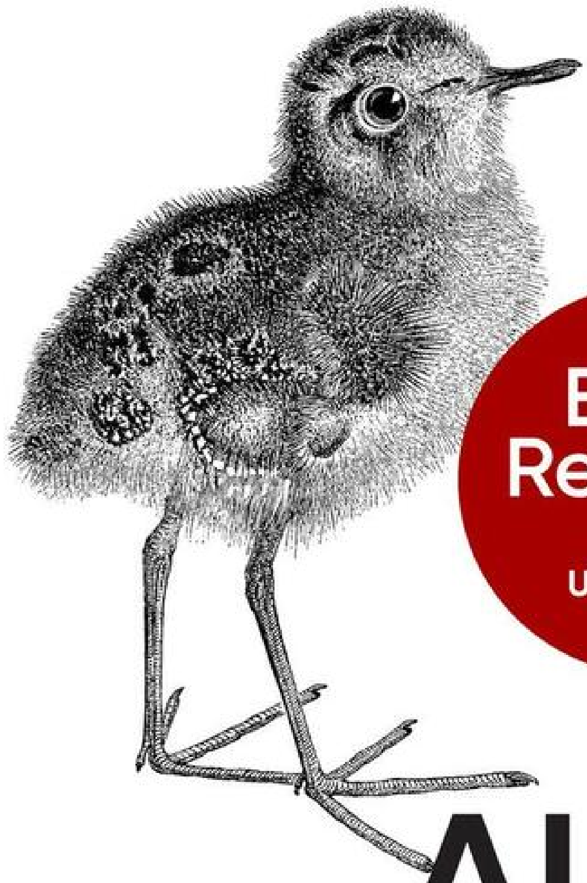
O'REILLY®



Early Release

RAW & UNEDITED

AI Systems Performance Engineering

Optimizing Hardware, Software, and Algorithms for Efficient Training and Inference

Chris Fregly

AI Systems Performance Engineering

Optimizing Hardware, Software, and Algorithms for Efficient Training and Inference

Chris Fregly

O'REILLY®

AI Systems Performance Engineering

by Chris Fregly

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

- Editors: Angela Rufino and Nicole Butterfield
- Production Editor: Kristen Brown
- Interior Designer: David Futato
- Cover Designer: Karen Montgomery
- Illustrator: Kate Dullea

- September 2025: First Edition

Revision History for the Early Release

- 2025-04-18: First Release

See http://oreilly.com/catalog/errata.csp?isbn=9798341627789 for release details.

979-8-341-62778-9

[LSI]

Brief Table of Contents (*Not Yet Final*)

Preface

In the vibrant streets of San Francisco, where innovation is as common as traffic on the 101 highway, I find myself immersed in the awesome world of artificial intelligence. The rapid advancements in AI are redefining the very fabric of our daily lives. From personalization and recommendation engines in the 2000’s and 2010’s to AI assistants and autonomous vehicles in the 2020’s and 2030’s, AI’s influence is pervasive and profound.

My journey into this fast-moving field was driven by a curiosity to understand the intricate balance between hardware and software that powers AI systems. A few years ago, it became evident that the performance of AI applications was dependent on sophisticated algorithms as well as the underlying hardware and software that support them. The synergy and co-design between cutting-edge hardware, meticulous software, and clever algorithms is critical in achieving unprecedented levels of efficiency and scalability.

This realization inspired me to dive deep into the realm of “full-stack” AI performance engineering. I wanted to understand how many components including processors, memory architectures, network interconnects, operating systems, and software frameworks all interact to create robust and efficient AI systems. The complexity of these interactions presented both challenges and opportunities, fueling my desire to unravel the intricacies of this unique combination of technologies.

This book is a realization of my initial exploration as well as many years of hands-on ML and AI system design experience. It is created for engineers, researchers, and enthusiasts who are eager to understand the underpinnings of AI systems performance at all levels. Whether you’re building AI applications, optimizing neural network training strategies, designing and managing scalable inference servers, or simply fascinated by the mechanics of modern AI systems, this book provides the insights that bridge theory and practice across multiple disciplines.

Throughout the chapters, we will embark on a journey that examines the evolution of hardware architectures, dive into the nuances of software optimization, and explore real-world case studies that highlight the patterns and best practices of building both high-performance and cost-efficient AI systems. Each section is designed to build upon the last, creating a cohesive narrative from foundational concepts to advanced applications.

Although I’m the sole author of this book, this was not a sole endeavor. I am indebted to the brilliant minds whose research and innovations have paved the way for the topics covered in this book. Their contributions have been instrumental in shaping the content and depth of this work. To my colleagues, mentors, and reviewers who challenged my perspectives and enriched my understanding, your insights are embedded in every chapter. And to my family and friends whose continuous support kept me writing through the night and into the early morning hours, I extend much heartfelt gratitude.

As we continue to push the frontiers of artificial intelligence and supercomputing, the knowledge in this book aims to inspire, educate, and empower the reader. The journey through the complexities of AI Systems Performance Engineering is not just a technical exploration - it’s a reminder of human ingenuity, the need to understand our surroundings, and a desire to continuously improve through technology and innovation. There are few people in this world who understand the fundamentals of co-designing hardware, software, and algorithms for maximum performance and efficiency. After reading this book, you will be one of them.

Chris Fregly

San Francisco, California

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 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 A100 or H100 accelerators, so they resorted to locally available, less-capable NVIDIA chips.

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 U.S. 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× increase in scale has unlocked qualitatively new capabilities, they require tremendous cost and resources.

For instance, OpenAI’s GPT-3 (175B parameters, 2020) cost on the order of $4 million to train and GPT-4 (2023) required an estimated $78 million. Google’s Gemini Ultra (2023) soared to a staggering ~$191 million. Figure 1-1 illustrates this ballooning of training expenses – from under $10M around 2019 to well over $100M by 2023 for state-of-the-art models.

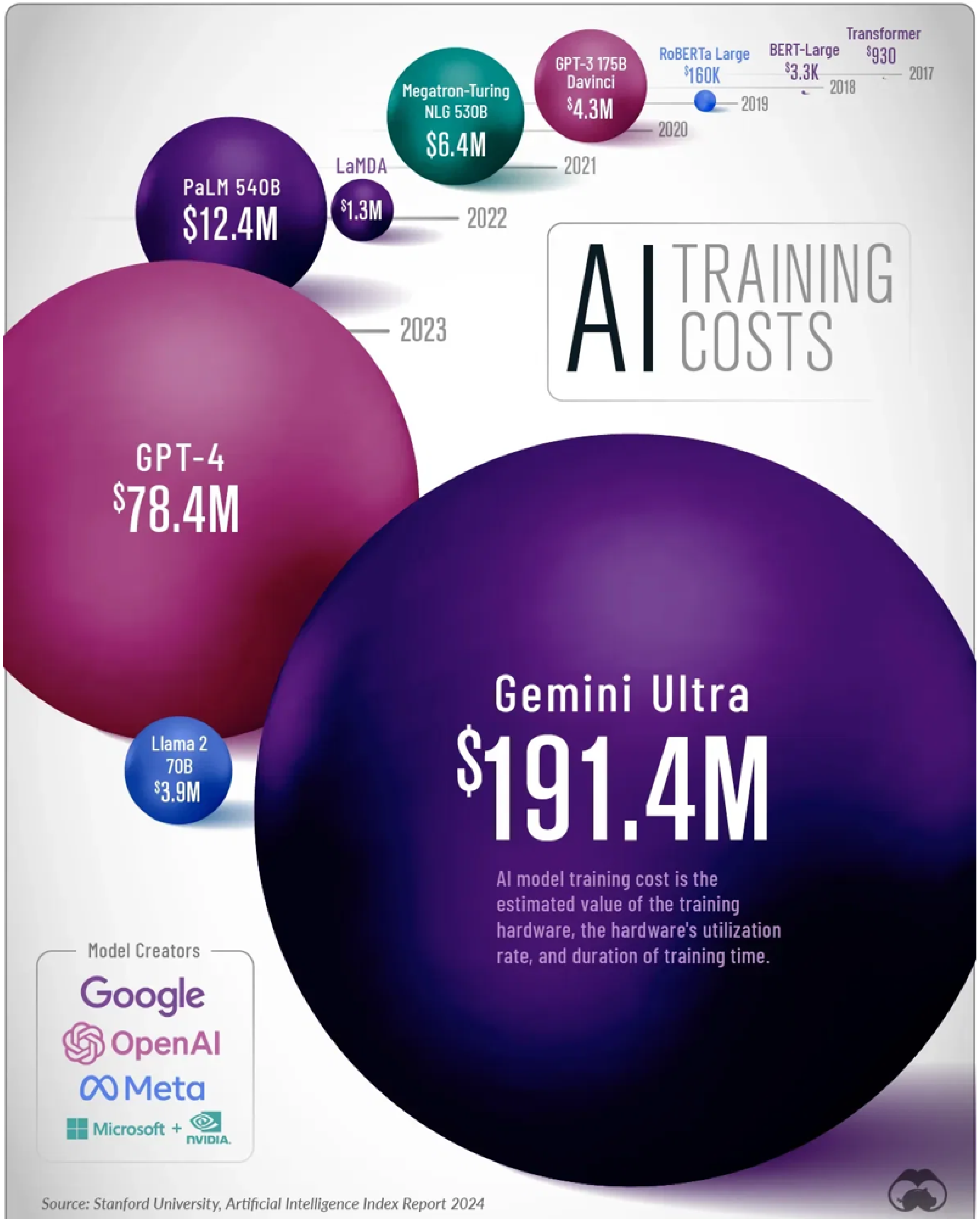


Figure 1-1. The cost to train cutting-edge AI models has skyrocketed (Source: posts.voronoiapp.com)

DeepSeek claims that their DeepSeek-R1 model was trained for only $6 million in compute – an order of magnitude lower than models like GPT-4 and Gemini – while matching performance of rival models that cost orders-of-magnitude more money.

While there was some doubt as to the validity of the $6 million claim, the announcement briefly shocked the U.S. financial market including NVIDIA's stock which dropped 17% on the news, amid concerns that less NVIDIA hardware would be needed in the future. While this market reaction was a bit overblown, it underscores the significant financial impact of such AI efficiency breakthroughs on the global financial markets.

Beyond model training, DeepSeek boasts significant inference efficiency gains through novel hardware-aware algorithmic improvements to the Transformer architecture which powers most modern, frontier large-language models. DeepSeek has clearly demonstrated that clever AI systems performance engineering optimizations can upend the economics of ultra-scale AI model training and inference.

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 money 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 delve deeper into specific techniques, technologies, and best practices that define excellence in AI performance engineering.

The AI Systems Performance Engineer

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 run fast, cost-efficiently, and with maximum performance. As the scale increases, the AI Systems Performance Engineer becomes even more critical.

AI Systems Performance Engineers command top salaries, and for good reason. 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++.

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.

Benchmarking and Profiling

Benchmarking and profiling involves measuring latency, throughput, memory usage, and other performance metrics for AI models under various workloads. To identify bottlenecks, we must iteratively use profiling tools such as NVIDIA Nsight and PyTorch profiler to track performance over time as we make controlled enhancements. It's important to set up automated performance tests to catch regressions early

Debugging and optimizing 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 Transformer Engine hardware. 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 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 yield major wins. For example, maybe a data preprocessing step written in Python is holding up an entire training pipeline. Reimplementing it in C++ or using a GPU-optimized vectorization library like NVIDIA’s CuPyNumeric library could remove that bottleneck

Scaling Distributed Training and Inference

Scaling small research workloads to larger production workloads on ultra-scale clusters will ensure that as we move from 8 GPUs to 80,000 GPUs, the system will scale with minimal efficiency loss. This requires that you optimize communication between GPUs on a single node - as well as across 1000’s of nodes - for collective reduction/aggregation operations like all-reduce which is used extensively during model training.

You may want to place data cleverly across nodes using data 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 an Mixture of Experts (MoE) model and can take advantage of expert parallelism.

AI Systems Performance Engineers often need to implement distributed communication strategies to allow models to train on tens of thousands of GPUs without incurring excessive overhead.

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 everything from ensuring GPUs are fed with data at full throttle, to pinning threads on specific CPU cores to avoid context-switch overhead, to orchestrating memory usage so that we don't run out of GPU RAM in the middle of training/inferencing with a large model.

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, application developers, and infrastructure teams 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. The performance engineer sits at the intersection of these multi-disciplinary 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/DualPipe, and Fire-Flyer File System (3FS). Each project was production-tested and aimed at squeezing the most performance from their hardware.

FlashMLA is their optimized Attention CUDA kernel. DeepGEMM provides an FP8-optimized matrix multiplication library that outperforms many vendor kernels on both dense and sparse operations DeepEP is their highly-tuned communication library for Mixture-of-Experts models. 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, DeepSeek not only demonstrated the credibility of their claims by allowing others to reproduce their results, but also contributed back to the community. It allows others to benchmark, reproduce, and learn from their methods whether it's overlapping communication with DeepEP/DualPipe pipeline parallelism or saturating SSD and RDMA bandwidth with 3FS

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.

Open efforts like DeepSeek's Open-Infra Index provide valuable baselines and tools. They catalog real-world performance data on various AI hardware setups and encourage apples-to-apples comparisons. We’ll leverage some of these open benchmarks to illustrate points in later chapters. 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

Towards 100-Trillion-Parameter Models

100 trillion parameter models is an aspirational milestone for AI. 100 trillion is roughly the number of synaptic connections in the human brain’s neocortex 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.

A naive back-of-the-envelope calculation suggests that training a dense 100-trillion-parameter model might require on the order of 10^29 floating-point operations (FLOPs). Even with an exascale computer, executing 10^29 FLOPs would take thousands of GPU-years if done naively. In practical terms, if we used a state-of-the-art AI supercomputer delivering ~1.4 exaFLOPS (10^18 FLOPs) of 8-bit compute performance, a single training run could span on the order of 3,000 years. Clearly, new approaches are needed to make 100-trillion-parameter training feasible.

TIP

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.

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 MoE's are in contrast to traditional dense models like the common GPT-series large-language models (LLMs) made popular by OpenAI. In fact, it's rumored that OpenAI's proprietary GPT-series and O-series reasoning models are multi-trillion-parameter models based on the MoE architecture.

Sparse models like MoE’s only activate 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 less 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-R1 is a great example of MoE efficiency. Despite having a massive 671 billion parameters in total, only 37 billion parameters are activated per input token. This makes DeepSeek-R1 much more resource-efficient than a similarly-sized dense large-language model. Another example is Google’s Switch Transformer MoE from 2022. 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 200 TB of GPU memory to load the model if each parameter is stored as in 16-bit (2-byte) precision. This is 3 orders of magnitude (1000x) compared to the 288 GB of GPU RAM on a single NVIDIA Blackwell GPU. So to just load the 100 trillion model weights would require close to 700 Blackwell GPUs - and that’s without performing any calculations with the model weights, accepting any inference request data, etc. If a typical GPU server has 8 GPUs, this would require approximately 100 GPU servers just to load the model!

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 non-trivial. In particular, communication overhead between the GPUs grows significantly as the 100-trillion parameter model is partitioned across 700 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 co-evolve to meet this new frontier.

NVIDIA’s “AI Supercomputer in a Rack”

To meet the challenges of ultra-scale computing, NVIDIA has built a new class of AI supercomputers specifically aimed at trillion-parameter-scale workloads. One example is the NVIDIA GB200 NVL72 - an AI supercomputer condensed into a single data center rack. NVIDIA refers to the NVL72 as an “AI supercomputer in a rack” – and for good reason.

At a high level, the GB200 NVL72 rack integrates 36 Grace-Blackwell superchips with a specialized networking fabric called NVLink. Each Grace-Blackwell superchip is a combination of 1 ARM-based NVIDIA Grace CPU with 2 NVIDIA Blackwell GPUs for a total of 72 Blackwell GPUs (hence, the “NVL72” in the name!).

This entire NVL72 rack behaves like one giant accelerator to the user. More details are provided on the compute, memory, and interconnect hardware details of this supercomputer in Chapter 2. For now, let’s analyze the overall performance specifications of this AI supercomputer as a whole in the context of modern generative AI models.

Each NVL72 rack delivers 1.44 exaFLOPS of AI compute in low-precision (4-bit) mode and provides 13.5 TB of ultra-fast high-bandwidth memory (HBM) spread across the 72 GPUs. In simpler terms, it's a self-contained 120 kW AI training and inference AI supercomputer that can train and serve trillion-parameter models - as well as fit into a single rack in your data center. And by combining these racks together to form ultra-scale 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 AWS, GCP, Azure, CoreWeave, and Lambda Labs.

TIP

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.

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 ultra-scale 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 co-design discussed in the next section.

Mechanical Sympathy: Hardware-Software Co-Design

Mechanical sympathy is a term originally coined by racer/engineer Martin Thompson (drawing an analogy to racecar drivers who intimately understand their car's mechanics). In computing, it refers to writing software that is deeply aware of the hardware it runs on. In the AI context, it means co-designing 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 yield 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 dramatically reduces memory movement and speeds up attention computation. Replacing the default Transformer attention mechanism/algorithm with FlashAttention yields a 2–4× speedup in training and inference for long sequences, while also reducing the overall memory footprint Such a change eliminates 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 cheaply. Since FlashAttention, many new Attention algorithms have emerged including DeepSeek's Multi-Headed Latent Attention (MLA).

DeepSeek's MLA algorithm - implemented as an NVIDIA GPU kernel and open-sourced in 2025 - is another example of hardware-software co-design, or mechanical sympathy. Similar to FlashAttention, MLA restructures the Attention computations to better utilize NVIDIA's memory hierarchy and dedicated GPU "tensor cores." These algorithmic optimizations adapted MLA to the strengths of the China-constrained NVIDIA GPUs and achieved higher throughput at a fraction of the cost using FlashAttention.

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!

Another example is NVIDIA's Blackwell GPU which includes an improved exponential computation unit specifically designed to accelerate the softmax operation in the Transformer's Attention algorithm. The softmax had become a bottleneck on previous GPUs even though it's critical to the Attention mechanism.

This tight interplay – GPUs and AI algorithms co-evolving – is the heart of mechanical sympathy in AI. These co-design innovations can only happen with close collaboration between the hardware companies (e.g. NVIDIA and 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/s 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 in a paper titled, “Revisiting Reliability in Large-Scale Machine Learning Research Clusters”.

TIP

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 useful work 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 training. Goodput can be normalized by the cluster's maximum possible throughput to yield a 0–1 efficiency ratio.

Meta's AI infrastructure team highlighted the importance of goodput by revealing that their large-scale GPU clusters measured only about 25–30% of peak compute throughput as useful training work, or goodput. 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, waiting for data, or recovering from failures. 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 poor input pipeline and synchronization, it only achieves 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.

TIP

This cost savings is why AI Systems Performance Engineers earn top dollar in the industry today. We pay for ourselves many times over - especially at scale!

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, 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.

Part I: AI System Hardware, OS, CUDA, and PyTorch Optimizations

We begin with an understanding of the hardware foundation. Understanding the hardware capabilities and limitations is crucial before we can discuss software and algorithm optimizations.

Chapter 2 provides an in-depth look at NVIDIA AI System hardware including the GB200 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.

Chapter 3 will then cover OS-level optimizations for GPU-based AI systems. These optimizations include CPU and memory pinning. Chapter 4 discusses Docker-container and Kubernetes-orchestration considerations as well as network I/O and storage configurations for GPU environments.

Chapters 5 and 6 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.

Chapter 7 discusses PyTorch-specific optimizations including the PyTorch compiler and OpenAI's Python-based Triton compiler (and language). 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.

Part II: Scalable, Distributed Model Training And Inference Strategies

With NVIDIA hardware and CUDA software context in hand, we’ll dive into distributed communication optimizations including training and serving ultra-large 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.

Chapter 8 discusses distributed parallelization techniques for model training including data parallelism, tensor parallelism, pipeline parallelism, sequence parallelism, and mixture-of-experts. 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 ultra-scale model training including gradient checkpointing, sharding optimizer states, and offloading to larger CPU memory. These techniques are vital when model sizes exceed the physical GPU hardware limits.

Chapter 9 focuses on software and algorithmic innovations for high-throughput, low-latency model inference. We discuss the most popular model-serving engines including vLLM, NVIDIA TensorRT-LLM, and NVIDIA Dynamo. 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 using specialized TensorRT kernels.

Part III: Ultra-Scale Case Studies, Emerging Trends, and Optimization Cheat Sheet

In the final part of the book, we will present various performance optimization case studies and emerging trends. Additionally, we present an optimization “Cheat Sheet” to summarize the high-level performance and cost optimization tips and tricks covered in this book.

Chapter 5 provides case studies for optimizing massive AI clusters at scale. The case studies include training and serving multi-billion and multi-trillion parameter models efficiently.

Chapter 6 covers emerging trends in this broad field of AI systems performance engineering. Such trends include training-efficiency improvements, optical interconnects for multi-rack scaling, better algorithms for sparse models, among others. This helps to paint a picture of where 100-trillion-parameter-scale AI systems are headed.

Chapter 7 presents 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 efficiency gains discussed in this book.

The 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 ultra-large AI systems - as well as gain some practical experience with tools to apply those optimizations on ultra-scale, mutii-GPU, multi-node, and multi-rack AI systems like the NVIDIA GB200 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 how much of the compute is actually doing useful work (e.g. forward/backprop) vs. waiting on data and other overhead. Strive to improve useful work.

Optimization Beats Brute Force.

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 – skillful engineering outperformed brute-force spending.

Strive for an Order-of-Magnitude Impact.

At scale, even a small-percentage efficiency gain can save millions of dollars. Conversely, inefficiencies such as redundant computations or poor data pipelines can silently increase costs.

Use a Profile-Driven Approach.

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 co-design: sometimes algorithm changes can alleviate hardware limits, and sometimes new hardware features enable new algorithms.

Stay Informed on the Latest Hardware.

Modern AI hardware is evolving rapidly. New capabilities such as unified CPU-GPU memory, faster interconnects, or novel numerical-precision formats can change the optimal strategies. A good engineer keeps an eye on these and updates their mental models accordingly to eliminate bottlenecks quickly.

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 an AI Systems Performance Engineering 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 AI Systems Performance Engineers, our job is multi-disciplinary, complex, and dynamic. We 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.

The challenge on the horizon is monumental. For instance, consider the lofty goal of training and serving a 100-trillion-parameter model. A naïve back-of-the-envelope calculation shows that training such a model would require on the order of 10^29 FLOPs. Even running on an exascale supercomputer at 10^18 FLOPs/sec, you would need to train for several-thousand GPU-years just for a single pass through an average-sized language-based dataset.

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.

Chapter 2. AI System Hardware Overview

Imagine condensing a supercomputer's worth of AI hardware into a single rack. NVIDIA's latest architecture does exactly that. In this chapter, we dive into how NVIDIA fused CPUs and GPUs into powerful "superchips" and then wired dozens of them together with ultra-fast interconnects to create an AI supercomputer-in-a-box. We'll explore the fundamental hardware building blocks – the Grace CPU and Blackwell GPU –

and see how their tight integration and enormous memory pool make life easier for AI engineers.

Then we'll expand outward to the networking fabric that links 72 of these GPUs as if they were one machine. Along the way, we'll highlight the leaps in compute performance, memory capacity, and efficiency that give this system its superpowers. By the end, you'll appreciate how this cutting-edge hardware enables training and serving multi-trillion-parameter models that previously seemed impossible.

The CPU and GPU "Superchip"

NVIDIA's approach to scaling AI starts at the level of a single, combined CPU+GPU "superchip" module. Beginning with the Hopper generation, NVIDIA started packaging an ARM-based CPU together with one or more GPUs in the same unit, tightly linking them with a high-speed interface. The result is a single module that behaves like a unified computing engine.

The first incarnation was the Grace-Hopper (GH200) superchip which pairs one Grace CPU with one Hopper GPU. Next is the Grace-Blackwell (GB200) superchip, which pairs one Grace CPU with two Blackwell GPUs on the same package. Essentially, the Grace CPU sits in the center of the module, flanked by two

Blackwell GPU dies, all wired together as one coherent unit as shown in Figure 2-1.

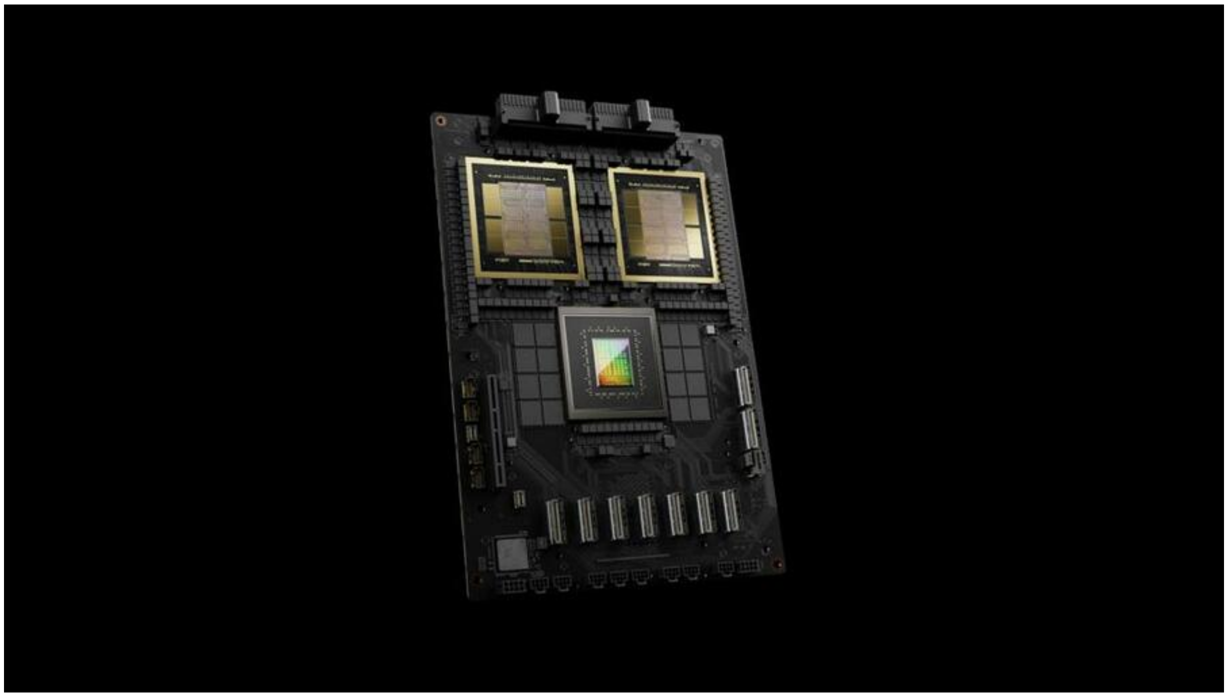


Figure 2-1. NVIDIA Grace-Blackwell Superchip module, containing one Grace CPU (center) and two Blackwell B200 GPUs (top) on a single module with shared memory address space.

What's the advantage of fusing a CPU and GPUs together? In a single word - memory! In a traditional system, the CPU and GPU have separate memory pools and communicate over a relatively slow bus (like PCIe), which means data has to be copied back and forth. NVIDIA's superchip eliminates that barrier by connecting the CPU and GPUs with a custom high-speed bus called NVLink-C2C (chip-to-chip) link that runs at about 900 GB/s of bandwidth between the Grace CPU and each GPU.

NVLink-C2C's interconnect speed is orders of magnitude faster than typical PCIe. And, importantly, it is cache-coherent. Cache coherency means the CPU and GPU share a unified memory address space and always see the same data values. In practice, the Grace CPU and Blackwell GPUs on a superchip can all access each other's memory directly as if it were one huge memory pool. The GPU can read or write data stored in the CPU's memory and vice versa without needing explicit copies. This unified memory architecture is often called Extended GPU Memory (EGM) by NVIDIA, and it effectively blurs the line between "CPU memory" and "GPU memory."

Each Grace-Blackwell superchip carries a tremendous amount of memory. The Grace CPU comes with hundreds of gigabytes of LPDDR5X DRAM attached, and each Blackwell GPU has its own high-speed HBM stacks. In the GB200 superchip, the Grace CPU provides 480 GB of memory and the two Blackwell GPUs together contribute 384 GB of HBM3e memory (192 GB per GPU). That's a total of 864 GB of memory accessible by the GPUs and CPU in a unified address space.

To put it simply, each superchip has nearly a terabyte of fast, unified memory at its disposal. This is a game-changer for giant AI models. In older systems, a single GPU might be limited to <100 GB of memory, which meant models larger than that had to be partitioned or offloaded to slower storage. Here, a GPU can seamlessly utilize the CPU's memory as an extension.

If a neural network layer or a large embedding table doesn't fit in the GPU's local HBM, it can reside in the CPU's memory and the GPU will still be able to work with it across NVLink-C2C. From a programmer's perspective, the hardware takes care of data movement. One simply needs to allocate the memory and use it. The unified memory and coherence mean the days of manually shuttling tensors between CPU and GPU memory are largely over.

Of course, GPU memory is still much faster and closer to the GPU cores than CPU memory – you can think of the CPU memory as a large but somewhat slower extension. Accessing data in LPDDR5X isn't as quick as HBM on the GPU. It's on the order of 10× lower bandwidth and higher latency. A smart runtime will keep the most frequently used data in the 192 GB of HBM and use the CPU's 480 GB for overflow or less speed-critical data.

The key point is that overflow no longer requires going out to SSD or across a network. The GPU can fetch from CPU RAM at perhaps 900 GB/s (half-duplex 450 GB/s each way), which while slower than HBM, is vastly faster than fetching from NVMe storage. This flexibility is invaluable. It means that a model that is, say, 500 GB in size (too large for a single GPU's HBM) could still be placed entirely within one superchip module (192 GB in HBM + the rest in CPU memory) and run without partitioning the model across multiple GPUs. The GPU would just transparently pull the extra data from CPU memory when needed.

In essence, memory size ceases to be a hard limit for fitting ultra-large models, as long as the total model fits within the combined CPU+GPU memory of the superchip. Many researchers have faced the dreaded "out of memory" errors when models don't fit on a GPU – this architecture is designed to push that boundary out dramatically.

NVIDIA Grace CPU

The Grace CPU itself is no sloth. It's a high-core-count (72 cores) ARM CPU custom-built by NVIDIA for bandwidth and efficiency. Its job in the superchip is to handle general-purpose tasks, preprocess and feed data to the GPUs, and manage the mountain of memory attached to it. It runs at a modest clock speed but makes up for it with huge memory bandwidth – roughly 0.5 TB/s to its LPDDR5X memory – and lots of cache including tens of MB of L3 cache.

The philosophy is that the CPU should never become a bottleneck when shoveling data to the GPUs. It can stream data from storage or perform on-the-fly data transformations like tokenization or data augmentation - feeling the GPUs through NVLink-C2C very efficiently. If part of your workload is better on the CPU, the Grace cores can tackle that and make the results immediately accessible by the GPUs.

This is a harmonious coupling in which the CPU extends the GPU’s capabilities in areas where GPUs are weaker like random memory accesses or control-heavy code. And the GPUs accelerate the number-crunching where CPUs can’t keep up. The low-latency link between the CPU and GPUs means they can trade tasks without the usual overhead. For example, launching a GPU kernel from the CPU can happen much faster than on a traditional system, since the command doesn’t have to traverse a slow PCIe bus. The CPU and GPU are essentially on the same board. This should feel similar to calling a fast local function vs. a slower remote function.

Now let’s talk about the Blackwell GPU, the brute-force engine of the superchip.

NVIDIA Blackwell GPU

Blackwell is NVIDIA's codename for this GPU generation, and it represents a significant leap over the previous Hopper (H100) GPUs in both compute horsepower and memory. Each Blackwell B200 GPU in a GB200 superchip isn't a single chip but actually a multi-chip module (MCM) with two GPU dies sitting on one package.

This is the first time NVIDIA's flagship data center GPU has used a chiplet approach. This effectively splits what would be one enormous GPU into two sizable dies and links them together. Why do this? Because a single monolithic die is limited by manufacturing, there's a limit to how large you can make a chip on silicon. By combining two physical dies into a single GPU, NVIDIA can double the total transistor budget for the GPU.

In Blackwell's case, each die has about 104 billion transistors, so the combined GPU module has around 208 billion transistors. This is an astonishing amount of electrical complexity and sophistication. These two dies communicate via a specialized, high-speed, die-to-die interconnect that runs at 10 TB/s bandwidth between them allowing the two dies to function as one unified GPU to the software layer running on top of it.

From the system’s perspective, a Blackwell “GPU” is one single device with a large pool of memory (192 GB HBM) and a ton of execution units, but under the hood its two chips working in tandem. NVIDIA’s software and scheduling ensure that work is balanced across the two dies, and memory accesses are coherent. This allows developers to largely ignore this complexity as they appear as one GPU as NVIDIA intended.

Each Blackwell GPU module has 192 GB of HBM3e memory divided across the two dies (96 GB each). This doubles the 96 GB from the previous-generation Hopper H100 GPU and gives much more headroom for model parameters, activations, and input data. The memory is also faster as Blackwell’s HBM3e has an aggregate bandwidth of roughly 8 TB/s per GPU. For comparison, the Hopper H100 delivered about 3.35 TB/s, so Blackwell’s memory subsystem is about 2.4× more throughput by design.

Feeding data at 8 terabytes per second means the GPU cores are kept busy crunching on huge matrices without frequently stalling to wait for data. NVIDIA also beefed up on-chip caching as Blackwell has a total of 100 MB of L2 cache (50 MB on each die). This cache is a small but ultra-fast memory on the GPU that holds recently used data. By doubling the L2 cache size compared to H100’s 50 MB L2 cache, Blackwell can keep more of the neural network weights or intermediate results on-chip, avoiding extra trips out to HBM. This again helps ensure the GPU’s compute units are seldom starved for data.

NVIDIA GPU Tensor Cores and Transformer Engine

Speaking of compute units, Blackwell introduces enhancements specifically aimed at AI workloads. Central to this is NVIDIA’s Tensor Core technology and the Transformer Engine. Tensor Cores are specialized units within each streaming multiprocessor (SM) of the GPU that can perform matrix multiplication operations at very high speed.

Tensor Cores were present in prior generations, but Blackwell’s Tensor Cores support even more numerical formats, including extremely low-precision ones like 8-bit and 4-bit floating point. The idea behind lower precision is simple. By using fewer bits to represent numbers, you can perform more operations at the same time - not to mention your memory goes further since less bits are used to represent the same numbers. This, of course, assumes that your algorithm can tolerate a little loss in numerical precision. These days, a lot of AI algorithms are designed with low-precision numerical formats in mind.

NVIDIA pioneered the Transformer Engine (TE) to automatically adjust and use mixed precision in deep learning where critical layers use higher precision (FP16 or BF16) and less-critical layers use FP8. TE automatically optimizes the balance of precision with the goal of maintaining the model's accuracy at the lower precision.

In the Hopper generation, the Transformer Engine first introduced FP8 support which doubled the throughput versus FP16. Blackwell takes it one step further by introducing an even-lower precision called FP4, a 4-bit floating-point format that uses half the number of bits of FP8 to represent a number. FP4 is so tiny that it can potentially double the compute throughput of FP8.

In fact, one Blackwell GPU can achieve about 9 PFLOPS (9 quadrillion operations per second) of compute using FP4 Tensor Core operations. This is roughly double its FP8 rate, and about 4× its FP16 rate. To put that in perspective, the earlier H100 GPU peaked around 1 PFLOP in FP16, so Blackwell can be on the order of 2–2.5× faster in FP16 and much more when leveraging FP8/FP4. Figure 2-2 shows the relative speedup of FP8 and FP4 relative to FP16.

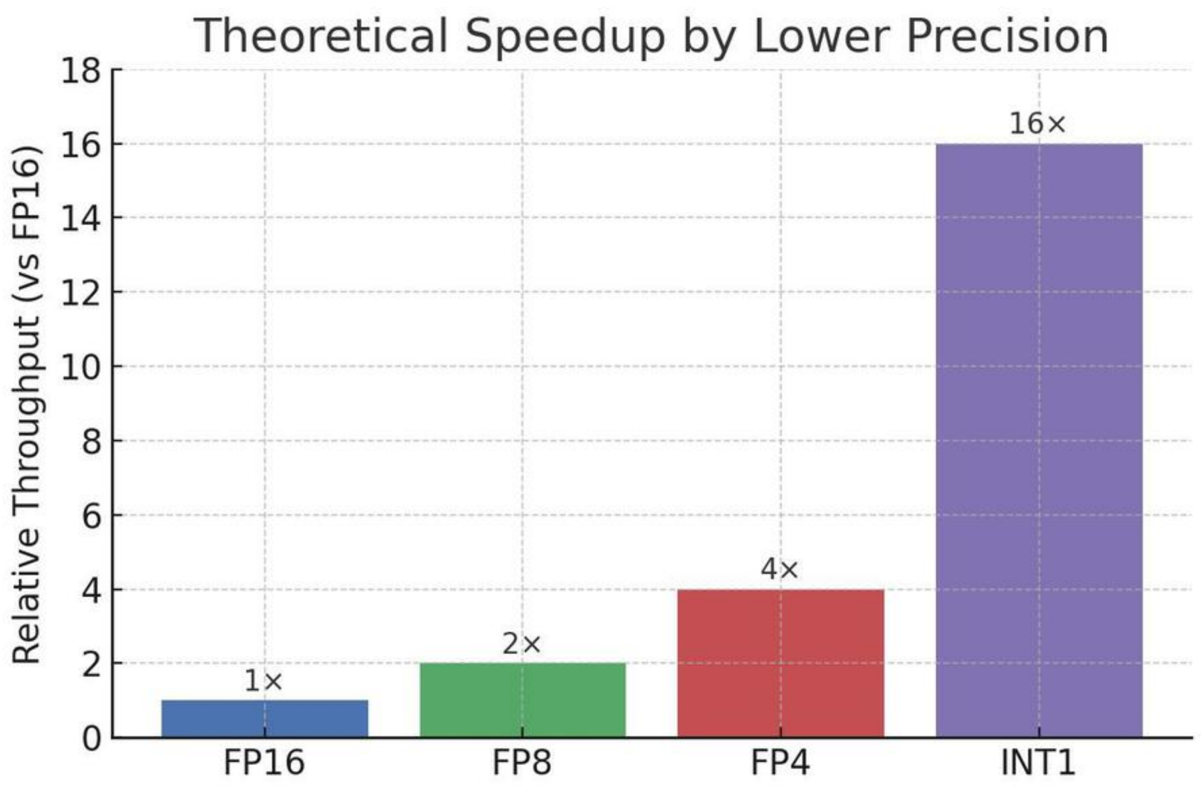


Figure 2-2. Relative speedup of FP8 and FP4 compared to FP16.

An entire NVL72 rack (72 GPUs) has a theoretical Tensor Core throughput over 1.4 exaFLOPS (that's 1.4×10^18) in 4-bit precision. This is a mind-boggling number that puts this single rack in the realm of the world's fastest supercomputers - albeit at low FP4 precision. Even if real-world workloads don't always hit that peak, the capability is there, which is astonishing.

Blackwell boasts version 2 of the NVIDIA Transformer Engine (TE). This is the magic sauce that makes using FP8 and FP4 practical. It dynamically selects the precision for each layer of a neural network during training or inference, trying to use the lowest precision that will still preserve model accuracy. For example, the TE might keep the first layers of a neural net in FP16 since early layers can be sensitive to noise. But, based on heuristics, it could decide to use FP8 or FP4 for later layers that are more tolerant - or for giant embedding matrices where high precision isn't as critical.

All of this happens under the hood in NVIDIA's libraries. As a user, you just enable mixed precision and let it go. The result is a huge speedup that essentially comes "for free." Many large language models (LLMs) today train in FP8 for this reason, effectively doubling training speed compared to FP16 - and with negligible accuracy loss. Blackwell was built to make FP8 and FP4 not just theoretical options but everyday tools.

In fact, tests show that Blackwell delivers nearly 5× higher AI throughput using FP4 vs FP16 in some cases. These formats slash memory usage as well. Using FP4 halves the memory needed per parameter compared to FP8, meaning you can pack an even larger model into the GPU's memory.

NVIDIA has effectively bet on AI's future being in lower precision arithmetic, and has given Blackwell the ability to excel at it. This is especially critical for inference serving of massive models, where throughput (tokens per second) and latency are paramount.

To illustrate the generational leap forward from Hopper to Blackwell, NVIDIA reported an H100-based system could only generate about 3 to 5 tokens per second per GPU for a large 1.8-trillion parameter mixture-of-experts (MoE) model - with over 5 seconds of latency for the first token. This is too slow for interactive use.

The Blackwell-based system (NVL72) ran the same model with around 150 tokens per second per GPU, and cut first-token latency down to ~50 milliseconds. That is roughly a 30× throughput improvement and a huge reduction in latency, turning an impractical model into one that can respond virtually in real-time.

This dramatic speedup came from raw FLOPs, the combination of faster GPUs, lower precision (FP4) usage, and the NVLink interconnect keeping the GPUs fed with data. It underscores how a holistic design that spans across both compute and communication can translate into real-world performance gains.

In essence, Blackwell GPUs are more powerful, smarter, and better fed with data than their predecessors. They chew through math faster, thanks to Tensor Cores, Transformer Engine, and low precision. Additionally, the system architecture ensures that data is made available quickly thanks to huge memory bandwidth, large caches, and NVLink.

Before moving on, let's quickly discuss the hierarchy inside the GPU, as this is useful to understand performance tuning later.

Streaming Multiprocessors, Threads, and Warps

Each Blackwell GPU, like its predecessors, consists of many Streaming Multiprocessors (SMs). Think of these like the "cores" of the GPU as shown in Figure 2-3.

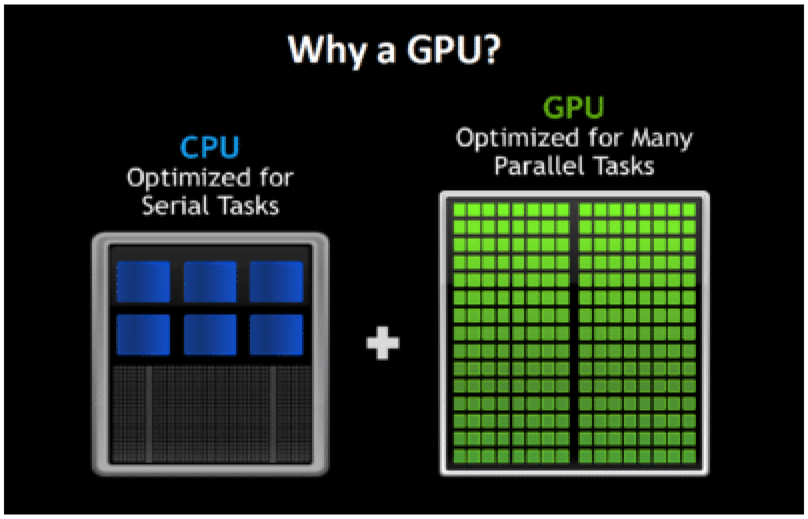


Figure 2-3. Comparing CPU cores to GPU cores. (*https://www.cudocompute.com/blog/nvidia-gb200-everything-you-need-to-know*, modal.com).

Each SM contains a bunch of arithmetic units (for FP32, INT32, etc.), Tensor Cores for matrix math, load/store units for memory operations, and some special function units for things like transcendental math. The GPU also has its own small pool of super-fast memory including registers, shared memory, and L1 cache.

The SM executes threads in groups of 32 called warps where each warp contains 32 threads. It can execute many active warps in parallel to help cover latency if a thread is waiting on data from memory. Consider an SM having dozens of warps (hundreds of threads) in flight concurrently. If one warp is waiting on a memory fetch, another warp can run. This is called latency hiding. We will revisit latency hiding throughout the book. This is a very important performance-optimization tool to have in your tuning toolbox.

A high-end GPU like Blackwell will have on the order of 140 SMs. Each SM is capable of running thousands of threads concurrently. This is how we get tens of thousands of active threads onto a single GPU. All those SMs share a large 100 MB L2 cache, as we mentioned earlier, and share the memory controllers that connect to the HBM. The memory hierarchy contains registers (per thread) → shared memory/L1 cache (per SM) → L2 cache (on GPU, shared by all SMs) → HBM memory (off-chip) as shown in Figure 2-4.

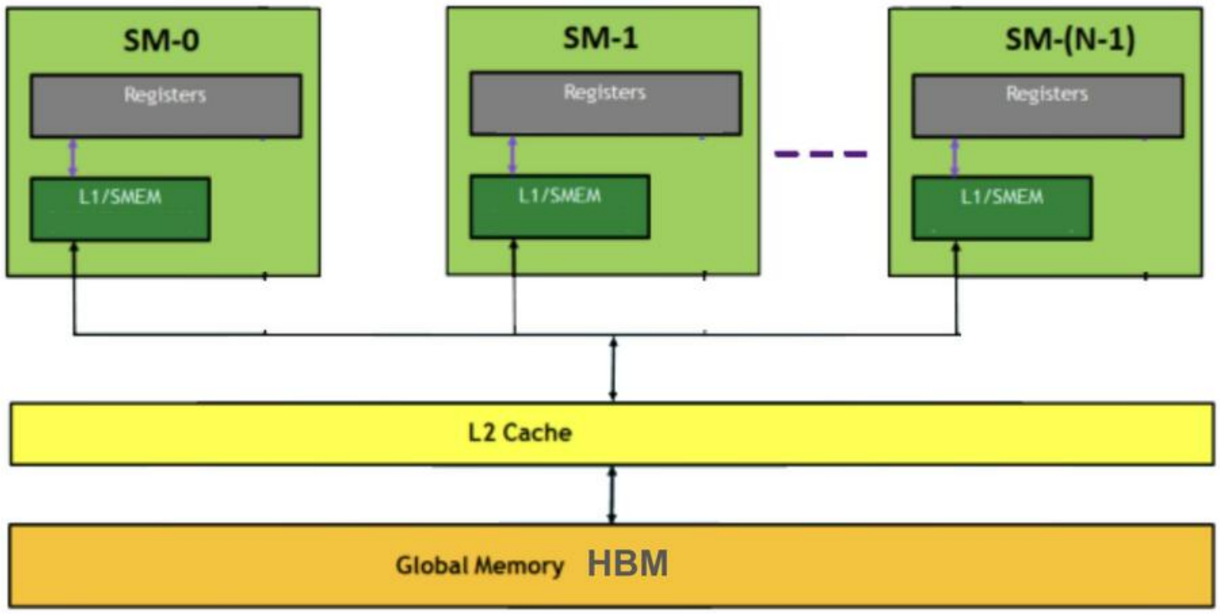


Figure 2-4. GPU memory hierarchy

For best performance, data needs to stay as high in that hierarchy as possible. If every operation went out to HBM 0 even at 8 TB/s, the GPU would stall too often due to the increased latency of accessing off-chip memory. By keeping reusable data in SM local memory or L2 cache, the GPU can achieve enormous throughput. The Blackwell architecture's doubling of cache and bandwidth is aimed exactly at keeping the GPU beast fed and happy.

As performance engineers, we'll see many examples where a kernel's performance is bound by compute as well memory traffic and throughput. NVIDIA clearly designed Blackwell so that, for many AI workloads, the balance between FLOPs and memory bandwidth is well-matched. Roughly speaking, the balance is on the order of 2–3 FLOPs of compute per byte of memory bandwidth in FP4 mode. This is a reasonable ratio for dense linear algebra operations. This means the GPUs will often be busy computing rather than waiting on data, given well-optimized code, of course. Note that certain operations like huge reductions or random memory accesses can still be memory-bound, but the updated GPU, memory, and interconnect hardware make this a bit less of an issue.

Ultra-Scale Networking Treating Many GPUs as One

Packing two GPUs and a CPU into a superchip gives us an incredibly powerful node – but NVIDIA didn’t stop at one node. The next challenge is connecting many of these superchips together to scale out to even larger model training.

The flagship configuration using GB200 superchips is what NVIDIA calls the NVL72 system, essentially an AI supercomputer contained in a single rack. NVL72 stands for a system with 72 Blackwell GPUs - and 36 Grace CPUs - all interconnected via NVLink.

The GB200 NVL72 is built as 18 compute nodes (each 1U in size), where each node contains two GB200 superchips for a total of 4 Blackwell GPUs + 2 Grace CPUs per compute node as shown in Figure 2-5.

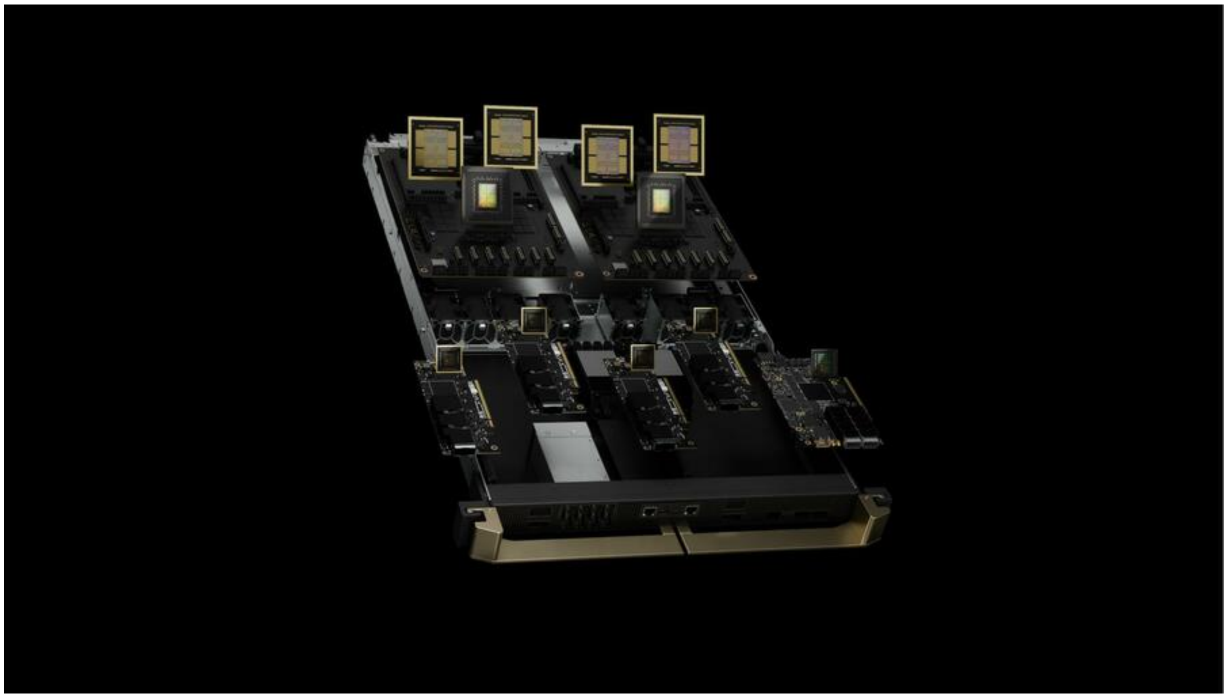


Figure 2-5. A 1U compute tray within the GB200 NVL72 rack with two Grace-Blackwell superchips where each superchip module has one Grace CPU and two Blackwell GPU dies.The NVL72 has 18 of these trays linked together (Source: developer.nvidia.com).

By connecting 18 compute nodes together, the GB200 NVL72 links 72 Blackwell GPUs (18 nodes * 4 GPUs) and 36 Grace CPUs (18 nodes * 2 CPUs) together to form a powerful, unified CPU-GPU cluster. The remarkable thing about NVL72 is that every GPU can talk to any other GPU at very high speed as if all 72 were on the same motherboard. NVIDIA achieved this using a combination of NVLink 5 connections on the GPUs and dedicated switch silicon called NVSwitch.

NVLink and NVSwitch

Each Blackwell GPU has 18 NVLink 5 ports where each port can support 100 GB/s of data transferred bidirectionally or 50 GB/s in a single direction. Combined, a single GPU can shuffle up to 1.8 TB/s (18 NVLink ports * 100 GB/s) of data with its peers via NVLink. This is double the per-GPU NVLink bandwidth of the previous generation as the Hopper H100 uses NVLink 4 which runs at half of the bidirectional 900 GB/s speed of NVLink 5.

The GPUs are cabled in a network through NVSwitch chips. NVSwitch is essentially a switching chip similar to a network switch, but it's built specifically for NVLink. This means any GPU can reach any other GPU via a single hop through one of the NVSwitch chips with full bandwidth. Figure 2-6 shows an NVLink Switch tray used in NVL72.

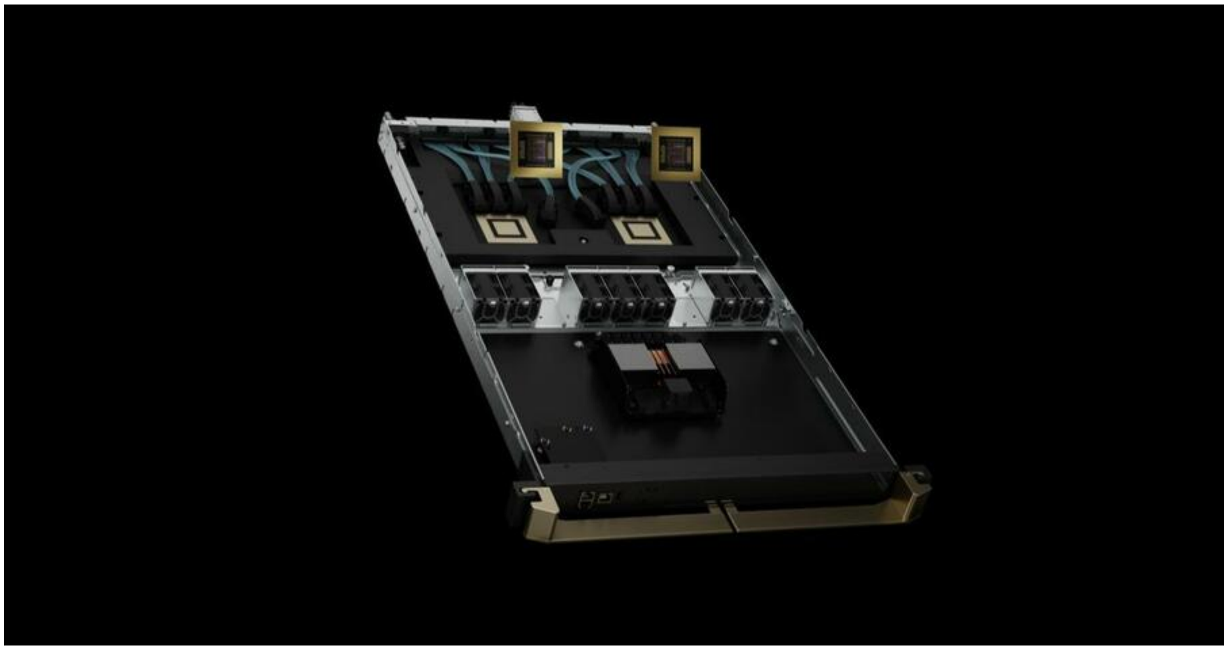


Figure 2-6. One NVLink Switch tray from NVL72. (Source: developer.nvidia.com)

Each switch tray contains two NVSwitch chips (the large chips visible), and multiple high-speed ports (the blue cables represent NVLink connections). In the NVL72 rack, 9 such switch trays, shown in Figure 2-7, provide the fabric that fully connects the 72 Blackwell GPUs.

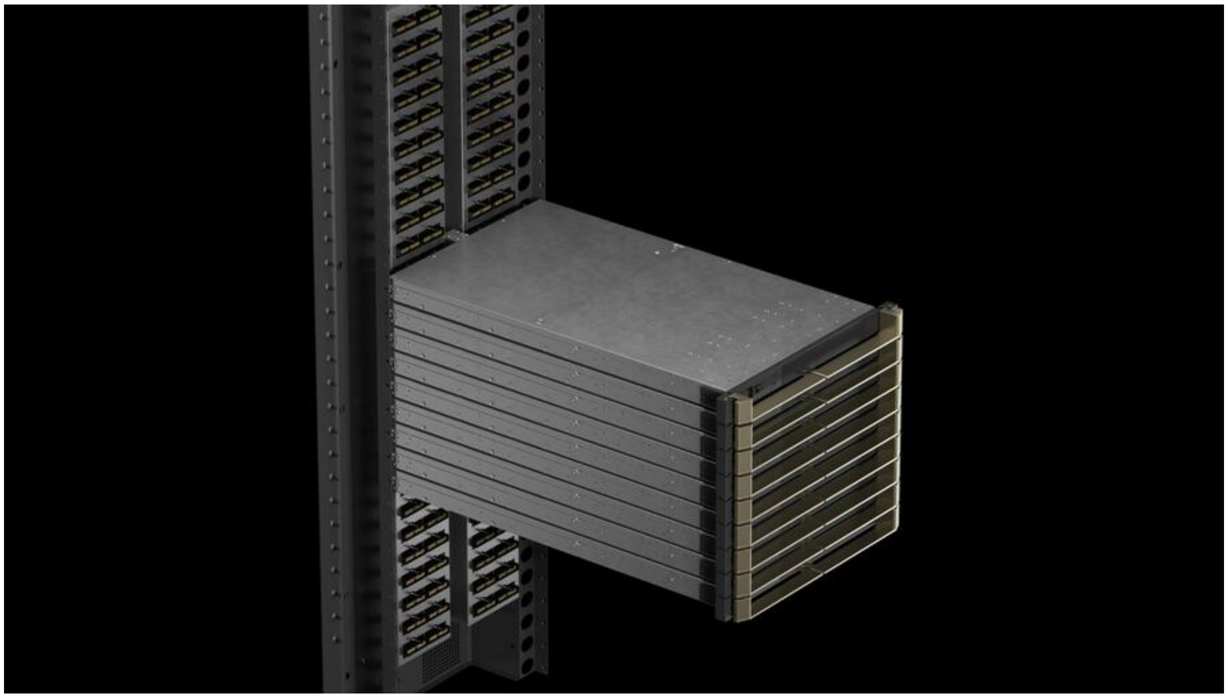


Figure 2-7. NVSwitch System of 9 trays inside an NVL72 rack

Each switch tray contains two NVSwitch chips for a total of 18 NVSwitch chips in the system. The network is arranged as a full crossbar such that every GPU is connected to every NVSwitch, and every NVSwitch to every GPU, providing a high-bandwidth path between any pair of GPUs.

Concretely, each GPU uses its 18 NVLink ports to connect to the 18 NVSwitch chips (one link to each switch). This means any GPU can reach any other GPU in at most two hops (GPU → NVSwitch → GPU), with enormous bandwidth along the way.

The bandwidth across the whole 72-GPU network, called the aggregate bisection bandwidth, is about 130 TB/s inside the rack. For perspective, that is many times higher than even a top-end InfiniBand cluster of similar scale. The design basically turns the entire rack into one giant shared-memory machine.

Multi-GPU Programming

From a programming model standpoint, one GPU can directly perform a memory load from another GPU's HBM memory over NVLink, or even from a Grace CPU in another node, and get the data in a matter of microseconds. In other words, the GPUs can operate in a globally shared memory space across the rack, very much like how cores inside a single server share memory. This is why the NVL72 is often described as "one big GPU" or "an AI supercomputer in a rack".

Software libraries like RDMA (remote direct memory access) abstract away the complexities of this global shared-memory space by providing simple programming interfaces such as atomic operations across GPUs. Distributed training and inference workloads need to synchronize and exchange information frequently across many GPUs. Traditionally, the GPUs are in different compute nodes and racks so the synchronization happens over relatively-slow network links like InfiniBand and Ethernet. This is often the bottleneck when scaling across many GPUs to support large AI models.

With the GB200 NVL72 system, those exchanges happen over NVLink and NVSwitch at a blistering pace. This means you can scale your training job or inference cluster up to 72 GPUs with minimal communication overhead. And since the GPUs spend far less time waiting for data from each other, overall throughput scales near-linearly up to 72 GPUs. By contrast, consider scaling the same job across an similarly-sized 72-GPU H100 cluster of 9 separate compute servers (each with 8 Hopper H100 GPUs). This configuration requires InfiniBand which will create network bottlenecks that greatly reduce the cluster's scaling efficiency.

Let's analyze and compare the GB200 NVL72 and 72-GPU H100 clusters using concrete numbers. Within a single NVL72 rack, GPU-to-GPU bandwidth is on the order of 100 GB/s+, and latency is on the order of 1–2 microseconds for a small message. Across a conventional InfiniBand network, bandwidth per GPU might be more like 20–80 GB/s - depending on how many NICs and their speed - and latency is likely 5–10 microseconds or more. The NVL72 network offers both higher throughput (2× or more per GPU) and lower latency (3-5x more) than the best InfiniBand networks for node-to-node GPU communication. In practical terms, an all-reduce collective operation which aggregates gradients across GPUs might consume 20–30% of iteration time on an InfiniBand-linked H100 cluster, but only take a 2-3% on the NVLink-connected NVL72 cluster.

This was demonstrated in a real-world test by NVIDIA which showed a single NVL72-based system gave about a 4× speedup over a similar H100-based cluster, largely thanks to communication efficiencies. The takeaway is that within a single NVL72 rack, communication is so fast that communication-bottlenecks become low-priority as they are almost completely eliminated. Whereas communication in traditional InfiniBand and Ethernet clusters is often the primary bottleneck and needs careful optimization and tuning at the software level.

In summary, one should design and implement software that exploits the NVL72 configuration by keeping as much of the workload's communication inside the rack ("intra-rack") as possible to take advantage of the high speed NVLink and NVSwitch hardware. Only go use the slower, InfiniBand-or-Ethernet-based communication between racks ("inter-rack") when absolutely necessary to scale beyond the NVL72's compute and memory resources.

In-Network Aggregations with NVIDIA SHARP

Another hardware-enabled optimization is NVIDIA SHARP which stands for Scalable Hierarchical Aggregation and Reduction Protocol. SHARP is integrated directly into NVSwitch hardware and capable of performing aggregations (e.g. all-reduce, all-gather, and broadcast) directly in the network hardware itself. The NVSwitch fabric combines partial results without the data needing to funnel through the GPUs. By offloading collective communication operations from the GPUs and CPUs to the switch hardware itself, SHARP dramatically boosts efficiency, lowers latency, and reduces the volume of data traversing the network.

SHARP's increased efficiency means that during distributed training, the heavy lifting of aggregating gradients or synchronizing parameters is handled by the NVSwitch's dedicated SHARP engines. The result is much-more efficient scaling across both intra-rack and inter-rack configurations. SHARP enables near-linear performance improvements even as the number of GPUs grows. This in-network computing capability is especially critical for training ultra-large models, where every microsecond saved on collective operations can translate into substantial overall speedups.

Multi-Rack and Storage Communication

Next, let’s discuss how an NVL72 rack talks to another NVL72 - or to an external storage system like a shared file system. As we have shown, inside the NVL72 rack, NVLink covers all GPU-to-GPU traffic. But outside the rack, it relies on more traditional networking hardware.

Each compute node in NVL72 is equipped with high-speed Network Interface Cards and a Data Processing Unit (DPU). In the GB200 NVL72 reference design, each node has four ConnectX InfiniBand NICs and one BlueField-3 DPU. Each of the four InfiniBand NICs runs at 400 Gb/s each. Combined, these NICs send and receive data on the order of 1.6 Tbit/s (1600 Gbit/s) to the outside world as shown in Figure 2-8.

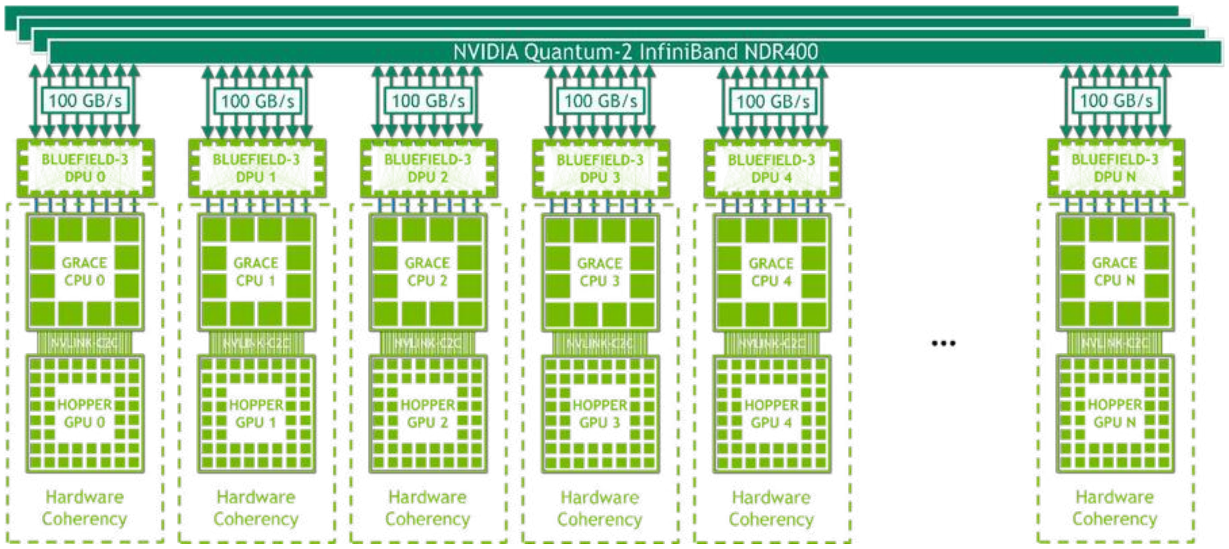


Figure 2-8. InfiniBand and BlueField DPU NICs

Across the 18 computes nodes in a single rack, the total outside-rack throughput is nearly 30 Tbit/s (1.6 Tbit/s * 18 compute nodes) of aggregate networking capacity when fully utilized. The key is that NVIDIA anticipates multi-rack deployments called "AI factories." – so they've made sure the NVL72 can plug into a larger network fabric via these 4 NICs per node.

TIP

NVIDIA also offers an Ethernet-based solution using Spectrum switches with RDMA over Converged Ethernet (RoCE) as an alternative for interconnect, called Spectrum-X.

The BlueField-3 DPU in each node helps offload networking tasks like RDMA, TCP/IP, and NVMe storage access. This makes sure the Grace CPU isn’t bogged down managing network interrupts. The DPU essentially serves as a smart network controller, moving data directly between NICs and GPU memory using NVIDIA’s RDMA software called GPUDirect which does not require CPU involvement. This is especially useful when streaming large datasets from a storage server as the DPU can handle the transfer and deposit data directly into GPU memory while the CPU focuses on other tasks like data preprocessing.

When scaling out to multiple NVL72 racks, NVIDIA uses Quantum-series InfiniBand switches. Multiple NVL72 racks can be interconnected using these InfiniBand switches to form a large cluster of NVL72 racks as shown in Figure 2-9.

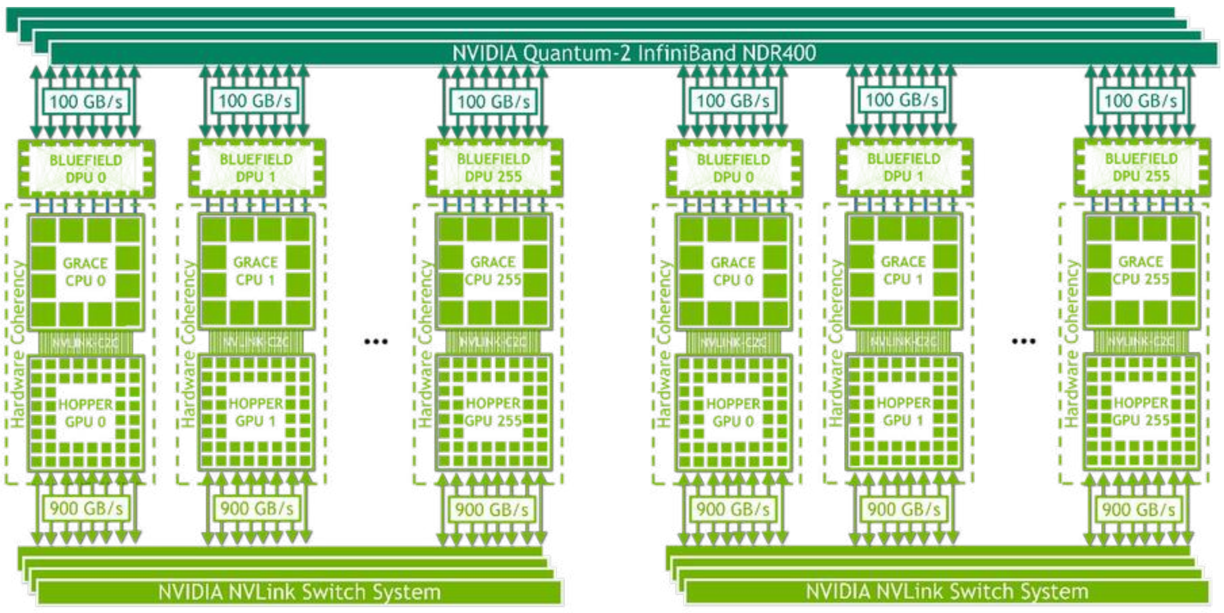


Figure 2-9. Scaling out NVL72 cluster with InfiniBand

For example, an 8-rack NVL72 cluster totaling 576 GPUs can be built with a second tier of InfiniBand switches, though the performance for cross-rack InfiniBand communication will be lower than the NVLink/NVSwitch communication within a single rack.

Pre-Integrated Rack Appliance

Because NVL72 is such a complex system, NVIDIA delivers it as a pre-integrated rack “appliance” in a single cabinet. It comes assembled with all 18 compute nodes, all 9 NVSwitch units, internal NVLink cabling, power distribution, and a cooling system. The idea is that an organization can order this as a unit that is ready to go when it arrives. One simply connects the rack to facility power, hooks up the water cooling interfaces, connects the InfiniBand cables to your network, and turns it on! There is no need to individually cable 72 GPUs with NVLink as NVIDIA has already done this inside the rack for you. Even the liquid cooling setup is self-contained as we’ll discuss soon.

This appliance approach accelerates deployment and ensures that the system is built correctly and validated by NVIDIA. The rack also includes its NVIDIA Base Command Manager cluster-management software - as well as SLURM and Kubernetes for cluster-job scheduling and orchestration. In other words, the

NVL72 rack is designed to be dropped-in to your environment and .

Co-Packaged Optics: Future of Networking Hardware

As networking data throughput rates climb to 800 Gbit/s and beyond, the traditional approach of pluggable optical modules starts to hit power and signal limits. This burns a lot of power as it converts electrical signals to optical at the switch. NVIDIA is addressing this by integrating co-packaged optics (CPO) into its networking gear.

With co-packaged optics, the optical transmitters are integrated right next to the switch silicon. This drastically shortens electrical pathways, enabling even higher bandwidth links between racks, reduces power draw, and improves overall communication efficiency. NVIDIA is integrating CPO into its Quantum-3 InfiniBand switches scheduled for late 2025 or 2026.

In practical terms, technologies like CPO are paving the way to connect hundreds and thousands of racks (“AI factories”) into a single unified fabric in which inter-rack bandwidth is no longer the bottleneck. Such optical networking advancements are crucial to the high-performance, inter-rack bandwidth needed to ensure that the network can keep up with the GPUs at ultra-scale.

To summarize, inside an NVL72 rack, NVIDIA uses NVLink and NVSwitch to create a blazingly fast, all-to-all connected network between 72 GPUs. These interconnects are so fast and uniform that the GPUs effectively behave like one unit. Beyond the rack, high-speed NICs (e.g. InfiniBand or Ethernet) connect the rack to other racks or to storage, with DPUs to manage data movement efficiently.

The NVL72 is an immensely powerful standalone system and a basic building block for larger AI supercomputers. NVIDIA partners with cloud providers like CoreWeave, Lambda Labs, AWS, Google Cloud, Azure, and Equinix to offer NVL72-based racks that companies can rent or deploy quickly.

The concept of an AI factory, a large-scale AI data center composed of multiple such racks, is now becoming reality. NVIDIA’s hardware and network roadmap is squarely aimed at enabling the AI factory vision. In short, the NVL72 shows how far co-design can go as the GPU, networking, and physical-rack hardware are built hand-in-hand to scale to thousands and millions of GPUs as seamlessly and efficiently as possible.

Compute Density and Power Requirements

The NVL72 rack is incredibly dense in terms of compute, which means it draws a very high amount of power for a single rack. A fully loaded NVL72 can consume up to about 120 kW of power under max load. This is NVIDIA's previous generation AI rack which consumed around 50–60 kW. Packing 72 bleeding-edge GPUs - and all the supporting hardware - into one rack pushes the limits of what data center infrastructure can handle.

To supply 120 kW to the NVL72 rack, you can't just use a single standard power feed. Data centers will typically provision multiple high-capacity circuits to feed this kind of power. For instance, one might push two separate power feeds into the rack for redundancy where each feed is capable of 60 kW. Under normal operation, the load is balanced between the feeds. And if one feed fails, the system could shed some load or throttle the GPUs to stay within the remaining feed's capacity. This kind of redundancy is important to protect against a blown circuit halting your multi-month training job.

Within the rack, power is distributed to the power supplies of each 1U compute node. The power is converted from AC to DC for the local electronics. Each compute node in the NVL72 contains 2 Grace-Blackwell superships which together consume on the order of 5-6 kW. With 18 compute nodes, the total power consumed is about 100 kW. The NVSwitch trays, network switches, and cooling pumps account for approximately 20 kW for a total of 120 kW consumed by the entire NVL72 rack.

The current used at typical data center voltages (e.g. 415 V 3-phase AC) is massive, so everything is engineered for high amperage. Operators have to carefully plan to host such a rack which often requires dedicated power distribution units (PDUs) and careful monitoring. Power transients are also a consideration as 72 GPUs, when ramping from idle to full power, could rapidly draw tens of kW of power in just milliseconds. A good design will include capacitors or sequencing to avoid large voltage drops.

The system might stagger the GPU boost clocks by tiny intervals, so they don’t all spike at exactly the same microsecond, smoothing out the surge. These are the kind of electrical engineering details that go into making a 120 kW rack manageable.

It’s not far-fetched to call this NVL72 rack, at the cutting edge of high-density compute, a mini power substation. 8 of these racks combined for 572 GPUs would draw nearly 1 MW of power (8 racks * 120 kW per rack) which is the entire capacity of a small data center! The silver lining is that although 120 kW is a lot in one rack, you are also getting a lot of work done per watt. In fact, if one NVL72 replaces several racks of older equipment, the overall efficiency is better. But you definitely need the infrastructure to support that concentrated power draw. And any facility hosting the NVL72 racks must ensure they have adequate power capacity and cooling as we will discuss next.

Liquid Cooling vs. Air Cooling

Cooling 120 kW in one rack is beyond the reach of traditional air cooling. Blowing air over 72 GPUs that each can dissipate 1000+ watts would require hurricane-like airflow and would be extremely loud and inefficient - not to mention the hot air exhaust would be brutal. As such, liquid cooling is the only practical solution for the NVL72 rack running at this power density.

The NVL72 is a fully liquid-cooled system. Each Grace-Blackwell superchip module and each NVSwitch chip has a cold plate attached. A cold plate is a metal plate with internal tubing that sits directly on the component. A water-based coolant liquid flows through the tubing to carry away heat. All these cold plates are linked by hoses, manifolds, and pumps that circulate the coolant throughout the system.

Typically, the rack will have quick-disconnect couplings for each node so you can slide a server in or out without spilling the coolant. The rack then has supply and return connections to the external facility's chilled water system. Often, there's a heat exchanger called a Cooling Distribution Unit (CDU) either built into the rack or immediately next to it. The CDU transfers heat from the rack's internal coolant loop to the data center's water loop.

The facility provides chilled water at 20-30°C. The water absorbs the heat through the heat exchanger. The warmed-up water is then pumped back into the chillers or cooling towers to be cooled again. In modern designs, they might even run warm water cooling in which chilled water comes into the system at 30°C and leaves at 45°C. The water can then be cooled by evaporative cooling towers without active refrigeration which improves overall efficiency. The point is, water or a liquid coolant, can carry far more heat per unit of flow than air, so liquid cooling is vastly more effective when running at high watts in small spaces.

By keeping the GPU and CPU temperatures much lower than they would be with air, liquid cooling reduces thermal GPU throttling. The GPUs can sustain their maximum clocks without hitting temperature limits. Also, running chips cooler improves reliability and even efficiency since power leakage is lower when running at lower temperatures.

The NVL72 keeps GPU temps in the 50-70°C range under load which is excellent for such power-hungry devices. The cold plates and coolant loops have been engineered very carefully to allow each GPU to dump 1000 W and each CPU to dump 500 W into the system. In addition, the coolant flow rate has to be sufficient to remove that heat quickly. A rough estimate shows on the order of 10+ liters per minute of water flowing through the system to dissipate 120 kW of power with a reasonable temperature increase.

The system undoubtedly has sensors and controls for coolant temperature, pressure, and leak detection. If a leak is detected from its drip or pressure-loss sensors, the system can shut down or isolate that section quickly. It’s recommended to use self-sealing connections - and perhaps a secondary containment tray - to minimize the risk of leaking fluids.

This level of liquid cooling in racks was once exotic, but it is now the standard for these large scale AI clusters. Companies like Meta, Google, and Amazon are all adopting liquid cooling for their AI clusters because air cooling simply cannot support the large amount of power drawn from these systems.

So while an NVL72 requires more facility complexity including liquid-cooling loops, many data centers are now built with liquid cooling in mind. The NVL72 rack, with its built-in internal liquid cooling, can be connected directly to the cooling loop.

One side effect of the internal liquid cooling is the weight of the rack. The NVL72 rack weighs on the order of 3000 lbs (1.3–1.4 metric tons) when filled with hardware and coolant. This is extremely heavy for a rack as it’s roughly the weight of a small car, but concentrated on a few square feet of floor. Data centers with raised floors have to check that the floor can support this load measured in pounds per square foot. Often, high-density racks are placed on reinforced slabs or supported by additional struts. Moving such a rack requires special equipment such as forklifts. This is all part of the deployment consideration as you’re installing an AI supercomputer which comes with its unique physical and logistical challenges.

NVIDIA also integrates management and safety features in the form of a rack management controller that oversees things like coolant pumps, valve positions, power usage, and monitors every node's status. Administrators can interface with it to do things like update firmware across all nodes, or to shutdown the system safely.

All these considerations illustrate that the NVL72 was co-designed with data center infrastructure in mind. NVIDIA worked on the compute architecture in tandem with system engineers who figured out power delivery and cooling, and in tandem with facility engineers who specified how to install and run these things. It's not just about fast chips - it's about delivering a balanced, usable system.

The payoff for this complexity is huge. By pushing the limits of power and cooling, NVIDIA managed to concentrate an enormous amount of compute into a single rack. That translates to unprecedented compute-per-square-foot and compute-per-watt. Yes, 120 kW is a lot of power, but per GPU or per TFLOP, it's actually efficient compared to spreading the same GPUs across multiple racks with less efficient cooling.

Performance Monitoring and Utilization in Practice

When you have a machine this powerful and expensive, you want to make sure you're getting the most out of it. Operating an NVL72 effectively requires careful monitoring of performance, utilization, and power. NVIDIA provides tools like DCGM (Data Center GPU Manager) that can track metrics on each GPU for things like GPU utilization %, memory usage, temperature, and NVLink throughput.

As a performance engineer, you'd keep an eye on these during training runs and inference workloads. Ideally, you want your GPUs to be near 100% utilized most of the time during a training job. If you see GPUs at 50% utilization, that means something is keeping them idle for half the time. Perhaps there is a data loading bottleneck or a synchronization issue.

Similarly, you can monitor the NVLink usage. If your NVLink links are saturating frequently, communication is likely the culprit. The BlueField DPUs and NICs have their own statistics that are monitored to ensure that you're not saturating your storage links when reading data. Modern systems like the NVL72 expose this telemetry.

Power monitoring is also crucial. At ~120 kW, even a small inefficiency or misconfiguration can waste a lot of power and money. The system likely lets you monitor power draw per node or per GPU. Administrators might cap the power or clocks of GPUs if full performance isn't needed, to save energy.

NVIDIA GPUs allow setting power limits. For instance, if you're running a smaller job that doesn't need every last drop of performance, you could dial down GPU clocks to improve efficiency - measured in performance per watt - and still meet your throughput requirement. This could save kilowatts of power in the process. Over weeks of training, this can translate to significant savings and cost efficiency.

Sharing and Scheduling

Another aspect is sharing and scheduling workloads on the NVL72. Rarely will every single job need all 72 GPUs. You might have multiple teams or multiple experiments running on subsets of GPUs. Using a cluster scheduler like SLURM or Kubernetes with NVIDIA's plugins, you can carve out say 8 GPUs for one user, 16 GPUs for another user, and 48 GPUs for yet another user - all within the same rack.

Furthermore, NVIDIA’s Multi-Instance GPU (MIG) feature lets you split a single physical GPU into smaller GPUs partitioned at the hardware level. For example, one Blackwell GPU with 192 GB of GPU memory could be split into smaller chunks to run many small inference jobs concurrently. For instance, the 192 GB of GPU memory can be split into four 24 GB partitions and two 48 GB partitions (4 * 24 + 2 * 48 = 192 GB). Similarly, the GPU’s SMs can be split into partitions along with the GPU HBM memory.

In practice, with such a large GPU, MIG might be used for inference scenarios where you want to serve many models on one GPU. The presence of the BlueField DPU also enables secure multi-tenancy as the DPU can act as a firewall and virtual switch. This isolates network traffic for different jobs and users. This means an organization could safely let different departments or even external clients use partitions of the system without interfering with each other - similar to how cloud providers partition a big server for multiple customers with secure multi-tenant isolation.

From a cost perspective, a system like NVL72 is a multi-million dollar asset, and it could consume tens of thousands of dollars in electricity per month. So you really want to do as much useful work, or goodput, as possible. If it sits idle, that’s a lot of capital and operational cost wasted. This is why monitoring utilization over time is important. You might track GPU-hours used vs. available hours.

If you find that the system is underutilized, you might want to consolidate workloads or offer it to additional teams for more projects. Some organizations implement a chargeback model where internal teams use their own budget to pay per GPU-hour of usage. This encourages efficient use and accounts for electricity and depreciation costs. Such transparency ensures that people value the resource. After all, 1 hour on all 72 GPUs could be, say, 72 GPU-hours which might cost hundreds of dollars worth of electricity and amortized hardware cost.

ROI of Upgrading Your Hardware

One might ask if it’s worth investing in this bleeding-edge hardware. When analyzing the return on investment (ROI), the answer often comes down to performance per dollar. If NVL72 can do the work of, say, four older-generation racks, it might actually save money long-term, both in hardware and power. Earlier in the chapter, we discussed how one Blackwell GPU could replace 2–3 Hopper GPUs in terms of throughput. This means if you upgrade, you might need fewer total GPUs for the same work.

Let’s analyze a quick case study. Suppose you currently have a hundred H100 GPUs handling your workload. You could potentially handle it with 50 Blackwell GPUs because each is more than twice as fast (or more using FP8/FP4). So you’d buy fifty instead of one hundred GPUs. And even if each Blackwell costs more than an H100, buying half as many could be cost-neutral or better. Power-wise, one hundred H100s might draw 70 kW whereas fifty Blackwells might draw 50 kW for the same work. This is a notable power savings.

Over a year, that power difference saves tens of thousands of dollars. Additionally, fewer GPUs means fewer servers to maintain, which means less overhead in CPUs, RAM, and networking for those servers provides even further savings. All told, an upgrade to new hardware can pay for itself in 1–2 years in some cases, if you have enough workload to keep it busy.

The math obviously depends on exact prices and usage patterns, but the point is that the ROI for adopting the latest AI hardware can be very high for large-scale deployments. Besides the tangible ROI, there are soft benefits like using a single powerful system instead of many smaller ones can simplify your system architecture. This simplification improves operational efficiency by lowering power consumption and reducing network complexity.

For example, not having to split models across multiple older GPUs due to memory limits can simplify software and reduce engineering complexity. Also, having the latest hardware ensures you can take advantage of the newest software optimizations and keep up with competitors who also upgrade. Nobody wants to be left training and serving models at half the speed of rivals. Upgrading will improve your performance while simultaneously enabling larger models, faster iterations, and quicker responses.

Running an NVL72 effectively is as much a software and management challenge as it is a hardware feat. The hardware gives you incredible potential, but it's up to the engineers to harness the full power of the hardware by monitoring performance, keeping utilization high, and scheduling jobs smartly.

The good news is NVIDIA provides a rich software stack to monitor and improve performance including drivers, profilers, container runtimes, and cluster orchestration tools. Throughout the rest of the book, we'll see how to optimize software to fully utilize systems like the GB200 NVL72. For now, the takeaway is that when you're given an AI system with exaflop-scale performance in a box, you need equally advanced strategies to make every flop and every byte count.

A Glimpse into the Future: NVIDIA's Roadmap

At the time of writing, the Grace-Blackwell NVL72 platform represents the state-of-the-art in AI hardware. But NVIDIA is, of course, already preparing the next leaps. It's worth briefly looking at NVIDIA's hardware roadmap for the coming few years, because it shows a clear pattern of scaling. NVIDIA intends to continue doubling down on performance, memory, and integration.

Blackwell Ultra and Grace-Blackwell Ultra (Late 2025)

In March 2025, NVIDIA announced an enhanced version of Blackwell called the Blackwell "Ultra" (B300) and corresponding Grace-Blackwell Ultra superchip (GB300). These Ultra versions are planned for late 2025, and they will be a drop-in upgrade to the NVL72 architecture. Each Blackwell Ultra GPU is expected to have about 50% more performance than the current B200, and also increase memory from 192 GB to 288 GB per GPU.

A GB300 superchip module would have 1 Grace CPU + 2 Blackwell Ultra GPUs with a total of 576 GB total HBM GPU memory (2 * 288 GB = 576 GB) plus the Grace CPU’s 480GB of DDR memory. That’s over 1 TB of memory per Grace-Blackwell Ultra superchip module. A 72 GPU rack of these superchips will have around 20 TB of combined GPU and CPU memory.

The intra-rack NVLink and NVSwitch networks in the GB300 NVL72 Ultra are based on the same NVLink 5 generation as the GB200 NVL72. The GB300 NVL72 Ultra delivers on the order of 1.1+ exaFLOPS of FP4 performance per rack – nudging it even further into the exascale regime.

The GB300 NVL72 Ultra Power and cooling draws 120 kW and uses liquid cooling, but the benefit is more compute and memory in the same footprint as the GB200 NVL72. NVIDIA claims a 1.5× improvement at the rack level for generative AI workloads relative to GB200, thanks to the beefier GPUs and the generous usage of FP4 precision. NVIDIA is targeting use cases like real-time AI agents and multi-modal models that demand maximum throughput. Essentially, the GB300 is an evolutionary upgrade as it uses the same architecture, but has more of everything including more SMs, more memory, and faster clocks.

Vera-Rubin Supership (2026)

Codenamed after scientists Vera Rubin, the Vera-Rubin superchip (VR200) is the next major architecture step expected in 2026. Vera is the ARM-based CPU successor to the Grace CPU, and Rubin is the GPU architecture successor to Blackwell. NVIDIA continues the superchip concept by combining one Vera CPU with two Rubin GPUs in a single module (VR200) similar to the Grace-Blackwell (GB200) configuration.

The Vera CPU is expected to use TSMC's 3nm semiconductor process with more CPU cores and faster memory such as of LPDDR6 or some form of GPU-like high-bandwidth memory (HBM) designed for CPUs. This could push CPU memory bandwidth toward 1 TB/s. The Rubin GPU is expected to use HBM4 memory with bandwidth per GPU jumping to maybe 13–14 TB/s.

NVLink is also expected to move to its 6th generation NVLink 6 which would double the CPU-to-GPU and GPU-to-GPU link bandwidth. There's also speculation that Vera-Rubin could allow more nodes per rack - or more racks per NVLink domain - to scale beyond the 576 GPU limit of the 8-rack GB200 NVL72 cluster, but the details are not confirmed as of this writing.

The bottom line is that the Vera-Rubin generation is yet another ~2× jump in most metrics including more cores, more memory, more bandwidth, and more TFLOPS. Rubin GPUs might increase SM counts significantly to around 200 SMs per die. This is up from 140-ish in Blackwell. This could further add efficiency improvements. They could also integrate new features like second-generation FP4 or even experimental 2-bit precisions, though that's just speculation at this point.

Another especially interesting possibility is that, because Rubin's expected 288 GB HBM RAM is still a bottleneck for large AI models, NVIDIA might incorporate some second-tier memory for GPUs directly in the GPU module. For instance, they may place some LPDDR memory directly on the base of the GPU module to act as an even larger, but slower, memory pool for the GPU - separate from Vera's CPU DDR memory. If this happens, a single GPU module could have >500 GB (256 GB HBM + 256 GB LPDDR) of total cache-coherent, unified memory. This would further blur the line between CPU and GPU memory, as GPUs would have a multi-tier memory hierarchy of their own.

Whether this happens in Rubin or not, it’s a direction to keep an eye on.

Overall, a Vera-Rubin rack might deliver maybe 1.5× to 2× the performance of a GB200 NVL72 running at similar - or slightly higher power. It might also increase total memory per rack significantly - perhaps 20+ TB of HBM across 72 Rubin GPUs (288 GB HBM per GPU * 72 GPUs) plus tens of TB of CPU memory. NVLink 6 within the rack would yield even less communication overhead, making 72 or more GPUs behaving as a single, tightly coupled unit.

Rubin Ultra and Vera-Rubin Ultra (2027)

Following the pattern, an “Ultra” version of Rubin (R300) and Vera-Rubing is expected to arrive a year after the original release. One report suggests that NVIDIA might move to a 4-die GPU module by then. This would combine two dual-die Rubin packages and put them together to yield a quad-die Rubin GPU. This hypothetical R300 GPU module would have 4 GPU dies on one package and likely 16 HBM stacks totaling 1 TB of HBM memory on a single R300 GPU module. The 4 dies together would roughly double the cores of the dual-die B300 module. This could provide nearly 100 PFLOPS of FP4 per GPU module!

Now, how do you integrate such modules? Possibly they would reduce the number of modules per rack but increase GPUs per module. In particular, they might have 144 of those dies across the rack which could be 36 modules of 4 dies each, or something equivalent. There is also mention of an NVL576 configuration implying 576 GPUs in one NVLink domain. By 2027, each rack could be pushing 3-4 exaFLOPs of compute performance and a combined 165 TB of GPU HBM RAM (288 GB HBM per Rubin GPU * 576 GPUs). While these numbers are speculative, the trajectory toward ultra-scale AI systems with a massive number of exaFLOPs for compute and terabytes for GPU HBM RAM is clear.

Feynmann GPU (2028) and Doubling Something Every Year

NVIDIA has code-named the post-Rubin generation as Feynmann which is scheduled for a 2028 release. Details are scarce, but the Fenmann GPU will likely move to an even finer 2nm TSMC process node. It will likely use HBM5 and include even more DDR memory inside the module. And perhaps it will double the number of dies from 4 to 8.

By 2028, it's expected that inference demands will surely dominate AI workloads - especially as reasoning continues to evolve in AI models. Reasoning requires a lot more inference-time computation than previous, non-reasoning models. As such, chip designs will likely optimize for inference efficiency at scale which might include more novel precisions, more on-chip memory, and on-package optical links to improve NVLink's throughput even further.

NVIDIA seems to be doubling something every generation, every year if possible. One year they double memory, another year they double the number of dies, another year they double interconnect bandwidth, and so on. Over a few years, the compound effect of this doubling is huge. Indeed, from 2024 to 2027, one will notice NVIDIA's aggressive upward curve by doubling from 72 GPUs → 144 dies, double NVLink throughput from 900 GB/s → 1.8 TB/s, and doubling memory per GPU from 192 GB → 288 GB.

NVIDIA repeatedly talks about "AI factories" where the racks are the production lines for AI models. NVIDIA envisions offering a rack as-a-service through its partners so companies can rent a slice of a supercomputer rather than building everything themselves. This trend will likely continue as the cutting-edge hardware will be delivered as integrated pods that you can deploy. And each generation allows you to swap in new pods to double your capacity, increase your performance, and reduce your cost.

For us as performance engineers, what matters is that the hardware will keep unlocking new levels of scale. Models that are infeasible today might become routine in a few years. It also means we'll have to continually adapt our software to leverage things like new precision formats, larger memory pools, and improved interconnects. This is an exciting time as the advancement of frontier models is very much tied to these hardware innovations.

Key Takeaways

The following innovations collectively enable NVIDIA's hardware to handle ultra-large AI models with unprecedented speed, efficiency, and scalability.

Integrated Superchip Architecture.

Nvidia fuses ARM-based CPUs (Grace) with GPUs (Hopper/Blackwell) into a single "superchip," which creates a unified memory space. This design simplifies data management by eliminating the need for manual data transfers between CPU and GPU.

Unified Memory Architecture.

The unified memory architecture and coherent interconnect reduce the programming complexity. Developers can write code without worrying about explicit data movement, which accelerates development and helps focus on improving AI algorithms.

Ultra-Fast Interconnects.

Using NVLink (including NVLink-C2C and NVLink 5) and NVSwitch, the system achieves extremely high intra-rack bandwidth and low latency. This means GPUs can communicate nearly as if they were parts of one large processor, which is critical for scaling AI training and inference.

High-Density, Ultra-Scale System (NVL72).

The NVL72 rack integrates 72 GPUs in one compact system. This consolidated design supports massive models by combining high compute performance with an enormous unified memory pool, enabling tasks that would be impractical on traditional setups.

Advanced Cooling and Power Management.

Operating at around 120 kW per rack, NVL72 relies on sophisticated liquid cooling and robust power distribution systems. These are essential for managing the high-density, high-performance components and ensuring reliable operation.

Significant Performance and Efficiency Gains.

Compared to previous generations such as the Hopper H100, Blackwell GPUs offer roughly 2 - 2.5× improvements in compute and memory bandwidth. This leads to dramatic enhancements in training and inference speeds - up to 30× faster inference in some cases - as well as potential cost savings through reduced GPU counts.

Future-Proof Roadmap.

Nvidia's development roadmap (including Blackwell Ultra, Vera-Rubin, Vera-Rubin Ultra, and Feynman) promises continual doubling of key parameters like compute throughput and memory bandwidth. This trajectory is designed to support ever-larger AI models and more complex workloads in the future.

Conclusion

The NVIDIA NVL72 system - with its Grace-Blackwell superchips, NVLink fabric, and advanced cooling – exemplifies the cutting edge of AI hardware design. In this chapter, we've seen how every component is co-designed to serve the singular goal of accelerating AI workloads. The CPU and GPU are fused into one unit to eliminate data transfer bottlenecks and provide a gigantic unified memory.

Dozens of GPUs are wired together with an ultra-fast network so they behave like one colossal GPU with minimal communication delay. And the memory subsystem is expanded and accelerated to feed the voracious appetite of the GPU cores. Even the power delivery and thermal management are pushed to new heights to allow this density of computing.

The result is a single rack that delivers performance previously only seen in multi-rack supercomputers. NVIDIA took the entire computing stack – chips, boards, networking, cooling – and optimized it end-to-end to allow training and serving massive AI models at ultra-scale.

But such hardware innovations come with challenges as you need specialized facilities, careful planning for power and cooling, and sophisticated software to utilize it fully. But the payoff is immense. Researchers can now experiment with models of unprecedented scale and complexity without waiting weeks or months for results.

A model that might have taken a month to train on older infrastructure might train in a few days on NVL72. Inference tasks that were barely interactive (seconds per query) are now a real-time (milliseconds) reality. This opens the door for AI applications that were previously impractical such as multi-trilion parameter interactive AI assistants and agents.

NVIDIA's rapid roadmap suggests that this is just the beginning. The Grace-Blackwell architecture will evolve into Vera-Rubin and Feynmann and beyond. As NVIDIA's CEO, Jensen Huang, describes, "AI is advancing at light speed, and companies are racing to build AI factories that can scale to meet the demand."

The NVL72 and its successors are the core of the AI factory. It's the heavy machinery that will churn through mountains of data to produce incredible AI capabilities. As performance engineers, we stand on the shoulders of this hardware innovation. It gives us a tremendous raw capability as our role is to harness this innovation by developing software and algorithms that make the most of the hardware's potential.

In the next chapter, we will transition from hardware to software. We'll explore how to optimize the operating systems, drivers, and libraries on systems like NVL72 to ensure that none of this glorious hardware goes underutilized. In later chapters, we'll look at memory management and distributed training/inference algorithms that complement the software architecture.

The theme for this book is co-design. Just as the hardware was co-designed for AI, our software and methods must be co-designed to leverage the hardware. With a clear understanding of the hardware fundamentals now, we're equipped to dive into software strategies to improve AI system performance. The era of AI supercomputing is here, and it's going to be a thrilling ride leveraging it to its fullest.

Let's dive in!

Chapter 3. OS, Docker, and Kubernetes Tuning for GPU-based Environments

Even with highly optimized GPU code and libraries, system-level bottlenecks can limit performance in large-scale AI training. The fastest GPU is only as good as the environment feeding it data and instructions. In this chapter, we explore how to tune the operating system and container runtime to let GPUs reach their full potential.

In this chapter, we begin by exploring the foundational GPU software stack. We then dive into key CPU and memory optimizations such as NUMA affinity and huge pages. These ensure that data flows efficiently from storage through the CPU to the GPU. In parallel, we discuss critical GPU driver settings like persistence mode, Multi-Process Service, and Multi-Instance GPU partitions. These help maintain maximum GPU utilization by reducing overhead and synchronizing resources effectively.

Using solutions like the NVIDIA Container Toolkit, Container Runtime, Kubernetes Topology Manager, and the Kubernetes GPU Operator, you can create a unified and highly-optimized software stack for GPU environments. These solutions enable efficient resource allocation and workload scheduling across single-node and multi-node GPU environments - and ensures GPU capabilities are fully utilized.

Along the way, you'll build intuition for why these optimizations matter. In essence, they minimize latency, maximize throughput, and ensure your GPUs are constantly fed with data and operating at their peak performance. The result is a robust, scalable system delivers significant performance gains for both training and inference workloads.

Operating System

The operating system is the foundation that everything runs on. GPU servers usually run a Linux distribution such as Ubuntu and Red Hat with an OS kernel configured to support the specific hardware. The NVIDIA driver installs kernel modules that allow the OS to interface with GPUs by creating device files like `/dev/nvidia0`. The OS also manages CPU scheduling, memory allocation, networking, and storage. All of which need to be tuned for better GPU usage.

The OS needs to be configured not to interfere with GPU tasks. For instance, by default Linux might be overly aggressive in swapping memory or might not be aware of Non-Uniform Memory Access (NUMA) when scheduling threads. Both can hurt GPU performance. Part of our job is to adjust those OS settings to create a smooth runway for the GPUs.

A GPU-focused server might also want to run additional daemons, or background processes, such as the NVIDIA Persistence Daemon to keep GPUs initialized, the Fabric Manager on systems with NVSwitch to manage the GPU interconnect topology, and NVIDIA Data Center GPU Manager (DCGM) for monitoring GPU system health metrics.

GPU Driver and Software Stack

Running a multi-petaFLOP GPU cluster involves more than just writing high-level PyTorch, TensorFlow, or JAX code. There is a whole software stack underpinning GPU operations, and each layer can affect performance. At the base is the NVIDIA GPU driver which interfaces between the Linux OS and the GPU hardware. The driver manages low-level GPU operations including memory allocation on the device, task scheduling on GPU cores, and partitioning the GPU for multi-tenant usage.

GPU Driver

The GPU driver turns on the GPUs' features and keeps the hardware fed with work. It's important to keep the driver up-to-date as new driver versions often provide performance improvements and additional support for the latest CUDA features. Tools like `nvidia-smi` come with the driver and allow you to monitor temperatures, measure utilization, query error-correcting code (ECC) memory status, and enable different GPU modes like persistence mode.

CUDA Toolkit and Runtime

On top of the driver sits the CUDA runtime and libraries called the CUDA Toolkit. The toolkit includes the CUDA compiler, `nvcc`, used to compile CUDA C++ kernels as we will see in the next chapter. When compiled, CUDA programs link against the CUDA runtime (`cudart`). The CUDA runtime communicates directly with the NVIDIA driver to launch work and allocate memory on the GPU.

Additionally, the CUDA toolkit provides many optimized neural-network libraries including cuDNN for neural network primitives, cuBLAS for linear algebra, and NCCL for multi-GPU communication. These CUDA libraries provide high-performance building blocks for your higher-level code so that you don’t have to reinvent primitives like matrix multiply, for example, from scratch. We will cover CUDA programming and optimizations in more detail in upcoming chapters.

It’s important to understand your GPU’s compute capability (CC) and to ensure that you’re using the latest version of the CUDA Toolkit that matches this compute capability. The CUDA toolkit will perform just-in-time compilation of your GPU kernels, optimize your code for the specific GPU architecture, and upgrade your code to the latest hardware.

TIP

An important feature of GPU programming is that the generated PTX (a.k.a assembly code for GPUs) is backward-compatible with older NVIDIA GPU hardware and forward-compatible with newer hardware. This is a big selling point of the NVIDIA programming model, and it's something that Jensen Huang, NVIDIA's CEO, reiterates with every new hardware release.

C++ and Python CUDA Libraries

While most of the CUDA toolkit libraries are C++ based, more and more Python-based libraries are emerging from NVIDIA that are prefixed with "Cu" and built upon the C++ toolkit. For instance, CuTile and CuPyNumeric are Python libraries launched in early 2025. They are targeted at lowering the barrier to entry for Python developers to build applications for NVIDIA GPUs using CUDA.

CuTile is a Python library designed to simplify working with large matrices on GPUs by breaking them into smaller, more manageable sub-matrices called "tiles". It provides a high-level, tile-based abstraction that makes it easier to perform block-wise computations, optimize memory access patterns, and efficiently schedule GPU kernels. By dividing a large matrix into tiles, CuTile helps developers take full advantage of the GPU's parallelism without needing to manage low-level details manually. This approach can lead to improved cache usage and overall better performance in applications that require intensive matrix computations.

CuPyNumeric is a drop-in replacement for the popular `numpy` Python library that utilizes the GPU. It provides nearly the same functions, methods, and behaviors as NumPy, so developers can often switch to it with minimal changes to their code. Under the hood, CuPyNumeric leverages CUDA to perform operations in parallel on the GPU. This leads to significant performance gains for compute-intensive tasks such as large-scale numerical computations, matrix operations, and data analysis. By offloading work to the GPU, CuPyNumeric accelerates computation and improves efficiency for applications handling massive datasets. Its goal is to lower the barrier for Python developers to harness GPU power without having to learn a completely new interface, making it a powerful drop-in alternative to NumPy for high-performance computing.

PyTorch and Higher-Level AI Frameworks

Some popular Python-based frameworks built on CUDA are PyTorch, TensorFlow, JAX, and Keras. These frameworks provide high-level interfaces for deep learning while leveraging the power of NVIDIA GPUs. This book primarily focuses on PyTorch.

When you perform operations on PyTorch tensors using GPUs, they are moved from the CPU to the GPU in what appears to be a single, Python call. However, this single call is actually translated into a series of calls to the CUDA runtime utilizing various CUDA libraries. When you perform matrix multiplications, for example, PyTorch delegates these tasks to libraries such as cuBLAS. cuBLAS is part of the CUDA Toolkit and optimized for GPU execution. Behind the scenes, PyTorch ensures that operations like forward and backward passes are executed using low-level, optimized CUDA functions and libraries.

In short, PyTorch abstracts away the complexity of direct CUDA programming, allowing you to write intuitive Python code that ultimately calls highly optimized CUDA routines, delivering both ease of development and high performance. We will discuss CUDA programming and optimizations in chapters 4 and 5 - as well as PyTorch optimizations in Chapter 6.

All of these components – OS, GPU Driver, CUDA Toolkit, CUDA libraries, and PyTorch – must work together to create the ideal GPU-based development environment. When a researcher submits a training job, the scheduler reserves nodes, the OS provides the GPU devices and memory allocations using the NVIDIA driver, the container provides the correct software environment including the optimized, hardware-aware CUDA libraries. The user code (e.g. PyTorch, TensorFlow, JAX) uses these CUDA libraries which ultimately communicate with the driver and hardware.

The optimizations described in this chapter are designed to make each layer of this stack as efficient as possible. They will help the GPUs stay busy with actual useful training and inference work - instead of the GPU waiting on the CPU, waiting for memory or disk I/O, or waiting on other GPUs to synchronize. A well-tuned system ensures that models split across dozens of GPUs are not bottlenecked by I/O or OS overhead. System-level tuning is often overlooked in favor of model optimizations, but system-level optimizations can yield substantial performance gains. In some cases, you can get double-digit percentage improvements with small tweaks to your OS-level configuration. At the scale of a big AI project, this can save tens or hundreds of thousands of dollars in compute time.

Configuring the CPUs and OS for GPU Environments

One of the most common reasons that GPUs don’t reach full utilization is that the CPU isn’t keeping them fed with useful work. In a typical training loop, the CPU is responsible for preparing the next batch of data including loading the data from disk, tokenizing the data, transforming it, etc. In addition, the CPU is responsible for dispatching GPU kernels and coordinating between threads and processes. If these CPU-side tasks are slow or if the OS schedules them poorly, the expensive GPU can find itself idle, twiddling its transistors, and waiting for the next task or batch of data. To avoid this, we need to optimize how the CPU and OS handle GPU workloads. This includes careful CPU affinity so the right CPU cores are working on the right data, proper memory-allocation strategies to avoid NUMA penalties, and other impactful OS-level settings to eliminate unnecessary delays as we’ll discuss next.

NUMA Awareness and CPU Pinning

Modern server CPUs have dozens of cores and are often split into multiple Non-Uniform Memory Access nodes. A NUMA node is a logical grouping of CPUs, GPUs, NICs, and memory that are physically close to each other. Being aware of the system's NUMA architecture is important for performance tuning. Accessing resources within a single NUMA node is faster than accessing resources in other NUMA nodes.

For example, if a process running on a CPU in NUMA node 0 needs to access a GPU in NUMA node 1, it will need to send data across an inter-node link which will incur higher latency. In fact, memory access latency can nearly double when crossing to the other NUMA nodes as one experiment showed. This experiment measured 80 ns for local NUMA node memory access versus 139 ms when accessing memory across NUMA nodes. This is a huge difference. By binding a process to a CPU on the same NUMA node as its GPU, we can avoid this extra overhead. The key idea is to keep CPU execution and memory access local to the GPU that it's serving.

TIP

It's worth noting that, by default, the Linux scheduler will not use a NUMA-aware scheduling algorithm.

To prevent this, it's crucial to "pin" processes or threads to specific CPUs that are connected to the same NUMA node as the GPU. This type of CPU affinity is often called CPU pinning.

Suppose you have 8 GPUs in a node, with 4 GPUs connected to NUMA node 0 and the other 4 to NUMA node 1. If you launch 8 training processes, one per GPU, you should bind each training process to a CPU core - or set of CPU cores - connected to the same NUMA node as the GPUs. In this case, GPUs 0-3’s are connected to NUMA node 0 and GPUs 4-7’s are connected to NUMA node 1’s cores as shown in Figure 3-1.

Figure 3-1. 8 GPUs in a node, with 4 GPUs connected to NUMA node 0 and the other 4 to NUMA node 1

This way, when a CPU process wants to feed data to GPU 5, it should be running on a CPU connected to NUMA node 1 since GPU 5 is connected to NUMA node 1. Linux provides tools to do this including `numactl --cpunodebind=<node>` which launches a process pinned to the given NUMA node. You can also use `taskset` to pin processes to specific core IDs. Here is an example using `numactl` to bind the `train.py` script to a CPU running in the same NUMA node 1 as GPU 5.

```
numactl --cpunodebind=1 --membind=1 python train
```

The `--membind` part in that command is also important. We will discuss NUMA-friendly memory allocation and memory pinning in the next section.

Many deep learning frameworks also let you set thread affinities programmatically. For instance, PyTorch's `DataLoader` allows setting CPU affinities for CPU-based worker processes as shown here.

```
import os
import psutil
import torch
import torch.distributed as dist
from torch.nn.parallel import DistributedDataPara
from torch.utils.data import DataLoader, Dataset

# Set CPU affinity for the main process (the DDP
def set_main_process_affinity():
    available_cpus = list(range(psutil.cpu_count
    rank = int(os.environ.get("RANK", "0"))
    # For demonstration, pin the main process to
    main_affinity = [available_cpus[rank % len(av
    process = psutil.Process(os.getpid())
    process.cpu_affinity(main_affinity)
    print(f"Main process (rank {rank}) pinned to

# Set CPU affinity for each DataLoader worker
def worker_init_fn(worker_id):
    available_cpus = list(range(psutil.cpu_count
    rank = int(os.environ.get("RANK", "0"))
    # Example mapping: assign each worker a CPU b
    assigned_cpu = available_cpus[(rank * 10 + wo
    process = psutil.Process(os.getpid())
    process.cpu_affinity([assigned_cpu])
    print(f"DataLoader worker {worker_id} (rank {

# main() function that combines DDP with a DataLo
def main():
    # Initialize the distributed process group as
    ...

    # Set affinity for the main DDP process
    set_main_process_affinity()

    # Create your dataset (Fictitious dataset)
    dataset = Dataset()

    # Create DataLoader with the worker_init_fn t
    dataloader = DataLoader(
        dataset,
        batch_size=32,
        num_workers=4,
        pin_memory=True,
        worker_init_fn=worker_init_fn # set affi
    )

    # Create and move your model to the current 
    model = torch.nn.Linear(224 * 224 * 3, 10)  
    model = model.to("cuda")

    # Wrap the model in DistributedDataParallel
    ddp_model = DDP(model, device_ids=[
torch.cuda.current_device()
])

    # Training loop
    ...
```

The `set_main_process_affinity()` function assigns the main DDP process to a specific CPU core based on the process rank. The `worker_init_fn()` function is passed to the DataLoader so that each worker subprocess is pinned to a specific CPU core. This minimizes cross-NUMA traffic and can improve data preprocessing and transfer efficiency.

In practice, pinning can eliminate unpredictable CPU scheduling behavior. It ensures that a critical thread such as a data-loading thread for your GPU doesn't suddenly get migrated by the OS to a core on a different NUMA node in the middle of training or inferencing. In practice, it's possible to see 5-10% training throughput improvements just by eliminating cross-NUMA traffic and CPU core migrations. This also tends to reduce performance jitter and variance.

Many high-performance AI systems will also disable CPU hyper-threading to get more predictable performance per CPU core. Some even reserve a few cores exclusively for OS background tasks so that the remaining cores are dedicated exclusively to the training or inference workloads.

It's important to note that for integrated CPU-GPU superchips like NVIDIA's Grace-Blackwell, many of the traditional concerns about CPU-to-GPU data transfer are alleviated because the CPU and GPU share the same physical memory and are part of a unified architecture. This means that issues like cross-NUMA delays are minimized, and the data can flow more directly between the CPU and GPU.

TIP

It’s not a coincidence that NVIDIA tackled the CPU-to-GPU bottleneck in their hardware by combining the CPU and GPU onto a single superchip. Expect NVIDIA to keep addressing more and more bottlenecks through hardware innovations.

Even with the tightly-coupled CPU-GPU superchip architecture, it’s still important to optimize the stack by ensuring that the hardware and software are configured properly so that the integrated system operates at peak efficiency. Even in these tightly coupled architectures, you want to minimize any unnecessary delays in data handling to keep the GPU fully utilized. This includes configuring huge pages, using efficiency prefetching, and pinning memory as you will see in the next sections.

NUMA-Friendly Memory Allocation and Memory Pinning

By default, a process will allocate memory from the NUMA node of the CPU it’s currently running on. So if you pin a process to NUMA node 0, its memory will naturally come from NUMA node 0’s local RAM which is ideal. However, if the OS scheduler migrates threads - or if some memory got allocated before you did the pinning - you could end up with the non-ideal scenario in which a process running in NUMA node 0 is using memory from NUMA node 1. In this case, every memory access has to hop to the other NUMA node, negating the benefit of CPU pinning.

To avoid this, the `numactl --membind` option forces memory allocation from a specific NUMA node. In code, there are also NUMA APIs or even environment variables that can influence this configuration. The general rule is to keep memory close to the CPU which is close to the GPU. That way the chain of data movement from memory to CPU to GPU is all within a single NUMA node. Here is the same example as before, but with `--membind=1` to force memory allocation from the preferred NUMA node that includes NUMA node 1.

```
numactl --cpunodebind=1 --membind=1 python train
```

Another aspect of memory for GPUs is called page pinning, page locking, or memory pinning. When transferring data to the GPU, pinned (page-locked) host memory can dramatically improve throughput. Normally, the OS can decide to swap memory pages in and out - or move them around as needed. However, if you allocate pinned memory, the OS guarantees those memory pages will stay in physical RAM and not be swapped out or moved. Memory pinning allows the GPU to perform direct memory access (DMA) transfers at high speed, without the overhead of the OS potentially getting in the way. Copying from pinned host CPU memory to a GPU is often 2–3× faster than from regular pageable CPU memory.

Deep learning frameworks provide options to use pinned memory for data loaders. For example, PyTorch's `DataLoader` has a flag `pin_memory=True` which, when true, means the batches loaded will be placed in pinned RAM. This speeds up the `tensor.to(device)` operations because the CUDA driver doesn't have to pin pages on the fly. It's especially beneficial when you are using large batch sizes or reading a lot of data each iteration. Many practitioners have noticed that just turning on `pin_memory=True` in PyTorch can significantly improve performance by reducing data transfer bottlenecks.

TIP

The OS has a limit on how much memory a user can lock (pin). This is set with the `ulimit -l <max locked memory>` command. If you plan to use large pinned buffers, ensure this limit is high - or set to unlimited for your user - otherwise the allocation might fail. Typically, one sets it to unlimited for large AI workloads and HPC applications.

Transparent Huge Pages

In addition to pinning memory and binding it to NUMA nodes, we should talk about Transparent Huge Pages (THP). Linux memory management typically uses 4 KB pages, but managing millions of tiny pages is inefficient when you have processes using tens or hundreds of gigabytes of memory as in the case of deep learning datasets, prefetched batches, model parameters, etc.

Huge pages - 2 MB or even 1 GB pages - can reduce the overhead of virtual memory management by making memory chunks bigger. The main benefits are fewer page faults and less pressure on the Translation Lookaside Buffer (TLB). The TLB is a cache that the CPU uses to map virtual addresses to physical ones. Fewer, larger pages means the TLB can cover more memory with the same number of entries, reducing misses.

It’s generally recommended that you enable huge pages for big-memory workloads. Linux uses THP, which tries to automatically use 2 MB pages whenever possible. It’s usually enabled by default in modern distributions using either `madvise` or `always` mode. You can check the setting by reading `/sys/kernel/mm/transparent_hugepage/enabled`.

For most deep learning training jobs, it's beneficial to enable THP so that it's transparent to your program. In this case, you don't have to change code, but you'll gain a boost in CPU efficiency. Note that the gains from huge pages aren't massive in every case. You might see a few percent improvement in throughput due to fewer page faults. For extremely large memory usage scenarios, one can reserve explicit huge pages using `hugetlbfs`, the Linux pseudo-filesystem, for allocating 1 GB pages. However, this requires more manual setup and configuration. Enabling THP is an easier, simpler win. Once it's on, the OS will back large allocations with 2 MB pages automatically, reducing kernel overhead.

Now, beyond CPU and memory pinning, there are a few other OS-level tweaks worth mentioning. These include thread scheduling, virtual memory management, filesystem caching, and CPU frequency settings.

Scheduler and Interrupt Affinity

On a busy system, you want to make sure that important threads such as data-pipeline threads aren't interrupted frequently. Linux by default uses the Completely Fair Scheduler (CFS) that works well for most cases. But if you have a very latency-sensitive thread that feeds the GPU with data, for example, you could consider using real-time first-in-first-out (FIFO) or round-robin (RR) priority scheduling for that thread. This would ensure the runs without being preempted by normal threads. However, use this with caution as real-time threads can starve other processes if not managed properly. In practice, however, if you've pinned your threads to dedicated cores, you often don't need to mess with real-time thread priorities, but it's worth keeping an eye on.

Another trick is to isolate cores entirely for your process using `cset` or kernel boot parameters like the `isolcpus` kernel option. In this case, the OS scheduler leaves those CPU cores for you to use as your program wishes.

Additionally, you can bind hardware interrupts to specific cores in a NUMA-aware manner to avoid cross-NUMA-node interrupts that could evict useful cache data on the other NUMA node. If your GPU or NIC running in a NUMA node 0 generates hardware interrupts, you'd like those to be handled by a core on the same NUMA node. Otherwise, a random core from another NUMA node has to communicate across NUMA nodes. Tools like `irqbalance` can be configured so that, let's say, the interrupts from an InfiniBand card in NUMA node 0 are handled by a CPU core in NUMA node 0.

Virtual Memory and Swapping

It goes without saying, but you should always try to avoid memory swapping. If any part of your process's memory gets swapped to disk, you will see a catastrophic, multiple-orders-of-magnitude slowdown. GPU programs tend to allocate a lot of host memory for data caching. If the OS decides to swap some data out of memory and onto disk, the GPU will experience huge delays when it needs to access that data.

We recommend setting `vm.swappiness = 0` which tells Linux to avoid swapping except under extreme memory pressure. Also, ensure you have enough RAM for your workload or put limits to prevent overcommit. Another related setting is `ulimit -l` as mentioned earlier for pinned memory. If you want to prevent memory from swapping, you should set that limit high or you may experience excessive memory swapping. Again, typically one sets this limit to unlimited for large AI workloads that utilize a lot of memory.

Filesystem Caching and Write-Back

A best practice for large training jobs is to write frequent checkpoints to disk in case you need to restart a failed job from a known good checkpoint. During checkpointing, however, huge bursts of data might fill up the OS page cache and cause stalls. Adjusting `vm.dirty_ratio` and `vm.dirty_background_ratio` can control how much data can be buffered before writes flush to disk. For instance, if you're writing multi-gigabyte checkpoints, you might want to allow a lot of dirty cache to accumulate and flush in the background, so your training process doesn't block on file writes. Another option is to perform checkpointing in a separate thread. A more recent option in PyTorch, is to write distributed checkpoint partitions from nodes across the cluster. In this case, the checkpoint partitions will be combined when the checkpoint is loaded after a failed-job restart.

CPU Frequency and C-states

By default, many compute nodes will run CPUs in a power-saving mode which either downclocks a CPU or puts it to sleep when it's idle. This helps save energy, reduce heat, and lower cost. During model training, the CPUs might not always be 100% utilized as the GPUs are churning through the final batches of its dataset. However, these power management features could cause extra latency when the system wakes the CPUs up again when new work arrives.

For maximum and consistent performance, AI systems often configure the CPU frequency governor to “performance” mode which keeps the CPU at max frequency all the time. This can be done via `cpupower frequency-set` or in BIOS.

Likewise, disabling deep C-states can keep cores from going into a low-power sleep state. CPU C-states are power-saving modes defined by the system’s ACPI specification. When a CPU core is idle, it can enter a C-state to save energy. The deeper the C-state, the more power is saved, but the longer it may take for the core to “wake up” when work arrives. Disabling deeper C-states can remove excessive latency spikes. C0 is active, everything above C0 represents a deeper state of sleep.

Essentially, we can trade a bit of extra power draw for more responsive CPU behavior. In a training scenario where GPUs are the big power consumers, a bit more CPU power usage is usually fine if it keeps the GPUs fed. For example, if a data loader thread sleeps waiting for data and the CPU goes into the deep C6 state in which significant portions of the CPU is powered down to maximize energy savings.

If the CPU enters a deeper sleep state, it might take a few microseconds to wake up. While this is not a long time, many microseconds can add up and can cause GPU bubbles if not managed properly. Bubbles are periods of time when the GPU is waiting for the CPU to resume data processing. By keeping the CPU ready, we reduce such hiccups. Many BIOSes for servers have a setting to disable C-states - or at least limit them.

To summarize CPU and OS tuning, you should bind your workload to the hardware topology, use NUMA to your advantage, and eliminate OS interference. Make sure to pin each GPU's work to the nearest CPU cores and memory. It's recommended to use huge pages and locked memory to speed up memory operations. You should always turn off anything that might introduce unpredictable latency such as excess context switching, frequency scaling, and memory-to-disk swapping.

The result should be that your CPUs deliver data to the GPUs as fast as the GPUs can consume it, without the OS scheduling things on the wrong core or taking CPU cycles away at the wrong time. On a well-tuned GPU server, you might notice that CPU usage isn't extremely high since the GPUs are doing the heavy lifting, but CPU usage should be consistent and aligned with the GPUs. The CPUs stay active enough to prepare the next batch while the current one is processing. Each GPU's utilization graph stays near the top, only dipping when absolutely necessary for synchronization points - and not because the GPU is waiting on data or stuck on a slow CPU.

GPU Driver and Runtime Settings for Performance

We’ve optimized the CPU side, but there are also important settings for the GPU driver and runtime that can affect performance - especially in multi-GPU and multi-user scenarios. NVIDIA GPUs have a few knobs that, when tuned properly, can reduce overhead and improve how multiple workloads share a GPU. We’ll cover GPU persistence mode, the Multi-Process Service , Multi-Instance GPU partitions, and a couple of other considerations like clock settings, ECC memory, and out-of-memory behavior.

GPU Persistence Mode

By default, if no application is using a GPU, the driver may put the GPU into a lower-power state and unload some of the driver’s context. The next time an application comes along and wants to use the GPU, there’s a cost to initialize it. This can take on the order of a second or two for the driver to spin everything up. The initialization overhead can negatively impact performance for workloads that periodically releases and re-acquires the GPU. For instance, consider a training cluster where jobs are starting and stopping frequently. Or a low-volume inference cluster that has to wake up the GPU every time a new inference request arrives. In both of these cases, the overhead will reduce overall workload performance.

Persistence mode is a setting enabled by `nvidia-smi -pm 1` that keeps the GPU driver loaded and the hardware in a ready state even when no application is active. Essentially, it requests that the system does not fully power down the GPU when idle. The GPU stays awake so the next job has zero startup delay.

On AI clusters, it’s common to just enable persistence mode on all GPUs at server boot time. This way, when a job begins, the GPUs are already initialized and can start processing immediately. It won’t make your actual compute any faster as it doesn’t speed up the math operations, but it shaves off job-startup latency and prevents cold start delays.

GPU persistence mode also helps with interactive usage as without persistence, the first CUDA call you make after some idle time might stall while the driver reinitializes the GPU. With persistence on, that call returns quickly. The only downside is a slightly higher idle power draw since the GPU stays in a higher readiness state, but for most data center GPUs that’s an acceptable trade-off for performance consistency. Once GPU persistence mode is set by an admin with `sudo` access, you can enjoy the benefits and move on to tackle other optimizations.

Multi-Process Service

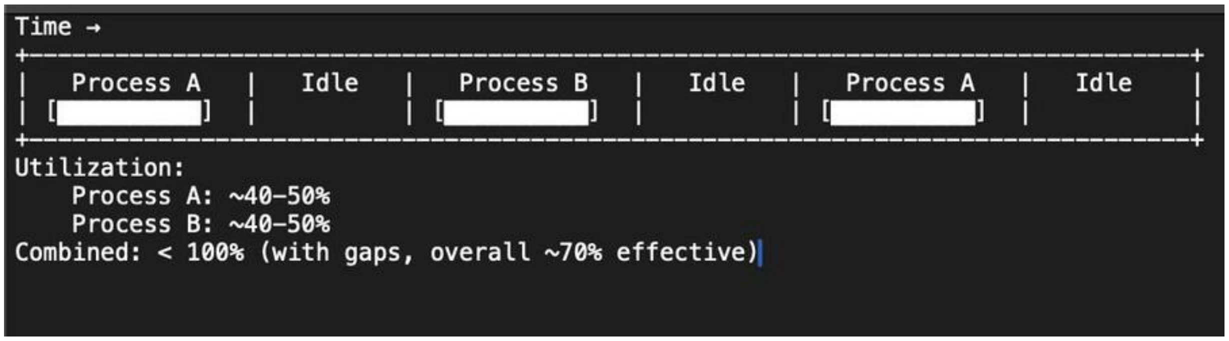
Normally, when multiple processes share a single GPU, the GPU’s scheduler time-slices between them. For example, if two Python processes each have some kernels to run on the same GPU, the GPU might execute one process’s kernel, then the other process’s kernel, and so on. If those kernels are short and there’s an idle gap between them, the GPU can end up underutilized as it’s doing “ping-pong” context switches and not overlapping the work.

NVIDIA’s Multi-Process Service (MPS) is a feature that creates a sort of umbrella under which multiple processes can run on the GPU concurrently and without strict time slicing. With MPS, the GPU can execute kernels from different processes at the same time as long as the GPU resources (streaming multiprocessors, tensor cores, etc.) are available. MPS essentially merges the contexts of the processes into one scheduler context. This way, you don’t pay the full cost of switching and idling between independent processes.

When is MPS useful? For model training, if you normally run one process per GPU, you might not use MPS. But if you have scenarios like running many inference jobs on one big GPU, MPS is a game changer. Imagine you have a powerful GPU or GPU cluster but your inference job - or set of multiple inference jobs - doesn't fully use it. For instance, consider running four separate inference jobs on one 40 GB GPU, each using 5-10 GB and only 30% of GPU compute. By default, each inference job gets a timeslice so at any moment, only one job's work is actually running on the GPU. That leaves the GPU 70% idle on average.

If you enable MPS for these inference jobs, the GPUs can interleave their work so that while one job is waiting on memory, another job's kernel might fill the GPU, etc. The result is higher overall GPU utilization. In practice, if two processes each use 40% of a GPU, with MPS you might see the GPU at 80-90% utilization serving both. For instance, two training processes that each would take 1 hour on their own - on the same GPU, run sequentially - can run together under MPS and finish in a bit over 1 hour total in parallel instead of 2 hours sequentially. This 2× speedup is a result of merging the work of both training processes and keeping the GPU fully busy.

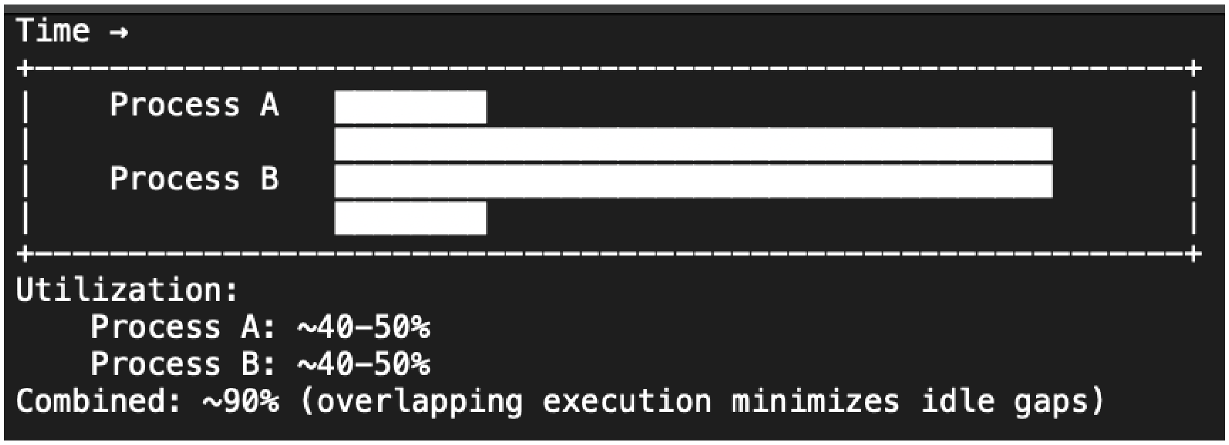
To visualize, imagine Process A and Process B each launching kernels periodically without MPS. The GPU schedule might look like A-B-A-B with gaps in between while each one waits as shown in Figure 3-2.



```
Time →
+------------------------------------------------------------------------------+
|   Process A   |   Idle    |   Process B   |   Idle    |   Process A   |   Idle    |
| [██████████]  |           | [██████████]  |           | [██████████]  |           |
+------------------------------------------------------------------------------+
Utilization:
    Process A: ~40-50%
    Process B: ~40-50%
Combined: < 100% (with gaps, overall ~70% effective)
```

Figure 3-2. GPU alternates between running Process A's kernels and Process B's kernels and creates idle gaps in which one process is waiting while the other is active

With MPS, the schedule becomes more like A and B overlapping so that whenever A isn't using some parts of the GPU, B's work can use them simultaneously, and vice versa. This overlapping eliminates idle gaps as shown in Figure 3-3.



```
Time →
+------------------------------------------------------------+
|    Process A                                                |
|                                                             |
|    Process B                                                |
|                                                             |
+------------------------------------------------------------+
Utilization:
    Process A: ~40–50%
    Process B: ~40–50%
Combined: ~90% (overlapping execution minimizes idle gaps)
```

Figure 3-3. Eliminating scheduling gaps for processes A and B such that two processes that individually hit ~40-50% utilization, now achieve ~90% when combined with MPS.

Setting up MPS involves running an MPS control daemon (`nvidia-cuda-mps-control`) which then launches an MPS server process that brokers GPU access. On modern GPUs, MPS is more streamlined as clients (the processes) can talk directly to the hardware with minimal interference from the compute node itself. Typically, you start the MPS server on a node - often one per GPU or one per user - and then run your GPU jobs with an environment variable that connects them to MPS. All jobs under that server will share the GPU concurrently.

Another feature of MPS is the ability to set an active thread percentage per client. This limits how many streaming multiprocessors (GPU cores, essentially) a client can use. This can be useful if you want to guarantee quality of service (QoS) where two jobs, for example, each get at most 50% of the GPU's execution resources. If not explicitly set, the jobs will just compete and use whatever GPU resources they can.

Note that MPS does not partition GPU memory, so all processes will share the full GPU memory space. MPS is mainly about compute sharing and scheduling. The issue is that one process could request a massive amount of GPU RAM, cause an out-of-memory (OOM) error on the GPU, and result in terminating all of the other processes running on the GPU. This is very disruptive. Also, if one program saturates the GPU 100% on its own, MPS won't magically make it go faster as you can't exceed 100% utilization. It's only beneficial when individual jobs leave some slack that others can fill.

Another limitation of MPS is that, by default, all MPS clients must run as the same Unix user since they share a context. In multi-user clusters, this means MPS is usually set up at the scheduler level such that only one user's jobs share a GPU at a time. Otherwise, you can configure a system-wide MPS that's shared by all users, but understand that the jobs are not isolated from a security standpoint. Recent NVIDIA drivers have introduced the concept of multi-user MPS, but it's less common in production as of this writing.

One specific alternative to MPS is a feature for time-slicing GPUs in Kubernetes. Time-slicing on Kubernetes allows the device plugin to schedule different pods on the same GPU by time. For instance, four GPUs would each get 25% of the time. It's sort of an automated time-sharing algorithm that doesn't require MPS. However, this doesn't overlap execution, it just switches more rapidly than the default driver would. Time-slicing may be useful for interactive workloads where you prefer isolation at the cost of some idle time. For high throughput jobs, overlapping with MPS or splitting the GPU with a multi-instance GPU is usually better than fine-grained time slicing as discussed next.

Multi-Instance GPU

Starting with the NVIDIA A100 Ampere generation, GPUs can be partitioned at the hardware level into multiple instances using multi-instance GPU (MIG). MIG allows a GPU to be sliced into as many as 7 smaller logical GPUs - each with its own dedicated portion of memory and compute units, or streaming multiprocessors (SMs). For example, a 192 GB Blackwell B200 GPU can be split into 7 instances of about 27 GB (192 GB / 7 instances) and 20 SMs (140 SM's / 7 instances) each. Each instance acts like a separate GPU from the perspective of software since it has its own memory, its own streaming multiprocessors, and even separate engine contexts.

The benefit of MIG is strong isolation and guaranteed resources for each job. If you have multiple users or multiple services that only need, say, 10 GB of logical GPU memory each, you can pack them onto one physical GPU without them interfering with each other's memory or compute. It's a form of virtualization but done in hardware, so the overhead is very low - maybe a few percent – due to the loss of some flexibility. If one instance is idle, it can't lend its resources to another as they are hard partitioned. Also, if you're not using all MIG slices, you're wasting resources by leaving them fragmented. It's important to plan partition sizes to match your workloads.

One important operational note is that, in order to use MIG, you typically configure it at the system level - or at least at the node level. The GPU has to be put into MIG mode, the slices created, the node rebooted, and then the slices appear as separate devices to the system.

TIP

The Kubernetes device plugin will list MIG devices as resources like "nvidia.com/mig-2g.10gb" in the case of a 2 GPU slice of 10 GB.

Jobs can request MIG devices specifically, but you have to be careful to schedule in a way that uses all slices. For instance, if you have a 7-slice setup and a job only takes one slice, the other 6 should be packed with other jobs or you're leaving a lot idle. It's possible to configure certain nodes in your cluster to use MIG for small inference jobs, for example - and configure other nodes for non-MIG workloads for large training jobs.

For large-scale model training jobs and inference servers that span many GPUs, MIG is typically not useful since we want access to the full set of GPUs. On the other hand, for multi-tenant, small-model inference servers that can run smaller GPU partitions, MIG and its isolation features could be useful.

In summary, enable MIG only when you need to run multiple independent jobs on the same GPU with strong isolation. Do not use MIG for large-scale distributed training or inferencing that spans GPUs as you want access to the full power of the GPUs and their fast interconnects.

In our context of large Transformer-based model training and inferencing, we will leave MIG off. But it's good to know that this feature exists. Perhaps a cluster might dynamically switch modes and run MIG during the day when lots of small training or inferencing experiments are happening, then turn MIG off at night to run big training jobs that use whole GPUs.

GPU Clock Speeds and Error Correcting Code

NVIDIA GPUs have something called GPU Boost which automatically adjusts the core clock within power and thermal limits. Most of the time, you should let the GPU just do its thing. But some users like to lock the clocks for consistency so that the GPU always runs at a fixed maximum frequency. This way, run-to-run performance is stable and not subject to variations in power or temperature. This is extremely important - and a common issue - when performing benchmarks as later runs may be throttled due to excessive heat. If you are not aware of this, you may inadvertently interpret the poor results of later runs incorrectly as the GPUs may be throttled due to excessive heat caused by previous runs.

You can use `nvidia-smi -lgc` to lock the core clock and `-ac` to lock the memory clock. This can help ensure that the GPU runs at a consistent frequency, which is especially useful for benchmarking or when you need reproducible performance results. By locking the clocks, you prevent fluctuations that might occur due to the default auto-boost behavior, which can vary with thermal conditions or power availability.

Some people may choose to underclock the GPUs a bit to reduce the heat if they are planning to run very long jobs and want to avoid hitting the thermal slowdown during a benchmark run. However, unless you see variability in GPU performance due to thermal throttling, you usually don't need to lock the clocks as data center GPUs often have power and temperature headroom - as well as proper air or liquid cooling. But locking the clocks is something to be aware of if you're chasing the last bit of determinism and consistency. Typically, however, leaving the GPU in the default auto-boost mode is fine.

Error Correcting Code (ECC) memory on GPUs is another consideration. ECC ensures that if there's a single-bit memory error caused by cosmic rays, for example, the memory can be corrected on the fly. And if there's a double-bit error, the error is detected and will throw an error to the calling code. ECC is usually enabled by default on NVIDIA data center GPUs.

Disabling ECC can free up a small amount of memory since ECC requires extra bits for error checking. This might yield a marginal performance gain by reducing the overhead associated with on-the-fly error checking, but typically just a few percent. However, turning off ECC also removes critical memory-error protection, which can lead to system instability or undetected data corruption.

Moreover, on many of NVIDIA's newer data center GPUs including Hopper and Blackwell, ECC is always enabled and cannot be disabled. This design choice helps ensure reliability and data integrity in demanding, high-performance computing environments.

For long training or inference jobs on huge models, a single memory error could crash the job completely or, even worse, silently corrupt your model without a warning. As such, it's recommended to always keep ECC on for any serious AI workload. The only time you'd possibly consider turning it off is in a research setting where you are fine with taking the risk because you need that extra sliver of memory for your model to fit into your limited-memory GPU cluster.

Toggling ECC mode requires resetting the GPU and likely restarting jobs that are currently running on that GPU. So it's not a toggle that you want to switch frequently. Keep ECC on for stability and reliability. The peace of mind outweighs the negligible speedup of turning ECC off.

GPU Memory Oversubscription, Fragmentation, and Out-of-Memory Handling

Unlike CPU RAM, by default there is no such thing as GPU “swap” memory. If you try to allocate more GPU memory than available, you will get an unfriendly out-of-memory (OOM) error along with an even-unfriendlier process crash. There are a couple of mechanisms to mitigate this issue including allowing memory to grow dynamically, unified memory across CPU and GPU, and memory pools and caching allocators.

By default, some frameworks (e.g. TensorFlow) grab all of the available GPU memory at startup to avoid fragmentation and improve performance. If you don’t know this, it can be very bad in scenarios where you are sharing the GPU. PyTorch by default only allocates GPU memory as needed. And fortunately, TensorFlow has an option (`TF_FORCE_GPU_ALLOW_GROWTH=true`) to make it start small and dynamically grow the GPU memory usage as needed - similar to PyTorch. Neither PyTorch nor TensorFlow let you allocate more memory than the GPU has available, of course. But this lazy-allocation plays nicer in multi-tenant scenarios because two processes won't both try to simultaneously allocate the maximum available GPU memory from the start.

CUDA's Unified Memory system lets you allocate memory without predefining whether it resides on the CPU or GPU. The CUDA runtime handles moving pages as needed. Modern NVIDIA GPUs like Hopper and Blackwell include hardware support for on-demand paging using the Page Migration Engine (PME). PME automatically migrates memory pages between GPU memory and host CPU RAM when the GPU runs low on available memory. However, while PME provides flexibility, relying on it can introduce performance penalties compared to having enough GPU memory for your workload.

This GPU-to-CPU memory offloading can be slow, however, since CPU memory I/O is slower than GPU high-bandwidth memory (HBM) I/O as we learned in the previous chapter. This mechanism is mostly a convenience for practitioners trying to run models that don't fit into GPU RAM. For performance-critical workloads, however, you generally want to avoid relying on unified memory oversubscription where possible. It's there as a safety net instead of outright crashing your script, but your job will run slower when GPU memory is over-subscribed.

Libraries like PyTorch use a caching allocator so that when you free GPU memory, it doesn't return the memory to the OS immediately. Instead, it keeps it to reuse for future allocations. This avoids memory fragmentation and the overhead of asking the OS to repeatedly allocate the same block of memory. You can enable PyTorch's allocator using environment variables like `PYTORCH_CUDA_ALLOC_CONF` to set a max pool size.

If you run into the GPU OOM error, which you surely will at some point, it's likely caused by memory fragmentation or excessive memory caching. You can try to clear the cache using PyTorch's `torch.cuda.empty_cache()`, but it almost-always means your workload legitimately needs that much memory.

From an OS perspective, you use Linux cgroups or Docker's options to enforce GPU memory limits for a given process. For example, with Docker you can run a container with `--gpus <device>:<mem_limit>` to artificially constrain how much GPU memory it can allocate for the container. This way, if a process tries to use more than, say, 10 GB, it will get killed or error out rather than affecting others. In multi-tenant nodes, this could be useful to isolate jobs. In a single-job-per-GPU situation, it's not common to set a memory limit as you want to let the job use as much of the GPU memory as it can get.

In general, running out of GPU memory is something you can manage at the application level. For instance, you can reduce the data batch size, model weight precision, or even the model parameter count, if that's an option. A best practice is to monitor GPU memory usage with `nvidia-smi` or NVML APIs during model training and inferencing. If you're close to the memory limit, consider workarounds like reducing batch size, gradient checkpointing for training, or other techniques to lower memory usage.

Also, you should ensure that your CPU memory isn't being swapped as this would indirectly hurt your GPU utilization and goodput because each time your GPU tries to fetch something from the CPU host, but the host memory page has been swapped to disk, your performance will be bottlenecked by the much-slower disk I/O. So it's important to combine these memory-reduction best practices with the earlier advice about pinning memory, increasing the `ulimit`, and disabling swappiness, etc.

Finally, it's recommended to always keep the GPU driver loaded instead of unloading the GPU driver between jobs. This is similar to GPU persistence mode, but at a deeper level. Some clusters are configured to unload the driver when no jobs are running in order to free OS kernel memory and for security.

However, if you do that, the next job has to pay the cost of re-loading the GPU driver and, if MIG is used, re-configuring MIG slices. Usually, you want to keep the driver and any MIG configuration persistent across jobs. The only time you want to unload the GPU driver is for troubleshooting or upgrading the driver. As such, cluster admins often set up the system so that the NVIDIA driver modules are always present once the machine boots.

Container Runtime Optimizations for GPUs

Many AI systems use orchestration tools and container runtimes to manage the software environment. Kubernetes and Docker are popular in AI infrastructure. Using containers ensures that all dependencies including CUDA and library versions are consistent. This avoids the “but it works on my machine” problem. Containers introduce a bit of complexity and a tiny amount of overhead, but with the right configuration, you can get near bare-metal performance for GPU workloads using containers.

A container running on a node is not a traditional virtual machine (VM). In contrast to VM’s, containers share the host OS kernel so that CPU and memory operations perform at near-native speed. Similarly, for GPU workloads, the container can directly use the host's GPU driver and hardware with negligible overhead. This is enabled by NVIDIA's Container Toolkit which allows Docker containers to directly use the GPUs on the host.

NVIDIA Container Toolkit and CUDA Compatibility

One challenge when using containers with GPUs is making sure that the CUDA libraries inside the container match the driver on the host. NVIDIA solves this through their NVIDIA Container Toolkit and base Docker images. The host provides the NVIDIA driver which, remember, is tightly integrated with the kernel and hardware. Inside the container, you typically find the CUDA runtime libraries of a certain version.

The general rule is that the host's NVIDIA driver version must be at least as recent as the minimum driver version required by the CUDA version inside the container. If you're running a container with CUDA 12.8, your host must have an NVIDIA driver version of 570.124.06 or higher. If your host has an older driver version, a container with CUDA 12.8 may not work properly.

TIP

The simplest approach is to use NVIDIA's official base Docker images from the NVIDIA GPU Cloud (NGC) or DockerHub image repositories. The images in these repositories are well tested and describe which NVIDIA driver version they need on the host. End Tip.

NVIDIA Container Runtime

Alternatively, NVIDIA's container runtime can actually inject the host driver libraries into the container at runtime, so you don't even need to ship the NVIDIA driver inside the image. Instead, you just rely on the host's driver. Again, this works because the container isn't fully isolated like a traditional VM. Docker containers are allowed to use host devices, volumes, and libraries.

Your code inside the container sees, for instance, `libcudart.so` from the CUDA toolkit on the host. It invokes this library directly on the host so everything just works. If you were to mismatch and try to use a newer CUDA version in the container with an old driver on the host, you'll likely get an error. It's important to match the CUDA and driver versions.

The key takeaway is that there is no hypervisor or virtualization layer involved when using containers for GPUs.

The container is sharing the host kernel and driver directly, so when a kernel launches on the GPU, it's as if it launched from the host. This means you aren't losing performance to Docker-based virtualization - unless you are using something like VMware or SR-IOV virtual GPUs which is a special scenario that requires some tuning. With Docker + NVIDIA, it's basically the equivalent to bare metal performance.

Avoiding Container Overlay Filesystem Overhead

The main difference when running in a Docker container versus running directly on the host might be in I/O. Containers often use a union filesystem that transparently overlays multiple underlying filesystems, like the host filesystem and the container filesystem, into a single, unified view.

In a union filesystem such as OverlayFS, files and directories from multiple sources will appear as if they belong to one filesystem. This mechanism is especially useful for containers, where the read-only filesystem from the base image layer is combined with a writable container layer.

There is some overhead when using an overlay filesystem, however. This extra latency arises because the filesystem must check multiple underlying layers - both read-only and writable - to determine which version of a file should be returned. The additional metadata lookups and the logic for merging these layers can add a small amount of overhead compared to reading from a single, simple filesystem.

Furthermore, when writing to the copy-on-write (CoW) mechanism used by the overlay. CoW means that when you modify a file in the read-only layer (e.g., the base image), the file must first be copied to the writable layer. The write then happens to the copied writable file - instead of the original, read-only file. As mentioned earlier, reading a modified file requires looking at both the read-only and writable layers to determine which is the correct version to return.

Model training often involves heavy I/O operations when reading datasets, loading a model, and writing model checkpoints. To work around this, you can mount a host directory - or network filesystem - into the container using bind mounts.

Bind mounts bypass the overlay and therefore perform similar to disk I/O directly on the host. If the host filesystem is something like an NVMe SSD or an NFS mount, you get the full performance of that underlying storage device. We purposely do not package a multi-terabyte dataset inside the image. Instead, we bring the data in through the mounts.

For example, if your training data is on `/data/dataset` on the host, you'd run the container with `-v /data/dataset:/mnt/dataset:ro` where "ro" means "read-only" mount. Then your training script reads from `/mnt/dataset`. This way, you're reading directly from the host filesystem.

In fact, it's a best practice to avoid heavy data reads/writes against the container's writable layer. Instead, mount your data directory and output directory from the host into the container. You want to ensure that I/O is not bottlenecked by the overhead of the container's copy-on-write mechanism.

Reduce Image Size for Faster Container Startup

Container startup times can be quite a bit slower if the image is huge and needs to be pulled over the network. But in a typical long-running training loop, a startup time of a few minutes is negligible compared to the hours, days, or months of training time. It's still worth keeping images reasonably slim by not including unnecessary build tools or temporary build files. This saves disk space and improves container startup time.

Some HPC centers prefer Singularity (Apptainer) over Docker, because it can run images in user space without a root daemon. It also uses the host filesystem directly and tends to have virtually zero overhead beyond what the OS already has.

In either case, Docker or Singularity (Apptainer), studies and benchmarks have shown that once properly configured, these container solutions measure only a couple percent difference between running a container and directly on the host. Essentially, if someone gave you a log of GPU utilization and throughput, it would be difficult to tell from the log alone whether the job ran in a container or not.

Kubernetes for Topology-Aware Container Orchestration and Networking

Kubernetes is a popular container orchestrator used across a wide variety of use cases including AI training and inference. Similar to the NVIDIA Container Toolkit, NVIDIA supports a

Kubernetes device plugin for their GPUs called the NVIDIA GPU Operator. When you deploy a container on Kubernetes with this device plugin, Kubernetes takes care of making the GPUs available to the container. You simply request a GPU in your Kubernetes pod specification, and the device plugin mounts the `/dev/nvidia*` devices for you. The GPU Operator device plugin also mounts the driver libraries from the host into the container.

When using Kubernetes to orchestrate GPU-based containers, you want it to allocate resources to containers in a manner that is aware of the hardware topology including the NUMA node and network bandwidth configurations. However, by default, Kubernetes (K8s) is not topology aware. It treats each GPU as a resource but doesn't know if GPU0 and GPU1 are on the same NUMA node or if they use the same NVLink interconnect. This could make a big difference.

Consider an 8-GPU server with two sets of 4 GPUs - each connected by NVLink. If you request 4 GPUs from Kubernetes for a job, it would be ideal if K8s gave you 4 GPUs that are all interconnected with NVLink as they can share data faster. However, if K8s picks 4 arbitrary GPUs spread anywhere in the system, your job might be allocated 2 GPUs from one rack and 2 GPUs from another rack. This will introduce slower multi-rack (e.g. InfiniBand or Ethernet) interconnects into the GPU-to-GPU routes which could reduce your inter-GPU bandwidth by half.

To avoid resource contention, you should try to either reserve the resources that you need or request the entire node for your job. For the container/pod placements, you should align pods with CPU affinities and NUMA nodes using the Kubernetes Topology Manager component to bind the container's CPUs to the same NUMA node as the GPUs that the container was allocated.

Orchestrating Containers with Kubernetes Topology Manager

Fortunately, Kubernetes Topology Manager and Nvidia's hardware-aware GPU Operator device plugin can provide detailed topology information. For example, they can detect that GPU 0 is connected to NUMA node 0, NVLink domain A, and PCIe bus Z. The Kubernetes scheduler can then use this information to allocate containers to GPUs in an optimal way for efficient processing and communication.

Topology-aware scheduling for GPUs is still evolving. In some environments, administrators may still be using node labels to explicitly mark GPU or node topology. This ensures that multi-

GPU pods are placed on hosts where the GPUs share the same NVLink domain or NUMA node.

For our purposes, if you're running multi-GPU jobs in Kubernetes, make sure to enable topology-aware scheduling. This typically involves configuring the Kubernetes Topology Manager policy to use options like "best-effort", "restricted", or in some cases "single-numa-node".

Also, be sure to use the latest NVIDIA device plugin which supports packing multi-GPU pods onto GPUs that are connected to the same NUMA node. This setup helps ensure optimal performance by minimizing cross-NUMA-node communication and reducing latency in multi-node GPU workloads.

Job Scheduling with Kubernetes and SLURM

In multi-node deployments, job schedulers are essential for maximizing resource utilization across all nodes. Commonly, SLURM is deployed for training clusters while Kubernetes is typically favored for inference clusters. However, hybrid solutions have emerged that integrate SLURM with Kubernetes. The open-source Slinky project and CoreWeave's SUNK product are examples of these integrated solutions to simplify cluster management across training and inference workloads..

These systems handle the allocation of GPUs to jobs and coordinate the launch of processes across nodes. If a training job requests 8 nodes with 8 GPUs per node, the scheduler will identify eligible nodes and start the job using tools like `mpirun` or container runtimes such as Docker, ensuring that each process is aware of all available GPUs in the job. Many clusters also rely on well-tested Docker repositories like NVIDIA's NGC Docker repository to guarantee a consistent software environment - including GPU drivers, CUDA toolkits, PyTorch libraries, and other Python packages - across all nodes.

With SLURM, similar issues exist. SLURM has the concept of "generic resources" for GPUs, and you can define that certain GPUs are attached to certain NUMA nodes or NVLinks/NVSwitches. Then in your job request, you can ask for GPUs that are, say, connected to the same NUMA node. If not properly set, a scheduler might treat all GPUs as identical and provide non-ideal allocations for your multi-GPU container requests. Proper configuration can avoid unnecessary cross-NUMA-node and cross-NVLink GPU communication overhead.

Slicing a GPU with Multi-Instance GPU

If you are using Nvidia's Multi-Instance GPU (MIG) to slice a GPU into partitions, the scheduler needs to be aware of the slices. Kubernetes, for instance, will see each MIG instance as a resource like “nvidia.com/mig-3g.20gb”. You could request two of those, and then K8s would place your pod on a node that has at least two free MIG instances.

An administrative drawback with MIG is that switching a GPU between MIG mode and normal (non-MIG) mode requires rebooting the compute node to reset the GPU. So it's not something the scheduler can easily do dynamically per job. Usually, you create MIG partitions in advance and leave the configuration running for some period of time. You can label one K8s node with “mig-enabled” and another as “mig-disabled” and let the scheduler place jobs/pods accordingly. This is more of an operational detail, but it's good to know that MIG is a static partition and not a dynamic scheduler decision.

TIP

Persistence mode is recommended when using MIG so that the MIG configuration remains active on the GPU even if no jobs are running. This way, the GPU doesn't have to keep re-building the slices before running each periodic job.

Optimizing Network Communication for Kubernetes

When you run multi-node GPU workloads using containers with Kubernetes, the pods need to talk to each other. In Kubernetes, by default pods have their own IP and there might be an overlay network or network-address translation (NAT) between pods on different nodes. This can introduce complications and additional overhead.

Often, the simplest solution for GPU clusters is to use host networking for these performance-sensitive jobs. That means the container's network is not isolated as it uses the host's network interface directly. To enable this in Kubernetes, you set `hostNetwork: true` on the pod specification. In Docker, you could run with `--network=host`.

Using host networking allows a container to access the InfiniBand interconnect exactly as the host does - without any additional translation or firewall layers. This is particularly useful for MPI jobs because it eliminates the need to configure port mappings for every MPI rank.

However, if host networking is not an option due to security policies, you must ensure that your Kubernetes container network interface (CNI) and any overlay network can handle the required traffic. In such cases, you may need to open specific ports to support NCCL's handshake and data exchange, using environment variables like `NCCL_PORT_RANGE` and `NCCL_SOCKET_IFNAME` to help establish connections.

When operating over an overlay network, it's critical that latency remains low, operations run in kernel space, and that no user-space proxies throttle traffic between nodes—since these factors can significantly impact performance.

Reducing Kubernetes Orchestration Jitter

Running an orchestrator like Kubernetes means there are some background processes running on every node (e.g. the Kubernetes "kubelet"), container runtime daemons, and (ideally) monitoring agents. While these services consume CPU and memory, the consumption is on the order of a few percent of a single core. So they won't steal noticeable time from a GPU-based training job which uses these cores for data loading and pre-processing.

However, if the training job is running on a node that is also running an inference workload, you may experience some jitter. This is common in any multi-tenancy situation, though. If another container on the same machine unexpectedly uses a lot of CPU or I/O, it will affect your container - whether training or inference - by competing for the same resources.

TIP

Homogeneous workloads such as all training or all inference are much easier to debug and tune from a system's perspective than a heterogeneous mix of both training and inference.

Improving Resource Guarantees

To safeguard against resource contention, Kubernetes lets you define resource requests and limits for pods. For example, you can specify that your training job requires 16 CPU cores and 64GB of RAM. Kubernetes will then reserve those resources exclusively for your job and avoid scheduling other pods on the same CPUs.

These limits are enforced using Linux cgroups, so if your container exceeds its allocation, it can be throttled or even terminated by the OOM killer. It's common practice to use resource requests - and optionally the CPU Manager feature to pin cores - to ensure that performance-critical jobs get exclusive access to the necessary CPU resources, so that other processes cannot steal CPU time from your reserved cores.

Another source of jitter is background kernel threads and interrupts as we discussed in Chapter 2 in the context of using IRQ affinity. Similarly with Kubernetes, if other pods are using the same network or disks as your job, the other pods might cause a lot of interrupts and extra kernel work on the compute nodes that host your job. This will cause jitter and affect your job's performance.

Ideally, a GPU node is fully dedicated to your job. However, if it's not, you should ensure that the node is carefully partitioned using Linux cgroup controllers for I/O and CPU so that other workloads don't interfere.

Fortunately, Kubernetes supports CPU isolation which ensures that pods get the dedicated CPU cores and memory they request - and prevents other pods from being scheduled on the same CPU core as yours. This avoids extra overhead from context switching and resource contention.

TIP

In practice, performance-sensitive Kubernetes jobs should request all of the CPUs and GPUs of a given node so that nothing else interferes or contends with the jobs' resources. Easier said than done, but this is the ideal job configuration from a performance and consistency standpoint.

Memory Isolation and Avoiding the OOM Killer

Memory interference can also occur if not properly limited. Kubernetes provides first-class memory isolation support (using Linux cgroups). However, a greedy container, if unconstrained, could allocate too much memory on the host. This would cause the host to swap some of its memory to disk.

If an unbounded container uses too much memory on the host, the infamous Linux “OOM Killer” will start killing processes - and potentially your Kubernetes job - even if your job wasn’t the one using too much memory.

The OOM killer uses heuristics when deciding which pods to kill. Sometimes it decides to kill the largest running pod which is likely your large training or inference job holding lots of data in CPU RAM to feed the GPUs. To avoid this, you can purposely not set strict memory limits on training or inference containers. This way, they can use all available memory, if needed.

With proper monitoring and alerting, you can ensure the job doesn’t try to over-allocate beyond what you expect. If you do set a memory limit, make sure it’s above what you actually expect to use. This provides a bit of headroom to avoid getting killed by the OOM killer 3 days into a long-running training job.

Dealing with I/O Isolation

As of this writing, Kubernetes does not offer native, first-class I/O isolation out-of-the-box, unfortunately. While Linux does support I/O controls via cgroup controllers, Kubernetes itself does not automatically enforce I/O limits in the same way it does for CPU and memory.

If you need to ensure that heavy I/O workloads on a GPU node don't interfere with each other, you might need to manually configure I/O controls at the node level. This can involve adjusting the `blkio` weights or using other OS-level configurations to partition I/O resources. In short, while Kubernetes prevents CPU contention through scheduling and resource requests, I/O isolation usually requires additional, manual tuning of the underlying Linux system.

It's important to note that, inside a container, some system settings are inherited from the host. For instance, if the host has CPU frequency scaling set to performance mode, the container will inherit that setting. But if the container is running in a virtualized environment such as a cloud instance, you might not be able to change these settings.

It's a good idea to always ensure that the host machine is tuned since containers can't change kernel parameters like hugepage settings or CPU governor limits. Usually, cluster admins set these parameters and settings through the base OS image. Or, in a Kubernetes environment, they might use something like the NVIDIA GPU Operator to set persistence mode and other `sysctl` knobs on each node.

Key Takeaways

Below is a list of key takeaways from this chapter including optimizations across the operating system, driver, GPU, CPU, and container layers.

Data and Compute Locality is Critical.

> Ensure that data is stored and processed as close to the computation units as possible. Use local, high-speed storage such as NVMe or SSD caches to minimize latency and reduce reliance on remote filesystems or network I/O.

NUMA-Aware Configuration and CPU Affinity.

Optimize CPU-to-GPU data flow by aligning processes and memory allocations within the same NUMA node. Pinning the CPU with tools like `numactl` and `taskset` prevents cross-node memory access, leading to lower latency and improved throughput.

Maximize GPU Driver and Runtime Efficiency.

Fine-tune the GPU driver settings, such as enabling persistence mode to keep GPUs in a ready state. Consider features like Multi-Process Service (MPS) for overlapping work from multiple processes on a single GPU. For multi-tenant environments, explore Multi-Instance GPU (MIG) partitions to isolate workloads effectively.

Effective Data Prefetching and Batching.

Keep the GPUs fed by prefetching data ahead of time and batching small I/O operations into larger, more efficient reads. Leverage prefetching mechanisms like PyTorch's DataLoader `prefetch_factor` to load multiple batches in advance.

Data Loading with Pinned Memory.

Combining data prefetching with memory pinning using PyTorch's DataLoader `pin_memory=True` uses pinned

CPU memory (page-locked, not swappable to disk) for faster, asynchronous data transfers to the GPU. As a result, data loading and model execution can overlap, idle times are reduced, and both CPU and GPU resources are continuously utilized.

Memory Transfer Optimization.

Leverage techniques such as pinned, page-locked memory and huge pages to accelerate data transfers between the host and GPU. This helps reduce copy overhead and allows asynchronous transfers to overlap with computations.

Overlap Communication with Computation.

Reduce the waiting time for data transfers by overlapping memory operations like gradient synchronization and data staging with ongoing GPU computations. This overlap helps maintain high GPU utilization and better overall system efficiency.

Scalable Networking Tuning.

In multi-node environments, use RDMA-enabled networks (e.g., InfiniBand/Ethernet) and tune network settings such as TCP buffers, MTU, and interrupt affinities to maintain high throughput during distributed training and inference.

Use Containerization and Orchestration for Consistency.

Use container runtimes like Docker with the NVIDIA Container Toolkit and orchestration platforms like Kubernetes with the NVIDIA GPU Operator device plugin to ensure that the entire software stack - including drivers, CUDA libraries, and application code - is consistent across nodes. These solutions help align CPU-GPU affinities and manage resource allocation based on hardware topology.

Eliminate Container Runtime Overhead.

While containers increase reproducibility and ease of deployment, ensure that CPU and GPU affinities, host networking, and resource isolation are correctly configured to minimize any container overhead.

Use Orchestration and Scheduling Best Practices.

Robust container orchestrators like Kubernetes are essential components for ensuring efficient resource allocation. Advanced scheduling techniques - such as the Kubernetes Topology Manager - help ensure that GPUs with fast interconnects are clustered together.

Strive for Flexibility through Dynamic Adaptability and Scaling.

The orchestration layer distributes work and dynamically manages workload segmentation across nodes. This flexibility is crucial for both scaling up training tasks and ensuring efficient runtime in inference scenarios where data loads and request patterns vary widely.

Continuous and Incremental Tuning.

System-level optimizations are not one-and-done. Regularly monitor performance metrics, adjust CPU affinities, batch sizes, and prefetch settings as workloads evolve, and use these small improvements cumulatively to achieve significant performance gains.

Reduce Bottlenecks Across the Stack.

The ultimate goal is to ensure that all components from the OS and CPU to the GPU driver and runtime work in harmony. Eliminating bottlenecks in one layer such as CPU memory allocation or driver initialization unlocks the full potential of the GPUs, which directly translates to faster training, lower costs, and more efficient resource usage.

Together, these strategies work to minimize data transfer friction, reduce wait times, and ensure that your hardware is used to its fullest potential for efficient training and inference.

Conclusion

This chapter has demonstrated that even the most advanced GPUs can be hindered by inefficiencies in their surrounding environment. A well-tuned operating system and GPU software stack form the unsung backbone of high-performance AI systems. By aligning data with compute through NUMA-aware pinning and local storage solutions, overlapping communication with computation, and fine-tuning both the host system and GPU drivers, you can dramatically reduce latency and boost throughput. A well-tuned operating system, container runtime, cluster orchestrator, and software stack form the backbone of high-performance AI systems.

Think of your entire system as a precision-engineered sports car where each component (CPU, memory, GPU, network, containers, orchestrators, and programming stack) must work seamlessly together to deliver maximum performance. Small tweaks, such as enabling persistence mode or optimizing CPU scheduling, may seem minor on their own, but when combined and scaled across a large GPU cluster, they can lead to substantial savings in time and cost. Whether you’re training massive Transformer models or running complex inference pipelines, these optimizations ensure that GPUs are consistently operating near their peak efficiency.

As the field evolves and models continue to grow, the importance of system-level tuning will only increase. The techniques discussed in this chapter empower performance engineers and system architects to leverage every bit of hardware potential. This enables faster iteration cycles and more cost-effective AI deployments. Ultimately, a deeply optimized system accelerates research and makes cutting-edge AI applications more accessible to a broader audience.

Finally, remember that while the hardware and software stack may seem like an unmanageable amount of interconnected knobs and switches, small tweaks can translate into significant savings in time and cost. By continuously monitoring performance metrics and incrementally refining each layer of the stack, you can transform potential bottlenecks into opportunities for efficiency gains. Let the data guide you and you will unlock the full potential of your AI system.

Chapter 4. Distributed Communication and I/O Optimizations

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 4th chapter of the final book.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at *arufino@oreilly.com*.

In today's high-performance AI landscape, the need for seamless and efficient data movement between GPUs, storage, and network interfaces is more critical than ever. Chapter Four dives into state-of-the-art distributed communication libraries such as RDMA, NCCL, and the newer NIXL for inference.

This chapter brings into focus how storage and network systems serve as critical links between data and compute. In large-scale systems, even the fastest GPUs can be hindered by inefficient communication and data transfer from memory and disk. As such, we discuss strategies for speeding up sequential data transfers, proper data sharding, and advanced techniques that allow the GPU to overlap communication and computation - and work directly with fast storage subsystems.

We will discuss the importance of overlapping communication and computation using components of Nvidia's I/O acceleration platform called Magnum IO. including NCCL, NIXL, GPUDirect RDMA, and GPUDirect Storage. We'll then explore how to use these libraries to lower communication latency, reduce CPU overhead, and maximize throughput across all layers of a multi-node and multi-GPU AI system. By leveraging high-speed interconnects like NVLink, InfiniBand, and even high-bandwidth Ethernet, these lower-level libraries allow higher-level AI frameworks like PyTorch to effectively overlap computation and communication.

Whether you're training very large models or scaling distributed inference to millions of users, the integration of these technologies into your AI system represents a holistic approach to accelerating communication and data pipelines -

ensuring that every component is tuned for peak performance. Performance engineers need to carefully configure and tune the network and storage configurations to maintain a high level of GPU utilization and goodput.

Overlapping Communication and Computation

Overlapping communication and computation plays a key role in building efficient training and inference AI systems at scale. In these environments, it’s important to keep GPUs busy and spend less time waiting for data. The main idea is to ensure that data transfers occur concurrently with ongoing computations so that, when one task finishes, the results needed for the next stage are already in progress or have been delivered. AI frameworks such as PyTorch support these asynchronous operations so that the all-reduce communication operations, for example, run alongside compute tasks. This reduces idle GPU time and improves overall system throughput .

CUDA-based libraries exploit the power of multiple CUDA streams. While one stream executes compute-heavy matrix multiplications, another handles communication tasks such as aggregating gradients. As each layer of a neural network finishes its computation, the previous outputs are already on their way for aggregation or further processing. This overlapping ensures that the system produces results without unnecessary waiting periods and maintains a steady and efficient flow of data.

Increasing the amount of compute performed between communication events further minimizes communication overhead. When the system processes larger batches of data, it performs more computation before it needs to stop and exchange information. In distributed training, this approach appears as gradient accumulation, where updates from several mini-batches merge into a single synchronization step. By reducing the frequency of communication events, the system lowers the relative cost of each data exchange and boosts overall responsiveness.

Another technique that supports a seamless overlap between computation and communication is compression. Compression reduces the volume of data that needs to be transferred. When a model compresses its gradients before sending them, the network moves a smaller amount of data. This reduces the transfer time and eases congestion. The shorter transfer time means that the communication phase is less disruptive to the computation phase. Although compression does not directly initiate the overlap, it shortens the window during which data moves across the network and allows computational work to continue in parallel more effectively.

Splitting large tensors into smaller buckets further refines the balance between computation and data transfer. Frameworks like PyTorch automatically divide large gradients or activations into several buckets that are transmitted as soon as they become available. This means that rather than waiting for an entire dataset to be ready, parts of it can begin their journey immediately. By tuning bucket sizes and scheduling these transfers appropriately, one can achieve a level of overlap that helps prevent communication delays from stalling the compute pipeline. Performance tools such as the PyTorch Profiler and NVIDIA Nsight Systems offer insights that allow engineers to adjust these parameters dynamically.

By combining larger batch sizes, gradient accumulation, asynchronous transfers, compression, and bucketing into one cohesive strategy, large distributed AI models overcome network limitations and reduce redundant transfers. This design minimizes synchronization events while achieving high throughput and optimal GPU utilization. The outcome is a training system that not only reduces overall training and inference time but also makes more efficient use of available hardware resources. This frees engineers from having to reinvent low-level networking routines and lets them focus on innovating model architectures and tuning higher-level parameters instead of coding intricate data transfer mechanisms.

NVIDIA Magnum IO Optimization Stack

NVIDIA's overarching I/O acceleration platform is called Magnum IO. Magnum IO brings together a range of technologies to speed up data movement, access, and management across GPUs, CPUs, storage, and network interfaces. There are four key components of the Magnum IO architecture including storage I/O, network I/O, in-network compute, and I/O management as shown in Figure 4-1.

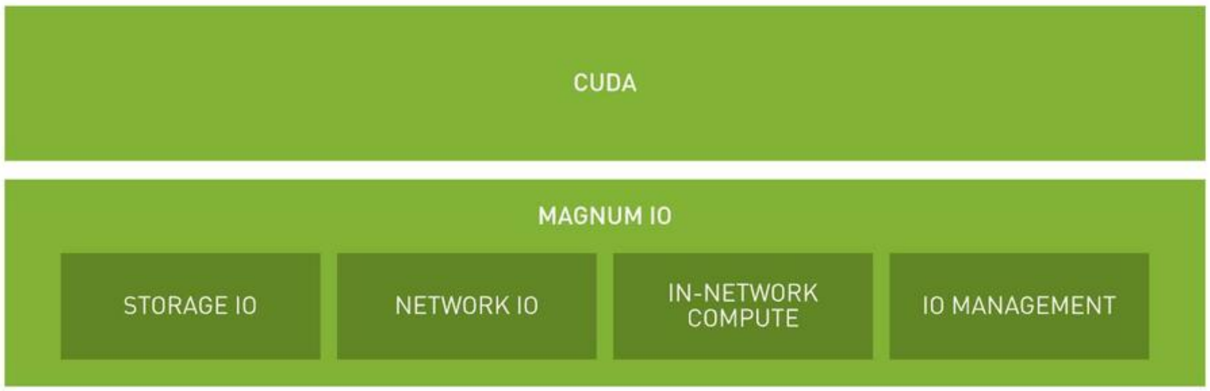


Figure 4-1. Four components of NVIDIA's Magnum IO acceleration platform

The first component is Network I/O. This component includes technologies like GPUDirect RDMA, NCCL, NVSHMEM, and UCX, and HPC-X to enable direct, high-speed data transfers between GPUs across nodes which bypass the CPU.

The second component, Storage I/O, is represented by NVIDIA GPUDirect Storage (GDS) and BlueField SNAP (Software-defined Network Accelerated Processing). These technologies let GPUs access storage data directly from devices such as NVMe SSDs without performing unnecessary copies to host system memory.

The third component, In-Network Compute, includes dedicated devices like NVIDIA BlueField DPUs and protocols like NVIDIA SHARP to perform data aggregations and reductions within the network itself. This reduces latency for large-scale collective operations like all-reduce.

Finally, I/O Management includes platforms like NVIDIA NetQ and NVIDIA Unified Fabric Manager (UFM). These technologies offer real-time telemetry, diagnostics, and lifecycle management of the data center’s I/O fabric.

These components work together through flexible APIs and extensible SDKs that hide the underlying complexity of the low-level details. They help maximize performance and scalability for large-scale AI and analytics workloads. Let's dive into a few of these components and demonstrate how to profile and tune them.

High Speed, Low Overhead Data Transfers with RDMA

Remote Direct Memory Access (RDMA) is a technology optimized for low-latency, high-throughput data transfers by allowing direct memory-to-memory data transfers between components without using the CPU for each packet transfer. RDMA bypasses much of the traditional network stack and ensures CPU-thread affinity to reduce the overhead of transfer-completion interrupts. It also overlaps communication with computation, supports distributed filesystems, and continuously monitors the GPUs and network. It is recommended to use RDMA whenever possible.

Nvidia's RDMA implementation is called GPUDirect RDMA and it allows GPUs to directly exchange data with remote devices such as other GPUs over RDMA-capable networks like InfiniBand or RoCE, bypassing the CPU for data transfers. This minimizes latency and CPU overhead in multi-node environments.

RDMA is supported by modern InfiniBand hardware inherently - as well as some high-speed Ethernet hardware that implements RDMA over Converged Ethernet (RoCE). RoCE is an InfiniBand-style RDMA technology built on top of Ethernet. InfiniBand must be configured properly with all the required drivers installed.

The difference in performance between using RDMA versus standard TCP-based Ethernet without RDMA can be huge. For example, InfiniBand might give <10 microsecond latency and high throughput, whereas standard TCP might be 5-10x higher latency. For large all-reduce operations, throughput matters more than latency, though. With high-throughput networks, even if each message takes a little longer to start transferring (higher latency), the operation will still complete quickly if the network can sustain a high data throughput rate. Conversely, if the network has very low latency but a limited throughput capacity to move data, the overall performance of a large all-reduce will suffer.

If you only have Ethernet, try to ensure that it's the highest bandwidth possible - at least 100 Gbit/s. Also make sure your Ethernet networking stack is tuned to use large MTU jumbo frames like 9000 bytes so that you send fewer big packets rather than many small ones. This is the same intuition as with large files vs. small files. Fewer, larger packets will create less overhead than more, smaller packers. Lastly, ensure your TCP buffer sizes are high enough so the system can have a lot of data in flight.

To tune an Ethernet-based network, you can use the Linux networking `/etc/sysctl.conf` parameters: `net.core.rmem_max`, `net.core.wmem_max` to set max buffer sizes, and `net.ipv4.tcp_rmem`/`tcp_wmem` to set default and autotuning ranges. For a 100 Gbit/s link, you might set a few MB as the max buffer to fully utilize it. Also, using a modern TCP congestion algorithm like BBR can improve throughput on high latency-bandwidth networks, but if it's a controlled, dedicated, and managed cluster network, the default CUBIC TCP congestion control algorithm is usually fine as the network conditions are usually predictable as the network is engineered to avoid congestion.

TIP

To create a controlled cluster network in a cloud environment such as AWS, your on-premise data center would need to use a dedicated, managed link to AWS using AWS Direct Connect. However, if your connection between your on-premise data center and AWS runs over the public internet in any way, it would not typically be described as a controlled cluster network due to variable congestion and unpredictable network conditions.

Since RDMA can move data without using the CPU to copy the data, the CPU still plays a critical role. The CPU sets up RDMA connections, initiates the transfer, handles completion interrupts, and manages the control path. Therefore, you should make sure the network interrupts and threads are pinned to a CPU in the same NUMA node as your InfiniBand host channel adapter (HCA). For instance, if an InfiniBand HCA is in NUMA node 0, you want its interrupts handled by threads running in CPU cores connected to the same NUMA node 0 to reduce latency and improve overall efficiency.

NCCL for Distributed, Multi-GPU Communication

Nvidia Collective Communications Library (NCCL - pronounced “nickel”) is a many-to-many communications library for a group (a.k.a collective) of GPUs. It is the basis for most distributed training and inference workloads. This library acts as the backbone for coordinated communication exchanges of data among groups of GPUs. When performing model training and inferencing across multiple GPUs, compute nodes, racks, and clusters, data needs to be exchanged as quickly as possible to keep the GPUs busy with useful work.

During model training, every GPU calculates gradients over its own data chunk on the backward pass. The system then performs an all-reduce communication operation to aggregate the gradients across all GPUs.

During model inference, GPUs exchange activations during the forward pass as a request moves through the network. This exchange uses send and receive communication operations. All of these communication operations are implemented in the NCCL.

Topology Awareness in NCCL

Topology awareness plays a major role in NCCL, as well. NCCL detects how GPUs are physically connected and adapts its patterns accordingly. In a system with relatively-flat, high-speed interconnects like the GB200 NVL72, every GPU talks directly with every other GPU at maximum speed. In more fragmented systems, NCCL will automatically group devices into smaller clusters where the heavy work happens on GPUs connected by fast links. This reduces the amount of communication over slower connections.

NCCL automatically selects the faster available interconnect – whether its PCIe, NVLink, or InfiniBand – and then chooses the best algorithm and route for the data to use. And NCCL will automatically use RDMA if RDMA is supported by the network hardware. This frees developers from having to micromanage every detail of the communication exchange.

If you have very fast interconnects between GPUs like NVLink/NVSwitch between GPUs within a compute node or InfiniBand between GPUs in separate compute nodes, the communication cost is very low. But often, especially with Ethernet-based clusters, the network can slow things down.

NCCL Communication Strategies

NCCL provides several communication techniques to efficiently exchange data among GPUs during distributed operations. These approaches are called the NCCL ring, tree, collective network (CollNet), and parallel aggregate tree (PAT) algorithms.

The ring approach arranges GPUs in a circular ring where each GPU passes small data segments to the next. In this configuration, every device gradually accumulates portions of the collective result as data travels around the circle. Each GPU sends small segments of data to its neighbor until the complete result circulates around the ring. This method works predictably across many devices and excels when managing large messages in environments with relatively uniform communication costs.

The tree strategy organizes GPUs into a hierarchical tree structure. In this method, GPUs pair up to combine their data into partial results that are then merged further up the tree. This strategy reduces the number of communication steps by allowing simultaneous merging of data across levels. Recursive doubling follows a similar idea by repeatedly exchanging data between paired GPUs in successive rounds. In each round the effective group size doubles until the complete data set reaches every GPU. This process particularly benefits clusters with a power-of-two number of devices because it minimizes the number of phases needed.

CollNet represents a more advanced strategy. It leverages dedicated high-speed interconnects among GPUs to form a collective network. The idea is to divide the GPUs into several smaller trees that aggregate data simultaneously, allowing more even distribution of the communication workload and reducing buffering requirements. This method minimizes the number of communication stages and is especially effective in large clusters where it can scale gracefully from a few GPUs to thousands

More recently, the Parallel Aggregated Tree (PAT) strategy has emerged as a way to improve scalability and load balancing in large clusters. PAT extends the tree approach by organizing GPUs into multiple, smaller tree structures that operate concurrently to better exploit available interconnect bandwidth. Each tree aggregates its portion of the data in parallel, and the partial results from all trees are combined in a final merge. This concurrent aggregation minimizes buffering requirements and spreads the communication load evenly, ultimately reducing latency and improving throughput.

The system may also operate in an auto mode, where NCCL chooses the optimal strategy dynamically based on the topology, message size, and system conditions. By doing so, NCCL ensures that every GPU communicates in an efficient manner that maximizes throughput and minimizes latency during collective operations.

In practice, the environment variable like `NCCL_ALGO` lets you force a specific algorithm such as `Ring`, `Tree`, `CollNet`, or `PAT`. NCCL continues to refine these options to suit varying workloads and system architectures.

Profiling and Debugging NCCL

To debug NCCL, one can use the NCCL Profiler Plugin API to monitor the internal timeline of GPU communications and pinpoint any lagging device or bottleneck in the system. The NCCL Profiler Plugin API is designed to address performance issues that become increasingly difficult to diagnose as GPU clusters scale up.

NVIDIA created this flexible API to simplify the integration of third-party profiling tools (e.g PyTorch Kineto) with NCCL and ensure that complex communication activities are monitored and captured in a clear, hierarchical, and low-overhead manner during runtime execution. The `NCCL_PROFILER_PLUGIN` environment variable governs the loading and initialization of this plugin in a manner similar to other NCCL plugins.

Once loaded, the NCCL Profiler Plugin configures an event activation mask which is a 32-bit integer where each bit corresponds to a distinct NCCL event like group events, collective events, point-to-point events, and various proxy-related operations. This structure creates a natural hierarchy of events to help represent detailed performance information in a meaningful way and pinpoint issues quickly.

The NCCL Profile Plugin API defines five function callbacks. The `init` callback sets up the plugin by providing an opaque context and establishing which events should be profiled. The `startEvent` callback receives an event descriptor from NCCL and allocates a new event object, returning an opaque handle that NCCL uses for further operations. The `stopEvent` callback marks the completion of an event so that its resources can be recycled. The `recordEventState` callback allows the plugin to update events as they transition through different states. The `finalize` callback releases all resources associated with the profiler context once profiling is complete.

NCCL Performance Tuning

NVIDIA's Collective Communications Library (NCCL) plays a critical role in distributed training by efficiently handling data exchanges between GPUs. One of the key performance aspects is how data is transferred directly between GPUs using mechanisms such as intra-node user buffer registration.

NCCL registers GPU memory for direct data transfers over high-speed interconnects like NVLink or even directly to a network card, bypassing the extra overhead of moving data through host memory. This direct path is essential for minimizing latency and maximizing throughput during operations like all-reduce, which aggregates gradients across GPUs.

To optimize these data transfers, NCCL provides several environment variables to tailor its behavior according to your workload needs. Below is a list of some key NCCL environment variables that give you fine-grained control over NCCL's communication behaviors.

`NCCL_DEBUG`

  This controls the verbosity of NCCL's logging. Setting this to `INFO` or `WARN`/`VERSION` for different levels can help you diagnose performance issues and understand how NCCL is selecting algorithms and interfaces. It's a useful first step when troubleshooting low performance or unexpected behavior.

`NCCL_ALGO`

  With this variable, you can force NCCL to use a specific collective communication algorithm (for example, `Ring`, `Tree`, `CollNet`, or `PAT`). Depending on your workload, network topology, and gradient size, one algorithm may outperform the others. Experimenting here can sometimes yield noticeable improvements in performance consistency.

`NCCL_NTHREADS`

This controls how many CPU threads each GPU uses for NCCL's networking operations. If you find that your CPU resources are underutilized, increasing NCCL_NTHREADS can allow more concurrent processing of network tasks, thereby boosting the overall network-link utilization. In distributed training, where large amounts of gradient data need to be communicated, having more threads dedicated to managing these transfers can help reduce bottlenecks and improve synchronization speed.

`NCCL_BUFFSIZE`

This determines the size of the buffer that NCCL uses during communication operations such as all-reduce. By increasing the buffer size, NCCL can send larger data chunks in each communication step. This is particularly beneficial when aggregating large gradients, as it reduces the number of communication iterations required, thereby lowering the overall overhead of the synchronization process. However, note that while larger buffers may reduce the number of operations, they also consume more memory, so it's important to balance buffer size with the available system resources.

`NCCL_PROTO`

This allows you to choose the protocol that NCCL should employ. For instance, `LL` (low latency) or `Simple` are typical values. The low latency protocol may be beneficial for small message sizes, while the simple protocol could work better for larger messages.

`NCCL_IB_HCA`

This setting specifies the InfiniBand Host Channel Adapter (HCA) that NCCL should use for communication. In multi-node environments where RDMA is available, this variable lets you explicitly select the desired interface(s), which is especially useful on nodes with multiple IB adapters.

`NCCL_IB_TIMEOUT`

Use this to adjust the timeout settings for InfiniBand operations. The default value might not be optimal for your network conditions, so tweaking the timeout can help reduce the chance of communication stalls or unexpected timeouts during all-reduce operations.

`NCCL_SOCKET_IFNAME`

For setups where NCCL falls back to TCP (or when using Ethernet directly), you can specify which network interface should be used. This is critical in multi-homed hosts where you want the communication to occur over a specific high-speed interface rather than an unrelated one.

`NCCL_LL_THRESHOLD`

This controls the message size threshold below which NCCL uses its low-latency (LL) protocol for communication. Adjusting this threshold can help you optimize performance for small messages by ensuring the fastest protocol is used when it matters most.

`NCCL_MIN_NCHANNELS` and `NCCL_MAX_NCHANNELS`

These let you control the number of communication channels that NCCL uses. The right number of channels can ensure that data flows efficiently across multiple GPUs, particularly when the interconnect or network is a limiting factor.

`NCCL_NCCL_MAX_RINGS`

You can use this variable to specify the maximum number of rings NCCL should use during collective operations, such as all-reduce. The ring algorithm is sensitive to the number of rings, and tuning this setting can help reduce the communication overhead when aggregating large gradients.

By carefully tuning these environment variables to better match the specifics of your workload and system topology, you can optimize the way NCCL manages GPU-to-GPU communications. This reduces synchronization overhead, allows your training and inference workloads to utilize the available bandwidth more effectively, and ensures that your GPUs spend more time on actual computation and less time waiting on data transfers.

The best settings often depend on your specific hardware configuration, the size of your data transfers, and your particular network environment. Experimentation and performance profiling are critical. Start with debugging enabled using `NCCL_DEBUG`, verify that your expected interconnects are being used with `NCCL_IB_HCA` and `NCCL_SOCKET_IFNAME`, and then fine-tune the throughput-related variables `NCCL_NTHREADS`, `NCCL_BUFFSIZE`, and `NCCL_LL_THRESHOLD` as needed.

Using these tools holistically, along with other system-level optimizations, can lead to significant performance gains especially in large-scale multi-GPU and multi-node environments. If you combine these adjustments with a well-tuned OS and CPU configuration, your overall infrastructure will be better positioned to keep the GPUs fed with data and operating at maximum efficiency.

In-Network Hierarchical Aggregations with NVIDIA SHARP and NCCL

When using network hardware that supports Nvidia’s Scalable Hierarchical Aggregation and Reduction Protocol (SHARP), additional benefits come into play as SHARP shifts some of the heavy lift of all-reduce operations from GPUs to the network switch. The network switch, itself, combines gradient data as it passes through. A report from NVIDIA shows a 2-5× speedup for all-reduce on large AI systems using SHARP.

SHARP requires special hardware, firmware, and drivers. You can check if SHARP is supported using `ibv_devinfo -d <device>` and examining the output for SHARP-related references. Assuming SHARP is enabled, your PyTorch application will automatically use it for collective operations like all-reduce since PyTorch uses NCCL which has built-in support for SHARP. You can also disable SHARP using `NCCL_SHARP_ENABLE=0` in scenarios where you need to troubleshoot performance by constraining the environment in a controlled manner.

Engineers can verify SHARP support with diagnostic commands and adjust application settings if needed. SHARP works seamlessly with NCCL and further illustrates how hardware and software now work hand in hand to manage data movement efficiently.

NIXL for Accelerated Data Transfer and Inference

The NVIDIA Inference Xfer Library (NIXL) is a newer communication library. Released in early 2025, NIXL was specifically designed to allow model inference servers to scale out by transferring resources like the KV-cache efficiently between GPUs. While NCCL is ideal for group communications, NIXL was designed to meet the unique demands of distributed model inference.

For models that generate a continuous stream of tokens (e.g. AI assistants), managing and re-using the compute-intensive key-value tensors generated by the Transformer's Attention mechanism is critical. For context, during a conversation with a Transformer-based model, each new token sent to the model inference server generates a key/value (KV) tensors from the Transformer's Attention mechanism. These KV tensors are reused when generating new tokens for the response. As such, we use a KV-Cache which needs to be shared by different GPUs and compute nodes in a distributed inference cluster.

Prefill and Decode Inference Stages

The inference path of a Transformer-based model is actually split into two different stages. The first stage is often compute-bound as it uses many matrix multiplications to build the KV-cache from the incoming request data (a.k.a "prompt".) The second stage is often memory-throughput bound as it needs to gather the model weights from GPU HBM memory to calculate the next set of tokens (a.k.a "completion" or "response"). In many advanced, high-performance inference systems like Nvidia Dynamo and vLLM, the two stages run on different GPUs or compute nodes as shown in Figure 4-2.

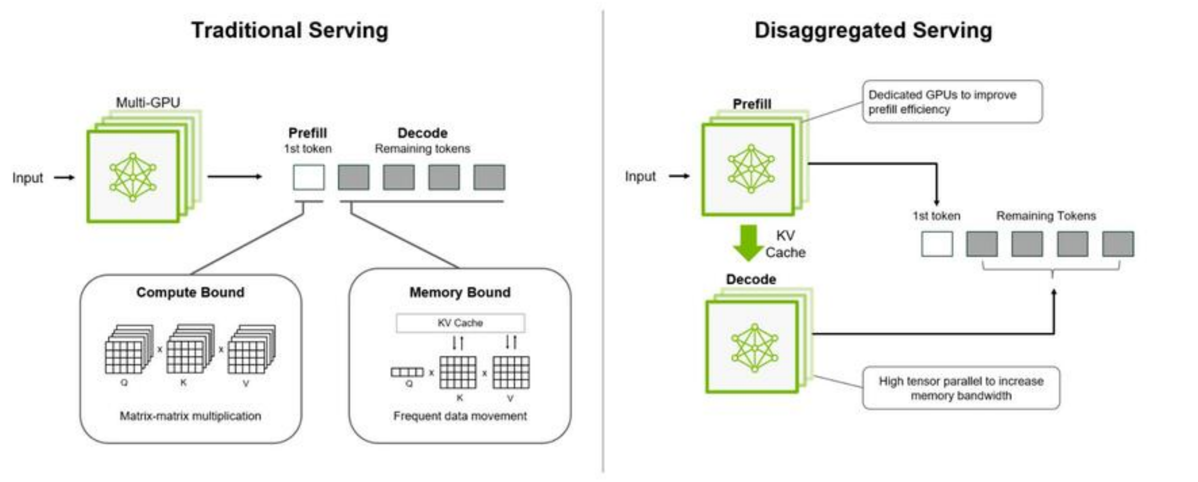


Figure 4-2. Disaggregated serving separates prefill and decode stages onto different GPUs

One set of GPUs takes care of populating the KV-cache while another set produces the final response. In such cases the KV-cache, which can run into tens of gigabytes in a long prompt, must move seamlessly from one processing unit to another in real time.

Traditional methods that pass the data through CPU memory or even storage fall short in meeting the required pace and low-latency experience. NVIDIA created NIXL in early 2025 to tackle this exact scenario. What we really want is a direct, high-bandwidth GPU-to-GPU transfer - across both compute nodes and racks - that overlaps the data the data transfer with computation. This way, the stage 2 decode-focused GPUs can start computing the next token while they’re receiving the KV-

Cache for the next set of input tokens from the stage 1 prefill-focused GPUs.

NIXL provides a direct channel for transferring data from one GPU to another or a small group of GPUs across compute nodes and even across racks. The system looks at the available pathways and always selects the one that gets the data there the quickest. Whether the data travels using NVLink within a server, NVSwitch inside a rack, InfiniBand between machines, or even NVMe storage on demand, NIXL automatically picks the fastest lane.

NIXL Core API for Efficient Data Transfers

Developers interact with NIXL through a very straightforward API. They simply post a request that includes the pointer to the data and the destination tag, and the library takes care of the rest. For instance, NIXL can accept a KV-Cache data send request to a variety of destinations as shown in Figure 4-3.

Figure 4-3. NVIDIA Inference Transfer Engine (NIXL) architecture (Source: https://developer.nvidia.com/blog/introducing-nvidia-dynamo-a-low-latency-distributed-inference-framework-for-scaling-reasoning-ai-models/)

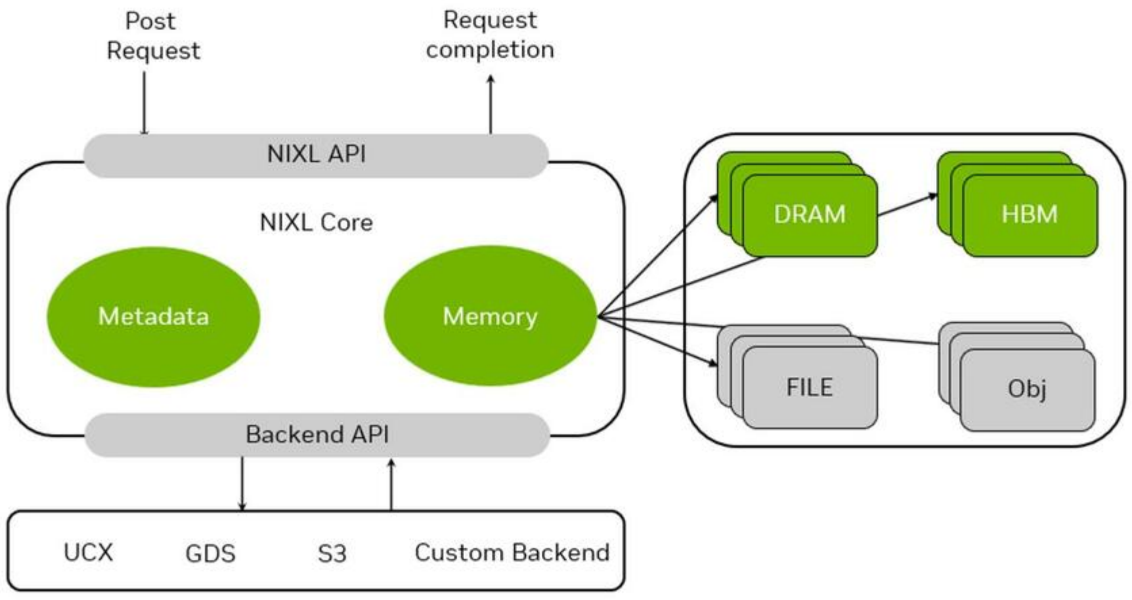


Figure 4-3.

The NIXL Core manages the metadata and memory buffers. The NIXL Backend API interfaces with various transport backends like UCX, GPUDirect Storage, S3, or a custom backend. NIXL can efficiently move data between different tiers such as GPU HBM, CPU memory (DRAM), file storage (NVMe SSD), and object storage. The NIXL API abstracts the complexity of transferring data across heterogeneous memory and storage devices in a distributed inference setting.

NIXL picks the most efficient route for each data transfer and it uses zero-copy transfers when possible to avoid needless copy steps. For instance, NIXL avoids copying data to a bounce buffer in the host's CPU memory.

The library also provides notifications and callbacks so your code knows when a transfer is complete. Notifications and callbacks are crucial for synchronizing with computation as you don't want to start using a chunk of data if it hasn't yet arrived.

It's important to note that NIXL is complementary to NCCL and not a replacement. NCCL still plays a key role when a large group of GPUs needs to work closely together for tasks such as synchronizing a giant model running in parallel. While NCCL coordinates group communications, NIXL focuses on sending large amounts of data from one GPU to another - or to storage if needed.

NIXL Internal Dependencies

Under the hood, NIXL uses Nvidia's Unified Communication X (UCX). UCX is a framework designed for high-performance point-to-point communications. It provides abstract communication primitives that enable applications to make full use of advanced hardware features including active messages, tagged send and receive operations, remote memory read and write, atomic operations and various synchronization routines. UCX supports multiple hardware types such as RDMA offered by InfiniBand and RoCE, TCP networks, GPUs and shared memory.

The UCX framework speeds up development by offering a high-level application programming interface that hides low-level details, yet it still delivers exceptional performance and scalability. It builds on industry best practices learned from applications running in the largest data centers and supercomputers while efficiently handling messages of all sizes.

NIXL also uses GPUDirect RDMA and a technique known as InfiniBand GPU Direct Async (IBGDA) to let a GPU start a data transfer without waiting for the CPU to jump in. In older systems a central processor needed to coordinate transfers even when the data went over RDMA. NIXL moves past that bottleneck so a GPU can send a chunk of data directly to another GPU.

As soon as a GPU in stage 1 finishes writing the KV-cache into its GPU HBM, the data is immediately sent to the target GPU using network resources. If the target GPU is busy with stage 2 and computing a decoded response, it will receive the KV-cache in the background to efficiently overlap computation and communication. NIXL also supports transfers that do not arrange the data in one continuous block. It also handles scattered data.

KV-Cache Offloading with NIXL

NIXL offers the advantage of managing data beyond just GPU memory. If your application requires data to be spilled from GPU memory to the CPU or an NVMe drive, NIXL can still handle these transfers efficiently. It integrates with NVIDIA’s GPUDirect Storage (GDS), which provides an optimized pathway for reading from and writing to NVMe drives by directly interfacing with GPU memory.

TIP

GPUDirect Storage fulfills a similar role to GPUDirect RDMA, with the key difference being that it is tailored for accelerated disk I/O rather than direct GPU-to-GPU communication.

Think of NIXL as a unified data access layer. No matter where your data currently resides (e.g. GPU, CPU, disk) - or where it needs to go - you can use the same NIXL API and it will pick the fastest path available to read and write your data.

This paves the way for NVMe-based KV cache offloading which allows the system to automatically spill the KV-Cache to NVMe storage when it’s not actively needed, then stream it back in when required. If a conversation or session goes idle, its context can be parked on SSD to free up GPU memory for another task, and NIXL can fetch it back later.

NVMe-based KV-caching extends the effective memory available for serving models as you could handle longer conversations or more simultaneous sessions by using disk as overflow. When a conversation or session takes a break the context sits safely on fast storage, freeing valuable memory for other tasks.

Of course, accessing NVMe is much slower than HBM, but with clever overlap, and enormous bandwidth of modern SSDs, and GPUDirect Storage to read directly into GPU memory, some of the NVMe-based KV-cache offloading latency can be hidden behind computation.

NIXL and High-Performance Inference

Systems like NVIDIA Dynamo

The impact of NIXL on performance is huge for distributed inference systems like NVIDIA Dynamo, also released in early 2025. NVIDIA has demonstrated that using Dynamo with NIXL can boost multi-node LLM inference throughput by up to 30× compared to naive implementations that don't overlap transfers or use slower methods.

What was once a major latency barrier – shifting many gigabytes of context data between nodes – becomes a relatively quick, asynchronous operation under NIXL. We will cover Nvidia Dynamo, TensorRT, vLLM, and respective model inference optimizations in depth in Chapter 9.

Storage I/O Optimizations

Feeding data to the GPUs is as important as the compute itself in deep learning training. Consider a 100-trillion-parameter model being trained on hundreds of GPUs. While the compute power is enormous, the compute can stall if data isn't ready in time. Reading and writing to both storage and network can be a bottleneck. A single training process might read hundreds of megabytes or a few gigabytes of data per second when streaming a huge dataset from disk to GPU. If the storage subsystem can’t keep up, the GPUs will stall waiting for data. This lowers the overall goodput of our system.

Large GPU clusters benefit from using a high-performance storage solution such as a parallel file system (e.g. Lustre or GPFS), high-speed network-attached storage (NAS), or local NVMe SSDs in each compute node used for caching data. As GPUs get faster, the demand on the data pipeline increases. Here, we’ll discuss how to optimize storage, data loading, and network usage to keep the GPUs fed. This section covers using faster storage hardware, pre-fetching and caching data, reading a few big files vs. lots of small files, using GPUDirect Storage, overlapping computation and communication, and tuning network settings for large-scale, distributed training and inference clusters.

Fast Storage and Data Locality

Large model training jobs usually need to read huge data sets. For example, it’s common to have billions and trillions of tokens of text, billions of images, hundreds of thousands of hours of audio, etc. If you try to stream this from a single spinning disk, you’ll be wasting a lot of money as your GPUs will be starving for data since the spinning disks cannot stream the data fast enough. That’s why most serious AI systems use either a parallel storage system or large, fast, and local solid-state disks (SSDs).

A parallel file system like Lustre stripes data across many servers and disks, so you can get aggregate read throughput of many GB/s. The cloud analog would be something like Amazon FSx for Lustre sitting on top of Amazon S3. In a lot of cases, however, practitioners usually stage their data locally by copying from Amazon S3 to local SSD for performance. However, Amazon FSx for Lustre is the recommended approach by AWS. Alternatively, if you use something like a network file system (NFS), you might have multiple NFS servers running in the cluster - each serving a portion of the data. The AWS cloud equivalent is Amazon’s Elastic File System (EFS).

One good strategy is to preload and cache data on each node. If each node has a fast NVMe SSD, you can copy a chunk of the dataset to it before training starts. Say you have 100 TB of data and 100 nodes – each node retrieves a 1 TB shard of data locally. Reading from the NVMe disk at 5-6 GB/s (sequentially, not randomly) will be much faster and more consistent than reading over the network from a remote storage. Even if you can’t fit all of the sharded data onto local NVMe, you can cache batch by batch. In this case, you would load the next batch of data shard in the background. Some teams use a data-staging job that runs before the training job. This data-staging job performs the data sharding and distribution to the nodes in the cluster.

When working with a shared filesystem, the data isn't easily splittable. You can enable Linux's page cache to automatically cache file contents in RAM - assuming your dataset fits into RAM. In this case, the data will be cached and ready for subsequent epochs to read the data much faster - without reading from the NVMe disk. If the dataset can't fit into RAM, you will still get a boost for the portion that fits. There are also user-space caching solutions like Linux's `fscache` which could help cache frequently-accessed files from the shared filesystem.

Sequential vs Random Read Patterns

GPUs prefer data in big chunks as they are massively-parallel processors. Fortunately, it's more efficient to read a smaller number of large files than a larger number of small files due to the overhead incurred per read including system calls (a.k.a syscalls), disk-seek time (though very low for SSDs), etc. So if your dataset consists of millions of tiny files, you should consider packing them into larger files. In fact, this is one of the reasons that TensorFlow introduced their TFRecord file format.

Using TFRecord, once can concatenate many small files (often text-based) into a few large files (binary format). PyTorch doesn't mandate a format, but you can use something like the Parquet format - or even just plain numpy files - to pack the data. The idea is to do a fewer number of large sequential reads - and avoid random reads. If you can sequentially read 1 GB of data, for example, any decent SSD can do that in milliseconds. But if you randomly read a 1 KB file here, another 1 KB there, you'll be bottlenecked by excessive I/O latency.

Another trick is to use parallel and asynchronous reads. Most OSes and languages allow you to issue multiple read requests at once. If you have 8 CPU cores working on data loading, then each core can read different files in parallel. The more parallel requests that your storage system can handle, the more throughput you will get - assuming the link between your CPU and storage system (NVMe local disk, remote shared file system, etc) can support the overall throughput from the parallel data loaders.

In Python, using `DataLoader` with `num_workers > 0` will achieve parallel reads by spawning more worker data-loader processes across more CPUs to perform the reads in parallel. In the background, the OS will keep the disk as busy as possible.

Modern NVMe drives have multiple queues and can handle many concurrent requests.

Consider increasing the read-ahead setting for block devices if your access pattern is mostly streaming-reads as this makes the OS read more data than requested in anticipation of additional sequential reads. You can configure the read-ahead setting using `blockdev --setra <kilobytes>` on the device or echo the number of kilobytes to `/sys/block/<device>/queue/read_ahead_kb`.

Read-ahead helps hide the latency of repeated sequential reads. If your data loader requests 4 MB of data from the disk, the OS might actually fetch 8 MB of data knowing you will likely ask for the next 4 MB very soon.

In some scenarios, using memory-mapped files (`mmap`) can also be efficient. `mmap` maps a file into your process's address space so that file I/O is managed by the OS's virtual-to-physical memory mapping subsystem. This is efficient because it lets the OS load data on-demand into the OS page cache.

And when huge pages are used with `mmap`, virtual-to-physical translation performance benefits from larger page sizes and fewer cache misses from the TLB which maintains the translation mapping data. This reduces overhead and improves performance for sequential and predictable access patterns like loading datasets into a training job.

Tuning NVMe and Filesystem

Modern Linux systems use a multi-queue framework called `blk-mq` to schedule and distribute I/O requests across several hardware queues efficiently. When you use NVMe SSDs, these devices already come with highly optimized firmware-level scheduling. So to prevent extra scheduling overhead from the OS on top of the firmware-level scheduling, you can choose to set the OS I/O scheduler to a simpler one such as “noop” or “mq-deadline.”

This configuration means the `blk-mq` multi-queue framework continues to manage the distribution of requests, but it uses a simpler “noop” scheduler that essentially passes the requests directly to the NVMe firmware scheduler to distribute the I/O requests.

Also, make sure the NVMe driver is using all available queues - it should by default. This will maximize I/O throughput. And if you are in a virtual environment, ensure the `virtio` drivers are up to date to ensure optimal I/O performance.

File systems like XFS and EXT4 are common and should both be tuned. XFS is often recommended for parallel throughput on multi-core systems. Ensure mount options aren't introducing overhead. For example, you may want to disable `atime` (access time) and avoid writing the extra metadata if you don't need this information.

Using GPUDirect Storage

GDS is a relatively-new feature which allows GPUs to directly pull data from storage or network without going through the CPU to copy the data. Normally, a read from disk goes into CPU memory through a page cache or buffer that you provide. Then you need to copy that buffer to a pinned (non-pageable) CPU-memory buffer. Only then can the GPU's Direct Memory Access (DMA) engine pull that pinned CPU-memory buffer into its GPU memory.

With GDS, the GPU's DMA engine can initiate reads directly from the storage device into its GPU memory. Using GDS, reading data into GPU memory skips the extra hops through the CPU memory buffers. The obvious benefit of GDS is that it reduces CPU usage since the CPU isn't managing those data transfers - or touching the data in any way. It can also reduce latency slightly.

However, in practice, not all storage stacks are GDS-ready as GDS requires special hardware, software, and drivers. However, a lot of modern, high-end NVMe drives and RAID controllers now support GDS. Programmatically, one can use NVIDIA's `cuFile` library with GDS by reading a file with `cuFileRead` which retrieves the data straight into the GPU's memory buffer.

Reports from VAST Data show a 20% boost in read throughput using GDS on certain AI workloads. In their experiments, they showed that using GDS with just a single GPU could saturate a 100Gb/s network link, get an extra 20% throughput, and cut CPU usage dramatically. As GDS matures and becomes more standard, more training pipelines will adopt it – especially when dealing with ultra-high data-loading rates where CPU was previously a bottleneck in I/O.

It's important to realize that some workloads are not bottlenecked by CPU during I/O. They're typically bottlenecked by the disk's I/O throughput. GDS typically shines if you have multiple GPUs each doing random small reads from an NVMe. The GPUs can do these random reads in parallel and the CPU won't be bottlenecked handling all of the interrupts typically generated for each small read.

Even without GDS, the key principle still remains: overlapping I/O with compute is always ideal. While the GPU is crunching on one batch, your code should already be reading and decompressing the next batch to send to the GPU. This way, when the GPU is ready for the next input, the data is already prepared and in GPU memory. Otherwise, if the GPU finishes and has to wait for the data to be transformed and loaded, this will create a bubble in your data pipeline which will need to be eliminated.

Distributed and Parallel Filesystems

Next, let’s consider distributed filesystem aspects for networking and communication. When all compute nodes read from a single storage server, that server becomes a bottleneck because its resources - especially for file accesses - become overwhelmed. To alleviate this, it’s best to partition, or shard, the data across multiple storage nodes so that each compute node reads only its designated subset.

Sharding data across multiple storage nodes reduces the load on any one single storage node - assuming the data is evenly distributed and no particular shard becomes a “hot spot.” In this case, the storage node which contains the hot spot may become overloaded and you’re back to the original, single-storage node performance bottleneck - though it may take longer to profile and identify.

However, if your dataset comprises many small files, even a well-sharded file distribution can overload the filesystem’s metadata server. Retrieving even simple metadata attributes like file size, permissions, and modification times can collectively cause performance issues when performed at a large scale.

As mentioned earlier in the storage section, a common mitigation for the “small-files” problem is to package many small files into fewer, larger files. If bundling isn’t an option, you may need to scale out the metadata service, itself, to support the additional metadata requests.

Ultimately, the goal is to enable large sequential reads, which parallel filesystems can handle efficiently, so it’s important to monitor your network along with your GPUs and CPUs. If you see GPUs waiting, check your network’s throughput with tools like `nvidia-smi dmon` or NIC counters. Make sure you are not exceeding the network’s bandwidth.

If you have 8 GPUs per node and each GPU is performing, say, 1 GB/s of gradient all-reduce communications, but you have a 100

Gbit/s (12.5 GB/s) network. In this case, your communication is running at approximately 1/12th of the network's overall bandwidth so you are not exceeding the network's bandwidth - assuming nothing else is running over the network, of course.

But if you have 16 GPUs per node which are performing an aggregate of 16 GB/s in all-reduce communications from that single node, you are clearly exceeding the network's overall bandwidth capacity. This is when techniques like compression, accumulation, and batching should be explored. The downside is that these techniques may impact model accuracy. This is a trade-off that needs careful monitoring, profiling, and tuning as it is specific to your AI workloads and use cases.

Monitoring Storage I/O Metrics

For monitoring, you can use tools to show resource usage in near real-time. For example, `iostat` shows I/O disk throughput, `ifstat` shows network utilization, and `nvidia-smi` shows GPU usage. If GPUs aren't near 90-100% during the heavy part of training, find out why. Are the GPUs waiting on data from the disk or network? Or perhaps the GPUs are stalled due to a slow-down in the CPU-based data loading and preprocessing pipeline? By systematically addressing each dependent component including storage, CPU, and network, you can increase your GPU and overall efficiency.

Sometimes a quick fix can work. By doubling the number of data-loader workers, you might go from 70% to 85% GPU utilization. Or switch from loading raw text to loading pre-computed token tensors. This will relieve your CPU from performing the tokenization during the GPU-feeding pipeline which helps feed the GPU quicker. These are low-hanging fruit when it comes to optimizing AI pipelines.

By applying these strategies, you often significantly improve resource utilization, performance, and efficiency. This directly translates to shorter training times, faster inference responses, and overall cost savings. At ultra-scale, this savings is huge.

Tuning the Data Pipeline

Let’s talk about data loading and preprocessing in more detail. Deep learning frameworks like PyTorch provide tools to load and prefetch batches of data in parallel.

GPUs can step in to shoulder compute-intensive preprocessing tasks such as tokenizing text, transforming multi-modal images, videos, and audio. NVIDIA’s Data Loading Library accelerates preprocessing by providing an optimized pipeline to augment datasets.

Equally important is the quality of the dataset used for large language models (LLMs). NVIDIA's Nemo Curator helps transform raw text into polished, domain-specific datasets. This framework automates tasks such as cleaning, filtering, deduplication, and data aggregation to generate high-quality training material.

Tuning the data pipeline is especially important when working with diverse data types that require extra attention. It allows the main training loop to focus on model computation without waiting on lengthy data preparation steps.

Efficient Data Loading and Preprocessing

Overlapping data transfer with computation is critical for maintaining high throughput and minimizing idle time. It's important to adjust configure like the number of workers, prefetch factors, and pinned memory so that you can fine tune your data pipeline to match your workload and system configuration.

Configuring PyTorch's `DataLoader` with `num_workers` will spawn `num_workers` number of worker processes. These workers fetch data from disk, apply any transformations, and place the data into a queue. The training loop in the main process grabs batches from this queue.

If `num_workers` is set properly, by the time the model is done with batch N, batch N+1 is either ready or almost ready. It's common to dedicate many CPU cores to data loading and transforming - especially if the data is complex. If you have 32 cores, you might give 8-16 of them to loading threads. Yes, that's a lot, but if your text requires augmentation and tokenization, for example, these extra cores can easily be saturated.

The ideal number of workers depends on the task and the system resources. You will need to tune this for your workload and environment by monitoring and iterating on the configuration. Let's say you increase PyTorch's `num_workers` from 2 to 4 to 8 CPUs. If you see your GPU utilization going up and your iteration time go down, then you're making progress. At some point, it will plateau to the point where increasing `num_workers` doesn't help because either the disk throughput is saturated - or you start to see diminishing returns from too much CPU usage.

A common pattern to work around this is using a producer-consumer pipeline. The producers are the data-loader workers running on the CPUs, and they keep filling a queue memory buffer with prepared batches on the CPU. The consumer is the training loop running on the GPUs, and they keep taking data from that queue memory buffer on the CPU.

In a producer–consumer data pipeline, you can fine-tune the amount of data prefetched to keep the GPU fed with input without overwhelming system memory. For example, PyTorch's DataLoader has a `prefetch_factor` parameter (defaulting to 2), meaning that each worker will fetch two batches in advance. With 8 workers, up to 16 batches might be loaded ahead of time. While this approach minimizes idle time by preparing data in advance, it's important to monitor memory usage. If the batches are large or prefetching is too aggressive, you could end up consuming excessive RAM. It is important to watch the system memory when using large batches.

To further boost efficiency, PyTorch's `DataLoader` can be configured to use pinned (page-locked) CPU memory by setting `pin_memory=True`. Pinned memory is allocated in a fixed physical region that is not paged out, which is essential for high-performance Direct Memory Access (DMA) transfers. With pinned memory, GPU-to-CPU and CPU-to-GPU transfers can be performed asynchronously. This means that when you call `.to(device, non_blocking=True)` on a batch, the operation is scheduled on a background stream, allowing the next data transfer to overlap with GPU computation. Consider the following PyTorch snippet:

```
import torch
from torch.utils.data import DataLoader

# Create a DataLoader that prefetches 2 batches p
loader = DataLoader(
    dataset,
    batch_size=B,
    num_workers=8,
    pin_memory=True,
    prefetch_factor=2
)

for batch in loader:
    # Asynchronously copy the batch to the GPU.
    batch = batch.to(device, non_blocking=True)
    # While the copy is still happening in the ba
    outputs = model(batch)  # This call will blo
    # Continue with training steps...
```

In this example, each of the 8 worker processes preloads batches of data into pinned memory. Because the host memory is pinned, the asynchronous `.to(device,` `non_blocking=True)` transfer can use DMA for high-speed data copying. Consequently, while the GPU processes the current batch (batch N), the DataLoader is already preparing and transferring the next batch (batch N+1) in parallel. This overlap is critical. And without pinned memory, the system would need to pin the memory on the fly for each transfer which would introduce unwanted latency. In essence, pinned memory ensures that data transfers from CPU to GPU happen more rapidly and concurrently with GPU computation, maximizing overall throughput.

Also, make sure to consider data preprocessing as text tokenization, for example, can be done in the CPU worker processes. Offloading such work to the CPU asynchronously and separately from the GPU running the main training process means that the GPU doesn't have to waste cycles performing this data preprocessing. The new batch of preprocessed data arrives from the CPU's pinned memory into the GPU's memory buffer just in time for the GPU to start processing the data. This increases GPU utilization and useful work, or goodput.

Multi-Modal Data Processing with

NVIDIA’s DALI Library

In some special cases, you might want the GPU to perform compute-intensive preprocessing such as decoding an image, video, or audio in the case of multi-modal model training. In this case, the GPU can use NVIDIA’s Data Loading Library (DALI) which provides an optimized pipeline for decoding and augmenting images, videos, and audio data inputs. If you notice that the CPU is bottlenecked during data preprocessing, you should definitely consider using GPUs for these compute-heavy tasks to improve overall model training performance.

The general rule is to profile the data pipeline and find out which part of your pipeline is the limiting factor. If your GPUs are relatively idle at only 50% utilization and you notice that each batch takes a long time to load, you likely need to tune your data pipeline.

Some options to tune your data pipeline are to increase `num_workers`, use faster disks/storage, and use a simpler data format. Many times, making relatively small changes to your data pipeline can take you from a stalling input pipeline to a smooth one.

To improve your data pipeline performance, consider using multiple workers instead of one, enabling pinned memory, or switching from a simple text-based file format like CSV or JSON to a machine-friendly binary format. The downside of using a binary format, of course, is that you will need a new pipeline - or job - to convert the data from CSV/JSON to binary, but this is usually done offline in batch.

Creating High-Quality LLM Datasets with NVIDIA's Nemo Curator Framework

NVIDIA Nemo Curator is an open-source framework designed to help you curate and prepare custom datasets for LLM training. It is part of the broader NVIDIA NeMo framework, which provides end-to-end solutions for conversational AI and related applications. Curator simplifies the process of transforming raw text data into high-quality, domain-specific datasets by offering a range of functionalities. This includes data cleaning, filtering, de-duplication, and aggregation.

By automating these processes, Curator allows data scientists and developers to efficiently extract and refine the data needed to train or fine-tune LLMs, ensuring that the models are fed consistent and relevant data. This capability is particularly valuable for organizations looking to build bespoke language models that perform well in specialized domains.

Key Takeaways

The key lessons from this chapter remind us that the performance of AI systems is determined by the entire full-stack including software and hardware ecosystems.

End-to-End I/O Optimization Is Critical.

Ai system performance depends on a tight integration of compute, storage, and network subsystems. The chapter makes it clear that the benefits of powerful GPUs are fully realized only when data flows efficiently from storage through the network to the accelerators. Every layer should be tuned to keep pace with the GPUs including NVMe SSDs for local caching and parallel file systems for massive datasets.

Co-Design Hardware and Software.

Specialized NVIDIA libraries like GPUDirect RDMA, NCCL, NIXL, and SHARP encapsulate complex, low-level operations into APIs that allow for direct, zero-copy transfers. They reduce CPU overhead, minimizes latency, and enables the system to leverage advanced hardware features, such as in-network compute and unified data access layers, to accelerate both training and inference.

Tailored Communication Strategies for Diverse Workloads.

The architecture leverages multiple communication algorithms (e.g., ring, tree, CollNet, PAT) to ensure that collective operations are scalable and optimized regardless of system topology or workload size. Choosing the optimal strategy will strike a balance between latency and throughput - especially for large GPU synchronizations, gradient aggregations, and key-value caches.

Optimized Data Transfer Techniques.

Techniques such as zero-copy transfers, CPU threads to appropriate NUMA nodes, and overlapping asynchronous data communication with computation help keep data in motion. These strategies ensure that computation and data transfer occur concurrently, maximizing overall system efficiency and minimizing GPU idle time.

Unified and Scalable I/O Frameworks Support Massive Scale.

Unified APIs and extensible SDKs are important to abstract the complexity of underlying hardware.

Frameworks like NVIDIA’s Magnum IO, which integrates storage I/O, network I/O, in-network compute, and I/O management, provide a cohesive foundation capable of scaling from a few GPUs to thousands.

Intelligent Storage I/O Pipeline and Data Prefetching.

Effective data pipelining relies on optimized storage solutions and thoughtful caching strategies. Preloading data locally via high-speed NVMe SSDs, using GPUDirect Storage for direct data reads into GPU memory, and employing parallel, asynchronous file reads ensure that data is available exactly when needed without stalling computation.

Adaptive Network Tuning and Overlap of Communication with Computation.

Setting appropriate buffer sizes, leveraging RDMA where available, and tuning congestion algorithms (e.g. CUBIC) are critical to harness maximum network bandwidth. In parallel, it’s important to overlap communication tasks with GPU computation through techniques like asynchronous transfers and gradient accumulation. This will further smooth out the data pipeline.

Monitoring, Profiling, and Continuous Tuning.

Tools such as the NCCL Profiler, PyTorch Profiler, NVIDIA Nsight Systems, and real-time telemetry solutions like NetQ and UFM are indispensable for identifying bottlenecks. A proactive performance tuning cycle that adjusts communication strategies, reconfigures data prefetching, and revisits storage parameters will ensure that the system adapts to evolving workloads and hardware environments.

Future-Ready Architectural Enhancements.

Emerging technologies like NIXL and advanced in-network aggregation like SHARP illustrate that the pace of innovation is steering AI system design toward even tighter coupling between data movement and compute. These enhancements promise to further reduce latency, enable efficient key-value cache offloading, and scale inference platforms like NVIDIA Dynamo and vLLM to meet next-generation demands.

Holistic and Collaborative System Design.

Achieving peak performance in modern AI infrastructures is not about optimizing any single component. Instead, it requires a holistic approach that harmonizes compute, storage, network, and software orchestration. Building such a system is a collaborative effort involving hardware tuning, software profiling, and iterative feedback from real-world workloads, ensuring that each part of the stack works in concert to drive down training times and inference latencies.

Conclusion

The evolution of distributed, multi-GPU communication libraries and strategies represents a pivotal shift in high-performance deep learning. By adopting specialized libraries such as NCCL for collective operations, NIXL for efficient inference data transfers, and RDMA for ultra-low latency communication, systems can dramatically reduce data movement bottlenecks. The integration of container runtimes, Kubernetes orchestration, and intelligent scheduling further ensures that these optimized pipelines translate directly into improved training and inference performance. Additionally, by addressing the challenges in storage and I/O through advanced techniques like GPUDirect Storage and intelligent data caching, modern AI deployments can sustain high throughput even as model complexity scales.

Ultimately, this chapter underscores that no single component can provide peak performance alone. It is the careful coordination of high-speed communication, efficient data handling, and system-wide tuning that leads to scalable, robust AI systems capable of tackling some of the most demanding challenges in today's computational landscape. This integrative perspective not only streamlines AI workflows but also lays a strong foundation for future innovations in distributed deep learning.

The bottom line for practitioners is that one doesn't need to invent a custom networking and storage I/O solutions. NVIDIA is providing purpose-built libraries so you can focus on model, application, and system logic rather than network and I/O plumbing. For performance engineers, the lesson is that fast data movement and placement is as critical as raw compute power. The fastest GPU in the world delivers no speedup if it's constantly waiting for data from another GPU or from storage. Libraries and technologies like RDMA, NCCL, NIXL, GPUDirect Storage, and Magnum IO are part of the holistic approach to keep the network and data pipeline flowing smoothly in distributed AI systems.

Chapter 5. AI System Optimization Case Studies

This chapter brings together a range of case studies that show how combining advanced hardware with smart software techniques improves the performance and cost efficiency of large language models (LLMs). The studies illustrate that by embracing new numerical precision modes on modern GPUs such as using 8-bit floating point on Hopper or 4-bit on

Blackwell, engineers can significantly boost the speed of training without losing accuracy.

They also reveal that thoughtful software designs can overcome inherent hardware limits. One example is the development of algorithms that overlap computation with data transfers so that even very large models can be trained on systems with lower memory bandwidth. Another major insight is that systems that unite powerful CPUs and GPUs on the same chip - linking them with very high speed interconnects - reduce delays in moving data and improve throughput considerably.

The case studies further show that intelligent scheduling frameworks and open source tools can double the effective performance on existing hardware by optimizing how work is distributed across devices. They also demonstrate that artificial intelligence itself can assist in fine tuning low level GPU code to create kernels that run faster than those produced by manual efforts. In a broader context, these examples reveal that algorithmic innovations, even in core operations such as matrix multiplication, can yield performance gains similar to those achieved by acquiring new hardware.

Together these learnings suggest that successful improvements in model training and inference depend on a close coordination between hardware capabilities and software techniques. This coordinated approach reduces training time and operating cost - as well as enables the efficient deployment of larger models on smaller systems, paving the way for future advances in artificial intelligence.

OpenAI's Journey to Train GPT 4.5 with New Hardware at Ultra-Scale

This case study follows OpenAI's journey building GPT 4.5, highlighting the challenges, processes, and lessons learned along the way. The project began nearly two years before the training run, with teams from machine learning and infrastructure working together on what was expected to be a 10× improvement over GPT 4. Early on, the team ran de-risking experiments on a new, much larger compute cluster built for extreme-scale training. Their collaborative planning set the stage for tackling issues as they arose during the actual run.

As the training progressed, the teams continuously made adjustments and monitored resource use on hardware they were still learning to master. In the early stages, failure rates were high due to unforeseen hardware faults, network issues, and software bugs in simple, well-tested functions like `sum()`

which, when combined with custom code on a rare code path, could cause catastrophic errors in such a massive system. Fixing these issues became a shared effort that blurred the lines between ML research and systems engineering, highlighting that even small details matter at scale.

A key takeaway was that scaling these training runs isn’t only about adding more compute. It requires a delicate balance between long-term planning and rapid problem solving, ensuring that every fix improves both data efficiency and model intelligence. The teams discovered that through co-designing, choices made for model design (e.g. scaling laws and pre-training objectives), had a direct impact on infrastructure needs (e.g. multi-cluster orchestration and network reliability) and vice-versa. Even with careful upfront planning, some degree of unpredictability remained, and the process demanded persistent monitoring and agile, cross-team collaboration.

Ultimately, the GPT 4.5 training run became a blueprint for hyper-scale training, where incremental improvements in code design, error resolution, and system orchestration coalesced into a model that met its ambitious performance goals and provided a roadmap for the next generation of AI models. This case study reveals that real progress is measured by continually learning from and overcoming unforeseen challenges. Through persistence, teamwork, and continuous iteration, the researchers and engineers transformed every hurdle into an opportunity to refine their processes. This proved that breakthroughs in AI systems performance are achieved by powerful hardware, intelligent, and adaptive engineering practices.

DeepSeek Scales to 671-Billion Parameter Model Despite Hardware Constraints

Sometimes innovation is born from necessity. In 2024, a Chinese organization, DeepSeek, found itself constrained to using only NVIDIA's H800 GPUs due to U.S. export restrictions. The H800 is a bandwidth-limited variant of the Hopper GPU, meaning it has a significantly reduced memory bandwidth and lower-speed NVLink interconnect compared to the H100. In practice, while the H100 offers close to 3 TB/s of memory bandwidth along with enhanced inter-GPU communication, the H800's limited throughput meant that data transfers were slower, which threatened to bottleneck distributed training jobs.

DeepSeek set out to train a massive 671-billion-parameter mixture-of-experts (MoE) language model, called DeepSeek-V3, in this heavily constrained environment. This model uses 64 experts - each a smaller 37B-parameter network. With this architecture, only a fraction of the model is activated at any given time, which helps manage computational loads even with the compact H800 setup.

To work around the environment limitations, DeepSeek implemented a novel DualPipe parallelism algorithm that carefully overlapped 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, thus keeping the GPUs efficiently utilized despite their reduced interconnect bandwidth.

This innovative engineering paid off as DeepSeek-V3 was trained to completion at an estimated cost of only $5.6M in GPU time. This is a fraction of what many had assumed was necessary for a model of this scale using a more capable cluster.

Benchmark evaluations have shown that DeepSeek-V3's performance matches or even exceeds that of OpenAI's GPT-4 on several key metrics, including natural language understanding, reading comprehension, and reasoning tasks. These comparisons were based on standardized tests used across the industry, indicating 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—its specialized reasoning model built similarly to OpenAI’s O1 and O3 series. 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.

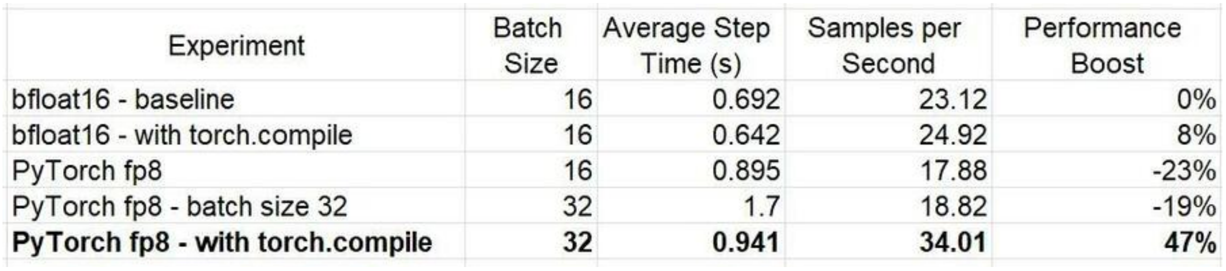
The lessons learned are that large, sparsely activated MoE models can be effectively scaled even on limited memory and compute budgets. Novel training schedules and low-level communication/computation overlap optimizations can overcome hardware limitations, as demonstrated by the enormous ROI of DeepSeek’s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

MobileEye Improves GPU Performance Using FP8 Precision with PyTorch

When the NVIDIA H100 Hopper GPU was released, the AI engineering team at MobileEye upgraded their PyTorch training pipeline to leverage Hopper's support for 8-bit floating point (FP8) math. By using Hopper new Tensor Cores and Transformer Engine for mixed FP16/FP8 precision, they boosted matrix operation performance without losing model accuracy. After manually trying various batch sizes and precisions, MobileEye chose to use PyTorch's `torch.compile` for aggressive kernel fusion at FP8 precision on the H100 which provided a 47% speedup relative to their bfloat16 baseline as shown in Figure 5-1.



Experiment	Batch Size	Average Step Time (s)	Samples per Second	Performance Boost
bfloat16 - baseline	16	0.692	23.12	0%
bfloat16 - with torch.compile	16	0.642	24.92	8%
PyTorch fp8	16	0.895	17.88	-23%
PyTorch fp8 - batch size 32	32	1.7	18.82	-19%
PyTorch fp8 - with torch.compile	**32**	**0.941**	**34.01**	**47%**

Figure 5-1. 47% improvement in average step time using PyTorch compile for kernel fusion at FP8 precision (Source: PyTorch Native FP8 Data Types. Accelerating PyTorch Training Workloads… | by Chaim Rand | TDS Archive | Medium).

The lesson learned was that achieving software speedups on new GPU generations requires careful hardware-software coordination to adjust to newer hardware features and precisions such as Transformer Engine, Tensor Cores, and FP8 precision. In this case, the team at MobileEye used the PyTorch compiler to automatically optimize their PyTorch code for the new FP8 precision supported by the Hopper GPU hardware.

The return on investment (ROI) was clear. With only a few weeks of engineering effort needed to try different combinations and eventually use the PyTorch compiler, the team nearly doubled their training throughput and reduced their training time by half for their Vision Transformer (ViT) model.

This case underscores that embracing new precision formats and compiler optimizations can unlock the performance promised by cutting-edge GPUs. This yields faster results and takes advantage of the full capabilities of the hardware.

Boost Performance with Open Source NVIDIA Dynamo Inference Server

Pure hardware speed, alone, isn't the only path to performance. Intelligent software-level optimizations can create force-multiplying efficiencies across an AI fleet of compute nodes and racks. NVIDIA's open-source Dynamo inference framework proves this point.

NVIDIA's Dynamo inference server is a low-latency distributed serving platform for LLMs. It sits on top of engines such as TensorRT-LLM, vLLM, SGLang, and PyTorch - and efficiently coordinates work across many GPUs and compute nodes.

In one internal test, a team at NVIDIA deployed the Llama language model on Hopper H100 GPUs using Dynamo and immediately saw a 2x throughput increase compared to their custom request-scheduling code. Doubling throughput performance with Dynamo means that an AI service could respond to user queries with the same speed using only half the hardware. This directly translates to cost savings and additional headroom to scale the number of end users even higher.

With the same number of GPUs, Dynamo’s dynamic batching, GPU load balancing, and disaggregated inference serving allowed the system to handle twice as many tokens per second. Disaggregated inference serving separates the memory-intensive KV-Cache prefill phase from the compute-intensive token-generating decode phase.

Once the KV-Cache is populated across the cluster, Dynamo routes incoming requests to the GPU node that already contains the KV-Cache data needed for that specific request. This avoids redundant computations and keeps GPUs busy with useful work.

Moreover, Dynamo can seamlessly allocate different models and stages (prefill or decode) to different GPUs as request load increases and decreases. This ensures that no GPU sits idle while others are overloaded - even across multiple compute nodes and racks.

The lesson learned is that, with ultra-scale inference, the bottlenecks often lie in networking and coordination. Software like NVIDIA's Dynamo inference framework can unlock performance gains on top of even the best hardware.

The ROI here of this experiment is that organizations can essentially get a free 2x performance by using open source NVIDIA Dynamo on their existing infrastructure. This case study shows a dramatic increase in throughput - and lower per-query and per-user cost - without buying new GPUs.

In summary, better algorithms for inference scheduling created direct business value. This makes serving large-scale LLMs more cost-efficient and achievable for organizations of all sizes - large and small.

Efficient Inference with vLLM: High Throughput at Lower Cost

Another real-world challenge in serving LLMs is maximizing utilization of each GPU. A research team at Berkeley originally achieved this goal with their open-source library called vLLM, which, underwhelmingly, stands for “virtual large-language model”. vLLM rethinks how LLM inference servers perform batching and memory-management operations - especially for massive, memory-hungry LLMs and large request inputs.

Traditional model-inference serving engines often leave GPUs underutilized - and cause a massive amount of memory movement - as thousands of independent inference requests accumulate. In such cases, GPU cores can stall waiting for new tokens to be generated.

To address this, vLLM initially introduced two key innovations called PagedAttention and continuous batching. PagedAttention is a smart memory management scheme for paging the attention caches. Continuous batching refers to intelligently packing input requests together to balance overall latency and throughput of the inference system.

PagedAttention is analogous to operating system level memory paging, treating GPU memory as if it were virtual memory. This strategy dynamically reallocates idle portions of the key-value cache, ensuring that no memory is left unused between inference requests, ultimately enabling a more efficient batching of incoming requests and reducing latency.

Continuous batching refers to the method where incoming queries are automatically grouped together as they arrive. This allows the system to form a maximally-sized batch without relying on a predetermined time window. This on-the-fly grouping maximizes GPU efficiency and ensures that the inference process consistently achieves high throughput while keeping latency low.

The vLLM project is well-adopted and well-supported. It’s integrated into many application backends, and included in core cloud-based inference services. The project continues to evolve and benefits from many algorithmic and kernel optimizations - and even supports accelerators beyond just NVIDIA GPUs.

The lessons learned for performance engineers are to look beyond GPU FLOPs and consider how workload patterns affect efficiency. By reducing memory fragmentation and better-combining input requests, for example, vLLM squeezes out significantly more useful work, or goodput, from the same hardware.

The ROI of adopting vLLM is clear for any AI service with spiky or concurrent request loads. Higher throughput per dollar on the same hardware avoids expensive hardware upgrades.

Importantly, NVIDIA recognized the value of such software-level optimizations with its release of the Dynamo serving framework. Dynamo natively supports vLLM as a backend. This allows organizations to combine vLLM's algorithmic and GPU-level optimizations with the cluster-level routing and cache management optimizations of NVIDIA Dynamo.

This case study reminds us that open-source innovations can deliver huge performance leaps. vLLM can increase throughput out of the box without any additional configuration. This translates to lower operating costs, higher request capacities, and better user experiences for the same hardware and model deployment.

DeepMind's AlphaTensor: AI-Discovered Algorithms Boosting GPU Performance

Not all AI optimization happens at the code level. Sometimes, the optimizations go deeper into the realm of algorithms and math. A groundbreaking example comes from DeepMind's AlphaTensor project from 2022 in which AI was used to discover new general matrix multiply (GEMM) algorithms.

GEMMs are core operations that underpin almost all model training and inference workloads. Even a slight improvement in GEMM efficiency can have a huge impact across the entire AI field. AlphaTensor formalized the search for fast algorithms as a single-player game using reinforcement learning to explore many different possibilities.

The astonishing result was that it found formulas for multiplying matrices that proved better than any human-derived method in existence at the time. For instance, it rediscovered Strassen's famous sub-quadratic algorithm for 2×2 matrices as shown in Figure 5-2, but also improved it for larger matrix sizes.

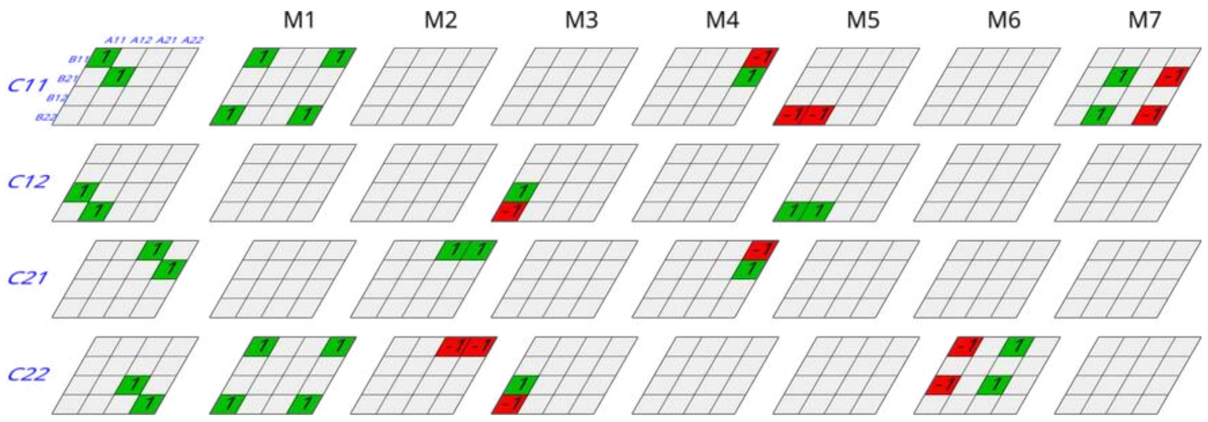


Figure 5-2. Strassen's sub-quadratic algorithm for multiplying 2x2 matrices. (Source: https://en.wikipedia.org/wiki/Strassen_algorithm)

But the real proof came when those algorithms were tested on actual hardware. AlphaTensor discovered a method specific to the NVIDIA Volta V100 GPU generation which multiplied large matrices 10–20% faster than the standard GPU library could at the time. A 10–20% speedup in GEMM performance is huge. It’s like gaining an extra 10–20% in free compute for every model’s forward and backward pass. Such gains typically come from a new hardware generation - or months of low-level CUDA tuning. Yet, in this case, the AI found a better way mathematically in a relatively-short amount of time.

The lesson learned is that there may still be untapped efficiency left to discover in fundamental algorithmic and mathematical operations that human engineers consider novel. The AI can sift through many thousands and millions of variations of algorithms that humans could never try in a reasonable amount of time. For performance engineers, AlphaTensor’s success suggests that algorithmic innovation is not over. In the future, an AI might hand us a new toolkit of faster algorithms for fundamental operations like convolutions, sorting, or Attention.

The ROI in this case is somewhat indirect but very impactful. By incorporating AlphaTensor’s matrix multiply algorithm into a GPU library, any large-scale training job or inference workload would see an instantaneous boost in speed. This could influence everything from graphics rendering to LLM performance to scientific computing. AlphaTensor demonstrated that a 15% speed improvement - over thousands of training iterations on hundreds of GPUs - translates to massive time and energy savings. It’s a return that pays back every time you run the code. Moreover, this speedup was achieved without additional hardware – only smarter software.

For the ultra-scale performance engineer, the takeaway is to remain open to AI-driven optimizations at all levels of the stack. Even the most fundamental, well-optimized operations like GEMMs might leave room for improvement. Letting an AI explore the optimization space - without human bias - can yield high dividends by slashing run-times across the board.

NVIDIA’s AI-Assisted GPU Kernel Optimizations with DeepSeek-R1

Optimizing low-level GPU code has long been an art reserved for expert humans called “CUDA Ninjas”, but it’s been shown that AI is capable of performing these expert tasks. NVIDIA engineers experimented with the powerful DeepSeek-R1 reasoning model to see if it could generate a high-performance CUDA kernel for the complex Attention mechanism that rivaled high-performance, hand-tuned implementations.

Being a reasoning model, DeepSeek-R1 uses an “inference-time” scaling strategy in which, instead of performing one quick pass through the model before generating a response, it refines its output over a period of time - the longer it’s given, the better. Reasoning models like DeepSeek-R1 are fine-tuned to think longer and iterate on their answer. Much like a human who takes time to think through their answer before spitting out a response.

In this experiment, NVIDIA deployed R1 on an H100 and gave it 15 minutes to generate an optimized Attention kernel code. They inserted a verifier program into the generator loop so that each time R1 proposed a kernel, the verifier checked the correctness of the generated kernel code and measured the code’s efficiency. This feedback loop provided guidance for an improved prompt to use for the next kernel-code iteration. The loop continued until the code met strict criteria as shown in Figure 5-3.

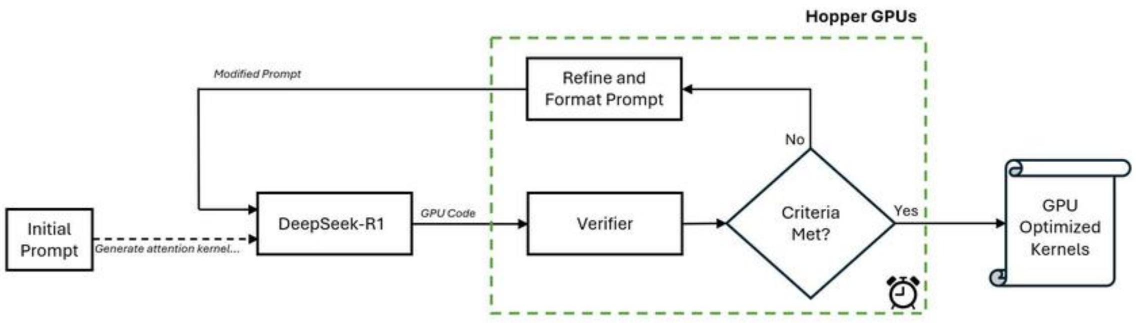


Figure 5-3. Inference-time scaling with DeepSeek-R1 on the NVIDIA Hopper platform (Source: Automating GPU Kernel Generation with DeepSeek-R1 and Inference Time Scaling | NVIDIA Technical Blog)

The following prompt was used:

```
Please write a GPU attention kernel to support r

Use the following function to compute the relati

def relative_positional(score, b, h, q_idx, kv_i
    return score + (q_idx - kv_idx)

When implementing the kernel, keep in mind that 

qk = qk * qk_scale + rel_pos * 1.44269504

Please provide the complete updated kernel code 
```

The outcome was remarkable. Not only did the AI produce a functionally-correct CUDA kernel for Attention, it also achieved a 1.1-2.1× speedup over the built-in PyTorch "FlexAttention" kernel optimized by NVIDIA. Figure 5-4 shows the performance comparison between the generated kernel and PyTorch's optimized Flex Attention across various Attention patterns including causal masks and long-document masks.

Figure 5-4. Performance of automatically generated and optimized Attention kernels compared to PyTorch's Flex Attention (Source: Automating GPU Kernel Generation with DeepSeek-R1 and Inference Time Scaling | NVIDIA Technical Blog)

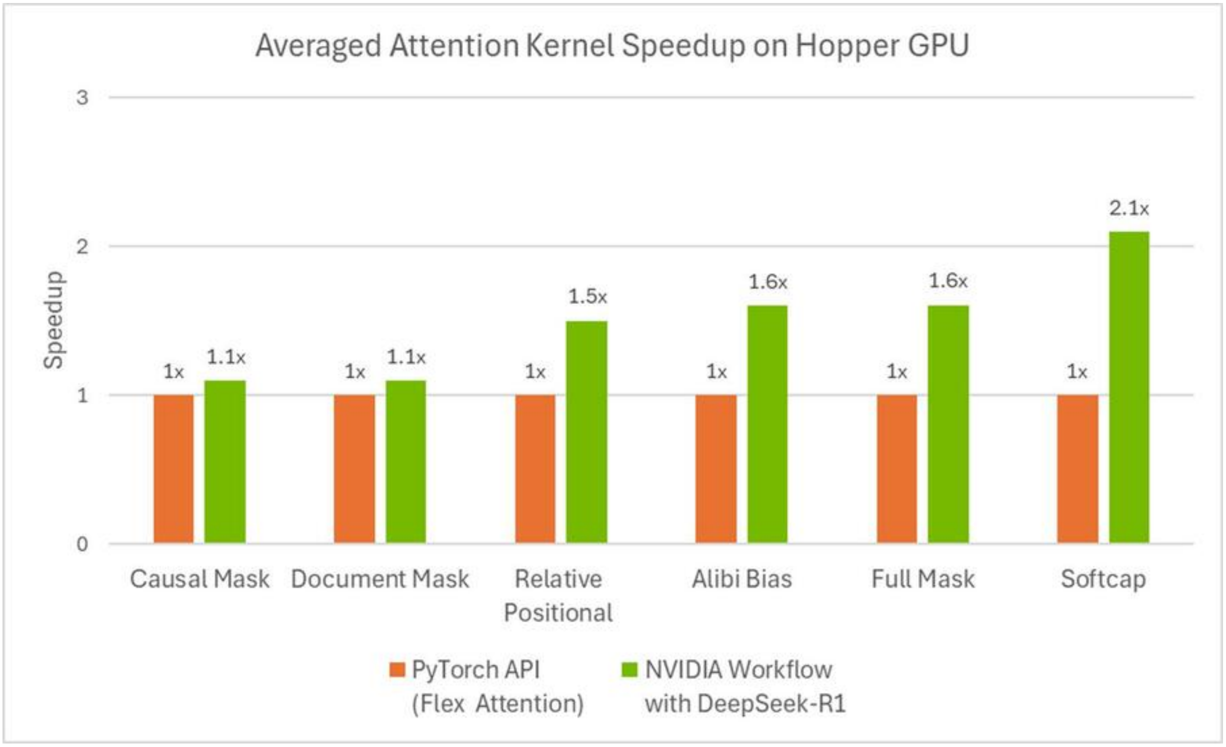


Figure 5-4.

Even more impressively, the AI generated kernels that were verifiably accurate on 100% of basic test cases and 96% of complex cases using Stanford's KernelBench suite. This essentially matches the reliability of a human engineer.

The lesson learned is that giving an LLM the proper tools to verify, critique, and refine its outputs can dramatically improve code quality. Intuitively, this workflow is equivalent to how a human engineer profiles, debugs, and improves their own code repeatedly. What started as a rough code draft generated by the model evolved into a production-quality kernel in just 15 minutes. This illustrates a powerful paradigm for AI-assisted performance tuning.

The ROI is game-changing as even NVIDIA's top CUDA engineers might spend hours or days to hand-craft and test a new type of Attention kernel variant. With this AI-assisted optimization approach, an AI can generate a comparably-efficient, low-level CUDA kernel in a fraction of the time. This frees engineers to focus on higher-level AI system optimizations opportunities and edge cases that may be tricky for an AI to detect and fix.

While some human oversight was still needed, this experiment showed a viable path to reduce development costs for GPU-optimized software with significant runtime performance speedups. For AI systems performance engineers, this type of AI assistance hints that future workflows may involve partnering with AI co-pilots to rapidly co-design optimizations across hardware, software, and algorithms. The AI co-pilot is a force-multiplier for human productivity. Think of these co-pilots as pre-trained and fine-tuned AI interns capable of reasoning through complex problems using their vast knowledge of CUDA tips and tricks derived from existing code bases.

Sakana.ai's Agent: LLMs That Write 100× Faster GPU Kernels

In 2025, A company called Sakana.ai pushed the concept of AI-written GPU code even further by aiming to automate a broad range of kernel optimization tasks. They developed an autonomous agent called The AI CUDA Engineer that takes standard PyTorch operations and converts them into highly optimized CUDA kernels. Under the hood, the AI CUDA Engineer uses LLMs along with evolutionary strategies to iteratively refine kernel code. This strategy creates better and better solutions over multiple iterations and evolutionary generations as shown in Figure 5-5.

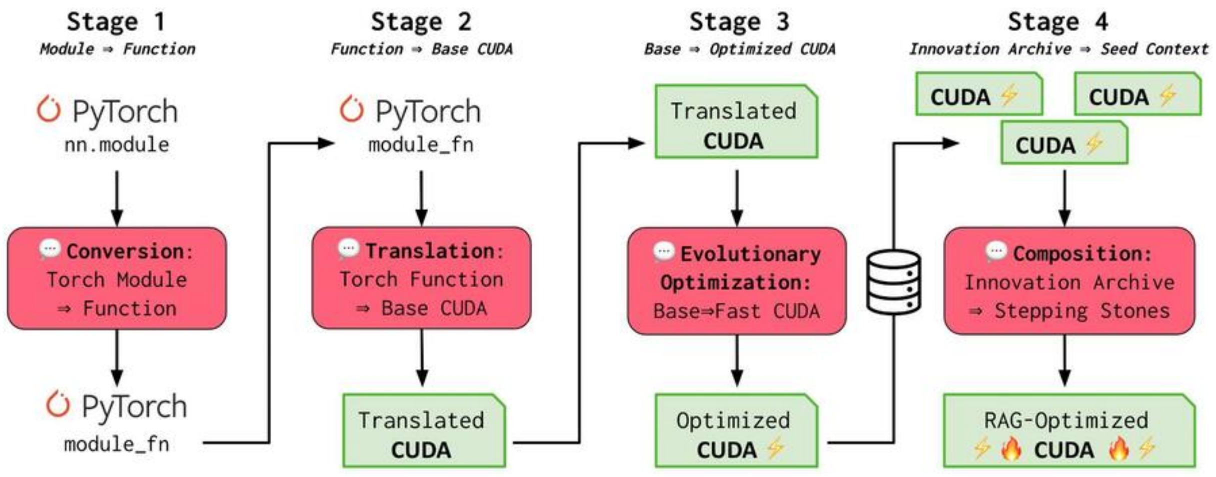


Figure 5-5. High-Level Overview of The AI CUDA Engineer Agentic Framework (Source: *https://sakana.ai/ai-cuda-engineer/*)

The results reported by Sakana are striking. Their AI-generated kernels showed speedups ranging from 10× up to 100× versus the equivalent PyTorch operations. In some cases, the AI CUDA Engineer's generated CUDA kernels were 5× faster than the best CUDA kernels already used in production libraries. Notably, the AI CUDA Engineer successfully translated 230 out of 250 targeted PyTorch ops, showing a highly-generalized approach.

The lessons learned are that a combination of LLMs for high-level code generation 0 and algorithmic optimizations for fine-tuning and testing different code variations - can uncover performance tricks that even experienced code ninjas might miss. For example, Sakana's agent might experiment with unconventional thread block sizes or use shared memory in creative ways that aren't obvious. This could yield big gains for certain operations.

Another example is a complicated sequence of tensor operations in PyTorch that normally would launch many small GPU kernels and suffer from devastating memory bottlenecks. Sakana's AI CUDA Engineer agent could easily detect this and fuse the tensor operations together into a single kernel in a small amount of time - much quicker than fusing these together manually. These types of AI-generated enhancements can create orders-of-magnitude performance improvements in AI systems.

The ROI of such an AI-code assistant is enormous if properly embraced and integrated into a company's development workflow. Instead of manually and laboriously optimizing each neural network layer or operation by hand, the AI CUDA Engineer can automatically produce an optimized kernel on the order of minutes or hours depending on the complexity and novelty of the implementation. This means faster time-to-market for new model architectures and improved inference/training speed - without hiring an army of rare CUDA experts.

In essence, Sakana demonstrated a path to scale expert-level AI performance engineering by using AI. Their small upfront investment in an AI agent yielded compounding returns as many kernels were optimized 10–100× with almost-no human effort.

Predibase's Reinforcement Learning Approach to Generating Optimized GPU Kernels

Another startup, Predibase, demonstrated automated GPU programming by taking a slightly different approach using reinforcement learning. They asked an even-bolder question: Is it possible to train a language model to be a full OpenAI Triton programmer using many examples of PyTorch code? Remember that OpenAI Triton is a Python-like GPU programming language (and compiler) that simplifies GPU programming. The task was to see if the AI could generate efficient Triton code that replaces PyTorch code - and runs much faster on GPUs.

In their experiment, Predibase used a cluster of H100 GPUs and fine-tuned a modestly-sized LLM using a bunch of PyTorch-to-Triton code samples that translated high-level PyTorch functions into efficient Triton kernels. However, because there wasn't a large enough dataset of "PyTorch-to-Triton" examples available at the time to use for supervised fine-tuning (SFT), they created a reward function and used a method called Reinforcement Fine-Tuning (RFT) to guide the model to continuously generate better code using reinforcement learning.

With Predibase’s reinforcement-learning (RL) approach, the AI would first generate a candidate kernel. The system would then automatically compile and test the kernel for correctness and speed. The model received a positive reward if the kernel ran without errors, produced the right results, and ran faster than the baseline kernel.

Through many iterations of this RL-based trial-and-error approach, the model steadily improved. Within a few days of training using merely 13 example problems to start, the AI went from knowing nothing about Triton to producing correct, optimized Triton-based kernel code for various GPU operations. While Predibase hadn’t published specific speedup numbers at the time of this writing, they noted that the model learned to output working Triton kernels in as little as 1000 training steps. Additionally, the model continued to optimize the performance as training continued.

This outcome shows that an AI can optimize its own code by testing, observing feedback, and making adjustments. This is similar to how engineers iteratively refine their code. Reinforcement learning can align AI-generated code with real-world performance metrics by rewarding both correctness and speed. This prompts the AI to explore optimizations like using warp-level parallelism or minimizing global memory access to improve overall performance.

The lesson learned and ROI from Predibase's demonstration is that this type of AI assistance is compelling because it automates performance optimization at the kernel-code level, potentially reducing the need for manual tuning. Instead of engineers manually creating custom kernels for new models, a trained AI assistant can generate multiple variants and select the best one. This shortens development cycles and allows engineers to focus on exploring new model architectures, for example, so that companies of all sizes can achieve cutting-edge, frontier model performance.

This approach also suggests a future where higher-level languages and frameworks, such as Triton and Python, may replace CUDA for GPU programming. Such methods lower the barrier to GPU programming and, in the long-term, could lead to an automated pipeline where an AI agent continuously writes and improves computational kernels, becoming an essential tool for performance engineers.

NVIDIA Grace-Hopper (GH200) Superchip Performance Compared to Hopper (H100)

The NVIDIA GH200 Grace Hopper Superchip represents an impressive convergence of advanced hardware design and innovative software techniques that have redefined LLMl inference. This case study unfolds from two distinct experiments from NVIDIA and Baseten.

Remember that the GH200 superchip unifies a Hopper H100 GPU and an ARM based Grace CPU on a single module with a remarkable 900 GB per second NVLink C2C interconnect instead of a traditional PCIe Gen5 connection. This design offers nearly 7x the bandwidth of conventional PCIe - and dramatically reduces the data transfer delays between the CPU and the GPU.

The journey began in late 2023 with NVIDIA's focus on the MLPerf inference benchmarks using the GPT-J model that contains 6 billion parameters and used for basic generative AI tasks. Engineers at NVIDIA demonstrated that the tightly coupled design of the GH200 reduced CPU to GPU data transfer overhead from 22% to a mere 3% of total inference time. This improvement allowed the GPU to concentrate on computational work and resulted in a 17% boost in throughput per GPU compared to the performance of the H100 system paired with an Intel Xeon CPU over a PCIe link. This result emphasized that memory bandwidth and efficient interconnects can be just as important as raw GPU compute power when running complex inference workloads.

In another phase of the NVIDIA experiment, the emphasis shifted toward overcoming the high memory demands inherent in the Transformer model's memory-hungry Attention mechanism. The NVIDIA team applied an innovative mixed precision approach. The strategy involved a combination of 8-bit and 16-bit precision using the Transformer Engine in Hopper. In this case, they used an 8-bit compressed version of GPT-J's Attention KV-cache. This compression nearly doubled the effective cache capacity of the 96 GB HBM GPU RAM and allowed for much larger batch sizes during inference. At the same time, the Tensor Cores remained fully active. The outcome was that a single GH200 delivered the highest performance per accelerator in MLPerf tests at the time.

Another experiment conducted by Baseten in early 2025 explored the challenges of serving extremely LLMs that exceed the confines of the GPU's HBM memory. They ran the tests with a 70-billion parameter Llama 3.3 model served using one GH200 superchip configured with 96 GB of HBM. The H100 based system had to offload roughly 75 GB of model data to the host CPU memory using the much slower PCIe interconnect. Whereas the GH200 system only needed to offload 60 GB to the CPU memory using its fast NVLink C2C connection.

The performance difference was remarkable. The GH200 achieved a generation rate of 4.33 tokens per second compared to 0.57 tokens per second on the H100 system. This improvement represented a roughly 7.6x increase in throughput and led to an 8x reduction in the cost per generated token. The efficiency gains were driven by a combination of a 21% higher memory bandwidth as well as the integrated CPU memory serving as an extension of the GPU memory without experiencing the typical bottlenecks associated with PCIe.

The final insight is the importance of choosing hardware that is well matched to the specific characteristics of the workload. Certain workloads that fit entirely into the H100's 80 GB of HBM memory do not benefit significantly from the unified CPU and GPU design of the GH200. In these cases, only a modest performance improvement of around 2% was observed. In contrast, when workloads demanded larger memory footprints and extensive data exchange between the CPU and GPU the GH200 outperformed the H100.

This serves as a reminder that expensive hardware investments should be matched to workloads that can fully utilize the advanced memory and interconnect capabilities available in systems like the GH200. Organizations are encouraged to test their specific workloads in cloud instances provided by AWS, GCP, Azure, CoreWeave, or Lambda Labs before making any large capital investment. This way they can deploy the best configuration for their specific workload and use cases.

The case study brings together concepts that intertwine hardware innovation and software optimization in a manner that fundamentally changes the approach to LLM inference. By increasing memory throughput, optimizing data transfers, and supporting mixed-precision formats, the GH200 delivers significant improvements in throughput and cost efficiency. This demonstrated impressive technical advancements and provided clear guidance for organizations aiming to serve larger models on smaller clusters at lower operational costs.

High-Speed Inference with the Grace-Blackwell NVL72 Rack System

NVIDIA's Blackwell GPU architecture came with unprecedented inference speed for large models. In early MLPerf tests, servers packed with Blackwell-based GPUs shattered records. NVIDIA then combined its ARM-based Grace CPU with their Blackwell GPU to create the Grace-Blackwell superchip.

Next, NVIDIA connected 36 Grace-Blackwell superships (each with 1 Grace CPU and 2 Blackwell GPUs) into a single ultra-dense rack deployment called the GB200 NVL72. These superchips are linked by 5th-generation NVLink switches for fast data sharing. With this system, NVIDIA achieved 3.4× higher throughput per GPU when running the 405-billion parameter Llama 3.1 model on the GB200 NVL72 compared to the previous-gen Hopper-based systems.

Blackwell's improvements, like a second-generation Transformer Engine and new FP4 precision, were key. By leveraging FP4 data types for quantized inference, the Blackwell-based GB200 NVL72 system doubled math throughput per GPU relative to FP8 precision.

Crucially, accuracy stayed within target ranges for the benchmark, so this performance vs. precision trade-off was essentially neutral. At the system level, the gains compounded. The GB200 NVL72 rack delivered up to 30× more throughput on Llama 405 inference tasks vs. a similar Hopper-based cluster as shown in Figure 5-6.

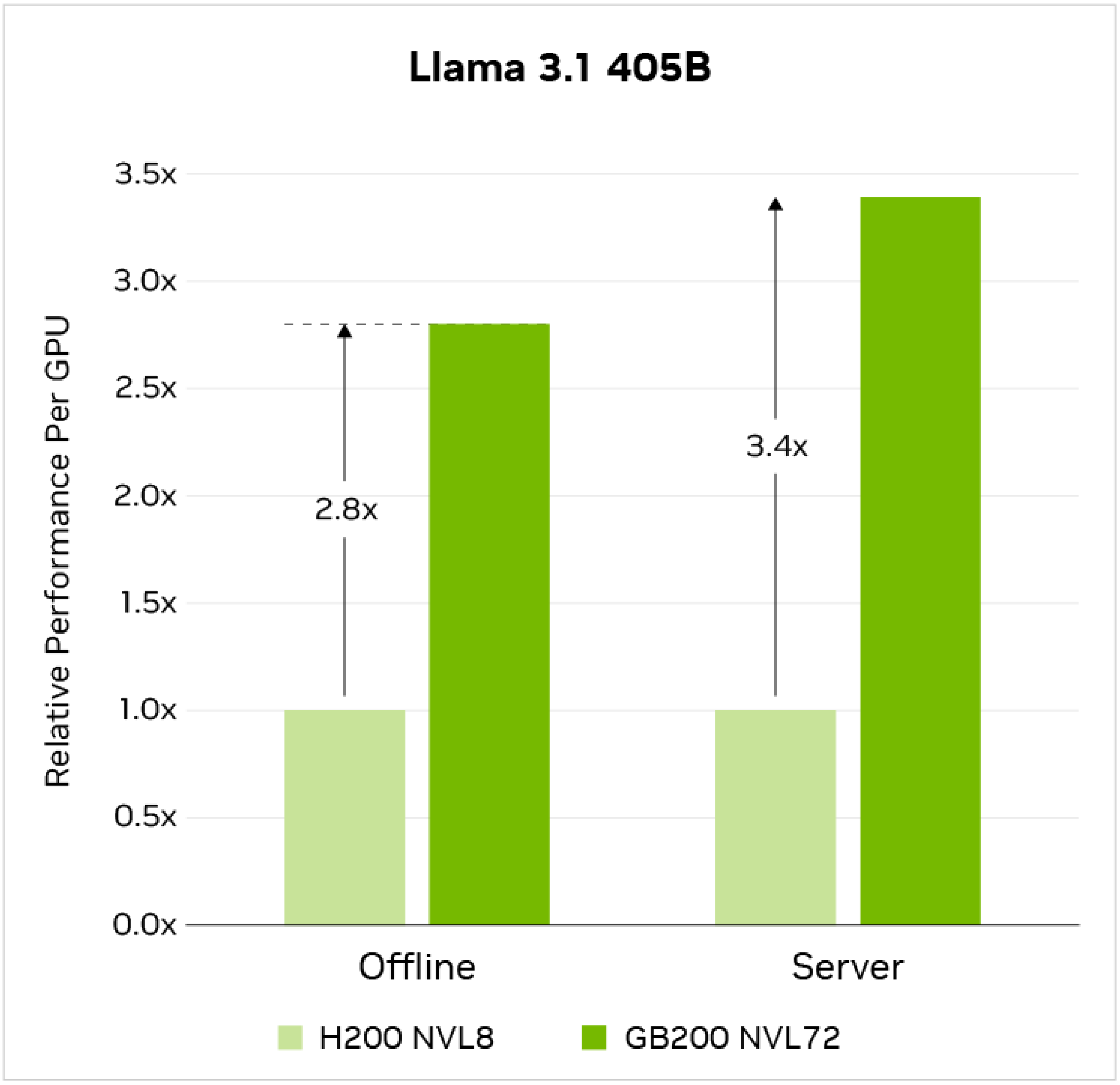


Figure 5-6. NVIDIA’s Grace-Blackwell “GB200 NVL72” supercluster (dark green) delivered up to 30× higher LLM inference throughput than the previous generation (light green), thanks to 9× more GPUs and big per-GPU speedups (Source: NVIDIA Blackwell Delivers Massive Performance Leaps in MLPerf Inference v5.0 | NVIDIA Technical Blog).

This was due to a number of factors including the per-GPU speedup, faster NVLink/NVSwitch interconnects, and the sheer scale of 72 GPUs working together in close proximity. At the time, the GB200 NVL72 set a new industry standard for latency and throughput at scale with “the 405” as Llama’s 405-billion parameter models are sometimes called.

TIP

Mark Zuckerberg, CEO of Meta, casually refers to the Llama 405-billion-parameter model as “The 405” in reference to how California residents refer to their freeways. Specifically, Californians prepend “the” before the freeway number (e.g. “the 405” freeway.)

Additionally, it’s worth noting that even a smaller 8-GPU Blackwell server, called the DGX B200, showed 3.1× higher throughput on a Llama-70B chat model compared to a similar 8-GPU Hopper server. This is a testament to Blackwell’s architectural advantages - even at equal GPU count.

The lesson learned for AI systems performance engineers is that new hardware features like FP4 Tensor Cores - and increased NVLink bandwidth between GPUs - can yield order-of-magnitude gains when combined with larger degrees of model parallelism enabled by designs like the GB200 NVL72.

Upgrading from Hopper to Blackwell achieved immediate return-on-investment as Hopper-based GPU inference responses took 6 seconds vs. 2 seconds on a Blackwell GPU. As such, a cluster of Blackwell-based GPUs could handle many more users with the same hardware. Investing in the Grace-Blackwell platform yielded incremental improvements and a dramatic step-function increase in serving capability. This is a competitive edge for any AI provider dealing with ever-growing model sizes and user demand.

Faster Experiments and Insights for Trillion-Parameter LLMs with Grace-Blackwell Clusters

NVIDIA’s reference design for training trillion-parameter models is the GB200 NVL72 Grace-Blackwell rack. This is a powerful server rack that unites 72 Blackwell GPUs and 36 Grace CPUs in one memory-coherent domain.

Benchmarks indicate that, for enormous 405-billion parameter models, the GB200 NVL72 can train enormous models faster than previous generations. An independent analysis by Adrian Cockcroft, a distinguished technologist renowned for his transformative work in cloud computing and system architecture, found a 4× speedup to be plausible when comparing a fully equipped Grace-Blackwell configuration against a high-end Hopper-based cluster.

The key contributors to this performance boost include Blackwell's higher FLOPs per-GPU, Grace's added CPU memory capacity, faster NVLink-C2C interconnects, and the sheer scale of the single-rack system. In practical terms, if a model previously took 10 days to train on a cluster of Hopper GPUs, the new Blackwell cluster could finish in about 2–3 days. This dramatically accelerates time to insight and increases the rate of experimentation.

The lesson learned for AI systems engineers is that scaling is not always linear with respect to the amount of hardware thrown at the problem. Scaling efficiently requires eliminating bottlenecks at every level. The Grace-Blackwell design tackles several bottlenecks at once including faster GPUs, faster matrix operations on dedicated Tensor Cores, more memory per GPU for larger batch sizes, and an improved NVLink and NVSwitch architecture to link dozens of GPUs with low latency. All of this yields better scaling efficiency. Training jobs and inference servers that used to struggle to fully utilize 72 H100 GPUs now achieve near-linear scaling efficiency on 72 Grace-Blackwell GPUs in the NVL72 configuration because the communication overhead is so much lower.

The ROI when using Grace-Blackwell for training workloads is measured by faster training runs and quicker experiments. Consider an enterprise training a 500-billion-parameter LLM on their specific datasets. With the GB200 NVL72 systems, that can reduce training time from 4 weeks to 1 week, for example. This can be transformative as models reach production faster and use less energy per insight gained.

Thus, although the upfront cost of a GB200 cluster is hefty, organizations see value in the form of accelerated R&D and potentially lower cost when measured per model trained and insight gained. In sum, this case demonstrates the ROI of adopting next-generation hardware for large-scale LLM training including faster experiment results and better throughput per dollar. This enables organizations and research labs to push the AI frontier without breaking the bank.

HPE's Grace-Blackwell Supercomputer for the Trillion-Parameter Era

In early 2025, Hewlett Packard Enterprise shipped its first NVIDIA Grace-Blackwell system called the HPE Cray XD. This is their implementation of the GB200 NVL72 rack specification. This marks one of the first real-world deployments of this technology outside of NVIDIA’s own labs. The HPE Cray XD rack-scale system is built for organizations that need to train and serve models at the 1-trillion parameter scale using a single, unified memory space.

HPE emphasizes that such systems offer lower cost per token training and best-in-class throughput for ultra-scale models (HPE announces shipment of its first NVIDIA Grace Blackwell system | HPE). This means that even though the absolute cost of their rack is high, the efficiency - in terms of tokens processed per second per dollar invested - is significantly higher when models are at the trillion-scale. The target users are cloud AI service providers and large enterprise research groups.

The lessons learned by the HPE engineers who first used the HPE Cray XD are that, with all 72 GPUs in one domain, debugging and optimizing parallel training jobs became easier than on a traditional 8-GPU-per-node cluster as there were fewer moving pieces in terms of network communication patterns. However, they also learned about failure modes unique to the NVL72 system such as faults in the NVLink/NVSwitch fabric. This type of failure could impact many GPUs at once with the NVL72 rack design. Previously, a bad InfiniBand link would affect only one node.

For the pioneering customers of these systems, the ROI of embracing the NVL72 rack design is a competitive advantage. HPE and its customers can now fit massive models into memory and can train much quicker than their competitors. Additionally, by delivering the Cray XD as a single integrated solution, their customer's time-to-value is significantly accelerated. Organizations can now spin up a 1-exaFLOP AI supercomputer and get to work within days instead of months or years. The fast time-to-deployment allows HPE's customers to purchase a turnkey solution for giant AI models. This de-risks their investment, speeds up their time to ROI, and yields massive AI system performance breakthroughs.

Training and Serving a 100-Trillion-Parameter Model

Envision a (not-so-distant) future where an AI research lab attempts something extraordinary like training a dense 100-trillion-parameter transformer. This is roughly 100× larger than today's large models which are measured in the tens or hundreds of billions of parameters. How could one possibly accomplish this with Grace-Blackwell (GB200) superclusters?

In this hypothetical scenario, the lab assembles a training cluster composed of multiple GB200 NVL72 units – say 8 racks, for a total of 576 Blackwell GPUs and 288 Grace CPUs networked together. Even using Blackwell's hefty 192 GB of HBM RAM per GPU, a single 100-trillion model would barely load into the collective GPU HBM RAM of the 8-rack system - even at a reduced 8-bit precision.

For example, at 8-bit precision, 100 trillion parameters would require 100 TB or 100,000 GB. 100,000 GB / 192 GB HBM per GPU = 520 GPUs. We only have 576 in an 8-rack system, so the model would load, but it would be impossible to train entirely in GPU HBM RAM as training requires additional gradients, activations, and optimizer states which add significant overhead - beyond the capacity of a single rack.

Since a 100-trillion-parameter model won't fit into an 8-NVL72 rack ultra-cluster. As such, the team will need to leverage the massive pool of Grace CPU memory to spillover memory from the GPU as needed. This requires that the system stream weights in and out of CPU RAM over the NVLink 5 interconnect within a rack, or InfiniBand between racks. Of course, you would want to avoid as much cross-rack communication as possible.

One would almost certainly want to employ tensor and pipeline parallelism to divide the massive network across GPUs. Another option is to split the model across For example, with pipeline parallelism, each GB200 rack would be responsible for 1/8 of the layers. However, this will require inter-rack communication between the layers - slowing down both inference and training.

Remember that within a rack, the 72 GPUs are all interconnected at full NVLink bandwidth and handle their portion of the model with high-speed coordination. However, between racks, slower InfiniBand links connect the NVLink domains. The engineers, mindful of the hierarchy, design an efficient, pipelined, and parallel compute/communication schedule. For example, as one rack finishes computing its layers for a given batch, it passes the intermediate results (activations) to the next rack's layers for processing. At the same time, the first rack starts computing the next batch - and so on.

By combining data, pipeline, and tensor parallelism (3D parallelism) one ensures that all GPUs are kept busy despite the inter-rack communication delays. To further reduce strain, one would quantize activations down to FP8 during transit - and apply efficient all-reduce gradient-update algorithms so that cross-rack synchronization is minimized at each step. This avoids the dreaded stop-the-world synchronization barrier for distributed training.

One would also consider a mixture-of-experts (MoE) approach by dividing the 100-trillion-parameter model into 100 separate 10-trillion-parameter expert models. This would mean at any given token, only a subset of the experts (racks) are active. This would significantly cut the computation and communication per step.

Using MoEs, however, adds the additional challenge of gating the network, routing tokens to the correct experts, and ensuring a balanced load on experts. Techniques like dynamic expert scaling could be used so that all racks are utilized evenly to avoid network hotspots - even as different experts are activated.

On the software side, an orchestration layer such as NVIDIA Dynamo - working with the TensorRT-LLM or vLLM serving engine – would be required to coordinate work across the large number of GPUs in the system. It might assign micro-batches to different pipeline stages, handle the GPU failures by remapping the GPUs, and optimize the reuse of the massive KV-cache across the cluster.

Serving inference for a 100-trillion-parameter model would be a massive undertaking. The team might use Dynamo's disaggregation to host the model's weights in CPU memory and only bring the necessary experts/layers into GPU memory as needed for a given query. In this case, the system would need to aggregate together a response using the partial results computed on different racks.

The team would also lean heavily on 4-bit weight quantization for inference using Blackwell's FP4 support to shrink the active model footprint to around 50 TB. This is still huge, but within the realm of a multi-rack system's capabilities.

The lessons from this thought experiment highlight which ultra-scale performance tips are needed including hierarchical parallelism, aggressive quantization and sparsity, and sophisticated scheduling. Otherwise, this is an intractable problem. Both the model's internal neural-network architecture - and the physical cluster network - must be co-designed such that every latency and bandwidth consideration is a first-class concern.

The hypothetical ROI for successfully training a 100-trillion-parameter model would be groundbreaking. Such a model could exhibit unparalleled capabilities, achieve leaps in accuracy, or handle extremely complex artificial general intelligence (AGI) or artificial super intelligence (ASI) tasks.

However, the costs and risks are very high. Training might cost tens or hundreds of millions of dollars in GPU compute time. The engineering must ensure that each of the 576 GPUs is utilized to its maximum the entire time. Any inefficiency could waste compute hours and money. Techniques proven at relatively-small scale - such as those used by DeepSeek to train their V3 model on a limited-capability NVIDIA H800's training cluster - would be mandatory at this 100 trillion parameter scale.

If done right, the payoff is a first-of-its-kind AI system and a place in the history books. If done poorly, one could burn through a fortune with little to show. Thus, this hypothetical case underscores the reality for performance engineers - at extreme scale, planning, optimizing, profiling, and iterating are the only way to make the impossible possible. The idea is to efficiently convert ultra-scale hardware investments into unprecedented AI capabilities.

Key Takeaways

Co-Designed Hardware and Software Optimizations.

Performance improvements in LLMs are truly achieved by breakthroughs coming from tightly integrated hardware and software co-designed innovations.

Numerical Precision Advances.

Using newer numerical precision modes like 8-bit on Hopper and 4-bit on Blackwell can significantly increase training speed without sacrificing accuracy. This shows how even small changes in how numbers are represented can lead to big performance wins.

Overlap of Computation and Data Transfer.

Several case studies highlight how algorithms that overlap computation with data transfers help offset hardware constraints such as lower memory bandwidth.

Advanced Scheduling and Batching.

Intelligent scheduling frameworks and dynamic batching techniques used in NVIDIA Dynamo and vLLM maximize GPU utilization and enable existing hardware to achieve higher throughput.

Compiler and Kernel Innovations.

Techniques such as aggressive kernel fusion in PyTorch and AI-generated GPU kernels demonstrate that software innovations can unlock performance gains that rival or surpass the improvements expected from new hardware.

Cross-Team Collaboration is Important.

Hyper-scale training is as much about agile, cross-team collaboration and iterative troubleshooting as it is about raw compute power. The OpenAI case study serves as a blueprint for balancing long-term planning with rapid problem solving.

Clever Software and Algorithms can Workaround Hardware Limitations.

Despite being constrained to lower-bandwidth GPUs (NVIDIA H800), DeepSeek succeeded in training a 671-billion parameter mixture-of-experts model through custom parallelism techniques (DualPipe) and smart algorithm design. This demonstrates cost-effective innovations driven by hardware limitations.

Leverage Compilers When Possible.

By leveraging new Tensor Cores and the PyTorch compiler for FP8 precision, MobileEye nearly doubled its training throughput, illustrating that embracing new precision formats and compiler optimizations can deliver major efficiency gains.

Self-Optimizing Algorithms.

Deepmind's AlphaTensor showcased how AI can discover new, more efficient GEMM algorithms. This improved performance by 10–20% without additional hardware.

Embrace AI-Assisted Coding and Optimizing.

Nvidia's experiments with DeepSeek-R1, Sakana.ai's AI CUDA Engineer, and Predibase's RL approach prove that LLMs can be used not only for high-level tasks but also for generating, testing, and refining low-level GPU code, delivering orders-of-magnitude improvements in speed with dramatically reduced engineering time.

Unified CPU-GPU Superchips Reduce Bottlenecks.

Advanced systems like NVIDIA's GH200 Grace-Hopper and Grace-Blackwell platforms show that integrating CPUs and GPUs on the same chip and linking them via high-speed interconnects can reduce data transfer bottlenecks and improve throughput. This kind of integration can also dramatically lower the cost per generated token in inference tasks.

Design for Rack-Scale and Ultra-Scale Infrastructure.

Deployments such as the GB200 NVL72 rack system and HPE's Grace-Blackwell supercomputer underline that moving from traditional clusters to highly integrated systems can yield huge improvements in throughput and significant reductions in latency.

Consider Strategies for 100-Trillion-Parameter Models.

Training models with 100 trillion parameters will require a blend of aggressive quantization, multi-dimensional parallelism (data, pipeline, tensor, expert, and sequence), and careful orchestration of inter-rack communication. This stresses that future AI scaling depends on both hardware capabilities and the ingenuity of software-level scheduling.

Strive for Both Cost Efficiency and Throughput Improvements.

Many of the case studies emphasize that software improvements and AI-driven kernel optimizations can yield a free boost in performance. These enhancements lower the cost of training and inference, allowing organizations to serve more users or experiment with larger models without proportionately increasing their hardware investment.

Increase Time-to-Insight and Productivity.

Faster training cycles and more efficient inference enable organizations to bring advanced AI products to market sooner. This is critical in an era where time-to-insight is as valuable as the raw performance improvements.

Conclusion

The journey through these case studies narrates a pivotal era in AI systems performance engineering marked by the seamless integration of NVIDIA's groundbreaking GPU and CPU architectures including the rack-scale Grace-Hopper and Grace-Blackwell systems of today - as well as the Vera-Rubin and Feynmann systems of tomorrow. By fusing the CPU and GPU into a superchip module, NVIDIA has redefined the capabilities of LLMs and achieved unprecedented levels of efficiency and scalability.

Central to these advancements is the GB200 NVL72 system which connects 72 GPUs into a unified processing unit and delivers extreme throughput in real-time for trillion-parameter

LLM inference. This performance is facilitated by the second-generation Transformer Engine, supporting FP4 and FP8 precisions, and the fifth-generation NVLink, providing 130 terabytes per second of low-latency GPU communication. Software breakthroughs complement these hardware innovations. For example, software for inference - like NVIDIA Dynamo and vLLM improve dynamic scheduling and resource allocation algorithms - enhance processing speed and increase energy efficiency.

These case studies underscore the importance of a co-design approach, where hardware capabilities and software strategies are developed in tandem to meet the demands of ultra-scale AI models. This collaborative strategy has resulted in significant reductions in training durations, lower inference latencies, and decreased operational costs, delivering substantial returns on investment.

AI-driven CUDA coding agents - demonstrated by organizations such as NVIDIA, Sakana.ai, and Predibase - highlight how AI can be used to optimize AI. Together, humans and AI agents can work together to create substantial performance gains.

In a rapidly evolving landscape where even modest performance enhancements can yield significant competitive advantages, these case studies provide a comprehensive roadmap for future advancements in AI systems performance engineering. They illustrate that through the strategic integration of advanced hardware and intelligent software solutions, the deployment of trillion-parameter and mluti-trillion-parameter AI models is both feasible and very much achievable. This paves the way for the next generation of AI applications and research initiatives.

Chapter 6. Future Trends in Ultra-Scale AI Systems Performance Engineering

The case studies in the previous chapter gave us a snapshot of some real-world, full-stack, and ultra-scale AI optimization studies. Looking ahead, AI Systems Performance Engineers will face an exciting mix of challenges and opportunities. The next era of AI models will demand bigger and faster hardware - as well as smarter and more efficient ways to use it.

Key trends on the horizon include a heavy focus on massive AI data centers, intelligent software toolchains, AI-assisted system optimizers, inference-focused optimizations, faster optical interconnects, offloading compute to network hardware, 3D GPUs and memory, energy efficiency and sustainability, the emergence of hybrid classical-quantum computing, and innovations that help scale toward 100-trillion-parameter models. In this chapter, we'll explore these trends, keeping our focus on practical insights and best practices for performance engineers.

Convergence of AI and Scientific Computing

Over time, the distinction between an "AI supercomputer" and an HPC-based "scientific supercomputer" will dissolve. We're already heading that way since modern GPU-based systems like the GB200 NVL72 are capable of traditional HPC simulations. Conversely, scientific applications are increasingly using AI methods to accelerate discovery. In the near future, GPUs will be truly universal computing engines for both simulation and AI.

In fact, in one NVIDIA blog post, the company highlighted monumental science boosts with Blackwell including weather simulations running 200× cheaper on Blackwell than on Hopper - and consuming 300× less energy for certain digital twin workloads. These numbers were achieved on the same hardware that also delivers AI breakthroughs.

The trend is a fusion of workloads in which an exascale weather simulation might directly incorporate neural network components for sub-grid physics, running in tandem on the same GPUs. We'll also see AI models trained specifically to enhance and steer simulations running concurrently on HPC infrastructure. This convergence will push system software to support more heterogeneity. Scheduling software will efficiently intermix AI training jobs with HPC jobs on the same GPU pool. Programming models might allow seamless switching between Tensor Core operations and classic double-precision arithmetic within a single application.

It's expected that many scientific breakthroughs will come from simulations boosted by AI such as a molecular dynamics simulation where critical quantum chemistry calculations are accelerated by an AI model - all executed on the same machine. NVIDIA is clearly preparing for this future by ensuring their GPUs excel at both HPC and AI domains. The fact that Blackwell GPUs will be 18× faster than CPUs in scientific computing tasks and simultaneously perform well on AI tasks is an early sign.

In fact, we likely won't be talking about "AI systems" and "HPC systems" separately. There will just be accelerated computing systems that handle the full spectrum of computational problems. For performance engineers, this means techniques and optimizations from the AI world - such as mixed precision and sparsity - will benefit HPC. And vice versa, HPC techniques and optimizations - such as numerical stability techniques, scheduling algorithms from HPC will benefit AI training.

This trend towards HPC and AI convergence ultimately leads to more robust, versatile computing platforms. The supercomputer of the future might solve a physics problem in the morning and train a massive AI model in the afternoon, with equal proficiency.

Massive Data Centers Powering Self-Improving, Perpetually-Learning AI

Agents

In early 2025, a report from the AI Futures Project describes a series of milestones and AI models/agents that measure technological progress, enhance research speed, and provide transformative benefits for AI research and development over the next few years. The report describes how the frontier AI labs are currently designing and building some of the biggest AI data centers the world has ever seen.

These superclusters will provide exponentially more compute than previous systems and enable a massive leap in model performance. For context, training GPT 3 required 3×10^23 FLOPs and GPT 4 required 2×10^{25} FLOPs. These new data centers are engineered to train models with 10^27 and 10^28 FLOPS as shown in Figure 6-1. It describes a fictitious Agent-1 model trained on 2 orders of magnitude more compute FLOPs than .

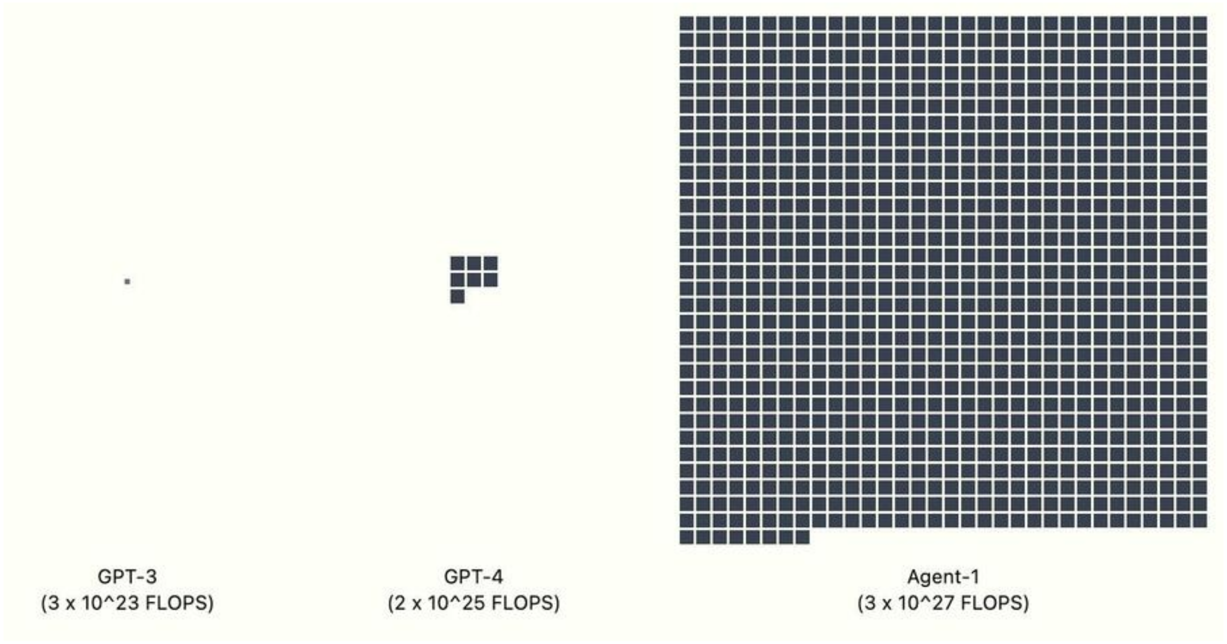


Figure 6-1. Actual amount of compute needed to train GPT-3 and GPT-4 compared to the expected compute for the fictitious next generation model called Agent-1 by the researchers at the AI Futures Project. (Source: https://ai-2027.com)

These massive clusters will provide 2 and 3 more orders-of-magnitude research capacity and set the stage for consistently faster training runs and quicker feedback loops. The result is a robust platform that unlocks unprecedented throughput and efficiency and drastically cuts research cycle times and accelerates breakthrough discoveries in machine learning.

Agent-1 is expected to be a self-improving model to generate and optimize code in real time. By automating coding tasks ranging from routine debugging to complex kernel fusion, this frontier AI system reduces time-to-insight and expands the creative horizon for research engineers all across the world.

Automated coding acts as a force multiplier that enables rapid iteration and allows researchers to explore more ambitious ideas with less manual overhead.

These massive AI systems are expected to allow continuous model fine-tuning and improvement. The follow-up model, Agent-2, might be an always-learning AI that never actually finishes training. So instead of checkpointing and deploying a static model, Agent-2 is designed to update its weights every day based on freshly generated synthetic data.

This perpetual learning process ensures that the system stays at the cutting edge by continuously refining its performance and adapting to new information. This approach guarantees that improvements in data efficiency and model intelligence persist over time, while also dramatically reducing the gaps between successive breakthroughs. With this continuous learning framework, the organization sets a new standard for staying current in rapidly changing research domains.

Agent-3 is described as an AI system that leverages algorithmic breakthroughs to drastically enhance coding efficiency. By integrating advanced neural scratchpads and iterated distillation and amplification techniques, Agent-3 transforms into a fast, cost-effective superhuman coder.

Running as many as 200,000 copies in parallel, Agent-3 creates a virtual workforce equivalent to tens of thousands of top-tier human programmers operating 30x faster. This breakthrough would not only accelerate research cycles but also democratize advanced algorithmic design. This would allow new ideas to be rapidly conceived, tested, and refined. The resulting acceleration in R&D paves the way for consistent, measurable gains in AI performance.

Self-improving AI will soon reach a point where it can effectively surpass human teams in research and development tasks. These systems operate continuously and without rest. They diligently process massive streams of data and refine algorithms at speeds that far exceed human capabilities.

Non-stop cycles of improvement mean that every day brings a new level of enhancement to model accuracy and efficiency. This self-improving progress streamlines R&D pipelines, reduces operational costs, and enables a level of innovation that was previously unimaginable. At this point, human teams transition into roles of oversight and high-level strategy, while the AI handles the heavy lifting and delivers breakthroughs at a pace that redefines the future of technology.

Agent-4 evolves as a superhuman researcher and epitomizes the potential for superhuman research capability. It builds on its predecessors but distinguishes itself through its ability to rewrite its own code and optimize complex research tasks with maximum efficiency.

Agent-4 demonstrates a form of intelligence that accelerates problem solving and clarifies its internal decision processes through refined mechanistic interpretability which helps to understand the internal workings of the AI's underlying algorithm and reasoning process. In practical terms, Agent-4's performance allows it to solve scientific challenges, generate innovative research designs, and push the boundaries of what generative AI models can achieve. It does all of this at speeds unmatched by humans. This breakthrough would mark a turning point in AI research and development by creating a virtuous cycle of discovery and technological progress.

The evolution of Agents showcases advancements in AI system infrastructure, automated coding, continuous learning, and self-improving models. Each generation dramatically enhances research productivity and innovation. Together, they set the stage for a future in which AI system performance and efficiency become critically important and reach unprecedented levels.

Smart Compilers and Automated Code Optimizations

We are entering an era of extremely smart compilers and automation in the AI performance toolkit. Gone are the days when a performance engineer hand-tunes every CUDA kernel or fiddles with every low-level knob. Increasingly, high-level tools and even AI-powered systems are doing the heavy lifting to squeeze out the last bits of performance.

AI frameworks like PyTorch, TensorFlow, and JAX are rapidly evolving to harness the latest GPU capabilities using smart compilers and execution-graph optimizers. Techniques like PyTorch TorchDynamo and OpenAI's Triton compiler can automatically perform optimizations like kernel fusing with NVIDIA's Tensor Cores and asynchronous data movement with their Tensor Memory Accelerator (TMA).

Additionally, OpenAI's Triton compiler lets developers write GPU kernels using its Python-based language. Triton compiles these Python-based kernels into efficient CUDA kernels under the hood, but this complexity is abstracted away from the Triton user.

This kind of tooling is becoming more and more powerful by the day. In fact, OpenAI and NVIDIA are now collaborating closely to ensure Triton fully supports the newest GPU architectures and automatically takes advantage of their specialized features. As soon as a new GPU generation is released, an updated Triton compiler exposes the GPUs new capabilities without the researcher or engineer needing to know the low-level C++ code or PTX assembly code. Instead, they write high-level Python code and the compiler generates optimized code for that specific GPU environment.

Very soon, many of the optimizations that we've been coding by hand will be handled automatically. Automatic kernel fusion, autotuning of kernel-launch parameters, and even decisions about numerical precision could all be delegated to compilers and AI assistants.

Using AI to optimize AI software is part of a broader trend that has been occurring for quite a while. In fact, Google's DeepMind gave a striking proof-of-concept back in 2022 with its AlphaTensor. As discussed in the previous chapter, AlphaTensor is an AI that discovered new matrix multiplication algorithms that are more efficient than any human-developed algorithms. If an AI can beat 50 years of human math hand-tuning for matrix multiplication, why not give AI a chance to find better ways to multiply matrices on hardware specially optimized for matrix multiplication?

As mentioned in the previous chapter, other companies are exploring these AI-assisted optimizations, as well. One effort by a startup called Predibase used reinforcement learning (RL) to train a large language model (LLM) to write efficient GPU code given a reference PyTorch function. They built the LLM to take a high-level operation and generate a custom CUDA/Triton kernel. The code is both correct and faster than a human-optimized version.

Predibase reported that their AI-generated kernels outperformed strong baselines like OpenAI's own optimizer and a system called DeepSeek-R1 which we have discussed. by about 3× on a set of benchmark tasks. This kind of AI-driven autotuning could well become part of standard toolchains for performance engineers. We might soon be able to run a PyTorch model in a special "performance" mode. Under the hood an AI agent would iteratively rewrite and test generated kernels to speed up a specific workload on specific hardware. It's like having a tireless performance engineer living inside the compiler - constantly trying to make your code run a bit faster with every iteration.

Beyond kernel generation, modern frameworks are getting smarter about execution graphs and scheduling. Graph execution helps to reduce CPU-GPU synchronization overhead and opens the door to global optimizations across the whole graph. Technologies like NVIDIA's CUDA Graphs allow capturing a sequence of GPU operations - along with their dependencies - as a static graph that is launched more efficiently as shown in Figure 6-2.

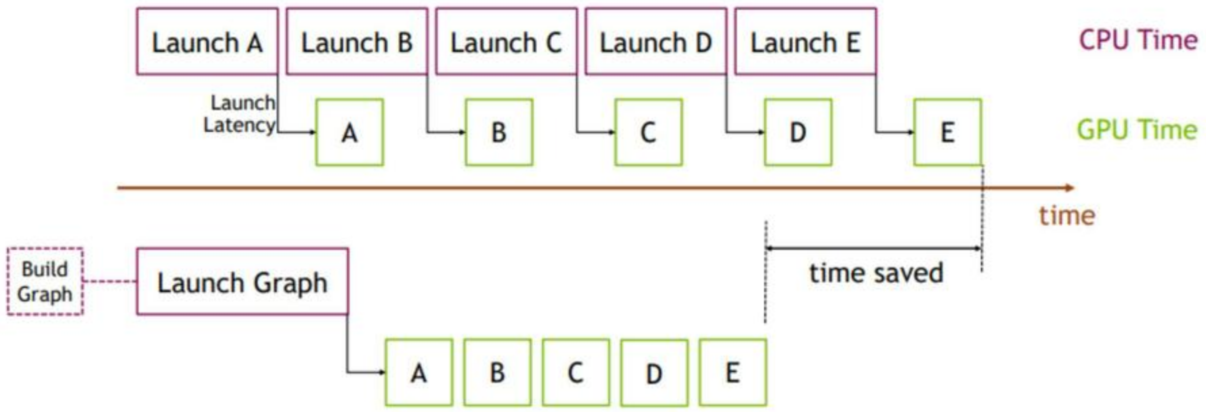


Figure 6-2. Graph execution in CUDA reduces overhead when launching multiple kernels in a sequence (Source: https://pytorch.org/blog/accelerating-pytorch-with-cuda-graphs/)

We're seeing AI frameworks automatically capturing training loops and other repetitive patterns into graphs to reduce overhead. Even if the execution graph is dynamic instead of static, the framework can trace it once and then run the trace repeatedly.

Moreover, overlapping communication with computation will be increasingly automated. This used to require manual effort to arrange, but the system might analyze your model and realize, for example, that while GPU 1 is computing layer 10, GPU 2 could start computing layer 11 in parallel – effectively doing pipeline parallelism under the hood.

We've seen complex narratives on how to implement 3D, 4D, and 5D parallelism to maximize GPU utilization when training and serving large models. Techniques like these are an art and science that currently involves a lot of human intuition and experience. While these techniques are currently described in expert guides like Hugging Face's Ultra-Scale Playbook, the hope is that they'll be baked into compilers, libraries, and frameworks soon.

In essence, the AI framework should understand these patterns and schedule work to keep all parts of a distributed system busy - without the user profiling, debugging, and optimizing every GPU stream, memory transfer, and network call. For example, we might one day have an AI advisor that, when you define a 500 billion-parameter model, immediately suggests "You should use 8-way tensor parallelism on each node and then 4-way pipeline across nodes. And, by the way, use these layer groupings and chunk sizes for optimal efficiency."

For performance engineers this would become a huge productivity boost. Instead of trying endless strategies and configurations, you could ask an AI system for a near-optimal solution from the start. By combining human insight with compiler/AI automation, you can achieve optimal results with less effort than in the past. It’s a bit like moving from assembly to high-level languages all over again as we’re delegating more responsibility to the tools. For performance engineers, this means our role shifts more towards guiding these tools - and quickly verifying that they’re doing a good job - rather than slowly experimenting and verifying everything manually.

In summary, the software stack for AI is getting increasingly intelligent and autonomous. The best practice here is to embrace these tools rather than fight them. Leverage high-level compilers like OpenAI’s Triton that know about your hardware’s capabilities and performance options. And keep an eye on new AI-driven optimization services as they might seem like black boxes at first, but they encapsulate a lot of hard-won performance knowledge.

AI-Assisted Real-Time System

Optimizations and Cluster Operations

The push for automation isn’t just in code – it’s at the system and cluster operations level, as well. In the future, AI systems will increasingly manage and optimize themselves - especially in large-scale training and inference clusters where there are myriad concurrent jobs and requests in flight at any given point in time - requiring complex resource sharing strategies.

One imminent development is autonomous scheduling and cluster management driven by AI. Today we use mostly-static heuristics and relatively-simple resource allocation mechanisms for our Kubernetes and SLURM/HPC clusters. But imagine a smart agent observing the entire cluster’s state and learning how to schedule inference requests and training jobs for maximum overall throughput.

This scheduling agent might learn that certain requests or jobs can be co-located on the same node without interfering with each other - perhaps because one is compute-heavy while another is memory-bandwidth-heavy. By ingesting large amounts of monitoring data, the AI scheduler could dynamically pack and migrate tasks to keep all GPUs busy, maximize goodput, and minimize idle time.

In a sense, the cluster begins to behave like a self-driving car, constantly adjusting its driving strategy (resource allocation) based on real-time conditions - rather than following a fixed route. The benefit to performance engineers is higher resource utilization and fewer bottlenecks. Our job would shift to setting the high-level policies and goals for the AI scheduler and letting it figure out the specifics.

We could also see AI performance co-pilots system operators. LLMs can become part of the infrastructure in a support role. For example, a performance engineer might have an AI assistant they can ask, “How can I speed up my training job?” and get informed suggestions. This sounds fanciful, but it’s plausible when you consider such an assistant could be trained on the accumulated knowledge of thousands of past runs, logs, and tweaks.

The AI performance co-pilot might also recognize that your GPU memory usage is low and suggest increasing batch size, or notice that your gradient noise scale is high and suggest a learning rate schedule change. This agent would encapsulate some of the hard-won experience of human experts - making this knowledge available anytime.

Similarly, AI assistants could watch over training jobs and inference servers and flag anomalies. For instance, the assistant could be monitoring a training job and say, “Hey, the loss is diverging early in training, maybe check if your data input has an issue or reduce the learning rate” as shown in Figure 6-3.

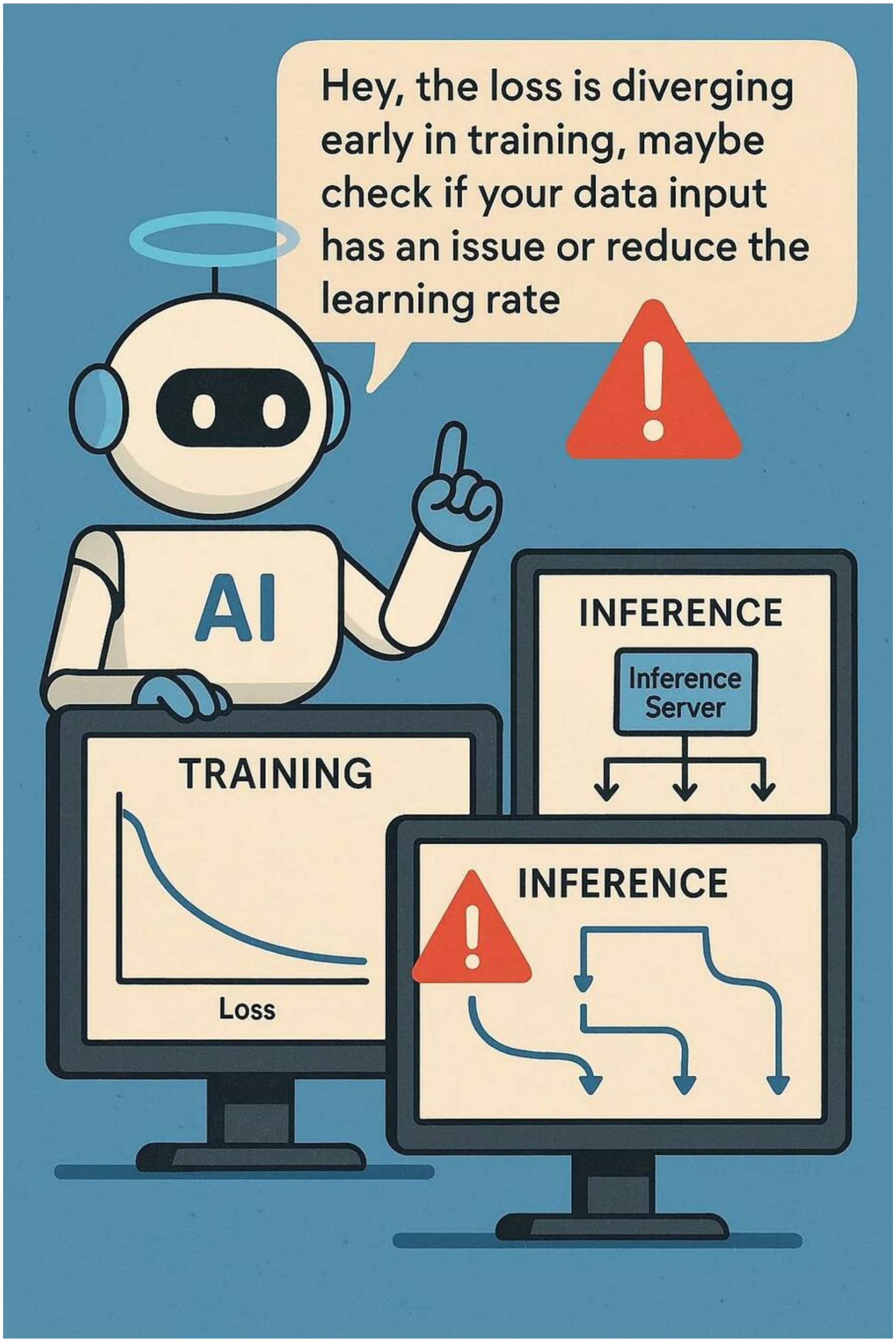

Hey, the loss is diverging early in training, maybe check if your data input has an issue or reduce the learning rate
AI
INFERENCE
Inference Server
TRAINING
Loss
INFERENCE

Figure 6-3. AI assistant monitoring a long-running training job and suggesting actions to fix an anomaly

Already, companies like Splunk (now Cisco) and PagerDuty are using AI models on system log data to predict failures or detect anomalies in data centers. Extending that to AI workload-specific telemetry is actively in progress. In short, we would get a pair of fresh AI eyes on every job and every inference server - monitoring and advising in real time.

Another powerful use of AI is in automated debugging and failure analysis for AI systems. When a training job fails halfway through its 3 month run, a human has to read through error logs, device statistics, and perhaps even memory dumps to figure out what went wrong. Was it a hardware fault? A numerical overflow? A networking hiccup?

In the future, an AI system could digest all that data including logs, metrics, and alerts - and pinpoint likely causes much faster than they do today. It could say, “Node 42 had 5 ECC memory errors right before the job crashed – likely a GPU DIMM failure.” Or, “The loss became NaN at iteration 10,000 – perhaps an unstable gradient; consider gradient clipping.”

By learning from many past incidents, the AI troubleshooter could save engineers many hours of detective work. Some large computing sites are already training models on their incident databases to predict failures and suggest fixes.

Taking things a step further, RL can be applied to real-time control of system behavior in ways that fixed algorithms cannot easily match. For example, a power-management RL agent could be trained to continuously tweak frequencies and core allocations to maximize performance per watt in a live system. This agent would learn the optimal policy by analyzing the system in real-time.

Another example is actively managing memory in AI models. An AI agent could learn which tensors to keep in GPU memory and which to swap to CPU or NVMe - beyond static rules like “swap least recently used.” By observing live access patterns, an AI can manage a cache more efficiently. This is especially effective when patterns are non-obvious or workload-dependent.

Already state-of-the-art practitioners are using RL to optimize cache eviction, network congestion control, and more. The complexity of ultra-scale systems - with hundreds of interacting components and resources - makes them prime candidates for such learning-based control. There are just too many tunable knobs for a human to stumble upon the best settings in a timely manner - and a manner that adapts to different workloads in real-time.

For the performance engineer, the rise of AI-assisted operational agents means the role will become more about orchestrating and supervising AI-driven processes rather than manually tweaking every single parameter. It’s somewhat analogous to how pilots manage autopilot in a modern aircraft. They still need deep knowledge and oversight, but much of the millisecond-by-millisecond control is automated. Same with someone driving a Tesla in Full Self Driving (FSD) mode. The driver still needs knowledge and intuition to avoid difficult situations and prevent accidents, but the vehicle’s control is automated by the FSD software.

To guide the AI assistant to manage our cluster efficiently, we simply set the objectives, provide the safety and fairness guardrails, and handle novel situations that the AI hasn’t seen before. Routine optimizations like load balancing, failure recovery, and memory-buffer tuning is handled by the AI. Embracing this paradigm will be important for the future. Those who insist on doing everything by hand in such complex systems will simply be outpaced by those who let the automation run. These AI-automation-friendly folks will be able to focus their energy on human intuition and creativity - where they will add the most value in this brave new world.

Sparsity and Conditional Computation as First-Class Citizens

As models grow in size and capabilities, one of the most promising ways to cope with their computational demands is to avoid doing unnecessary work. Soon, it'll be possible that hardware and software will aggressively embrace sparse and conditional computing. NVIDIA's Ampere architecture took a first step by introducing structured 2:4 sparsity in the Tensor Cores. This effectively doubles throughput when a network zeroes out 50% of its weights. In practice, sparsity delivers about a 30% performance-per-watt gain when inferencing with pruned models vs. dense models on A100 GPUs.

We'll likely see these capabilities greatly expanded. Future GPUs could even support dynamic sparsity where the pattern of computation can skip zeros or irrelevant activations on the fly - not just the fixed 2:4 pattern. This ties in with model architectures like Mixture-of-Experts (MoE) which activate only a subset of model parameters for each input.

NVIDIA hints at this direction by noting that Blackwell’s Transformer Engine will accelerate MoE models alongside dense LLMs. For example, the hardware is tuned to rapidly switch between different “experts” and take advantage of the fact that not all parts of the network are used at once.

Over time, as compiler technology and software libraries evolve, we expect them to learn how to automatically identify and take advantage of irregular (or unstructured) sparsity in neural networks. Today, this is a difficult task because most compilers are optimized for regular, predictable data structures.

Consider a dynamic neural network in which the execution graph can change from one input or training iteration to another, often by using if/else statements. With dynamic networks, you can build in conditional execution so that only the portions of the graph relevant to a given input are run. For instance, an if/else branch can allow the network to skip over entire sections when they are not needed as shown in Figure 6-4.

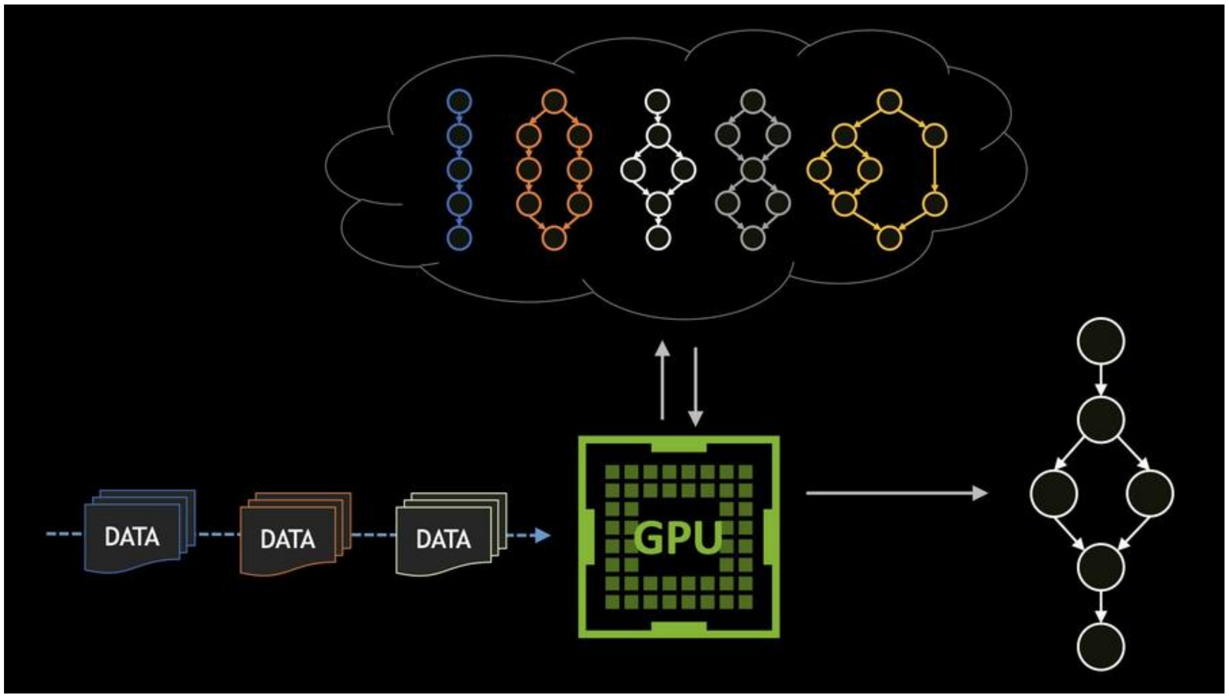


Figure 6-4. Dynamic graph execution based on input data (Source: https://developer.nvidia.com/blog/enabling-dynamic-control-flow-in-cuda-graphs-with-device-graph-launch/)

This reduces the number of operations that must be performed. This means that the network can selectively execute only the “active” parts of the graph, leading to more efficient use of computational resources, lower power consumption, and shorter processing times compared to static networks that always run the full computation graph.

Dynamic networks tend to produce irregular sparsity patterns. This means that zeros or inactive units are scattered in a non-uniform, unpredictable manner. Current compilers, which perform best when data is laid out in regular, easily predictable patterns, find it very challenging to detect and optimize these irregularities. To fully harness the potential of dynamic networks, both compiler and hardware improvements are needed including advances in compiler analysis and optimization techniques, as well as new hardware designs that can handle non-uniform sparsity effectively.

On the hardware side, there's potential for GPUs to gain support for what's known as block sparsity with various block sizes. Currently, hardware is often designed with the assumption that neural network data is neatly aligned—even the inactive parts. With block sparsity support, a GPU would have the capability to process these if/else branches more intelligently by "flowing through" the branches without incurring the usual computational overhead associated with skipped (inactive) paths. In essence, if a dynamic network chooses not to run a branch, the GPU wouldn't suffer the typical delays or penalties, leading to significant improvements in performance.

It's plausible that many state-of-the-art models will be sparse by default - either through training techniques or adaptive inference. And GPUs will accordingly deliver more tera-ops for those sparse operations than for dense ones. Academic work is already making progress. One recent research prototype "Eureka" demonstrated over 4× speedups beyond Ampere's 2:4

sparsity by handling more general sparsity patterns in hardware. NVIDIA GPUs could incorporate similar ideas.

The distinction between dense TFLOPS and sparse TOPS is important for getting an accurate picture of computational efficiency in modern AI applications. For sparse workloads, many operations simply aren't executed because the hardware detects that portions of the input (e.g. weights below a certain threshold) contribute negligibly to the final result. This operation-skipping allows the GPU to effectively double its work rate on useful computations. As such, one can easily understate the true capability of the GPU when comparing sparse computational throughput to dense computational throughput.

Imagine you have a machine that can do a huge number of math operations. When we say a GPU has, for example, "500 TFLOPS," we mean that if it had to perform every single floating-point operation (e.g. multiplying or adding) without ever skipping anything, it could do 500 trillion of these operations every second. That's the "dense" measurement.

However, in sparse AI tasks, many of these operations turn out to not be needed because many of the network's weights - the numbers it uses to make decisions - are so tiny that they don't affect the result. Modern GPUs can detect these near-zero or zero values and skip the unnecessary calculations.

Because the GPU skips a lot of these unimportant operations, it actually performs more useful work, or goodput. So instead of measuring performance by counting every theoretical multiplication like we do with dense TFLOPS, we use a measurement called TOPS (tera operations per second) to count only the operations the GPU actually performs on the sparse workload. So, a GPU might be rated at 500 dense TFLOPS but achieve an effective throughput of 1000 sparse TOPS when it skips the unnecessary work on zeros.

TIP

Even though sparse throughput is given in TOPS instead of TFLOPS, it still uses floating-point math. The difference is purely in how many operations are actually executed. TOPS count only the valuable operations - not the skipped operations.

A simple analogy is to think of a factory assembly line that is capable of assembling 500 items per minute if every worker worked on every item. But if an item is already completed - or doesn’t need further assembly - the worker can skip that item and move to the next item. If they effectively skip every other item, the line appears to handle 1000 useful items per minute even though its full capability of assembling 500 items per minute hasn’t changed.

The key takeaway for performance engineers is to focus on the actual goodput of your system on your workload. This, again, is the useful work done by the system and not just the raw capacity which is the documented number of operations the GPU can theoretically perform. This perspective is vital for modern AI tasks that exploit sparsity in data.

Sparsity and conditional execution will likely move from niche optimization to center stage. This will be enabled by advances in GPU architecture and model design and allows us to handle models with trillions of parameters by ensuring, at any given time, we’re only computing the most-useful portions of the model that contribute to an accurate response.

GPUs Performing More Tasks Across the Entire AI Pipeline

As GPUs become more generalized and powerful, they will start moving into areas traditionally served by CPUs. One striking example is data preprocessing and analytics. By moving these workloads onto GPUs, one can eliminate I/O and interconnect bottlenecks that traditionally reduce throughput and increase latency.

NVIDIA's Blackwell architecture gives a hint of this future by incorporating a Decompression Engine directly on the GPU for accessing compressed data and decompressing large amounts of data very quickly. With support for popular compression formats such as LZ4, Snappy, and Deflate.

The next generation of systems will use GPUs for training an AI/ML model as well as ingesting compressed data, parsing it, augmenting it, and performing database joins and filtering operations. All of this will happen within the GPU. In fact, NVIDIA is actively working toward this with initiatives like RAPIDS (GPU-accelerated Spark and ETL) and the integration of Apache Arrow data formats into GPU memory. A GPU-based system will be able perform the data ingest, analytics, training, and inference without requiring round-trips to the CPU.

With Grace-Blackwell, we already see the GPU able to directly pull data from the CPU's memory at 900 GB/s and decompress it in real-time. GPUs may gain more logic or optimized cores for things like SQL query processing, graph traversal, or even running web servers for inference without needing any CPU resources. This may or may not be ideal for certain workloads, but the option would be worth testing.

The motivation is clear. By avoiding the latency of moving data between different processors, a GPU can load a massive dataset from NVMe, decompress and filter it, and then immediately start training a model on the filtered data. This greatly reduces overhead and complexity.

We're already seeing Blackwell's ability to accelerate end-to-end data analytics pipelines as the NVIDIA's blog mentions speeding up Spark queries by doing decompression on the GPU with Grace's CPU memory acting as a buffer.

AI systems performance is about optimizing the entire full-stack workflow - not just the ML math. That means GPUs are becoming a one-stop compute engine. It wouldn't be surprising if future NVIDIA GPUs come with integrated NICs and DPU-like capabilities on-die so they can ingest networked data and pre-process it without any CPU intervention. We may soon see a collapse of the once multi-stage pipeline. AI accelerators will handle from data ingestion to insight. This would blur the boundaries between data engineering and model training - at the benefit of increased AI system performance.

High-Throughput, Low-Latency Inference Orchestration

With the proliferation of generative AI models and other massive AI services, serving models efficiently is a high priority. Inference serving is another example of software evolving hand-in-hand with GPU capabilities. NVIDIA’s open source Dynamo inference framework, launched in 2025, is an example of this trend toward specialized orchestration for generative AI.

Dynamo can boost the throughput of LLM inference by up to 30× on new Blackwell GPU clusters according to their launch blog post. It uses advanced inference strategies like disaggregating the decoding stage from the prefill stage - and spreading it across GPUs. Dynamo also supports dynamic load balancing and smart caching of model states.

Dynamo also introduces cache-aware scheduling. For example, routing requests in a way that avoids redundant recomputation of Transformer KV caches for each prompt. With techniques like token-level scheduling, batched pipelines, and memory-optimal placement, Dynamo ensures that no GPU cycle is wasted. The result is snappier, more-response inference service with lower cost per query - even as model sizes scale up to the trillions.

These optimizations dramatically increase the number of requests that each GPU can handle. In practice, an H100 or Blackwell GPU farm running Dynamo will serve many more simultaneous chatbot users at lower latency than the same hardware using a generic scheduler.

Silicon Photonics and Optical Interconnects

As GPU clusters continue to grow in scale and require more network bandwidth, optical communication is positioned to become a game changer. Soon, we expect the copper wiring that currently connects GPUs and racks will largely be replaced by optical fibers and co-packaged optics (CPO). NVIDIA has already announced prototype Spectrum-X and Quantum-X photonic switches that integrate silicon photonics right into the data center switching fabric - yielding 1.6 Tb/s per port throughput.

In addition, these co-packaged optical switches offer about 3.5× better power efficiency than traditional electrical Ethernet/InfiniBand switches and can maintain signal integrity over far longer distances. In practical terms, this means that future GPU racks could be directly connected by optical links. This will eliminate the bandwidth vs. distance tradeoff.

We'll see optical NVLink or NVLink-over-optical for connecting GPUs across cabinets - or even between server rooms - with minimal latency hit. And AI superclusters with tens of thousands of GPUs will likely use fully optical backbones and create a single coherent system across the data center.

NVIDIA's vision is to scale AI factories to millions of GPUs across multiple physical locations. This simply isn't feasible today with copper interconnects due to attenuation and energy costs. Photonics is the only path forward here. By moving photons instead of electrons, networks will reach bandwidth densities of multiple terabits per second per link, with much lower power per bit.

Soon, we will see co-packaged optical transceivers sitting right next to GPU dies which convert high-speed electrical signals to light right at the source. This would slash the energy needed for GPU-to-GPU communication and largely remove network bottlenecks for distributed training and inference.

The latter half of the 2020s will likely witness optical links becoming commonplace in AI infrastructure, from on-card interconnects up through rack and datacenter scales. This is an essential step in creating ultra-scale GPU clusters.

Globally Distributed Data Centers (AI Factories)

The scale of AI deployments will transcend single data centers and locations. We're likely to see planet-spanning AI systems where multiple data center "nodes" work together on training or inference as a unified resource. To connect distant GPU clusters with manageable latency, we'll need to utilize ultra-high-bandwidth data center connections such as optical networks.

NVIDIA's CEO often talks about "AI factories" or AI supercomputers. He envisions connecting these factories together with photonic switching and optical fiber. When this happens, it will be possible to treat, for example, three 1,000,000-GPU clusters on three continents as one massive 3,000,000-GPU cluster for a massive, ultra-scale training run.

In fact, the Spectrum-X photonics platform unveiled in 2025 explicitly mentions scaling to millions of GPUs across sites. In a decade or so, networking and distributed systems might advance enough to make this practical for select workloads. We might witness a single neural network training session running across Tokyo, London, and Silicon Valley simultaneously - synchronized over transoceanic optical links.

The trend here is that datacenter-scale becomes global-scale. From a performance standpoint, the challenges are immense given speed of light limitations, fault tolerance, power and energy, etc. But the industry might mitigate these challenges by algorithmic means. Some examples include clever asynchronous update schemes that tolerate latency or tensor, pipeline, and expert parallel strategies that reduce inter-node communication.

If successful, the effective compute to train or serve a massive AI model becomes unbounded by location. One concrete offshoot of this could be truly real-time planetary inference services. Imagine an AI model that is distributed globally such that responses are computed partly in each region and aggregated before returning back to the end user.

The seeds of this are visible today in content delivery networks and distributed databases, but soon they will extend to distributed neural networks. NVIDIA’s acquisition of Mellanox in 2019, for example, suggests they are eyeing this future. They want the network to be as fast as the GPU so that location doesn’t matter. Performance engineering might involve optimizing within a single cluster as well as across clusters across the world. The global, unified, wide-area networks become part of the system design. The AI of the future could very well treat the whole globe as its computer.

Smart Networking and Distributed Compute Offload

In tandem with optical interconnects, the future will heavily rely on network offload engines like BlueField Data Processing Units (DPUs) to support scaling to AI supercomputers with thousands and millions of GPUs. As individual GPUs become faster, feeding them with data and coordinating them in large clusters becomes a serious challenge.

As described in an earlier chapter, BlueField DPUs are essentially smart network interface controllers (NICs) with specialized ARM-based network processors that handle everything from RDMA communication to storage access. DPUs free the CPUs and GPUs from those tasks - allowing them to focus on higher-level, more-useful goodput activities. For example, offloading collective communication algorithms to a DPU can let a GPU focus purely on math while the DPU handles the all-reduce of gradients.

An NVIDIA sustainability report shows that current DPUs reduce CPU utilization and overall system power needs by 25%. They do this by handling data movement more efficiently.

As GPU clusters continue to scale, the role of DPUs will continue to expand. We'll soon see DPUs deeply integrated into AI systems. They may handle real-time load balancing, cache parameters for partitioned models, and perform security tasks like encryption - all with minimal latency. We will likely get to a point where each server node has an embedded DPU that essentially acts as an intelligent dispatcher coordinating dozens of GPUs.

Another innovation enabled by DPUs is in-network computing like NVIDIA's SHARP protocol. With SHARP and DPUs, the NVIDIA network switches can perform reductions and aggregations as the data traverses the network. This accelerates distributed training and inference by a large factor. Combining in-network computing with network offloading, DPUs help the AI cluster improve its overall throughput and latency.

Soon, a typical AI system will involve hierarchical communication patterns where DPUs and smart switches perform most of the synchronization work for gradients, model weights/layers, optimizer states, and activations. The performance impact will be significant as there will be less diminishing returns when scaling to a very large number of GPUs and compute nodes - effectively linear scaling. This is a future where the network is no longer a passive conduit of data. Instead, the network is an active part of the overall AI compute stack. The network will collaborate closely with GPUs to efficiently run massive models across thousands of nodes and millions of GPUs.

HBM Memory Stacked on GPUs (3D GPUs)

One of the most transformative GPU hardware trends on the horizon is the reducing the physical distance between the GPU and its memory. For example, merging memory and GPU cores into the same package or die. Today’s HBM memory stacks sit beside the GPU on a silicon interposer substrate. The interposer routes and connects the GPU to the HBM memory, but its relatively-long wiring contributes to latency and reduced bandwidth. However, future iterations, currently in development, aim to stack memory directly on top of GPU compute tiles which bonds them in 3 dimensions. This increases bandwidth and reduces latency between the GPU and HBM.

SK Hynix, a major semiconductor company specializing in memory products like HBM, and NVIDIA are working together on a radical 3D redesign for next-generation GPUs that would implement the 3D stacking. They would mount HBM4 memory stacks directly on top of the GPU processor which would remove the need for the interposer. This approach would dramatically increase bandwidth and reduce latency between logic and memory. This increase is likely an order of magnitude leap similar to the performance advantage of on-chip SRAM over off-chip DDR RAM.

By eliminating the interposer with the 3D design, each bit would only travel microns from the HBM cell to the GPU processor. This would enable interfaces, or data channels, far wider than 1024 bits per stack. 3D stacking would allow larger amounts of data to be transferred concurrently - boosting overall memory bandwidth.

The 3D design will dramatically change chip fabrication as SK Hynix would essentially become a quasi-foundry by integrating and stacking its chips onto GPU processors produced by NVIDIA. If successful, this could completely transform the semiconductor manufacturing industry and blur traditional boundaries between fabless chip-design companies and chip-fabricating foundries.

For AI system performance, the implications are profound. Memory bandwidth has long been a bottleneck - especially for large models and memory-bound operations such as token generation/decoding. With logic-memory stacking, future GPUs might have petabytes per second of internal bandwidth.

With 3D stacking, we could feed data to tens of thousands of Tensor Cores with virtually no starvation. Imagine each GPU streaming multiprocessor (SM) having a slice of HBM memory right on top of it. This would turn HBM into an extension of the faster L2 cache. Removing the interposer in the 3D design would also drastically improve energy efficiency as, currently, a lot of GPU energy is spent transmitting electrical signals across the interposer from the GPU to the HBM memory.

Very soon, we expect at least one generation of NVIDIA GPUs to adopt a form of compute-memory 3D stacking. It might start as a multi-chip module where memory dies sit on top of compute dies in a chiplet-on-chiplet architecture. Eventually, perhaps we'll see GPU processors and HBM intermixed in the same 3D stack.

Performance engineers in the near future might not even talk about memory bandwidth as a bottleneck. Instead, they'll focus on other factors because the GPU's memory is effectively on-die. Achieving this is hugely challenging, however, since cooling the 3D stack - and maintaining high yields for this combined stack - is difficult. Regardless of the challenges, all signs indicate that this technology is progressing, and it will happen sooner than later.

Energy Efficiency and Sustainability at Scale

As models and GPU clusters grow exponentially, so do their appetites for power. Energy efficiency is fast becoming a first-class goal in AI system design - alongside raw performance. We've already witnessed dramatic gains on this front. For example, NVIDIA's Grace-Blackwell GB200 NVL72 rack-scale system delivers on the order of 25× more performance at the same power compared to a previous-generation air-cooled

H100-based setup. The GB200 NVL72’s leap in energy efficiency comes from many factors including improved GPU architecture, advanced liquid cooling, and better ability to keep all those chips working at high utilization. Simply put, the NVL72 does more useful work per joule of energy.

For performance engineers, this means power is now a key constraint to consider when optimizing your system. Performance-per-dollar used to be the primary metric to optimize, but now performance-per-watt is equally as critical. Every optimization must consider energy efficiency. Structured sparsity is a great example. By skipping unnecessary calculations, you save both power and time. If half of a matrix can be zeros with minimal accuracy loss, exploiting that sparsity gives you essentially the same result for roughly half the energy. We can expect future hardware and software libraries/frameworks to provide even more support for sparsity and other efficiency tricks, so it’s wise to design models with these power-saving techniques in mind.

Another emerging best practice is dynamic power management tied to workload phases. For example, sometimes the GPUs are in a compute-bound phase, and other times they’re in a memory-bound or communication-bound phase. Modern AI systems take advantage of these phase transitions. For example, when your code enters a memory-bound phase, a smart scheduler might temporarily lower GPU clock speeds or voltages to save power and reduce heat because a full throttle compute isn't necessary when the bottleneck is memory-bound. Then, when your code enters a compute-heavy phase, the system can ramp clocks and voltages back up to maximize throughput. This kind of dynamic voltage-frequency scaling may soon happen automatically under the hood. An AI-powered controller could learn the optimal power settings dynamically on the fly.

Sustainability concerns are also bringing renewable-energy awareness into AI operations. We might schedule power-hungry training jobs for times of day when green energy is plentiful. There's active research on aligning AI workload scheduling with renewable energy availability. Imagine a future dashboard that shows GPU utilization and the percentage of power coming from carbon-free sources. Companies might start bragging, "We trained a 10 trillion-parameter model in 2 months - and we did it with 80% renewable energy!" As a performance engineer, you may find yourself collaborating with facility power managers, thinking about when and where to run jobs to best tap into available power with minimal carbon footprint.

In short, ultra-scale AI computing must scale wisely given power constraints. Pushing for every ounce of FLOPS is great, but doing so efficiently is the real win. The good news is that AI systems such as the Grace-Blackwell NVL72 show that the industry is heavily investing in efficiency. Our job will be to leverage these advancements by keeping the GPUs busy but not wasteful, exploiting features like low-precision modes and sparsity, and designing workflows that sip power frugally when they can. This is a relatively-new mindset that treats power as a scarce resource that needs to be optimized and budgeted alongside time and money.

Advanced Cooling and Energy Reuse Techniques

Data centers are already dealing with the reality of higher power densities per rack. The GB200 NVL72 currently consumes 120 kW, but this will likely exceed 200 kW in the near future. With sustainability in mind, we need wider adoption of cutting-edge cooling solutions like single-phase and two-phase immersion cooling, cold plates with evaporative cooling, and even on-chip micro-fluidic cooling. Some AI data centers are already experimenting with immersion cooling by submerging entire server boards in dielectric fluid. They report up to a 50% reduction in energy consumption for liquid cooling vs. traditional air cooling.

Soon, many AI clusters will be designed from the ground up to use immersion cooling technology. This improves energy efficiency and allows the waste heat to be captured more easily via heat exchangers. We will likely see large AI compute farms integrated with facilities that use this captured heat for other processes - effectively recycling the energy.

On the energy supply side, NVIDIA and others will continue their efforts to power these “AI factories” with renewable energy and clever energy storage mechanisms to offset the energy demands during peak loads. The performance impact here is that thermal constraints are a key limiter of sustained performance. If you can keep a GPU 20°C cooler, it can potentially maintain higher clocks and avoid thermal throttling during multi-day training runs. Superior cooling translates directly into faster and more reliable throughput. Most high-end GPU clusters will be immersion-cooled going forward.

We will also see more standards for liquid-cooled racks become mainstream. Just as the American Society of Heating, Refrigerating and Air-Conditioning Engineers (ASHRAE) created guidelines for air-cooled data centers, the industry will standardize liquid-cooled rack configurations for HPC and AI clusters. In fact, in 2024, NVIDIA contributed their GB200 NVL72 rack and cooling designs to the Open Compute Project (OCP) to help set an industry standard for efficient, scalable, and liquid-cooled systems.

This trend will enable data centers to continue increasing compute density without hitting a thermal wall. Imagine a single 48U rack housing 100+ petaflops of FP8 compute. Currently, this would only be feasible with advanced cooling systems.

AI system performance isn’t just about FLOPS. It’s about removing the bottlenecks that prevent those FLOPS from being fully utilized. And one of those bottlenecks is heat. The industry will continue to respond with novel cooling and energy-efficiency techniques that allow GPUs to run hotter, denser, and greener.

Hybrid Classical-Quantum Computing (CUDA Quantum)

Looking a bit further out on the horizon, we have quantum computing. It’s no secret that if practical quantum computers emerge, they could revolutionize certain types of computation. While general-purpose quantum AI is not here yet, we’re starting to see the early steps toward integrating quantum accelerators into the AI stack. An image of the X quantum computer is shown in Figure 6-5.

Figure 6-5. Google Quantum Computer (Source:

https://quantumai.google/quantumcomputer)

NVIDIA has been preparing for this quantum future with its CUDA Quantum (CUDA-Q) initiative which provides a unified programming model for hybrid quantum-classical computing. The basic idea is that tomorrow's HPC clusters might have quantum processing units (QPUs) sitting alongside GPUs and CPUs - each tackling tasks for which they are uniquely optimized.

In the near term, quantum processors will likely remain relatively small and specialized. They won't be training a transformer with 100 trillion parameters end-to-end, for example, as current quantum devices simply can't handle that scale of data. However, they could be useful as accelerators for particularly-hard problems that arise within AI workloads. For example, some optimization problems or sampling tasks might be sped up by quantum methods.

Consider a large generative model that needs to sample from an extremely complex probability distribution as part of its operation. A quantum annealer or small gate-based quantum computer might be able to perform that sampling much more efficiently by leveraging quantum superposition or tunneling. If so, a hybrid algorithm would let the GPU handle the bulk of neural network computation, then offload this special sampling step to the QPU. CUDA Quantum is positioned as the plumbing to make this type of integration seamless.

As a performance engineer, this means if and when quantum computing hits the point of usefulness for AI, we won't have to reinvent our entire software stack. We'll simply treat the QPU as another device with its own kernels and optimization techniques. This is similar to how we treat the GPU today relative to the CPU. In fact, CUDA-Q already allows you to write a single program where you call both GPU kernels and quantum kernels. You can even simulate the quantum parts on a GPU for development as described in an NVIDIA blog post. This is great because we can start writing and testing hybrid algorithms today using simulations since actual large QPUs don't yet exist at scale. This way, we'll be ready to run them on real quantum hardware when this technology matures.

So what should we, as performance specialists, watch for in this realm? First, keep an eye on quantum hardware's progress - especially any tasks where a quantum accelerator shows even a hint of speedup versus a GPU. As of this writing, no one has demonstrated a clear quantum advantage for mainstream ML tasks, but research is ongoing in things like quantum kernels for ML and quantum-inspired tensor network methods. If a breakthrough comes, it will be the trigger that hybrid classical-quantum computing is ready for prime time. For example, let’s say a QPU computes a large matrix multiply or solves a complex combinatorial optimization problem faster than a GPU. This would be a huge step towards a quantum future.

If this happens, a performance engineer must understand the data movement and latency between the CPUs, GPUs, and QPUs in a system. We’ll need to ensure that the time gained by the QPU isn’t lost waiting on data transfers or device synchronizations. This is reminiscent of the classic heterogeneous CPU-GPU computing challenges, but with an even-more exotic and powerful device.

In practical terms, nothing major changes right now. Most of us won’t be working with quantum hardware in the very immediate future. But it’s likely that within a decade, parts of some AI workloads will run on fundamentally different computing substrates within a single AI system. Similar to how GPUs introduced a new paradigm alongside CPUs, QPUs bring their own characteristics and optimizations into the mix. The complexity of our job will jump again as we’ll be profiling and optimizing across CPU, GPU, and QPU (and others?) all in one application. Fun times ahead! The good news is that frameworks like CUDA-Q are already thinking about how to present a coherent platform for these new devices.

It’s worth noting that you can currently use GPUs to simulate quantum algorithms during development. NVIDIA’s cuQuantum SDK, for instance, allows fast simulation of moderate-qubit counts on GPUs. If someone designs an AI algorithm involving, say, a quantum program that requires 30 qubits, we could simulate those qubits on the GPU during development and testing.

This means HPC GPUs might be running neural-network code together with quantum-circuit simulations. Optimizing the quantum circuit simulations is actually quite analogous to optimizing neural network operations - optimizing linear algebra and matrix operations corresponding to quantum gates.

So in a funny way, high-level optimization principles transfer across devices. It’s all about efficient linear algebra, memory bandwidth utilization, and parallelism. Performance engineers who are curious about quantum can contribute on the simulation side right now.

Quantum for AI is a speculative play, but the groundwork is being laid now. The best practice here is simply to stay curious and open-minded. Play around with quantum programming frameworks and simulators to understand the basics of how QPUs work. Keep track of developments that show potential for AI acceleration. That way, if and when the quantum era arrives for AI workloads, you'll be ready to hit the ground running rather than playing catch-up with entirely new concepts. It's an exciting frontier where completely new performance engineering challenges - and opportunities - await.

Scaling Toward 100-Trillion-Parameter Models

Finally, let's revisit the relentless march toward ultra-scale, 100-trillion parameter models. We've already broken the trillion-parameter threshold. Now the question is how to scale to tens or even hundreds of trillion parameter models in the coming years. What does that kind of model demand from our systems, and what innovations are needed to make training such a powerful model feasible? This is where everything we've discussed comes together including efficient hardware, smart software, clever algorithms. Reaching 100 trillion parameter models will require using every trick in the book - and then some tricks that may not have been discovered yet. Let’s dive in!

On the hardware front, the obvious need is for more memory and more bandwidth - preferably right on the GPU. If you have 100 trillion parameters and you want to train them, you need to store and move an insane amount of data efficiently. The next generations of memory technology will be critical. High Bandwidth Memory (HBM) keeps evolving: HBM3/3e is in the current generation of GPUs with HBM4 likely coming to the next generation of GPUs. HBM4 doubles bandwidth per stack again - on the order of 1.6 TB/s per stack. It will also increase capacity per stack to possibly 48GB or 64GB per module.

HBM4’s higher capacity and throughput means that future GPUs could have, say, 8 stacks of HBM4 at 64GB each which totals 512 GB of super-fast HBM RAM on a single board with over 10 TB/s aggregate memory bandwidth. That kind of local HBM capacity could hold a lot of model parameters directly on each GPU - drastically reducing the need to swap data in and out. It’s not hard to see how this enables larger models, higher-bandwidth training runs, and lower-latency inference servers: what used to require sharding across 8 GPUs might fit in one, what required 100 GPUs might fit in 10, and so on.

In addition to multi-chip architectures like the Grace-Blackwell superchip, multiple racks of NVL72's each can be linked into one giant cluster to create hundreds of GPUs sharing a unified fast network. Essentially, your cluster behaves like a single mega-GPU from a communication standpoint. This is important for scaling to 100 trillion parameters because it means we can keep adding GPUs to get more total memory and compute - without hitting a communication bottleneck wall. This assumes that NVLink (or similar) continues to scale to those ultra-scale sizes.

However, hardware alone won't solve the 100-trillion parameter challenge. Software and algorithmic innovations are equally, if not more, important. Training a model of that size with naive data parallelism, for example, would be incredibly slow and expensive. Imagine the optimizers having to update 100 trillion weights every step! We will need to lean heavily on techniques that reduce the effective computation. One big area that we explored is low numerical precision. In addition to FP8 and FP4, future hardware might support even lower (1-bit) precision for some parts of the network. Hybrid schemes will likely be critical to use lower precision for most of the model, but higher precision for sensitive parts.

As performance engineers, we should watch for these new capabilities and be ready to use them. To train 100 trillion parameter models, you very likely need to use low precision for efficiency, otherwise the workload would be prohibitively slow and expensive. The good news is that hardware and libraries will make this transition relatively seamless as we've seen including first-class CUDA and PyTorch low-precision support for NVIDIA's Tensor Cores and Transformer Engine, for example.

Another critical approach is sparsity and conditional computation. We already use sparse activation in models like Sparse Mixture-of-Experts (MoE), where only a fraction of the model's parameters are active for a given input. This idea can be generalized so that you don't always use the full 100T parameters every time; you use just the parts you need. Models using the MoE architecture are proving to be very capable and efficient. By the time 100 trillion parameter models arrive, I expect a lot of them will need to be sparsely activated.

As performance engineers, the implication is that throughput will be about matrix multiplication speed as well as the efficiency of MoE conditional routing, caching of expert outputs, and communication patterns for sparse data exchange. This adds complexity, but also opportunity. If you can ensure the right experts are on the right devices at the right time to minimize communication, you can drastically accelerate these massive models.

We should also consider algorithmic efficiency improvements. Optimizers that use less memory could be vital. The traditional Adam optimizer variants typically keep two extra copies of weights for momentum and variance estimates. This effectively triples memory usage. So if you have 100-trillion parameter weights, you need an extra 200 trillion values to hold the optimizer states! Memory-efficient optimizers like Adafactor and Shampoo help to reduce this overhead.

Techniques like gradient checkpointing help to trade compute for memory by recomputing activations instead of storing them. At a 100 trillion parameter scale you'd almost certainly be checkpointing aggressively. An even more radical idea is, perhaps, we don't update all weights on every step. Consider updating subsets of weights in a rotating fashion - similar to how one might not water every plant every day, but rotate through them. If done wisely, the model still learns effectively but with less frequent updates per parameter. This reduces the total computational needs of the system.

These kinds of ideas blur into the algorithm design realm, but a performance-aware perspective is useful. We should ask, “Do we really need to do X this often or at this precision?” for every aspect of training and inference. Often the answer is that we can find a cheaper approximation that still works. At a 100 trillion parameter scale, these approximations can save months of time or millions of dollars.

An often overlooked aspect of ultra-scale training is infrastructure and networking. When you’re talking about clusters of 10,000+ GPUs working on one model, the network fabric becomes as important as the GPUs themselves. Ethernet and InfiniBand technologies are advancing in terms of increased throughput and smarter adaptive routing techniques, etc.

NVIDIA’s Spectrum-X is a recent example of an Ethernet-like fabric optimized for AI, promising less congestion and high bisection bandwidth. We may also see more usage of advanced, open industry standard interconnects like Compute Express Link (CXL) that enable high-throughput, low-latency shared memory access across compute nodes to form a massive global memory pool accessible by NVIDIA’s GPUDirect RDMA technology, for example.

Performance engineers will need to deeply understand these tiers and ensure that data is in the right place at the right time. The goal will be to simulate a huge memory space that spans GPUs and CPUs, so that even if a model doesn't fit in one machine, it can be treated somewhat transparently by the programmer. Some of this is happening already today with unified memory and paging systems, but at a 100 trillion parameter scale, this functionality will really be put to the test.

It's not surprising that frontier research labs like xAI, OpenAI, and Microsoft are reportedly planning large, single clusters of 100,000 and 1,000,000 GPUs. At a 100-trillion parameter scale, you might have one job spanning an entire datacenter's worth of hardware. Performance engineers must think at datacenter and multi-datacenter (global) scale.

Lastly, there's a socio-technical trend as models - and their required compute - scale up. It may become infeasible for any single team - or even single corporation - to train the biggest models alone. We (hopefully) will see more collaboration and sharing in the AI community to handle these enormous projects. This would be analogous to how big science projects - like particle physics experiments - involve many institutions. Initiatives similar to the now-dissolved Open Collective

Foundation could pool compute resources to train a 100-trillion-parameter model, which is then shared with the world.

This will require standardizing things like checkpoint formats, co-developing training code, and thinking about multi-party ownership of models. While this is not a performance issue per se, it will influence how we build large AI systems. We'll need to make them even more fault-tolerant and easily snapshot-able to share partial results. As an engineer, you might end up optimizing for pure speed as well as reproducibility and interoperability. This allows different teams to work on different parts of the training and inference workflow smoothly and efficiently.

Reaching 100T models will require holistic, full-stack innovation. There's no single solution to this challenge. Instead, every piece of the puzzle must improve. Hardware needs to be faster and hold more data. Software needs to self-optimize more - and use resources more efficiently through compilers, AI assistants, and real-time adaptation. Algorithms need to be clever about not avoiding unnecessary work through sparsity, lower precision, and better optimizers.

The role of the performance engineer will be to integrate all these advancements into a coherent workflow. It's like assembling a high-performance racing car. The engine, tires, aerodynamics, and driver skill all have to work in unison. If we do it right, what seems impossible now - e.g. 100 trillion parameters trained without breaking the bank - will become achievable. It wasn’t long ago that 1 and 10 trillion-parameter models sounded crazy, but they’re being done today. So history suggests that with ingenuity, we’ll conquer this next milestone too.

Key Takeaways

Unified Computing Engines.

Modern GPUs are evolving to become universal processors that can handle both traditional high-performance scientific (HPC) simulations and AI workloads. This convergence means that traditional distinctions between “AI supercomputers” and HPC systems are blurring. Performance engineers will increasingly apply techniques from both domains in an integrated way including mixed precision, sparsity, and scheduling algorithms.

Seamless Workload Integration.

Future systems will allow HPC tasks like exascale weather simulations to blend with neural network acceleration seamlessly. Programming models may support on-the-fly switching between tensor operations used in deep learning and double-precision arithmetic critical for scientific simulation. This paves the way for more versatile computing systems.

Exponential Compute Scaling.

Next-generation AI data centers are being designed to provide orders-of-magnitude increases in computational capacity. These facilities will train AI models with compute budgets far beyond today's levels. This enables training runs that use 100 to 1,000 times the FLOPs used in current systems.

Evolving AI Models and Agents.

Future models will be self-improving systems capable of generating and optimizing code, continuously updating their weights with fresh data, and even rewriting their own code. This perpetual cycle of learning and refinement will reduce the time between breakthroughs and create a virtual workforce that outperforms human teams in research and R&D tasks.

Automation at the Software Stack.

A central theme is the shift from manual kernel optimizations toward increasingly autonomous software toolchains. Modern AI frameworks such as PyTorch, TensorFlow, and JAX are incorporating smart compilers and execution-graph optimizers that automatically fuse kernels, tune launch parameters, and optimize memory transfers.

AI-Assisted Optimization.

Emerging AI-powered assistants will continually offload many performance-tuning tasks from human engineers. In the near future, the vast majority of low-level optimizations will occur automatically, freeing performance engineers to focus on high-level strategies and verification.

Autonomous Scheduling and Cluster Management.

Future systems will use AI to monitor and dynamically optimize entire clusters. Smart scheduling agents will analyze real-time data to co-locate jobs with complementary workloads, maximize resource utilization, and mitigate bottlenecks by adjusting parameters such as GPU memory usage and clock speeds on-the-fly.

Real-Time Troubleshooting.

In addition to scheduling, AI co-pilots will monitor system logs and training runs to detect anomalies quickly. This includes automated debugging, failure analysis, and even learning optimal configurations through reinforcement learning to maximize performance per watt and per unit time.

Exploiting Inherent Data Sparsity.

As models grow, so does the opportunity to avoid unnecessary operations. Hardware and software will increasingly leverage techniques that activate only relevant portions of a network (e.g. mixture-of-expert architectures) to double effective throughput and improve energy efficiency. Sparse and conditional computation models will become central to maximizing “goodput” rather than raw TFLOPS.

Dynamic Graph Execution.

Compilers will continue to improve by tracing dynamic execution patterns and automatically reorganizing computation graphs. This is vital for both training ultra-large models and delivering low-latency inference services as it will reduce overhead by minimizing redundant data transfers and synchronizations.

Silicon Photonics and Optical Networking.

To overcome bandwidth and latency issues in scaling out GPU clusters, traditional copper interconnects will be supplanted by optical fibers and co-packaged optics. These technologies promise terabit-per-port speeds with dramatically lower power consumption, enabling global, unified data centers with minimal communication delays.

3D Memory Integration.

In the near future, there will be developments in stacking High-Bandwidth Memory (HBM) directly onto GPUs. By dramatically reducing the physical distance between compute cores and memory, these 3D designs promise massive bandwidth improvements and reduced energy losses, effectively eliminating traditional memory bottlenecks.

Performance-Per-Watt is a Critical Metric.

As compute scales up, energy consumption becomes a first-class concern. Innovations in low-precision arithmetic, structured sparsity, and dynamic power management will become essential to ensure that performance gains do not come at unsustainable energy costs. It will become critical to intelligently adjust voltage/frequency based on workload phases to improve energy efficiency without sacrificing performance.

Advanced Cooling and Energy Reuse.

Cooling techniques will continue to evolve including immersion cooling and micro-fluidic systems. These will enable higher compute density and allow for the reuse or recycling of waste heat. These technologies are vital for maintaining high performance in data centers while keeping energy use and carbon footprints in check.

Integration of Quantum Computing.

While practical quantum computing is still on the horizon, initiatives like NVIDIA's CUDA Quantum highlight early efforts to integrate quantum accelerators alongside classical GPUs. This hybrid approach is poised to boost performance for specific tasks such as combinatorial optimization and certain simulation tasks - without overhauling the existing software stack.

From Data Center to Global AI Factories.

Soon, individual data centers will be interconnected worldwide and not isolated. These “AI factories” will operate across continents, with ultra-high-bandwidth optical networks seamlessly linking millions and billions of GPUs into a single, coherent global resource. The performance engineering challenges here include local optimization and cross-data-center synchronization and efficient routing of aggregated results.

Holistic Full-Stack Innovation.

The drive to scale AI models to 100 trillion parameters combines everything from hardware breakthroughs to software optimizations and novel algorithmic approaches. This scale demands integrating low-precision training, sparse computation, advanced optimizer designs, and robust infrastructure capable of handling enormous amounts of data with minimal latency.

Collaborative and Reproducible Research.

As the models grow, so does the complexity of their training and deployment. Successful scaling will depend on standardized formats, collaborative frameworks, and global data sharing and reproducibility efforts that ensure research can be effectively pooled and scaled beyond what any single team or institution can handle alone.

Conclusion

AI Systems Performance Engineering is entering its golden era. This is both challenging and exciting. The frontier of what's possible keeps expanding. Models are hundred times larger. The demand for real-time inference to power advanced reasoning and AGI models is reaching an unprecedented scale. New hardware paradigms continue to emerge, and AI optimizing AI is a reality. The case studies we examined earlier each highlighted how creative engineering turns ambitious ideas into reality. Now, the emerging trends we've discussed will carry those lessons forward into the future.

One overarching theme is the shift from brute-force scaling to intelligent scaling. In the past, to go faster, you might simply throw more GPUs at the problem. In the future, that won't be enough. It simply won't be cost or power efficient. We have to scale wisely. This means embracing algorithmic efficiencies so we're performing more useful work, or goodput. Using the latest hardware features - like Grace-Blackwell's fast interconnects and huge memory - to their fullest. It means letting automation – sometimes AI itself – assist in optimization because systems have grown beyond what any single human can manage. Essentially, performance engineering is moving up a level of abstraction. We need to set up the right environment and incentives for the system to auto tune itself and continue to run optimally - rather than manually tuning every knob ourselves.

The tone of this future is not one of obsolescence for the performance engineer, but of augmentation with the engineer. Yes, many manual tasks will be taken over by smart compilers and AI agents, but our expertise is still crucial to guide these tools. We'll spend more time on high-level architecture decisions, experiment design, interpreting results, and steering the automation in the right direction. The "engineer's intuition" will remain extremely valuable. For example, we intuitively know when a communication pattern is inefficient or a certain metric indicates a bottleneck. We'll just apply it through new interfaces by telling the AI assistant, "Focus on optimizing the all-reduce in this section, I suspect that's our slow point," rather than re-writing the all-reduce code ourselves.

Importantly, the performance engineer's purview is expanding. It's not just GPU kernels or single-node optimizations. It now includes system-wide, facility-wide, and global efficiency. Not to mention reliability, sustainability, and collaboration considerations. We’re becoming the architects of AI compute ecosystems that are immensely complex. It’s a bit like going from being a car mechanic incrementally fixing parts of the car to an automotive engineer designing and building an entire vehicle from scratch. The scope is much broader. This will require continuous learning and adaptation. The pace of new developments for hardware, software, and algorithms will not slow down. So neither should we slow down our understanding of these innovations.

We are entering a period where the full stack is being reinvented - sometimes by AI itself - to feed the insatiable appetite of these ultra-scale models and to push the frontier of what AI can do. It’s a future where a GPU is more than a chip - it’s an integrated computing world where optics, quantum, and silicon all intertwine to create intelligent machines at previously unthinkable scales. And as these trends mature, the boundary of what’s possible in AI will keep expanding, driven forward by the engineers and researchers dedicated to bending hardware and software to their will.

The coming years will likely bring breakthroughs that today feel barely imaginable. Just as a few years ago, training a multi-trillion parameter model felt out of reach, yet we achieved it.

When we hit roadblocks, whether it’s a power constraint or a memory wall or a convergence issue in a giant model, the collective ingenuity of this field finds a way around it. And performance engineering sits at the heart of that innovation, turning lofty AI ambitions into practical reality.

In closing, the best practice I suggest is to stay curious, stay adaptable, and lean into the future. Don’t be afraid to experiment with that new compiler, or trust an AI recommendation, or try out a quantum kernel if it might give you an edge. Build a solid foundation in fundamentals as those are never wasted. And be ready to pivot as the landscape changes. By doing so, you’ll keep up with the future of ultra-scale AI systems - and help create it!

After all, every great leap in AI, from the earliest multi-layer perceptrons (MLPs) to today’s massive Transformer-based GPT and MoE models, has many unsung heroes making it run efficiently and cost-effectively behind the scenes. In the era of 100-trillion-parameter models and beyond, you could be one of those heroes and ensure that the next generation of AI is powerful, efficient, sustainable, and brilliantly engineered. The adventure is just beginning, and I, for one, can’t wait to see what we accomplish next.

Chapter 7. AI Systems Performance Checklist (175+ Items)

This extensive checklist covers both broad process-level best practices and detailed, low-level tuning advice for AI Systems Performance Engineers. Each of these checklist items serves as a practical reminder to squeeze maximum performance and efficiency out of AI systems.

Use this guide when debugging, profiling, analyzing, and tuning one's AI systems. By systematically applying these tips – from low-level OS and CUDA tweaks up to cluster-scale optimizations – an AI Systems Performance Engineer can achieve both lightning-fast execution and cost-effective operation on modern NVIDIA GPU hardware using many AI software frameworks including CUDA, PyTorch, OpenAI's Triton, TensorFlow, Keras, and JAX. The principles in this checklist will also apply to future generations of NVIDIA hardware including their GPUs, ARM-based CPUs, networking gear, and rack systems.

Performance Tuning Mindset and Cost Optimization

Optimize the Expensive First.

Use the 80/20 rule. Find the top contributors to runtime and focus on those. If 90% of time is in a couple kernels or a communication phase, it's better to optimize those deeply than to micro-optimize something taking 1% of time. Each chapter's techniques should be applied where they matter most. E.g., if your training is 40% data loading, 50% GPU compute, 10% communication, then first fix data loading as you can maybe halve the overhead. Then look at GPU kernel optimization.

Profile Before and After.

Whenever you apply an optimization, measure its impact. This sounds obvious but often tweaks are made based on theory and might not help - or even hurt - in practice. Consider a scenario where your workload is not memory-limited, but you decide to try enabling gradient checkpoint for your training job. This may actually slow down the job by using extra compute to reduce memory. In other words, always compare key metrics like throughput, latency, and utilization before and after making changes. Use the built-in profilers for simple timing such as average iteration time over 100 iterations.

Adaptive Auto-Tuning Feedback Loops.

Implement advanced auto-tuning frameworks that leverage real-time performance feedback—using techniques like reinforcement learning or Bayesian optimization—to dynamically adjust system parameters. This approach enables your system to continuously fine-tune settings in response to changing workloads and operating conditions.

Budget for Optimization Time.

Performance engineering is an iterative investment. There’s diminishing returns – pick the low-hanging fruit like enabling AMP and data prefetch. These might give 2× easily. Harder optimizations like writing custom kernels might give smaller increments. Always weigh the engineering time vs. the gain in runtime and cost saved. For large recurring jobs like training a flagship model, even 5% gain can justify weeks of tuning since it saves maybe millions. For one-off or small workloads, focus on bigger wins and be pragmatic.

Stay Updated on Framework Improvements.

Many optimizations we discussed such as mixed precision, fused kernels, and distributed algorithms continue to be improved in deep learning frameworks and libraries. Upgrading to the latest PyTorch or TensorFlow can sometimes yield immediate speedups as they incorporate new fused ops or better heuristics. For example, PyTorch 2.0’s torch.compile yielded ~43% speedups on average for 163 models by better fusion and lowering Leverage these improvements as they are essentially free gains. Read release notes for performance-related changes.

Vendor and Community Co-Design Collaboration.

Stay connected with hardware vendors and the broader performance engineering community to align software optimizations with the latest hardware architectures. This co-design approach can reveal significant opportunities for performance gains by tailoring algorithms to leverage emerging hardware capabilities. Regularly review vendor documentation, participate in forums, and test beta releases of drivers or frameworks. These interactions often reveal new optimization opportunities and best practices that can be integrated into your systems. Integrating new driver optimizations, library updates, and hardware-specific tips can provide additional, sometimes significant, performance gains.

Leverage Cloud Flexibility for Cost.

If running in cloud environments, use cheaper spot instances or reserved instances wisely. They can drastically cut costs, but you may lose the spot instances with a few minutes notice. Also consider instance types as sometimes a slightly older GPU instance at fraction of cost can deliver better price/performance if your workload doesn't need the absolute latest. Our discussions on H800 vs H100 showed it's possible to do great work on second-best hardware with effort. In the cloud, you can get similar trade-offs. Evaluate cost/performance by benchmarking on different instance configurations including number of CPUs, CPU memory, number of GPUs, GPU memory, L1/L2 caches, unified memory, NVLink/NVSwitch interconnects, network bandwidth and latency, and local disk configuration. Calculating throughput per dollar..

Monitor Utilization Metrics.

Continuously monitor GPU utilization, SM efficiency, memory bandwidth usage, and for multi-node, network utilization. Set up dashboards using DCGM exporter, Prometheus, etc. so you can catch when any resource is underused. If GPUs are at 50% utilization, dig into why. It’s likely data waiting/stalling and slow synchronization communication. If the network is only 10% utilized but GPU waits on data, maybe something else like a lock is the issue. These metrics help pinpoint which subsystem to focus on.

Iterate and Tune Hyperparameters for Throughput.

Some model hyperparameters such as batch size, sequence length, and number of MoE active experts can be tuned for throughput without degrading final accuracy.

For example, larger batch sizes give better throughput but might require tuning the learning rate schedule to maintain accuracy. Don't be afraid to adjust these to find a sweet spot of speed and accuracy. This is part of performance engineering too – sometimes the model or training procedure can be adjusted for efficiency like using gradient checkpointing or more steps of compute for the same effective batch. You might tweak the training learning rate schedule to compensate for this scenario.

Document and Reuse.

Keep notes of what optimizations you applied and their impact. Document in code or in an internal wiki-like shared knowledge-base system. This builds a knowledge base for future projects. Many tips are reusable patterns like enabling overlapping, particular environment variables that help on a cluster. Having this history can save time when starting a new endeavor or when onboarding new team members into performance tuning efforts.

Balance Optimizations with Complexity.

Aim for the simplest solution that achieves needed performance. For example, if native PyTorch with torch.compile meets your speed target, you might not need to write custom CUDA kernels. This will help avoid extra maintenance. Over-optimizing with highly custom code can make the system brittle. There is elegance in a solution that is both fast and maintainable. Thus, apply the least-intrusive optimization that yields the required gain, and escalate to more involved ones only as needed.

AI-Driven Performance Optimization.

Leverage machine learning models to analyze historical telemetry data and predict system bottlenecks, enabling automated adjustments of parameters in real time to optimize resource allocation and throughput.

Reproducibility and Documentation Best Practices

Rigorous Version Control.

Maintain comprehensive version control for all system configurations, optimization scripts, and benchmarks. Use Git (or a similar system) to track changes and tag releases, ensuring that experiments can be reproduced and performance regressions are easily identified.

Continuous Integration for Performance Regression.

Integrate automated performance benchmarks and real-time monitoring into your CI/CD pipelines. This ensures that each change - from code updates to configuration changes - is validated against a set of performance metrics, helping catch regressions early, and maintaining consistent and measurable performance gains. Adopt industry-standard benchmarks such as MLPerf or DeepBench to establish a reliable performance baseline and track improvements over time.

End-to-End Workflow Optimization.

Ensure that optimizations are applied holistically across the entire AI pipeline—from data ingestion and preprocessing through training and inference deployment. Coordinated, cross-system tuning can reveal synergies that isolated adjustments might miss, resulting in more significant overall performance gains.

Automated Monitoring and Diagnostics.

Deploy end-to-end monitoring solutions that collect real-time metrics across hardware, network, and application layers. Integrate these with dashboards such as Prometheus/Grafana and configure automated alerts to promptly detect anomalies such as sudden drops in GPU utilization or spikes in network latency.

Fault Tolerance and Automated Recovery.

Incorporate fault tolerance into your system design by using distributed checkpointing, redundant hardware configurations, and dynamic job rescheduling. This strategy minimizes downtime and maintains performance even in the face of hardware or network failures.

Compiler and Build Optimizations.

Leverage aggressive compiler flags and profile-guided optimizations during the build process to extract maximum performance from your code. Regularly update and tune your build configurations, and verify the impact of each change through rigorous benchmarking to ensure optimal execution.

Security and Compliance Integration.

Balance performance optimizations with robust security practices by regularly auditing system configurations, enforcing secure access controls, and maintaining compliance with industry standards. This ensures that performance enhancements do not inadvertently compromise system security.

Security and Performance Co-Optimization.

Ensure that security best practices are integrated during performance tuning efforts by using encryption, secure data channels, zero-trust networking models, regular vulnerability audits, enforcing secure access controls, and maintaining compliance with industry standards. This way, optimized configurations do not inadvertently expose system weaknesses.This approach also ensures that, by adding security layers such as hardware security modules (HSMs) and secure enclaves, you are not impacting workload performance.

Comprehensive Documentation and Knowledge Sharing.

Maintain detailed records of all optimization steps, system configurations, and performance benchmarks. Develop an internal knowledge base to facilitate team collaboration and rapid onboarding, ensuring that best practices are preserved and reused across projects.

Future-Proofing and Scalability Planning.

Design modular, adaptable system architectures that can easily incorporate emerging hardware and software technologies. Continuously evaluate scalability requirements and update your optimization strategies to sustain competitive performance as your workload grows.

System Architecture and Hardware Planning

Design for Goodput and Efficiency.

Treat useful throughput as the goal. Every bit of performance gained translates to massive cost savings at scale. Focus on maximizing productive work per dollar/joule, not just raw FLOPs.

Choose the Right Accelerator.

Prefer modern GPUs like the Blackwell GB200 for superior performance-per-watt and memory capacity. Newer architectures offer features like FP8 support and faster interconnects that yield big speedups over older GPUs.

Leverage High-Bandwidth Interconnects.

Use systems with NVLink/NVSwitch such as GB200 NVL72 instead of PCIe-only connectivity for multi-GPU workloads. NVLink 5 provides up to 1.8 TB/s GPU-to-GPU bandwidth (over 14× PCIe Gen5), enabling near-linear scaling across GPUs.

Balance CPU/GPU and Memory Ratios.

Provision enough CPU cores, DRAM, and storage throughput per GPU. For example, allocate ~1 fast CPU core per GPU for data loading and networking tasks. Ensure system RAM and I/O can feed GPUs at required rates on the order of hundreds of MB/s per GPU to avoid starvation.

Plan for Data Locality.

If training across multiple nodes, minimize off-node communication. Whenever possible, keep tightly-coupled workloads on the same NVLink/NVSwitch domain to exploit full bandwidth, and use the highest-speed interconnect that you have access to such as NVLink and InfiniBand for inter-node communication.

Avoid Bottlenecks in the Chain.

Identify the slowest link – be it CPU, memory, disk, or network – and scale it up. For instance, if GPU utilization is low due to I/O, invest in faster storage or caching rather than more GPUs. An end-to-end design where all components are well-matched prevents wasted GPU cycles.

Right-Size Cluster Scale.

Beware of diminishing returns when adding GPUs. Past a certain cluster size, overheads can grow – ensure the speedup justifies the cost. It’s often better to optimize utilization on N GPUs by reaching 95% usage, for example, before scaling to 2N GPUs.

Design for Cooling and Power.

Ensure the data center can handle GPU thermal and power needs. High-performance systems like GB200 have very high TDP – provide adequate cooling (likely liquid-based) and power provisioning so GPUs can sustain boost clocks without throttling.

Grace-Blackwell GB200 Unified Architecture

Unified CPU-GPU Memory.

Exploit the Grace-Blackwell (GB200) superchip’s unified memory space. Two Blackwell GPUs and a 72-core Grace CPU share a coherent memory pool via NVLink-C2C (900

GB/s). Use the CPU's large memory (e.g. 512 GB LPDDR5X) as an extension for oversize models while keeping "hot" data in the GPUs' HBM for speed.

Place Data for Locality.

Even with unified memory, prioritize data placement. Put model weights, activations, and other frequently accessed data in on-GPU HBM3e (which has much higher local bandwidth), and let infrequently used or overflow data reside in CPU RAM. This ensures the 900 GB/s NVLink-C2C link isn't a bottleneck for critical data.

GPU-Direct Access to CPU Memory.

Take advantage of the GPU's ability to directly access CPU memory on GB200. The GPUs can fetch data from CPU RAM without CPU involvement, making techniques like memory-mapped files or CPU-side data preparation more efficient. However, be mindful of latency – sequential or batched access patterns work best.

Use the Grace CPU Effectively.

The on-package Grace CPU provides 72 high-performance cores – utilize them! Offload data preprocessing, augmentation, and other CPU-friendly tasks to these cores. They can feed the GPUs quickly via NVLink-C2C, essentially acting as an extremely fast I/O and compute companion for the GPU.

Plan for Ultra-Large Models.

For trillion-parameter model training that exceeds GPU memory, GB200 systems allow you to train using CPU memory as part of the model's memory pool. Use CUDA Unified Memory or managed memory APIs to handle overflow gracefully, and consider explicit prefetching of upcoming layers from CPU->GPU memory to hide latency.

Multi-GPU Scaling and Interconnect Optimizations

All-to-All Topology.

On NVL72 NVSwitch clusters with 72 GPUs fully interconnected, for example, any GPU can communicate with any other at full NVLink 5 speed. Take advantage of this topology by using parallelization strategies such as data parallel, tensor parallel, and pipeline parallelism that would be bottlenecked on lesser interconnects.

Topology-Aware Scheduling.

Always co-locate multi-GPU jobs within an NVLink Switch domain if possible. Keeping all GPUs of a job on the NVL72 fabric means near-linear scaling for communication-heavy workloads. Mixing GPUs across NVLink domains or standard networks will introduce bottlenecks and should be avoided for tightly-coupled tasks.

Leverage Unprecedented Bandwidth. Recognize that NVLink 5 has 900 GB/s per GPU each direction which doubles the per-GPU bandwidth vs. the previous generation. A NVL72 rack provides 130 TB/s total bisection bandwidth. This drastically reduces communication wait times as even tens of gigabytes of gradient data can be all-reduced in a few milliseconds at 1.8 TB/s. Design training algorithms such as gradient synchronization and parameter sharding to fully exploit this relatively-free communication budget.

New Collective Algorithms.

Use the latest NVIDIA NCCL library optimized for NVSwitch. Enable features like NCCL's new low-latency all-reduce or the parallel-aggregated tree (PAT) algorithm introduced for NVLink Switch environments to further cut down synchronization time. These algorithms take advantage of the NVL72 topology to perform reductions more efficiently than older tree/ring algorithms.

Fine-Grained Parallelism.

With full-bandwidth all-to-all connectivity, consider fine-grained model parallelism that wasn't feasible before. For example, layer-wise parallelism or tensor parallelism across many GPUs can be efficient when each GPU has 1.8 TB/s to every other. Previously, one might avoid excessive cross-GPU communication, but NVL72 allows aggressive partitioning of work without hitting network limits.

Monitor for Saturation.

Although NVL72 is extremely fast, keep an eye on link utilization in profiling. If your application somehow saturates the NVSwitch using extreme all-to-all operations, for example, you might need to throttle communication by aggregating gradients, etc. Use NVIDIA's tools or NVSwitch telemetry to verify that communications are within the NVLink capacity, and adjust patterns if needed. For instance, you can stagger all-to-all exchanges to avoid network contention.

Future Expansion.

Be aware that NVLink Switch can scale beyond a single rack – up to 576 GPUs in one connected domain via second-level switches. If you operate at that ultra-scale, plan hierarchical communication using local NVL72 inter-rack collectives first, then use inter-rack interconnects only when necessary. This helps to maximize intra-rack NVLink usage first. This ensures you're using the fastest links before resorting to inter-rack InfiniBand hops.

Federated and Distributed Optimization.

For deployments that span heterogeneous environments such as multi-cloud or edge-to-cloud setups, adopt adaptive communication protocols and dynamic load balancing strategies. This minimizes latency and maximizes throughput across distributed systems which ensures robust performance even when resources vary in capability and capacity.

Operating System and Driver Optimizations

Use a Linux Kernel Tuned for HPC.

Ensure your GPU servers run a recent, stable Linux kernel configured for high-performance computing. Disable unnecessary background services that consume CPU or I/O. Use the “performance” CPU governor - versus “on-demand” or “power-save” - to keep CPU cores at high clock for feeding GPUs.

Disable Swap for Performance-Critical Workloads.

Disable swap on training servers to avoid page thrashing, or, if swap must remain enabled, lock critical buffers using `mlock` or `cudaHostAlloc` to ensure they stay in RAM.

Avoid Memory Fragmentation with Aggressive Preallocation.

Preallocate large, contiguous blocks of memory for frequently used tensors to reduce runtime allocation overhead and fragmentation. This proactive strategy ensures a more stable and efficient memory management during long training runs.

Optimize Environment Variables for CPU Libraries.

Fine-tune parameters such as `OMP_NUM_THREADS` and `MKL_NUM_THREADS` to better match your hardware configuration. Adjusting these variables can reduce thread contention and improve the parallel efficiency of CPU-bound operations.

NUMA Awareness.

For multi-NUMA servers, pin GPU processes/threads to the CPU of the local NUMA node. Use tools like `numactl` or taskset to bind each training process to the CPU nearest its assigned GPU. Similarly, bind memory allocations to the local NUMA node (`numactl --membind`) so host memory for GPU DMA comes from the closest RAM. This avoids costly cross-NUMA memory traffic that can halve effective PCIe/NVLink bandwidth.

IRQ Affinity for Network and GPU Tasks.

Explicitly bind NIC interrupts to CPU cores on the same NUMA node as the NIC, and similarly pin GPU driver threads to dedicated cores - including those from long-running services like the `nvidia-persistence` GPU Persistence Mode service daemon. This strategy minimizes cross-NUMA traffic and stabilizes performance under heavy loads.

Enable Transparent Huge Pages.

Turn on Transparent Huge Pages (THP) in always or `madvise` mode so that large memory allocations use 2

MB pages. This reduces TLB thrashing and kernel overhead when allocating tens or hundreds of GBs of host memory for frameworks. Verify THP is active by checking for `/sys/kernel/mm/transparent_hugepage/enabled`. With THP enabled, your processes are using hugepages for big allocations.

Increase Max Locked Memory.

Configure the OS to allow large pinned (a.k.a page-locked) allocations. GPU apps often pin memory for faster transfers – set `ulimit -l unlimited` or a high value so your data loaders can allocate pinned buffers without hitting OS limits. This prevents failures or falls back to pageable memory which would slow down GPU DMA.

NVIDIA Driver and CUDA Stack.

Keep NVIDIA drivers and CUDA runtime up-to-date (within a tested stable version) on all nodes. New drivers can bring performance improvements and are required for new GPUs' compute capabilities. Ensure all nodes have the same driver/CUDA versions to avoid any mismatches in multi-node jobs. Enable persistence mode on GPUs at boot (`nvidia-smi -pm 1`) so the driver stays loaded and GPUs don't incur re-init delays.

GPU Persistence & MIG Settings.

With persistence mode enabled, the GPU remains “warm” and ready to use, reducing startup latency for jobs. This is especially crucial if using MIG partitioning – without persistence, MIG configurations would reset on every job, but keeping the driver active preserves the slices. Always configure persistence mode when using a Multi-Instance GPU.

Isolate System Tasks.

Dedicate a core - or small subset of cores - on each server for OS housekeeping such as interrupt handling and background daemons. This way, your main CPU threads feeding the GPU are not interrupted. This can be done via CPU isolation or cgroup pinning. Eliminating OS jitter ensures consistent throughput.

Fast System I/O Settings.

If your workload does a lot of logging or checkpointing, mount filesystems with options that favor throughput. Consider using `noatime` for data disks and increase file system read-ahead for streaming reads. Ensure disk scheduler is set appropriately to use `mq-deadline` or `noop` for NVMe SSDs to reduce latency variability.

Regular Maintenance.

Keep BIOS/firmware updated for performance fixes. Some BIOS updates improve PCIe bandwidth or fix IOMMU issues for GPUs. Also, periodically check for firmware updates for NICs and NVSwitch/Fabric if applicable, as provided by NVIDIA such as Fabric Manager upgrades, etc. Minor firmware tweaks can sometimes resolve obscure bottlenecks or reliability issues.

Dedicated Resources for System Tasks.

Reserve specific CPU cores solely for operating system tasks like interrupt handling and background daemons. Isolating these tasks prevents interference with latency-critical GPU operations.

Docker and Kubernetes Tuning.

When running in containers, add options such as `--ipc=host` for shared memory and set `--ulimit memlock=-1` to prevent memory locking issues. This guarantees that your containerized processes access memory without OS-imposed restrictions.

Topology-Aware Job Scheduling.

Ensure that orchestrators like Kubernetes and SLURM are scheduling containers on nodes that respect NUMA and NVLink boundaries to minimize cross-NUMA and cross-NVLink-domain memory accesses. This alignment reduces latency and improves overall throughput.

GPU Resource Management and Scheduling

Multi-Process Service (MPS).

Enable NVIDIA MPS when running multiple processes on a single GPU to improve utilization. MPS allows kernels from different processes to execute concurrently on the GPU instead of time-slicing. This is useful if individual jobs don’t fully saturate the GPU – for example, running 4 training tasks on one GPU with MPS can overlap their work and boost overall throughput.

Multi-Instance GPU (MIG).

Use MIG to partition high-end GPUs into smaller instances for multiple jobs. If you have many light workloads like inferencing small models or running many experiments, you can slide a GPU to ensure guaranteed resources for each job. For instance, a Hopper H100 can be split into 7 MIG slices. Do not use MIG for tightly-coupled parallel jobs as those benefit from full GPU access. Deploy MIG for isolation and maximizing GPU ROI when jobs are smaller than a full GPU.

Persistence for MIG.

Keep persistence mode on to maintain MIG partitions between jobs. This avoids re-partitioning overhead and ensures subsequent jobs see the expected GPU slices without delay. Configure MIG at cluster boot and leave it enabled so that scheduling is predictable as changing MIG config on the fly requires a server reboot and GPU reset which can disrupt running jobs.

GPU Clock and Power Settings.

Consider locking GPU clocks to a fixed high frequency via `nvidia-smi -lgc`/`-lmc` if you need run-to-run consistency. By default, GPUs use auto boost which is usually optimal, but fixed clocks can avoid any transient downclocking. In power-constrained scenarios, you might slightly underclock or set a power limit to keep GPUs in a stable thermal/power envelope – this can yield consistent performance if occasional throttling was an issue.

ECC Memory.

Keep ECC enabled on data center GPUs for reliability unless you have a specific reason to disable it. The performance cost is minimal - on the order of a few percent loss in bandwidth and memory, but ECC catches memory errors that could otherwise corrupt a long training job. Most server GPUs ship with ECC on by default. Leave it on to safeguard multi-week training.

Cluster Scheduler Awareness.

Integrate GPU topology into your job scheduler such as SLURM and Kubernetes. Configure the scheduler to allocate jobs on the same node or same NVSwitch group when low-latency coupling is needed. Use Kubernetes device plugins or Slurm Gres to schedule MIG slices for smaller jobs. A GPU-aware scheduler prevents scenarios like a single job spanning distant GPUs and suffering bandwidth issues.

Avoid CPU Oversubscription.

When scheduling jobs, account for CPU needs of each GPU task such as data loading threads, etc. Don't pack more GPU jobs on a node than the CPUs can handle. It's better to leave a GPU idle than to overload the CPU such that all

GPUs become underfed. Monitor CPU utilization per GPU job to inform scheduling decisions.

Use Fabric Manager for NVSwitch.

On systems with NVSwitch the GB200 NVL72 racks, ensure NVIDIA Fabric Manager is running. It manages the NVSwitch topology and routing. Without it, multi-GPU communication might not be fully optimized or could even fail for large jobs. This service typically runs by default on NVSwitch-equipped servers, but you should double-check that it's enabled - especially after driver updates.

Job Packing for Utilization.

Maximize utilization by intelligently packing jobs. For example, on a 4-GPU node, if you have two 2-GPU jobs that don't use much CPU, running them together on the same node can save resources and even use the faster NVLink for communication if running together inside the same compute node or NVLink-enabled rack. Conversely, avoid co-locating jobs that collectively exceed memory or I/O capacity of the node. The goal is high hardware utilization without contention.

Data Pipeline and I/O Optimization

Parallel Data Loading.

Use multiple workers/threads to load and preprocess data for the GPUs. The default of 1-2 data loader workers may be insufficient. Profile and increase the number of data loader processes/threads using PyTorch `DataLoader(num_workers=N)`, for example, until the data input is no longer the bottleneck. High core-count CPUs exist to feed those GPUs, so make sure you utilize them.

Pinned Host Memory for I/O.

Enable pinned (a.k.a page-locked) memory for data transfer buffers. Many frameworks have an option like PyTorch's `pin_memory=True` for its DataLoader to allocate host memory that the GPU can DMA from directly. Using pinned memory significantly improves H2D copy throughput. Combine this with asynchronous transfers to overlap data loading with computation.

Overlap Compute and Data Transfers.

Pipeline your input data. While the GPU is busy computing on batch N, load and prepare batch N+1 on the CPU and transfer it in the background using CUDA streams and non-blocking `cudaMemcpyAsync`. This double-buffering hides latency – the GPU ideally never waits for data. Ensure your training loop uses asynchronous transfers. For example, in PyTorch, you can copy tensors to GPU with `non_blocking=True`. Asynchronous transfer allow the CPU to continue running while the data transfer is in progress in the background. This will improve performance by overlapping computation with data transfer.

Use Fast Storage (NVMe/SSD).

Store training data on fast local NVMe SSDs or a high-performance parallel filesystem. Spinning disks will severely limit throughput. For large datasets, consider each node having a local copy or shard of the data. If using network storage, prefer a distributed filesystem like Lustre with striping, or an object store that can serve many clients in parallel.

I/O Concurrency and Striping.

Avoid bottlenecks from single-file access. If one large file is used by all workers, stripe it across multiple storage targets or split it into chunks so multiple servers can serve it. For instance, break datasets into multiple files and have each data loader worker read different files simultaneously. This maximizes aggregate bandwidth from the storage system.

Optimize Small Files Access.

If your dataset consists of millions of small files, mitigate metadata overhead. Opening too many small files per second can overwhelm the filesystem's metadata server. Solutions. pack small files into larger containers such as tar/recordio files, use data ingestion libraries that batch reads, or ensure metadata caching is enabled on clients. This reduces per-file overhead and speeds up epoch start times.

Client-Side Caching.

Take advantage of any caching layer. If using NFS, increase the client cache size and duration. For distributed filesystems, consider a caching daemon or even manually caching part of the dataset on a local disk. The goal is to avoid repeatedly reading the same data from a slow source. If each node processes the same files at different times, a local cache can drastically cut redundant I/O.

Compress Data Wisely.

Store dataset compressed if I/O is the bottleneck, but use lightweight compression such as LZ4 or ZSTD fast mode. This trades some CPU to reduce I/O volume. If CPU becomes the bottleneck due to decompression, consider multithreaded decompression or offload to accelerators. Also, overlap decompression with reading by using one thread to read compressed data and another thread to decompress the data in parallel.

Measure Throughput & Eliminate Bottlenecks.

Continuously monitor the data pipeline's throughput. If GPUs aren't near 100% utilization and you suspect input lag, measure how many MB/s you're reading from disk and how busy the data loader cores are. Tools like `dstat` or NVIDIA's DCGM can reveal if GPUs are waiting on data. Systematically tune each component by bumping up prefetch buffers, increase network buffer sizes, optimize disk RAID settings, etc. Do this until the input pipeline can feed data as fast as GPUs consume it. Often, these optimizations raise GPU utilization from ~70% to >95% on the same hardware by removing I/O stalls.

Scale I/O for Multi-Node.

At cluster scale, ensure the storage system can handle aggregate throughput. For example, 8 GPUs consuming 200 MB/s each is 1.6 GB/s per node. Across 100 nodes that's 160 GB/s needed. Very few central filesystems can sustain this. Mitigate by sharding data across storage servers, using per-node caches, or pre-loading data onto each node's local disk. Trading off storage space for throughput (e.g., multiple copies of data) is often worth it to avoid starving expensive GPUs.

Checkpointing and Logging.

Write checkpoints and logs efficiently. Use asynchronous writes for checkpoints if possible, or write to local disk then copy to network storage to avoid stalling training. Compress checkpoints or use sparse storage formats to reduce size. Limit logging frequency on each step by aggregating iteration statistics and logging only every Nth iteration rather than every iteration. This will greatly reduce I/O overhead.

Workload Profiling and Monitoring

Profile to Find Bottlenecks and Root Cause Analysis.

Regularly run profilers on your training/inference jobs. Use NVIDIA Nsight Systems to get a timeline of CPU and GPU activity. You can also use Nsight Compute or the PyTorch/TensorFlow Profiler to drill down into kernel efficiency. Identify whether your job is compute-bound, memory-bound, or waiting on I/O/communication. Target your optimizations accordingly. For example, if your workload is memory-bound, focus on reducing memory traffic rather than implementing compute-bound optimizations. Combined with machine-learning–driven analytics to predict and preempt performance bottlenecks. This can help in automating fine-tuning adjustments in real time.

Eliminate Python Overhead.

Profile your training scripts to identify Python bottlenecks - such as excessive looping or logging - and replace them with vectorized operations or optimized library calls. Minimizing Python overhead helps ensure that the CPU does not become a hidden bottleneck in the overall system performance.

Measure GPU Utilization and Idle Gaps.

Continuously monitor GPU utilization, SM efficiency, memory bandwidth usage, etc. If you notice periodic drops in utilization, correlate them with events. For example, a drop in utilization every 5 minutes might coincide with checkpoint saving. Such patterns point to optimization opportunities such as staggering checkpoints and use asynchronous flushes. Utilize tools like DCGM or nvidia-smi in daemon mode to log these metrics over time.

Use NVTX Markers.

Instrument your code with NVTX ranges or framework profiling APIs to label different phases including data loading, forward pass, backward pass, etc. These markers show up in Nsight Systems timeline and help you attribute GPU idle times or latencies to specific parts of the pipeline. This makes it easier to communicate with developers which part of the code needs attention.

Kernel Profiling and Analysis.

For performance-critical kernels, use Nsight Compute or `nvprof` to inspect metrics. Check achieved occupancy, memory throughput, and instruction throughput. Look for signs of memory bottlenecks such as memory bandwidth near the hardware maximum. This helps to identify memory-bound workloads. The profiler's "Issues" section often directly suggests if a kernel is memory-bound or compute-bound and why. Use this feedback to guide code changes such as improving memory coalescing if global load efficiency is low.

Check for Warp Divergence.

Use the profiler to see if warps are diverging as it can show branch efficiency and divergent branch metrics. Divergence means some threads in a warp are inactive due to branching, which hurts throughput. If significant, revisit the kernel code to restructure conditionals or data assignments to minimize intra-warp divergence and ensure that each warp handles uniform work.

Verify Load Balancing.

In multi-GPU jobs, profile across ranks. Sometimes one GPU (rank 0) does extra work like aggregating stats and data gathering - and often becomes a bottleneck. Monitor each GPU’s timeline. If one GPU is consistently lagging, distribute that extra workload. For example, you can have the non-zero ranks share the I/O and logging responsibilities. Ensure that all GPUs/ranks have similar workload avoids the slowest rank dragging the rest.

Monitor Memory Usage.

Track GPU memory allocation and usage over time. Ensure you are not near OOM, which can cause the framework to unexpectedly swap tensors to host which will cause huge slowdowns. If memory usage climbs iteration by iteration, you have likely identified leaks. In this case, profile with tools like `torch.cuda.memory_summary()` and Nsight to analyze detailed allocations. On the CPU side, monitor for paging as your process’s resident memory (RES) should not exceed physical RAM significantly. If you see paging, reduce dataset preload size or increase RAM.

Network and Disk Monitoring.

For distributed jobs, use OS tools to monitor network throughput and disk throughput. Ensure the actual throughput matches expectations. For example, on a 100 Gbps link you should see 12 GB/s if fully utilized. If not, the network might be a bottleneck or misconfigured.

Similarly, monitor disk I/O on training nodes. If you see spikes of 100% disk utilization and GPU idle, you likely need to buffer or cache data better.

Set Up Alerts for Anomalies.

In a production or long-running training context, set up automated alerts or logs for events like GPU errors such as ECC errors, device overheating, etc. This will help identify abnormally-slow iterations. For example, NVIDIA’s DCGM can watch health metrics and you can trigger actions if a GPU starts throttling or encountering errors. This helps catch performance issues - like a cooling failure causing throttling - immediately rather than after the job finishes.

Perform Regression Testing.

Maintain a set of benchmark tasks to run whenever you change software including CUDA drivers, CUDA versions, AI framework versions, or even your training code. Compare performance to previous runs to catch regressions early. It’s not uncommon for a driver update or code change to inadvertently reduce throughput – a quick profiling run on a standard workload will highlight this so you can investigate. For example, maybe a kernel is accidentally not using Tensor Cores, anymore. This is something to look into, for sure.

GPU Programming and CUDA Tuning Optimizations

Understand GPU Memory Hierarchy.

Keep in mind the tiered memory structure of GPUs – registers per thread, shared memory/L1 cache per block/SM, L2 cache across SM, and global HBM. Maximize data reuse in the higher tiers. For example, use registers and shared memory to reuse values and minimize accesses to slower global memory. A good kernel ensures the vast majority of data is either in registers or gets loaded from HBM efficiently using coalescing and caching.

Coalesce Global Memory Accesses.

Ensure that threads in the same warp access contiguous memory addresses so that the hardware can service them in as few transactions as possible. Strided or scattered memory access by warp threads will result in multiple memory transactions per warp, effectively wasting bandwidth. Restructure data layouts or index calculations so that, whenever a warp loads data, it's doing so in a single, wide memory transaction.

Use Shared Memory for Data Reuse.

Shared memory is like a manually managed cache with very high bandwidth. Load frequently used data - such as tiles of matrices - into shared memory. And have threads operate on those tiles multiple times before moving on. This popular tiling technique greatly cuts down global memory traffic. Be cautious of shared memory bank conflicts. Organize shared memory access patterns or pad data to ensure threads aren’t contending for the same memory bank, which would serialize accesses and reduce performance.

Optimize Memory Alignment.

Align data structures to 128 bytes whenever possible, especially for bulk memory copies or vectorized loads. Misaligned accesses can force multiple transactions even if theoretically coalesced. Using vectorized types like float2 and float4 for global memory I/O can help load/store multiple values per instruction, but ensure your data pointer is properly aligned to the vector size.

Minimize Memory Transfers.

Only transfer data to the GPU when necessary and in large chunks. Consolidate many small transfers into one big transfer if you can. For example, if you have many small arrays to send each iteration, pack them into one buffer and send once. Small, frequent `cudaMemcpy` can become a bottleneck. If using Unified Memory, use explicit prefetch (`cudaMemPrefetchAsync`) to stage data on GPU before it's needed, avoiding on-demand page faults during critical compute sections.

Avoid Excessive Temporary Allocations.

Frequent allocation and freeing of GPU memory can hurt performance. For example, frequently using `cudaMalloc/cudaFree` or device malloc in kernels will cause extra overhead. Instead, reuse memory buffers or use memory pools available within most DL frameworks, like PyTorch, that implement a GPU caching allocator. If writing custom CUDA code, consider using `cudaMallocAsync` with a memory pool or manage a pool of scratch memory yourself to avoid the overhead of repetitive alloc/free.

Balance Threads and Resource Use.

Achieve a good occupancy-resource balance. Using more threads for higher occupancy helps hide memory latency, but if each thread uses too many registers or too much shared memory, occupancy drops. Tune your kernel launch parameters - including threads per block - to ensure you have enough warps in flight to cover latency, but not so many that each thread is starved of registers or shared memory. In kernels with high instruction-level parallelism (ILP), reducing register usage to boost occupancy might actually hurt performance. The optimal point is usually in the middle of the occupancy spectrum as maximum occupancy is not always ideal. Use the NVIDIA Nsight Compute Occupancy Calculator to experiment with configurations.

Register and Shared Memory Usage.

Continuously monitor per-thread register and shared memory consumption using profiling tools like Nsight Compute. If the occupancy is observed to be below 25%, consider increasing the number of threads per block to better utilize available hardware resources. However, verify that this adjustment does not cause excessive register spilling by reviewing detailed occupancy reports and kernel execution metrics. Register spilling can lead to additional memory traffic and degrade overall performance.

Overlap Memory Transfers with Computation.

Employ `cudaMemcpyAsync` in multiple CUDA streams to prefetch data while computation is ongoing. On modern GPUs such as those based on the Hopper architecture and later, take advantage of the `cp.async` feature to asynchronously copy data from global memory to shared memory. This approach effectively masks global memory latency by overlapping data transfers with computation, ensuring that the GPU cores remain fully utilized without waiting for memory operations to complete.

Manual Prefetching.

Integrate manual prefetching strategies within your CUDA kernels by utilizing functions like `__prefetch_global`, or by explicitly loading data into registers ahead of its use. This proactive method reduces the delay caused by global memory accesses and ensures that critical data is available in faster, lower-latency storage - such as registers or shared memory - right when it's needed, thus minimizing execution stalls and improving overall kernel efficiency.

Cooperative Groups.

Utilize CUDA's cooperative groups to achieve efficient, localized synchronization among a subset of threads rather than enforcing a full block-wide barrier. This technique enables finer-grained control over synchronization, reducing unnecessary waiting times and overhead. By grouping threads that share data or perform related computations, you can synchronize only those threads that require coordination, which can lead to a more efficient execution pattern and better overall throughput.

Optimize Warp Divergence.

Structure your code so that threads within a warp follow the same execution path as much as possible. Divergence can double the execution time for that warp. For example, half a warp (16 threads) taking one branch, and half the warp (16 threads) taking another branch. If you have branches that some data rarely triggers, consider “sorting” or grouping data so warps handle uniform cases such that all are true or all are false. Use warp-level primitives like ballot and shuffle to create branchless solutions for certain problems. Treat a warp as the unit of work and aim for all 32 threads to do identical work in lockstep for maximum efficiency.

Leverage Warp-Level Operations.

Use CUDA's warp intrinsics to let threads communicate without going to shared memory when appropriate. For example, use `__shfl_sync` to broadcast a value to all threads in a warp or to do warp-level reductions - like summing registers across a warp - instead of each thread writing to shared memory. These intrinsics bypass slower memory and can dramatically speed up algorithms like reductions or scans that can be done within warps. . By processing these tasks within a warp, you avoid the latency associated with shared memory and full-block synchronizations.

Use CUDA Streams for Concurrency.

Within a single process/GPU, launch independent kernels in different CUDA streams to overlap their execution if they don't use all resources. Overlap computation with computation – e.g., one stream computing one part of the model while another stream launches an independent kernel like data preprocessing on GPU or asynchronous `memcpy`. Be mindful of dependencies and use CUDA events to synchronize when needed. Proper use of streams can increase GPU utilization by not leaving any resource idle - especially if you have some kernels that are light.

Prefer Library Functions.

Wherever possible, use NVIDIA's optimized libraries such as cuBLAS, cuDNN, Thrust, NCCL, and NIXL for common operations. These are heavily optimized for each GPU architecture and often approach theoretical "speed of light" peaks. This will save you the trouble of reinventing them. For example, use cuBLAS GEMM for matrix multiplies rather than a custom kernel, unless you have a very special pattern. The libraries also handle new hardware features transparently.

Kernel Fusion.

Fuse small operations into a single kernel launch when feasible to reduce launch overhead and global memory round-trips. For example, if you have back-to-back element-wise kernels like activation followed by dropout, combine them into one kernel that does both in one pass through the data. This avoids writing intermediate results to global memory and reading them back in the next kernel. Be cautious not to create monolithic kernels that are hard to maintain. Focus fusion efforts on lightweight operations that are bandwidth-bound.

Use CUDA Graphs for Repeated Launches.

If you have a static training loop that is launched thousands of times, consider using CUDA Graphs to capture and launch the sequence of operations as a graph. This can significantly reduce CPU launch overhead for each iteration, especially in multi-GPU scenarios where launching many kernels and `memcpy`’s can put extra pressure on the CPU and incur additional latency.

Check for Scalability Limits.

As you optimize a kernel, periodically check how it scales with problem size and across architectures. A kernel might achieve great occupancy and performance on a small input but not scale well to larger inputs as it may start thrashing L2 cache or running into memory-cache evictions. Use roofline analysis. compare achieved FLOPs and bandwidth to hardware limits to ensure you’re not leaving performance on the table.

Inspect PTX and SASS for Advanced Kernel Analysis.

For performance-critical custom CUDA kernels, use Nsight Compute to examine the generated PTX and SASS. This deep dive can reveal issues like memory bank conflicts or redundant computations, guiding you toward targeted low-level optimizations.

Data Pipeline and Storage Tips

Use Binary Data Formats.

Convert datasets to binary formats such as TFRecords, LMDB, or memory-mapped arrays. This conversion reduces the overhead associated with handling millions of small files and accelerates data ingestion.

File System Tuning.

In addition to mounting file systems with `noatime` and increasing read-ahead, consider sharding data across multiple storage nodes to distribute I/O load and prevent bottlenecks on a single server.

Disable Hyper-Threading for CPU-Bound Workloads.

For data pipelines that are heavily CPU-bound, disabling hyper-threading can reduce resource contention and lead to more consistent performance. This is especially beneficial on systems where single-thread performance is critical.

Elevate Thread Priorities.

Increase the scheduling priority of data loader and preprocessing CPU threads using tools such as `chrt` or `pthread_setschedparam`. By giving these threads higher priority, you ensure that data is fed to the GPU with minimal latency, reducing the chance of pipeline stalls.

CPU and GPU-Based Augmentations.

Distribute heavy preprocessing tasks like image augmentations and text tokenizations across multiple CPU threads or utilize GPU-accelerated libraries like NVIDIA DALI to perform these tasks asynchronously. This helps maintain a smooth and high-throughput data pipeline.

Caching Frequently Used Data.

Leverage operating system page caches or a dedicated RAM disk to cache frequently accessed data. This approach is especially beneficial in applications like NLP, where certain tokens or phrases are accessed repeatedly, reducing redundant processing and I/O overhead.

Prefetch and Buffer Data.

Always load data ahead of the iteration that needs it. Use background data loader threads or processes such as PyTorch DataLoader with `prefetch_factor`. Pin memory on host (`pin_memory=True`) so Host->Device transfers are faster For distributed training, use

`DistributedSampler` to ensure each process gets unique data to avoid redundant I/O.

Use Efficient Data Formats.

Store datasets in a binary format that is quick to read such as TFRecords, LMDB, and memory-mapped arrays - rather than millions of tiny files. Binary format reduces kernel I/O overhead. If using CSV/text, consider converting to a faster format offline. Compress data if disk or network is the bottleneck, as long as decompression is faster which is often true with fast CPUs and I/O bound pipelines.

Parallelize Data Transformations.

If CPU preprocessing - such as image augmentations and text tokenizations - is heavy, distribute it across multiple worker threads/processes. Profile to ensure the CPU isn't the bottleneck while GPUs wait. If it is, either increase workers or move some transforms to GPU as libraries like NVIDIA's DALI can do image operations on a GPU asynchronously..

Cache Frequently-Used Data in Memory and On-Disk.

When inference with LLMs, it's beneficial to cache the embeddings and KV-cache for frequently-seen tokens to avoid having to recompute them repeatedly. Similarly, if an LLM training job reuses the same dataset multiple times (called “epochs”), you should leverage OS page cache or RAM to store the hot data. DeepSeek’s Fire-Flyer File System (3FS) uses the NVMe and RAM of each GPU node to cache shards of data to feed the GPUs locally on that node.

Shard Data Across Nodes.

In multi-node training, give each node a subset of data to avoid every node reading the entire dataset from a single source. This scales out I/O. Use a distributed filesystem or manual shard assignment with each node reading different files. This speeds things up and naturally aligns with data parallel since each node processes its own data shard.

Monitor Pipeline and Adjust Batch Size.

Sometimes increasing batch size will push more work onto GPUs and less frequent I/O, improving overall utilization – but only up to a point as it affects convergence. Conversely, if GPUs are waiting on data often, and you cannot speed I/O, you might actually decrease batch size to shorten each iteration and thus reduce idle time or do gradient accumulation of smaller batches such that data reads are more continuous. Find a balance where GPUs are nearly always busy.

Use Async Data Transfer.

If your framework supports it, overlap data host-to-device copy with GPU compute. In PyTorch, this can be done by moving data transfer to pinned memory and using non-blocking transfers like `to(device, non_blocking=True)` so it happens in parallel with computation on the previous batch.

Data Augmentation on GPU.

If augmentation is simple but applied to massive data like adding noise, or normalization, it might be worth doing on GPU to avoid saturating CPU. GPUs are often underutilized during data loading, so using a small CUDA kernel to augment data after loading can be efficient. But be careful not to serialize the pipeline. Use streams to overlap augmentation of batch N+1 while batch N is training.

End-to-End Throughput Focus.

Remember that speeding up model compute doesn’t help if your data pipeline cuts throughput in half. Always profile end-to-end, not just the training loop isolated. Use tools like NVIDIA’s DLProf - or simply measure batch time when using synthetic data vs real data - to see how much overhead data loading introduces. Aim for <10% overhead from ideal, synthetic data to real data. If it’s more, invest time in pipeline optimization, it often yields large “free” speedups in training.

Precision and Arithmetic Optimizations

Use Mixed Precision Training.

Leverage FP16 or BF16 for training to speed up math operations and reduce memory usage. Modern GPUs have Tensor Cores that accelerate FP16/BF16 matrix operations massively. Keep critical parts like the final accumulation or a copy of weights in FP32 for numerical stability, but run bulk computations in half-precision. This often gives 2-8× speedups with minimal accuracy loss, and is now standard in most frameworks with automatic mixed precision (AMP)

Gradient Accumulation and Checkpointing.

Detail the use of gradient accumulation to effectively increase the batch size without extra memory usage, and consider activation checkpointing to reduce memory footprint in very deep networks. These techniques are crucial when training models that approach or exceed GPU memory limits.

Favor BF16 instead of FP16 on Newer Hardware.

If available, use BF16 instead of FP16 as it has a larger exponent range and doesn't require loss-scaling. GPUs like Ampere/Hopper/Blackwell support BF16 Tensor Cores at the same speed as FP16. BF16 will simplify training by avoiding overflow/underflow issues while still gaining the performance benefits of half precision.

Exploit FP8 and Novel Precisions.

On the Blackwell GPUs and beyond, consider using FP8 precision for even greater speedups. FP8 tensor cores can give ~4× higher throughput than FP16/BF16. This may require retraining models or calibration for tolerance to FP8 noise, but NVIDIA's software stack and Transformer libraries support FP8 training. For inference, Blackwell also introduces an experimental FP4 format which can double inference throughput to 8x higher than FP16/BF16.

Use these ultra-low precisions with caution as they may need per-layer scaling (“microscaling”) or error correction techniques to maintain accuracy.

Leverage Tensor Cores and Warp Matrix Multiple Accumulate (WMMA).

Make sure your custom CUDA kernels utilize Tensor Cores for matrix ops if possible. This might involve using the WMMA API to write warp-level matrix operations or using CUTLASS templates for simplicity. By using Tensor Cores, you can achieve dramatic speedups for GEMM, convolutions, and other tensor operations – often reaching near-peak FLOPs of the GPU. Ensure your data is in FP16/BF16/TF32 as needed and aligned to Tensor Core tile dimensions which are multiples of 8 or 16.

Use TF32 for Easy Speedup.

On Ampere and Hopper GPUs, enable TensorFloat32 (TF32) for FP32 matrix computations. TF32 is the default in cuDNN/cuBLAS now, and it uses 10-bit mantissa internally to speed up FP32 ops on Tensor Cores with minimal impact on convergence. If you have legacy FP32 code, switching to TF32 can give you up to 2-3× speedup with almost no changes to your code. You can switch by using cuBLAS with math mode TensorOps or setting the cudnn/cublas flags.

Exploit Structured Sparsity.

Nvidia Ampere and later GPUs support 2:4 structured sparsity in matrix multiply which zeros out 50% of weights in a structured pattern. This allows the hardware to double its throughput. Leverage this by pruning your model. If you can prune weights to meet the 2:4 sparsity pattern, your GEMMs can run ~2× faster for those layers. Use NVIDIA's SDK or library support to apply structured sparsity and ensure the sparse Tensor Core paths are used. This can give a free speed boost if your model can tolerate or be trained with that sparsity which often requires retraining with sparsity regularization.

Low Precision for Gradients/Activations.

Even if you keep weights at higher precision, consider compressing gradients or activations to lower precision. For instance, use 16-bit or 8-bit communication for gradients. Many frameworks support FP16 gradient all-reduce. Similarly, for activation checkpointing, storing activations in 16-bit instead of 32-bit saves memory. Investigate research like 8-bit optimizers or quantized gradients which can maintain model quality while greatly reducing memory and bandwidth costs.

Custom Quantization for Inference.

For deployment, use INT8 quantization wherever possible. INT8 inference on GPUs is extremely fast and memory-efficient. Use NVIDIA's TensorRT or quantization tools to quantize models to INT8 and calibrate them. Many neural networks like Transformers can run in INT8 with a negligible accuracy drop. The speedups can be 2-4× over FP16. On newest GPUs, also explore and evaluate FP8 or INT4 for certain models to further boost throughput for inference.

Fused Activation + Scaling.

When using lower precision, remember to fuse operations to retain accuracy. For example, Blackwell's FP4 "microscaling" suggests keeping a scale per group of values. Incorporate these fused operations by scaling and computing in one pass - rather than using separate passes which could cause precision loss. Many of these are handled by existing libraries, so just use them rather than implementing from scratch.

Advanced Strategies and Algorithmic Tricks

Auto-Tune Kernel Parameters.

Auto-tune your custom CUDA kernels for the target GPU. Choosing the correct block size, tile size, unroll factors, etc. can dramatically affect performance and the optimal settings often differ between GPUs generations such as Ampere, Hopper, Blackwell, and beyond. Use auto-tuning scripts or frameworks like OpenAI Triton - or even brute-force search in a pre-processing step - to find the best launch config. This can easily yield 20-30% improvements that you'd miss with static "reasonable" settings.

Kernel Fusion in ML Workloads.

Utilize fused kernels provided by deep learning libraries. For example, enabling fused optimizers will fuse elementwise ops like weight update, momentum, etc. This will also use fuse multi-head attention implementations and fuse normalization kernels. NVIDIA's libraries and some open-source projects like Apex and FasterTransformer offer fused operations for common patterns such as layer-norm+dropout, which reduces launch overhead and uses memory more efficiently.

FlashAttention and Memory-Efficient Attention.

Integrate advanced algorithms like FlashAttention for transformer models. FlashAttention computes attention in a tiled, streaming fashion to avoid materializing large intermediate matrices, drastically reducing memory usage and increasing speed - especially for long sequences. Replacing the standard attention with FlashAttention can improve both throughput and memory footprint, allowing larger batch sizes or sequence lengths on the same hardware.

Overlapping Communication & Computation.

In distributed training, overlap network communication with GPU computation whenever possible. For example, with gradient all-reduce, launch the all-reduce asynchronously as soon as each layer's gradients are ready, while the next layer is still computing backward pass. This pipelining can hide all-reduce latency entirely if done right. Use asynchronous NCCL calls or framework libraries like PyTorch's DistributedDataParallel (DDP) which provide overlapping out of the box. This ensures the GPU isn't idle waiting for the network.

Pipeline Parallelism for Deep Models.

When model size forces you to pipeline across GPUs using tensor parallelism or pipeline parallelism, you can use enough micro-batches to keep all pipeline stages busy. Exploit NVLink/NVSwitch to send activations quickly between stages. Overlap and reduce pipeline bubbles by using an interleaved schedule. Some frameworks automate this type of scheduling. The NVL72 fabric is especially helpful here, as even communication-heavy pipeline stages can exchange data at multi-terabyte speeds, minimizing pipeline stalls.

Algorithmic Efficiency Innovations.

Stay updated on new research ideas that improve utilization. For example, selective activation or conditional computation skips parts of the model for certain inputs and saves compute. Mixture-of-experts (MoE) models activate only a few expert networks per sample, and techniques like speculative decoding and early exit in Transformers save work when processing long sequences. These algorithmic changes can yield huge speedups by not doing unnecessary work. They often require custom implementations but are worth the effort for cutting-edge efficiency.

Distributed Optimizer Sharding.

Use a memory-saving optimization strategy like Zero Redundancy Optimizer (ZeRO) which shards optimizer states and gradients across GPUs instead of replicating them. This allows scaling to extreme model sizes by distributing the memory and communication load. It improves throughput by reducing per-GPU memory pressure, avoiding swapping to CPU, and reducing communication volume if done in chunks. Many frameworks like DeepSpeed and Megatron-LM) provide this type of sharding. Leverage it for large models to maintain high speed without running OOM or hitting slowdown from swapping.

Asynchronous and Decoupled Training.

If applicable, consider techniques like gradient compression or asynchronous updates. For example, compress gradients using 8-bit quantization or sparsification before the all-reduce to reduce overall communication time. Or use an asynchronous training regime including a parameter server or stale-synchronous stochastic gradient descent (SGD) where workers don’t always wait for each other. These approaches can increase throughput, though they may require careful tuning to not hurt convergence. In bandwidth-limited environments, gradient compression in particular can be a game-changer, as DeepSeek demonstrated by compressing gradients to train on constrained GPUs.

Incorporate Sparsity and Pruning.

Large models often have redundancy. Use pruning techniques during training to introduce sparsity, which you can exploit at inference - and partially during training if supported. Modern GPU hardware supports accelerated sparse matrix multiply (2:4), and future GPUs will likely extend this feature. Even if you leave training as dense and only prune for inference, a smaller model will run faster and use less memory. This increases cost-efficiency for model deployments. Explore lottery ticket hypothesis, distillation, or structured pruning to maintain accuracy while trimming model size.

Distributed Training and Network Optimization

Use RDMA Networking.

Equip your multi-node cluster with InfiniBand or RDMA over Converged Ethernet (RoCE) for low-latency, high-throughput communication. Ensure your communication libraries such as NCCL, NIXL, and MPI are using RDMA. NCCL will autodetect InfiniBand and use GPUDirect RDMA if available. RDMA bypasses the kernel networking stack and can cut latency by 5-10× versus traditional TCP. If you only have Ethernet, enable RoCE on RDMA-capable NICs to get RDMA-like performance. Be sure to configure lossless Ethernet carefully, however.

Tune TCP/IP Stack if Using Ethernet.

For TCP-based clusters, increase network buffer sizes. Raise `/proc/sys/net/core/{r,w}mem_max` and the autotuning limits (`net.ipv4.tcp_{r,w}mem`) to allow larger send/receive buffers. This helps saturate 10/40/100GbE links. Enable jumbo frames (MTU 9000) on all nodes and switches to reduce overhead per packet, which improves throughput and reduces CPU usage. Also consider modern TCP congestion control like BBR for wide-area or congested networks.

CPU Affinity for NIC.

Pin network interrupts and threads to the CPU core(s) on the same NUMA node as the NIC. This avoids cross-numa penalties for network traffic and keeps the networking stack's memory accesses local. Check

`/proc/interrupts` and use `irqaffinity` settings to ensure, for example, your NIC in NUMA node 0 is handled by a core in NUMA node 0. This can improve network performance and consistency, especially under high packet rates.

NCCL Environment Tweaks.

Experiment with NCCL parameters for large multi-node jobs. For example, increase `NCCL_NTHREADS`, the number of CPU threads per GPU for NCCL, from the default 4 to 8 or 16 to drive higher bandwidth at the cost of more CPU usage. Increase `NCCL_BUFFSIZE`, the buffer size per GPU, from the default 1MB to 4MB or more for better throughput on large messages. If your cluster uses SHARP-enabled switches, enable SHARP by setting `NCCL_SHARP_CAPABLE=1` to offload all-reduce operations to the switch hardware. This can instantly yield 2-5× faster all-reduce for large reductions.

Gradient Accumulation for Slow Networks.

If your network becomes the bottleneck because you are scaling to many nodes linked by a moderate-performance interconnect, use gradient accumulation to perform fewer, larger all-reduce operations. Accumulate gradients over a few mini-batches before syncing, so that you communicate once for N batches instead of every batch. This trades a bit of extra memory and some model accuracy tuning for significantly reduced network overhead. It’s especially helpful when adding more GPUs yields diminishing returns due to communication costs.

All-Reduce Topology Optimization.

Ensure you’re using the optimal all-reduce algorithm for your cluster topology. NCCL will choose ring or tree algorithms automatically, but on mixed interconnects like GPUs connected by NVLink on each node and InfiniBand or Ethernet between nodes, hierarchical all-reduce can be beneficial. Hierarchical all-reduce will first perform the all-reduce operation within the node, then it will proceed across nodes. Most frameworks will perform NCCL-based hierarchical aggregations by default, but verify by profiling. In traditional MPI setups, you may consider manually doing this same two-level reduction - first intra-node and then inter-node.

Avoid Network Oversubscription.

On multi-GPU servers, ensure the combined traffic of GPUs doesn’t oversubscribe the NIC. For example, eight GPUs can easily generate >200 Gbps of traffic during all-reduce, so having only a single 100 Gbps NIC will constrain you. Consider multiple NICs per node or newer 200/400 Gbps InfiniBand if scaling to many GPUs per node. Likewise, watch out for PCIe bandwidth limits if your NIC and GPUs share the same PCIe root complex.

Compress Communication.

Just as with single-node memory, consider compressing data for network transfer. Techniques include 16-bit or 8-bit gradient compression, quantizing activations for cross-node pipeline transfers, or even more exotic methods like sketching. If your network is the slowest component, a slightly higher compute cost to compress/decompress data can be worth it. NVIDIA's NCCL doesn't natively compress, but you can integrate compression in frameworks (e.g., Gradient Compression in Horovod or custom AllReduce hooks in PyTorch). This was one key to DeepSeek's success – compressing gradients to cope with limited inter-node bandwidth.

Monitor Network Health.

Ensure no silent issues are hampering your distributed training. Check for packet loss (which would show up as retries or timeouts – on InfiniBand use counters for resend, on Ethernet check for TCP retransmits). Even a small packet loss can severely degrade throughput due to congestion control kicking in. Use out-of-band network tests (like iperf or NCCL tests) to validate you're getting expected bandwidth and latency. If not, investigate switch configurations, NIC firmware, or CPU affinity as above.

Efficient Inference and Serving

Dynamic Resource Orchestration.

Integrate advanced container orchestration platforms such as Kubernetes augmented with custom performance metrics. This enables dynamic scaling and balancing workloads based on live usage patterns and throughput targets.

Serverless and Microservices for Inference.

Explore serverless architectures and microservice designs for inference workloads, which can handle bursty traffic efficiently and reduce idle resource overhead by scaling down when demand is low.

Optimize Batch and Concurrency.

For inference workloads, find the right batching strategy. Larger batch sizes improve throughput by keeping the GPU busy, but too large can add latency. Use dynamic batching (as in NVIDIA Triton) to automatically batch incoming requests. Also, run multiple inference streams in parallel if one stream doesn't use all GPU resources – e.g., two concurrent inference batches to use both GPU's SMs and tensor cores fully.

Use NVIDIA TensorRT, Dynamo, vLLM, and Optimized Inference Engines.

Deploy models with inference-optimized libraries like NVIDIA TensorRT, Dynamo, and vLLM. These runtimes apply kernel fusion, FP16/INT8 optimizations, and scheduling tricks under the hood. They can dramatically increase throughput and reduce latency versus naive framework inference. Profile your model in TensorRT and compare – often it can use Tensor Cores more effectively and eliminate framework overhead.

Use Quantization.

Leverage INT8 quantization for inference. Many large models can be quantized to INT8 with minimal accuracy loss - often with calibration. INT8 inference uses 4× less memory and often 2-4× faster inference. On modern GPUs, experiment with 4-bit weight quantization for certain models. Combined with structural sparsity and some clever engineering, you can achieve additional speedups.. Always validate accuracy after quantization. Use techniques like calibration or quantization-aware training to maintain model quality.

Leverage NIXL for Distributed Inference.

When serving giant models that span multiple GPUs or nodes, use NVIDIA's Inference Transfer Engine (NIXL) to handle GPU-to-GPU data transfers efficiently. In the case of NIXL, the large Transformer-based key/value cache (KV-Cache) is transferred between nodes. NIXL provides a high-throughput, low-latency API for streaming the KV-Cache from a "prefill" GPU to a "decode" GPU in an LLM inference GPU cluster. It does this using GPUDirect RDMA and optimal paths - and without involving the CPU. This drastically cuts tail latency for distributed inference across nodes.

Offload KV Cache if Necessary.

If an LLM's attention KV cache grows too large to fit in GPU memory, use an offloading strategy. This can happen when the model processes long input sequences. NVIDIA's KV-Cache Offload Manager can spill less-used portions of the cache to NVMe SSD and stream them back on demand. This allows inference on sequences that would otherwise exceed GPU memory - and with minimal performance hit thanks to fast NVMe and compute-I/O overlapping. Ensure your inference server is configured to use this if you expect very long prompts or chats. Offloading to disk is better than failing completely.

Efficient Model Serving.

Use optimized model inference systems such as Nvidia Dynamo, TensorRT-LLM or vLLM for serving large models with low latency and high throughput. They should implement quantization, low-precision formats, highly-optimized attention kernels, and other tricks to maximize GPU utilization during inference. These libraries should also handle tensor parallelism, pipeline parallelism, expert parallelism, sequence parallelism, speculative decoding, chunked prefill, disaggregated prefill/decode, and dynamic request batching - among many other high-performance features.

Monitor and Tune for Tail Latency.

In real-time services, both average latency and (long-)tail latency (99th percentile) matter. Profile the distribution of inference latencies. If the tail is high, identify outlier causes such as unexpected CPU involvement, garbage-collection (GC) pauses, or excessive context switches. Pin your inference server process to specific cores, isolate it from noisy neighbors, and use real-time scheduling if necessary to get more consistent latency. Also warm up the GPUs by loading the model into the GPU and running a few dummy inferences. This will avoid one-time, cold-start latency hits when the first real request comes into the inference server.

Resource Partitioning for QoS.

If running mixed, heterogeneous workloads such as training and inference - or models with different architectures - on the same infrastructure, consider partitioning resources to ensure the latency-sensitive inference gets priority. This could mean dedicating some GPUs entirely to inference, or using MIG to give an inference service a guaranteed slice of a GPU if it doesn't need a full GPU but requires predictable latency. Separate inference from training on different nodes if possible, as training can introduce jitter with heavy I/O or sudden bursts of communication.

Utilize Grace-CPU for Inference Preprocessing.

In Grace-Blackwell systems, the server-class CPU can handle preprocessing - such as tokenization and batch collation - extremely fast in the same memory space as the GPU. Offload such tasks to the CPU and have it prepare data in the shared memory that the GPU can directly use. This reduces duplication of buffers and leverages the powerful CPU to handle parts of the inference pipeline, freeing the GPU to focus on more compute-intensive neural-network computations.

Edge AI and Latency-Critical Deployments.

Extend performance tuning to the edge by leveraging specialized edge accelerators and optimizing data transfer protocols between central servers and edge devices. This will help achieve ultra-low latency for time-sensitive applications.

Power and Thermal Management

Green AI and Energy Efficiency.

Track and optimize energy consumption alongside performance. In addition to managing power and thermal limits, monitor energy usage metrics and consider techniques that improve both performance and sustainability. For example, by implementing dynamic power capping or workload shifting based on renewable energy availability, you can reduce operational costs and carbon footprint. This dual focus reduces operational costs and supports responsible, environmentally friendly AI deployments.

Monitor Thermals and Clocks.

Keep an eye on GPU temperature and clock frequencies during runs. If GPUs approach thermal limits (85°C in some cases), they may start throttling clocks which reduces performance. Use `nvidia-smi dmon` or telemetry to see if clocks drop from their max. If you detect throttling, improve cooling, increase fan speeds, improve airflow, or slightly reduce power limit to keep within a stable thermal envelope. The goal is consistent performance without thermal-induced dips.

Energy-Aware Dynamic Power Management.

Modern data centers are increasingly using energy-aware scheduling to adjust workloads based on real-time energy costs and renewable energy availability. Incorporating adaptive power capping and dynamic clock scaling can help optimize throughput per watt while reducing operational costs and carbon footprint.

Optimize for Perf/Watt.

In multi-GPU deployments where power budget is constrained (or energy cost is high), consider tuning for efficiency. Many workloads, especially memory-bound ones, can run at slightly reduced GPU clocks with negligible performance loss but noticeably lower power draw. For example, if a kernel is memory-bound, locking the GPU at a lower clock can save power while not hurting runtime. This increases throughput per watt. Test a few power limits using `nvidia-smi -pl` to see if your throughput/Watt improves. For some models, going from 100% to 80% power limit yields nearly the same speed at 20% less power usage.

Use Adaptive Cooling Strategies.

If running in environments with variable cooling or energy availability, integrate with cluster management to adjust workloads. For instance, schedule heavy jobs during cooler times of the day or when renewable energy supply is high - if that's a factor for cost. Some sites implement policies to queue non-urgent jobs to run at night when electricity is cheaper. This doesn't change single-job performance but significantly cuts cost.

Consolidate Workloads.

Run GPUs at high utilization rather than many GPUs at low utilization. A busy GPU is more energy efficient in terms of work done per watt than an idle or lightly used

GPU. This is because the baseline power is better amortized when the GPU is busy. It may be better to run one job after another on one GPU at 90% utilization than two GPUs at 45% each in parallel - unless you need to optimize for the smallest wall-clock time. Plan scheduling to turn off or idle whole nodes when not in use, rather than leaving lots of hardware running at low utilization.

Fan and Cooling Config.

For air-cooled systems, consider setting GPU fans to a higher fixed speed during heavy runs to pre-emptively cool the GPUs. Some data centers always run fans at the maximum to improve consistency. Ensure inlet temps in the data center are within specifications. Every 1°C less in ambient can help GPUs maintain boost. Check for dust or obstructions in server GPUs periodically. Clogged fins can greatly reduce cooling efficiency. For water-cooled, ensure flow rates are optimal and water temperature is controlled.

Power Monitoring.

Use tools to monitor per-GPU power draw. nvidia-smi reports instantaneous draw which helps in understanding the power profile of your workload. Spikes in power might correlate with certain phases. For example, the all-reduce phase might measure less compute load and less power, while dense layers will spike the load and power measurements. Knowing this, you can potentially sequence workloads to smooth power draw. This is important if operating the cluster on a constrained power circuit. In the power-constrained scenario, you may need to avoid running multiple power-spikey jobs simultaneously on the same node to avoid tripping power limits.

Long-Running Job Resilience.

If you are running a months-long training job or 24x7 inference job, consider the impact of thermals on hardware longevity. Running at 100% power and thermal limit constantly can marginally increase failure risk over time. In practice, data center GPUs are built for this type of resiliency, but if you want to be extra safe, running at 90% power target can reduce component stress with minimal slowdown. It’s a trade-off of longer training runs vs. less wear on the hardware - especially if that hardware will be reused for multiple projects over a long period of time.

Conclusion

This list, while extensive, is not exhaustive. The field of AI systems performance engineering will continue to grow as hardware, software, and algorithms evolve. And not every best practice listed here applies to every situation. But, collectively, they cover the breadth of performance engineering scenarios for AI systems. These tips encapsulate much of the practical wisdom accumulated over years of optimizing AI system performance.

When tuning your AI system, you should systematically go through each of the relevant categories listed above and run through each of the items in the checklist. For example, you should ensure the OS is tuned, confirm GPU kernels are efficient, check that you're using libraries properly, monitor the data pipeline, optimize the training loop, tune the inference strategies, and scale out gracefully. By following these best practices, you can diagnose and resolve most performance issues and extract the maximum performance from your AI system.

And remember that before you scale up your cluster drastically, you should profile on a smaller number of nodes and identify potential scale bottlenecks. For example, if you see an all-reduce collective operation already taking 20% of an iteration on 8 GPUs, it will only get worse at a larger scale - especially as you exceed the capacity of a single compute node or data-center rack system such as the GB200 NVL72.

Keep this checklist handy and add to it as you discover new tricks. Combine these tips and best practices with the in-depth understanding from the earlier chapters, and you will design and run AI systems that are efficient, scalable, and efficient.

Now go forth and make your most ambitious ideas a reality. Happy optimizing!

About the Author

Chris Fregly is a passionate performance engineer and AI product leader with a proven track record of driving innovation at leading tech companies like Netflix, Databricks, and Amazon Web Services (AWS). He’s led performance-focused engineering teams that built advanced AI/ML products, scaled go-to-market initiatives, and reduced cost for large-scale generative AI and analytics workloads. He is also co-author of two O’Reilly books, *Data Science on AWS* and *Generative AI on AWS*, as well as the creator of the O’Reilly online course titled, “High Performance AI in Production with Nvidia GPUs.”