Inference Compute is Going to Explode!

Inference is AI in action.

Its adoption is accelerating rapidly, driven by both existing use cases and emerging applications at the edge and in autonomous systems.

So far, the impact has been significant, but the next phase promises far broader influence.

Inference is where growth is concentrated— both visible and underlying. Let's examine the forces driving it.

Yes, inference compute is going to explode.

Inference Defined

Inference is AI in use. Inference occurs every time you use AI. It is the process where some input is processed by a previously trained model to produce an output.

Training, a long and compute-intensive process, builds the model. Training a large language model may require hundreds of thousands of GPU hours, consuming massive amounts of energy and data to adjust billions of parameters. Inference is the second part of this asymmetric process: training happens once to create the model, and inference happens repeatedly, millions or even billions of times, to put that model to work.

When you type a message into a chatbot, your words are converted into tokens and evaluated by the trained model. The system doesn't relearn anything; it applies what training already established to generate a coherent response in real time. A single chatbot response takes only a fraction of a second and a tiny fraction of the compute used during training.

In Tesla's Full Self-Driving system, the car takes in camera data—streams of photons—and passes it through the trained model to produce steering, braking, and acceleration commands. This process runs continuously, 15 to 30 times per second, and is repeated billions of times over the life of the vehicle.

Training builds the model once; inference executes it continuously.

Inference Compute

To understand inference, it helps to first contrast it with training.

During **training**, a model is built. This requires enormous computational power, measured in days or weeks, as the dataset is processed over many passes called *epochs*. Each epoch consists of a forward pass, a backward pass (backpropagation), and weight updates. Roughly 90–95% of all floating-point operations (FLOPs) in training are matrix multiplications, the dominant

mathematical workload. High precision formats such as FP32 or FP64 are typically used to maintain accuracy during learning. Training emphasizes throughput and memory bandwidth, which is why it is almost always performed in datacenter environments with abundant power.

During **inference**, the trained model is deployed in a read-only state. Input data is pushed through the network to produce predictions; for example, an image might be classified as 93% zebra, 6% horse, and 1% other. Unlike training, inference runs only a forward pass, with no weight updates. It is still dominated by matrix multiplications, but the focus shifts to latency—how quickly a result can be produced.

Inference often operates under tighter constraints:

- **Precision**: Lower formats such as FP16, BF16, INT8, or even INT4 are commonly used, reducing compute and memory costs without significantly hurting accuracy.
- **Batching**: Training always benefits from large batch sizes to maximize throughput. Inference, however, is workload-dependent. Real-time applications (vehicle autonomy, speech recognition, AR/VR) usually run with small or single-item batches to minimize latency, while offline or batch-oriented tasks (recommendation systems, video processing) can leverage large batches to improve throughput and efficiency.
- **Deployment environment**: Training is power-hungry and centralized, whereas inference frequently runs at the edge, where efficiency and low power consumption are critical.

In summary, whether for training or inference, neural network workloads fall under the umbrella of **accelerated computing**—architectures designed for massive parallelism to shrink compute time.

For a deeper exploration of accelerated compute architectures, see my companion article:

(1) phil beisel on X: "Accelerated Computing" / X

The contrast between training's throughput-focused workload and inference's emphasis on low latency and energy efficiency shapes hardware choices. Training relies on high-performance accelerators like NVIDIA's A100 and H100, AMD's MI300 series, and Google's TPU v4, which deliver massive parallelism and mixed-precision math (e.g., FP16/FP32) for processing large datasets over extended periods in data centers.

Inference, conversely, prioritizes fast, efficient execution, often balancing low latency with high throughput for cloud or edge applications. This drives the use of specialized inference accelerators like NVIDIA's H100 (for cloud) and Jetson (for edge), Qualcomm's Cloud AI 100, Apple's Neural Engine, and NPUs in mobile devices. While NVIDIA, AMD, and Google dominate training accelerators, the inference market is more diverse, with leadership split between GPU

vendors and edge-focused players like Qualcomm and Apple. These compute differences not only influence software design but also dictate chip architectures, often requiring co-optimized frameworks and hardware for peak performance.

Inference At The Edge

Today, inference mostly happens in data centers, delivered as Software-as-a-Service (SaaS). Thin clients—web browsers or smartphone apps—depend on accelerators like NVIDIA's H100 or Google's TPUs to run models for text, vision, and more. The compute is remote; the device is just an access point.

That's starting to change.

Smartphones now carry their own AI engines. Apple's Neural Engine (inside its A-series chips) powers Apple Intelligence features like photo edits, text summarization, and message categorization directly on-device—fast, private, and offline. Android follows suit: Snapdragon 8 Gen 3, Google's Tensor G4, and MediaTek's Dimensity 9300 all ship with dedicated NPUs, enabling features such as computational photography, real-time translation, and generative AI tools like Magic Eraser or AI wallpapers.

These gains are real but bounded. Phones must conserve power, so NPUs run optimized, low-precision models rather than data center–scale workloads. Today that means targeted use cases— image cleanup, translation, personalization, AR— while future SoCs may push 60–100 TOPS, expanding what can be done locally. The trade-offs remain: power, heat, and developer adoption.

Still, edge inference is breaking the cloud monopoly. The next leap goes well beyond smartphones.

Enter the robots.

Robotaxi and FSD

Tesla Robotaxi (or any Tesla vehicle running its Full Self-Driving software) is a massive inference machine. Input data-- in the form of camera frames-- is processed 15 to 30 times a second to output driving commands.

Tesla vehicles use a custom inference computer known as Hardware 4 (HW4, also referred to as AI4). Each HW4 board delivers roughly 500 TOPS (trillions of operations per second), powered

by two Tesla-designed SoCs that provide both performance and redundancy. That's more than 10× the NPU-only throughput of a top smartphone (Apple's A18 Pro Neural Engine is ~35 TOPS).

Beyond raw TOPS, HW4 is engineered for sustained, 24/7 automotive workloads—processing multi-camera video streams and running billion-parameter perception and trajectory models. Smartphones, by contrast, can only sustain peak compute for short bursts before throttling.

Tesla's roadmap extends well beyond HW4. The upcoming AI5, due in mid-2026, is rated at ~2,000–2,500 TOPS. Looking further ahead, Tesla has already announced AI6, which is projected to push performance up to 10× higher still.

Humanoids!

Humanoids as embodied AI will drive massive inference at the edge. Tesla's entrant is Optimus, but many other humanoid platforms are also in development.

Like FSD deployments, Optimus will be equipped with two Tesla AI chips—first AI5, and later AI6. The inference demand will be extraordinary. The bot must process continuous vision input and map that into coordinated movements across its actuators: arms/hands, legs/feet, and head.

Beyond locomotion, Optimus will have natural voice interaction. Listening, transcribing, reasoning via an onboard AI assistant (likely a variant of xAI's Grok), and generating human-like speech all consume compute. Expectations will be high: people will want Optimus to remember and recall details from past interactions or sensory input. Imagine asking: "Do I have the Isaacson Da Vinci biography in my library?" or "When did I last have coffee with Michael, and can you summarize our discussion?"

It's no surprise Tesla is developing AI6 with these use cases in mind. Humanoids like Optimus will push inference workloads to their limits and define the future of edge AI.

The Bitstream: Inference Unleashed

A neural network is a function. It ingests data and produces data. That flow—encoded as 0's and 1's—is the bitstream.

So far, AI inference has been applied in domains we can point to: self-driving cars, chatbots, image generation, video synthesis. These are impressive, but they're bounded use cases. The real breakthrough comes when inference itself replaces apps—when functions are no longer pre-coded, but generated directly from the stream of data in real time.

Today's computing model is built on apps: word processors, spreadsheets, slide decks. An entire trillion-dollar industry revolves around static binaries written by humans, wrapped and sold as "applications." The app store is the distribution layer, but the app itself is the bottleneck.

The bitstream blows through that bottleneck. What if functions were not pre-coded, but generated in real time by AI? What if every interaction—analysis, visualization, communication—were synthesized on demand by inference, not by launching a frozen binary?

Elon Musk has spoken of this vision directly. Video games that generate themselves frame by frame, worlds streamed into existence on the fly. Recently, he announced a new venture, Macrohard—a tongue-in-cheek play on "Microsoft"—aimed at disrupting the app industry by replacing pre-built software with live AI-driven applications, simulated in real time without predefined boundaries.

The implications are immediate. Feed in a dataset and ask AI to:

Build a financial model

Generate a presentation explaining it

Build communications that share the results

That's three apps collapsed into one stream of inference. No static binaries, no app suites—just data driving function, continuously.

This is why the bitstream is not just a metaphor but a force multiplier for inference. It transforms inference from a tool that powers individual apps into the foundation that replaces apps entirely. Every prompt becomes a micro-app. Every output is synthesized live. This is how inference accelerates past its current trajectory and becomes the defining driver of exponential adoption.

The bitstream is the next layer of abstraction in computing. If training built the model and inference applies it, then the bitstream transforms inference into the operating system—perpetually active, generating every function on the fly, and continuously replacing what static apps once did. Because every interaction, every computation, every output is now live and AI-driven, inference is no longer intermittent—it happens all the time, across every function.

Inference Chips

Inference chips differ from their training counterparts in several key ways. Both classes are built for parallelism and matrix multiplication, but their design priorities diverge. Training chips emphasize throughput and bandwidth, while inference chips focus on minimizing latency—delivering results quickly. Training typically uses high-precision formats like FP32 or FP64, whereas inference favors lower precision (e.g., FP16 or INT8) to cut computation time and improve efficiency.

Inference in the data center and at the edge also differs significantly. At the edge, power is the dominant constraint since most devices are battery-powered. Vehicles used for autonomy have

more power available, but efficiency matters: every watt spent on inference is a watt not driving the wheels, reducing miles per kWh. Humanoid robots are even more constrained, with batteries often an order of magnitude smaller than vehicles, and they too must dedicate power to actuating motors for locomotion. Smartphones are the most power-limited of all, with only a sliver of their energy budget available for inference. Fortunately, mobile inference workloads remain narrow in scope and are not as continuous or demanding as vehicle autonomy.

NVIDIA

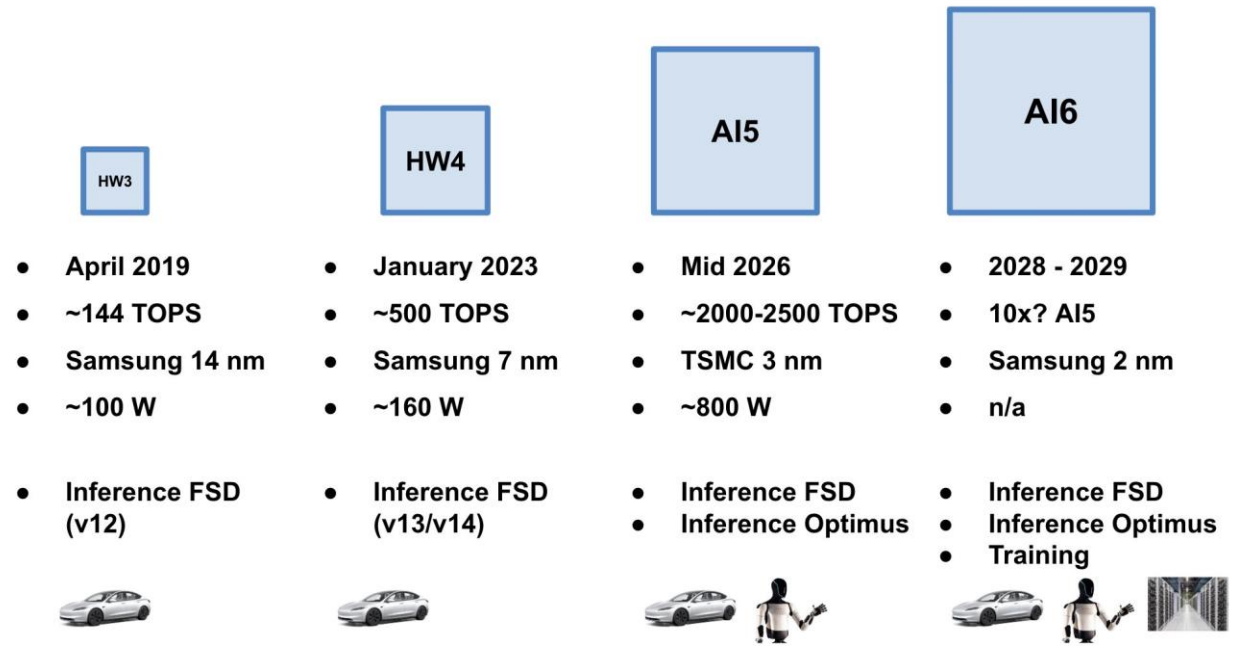
NVIDIA's inference portfolio is a cornerstone of its AI dominance, with hardware tuned for efficiently deploying trained models at scale. In the cloud and data center, the Blackwell architecture (e.g., GB200 systems) and Hopper GPUs (H100/H200) are the current flagships, powering everything from generative AI to recommendation systems and real-time analytics. Hyperscalers like AWS and Google Cloud, along with major enterprises, have standardized on these platforms to run massive inference workloads where high throughput and low latency are critical. Blackwell in particular is being adopted for token-efficient inference in multimodal LLMs, supported by NVIDIA's growing software stack such as the Dynamo framework.

On the edge, NVIDIA's Jetson lineup leads the market. The newly released Jetson Thor delivers 7.5x more AI compute than Jetson AGX Orin, targeting robotics, autonomous vehicles, and other embodied AI applications that require real-time reasoning.

Looking ahead, NVIDIA's roadmap extends well beyond Blackwell. The Rubin architecture (R100) is slated for 2026, followed by Rubin Ultra in 2027, and next-generation NVL576 "AI factory" systems by late 2027. These are projected to deliver up to 21x the inference performance of GB200, aimed at powering large-scale deployments of agentic AI, humanoids, and physical AI applications. With this cadence, NVIDIA is positioning itself to dominate both centralized cloud inference and distributed edge intelligence well into the next decade.

The Tesla Lineup

When you think of accelerated computing GPUs, the first company that comes to mind is undoubtedly NVIDIA, but for edge inference the company to watch is Tesla. Tesla was ahead of the game with its HW3 chip, introduced in 2019. Its primary use case was inference for vehicle autonomy (FSD). Until it was replaced by the next-generation HW4 (aka AI4), it shipped in every vehicle Tesla manufactured.



AI5 is due in mid-2026 and will deliver 4–5x the compute of AI4. AI6 will follow, built to meet the embodied AI inference requirements of Tesla's products.

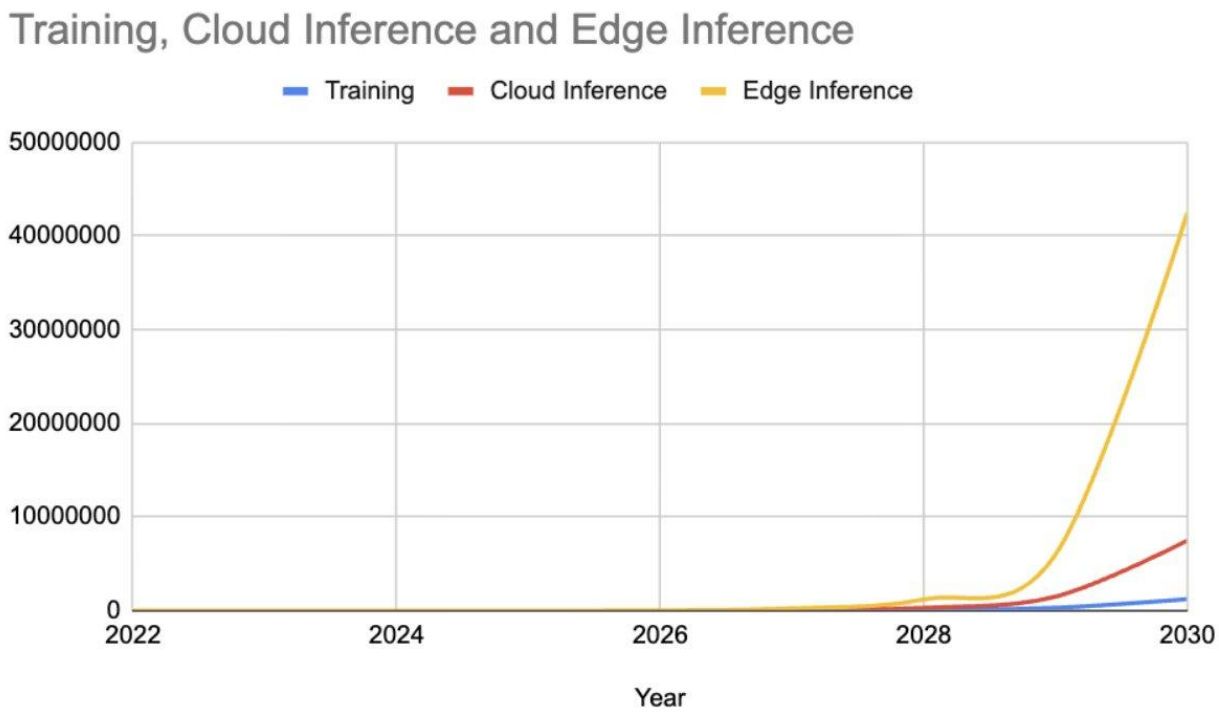
Tesla AI6 In Tesla's 2025 Q2 earnings call, Elon Musk announced plans for a new inference chip: AI6. Days later came news of a multi-billion-dollar exclusive deal with Samsung to produce AI6 at its Texas fab. Within a week, Tesla's Dojo supercomputer team was dissolved.

This sends a strong signal that Musk views inference as Tesla's primary AI need. AI6 will power Robotaxi, customer FSD, and Optimus. It will also stand in for Dojo's D1 effort—a chip "good enough" for training but purpose-built for inference. For Tesla, AI6 represents *convergence*: a unified AI chip strategy where inference takes precedence.

Low-Power Players Inference is increasingly moving to low-power devices where efficiency, latency, and thermal limits matter most. Smartphones, tablets, wearables, and IoT devices now carry dedicated AI engines: Apple's Neural Engine powers on-device photo edits, text summarization, and assistant features; Qualcomm's Snapdragon 8 Gen 3, Google's Tensor G4, and MediaTek's Dimensity 9300 enable computational photography, real-time translation, and lightweight generative AI. Beyond phones, low-power accelerators appear in AR/VR headsets, smart cameras, industrial sensors, and connected home devices, handling tasks like object detection, motion tracking, and voice recognition locally. Though these NPUs deliver far fewer TOPS than data-center GPUs or Tesla's vehicle SoCs, their efficiency is critical—balancing performance, energy, and heat to bring AI everywhere users interact with technology.

The Inference Surge

Figure 4 presents global AI compute trends from 2022–2030, based on historical estimates (Epoch AI), market analyses, and hardware adoption forecasts. Training grows ~4× annually, reaching 1.23 million TOPS by 2030, while cloud inference rises to 7.5 million and edge inference to 42.5 million, reflecting the shift toward distributed, on-device compute.



Even these projections only capture part of the picture. Accelerants like the bitstream, live AI-driven applications, and next-generation autonomous systems will push inference demand further. Once inference is embedded everywhere—perpetually active, continuously generating outputs, replacing static functions—compute requirements will grow dramatically.

As AI adoption spreads, inference workloads will increase continuously, driven by live applications, edge devices, and embodied AI platforms. This sustained demand positions inference as the dominant driver of compute in the coming decade, shaping both hardware development and the trajectory of AI innovation.

(1) phil beisel on X: "Accelerated Computing" / X

Accelerated Computing

Starting in late 2023, *accelerated computing* became part of the mainstream technology lexicon, and for good reason. The world took notice of this pivotal technology trend, which emphasized speed, scale and efficiency in computing tasks. This period marked the start of the meteoric rise of NVIDIA, closely tied to its GPU technologies, and the introduction of OpenAI's seminal tool,

ChatGPT (what is often referred to as the "ChatGPT moment" to mark the start of the AI revolution).

Accelerated computing refers to a new paradigm that enables computing on massive datasets through **parallelism**— handling multiple compute operations simultaneously. This approach is essential for efficiently processing these datasets in fields requiring high computational power, such as artificial intelligence, scientific simulations, and real-time data analysis.

Ultimately its all about the data-- moving it and processing it. A lot of it!

So let's dig in.

The CPU

The CPU (Central Processing Unit) is the core component of a computer responsible for executing instructions and managing the flow of data. Often referred to as the "brain" of the computer, the CPU performs calculations, processes data, and coordinates the actions of other hardware components.

It operates based on a fetch-decode-execute cycle: fetching instructions from memory, decoding them into commands, and executing them to perform tasks. A CPU is driven by a clock (an external quartz crystal oscillator) that determines its execution speed. A CPU processes one instruction per clock cycle; for example a 3 GHz processor can process ~3,000,000,000 instructions per second[1].

Modern CPUs are composed of multiple cores (typically 8 to 16), allowing them to handle multiple processes simultaneously, enhancing performance and efficiency. Integrated with cache memory, the CPU optimizes data retrieval speed, reducing latency and improving overall system responsiveness.

CPUs are part of accelerated computing systems-- they are still the "brain". But they are not the brawn.

The GPU

The GPU (Graphics Processing Unit) is a specialized chip designed primarily to accelerate the rendering of images, animations, and video by performing rapid mathematical calculations in parallel.

Unlike a Central Processing Unit (CPU), which is built for general-purpose computing, a GPU has a highly parallel structure that makes it more efficient for tasks where processing of large blocks of data is done simultaneously.

The first GPU, known as the GeForce 256, was introduced by NVIDIA on August of 1999. This GPU was marketed as "the world's first GPU" and was designed to handle the complex graphics of modern video games and other visual computing tasks, setting the stage for the evolution of graphics processing technology.

In the early 2000s the concept of using the GPU for general-purpose computing started to gain traction. In 2007 NVIDIA launched CUDA (Compute Unified Device Architecture), a platform that allowed developers to use NVIDIA GPUs for general computing. This was a pivotal moment as it provided the tools needed to program GPUs for a variety of applications, including AI and scientific computing. However, at this stage, the focus was more on scientific computations rather than AI specifically.

In the early 2010's, the deep learning revolution, sparked by the success of AlexNet in the ImageNet competition in 2012, showcased the power of GPUs in training neural networks. This event is often cited as when the broader AI community recognized the immense potential of GPUs for machine learning tasks.

Around this time, especially with the rise of Bitcoin, "miners" started to realize that GPUs were far more efficient than CPUs for the proof-of-work algorithms used in cryptocurrency mining.

By the mid-2010s, it was widely recognized in tech communities that GPUs were not just for graphics but were crucial for AI due to their ability to handle parallel computations much more efficiently than CPUs.

The GPU is the "muscle" of accelerated computing.

Matrix Math on the CPU

One of the key operations in AI computation is **matrix multiplication**. It is used extensively in training and inference (building and executing AI neural networks, respectively). All LLM's (e.g., ChatGPT, Grok, etc.) are neural networks, so too is Tesla's FSD (full self-driving), and just about anything that recognizes images, handwriting or voice.

Matrix multiplication is a great example that illustrates the power of the GPU.

A **matrix** is a rectangular arrangement of numbers, symbols, or expressions, organized in rows and columns. As shown in the Figure 1 below, this is a m x n matrix of size 5 x 4. It is a 2-dimensional (2D) matrix.

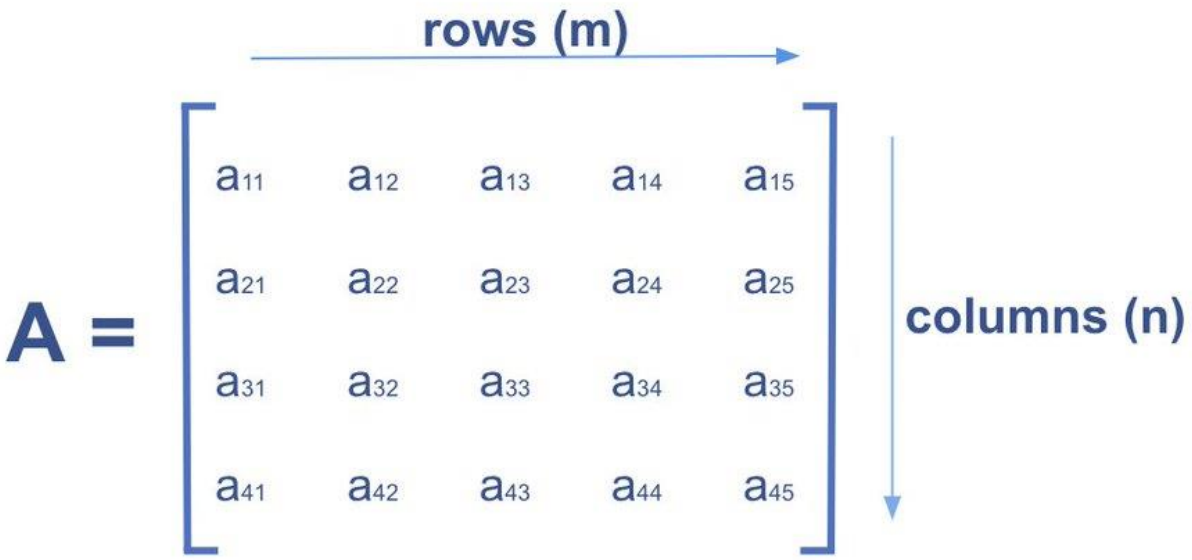


Figure 1: Example matrix

Matrices can be of higher dimension; for example, a 3-dimensional matrix denoted as m × n × p. These are often called **tensors** or **multidimensional arrays.**

To multiply two matrices, **A** and **B**, the number of **columns** in matrix **A** must be equal to the number of **rows** in matrix **B**. More specifically, if **A** is an **m x n** matrix (m rows, n columns), and **B** is an **n x p** matrix (n rows, p columns), then the product **AB** will be defined, resulting in an **m x p** matrix.

The element in the ith row and jth column of the resulting matrix **C** (where **C = AB**) is calculated as:

$$C_{ij} = \sum_{k=1}^{n} A_{ik} \cdot B_{kj}$$

Figure 2: Per element formula for matrix multiplication

In other words, *c11* are the 1st row elements of **A** multiplied by the 1st column elements of **B** and summed:

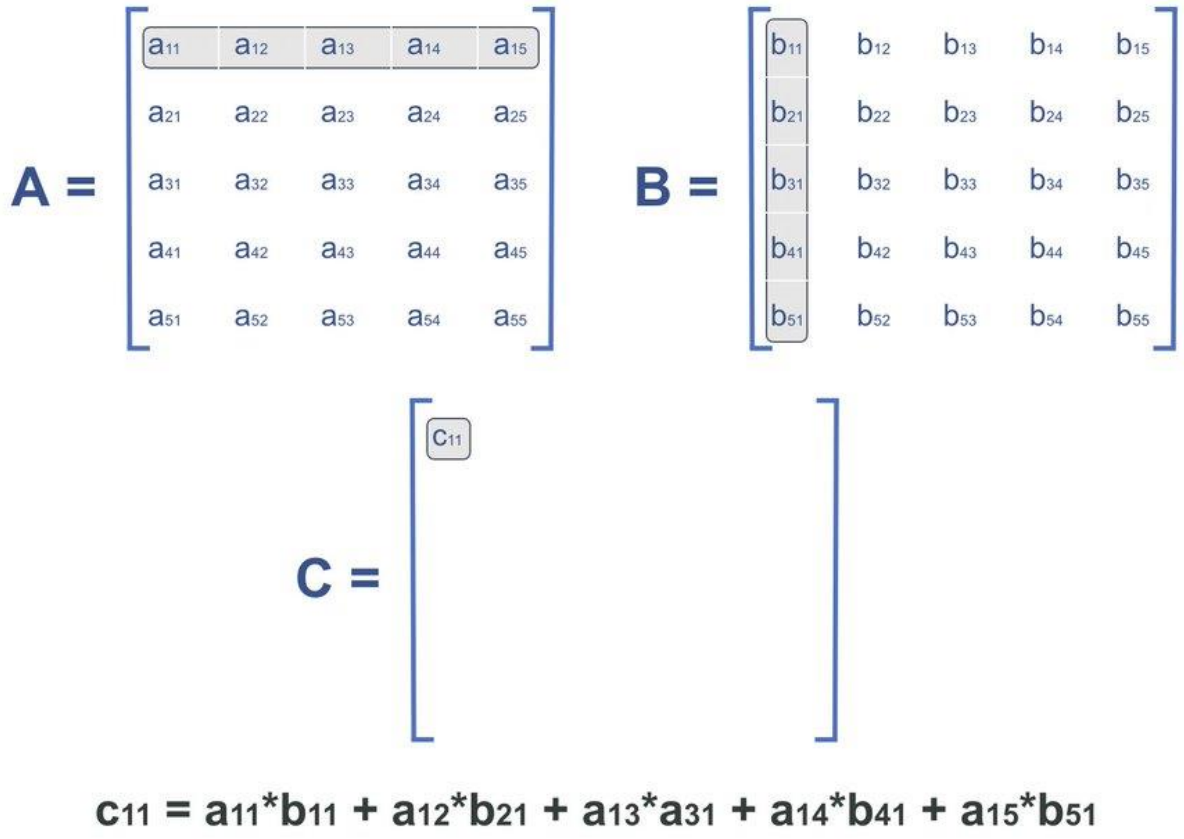


Figure 3: Matrix multiplication C = A * B

While the math is simple (multiplication and addition), there are *several hundred trillion* matrix multiplications performed on vastly larger matrices to train a LLM (large language model). And then billions more when the model is used (during inference).

Let's see how a CPU might do this.

Below is a function written in the C programming language that multiplies two matrixes **A** (of size M x N) and **B** (of size N x K) resulting in matrix **C** (of size M x K):

```
void matrixMultiply(int **C, int **A, int **B) {
    for (int i = 0; i < M; i++) {
        for (int j = 0; j < K; j++) {
            C[i][j] = 0;
            for (int x = 0; x < N; x++) {
                C[i][j] += A[i][x] * B[x][j];
            }
        }
    }
}
```

Figure 4: "C" code; matrix multiply function; CPU

The function is simply looping for each required element in matrix C. Thus, the line **C[i][j] += A[i][x] * B[x][j];** is executed **M * K * N** times.

This function of course gets compiled into machine code for the target processor. Below is the machine code[2] for an ARM-based CPU:

```
; Assume registers for loop counters and pointers:
; r0-r3 for loop counters i, j, x, and temporary
; r4-r9 for storing pointers to C, A, B respectively
; r10 for the temporary result of multiplication

matrixDotProduct:
    ; Assume M, K, N are passed in registers or loaded from memory
    ; Load M, K, N into registers r11, r12, r13
    LDR r11, =M  ; Load M
    LDR r12, =K  ; Load K
    LDR r13, =N  ; Load N

    ; Initialize outer loop counter (i)
    MOV r0, #0  ; i = 0

outer_loop:
    CMP r0, r11  ; Compare i with M
    BGE end_outer_loop  ; If i >= M, exit outer loop

    ; Initialize middle loop counter (j)
    MOV r1, #0  ; j = 0

middle_loop:
    CMP r1, r12  ; Compare j with K
    BGE end_middle_loop  ; If j >= K, exit middle loop

    ; Reset C[i][j] to zero
    MOV r10, #0  ; Temporary sum for C[i][j]
    STR r10, [r4, r0, LSL #2, r1, LSL #2]  ; Store 0 to C[i][j], assumir

    ; Initialize inner loop counter (x)
    MOV r2, #0  ; x = 0

inner_loop:
    CMP r2, r13  ; Compare x with N
    BGE end_inner_loop  ; If x >= N, exit inner loop

    ; Load A[i][x]
    LDR r6, [r5, r0, LSL #2, r2, LSL #2]  ; Load A[i][x]

    ; Load B[x][j]
    LDR r7, [r6, r2, LSL #2, r1, LSL #2]  ; Load B[x][j]

    ; Multiply A[i][x] * B[x][j]
    MUL r10, r6, r7

    ; Add to C[i][j]
    LDR r8, [r4, r0, LSL #2, r1, LSL #2]  ; Load current value of C[i][j
    ADD r8, r8, r10  ; Add to the sum
    STR r8, [r4, r0, LSL #2, r1, LSL #2]  ; Store back

    ADD r2, r2, #1  ; Increment x
    B inner_loop

end_inner_loop:
    ADD r1, r1, #1  ; Increment j
    B middle_loop

end_middle_loop:
    ADD r0, r0, #1  ; Increment i
    B outer_loop

end_outer_loop:
    ; Function return
    MOV pc, lr  ; Return from subroutine
```

Figure 5: machine code; matrix multiply function; CPU

In the machine code above, the number of instructions executed is: (**10 * N + 6 * K + 6 * M) * M * K**. For example, if we multiply a **100 x 100** matrix by a 1**00 X 1** matrix, **160,600** ARM instructions are executed.

ADD and **MUL** are the needed math operations, they are executed inside the ALU (Arithmetic Logic Unit) of the CPU. They are relatively fast (think 1 processor cycle each). But in training LLMs the matrix elements are floating point numbers not integers (typically floating point with 16-bit precision). Floating point operations are 3 to 5 times slower than their integer equivalents.

And while math on the data is fast, getting the data isn't. And that begs the question: where is the the data to begin with?

Latency refers to the time delay before a system can produce a result. There are two types of latency related to CPU performance: **compute latency**, which is the time it takes the CPU to process and compute a result, and **data latency**, which is the time it takes for data to be transferred to the CPU's compute units for processing.

Compute latency is *minor* compared to data latency.

But where is the data?

Data can originate from storage, either over a network or locally on the compute device. Retrieving data from any type of storage is relatively slow. Once retrieved, the data is first loaded into **DRAM** (Dynamic Random Access Memory), which serves as the computer's main memory. From there, the data may be further loaded into the CPU's caches (L1, L2, L3), which are types of **SRAM** (Static Random Access Memory), to speed up access during computation.

The speed of light is approximately 300,000,000 meters per second. Assuming a typical CPU operates at 3 GHz (3,000,000,000 cycles per second), light would travel about 100 mm (roughly 4 inches) in a single clock cycle. However, electrical signals in silicon travel significantly slower—around 60,000,000 meters per second. In practice, this means that data moves about 20 mm (about 0.8 inches) per clock cycle.

A modern CPU is typically housed on a silicon die measuring around 35 mm by 25 mm. Therefore moving data from one side of the chip to the other will take around 1 clock cycle alone, which introduces noticeable latency. Now, consider if the data is in DRAM—located a few inches away from the CPU—the delays grow even more substantial.

Modern CPUs contain multiple cores (typically 8 to 16)—independent processing units that can execute instructions simultaneously. Each core can handle its own stream of instructions, allowing for greater parallel processing.

Matrix multiplication is highly parallelizable because computing each element of the result matrix (e.g., c11, c12, etc.) can be done independently. To take advantage of this in code, matrix multiplication can be rewritten to use threads[3]. Each thread, running on a separate core, computes a specific element of matrix C. Multiple threads can operate simultaneously, and of course this yields a faster overall result.

Even with 16 cores, the maximum performance speedup would be limited to around 16x. While modern CPU architectures are designed to reduce latency and improve processing efficiency, the *laws of physics* impose inherent limitations on how quickly data can be transferred to the compute elements responsible for operations like multiplication and addition.

In practice, a significant portion of a CPU's time is spent **waiting for data to arrive from memory**. Data latency is *the* significant bottleneck in matrix multiplication on a CPU.

Matrix Math on the GPU

So how is data latency combated to produce a quicker result? The choices are: optimize to reduce latency or simply apply more parallelism, turning the processor into a throughput machine. With CPUs, optimization has reached diminishing returns-- Moore's Law[4] has seemingly run its course. And parallelism is limited.

Let's turn to the GPU, it's a different beast.

Much like a CPU, a GPU contain multiple processing cores, but many more. On NVIDIA GPUs[5] these are called SMs (Streaming Multiprocessors). The A100 NVIDIA GPU contains 108 SMs. The register file[6] per SM is 256kB and the L1 Cache and Shared Memory per SM is 192kB. The L2 cache is 40MB, shared across all SMs. This is to say that the GPU maintains a lot of memory *close* to the processing cores (to reduce data latency).

The NVIDIA A100 GPU maintains **64 warps** per SM (organized in blocks), and each warp consists of **32 threads**. With 108 SMs on the A100, that results in a total of **221,184 threads** across the entire GPU (64 warps × 32 threads × 108 SMs).

Each clock cycle, a warp is scheduled for execution. The 32 threads within a warp all execute a common instruction *simultaneously*, but each operates on different data, following the **SIMT** (Single Instruction, Multiple Threads) model. Context switching between warps is *virtually instantaneous*, allowing efficient handling of latency, such as memory access delays.

There are many more threads waiting (alive) than executing and that is on purpose. The GPU is said to be *oversubscribed* and by being oversubscribed the GPU always has work to do while its fetching more data from memory.

This high level of parallelism increases throughput, meaning more operations are performed in a given period, which helps hide or mask latency.

Let's look at a code example:

Using NVIDIA's CUDA[7] framework, below is code of the matrix multiply function that takes advantage of the GPU.

```
__global__ void matrixMultiplication(const float* A, const float* B, float* C, int numARows, int numAColumns, int numBRows, int numBColumns) {
    int row = blockIdx.y * blockDim.y + threadIdx.y;
    int col = blockIdx.x * blockDim.x + threadIdx.x;

    if (row < numARows && col < numBColumns) {
        float sum = 0;
        for (int i = 0; i < numAColumns; ++i) {
            sum += A[row * numAColumns + i] * B[i * numBColumns + col];
        }
        C[row * numBColumns + col] = sum;
    }
}
```

Figure 6: "C" code; matrix multiply function; GPU

The keyword **__global__** indicates that this function will be scheduled to run on the GPU (while being called from the CPU). The line **int row = blockIdx.y * blockDim.y + threadIdx.y;** calculates the specific thread's row index by combining its block and thread indices, helping to determine which portion of the data this thread will process. Similarly for the column.

But we can do better.

The A100 has **tensor cores**. Tensor cores are specialized hardware units designed to accelerate matrix operations. They perform matrix multiplications in a *single operation*.

The code below illustrates the matrix multiplication function using the A100's tensor cores:

```
__global__ void matrixMultiplicationTensorCore(half *A, half *B, float *C, int M, int K, int N) {
    // Declare the fragments
    wmma::fragment<wmma::matrix_a, 16, 16, 16, half, wmma::row_major> a_frag;
    wmma::fragment<wmma::matrix_b, 16, 16, 16, half, wmma::row_major> b_frag;
    wmma::fragment<wmma::accumulator, 16, 16, 16, float> c_frag;

    // Initialize the output to zero
    wmma::fill_fragment(c_frag, 0.0f);

    // Calculate global thread coordinates
    int warpM = (blockIdx.x * blockDim.x + threadIdx.x) / warpSize;
    int warpN = blockIdx.y * blockDim.y + threadIdx.y;

    // Loop over K
    for (int k_step = 0; k_step < K; k_step += 16) {
        // Load fragments
        int a_row = warpM * 16;
        int a_col = k_step;
        int b_row = k_step;
        int b_col = warpN * 16;

        // Check if we're within bounds
        if (a_row < M && a_col < K) {
            wmma::load_matrix_sync(a_frag, A + a_row * K + a_col, K);
        }
        if (b_col < N && b_row < K) {
            wmma::load_matrix_sync(b_frag, B + b_row * N + b_col, N);
        }

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

    // Store the result
    int c_row = warpM * 16;
    int c_col = warpN * 16;
    if (c_row < M && c_col < N) {
        wmma::store_matrix_sync(C + c_row * N + c_col, c_frag, N, wmma::mem_row_major);
    }
}
```

Figure 7: "C" code; matrix multiply function; GPU Tensor Core

The reference to **16** in this code is due to the dimensions of the **Tensor Core tiles** (or fragments) that are used in NVIDIA's WMMA (Warp Matrix Multiply and Accumulate) API, specifically for matrix multiplication operations. The tensor cores on NVIDIA GPUs, like the A100, process data in **16 x 16 tiles** (or blocks) of matrix elements.

The key function call is **wmma::mma_sync(c_frag, a_frag, b_frag, c_frag);** which invokes the tensor code matrix multiply function.

Using the NVIDIA tensor core model for a matrix multiply operations is between 2 to 3 orders of magnitude faster (between **100X** to **1000X** faster) than using a CPU (depending on the size of the matrices). That's quite a difference!

Matrix Math in Large Language Models

If you are familiar with chatGPT you are familiar with **large language models** (or LLMs). LLMs allow users to interact with a knowledgeable conversationalist that can provide information, generate text, answer questions, or assist with tasks across a vast range of topics.

Training

A large language model (LLM) is built through a process called **training**, where vast amounts of text (e.g., from the internet or libraries) is processed to learn patterns in language. The model is a neural network consisting of billions of parameters that capture relationships between words, context, and meaning. These parameters enable the model to generate statistically likely sequences of words based on the input, while also preserving the broader context.

In November 2022, OpenAI released ChatGPT, based on the **GPT-3.5** model.

Let's characterize the number of matrix multiplication operations involved in the training process for the GPT-3.5 model (without getting into details of the neural network itself).

The GPT-3.5 model has **96** layers, a hidden dimension of **12,288**, and a sequence length of **1024**. We will assume a batch size of **64** sequences, each with **1024** tokens.

A large language model (LLM) consists of several matrices that make up the neural network:

- **Embedding Matrix**
- **Self-Attention Matrices (Q, K, V)**
- **Feed-Forward Matrices**
- **Output Projection Matrix**

With an assumed vocabulary size of **50,000**, the embedding matrix size is **50,000 × 12,288**. The embedding matrix is multiplied by a token vector of size **12,288 × 1** (a vector is just a 1-dimensional matrix). For one batch, this results in **50,000 * 12,288 * 64** multiplications (which is **3.2 billion** multiplications).

For each layer (**96** layers), there are **3** matrix multiplications for the self-attention matrices Q, K and V. Each matrix of size **12,288 × 12,288** is multiplied by the hidden states of size **1024 × 12,288**. The total multiplications for self-attention per layer is **3 * (1024 * 12,2882 * 12,882)**. For **96** layers, this amounts to about **1.44 trillion** multiplications.

The feed-forward matrices are sized at **12,288 × 49,152** and **49,152 × 12,288**. For each token, this requires **2** matrix multiplications. Per layer, the operations are:

1. **1024 * (12,288 * 49,152)**
2. **1024 * (49,152 * 12,288)**

This results in approximately **1.22 trillion** multiplications per layer. Across **96** layers, that totals about **117.12 trillion** multiplications.

Finally, the output projection matrix is sized at **12,288 × 50,000** and is multiplied by the final hidden state of size **1024 × 12,288**. This totals 1**024 * (12,288 * 50,000)** multiplications, which is around **629 billion** multiplications.

To summarize the multiplications per forward pass:

- Embedding Matrix: **~3.2 billion**
- Self-Attention Matrices: **~1.44 trillion**
- Feed-Forward Matrices: **~117.12 trillion**
- Output Projection Matrix: **~629 billion**

For **64** sequences each with **1024** tokens, the total number of multiplications is:

3.2 billion + **1.44 trillion** + **117.12 trillion** + **629 billion** ≈ **119.1 trillion**

During training, each input requires both a forward pass and a backward pass. The backward pass requires about twice the number of operations as the forward pass. Therefore, we have approximately **3 * 119.1 trillion** ≈ **357.3 trillion** matrix multiplications.

Assuming **300 billion** tokens, and each batch processing **65,536** tokens, the total number of training steps is:

300 billion tokens / **65,536** tokens per batch ≈ **4.58 million** steps

So the total number of matrix multiplications is:

357.3 trillion matrix multiplications per step × **4.58 million** steps ≈ **1,630,000,000,000,000,000,000** (**1,630 quintillion**) matrix multiplications.

That's a lot!

Inference

Neural networks are *asymmetric* algorithms. The training process, which occurs once (or infrequently), builds the model. After training, the model is used to perform **inference** during *each subsequent use*.

Let's characterize the number of matrix multiplication operations involved in an inference process for the GPT-3.5 model.

Imagine we issue the prompt "**What is the circumference and diameter of Earth?**". This prompt consists of **8** words or tokens[8].

Even though training is more computationally demanding, inference still uses a huge amount of matrix multiplications due to the sheer size of the model and the complexity of the architecture.

The embedding matrix is **50,000 × 12,288**. The number of matrix multiplications for **8** tokens is **8 * (50,000 * 12,288) ≈ 4.91 billion**.

The self-attention matrices Q, K, and V are of size **12,288 × 12,288**. Therefore the number of matrix multiplications for **8** tokens is **3 * (8 * (12,288 * 12,288)) ≈ 3.62 billion** per layer. But since there are **96** layers this is: **96 * 3.62 billion ≈ 348.86 billion**.

For the first feed-forward matrix of size **12,288 × 49,152** and for **8** tokens that is **8 * (12,288 * 49,152) = 4.82 billion**. For the second feed-forward matrix of size **49,152 × 12,288** and for **8** tokens that is **8 * (49,152 * 12,288) = 4.82 billion.** The total is **2 * 4.82 billion = 9.66 billion.** But since there are **96** layers this is: **96 * 9.66 billion ≈ 927.36 billion**.

Finally for the output projection matrix of size **12,288 × 50,000** and for **8** tokens that is **8 * (12,288 * 50,000) ≈ 4.91 billion.**

4.91 billion + 348.86 billion + 927.36 billion + 4.91 billion = 1.286 trillion

Also a lot!

But as we know a GPU is fast at matrix multiplications. A NVIDIA A100 GPU can deliver up to **312** teraflops for FP16 operations (using tensor cores), with a few calculations (not presented), this query can be run in ~**28.3 milliseconds**. (NVIDIA's new "Blackwell" GPU can do this 5x faster, achieving about **1,500** teraflops).

But we must account for data latency. If we assume that all data elements are FP16 (16-bit floating point numbers) the matrices required add up to **234.7 GB** of data**.** If the data is transferred to the GPU using the fastest possible data link, NVIDIA's NVLink with bandwidth of **~900 GB/s**, then the data latency will accumulate to **261 ms**.

28.3 ms (compute latency) + **261 ms** (data latency) = **289.3 ms**. The data latency accounts for over 90% of the "processing" time.

1.286 trillion matrix multiplications in **289.3 ms** (or **.28 seconds**). Pretty fast!

Accelerated Computing in the Data Center

In short, accelerated computing is fundamentally about **parallelism**. While we've discussed how GPUs function as parallel processing devices, it's important to recognize that training a model as large as GPT-3.5 would require far more than a single GPU—it demands a vast array of them working in parallel.

For example, GPT-3, with its **175 billion** parameters, required an immense amount of computational power. The estimated compute needed to train GPT-3 is approximately **3640 petaflop/s-days**[9] (assuming FP16 precision).

Given that an NVIDIA A100 GPU can deliver up to **312 teraflops** for FP16 operations (using tensor cores), training GPT-3 likely involved between 1,024 and 2,048 A100 GPUs, distributed across hundreds of nodes in a large-scale cloud or data center environment.

And if inference takes place in the cloud (e.g., chatGPT presents a web-based interface to the user or provides an app, but all queries are executed in the cloud), then a large number of CPU/GPU servers (a server farm) must be reserved to handle the inference load.

More FLOPS?

FLOPS refers to floating-point operations per second and is often pushed as the key metric for accelerated computing. While FLOPS measure the computational capability of a CPU or GPU, they don't tell the whole story. *FLOPS don't matter*. In real-world applications, processors spend much of their time waiting for data rather than performing computations.

Data latency— the time it takes to move data between different levels of memory or between memory and the processor— is the real bottleneck limiting throughput. While parallelism doesn't solve the latency problem, it helps mask it by allowing for simultaneous computations and data transfers. This means that while one process is waiting for data, others can continue executing.

The key challenge is moving data as close to the processor cores as possible to reduce latency. GPUs have specialized memory structures for this purpose, including register memory for fast, core-local storage, and shared memory for efficient data exchange between threads, reducing the need to access slower global memory.

Closing Thoughts

Technological advancements often follow a "chicken and egg" cycle, where progress in one area depends on developments in another. Take machine learning, for instance— a subset of artificial intelligence (AI) that has driven innovations like OpenAI's ChatGPT and Tesla's Full Self-Driving

(FSD). Machine learning relies heavily on data, and more data generally leads to better outcomes. A key component, particularly in neural networks, is the compute-intensive training phase, where input data is used to build statistical models that map inputs to outcomes, replacing traditional algorithmic approaches.

However, without the rise of large-scale data centers— commonly known as cloud computing— there would be no infrastructure to support the thousands of servers required for accelerated computing. Similarly, without the vast storage capabilities of these data centers, there would be nowhere to house the immense datasets needed for AI training. Many AI techniques in use today were invented decades ago but lacked the computational and storage resources to be practically implemented at the time.

A key technological breakthrough that has enabled this leap forward is the use of Graphics Processing Units (GPUs). Originally designed to render graphics in parallel, GPUs have become indispensable in machine learning because their architecture is optimized for parallelism. Unlike CPUs, which excel at handling a few tasks sequentially (and generally provide the brains of the operation), GPUs can perform thousands of operations simultaneously. This parallelism is crucial in the training phase of neural networks, where many computations—like matrix multiplications—can be distributed across hundreds or thousands of cores, dramatically accelerating processing speed.

While parallelism doesn't eliminate data latency issues, it helps mitigate them. GPUs enable the simultaneous movement and processing of data, allowing computations to continue even when some tasks are stalled waiting for data. Furthermore, GPUs feature specialized memory structures designed to keep data close to the cores, minimizing the need to access slower, distant memory, and thus masking some of the latency.

In essence, the synergy between massive parallelism in GPUs and the robust infrastructure of cloud data centers—enabling scalable computation, data access, and connectivity—has redefined the *realm of possibility* in AI. Problems that were once considered insurmountable due to the sheer volume of data and computational demands can now be tackled head-on. Where traditional computing systems would have faltered, this new infrastructure has unlocked the ability to build models on unprecedented scales, making breakthroughs in areas like natural language processing and autonomous driving not just faster, but achievable.

And this technological foundation will continue to evolve. In parallel.

NOTES:

[1] This is not exactly correct. A RISC processor (Reduced Instruction Set Computing) typically will execute 1 instruction per cycle, but due to pipelining it can execute more. A CISC processor

(Complex Instruction Set Computing) may take 1 or more cycles to execute an instruction. ARM processors are RISC, Intel (i386) processors are CISC.

[2] Technically what is show in *assembly*. This is a human-readable mnemonic representation of machine code. Each line of assembly corresponds directly to one or more machine instructions but is still in a format that humans can understand and write. e.g., **mov x19, x0** translates to AA 00 00 20 (ARM64 instruction format for mov x19, x0)

[3] A thread is the smallest unit of processing that can be scheduled by an operating system. It represents a sequence of instructions that can be executed independently of other threads.

[4] Moore's Law is the observation that the number of transistors on a microchip doubles approximately every two years, leading to an exponential increase in computing power, while the cost halves.

[5] The discussion of GPUs will focus NVIDIA. In particular the A100 series GPU based on the Ampere architecture. This A100 was introduced in 2020. In 2022, the H100 GPU was released, based on the Hopper architecture. And recently the B100, based on the Blackwell architecture.

[6] Registers are small, high-speed storage locations located within the central processing unit (CPU) of a computer. GPUs have registers too.

[7] CUDA (Compute Unified Device Architecture) is a parallel computing platform and programming model developed by NVIDIA that enables developers to leverage the massive computational power of NVIDIA GPUs for general-purpose processing. It allows tasks to be broken down into many smaller parallel tasks that run efficiently on the GPU's thousands of cores. With CUDA, developers can write programs in familiar languages like C, C++, and Python.

[8] In a neural network, **tokens** are the smallest units of data (like words or subwords) that the model processes. These tokens are converted into numerical representations (embeddings) and passed through layers of the network to make predictions or generate outputs. Tokenization is essential for transforming raw input data (like text) into a format the neural network can understand.

[9] 3640 petaflop/s-days conveys that you have a system that can execute 3640 petaflop/s of performance sustained for 1 day.