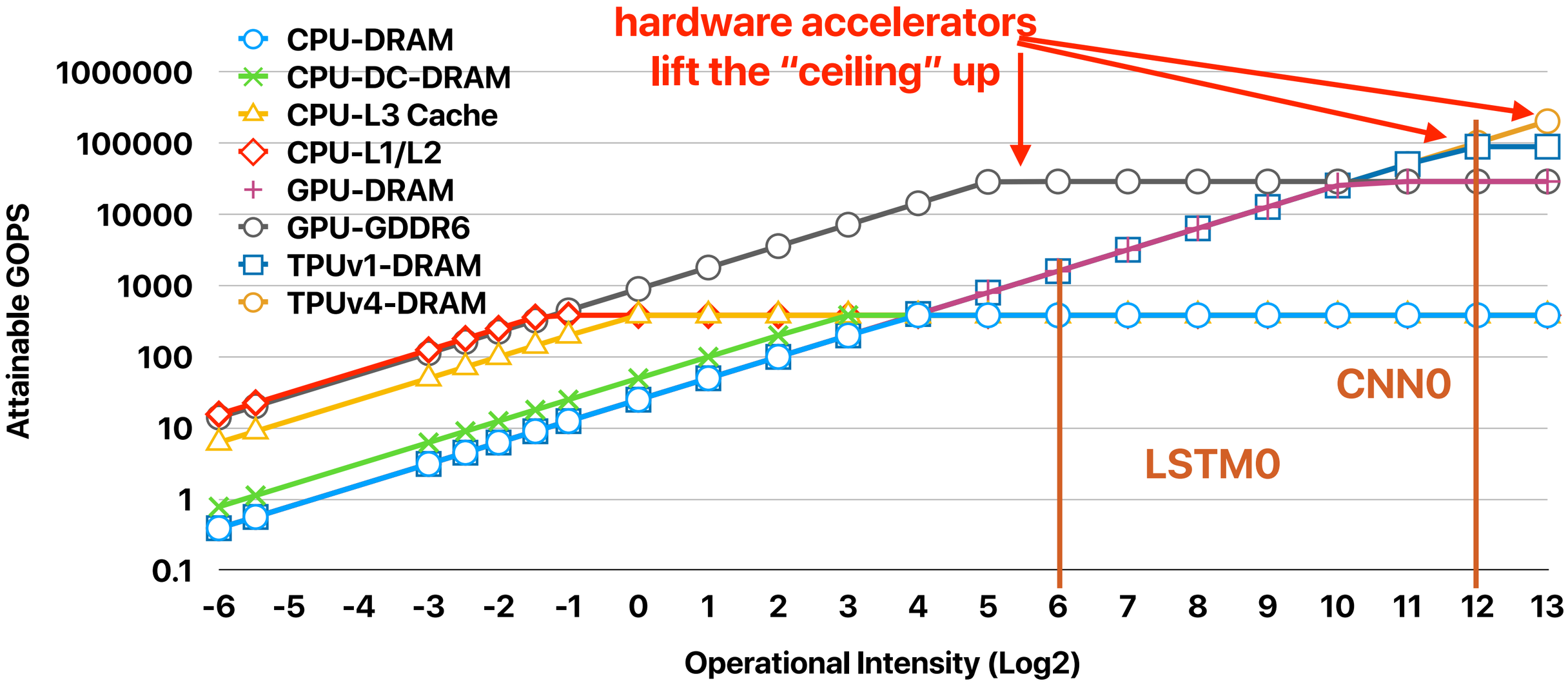
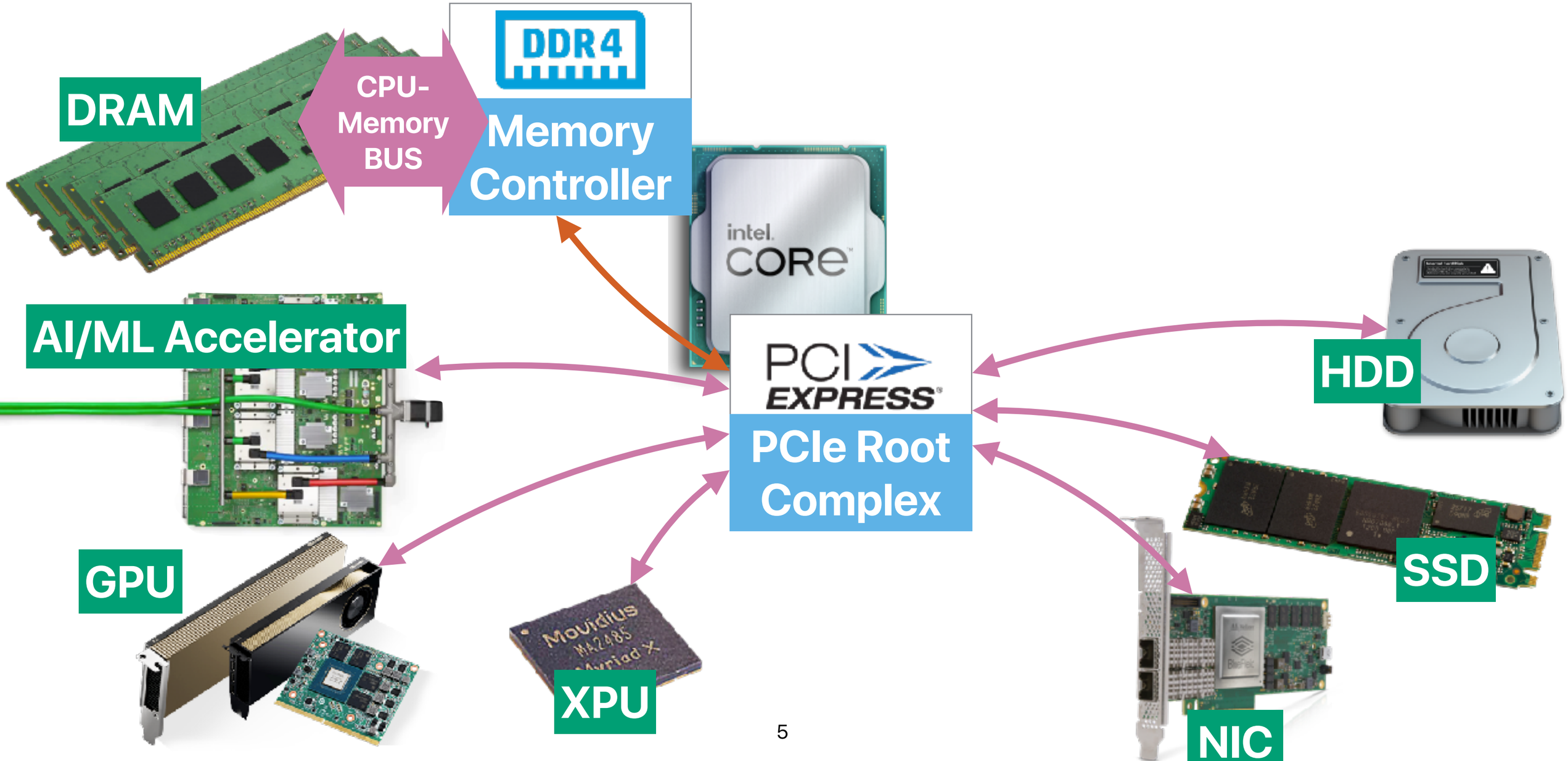
Hardware accelerators (2)

Hung-Wei Tseng

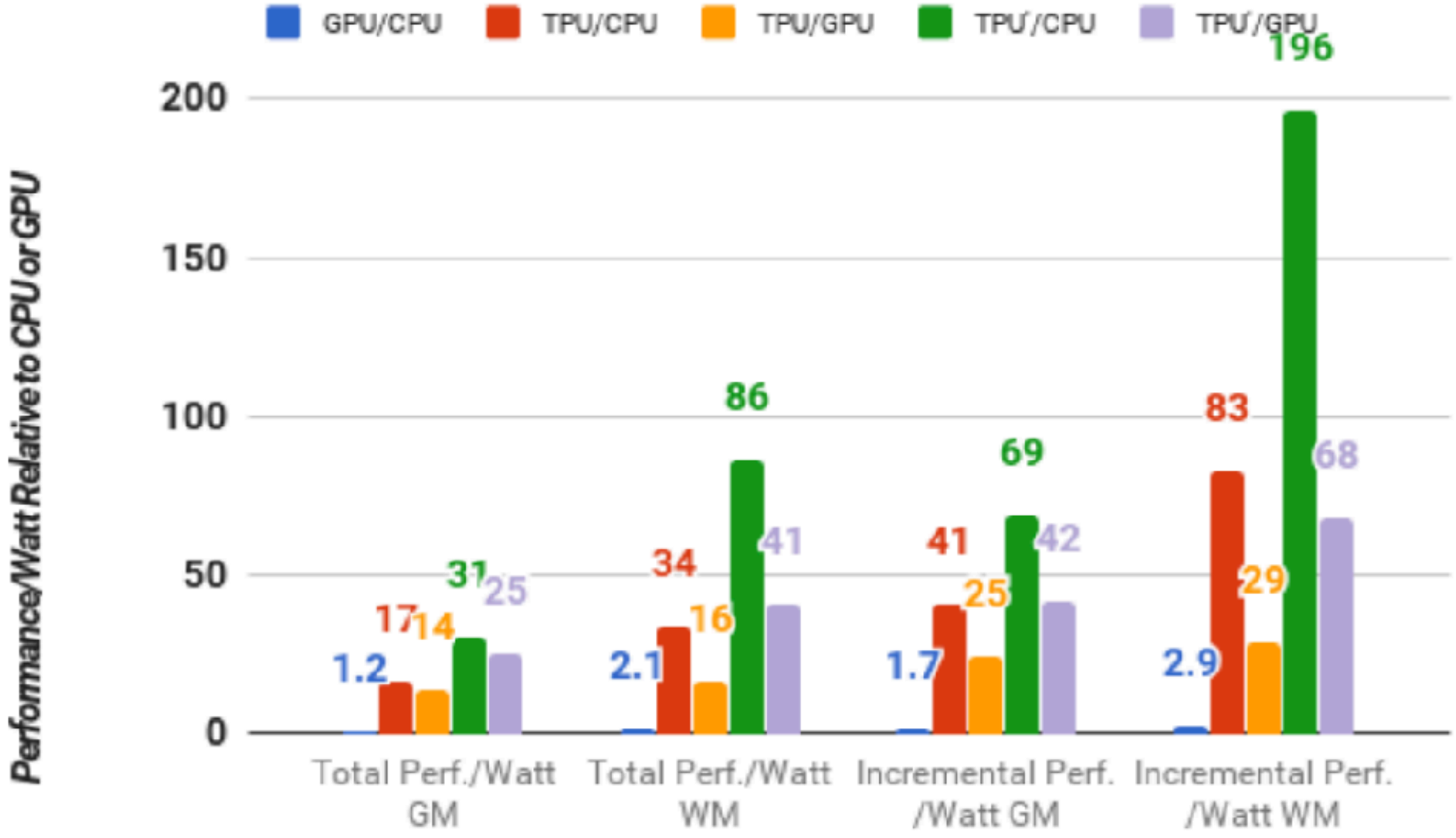
Recap: the rooflines of modern systems



Recap: the landscape of heterogeneous computing



Energy efficiency is better! — reduced cost!



Recap: 10 lessons learned from 3 generations of Google TPUs

- Logic, wires, SRAM & DRAM improve unequally
- Leverage prior compiler optimizations
- Design for performance per TCO vs CapEx
- Backwards ML compatibility
- Inference DSAs need air cooling for global scale
- Some inference apps need floating point arithmetic
- Production inference normally needs multi-tenancy
- DNNs grow ~1.5x/year in memory and compute
- DNN workloads evolve with DNN breakthroughs
- Inference SLO limit is P99 latency, not batch size

N. P. Jouppi *et al.*, "Ten Lessons From Three Generations Shaped Google's TPUv4i : Industrial Product," *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, Valencia, Spain, 2021, pp. 1-14, doi: 10.1109/ISCA52012.2021.00010.

Outline

- Recent development of TPUs
- Ray Tracing Accelerators
- General-purpose approximate computing on NPUs

Recent advancement of TPUs

TPU v4

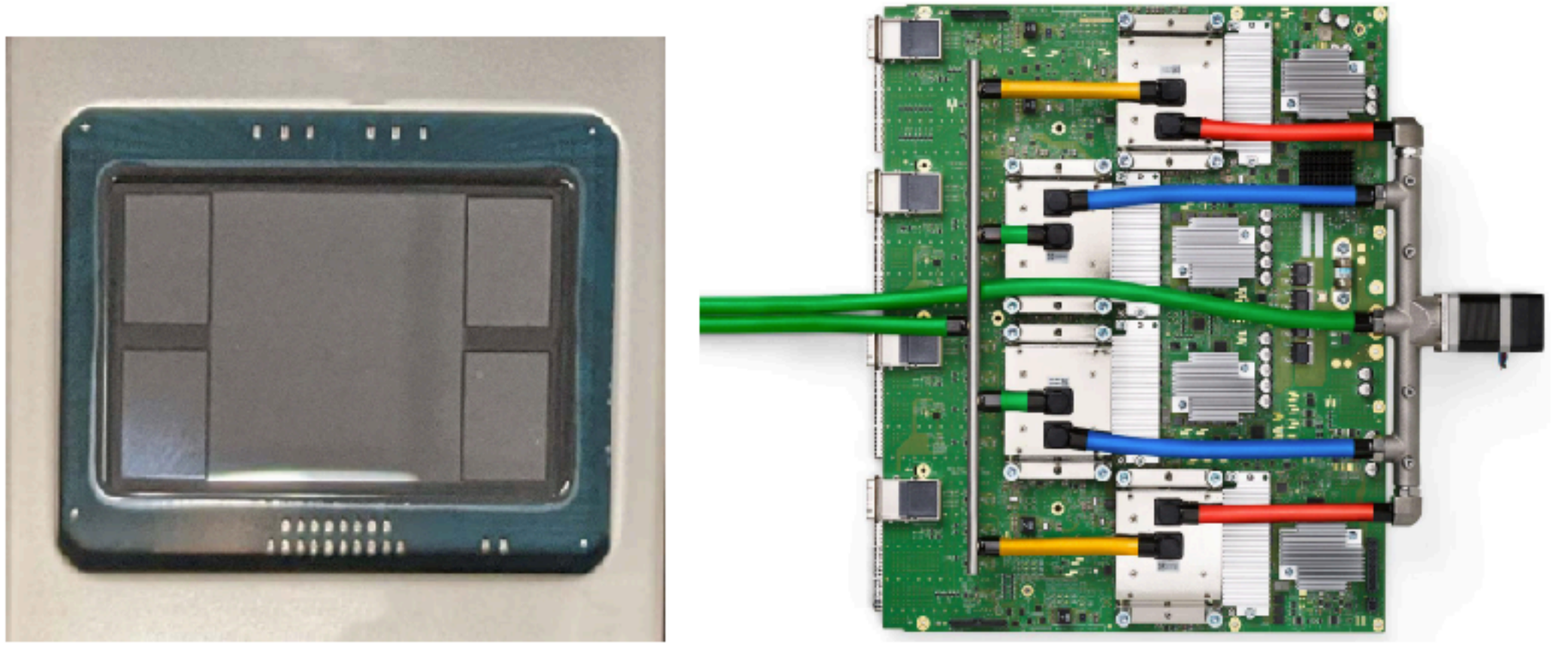


Figure 2: The TPU v4 package (ASIC in center plus 4 HBM stacks) and printed circuit board with 4 liquid-cooled packages. The board's front panel has 4 top-side PCIe connectors and 16 bottom-side OSFP connectors for inter-tray ICI links.

Four generations of TPUs

Table 1: Workloads by DNN model type (% TPUs used). Over 90% of training at Google is on TPUs. The parenthesized entries split Transformer models into the subtypes of BERT and LLM. Columns 2 to 4 show workloads for inference [25], training and inference [26], and inference [27]. The last workload is for training on TPU v4s over 30 days in October 2022.

DNN Model	*TPU v1 7/2016 (Inference)*	*TPU v3 4/2019 (Training & Inference)*	*TPU v4 Lite 2/2020 (Inference)*	*TPU v4 10/2022 (Training)*
MLP/DLRM	61%	27%	25%	24%
RNN	29%	21%	29%	2%
CNN	5%	24%	18%	12%
Transformer	--	21%	28%	57%
(BERT)	--	--	*(28%)*	*(26%)*
(LLM)	--	--	--	*(31%)*

Norm Jouppi, George Kurian, Sheng Li, Peter Ma, Rahul Nagarajan, Lifeng Nai, Nishant Patil, Suvinay Subramanian, Andy Swing, Brian Towles, Clifford Young, Xiang Zhou, Zongwei Zhou, and David A Patterson. 2023. TPU v4: An Optically Reconfigurable Supercomputer for Machine Learning with Hardware Support for Embeddings. In Proceedings of the 50th Annual International Symposium on Computer Architecture (ISCA '23).

RTX on—The NVIDIA Turing GPU

3:00

Why RTX architecture?

Turing architecture — specialized for key workloads

- CUDA cores for vector processing
- Tensor cores for AI
- RT cores for VR/AR

> While Turing is packed with features and horsepower,[1] we made fundamental advancements in several key areas—streaming multiprocessor (SM) efficiency, a Tensor Core for AI inferencing, and an RTCore for ray-tracing acceleration.

Tensor Cores

Tensor cores for deep learning

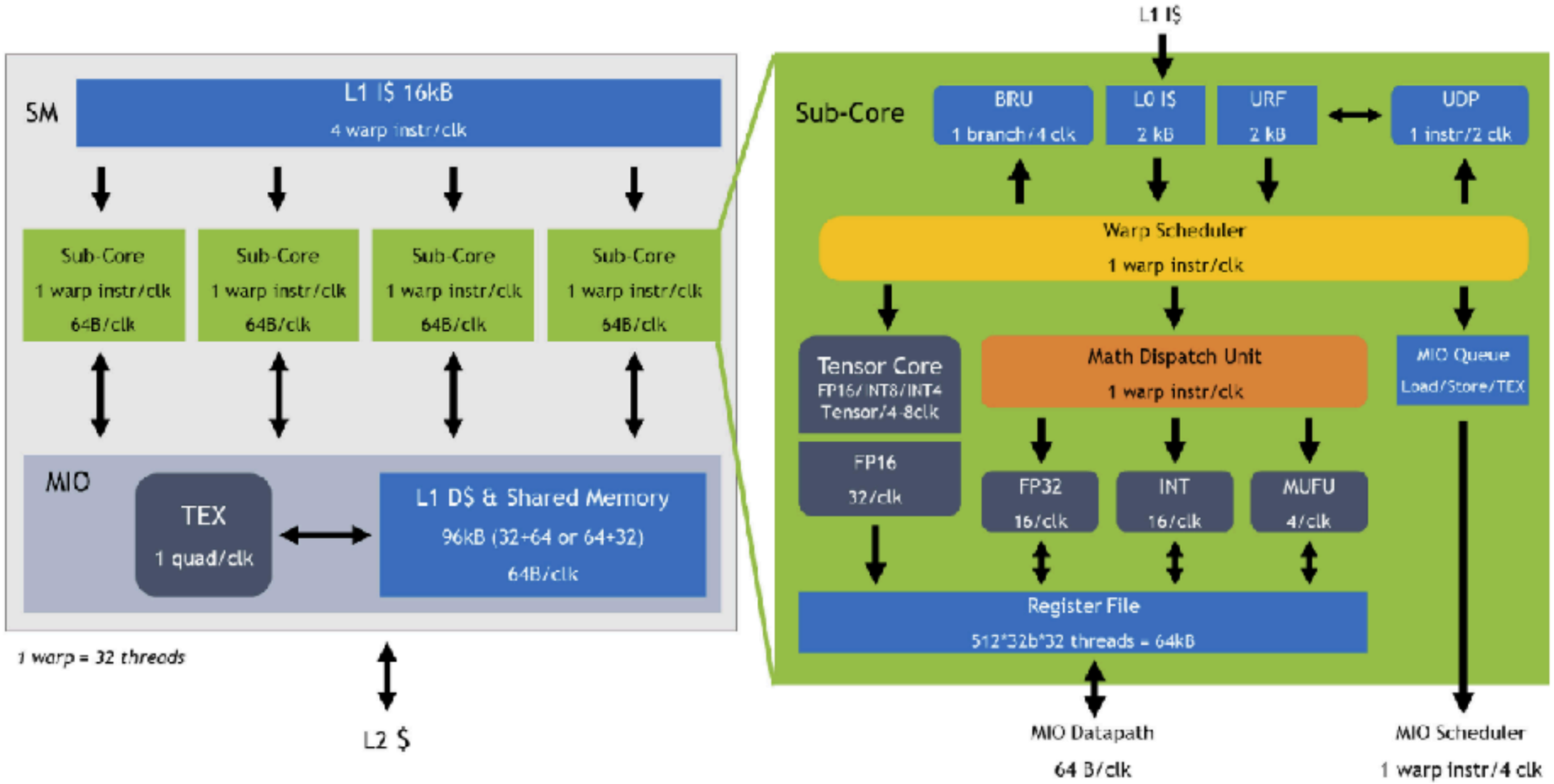


Figure 1. Turing GPU SM, comprising four subcores and a memory interface (MIO). Math throughput, memory bandwidth, L1 data cache topology, register file and cache capacity, and a new uniform datapath were all designed or modified to increase processor efficiency over the previous generation.

Turing Architecture

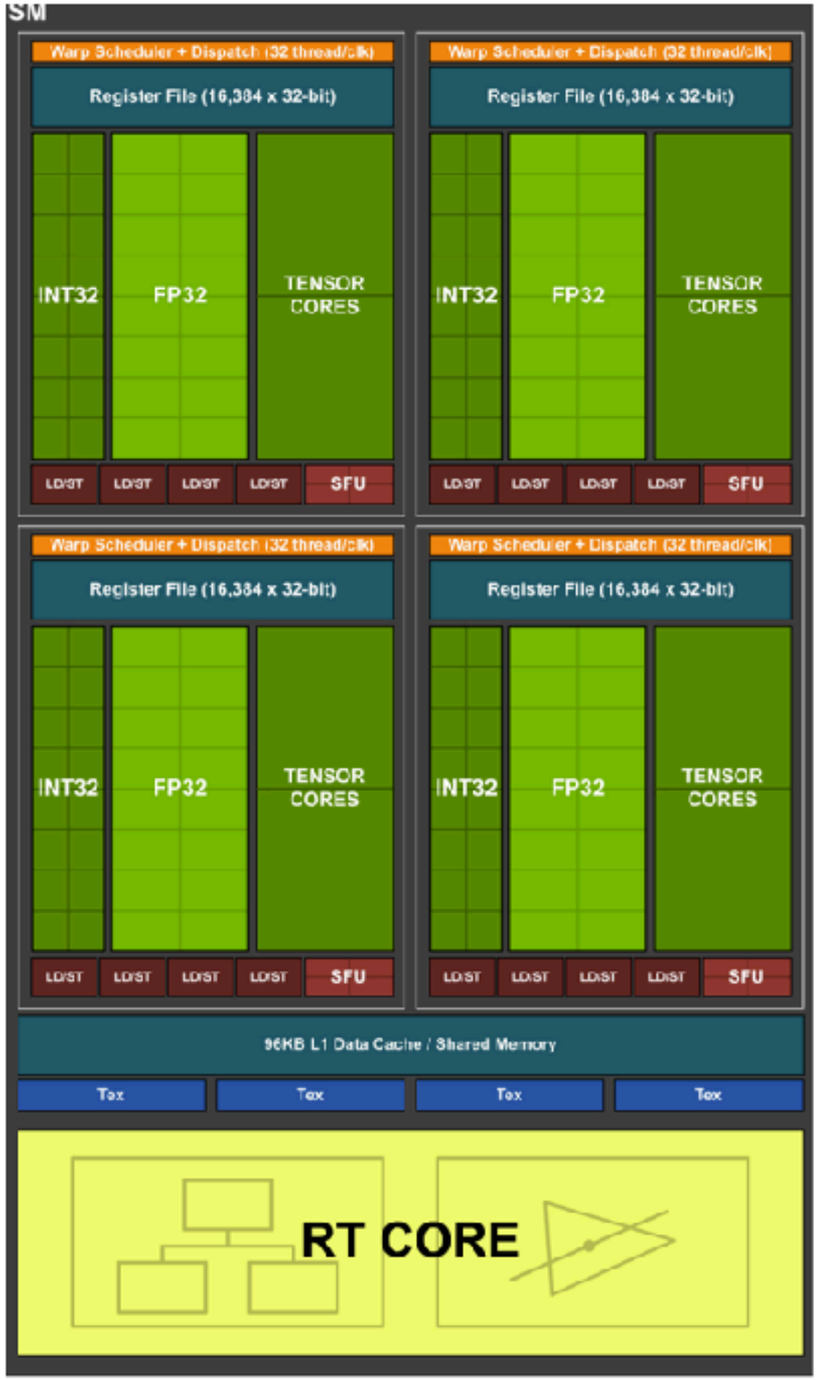


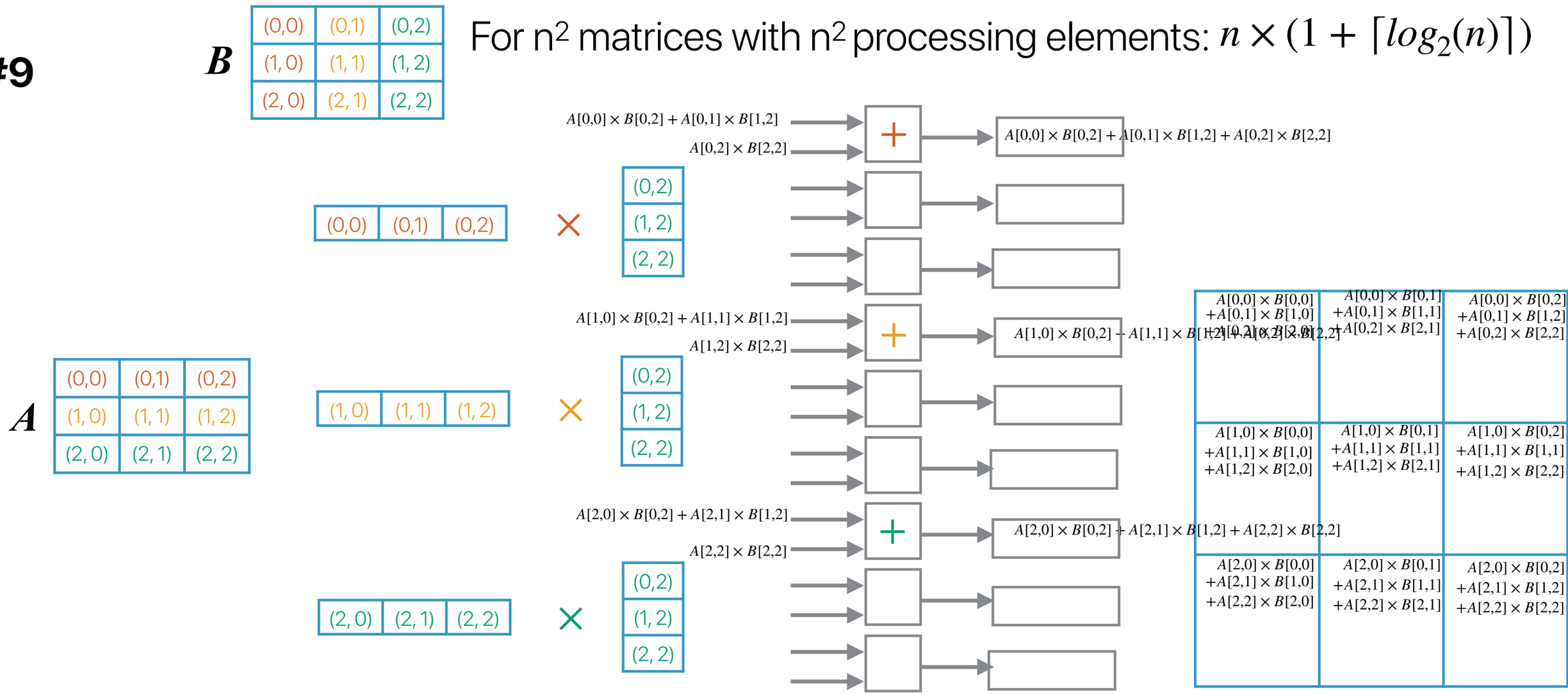
Figure 4. Turing TU102/TU104/TU106 Streaming Multiprocessor (SM)

NVIDIA, "NVIDIA Turing GPU architecture: Graphics reinvented," 2018. [Online]. Available: https://www. nvidia.com/content/dam/en-zz/Solutions/designvisualization/ technologies/ turing-architecture/NVIDIATuring- Architecture-Whitepaper.pdf

Vector processing for MM

#9

For n² matrices with n² processing elements: $n \times (1 + \lceil log_2(n) \rceil)$

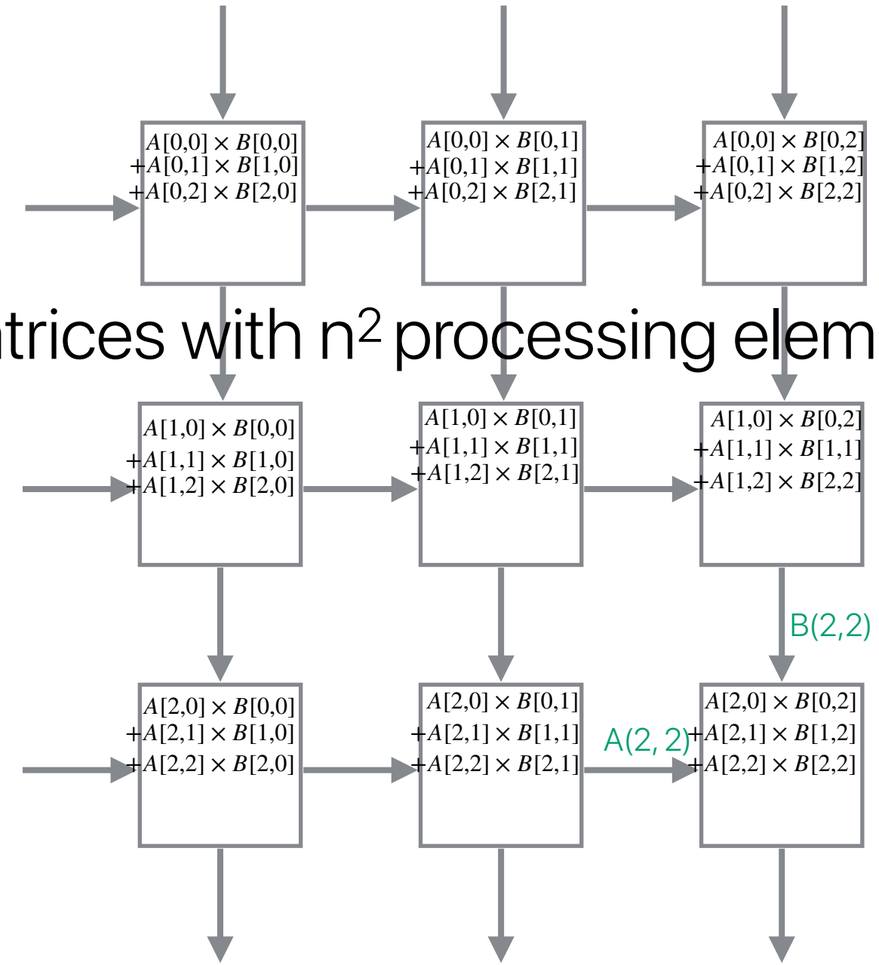


Systolic Arrays used by AI/ML Accelerators

#7

B

(0,0)	(0,1)	(0,2)
(1, 0)	(1, 1)	(1, 2)
(2, 0)	(2, 1)	(2, 2)



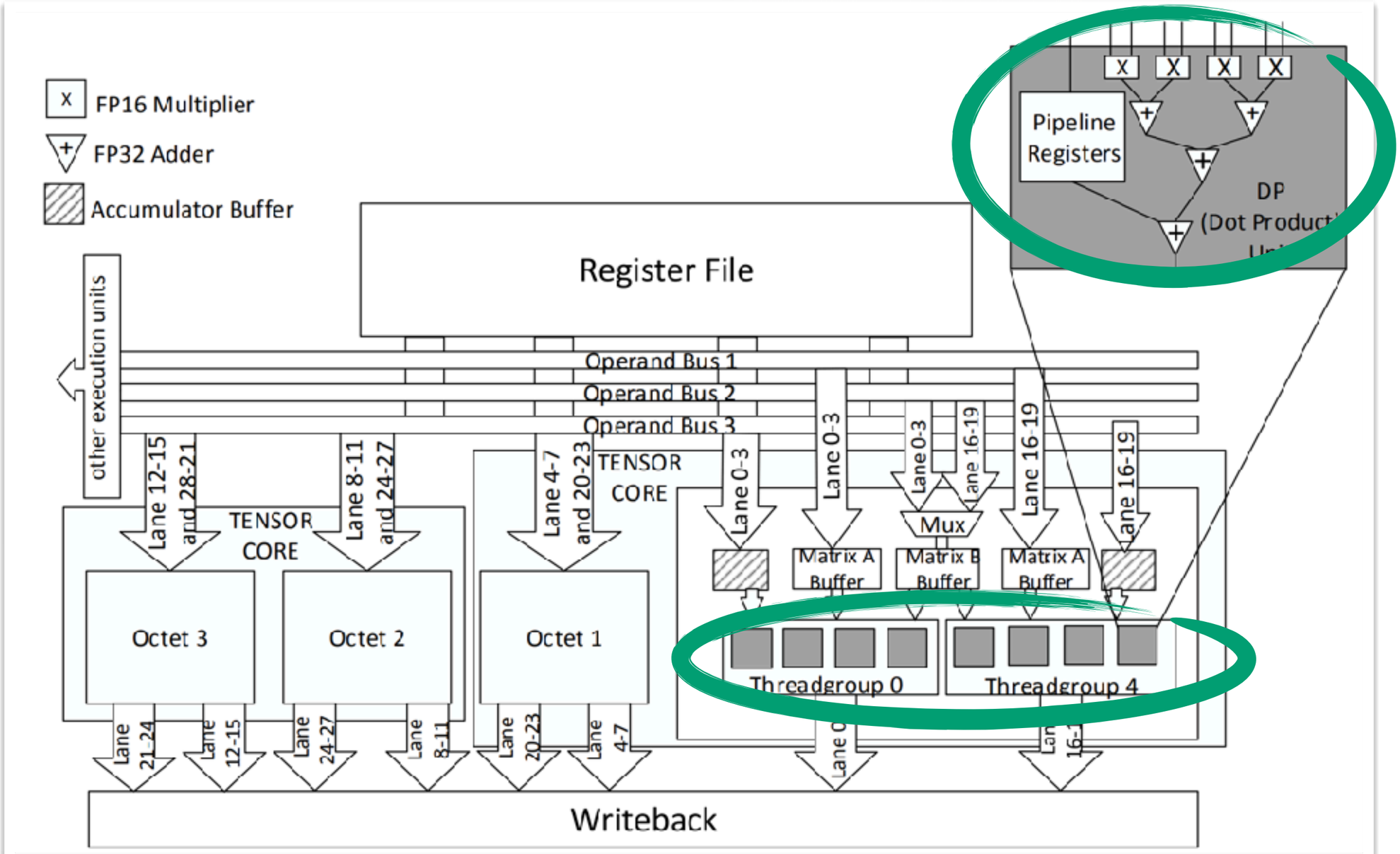
For n² matrices with n² processing elements: $3 \times n - 2$

A

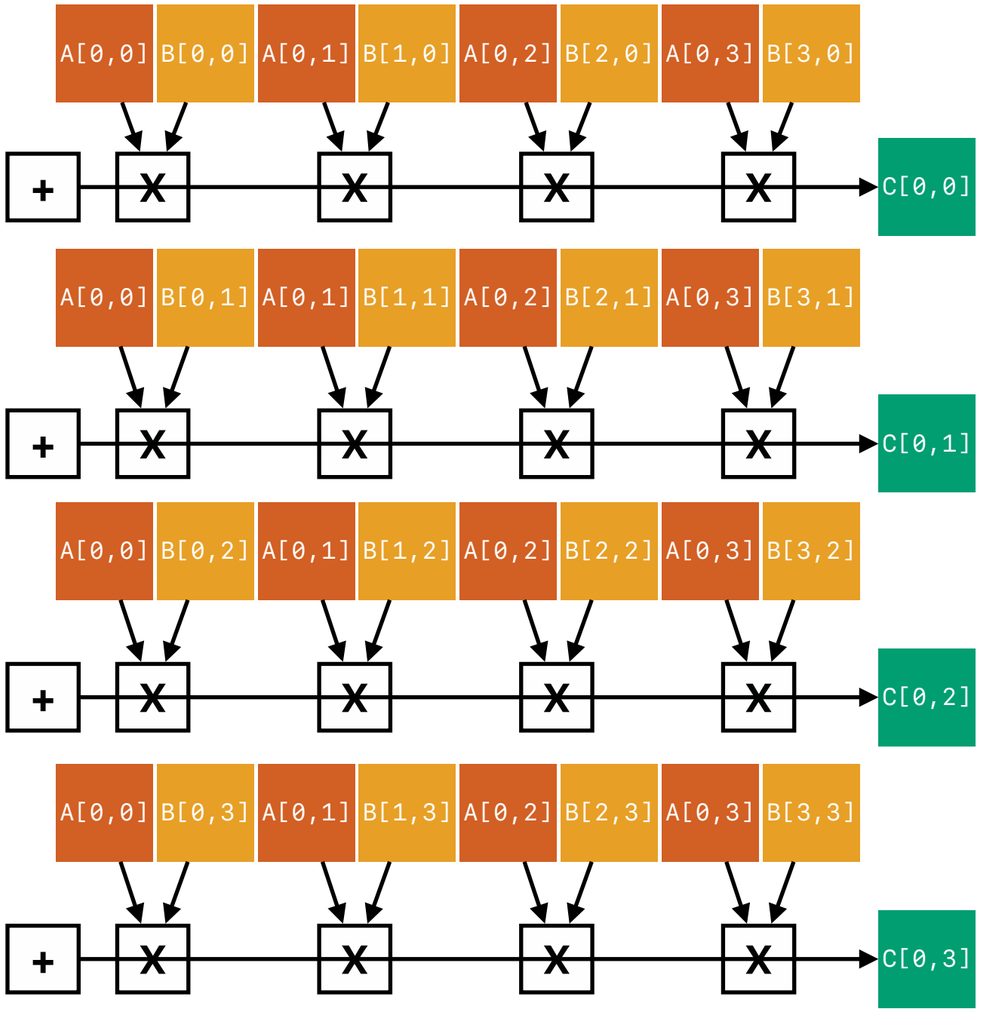
(0,0)	(0,1)	(0,2)
(1, 0)	(1, 1)	(1, 2)
(2, 0)	(2, 1)	(2, 2)

H.T. Kung, "Why systolic architectures?," in IEEE Computer, vol. 15, no. 1, pp. 37-46, Jan. 1982, doi: 10.1109/MC.1982.1653825.

The Tensor Core architecture



Tensor Cores



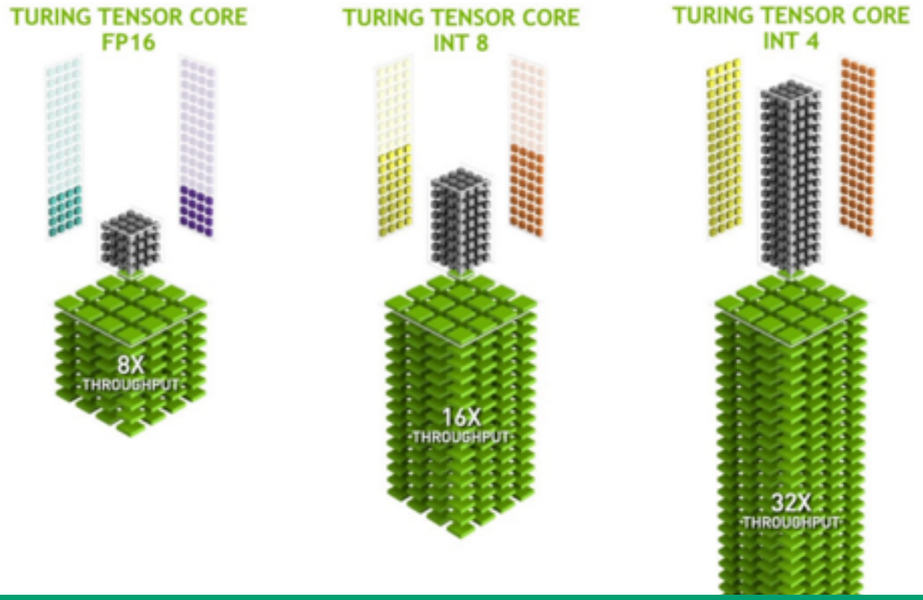
A[0,0]	A[0,1]	A[0,2]	A[0,3]
A[1,0]	A[1,1]	A[1,2]	A[1,3]
A[2,0]	A[2,1]	A[2,2]	A[2,3]
A[3,0]	A[3,1]	A[3,2]	A[3,3]

B[0,0]	B[0,1]	B[0,2]	B[0,3]
B[1,0]	B[1,1]	B[1,2]	B[1,3]
B[2,0]	B[2,1]	B[2,2]	B[2,3]
B[3,0]	B[3,1]	B[3,2]	B[3,3]

C[0,0]	C[0,1]	C[0,2]	C[0,3]
C[1,0]	C[1,1]	C[1,2]	C[1,3]
C[2,0]	C[2,1]	C[2,2]	C[2,3]
C[3,0]	C[3,1]	C[3,2]	C[3,3]

Efficiency of Tensor Cores

The NVIDIA Tesla T4 GPU includes 2,560 CUDA Cores and 320 Tensor Cores, delivering up to 130 TOPs (Tera Operations per second) of INT8 and up to 260 TOPS of INT4 inferencing performance (see Appendix A, *Turing TU104 GPU* for more Tesla T4 specifications). Compared to CPU-based inferencing, the Tesla T4, powered by the new Turing Tensor Cores, delivers up to 40X higher inference performance[6] (see Figure 9).



FLOPS of 8K by 8K matrix multiplications =

$$8192 \times 8192 \times 8192 \times 2$$

$$\frac{8192 \times 8192 \times 8192 \times 2}{2560} = 429496730 \ CUDA \ core \ cycles$$

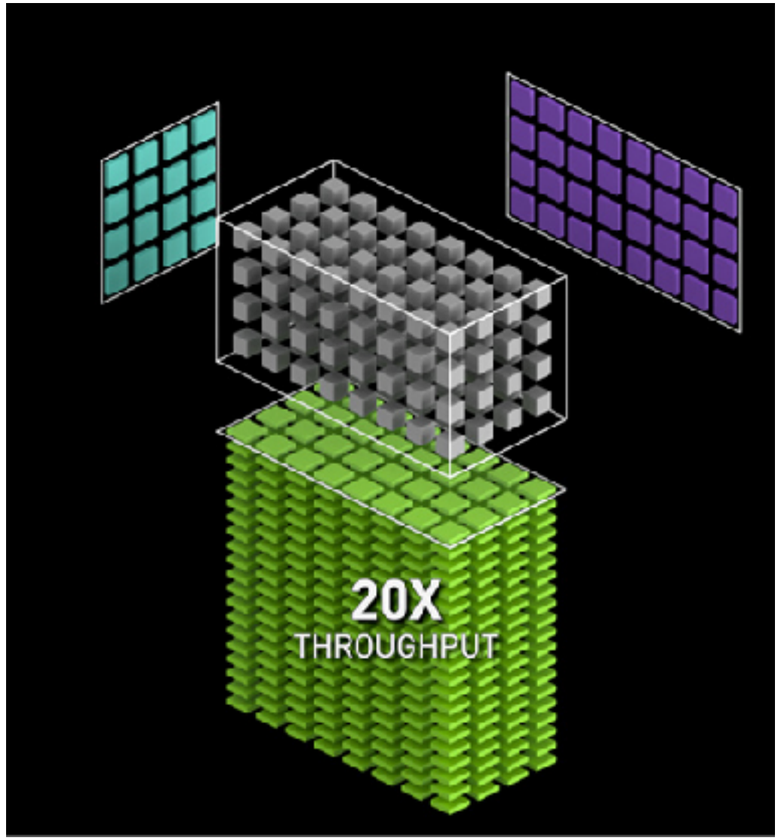
$$\frac{8192 \times 8192 \times 8192 \times 2}{320 \times 128} = 26843546 \ Tensor \ core \ cycles$$

Tensor cores make matrix processing 16x faster

Each tensor core operation performs 4x4x4 MMA in one cycle
~ 128 FP operations in conventional scalar models

NVIDIA, "NVIDIA Turing GPU architecture: Graphics reinvented," 2018. [Online]. Available: https://www. nvidia.com/content/dam/en-zz/Solutions/designvisualization/technologies/turing-architecture/NVIDIATuring- Architecture-Whitepaper.pdf

Efficiency of Tensor Cores (RTX 4090)



	RTX 4090
CUDA Cores	16384
Tensor Cores	512

FLOPS of 8K by 8K matrix multiplications =

$$8192 \times 8192 \times 8192 \times 2$$

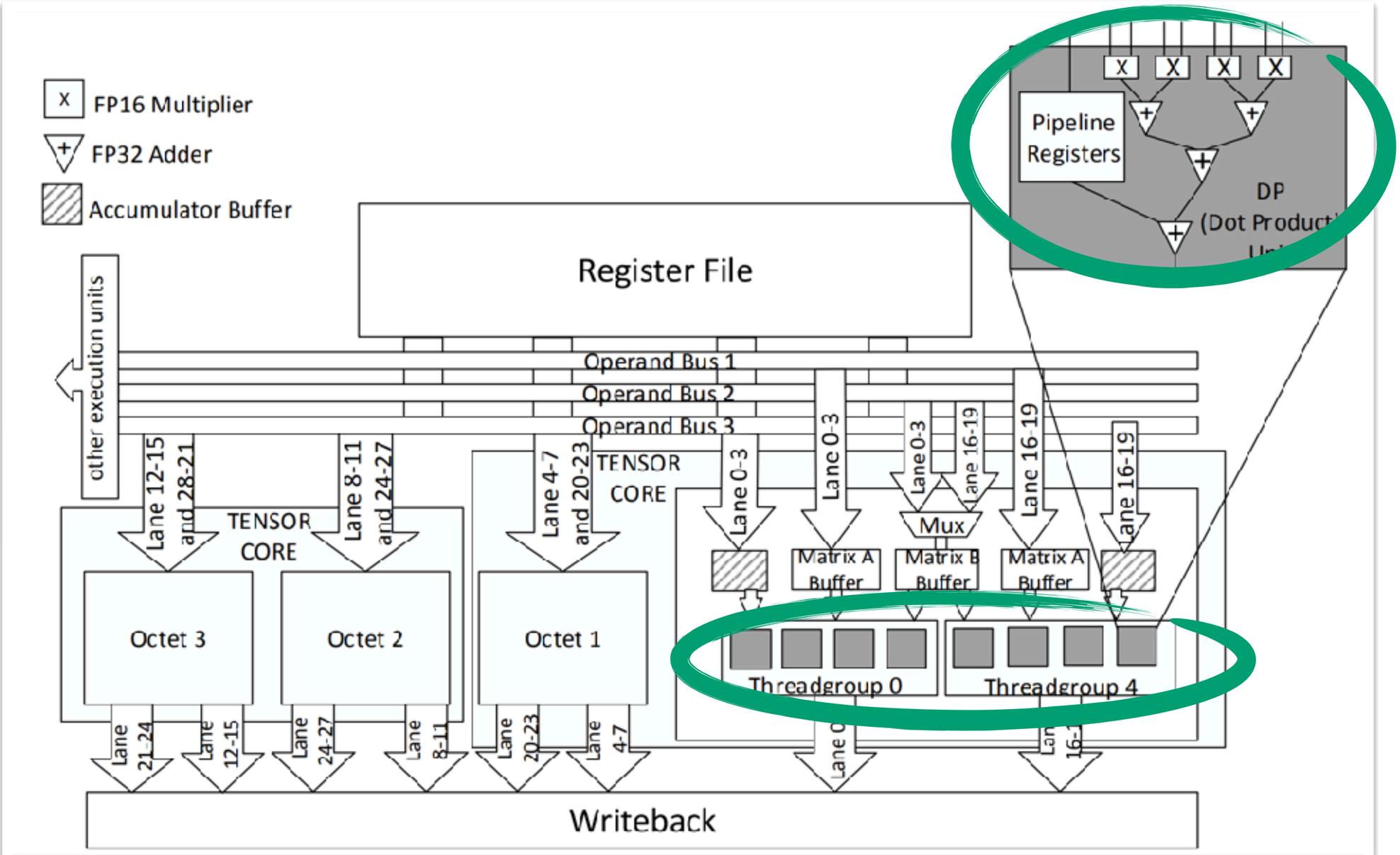
$$\frac{8192 \times 8192 \times 8192 \times 2}{16384} = 67{,}108{,}864 \ \textit{CUDA core cycles}$$

$$\frac{8192 \times 8192 \times 8192 \times 2}{512 \times 256} = 8{,}388{,}608 \ \textit{Tensor core cycles}$$

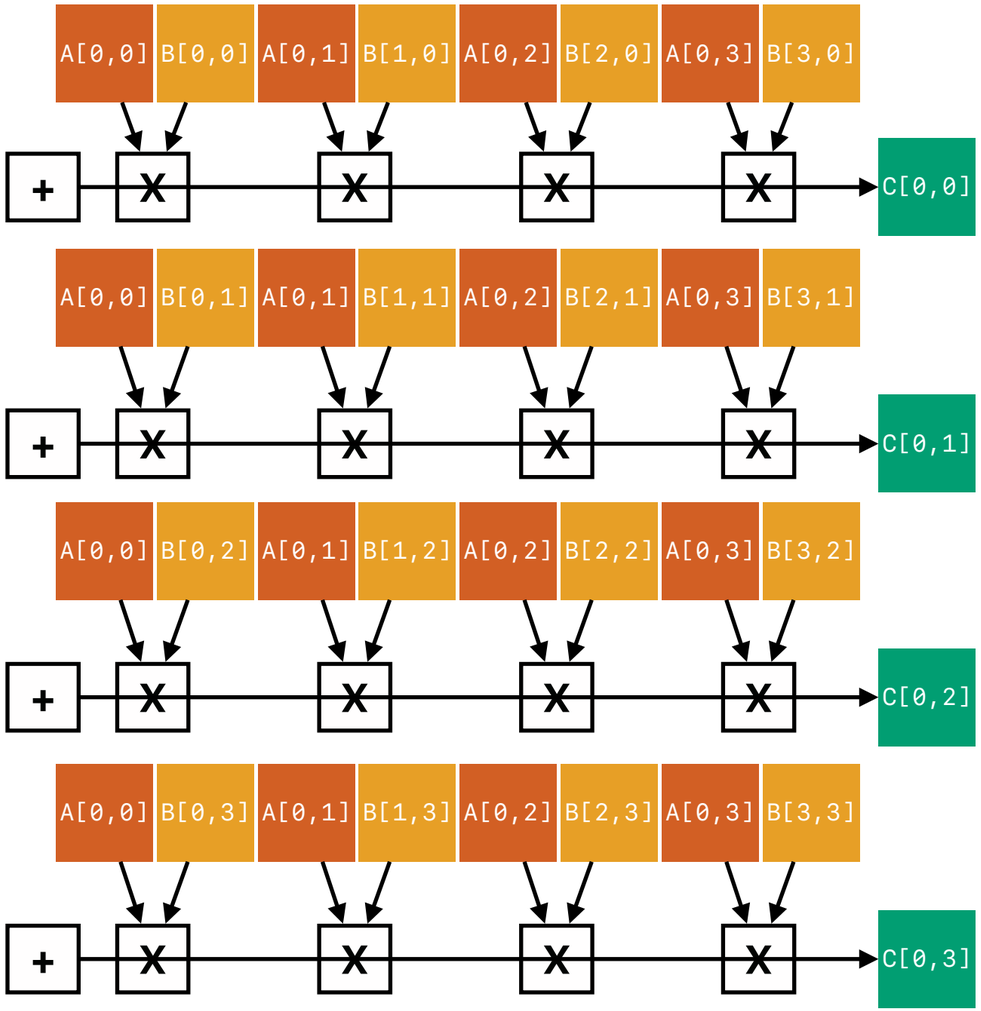
Each tensor core operation performs 8x4x4 MMA in one cycle
~ 256 FP operations in conventional scalar models

Tensor cores make matrix processing 8x faster

Tensor Core Architecture



Tensor Cores



A[0,0]	A[0,1]	A[0,2]	A[0,3]
A[1,0]	A[1,1]	A[1,2]	A[1,3]
A[2,0]	A[2,1]	A[2,2]	A[2,3]
A[3,0]	A[3,1]	A[3,2]	A[3,3]

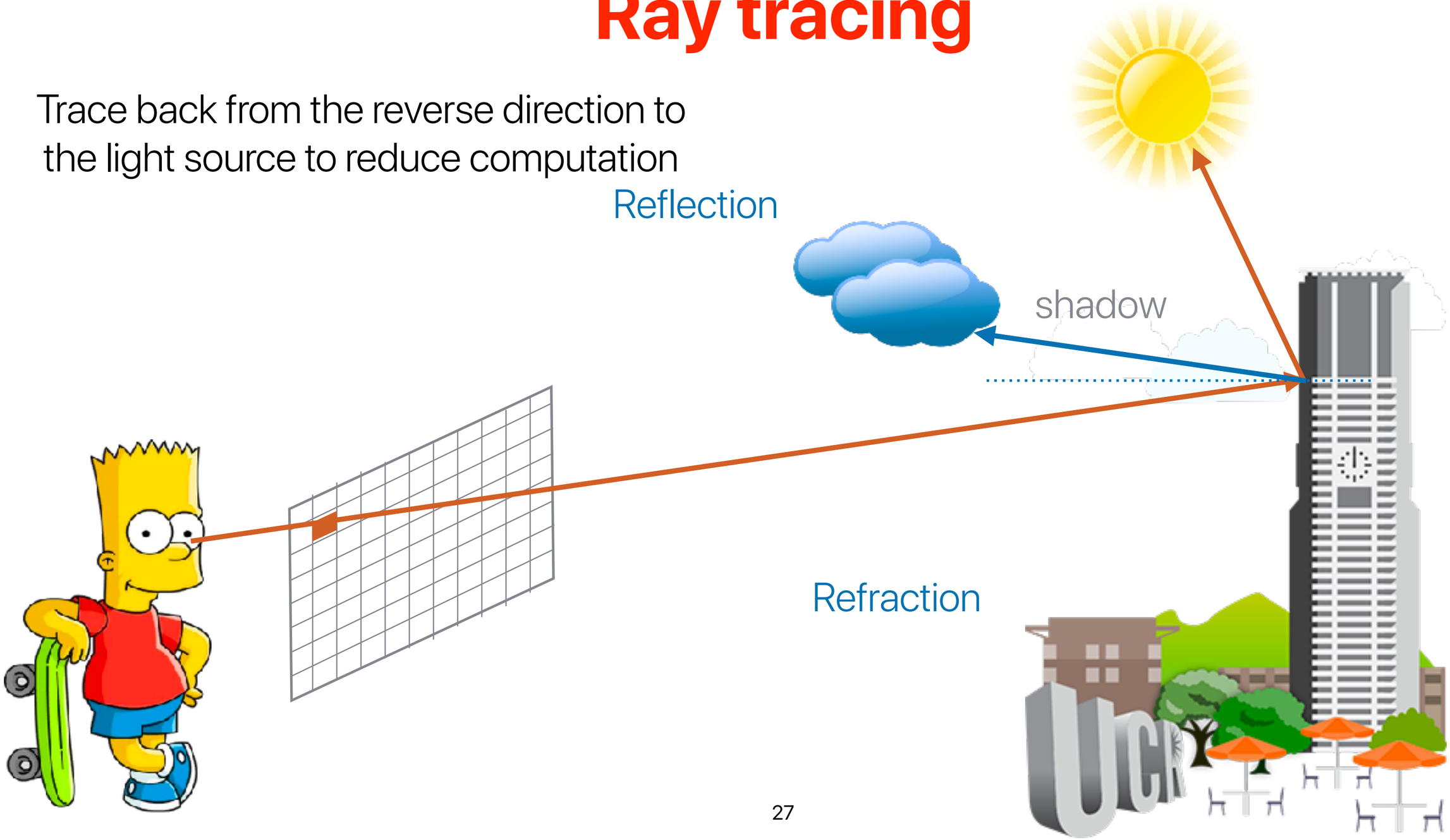
B[0,0]	B[0,1]	B[0,2]	B[0,3]
B[1,0]	B[1,1]	B[1,2]	B[1,3]
B[2,0]	B[2,1]	B[2,2]	B[2,3]
B[3,0]	B[3,1]	B[3,2]	B[3,3]

C[0,0]	C[0,1]	C[0,2]	C[0,3]
C[1,0]	C[1,1]	C[1,2]	C[1,3]
C[2,0]	C[2,1]	C[2,2]	C[2,3]
C[3,0]	C[3,1]	C[3,2]	C[3,3]

RT Cores

Ray tracing

Trace back from the reverse direction to the light source to reduce computation



Simplest ray tracing algorithm

```
for (int j = 0; j < imageHeight; ++j) {
    for (int i = 0; i < imageWidth; ++i) {
        // compute primary ray direction
        Ray primRay;
        computePrimRay(i, j, &primRay);
        // shoot prim ray in the scene and search for the intersection
        Point pHit;
        Normal nHit;
        float minDist = INFINITY;
        Object object = NULL;
        for (int k = 0; k < objects.size(); ++k) {
            if (Intersect(objects[k], primRay, &pHit, &nHit)) {
                float distance = Distance(eyePosition, pHit);
                if (distance < minDistance) {
                    object = objects[k];
                    minDistance = distance;  //update min distance
                }
            }
        }
        if (object != NULL) {
            // compute illumination
            Ray shadowRay;
            shadowRay.direction = lightPosition - pHit;
            bool isShadow = false;
            for (int k = 0; k < objects.size(); ++k) {
                if (Intersect(objects[k], shadowRay)) {
                    isInShadow = true;
                    break;
                }
            }
        }
        if (!isInShadow)
            pixels[i][j] = object->color * light.brightness;
        else
            pixels[i][j] = 0;
    }
}
```

https://www.scratchapixel.com/lessons/3d-basic-rendering/introduction-to-ray-tracing/raytracing-algorithm-in-a-nutshell.html

```
for (int j = 0; j < imageHeight; ++j) {
    for (int i = 0; i < imageWidth; ++i) {
        // compute primary ray direction
        Ray primRay;
        computePrimRay(i, j, &primRay);
        // shoot prim ray in the scene and search for the intersection
        Point pHit;
        Normal nHit;
        float minDist = INFINITY;
        Object object = NULL;
        for (int k = 0; k < objects.size(); ++k) {
            if (Intersect(objects[k], primRay, &pHit, &nHit)) {
                float distance = Distance(eyePosition, pHit);
                if (distance < minDistance) {
                    object = objects[k];
                    minDistance = distance;  //update min distance
                }
            }
        }
        if (object != NULL) {
            // compute illumination
            Ray shadowRay;
            shadowRay.direction = lightPosition - pHit;
            bool isShadow = false;
            for (int k = 0; k < objects.size(); ++k) {
                if (Intersect(objects[k], shadowRay)) {
                    isInShadow = true;
                    break;
                }
            }
        }
        if (!isInShadow)
            pixels[i][j] = object->color * light.brightness;
        else
            pixels[i][j] = 0;
    }
}
```

3:00

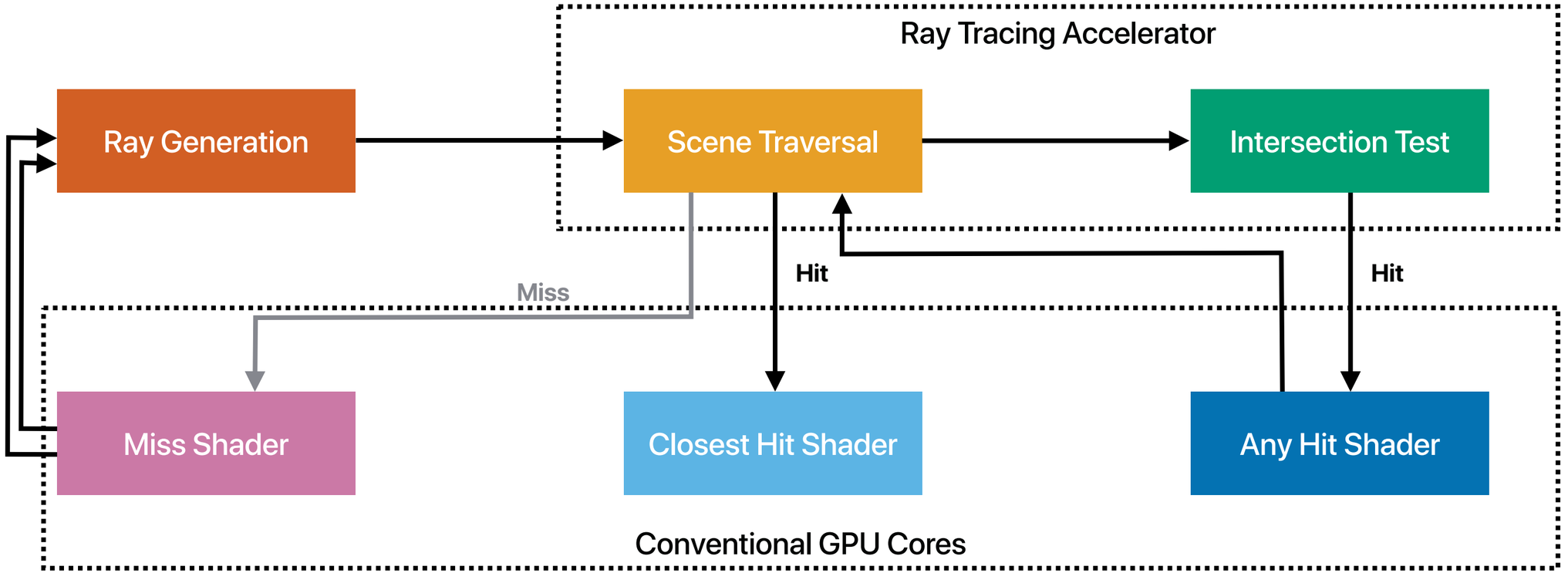
Why we need an accelerator instead of just using GPUs?

Simplest ray tracing algorithm

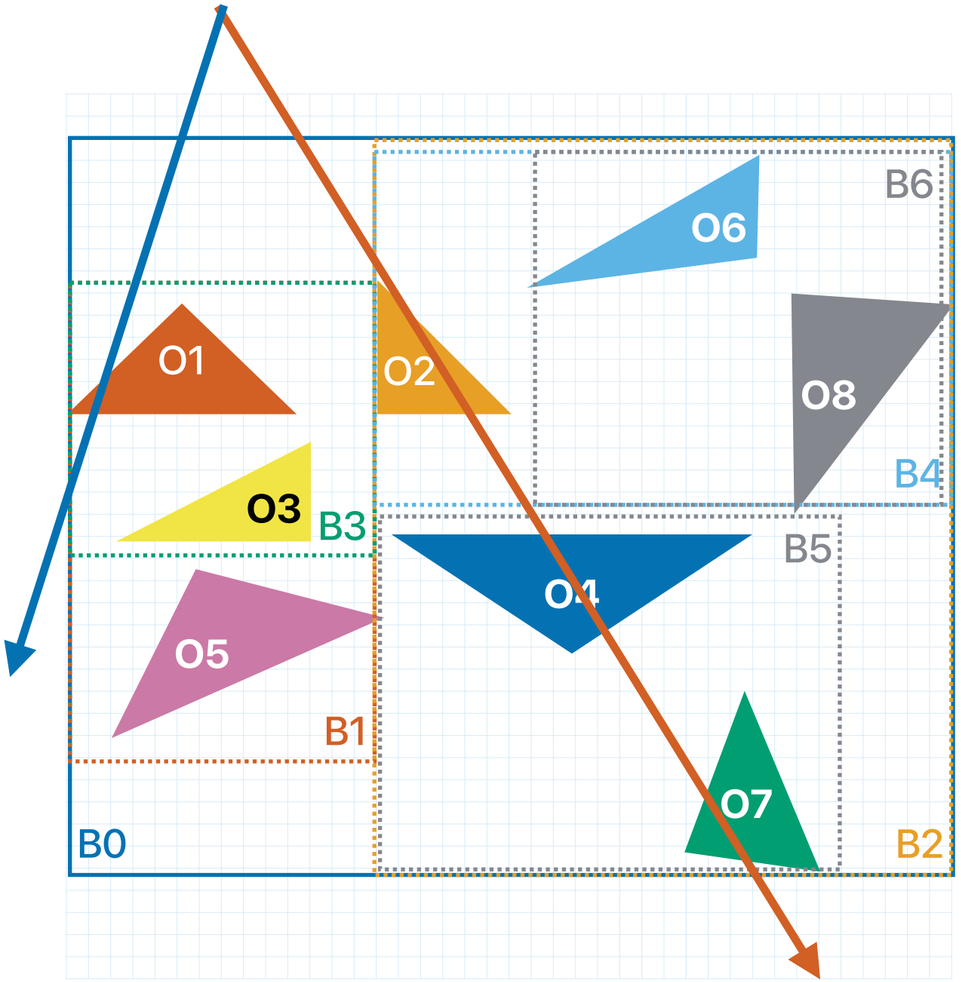
```
for (int j = 0; j < imageHeight; ++j) {
    for (int i = 0; i < imageWidth; ++i) {
        // compute primary ray direction
        Ray primRay;
        computePrimRay(i, j, &primRay);
        // shoot prim ray in the scene and search for the intersection
        Point pHit;
        Normal nHit;
        float minDist = INFINITY;
        Object object = NULL;
        for (int k = 0; k < objects.size(); ++k) {
            if (Intersect(objects[k], primRay, &pHit, &nHit)) {
                float distance = Distance(eyePosition, pHit);
                if (distance < minDistance) {
                    object = objects[k];
                    minDistance = distance;  //update min distance
                }
            }
        }
        if (object != NULL) {
            // compute illumination
            Ray shadowRay;
            shadowRay.direction = lightPosition - pHit;
            bool isShadow = false;
            for (int k = 0; k < objects.size(); ++k) {
                if (Intersect(objects[k], shadowRay)) {
                    isInShadow = true;
                    break;
                }
            }
        }
        if (!isInShadow)
            pixels[i][j] = object->color * light.brightness;
        else
            pixels[i][j] = 0;
    }
}
```

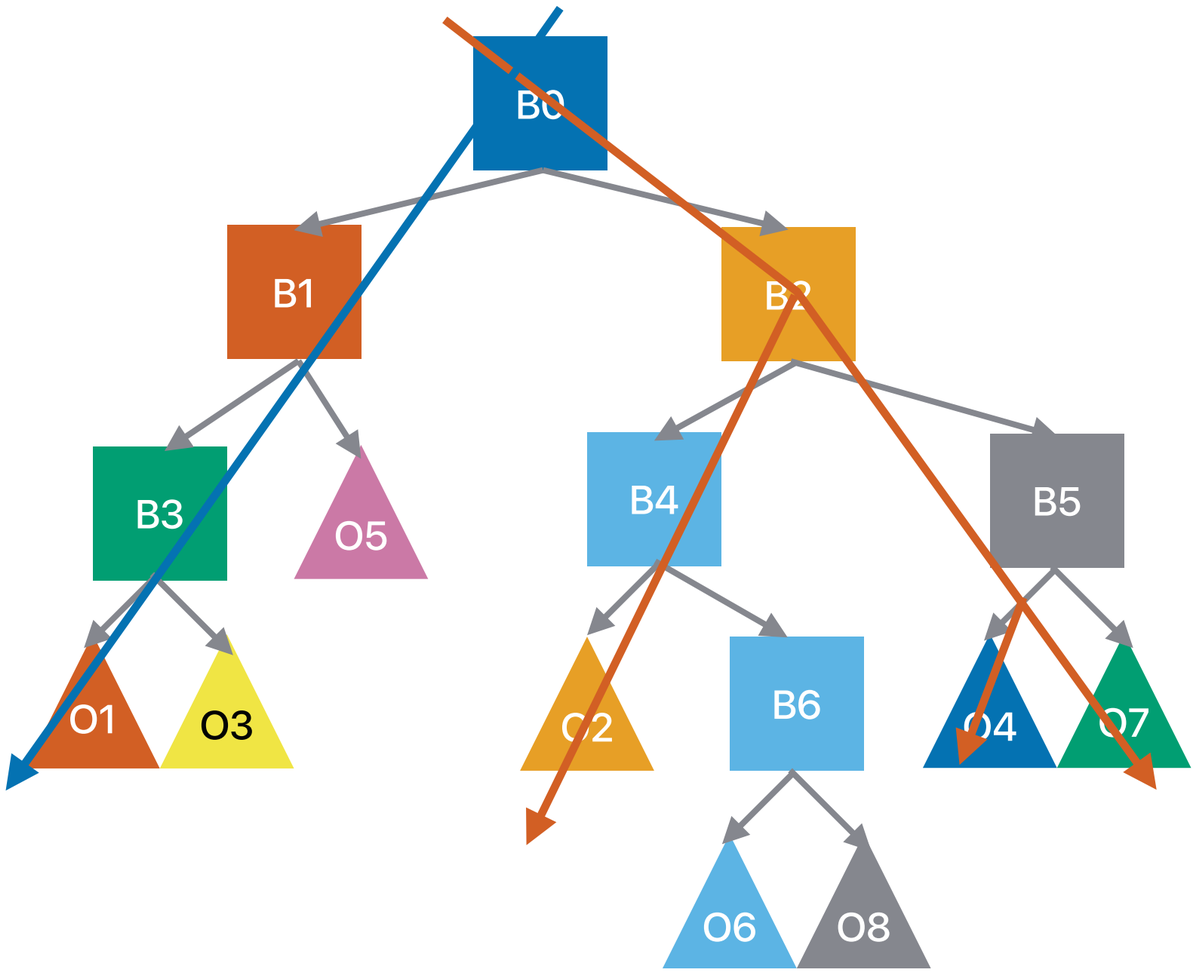
https://www.scratchapixel.com/lessons/3d-basic-rendering/introduction-to-ray-tracing/raytracing-algorithm-in-a-nutshell.html

Ray tracing pipeline



Accelerated search structure: bounding volume hierarchy tree





Recent development of NVIDIA GPUs

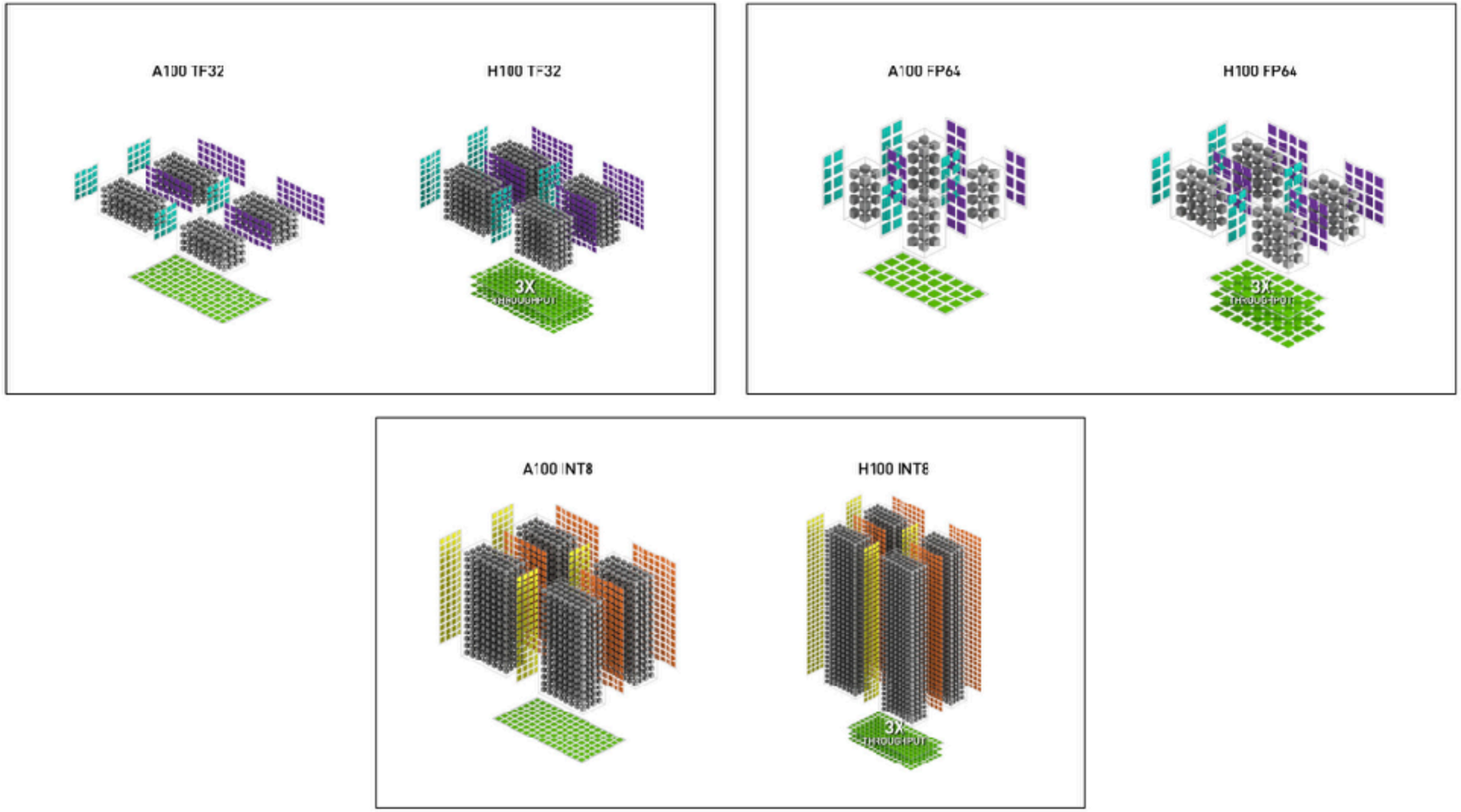


Table 2. H100 speedup over A100 (H100 Performance, TC=Tensor Core)

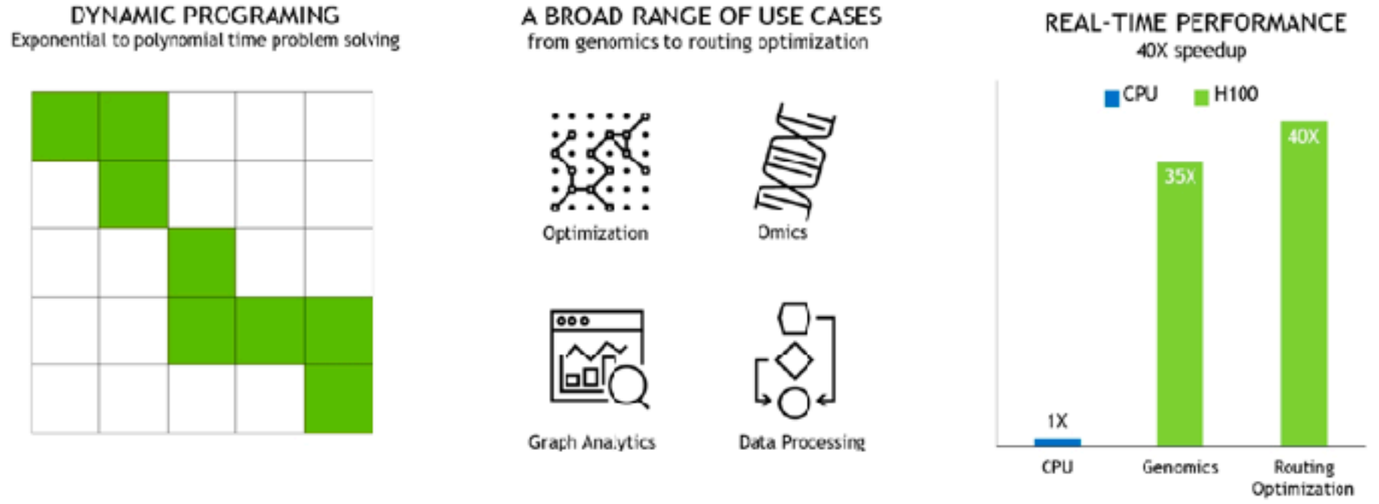
	A100	A100 Sparse	H100 SXM5	H100 SXM5 Sparse	H100 SXM5 Speedup vs A100
FP8 Tensor Core	NA	NA	1978.9 TFLOPS	3957.8 TFLOPS	6.3x vs A100 FP16 TC
FP16	78 TFLOPS	NA	133.8 TFLOPS	NA	1.7x
FP16 Tensor Core	312 TFLOPS	624 TFLOPS	989.4 TFLOPS	1978.9 TFLOPS	3.2x
BF16 Tensor Core	312 TFLOPS	624 TFLOPS	989.4 TFLOPS	1978.9 TFLOPS	3.2x
FP32	19.5 TFLOPS	NA	66.9 TFLOPS	NA	3.4x
TF32 Tensor Core	156 TFLOPS	312 TFLOPS	494.7 TFLOPS	989.4 TFLOPS	3.2x
FP64	9.7 TFLOPS	NA	33.5 TFLOPS	NA	3.5x
FP64 Tensor Core	19.5 TFLOPS	NA	66.9 TFLOPS	NA	3.4x
INT8 Tensor Core	624 TOPS	1248 TOPS	1978.9 TFLOPS	3957.8 TFLOPS	3.2x

New DPX Instructions for Accelerated Dynamic Programming

Many "brute force" optimization algorithms have the property that a sub-problem solution is reused many times when solving the larger problem. Dynamic Programming is an algorithmic technique for solving a complex recursive problem by breaking it down into simpler sub-problems. By storing the results of sub-problems, without the need to recompute them when needed later, Dynamic Programming algorithms reduce the computational complexity of exponential problem sets to a linear scale.

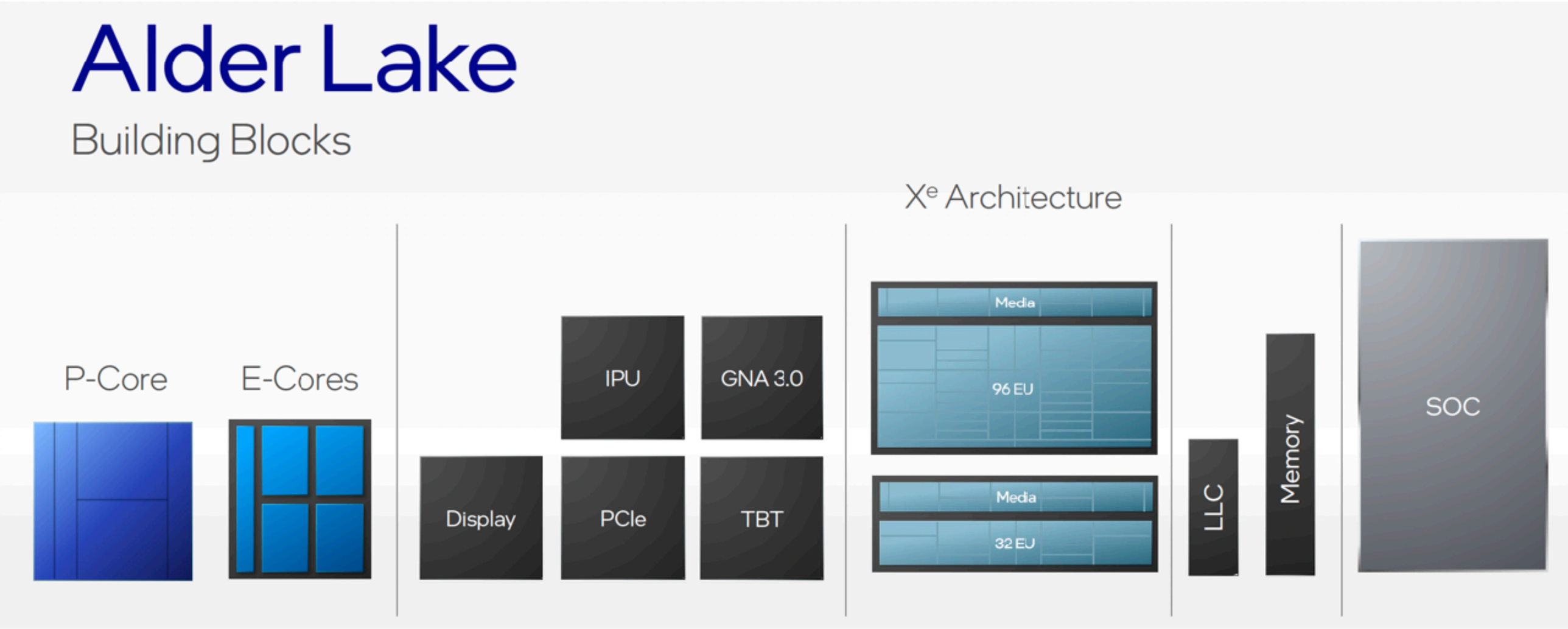
Dynamic programming is commonly used in a broad range of optimization, data processing, and genomics algorithms. In the rapidly growing field of genome sequencing, the Smith-Waterman dynamic programming algorithm is one of the most important methods in use. In the robotics space, Floyd-Warshall is a key algorithm used to find optimal routes for a fleet of robots through a dynamic warehouse environment in real-time.

H100 introduces DPX instructions to accelerate the performance of Dynamic Programming algorithms by up to 7x compared to Ampere GPUs. These new instructions provide support for advanced fused operands for the inner loop of many DP algorithms. This will lead to dramatically faster times-to-solutions in disease diagnosis, logistics routing optimizations, and even graph analytics.

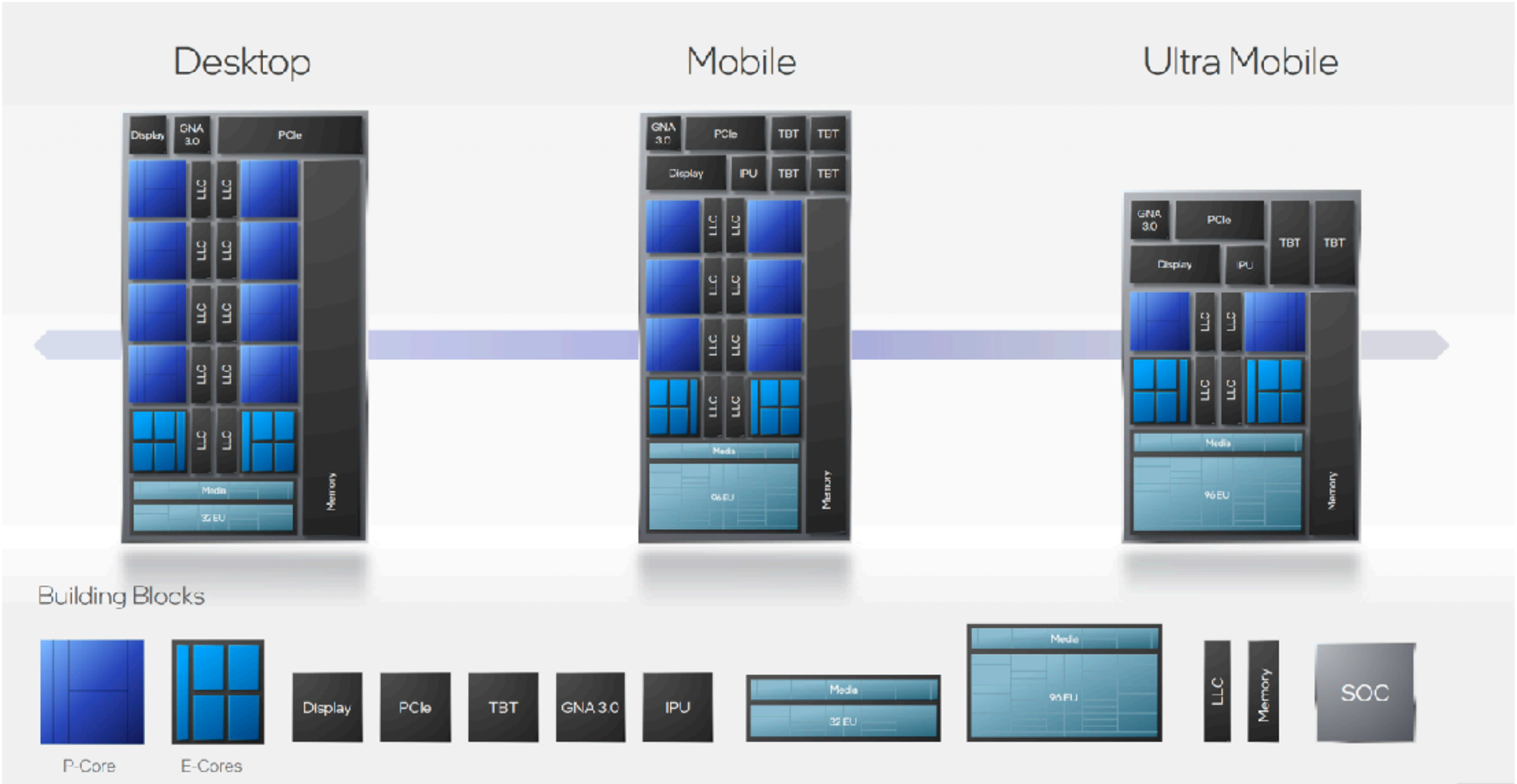


Processors are also heterogeneous now!

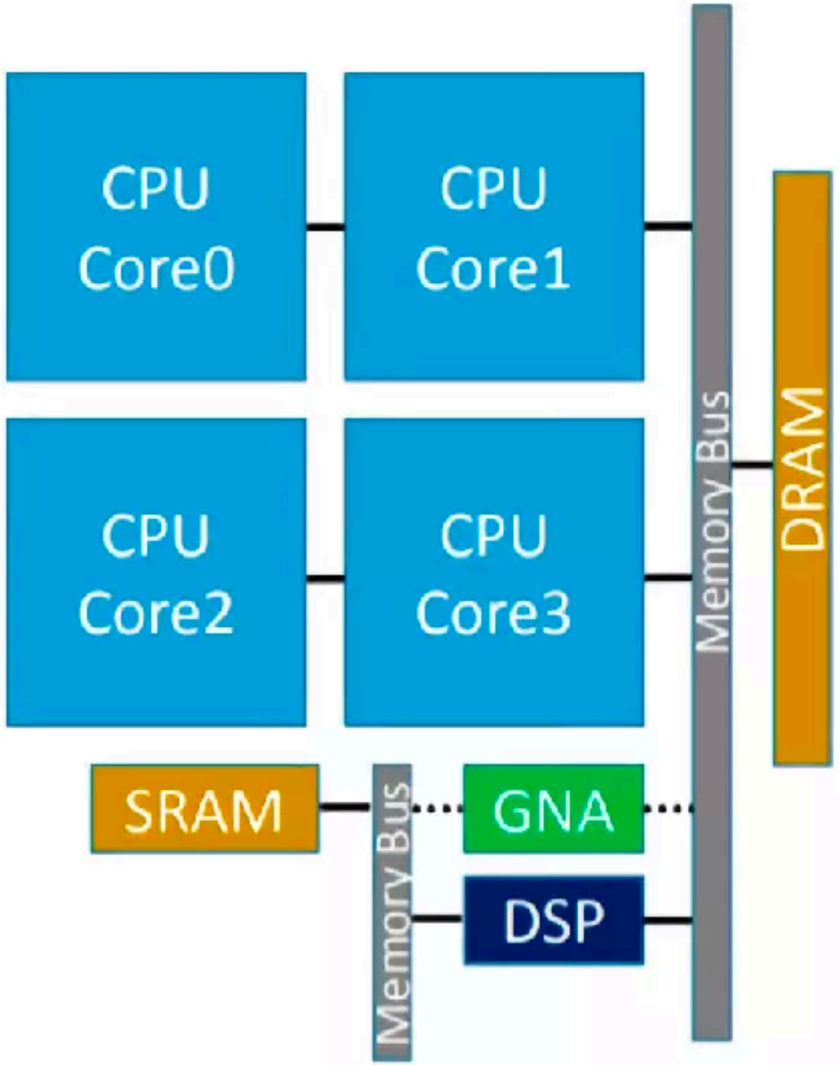
The intel Alder Lake



The intel Alder Lake



intel processor architecture



3:00

Is SoC (i.e., a processor with GPUs/ accelerators in the same package) a good idea compared with standalone accelerators?

Single-chip or standalone ones?

Is single-chip, heterogeneous processor a good idea?

- Potentially reduce the system interconnect bandwidth demand (still need to use intra-chip interconnect)
- Reduce the total system size and cost
- The design of the memory controller is complicated. For example, CPUs are more latency sensitive and want pairs of scalar values. GPUs/accelerators are more bandwidth demanding and want pairs of vectors
  - J. Power, M. D. Hill and D. A. Wood, "Supporting x86-64 address translation for 100s of GPU lanes," 2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA), 2014, pp. 568-578
- The power consumption per-chip is still limiting the overall performance
  - You can have only moderate processor cores with moderate GPU cores
  - You can have performance processor cores with wimpy graphic cores
  - You have to dynamically switch each part on/off
- Or you have to increase power consumption — NVIDIA's H100 goes up to 700W!!!

Is system-wide heterogeneous processing a better idea?

- Easier for heat dissipation
  - Each processor can be more powerful
    - gamers/datacenters still prefers discrete GPUs
    - Cloud TPUs are standalone ones
- System size is larger, cost of ownership can be higher
- Data movement going through system interconnects

Programming interface — NV's WMMA

```
__global__ void WMMAF16TensorCore(half *A, half *B, float *C, float *D) {

    int ix = (blockIdx.x * blockDim.x + threadIdx.x)/WARP_SIZE;
    int iy = (blockIdx.y * blockDim.y + threadIdx.y);

    wmma::fragment<wmma::matrix_a, M, N, K, half, wmma::row_major> a_frag;
    wmma::fragment<wmma::matrix_b, M, N, K, half, wmma::col_major> b_frag;
    wmma::fragment<wmma::accumulator, M, N, K, float> ab_frag;
    wmma::fragment<wmma::accumulator, M, N, K, float> c_frag;

    wmma::fill_fragment(ab_frag, 0.0f);

    // AB = A*B

    int a_col, a_row, b_col, b_row, c_col, c_row;

    a_row = ix * M;
    b_row = iy * N;

    for (int k=0; k<K_TOTAL; k+=K) {
        a_col = b_col = k;

        if (a_row < M_TOTAL && a_col < K_TOTAL && b_row < K_TOTAL && b_col <
N_TOTAL) {
            // Load the inputs
            wmma::load_matrix_sync(a_frag, A + a_col + a_row * M_TOTAL,
M_TOTAL);
            wmma::load_matrix_sync(b_frag, B + b_col + b_col * K_TOTAL,
K_TOTAL);

            // Perform the matrix multiplication
            wmma::mma_sync(ab_frag, a_frag, b_frag, ab_frag);
        }
    }

    // D = AB + C
    c_col = b_row;
    c_row = a_row;

    if (c_row < M_TOTAL && c_col < N_TOTAL) {
        wmma::load_matrix_sync(c_frag, C + c_col + c_row * N_TOTAL, N_TOTAL,
wmma::mem_row_major);

        for (int i = 0; i < c_frag.num_elements; i++) {
            c_frag.x[i] = ab_frag.x[i] + c_frag.x[i];
        }
        // Store the output
        wmma::store_matrix_sync(D + c_col + c_row * N_TOTAL, c_frag,
N_TOTAL, wmma::mem_row_major);
    }
}
```

intel GNA

```
// Open selected  device
status = Gna2DeviceOpen(deviceIndex);
CleanupOrError(status, deviceIndex, nullptr, GNA2_DISABLED, GNA2_DISABLED,
               "main", "Gna2DeviceOpen()");

/* Calculate model memory parameters for GnaAlloc. */
int buf_size_weights = Gna2RoundUpTo64(sizeof(weights));
// note that buffer alignment to 64-bytes is required by GNA HW
int buf_size_inputs = Gna2RoundUpTo64(sizeof(inputs));
int buf_size_biases = Gna2RoundUpTo64(sizeof(biases));
int buf_size_outputs = Gna2RoundUpTo64(H * B * 4); // (4 out vectors, H elems
in each one, 4-byte elems)

auto rw_buffer_size = Gna2RoundUp(buf_size_inputs + buf_size_outputs, 0x1000);
auto bytes_requested = rw_buffer_size + buf_size_weights + buf_size_biases;

// Allocate GNA memory (obtains pinned memory shared with the device)
uint32_t bytes_granted;
void* memory = nullptr;
status = Gna2MemoryAlloc(bytes_requested, &bytes_granted, &memory);

/* Prepare model memory layout. */
auto model_memory = reinterpret_cast<uint8_t*>(memory);

auto rw_buffers = model_memory;

auto pinned_inputs = reinterpret_cast<int16_t*>(rw_buffers);
memcpy_s(pinned_inputs, buf_size_inputs, inputs, sizeof(inputs)); // puts the
inputs into the pinned memory
rw_buffers += buf_size_inputs; // fast-forwards current pinned memory pointer
to the next free block

auto pinned_outputs = reinterpret_cast<int32_t*>(rw_buffers);
rw_buffers += buf_size_outputs; // fast-forwards the current pinned memory
pointer by the space needed for outputs

model_memory += rw_buffer_size;
auto weights_buffer = reinterpret_cast<int16_t*>(model_memory);
memcpy_s(weights_buffer, buf_size_weights, weights, sizeof(weights)); // puts
the weights into the pinned memory
model_memory += buf_size_weights; // fast-forwards current pinned memory
pointer to the next free block

auto biases_buffer = reinterpret_cast<int32_t*>(model_memory);
memcpy_s(biases_buffer, buf_size_biases, biases, sizeof(biases)); // puts the
biases into the pinned memory
model_memory += buf_size_biases; // fast-forwards current pinned memory pointer
to the next free block

/* Prepare neural network topology,
 * Single FullyConnectedAffine layer in this example. */

/* Prepare and initialize FullyConnectedAffine operation operands with GNA API
helpers */
auto inputTensor = Gna2TensorInit2D(W, B, Gna2DataTypeInt16, pinned_inputs);
auto outputTensor = Gna2TensorInit2D(H, B, Gna2DataTypeInt32, pinned_outputs);
auto weightTensor = Gna2TensorInit2D(H, W, Gna2DataTypeInt16, weights_buffer);
auto biasTensor = Gna2TensorInit1D(H, Gna2DataTypeInt32, biases_buffer);
auto activationTensor = Gna2TensorInitDisabled();

/* Create single FullyConnectedAffine operation (layer) */
auto operation = Gna2Operation{};
status = Gna2OperationInitFullyConnectedAffine(&operation, customAlloc,
                                               &inputTensor, &outputTensor,
&weightTensor, &biasTensor,
                                               &activationTensor);

/* Create data-flow model with single operation (layer) */
Gna2Model model = {1, &operation};
uint32_t modelId = GNA2_DISABLED;
status = Gna2ModelCreate(deviceIndex, &model, &modelId);

// Create request configuration used for queueing inference requests
uint32_t configId = GNA2_DISABLED;
status = Gna2RequestConfigCreate(modelId, &configId);

// Set model input data buffer, operation 0, operand 0, for this sample
status = Gna2RequestConfigSetOperandBuffer(configId, 0, 0, pinned_inputs);

// Set model output data buffer, operation 0, operand 1, for this sample
status = Gna2RequestConfigSetOperandBuffer(configId, 0, 1, pinned_outputs);

// [optional] Set acceleration mode automatic (software emulation used if no
hardware detected)
status = Gna2RequestConfigSetAccelerationMode(configId,
Gna2AccelerationModeAuto);

// Enqueue inference request (non-blocking call)
```

3:00

How do hardware accelerators change our programming paradigms?
What’re the benefits and limitations of the new programming paradigm?

Lessons learned from accelerators?

- Accelerators “lift up” the roofline
  - Applications/compute kernels with higher arithmetic densities may be feasible
    - NN is feasible after GPGPU
    - Trade “complexity” with parallelism
  - Applications are more likely to be memory-bound
  - Your software should try to avoid frequent memory access
    - Try to use memory closer to the processing elements
  - The hardware design must not ignore the importance of memory bandwidth
- The most “efficient” system design must land on the “turning point” of your roofline model
  - TPU’s 167GB/sec memory bandwidth is an example

Roogle Project Meetings

- Make an appointment through the Google Calendar
- We're trying to make a company project theme
- Available resources
  - SmartSSD, Edge TPUs, CUDA GPUs, Tensor Cores, or something else
- Project ideas
  - Accelerating applications through AI/ML accelerators (e.g., EdgeTPUs or tensor cores)
  - Accelerating applications through innovative parallel programming models that hardware accelerators enable and evaluate the result using FPGAs
  - Static tools to perform roofline analysis on AI/ML workloads
  - Accelerating applications through intelligent storage devices (smartSSD)
  - Literature review (more than 20 papers) on a certain topic related to this class
  - Anything related to what we discussed in this class!

Electrical
Computer Science
Engineering

277

つづく