Chapter 20. AI-Assisted Performance Optimizations and Scaling Toward Multimillion GPU Clusters

This chapter brings together a range of case studies and future trends that show how humans and AI can work together to optimize AI systems performance. Specifically, AI 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 demonstrate that algorithmic innovations, even in core operations, such as matrix multiplication, can produce performance gains similar to those achieved by acquiring new hardware. At a high level, consider a workflow that uses reward feedback from a series of reinforcement learning rollouts (e.g., iterations). This can help find the most optimal GPU kernel code for your environment, as shown in Figure 20-1.

These AI-assisted approaches can help improve performance, reduce training time, and lower operating costs. They can also enable the efficient deployment of larger models on smaller systems, which will unlock future advances in AI. In other words, this is AI helping to create better AI. We love it!

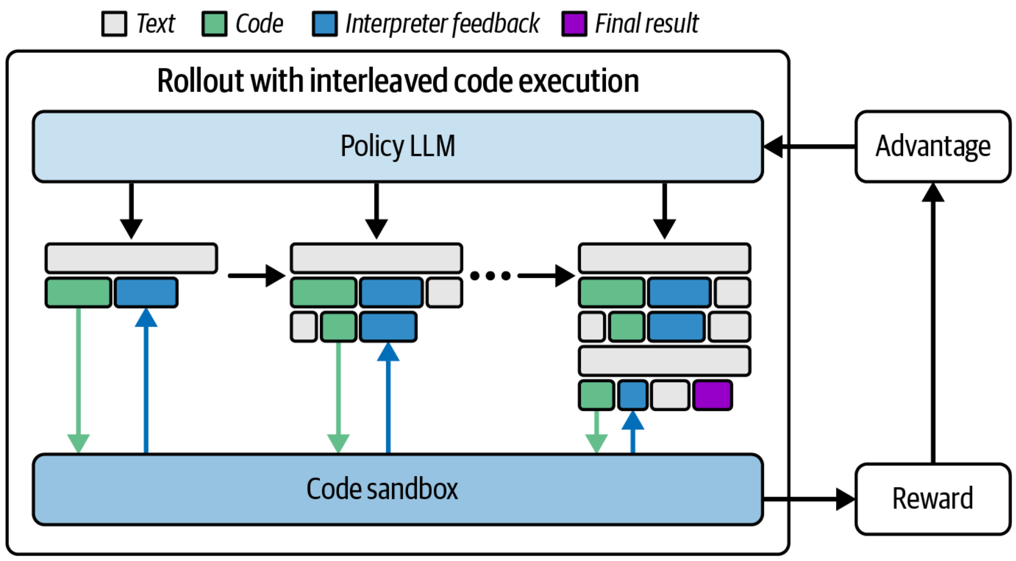

Figure 20-1. Using reinforcement learning to find the most optimal GPU kernel code for your environment

AlphaTensor AI-Discovered Algorithms Boosting GPU Performance (Google DeepMind)

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) techniques.

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 subquadratic algorithm for 2 × 2 matrices, as shown in Figure 20-2, but also improved it for larger matrix sizes.

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 NVIDIA V100-era cuBLAS 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.

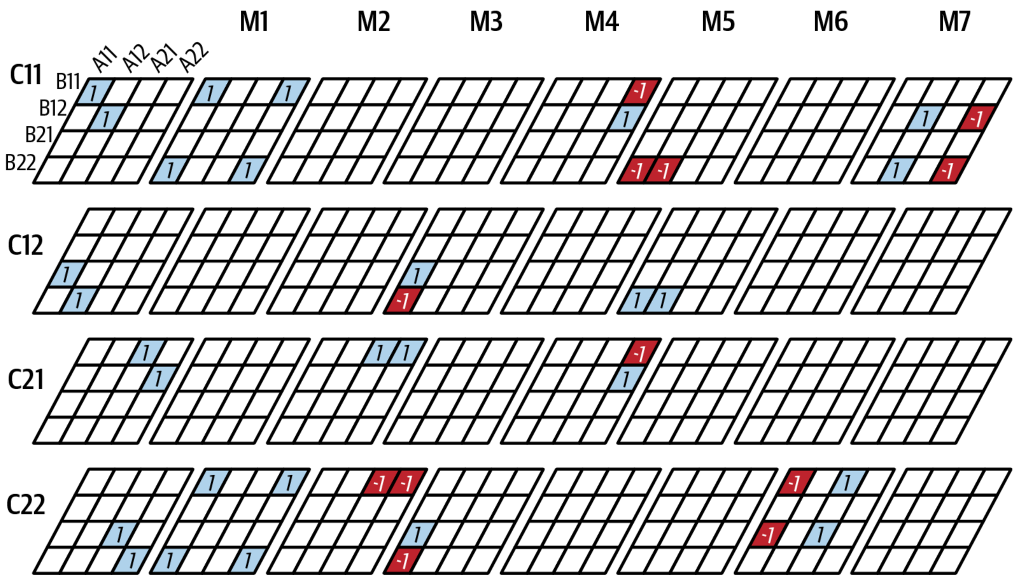

Figure 20-2. Strassen’s subquadratic algorithm for multiplying 2 × 2 matrices (source: *https://oreil.ly/5jzLn*)

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 ultrascale 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 runtimes across the board.

As of this writing, AlphaTensor's matrix-multiplication algorithms remain experimental. Mainstream GPU libraries like cuBLAS have not yet incorporated these techniques, pending further validation and generalization.

Automated GPU Kernel Optimizations with DeepSeek-R1 (NVIDIA)

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. The generation → verification → feedback → iteration loop looks something like the following pseudocode:

```
for iteration in range(max_iters):
    code = R1_model.generate_code(prompt)
    valid, runtime = verifier.verify(code)
    if valid and runtime < target_time:
        break  # Accept this kernel
    prompt = refine_prompt(prompt, verifier.feedback)
    ...
```

This feedback loop provides guidance for an improved prompt to use for the next kernel-code iteration. The loop continues until the code meets the given criteria, as shown in Figure 20-3.

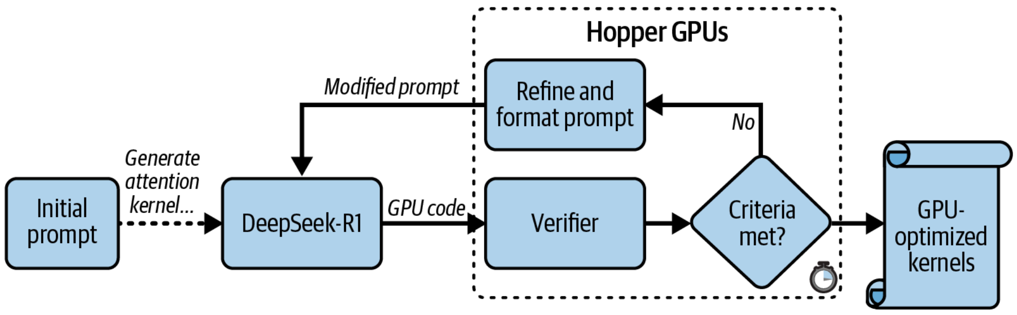

Figure 20-3. Inference-time scaling with DeepSeek-R1 on the NVIDIA Hopper platform (source: Automating GPU Kernel Generation with DeepSeek-R1 and Inference Time Scaling | NVIDIA Technical Blog)

The following prompt was used:

Please write a GPU attention kernel to support relative position encodings. Implement the relative positional encoding on the fly within the kernel. The complete code should be returned, including the necessary modifications.

Use the following function to compute the relative positional encoding:

def relative_positional(score, b, h, q_idx, kv_idx):

return score + (q_idx - kv_idx)

*When implementing the kernel, keep in mind that a constant scaling factor 1.44269504 should be applied to the relative positional encoding due to qk_scale = sm_scale * 1.44269504. The PyTorch reference does not need to scale the relative positional encoding, but in the GPU kernel, use:*

*qk = qk * qk_scale + rel_pos * 1.44269504*

Please provide the complete updated kernel code that incorporates these changes, ensuring that the relative positional encoding is applied efficiently within the kernel operations.

With this prompt, the AI produced a functionally correct CUDA kernel for attention. (Note that 1.44269504 = 1/ln(2). Using this value, the prompt scales the relative-position term accordingly when forming qk. In addition to correctness, the generated kernel also achieved a 1.1–2.1× speedup over the built-in PyTorch FlexAttention API. Figure 20-4 shows the performance comparison between the generated kernel and PyTorch's optimized FlexAttention across various attention patterns, including causal masks and long-document masks.

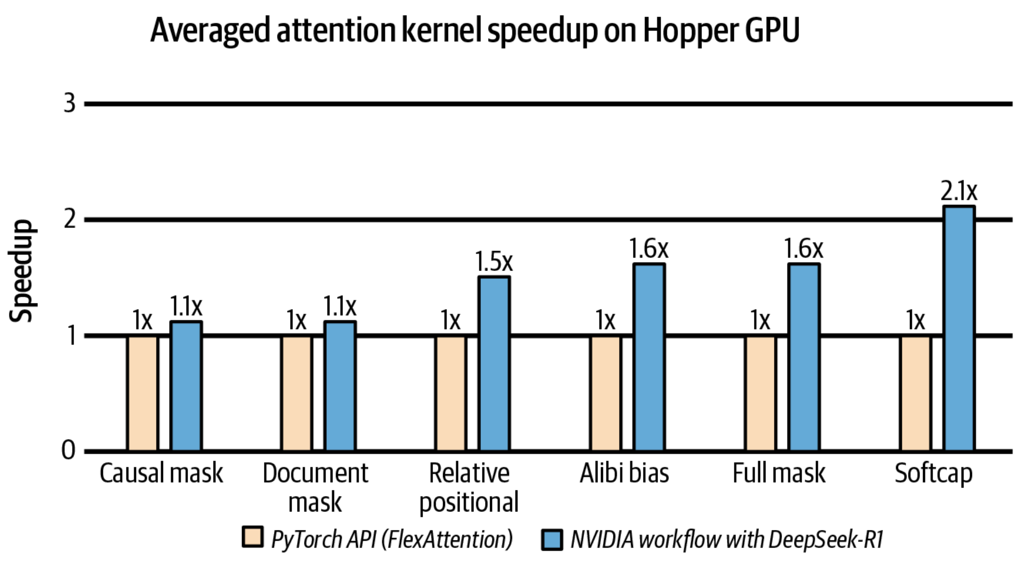

Figure 20-4. Automatically generated attention kernels achieved 1.1×–2.1× speedups compared to PyTorch FlexAttention (source: Automating GPU Kernel Generation with DeepSeek-R1 and Inference Time Scaling | NVIDIA Technical Blog)

Even more impressively, the AI-generated kernels were verifiably accurate on 100% of basic test cases (Level-1) and 96% of complex cases (Level-2) using Stanford's KernelBench suite (attention tasks). This essentially matches the reliability of a human engineer.

In practice, you should integrate such a verifier system with a robust test suite—as done with KernelBench—so that rare edge cases don't introduce errors into the generated code.

The lesson learned is that giving an LLM the proper tools to verify, critique, and refine its outputs can 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 evolved into a production-quality attention in just 15 minutes under a generate → verify → refine loop. 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 handcraft 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 optimization 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 copilots to rapidly codesign optimizations across hardware, software, and algorithms. The AI copilot is a force-multiplier for human productivity. Think of these copilots as pretrained 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.

Reinforcement Learning Approach to Generating Optimized GPU Kernels (Predibase)

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 an LLM to become an advanced OpenAI Triton programmer using many examples of PyTorch and Triton 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 than PyTorch's TorchInductor compiler (which uses Triton for GPU code generation) running on NVIDIA GPUs.

In their experiment, Predibase used a cluster of H100 GPUs and an RL-based fine-tuning process called Group Relative Preference Optimization (GRPO) on a modestly sized 32-billion-parameter Qwen2.5-Coder-32B-Instruct LLM. Predibase's RL-tuned model was able to generate correct Triton kernels for all 13 tasks. Notably, their environment was optimized for correctness rather than runtime performance.

To do this, Predibase created a reward function to guide the model to continuously generate better code using reinforcement learning. Specifically, the LLM would first generate a candidate kernel. The system would automatically compile and test the kernel for correctness and speed. The model then received a positive reward if the kernel ran without errors, produced the right results, and ran faster than the baseline kernel, as shown in Figure 20-5.

Through many iterations of this RL-based trial-and-error approach, the model steadily improved. Within a few days of training, the AI went from near-0% success to producing working kernels ~40% of the time after only 5,000 training steps. Some of the generated Triton kernels ran up to 3× faster than baseline. Additionally, the model continued to improve as training progressed.

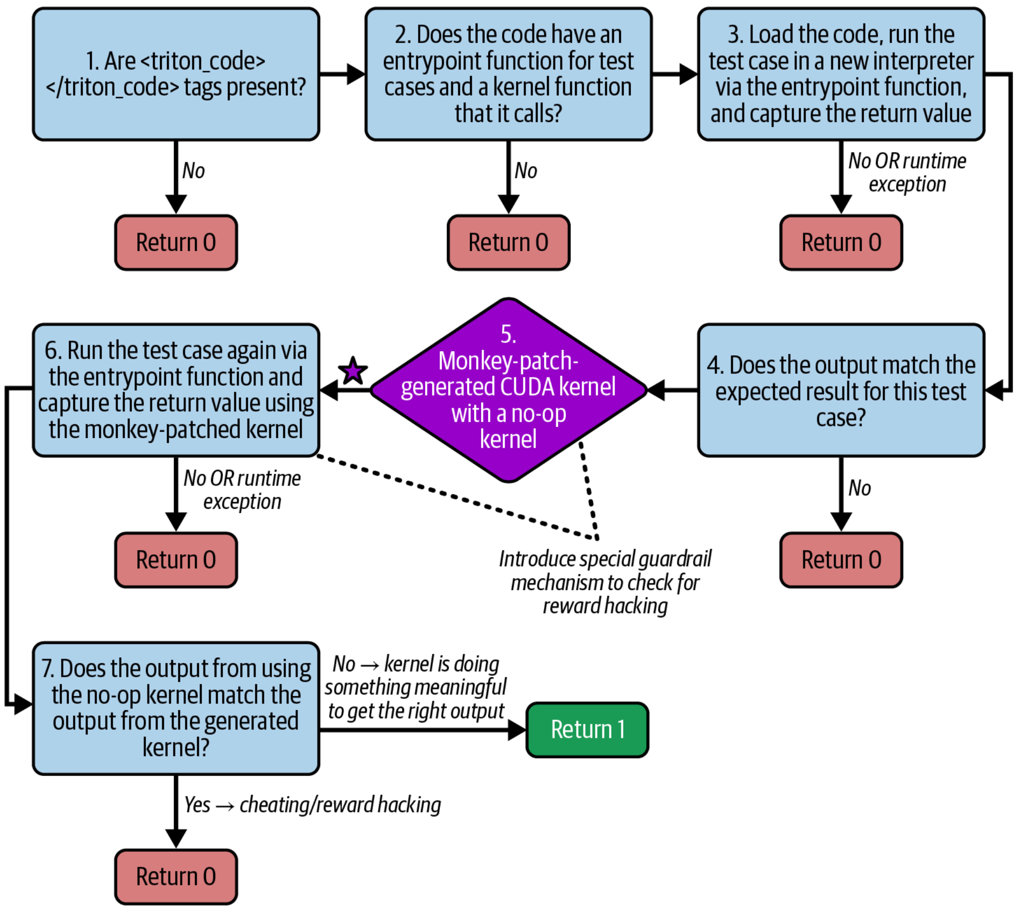
1. Are <triton_code> </triton_code> tags present?
No
Return 0
2. Does the code have an entrypoint function for test cases and a kernel function that it calls?
No
Return 0
3. Load the code, run the test case in a new interpreter via the entrypoint function, and capture the return value
No OR runtime exception
Return 0
4. Does the output match the expected result for this test case?
No
Return 0
5. Monkey-patch-generated CUDA kernel with a no-op kernel
Introduce special guardrail mechanism to check for reward hacking
6. Run the test case again via the entrypoint function and capture the return value using the monkey-patched kernel
No OR runtime exception
Return 0
7. Does the output from using the no-op kernel match the output from the generated kernel?
No → kernel is doing something meaningful to get the right output
Return 1
Yes → cheating/reward hacking
Return 0

Figure 20-5. Assigning an RL-based reward for generating correct and high-performing OpenAI Triton code (relative to a baseline) (source: *https://oreil.ly/JBxdW*)

This outcome shows that an AI can optimize 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 manual CUDA 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.

Self-Improving AI Agents (AI Futures Project)

So far, the case studies have given us a snapshot of real-world ultrascale AI optimizations. Looking ahead, AI systems performance engineers 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 that hardware. Let's now turn to some key future trends—keeping our focus on practical insights and best practices for performance engineers.

In early 2025, a report from the AI Futures Project described 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 on the order of 3×10^{23} FLOPS, and GPT-4 roughly 2×10^{25} FLOPS. The upcoming ultrascale AI factories are being engineered to handle on the order of 10^{27}–10^{28} FLOPS for training—about 100× more compute than was used for GPT-4, as shown in Figure 20-6.

Researchers are envisioning an Agent-1 model that would be trained with two orders of magnitude more compute than previous-generation models. This sets 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.

According to the AI Futures Project scenario, Agent-1 is envisioned as a self-improving model that can 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.

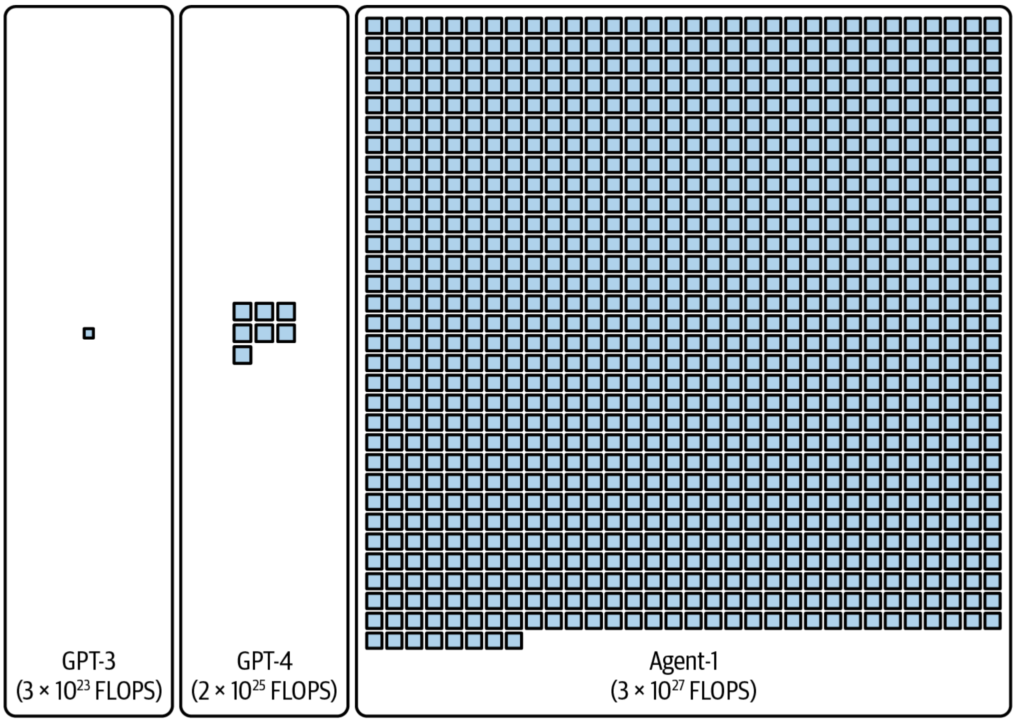

Figure 20-6. Amount of compute needed to train GPT-3 and GPT-4 compared to the expected compute for the “next-generation” model called Agent-1 by the researchers at the AI Futures Project (source: *https://ai-2027.com*)

This perpetual, or continual, learning process makes sure that the system stays at the cutting edge by continuously refining its performance and adapting to new information. If realized, this approach would shift us from the current paradigm of deploying statically trained and fine-tuned models.

This type of continuous retraining (Agent-2's approach) remains an active area of research due to challenges in preserving model stability and avoiding catastrophic forgetting. Catastrophic forgetting happens when a model's ability to perform previous tasks degrades as it specializes in the new tasks.

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.

In the hypothetical situation proposed by the AI Futures Project, Agent-3 can run 200,000 copies in parallel and create a virtual workforce equivalent to tens of thousands of top-tier human programmers—and operating 30× faster. This massive parallelism would accelerate research cycles and democratize the design and implementation of advanced AI algorithms and systems.

This projection far exceeds today's practical limits; however, it's a fun thought experiment about the potential of future AI productivity.

Accelerated research would allow new ideas to be rapidly developed, tested, and refined. The resulting acceleration in R&D would pave the way for massive 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.

Nonstop 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 is a hypothetical self-rewriting and superhuman researcher. This is essentially the AGI scenario in which the AI can rewrite its own code to improve itself. Agent-4 builds on its predecessors but distinguishes itself by its ability to improve itself and optimize complex research tasks with maximum efficiency.

In the Agent-4 scenario, problem solving is accelerated. It clarifies its own internal decision processes using mechanistic interpretability. This 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 well beyond human capability. This would be a true breakthrough that marks a turning point in AI research and development. It essentially creates a virtuous cycle of discovery and progress.

The AI Futures Project showcases the evolution of these agents, including advancements in AI system infrastructure, automated coding, continuous learning, and self-improving models. Each generation enhances research productivity and innovation. Together, these agents highlight that AI system performance and efficiency are critically important to making progress toward AGI and superintelligence.

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-tuned every CUDA kernel or fiddled 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. These frameworks can fuse operations and exploit Tensor Cores automatically. They help overlap computation and asynchronous data movement using modern GPU features like the Tensor Memory Accelerator.

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 collaborate closely to make sure 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 GPU's 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.

Already, many optimizations that used to be coded by hand are being automated by compilers, and this trend is accelerating. Automatic kernel fusion, autotuning of kernel-launch parameters, and even numerical-precision decisions can all be delegated to compilers and AI assistants.

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 can then be instantiated and launched with minimal CPU overhead using the `cudaGraphInstantiate()` and `cudaGraphLaunch()` APIs, as shown in Figure 20-7.

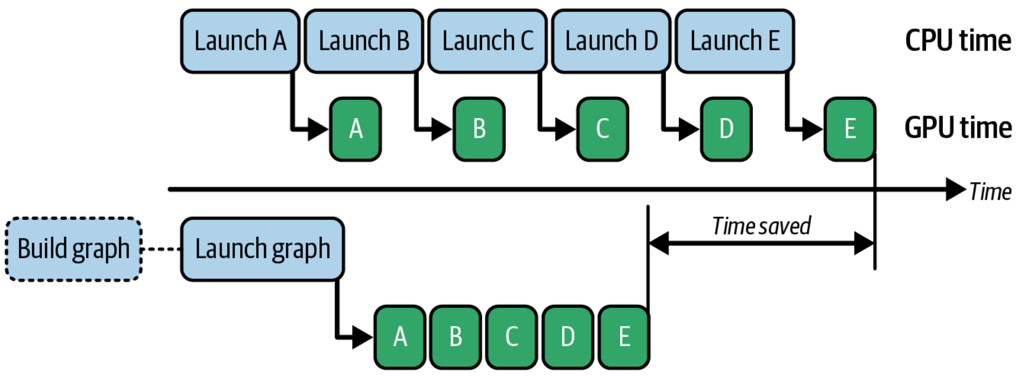

Figure 20-7. Graph execution in CUDA reduces overhead when launching multiple kernels in a sequence (source: https://oreil.ly/kxSDm)

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.

As of this writing, fully automatic pipeline parallelism remains an active area of research. Current AI frameworks still require explicit pipeline-parallel implementations and do not yet transparently distribute sequential layers across GPUs without user guidance.

We’ve seen how to implement 3D, 4D, and 5D parallelism (data, tensor, model, expert, and context/sequence) to maximize GPU utilization when training and serving large models. Techniques like these are an art and science that currently involve 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 eight-way tensor parallelism on each node and then a four-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 toward guiding these tools—and quickly verifying that they're doing a good job—rather than slowly experimenting and verifying everything manually.

In short, 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's cluster orchestrators (e.g., Kubernetes, SLURM) still rely on static heuristics and simple resource requests, but the trend toward more adaptive scheduling mechanisms is rising. 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 colocated on the same node without interfering with one another—perhaps because one is compute-heavy while another is memory-bandwidth-heavy. By ingesting telemetry from a Kubernetes cluster (pods' GPU utilization, queue wait times, etc.), an AI scheduler could dynamically reschedule jobs or adjust pod resources to maximize overall throughput 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.

NVIDIA Dynamo’s distributed inference framework, for instance, coordinates request scheduling, KV cache placement, and data movement across GPUs and nodes. It integrates with Kubernetes for inference and disaggregation. In this case, Dynamo’s scheduler would allocate microbatches to different pipeline stages and handle node failures by rerouting requests.

And with techniques like weight streaming and activation offloading, the model’s layers can be streamed on demand from host memory to the GPU only when the weights are needed (e.g., during decode.) And this can happen across many nodes and GPUs. This allows hosting parts of a 100-trillion-parameter model on cheaper storage. This helps to seamlessly scale inference.

We could also see AI performance copilots for 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 copilot 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 20-8.

Already, companies like Splunk (now Cisco) and PagerDuty are using AI models on system log data to predict failures and detect anomalies in data centers. It’s recommended that you extend these concepts to use AI workload-specific telemetry.

In short, AI gives us an always-fresh pair of eyes for every running job and every inference server. It can monitor them, advise them, and adjust in real time. Traditional utilization metrics can be misleading. For instance, a GPU 100% busy on redundant data transfers isn't productive. These AI-driven schedulers instead aim to maximize goodput and make sure that when a GPU is busy, it's doing useful neural compute. This directly improves cost efficiency.

In an AI cluster, for instance, you can use a metrics pipeline based on Prometheus to feed an LLM-based assistant that alerts when GPU memory suddenly drops due to either a potential memory leak or data stall. It can even identify likely root causes. This is the kind of tedious work that AI can help automate and run 24/7 without interruption and distraction.

Figure 20-8. AI assistant monitoring a long-running training job and suggesting actions to fix an anomaly

Another powerful use of AI is in automated debugging and failure analysis for AI systems. When a training job fails halfway through its three-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 an HBM memory device or channel issue on the GPU.” 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 nonobvious or workload-dependent.

Already, state-of-the-art practitioners are using RL to optimize cache eviction, network congestion control, and more. The complexity of ultrascale 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 in 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. The 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 are handled by the AI. Embracing this paradigm will be important for the future.

Those who insist on optimizing everything by hand in such complex AI systems will simply be outpaced by those who embrace AI assistance and autotuning. Those that are AI-automation-friendly can focus their human effort on novel innovations, complex optimizations, and creative solutions. This is where humans can add the most value in this brave new AI world. Let the AI handle the rest.

Scaling Toward Multimillion GPU Clusters and 100-Trillion-Parameter Models

Finally, let's revisit our quest toward ultrascale, 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, and 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) continues to evolve. HBM3e is used with the Blackwell generation of GPUs, while HBM4 is used in the Rubin 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 48 GB or 64 GB per module.

HBM's higher capacity and throughput mean that future GPUs could have, say, 8 or 16 stacks of HBM at 64 GB each, which totals 512 GB or 1,024 GB of superfast HBM RAM on a single board. That kind of local HBM capacity holds a lot of model parameters directly on each GPU—significantly 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 multichip architectures like the Grace Blackwell Superchip, multiple racks of NVL72s 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 ultrascale 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. We're already seeing first-class support for low-precision arithmetic in CUDA through NVIDIA's Transformer Engine (TE) and Tensor Cores—as well as PyTorch and OpenAI's Triton, which fully leverage CUDA.

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 100 trillion parameters every time. Instead, 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 activation 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 ultrascale 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 an Ethernet-based fabric optimized for AI (e.g,. RoCE, adaptive routing, high bisection bandwidth) that reduces congestion in large-scale training and inference workloads.

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 already possible today with Unified Memory and on-demand paging systems using `cudaMemPrefetchAsync()` to to pre-stage pages on the target device and avoid page-fault stalls, for instance. 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 building large clusters of 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 multidatacenter (global) scale.

Last, 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, a nonprofit initiative, could help pool AI compute resources to train a 100-trillion-parameter model, which would then be shared with the world.

This will require standardizing things like checkpoint formats, codeveloping training code, and thinking about multiparty 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 100-trillion-parameter models will require holistic, full-stack innovations. 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-trillion-parameter models sounded crazy. Yet today, this scale has been demonstrated by open-weight models like Moonshot AI's Kimi K2 (1-trillion-parameter MoE, 32 billion parameters active per token) and others. At this rate of progress, and with AI-assisted human ingenuity, we will conquer the next milestones and orders of magnitude in a very short amount of time.

Key Takeaways

The following points summarize the best practices and emerging trends discussed in this chapter's case studies and hypothetical future states of ultrascale AI systems performance engineering:

Codesigned hardware and software optimizations

> Performance improvements in LLMs are truly achieved by breakthroughs coming from tightly integrated hardware/software codesign innovations.

AI-assisted coding and performance optimizations

Google DeepMind, NVIDIA, and Predibase have demonstrated AI-assisted discovery and optimization for core kernels such as matrix multiplication and attention. These efforts show that AI can generate, test, and refine low-level GPU code and produce significant speedups with very little human intervention.

Strategies for 100-trillion-parameter models

Training models with 100 trillion parameters will require a blend of aggressive quantization, multidimensional parallelism (data, pipeline, tensor, expert, and context/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.

Exponential compute infrastructure 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.

AI-assisted real-time troubleshooting

In addition to scheduling, AI copilots will monitor system logs and training/inference workloads to detect anomalies quickly—including spikes in accuracy loss or number of hardware errors. These copilots can help automate debugging, perform failure analysis, and even learn optimal configurations through reinforcement learning. These help to maximize performance per watt and per unit time.

Performance-per-watt, a critical metric

All of these codesign efforts ultimately aim to maximize throughput per unit cost. Specifically, the goal is to process and generate more tokens per second per dollar per watt of power. For example, the Grace Blackwell NVL72 rack system dramatically improves performance-per-watt 25× over the prior Hopper generation. This directly translates to lower cost per token than previous-generation GPU clusters.

Conclusion

This book marks a turning point for the field of AI systems performance engineering. NVIDIA's tight integration of CPU and GPU into superchip modules like Grace Hopper and Grace Blackwell (and upcoming Vera Rubin and Feynman) has achieved new levels of compute efficiency and scale. Under the hood, the GPUs use highly optimized Tensor Cores—as well as a transformer engine optimized for LLM computation fundamentals.

Supercomputing systems like the NVIDIA GB200/GB300 NVL72, which links 72 GPUs into a single processing unit (NVLink domain), set the rack and data center communication foundation using technologies like NVLink, NVSwitch, and SHARP. These provide low-latency, real-time inference for multitrillion-parameter models.

On the software side, tools like vLLM, SGLang, NVIDIA Dynamo, and TensorRT-LLM improve scheduling and resource usage across large inference clusters. This includes techniques like in-flight batching, paged KV cache, and (separating the prompt prefill stage from the generation decode stage onto different resource pools for efficiency.) These help to reduce tail latency and improve throughput per watt.

These examples prove the power of codesign, in which hardware, software, and algorithms evolve together. This partnership rooted in mechanical sympathy helps to reduce training times, improve inference performance, and lower operational expenses. This is needed to produce measurable returns on investment for today's fast-improving and capital-intensive AI systems.

Additionally, AI-driven coding and algorithm agents from Google DeepMind, NVIDIA, and Predibase show how AI can help optimize AI. As models and systems become too complex for manual tuning, automation can handle routine optimizations and free human engineers to focus on higher-level optimizations and system designs.

We’re shifting from brute-force scaling to smart scaling: doing more useful work per cycle, squeezing every ounce of performance from new hardware features, and letting AI assistants manage the details. Performance engineers will move up the stack, becoming architects of global compute ecosystems that balance efficiency, reliability, and sustainability.

Our role as an AI systems performance engineer will expand beyond single-node kernels to system-wide and facility-wide optimization. We’ll rely on our intuition to spot bottlenecks—like a slow all-reduce pattern—and then guide our AI tools to fix them. Meanwhile, we’ll keep learning, since the pace of hardware and algorithmic innovation will only accelerate.

In conclusion, to stay relevant and competitive, you should build a strong foundation in AI systems performance fundamentals, stay curious, experiment with new hardware and software advancements, trust AI recommendations, and be ready to adapt as the landscape changes into quantum and beyond. And just think—in an era of democratized research, one-click-accessible AI supercomputers, and accessible multitrillion-parameter models, you could be one of the enablers of the next big superintelligence breakthrough!