# Appendix. 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, CPU-GPU superchips, networking gear, and rack systems.

# Performance Tuning and Cost Optimization Mindset

A pragmatic, documented loop—quick wins before deep work—turns engineering time into measurable ROI. Start by targeting the biggest runtime and cost drivers, and always profile before and after to verify impact.

Combine auto-tuning, framework upgrades, cloud pricing levers, and utilization dashboards for high-ROI wins, documenting results and favoring simple, maintainable fixes. Tune throughput-sensitive hyperparameters when accuracy allows. Here are some tips on the performance tuning and cost optimization mindset:

*Optimize the expensive first*

Use the 80/20 rule. Find the top contributors to runtime and focus on those. If 90% of the time is in a couple of kernels or a communication phase, it's better to optimize those deeply than to microoptimize something taking 1% of the time. Each chapter's techniques should be applied where they matter most. For example, if your training is 40% data loading, 50% GPU compute, and 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 activation checkpointing 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.

*Embrace adaptive autotuning feedback loops*

Implement advanced autotuning 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 are 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 versus the gain in runtime and cost saved. For large recurring jobs like training a flagship model, even a 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. Leverage these improvements, as they are essentially free gains. Read release notes for performance-related changes.

*Codesign collaboratively with vendors and community members*

Stay connected with hardware vendors and the broader performance engineering community to align software optimizations with the latest hardware architectures. This codesign 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 a fraction of the cost can deliver better price/performance if your workload doesn't need the absolute latest. Our discussions on H800 versus 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. Calculate metrics like throughput per dollar to guide your optimization decisions.

*Monitor utilization metrics*

Continuously monitor GPU utilization, SM efficiency, memory bandwidth usage, and, for multinode, 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 the 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 activation 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 and 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.

*Optimize AI-driven performance*

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

Performance wins don't stick unless they're reproducible, versioned, and continuously checked, or they'll regress quietly over time. Treat docs, CI benchmarks, and shared knowledge as the glue that preserves speedups and accelerates onboarding and audits.

Lock down versions, configs, and benchmarks in source control so experiments are repeatable and regressions traceable. Bring performance checks into CI/CD, instrument end-to-end monitoring and alerts, and pair optimization with security and thorough documentation to create a durable, auditable practice. The following is a list of tips to improve reproducibility and documentation:

*Rigorous version control*

Maintain comprehensive version control for all system configurations, framework/driver versions, OS settings, optimization scripts, and benchmarks. Use Git (or a similar system) to track changes and tag releases. This way, experiments can be reproduced exactly—and performance regressions can be 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, 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, compliance, and performance*

Integrate and codesign security, compliance, and performance. Regularly audit configurations, enforce access controls, and maintain industry-standard safeguards, including encryption, secure data channels, zero-trust networking, hardware security modules (HSMs), and secure enclaves. And make sure that performance tuning never compromises system security. Similarly, make sure security doesn’t incur unnecessary performance overhead.

*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

Your hardware, interconnects, and data paths set the ceiling for performance and cost-efficiency—no software tweak can outrun a starved GPU. Plan for goodput per dollar/watt by matching accelerators, CPU/DRAM/I/O, and cooling/power to the workload to avoid bottlenecks from the start.

Specifically, design for goodput—useful work per dollar/watt—and not just raw FLOPS. Match accelerators and interconnects to workload, right-size CPU/memory/I/O to keep GPUs fed, keep data local, and plan power/cooling so hardware sustains peak clocks. Evaluate scaling efficiency before adding more GPUs. Here are some tips for optimizing system architecture and improving hardware planning efficiency:

*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/watt—and not just raw FLOPS.

*Choose the right accelerator*

Prefer modern GPUs for superior performance-per-watt and memory capacity. Newer architectures offer features like native FP8 and FP4 precision support—along with much faster interconnects. These produce big speedups over older-generation GPUs and systems.

*Leverage high-bandwidth interconnects*

Use systems with NVLink/NVSwitch, such as GB200/GB300 NVL72, instead of PCIe-only connectivity for multi-GPU workloads. NVLink 5 provides up to 1.8 TB/s bidirectional GPU-to-GPU bandwidth (over 14× PCIe Gen5), enabling near-linear scaling across GPUs. NVLink Switch domains can be scaled with second-level switches to connect up to 576 GPUs in one NVLink domain. This enables hierarchical collectives that stay on NVLink as long as possible before falling back to the inter-rack fabric.

*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. Ideally, this is NVLink for intranode and intra-rack communication and InfiniBand for inter-rack 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.

*Choose an appropriate cluster size*

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/GB300 have very high TDP. Provide adequate cooling (likely liquid-based) and power provisioning so the GPUs can sustain boost clocks without throttling.

# Unified CPU-GPU “Superchip” Architecture

Unified memory and on-package links let you fit larger models and cut copy overhead when you place the right data in the right tier. Using Grace for preprocessing and HBM for “hot” tensors turns the superchip into a tightly coupled engine with fewer stalls.

On Grace Blackwell Superchips, treat CPU and GPU as a shared-memory complex. Keep hot weights/activations in HBM and overflow or infrequent data in Grace LPDDR via NVLink-C2C. Use the on-package Grace CPU for preprocessing/orchestration and prefetch or pipeline-managed memory to hide latency for ultralarge models. Take advantage of the superchip architecture as follows:

*Utilize unified CPU-GPU memory*

Exploit the Grace Blackwell (GB200/GB300) Superchip’s unified memory space. Two Blackwell GPUs and a 72-core Grace CPU share a coherent memory pool with NVLink-C2C (900 GB/s). Use the CPU’s large memory (e.g., 480 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 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.

*Take advantage of CPU-GPU direct memory access when available*

Use the GPU's ability to directly access CPU memory on combined CPU-GPU superchips like the GB200 and GB300. GPUs can read and write Grace LPDDR memory coherently over NVLink-C2C without staging over host PCIe. Bandwidth and latency are still lower than HBM, so prefetch managed pointers, stage data, and pipeline transfers to hide latency. As such, it's recommended to keep hot activations and KV cache in HBM and use CPU memory as a lower-tier cache with explicit prefetch.

*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 using NVLink-C2C, essentially acting as an extremely fast I/O and compute companion for the GPU.

*Plan for ultralarge models*

For trillion-parameter model training that exceeds GPU memory, GB200/GB300 systems allow you to train using CPU memory as part of the model's memory pool. Prefer framework caching allocators and use `cudaMallocAsync` in custom code to minimize fragmentation and enable graph capture. Use CUDA Unified Memory or managed memory APIs to handle overflow gracefully, and consider explicit prefetching (e.g., `cudaMemPrefetchAsync`) of upcoming layers from CPU → GPU memory to hide latency.

*Consider superchip-optimized algorithms*

SuperOffload is an example of a superchip-optimized set of algorithms focused on improving efficiency of offload and tensor cast/copy strategies. Innovations include speculation-then-validation (STV), heterogeneous optimizer computation, and an ARM-based CPU optimizer. Designed specifically for NVIDIA superchips (e.g., Grace Hopper, Grace Blackwell, Vera Rubin), SuperOffload increases token-processing throughput and chip utilization relative to traditional offload strategies.

# Multi-GPU Scaling and Interconnect Optimizations

Scaling pays only when communication is fast and topology-aware—otherwise added GPUs just wait on one another. Lean on NVLink/NVSwitch bandwidth, modern collectives, and fabric-aware placement to approach linear speedups.

Specifically, exploit NVLink/NVSwitch domains (e.g., NVL72) for near-linear scaling, and choose parallelism strategies that fit the fabric. Use topology-aware placement, updated NCCL collectives (e.g., PAT), and telemetry to verify you're using the ~1.8 TB/s bidirectional throughput per-GPU bandwidth effectively. Plan hierarchical communications as you expand. The following are tips on utilizing multi-GPU scaling through interconnect and topology optimizations:

*Design for high-speed all-to-all topology*

On NVL72 NVSwitch clusters with 72 fully interconnected GPUs, for example, any GPU can communicate with any other at full NVLink 5 speed. At the fabric level, the NVLink Switch domain is nonblocking. Application-level throughput can vary with concurrent traffic and path scheduling, so verify behavior using DCGM NVLink counters and Nsight Systems traces before assuming per-pair saturation. 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.

*Utilize topology-aware scheduling*

Always colocate 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 in each direction, which doubles the per-GPU bandwidth versus the previous generation.

An NVL72 rack provides ~130 TB/s total intra-rack bandwidth in aggregate. 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.

*Embrace modern collective algorithms*

Use the latest NVIDIA NCCL library optimized for NVSwitch. Specifically, enable the parallel aggregated tree (PAT) algorithm, which was introduced for NVLink Switch topologies. This further reduces synchronization time by taking advantage of the NVL72 topology to perform reductions more efficiently than other tree/ring algorithms.

*Consider 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 bidirectional throughput 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. DCGM exposes NVLink counters that can help verify link balance and detect hotspots during collectives.

*Plan for future expansion*

Be aware that NVLink Switch can scale beyond a single rack—up to 576 GPUs in one connected domain using second-level switches. If you operate at that ultrascale, 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.

*Identify opportunities for federated and distributed optimizations*

For deployments that span heterogeneous environments, such as multicloud 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

OS jitter, NUMA misses, and driver mismatches quietly drain throughput and create variability you can't tune around. Hardening the stack (huge pages, affinities, consistent CUDA/driver, persistence) creates a stable, high-performance baseline.

Run a lean, HPC-tuned Linux. Set NUMA/IRQ affinities and enable THP and high memlock. Keep NVIDIA drivers/CUDA consistent across nodes. Isolate system jitter, tune CPU libraries/storage, set container limits correctly, and keep BIOS/firmware/NVSwitch fabric up to date for predictable throughput. Here are some host, OS, and container optimizations that you should explore in your environment:

*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 a 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 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.

*Design for 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.

*Utilize 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` service daemon. This strategy minimizes cross-NUMA traffic and stabilizes performance under heavy loads.

*Enable transparent hugepages*

Turn on transparent hugepages (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.

Prefer THP in `madvise` mode if your workload is latency-critical and you observe jitter.

*Increase max locked memory*

Configure the OS to allow large pinned (aka 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 fallbacks to pageable memory, which would slow down GPU DMA.

*Use the latest 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. Make sure all nodes have the same driver/CUDA versions to avoid any mismatches in multinode 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. Update the NVIDIA driver and toolkit on all nodes to inherit bug fixes and performance improvements.

*Enable GPU persistence when using a MIG configuration*

With persistence mode enabled, the GPU remains "warm" and ready to use, reducing startup latency for jobs. This is especially crucial if using a Multi-Instance GPU (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 MIG.

*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 using CPU isolation or cgroup pinning. Eliminating OS jitter ensures consistent throughput.

*Optimize 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 filesystem read-ahead for

streaming reads. Ensure the disk scheduler is set appropriately to use `mq-deadline` or `noop` for NVMe SSDs to reduce latency variability.

*Perform regular maintenance*

Keep BIOS/firmware updated for performance fixes. Some BIOS updates improve PCIe bandwidth or fix input–output memory management unit (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.

*Tune Docker and Kubernetes configurations for maximum performance*

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.

# GPU Resource Management and Scheduling

Smarter placement and partitioning raise utilization without buying new hardware—and protect predictability for mixed workloads. Respect topology, use MPS/MIG where appropriate, and control clocks/power to minimize contention and tail latency.

Schedule with GPU/NUMA/NVLink topology in mind, and use MPS or MIG to raise utilization for smaller jobs while retaining ECC and persistence for reliability. Lock clocks or power limit for stability when needed, avoid CPU oversubscription, and pack jobs intelligently to maximize ROI without contention. Here are some GPU resource management and scheduling tips:

*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.

*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 slice a GPU to ensure guaranteed resources for each job. For instance, modern GPUs can be split into multiple MIG slices (up to 7). 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 repartitioning 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 resetting the GPU, which can disrupt running jobs. Plan for maintenance windows as MIG device partitions are not persisted by the GPU across reboot. Use NVIDIA's MIG Manager to automatically recreate the desired layout on boot.

*GPU clock and power settings*

Consider locking GPU clocks to a fixed high frequency with `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 multiweek training.

*Job 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.

*CPU oversubscription*

When scheduling jobs, account for the 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 NVIDIA Fabric Manager for NVSwitch*

On systems with NVSwitch, the GB200/GB300 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. The Fabric Manager service typically runs by default on NVSwitch-equipped servers, but you should double-check that it’s enabled and running—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

colocating jobs that collectively exceed the memory or I/O capacity of the node. The goal is high hardware utilization without contention.

# I/O Optimization

If data can't keep up, GPUs idle—often the largest, cheapest speedups come from fixing input, not math. Parallelism, pinned memory, async transfers, and fast storage ensure the model is continuously fed.

Keep the GPUs fed by parallelizing data loaders, using pinned memory and async transfers, and storing data on fast NVMe—preferably with GPUDirect Storage. Stripe, cache, and compress wisely. Measure end-to-end throughput so I/O scales with cluster size, and write checkpoints/logs asynchronously. Here are some tips on I/O optimizations for your data pipeline:

*Load data in parallel*

Use multiple workers/threads to load and preprocess data for the GPUs. The default of one to two 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.

*Pin host memory for I/O*

Enable pinned (aka 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 nonblocking `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 allows 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. If available, enable GPUDirect Storage (GDS) so that GPUs can stream data directly from NVMe or network storage—bypassing the CPU. This further reduces I/O latency and CPU load when reading large datasets. 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.

*Tune 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` or `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.

*Use client-side caching when available*

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 the 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 the CPU becomes the bottleneck due to decompression, consider multithreaded decompression or offloading to accelerators. Also, overlap decompression with reading by using one thread to read compressed data and another thread to decompress the data in parallel. Modern GPUs can perform on-the-fly data decompression using GPU computing resources (or specialized decoders for image/visual data) when paired with GPUDirect Storage and the `cuFile` I/O stack.

*Measure throughput and 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, increasing network buffer sizes, optimizing 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 multinode*

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 preloading 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.

*Minimize checkpointing and logging overhead*

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.

You can also suspend a running GPU process with `cuda-checkpoint` and Checkpoint/Restore in Userspace (CRIU) to persist the process image. When ready to resume, the CUDA driver can restore device memory and CUDA state—even on to other GPUs of the same device type. Treat this as complementary to your model's state-dict or sharded checkpoint files rather than a replacement.

# Data Processing Pipelines

The format, layout, and locality of data determine how smoothly the pipeline runs at scale. Binary formats, sharding, caching, and prioritized threads turn I/O from a bottleneck into a steady stream.

Convert datasets to binary or memory-mapped formats, shard across storage and nodes, and raise thread priorities or move simple augments to the GPU to prevent stalls. Cache hot data/KV states, prefetch and buffer aggressively, and size batches to keep the pipeline smooth from disk to device. The following are tips for improving your data processing:

*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.

*Tune the file system*

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 hyperthreading for CPU-bound workloads*

For data pipelines that are heavily CPU-bound, disabling hyperthreading 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.

*Cache 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`. For distributed training, use `DistributedSampler` to ensure each process gets unique data to avoid redundant I/O.

*Parallelize data transformations*

If CPU preprocessing—such as image augmentation and text tokenization—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 model states and outputs*

When inferencing with LLMs, it's beneficial to cache the embeddings and V 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.

*Shard data across nodes*

In multinode 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 parallelism since each node processes its own data shard. DeepSeek's Fire-Flyer File System (3FS) is one example of a distributed dataset sharding filesystem. DeepSeek's 3FS achieves multiterabyte-per-second throughput by distributing dataset shards across NVMe SSDs on each node—while minimizing traditional caching. This design feeds each GPU with local high-speed data, avoiding I/O bottlenecks.

*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.

*Apply 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. Utilize GPU-accelerated libraries like NVIDIA DALI to perform these tasks asynchronously. This helps maintain a smooth and high-throughput data pipeline.

*Focus on end-to-end throughput (e.g., tokens per second)*

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 Nsight Systems and Nsight Compute to measure kernel timelines and stalls, or the PyTorch profiler for framework-level attribution. Then compare iteration time with synthetic versus real data to see how much overhead data loading introduces. Aim for less than 10% overhead from ideal. If it's more than that, invest time in pipeline optimization; it often yields large "free" speedups in training.

# Performance Profiling, Debugging, and Monitoring

You can't optimize what you don't measure; profiling reveals if you're compute-bound, memory-bound, I/O-bound, or network-bound so you target the right fix. Continuous telemetry and regression tests keep wins from eroding as code, drivers, and data evolve.

Specifically, use Nsight Systems/Compute and framework profilers with NVTX to determine whether you're compute-bound, memory-bound, I/O-bound, or communication-bound. Trim Python overhead, watch utilization gaps, balance work across ranks, track memory/network/disk health, and gate changes with performance regression tests and alerts. Use the following guidance to profile, monitor, and debug the performance of your AI workloads:

*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 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. Combine with machine-learning–driven analytics to predict and preempt performance bottlenecks. This can help in automating fine-tuning adjustments in real time. When using GPUDirect Storage, enable GDS tracing to correlate `cuFile` activity with kernel gaps.

*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 using 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 the Nsight Systems or Perfetto timeline and help you attribute GPU idle times or latencies to specific parts of the pipeline. This makes it easier to communicate to developers which part of the code needs attention. For PyTorch, you can use `torch.profiler.record_function()` .

*Utilize kernel profiling and analysis tools beyond just the PyTorch profiler*

For performance-critical kernels, use Nsight Compute to examine kernel-level metrics like occupancy and throughput, or Nsight Systems to analyze GPU/CPU timelines and overlap. 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 intrawarp 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 nonzero ranks share the I/O and logging responsibilities. Ensuring that all GPUs/ranks have similar workloads 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 Systems' GPU memory trace 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.

*Monitor network and disk*

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 12.5 GB/s (12.5 GB/s = 100 Gb/s ÷ 8 bits per byte) 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

Aligning kernels with the memory hierarchy and hardware features is where large, durable gains come from. Fusion, Tensor Cores, CUDA Graphs, and compiler paths (e.g., `torch.compile` and OpenAI's Triton) convert launch overhead into useful math.

Optimize for the memory hierarchy: coalesce global loads, tile into shared memory, manage registers/occupancy, and overlap transfers (e.g., `cp.async` /TMA) with compute. Prefer tuned libraries and CUDA Graphs, leverage `torch.compile` and OpenAI's Triton for fusion, and validate scalability with roofline analysis and PTX/SASS inspection. The following are some GPU and CUDA programming optimization tips and techniques:

*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.

*Monitor 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*

Overlap memory transfers with computation whenever possible. Use `cudaMemcpyAsync` in multiple CUDA streams to prefetch while kernels run. Prefer the Tensor Memory Accelerator for bulk movement to shared memory, and use `cp.async` for fine-grained staged copies and prefetch. These approaches effectively mask global memory latency by overlapping data transfers with computation, making sure the GPU cores remain fully utilized without waiting for memory operations to complete.

*Use bulk prefetching when possible*

For predictable patterns, prefetch into L2 using the PTX `cp.async.bulk.prefetch.tensor.[1-5]d.L2.global*` (or the `prefetch.global.L2` family), and use TMA (e.g., `cp.async.bulk.tensor`) to stage blocks into shared memory. You can also use cp.async to stage global memory into shared memory asynchronously and overlap copy with compute. You can also explicitly load data into registers ahead of use. These proactive methods reduce the delay caused by global memory accesses and make sure 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.

*Utilize 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 the 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 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, and NCCL, for core math and collective operations. For point-to-point GPU data movement in distributed inference, use NIXL where available. You can also use NVSHMEM when you need fine-grained GPU-initiated transfers. 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. AI frameworks like PyTorch (and its compiler) use these optimized libraries under the hood.

*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.

*Use the PyTorch compiler*

Take advantage of PyTorch's `torch.compile` to fuse Python-level operations into optimized kernels through TorchInductor. The compiler can also reduce launch overhead by integrating CUDA Graphs. Typical gains of about 10%–40% are common once the optimizations are warmed up. This eliminates interpreter overhead and unlocks compiler-level optimizations.

In practice, enabling `torch.compile` has produced substantial speedups (e.g., 20%–50% on many models) by automatically combining kernels and utilizing NVIDIA GPU hardware (e.g., Tensor Cores) more efficiently. Always test compiled mode on your model. While it can massively boost throughput, you should ensure compatibility and correctness before deploying. When graphs are stable, enable CUDA Graphs to reduce per-iteration CPU overhead. Keep static memory pools to satisfy pointer-stability constraints.

*Plan for dynamic shapes*

If your input sizes vary, use `torch._dynamo.mark_dynamic()` to annotate dynamic dimensions or export shape-polymorphic graphs with `torch.export()`, and then compile. Control recompilation behavior with `torch.compiler.set_stance()` using `"fail_on_recompile"` and `torch._dynamo.error_on_`

`graph_break()` to surface problematic shape churn in testing and CI. Use static shapes where possible to enable CUDA Graphs to reduce per-iteration CPU overhead.

*Leverage Triton kernels using* `torch.compile`

If PyTorch doesn't fuse an operation well, consider writing a custom GPU kernel in Triton and integrating it. PyTorch makes it easy to register a custom GPU kernel with `torch.library.triton_op` .

*Use autotuning when available*

Enable library autotuning features to maximize low-level performance. For example, set `torch.backends.cudnn.benchmark=True` when input sizes are fixed. This lets NVIDIA's cuDNN library try multiple convolution algorithms and pick the fastest one for your hardware. The one-time overhead leads to optimized kernels that can accelerate training and inference. If exact reproducibility isn't required, allow nondeterministic algorithms by disabling `cudnn.deterministic` to unlock these faster implementations.

*Leverage the read-only path*

Mark frequently used constants or coefficients as read-only so the GPU can cache them in the dedicated L1 read-only cache. In CUDA C++, you can use `const __restrict__` pointers to hint that data is immutable. On modern GPU architectures, the compiler generates cached global loads for `const __restrict__` qualified pointers. When using AI frameworks and libraries, make sure that lookup tables or static weights are on the device and treated as constant. This optimization reduces global memory traffic and latency for those values, as each SM can quickly fetch them from cache instead of repeatedly accessing slow DRAM.

# Kernel Scheduling and Execution Optimizations

Launch overhead and unnecessary syncs create idle gaps that crush throughput. Fusing small kernels and using persistent/dynamic strategies keeps the device busy and latency hidden.

Keep the device busy by minimizing synchronizations, fusing small kernels, and using persistent kernels when launching the same work repeatedly. For irregular tasks, consider GPU dynamic parallelism—but use it judiciously to avoid adding overhead. The following are tips on improving kernel scheduling and execution:

*Minimize GPU synchronization calls*

Avoid unnecessary global synchronizations that stall GPU progress. Excessive use of `cudaDeviceSynchronize()` or blocking GPU operations (like synchronous memory copies) will insert idle gaps where neither the CPU nor GPU can do useful work. Synchronize only when absolutely needed. For instance, synchronize when transferring final results or when debugging. By letting asynchronous operations queue up, you keep the GPU busy and the CPU free to prepare further work. This leads to a more continuous execution pipeline.

*Fuse small kernels to amortize launch overhead*

If you have many tiny GPU kernels launching back-to-back, consider merging their operations to run in a single kernel where possible. Every kernel launch has a fixed cost on the order of tens of microseconds, so combining operations through manual CUDA kernel fusion, XLA fusion, or tools like NVIDIA CUTLASS/Triton for custom ops can improve throughput. Fused kernels spend more time doing actual work and less time in launch overhead or memory round trips. This is especially helpful in inference or preprocessing pipelines where chains of elementwise ops can be executed in one go. Try `torch.compile(mode="reduce-overhead")` first. The compiler can fuse operation chains and wrap steady regions in CUDA Graphs. This will reduce CPU launch overhead. For unfused hotspots, consider migrating them to Triton kernels and using asynchronous TMA and automatic warp specialization where applicable.

*Utilize GPU dynamic parallelism for intra-GPU work scheduling*

Utilize CUDA's Dynamic Parallelism to let GPU kernels launch other kernels from the GPU without returning to the CPU. In scenarios with unpredictable or iterative work, such as an algorithm that needs to spawn additional tasks based on intermediate results, dynamic parallelism cuts latency by removing the CPU launch bottleneck. For example, a parent kernel can divide and launch child kernels for

further processing directly on the device. This keeps the entire workflow on the GPU, avoiding CPU intervention and enabling better overlap and utilization. Use this judiciously, however, as it can introduce its own overhead if overused.

*Use persistent kernels for repeated workloads*

Use a persistent kernel strategy when a workload involves launching identical kernels in rapid succession, such as processing a work queue or streaming batches with the same computation. A persistent kernel is launched once and remains active, reusing threads to handle many units of work in a loop, rather than launching a fresh kernel for each unit. This approach trades a more complex kernel design for significantly lower scheduling overhead. By keeping the kernel alive, you avoid repeated launch costs and can achieve higher sustained occupancy. High-performance distributed training and inference systems often employ this technique to maximize throughput and minimize latency for iterative tasks.

*Evaluate thread block clusters*

Thread block clusters to keep data close and reduce relaunch overheads. Up to 16 thread blocks can form a cluster on Blackwell (after increasing the non-portable limit). Use cluster-aware synchronization and shared-memory residency to improve locality in persistent-style designs. Profile occupancy vs. residency trade-offs with kernel-level profiling tools like Nsight Compute.

# Arithmetic Optimizations and Reduced/Mixed Precision

Lower precisions and sparsity let you trade bits for big speed and memory wins—often with negligible accuracy impact. Mixed precision, TF32/FP8/INT8, and fused scaling exploit hardware math paths to raise throughput per dollar.

Specifically, use mixed precision (BF16/FP16) and Tensor Cores for big gains, adopt TF32 for easy FP32 speedups, and evaluate FP8/FP4 where quality allows. Exploit structured sparsity, lower-precision gradients/communications, and INT8/INT4 quantization for inference—fusing

scales/activations to preserve accuracy. The following optimization techniques apply to improving the performance of arithmetic computations and utilizing reduced/mixed precision:

*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 massively accelerate FP16/BF16 matrix operations. 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 about a 1.5–3.5× speedup (depending on the model and kernel mix, with larger gains on `matmul` -heavy workloads) with minimal accuracy loss and is now standard in most frameworks with automatic mixed precision (AMP).

*Embrace gradient accumulation and activation 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. Modern GPUs 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, novel precisions, and scaling techniques*

On modern GPUs, FP8 Tensor Cores provide roughly double the math throughput of FP16 or BF16 on compute-bound kernels while, at the same time, reducing activation and weight bandwidth. Additionally, FP4 (NVFP4) Tensor Cores double the throughput of FP8 and are used for inference with micro tensor scaling (an error-correction technique to maintain accuracy) to raise token throughput. For training, use FP8 with the NVIDIA Transformer Engine and maintain FP16 or FP32 accumulators when required. For inference, evaluate FP8 first and

adopt NVFP4 only after calibration shows acceptable quality for your task. It's recommended to use hybrid FP8 (E4M3 for forward activations/weights and E5M2 for gradients) for training. Specifically, consider using E4M3 for the forward pass (e.g., activations and weights) and E5M2 for the backward pass (e.g., gradients). It's often beneficial to use a delayed scaling window of 256–1024. For inference, consider NVFP4 after calibration. TE integrates with PyTorch and is supported by modern GPU hardware. Prefer framework TE kernels over ad-hoc FP8 custom operations. End-to-end speedup depends on kernel mix, memory bandwidth, and calibration, so validate accuracy and performance on your model and workload.

*Leverage Tensor Cores and the Tensor Memory Accelerator (TMA)*

Make sure your custom CUDA kernels utilize Tensor Cores for matrix ops if possible. This might involve using CUTLASS templates for simplicity. By using Tensor Cores and TMA for asynchronous tensor movement to shared memory, 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*

For 32-bit matrix multiplies, set `torch.set_float32_matmul_precision("high")` to enable TF32 (fast FP32) for operations that are numerically safe in PyTorch. Libraries like cuBLAS and cuDNN will automatically pick optimal Tensor Core code paths on modern GPU hardware. If you force full-precision FP32 with "highest" (instead of "high"), make sure to understand the performance impact.

*Exploit structured sparsity*

Modern NVIDIA 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.

*Reduce precision for gradients and activations when possible*

Even if you keep weights at higher precision, consider compressing gradients or activations to lower precision. For instance, use FP16/BF16 or FP8 communication for gradients. Many frameworks support FP16 gradient all-reduce. Similarly, for activation checkpointing, storing activations in 16-bit instead of FP32 saves memory. Research continues on FP8 and FP4 optimizers and quantized gradients. These help maintain model quality while reducing memory and bandwidth costs. In bandwidth-limited environments, gradient compression in particular can be a game changer. DeepSeek demonstrated this by compressing gradients to train on constrained GPUs.

*Use 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 the newest GPUs, also explore and evaluate FP8 or INT4 for certain models to further boost throughput for inference.

*Fuse scaling and computing operations when possible*

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 them from scratch.

# Advanced Tuning Strategies and Algorithmic Tricks

Algorithmic shifts routinely beat hardware upgrades on ROI by reducing work rather than pushing it faster. Autotuning, FlashAttention, overlap of comm/compute, and sharding unlock scale while cutting waste.

Specifically, autotune kernel and layer parameters, swap in fused/FlashAttention kernels, and overlap communication with computation in distributed training. Scale deep models with pipeline/tensor parallelism and ZeRO sharding, and consider asynchronous updates or pruning/sparsity to trade a little accuracy work for big throughput wins. The following are some advanced performance optimizations and algorithmic tricks:

*Autotune kernel parameters*

Autotune your custom CUDA kernels for the target GPU. Choosing the correct block size, tile size, unroll factors, etc., can affect performance, and the optimal settings often differ between GPUs' generations, such as Ampere, Hopper, Blackwell, and beyond. Use autotuning scripts or frameworks like OpenAI Triton—or even brute-force search in a preprocessing step—to find the best launch config. This can easily yield 20%–30% improvements that you'd miss with static "reasonable" settings. Use Triton features in your autotuning loop—for instance, set `num_warps` and `num_stages`, enable automatic warp specialization, and test asynchronous TMA layouts. Prefer tensor map descriptor APIs for shared-memory staging. Re-benchmark tile shapes when migrating to different hardware, as optimal choices will differ across GPU generations.

*Use 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 fused multihead attention implementations and fused normalization kernels. NVIDIA's libraries and some open source projects like Transformer Engine and FasterTransformer provide fused operations for common patterns, such as fused LayerNorm + dropout. These reduce launch overhead and use memory more efficiently.

*Utilize memory-efficient attention like FlashAttention*

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.

*Overlap communication and 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 the backward pass. This pipelining can hide all-reduce latency entirely if done right. Use asynchronous NCCL calls or framework libraries like PyTorch's Distributed Data Parallel (DDP), which provide overlapping out of the box. This ensures the GPU isn't idle waiting for the network.

*Use pipeline parallelism for deep models*

When model size forces you to pipeline across GPUs using tensor parallelism or pipeline parallelism, you can use enough microbatches 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 multiterabyte speeds, minimizing pipeline stalls.

*Utilize distributed optimizer sharding*

Use a memory-saving optimization strategy like Zero Redundancy Optimizer (ZeRO), which shards tensors like 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.

*Train asynchronously when possible*

If applicable, consider asynchronous updates. For example, you can use stale stochastic gradient descent (SGD) in which workers don't always wait for one another to share updates. This approach can increase throughput, though it may require careful tuning to not impact convergence. Asynchronous training can provide large performance benefits if done properly.

*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 prune only for inference, a smaller model will run faster and use less memory. This increases cost-efficiency for model deployments. Explore the lottery ticket hypothesis, distillation, or structured pruning to maintain accuracy while trimming model size.

# Distributed Training and Network Optimization

At cluster scale, the network becomes the limiter. Untreated, the network can break linear scaling and inflate costs. RDMA/Jumbo frames, hierarchical collectives, affinity, and compression protect bandwidth and tame latency.

Use RDMA (InfiniBand/RoCE) when available; if on Ethernet, tune TCP buffers, enable jumbo frames, and select modern congestion control. Align NIC/CPU affinity, adjust NCCL threads/buffers (and SHARP/CollNet where supported), compress or accumulate gradients, and test the fabric to catch loss or misconfigurations. Follow this guidance to optimize your network for distributed environments such as multi-GPU and multinode model training:

*Use RDMA networking when available*

Equip your multinode cluster with InfiniBand or RoCE for low latency and high throughput. Ensure NCCL and MPI are using RDMA for training. NCCL will autodetect InfiniBand and use GPUDirect RDMA if available. RDMA bypasses the kernel networking stack and can reduce latency significantly versus traditional TCP. If you only have Ethernet, enable RoCE on RDMA-capable NICs to get RDMA-like performance. On NVLink domain systems (NVL72, GB200/GB300, etc.), keep collectives on-fabric when possible. Reserve host networking for inter-island links. Align NCCL topology hints with your NVLink/NVSwitch domains.

*Tune the 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 10/40/100 GbE 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.

*Assign CPU affinity for NICs*

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.

*Optimize NCCL environment variables for your environment*

Experiment with NCCL parameters for large multinode 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 1 MB to 4 MB or more for better throughput on large messages. If your cluster uses SHARP-capable switches, install the NCCL SHARP plugin and enable CollNet by setting `NCCL_COLLNET_ENABLE=1` , then use the

SHARP plugin variables such as `SHARP_COLL_LOCK_ON_COMM_INIT=1` and `SHARP_COLL_NUM_COLL_GROUP_RESOURCE_ALLOC_THRESHOLD=0` as documented. Expect speedups only when your reductions are large enough and the network fabric supports SHARP offload.

*Use gradient accumulation for slow networks*

If your network becomes the bottleneck because you are scaling too many nodes linked by a moderate-performance interconnect, use gradient accumulation to perform fewer, larger all-reduce operations. Accumulate gradients over a few minibatches 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.

*Optimize all-reduce topologies*

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 intranode and then internode.

*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 more than 200 Gbps of traffic during all-reduce, so having only a single 100 Gbps NIC will constrain you. Consider multiple NICs per node and 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 internode 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, and 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.

# Efficient Inference and Serving

Serving is a cost-and-latency game—utilization rises through orchestration and batching, not just bigger GPUs. Specialized runtimes, KV cache strategies, and warmups keep throughput high without violating SLOs.

Orchestrate for demand with autoscaling, microservices, and dynamic/continuous batching to keep GPUs hot without violating latency SLOs. Use specialized runtimes (vLLM, SGLang, TensorRT-LLM), exploit NIXL and KV cache offloading for disaggregated serving, warm models, and isolate resources to control tail latency. Follow these techniques to improve model inference efficiency and performance:

*Orchestrate dynamic resources efficiently*

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.

*Embrace serverless architectures 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. For inference workloads, favor dynamic or continuous batching to automatically batch incoming requests. Larger batch sizes improve throughput by keeping the GPU busy, but too large can add latency. 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 SMs and Tensor Cores fully.

*Leverage NIXL for distributed inference*

When serving large models across GPUs or nodes, use the NVIDIA Inference Xfer Library to stream KV cache between prefill and decode workers over RDMA. In the case of NIXL, the large transformer-based 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 a disaggregated LLM inference cluster. It does this using GPUDirect RDMA and optimal paths—and without involving the CPU. This reduces tail latency for disaggregated prefill decode serving across nodes.

*Offload KV cache if necessary*

If an LLM's attention KV cache grows beyond GPU memory, use hierarchical offloading. NVIDIA Dynamo's Distributed KV Cache Manager offloads less frequently accessed KV pages to CPU memory, SSD, or networked storage, while inference engines like TensorRT-LLM and vLLM support paged and quantized KV caches. Reuse caches to lower memory pressure and first-token latency. Validate end-to-end impact because offloaded misses introduce extra I/O latency. 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.

*Serve models efficiently*

Use optimized model inference systems, such as vLLM, SGLang, NVIDIA Dynamo, and NVIDIA TensorRT-LLM for serving large models with low latency and high throughput. They should implement quantization, low-precision formats, fusion, highly optimized attention kernels, and other tricks to maximize GPU utilization during inference. These libraries should also handle tensor parallelism, pipeline parallelism, expert parallelism, context 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.

*Warm up to avoid cold-start latency*

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.

*Partition resources efficiently for quality of service (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.

*Tune carefully for 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 ultralow latency for time-sensitive applications.

# Multinode Inference and Serving

Disaggregating prefill/decode and sharding models lets you handle bigger contexts and more users with higher occupancy. Continuous batching and hierarchical memory/offload maintain flow even under long prompts and heavy concurrency.

Specifically, disaggregate prefill and decode across devices, continuously pool tokens across requests, and shard oversized models via tensor/pipeline parallelism. Add hierarchical memory/offload for very long contexts so you serve more without OOMs, trading small latency for much higher capacity. The following performance tips apply to multinode inference and serving:

*Disaggregate inference pipelines*

Separate the inference workflow into distinct phases, including the “prefill” phase that processes the input prompt through all model layers, and the iterative “decode” phase that generates outputs token by token. Allocate these phases to different resources to allow for independent scaling. This two-stage approach prevents faster tasks from being bottlenecked by slower ones. For large language models, one strategy is to run the full model to encode the prompt, then handle autoregressive decoding on a stage-wise basis, possibly with

specialized workers for each phase. By disaggregating the pipeline, you ensure that GPUs continuously work on the portion of the task they're most efficient at, avoiding head-of-line blocking, where one long generation stalls others behind it.

*Use continuous batch processing for LLMs*

Move beyond simple request batching and use continuous batching strategies to maximize throughput under heavy loads. Traditional dynamic batching groups incoming requests and processes them as a batch to improve GPU utilization. Continuous batching takes this further by dynamically merging and splitting sequences of tokens across requests in real time. Systems like vLLM implement token pooling, where as soon as any thread is ready to generate the next token, it gets grouped with other ready threads to form a new batch. This approach keeps the GPU at high occupancy at all times and drastically reduces idle periods. The result is significantly higher token throughput and better latency consistency, especially when serving many concurrent users with varying sequence lengths.

*Shard models efficiently across GPUs and nodes*

For models that are too large to fit into a single GPU's memory, employ model-parallel inference techniques by partitioning the model across multiple GPUs or even multiple servers. This can be done with tensor parallelism, in which it splits each layer's weights and computation across devices, or pipeline parallelism, which splits the model's layers into segments hosted on different GPUs and streams the data through them sequentially. While model sharding introduces communication overhead and some added latency as data must flow between shards, it enables deployment of trillion-parameter models that would otherwise be impossible to serve. Ensure high-speed interconnects, such as NVLink or InfiniBand, between GPUs to make this feasible, and overlap communication with computation where possible. The key is to balance the load so all devices work in parallel and no single stage becomes a bottleneck.

*Offload memory for extended contexts*

Use hierarchical memory strategies to support inference workloads that demand more memory than GPUs have available. Incorporate memory offloading when serving very large models or long sequence

contexts, such as long multiturn conversations and large documents. Less frequently used data, such as old attention KV cache entries or infrequently accessed model weights, can be moved to CPU RAM or even NVMe storage when GPU memory gets tight. Modern inference frameworks can automatically swap out these tensors and bring them back on the fly when needed. While this introduces additional latency for cache misses, it prevents out-of-memory errors and allows you to handle extreme cases. By thoughtfully offloading and prefetching data, you trade a bit of speed for the ability to serve requests with large working sets, achieving a better overall throughput under memory constraints.

# Power and Thermal Management

Performance per watt is a first-class metric—thermal or power throttling erases tuning gains and shortens hardware life. Power caps, efficient packing, and proactive cooling stabilize clocks while cutting energy spend.

Track perf/watt and thermals alongside speed: cap power or underclock memory-bound workloads for better efficiency with minimal throughput loss. Proactively manage cooling, consolidate jobs to run GPUs near full, monitor per-GPU power draw, and schedule around energy price/renewables when it reduces cost. Here are some tips on managing your power and thermal characteristics of your AI systems:

*Utilize efficient and environmentally friendly energy when possible*

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 the power limit to keep within a stable thermal envelope. The goal is consistent performance without thermal-induced dips.

*Use 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 a 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 nonurgent jobs to run at night when electricity is cheaper. This doesn't change single-job performance but significantly cuts costs.

*Consolidate workloads*

Run GPUs at high utilization rather than running 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.

*Configure cooling efficiently*

For air-cooled systems, consider setting GPU fans to a higher fixed speed during heavy runs to preemptively 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. Check periodically for dust or obstructions in server GPUs. Clogged fins can greatly reduce cooling efficiency. For water-cooled, ensure flow rates are optimal and water temperature is controlled.

*Monitor power carefully*

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.

*Improve job resilience for long-running jobs*

If you are running a months-long training job or 24-7 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 versus less wear on the hardware—especially if that hardware will be reused for multiple projects over a long period of time.

# Conclusion

Treat the checklist as a repeatable playbook: profile, tune the right bottleneck at the right layer, and verify gains before scaling out. By methodically applying these practices—from OS and kernels to distributed comms and serving—you'll achieve fast, cost-efficient, and reliable AI systems at any size.

This list, while comprehensive, 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 in this chapter 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 Grace Blackwell GB200 and GB300 NVL72 and Vera Rubin VR200 and VR300 NVL systems.

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, maintainable, cost-effective, and reliable.

Now go forth and make your most ambitious ideas a reality. Happy optimizing!