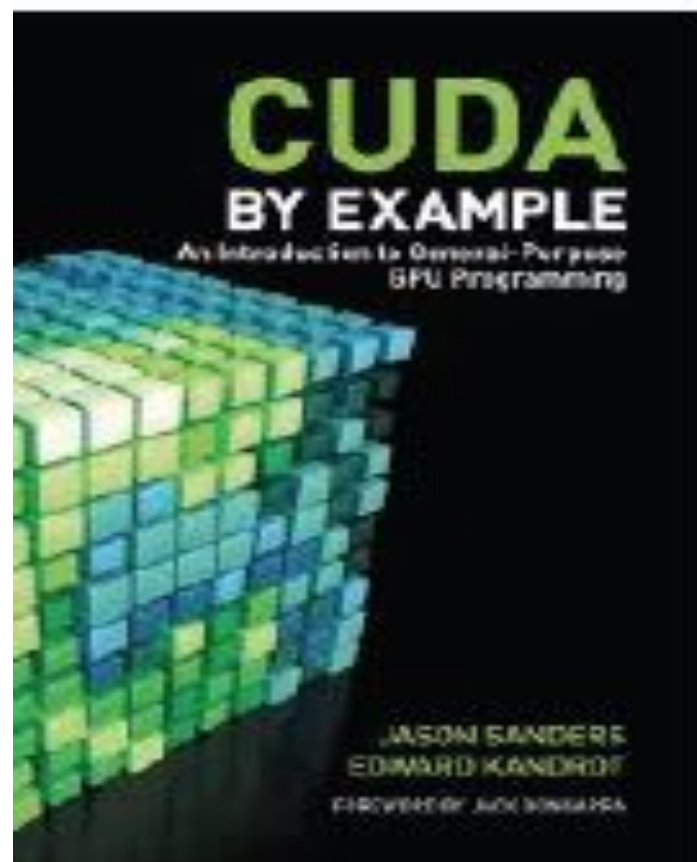
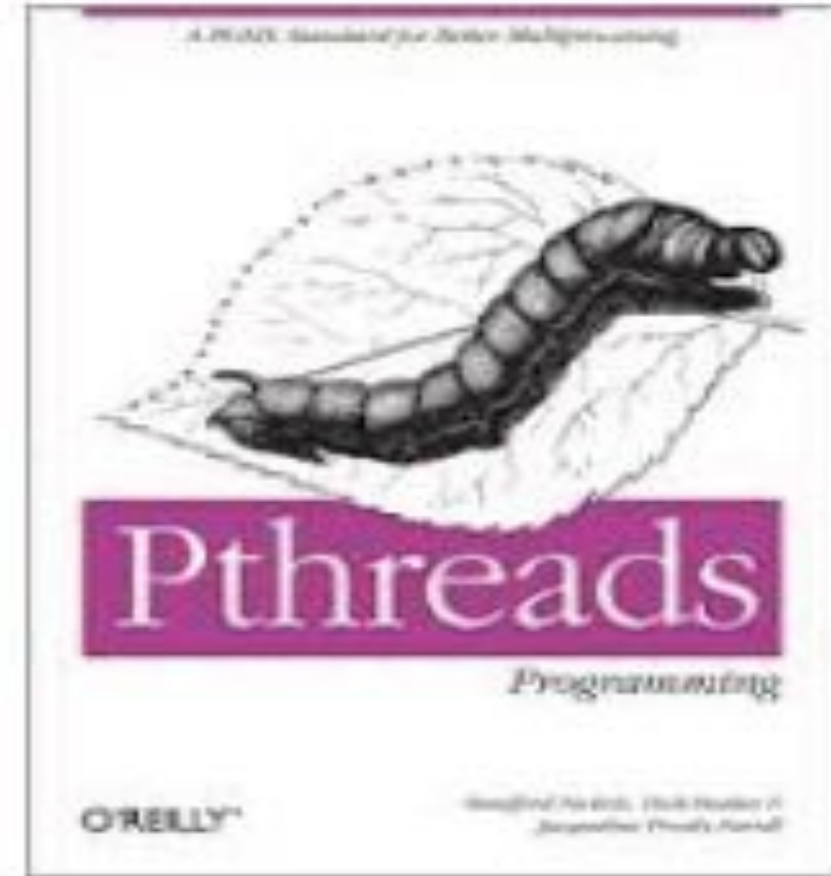
- **Exploit locality (restructure algorithms to reuse on-chip data as much as possible)**
- **Aggressive use of compression: perform extra computation to compress application data before transferring to memory (likely to see fixed-function HW to reduce overhead of general data compression/decompression)**

Summary: heterogeneous processing for efficiency

- **Heterogeneous parallel processing: use a mixture of computing resources that fit mixture of needs of target applications**
 - Latency-optimized sequential cores, throughput-optimized parallel cores, domain-specialized fixed-function processors
 - Examples exist throughout modern computing: mobile processors, servers, supercomputers
- **Traditional rule of thumb in “good system design” is to design simple, general-purpose components**
 - This is not the case in emerging systems (optimized for perf/watt)
 - Today: want collection of components that meet perf requirement AND minimize energy use
- **Challenge of using these resources effectively is pushed up to the programmer**
 - Current CS research challenge: how to write efficient, portable programs for emerging heterogeneous architectures?

Part 2:
**Programming heterogeneous machines with
domain specific programming languages**

EXPERT PROGRAMMERS \Rightarrow LOW PRODUCTIVITY



numa(3) - Linux man page

Name
numa - NUMA policy library

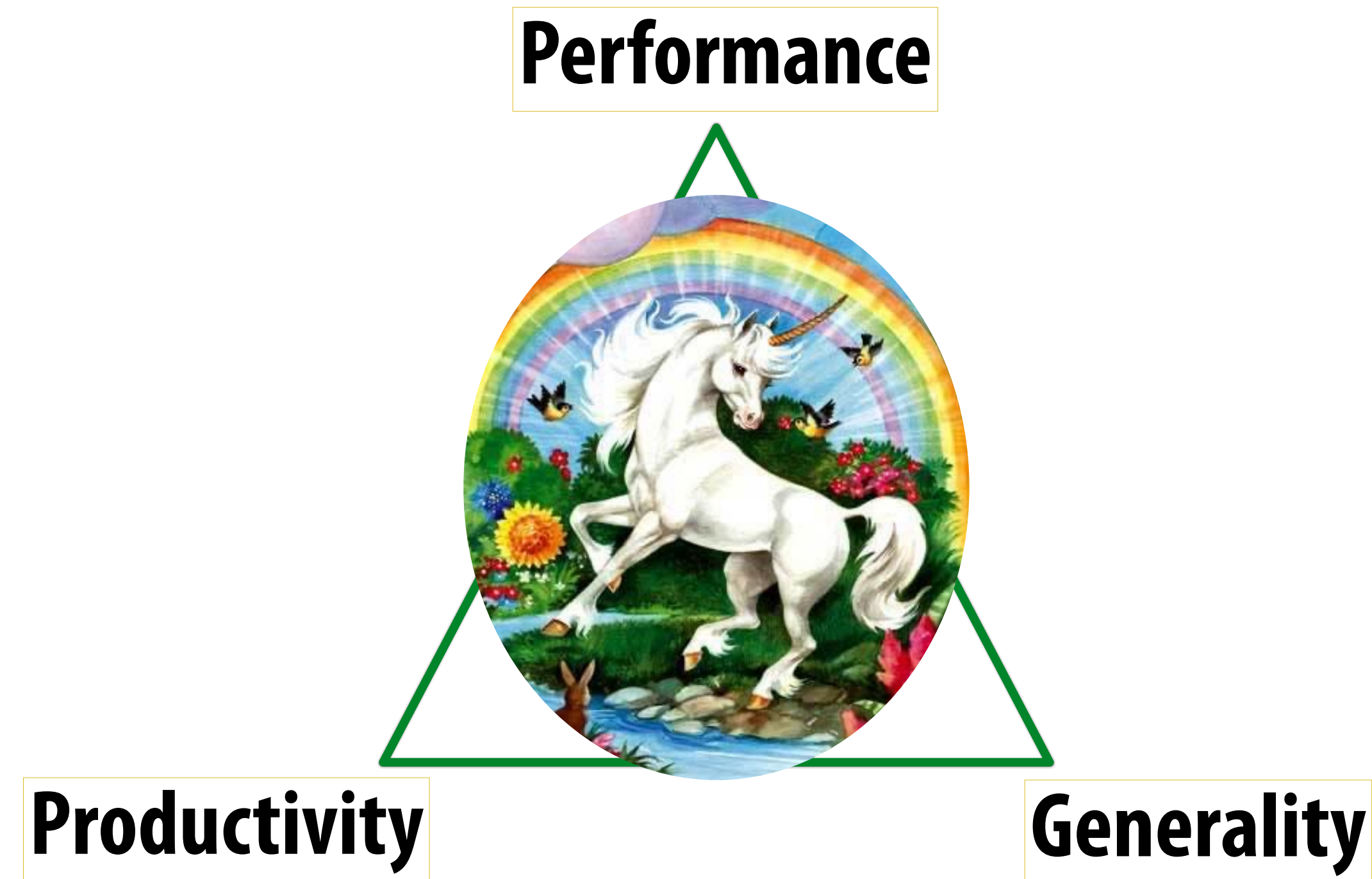
Synopsis
#include <numa.h>
cc ... -lnuma

Library
linux docs
linux man pages
online dictionary
page load time

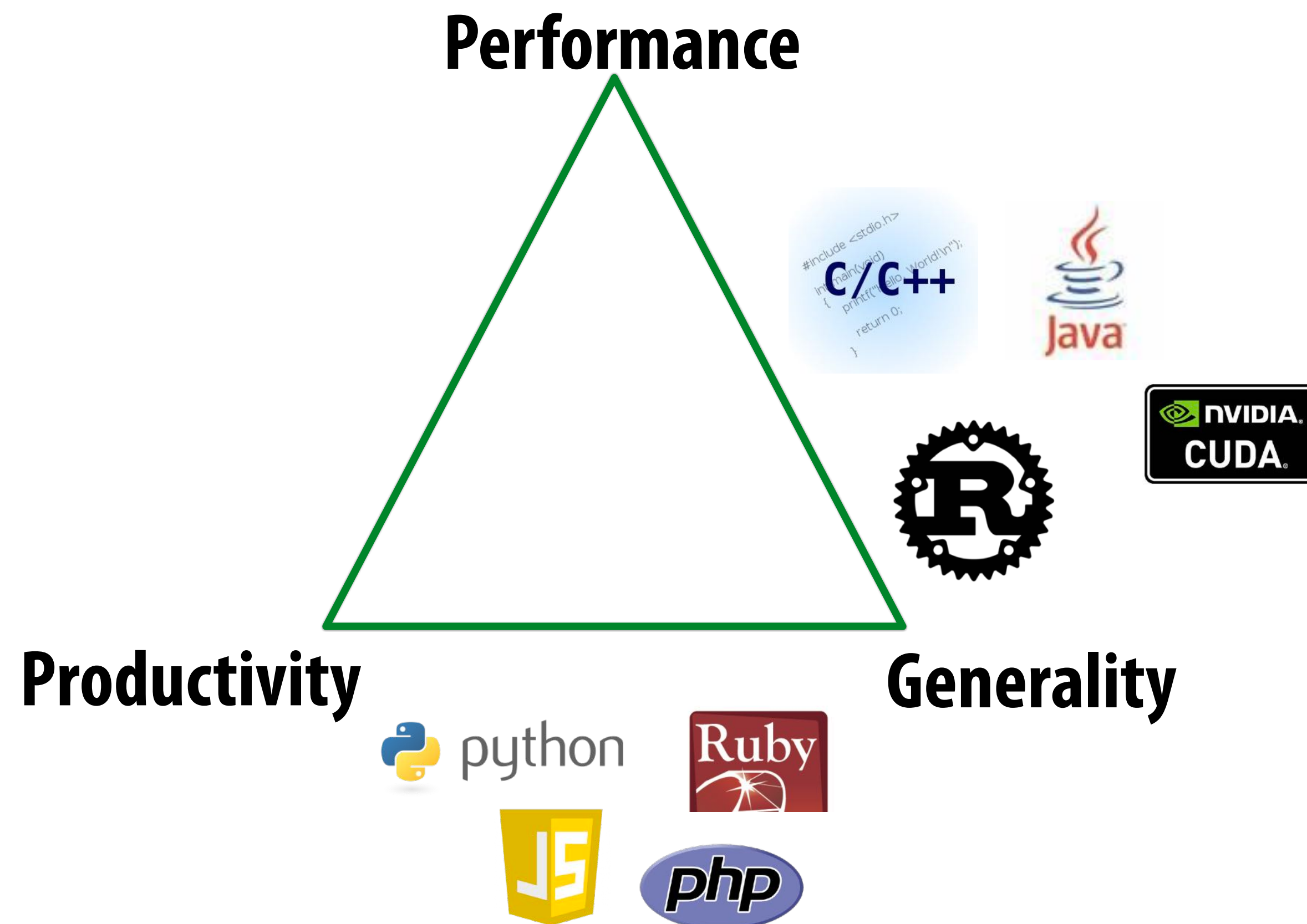
Toys
world sunlight
moon phase
trace explorer

```
int numa_available(void);
int numa_max_possible_node(void);
int numa_num_possible_nodes();
int numa_max_node(void);
int numa_num_configured_nodes();
struct bitmask *numa_get_mems_allowed();
int numa_num_configured_cpus(void);
struct bitmask *numa_all_nodes_ptr;
struct bitmask *numa_no_nodes_ptr;
struct bitmask *numa_all_cpus_ptr;
int numa_num_task_cpus();
int numa_num_task_nodes();
```

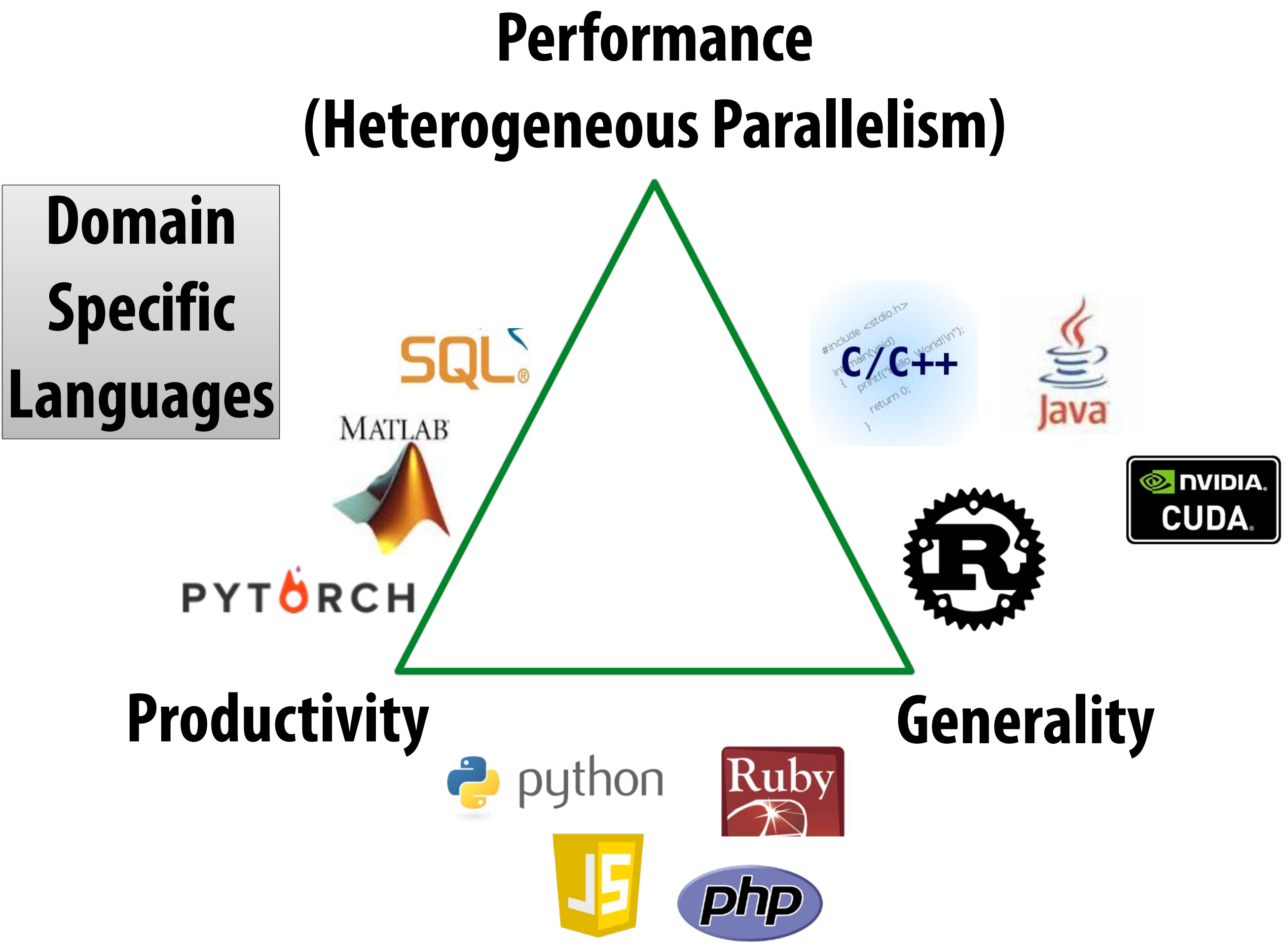
The ideal parallel programming language



Popular languages (not exhaustive ;-))



Way forward \Rightarrow domain-specific languages



Credit: Pat Hanrahan for this slide design

DSL hypothesis

It is possible to write one program...

and

run it efficiently on a range of heterogeneous parallel systems

Domain Specific Languages

- **Domain Specific Languages (DSLs)**
 - **Programming language with restricted expressiveness for a particular domain**
 - **High-level, usually declarative, and deterministic**



Domain-specific programming systems

- **Main idea: raise level of abstraction for expressing programs**
 - **Goal: write one program, and run it efficiently on different machines**
- **Introduce high-level programming primitives specific to an application domain**
 - **Productive:** intuitive to use, portable across machines, primitives correspond to behaviors frequently used to solve problems in targeted domain
 - **Performant:** system uses domain knowledge to provide efficient, optimized implementation(s)
 - **Given a machine:** system knows what algorithms to use, parallelization strategies to employ for this domain
 - **Optimization goes beyond efficient mapping of software to hardware! The hardware platform itself can be optimized to the abstractions as well**
- **Cost: loss of generality/completeness**

A DSL example:

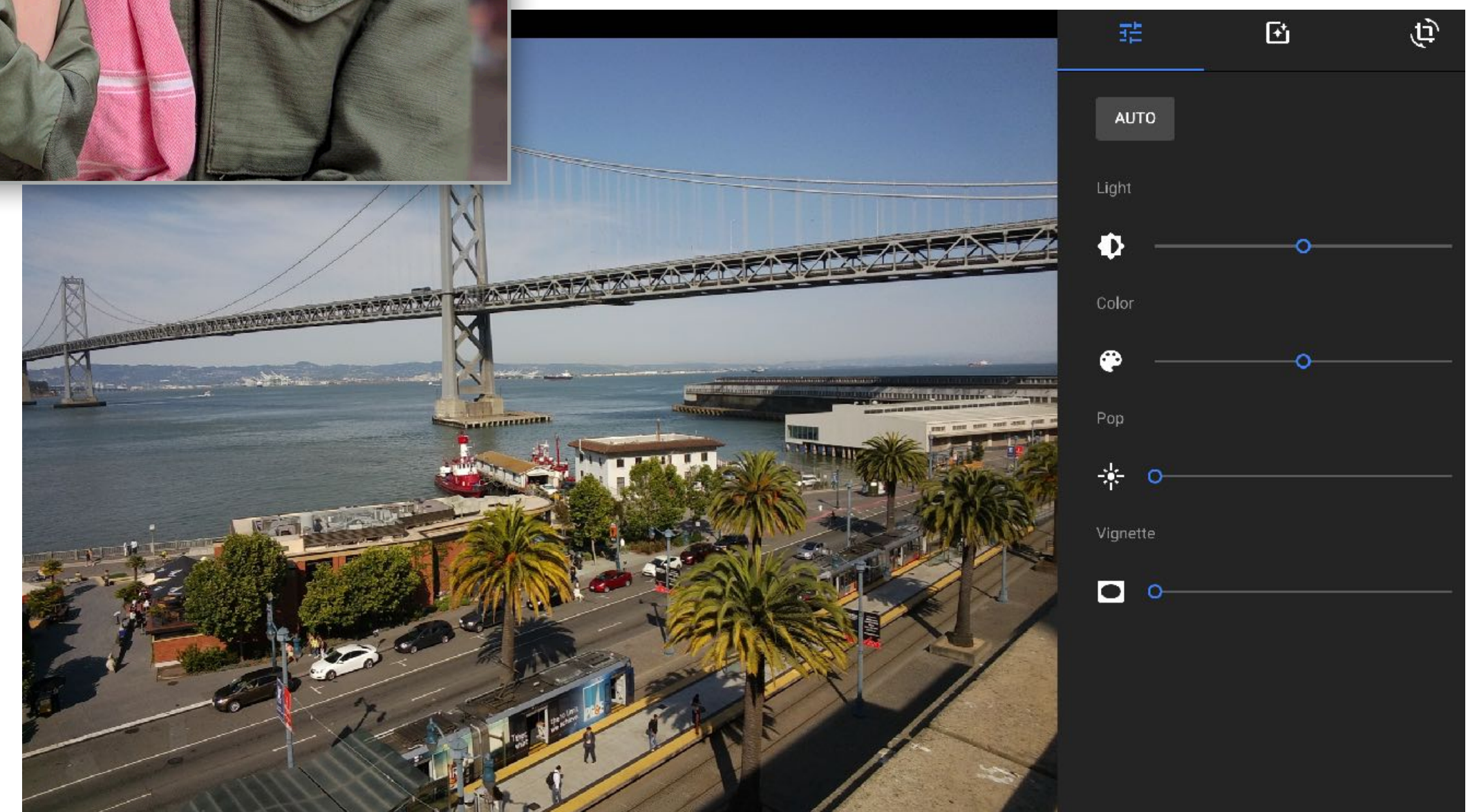
Halide: a domain-specific language for image processing

Jonathan Ragan-Kelley, Andrew Adams et al.

[SIGGRAPH 2012, PLDI 13]

Halide used in practice

- Halide used to implement camera processing pipelines on Google phones
 - HDR+, aspects of portrait mode, etc...
- Industry usage at Instagram, Adobe, etc.



A quick tutorial on high-performance image processing

What does this code do? 🤔😱😞😭

Good: ~10x faster on a quad-core CPU than my original two-pass code

Bad: specific to SSE (not AVX2), CPU-code only, hard to tell what is going on at all!

```
void fast_blur(const Image &in, Image &blurred) {
    _m128i one_third = _mm_set1_epi16(21846);
    #pragma omp parallel for
    for (int yTile = 0; yTile < in.height(); yTile += 32) {
        _m128i a, b, c, sum, avg;
        _m128i tmp[(256/8)*(32+2)];
        for (int xTile = 0; xTile < in.width(); xTile += 256) {
            _m128i *tmpPtr = tmp;
            for (int y = -1; y < 32+1; y++) {
                const uint16_t *inPtr = &(in(xTile, yTile+y));
                for (int x = 0; x < 256; x += 8) {
                    a = _mm_loadu_si128((__m128i*)(inPtr-1));
                    b = _mm_loadu_si128((__m128i*)(inPtr+1));
                    c = _mm_load_si128((__m128i*)(inPtr));
                    sum = _mm_add_epi16(_mm_add_epi16(a, b), c);
                    avg = _mm_mulhi_epi16(sum, one_third);
                    _mm_store_si128(tmpPtr++, avg);
                    inPtr += 8;
                }
                tmpPtr = tmp;
            }
            for (int y = 0; y < 32; y++) {
                _m128i *outPtr = (_m128i *)(&(blurred(xTile, yTile+y)));
                for (int x = 0; x < 256; x += 8) {
                    a = _mm_load_si128(tmpPtr+(2*256)/8);
                    b = _mm_load_si128(tmpPtr+256/8);
                    c = _mm_load_si128(tmpPtr++);
                    sum = _mm_add_epi16(_mm_add_epi16(a, b), c);
                    avg = _mm_mulhi_epi16(sum, one_third);
                    _mm_store_si128(outPtr++, avg);
                }
            }
        }
    }
}
```

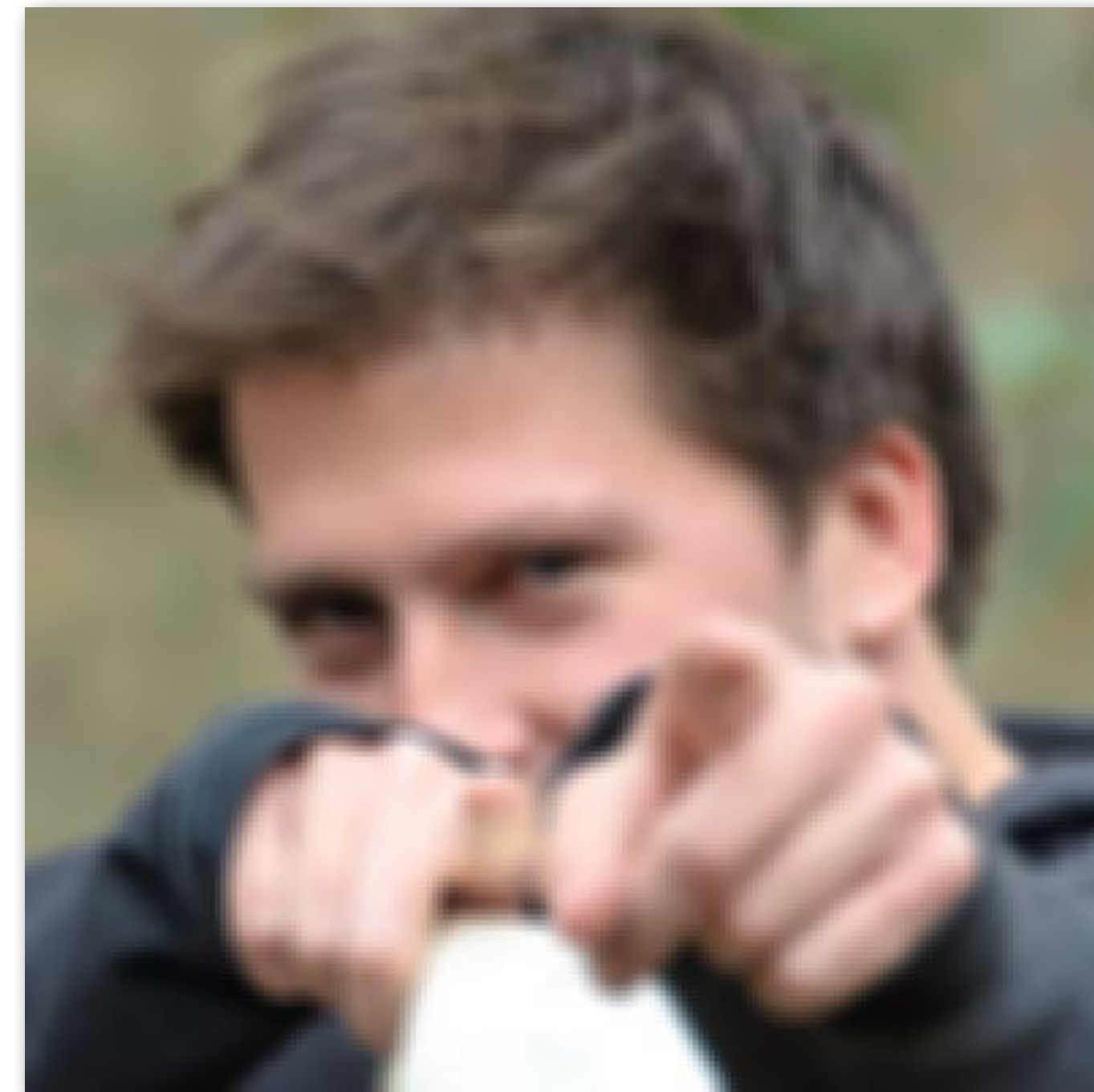
What does this C code do?

```
int WIDTH = 1024;
int HEIGHT = 1024;
float input[(WIDTH+2) * (HEIGHT+2)];
float output[WIDTH * HEIGHT];

float weights[] = {1.f/9, 1.f/9, 1.f/9,
                  1.f/9, 1.f/9, 1.f/9,
                  1.f/9, 1.f/9, 1.f/9};

for (int j=0; j<HEIGHT; j++) {
    for (int i=0; i<WIDTH; i++) {
        float tmp = 0.f;
        for (int jj=0; jj<3; jj++)
            for (int ii=0; ii<3; ii++)
                tmp += input[(j+jj)*(WIDTH+2) + (i+ii)] * weights[jj*3 + ii];
        output[j*WIDTH + i] = tmp;
    }
}
```

The code on the previous slide performed a 3x3 box blur



(Zoomed view)

3x3 image blur

```
int WIDTH = 1024;
int HEIGHT = 1024;
float input[(WIDTH+2) * (HEIGHT+2)];
float output[WIDTH * HEIGHT];

float weights[] = {1.f/9, 1.f/9, 1.f/9,
                  1.f/9, 1.f/9, 1.f/9,
                  1.f/9, 1.f/9, 1.f/9};

for (int j=0; j<HEIGHT; j++) {
    for (int i=0; i<WIDTH; i++) {
        float tmp = 0.f;
        for (int jj=0; jj<3; jj++)
            for (int ii=0; ii<3; ii++)
                tmp += input[(j+jj)*(WIDTH+2) + (i+ii)] * weights[jj*3 + ii];
        output[j*WIDTH + i] = tmp;
    }
}
```

Total work per image = 9 x WIDTH x HEIGHT

For NxN filter: N^2 x WIDTH x HEIGHT

Two-pass blur

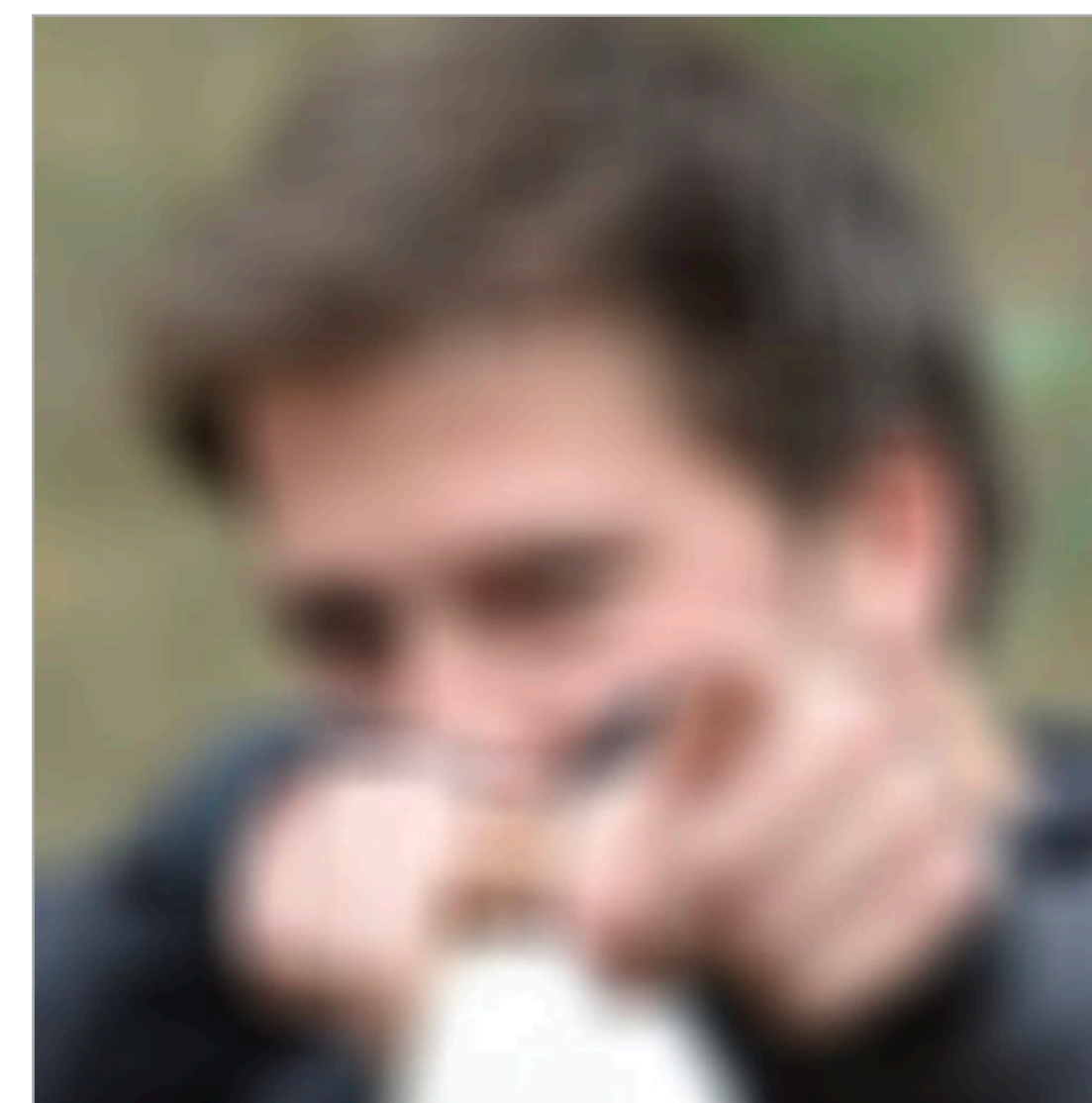
A 2D separable filter (such as a box filter) can be evaluated via two 1D filtering operations



Input



Horizontal Blur



Vertical Blur

Note: I've exaggerated the blur for illustration (the end result is actually a 30x30 blur, not 3x3)

Two-pass 3x3 blur

```
int WIDTH = 1024;
int HEIGHT = 1024;
float input[(WIDTH+2) * (HEIGHT+2)];
float tmp_buf[WIDTH * (HEIGHT+2)];
float output[WIDTH * HEIGHT];

float weights[] = {1.f/3, 1.f/3, 1.f/3};

for (int j=0; j<(HEIGHT+2); j++)
  for (int i=0; i<WIDTH; i++) {
    float tmp = 0.f;
    for (int ii=0; ii<3; ii++)
      tmp += input[j*(WIDTH+2) + i+ii] * weights[ii];
    tmp_buf[j*WIDTH + i] = tmp;
  }

for (int j=0; j<HEIGHT; j++) {
  for (int i=0; i<WIDTH; i++) {
    float tmp = 0.f;
    for (int jj=0; jj<3; jj++)
      tmp += tmp_buf[(j+jj)*WIDTH + i] * weights[jj];
    output[j*WIDTH + i] = tmp;
  }
}
```

1D horizontal blur

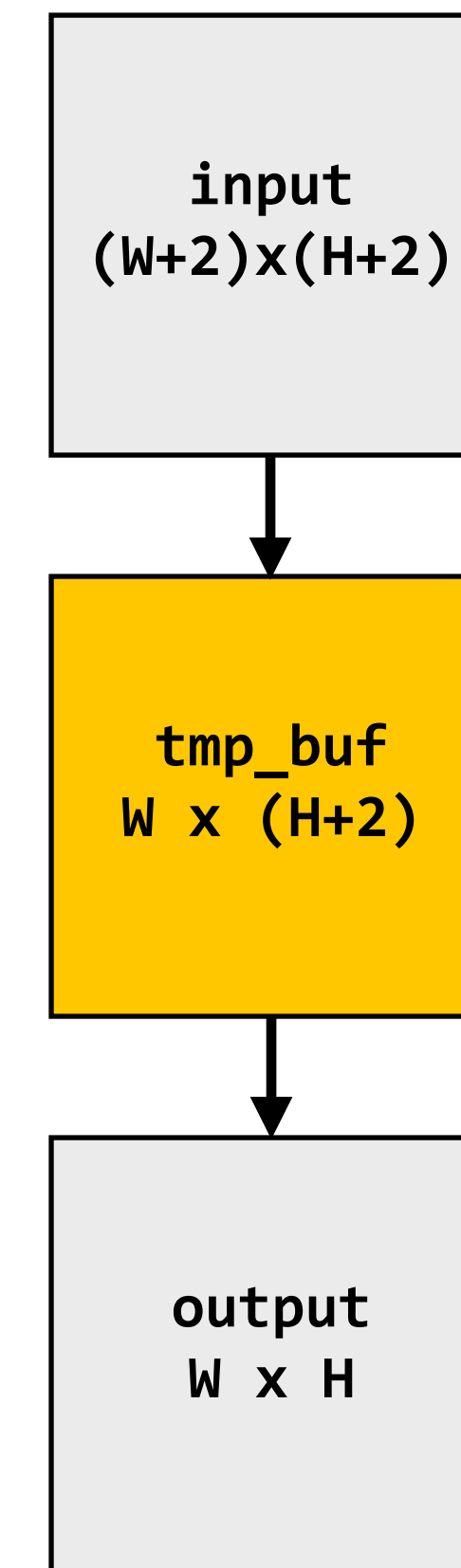
1D vertical blur

Total work per image = $6 \times \text{WIDTH} \times \text{HEIGHT}$

For $N \times N$ filter: $2N \times \text{WIDTH} \times \text{HEIGHT}$

$\text{WIDTH} \times \text{HEIGHT}$ extra storage

2x lower arithmetic intensity than 2D blur. Why?



Two-pass image blur: locality

```
int WIDTH = 1024;
int HEIGHT = 1024;
float input[(WIDTH+2) * (HEIGHT+2)];
float tmp_buf[WIDTH * (HEIGHT+2)];
float output[WIDTH * HEIGHT];

float weights[] = {1.f/3, 1.f/3, 1.f/3};

for (int j=0; j<(HEIGHT+2); j++)
  for (int i=0; i<WIDTH; i++) {
    float tmp = 0.f;
    for (int ii=0; ii<3; ii++)
      tmp += input[j*(WIDTH+2) + i+ii] * weights[ii];
    tmp_buf[j*WIDTH + i] = tmp;
  }

for (int j=0; j<HEIGHT; j++) {
  for (int i=0; i<WIDTH; i++) {
    float tmp = 0.f;
    for (int jj=0; jj<3; jj++)
      tmp += tmp_buf[(j+jj)*WIDTH + i] * weights[jj];
    output[j*WIDTH + i] = tmp;
  }
}
```

**Intrinsic bandwidth requirements of blur algorithm:
Application must read each element of input image and it must write each element of output image.**

Data from `input` reused three times. (immediately reused in next two `i`-loop iterations after first load, never loaded again.)

- Perfect cache behavior: never load required data more than once
- Perfect use of cache lines (don't load unnecessary data into cache)

Two pass: loads/stores to `tmp_buf` are overhead (this memory traffic is an artifact of the two-pass implementation: it is not intrinsic to computation being performed)

Data from `tmp_buf` reused three times (but three rows of image data are accessed in between)

- Never load required data more than once... if cache has capacity for three rows of image
- Perfect use of cache lines (don't load unnecessary data into cache)

Two-pass image blur, "chunked" (version 1)

```
int WIDTH = 1024;
int HEIGHT = 1024;
float input[(WIDTH+2) * (HEIGHT+2)];
float tmp_buf[WIDTH * 3];
float output[WIDTH * HEIGHT];

float weights[] = {1.f/3, 1.f/3, 1.f/3};

for (int j=0; j<HEIGHT; j++) {
    for (int j2=0; j2<3; j2++)
        for (int i=0; i<WIDTH; i++) {
            float tmp = 0.f;
            for (int ii=0; ii<3; ii++)
                tmp += input[(j+j2)*(WIDTH+2) + i+ii] * weights[ii];
            tmp_buf[j2*WIDTH + i] = tmp;
        }

    for (int i=0; i<WIDTH; i++) {
        float tmp = 0.f;
        for (int jj=0; jj<3; jj++)
            tmp += tmp_buf[jj*WIDTH + i] * weights[jj];
        output[j*WIDTH + i] = tmp;
    }
}
```

Only 3 rows of intermediate buffer need to be allocated

Produce 3 rows of tmp_buf (only what's needed for one row of output)

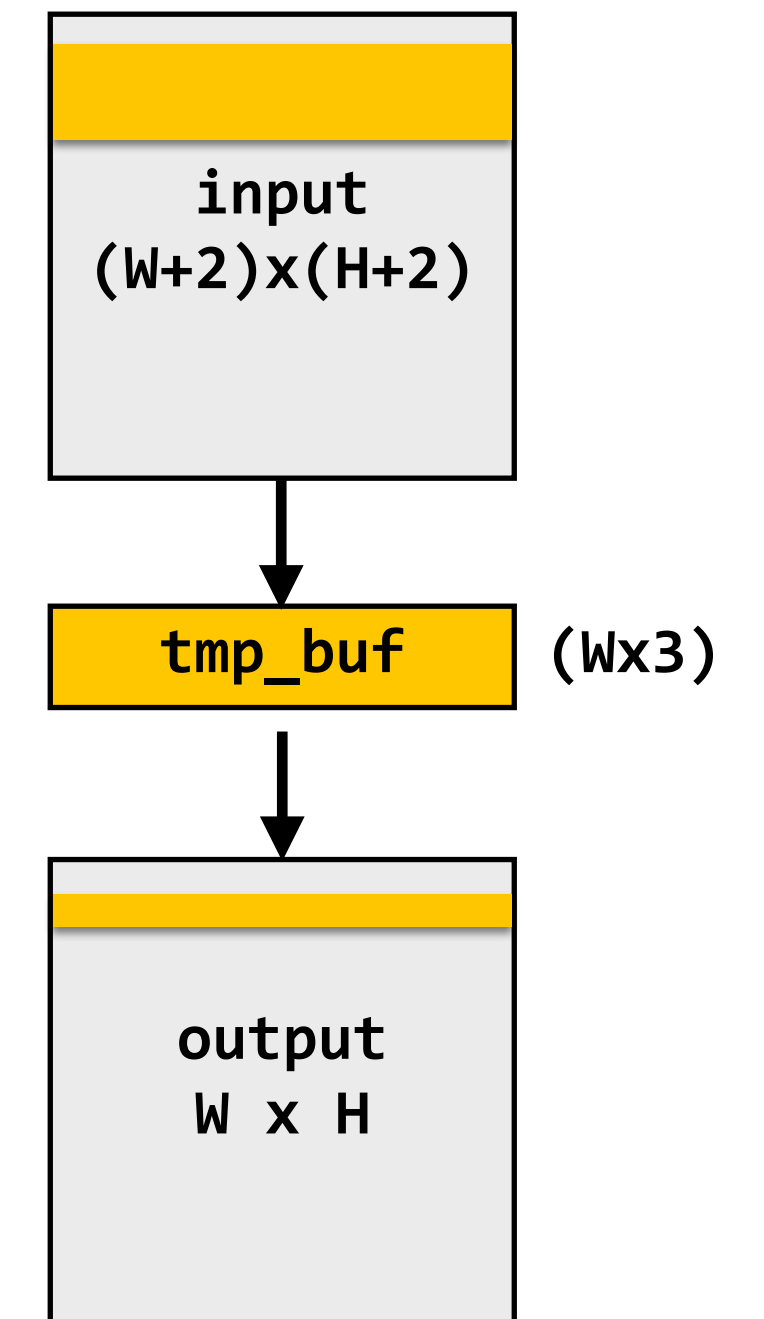
Combine them together to get one row of output

Total work per row of output:

- step 1: 3 x 3 x WIDTH work
- step 2: 3 x WIDTH work

Total work per image = 12 x WIDTH x HEIGHT ????

Loads from tmp_buffer are cached (assuming tmp_buffer fits in cache)



Two-pass image blur, "chunked" (version 2)

```
int WIDTH = 1024;
int HEIGHT = 1024;
float input[(WIDTH+2) * (HEIGHT+2)];
float tmp_buf[WIDTH * (CHUNK_SIZE+2)];
float output[WIDTH * HEIGHT];

float weights[] = {1.f/3, 1.f/3, 1.f/3};

for (int j=0; j<HEIGHT; j+CHUNK_SIZE) {
    for (int j2=0; j2<CHUNK_SIZE+2; j2++)
        for (int i=0; i<WIDTH; i++) {
            float tmp = 0.f;
            for (int ii=0; ii<3; ii++)
                tmp += input[(j+j2)*(WIDTH+2) + i+ii] * weights[ii];
            tmp_buf[j2*WIDTH + i] = tmp;
        }
    for (int j2=0; j2<CHUNK_SIZE; j2++)
        for (int i=0; i<WIDTH; i++) {
            float tmp = 0.f;
            for (int jj=0; jj<3; jj++)
                tmp += tmp_buf[(j2+jj)*WIDTH + i] * weights[jj];
            output[(j+j2)*WIDTH + i] = tmp;
        }
}
```

Sized so entire buffer fits in cache
(capture all producer-consumer locality)

Produce enough rows of tmp_buf to
produce a CHUNK_SIZE number of rows
of output

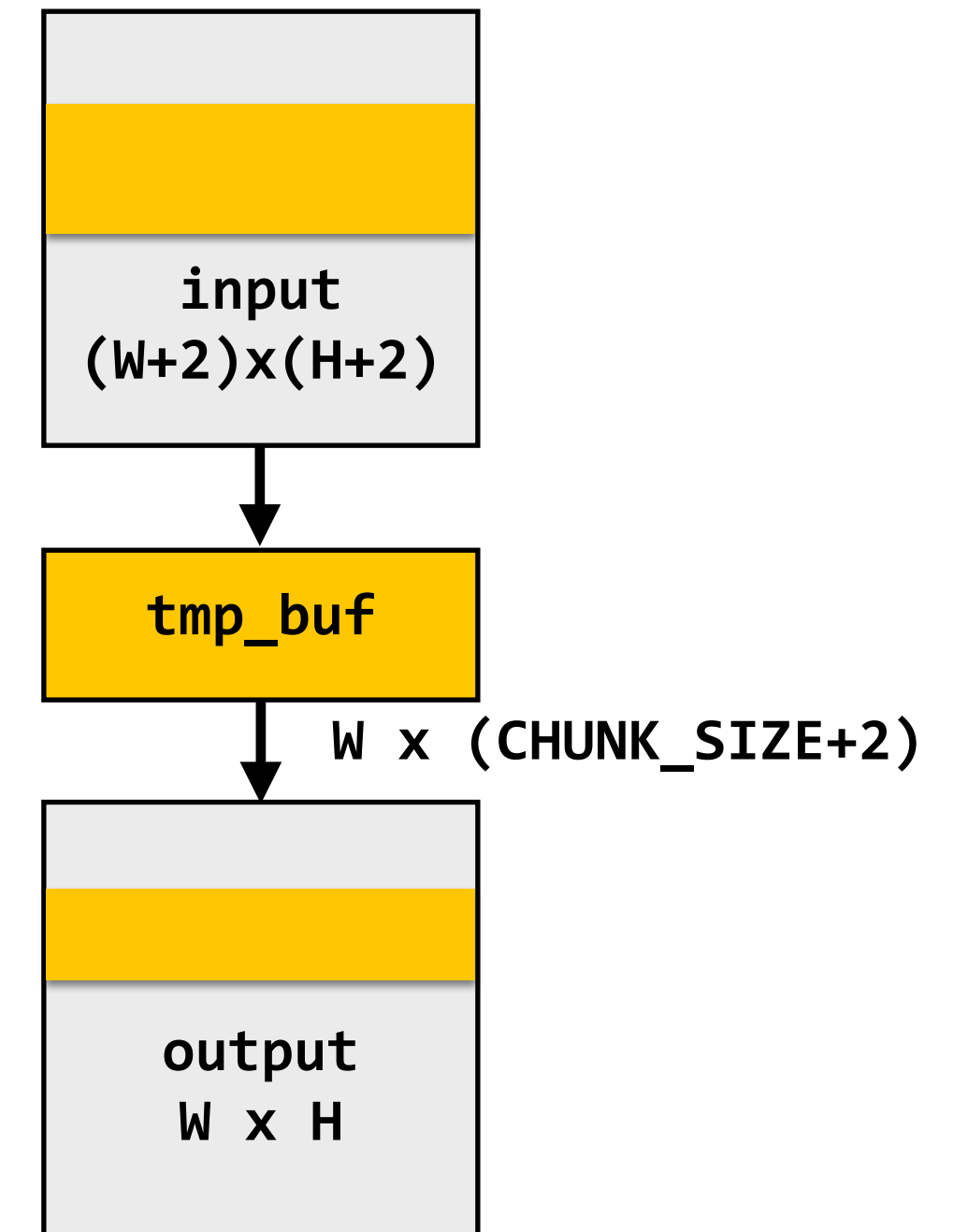
Produce CHUNK_SIZE rows of output

Total work per chunk of output: (assume CHUNK_SIZE = 16)

- Step 1: 18 x 3 x WIDTH work
- Step 2: 16 x 3 x WIDTH work

Total work per image: $(34/16) \times 3 \times \text{WIDTH} \times \text{HEIGHT}$
 $= 6.4 \times \text{WIDTH} \times \text{HEIGHT}$

Trends to ideal value of 6 x WIDTH x HEIGHT as CHUNK_SIZE is increased!



Still not done

- **We have not parallelized loops for multi-core execution**
- **We have not used SIMD instructions to execute loops bodies**
- **Other basic optimizations: loop unrolling, etc...**

Optimized C++ code: 3x3 image blur 🤔😱😞😭

Good: ~10x faster on a quad-core CPU than my original two-pass code

Bad: specific to SSE (not AVX2), CPU-code only, hard to tell what is going on at all!

```
void fast_blur(const Image &in, Image &blurred) {
    __m128i one_third = _mm_set1_epi16(21846);
    #pragma omp parallel for
    for (int yTile = 0; yTile < in.height(); yTile += 32) {
        __m128i a, b, c, sum, avg;
        __m128i tmp[(256/8)*(32+2)];
        for (int xTile = 0; xTile < in.width(); xTile += 256) {
            __m128i *tmpPtr = tmp;
            for (int y = -1; y < 32+1; y++) {
                const uint16_t *inPtr = &(in(xTile, yTile+y));
                for (int x = 0; x < 256; x += 8) {
                    a = _mm_loadu_si128((__m128i*)(inPtr-1));
                    b = _mm_loadu_si128((__m128i*)(inPtr+1));
                    c = _mm_load_si128((__m128i*)(inPtr));
                    sum = _mm_add_epi16(_mm_add_epi16(a, b), c);
                    avg = _mm_mulhi_epi16(sum, one_third);
                    _mm_store_si128(tmpPtr++, avg);
                    inPtr += 8;
                }
            }
            tmpPtr = tmp;
            for (int y = 0; y < 32; y++) {
                __m128i *outPtr = (__m128i *)&(blurred(xTile, yTile+y));
                for (int x = 0; x < 256; x += 8) {
                    a = _mm_load_si128(tmpPtr+(2*256)/8);
                    b = _mm_load_si128(tmpPtr+256/8);
                    c = _mm_load_si128(tmpPtr++);
                    sum = _mm_add_epi16(_mm_add_epi16(a, b), c);
                    avg = _mm_mulhi_epi16(sum, one_third);
                    _mm_store_si128(outPtr++, avg);
                }
            }
        }
    }
}
```

Multi-core execution
(partition image vertically)

Modified iteration order:
256x32 tiled iteration (to
maximize cache hit rate)

use of SIMD vector
intrinsics

two passes fused into one:
tmp data read from cache

Halide language

[Ragan-Kelley / Adams 2012]

Simple domain-specific language embedded in C++ for describing sequences of image processing operations

```
Var x, y;  
Func blurx, blurry, bright, out;  
Halide::Buffer<uint8_t> in = load_image("myimage.jpg");  
Halide::Buffer<uint8_t> lookup = load_image("s_curve.jpg"); // 255-pixel 1D image
```

“Functions” map integer coordinates to values
(e.g., colors of corresponding pixels)

```
// perform 3x3 box blur in two-passes  
blurx(x,y) = 1/3.f * (in(x-1,y) + in(x,y) + in(x+1,y));  
blurry(x,y) = 1/3.f * (blurx(x,y-1) + blurx(x,y) + blurx(x,y+1));
```

Value of `blurx` at coordinate `(x,y)` is given by
expression accessing three values of `in`

```
// brighten blurred result by 25%, then clamp  
bright(x,y) = min(blurry(x,y) * 1.25f, 255);
```

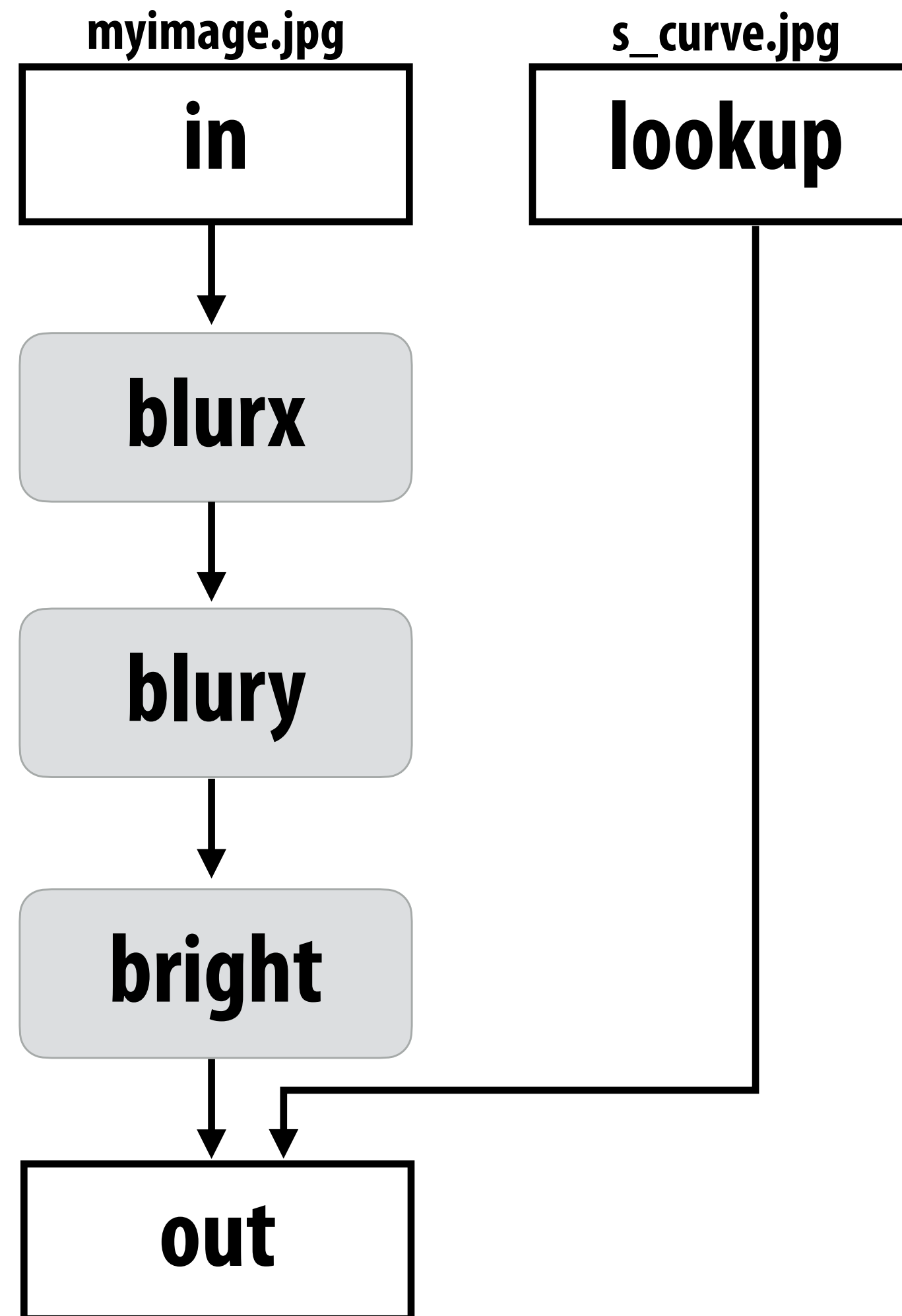
```
// access lookup table to contrast enhance  
out(x,y) = lookup(bright(x,y));
```

```
// execute pipeline to materialize values of out in range (0:1024,0:1024)  
Halide::Buffer<uint8_t> result = out.realize(1024, 1024);
```

Halide function: an infinite (but discrete) set of values defined on N-D domain

Halide expression: a side-effect free expression that describes how to compute a function's value at a point in its domain in terms of the values of other functions.

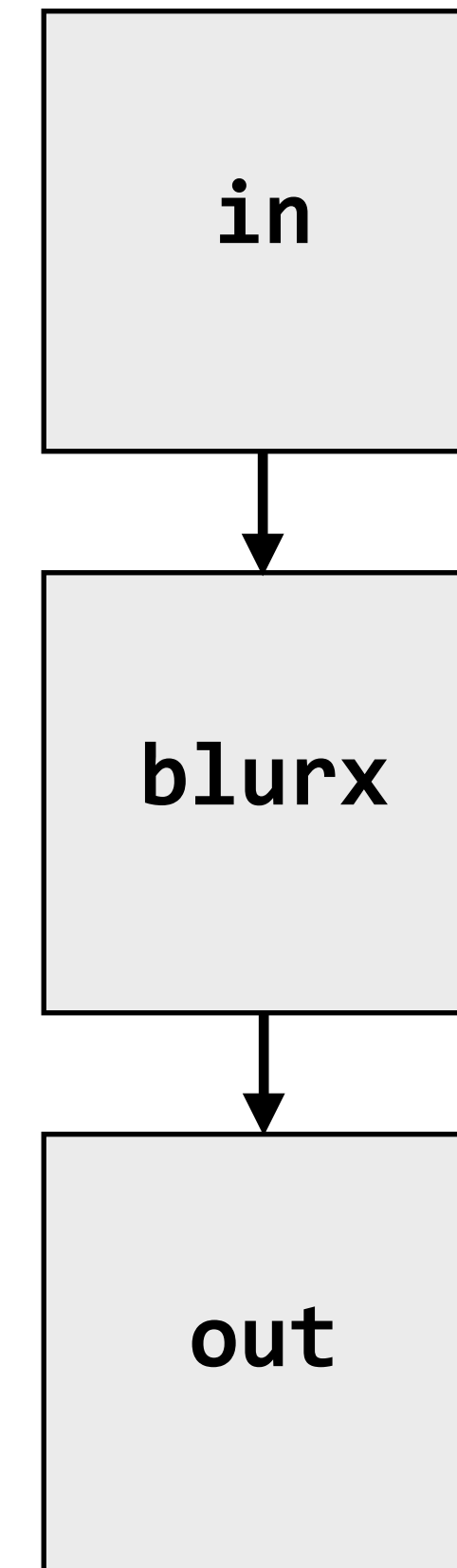
Image processing application as a DAG



Key aspects of representation

- Intuitive expression:
 - Adopts local “point wise” view of expressing algorithms
 - Halide language is declarative. It does not define order of iteration, or what values in domain are stored!
 - **It only defines what is needed to compute these values.**
 - **Iteration over domain points is implicit (no explicit loops)**

```
Var x, y;  
Func blurx, out;  
Halide::Buffer<uint8_t> in = load_image("myimage.jpg");  
  
// perform 3x3 box blur in two-passes  
blurx(x,y) = 1/3.f * (in(x-1,y) + in(x,y) + in(x+1,y));  
out(x,y) = 1/3.f * (blurx(x,y-1) + blurx(x,y) + blurx(x,y+1));  
  
// execute pipeline on domain of size 1024x1024  
Halide::Buffer<uint8_t> result = out.realize(1024, 1024);
```



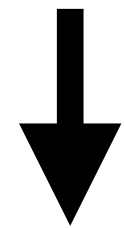
Real-world image processing pipelines feature complex sequences of functions

Benchmark	Number of Halide functions
Two-pass blur	2
Unsharp mask	9
Harris Corner detection	13
Camera RAW processing	30
Non-local means denoising	13
Max-brightness filter	9
Multi-scale interpolation	52
Local-laplacian filter	103
Synthetic depth-of-field	74
Bilateral filter	8
Histogram equalization	7
VGG-16 deep network eval	64

Real-world production applications may features hundreds to thousands of functions!
Google HDR+ pipeline: over 2000 Halide functions.

One (serial) implementation of Halide

```
Func blurx, out;  
Var x, y, xi, yi;  
Halide::Buffer<uint8_t> in = load_image("myimage.jpg");  
  
// the "algorithm description" (declaration of what to do)  
blurx(x,y) = (in(x-1, y) + in(x,y) + in(x+1,y)) / 3.0f;  
out(x,y)    = (blurx(x,y-1) + blurx(x,y) + blurx(x,y+1)) / 3.0f;  
  
// execute pipeline on domain of size 1024x1024  
Halide::Buffer<uint8_t> result = out.realize(1024, 1024);
```

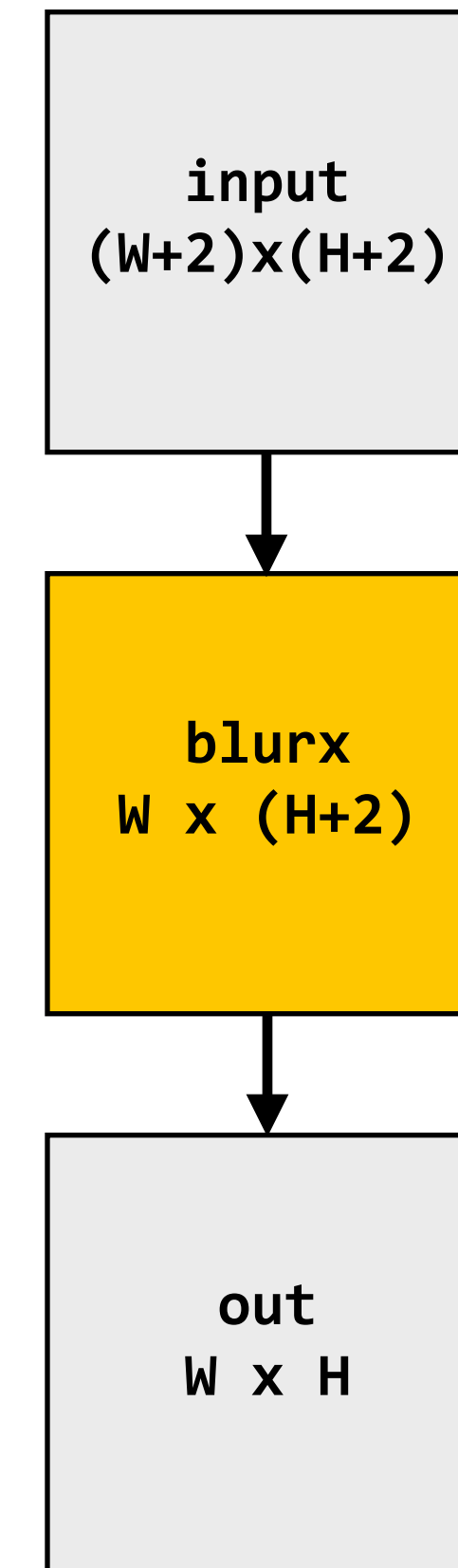


Equivalent "C-style" loop nest:

```
allocate in(1024+2, 1024+2); // (width,height)... initialize from image  
allocate blurx(1024,1024+2); // (width,height)  
allocate out(1024,1024); // (width,height)
```

```
for y=0 to 1024:  
  for x=0 to 1024+2:  
    blurx(x,y) = ... compute from in
```

```
for y=0 to 1024:  
  for x=0 to 1024:  
    out(x,y) = ... compute from blurx
```



Key aspect in the design of any system:

Choosing the “right” representations for the job

- **Good representations are productive to use:**
 - Embody the natural way of thinking about a problem
- **Good representations enable the system to provide the application useful services:**
 - Validating/providing certain guarantees (correctness, resource bounds, conservation of quantities, type checking)
 - Performance (parallelization, vectorization, use of specialized hardware)

Now the job is not expressing an image processing computation, but generating an efficient implementation of a specific Halide program.

A second set of representations for “scheduling”

```
Func blurx, out;  
Var x, y, xi, yi;  
Halide::Buffer<uint8_t> in = load_image("myimage.jpg");  
  
// the “algorithm description” (declaration of what to do)  
blurx(x,y) = (in(x-1, y) + in(x,y) + in(x+1,y)) / 3.0f;  
out(x,y)   = (blurx(x,y-1) + blurx(x,y) + blurx(x,y+1)) / 3.0f;
```

```
// “the schedule” (how to do it)  
out.tile(x, y, xi, yi, 256, 32).vectorize(xi,8).parallel(y);
```

```
blurx.compute_at(x).vectorize(x, 8);
```

Produce elements `blurx` on demand for each tile of output.

Vectorize the `x` (innermost) loop

When evaluating `out`, use 2D tiling order (loops named by `x, y, xi, yi`).
Use tile size 256 x 32.

Vectorize the `xi` loop (8-wide)

Use threads to parallelize the `y` loop

“Schedule”

```
// execute pipeline on domain of size 1024x1024  
Halide::Buffer<uint8_t> result = out.realize(1024, 1024);
```

Scheduling primitives allow the programmer to specify a high-level “sketch” of how to schedule the algorithm onto a parallel machine, but leave the details of emitting the low-level platform-specific code to the Halide compiler

Primitives for iterating over N-D domains

1	2	3	4	5	6
7	8	9	10	11	12
13	14	15	16	17	18
19	20	21	22	23	24
25	26	27	28	29	30
31	32	33	34	35	36

serial y, serial x

1	7	13	19	25	31
2	8	14	20	26	32
3	9	15	21	27	33
4	10	16	22	28	34
5	11	17	23	29	35
6	12	18	24	30	36

serial x, serial y

Specify both order and how to parallelize
(multi-thread, vectorize via SIMD instr)

1	2
3	4
5	6
7	8
9	10
11	12

serial y
vectorized x

1	2
1	2
1	2
1	2
1	2
1	2
1	2

parallel y
vectorized x

t0
t1

1	2	5	6	9	10
3	4	7	8	11	12
13	14	17	18	21	22
15	16	19	20	23	24
25	26	29	30	33	34
27	28	31	32	35	36

split x into $2x_0+x_i$,
split y into $2y_0+y_i$,
serial y_0, x_0, y_i, x_i

2D blocked iteration order

(In diagram, numbers indicate sequential order of processing within a thread)

Specifying loop iteration order and parallelism

```
blurx(x,y) = (in(x-1, y) + in(x,y) + in(x+1,y)) / 3.0f;  
out(x,y)   = (blurx(x,y-1) + blurx(x,y) + blurx(x,y+1)) / 3.0f;
```

Given this schedule for the function “out”...

```
out.tile(x, y, xi, yi, 256, 32).vectorize(xi,8).parallel(y);
```

Halide compiler will generate this parallel, vectorized loop nest for computing elements of out...

```
for y=0 to HEIGHT  
  for x=0 to WIDTH  
    blurx(x,y) = ...
```

```
for y=0 to num_tiles_y: // parallelize this loop with threads  
  for x=0 to num_tiles_x:  
    for yi=0 to 32:  
      for xi=0 to 256 by 8: // vectorize this loop with SIMD instr  
        idx_x = x*256+xi;  
        idx_y = y*32+yi  
        out(idx_x, idx_y) = ... (simd arithmetic here)
```

Primitives for how to interleave producer/consumer processing

```
blurx(x,y) = (in(x-1, y) + in(x,y) + in(x+1,y)) / 3.0f;  
out(x,y)   = (blurx(x,y-1) + blurx(x,y) + blurx(x,y+1)) / 3.0f;
```

```
out.tile(x, y, xi, yi, 256, 32);
```

```
blurx.compute_root();
```

**Do not compute blurx within out's loop nest.
Compute all of blurx, then all of out**

```
allocate buffer for all of blurx(x,y)  
for y=0 to HEIGHT:  
  for x=0 to WIDTH:  
    blurx(x,y) = ...
```

all of blurx is computed here

```
for y=0 to num_tiles_y:  
  for x=0 to num_tiles_x:  
    for yi=0 to 32:  
      for xi=0 to 256:  
        idx_x = x*256+xi;  
        idx_y = y*32+yi  
        out(idx_x, idx_y) = ...
```

values of blurx consumed here

Primitives for how to interleave producer/consumer processing

```
blurx(x,y) = (in(x-1, y) + in(x,y) + in(x+1,y)) / 3.0f;  
out(x,y)   = (blurx(x,y-1) + blurx(x,y) + blurx(x,y+1)) / 3.0f;
```

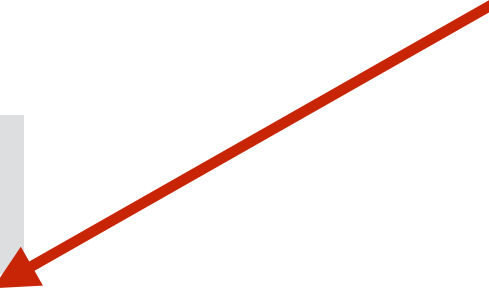
```
out.tile(x, y, xi, yi, 256, 32);
```

```
blurx.compute_at(out, xi);
```

Compute necessary elements of blurx
within out's xi loop nest

```
for y=0 to num_tiles_y:  
  for x=0 to num_tiles_x:  
    for yi=0 to 32:  
      for xi=0 to 256:  
        idx_x = x*256+xi;  
        idx_y = y*32+yi
```

Note: Halide compiler performs
analysis that the output of each
iteration of the xi loop required 3
elements of blurx



```
allocate 3-element buffer for tmp_blurx
```

```
// compute 3 elements of blurx needed for out(idx_x, idx_y) here  
for (blur_x=0 to 3)  
  tmp_blurx(blur_x) = ...
```

```
out(idx_x, idx_y) = ...
```


Primitives for how to interleave producer/consumer processing

```
blurx(x,y) = (in(x-1, y) + in(x,y) + in(x+1,y)) / 3.0f;  
out(x,y)   = (blurx(x,y-1) + blurx(x,y) + blurx(x,y+1)) / 3.0f;
```

```
out.tile(x, y, xi, yi, 256, 32);
```

```
blurx.compute_at(out, x);
```

Compute necessary elements of blurx within out's x loop nest (all necessary elements for one tile of out)

```
for y=0 to num_tiles_y:  
  for x=0 to num_tiles_x:
```

```
    allocate 258x34 buffer for tile blurx
```

```
    for yi=0 to 32+2:
```

```
      for xi=0 to 256+2:
```

```
        tmp_blurx(xi,yi) = // compute blurx from in
```

tile of blurx is
computed here

```
    for yi=0 to 32:
```

```
      for xi=0 to 256:
```

```
        idx_x = x*256+xi;
```

```
        idx_y = y*32+yi
```

```
        out(idx_x, idx_y) = ...
```

tile of blurx is consumed here

Summary of scheduling the 3x3 box blur

```
// the "algorithm description" (declaration of what to do)
blurx(x,y) = (in(x-1, y) + in(x,y) + in(x+1,y)) / 3.0f;
out(x,y)    = (blurx(x,y-1) + blurx(x,y) + blurx(x,y+1)) / 3.0f;

// "the schedule" (how to do it)
out.tile(x, y, xi, yi, 256, 32).vectorize(xi,8).parallel(y);
blurx.compute_at(out, x).vectorize(x, 8);
```

Equivalent parallel loop nest:

```
for y=0 to num_tiles_y: // iters of this loop are parallelized using threads
  for x=0 to num_tiles_x:
    allocate 258x34 buffer for tile blurx
    for yi=0 to 32+2:
      for xi=0 to 256+2 BY 8:
        tmp_blurx(xi,yi) = ... // compute blurx from in using 8-wide
                               // SIMD instructions here
                               // compiler generates boundary conditions
                               // since 256+2 isn't evenly divided by 8

    for yi=0 to 32:
      for xi=0 to 256 BY 8:
        idx_x = x*256+xi;
        idx_y = y*32+yi
        out(idx_x, idx_y) = ... // compute out from blurx using 8-wide
                                // SIMD instructions here
```

What is the philosophy of Halide

- **Programmer** is responsible for describing an image processing algorithm
- **Programmer** has knowledge of how to schedule the application efficiently on machine (but it's slow and tedious), so Halide gives programmer a language to express high-level scheduling decisions
 - Loop structure of code
 - Unrolling / vectorization / multi-core parallelization
- **The system** (Halide compiler) is not smart, it provides the service of mechanically carrying out the details of the schedule in terms of mechanisms available on the target machine (pthreads, AVX intrinsics, etc.)

Constraints on language

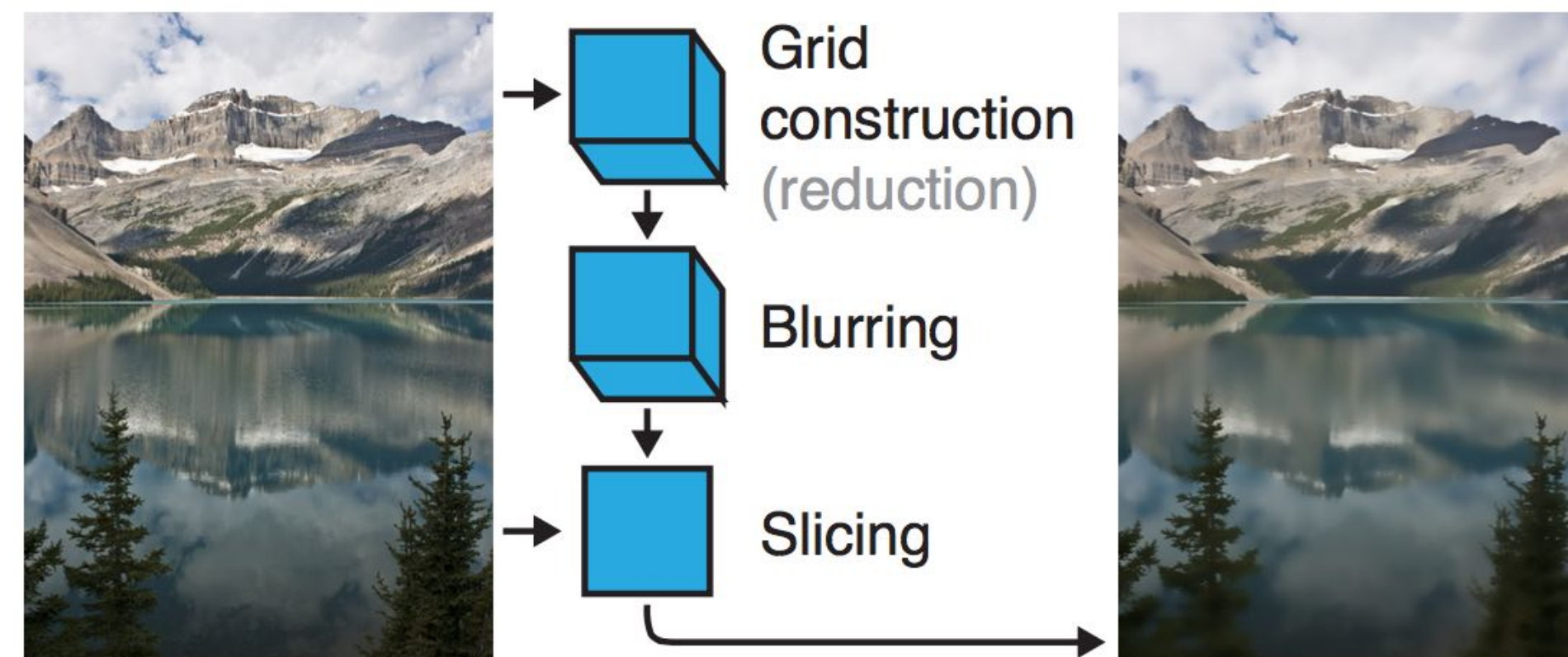
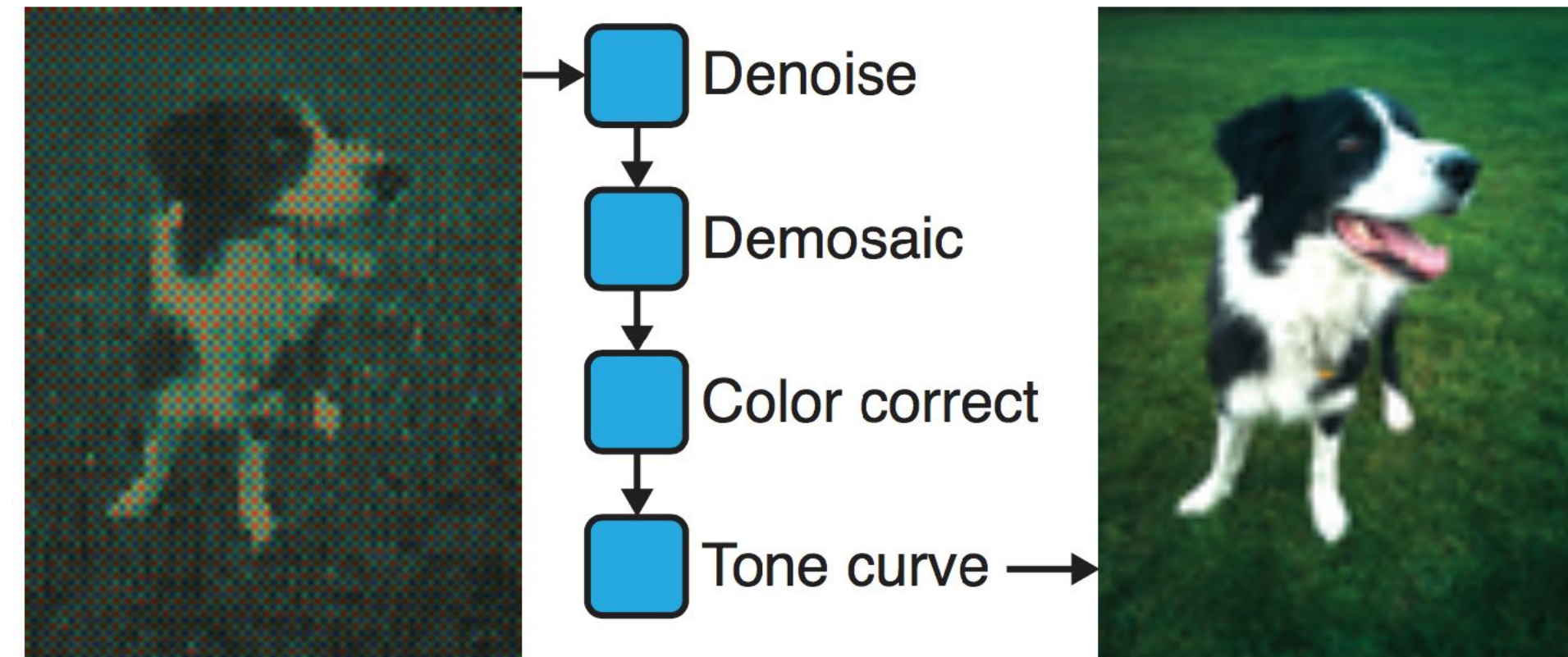
(to enable compiler to provide desired services)

- **Application domain scope: computation on regular N-D domains**
- **Only feed-forward pipelines (includes special support for reductions and fixed recursion depth)**
- **All dependencies inferable by compiler**

Initial academic Halide results

[Ragan-Kelley 2012]

- **Application 1: camera RAW processing pipeline**
(Convert RAW sensor data to RGB image)
 - **Original: 463 lines of hand-tuned ARM NEON assembly**
 - **Halide: 2.75x less code, 5% faster**
- **Application 2: bilateral filter**
(Common image filtering operation used in many applications)
 - **Original 122 lines of C++**
 - **Halide: 34 lines algorithm + 6 lines schedule**
 - **CPU implementation: 5.9x faster**
 - **GPU implementation: 2x faster than hand-written CUDA**



Stepping back: what is Halide?

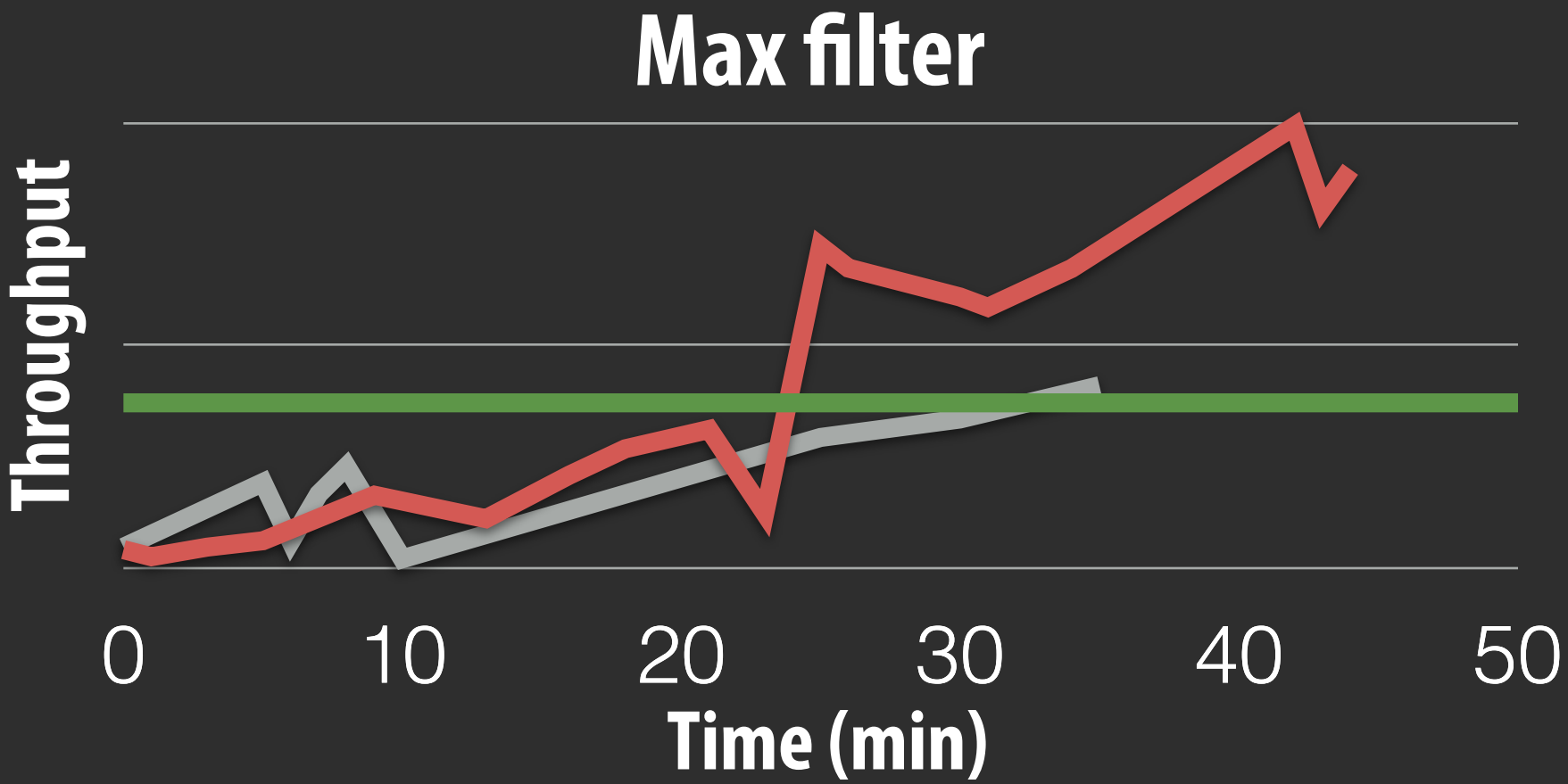
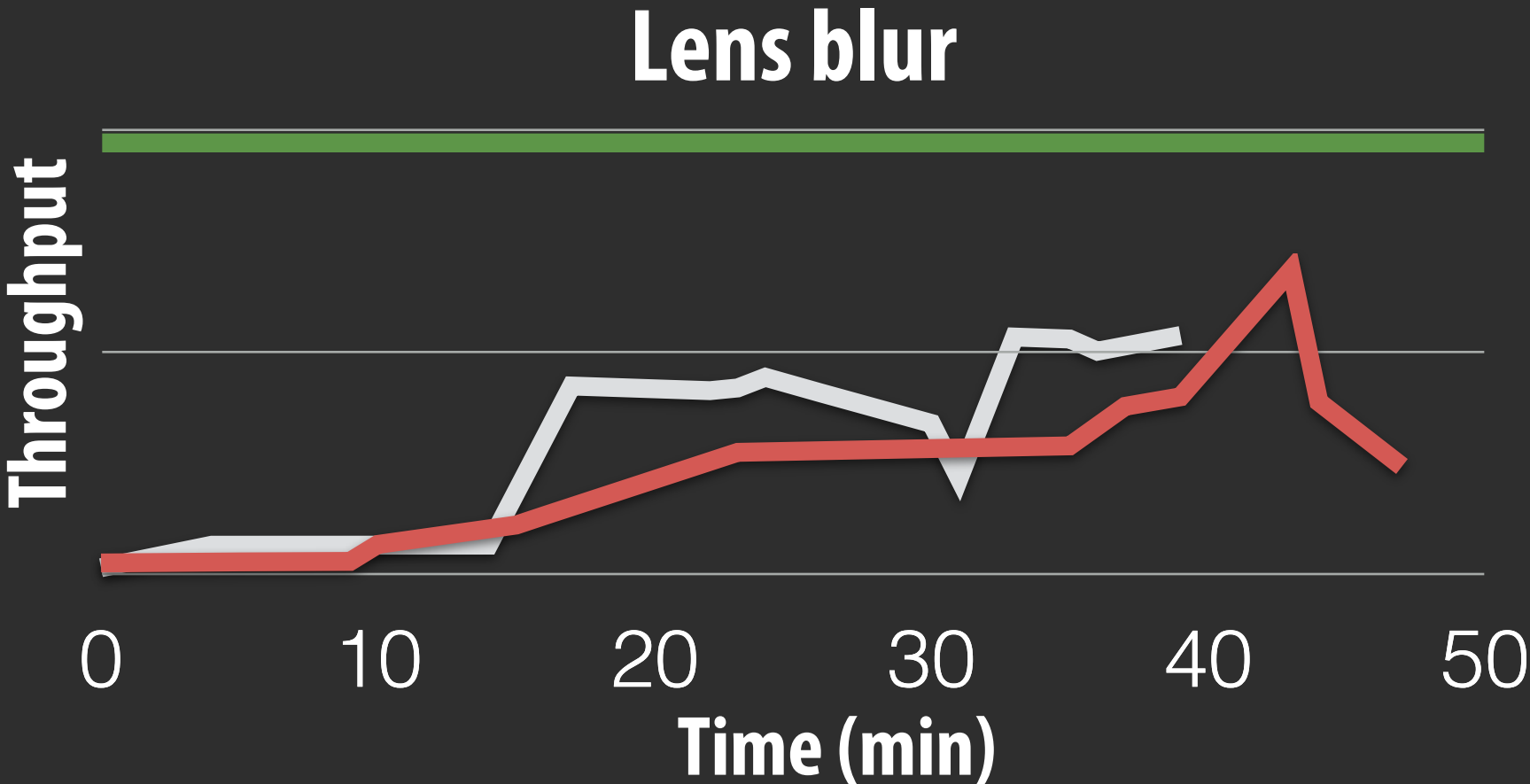
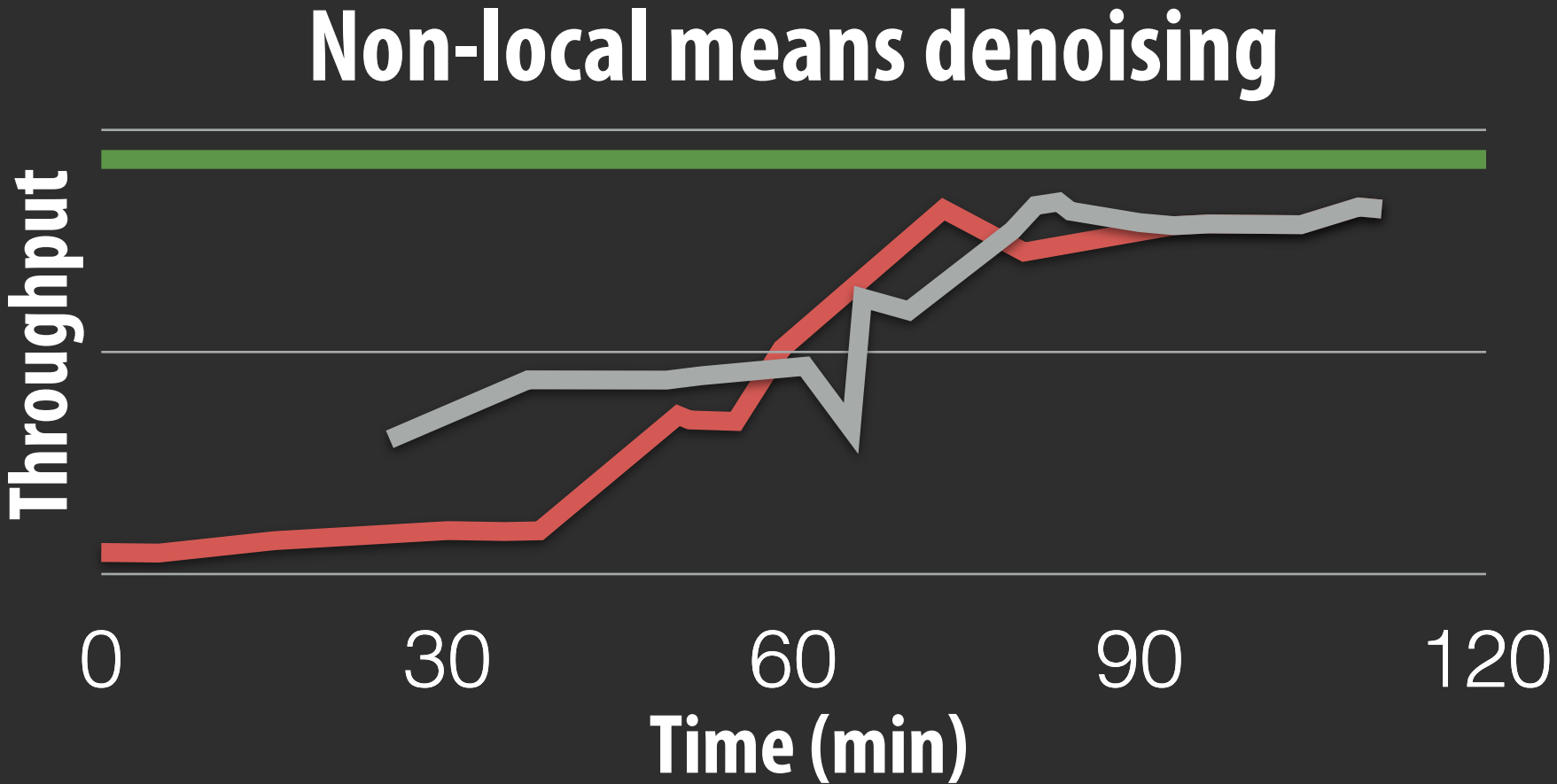
- **Halide is a DSL for helping expert developers optimize image processing code more rapidly**
 - **Halide does not decide how to optimize a program for a novice programmer**
 - **Halide provides primitives for a programmer (that has strong knowledge of code optimization) to rapidly express what optimizations the system should apply**
 - **Halide compiler carries out the nitty-gritty of mapping that strategy to a machine**

Automatically generating Halide schedules

- **Problem: it turned out that very few programmers have the ability to write good Halide schedules**
 - 80+ programmers at Google write Halide
 - Very small number trusted to write schedules
- **Recent work: compiler analyzes the Halide program to automatically generate efficient schedules for the programmer [Adams 2019]**
 - As of [Adams 2019], you'd have to work pretty hard to manually author a schedule that is better than the schedule generated by the Halide autoscheduler for image processing applications

Autoscheduler saves time for experts

Early results from [Mullapudi 2016]



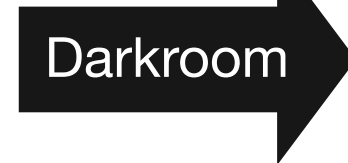
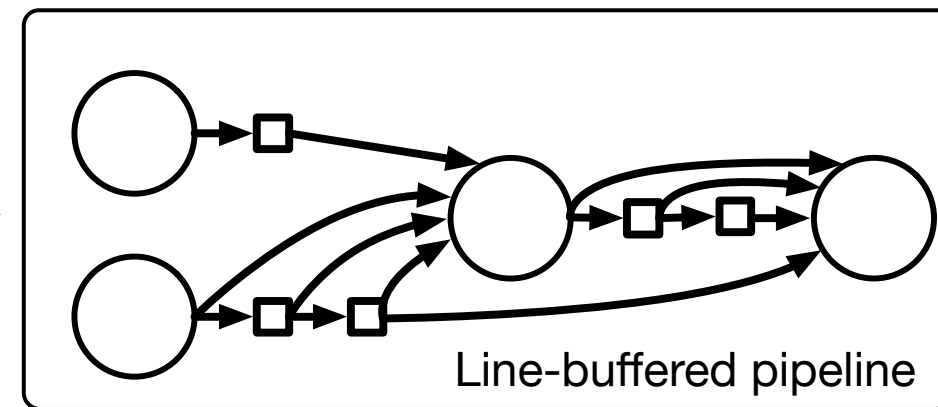
- Auto scheduler
- Dillon
- Andrew

Darkroom/Rigel/Aetherling

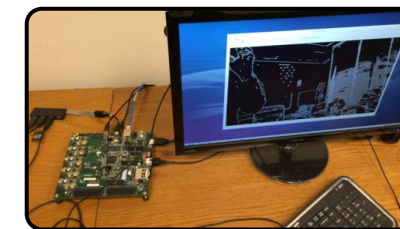
[Hegarty 2014, Hegarty 2016, Durst 2020]

Goal: directly synthesize ASIC or FPGA implementation of image processing pipelines from a high-level algorithm description (a constrained “Halide-like” language)

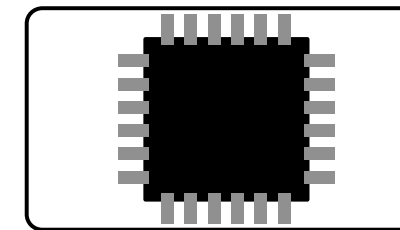
```
bx = im(x,y)
  (I(x-1,y) +
   I(x,y) +
   I(x+1,y))/3
end
by = im(x,y)
  (bx(x,y-1) +
   bx(x,y) +
   bx(x,y+1))/3
end
sharpened = im(x,y)
  I(x,y) + 0.1*
  (I(x,y) - by(x,y))
end
Stencil Language
```



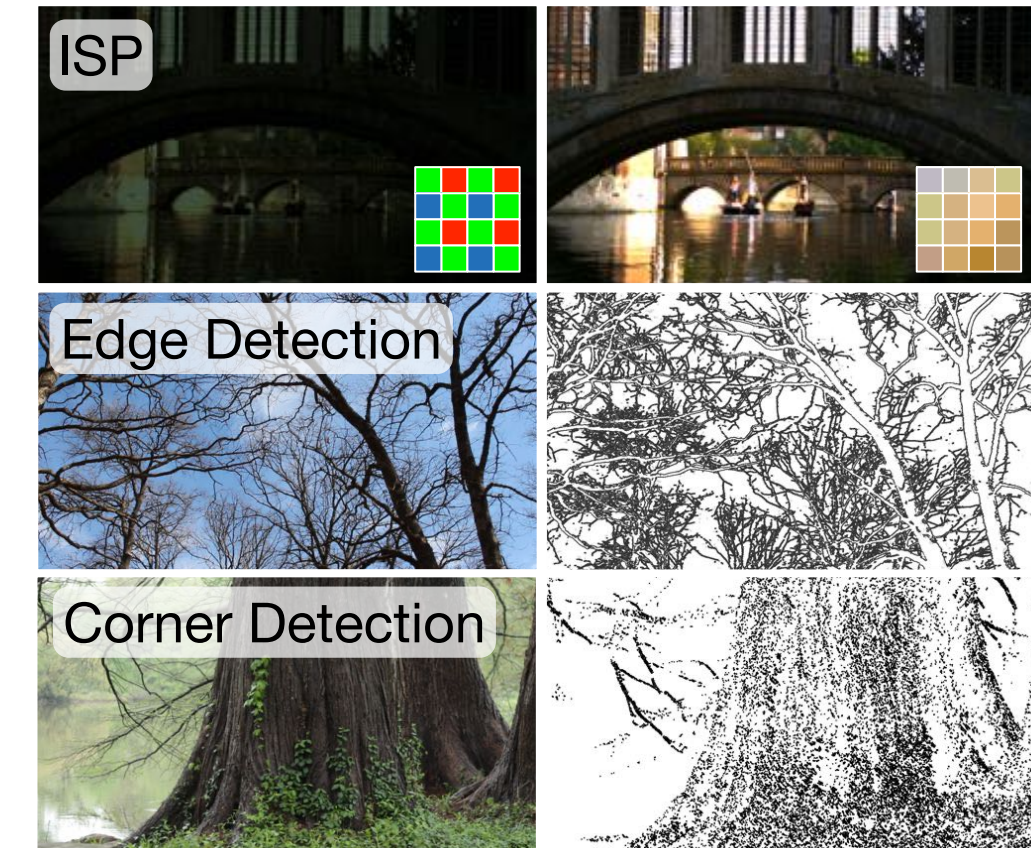
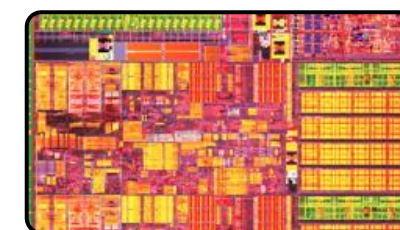
FPGA



ASIC



CPU



Goal: very-high efficiency image processing

Many other recent domain-specific programming systems



Less domain specific than examples given today,
but still designed specifically for:
data-parallel computations on big data for
distributed systems (“Map-Reduce”)



DSL for graph-based machine learning computations
Also see Ligra
(DSLs for describing operations on graphs)



Model-view-controller paradigm for
web-applications



DSL for defining deep neural
networks and training/inference
computations on those networks



Language for real-time 3D graphics



Numerical computing

Ongoing efforts in many domains...

Languages for physical simulation: Simit [MIT], Ebb [Stanford]

Opt: a language for non-linear least squares optimization [Stanford]

Summary

- **Modern machines: parallel and heterogeneous**
 - Only way to increase compute capability in energy-constrained world
- **Most software uses small fraction of peak capability of machine**
 - Very challenging to tune programs to these machines
 - Tuning efforts are not portable across machines
- **Domain-specific programming environments trade-off generality to achieve productivity, performance, and portability**
 - Case study today: Halide
 - Leverage explicit dependencies, domain restrictions, domain knowledge for system to synthesize efficient implementations