https://haoailab.com/cse291-s26/

CSE/DSC 291: Deep Learning Systems
Spring 2026

LLM, diffusion, and case studies

Optimizations and Parallelization

Basics

MCQ Time

What is the arithmetic intensity for the following function:

```
# A, B are 2-D matrices of shape [4,4]
func(matrix A, matrix B):
      Load A
      Load B
      C = matmul(A,B)
```

Important Notes!!

FLOPs of matmul: C = A x B

- A: (m, n)
- B: (n, p)
- C: (m, p)
- Flops: 2mnp

Which of the following Tensor manipulations cannot benefit from strided representation

A. Broadcast_to

B. Slice

C. Reshape

D. Permute dimensions

E. Transpose

F. contiguous

G. indexing like t[:, 1:5]

If we have tensor of shape [2,9,1] stored contiguous in memory following row Major, what is its strides?

A. (9,1,1)
B. (2,9,1)
C. (1,9,2)
D. (9,9,9)

Which of the following is True for *Cache Tiling* in Matmul

A. It saves memory allocated in Cache

B. It reduce the memory movement between Cache to Register

C. It reuses memory movement between Dram and Cache

D. It increases arithmetic intensity because it makes the computation faster

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 of Our Matmul Progress

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];
    }
  }
```

```
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)
         }
       }
   }
}
```

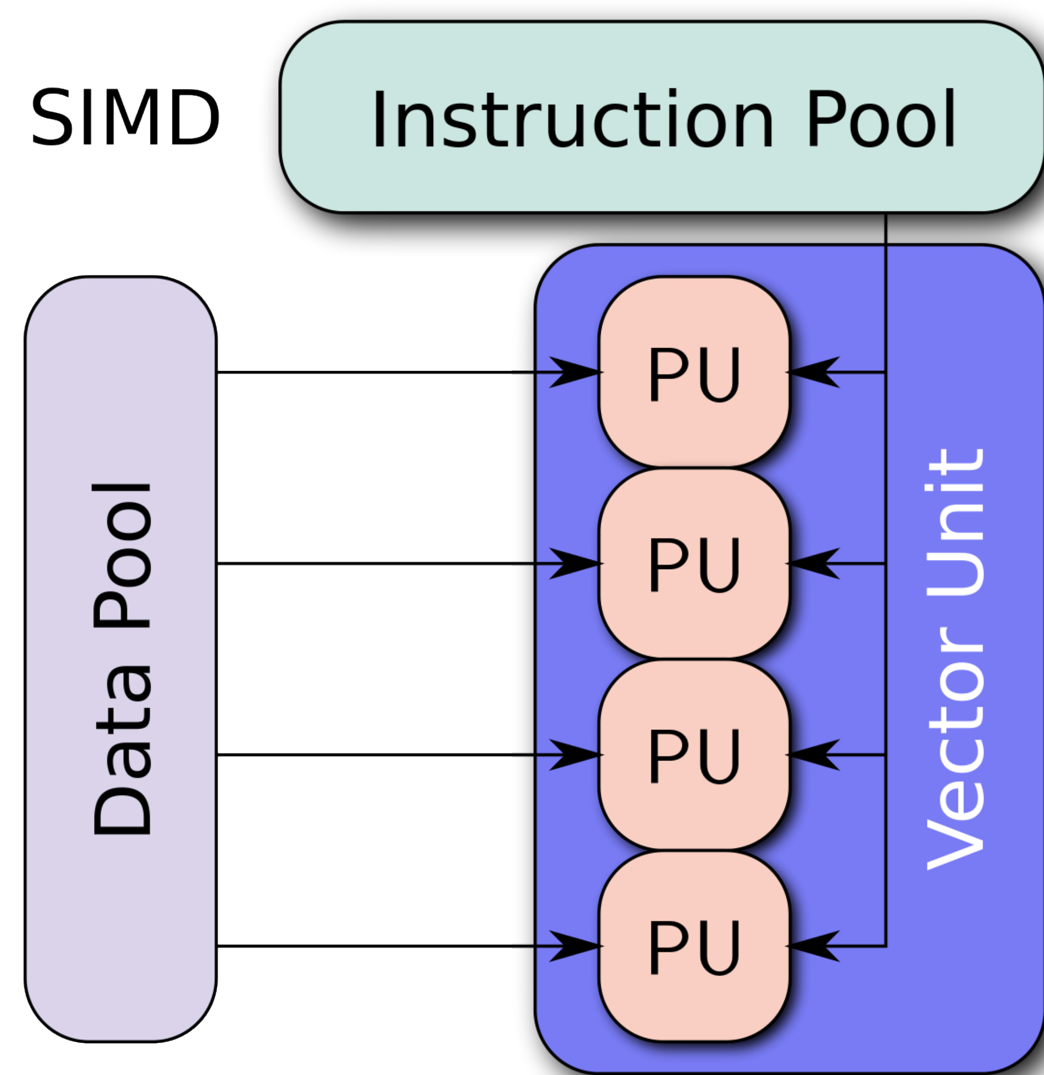
Recap CPU parallelization

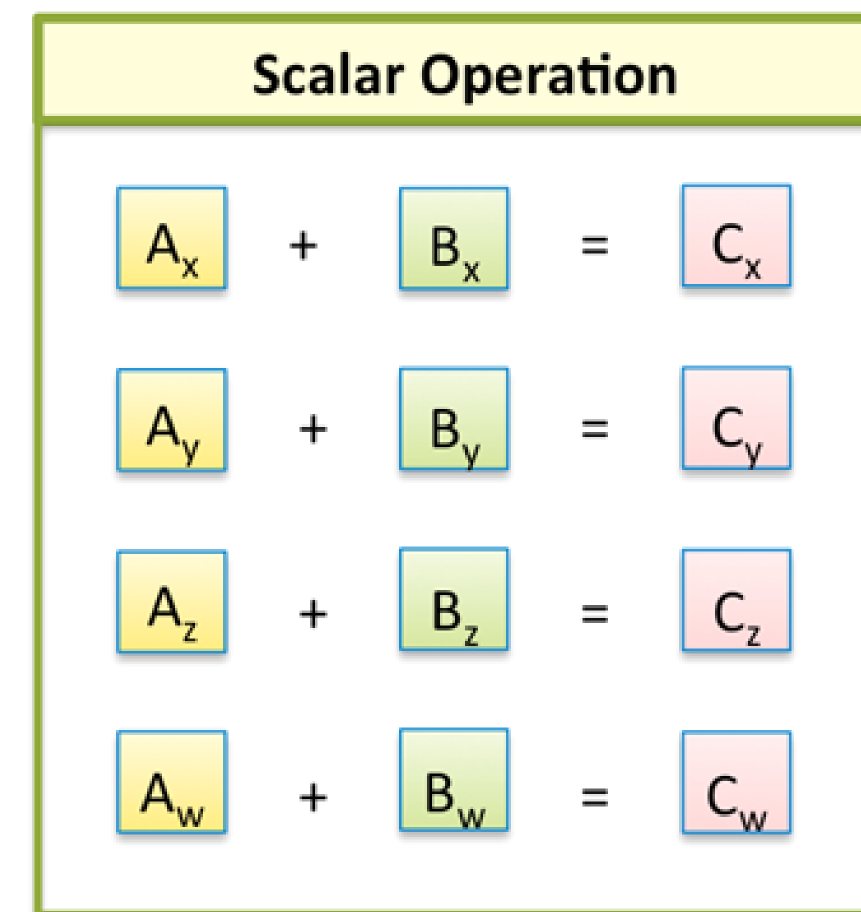
- We can parallelize this loop using CPU threads
- == using many concurrent cores

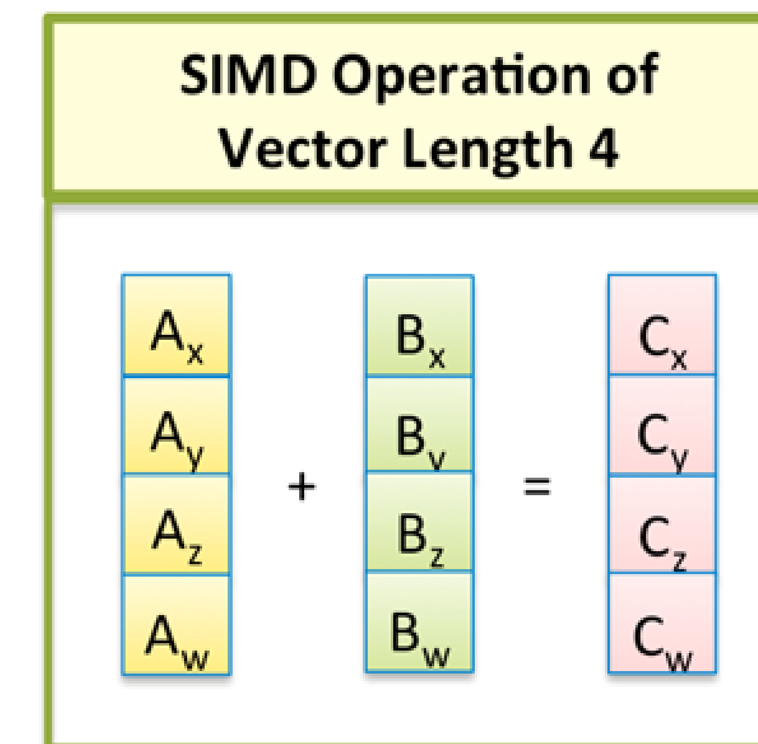
```
#pragma omp parallel for
for (int i = 0; i < 64; ++i) {
      float4 a = load_float4(A + i*4);
      float4 b = load_float4(B + i*4);
      float4 c = add_float4(a, b);
      store_float4(C * 4, c);
}
```

Vectorized & parallelized

Single-Instruction Multiple-Data



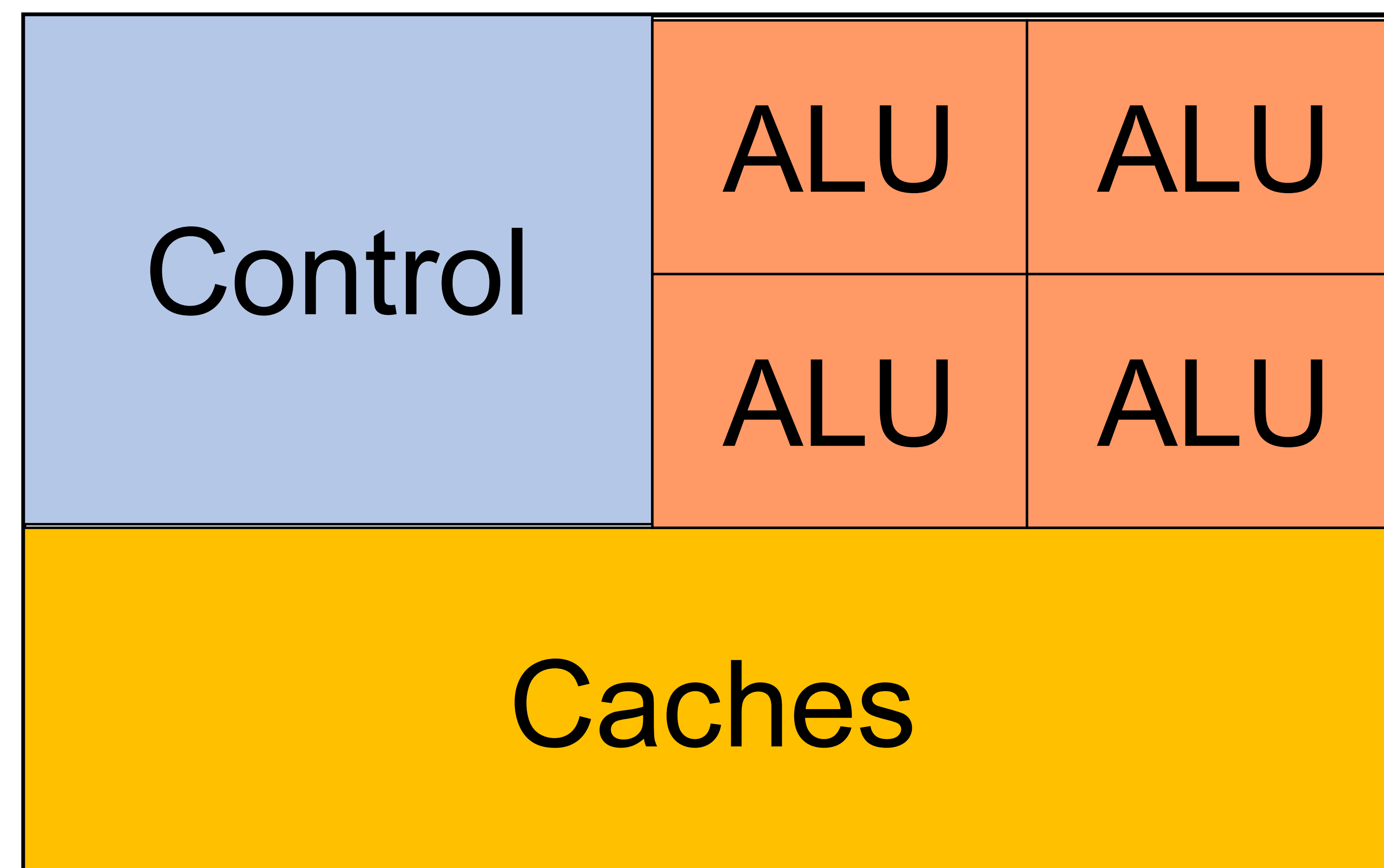


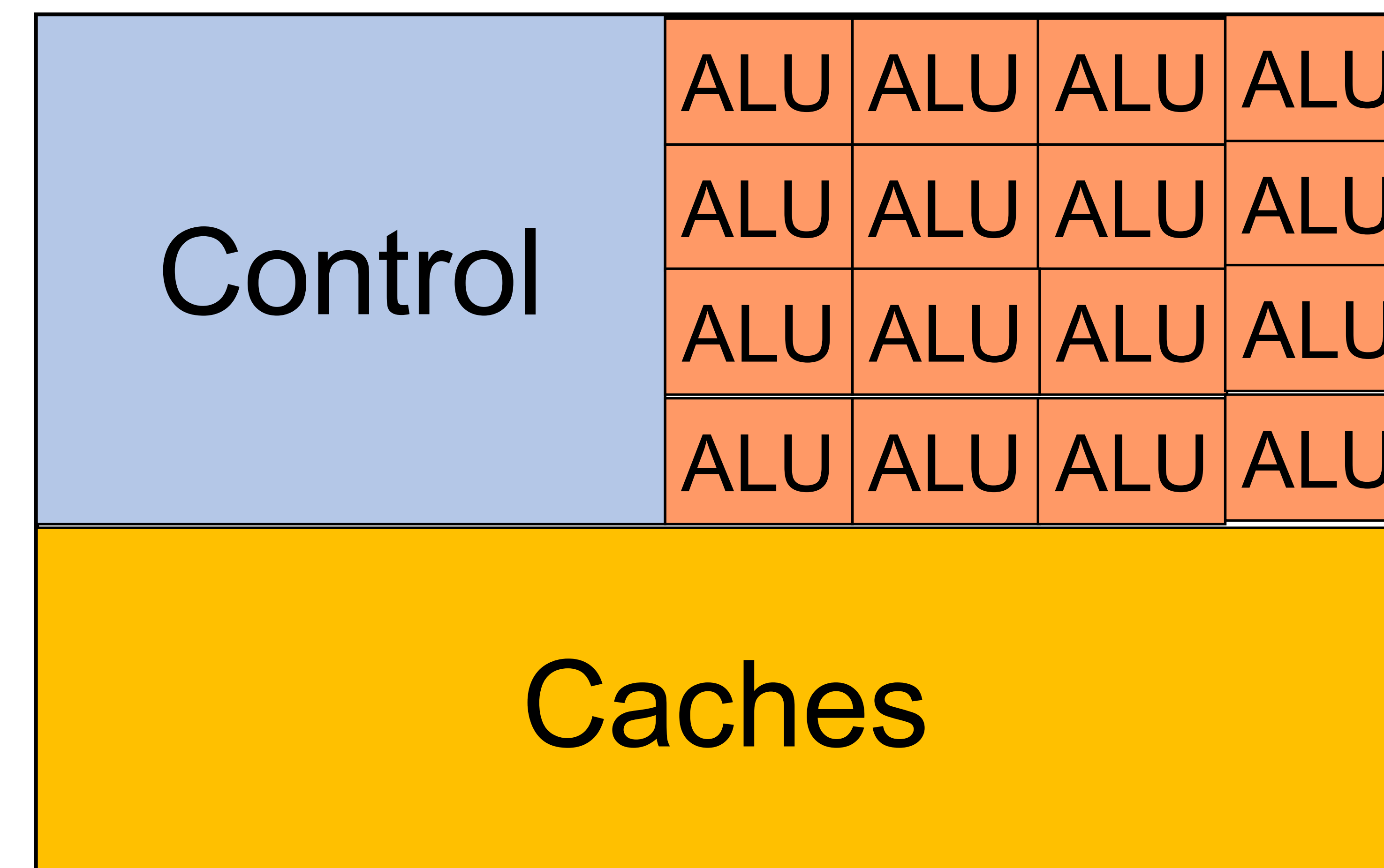


Intel® Architecture currently has SIMD operations of vector length 4, 8, 16

Chip Design Trajectory: SIMD

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

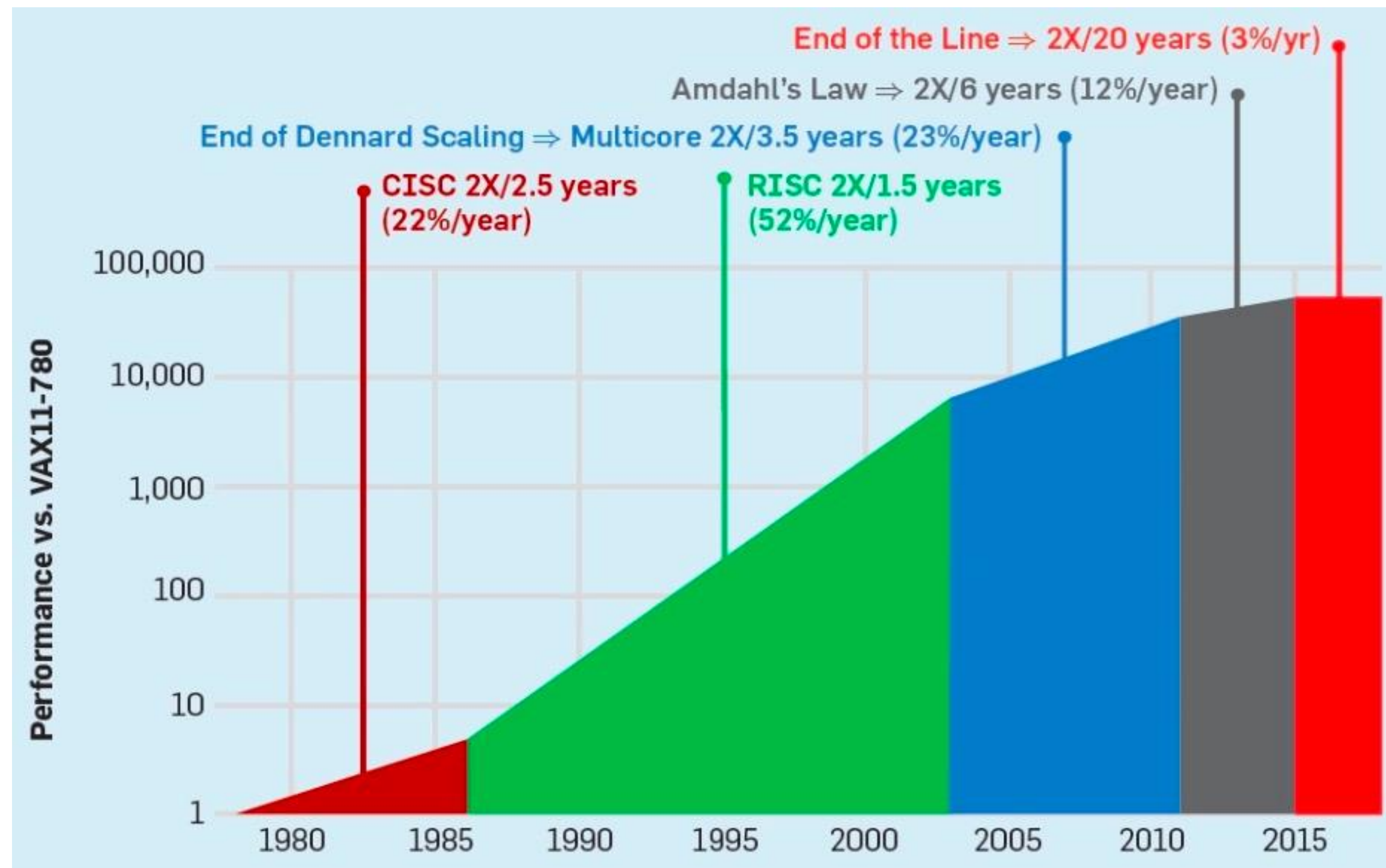




Chip Industry: 70nm -> 60nm -> 50nm -> … ->?

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

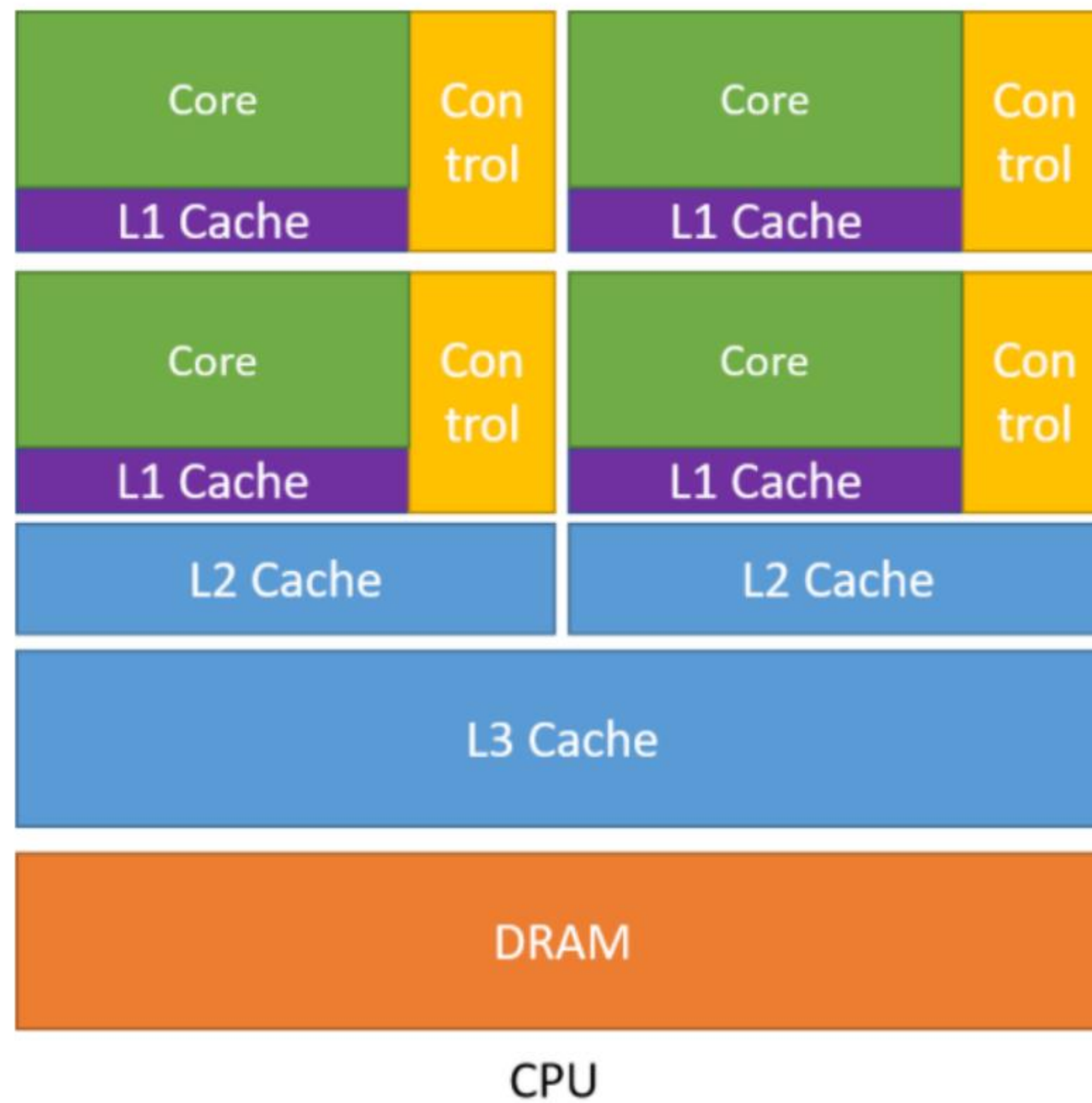
Chip Industry: Moore's Law Comes to an End

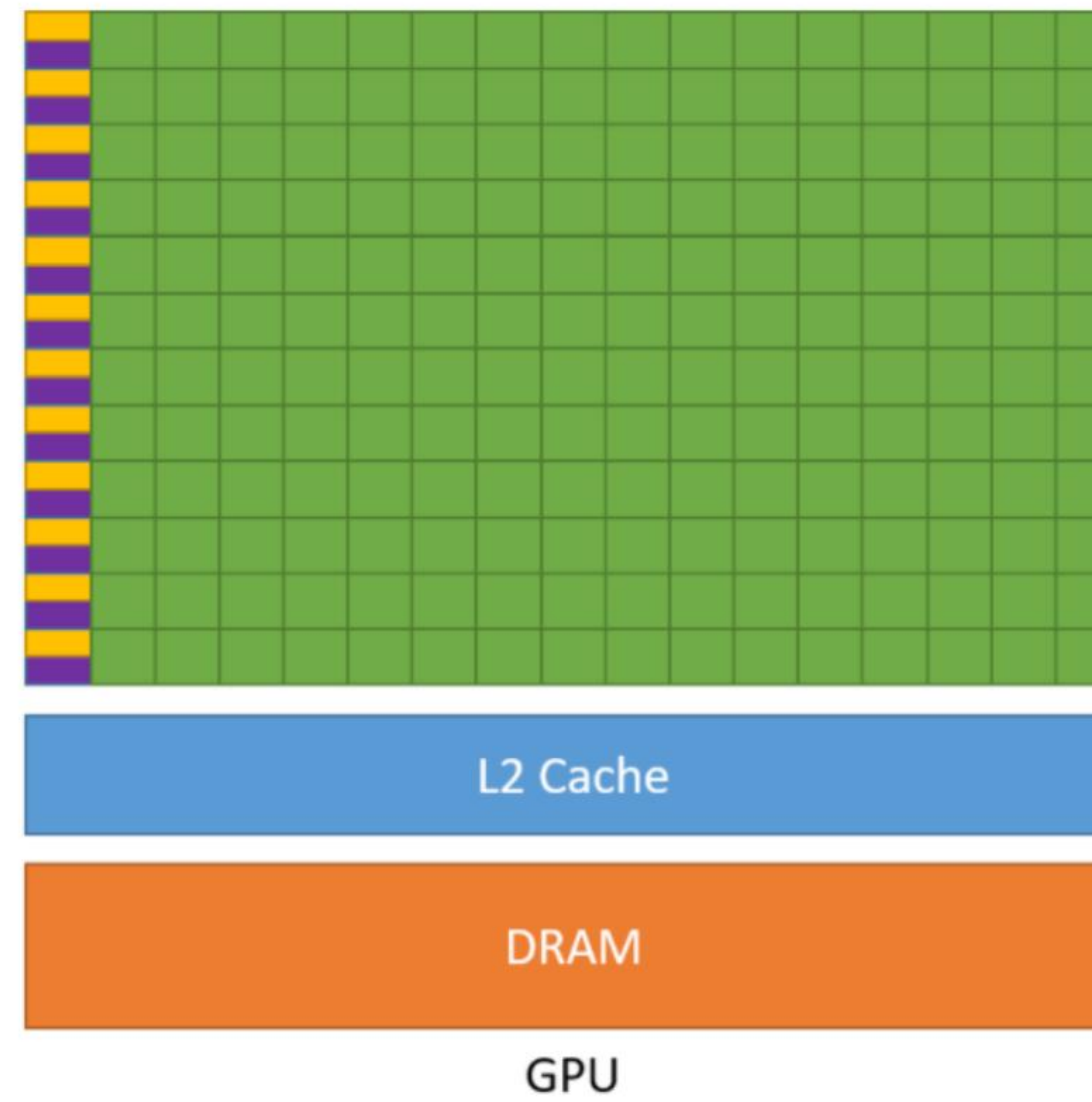


Option 1: Quantum computing

Option 2: Specialized hardware

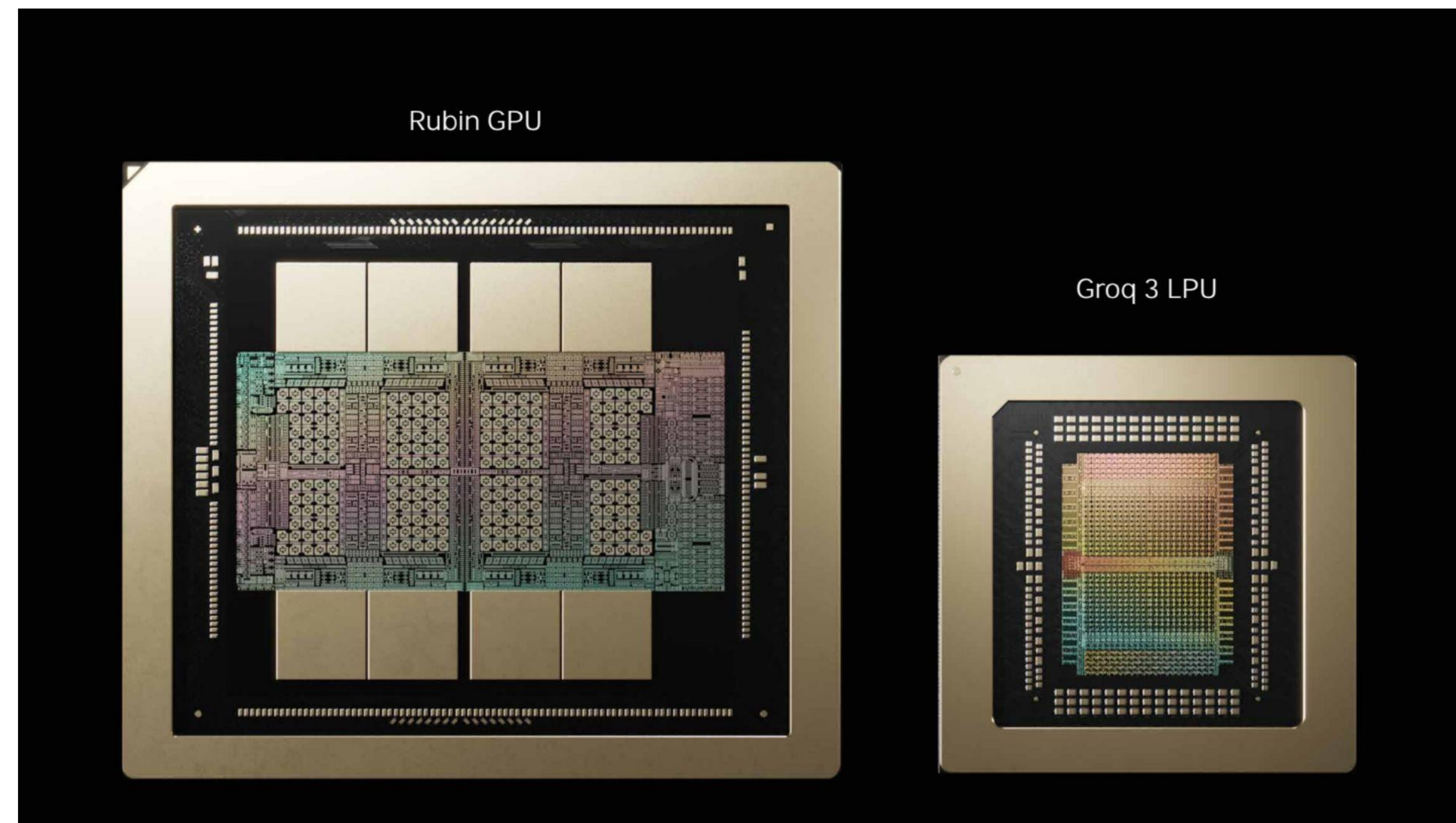
Idea: How about we use a lot of weak/specialized cores





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



Other Hardware Accelerators

- E.g.
  - Tensor Processing Unit (TPU)
  - An "application-specific integrated circuit" (ASIC) created by Google in mid 2010s; used for AlphaGo

What Does It Mean by “Specialized” In accelerator world

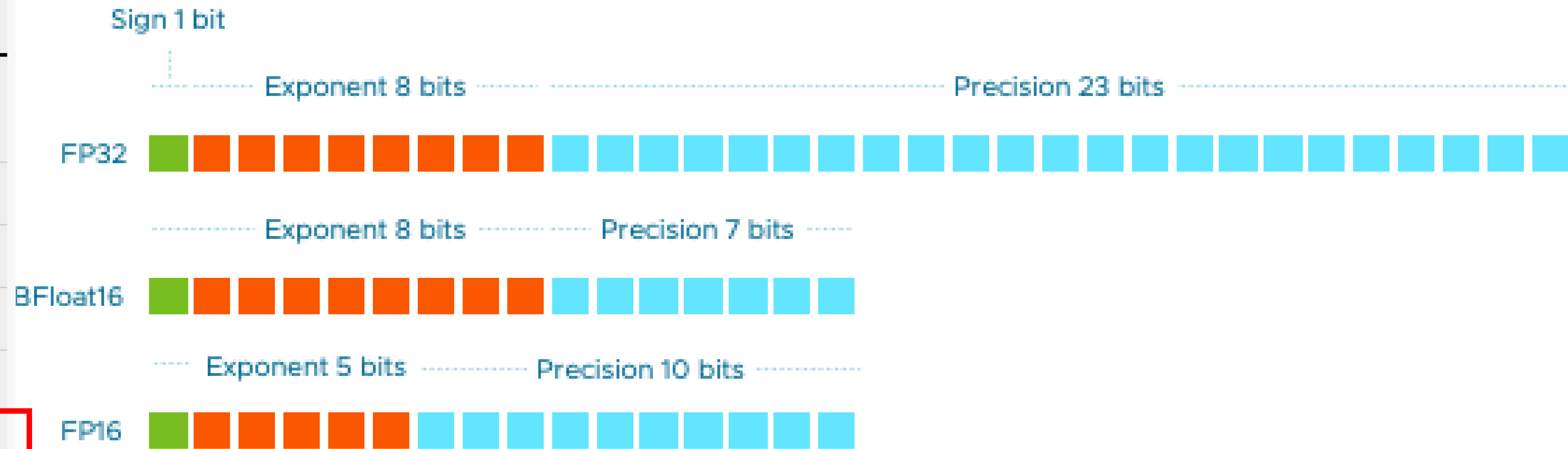
- Functionality-specialized:
  - Can only compute certain computations: matmul, w/ sparsity
  - Mixing specialized cores with versatile cores
- Reduce precision
  - Floating point operations: fp32, fp16, fp8, int8, int4, …
- Tune the distribution of different components for specific workloads
  - SRAM,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

Case Study 1: Nvidia GPU Specification

	NVIDIA Vera Rubin NVL72	NVIDIA Vera Rubin Superchip	NVIDIA Rubin GPU
Configuration	72 NVIDIA Rubin GPUs | 36 NVIDIA Vera CPUs	2 NVIDIA Rubin GPUs | 1 NVIDIA Vera CPU	1 NVIDIA Rubin GPU
NVFP4 Inference	3,600 PFLOPS	100 PFLOPS	50 PFLOPS
NVFP4 Training[2]	2,520 PFLOPS	70 PFLOPS	35 PFLOPS
FP8/FP6 Training[2]	1,260 PFLOPS	35 PFLOPS	17.5 PFLOPS
INT8[2]	18 POPS	0.5 POPS	0.25 POPS
FP16/BF16[2]	288 PFLOPS	8 PFLOPS	4 PFLOPS
TF32[2]	144 PFLOPS	4 PFLOPS	2 PFLOPS
FP32	9,360 TFLOPS	260 TFLOPS	130 TFLOPS
FP64	2,400 TFLOPS	67 TFLOPS	33 TFLOPS
FP32 SGEMM[3]	28,800 TFLOPS	800 TFLOPS	400 TFLOPS
FP64 DGEMM[3]	14,400 TFLOPS	400 TFLOPS	200 TFLOPS
GPU Memory | Bandwidth	20.7 TB HBM4 | 1,580 TB/s	576 GB HBM4 | 44 TB/s	288 GB HBM4 | 22 TB/s
NVLink Bandwidth	260 TB/s	7.2 TB/s	3.6 TB/s

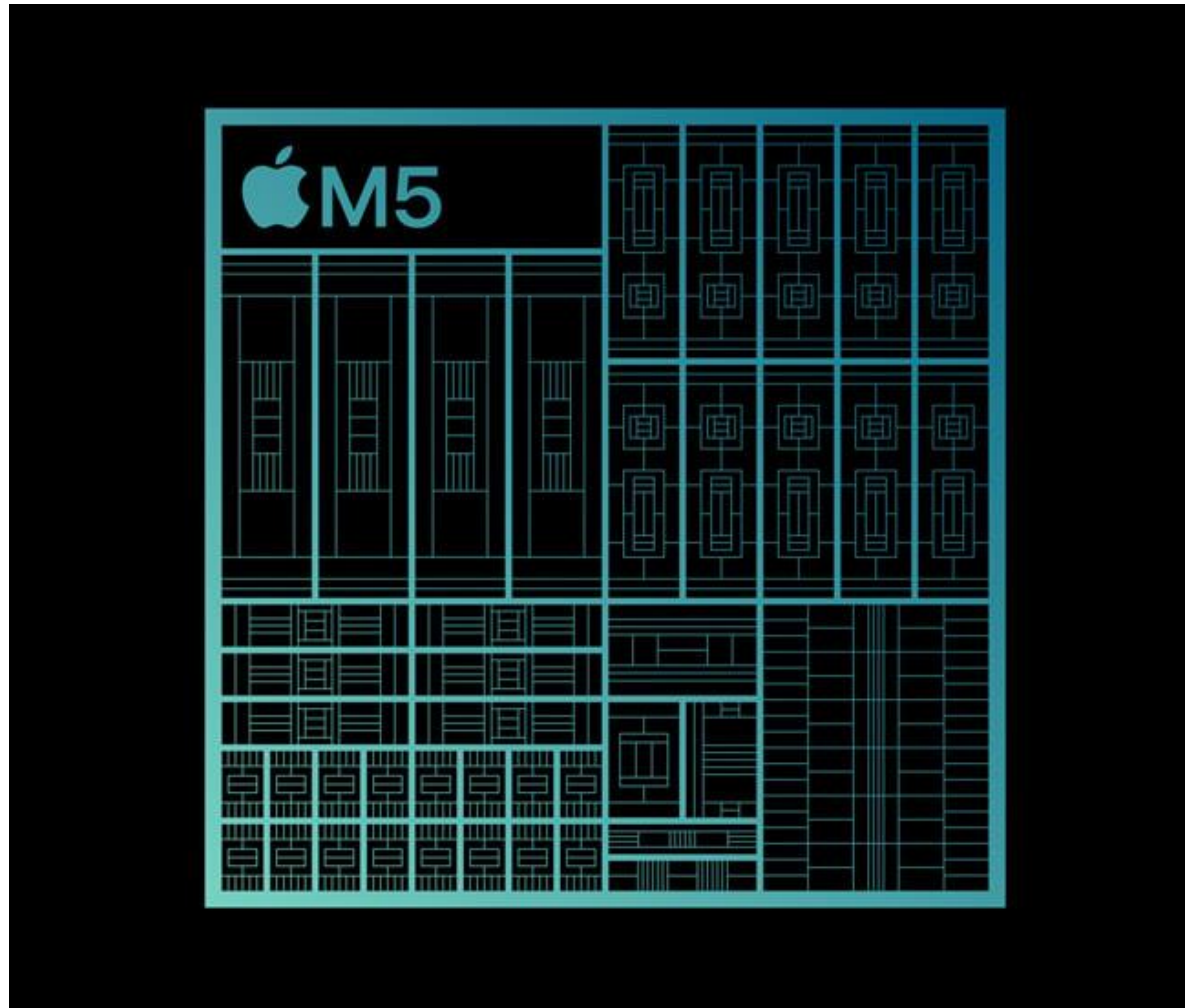
Case Study 1: Nvidia GPU Specification

	NVIDIA Vera Rubin NVL72	NVIDIA Vera Rubin Superchip	NVIDIA Rubin GPU
Configuration	72 NVIDIA Rubin GPUs | 36 NVIDIA Vera CPUs	2 NVIDIA Rubin GPUs | 1 NVIDIA Vera CPU	1 NVIDIA Rubin GPU
NVFP4 Inference	3,600 PFLOPS	100 PFLOPS	50 PFLOPS
NVFP4 Training[2]	2,520 PFLOPS	70 PFLOPS	35 PFLOPS
FP8/FP6 Training[2]	1,260 PFLOPS	35 PFLOPS	17.5 PFLOPS
INT8[2]	18 POPS	0.5 POPS	0.25 POPS
FP16/BF16[2]	288 PFLOPS	8 PFLOPS	4 PFLOPS
TF32[2]	144 PFLOPS	4 PFLOPS	2 PFLOPS
FP32	9,360 TFLOPS	260 TFLOPS	130 TFLOPS
FP64	2,400 TFLOPS	67 TFLOPS	33 TFLOPS
FP32 SGEMM[3]	28,800 TFLOPS	800 TFLOPS	400 TFLOPS
FP64 DGEMM[3]	14,400 TFLOPS	400 TFLOPS	200 TFLOPS
GPU Memory | Bandwidth	20.7 TB HBM4 | 1,580 TB/s	576 GB HBM4 | 44 TB/s	288 GB HBM4 | 22 TB/s
NVLink Bandwidth	260 TB/s	7.2 TB/s	3.6 TB/s

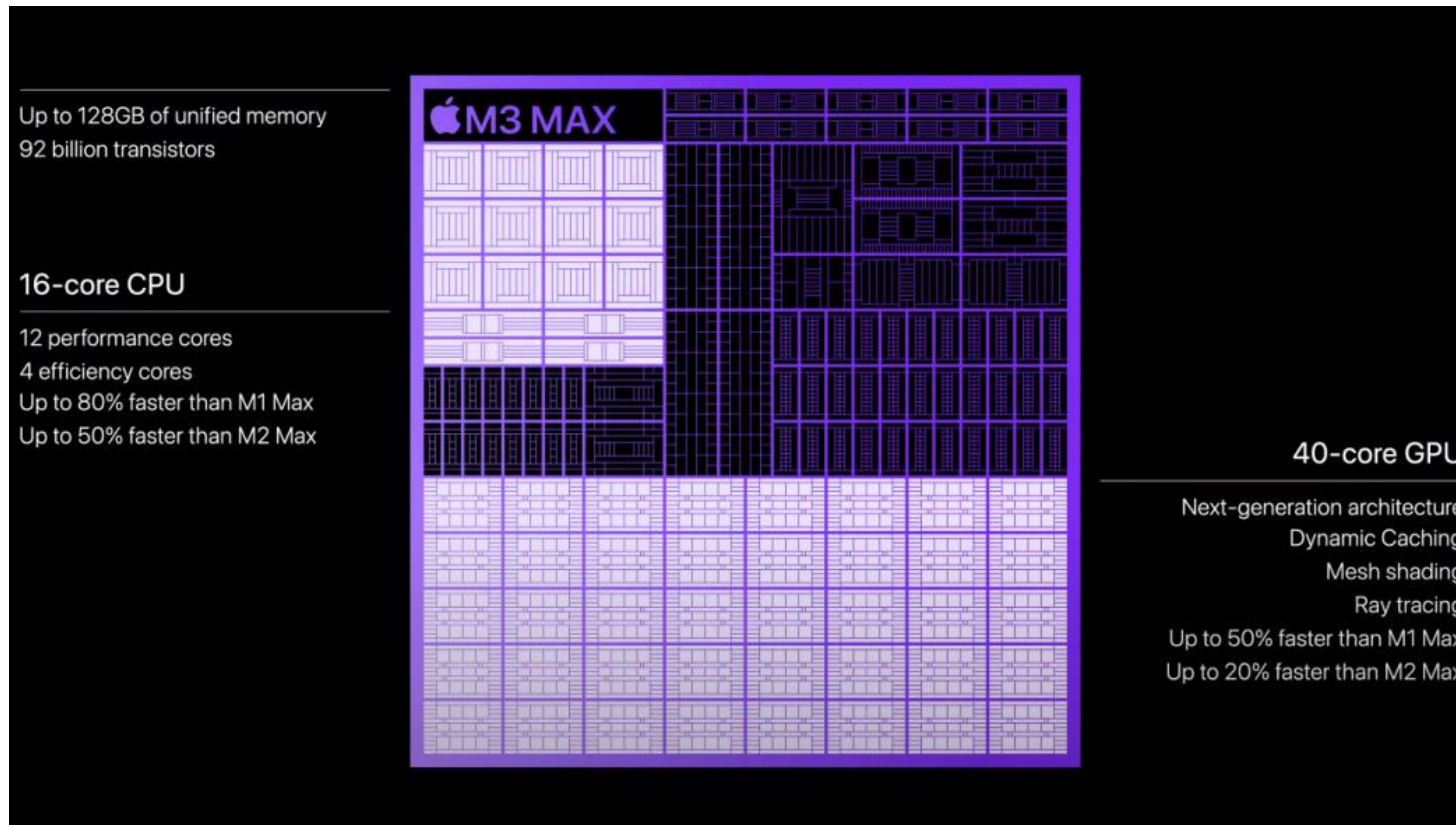


Question: why this could work in ML programs?

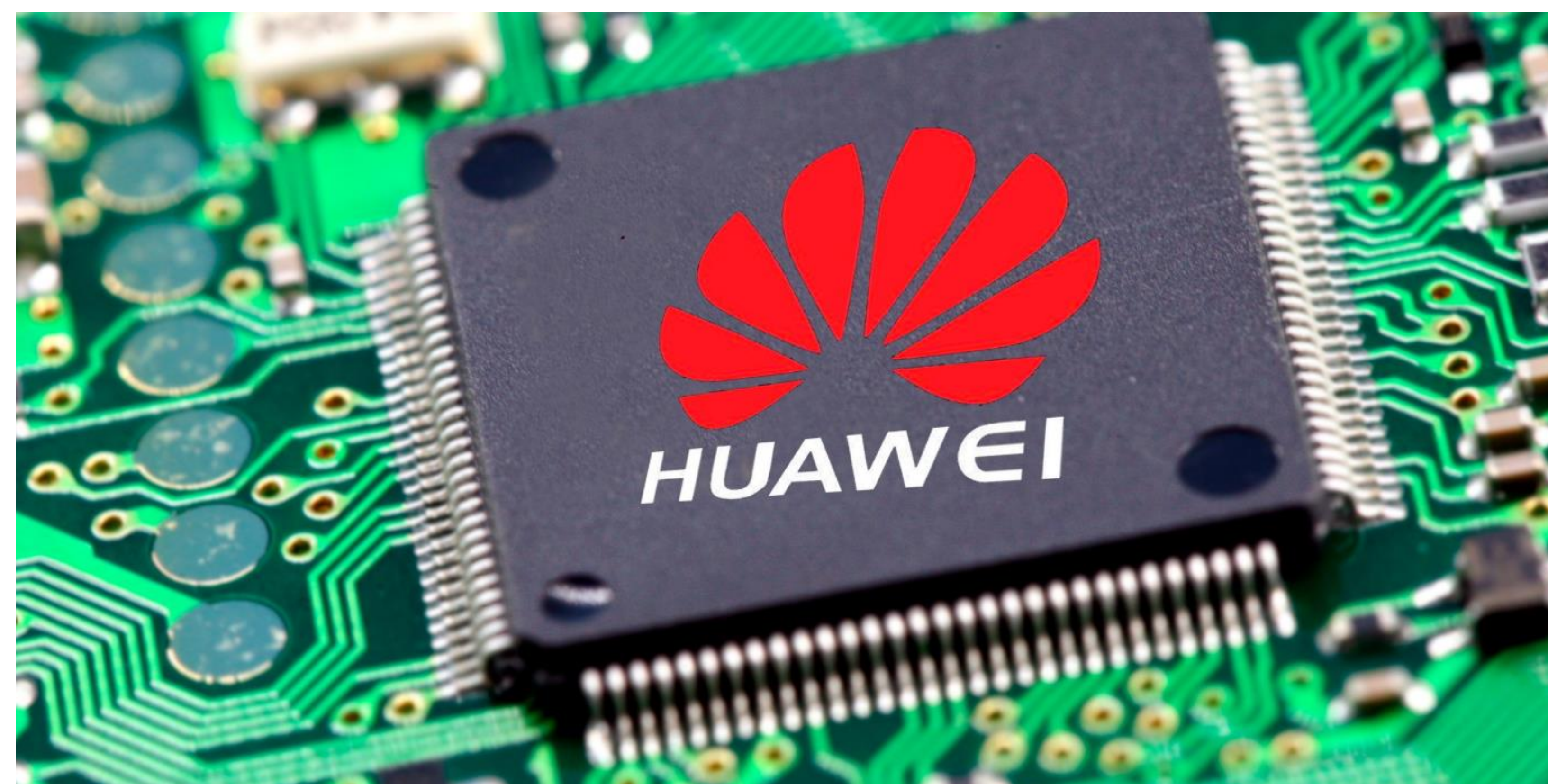
Case Study 2: Apple Silicon



Case Study 2: Apple Silicon Revealed



Case Study 3: Leading Chip Startups

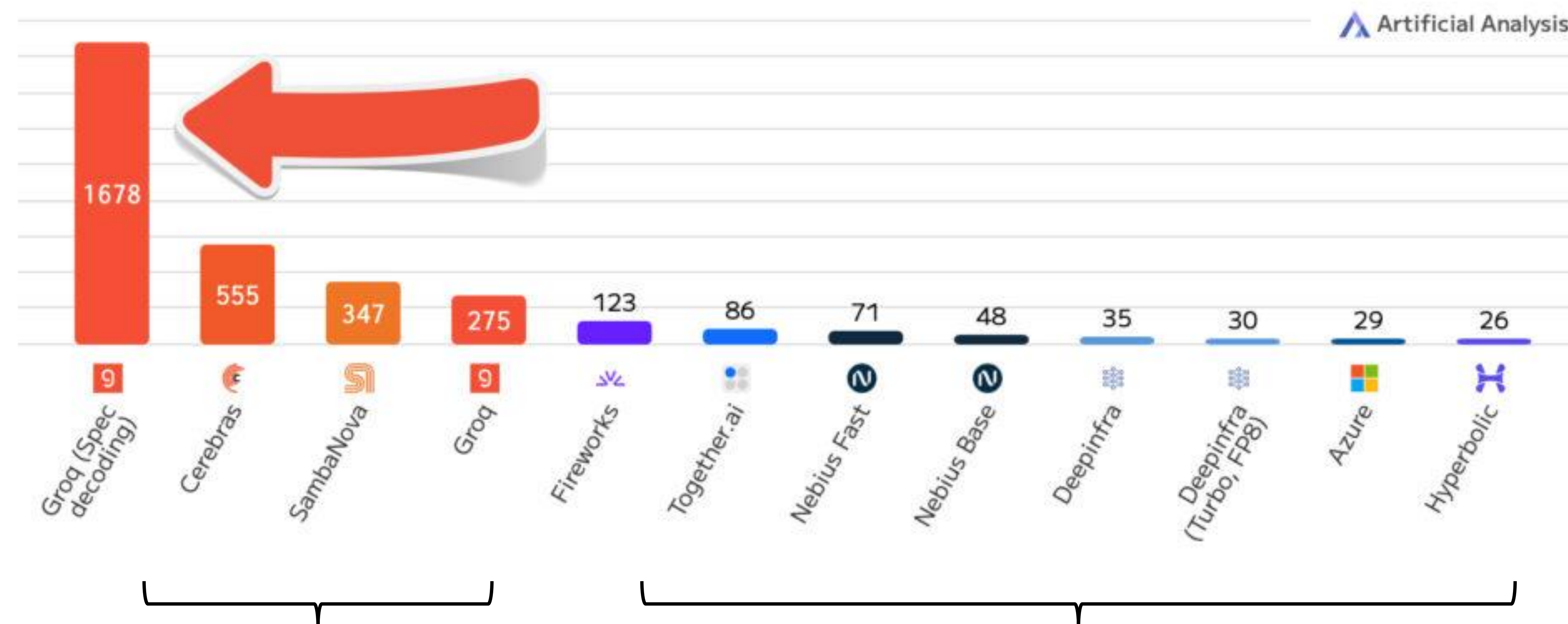


Case Study 3: Groq

Question: How did Groq achieve that?

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

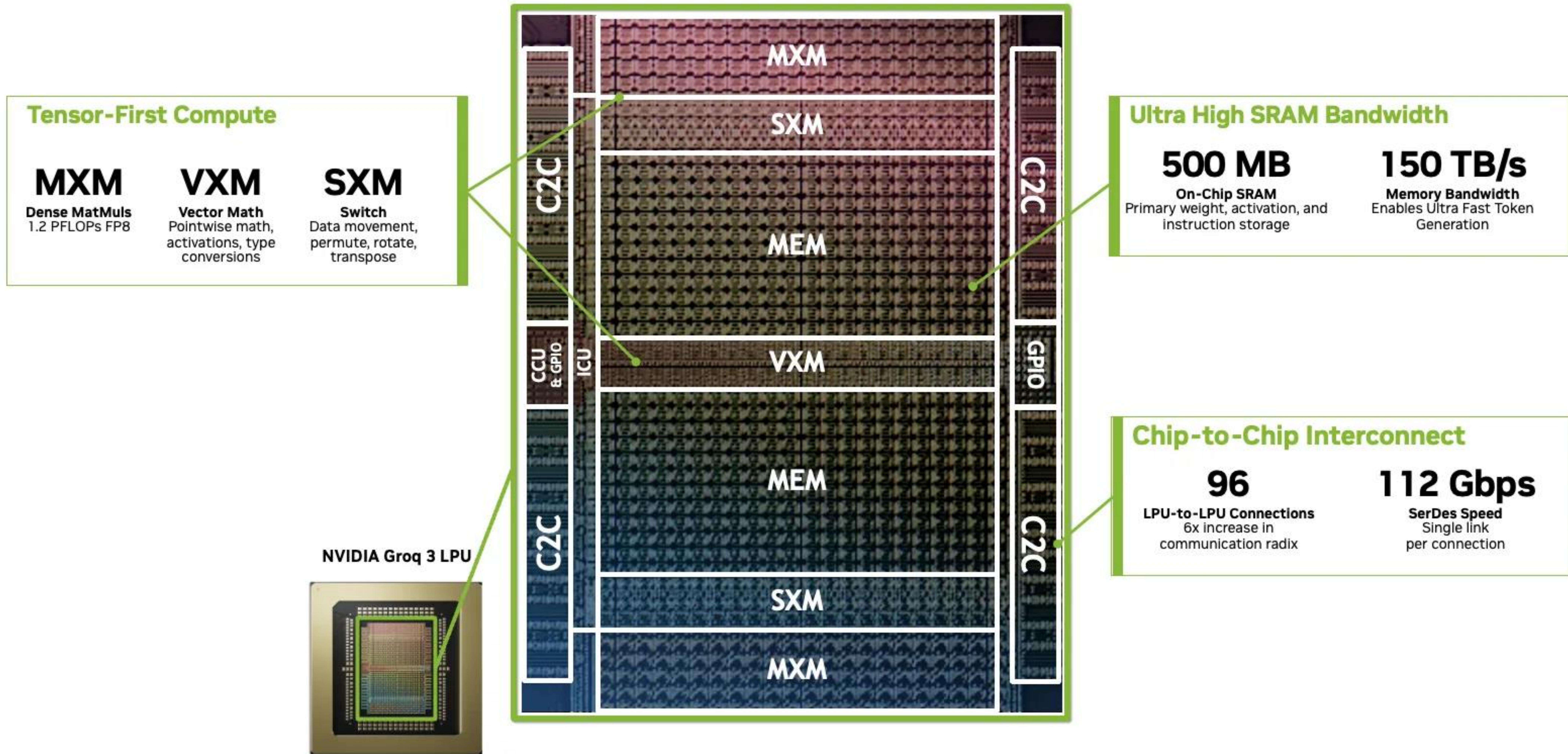
Output Tokens per Second; Higher is better



More specialized hardware

Nvidia GPUs

NVIDIA Groq 3 LPU Chip Architecture



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 Vera Rubin specification and compare it to B200

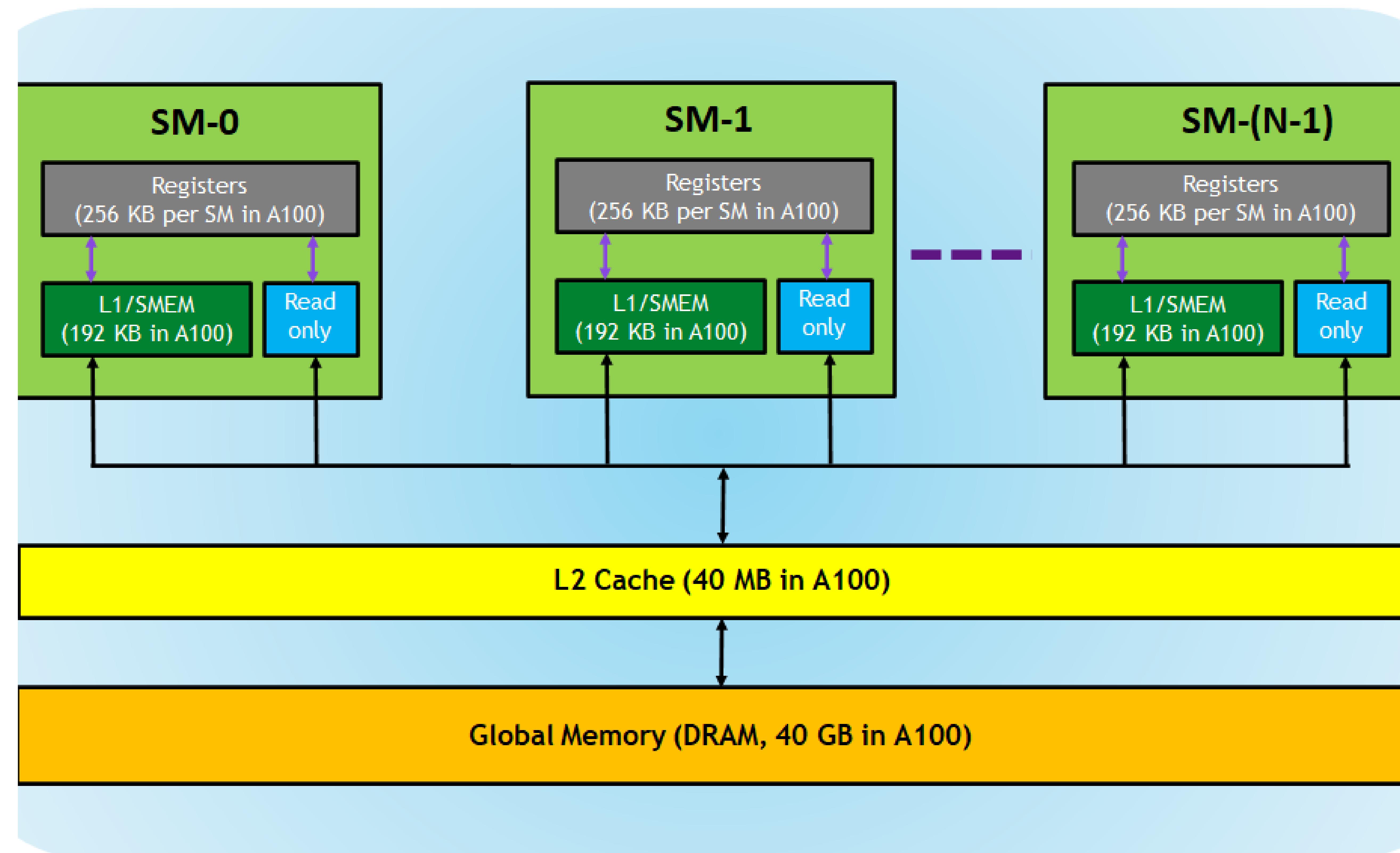
Economic Question

Question: What is Nvidia's Moat?

Today: GPU and CUDA

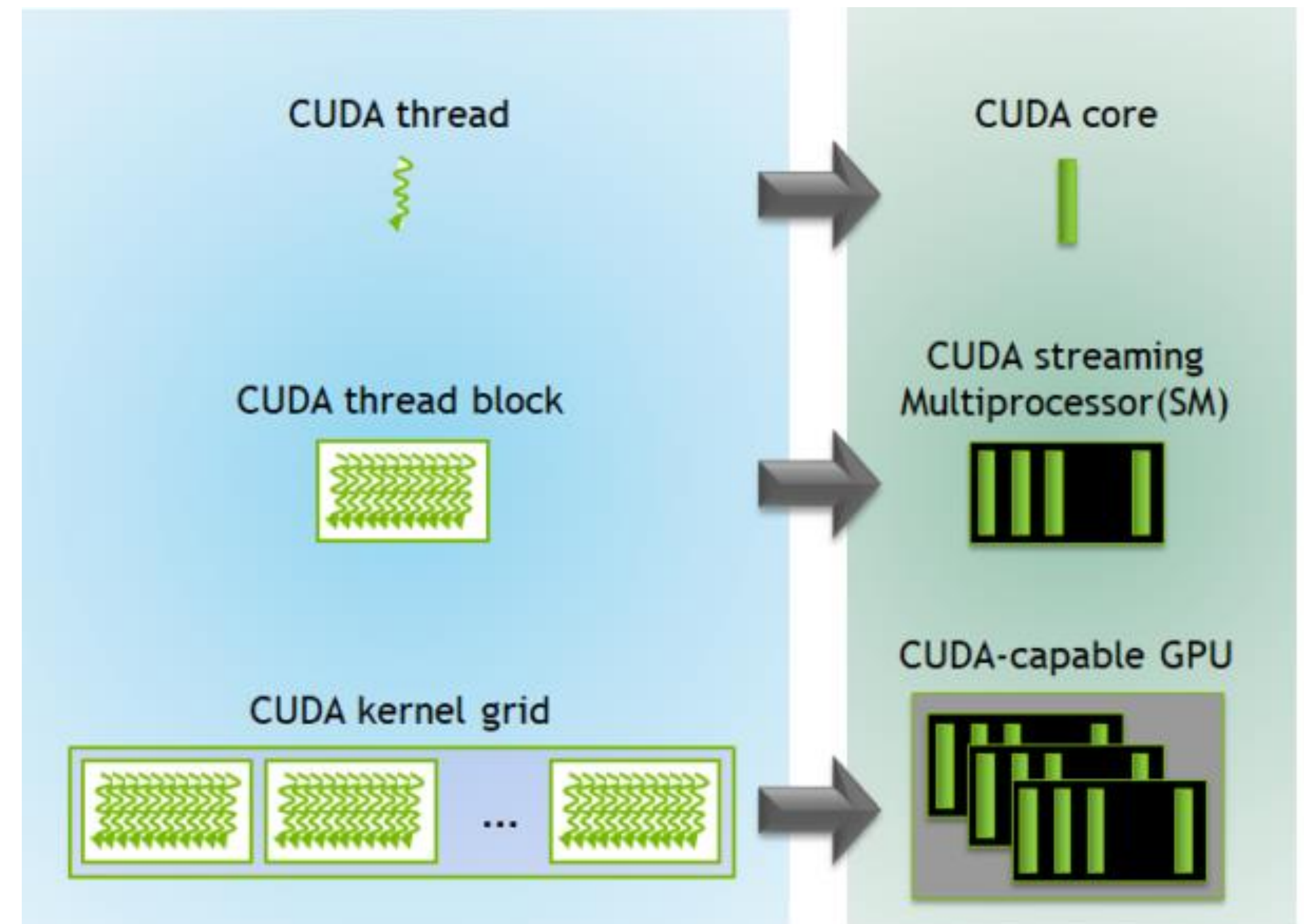
- Basic concepts in GPUs
  - Execution Model
  - Memory
- Programming abstraction

GPU Overview



Kernel, 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
- Kernel: CUDA program executed by many CUDA cores in parallel

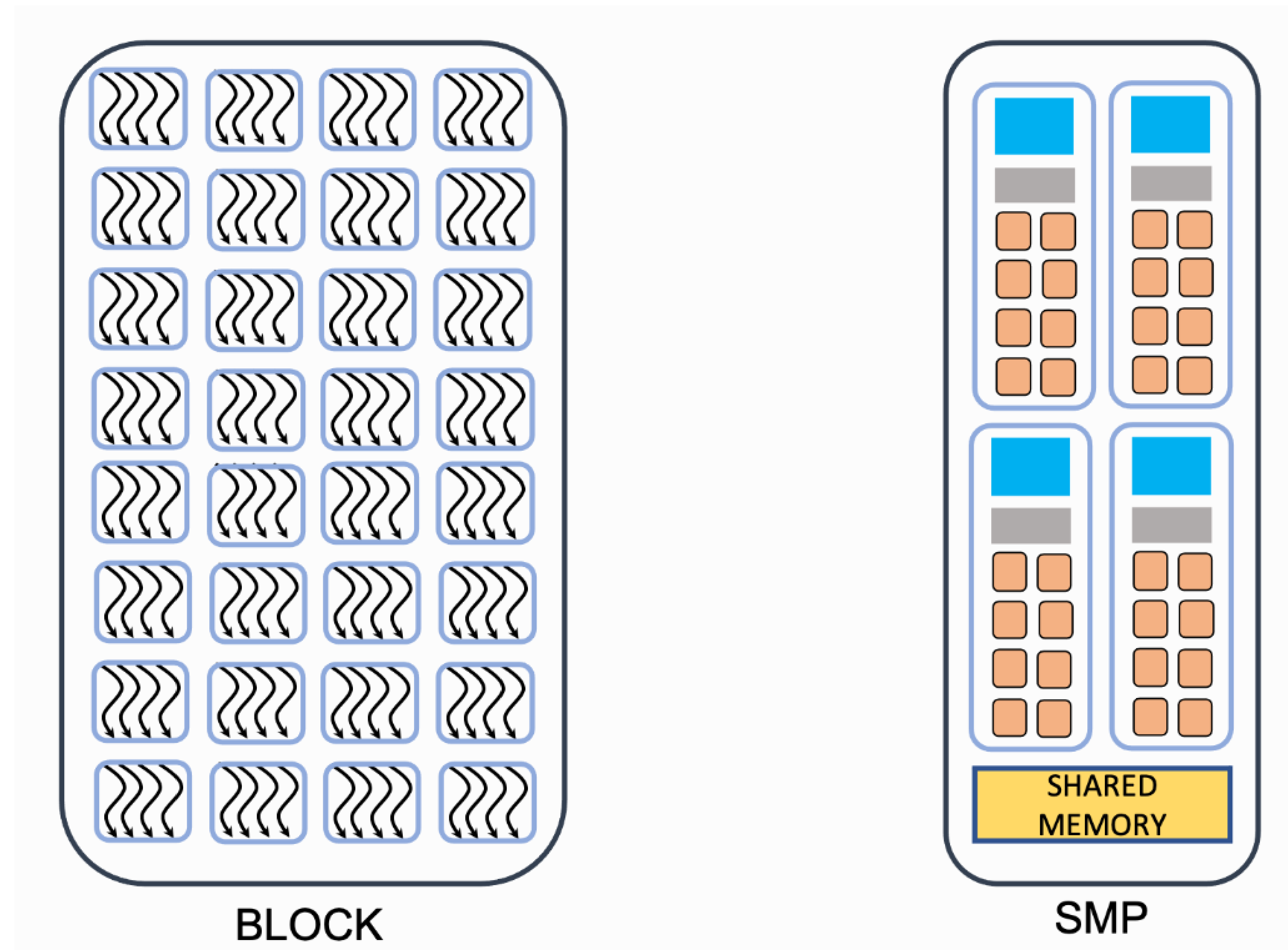


Threads

GPU/CUDA thread vs. OS thread?

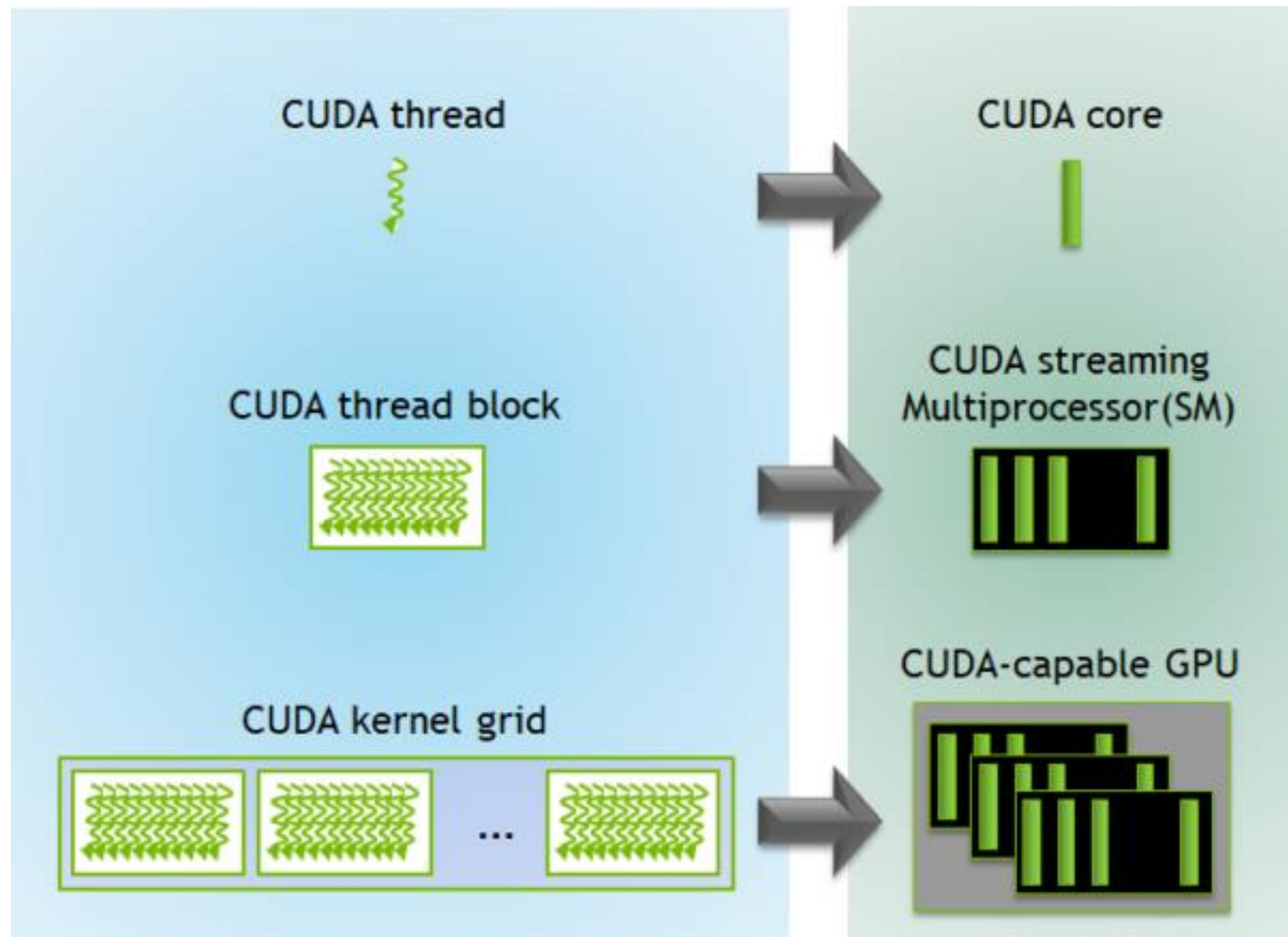
Thread Block

- A collection of many threads mapped to a streaming multiprocessor (SM/SMP)



Grid

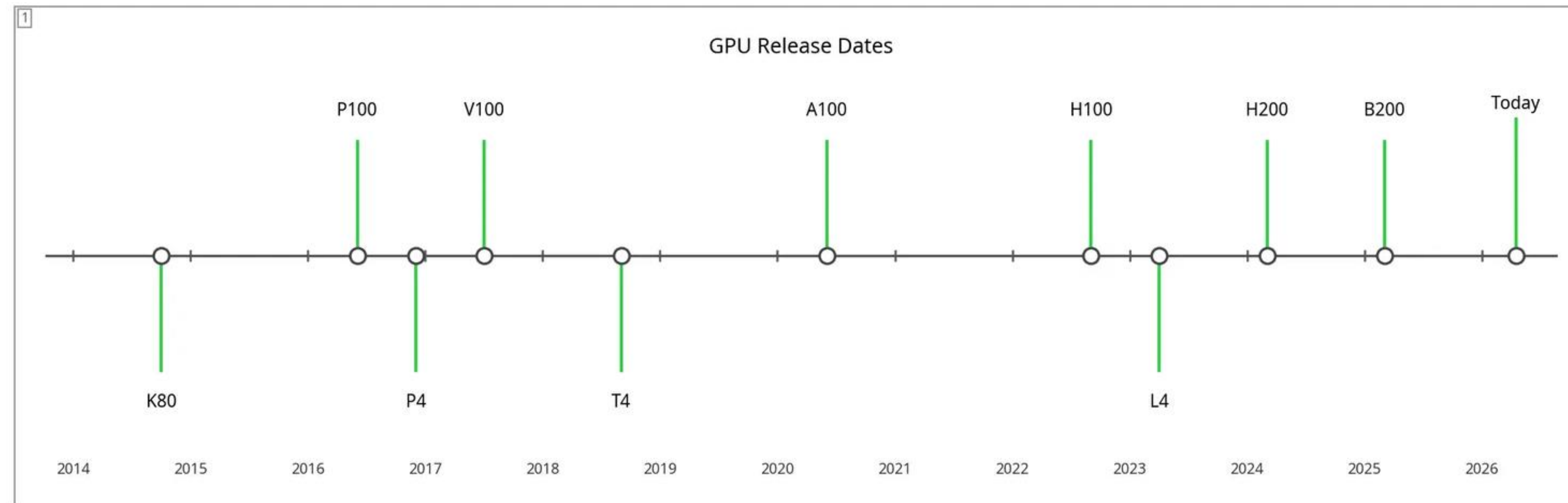
- A collection of blocks (SMs) that execute the same kernel

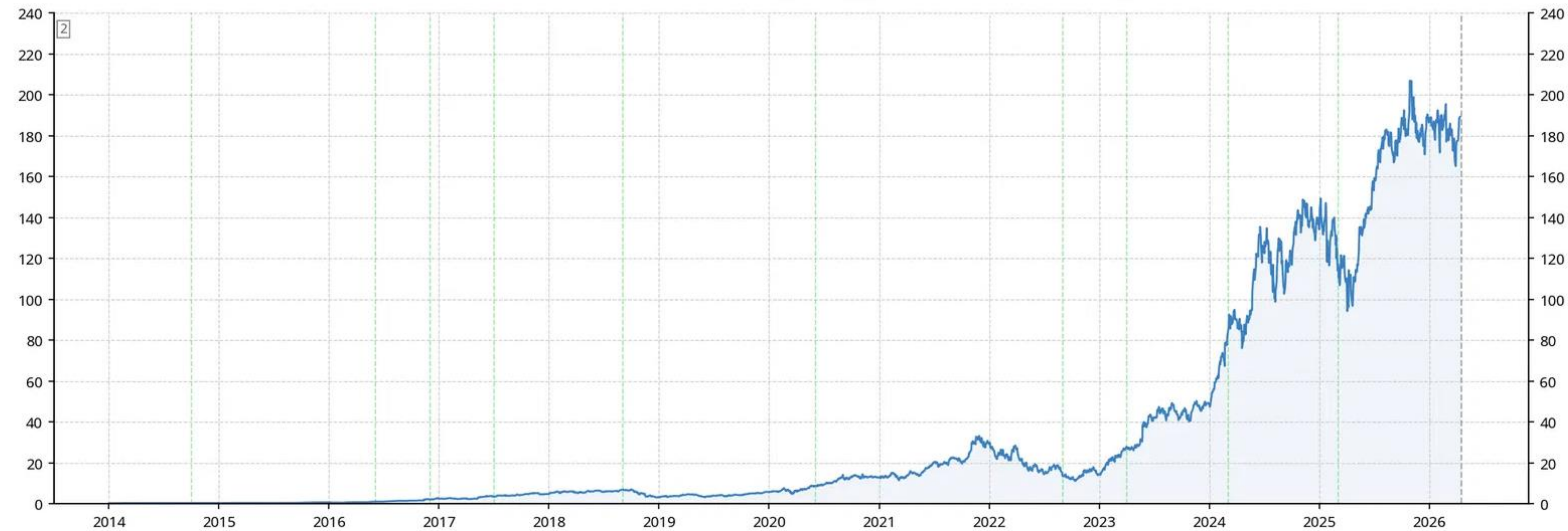


More powerful GPU means:

- More SMs
- More core/SM
- More powerful cores

Nvidia ML GPU Release Timeline





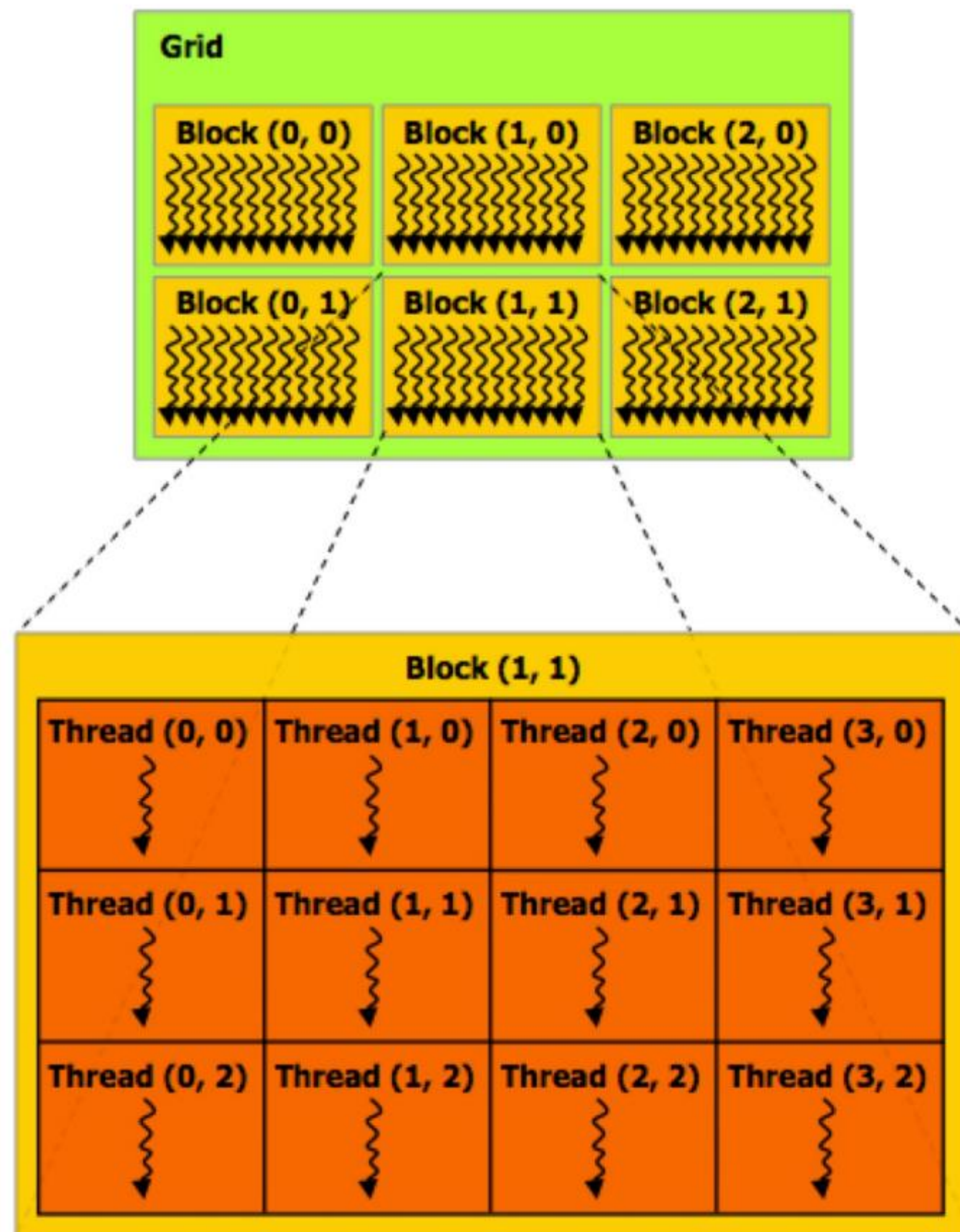
SMs/Threads on Nvidia’s GPUs and AWS on-demand Price

- V100 (2018 - Now): 80 SMs, 2048 threads/SM,
  - ~$3/hour/GPU
- A100 (2020 - Now): 108 SMs, 2048 threads/SM,
  - ~$4/hour/GPU
- H100 (2022 - Now): 144 SMs, 2048 threads/SM
  - ~$12/hour/GPU
- B100 and B200 (2025 -)
  - How many SMs? How many threads per SM?
  - How many dollars per GPU hour?

CUDA

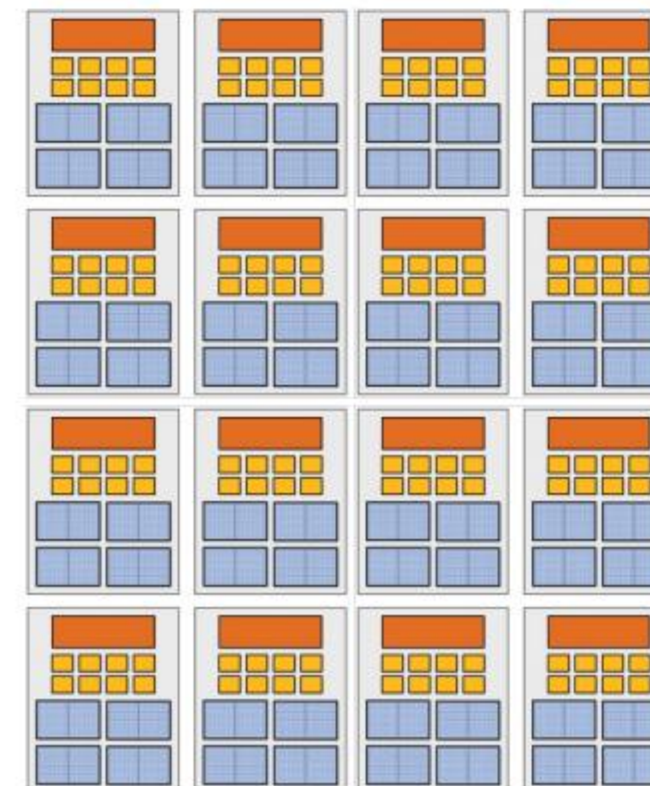
- Introduced in 2007 with NVIDIA Tesla architecture
- C-like languages for programming GPUs
- CUDA’s design matches the grid/block/thread concepts in GPUs

CUDA Programs contain A Hierarchy of Threads



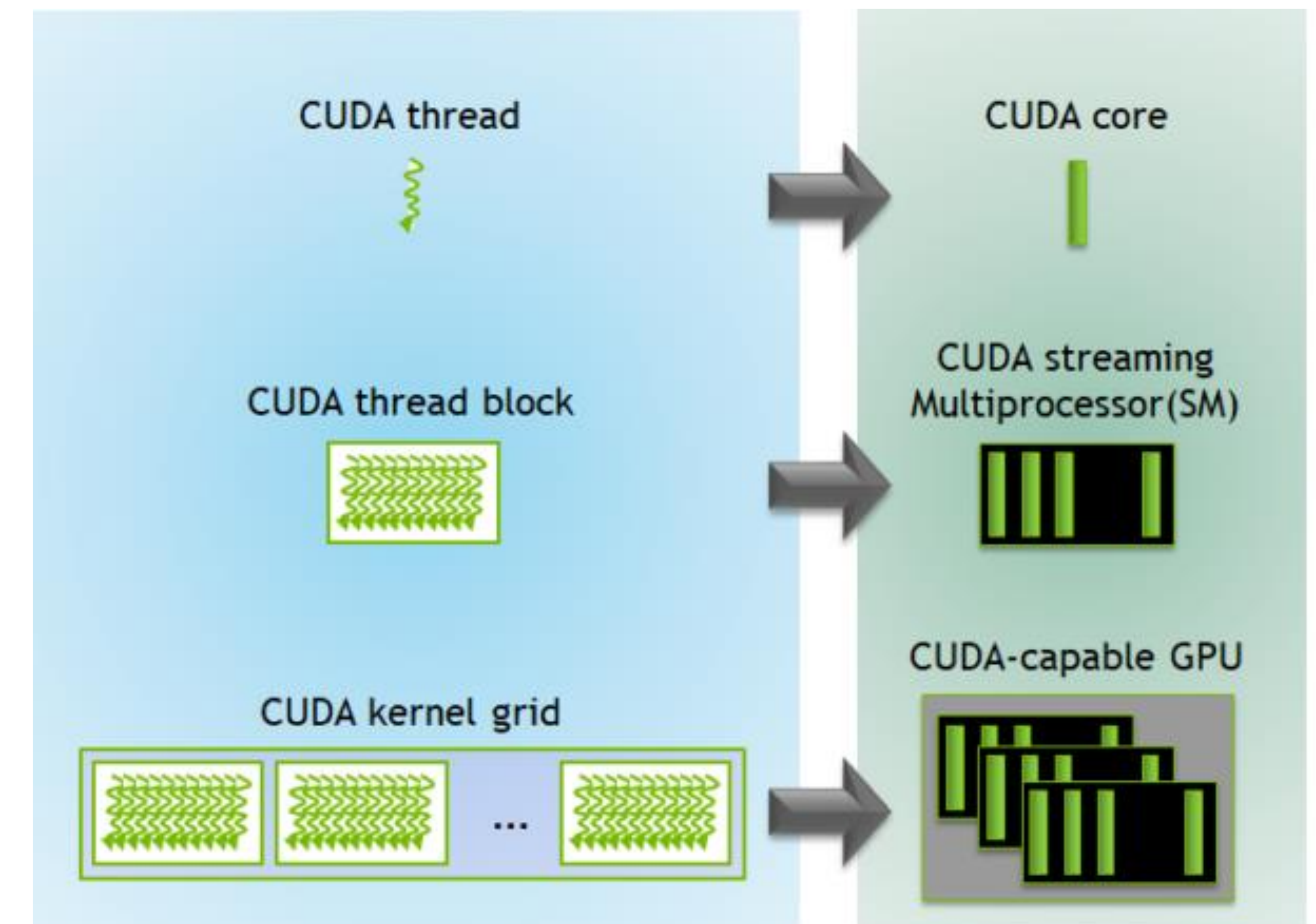

```
const int Nx = 12;
const int Ny = 6;

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?

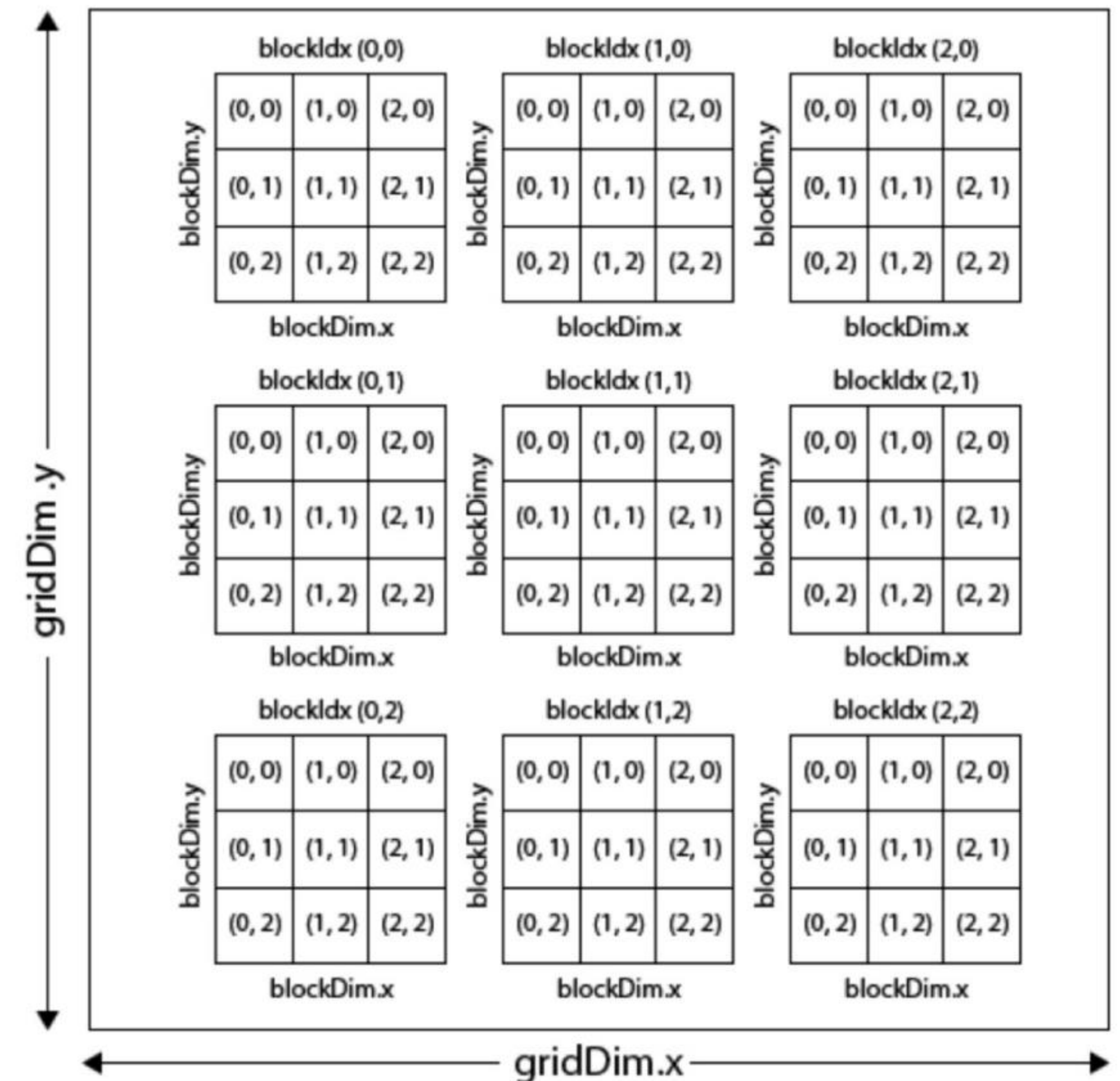
```
const int Nx = 12;
const int Ny = 6;

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: Matrix Add

```
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

Separate 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

Question

```
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]);
}
```

What happens post launching the kernel?

- Will the CPU program continue
- What if the function has return values?

#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

On potential factor many compiler (torch.compile) to require static shapes

Hence it is Important to:

- Check boundary conditions

SIMD Constraints: how to handle control flow?

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

- Why?
- Let's look at a control flow example

```
// 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;
}
```