CHAPTER 15 · PART V — INFRASTRUCTURE AT SCALE

# Storage & I/O: Feeding 10,000 GPUs Without Starving Them

*The Unsung Bottleneck — When the Network Is Fast but the Disk Is Slow*

**Part V:** Infrastructure at Scale **Depth:** Principal Level

**Cross-refs:** Ch 10 (Training), Ch 11 (KV Cache), Ch 14 (Networking), Ch 16 (Cluster)

> *"A 10,000-GPU cluster consuming 100 GB/s of training data needs a storage system that delivers 100 GB/s reliably, every second, for weeks. A single slow checkpoint can stall $2M worth of GPU time."*
>
> — AI Infrastructure Operations Principle

Storage is consistently the most underestimated component in AI infrastructure design. Engineers spec GPUs and networking carefully — and then discover that a poorly configured NFS mount causes GPUs to idle 15% of the time waiting for data, wasting millions of dollars of compute. The problem is invisible: GPU utilization dashboards report 98%, but that number reflects when the GPU is *scheduled*, not when it is computing. Storage stalls look like compute, until you measure the right thing.

This chapter covers the complete AI storage stack: training data ingestion, distributed filesystems, the PyTorch data pipeline, checkpoint engineering, model serving storage, and production diagnostics. Every section is grounded in production numbers derived from hardware specifications — not marketing claims. You will leave this chapter able to size storage systems, diagnose pipeline bottlenecks, and make principled tradeoffs between cost, latency, and throughput at every scale from 32 GPUs to 100,000 GPUs.

**LEARNING OBJECTIVES**

**Level 1 — Foundational:**

Understand the five-tier AI storage hierarchy, why GPU idle time from storage is invisible, and the difference between data reading and checkpoint writing requirements.

**Level 2 — Intermediate:**

Size and configure distributed storage for any AI training workload. Design a data pipeline that keeps 1,024 GPUs fully fed. Calculate checkpoint sizes and timing overhead. Implement async checkpointing.

**Level 3 — Principal:**

Diagnose and fix storage-related GPU idle time in production. Choose the right filesystem technology and configuration for any scale. Design a complete three-tier checkpoint strategy with retention policies. Quantify the ROI of storage optimizations in dollar terms.

## 15.1 The AI Storage Hierarchy

Modern AI clusters operate across five distinct storage tiers, each optimized for a different combination of capacity, bandwidth, and latency. Understanding which tier handles which data — and why — is the foundation for every design decision in this chapter.

The key insight: *training data reading is not the hard problem*. A 70B model training run on 1,024 GPUs consumes only 0.32 GB/s of training data — trivially satisfied by any modern filesystem. The hard problems are **checkpoint writing** (burst writes of 1+ TB in seconds) and **model weight loading** (sequential reads of 70–400 GB at serving startup). Every storage system decision flows from these two constraints.

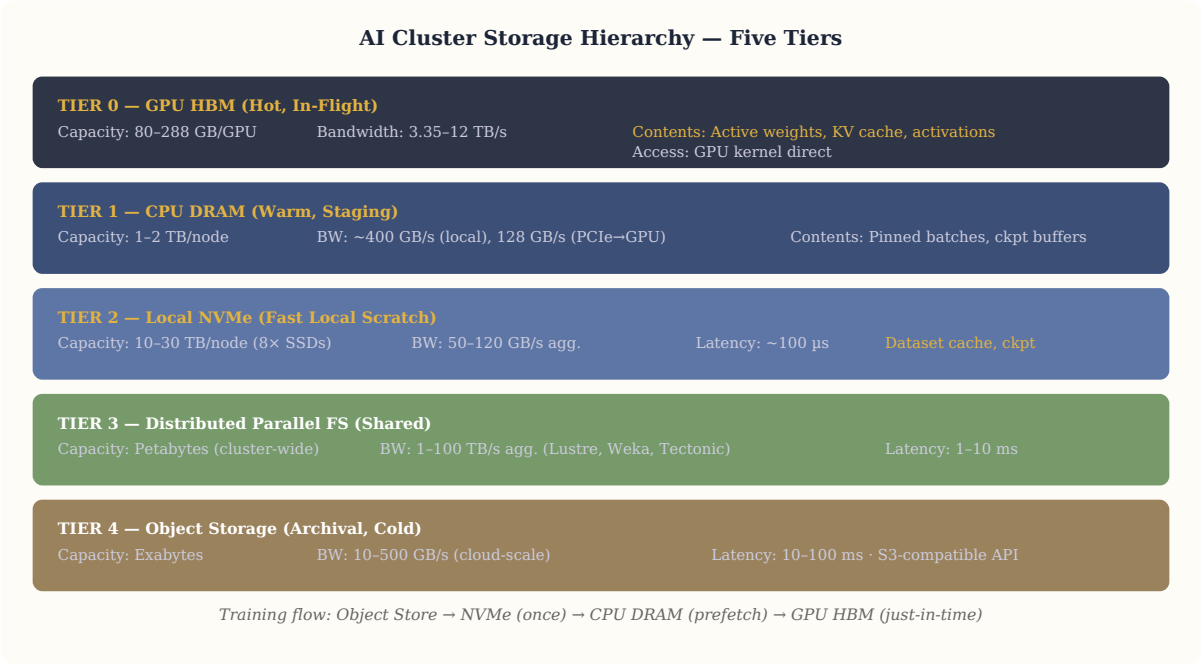



*Figure 15.1 — Five-tier AI storage hierarchy. Bandwidth and capacity increase with tier depth; latency rises from nanoseconds (HBM) to milliseconds (object store).*

```
TRAINING DATA CONSUMPTION RATE — Why Reading Is Not the Hard Problem:

Step config: 70B model, 1,024 GPUs, batch = 4M tokens
  GPU step time:       ~25 seconds
  Data per step:       4M tokens × 2 bytes/token (BF16 IDs) = 8 MB
  Required read BW:    8 MB / 25 sec = 0.32 GB/s  ← TRIVIALLY LOW

For comparison:
  Single NVMe SSD:        14 GB/s sequential read
  NFS single server:       5 GB/s (generous estimate)
  Lustre 8-OST cluster:   ~80 GB/s

  Verdict: ANY filesystem technology exceeds training data read requirements.
  The bottleneck is NEVER training data reads — it is CHECKPOINT WRITES.
```

## 15.2 Distributed Filesystems for AI Training

Choosing the wrong filesystem for an AI cluster is one of the most expensive mistakes an infrastructure architect can make. An NFS mount on a 1,024-GPU cluster will cause 15–20% GPU idle time from I/O stalls — wasting $4,000–$5,000 per hour at market rates. The right choice depends on cluster scale, checkpoint frequency, and operational maturity.

### 15.2.1 Filesystem Technology Comparison

**Filesystem Technology Comparison for AI Training**

| FILESYSTEM | PEAK BANDWIDTH | LATENCY | GPU SCALE | KEY TRADEOFF |
|---|---|---|---|---|
| **NFS** Simple, ubiquitous | 5–20 GB/s | 1–5 ms | <32 GPUs only | Single-server IOPS bottleneck; do not use |
| **Lustre** HPC standard | Multi-TB/s (>1,000 OSTs) | 1–5 ms | Unlimited scale | Complex operations; requires Lustre expertise |
| **Weka** Modern NVMe-based | 10+ GB/s per node | <0.5 ms | 32–10,000+ GPUs | Azure, AWS; higher cost than Lustre HW |
| **GPFS/Spectrum Scale** | High throughput | 1–5 ms | Enterprise scale | Expensive licensing; IBM ecosystem |
| **Tectonic (Meta)** Custom hyperscale | 10+ TB/s aggregate | ~1 ms | 100K+ GPU clusters | Meta-internal only; not available externally |

**Decision Rule: NFS (<32 GPU) → Weka/Lustre (32–256) → Custom/Lustre+100 OSTs (>256)**

*[SHIPPED] all bandwidth figures; Tectonic figures from Meta engineering blog, 2024*

*Figure 15.2 — Filesystem technology comparison. Scale determines the right choice; NFS is never correct above 32 GPUs for production training.*

### 15.2.2 The Critical Lustre Stripe Configuration

A Lustre cluster with 100 Object Storage Targets (OSTs) at 2 GB/s each delivers 200 GB/s aggregate — but *only if files are striped across all OSTs*. The default Lustre stripe count is 1, meaning all data for a single file goes to one OST regardless of how many you have. This is the most common and most expensive production misconfiguration in AI storage.

```
LUSTRE STRIPE CONFIGURATION — Critical Setup Commands:

# Before writing ANY training data:
# Set stripe count to -1 (all OSTs) on the dataset directory
lfs setstripe -c -1 /lustre/training_data/

# Verify stripe configuration:
lfs getstripe /lustre/training_data/my_dataset.jsonl
# Output: stripe_count: 96  stripe_size: 1048576  pool: OST_pool_0

# CANNOT be changed retroactively (data must be rewritten)
# -c -1 = use all available OSTs
# -S 4M = 4 MB stripe size (optimal for large sequential writes)

# For checkpoint directories (large sequential writes):
lfs setstripe -c -1 -S 4m /lustre/checkpoints/

# Bandwidth gain from correct striping:
#   Default (c=1):  2 GB/s (one OST)
#   Full stripe:    96 × 2 GB/s = 192 GB/s
#   Improvement:   96× bandwidth — a single config line worth millions/year

# Check cluster OST count:
lfs df /lustre | grep OST | wc -l
```

> **PRINCIPAL INSIGHT: THE STRIPE COUNT TAX**
>
> Every AI cluster using Lustre with default stripe settings is paying a "stripe tax." At 1,024 GPUs writing checkpoints simultaneously at 8.76 GB/GPU, with c=1 you serialize through one OST: 1,024 × 8.76 GB / 2 GB/s = 4,484 seconds per checkpoint. With c=96: 4,484 / 96 ≈ 47 seconds. The difference is entirely a configuration choice — no hardware changes required. This is the highest-ROI storage optimization in production AI infrastructure.

## 15.3 The Data Loading Pipeline — Keeping GPUs Fed

The PyTorch training loop has a subtle performance trap: the DataLoader runs in the Python main process by default, meaning data loading blocks GPU computation. Fixing this — and the cascade of optimizations that follow — is often worth 10–20% effective throughput recovery with zero hardware changes.

### 15.3.1 DataLoader Configuration for Production

The GPU step time for a 70B model is 15–30 seconds. The DataLoader must deliver each batch within that window or the GPU stalls waiting. The problem is that PyTorch's default settings deliver data in ~2 seconds on a well-configured NVMe system — but on a network filesystem with default settings, that rises to 8–15 seconds, causing per-step stalls.

```
# production_dataloader.py — Optimized DataLoader for large-scale training

from torch.utils.data import DataLoader, Dataset
import torch
import numpy as np
import os

class TokenizedDataset(Dataset):
    """
    Memory-mapped dataset for pre-tokenized training data.
    Uses numpy mmap for zero-copy reads from NVMe.

    Why mmap? The OS page cache handles prefetching automatically.
    Large sequential reads trigger read-ahead, so data arrives in CPU DRAM
    before the DataLoader worker requests it. No explicit prefetch needed.
    """
    def __init__(self, data_path: str, seq_len: int = 4096):
        # mmap mode: reads are demand-paged from filesystem
        # 'r' mode: read-only, allows sharing across workers
        self.data = np.memmap(data_path, dtype=np.uint16, mode='r')
        self.seq_len = seq_len
        self.n_sequences = len(self.data) // (seq_len + 1)

    def __len__(self):
        return self.n_sequences

    def __getitem__(self, idx):
        start = idx * (self.seq_len + 1)
        chunk = self.data[start : start + self.seq_len + 1]
        x = torch.from_numpy(chunk[:-1].astype(np.int64))  # input
        y = torch.from_numpy(chunk[1:].astype(np.int64))   # target
        return x, y


def build_production_dataloader(data_path, seq_len=4096, batch_size=32,
                                rank=0, world_size=1):
    dataset = TokenizedDataset(data_path, seq_len)

    # Distributed sampler: each rank gets non-overlapping data
    sampler = torch.utils.data.distributed.DistributedSampler(
        dataset, num_replicas=world_size, rank=rank, shuffle=True
    )

    return DataLoader(
        dataset,
        batch_size=batch_size,
        sampler=sampler,
        # ── CRITICAL PERFORMANCE PARAMETERS ──────────────────────
        num_workers=8,           # 8 workers per GPU (match NVMe queue depth)
        pin_memory=True,         # Pinned CPU DRAM → faster PCIe DMA to GPU
        pin_memory_device='cuda',# Explicitly pin for current GPU
```

```
        prefetch_factor=4,       # Each worker pre-fetches 4 batches ahead
        persistent_workers=True, # Keep workers alive across epochs
        # ──────────────────────────────────────────────────────────
        drop_last=True,          # Required for consistent batch size in DDP
    )
```

> **THE NUM_WORKERS RULE OF THUMB**
>
> Set `num_workers = min(CPU_cores_per_GPU, 8)`. On a DGX H100 (2× 52-core CPUs, 8 GPUs): 104 / 8 = 13 → cap at 8 workers. More than 8 workers per GPU causes CPU memory pressure from prefetched batches without proportional throughput gain. With `pin_memory=True` and `prefetch_factor=4`, each worker pre-allocates 4 pinned batches in CPU DRAM — at batch_size=32, seq_len=4096: 8 workers × 4 batches × 32 × 4096 × 2 bytes = 8.4 GB of pinned CPU memory per GPU. Budget accordingly.

### 15.3.2 WebDataset for Object Store Streaming

When training data lives in object storage (S3, GCS), copying to local NVMe first is strongly preferred. But for datasets too large to cache (multi-petabyte), WebDataset enables streaming directly from object store with minimal overhead:

```
# WebDataset streaming from S3 for massive datasets
import webdataset as wds

dataset = (
    wds.WebDataset("s3://my-bucket/data/shard-{00000..04999}.tar",
                   shardshuffle=True)          # Shuffle shard order each epoch
    .shuffle(1000)                             # Buffer 1000 samples in memory
    .decode("torch")                           # Auto-decode tensors
    .to_tuple("input.pth", "target.pth")       # Extract named keys
    .batched(batch_size, partial=False)        # Consistent batch sizes
)

# Key: shardshuffle=True is critical — without shard-level shuffle,
# all ranks read shards in the same order → correlated training batches
# WebDataset transfers ~2-5 GB/s per node from S3 — adequate for training
# but much slower than local NVMe (50-120 GB/s)
```

### 15.3.3 NVIDIA DALI for Vision Workloads

For multimodal LLMs with image inputs, the PyTorch DataLoader's CPU-based JPEG decoding is the bottleneck. Decoding 224×224 images on CPU takes ~2 ms each; at batch_size=256, that is 512 ms of CPU decoding per step — longer than the GPU forward pass. NVIDIA DALI moves decoding and augmentation to the GPU, achieving 3–5× speedup:

```
import nvidia.dali as dali
import nvidia.dali.fn as fn
from nvidia.dali.pipeline import pipeline_def

@pipeline_def(batch_size=256, num_threads=4, device_id=0)
def image_pipeline(image_dir):
    jpegs, labels = fn.readers.file(
        file_root=image_dir, random_shuffle=True, name="Reader"
    )
```

```
    # GPU decode: NVMe → GPU decode → augment (no CPU round-trip)
    images = fn.decoders.image(jpegs, device="mixed")
    images = fn.random_resized_crop(images, size=224, device="gpu")
    images = fn.normalize(images, device="gpu",
                          mean=[0.485, 0.456, 0.406],
                          std=[0.229, 0.224, 0.225])
    return images, labels

# DALI advantage: CPU-side decode is 2ms/image → GPU decode is 0.3ms/image
# At batch=256: CPU = 512ms/step → GPU = 77ms/step → 6.6× speedup
# For text-only NLP: DALI adds no value (no image decode)
```

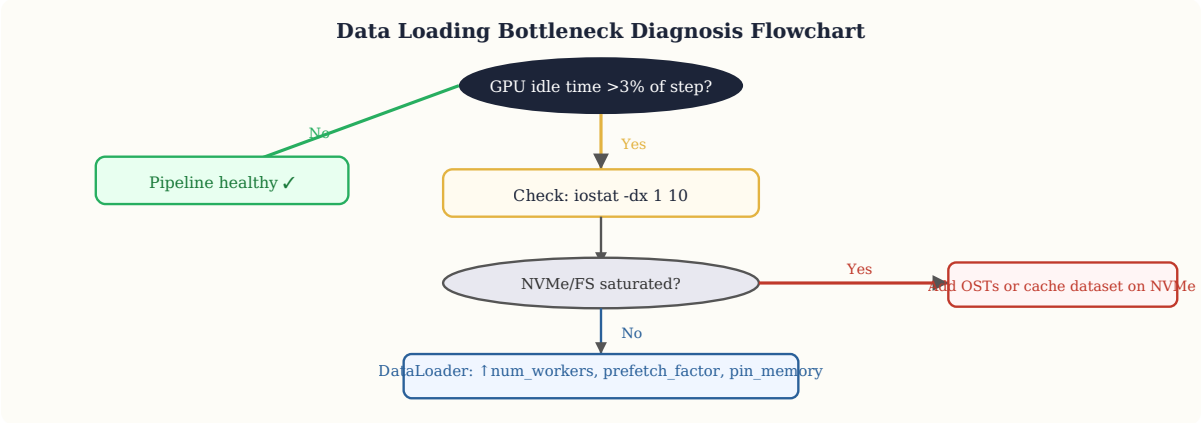



*Figure 15.3 — Data loading bottleneck diagnosis flowchart. The root cause is almost always either filesystem saturation or DataLoader configuration — iostat distinguishes them in under 30 seconds.*

## 15.4 Checkpoint Engineering — The Critical I/O Problem

Checkpointing is not a storage problem. It is a *training efficiency problem*. Every second the GPU spends waiting for a checkpoint to complete is a second of wasted compute. At 10,000 GPUs at $3/GPU-hour, one second of stall costs $8.33. A naïve checkpoint that blocks for 90 seconds every 1,000 steps (25 seconds each) wastes $750 in GPU time per checkpoint — before considering the risk of losing that GPU-time to a hardware failure.

### 15.4.1 Checkpoint Size Analysis

```
CHECKPOINT SIZE BREAKDOWN — 70B Model, AdamW, ZeRO-3, 128 GPUs:

Component              Size/GPU    Total (128 GPUs)   Notes
──────────────────────────────────────────────────────────────────
Model weights (BF16)   1.09 GB       140 GB           2 bytes × 70B params / 128
Optimizer: FP32 wts    2.19 GB       280 GB           4 bytes × 70B / 128
Optimizer: m (FP32)    2.19 GB       280 GB           1st moment
Optimizer: v (FP32)    2.19 GB       280 GB           2nd moment
Gradients (BF16)       1.09 GB       140 GB           for gradient accumulation
Misc (RNG, step, LR)   0.01 GB         1 GB           metadata
──────────────────────────────────────────────────────────────────
TOTAL per GPU:         8.76 GB
TOTAL cluster:                     1,121 GB (~1.1 TB)

CHECKPOINT WRITE TIME by storage speed:
  Naive single-file write (all 128 GPUs → rank 0 → one file):
    128 × 8.76 GB sequential = 1,121 GB through one bottleneck
    At 10 GB/s NFS: 112 seconds  ← CATASTROPHIC (4.5% of 2,500s/step training)

  Per-rank sharding (each rank writes its own file in parallel):
    Each GPU writes 8.76 GB simultaneously
    At 100 GB/s Lustre (96 OSTs): 8.76 / 100 × 128 = 11.2 seconds (distributed!)
    Overhead: 11.2 / 25 = 0.45% per checkpoint — acceptable

  With async checkpoint (GPU→CPU snapshot + background write):
    GPU sees: ~4 seconds (non_blocking CPU copy)
    Background thread does: 11.2 second disk write
    GPU overhead: 4/25 = 0.16% — excellent
```

### 15.4.2 Production Async Checkpointing

```
# async_checkpoint.py — Zero-stall async checkpoint

import torch
import torch.distributed as dist
from concurrent.futures import ThreadPoolExecutor
import threading, os, time

class AsyncCheckpointer:
    """
    Zero-stall async checkpoint: saves in background thread while
    training continues. GPU sees only the CPU copy (fast); disk
    write happens in parallel with the next training steps.
    """
    def __init__(self, checkpoint_dir: str):
        self.checkpoint_dir = checkpoint_dir
        self.executor = ThreadPoolExecutor(max_workers=1)
```

```
        self.pending_future = None

    def save_async(self, model, optimizer, step: int):
        # Wait for any previous checkpoint to finish first
        if self.pending_future is not None:
            self.pending_future.result()

        # GPU → CPU copy (fast: ~4 seconds even for 70B)
        # non_blocking=True: CPU copy runs concurrently with GPU compute
        state_cpu = {}
        for name, param in model.named_parameters():
            state_cpu[name] = param.data.cpu().clone()   # non_blocking not needed here

        optim_state = {k: v.cpu() if torch.is_tensor(v) else v
                       for k, v in optimizer.state_dict().items()}

        # Launch background write — training resumes immediately
        path = os.path.join(self.checkpoint_dir, f"step_{step:07d}")
        self.pending_future = self.executor.submit(
            self._write_to_disk, state_cpu, optim_state, path, step
        )

    def _write_to_disk(self, model_state, optim_state, path, step):
        """Background thread: write to distributed storage (Lustre/Weka)."""
        os.makedirs(path, exist_ok=True)
        rank = dist.get_rank()
        torch.save({'model': model_state, 'optimizer': optim_state, 'step': step},
                   os.path.join(path, f"rank_{rank:04d}.pt"))
        if rank == 0:
            torch.save({'step': step, 'ts': time.time()},
                       os.path.join(path, "metadata.pt"))

# PyTorch Distributed Checkpoint (DCP) — preferred for FSDP/ZeRO models
from torch.distributed.checkpoint import save, FileSystemWriter
from torch.distributed.checkpoint.state_dict import get_state_dict

def save_dcp_checkpoint(model, optimizer, step, path):
    """Sharded DCP save — each rank writes its own portion automatically."""
    model_state, optimizer_state = get_state_dict(model, optimizer)
    save({"model": model_state, "optimizer": optimizer_state, "step": torch.tensor(step)},
         storage_writer=FileSystemWriter(f"{path}/step_{step:07d}"))
```

### 15.4.3 Three-Tier Checkpoint Strategy

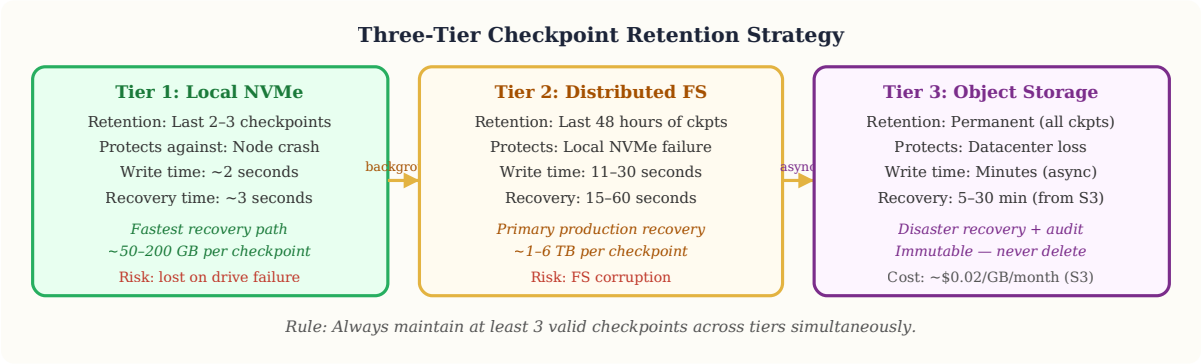



*Figure 15.4 — Three-tier checkpoint retention strategy. Each tier protects against different failure modes; all three are required for a fault-tolerant training cluster.*

## 15.5 Model Serving Storage — Fast Weight Loading

Inference serving has a storage challenge entirely different from training: instead of writing 1.1 TB every few hours, it must *read* 35–400 GB at startup, and must do so quickly enough to support rapid scaling and daily model updates. The difference between a 15-second and a 0.65-second weight load time determines whether auto-scaling events cause user-visible latency spikes.

### 15.5.1 Weight Loading Analysis

```
WEIGHT LOADING COMPARISON — 70B FP8 Model (70 GB):

Storage Path           Effective BW    Load Time    Notes
──────────────────────────────────────────────────────────────────
NFS server             5–10 GB/s       7–14 sec     Single server IOPS limit
Local NVMe (warm)      50–112 GB/s     0.6–1.4 sec  After first-load caching
Local NVMe (first)     15–30 GB/s      2.3–4.7 sec  From NFS → NVMe (network)
GPUDirect Storage      ~112 GB/s       ~0.65 sec    NVMe → GPU direct (no CPU)
GPU P2P (NVLink)       ~600 GB/s       ~0.12 sec    From warm neighbor GPU

WITHOUT GDS (legacy path):
  NVMe → PCIe → CPU DRAM (bounce buffer) → PCIe → GPU HBM
  CPU bottleneck: 2–3 cores handling DMA
  Effective BW: min(CPU DRAM BW, PCIe) = 128 GB/s → 0.55 sec
  But CPU bounce buffer contention: practical ~0.8 sec

WITH GPUDirect Storage (GDS):
  NVMe → PCIe → GPU HBM (direct, no bounce buffer)
  Bottleneck: NVMe aggregate = 8 × 14 = 112 GB/s → 0.63 sec
  CPU usage: near zero (no DMA coordination)

  Enabling GDS:
    gdscheck -p                    # Verify GDS support
    # PyTorch/Megatron auto-detect; vLLM uses when download_dir is GDS NVMe
```

### 15.5.2 Production Weight Caching Strategy

```
from vllm import LLM

llm = LLM(
    model="meta-llama/Llama-3-70b",
    download_dir="/nvme/model_cache",  # Local NVMe cache (fast subsequent loads)
    load_format="auto",                # Selects safetensors (fastest format)
    dtype="float8",                    # FP8: 70 GB vs 140 GB BF16
    max_model_len=32768,
)
# First load: NFS → NVMe (~10 sec) + NVMe → GPU (~0.65 sec) = ~11 sec
# Subsequent (restart, scale-out): NVMe → GPU only = ~0.65 sec
```

### 15.5.3 KV Cache Storage Tiering for Inference

At high concurrency, the HBM KV cache fills and requests must be evicted. Eviction to CPU DRAM costs 16 ms round-trip (PCIe latency at small block sizes); eviction to NVMe costs 200 µs for sequential blocks with NVMe-optimized access. This 80× advantage makes NVMe a viable KV overflow tier — but only under specific concurrency conditions. (See §11.4 for KV cache fundamentals.)

```
KV CACHE TIER SELECTION — LLaMA-3 70B, H100 SXM5, FP8 KV, TP=2:

HBM KV budget (per GPU pair after weights):
  Total HBM: 2 × 80 GB = 160 GB
  FP8 weights (70B): 70 GB
  Activations + overhead: ~15 GB
  KV budget: 160 - 70 - 15 = 75 GB

At 8K context, FP8, LLaMA-3 70B (80 layers, 8 KV heads, dim=128):
  KV per token = 2 × 80 × 8 × 128 × 1 byte = 163,840 bytes ≈ 0.156 MB/token
  KV per user at 8K: 8,192 × 0.156 MB = 1.28 GB

Max concurrent users in HBM: 75 GB / 1.28 GB = 58 users

NVMe overflow tier (8× drives, ~110 GB/s):
  Additional KV capacity: e.g., 10 TB NVMe = 10,000 GB / 1.28 GB/user = 7,812 users
  BUT: NVMe access latency = 200 µs per token fetch
  At 15 ms/token decode and 10% NVMe fetches: penalty = 0.1 × 200µs = 20µs = 0.13% overhead
  → NVMe KV tiering is viable if <20% of tokens require NVMe fetch

Decision rule:
  Decode latency SLA 30ms+: NVMe KV viable
  Decode latency SLA <10ms: NVMe KV too slow (use CPU DRAM instead)
```

## 15.6 Storage Performance Diagnostics Cookbook

Storage problems masquerade as compute problems. GPU utilization shows 95%, but actual MFU is 42%. The gap is DataLoader idle time — time when the GPU is "active" (not in CUDA idle mode) but blocked waiting for the next batch in Python. The following diagnostic workflow resolves storage root causes in under 10 minutes.

### 15.6.1 Four-Step Diagnosis Protocol

```
## STEP 1: Quantify DataLoader wait time per step
## Add this to your training loop:

import time
data_wait_times = []
for step, batch in enumerate(dataloader):
    t0 = time.perf_counter()
    input_ids, labels = batch   # This is where DataLoader blocking happens
    data_wait = time.perf_counter() - t0
    data_wait_times.append(data_wait)

    # Forward/backward pass
    ...

print(f"Mean DataLoader wait: {np.mean(data_wait_times)*1000:.1f} ms")
```

```
print(f"P99 DataLoader wait:  {np.percentile(data_wait_times, 99)*1000:.1f} ms")
# Target: mean <200ms (for 25-second GPU steps), P99 <500ms
# If P99/mean ratio >3×: storage variance problem (Lustre OST contention, NFS locks)

## STEP 2: Check disk I/O saturation on compute nodes
iostat -dx 1 10   # Device extended stats, 1s interval, 10 samples
# Watch: %util column — if any device >80%, it's saturated
# Watch: rkB/s — read throughput per device
# Watch: r_await — average read latency (target <1ms for NVMe)

## STEP 3: Check network filesystem bandwidth (Lustre)
lfs df -i /lustre                        # Inode usage (metadata pressure)
lctl get_param llite.*.read_bytes        # Lustre client read bytes
lctl get_param osc.*.stats | grep cache  # OST cache hit rate

## STEP 4: Verify Lustre stripe configuration
lfs getstripe -R /lustre/training_data/ | grep stripe_count | sort | uniq -c
# Expected: all files should show stripe_count = N (where N = OST count)
# If stripe_count=1 is common: critical performance bug — must rewrite data
```

### 15.6.2 Common Storage Failure Modes

| Symptom | Root Cause | Diagnostic Command | Fix |
|---|---|---|---|
| GPU idle >5%, data steps slow | DataLoader bottleneck | Profile DataLoader wait time (see above) | Increase `num_workers`, `prefetch_factor` |
| Step time variance 3–5× | Lustre OST contention / NFS lock | `lctl get_param osc.*.stats` | Increase stripe count; dedicated storage network |
| Checkpoint write 10× slower than expected | Single-rank save (not sharded) | `iostat -dx` during checkpoint — only 1 device active | Switch to PyTorch DCP (per-rank sharding) |
| GPU stall during checkpoint | Synchronous checkpoint blocking training loop | Nsight timeline shows GPU idle during save | Implement `AsyncCheckpointer` (§15.4.2) |
| Job fails at checkpoint save | Storage full | `df -h /lustre` | Alert at 80% capacity; auto-purge old checkpoints |
| Model load time >10 seconds | Loading from NFS on serving restart | `time python -c "torch.load('model.pt')"` | Cache to local NVMe; enable GDS |
| NVMe not saturating during reads | I/O queue depth too low | `fio --iodepth=1 vs --iodepth=32` comparison | Use `io_uring`; increase `read_ahead_kb` |

### 15.6.3 Sequence Packing — The Most Underutilized Optimization

Web-scale training data is highly variable in length. A typical Common Crawl shard has a mean document length of ~400 tokens with a maximum of 4,096. Without sequence packing, padding fills the gaps — and padding is wasted compute. At a padding ratio of 85%, you are spending 85% of every GEMM operation on zeros.

```
SEQUENCE PACKING COMPUTE RECOVERY:

Input: web-crawl dataset, mean doc length 400 tokens, max_seq=4096, batch=32

Without packing:
  Tokens per sequence: 4,096 (padded to max)
  Real tokens: ~400 average
  Padding fraction: (4096 - 400) / 4096 = 90.2% wasted compute!

With packing (First-Fit Decreasing algorithm):
  Pack multiple docs into each 4,096-token window
  e.g., Seq = [doc_A=2048][doc_B=1024][doc_C=1024] = 4,096 real tokens
  Packing efficiency: typically 95–98% (nearly zero padding)

Attention masking for packed sequences:
  Standard mask: lower-triangular (all tokens attend to prior)
  Packed mask: block-diagonal (doc boundaries prevent cross-attention)
  position_ids = [0..2047, 0..1023, 0..1023]  ← resets mark doc boundaries
  FlashAttention varlen: flash_attn_varlen_qkvpacked_func handles this natively

Throughput impact (70B on 512 H100s):
  Before packing: 42% MFU (baseline)
  After packing:  ~50% MFU (+8 MFU points)
  = 512 × 8 MFU-points × $3/hr = $11,500/day of recovered compute
  One-time cost: pre-process dataset with packing (~2-4 hours engineer time)
```

## 15.7 Key Numbers and Rules of Thumb

```
STORAGE QUICK REFERENCE — Principal-Level Numbers:

BANDWIDTH TARGETS:
  Training data read:        >10 GB/s aggregate (per 1,024 GPUs — trivially met)
  Checkpoint write burst:    >200 GB/s (1,024 GPUs × 8.76 GB / 45-second window)
  Model weight load (NVMe):  ~112 GB/s (8× NVMe per node, GDS)
  Model weight load (NFS):   5–20 GB/s (avoid for inference)

CHECKPOINT SIZES (full checkpoint = model + optimizer + gradients × 16 bytes):
  7B model, AdamW:    7B × 16 bytes = 112 GB
  70B model, AdamW:   70B × 16 bytes = 1.12 TB
  405B model, AdamW:  405B × 16 bytes = 6.48 TB
  ZeRO-3 per rank:    total / num_ranks (128 GPUs: 1.12 TB / 128 = 8.75 GB/rank)

CHECKPOINT TIMING RULES:
  Async CPU copy (GPU→CPU):  <5 seconds (even for 70B, via non_blocking)
  Async disk write (to FS):  checkpoint_size / storage_BW (parallel across ranks)
  Checkpoint interval:       every 500–2,000 steps (~2–8 hours of training)
  Recovery time target:       <10 minutes (write time + job restart)

FILESYSTEM SELECTION BY SCALE:
  <32 GPUs, dev/research:   NFS (simple, fine for this scale)
  32–256 GPUs, production:  Weka (low latency) or Lustre (high expertise)
  >256 GPUs, hyperscale:   Custom (Tectonic, GPFS) or Lustre 100+ OSTs
  All scales: Lustre stripe = -1 BEFORE writing any data

GPU IDLE TARGETS (storage-caused):
  Data loading contribution:  <3% of step time
  Checkpoint contribution:    <1% of step time (with async)
  Combined storage overhead:  <5% of total training time
```

## 15.8 Worked Example: Sizing Storage for a 1,024-GPU Cluster

WORKED EXAMPLE — PRINCIPAL LEVEL

### Problem: Size the storage system for a 1,024-GPU H100 cluster training LLaMA-3 70B models

**Given:** 1,024 H100 SXM5 GPUs, ZeRO-3, batch=4M tokens, seq_len=4,096, step_time≈25 sec, checkpoint every 1,000 steps, async window of 30 seconds, target checkpoint overhead <1%.

**Step 1: Training data read bandwidth required**

```
Data per step: 4M tokens × 2 bytes = 8 MB
Step time:     25 seconds
Required BW:   8 MB / 25 sec = 0.32 GB/s  ← minimal

Conclusion: ANY filesystem meets this. Not the bottleneck.
```

**Step 2: Checkpoint write bandwidth required**

```
Checkpoint size: 70B × 16 bytes = 1.12 TB (full 70B AdamW checkpoint)
With ZeRO-3 (1,024 ranks): 1.12 TB / 1,024 = 1.09 GB per rank
Async window: 30 seconds (GPU resumes after CPU copy, disk write in background)

Required aggregate BW: 1,024 × 1.09 GB / 30 sec = 37.3 GB/s

With safety margin (2×): target 75 GB/s aggregate write bandwidth
```

**Step 3: Lustre configuration specification**

```
Target: 75 GB/s aggregate write
Per-OST bandwidth (commodity NVMe OSTs): ~2 GB/s sustained write

OSTs needed: 75 GB/s / 2 GB/s = 38 OSTs minimum → round to 48 OSTs

Lustre setup:
  MDTs: 2 (for metadata redundancy)
  OSTs: 48 (for checkpoint write bandwidth)
  Stripe: lfs setstripe -c -1 -S 4m /lustre/checkpoints/
  Network: 200 GbE or InfiniBand HDR per OSS node
  Estimated annual cost: 48 OSTs × $25K/node = $1.2M storage hardware
```

**Step 4: Model weight load time for inference**

```
70B FP8 model: 70 GB
With NVMe cache (GDS enabled): 70 GB / 112 GB/s = 0.63 seconds
Without NVMe cache (from NFS):  70 GB / 7.5 GB/s = 9.3 seconds

Decision: local NVMe model cache is required for serving SLA compliance
Auto-scaling from 0→serving in <2 seconds requires NVMe + GDS
```

**Verdict:** Specify 48 OST Lustre for checkpoint writes, local NVMe per node for dataset caching and model serving, S3-compatible object store for long-term checkpoint archival. Total cost dominated by Lustre OST hardware ($1.2M) — justified by preventing >1% checkpoint overhead ($100K+/month in GPU waste at $3/GPU-hr).

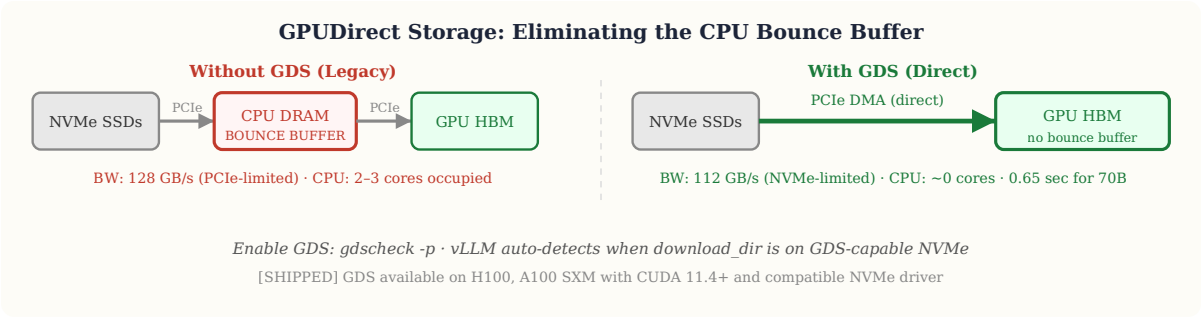



*Figure 15.5 — GPUDirect Storage eliminates the CPU DRAM bounce buffer, reducing 70B model load time from ~0.8 seconds to ~0.65 seconds while freeing 2–3 CPU cores from DMA coordination.*

## Key Takeaways

CORE CONCEPTS

- **Training data reading is never the bottleneck** — a 70B model on 1,024 GPUs needs only 0.32 GB/s of data bandwidth, met by any filesystem. Checkpoint writing (1.1 TB burst in <30 seconds) and model weight loading (70 GB at serving startup) are the actual hard problems.
- **Async checkpointing with per-rank sharding reduces overhead from 5–16% to <0.5%.** The GPU sees only the CPU snapshot time (~4 seconds); the disk write continues in a background thread while training resumes. Single-rank saving without sharding serializes all checkpoint data through one bottleneck.
- **Lustre stripe count is the single highest-ROI storage tuning parameter.** Default stripe_count=1 routes all data through one OST regardless of cluster size. Setting `lfs setstripe -c -1` multiplies available bandwidth by the OST count — a 96× improvement from one configuration line.
- **Sequence packing eliminates padding waste** and typically delivers 15–25% throughput improvement on variable-length corpora. On web-scale data with mean length 400 tokens and max 4,096, the padding fraction without packing exceeds 90%.

COST & PRODUCTIVITY IMPACT

- **Fixing DataLoader configuration (num_workers, prefetch_factor, pin_memory)** can recover 10–20% MFU with zero hardware changes. At 1,024 H100s at $3/GPU-hr, 15% MFU recovery = $13,800/day of effectively free compute.
- **Sequence packing on a 512-GPU cluster** recovering 8 MFU points is worth $11,500/day. One-time dataset preprocessing cost: 2–4 engineer-hours. Payback: under 1 hour of training.
- **NVMe local model cache for inference** reduces weight load time from 9.3 seconds (NFS) to 0.65 seconds (GDS). On a serving cluster with 100 restarts/day across 50 nodes, this saves 100 × 50 × 8.65 seconds = 12 GPU-hours/day = $36/day per server fleet.

PRINCIPAL-LEVEL INSIGHT

- **Checkpoint strategy requires three tiers for correctness**, not just one backup. Local NVMe protects against node crash (fastest recovery), distributed FS protects against local failure (primary recovery path), object storage is permanent and immutable (disaster recovery + reproducibility). A corrupted checkpoint with no fallback means restarting training from zero — the worst possible outcome on a multi-month training run.
- **GPU utilization dashboards are misleading for storage diagnosis.** DataLoader stalls appear as "active" Python time, not GPU idle time. The correct signal is step-time variance: a P99/P50 ratio >3× on step time almost always indicates filesystem contention (Lustre OST hotspot, NFS lock contention), not compute or network issues.
- **NVMe local storage serves three distinct and non-competing functions**: dataset shard caching (avoid repeated Lustre reads), checkpoint staging (50–120 GB/s write, faster than network), and KV cache overflow for inference (200 µs vs 16 ms for CPU DRAM swap). Provision all three use cases when specifying per-node NVMe capacity — typically 10–30 TB per node in AI clusters.

## Review Questions

FOUNDATIONAL Build the mental model — no math required

1. A colleague argues that their AI cluster's storage bottleneck is training data reading and wants to buy a faster filesystem to increase read throughput. Using the five-tier storage hierarchy from §15.1, explain why this is almost certainly the wrong diagnosis. What are the two storage operations that actually determine storage requirements at scale, and why does training data reading rarely limit throughput?
2. Explain the difference between synchronous and asynchronous checkpointing. If a training run on 1,024 GPUs uses synchronous checkpointing and checkpoints every 1,000 steps (each step takes 25 seconds), what happens to GPU utilization during the checkpoint write? Why is the "GPU utilization" metric misleading in this scenario?

3. What is sequence packing, and why does it matter for training data with variable document lengths? Describe in plain terms what happens to compute efficiency without packing on a web-scale corpus where most documents are 200–500 tokens but the maximum context is 4,096 tokens.

**INTERMEDIATE** Apply and calculate — production context

1. A 70B model training run on 1,024 GPUs (ZeRO-3, DP=128) experiences 12% GPU idle time attributed to data loading. The dataset is pre-tokenized and stored on a Lustre filesystem with default stripe settings. (a) What is the most likely root cause given the "default stripe settings" detail? (b) Write the two commands needed to diagnose and then fix this. (c) What is the expected improvement in GPU effective throughput after the fix?

2. Calculate the checkpoint write bandwidth required for a 405B model training run on 2,048 GPUs (ZeRO-3) that checkpoints every 500 steps with a 45-second async window. The model uses AdamW. (a) What is the total checkpoint size? (b) What is the per-rank write size? (c) What aggregate write bandwidth is required? (d) What Lustre configuration (OST count) would satisfy this at 2 GB/s per OST?

3. A serving cluster restarts inference servers nightly for model updates. Each server loads a 70B FP8 model (70 GB). Currently loading from NFS takes 9.3 seconds. What two architectural changes would reduce this to under 1 second? For each: state the mechanism and quantify the expected improvement using the bandwidth numbers from §15.5.

**PRINCIPAL LEVEL** Design, decide, and defend — architecture reviews

1. **Storage Architecture Design for a 2,048-GPU Cluster.** You are specifying storage for a new H100 cluster (2,048 GPUs) that will train 70–405B models. Requirements: checkpoint write latency <60 seconds (sync equivalent), dataset read throughput sufficient to keep all GPUs fed at batch=4M tokens and step_time=20 seconds, model weight load time <2 seconds at serving startup, and support for daily checkpoint archival. (a) Specify the complete storage architecture: filesystem type, OST count, per-node NVMe config, object store. (b) Justify each decision with bandwidth math. (c) Estimate the 3-year TCO of your storage design assuming $25K per OST node and $0.02/GB/month for object storage at 6 TB checkpoint/day. (d) At what cluster size would you switch from Lustre to a custom distributed filesystem, and what would trigger that decision?

2. **Diagnosing a Production Storage Regression.** Your 512-GPU training cluster's checkpoint write time suddenly increased from 12 seconds to 95 seconds over a weekend, blocking training for 83 seconds per checkpoint (3.3% overhead on 25-second steps). GPU utilization monitors show no change. (a) List the three most likely root causes in order of probability. (b) Write the exact sequence of diagnostic commands to isolate the root cause within 10 minutes. (c) For each root cause you listed, write the specific fix and estimated recovery time. (d) What monitoring alert would have caught this regression before the training team noticed it?

3. **KV Cache Tiering System Design.** You are designing a KV cache tiering system for a production LLaMA-3 70B serving cluster (H100 SXM5, FP8 KV, TP=2). The cluster serves 1,000 concurrent users at average 8K context. HBM KV budget is 75 GB per GPU pair. You want to extend capacity using local NVMe (8× drives, ~110 GB/s). (a) Calculate the maximum HBM-resident concurrent users. (b) Calculate how many additional users the NVMe tier enables. (c) At what serving throughput (tokens/second) does the NVMe access latency (200 µs) exceed 5% of decode latency (assume 15 ms/token at batch=32)? Is NVMe KV tiering viable for a <50ms P99 TTOT SLA? (d) Sketch the architecture of the KV tier router: what is the eviction policy, how does the router decide which KV blocks to evict to NVMe vs. CPU DRAM, and how does it handle the case where a request is resumed and its KV is on NVMe?



CROSS-REFERENCES: §10.5 (ZERO CHECKPOINTING) · §11.4 (KV CACHE PAGING) · §14.1 (NETWORK BANDWIDTH) · §16.5 (SLURM CHECKPOINT INTEGRATION)