# ECE7115 ~~Multimodal VLM~~ LLM

## 8. Understanding GPUs

Spring 2026

Namhyuk Ahn, Inha University

# Last Week: LLM Case Study

- What hasn't changed since 2023 - 2025?
  - Decoder-only architecture + autoregressive left-to-right generation
  - RoPE(though hybrid RoPE + NoPE is used)
  - SwiGLU for FFN activation
  - BPE-based tokenization
  - Pre-norm placement

# Last Week: LLM Case Study

- The winning formula in 2023-2025
  - Norm: RMSNorm + QK-Norm (mostly pre-norm except OLMo 2)
  - Attention: GQA with QK-Norm or MLA (+DSA for DeepSeek)
    - Hybrid linear attentions

  - FFN: SwiGLU - unchanged since LLaMA
  - PE: RoPE is dominant, but NoPE is emerging as alternatives
  - MoE: Fine-grained

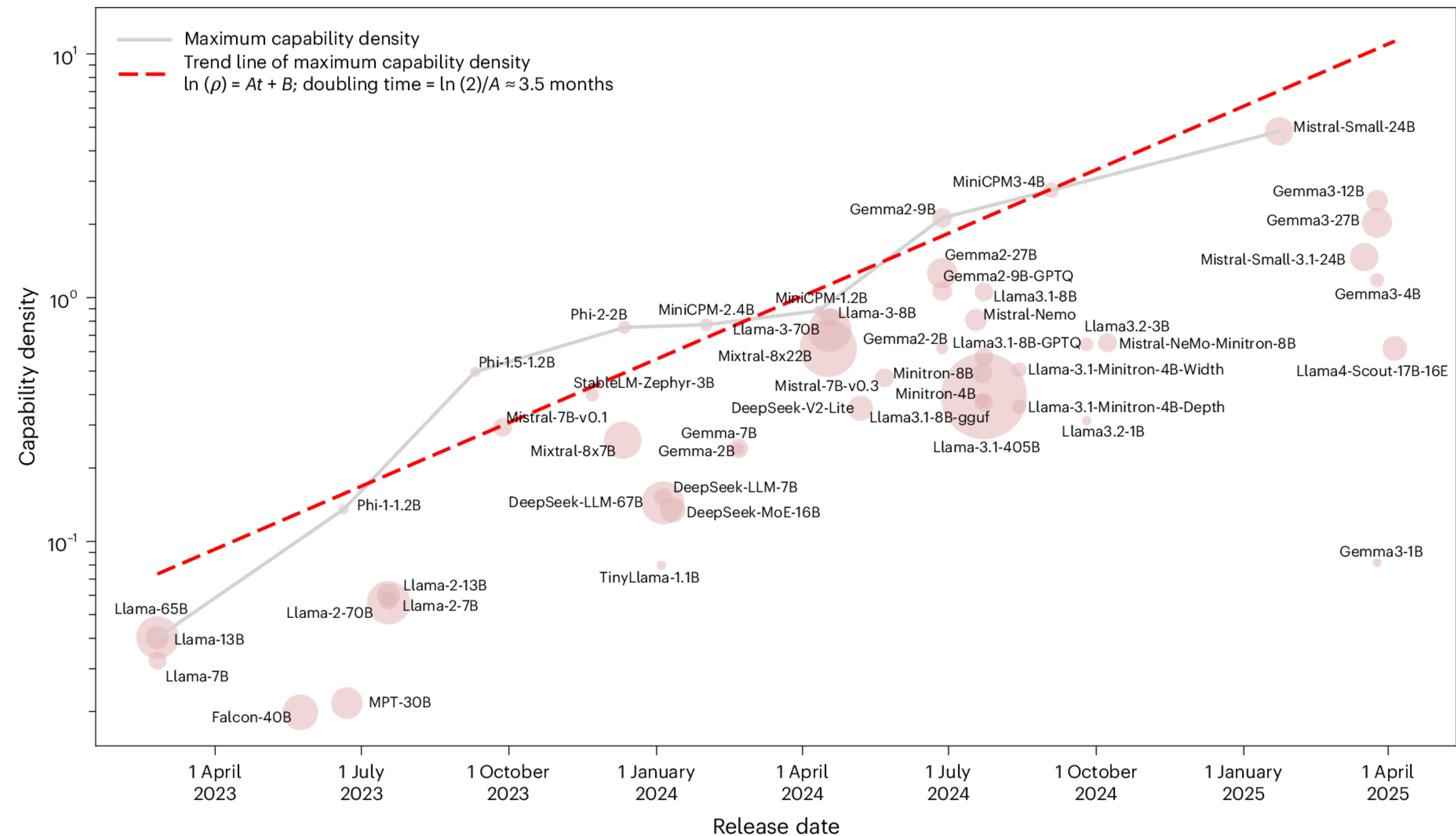
  - Training: WSD or cosine decay, MTP, FP8
  - Optimizer: Still AdamW but Muon family emerging

# Last Week: LLM Case Study

| Component | 2023 | 2024 | 2025 |
|---|---|---|---|
| Attention | MHA → GQA | + MLA, Linear Attn. | GQA(standard), MLA+DSA, Linear hybrids |
| Pos Encoding | RoPE | RoPE | RoPE or RoPE + NoPE |
| MoE Experts | 8 (Mixtral) | 64 → 256 | 128 → 384 |
| MoE Routing | Aux loss | + Aux-loss-free | Both used |
| Stabilization | — | Logit Soft-Cap | → QK-Norm |
| Optimizer | AdamW | AdamW + WSD | Still AdamW but Muon? |
| Data Scale | 1-2T | up to 18T | up to 36T |

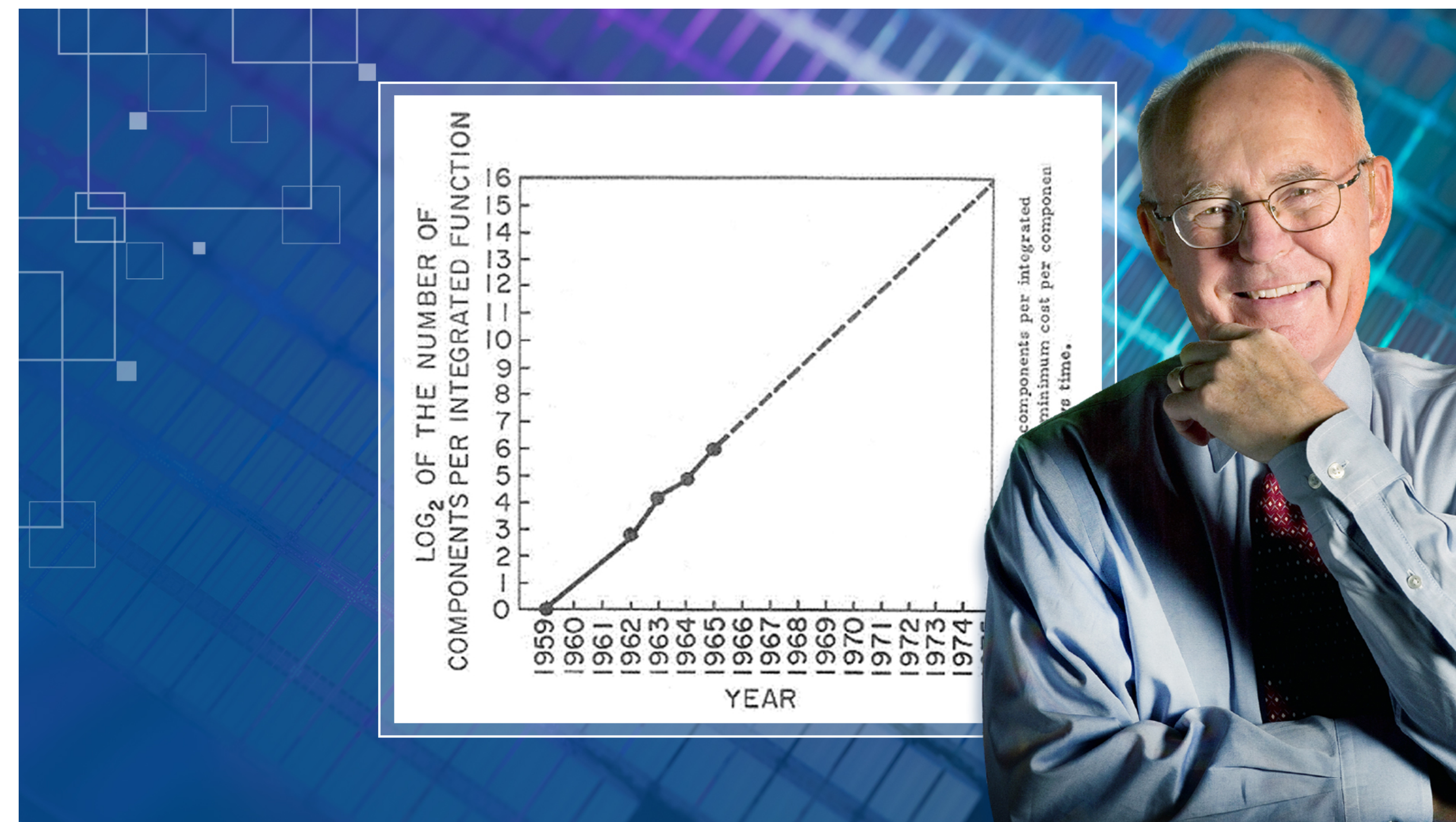
# What We've Learnt...

- Pre-training gigantic Transformers on a web-scale dataset
- Oftentimes, compute leads to predictable performance gains for LLMs
- Faster hardware, improved parallelization can drive progress



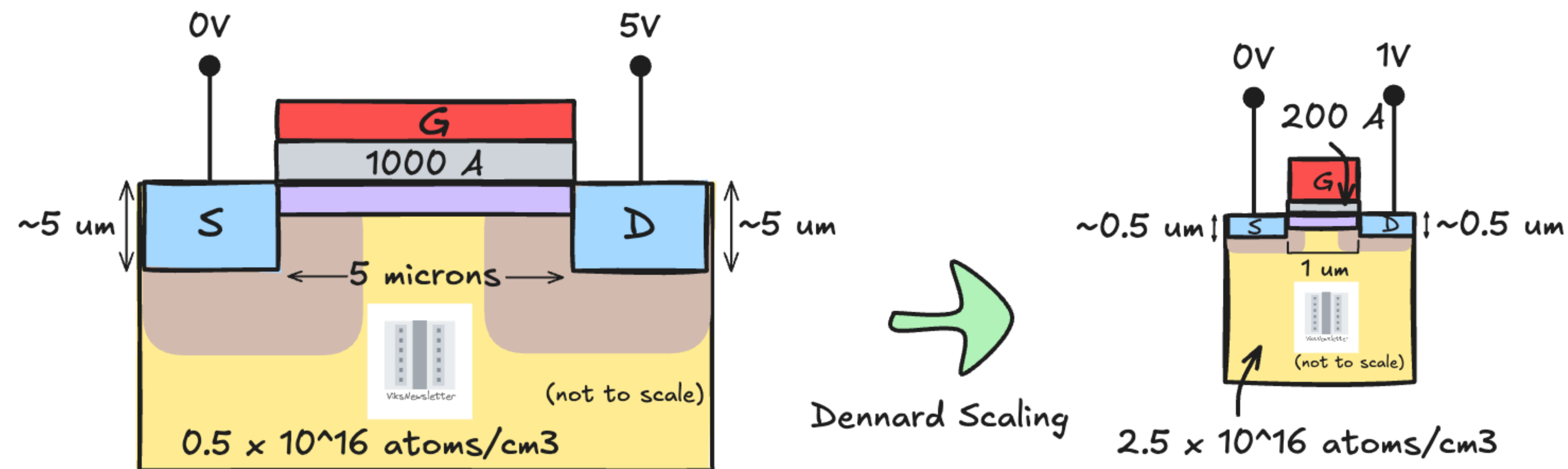
# Earlier Compute Scaling

- Moore's law
  - The observation that transistor density doubles every ~2 years
  - Exponential growth in compute capacity and cost-efficiency
  - This provided a predictable roadmap for hardware scaling and exponential performance gains for decades



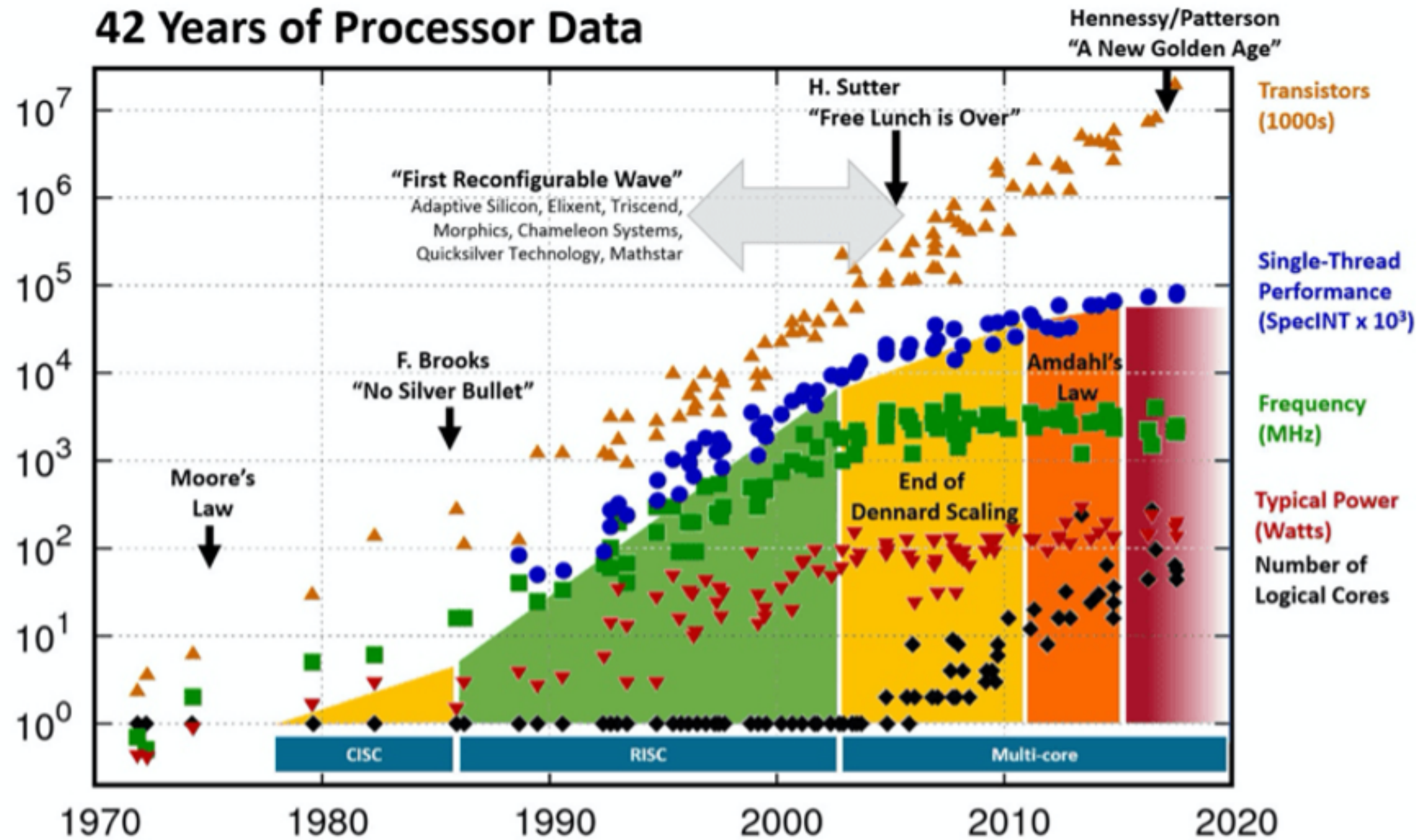
# Earlier Compute Scaling

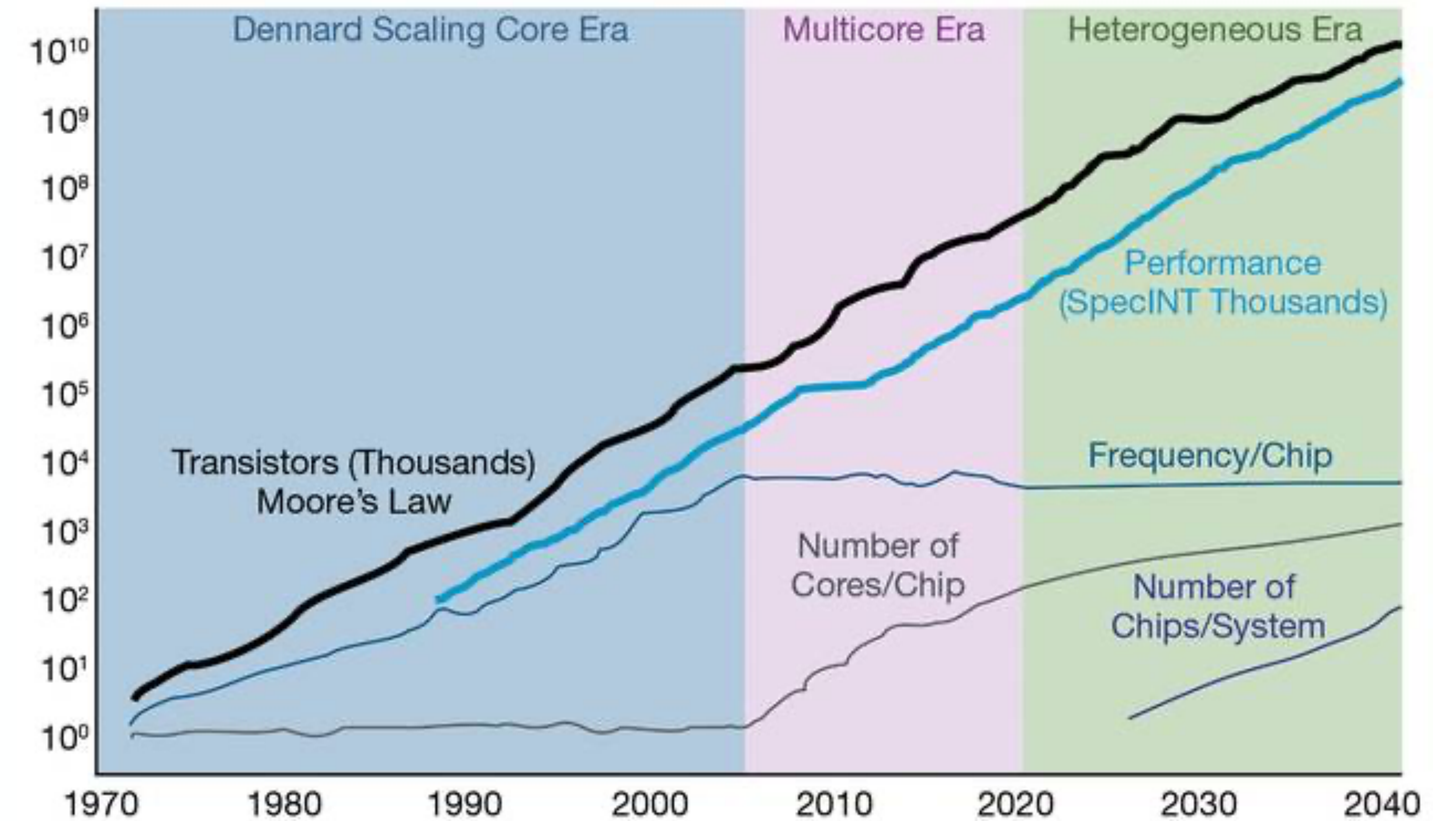
- Dennard Scaling (MOSFET Scaling)
  - Power density remains constant as transistors shrink
  - It means that we can get higher clock frequencies without increasing total power consumption




  - Problem: It ended around 2006 due to leakage current ("power wall")

# MultiCore / Parallel Scaling

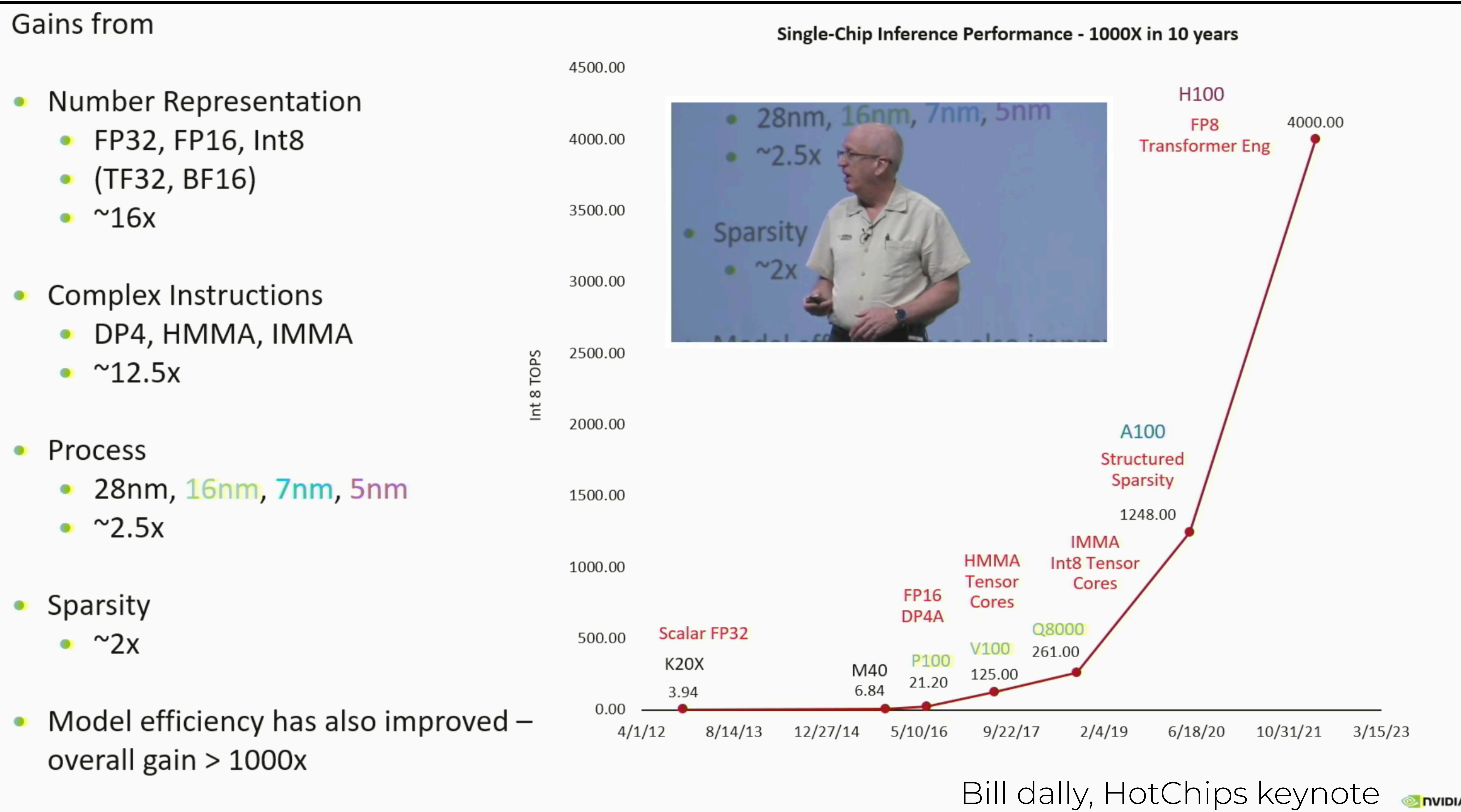


# Parallel Scaling With GPUs

Parallel scaling with GPUs has scaled > 1000x in 10 years

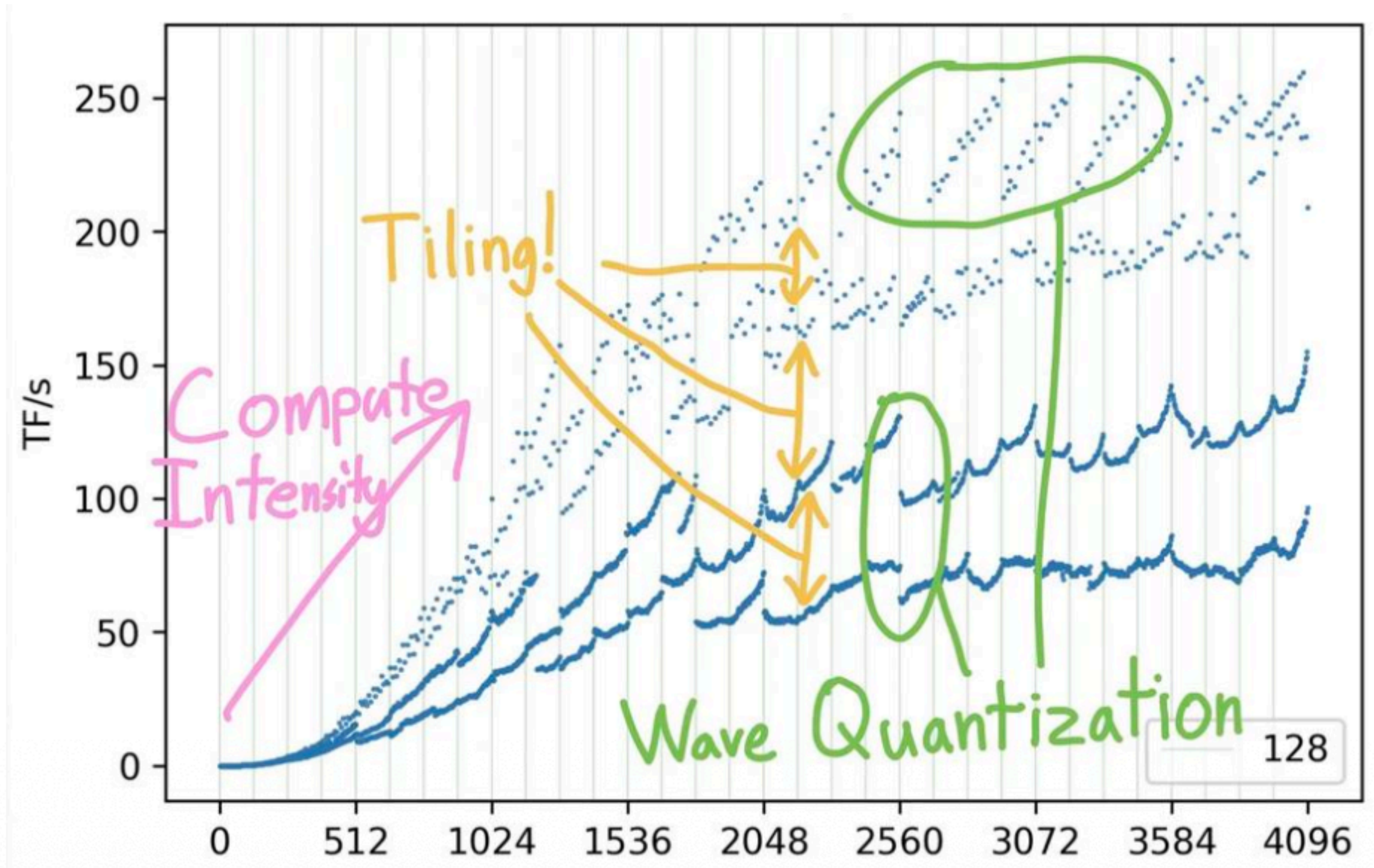
There is no LLM scaling without GPU scaling



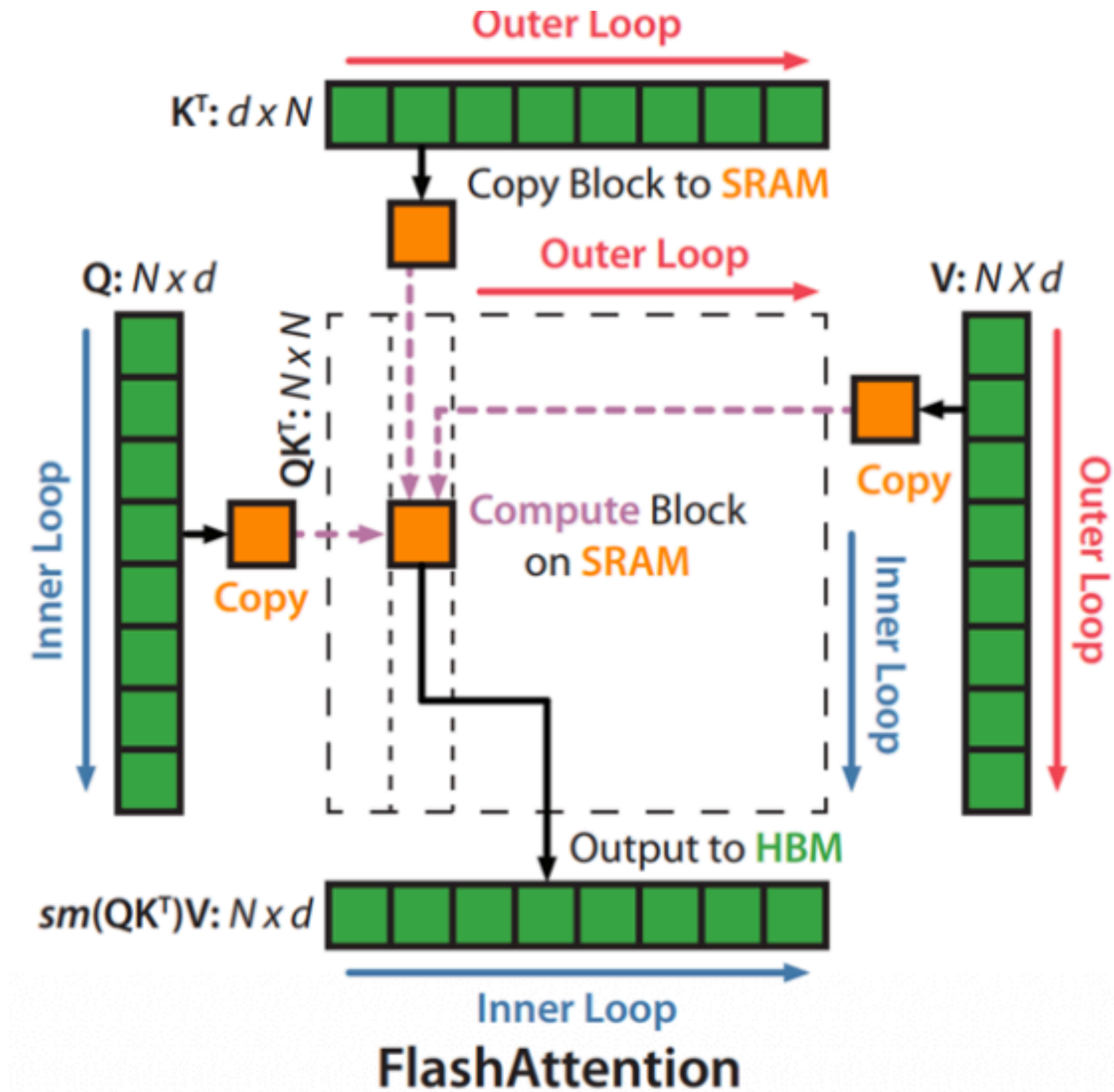
# Lecture Goal

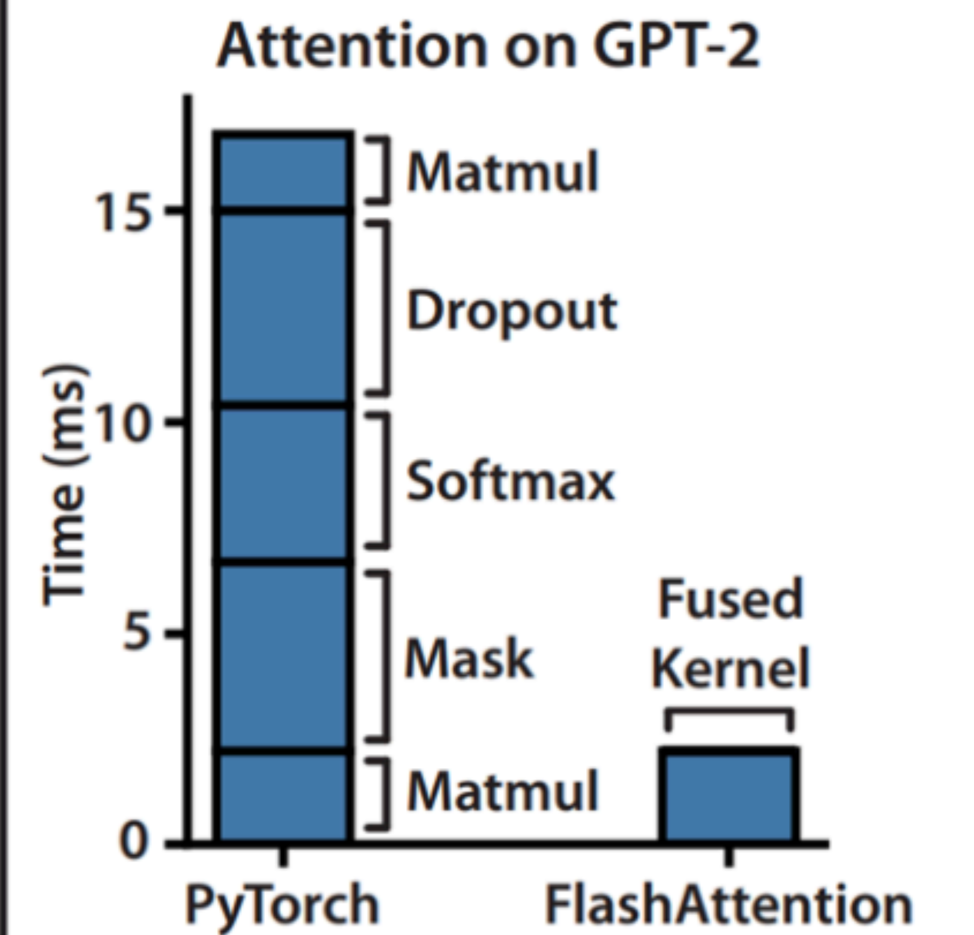
- Understand how GPUs work, not treating this as a magical thing
- Then, we will learn multi-GPU system (parallelism) in next week

Understand when GPUs get slow




Understand how to make fast algorithms



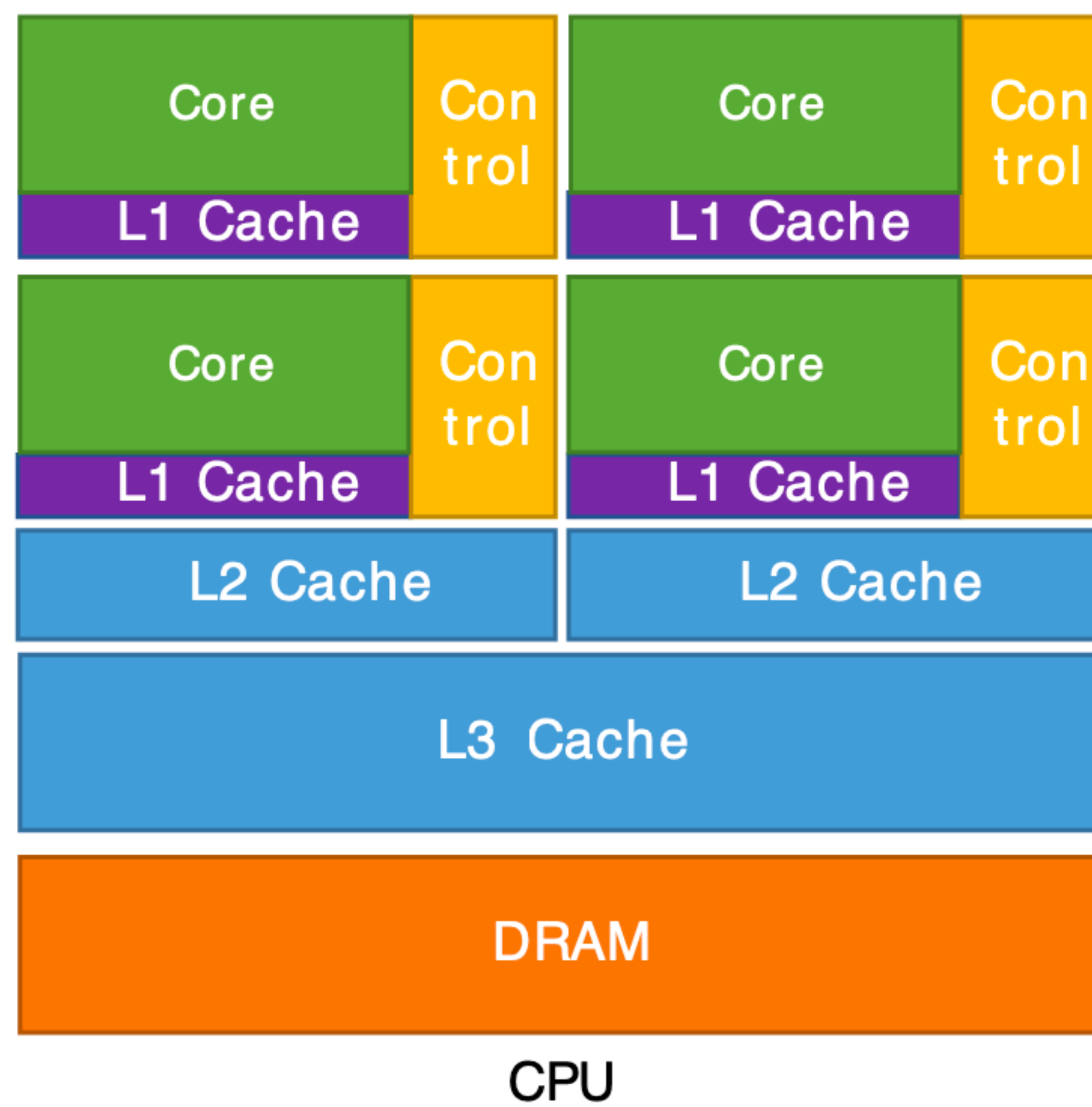
# Lecture Overview

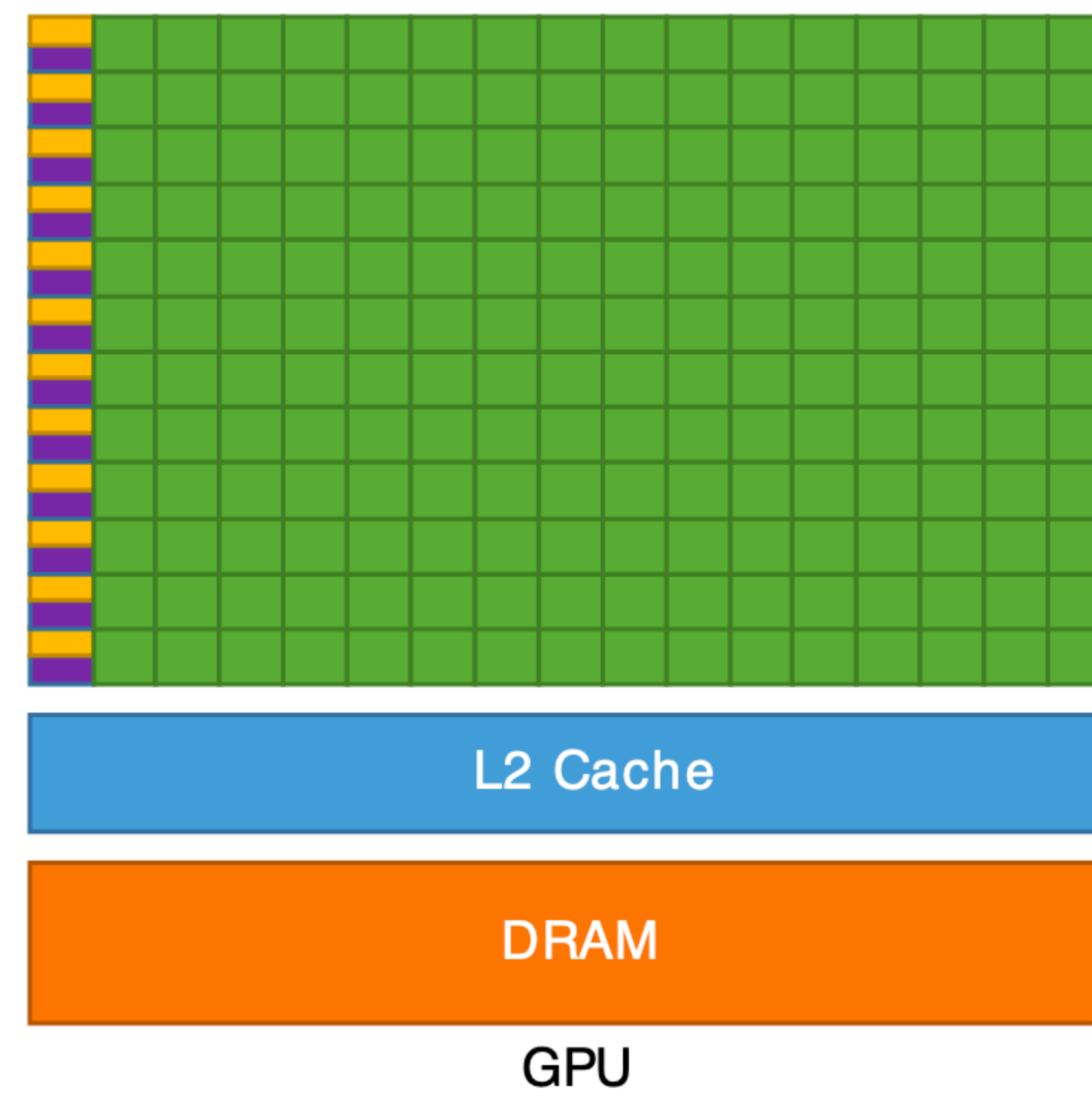
- Introduction to GPUs – how they work and important parts
- Understanding GPU performance
- Putting it together – unpacking FlashAttention

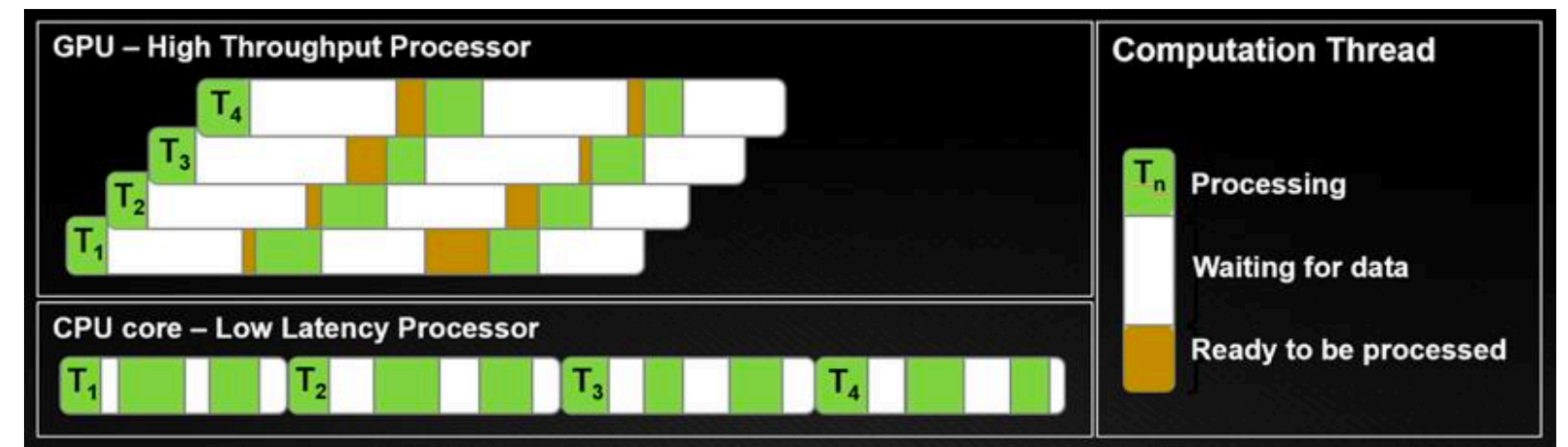
# Introduction to GPUs

# How is a GPU different from a CPU?

- CPUs optimize for a few, fast threads, GPUs optimize for many threads



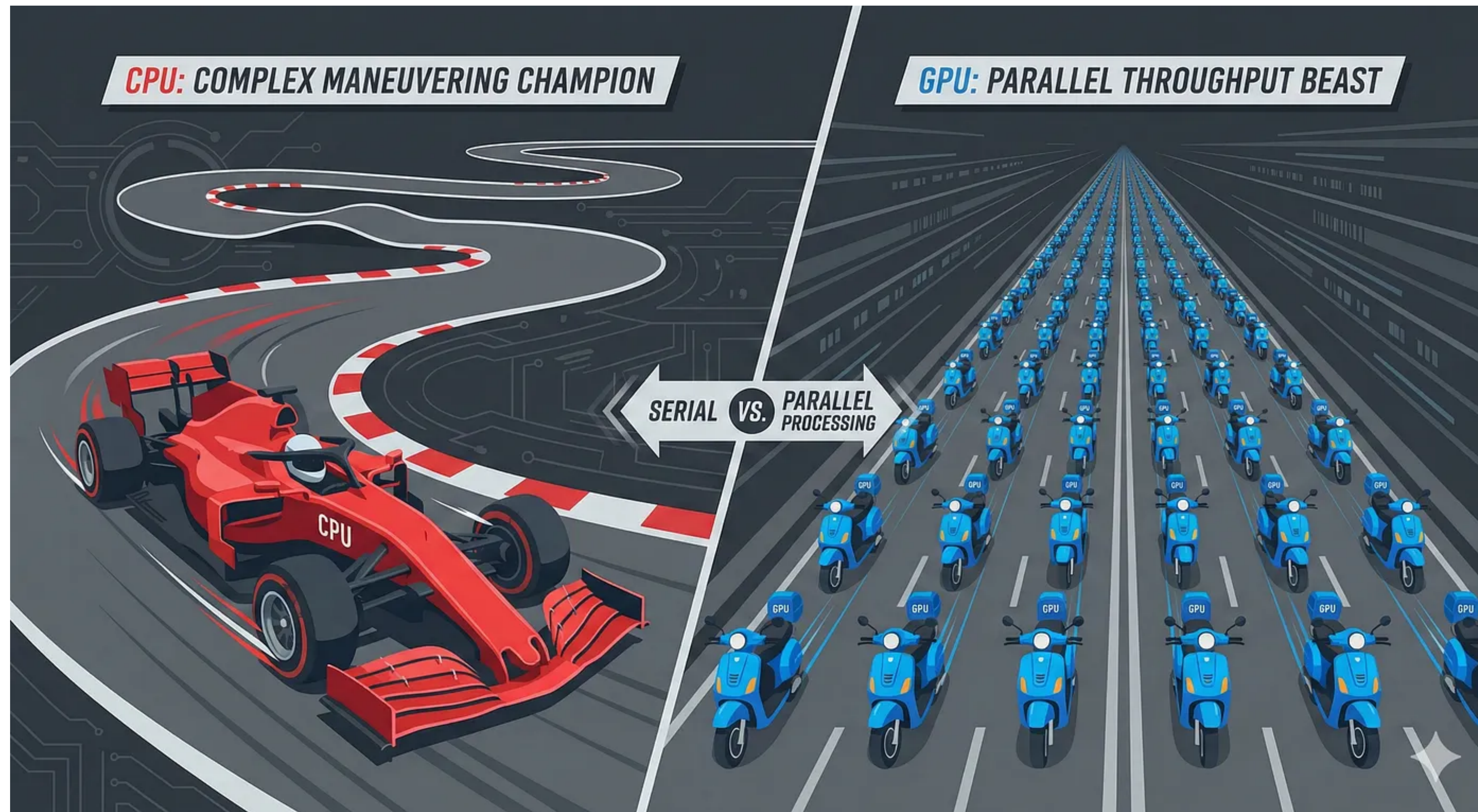




Many tiny compute units (ALUs or core)
Much less support for branching (control, cache)

CPUs optimize for latency
(each thread finishes quickly)
GPUs optimize for throughput
(total processed data)

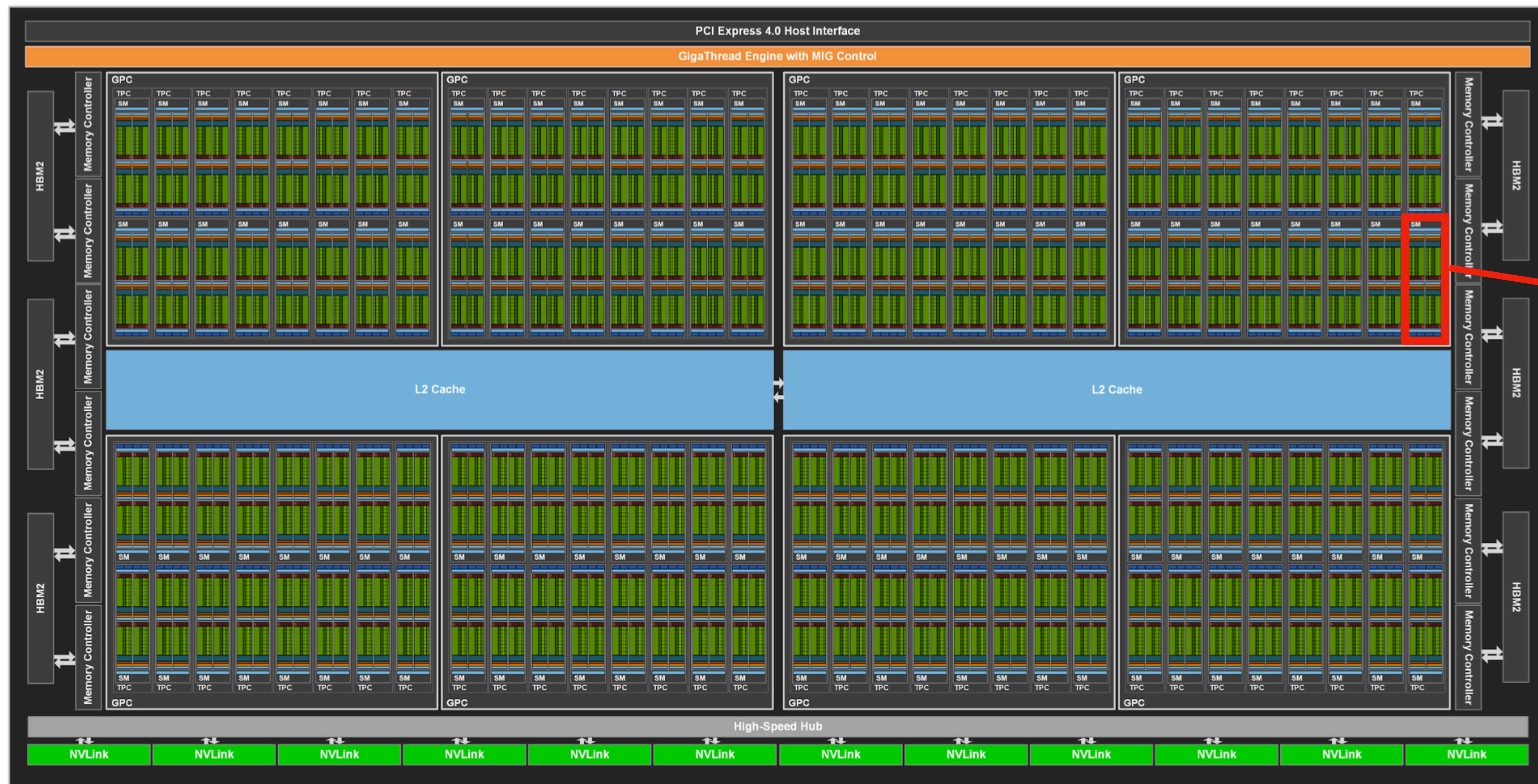
# How is a GPU different from a CPU?

- CPUs optimize for a few, fast threads, GPUs optimize for many threads



# Anatomy of a GPU (Execution Units)

GA100 Full GPU with 128 SMs




SM

GPUs have many SM (streaming multiprocessors) that independently execute 'blocks' (jobs)

Each SM further contains many SPs (streaming processor) that can execute 'threads' in parallel

# Anatomy of a GPU (Memory)

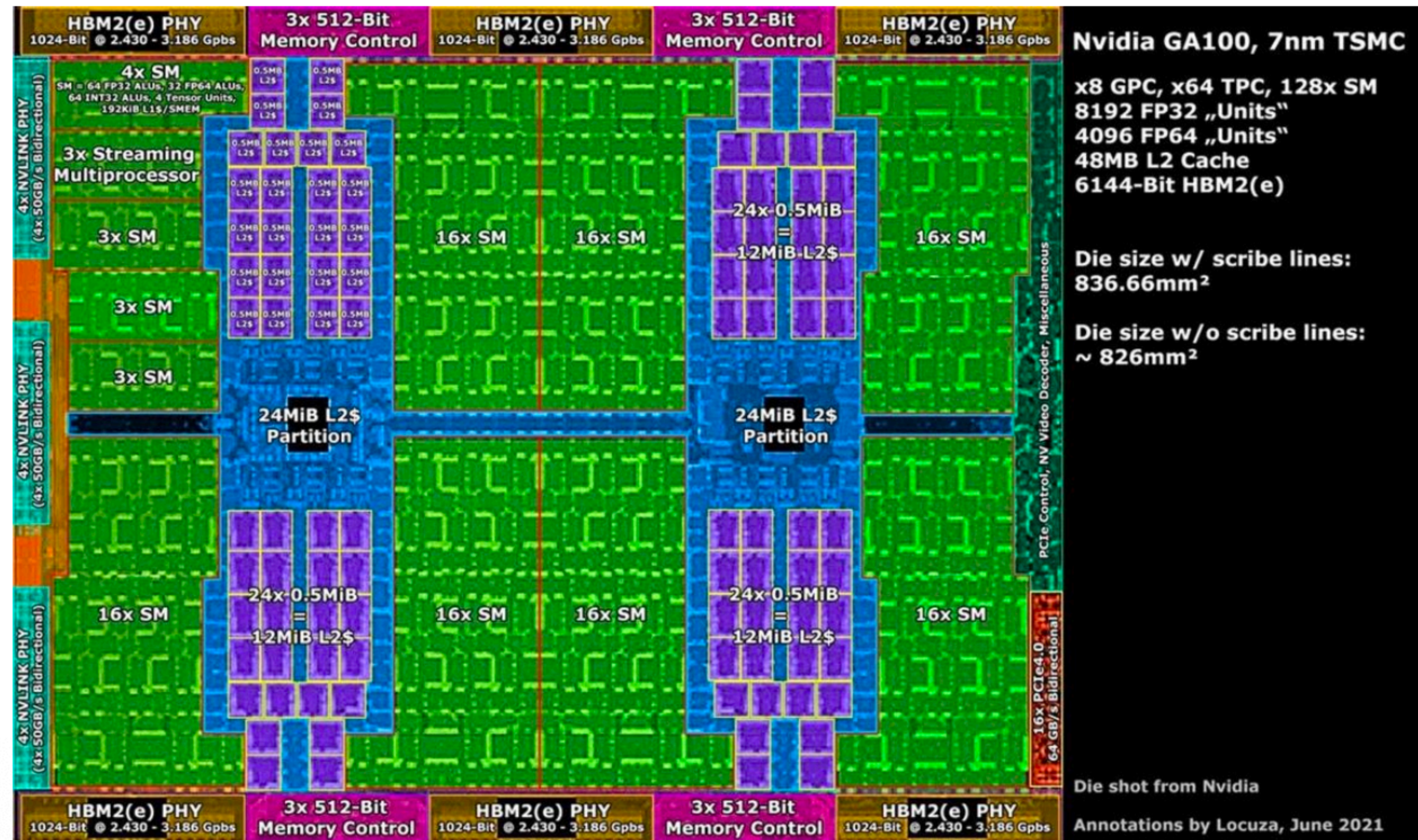
- L1 cache and shared memory is inside the SM
- L2 is on die, and global memory are the memory chips next to the GPU
- ❗❗❗: The closer the memory to the SM, the faster it is

TABLE IV
THE MEMORY ACCESSES LATENCIES

| Memory type | *CPI (cycles)* |
|---|---|
| Global memory | 290 |
| L2 cache | 200 |
| L1 cache | 33 |
| Shared Memory (ld/st) | (23/19) |

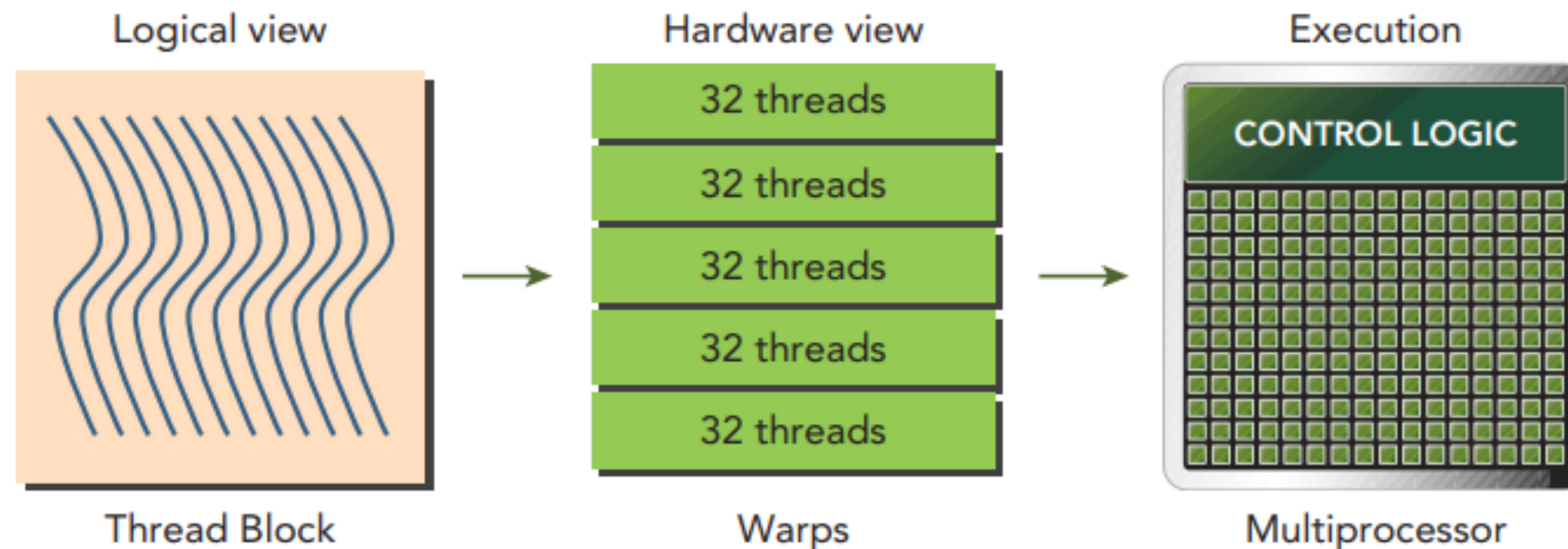
For example (A100),

- DRAM: 80GB (big, slow)
- L2: 40MB
- L1: 192KB per SM (small, fast)



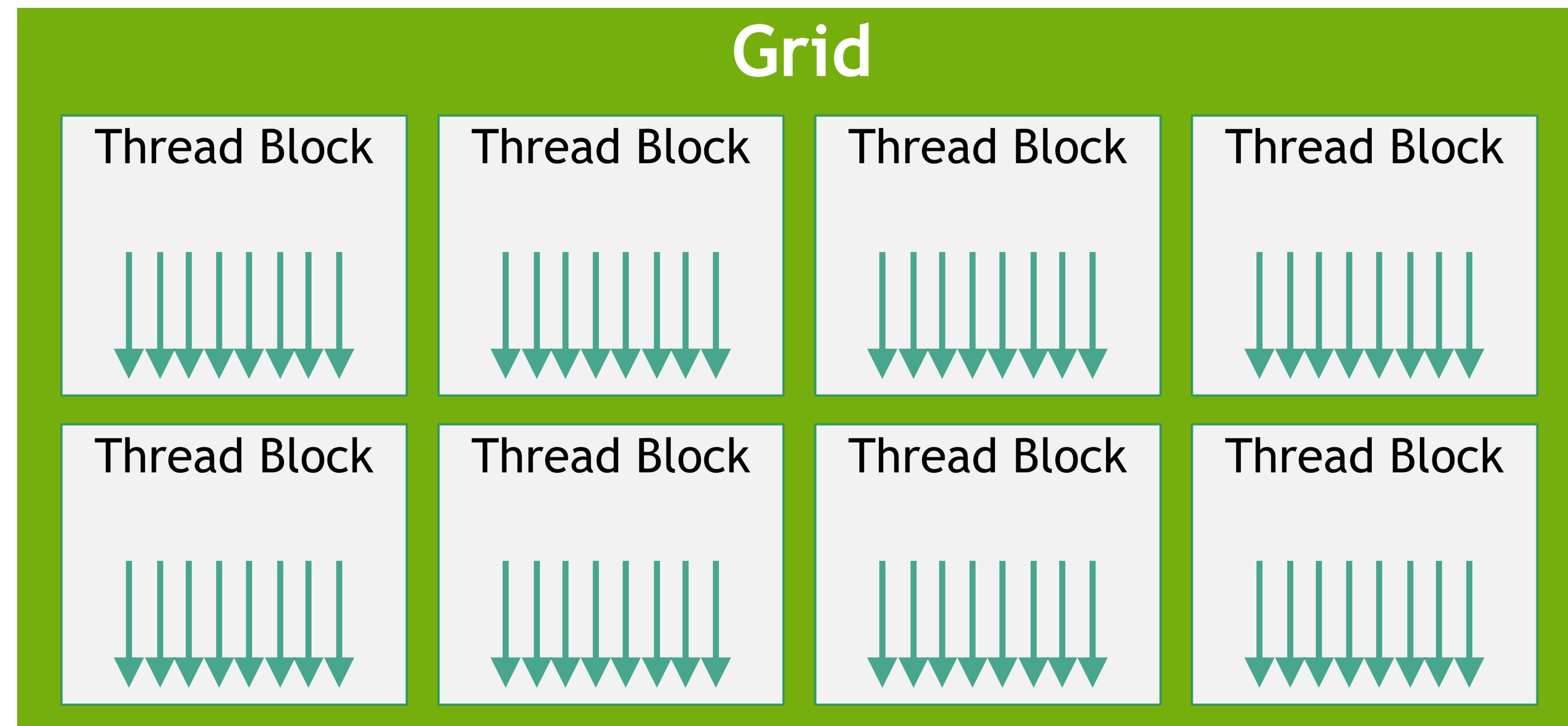
# GPU Execution Model

- There are three important players in the execution model
  - Threads: The fundamental execution units
  - Thread blocks: Group of threads
  - Warp: Threads always execute in a warp of 32 threads each



# GPU Execution Model

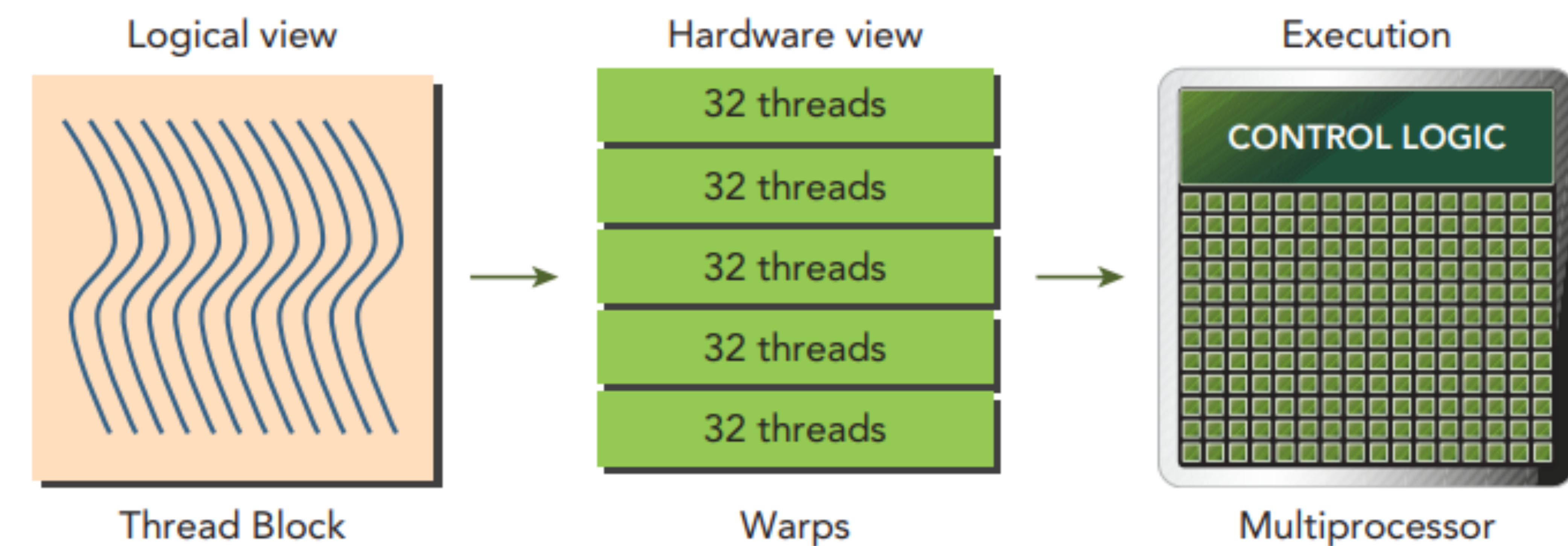
- Threads: The fundamental execution units in a GPU
  - Each thread computes a small part of the task independently
- Thread blocks: A logical group of threads assigned to the same task
  - Threads within the same block can communicate efficiently using fast on-chip shared memory and synchronize their execution



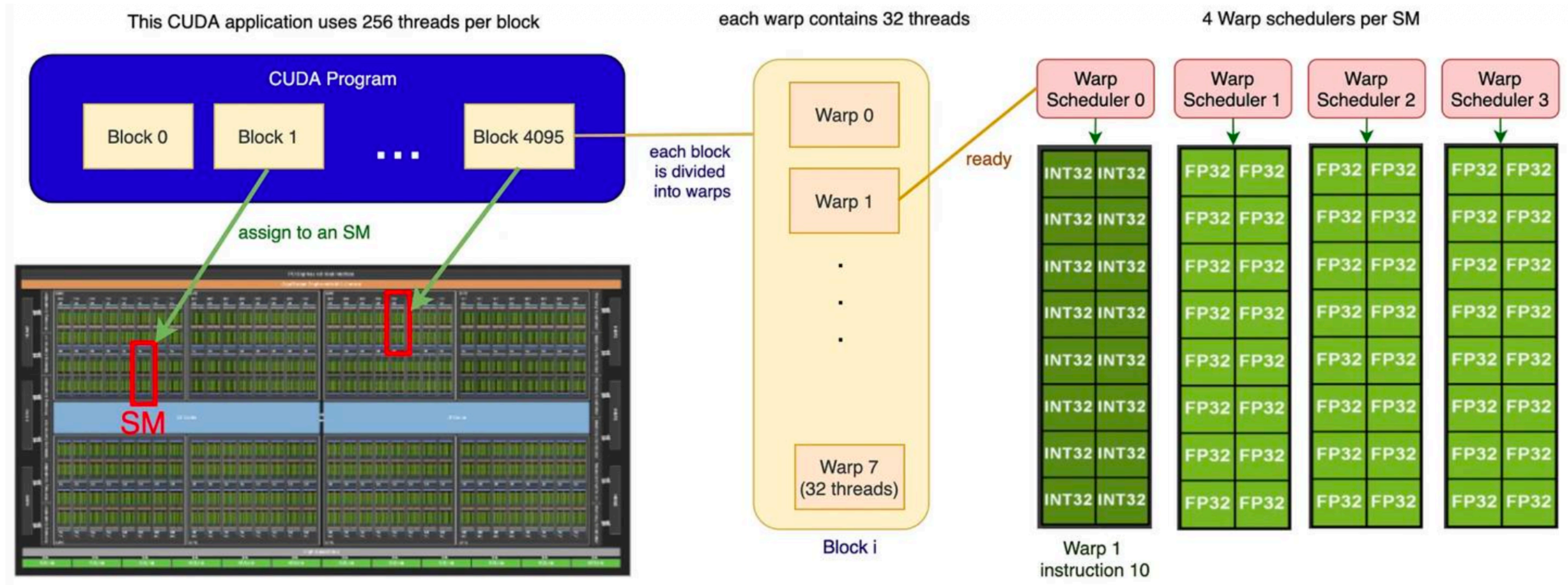
# GPU Execution Model

- Warps: The fundamental hardware execution unit
  - Threads always execute in a warp of exactly 32 threads each
  - Thread block is strictly partitioned into multiple warps for execution

- SIMT (Single Instruction, Multiple Threads)
  - Scheduler fetches one instruction and broadcasts to warp
  - All 32 threads in a warp execute the same instruction simultaneously, but on different data

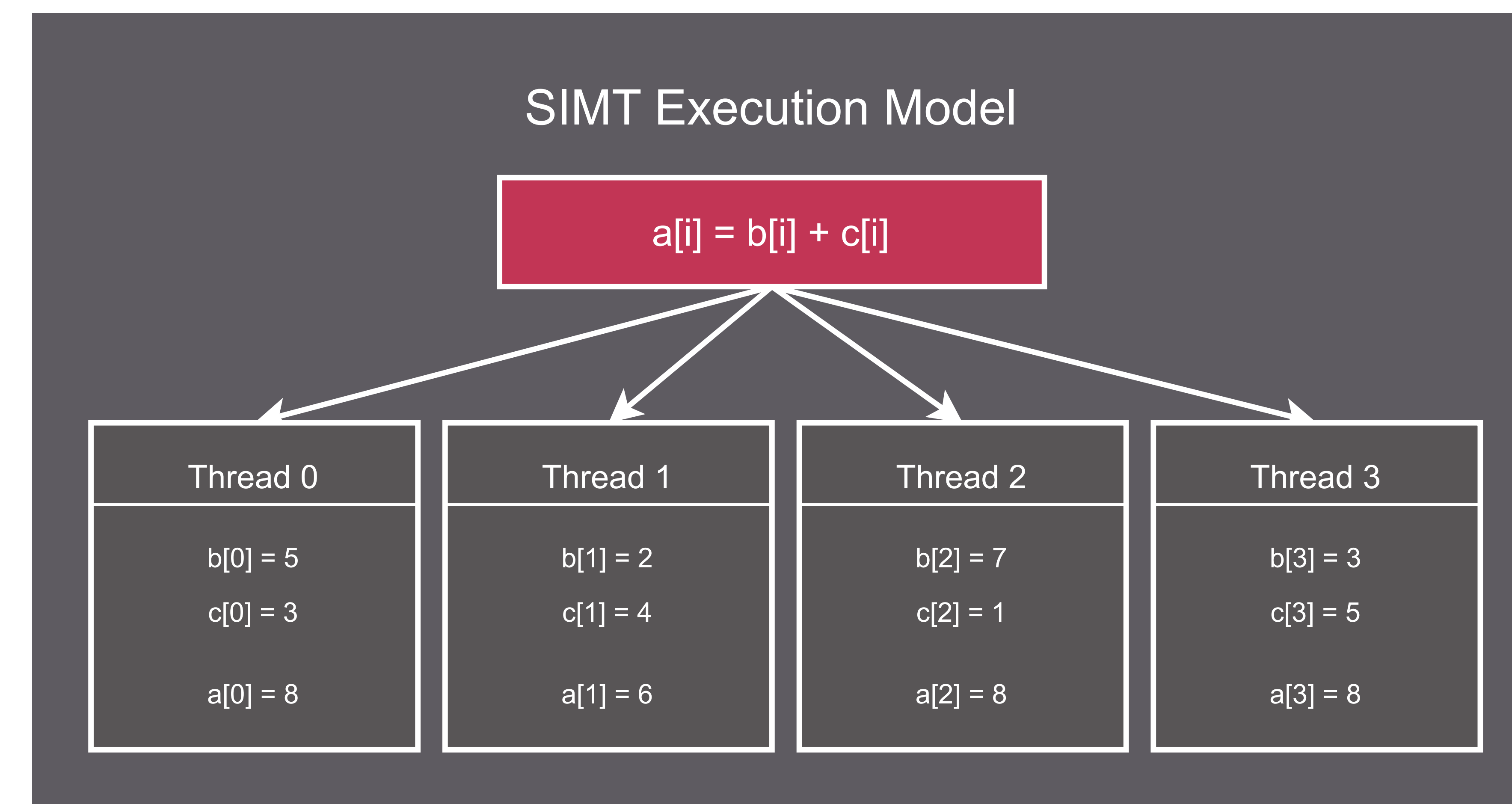


# GPU Execution Model



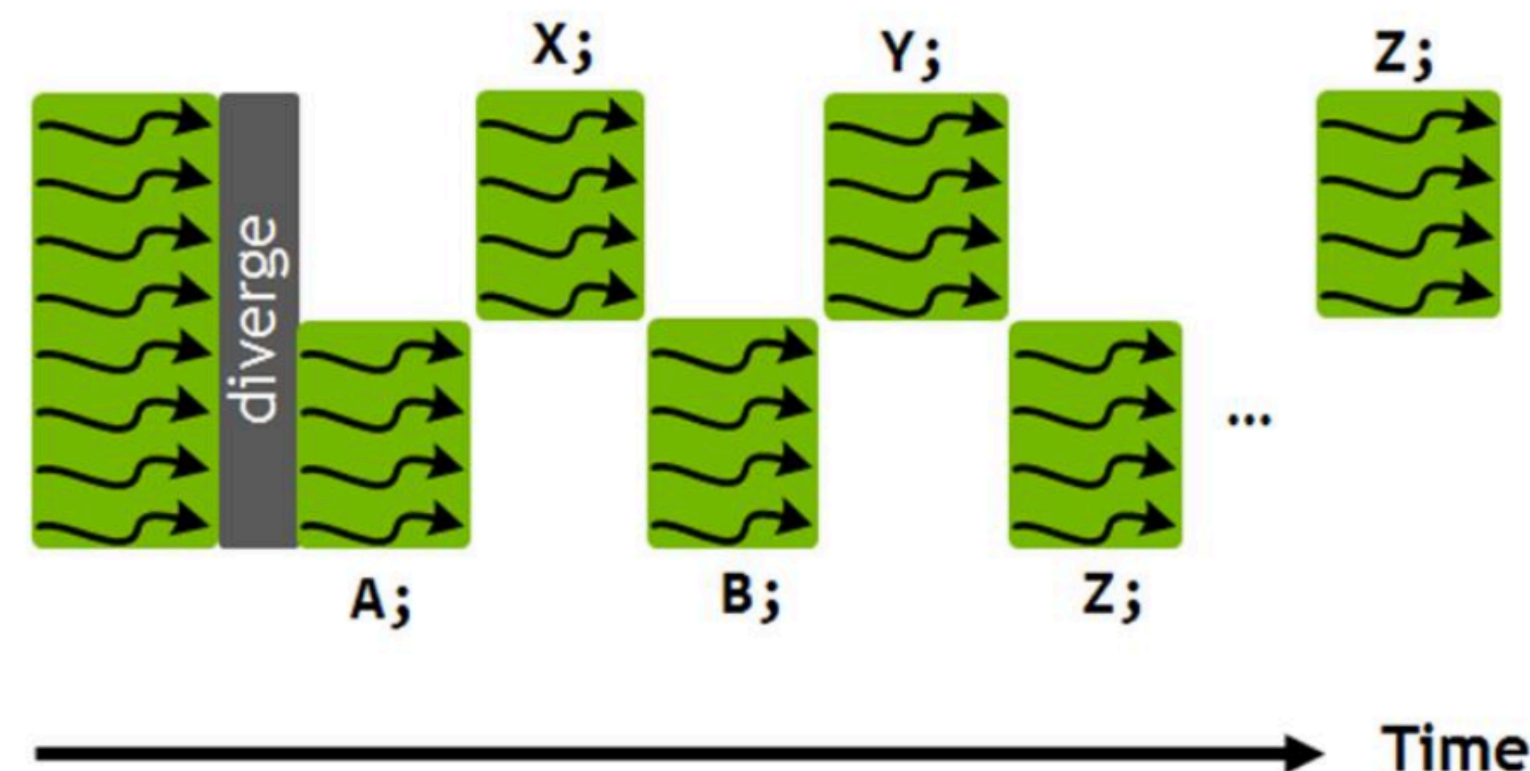
# Issue: Control Divergence

- SIMT model: every thread in a warp is executing the same instruction




- Conditionals lead to significant overhead from the execution model
  - When executing "if" threads, "else" threads are masked out
  - Waste ALU cycles as inactive threads remain idle during divergent execution paths

```
if (threadIdx.x < 4) {
    A;
    B;
} else {
    X;
    Y;
}
Z;
```



# GPU Memory Model

- Each thread can access its own shared memory within the block
- But, information that goes across thread blocks needs to be read/written to global memory (slow)

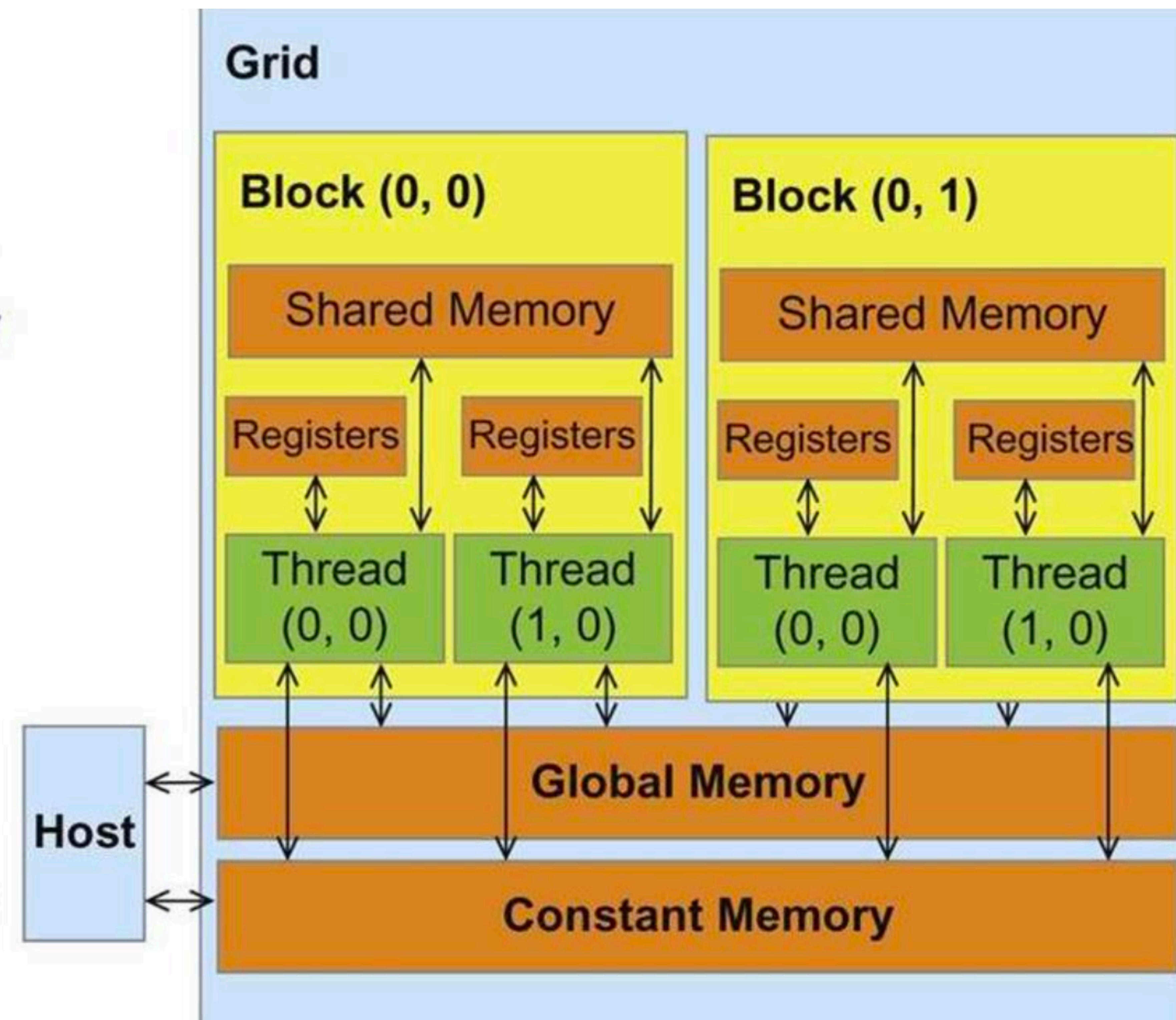
| Memory type | *CPI (cycles)* |
|---|---|
| Global memory | 290 |
| L2 cache | 200 |
| L1 cache | 33 |
| Shared Memory (ld/st) | (23/19) |

Device code can:

- R/W per-thread registers
- R/W per-thread local memory
- R/W per-block shared memory
- R/W per-grid global memory
- Read only per-grid constant memory

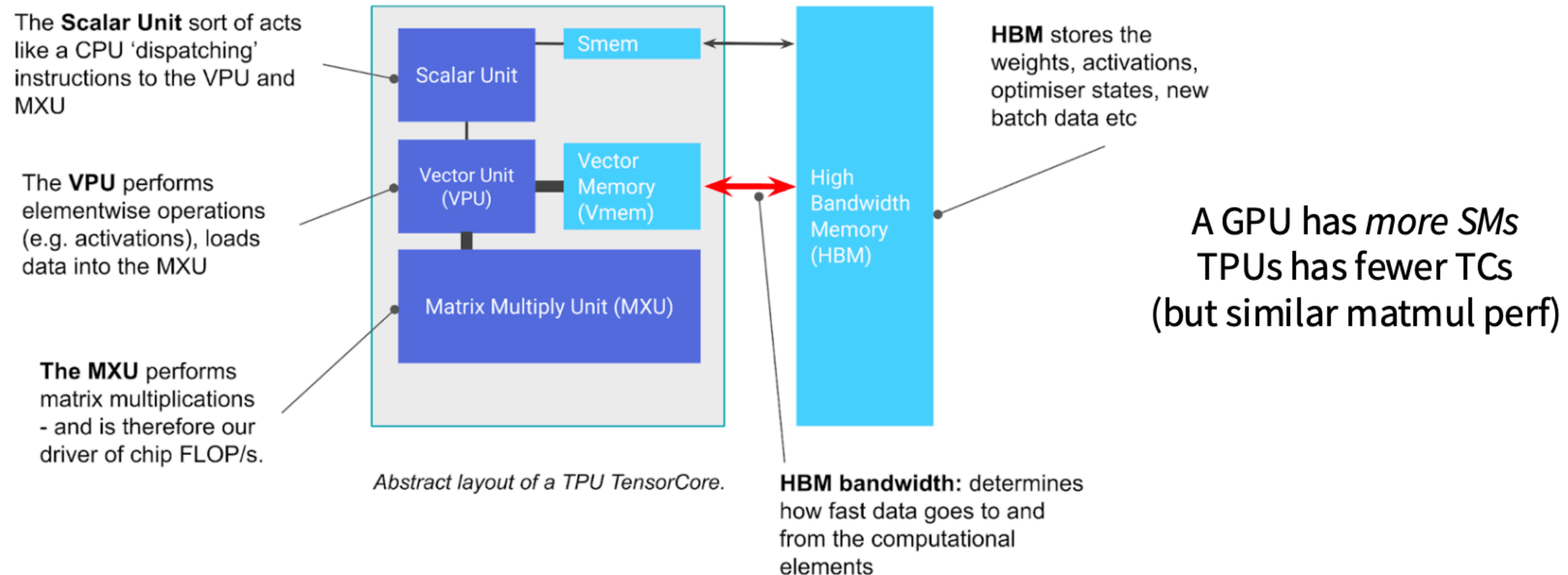
Host code can

- Transfer data to/from per grid global and constant memories



# TPU Memory Model

- GPUs, TPUs, and many other accelerators are at a high level, similar




*Abstract layout of a TPU TensorCore.*

- Core structure: lightweight control, fast (big) matmul unit, fast memory
- Difference: networking (will see in the parallelism), no warps

# GPUs as Fast Matrix Multipliers

- Early days of NVIDIA GPUs were programmable shaders (e.g. game)
  - Researchers hacked this to do matmuls

**Fast Matrix Multiplies using Graphics Hardware**

E. Scott Larsen
Department of Computer Science
University of North Carolina at Chapel Hill
Chapel Hill, NC 27599-3175 USA
larsene@cs.unc.edu

David McAllister
Department of Computer Science
University of North Carolina at Chapel Hill
Chapel Hill, NC 27599-3175 USA
davemc@cs.unc.edu

**Implementation**

We mention here some observations we made during our implementation that may be of interest to those duplicating our results.

**Refresh Rate** We found that setting the refresh rate on the monitor as low as possible made marginal improvements (about 10%).

**RGBA** We found that 4 numbers can be packed into a single pixel, by setting the red, green, blue, and alpha channels to different values.

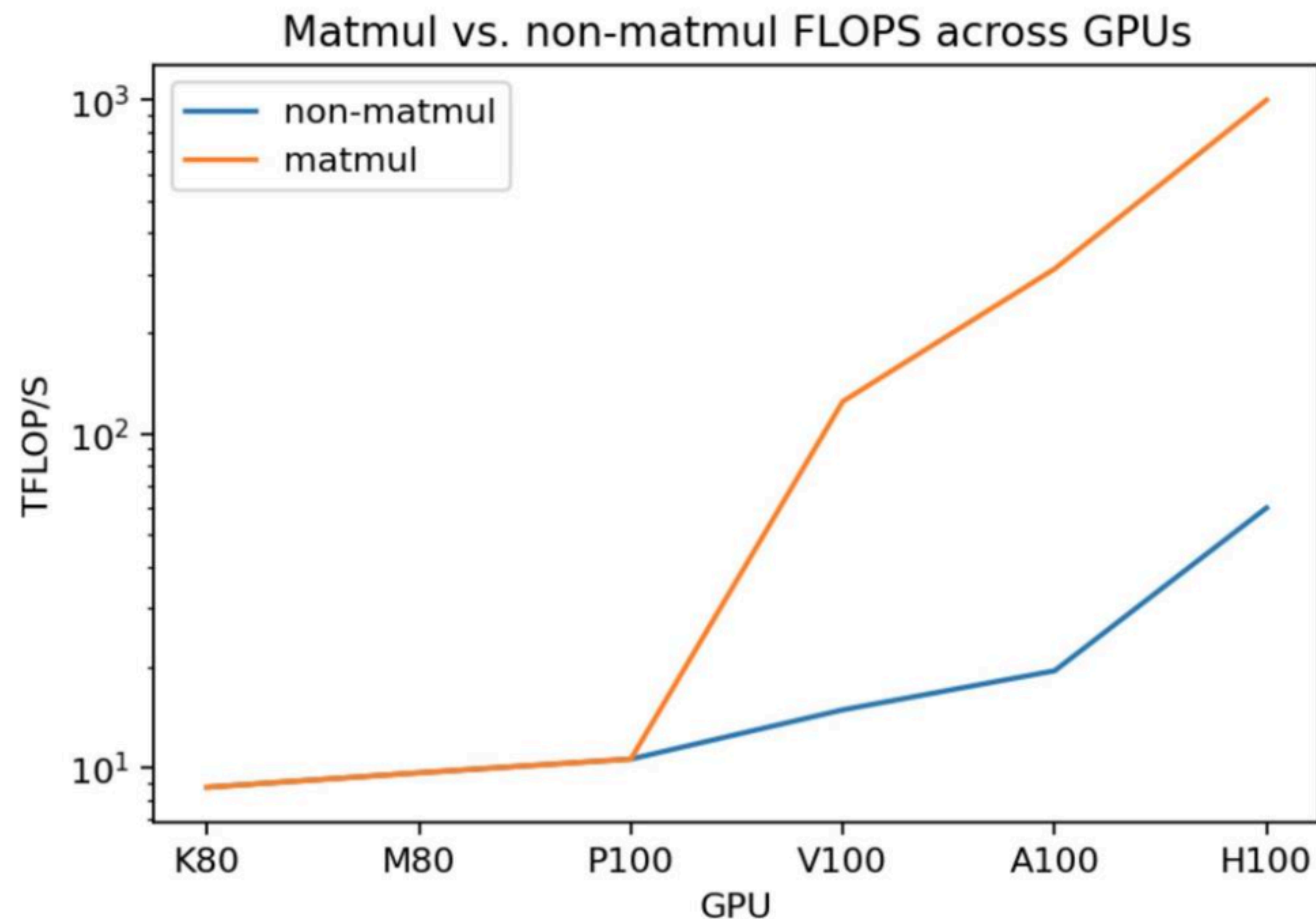
**Texture Format** Changing the format of the texture creation and read-back from RGBA to ABGR_EXT (in OpenGL) gave about 40% improvement on our hardware. This is because the hardware driver avoids reformatting the data from the application format to the card format. There is a number of options here, with near equal performance for each option except the one used natively on the specific hardware. The native format should give significant improvement.

**Full Screen** Running full screen instead of in a window provides improved performance.

Various other optimizations yielded minor (<1%) improvements.

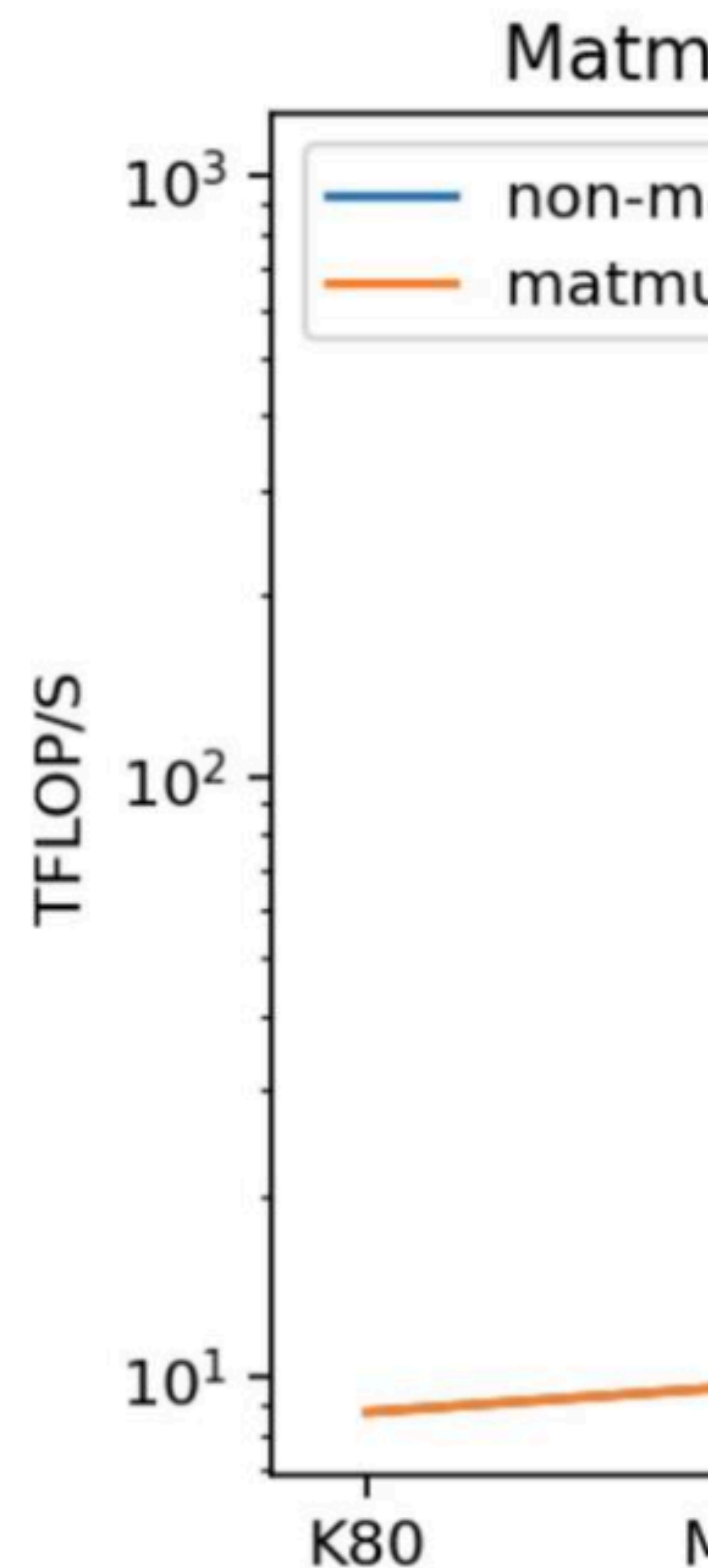
# GPUs as Fast Matrix Multipliers

- Tensor cores (introduced in V, T series) are specialized matmul circuits
- Matmuls are >10x faster than other floating point ops



# GPUs as Fast Matrix Multipliers

- Tensor cores (introduced in
- Matmuls are >10x faster tha




ksa @kosa12m · 2h
still thinking about this

ksa @kosa12matyas · 1d
I never thought this would take my job.

$$\begin{bmatrix} a_1 & a_2 & a_3 \\ a_4 & a_5 & a_6 \\ a_7 & a_8 & a_9 \end{bmatrix} \begin{bmatrix} b_1 & b_2 & b_3 \\ b_4 & b_5 & b_6 \\ b_7 & b_8 & b_9 \end{bmatrix} = \begin{bmatrix} c_1 & c_2 & c_3 \\ c_4 & c_5 & c_6 \\ c_7 & c_8 & c_9 \end{bmatrix}$$

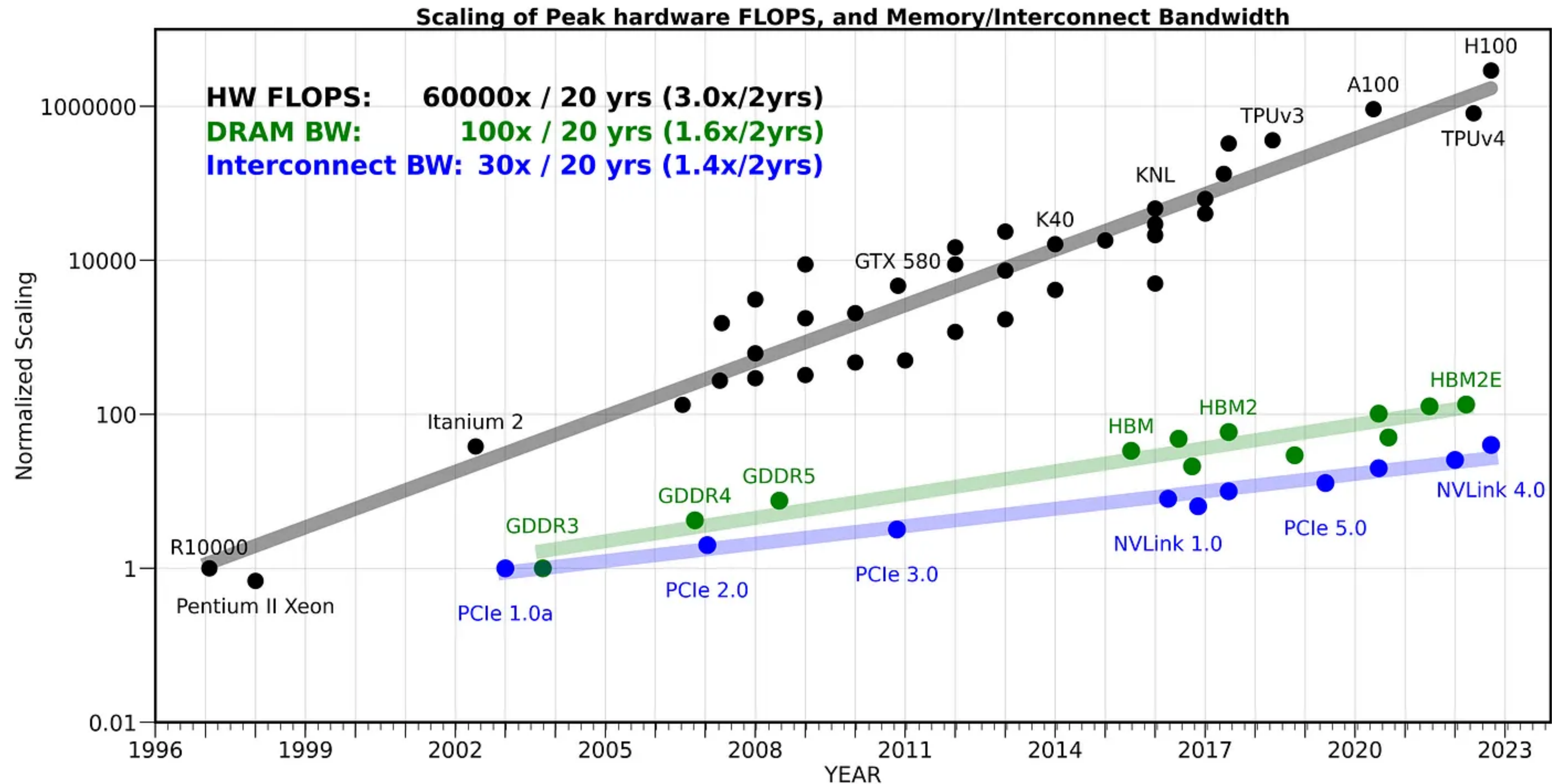
172 1,8K 23K 4,5M

FJ.dgb @fabianjaxon · 1d
@grok please explain

5 1 19 20K

# Compute Scaling vs. Memory Scaling

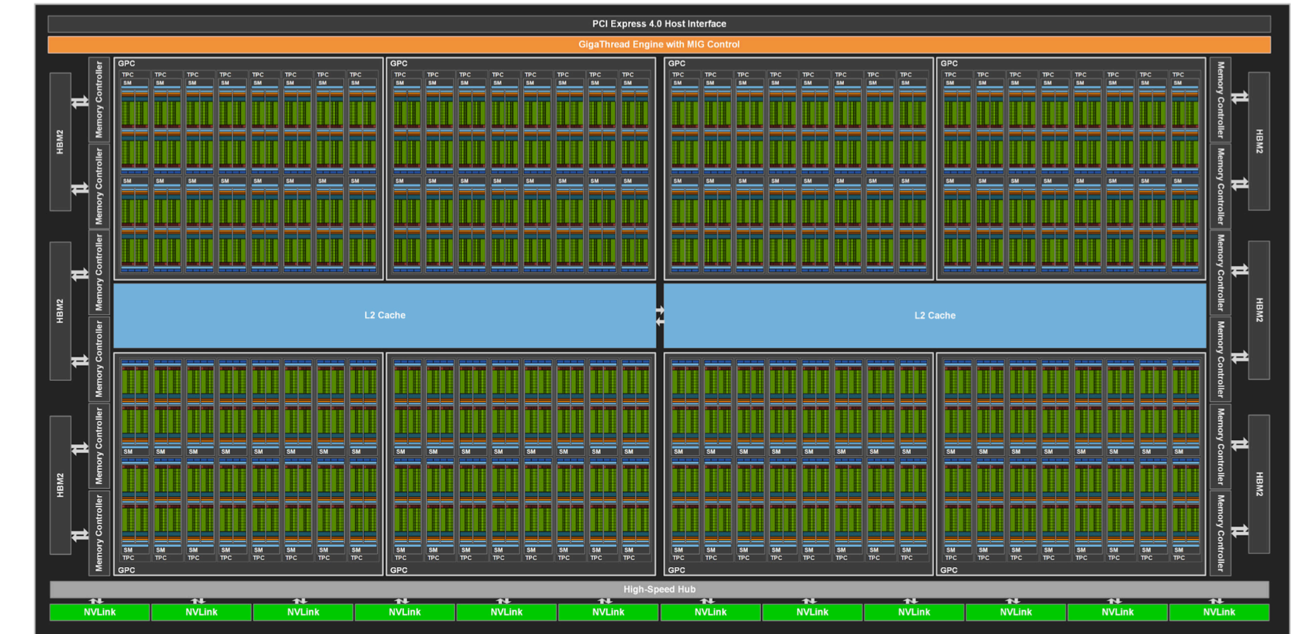
- FLOPs scale faster than memory



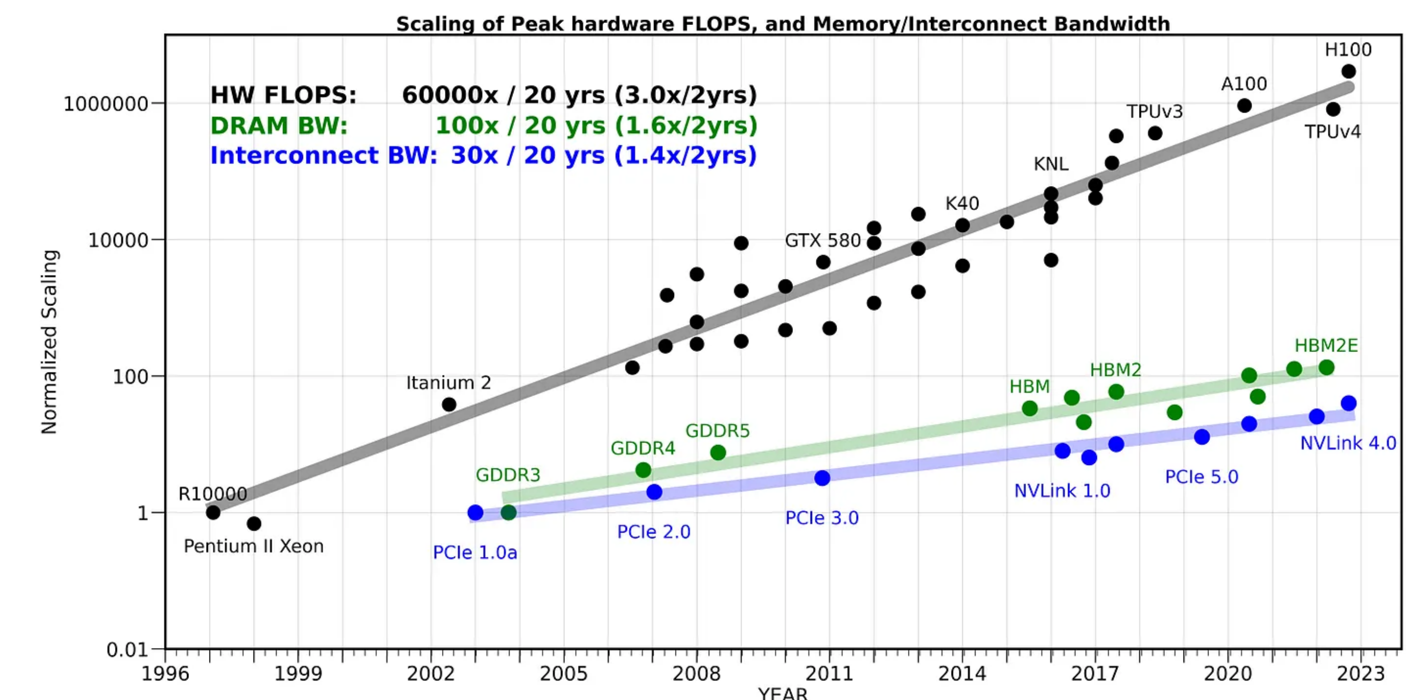

https://medium.com/riselab/ai-and-memory-wall-2cb4265cb0b8

# Recap

- GPUs are massively parallel; same instructions applied across many workers



- Compute (and especially matmuls) have scaled faster than memory



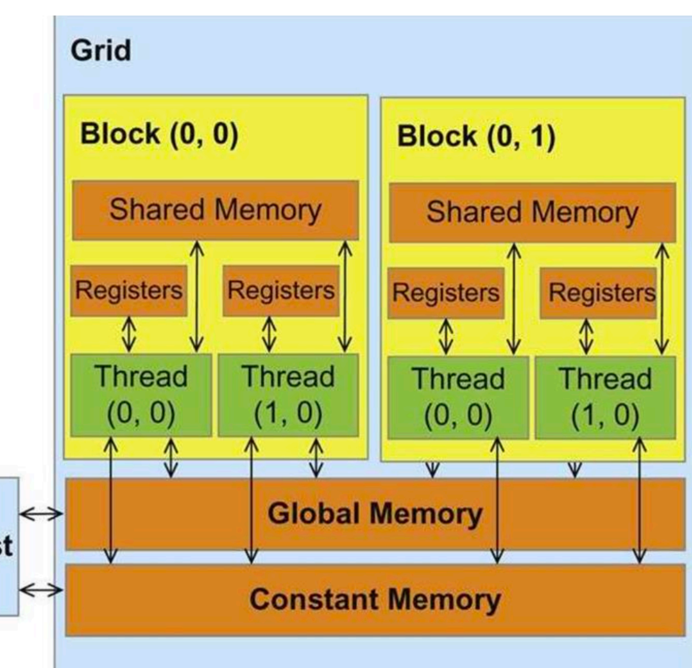

- We have to respect the memory hierarchy to make things go fast

Device code can:

- R/W per-thread registers
- R/W per-thread local memory
- R/W per-block shared memory
- R/W per-grid global memory
- Read only per-grid constant memory

Host code can

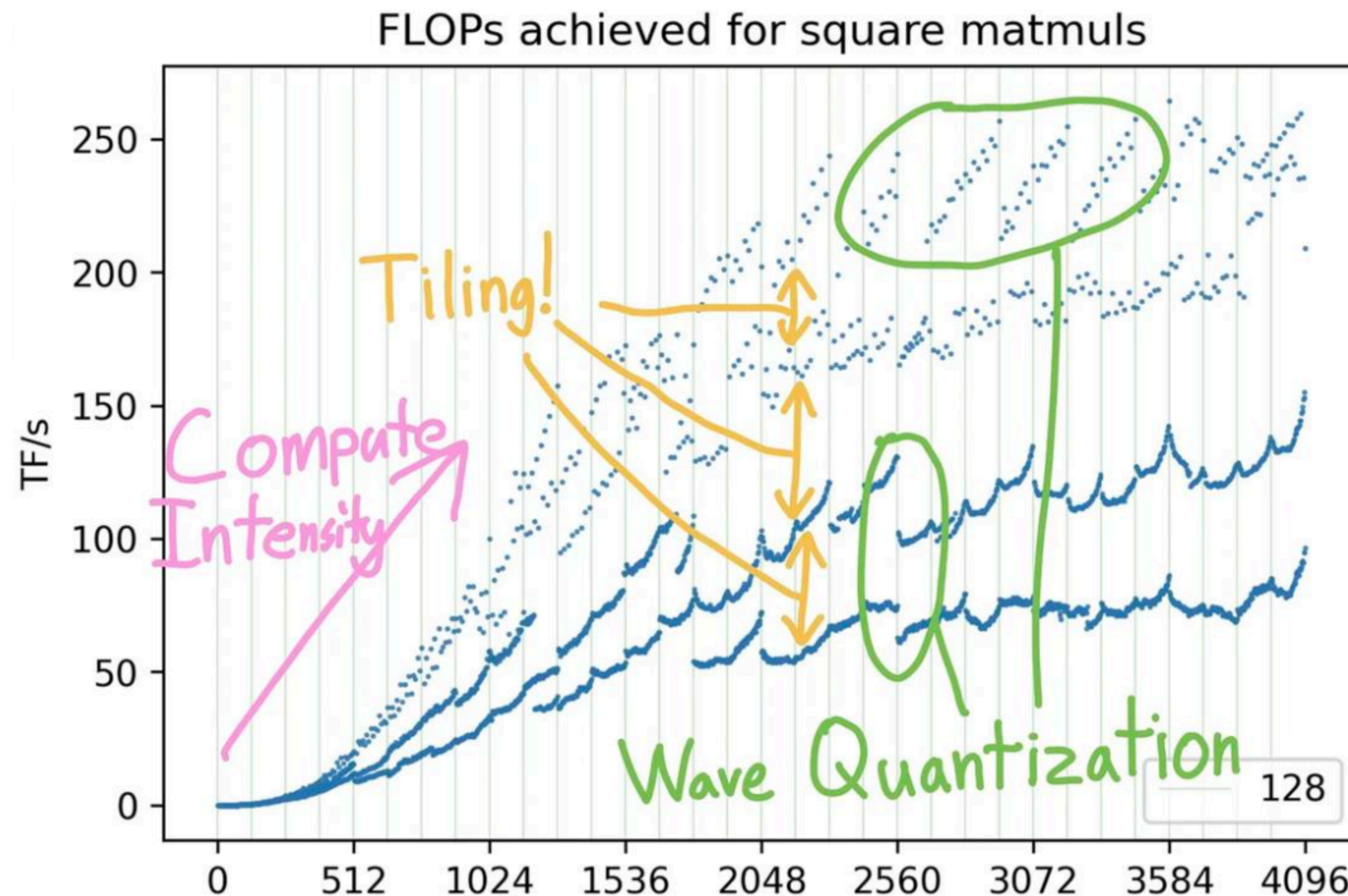
- Transfer data to/from per grid global and constant memories



# Understanding GPU performance

# Making Workloads Fast on a GPU

- But performance on a GPU can be complex, even for something as simple as a square matmul



# Execution time

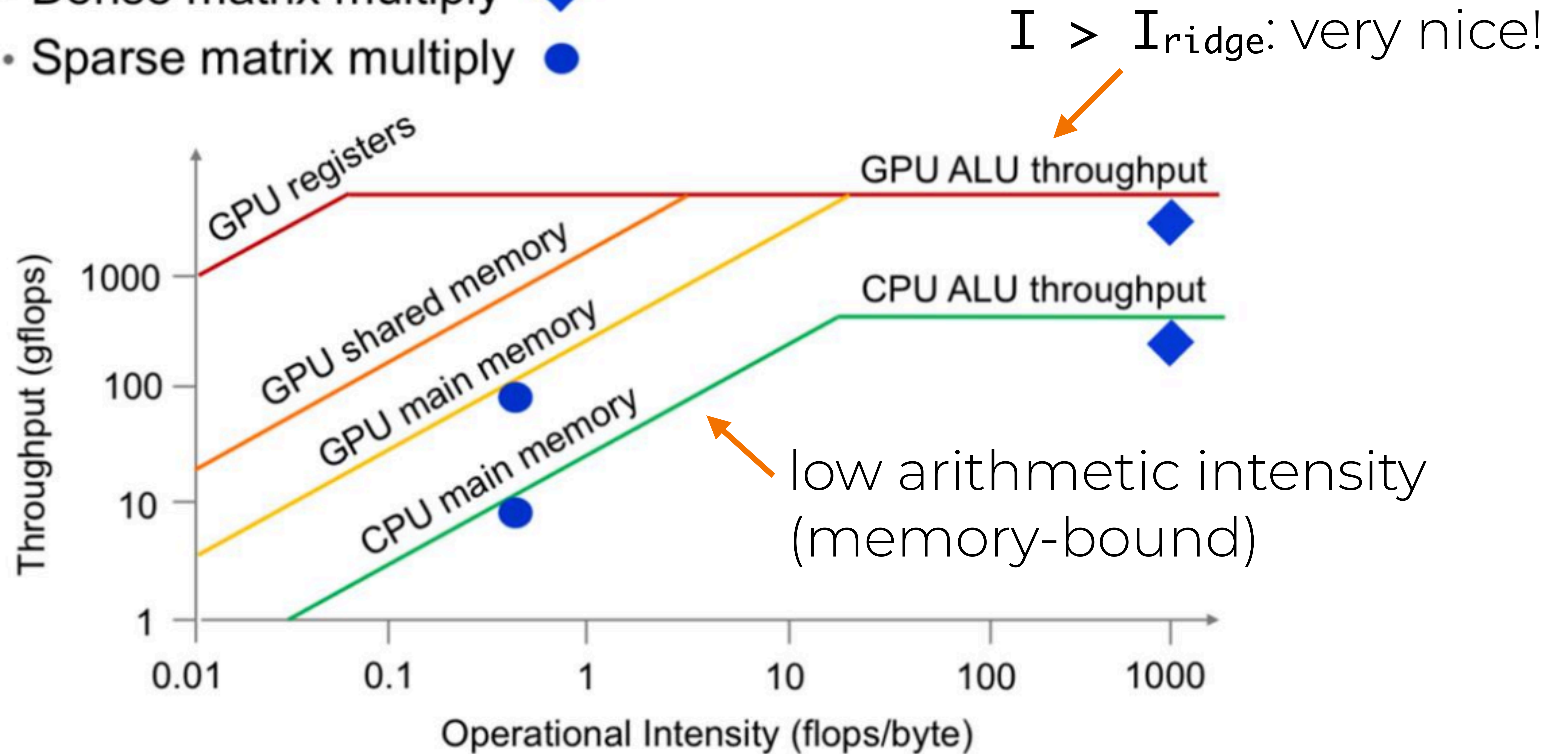
- The execution time of models is primarily determined by these
- Computation
  - Peak throughput of the hardware (GPU/TPU's FLOP/s)
  - e.g. H100 = 9.89e14 FLOP/s, TPU v6e = 9.1e14 FLOP/s
- Communication
  - Overhead associated with data movement
  - Transfer data between memory (HBM, DRAM) and ALU (SM/Core)
- `execution time = max(computation, communication)`

# Arithmetic Intensity

- Operational efficiency of an algorithm relative to data movement
  - `I = Total Operations (FLOPs) /  Total Communication (Bytes)`
  - Higher $I$: Efficient; high volume of computation per data fetch
  - Lower $I$: Inefficient; performance is bottlenecked by data movement

- Peak arithmetic intensity (ridge point)
  - Every hardware architecture has its own optimal threshold $I_{ridge}$
    - e.g. H100 has 295 FLOPs/Byte
  - $I < I_{ridge}$: Comm.-bound (perf. limited by memory bandwidth)
  - $I > I_{ridge}$: Compute-bound (fully utilizing compute capacity)

# Roofline Model




$I > I_{ridge}$: very nice!

low arithmetic intensity (memory-bound)

# Example: Dot Product

- Setup: dot product **x** and **y**, which are both length `D`
- Communication (Bytes)
  - Load `x,y`: 2 x 2 x D (fp16) / Save `x*y`: 2
  - Total: 4 x D + 2

- Compute (FLOPs)
  - Total: 2 x D (D mult and D add)

- Arithmetic intensity
  - I = (2 X D) / (4 x D + 2) = 0.5
  - Take 2 bytes of data and a single FLOP computation

# Example: Matmul

- Setup: matmul `X` (`[BxD]`) and `W` (`[DxF]`) and produce `Y`
- Communication (Bytes)
  - Load `X,W`: 2 x B x D+ 2 x D x F / Save `Y`: 2 x B x F
  - Total: (2 x B x D)+ (2 x D x F) + (2 x B x F)

- Compute (FLOPs)
  - Total: 2 x B x D x F (2 for multiplication and addition)

- Arithmetic intensity
  - I = (2 x B x D x F) / ((2 x B x D)+ (2 x D x F) + (2 x B x F))
  - I = B (if we assume B is much less than D and F)

# Example: Matmul

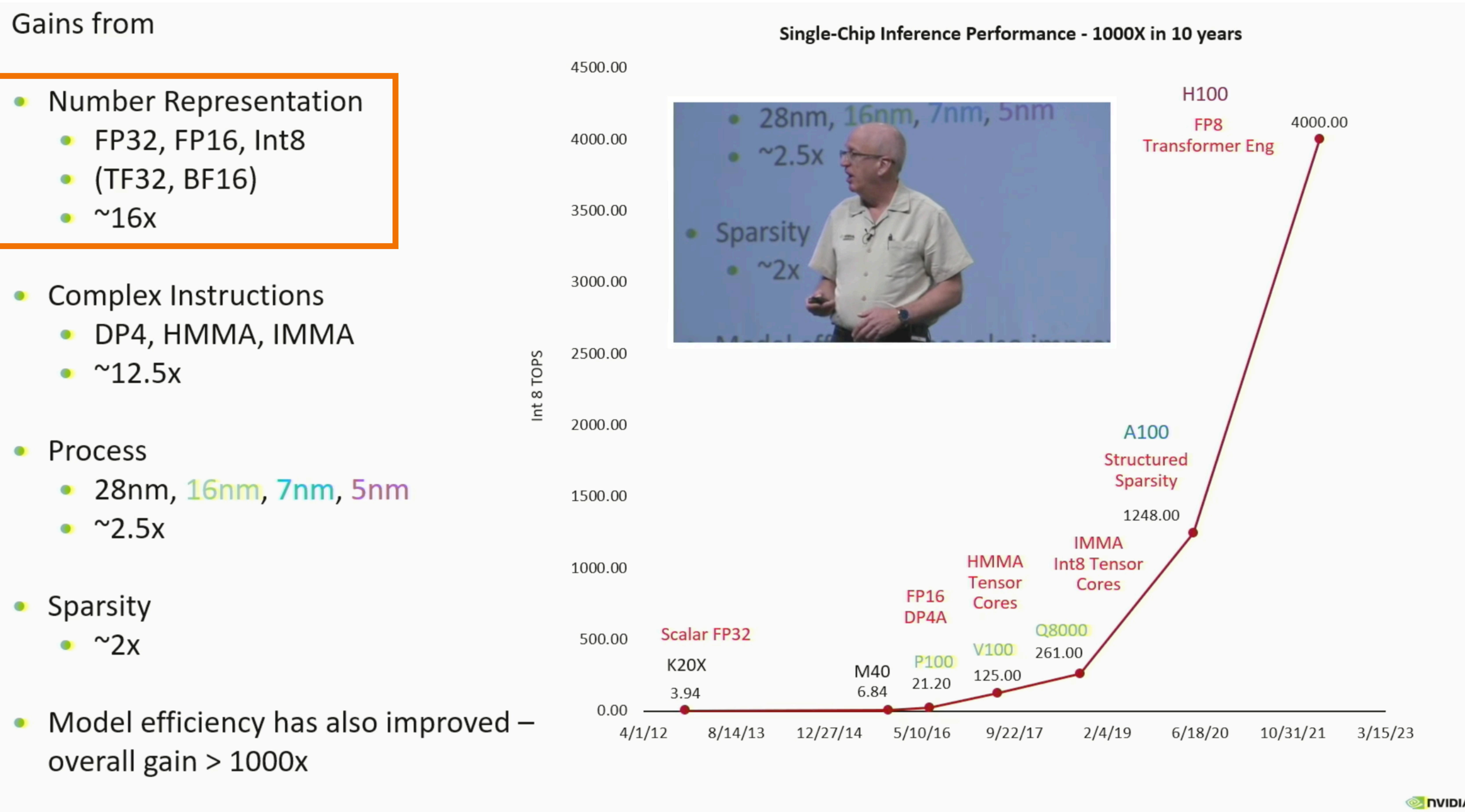
- Arithmetic intensity
  - $I = B$
- Peak arithmetic intensity of H100
  - $I_{ridge}$ = 295 (FLOP/s = 989e12, memory bandwidth = 3.36e12)
- Conclusion: compute-limited iff B > 295
  - Why? in matmul, elems in row and col are reused multiple times
- But extreme case; B = 1, then I = 1 (memory-limited)
  - Why? read D x F matrix, perform only 2xDxF FLOPs

# How Do We Make GPUs Go Fast?

- Low precision computation
- Kernel fusion
- Recomputation
- Coalescing memory
- Tiling

# Trick 1: Low Precision Computation

- If you have fewer bits, you have fewer bits to move



# Trick 1: Low Precision Computation

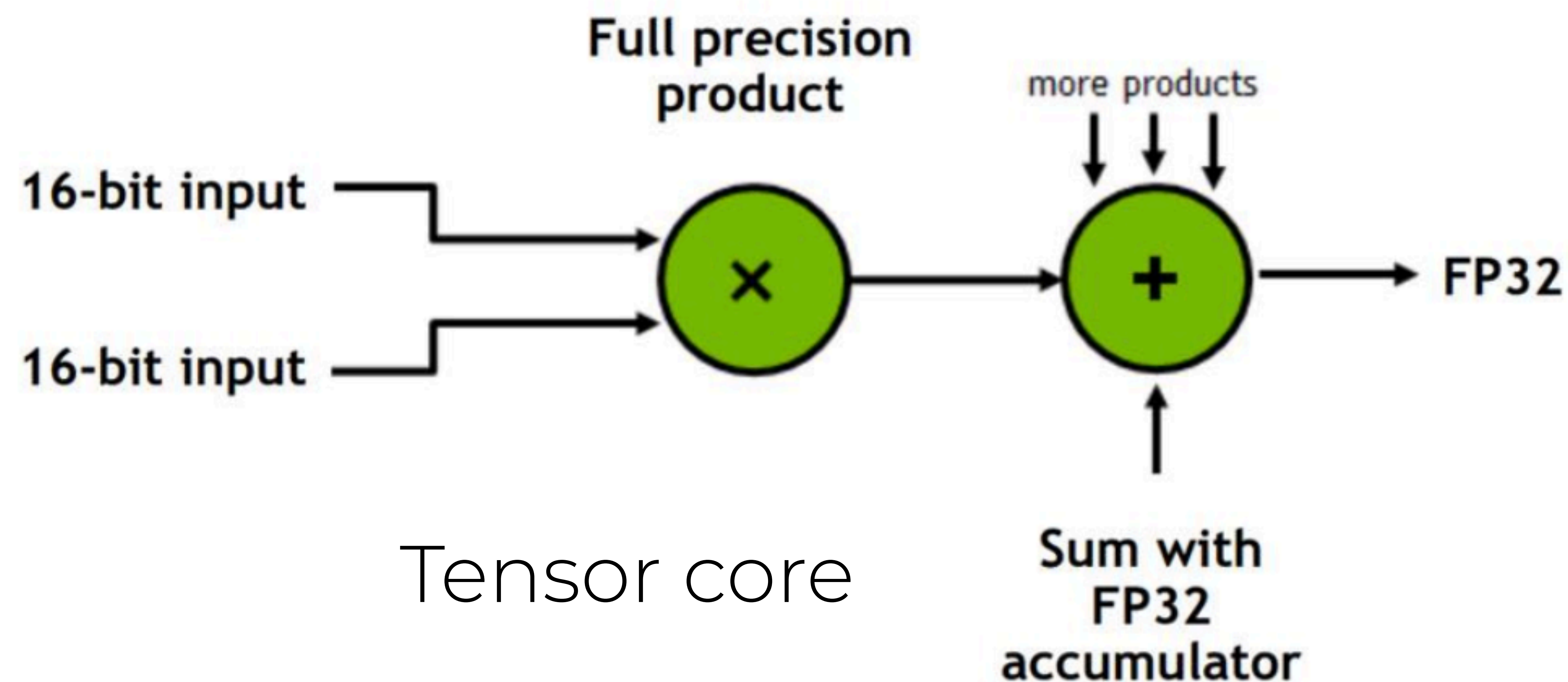
- Low precision improves arithmetic intensity
- Example: elementwise ReLU (`x = max(0, x)`) on a vector of size `n`

- fp32 case:
  - Memory access: 1 read, 1 write, → 2x4xN bytes
  - Operations: N FLOPs
  - Arithmetic intensity: 0.125

- fp16 case:
  - Memory access: 1 read, 1 write, → 2x2xN bytes
  - Operations: N FLOPs
  - Arithmetic intensity: 0.25

# Trick 1: Low Precision Computation

- Low precision drives faster matrix multiplies
  - Lots of operations in modern GPUs are accelerated via low/mixed precision operations




Tensor core

**Operations that can use 16-bit storage (FP16/BF16)**

- Matrix multiplications
- Most pointwise operations (e.g. relu, tanh, add, sub, mul)

**Operations that need more precision (FP32/FP16)**

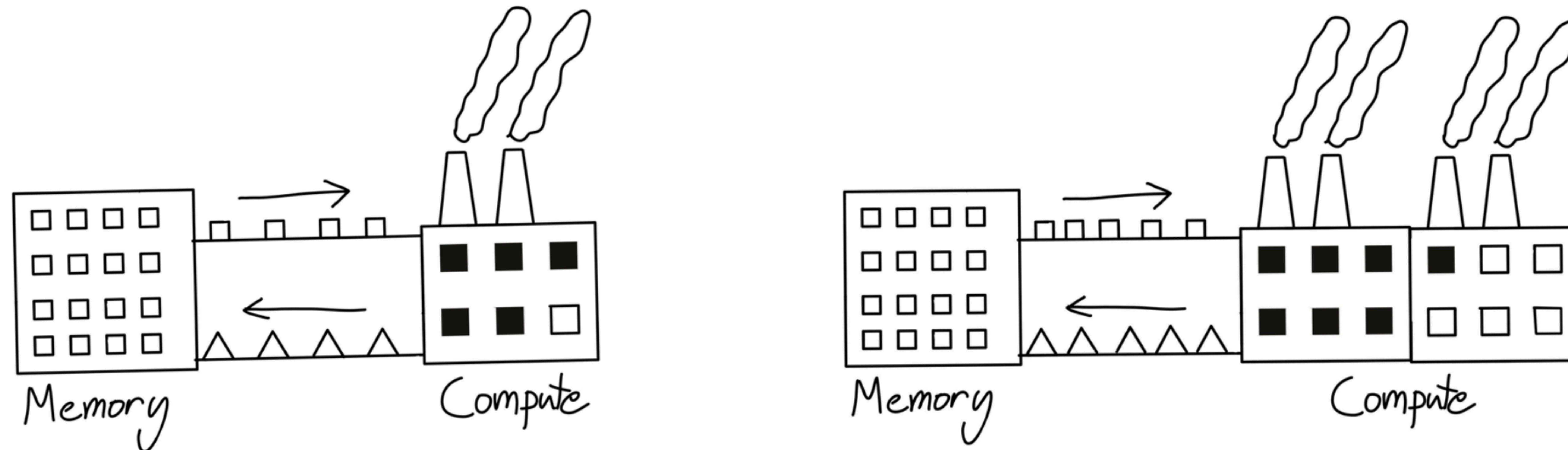
- Adding small values to large sums can lead to rounding errors
- Reduction operations (e.g. sum, softmax, normalization)

**Operations that need more range (FP32/BF16)**

- Pointwise operations where $|f(x)| \gg |x|$ (e.g. exp, log, pow)
- Loss functions

# Trick 2: Kernel Fusion

- Think of a GPU like a warehouse and a factory
  - Inputs come from a warehouse (memory) and are processed at a factory (core)

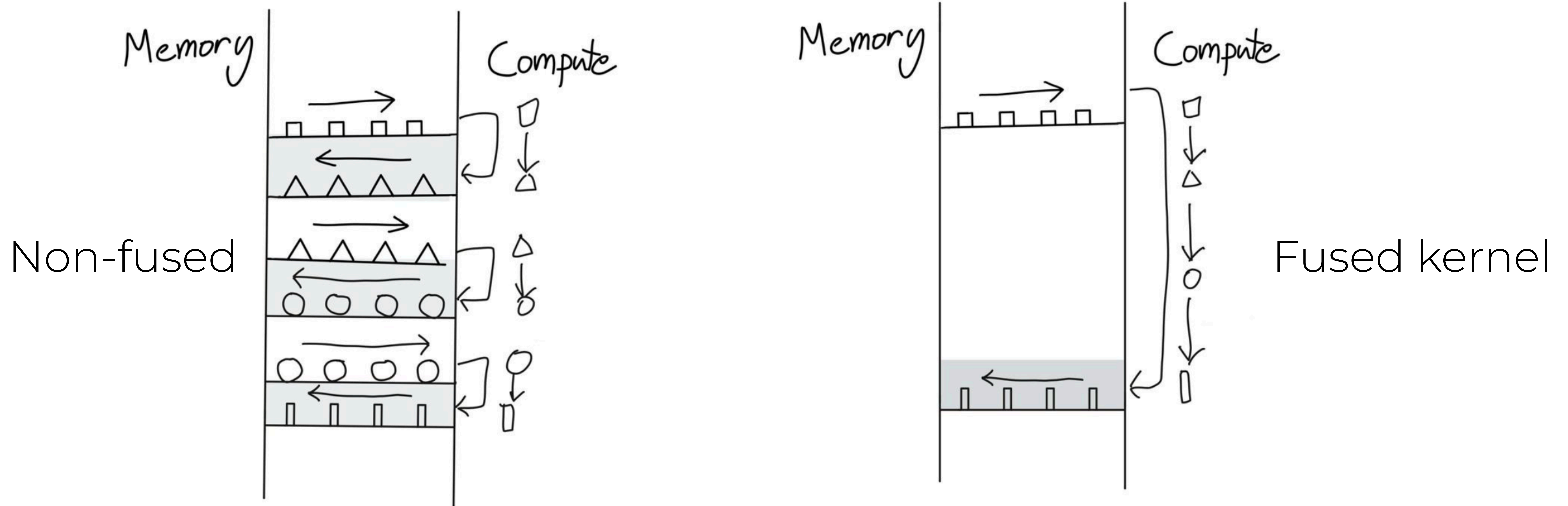



Compute scales up, memory bandwidth doesn't

https://horace.io/brrr_intro.html

# Trick 2: Kernel Fusion

- What if we have to do many operations?
  - Shipping back and forth is somewhat silly. Fuse these!




https://horace.io/brrr_intro.html

# Trick 2: Kernel Fusion

- Example: `Y = ReLU(X + B)` (all tensors are size of `D`)
- Naive non-fused kernel
  - `add` kernel: 1) load `X` and `B` from HBM, 2) do `add`, 3) save `X+B` to HBM
  - `relu` kernel: 1) load `X+B` from HBM, 2) do `relu`, 3) save `Y`
  - Arithmetic intensity: (2xD) / (5xD) = 0.4

- Fused kernel
  - `fused` kernel: 1) load `X` and `B` from HBM, 2) do `add` and `relu`, 3) save `Y`
    - Use on-chip register or SRAM (shared memory)
  - Arithmetic intensity: (2xD) / (3xD) = 0.67
    - Reuse loaded tensors so minimize HBM read/write round-trip

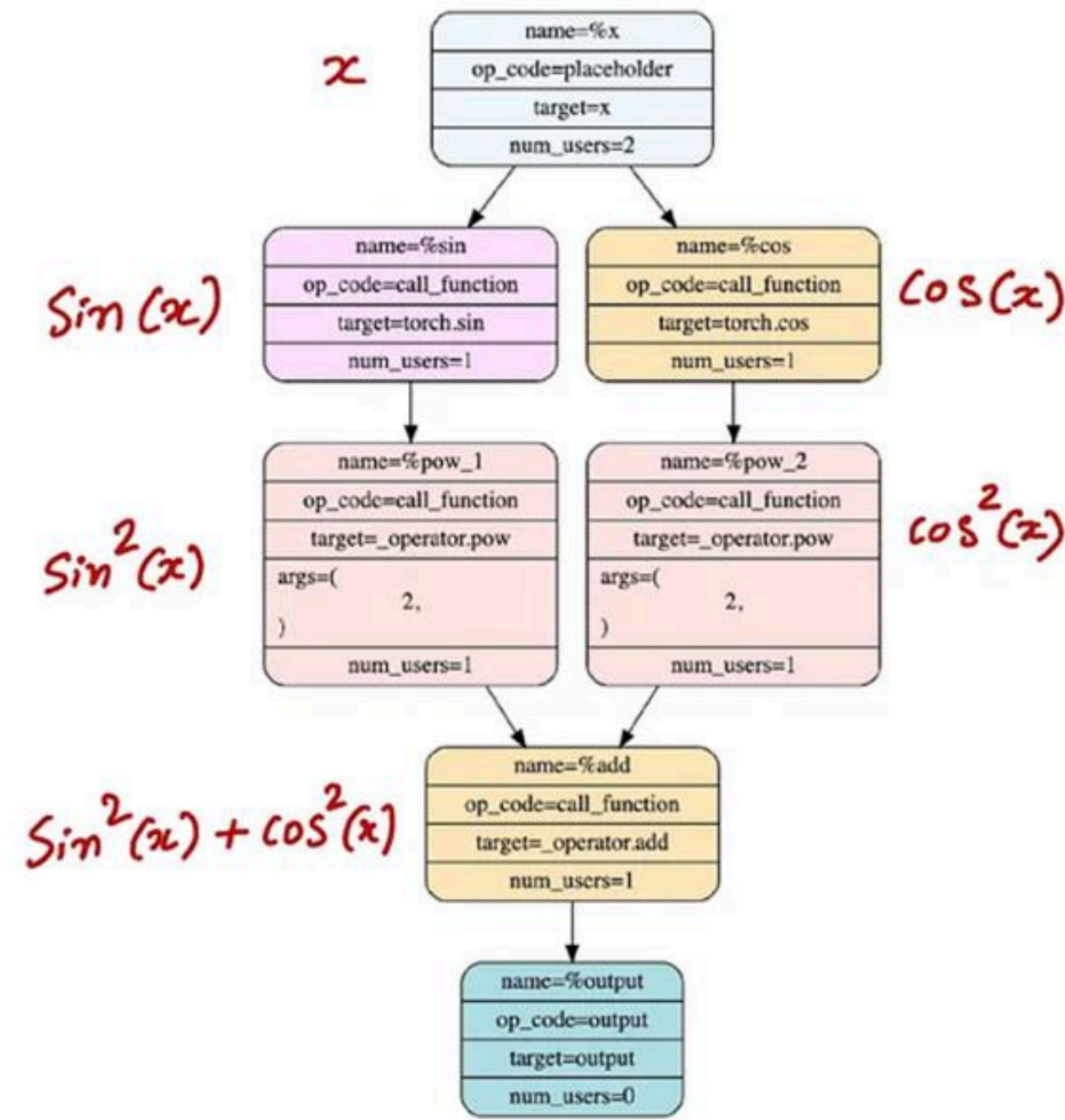
# Trick 2: Kernel Fusion

- Example: sines and cosines
  - Computing $sin^2 x + cos^2 x$ naively launches 5 CUDA kernels

FX GRAPH
IR

```
class GraphModule(torch.nn.Module):
    def forward(self, x : torch.Tensor):
        # File: /tmp/ipykernel_2583/1502985755.py:2, code:
        sin = torch.sin(x)
        pow_1 = sin ** 2;  sin = None
        cos = torch.cos(x);  x = None
        pow_2 = cos ** 2;  cos = None
        add = pow_1 + pow_2;  pow_1 = pow_2 = None
        return (add,)
```

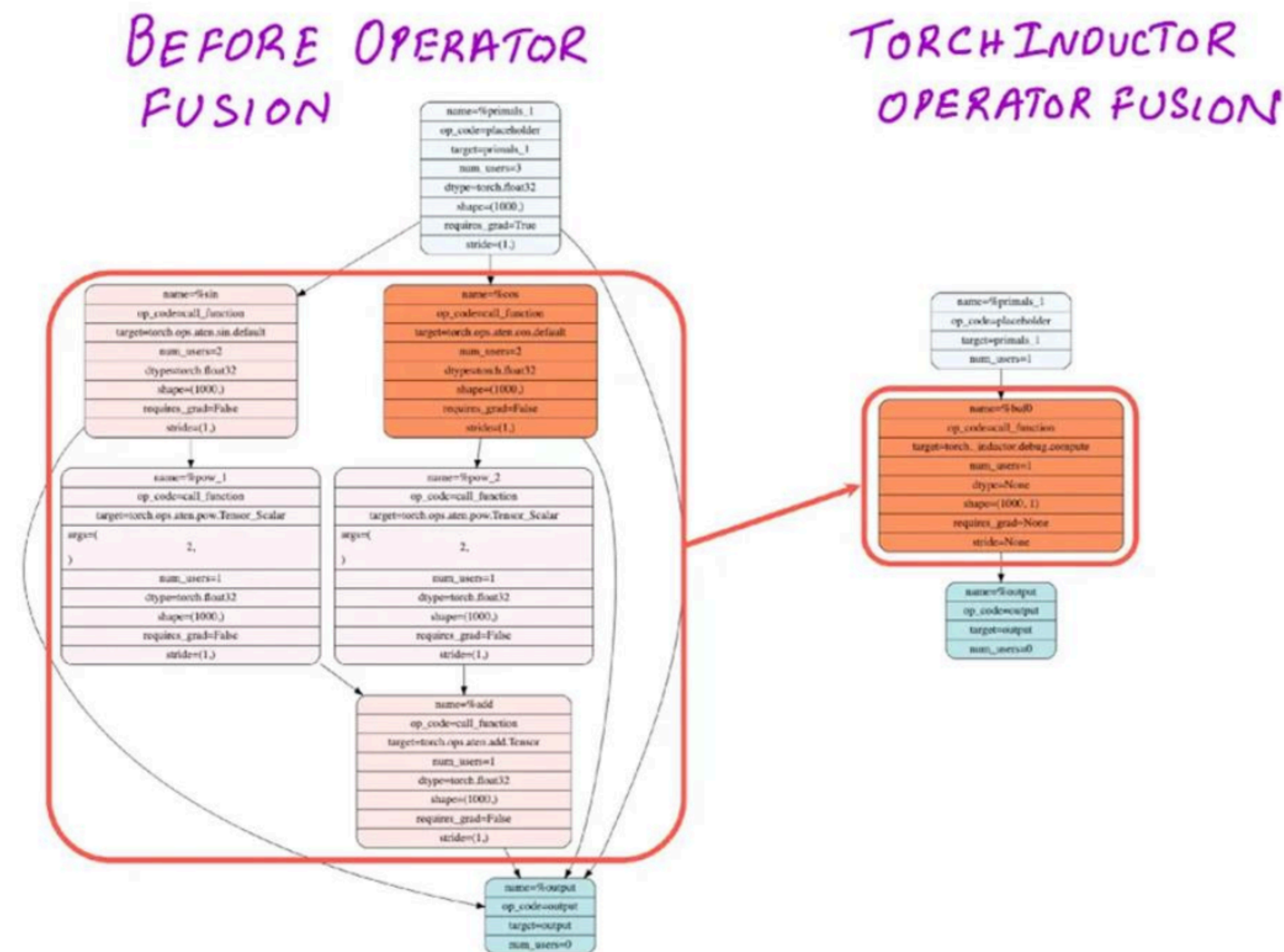
GRAPH VIZ




https://towardsdatascience.com/how-pytorch-2-0-accelerates-deep-learning-with-operator-fusion-and-cpu-gpu-code-generation-35132a85bd26

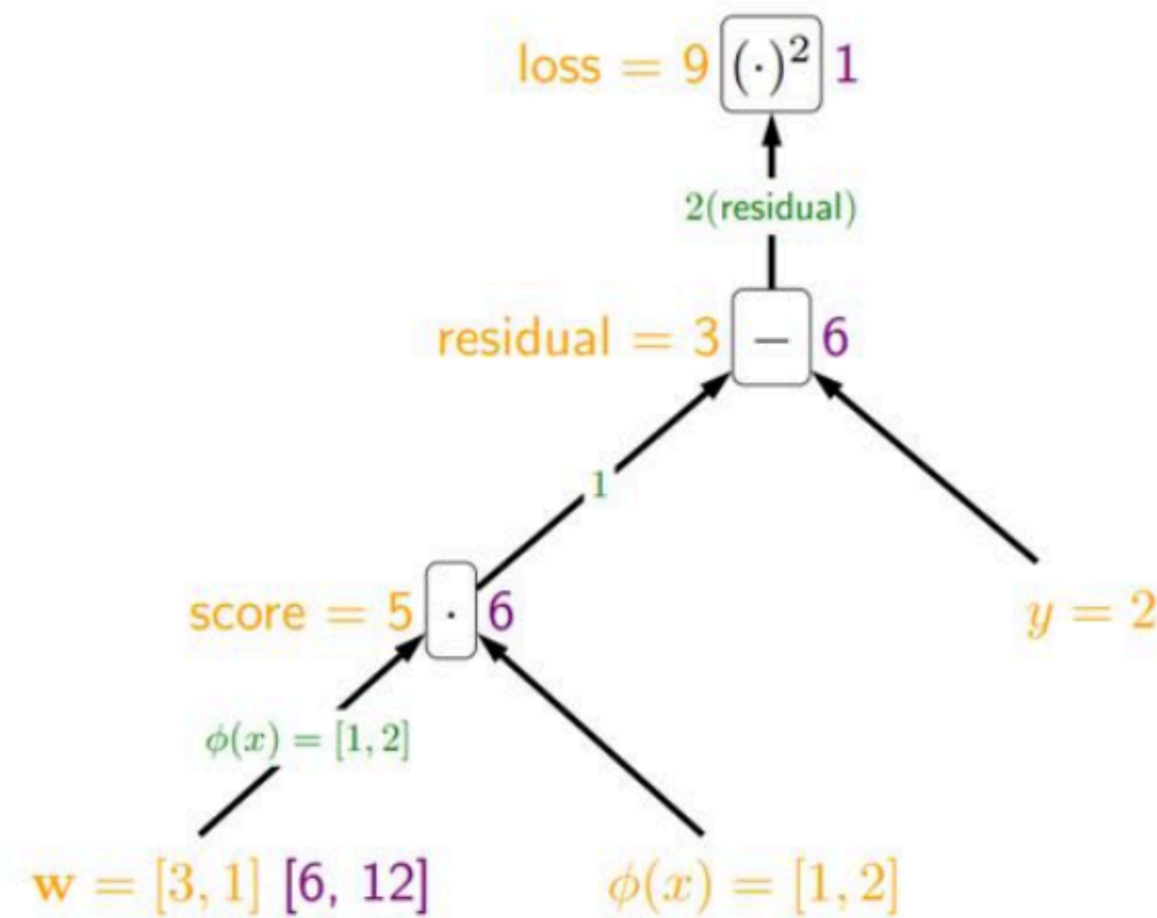
# Trick 2: Kernel Fusion

- Example: sines and cosines
  - All 5 pointwise operations can be fused into a single kernel call
  - Easy fusions like this can be done automatically by compilers (torch.compile)




https://towardsdatascience.com/how-pytorch-2-0-accelerates-deep-learning-with-operator-fusion-and-cpu-gpu-code-generation-35132a85bd26

# Trick 3: Recomputation



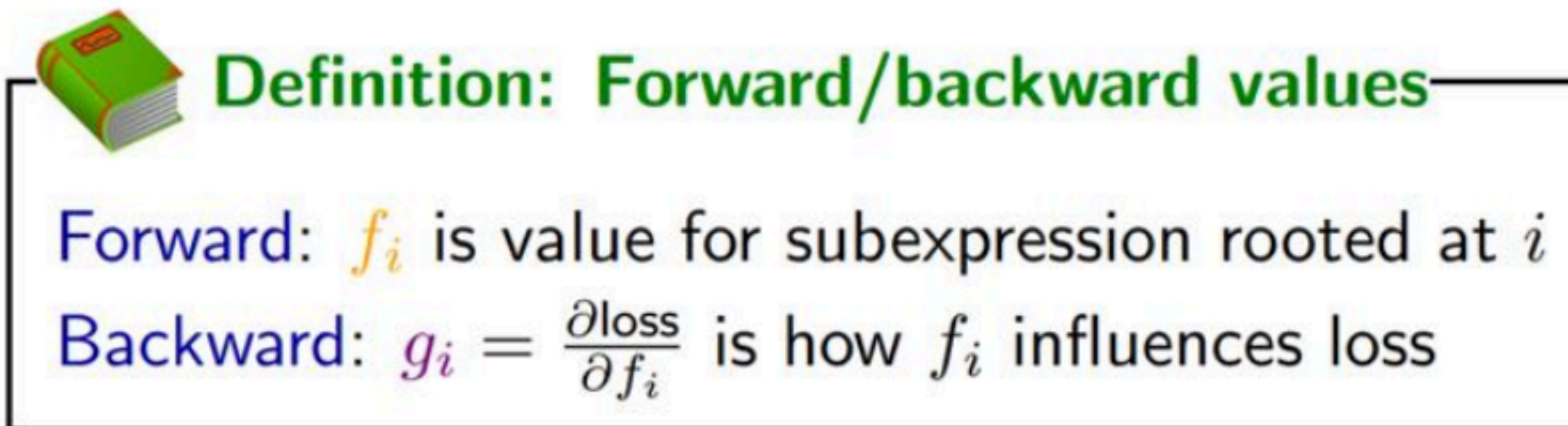

$\text{Loss}(x, y, \mathbf{w}) = (\mathbf{w} \cdot \phi(x) - y)^2$

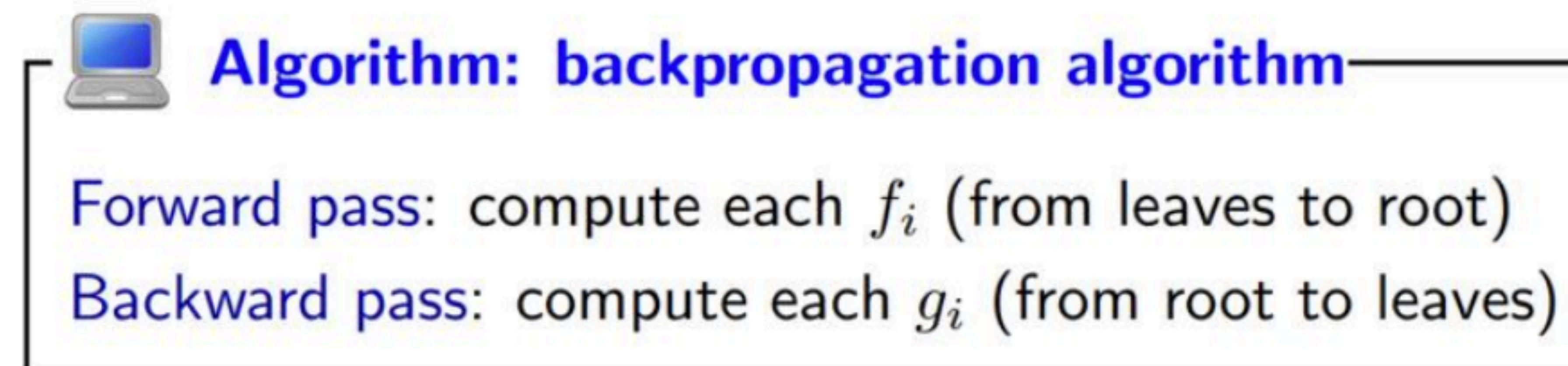
$\mathbf{w} = [3, 1], \phi(x) = [1, 2], y = 2$

backpropagation

$\nabla_{\mathbf{w}} \text{Loss}(x, y, \mathbf{w}) = [6, 12]$

Image credit: Stanford CS221



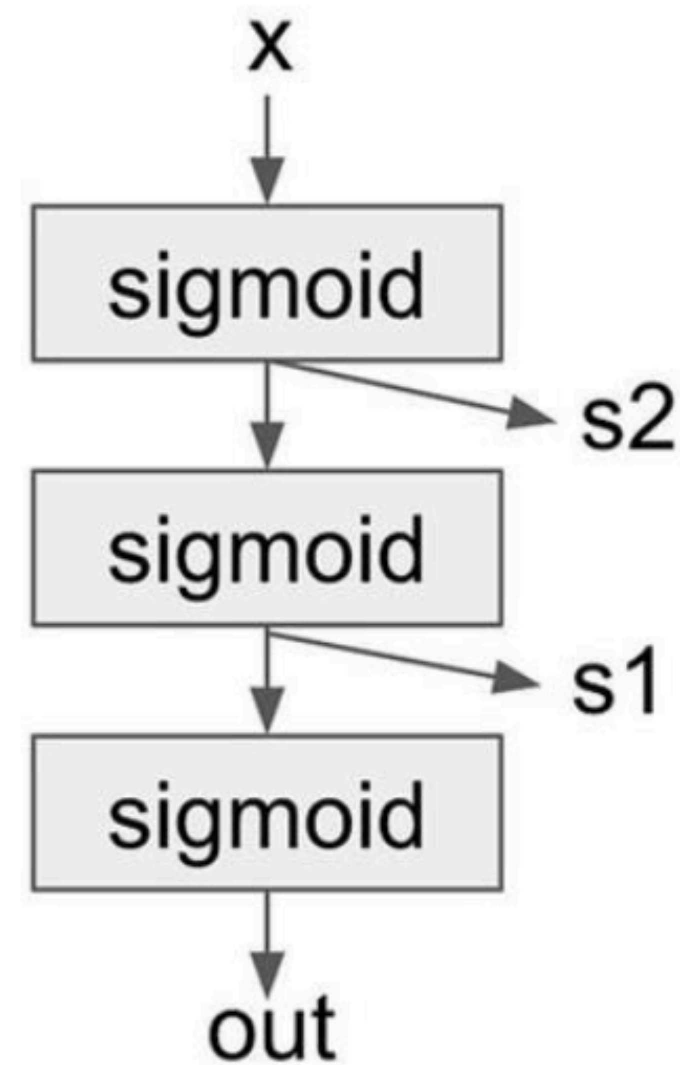



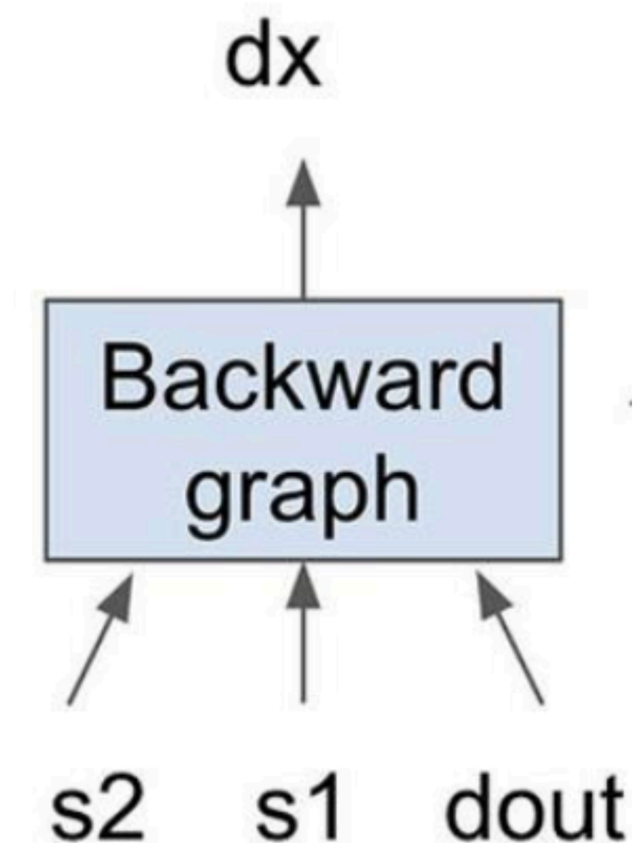

- In forwardprop, we store the activations (yellow)
- In backprop, using these, compute gradients (green)

# Trick 3: Recomputation

- Storing (and retrieving) activations can be expensive
- Let's say we stack 3 sigmoids on top of each other

Forward pass
- 1 mem read
- 3 mem writes



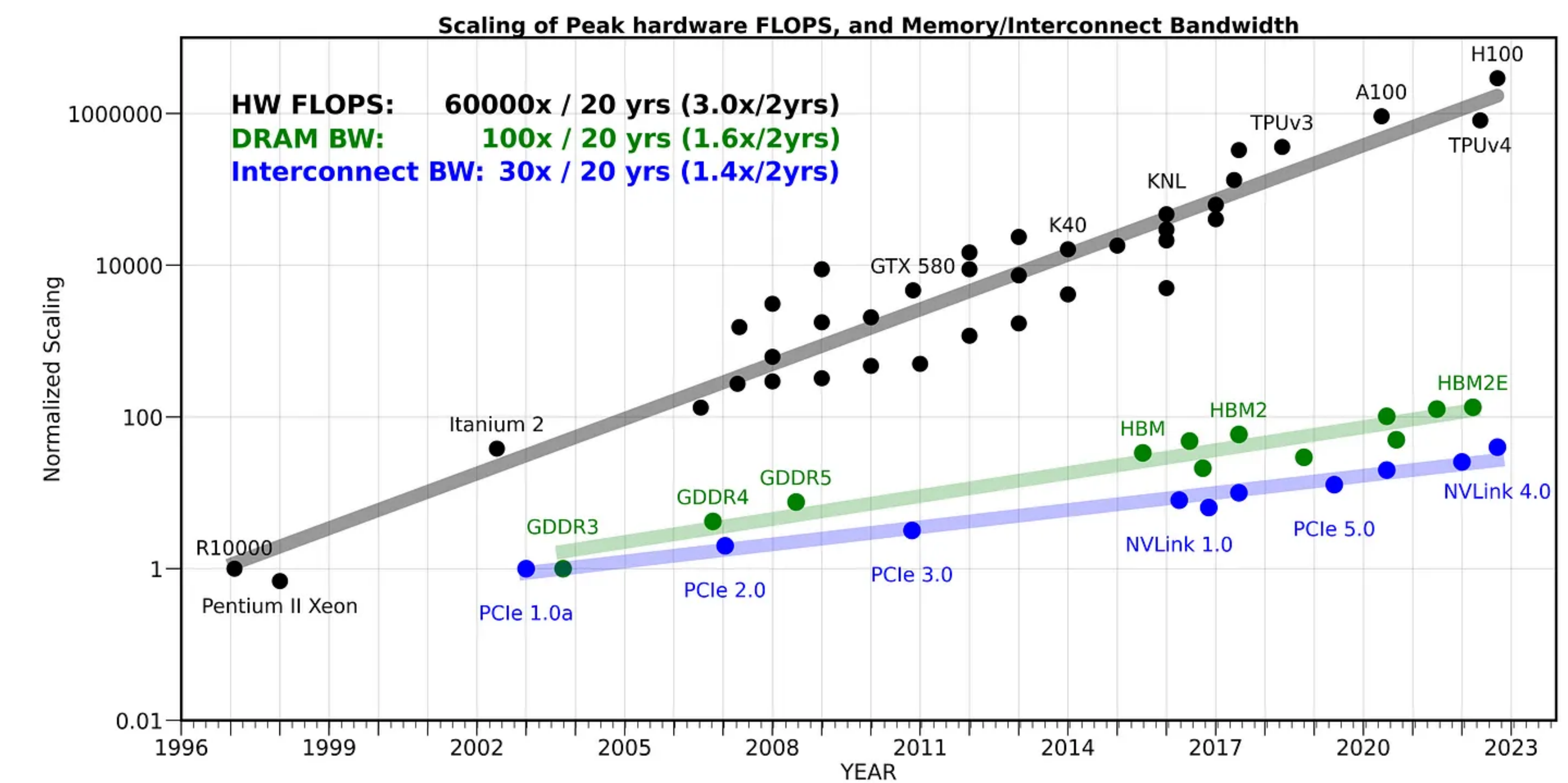



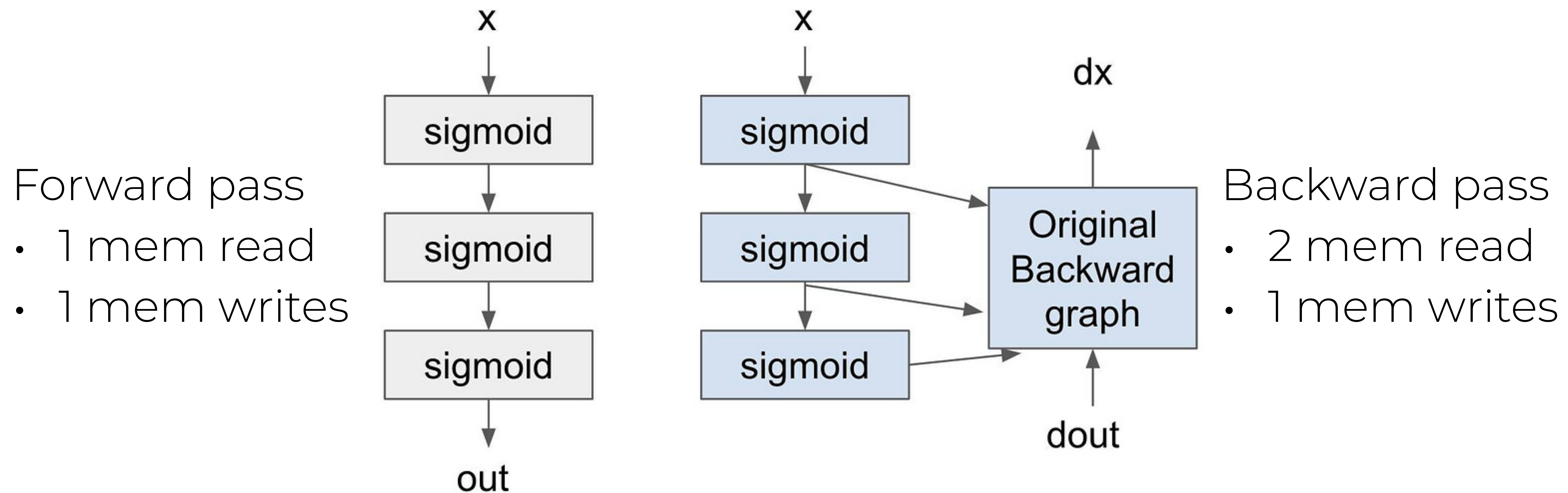

Backward pass
- 3 mem read
- 1 mem writes

- This is really terrible for performance
  - 8 memory access, very low arithmetic intensity

# Trick 3: Recomputation

- Compute is cheap, memory is limited!
- Throw away the activations, re-compute them

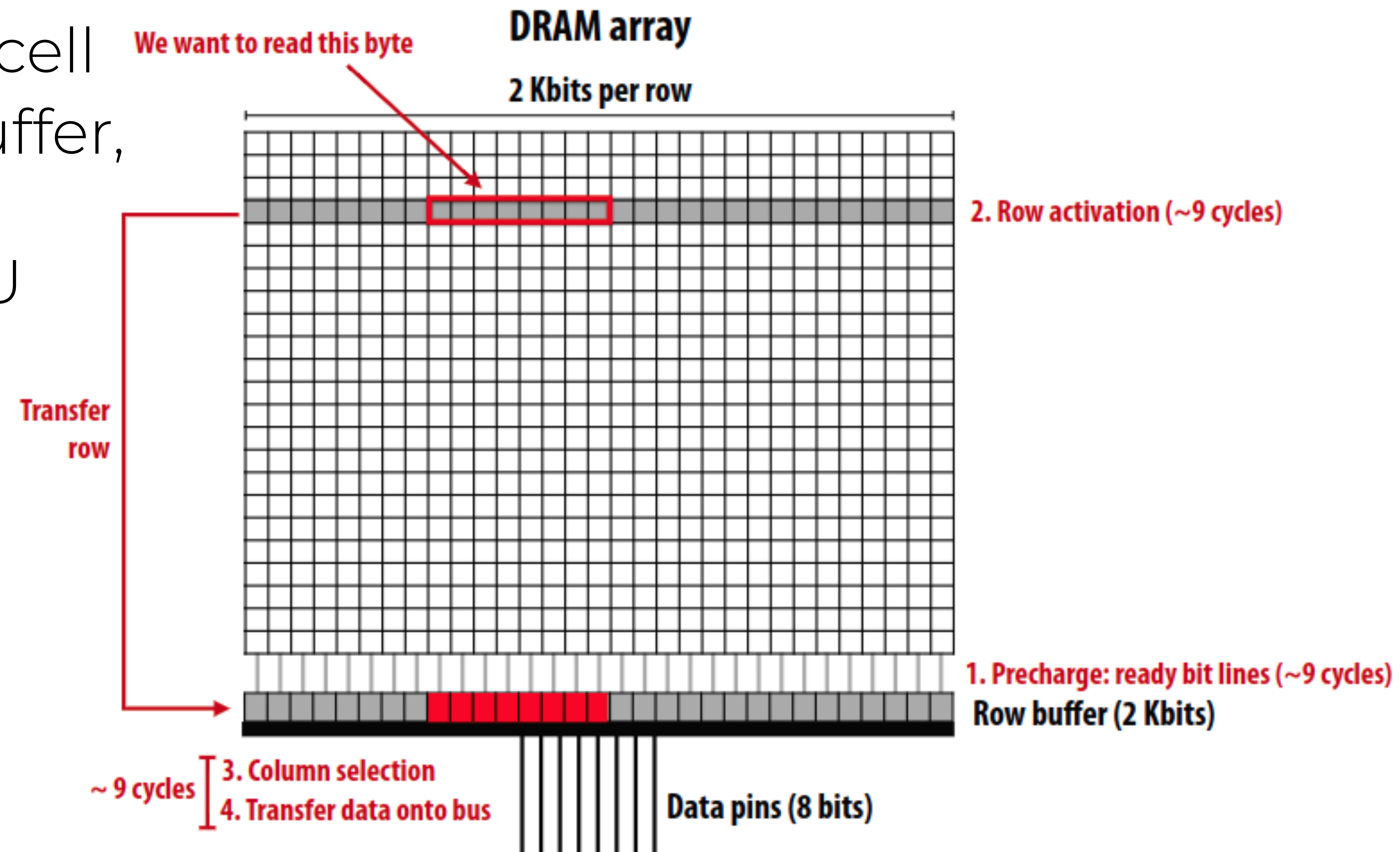



- Throwing away activations can actually be optimal, with 5/8 the memory accesses with extra computation

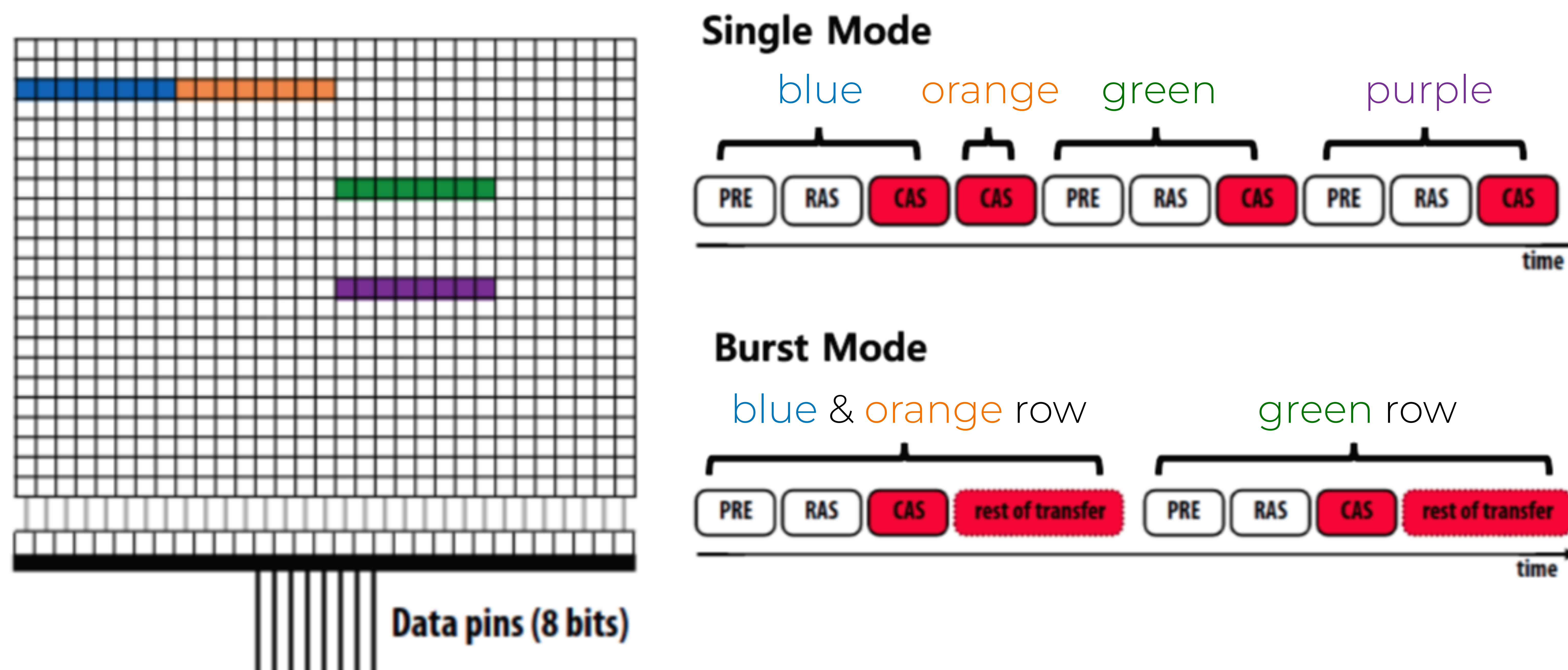
# Trick 4: Memory Coalescing

- DRAM array access
  - We cannot directly access cell
  - Instead, copy cell row to buffer, and select data from row buffer, then transfer to CPU



# Trick 4: Memory Coalescing

- DRAM (global memory) is read in burst mode
  - Even if you read only 1 byte, each read gives you many bytes
  - PRE and RAS are slow, so don't waste copied row data
  - PRE: pre-charge / RAS: row activate / CAS: column access



# Trick 4: Memory Coalescing

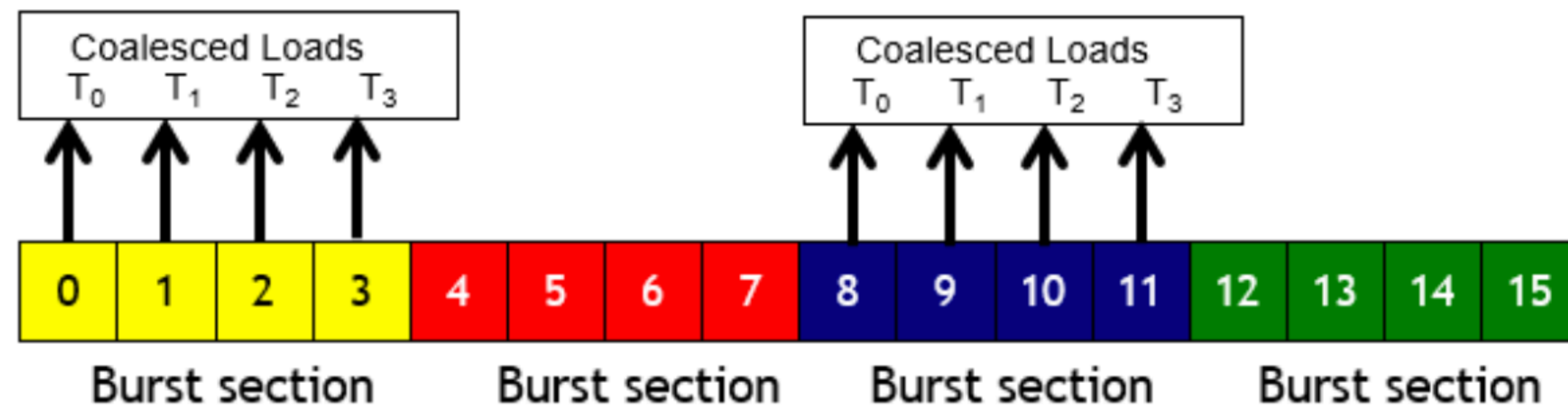
- DRAM (global memory) is read in burst mode
  - Even if you read only 1 byte, each read gives you many bytes

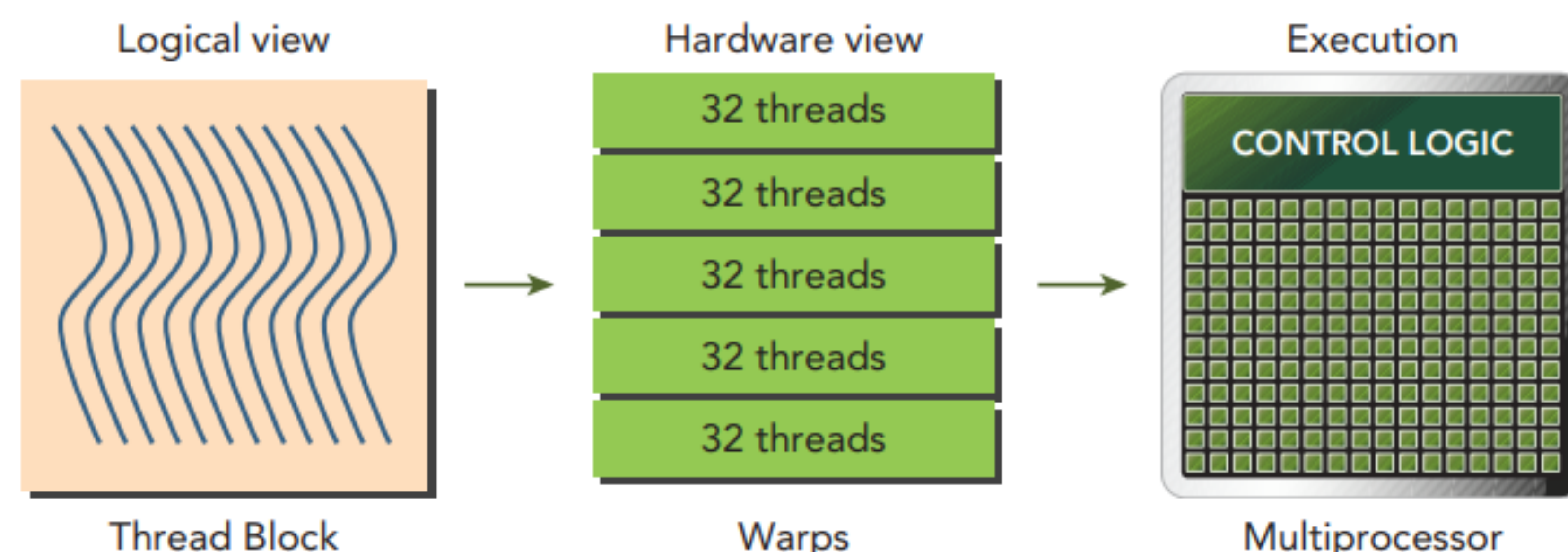
- Each address space is partitioned into burst sections
  - Whenever a location is accessed, all other locations in the same section are also delivered to the processor
- Basic example: a 16-byte address space, 4-byte burst sections
  - In practice, we have at least 4GB address space, burst section sizes of 128-bytes or more

https://blog.csdn.net/xll_bit

# Trick 4: Memory Coalescing

- Mem acc. are coalesced if all threads (in a warp) are in the same burst
  - With a single mem r/w, can get data for multiple threads

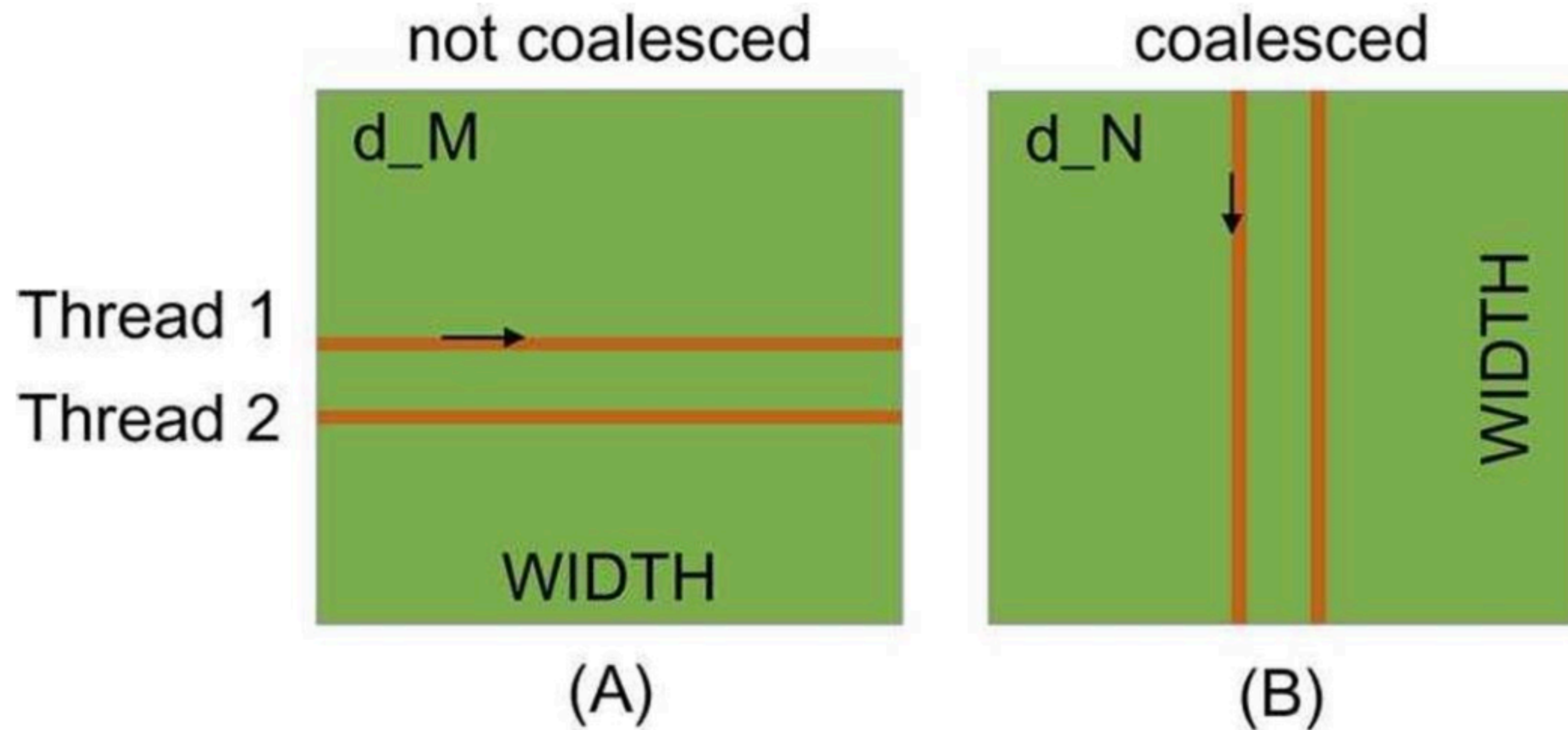


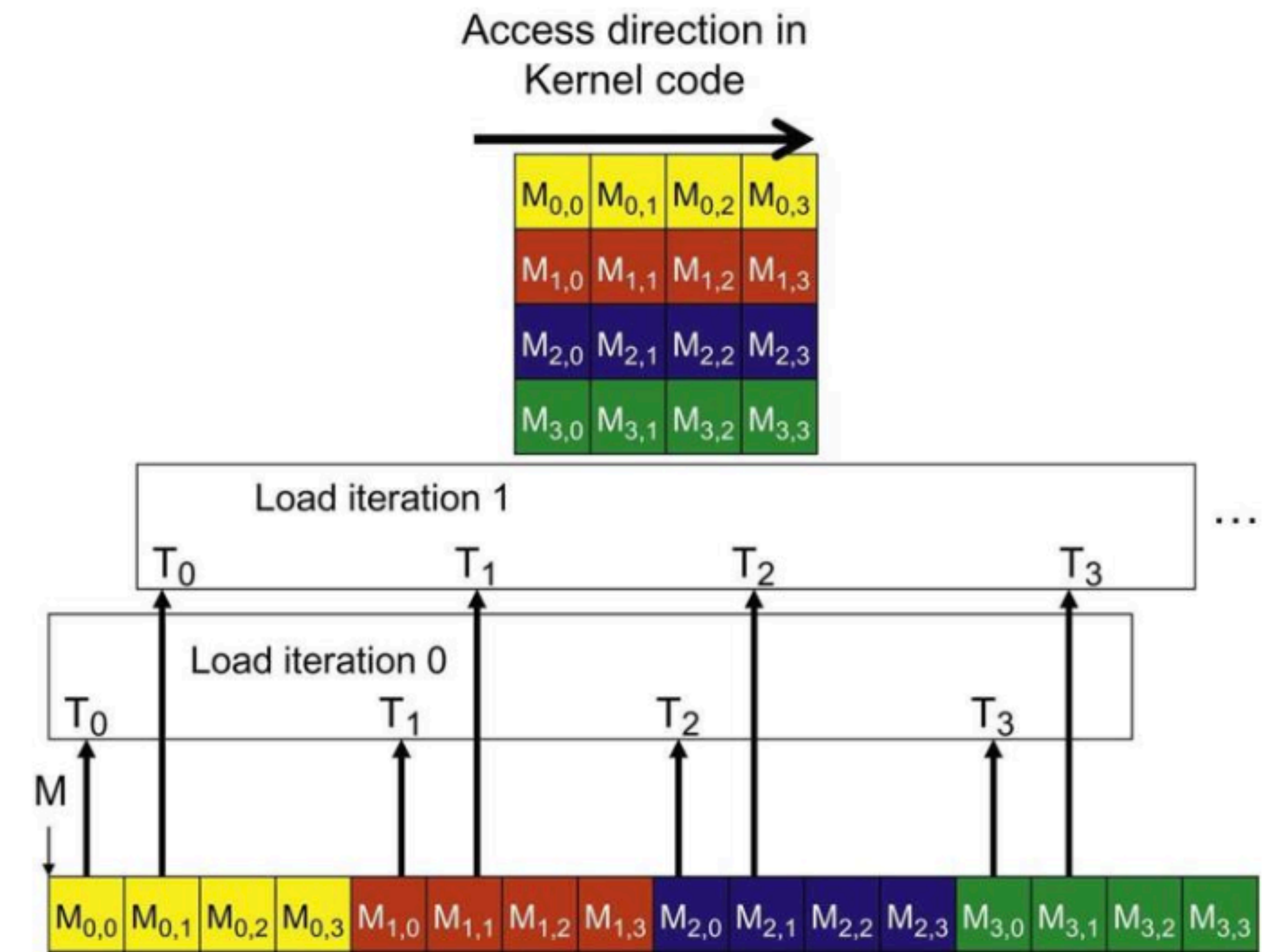




Reminder: a warp is a set of 32 consecutively numbered threads that execute together in a block. Memory accesses happen together

# Trick 4: Memory Coalescing

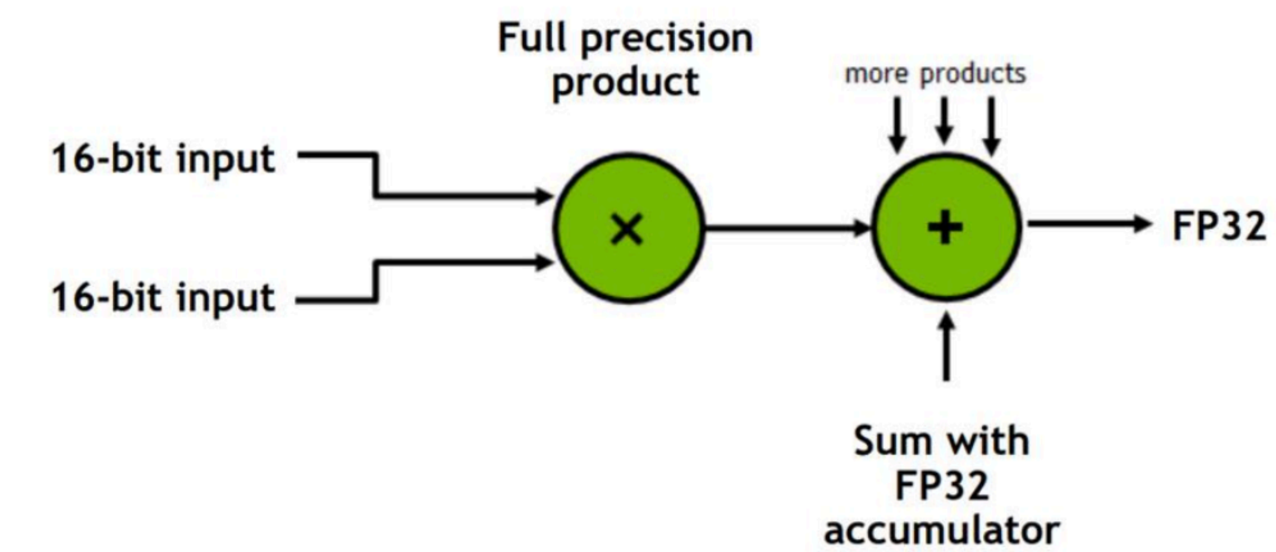
- Coalescing for matmul
  - For row-major, threads that move along rows are not coalesced
  - Note how the right diagram reads the entire vector at each step!

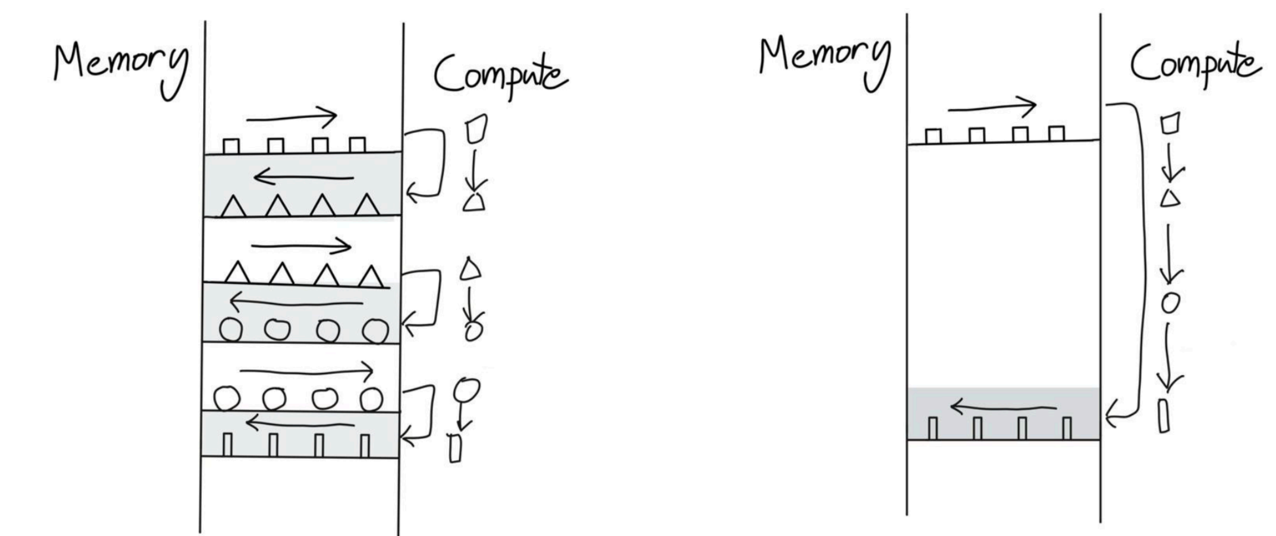


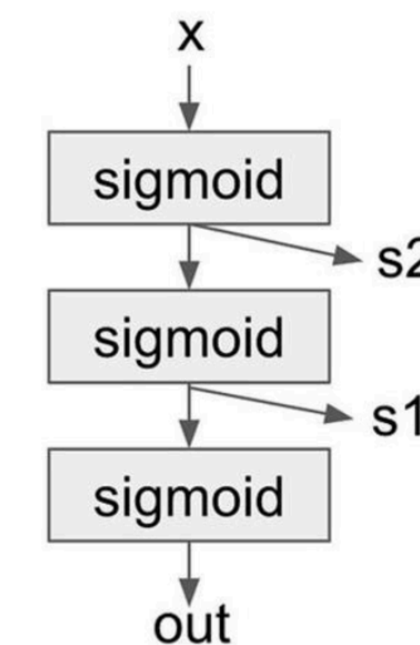



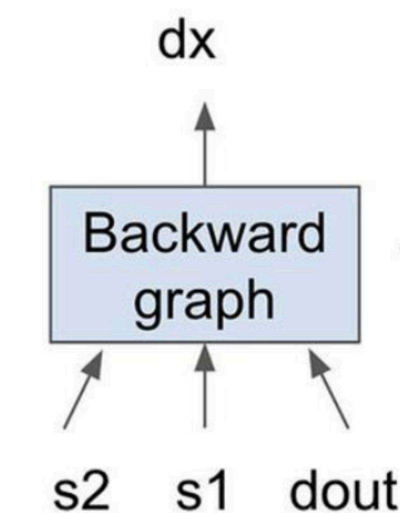
# Recap

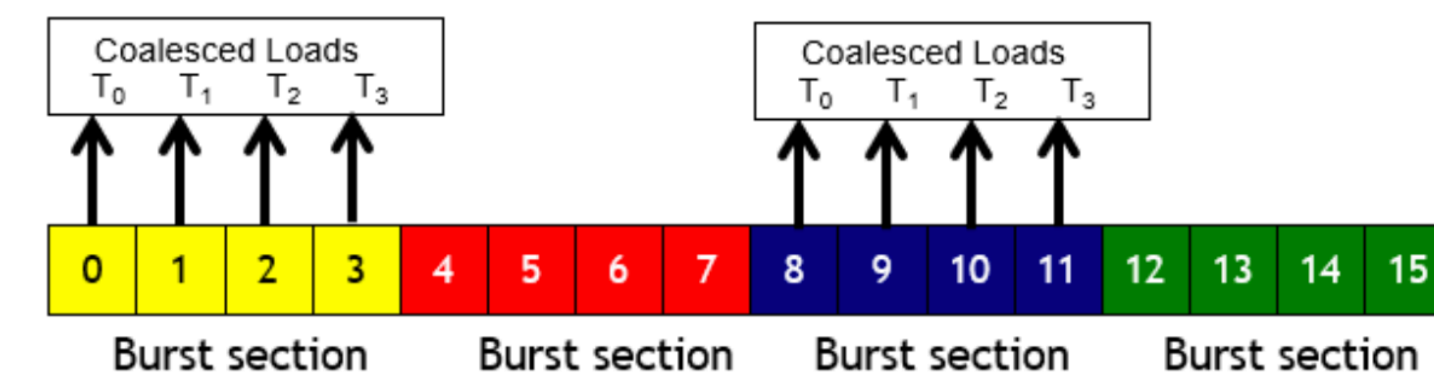
- Low precision computation
  - Shrink bits to boost arithmetic intensity
- Kernel fusion
  - Fuse ops to skip slow HBM round-trip
- Recomputation
  - Trade FLOPs to save VRAM capacity
- Coalescing memory
  - Read data smart by exploiting burst



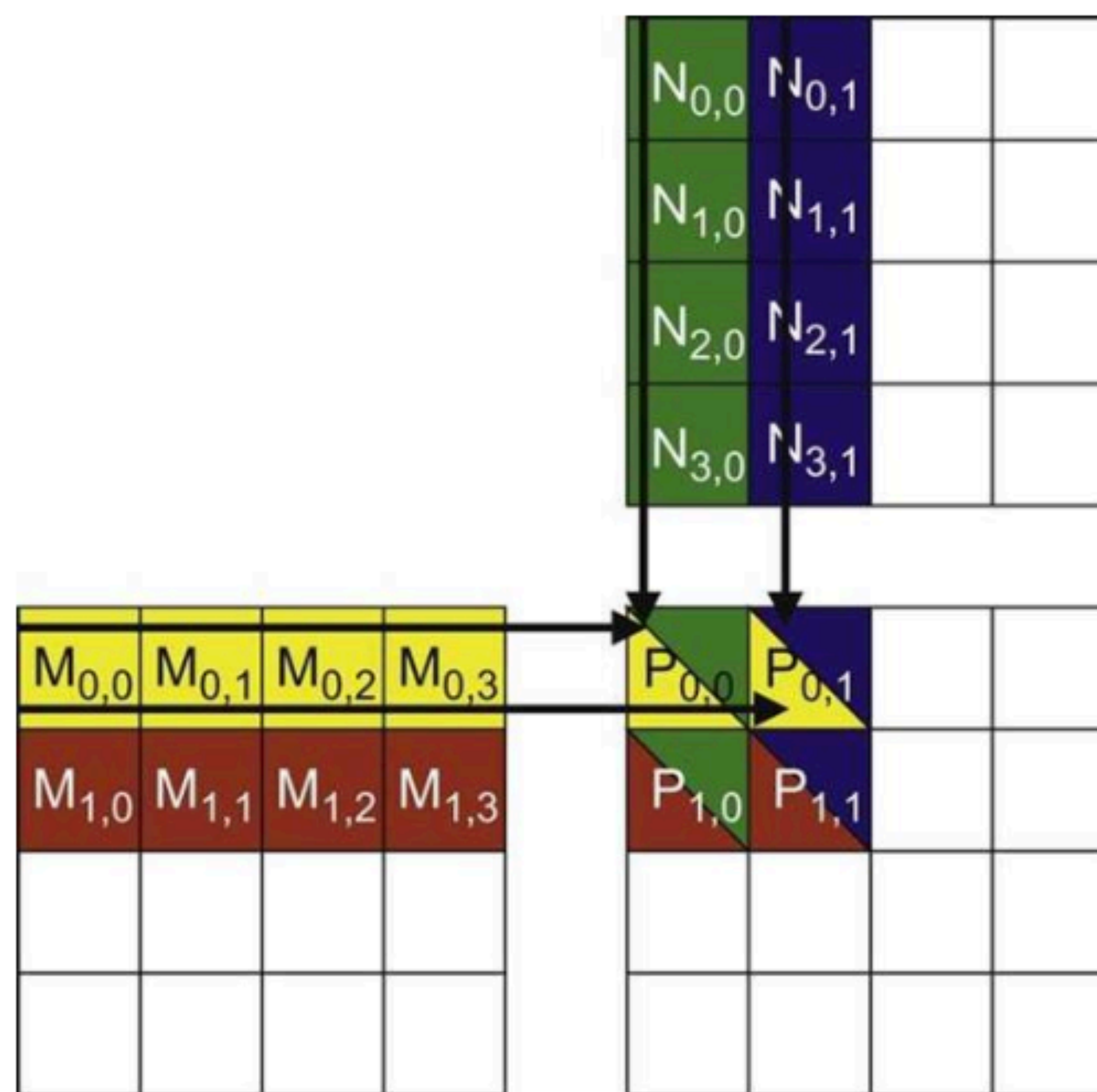



# Trick 5: Tiling

- Let's go back to matmul
  - Some elems are repeatedly accessed from HBM (e.g. $M_{0,0}$, $N_{1,0}$)

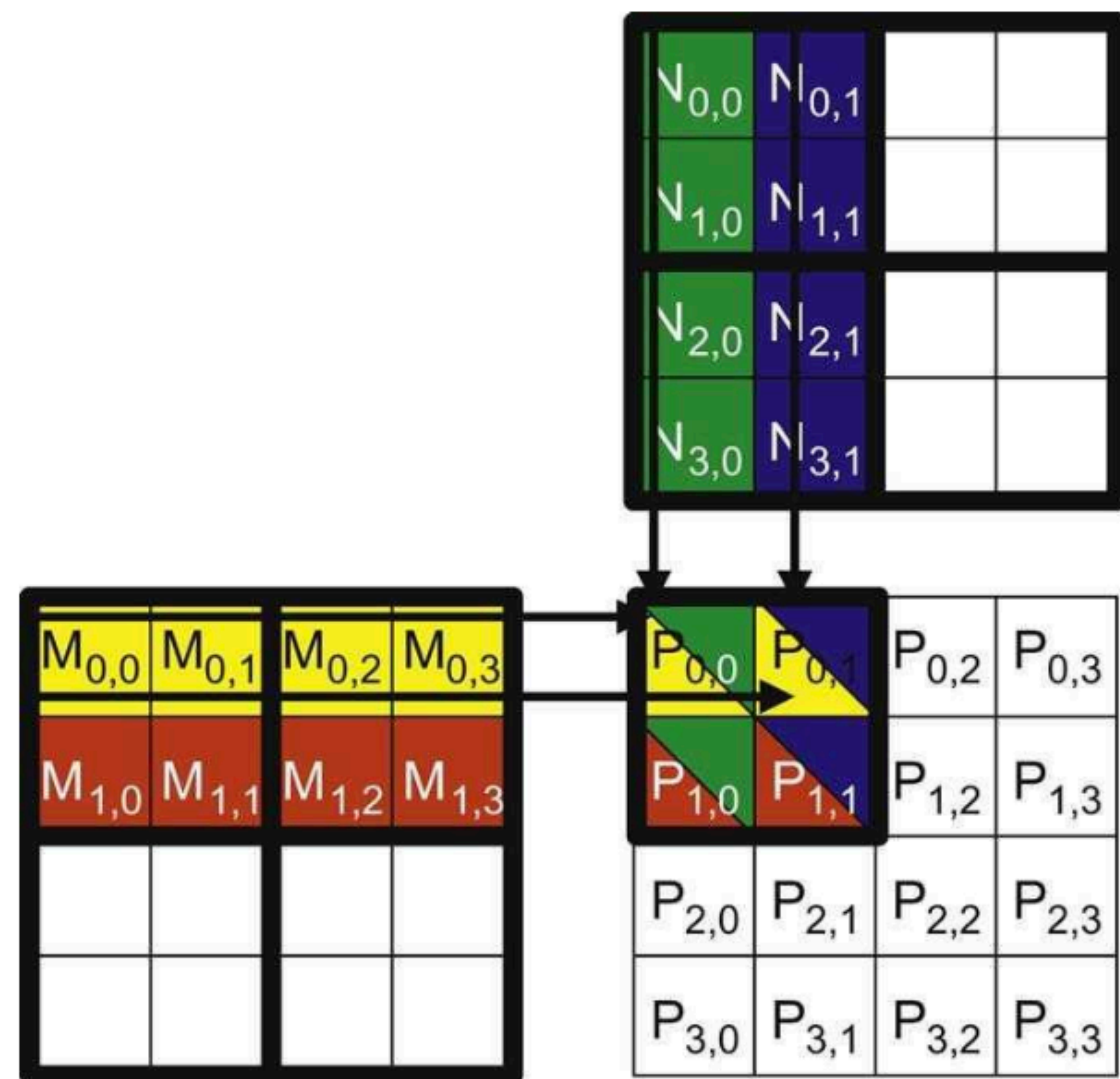



Access order →

| | | | | |
|---|---|---|---|---|
| $thread_{0,0}$ | $M_{0,0} * N_{0,0}$ | $M_{0,1} * N_{1,0}$ | $M_{0,2} * N_{2,0}$ | $M_{0,3} * N_{3,0}$ |
| $thread_{0,1}$ | $M_{0,0} * N_{0,1}$ | $M_{0,1} * N_{1,1}$ | $M_{0,2} * N_{2,1}$ | $M_{0,3} * N_{3,1}$ |
| $thread_{1,0}$ | $M_{1,0} * N_{0,0}$ | $M_{1,1} * N_{1,0}$ | $M_{1,2} * N_{2,0}$ | $M_{1,3} * N_{3,0}$ |
| $thread_{1,1}$ | $M_{1,0} * N_{0,1}$ | $M_{1,1} * N_{1,1}$ | $M_{1,2} * N_{2,1}$ | $M_{1,3} * N_{3,1}$ |

# Trick 5: Tiling

- Tiling is the idea of grouping threads to minimize global mem access
  - Cut up the matrix into smaller tiles, and load this into shared mem

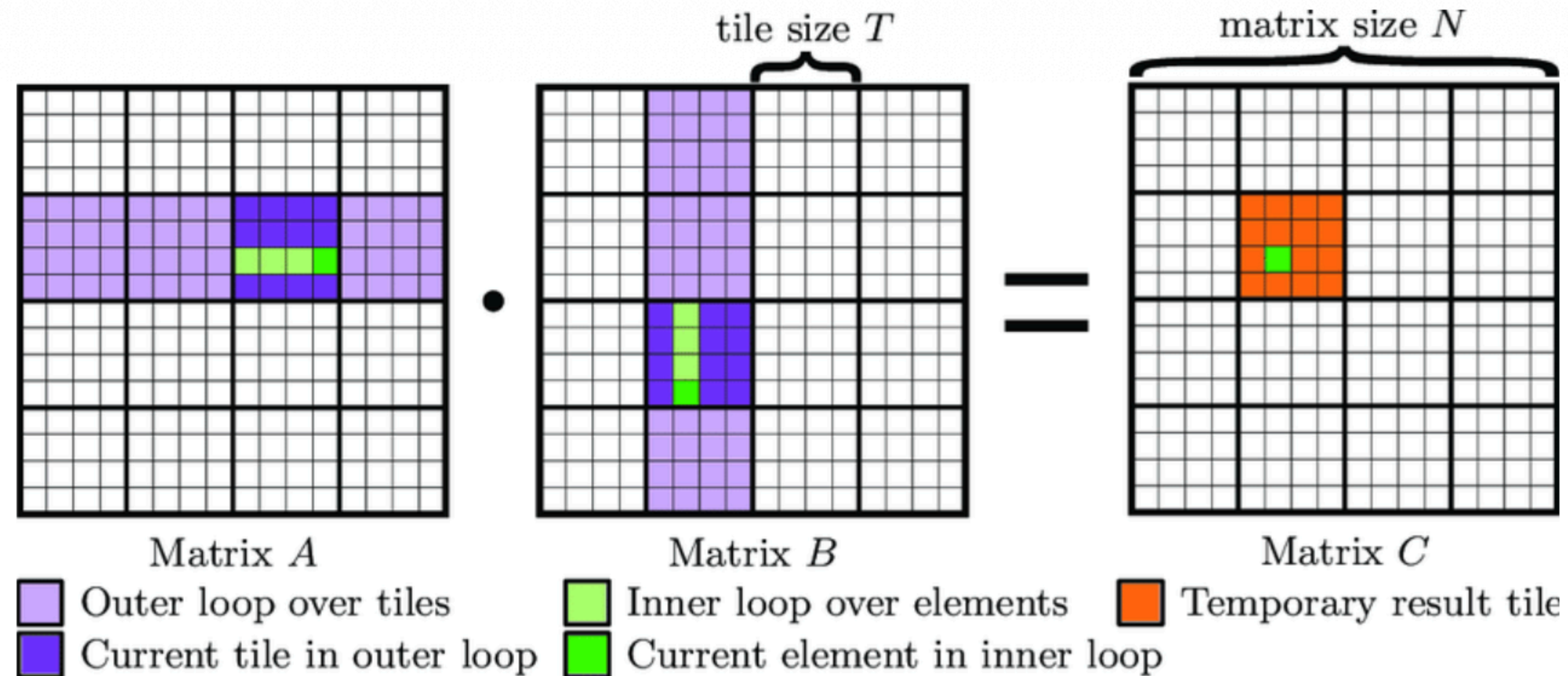



Compute matmul in phases

1. Load $M_{0,0}$ and $N_{0,0}$ tiles into shared memory
2. Compute partial sums for P
3. Load $M_{0,0}$ and $N_{2,0}$ tiles into shared memory
4. …

- Advantages: repeated reads now access shared memory, not global memory and memory access can be coalesced

# Trick 5: Tiling

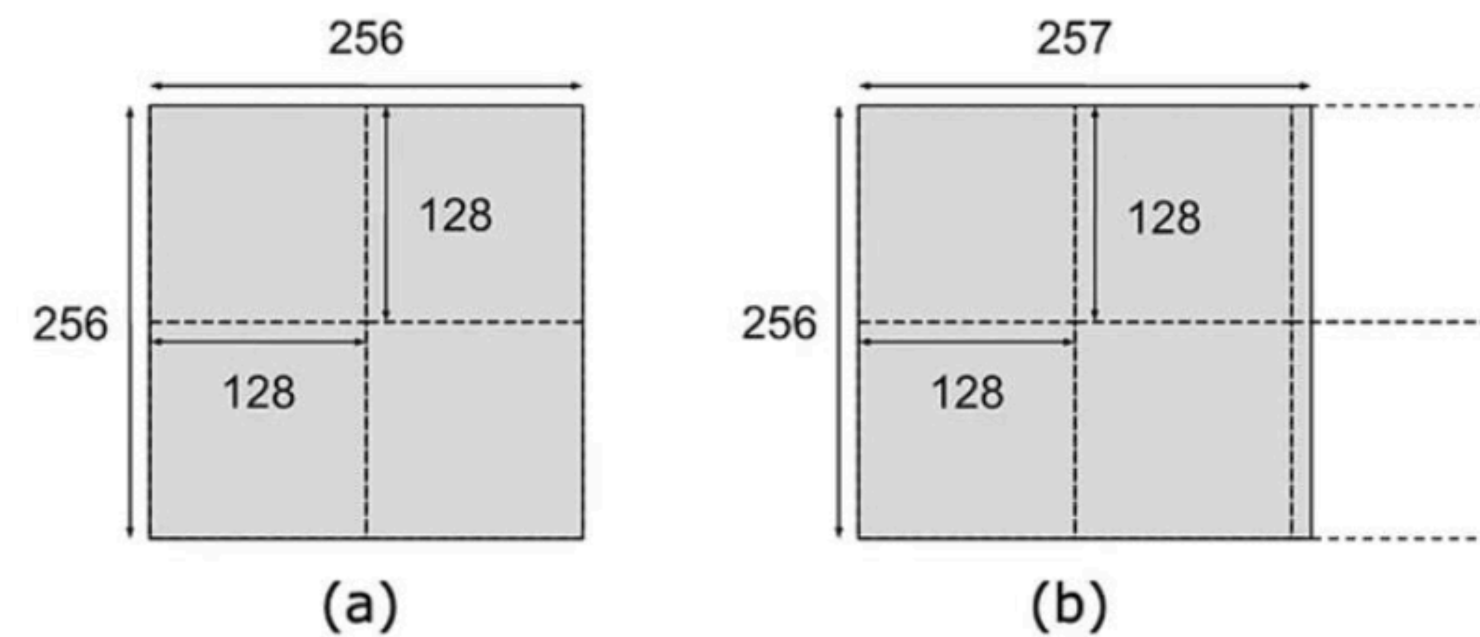



- Non-tiled matmul: each input is read N times from global memory
- Tiled matmul: each input is read N/T times from global memory, and T times within each tile (a factor of T reduction in global memory access)

# Trick 5: Tiling

- Complexities with tiling
  - Tile sizes may not divide the matrix size and lead to low utilization

*Figure 6. Example of tiling with 128x128 thread block tiles. (a) Best case - matrix dimensions are divisible by tile dimensions (b) Worse case - tile quantization results in six thread blocks being launched, two of which waste most of their work.*

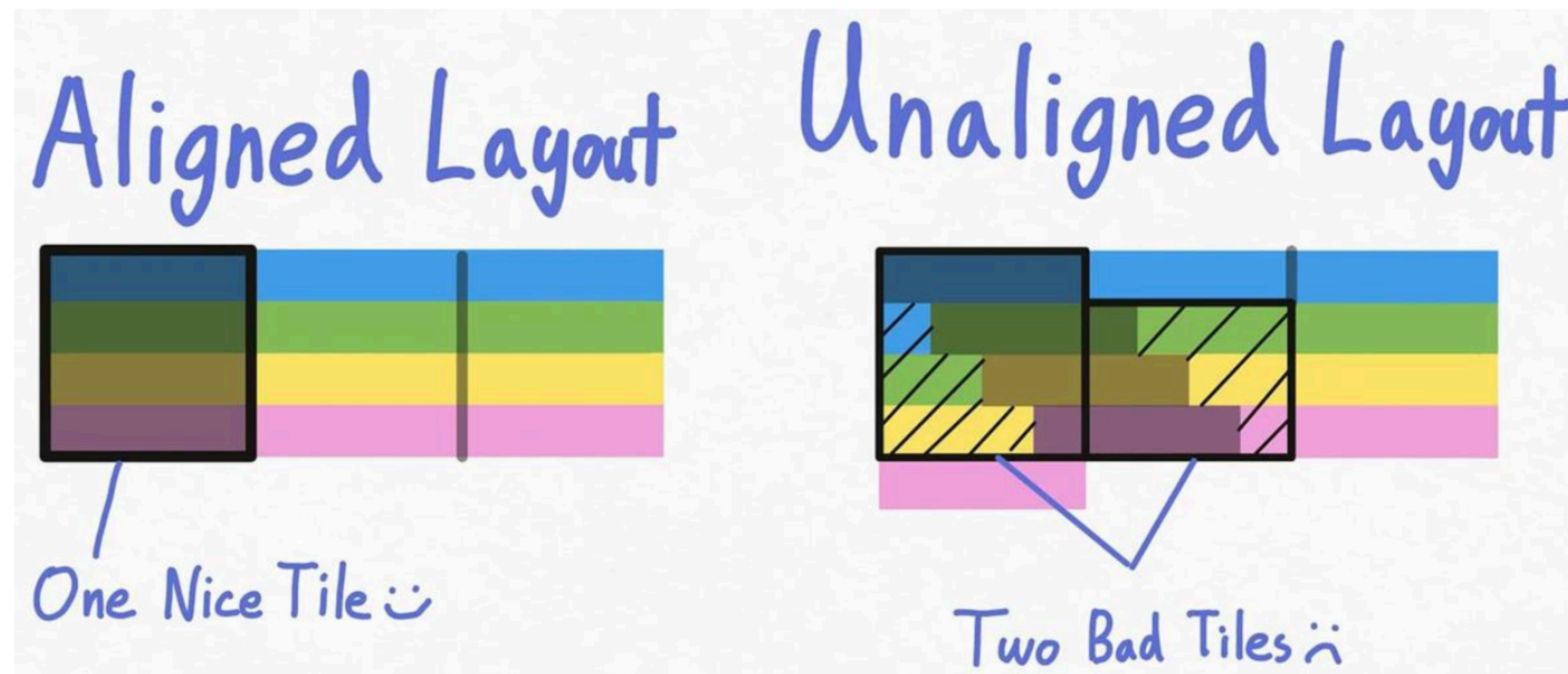



  - Factors affecting tile sizes
    - Coalesced memory access
    - Shared memory size
    - Divisibility of the matrix dim

# Trick 5: Tiling

- Complexities with tiling
  - Memory comes in bursts

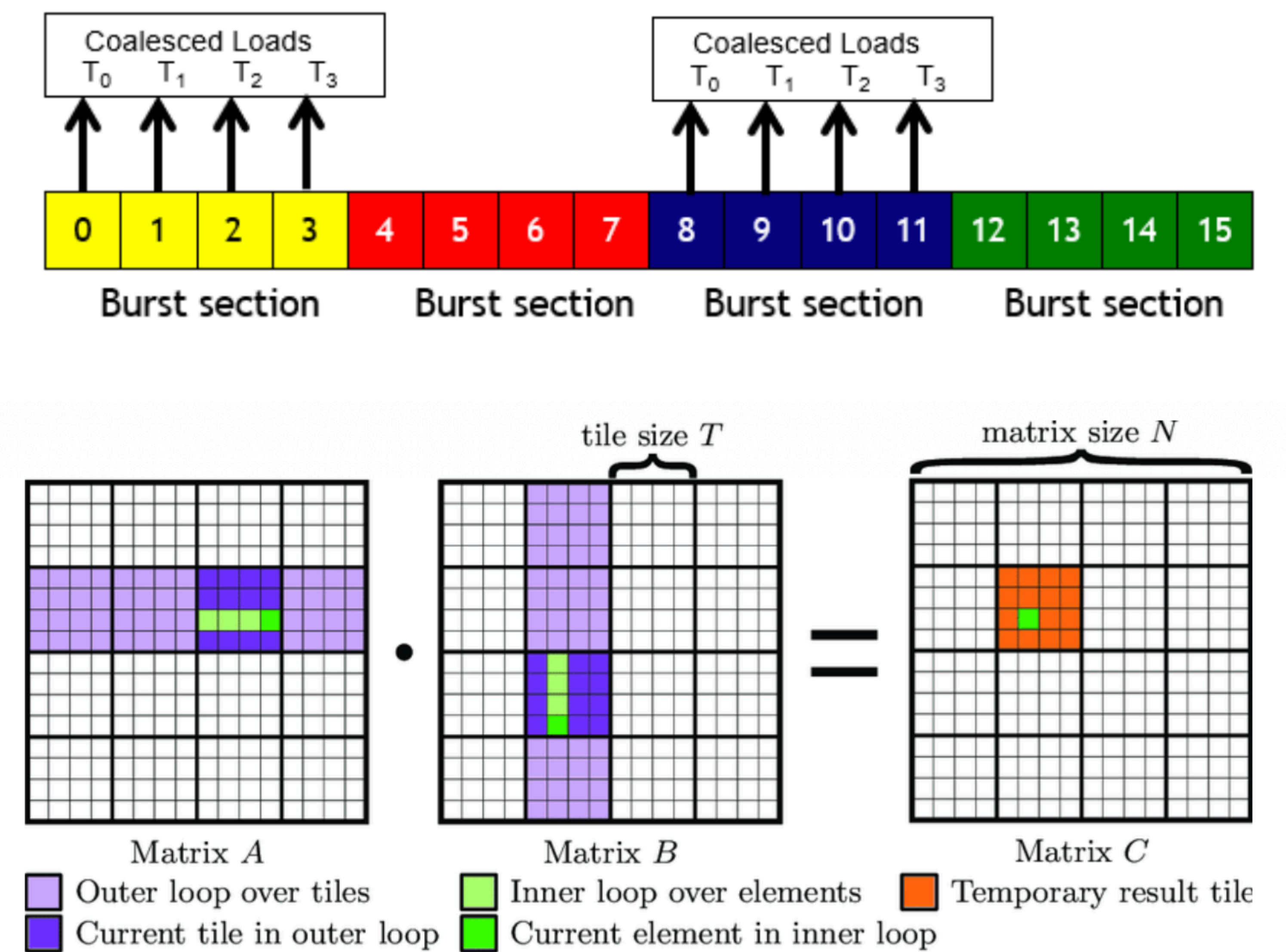
  - Loading tiles are fast if bursts align with the matrix




  - Coalesced accesses may be impossible depending on matrix dims

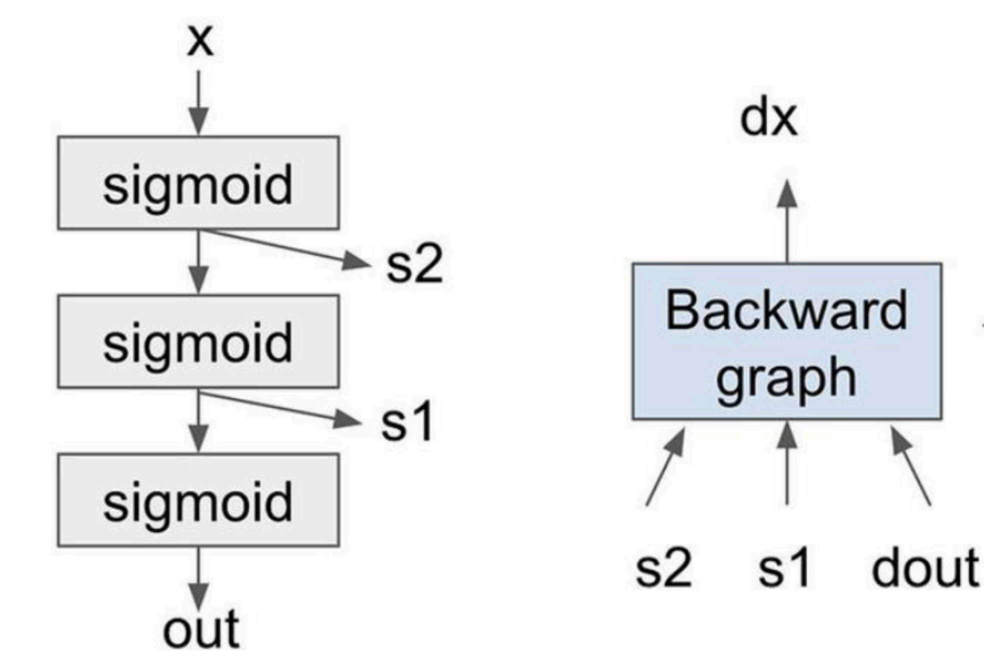
# Recap of GPU Performance

- Reducing memory accesses
  - Coalescing (get many data at once)
  - Kernel fusion (reduce data movement)



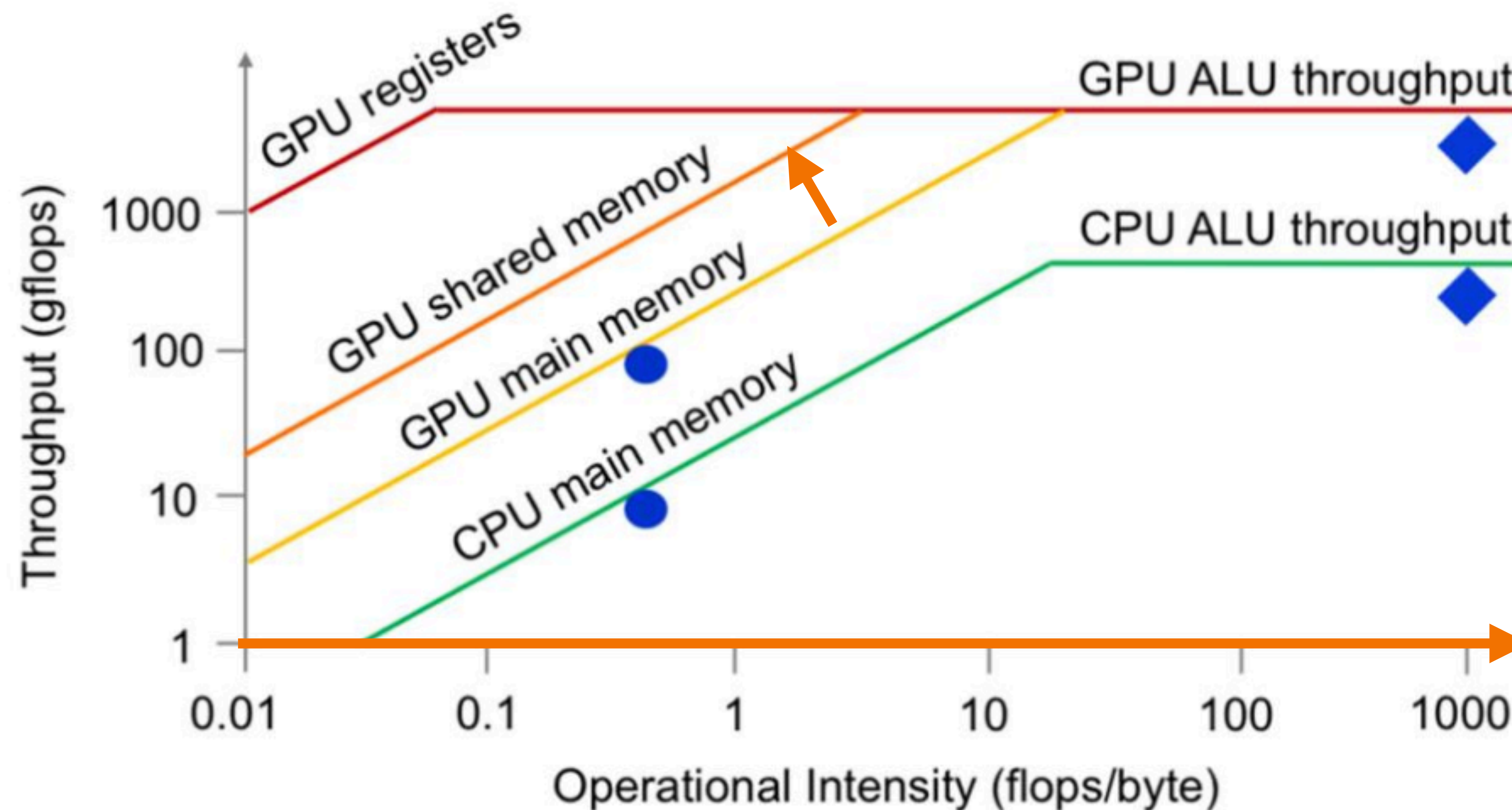

- Move memory to shared memory
  - Tiling (try my best to keep in shared mem)

- Trade memory for compute/accuracy
  - Quantization (trade memory and accuracy)
  - Recomputation (trade memory and compute)



# The Goal of 5 Tricks

- Move our operations to the upper roofline
- Move our operations to the higher x-axis (increased intensity)



# Understanding Matrix Mystery

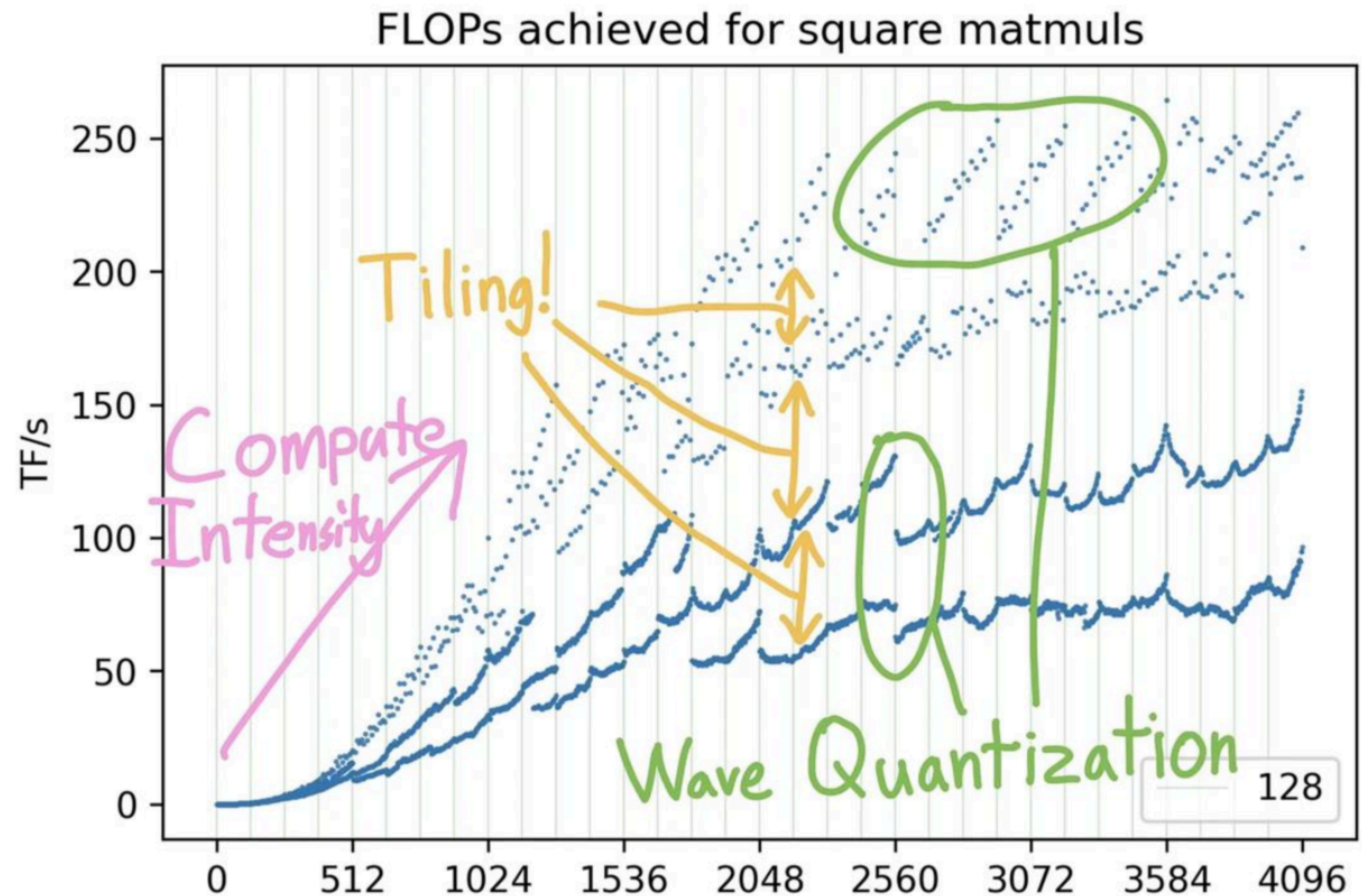
**Andrej Karpathy**
@karpathy

The most dramatic optimization to nanoGPT so far (~25% speedup) is to simply increase vocab size from 50257 to 50304 (nearest multiple of 64). This calculates added useless dimensions but goes down a different kernel path with much higher occupancy. Careful with your Powers of 2.

10:36 AM · Feb 3, 2023 · **1.2M** Views

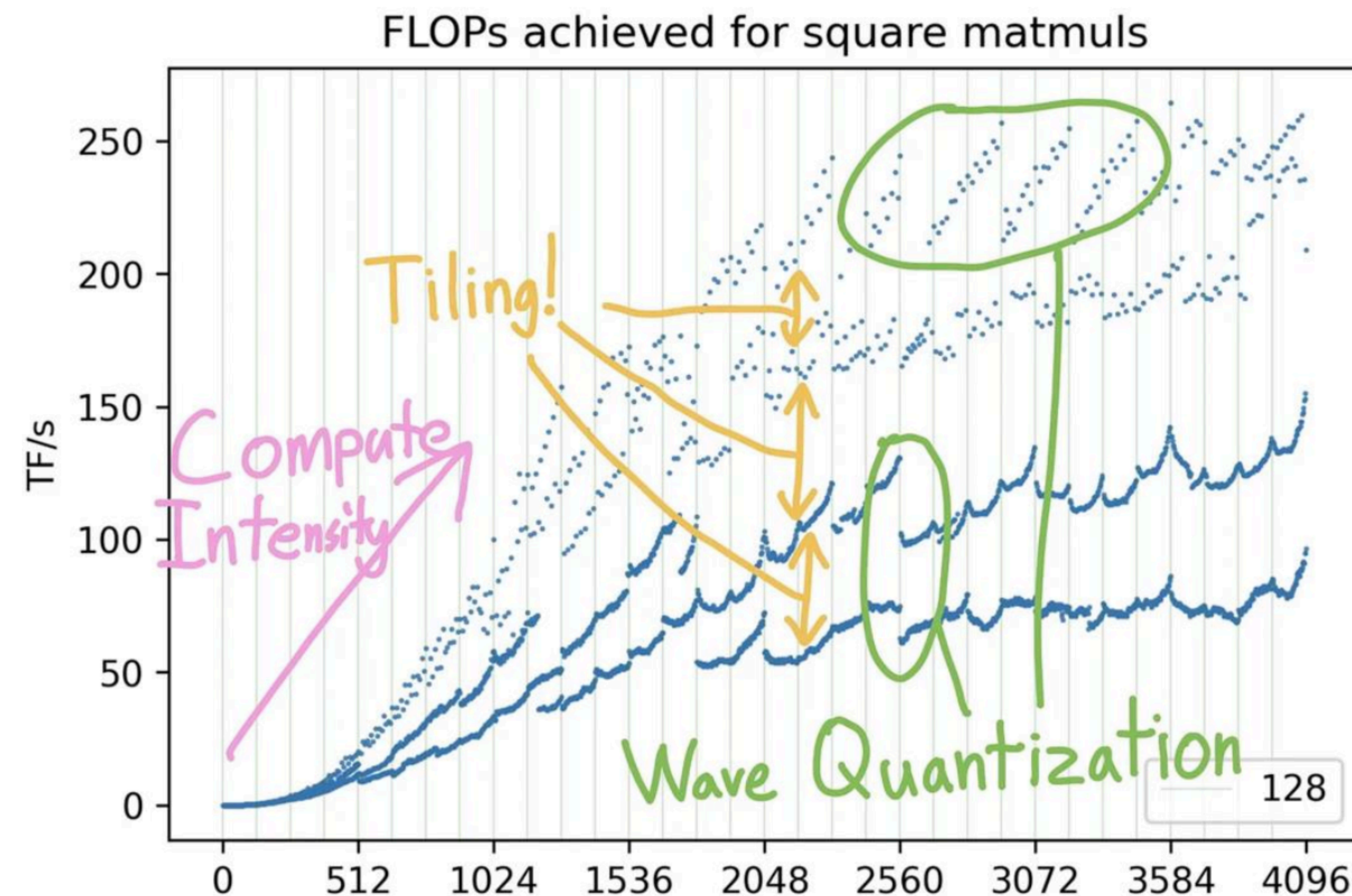
- Why is it faster to have bigger matrices?

# Understanding Matrix Mystery



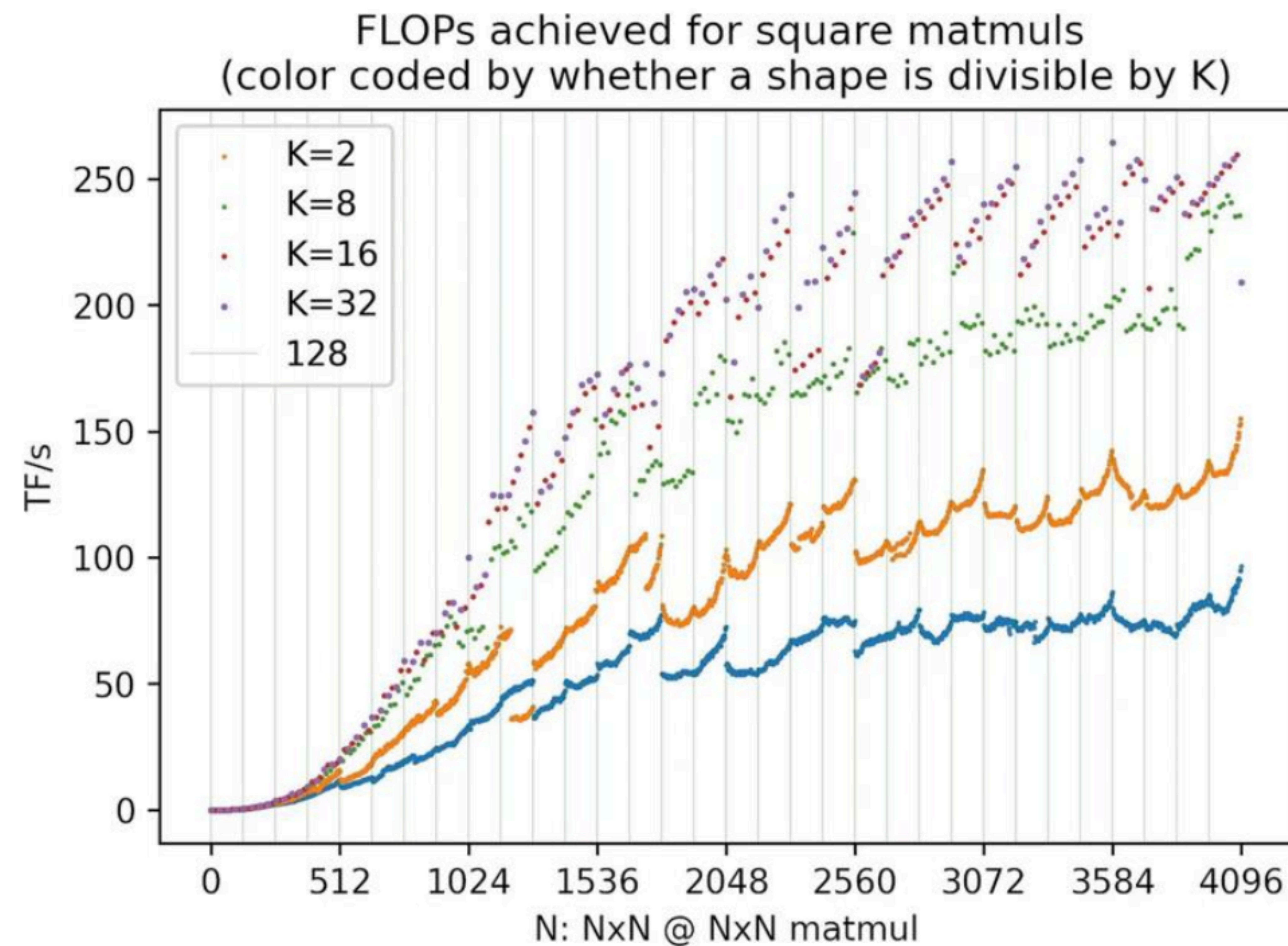
# Understanding Matrix Mystery

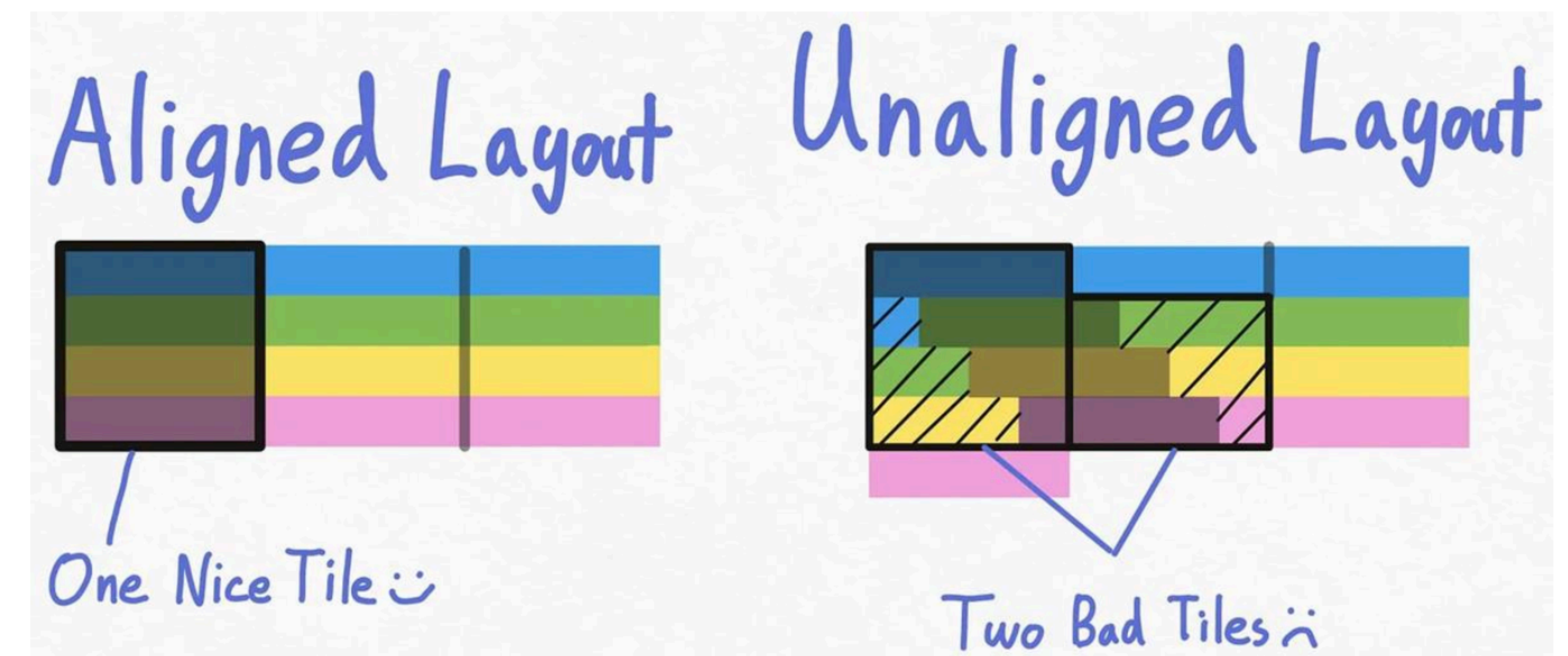
- 1. Compute intensity
  - Increasing matrix size → increasing arithmetic intensity



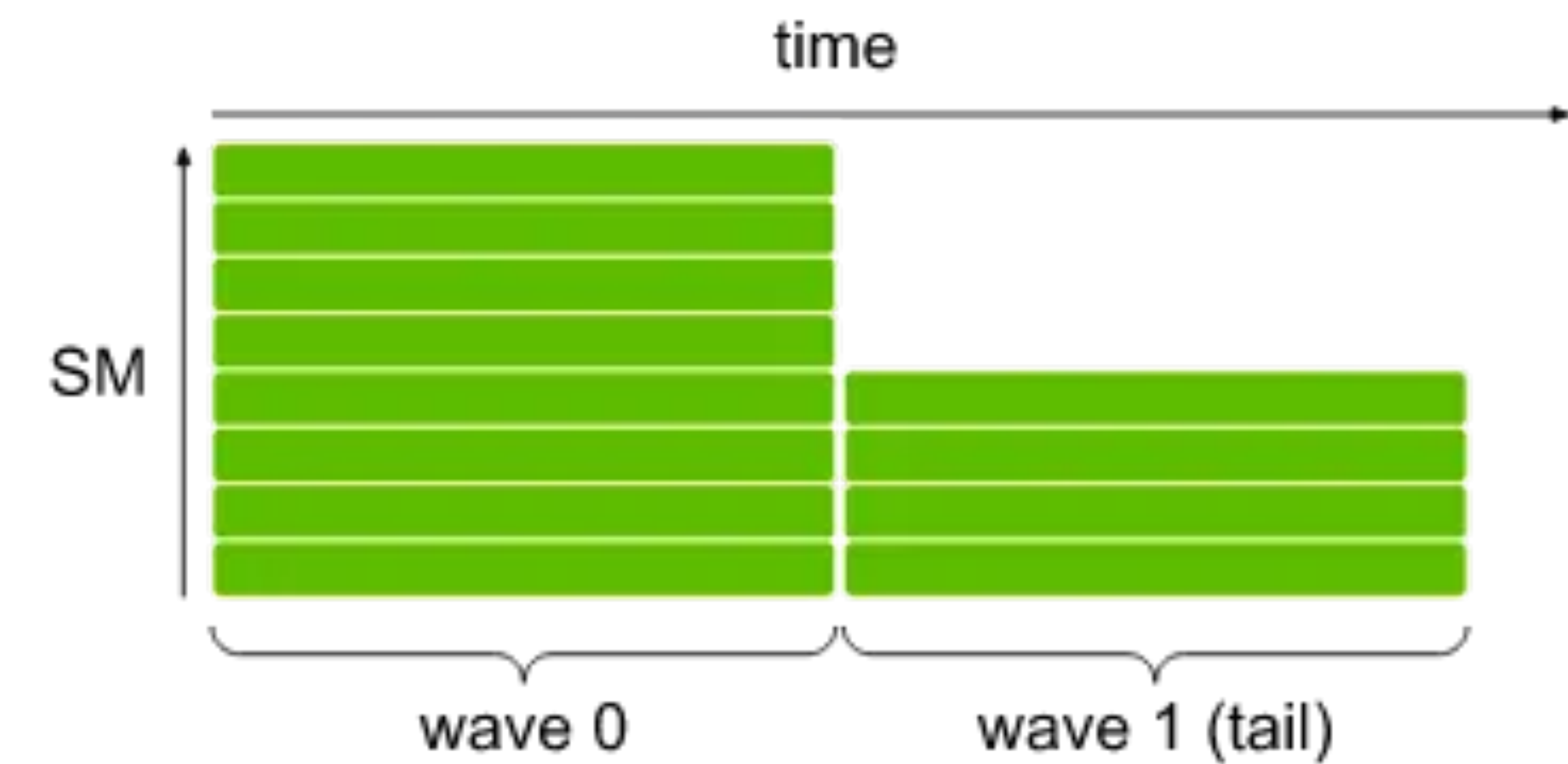
# Understanding Matrix Mystery

- 2. Tiling (with burst mode)
  - If K=2, very few tiles are aligned / if K = 32, most of tiles are aligned



# Understanding Matrix Mystery




- 3. Wave quantization
  - What's with the periodic behavior?

In this case, it happens at N = 1792 to 1793

- Why? Using a tile size of 256 x 128, there are (1792/256) x (1792/128) = 7x14 = 98 tiles
- If we increase N to 1793, we have 8x15 = 120 tiles
- An A100 has 108 SMs, so it cannot execute all 120

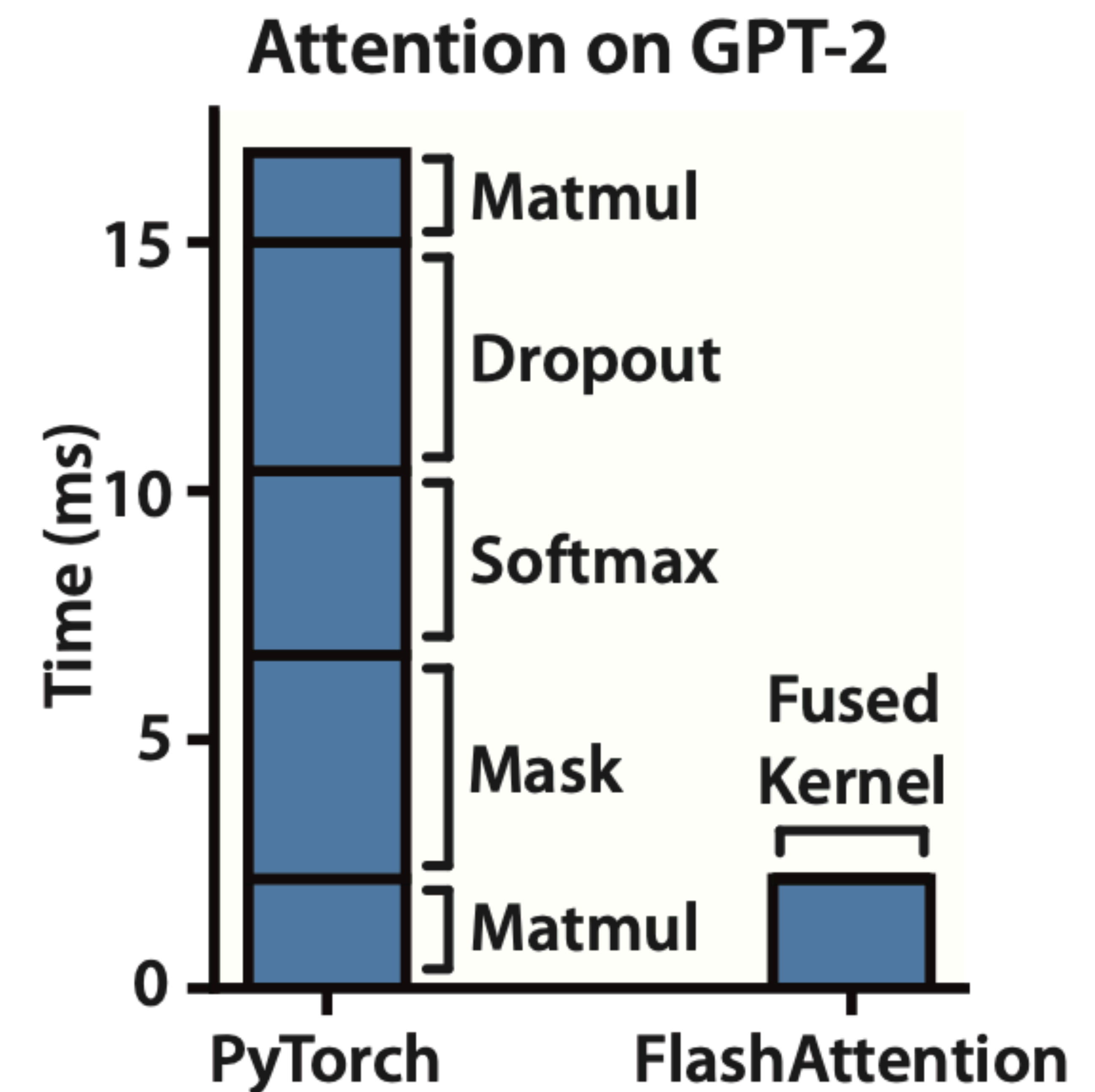
# FlashAttention

# FlashAttention [Dao+ 2022]

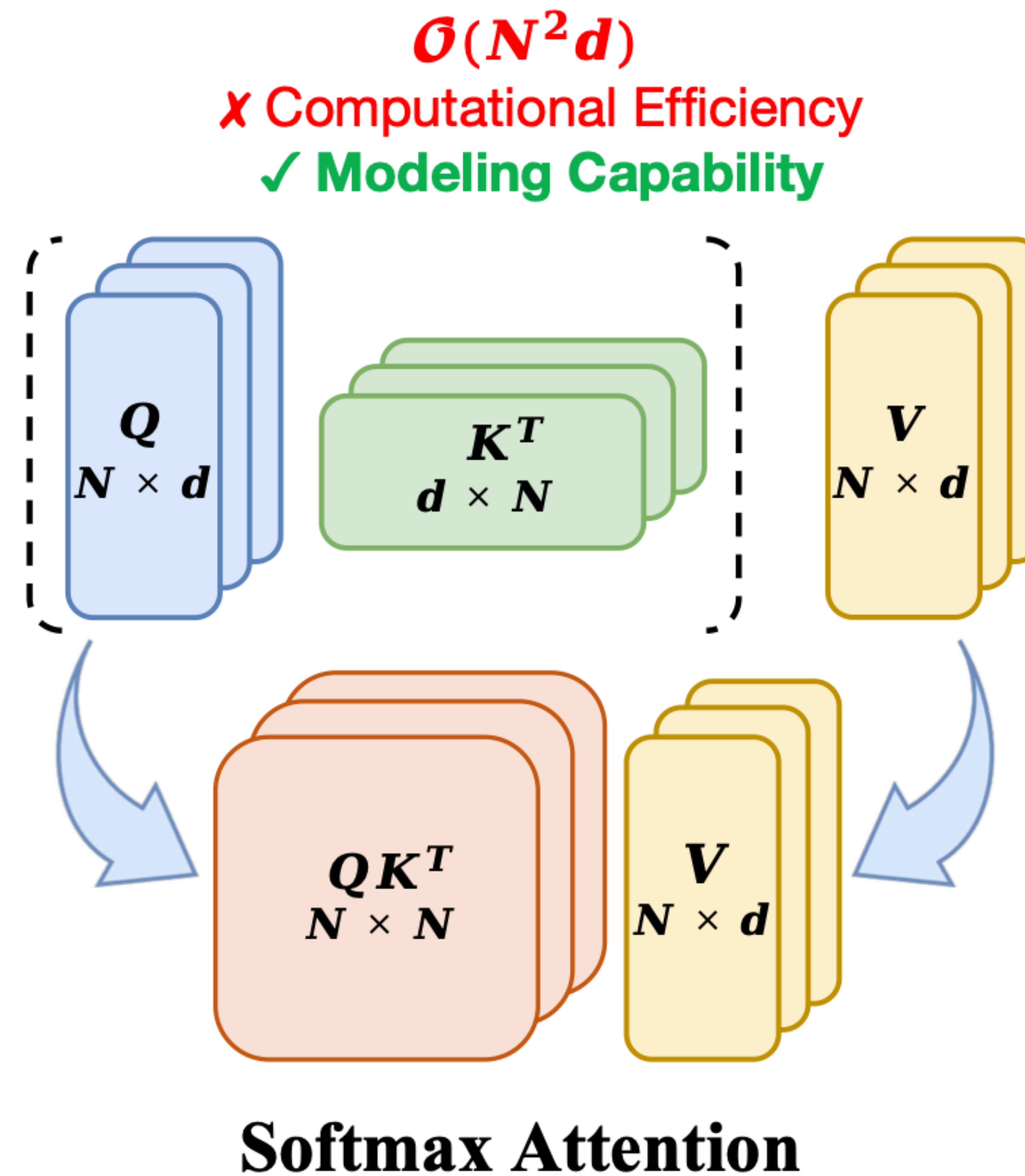
- Fast and Memory-efficient Exact Attention with IO-Awareness
  - Tiling
  - Kernel fusion
  - Recomputation

| Attention | Standard | FlashAttention |
|---|---|---|
| GFLOPs | 66.6 | 75.2 |
| HBM R/W (GB) | 40.3 | 4.4 |
| Runtime (ms) | 41.7 | 7.3 |

More FLOPs but less runtime
due to the low HBM access

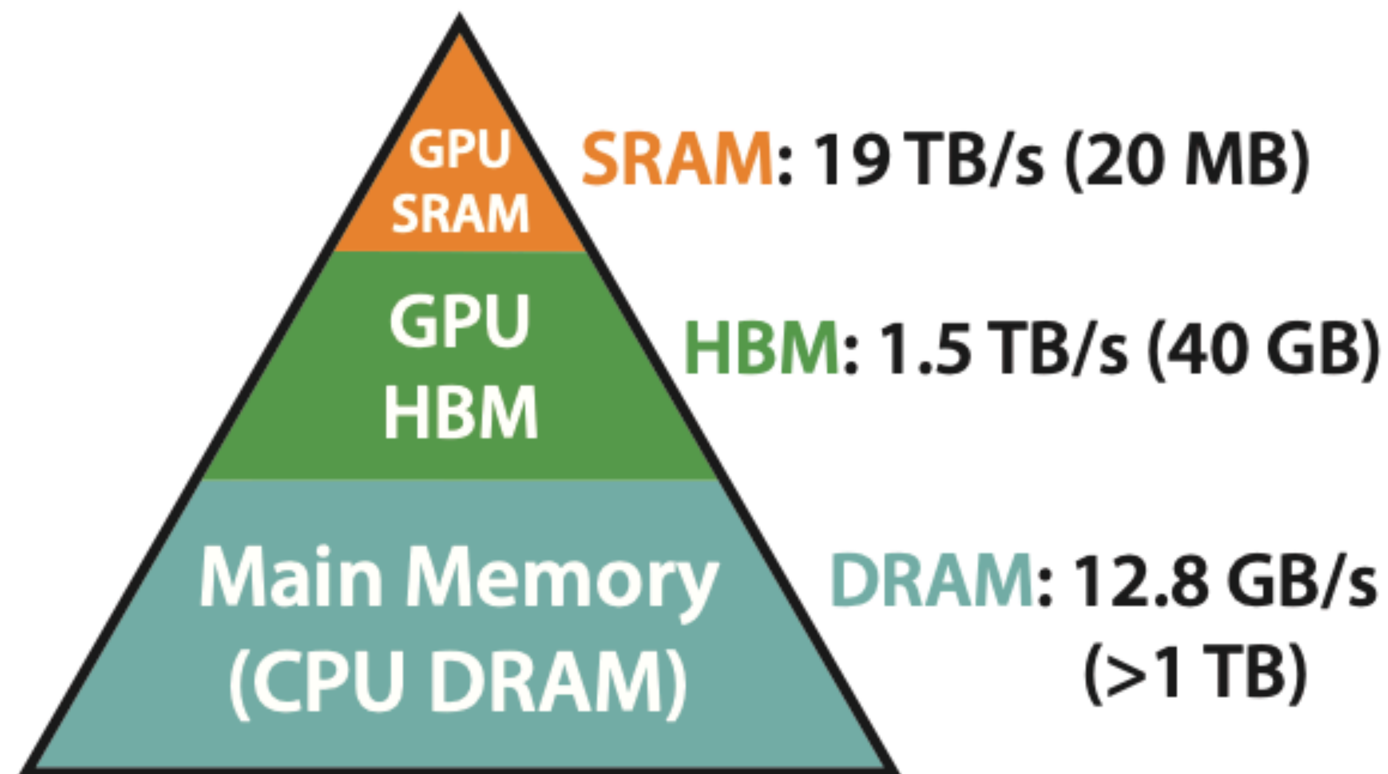


# Recap: Attention Computation

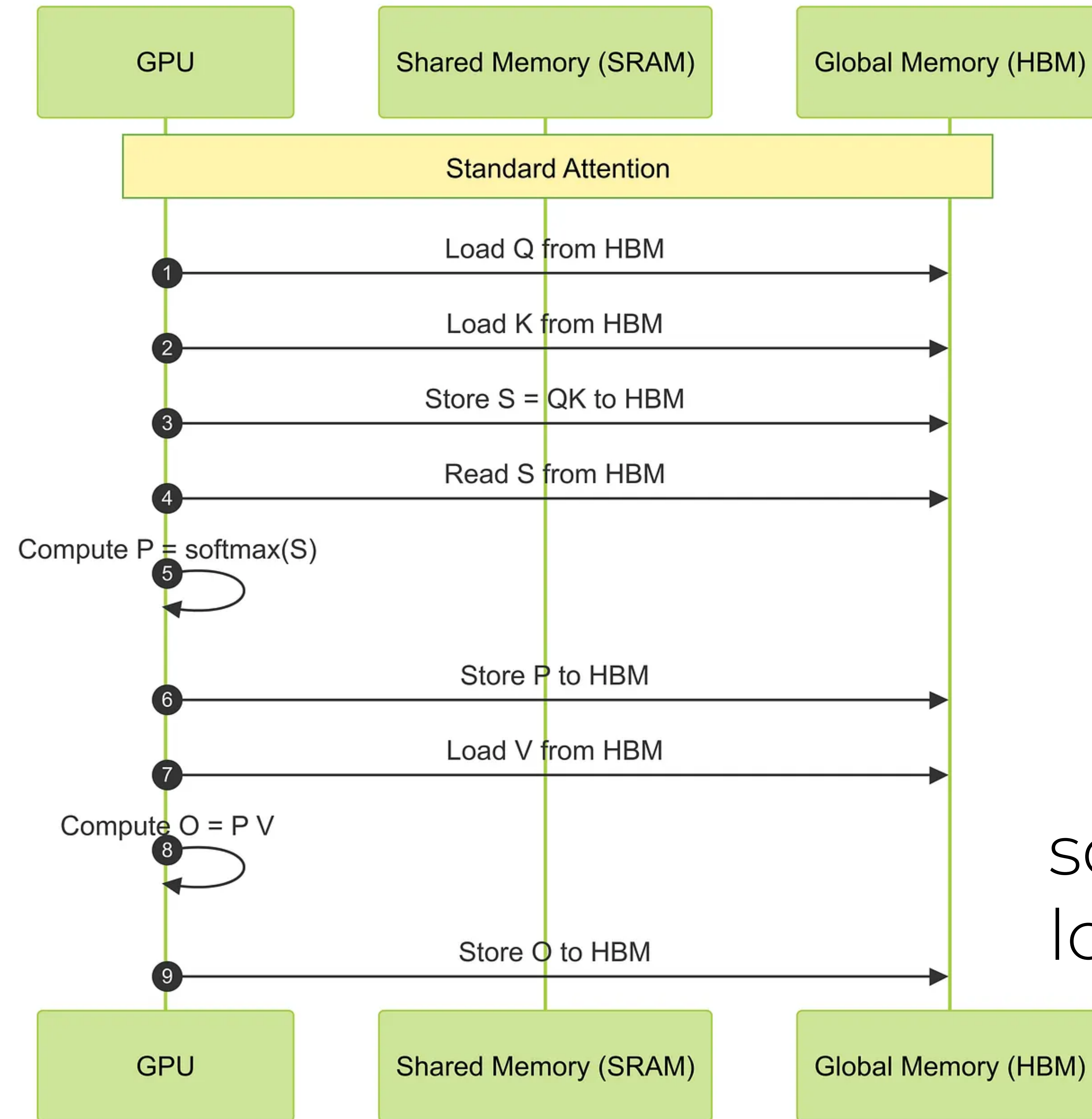


# Attention: Communication

https://medium.com/@najeebkan/flashattention-one-two-three-6760ad030ae0



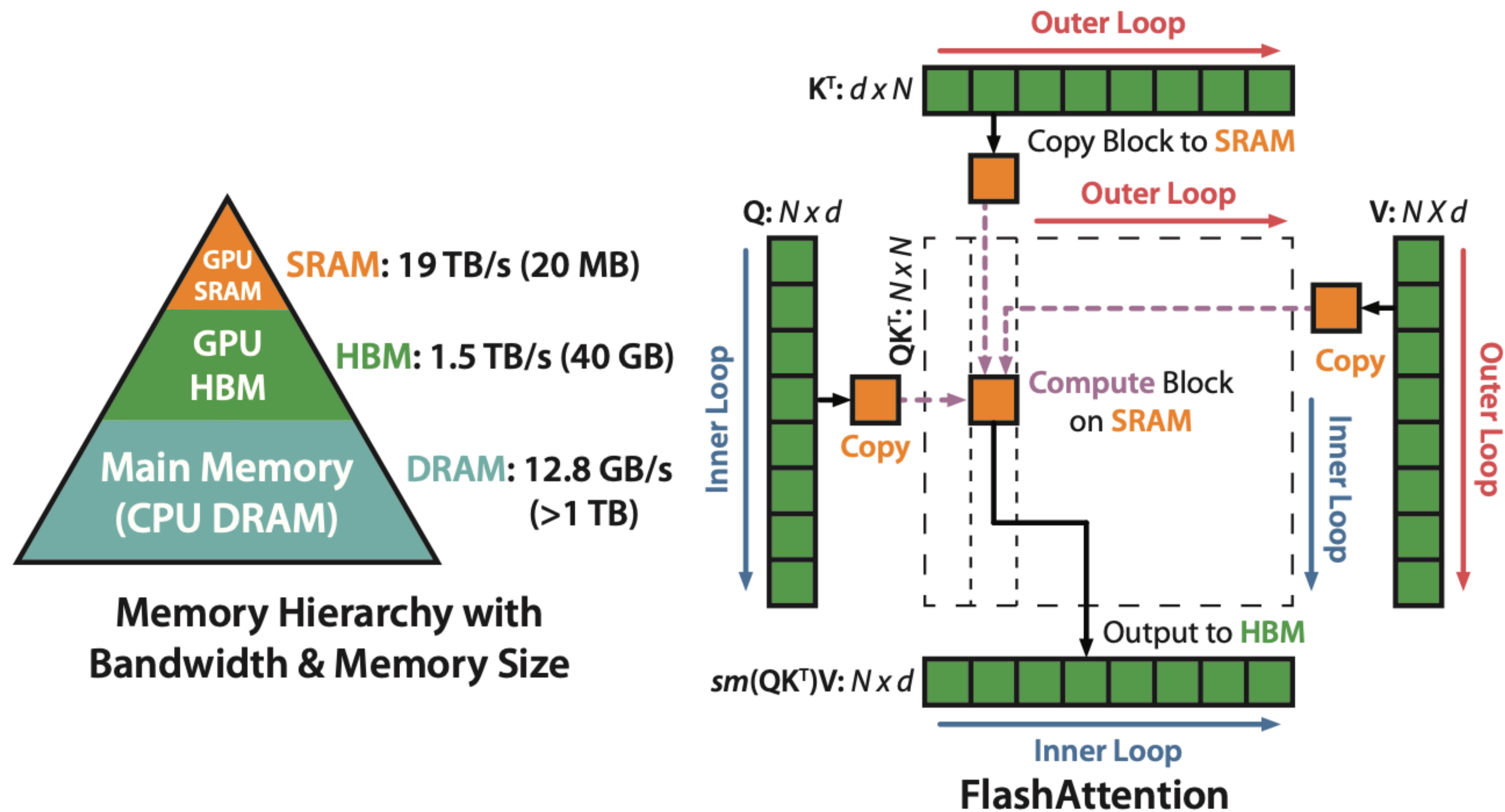

Memory Hierarchy with Bandwidth & Memory Size




so many
load and store!

# FlashAttention: Tiling

- FlashAttention just uses tiling for a KQV matmul
  - Wait.. but what do we do about Softmax?



# Why Softmax Matters? [Milakov+ 2018]

- Softmax denominator requires aggregation over the sequence
  - For safe softmax, we also need max value
  - We do not have access to the whole length as we load data in tiles

$$P(y_i) = \frac{e^{x_i}}{\sum_{j=1}^{V} e^{x_j}}$$

Softmax

- Load: 2N (sum, div)
- Store: N (div)

**Algorithm 1** Naive softmax

1: $d_0 \leftarrow 0$
2: **for** $j \leftarrow 1, V$ **do**
3: $\quad d_j \leftarrow d_{j-1} + e^{x_j}$
4: **end for**
5: **for** $i \leftarrow 1, V$ **do**
6: $\quad y_i \leftarrow \frac{e^{x_i}}{d_V}$
7: **end for**

# Why Softmax Matters? [Milakov+ 2018]

- Softmax denominator requires aggregation over the sequence
  - For safe softmax, we also need max value
  - We do not have access to the whole length as we load data in tiles

$$P(y_i) = \frac{e^{x_i - m}}{\sum_{j=1}^{V} e^{x_j - m}}$$

Safe Softmax

- Load: 3N (max, sum, div)
- Store: N (div)

**Algorithm 2** Safe softmax

1: $m_0 \leftarrow -\infty$
2: **for** $k \leftarrow 1, V$ **do**
3: $\quad m_k \leftarrow \max(m_{k-1}, x_k)$
4: **end for**
5: $d_0 \leftarrow 0$
6: **for** $j \leftarrow 1, V$ **do**
7: $\quad d_j \leftarrow d_{j-1} + e^{x_j - m_V}$
8: **end for**
9: **for** $i \leftarrow 1, V$ **do**
10: $\quad y_i \leftarrow \frac{e^{x_i - m_V}}{d_V}$
11: **end for**

# Online Softmax [Milakov+ 2018]

**Algorithm 3** Safe softmax with online normalizer calculation

1: $m_0 \leftarrow -\infty$
2: $d_0 \leftarrow 0$
3: **for** $j \leftarrow 1, V$ **do**
4: $\quad m_j \leftarrow \max\left(m_{j-1}, x_j\right)$
5: $\quad d_j \leftarrow d_{j-1} \times e^{m_{j-1}-m_j} + e^{x_j - m_j}$
6: **end for**
7: **for** $i \leftarrow 1, V$ **do**
8: $\quad y_i \leftarrow \frac{e^{x_i - m_V}}{d_V}$
9: **end for**

**Algorithm 2** Safe softmax

1: $m_0 \leftarrow -\infty$
2: **for** $k \leftarrow 1, V$ **do**
3: $\quad m_k \leftarrow \max(m_{k-1}, x_k)$
4: **end for**
5: $d_0 \leftarrow 0$
6: **for** $j \leftarrow 1, V$ **do**
7: $\quad d_j \leftarrow d_{j-1} + e^{x_j - m_V}$
8: **end for**
9: **for** $i \leftarrow 1, V$ **do**
10: $\quad y_i \leftarrow \frac{e^{x_i - m_V}}{d_V}$
11: **end for**

To keep track of the max, incrementally update the max, and set up a telescoping sum

# Online Softmax [Milakov+ 2018]

**Algorithm 3** Safe softmax with online normalizer calculation

1: $m_0 \leftarrow -\infty$
2: $d_0 \leftarrow 0$
3: **for** $j \leftarrow 1, V$ **do**
4: $\quad m_j \leftarrow \max(m_{j-1}, x_j)$
5: $\quad d_j \leftarrow d_{j-1} \times e^{m_{j-1}-m_j} + e^{x_j - m_j}$ ← how?
6: **end for**
7: **for** $i \leftarrow 1, V$ **do**
8: $\quad y_i \leftarrow \frac{e^{x_i - m_V}}{d_V}$
9: **end for**

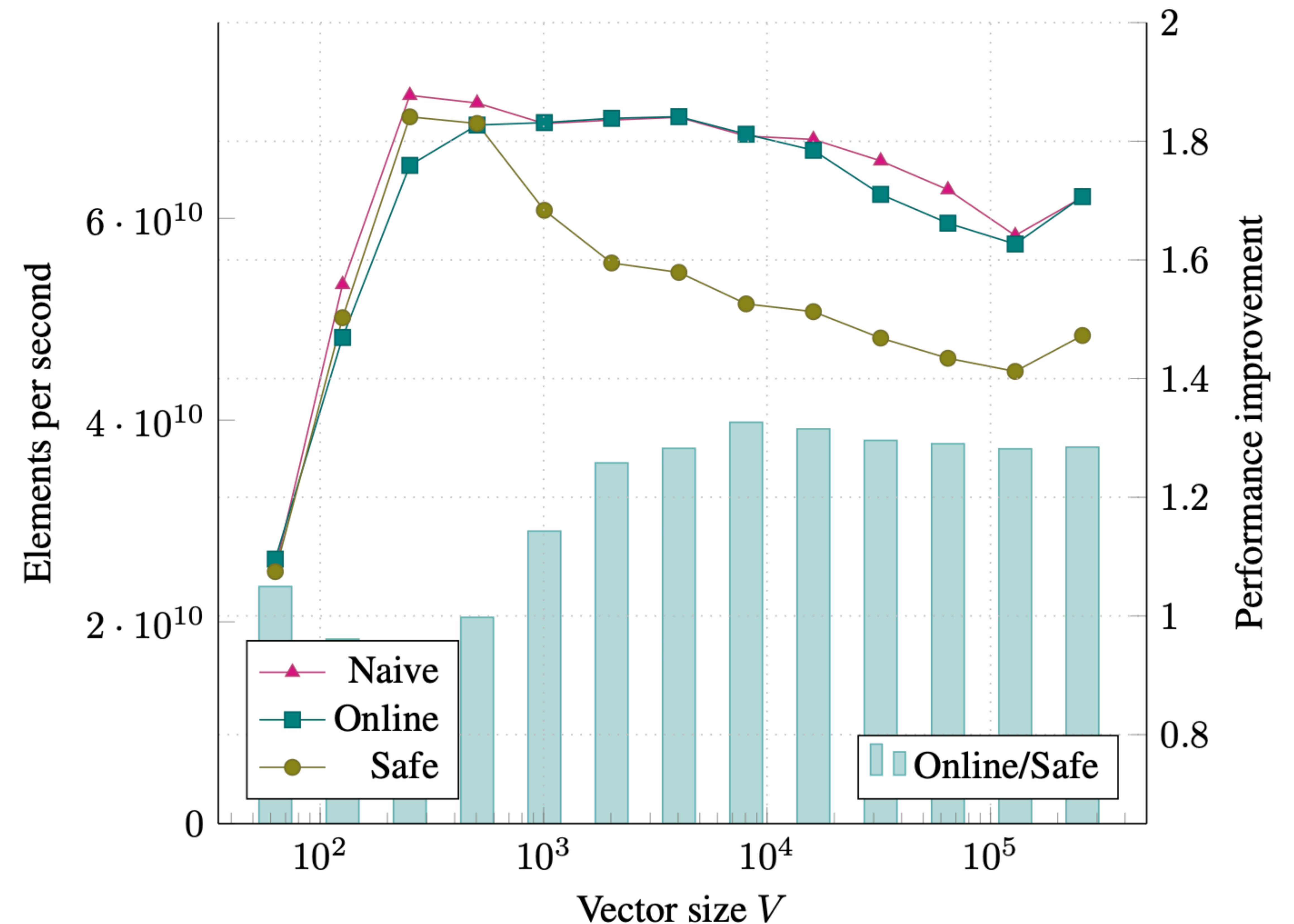
**Algorithm 2** Safe softmax

1: $m_0 \leftarrow -\infty$
2: **for** $k \leftarrow 1, V$ **do**
3: $\quad m_k \leftarrow \max(m_{k-1}, x_k)$
4: **end for**
5: $d_0 \leftarrow 0$
6: **for** $j \leftarrow 1, V$ **do**
7: $\quad d_j \leftarrow d_{j-1} + e^{x_j - m_V}$
8: **end for**
9: **for** $i \leftarrow 1, V$ **do**
10: $\quad y_i \leftarrow \frac{e^{x_i - m_V}}{d_V}$
11: **end for**

To keep track of the max, incrementally update the max, and set up a telescoping sum

# Online Softmax [Milakov+ 2018]

- Softmax
  - Load: 2N (sum, div)
  - Store: N (div)
- Safe Softmax
  - Load: 3N (max, sum, div)
  - Store: N (div)
- Online Softmax
  - Load: 2N (fused max/sum, div)
  - Store: N (div)



# Online Softmax

- How we derive online Softmax?
  - Create a surrogate $d_i' := \sum_{j=1}^{i} e^{x_j - m_i}$
  - Then, we can find a recurrence relation between $d_i'$ and $d_{i-1}'$

$$
\begin{aligned}
d_i' &= \sum_{j=1}^{i} e^{x_j - m_i} \\
&= \left( \sum_{j=1}^{i-1} e^{x_j - m_i} \right) + e^{x_i - m_i} \\
&= \left( \sum_{j=1}^{i-1} e^{x_j - m_{i-1}} \right) e^{m_{i-1} - m_i} + e^{x_i - m_i} \\
&= d_{i-1}' \, e^{m_{i-1} - m_i} + e^{x_i - m_i}
\end{aligned}
$$

**Algorithm 3** Safe softmax with online normalizer calculation

1: $m_0 \leftarrow -\infty$
2: $d_0 \leftarrow 0$
3: **for** $j \leftarrow 1, V$ **do**
4: $\quad m_j \leftarrow \max(m_{j-1}, x_j)$
5: $\quad d_j \leftarrow d_{j-1} \times e^{m_{j-1} - m_j} + e^{x_j - m_j}$
6: **end for**
7: **for** $i \leftarrow 1, V$ **do**
8: $\quad y_i \leftarrow \frac{e^{x_i - m_V}}{d_V}$
9: **end for**

**Algorithm 2** Safe softmax

1: $m_0 \leftarrow -\infty$
2: **for** $k \leftarrow 1, V$ **do**
3: $\quad m_k \leftarrow \max(m_{k-1}, x_k)$
4: **end for**
5: $d_0 \leftarrow 0$
6: **for** $j \leftarrow 1, V$ **do**
7: $\quad d_j \leftarrow d_{j-1} + e^{x_j - m_V}$
8: **end for**
9: **for** $i \leftarrow 1, V$ **do**
10: $\quad y_i \leftarrow \frac{e^{x_i - m_V}}{d_V}$
11: **end for**

# From Online Softmax to FlashAttention

- Still, online Softmax has two loops, can we minimize to reduce I/O?
  - No, but we can combine Softmax and attention
- Let's formulate the k-th row of self-attention as recurrence algorithm

# From Online Softmax to FlashAttention

- Self-attention

NOTATIONS

$Q[k,:]$: the $k$-th row vector of $Q$ matrix.
$K^T[:,i]$: the $i$-th column vector of $K^T$ matrix.
$O[k,:]$: the $k$-th row of output $O$ matrix.
$V[i,:]$: the $i$-th row of $V$ matrix.
$\{\boldsymbol{o}_i\}$: $\sum_{j=1}^{i} a_j V[j,:]$, a row vector storing partial aggregation result $A[k,:i] \times V[:i,:]$

BODY

**for** $i \leftarrow 1, N$ **do**

$$\begin{aligned} x_i &\leftarrow Q[k,:]\, K^T[:,i] \\ m_i &\leftarrow \max(m_{i-1}, x_i) \\ d'_i &\leftarrow d'_{i-1} e^{m_{i-1}-m_i} + e^{x_i - m_i} \end{aligned}$$

**end**

**for** $i \leftarrow 1, N$ **do**

$$a_i \leftarrow \frac{e^{x_i - m_N}}{d'_N} \tag{11}$$

$$\boldsymbol{o}_i \leftarrow \boldsymbol{o}_{i-1} + a_i V[i,:] \tag{12}$$

**end**

$$O[k,:] \leftarrow \boldsymbol{o}_N$$

# From Online Softmax to FlashAttention

**for** $i \leftarrow 1, N$ **do**

$$a_i \leftarrow \frac{e^{x_i - m_N}}{d'_N} \tag{11}$$

$$\boldsymbol{o}_i \leftarrow \boldsymbol{o}_{i-1} + a_i V[i,:] \tag{12}$$

**end**

$$O[k,:] \leftarrow \boldsymbol{o}_N$$

- We can replace combine eq. 11 and 12: $\boldsymbol{o}_i := \sum_{j=1}^{i} \left( \frac{e^{x_j - m_N}}{d'_N} V[j,:] \right)$
  - This still depends on $m_N$ and $d_N$ (which cannot be determined until the previous loop completes)
- But we can use surrogate trick again: $\boldsymbol{o}'_i := \left( \sum_{j=1}^{i} \frac{e^{x_j - m_i}}{d'_i} V[j,:] \right)$

# From Online Softmax to FlashAttention

- Now we can find a recurrence relation between $o_i'$ and $o_{i-1}'$

$$
\begin{aligned}
\boldsymbol{o}_i' &= \sum_{j=1}^{i} \frac{e^{x_j - m_i}}{d_i'} V[j,:] \\
&= \left( \sum_{j=1}^{i-1} \frac{e^{x_j - m_i}}{d_i'} V[j,:] \right) + \frac{e^{x_i - m_i}}{d_i'} V[i,:] \\
&= \left( \sum_{j=1}^{i-1} \frac{e^{x_j - m_{i-1}}}{d_{i-1}'} \frac{e^{x_j - m_i}}{e^{x_j - m_{i-1}}} \frac{d_{i-1}'}{d_i'} V[j,:] \right) + \frac{e^{x_i - m_i}}{d_i'} V[i,:] \\
&= \left( \sum_{j=1}^{i-1} \frac{e^{x_j - m_{i-1}}}{d_{i-1}'} V[j,:] \right) \frac{d_{i-1}'}{d_i'} e^{m_{i-1} - m_i} + \frac{e^{x_i - m_i}}{d_i'} V[i,:] \\
&= \boldsymbol{o}_{i-1}' \frac{d_{i-1}' e^{m_{i-1} - m_i}}{d_i'} + \frac{e^{x_i - m_i}}{d_i'} V[i,:]
\end{aligned}
$$

# From Online Softmax to FlashAttention

- We have one-pass FlashAttention!

**for** $i \leftarrow 1, N$ **do**

$$\begin{aligned} x_i &\leftarrow Q[k,:]\,K^T[:,i] \\ m_i &\leftarrow \max\left(m_{i-1}, x_i\right) \\ d_i' &\leftarrow d_{i-1}'\,e^{m_{i-1}-m_i} + e^{x_i - m_i} \\ \boldsymbol{o}_i' &\leftarrow \boldsymbol{o}_{i-1}' \frac{d_{i-1}'\,e^{m_{i-1}-m_i}}{d_i'} + \frac{e^{x_i - m_i}}{d_i'} V[i,:] \end{aligned}$$

**end**

$$O[k,:] \leftarrow \boldsymbol{o}_N'$$

# From Online Softmax to FlashAttention

- Combining with tiling

$b$: the block size of the tile
#tiles: number of tiles in the row, $N = b \times \#\text{tiles}$.
$\boldsymbol{x}_i$: a vector storing the $Q[k]\,K^T$ value of the $i$-th tile $[(i-1)\,b : i\,b]$.
$m_i^{(\text{local})}$: the local maximum value inside $\boldsymbol{x}_i$.

**for** $i \leftarrow 1, \#\text{tiles}$ **do**

$$\begin{aligned}
\boldsymbol{x}_i &\leftarrow Q[k,:]\,K^T[:,(i-1)\,b : i\,b] \\
m_i^{(\text{local})} &= \max_{j=1}^{b}\left(\boldsymbol{x}_i[j]\right) \\
m_i &\leftarrow \max\left(m_{i-1}, m_i^{(\text{local})}\right) \\
d_i' &\leftarrow d_{i-1}'\,e^{m_{i-1}-m_i} + \sum_{j=1}^{b} e^{\boldsymbol{x}_i[j]-m_i} \\
\boldsymbol{o}_i' &\leftarrow \boldsymbol{o}_{i-1}'\,\frac{d_{i-1}'\,e^{m_{i-1}-m_i}}{d_i'} + \sum_{j=1}^{b} \frac{e^{\boldsymbol{x}_i[j]-m_i}}{d_i'}\,V[j+(i-1)\,b,:]
\end{aligned}$$

**end**

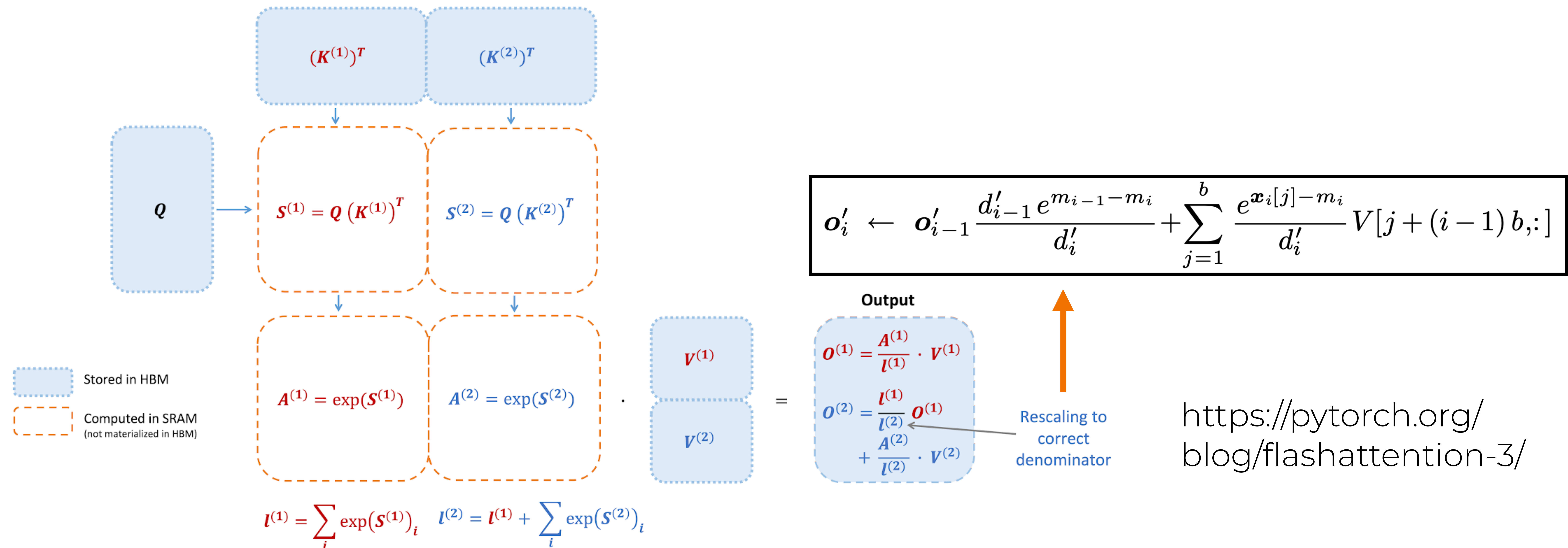
$$O[k,:] \leftarrow \boldsymbol{o}_{N/b}'$$

# Recomputation & Kernel Fusion [Dao+ 2022]

**Recomputation.** One of our goals is to not store $O(N^2)$ intermediate values for the backward pass. The backward pass typically requires the matrices $\mathbf{S}, \mathbf{P} \in \mathbb{R}^{N \times N}$ to compute the gradients with respect to $\mathbf{Q}, \mathbf{K}, \mathbf{V}$. However, by storing the output $\mathbf{O}$ and the softmax normalization statistics $(m, \ell)$, we can recompute the attention matrix $\mathbf{S}$ and $\mathbf{P}$ easily in the backward pass from blocks of $\mathbf{Q}, \mathbf{K}, \mathbf{V}$ in SRAM. This can be seen as a form of selective gradient checkpointing [10, 34]. While gradient checkpointing has been suggested to reduce the maximum amount of memory required [66], all implementations (that we know off) have to trade speed for memory. In contrast, even with more FLOPs, our recomputation speeds up the backward pass due to reduced HBM accesses (Fig. 2). The full backward pass description is in Appendix B.

**Implementation details: Kernel fusion.** Tiling enables us to implement our algorithm in one CUDA kernel, loading input from HBM, performing all the computation steps (matrix multiply, softmax, optionally masking and dropout, matrix multiply), then write the result back to HBM (masking and dropout in Appendix B). This avoids repeatedly reading and writing of inputs and outputs from and to HBM.

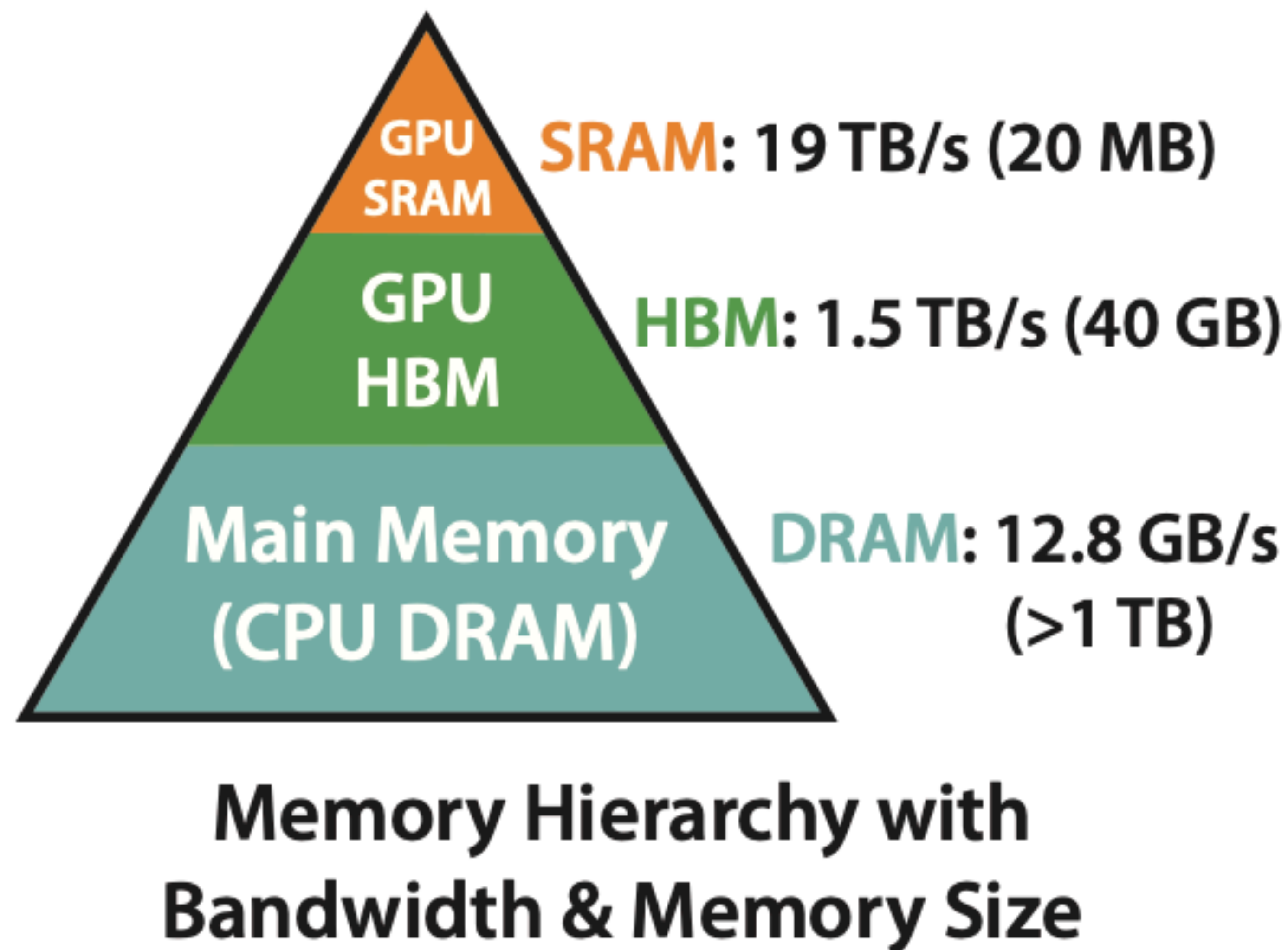
# FlashAttention [Dao+ 2022]

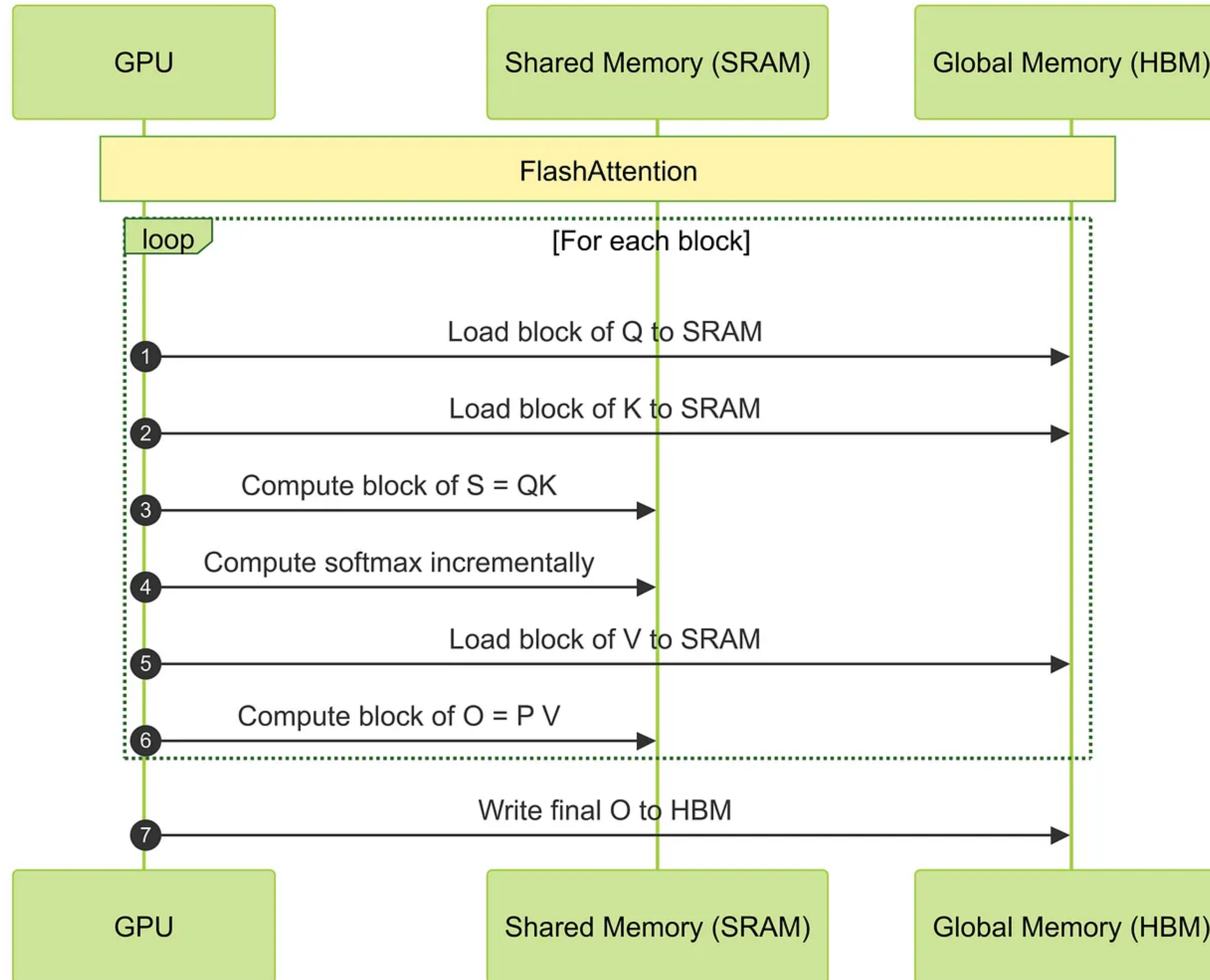



- Tile-wise computation of the inner product (S)
- Fusion of the exponential operator
- Tile-wise computation of the Softmax via online

# FlashAttention: Communication

https://medium.com/@najeebkan/flashattention-one-two-three-6760ad030ae0



# Summary

- Hardware powers scale, and low-level details determine what scales or doesnt
- Curent GPU based compute strongly encourages thinking about matmul + data movement
- Thinking carefully about the GPU (coalescing, tiling, fusion) leads us to good performance