

Chapter 1. Introduction and AI System Overview

In late 2024, a small startup in China called DeepSeek.AI stunned the AI community by training a *frontier* large language model (LLM) without access to the latest, state-of-the-art NVIDIA GPUs at the time. Due to export restrictions, DeepSeek's engineers could not obtain top-tier NVIDIA Blackwell (B200, B300, etc.) or Hopper (H100, H200, etc.) GPUs, so they resorted to locally available, export-compliant alternatives at the time, including the NVIDIA H800 GPU. They used custom kernels and advanced optimization techniques such as model distillation to squeeze out maximum performance from these less capable GPUs.

Despite these limitations, DeepSeek.AI trained their DeepSeek-R1 model and achieved reasoning capabilities near the performance of leading frontier models that were trained on the most capable NVIDIA chips at the time. This case underscores that practitioners and researchers skilled in AI systems performance engineering can get the most out of their available hardware—no matter the constraints.

For example, DeepSeek's engineers treated communication bandwidth as a *scarce resource*, optimizing every byte over the wire to achieve what many thought impossible on that infrastructure. They scaled out to thousands of these constrained GPUs—connected with limited-bandwidth interconnects—using novel software and algorithmic optimizations to overcome these limitations.

Contrast DeepSeek's approach with the “brute force” path taken by the largest AI frontier labs in the United States and Europe. These labs continue to pursue larger compute clusters and larger models. Model sizes have exploded from millions to billions and now to trillions of parameters. And while each 10 \times increase in scale has unlocked qualitatively new capabilities, they require tremendous cost and resources.

For instance, training OpenAI’s GPT-4 (2023) reportedly cost an estimated ~\$100 million, while training Google’s Gemini Ultra (late 2023) is estimated at a staggering ~\$191 million. This demonstrates the need for resource efficiency going forward as these models scale up in size and cost.

DeepSeek claims that their DeepSeek-R1 model was trained for less than \$6 million in compute—an order of magnitude lower than models like GPT-4 and Gemini Ultra. At the same time, DeepSeek-R1 matches the performance of rival models that cost orders of magnitude more money.

And while there is some doubt as to the validity of the \$6 million claim—and what exactly it includes (e.g., just a single training run) or excludes (e.g., experimentation and the model-development pipeline)—the announcement briefly shocked the US financial markets, including NVIDIA’s stock, which dropped ~17% in a single day based on this news. This was caused by concerns that DeepSeek’s efficiency innovations would somehow require less NVIDIA hardware in the future. While this market reaction was a bit overblown—and NVIDIA stock recovered in subsequent trading sessions—it demonstrates the significant financial impact that breakthroughs in AI efficiency can have on global financial markets.

Beyond model training, DeepSeek boasts significant inference efficiency gains through novel hardware-aware algorithmic improvements to the transformer architecture that powers most modern, frontier LLMs. DeepSeek has clearly demonstrated that clever AI systems performance engineering optimizations can upend the economics of ultrascale AI model training and inference. These optimizations are covered throughout the rest of the book.

The takeaway is a profound realization that, at these scales, every bit of performance squeezed out of our systems could translate to millions, or even billions, of dollars saved. Every bottleneck eliminated can have an outsized impact on training throughput and inference latency. This, in turn, reduces cost and increases overall end-user happiness. In short, AI systems performance engineering isn’t just about speed—it’s about making the previously impossible both possible and affordable.

In [Chapter 1](#), we embark on an in-depth exploration of the AI systems performance engineer—a role that has become pivotal in the era of large-scale artificial intelligence. This chapter serves as a comprehensive guide to

understanding the multifaceted responsibilities and the critical impact of this profession on modern AI systems.

We begin by tracing the evolution of AI workloads, highlighting the transition from traditional computing paradigms to the demands of contemporary AI applications. This context sets the stage for appreciating the necessity of specialized performance engineering in AI.

The chapter then dives into the core competencies required for an AI systems performance engineer. We examine the technical proficiencies essential for the role, including a deep understanding of hardware architectures, software optimization techniques, and system-level integration. Additionally, we discuss the importance of soft skills such as problem solving, communication, and collaboration, which are vital for navigating the interdisciplinary nature of AI projects.

A significant portion of the chapter is dedicated to the practical aspects of the role. We explore how performance engineers analyze system bottlenecks, implement optimization strategies, and ensure the scalability and reliability of AI systems. Real-world scenarios and case studies are presented to illustrate these concepts, providing tangible examples of challenges and solutions encountered in the field.

Furthermore, we discuss the tools and methodologies commonly employed by performance engineers, offering insights into performance testing, monitoring, and benchmarking practices. This includes an overview of industry-standard tools and how they are applied to assess and enhance system performance.

By the end of [Chapter 1](#), readers will have a thorough understanding of the AI systems performance engineer's role, the skills required to excel in this position, and the critical importance of performance engineering in the successful deployment and operation of AI systems. This foundational knowledge sets the stage for the subsequent chapters, where we dive deeper into specific techniques, technologies, and best practices that define excellence in AI performance engineering.

The AI Systems Performance Engineer

The AI systems performance engineer is a specialized role focused on optimizing the performance of AI models *and* the underlying systems they run on. These engineers ensure that AI training and inference pipelines are fast, cost-efficient, and performant—as well as reliable and highly available. As the scale increases, the AI systems performance engineer becomes even more critical.

An AI systems performance engineer commands top salaries—and for very good reasons. Our work has a clear impact on the bottom line. We blend expertise across hardware, software, and algorithms. We must understand low-level OS considerations, memory hierarchies, networking fundamentals, and multiple languages like Python and C++, as well as different AI frameworks and libraries such as PyTorch, OpenAI’s Triton, and NVIDIA’s Compute Unified Device Architecture (CUDA).

On any given day, an AI systems performance engineer might be examining low-level GPU kernel efficiency, optimizing OS thread scheduling, analyzing memory access patterns, increasing network throughput efficiency, or debugging distributed training algorithms. Key responsibilities of an AI systems performance engineer include benchmarking, profiling, debugging, optimizing, scaling, and managing resources efficiently. And while performance engineers may specialize in a combination of hardware, software, and algorithms, the point is that these specializations need to be codesigned together (see [Figure 1-1](#)). As such, it’s good to understand their trade-offs and how they affect one another.

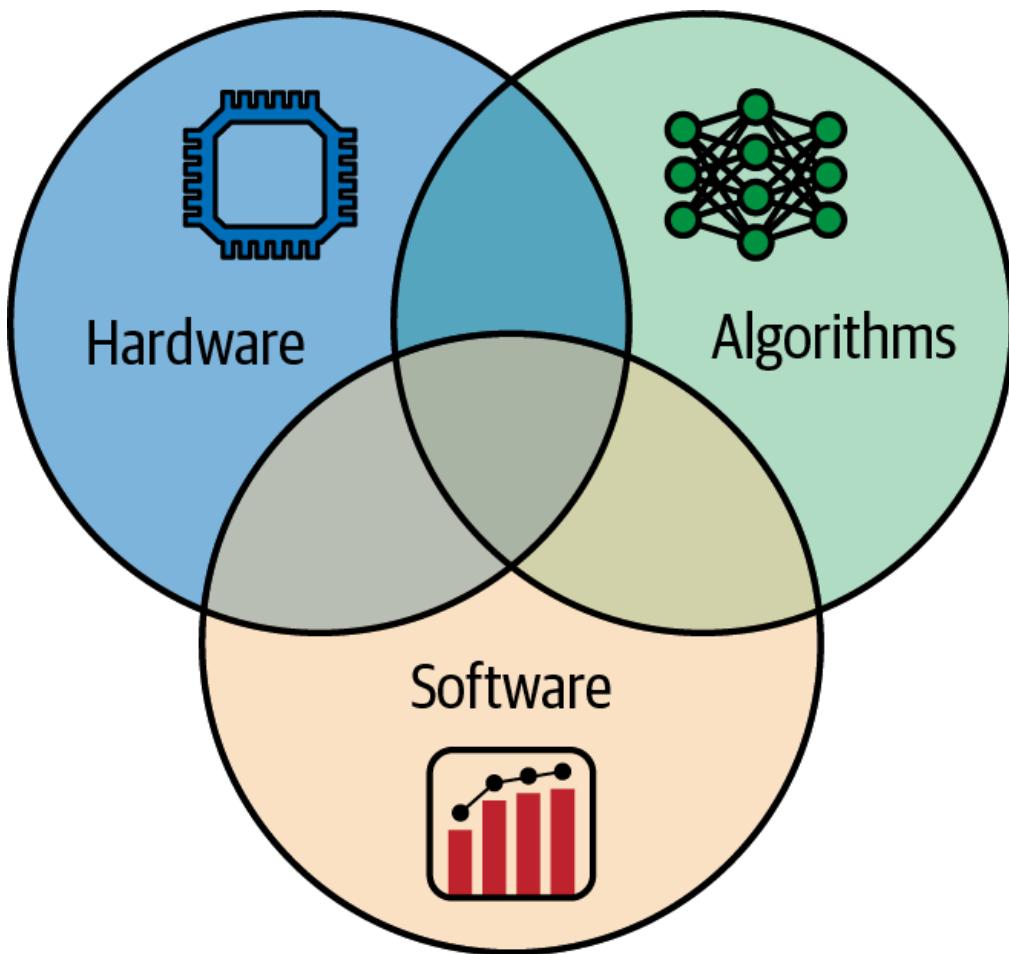


Figure 1-1. Codesigning hardware, software, and algorithms

Benchmarking and Profiling

Benchmarking and profiling involve measuring latency, throughput, memory usage, and other performance metrics for AI models under various workloads, including training and inference. To identify bottlenecks, we must iteratively use NVIDIA Nsight Systems and NVIDIA Nsight Compute together with the PyTorch profiler. Combined, these tools help pinpoint bottlenecks and track performance over time at different levels of the stack as we continue to improve overall performance of our AI system.

It's important to set up automated performance tests to catch regressions, reductions in performance, early in the development cycle.

Debugging and optimizing performance issues requires that we trace performance issues to their root cause, whether it's a suboptimal CUDA kernel, an unnecessary communication overhead, or an imbalance in our training or inference workload.

In one case, we may want to use more efficient matrix operations that take advantage of the latest NVIDIA Transformer Engine (TE) hardware optimized for modern LLMs that use the [transformer](#) architecture. In another case, we can improve the software framework by configuring a higher degree of parallelism for our “embarrassingly parallel” inference workload. In yet another case, we may try to improve the transformer’s [attention algorithm](#) by implementing better memory management and reducing the amount of memory moved in and out of GPU RAM relative to the number of GPU computations required.

Even minor code tweaks can produce major wins. For example, maybe a data preprocessing step written in Python is holding up an entire training pipeline. You can remove that bottleneck by reimplementing the code in C++ or use NVIDIA [cuPyNumeric](#), a drop in NumPy replacement, that can distribute array-operation workloads across both CPUs and GPUs.

Scaling Distributed Training and Inference

Scaling small research workloads to larger production workloads on ultrascale clusters will ensure that as we move from 8 GPUs to 80,000 GPUs, the system will scale with minimal overhead and loss of efficiency. This requires optimizing communication using NVIDIA Collective Communications Library (NCCL, pronounced “nickel”) for distributed collectives like all-reduce commonly seen in training runs. In addition, the NVIDIA Inference Xfer Library (NIXL) provides high throughput, low latency, and point-to-point data movement across GPU memory and storage tiers for distributed inference. This communication can be between GPUs either on a single node or across thousands of nodes. They also optimize communication and collective aggregation operations like all-reduce, all-to-all, and all-gather, which are used extensively during model training and inference.

You may want to place data cleverly across nodes using data, tensor, and pipeline parallelism. Or you may need to redesign the workload to use tensor parallelism or pipeline parallelism because the model is so large that it doesn’t fit onto a single GPU. Perhaps you are using a *mixture of experts* (MoE) model and can take advantage of expert parallelism.

Managing Resources Efficiently

It's important to optimize how models utilize resources like CPU cores, GPU memory, interconnect bandwidth, and storage I/O. This can involve many efforts such as ensuring GPUs are fed with data at full throttle, pinning threads on specific CPU cores, reducing context-switch overhead, orchestrating memory usage, and avoiding out-of-memory (OOM) errors on GPUs when training and inferencing with large models. Techniques like GPU virtualization (e.g., NVIDIA's Multi-Instance GPU [MIG]) can partition GPU resources for better overall utilization when full GPU power isn't needed for a job.

Cross-Team Collaboration

Cross-team collaboration is absolutely critical for AI systems performance engineers. It's important to work hand in hand with researchers, data scientists, and application developers—as well as infrastructure teams, including networking and storage.

Improving performance might require modifying model code, which involves coordination with researchers. Or you may want to deploy a new GPU driver to improve efficiency, which requires the infrastructure team.

Often, performance improvements span multiple teams. For instance, updating the CUDA driver or CUDA version for bug fixes and efficiency will involve careful coordination with DevOps, infrastructure, and support teams. And improving model code for performance involves close work with researchers. The performance engineer sits at the intersection of these multidisciplinary domains and speaks the language of AI, computer science, and systems engineering.

Transparency and Reproducibility

In performance engineering, it's vital to measure everything and trust data, not assumptions. By publishing your work, others can learn, reproduce, and build upon your findings.

One notable aspect of DeepSeek's story is how openly they shared their infrastructure optimizations. During DeepSeek's [Open-Source Week](#) in

February 2025, they released a suite of open source GitHub repositories, including [FlashMLA](#), [DeepGEMM](#), [DeepEP](#), [expert parallelism load balancer](#) (EPLB), [DualPipe](#), and [Fire-Flyer File System](#) (3FS). Each project was production-tested and aimed at squeezing the most performance from their hardware. These projects are described in the [DeepSeek-V3 Technical Report](#).

FlashMLA is their optimized attention kernel written in CUDA C++. DeepGEMM provides an FP8-optimized matrix multiplication library that reportedly outperforms many vendor kernels on both dense and sparse operations. Deep Experts Parallelism (DeepEP) is their highly tuned communication library for mixture-of-experts (MoE) models. EPLB implements a redundant expert strategy that duplicates heavily loaded experts to handle the additional load. DualPipe is a bidirectional pipeline parallelism algorithm that overlaps the forward/backward computation and communication phases to reduce pipeline bubbles. And 3FS is their high-performance distributed filesystem, reminding us that every layer needs to be optimized—including the filesystem—to get the most performance out of our AI system.

By open sourcing these [projects](#) on GitHub during their February 2025 “Open-Source Week,” DeepSeek not only demonstrated the credibility of their claims by allowing others to reproduce their results but also contributed back to the community. This transparency allows other developers to benchmark, reproduce, and learn from their methods, including overlapping communication with DeepEP/DualPipe pipeline parallelism and saturating NVMe SSD/RDMA bandwidth with 3FS.

Open efforts like DeepSeek’s [Open Infra Index](#) provide valuable baselines and tools. They provide real-world performance measurements on various AI hardware setups and encourage apples-to-apples comparisons and reproducibility. Similarly, the [MLPerf](#) open benchmark suite provides a standard for reproducibly comparing training and inference performance across hardware and software setups.

Industry benchmarks such as MLPerf have quantified these kinds of codesigned optimizations across hardware generations. In MLPerf [Training](#) v5.0 (2025), a Blackwell-based NVIDIA GB200 NVL72 system produced up to 2.6× higher training throughput per GPU over an equivalent Hopper system, as shown in [Figure 1-2](#). In MLPerf [Inference](#) v5.0 (2025), the Blackwell NVL72 achieved about 3.4× higher inference throughput per GPU

over an equivalent Hopper cluster due to higher per-GPU performance and the much larger NVLink domain. These results are shown in [Figure 1-3](#).

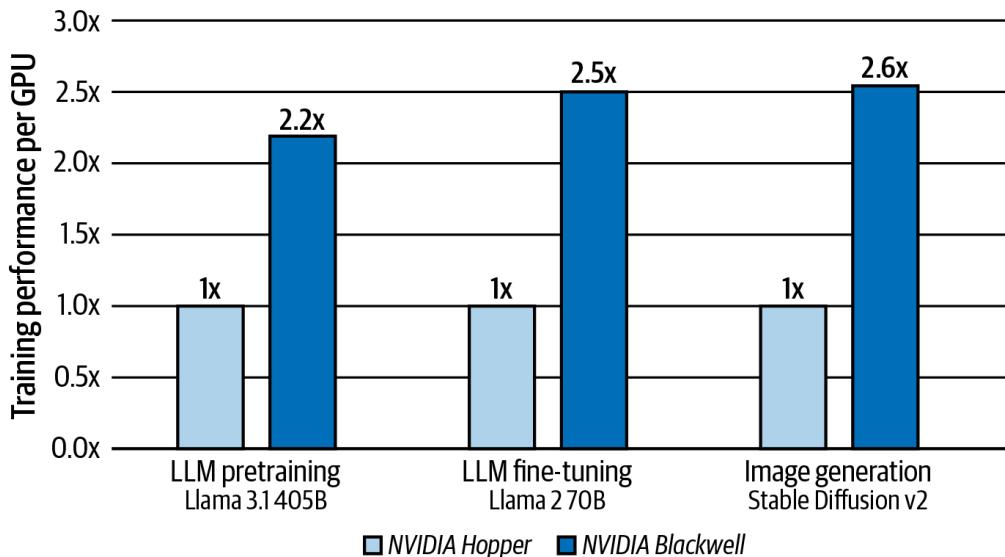


Figure 1-2. NVIDIA GB200 NVL72 MLPerf Training v5.0 per-GPU throughput improvement over an equivalent NVIDIA Hopper cluster (source: <https://oreil.ly/AoII8>)

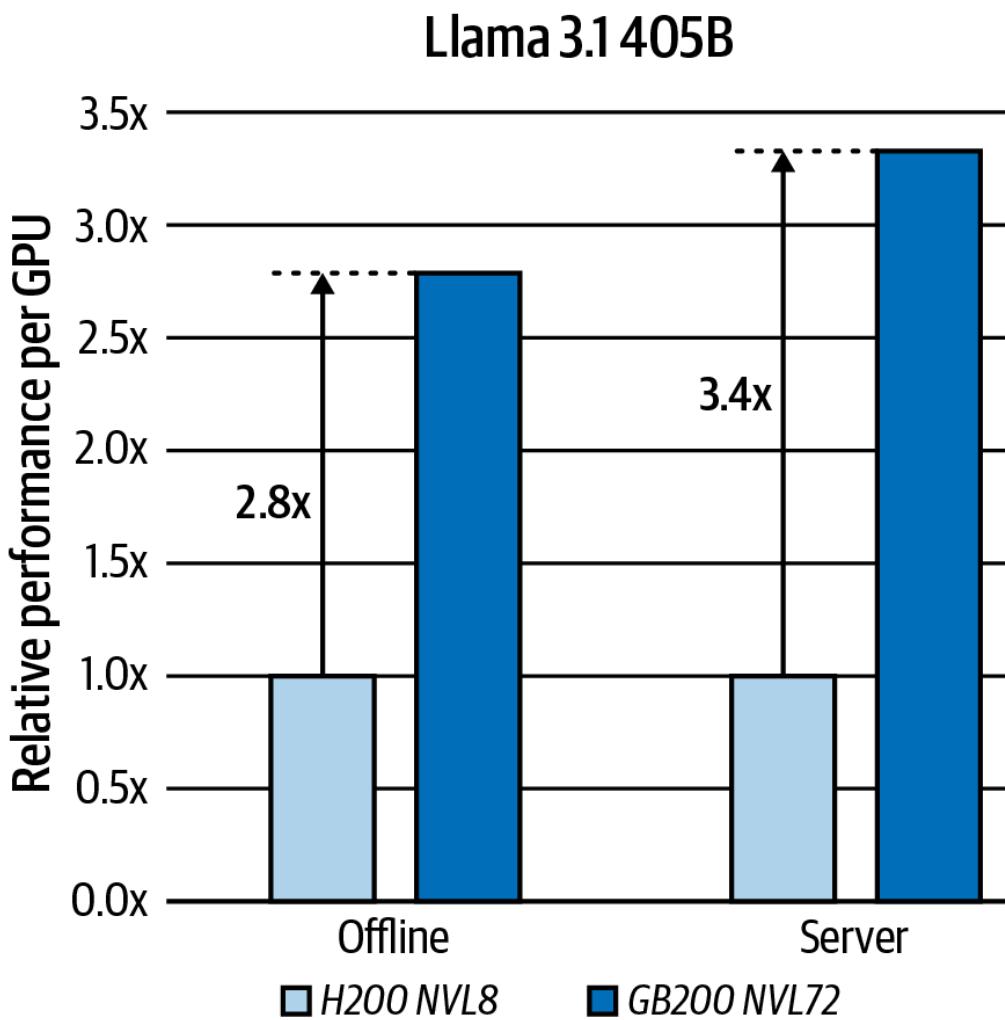


Figure 1-3. NVIDIA GB200 NVL72 MLPerf Inference v5.0 per-GPU throughput improvement over an equivalent NVIDIA Hopper cluster (source: <https://oreil.ly/V-ze>)

In later chapters, we'll reference some of these open benchmarks to support various performance tuning concepts. For instance, when discussing GPU kernel optimizations, we will reference DeepSeek's published profiles

showing how their custom kernels achieved near-peak memory bandwidth utilization on NVIDIA GPUs.

When we cite MLPerf results, remember that MLPerf cautions that per-GPU results are not a primary metric across platforms. Use system-level, end-to-end throughput as the correct basis for comparison. Consider per-GPU numbers only as component-level indicators.

Experimental transparency and reproducibility are critical in moving the field of AI performance engineering forward. It's easy to fall into the trap of anecdotal "vibe" optimizations ("We did X and things felt faster"). Instead, I'm advocating for a rigorous, scientific approach that develops hypotheses, measures the results with reproducible benchmarks, adjusts to improve the results, reruns the benchmarks, and shares all of the results at every step.

DeepSeek Scales to ~680-Billion Parameter Models Despite US Export Hardware Restrictions in China

Sometimes systems-performance innovations are born from necessity. As mentioned, DeepSeek found itself constrained to using only NVIDIA's H800 GPUs due to US export restrictions. The H800 is an export-compliant variant of the Hopper GPU. Compared to the H100, it reduces NVLink interconnect bandwidth and FP64 performance while keeping HBM capacity and bandwidth largely similar.

For context, an NVIDIA H100 provides about 900 GB/s NVLink interconnect bandwidth per GPU, while H800 provides about 400 GB/s bandwidth per GPU. This makes inter-GPU data transfers slower and ultimately limits multi-GPU scalability. Additionally, while the H100 offers 3.35 TB/s of memory bandwidth, the H800's limited throughput means that data transfers are much slower. This threatens to bottleneck distributed training jobs and limit scalability efficiency.

DeepSeek set out to train a massive ~680-billion-parameter MoE language model, called DeepSeek-V3, in this heavily constrained environment. This

MoE model uses only about 37 billion active parameters per input token—rather than all ~680 billion at once.

With this architecture, only a fraction of the model is activated at any given time. This helps manage computational load—even with the compact H800 setup. Specifically, for each token, DeepSeek-V3 uses 1 shared expert plus 8 router-selected experts (out of 256 experts) for a total of 9 active experts per token, as shown in [Figure 1-4](#).

We'll cover MoEs in more detail in the upcoming chapters. But just know that, to work around the environment limitations, DeepSeek implemented a novel DualPipe parallelism algorithm that carefully overlaps computation and communication to mask the H800's inherent weaknesses.

By designing custom CUDA kernels to bypass some of the default NCCL communication collectives, DeepSeek was able to coordinate data transfers in tandem with ongoing computations. This keeps the GPUs efficiently utilized despite their reduced interconnect bandwidth. This kind of communication/computation overlap is a theme we will revisit throughout the rest of the book.

This innovative engineering paid off, as DeepSeek-V3 was trained to completion at a fraction of the GPU time (and cost) of similarly sized frontier models from OpenAI, Meta, DeepMind, and others. This is a fraction of the resources that many assumed were necessary to train a model of this scale using a more capable cluster.

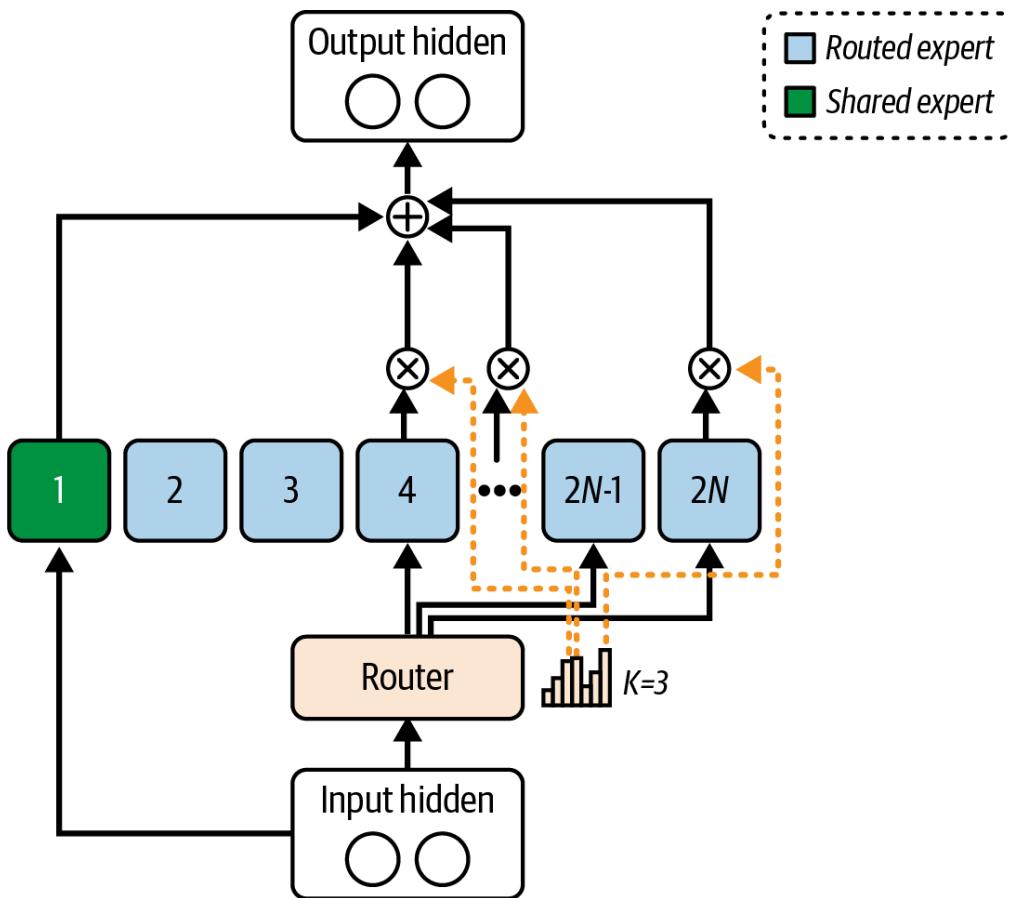


Figure 1-4. DeepSeek-V3’s expert routing

DeepSeek [reports](#) that V3’s performance approaches GPT-4’s on several standardized benchmarks, including language understanding, reading comprehension, and reasoning. It even matches or slightly exceeds GPT-4 on some tests. These comparisons were based on standardized tests used across the industry. This implies that an open MoE model can rival the best closed models—despite using less capable hardware.

Building on the DeepSeek-V3 model, the team then created DeepSeek-R1, which is its specialized reasoning model built similarly to OpenAI’s o1 and o3 [series](#) of reasoning models. Instead of relying heavily on costly human feedback loops for fine-tuning, DeepSeek pioneered a “cold start” strategy that used minimal supervised data, instead emphasizing reinforcement learning techniques to embed chain-of-thought reasoning directly into R1. This approach reduced training cost and time and underscored that smart software and algorithm design can overcome hardware bottlenecks.

The lessons learned are that large, sparsely activated MoE models can be effectively scaled even on limited memory and compute budgets. Novel training schedules and low-level communication/computation overlap optimizations can overcome hardware limitations, as demonstrated by the enormous return on investment (ROI) of DeepSeek’s efforts.

The ROI is clear, as DeepSeek’s unconventional approach brought about huge efficiencies and created a series of powerful models at much lower training cost and time. By extracting every ounce of performance from NVIDIA’s H800 GPU—even under the constraints of reduced communication bandwidth—the team delivered GPT-4-level model performance for millions of dollars less. Additionally, DeepSeek saved even more money by not requiring as much human-labeled data during R1’s fine-tuning stage for reasoning.

In short, smart software and algorithm design overcame brute-force hardware limitations. This enabled DeepSeek to develop large-scale AI models on a tight cost and hardware budget.

Toward 100-Trillion-Parameter Models

100-trillion-parameter models are an aspirational milestone for AI and are often compared against the estimated 100 trillion synaptic connections in the human neocortex. Each synaptic connection is equivalent to a model parameter. Achieving a model of this size is theoretically possible, but it demands an extraordinary amount of resources—and money. Scaling to 100-trillion-parameter models by brute force would be impractical for all but the absolute wealthiest organizations.

As a rough order of magnitude, a dense 100-trillion-parameter model trained on about 29 trillion tokens would require on the order of 1.2×10^{29} floating point operations (FLOPS). Larger token counts scale this linearly, while using sparse (MoE) models can reduce the effective compute. However, even with sparsity, this demonstrates that mere brute force is not enough to satisfy the compute needs of ultrascale models. New approaches are needed to make 100-trillion-parameter training feasible in a reasonable amount of time. Achieving this kind of scale requires breakthrough efficiencies in hardware, software, and algorithms codesign—rather than just continuing to scale out current methods.

While the optimizations discussed in this book can be applied to smaller models and cluster sizes, I will continue to revisit the 100-trillion-parameter model to enforce the idea that we can’t just throw hardware at the scaling problem. We need clever software and algorithmic innovations that are codesigned with hardware, including compute, networking, memory, and storage.

Such explosive growth in training cost is driving a search for new AI systems and software engineering techniques to increase performance, reduce cost, and make extreme-scale AI feasible with limited compute resources, power constraints, and money. Researchers are always exploring novel techniques to reduce the effective compute requirements.

One prominent idea is to use sparsity and, specifically, MoE models. Sparse models like MoEs are in contrast to traditional dense models like the common GPT-series large language models (LLMs) made popular by OpenAI. In fact, there is public speculation that some frontier models like OpenAI's proprietary GPT-series and o-series reasoning models are based on the MoE architecture.

Sparse models like MoEs activate only parts of the model for each input token. By routing each input token through only a subset of its many internal "experts," the FLOPS per token stays roughly constant even as total parameters grow. Such sparse models prove that scaling to multi-trillion-parameter models is done without an equivalent explosion in computation cost. Additionally, since these models use fewer active parameters during inference, request-response latencies are typically much lower for MoE models compared to their dense equivalents. These are crucial insights toward training and serving 100-trillion-parameter-scale models.

DeepSeek-V3 (base model) and -R1 (reinforcement-learning-based reasoning variant) are great examples of MoE efficiency. They contain ~680 billion total parameters with about 37 billion active per input token. DeepSeek's technical report and public write-ups describe an MoE with 1 shared expert and 8 selected experts out of 256 per token, which yields about 9 active experts and roughly 37 billion active parameters per token. This makes DeepSeek-V3 and -R1 much more resource-efficient than similarly sized dense large language models. Another example is Google's [Switch Transformer](#) MoE from 2021. This 1.6-trillion-parameter MoE model achieved the same accuracy as a dense model with only a fraction of the computation. It was trained 7× faster than a comparable dense approach.

In addition to massive compute requirements, memory is also a major bottleneck. For example, a 100-trillion-parameter model would require approximately 182 TB of GPU memory ($182 \text{ TB} = 100 \text{ trillion parameters} \times 16 \text{ bits per weight} \times 8 \text{ bits per byte}$) to load the model if each parameter is stored in 16-bit (2-byte) precision. This is 3 orders of magnitude ($1,000\times$)

compared to the 192 (180 usable) GB of GPU RAM on a single NVIDIA Blackwell B200 GPU.

To simply load the 100-trillion model weights would require close to 1,000 Blackwell B200 GPUs (192 [180 usable] GB each)—or 700 Blackwell Ultra B300 GPUs (~288 GB each). And this estimate is just to load the model and does not include activation memory, optimizer states, and inference input. These would further increase the total memory required.

For context, a typical B200 GPU compute node contains just 8 B200 GPUs. To use these, you would require ~125 GPU nodes just to load the model with a B200, and ~86 GPU nodes to load the model with a cluster of Ultra B300 GPU compute nodes (8 B300 GPUs per node).

Additionally, loading training data also becomes extremely difficult since feeding such a model with data fast enough to keep all of those GPUs busy is nontrivial. In particular, communication overhead between the GPUs grows significantly as the 100-trillion-parameter model is partitioned across 1,000 B200 GPUs or 700 B300 GPUs. Training a single model could consume millions of GPU-hours and megawatt-hours of energy. This is an enormous amount of cost—and energy consumption—to scale out large enough to train and serve a 100-trillion-parameter model.

The era of 100-trillion-parameter AI will force us to completely rethink system design to make training and deployment practical at this scale. Hardware, algorithms, and software all need to coevolve to meet this new frontier.

NVIDIA’s “AI Supercomputer in a Rack”

To meet the challenges of ultrascale computing, NVIDIA has built a new class of AI supercomputers specifically aimed at trillion-parameter-scale workloads. Some examples include the NVIDIA Grace Blackwell GB200 NVL72 in 2024 and the GB300 NVL72 Ultra, which uses Blackwell Ultra GPUs with 288 GB HBM3e per GPU and maintains a 72-GPU NVLink domain with about 130 terabytes per second aggregate bandwidth.

Vera Rubin VR200 (2026) and Feynman (2028) systems continue this trend of exascale supercomputers condensed into a single data center rack. In fact,

NVIDIA refers to these rack systems like NVL72 as “AI supercomputers in a rack”—and for good reason.

Each GB200/GB300 NVL72 rack integrates 36 Grace Blackwells connected through NVLink with NVSwitch, providing the rack scale switching fabric.

Each Grace Blackwell Superchip is a combination of one NVIDIA Grace CPU (with 72 CPU cores) and two NVIDIA Blackwell GPUs for a total of 36 Grace CPUs and 72 Blackwell GPUs—hence, the “72” in the name “NVL72.”

If you haven’t guessed already, the NVL in NVL72 stands for *NVLink*. This helps remind you that the GPUs in this system are interconnected using NVLink. Just in case we forget!

Each Grace Blackwell board connects to other boards using NVSwitch, the on-rack NVLink switch network. This way, all 72 GPUs can communicate with one another at full NVLink 5 bandwidth of 1.8 TB/s bidirectional per GPU (18×100 GB/s links). Across the rack, the NVLink Switch System provides about 130 TB/s aggregate GPU to GPU bandwidth within one NVL72 domain.

Effectively, the NVL72’s internal fabric unifies all 72 Grace Blackwell GPUs into one high-speed cluster. As such, frameworks like PyTorch for training and vLLM for inference can use the single NVLink domain for efficient data tensor pipeline and expert parallelism. CUDA Unified Memory can migrate or remotely access pages across NVLink when needed. However, remote memory has distinct latency and bandwidth that should be treated as nonuniform. When relying on managed allocations across Grace and Blackwell, prefer explicit prefetch using `cudaMemPrefetchAsync` and `cudaMemAdvise` to reduce page-fault stalls.

More details are provided on the compute, memory, and interconnect hardware details of this supercomputer in upcoming chapters. For now, let’s analyze the overall performance specifications of this AI supercomputer as a whole in the context of modern generative AI models.

A full GB200 NVL72 rack can theoretically reach about 1.44 exaFLOPS for FP4 with 2 to 1 structured sparsity and about 720 petaFLOPS for FP8 with 2 to 1 structured sparsity. It provides roughly 13.5 TB (13,824 GB = 192 GB per

$\text{GPU} \times 72$ GPUs) of HBM3e across the 72 GPUs and around 30 TB total when counting Grace CPU memory in the same NVLink domain.

NVLink provides pooled access across the 72 GPUs, and CUDA Unified Memory can migrate or remotely access pages over NVLink, but remote access has different performance, and the rack does not behave as a single uniform memory device.

In short, the GB200 NVL72 is a self-contained 72 GPU 1.44 exaFLOPS 30 TB memory system with about 120 to 132 kW rack power, depending on vendor configuration and cooling. It's truly an AI supercomputer that can train and serve multi-trillion-parameter models in a single rack.

By combining these racks to form ultrascale clusters, you can support massive multi-trillion-parameter models. Even better, you can provision these racks and rack clusters with a few clicks (and quite a few dollars!) using your favorite cloud provider, including Amazon Web Services (AWS), Google Cloud Platform (GCP), Microsoft Azure, CoreWeave, Lambda Labs, and many others.

While this book focuses heavily on the Grace Blackwell generation of NVIDIA chips, the optimization principles discussed are derived from many previous generations of NVIDIA hardware. And these optimizations will continue to apply and evolve to many future NVIDIA chip generations to come, including Vera Rubin (2026), Feynman (2028), and beyond. This roadmap continues the pattern of doubling performance, memory, and integration with each new GPU generation.

Throughout the book, you will learn how each generation's innovation in compute, memory, networking, and storage will contribute to more AI scaling in the form of ultrascale clusters, multi-trillion-parameter models, high-throughput training jobs, and extreme-low-latency model inference servers. These innovations are fueled by hardware-aware algorithms that enforce the principles of mechanical sympathy and hardware-software codesign discussed in the next section.

Mechanical Sympathy: Hardware-

Software Codesign

Mechanical sympathy is a term originally coined by software engineer Martin Thompson, who was drawing an analogy to the British Formula One champion [Jackie Stewart](#), who intimately understood his cars' mechanical details. In computing, it refers to writing software that is deeply aware of the hardware it runs on. In the AI context, it means codesigning algorithms hand in hand with hardware capabilities to maximize performance.

Real-world experience has shown that even minor tweaks in GPU kernels or memory access patterns can produce outsized gains. A classic example is [FlashAttention](#), a novel algorithm that reimplements the transformer attention mechanism in a hardware-aware way.

FlashAttention “tiles” GPU computations, which minimizes the number of reads and writes issued to the GPU’s memory. FlashAttention significantly reduces memory movement and speeds up attention computation. Replacing the default transformer attention mechanism/algorithm with FlashAttention produces a $2\times\text{--}4\times$ speedup in training and inference for long sequences while also reducing the overall memory footprint.

This kind of change reduces what used to be a major bottleneck (attention) down to a fraction of overall runtime. FlashAttention became the default in many libraries almost overnight because it let models handle longer sequences faster and more efficiently. Since FlashAttention, many new attention algorithms have emerged, including DeepSeek’s Multi-Head Latent Attention (MLA).

DeepSeek’s MLA algorithm—implemented as an NVIDIA GPU kernel and open sourced in 2025—is another example of hardware-software codesign, or mechanical sympathy. Similar to FlashAttention, MLA restructures the attention computations to better utilize NVIDIA’s memory hierarchy and dedicated GPU “Tensor Cores.” These optimizations allowed MLA to exploit the constrained H800 GPUs’ architecture, achieving higher throughput at a fraction of the cost—surpassing even FlashAttention’s performance on those same H800 systems.

This entire book is effectively a study in mechanical sympathy. We will see countless cases where new hardware features—or hardware constraints, as in

the case of DeepSeek—inspire novel new software and algorithmic techniques. Conversely, we'll see where new software algorithms encourage new hardware innovations.

For instance, the rise of transformer models and reduced-precision quantization (e.g., FP8/FP4) led NVIDIA to add specialized hardware like the Transformer Engine and dedicated reduced-precision Tensor Cores for faster matrix-math computation units. These hardware innovations, in turn, enable researchers to explore novel numeric optimizers and neural-network architectures. This, then, pushes hardware designers even further, which then unlocks even newer algorithms, etc. It's a virtuous cycle!

Modern GPUs use the latest Transformer Engine with FP4 support and microscaling. Combined with the latest generation NVLink, these features increase attention throughput primarily through faster Tensor Core math, higher memory bandwidth, and improved Special Function Units (SFUs.) For instance, the [exponential computation unit](#) is specifically designed to accelerate the softmax operation used heavily by the transformer's attention algorithm—a key part of today's LLM models. With the popularity of transformers, softmax latency had become a bottleneck on previous GPUs, given that it's critical to the transformer attention mechanism.

By improving special function throughput and fast math pipelines with modern GPUs, NVIDIA demonstrates mechanical sympathy and hardware-software-algorithm codesign in its best and purest form. They work directly with researchers and practitioners to address and reduce the bottlenecks of modern LLM algorithms (e.g., attention) running on their hardware.

This tight interplay—GPUs and AI algorithms coevolving—is the heart of mechanical sympathy in AI. These codesign innovations can happen only with close collaboration between the hardware companies (e.g., NVIDIA and Advanced RISC Machine [ARM]), AI research labs (e.g., OpenAI and Anthropic), and AI systems performance engineers (e.g., us!).

Measuring “Goodput” Useful Throughput

When operating clusters of hundreds, thousands, or millions of GPUs, it’s important to understand how much of the theoretical hardware capability is actually performing useful work. Traditional throughput metrics like FLOPS and device utilization are misleadingly high, as much of the time is likely spent on stalled communication, idling computation, or failed job restarts. This is where the concept of “goodput” comes in—as described by Meta as “effective training-time ratio” in a [2025 paper](#).

NVIDIA calls the theoretical hardware maximum the *speed of light*, as you may have seen in NVIDIA blogs, documentation, webinars, and conference talks.

In simple terms, goodput measures the throughput of useful work completed (number of tokens processed or inference requests completed) per unit time—discounting everything that doesn’t directly contribute to model training or inference. It’s effectively the end-to-end efficiency of the system from the perspective of productive model training or inference. Goodput can then be normalized by the cluster’s maximum possible throughput to produce a percent efficiency value.

For example, suppose a node with 8 GPUs can process 100,000 tokens in 10 seconds. In this case, its goodput is 10,000 tokens per second. If each GPU in the node can achieve a peak theoretical throughput of 1,500 tokens per second, or 12,000 tokens per second across all 8 GPUs, the node’s efficiency is 83.3% ($0.833 = 10,000 \text{ achieved throughput} / 12,000 \text{ peak throughput}$).

Meta’s AI infrastructure team highlighted the importance of goodput in the “Revisiting Reliability” paper. The paper introduces the *effective training time ratio metric* and shows how preemptions resource fragmentation and failures reduce realized training time even when headline utilization is high. In this paper, Meta’s team analyzes how preemptions, hardware faults, and network congestion reduce realized throughput.

In other words, while the cluster appeared to be 100% utilized, 70%–75% of the compute was lost due to overheads like communication delays, suboptimal parallelization, data delays, or failure recovery. Additionally, Meta’s analysis

showed that, at scale, issues like job preemptions, network hotspots, and unrecoverable faults were major contributors to lost goodput.

For example, imagine a training job that could theoretically process 1,000 samples/second on ideal hardware, but due to a poor input pipeline and excessive synchronization, it achieves only 300 samples/second of actual training throughput. We'd say that the job is running at 30% goodput. The remaining 70% capacity is essentially wasted.

Identifying these gaps and closing them is a core part of our work. For instance, if GPUs are waiting on data loading from storage, we might introduce caching or async prefetch. If they're idling during the gradient synchronization step of our model training process, we likely want to overlap the GPU computation (e.g., calculating the gradients) with the communication between GPUs (e.g., synchronizing the gradients). Our goal is to turn wasted, inefficient cycles into useful work.

This gap between theoretical and realized performance is the value proposition of the AI systems performance engineer role. Our mission is to drive that goodput number as high as possible—ideally increasing it closer to 100%—by attacking inefficiencies and reducing cost at every level of the stack, including hardware, software, and algorithms.

Investments in AI systems performance engineers consistently generate returns well above cost. Consider a performance engineer helping to achieve a 20% boost in cluster efficiency. This can reduce hardware costs by millions of dollars in large-scale AI environments.

By focusing on goodput, we are optimizing what truly matters—the amount of useful training done per dollar of cost and per joule of power. Goodput is the ultimate metric of success—more so than raw FLOPS or device utilization—because it encapsulates how well hardware, software, and algorithms are harmonized toward the end goal of training AI models faster and cheaper.

Improving goodput requires a deep understanding of the interactions between the hardware (e.g., CPUs, GPUs, network topologies, memory hierarchies, storage layouts), software (e.g., operating system configurations, paged memory, I/O utilization), and algorithms (e.g., transformer architecture

variants, attention mechanism alternatives, and different caching and batching strategies).

This broad and deep understanding of multiple disciplines—including hardware, software, and algorithms—is why AI systems performance engineers are so scarce today. This is also why I’m writing this book! Next is the roadmap and methodology that maps out the rest of this book.

Book Roadmap and Methodology

How will we approach the optimization of 100-trillion-parameter AI systems? This book is organized to take you from the hardware fundamentals up through the software stack and algorithmic techniques—with an emphasis on hands-on analysis at each level. Here is a breakdown of the rest of the book.

[Chapter 2](#) provides an in-depth look at NVIDIA AI system hardware, including the GB200/GB300 NVL72 “AI supercomputer in a rack,” which combines Grace Blackwell Superchip design with the NVLink network to create performance/power characteristics of an AI supercomputer.

Chapters [3–5](#) will then cover OS-level, networking, and storage optimizations for GPU-based AI systems. These optimizations include CPU and memory pinning, as well as Docker container and Kubernetes orchestration considerations, including network I/O and storage configurations for GPU environments.

Chapters [6–12](#) discuss NVIDIA CUDA programming fundamentals and CUDA-kernel optimizations that are essential for developing novel hardware-aware algorithms. Such popular algorithms include FlashAttention and DeepSeek’s MLA. These algorithms target the resource-intensive attention mechanism of the transformer architecture, which dominates today’s generative AI workloads.

With NVIDIA hardware and CUDA software context in hand, we’ll dive into distributed communication optimizations, including training and serving ultralarge models efficiently. We’ll examine strategies to minimize communication, such as overlapping computation with communication—a pattern that applies to many layers of the AI system stack.

Chapters [13](#) and [14](#) discuss PyTorch-specific optimizations, including the PyTorch compiler stack and OpenAI’s Python-based Triton language and compiler for custom GPU kernels. These compilers lower the barrier for developing novel CUDA kernels, as they don’t require a deep understanding of C++ typically required to develop CUDA kernels. These chapters also discuss distributed parallelization techniques for model training, including data parallelism (DP), fully sharded data parallelism (FSDP), tensor parallelism (TP), pipeline parallelism (PP), context parallelism (CP), and mixture-of-experts (MoE). We will show how multi-trillion-parameter models are split and trained across many GPUs efficiently. And we will discuss techniques for memory optimization during ultrascale model training, including activation checkpointing, sharding optimizer states, and offloading to larger CPU memory. These techniques are vital when model sizes exceed the physical GPU hardware limits.

Chapters [15–19](#) focus on software and algorithmic innovations for high-throughput, low-latency model inference and agentic AI systems. This includes disaggregated prefill and decode, which is now supported in NVIDIA Dynamo and community stacks with key-value (KV) cache movement over UCX with GPUDirect RDMA, NCCL point-to-point, or framework-provided transports. We also discuss the widely used model serving engines, including vLLM, SGLang, and NVIDIA TensorRT LLM. We then cover the NVIDIA Dynamo distributed inference framework, which integrates with these engines and includes NIXL for low-latency KV cache transfer in disaggregated prefill decode setups.

We’ll also look at leveraging the Grace CPU in the NVL72 for preprocessing, co-running smaller “draft” models for high-performance inference algorithms such as speculative decoding, and efficient request routing and batching to maximize overall throughput of the inference system. We will also explore model compression and acceleration techniques such as 4-bit quantization, knowledge distillation to teach smaller “student” models from wiser “teacher” models, sparsity and pruning, and the use of specialized TensorRT kernels. There is a heavy focus on disaggregated prefill-decode and adaptive techniques to dynamically tune a system at runtime.

[Chapter 20](#) describes modern efforts to use AI-assisted tools to optimize kernel and AI system performance. The chapter also includes case studies on training and serving multi-billion- and multi-trillion-parameter models efficiently. It also covers emerging trends for self-improving AI system

optimizations and agents. This helps to paint a picture of where 100-trillion-parameter-scale AI systems are headed—and how to position ourselves for the future of AI systems performance engineering.

The [Appendix](#) provides a checklist of common performance-optimization and cost-saving tips and tricks to apply to your own AI system. This is a summary of actionable optimizations and efficiency gains discussed throughout this book.

In short, this book implements a hands-on and empirical methodology to apply performance optimizations. We will frequently analyze actual runs, case studies, benchmark results, and profiling data to understand bottlenecks and verify improvements. By the end of the book, you should grasp the principles of optimizing ultralarge AI systems—as well as gain some practical experience with tools to apply those optimizations on ultrascale, multi-GPU, multinode, and multirack AI systems like the NVIDIA GB200/GB300 NVL72 AI supercomputer in a rack—or similar AI systems now and in the future.

Key Takeaways

The following qualities collectively define the role of the AI systems performance engineer, whose expertise in merging deep technical knowledge with strategic, profile-driven optimizations transforms raw hardware into cost-effective, high-performance AI solutions:

Measure goodput

Look beyond raw FLOPS or utilization. Instead, measure the ratio of time the GPU spends performing useful work (e.g., forward/backprop computations) versus waiting on data or other overhead. Use NVIDIA Nsight Systems/Compute or PyTorch profiler to measure this ratio. Strive to improve this ratio, as goodput focuses on effective, useful GPU utilization.

Prefer skillful engineering optimizations instead of brute-force spending

More hardware isn't a silver bullet. Clever software and system optimizations can bridge the gap when hardware is limited, enabling results that would otherwise require far more expensive infrastructure. We saw this with DeepSeek's achievement. By optimizing

communication and hardware usage, their engineers trained a frontier model on restricted H800 GPUs (with limited interconnect bandwidth) at a fraction of the cost. The DeepSeek model matched the performance of frontier models trained on far more powerful hardware. In other words, skillful engineering outperformed brute-force spending.

Look for order-of-magnitude impact with incremental optimizations

At scale, even a small-percentage efficiency gain can save millions of dollars. Said differently, small inefficiencies such as redundant computations and slow data pipelines can silently increase costs as the system scales.

Approach performance tuning with a profile-driven mindset

Use data and profiling tools to guide optimizations. Use profilers to identify the true bottlenecks—whether it's compute utilization, memory bandwidth, memory latency, cache misses, or communication/network delays. Then apply targeted optimizations for that bottleneck.

Maintain a holistic view

Improving AI systems performance spans hardware, including the GPU, CPU, memory, and network—as well as software such as algorithms and libraries. A weakness in any layer can bottleneck the whole. The best performance engineers consider hardware-software codesign: sometimes algorithm changes can alleviate hardware limits, and sometimes new hardware features enable new algorithms.

Stay informed on the latest hardware, software, and algorithms

Modern AI hardware and software are evolving rapidly. New capabilities such as unified CPU-GPU memory, faster interconnects, and novel numerical-precision formats can change the optimal strategies. A good performance engineer keeps an eye on these and updates their mental models accordingly to eliminate bottlenecks quickly. Additionally, the MLPerf [benchmark](#) suites are a great resource to understand AI hardware performance for various models.

Conclusion

This introductory analysis underscores that optimizations are not optional at large scale—they are absolutely necessary. It is the difference between a system that works and one that is utterly impractical. Traditional approaches, whether in hardware or algorithms, break down at this scale. To push forward, we need both advanced hardware and smart software techniques.

It's clear that AI models are pushing physical resource limits. Hardware is racing to keep up with new model architectures and algorithms. And performance engineers are the ones in the driver's seat to ensure that all this expensive machinery is actually delivering results.

We have demonstrated that the role of the AI systems performance engineer is gaining more and more importance. Simply throwing money and hardware at the problem is not enough. We need to co-optimize everything—model architectures, algorithms, hardware, and system design—to push toward the next leaps in AI capability.

As an AI systems performance engineer, your job is multidisciplinary, complex, and dynamic. You will dive into GPU profiling one day, network topology the next day, and perhaps algorithmic complexity the following day. This is a role for a “full-stack” performance geek who loves to squeeze out every drop of available performance from both hardware and software.

In essence, an AI systems performance engineer’s mantra is “mechanical sympathy.” We deeply understand the machinery—both hardware and software—so that we can tailor efficient solutions that exploit the entire stack’s performance capabilities to the fullest.

As we saw earlier, a 100-trillion-parameter model is well beyond the reach of today’s hardware—even an exascale system would require several millennia of GPU time for one training run. This is clearly impractical, underscoring why we need both advanced hardware and clever software optimizations to ever reach this scale.

In the coming chapters, we will demonstrate how to break down the components of an AI system from processors to memory to interconnects to software frameworks—and learn how to optimize each component in a

principled way. We'll study concrete case studies where making small changes brings about huge performance and cost improvements. Doing so, we will help create a mental model for reasoning about performance optimization along multiple dimensions.

By the end of this journey, you as a reader and practitioner will be equipped with knowledge of today's best practices as well as an engineering mindset to tackle tomorrow's challenges. You will have an arsenal of techniques to push AI systems to their limits—now and in the future. For AI systems performance engineers, the mandate is clear. We must learn from these innovations and be ready to apply aggressive optimizations at every level of the stack.

Now, with the context established, let's dive into the hardware components of modern AI systems, including the CPUs, GPUs, memory technologies, network fabrics, and storage mechanisms. By studying the components that underpin contemporary AI supercomputers, you will learn the fundamentals that provide the foundation of subsequent deep dives into optimization techniques in later chapters.