XINNOR

The International Conference on Massive Storage Systems and Technology

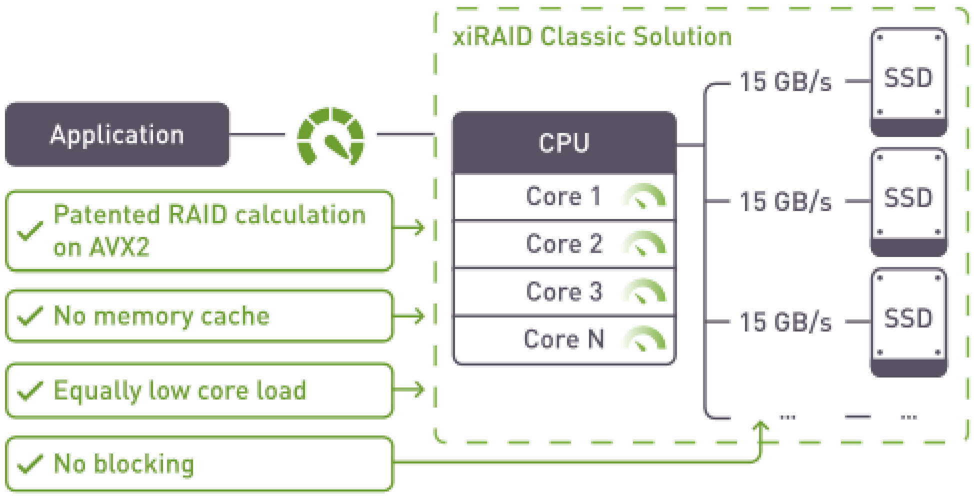
Scaling Up NFS Storage

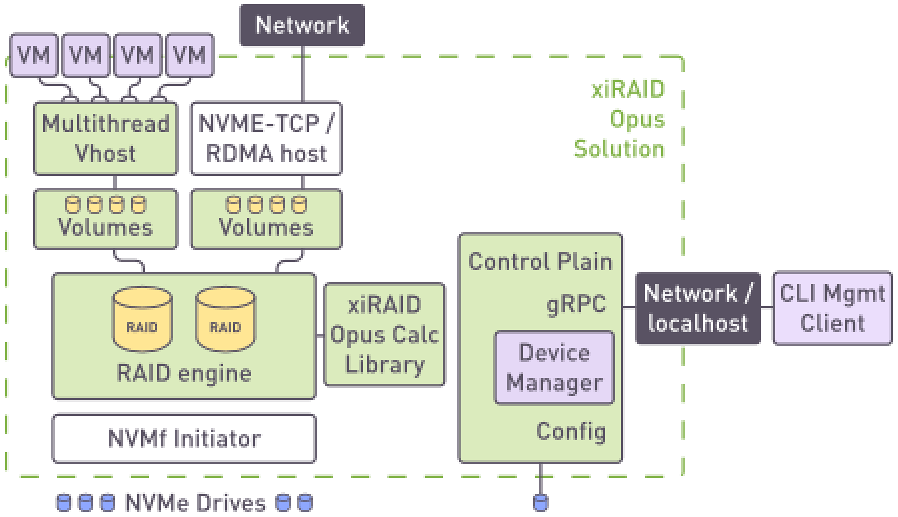
Sergei Platonov
VP of Strategy, Xinnor

Davide Villa
CRO, Xinnor

What is xiRAID

xiRAID aggregates local and network-attached NVMe drives at the maximum possible performance, to create a pool of drives protected in case of multiple drives failure.





xiRAID Classic – for current HW technology

In production

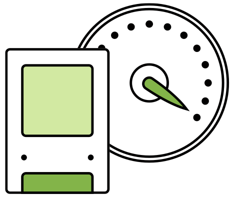
- Linux kernel block device for local or parallel file systems or block storage appliances
- Supporting high availability (drive failures as well as server failures)
- It works on any x86 server

xiRAID Opus – overcoming the kernel limits

Released Sep 2025

- Linux user space block device for NVMe-oF and Virtual environments
- Built-in VirtIO-BLK and NVMeoF target and initiator
- It works on any x86 server as well as DPU/ARM

xiRAID’s advantages



Superior performance in
normal operation

Protects NVMe drives while delivering 97% of their theoretical performance

Demonstrated by the 3rd fastest production deployment worldwide in the IO500 list

Helma Storage Cluster at NHR@FAU

MEGWARE | FAU FRIEDRICH-ALEXANDER UNIVERSITÄT ERLANGEN-NÜRNBERG

5PB HA storage cluster to serve 768 GPUs

IOR & FIND	
EASY WRITE	811.33 GiB/s
EASY READ	1,798.77 GiB/s
HARD WRITE	60.52 GiB/s
HARD READ	419.04 GiB/s
FIND	3,017.00 kIOP/s

METADATA	
EASY WRITE	1,819.16 kIOP/s
EASY STAT	8,221.83 kIOP/s
EASY DELETE	1,420.24 kIOP/s
HARD WRITE	387.63 kIOP/s
HARD READ	2,236.33 kIOP/s
HARD STAT	3,358.07 kIOP/s
HARD DELETE	235.84 kIOP/s

https://io500.org/submissions/view/736

IO500

Production ISC25 List

# ↑	INFORMATION							
	BOF	INSTITUTION	SYSTEM	STORAGE VENDOR				SCORE ↑
1	SC23	Argonne National Laboratory	Aurora	Intel				32,165.90
2	SC23	LRZ	SuperMUC-NG-Phase2-EC	Lenovo				2,508.85
3	ISC25	Erlangen National High Performance Computing Center	Helma	MEGWARE	Lustre	186	18,600	838.99
4	ISC25	Samsung Electronics	SSC-24	WekaIO	WekaIO	291	16,005	826.86
5	SC23	King Abdullah University of Science and Technology	Shaheen III	HPE	Lustre	2,080	16,640	797.04
6	SC24	MSKCC	IRIS	WekaIO	WekaIO	261	27,144	665.49
7	ISC23	EuroHPC-CINECA	Leonardo	DDN	EXAScaler	2,000	16,000	648.96
8	SC24	SoftBank Corp	CHIE-3	DDN	EXAScaler	240	26,880	500.20
9	ISC25	Joint Center for Advanced High Performance Computing	Miyabi-G	DDN	Lustre	200	1,600	391.60
10	SC24	Danish Centre for AI innovation AS	GEFION	DDN	EXAScaler	128	12,288	368.56

The software stack includes both open-source and proprietary components:

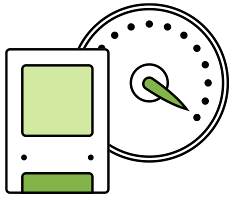
- The operating system (AlmaLinux 9.4) is available as open-source
- The file system (Lustre 2.16.1) is also available as open-source
- Xinnor xiRAID Classic (4.2.0) is proprietary software RAID solution requiring a purchased license

The most efficient IO500 storage cluster significantly improves energy efficiency

- Helma (Lustre + xiRAID) scored 838.99 using 20 storage servers. Competing high scorers need many more storage servers for lower results.
- Fewer storage servers → fewer PSUs, NICs, fans, and less cooling for a given IO500-class result.

#	System	Solution (Vendor)	Score	Storage servers	Score / storage server
3	Helma	xiRAID + Lustre (Xinnor)	838.99	20	41.95
4	SSC-24	WekaFS (WekaIO)	826.86	40	20.67
5	Shaheen III	Lustre (HPE)	797.04	160	4.98
7	Leonardo	ExaScaler (DDN)	648.96	29	22.37
9	Miyabi-G	Lustre (DDN)	391.60	44	8.9

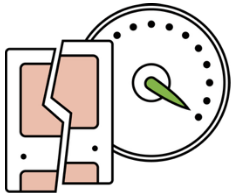
xiRAID's advantages



Superior performance in normal operation

Protects NVMe drives while delivering 97% of their theoretical performance

Demonstrated by the 3rd fastest production deployment worldwide in the IO500 list



High performance in degraded mode

>10-30x performance boost vs competitive options

Joint solution brief with Solidigm demonstrating 25x performance improvement in QLC drive rebuild time

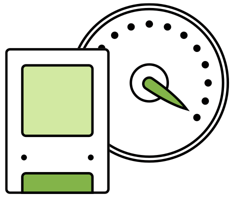
QLC – Rebuild With Workload

Rebuilding 1x Solidigm D5-P5336 61.44TB QLC in RAID 5 over 9 drives

RAID Engine	**Rebuild time**	**Rebuild speed**	**WAF** (lower is better)	**Workload speed under rebuild**
mdraid	>67 days	10.5 MB/s	1.58	Read: ~100MB/s Write: ~45MB/s
xiRAID Classic 4.3	53h 53m 25x faster rebuild	316 MB/s 30x higher throughput	1.21 23% lower WAF	Read: 44GB/s Write: 13GB/s 290-440x higher

https://www.solidigm.com/products/technology/raid-rebuild-with-xiraid-and-qlc-ssds.html

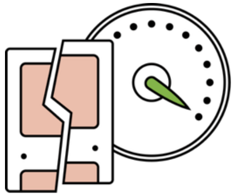
xiRAID’s advantages



Superior performance in normal operation

Protects NVMe drives while delivering 97% of their theoretical performance

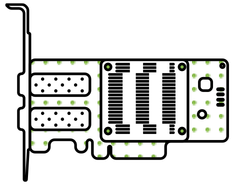
Demonstrated by the 3rd fastest production deployment worldwide in the IO500 list



High performance in degraded mode

>10-30x performance boost vs competitive options

Joint solution brief with Solidigm demonstrating 25x performance improvement in QLC drive rebuild time



No PCIe taxation

Software-only solution with minimal CPU load for checksum calculation.

No need for dedicated hardware, freeing up 16PCIe Lanes and one PCIe slot for additional drives or network connectivity

Why do we need high performance NFS server?

We need to keep GPU busy!

The most expensive part of modern Datacenter is GPU time

Different workloads require different storage performance characteristics

- Training
- Checkpointing
- RAG

Why NFS fits AI

- **Ubiquity & simplicity**
  - Ships with every Linux distribution; one mount command and you're done
- **POSIX semantics**
- **Great fit for some AI I/O patterns**
- **Performance features, when needed**
  - NFSv4.1/4.2 sessions & delegations; server-side copy (v4.2); TCP multistreaming; NFSoRDMA, NFS LOCAL_IO
- **Operational efficiency**
  - Mature observability (nfsstat, mountstats, /proc/fs/nfsd),
  - straightforward tuning (nfsd threads).
- **Security options**
  - From fast sec=sys to Kerberos (krb5/krb5i/krb5p) when compliance requires it.

xiRAID + NFS

Where NFS fits for Xinnor:

- For **small installations**: a tuned NFS server on top of fast local RAID/NVMe delivers the required throughput and simplicity.
- For **large installations**: modular NFS storage can act as a **component** (e.g., pNFS data servers) inside a broader architectures.
- For NFS-on-Demand solution for GPU cloud installation

Our approach

1. Presenting a high-performance RAID (local or composable NVMe-oF)
2. Format correctly (XFS/EXT4, aligned)
3. Export via **NFSv4.2** with either **TCP + nconnect** or **RDMA** to hit both streaming bandwidth and low tail latency.

Xinnor NFS Solution Architecture

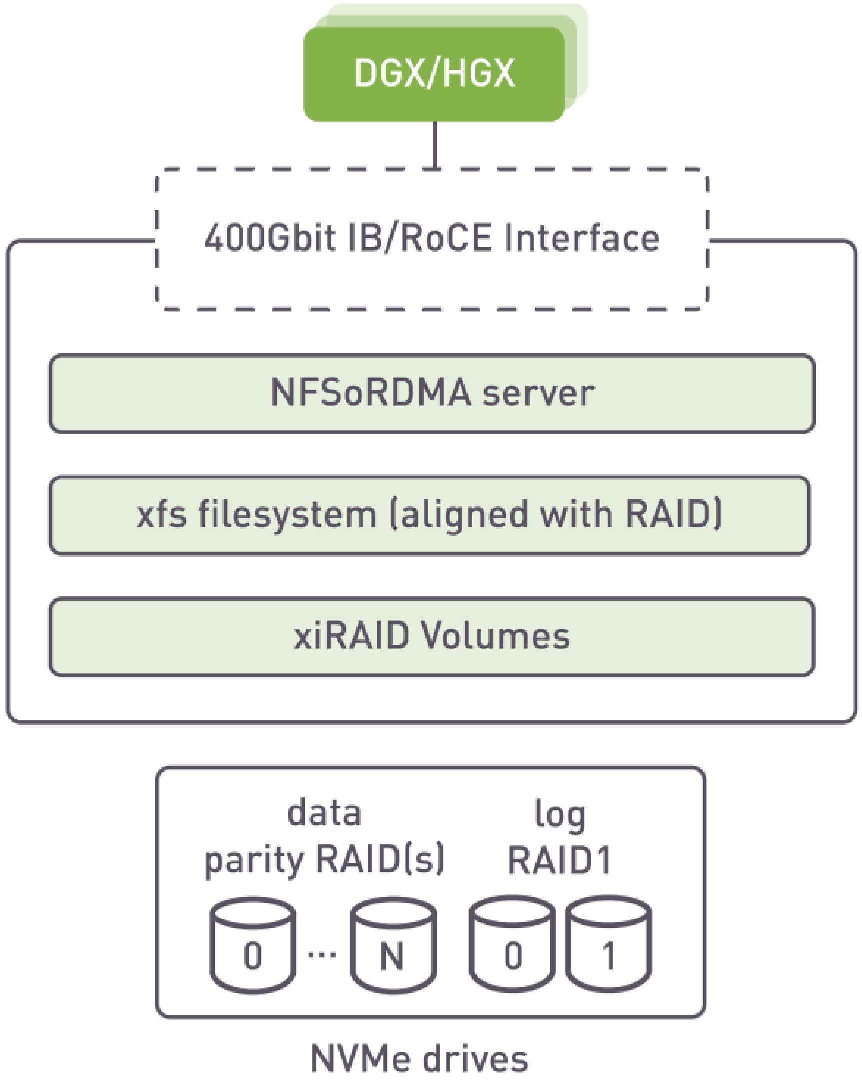
Competitive advantages

- Extremely fast NFSv4 node for checkpointing
- 4x times faster than tier1 NFS vendor per node
- Plug-n-Play capability for easy installations

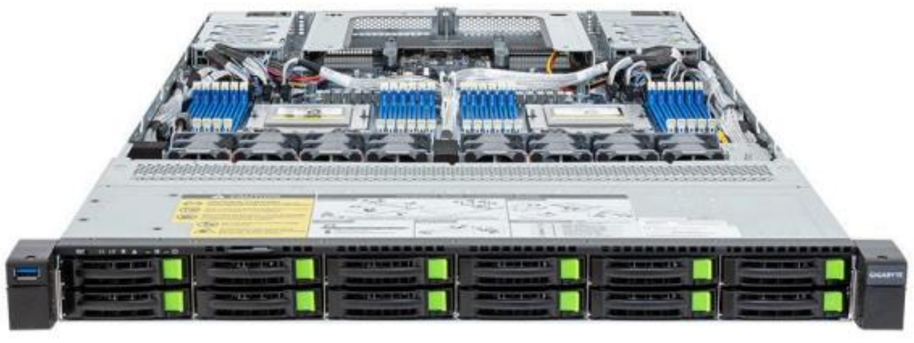
Reference architecture
https://xinnor.io/blog/saturating-infiniband-bandwidth-with-xiraid-to-keep-nvidia-dgx-busy/

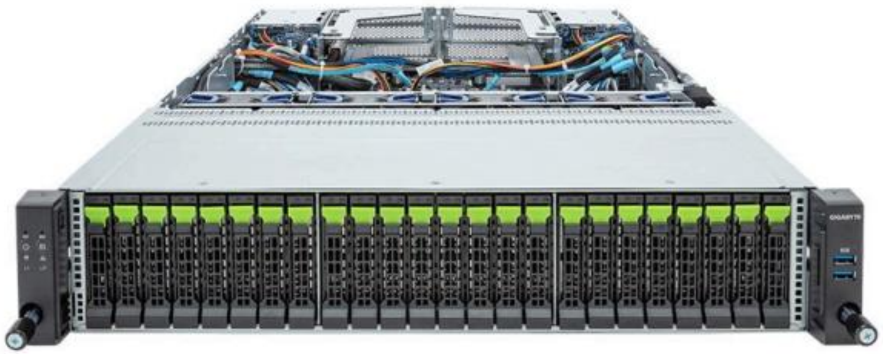
Performance results
https://xinnor.io/blog/saturating-infiniband-bandwidth-with-xiraid-to-keep-nvidia-dgx-busy/

Step by step deployment guide
https://xinnor.io/blog/how-to-build-high-performance-nfs-storage-with-xiraid-backend-and-rdma-access/



Configuration examples





1U12 Server

- Single CPU with 32 cores
- 128+ GB RAM
- 1 x 400Gbs CX7 cards
- 12 x 2.5" PCIe Gen5 drives
- RAID6 with 10 drives for data
- RAID1 with 2 small drives for FS journal

Expected Performance (2 clients):

Sequential read: **~ 45 GB/s**
Sequential write: **~ 40 GB/s**

2U24 Server

- Single CPU with 48 cores
- 128+ GB RAM
- 2 x 400Gbs CX7 cards
- 20x 2.5" PCIe Gen5 drives
- RAID50 with 18 drives for data
- RAID1 with 2 small drives for FS journal

Expected Performance (2 clients):

Sequential read: **~ 90 GB/s**
Sequential write: **~60 GB/s**

What is xiRAID for scale-up NFS servers

Near line-rate writes (streaming):
Sustains **~90–95% of backend media bandwidth** on sequential write/checkpoint paths—turning expensive links (100–400 Gb/s) into useful throughput instead of headroom.

Resource isolation = no contention with NFSD:
Run RAID workers in dedicated **NUMA-aware cpusets.** Result: RAID rebuild/compute and NFSD request handling **don't starve each other**.

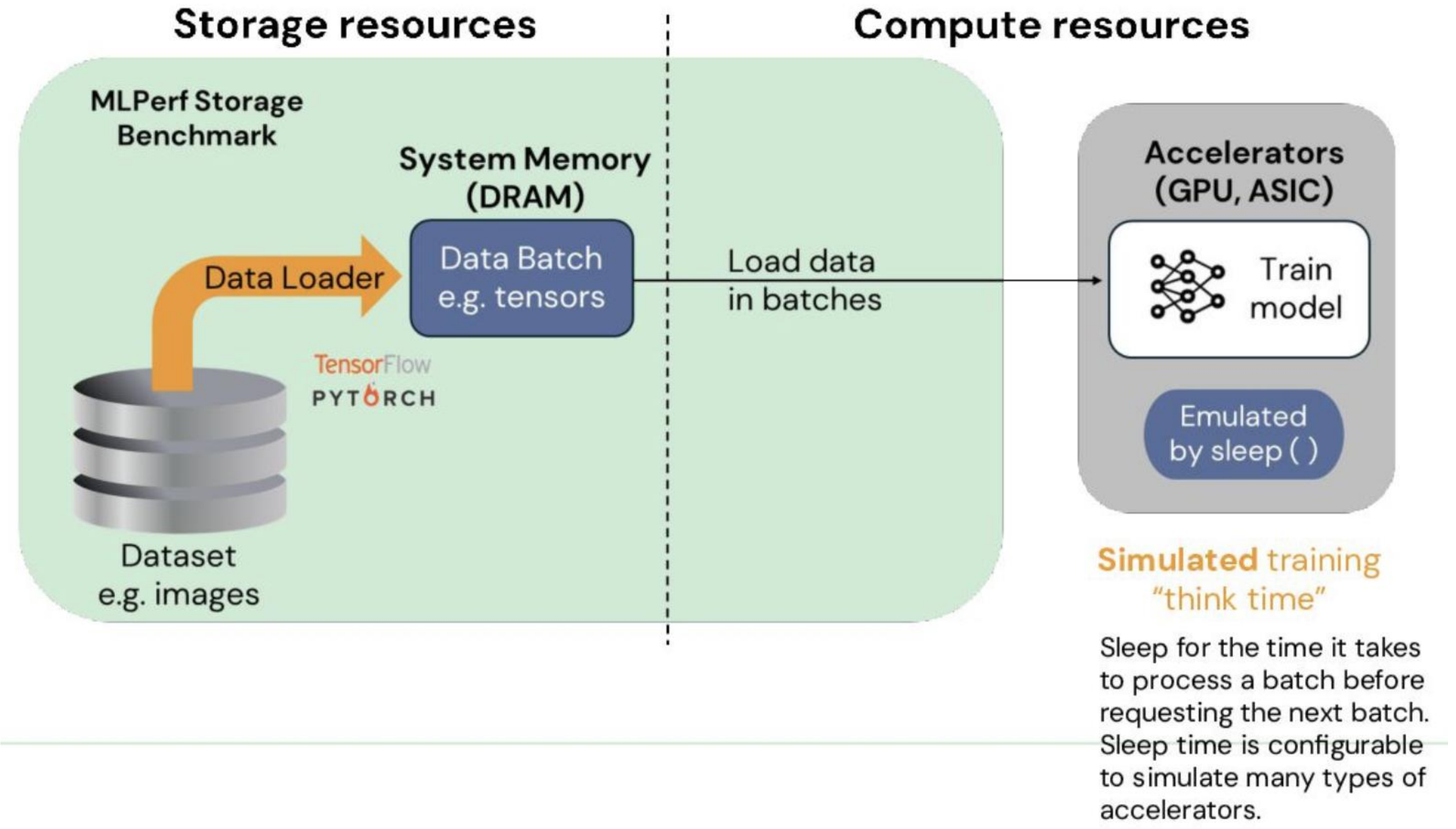
High performance even in degraded mode:
On NVMe failure, xiRAID maintains **~90–95% of available performance** — so data stream keep **feeding GPUs at speed**.

Fast rebuilds → QLC-friendly:
Aggressive, parallel rebuild logic shrinks the vulnerable window and **keeps tail latency flat**, enabling adoption of **large QLC drives**.

What it means for AI:

- Stable **checkpoint throughput** and **smooth P95** during training.
- Predictable performance under load spikes and failures.
- Capacity scaling with QLC, **without** giving up GPU utilization.

MLPerf Storage Benchmark

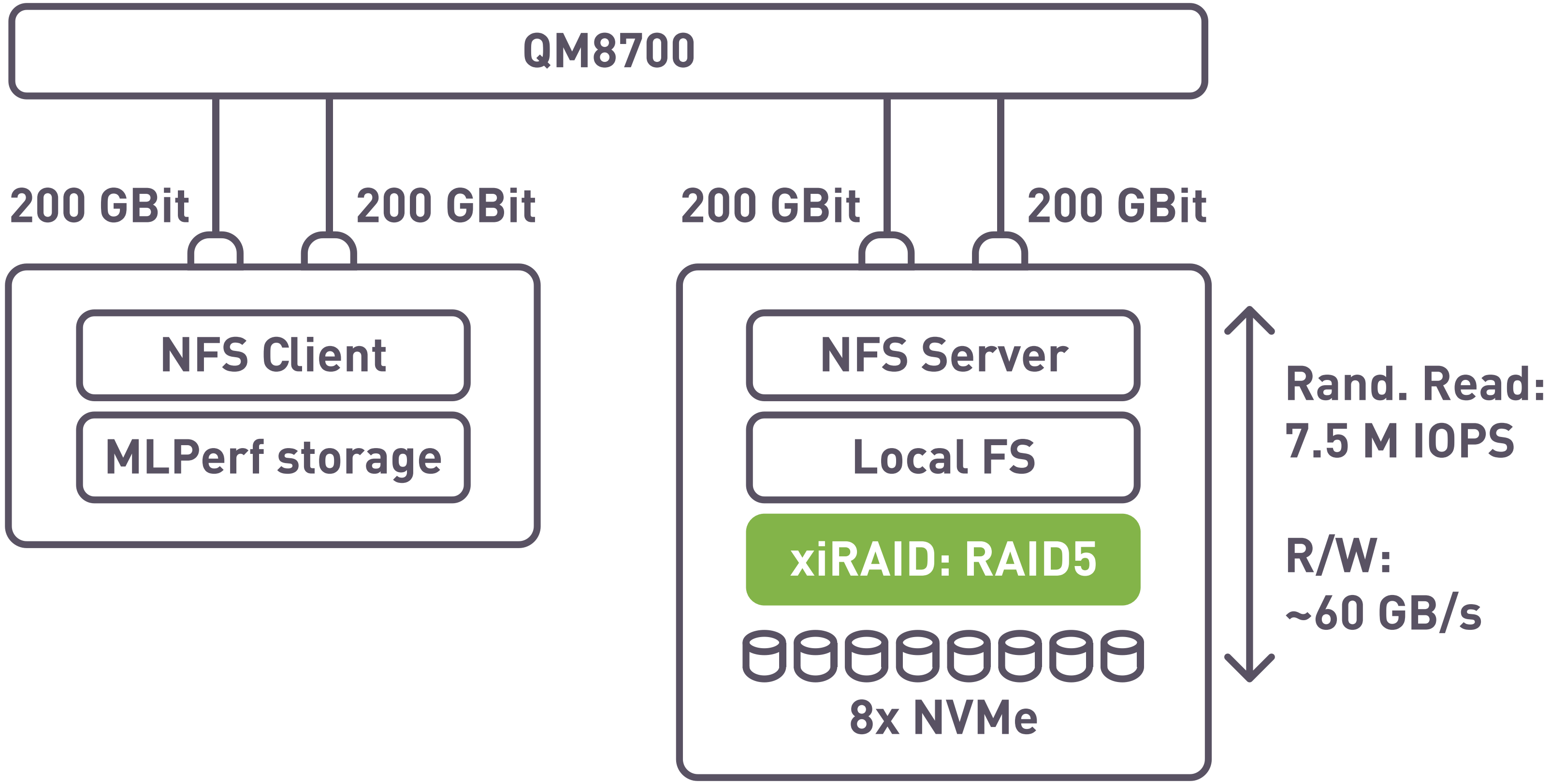


Source: ML Commons, IT Press Tour 60

Workloads simulated by MLPerf Storage

Workload		Reference Network	Sample size	Framework	Reference Quality
Image segmentation (medical)	Synthetic – from KiTS19	3D-Unet	146 MB	PyTorch	maximize MB/s, and # of accelerators with >90% accelerator utilization
Checkpointing		LLAMA3-{8b,70b,405b, 1t}	502M-8.9G file size	PyTorch	Maximize MB/s for Checkpoint Save and Load operations Minimize checkpoint Save and Load Time
Image classification	Synthetic – from ImageNet	ResNet50	150 KB	Tensorflow	maximize MB/s, and # of accelerators with >90% accelerator utilization
Scientific (cosmology)	Synthetic – from CosmoFlow N-body simulation	Parameter prediction	2 MB	Tensorflow	maximize MB/s, and # of accelerators with >70% accelerator utilization

Test bed description



The node configuration:
48 CPU cores, 512 GB RAM, 8xPCIe 4.0 NVMe drives
Ubuntu 24.04 with a customized 6.16 kernel.

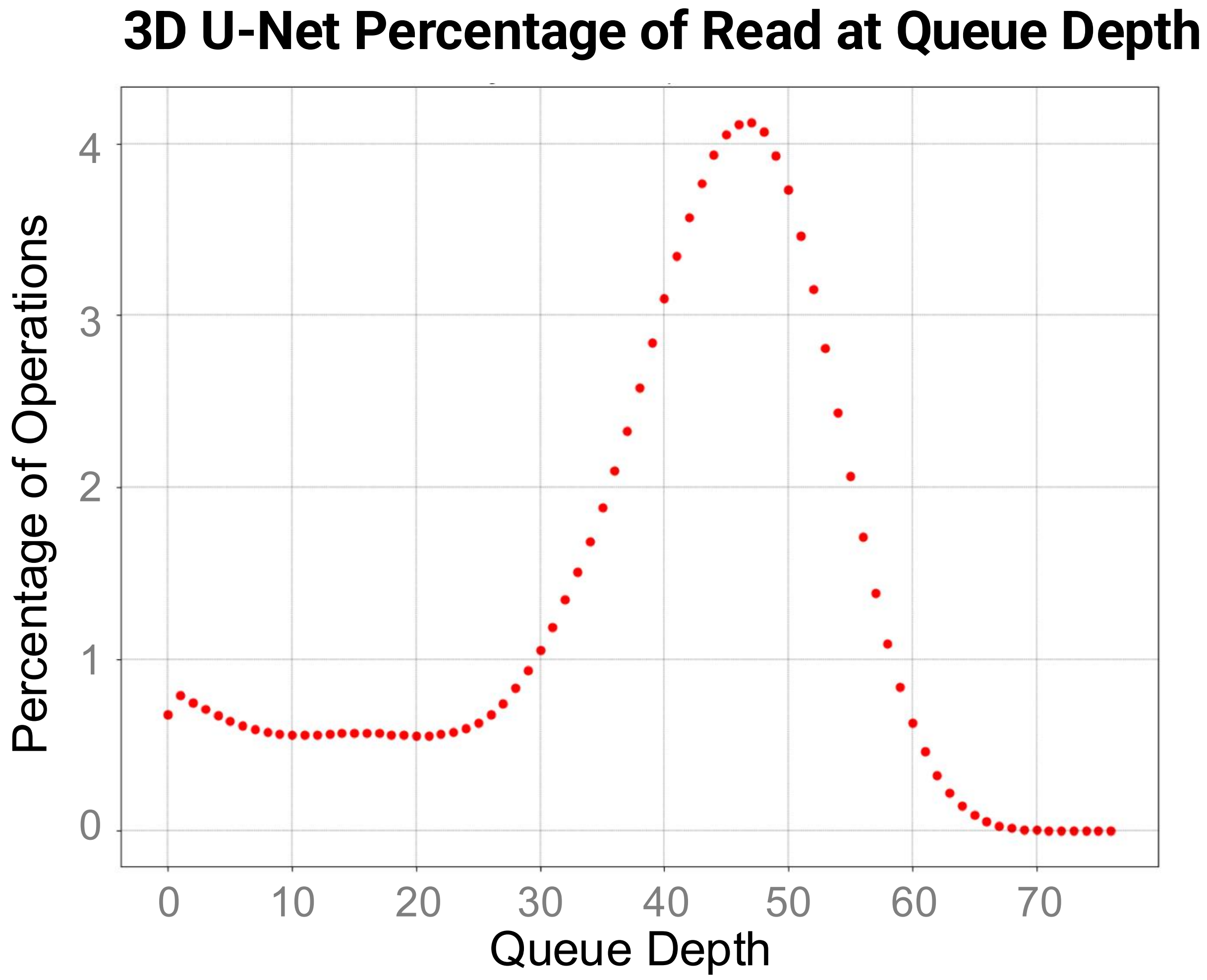
3D U-Net / Checkpointing storage patterns

3D U-Net Training I/O pattern:

- 128 KB reads
- The io queue depth distribution is demonstrated at the right part of the slide

Checkpointing I/O pattern:

- 128 KB writes and reads
- Utilized PyTorch save/load;
- We ran the checkpointing workload with the psync=true parameter set



Source: https://files.futurememorystorage.com/proceedings/2024/20240808_AIML-303-1_Vaske.pdf

Test approach

Workflow
Calculate minimum dataset size → Generate the dataset → Run the benchmark → Generate report

What is "success": throughput (samples/sec) while keeping **Average Accelerator Utilization (AU) ≥ 90%** (the benchmark's "passing" utilization threshold; results pages describe throughput at ≥ 90% AU).

We will focus on 3D U-Net model training and LLMA-405b checkpointing as the most storage-intensive workloads.

During testing, we used a set of tools to monitor parameters and reconfigure the system.

Benchmark parameters:

- mlpstorage training run --hosts 127.0.0.1 --num-client-hosts 1 --client-host-memory 512 --num-accelerators {variable} --accelerator-type h100 --model 3D U-Net --data-dir 3D U-Net_data --results-dir 3D U-Net_results --param dataset.num_files_*tra*in=65000 ***reader.odirect=true*** reader.read_threads=8 reader.prefetch_size=4 --allow-run-as-root
- mlpstorage checkpointing run -rd ch_r3 -m llama3-405b --client-host-memory-in-gb 512 -np 36 -cf CP --allow-run-as-root --param ***parameters.checkpoint.fsync=true*** parameters.framework=pytorch parameters.model.parallelism.pipeline=32 parameters.model.parallelism.tensor=16

NFS Perf Test Toolbox

- Client-side NFS
  - nfsstat -m — negotiated vers/proto/rsize/wsize.
  - nfsiostat 1 — per-mount ops/s, kB/s, avg RTT/queue
  - mountstats /mnt/nfs — per-op latencies.
  - rpcctl client

- Server-side NFS
  - nfsstat -s — server op mix & retrans.
  - watch -n1 cat /proc/net/rpc/nfsd — RPC queues/threads.
  - cat /proc/fs/nfsd/{threads,versions,portlist,max_block_size} — live params (6.16+ max_block_size).
  - rpcinfo -p | egrep '2049|20049' — TCP(2049) & RDMA(20049) services.
  - ss -lntp | egrep ':2049|:20049' — listeners & bound IPs.

NFS Perf Test Toolbox

- CPU & scheduler
  - mpstat -P ALL 1 — per-CPU utilization.
  - pidstat -t -C nfsd 1 — per-thread nfsd usage.
  - perf top / perf record -g (optional deep dive).
- Storage / FS backend
  - iostat -x 1 — device util/await/avgqu-sz.
  - xfs_info /mnt/fs — stripe/alignment sanity (XFS).

How easy it is to do **badly**

```
W0000 00:00:1756809703.126950   88089 computation_placer.cc:177] computation placer already registered. Please check linkage and avoid linking the same targe
[OUTPUT] 2025-09-02T12:41:48.109123 Running DLIO [Training & Checkpointing] with 8 process(es)
[OUTPUT] 2025-09-02T12:41:49.433245 Model size: 0.000010 GB
[OUTPUT] 2025-09-02T12:41:49.485327 Total checkpoint size: 0.000010 GB
[OUTPUT] 2025-09-02T12:41:49.545071 Max steps per epoch: 178 = 1 * 10000 / 7 / 8 (samples per file * num files / batch size / comm size)
[OUTPUT] 2025-09-02T12:41:49.634427 Starting epoch 1: 178 steps expected
[OUTPUT] 2025-09-02T12:41:49.692760 Starting block 1
[OUTPUT] 2025-09-02T13:00:50.200696 Ending block 1 - 178 steps completed in 1140.51 s
[OUTPUT] 2025-09-02T13:00:50.205989 Epoch 1 - Block 1 [Training] Accelerator Utilization [AU] (%): 5.3253
[OUTPUT] 2025-09-02T13:00:50.206400 Epoch 1 - Block 1 [Training] Throughput (samples/second): 9.1782
[OUTPUT] 2025-09-02T13:00:50.206727 Epoch 1 - Block 1 [Training] Computation time per step (second): 0.3231+/-0.0000 (set value: {'mean': 0.323})
[OUTPUT] 2025-09-02T13:00:50.214275 Ending epoch 1 - 178 steps completed in 1140.58 s
[OUTPUT] 2025-09-02T13:00:50.229324 Starting epoch 2: 178 steps expected
[OUTPUT] 2025-09-02T13:00:50.230466 Starting block 1
[OUTPUT] 2025-09-02T13:19:35.228000 Ending block 1 - 178 steps completed in 1125.00 s
[OUTPUT] 2025-09-02T13:19:35.230258 Epoch 2 - Block 1 [Training] Accelerator Utilization [A ] (%): 5.3205
[OUTPUT] 2025-09-02T13:19:35.230608 Epoch 2 - Block 1 [Training] Throughput (samples/second : 9.1699
[OUTPUT] 2025-09-02T13:19:35.230905 Epoch 2 - Block 1 [Training] Computation time per step [illegible] (set value: {'mean': 0.323})
[OUTPUT] 2025-09-02T13:19:35.234168 Ending epoch 2 - 178 steps completed in 1125.00 s
[OUTPUT] 2025-09-02T13:19:35.250804 Starting epoch 3: 178 steps expected
[OUTPUT] 2025-09-02T13:19:35.251325 Starting block 1
[OUTPUT] 2025-09-02T13:38:21.594688 Ending block 1 - 178 steps completed in 1126.34 s
[OUTPUT] 2025-09-02T13:38:21.596913 Epoch 3 - Block 1 [Training] Accelerator Utilization [AU] (%): 5.3165
[OUTPUT] 2025-09-02T13:38:21.597288 Epoch 3 - Block 1 [Training] Throughput (samples/second): 9.1631
[OUTPUT] 2025-09-02T13:38:21.597581 Epoch 3 - Block 1 [Training] Computation time per step (second): 0.3231+/-0.0000 (set value: {'mean': 0.323})
[OUTPUT] 2025-09-02T13:38:21.601204 Ending epoch 3 - 178 steps completed in 1126.35 s
[OUTPUT] 2025-09-02T13:38:21.616529 Starting epoch 4: 178 steps expected
[OUTPUT] 2025-09-02T13:38:21.617093 Starting block 1
[OUTPUT] 2025-09-02T13:57:05.304699 Ending block 1 - 178 steps completed in 1123.69 s
[OUTPUT] 2025-09-02T13:57:05.306801 Epoch 4 - Block 1 [Training] Accelerator Utilization [AU] (%): 5.3294
[OUTPUT] 2025-09-02T13:57:05.307202 Epoch 4 - Block 1 [Training] Throughput (samples/second): 9.1853
[OUTPUT] 2025-09-02T13:57:05.307527 Epoch 4 - Block 1 [Training] Computation time per step (second): 0.3231+/-0.0000 (set value: {'mean': 0.323})
[OUTPUT] 2025-09-02T13:57:05.311171 Ending epoch 4 - 178 steps completed in 1123.69 s
[OUTPUT] 2025-09-02T13:57:05.328231 Starting epoch 5: 178 steps expected
[OUTPUT] 2025-09-02T13:57:05.328809 Starting block 1
```

Let’s enable **RDMA**

```
[METRIC] ==========================================================
[METRIC] Number of Simulated Accelerators: 8
[METRIC] Training Accelerator Utilization [AU] (%): 58.5000 (8.6964)
[METRIC] Training Throughput (samples/second): 100.8224 (14.9877)
[METRIC] Training I/O Throughput (MB/second): 14095.9107 (2095.4251)
[METRIC] train_au_meet_expectation: fail
[METRIC] ==========================================================
```

We need to do some tuning to achieve
reasonable performance

What affects performance?

Tuning Steps:

1. Backend Storage (don’t forget about Degraded and Rebuild mode)
2. Filesystem format and mount options
3. NFS server options and capabilities
4. Network options (won’t be covered today)
5. NFS client options
6. Test Parameters

How to read the results

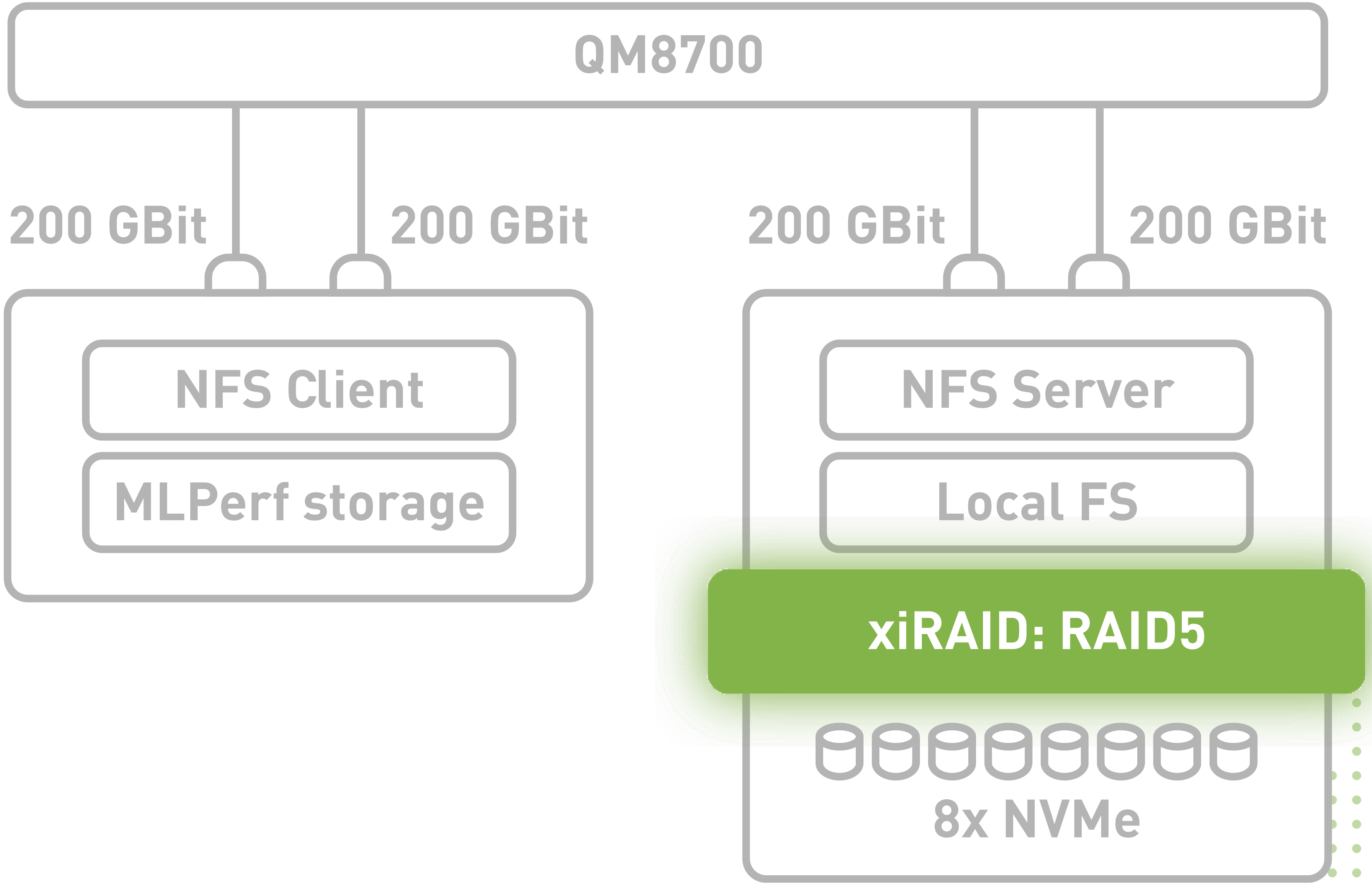
Workloads	Parameters
	Threads=1
3D U-Net	1@98%
CPU load @ Training	49%@1
Checkpointing Save / Load	2.1 GBps / 10 GBps
CPU Load @ Checkpointing	95%@1

Max count of H100 GPU @ AU

Average server CPU load generated by NFSd @ Number of CPU cores utilized

Performance

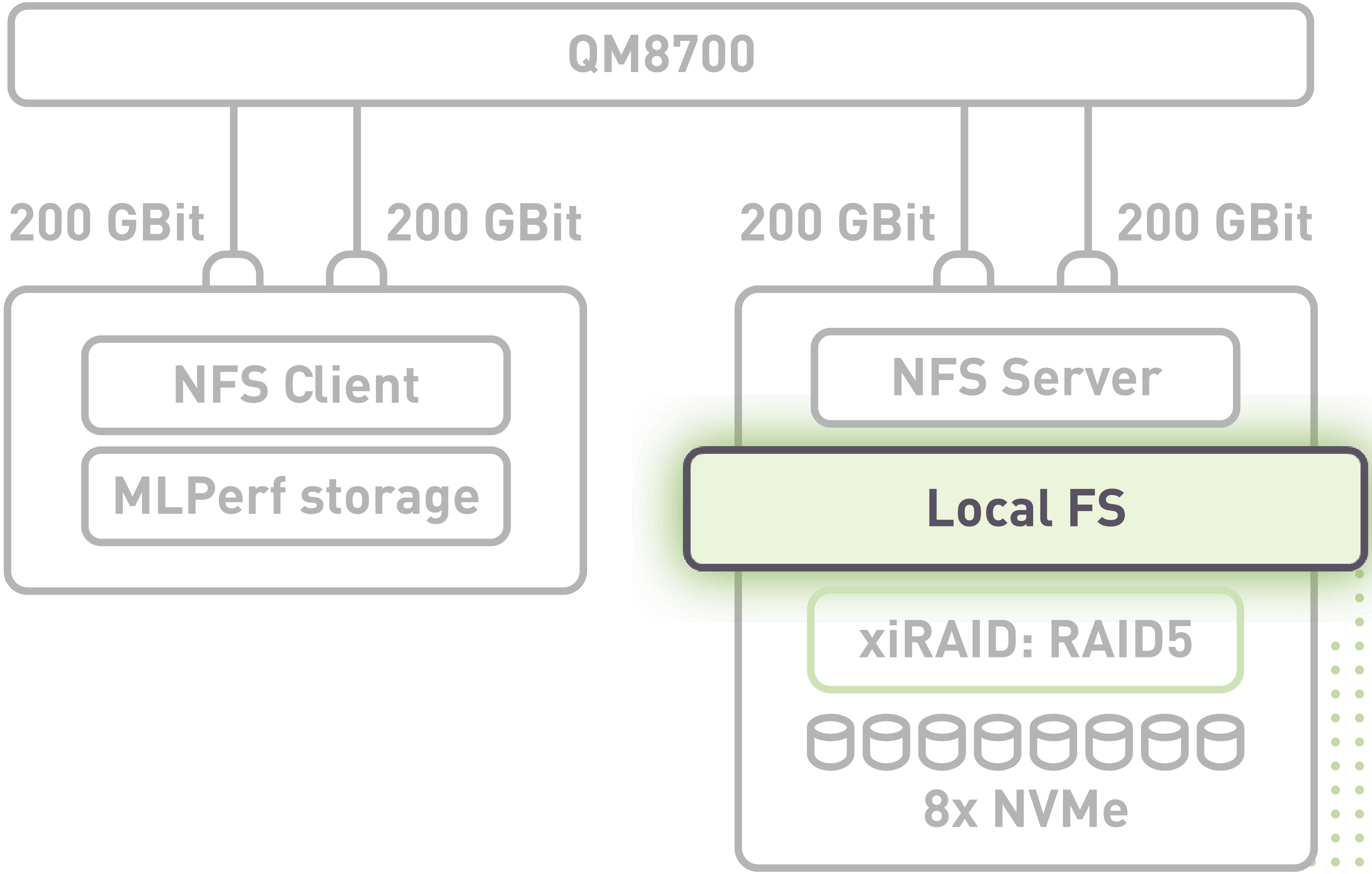
The results have been rounded for simplicity.


QM8700
200 GBit
200 GBit
200 GBit
200 GBit
NFS Client
MLPerf storage
NFS Server
Local FS
xiRAID: RAID5
8x NVMe

Backend storage health impact

	xiRAID Normal	MDRAID Normal	xiRAID Degraded	MDRAID Degraded
3D U-Net	14 @ 93%	13 @ 90%	14 @ 91%	1 @ 56%
CPU load 3D U-Net	55% @ 48	50% @ 48	52% @ 48	11% @ 48
Checkpointing Save / Load	17.2 GBps / 18.5 GBps	3.2 GBps / 18.6 GBps	15.4 GBps / 17.6 GBps	2.6 GBps / 2.3 GBps
CPU load Checkpointing	33% @ 48	15% @ 48	31%@48	11% @ 48

Results achieved with the Server and Client setting are described further


QM8700
200 GBit
200 GBit
200 GBit
200 GBit
NFS Client
MLPerf storage
NFS Server
Local FS
xiRAID: RAID5
8x NVMe

Filesystem format and mount options

	XFS default	XFS OPT	EXT4 DEF	EXT4 OPT
3D U-Net	18@91%	22@94%	16@95%	18@91%
Checkpointing Save / Load	17.2 GBps / 20.3 GBps	28.1GBps / 26.4 GBps	18.5 GBps / 16.4 GBps	21.8 GBps / 20.6 GBps
FIO 1M sequential WRITE/READ	55.4 GBps / 56.4 GBps	55.8 GBps / 56.4 GBps	55.4 GBps / 56.3 GBps	55.5 GBps / 56.4 GBps
FIO 4k random READ async	7.5 M IOPS @ 494 us 95 lat	7.5 M IOPS @ 477 us 95 lat	7.5 M IOPS @ 481 us 95 lat	7.5 M IOPS @ 475 us 95 lat
FIO 4k random READ sync	577 k IOPS @ 99 us 95 lat	582 k IOPS @ 92 us 95 lat	557 k IOPS @ 102 us 95 lat	571 k IOPS @ 92 us 98 lat

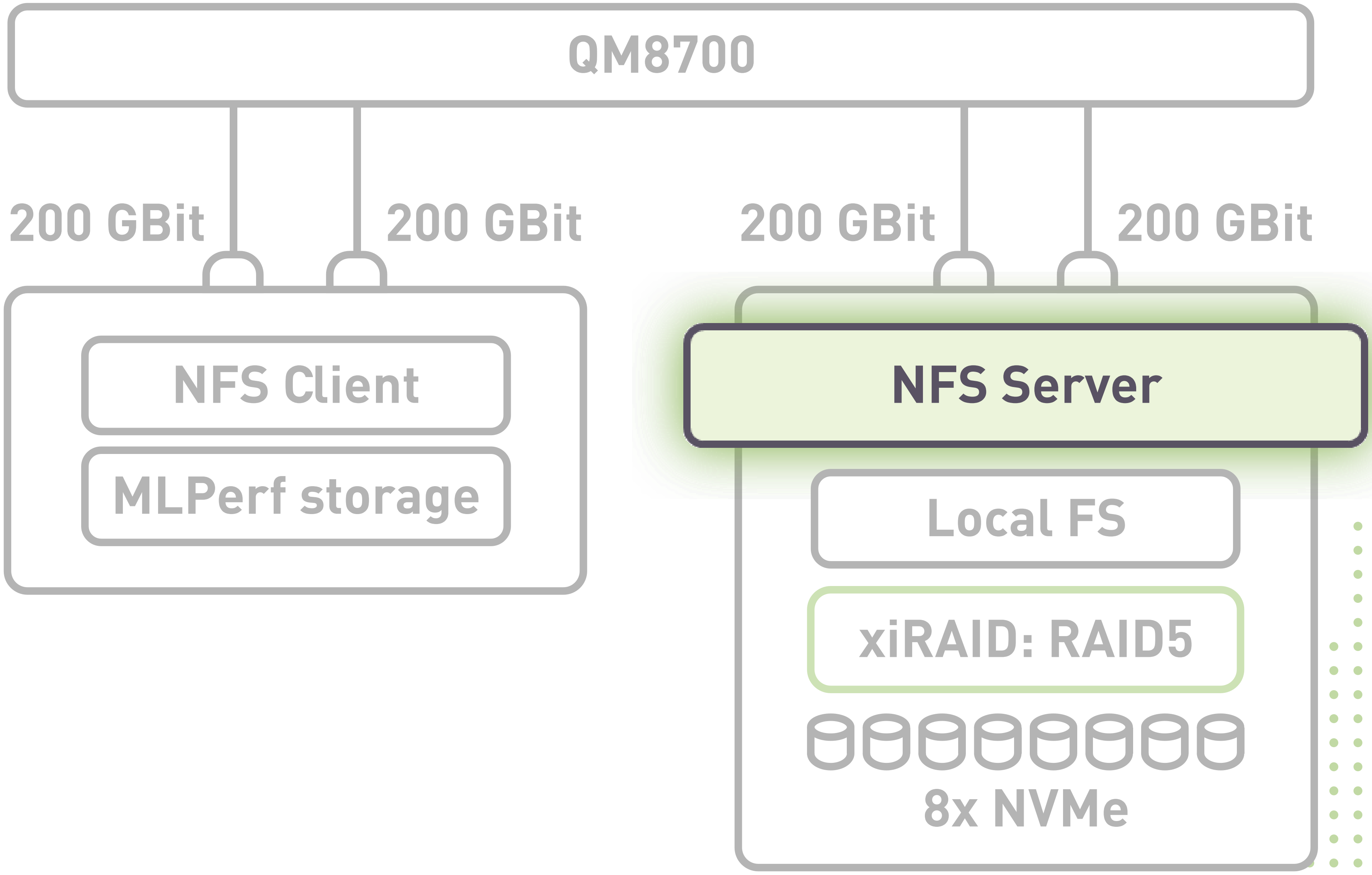
Optimal XFS Settings

- sudo mkfs.xfs -f -b size=4096 -d su=64k,sw=7,agcount=128 logdev=/dev/xi_raid10 sectsize=4096,size=1024m /dev/xi_raid6
- sudo mount -t xfs -o noatime,nodiratime,logbsize=256k,logbufs=8,allocsize=1M,largeio,inode64,logdev=/dev/xi_raid10 /dev/xi_raid6 /srv/nfs/

File system tuning recommendations

- **End-to-end alignment reduces wasted stripes.** Format with correct RAID hints (e.g., mkfs.xfs -d su=<stripe>,sw=<width>) so writes land on **full stripes when possible**.
- **External log reduces checkpoint stalls.** Place the XFS log on a fast NVMe (-l logdev=/dev/… ,sectsize=4096,size=2–4G) to cut metadata/journal contention during rename()+fsync() heavy checkpoints.
- **Parallelism from AGs.** Use sensible **AG count** (e.g., -d agcount=64–128 for multi-core servers) to enable parallel allocators without excessive fragmentation.

- **Mount options deliver performance gains.** Prefer noatime,inode64,logdev=/dev/… (and keep default delayed logging).
- **Increase device readahead** for scans (blockdev --setra 16384–65536), and smooth write-back with vm.dirty_bytes / vm.dirty_background_bytes.
- **Expected impact.** Typically, **+20−30%** sustained BW and **smoother tails** on sequential I/O vs. default format/mount; CPU per GB written often drops as well.


QM8700
200 GBit
200 GBit
200 GBit
200 GBit
NFS Client
MLPerf storage
NFS Server
Local FS
xiRAID: RAID5
8x NVMe

NFS — What's New (Linux 5.3 → 6.17)

- **Parallelism & bandwidth:** nconnect (multi-TCP per mount) and **NFSv4.1 session trunking** (multi-IP, HA) remove single-flow limits and fully utilize fast NICs.
- **Smarter data & metadata: READ_PLUS** skips sending zero-filled holes in sparse files; **writes=eager/wait** gives precise write semantics; fewer redundant **GETATTR** calls
- **LOCALIO (loopback):** bypass TCP/RPC for same-host client+server; now with **O_DIRECT** for near-native performance; visibility in sysfs (6.16+)
- **NFS Inter-Server Copy:** Client triggers a **server-to-server** copy; bytes flow from **source NFS server** → **destination NFS server** without passing through the client.

NFS server options

```
[nfsd]
debug=0
threads=64
host=10.10.10.1,30.30.30.1
port=2049
grace-time=45
lease-time=45
udp=n
tcp=y
vers4.1=y
vers4.2=y
rdma=y
rdma-port=20049
```

nfsd threads — practical recommendations

- **Start point:** threads ≈ number of effective cores servicing the NFS NIC (think physical cores feeding that NIC's RX/TX queues; don't count SMT unless you've verified wins).
- **Rule-of-thumb bands:**
  - Small/medium fleets: **32–64** threads.
  - Large fan-in (100s of clients) or heavy small-IO metadata: **64–96**.
  - Going **>128** rarely helps and often increases lock contention/context switches.
- **Turn up when:** RPC backlog > 0 under load, nfsd worker CPU < 70% but requests queue;
- **Turn down when:** run-queue per core > 2, system time spikes
- **Validate:** watch /proc/net/sunrpc/nfsd (queue/threads), nfsstat -s, and mpstat -P ALL 1 during load.

Server options: number of nfsd threads

	Threads=1	Threads=CPU core count	Threads=Defaults (8)	Threads=2 CPU core count
3D U-Net	1@98%	14@93%	7@91%	10@93%
CPU load 3D U-Net	67%@1	60%@48	78%@8	90%@48
Checkpointing Save / Load	2.1 GBps / 4.2 GBps	7.3 GBps / 17.5 GBps	7.3 GBps / 17.3 GBps	7.6 GBps / 17.2 GBps
CPU Load Checkpointing	95%@1	15%@48	20%@24	15%@48
Fio Seq Writes/Reads	2.9 GBps / 5.2 GBps	26.4GBps / 41.1GBps	13.2GBps / 27.5GBps	24.7GBps / 26.7GBps
Fio Random Reads 4k	112k @ 628 us 95% lat	333k @ 190 us 95% lat	236 k @ 192 us 95% lat	331k @ 327 us 95% lat

Client mount parameters:

mount -t nfs -o vers=4.2, proto=rdma, port=20049, rsize=1048576, wsize=1048576, max_connect=16, sync, trunkdiscovery
nfsserv:/srv/nfs/ /mnt/nfstest

NFS server configuration recommendations

- **Defaults aren't enough.** Out-of-the-box NFS/NFSD settings limit throughput; they don't deliver acceptable performance for modern ML/AI or checkpointing workloads for large – scale NFS servers.
- **Set threads ≈ cores.** Adjust with awareness of the **storage backend's CPU demand** (RAID / erasure coding / SPDK / checksumming) so you don't starve it. Recommendations differ for TCP with multiple streams.
- **More threads ≠ better performance.** Increasing nfsd threads **beyond core count** typically adds context switches and lock contention.
- **Threads ≈ cores ⇒ NIC-limited performance.** With proper IRQ/NUMA locality and no storage bottleneck, threads near core count achieves **maximum practical NIC throughput** (approaches line-rate).
- **Checkpoint is different.** For large sequential Checkpoint operations, NFSD thread count has **negligible effect after NFSd threads count > 4**; observed checkpoint performance remains **unsatisfactory** under current settings.
- **Implication.** Improving Checkpoint requires **further system-level tuning** (filesystem/journal, write-back policy, I/O path, and data layout)—not just NFSD thread adjustments.

Export options: wdelay vs no_wdelay

	wdelay	no_wdelay
Checkpointing Save / Load	7.3 GBps /17.5 GBs	12.8 GBps /17.5 GBps
FIO Sequential write	22.6 GBps	23.5 GBps

Since the checkpointing workload is highly synchronous and latency-sensitive, enabling the no_wdelay parameter significantly improves performance.

Client mount parameters:

mount -t nfs -o vers=4.2, proto=rdma, port=20049, rsize=1048576, wsize=1048576,
max_connect=16,sync, trunkdiscovery
nfsserv:/srv/nfs/ /mnt/nfstest

Export options: sync vs async

	sync	async
Checkpointing Save / Load	12.8 GBps /17.5 GBps	13.5 GBps /18.5 GBps
FIO Sequential write/read	23.5 GBps / 41.2 GBps	25.3 GBps / 41.2 GBps

Async mode shows slightly better performance, but on practice it tends to be unstable on more powerful systems.

Client mount parameters:

mount -t nfs -o vers=4.2, proto=rdma, port=20049, rsize=1048576, wsize=1048576, max_connect=16, nconnect=8, async, trunkdiscovery nfsserv:/srv/nfs/ /mnt/nfstest

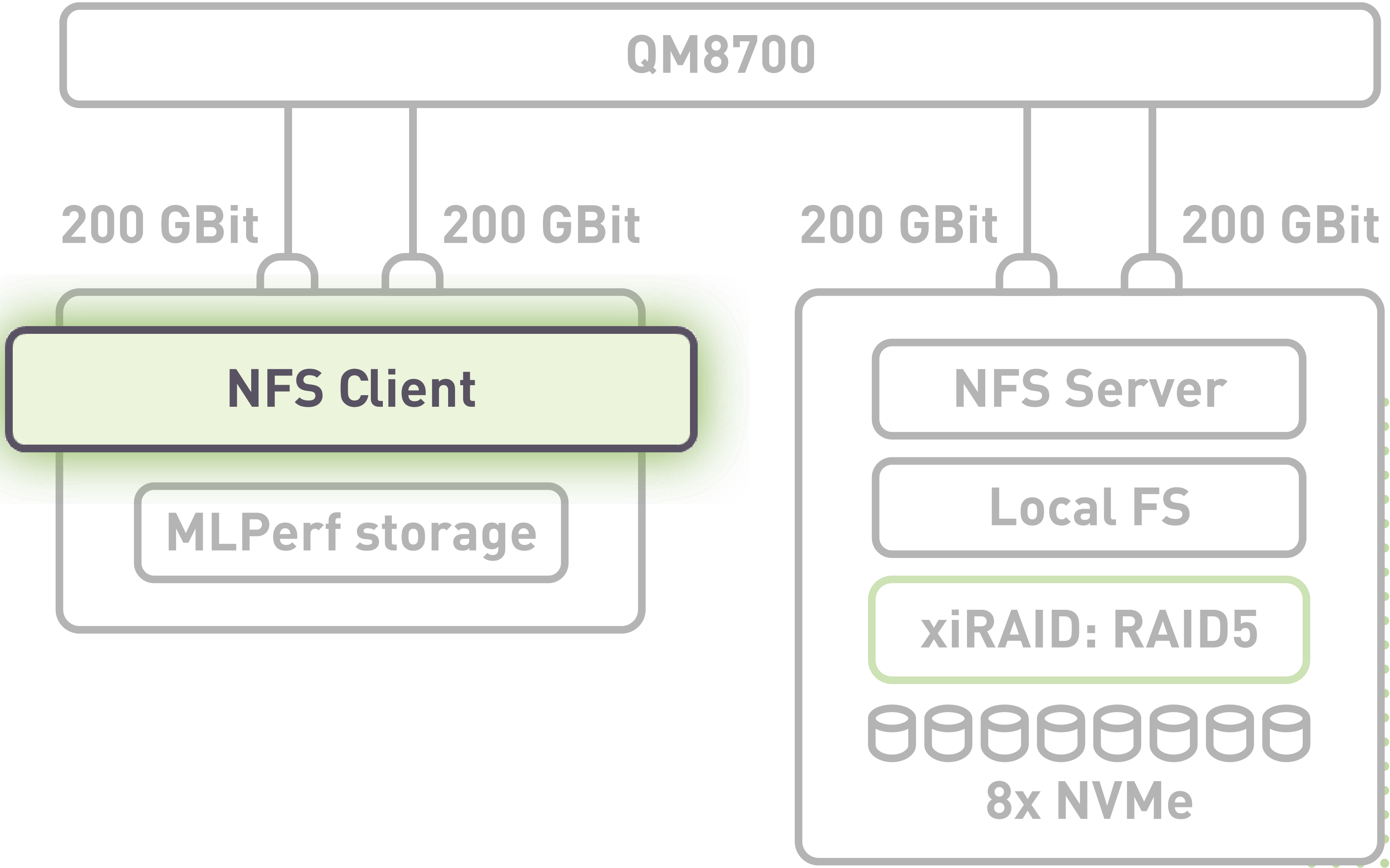
Conclusions: NFS Server Settings for High-Performance AI

- **Enable NFSoRDMA**
- **Right-size nfsd threads (≈ number of physical cores)**
  - Match threads to effective cores/NUMA (avoid oversubscription);
- **Advertise multiple server IPs / listen on all interfaces**
  - Present a hostname with multiple A/AAAA records and ensure nfsd listens on them. This enables **session trunking** so clients can spread load across paths and NIC queues.
- **For SYNC workloads, prefer no_wdelay (with sync exports)**
  - Eliminates small write coalescing delays; combine with a fast journal/log device. (If policy allows, async yields max throughput—let apps fsync() at checkpoints.)

→

Expected impact

With correct backend and NIC tuning, these changes typically improve aggregate throughput and stabilize P95/P99 by **~2–4X** over defaults, keeping GPUs fed even under heavy checkpoints.


QM8700
200 GBit
200 GBit
200 GBit
200 GBit
NFS Client
MLPerf storage
NFS Server
Local FS
xiRAID: RAID5
8x NVMe

NFS Client:
What we can change for high performance

Bucket 1 — NFS module tunables (system-wide):

- NFS requests concurrency

Bucket 2 — Per-mount options (tuned per share/workload):

- Parallelism: multiple streams and session trunking
- Transport & version
- I/O size
- Write policy

NFS module tunables (system-wide)

Param	Training / Checkpointing (Throughput)	Mixed (RAG)
max_session_slots	**! 128–256**	128–192
max_session_cb_slots	32–64	24–48
callback_nr_threads	8–12	8–12
nfs4_disable_idmapping	1 if sec=sys & unified UID/GID; else 0	per env
nfs_idmap_cache_timeout	600–1200s	600–900s
delay_retrans	-1 (default backoff)	0–1
nfs_access_max_cachesize	1M	128k–256k
enable_ino64	1	1

max_session_slots (Parallelism = Bandwidth)

- **What.** Maximum number of outstanding NFSv4.1 requests negotiated by the client.
- **Why for AI.** High concurrency is crucial for saturating fast NICs during large tensor/checkpoint I/O.
- **Recommend.** 128–256 for bandwidth-bound training; keep closer to 64–128 for pure low-latency small I/O.
- **Set:**

```
# Temporary
    echo 256 | sudo tee /sys/module/nfs/parameters/max_session_slots
# Persistent (/etc/modprobe.d/nfs.conf)
    options nfs max_session_slots=256
```

- **Watch-outs**

Benefits depend on server slot limits; too high can increase queuing delay.

NFS client kernel module options: Defaults vs Optimal parameters

	Defaults	Optimal (Training preset)
3D U-Net	12@92%	14@93%
Checkpointing Save / Load	12.8GBps / 17.5GBps	17.2 GBps / 18.5 GBps
FIO Sequential write/read	23.6 GBps / 41GBps	29.2 GBps / 43.2 GBps

Client mount parameters:

mount -t nfs -o vers=4.2, proto=rdma, port=20049, rsize=1048576, wsize=1048576, max_connect=16,
sync, trunkdiscovery
nfsserv:/srv/nfs/ /mnt/nfstest

Per-mount options. nconnect and max_connect

- **nconnect=<1..16>**: Multiple TCP/RDMA connections to one server IP for a given mount; boosts throughput and mitigates head-of-line blocking.
- **max_connect=<1..16>**: For **NFSv4.1+ session trunking** across **multiple server IPs** that belong to the same server; improves bandwidth & resiliency. Mount via each IP (or rely on trunking discovery where supported).
- **Rule of thumb (TCP)**
  - **Throughput per lane ≈ 1.5–2.0 GB/s** (sync-heavy, checkpoint/recording).
  - IOPs per lane **≈ 110k @4k**
  - **Ajust a number of connections with expected performance**

Other options are described in the Appendix

TCP Multiple Streams

- **Single stream = single bottleneck**
  - **~ 2.5 GBps Reads/ 1.6 GBps Writes** per single connection
  - 110k 4k IOPS per single connection
  - One TCP flow ⇒ one congestion window, one socket queue, more head-of-line blocking.
  - A single receive/transmit queue pair under-utilizes RSS and CPU cores.
- **nconnect: parallel lanes on one mount (one server IP)**
  - Opens **N independent TCP connections** per mount.
  - Aggregates congestion windows; spreads packets across **RSS queues/CPUs**.
  - More **in-flight RPCs** without fighting a single socket's limits.
- **Session trunking scale the path to data**
  - **NFSv4.1/4.2 session trunking** fans one session across **multiple server IPs** (more paths, HA).

RDMA vs TCP Multistream (nconnect option) with 1 IP

	RDMA Defaults	RDMA nconnect=16	TCP Defaults	TCP nconnect=4	TCP nconnect=8	TCP nconnect=16
3D U-Net	7@94%	7@93%	1@64%	3@96%	5@92%	6@97%
CPU load 3D U-Net	28%@48	28%@48	71%@1	75%@4	78%@8	74%@16
Checkpointing Save / Load	15.7 GBps / 16.1 GBps	17.1 GBps / 16.4 GBps	1.6 GBps / 2.1 GBps	6.2 GBps / 8.1 GBps	11.7 GBps / 14.2 GBps	13.7 GBps / 15.6 GBps
CPU load Checkpointing	33% @ 48	34% @ 48	72%@1	74% @ 4	78% @ 8	82% @ 16
Fio Sequential Write / Read	20.9 GBps / 22.5 GBps	23.6 GBps / 23.1 GBps	1.6 GBps / 2.6 GBps	7.2 GBps / 10 GBps	13.2 GBps / 14.5 GBps	17.2 GBps / 18.2 GBps
Fio Random Reads	110k @ 545 95 lat	335k @ 151 us lat	49k @ 1.3 ms 95 lat	160k @ 570 us 95 lat	260k@ 337 us 95 lat	279k @ 288 us 95 lat

Client mount parameters:

mount -t nfs -o vers=4.2, proto=rdma, port=20049, rsize=1048576, wsize=1048576, nconnect=**{variable},** sync, nfsserv:/srv/nfs/ /mnt/nfstest

Back to count of NFSd threads

nconnect	count of NFSd threads	NFSoRDMA	NFSoTCP
1	1	1@98% 2.9 GBps / 4.8 GBps	1@64% 1.6 GBps / 2.6 GBps
1	48	7@94% 20.9 GBps / 22.5 GBps	1@58% 1.6 GBps / 2.3 GBps
4	4	5@93% 10.5 GBps / 23.7 GBps	3@96% 7.2 GBps / 10 GBps
4	48	7@92% 22.5 GBps / 24.1 GBps	2@98% 7.2 GBps / 8.3 GBps
8	8	7@91% 13.2GBps / 23.5GBps	5@92% 13.2 GBps / 14.5 GBps
8	48	7@94% 23.2GBps / 23.5GBps	5@90% 13.0 GBps / 13.7 GBps
16	16	7@93% 18.6 GBps / 23.1 GBps	6@97% 17.2 GBps / 18.2 GBps
16	48	7@93% 23.6 GBps / 23.1 GBps	6@93% 17 GBps / 15.7 GBps

A single 200 Gbit network Interface

RDMA and TCP Session Trunking (max_connect+trunkdiscovery) with 1 and 2 ports

	RDMA max_connect=16	TCP max_connect=16
3D U-Net	14@93%	10@96%
CPU load	55%@48	82%@16
Checkpointing Save / Load	17.2 GBps / 18.5 GBps	14.2 GBps / 17.9 GBps
CPU load	33%@48	84%@16
Fio Sequential Read / Write	29.2 GBps / 43.2 GBps	19.2 GBps / 32.8 GBps
Fio Random Reads	335k @ 154 us 95 lat	282k @ 255 us 95 lat

Client mount parameters:

mount -t nfs -o vers=4.2, proto=rdma, port=20049, rsize=1048576, wsize=1048576, nconnect={variable}, max_connect={variable} sync, trunkdiscovery
nfsserv:/srv/nfs/ /mnt/nfstest

Ubuntu session trunking issue

With vers=4.2,proto=tcp,trunkdiscovery, nconnect=8,max_connect=16 the client creates **8 TCP sessions** to IP#1 but only **1 session** to IP#2.

As result, we get poor fan-out across paths; we can’t reach expected throughput on dual-port controllers.

Workaround:

- Assign **multiple secondary IPs on both controller ports** (e.g., 4 IPs per port).
- Publish one **hostname** with **all** those A-records.
- Remount with trunking; the client opens transports across **many IPs**, not just two.

```
root@xiNAS-D3D92343D893194A:/home/xinnor# sudo rpcctl client | grep -E 'xprt-.*tcp,'
        xprt-0: tcp, 10.10.10.10 [main]
        xprt-1: tcp, 10.10.10.10
        xprt-16: tcp, 30.30.30.10
        xprt-2: tcp, 10.10.10.10
        xprt-3: tcp, 10.10.10.10
        xprt-4: tcp, 10.10.10.10
        xprt-5: tcp, 10.10.10.10
        xprt-6: tcp, 10.10.10.10
        xprt-7: tcp, 10.10.10.10
        xprt-0: tcp, 10.10.10.10 [main]
        xprt-1: tcp, 10.10.10.10
        xprt-16: tcp, 30.30.30.10
        xprt-2: tcp, 10.10.10.10
        xprt-3: tcp, 10.10.10.10
        xprt-4: tcp, 10.10.10.10
        xprt-5: tcp, 10.10.10.10
        xprt-6: tcp, 10.10.10.10
        xprt-7: tcp, 10.10.10.10
```

Client side conclusions

Core module parameters (system-wide; set once)

- Raise concurrency ceilings on high-perf systems (**~+15% vs defaults**):

Per-mount tuning (per share; per workload)

- **Parallelism (TCP):** use nconnect=8–16 to open many lanes per mount. On a single 100–200 Gb link, this typically reaches **~80% of RDMA** on the same NIC.
- **Parallelism (multi-IP):** enable **session trunking** (trunkdiscovery,max_connect=…) so lanes spread across multiple server IPs/NIC queues.

RDMA specifics

- nconnect does **not** massively lift sequential flow rate on RDMA (already low-overhead), but it helps **small-block random** paths that otherwise bottleneck on a single connection (**~110k ops/s** ceiling seen).
- For linear scale on RDMA, add **session trunking** (more IPs/paths), not just more lanes to one IP.

Client side conclusions

Threading guidance (server tie-in)

- **TCP, single busy client:** align **nfsd threads ≈ total client lanes** to avoid server-side queuing.
- **Many clients / high-core servers:** set threads ≈ **physical cores** (with sunrpc.svc_pool_mode=percpu).
- **RDMA:** fewer threads can suffice (lower per-op CPU); still ensure you’re not starved under bursts.

Ubuntu TCP trunking quirk (FYI)

- Symptom: only **1 lane** to the second IP with nconnect>1.
- **Workaround:** assign **multiple secondary IPs per port** and mount via a hostname listing them; the client will fan out across all addresses.

NFS overhead: best NFS tuning vs best local file system

	Local FS	NFSoRDMA Threads=CPU core count	RATIO
3D U-Net	22@94%	14@93%	63%
Checkpointing Save / Load	28.1GBps / 26.4 GBps	17.2 GBps / 18.5 GBps	61% / 70%
Fio Seq Writes/Reads	55.8 GBps / 56.4 GBps	29.2 GBps / 43.2 GBps	52% / 76%
Fio Random Reads 4k (sync)	582 k IOPS @ 92 us 95 lat	335k @ 154 us 95% lat	57%

ℵINNOR

NFS Local IO

NFS LOCAL IO

- **What is it:** Local fast path that preserves NFSv4 semantics while bypassing the network stack when client and nfsd are on the **same machine.**
- Lower **P95/P99 latency,** fewer context switches/IRQs, lower CPU overhead, higher sustained BW for big sequential I/O.
- **Caveat:** Results do **not** reflect multi-node behavior (no NIC queues, no nconnect, no RDMA link effects).

Usecases:

- **Tier-0 Training Scratch:** on-node NVMe exported via LocalIO keeps GPU feeders hot;
- **Checkpoint Sink + Async Push:** write checkpoints locally at wire-speed; a background job mirrors to NAS/object/PFS. Result: fast fsync locally, policy-driven durability later.
- **RAG / Indexing Intermediate Store:** local write-heavy index builds

NFS LocalIO challenges and proposed solution

The challenges

- **Single-node failure domain.** Local media = no built-in HA. A disk/node failure can stall training and risks data loss without extra protection.
- **Capacity & scale limits.** Local chassis slots bound capacity; adding/reshuffling drives is intrusive and not elastic across nodes.
- **QLC-era copy times explode.** With 122 TB today (244 TB tomorrow), (re)seeding or evacuating LocalIO via ordinary NFS takes *a very long time*—per-share throughput, metadata overhead, and network hops become the bottleneck.

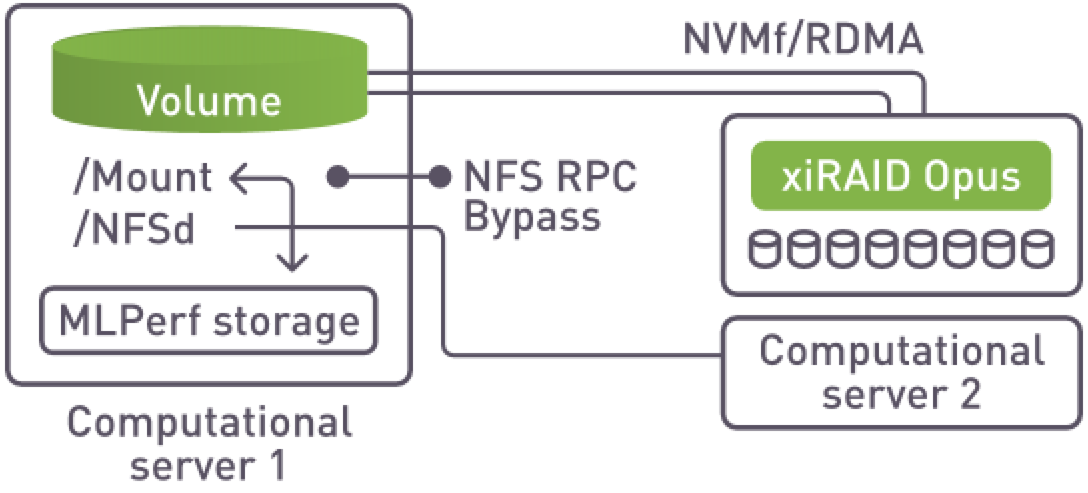
The fix (architecture)

- **Back LocalIO with a protected, network volume.** Deliver a high-performance, fault-tolerant block volume (erasure-coded) over **NVMe/TCP or NVMe/RDMA** to the compute node.
- **Run NFS server on top of that volume and use LocalIO for apps.**
- Apps see the NFS path via LocalIO (no NIC in the hot path), while durability and scale live in the backend.

The results

- **Elastic capacity on demand.** Provision and **grow the volume online**; the NFS export expands without host rebuilds.
- **Mobility without bulk copying.** **Detach/reattach** the volume to another node for maintenance or failover; if needed, migrate fast via **NVMe/RDMA** rather than file-level copies.
- **Faster (re)population.** Use **NFS v4.2 inter-server copy**, block-level copy, or direct NVMe-oF reattach to seed/evacuate datasets much faster than client-mediated NFS copies.

Storage Disaggregation and NFS LocalIO



Run an NFS server on the node
Export that filesystem and build a **single, unified namespace**. Remote nodes consume it over TCP (nconnect) or RDMA as appropriate.

Local I/O fast path
On the hosting node, apps hit the **NFS Local I/O** path (kernel short-circuit, no TCP/RDMA), avoiding extra **RPC** overhead and minimizing tail latency/CPU.

Present storage with xiRAID Opus
Expose volumes to compute nodes via **NVMe-oF (RDMA or TCP)** — line-rate, low-latency block access right on the node.

Format & mount locally
On each compute node, create an aligned local filesystem (e.g., XFS with proper su/swidth,) and mount it for the workloads.

Why this works:

- **Flexibility**: Opus composes and places capacity exactly where compute needs it.
- **Performance**: direct NVMe-oF locally; NFS provides high-BW multi-reader/writer semantics to the rest of the cluster.
- **Operational simplicity:** one POSIX view, standard tools (nfsd, nfsstat, mountstats), easy policy (quotas, auth).

NFS LOCAL IO + NVMf

	NFS Dual Port, TCP	NFS Dual Port, RDMA	NVMf/TCP	NVMf/RDMA
3D U-Net	10@96%	14@93%	14@91%	16@93%
Checkpointing Save / Load	14.2 GBps / 17.9 GBps	17.2 GBps / 18.5 GBps	24 GBps / 26 GBps	24GBps / 28 GBps
Fio Sequential Write / Read	19.2 GBps / 32.8 GBps	29.2 GBps / 43.2 GBps	36.4 GBps / 39.1 GBps	43.2 GBps / 43.9 GBps
Fio Random Read (async)	282k @ 255 us 95 lat	335k @ 154 us 95% lat	1M IOPS @ 289 us lat	1M IOPS @ 212 us 95 lat
Fio Random Read (sync)	282k @ 255 us 95 lat	335k @ 154 us 95% lat	388k IOPS @ 199 us 95 lat	365k IOPS @ 180 us 95 lat

Conclusions (1)

- Backend storage for NFS should provide performance for network saturation in both ***normal*** (2-5X) **and *degraded*** (20X) **modes**
- Local File System should be tuned, ***XFS*** is the optimal: ***full stripe allocation***, ***external log*** and ***AGs parallelism*** are the most important settings
- NFS Server default settings aren’t enough.
  - ***NFSD threads*** equal to the **CPU cores** is optimal for training but not enough for checkpointing
  - **"*no_wdelay*"** siginificantly improve checkpointing **(2X)**
  - **"*async*"** **(15-20%)** further slightly improve checkpointing but it can influence on system stability

Conclusions (2)

- NFS Client should be tuned: "***max_session_slots"*** is the most important setting **(15%)** for the client kernel module.
- NFS client mount options matters for both training and checkpointing:
  - **nconnect** is providing scaling for TCP with 1 IP. Default nconnect is fine for RDMA with 1IP
  - each TCP lane as ~**1.5–2.0 Gb/s (writes)**. With up to **16 lanes per mount**, we can **budget and accumulate** throughput by adding lanes until we hit NIC or backend limits.
  - With nfsd threads count aligned to client lanes, **TCP+nconnect reaches ~70–85% of RDMA** on the same interface for streaming AI I/O.
  - **Session trunking (TCP) aggregate performance scales close to linearly** as lanes/paths are added. **RDMA trunking** scales cleanly
  - On Ubuntu, trunking may fully fan out only to the first IP. **Workaround:** assign **multiple secondary IPs per port** and mount via a hostname listing all of them; set max_connect ≥ IPs × nconnect. This restores multi-path fan-out.

Conclusions (3)

- **NFS LocalIO**
  - **RPC bypass gives low latency and high throughput.** LocalIO removes overhead on same-host client/server.
  - **Small-op boost:** With **asynchronous I/O**, LocalIO typically delivers **3–4X** higher performance on small operations vs standard NFS datapath.
  - **Pair with disaggregated storage for flexibility + durability.** Mount a high-performance, protected network volume under LocalIO to get **elastic capacity**, **easy scaling**, and **fast mobility** (grow/move without long file-level copies) and improves performance.

What's next

Objective

- Prove performance and stability on a bigger topology and quantify gains from server/client tuning at 400 Gb/s.

Topology under test:

- 2× storage nodes: NVMe PCIe Gen5 arrays, dual 400 Gb links each.
- 4× clients: 1x400GBit Each, 64 CPU Cores Each

Test matrix (A/B comparisons):

- NFSv4.2 RDMA vs TCP + nconnect (multichannel).
- SMB Direct vs SMB Multichannel

Workloads to run:

- MLPerf Storage "training" (datasize → datagen → run).
- Checkpoint streaming (1–4 MiB writes, multi-writer).
- GPU/Accelerator Utilization ≥ 90% where applicable.
- Target ≥ 90% of link rate sustained without tail blow-ups.

ℵINNOR

Thank you for attending!

Appendix

NFS server options

Server-side max payload per READ/WRITE can be raised to 4 MiB in 6.16 kernel.

```
# check current limit
cat /proc/fs/nfsd/max_block_size
# raise to 4 MiB (4194304) and apply
echo 4194304 | sudo tee /proc/fs/nfsd/max_block_size
sudo systemctl restart nfs-server    # or nfs-kernel-server
```

Client: check negotiated sizes:

```
nfsstat -m
grep -E 'rsize|wsize' /proc/self/mountstats
```

Server Options

vm.dirty_bytes = 1073741824 (1 GiB)
Absolute cap (bytes) at which a process doing writes must start **writeback itself.** Smooths large write bursts and prevents massive “all-at-once” flushes. Too high ⇒ long stalls during flush; too low ⇒ over-eager flushing.

vm.dirty_background_bytes = 268435456 (256 MiB)
Absolute threshold that wakes the kernel’s **background flusher threads** to start draining dirty pages. Keeps a steady writeback pipeline so foreground I/O isn’t jolted by sudden flushes.

vm.swappiness =
Biases the kernel to **avoid swapping** anonymous memory unless truly necessary, preserving page cache for filesystem I/O. Good for storage servers with ample RAM (reduces cache churn).

net.core.rmem_max = 268435456
Upper bound for **per-socket receive buffers**. Allows TCP/UDP autotuning (and RDMA ULPs using sockets) to grow windows on high-BDP paths. Doesn’t force buffers by itself; it raises the ceiling.

net.core.wmem_max = 268435456
Upper bound for **per-socket send buffers.** Lets autotuning open bigger send windows for long, fat links (useful with multi-stream TCP NFS).

net.core.netdev_max_backlog = 250000
Maximum packets queued on the **ingress backlog** when the NIC delivers faster than the stack can process. Higher values absorb short bursts and reduce drops; if set too high on an overloaded CPU, it can add queuing latency.

max_session_cb_slots + callback_nr_threads

What:

- max_session_cb_slots — parallel callbacks (delegations, pNFS recalls) the client can process from a server.
- callback_nr_threads — number of kernel threads handling those callbacks.
- **Why for AI:** With pNFS/flexfiles or heavy parallel opens/closes, responsive callback handling prevents stalls and delegation recalls from becoming a bottleneck.
- **Recommend:** max_session_cb_slots=32–64, callback_nr_threads=8–12 (up to 16 for metadata-intensive loaders).

Set:

- echo 64 | sudo tee /sys/module/nfs/parameters/max_session_cb_slots
- echo 12 | sudo tee /sys/module/nfs/parameters/callback_nr_threads
- # persistent
- options nfs max_session_cb_slots=64 callback_nr_threads=12

nfs4_disable_idmapping & nfs_idmap_cache_timeout

What:

- nfs4_disable_idmapping=1 (with sec=sys) skips v4 idmapping and uses numeric UID/GID directly.
- nfs_idmap_cache_timeout controls TTL of idmap cache.
- **Why for AI:** Reduces metadata RPC churn during massive parallel file access by many workers; keeps stat()/open() paths light.

Recommend:

- If all nodes share **identical numeric UID/GID**, set nfs4_disable_idmapping=1.
- nfs_idmap_cache_timeout=600–1200s (throughput) or 300–600s (latency-sensitive small-file workloads).
- **Set:** options nfs nfs4_disable_idmapping=1 nfs_idmap_cache_timeout=900
- **Watch-outs:** Only use nfs4_disable_idmapping=1 when UID/GID spaces are truly aligned.

delay_retrans (Fast Fail for Small-IO Paths)

- **What:** After server replies NFS4ERR_DELAY, limit retries before returning EAGAIN.
- **Why for AI:** Dataloaders and micro-services often prefer quick retry over long stalls.
- **Recommend:** 0–1 for latency-sensitive small I/O; keep -1 (default) for pure bulk-throughput training.
- **Set:**
- echo 1 | sudo tee /sys/module/nfs/parameters/delay_retrans
- # persistent
- options nfs delay_retrans=1

nfs_access_max_cachesize (Access Cache Budget)

- What: Global budget for caching ACCESS results (permission checks).
- **Why for AI:** Many processes (workers) touching vast directory trees benefit from a larger ACCESS cache, cutting metadata round-trips.
- **Recommend:** 128k–512k for large training sets; 64k–256k for small-file/latency paths to control memory.
- Set:
- echo 262144 | sudo tee /sys/module/nfs/parameters/nfs_access_max_cachesize
- options nfs nfs_access_max_cachesize=262144
- **Watch-outs:** Too small ⇒ excess RPC; too large ⇒ client RAM overhead.

I/O Sizes: rsize / wsize

- **Set to 1048576 (1 MiB)** — current Linux client cap per RPC. Verify with nfsstat -m and /proc/self/mountstats.
- Kernel 6.16 supports for 4M for the storage side.

Reliability & Timeouts

- **hard** *(default for v4)*: Required for training/checkpoints to avoid silent corruption.
- **timeo= / retrans=**: Use defaults for bulk; for latency-sensitive small-IO consider slightly lower timeo and verify behavior under loss.
- **retrans**: Don’t set too low; allow the client to ride out transient blips during epochs.

Caching & Coherency (metadata)

- **lookupcache=**:
  - all (aggressive): fastest for read-mostly, may delay visibility of new files created by others.
  - positive: good balance for dataloaders (cache hits for existing entries, fewer negatives).
  - none: strongest coherency; avoid unless required (metadata RPC storm).

- **Attribute cache:** acregmin/max, acdirmin/max, or coarse actimeo=<sec> to set all four.
- **Training/checkpoints (read-mostly):** longer timers (e.g., actimeo=600).
- **Dataloaders:** shorter timers (e.g., acregmax=60,acdirmax=60).
- **nocto**: disables close-to-open consistency; choose **only** on strictly read-only datasets staged once.