INTRODUCTION TO NATURAL LANGUAGE PROCESSING

GPUs & Parallelism in Deep Learning

Victor Zhong

The Compute Revolution

The Scaling Hypothesis

Modern NLP progress is largely driven by scaling compute, data, and parameters.

The Hardware Lottery

Research directions often win not because they are theoretically superior, but because they fit the available hardware (Hooker, 2020).

Example

Neural Networks (highly parallelizable) beat symbolic AI partly because GPUs (created for gaming) accidentally provided the perfect substrate for matrix math.

Efficiency: Time vs. Data

A good neural network must be efficient in two ways:

Data Efficiency

Achieving high performance with fewer training examples (sample complexity).

Time Efficiency

Achieving high performance with less wall-clock time or energy.

Computation ≈ Model Size × Number of Examples

The End of Dennard Scaling

Dennard Scaling (1974-2005)

As transistors got smaller, they got faster and used the same power density.

Result: Clock speeds doubled every ~18 months ("Free Lunch").

Different than Moore’s Law: # transistors on a chip.

The Breakdown

Around 2005, physics limits (leakage current, heat) stopped frequency scaling. CPUs stalled at ~3-5 GHz.

The New Era: We can't make processors faster, so we make more of them (Parallelism).

The Rise of Parallelism

Single-chip AI inference performance increased **1000x** in 10 years (2012-2022).

1. Parallelism

Moving from few cores (CPU) to thousands (GPU).

2. Specialization

Hardware specifically for matrix math (Tensor Cores).

3. Reduced Precision

FP32 → FP16 → BF16 → FP8.

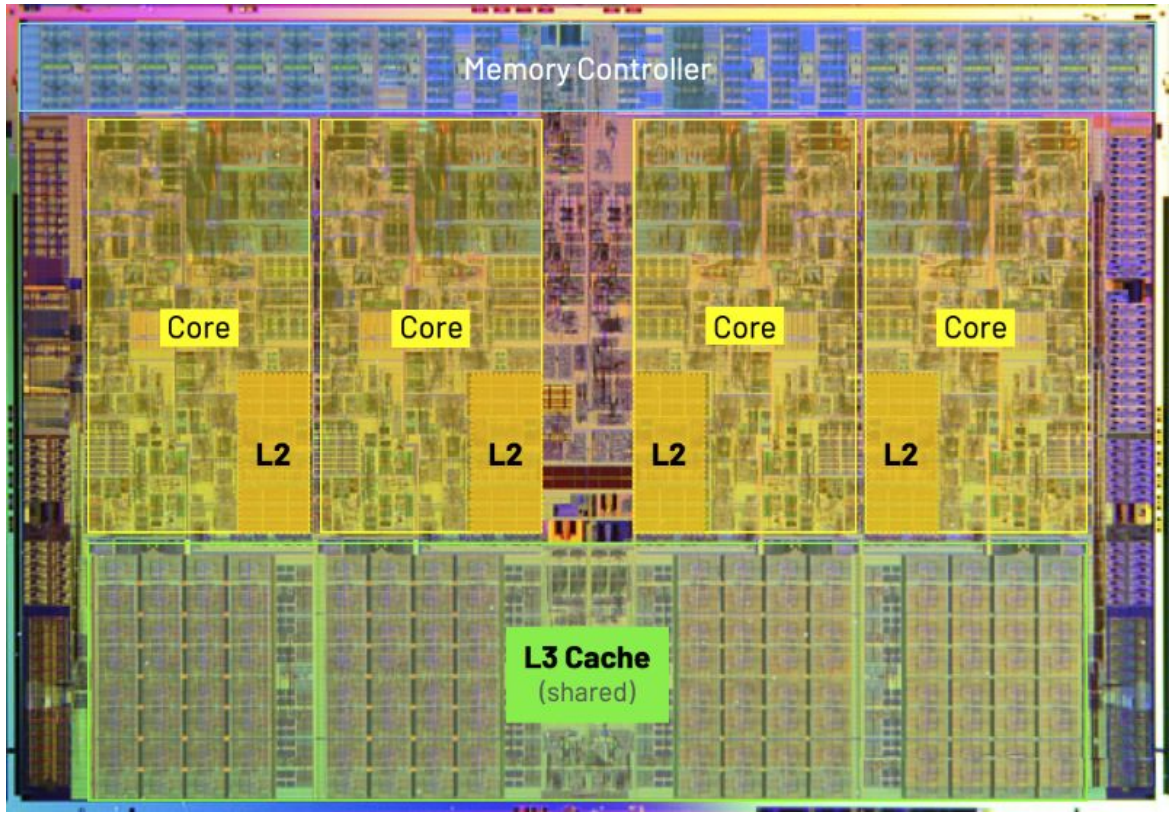
CPU: The Latency Optimizer

Design Goal

Minimize latency for a single serial thread.

Structure

- Few, powerful cores (ALUs).
- Massive caches (L1/L2/L3) to hide memory latency.
- Complex control logic (Branch Prediction, Out-of-Order execution).



CPUs dedicate huge area to control logic and cache. 45nm Intel Nehalem CPU (circa 2008).

Image from: https://pikuma.com/blog/understanding-computer-cache

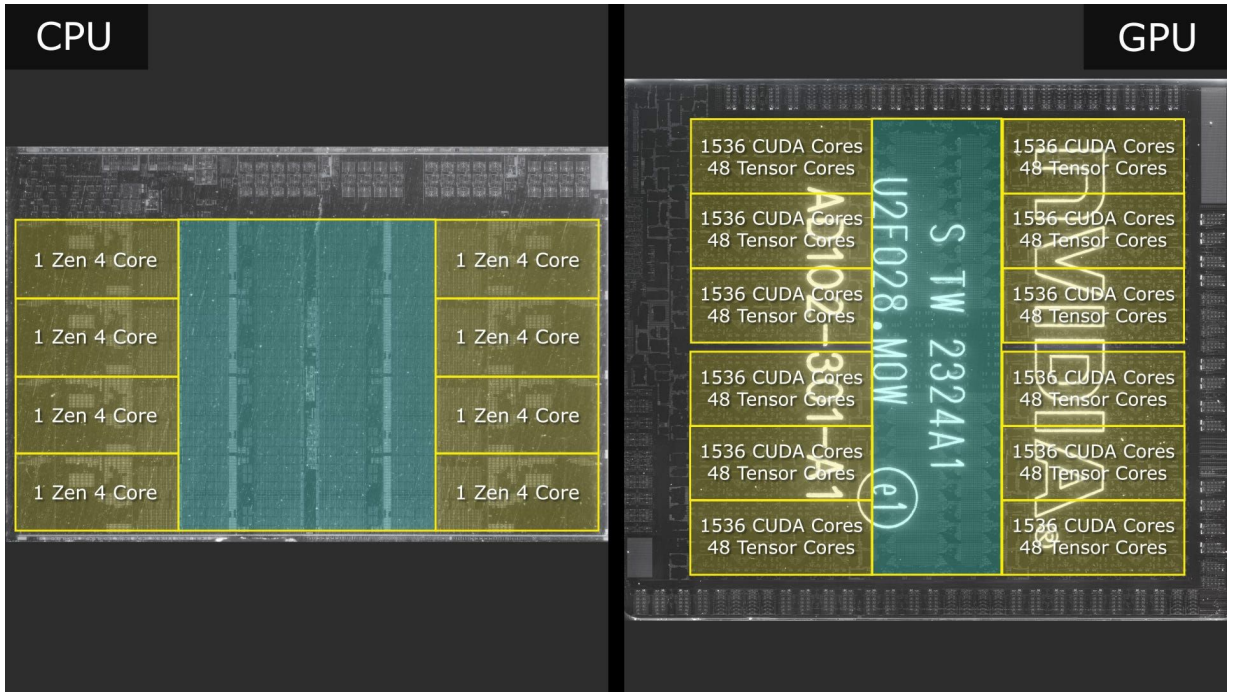
GPU: The Throughput Optimizer

Design Goal

Maximize **throughput** for massive parallel data.

Structure

- Thousands of tiny, simple cores.
- Minimal control logic (no complex branch prediction).
- High Bandwidth Memory (HBM).



GPUs dedicate area to thousands of ALU cores. AMD Ryzen 5 7600 vs NVIDIA RTX 4090.

Image from: https://www.computerenhance.com/p/zen-cuda-and-tensor-cores-part-i

CPU vs. GPU Analogy

CPU: A Ferrari

Moves a small payload (1 person) extremely fast.

GPU: A Cargo Ship

Moves a massive payload (10,000 containers) slowly, but total throughput is massive.

Deep Learning is a bulk shipping problem.

TPU (Tensor Processing Unit)

Design Goal: Specialized ASIC (Application Specific Integrated Circuit) for DL.

Architecture: Systolic Arrays

Data flows through a grid of multipliers like a conveyor belt (heartbeat).

Processing Unit A passes result directly to B (no memory access in between).

Trade-off

Highest efficiency for Matrix Mul, but rigid/hard to program for custom logic.

"The Workbench". Very Fast (~19 TB/s). ~200KB per block.

The GPU Memory Hierarchy

Data movement is the bottleneck. The GPU has layers of memory.

Global Memory (HBM)

In the GBs, on the order of 1 TB/s

Shared Memory (SRAM)

In the MBs, on the order of 10 TB/s

Great read on HBM vs SRAM: https://www.viksnewsletter.com/p/a-close-look-at-sram-for-inference

The Memory Wall

The Scalability Gap

Compute Capability (FLOPs) has scaled **~60,000x** in 20 years.

Memory Bandwidth (HBM) has scaled only **~100x**.

The Gap

We can compute math vastly faster than we can load data.

Implication: Most LLM operations are Memory Bound (GPU cores sit idle).

The Memory Wall

Market caps reflect the physical bottlenecks: NVDA dominates **compute**, MU dominates **memory**.

Execution Model: SIMT

SIMT: Single Instruction, Multiple Threads.

The Warp

The GPU does not execute 1 thread at a time. It executes groups of 32 threads called a **Warp**.

Lockstep: All 32 threads in a Warp execute the exact same instruction at the same time (e.g., ADD).

Consideration 1: Warp Divergence

What happens if we have an if/else statement?

The "If" Path

Thread 1 runs the code.
Thread 2 is masked (asleep).

The "Else" Path

Thread 2 runs the code.
Thread 1 is masked (asleep).

Result: Performance is cut in half. Avoid branching in kernels!

Consideration 2: Memory Coalescing

Global Memory is accessed in "bursts" (transactions).

Coalesced Access

Threads read contiguous addresses.

1 transaction serves the whole warp.

Uncoalesced Access

Threads read random addresses.

32 separate transactions. Bandwidth drops to ~3%.

The "Powers of 2" Rule

GPUs love powers of 2.

Memory alignment, warp sizes (32), and tensor core dimensions align with 2, 4, 8, 16, 32...

Tip

Always set batch sizes, hidden dimensions, and vocab sizes to multiples of **64** or **128**.

Example: Increasing vocab from 50,000 to 50,304 (multiple of 64) speeded up NanoGPT training by ~25% (Karpathy 2023).

https://x.com/karpathy/status/1621578354024677377

Consideration 3: Arithmetic Intensity

How do we determine if an algorithm will run fast?

High Intensity

Matrix Multiplication, Convolution.
(Math bound)

Low Intensity

Element-wise Addition, Activations.
(Memory bound)

The Roofline Model

Slanted Region (Memory Bound)

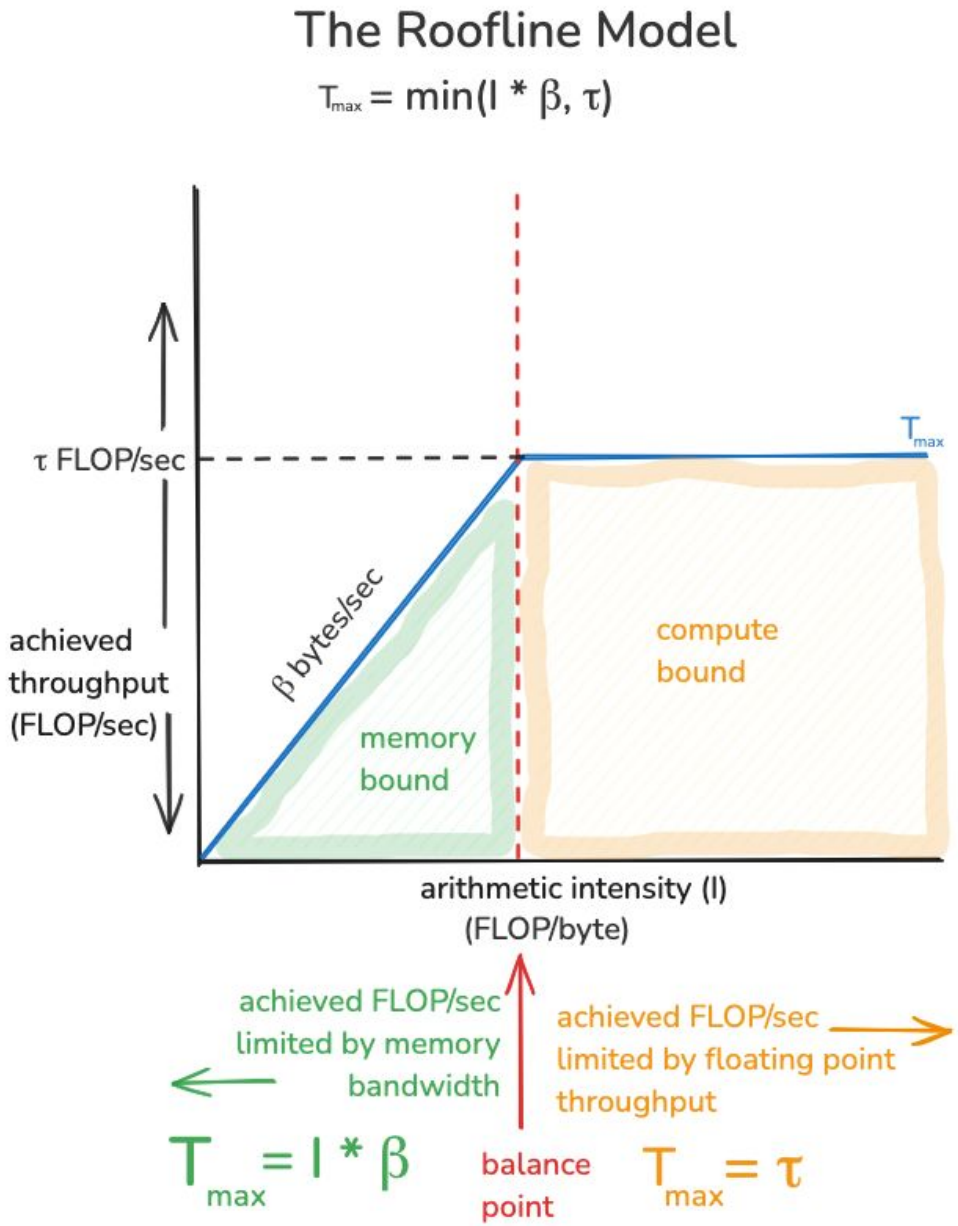
Performance scales with Memory Bandwidth.

"Feeding the beast".

Flat Region (Compute Bound)

Performance capped by Compute Units.

"Crunching numbers".



https://alexarmbr.github.io/2024/08/10/How-To-Write-A-Fast-Matrix-Multiplication-From-Scratch-With-Tensor-Cores.html

Case Study 1: Vector Addition

SAXPY: y = ax+y

Math

O(N) FLOPs

Memory

O(N) Bytes moved

Intensity

Roughly:

Total Work / Total Traffic = O(1)

Verdict: Extremely Memory Bound

H100 cores will be often idle.

Case Study 2: Matrix Multiplication

SGEMM: C = A@B+C

Math

O(N^3)

Memory

O(N^2)

Intensity

O(N)

Verdict: Compute Bound

Intensity grows with N. This is why Transformers win (Attention is matrix math).

Tensor Cores

Standard Cores

Can do any general scalar math

Tensor Cores

Specialized hardware units that perform a 4×4 matrix multiply (D=A@B+C) in **one clock cycle**.

Volta (V100)

8x speedup over FP32.

Hopper (H100)

Transformer Engine (FP8).

Constraint

Dimensions must be multiples of 8 or 16.

Consideration 4: Scaling Beyond One GPU

The Limit

A single H100 has 80GB VRAM.

Llama-3-70B requires ~140GB just for weights (in FP16).

We cannot fit the model on one GPU.

We need Parallelism.

Data Parallelism (DP)

Use Case: Model fits on one GPU, but we want to train faster.

Method

- Replicate model across GPUs.
- Split batch: GPU 1 gets Data A, GPU 2 gets Data B.
- Compute Gradients independently.
- **All-Reduce:** Average gradients across GPUs.

Tensor Parallelism (TP)

Use Case: A single layer (e.g., huge matrix multiplication) is too big for one GPU.

Method

Split the matrix itself across GPUs.
GPU 1 stores top half of W.
GPU 2 stores bottom half.

Constraint

Requires very fast communication (NVLink) because every layer needs synchronization.

Because of this NVLink requirement, **TP is almost always restricted to a single node** (e.g., 8 GPUs in one server box) and cannot easily span across a data center like Data Parallelism

Pipeline Parallelism (PP)

Use Case: Model is too deep.

Method: Partition Layers

GPU 1: Layers 1-10.
GPU 2: Layers 11-20.

Problem

Pipeline Bubble: GPU 2 sits idle while GPU 1 processes the first batch.

Solution

"Micro-batches" overlap computation on one GPU with communication to the next.

Mixture of Experts (MoE) Parallelism

Structure: Sparse models with routed experts (e.g., Mixtral).

Expert Parallelism

- GPU 1 hosts Experts 1 & 2.
- GPU 2 hosts Experts 3 & 4.

Challenge

All-to-All: When a token needs Expert 3, it must be sent over the network to GPU 2. Bandwidth becomes the bottleneck.

Consideration 5: Kernel Launch Overhead

The Sequence

The CPU tells the GPU what to do ("Launch kernel A").

Launching a kernel takes **~5-10 microseconds**.

The Trap

If your kernel runs faster than the launch time (e.g., adding two tiny vectors), the GPU sits idle.

Lesson: Do massive work in each kernel. Don't write loops in Python.

Optimization 1: Tiling

Problem: Naive Matrix Mul repeatedly reads A and B from slow global memory.

Solution: Tiling (Blocking)

1. Load small tiles of A and B into fast **Shared Memory (SRAM)**.
2. Threads compute partial products using only SRAM data.
3. Load next tile.

Effect: Reduces Global Memory access by a factor of Tile Size.

Tile Quantization ("Wave Quantization")

The GPU processes blocks in "waves".

The Problem

If the matrix size isn't perfectly divisible by the tile size, we get "leftover" tiles.

The Cost

If the last wave is only partially full, the entire GPU waits for a few SMs to finish.

Result: Performance drops periodically as matrix size increases (sawtooth pattern).

Optimization 2: Operator Fusion

Scenario: LayerNorm(x) → ReLU(x) → Dropout(x)

Naive

- Read x, norm, Write temp1.
- Read temp1, relu, Write temp2.
- Read temp2, dropout, Write out.

Fused (Kernel Fusion)

- Read x.
- Compute norm, relu, dropout **in registers**.
- Write out.

Benefit: 3x reduction in memory traffic.

Optimization 3: Low Precision

Format	Bits	Description
FP32	32	Old standard.
TF32	19	NVIDIA specific. FP32 range, FP16 precision.
BF16	**16**	**Google Brain format. Standard for LLMs.**
FP8	8	H100 standard. Requires scaling.

Benefits: Halves memory usage, doubles bandwidth, doubles compute throughput.

Optimization 4: Recomputation

Training Memory Cost: Storing activations for backpropagation takes huge VRAM.

Idea: Gradient Checkpointing

Don't store all activations. Store checkpoints.

Backward Pass: When you need a missing activation, **recompute it** from the nearest checkpoint.

Trade-off: Trades Compute (re-running forward pass) for Memory. Allows training significantly larger batch sizes.

Optimization 5: The Attention Bottleneck

Standard Attention: $\text{Softmax}(QK^T)V$

Algorithm 0 Standard Attention Implementation

Require: Matrices $\mathbf{Q}, \mathbf{K}, \mathbf{V} \in \mathbb{R}^{N \times d}$ in HBM.

1: Load $\mathbf{Q}, \mathbf{K}$ by blocks from HBM, compute $\mathbf{S} = \mathbf{Q}\mathbf{K}^\top$, write $\mathbf{S}$ to HBM.
2: Read $\mathbf{S}$ from HBM, compute $\mathbf{P} = \text{softmax}(\mathbf{S})$, write $\mathbf{P}$ to HBM.
3: Load $\mathbf{P}$ and $\mathbf{V}$ by blocks from HBM, compute $\mathbf{O} = \mathbf{P}\mathbf{V}$, write $\mathbf{O}$ to HBM.
4: Return $\mathbf{O}$.

For long sequences, NxN is massive. HBM reads/writes dominate.

https://gordicaleksa.medium.com/eli5-flash-attention-5c44017022ad

FlashAttention Intuition

Key Idea

Can we compute Attention **without ever materializing the full matrix** in global memory?

Method

Tiling + Recomputation.

Compute the Softmax one small block at a time inside SRAM (fast shared memory).

4. Write

Write **only the final output** to HBM.

FlashAttention Mechanics

1. Load

Load blocks of KVQ into SRAM.

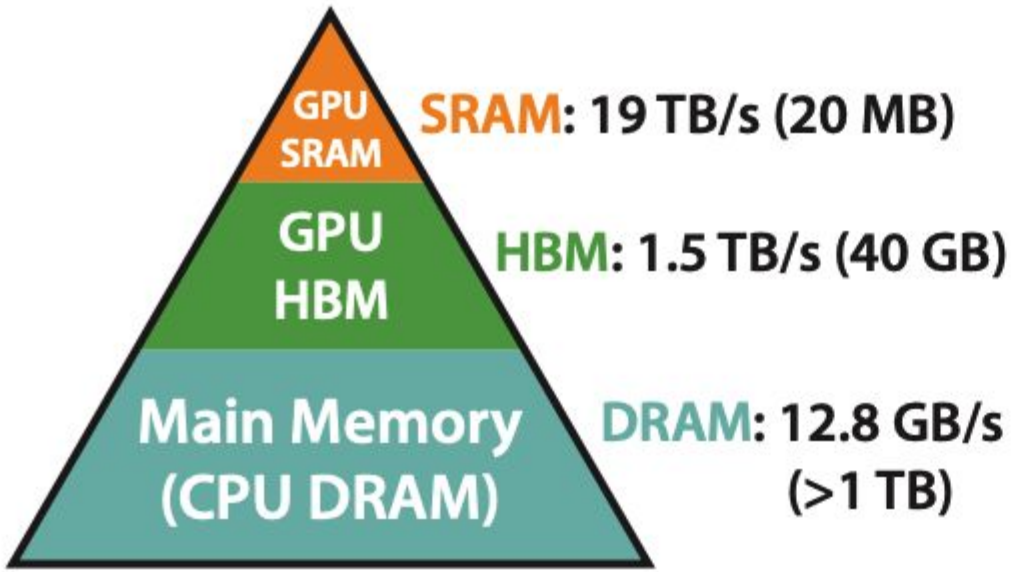
2. Compute

Compute block of QK^T.

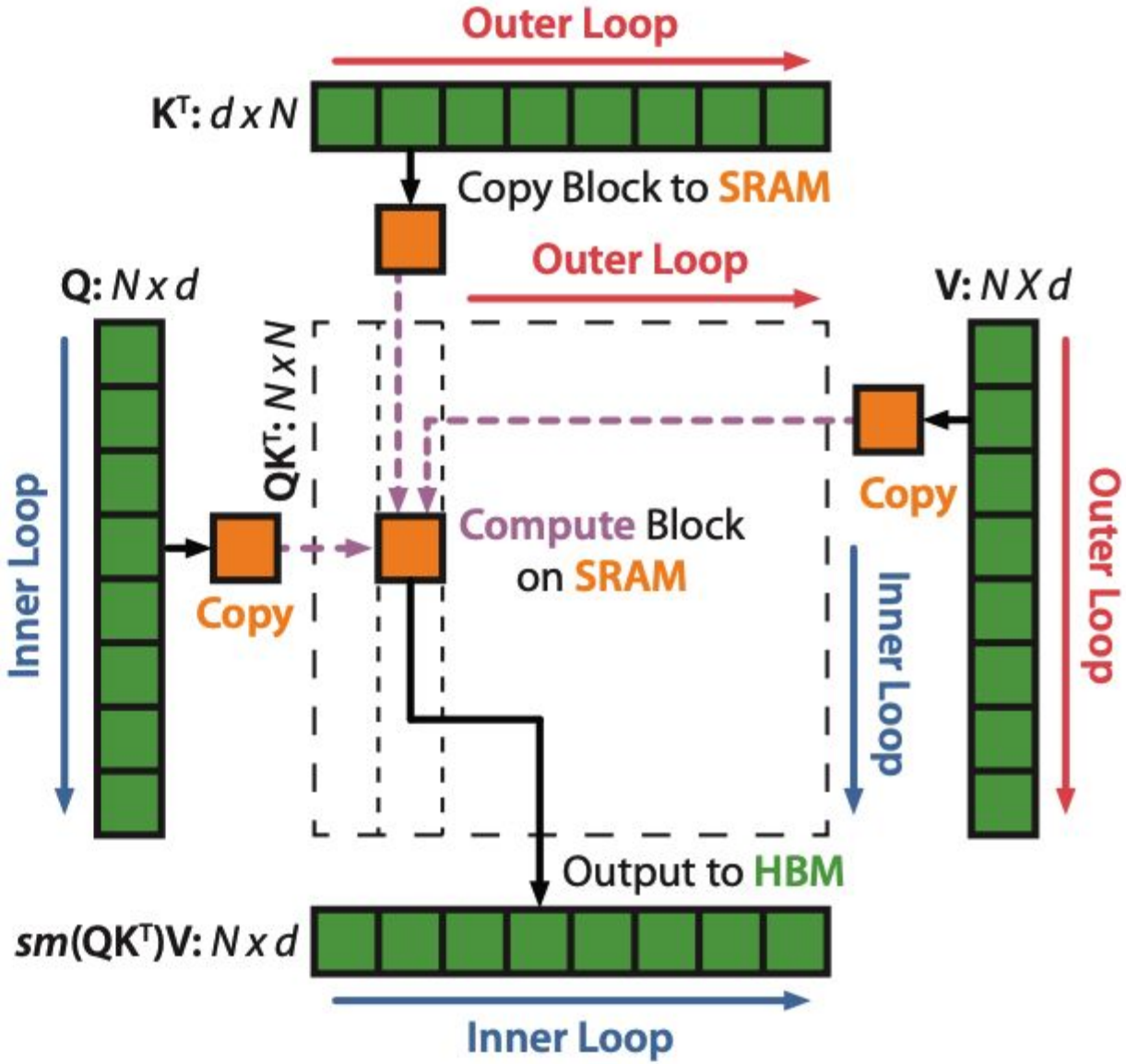
3. Online Softmax

Update Softmax statistics (max, sum) on the fly using a rescaling trick.

https://gordicaleksa.medium.com/eli5-flash-attention-5c44017022ad



Memory Hierarchy with Bandwidth & Memory Size



FlashAttention

https://arxiv.org/pdf/2205.14135

FlashAttention Impact

Speed

2x-4x faster than standard attention.

Memory

Linear memory complexity O(N) instead of Quadratic O(N^2).

Capability

Enabled context windows to jump from 2k → 32k → 128k.

Lesson: Hardware-aware algorithms beat pure mathematical simplifications.

Profiling & Debugging

nvidia-smi

Check GPU utilization and memory usage.

PyTorch Profiler

See exactly which kernels are taking time.

Common Bug

CPU Fallback

tensor[0].item() + 1 inside a loop moves data GPU→CPU→GPU. Kills performance.

Summary

- **Hardware defines Software:** The success of Transformers is linked to their hardware efficiency on GPUs.
- **Memory is the Bottleneck:** Moving data is expensive; math is cheap.
- **Tiling & Fusion:** Key techniques to keep data in fast memory (SRAM/Registers).
- **FlashAttention:** A prime example of hardware-aware algorithm design.

Final Thoughts

- We are entering an era of **Hardware-Software Co-Design**.
- You cannot be a great ML Systems engineer without understanding the metal.
- The next breakthroughs will likely come from algorithms that exploit new hardware capabilities (e.g., Sparse Tensor Cores).