

Dusk and dawn: dark silicon and the new golden age of computer architecture (2)

Hung-Wei Tseng

Recap: Power consumption & power density

$$\cdot P_{dynamic} \sim \alpha \times C \times V^2 \times f \times N$$

- α : average switches per cycle

- C : capacitance

- V : voltage

- f : frequency, usually linear with V

- N : the number of transistors

$$\cdot P_{leakage} \sim N \times V \times e^{-V}$$

- N : number of transistors

- V : voltage

- V_t : threshold voltage where transistor conducts (begins to switch)

- Power density:

$$P_{density} = \frac{P}{area}$$

Dennard scaling discontinued —
we cannot make voltage lower

Moore's Law allows higher frequencies as transistors are smaller
Moore's Law makes this smaller

Recap: Dark silicon problem

- We are given the same area, twice amount of transistors
- We are given the same power budget
- We are given a certain applications
- How can we maximize the performance for these applications within the given constraints?

Given the same power budget, maximize the efficiency per chip

Slowdown all of them some are not functioning

=50W!

=50W!

Chip

1	1	1	1	1
1	1	1	1	1
1	1	1	1	1
1	1	1	1	1
1	1	1	1	1
1	1	1	1	1
1	1	1	1	1
1	1	1	1	1
1	1	1	1	1
1	1	1	1	1

On ~

50W

1	1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1	1

Off ~

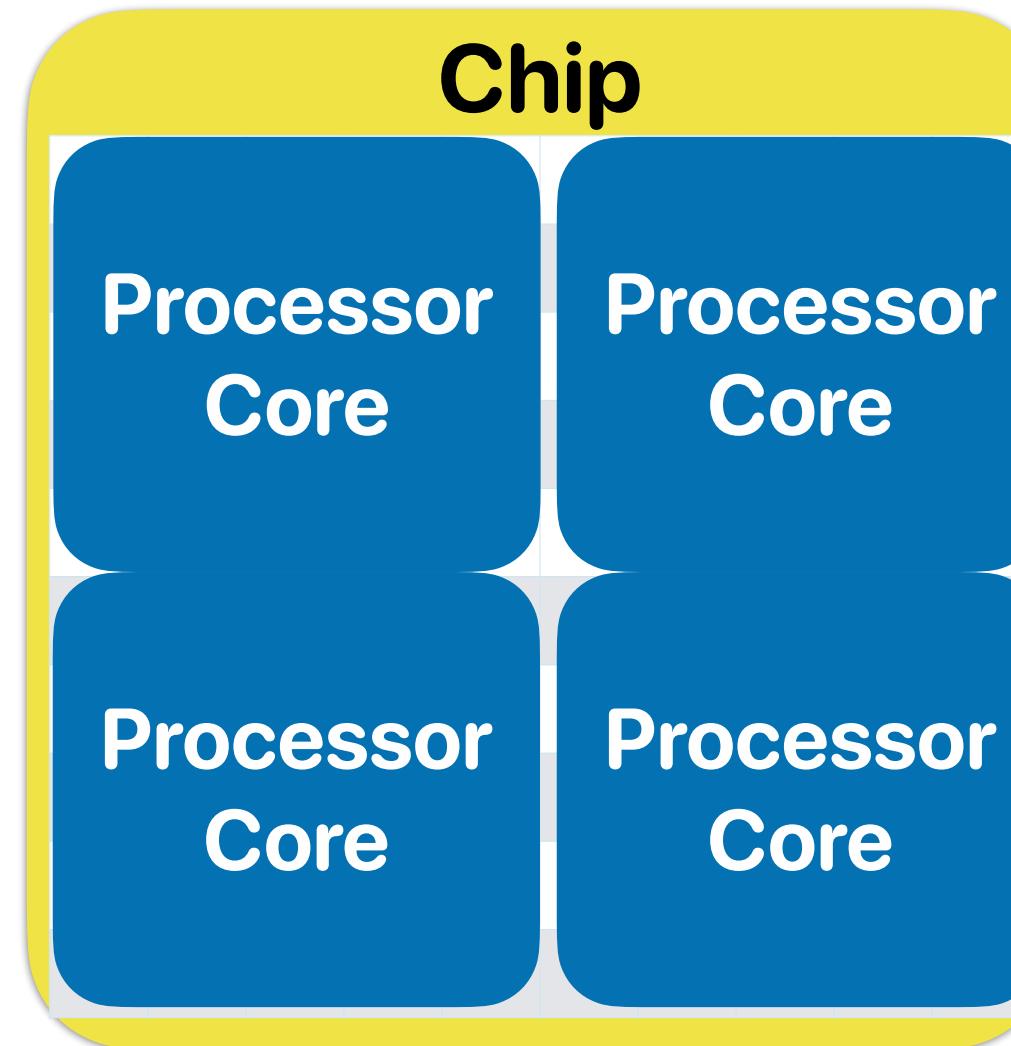
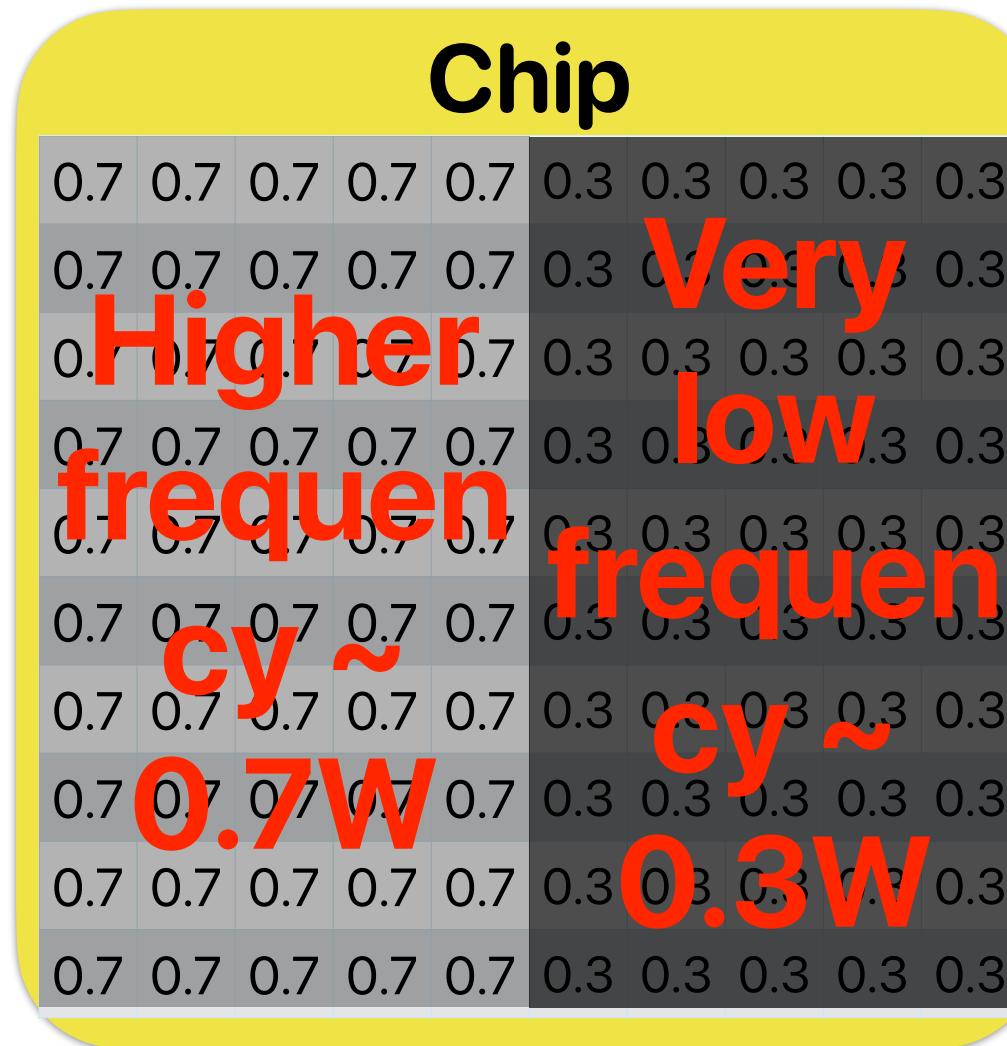
OW

Dark!

=50W!

Given the same power budget, maximize the efficiency per chip

**Some faster,
some slower**



Aggressively adjust the frequency of processor cores — If only one program is compute-intensive, having it running at full speed, others at lower frequency or turning off

Given the same power budget, maximize the efficiency per chip

Slowdown all of them

Chip

0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5
0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5
0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5
0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5
0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5
0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5
0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5
0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5
0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5
0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5

Low frequency
~0.5W

Chip

Processor Core				
Processor Core				
Processor Core				
Processor Core				
Processor Core				
Processor Core				
Processor Core				
Processor Core				
Processor Core				
Processor Core				

All of them are
smaller, simpler,
lower-frequency,
lower-power cores

=50W!

Take-aways: Challenges and SOTA solutions in the dark silicon era

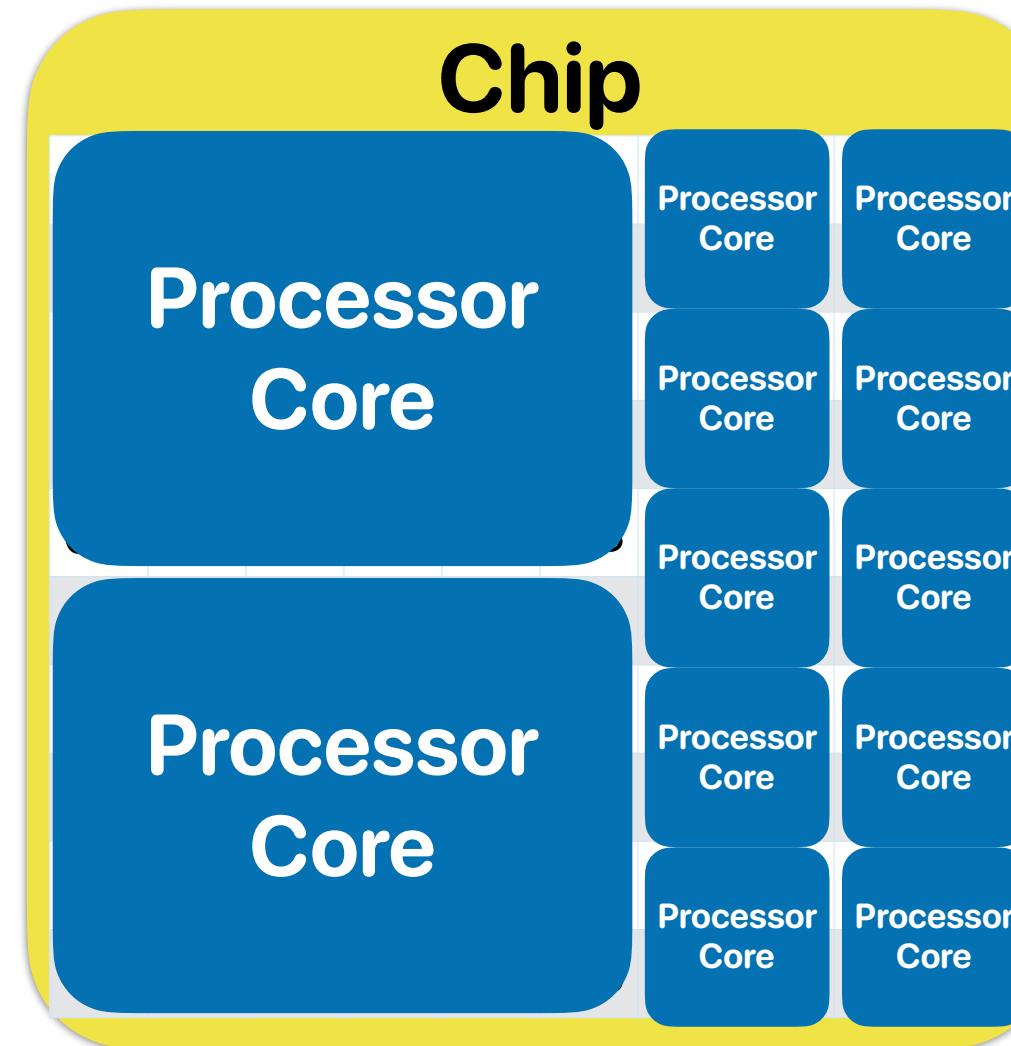
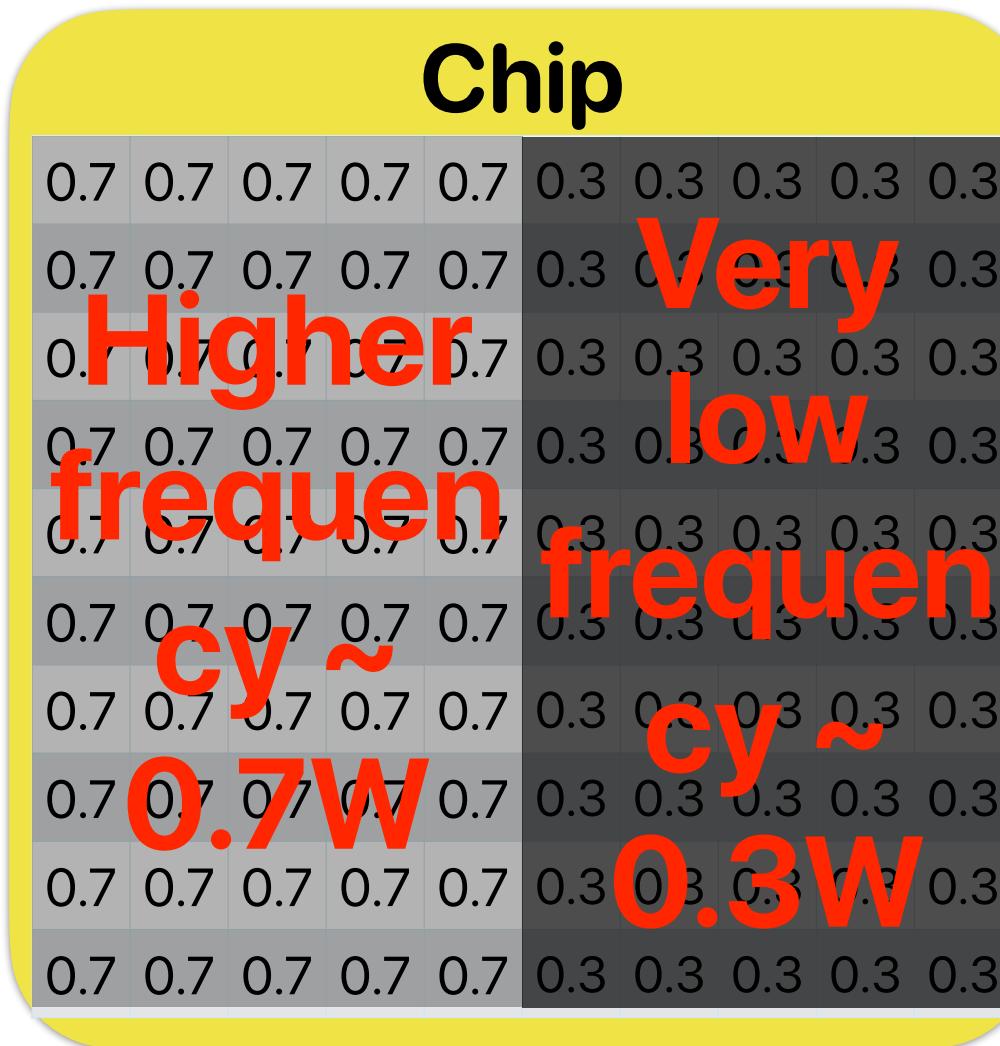
- Even if we can address all programming challenges, multi-core performance has stopped to scale due to the Dark Silicon problem
- Aggressive dynamic frequency/voltage scaling on CMP to accommodate the demand of **latency**-sensitive, parallelism-limited applications, but the area-efficiency of the slower cores is not great
- GPUs/many-core processors improve the **throughput** per-chip through providing massive parallelism where each processing element operates at a lower speed, but not-ideal for latency-sensitive workloads

Outline

- Challenges and state-of-the-art solutions in the dark silicon era (cont.)
- The new golden age of Computer Architecture

Given the same power budget, maximize the efficiency per chip

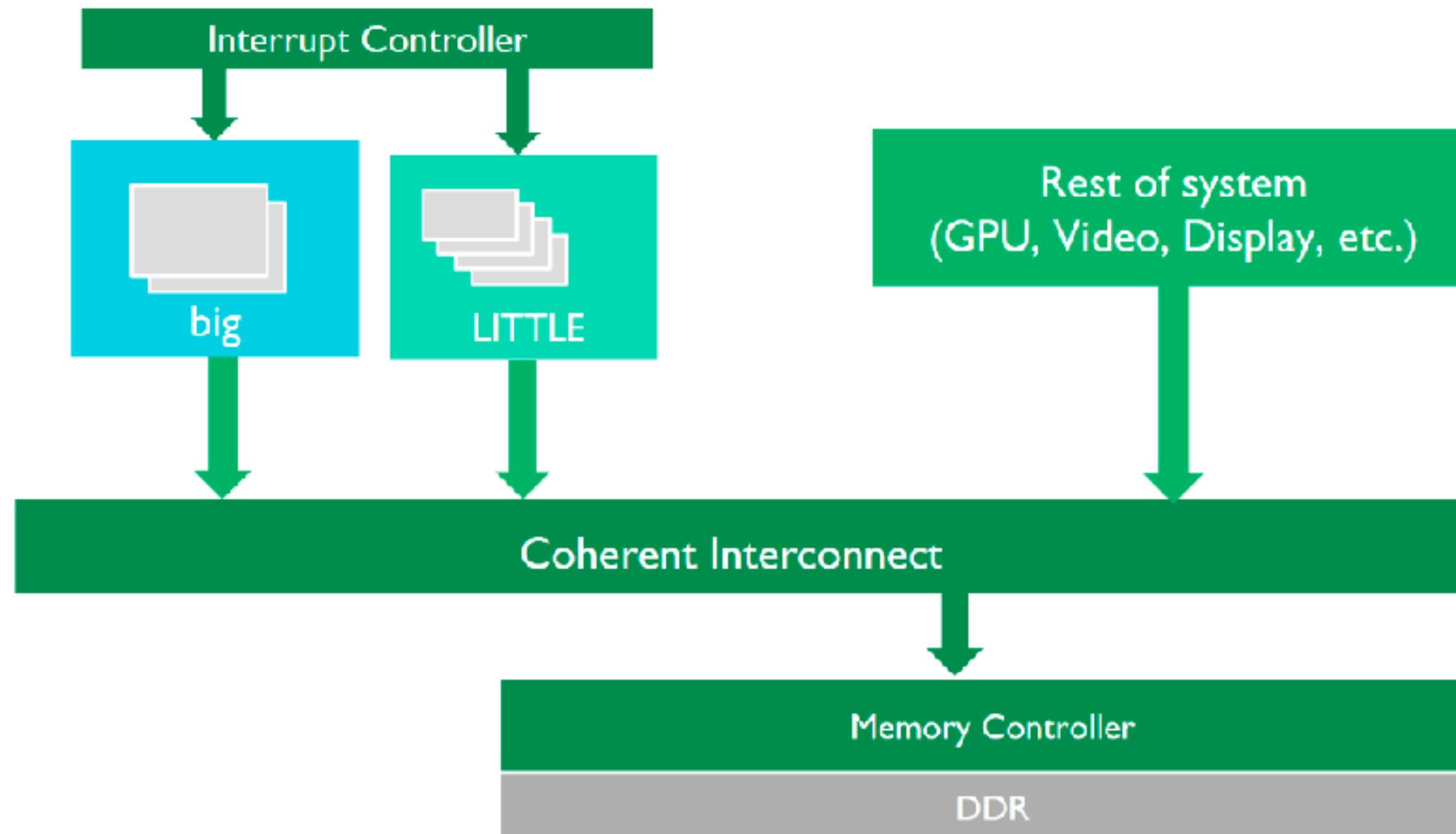
Some faster,
some slower



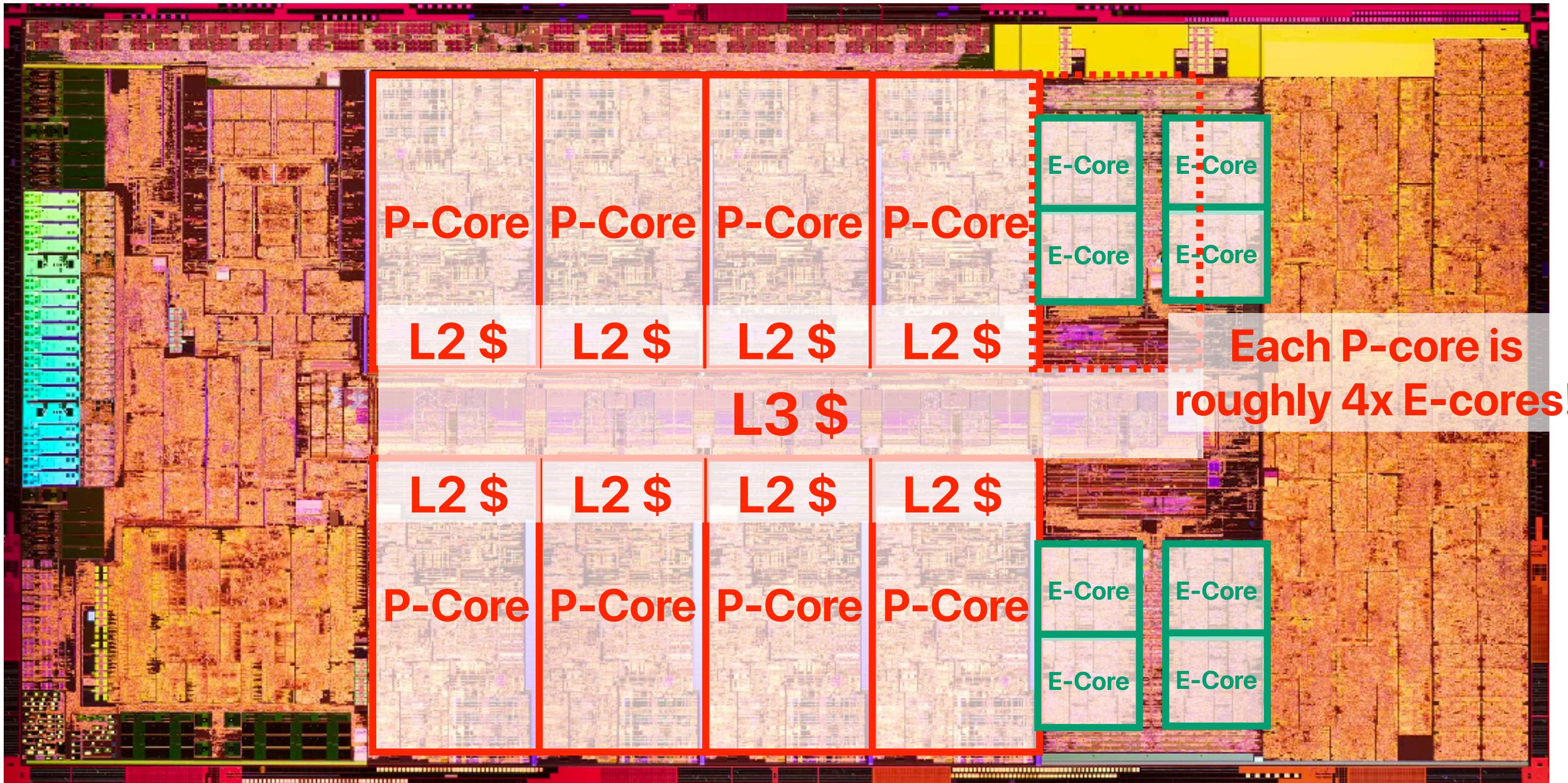
Some powerful cores for latency sensitive workloads, many small cores for highly parallelizable workloads

Single ISA heterogeneous CMP in ARM's big.LITTLE architecture

big.LITTLE system



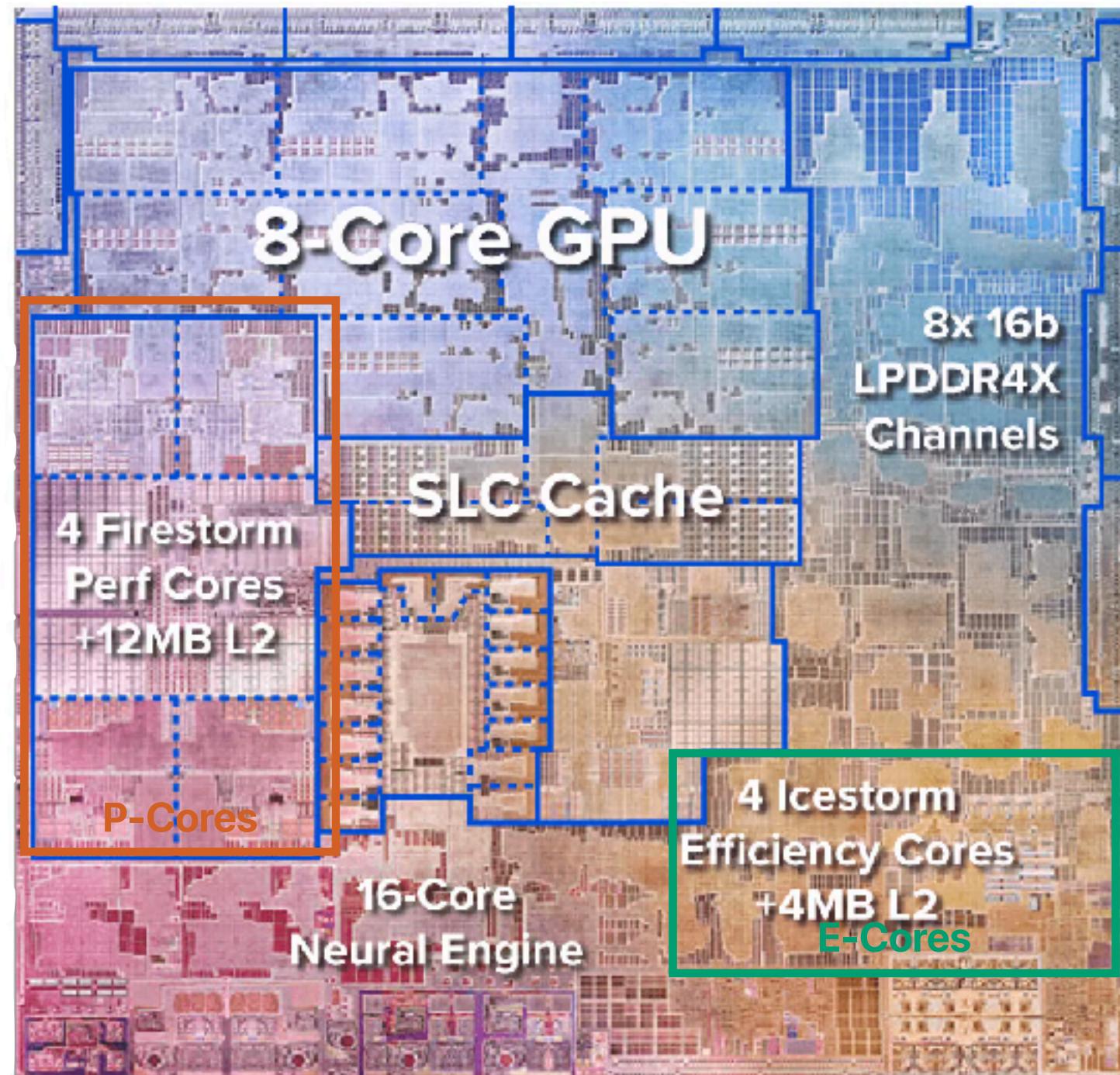
Intel Alder Lake



Single ISA heterogeneous CMP in Intel Processors



Single ISA heterogeneous CMP in Apple's M1





Single ISA heterogeneous CMP (big.Little)

- Assume we have two processor core architectures and the potential configurations within the same chip area:

	Big Only	Little Only	big.Little
P-Cores: higher performance, but higher power consumption and larger size	4		2
E-Cores: lower performance, but lower power consumption and smaller size		24	12

Please identify the correct expectations of their performance

- ① For a program with limited thread-level parallelism, **big.Little** would deliver better than or at least the same level of performance as **Little Only**
- ② For a program with limited thread-level parallelism, **big.Little** would deliver better than or at least the same level of performance as **Big Only**
- ③ For a program with rich thread-level parallelism, **big.Little** would deliver better or at least the same level of performance than **Little Only**
- ④ For a program with rich thread-level parallelism, **big.Little** would deliver better or at least the same level of performance than **big Only**

- A. 0
- B. 1
- C. 2
- D. 3
- E. 4

Single ISA heterogeneous CMP (big.Little)

- Assume we have two processor core architectures and the potential configurations within the same chip area:

	Big Only	Little Only	big.Little
P-Cores: higher performance, but higher power consumption and larger size	4		2
E-Cores: lower performance, but lower power consumption and smaller size		24	12

Please identify the correct expectations of their performance

- For a program with limited thread-level parallelism, **big.Little** would deliver better than or at least the same level of performance as **Little Only**
- ② For a program with limited thread-level parallelism, **big.Little** would deliver better than or at least the same level of performance as **Big Only**
- ③ For a program with rich thread-level parallelism, **big.Little** would deliver better or at least the same level of performance than **Little Only**
- For a program with rich thread-level parallelism, **big.Little** would deliver better or at least the same level of performance than **big Only**
 - A. 0
 - B. 1
 - C. 2
 - D. 3
 - E. 4

Take-aways: Challenges and SOTA solutions in the dark silicon era

- Even if we can address all programming challenges, multi-core performance has stopped to scale due to the Dark Silicon problem
- Aggressive dynamic frequency/voltage scaling on CMP to accommodate the demand of **latency**-sensitive, parallelism-limited applications, but the area-efficiency of the slower cores is not great
- GPUs/many-core processors improve the **throughput** per-chip through providing massive parallelism where each processing element operates at a lower speed, but not-ideal for latency-sensitive workloads
- Single ISA, heterogeneous CMPs (e.g., big.Little cores, Intel/Apple's P-cores/E-cores) find a balance the trade-offs of general-purpose workloads, but won't be ideal if your applications go to either extreme of throughput (massive parallelism) or latency

A New “Golden” Age of Architects



Hardware Acceleration Solutions

- Using hardware accelerations, GPUs, Field Programmable Gate Arrays (FPGAs) and Domain-specific accelerators (DSAs), are increasingly popular to help tackle the demand of applications. Assume we have **optimized** implementations on each hardware acceleration options when running the same application, please identify how many of them are correct.
 - ① FPGAs typically can outperform GPUs and offer better performance-per-watt as FPGAs can realize more efficient logics using their internal look-up-tables
 - ② DSAs offer the best performance-per-watt than GPUs and FPGAs
 - ③ DSAs has the worst programmability compared to GPUs and FPGAs
 - ④ DSAs are more likely to be memory-bounded than others

A. 0
B. 1
C. 2
D. 3
E. 4



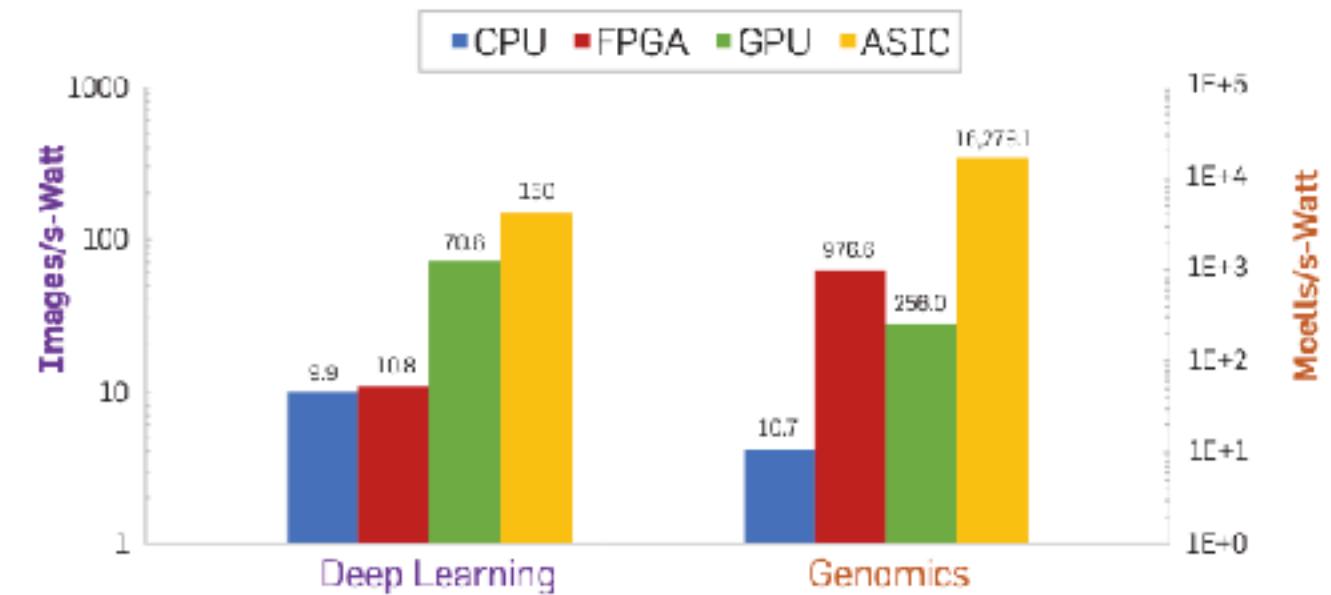
Hardware Acceleration Solutions

- Using hardware accelerations, GPUs, Field Programmable Gate Arrays (FPGAs) and Domain-specific accelerators (DSAs), are increasingly popular to help tackle the demand of applications. Assume we have **optimized** implementations on each hardware acceleration options when running the same application, please identify how many of them are correct.
 - ① FPGAs typically can outperform GPUs and offer better performance-per-watt as FPGAs can realize more efficient logics using their internal look-up-tables
 - ② DSAs offer the best performance-per-watt than GPUs and FPGAs
 - ③ DSAs has the worst programmability compared to GPUs and FPGAs
 - ④ DSAs are more likely to be memory-bounded than others
- A. 0
B. 1
C. 2
D. 3
E. 4

“Acceleration” Options

- ① FPGAs does not outperform GPU all the time
- ② DSAs offer the best performance-per-watt than GPUs and FPGAs
- ③ DSAs has the worst programmability compared to GPUs and FPGAs

Figure 2. Comparison of computation efficiency (in Tasks/s-Watt) for CPU, FPGA, GPU, and ASIC for deep learning and genomics domains.^a



^a Deep learning efficiency is measured in terms of inference images/s-Watt for Resnet-50²² (sources: CPU,¹³ FPGA,⁵³ GPU,³⁶ ASIC¹⁵). Genomics efficiency is measured in Moells/s-Watt for Banded Smith-Waterman algorithm (sources: CPU,⁸ FPGA,⁵⁰ GPU (our implementation on NVIDIA V100), ASIC⁵⁰).

A Cloud-Scale Acceleration Architecture

Adrian Caulfield, Eric Chung, Andrew Putnam, Hari Angepat, Jeremy Fowers, Michael Haselman, Stephen Heil, Matt Humphrey, Puneet Kaur, Joo-Young Kim, Daniel Lo, Todd Massengill, Kalin Ovtcharov, Michael Papamichael, Lisa Woods, Sitaram Lanka, Derek Chiou, Doug Burger
Microsoft

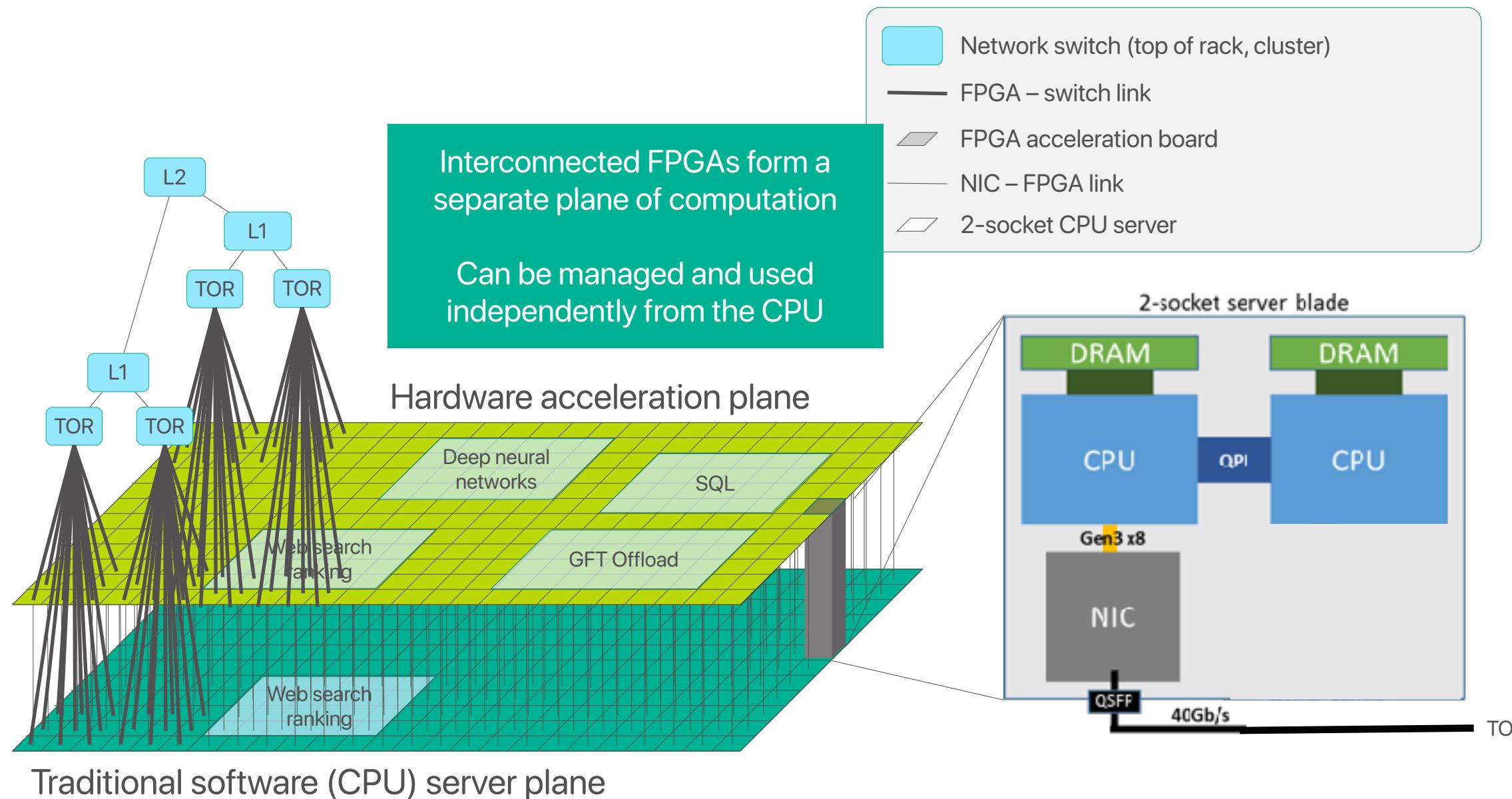
FPGA

- Field Programmable Gate Array
 - An array of “Lookup tables (LUTs)”
 - Reconfigurable wires or say interconnects of LUTs
 - Registers
- An LUT
 - Accepts a few inputs
 - Has SRAM memory cells that store all possible outputs
 - Generates outputs according to the given inputs
- As a result, you may use FPGAs to emulate any kind of gates or logic combinations, and create an ASIC-like processor



FPGA

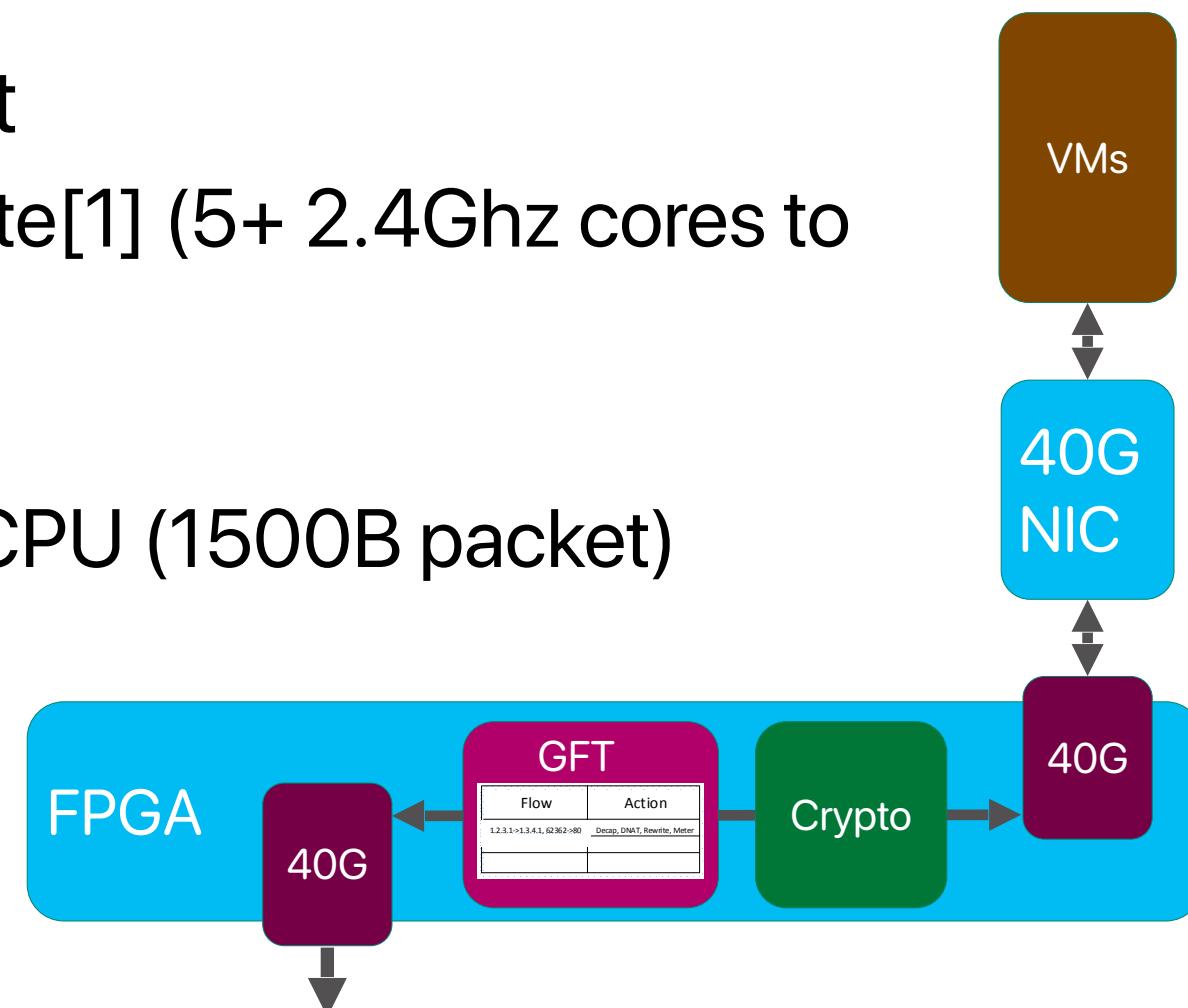
Configurable cloud



Accelerated networking

- Software defined networking
 - Generic Flow Table (GFT) rule based packet processing
 - 10x latency reduction vs software, CPU load reduction
 - 25Gb/s throughput at 25 μ s latency – the fastest cloud network
- Capable of 40 Gb line rate encrypt and decrypt
 - On Haswell, AES GCM-128 costs 1.26 cycles/byte[1] (5+ 2.4Ghz cores to sustain 40Gb/s)
 - CBC and other algorithms are more expensive
 - AES CBC-128-SHA1 is 11 μ s in FPGA vs 4 μ s on CPU (1500B packet)
 - Higher latency, but significant CPU savings

Our FPGA implementation supports full 40 Gb/s encryption and decryption. The worst case half-duplex FPGA crypto latency for AES-CBC-128-SHA1 is 11 μ s for a 1500B packet, from first flit to first flit. In software, based on the Intel numbers, it is approximately 4 μ s. AES-CBC-SHA1 is, however,



Why FPGA?

This model offers significant **flexibility**. From the local perspective, the FPPGA is used as a compute or a network accelerator. From the global perspective, the FPGAs can be managed as a large-scale pool of resources, with acceleration

These programmable architectures allow for hardware homogeneity while allowing fungibility via software for different services. They must be highly **flexible** at the system level, to

hyperscale infrastructure. The acceleration system we describe is sufficiently **flexible** to cover three scenarios: local compute acceleration (through PCIe), network acceleration, and global application acceleration, through configuration as pools of remotely accessible FPGAs. Local acceleration handles high-

Flexible

This paper described Configurable Clouds, a datacenter-scale acceleration architecture, based on FPGAs, that is both scalable and **flexible**. By putting in FPGA cards both in I/O

In addition to architectural requirements that provide sufficient **flexibility** to justify scale production deployment, there are also physical restrictions in current infrastructures that

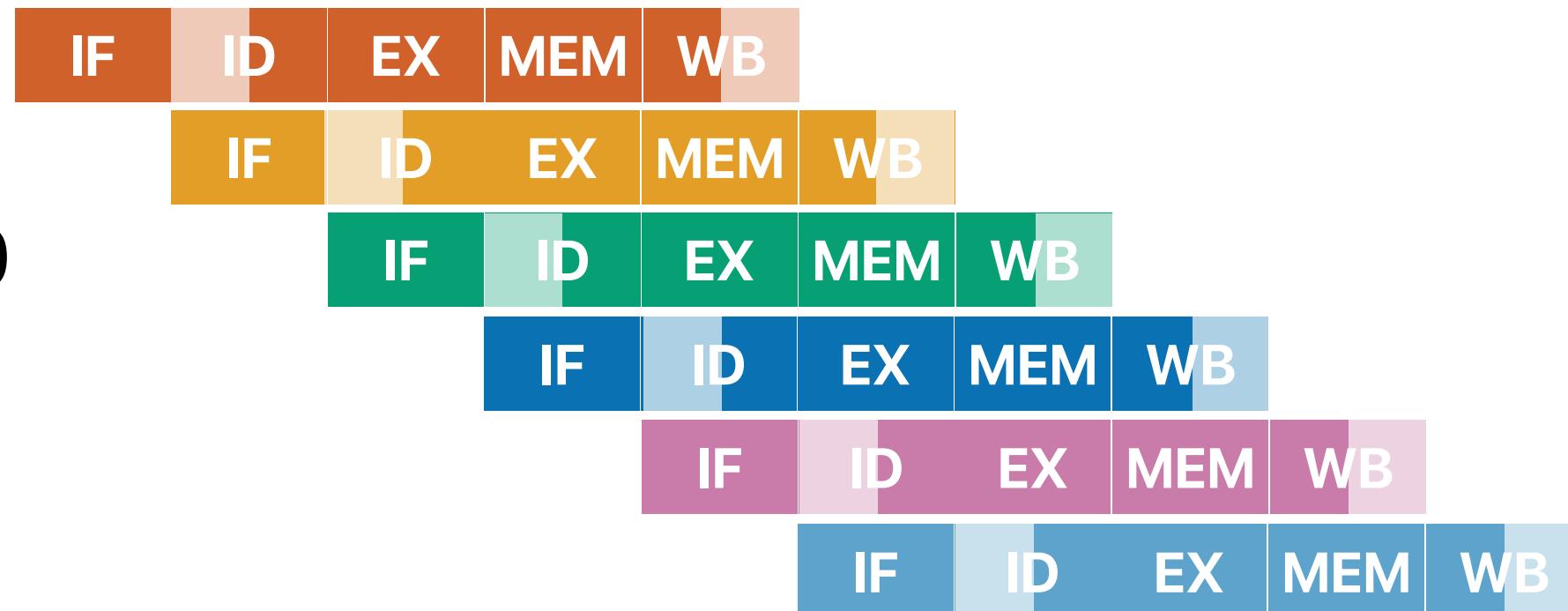
Domain Specific Hardware Accelerators

William J. Dally, Yatish Turakhia, and Song Han.
NVIDIA, UC Santa Cruz, and MIT

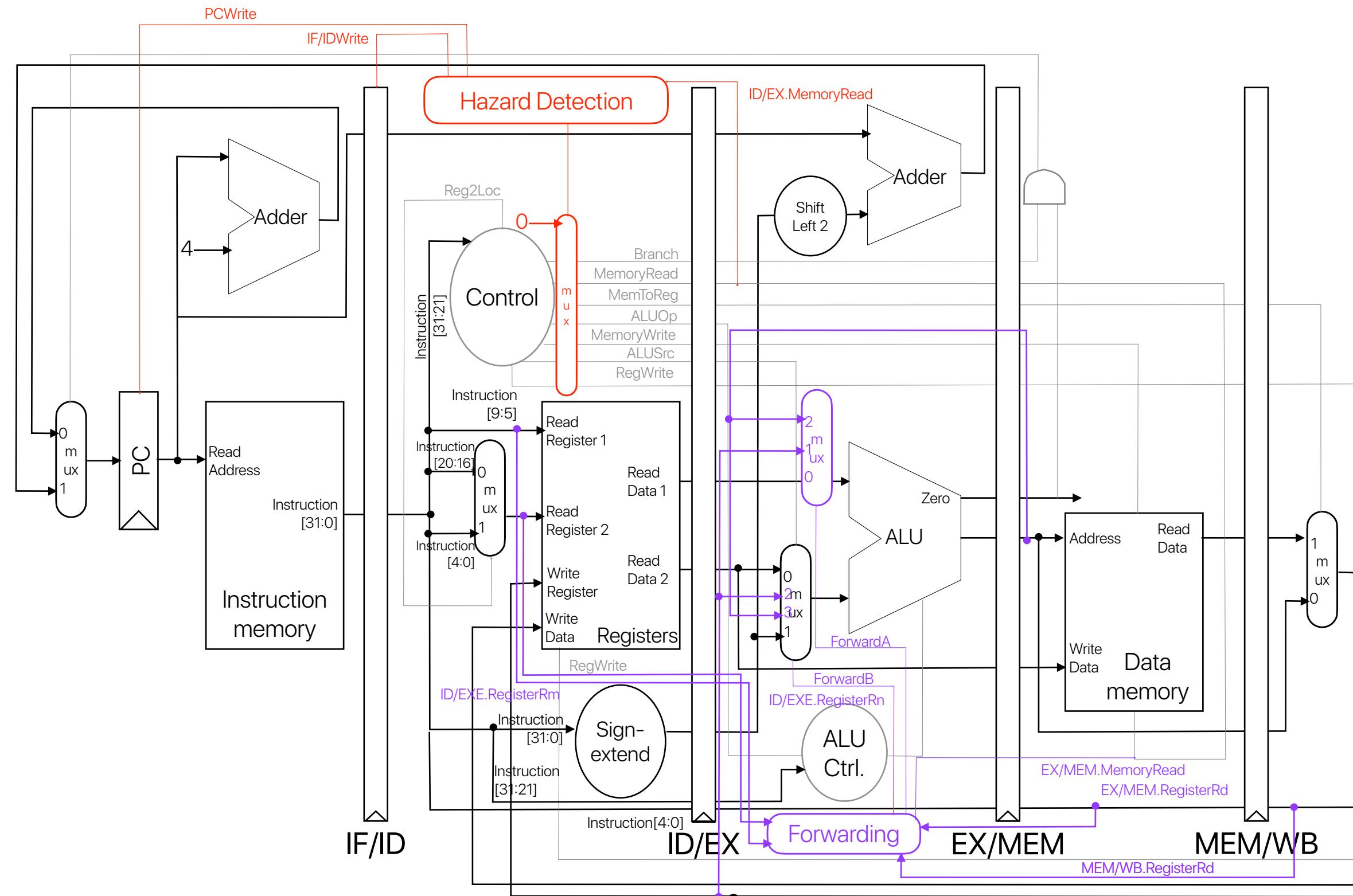
Say, we want to implement $a[i] += a[i+1]*20$

- This is what we need in RISC-V in each iteration

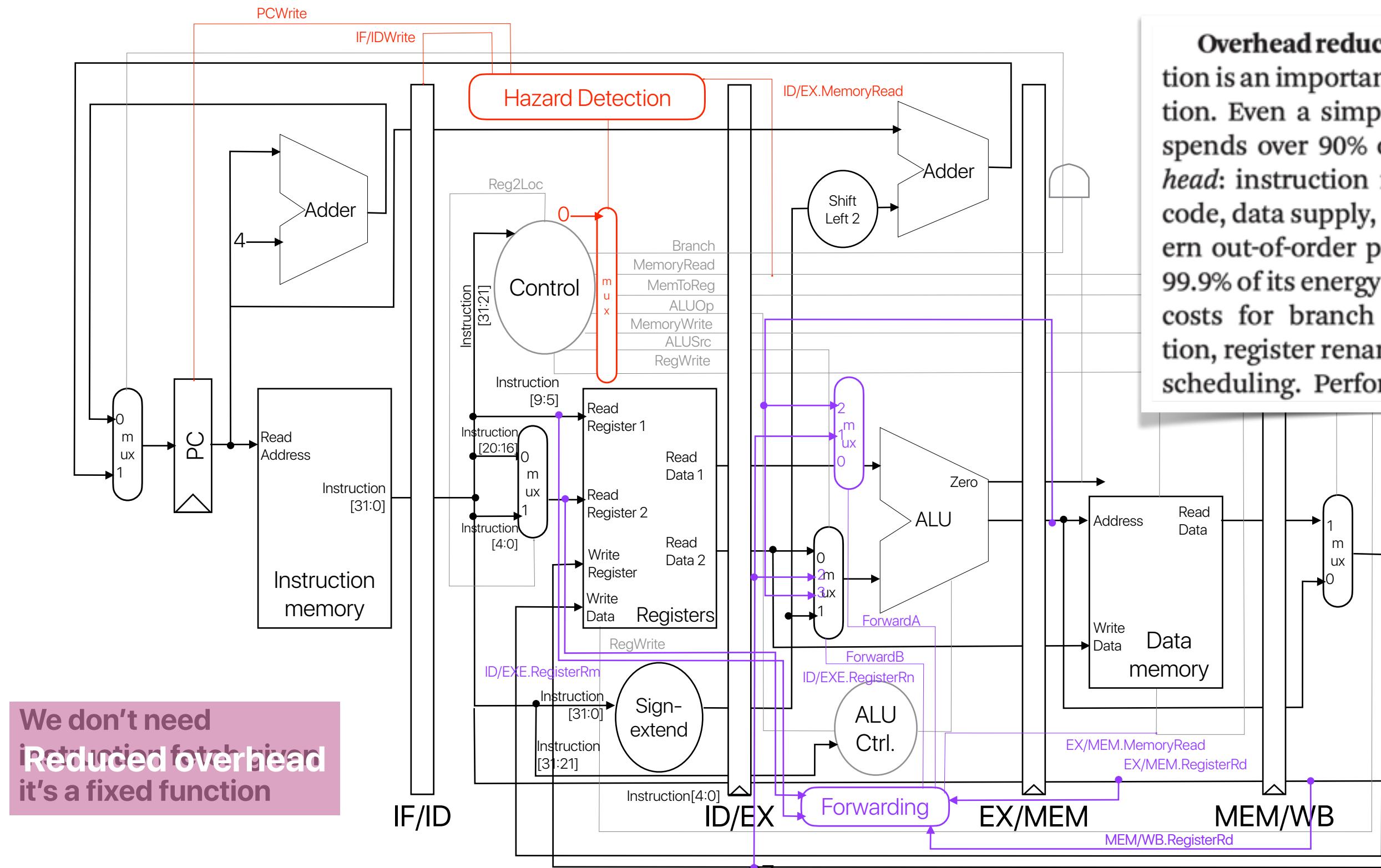
ld	X1,	0(X0)
ld	X2,	8(X0)
add	X3,	X31, #20
mul	X2,	X2, X3
add	X1,	X1, X2
sd	X1,	0(X0)



This is what you need for these instructions



Specialize the circuit/datapath

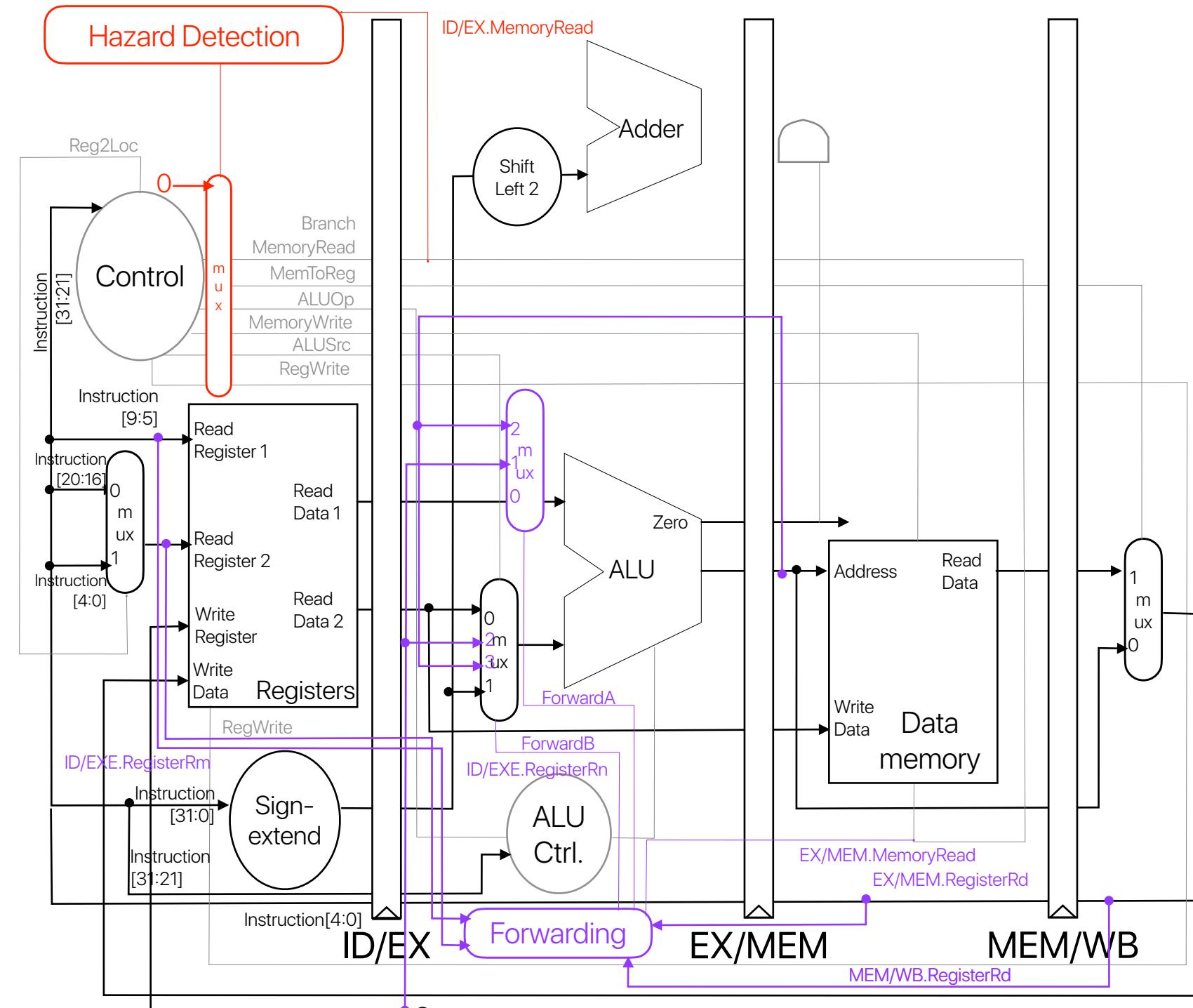


Overhead reduction. Overhead reduction is an important aspect of specialization. Even a simple in-order processor spends over 90% of its energy on *overhead*: instruction fetch, instruction decode, data supply, and control.¹⁰ A modern out-of-order processor spends over 99.9% of its energy on overhead⁵¹ adding costs for branch prediction, speculation, register renaming, and instruction scheduling. Performing a 32-b integer

Specialize the circuit/datapath

We don't need these many registers, complex control, decode

We don't need instruction fetch, we Reduced overhead it's a fixed function

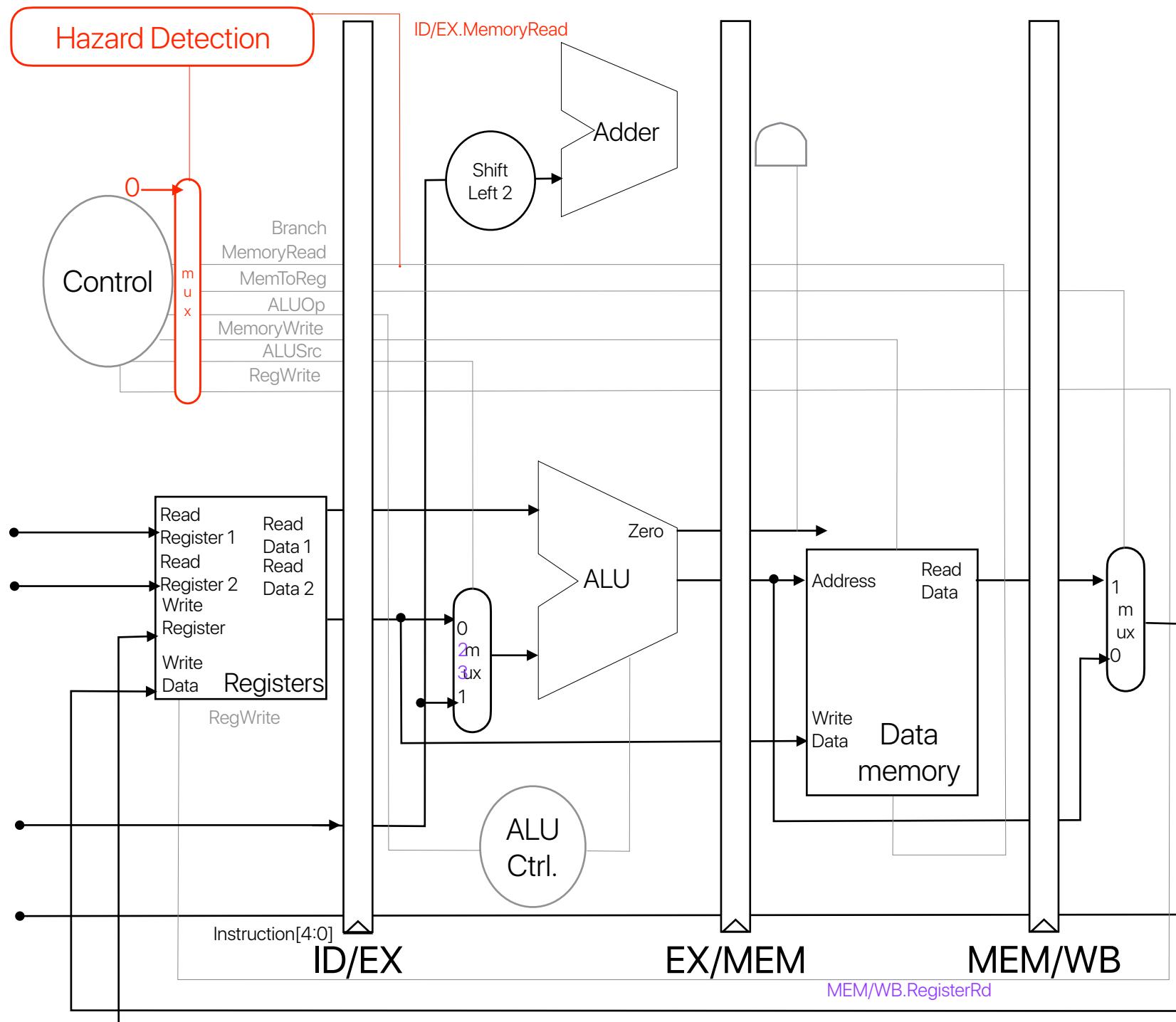


Specialize the circuit/datapath

We don't need ALUs,
branches, hazard
detections...

We don't need
these many registers, complex
control, decode

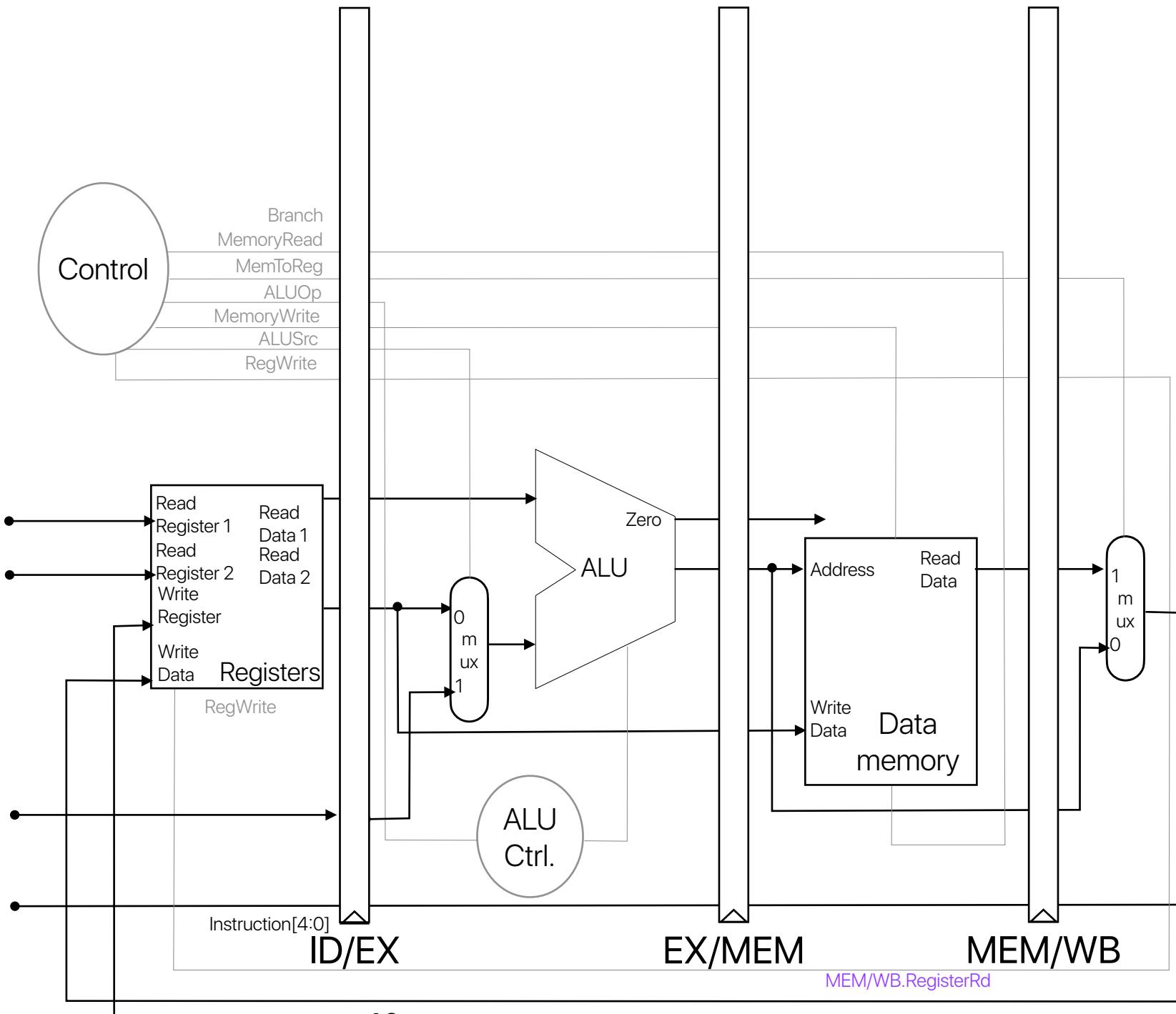
We don't need
instruction fetch
Reduced overhead
it's a fixed function



Specialize the circuit/datapath

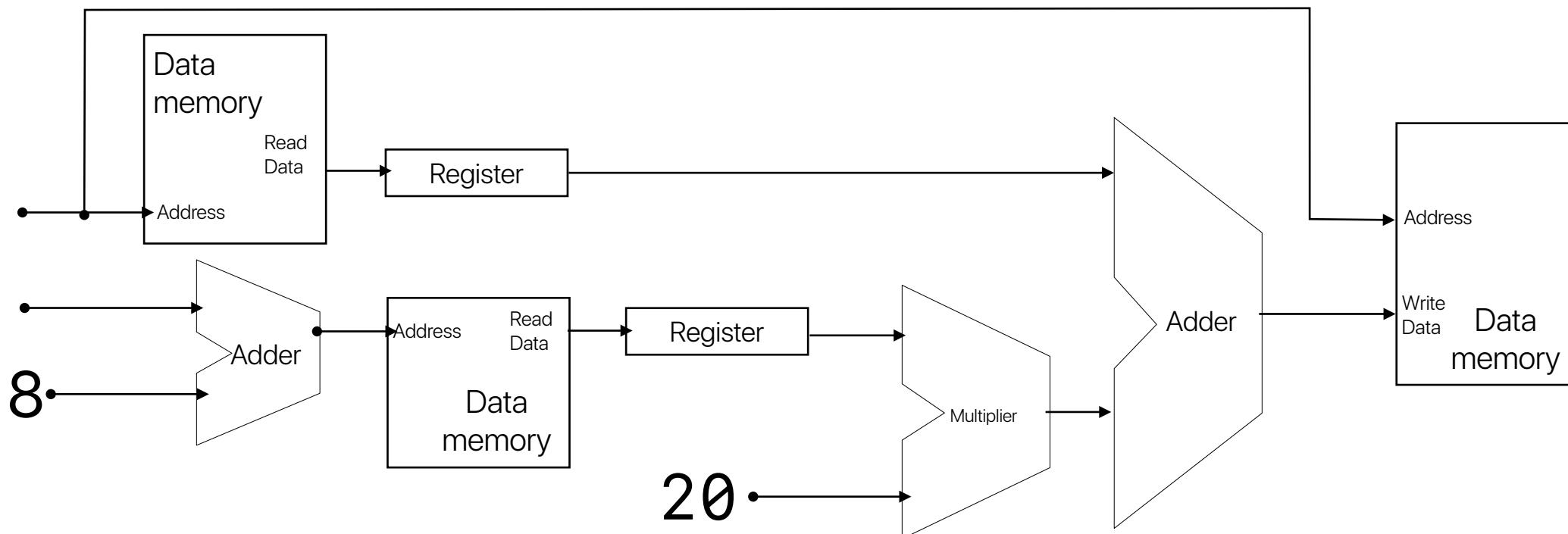
We don't need big ALUs,
branches, hazard
detections...
Datapath specialization
We don't need these
many registers, complex
control, decode

We don't need
instruction fetch
Reduced overhead
it's a fixed function



Rearranging the datapath

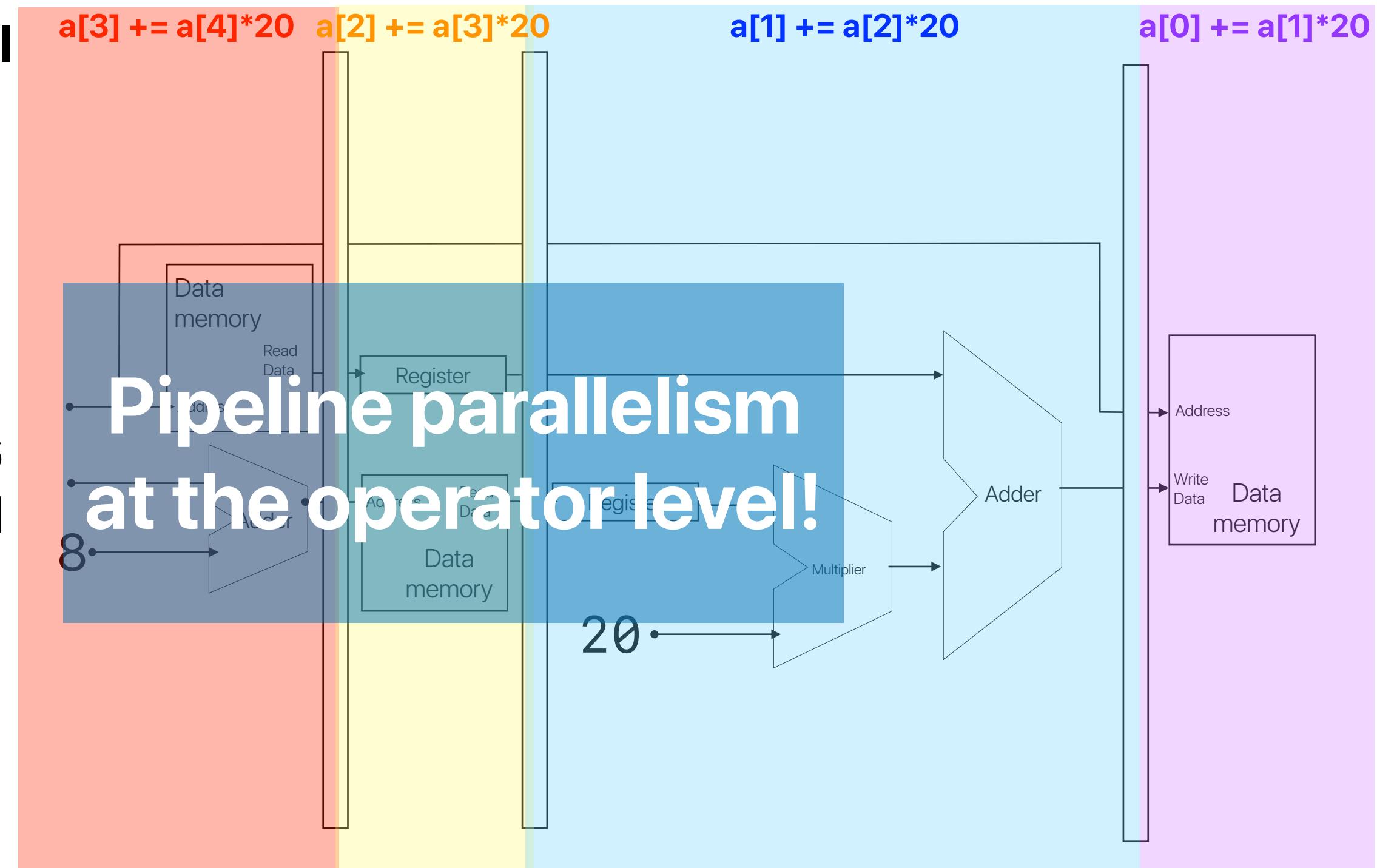
```
ld    X1, 0(X0)
ld    X2, 8(X0)
add   X3, X31, #20
mul   X2, X2, X3
add   X1, X1, X2
sd    X1, 0(X0)
```



The pipeline for $a[i] += a[i+1]*20$

Each stage can still
be as fast as the
pipelined
processor

But each stage is
now working on
what the original 6
instructions would
do



Why DSAs are more efficient?

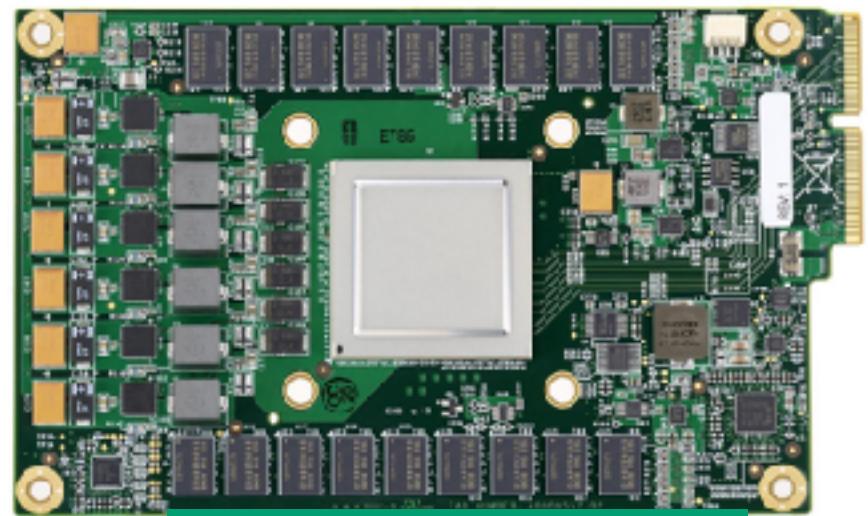
- Datapath specialization
- Parallelism
- Local and optimized memory
- Reduced overhead

In-Datacenter Performance Analysis of a Tensor Processing Unit

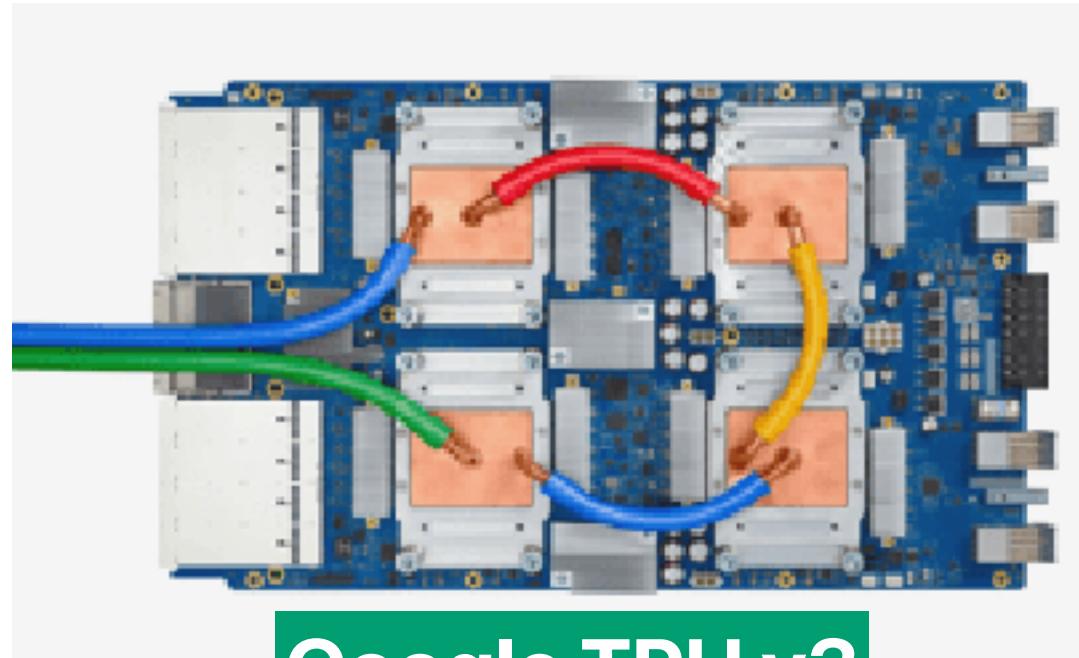
N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers, R. Boyle, P.-I. Cantin, C. Chao, C. Clark, J. Coriell, M. Daley, M. Dau, J. Dean, B. Gelb, T. V. Ghaemmaghami, R. Gottipati, W. Gulland, R. Hagmann, C. R. Ho, D. Hogberg, J. Hu, R. Hundt, D. Hurt, J. Ibarz, A. Jaffey, A. Jaworski, A. Kaplan, H. Khaitan, D. Killebrew, A. Koch, N. Kumar, S. Lacy, J. Laudon, J. Law, D. Le, C. Leary, Z. Liu, K. Lucke, A. Lundin, G. MacKean, A. Maggiore, M. Mahony, K. Miller, R. Na- garajan, R. Narayanaswami, R. Ni, K. Nix, T. Norrie, M. Omernick, N. Penukonda, A. Phelps, J. Ross, M. Ross, A. Salek, E. Samadiani, C. Severn, G. Sizikov, M. Snelham, J. Souter, D. Steinberg, A. Swing, M. Tan, G. Thorson, B. Tian, H. Toma, E. Tuttle, V. Vasudevan, R. Wal- ter, W. Wang, E. Wilcox, and D. H. Yoon

Google Inc.

Tensor Processing Units



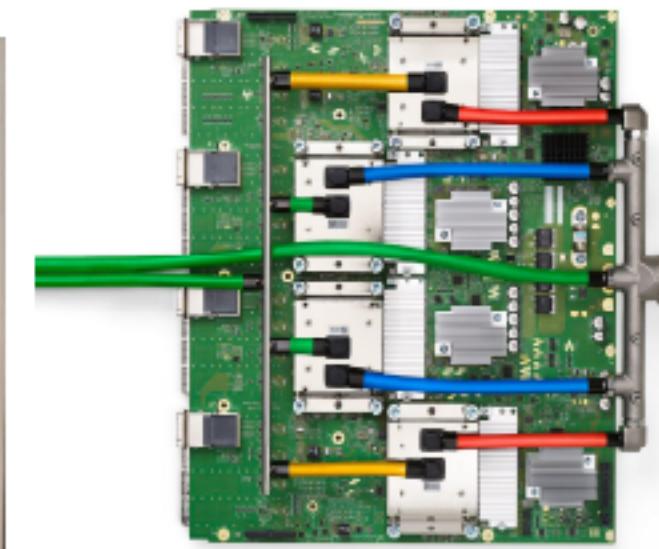
Google TPU v1



Google TPU v3



Google TPU v4



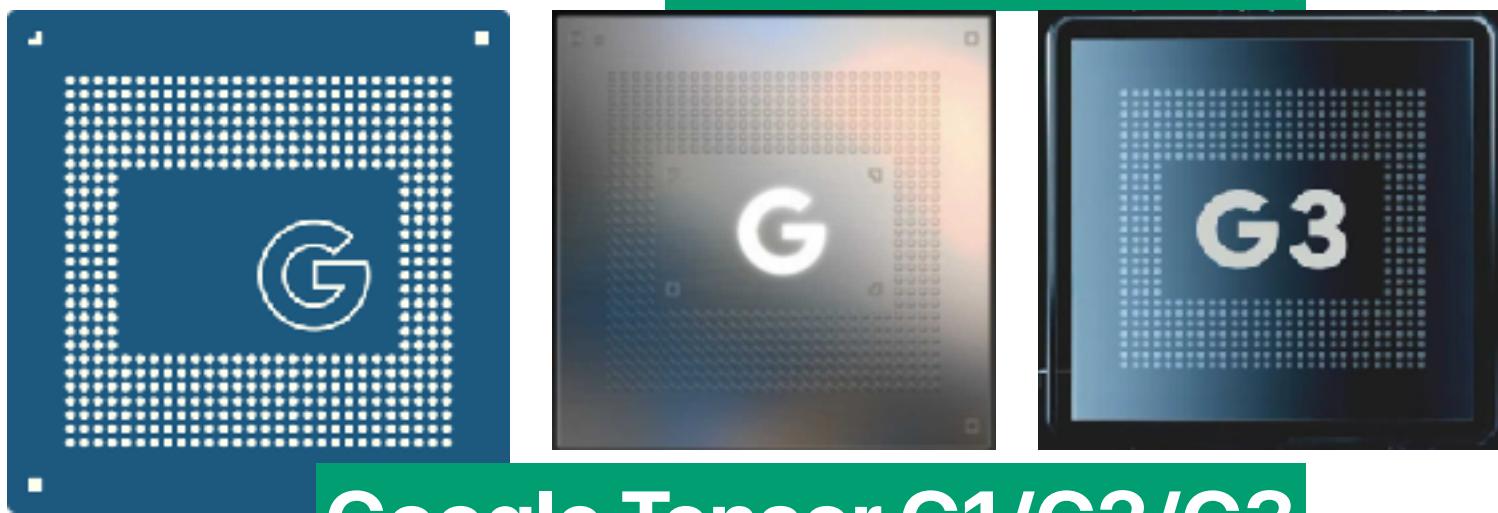
Edge TPU



Google TPU v2



Google TPU v5e



Google Tensor G1/G2/G3



TPU (Tensor Processing Unit)

- Regarding TPUs, please identify how many of the following statements are correct.
 - ① TPU is optimized for highly accurate matrix multiplications
 - ② TPU is designed for dense matrices, not for sparse matrices
 - ③ TPU can reduce the “IC” in the performance equation
 - ④ TPU features an instruction set architecture that facilitates pipeliningA. 0
B. 1
C. 2
D. 3
E. 4



TPU (Tensor Processing Unit)

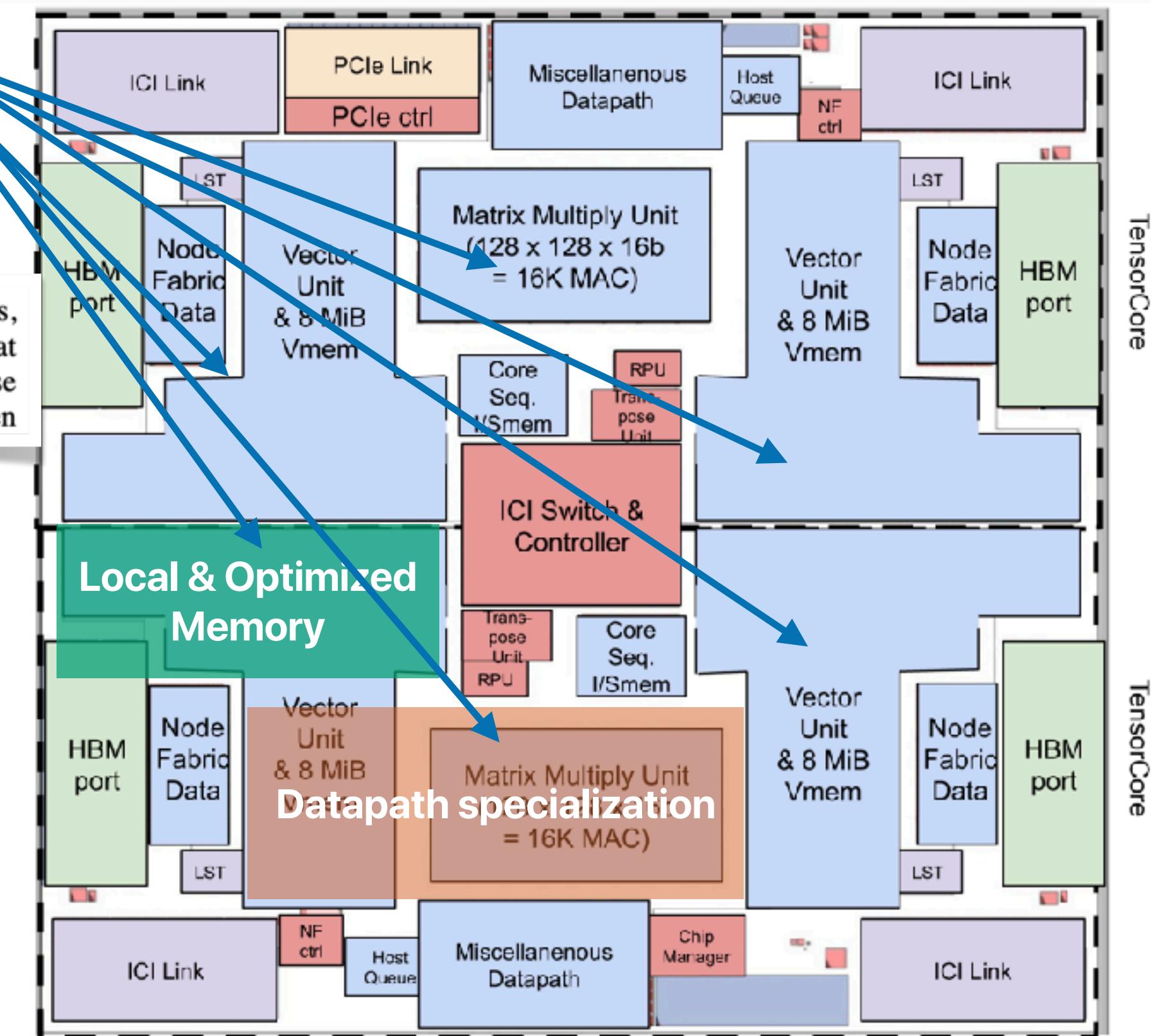
- Regarding TPUs, please identify how many of the following statements are correct.
 - ① TPU is optimized for highly accurate matrix multiplications
 - ② TPU is designed for dense matrices, not for sparse matrices
 - ③ TPU can reduce the “IC” in the performance equation
 - ④ TPU features an instruction set architecture that facilitates pipelining
- A. 0
- B. 1
- C. 2
- D. 3
- E. 4

TPUv2 floor plan

Parallelism

As instructions are sent over the relatively slow PCIe bus, TPU instructions follow the CISC tradition, including a repeat field. The average clock cycles per instruction (CPI) of these CISC instructions is typically 10 to 20. It has about a dozen

Reduced overhead



T. Norrie et al., "The Design Process for Google's Training Chips: TPUv2 and TPUv3," in IEEE Micro, vol. 41, no. 2, pp. 56-63

TensorFlow — the standard TPU programming interface

```
resolver = tf.distribute.cluster_resolver.TPUClusterResolver(tpu='')
```

```
tf.config.experimental_connect_to_cluster(resolver)
```

```
# This is the TPU initialization code that has to be at the beginning.
```

```
tf.tpu.experimental.initialize_tpu_system(resolver)
```

```
print("All devices: ", tf.config.list_logical_devices('TPU'))
```



```
a = tf.constant([[1.0, 2.0, 3.0], [4.0, 5.0, 6.0]])
```

```
b = tf.constant([[1.0, 2.0], [3.0, 4.0], [5.0, 6.0]])
```



```
with tf.device('/TPU:0'):
```

```
    c = tf.matmul(a, b)
```



```
print("c device: ", c.device)
```

```
print(c)
```

Only very high-level mathematical/domain-specific functions are exposed to the user!

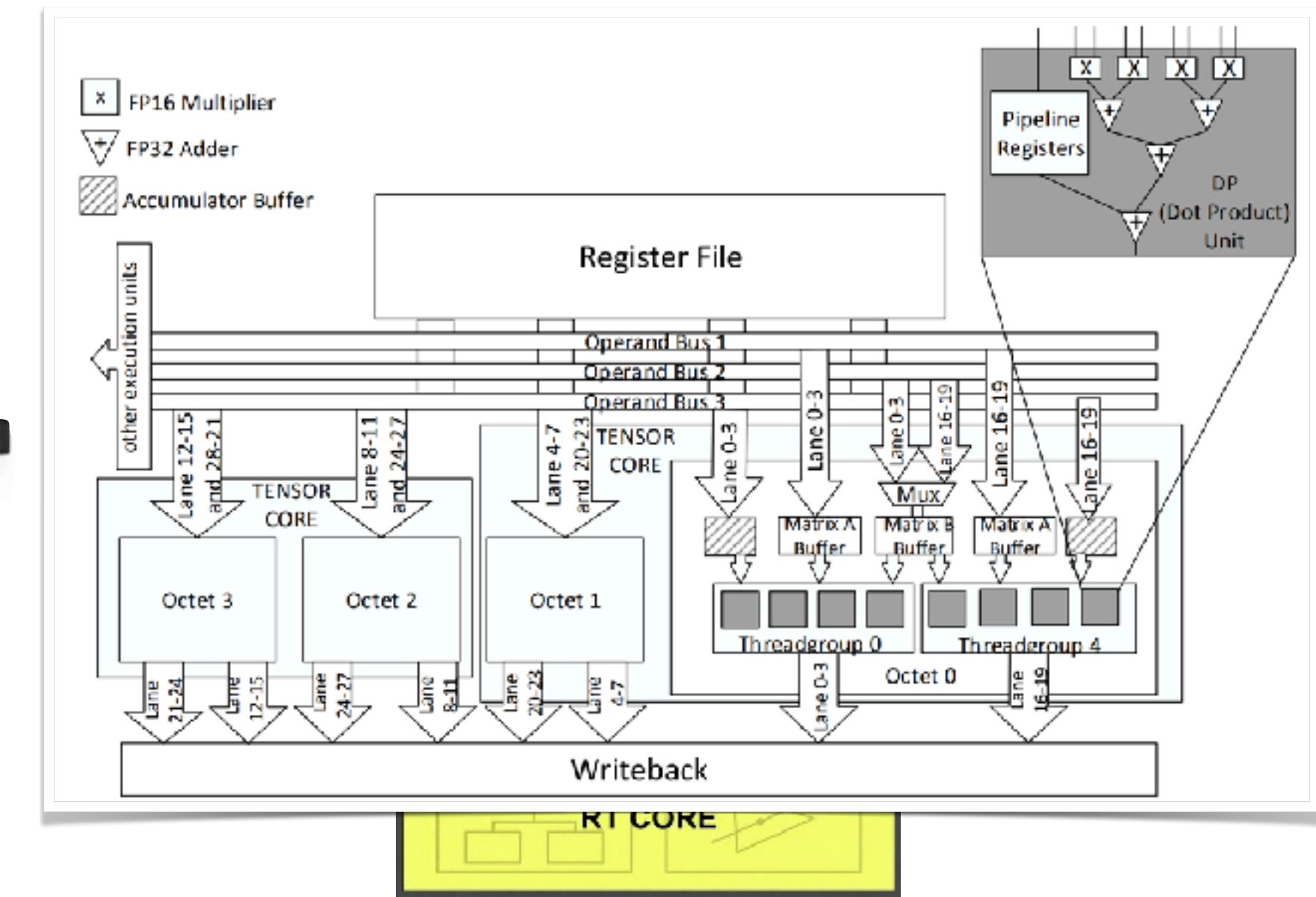
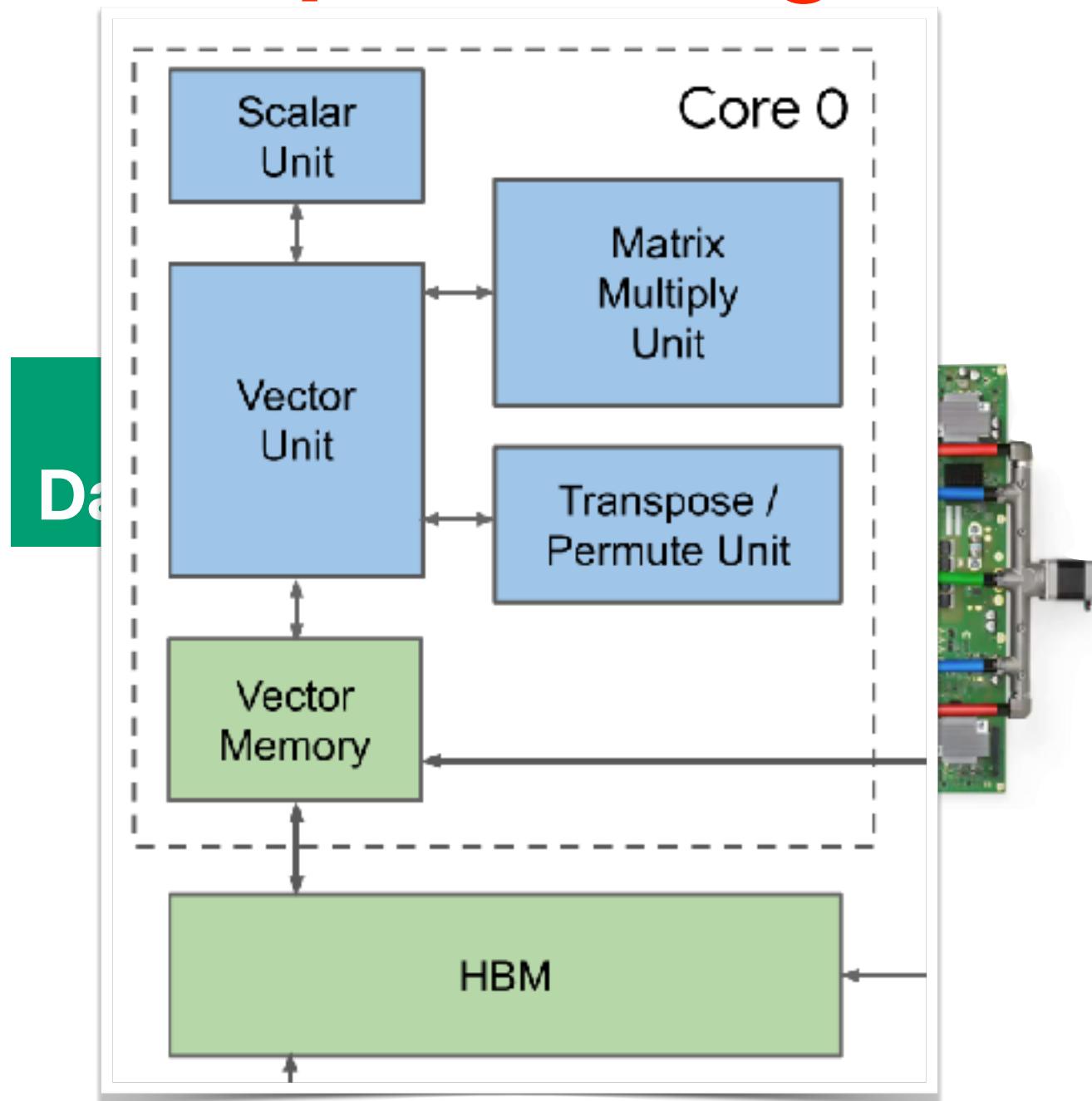
TPU (Tensor Processing Unit)

- Regarding TPUs, please identify how many of the following statements are correct.
 - ① TPU is optimized for highly accurate matrix multiplications
 - ② TPU is designed for dense matrices, not for sparse matrices
 - ③ TPU can reduce the “IC” in the performance equation
 - ④ TPU features an instruction set architecture that facilitates pipelining peak performance.
- A. 0
- B. 1
- C. 2
- D. 3
- E. 4
- Pitfall: Being ignorant of architecture history when designing a domain-specific architecture.*
- Ideas that didn't fly for general-purpose computing may be ideal for domain-specific architectures. For the TPU, three important architectural features date back to the early 1980s: systolic arrays [31], decoupled-access/execute [54], and CISC instructions [41]. The first reduced the area and power of the large matrix multiply unit, the second fetches weights concurrently during operation of the matrix multiply unit, and the third better utilizes the limited bandwidth of the PCIe bus for delivering instructions. History-aware architects could have a competitive edge.
- only possible if you have a limited domain
— inference and CNN are denser
— reduce the control overhead!
— one instruction performs multiple operations
- When using a mix of 8-bit weights and 16-bit activations (or vice versa), the Matrix Unit computes at half-speed, and it computes at a quarter-speed when both are 16 bits. It reads and takes to shift a tile in). This unit is designed for dense matrices. Sparse architectural support was omitted for time-to-deployment reasons. The weights for the matrix unit are staged through an on-
- As instructions are sent over the relatively slow PCIe bus, TPU instructions follow the CISC tradition, including a repeat field. The average clock cycles per instruction (CPI) of these CISC instructions is typically 10 to 20. It has about a dozen

Hardware Acceleration Solutions

- Using hardware accelerations, GPUs, Field Programmable Gate Arrays (FPGAs) and Domain-specific accelerators (DSAs), are increasingly popular to help tackle the demand of applications. Assume we have **optimized** implementations on each hardware acceleration options when running the same application, please identify how many of them are correct.
 - ① FPGAs typically can outperform GPUs and offer better performance-per-watt as FPGAs can realize more efficient logics using their internal look-up-tables
 - ② DSAs offer the best performance-per-watt than GPUs and FPGAs
 - ③ DSAs has the worst programmability compared to GPUs and FPGAs
 - ④ DSAs are more likely to be memory-bounded than others
 - A. 0
 - B. 1
 - C. 2
 - D. 3
 - E. 4

Matrix processing — the core of AI/ML accelerators

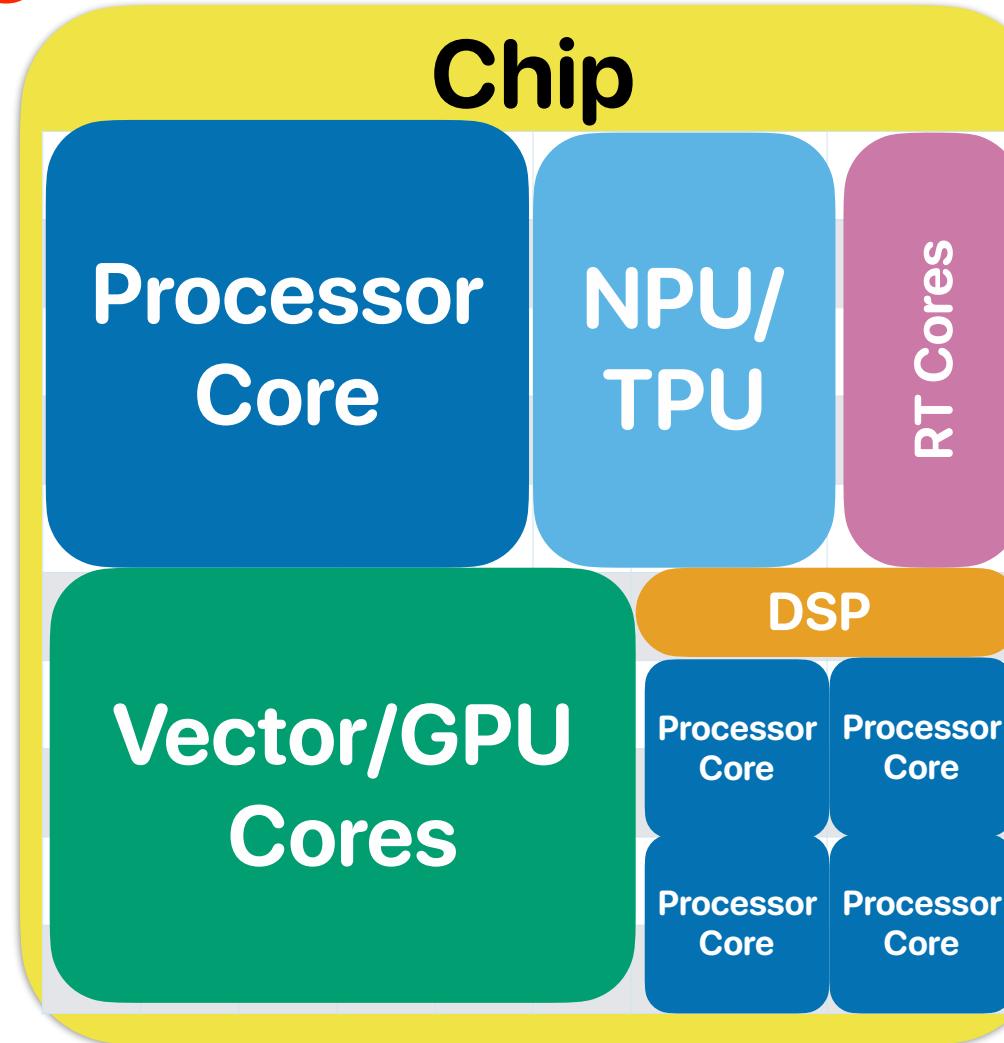


T. Norrie et al., "The Design Process for Google's Training Chips: TPUv2 and TPUv3," in IEEE Micro, vol. 41, no. 2, pp. 56-63

M. Raihan, N. Goli and T. Aamodt, "Modeling Deep Learning Accelerator Enabled GPUs, ISPASS 2019

Given the same power budget, maximize the efficiency per chip

**Some at top speed,
me are not functioning.**



**Turn off unnecessary
ones when we don't
need specialized
functions!**

NVIDIA's Tensor Cores



What are Tensor Cores?

Tesla V100's Tensor Cores are programmable matrix-multiply-and-accumulate units that can deliver up to 125 Tensor TFLOPS for training and inference applications. The Tesla V100 GPU contains 640 Tensor Cores: 8 per SM. Tensor Cores and their associated data paths are custom-crafted to dramatically increase floating-point compute throughput at only modest area and power costs. Clock gating is used extensively to maximize power savings.

Each Tensor Core provides a 4x4x4 matrix processing array which performs the operation $\mathbf{D} = \mathbf{A} * \mathbf{B} + \mathbf{C}$, where \mathbf{A} , \mathbf{B} , \mathbf{C} and \mathbf{D} are 4x4 matrices as Figure 1 shows. The matrix multiply inputs \mathbf{A} and \mathbf{B} are FP16 matrices, while the accumulation matrices \mathbf{C} and \mathbf{D} may be FP16 or FP32 matrices.

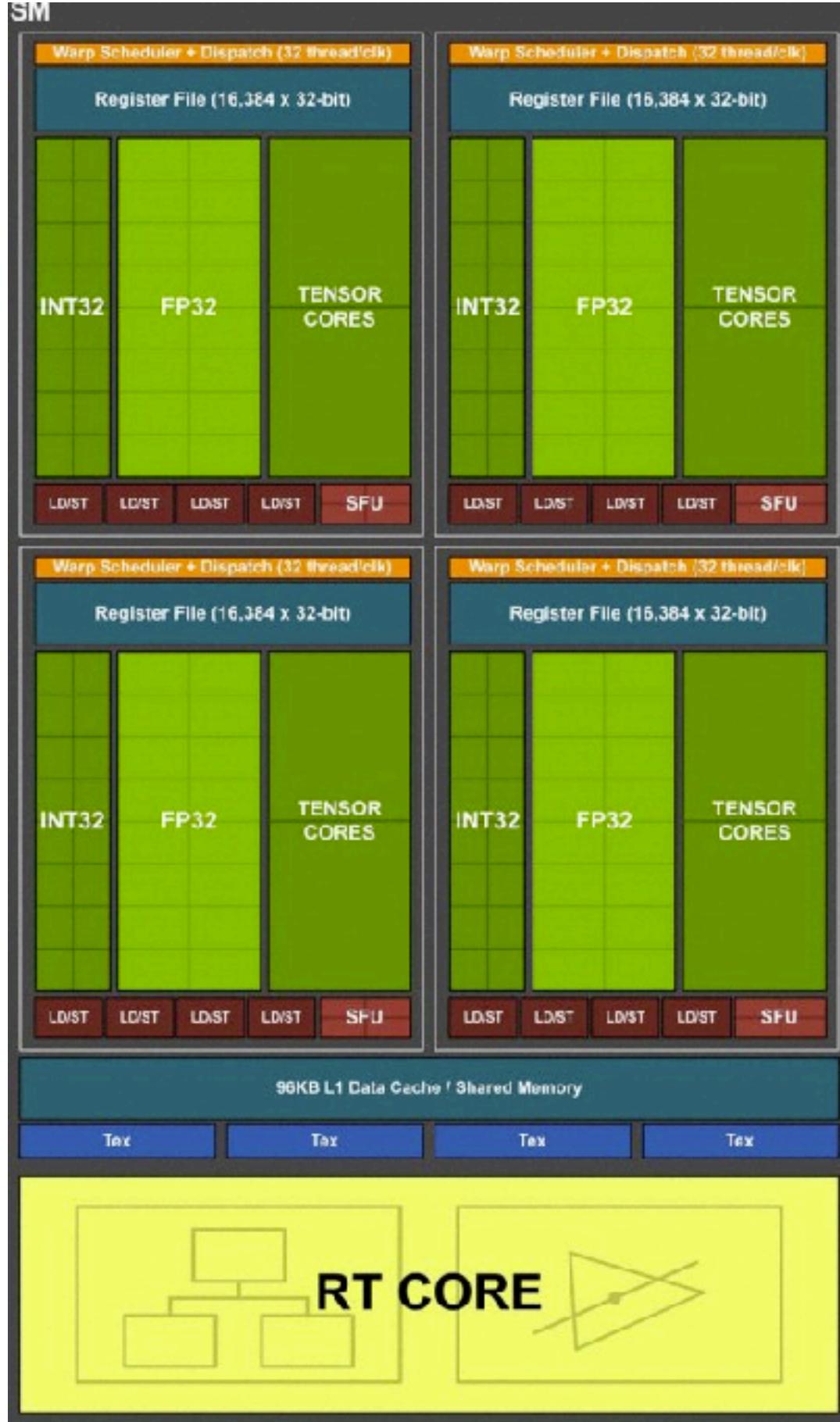
$$\mathbf{D} = \left(\begin{array}{cccc} \mathbf{A}_{0,0} & \mathbf{A}_{0,1} & \mathbf{A}_{0,2} & \mathbf{A}_{0,3} \\ \mathbf{A}_{1,0} & \mathbf{A}_{1,1} & \mathbf{A}_{1,2} & \mathbf{A}_{1,3} \\ \mathbf{A}_{2,0} & \mathbf{A}_{2,1} & \mathbf{A}_{2,2} & \mathbf{A}_{2,3} \\ \mathbf{A}_{3,0} & \mathbf{A}_{3,1} & \mathbf{A}_{3,2} & \mathbf{A}_{3,3} \end{array} \right) \left(\begin{array}{cccc} \mathbf{B}_{0,0} & \mathbf{B}_{0,1} & \mathbf{B}_{0,2} & \mathbf{B}_{0,3} \\ \mathbf{B}_{1,0} & \mathbf{B}_{1,1} & \mathbf{B}_{1,2} & \mathbf{B}_{1,3} \\ \mathbf{B}_{2,0} & \mathbf{B}_{2,1} & \mathbf{B}_{2,2} & \mathbf{B}_{2,3} \\ \mathbf{B}_{3,0} & \mathbf{B}_{3,1} & \mathbf{B}_{3,2} & \mathbf{B}_{3,3} \end{array} \right) + \left(\begin{array}{cccc} \mathbf{C}_{0,0} & \mathbf{C}_{0,1} & \mathbf{C}_{0,2} & \mathbf{C}_{0,3} \\ \mathbf{C}_{1,0} & \mathbf{C}_{1,1} & \mathbf{C}_{1,2} & \mathbf{C}_{1,3} \\ \mathbf{C}_{2,0} & \mathbf{C}_{2,1} & \mathbf{C}_{2,2} & \mathbf{C}_{2,3} \\ \mathbf{C}_{3,0} & \mathbf{C}_{3,1} & \mathbf{C}_{3,2} & \mathbf{C}_{3,3} \end{array} \right)$$

FP16 or FP32 FP16 FP16 FP16 or FP32

Figure 1: Tensor Core 4x4x4 matrix multiply and accumulate.

<https://developer.nvidia.com/blog/programming-tensor-cores-cuda-9/>

SM



NVIDIA GPU =
CUDA Cores (vector) +
Tensor Cores (matrix) +
RT Cores (tree intersection) +
DPX Units (dynamic programming)

Programming in Turing Architecture

Use tensor cores

```
cublasErrCheck(cublasSetMathMode(cublasHandle, CUBLAS_TENSOR_OP_MATH));
```

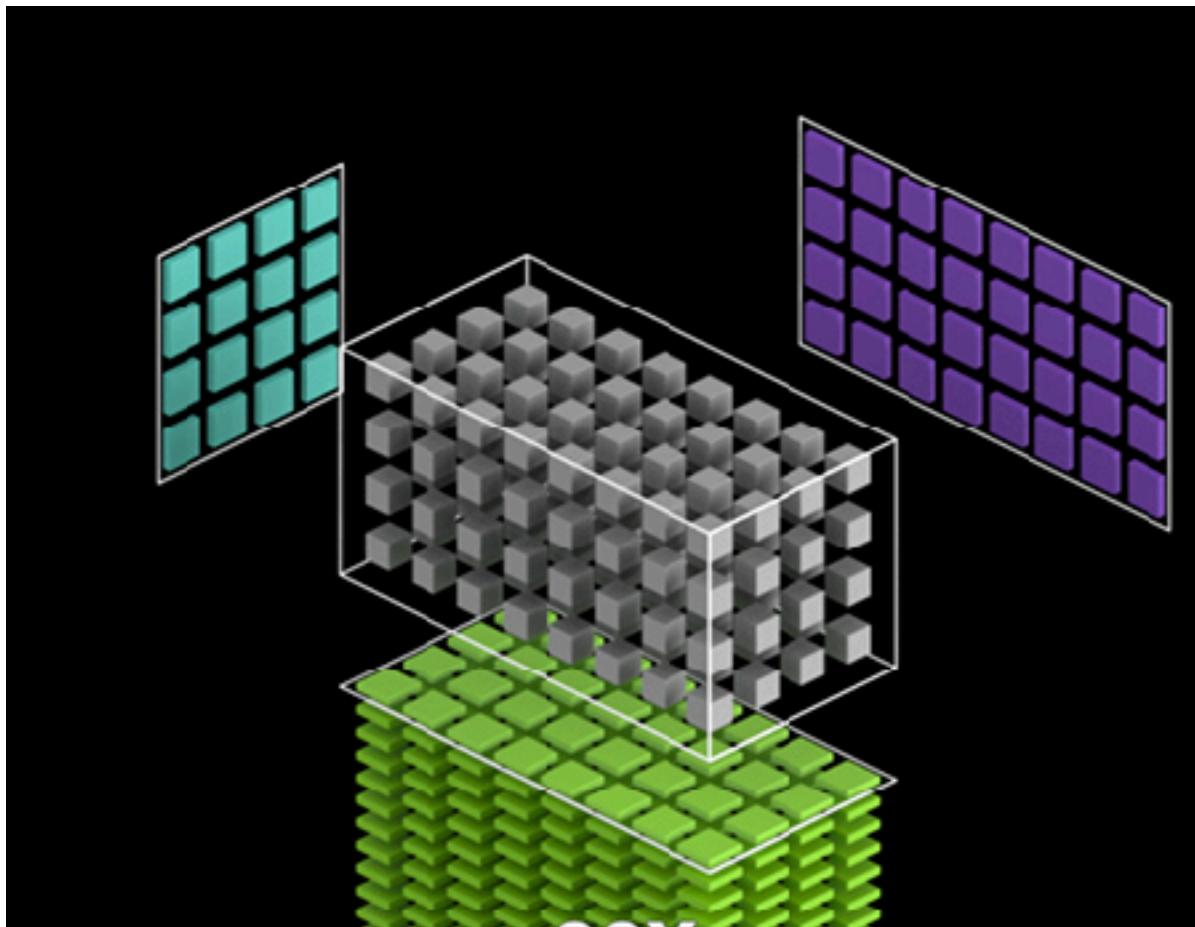
Make them 16-bit

```
convertFp32ToFp16 <<< (MATRIX_M * MATRIX_K + 255) / 256, 256 >>> (a_fp16, a_fp32,  
MATRIX_M * MATRIX_K);  
convertFp32ToFp16 <<< (MATRIX_K * MATRIX_N + 255) / 256, 256 >>> (b_fp16, b_fp32,  
MATRIX_K * MATRIX_N);
```

```
cublasErrCheck(cublasGemmEx(cublasHandle, CUBLAS_OP_N, CUBLAS_OP_N,  
    MATRIX_M, MATRIX_N, MATRIX_K,  
    &alpha,  
    a_fp16, CUDA_R_16F, MATRIX_M,  
    b_fp16, CUDA_R_16F, MATRIX_K,  
    &beta,  
    c_cublas, CUDA_R_32F, MATRIX_M,  
    CUDA_R_32F, CUBLAS_GEMM_DFALT_TENSOR_OP));
```

call Gemm

How can Tensor Cores faster despite higher algorithm complexity?



Each tensor core operation performs 8x4x4 MMA in one cycle
~ 256 FP operations in conventional scalar models

	RTX 3090
CUDA Cores	10496
Tensor Cores	328

FLOPS of 8K by 8K matrix multiplications =
 $8192 \times 8192 \times 8192 \times 2$

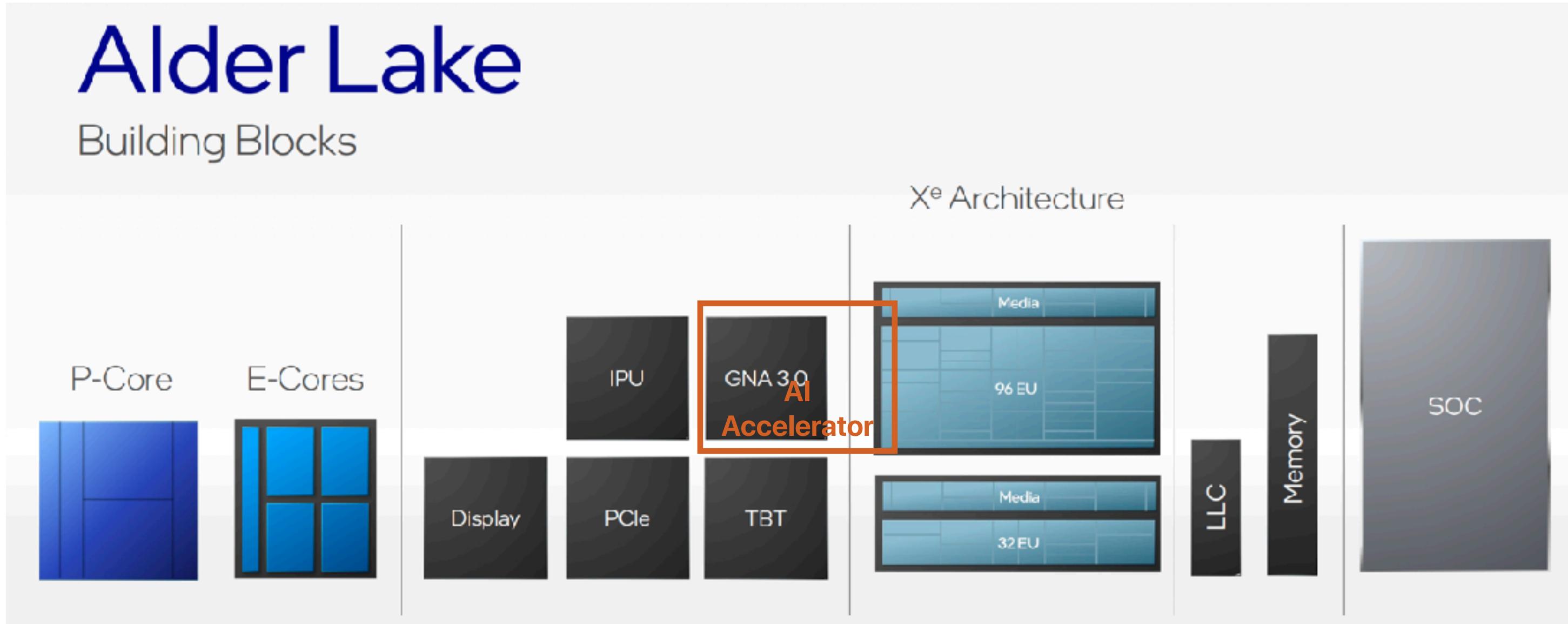
$$\frac{8192 \times 8192 \times 8192 \times 2}{10496} = 104,755,300 \text{ CUDA core cycles}$$

$$\frac{8192 \times 8192 \times 8192 \times 2}{328 \times 256} = 13,094,413 \text{ Tensor core cycles}$$



Tensor cores are 8x faster

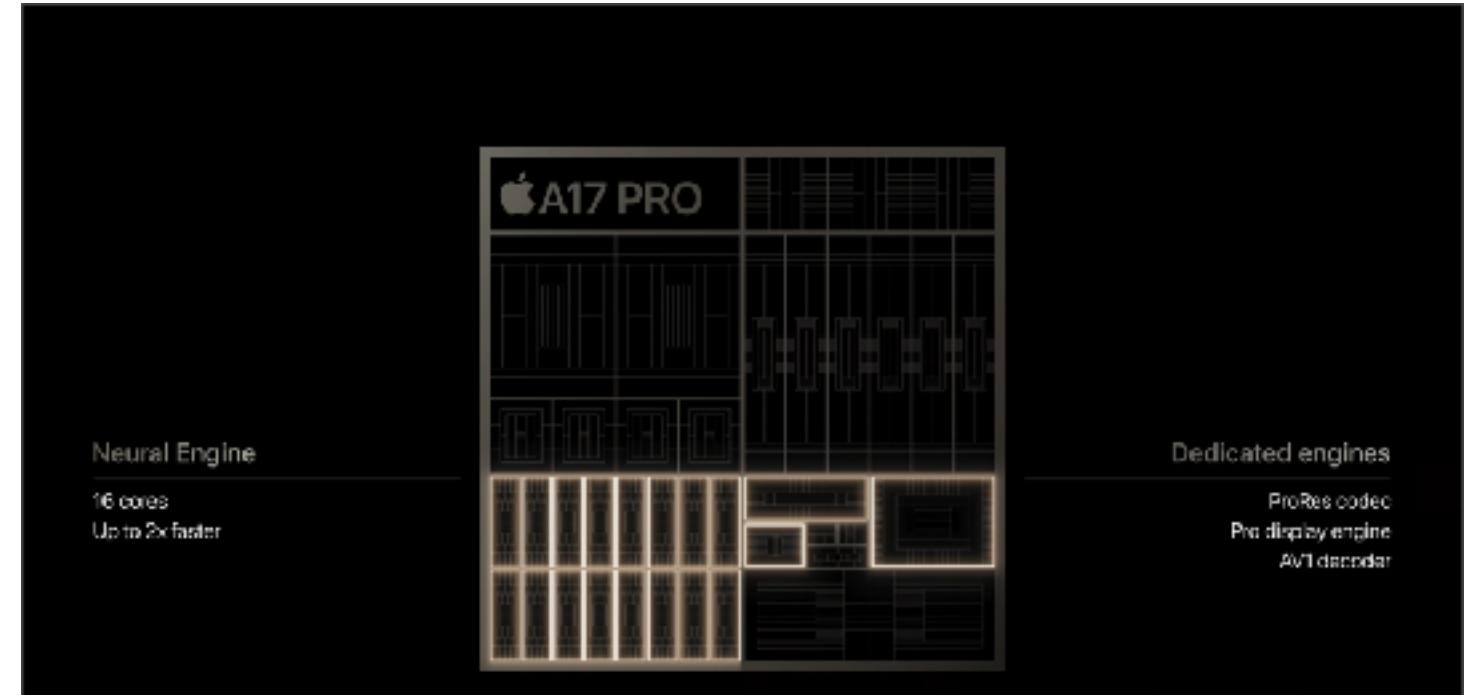
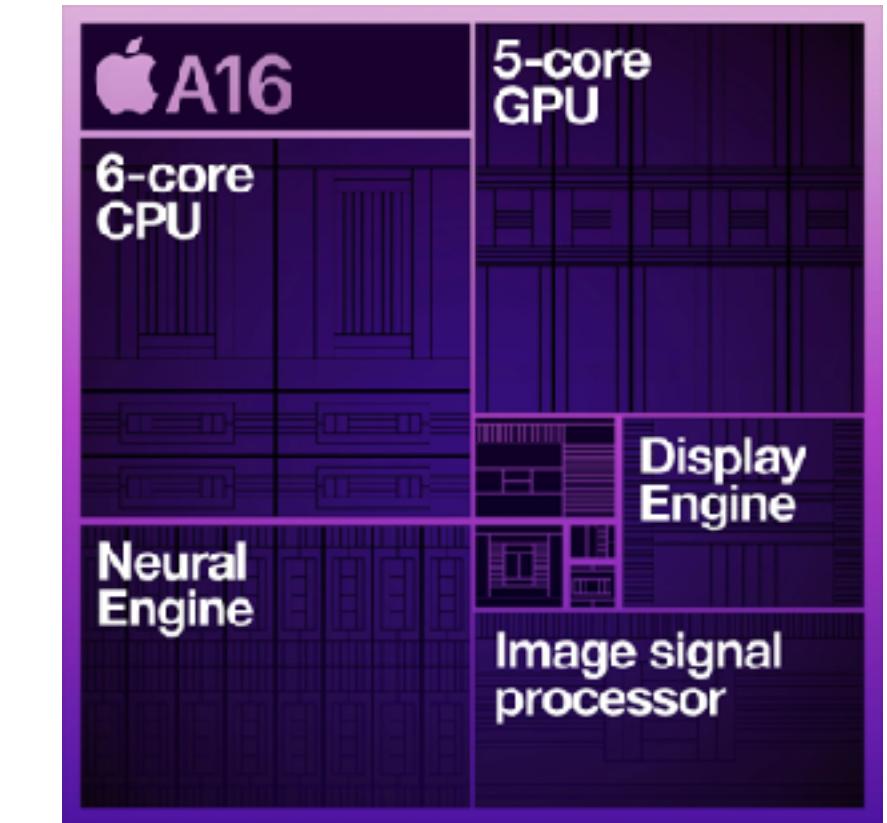
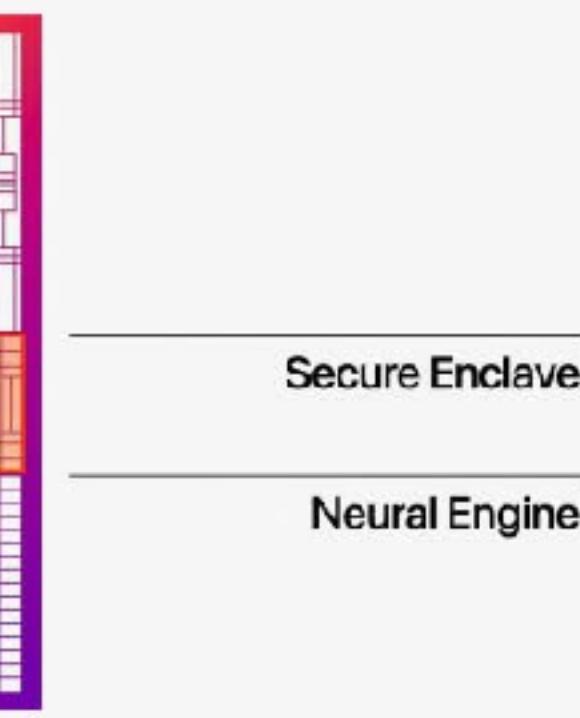
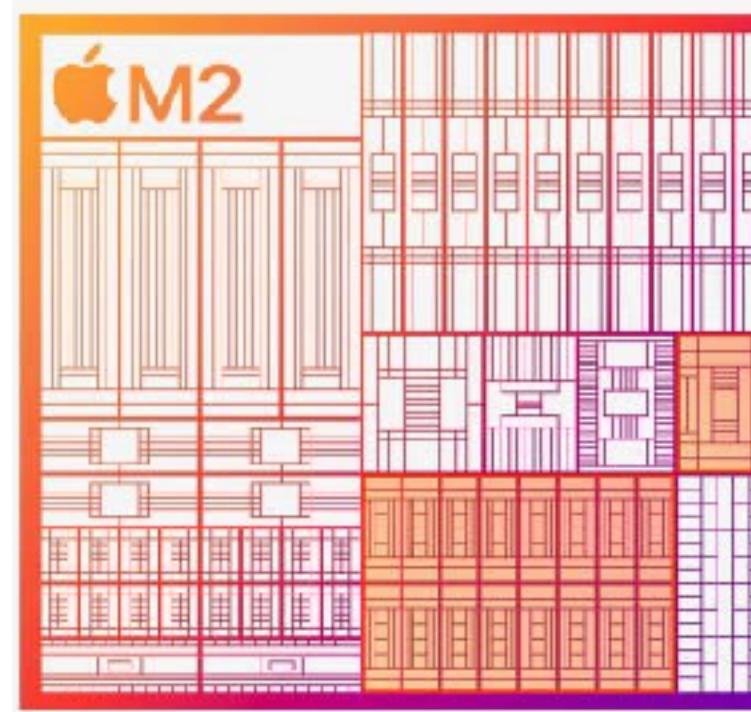
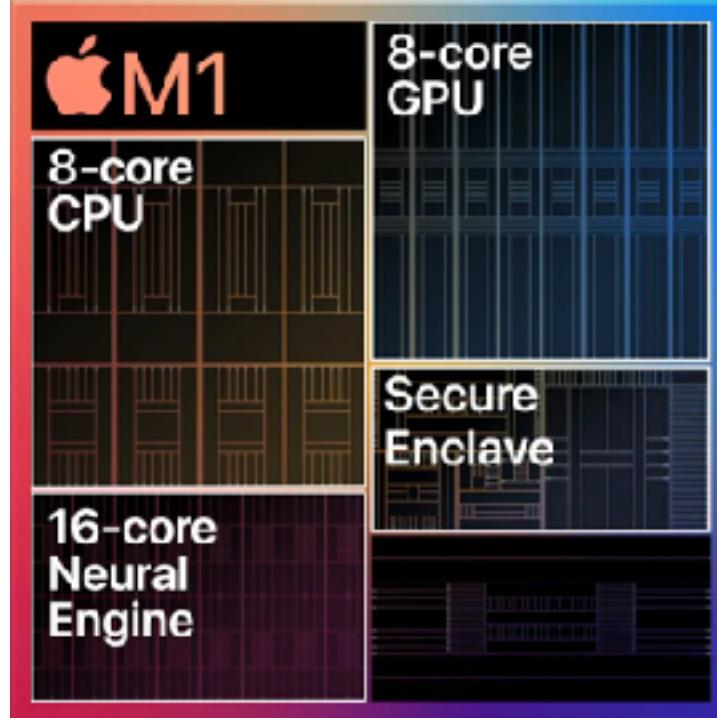
Modern processors also contain “accelerators”



Intel's Neural Processing Unit



Apple's Neural Engine

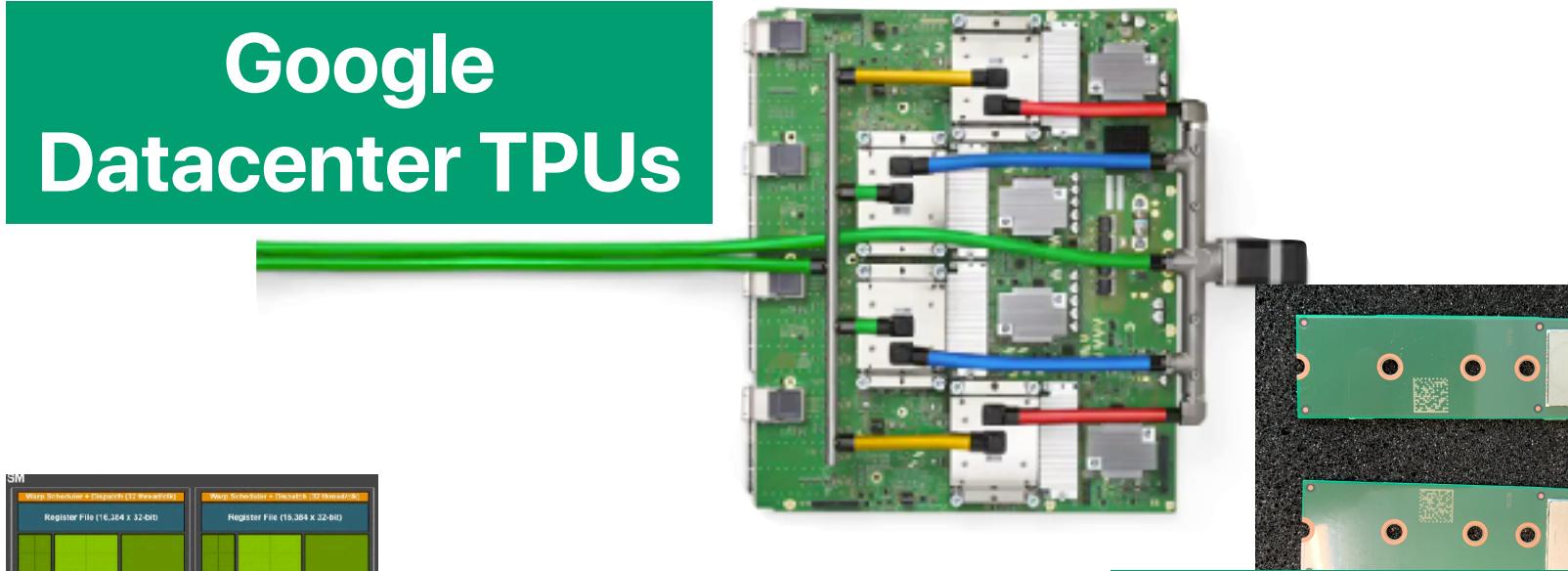
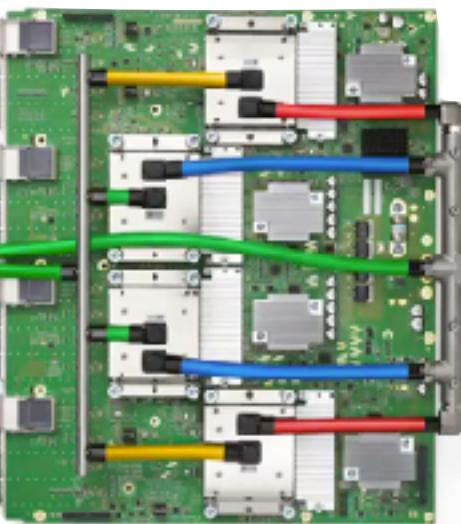


The “landscape” of modern computers

Google Datacenter TPUs

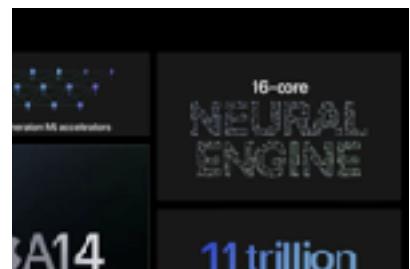


NVIDIA Tensor Core Units

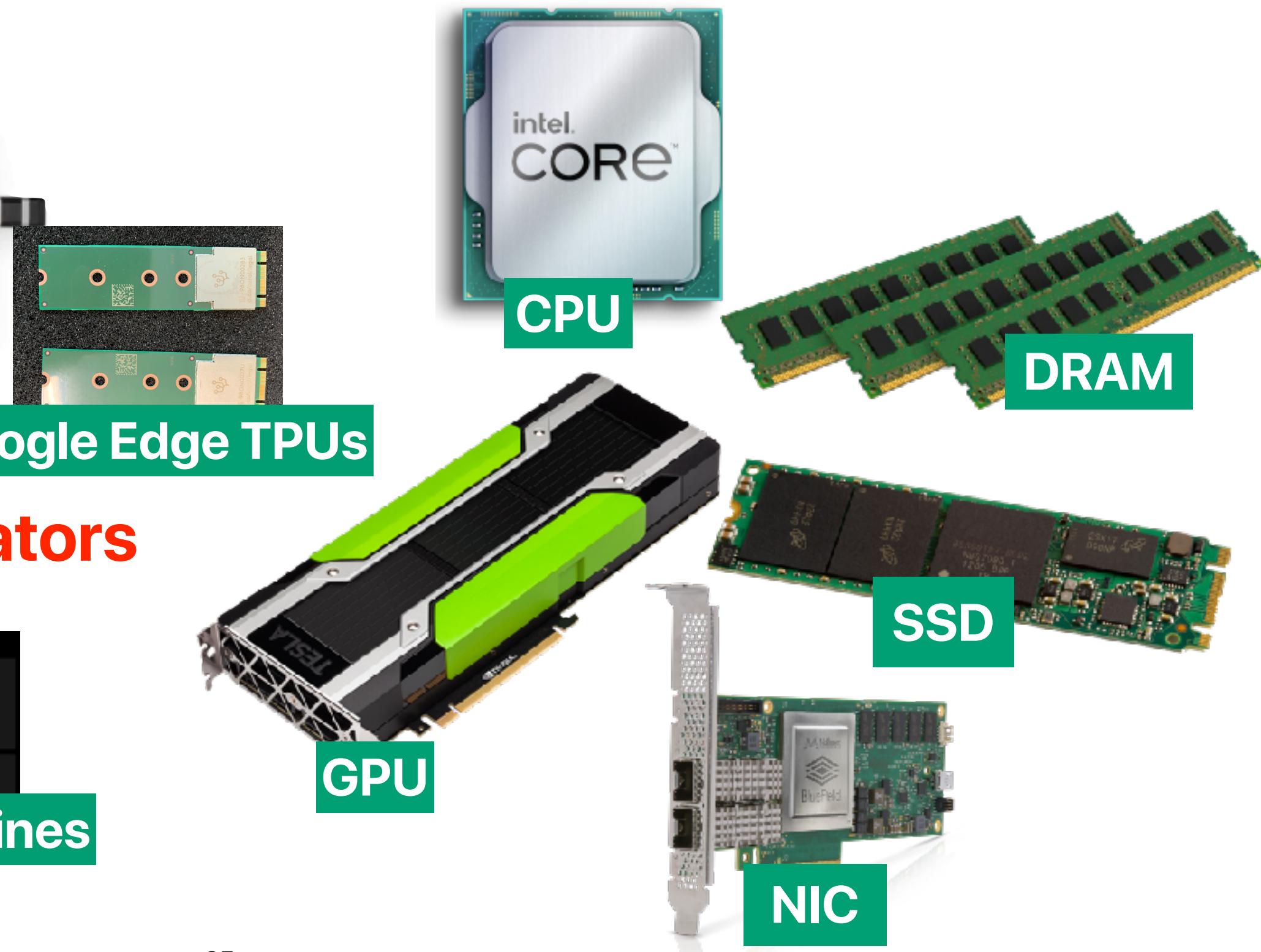


Google Edge TPUs

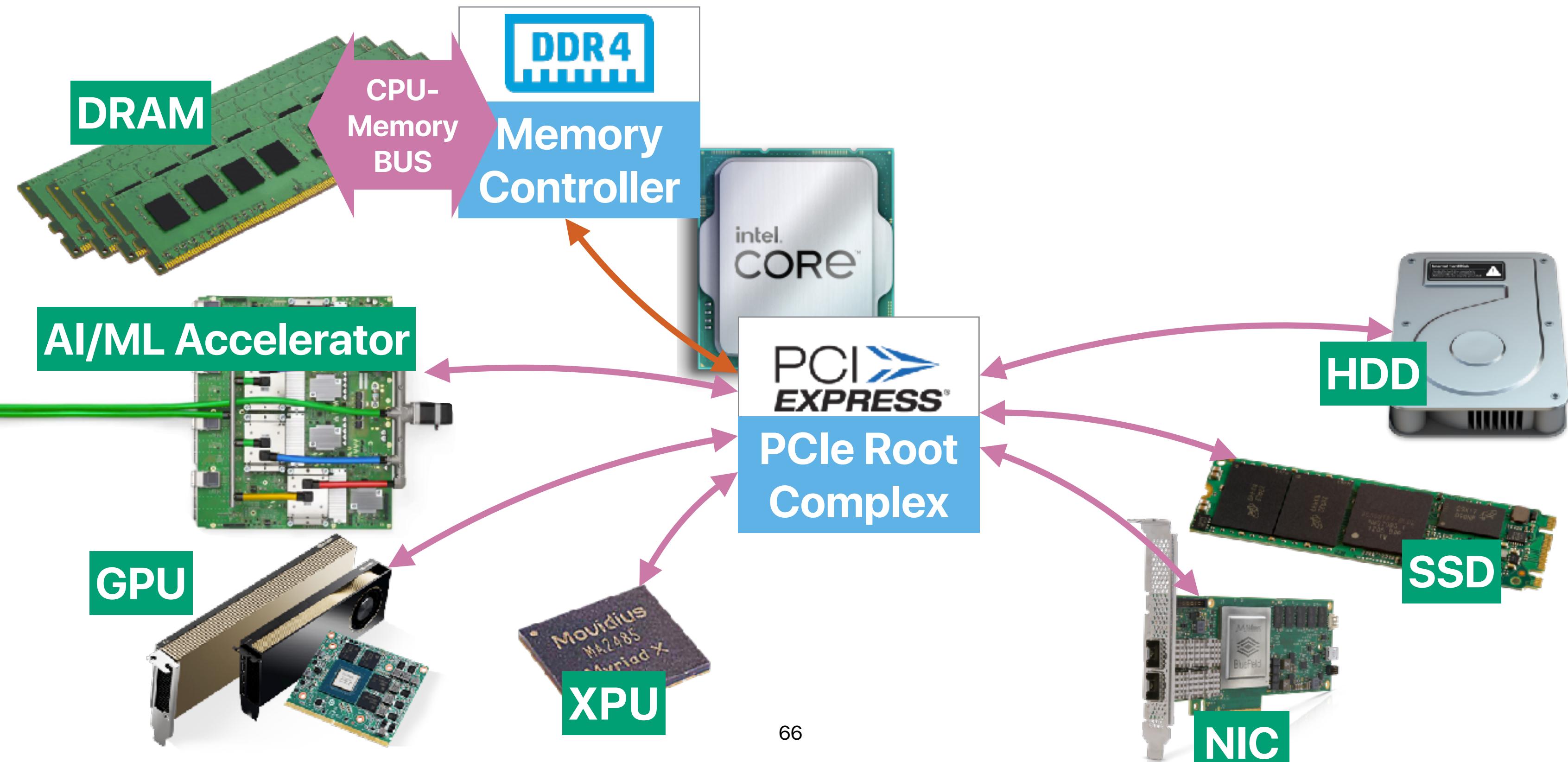
AI/ML Accelerators



Apple Neural Engines



No free lunch! — Data movement as an overhead



Data movement overhead before calling GPUs

```
// Allocate memory space on the device
D_TYPE *d_a, *d_b, *d_c;
cudaMalloc((void **) &d_a, sizeof(D_TYPE)*ARRAY_SIZE*ARRAY_SIZE);
cudaMalloc((void **) &d_b, sizeof(D_TYPE)*ARRAY_SIZE*ARRAY_SIZE);
cudaMalloc((void **) &d_c, sizeof(D_TYPE)*ARRAY_SIZE*ARRAY_SIZE);

// copy matrix A and B from host to device memory
cudaMemcpy(d_a, a, sizeof(D_TYPE)*ARRAY_SIZE*ARRAY_SIZE, cudaMemcpyHostToDevice);
cudaMemcpy(d_b, b, sizeof(D_TYPE)*ARRAY_SIZE*ARRAY_SIZE, cudaMemcpyHostToDevice);

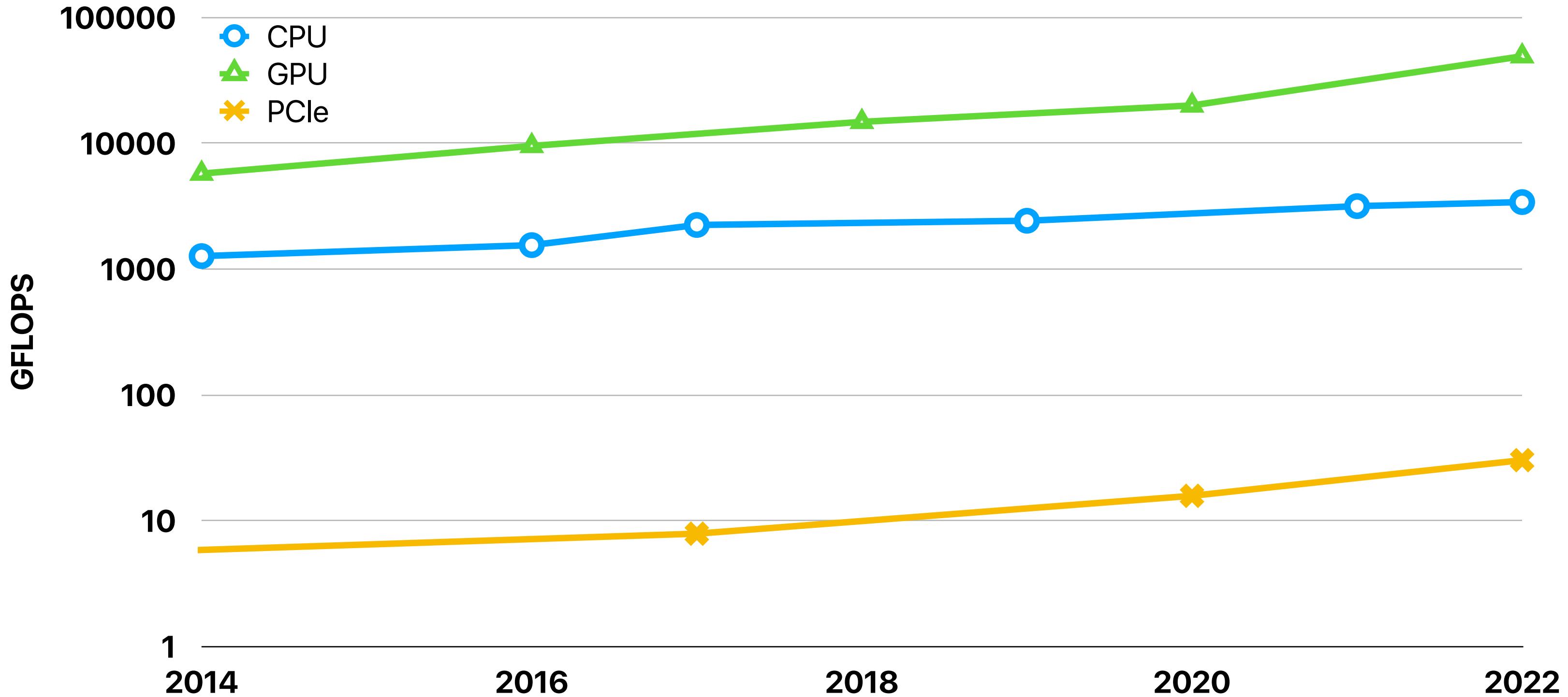
unsigned int grid_rows = (ARRAY_SIZE + BLOCK_SIZE - 1) / BLOCK_SIZE;
unsigned int grid_cols = (ARRAY_SIZE + BLOCK_SIZE - 1) / BLOCK_SIZE;
dim3 dimGrid(grid_cols, grid_rows);
dim3 dimBlock(BLOCK_SIZE, BLOCK_SIZE);

// Launch kernel
gpu_block_matrix_mult<<<dimGrid, dimBlock>>>(d_a, d_b, d_c, ARRAY_SIZE);

// Transfer results from device to host
cudaMemcpy(c, d_c, sizeof(D_TYPE)*ARRAY_SIZE*ARRAY_SIZE, cudaMemcpyDeviceToHost);
cudaThreadSynchronize();
// time counting terminate
cudaEventRecord(stop, 0);
cudaEventSynchronize(stop);

// compute time elapse on GPU computing
cudaEventElapsedTime(&gpu_elapsed_time_ms, start, stop);
```

The “speed” of PCIe compared to computing



Take-aways: the new golden age of computer architectures

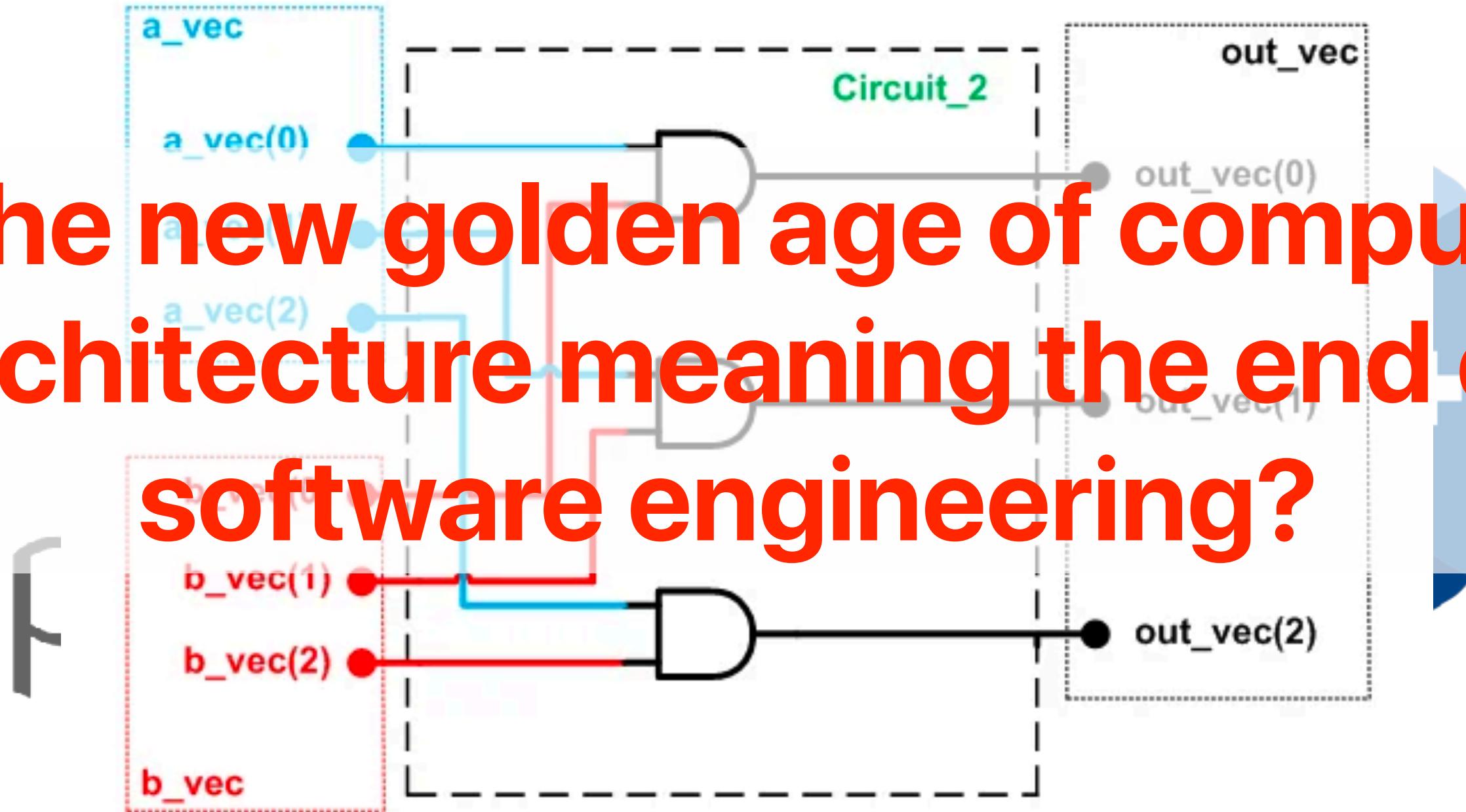
- Challenges and SOTA solutions in the dark silicon era
 - GPUs/many-core processors improve the **throughput** per-chip through providing massive parallelism where each processing element operates at a lower speed, but not-ideal for latency-sensitive workloads
 - Aggressive dynamic frequency/voltage scaling on CMP to accommodate the demand of **latency**-sensitive, parallelism-limited applications, but the area-efficiency of the slower cores is not great
 - Single ISA, heterogeneous CMPs (e.g., big.Little cores, Intel/Apple's P-cores/E-cores) find a balance the trade-offs of general-purpose workloads, but won't be ideal if your applications go to either extreme of throughput or latency
 - Domain-specific accelerators or application-specific ICs make more efficient use of chip areas to fulfill the latency or throughput demand of applications, but sacrifices flexibility and application designers are now also hardware designers



Conclusion: A New *Golden Age*

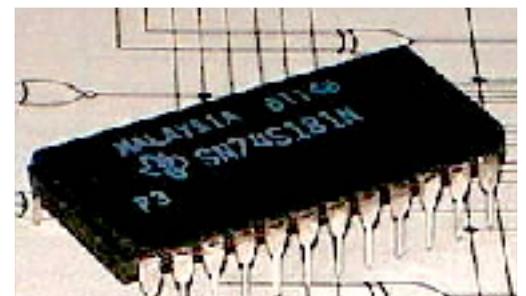


Is the new golden age of computer
architecture meaning the end of
software engineering?





The evolutionary cycle of computing



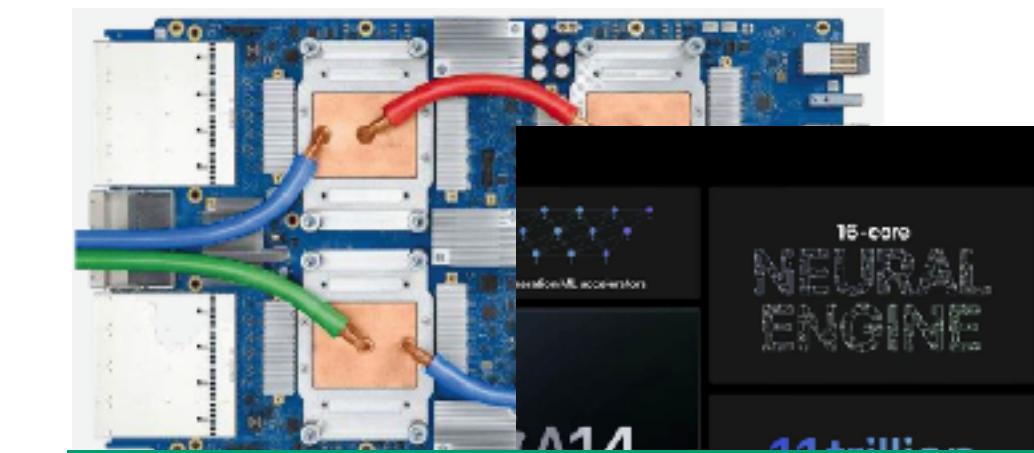
74181 Integer ALU



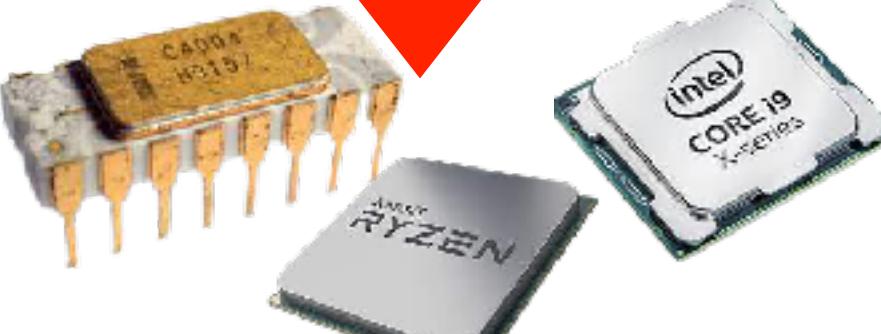
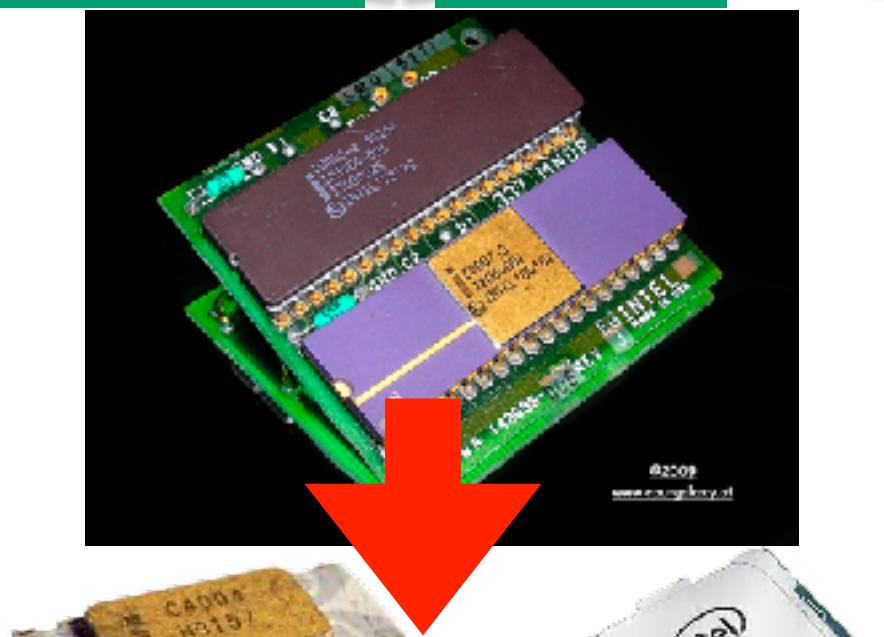
8087 FPU



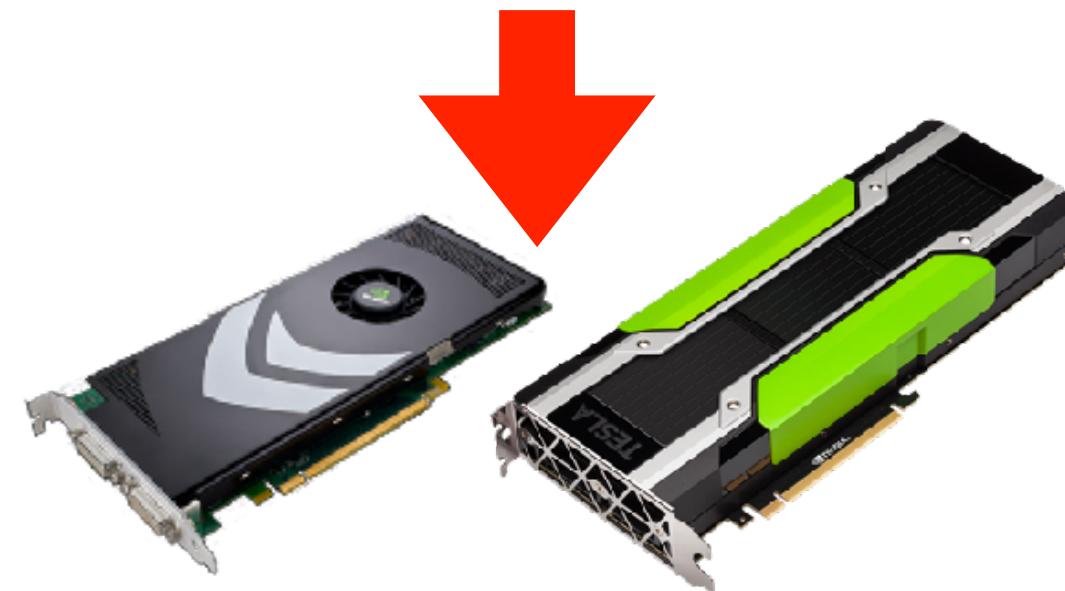
Graphics "Accelerator"



AI/ML/NN
"Accelerators"



General-purpose
microprocessors

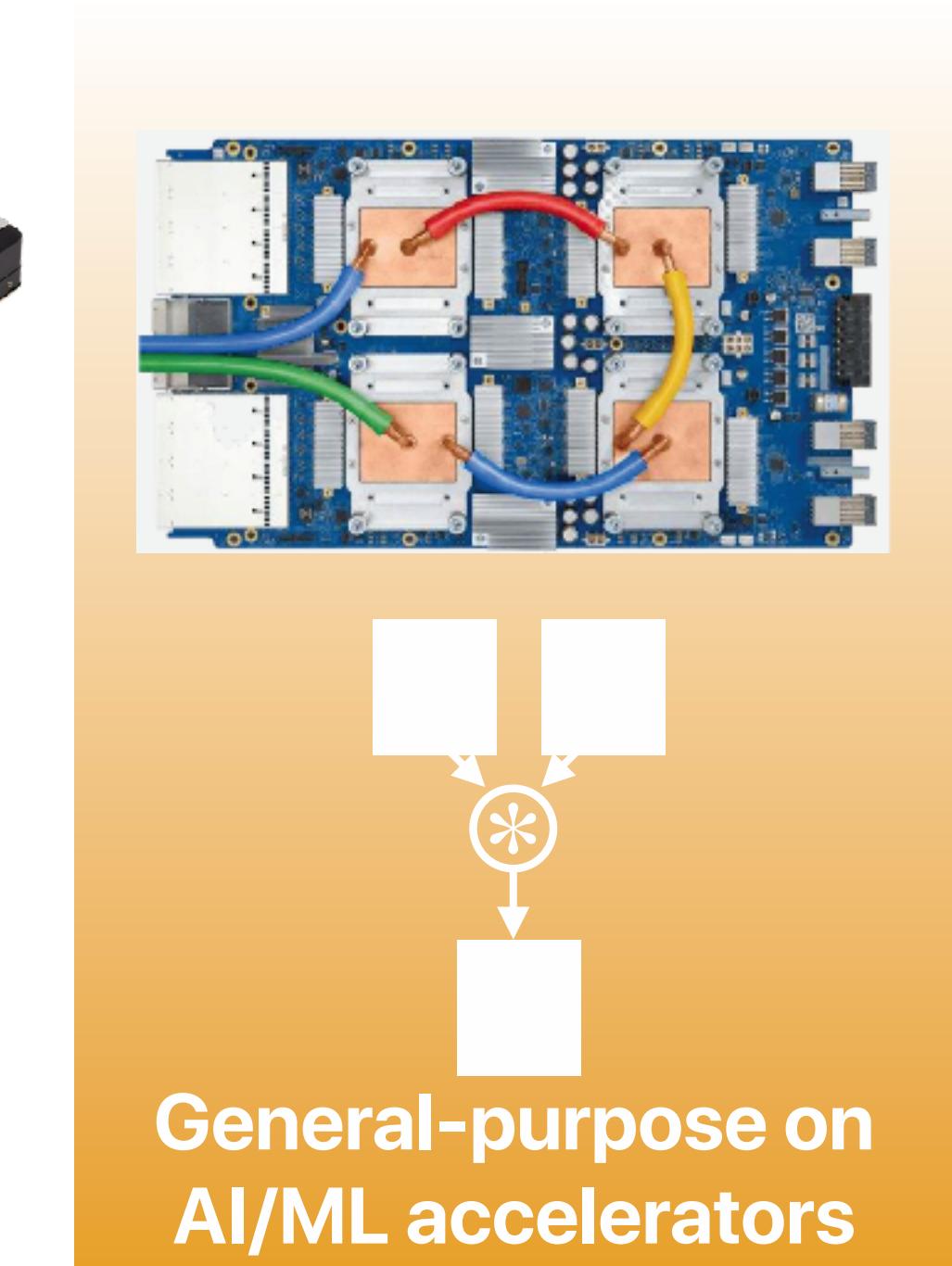
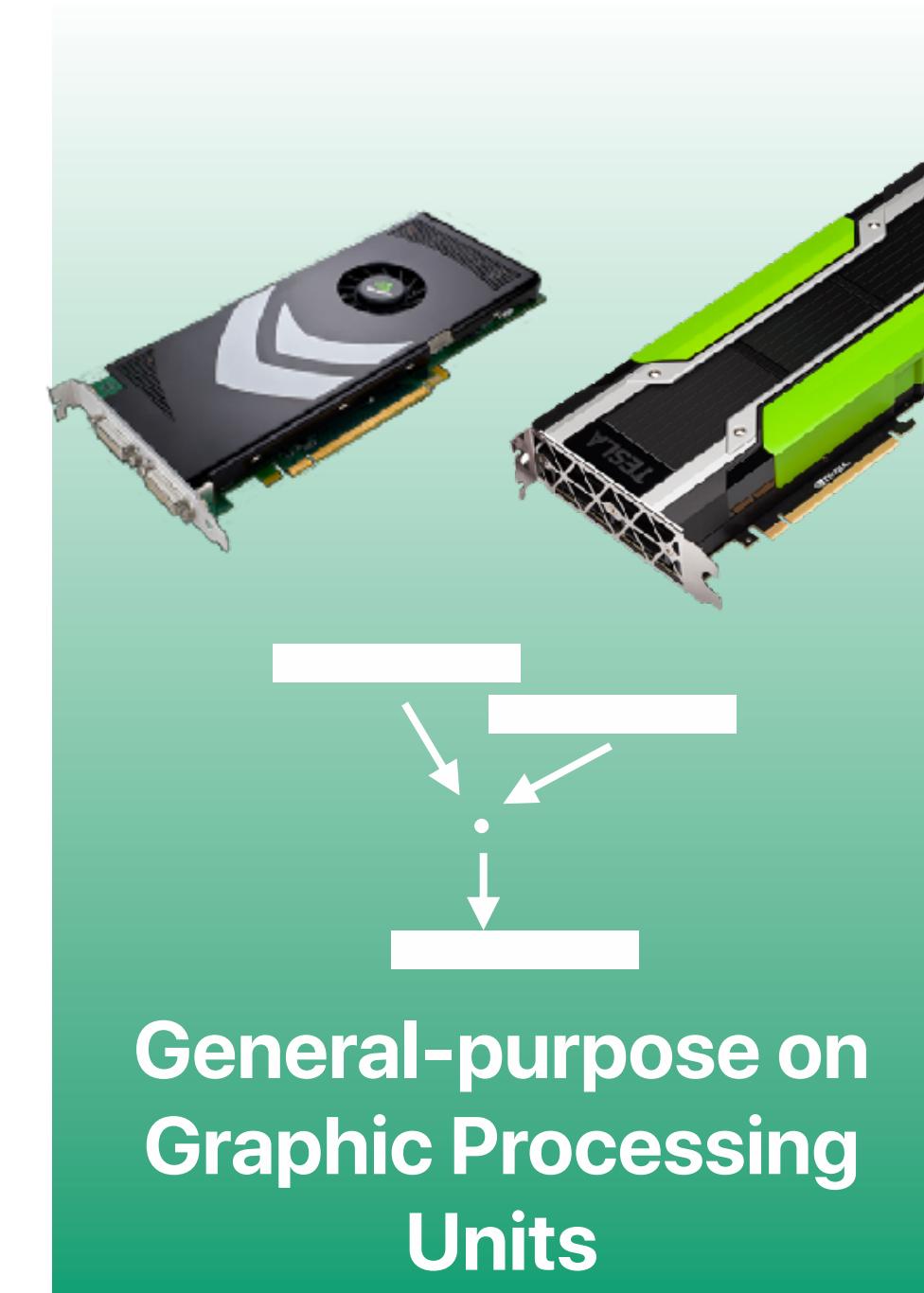
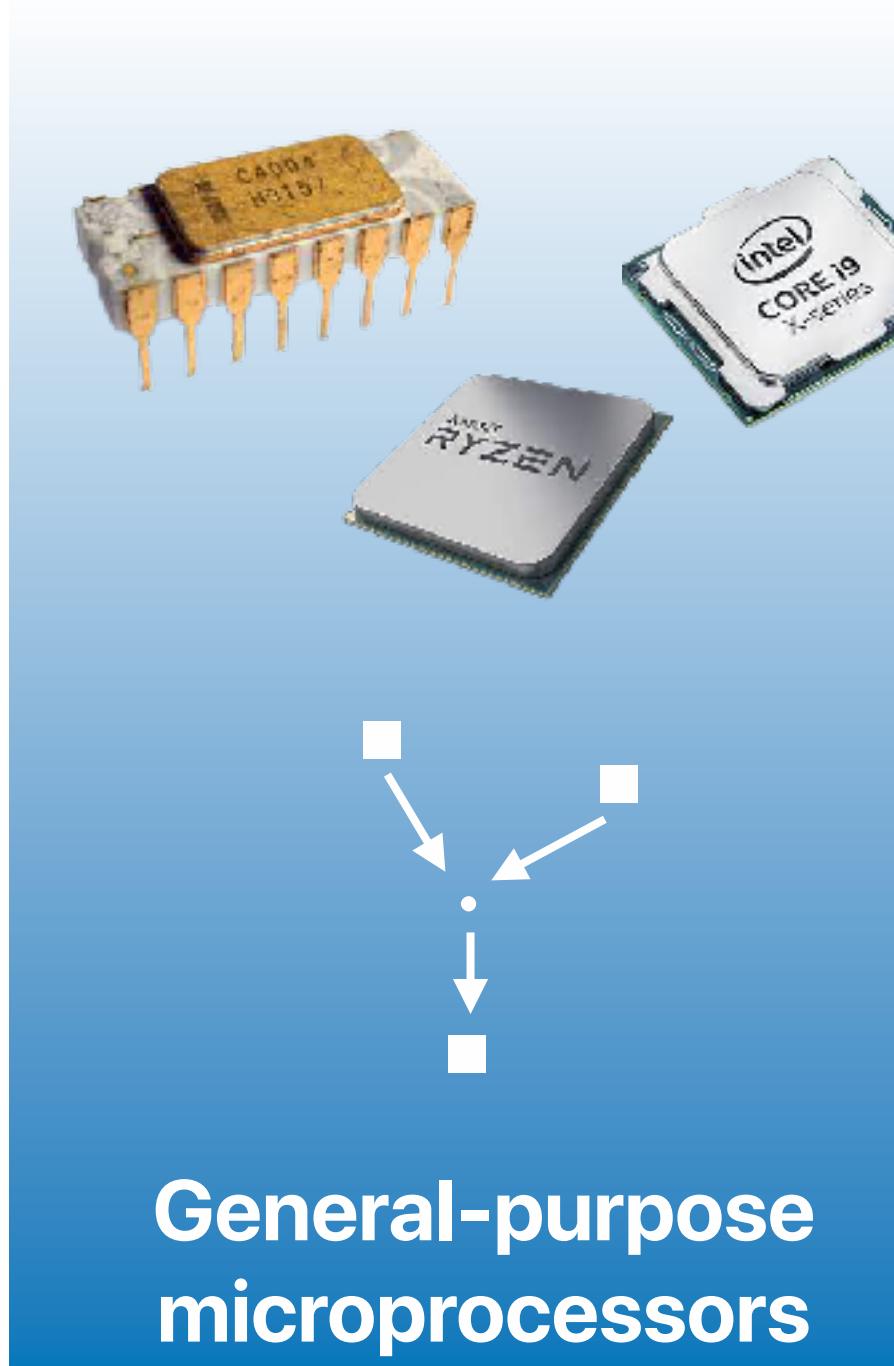


General Purpose
Computing On GPUs

General Purpose
Computing On Modern
Accelerators

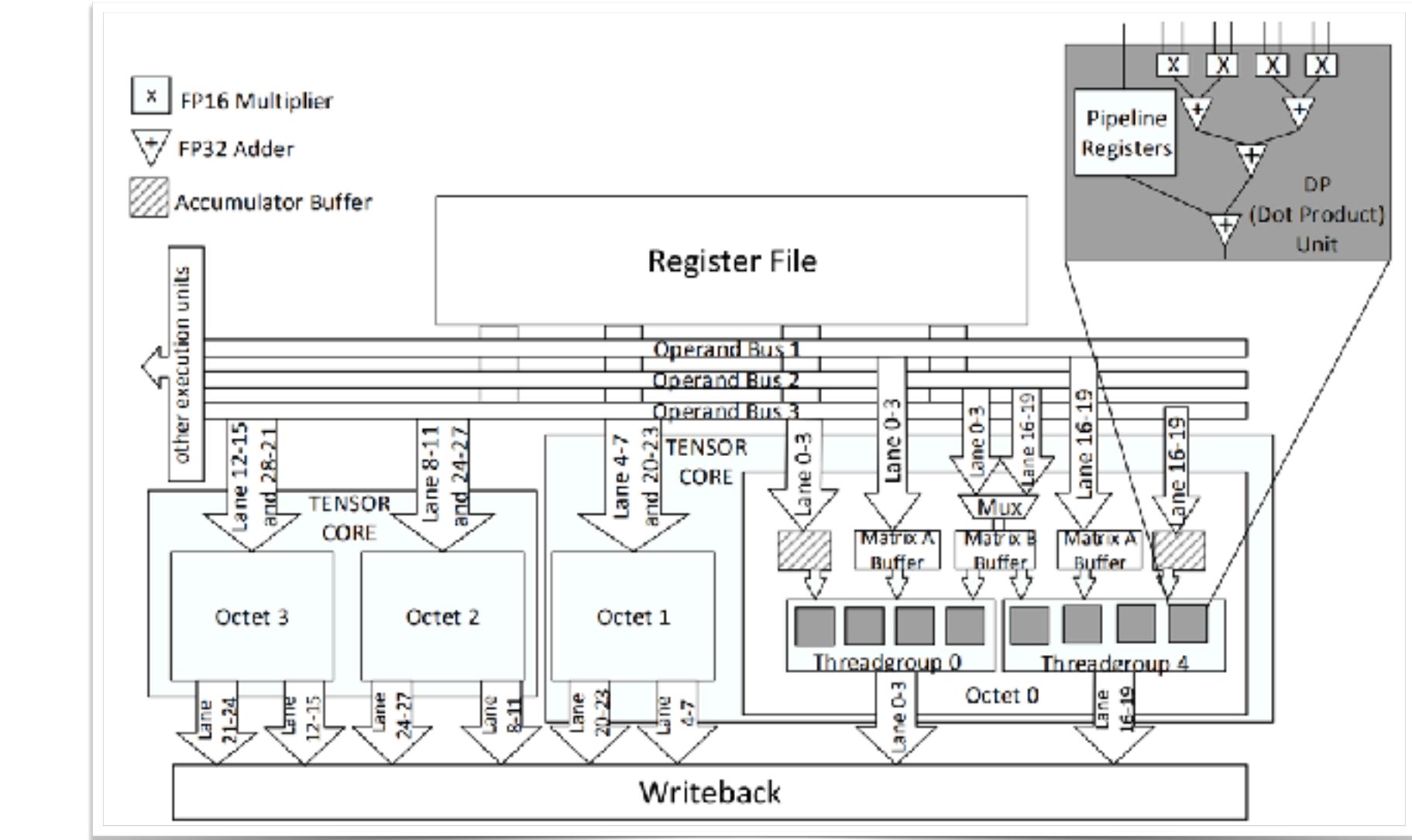
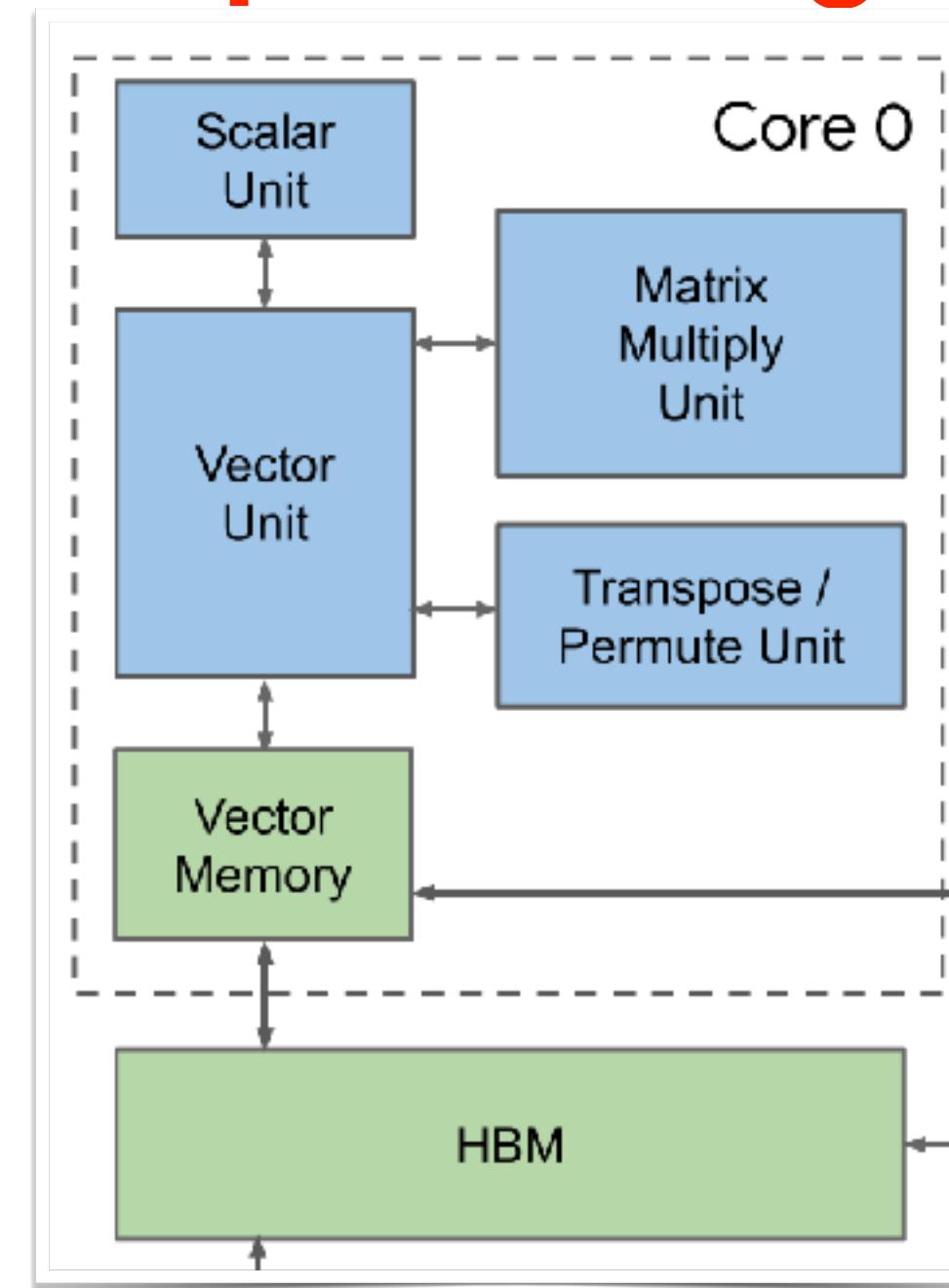


The evolutionary cycle of computing





Matrix processing — the core of AI/ML accelerators



T. Norrie et al., "The Design Process for Google's Training Chips: TPUv2 and TPUv3," in IEEE Micro, vol. 41, no. 2, pp. 56-63

M. Raihan, N. Goli and T. Aamodt, "Modeling Deep Learning Accelerator Enabled GPUs, ISPASS 2019



Accelerating Applications using Edge Tensor Processing Units

Kuan-Chieh Hsu and Hung-Wei Tseng

University of California, Riverside and Megagon Labs*

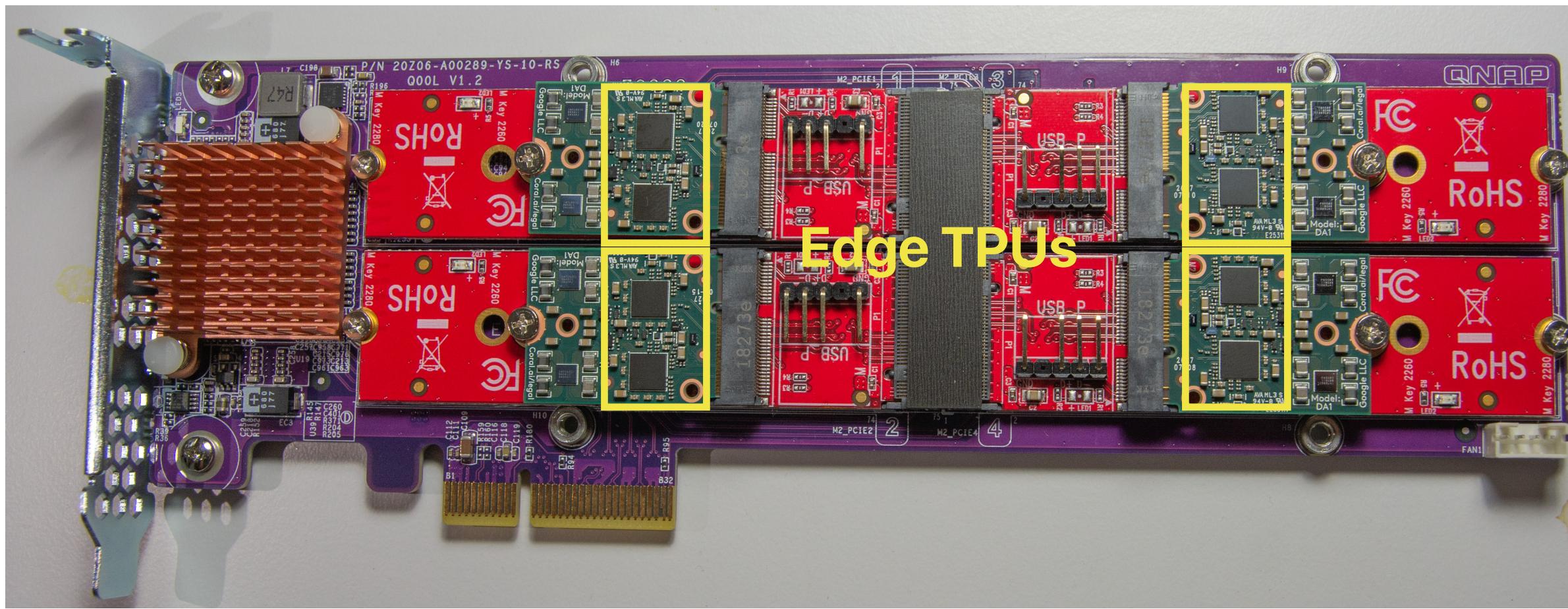
In the International Conference for High Performance
Computing, Networking, Storage and Analysis, 2021 (SC 2021)

<https://github.com/escalab/GPTPU>

ESCAL



Our GPTPU Expansion Card



- Each TPU can deliver up to 4 Trillion Operations (TOPS) with 2 Watts of Power and cost USD 29
- Each card contains 4 TPUs — 16 TOPS
- GPUs can deliver 130 TOPS — GPU is still more powerful, but more power consuming, costs a lot more



Programming Interface

Synopsis	Definition
<code>openctpu_dimension *openctpu_alloc_dimension(int dimensions, ...)</code>	This function allocates memory for input dimensions. It takes the number of dimensions and their values.
<code>openctpu_buffer_t *openctpu_create_buffer(openctpu_dimension *dimension, void *data, unsigned flags)</code>	This function creates a buffer from raw data. It takes the dimensions of the tensor and the raw data pointer.
<code>int *openctpu_enqueue(void *(func) (void *), ...)</code>	This function enqueues a task. It takes a function pointer and other parameters.
<code>int *openctpu_invoke_operator(enum tpu_ops op, unsigned flags, ...)</code>	This function invokes a TPU operator. It takes the operator type and flags.
<code>int *openctpu_sync()</code>	This function synchronizes all pending tasks.
<code>int *openctpu_wait(int task_id)</code>	This function waits for a specific task to complete.

Type	Operator
Matrix-wise	<code>conv2D</code>
Matrix-vector	<code>FullyConnected</code>
Pair-wise	<code>sub</code> <code>add</code> <code>mul</code> <code>crop</code> <code>ext</code> <code>mean</code> <code>max</code> <code>tanh</code> <code>relu</code>
Single Matrix	<code>CTPU</code>

```
#include <stdio.h>
#include <stdlib.h>
#include <gptpu.h>

// The TPU kernel
void *kernel(openctpu_buffer *matrix_a,
             openctpu_buffer *matrix_b,
             openctpu_buffer *matrix_c)
{
    // invoke the TPU operator
    openctpu_invoke_operator(conv2D, SCALE, matrix_a, \
                             matrix_b, matrix_c);

    return 0;
}

int main(int argc, char **argv)
{
    float *a, *b, *c; // pointers for raw data
    openctpu_dimension *matrix_a_d, *matrix_b_d, *matrix_c_d;
    openctpu_buffer *tensor_a, *tensor_b, *tensor_c;
    int size; // size of each dimension

    // skip: data I/O and memory allocation/initialization

    // describe a 2-D tensor (matrix) object for a
    matrix_a_d = openctpu_alloc_dimension(2, size, size);
    // describe a 2-D tensor (matrix) object for b
    matrix_b_d = openctpu_alloc_dimension(2, size, size);
    // describe a 2-D tensor (matrix) object for c
    matrix_c_d = openctpu_alloc_dimension(2, size, size);

    // create/fill the tensor a from the raw data
    tensor_a = openctpu_create_buffer(matrix_a_d, a);
    // create/fill the tensor b from the raw data
    tensor_b = openctpu_create_buffer(matrix_b_d, b);
    // create/fill the tensor c from the raw data
    tensor_c = openctpu_create_buffer(matrix_c_d, c);

    // enqueue the matrix_mul TPU kernel
    openctpu_enqueue(kernel, tensor_a, tensor_b, tensor_c);
    // synchronize/wait for all TPU kernels to complete
    openctpu_sync();

    // skip: the rest of the program
    return 0;
}
```



Our GPTPU framework

```
#include <stdio.h>
#include <stdlib.h>
#include <gptpu.h>

// The TPU kernel
void *kernel(openctpu_buffer *matrix_a,
             openctpu_buffer *matrix_b,
             openctpu_buffer *matrix_c)
{
    // invoke the TPU operator
    openctpu_invoke_operator(conv2D, SCALE, matrix_a, \
                             matrix_b, matrix_c);
    return 0;
}

int main(int argc, char **argv)
{
    float *a, *b, *c; // pointers for raw data
    openctpu_dimension *matrix_a_d, *matrix_b_d, *matrix_c_d;
    openctpu_buffer * tensor_a, * tensor_b, * tensor_c;
    int size; // size of each dimension

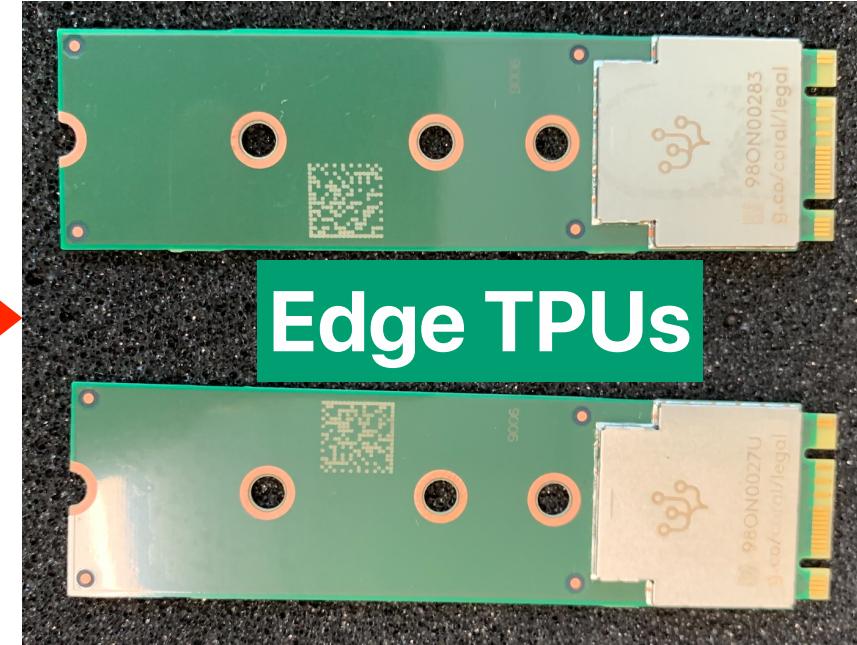
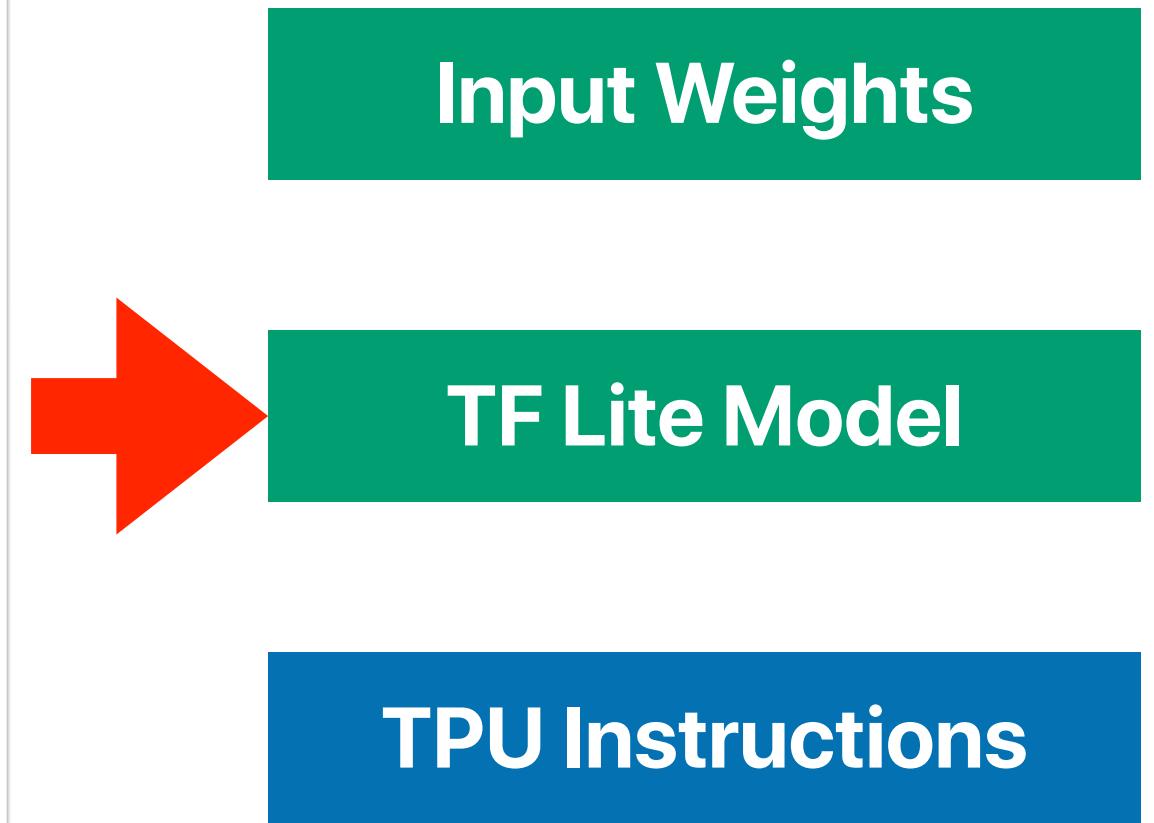
    // skip: data I/O and memory allocation/initialization

    // describe a 2-D tensor (matrix) object for a
    matrix_a_d = openctpu_alloc_dimension(2, size, size);
    // describe a 2-D tensor (matrix) object for b
    matrix_b_d = openctpu_alloc_dimension(2, size, size);
    // describe a 2-D tensor (matrix) object for c
    matrix_c_d = openctpu_alloc_dimension(2, size, size);

    // create/fill the tensor a from the raw data
    tensor_a = openctpu_create_buffer(matrix_a_d, a);
    // create/fill the tensor b from the raw data
    tensor_b = openctpu_create_buffer(matrix_b_d, b);
    // create/fill the tensor c from the raw data
    tensor_c = openctpu_create_buffer(matrix_c_d, c);

    // enqueue the matrix_mul TPU kernel
    openctpu_enqueue(kernel, tensor_a, tensor_b, tensor_c);
    // synchronize/wait for all TPU kernels to complete
    openctpu_sync();

    // skip: the rest of the program
    return 0;
}
```





Who is my most loyal customer? — conventional version

```
SELECT A.customer, B.brand, SUM(A.Quantity * B.Price) AS value
FROM A INNER JOIN B WHERE ON
A.ProductID = B.ProductID GROUP BY A.customer, B.brand;
```

A

Joined A*B

Compare and sum

Customer	ProductID	Brand	Quant.
Abe	i9-12900	Intel	1
Abe	i7-12700	Intel	1
Abe	RTX3080	NVIDIA	1
Abe	660p	Intel	2
Bob	RyZen 5800G	AMD	1
Bob	980Pro	Samsung	1
Cindy	RTX3080	NVIDIA	1
Cindy	i7-12700	Intel	2
Cindy	660p	Intel	2
Diana	RyZen 5800G	AMD	1
Diana	RX5000	AMD	1
Diana	980Pro	Samsung	1

B

Brand	ProductID	Price
Intel	i9-12900	600
Intel	i7-12700	400
Intel	660p	100
AMD	RX5000	500
AMD	RyZen 5800G	200
NVIDIA	RTX3080	1200
Samsung	980Pro	100

Customer	ProductID	Brand	Quant.	Price	Value
Abe	i9-12900	Intel	1	600	600
Abe	i7-12700	Intel	1	400	400
Abe	RTX3080	NVIDIA	1	1200	1200
Abe	660p	Intel	2	100	200
Bob	RyZen 5800G	AMD	1	400	400
Bob	980Pro	Samsung	1	100	100
Cindy	RTX3080	NVIDIA	1	1200	1200
Cindy	i7-12700	Intel	2	400	800
Cindy	660p	Intel	2	100	200
Diana	RyZen 5800G	AMD	1	400	400
Diana	RX5000	AMD	1	500	500
Diana	980Pro	Samsung	1	100	100

Customer	Brand	Value
Abe	Intel	1200
Abe	NVIDIA	1200
Bob	AMD	400
Bob	Samsung	100
Cindy	Intel	1000
Cindy	NVIDIA	1200
Diana	AMD	900
Diana	Samsung	100

Memory copy

Compare & Element-wise Memory copy

Vector multiplications



Who is my most loyal customer? — Matrix Version

```
SELECT A.customer, B.brand, SUM(A.Quantity * B.Price) AS value
FROM A INNER JOIN B
WHERE ON A.ProductID = B.ProductID
GROUP BY A.customer, B.brand;
```

Customer	ProductID	Brand	Quant.
Abe	i9-12900	Intel	1
Abe	i7-12700	Intel	1
Abe	RTX3080	NVIDIA	1
Abe	660p	Intel	2
Bob	RyZen 5800G	AMD	1
Bob	980Pro	Samsung	1
Cindy	RTX3080	NVIDIA	1
Cindy	i7-12700	Intel	2
Cindy	660p	Intel	2
Diana	RyZen 5800G	AMD	1
Diana	RX5000	AMD	1
Diana	980Pro	Samsung	1

B

Brand	ProductID	Price
Intel	i9-12900	600
Intel	i7-12700	400
Intel	660p	100
AMD	RX5000	500
AMD	RyZen 5800G	200
NVIDIA	RTX3080	1200
Samsung	980Pro	100

Customer	Intel	AMD	NVIDIA	Samsung
Abe	1200	0	1200	0
Bob	0	400	0	100
Cindy	1000	0	1200	0
Diana	0	900	0	100

Matrix multiplications

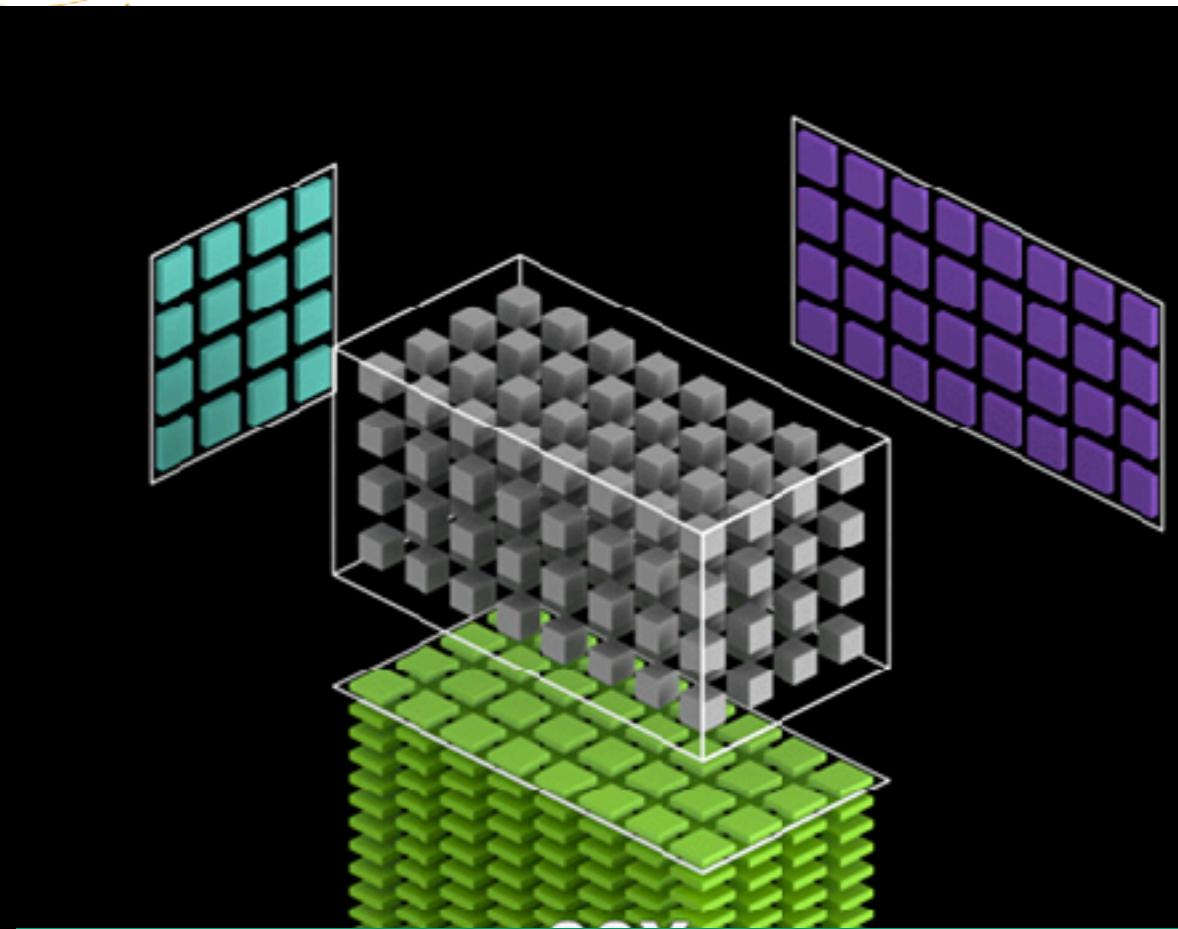
Customer/ ProductID	i9-1290	i7-1270	660	RX500	RyZen 5800G	RTX308	980Pr
	0	0	p	0	0	0	o
Abe	1	1	2	0	0	1	0
Bob	0	0	0	0	1	0	1
Cindy	0	2	2	0	0	1	0
Diana	0	0	0	1	1	0	1

ProductID/ Brand	Intel	AMD	NVIDIA	Samsung
i9-12900	600	0	0	0
i7-12700	400	0	0	0
660p	100	0	0	0
RX5000	0	500	0	0
RyZen 5800G	0	400	0	0
RTX3080	0	0	1200	0
980Pro	0	0	0	100

Memory copy



How can Tensor Cores faster despite higher algorithm complexity?



Each tensor core operation performs 8x4x4 MMA in one cycle
~ 256 FP operations in conventional scalar models

RTX 3090	
CUDA Cores	10496
Tensor Cores	328

FLOPS of 8K by 8K matrix multiplications =
 $8192 \times 8192 \times 8192 \times 2$

$$\frac{8192 \times 8192 \times 8192 \times 2}{10496} = 104,755,300 \text{ CUDA core cycles}$$

$$\frac{8192 \times 8192 \times 8192 \times 2}{328 \times 256} = 13,094,413 \text{ Tensor core cycles}$$



Tensor cores are 8x faster



TCUDB: Accelerating Database with Tensor Processors

Yu-Ching Hu, Yuliang Li* and Hung-Wei Tseng
University of California, Riverside and Megagon Labs*
In The ACM SIGMOD/PODS 2022 International Conference
on Management of Data.
<https://github.com/escalab/TCUDB>



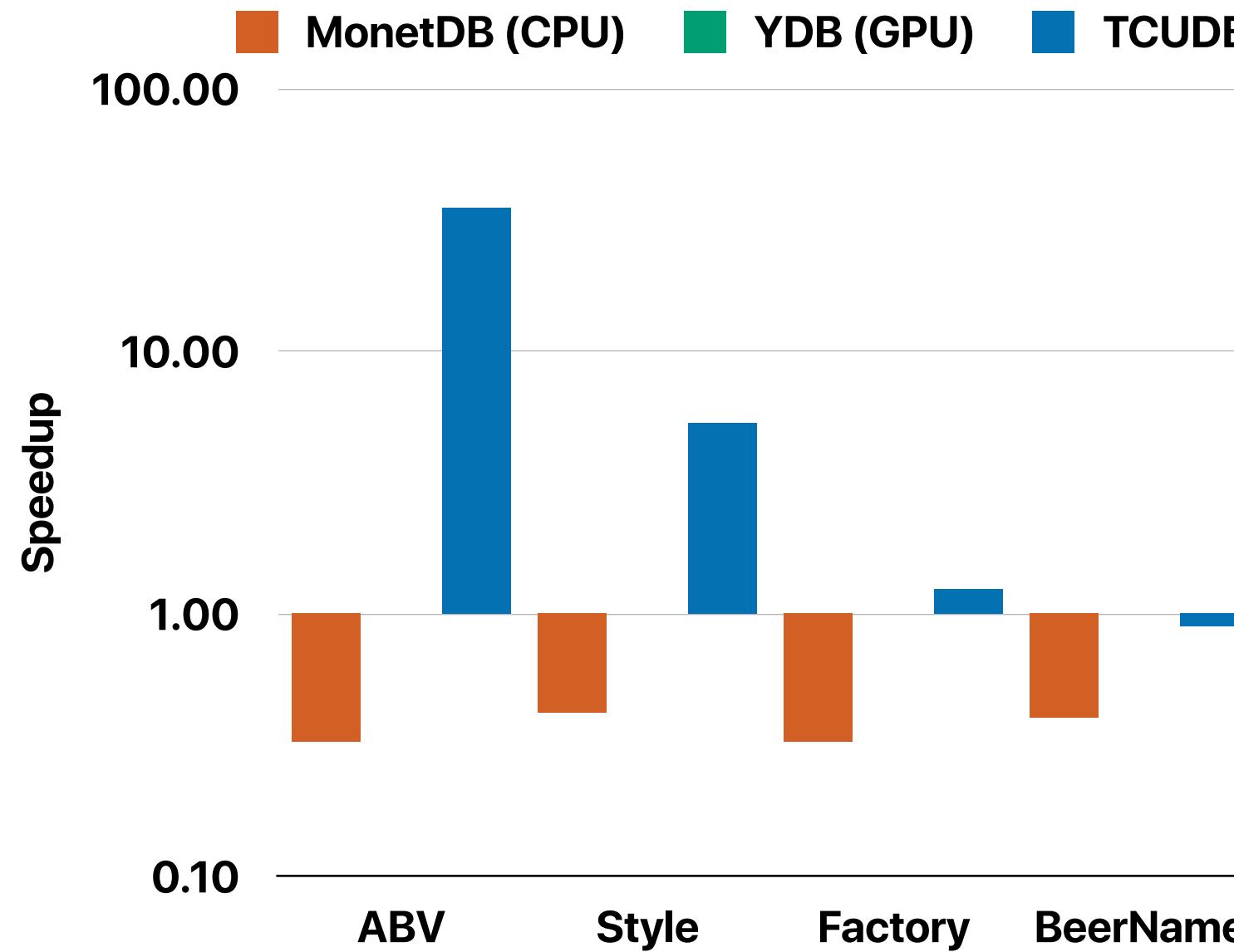
Megagon Labs

ESCAL

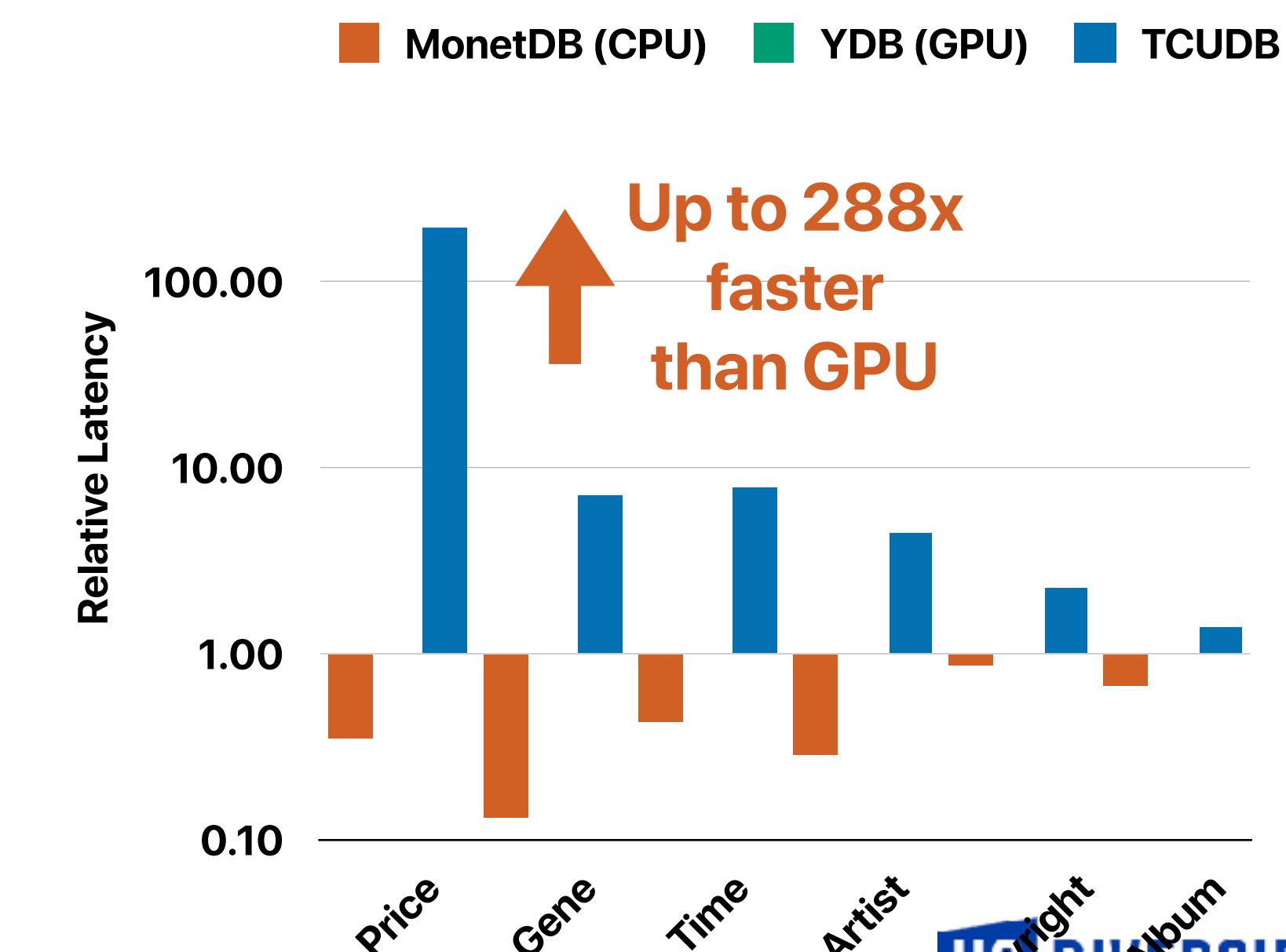


Entity matching

- `SELECT TABLE_A.ID, TABLE_A.BEER_NAME, TABLE_B.ID, TABLE_B.BEER_NAME FROM TABLE_A, TABLE_B WHERE TABLE_A.ABV = TABLE_B.ABV;`



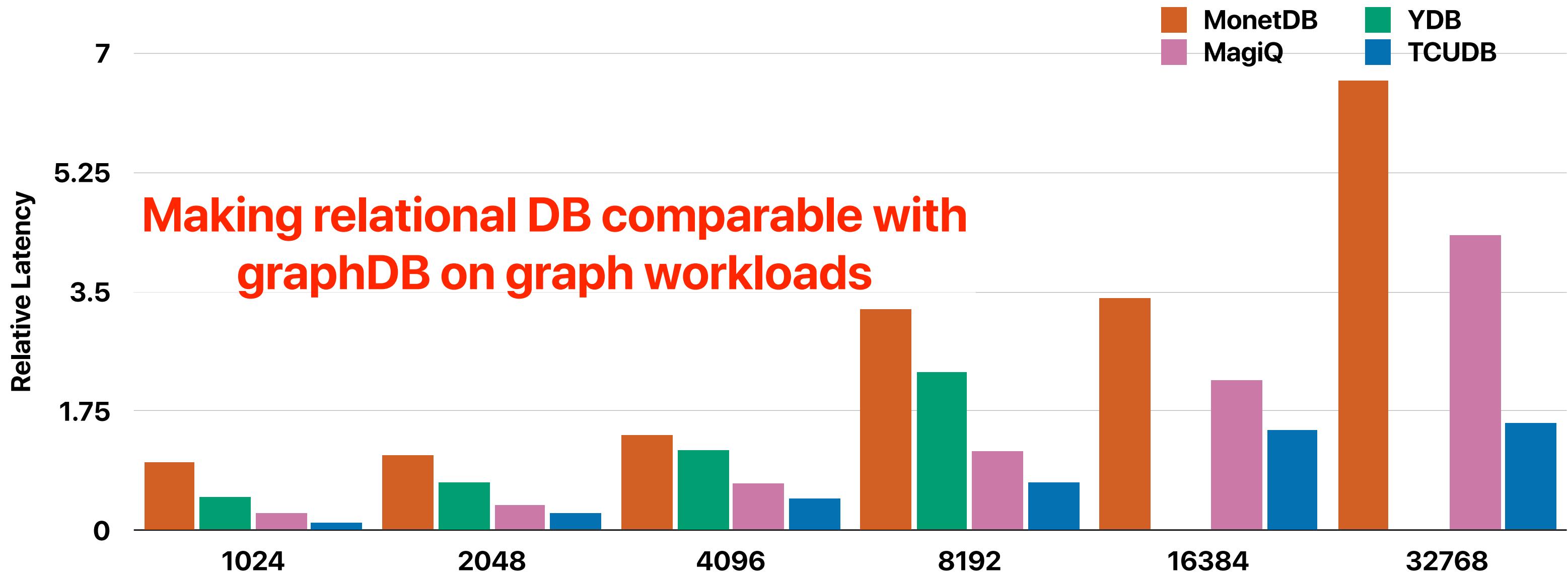
- `SELECT TABLE_A.ID, TABLE_A.SONG, TABLE_B.ID, TABLE_B.SONG FROM TABLE_A, TABLE_B WHERE TABLE_A.ARTIST = TABLE_B.ARTIST;`





PageRank

```
●SELECT SUM(@alpha * PAGERANK.rank / OUTDEGREE.DEGREE) + (1-@alpha)/  
@num_node FROM PAGERANK, OUTDEGREE WHERE PAGERANK.ID = OUTDEGREE.ID;
```



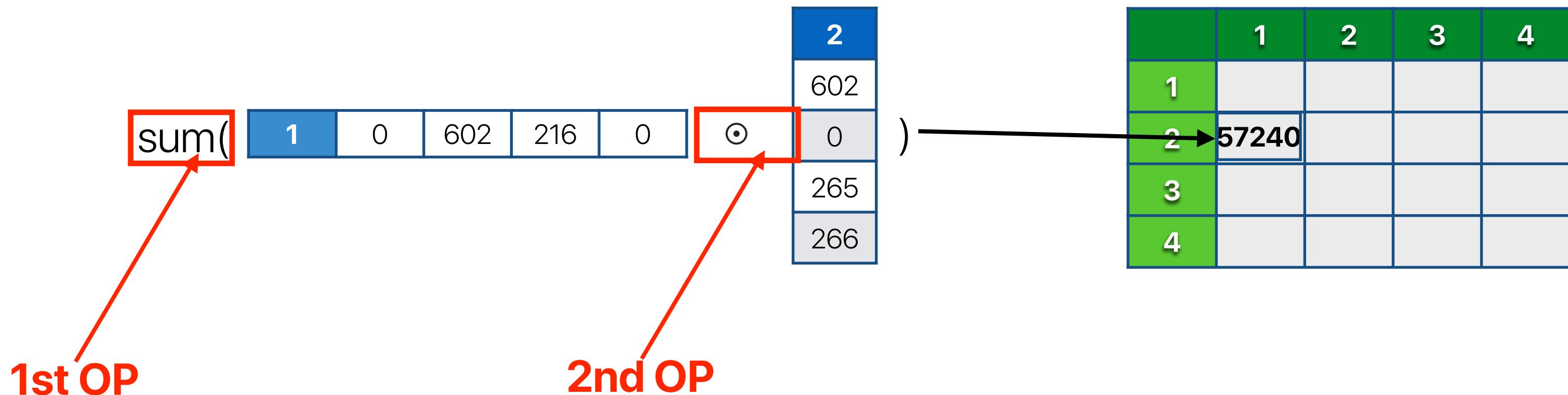


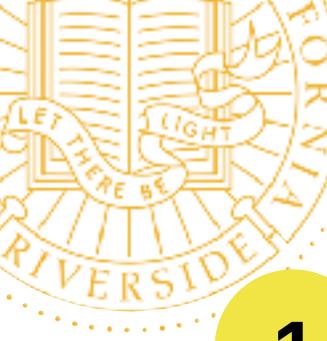
Recap: Matrix Multiplications

	1	2	3	4
1	0	602	216	0
2	602	0	265	266
3	216	265	0	218
4	0	266	218	0

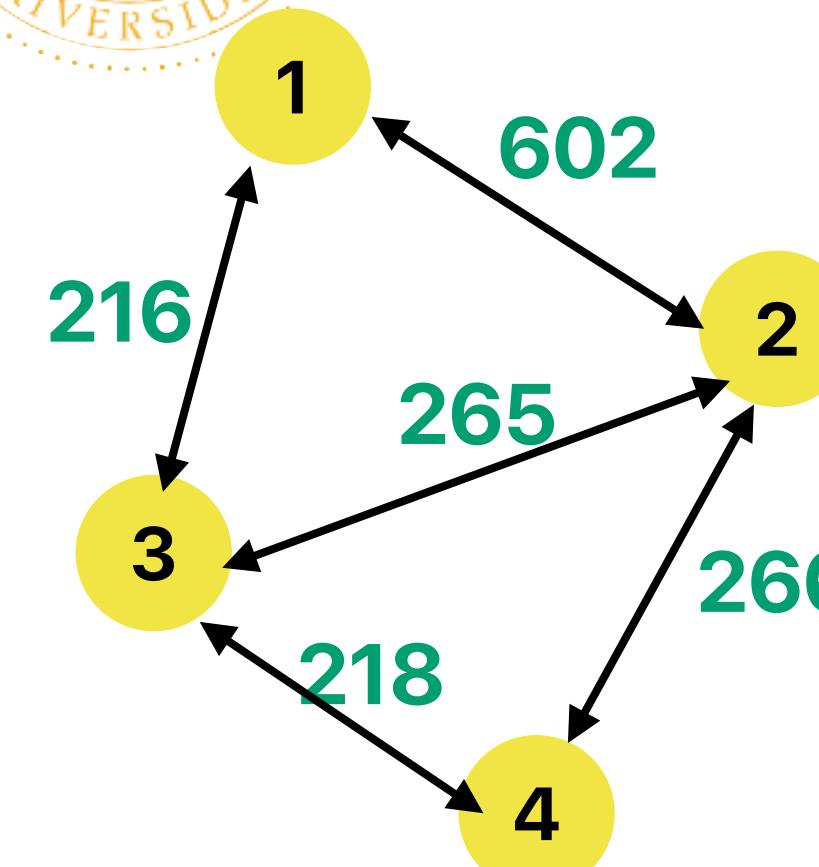
×

	1	2	3	4
1	0	602	216	0
2	602	0	265	266
3	216	265	0	218
4	0	266	218	0





Recap: All-Pair-Shortest-Path



	1	2	3	4
1	—	602	216	∞
2	602	—	265	266
3	216	265	—	218
4	∞	266	218	—

	1	2	3	4
1	—	602	216	∞
2	602	—	265	266
3	216	265	—	218
4	∞	266	218	—

1st OP

$\min($

1

—

602

216

∞

)

+

2	602
	—
	265
	266

2nd OP

87

	1	2	3	4
1	—			
2	2	—		
3				
4				



Similarity of matrix problems

	1st OP	2nd OP
Matrix Multiplications	+	×
All pair shortest path	min	+
Critical path	max	+
Minimum reliability paths	min	×
Maximum reliability paths	max	×
Minimum spanning tree	min	max
Maximum capacity paths	max	min
Transitive and reflexive closure	or	and
L2 Distance	+	$ a-b ^2$

TensorCV: Accelerating ML-adjacent Computation Using Tensor Processors

Dongho Ha⁺, Won Woo Ro^{*} and Hung-Wei Tseng⁺

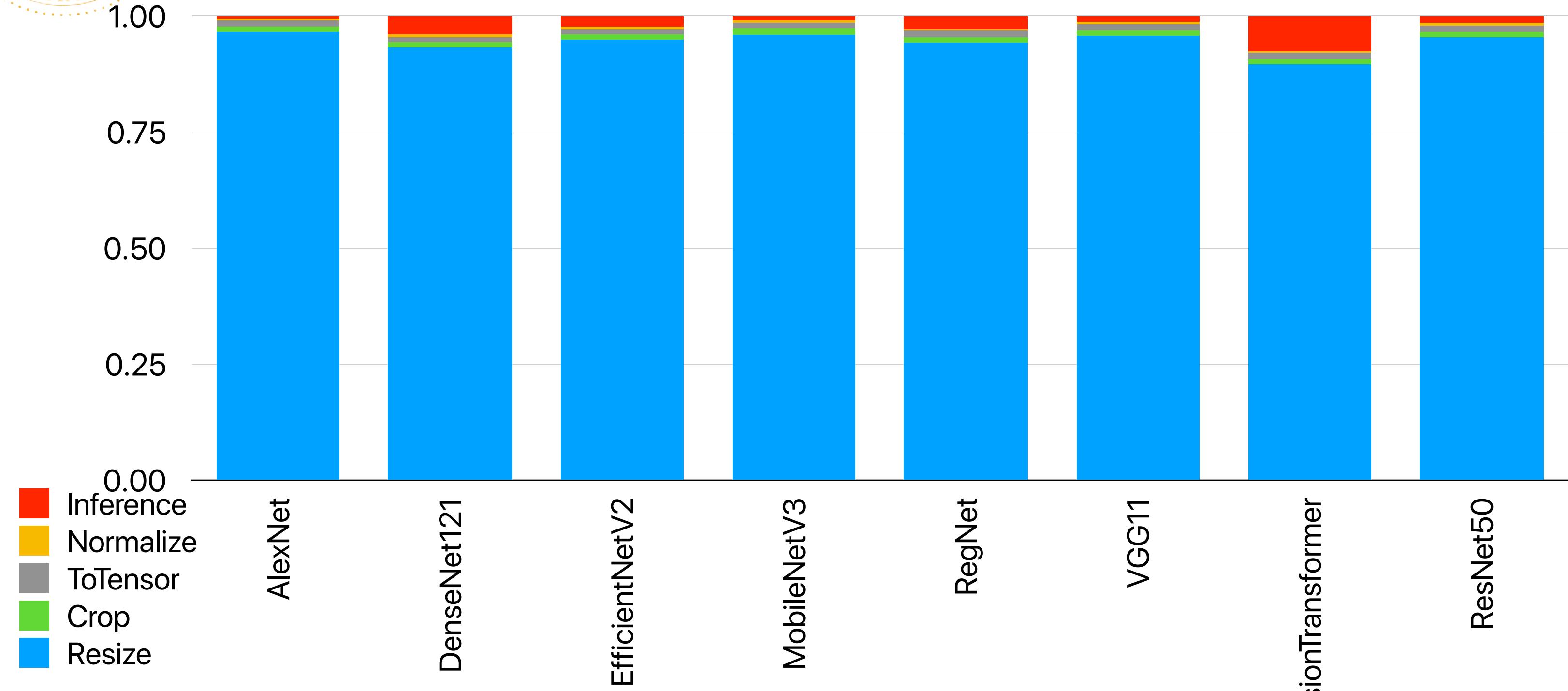
Yonsei University^{*} and University of California, Riverside⁺

In International Symposium on Low Power Electronics and Design (ISLPED 2023)



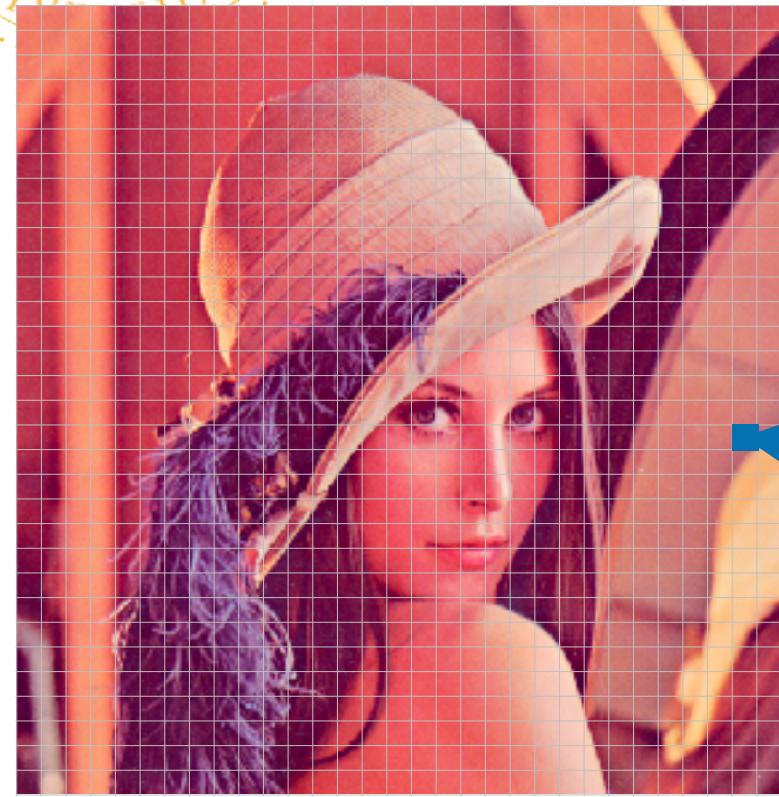


Image pre-processing is the new bottleneck





Traditional image pre-processing



Pixel position: (i, j)

Pixel value: (R_{ij}, G_{ij}, B_{ij})

$$\begin{bmatrix} i_{rot} \\ j_{rot} \end{bmatrix} = \begin{bmatrix} i \\ j \end{bmatrix} \begin{bmatrix} \cos(\theta) & -\sin(\theta) \\ \sin(\theta) & \cos(\theta) \end{bmatrix}$$

Matrix-vector multiplications

$$Rotated_image_{i_{rot},j_{rot}} = (R_{ij}, G_{ij}, B_{ij})$$

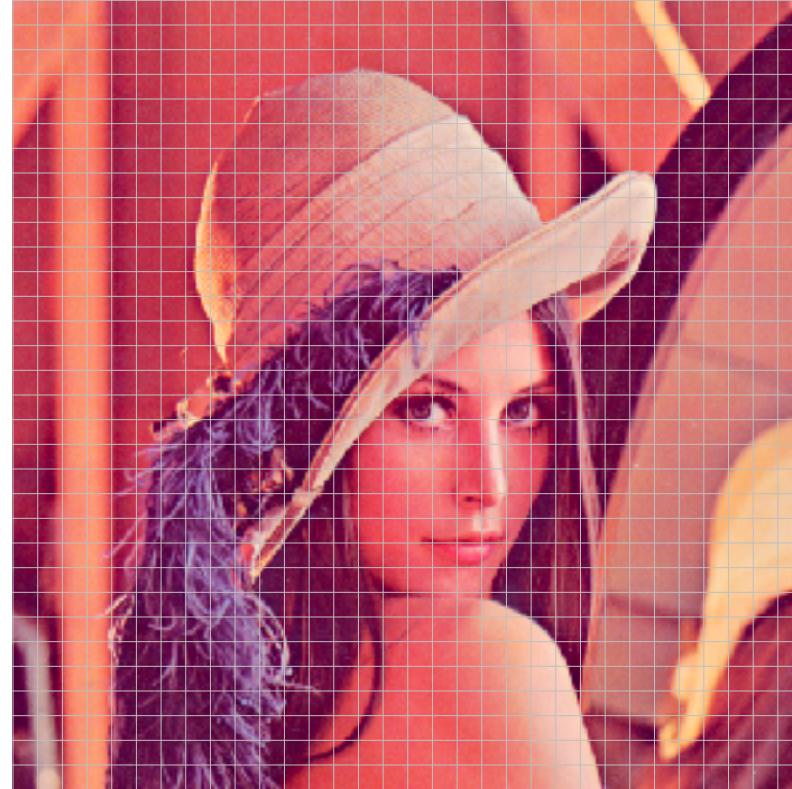
Memory operation



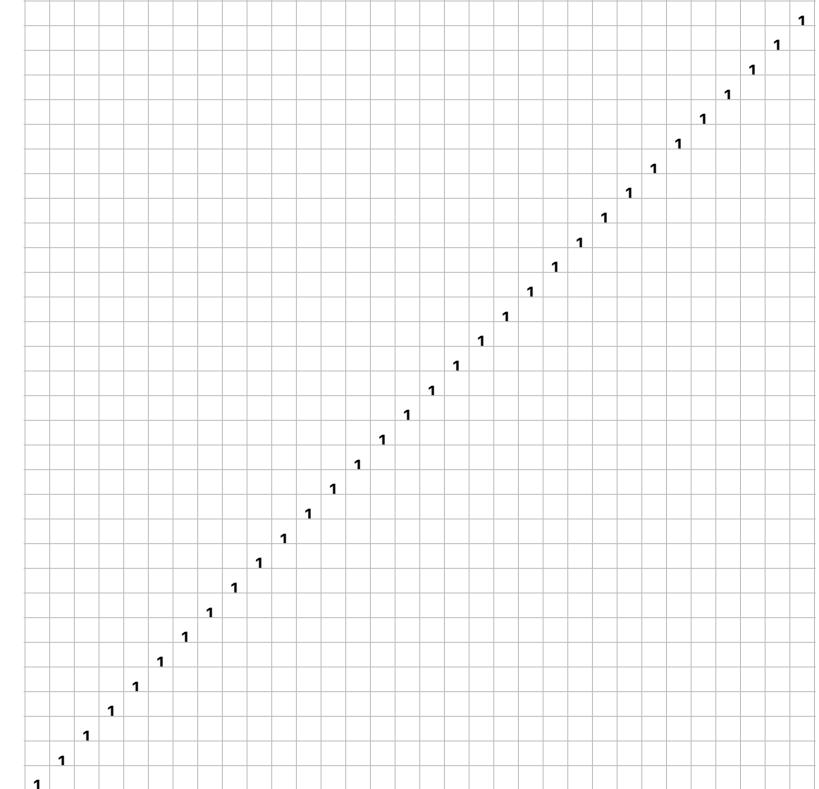
$$\begin{bmatrix} i_{rot} \\ j_{rot} \end{bmatrix} = \begin{bmatrix} i \\ j \end{bmatrix} \begin{bmatrix} 0 & -1 \\ 1 & 0 \end{bmatrix}$$



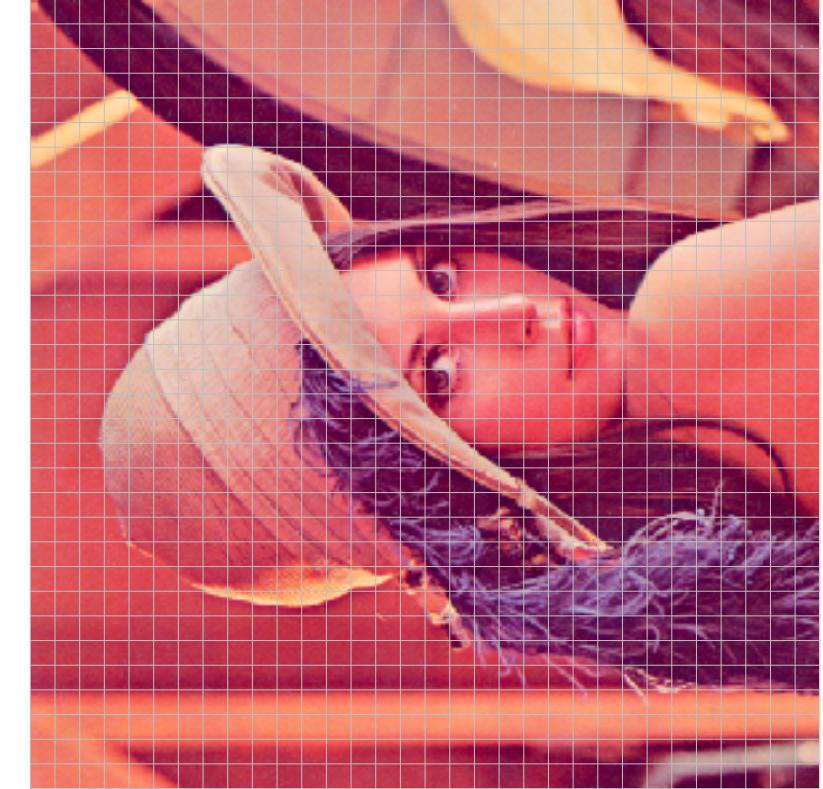
"Tensorized" image processing algorithm



X



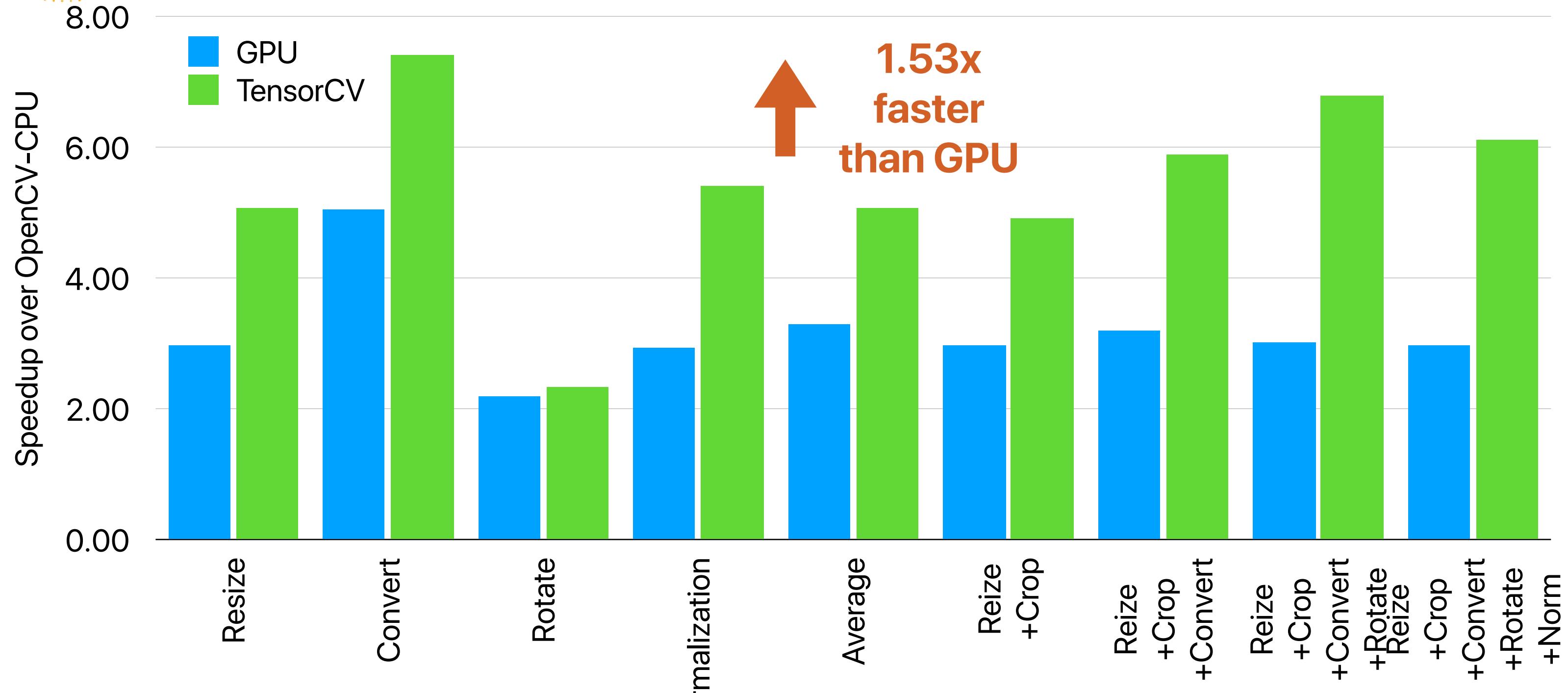
=



Only 1 mac operation in Tensor Processors!



Speedup over CPU and GPU



SIMD²: A Generalized Matrix Instruction Set for Accelerating Tensor Computation beyond GEM

Yunan Zhang, Po-An Tsai and Hung-Wei Tseng
In The International Symposium on Computer
Architecture (ISCA). 2022

<https://github.com/escalab/SIMD2>



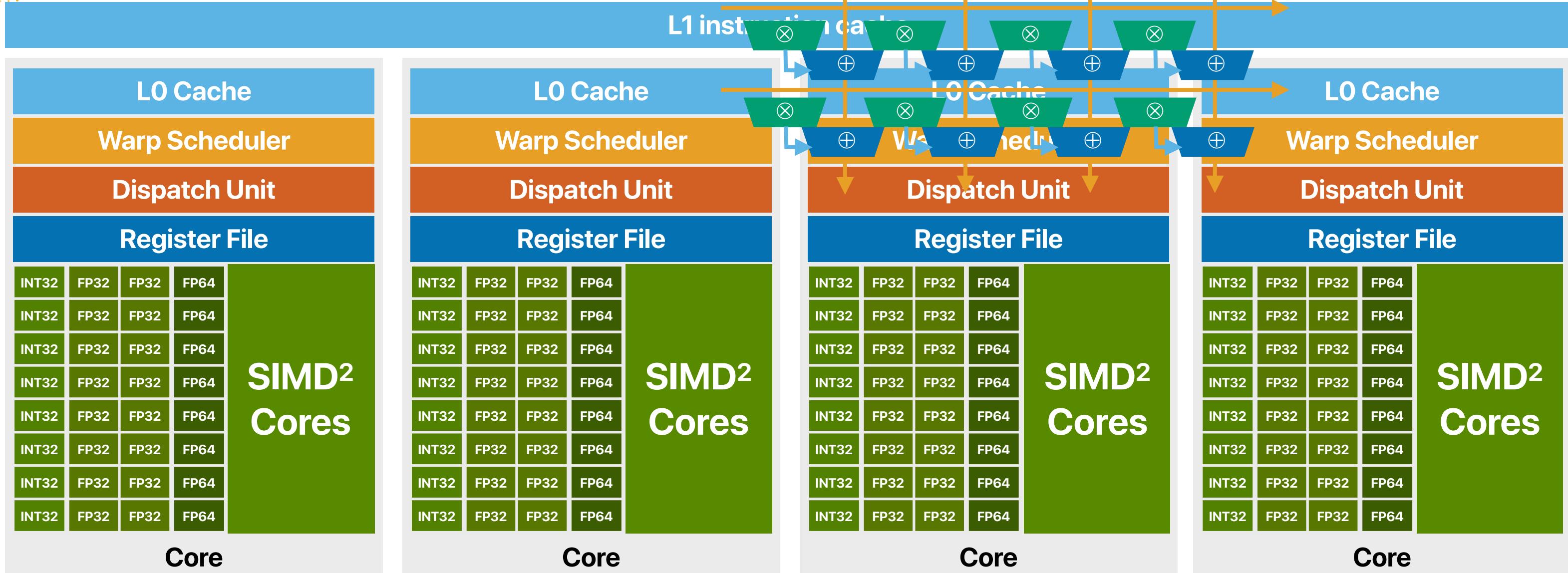


SIMD² instructions

	1st OP	2nd OP
Matrix Multiplications	+	×
All pair shortest path	min	+
Critical path	max	+
Minimum reliability paths	min	×
Maximum reliability paths	max	×
Minimum spanning tree	min	max
Maximum capacity paths	max	min
Transitive and reflexive closure	or	and
L2 Distance	+	$ a-b ^2$

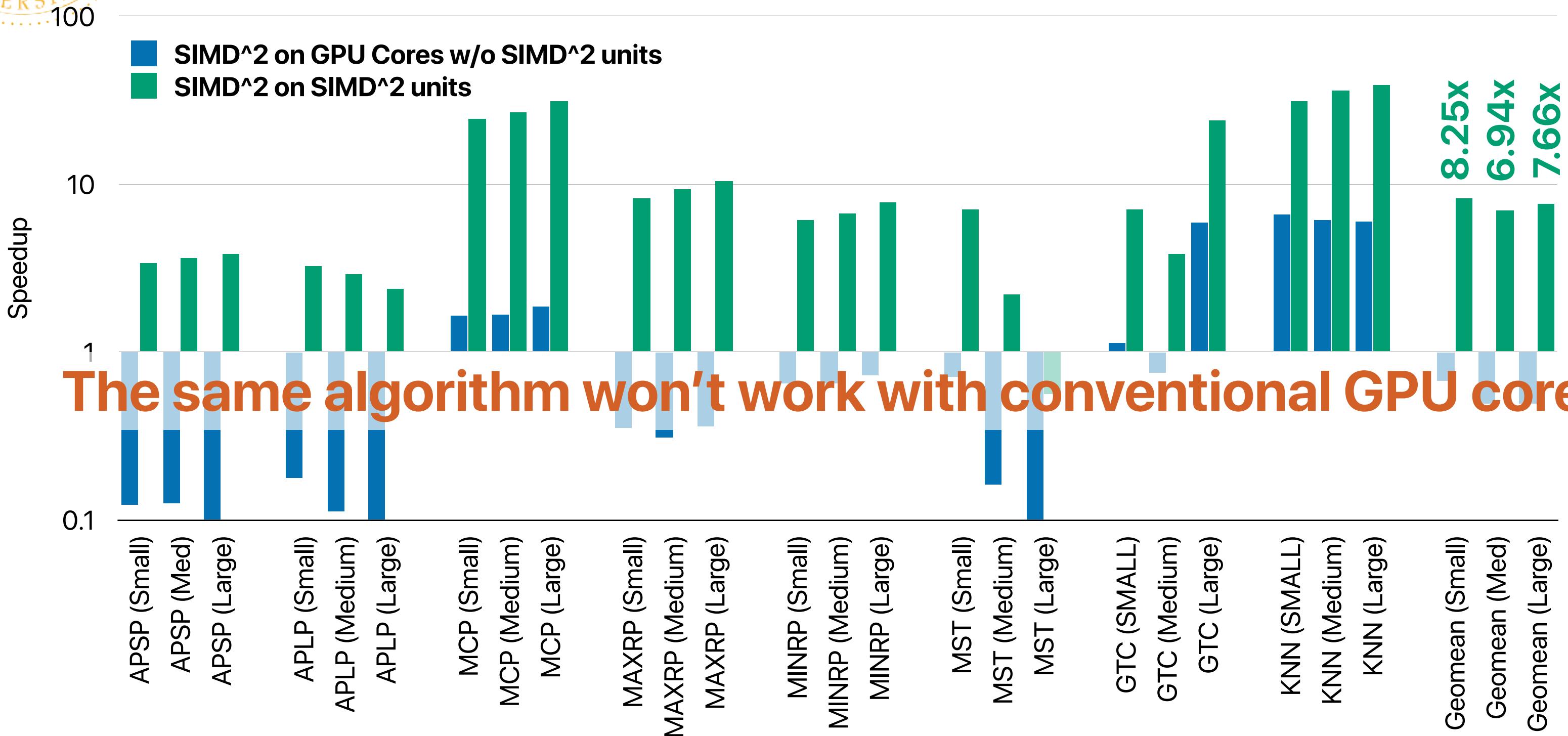


Integration of SIMD2





SIMD² performance



Final words

Conclusion

- Computer architecture is now more important than you could ever imagine in the dark silicon era + the “new” golden age of computer architecture
- Just being a “programmer” is easy. You need to know architecture a lot to be a “performance programmer”
 - Optimizing locality and footprint for caching
 - Make your code or data more branch prediction friendly
 - Exploiting more instruction-level parallelism via carefully revisiting the critical path of execution
- Multicore era — to get your multithreaded program correct and perform well, you need to take care of coherence and consistency
- We’re now in the “dark silicon era”
 - Single-core isn’t getting any faster
 - Multi-core doesn’t scale anymore
 - We will see more and more ASICs
 - You need to write more “system-level” programs to use these new ASICs.
- Can we be more creative in using ASICs?
- If you’re interested in more about the “Golden Age of Computer Architecture” — take EE/CS277 in Winter!

**Thank you all for this great quarter!
Let's take a group photo now!**

Computer Science & Engineering

203



One more thing...

Sample Final

Format of the online final

- Multiple choices (15 questions)
 - They're like your clicker/midterm multiple choices questions
 - Cumulative, don't forget your midterm and midterm review
- Problem sets * 3 problem sets,
- Open-ended * 5 questions in total
 - Explain your answer clearly and "correctly". If you include "incorrect" information in your answer, it's going to hurt your grade
 - May not have a standard answer. You need to understand the concepts to provide a good answer
 - You should review your assignments carefully
 - You should read the papers + the suggested ones

Multiple choices

How many dependencies do we have?

- How many pairs of data dependences are there in the following x86 instructions?

```
movl    (%rdi), %eax
xorl    (%rsi), %eax
movl    %eax, (%rdi)
xorl    (%rsi), %eax
movl    %eax, (%rsi)
xorl    %eax, (%rdi)
```

- A. 1
- B. 2
- C. 3
- D. 4
- E. 5

The effect of code optimization

- By reordering which pair of the following instruction stream can we eliminate all stalls without affecting the correctness of the code?
 - ① `movl (%rdi), %ecx`
 - ② `addl %ecx, %eax`
 - ③ `addq $4, %rdi`
 - ④ `cmpq %rdx, %rdi`
 - ⑤ `jne .L3`
 - ⑥ `ret`
 - A. (1) & (2)
 - B. (2) & (3)
 - C. (3) & (4)
 - D. (4) & (5)
 - E. None of the pairs can be reordered

Pros/Cons of SMT

- How many of the following statements about SMT is/are correct?
 - ① SMT makes processors with deep pipelines more tolerable to mis-predicted branches
 - ② SMT can improve the throughput of a single-threaded application
 - ③ SMT processors can better utilize hardware during cache misses comparing with superscalar processors with the same issue width
 - ④ SMT processors can have higher cache miss rates comparing with superscalar processors with the same cache sizes when executing the same set of applications.
 - A. 0
 - B. 1
 - C. 2
 - D. 3
 - E. 4

Linked list v.s. arrays

- We can use either a linked list or an array to store a list of data, compare the performance of the array (version A) and linked list (version B) implementations that can achieve the same outcome as below. Assume we have a processor with a reasonably good branch predictor and unlimited fetch/issue width, please identify the correct statements.
 - If the dataset is large, the A will outperform the B
 - If the dataset is small, there is very little performance difference between A and B
 - If the dataset is small, B will outperform A as A has more branch instructions
 - If the dataset is small, A will outperform B as A has fewer data dependencies

- A. 0
- B. 1
- C. 2
- D. 3
- E. 4

A

```
for(i=0;i<size;i++)
{
    if(node[i].next)
        number_of_nodes++;
}
```

B

```
while(node)
{
    node = node->next;
    number_of_nodes++;
}
```

Why is C better than B?

- How many of the following statements explains the reason why B outperforms C with compiler optimizations
 - C has lower dynamic instruction count than B
 - C has significantly lower branch mis-prediction rate than B
 - C has significantly fewer branch instructions than B
 - C has better CPI than B

A. 0

B. 1

C. 2

D. 3

E. 4



```
inline int __popcount(uint64_t x) {
    int c = 0;
    int table[16] = {0, 1, 1, 2, 1,
                     2, 2, 3, 1, 2, 2, 3, 2, 3, 3, 4};
    while(x) {
        c += table[(x & 0xF)];
        x = x >> 4;
    }
    return c;
}
```



```
inline int __popcount(uint64_t x) {
    int c = 0;
    while(x) {
        c += x & 1;
        x = x >> 1;
        c += x & 1;
        x = x >> 1;
        c += x & 1;
        x = x >> 1;
        c += x & 1;
        x = x >> 1;
    }
    return c;
}
```

Amdahl's Law on Multicore Architectures

- Regarding Amdahl's Law on multicore architectures, how many of the following statements is/are correct?
 - ① If we have unlimited parallelism, the performance of each parallel piece does not matter as long as the performance slowdown in each piece is bounded
 - ② With unlimited amount of parallel hardware units, single-core performance does not matter anymore
 - ③ With unlimited amount of parallel hardware units, the maximum speedup will be bounded by the fraction of parallel parts
 - ④ With unlimited amount of parallel hardware units, the effect of scheduling and data exchange overhead is minor

A. 0
B. 1
C. 2
D. 3
E. 4

Virtual indexed, physical tagged cache limits the cache size

- If you want to build a virtual indexed, physical tagged cache with 32KB capacity, which of the following configuration is possible? Assume the system use 4K pages.
 - A. 32B blocks, 2-way
 - B. 32B blocks, 4-way
 - C. 64B blocks, 4-way
 - D. 64B blocks, 8-way

Power & Energy

- Regarding dynamic frequency scaling, how many of the following statements are correct?
 - ① Lowering the frequency helps extending the battery life
 - ② Lowering the frequency helps reducing the heat generation
 - ③ Lowering the frequency helps reducing the electricity bill
 - ④ A CPU operating at 25% of the peak frequency can still consume more than 50% of the peak power

A. 0

B. 1

C. 2

D. 3

E. 4

Why is D better than C?

- How many of the following statements explains the main reason why B outperforms C with compiler optimizations
 - D has lower dynamic instruction count than C
 - D has significantly lower branch mis-prediction rate than C
 - D has significantly fewer branch instructions than C
 - D can incur fewer memory accesses than C

A. 0

```
inline int __popcount(uint64_t x) {  
    int c = 0;  
    int table[16] = {0, 1, 1, 2, 1,  
                    2, 2, 3, 1, 2, 2, 3, 2, 3, 3, 4};  
    while(x) {  
        c += table[(x & 0xF)];  
        x = x >> 4;  
    }  
    return c;  
}
```

B. 1

C. 2

D. 3

E. 4



```
inline int __popcount(uint64_t x) {  
    int c = 0;  
    int table[16] = {0, 1, 1, 2, 1,  
                    2, 2, 3, 1, 2, 2, 3, 2, 3, 3, 4};  
    for (uint64_t i = 0; i < 16; i++) {  
        c += table[(x & 0xF)];  
        x = x >> 4;  
    }  
    return c;  
}
```

Demo revisited

- Why the performance is better when option is not “0”
 - ① The amount of dynamic instructions needs to execute is a lot smaller
 - ② The amount of branch instructions to execute is smaller
 - ③ The amount of branch mis-predictions is smaller
 - ④ The amount of data accesses is smaller

A. 0 **if**(option)
 std::sort(data, data + arraySize);

B. 1
C. 2 **for** (**unsigned** i = 0; i < 100000; ++i) {
D. 3 **int** threshold = std::rand();
E. 4 **for** (**unsigned** i = 0; i < arraySize; ++i) {
 if (data[i] >= threshold)
 sum ++;
 }
 }

What if the code look like this?

- D-L1 Cache configuration of NVIDIA Tegra X1
 - Size 64KB, 4-way set associativity, 64B block, LRU policy, write-allocate, write-back, and assuming 64-bit address.

```
int a[16384], b[16384], c[16384];
/* c = 0x10000, a = 0x20000, b = 0x30000 */
for(i = 0; i < 512; i++)
    c[i] = a[i]; //load a and then store to c
for(i = 0; i < 512; i++)
    c[i] += b[i]; //load b, load c, add, and then store to c
```

What's the data cache miss rate for this code?

- A. 6.25%
- B. 56.25%
- C. 66.67%
- D. 68.75%
- E. 100%

What kind(s) of misses can matrix transpose remove?

- By transposing a matrix, the performance of matrix multiplication can be further improved. What kind(s) of cache misses does matrix transpose help to remove?

Block

```
for(i = 0; i < ARRAY_SIZE; i+=(ARRAY_SIZE/n)) {  
    for(j = 0; j < ARRAY_SIZE; j+=(ARRAY_SIZE/n)) {  
        for(k = 0; k < ARRAY_SIZE; k+=(ARRAY_SIZE/n)) {  
            for(ii = i; ii < i+(ARRAY_SIZE/n); ii++)  
                for(jj = j; jj < j+(ARRAY_SIZE/n); jj++)  
                    for(kk = k; kk < k+(ARRAY_SIZE/n); kk++)  
                        c[ii][jj] += a[ii][kk]*b[kk][jj];  
        }  
    }  
}
```

- A. Compulsory miss
- B. Capacity miss
- C. Conflict miss
- D. Capacity & conflict miss
- E. Compulsory & conflict miss

Block + Transpose

```
// Transpose matrix b into b_t  
for(i = 0; i < ARRAY_SIZE; i+=(ARRAY_SIZE/n)) {  
    for(j = 0; j < ARRAY_SIZE; j+=(ARRAY_SIZE/n)) {  
        b_t[i][j] += b[j][i];  
    }  
}  
  
for(i = 0; i < ARRAY_SIZE; i+=(ARRAY_SIZE/n)) {  
    for(j = 0; j < ARRAY_SIZE; j+=(ARRAY_SIZE/n)) {  
        for(k = 0; k < ARRAY_SIZE; k+=(ARRAY_SIZE/n)) {  
            for(ii = i; ii < i+(ARRAY_SIZE/n); ii++)  
                for(jj = j; jj < j+(ARRAY_SIZE/n); jj++)  
                    for(kk = k; kk < k+(ARRAY_SIZE/n); kk++)  
                        // Compute on b_t  
                        c[ii][jj] += a[ii][kk]*b_t[jj][kk];  
        }  
    }  
}
```

What data structure is performing better

	Array of objects	object of arrays
	<pre>struct grades { int id; double *homework; double average; };</pre>	<pre>struct grades { int *id; double **homework; double *average; };</pre>
average of each homework	<pre>for(i=0;i<homework_items; i++) { gradesheet[total_number_students].homework[i] = 0.0; for(j=0;j<total_number_students;j++) gradesheet[total_number_students].homework[i] +=gradesheet[j].homework[i]; gradesheet[total_number_students].homework[i] /= (double)total_number_students; }</pre>	<pre>for(i = 0;i < homework_items; i++) { gradesheet.homework[i][total_number_students] = 0.0; for(j = 0; j <total_number_students;j++) { gradesheet.homework[i][total_number_students] += gradesheet.homework[i][j]; } gradesheet.homework[i][total_number_students] /= total_number_students; }</pre>

- Considering your workload would like to calculate the average score of **one of the homework for all students**, which data structure would deliver better performance?
 - Array of objects
 - Object of arrays

Cache coherency

- Assuming that we are running the following code on a CMP with a cache coherency protocol, how many of the following outputs are possible? (a is initialized to 0 as assume we will output more than 10 numbers)

thread 1	thread 2
while(1) printf("%d ", a);	while(1) a++;

- ① 0123456789
- ② 1259368101213
- ③ 1111111164100
- ④ 111111111100
- A. 0
- B. 1
- C. 2
- D. 3
- E. 4

Performance comparison

- Comparing implementations of thread_vadd — L and R, please identify which one will be performing better and why

Version L

```
void *threaded_vadd(void *thread_id)
{
    int tid = *(int *)thread_id;
    int i;
    for(i=tid;i<ARRAY_SIZE;i+=NUM_OF_THREADS)
    {
        c[i] = a[i] + b[i];
    }
    return NULL;
}
```

```
void *threaded_vadd(void *thread_id)
{
    int tid = *(int *)thread_id;
    int i;
    for(i=tid*(ARRAY_SIZE/NUM_OF_THREADS);i<(tid+1)*(ARRAY_SIZE/NUM_OF_THREADS);i++)
    {
        c[i] = a[i] + b[i];
    }
    return NULL;
}
```

Version R

- L is better, because the cache miss rate is lower
- R is better, because the cache miss rate is lower
- L is better, because the instruction count is lower
- R is better, because the instruction count is lower
- Both are about the same

FalseSharing

Main thread

```
for(i = 0 ; i < NUM_OF_THREADS ; i++)
{
    tids[i] = i;
    pthread_create(&thread[i], NULL, threaded_vadd, &tids);
}
for(i = 0 ; i < NUM_OF_THREADS ; i++)
    pthread_join(thread[i], NULL);
```

Free-answer questions

Branch prediction performance

```
i = 0;  
do {  
    if( i % 2 != 0) // Branch X, taken if i % 2 == 0  
        a[i] *= 2;  
    a[i] += i;  
} while ( ++i < 100)// Branch Y
```

- What's the branch prediction accuracy if the processor uses?
 - A 2-bit local predictor
 - A (4, 2) global history predictor
- Can you rewrite the code to generate the same result but eliminate branch X?

Register renaming

- For the following C code:

```
do {  
    number_of_nodes++;  
    current = current->next;  
} while ( current != NULL );
```

- Assume we have a dual-fetch, dual-issue, out-of-order pipeline where
 - INT ALU takes 1 cycle
 - MEM pipeline: 5 cycles in total
 - BR takes 1 cycle to resolve
- If the loop is taken twice, how many cycles it takes to issue all instructions?
- If the loop is taken 100 times, what's the average CPI?

Cache simulation

```
uint64_t* conflict(uint64_t * data, uint64_t size, uint64_t arg1) {  
  
    uint64_t sum=0;  
  
    for(uint i = 0; i < arg1; i++)  
    {  
        for(uint x = i; x < size; x+=arg1) {  
            sum += data[x];  
        }  
    }  
    data[0] = sum;  
    return data;  
}
```

- On an intel Core i3 12100F, what values of `arg1` and `size` can —
 - Lead to 100% miss rate?
 - Lead to 50% miss rate?
 - Lead to almost 0% miss rate?

Reverse caching

- Below, we have given you four different sequences of addresses generated by a program running on a processor with a data cache. Cache hit ratio for each sequence is also shown below. Assuming that the cache is initially empty at the beginning of each sequence, find out the following parameters of the processor's data cache (ensure that you sufficiently explain your answer)
 - Associativity (1, 2, or 4 ways)
 - Block size (1, 2, 4, 8, 16, or 32 bytes)
 - Total cache size (256B, or 512B)
 - Replacement policy (LRU or FIFO)
- Address Sequence 1: [0, 2, 4, 8, 16, 32] Hit Ratio: 0.33
 - Address Sequence 2: [0, 512, 1024, 1536, 2048, 1536, 1024, 512, 0] Hit Ratio: 0.33
 - Address Sequence 3: [0, 64, 128, 256, 512, 256, 128, 64, 0] Hit Ratio: 0.33
 - Address Sequence 4: [0, 512, 1024, 0, 1536, 0, 2048, 512] Hit Ratio: 0.25

Open-ended questions

SMT v.s. CMP

- An SMT processor is basically a SuperScalar processor with multiple instruction front-end. Assume within the same chip area, we can build an SMT processor supporting 4 threads, with 6-issue pipeline, 64KB cache or — a CMP with 4x 2-issue pipeline & 16KB cache in each core. Please identify how many of the following statements are/is correct when running programs on these processors.
 - ① If we are just running one program in the system, the program will perform better on an SMT processor
 - ② If we are running 4 applications simultaneously, the cache miss rates will be higher in the SMT processor
 - ③ If we are running 4 applications simultaneously, the branch mis-prediction will be higher in the SMT processor
 - ④ If we are running one program with 4 parallel threads, the cache miss rates will be higher in the SMT processor
 - ⑤ If we are running one program with 4 parallel threads simultaneously, the branch mis-prediction will be longer in the SMT processor
- A. 1
B. 2
C. 3
D. 4
E. 5

Latency numbers every programmer should know

- Review this and explain why
- https://colin-scott.github.io/personal_website/research/interactive_latency.html

Other open-ended questions

- Given the instruction front-end is decoupled from the backend of the pipeline ALUs, do you think ISA still affect performance?
 - Emily Blem, Jaikrishnan Menon, and Karthikeyan Sankaralingam. 2013. Power struggles: Revisiting the RISC vs. CISC debate on contemporary ARM and x86 architectures. In Proceedings of the 2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA) (HPCA '13). <https://minds.wisconsin.edu/handle/1793/64923>
 - Ashish Venkat and Dean M. Tullsen. 2014. Harnessing ISA diversity: design of a heterogeneous-ISA chip multiprocessor. In Proceeding of the 41st annual international symposium on Computer architecuture (ISCA '14). <http://www.cs.virginia.edu/venkat/papers/isca2014.pdf>
- Reading the following papers and think of good use cases of simultaneous multithreading
 - Weifeng Zhang, Dean M. Tullsen, Brad Calder. Accelerating and Adapting Precomputation Threads for Efficient Prefetching. In 13th International Symposium on High Performance Computer Architecture, January, 2007.
 - A. Roth and G. S. Sohi. Speculative data-driven multithreading. Proceedings HPCA Seventh International Symposium on High-Performance Computer Architecture, Monterrey, Mexico, 2001, pp. 37-48, doi: 10.1109/HPCA.2001.903250.
 - J. D. Collins et al.. Speculative precomputation: long-range prefetching of delinquent loads. Proceedings 28th Annual International Symposium on Computer Architecture, Gothenburg, Sweden, 2001, pp. 14-25, doi: 10.1109/ISCA.2001.937427.

Other open-ended questions

- If you're asked to design a machine learning hardware, what will you do?
- If you're asked to optimize an AI/ML application, what will you do?
- If you're asked to build an Xeon Phi type processor where each core also has many-way SMT, are you going to give the processor more cache or better branch predictor?
- Can we focus on improving the throughput of computing instead of latency? Can you give an example on what type of applications will not work well in this way
- Pros and cons for branch prediction using perceptrons?
- What compiler optimizations would not be effective given OoO execution hardware?

Other open-ended questions (cont.)

- What's Amdahl's Law implication on parallelism? How does that guide future software design?
- What's Dark Silicon Problem? What are the new design trends in addressing the problem?
- If the OoO pipeline is highly optimized, do we still care about the ISA design?
- What's volatile? What's inline? Why we need them? The pros and cons?

Announcements

- iEVAL started and ends on 12/05/2024
 - Submit the prove of your participation in course evaluation through Gradescope
 - It can become a full credit reading quiz (it helps to amortize the penalty of another least performing one) — we will drop a total of three now
- **Assignment 5 and programming assignment 3 due tonight**
- **Final exam**
 - 12/10 8a-11a
 - Closed book, no cheatsheet — the same rules as the midterm

Computer Science & Engineering

203

