https://hao-ai-lab.github.io/cse234-w25/

CSE 234: Data Systems for Machine Learning
Winter 2025

LLMSys

Optimizations and Parallelization

MLSys Basics

Enrollment Updates

- We have sent out two batches of enrollment invitations (on Wed and Thursday)
  - To ~100 waitlisted students in total
  - With a 24h expiration
- If there are still rooms (which is highly likely)
  - We will send the next batch Friday morning

Dataflow Graph

Autodiff

Graph Optimization

Parallelization

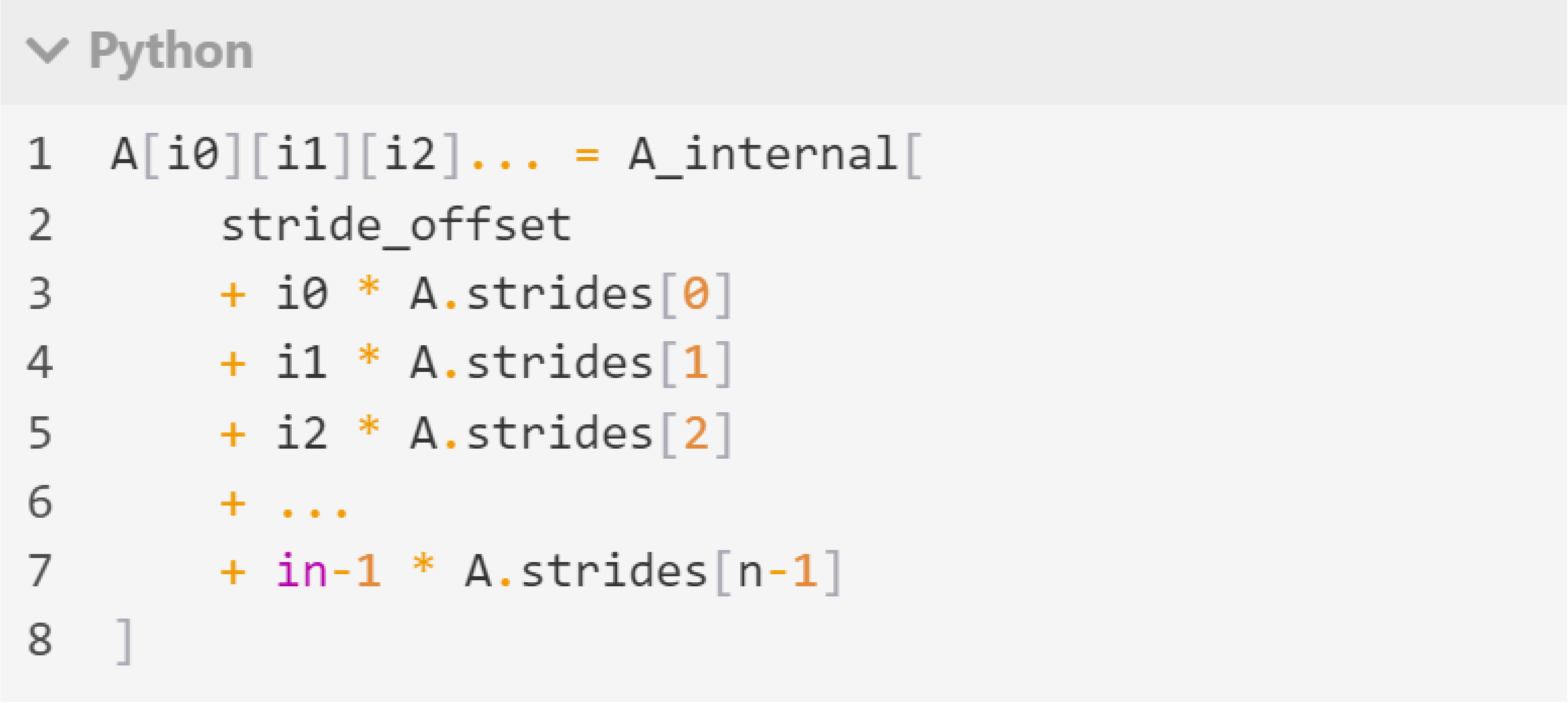
Runtime: schedule / memory

Operator

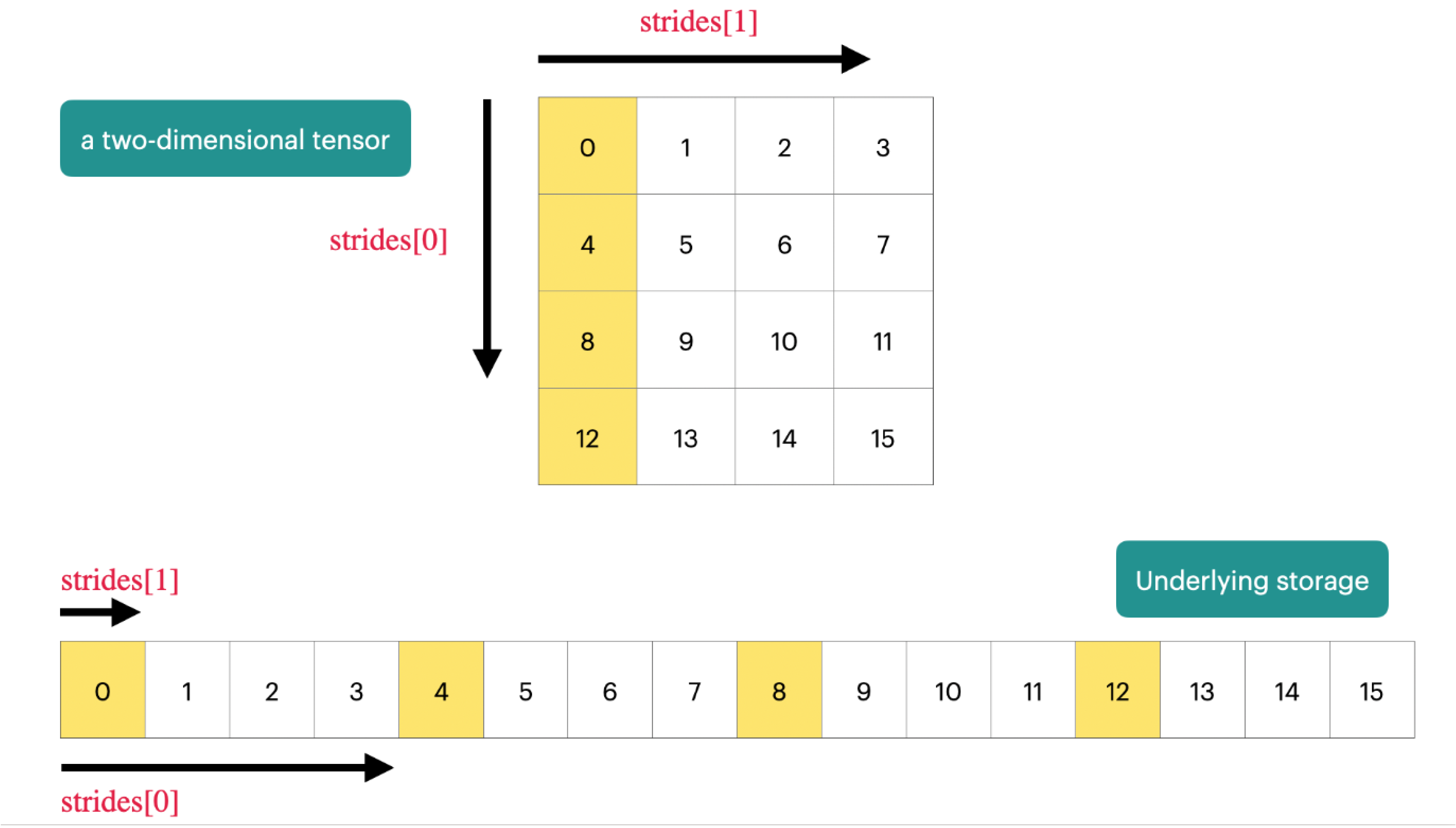
Today’s Learning Goals

- How to make operators fast in general?
  - ✅ Vectorize
  - ❓ Data layout
  - ❓ Parallelization (at the operator level)
- Matmul-specific optimization
- GPUs and accelerators
  - High-level Idea
  - The accelerator market

Recap: Strides



```python
A[i0][i1][i2]... = A_internal[
    stride_offset
    + i0 * A.strides[0]
    + i1 * A.strides[1]
    + i2 * A.strides[2]
    + ...
    + in-1 * A.strides[n-1]
]
```

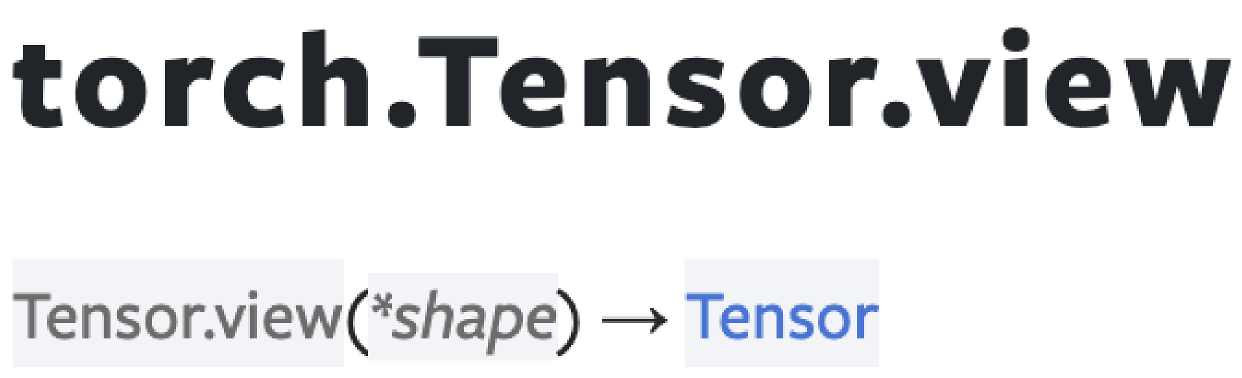


0	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

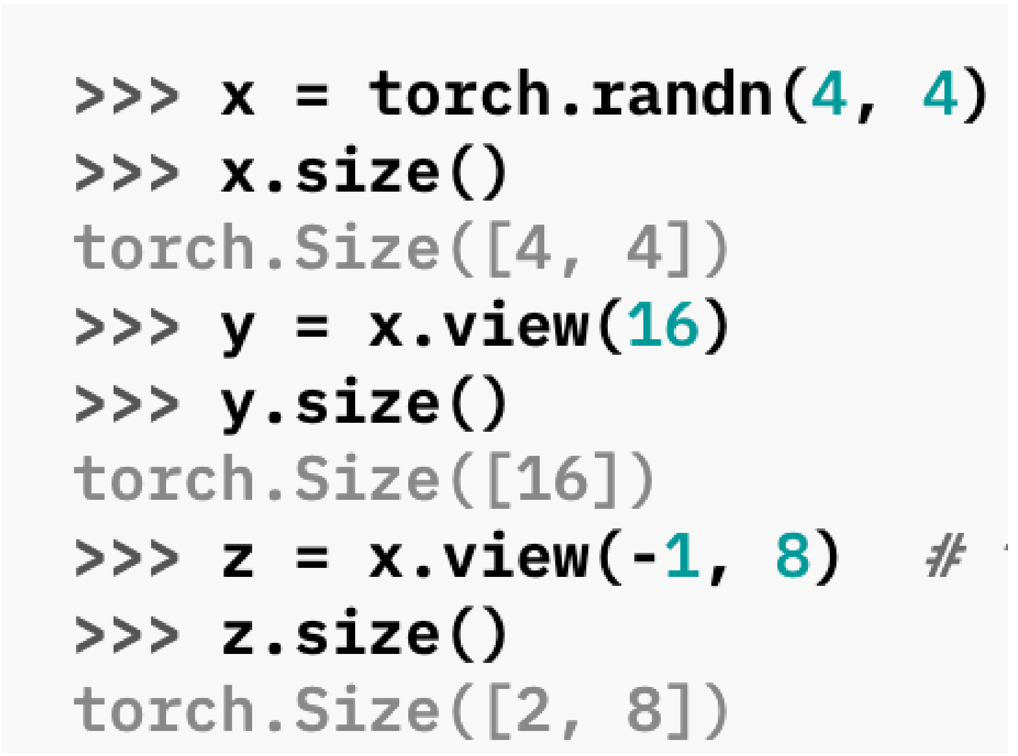
Why we bother saving “strides” when saving tensors

- Strides can separate the underlying storage and the view of the tensor



torch.Tensor.view

Tensor.view(*shape) → Tensor



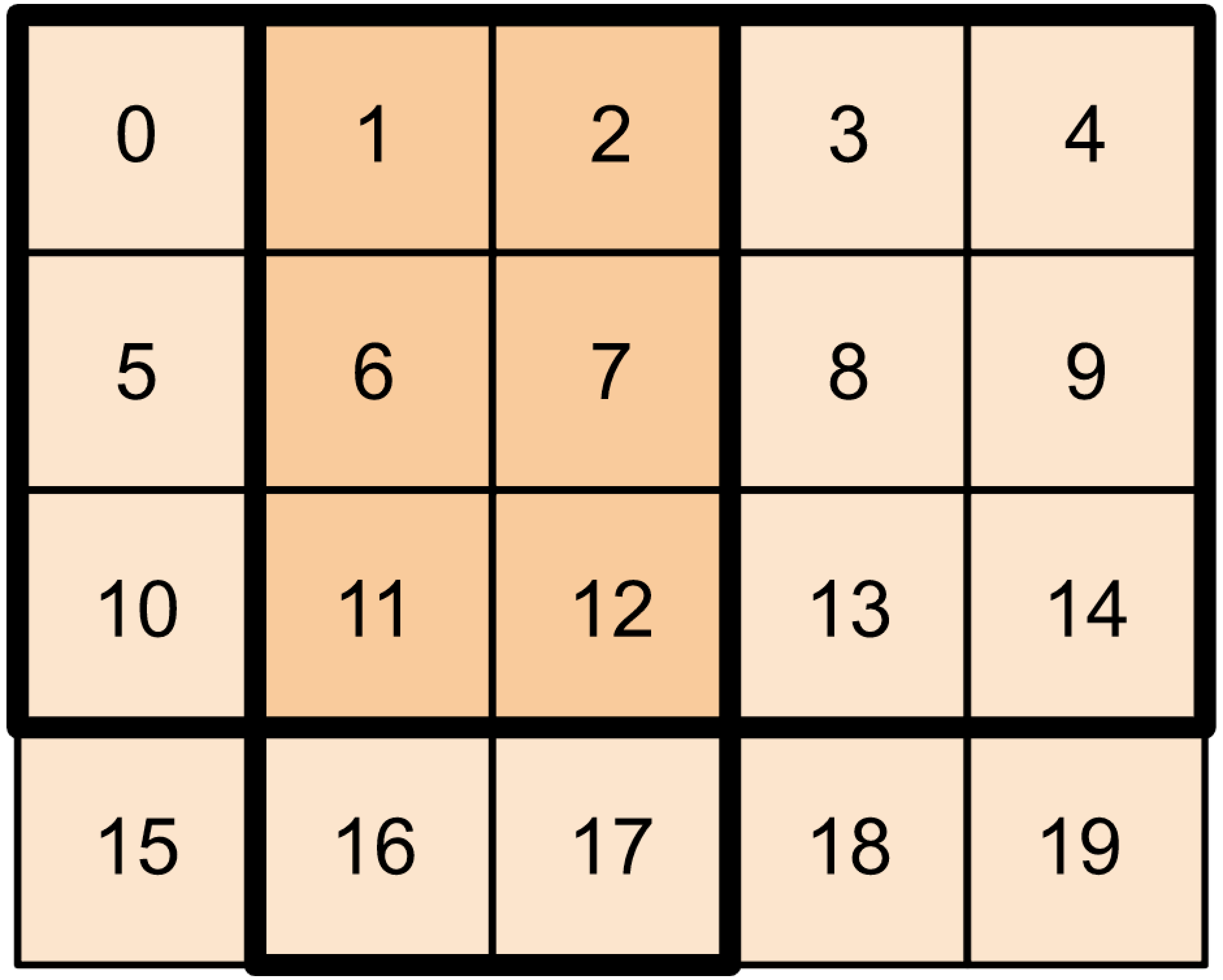
```
>>> x = torch.randn(4, 4)
>>> x.size()
torch.Size([4, 4])
>>> y = x.view(16)
>>> y.size()
torch.Size([16])
>>> z = x.view(-1, 8)  #
>>> z.size()
torch.Size([2, 8])
```

- Enable zero-copy of some very frequently used operators

Operator: Slice

- Change the offset by +1
- Reduce the shape to [3, 2]

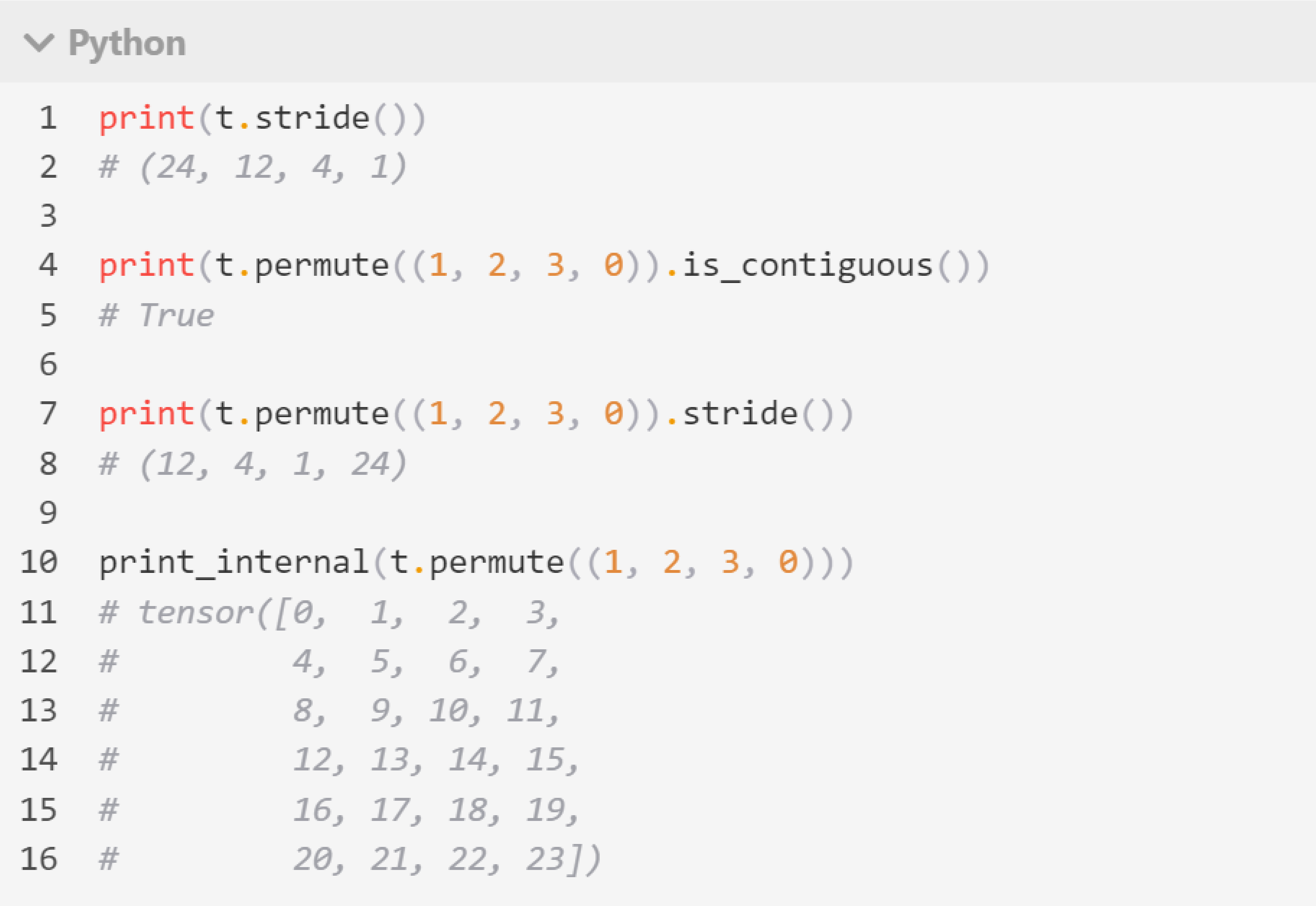
- Note: zero copy



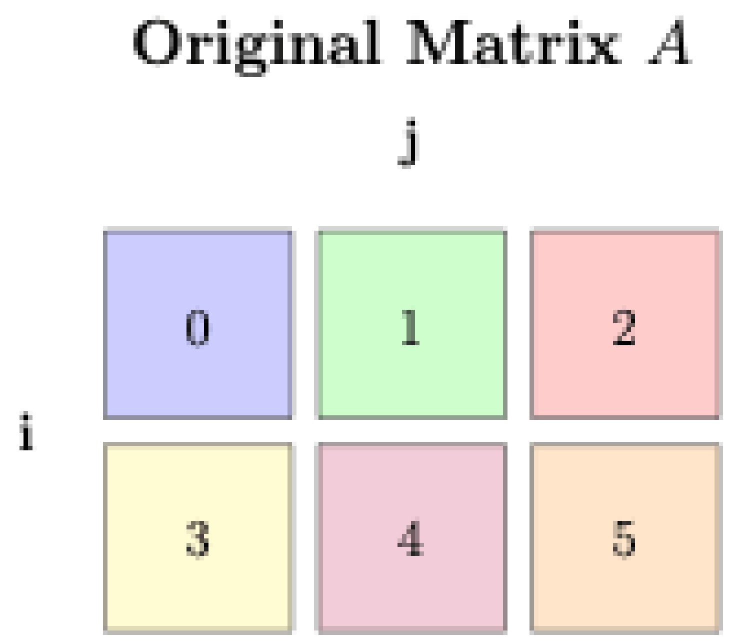
0	1	2	3	4
5	6	7	8	9
10	11	12	13	14
15	16	17	18	19

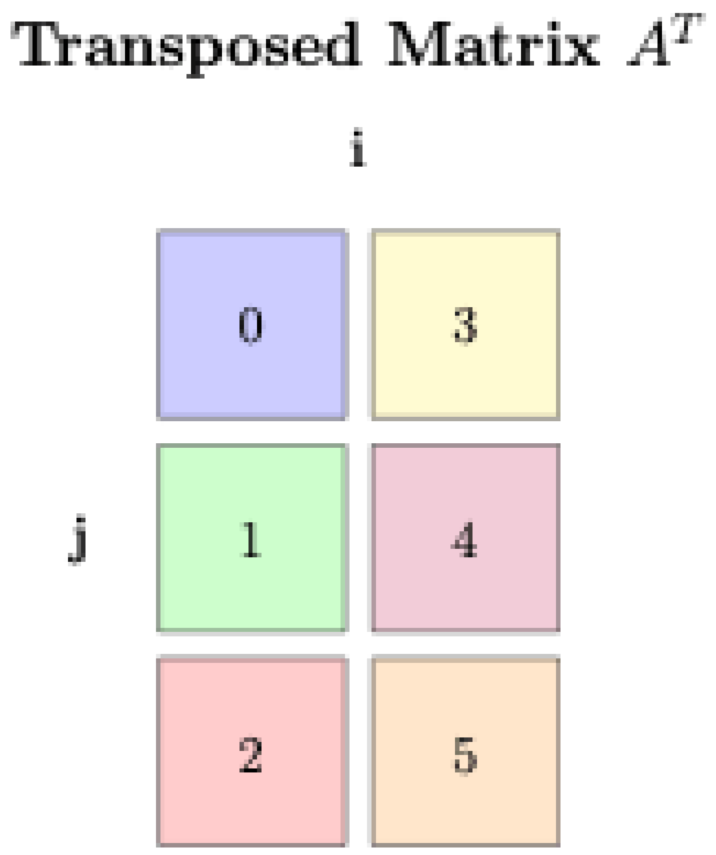
Operator: Transpose

- How to do Transpose?



```python
print(t.stride())
# (24, 12, 4, 1)

print(t.permute((1, 2, 3, 0)).is_contiguous())
# True

print(t.permute((1, 2, 3, 0)).stride())
# (12, 4, 1, 24)

print_internal(t.permute((1, 2, 3, 0)))
# tensor([0,   1,   2,   3,
#         4,   5,   6,   7,
#         8,   9,  10,  11,
#         12,  13,  14,  15,
#         16,  17,  18,  19,
#         20,  21,  22,  23])
```

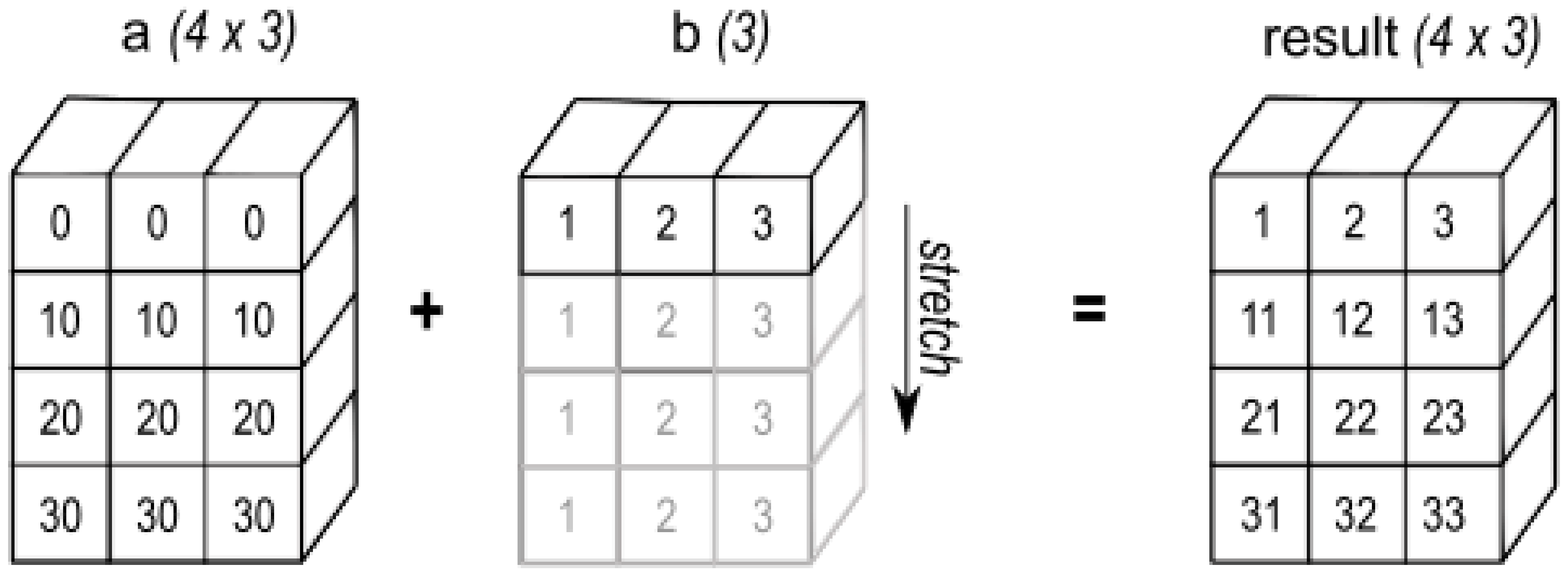

Original Matrix A


Transposed Matrix A^T

- Note 1: zero copy
- Note 2: underlying storage is unchanged

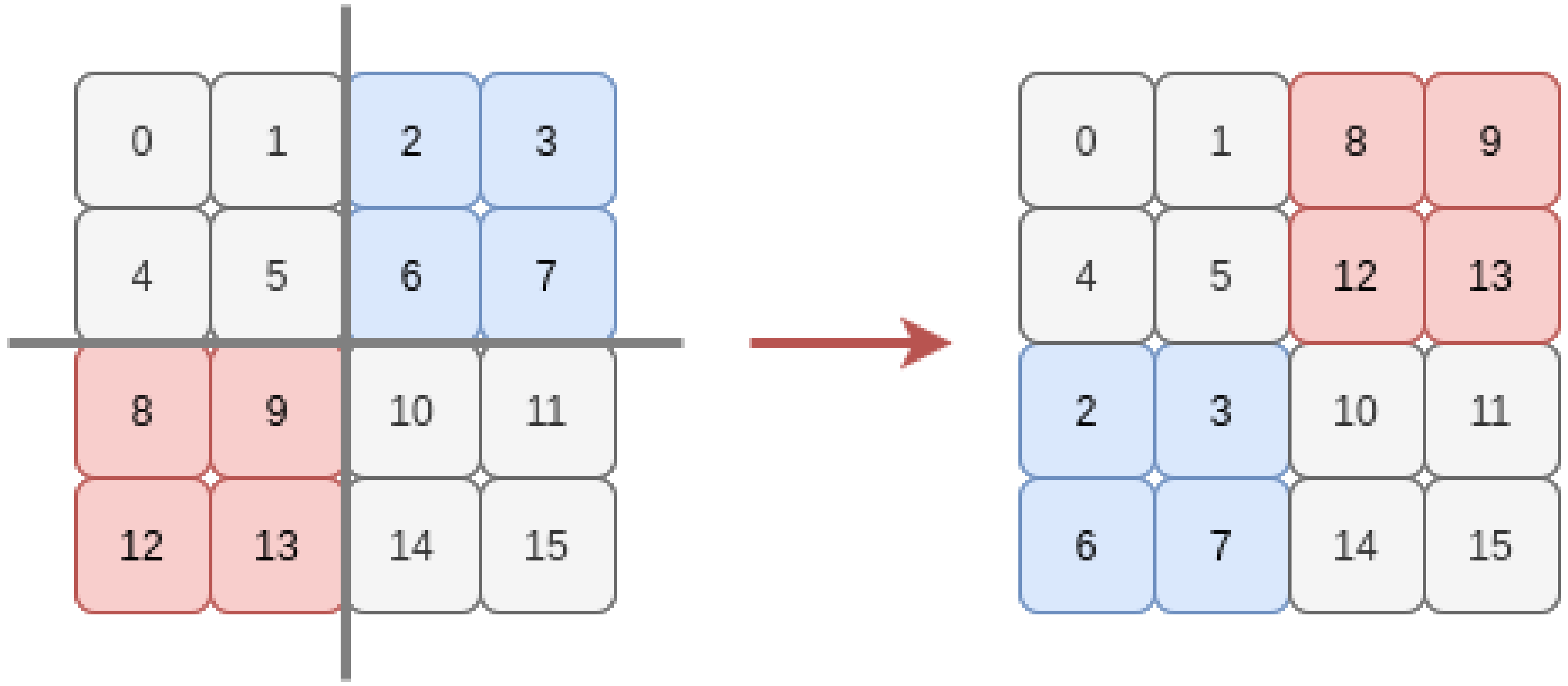
Broadcast

- Question: how to do broadcast?



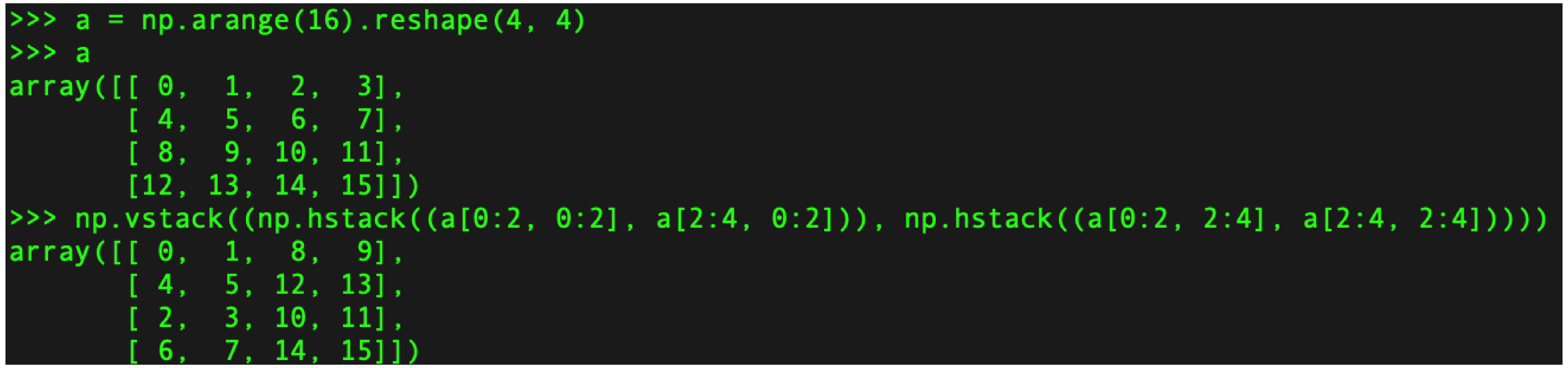
- b.strides=[1], b.shape=[1,3], b.data=[1, 2, 3]
- After broadcast: b.shape=[4,3], b.data=[1, 2, 3], b.strides=?
- Recall the def of strides: A[i, j] = A.data[offset + i*A.strides[0] + j * A.strides[1]]

Home Exercise: Swapping tiles



0	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15

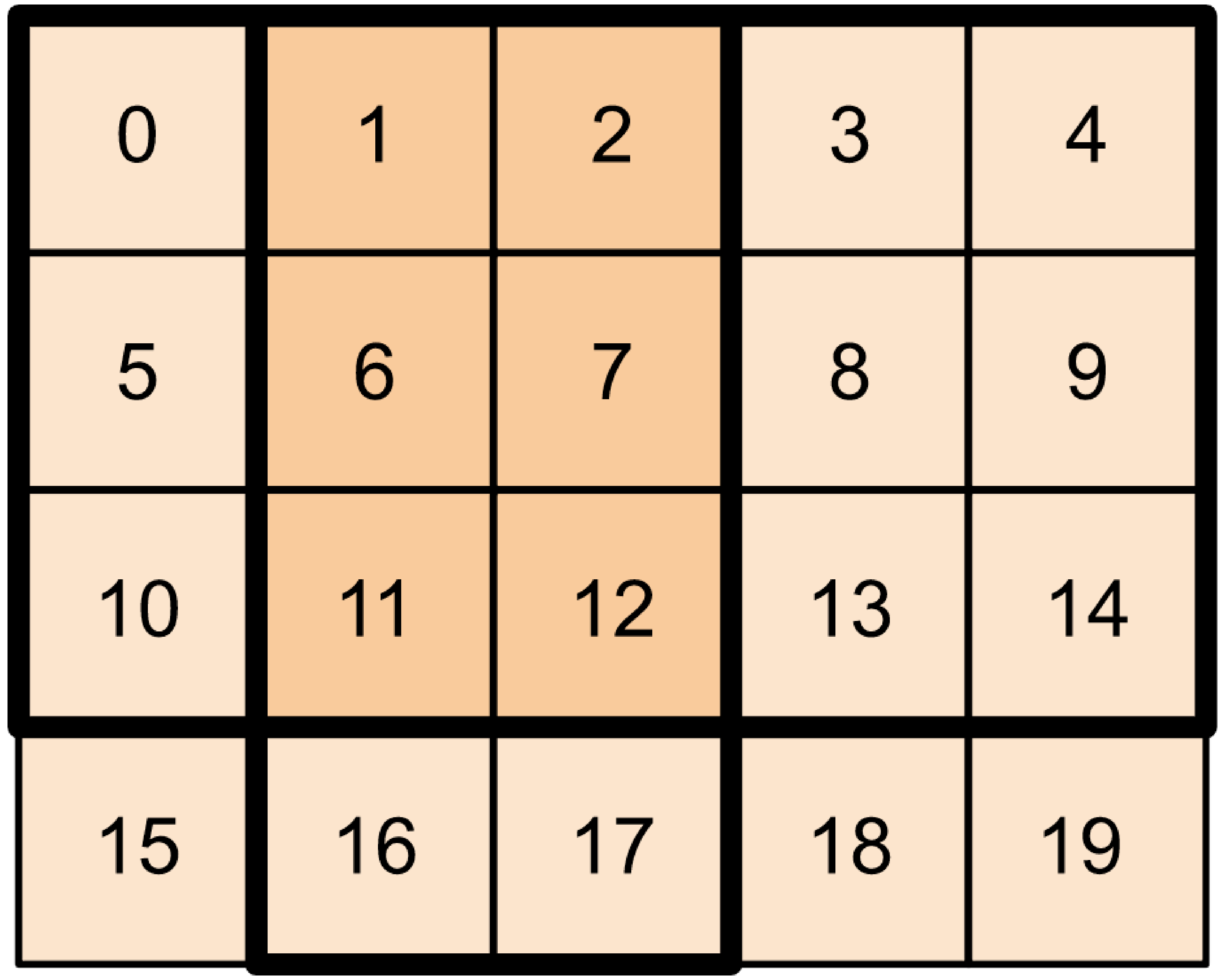
0	1	8	9
4	5	12	13
2	3	10	11
6	7	14	15



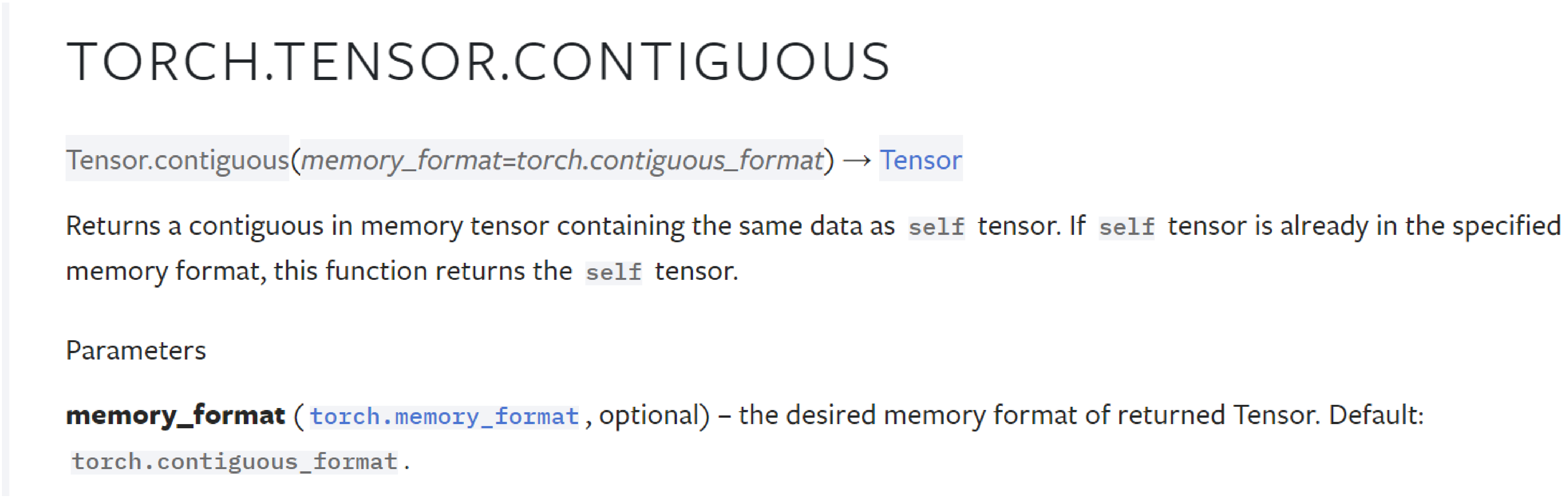
```
>>> a = np.arange(16).reshape(4, 4)
>>> a
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11],
       [12, 13, 14, 15]])
>>> np.vstack((np.hstack((a[0:2, 0:2], a[2:4, 0:2])), np.hstack((a[0:2, 2:4], a[2:4, 2:4]))))
array([[ 0,  1,  8,  9],
       [ 4,  5, 12, 13],
       [ 2,  3, 10, 11],
       [ 6,  7, 14, 15]])
```

Problems of Strides

- Memory Access may become not continuous
  - Many vectorized ops requires continuous storage
  - What's the underlying storage after slice?



0	1	2	3	4
5	6	7	8	9
10	11	12	13	14
15	16	17	18	19



TORCH.TENSOR.CONTIGUOUS

Tensor.contiguous(*memory_format=torch.contiguous_format*) → Tensor

Returns a contiguous in memory tensor containing the same data as `self` tensor. If `self` tensor is already in the specified memory format, this function returns the `self` tensor.

Parameters

memory_format (`torch.memory_format`, optional) – the desired memory format of returned Tensor. Default: `torch.contiguous_format`.

Parallelization (Elementise)

- How to parallelize the loop?

We'll com
back to this
later

```
for (int i = 0; i < 64; ++i) {
      float4 a = load_float4(A + i*4);
      float4 b = load_float4(B + i*4);
      float4 c = add_float4(a, b);
      store_float4(C + i* 4, c);
```

vectorized

```
    #pragma omp parallel for
    for (int i = 0; i < 64; ++i) {
          float4 a = load_float4(A + i*4);
          float4 b = load_float4(B + i*4);
          float4 c = add_float4(a, b);
          store_float4(C * 4, c);
}
```

Vectorized &
parallelized

Summary

- Vectorization
  - Leverage platform-specific vectorized functions
  - reduce seek time
- Data layout
  - Stride format
  - Zero copy
  - Enable fast array-manipulation: slice, transpose, broadcast, etc.
- Parallelization on CPUs

Next

- How to make operators fast in general?
  - ✅ Vectorize
  - ✅ Data layout
  - ✅ Parallelization (at the operator level)
- ❓Matmul-specific optimization
- GPUs and accelerators
  - High-level Idea
  - The accelerator market

What is Matmul in Code?

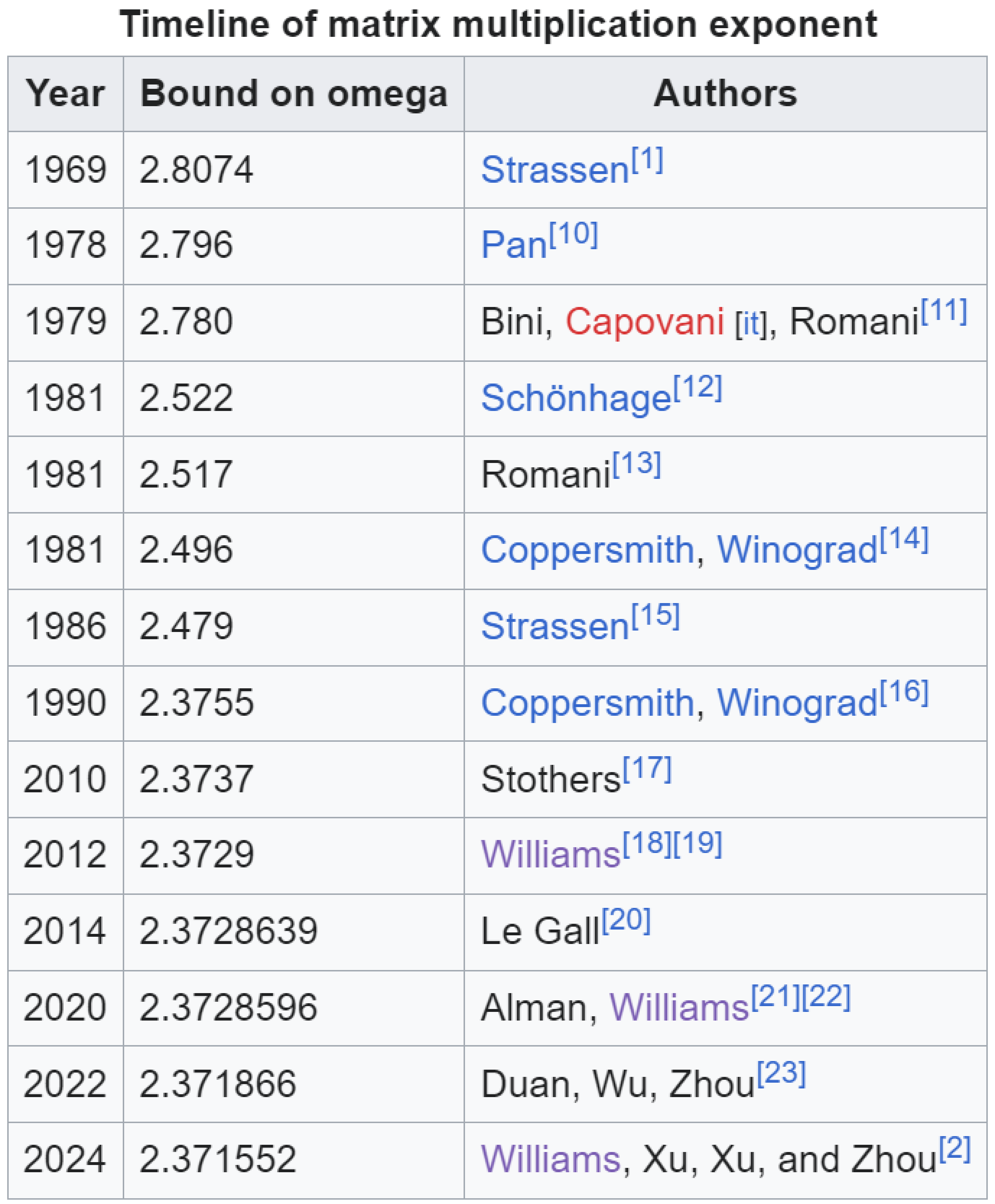
```
Compute C = dot(A, B.T)

float A[n][n], B[n][n], C[n][n];

for (int i = 0; i < n; ++i)
  for (int j = 0; j < n; ++j) {
    C[i][j] = 0;
    for (int k = 0; k < n; ++k) {
      C[i][j] += A[i][k] * B[j][k];
    }
  }
```

- What is the time complexity of 2D matmul?
- O(n^3)

- What is the best complexity we can achieve?
- O(n^2.371552)

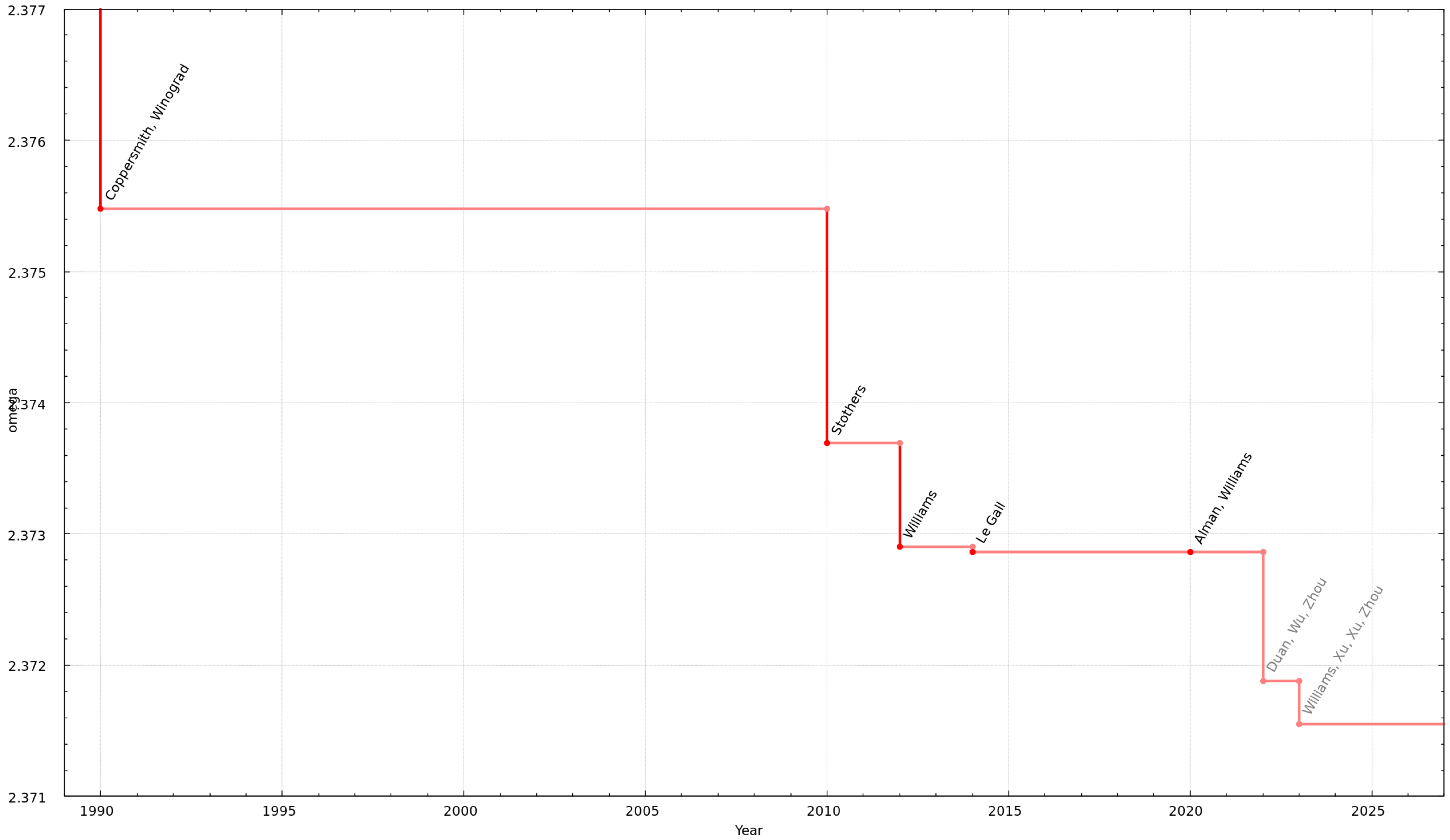
Matmul Complexity

Not a good area
to do research ☺



Timeline of matrix multiplication exponent

Year	Bound on omega	Authors
1969	2.8074	Strassen[1]
1978	2.796	Pan[10]
1979	2.780	Bini, Capovani [it], Romani[11]
1981	2.522	Schönhage[12]
1981	2.517	Romani[13]
1981	2.496	Coppersmith, Winograd[14]
1986	2.479	Strassen[15]
1990	2.3755	Coppersmith, Winograd[16]
2010	2.3737	Stothers[17]
2012	2.3729	Williams[18][19]
2014	2.3728639	Le Gall[20]
2020	2.3728596	Alman, Williams[21][22]
2022	2.371866	Duan, Wu, Zhou[23]
2024	2.371552	Williams, Xu, Xu, and Zhou[2]

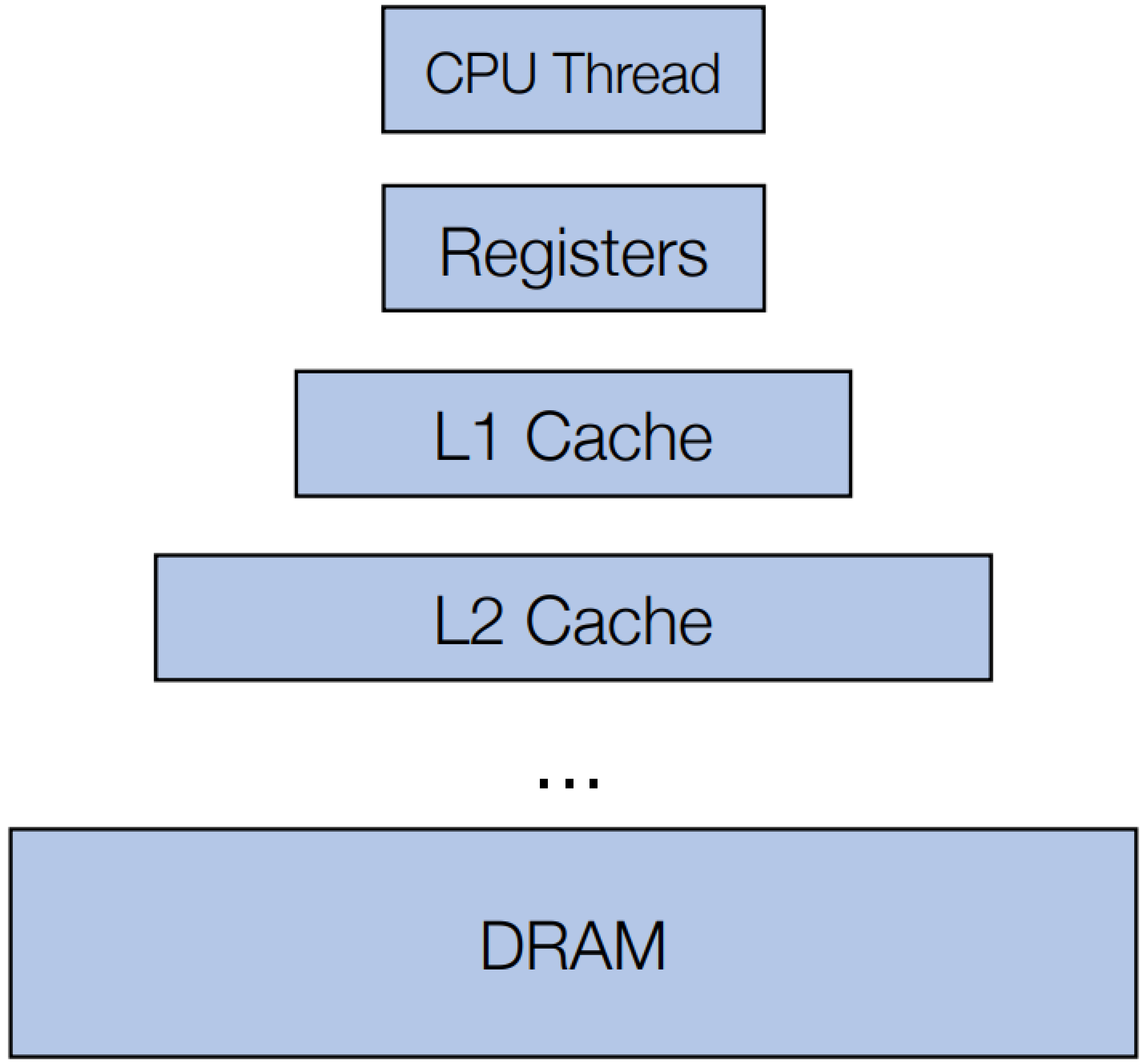


How to Make Matmul Fast

$$\text{max AI} = \overset{\uparrow}{\#\text{ops}} \,/\, \overset{\downarrow}{\#\text{bytes}}$$

Recall: Memory Hierarchy

- Ideally: we want everything to be local to processors (In registers)
- But registers are expensive and small, hence memory hierarhcy



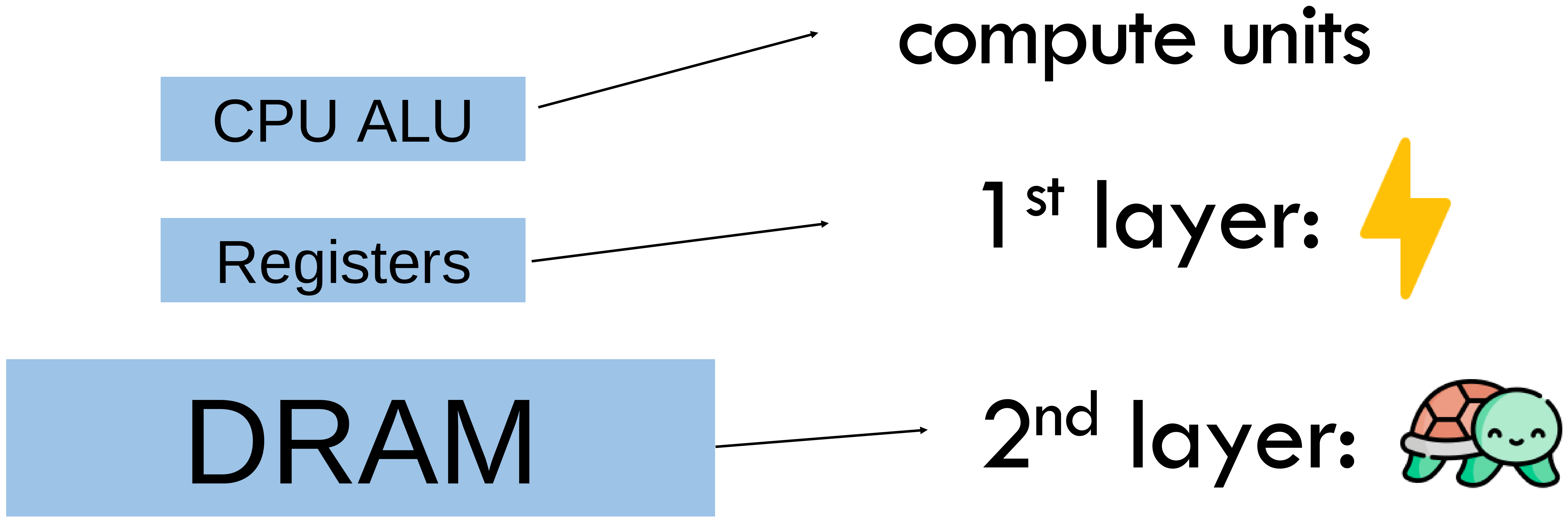
Latency
Source: Latency numbers every programmer should know

0.5 ns

7ns 14x L1 cache

200ns 20x L2 cache, 200x L1 cache

Simplify It a bit



CPU ALU → compute units

Registers → 1st layer: ⚡

DRAM → 2nd layer: 🐢

Recall How to Estimate AI: count loads

```
float*  A, *B,  *C,  *D,  *E,  *tmp1, *tmp2;
//  assume arrays are  allocated here
//  compute  E  = D + ((A  + B) *  C)
add(n, A, B, tmp1);
mul(n,  tmp1,  C, tmp2);
add(n, tmp2,  D, E);



void fused(int  n, float*  A, float*  B, float*  C, float*  D,
   float*  E) {
   for  (int  i=0; i<n; i++)
      E[i]  = D[i]  + (A[i] + B[i])  *  C[i];
}
//  compute  E  = D + (A + B) *  C
fused(n,  A, B,C,   D, E);
```

Overall arithmetic intensity = 1/3

Four loads, one store per 3 math ops
arithmetic intensity = 3/5

Review Matmul loop

```
dram float A[n][n], B[n][n], C[n][n];
for (int i = 0; i < n; ++i) {
    for (int j = 0; j < n; ++j) {
        register float c = 0;
        for (int k = 0; k < n; ++k) {
          register float a = A[i][k];
          register float b = B[j][k];
          c += a * b;
        }
        C[i][j] = c;
    }
}
```

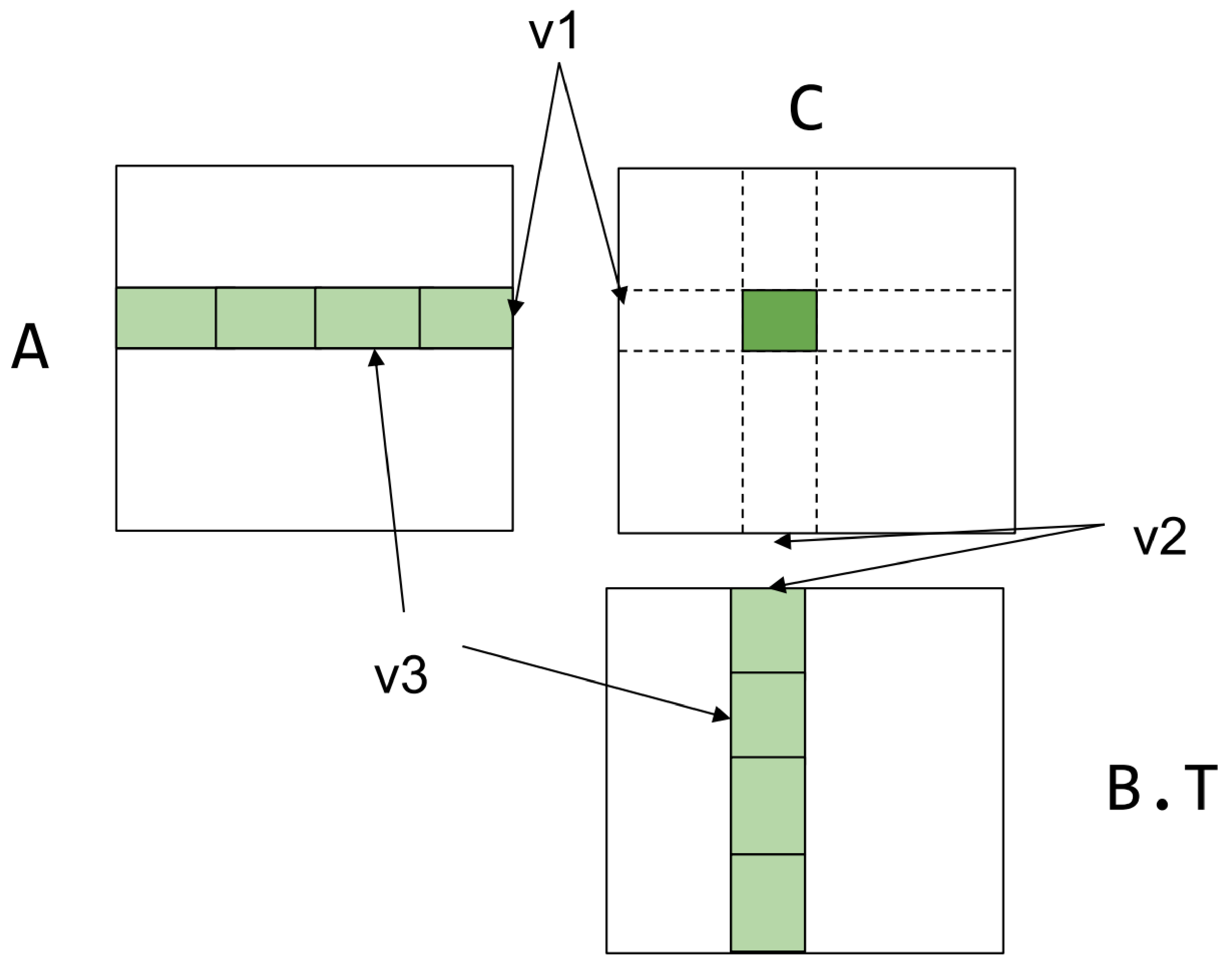
Read a	n^3
Read b	n^3
Write c	n^2

#reigsters needed:
1 + 1 + 1 =3

Read cost:
2 * n^3 * speed(dram -> register)

Register Tiled Matrix Multiplication

```
dram float A[n/v1][n/v3][v1][v3];
dram float B[n/v2][n/v3][v2][v3];
dram float C[n/v1][n/v2][v1][v2];

for (int i = 0; i < n/v1; ++i) {
   for (int j = 0; j < n/v2; ++j) {
      register float c[v1][v2] = 0;
      for (int k = 0; k < n/v3; ++k) {
        register float a[v1][v3] = A[i][k];
        register float b[v2][v3] = B[j][k];
        c += dot(a, b.T);
      }
      C[i][j] = c;
   }
}
```



Register Tiled Matrix Multiplication

```
dram float A[n/v1][n/v3][v1][v3];
dram float B[n/v2][n/v3][v2][v3];
dram float C[n/v1][n/v2][v1][v2];

for (int i = 0; i < n/v1; ++i) {
    for (int j = 0; j < n/v2; ++j) {
        register float c[v1][v2] = 0;
        for (int k = 0; k < n/v3; ++k) {
          register float a[v1][v3] = A[i][k];
          register float b[v2][v3] = B[j][k];
          c += dot(a, b.T);
        }
        C[i][j] = c;
    }
}
```

Read a	N^3 / v_2
Read b	N^3 / v_1
Write c	N^2

#reigsters needed:
v1*v3 + v2*v3 + v1*v2

Read cost:
(n^3/v2 + n^3 / v1) * speed(dram -> register)

Register Tiled Matrix Multiplication

- Q: is the load cost related to v3?

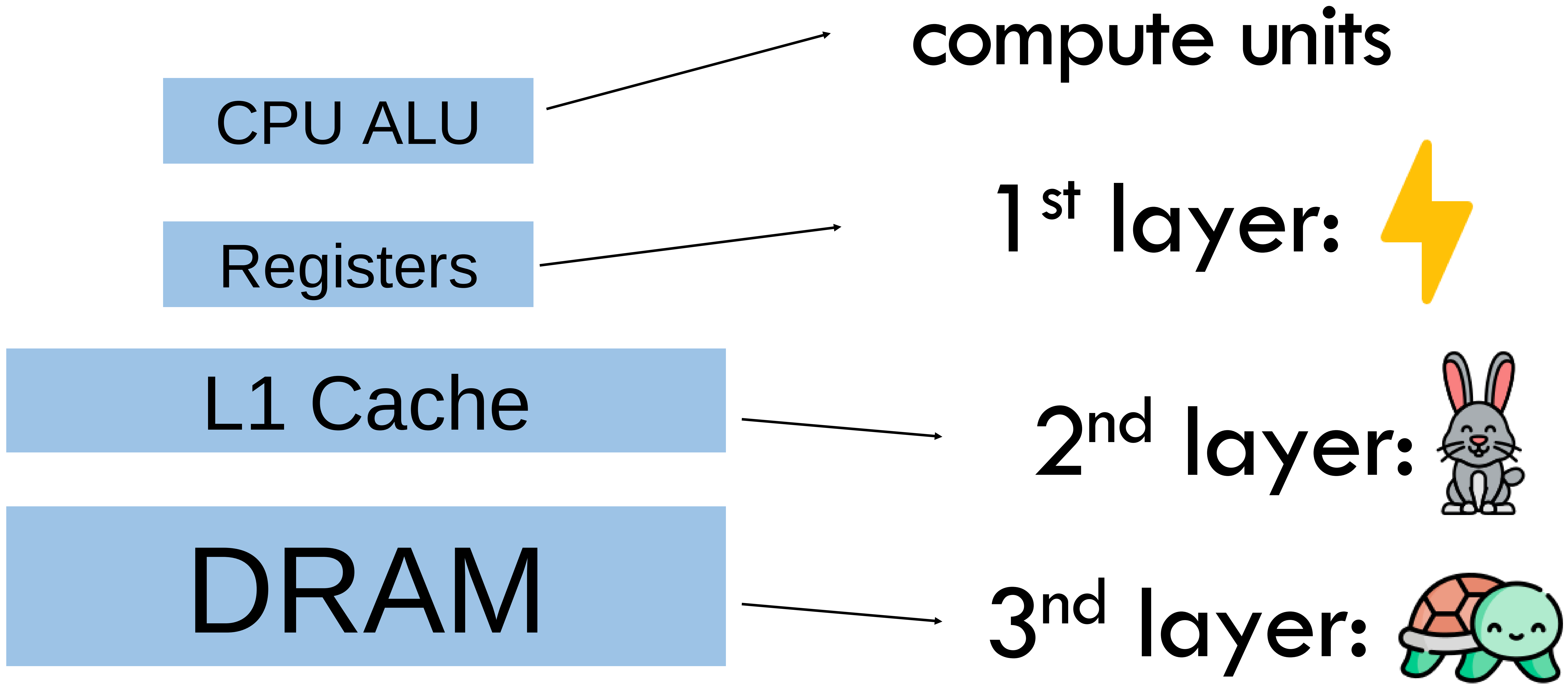
Read a	N^3 / v_2
Read b	N^3 / v_1

- Q: How to set v1 / v2?
  - What are the constraints?

#reigsters needed:
v1*v3 + v2*v3 + v1*v2

- Q: Why essentially can tiling reduce read cost?

Make it more complicated: Consider L1 cache



CPU ALU

Registers

L1 Cache

DRAM

compute units

1st layer:

2nd layer:

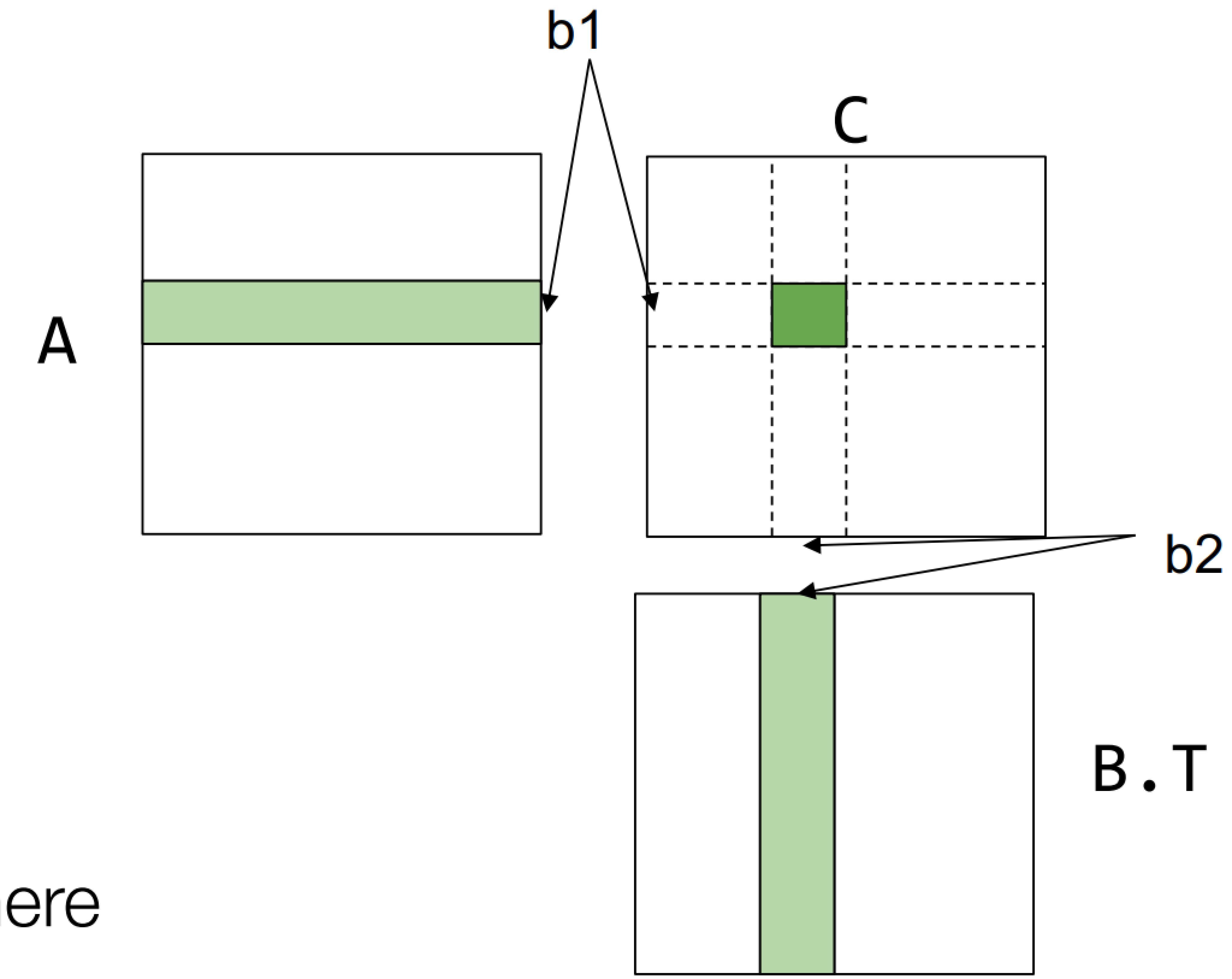
3nd layer:

Cache-aware tiling

- We can further tile the array [b1][n] or [b2][n] using at the L1-> register level
- What's the required condition?

```
dram float A[n/b1][b1][n];
dram float B[n/b2][b2][n];
dram float C[n/b1][n/b2][b1][b2];
for (int i = 0; i < n/b1; ++i) {
   l1cache float a[b1][n] = A[i];
   for (int j = 0; j < n/b2; ++j) {
      l1cache b[b2][n] = B[j];

      C[i][j] = dot(a, b.T);
   }
}
```

Sub-procedure, can apply register tiling here



Cache-aware tiling

```
dram float A[n/b1][b1][n];
dram float B[n/b2][b2][n];
dram float C[n/b1][n/b2][b1][b2];
for (int i = 0; i < n/b1; ++i) {
   l1cache float a[b1][n] = A[i];
   for (int j = 0; j < n/b2; ++j) {
      l1cache b[b2][n] = B[j];

      C[i][j] = dot(a, b.T);
   }
}
```

Later we apply register tiling here

Data movement path:

1.Dram

2.Dram -> l1cache (cache tiling)

3.L1cache -> register (reg tiling)

Cache-aware tiling

```
dram float A[n/b1][b1][n];
dram float B[n/b2][b2][n];
dram float C[n/b1][n/b2][b1][b2];
for (int i = 0; i < n/b1; ++i) {
   l1cache float a[b1][n] = A[i];
   for (int j = 0; j < n/b2; ++j) {
      l1cache b[b2][n] = B[j];

      C[i][j] = dot(a, b.T);
   }
}
```

A’s dram -> l1 cost:

$n / b1 * n * b1 = n^2$

B’s dram -> l1 time cost:

$n / b1 * n / b2 * b2 * n = n^3 / b1$

Vs. previous untiled version?

s.t.

- $b1 * n + b2 * n <$ L1 cache size

Putting Things Together

We set v3 = 1 (we know it does not matter)

```
dram float A[n/b1][b1/v1][n][v1];
dram float B[n/b2][b2/v2][n][v2];

for (int i = 0; i < n/b1; ++i) {
  l1cache float a[b1/v1][n][v1] = A[i];
  for (int j = 0; j < n/b2; ++j) {
   l1cache b[b2/v2][n][v2] = B[j];
   for (int x = 0; x < b1/v1; ++x)
    for (int y = 0; y < b2/v2; ++y) {
     register float c[v1][v2] = 0;
     for (int k = 0; k < n; ++k) {
        register float ar[v1] = a[x][k][:];
        register float br[v2] = b[y][k][:];
            C += dot(ar, br.T)
     }
    }
  }
}
```

Outside: cache tiling
Inside: register tiling

- Cache tiling using b1 and b2
- DRAM → L1 cache reads here

- Register tiling using v1 and v2
- L1→ register cache reads here

Putting Things Together

```
dram float A[n/b1][b1/v1][n][v1];
dram float B[n/b2][b2/v2][n][v2];

for (int i = 0; i < n/b1; ++i) {
  l1cache float a[b1/v1][n][v1] = A[i];
  for (int j = 0; j < n/b2; ++j) {
   l1cache b[b2/v2][n][v2] = B[j];
   for (int x = 0; x < b1/v1; ++x)
    for (int y = 0; y < b2/v2; ++y) {
     register float c[v1][v2] = 0;
     for (int k = 0; k < n; ++k) {
        register float ar[v1] = a[x][k][:];
        register float br[v2] = b[y][k][:];
                C += dot(ar, br.T)
     }
    }
  }
}
```

Outside: cache tiling
Inside: register tiling

Cost: dram -> l1

- n/b1*b1/v1*n*v1 = n^2
- n/b1*n/b2*b2/v2*n*v2=n^2/b1

Cost: l1 -> register:

- n / b1 * n / b2 * b1 / v1 * b2 / v2 * n * v1 = n^3 / v2
- n^3 / v1

In practice

- On CPUs: We have disk -> dram -> L2 -> L1 -> Register
- How to choose v1, v2, b1, b2, c1, c2?
- While we are reading from dram -> L2, can we concurrently read:
    - L2 -> L1
    - L1 -> register
- S.t. sizes of L2, L1, registers

Why tiling works: **reuse** loading

```
float A[n][n];
float B[n][n];
float C[n][n];

C[i][j] = sum(A[i][k] * B[j][k], axis=k)
```

Access of A is independent of the dimension of j

Tile the j dimension by v1 enables resue of A for v1 times

Homework Candidate?

- Q: How to tile?

```
for n in range(0, N):
 for co in range(0, CO):
  for h in range(0, H):
   for w in range(0, W):
    for ci in range(0, CI):
     for kh in range(0, KH):
      for kw in range(0, KW):
       C[n,co,h,w] += A[n,co,h+kh,w+kw] x B[kh,kw,co,ci]
```

Simple spatial loops.

Stencil computation loops.

Reduction loop.

Reduction loops. But usually too small (<= 5) for parallelization.

Where we are

- How to make operators fast in general?
  - ✅ Vectorize
  - ✅ Data layout
  - ✅ Parallelization (at the operator level)
- ✅ Matmul-specific optimization
- ❓ GPUs and accelerators
  - High-level Idea
  - The accelerator market

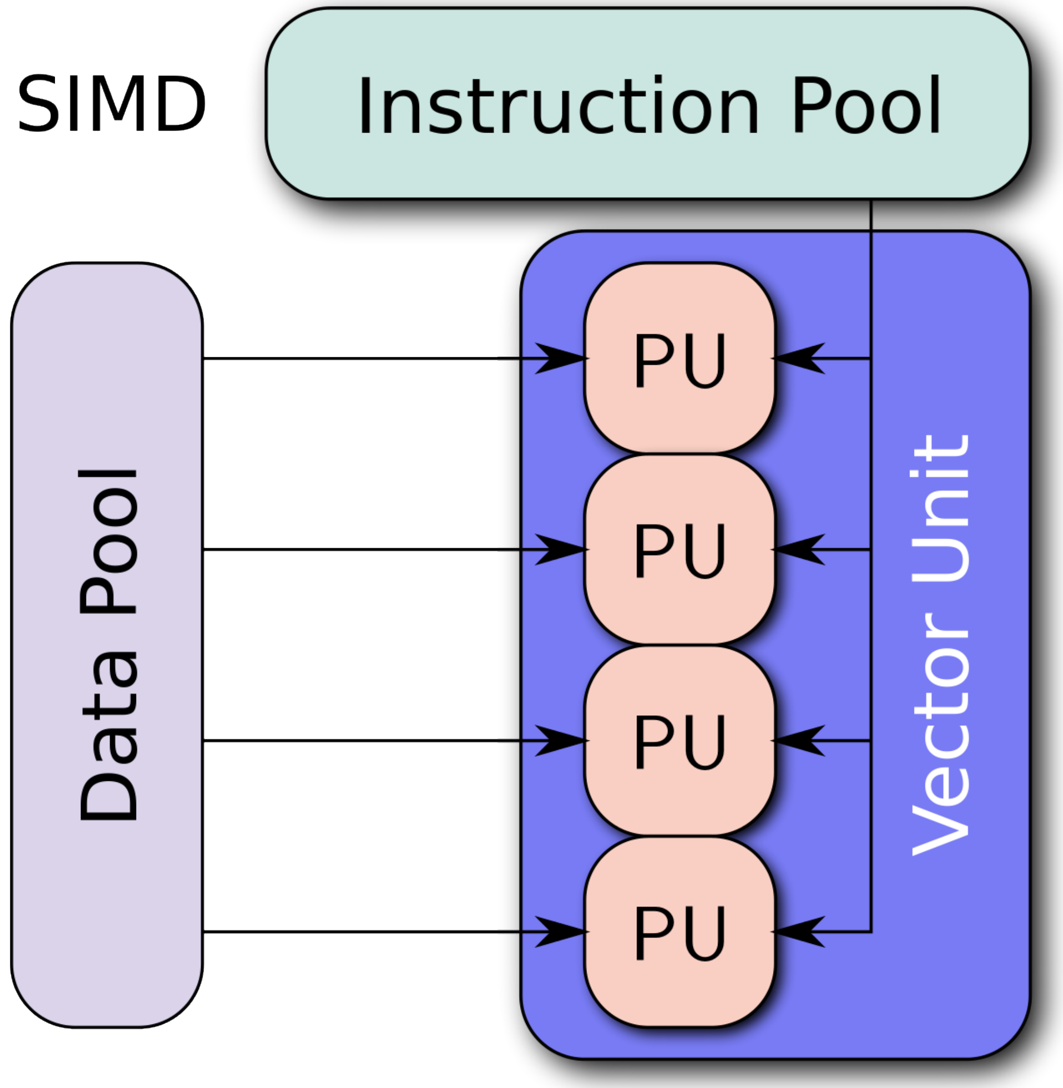
Recap CPU parallelization

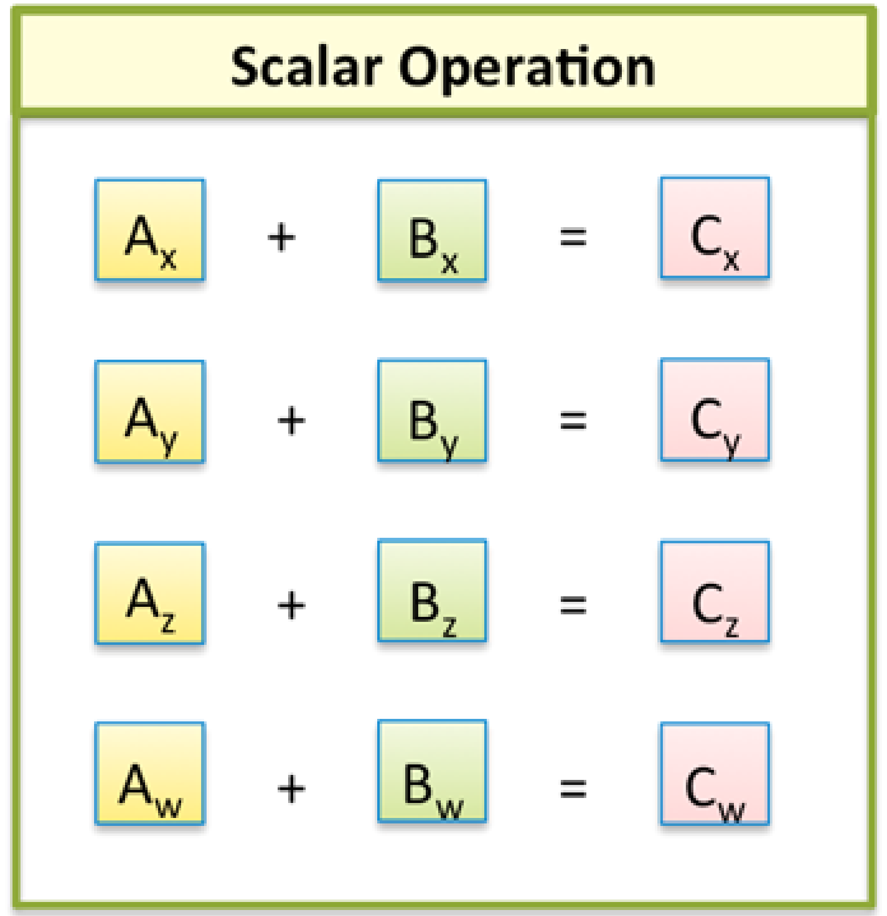
- We can parallelize this loop using CPU threads
- == using many concurrent cores

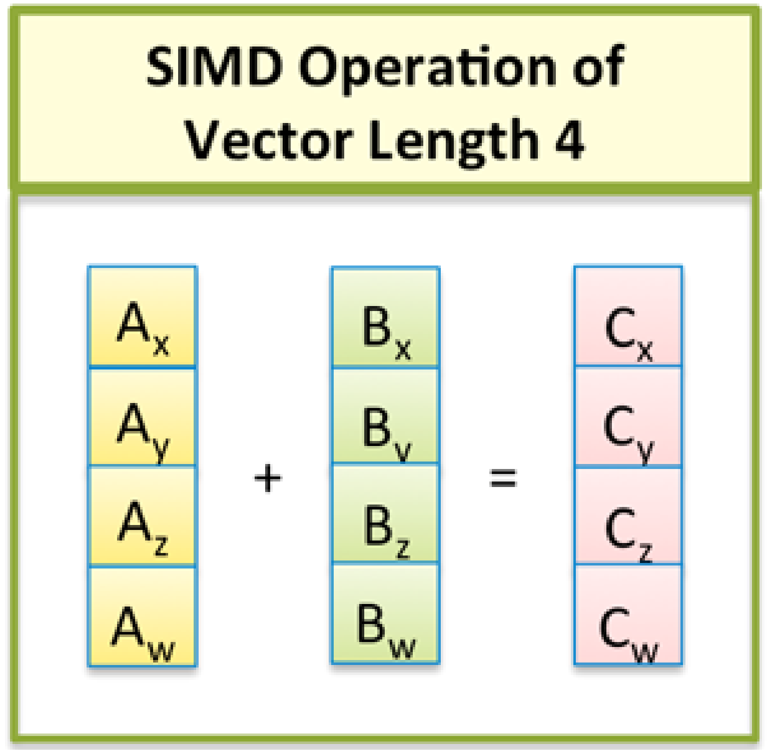
```
   #pragma omp parallel for
   for (int i = 0; i < 64; ++i) {
         float4 a = load_float4(A + i*4);
         float4 b = load_float4(B + i*4);
         float4 c = add_float4(a, b);
         store_float4(C * 4, c);
}
```

Vectorized &
parallelized

Single-Instruction Multiple-Data

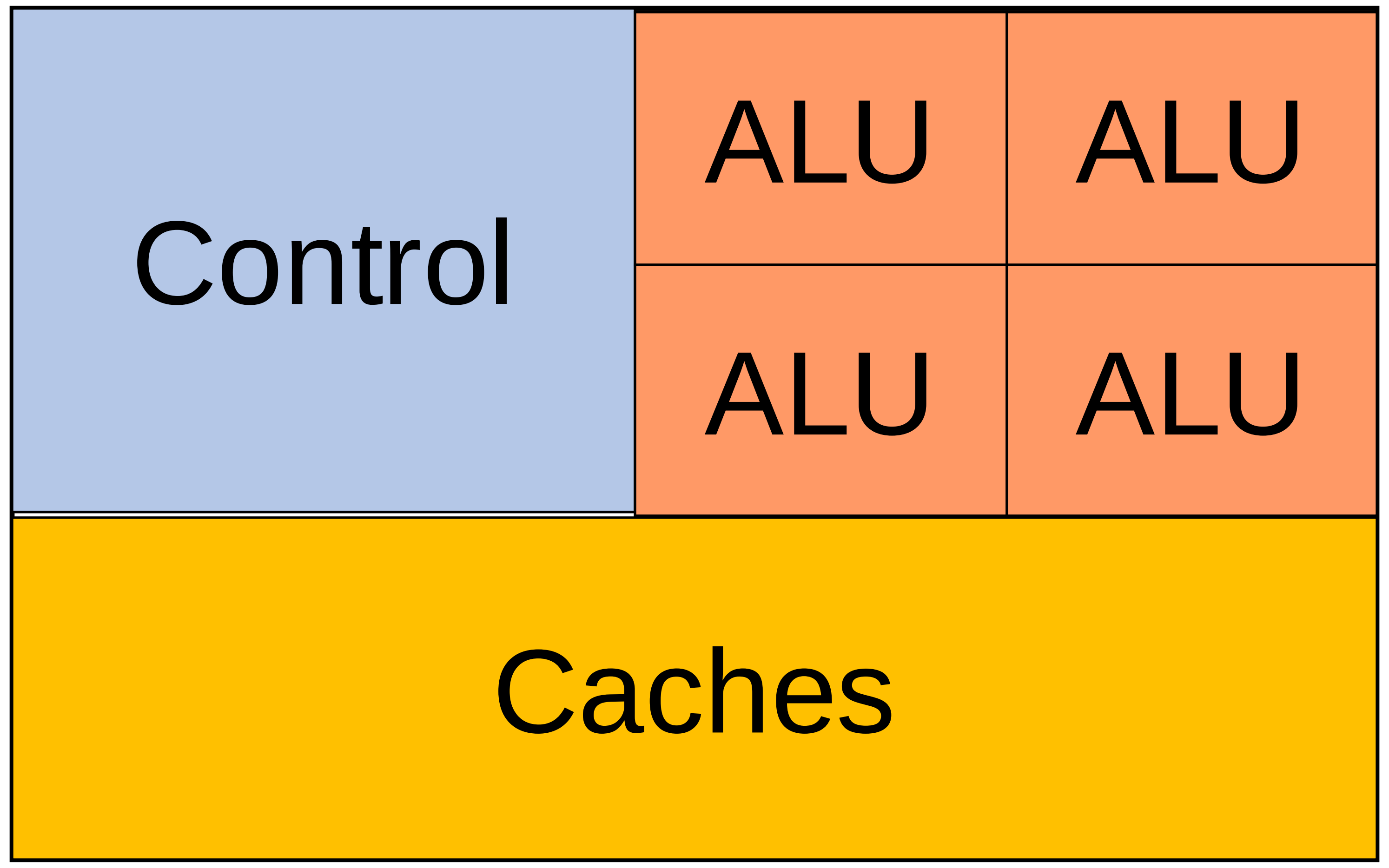




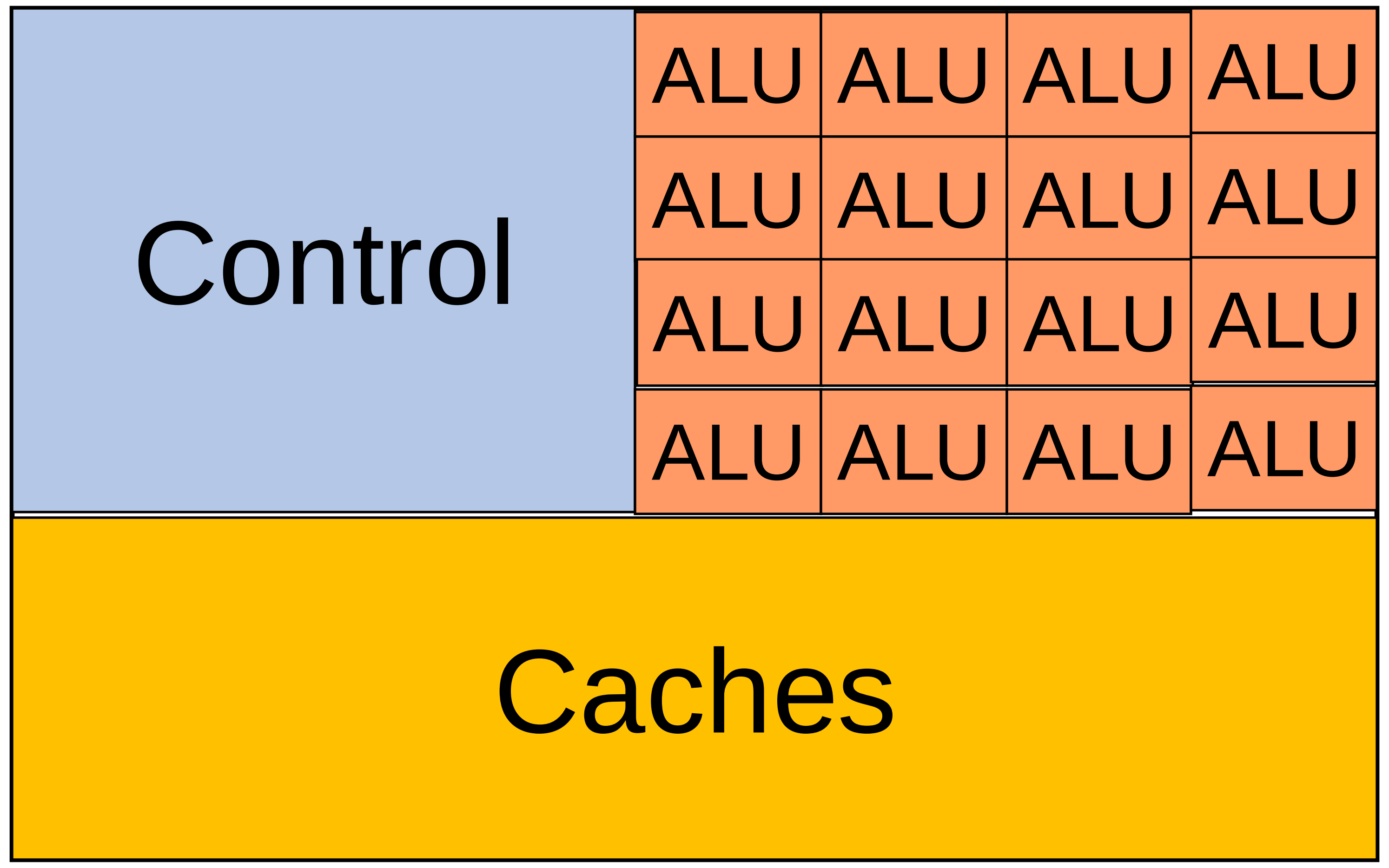


Intel® Architecture currently has SIMD operations of vector length 4, 8, 16

Chip Design Trajectory: SIMD



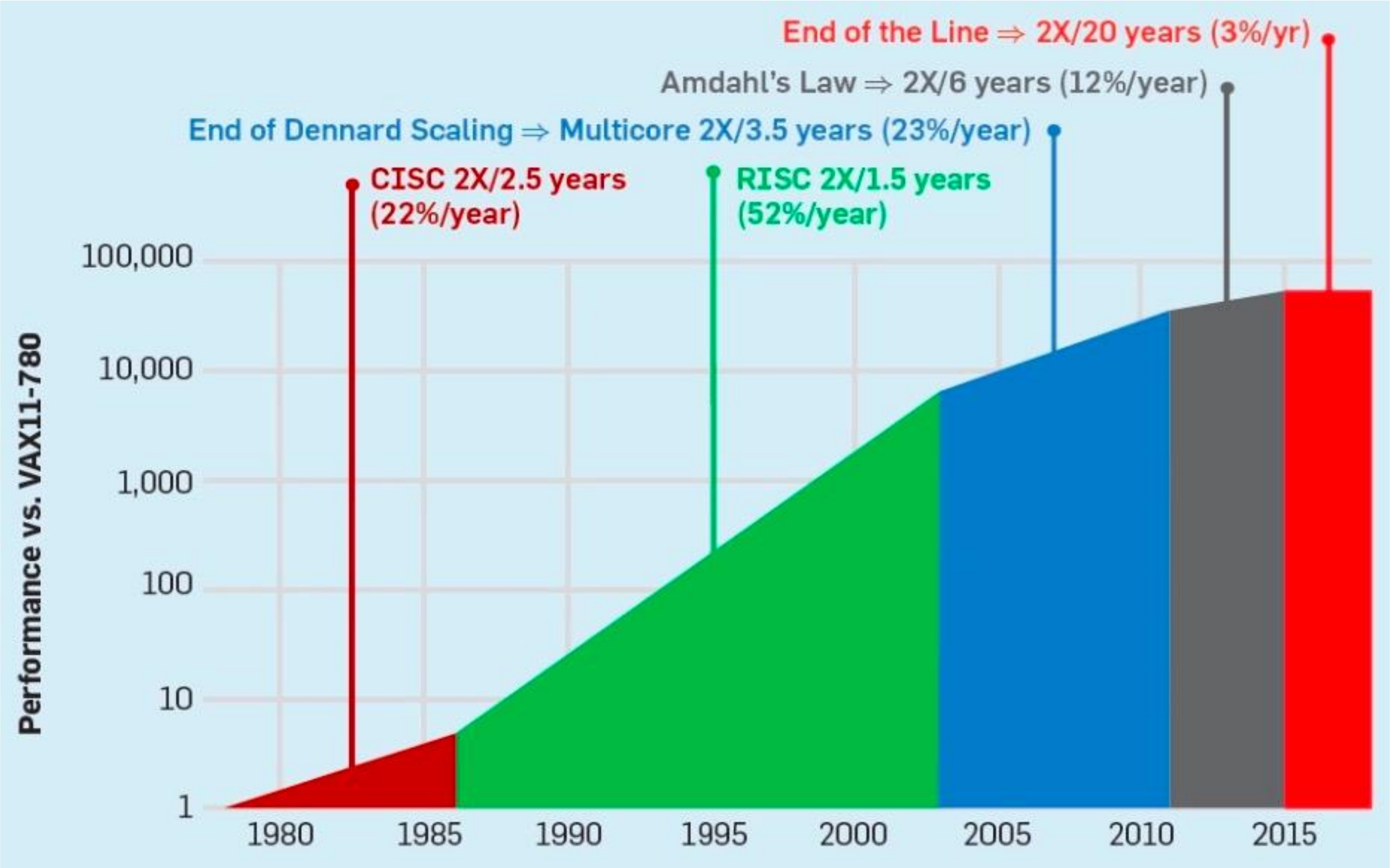
If we're able to reduce size of ALU
(transistors) while keeping its power



Chip Industry: 70nm -> 60nm -> 50nm -> … ->?

- Problem: this is not substantiable; there are also power/heat issues when you put more ALUs in a limited area (s.t. physics limitations)

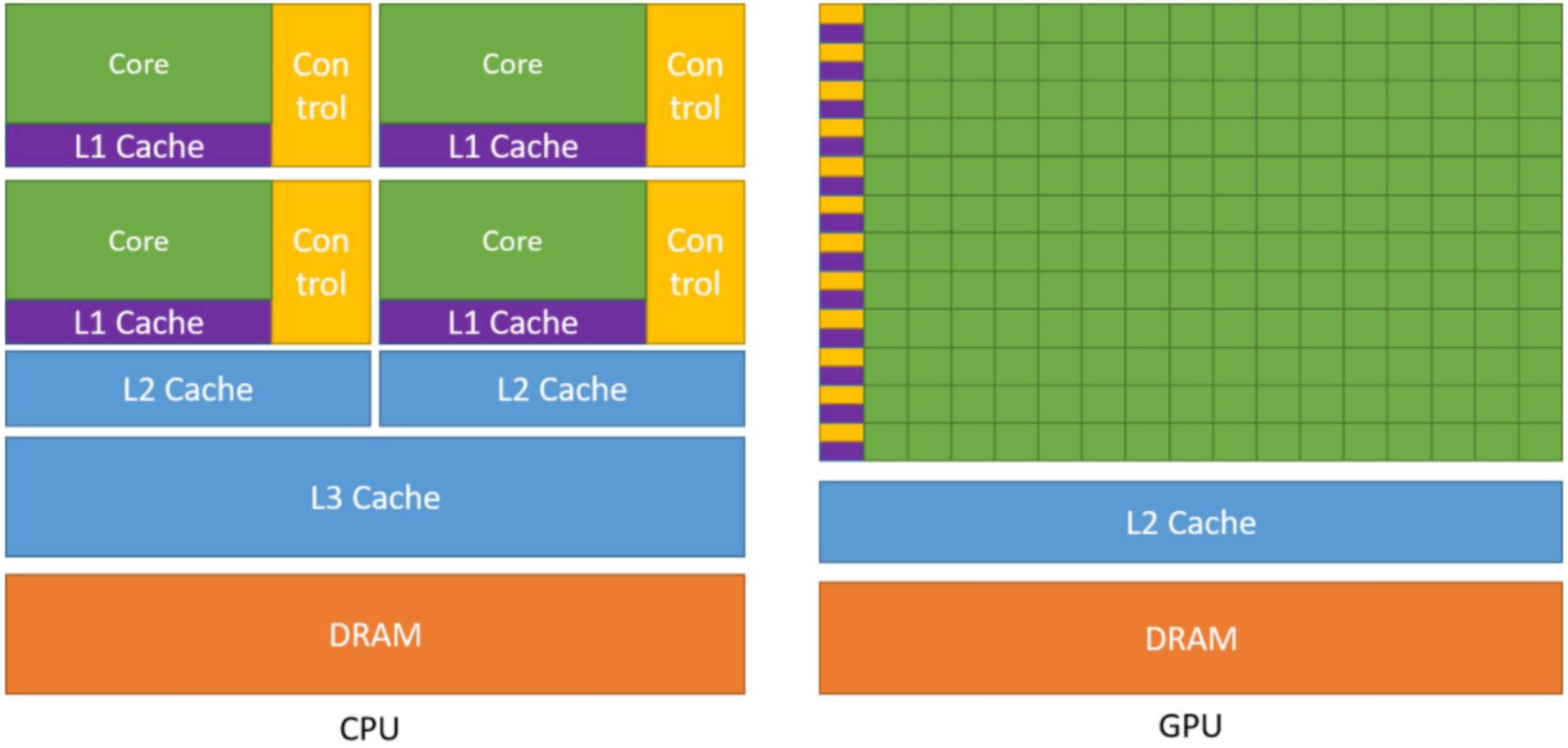
Chip Industry: Moore's Law Comes to an End



Option 1: Go to the quantum world

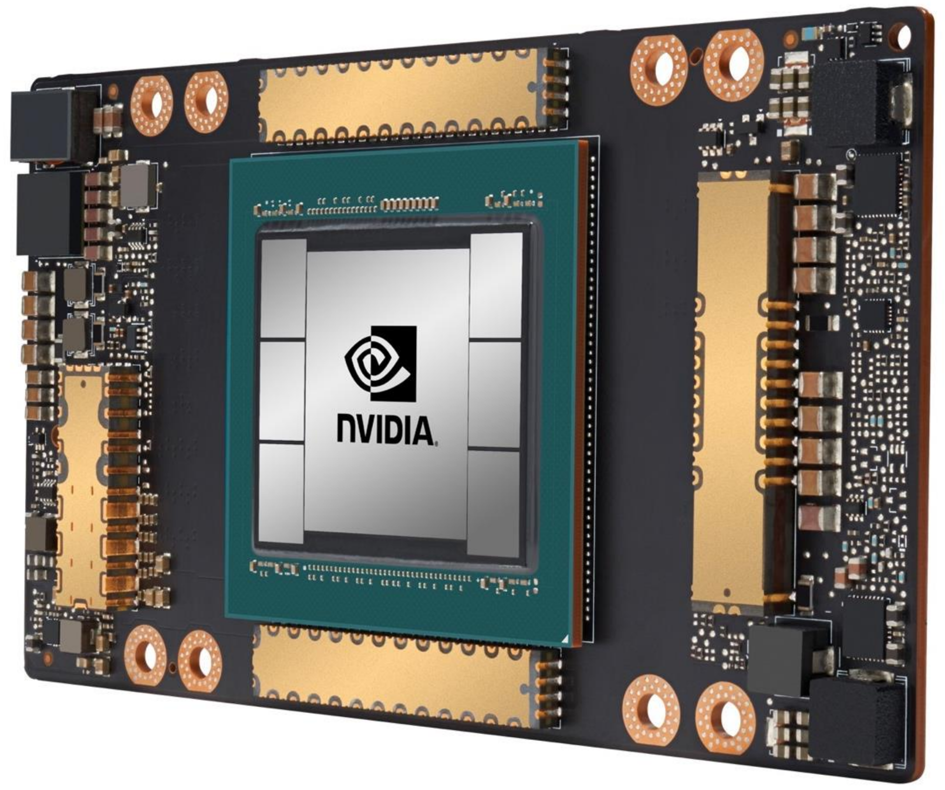
Option 2: Specialized hardware

Idea: How about we use a lot of weak/specialized cores



Hardware Accelerators: GPUs

- **Graphics Processing Unit (GPU)**: Tailored for matrix/tensor ops
- Basic idea: Use tons of ALUs (but weak and more specialized); massive data parallelism (SIMD on steroids);
- Popularized by NVIDIA in early 2000s for video games, graphics, and multimedia; now ubiquitous in DL
- CUDA released in 2007; later wrapper APIs on top: CuDNN, CuSparse, CuDF (RapidsAI), NCCL, etc.



Other Hardware Accelerators

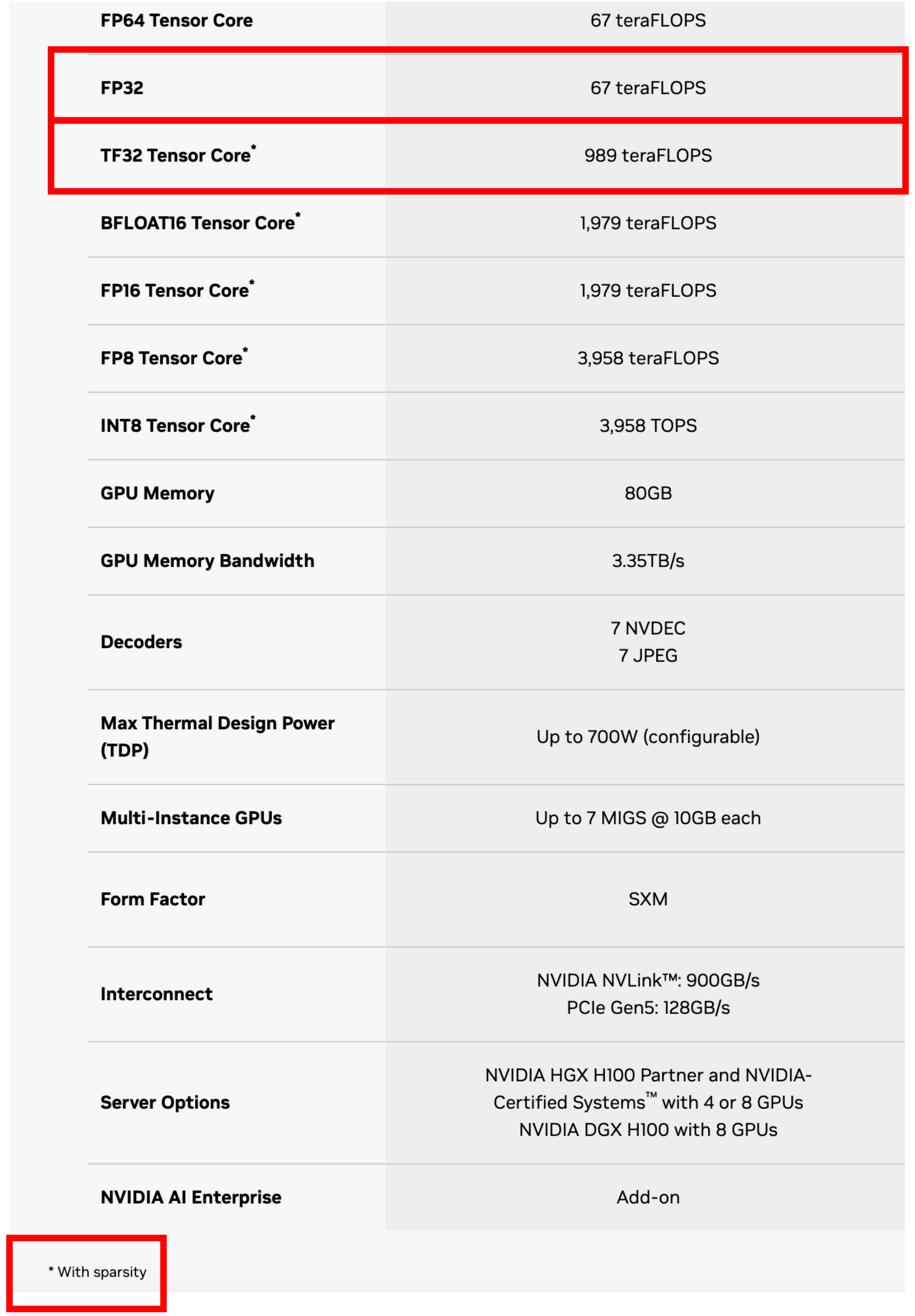
- E.g.
  - Tensor Processing Unit (TPU)
  - An “application-specific integrated circuit” (ASIC) created by Google in mid 2010s; used for AlphaGo
- E.g.
  - B200 (projected release 2025): fp4 / fp8 Tensorcore
- E.g.
  - M3 max: mixing tensorcore and normal core

What Does It Mean by “Specialized” In accelerator world

In General:

- Functionality-specialized:
  - Can only compute certain computations: matmul, w/ sparsity
  - Mixing specialized cores with versatile cores
- Reduce precision
  - Floating point operations: fp32, fp16, fp8, int8, int4, …
- Tune the distribution of different components for specific workloads
  - SRAM, cache, registers, etc.

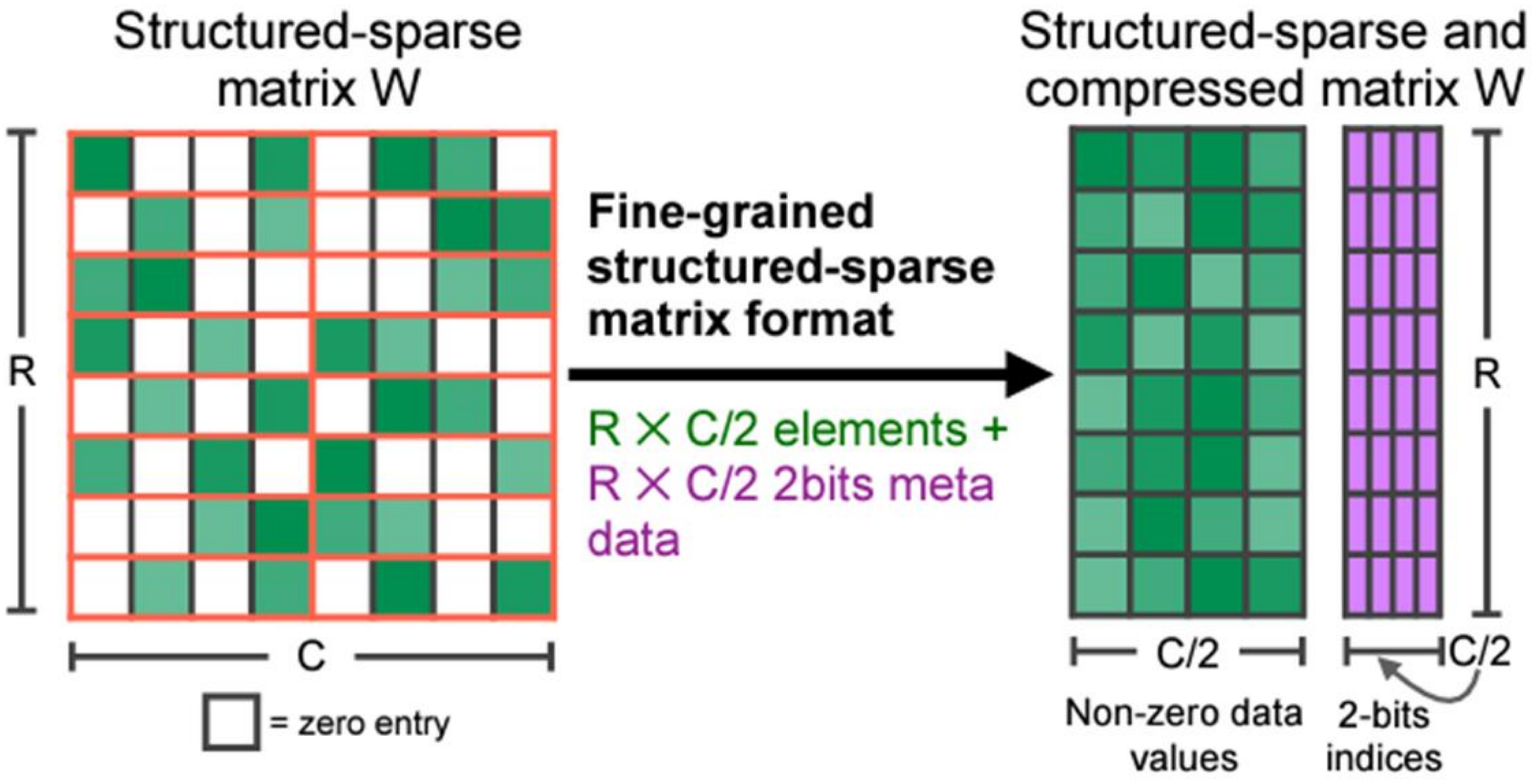
Case Study 1: Nvidia GPU Specification



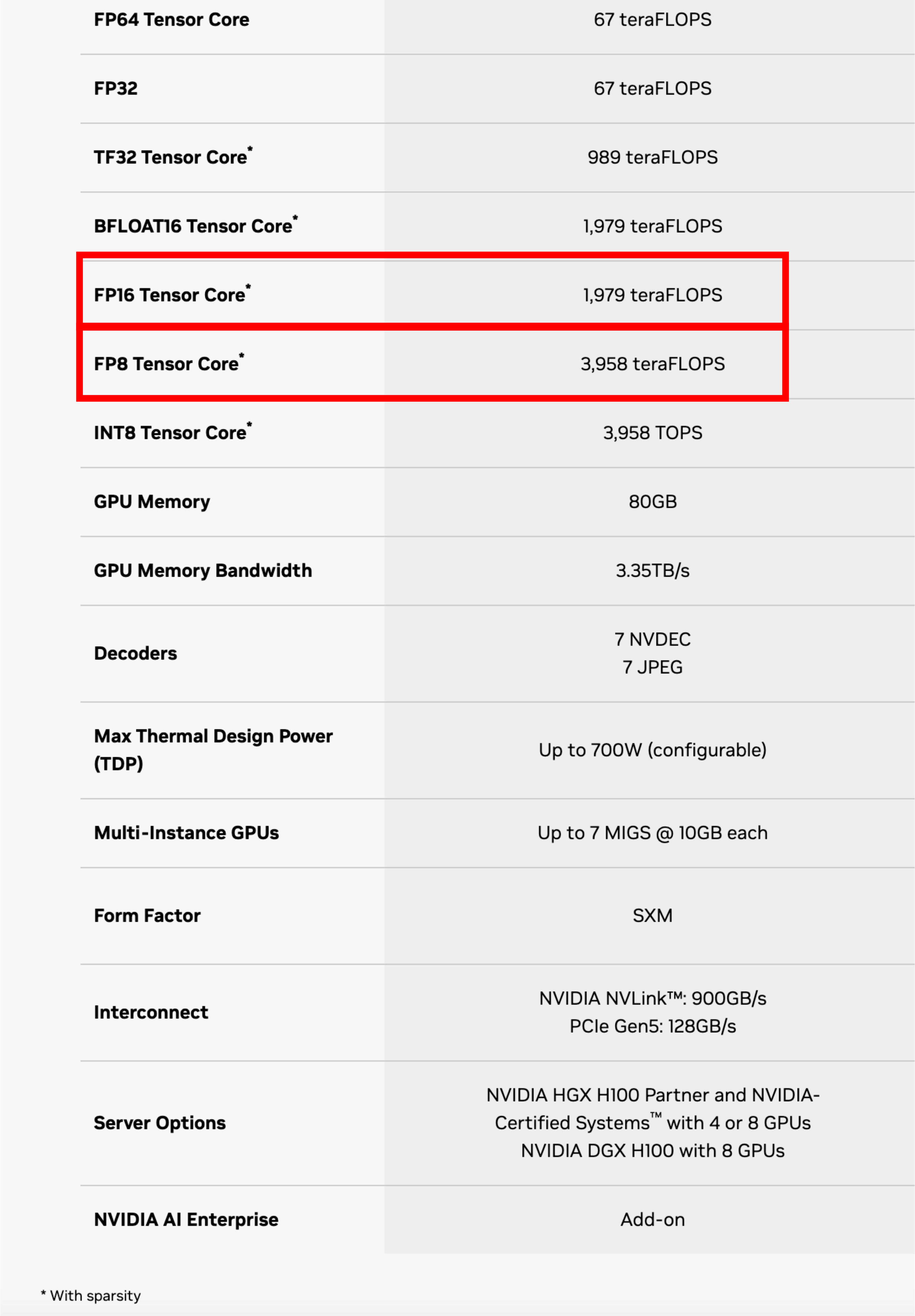
FP64 Tensor Core	67 teraFLOPS
FP32	67 teraFLOPS
TF32 Tensor Core*	989 teraFLOPS
BFLOAT16 Tensor Core*	1,979 teraFLOPS
FP16 Tensor Core*	1,979 teraFLOPS
FP8 Tensor Core*	3,958 teraFLOPS
INT8 Tensor Core*	3,958 TOPS
GPU Memory	80GB
GPU Memory Bandwidth	3.35TB/s
Decoders	7 NVDEC 7 JPEG
Max Thermal Design Power (TDP)	Up to 700W (configurable)
Multi-Instance GPUs	Up to 7 MIGS @ 10GB each
Form Factor	SXM
Interconnect	NVIDIA NVLink™: 900GB/s PCIe Gen5: 128GB/s
Server Options	NVIDIA HGX H100 Partner and NVIDIA-Certified Systems™ with 4 or 8 GPUs NVIDIA DGX H100 with 8 GPUs
NVIDIA AI Enterprise	Add-on

* With sparsity

Case Study 1: Nvidia GPU Specification

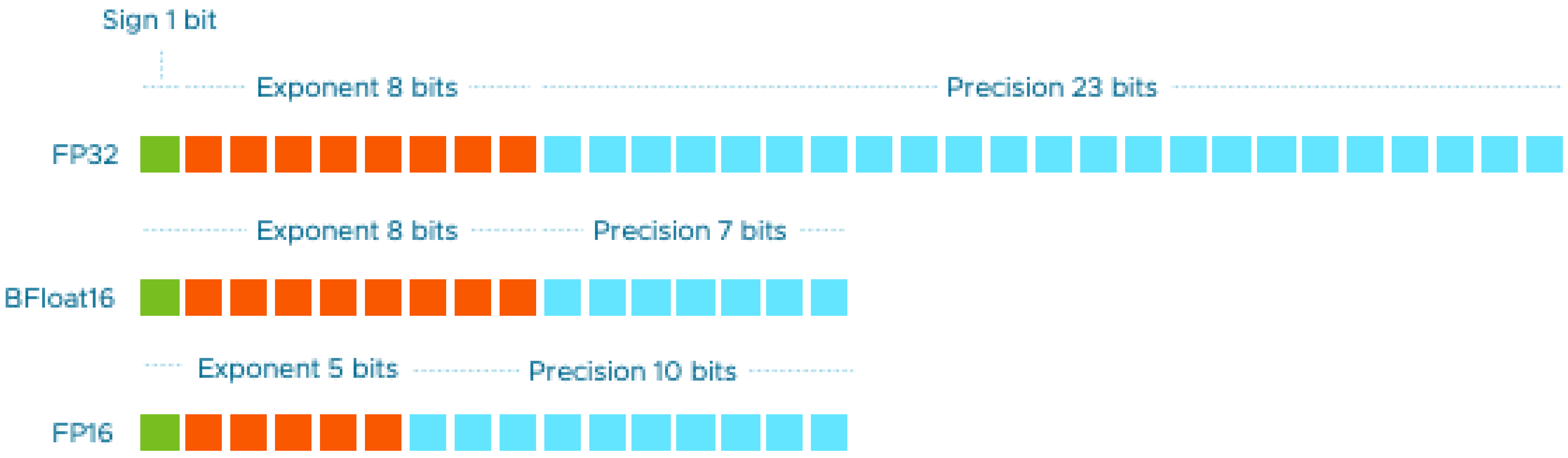

Structured-sparse matrix W
R
C
□ = zero entry
Fine-grained structured-sparse matrix format
R × C/2 elements +
R × C/2 2bits meta data
Structured-sparse and compressed matrix W
R
C/2
C/2
Non-zero data values
2-bits indices

Case Study 1: Nvidia GPU Specification



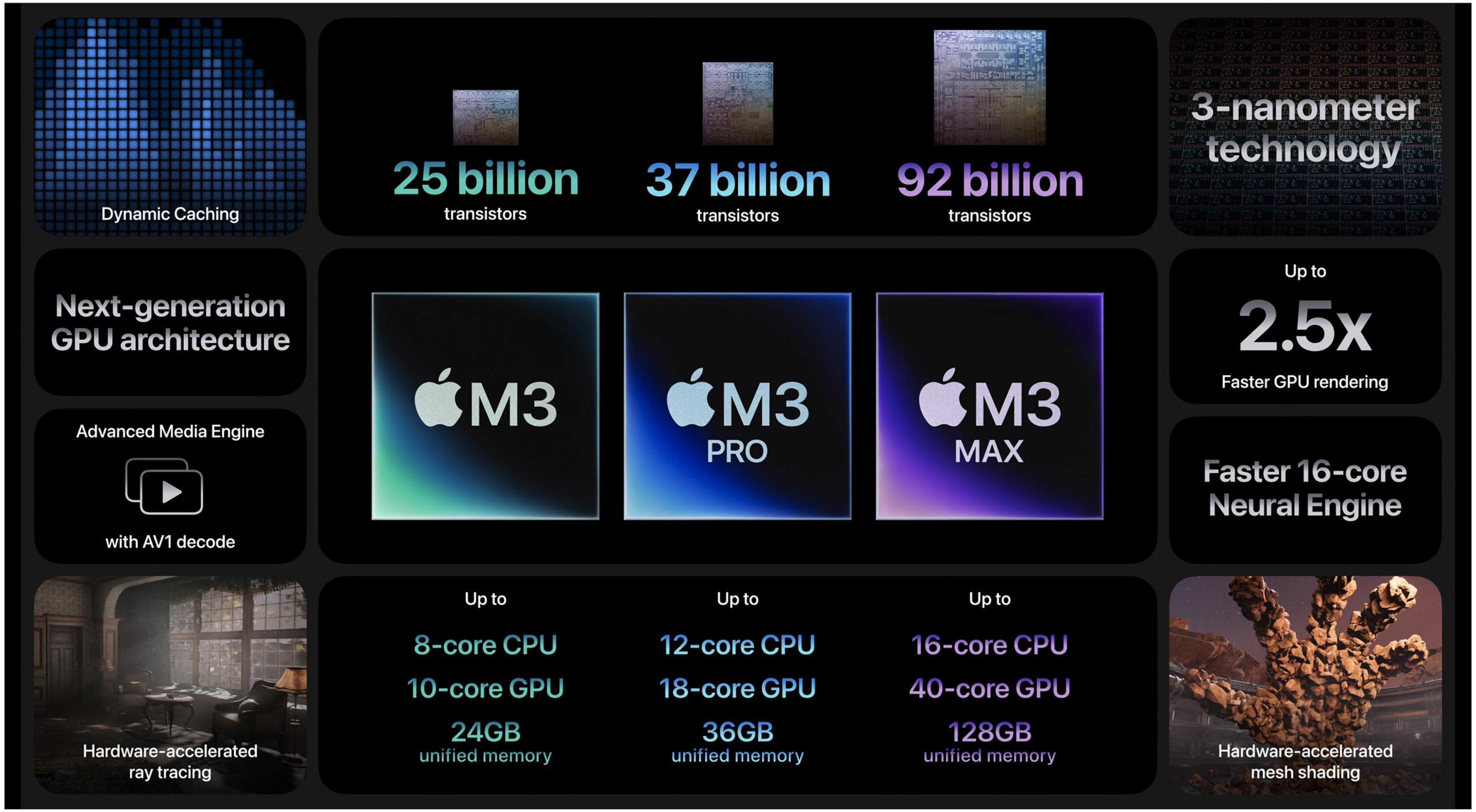
FP64 Tensor Core	67 teraFLOPS
FP32	67 teraFLOPS
TF32 Tensor Core*	989 teraFLOPS
BFLOAT16 Tensor Core*	1,979 teraFLOPS
FP16 Tensor Core*	1,979 teraFLOPS
FP8 Tensor Core*	3,958 teraFLOPS
INT8 Tensor Core*	3,958 TOPS
GPU Memory	80GB
GPU Memory Bandwidth	3.35TB/s
Decoders	7 NVDEC 7 JPEG
Max Thermal Design Power (TDP)	Up to 700W (configurable)
Multi-Instance GPUs	Up to 7 MIGS @ 10GB each
Form Factor	SXM
Interconnect	NVIDIA NVLink™: 900GB/s PCIe Gen5: 128GB/s
Server Options	NVIDIA HGX H100 Partner and NVIDIA-Certified Systems™ with 4 or 8 GPUs NVIDIA DGX H100 with 8 GPUs
NVIDIA AI Enterprise	Add-on

* With sparsity



Question: why this could work in ML programs?

Case Study 2: Apple Silicon



Dynamic Caching

25 billion transistors

37 billion transistors

92 billion transistors

3-nanometer technology

Next-generation GPU architecture

Advanced Media Engine with AV1 decode

M3

M3 PRO

M3 MAX

Up to 2.5x Faster GPU rendering

Faster 16-core Neural Engine

Hardware-accelerated ray tracing

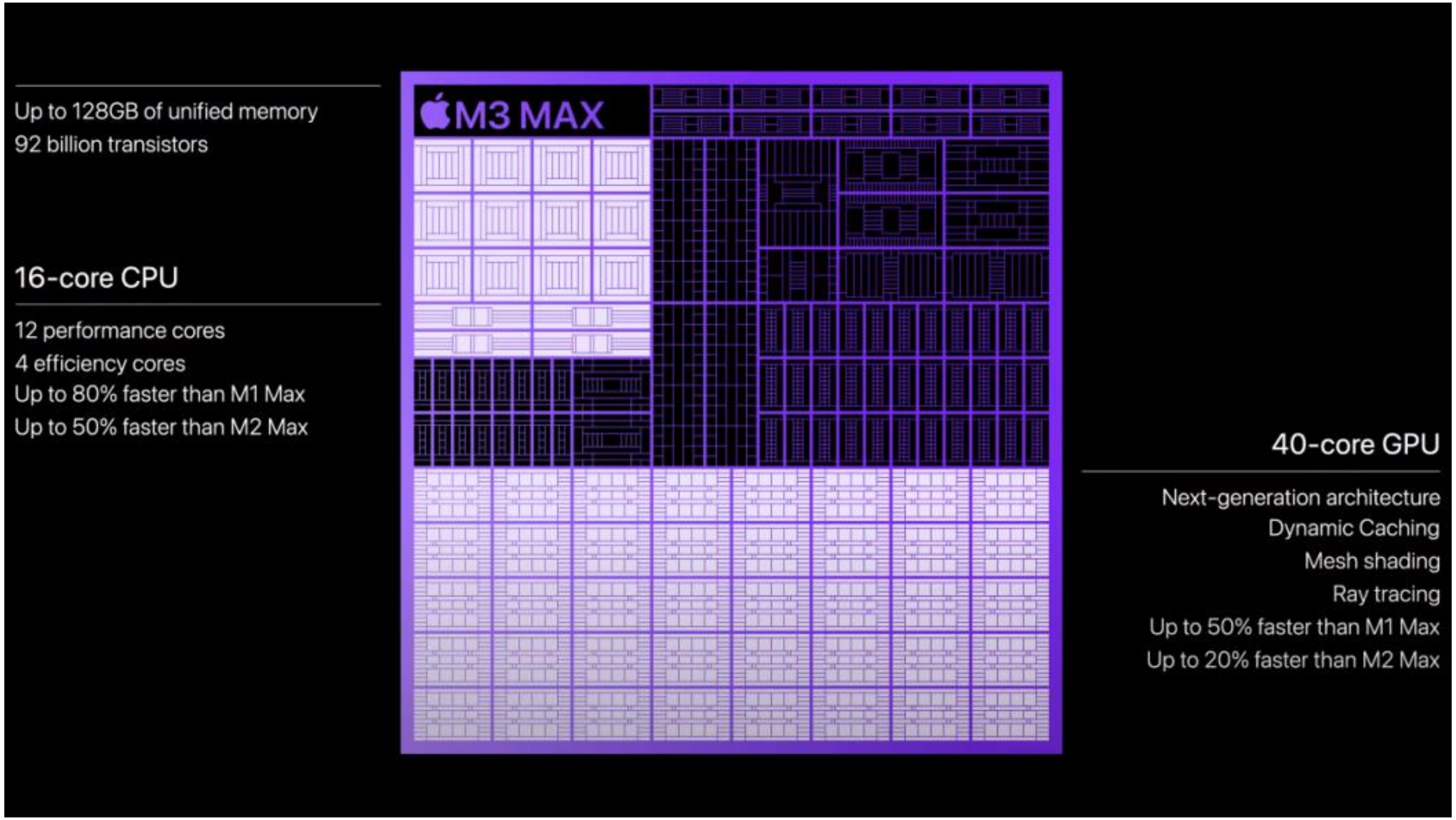
Up to
8-core CPU
10-core GPU
24GB unified memory

Up to
12-core CPU
18-core GPU
36GB unified memory

Up to
16-core CPU
40-core GPU
128GB unified memory

Hardware-accelerated mesh shading

Case Study 2: Apple Silicon Revealed



Up to 128GB of unified memory
92 billion transistors

16-core CPU

12 performance cores
4 efficiency cores
Up to 80% faster than M1 Max
Up to 50% faster than M2 Max

40-core GPU

Next-generation architecture
Dynamic Caching
Mesh shading
Ray tracing
Up to 50% faster than M1 Max
Up to 20% faster than M2 Max

Case Study 3: Leading Chip Startups

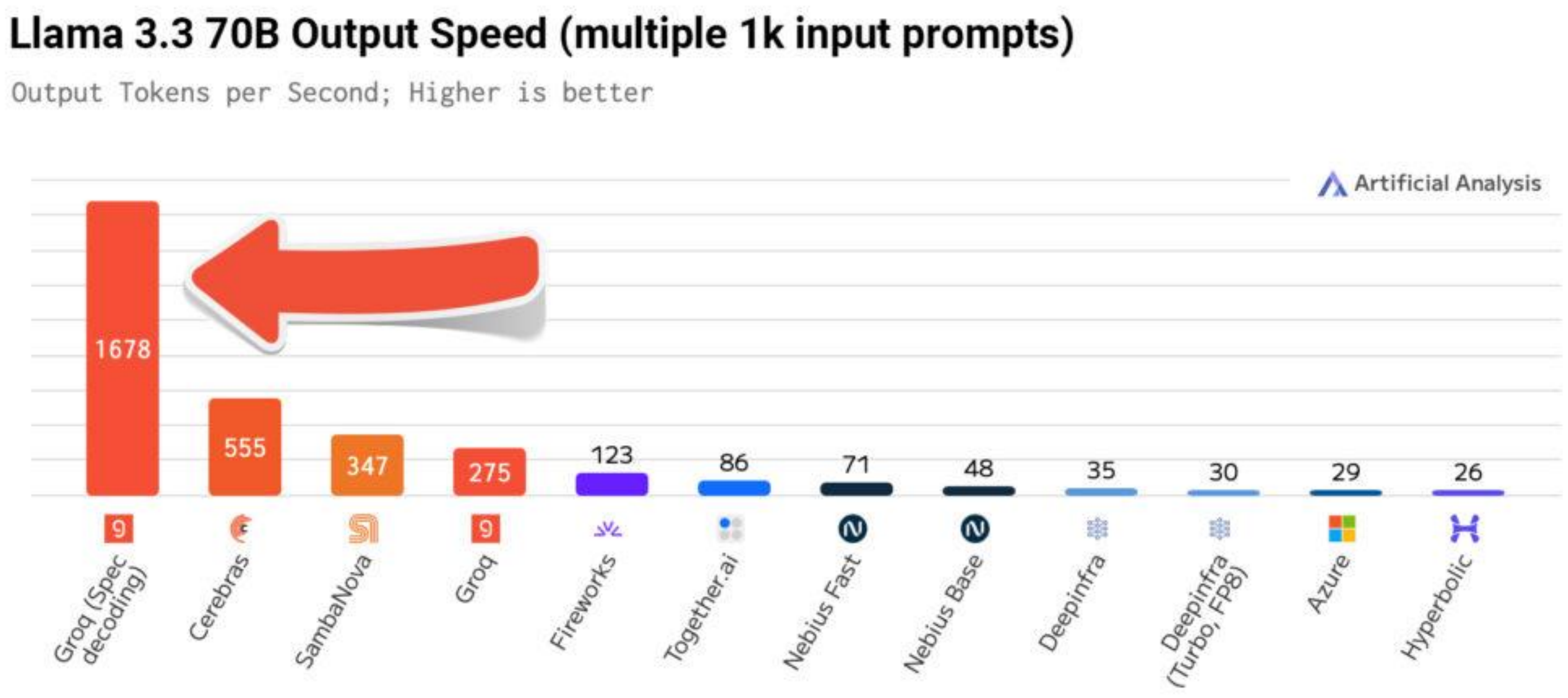
groq

cerebras

SambaNova®
SYSTEMS

Case Study 3: Groq

Question: How did Groq achieve that?



Llama 3.3 70B Output Speed (multiple 1k input prompts)

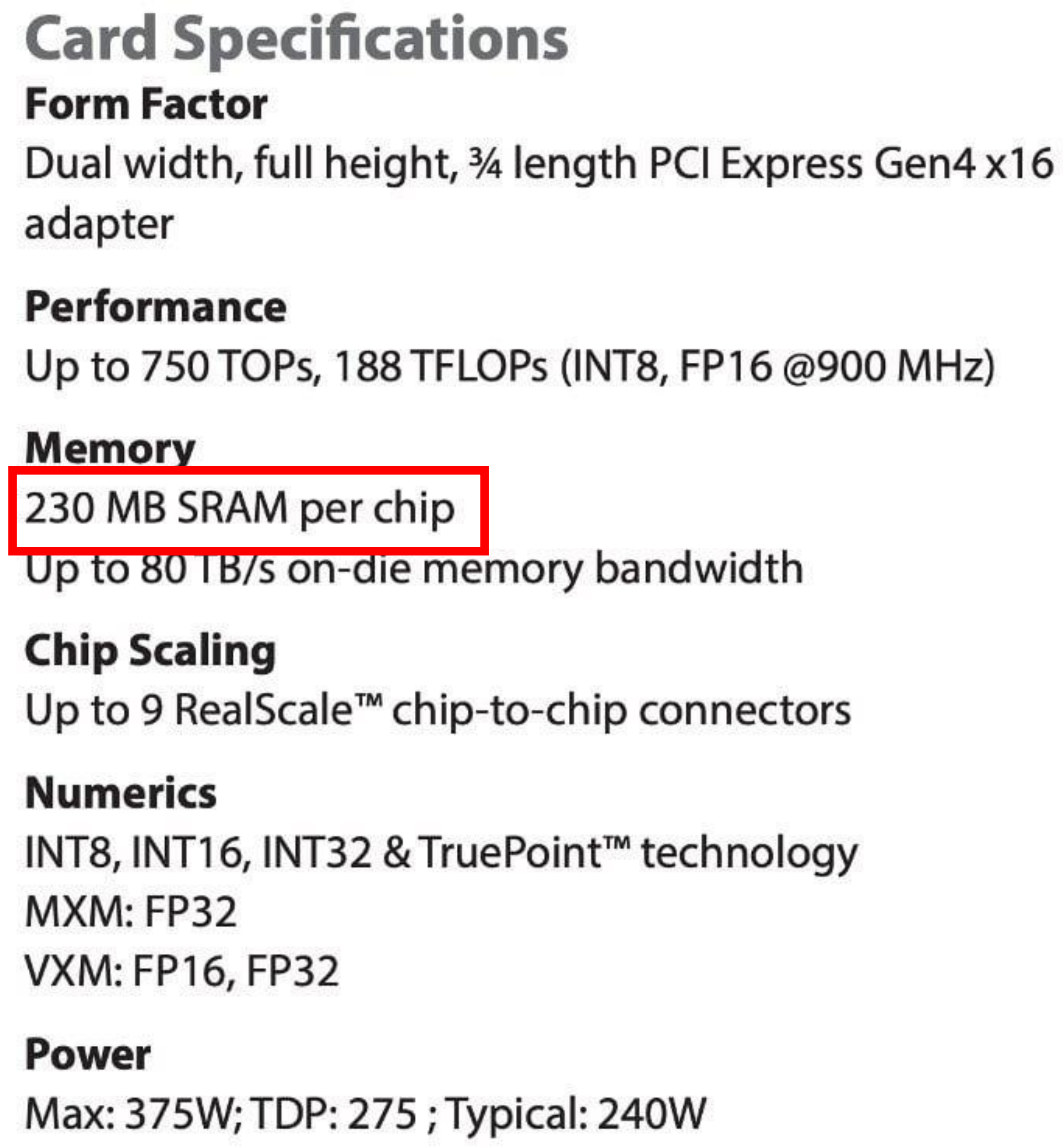
Output Tokens per Second; Higher is better

More specialized hardware

Nvidia GPUs

Case Study 3: Groq

GroqCard™



Card Specifications

Form Factor

Dual width, full height, ¾ length PCI Express Gen4 x16 adapter

Performance

Up to 750 TOPs, 188 TFLOPs (INT8, FP16 @900 MHz)

Memory

230 MB SRAM per chip

Up to 80 TB/s on-die memory bandwidth

Chip Scaling

Up to 9 RealScale™ chip-to-chip connectors

Numerics

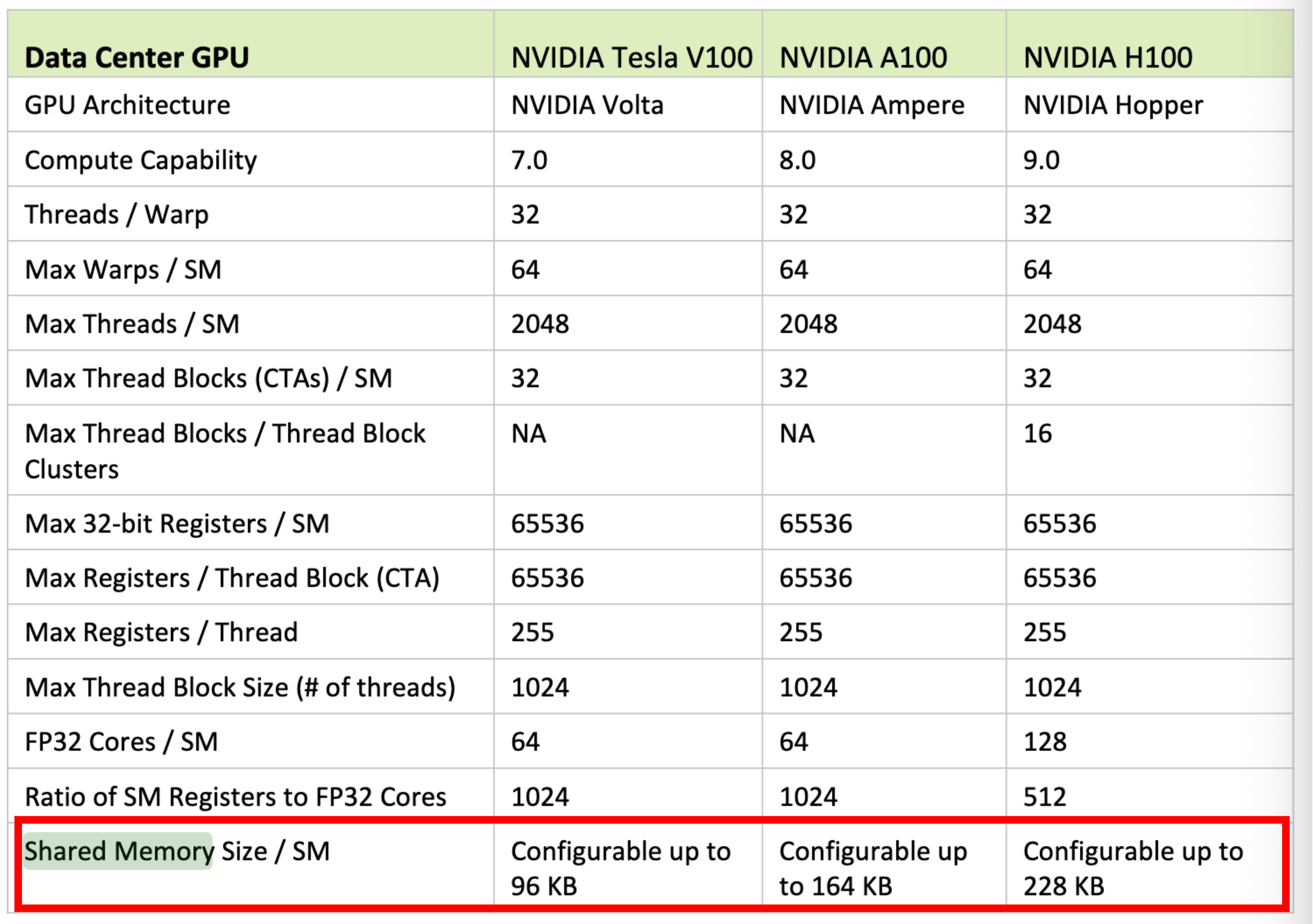
INT8, INT16, INT32 & TruePoint™ technology

MXM: FP32

VXM: FP16, FP32

Power

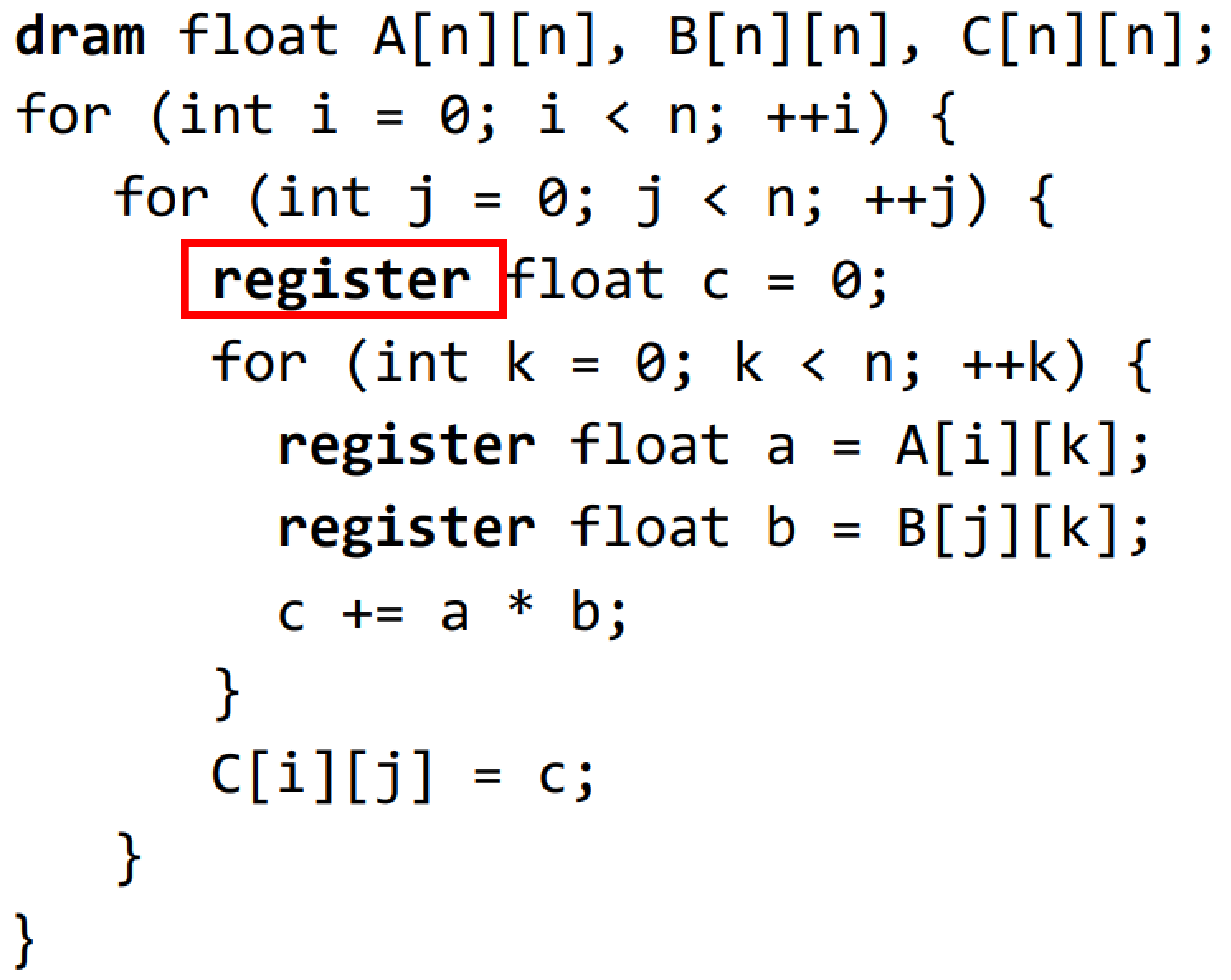
Max: 375W; TDP: 275 ; Typical: 240W



Data Center GPU	NVIDIA Tesla V100	NVIDIA A100	NVIDIA H100
GPU Architecture	NVIDIA Volta	NVIDIA Ampere	NVIDIA Hopper
Compute Capability	7.0	8.0	9.0
Threads / Warp	32	32	32
Max Warps / SM	64	64	64
Max Threads / SM	2048	2048	2048
Max Thread Blocks (CTAs) / SM	32	32	32
Max Thread Blocks / Thread Block Clusters	NA	NA	16
Max 32-bit Registers / SM	65536	65536	65536
Max Registers / Thread Block (CTA)	65536	65536	65536
Max Registers / Thread	255	255	255
Max Thread Block Size (# of threads)	1024	1024	1024
FP32 Cores / SM	64	64	128
Ratio of SM Registers to FP32 Cores	1024	1024	512
Shared Memory Size / SM	Configurable up to 96 KB	Configurable up to 164 KB	Configurable up to 228 KB

Case Study 3: Groq

- Recall



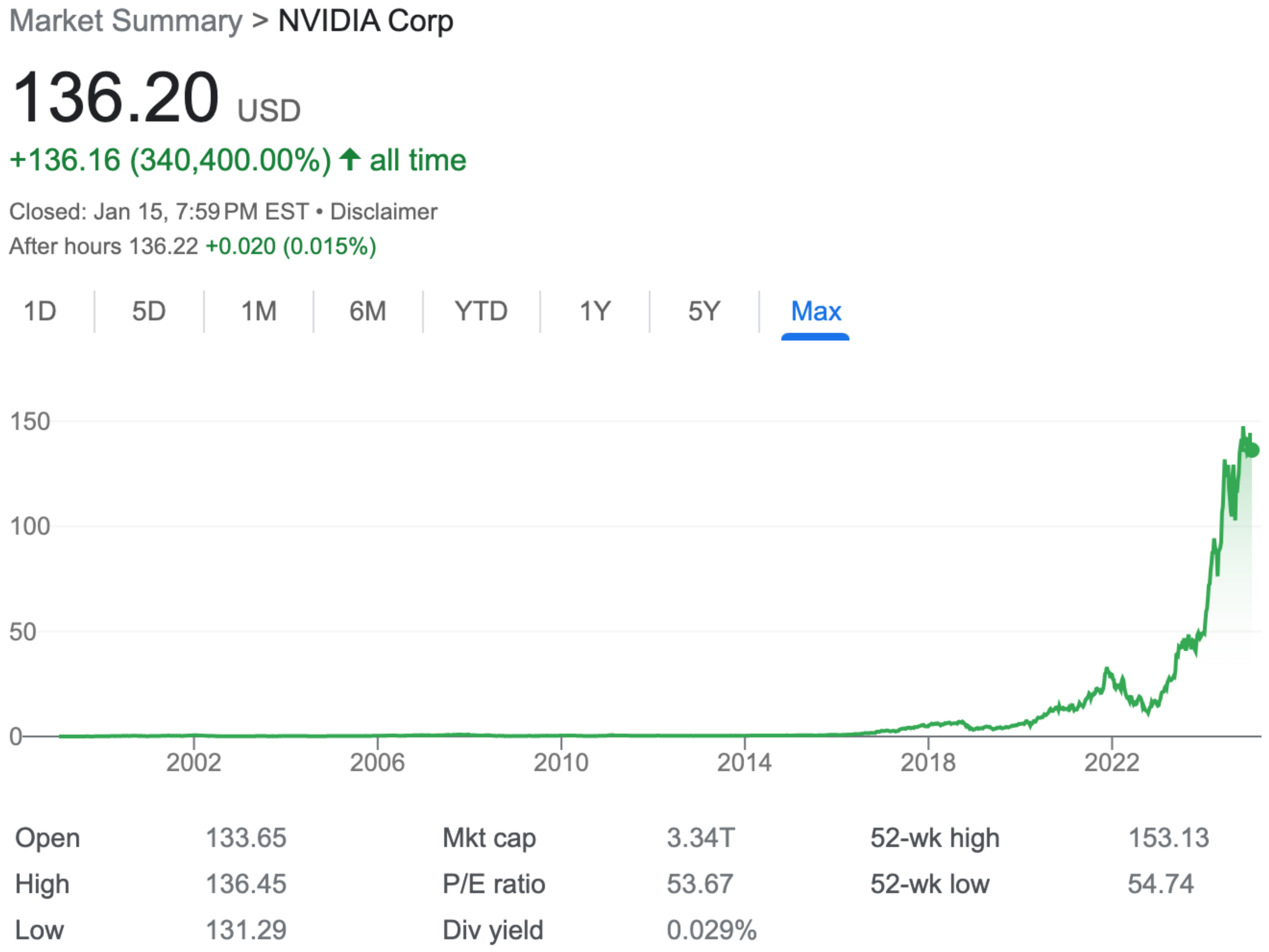
```
dram float A[n][n], B[n][n], C[n][n];
for (int i = 0; i < n; ++i) {
   for (int j = 0; j < n; ++j) {
      register float c = 0;
      for (int k = 0; k < n; ++k) {
        register float a = A[i][k];
        register float b = B[j][k];
        c += a * b;
      }
      C[i][j] = c;
   }
}
```

Take-home Exercise

- Study B100 specification and compare it to H100
    - How nvidia claims another 2x from H100 -> B100?
- How about B200?

Economic Question

Question: What is Nvidia's Moat?



Market Summary > NVIDIA Corp

136.20 USD

+136.16 (340,400.00%) ↑ all time

Closed: Jan 15, 7:59 PM EST • Disclaimer

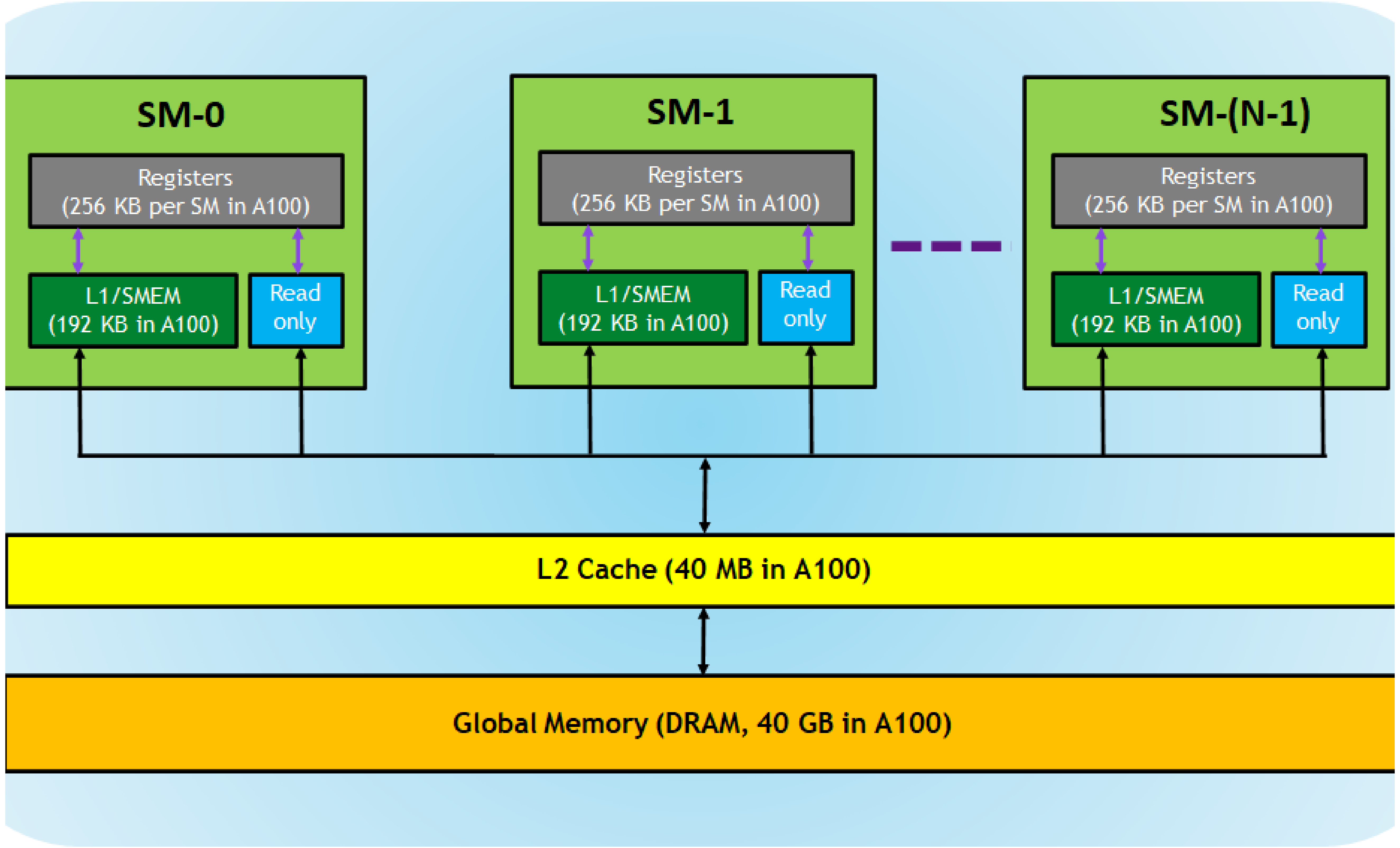
After hours 136.22 +0.020 (0.015%)

Open	133.65	Mkt cap	3.34T	52-wk high	153.13
High	136.45	P/E ratio	53.67	52-wk low	54.74
Low	131.29	Div yield	0.029%		

Next: GPU and CUDA

- Basic concepts and Architecture
- Programming abstraction
- Case study: Matmul

GPU Overview



SM-0

Registers
(256 KB per SM in A100)

L1/SMEM
(192 KB in A100)

Read only

SM-1

Registers
(256 KB per SM in A100)

L1/SMEM
(192 KB in A100)

Read only

SM-(N-1)

Registers
(256 KB per SM in A100)

L1/SMEM
(192 KB in A100)

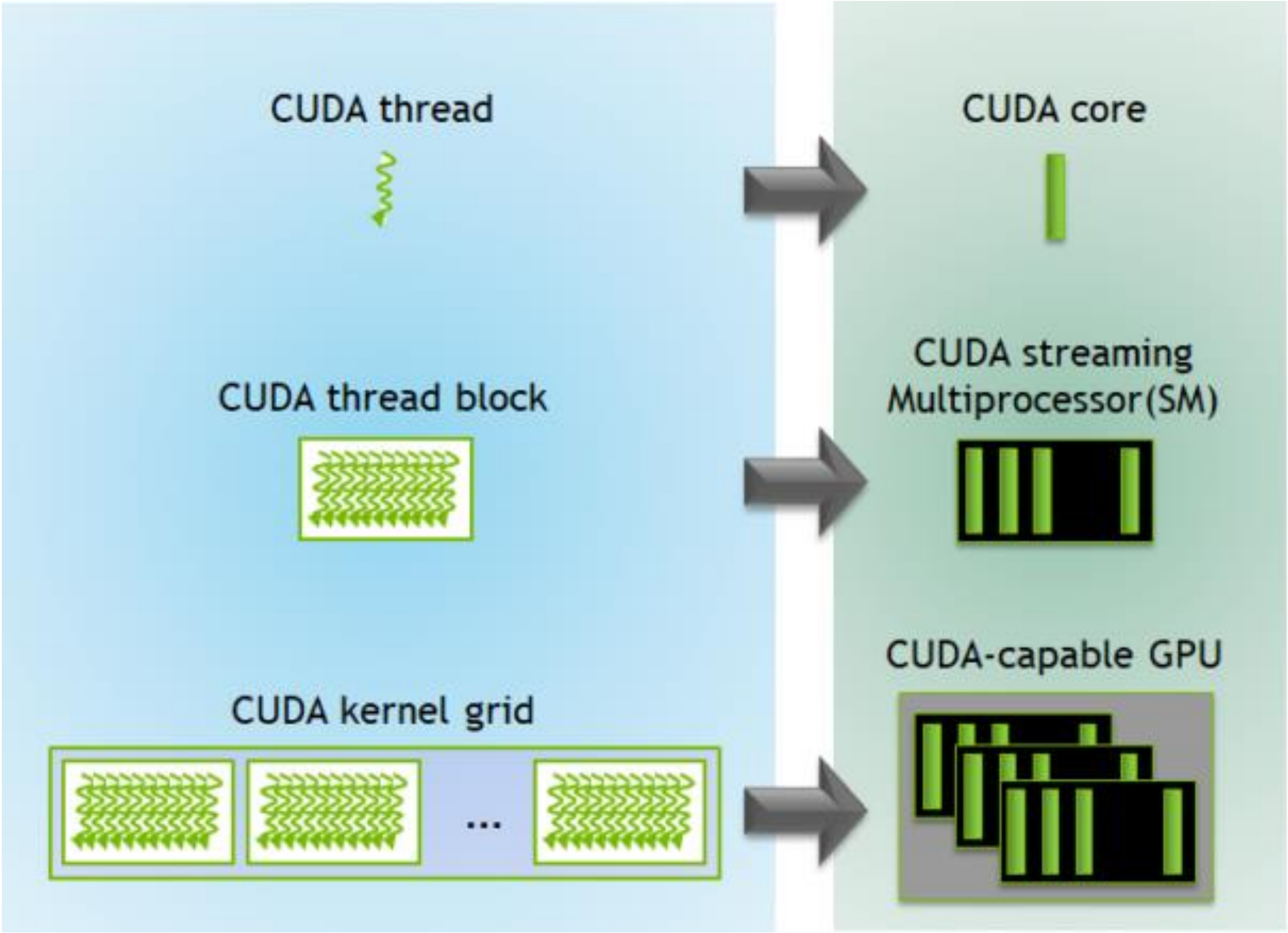
Read only

L2 Cache (40 MB in A100)

Global Memory (DRAM, 40 GB in A100)

Threads, Blocks, Grids

- Threads: smallest units to process a chunk of data
- Blocks: A group of threads that share memory
- Grid: A collection of blocks that execute the same kernel



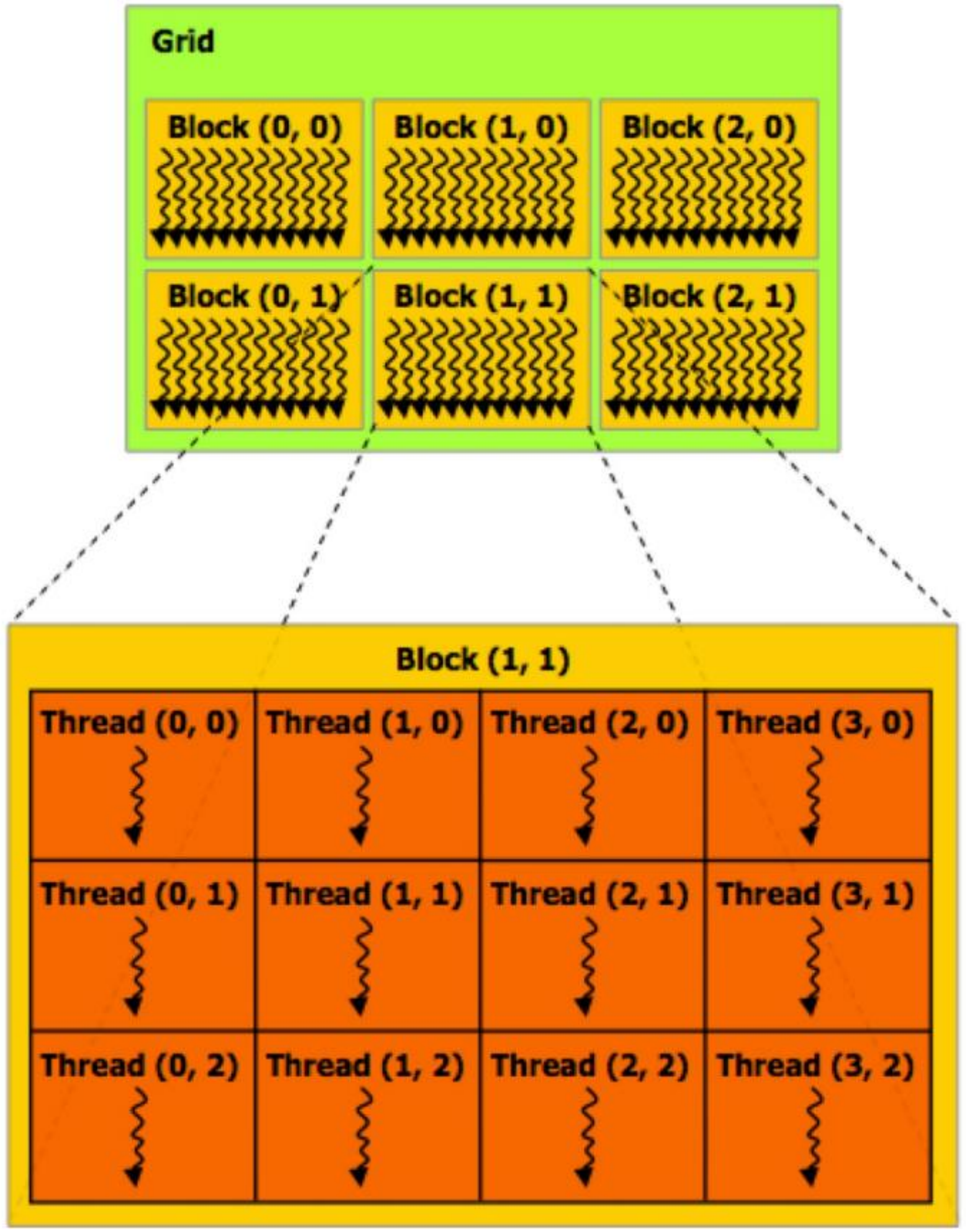
How many SMs/Threads we have?

- V100 (2018 - Now): 80 SMs, 2048 threads/SM
- A100 (2020 - Now): 108 SMs, 2048 threads/SM
- H100 (2022 - Now): 144 SMs, 2048 threads/SM
- B100 (2025 -): go surveying the numbers

CUDA

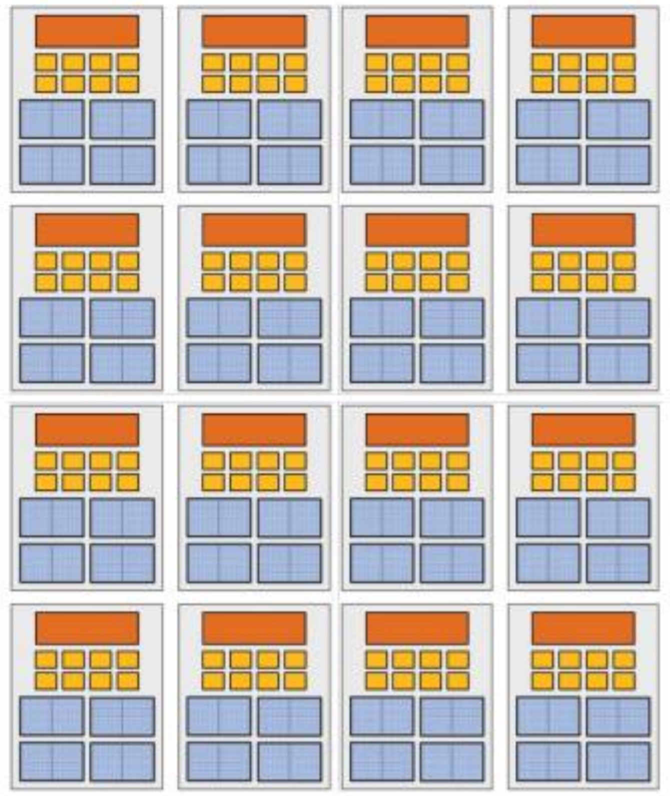
- Introduced in 2007 with NVIDIA Tesla architecture
- C-like languages for programming GPUs
- CUDA’s design matches the grid/block/thread concepts in GPUs

CUDA Programs contain A Hierarchy of Threads

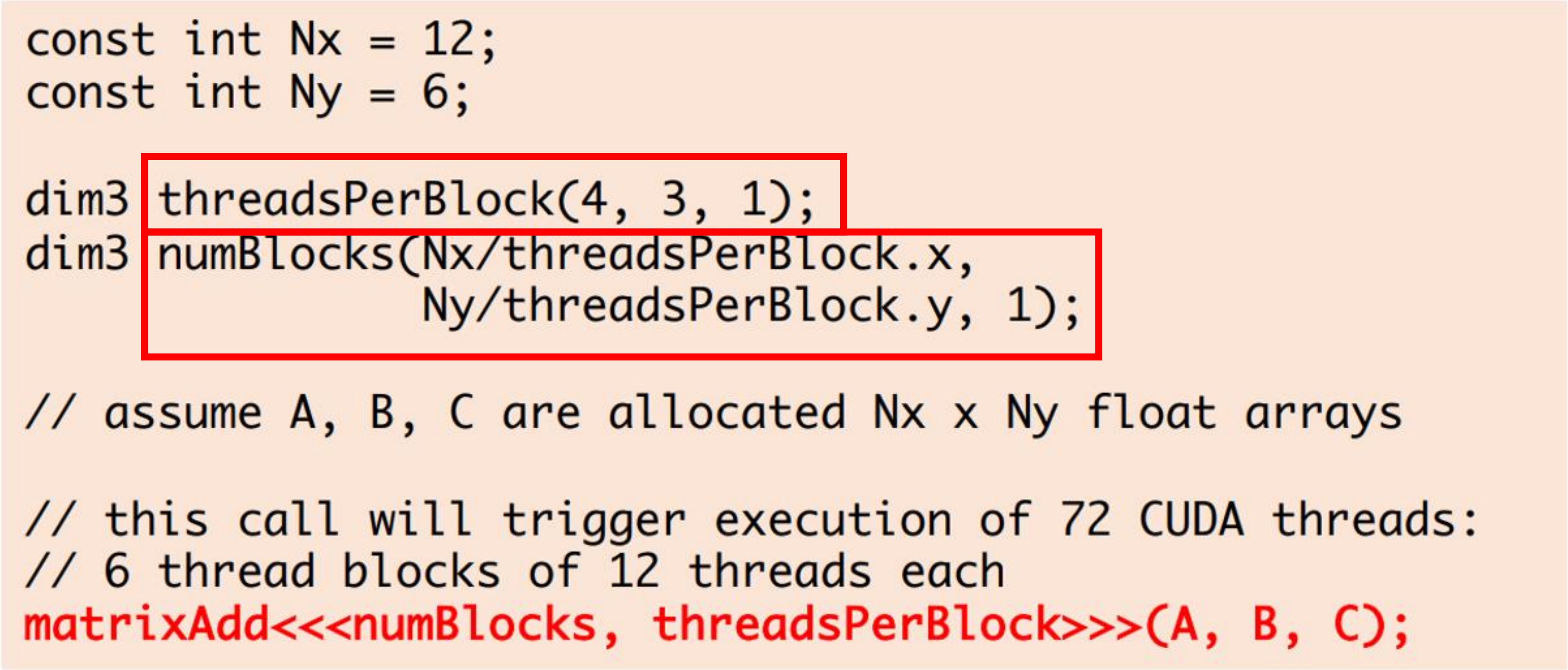


```
const int Nx = 12;
const int Ny = 6;

dim3 threadsPerBlock(4, 3, 1);
dim3 numBlocks(Nx/threadsPerBlock.x,
               Ny/threadsPerBlock.y, 1);

// assume A, B, C are allocated Nx x Ny float arrays

// this call will trigger execution of 72 CUDA threads:
// 6 thread blocks of 12 threads each
matrixAdd<<<numBlocks, threadsPerBlock>>>(A, B, C);
```

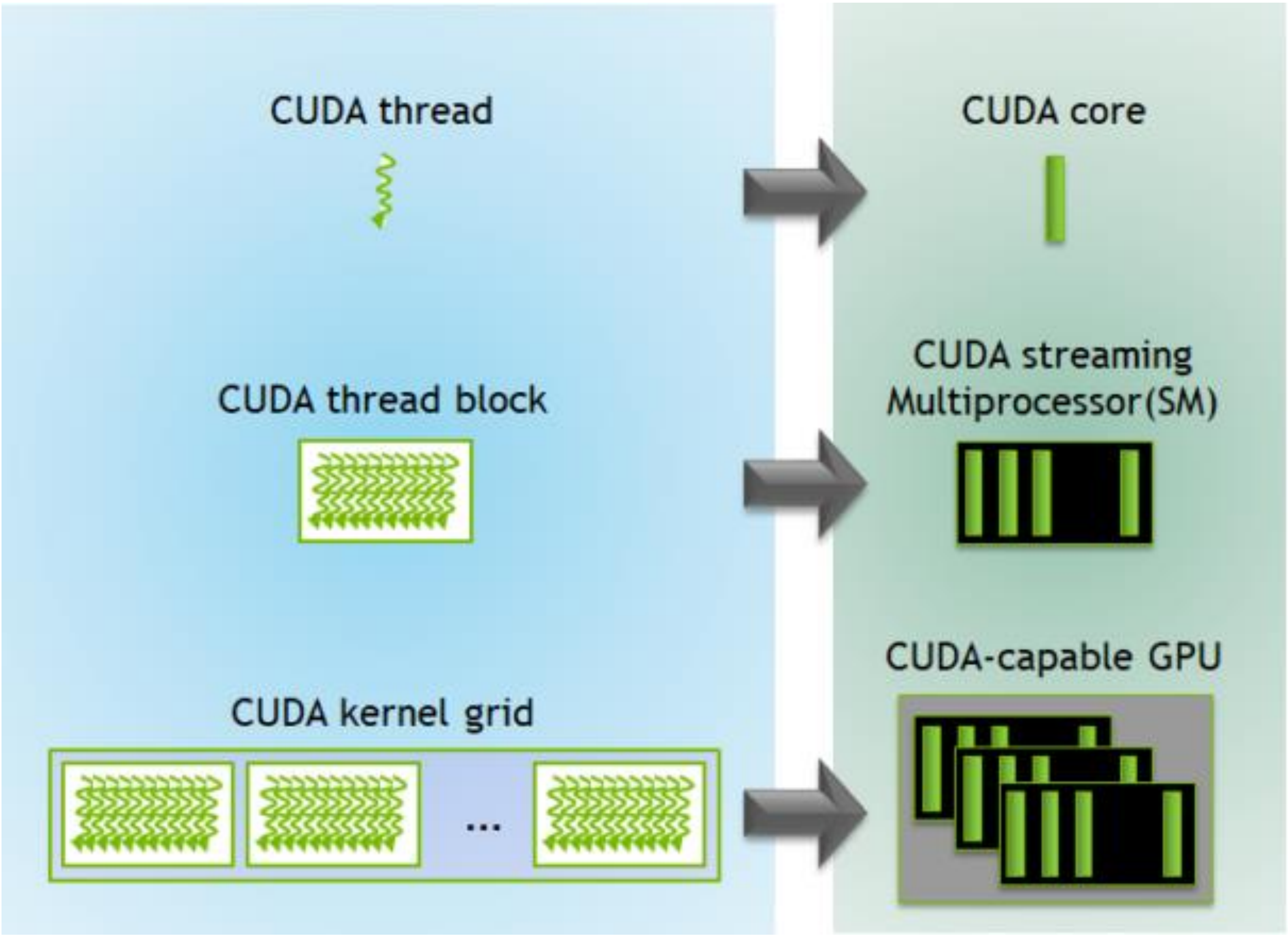
Run on CPU



How Many threads/Blocks it runs on?



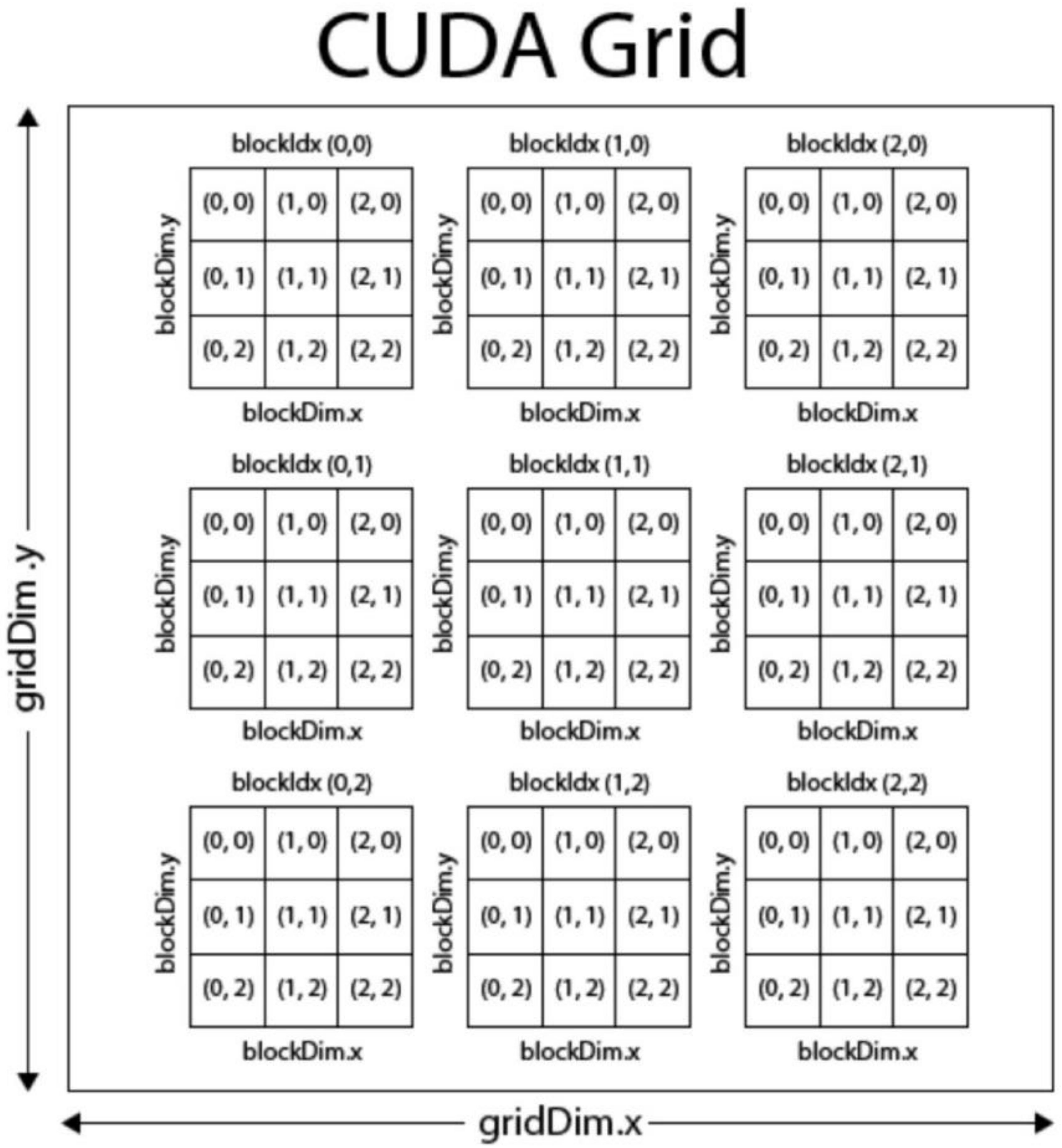
```
const int Nx = 12;
const int Ny = 6;

dim3 threadsPerBlock(4, 3, 1);
dim3 numBlocks(Nx/threadsPerBlock.x,
               Ny/threadsPerBlock.y, 1);

// assume A, B, C are allocated Nx x Ny float arrays

// this call will trigger execution of 72 CUDA threads:
// 6 thread blocks of 12 threads each
matrixAdd<<<numBlocks, threadsPerBlock>>>(A, B, C);
```



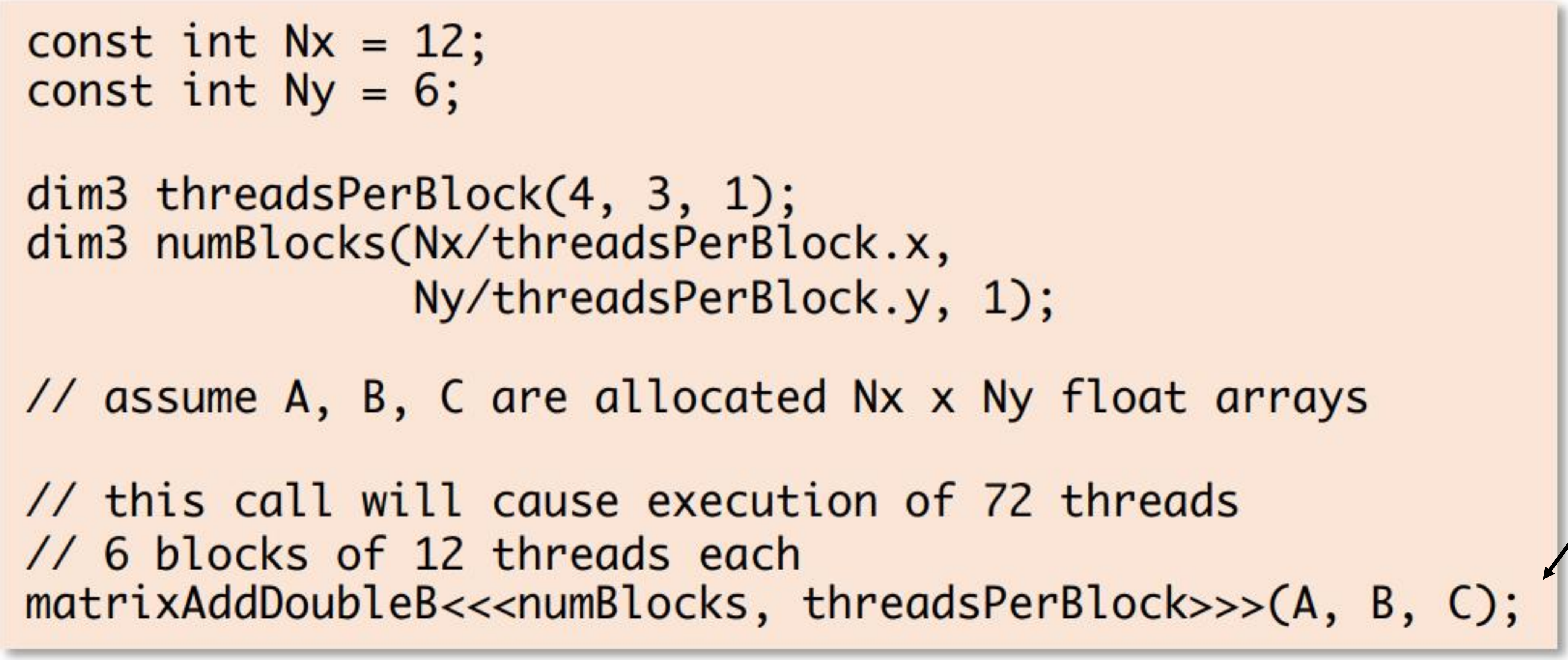
Grid, Block, and Thread

- GridDim: The dimensions of the grid
- blockIdx: The block index within the grid
- blockDim: The dimensions of a block
- threadIdx: The thread index within a block

- What About GridId?
- What about threadDim?

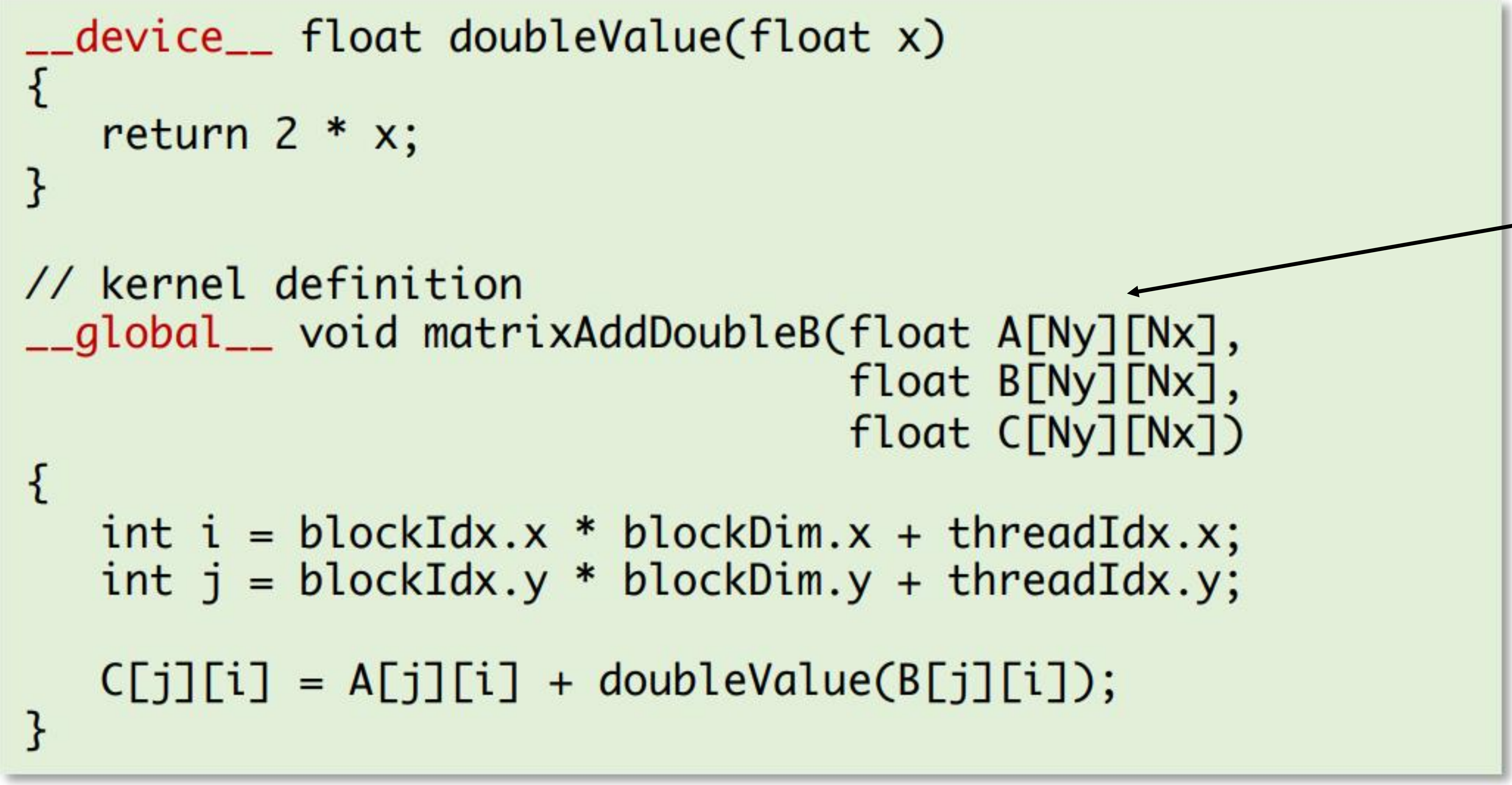


An Example CUDA Program



```
const int Nx = 12;
const int Ny = 6;

dim3 threadsPerBlock(4, 3, 1);
dim3 numBlocks(Nx/threadsPerBlock.x,
               Ny/threadsPerBlock.y, 1);

// assume A, B, C are allocated Nx x Ny float arrays

// this call will cause execution of 72 threads
// 6 blocks of 12 threads each
matrixAddDoubleB<<<numBlocks, threadsPerBlock>>>(A, B, C);
```

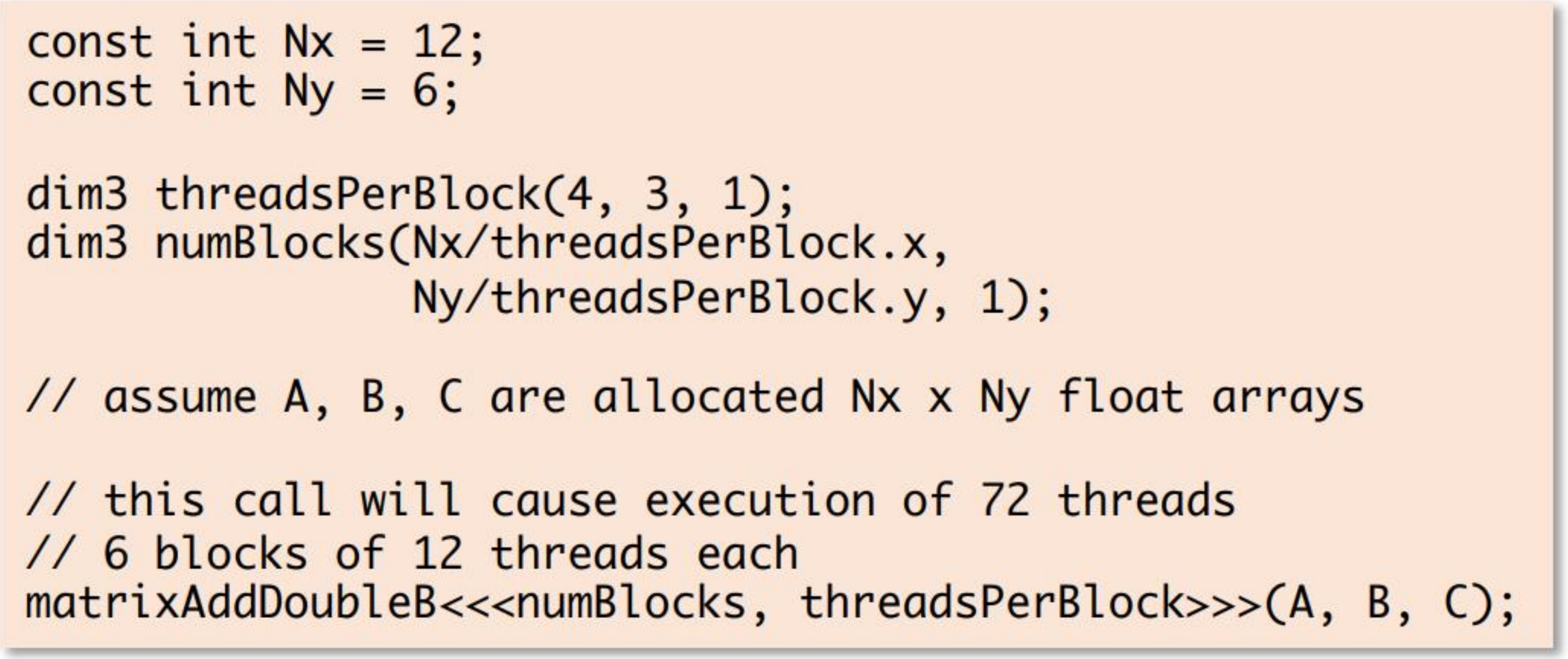
- “launch a grid of CUDA thread
- blocks” Call returns when all threads have terminated



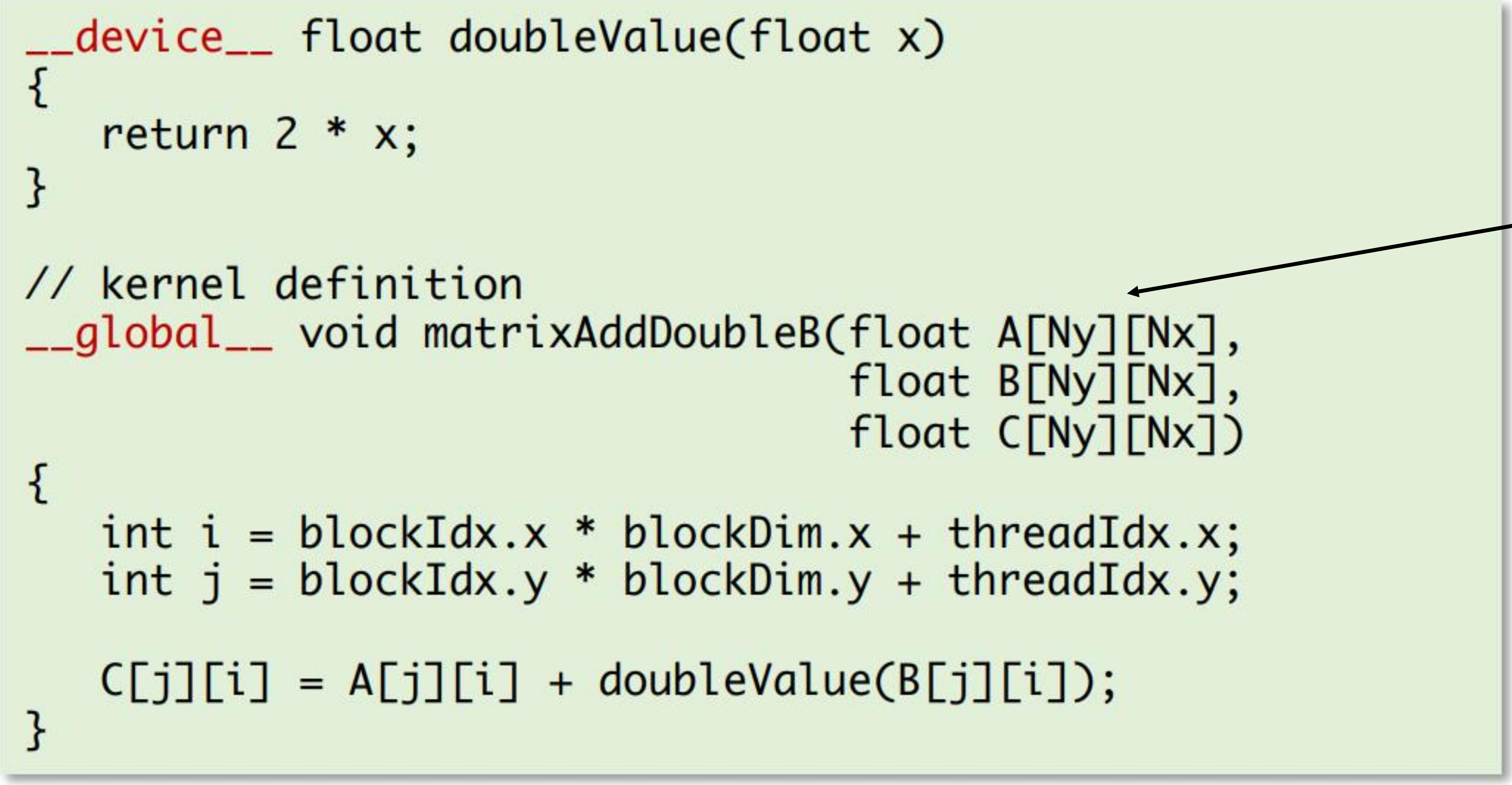
```
__device__ float doubleValue(float x)
{
   return 2 * x;
}

// kernel definition
__global__ void matrixAddDoubleB(float A[Ny][Nx],
                                 float B[Ny][Nx],
                                 float C[Ny][Nx])
{
   int i = blockIdx.x * blockDim.x + threadIdx.x;
   int j = blockIdx.y * blockDim.y + threadIdx.y;

   C[j][i] = A[j][i] + doubleValue(B[j][i]);
}
```

- __global__ denotes a CUDA kernel function runs on GPU
- Each thread indexes its data using blockIdx, blockDim, threadIdx and execute the compute

Separation CPU and GPU Execution



```
const int Nx = 12;
const int Ny = 6;

dim3 threadsPerBlock(4, 3, 1);
dim3 numBlocks(Nx/threadsPerBlock.x,
               Ny/threadsPerBlock.y, 1);

// assume A, B, C are allocated Nx x Ny float arrays

// this call will cause execution of 72 threads
// 6 blocks of 12 threads each
matrixAddDoubleB<<<numBlocks, threadsPerBlock>>>(A, B, C);
```

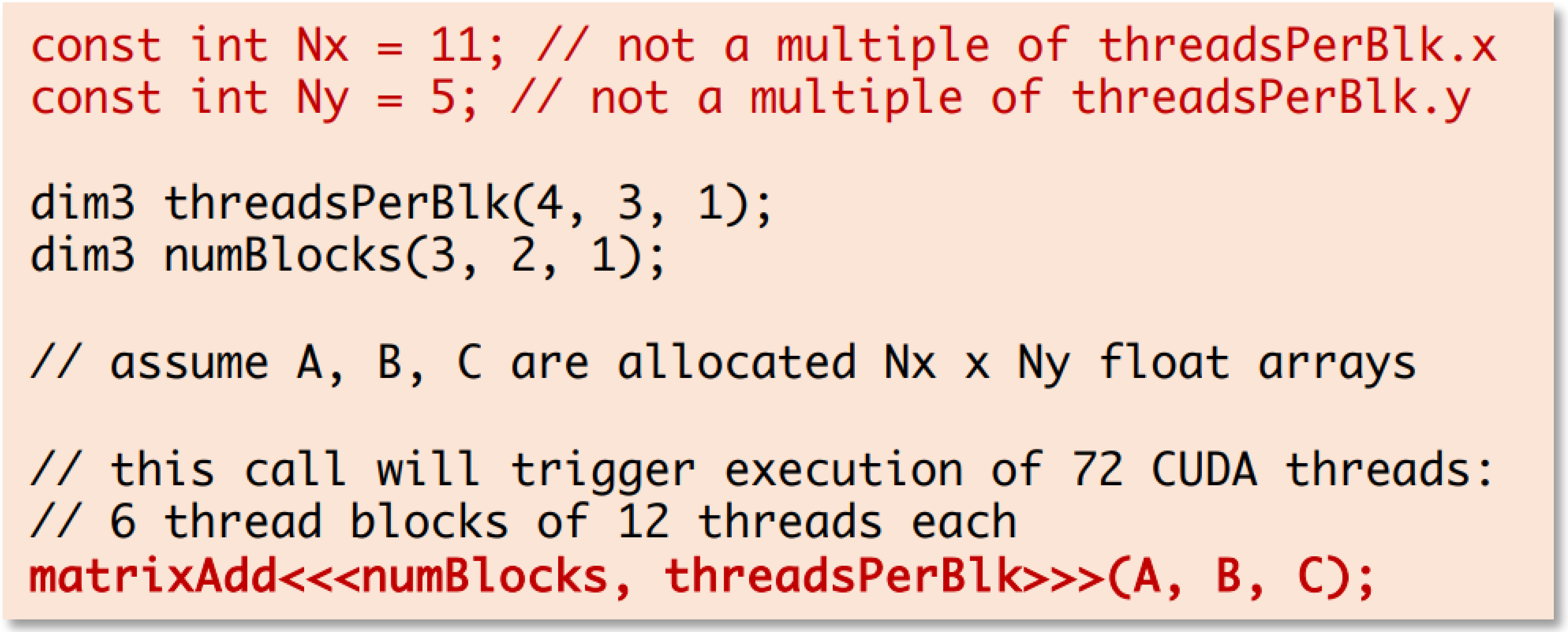


```
__device__ float doubleValue(float x)
{
   return 2 * x;
}

// kernel definition
__global__ void matrixAddDoubleB(float A[Ny][Nx],
                                 float B[Ny][Nx],
                                 float C[Ny][Nx])
{
   int i = blockIdx.x * blockDim.x + threadIdx.x;
   int j = blockIdx.y * blockDim.y + threadIdx.y;

   C[j][i] = A[j][i] + doubleValue(B[j][i]);
}
```

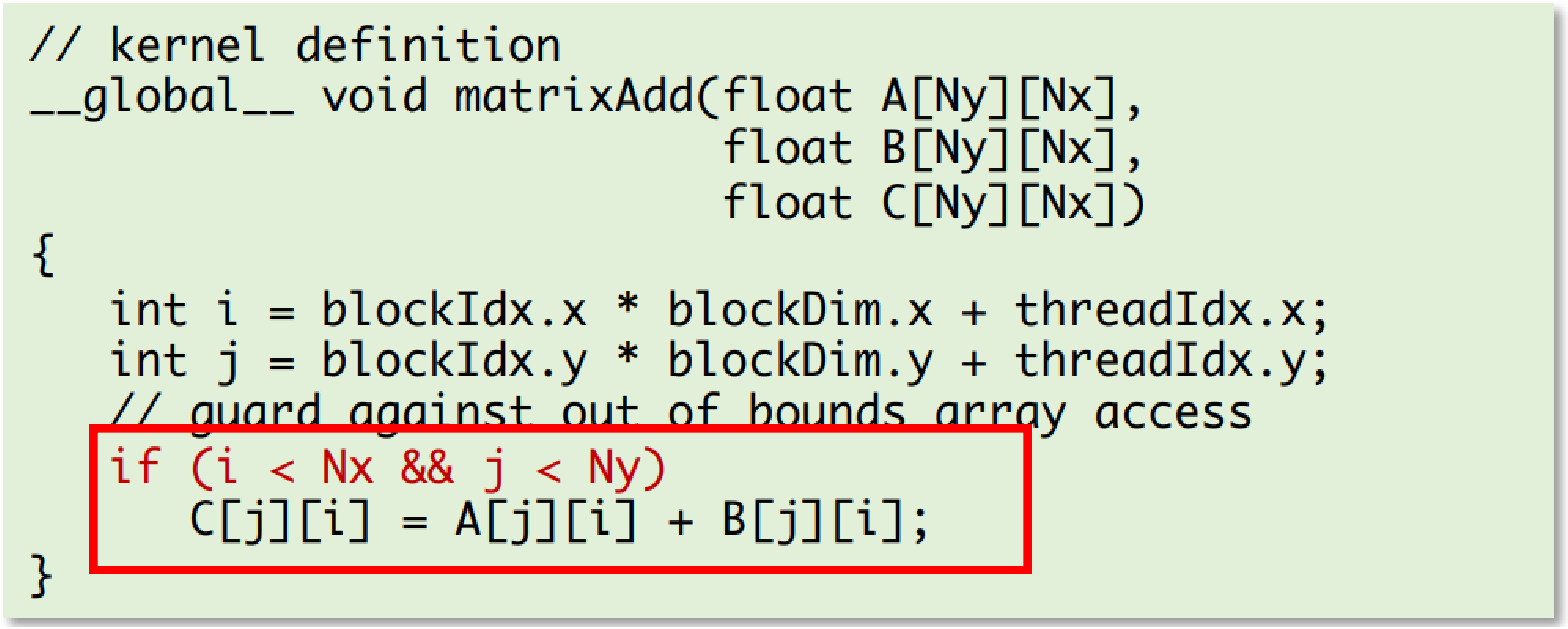
- Host code: serial execution on CPU

- Device code: SIMD parallel execution on GPUs

#Threads is Explicit and Static in Programs



```
const int Nx = 11; // not a multiple of threadsPerBlk.x
const int Ny = 5; // not a multiple of threadsPerBlk.y

dim3 threadsPerBlk(4, 3, 1);
dim3 numBlocks(3, 2, 1);

// assume A, B, C are allocated Nx x Ny float arrays

// this call will trigger execution of 72 CUDA threads:
// 6 thread blocks of 12 threads each
matrixAdd<<<numBlocks, threadsPerBlk>>>(A, B, C);
```



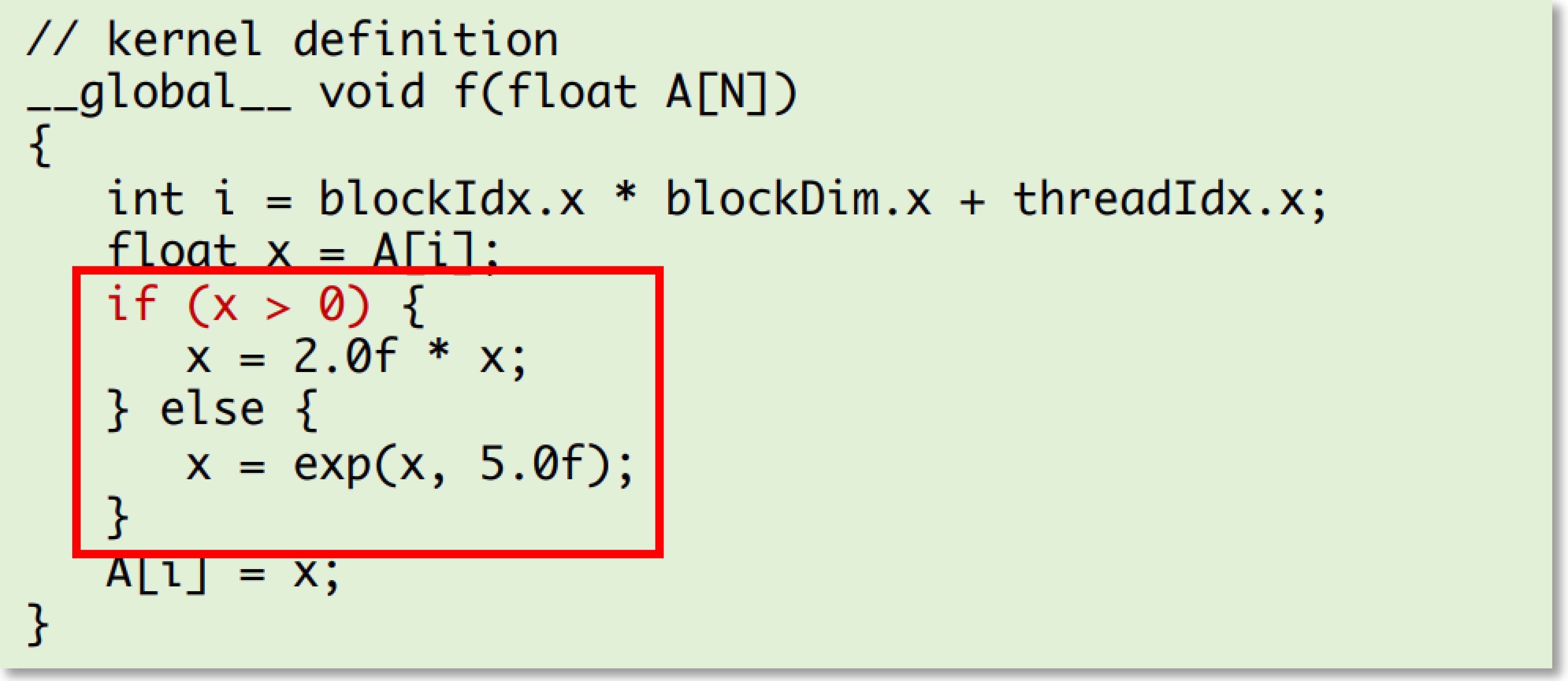
```
// kernel definition
__global__ void matrixAdd(float A[Ny][Nx],
                          float B[Ny][Nx],
                          float C[Ny][Nx])
{
   int i = blockIdx.x * blockDim.x + threadIdx.x;
   int j = blockIdx.y * blockDim.y + threadIdx.y;
   // guard against out of bounds array access
   if (i < Nx && j < Ny)
      C[j][i] = A[j][i] + B[j][i];
}
```

Developers to:

- To provide CPU/GPU code separation
- Statically declare blockDim, shapes.
- Map data to blocks/threads
- Check boundary conditions

SIMD Constraints: how to handle control flow?

SIMD requires all ALUs/Core Must proceed in the same pace



```
// kernel definition
__global__ void f(float A[N])
{
   int i = blockIdx.x * blockDim.x + threadIdx.x;
   float x = A[i];
   if (x > 0) {
      x = 2.0f * x;
   } else {
      x = exp(x, 5.0f);
   }
   A[i] = x;
}
```