Chapter 4. Tuning Distributed Networking Communication

In today's AI landscape, the need for seamless and low-latency data movement between GPUs, storage, and network interfaces is a must. In this chapter, we cover NVIDIA Magnum IO (e.g., NCCL, GPUDirect RDMA, GDS) for training and NIXL for disaggregated inference. We'll discuss these in the context of modern GPUs and clusters like the NVL72. You'll learn how these libraries—and their underlying supported hardware—form the critical fabric needed for ultrascale AI systems.

In large-scale systems, even the fastest GPUs can be hindered by inefficient communication and data transfer from memory and disk. We discuss strategies for speeding up data transfers, proper data sharding techniques, how to work directly with fast storage subsystems, and advanced patterns for overlapping communication and computation on GPUs. Overlapping communication and computation is a common pattern that we revisit frequently throughout our journey on AI systems performance engineering.

We explore the importance of overlapping communication and computation using components of NVIDIA's IO acceleration platform called Magnum IO, which includes NCCL, GPUDirect RDMA, and GPUDirect Storage (GDS). We demonstrate how to use these libraries to lower communication latency, reduce CPU overhead, and maximize throughput across all layers of a multi-node, multi-GPU AI system.

High-level AI frameworks like PyTorch can use these low-level libraries to overlap computation with communication. Integrating these technologies into your AI system represents a holistic approach to accelerating communication and data pipelines for both training ultrascale models and scaling distributed inference servers to power applications with billions of users.

All of these optimizations ensure that every component is tuned for peak performance. Performance engineers need to carefully configure and tune the

network and storage fabric to maintain a high level of GPU utilization and "goodput" (useful throughput).

Overlapping Communication and Computation (Pipelining)

Overlapping communication and computation, or *pipelining*, plays a key role in building efficient training and inference systems at scale. In these environments, it's important to keep GPUs busy and spend less time waiting for data.

The main idea is to ensure that data transfers occur concurrently with ongoing computations so that when one task finishes, the results needed for the next stage are already in progress or have been delivered. Modern frameworks such as PyTorch support asynchronous operations so that collective communication (e.g., all-reduce of gradients) can run alongside compute tasks. This reduces idle GPU time (see Figure 4-1)—and improves overall system throughput.

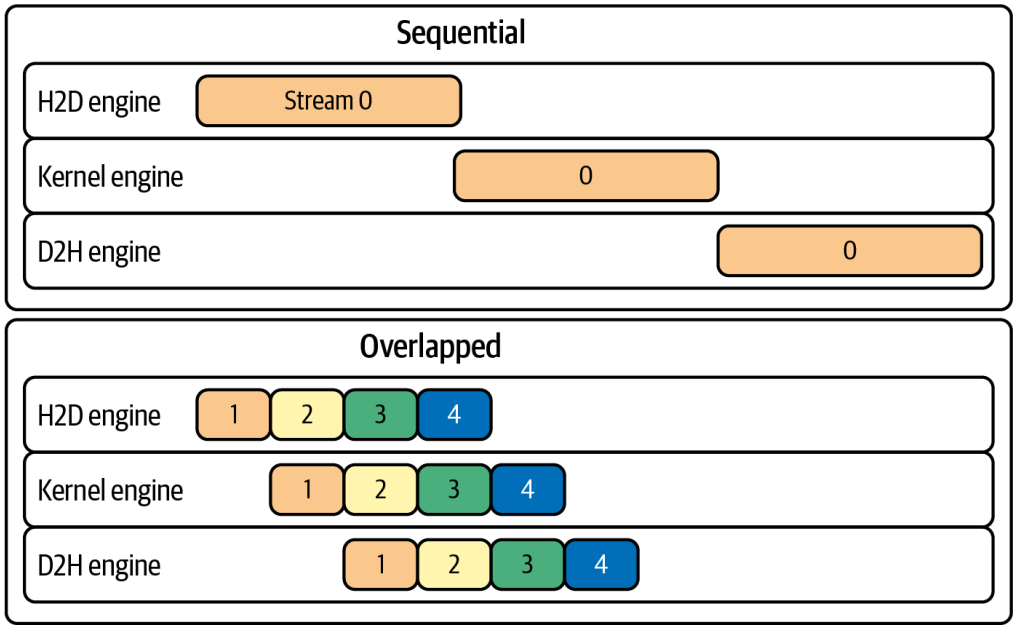

Figure 4-1. Overlapping host-to-device (H2D) and device-to-host (D2H) communication with computation on multiple CUDA streams 0–3

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

Increasing the amount of compute performed between communication events can further minimize the relative overhead of communication. When the system processes larger batches of data, it performs more computation before needing to stop and exchange information.

In distributed training, for instance, this appears as gradient accumulation, where updates from several minibatches are combined into a single synchronization step. By reducing the frequency of communication events, the system lowers the overhead of each exchange and boosts overall throughput.

Another technique that supports seamless overlap between computation and communication is compression. Compression reduces the volume of data that needs to be transferred. For example, if a model compresses its gradients before sending them, the network moves a smaller amount of data. This reduces transfer time and eases congestion.

The shorter transfer time means that the communication phase is less disruptive to the computation phase. Although compression does not directly cause overlap, it shortens the window during which data moves across the network and allows computational work to continue in parallel more effectively.

Modern deep learning frameworks also split large tensor communications into smaller buckets to facilitate overlap. Frameworks like PyTorch automatically divide large gradients or activation tensors into several buckets that are transmitted as soon as they become available. Rather than waiting for an entire layer’s gradients to be ready, for instance, portions of them can begin their all-reduce immediately.

By tuning bucket sizes and scheduling these transfers appropriately, one can achieve a higher degree of overlap and prevent communication delays from stalling the compute pipeline. Tools such as the PyTorch profiler and NVIDIA Nsight Systems offer insight into whether your computation and communication are overlapping, allowing engineers to adjust these parameters for maximal efficiency.

By combining larger batch sizes, gradient accumulation, asynchronous transfers, compression, and bucketing into one cohesive strategy, large, distributed AI models can overcome network limitations and reduce idle time.

This design minimizes synchronization events while achieving high throughput and optimal GPU utilization.

The outcome of these optimizations is a training system that reduces overall training and inference time and makes more efficient use of available hardware resources. Engineers are freed from reinventing low-level networking routines and can focus on innovating model architectures and tuning higher-level parameters, rather than coding intricate data transfer mechanisms.

Asynchronous Execution with Streams

Achieving overlap fundamentally relies on asynchronous execution. GPUs support multiple streams, or queues of operations, that can execute concurrently or overlap if they target different resources. One stream can handle compute kernels such as matrix multiplies, while another stream handles communication such as data copies and all-reduce calls.

By assigning work to different streams and using nonblocking operations, communication can happen in the background. For example, an all-reduce operation can be launched to a separate stream without waiting for completion.

Meanwhile, the default stream proceeds with further computations on independent data. This requires that communication libraries such as NCCL use nonblocking calls that return control immediately. By doing this, the programmer ensures proper synchronization when needed.

In practice, AI frameworks hide most of this complexity. PyTorch's `DistributedDataParallel` automatically installs hooks on the backward pass so that each gradient bucket triggers an asynchronous NCCL all-reduce on a dedicated communication CUDA stream, while the default CUDA stream continues computing gradients for subsequent layers.

We will dive deeper into CUDA streams in a later chapter, but just know that they are useful for overlapping communication and computation—and avoiding unnecessary synchronization points in your code.

This interleaving of computation and communication with CUDA streams creates a steady wave, or cascading pipeline, of work that hides communication latency and keeps the GPU busy at all times. To maintain proper overlapping, avoid unnecessary synchronization points with `torch.cuda.synchronize()` or inadvertently triggering a full device sync by moving tensors to the CPU with `torch.Tensor.item()`. If you do need to measure the overall iteration time, place a single synchronization at the very end of the iteration to wait for all outstanding GPU work to finish without disrupting the ongoing pipeline.

Reducing Communication Frequency and Volume

As noted, performing more work per communication step can increase overlap and efficiency. Gradient accumulation during model training is one such technique. Instead of all-reducing gradients for every single minibatch, you accumulate gradients over a few minibatches, sum them locally, and then do one all-reduce. This effectively trades off memory to store the unreduced gradients for fewer synchronization points.

Consider accumulating four minibatches; you cut the all-reduce frequency by 4×. This allows more computations to happen in between synchronizations. The downside is that your effective batch size increases. This can affect model convergence and memory usage. You can often find a sweet spot that creates a good balance between communication frequency, memory usage, and convergence.

Another approach to reduce communication volume is compressing or quantizing the data exchanged. Techniques like gradient compression can reduce the amount of data sent in each communication without significantly impacting model quality. Less data to send means faster transfers and more opportunity to hide those transfers behind computation. The extreme of this is sparsification. When using sparsity, you send only a fraction of the gradients. This typically requires algorithmic changes to preserve accuracy, however.

Bucketing, as implemented in PyTorch's Distributed Data Parallel (DDP) communication mechanism, also reduces per-call overhead by grouping many small tensors into larger messages. However, bucket sizing is a trade-off. Very large buckets maximize bandwidth utilization but delay the start of communication since you wait for more gradients to accumulate before

kicking off the all-reduce. Very small buckets start transfers earlier but incur more overhead due to many small NCCL calls.

As of this writing, the default bucket size in PyTorch DDP is 25 MB. This is a balance that overlaps well in most cases. However, if you have a model with very large layers, you might increase this to reduce overhead. If you have a model with many small layers, you might actually benefit from smaller buckets to start communication sooner. Ultimately, achieving maximal overlap may require profiling different bucket sizes to see which yields the best iteration time.

Achieving Maximal Overlap in Practice

To see the benefit of overlapping communication with computation, let's run through an example that compares two scenarios in which one performs gradient communication synchronously after all computation with no overlap, and one where communication is overlapped with computation such as DDP. We'll simulate a simple distributed training step with two GPUs to illustrate the difference.

Suppose we don't use DDP's built-in overlap and instead implement distributed training manually such that each process computes all gradients locally, then performs an all-reduce on those gradients at the end of the backward pass. This will mimic the unoptimized, nonoverlapping scenario since communication happens only after all computation is done. We can simulate this using PyTorch's distributed primitives, disabling DDP's hooks, and explicitly calling `dist.all_reduce` after `loss.backward()`.

In the code that follows, we launch two processes on gpu/rank 0 and gpu/rank 1 on a single node, in this case, with a simple model. We'll run a forward and backward pass, then manually average gradients across the two processes:

```
import torch
import torch.nn as nn
import torch.optim as optim
import torch.distributed as dist
import torch.multiprocessing as mp

# A simple model with multiple layers to produce multip
class MultiLayerNet(nn.Module):
    def __init__(self, size):
```

```
        super().__init__()
        self.fc1 = nn.Linear(size, size)
        self.fc2 = nn.Linear(size, size)
        self.fc3 = nn.Linear(size, 1)
    def forward(self, x):
        x = torch.relu(self.fc1(x))
        x = torch.relu(self.fc2(x))
        return self.fc3(x)

def train_no_overlap(rank, world_size):
    dist.init_process_group("nccl", init_method="env://
                            world_size=world_size, rank
    torch.cuda.set_device(rank)

    # Each rank synthesizes its own data
    # (avoid sending big tensors via spawn)
    batch_size = 256
    data = torch.randn(batch_size, 1024, device=rank)
    target = torch.randn(batch_size, 1, device=rank)

    model = MultiLayerNet(data.size(1)).to(rank)
    optimizer = optim.SGD(model.parameters(), lr=0.01)

    # Forward + backward (manual, no overlap)
    output = model(data)
    loss = nn.functional.mse_loss(output, target)
    loss.backward()

    # Synchronous gradient all-reduce after backward
    for p in model.parameters():
        dist.all_reduce(p.grad, op=dist.ReduceOp.SUM)
        p.grad /= world_size

    optimizer.step()
    dist.destroy_process_group()

if __name__ == "__main__":
    import torch.multiprocessing as mp
    mp.set_start_method("spawn", force=True)
    world_size = min(2, torch.cuda.device_count() or 1)
    if world_size > 1:
        mp.spawn(train_no_overlap, args=(world_size,),
                 join=True)
    else:
        train_no_overlap(0, 1)
```

In this code, each process computes gradients for `MultiLayerNet` independently. After `loss.backward()`, we explicitly perform an all-reduce for each parameter's gradient to average them. This is effectively what DDP does internally, but here we wait and do it after the entire backward pass has finished—rather than doing the all-reduce concurrently during the backward pass.

If we were to time this iteration, the all-reduce operations would add directly to the iteration time since it's not overlapping with any other steps. For example, say the forward and backward computations together take 10 ms and the gradient all-reduces take 12 ms. The total iteration time would be roughly 22 ms in this approach. In contrast, a fully overlapped implementation might achieve a total time closer to the max of these values, or 12 ms in our case. This is possible since the all-reduce communication can be almost completely hidden under the computation.

In practice, if we profile this no-overlap case with a tool like Nsight Systems, we would see all the backward computation kernels for fc1, fc2, and fc3 execute first, and only after they complete do we see NCCL all-reduce kernels for each gradient. There is a clear separation in which compute happens first, then communication. During the communication phase, the GPUs would be idle aside from the NCCL work since no further computations are happening.

Now let's use PyTorch's DDP to perform the same operation with overlap. DDP will hook into the backward pass and overlap gradient reduction with backward computation. The code is similar, but we simply wrap the model with `DistributedDataParallel` and let it handle synchronization:

```
import torch
import torch.nn as nn
import torch.optim as optim
import torch.distributed as dist
import torch.multiprocessing as mp

class MultiLayerNet(nn.Module):
    def __init__(self, size):
        super().__init__()
        self.fc1 = nn.Linear(size, size)
        self.fc2 = nn.Linear(size, size)
        self.fc3 = nn.Linear(size, 1)
    def forward(self, x):
```

```
        x = torch.relu(self.fc1(x))
        x = torch.relu(self.fc2(x))
        return self.fc3(x)

def train_ddp(rank, world_size):
    rank = int(os.environ.get("LOCAL_RANK", rank))
    torch.cuda.set_device(rank)
    dist.init_process_group("nccl", init_method="env://
                            world_size=world_size, rank
    torch.cuda.set_device(rank)

    model = MultiLayerNet(1024).to(rank)
    ddp_model = nn.parallel.DistributedDataParallel(mod
    optimizer = optim.SGD(ddp_model.parameters(), lr=0.

    # Each rank makes its own data
    batch_size = 256
    data = torch.randn(batch_size, 1024, device=rank)
    target = torch.randn(batch_size, 1, device=rank)

    output = ddp_model(data)
    loss = nn.functional.mse_loss(output, target)
    loss.backward()  # DDP overlaps gradient all-reduce
    optimizer.step()
    dist.destroy_process_group()

def main():
    world_size = min(2, torch.cuda.device_count() or 1)
    mp.set_start_method("spawn", force=True)
    if world_size > 1:
        mp.spawn(train_ddp, args=(world_size,), nprocs=
    else:
        print("Only one GPU present; running DDP demo w
        train_ddp(0, 1)

if __name__ == "__main__":
    main()
```

With the DDP overlapping approach, the code is simpler since we rely on `DistributedDataParallel` to handle gradient synchronization rather than writing it ourselves. When `loss.backward()` is called, DDP's internal reducer splits the gradients into buckets and launches NCCL all-reduce operations as soon as each bucket is ready on a separate CUDA stream. For instance, it might all-reduce the `fc3.weight` and `fc3.bias` gradients immediately after they are computed since those are from the last layer and

are computed last in the backward pass, while the `fc1` and `fc2` gradients from earlier layers will have already been all-reduced by the time we reach the end of the backward pass.

If the model is very small such that all gradients fit into one bucket, DDP might do only one all-reduce at the end, which wouldn't overlap much. But with larger models and bigger batch sizes, there will be multiple buckets and significant overlap—showing even more impressive performance gains from this technique.

Profiling the DDP case would show NCCL all-reduce kernels interleaved with backward compute kernels. So instead of a clear two-phase split, we'd see a sawtooth pattern in the timeline with some compute happening, then some NCCL communication happening, then compute, and so on.

Key signs of good overlap are that the total iteration time is lower than the sum of compute + comm times, and the GPU is rarely idle waiting for communication because the all-reduces happen during the compute, as shown in Table 4-1.

Table 4-1. The benefit of overlap

Metric	No overlap (manual sync)	Overlap (DDP)	Notes
Total backward + comm time	100% (baseline)	~70% of baseline	e.g., 30% faster per iteration due to overlap (illustrative)
Comm start time	After backward complete	During backward	In DDP, comm begins midway through backward
GPU idle during comm phase	Yes—after backward, GPUs wait during all-reduce	Minimal—comm runs while other layers still computing	DDP hides most of the latency
SM (GPU) utilization	Lower (some cycles where SMs idle during comm)	Higher (continuous activity)	Overlap keeps GPU busy more consistently
Overlap achieved (% of comm covered by compute)	0% (serial execution)	~50% (or more)	Rough estimate: larger models or batches can overlap more

---

Note: The numeric values in all metrics tables are illustrative to explain the concepts. For actual benchmark results on different GPU architectures, see the GitHub repository.

In this example, overlapping communication with computation yields roughly a 30% iteration time improvement in our example workload. In larger training jobs, the gains would be more substantial since larger models create more potential for communication bottlenecks. PyTorch `DistributedDataParallel` uses 25 MiB buckets by default. This way, it launches an all-reduce for each bucket as soon as it is ready. Tuning `bucket_cap_mb` can help increase overlap for your specific model topology, but larger buckets increase latency for the last bucket.

The takeaway is that a well-tuned DDP should overlap most of the gradient communication with computation. Often the only portion of communication that cannot be overlapped is the tail end, or the last gradient bucket, if it happens to finish after the last compute. Research efforts are ongoing to even overlap the optimizer step with communication or to use techniques like tensor partitioning to achieve further overlap, but PyTorch's DDP's default overlap strategy is often described as *wait-free backpropagation* (WFBP), which bucketizes gradients and launches reductions as soon as each bucket is ready.

It's worth noting that certain coding patterns can inadvertently eliminate overlap. For instance, if you perform any operation that forces a synchronization between backward and the next iteration, you will stall the computation until all communications are done.

Avoid operations that inadvertently move tensors from the GPU to the CPU (e.g., calling `.item()` on a tensor) until you're sure that all asynchronous GPU work is finished. Otherwise, you will force a synchronization, stall the computation, and slow down your training or inference workload. This typically happens when adding `print()` or `log()` statements for debugging. These can be disastrous for performance.

You should move such operations to a separate stream if possible. Also, manual calls to `torch.cuda.synchronize()` should be minimized and used only for accurate benchmarking—or when required for correctness. Otherwise, they will serialize GPU work and negatively impact performance. DDP's design and PyTorch's operations are already asynchronous and handle dependencies correctly. Explicit synchronization is rarely needed in user code.

In summary, overlapping computation and communication is one of the most effective techniques for distributed performance engineering. Properly utilized, it can significantly reduce training time by hiding communication latencies behind useful work.

In the following sections, we'll explore the software and hardware infrastructure that makes this possible and how to ensure you're getting the maximum overlap on modern systems. Next, we will overview NVIDIA's

Magnum IO stack, which includes technologies that make overlapping of compute and communication possible.

NVIDIA Magnum IO Optimization Stack

Magnum IO, NVIDIA's overarching I/O acceleration platform, brings together a range of technologies to speed up data movement, access, and management across GPUs, CPUs, storage, and network interfaces. There are four key components of the Magnum IO architecture spanning storage, network, in-network computing, and I/O management, as shown in Figure 4-2.

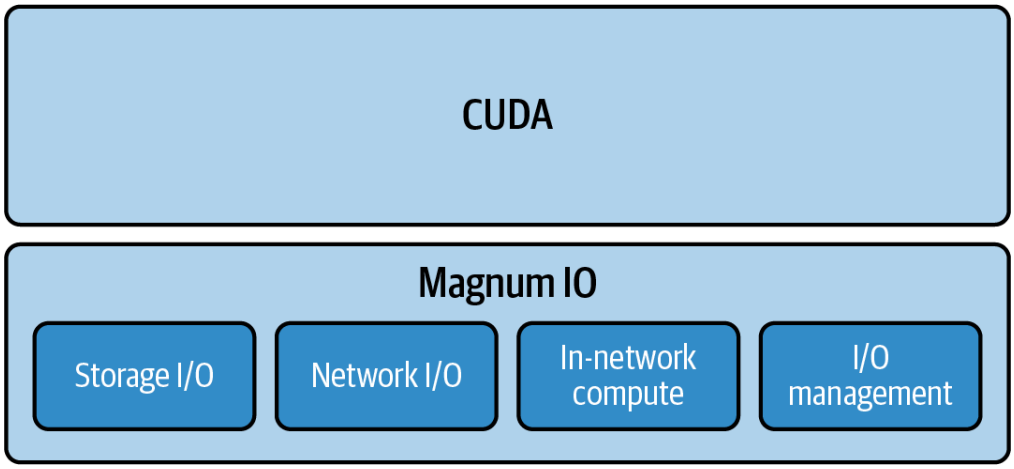

Figure 4-2. Four components of NVIDIA's Magnum IO acceleration platform

Here is a description of the four components in Figure 4-2:

Storage I/O

This is implemented by technologies such as NVIDIA GPUDirect Storage (GDS) and BlueField SNAP. These let GPUs access storage including NVMe SSDs directly without unnecessary copies through host CPU memory. We'll dive deeper into GDS in Chapter 5.

Network I/O

This includes technologies like GPUDirect RDMA, NCCL, NVSHMEM, UCX, and HPC-X (MPI/SHMEM software bundle) to enable direct, high-speed data transfers between GPUs across nodes, bypassing the CPU for internode communication.

In-network compute

SHARP performs in-network reductions inside Quantum-class InfiniBand switches. The reduction arithmetic happens in the switch silicon. BlueField DPUs offload networking and can host control services such as the Subnet Manager and the SHARP Aggregation Manager. When the NCCL RDMA SHARP plugin is enabled and the fabric has SHARP firmware and an active Aggregation Manager, eligible collectives can be offloaded to the IB switches, reducing host and GPU overhead. We will cover NVIDIA SHARP in more detail in a bit.

Ethernet-based GPU clusters rely on technologies like RoCEv2 for RDMA but generally lack features like SHARP. This is one of the reasons that many ultrascale AI systems use InfiniBand or similar high-performance interconnects instead of Ethernet. SHARP provides a significant performance boost and should be utilized when available.

I/O management

Tools like NVIDIA NetQ and Unified Fabric Manager (UFM) fall in this category, providing real-time telemetry, diagnostics, and lifecycle management for the data center's I/O fabric.

These components work together through flexible APIs and SDKs that hide much of the low-level complexity. Magnum IO's goal is to maximize end-to-end throughput for large-scale AI and data analytics workloads. Magnum IO continues to evolve alongside new hardware.

Magnum IO now integrates support for NVLink Switch network domains, which enables intra-rack GPU communication at fabric scale. It also leverages advancements in InfiniBand (Quantum-2 and Quantum-X800 families) and Ethernet (Spectrum-X) to further reduce communication overhead.

Throughout this chapter, we will dive into several of these components such as RDMA for network transfers, NCCL for collectives, NIXL for inference data movement, and GDS for efficient data loading. Additionally, we will see how to profile and tune each of them. Let's dive right into it!

High-Speed, Low-Overhead Data Transfers with RDMA

RDMA is a technology optimized for low-latency, high-throughput data transfers. RDMA works by allowing direct memory-to-memory communication between devices without burdening the CPU with unnecessary data-copy operations. In a nutshell, RDMA bypasses much of the traditional kernel network stack and allows a NIC to directly read/write application memory. This avoids CPU involvement in each packet and reduces context switches and buffer copies. Prefer RDMA paths where available and verified. And always confirm (and continuously reconfirm) with logs and microbenchmarks that the RDMA data path is active.

In container environments like Docker and Kubernetes, ensure the container has direct access to the host's InfiniBand devices (e.g., `/dev/infiniband`). Otherwise, NCCL may silently fall back to TCP sockets instead of GPUDirect RDMA—and without any obvious errors to highlight the degradation. This results in throughput dropping from tens of GB/s to only a few Gb/s, with no obvious error messages.

A related container pitfall arises when the container's GID assignments don't match the host, as in some "rdma-shared" Docker images. This prevents GPUDirect registration and uses CPU-driven RDMA copies instead of using true GPU-based RDMA.

Always verify that it is true GPUDirect RDMA. Confirm that the kernel module is loaded with `lsmod | grep nvidia_peermem`, and check `dmesg` for initialization. For an end-to-end check, run NCCL with `NCCL_DEBUG=INFO` to confirm NET/IB paths and use RDMA perftests with `--use_cuda` to validate GPU-to-GPU transfers. Verifying will help prevent stealthy performance degradations.

The reduced throughput would show up in the profiler as an order-of-magnitude reduction in throughput. But you have to be aware of this possibility and continuously monitor your system for these types of subtle fallbacks.

NVIDIA's RDMA implementation for GPUs is called GPUDirect RDMA. GPUDirect RDMA lets an RDMA-capable NIC such as InfiniBand and RDMA over Converged Ethernet (RoCE) perform direct memory access (DMA) to and from the GPU's device memory across two servers—bypassing host CPU and system RAM entirely. A data transfer with RoCE is shown in Figure 4-3.

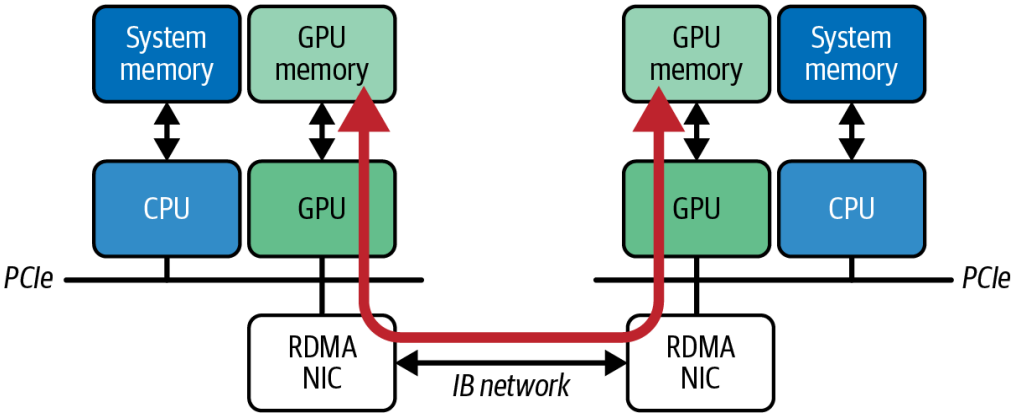

Figure 4-3. GPU-to-GPU direct data transfer with RoCE

By registering GPU buffers with the NIC, GPUDirect RDMA enables one-sided RDMA reads and writes between remote GPUs. This minimizes both latency and CPU overhead in multinode training.

RDMA is supported inherently by InfiniBand and also by some high-speed Ethernet networks through RoCE. With RoCE, you get RDMA-like zero-copy transfers over Ethernet, assuming the network gear supports RDMA and is properly configured for it. Using RDMA with RoCE usually requires a properly configured system with the necessary drivers, including NVIDIA OFED for InfiniBand/RoCE.

The performance difference between using RDMA versus standard TCP/IP networking can be huge. For example, a modern InfiniBand link might provide latency on the order of a few microseconds for a small message, whereas standard TCP over Ethernet might incur 5–10× higher latency.

For large transfers that are network-bandwidth-bound, RDMA on InfiniBand can sustain very high throughput on the order of hundreds of Gbps. In contrast, a typical TCP/IP network might be limited by kernel overhead and NIC speed—often 100 Gbps or less unless using 200–400 Gbps Ethernet with RDMA capabilities. TCP/IP networks also incur more overhead.

In distributed deep learning, large-message throughput tends to matter more than tiny message latency since gradients are large, for instance. Both high bandwidth and low latency keep the protocol efficient—and the GPUs busy.

If you only have Ethernet available, you should use the highest bandwidth and lowest-latency configuration possible. For instance, 200+ Gbps Ethernet with RDMA (RoCE) will perform much better for all-reduce traffic than a basic 10–25 Gbps TCP network. At a minimum, ensure you are using jumbo frames such as MTU 9000. Enabling this configuration for your cluster network, data transfers will send fewer large packets instead of many small ones. This reduces CPU overhead and improves efficiency similarly to how larger disk block sizes improve sequential disk throughput.

Also, it's important to tune the TCP stack for a similar reason. You should verify that Linux `sysctl` parameters like `net.core.rmem_max` / `wmem_max` and the autotuning ranges `net.ipv4.tcp_rmem` / `tcp_wmem` are set high enough to fully utilize a high-bandwidth link.

And, of course, it's important to use a modern TCP congestion control algorithm to improve throughput on high-latency links. On a well-engineered, dedicated cluster network with no external internet traffic, the default CUBIC congestion control typically performs adequately since it's engineered to avoid congestion.

For any high latency-bandwidth links, consider using a modern TCP congestion algorithm like Bottleneck Bandwidth and Round-trip propagation time (BBR) and adjust buffer sizes to ensure full utilization. Always validate that the default settings are not limiting throughput. Use tools like `sysctl net.ipv4.tcp_congestion_control` to inspect and tune the setting.

In cloud or hybrid environments, be cautious of whether you truly have a controlled high-speed connection. For example, if you use AWS EC2 instances with Elastic Fabric Adapter (EFA), you get something similar to InfiniBand-level RDMA between instances deployed in the same "placement group." But if you try to run a multinode training or inference job that spans both an on-premises data center and the cloud without direct connectivity,

your traffic will likely traverse the public internet. This will introduce unpredictable latency and congestion.

Always ensure your multinode setup is on a properly configured, high-performance, low-congestion network. Work directly with your cloud provider to understand every hop in the network architecture.

Even when using RDMA, the CPU is not completely out of the picture. The host still sets up RDMA transfers and handles communication-completion events. Therefore, proper CPU affinity is important. Remember to pin the network interrupt handling—or polling threads—to a CPU core on the same NUMA node as the NIC and, ideally, the GPU as well. For example, if an InfiniBand host channel adapter is on NUMA node 0, bind its interrupt CPU affinity cores to node 0. This reduces cross-NUMA traffic and latency for control operations.

Tuning Multinode Connectivity

For distributed, multinode training with GPUs, it is crucial to ensure that the network is not a bottleneck. This involves using the right communication and networking technologies as previously described—as well as configuring these technologies properly. Here are some tips to adopt and pitfalls to avoid:

Understand the topology

Use `nvidia-smi topo -m` to get a basic GPU interconnect view, but for NVSwitch- and NVLink-based systems, it's recommended to also use `nvidia-smi nvlink` or Nsight Systems to understand multihop switch fabric connectivity.

Leverage NVLink Switch domains if available

The multinode NVIDIA's GB200 and GB300 NVL72 rack solutions connect up to 72 GPUs in a single NVLink domain using NVLink Switch, which provides extremely low per-hop latency—on the order of a few hundred nanoseconds. The GB200 NVL72 architecture provides up to ~130 TB/s of all-to-all bandwidth with submicrosecond latencies across all GPUs in the rack. If your cluster includes such infrastructure, make sure your jobs are placed within the same NVLink

domain to fully utilize this ultrafast interconnect. This can significantly reduce the need for slower InfiniBand and Ethernet communication between nodes. Fortunately, modern InfiniBand switches such as NVIDIA's Quantum series provide up to 800 Gb/s per link and in-network computing features. However, NVLink's massive intra-rack bandwidth and < 1µs latency are preferred. Keep traffic on NVLink/NVSwitch whenever possible.

Use RDMA whenever possible

If running on InfiniBand or RoCE-capable hardware, make sure your communication library, such as NCCL, is actually using RDMA. NCCL will automatically use GPUDirect RDMA if available. But if RDMA is misconfigured or unsupported, NCCL may silently fall back to TCP. One red flag for this is if you notice that during all-reduce operations, GPU utilization drops and CPU utilization spikes. This indicates that the CPU is copying data for communications.

Aggregate bandwidth with multiple NICs if available

Some servers have multiple network interfaces (NICs). NCCL can stripe traffic across multiple NICs (called *multirail*) to increase bandwidth. But you may need to set some environment variables like `NCCL_NSOCKS_PERTHREAD` and `NCCL_SOCKET_NTHREADS` to optimize this. We'll discuss these in more detail in a bit. Just make sure that each NIC is on a different subnet and that NCCL can discover both. With proper setup, using two 800 Gbps NICs in parallel, for instance, gives an aggregate of 1.6 Tbps for NCCL traffic. And four such NIC links (e.g., two dual-port NICs) can achieve ~3.2 Tbps.

Utilize optimized "direct NIC" mode when available

Favor high-bandwidth, multirail NIC configurations that give each GPU or small groups of GPUs sufficient dedicated network bandwidth. Physically, NICs attach using PCIe to the host CPU or to a DPU. With modern GPU systems, NCCL supports GPU-initiated networking with InfiniBand GPUDirect Async (IBGDA) and the direct NIC path, as shown in Figure 4-4. This lets the GPU drive full-bandwidth RDMA without CPU intervention.

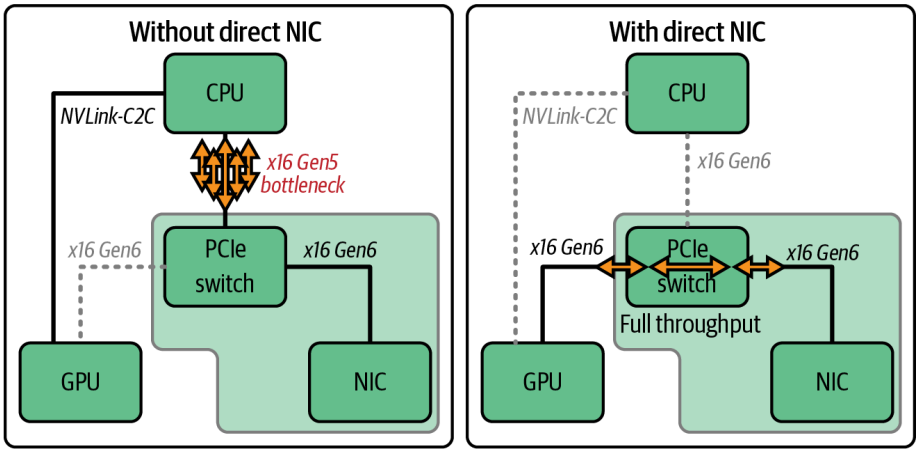

Figure 4-4. Bypassing CPU bottlenecks with direct connectivity between GPUs and NICs

Check for misconfigurations

A common pitfall is a mismatch in network configuration that causes a fallback to a slower path. If RDMA is not working due to a misconfiguration, for instance, NCCL might be using TCP on a 100 Gbps Ethernet network but getting only a fraction of that due to kernel overhead. Even worse, if the cluster's high-speed network is misidentified, traffic might go over a slower management network running only 10 Gbps Ethernet without the user realizing. Tools like NCCL's debugging output and network interface counters (`ibstat` , `ifstat`) can help verify which interface is being used more heavily. For modern systems with large 200–400 Gbps paths, dropping to 10 Gbps would cause a severe bottleneck.

Multinode Communication Pitfalls

Scaling training across multiple nodes in a cluster introduces a new class of pitfalls. Here we highlight a few common issues and demonstrate how to fix them with concrete examples.

Pitfall #1: Using a CPU-bound Gloo backend instead of NCCL

PyTorch's distributed framework supports multiple backends for communication. For multi-GPU training, NCCL is the preferred backend for NVIDIA GPUs, but there is also a fallback backend called Gloo, which uses CPUs and TCP sockets. If one mistakenly initializes `ProcessGroup` with Gloo for GPU training—or if NCCL fails to initialize and it falls back to Gloo, the training will still function correctly but all cross-GPU communication will go through the CPU and Ethernet stack. This results in extremely slow performance.

Unfortunately, it's fairly common to accidentally use this misconfiguration since the code appears to work normally and not crash. It just runs an order of magnitude slower and requires either a profiler or careful log analysis to detect. In summary, always specify NCCL for multi-GPU training to utilize RDMA communication. Fortunately, this is PyTorch's default. Otherwise, falling back to Gloo will silently limit your performance (or even error out completely.)

Let's illustrate this with code. We'll simulate two processes running on gpu/rank 0 and gpu/rank 1 on two different "nodes" connected with an 800 Gb/s (100 GB/s) InfiniBand interconnect and perform a large all-reduce of a tensor. First, we intentionally use the Gloo backend (CPU bound) for PyTorch distributed communication as shown here:

```
# dist_allreduce.py

#!/usr/bin/env python

import os
import argparse
import torch
import torch.distributed as dist

def main():
    parser = argparse.ArgumentParser(description="Multi
    parser.add_argument(
        "--data-size",
        type=int,
        default=1024 * 1024 * 100,  # 100M floats ≈ 40(
        help="Number of elements in the tensor",
    )
    args = parser.parse_args()

    # Initialize the default ProcessGroup over env://
    # (uses MASTER_ADDR, MASTER_PORT, etc.)
    dist.init_process_group(backend="gloo", init_methoc

    rank = dist.get_rank()
    world_size = dist.get_world_size()

    # Allocate a large CPU tensor (Gloo is CPU-bound)
    tensor = torch.ones(args.data_size,
       dtype=torch.float32, device="cpu")
```

```
        # Warm up and barrier
        dist.barrier()

        # All-reduce (sum) across all ranks
        dist.all_reduce(tensor, op=dist.ReduceOp.SUM)

        # Barrier
        dist.barrier()

        ...

        dist.destroy_process_group()

    if __name__ == "__main__":
        main()
```

In this code, we intentionally chose `backend="gloo"`. The tensor (400 MB) is allocated on the CPU for each rank (`device="cpu"`) because Gloo is CPU-bound. When you use Gloo, collectives operate on CPU memory and communicate over TCP. If you instead attempt GPU tensors with Gloo, PyTorch will stage through the CPU and be limited by that path. Either way, the result is far slower than NCCL on GPUs. This is inefficient compared to letting GPUs talk directly using RDMA.

In the preceding code, we intentionally chose `backend="gloo"`. If you attempt `dist.all_reduce()` with a Gloo process group, the run will either fall back to host-staged paths or fail depending on the build. This defeats the point of measuring GPU collectives.

Running this code with proper timing would yield the following:

```
Rank0: All-reduce of 400.0 MB took 200.00 ms (2 GB/s)
```

For 400 MB of data, 200 ms is quite slow since this is 2 GB/s aggregated throughput—far below expectations or 100 GB/s for our 800 Gb/s InfiniBand hardware that we're using in this example. This indicates a lot of extra overhead incurred on the CPU path. We confirm this by profiling the CPU utilization, which, in this case, is near 100%.

Let's change the backend to NCCL and see the difference. We simply set `backend='nccl'` and ensure the environment is configured to allow GPU-direct communication. Assuming NCCL is properly configured, this code will use direct GPU-to-GPU communication. The improvement is dramatic. Running the updated code with timing would show something like the following:

```
Rank0: All-reduce of 400.0 MB took 4.00 ms (100 GB/s)
```

Here, we see 4 ms for 400 MB, which is 100 GB/s. This is two orders of magnitude faster—and running at the line rate limit of our 800 Gb/s InfiniBand hardware. This is far better than the 2 GB/s we saw earlier with Gloo. This demonstrates how crucial it is to use NCCL for GPU multinode communication. Using the wrong backend can degrade performance significantly. In short, use `backend="nccl"` so that collectives run on the GPU with GPUDirect RDMA when available.

You can verify the backend in PyTorch by calling `torch.distributed.get_backend()`.

From a GPU's perspective, NCCL's all-reduce collective is done by the GPU using direct memory access to the other GPU's memory. As such, the GPU stays busy doing communication. Either the SMs will be executing some network-copy kernels or the GPU's DMA engines will be active. In contrast, with Gloo, the GPU was essentially idle during the communication since the CPU was involved and the GPU had to wait for data to travel through the CPU's memory buffers over TCP instead of InfiniBand.

In a production cluster environment with multiple NICs, you should explicitly set `NCCL_SOCKET_IFNAME=ib0` so that NCCL's initial TCP handshake runs over the InfiniBand host channel adapter (HCA). This ensures it bootstraps correctly and then hands off to GPUDirect RDMA on the fastest path. Otherwise, you may see failed connections or, worse, silently fall back to a much slower interface. Be sure that all nodes can reach one another over the selected interconnect.

Pitfall #2: Mismatched NCCL versions

If you run PyTorch's bundled NCCL (e.g., `torch.cuda.nccl.version() == ()`) against a different version of the system-installed `libnccl` , you will hang the system. Or worse, you will silently fall back to a slower implementation. This can be difficult to detect. Make sure you have alignment by matching `nvidia-nccl-cu*` packages or rebuilding PyTorch against the system NCCL and avoid these compatibility pitfalls.

Pitfall #3: TCP port exhaustion during NCCL bootstrap

NCCL uses ephemeral TCP ports for its out-of-band setup, and if your OS's `net.ipv4.ip_local_port_range` is too narrow, you can exhaust available ports, causing failed or stalled handshakes. It's recommended that you widen your port range in `/proc/sys/net/ipv4/ip_local_port_range` (e.g., `50000 51000`) to avoid hidden bootstrap failures. Note that modern NCCL versions have improved bootstrap handling, but it's still best to proactively set a broad port range on large clusters.

Pitfall #4: Insufficient network bandwidth or misconfigured NICs

Another multinode pitfall is simply not having enough network bandwidth for the amount of data being synced—or not using all of the available interfaces. This pitfall becomes more common as GPU clusters scale. For example, saturating a single 400 Gbps link per node is easy with Blackwell GPUs.

When profiling your workload under these unfortunate conditions, you will observe that scaling to multiple nodes significantly slows down training. In other words, the "per-GPU throughput" will drop. In this case, check the network links.

It's often useful to monitor the network throughput using `nvidia-smi dmon` , for instance, to collect NVLink/PCIe/Network statistics. You can also use built-in tools like `ethtool -S <iface>` or `ip -s link show <iface>` for byte/packet counters, or launch interactive monitors such as `iftop` or `nload` to watch live NIC throughput.

You can also try to utilize multiple interfaces, if available. If you're saturating an 800 Gbps (100 GB/s) InfiniBand link, for instance, and your job needs

more network throughput, consider enabling NCCL's multi-NIC support—assuming that you have multiple NICs. Make sure that `NCCL_NSOCKS_PERTHREAD` and `NCCL_SOCKET_NTHREADS` are tuned, as these control how many parallel connections and threads NCCL uses for network transfers. In cases with multiple NICs, increasing these environment variable values from their platform-dependent defaults can help utilize both NICs.

For example, if you have two NICs, you might set `NCCL_NSOCKS_PERTHREAD=2` and keep `NCCL_SOCKET_NTHREADS=2` (since 2 threads × 2 sockets = 4 total connections per process). However, do not arbitrarily increase these values. Remember that the product of threads and sockets should not exceed 64 per NVIDIA guidance since more threads mean more CPU usage.

Increase these thread-related settings stepwise (e.g., 2 → 4 → 8), and continuously measure the throughput. Too many threads will contend for resources and potentially diminish returns.

Pitfall #5: Straggler nodes or processes

In multinode training, the slowest node, or GPU, will determine the overall pace because synchronization needs to wait for every node and GPU to respond. If one machine has a slower network link, or is overloaded with other tasks, it will slow down the entire job.

To avoid stragglers, it's important to use homogeneous hardware and dedicated cluster resources for each training job, if possible. This way, your environment is predictable. If you're running in a cloud environment, for instance, mixing different instance types or using different switch fabrics can introduce variability.

Using monitoring tools like NVIDIA's DCGM or InfiniBand counters on each node can help spot if one node has degraded performance due to NIC link flapping or GPU thermal throttling. It's also useful to use collective profiling tools such as PyTorch's `torch.distributed.monitored_barrier` to identify if a particular rank is consistently lagging, as shown here:

```
# barrier_straggler

import torch
import torch.distributed as dist
import os
import datetime

def run(rank, world_size):
    dist.init_process_group(backend="nccl", init_methoc
    local_rank = int(os.environ["LOCAL_RANK"])
    torch.cuda.set_device(local_rank)

    # ... your forward/backward work here ...

    # Before syncing at end of iteration, use a monitor
    try:
        # Wait up to 30 seconds for all ranks
        # if one lags, you'll get a timeout on that rar
        dist.monitored_barrier(timeout=datetime.timedel
    except RuntimeError as e:
        print(f"Rank {rank} timed out at barrier: {e}")

    # Now proceed knowing all ranks are roughly in sync
    dist.destroy_process_group()
```

Here, `dist.monitored_barrier(timeout=datetime.timedelta(seconds=30))` will raise an error on any GPU that doesn't arrive within 30s. This will help you pinpoint stragglers. Combine this with `NCCL_DEBUG=INFO` and `NCCL_ASYNC_ERROR_HANDLING=1` to get both PyTorch and NCCL logs around which rank or link is slow.

Modern monitoring tools like PyTorch's `torch.distributed.monitored_barrier` and NCCL's asynchronous error handling should be used to detect these types of issues quickly.

Pitfall #6: GPU memory fragmentation under UCX/RDMA

PyTorch's caching allocator holds onto GPU memory across iterations. In distributed settings using UCX/RDMA, these long-lived allocations can exhaust registration pools or fragment memory, causing sporadic allocation failures or performance cliffs. Monitoring

`torch.cuda.memory_reserved()` versus `memory_allocated()` helps surface these edge cases:

```
import torch
import torch.distributed as dist
import os
import time

def log_mem(iteration):
    reserved = torch.cuda.memory_reserved()
    allocated = torch.cuda.memory_allocated()
    print(f"[Iter {iteration:02d}] Reserved: {reserved/
          f"Allocated: {allocated/1e9:.3f} GB")

def run(rank, world_size):
    # Standard DDP / UCX init
    dist.init_process_group(backend="nccl", init_methoc
    rank = int(os.environ["LOCAL_RANK"])
    torch.cuda.set_device(rank)

    # Pre-allocate a big buffer that UCX will register
    big_buffer = torch.empty(int(2e8), device=rank)  #
    log_mem(0)

    for i in range(1, 11):
        # Simulate per-iteration tensor allocations of
        small = torch.randn(int(1e7), device=rank)  # ~
        medium = torch.randn(int(5e7), device=rank) # ~

        # Free them explicitly to return to allocator c
        del small, medium
        torch.cuda.synchronize()

        # Log memory after freeing
        log_mem(i)

        # Barrier so all ranks print in sync
        dist.barrier()
        time.sleep(0.1)

    dist.destroy_process_group()

if __name__ == "__main__":
    run(0, 1)
```

The output code show how the reserved memory grows each iteration as the caching allocator holds onto freed blocks. This speeds up future allocations. However, it never returns them to the OS or UCX registration pool. Allocated memory drops back to zero after each free, since no live tensors remain:

```
[Iter 00] Reserved: 0.800 GB, Allocated: 0.800 GB
[Iter 01] Reserved: 1.040 GB, Allocated: 0.000 GB
[Iter 02] Reserved: 1.240 GB, Allocated: 0.000 GB
...
[Iter 10] Reserved: 1.240 GB, Allocated: 0.000 GB
```

To mitigate this situation, be sure to upgrade to the latest CUDA runtime. You can also try using `torch.cuda.empty_cache()` as a last resort to recover from fragmentation issues. However, `empty_cache()` is not a long-term solution. Some long-term solutions include tuning the allocator, tracking down the issue, and fixing it.

Let's summarize multinode pitfalls as follows: always use NCCL for GPU communication, ensure RDMA/high-speed network is active, utilize all available bandwidth (multiple NICs if possible), use the latest NCCL that ships with CUDA to inherit the latest fixes, and watch out for any configuration that would cause a fallback to slower communication. In the next sections, we focus on NCCL specifics and how to optimize intranode and internode GPU communication.

NCCL for Distributed Multi-GPU Communication

NVIDIA NCCL is a many-to-many communication library for operations, called *collectives*, used by groups of GPUs to share data. NCCL underpins most multi-GPU training workloads in NVIDIA's ecosystem.

NCCL provides optimized implementations of collective communication operations like all-reduce, all-gather, broadcast, and reduce-scatter that scale from a few GPUs to many thousands and, someday, millions. When performing model training and inference across multiple GPUs, data such as model weights, gradients, and activations must be exchanged quickly to keep

the GPUs busy. NCCL is the library that orchestrates these exchanges efficiently.

During distributed training, each GPU computes gradients on its portion of data. NCCL is then used to perform an all-reduce of these gradients across all GPUs such that each GPU updates the model weights with the averaged gradients.

During distributed inference, GPUs need to exchange activations and other intermediate results.

In the inference case, some frameworks use NCCL's send() and recv(), but many deployments prefer transports exposed using UCX or specialized libraries like NIXL for lower tail latency and better overlap.

NCCL is optimized for NVIDIA GPUs and supports communication over various interconnects such as PCIe, NVLink, NVSwitch, InfiniBand, and TCP sockets. It will automatically choose the fastest path available between any two GPUs.

Complementing NCCL is the newer NVIDIA Inference Xfer Library (NIXL), which is optimized for inference and point-to-point transfers like KV cache movement.

NIXL provides pluggable storage backends, including POSIX files and GPUDirect Storage (GDS). Object-store support such as Amazon S3 is provided through its object-store plugin and is deployment-dependent. These plugins move KV cache fragments between the memory hierarchy and the storage backends when appropriate. We'll cover NIXL in depth in a bit—as well as in future chapters focused on tuning inference workloads.

Many inference deployments now favor NIXL for these point-to-point transfers due to its lower latency, as described later. However, NCCL send() and recv() are still available, as well. However, they are not as optimized as NIXL for minimal latency. In practice, large-scale inference workloads prefer NIXL for one-to-one transfers due to its lower overhead and latency, whereas NCCL send/recv is used more rarely when custom integration is needed.

Topology Awareness in NCCL

Topology awareness plays a major role in NCCL's performance. NCCL detects how GPUs are physically connected and optimizes its communication pattern accordingly. For example, in a system of fully connected NVLink and NVSwitches, every GPU will communicate with every other GPU using these high-speed interconnects.

While NCCL can use a simple pattern communication like ring all-reduce to communicate with each link equally, it will automatically use a topology-aware hierarchical communication pattern to maximize communication performance. For systems with multiple NUMA node domains, for instance, NCCL might first do an intranode reduce, then a cross-node reduce, then an intranode broadcast, which is effectively a hierarchical all-reduce. The goal is to maximize traffic over the fastest interconnects.

Concretely, consider a topology in which GPUs 0–1 share an NVLink connection and GPUs 2–3 share another NVLink connection. However, any communication between the 0–1 pair and the 2–3 pair must go over a lower PCIe interconnect. In this case, NCCL's hierarchy algorithm will perform the reduce collective on each NVLink-connected pair first, then do a reduce exchange across the PCIe link with one GPU from each pair, then distribute the data within each pair again. This way, the slow PCIe link handles only a fraction of the data.

NCCL will usually choose the most performant approach when it detects such topologies. However, in some cases, the automatic detection might not activate, for instance, if the topology choices are comparable—or if the message sizes are small, etc. It is possible to override NCCL's algorithm selection with the environment variable `NCCL_ALGO` (e.g., `NCCL_ALGO=NVLS,NVLSTree,Tree,Ring,PAT`, etc.), but generally NCCL does a good job of automatically choosing the best path based on the topology. Manual override is usually only for specific situations like troubleshooting, research experiments, and more.

To illustrate topology effects on NCCL, consider a scenario with four GPUs on a system with two PCIe switches (GPUs 0–1 on one switch, 2–3 on another). A naive all-reduce using a single ring over all four GPUs would end up passing a lot of data over the PCIe interswitch link. This would be a major communication bottleneck. In contrast, a hierarchical approach with two

separate rings of 0–1 and 2–3 combined with an exchange between one GPU from each ring/pair would reduce the communication pressure. In practice, NCCL would choose this communication pattern under the hood if it detects the slower link in the topology.

A quick experiment with profiling tools can reveal if NCCL is topology-optimized. Using Nsight Systems or NCCL traces, you will see multiple NCCL CUDA kernels when hierarchy is used. For instance, you would see some intragroup all-reduce kernels and some intergroup kernels. We will also see performance differences. For example, the nonoptimized, topology-unaware algorithm will achieve on the order of tens of GB/s per GPU because one stage of the all-reduce goes over the slower PCIe link, whereas the topology-aware algorithm can achieve hundreds of GB/s per GPU by utilizing NVLink fully and minimizing PCIe usage.

Consider a naive approach: GPUs waiting on the PCIe interconnect measuring only 60% SM utilization and a total iteration time of 100 ms without overlap. In this case, many warps are stalled on memory access since they are waiting for data transfers running over the slow PCIe interconnect. In the topology-aware approach, however, SM utilization jumps to 90%, and total iteration time drops by 30% from 100 ms down to 70 ms—showing far fewer memory stall cycles. This indicates that the GPUs were much more fully utilized with less waiting for data since transfers were happening over NVLink instead of PCIe, as shown in Table 4-2.

Table 4-2. Key GPU performance metrics before and after applying topology-aware NCCL and compute–communication overlap optimizations

Metric	Before (no overlap)	After (with overlap)
SM busy	60%	90%
Memory stall warps	High	Much lower
Iteration time	100 ms (no overlap)	70 ms (with overlap)

In summary, topology awareness can make or break the performance of scaling multi-GPU systems. A rule of thumb is to keep as much communication as possible on the fastest interconnect available—likely NVLink/NVSwitch for intranode communication—and minimize transfers over slow paths such as PCIe and inter-NUMA node links.

If your multi-GPU job isn't scaling on one node, first verify that traffic isn't being forced over slow paths. And remember that GPUs have only a fixed number of direct NVLink lanes. For instance, each Blackwell GPU in a GB200/GB300 NVL72 system supports 18 NVLink 5 links at ~100 GB/s each for a total of ~1.8 TB/s of GPU-to-GPU bidirectional aggregate. This is double the previous generation's 900 GB/s.

Communicating between devices that aren't directly paired can drop you back to fewer lanes—or even PCIe. If data needs to cross NUMA domains, your throughput will drop significantly. In an NVL72 rack, all 72 Blackwell GPUs are part of a single NVLink Switch domain. Within an NVL72 rack, any GPU can reach any other GPU in a single NVSwitch stage at full bisection bandwidth. The GPU domain provides uniform all-to-all connectivity with NVLS support.

NCCL's hierarchy algorithm should automatically pick the highest-bandwidth routes, but you should confirm this in your profiling. If you still hit a bandwidth wall, constrain your job to a tightly connected subset of GPUs.

For instance, four GPUs on the same NUMA node or within a single NVSwitch island is much better than spanning all eight GPUs across slower links. The extra synchronization overhead over those limited or indirect links will often outweigh the benefit of using the additional devices.

It's also worth mentioning that the NVIDIA superchips blur the line between CPU and GPU memory. They offer extremely fast CPU-GPU interconnects on the order of 900 GB/s NVLink-C2C between CPU and GPU in a Grace Blackwell Superchip, for instance. This allows the CPU memory to act as a high-speed extension of GPU memory.

This means that even if some part of the all-reduce involves the CPU or system memory, it may still be as fast as older GPU-GPU links. The key takeaway is to optimize your intranode communication by using the fastest paths available—and minimize usage of slow paths.

Tools like NVIDIA Nsight Systems—or NCCL's own traces with `NCCL_DEBUG=INFO` and `NCCL_TOPO_DUMP_FILE=<path>` —will show if NVLink paths are being utilized fully.

NCCL Communication Algorithms

Internally, NCCL can employ different communication algorithms depending on the size of data, number of GPUs, and topology. The primary algorithms NCCL uses for collectives are Ring, Tree, CollTree, CollNet, and Parallel Aggregated Tree (PAT), among others. Let's dive into each of these:

Ring

In the ring all-reduce, GPUs are logically arranged in a ring. Each GPU sends data to its neighbor and receives data from its other neighbor in a pipelined fashion. For an all-reduce, each chunk of the data will circulate around the ring, accumulating partial sums. The ring algorithm has the nice property that it perfectly balances network load such that each GPU sends and receives exactly the same amount of data. It is bandwidth-optimal, as each link transmits $2 \times$ (`data_size` $\div$ `num_gpus`) bytes in an all-reduce collective. The downside is latency. The total time scales with the number of GPUs since the data has to traverse all hops. Ring is often great for large messages because, when you send very large messages, the time spent actually moving bytes far outweighs the cost of actually starting the transfer. This is known as a *bandwidth-dominated workload*.

Tree and NVLSTree

In the tree algorithm, reductions and broadcasts are done in a tree structure using the spanning tree algorithm. An all-reduce is actually a reduce-scatter followed by a broadcast. A tree can complete an all-reduce in O(log N) steps for *N* GPUs—as opposed to O(N) for the ring. As such, a tree algorithm provides lower latency for smaller messages. However, it may not fully utilize all links for large messages because not all GPUs transmit all the time. Some GPUs are leaves of the tree and send only one time up the tree, for instance. NCCL's tree algorithm is optimized and often used for smaller message sizes in which the total time is dominated by the transfer-startup latency. This is known as a *latency-dominated workload*—in contrast to the ring algorithm's bandwidth-dominated use case for large messages. Using `NVLSTree` will enable NVLink SHARP offload.

CollTree (hierarchical tree collectives)

CollTree builds a two level tree to reduce latency while preserving high local bandwidth. Within each fast local domain (e.g., all GPUs on a node or within one NVSwitch island), it forms a local tree and performs a reduce scatter followed by a broadcast. A single leader from each local group then participates in a second-level tree across groups over RDMA. The two levels are pipelined so that tensor chunks that finish the local phase can immediately enter the cross group phase. Compared with a flat ring, this reduces the number of cross node steps to O(log N) and shortens latency for small and medium messages. At the same time, it still uses full bandwidth inside the node. On NVLink domains, the local tree is routed over NVSwitch and can use NVLink SHARP for in-switch aggregation when available. On InfiniBand fabrics, the cross group tree can be offloaded to SHARP when the NCCL SHARP plugin is enabled. For very large messages, Ring or parallel aggregated tree (or PAT, discussed later in this list) may achieve higher peak throughput while CollTree is preferred when cross node latency dominates.

CollNet (hierarchical collectives across nodes)

CollNet, also known as tree parallelism, combines two collective strategies to optimize communication at different scales. First, it groups GPUs that share a fast local interconnect, such as all GPUs in a single node or within an NVSwitch island. CollNet then applies a high-throughput algorithm such as a ring or local tree to aggregate the data within each group of GPUs. One designated leader GPU from each group participates in the second-level tree reduction across groups. This minimizes the number of cross-group communication rounds. By layering a local reduction on top of a global tree exchange, CollNet delivers both low latency for internode transfers and high bandwidth for intranode traffic. This makes it especially effective at reducing network load in very large, multinode GPU clusters.

Parallel aggregated tree (PAT)

PAT is NCCL's pipelined hybrid of ring and tree algorithms. As soon as one segment of the tensor has been reduced across its tree of GPUs, the next segment simultaneously begins its own tree reduction in a staggered, round-robin manner. This overlap of successive reduction phases lets PAT keep links saturated and achieve bandwidth close to a pure ring all-reduce. At the same time, it bounds transfer-startup

latency to O(log N) per segment, similar to the tree algorithm. In practice, PAT splits a large message into multiple chunks, launches a tree-based reduce-scatter on chunk 1, then immediately issues the same on chunk 2, and so on. This interleaves so that there is always work in flight. The result is near–ring-level throughput for large data transfers plus tree-level latency advantages for smaller segments. It's a best-of-both-worlds approach.

As when choosing any communication algorithm, the choice of NCCL algorithm typically comes down to message size and topology. Small messages (on the order of 10s of megabytes) favor tree algorithms since there are fewer steps. Large messages favor ring algorithms because they provide better bandwidth utilization.

By default, NCCL will automatically choose the best algorithm for a given collective operation, message size, and topology. NCCL supports symmetric memory optimizations and low-latency kernels that reduce all-reduce latency for small and medium messages on NVLink-connected systems. In some cases, this reduction has been measured up to ~7.6× for small and medium messages on NVLink-connected systems. These improvements, alongside algorithms like PAT, further decrease communication overhead on systems like the NVL72.

NCCL continues to evolve its communication strategies to be topology aware. For instance, NCCL can utilize NVSwitch's hardware multicast for one-hop broadcasts within an NVLink domain. This is ideal for situations in which you need to send identical data, such as updated model weights, to all GPUs at once.

NCCL also utilizes the latest hardware advancements. For instance, it can leverage SHARP on InfiniBand and NVLink SHARP (NVLS) on NVSwitch-based fabrics. This will accelerate collectives such as all-gather and reduce-scatter on supported systems when the NCCL-SHARP plugin is configured.

Additionally, NCCL implements the PAT algorithm, which combines ringlike throughput with treelike latency, as mentioned earlier. This algorithm divides large messages into chunks, inspects the physical GPU and switch topology layout, and uses this information to interleave reduce-scatter and all-gather phases across different CUDA streams appropriately. This takes full advantage

of the network topology to balance the best of both worlds: treelike latency and ringlike bandwidth when the hardware and network support it.

By default, NCCL's communicator initialization automatically inspects the message size, interconnect topology, and GPU generation to automatically pick the fastest algorithm and protocol combination for each collective and topology. However, if profiling your workload reveals suboptimal communication, such as unexpectedly high cross-node latency, you can override the communication algorithm on a case-by-case basis by setting the `NCCL_ALGO` environment variable (e.g., `NCCL_ALGO=NVLSTree,PAT`). This will force NCCL to use a particular algorithm on that communicator. If setting this variable in code, make sure to do it before calling `ncclCommInitRank()`.

Distributed Data Parallel Strategies

In practice, large-scale training and inferencing of multi-billion- and multi-trillion-parameter models requires a combination of parallelism strategies, including data parallel, tensor model parallel, pipeline parallel, expert parallel, context parallel, etc. These are required to scale your training clusters linearly and not waste GPU resources with excessive overhead.

The key is to overlap communication with computation at every level. This includes using NCCL for all-reduce and NIXL for one-to-one transfers. Using these mechanisms, you can scale to thousands and millions of GPUs with high efficiency.

Other techniques like gradient accumulation and activation checkpointing are also critical at ultrascale to manage the memory footprint without sacrificing throughput.

When scaling to multiple GPUs on a single node, PyTorch offers both data-parallel (split the data) and model-parallel (split the model) approaches at the framework level. We'll cover these in more detail in a later chapter, but for now, let's compare two of the most basic data-parallel strategies from a systems performance standpoint: `nn.DataParallel` (DP) and `torch.distributed.DistributedDataParallel` (DDP). It's important to understand their differences as choosing the wrong one can severely impact performance:

Data parallelism (DP)

DP is an easy-to-use API that involves a single process, or single Python thread, controlling multiple GPUs. The module automatically splits each input batch across the available GPUs. It then performs forward passes on each split, gathers the outputs back to the main GPU, and computes the aggregated loss. Finally, during the backward pass, it gathers gradients back to the main GPU, averages them, and broadcasts back to the others.

In DataParallel, the entire training loop is single-process. This makes it simpler to integrate since there is no need to launch multiple processes. Even though DP uses multithreading, it is limited by Python's GIL (Global Interpreter Lock) for launching operations on different devices. As such, DP does not scale well beyond 2–4 GPUs because the single Python thread becomes a bottleneck and the GPU utilization suffers. Additionally, the gradient gathering step in DP is synchronous and does not overlap with computation. This means it behaves similarly to our "no overlap" scenario described earlier.

Fully sharded data parallelism (FSDP)

FSDP avoids full model replicas by sharding activations, gradients, and parameters across GPUs, greatly reducing memory overhead. For ultrascale models, FSDP is often combined with other parallelism strategies like tensor parallel and pipeline parallel.

We'll talk about FSDP—and other ultrascale training techniques like expert parallelism—in Chapter 13. This section will focus on DP and DDP to establish the foundation for the additional complexity discussed in later chapters.

Distributed Data Parallel (DDP)

DDP uses one process per GPU device and relies on NCCL to communicate gradients. Like most simple data parallel strategies (FSDP being the exception), each process has its own copy of the model.

During the backward pass, gradients are exchanged, or all-reduced, directly among GPUs. This all-reduce communication is typically overlapped with the backward computation, which is ideal, as we discussed earlier.

DDP avoids the GIL issue entirely by using separate processes. And NCCL's efficient C++ kernels handle communication. The result is that DDP nearly always outperforms DP for multi-GPU training. In fact, PyTorch developers recommend using `DistributedDataParallel` over `DataParallel` for any serious multi-GPU work because DP's Python threading, limited by the dreaded GIL, often becomes a bottleneck.

Let's revisit the example from earlier, but in the context of comparing DP and DDP on a single node. We will use a simple model and measure a single training step with DP and DDP.

In this scenario, we use DataParallel to wrap a model so that PyTorch splits each input batch and uses two GPUs. We'll time a single training iteration:

```
# before_dataparallel.py

import torch
import torch.nn as nn
import torch.optim as optim

# Dummy model and dataset
class SimpleNet(nn.Module):
    def __init__(self, input_size, hidden_size):
        super(SimpleNet, self).__init__()
        self.linear1 = nn.Linear(input_size, hidden_siz
        self.relu = nn.ReLU()
        self.linear2 = nn.Linear(hidden_size, 1)
    def forward(self, x):
        return self.linear2(self.relu(self.linear1(x)))

# Setup model and data
input_size = 1024
hidden_size = 256

model = SimpleNet(input_size, hidden_size)

model.cuda()  # move model to GPU 0, it will also repli

model = nn.DataParallel(model)    # utilize 2 GPUs (0 a
```

```
optimizer = optim.SGD(model.parameters(), lr=0.01)
data = torch.randn(512, input_size).cuda()   # batch of
target = torch.randn(512, 1).cuda()          # target c

# Run a single training step
output = model(data)                         # forward (DP
loss = nn.functional.mse_loss(output, target)
loss.backward()                              # backward (D
optimizer.step()
```

Here, the forward pass of `nn.DataParallel` splits the input tensor (matrix) of shape [512,1024] into two tensors of size [256,1024]. One tensor is sent to GPU 0 and one is sent to GPU 1. Both GPU 1 and GPU 2 contain replicas of the same model using `model.cuda()` in the preceding code.

Technically, the model was initially only on GPU 0, but when `forward` is called for the first time, DP automatically copies the model to GPU 1—and any other GPUs involved—before executing the computation.

Then `DataParallel` launches the forward pass on GPU 0 and GPU 1 in parallel using one thread per GPU to enqueue each replica's computation. However, because these threads share the single Python process and contend for our nemesis, GIL, the kernel-launch calls occur sequentially in Python. Fortunately, the enqueued GPU work runs on the device concurrently.

After the forward pass, `DataParallel` gathers all per-GPU outputs onto the primary device (GPU 0) for loss calculation. During the backward pass, it similarly collects and sums gradients from all replicas onto GPU 0, broadcasts the aggregated gradients from GPU 0 back to the remaining GPUs (GPU 1 in this case), then runs the optimizer step.

A couple things to highlight in this example with `DataParallel` . GPU 0 bears the extra burden of gathering and summing all gradients. Moreover, each gradient reduction (GPU 1 → GPU 0 and back) is performed synchronously—blocking further work on the backward pass. This design incurs two key penalties. First, the Python controller thread must serialize kernel launches on each device, which adds CPU-side overhead. Second, because gradient aggregation isn't overlapped with ongoing computation,

GPU 0 can become a performance bottleneck. Let's now compare how `DistributedDataParallel` addresses these issues.

After switching to `DistributedDataParallel`, we start one process per GPU using `torch.multiprocessing.spawn`, for example. Each process holds its own complete model replica and works on a separate slice of the batch. In the next example, the batch size is 256. With two processes running, the effective total batch size remains at 512, which matches the DataParallel setup for a fair comparison:

```
# after_ddp.py

import os, time
import torch
import torch.nn as nn
import torch.optim as optim
import torch.distributed as dist
import torch.multiprocessing as mp

class SimpleNet(nn.Module):
    def __init__(self, input_size, hidden_size):
        super(SimpleNet, self).__init__()
        self.linear1 = nn.Linear(input_size, hidden_siz
        self.relu = nn.ReLU()
        self.linear2 = nn.Linear(hidden_size, 1)
    def forward(self, x):
        return self.linear2(self.relu(self.linear1(x)))

def train_ddp(rank, world_size):
    rank = int(os.environ.get("LOCAL_RANK", rank))
    torch.cuda.set_device(rank)
    dist.init_process_group("nccl",
       init_method="env://",
       world_size=world_size, rank=rank)
    model = SimpleNet(input_size=1024,
       hidden_size=256)

    model.cuda(rank)

    ddp_model =
       nn.parallel.DistributedDataParallel(model,
         device_ids=[rank])
    optimizer = optim.SGD(ddp_model.parameters(),
       lr=0.01)
```

```
    # Each process gets its own portion of data
    batch_size = 256
    data = torch.randn(batch_size, 1024).cuda(rank)
    target = torch.randn(batch_size, 1).cuda(rank)

    # Run one training iteration
    output = ddp_model(data)
    loss = nn.functional.mse_loss(output, target)
    loss.backward()
    optimizer.step()

    dist.destroy_process_group()

if __name__ == "__main__":
    main()
```

You can run this using the `torchrun` launcher (or your cluster's MPI/SLURM/Kubernetes integration) and setting the `MASTER_ADDR` and `MASTER_PORT` environment variables. Running the following script, you will see output like the following (note: these results are workload and hardware specific and will not necessarily match your results):

```
# Environment (common gotchas)
export NCCL_DEBUG=INFO
export NCCL_ASYNC_ERROR_HANDLING=1
export NCCL_SOCKET_IFNAME=ib0      # use your HCA (e.g.
# Optional: for multi-rail IB, set NIC ordering
# so ranks use distinct rails.

# Local bring-up, 2 GPUs
torchrun --standalone --nproc-per-node=2 after_ddp.py

# SLURM (example)
srun --ntasks=$WORLD_SIZE --gpus-per-task=1 --nodes=$NN
     --cpus-per-task=8 python after_ddp.py

### Output: ###

DDP step took 30.00 ms
```

While your exact number will vary, the DDP iteration is significantly faster than DP. In this case, DDP is 33% faster than DP even though they're both

processing the same amount of overall data.

The improvement comes from multiple factors. First, each process handles half the batch without Python GIL contention, so they truly run in parallel. Next, the gradients are all-reduced using NCCL, which overlaps communication with the backward computation. Additionally, there is no extra copying of gradients to a single aggregation GPU (e.g., GPU 0) and back. Also, each GPU's gradients are directly exchanged and averaged in place. Last, the communication work is spread across all GPUs that participate in this all-reduce collective rather than burdening a single GPU responsible for aggregating data—among many other things.

In our example, DDP required 30 ms to complete versus DP's 45 ms. In larger models, the gap will be even bigger—especially as you scale to thousands of GPUs. DP might actually degrade superlinearly when scaling beyond 2–4 GPUs because the main thread becomes overwhelmed. DDP, on the other hand, tends to scale well to the number of GPUs, limited mostly by communication bandwidth and not by CPU or single-GPU overhead.

To sum up, one should always prefer `DistributedDataParallel` over `DataParallel` for multi-GPU training—plain and simple. The PyTorch team explicitly recommends this because of the performance and scalability benefits. The only situation where DP might be acceptable is for quick prototyping on 2 GPUs where ease of use is valued and performance is secondary.

If you care about training throughput, it takes only a few lines to switch from `DataParallel` to `DistributedDataParallel`. You can do this even on one node with multiple GPUs. By spawning one process per GPU (e.g., `torch.multiprocessing.spawn`), each process holds its own model replica and data partition.

Under the hood, DDP uses asynchronous NCCL all-reduces to overlap gradient communication with computation and avoids Python GIL contention entirely. This results in much better GPU utilization and faster end-to-end iteration times. For these reasons, performance-minded engineers on multi-GPU systems almost always favor `DistributedDataParallel` over `DataParallel`.

NCCL Communicator Lifecycle and Environment Gotchas

While NCCL abstracts most low-level details, how we use NCCL in our code can still affect performance. Additionally, NCCL has many environment variables to control its behavior. Misconfiguring these variables can degrade performance or even cause hangs.

In this section, we cover common pitfalls related to NCCL communicators and environment settings. We also show how to diagnose and avoid these issues.

Pitfall #1: Creating NCCL communicators too often

A NCCL communicator represents a group of GPUs, or ranks, that can communicate collectively. Creating a communicator with either C++'s `ncclCommInitRank` in C++ or PyTorch's `torch.distributed.init_process_group` is an expensive operation. Initializers require that all ranks exchange information with one another, including unique IDs, network addresses, etc. They also set up rings/trees and allocate buffers.

If your code repeatedly initializes NCCL communicators, you'll pay a heavy cost each time. Consider a system with 32 GPUs. If you create 32 separate NCCL communicators, one per rank, this could require 2–3 minutes as opposed to 2–3 seconds (or quicker). Communicator initialization can have worse-than-linear scaling with number of ranks because it often requires all-to-all handshakes and coordination among the many GPUs.

In PyTorch's DDP, this is handled for you. You simply call `init_process_group` once at the start of your program and DDP will create one communicator for all processes. This is subsequently used for all collectives at each iteration.

To illustrate the cost of creating NCCL communicators on every iteration, here is a PyTorch example where someone naively initializes and destroys a NCCL process group every iteration of training:

```
import torch
import torch.distributed as dist
```

```
import torch.multiprocessing as mp

def run(rank, world_size):
    rank = int(os.environ.get("LOCAL_RANK", rank))
    device = torch.cuda.device(rank)
    for i in range(5):  # simulate 5 iterations
        // This naive approach re-initializes NCCL eacl
        // iteration. THIS IS EXTREMELY SLOW AND NOT RE
        dist.init_process_group("nccl", init_method="er
                                world_size=world_size,
        # do a tiny all-reduce to simulate some work
        tensor = torch.ones(1).cuda(rank)
        dist.all_reduce(tensor)
        if rank == 0:
            print(f"Iter {i} done")
        dist.destroy_process_group()
```

If you run this, you will notice it's extremely slow even though the all-reduce is trivial because most of the time is spent in `init_process_group` and `destroy_process_group`. In a real scenario with more ranks, the cost multiplies.

Since the `init_process_group` call is designed to be called once at startup, you should avoid any design that reinitializes it on every iteration. The fix is to initialize once outside the loop, as shown here:

```
import torch
import torch.distributed as dist
import torch.multiprocessing as mp
import os

def run(rank, world_size):
    # Pin GPU
    rank = int(os.environ.get("LOCAL_RANK", rank))
    device = torch.cuda.device(rank)

    # Initialize NCCL communicator once
    dist.init_process_group(
        backend="nccl",
        init_method="tcp://127.0.0.1:45678",
        world_size=world_size,
        rank=rank
    )
```

```
        # Simulate 5 training iterations
        for i in range(5):
            tensor = torch.ones(1, device=rank)
            dist.all_reduce(tensor)

        # Cleanup once at the end
        dist.destroy_process_group()

    if __name__ == "__main__":
        world_size = 2
        mp.spawn(run, args=(world_size,), nprocs=world_size
```

By moving the communicator setup and teardown out of the loop, you eliminate the 48 ms initialization overhead incurred each iteration. This reduces total iteration time by over 98%, as shown in Table 4-3.

Table 4-3. Impact of avoiding repeated communicator init/destroy on per-iteration time

Metric	Before (per iter)	After (per iter)
init_process_group + destroy	48.0 ms	0 ms
dist.all_reduce (1-element tensor)	0.5 ms	0.5 ms
Total iteration time	48.5 ms	0.5 ms

The tiny all-reduce itself remains at 0.5 ms, but previously it was completely dominated by the frequent initialization cost. In real multinode scenarios with more ranks, the savings will multiply. Initializing once is a clear performance best practice.

Pitfall #2: Do not create and destroy NCCL communicators on every iteration

Because NCCL communicator initialization is so expensive, be careful not to accidentally create new NCCL communicators when defining subgroups of processes for model parallelism and pipeline parallelism. Instead, create the subcommunicators once at the beginning using PyTorch's `torch.distributed.new_group()` and reuse these communicators. Never create and destroy communicators on every iteration.

If you need to create multiple communicators because, for instance, you have a dynamic runtime membership scenario or a staged initialization, NCCL provides a C++ API to initialize multiple communicators together using

`ncclGroupStart()`, `ncclCommInitRank(...)`, and `ncclGroupEnd()`. This will greatly reduce overhead.

As of this writing, PyTorch does not support fully dynamic membership changes at runtime without a full communicator teardown. All ranks must invoke creation and destruction calls in lockstep to prevent hangs.

Pitfall #3: Avoid overtuning or disabling NCCL features with environment variables

NCCL has many environment variables such as `NCCL_BUFFSIZE`, `NCCL_NSOCKS_PERTHREAD`, `NCCL_P2P_LEVEL`, `NCCL_SHM_DISABLE`, etc. It's usually best to leave them at their defaults unless you have a specific reason. Better yet, set their values to their current defaults to be explicit and not rely on defaults! Be sure to review the release notes and adjust the values accordingly.

While `NCCL_BUFFSIZE` can be increased to improve bandwidth for large all-reduce operations, it must be sized carefully. Setting it too high can cause GPU memory pressure or force smaller models to evict their working sets. Start at 4 MB and increase stepwise. Monitor GPU memory usage as you increase this value.

A common mistake is to disable features while debugging and forget to reenable them in production. For example, never leave peer-to-peer (P2P) or shared-memory transports turned off in production.

Disabling direct P2P GPU copies with `NCCL_P2P_DISABLE=1` might help isolate a problem during troubleshooting, but it will drastically reduce intranode performance if left enabled. This is because it forces all intranode traffic to go through CPU host-staged intermediate buffers instead of GPU-direct NVLink links. This adds extra hops and CPU work that can increase latency from a few microseconds to tens of microseconds and cut bandwidth from hundreds of GB/s down to tens of GB/s.

Use `NCCL_P2P_DISABLE=1` only when diagnosing an issue. Remember to reenable P2P with `NCCL_P2P_DISABLE=0` (or remove the environment variable altogether) when you're done debugging.

Likewise, leaving shared-memory exchanges disabled (`NCCL_SHM_DISABLE=1`) would force NCCL to not use shared memory for intranode communication. This causes a fallback to network or host-mediated copies, which incurs additional kernel-driver overhead and context switches that further increase latency and throttle throughput.

Change performance-critical environment variables only briefly during debugging. And don't forget to set them back to production settings before returning to normal operations.

Another variable is `NCCL_DEBUG`. Setting `NCCL_DEBUG=INFO` or `DEBUG` is useful to log NCCL operations, as logs can hint at issues like falling back to Ethernet, for instance. But additional logging does incur overhead. Don't run at `DEBUG` level in production; use it when needed. For performance reasons, however, you may want to lower the setting to `WARN` (default) or even just `VERSION`. This would silence everything except the NCCL version but will make things difficult to debug when the time comes—and it will! To tune performance, some of the useful variables include the following:

NCCL_NSOCKS_PERTHREAD and NCCL_SOCKET_NTHREADS

As mentioned, if you have multiple NICs or very high network bandwidth, increasing these might help. If you have, say, two NICs, you might set `NCCL_NSOCKS_PERTHREAD=2` so that each thread handles two sockets for a total of four connections allowed. Defaults vary by platform and build, but what matters is that the product of `NCCL_NSOCKS_PERTHREAD` and `NCCL_SOCKET_NTHREADS` must not exceed 64 per NVIDIA guidance.

This 64 limit is NCCL's built-in maximum for the total number of TCP socket connections allowed per communicator. It's defined as the product of `NCCL_SOCKET_NTHREADS * NCCL_NSOCKS_PERTHREAD`. It's designed to bound CPU and network resource usage—and avoid exceeding operating-system and hardware limits.

NCCL_MIN_NCHANNELS and NCCL_MAX_NCHANNELS

These control how many subrings NCCL may use since NCCL splits data across multiple rings to use multiple NVLinks in parallel, if possible. It's recommended to leave these values as default. On GPU systems with NVSwitch, NCCL will auto-tune the number of channels based on topology and message size. Additionally, NCCL creates the same number of subrings as channels to match the number of concurrent hardware links. Each channel corresponds to one CUDA block used for communication. As such, a higher number of channels will require more GPU resources.

NCCL_TOPO_FILE

You can set this variable with a topology file for the system to guide NCCL to make wise decisions. This is useful in complex networks or cloud environments where NCCL might not detect the topology correctly. To capture what NCCL detects at runtime, set `NCCL_TOPO_DUMP_FILE` to an output path and inspect the generated file.

NCCL_MNNVL_ENABLE

Enable NVIDIA multi-node NVLink (MNNVL). This is designed for high-speed communication in systems with support for multi-node NVLink switches (e.g., NVL72 GB200/GB300).

NCCL_SHARP_DISABLE

This setting controls the usage of SHARP for in-network aggregation. We will discuss SHARP in a bit. By default, if SHARP is available and the job is configured to use it, NCCL will enable it. You can disable SHARP explicitly by setting `NCCL_SHARP_DISABLE=1` for A/B testing and troubleshooting.

In summary, use environment defaults unless you have evidence that a specific tunable will help. And if you do change these values, make sure to

document them and continuously monitor that their effects are still beneficial when upgrading to newer hardware and NCCL versions.

Pitfall #4: Verify CPU-GPU NUMA-node affinity for NCCL threads

NCCL launches background CPU threads for network polling and kernel dispatch. When you start multiple processes with `torch.multiprocessing` or Message Passing Interface (MPI), each process inherits a CPU affinity mask that may target all cores or only a subset if bound with tools such as `taskset` or `numactl`.

NCCL normally assigns its threads to the cores nearest the GPUs they serve, but if the process is pinned to a narrow set of cores, it may collapse all NCCL threads onto a single core and suffer from poor scheduling and low throughput. To prevent this, set the environment variable `NCCL_IGNORE_CPU_AFFINITY=1` so that NCCL ignores the inherited CPU affinity mask and freely spreads its worker threads across the cores in the local NUMA domain. The recommended approach is to bind each GPU process to the CPU cores for its NUMA domain and then set `NCCL_IGNORE_CPU_AFFINITY=1` so that NCCL can fine-tune thread placement within those cores.

Consider a compute node with two NUMA node domains and eight GPUs. If the GPUs 0–3 are attached to the first CPU and devices 4–7 are attached to the second CPU, you would bind ranks 0–3 to the first set of CPU cores and ranks 4–7 to the second set of CPU cores. Next, you would set `NCCL_IGNORE_CPU_AFFINITY=1` to ignore the inherited CPU affinity mask.

In practice, using `numactl` or setting `CUDA_DEVICE_ORDER` and `CUDA_VISIBLE_DEVICES` can help enforce this binding. PyTorch's launch utilities handle much of this automatically, but it's good to verify.

You can also specify an explicit topology file to further reduce latency and improve throughput. If you prefer not to pin processes manually, you can rely on MPI runtime binding or job scheduler options such as SLURM `--cpu-bind` to ensure each rank lands on the correct cores.

Pitfall #5: Resist the temptation to ignore NCCL warnings and errors

NCCL prints many logs if logging is judiciously enabled. In those logs, there could be warnings about falling back to slower PCIe bandwidth, for instance. These are important warnings to address.

If you see logs like "unable to enable P2P, falling back to copy," don't ignore these! They often indicate suboptimal conditions. If you see this warning, it means that NCCL was unable to establish direct GPU P2P between two GPUs. Perhaps because they're on different PCIe root complexes with no support. This means that data transfers will be much slower, as they must travel through host CPU memory buffers.

Warnings could prompt you to rearrange which GPUs are used in a process. The solution would be to ensure that GPUs that need to talk to one another are on the same NUMA node or using a different pairing schema. Another example is the `NCCL INFO NET/Socket: using Ethernet interface eth0` warning, which tells you which interface was picked. If that's not the highest-performing interconnect, you might need to set `NCCL_SOCKET_IFNAME` explicitly. For instance, you can set `NCCL_SOCKET_IFNAME=ib0` so the bootstrap handshake uses the intended fabric. You should track down why this is not being automatically set to the fastest interface. This is likely a larger issue.

Pitfall #6: NCCL communicator hangs, errors, or shuts down completely

Occasionally, if a process crashes or one GPU rank hits an error, NCCL communicators might hang the other ranks since the collectives won't complete. Unfortunately this is quite common in large-scale clusters given the relatively high frequency of GPU failures at scale, as described by Meta in Table 4-4.

Table 4-4. Root-cause categorization of unexpected interruptions during a 54-day period of Llama 3 405B pretraining (source: https://oreil.ly/z8QKu)

Component	Category	Interruption count	% of Interruptions
Faulty GPU	GPU	148	30.1%
GPU HBM3 memory	GPU	72	17.2%
Software bug	Dependency	54	12.9%
Network switch/cable	Network	35	8.4%
Host maintenance	Unplanned Maintenance	32	7.6%
GPU SRAM memory	GPU	19	4.5%
GPU system processor	GPU	17	4.1%
NIC	Host	7	1.7%
NCCL watchdog timeouts	Unknown	7	1.7%
Silent data corruption	GPU	6	1.4%
GPU thermal interface and sensor	GPU	6	1.4%
SSD	Host	3	0.7%
Power supply	Host	3	0.7%
Server chassis	Host	2	0.5%
IO expansion board	Host	2	0.5%
Dependency	Dependency	2	0.5%
CPU	Host	2	0.5%
System memory	Host	2	0.5%

Enabling `NCCL_ASYNC_ERROR_HANDLING=1` can improve resiliency by allowing NCCL to abort on errors asynchronously, but this may incur a slight overhead. PyTorch sets this by default in recent versions when you use

`init_process_group` . However, it's a good idea to keep explicitly setting this value for clarity and reproducibility.

Never rely on default values. Always be explicit! Default values can sometimes change from version to version—and they are extremely hard to debug when they change. Setting these values explicitly during initialization avoids version-dependent behavior.

It's important to treat NCCL as a high-performance engine that just works. But be mindful of how you initialize and use NCCL. Initialize once, pin CPUs appropriately, use the environment variable to adjust affinity if needed, and be cautious with environment variables and their defaults.

Also, one should always review NCCL release notes when upgrading. New versions often bring optimizations—especially when new networking hardware emerges. And always test after upgrading NCCL as default settings and performance might change. Typically, NCCL performance improves with each new version. But default values may sometimes change, which would require retuning your system if you have not explicitly pinned the NCCL environment variables.

Profiling and Debugging NCCL

NCCL supports asynchronous error handling and failover for cases like network errors. To enable asynchronous error handling, set the environment variable `NCCL_ASYNC_ERROR_HANDLING=1` . And when debugging NCCL, make sure to also enable `NCCL_DEBUG=WARN` or `INFO` . This way, you can check for common issues like unmatched ranks or socket misconfigurations.

Also available to debug NCCL is the NCCL profiler plugin API. This API lets you monitor the internal timeline of GPU communications and pinpoint any lagging device or bottleneck in the system. The NCCL profiler plugin API is designed to address performance issues that become increasingly difficult to diagnose as GPU clusters scale up.

The NCCL profiler plugin can be dynamically loaded via the `NCCL_PROFILER_PLUGIN` interface and integrated by tools that support it. The `NCCL_PROFILER_PLUGIN` environment variable governs the loading and initialization of this plugin in a manner similar to other NCCL plugins.

NVIDIA created this flexible API to simplify the integration of third-party profiling tools (e.g., PyTorch Kineto) with NCCL and ensure that complex communication activities are monitored and captured in a clear, hierarchical, and low-overhead manner during runtime execution. PyTorch's Kineto can also gather NCCL activity using CUPTI and NVTX if the NCCL plugin is not enabled.

The NCCL profiler plugin is bundled with NVIDIA's tools and third-party profilers like the PyTorch/Kineto profiler. Use it to give timeline views of all-reduce operations.

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

The NCCL profile plugin API defines five function callbacks. The `init` callback sets up the plugin by providing an opaque context and establishing which events should be profiled. The `startEvent` callback receives an event descriptor from NCCL and allocates a new event object, returning an opaque handle that NCCL uses for further operations.

The `stopEvent` callback marks the completion of an event so that its resources can be recycled. The `recordEventState` callback allows the plugin to update events as they transition through different states. The `finalize` callback releases all resources associated with the profiler context once profiling is complete.

In-Network SHARP Aggregation

When using advanced network hardware that supports in-network computing, such as NVIDIA's Scalable Hierarchical Aggregation and Reduction Protocol (SHARP), additional performance gains can be realized by offloading parts of collective operations to the network itself.

SHARP is an InfiniBand in-network reduction technology used with Quantum-class InfiniBand switches using the NCCL-SHARP plugin. In

NVLink domains, the analogous capability is NVLink SHARP (NVLS), which offloads collectives within the NVSwitch fabric. In modern NVLink Switch domains (e.g., NVL72), NVLS accelerates collectives and enables efficient all-to-all and broadcast across the domain (e.g., 72-GPU NVL72 domain).

In practical terms, SHARP enables collectives such as all-reduce to be partially computed by the network fabric. As data from multiple GPUs flows into the switch, it will reduce/aggregate (e.g., sum) the data and share the partially reduced result. This saves each GPU from having to redundantly transfer many intermediate results between other GPUs. This reduces the overall amount of data that each GPU must handle, which reduces latency for large MPI and NCCL collectives.

Specifically, for a ring reduce-scatter operation, each GPU normally receives $B\,(n-1)/n$ bytes across $(n-1)$ hops. With in-network reduction, the switch aggregates and returns only B/n to each GPU. this results in a $1/(n-1)$ per-endpoint receive versus full-ring.

For an all-gather operation, hardware multicast with NVLS lets each GPU send its B/n segment once while the network replicates it. This reduces the sender's volume by $1/(n-1)$ versus the full ring. And when you overlap a multicast all-gather with an in-network reduce-scatter, the end-to-end phase time can drop by ~1/2 for bandwidth-bound phases since the network performs the aggregation and replication work instead of the endpoints. In this case, the effective wall time for a shard exchange is the max of the two operations instead of the sum.

In short, NVLS results in less data per endpoint and fewer serialized hops. This produces higher effective bandwidth and shorter stalls during sharded training/inference phases.

All-gather has no arithmetic reduction, so NVLS mainly helps by performing multicast replication. The speedup depends on topology and message size, but it's smaller than the performance gains for NVLS with all-reduce and reduce-scatter.

NCCL can offload collective operations such as all-reduce to SHARP-enabled InfiniBand fabrics by using the NCCL RDMA SHARP plugin with SHARP

firmware on the switches and a SHARP Aggregation Manager running on a management server alongside the Subnet Manager. Additionally, each host must load the GPUDirect RDMA kernel module. Once the fabric and hosts are configured and the NCCL RDMA SHARP plugin is selected, NCCL can offload eligible collectives to SHARP. SHARP's performance impact can be substantial. In some cases, NVIDIA reports 2× to 5× speedups for all-reduce on large-scale AI systems using SHARP.

The gains from SHARP are more pronounced at scale with many GPUs and compute nodes in which the network is usually the bottleneck. On a smaller cluster, say two to four GPU compute nodes, you might not notice as much of a performance improvement. But on 32 nodes, SHARP can reduce collective latency significantly by cutting down the number of overall communication steps.

SHARP is not enabled by default. It must be configured with a plugin selection or policy. You can disable it with `NCCL_SHARP_DISABLE=1` for A/B testing to verify the performance impact. However, it's recommended to keep it enabled to improve all-reduce latency at scale.

Using SHARP doesn't typically require code changes. One can verify if SHARP is in use by looking at the NCCL logs (`NCCL_DEBUG=INFO`). The logs will mention SHARP if it's being used. There are also diagnostic tools (`ibv_devinfo` etc.) to check if a device supports SHARP.

In summary, SHARP moves reduction computation into the network. This further demonstrates how modern system design blurs the boundaries between computing and communication. For the performance engineer, if your cluster is running on a high-end InfiniBand network, it's worth checking that SHARP is enabled and utilized. It can provide a "turbo button" to provide faster scaling and improved efficiency for massive all-reduce operations. SHARP complements NCCL's GPU-centric optimizations with network-centric optimizations.

It's worth noting that as of this writing, SHARP is primarily an InfiniBand technology. While NVIDIA's Spectrum-X Ethernet platform improves all-reduce performance with congestion control and adaptive routing, as of this writing, it still does not expose switch-resident reduction engines similar to

SHARP with InfiniBand. Public material highlights end-to-end telemetry and congestion control for improved NCCL performance across large domains; it does not expose switch-resident reduction engines analogous to SHARP.

SHARP can sometimes incur extra memory overhead on the switches. And since there's a finite amount of buffer for performing reductions on the switch, very large collectives with many MB or GB messages might fall back to regular methods if they exceed hardware limits.

It's recommended to continuously monitor the NCCL logs and set up an alert if it starts falling back to non-SHARP aggregations due to memory pressure over time.

Persistent NCCL User Buffers and Zero-Copy Registration

NCCL supports user buffer registration, which lets collectives operate directly on your tensor buffers without requiring internal staging. This reduces copies and internal channel pressure.

Persistent NCCL user buffers are essential to achieving the best paths using SHARP for both on-node (NVLS) and off-node (InfiniBand) scenarios. Zero-copy registration can accelerate collectives and reduce SM/channel usage.

You can register and deregister persistent NCCL user buffers using explicit `ncclCommRegister()` and `ncclCommDeregister()`. If any rank in a communication uses registered buffers, all rank must use them. Furthermore, for some algorithms the offset from the buffer head must match across the ranks.

NVIDIA's NIXL and Disaggregated Inference

While NCCL excels at many-to-many group communication patterns often used during model training, modern AI inference at scale has introduced new

communication needs. NVIDIA's NIXL is an open source, high-throughput, low-latency, point-to-point communication library released in early 2025.

NIXL was designed specifically to accelerate large-scale LLM distributed and disaggregated inference. We'll cover how disaggregated inference separates different stages of inference into separate workers. Disaggregating inference uses NIXL for fast data exchange between these stages across GPUs with minimal latency and overhead.

Disaggregated inference and NIXL are best practices for serving giant models efficiently across a cluster of nodes.

NIXL is a core component of NVIDIA's open source Dynamo inference engine. NIXL streamlines one-to-one and one-to-few data transfers such as moving a key-value (KV) cache (shared by the disaggregated stages) with minimal latency and overhead. It complements NCCL, which is mainly used for many-to-many collectives.

NIXL has a consistent asynchronous API for moving data across GPUs, CPUs, SSDs, and shared network storage. It always picks the fastest path for the placement of each cache chunk of data being moved. This hierarchy is shown in Figure 4-5 in the context of NVIDIA Dynamo's KV Cache Manager, which uses NIXL to choose the fastest path available for each KV cache transfer.

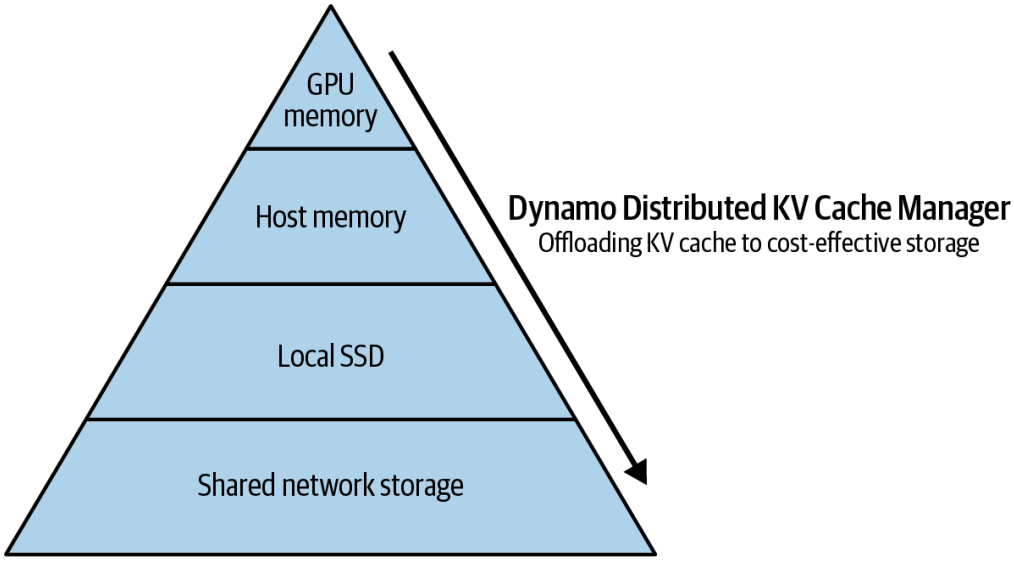

Figure 4-5. NVIDIA Dynamo Distributed KV Cache Manager offloads less frequently accessed KV cache to more economical memory hierarchies (source: *https://oreil.ly/nsxdl*)

When scaling LLM inference, it's important to efficiently transfer large data caches (e.g., transformer's attention KV cache) between peers in a cluster of GPUs, CPUs, compute nodes, and racks. For example, with NIXL, an inference engine can offload a large (e.g., 100 GB) KV cache from a GPU to a peer using NVLink/InfiniBand with minimal overhead. This frees up the GPU to handle new requests. This is critical when serving LLMs with large context windows.

NIXL leverages GPUDirect RDMA to move data directly between GPUs across nodes, entirely bypassing host memory. In practice, the RDMA-capable NIC (or DPU) performs the transfer directly between GPU memories. This is why latency is so low. The CPU is not involved in the data path.

NCCL is still used for synchronized collective operations, but NIXL is focused on efficient one-to-one or one-to-many data transfer scenarios common in LLM inference systems. NVIDIA Dynamo uses NIXL extensively for its disaggregated inference stages of prefill and decode, as we'll cover next.

NCCL remains the standard for many-to-many collective operations common in large-scale training such as all-reduce. NIXL, however, targets one-to-one or one-to-few data transfers that are common in large-scale inference such as moving KV cache data.

NIXL's use in NVIDIA's Dynamo inference framework demonstrates NIXL's throughput boost in multinode LLM serving scenarios. NIXL complements—not replaces—NCCL for high-performance inference pipelines.

Separate Prefill and Decode Inference Stages

We will dive deeper into the performance details of a highly tuned inference system in a later chapter, but it's important to understand a bit of context before going further with NIXL. The inference path of a transformer-based model is actually split into two different stages: prefill and decode.

The first stage, prefill, is often compute bound as it uses many matrix multiplications to build the KV cache from the incoming request data (aka *prompt*). The second stage, decode, is often memory-throughput bound, as it needs to gather the model weights from GPU HBM memory to calculate the next set of tokens (aka *completion* or *response*).

This prefill/decode split is implemented in common inference engines vLLM, SGLang, and NVIDIA's Dynamo and TensorRT-LLM. The *prefill* (prompt ingestion) creates the KV cache, and the *decode* (generation) uses this cache. NIXL specifically accelerates the transfer of the KV cache between nodes in this workflow. Figure 4-6 compares the traditional "monolithic" serving model to the "disaggregated" serving model in which two stages run on different GPU-based compute nodes to increase scale and maximize throughput.

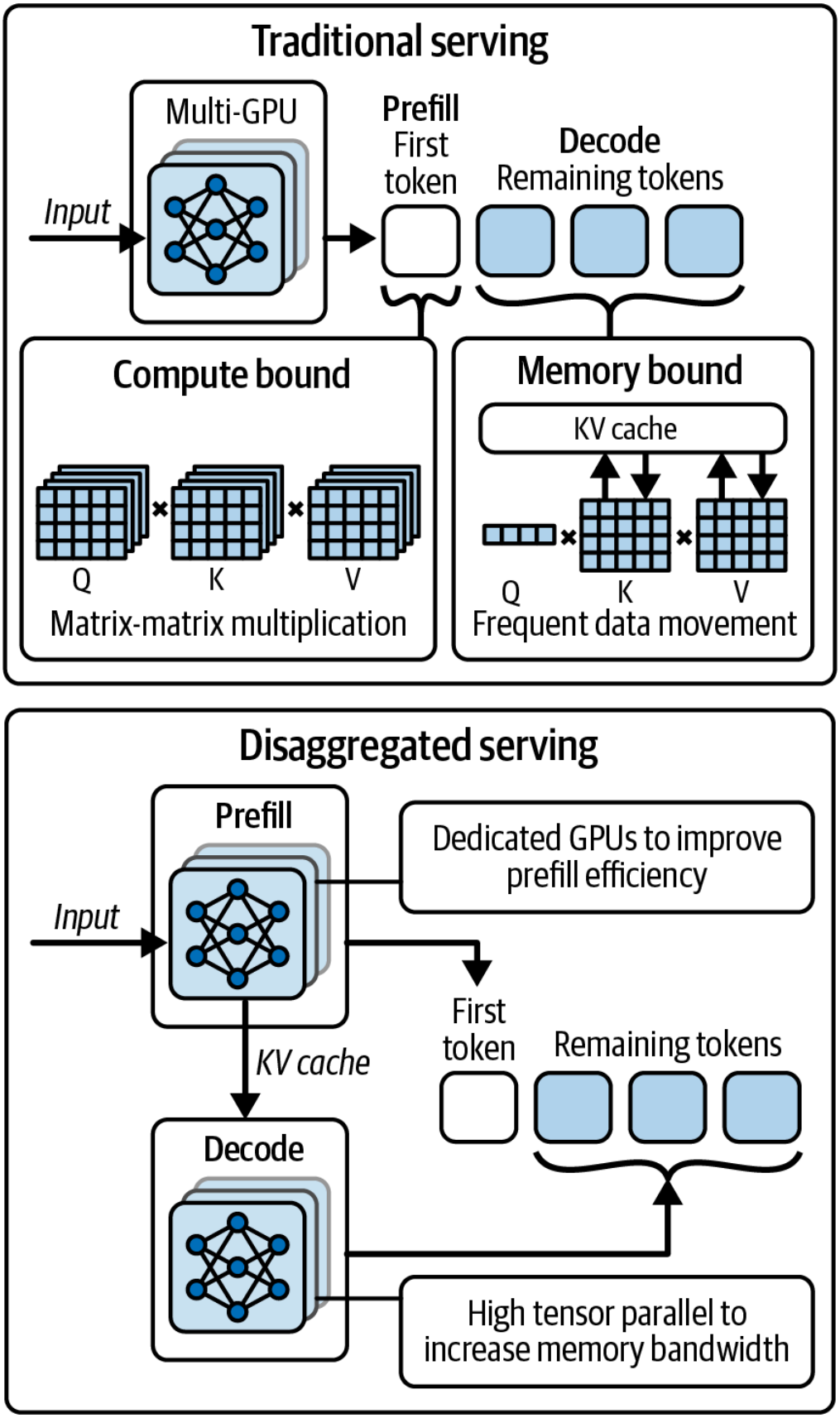

Figure 4-6. Disaggregated serving separates prefill and decode stages onto different GPU clusters (source: https://oreil.ly/nsxdl)

Here, the traditional setup is shown on the top, in which each GPU node handles both the prefill (compute-bound) and decode (memory-bound, I/O-bound) phases. On the bottom, the disaggregated serving configuration places the prefill workers in the GPU cluster and the decode workers in another GPU cluster.

A GPU in the prefill cluster generates the KV cache for the input sequence and uses NIXL to transfer it to a GPU in the decode cluster. This specialization produces higher overall throughput and advanced scaling configurations.

In such cases, the KV cache, which can run into tens of gigabytes in a long prompt, must move seamlessly from one processing unit to another in near-real time. This way, the text generation happens at speeds that are unnoticeable to end users.

Traditional methods that pass the data through CPU memory or even storage fall short in meeting the required pace and low-latency experience. NVIDIA created NIXL to tackle this exact scenario. NIXL allows multinode inference to scale without being bottlenecked by interconnect latency.

What we really want is a high-bandwidth GPU-to-GPU direct transfer between components. And we want this communication to overlap with computation. This way, the destination GPUs can start computing the next token while they're receiving the KV cache for the next set of input tokens from the source GPUs.

NIXL provides a direct channel for transferring data from one GPU to another or a small group of GPUs across compute nodes and even across racks. The system looks at the available pathways and always selects the one that gets the data there the quickest.

This intelligent routing is analogous to NCCL's path selection but optimized for inference patterns, including one-to-one, large-message transfers. For example, in a GB200/GB300 NVL72 rack, NIXL leverages the NVSwitch network first, whereas across NVL72 racks, it will automatically switch to InfiniBand or Ethernet RDMA, depending on what is supported.

In short, NIXL automatically picks the fastest lane, whether it's NVLink/NVLink-C2C on the same board, NVSwitch across a rack domain, InfiniBand/RoCE between racks, or even direct NVMe storage access.

Intelligent Interconnect Routing for KV Cache Transfers

Traditionally, one might transfer this data from GPU to GPU through a CPU, but this would be too slow, as we've already discussed. Another option to improve performance is to require that the source and destination GPUs are on the same compute node, but this limits our scaling flexibility. NIXL was created to address this issue. It was designed for direct GPU-to-GPU transfers of large payloads like the KV cache across GPUs, compute nodes, and racks, if necessary.

NIXL operates at high bandwidth and overlaps communication with computation as much as possible. This allows the destination GPUs to start generating the next set of tokens while they are receiving the KV cache from the source GPU.

Additionally, NIXL is interconnect-agnostic. It will use NVLink if GPUs are in the same compute node, NVSwitch if in the same compute node, InfiniBand or Ethernet with RDMA across nodes, or even PCIe or NVMe if needed. Similar to NCCL, NIXL will always select the fastest interconnects to route the data transfer. It also supports transferring to and from different memory tiers in a unified way across GPU HBM, CPU DRAM, and even to NVMe SSD!

NIXL Asynchronous API with Callbacks

From a developer's perspective, NIXL offers a straightforward API. You post a transfer request with a pointer to the data and a destination—either GPUs, CPUs, or storage targets like Amazon S3. NIXL will transfer that data as fast as possible.

For instance, a NIXL transfer request could send a KV cache segment to another GPU, a CPU host memory buffer, or even an object storage service. And it can do this all within the same API, as shown in Figure 4-7.

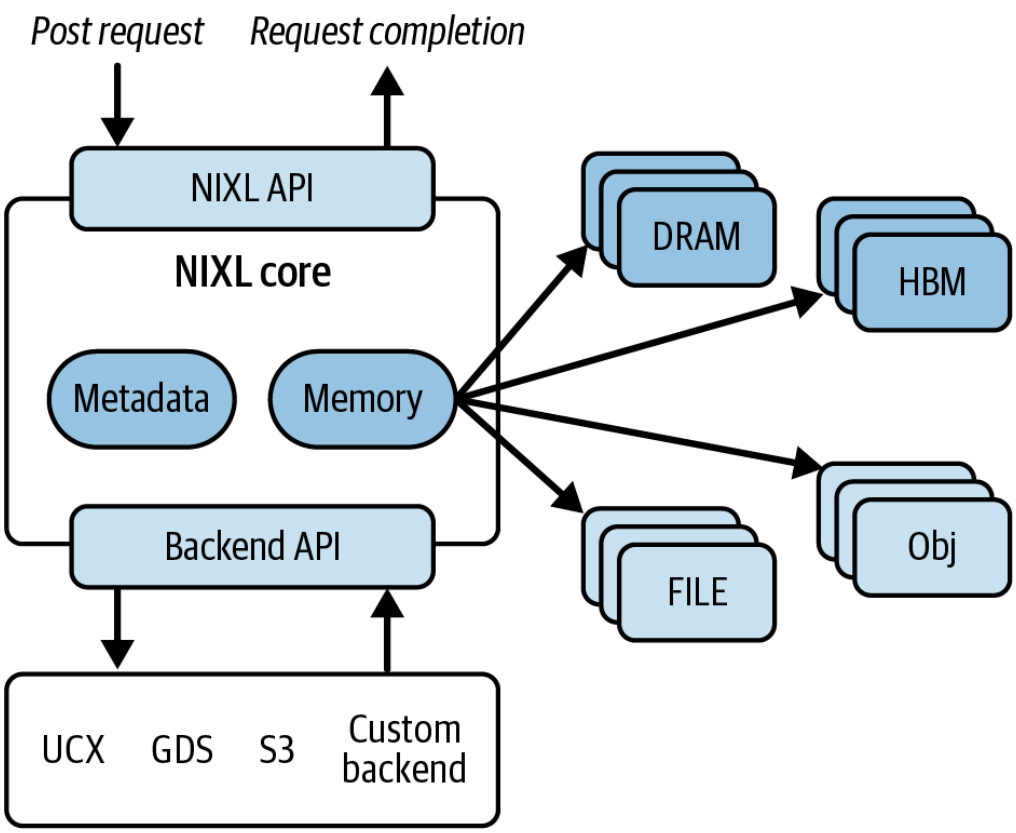

Figure 4-7. NIXL architecture (source: *https://oreil.ly/nsxdl*)

This modular design means that NIXL can adopt future transports as well. For example, it can incorporate upcoming protocols or faster storage-class memory without changing the user-facing API. Under the hood, NIXL coordinates the data movement using whatever backend is appropriate.

NIXL efficiently moves data between different tiers such as GPU HBM, CPU memory (DRAM), file storage (NVMe SSD), and object storage. It provides a single, unified API that automatically selects the fastest transport (e.g., GPU → GPU over NVLink, GPU → NVMe SSD with GDS, or NVSwitch fabric.) This way, you always get near line-rate performance when offloading KV cache segments.

Under the hood, NIXL hides all of this complexity. You register memory with `registerMem`, obtain transfer descriptors with `trim`, prepare a nonblocking request with `prepXfer`, and submit it with `postXfer`. NIXL chooses whether to perform a direct PCIe or NVLink copy, an RDMA transfer, or a storage path such as GPUDirect Storage.

The NIXL library is nonblocking and returns a request handle that you poll with `checkXfer` to detect completion. This pattern overlaps communication and computation with minimal CPU overhead. The NIXL nonblocking API allows downstream kernels to consume incoming data without blocking on the transfer itself. For example, the destination GPU can begin consuming

received KV cache chunks as soon as they arrive—even while the remaining chunks are still in flight.

A `nixlAgent` is NIXL's core transfer object. It encapsulates the endpoint configuration, memory registrations, and backend selection. It also manages metadata, connection information, and asynchronous transfer requests to and from other agents.

You need two agents for a transfer because each `nixlAgent` instance represents one endpoint in the transfer. The source agent (`agentSrc`) encapsulates the context, memory registrations, and backends for the origin of the data. The destination agent (`agentDst`) does the same for the receiver side.

With one agent per endpoint, `agentSrc` and `agentDst` , NIXL negotiates the optimal path between these endpoints and manages their independent resources and request lifecycles. These NIXL agents are shown in the following code to transfer data between a source and destination GPU:

```
// NIXL 0.5.x style example: nonblocking VRAM->VRAM tra
#include <nixl.h>
#include <nixl_types.h>

#include <cuda_runtime.h>
#include <iostream>
#include <thread>
#include <vector>
#include <cassert>
#include <cstdint>

int main() {
    // 1) Configure agents. Prefer UCX for GPU<->GPU,
    // allow GDS if you later target storage.
    nixl_agent_config cfg{};
    cfg.backends = {"UCX"};  // {"UCX","GDS"} if you al
    cfg.thread_safe = true;  // thread-safe mode added

    // 2) Create source and destination agents
    nixlAgent agentSrc("srcAgent", cfg);
    nixlAgent agentDst("dstAgent", cfg);

    // 3) Allocate simple test buffers on the same GPU
    int deviceId = 0;
```

```
cudaSetDevice(deviceId);
const size_t bytes = 1 << 20; // 1 MiB

void* d_src = nullptr;
void* d_dst = nullptr;
cudaMalloc(&d_src, bytes);
cudaMalloc(&d_dst, bytes);

// 4) Build registration descriptors for VRAM
//    Each descriptor uses an address, length, and
nixl_desc_t srcDesc{};
srcDesc.addr      = reinterpret_cast<uintptr_t>(d_s
srcDesc.len       = bytes;
srcDesc.devId     = deviceId;
srcDesc.seg       = VRAM_SEG;

nixl_desc_t dstDesc{};
dstDesc.addr      = reinterpret_cast<uintptr_t>(d_c
dstDesc.len       = bytes;
dstDesc.devId     = deviceId;
dstDesc.seg       = VRAM_SEG;

std::vector<nixl_desc_t> srcList{srcDesc};
std::vector<nixl_desc_t> dstList{dstDesc};

// 5) Register memory with each agent and trim to x
auto srcRegs = agentSrc.registerMem(srcList);
auto dstRegs = agentDst.registerMem(dstList);

auto srcXfer = srcRegs.trim();  // metadata-free de
auto dstXfer = dstRegs.trim();

// 6) Prepare a WRITE from srcAgent->dstAgent, ther
nixlReqH reqHandle = nullptr;

// prepare + post
if (agentSrc.prepXfer(NIXL_WRITE, srcXfer, dstXfer,
  != NIXL_SUCCESS) {
    std::cerr << "prepXfer failed\n";
    return 1;
}
if (agentSrc.postXfer(NIXL_WRITE, srcXfer, dstXfer,
  != NIXL_SUCCESS) {
    std::cerr << "postXfer failed\n";
    return 1;
}
```

```
        std::cout << "Transfer posted — doing other work...

        // 7) Poll for completion (replaces deprecated getN
        nixl_status_t st;
        do {
            st = agentSrc.checkXfer(reqHandle);
            if (st == NIXL_INPROGRESS) std::this_thread::yi
        } while (st == NIXL_INPROGRESS);

        if (st != NIXL_SUCCESS) {
            std::cerr << "Transfer completed with error: "
            agentSrc.releaseReqH(reqHandle);
            agentSrc.deregisterMem(srcRegs);
            agentDst.deregisterMem(dstRegs);
            cudaFree(d_src);
            cudaFree(d_dst);
            return 1;
        }

        std::cout << "Transfer completed!\n";

        // 8) Cleanup
        agentSrc.releaseReqH(reqHandle);
        agentSrc.deregisterMem(srcRegs);
        agentDst.deregisterMem(dstRegs);

        cudaFree(d_src);
        cudaFree(d_dst);
        return 0;
    }
```

Here, the NIXL agents are initialized with names and a configuration. Memory is registered with `registerMem` and trimmed to transfer descriptors with `trim`. A nonblocking write from `srcAgent` to `dstAgent` is prepared with `prepXfer` and submitted with `postXfer`. The program continues doing other work while the transfer is in flight. Completion is detected by polling the request handle with `checkXfer` while yielding the thread if the request is in progress. After success, the handle is released with `releaseReqH` and the registrations are deregistered.

Internally, NIXL uses Unified Communication X (UCX), an HPC library that provides a unified API over various interconnects that NIXL leverages for low-level transport, including InfiniBand, TCP, shared memory, etc. NIXL

also uses GPUDirect RDMA and InfiniBand GPUDirect Async (IBGDA) to allow the GPU to initiate transfers without CPU involvement. This is an important optimization, as in older systems, the CPU might need to kick off the transfer—even if the data path is purely RDMA. IBGDA offloads that initiation to the GPU/NIC, which further reduces latency.

Another interesting feature of NIXL is that it avoids unnecessary copies such as staging buffers. For example, if data is in pageable CPU memory, it would choose to pin the data to prevent it from being paged out. But if the data is in GPU memory, it would just send the data directly. In other words, NIXL tries to avoid staging buffers that copy to an intermediate host buffer before transferring the data to the destination.

KV Cache Offloading with NIXL

The design motivation for NIXL is closely related to best practices for handling large memory for LLM inference. If GPUs don't have enough memory to hold the entire KV cache for a long sequence or multiturn conversation, NIXL allows the inference server (e.g., NVIDIA Dynamo) to offload the KV cache to CPU memory—or even NVMe SSD—and bring it back as needed.

NIXL, in conjunction with the KV Cache Manager in Dynamo, for instance, can manage this transfer hierarchy efficiently. Consider NVIDIA's Grace Hopper and Grace Blackwell Superchips with a huge amount of unified CPU and GPU memory shared over the fast NVLink interconnect (see Figure 4-8).

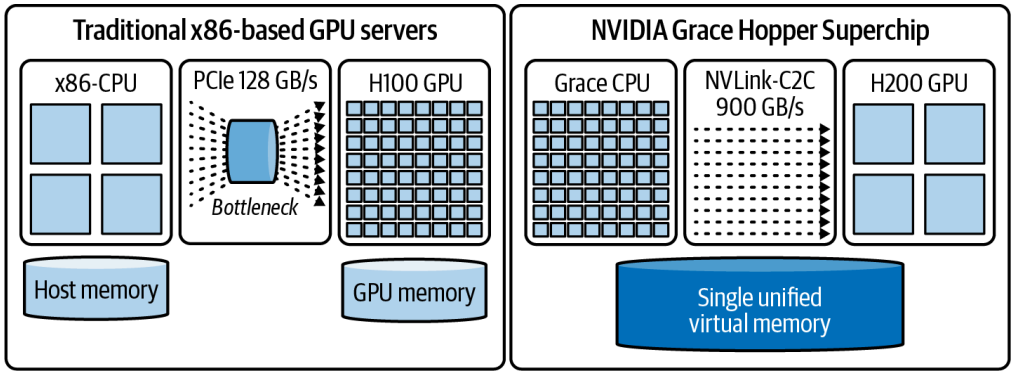

Figure 4-8. ARM-based Grace Hopper Superchip architecture leverages the 900 GB/s NVLink-C2C and overcomes traditional PCIe bottlenecks (source: *https://oreil.ly/zf6rF*)

The inference server can quickly offload a large KV cache to the large CPU memory to free up the limited GPU HBM. This yields a huge boost in inference performance. Specifically, a PCIe-based x86 + H100 system can

improve time-to-first-token (TTFT) latency by as much as 14× for long input sequences compared to recomputing the cache. This speedup is shown in Figure 4-9.

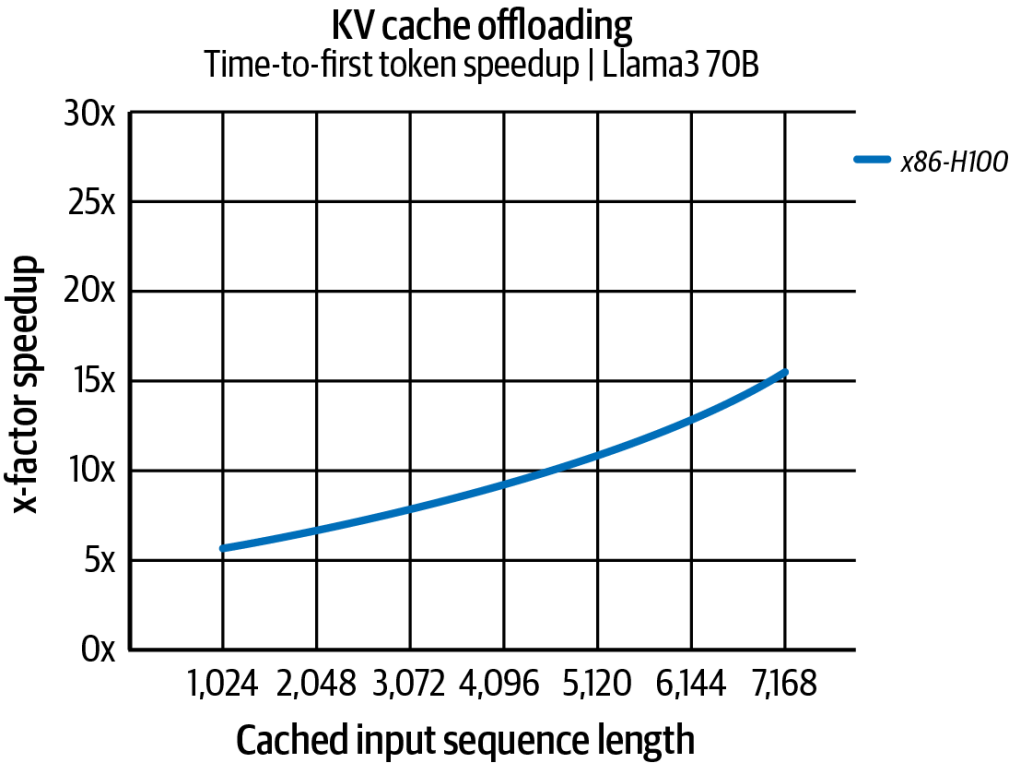

Figure 4-9. Measure 14× TTFT speedup with KV cache offloading for an x86-based NVIDIA H100 GPU system on large input sequence lengths compared to recalculating it from scratch (source: *https://oreil.ly/zf6rF*)

Furthermore, with its 900 GB/s NVLink-C2C interconnect, the ARM-based Grace Hopper Superchip delivers 2× faster TTFT latency compared to the non-superchip, x86-based H100 version described previously. This speedup is shown in Figure 4-10.

These are impressive gains enabled by codesigning NIXL software with NVIDIA superchip hardware. And NIXL, designed exactly with those numbers in mind, makes offloading the KV cache a viable option by keeping transfer costs low. KV cache offloading is a key part of large-scale inference deployments, as we'll see in an upcoming chapter—especially for massive LLMs where memory capacity is a limiting factor.

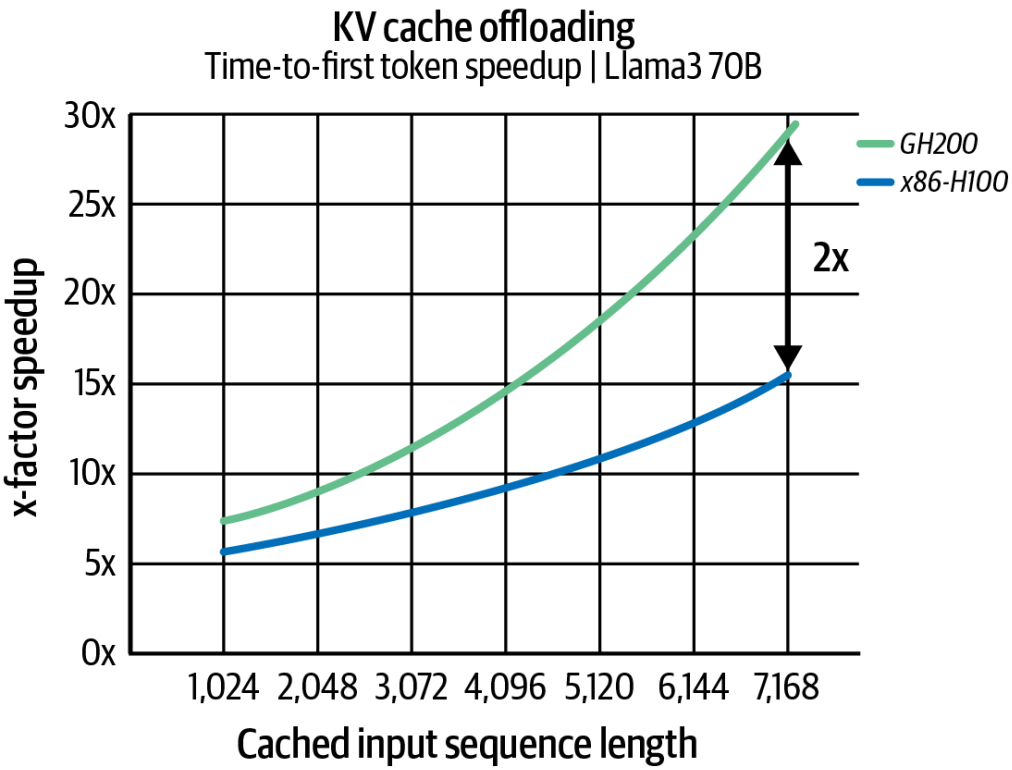

Figure 4-10. 2× speedup in TTFT for the ARM-based Grace Hopper Superchip compared to the x86-based H100 GPU system due to KV cache offloading with the 900 GB/s NVLink-C2C interconnect (source: https://oreil.ly/zf6rF)

As models get bigger and workloads get more complex, having a library like NIXL to efficiently move large blobs of data asynchronously is critical. For performance engineers, if your use case involves moving large amounts of data between stages (e.g., pipeline parallelism) and other components (e.g., GPUs or storage) in a system, consider whether NCCL is sufficient or whether a specialized solution like NIXL is an option to optimize that flow of data.

NIXL and High-Performance Inference Systems Like NVIDIA Dynamo

The impact of NIXL on performance is huge for distributed inference systems like NVIDIA Dynamo, also released in early 2025. According to NVIDIA's internal testing, the open source NVIDIA Dynamo framework used NIXL to achieve up to 30× higher inference throughput on a ~680B-parameter LLM when using a 72-GPU Blackwell NVL72 rack.

What was once a major latency barrier—shifting many gigabytes of context data between nodes—is now a relatively quick, asynchronous operation under NIXL. We will cover NVIDIA Dynamo, TensorRT, vLLM, and respective model inference optimizations in depth in a later chapter.

NCCL Versus NIXL

It's important to note that NIXL is not a replacement for NCCL but a complement. NCCL still handles synchronized collectives for GPUs working on a single task/stage in parallel, such as an all-reduce split across multiple GPUs. NIXL, on the other hand, performs asynchronous data transfers between tasks/stages—or between distinct components (e.g., GPUs, CPUs, storage) in a distributed system. Table 4-5 shows a comparison of NCCL versus NIXL.

Table 4-5. Comparison of NCCL and NIXL

Aspect	NCCL (collective communication)	NIXL (point-to-point communication)
Primary use case	Many-to-many collectives (e.g., all-reduce, all-gather) for tightly coupled GPU groups in training.	One-to-one or one-to-few transfers (e.g., sending large tensors or caches) for distributed inference or pipelining.
Communication pattern	Synchronized collective operations—all participants must reach the call (barrier semantics).	Asynchronous send/receive—one initiator, one or multiple targets (supports one-way data movement).
Overlap with compute	Can overlap to some extent (e.g., overlap backward compute with all-reduce in DDP) using separate CUDA streams.	Designed for maximal overlap—transfers run fully in parallel with compute, with polling notification to detect completion.
Topology awareness	Yes—auto-detects topology, uses rings/trees and NVLink/NVSwitch optimally for collectives.	Yes—interconnect-agnostic; automatically uses NVLink, NVSwitch, PCIe, InfiniBand/RDMA, or GDS, depending on source-dest locations.
Data scope	Typically small-to-medium tensors (e.g., gradients) that need aggregation across all GPUs.	Optimized for large data blobs (e.g., hundreds of MBs or more, like LLM KV cache or model shards) that need fast point-to-point transit.
Integration	Integrated in training frameworks (PyTorch DDP, Horovod, etc., call NCCL under the hood).	Provided as an open source library used by NVIDIA Dynamo. It is developed in the Dynamo project. The developer calls the NIXL API to send/receive as needed in the inference server or custom code.

Aspect	NCCL (collective communication)	NIXL (point-to-point communication)
Example	All-reduce of 100 MB of gradients across 8 GPUs in parallel.	Send a 1 GB KV cache from GPU 0 to GPU 1 (or to CPU memory or an NVMe SSD) during an inference pipeline.

While NCCL does support peer-to-peer `send()` / `recv()` operations, it's best suited for collective operations in synchronous training environments. NIXL, on the other hand, addresses the needs of asynchronous, point-to-point data transfers common in large-scale inference and pipeline parallelism. For instance, NVIDIA's Dynamo inference server, discussed in a later chapter, uses NIXL to orchestrate KV cache movement across inference components, including GPUs, CPUs, and SSDs.

In summary, NCCL is designed to maximize collective throughput across multiple GPUs both within a single host and across nodes. It automatically selects hardware topology-aware ring and tree communication algorithms that fully saturate the given PCIe, NVLink, InfiniBand, and Ethernet links. NCCL is typically associated with ultrascale model training workloads. NIXL builds on NCCL's high-performance principles and orchestrates asynchronous, hardware-agnostic point-to-point transfers across GPUs, CPUs, and storage devices. NIXL was designed for large-scale distributed inference workloads, including super fast KV cache data transfers.

Key Takeaways

With careful engineering—and using the techniques described in this chapter—you can often reach performance at or near the physical "speed of light" hardware limits. Here are some key lessons to remember when tuning your network layer:

Topology matters

The interconnects between nodes (internode, InfiniBand) and within nodes (intranode, NVLink/NVSwitch) influence the optimal communication strategy. Consider hierarchical approaches for multinode and multi-GPU configurations. Always ensure that you're using the fastest interconnect available—and not accidentally sending

data over slow paths due to a misconfiguration or unexpected default value! In practice, check NCCL's behavior and utilize features like SHARP for large-scale, in-network aggregations/reductions if available.

Tune the environment and system

Sometimes a single environment variable or OS setting can boost throughput. For instance, one can increase NIC buffers, enable/disable NCCL features and logging, and pin CPUs correctly. Performing system optimizations at the OS and driver levels (e.g., IRQ affinity) can help remove bottlenecks, as described in Chapter 3.

Utilize the latest hardware innovations

New hardware like NVIDIA Grace Hopper and Grace Blackwell Superchips provide massive CPU memory and fast CPU-GPU interconnects. Use them for things like hosting large datasets, splitting your data, partitioning your model, and offloading large KV caches to the CPU. In-network computing like SHARP can accelerate collective operations by 2×–5×—especially at scale. Stay informed on these new compute and networking hardware innovations because they will change the optimal configuration with each new generation.

You want to saturate your GPUs to the point where they're computing 100% of the time while simultaneously communicating in the background. You also want to saturate your network links with useful data. And you want to keep your disks streaming data at full throttle. All of this should happen together in perfect harmony.

Achieving all of this requires iterative tuning and validation, as well as some trade-offs like more memory usage and more code complexity. But this kind of tuning pays off with faster model training and inference—as well as better overall utilization of expensive infrastructure.

Conclusion

The evolution of high-performance, distributed, and multi-GPU communication and storage systems represents the foundation for tuning large, complex AI systems. By utilizing specialized libraries such as NCCL for collective operations, NIXL for efficient inference data transfers, and

RDMA for ultra-low-latency communication, AI systems can significantly reduce bottlenecks and increase performance.

The integration of smart networking hardware like NVSwitch and SHARP-enabled InfiniBand switches translates directly into higher training and inference performance. Likewise, keeping software up-to-date is key, as newer versions of CUDA and PyTorch come with these optimizations built in for the latest GPUs and networking technology (e.g., SHARP). Utilizing NVIDIA Dynamo, vLLM, and similar serving frameworks can help deploy these improvements easily for inference workloads.

Ultimately, this chapter highlights that no single component can provide peak performance alone. It is the careful coordination and codesign of high-speed communication, efficient data handling, and system-wide tuning that leads to scalable and robust AI systems.

For performance engineers, the lesson is that fast data movement is as critical as raw compute power. The fastest GPU in the world provides little benefit if it’s constantly waiting for data from a CPU or another GPU.

In the next chapter, we will explore GPU-based storage strategies and optimizations. Complementing networking protocols and libraries like RDMA, NCCL, and NIXL, GDS, and highly efficient input pipelines are part of the holistic approach to keeping the GPUs fed with continuous work.