Bits, FLOPS, and Watts: A Systems-Level Perspective of Scaling LLMs

Asheesh Goja
April 2025
San Jose, CA

Introduction

Scaling Large Language Models (LLMs) goes beyond simply increasing parameter counts. It emerges from a complex interplay of hardware, software, and algorithmic optimizations. These deep neural networks operate in high-dimensional vector spaces with complex loss landscapes and scaling power laws, where learning unfolds along intricate manifolds that often defy human intuition. Yet their existence is tethered to silicon and electricity—bound by thermodynamics, semiconductor physics, and the finite resources of our four-dimensional world. Ultimately, the mathematics of LLMs is bound by the very physics of silicon.

To understand their evolution, we must trace the journey of neural architectures from simple single-layer perceptrons to today's trillion-parameter Transformers. At every stage, algorithmic breakthroughs and hardware advancements have propelled each other forward. As transistor scaling, memory bandwidth, and parallelism improved, each wave of innovation pushed the boundaries of what neural networks could achieve, reinforcing the deep synergy between software and silicon. Today's relentless drive to build ever-larger LLMs stands upon that same foundation, revealing that the future of LLM scaling depends not on software or hardware alone, but on their careful co-design, emphasizing the need for a systems-level perspective on scaling.

Adopting such a perspective, the triad of Bits, FLOPS, and Watts provides a systems-level lens to understand the scaling dynamics of LLMs. Information capacity, measured in **bits**, is determined by parameter count, weight precision, model architecture, and data entropy, all of which collectively define a model's representational power. Computational throughput, measured in **FLOPS**, directly influences both training and inference speed, shaping the model's efficiency and responsiveness. And **watts** measure energy consumption, highlighting the growing power costs and associated economic implications of scaling up LLMs. Each bit, woven in LLMs vast network of weights, biases and activations, demands energy to store and process, unfolding into a cascade of FLOPS orchestrated by increasingly complex and power-hungry hardware. At the same time, the silent but absolute arbiter of watts sets the ultimate boundary for this computational symphony, reminding us that unlimited scaling remains tethered to the thermodynamic constraints of our physical universe. That limit is not arbitrary. Landauer's principle, a cornerstone of information theory, states that erasing even a single bit of information carries an irreducible energy cost. While we are orders of magnitude above this limit, it highlights the intimate connection between information, computation, and thermodynamics. As models scale to trillions of parameters, each bit processed accumulates a thermodynamic debt that must be repaid through heat dissipation. Absent rigorous hardware software co-design and efficient energy management, the trillion-parameter LLM era risks sliding into thermodynamic insolvency.

This challenge underscores the interplay between information capacity, computational throughput, and energy consumption. Optimizing these three dimensions in tandem is crucial. It spurs the development of novel architectures, new algorithms, and hardware innovations that cater to all three resource demands holistically. This two-part series takes a deep dive into these challenges from both software and hardware standpoints.

Part 1 Preview: The Laws of Scaling

We will begin part 1 by tracing the evolution of LLMs, highlighting the critical role of software and hardware co-design in shaping deep neural networks. From there, we develop an intuitive understanding of neural scaling laws and their relationships with model performance, compute, data, model size, and the emergence of intelligence. We will also examine how these scaling laws extend to inference-time compute in models like o1/o3 and DeepSeek-R1, and how they relate to advanced reasoning capabilities such as Chain-of-Thought (CoT). Throughout this exploration, we introduce pivotal theoretical concepts such as irreducible entropy, the efficient-compute frontier, emergent abilities, and the manifold hypothesis. These concepts will illuminate both the extraordinary potential and the fundamental limitations of LLMs.

Next, we conduct a thorough analysis of the **Transformer** architecture, uncovering its theoretical underpinnings and the mechanisms by which it generates language with unreasonable effectiveness. We explore why Transformers serve as the backbone of LLMs and how its design choices influence scalability, especially the significant demands placed on the underlying compute infrastructure by Transformers inherent algorithmic complexity. We will focus on topics such as the arithmetic intensity of self-attention, the space complexity of the KV cache, the overhead of non-linear operations, and the role of sparsity in attention. This exploration will reveal how this complexity necessitates increasing amounts of memory (bits), computational throughput (FLOPS), and, most critically, leads to the unsustainable cost of power (watts) required to train and run today's trillion-parameter LLMs.

By the conclusion of Part 1, we establish a central tension at the heart of LLM advancement: while neural scaling laws push us inexorably toward ever-larger models in our quest for Artificial General Intelligence (AGI), the algorithmic complexity of Transformers creates physical bottlenecks that cannot be overcome through software optimization alone. This comprehensive framework sets the stage for Part 2, where we will analyze the critical triad of bits, FLOPS, and watts from a systems-level perspective, exploring potential pathways forward in addressing these challenges.

Part 2 Preview: The Walls of Silicon

In Part 2, we explore how the fundamental interplay between bits, FLOPS, and watts erects three critical barriers in GPU-based accelerated computing: the **Memory Wall, Power Wall, and Interconnect Wall**. These barriers aren't merely engineering challenges, they represent fundamental physical principles governing semiconductor technology and information processing. At their core, these walls arise because information in physical form requires space (memory), energy (power), and efficient pathways (interconnects)—all finite resources constrained by transistor density, heat dissipation limits, and the speed of light. In silicon-based computers operating at the nanometer scale, electrons simultaneously serve as both carriers of information and energy, embodying bits and watts within a single quantum entity.

Our exploration begins with an examination of how modern GPUs operate, focusing on execution flow, tensor cores, and warp schedulers. We'll analyze a kernel performing WMMA (warp matrix multiply-accumulate) operation using low-level PTX ISA, revealing how the "**Memory Wall**" naturally emerges from the classical Von Neumann architecture. This wall represents the growing disparity between computational capability and memory performance—a gap that widens as processing power increases. To understand this challenge more deeply, we'll examine hierarchical memory structures, from on-chip SRAM to off-chip HBM, and use frameworks like the Berkeley Roofline Model to demonstrate how this bottleneck becomes particularly pronounced in Transformer-based LLMs, preventing GPUs from achieving their theoretical compute throughput.

As we push against memory limitations, the typical response is to scale up individual GPUs with more compute and memory. However, this approach eventually hits insurmountable limits due to photolithographic reticle size constraints and the complexities of advanced multi-chip module packaging techniques like CoWoS. These physical constraints force a shift toward scale-out architectures with massive GPU clusters. This transition, however, introduces our second major barrier: the **Interconnect Wall**. Coordinating computations and exchanging data across thousands of interconnected GPUs creates new bottlenecks, as LLM parallelization strategies demand extreme bandwidth and low latency. We'll examine how these requirements push interconnect technologies like NVLink and InfiniBand to their physical limits, further constraining our ability to scale efficiently.

The third constraint, the **Power Wall**, not only drives power consumption toward **gigawatt** levels in future GPU clusters but also prevents individual GPUs from hitting their theoretical peak in high-intensity LLM workloads. By examining the physics of CMOS technology and mathematical relationship between clock frequency, voltage, and power consumption, we discover why performance gains inevitably lead to large increases in power draw. Because a significant portion of that power turns into heat, TDP (Thermal Design Power) becomes an unyielding limit, prompting solutions like Dynamic Voltage and Frequency Scaling (DVFS) to prevent overheating at the cost of peak GPU performance. Beyond raw FLOPS, we'll also explore why data movement can consume orders of magnitude more energy than the computations themselves, creating a persistent efficiency bottleneck despite extraordinary GPU power.

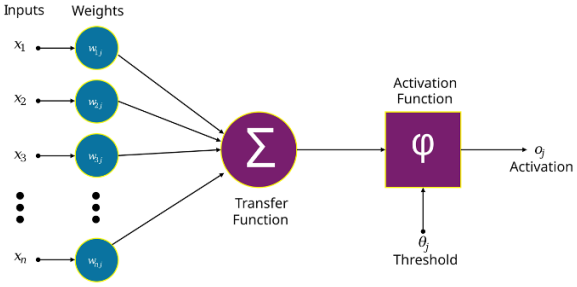
By the end of Part 2, it will be clear that these three "Walls of Silicon" fundamentally constrain our ambitions to scale LLMs. The path forward demands more than just optimizing our existing GPU hardware—it calls for truly novel compute architectures, new memory technologies, advanced interconnects, or alternative computing paradigms.

Part 1

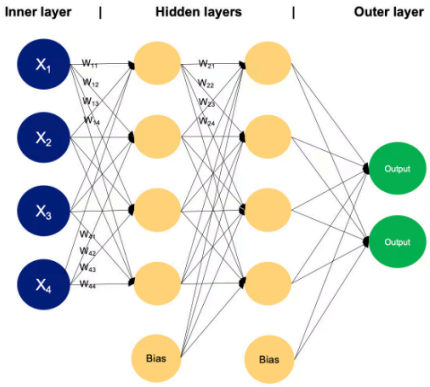
Laws Of Scaling

Early Neural Architectures - From Perceptrons to LSTMs

The journey of neural network development began with the simplest form: single-layer perceptrons, designed for basic classification tasks. These early models, while conceptually groundbreaking, quickly encountered limitations, particularly their inability to solve non-linear problems. A notable example was their failure to learn the XOR function, a fundamental logical operation. This limitation, among others, contributed to a period of reduced research activity known as the first "AI Winter."



Perceptron



MultiLayer Perceptron

The field experienced a resurgence with the development of Multilayer Perceptrons (MLPs) and the introduction of the backpropagation algorithm. Backpropagation, a method for calculating gradients of the loss function with respect to network weights, enabled the training of these deeper networks. However, progress was hampered by the limited processing capabilities of CPUs in the 1990s. These computational constraints restricted both the size and complexity of networks that could be practically trained and deployed. Despite these hardware limitations, the combination of multi-layer architectures and gradient-based optimization through backpropagation demonstrated the vast potential of neural networks and laid crucial theoretical foundations for future advancements.

A profound transformation occurred with the advent of General Purpose GPU (GPGPU) computing. GPUs, originally designed for graphics processing, possess a highly parallel architecture that proved to be exceptionally well-suited for the matrix and vector operations that dominate neural network training. This hardware advancement enabled the practical implementation of Convolutional Neural Networks (CNNs). AlexNet's groundbreaking performance in the 2012 ImageNet competition, served as a powerful demonstration of the capabilities of CNNs when combined with GPU acceleration.

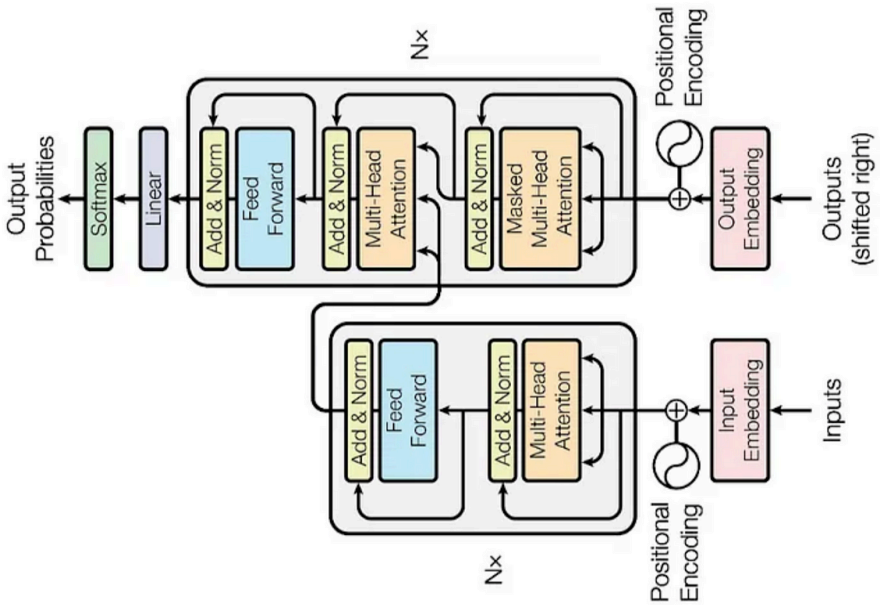
In parallel, sequential data processing advanced through Recurrent Neural Networks (RNNs) and Long Short-Term Memory (LSTM) architectures, with LSTM's gating mechanisms providing a solution to significantly mitigate vanishing gradient issues (shrinking of gradients during training, constraining model's ability to capture long-range dependencies). However, processing

very long sequences remained a challenge due to memory constraints and the difficulty of preserving information over many time steps. Also the inherently sequential processing nature of RNNs and LSTMs did not align well with the parallel processing capabilities of modern GPUs, leading to inefficiencies in training and inference.

Throughout these developments, it became evident that advancements in algorithms were inherently tied to hardware capabilities. Each theoretical breakthrough demanded corresponding improvements in processing power, memory capacity, and specialized hardware optimized for the linear algebra computations at the core of neural networks—all while balancing memory throughput, floating-point operations, and power constraints.

The Transformer Revolution

The introduction of the Transformer architecture in 2017 marked a pivotal shift in natural language processing and beyond. Its core innovation was the replacement of recurrent processing with self-attention, a mechanism that allows models to process all tokens in a sequence concurrently. This enabled parallel computation, aligning seamlessly with the Single Instruction, Multiple Thread (SIMT) architecture of GPUs, leading to dramatic speedups in training. By eliminating the hidden state bottleneck of RNNs, Transformers unlocked scalability to hundreds of layers and billions of parameters.



While parallelization increased computational throughput and improved FLOPS efficiency, it also introduced significant memory demands as model sizes grew. To address this, specialized hardware blocks such as tensor cores were developed to optimize the mixed-precision matrix multiplications at the heart of Transformer computations, illustrating the co-evolution of hardware and software to accommodate ever-growing model complexity.

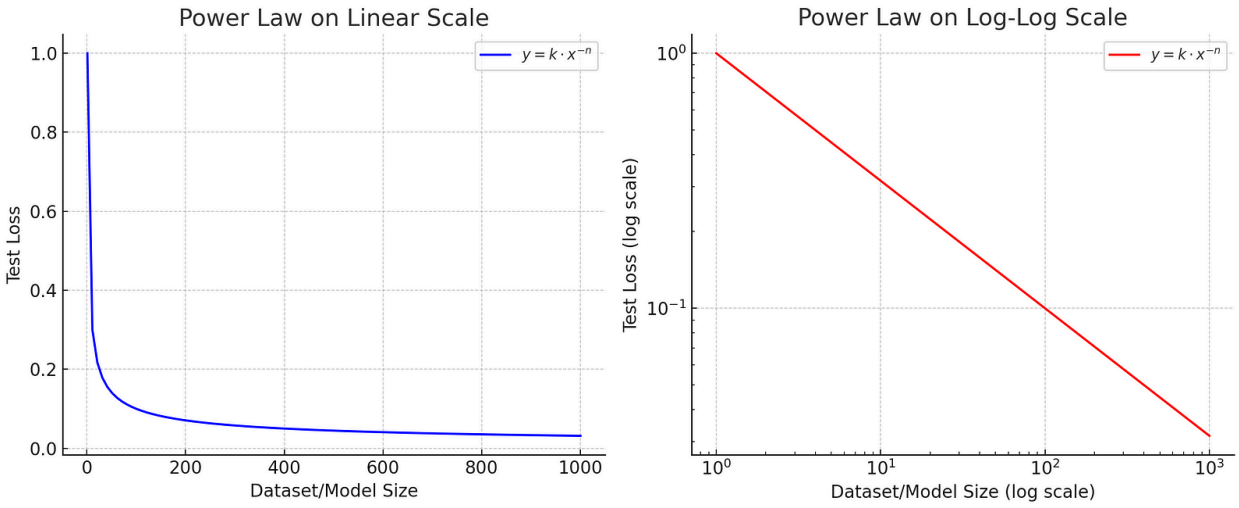
LLMs have demonstrated that increasing the number of layers and parameters can lead to significant improvements in performance, where performance refers to a model's ability to effectively complete its intended tasks. This scaling has allowed models to tackle problems once deemed too complex for neural networks. However, this raises a fundamental question: *how exactly does performance improve as models scale?* Understanding the relationship between model size and its effectiveness is crucial, and this is where scaling laws come into play.

The Laws of Scale

Scaling laws provide a framework for understanding how model size, dataset size, and computational budget collectively influence overall performance. LLMs have consistently followed a "bigger is better" trend, where increasing these factors leads to measurable improvements. These neural scaling laws have proven remarkably predictive across architectures and tasks, not only mapping out how performance improves but also revealing the points where gains may start to plateau. By examining the mathematics behind these laws, we can see how they unify seemingly unrelated concepts such as test loss, irreducible entropy in natural data, and the interplay between model, dataset, and compute size into a cohesive understanding of LLM scaling behavior.

Power Laws

In the early days of deep learning, researchers made a crucial observation. When they plotted test loss, which measures the error between a model's predictions and actual values, against dataset size or model size on logarithmic scales, the data points formed remarkably straight lines. This linear trend on a log-log plot is a hallmark of an inverse power-law relationship, which can be mathematically expressed as: $\mathbf{y = k \cdot x^{-n}}$



Here, ***x*** can represent the model size (***N***), the dataset size (***D***), or the total compute used (**C**). ***y*** typically represents the validation or test cross-entropy loss, a key metric for evaluating the

performance of LLMs. The negative power exponent (-***n***) is of particular interest, as it quantifies the rate at which the loss decreases as *N*, *D*, or *C* increases.

The practical significance of this power-law relationship is immense. Researchers can fit this equation to the results of relatively small-scale, and therefore less expensive, experiments. They can then use the fitted equation to extrapolate and predict the performance of much larger, more expensive runs.

This predictive capability is not only scientifically illuminating but also economically advantageous. It allows research teams to forecast performance gains, manage financial risks associated with large-scale training runs, and allocate resources effectively, ensuring the right proportion of data tokens to model parameters.

Cross-Entropy Loss

When training LLMs, a fundamental question arises: *how do different aspects of performance improve as we scale up the models?* Cross-entropy loss serves as one of our primary metrics for measuring this improvement. It evaluates how accurately a model can predict the next token in a text sequence by comparing the model's predicted probability distribution to the actual distribution of the next token. During training, when the model generates a probability distribution over its entire vocabulary, cross-entropy quantifies the discrepancy between this predicted distribution and the true distribution of the next token.

The cross-entropy loss for a single prediction is calculated as $-\log(p)$, where p represents the probability the model assigned to the correct next token. The use of the negative logarithm imposes steep penalties on predictions that are far off, meaning the loss decreases towards zero as the model becomes more confident and accurate in its predictions. However, the inherent ambiguity and variability of natural language mean that even massive models cannot achieve a perfect cross-entropy score of 0.0 on typical text datasets. This is because, for any given context, there are often multiple plausible continuations.

This fundamental limitation indicates that while scaling laws demonstrate consistent reductions in cross-entropy as we increase computational resources, training data, and model parameters, there exists a minimum threshold that cannot be crossed through additional training. The characteristics and exact value of this threshold are directly related to the concept of irreducible entropy, which we will explore next.

The Role of Irreducible Entropy

Even with virtually unlimited parameters, vast datasets, and boundless computational resources, LLMs cannot reduce the cross-entropy loss to zero on natural language. This inherent limitation is tied to what researchers call the irreducible entropy of the data. Natural text, as well as other modalities like images and audio, often exhibits multiple equally valid continuations or encodings. Language is replete with synonyms, colloquialisms, and stylistic flourishes that make the concept of "the single correct next word" inherently ambiguous.

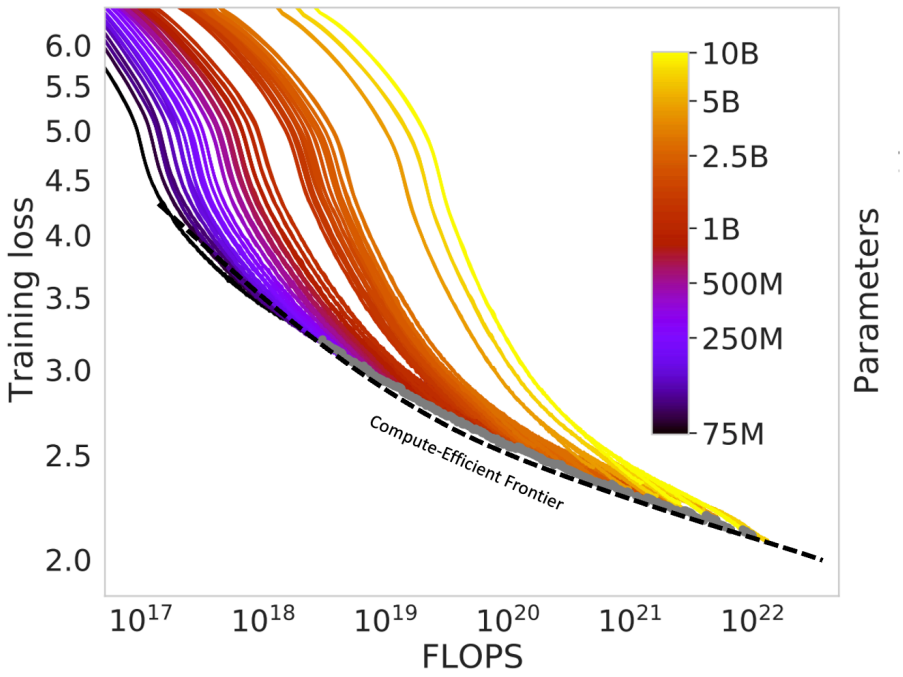
One way to see irreducible entropy is to look at where the loss curves “bend.” In an idealized scenario, if the data were fully deterministic, the loss trend might continue straight toward zero. In reality, these curves eventually flatten, revealing a nonzero lower bound. While early scaling experiments by OpenAI found no obvious leveling off for large-scale language data, subsequent research from DeepMind suggested that there may be at least a small curving in the scaling plots, indicating a nonzero asymptote.

The precise value of language’s irreducible entropy remains an active area of research. Theoretical work suggests it might be around **1.69 nats or 2.44 bits** (equivalent to an average cross-entropy loss of 1.69 on a dataset), but conclusively determining this value would necessitate experiments even larger than current capabilities allow.

Understanding irreducible entropy is crucial because it establishes the **ultimate performance ceiling** that LLM can achieve. Beyond this ceiling, additional scaling will not yield further improvements in terms of cross-entropy loss.

The Efficient-Compute Frontier

When we plot model performance (such as cross-entropy loss or error rate) against computational resources on logarithmic scales, we often observe multiple descending lines. Each line typically represents a distinct model architecture or a specific set of parameters. These lines illustrate how increasing the computational budget allocated for training tends to reduce the cross-entropy loss or error rate for models of a given size. However, each line eventually plateaus, reflecting the diminishing returns that occur as we continue to invest more compute into training a model of a fixed size.



By examining multiple such curves, each representing a different model size, we can identify what's known as the **efficient-compute frontier.** This frontier represents the outer boundary connecting the most efficient points across all the curves. These are the points that achieve the minimum error for any given computational budget. When aiming for a specific performance target, the goal is to identify the point on this frontier that reaches the target with the least amount of computational resources.

The power of this frontier lies in its ability to inform resource allocation decisions. If we under-invest in data relative to model size, or vice versa, we will find ourselves operating "inside" the frontier, effectively wasting computational resources.

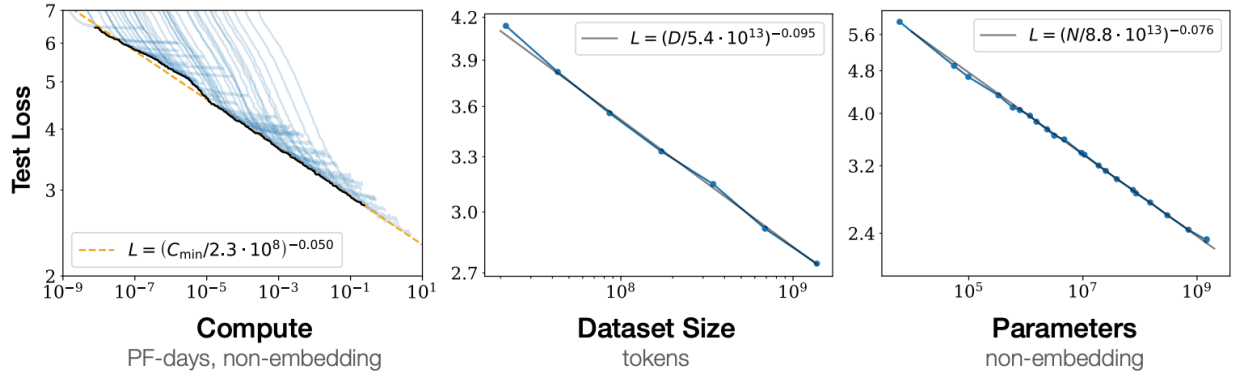
The frontier thus provides a generalized answer to a crucial question in LLM development: **should we train a very large model for a smaller number of steps, or a smaller model for a larger number of steps?** The optimal answer depends on where we aim to land along the efficient-compute boundary.

Kaplan Scaling Laws

In a seminal 2020 paper titled "Scaling Laws for Neural Language Models," researchers at OpenAI, led by Jared Kaplan, made a groundbreaking observation. They found that the error of a language model, as measured by cross-entropy loss, follows a power-law relationship with respect to three key factors: model size (*N*), dataset size (*D*), and the amount of training compute (C) measured in PF-days (PF-day = $10^{15} \times 24 \times 3600 = 8.64 \times 10^{19}$ FLOP). The simplest form of their empirical result can be expressed as:

$$\text{Loss}(N, D, C) \sim \left(\frac{1}{N^{\alpha_N}} + \frac{1}{D^{\alpha_D}} + \frac{1}{C^{\alpha_C}} \right),$$

Here αN, αD, and αC are exponents that quantify the sensitivity of model loss to changes in model size, dataset size, and compute resources.



When these relationships are visualized on logarithmic scales, they manifest as straight lines, with each exponent determining the slope of the corresponding line. A striking finding from the Kaplan team's experiments was the strong sensitivity of loss to model size (N). Based on these findings, they concluded that if one had to choose a single dimension to scale within a fixed compute budget, increasing the model size would be the most effective strategy for enhancing performance.

This "bigger is better" philosophy quickly shaped AI research landscape. In the same year, OpenAI scaled from models in the billions of parameters to GPT-3 with 175 billion parameters. The remarkable alignment between the theoretical predictions of the scaling laws and the real-world performance of GPT-3 fueled the notion that these laws are robust and that continuing to scale up model size is a straightforward path to pushing the boundaries of AI capabilities.

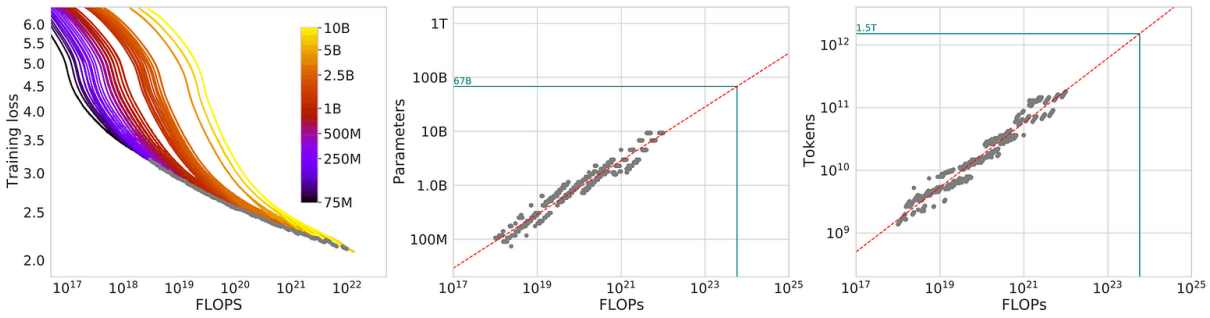
Chinchilla Scaling Laws

While the original Kaplan laws emphasized increasing model size, a later study by Hoffmann et al. (nicknamed **Chinchilla**) found that training data quantity is just as critical. They proposed revised scaling laws, described by the following equation:

$$L(N, D) = E + \frac{A}{N^{\alpha}} + \frac{B}{D^{\beta}}.$$

where ***L(N, D)*** represents the cross-entropy loss as a function of model size ***(N)*** and dataset size ***(D)***. The term ***E*** is the irreducible loss, representing the inherent entropy of the data. Constants ***A*** and ***B*** capture the contributions of model size and dataset size, while ***α*** and ***β*** serve as scaling exponents, quantifying the sensitivity of the loss to variations in ***N*** and ***D***, respectively.

This equation highlights a key insight: for sufficiently large models and efficient training, the optimal strategy is to scale both the model and the data together. They discovered that to minimize loss for a given compute budget, parameter count (***N***) and dataset size (***D***) should be scaled in tandem. If either is neglected, you leave performance "on the table."



The irreducible term (***E***) in the Chinchilla equation defines a fundamental limit on language modeling performance, tied directly to irreducible entropy. It implies that beyond a certain point, simply adding more parameters or data won't yield further loss improvements. While increasing model or dataset size can significantly reduce loss (represented by the power-law term), the irreducible term establishes a finite lower bound on achievable loss.

Chinchilla's main contribution was identifying the "**compute at parity**" sweet spot: the optimal balance between model size and training data volume that maximizes the return on each FLOP. Over-investing in parameters without sufficient data leads to inefficiency, as the model cannot fully utilize its capacity. Conversely, a small model cannot fully exploit a large dataset, resulting in diminishing returns.

This perspective doesn't invalidate Kaplan's findings but refines them. It demonstrates that the optimal scaling exponent depends on the training regime. Larger models not only require more compute but also proportionally more data to reach their full potential.

Beyond Train-Time Compute: Test-Time Compute Scaling

Foundational work such as the Kaplan and Chinchilla scaling laws guides us on how to balance model size and data to maximize performance during pre-training. However, the conversation is no longer restricted to pre-training alone. An emerging paradigm, often called *test-time* or *inference-time* compute scaling, has begun to reframe the way we think about model capacity and performance. A 2024 paper by Snell et al. proposes that strategically allocating computational resources during inference can significantly boost LLMs performance, sometimes even surpassing the gains achieved by simply scaling up pre-training.

The core idea is to adopt a "**compute-optimal**" approach during inference, dynamically adjusting the amount of computational effort based on the complexity of each prompt. Imagine a model that can decide, for each query it receives, how much "thinking time" it needs. This strategy can outperform simpler methods, like "best-of-N" sampling (where a model generates N responses and picks the best one), and can even rival or exceed the performance of models that are significantly larger in terms of pre-training parameters, depending on the query's nature and the available inference budget. While this work doesn't present a single, closed-form law like those of Kaplan or Chinchilla, it establishes a conceptual framework for scaling computation intelligently during inference.

Formally, if $y^*(q)$ denotes the correct answer for a query ***q*** and ***θ*** represents a particular search policy (method plus hyperparameters), then the compute-optimal scaling strategy $\theta_{(q,y)}^*(q)^*(N)$, aims to satisfy:

$$\theta^*_{q,a^*(q)}(N) = \text{argmax}_\theta \left(\mathbb{E}_{y\sim\text{Target}(\theta,N,q)}\left[\mathbb{1}_{y=y^*(q)}\right]\right)$$

where ***N*** is the fixed test-time compute budget. In practice, each query ***q*** has a difficulty that can be estimated (e.g., via pass rates or an internal difficulty scorer). One can then adaptively select how best to spend the test-time budget ***N***. For easier queries, few or no refinements may be necessary. For more challenging queries, the strategy might spawn multiple solution paths or iteratively refine a single path until it converges on a plausible solution.

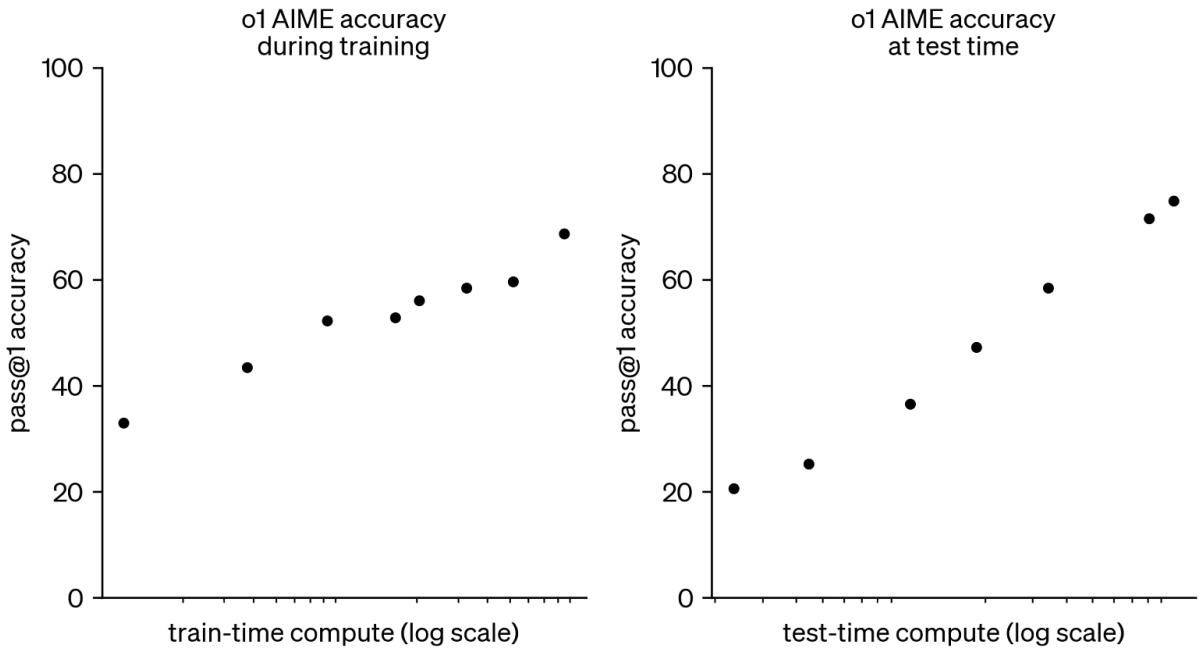
Test-time compute scaling often falls into two broad categories:

- **Verifiers**, which involve searching over partial solutions, generate and evaluate multiple solution paths using dedicated models or heuristics to select the best option. Methods range from simple Best-of-N sampling with outcome reward models (ORMs) to more complex approaches like beam search and Monte Carlo Tree Search (MCTS) that use process reward models (PRMs) to score and prune intermediate reasoning steps.
- **Revisions**, or iteratively improving the model's output distribution, focus on refining a single solution through techniques like self-reflection, token re-ranking, and targeted rewriting of problematic sections. The model iteratively improves its output by identifying and correcting errors or suboptimal paths.

The theoretical frameworks for test-time compute scaling have moved from research into industry implementations, with OpenAI's "o1" and DeepSeek-R1 serving as prime examples of different approaches to deploying these concepts at scale.

OpenAI's "o1/o3"

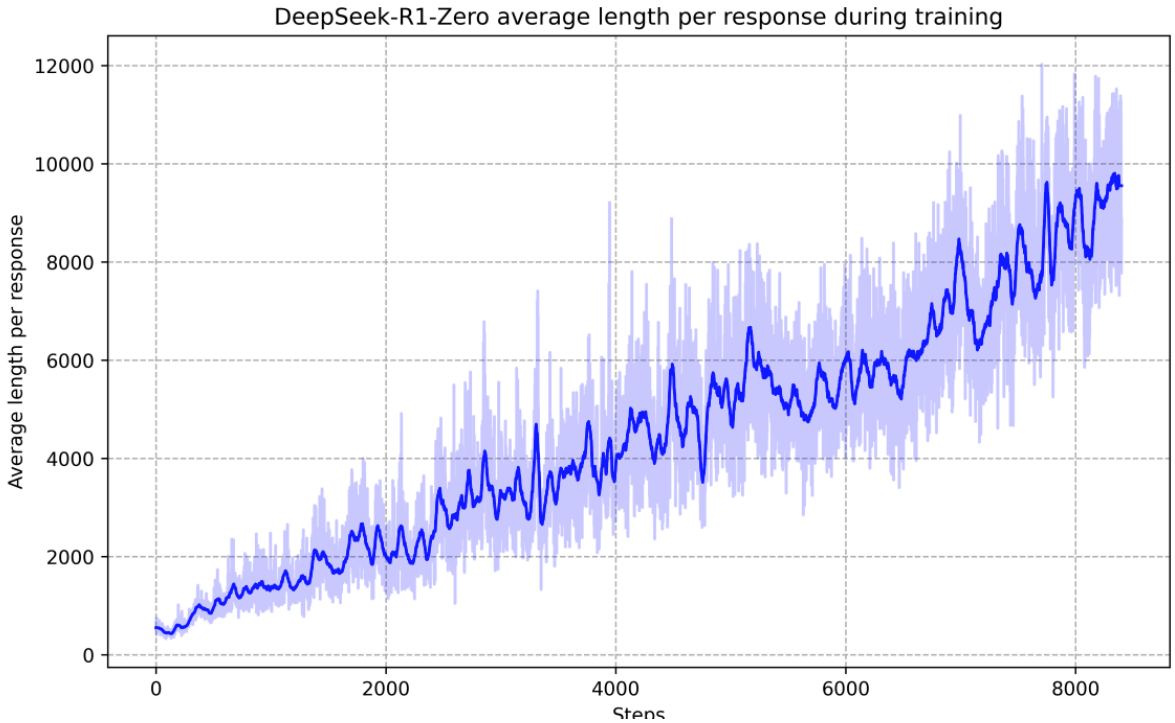
o1 showcases test-time compute capabilities through its dynamic, step-by-step reasoning approach. While the model isn't fully open-sourced, its problem-solving methodology, particularly in mathematics, clearly demonstrates this systematic reasoning process. The model's effectiveness increases with the number of permitted reasoning steps, suggesting a sophisticated internal mechanism. Early analysis indicates that o1 employs something akin to self-verification or chain-of-thought refinement, iteratively developing and critiquing partial solutions before synthesizing a final, coherent answer. This capability is further evidenced by o3's impressive 87.5% score on the ARC-AGI benchmark, marking a significant leap forward in AI's ability to tackle novel and unfamiliar challenges.



Researchers have proposed that o1 likely employs a hybrid approach: generating multiple solution paths for complex questions while streamlining the process for simpler ones. This adaptive strategy aligns with the compute-optimal approach that different problems warrant different amounts of test-time computational resources.

DeepSeek-R1

DeepSeek-R1 takes a novel approach by applying reinforcement learning (RL) directly to improve reasoning capabilities without relying heavily on external verification or supervised fine-tuning (SFT). While DeepSeek researchers initially explored PRMs and MCTS, they found these approaches had significant limitations. PRMs struggled with defining and scoring intermediate reasoning steps and risked reward hacking, while MCTS became impractical due to the vast token-level search space and challenges in training fine-grained value models. Instead, through pure RL training with simple rule-based rewards for accuracy and format, the model naturally learned to leverage extended test-time computation by generating longer chains-of-thought (CoT) and incorporating self-verification behaviors.



The average response length increases significantly during RL training, reaching thousands of tokens as the model learns to break down complex problems and verify its own reasoning. What makes this particularly remarkable is the emergence of sophisticated behaviors during test-time computation. The model spontaneously develops reflection capabilities—revisiting and reevaluating its previous steps—and explores alternative problem-solving approaches without explicit programming.

These **emergent behaviors** arise purely from the model's interaction with the reinforcement learning environment, where increased **test-time compute** leads to more sophisticated reasoning.

Emergent Abilities

The scaling laws discussed above describe how LLMs steadily reduce cross-entropy loss or other error metrics with increasing data, model size, and compute. However, a fascinating and somewhat mysterious aspect of LLM behavior is the emergence of qualitatively new abilities as models scale up. These are often referred to as "emergent abilities". An ability is emergent if it is not present in smaller models but suddenly appear in larger ones. These abilities include multi-step reasoning (e.g., chain-of-thought), instruction following, arithmetic problem-solving, improved truthfulness in answers, enhanced semantic understanding, and the ability to handle diverse knowledge tasks across multiple languages and domains. These capabilities often appear abruptly and were either entirely absent or only rudimentary in smaller models.

What is particularly intriguing about these emergent abilities is that they do not map neatly onto the smooth, predictable power-law declines in cross-entropy loss described by the scaling laws.

Instead, they seem to manifest somewhat abruptly, suggesting the existence of thresholds or phase transitions in model capabilities as a function of scale.

> However, while Scaling laws reliably predict **quantitative improvements**, such as reduced test loss with increased compute, they do not guarantee specific **qualitative changes** in model behavior. While larger models become more robust and expressive, emergent capabilities remain unpredictable, making it unclear how scaling translates into intelligence or complex reasoning abilities.

These emergent behaviors are still connected to the training regime. A model that is trained in a balanced manner, with appropriate scaling of both data and parameters, is more likely to exhibit these emergent capabilities than a model that is trained in an unbalanced way.

The Manifold Hypothesis

Why do these empirical power laws arise so consistently across different models and datasets? One compelling explanation invokes the **manifold hypothesis**, a concept that has gained traction in the field of deep learning. This hypothesis posits that natural data, such as text, images, and audio, tends to lie on a lower-dimensional manifold embedded within a much higher-dimensional space. For instance, consider a dataset of 28x28 pixel grayscale images. Each image can be represented as a point in a 784-dimensional space (28 * 28 = 784). However, real-world images do not uniformly fill this entire 784-dimensional space. Instead, they tend to cluster along a manifold with a much lower intrinsic dimensionality. This is because natural images share common structures and patterns, and not all possible combinations of pixel values correspond to meaningful images.

Deep neural networks learn to approximate the geometry of this manifold. As model size (N) or dataset size (D) grows, the network can represent the manifold with finer resolution, thereby reducing error. Simple geometric arguments suggest that the average distance between data points on a manifold of dimension (d) scales as $D^{-1/d}$. Cross-entropy loss, which effectively squares these distance errors, can scale as $D^{-4/d}$. This insight helps explain why feeding LLMs ever more data yields a predictable drop in loss, even in extremely high-dimensional settings. For text, estimating the intrinsic dimension is difficult. Early attempts suggest a very high dimension (d≈100), which complicates perfect fits.

However, despite some alignment with real-world experiments, no closed-form “unified theory” of neural scaling has been universally validated. The manifold hypothesis provides a powerful heuristic for understanding why larger datasets and models generally lead to improved performance. It suggests that we can conceptualize the process of scaling as learning increasingly refined approximations of a complex, high-dimensional manifold. However, this learning process is still fundamentally bounded by the irreducible entropy inherent in the language data.

Having established how scaling laws dictate the balance among model performance, size, data, and compute, we now turn to the blueprint that has made such ambitious scaling feasible. At the

heart of LLMs lies the **Transformer Architecture**, a design whose self-attention mechanism not only unlocks parallelized training but also overcomes the long-range dependency issues that held back its sequential predecessors. By examining how Transformers process language we can see exactly why this architecture forms the backbone of today's LLMs.

The Transformer Architecture

The introduction of the Transformer architecture in "Attention Is All You Need" revolutionized natural language processing (NLP). While previous approaches like RNNs and LSTMs processed text sequentially, limiting their scalability on modern GPUs, Transformers introduced a groundbreaking parallel processing approach through self-attention. This mechanism allows the model to simultaneously analyze relationships between all tokens in a sequence, significantly improving both training efficiency and contextual understanding.

Despite the proliferation of different model sizes and training objectives, the core Transformer architecture has maintained its elegant simplicity through three key stages. In the input embedding stage, data from various modalities (text, images, audio) is converted into high-dimensional vector representations. These embeddings then pass through transformer blocks, where multi-headed self-attention and feed-forward layers iteratively refine them by capturing complex patterns and relationships. Finally, the output processing stage maps these refined embeddings into the desired predictions.

This versatile architecture comes in three main variants, each specialized for different tasks:

- Encoder-only models like BERT and RoBERTa excel at classification tasks by generating fixed-size representations of input data.
- Encoder-decoder models like T5, BART, and mT5 power machine translation by using separate components to process source and target languages.
- Decoder-only models like GPT, LLaMA, and Claude focus on text generation, making them the primary choice for LLMs.

For building modern LLMs, the decoder-only variant has emerged as the dominant architecture. Its ability to generate coherent text from input prompts, combined with the Transformer's inherent parallel processing capabilities, makes it particularly well-suited for training efficient and powerful language models on GPU hardware. Let's examine the core building blocks of this architecture:

Input Embedding

The foundation of Transformer processing lies in token embeddings, which convert raw inputs, such as words, into dense numerical vectors within a high-dimensional space. Think of this space as a complex semantic map, where each dimension represents a distinct aspect of meaning. By placing tokens at specific coordinates within this map, the embeddings capture subtle relationships between different pieces of information. This mathematical representation

allows neural networks to grasp the nuances of language, as words with similar meanings naturally cluster together in the embedding space.

The specific technique for creating these embeddings varies depending on the type of input being processed, whether it's text, images, or other forms of data. Text is broken into subword tokens using methods like Byte Pair Encoding (BPE), which are then converted to vectors. Images are divided into patches that are flattened and projected into embeddings. Audio is first transformed into spectrograms, which can then be processed similarly to images by splitting them into patches. Once this conversion is complete, a single batch of input can contain token embeddings derived from text, images, audio, or any other modality, all ready to be consumed by the Transformer blocks.

Positional Encoding

While the core self-attention mechanism within transformer blocks treats inputs as unordered sets of embeddings, language requires a precise understanding of word order to convey meaning. Without maintaining sequential position information for each token, the model would lose crucial context and grammatical structure. To address this, various positional encoding strategies have been developed. Early approaches focused on absolute positions, such as learned positional embeddings (unique vectors trained for each position up to a maximum length) and fixed sinusoidal encodings (deterministic vectors based on varying frequencies). In these absolute methods, the positional vectors are typically added directly to the token embeddings before the first transformer block, maintaining the embedding dimensionality.

However, many modern approaches emphasize relative positioning to better capture pairwise relationships and improve generalization. Relative Positional Encoding (RPE) encompasses techniques that encode the distance or relationship between tokens, often by modifying attention scores directly or using specific relative embeddings within the attention calculation itself. Furthermore, Rotary Positional Encoding (RoPE), injects relative positional information in a unique way by applying position-dependent rotations to the query and key vectors before the attention computation. These diverse mechanisms are crucial for enabling Transformers to effectively process sequence data.

While text-based applications typically use one-dimensional embeddings to capture linear sequence order, image processing often employs two-dimensional position embeddings to represent both height and width coordinates. This positional encoding mechanism elegantly bridges the gap between naturally ordered data and the set-based nature of self-attention, enabling the model to effectively process sequential, grid-based, or higher-dimensional structured inputs.

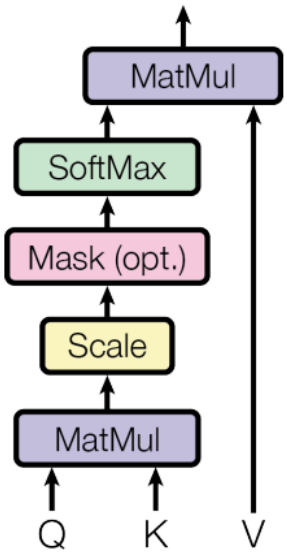
The Self-Attention Operation

After embedding the tokens and adding positional information, we arrive at the heart of the Transformer, the attention mechanism. This ingenious system allows tokens to interact and share information with each other, creating a rich understanding of context. The process works

by transforming each token's embedding (a row in the input matrix X) into three distinct, learned representations using separate weight matrices:

- **Queries (Q)**: Generated by multiplying the input X by a learned weight matrix WQ. Think of this as a token formulating a question about what information it needs from others.
- **Keys (K)**: Generated by multiplying X by a different learned weight matrix WK. This acts like a label or descriptor for the token, indicating the kind of information it holds, making it discoverable by relevant queries.
- **Values (V)**: Generated by multiplying X by a third learned weight matrix WV. This contains the actual information or features of the token that will be shared.

Think of this as a sophisticated communication system. Each token uses its query to determine which other tokens it wants to gather information from, while its key helps determine which tokens will seek information from it. When a token's query aligns strongly with another token's key, it receives more of that token's value, the actual information being shared. Through these learned projections (WQ, WK, WV), the model creates dynamic pathways for information to flow between tokens, allowing them to build a nuanced understanding of their relationships within the sequence. This attention-based approach is what gives Transformers their remarkable ability to process language with such sophistication.



Scaled Dot Product Attention

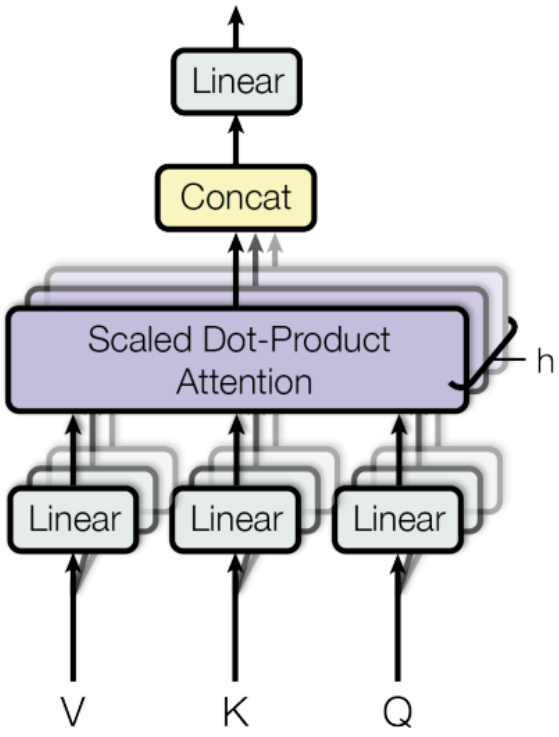
Scaled dot-product attention is the mathematical realization of this process. It computes an attention matrix by taking the dot product of the queries and keys, resulting in scores for every pair of tokens. This matrix has dimensions **T×T** for a sequence of length **T** in self-attention. To stabilize training, the scores are divided by $\sqrt{\mathbf{d}_k}$, where $\mathbf{d}_k$ is the dimension of the key vectors and $\mathbf{K}^T$ is the transpose of the key vector **K**. Finally, a softmax function normalizes each row in the attention matrix, converting the raw dot products into a set of weights that sum to 1. These weights determine how much each token "pays attention" to every other token, and the

weighted sum of the value vectors (**V**) gives each token a refined representation based on context. Mathematically:

$$\text{Attention}(Q, K, V) = \text{softmax}(\frac{QK^T}{\sqrt{d_k}})V$$

Multi-Headed Attention (MHA)

While a single attention operation suffices to compute context-dependent representations, Transformers extend the concept with multiple heads. Rather than using one set of **Q, K**, and **V** projections, the embedding dimension is partitioned across several heads, and each head performs its own attention calculation on a subset of the dimension. This parallel processing enhances the model's ability to learn diverse relationships and focus on different aspects of the sequence simultaneously.



After each head computes its own weighted combination of the value vectors, the outputs of all heads are concatenated and passed through a linear projection that restores the dimension back to the original size. By operating across multiple subspaces, multi-head attention can capture varied token relationships that a single head might miss.

Masked Multi-Headed Self-Attention

While MHA captures global context, decoder-only Transformers must ensure that each position in the sequence only attends to itself and the tokens before it. This requirement is critical for autoregressive language generation, where a token should not “peek” at future positions. To impose this constraint, masked multi-headed self-attention introduces a causal mask $M \in \mathbb{R}^{n \times n}$ **(**a matrix with n rows and n columns) applied to the attention weights to disallow attention to future positions (causality). This is implemented as:

$$\text{MaskedAttention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}} + M\right)V$$

After forming the [T,T]-sized attention matrix by multiplying the queries (Q) with the transposed keys (**K^T)**, all entries corresponding to “future” tokens are masked out by setting them to negative infinity (or a very large negative constant) before the softmax operation. In other words, token t_i does not attend to positions j_i where **j > i** . Once the softmax is applied, those masked entries become effectively zero, ensuring each token can only gather information from itself and preceding tokens.

Key-Value (KV) Cache

MHA excels at capturing token dependencies, but its computational demands, especially during inference, have driven algorithmic innovations like the **Key-Value (KV) cache**. During text inference, recomputing K and V vectors for previously processed tokens at each step is inefficient. Instead, these vectors are stored in a KV cache, enabling reuse and significantly accelerating inference. However, as sequence length increases, the KV cache grows linearly, creating a memory bottleneck that impacts long-context LLMs. This memory bottleneck is precisely what the following MHA variants aim to address.

Multi-Query Attention (MQA)

MQA tackles this memory bottleneck by implementing a shared communication system. Rather than each attention head maintaining separate **K a**nd **V** vectors, MQA uses a single set shared across all heads. While each head retains its unique Query, they all attend over this common Key-Value space. This reduces the **KV cache** size and accelerates inference, particularly for long sequences and larger batch sizes. However, this efficiency comes at a cost, the shared Keys and Values limit the model’s ability to capture diverse relationships and nuanced information. MQA can also introduce instability during fine-tuning, especially with longer sequences.

Grouped-Query Attention (GQA)

GQA emerges as a middle ground between MHA and MQA. Instead of either complete independence (MHA) or total sharing (MQA) of KV pairs, GQA organizes attention heads into groups that share Keys and Values within each group. The number of groups (G) becomes a

important hyperparameter. When G equals the number of heads, GQA functions like MHA and when G is 1, it behaves like MQA. Setting G to an intermediate value (typically 4 to 8) achieves an optimal balance between generation quality and inference speed. This approach has been widely adopted in models like the Llama 3 series and Mistral 7B models.

Multi-Head Latent Attention (MLA)

MLA as implemented in DeepSeek models, takes a novel approach to Key-Value (KV) cache reduction through **Low-rank Key-Value Joint Compression**. For each token, MLA first projects the Keys and Values into a lower-dimensional latent space using a down-projection matrix shared across all attention heads. During inference, these compressed representations are reconstructed using up-projection matrices unique to each head, enabling each head to operate on individualized reconstructed Key and Value vectors. This strategy achieves a 57x reduction in KV cache size—from 4MB per token to just 70KB—while preserving or even improving model quality compared to traditional MHA or GQA.

Unlike MQA, where all heads share identical KV matrices (sacrificing specialization), or GQA, which shares KV matrices across subsets of heads, MLA maintains head-specific expressiveness by learning head-specific up-projection matrices. The compressed latent cache is thus shared, but its reconstruction into per-head Key and Value representations allows nuanced specialization. Crucially, DeepSeek optimizes the compute path: through clever linear algebra, they absorb the head-specific projection matrices into the query and output paths, meaning that these transformations are only computed once and do not add runtime overhead during inference.

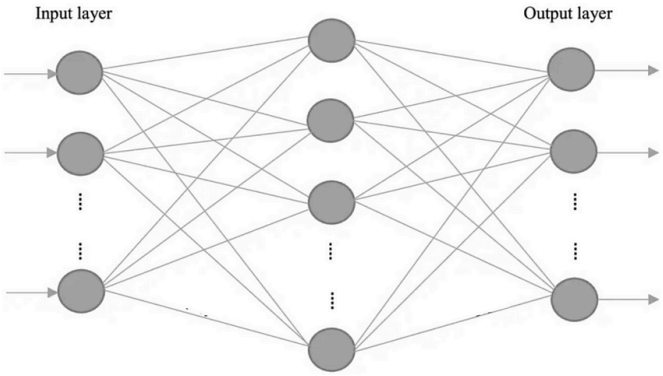
MLA fundamentally rethinks the Transformer's KV cache bottleneck. By learning how to efficiently compress and reconstruct attention information, it breaks the typical scaling dependency between KV cache size and the number of attention heads. This innovation enables DeepSeek R1 to scale to 100,000-token contexts using a manageable memory footprint, generating tokens 6x faster than standard Transformers, and redefines what is computationally feasible for high-context, high-performance LLMs.

Feed-Forward Network (FFN)

The feed-forward network (FFN) that follows the self-attention module serves a distinct but complementary purpose in processing token embeddings. While self-attention captures relationships between tokens across the sequence, the FFN operates on each token independently, focusing on enriching individual token representations.

In a typical decoder-only Transformer block, the FFN first projects each token's embedding into a higher-dimensional space, usually four times larger than the input. This expansion creates room for the network to learn more nuanced features. The expanded representation then passes through a non-linear activation function, such as GeLU, ReLU or SwiGLU, which introduces non-linear transformations. A second linear projection then compresses the representation back to its original dimensionality, maintaining consistency in the token

embeddings that flow between Transformer blocks. This architectural choice of expanding and then contracting the representation, with non-linearity in between, achieves an elegant balance between computational efficiency and expressiveness.



The synergy between attention and feed-forward layers creates a powerful processing pipeline: attention mechanisms discover how tokens relate to and influence each other, while FFNs deeply transform individual token representations to capture richer meanings. This relationship modeling in attention layers and individual token processing in FFNs forms the backbone of the Transformer's effectiveness.

Layer Normalization (Norm)

Layer Normalization stabilizes neural network activations, reducing internal covariate shift and accelerating convergence. Without normalization, the distribution of activations can shift dynamically across layers, making it harder for the network to learn effectively. In a decoder-only Transformer, LayerNorm computes the mean and variance along the embedding dimension of each token vector, rather than across a batch, ensuring stable normalization independent of batch size. Mathematically, LayerNorm is defined as:

$$\text{LayerNorm}(x) = \frac{x - \mu}{\sigma + \epsilon} \cdot \gamma + \beta$$

where **μ and σ** are the mean and standard deviation along the embedding dimension, **ϵ** is a small constant for numerical stability, and **γ and β** are learnable parameters that apply an affine transformation. The affine transformation ensures that the normalized output can still capture meaningful variations in the data, rather than simply standardizing it to zero mean and unit variance. RMSNorm is another popular alternative.

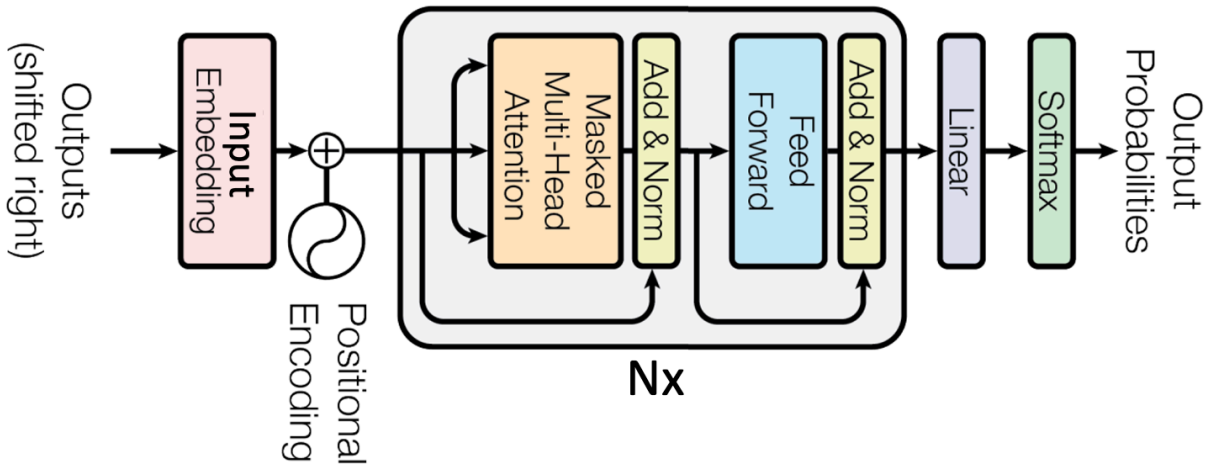
Residual Connections (Add)

Residual connections help mitigate vanishing and exploding gradients by providing a "shortcut" for gradient flow. After the self-attention or feed-forward sub-layer transforms the token vectors, the original input is **added** to that transformed output: **Output=Input+Layer(Input).**

Because residual connections preserve the dimensionality of their input, they integrate seamlessly with the self-attention and feed-forward modules within the Transformer block. They provide a “shortcut” for the gradient to flow back through the network during backpropagation. This helps to stabilize training and improve the overall performance of Transformer models.

Putting It All Together: The Decoder-Only Transformer Block

With these elements defined, we can now assemble the complete decoder-only Transformer block, the fundamental unit upon which these powerful LLMs are built. These blocks are stacked sequentially (**Nx**), creating a deep network capable of learning complex language representations. Each block processes the input it receives and passes its transformed output to the next block in the stack, preserving the dimensionality of the token embeddings throughout the process.



Each Transformer block consists of two primary processing stages:

1. **Masked Multi-Headed Self-Attention:** This stage is the heart of the Transformer’s ability to capture contextual relationships.
2. **Feed-Forward Network:** Following the attention stage, each token’s representation is independently processed by a FFN.

These two core stages are supported by two crucial mechanisms—**Add & Norm**:

- **Layer Normalization (Norm):** Layer normalization, whether applied before (Pre-LN) or after (Post-LN) each stage (of attention and FFNs), helps stabilize activation distributions.
- **Residual Connections (Add):** After each stage, the original input to that stage is added to its output.

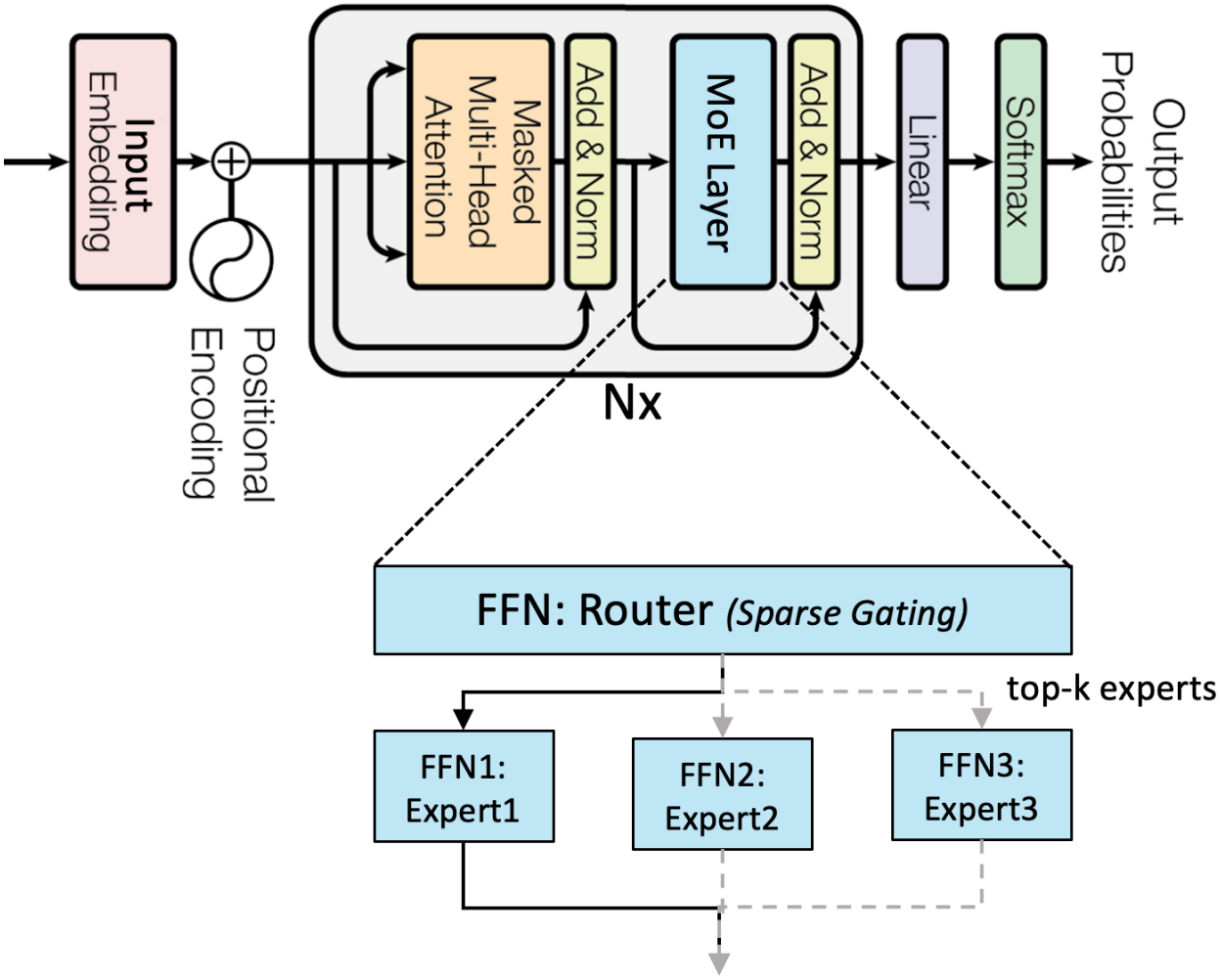
The output of a single Transformer block is a set of transformed token embeddings, which serves as the input to the next block in the sequence. Following the final block, a concluding linear layer—often called the 'output projection layer' or 'language modeling head'—processes the representations for all token positions. This layer maps each token's high-dimensional embedding to logits (raw, unnormalized scores), with one logit score per word in the model's vocabulary. While logits are generated for every position, during typical text generation, it's the logits corresponding to the last token of the input sequence that are converted via a softmax function into a probability distribution. This distribution is then used to predict or sample the single most likely next token to extend the sequence.

This modular design, with its repeating blocks of self-attention, feed-forward networks, layer normalization, and residual connections, is key to the Transformer's success. Having established the fundamental structure and functionality of the decoder-only Transformer block, let's now see how a powerful architectural innovation allows us to scale these models even further: the Mixture of Experts (MoE) approach, which introduces sparsity to enhance capacity while maintaining efficiency.

Mixture of Experts

Mixture of Experts (MoE) is a powerful technique for scaling Transformer models. It modifies the standard Transformer by substituting the FFN with a collection of "expert" FFNs, each specializing in a different aspect of the data. A "router," or gating network, intelligently directs each input token to one or a few of these experts. This routing is dynamic, based on learned scores that are typically normalized using a softmax function.

The core advantage of MoE lies in its **sparsity** as only a subset of experts is activated for each token. This contrasts sharply with **dense** FFNs, where all neurons are involved. For instance, models like GPT, LLaMA, and Claude use dense architectures where all parameters are active for every input. On the other hand, MoE models such as Mixtral 8x7B and DeepSeek-R1 activate only a subset of their total parameters for each input. Sparsity enables MoE models to significantly increase their capacity (number of parameters) without a proportional surge in computational demands during inference. To maintain efficiency, most MoE models employ sparse gating, selecting only the top-k experts (usually top-1 or top-2) for each token.



Load-balancing mechanisms, like auxiliary losses and capacity factors, ensure that all experts are used effectively and prevent the router from simply favoring a few popular experts. Although MoE can increase the memory footprint as all experts need to be loaded, it offers a way to enhance the Transformer's representational power without a linear increase in computational cost.

The Scaling Bottlenecks of Transformers

While the Transformer architecture, particularly its self-attention mechanism, offers significant advantages in parallel processing and capturing long-range dependencies, scaling these models to billions or trillions of parameters presents substantial computational and memory challenges. These challenges stem from the inherent complexity of the Transformer's components and their interactions with modern compute hardware.

Quadratic Complexity of Self-Attention

A significant bottleneck arises from the self-attention mechanism itself. Its computational cost scales quadratically with the sequence length (*n*). Specifically, the time complexity of multi-headed attention is $O(n^2 \cdot d + n \cdot d^2)$, where *d* is the embedding dimension. The $n^2 \cdot d$ term comes from calculating the attention scores between all pairs of tokens, while the $n \cdot d^2$ term accounts for the linear projections of tokens into queries, keys, and values.

Component	Time Complexity	Space Complexity
Multi-Head Attention (MHA)	$O(n^2 \cdot d + n \cdot d^2)$	$O(n^2 + n \cdot d)$
Feed-Forward Networks (FFNs)	$O(n \cdot d^2)$	$O(n \cdot d^2)$
Softmax (Non-linear)	$O(n^2)$	$O(n^2)$

Similarly, while the space complexity is $O(n^2 + n \cdot d)$, from storing the full attention matrix, modern implementations use optimizations like FlashAttention. These techniques cleverly avoid materializing the entire *n x n* matrix, mitigating the quadratic memory bottleneck and achieving practical memory usage closer to linear $O(n \cdot d)$ scaling with sequence length. Nonetheless, the computational complexity remains quadratic $O(n^2 \cdot d)$. This means that doubling the sequence length, even if memory scaling is managed, still roughly quadruples the computational load required for self-attention, posing a major obstacle for processing very long sequences.

Arithmetic Intensity of Self-Attention

The performance of Transformer models is influenced not only by the quadratic cost of self-attention but also by low arithmetic intensity, particularly during inference. Arithmetic intensity refers to the ratio of compute operations (like matrix multiplications in MHA and FFN) to memory accesses, like reading Q, K, V vectors or loading parameters from GPU memory.

GPUs, given their graphics lineage, are designed for high arithmetic intensity, where workloads maximize computational throughput relative to memory accesses. They achieve peak efficiency when computations involve dense matrix operations, as seen in training workloads where large tensor multiplications dominate. During training, the ability to process entire sequences in parallel, especially within the self-attention mechanism, helps maintain high arithmetic intensity.

However, this dynamic shifts during inference, which consists of two stages: **Prefill** and **Decoding**.

- **Prefill:** In this stage, the model ingests the prompt tokens in parallel, efficiently leveraging high arithmetic intensity to rapidly populate the key and value vectors **(KV) cache**. The KV cache serves as the state for the model within the attention operation, reducing redundant computation in later steps. Since no tokens are being generated at this point, computations primarily involve tensor multiplications and memory writes to populate the cache, making this phase well-suited for **high arithmetic intensity** GPU acceleration.
- **Decoding:** In the decoding phase, the model uses the previously stored KV cache to sample and generate the next token. Unlike Prefill, which is highly parallel, decoding is inherently sequential—each new token depends on the previously generated tokens, limiting parallelism and shifting the computation towards **low arithmetic intensity**. The KV cache mitigates some inefficiency by avoiding redundant recomputation, but decoding remains memory-bound, as each new token retrieval involves fetching and updating intermediate representations (such as Q, K, and V vectors). Since most inference time is spent in decoding for text generation, this stage significantly impacts scalability.

Space Complexity of the KV Cache

While decoding is predominantly memory-bound due to limited parallelism and repeated memory accesses, the KV cache offers a partial solution by reusing previously computed Key (K) and Value (V) vectors for already-processed tokens. This optimization directly addresses the growing computational burden imposed by the quadratic cost of self-attention, since caching reduces the cost per new token from quadratic **$O(n^2)$** to line**ar $O(n)$**. At the same time, the transition from high arithmetic intensity (as seen in training and prefill) to the more memory-intensive decoding stage naturally benefits from the KV cache's ability to avoid redundant computations.

However, the efficiency gains come at a steep memory cost, the total KV cache size expands with sequence length, model depth, number of attention heads, hidden dimension, and batch size. **Batch** is a collection of input sequences processed simultaneously during training or inference, enhancing efficiency by enabling parallel processing of multiple inputs. For large models and long sequences, this rapidly becomes the dominant memory bottleneck. For instance, in a 175B parameter GPT-3 model, a batch size of 64 with a 4096-token sequence can push the KV cache to require approximately 1208 GB of GPU memory, more than triple the memory needed to store the model's weights. Such massive memory requirements limit the number of concurrent requests and pose a significant scalability challenge. As explained in the earlier sections, algorithmic advances like MQA, GQA and MLA seek to strike a balance between mitigating the quadratic complexity of self-attention and the space complexity of KV cache.

Dynamic Memory Access Patterns

Unlike the predictable data access in convolutional networks, self-attention involves data-dependent memory accesses where the locations to be accessed are determined by the input sequence itself. This unpredictability hinders the effectiveness of hardware prefetchers, leading to increased cache misses and cache pollution. Also, the often scattered and non-contiguous nature of these memory accesses results in inefficient utilization of memory bandwidth, which is optimized for transferring large, contiguous blocks of data. These dynamic and irregular memory access patterns are a bottleneck, limiting the performance and scalability of Transformers, especially during inference.

Overhead of Non-Linear Operations

Non-linear operations such as Softmax, GeLU, and LayerNorm are critical to Transformer models but introduce significant computational overhead on GPUs. Unlike the highly optimizable matrix multiplications in linear layers, these element-wise operations often involve expensive reductions (e.g., for Softmax and LayerNorm) and multiple kernel launches, creating synchronization points that restrict parallelism. Their fine-grained nature disrupts the fusion of operations essential for GPU performance, resulting in excessive data movement, reduced occupancy, and underutilized compute resources, bottlenecks that worsen as LLMs scale.

Moreover, the complex, input-dependent nature of these non-linearities complicates optimization and makes them difficult to fuse with other operations. Reduction steps across sequence or embedding dimensions add synchronization overheads, and Softmax in particular (with its exponentiation, summation, and division) can dominate the execution time of the attention mechanism for longer sequences. While solutions like NVIDIA's specialized cuDNN kernels and TensorFlow's XLA compiler offer partial optimizations, the inherent complexity and data dependencies of these operations continue to limit their effectiveness, posing a persistent challenge in the efficient scaling and deployment of Transformer models.

Sparsity

Sparsity in Transformer attention poses optimization challenges due to its unstructured, dynamic nature. Unlike predictable sparsity patterns, the non-zero elements in the attention matrix shift with each input, reflecting the quadratic cost of self-attention. Growing sequence lengths exacerbate memory and compute demands for storing Key-Value (KV) pairs, even if many are ultimately unused. Standard sparse matrix libraries offer limited assistance because the dynamic sparsity prevents pre-computation, leading to runtime adaptation overhead. Indexing and load-balancing for scattered non-zero elements introduce further challenges, potentially limiting hardware acceleration. While sparse models aim to retain relevant tokens over long contexts, aggressive pruning can lead to loss of crucial information, impacting memory optimizations and model performance.

As these demands evolve, we must move beyond traditional von Neumann architectures and embrace specialized AI accelerators that seamlessly integrate software innovations. Future

systems will require carefully tailored instruction set architectures (ISAs), dataflow architectures, and efficient memory hierarchies to handle increasingly complex tasks. Effective model parallelism will hinge on interconnect fabrics designed to minimize bottlenecks, enabling large-scale data transfers without compromising performance. Realizing this vision calls for close collaboration between hardware engineers and ML researchers to drive novel computing paradigms.

Conclusion

From single-layer perceptrons to today's trillion-parameter Transformers, every leap in capability has been driven by the synergy of hardware and software, shaped by empirically observed scaling laws. These laws—governing model size, data volume, and compute—reveal power-law trends in performance gains while also exposing the irreducible entropy that limits achievable performance. Yet the emergence of unexpected abilities at certain scale thresholds suggests we're still uncovering the deeper connections between scale and intelligence.

At the heart of this evolution, the Transformer architecture's self-attention and feed-forward blocks have demonstrated remarkable efficiency—yet at extreme scales, they demand immense memory and power. The key to future advancements lies not in algorithmic improvements or hardware innovations alone, but in their thoughtful co-design. **Bits, FLOPS,** and **watts** must be treated as interdependent dimensions of a single scaling challenge, requiring optimizations that cut across all three.

In **Part 2**, we will explore these hardware challenges in-depth, tracing the path from algorithmic complexity to silicon constraints. By examining GPU architectures, interconnect bottlenecks, and the underlying CMOS physics, we will see how future progress hinges not just on building larger models, but on a thoughtful and innovative co-design of hardware and software to drive the next leap in LLM capabilities.

Part 2

Walls Of Silicon

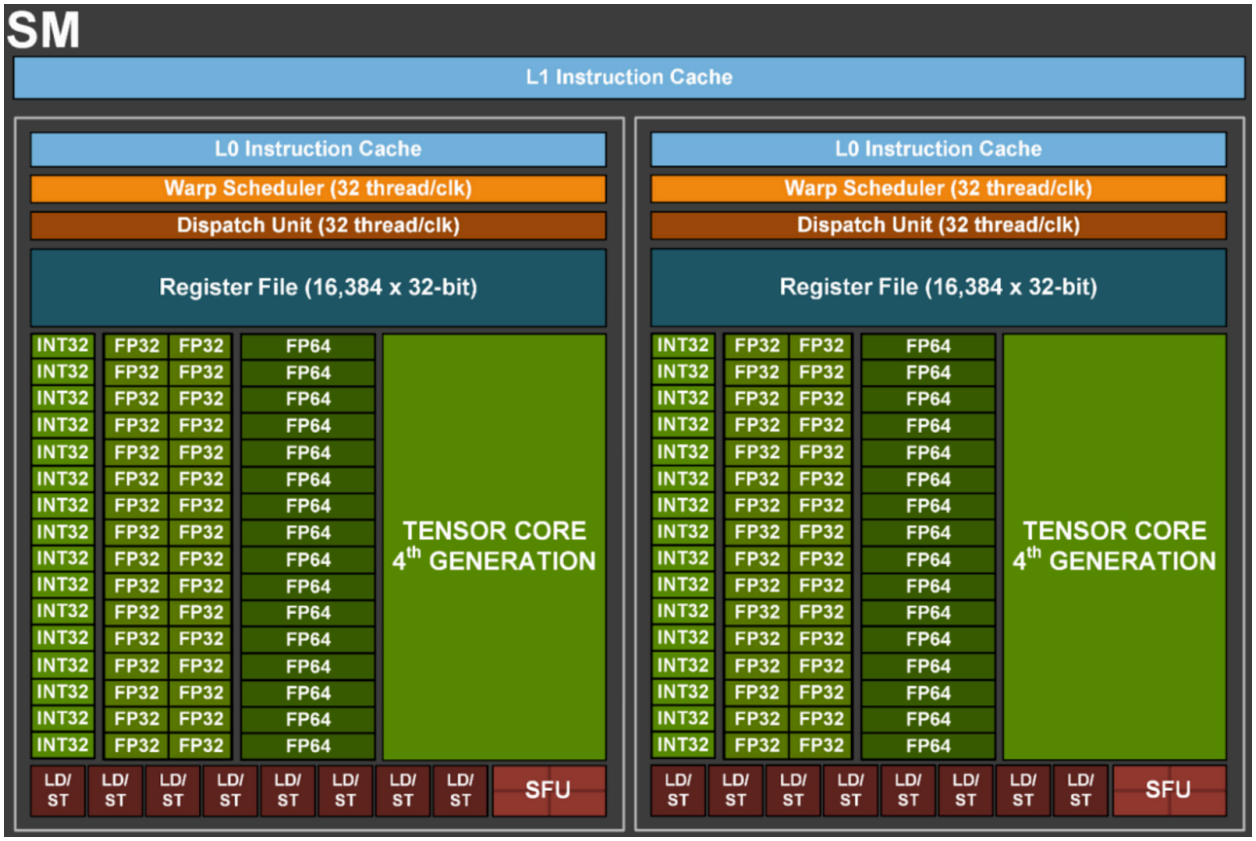
Introduction

In [Part 1], we explored the algorithmic foundations driving LLM scaling; now, we shift our focus to the silicon that underpins it. This brings us to the **"**Walls of Silicon**"**—the fundamental physical constraints that define the performance and scalability of GPU-accelerated compute: the Memory Wall, Power Wall, and Interconnect Wall. These walls arise at the intersection of Transformer algorithmic complexity and silicon physics, reshaping the calculus of bits, FLOPS, and watts that drives LLM performance.

To understand why these barriers inevitably arise, we first need to examine how GPUs operate at a fundamental level. By grasping the intricacies of GPU architecture and execution flows, we can pinpoint precisely where LLM workloads stress hardware resources and how that pressure eventually hits the three Scaling Walls. Let's begin by exploring the internal architecture of a modern NVIDIA GPU.

The GPU Architecture

NVIDIA GPUs are built around **Streaming Multiprocessors (SMs)**, each housing a diverse set of computational units and on-chip caches. These SMs are the backbone of parallel processing, with specialized units handling different aspects of the workload:



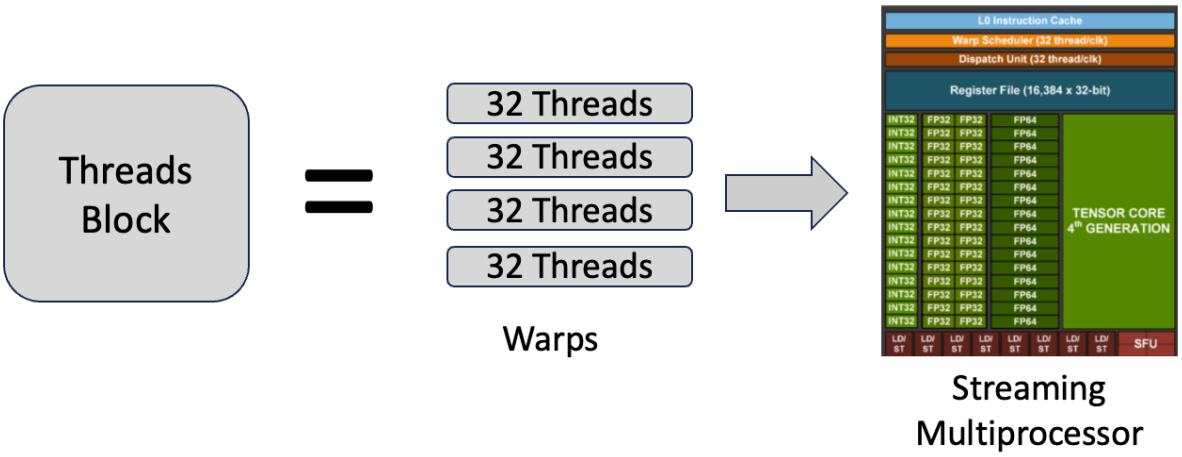
- **CUDA Cores** execute general-purpose arithmetic and memory operations, such as addition, multiplication, and data movement. These cores are highly pipelined to support a massive number of concurrent, small-scale computations.
- **Tensor Cores** specialize in matrix multiply-accumulate (MMA) operations, crucial for deep learning workloads. By fusing multiplication and addition into a single cycle, they significantly boost throughput, particularly for reduced-precision formats like FP16 and BF16.
- **Special Function Units (SFUs)** accelerate transcendental functions, including trigonometric and exponential calculations, making them vital for activation functions and other mathematical transformations in neural networks.
- **LD/ST (Load/Store) Units** manage memory operations, transferring data between the SMs and the GPU's memory hierarchy. These units fetch data from either the low-latency on-chip caches (L1, shared memory) or the high-capacity but higher-latency global memory, typically High Bandwidth Memory (HBM).

Each SM also includes **local register files** to store temporary variables for active threads, along with a **fast L1 cache and shared memory region** for efficient intra-thread block communication. All SMs share a larger **L2 cache**, which serves as an intermediary between the compute units and the global HBM. This multi-tiered memory hierarchy—spanning registers, caches, and HBM—ensures that data remains as close to the compute units as possible, reducing global memory latency and enabling massive parallel throughput.

GPU Execution Flow

A GPU executes workloads using thousands of concurrent threads, organized hierarchically into **warps, thread blocks, and grids** to maximize parallelism.

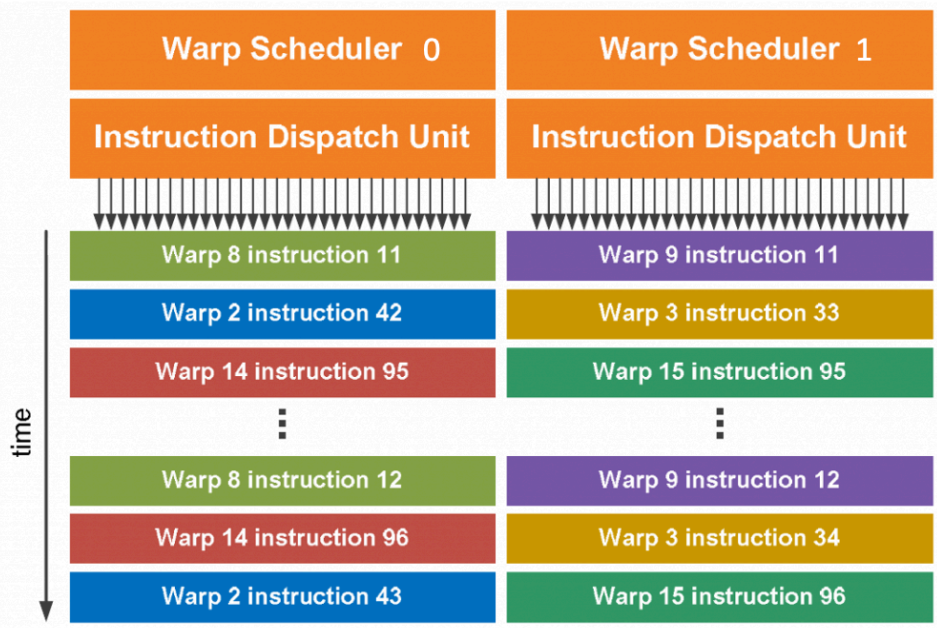
- **Warp**: The fundamental execution unit in CUDA, consisting of 32 threads. Each SM can run 2048 threads simultaneously, grouping them into multiple warps.
- **Thread Block**: A collection of threads, grouped as warps, that execute together on a single SM. All threads in a block have access to the same shared memory and can synchronize with each other.
- **Grid**: A higher-level grouping of multiple thread blocks distributed across the GPU's many SMs, working together to complete execution.



At a high level, a CUDA program launches **kernels** (parallel functions) that execute across a grid of thread blocks. All threads in a grid execute the same kernel. This follows the Single-Instruction, Multiple-Thread (**SIMT**) paradigm, where all threads within a warp execute the same instruction but operate on different data elements. SIMT extends SIMD (Flynn's Taxonomy) by reducing instruction-fetch overhead, enabling efficient execution across thousands of threads.

Warp Scheduler

To keep GPU arithmetic units busy and hide memory latency, the warp scheduler on each SM interleaves instructions from different warps each clock cycle. The scheduler can seamlessly switch between concurrent warps from any block within any kernel without incurring any overhead. If a warp stalls due to an instruction that can't execute immediately, the scheduler switches to a ready warp, one whose next instruction has all required data available. This efficient warp-switching helps mask instruction latency when enough warps are available on the SM. When multiple warps are ready, the SM uses a scheduling policy to select which warp's instruction to fetch next. This concurrency model is one of the key reasons GPUs excel at matrix and tensor workloads as thousands of lightweight threads can be orchestrated to deliver immense throughput.

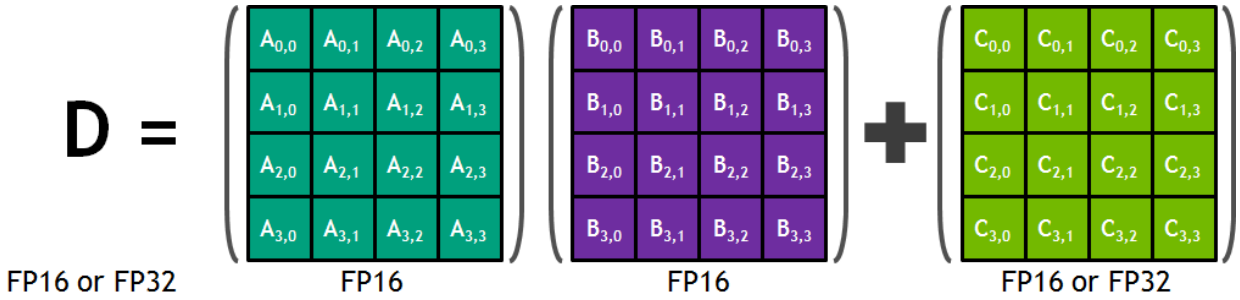


However, managing this parallelism is not without its challenges. When threads within the same warp diverge due to conditional statements, the warp must serialize different execution paths. This phenomenon, called "**warp divergence**," leads to underutilization of SM cores. Consequently, memory access patterns and branching structures become important design considerations for high-performance kernels.

Once a kernel completes, results are written back to global memory (HBM or system memory via PCIe/NVLink), where they can be accessed by subsequent kernels or by the CPU. The GPU's memory subsystem, including the register files, shared memory, L1 cache, L2 cache, and HBM, is carefully designed to keep data movement efficient, but this is exactly where some of the fundamental constraints of scaling, particularly with memory bandwidth and capacity show up.

Transformers and Tensor Cores

Transformers are ideally suited to exploit the **Tensor Core** units in NVIDIA GPUs. Transformers rely heavily on matrix multiplications to compute attention, feed-forward transformations, and gradient updates during backpropagation. Each multi-head attention layer involves large matrix multiplications of the input embedding matrix with query, key, and value weight matrices. Similarly, FFNs perform matrix multiplications interspersed with non-linear activations.



Since Tensor Cores can deliver up to an 8x speedup for half-precision FP16 matrix operations compared to standard FP32 operations, they have become vital for both training and inference at scale. However, to maximize throughput on Tensor Cores, data dimensions must align with certain constraints. Tensor Cores achieve optimal performance when operations align with specific memory boundaries: multiples of 4 for TF32, 8 for FP16, and 16 for INT8 (16-byte memory alignment). Similarly, batch dimensions also need to be properly tuned. When these conditions are met, Tensor Cores train models much faster. For instance, the backward pass can reuse the same matrix multiplication logic to compute gradients, taking advantage of reduced-precision formats that preserve enough numerical fidelity for stable training while significantly improving performance and reducing memory traffic.

Under the hood—A WMMA Kernel

To illustrate Tensor Core operations at the warp level, let's examine a simple CUDA kernel that performs a warp-level matrix multiply-accumulate (WMMA) using low level PTX ISA.
(Use this test code and instructions to compile and run this kernel on an NVIDIA A10 GPU.)

wmma_kernel_demo.ptx

```
.version 7.0
.target sm_80
.address_size 64

.visible .entry wmma_kernel(
    .param .u64 a,          // Input matrix A (fp16)
    .param .u64 b,          // Input matrix B (fp16)
    .param .u64 c,          // Input accumulator matrix C (fp32)
    .param .u64 d           // Output matrix D (fp32)
) {
    .reg .b32 %r<8>;       // General purpose registers
    .reg .b64 %rd<8>;      // Address registers
    .reg .pred %p;         // Predicate register

    // Declare matrix fragments - corrected for SM90
    .reg .f32 %C_reg<8>;     // Fragment C registers
    .reg .f32 %D_reg<8>;     // Fragment D registers
    .reg .f16x2 %A_reg<8>;   // Fragment A registers - increased to 8
    .reg .f16x2 %B_reg<8>;   // Fragment B registers - increased to 8

    // Get thread ID and check if thread is in first warp
    mov.u32 %r1, %tid.x;
    setp.lt.u32 %p, %r1, 32;
    @!%p bra EXIT;         // Only first warp performs WMMA

    // Load input pointers
    ld.param.u64 %rd1, [a];
    ld.param.u64 %rd2, [b];
    ld.param.u64 %rd3, [c];
    ld.param.u64 %rd4, [d];

    // Load matrix tiles into Tensor Core registers
    wmma.load.a.sync.aligned.m16n16k16.row.f16 {%A_reg0, %A_reg1, %A_reg2, %A_reg3, %A_reg4, %A_reg5, %A_reg6, %A_reg7}, [%rd1];
    wmma.load.b.sync.aligned.m16n16k16.col.f16 {%B_reg0, %B_reg1, %B_reg2, %B_reg3, %B_reg4, %B_reg5, %B_reg6, %B_reg7}, [%rd2];
    wmma.load.c.sync.aligned.m16n16k16.row.f32 {%C_reg0, %C_reg1, %C_reg2, %C_reg3, %C_reg4, %C_reg5, %C_reg6, %C_reg7}, [%rd3];

    // Perform matrix multiply-accumulate with correct register counts
    wmma.mma.sync.aligned.m16n16k16.row.col.f32.f32 {%D_reg0, %D_reg1, %D_reg2, %D_reg3, %D_reg4, %D_reg5, %D_reg6, %D_reg7},
        {%A_reg0, %A_reg1, %A_reg2, %A_reg3, %A_reg4, %A_reg5, %A_reg6, %A_reg7},
        {%B_reg0, %B_reg1, %B_reg2, %B_reg3, %B_reg4, %B_reg5, %B_reg6, %B_reg7},
        {%C_reg0, %C_reg1, %C_reg2, %C_reg3, %C_reg4, %C_reg5, %C_reg6, %C_reg7};

    // Store result
    wmma.store.d.sync.aligned.m16n16k16.row.f32 [%rd4], {%D_reg0, %D_reg1, %D_reg2, %D_reg3, %D_reg4, %D_reg5, %D_reg6, %D_reg7};

EXIT:
    ret;
}
```

The kernel starts with thread selection using (*mov.u32, setp.lt.u32*) to ensure single-warp execution of 32 threads. That is followed by register allocation for matrix fragments using mixed-precision formats: FP16 for input matrices **A/B**, FP32 for accumulator **C** and output **D**. The SM's register file (RF) allocation uses ***.reg*** directives to set up *f16*x*2* and *f32* register arrays, enabling efficient data access patterns for the Tensor Core operations. The core data movement occurs through ***wmma.load*** operations, transferring 16x16 matrix data from global memory to specialized registers in the required format (row/column-major) and precision, utilizing the GPU's memory hierarchy through L1/L2 caches.

The key Tensor Core computation is performed via the ***wmma.mma.sync*** instruction, which executes the matrix multiply-accumulate operation (**D = A×B + C**) on 16x16 matrices, called **tiles**. This operation leverages the hardware's ability to perform fused multiply-add (FMA) operations in a single cycle across the warp's 32 threads. The SIMD execution model ensures all threads in the warp execute the same instruction but operate on different data elements, while the warp scheduler manages instruction interleaving to hide memory latency. Finally, ***wmma.store.d.sync*** writes the computed results back to global memory through the cache hierarchy, completing the matrix multiplication sequence. All operations maintain warp-wide synchronization through ***.sync*** barriers, ensuring coordinated execution across the 32 threads of the warp.

The entire operation sequence aligns with specific memory boundaries (16-byte alignment for FP16 operations) and leverages the SM's computational units: LD/ST units handle memory transactions, Tensor Cores perform the core MMA operations, and the warp scheduler manages execution flow. This hardware-software coordination enables efficient parallel processing of matrix operations, making it particularly suitable for Transformer's GEMM-heavy (General Matrix to Matrix Multiplication) structure of attention and feed-forward layers, where aligned 16x16x16 tiles maximize Tensor Core utilization, accelerating both inference and training.

While the WMMA kernel exemplifies the remarkable engineering that enables efficient matrix operations on modern GPUs, it also provides a window into the fundamental challenges of scaling LLMs to even greater sizes. The careful orchestration required at the hardware level, from warp-level execution to memory alignment and specialized compute units, shows a delicate balance between computational power and physical constraints. As we push towards trillion-parameter LLMs, these constraints begin to manifest not as mere engineering hurdles, but as major barriers that define the boundaries of what's possible with current technology.

Walls of Silicon

These barriers emerge from the fundamental physics of semiconductor technology and the underlying computer architecture, manifesting as three interconnected walls that constrain our ability to scale LLMs:

1. **Memory Wall**
2. **Interconnect Wall**
3. **Power Wall**

Each wall represents a different facet of the same underlying challenge: the physical limits of silicon-based computing. By examining these constraints and their interplay, we can better understand both their immediate impact on model scaling and their implications for the future of AI hardware architecture. These walls arise from inherent physical and engineering constraints. They impose significant limits on how many GPUs can be scaled out, how large on-chip memory can be, how much power a data center can provide, and how efficiently data can be moved and processed across devices.

In the following sections, we will go deeper into each of these barriers, explore their root causes, and discuss strategies used to address or mitigate them. Understanding these challenges is essential for advancing research and deployment of LLMs, where increasing model size no longer guarantees proportional performance improvements.

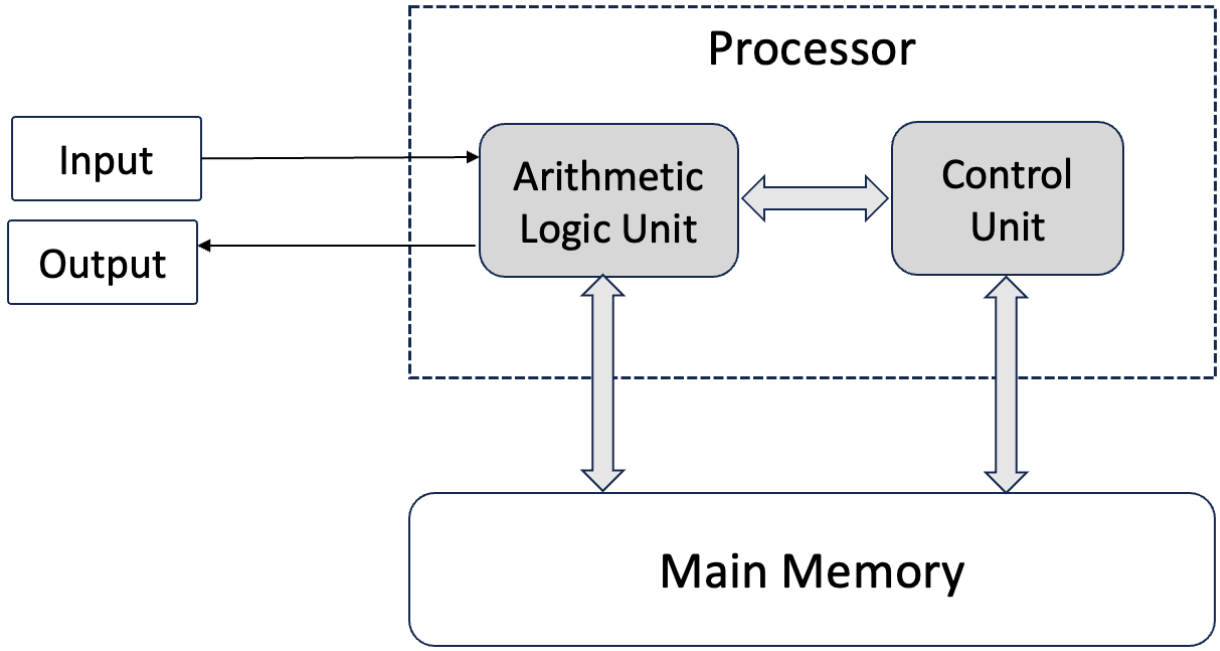
The Memory Wall

Understanding the intricacies of GPU architecture, its reliance on parallel SMs, layered memory hierarchies, and high-bandwidth operations, highlights how performance hinges on efficient data movement. GPUs thrive on feeding their computational cores with a steady stream of data from memory, yet as LLMs scale to trillions of parameters, this demand outpaces the capacity of GPU's memory systems. This challenge manifests as the "Memory Wall"—a widening gap between computational capability and memory performance. This fundamental problem presents itself in three critical ways. First, the movement of data between memory elements and processing units introduces significant latency overhead. Second, data movement through memory hierarchies is severely constrained by bandwidth limitations. Finally, and perhaps most critically, the energy cost of data movement is immense. In fact, transferring data between computing units and off-chip memory can consume nearly 100 to 200 times more power than the floating-point computations themselves. We will explore this energy cost in greater detail in the **Power Wall** section.

At the root of this challenge lies the **Von Neumann architecture**, the foundational model upon which modern computing systems are built. By examining this architecture, we can pinpoint the structural bottlenecks in data movement and understand the limitations GPUs encounter in meeting the escalating demands of LLMs.

The Von Neumann Architecture

Computing systems, such as GPUs, are fundamentally influenced by the Von Neumann architecture, which logically separates compute and memory elements. In this model, a processor— equipped with high-speed registers, arithmetic logic units (ALUs), and control units—fetches instructions and data from a main memory, typically implemented using DRAM. This memory stores both program code and the data the program operates on.



These components work together to enable efficient execution of instructions. At the core, the **processor** consists of arithmetic logic units (ALUs), registers, and control logic, handling computations and instruction execution. Supporting this, **main memory (**DRAM**)** serves as high-capacity storage for both instructions and data, though it operates at a slower speed compared to the processor. To facilitate communication, **buses** act as shared pathways, transmitting data and control signals between the processor, memory, and I/O devices. **I/O devices** serve as the interface between the system and external data sources or end users, ensuring interaction with the outside world. This entire system operates in a continuous **fetch-decode-execute cycle**, where the processor retrieves an instruction from memory, deciphers it, and executes it before moving to the next.

While GPUs adhere to these core Von Neumann principles, they optimize for parallel computation through specialized hardware. Unlike CPUs, which are primarily optimized for low-latency execution on fewer threads, GPUs contain multiple SMs, which collectively house thousands of ALUs capable of simultaneous operations. They implement SIMT execution models, enabling massive parallelism. This parallel execution is further supported by a hierarchical memory structure consisting of HBM, shared memory, caches, and registers. This intricate memory hierarchy is essential for feeding the thousands of processing cores and operates synergistically with the SIMT execution model and its underlying warp scheduling mechanisms to maximize bandwidth, manage data efficiently, and crucially, hiding the latency inherent in accessing main memory. However, this technique of “hiding the latency” itself becomes a problem for workloads with low arithmetic intensity, as we will see in subsequent sections.

While fetch-decode-execute cycle forms the foundation of computation, a persistent challenge arises due to the growing performance gap between processor clock rates and memory speeds, often resulting in stalls when the processor waits for data retrieval.

The Von Neumann Bottleneck

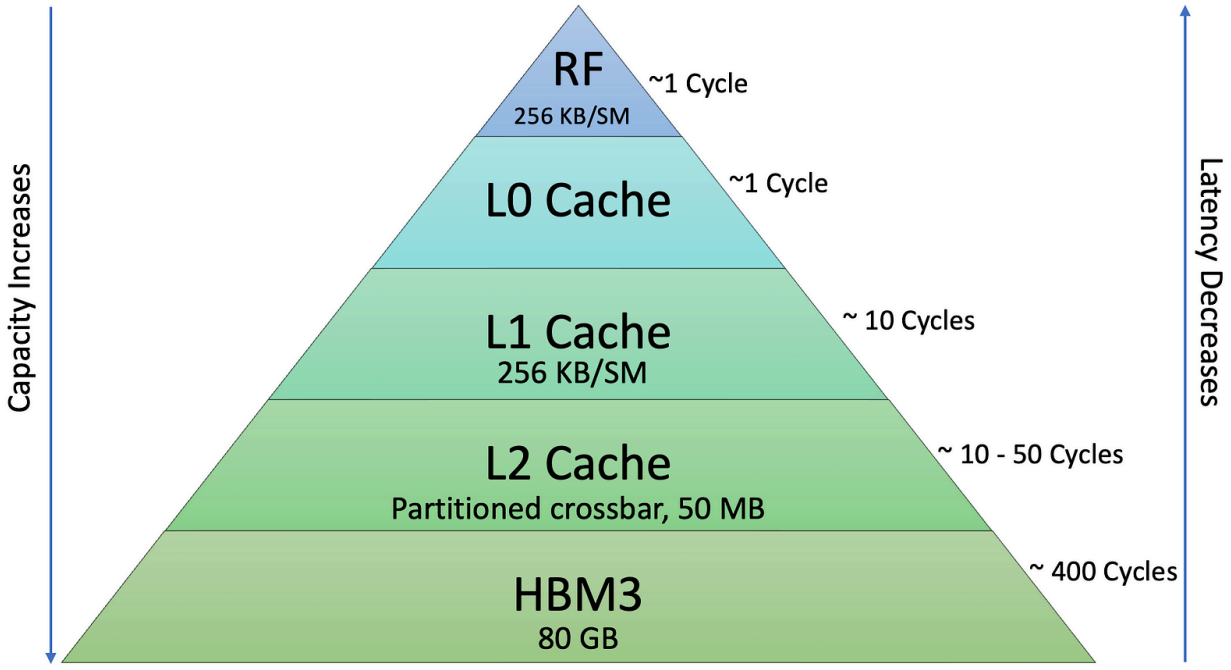
This performance mismatch between on-chip processor elements and comparatively slow off-chip main memory leads to the Von Neumann Bottleneck. While processor speeds and transistor densities have increased exponentially over the decades, improvements in main memory (DRAM) latency and bandwidth have been relatively modest. This disparity means that even highly efficient GPUs often spend a significant portion of their time waiting idly for data to arrive from memory, wasting cycles and reducing overall throughput. This “Von Neumann Bottleneck” is particularly pronounced for LLMs, which require rapid, large-scale data movement to and from memory during both training and inference.

The bottleneck arises from the trade-off between data transfer rates and energy efficiency—the interplay between **bits** per second (bps) and **watts** per bit. From an electrical engineering standpoint, data movement is constrained by signal propagation physics, where energy use scales with frequency and distance due to capacitance, resistance, impedance, and inductance. As data moves across longer paths, especially to off-chip DRAM , it must traverse chip boundaries, package pins, and board traces, increasing energy costs and exacerbating both latency and thermal constraints. Accelerators like GPUs demand ever-higher memory bandwidth but must operate within strict power envelopes, creating a fundamental bottleneck: increasing data throughput drives up energy consumption. Simply put, the faster we move **bits**, the more **watts** we burn.

Memory Hierarchies

To mitigate these effects, GPUs employ a multi-level memory hierarchy designed to minimize the performance penalty of accessing off-chip DRAM. This hierarchy typically consists of:

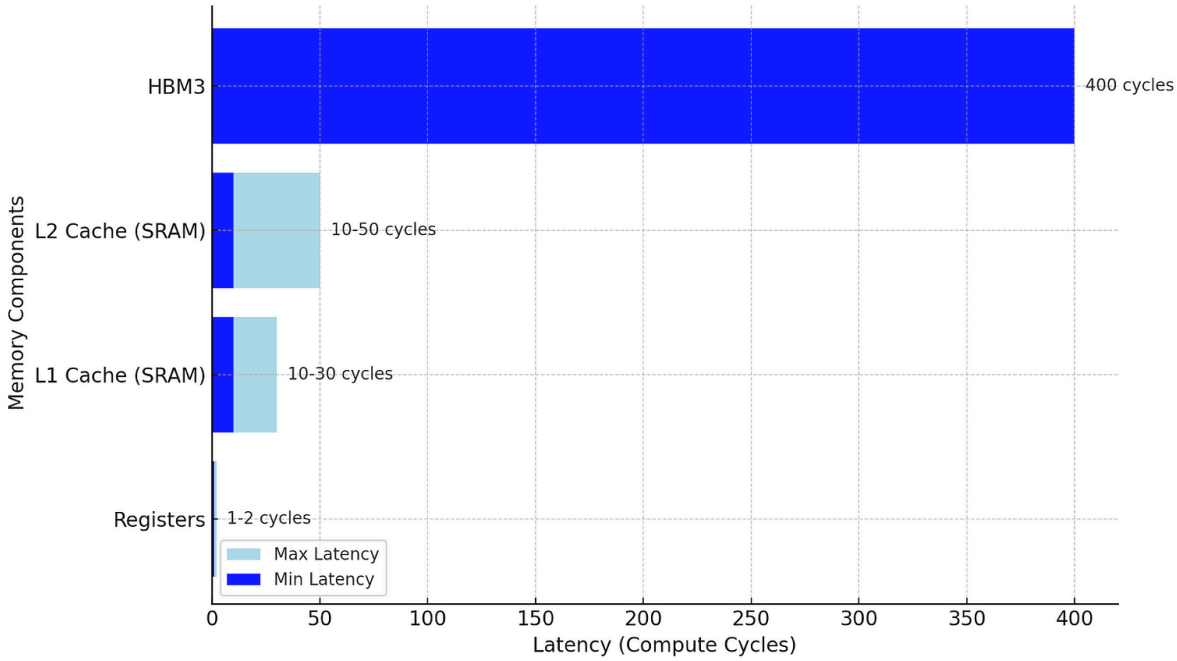
- **L0 and L1 caches**—Small, fast SRAM caches located near compute units, supplying operands at near-processor speeds (~1–10 cycles).
- **L2 cache**—A larger, on-chip SRAM cache that buffers intermediate results shared among multiple execution units.
- **High Bandwidth Memory (HBM)**—Off-chip DRAM stacks serving as the primary high-capacity memory for the GPU.



H100 Memory Hierarchy

To illustrate this hierarchy in practice, let's examine the NVIDIA H100:

- **256 KB Register File (RF) per SM**—The fastest on-chip SRAM for immediate computations.
- **L0 Instruction Cache**—Private to SM quadrants for quick instruction fetch.
- **256KB L1 Cache per Streaming Multiprocessor (SM)**—Provides low-latency memory for frequently accessed data.
- **50MB L2 Cache**—A large, partitioned-crossbar cache that minimizes HBM3 accesses by sharing data across SMs.
- **HBM3 Memory (up to 80GB)**—The primary high-capacity VRAM, responsible for handling large datasets efficiently.



This hierarchical structure improves enhances memory efficiency, ensuring high-speed access to critical data while reducing reliance on slower, off-chip memory.

Revisiting the PTX Kernel Example

Now let's deep dive into how this GPU hierarchical memory structures help mitigate the Von Neumann Bottleneck by revisiting the WMMA kernel example from the previous section. The kernel begins by loading pointers for matrices A, B, C, and D into registers using ***ld.param.u64***. If bandwidth falls short during these loads, the pipeline stalls, and the Tensor Cores sit idle. Next, the kernel performs ***wmma.mma.sync***, a fused multiply-accumulate operation that leverages specialized Tensor Cores for parallel half-precision inputs and single-precision accumulators. While these cores can significantly increase throughput, their potential goes underutilized whenever data fails to arrive on time, exacerbating the Von Neumann Bottleneck.

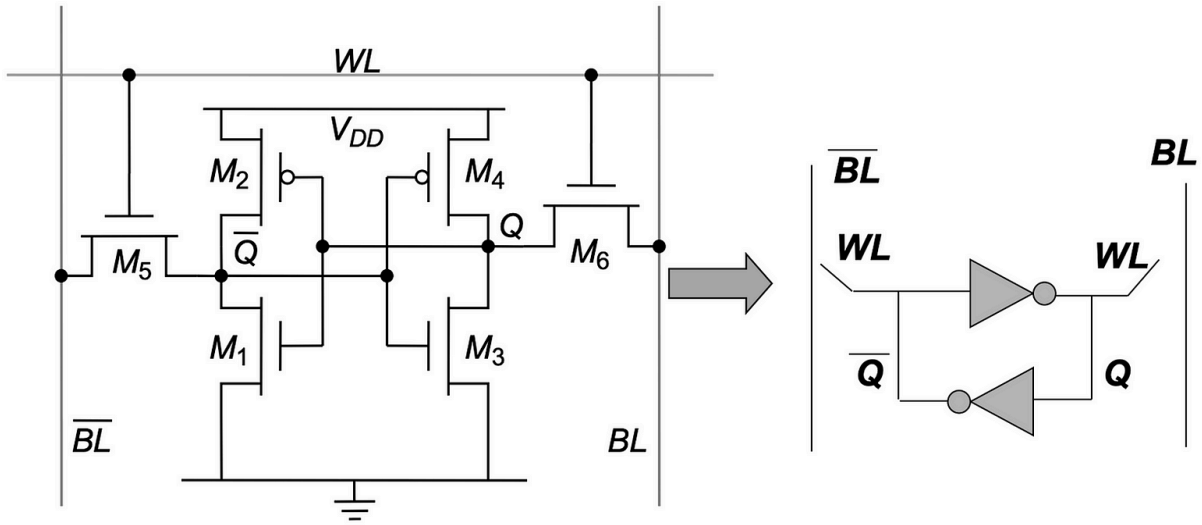
GPU's mitigate this bottleneck by layering multiple tiers of memory. At the smallest scale, registers (*%A_reg to %D_reg*) provide near-instant access to critical data operands. Separately, an **L0** instruction cache provides rapid access to fetched instructions, ensuring a steady instruction stream for the SM. The **L1** cache brings frequently used operands closer to each SM, while a sizable **L2** cache, spanning tens of megabytes, reduces the need for repeated fetches from **HBM**. Although HBM3 provides large storage capacity, its higher latency still poses a challenge. To address this, instructions like ***wmma.load*** are designed to leverage this caching hierarchy to ensure that Tensor Cores remain consistently fed, minimizing idle cycles. This interplay of memory hierarchy and advanced compute capabilities helps manage the Von Neumann Bottleneck, particularly for matrix-intensive operations.

SRAM vs. DRAM: Latency vs. Capacity

At the core of modern memory hierarchies are Static Random Access Memory (**SRAM**) and Dynamic Random Access Memory (**DRAM**), two fundamental memory technologies that each balance speed and capacity in different ways. SRAM is optimized for extremely low latency and serves as the foundation of on-chip processor caches, such as L1 and L2, which sit closest to compute units. DRAM, in contrast, operates off-chip and offers higher capacity but at slower access speeds, thus forming a hierarchical structure in which each memory level is carefully positioned to balance latency, bandwidth, and storage capacity. To understand why SRAM achieves such low latency, it is important to first examine its architecture in detail.

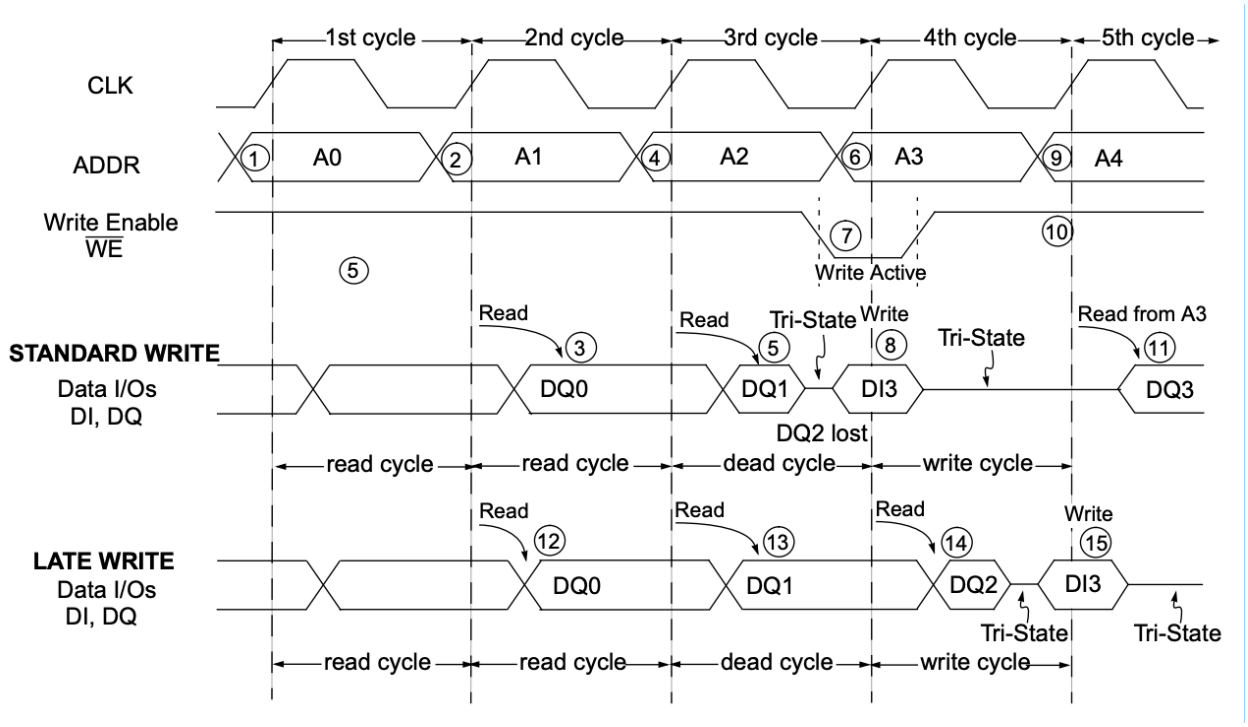
SRAM Architecture

SRAM uses a 6-transistor (6T) design, in which two cross-coupled inverters form a stable latch to store a single **bit** (0 or 1), and two additional transistors act as access gates for read and write operations. The cross-coupled inverters ensure bistability, meaning that once a bit is written, it remains in that state until explicitly changed. This sharply contrasts with DRAM, which stores bits in capacitors that discharge over milliseconds, causing state decay and requiring periodic refreshes to maintain data integrity. The access transistors, controlled by the word line (**WL**), connect the storage nodes to the bit-lines (**BL** and $\overline{\mathbf{BL}}$), facilitating faster read and write operations.



During a write operation, the bit-lines are driven to complementary values (BL = VDD, $\overline{BL}$ = 0 for writing a '**binary 1**') before enabling the word line. This forces the corresponding values into the storage nodes, overriding the previous state. For a read operation, both bit-lines are precharged to VDD, and then the word line is activated, allowing the stored value to slightly perturb one of the bit-lines, which is then detected by a sense amplifier. Sense amplifiers also improve SRAM power efficiency by reducing bitline voltage swings, facilitating lower supply voltages, accelerating read operations, and mitigating leakage current.

Because it does not rely on capacitors, SRAM does not require the periodic refresh cycles that DRAM or HBM necessitate, which in turn enables faster access times and lowers idle power consumption. Its latch-based mechanism removes the delay associated with the charge-based read and write operations found in DRAM, allowing data retrieval in as few as 1–10 clock cycles. The near-instantaneous transistor switching enabled by this architecture makes SRAM ideal for L1 and L2 caches, where supplying data at near-core speeds is essential. At a fundamental level, SRAM's low latency is achieved through its latch-based architecture and rapid transistor switching, and its operation is managed by precise timing control driven by external clock signals.



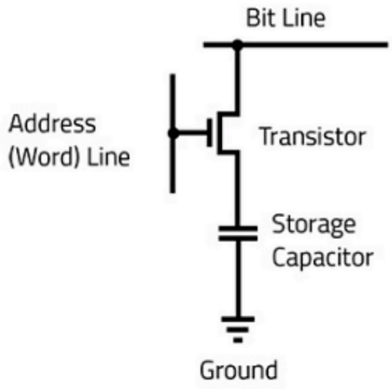
These signals include the clock (**CLK**) for synchronization, the address (**ADDR**) for selecting specific storage locations, and signals such as Chip Select (**CS**), Write Enable (**WE**), and Output Enable (**OE**) that determine when the SRAM is active, when data can be written, and when data can be read. Data flows through dedicated data input/output (**DQ**) pins during these read and write operations. When performing a read, the address is registered on the rising edge of the clock, and valid data arrives at the output pins after the specified access time, which can sometimes be as short as a single processor cycle. Similarly, write operations follow a clock-synchronized sequence, although different implementations optimize how dead cycles between read and write operations are handled.

Despite its speed, SRAM's 6-transistor design reduces storage density and increases cost per bit, which limits on-chip cache sizes to megabytes. Larger capacity memory resources thus turn to DRAM, such as off-chip HBM, which offers the density required by LLMs at the expense of higher latency. Even though small SRAM caches can mitigate some of DRAM's latency, understanding the intricacies of DRAM architecture is crucial for recognizing the remaining bottlenecks.

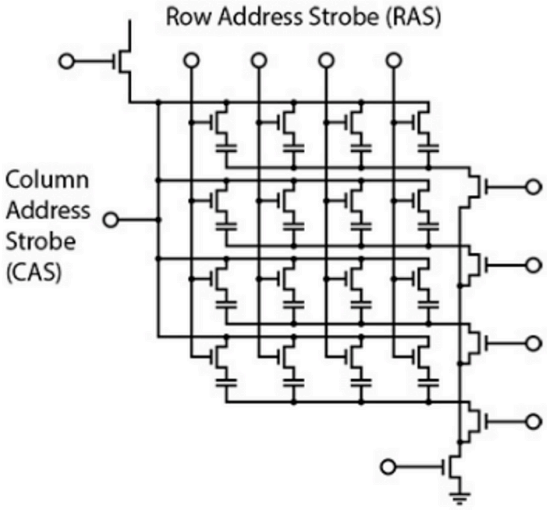
DRAM Architecture

Dynamic Random-Access Memory (DRAM) is built on a 1T1C (1 Transistor, 1 Capacitor) memory cell architecture. Each cell consists of a capacitor, which stores charge representing a **bit** (0 or 1), and a transistor that acts as an access gate. These cells are organized in a grid-like array of **wordlines** (rows) and **bitlines** (columns) to enable read and write operations through the activation of specific intersections.

When a wordline is activated, the transistor connects the capacitor to the bitline, allowing data transfer. A charged capacitor represents **binary 1,** while a discharged capacitor signifies **binary 0**. However, reading data depletes the capacitor's charge due to charge sharing with the bitline, requiring immediate restoration via a **sense amplifier** to maintain data integrity. Furthermore, capacitors leak charge over time, necessitating periodic **refresh cycles** to prevent data loss.



Single Memory Cell



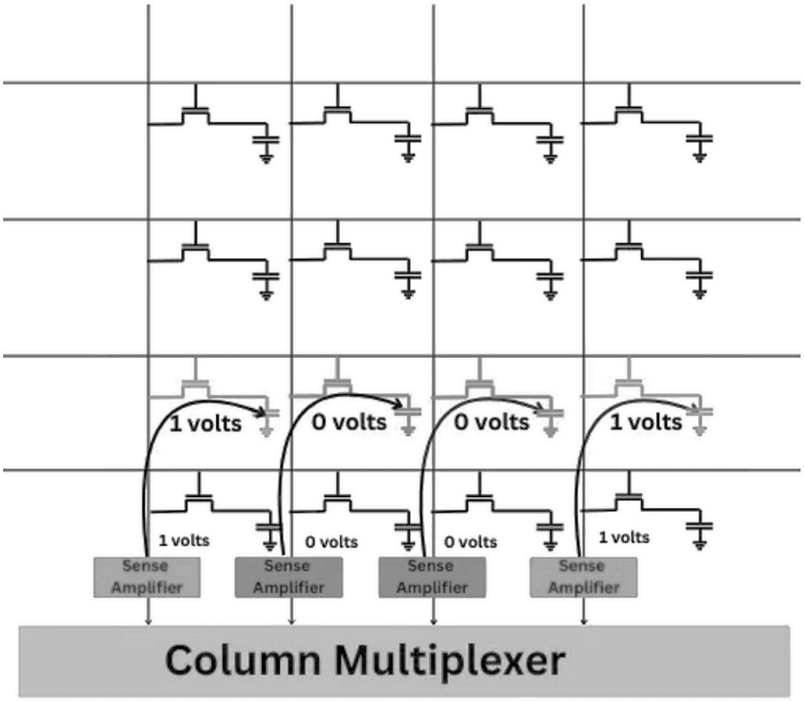
Memory Cell Array

To achieve high-density storage within a compact form factor, DRAM capacitors employ extreme aspect ratios **(~100:1)**, increasing surface area to maintain capacitance (~6 fF) as DRAM cells shrink. This structural optimization enables DRAM to balance **s**peed, cost, and density, making it faster than NAND but more cost-effective than SRAM. The Row Address Strobe (**RAS**) and Column Address Strobe (**CAS**) signals facilitate efficient memory access, though the need for periodic refreshing introduces latency. Despite this limitation, DRAM remains the dominant choice for main memory in GPUs due to its ability to deliver an optimal mix of performance and scalability.

DRAM Refresh

The inherent physics of DRAM's capacitor-based cells require constant attention to prevent data loss. It stems from the inherent leakage of charge in capacitor-based memory cells. Without periodic reinforcement, the charge in each cell would decay below detectable thresholds within

64 ms at room temperature, leading to data loss. At higher temperatures (above 85°C), leakage accelerates, requiring refresh intervals as short as 32 ms to maintain data reliability.



The refresh process involves activating a specific row via the **RAS signal**, precharging bitlines to a mid-level voltage (e.g., 0.5V). The capacitors' charge interacts with the bitlines, and the **sense amplifier** detects voltage changes to determine if the cell stores a 0 or 1. The sense amplifier then reinforces these values, restoring the capacitors to their original states.

DRAM Density

As DRAM density increases, two key factors drive up power demands. First, smaller capacitors exhibit faster charge leakage, requiring more frequent refresh intervals. Second, with increased density, the total time spent on refresh operations increases, due to more rows that must be refreshed, leading to longer periods where memory banks are active and consuming power.

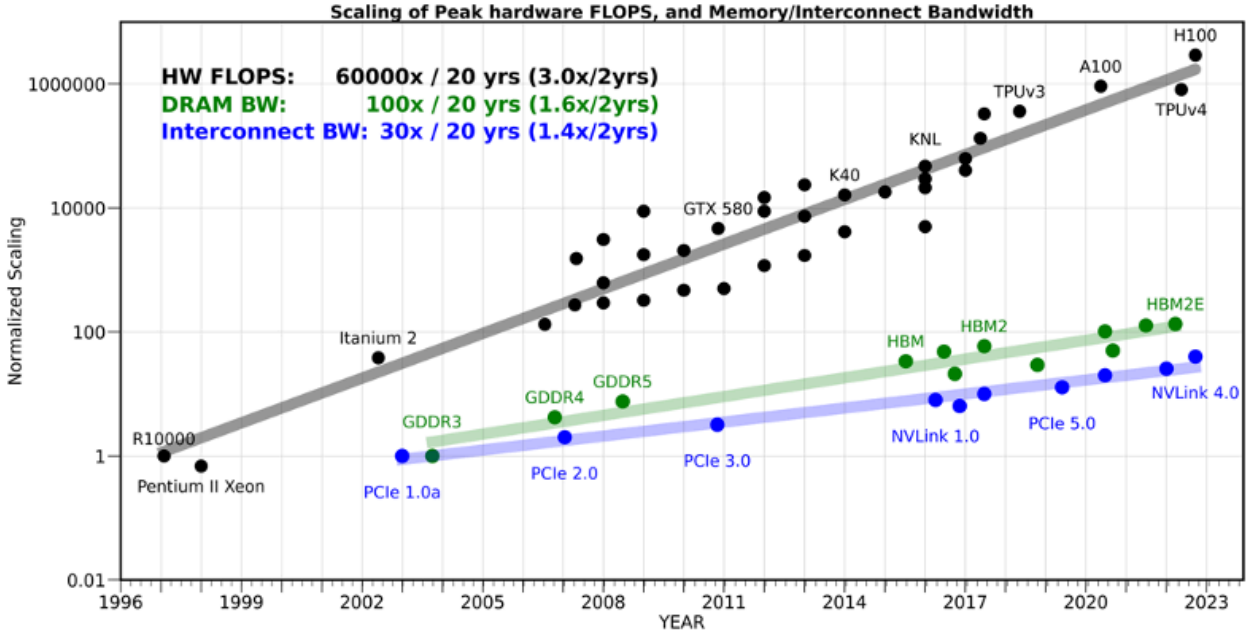
Moreover, higher cell density amplifies electrical interference (**crosstalk**) and susceptibility to power supply noise, increasing the risk of data corruption. During refresh operations, DRAM temporarily blocks access to memory banks for the duration of the **tRFC** (Refresh Cycle Time). This forces the memory controller to queue incoming read and write requests, increasing latency and potentially reducing system performance. The impact is particularly pronounced in high-density DRAM modules, where frequent refresh cycles substantially degrade bandwidth and response times.

To mitigate these performance bottlenecks, modern memory controllers utilize strategies such as distributed refresh and intelligent scheduling, minimizing the impact of refresh cycles. However, despite these optimizations, refresh cycles remain a contributor to power

consumption, especially in GPUs. Additionally, memory access operations, including reads, also contribute to dynamic power consumption within the DRAM, adding to the overall power budget. Consequently, a portion of the GPU's power budget is allocated to managing DRAM state through refresh operations, rather than solely for computations. This issue is exacerbated in memory-intensive workloads like LLMs, where frequent refresh cycles interfere with low-latency memory access, ultimately limiting efficiency.

Implications on DRAM scaling

The physical limitations of shrinking DRAM cells, such as increased refresh rates and higher power consumption, have far-reaching consequences for DRAM scaling. Although computational capabilities have experienced exponential growth, advancements in DRAM bandwidth have not kept pace.

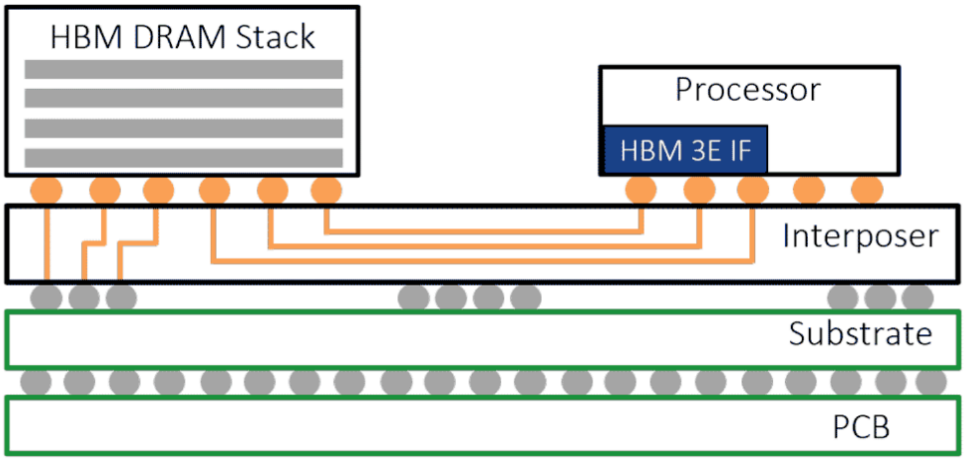


This disparity is highlighted by the fact that even if compute logic can achieve a staggering **60,000-fold** improvement in performance, memory bandwidth may have only increased by a factor of **100** over the same time frame. Furthermore, the energy cost of moving data significantly outweighs that of performing computations. Accessing data from off-chip DRAM can consume 100 to 200 times more energy compared to executing a single floating-point operation.

High Bandwidth Memory (HBM)

High Bandwidth Memory (HBM) addresses this growing disparity between compute performance and memory bandwidth. HBM's advanced 3D die stacking and ultra-wide data buses provide a significant upgrade over traditional DRAM, enabling faster data transfer and mitigating memory bottlenecks that impede system performance. HBM can be likened to a

high-speed, multi-lane superhighway for data, in contrast to the congested, narrow lanes of traditional DRAM.



By vertically stacking multiple DRAM layers (8 or more) and interconnecting them using Through-Silicon Vias (TSVs), HBM achieves a 1024-bit wide data bus per stack, which is 16 times wider than DDR5 DRAM's 64-bit bus, enabling up to 1.2 TB/s bandwidth (HBM3E) per stack. This increased bus width substantially accelerates data transfer between the processor and memory. Moreover, HBM is co-packaged with the GPU on an interposer, significantly reducing the physical distance between memory and processor to mere millimeters, compared to the few centimeter gap in DRAM (DDR in DIMM) memory. This close proximity results in lower latency and reduced energy consumption per bit transferred (**pJ/bit**), making HBM more power-efficient despite its high data throughput. By offering a high-bandwidth, low-latency, and energy-efficient memory solution, HBM prioritizes bandwidth over capacity, thereby helping to alleviate the Von Neumann bottleneck.

Transformers and the Memory Wall

After this deep technical exploration of the Memory Wall, one cannot but wonder and ask this crucial question: ***Are current GPU architectures inherently suited for the evolving computational complexity of scaling Transformer-based LLMs?***

The answer lies in understanding the interplay of Transformer's algorithmic complexity and the underlying GPU architecture. GPUs historically were designed for graphics workloads, which are highly parallel and uniform in their arithmetic intensity (a measure of the amount of computation performed per unit of data accessed from memory). Tasks like pixel processing and vertex shading involve performing the same operations on large datasets in parallel, making them both arithmetic-heavy and predictable. In contrast, Transformer-based LLMs exhibit a very different computational profile. While they rely on dense operations such as matrix multiplications, they also incorporate sparse computations like sparse attention and MoE, along with numerous nonlinear kernels such as Softmax, GELU, and Layer Normalization. These operations create data-dependent execution paths and irregular memory access patterns, leading to suboptimal scaling efficiency and underutilization of available computational throughput (FLOPS).

Despite innovations such as HBM and hierarchical caches, GPUs are still constrained by the von Neumann architecture's fundamental separation of compute and memory units. This physical divide forces constant data movement between memory and processing elements, where each transfer consumes significantly more energy (**pJ/bit**) than the actual computation (FLOPS). Transformers exacerbate this issue through dynamic, data-dependent attention mechanisms that produce irregular memory access patterns. As discussed in Part 1, inference, in particular the decoding phase, further amplifies these problems. Unlike the parallel training process, decoding is inherently sequential and memory-bound, making it less efficient. Self-attention exhibits quadratic scaling with sequence length at $O(n^2 \cdot d)$, thus rapidly increasing memory and compute requirements. Although optimizations such as the KV cache can reduce per-token complexity from quadratic $O(n^2)$ to linear $O(n)$, they introduce substantial memory overhead for long sequences and CoT based reasoning models, making large-scale inference increasingly challenging.

Frequent cache misses force GPUs to access off-chip memory, but it is the higher off-chip latency that amplifies the problem, causing excessive stalls as data is repeatedly fetched. This bottleneck results in significant energy costs, as data traverses chip boundaries, interposers, and circuit board traces. The repeated transfers not only degrade cache utilization and overwhelm hardware prefetching mechanisms but also trigger even more off-chip operations, creating a self-perpetuating cycle of inefficiency. While methods such as MQA, GQA, and MLA can mitigate some of these inefficiencies, the memory burden posed by Transformers remains a significant challenge to both scalability and performance.

With these challenges in mind, the next key factor to examine is **arithmetic intensity**—and how it contributes to underutilizing the GPU's compute cores in Transformer workloads.

Arithmetic Intensity and Underutilization of Compute Cores

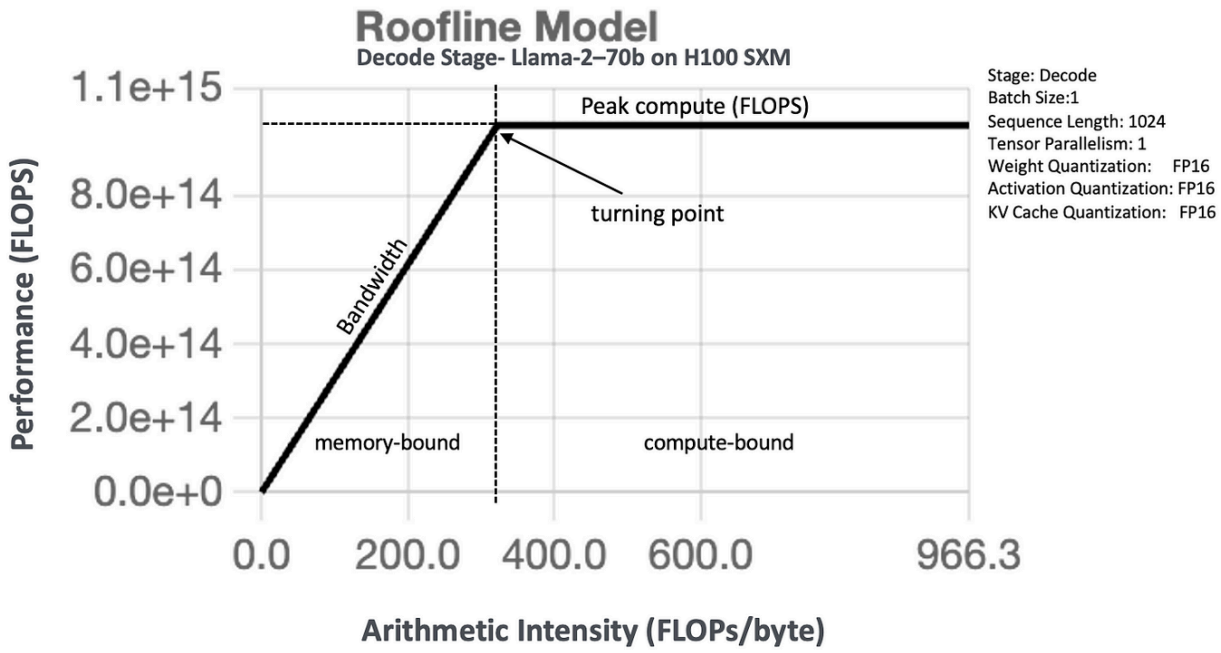
Arithmetic intensity, measured in (OPs/byte) or (FLOPs/byte), represents the ratio of floating-point computations to memory operations. It is quite useful in maximizing GPU performance for Transformer models. Models with high arithmetic intensity perform more computation per unit of data transfer, allowing better utilization of compute resources. However, LLMs often experience phases of low arithmetic intensity, especially during token-by-token decoding. They alternate between compute-heavy operations, such as matrix multiplications for FFNs or attention mechanisms, and memory-intensive operations, such as retrieving token embeddings or reading and writing KV caches. When memory bandwidth fails to keep pace with data demands, memory transfers become a bottleneck, leading to underutilization of the GPU's available FLOPS. To analyze and mitigate this issue, the Berkeley Roofline Model offers a valuable framework for understanding performance limitations.

Berkeley Roofline Model

This model visually maps achievable performance (FLOPS) against arithmetic intensity, where FLOPs per byte fetched from the memory hierarchy are plotted along the x-axis (FLOPs/byte). It helps determine whether a workload is constrained by memory bandwidth or compute capability. The model creates a characteristic "roof" shape. The slanted line on the left represents the memory bandwidth limit or the **bandwidth roof**. This limit shows that at low arithmetic intensity, performance is bottlenecked by how quickly data can be fetched from memory. The horizontal line at the top signifies the **peak compute roof**, representing the theoretical maximum computational performance of the given hardware. The actual roofline, and thus the realistic achievable performance, is the lower bound of these two limits at any given arithmetic intensity, visually forming the "roof".

> Transformer-based LLMs often fall within the memory-bound region, particularly during token-by-token inference, meaning their performance is constrained by the speed at which data moves from memory rather than by raw compute power.

By analyzing where different layers or phases of a model lie on this chart, researchers and engineers can determine whether the workload is memory-bound or compute-bound and apply optimizations accordingly. This understanding guides optimization efforts. For layers identified as memory-bound, prioritizing techniques to reduce memory transfers is key. Examples include model quantization or data layout optimizations to improve cache usage. For compute-bound layers, improving mathematical efficiency becomes more critical, which might involve using more efficient algorithms or optimized kernel implementations.



Practical Example: Llama-2–70b on H100 SXM5

This roofline chart for Llama-2–70B's decode stage on an H100 SXM shows a workload that is predominantly memory-bound, as indicated by the performance curve adhering to the "bandwidth slope" rather than approaching the horizontal "peak compute" line. In token-by-token inference, large model weights must be repeatedly fetched from memory with minimal reuse, creating a low arithmetic intensity that limits achievable FLOPS. Although the theoretical peak for FP16 Tensor Core operations on the H100 SXM (with sparsity) can reach nearly 1.979×10^{15} *FLOPS*, single-batch decode sustains around 1.1×10^{15} *FLOPS*, corresponding to ~55% of that nominal peak. This figure should not be viewed as unusually low, because real-world LLM workloads rarely approach 100% of a GPU's maximum advertised performance. This significant gap between the achievable performance and the theoretical maximum is a direct consequence of the **memory wall**. As the decode stage spends a disproportionate amount of time waiting for data to be fetched rather than performing computations, the immense computational horsepower of the H100 SXM remains partially untapped. Consequently, the roofline visualization not only confirms the memory-bound nature of the Llama-2–70b decode stage but also highlights the magnitude of the memory bottleneck in limiting the utilization of the H100 SXM peak compute potential.

Memory access patterns are only one part of the challenge. The computational patterns within transformers introduce another layer of complexity, because they heavily rely on specialized mathematical operations that, while still leveraging linear algebra, push the demands beyond the core linear algebra workloads GPUs were originally designed to accelerate. This places additional strain on traditional GPU architectures, necessitating new hardware strategies that better align with the unique demands of transformer-based LLMs.

Non-Linear and Irregular Operations

Transformers rely on nonlinear operations such as softmax, GeLU, and layer normalization, which are crucial for stabilizing training and learning expressive representations. Unlike large matrix multiplications, which GPUs handle efficiently due to their inherent parallelism and optimized data access patterns, these nonlinear functions introduce unique optimization challenges. They often involve branching computations and operations like conditional checks, exponentials, and normalizations. These introduce more complex control flow within threads and can lead to reduced peak efficiency on SIMT architectures compared to the maximum throughput achievable with perfectly regular and predictable operations like matrix multiplications.

Softmax Example

Softmax computation involves calculating exponentials for every element in an input vector, normalizing probabilities by summing these values, and addressing floating-point overflow or underflow through techniques like Max-Shift or Log-Sum-Exp (LSE). Since the exponential function amplifies differences in input values, threads within a warp may encounter vastly different numerical ranges, leading to potential overflow, underflow, or denormal numbers. Managing these exceptional cases introduces warp divergence, where some threads proceed with normal execution while others handle slower exception cases or denormal operations. Unlike matrix multiplication, where threads operate in uniform lockstep, this divergence degrades GPU performance.

$$s\left(x_i\right) = \frac{e^{x_i}}{\sum_{j=1}^{n} e^{x_j}}$$

Additionally, computing the sum Σ exp(…) requires communication across threads, typically handled by parallel reduction algorithms. These reductions involve operations such as intra-warp shuffles and inter-warp communication through shared or global memory, introducing latency and synchronization points where threads may idle, further exacerbating performance bottlenecks. Furthermore, softmax computation exhibits relatively low arithmetic intensity compared to operations like large matrix multiplications (GEMMs). It requires reading the input vector, accessing intermediate values such as the maximum and summed exponentials, and writing the output vector, leading to frequent memory accesses. Consequently, the ratio of computation (exponentials, summations, and divisions) to memory operations is significantly lower than in GEMMs, often causing softmax performance to be constrained by memory bandwidth rather than compute capability, resulting in compute units frequently sitting idle while awaiting data. Optimizing softmax thus demands careful attention to minimizing thread divergence and memory latency, posing a distinctly different set of challenges compared to optimizing large-scale matrix multiplication kernels.

Divergence in Nonlinear Kernels

As explained in the GPU architecture section, thousands of threads are organized into warps, thread blocks, and grids, with each warp executing the same instruction in lockstep. This structure excels in dense matrix multiplications because identical operations can be repeatedly applied to contiguous chunks of data, allowing the warp scheduler to hide memory latencies by interleaving instructions from different warps and smoothly saturating compute resources.

Nonlinear kernels, including FFN activation functions (ReLU, GeLU, sigmoid, tanh), normalization layers (RMSNorm, LayerNorm), introduce greater control flow and memory irregularities. When some threads within the same warp encounter exponentials or conditional logic while others do not, the warp must serialize the divergent paths, leading to underutilization of the SMs. Essentially, the warp executes each branch of the divergent code path sequentially, forcing threads that do not take a particular branch to remain idle while others execute it. This serialization disrupts what could ideally be parallel execution.

Memory Access Patterns and Reductions

Additionally, these kernels often exhibit uncoalesced or strided memory accesses when computing normalization statistics or partial sums in activation layers, further increasing pressure on the memory subsystem and caches. For instance, LayerNorm's mean and variance calculations involve reductions across the feature dimension. However, the combination of parallel processing and reduction operations, particularly given common data layouts in memory, can result in misaligned memory accesses, increasing memory pressure and reducing cache efficiency due to inefficient coalescing.

Similarly, partial sums or reductions in activation layers may not follow perfectly contiguous memory access patterns, unlike dense matrix multiplications, particularly if the data layout is not optimized for such operations. Uncoalesced or strided accesses often result in cache misses, as memory accesses fail to align with cache lines, lowering cache hit rates and necessitating more frequent accesses to slower off-chip DRAM. Although GPUs offer extremely high bandwidth and can schedule warps from any thread block to hide memory latencies, the inherent variance in elementwise operations, dynamic range checks, and branching significantly reduces effective throughput.

This interplay of divergence, irregular access patterns, and the computational overhead of transcendental functions contributes to bottlenecks in both training and inference. Tensor Cores can deliver enormous speedups for matrix operations by exploiting half-precision or TF32 formats, but they offer less benefit to nonlinear kernels that do not align with the specialized numeric pipelines. The performance gap grows when sequences become longer or require more complex attention mechanisms, compounding the dataflow disruptions in every pass of forward and backward propagation. Although these nonlinear operations remain essential for stable training and expressive power, their unpredictable data dependencies and branching logic create a mismatch with GPU execution flow, and they often require specialized kernel optimizations or algorithmic restructuring to mitigate the underutilization they can cause.

Sparsity and Data-Dependent Patterns

Sparsity and data-dependent patterns in Transformer-based LLMs introduce a set of challenges that go well beyond the usual concerns of dense matrix-matrix multiplication. Traditional Transformer layers already stress GPU memory bandwidth and computational throughput, but newer techniques like sparse attention, mixture-of-experts layers, and selective token routing exacerbate these bottlenecks by introducing inherently irregular dataflows. Rather than operating on neatly packed tensors, these operations frequently perform dynamic indexing, gather/scatter data movement, and conditional computation that cannot be easily "vectorized" or coalesced. The result is a departure from the dense, straightforward access patterns that GPUs were originally optimized to handle.

From an architectural standpoint, GPUs rely on a hierarchical memory system to hide memory latency and keep hundreds of threads running concurrently. Sparse operations undermine this design by triggering random or highly irregular access patterns, which break the assumptions of spatial and temporal locality that caches and memory coalescers depend on. When threads in a warp request data scattered across memory, the hardware struggles to batch and combine these loads, resulting in multiple smaller transactions instead of a single large coalesced one. This non-coalesced access lowers effective bandwidth and increases the frequency of cache misses, which in turn stalls the SMs. Over time, these stalls accumulate into larger latency penalties, further slowing down the pipeline.

Complicating matters, the GPU's warp-based execution model assumes that all threads in a warp will follow a similar execution path. Sparse operations, however, typically involve conditional branches and partial computations. Some tokens may be routed to different experts in an MoE layer or attend to different subsets of tokens in a sparse attention mechanism. Such data-dependent branching can cause warp divergence, where subsets of threads in the same warp are forced to idle while others carry out different operations. Divergence also complicates the use of shared memory for caching intermediate results, since different threads within the same block may be fetching data from disjoint memory regions. Meanwhile, partial synchronization points can arise when only a subset of threads must wait for certain sparse kernels to complete before proceeding. Since GPUs rely on massive parallelism and deep pipelining, these synchronization stalls disrupt execution, reducing overall throughput. The impact extends beyond idle threads within a warp; entire warps or blocks may also remain idle, waiting for others to synchronize. This further diminishes the concurrency that GPUs depend on for efficiency.

These factors collectively prevent GPUs from reaching their theoretical FLOPS and bandwidth ceilings when executing sparse, data-dependent Transformer workloads. The result is lower occupancy, fewer active warps per SM, and higher latency, as hardware resources frequently stall on memory or synchronization. GPUs like the H100 mitigate some inefficiencies with **structured sparsity**, accelerating statically pruned weights. However, they struggle with dynamic sparsity in advanced LLMs, where sparse attention and MoE introduce irregular memory access and warp divergence. Unlike structured sparsity, which optimizes pre-defined weight pruning, these patterns are algorithmically driven and unpredictable, making static

optimizations insufficient. As a result, tighter integration between algorithm design and hardware architecture to explicitly address dynamic sparsity is needed. While sparsity and data-dependent execution are effective for managing large context windows and parameter counts, they also exacerbate the “Memory Wall,” underscoring the fundamental mismatch between general-purpose GPU architectures and the irregular execution flows of modern Transformer variants.

So far, we’ve explored how the interplay between Transformer architecture and the Memory Wall creates bottlenecks within a **single GPU**. However, as LLMs continue to grow, they far exceed the capacity of any single accelerator, necessitating a shift towards multi-GPU scale-out architectures. Distributing the workload across multiple GPUs helps circumvent the limitations of individual devices, but it doesn’t eliminate bottlenecks—it merely shifts them to a new frontier: the **Interconnect Wall**. The bandwidth and latency challenges we’ve discussed now grow rapidly, as hundreds or even thousands of interconnected GPUs must work in unison. In the next section, we’ll delve into the intricacies of this Interconnect Wall and its profound impact on the scalability of ever-larger LLMs.

The Interconnect Wall

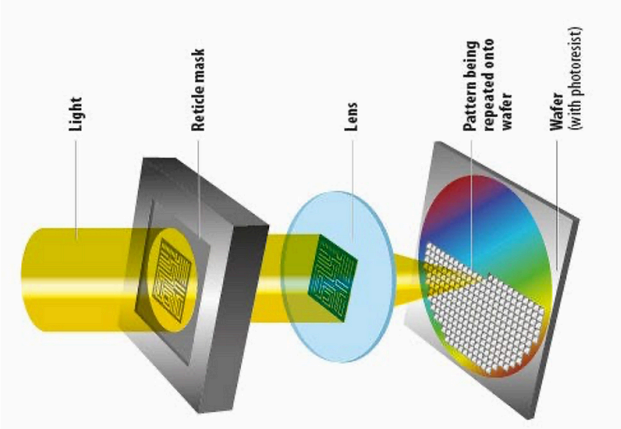
Scaling LLMs to meet growing compute and memory demands initially relied on a **scale-up** approach. This strategy involved enhancing individual accelerator chips by increasing core counts, memory, and bandwidth. Additionally, multiple accelerators were interconnected within the same server to boost overall compute and memory capacity. The original Transformer model from Vaswani et al. (2017) exemplifies this early approach: it was trained on a single machine with just 8 NVIDIA P100 GPUs, with each training step taking approximately 0.4 seconds for base models. These models required only 100,000 steps or 12 hours of total training time.

However, as LLMs have grown exponentially in size, the scale-up approach has become inadequate, necessitating a shift to **scale-out** architectures. Instead of relying on a single multi-GPU server, modern training runs are distributed across massive clusters of accelerators. This shift is exemplified by Grok 3, which was trained on the Colossus supercomputer, utilizing 200,000 NVIDIA H100 GPUs. Grok 3's training spanned 80 days and consumed an estimated 200 million GPU-hours. The contrast is striking: once, 8 P100 GPUs could train a model in under 12 hours, whereas today, 200,000 H100 GPUs, each delivering 105 times the FP16 FLOPS of a P100, train for over two months. This massive scale highlights the current paradigm in LLM training, where advancements hinge not just on model architecture improvements but on the ability to orchestrate compute across large-scale distributed systems.

While scale-out architectures have unlocked unprecedented model sizes and capabilities, they have introduced the "**Interconnect Wall**" as a critical bottleneck. Unlike single-machine training where data exchange is minimal, large-scale clusters must coordinate massive parameter and gradient exchanges. Inefficiencies in interconnect bandwidth and synchronization can lead to extreme slowdowns, with some collective operations consuming up to 90% of training time. This makes hardware-software co-design essential, balancing computation with data movement to ensure scalability.

Limits of Scaling-up a Single GPU

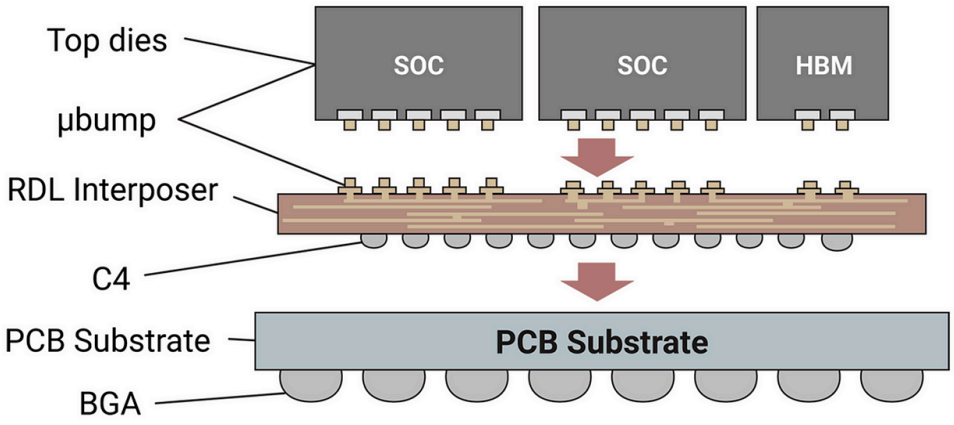
A scale-up approach that integrates more compute cores, larger on-die SRAM, and specialized compute units into a single package eventually encounters fundamental manufacturing limits. One of the primary constraints is the **reticle size** in semiconductor fabrication, which is governed by the capabilities of photolithography.



The size of the reticle in semiconductor manufacturing plays a crucial role in determining chip capabilities. This reticle serves as a template during **photolithography**, the process used to imprint intricate circuit patterns onto silicon wafers. However, the constraints of optical physics and semiconductor fabrication impose strict limits on its dimensions. For a given process-node and photolithography equipment, there is a maximum reticle size that can be effectively utilized. This restriction directly impacts chip design by capping the total area available for transistors. Since transistors are the fundamental building blocks of computation, a limit on their number directly restricts the computational capacity that can be achieved on a single integrated circuit. Therefore, the reticle size becomes a primary constraint, acting as a bottleneck that restricts the potential processing power of individual chips by limiting transistor density.

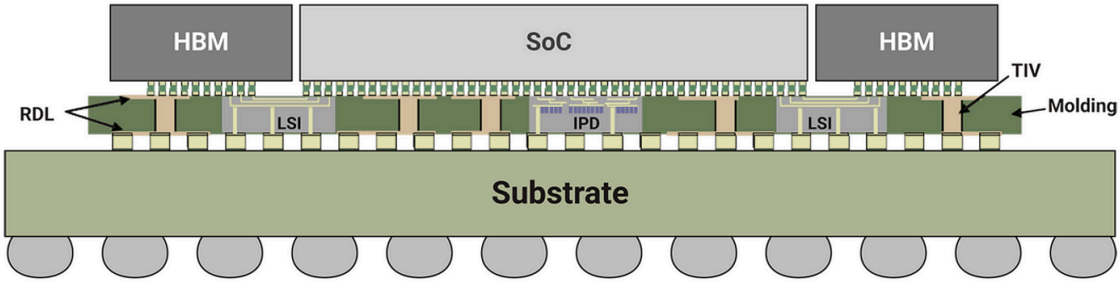
Addressing Single Chip Scale-Up Limitations : Advanced Packaging

Advanced packaging techniques, such as chiplet-based Multi-Chip Modules (MCMs), address these scaling limitations of single package monolithic chip designs by disaggregating Systems-on-Chips (**SoCs**) into smaller, specialized **chiplets**. These chiplets are interconnected within a single package, allowing MCMs to overcome the fabrication constraints of single-die architectures. Among advanced packaging techniques, **2.5D** architectures, such as TSMC's **CoWoS** (Chip-on-Wafer-on-Substrate), enable high-bandwidth, low-latency interconnects by placing chiplets on an intermediary substrate.



Source: TSMC **CoWoS** Architecture

CoWoS relies on several key components working in concert. The **interposer** serves as the foundation offering a dense connectivity layer where chiplets are arranged. **TSVs** acting as microscopic conduits enable signal transmission through the silicon, facilitating dense interconnectivity. Redistribution Layers (**RDLs**), layered atop the interposer and chiplets, form intricate wiring networks that fan out and route signals from TSVs across the interposer. The attachment of chiplets to the interposer is achieved using **microbumps**, fine-pitch connectors that establish high-bandwidth links between chiplets and the interposer. Finally, the **substrate** provides the mechanical foundation for the entire package, supporting the interposer and chiplets while also routing power and signals to the system's motherboard. This integration is critical for workloads such as LLM inference, where substantial computational power and memory bandwidth are essential.



CoWoS is available in three main variants: CoWoS-S, CoWoS-R, and CoWoS-L. The **CoWoS-L** variant, in particular, has significantly expanded the capabilities of 2.5D packaging. By utilizing RDL interposers with Local Silicon Interconnect (**LSI**) bridges, CoWoS-L extends the package size to span up to multiple reticle limits, pushing performance boundaries while introducing new thermal design challenges. CoWoS also enhances design flexibility by enabling modular integration of chiplets from different suppliers. This capability allows GPU manufacturers, such as NVIDIA, to integrate high-bandwidth memory (HBM) from SK Hynix or Micron, creating optimized single-package solutions that leverage best-in-class components.

Industry Example: Scaling-up NVIDIA Blackwell GPU

Blackwell GPU exemplifies the cutting edge of semiconductor manufacturing, leveraging **2.5D** packaging with **CoWoS-L** to push the boundaries of GPU design. To overcome the reticle size limitation in extreme ultraviolet (**EUV**) lithography, Blackwell splits its GPU logic across two reticle-sized dies, interconnected using **NV-HBI**—a high-speed interconnect delivering ~10 TB/s of bandwidth. This architecture effectively combines two dies into a single GPU, increasing compute density while circumventing the physical constraints imposed by monolithic chip scaling.

As explained earlier, a major challenge in single GPU scaling is the **reticle size limit** of extreme ultraviolet (**EUV**) lithography. The Blackwell chip is fabricated using TSMC's 4NP process-node that utilizes ASML's EUV machine with a Numerical Aperture of **0.33 NA**. In EUV lithography, this 0.33 NA specification limits the exposure field to **858 mm²** (26 mm x 33 mm). This restriction stems from the 4x reduction optics employed in 0.33 NA EUV systems, where the standardized mask size of 104 mm x 132 mm translates to a maximum exposure field of 26 mm x 33 mm. The fundamental constraints imposed by 0.33 NA projection optics prevent further scaling of the field size, as any increase would degrade image fidelity due to optical aberrations. These limitations directly affect process efficiency, lithographic accuracy, and overall manufacturability.

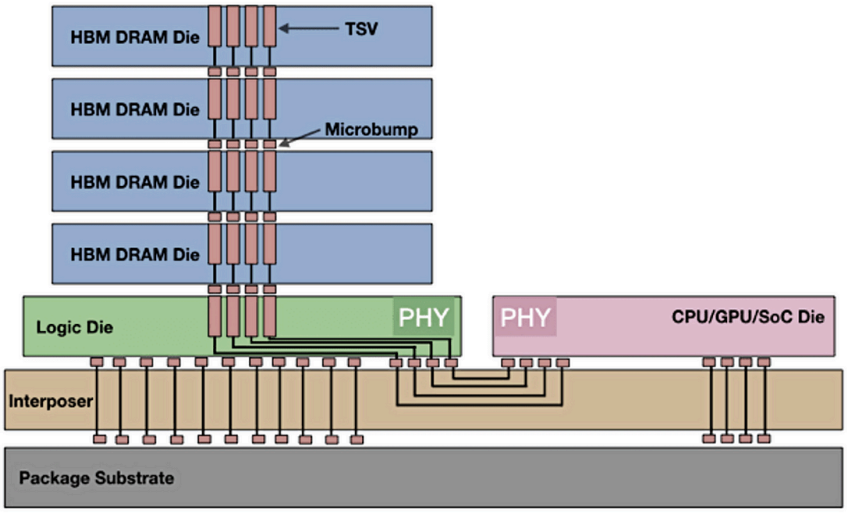
To meet its performance targets requiring approximately 208 billion transistors, which far exceeded what could fit on a single die within the 858 mm² reticle limit, NVIDIA split the chip into two co-packaged dies of about 429 mm² each. This design approach effectively doubled the available transistor budget and improved manufacturing yields. Unlike previous NVIDIA GPUs that utilized **CoWoS-S,** Blackwell required the more advanced **CoWoS-L** packaging due to its finer "bump pitch"—the spacing between micro-bumps connecting the chip to the substrate. This reduced bump pitch significantly increased the density of contact pads, a critical factor in enabling Blackwell's high-performance features, such as the **NV-HBI 10TB/s** die-to-die interconnect for rapid data transfer between chiplets. However, it also introduced a new bottleneck at the interconnect between the two dies. To operate as a unified processor, data must now communicate via NVIDIA's NV-HBI.

While NV-HBI is an advanced interconnect, fundamental laws of physics and engineering constraints prevent communication between two separate pieces of silicon from matching the speed and efficiency of a monolithic die. Several factors contribute to this limitation: the increased physical distance signals must travel, the overhead imposed by the NV-HBI protocol and its physical transceivers for off-chip signaling, and the added complexity of maintaining cache coherency across dies. Consequently, even with cutting-edge NV-HBI, the computational throughput of this dual-die architecture remains inherently constrained, representing a theoretical performance ceiling compared to a single-chip implementation.

Challenges of Advanced Packaging

Advanced packaging designs promise significant performance gains but also bring formidable engineering challenges.

Thermal Management
Dies stacked vertically face challenges dissipating heat from upper layers. CoWoS-L technology, as implemented in NVIDIA's Blackwell GPUs, amplifies this thermal management dilemma. CoWoS technology, when used to integrate vertical HBM stacks and logic dies with dense TSV interconnects, already presents significant heat concentration challenges. CoWoS-L further intensifies this issue with its significantly smaller bump pitch, which, while enhancing interconnect speeds and power efficiency, results in even greater heat density. Furthermore, the smaller size of the micro-bumps in CoWoS-L makes them inherently more vulnerable to thermal stress.



Source

Mechanical Reliability
Disparities in the coefficients of thermal expansion (**CTE**) between silicon chiplets, the interposer, and the organic substrate becomes increasingly problematic under thermal fluctuations. During operational thermal cycles, these CTE mismatches induce differential expansion and contraction, concentrating mechanical stresses at critical interconnection points. Solder joints, TSV-based connections, and micro-bump arrays, all vital for electrical and mechanical integrity, become focal points for these stresses. Repeated thermal cycling then risks solder joint fatigue, interposer cracking, and degradation of micro-bump interconnections, posing a constant threat to the package's structural integrity and long-term operational reliability.

Signal Integrity and Crosstalk
Achieving multi-terabit-per-second communication in chiplet systems immediately confronts limitations in signal integrity within densely packed wiring. The sheer demand for bandwidth

necessitates extremely high-density interconnects, inherently leading to timing skew and electromagnetic interference. Close proximity of signal lines, essential for these speeds, inevitably fosters crosstalk, where signals bleed from one line to another, corrupting data. Simultaneously, variations in electrical path lengths and impedance mismatches across these dense interconnects create timing skew, causing signals to arrive out of sync and potentially leading to data errors. These signal integrity issues are not minor disturbances; they are core obstacles manifesting as degraded signal quality, increased error rates, and ultimately, restrictions on the attainable communication bandwidth between chiplets.

Power Delivery Network (PDN)
Powering chiplet assemblies introduces significant complexities arising from the diverse power requirements of individual chiplets. Each chiplet, performing specialized functions, may demand distinct power levels, creating a heterogeneous power landscape within the package. This diversity amplifies the sensitivity of these high-performance systems to noise propagation. Noise originating in one area of the Power Delivery Network—whether from switching activities or fluctuating current demands—has the potential to cascade throughout the entire package. This ripple effect across interconnected power planes can disrupt the performance of other chiplets, making the management of noise and the provision of stable power a formidable challenge in ensuring reliable chiplet operation.

Cost and Yield
Fabricating large interposers and implementing fine-pitch micro-bumping, processes essential for high-performance chiplets, are inherently complex and expensive. Wafer thinning, also frequently employed, introduces additional process steps and yield risks. These sophisticated manufacturing requirements translate directly into escalating production costs. Furthermore, maintaining acceptable yield rates in these intricate processes presents a persistent challenge. Process variations, equipment calibration issues, and the sheer complexity of handling large, thin wafers all contribute to potential yield losses, making cost control and yield optimization critical, yet inherently difficult, aspects of chiplet manufacturing viability.

As LLMs surpass the trillion-parameter mark, even the most advanced scale-up architectures struggle to keep pace. Training such LLMs demand terabytes of memory to store parameters, activations, gradients, and optimizer states. For instance, a trillion-parameter model using a stateful optimizer like Adam requires nearly 16 TiB of GPU memory at 16-bit precision. These extreme memory requirements, coupled with skyrocketing computational demands, render scale-up solutions impractical. Consequently, the industry shifted toward scale-out architectures to sustain LLM growth.

Scale-Out Architectures

Scale-out architectures solve this challenge by distributing the workload across horizontally scaled GPU clusters. This approach not only makes trillion-parameter LLMs feasible, but also provides the flexibility to expand hardware resources as new, larger models emerge. To optimize performance across large GPU clusters, several complementary parallelization strategies efficiently distribute data and computation.

Data Parallelism splits the training dataset among GPUs, allowing each to process mini-batches independently while synchronizing updates—ideal for distributed training without modifying the model. Model Parallelism divides the model itself across GPUs, enabling the training of massive architectures that exceed single-device memory limits. Pipeline Parallelism partitions the model into sequential stages, assigning each stage to a different GPU to overlap computation and communication, improving throughput. Tensor Parallelism further optimizes efficiency by distributing large matrix operations across multiple GPUs, reducing memory bottlenecks in large-scale LLMs.

Achieving efficient scale-out parallelization requires low-latency, high-bandwidth communication both within a node and across a GPU cluster. Different solutions cater to these two levels: **intra-node** interconnects like NVLink, and NVSwitch facilitate high-speed communication between GPUs within a server-node (such as HG B200) or a SuperPOD (such as GB200 NVL72), while **inter-node** networking technologies such as Ethernet, RoCE, and InfiniBand enable scaling across multiple servers. Let's examine these technologies and how they evolved to address the growing demands of large-scale LLM training.

Intra-Node GPU Communication

PCI Express (PCIe)

Initially, PCI Express (PCIe) connected GPUs to the system's root complex, providing offload capabilities at generations like PCIe 4.0 (~32 GB/s for x16) and 5.0 (~64 GB/s for x16). While sufficient for early GPU computing, PCIe's point-to-point interconnect design and relatively high latency made it a bottleneck in multi-GPU deep learning, as data repeatedly traversed the CPU or chipset.

NVLink

To address PCIe's limitations, NVIDIA developed NVLink, offering direct, point-to-point GPU-to-GPU connections for large data transfers. NVLink was designed as a dedicated, high-bandwidth, low-latency interconnect specifically engineered to address the intra-node communication bottleneck. Unlike PCIe, which is a general-purpose bus, NVLink is a point-to-point, direct connection between GPUs. This direct connection, leveraging advanced signaling and streamlined protocols, allows NVLink to deliver significantly higher bi-directional bandwidth and lower latency compared to PCIe for GPU-to-GPU data exchange. NVLink effectively bypasses the PCIe bus for critical GPU-to-GPU communication, creating a fast efficient pathway for data sharing and synchronization within a single server-node/SuperPOD. Often, GPUs incorporating NVLink are packaged in **SXM (Server PCI Express Module)** form factors, which provide the necessary power and cooling for these high-performance components and facilitate dense GPU deployments in server environments.

NVSwitch

As systems grew to eight GPUs or more, wiring a full NVLink mesh became complex. NVSwitch acts as a high-radix, non-blocking crossbar switch designed to aggregate multiple NVLink connections. It essentially creates a centralized hub that allows every GPU within a node to communicate with every other GPU via NVLink, through the NVSwitch fabric. This allows for the

creation of highly interconnected intra-node GPU systems, significantly expanding the number of GPUs that can efficiently communicate at NVLink speeds within a server-node/SuperPOD. NVSwitch enables the construction of powerful single-node “GPU servers” with a large count of interconnected GPUs, facilitating larger local model partitions and minimizing reliance on slower inter-node communication for certain aspects of training. By managing and optimizing the communication paths between GPUs within a node, NVSwitch further enhances the efficiency and scalability of intra-node parallelism, building upon the foundation laid by NVLink. In products like NVIDIA DGX servers, multiple NVSwitches efficiently interconnect all GPUs, removing in-node bottlenecks for large-scale distributed training.

Inter-Node Networking

Ethernet (TCP/IP)
For multi-node clusters, early setups used **Ethernet** running **TCP/IP**. The traditional TCP/IP stack, designed for general-purpose networking, introduces significant CPU overhead for data processing and transmission. This CPU involvement in every network operation increases latency and reduces the effective bandwidth available for application-level communication, especially when scaling to a large number of nodes requiring frequent inter-node data exchange. This overhead significantly impacts all-reduce-intensive workloads, making standard Ethernet less optimal for large-scale LLM training.

RoCE (RDMA over Converged Ethernet)
To mitigate TCP/IP overhead, HPC practitioners turned to **RDMA** (Remote Direct Memory Access), allowing direct reads and writes to remote memory buffers without extensive CPU involvement. RoCE (RDMA over Converged Ethernet) encapsulates RDMA within Ethernet frames, reducing latency and boosting throughput. While RoCE helps Ethernet-based clusters close some of the performance gap with specialized interconnects, ultra-high bandwidth needs still often push adopters toward purpose-built networking solutions.

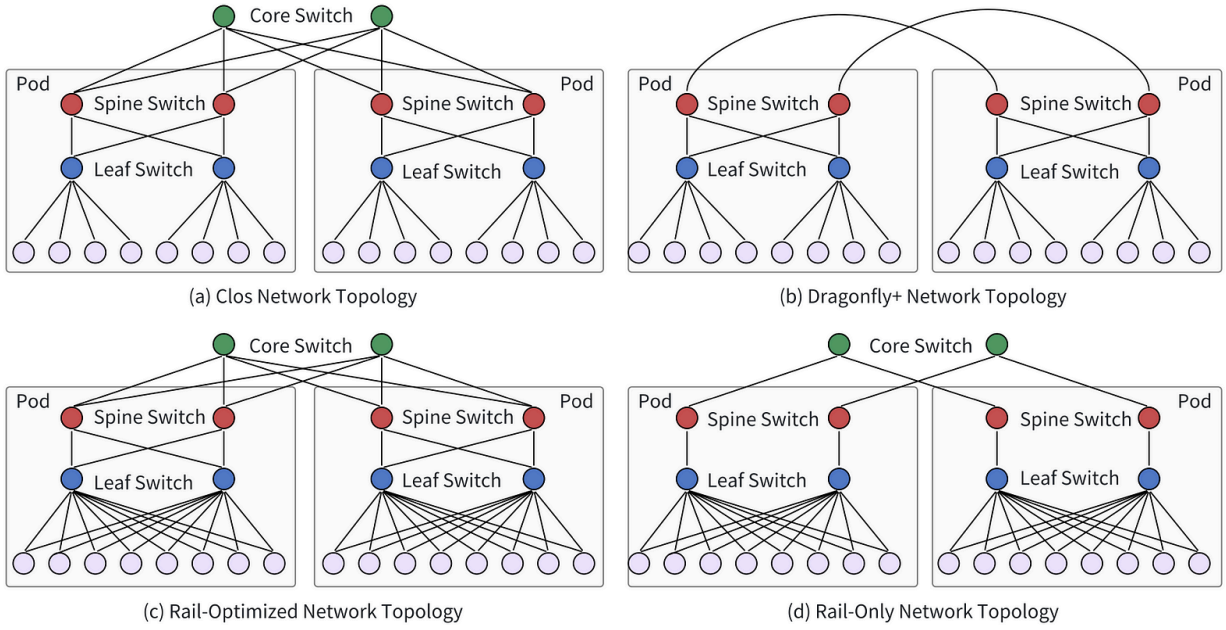
InfiniBand
For the most demanding distributed LLM training scenarios, where absolute performance and scalability are paramount, InfiniBand has emerged as the leading interconnect technology. InfiniBand is a purpose-built networking standard designed from the ground up for high-performance computing and distributed environments. It incorporates RDMA in its hardware and protocol design, offering native RDMA support without relying on encapsulation like RoCE. This native RDMA implementation, combined with optimized hardware and protocols, allows InfiniBand to consistently deliver superior bandwidth and lower latency compared to Ethernet-based solutions, including RoCE. InfiniBand also includes advanced features like sophisticated congestion management and Quality of Service (QoS) mechanisms to ensure reliable and predictable performance even at massive scale.

NCCL and CUDA-Aware Networking
Irrespective of the underlying interconnect technology realizing the full potential of these hardware advancements necessitates a sophisticated software layer. Hardware performance is unlocked through software like NCCL (NVIDIA Collective Communications Library), which

implements optimized all-reduce, broadcast, and other collective primitives. NCCL automatically detects and exploits whichever interconnect is present—NVLink, NVSwitch, RoCE, or InfiniBand—while CUDA-Aware Networking integrates these transfers directly with the GPU's memory space. Together, they minimize CPU intervention and ensure near-peak interconnect utilization in deep learning frameworks like PyTorch and TensorFlow.

HPC Topologies for Large-Scale GPU Clusters

As these GPU clusters expand to encompass hundreds or even thousands of nodes, tasked with exchanging multi-terabyte-per-second data streams, the arrangement of these interconnects—the network topology—becomes equally critical. Standard, simpler network layouts often falter under such immense traffic demands. To address these challenges and ensure efficient, scalable communication in large GPU clusters for demanding workloads like LLM training, specialized **HPC network topologies** are essential. Topologies such as **Fat-Tree, Dragonfly, and Torus** are specifically designed to overcome these limitations and are crucial for maintaining performance in large-scale distributed LLM training. These topologies address the challenges of massive scale by strategically reducing network diameter, providing multiple redundant paths, and more effectively balancing traffic loads.



By minimizing the **network diameter**, which is the maximum number of network hops between any two nodes, HPC topologies directly reduce communication latency. Instead of relying on a single, potentially congested route between nodes, HPC networks are characterized by **multiple redundant paths**. This redundancy is vital for workloads that involve heavy all-to-all or many-to-many communication patterns, as is common in data-parallel and model-parallel LLM training. These parallel paths help to avoid the creation of congestion hotspots, ensuring more consistent and predictable network performance across the cluster.

These topologies also deliver **scalability** and **fault tolerance**. The architecture of **Fat-Tree** designs, for instance, allows for incremental expansion by adding switch tiers, making it relatively straightforward to scale the network as the cluster grows in size. **Dragonfly** and **Torus** layouts employ a combination of local and global connectivity to efficiently link thousands of endpoints. This hierarchical and structured approach enables these topologies to meet the massive bandwidth requirements of modern LLMs without a drastic degradation in performance as the node count increases. This scalability is essential for accommodating the ever-growing size and complexity of LLMs. If one path experiences congestion or even fails entirely, the network can seamlessly reroute traffic through alternative paths. This inherent fault tolerance is critical for maintaining high cluster throughput and ensuring the stability of long-running training jobs, protecting against disruptions caused by transient network issues.

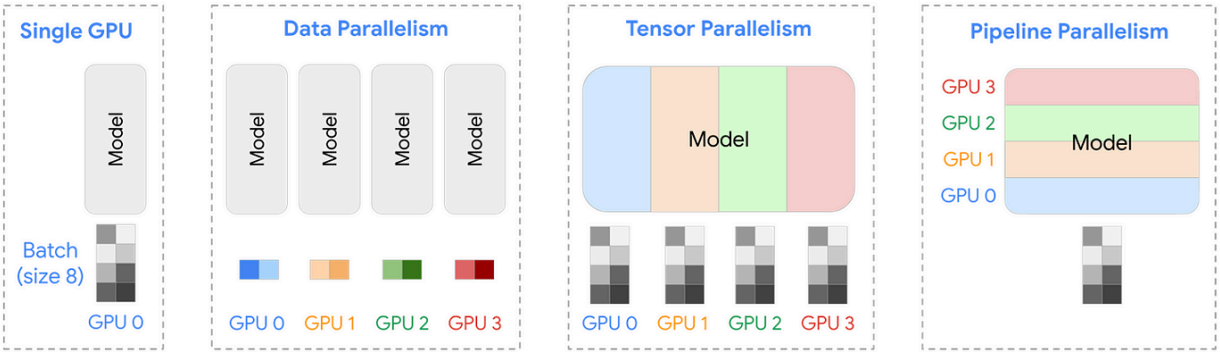
Despite advancements in high-bandwidth, low-latency interconnects and sophisticated network topologies, scaling massive LLMs reveals fundamental limits in scale-out architectures. As clusters grow to hundreds or thousands of nodes, the assumption of linear scalability breaks down. Communication overhead quickly saturates interconnects, creating a critical bottleneck—inter-accelerator communication constraints, known as the **“Interconnect Wall”.**

Scale-Out Limitations: Emergence of the Interconnect Wall

As distributed training scales from a handful of accelerators to hundreds or thousands, communication overhead becomes a formidable barrier. This “Interconnect Wall” arises when the cost of coordinating data and synchronizing computations outweighs the benefits of parallelization. In smaller clusters (2–8 accelerators), high-speed links effectively mask latency and bandwidth limitations; however, as the cluster’s **world size** (number of interconnected devices) grows, even minor inefficiencies—such as suboptimal routing or insufficient bisection bandwidth—can stall the entire job.

Although raw bandwidth is critical, factors like round-trip latency, contention, and fault tolerance also shape the Interconnect Wall. For collective operations such as all-reduce or all-gather, a single saturated link or a slight increase in latency can cause cluster-wide ripple effects, drastically reducing training throughput. These underlying bottlenecks expose the fundamental limits of scale-out architectures, highlighting a complex interplay of communication overhead, memory constraints, and system-level inefficiencies that collectively impede large-scale distributed LLM training.

Parallelization Strategies and The Interconnect Wall



Source

Data Parallelism: Distributing the Dataset
Data Parallelism partitions the training dataset into distinct subsets, assigning each subset to a separate GPU. Each GPU processes mini-batches independently using a full replica of the model, ensuring consistency through periodic synchronization of model updates. This approach effectively utilizes multiple GPUs without requiring model modifications, making it a widely used strategy for distributed training. However, it depends on an **all-reduce** operation to aggregate gradients across GPUs.

In large-scale deployments, all-reduce operations can consume over 90% of each training step, creating a significant bottleneck. To optimize memory usage, techniques like ZeRO and FSDP shard model states across GPUs, reducing per-device memory requirements. However, this optimization introduces additional communication overhead, as parameters, gradients, and optimizer states must be frequently exchanged to maintain consistency.

While data parallelism ensures logical consistency, frequent cluster-wide communication can slow down training, especially when substantial gradient updates require high-bandwidth networks. Since all-reduce is central to data parallelism, inefficiencies, whether due to limited bandwidth, network congestion, or synchronization overhead , can significantly impact performance. As cluster sizes grow, all-reduce overhead often becomes the dominant factor in per-step runtime, limiting scalability.

Tensor Parallelism: Distributing Matrix Operations
Tensor Parallelism focuses on distributing a model's weight matrices across multiple devices. By splitting large matrix operations across GPUs, this method enables efficient parallel computation of the dominant workloads in LLMs. Particularly useful for large LLMs that exceed the memory limits of a single device, tensor parallelism helps balance both computational and memory demands.

Tensor parallelism partitions each layer's parameters among multiple GPUs, reducing local memory usage but introducing significant communication of intermediate activations. Each micro-batch forward or backward pass now generates partial results that must be broadcast or gathered. If a layer is split across four GPUs, every forward pass can involve multiple parallel matrix multiplications interspersed with data exchanges. Tensor parallelism is frequently

combined with data parallelism, forming a hybrid scheme that can improve GPU utilization. Yet this approach intensifies demands on the interconnect, requiring fast and reliable data exchanges to keep partial activations synchronized.

Pipeline Parallelism: Overlapping Computation
Pipeline Parallelism introduces a staged execution model by dividing the LLM into sequential stages, each assigned to a different GPU. This approach enables overlapping computation and communication, improving throughput. While one GPU processes a given stage, another GPU can begin processing the next mini-batch, reducing idle time and mitigating memory bottlenecks.

In practice, a network's layers are divided into pipeline stages, and each stage resides on a specific GPU or group of GPUs. Micro-batches flow through the pipeline, and GPUs must wait for data from preceding stages. While pipeline parallelism can reduce memory usage (fewer layers per device), it also introduces pipeline bubbles and stage-to-stage communication overhead. GPUs must wait for output from the preceding stage, which can lower overall throughput if the network introduces extra delays.

Sequence parallelism
Sequence parallelism focuses on partitioning long input sequences among multiple GPUs. Though it alleviates memory demands for attention, distributing the sequences requires frequent exchange of partial attention maps. When combined with tensor parallelism, network traffic increases substantially in both the depth (layers) and width (sequence length) dimensions. This multidimensional partitioning further reveals bandwidth and latency constraints in the interconnect.

Expert Parallelism
Expert Parallelism (EP) is a specialized form of model parallelism tailored for Mixture of Experts (MoE) models, where only the expert layers are distributed across multiple GPUs, while the rest of the model remains unchanged. This approach enables efficient scaling by leveraging all-to-all redistribution of tokens or activations, offering flexibility but introducing frequent data transfers that can strain bandwidth.

A key advantage of MoE is its ability to increase the model's parameter count without a proportional rise in computation. However, routing tokens to specific experts often necessitates large-scale all-to-all operations, which become costly when tokens within a batch are assigned to different experts on separate GPUs. Each micro-batch step incurs expensive cross-node data shuffles, further compounded by load imbalance, where certain experts become "hotspots," receiving an excessive number of tokens, leading to increased communication and compute overhead on specific GPUs.

To mitigate these challenges, strategies such as Switch Transformer, which restricts each token to a single expert, and dynamic load balancing techniques help reduce congestion. However, efficient scaling ultimately depends on high-performance networking, ensuring that distributed systems can avoid bottlenecks and sustain optimal throughput.

Checkpointing, Fault Tolerance, and Concurrency Overheads

Beyond communication overheads, another key challenge in distributed LLM training is ensuring resilience against failures and optimizing memory usage. As workloads scale across thousands of nodes, maintaining fault tolerance and managing stateful execution become increasingly complex. Checkpointing, Fault Tolerance, and Concurrency Overheads play a crucial role in addressing these challenges, enabling efficient recovery from failures, optimizing memory utilization, and balancing computational loads to sustain high-performance training.

Checkpoint Overheads
As model sizes grow, checkpoint data can range from tens to hundreds of gigabytes per snapshot. Writing these checkpoints to a distributed file system or object store can saturate network links. Full checkpoints demand simultaneous writes from all GPUs, intensifying overall cluster traffic and risking performance collapse if the storage or network layers are not carefully orchestrated. Layer-wise or asynchronous checkpoint attempt to stagger the write load, but background traffic can still contend with real-time gradient synchronization, compounding the Interconnect Wall.

Fault Tolerance and Blast Radius
At large scale, the probability of node or link failure within a multi-day or multi-week training job becomes significant. Failures can force partial restarts, re-partitions, or slow re-synchronizations that temporarily overload the network. Topologies like a torus or mesh can see entire segments of the network impacted by a single node failure. Even well-connected fat-tree networks experience performance dips if specialized congestion control or resilience mechanisms are not in place.

Orchestration Spikes
When a fault occurs or a reconfiguration is triggered (e.g., resizing job resources to match load or forcibly replacing defective nodes), orchestration frameworks (SLURM, Kubernetes, or HPC schedulers) initiate data movement in bulk. This can produce "**network storms,**" where nodes exchange large volumes of model state or re-shard the training data, further exposing the cluster to potential slowdowns if not meticulously managed.

Memory Management Pressures

As LLMs continue to scale, the need for efficient memory allocation becomes paramount. Techniques like activation checkpointing and optimizer state sharding aim to reduce per-GPU memory footprint, but they come with their own trade-offs, particularly in terms of recomputation and communication overheads.

Activation Checkpointing and Recomputation
LLMs frequently exceed available device memory when processing large batch sizes and extended sequence lengths. Activation checkpointing addresses this by selectively storing only a subset of activations in memory and recomputing the remaining activations on-demand. This approach effectively reduces peak memory usage, allowing larger models to fit onto fewer

GPUs. However, this memory-saving benefit comes with a trade-off: significant computational overhead due to additional forward passes needed during recomputation. Specifically, the backward pass latency increases due to these recomputation steps, leading to higher FLOPS and increased utilization of on-chip computation and memory bandwidth.

Sharding Optimizer States
Techniques like ZeRO or FSDP remove redundant storage of optimizer states by distributing them across multiple GPUs. While this approach successfully cuts per-device memory usage, it relies on low-latency, high-bandwidth communication to synchronize shards during backpropagation. Hardware features such as RDMA, NVLink can help, but at massive scale, inter-node connections still risk saturation. These memory management optimizations therefore offer a trade-off: they enable larger models to train, but they also intensify the very communication overheads that define the Interconnect Wall.

Despite sophisticated model parallelization strategies, advanced high-bandwidth interconnect hardware, and complex HPC network topologies, communication overhead remains a significant barrier to large-scale distributed LLM training. Whether GPUs are connected via high-speed NVLinks, NVSwitches or InfiniBand fabrics, once clusters grow to hundreds or thousands of accelerators, the cost of synchronizing gradients, sharding model states, and handling inevitable failures can overshadow gains in compute throughput. This is the essence of the Interconnect Wall: as we push larger models across more nodes, the time spent moving data increasingly limits overall performance.

Yet, as LLMs continue to grow in complexity and scale, the demand for greater computational throughput and memory bandwidth pushes against a more fundamental barrier—one that extends beyond latency or interconnect constraints. At its core, this challenge is not just about **bits** and **FLOPS** but also about energy consumption—the critical limit of **watts**. This constraint, known as the “**Power Wall**”, imposes a fundamental ceiling on performance scaling.

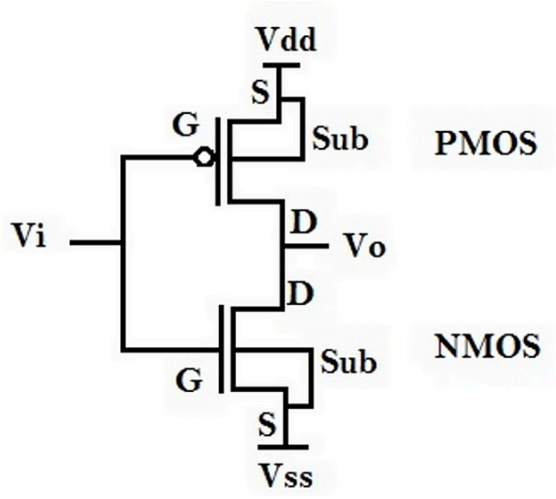
The Power Wall

The “**power wall**” represents a fundamental constraint in modern computing, significantly impacting the scalability of LLMs. This barrier arises from an increasingly difficult trade-off: the relentless demand for higher computational throughput, increased memory density, and greater bandwidth to support massive LLM workloads clashes directly with the rising power consumption and heat dissipation required to maintain operational stability. As GPUs scale to incorporate ever-larger transistor counts and denser memory architectures, their power consumption grows non-linearly, governed by fundamental semiconductor physics. In dense compute environments, this scaling issue becomes especially critical, as heat dissipation transitions from a design challenge to an absolute limiting factor. At a certain point, the heat generated—directly proportional to power consumption—can no longer be efficiently removed, imposing a hard thermal constraint. Beyond this threshold, even the most advanced cooling solutions become inadequate. When these limits are reached, thermal throttling mechanisms automatically activate, lowering clock frequencies and even deactivating cores, ultimately capping real-world performance. This creates the power wall, a fundamental barrier that prevents further scaling based solely on increasing computational and memory density. Consequently, the power wall is not just an energy consumption issue but, more crucially, a thermal management challenge that dictates the practical power delivery limits of computing devices.

The impact of the power wall is a widening gap between theoretical and practical compute capacity. While latest GPUs may boast over 1000+ TFLOPS (FP16/BF16) of theoretical performance, power constraints typically restrict sustained workloads to just 60–80% of this potential. This efficiency drop-off creates a significant bottleneck in real-world LLM training and inference, where sustained high compute throughput is essential. To fully understand this limitation, we must examine the fundamental physics of CMOS technology, which underpins modern processors and ultimately dictates the power-performance trade-offs shaping computing today.

CMOS Technology

Complementary Metal-Oxide-Semiconductor (CMOS) technology integrates pairs of p-type and n-type metal-oxide-semiconductor field-effect transistors (MOSFETs) to implement high-density logic functions efficiently. These complementary transistor pairs allow for push-pull circuit configurations where power is primarily consumed during switching operations. Ideally, static power consumption is very low; this historic advantage stems from the complementary nature of PMOS and NMOS transistors. In CMOS logic gates, paired PMOS and NMOS transistors are arranged so that when one conducts, the other remains off. This design ensures that no direct conductive path exists between the supply voltage (Vdd) and ground (Vss) under steady-state conditions, minimizing continuous power loss.



Unlike older bipolar junction transistors (BJTs) that consume power even when idle, CMOS transistors dissipate energy only during switching events, making them highly efficient. This enables CMOS gates to achieve remarkably low energy dissipation—measured in femtojoules per gate evaluation—while operating at picosecond speeds, facilitating high-speed computing. Coupled with exceptional density, CMOS technology allows for the fabrication of billion-transistor chips that maintain reliability. Additionally, its cost-effectiveness supports the production of complex chips at scale, highlighting an unmatched combination of performance, efficiency, density, and affordability.

This fundamental advantage has enabled the integration of billions of transistors on a single die, driving the advanced compute capabilities essential for modern GPUs. It also fueled the initial scaling benefits predicted by Moore's Law. However, these gains came with significant power management challenges. As transistor densities increased, the semiconductor industry relied on another key principle to control power consumption, known as **Dennard scaling**. However, this approach eventually broke down, leading to the power wall predicament we face today.

Dennard Scaling

Historically, Dennard scaling (also known as MOSFET scaling) offered an elegant solution to the power problem. First described by Robert H. Dennard in 1974, this principle stated that as transistors shrank in size, their power density would remain constant. Specifically, reducing transistor dimensions by a factor of **1/k** would permit voltage and current scaling by the same factor, allowing clock frequencies to increase by a factor of **k** for roughly the same power draw per unit area.

This remarkable property meant that smaller transistors could operate faster while consuming proportionally less power individually, keeping the overall power density unchanged. In practical terms, this allowed chip designers to pack more transistors into the same die area and run them at higher frequencies without encountering thermal issues. This virtuous cycle drove processor performance improvements for decades, with each new process-node delivering substantial gains in both performance and energy efficiency. However, these benefits have waned

significantly since the mid-2000s when process-nodes dropped below 90nm. Voltage no longer scales down seamlessly with transistor size due to quantum tunneling effects and subthreshold leakage. While the ideal threshold voltage (Vt) should scale with transistor dimensions, maintaining a minimum threshold voltage became necessary to prevent excessive leakage, creating a "**voltage wall**" around 1V.

Additionally, leakage currents rise sharply as transistors shrink, introducing significant static power consumption that wasn't accounted for in the original Dennard scaling model. This means boosting clock frequencies further escalates both dynamic and static power consumption in a superlinear fashion. Even though transistor counts continue to grow according to Moore's Law (doubling roughly every 18–24 months), processor performance today is bounded less by raw transistor availability and more by power delivery and thermal dissipation limits. This shift marks the era of ***post-Dennard scaling***, where increasing transistor density no longer directly translates to proportional performance gains.

The Switching Power Equation

The switching power equation provides a mathematical foundation for understanding the link between performance and power consumption (P) in CMOS chips:

$$P = QfCV^2 + VI_{leakage}$$

Where Q is the activity factor, C represents the load capacitance, V the operating voltage, and f the frequency of switching. This equation reveals that performance gains, often pursued through higher clock frequencies (f) and sometimes higher voltages (V) to drive transistors faster, directly and significantly increase power demands. The squared term for voltage (V^2) emphasizes the dramatic impact of voltage scaling on power. This relationship creates a critical dynamic: as GPUs are pushed for higher performance in LLM workloads, they inevitably face thermal throttling when heat dissipation exceeds the system's safe limits, especially under compute intensive workloads like LLM training. Modern GPUs typically operate very close to their thermal limits, with junction temperatures often reaching 85–95°C during sustained workloads, leaving minimal thermal headroom for performance optimization. Additionally, capacitance C is sum of gate capacitances, diffusion capacitances, and interconnect (wire) capacitance. As transistors continue to shrink, wire capacitance doesn't scale down as effectively as the others. As a result, it becomes a larger fraction of the total capacitance, which in turn has a significant impact on both power consumption and signal delay.

Moreover, the era of post-Dennard scaling highlights a concerning discrepancy between gains in transistor density and corresponding improvements in power efficiency. With each new semiconductor process node, transistors shrink by a scaling factor S (reducing linear dimensions by 1/S and increasing density by S^2), theoretically enabling performance scaling towards S^3 via higher density and frequency. However, the foundational principle of Dennard scaling—constant power density—ceased to hold primarily because supply voltage (Vdd) could

no longer be scaled down proportionally with transistor dimensions. Attempting to do so resulted in unacceptable increases in static leakage current as threshold voltages (Vt) approached physical limits, making static power a dominant factor in modern GPUs. While transistor switching speeds improved, this inability to aggressively reduce voltage negated much of the expected power benefit for dynamic switching. Additionally, factors influencing dynamic power, such as switched capacitance (C), scaled imperfectly—often simplified to roughly 1/S for gate capacitance but with interconnect capacitance scaling less favorably. Consequently, this combination of rising static power dominance and less-than-ideal dynamic power scaling means that overall power efficiency improvements now significantly lag behind the gains achieved purely through increased transistor density.

This discrepancy leads to a situation where power efficiency improvements lag behind performance gains. Instead of the full theoretical potential, power efficiency improves by roughly 40% per generation, according to estimations and empirical observations, further highlighting the growing power wall challenge. This efficiency gap explains why modern GPUs with billions of transistors still struggle with power density challenges despite advanced manufacturing processes.

Pollack's rule

The scaling effects of power and performance manifest directly in single-core computational capability. Pollack's rule, formulated by Intel's Fred Pollack, describes how scalar performance scales with the square root of transistor resource allocation, whereas power consumption can grow at an even faster rate. Mathematically, when using r transistor resources, scalar performance only rises to: Perf(r) = $\sqrt{r}$, while power consumption increases much more aggressively as $\mathbf{P = Perf^{\alpha}}$, where **α** is empirically observed to be approximately 1.75 for modern processors. This means power increases nearly quadratically with targeted performance improvements. The power can be expressed as:

$$P = \mathrm{Perf}^{\alpha} = (\sqrt{r})^{\alpha} = r^{\alpha/2}.$$

With α = 1.75, this becomes $P \propto r^{0.875}$, demonstrating how quickly power requirements escalate with increased transistor resources. For example, doubling the transistor count ($r \rightarrow 2r$) only improves performance by about 41% ($\sqrt{2} \approx 1.41$), but increases power consumption by approximately 84% ($2^{0.875} \approx 1.84$).

This disproportionate relationship between resource allocation and performance gains explains why simply adding more transistors to processors yields diminishing returns, pushing the industry toward specialized architectures and heterogeneous compute solutions that allocate transistor resources more efficiently based on workload characteristics. The implications of Pollack's rule become even more significant when considered alongside another major challenge in semiconductor scaling: the phenomenon known as dark silicon.

Dark Silicon and the Voltage Scaling Barrier

As transistor geometries shrank below the 65nm node, static leakage currents in nominally “off” transistors became a substantial component of total power consumption. While quantum tunneling through thin gate oxides contributes, the dominant factor is often subthreshold leakage, which increases exponentially as the transistor’s threshold voltage (Vt) is lowered to enable faster switching and lower operating voltages. Process engineers face a critical trade-off: a lower Vt enhances performance but drastically increases this leakage. Consequently, maintaining acceptable static power imposes a practical minimum Vt floor, exacerbated by quantum effects and other short-channel behaviors at advanced process-nodes.

Reliable high-frequency operation requires a supply voltage (Vdd) sufficiently above this Vt threshold to ensure transistors switch quickly enough. Attempting to operate below the Vdd required for a target frequency results primarily in timing failures and functional errors, rather than directly causing an exponential increase in leakage. Therefore, the practical Vt floor dictates a corresponding minimum Vdd floor needed to achieve high performance goals (historically in the ~0.7–1.0V range for many planar MOSFETs, though the exact value varies depending on transistor types like FinFETs or GAAFETs).

This inability to continue scaling voltage aggressively alongside feature size marks the breakdown of traditional Dennard scaling, meaning that power density (**watt**/mm²) no longer scales favorably with smaller transistors. While rooted in fundamental physics, this barrier presents immense ongoing engineering challenges, tackled by innovations like High-K Metal Gate stacks, FinFETs, and emerging Gate-All-Around (GAA) structures designed to improve electrostatic control and mitigate leakage.

The direct architectural consequence is that we can integrate far more transistors than we can afford to power and cool simultaneously at peak performance. This leads inevitably to designs where significant portions of the chip must be selectively power-gated (“**dark silicon**”), operated dynamically at reduced voltage and frequency states (“**dim silicon**” via DVFS), or functionality must be implemented using specialized hardware accelerators (like the Tensor Cores) which execute specific tasks far more energy-efficiently than general-purpose cores.

Thermal Design Power (TDP) Constraints

Managing “dark” and “dim” silicon is fundamentally tied to thermal constraints. While total chip power may be capped, the trend of packing more transistors into smaller areas significantly increases power density, making heat dissipation a central challenge in chip design and sustained performance. This thermal limitation is formally captured by the concept of Thermal Design Power (TDP), which represents the maximum amount of heat a chip is expected to generate under typical high-load conditions. TDP, in turn, defines the cooling requirements and plays a crucial role in shaping both the achievable performance and the architecture of modern processors and accelerators. Unlike transient power spikes, which may briefly exceed TDP during burst workloads, TDP sets a ceiling on sustained power draw—exceeding it over time

risks overheating, thermal damage, and long-term reliability issues. The relationship between temperature rise above ambient (ΔT) and TDP follows the basic heat transfer equation:

$$T_{\max} = \mathrm{TDP} \times (R_{\mathrm{conv}} + \frac{k}{A}),$$

where **Tmax** is the maximum temperature rise, **Rconv** is the heatsink thermal convection resistance, **k** incorporates many-core design properties and **A** is the chip area. This relationship reveals why power density becomes increasingly problematic as chips shrink or become more densely packed with transistors. When chip area (A) decreases, the temperature rise increases proportionally if TDP remains constant. Conversely, maintaining the same temperature requires reducing TDP as chip area shrinks, directly limiting performance.

CoWoS Packaging and Thermal Challenges

Balancing thermal design power (TDP) constraints is deeply intertwined with advanced packaging strategies that enhance memory bandwidth and interconnect density. As discussed earlier, CoWoS packaging offers significant performance advantages through high-density integration, but this same density introduces formidable thermal management challenges, exacerbating power wall constraints.

Several factors contribute to the complexity of thermal regulation in CoWoS packaging. Vertical HBM stacks create localized hotspots with high thermal resistance paths to heat sink, while densely packed TSVs can contribute to localized heating by concentrating power delivery and disrupting lateral heat spreading paths within the silicon. The silicon interposer further impedes heat dissipation, and the RDLs add additional thermal barriers. Compounding these issues, differences in the thermal expansion coefficients of various materials introduce mechanical stress under temperature fluctuations, raising reliability concerns. While this architecture mitigates traditional photolithographic reticle size limitations, it also increases power density, amplifying thermal challenges and complicating heat management.

Industry Example: NVIDIA**’s Blackwell—**The impact of CoWoS-L packaging is evident in NVIDIA’s Blackwell GPUs (B100 and B200), which utilize this technology to achieve 10 TB/s interconnect speeds through carefully positioned LSI bridges. However, thermal expansion mismatches between GPU chiplets, bridges, the RDL interposer, and the motherboard substrate can lead to warping and operational failures under stress. To address these challenges, NVIDIA redesigned the top metal layers and bump structures, improving yield and reliability. This example underscores the delicate balance between maximizing compute density and managing the thermal and mechanical complexities inherent in CoWoS-L packaging.

Thermal-Induced HBM Slowdown

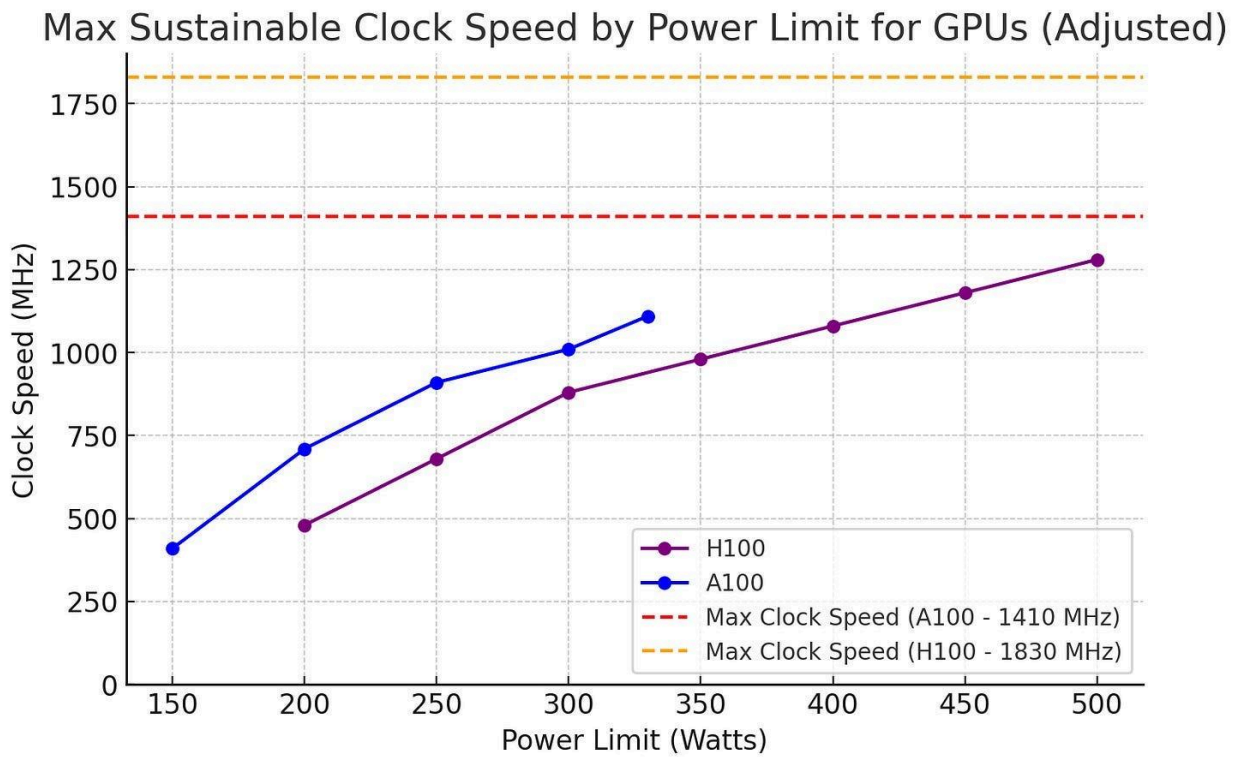
HBM performance also suffers significantly under heat because charge stored in its DRAM’s tiny capacitors leaks away faster at higher temperatures, requiring more frequent refresh cycles. As

GPUs operate at high utilization and generate intense heat, the nearby HBM becomes more vulnerable to thermal degradation and must refresh more often to prevent data corruption. Each refresh operation removes memory from availability and consumes extra power, creating a feedback loop in which the temperature of the HBM continues to climb, prompting even more refreshes. At smaller process-nodes (< 7nm) where leakage and rowhammer vulnerabilities become more pronounced, and with CoWoS packaging placing HBM in close proximity to hot GPU logic, managing heat dissipation grows increasingly complex. System architects must therefore consider thermal throttling, larger buffering mechanisms, and other design adjustments to ensure reliability and maintain memory throughput. This highlights how thermal density and power constraints are tightly intertwined across both GPU compute and HBM subsystems.

Thermal Limits and Power Throttling in GPUs

When GPUs operate under sustained high computational loads, typical during LLM training or inference, the physical limitations of silicon become evident through the complex interplay between switching power consumption and the thermal management required for cooling. Dynamic (switching) power consumption occurs as transistors change states during computation. As billions of transistors rapidly switch states under intense workloads, power consumption quickly approaches the GPU's TDP, typically up to 700W for an NVIDIA H100 (SXM) and up to 1KW for an B200.

To manage these power constraints, GPUs employ sophisticated techniques such as Dynamic Voltage and Frequency Scaling (DVFS). DVFS dynamically reduces the voltage and frequency of the SMs as the GPU approaches its maximum cooling capacity or TDP limit. Lowering voltage and frequency decreases the transistor switching speed, which directly impacts computational performance. As a result, GPUs experience a reduction in clock frequency, thereby lowering overall performance.



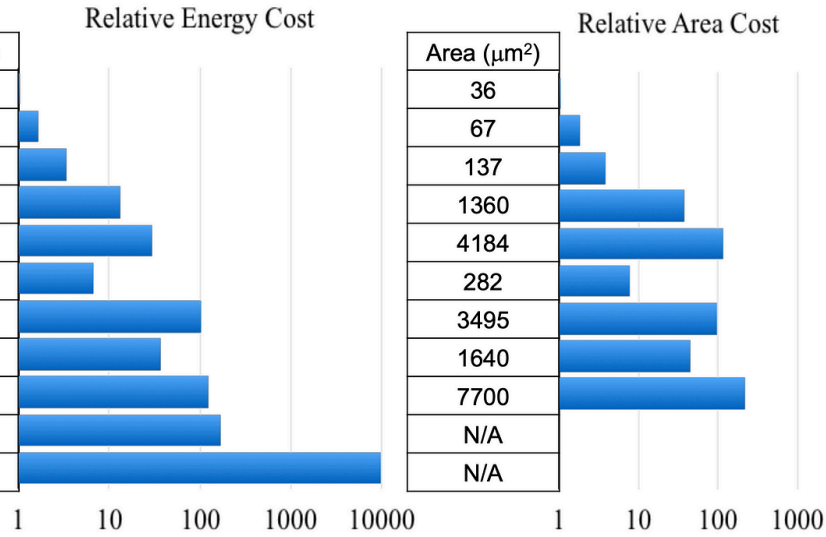
This power throttling mechanism creates a discrepancy between advertised theoretical performance and actual sustained performance. GPU manufacturers often highlight peak theoretical performance metrics based on maximum achievable clock frequencies. However, under realistic workloads, such peak frequencies cannot be sustained due to thermal and power constraints. Empirical data shows that GPUs running intensive matrix multiplication workloads often operate at lower clock speeds than advertised. For instance, while the NVIDIA H100 (SXM) has a theoretical TF32 Tensor Core performance (with sparsity) of 989 TFLOPS, achieving this requires sustaining a boost clock speed of 1.830 GHz. However, under sustained heavy loads like LLM workloads, TDP limits often prevent GPUs from consistently maintaining peak boost clocks. This throttling reduces sustained FLOPS below advertised peak theoretical levels. Consequently, the actual energy efficiency (**FLOPS/watt**) achieved differs from simple peak-based calculations, especially since peak theoretical efficiency often occurs below maximum power limits, not at the thermal ceiling.

Data Movement: The Hidden Power Sink

While power throttling mechanisms like DVFS mitigate thermal constraints, they do not address another often-overlooked challenge, the energy cost of data movement. As LLMs push the boundaries of computational demand, this cost becomes a critical bottleneck. The gap between the energy required for computation and data transport is several orders of magnitude wide, and this gap continues to widen. While GPUs excel at executing arithmetic operations with remarkable energy efficiency, the energy required to fetch data from memory is significantly higher. For example, the B100 (SXM) achieves an impressive **5 TFLOPS/watt** for sparse FP16/BF16 operations, translating to just **0.2 pJ/FLOP**. In contrast, fetching data from HBM

consumes around **6–10 pJ/bit**. Even with on-chip L0-L2 cache hierarchies designed to mitigate these costs, accessing them still incurs significant energy overhead. Instruction cache access alone can consume up to a third of a processor's instruction energy, while data cache access, though beneficial, restricts potential energy savings to just three to four times compared to an ideal system with no memory access overhead.

Operation:	Energy (pJ)	Area (μm²)
8b Add	0.03	36
16b Add	0.05	67
32b Add	0.1	137
16b FP Add	0.4	1360
32b FP Add	0.9	4184
8b Mult	0.2	282
32b Mult	3.1	3495
16b FP Mult	1.1	1640
32b FP Mult	3.7	7700
32b SRAM Read (8KB)	5	N/A
32b DRAM Read	640	N/A



All of this ties back to a deceptively simple principle often expressed in the equation: **$P = E_o \times OPS$,** where **P** is power, **E_o** is energy per operation, and **OPS** represents the operations per second. In the context of LLMs, where we often push trillions of operations per inference or training pass, this relationship becomes a central challenge. Even if each multiply-accumulate (MAC) or fused multiply-add (FMA) is highly efficient on its own, the sheer number of operations multiplies overall power consumption. And when each of those operations triggers data retrieval from caches, HBM, or even more remote memory layers, the effective **E_o** balloon thanks to these heavier data-transfer costs. As a result, simply adding more GPU cores or increasing clock speeds won't effectively boost OPS without also driving up power consumption (P). Instead, any real gains must come from reducing **E_o** at every step, especially the hidden cost of moving data.

In essence, the power equation forces a balancing act: to keep OPS high without blowing past TDP, engineers must relentlessly shrink **E_o**, both for the arithmetic itself and for every step of the memory journey that feeds those operations. This tight linkage between computation and data transport underscores why advanced packaging techniques like CoWoS or chiplet-based approaches only partially alleviate the problem. They may shorten some interconnect distances and improve heat dissipation, but fundamentally you still have to move vast volumes of data across the chip boundary and through on-die caches. Consequently, tackling data movement inefficiencies remains a top priority. Novel architectures that bring compute closer to memory, domain-specific accelerators that minimize off-chip accesses, and software optimizations that restructure models or batching strategies to boost data locality are all crucial.

Conclusion

In this dynamic interplay of bits, FLOPS, and watts, we glimpse not just the future of LLMs, but the fundamental nature of artificial intelligence itself, currently bound to the silicon that gives it form and the thermodynamic principles that shape its evolution. This reality challenges us to rethink how intelligence is realized within computational substrates. Nature, through billions of years of evolution, has optimized biological intelligence to operate at milliwatt-scale power budgets with remarkable efficiency. In contrast, silicon-based intelligence remains orders of magnitude less efficient, a gap highlighted by the growing energy demands of ever-scaling LLMs. As detailed in this two-part series, the pursuit of larger and more capable LLMs—driven by neural scaling laws—now collides with “Walls of Silicon”: limitations imposed by memory capacity, power consumption, and interconnect bandwidth. These constraints, rooted in CMOS density limits, thermal envelopes, and the finite speed of causality, underscore the need for a holistic approach that balances computational throughput, memory bandwidth, and energy efficiency.

Within existing architectures, hardware-aware algorithmic advancements like FlashAttention-3 and sparse attention mechanisms are already improving LLM performance. Meanwhile, alternative model architectures, such as state-space models (SSMs) and diffusion-based LLMs, explore different computational patterns that may present alternative hardware efficiency profiles compared to Transformers, though they still demand significant compute and energy resources and require their own hardware-specific optimizations. Adding further complexity, sophisticated reasoning models such as DeepSeek-R1 and OpenAI’s o1/o3 have increased computational demands during inference, not just during training. Despite innovations like DeepSeek’s MLA and MTP, such models rely on generating intermediate, non-deterministic representations—often called “thinking tokens”—before arriving at the output token. This multi-step CoT reasoning process demands significantly larger KV caches, more computational power, and higher memory bandwidth, ultimately resulting in increased latency and reduced efficiency in parallelization and batching. Furthermore, recent advancements in Agentic AI, often requiring numerous inference calls per task, are driving inference-time computational demands at an unprecedented pace, shifting focus from peak training throughput to inference latency, cost-per-query, and energy efficiency.

As these demands grow exponentially, they push the limits of current semiconductor technologies and energy infrastructures, making it increasingly vital to move beyond traditional von Neumann architectures. To meet these demands, the next generation of compute architectures is emerging—featuring tailored instruction set architectures (ISAs), dataflow-centric execution models, and memory hierarchies, each designed to maximize the efficiency required by modern LLM workloads. Realizing this ambitious future will require deep and sustained collaboration between hardware engineers and AI researchers to pioneer entirely new computing paradigms. This collaboration becomes even more critical with the rise of multimodal and embodied LLMs, which integrate vision, language, and other modalities into unified networks, placing extreme demands on memory systems and interconnect designs, which must efficiently support cross-modal attention and data flow.

Looking further, radical innovations in photonic computing, reversible logic, and biologically inspired neuromorphic systems offer intriguing opportunities to break through the walls of silicon-based hardware. Ultimately, the future of LLM scaling lies not in forcing tomorrow's models onto yesterday's hardware but in co-designing algorithms and computing substrates that work in harmony with—rather than against—the fundamental physics of information processing.

Upcoming

In the next installment of this series, we will explore one of the most promising frontiers in hardware-software co-design: ASIC-based Domain-Specific Architectures (DSAs). Unlike general-purpose accelerators, DSAs address computational challenges at their core rather than working around them. By redesigning computation at every level—from dataflow execution and optimized memory locality to specialized pipeline processing and deterministic execution—DSAs offer a path beyond the constraints of conventional computer architectures. In doing so, they unlock new dimensions of LLM scaling—balancing computational throughput, memory bandwidth and energy efficiency—and bring us closer to realizing the full potential of Artificial General Intelligence.