



Checkpoint I/O Sizing: Write Bandwidth for 70B-405B

Plan and build private AI compute with GPU Smith. We connect workload requirements with GPU hardware, networking, deployment and operating decisions so your infrastructure fits the work it must perform.

Published by GPU Smith · 2026-10-29 · llm checkpoint size 70b · how large is a checkpoint for training 405b model · checkpoint write bandwidth gpu cluster storage · async checkpointing pytorch dcp · zero optimizer state memory 16 bytes per parameter

Original article: <https://gpusmith.com/articles/en/checkpoint-io-sizing-parallel-file-systems>



In this article

1. The short answer
2. What a training checkpoint contains
3. Checkpoint sizes for 7B, 70B and 405B
4. Why each GPU writes a small shard
5. The stall budget: from checkpoint interval to GB/s
6. Asynchronous save changes which number you size
7. Sharded save, deduplication and resharding
8. Reducing the bytes
9. Parallel file system design principles
10. The restart path is a read problem
11. Worked example: 405B on 16,384 GPUs
12. Test plan and checklist

The short answer

Checkpoint write bandwidth depends on three numbers: bytes per checkpoint, how long training may block, and how often you save. With mixed-precision Adam a full training checkpoint is about 12 to 16 bytes per parameter: roughly 0.84 to 1.12 TB for 70B and 4.9 to 6.5 TB for 405B. Required bandwidth is size divided by allowed stall, so a synchronous 5.67 TB save inside 60 seconds needs about 95 GB/s sustained aggregate write. Asynchronous checkpointing blocks the GPUs only while state is copied to host memory, giving storage the whole interval to absorb the write. That turns a peak-bandwidth problem into a sustained-bandwidth and metadata problem.

Totals below are derived from per-parameter byte counts, not measured; check your own checkpoint with `du`.

What a training checkpoint contains

Per ZeRO, mixed-precision Adam training of Ψ parameters holds an fp16 copy of the parameters and gradients ($2\Psi + 2\Psi$ bytes), plus optimizer states of "an fp32 copy of the parameters, momentum and variance, with memory requirements of 4Ψ , 4Ψ , and 4Ψ bytes". With $K = 12$ that gives " $2\Psi + 2\Psi + K\Psi = 16\Psi$ bytes of memory requirement" (ZeRO, Rajbhandari et al.). Its example: a 1.5B-parameter GPT-2 needs at least 24 GB versus 3 GB for fp16 parameters alone. bf16 is also 2 bytes, so the accounting holds.

A checkpoint is not a dump of GPU memory:

Component	Bytes per parameter	Usually in a resumable checkpoint?
bf16/fp16 working weights	2	Often, though they can be rebuilt from the fp32 master copy
Gradients	2	Usually not; they are recomputed on the next step
fp32 master weights	4	Yes

Component	Bytes per parameter	Usually in a resumable checkpoint?
Adam first moment (momentum)	4	Yes
Adam second moment (variance)	4	Yes
Dataloader, RNG, scheduler state	negligible per parameter	Yes, but small

The planning range is **12 Ψ** (master weights plus moments), **14 Ψ** (plus bf16 weights, the central case here) and **16 Ψ** (ZeRO's full in-memory figure, gradients included). Activations are not checkpointed. For the in-memory side, see the [LLM training memory calculator](#).

Checkpoint sizes for 7B, 70B and 405B

These figures are derived arithmetic in decimal units (1 TB = 10^{12} bytes):

Model	Weights only, 2 Ψ (inference export)	12 Ψ	14 Ψ (central case)	16 Ψ
7B	14 GB	84 GB	98 GB	112 GB
70B	140 GB	840 GB	980 GB	1.12 TB
405B	810 GB	4.86 TB	5.67 TB	6.48 TB

Example: 405×10^9 parameters $\times$ 14 bytes = 5.67×10^{12} bytes = 5.67 TB.

A training checkpoint is 6 to 8 times the weights-only file you would ship to inference, so do not size training storage from model download sizes. Retention multiplies capacity: ten retained 405B checkpoints at 14 Ψ take about 57 TB before protection overhead. ByteCheckpoint's operators handle this with a cool-down mechanism that "migrates data from SSD to HDD storage" ([ByteCheckpoint](#), [Wan et al.](#)).

Why each GPU writes a small shard

The total is large, but under sharded training no single GPU writes much. Llama 3 405B was trained "on up to 16K H100 GPUs". The 16,384-GPU configuration used TP 8, PP 16 and DP 128, with FSDP that "shards optimizer states and gradients" ([Llama 3 Herd of Models](#)). Meta reports that "Checkpointing saves each GPU's model state, ranging from 1 MB to 4 GB per GPU" (same source).

An even split of 5.67 TB over 16,384 GPUs would be about 346 MB per GPU, or 2.8 GB per 8-GPU node. Meta's 1 MB to 4 GB range shows real layouts are uneven (embeddings, pipeline-stage imbalance, replicated tensors), and the largest shard sets a synchronous save's completion time. Size from the shard distribution, not the mean.

Sharding also sets file count. PyTorch Distributed Checkpoint (DCP) "produces multiple files per checkpoint, with at least one per rank", plus a global .metadata file ([PyTorch DCP docs](#), 2.15). At 16,384 ranks, each checkpoint is a burst of at least 16,384 file creates: a metadata event as much as a bandwidth event.

The stall budget: from checkpoint interval to GB/s

Step 1: pick the interval from the failure rate

In a 54-day snapshot of Llama 3 405B pre-training, Meta counted 466 job interruptions, of which "419 were unexpected" (Llama 3). That is one every $54 \times 24 / 419 = 3.09$ h, about 11,100 s.

Assume (first-order) a failure lands mid-interval on average, losing $T/2$ of work plus restart time:

Checkpoint interval T	Avg. work lost per interruption	Lost fraction of wall clock (excl. restart)
60 min	30 min	~16%
30 min	15 min	~8.1%
10 min	5 min	~2.7%
5 min	2.5 min	~1.3%

At this rate a 30-minute interval wastes about 8% of the cluster, which is why Meta aims "to minimize GPU pause time during checkpointing and increase checkpoint frequency" (Llama 3). Your rate will differ; the [cluster reliability budget](#) covers estimating it.

Step 2: turn interval and overhead target into a stall budget

A synchronous save blocking S seconds every T seconds costs S/T . For the 405B central case (5.67 TB):

Interval T	Overhead target	Allowed stall $S = T \times \text{target}$	Required aggregate write = 5.67 TB / S
30 min	2%	36 s	~158 GB/s
30 min	1%	18 s	315 GB/s
10 min	2%	12 s	~473 GB/s
10 min	1%	6 s	945 GB/s

At fixed stall budgets:

Model (14 Ψ)	10 s stall	60 s stall	5 min stall
70B (980 GB)	98 GB/s	~16.3 GB/s	~3.3 GB/s
405B (5.67 TB)	567 GB/s	94.5 GB/s	18.9 GB/s

These are sustained end-to-end rates at the client, including file creation, `fsync` and close, not a datasheet sequential-write number. For scale, Meta's Tectonic-based fabric for Llama 3 offered "240 PB of storage out of 7,500 servers equipped with SSDs" with "a sustainable throughput of 2 TB/s and a peak throughput of 7 TB/s", and Meta still calls "the highly bursty checkpoint writes that saturate the storage fabric for short durations" a major challenge (Llama 3). A synchronous design buys burst bandwidth that sits idle for 99% of the interval.

Asynchronous save changes which number you size

DCP's `async_save` "first de-stages the `state_dict` on to the staging storage (defaults to CPU memory), and then calls the save in a separate thread", and it returns a future ([PyTorch DCP docs, 2.15](#)). In PyTorch 2.15.0 (released 2026-10-28) the docs still mark it "experimental and subject to change", and it runs the save in a thread by default (`AsyncCheckpointType.THREAD`; a separate process is optional). The GPU-visible pause is the staging copy plus coordination; the storage write overlaps later training steps.

Published results:

- **PyTorch/IBM (PyTorch blog, November 2024):** 7B checkpoint downtime fell "from an average of 148.8 seconds to 6.3 seconds", and the GPU-to-CPU phase "can take from 6 – 14 seconds for 7B-13B model sizes" ([PyTorch blog](#)). Async "does not compress the actual serialization checkpointing time"; it moves it off the critical path. The team used a separate process group for checkpoint collectives to avoid hangs.
- **ByteCheckpoint (research paper, ByteDance production):** a "Text Transformer 405B" on 8,960 H800 GPUs (TP 8, DP 70, PP 16, Megatron-LM) showed 0.59 s blocking, 51.06 s end-to-end save and 129.49 s load ([ByteCheckpoint](#)). The paper does not give the checkpoint's byte size, so do not back out a bandwidth.

With async save, size against these constraints instead:

1. **Persist bandwidth $\geq$ size / interval, with headroom.** The save must finish before the next starts, or the job blocks or needs a second staging buffer. For 405B at 14 Ψ and a 10-minute interval: 5.67 TB / 600 s $\approx$ 9.45 GB/s minimum; plan 2 to 3 times that (about 20 to 30 GB/s) for contention.
2. **Host memory $\geq$ the node's shard per staging buffer.** About 2.8 GB per node for the even 405B split; about 7.7 GB per node for a 70B job on 1,024 GPUs (980 GB / 128 nodes). Small next to server DRAM, but a pinned-memory budget if pinned staging is on: "Pinned memory improves CPU-GPU transfer speeds but uses more memory" ([PyTorch DCP docs](#)).
3. **The write path stays busy most of the interval**, competing with dataloader reads. Test under concurrent read load.

Synchronous save at a 1% budget needs hundreds of GB/s for 405B; async needs tens of GB/s sustained plus metadata capacity for the file burst.

Sharded save, deduplication and resharding

Parallel per-rank writes. DCP "supports loading and saving models from multiple ranks in parallel" ([PyTorch DCP docs](#)). `FileSystemWriter` defaults to `thread_count=1`; raise it first when a node cannot fill its NIC.

Deduplicating replicas. Identical tensors on data-parallel replicas should be written once. Megatron Core's `FullyParallelSaveStrategyWrapper` exchanges only metadata and, "based on data replication on different ranks, we try to distribute the save as uniformly as possible" ([Megatron Core 0.19.2 API docs](#)). Spreading writers matters as much as removing duplicates; if one replica writes everything, a few NICs set the save time.

Load-time resharding. DCP "handles load-time resharding which enables saving in one cluster topology and loading into another" ([PyTorch DCP docs](#)). Restarts often run on fewer nodes, and a resharded load reads byte ranges from many files. ByteCheckpoint reports an average 54.20× reduction in checkpoint stalls versus DCP and Megatron MCP baselines, and up to 9.96× faster saves and 8.80× faster loads ([ByteCheckpoint](#)). These are the authors' comparisons on their HDFS backend, not independent results.

Reducing the bytes

- **Drop redundant state.** Skipping the bf16 copy moves 14Ψ to 12Ψ, about 14% fewer bytes, if your framework can resume from it.
- **Incremental and quantized checkpoints.** Check-N-Run reduced "the required write bandwidth by 6-17× and the required capacity by 2.5-8×" on Facebook models ([Check-N-Run](#), [Eisenman et al.](#)). Those are recommendation models, where only part of the embedding tables changes per interval; dense LLMs update every parameter each step, so do not assume similar gains for 70B to 405B transformers.

Parallel file system design principles

No WEKA, VAST, GPFS or DDN documentation was verified for this article, so this section stays at design-principle level, with the Lustre manual as the concrete reference. Validate any vendor throughput claim with the test plan below.

Aggregate write is a client-side sum. Multiply concurrent writer nodes by what each pushes through its storage NIC, then confirm the servers absorb that sum durably, not just into cache.

Match striping to file layout. Checkpoints are usually file-per-rank, hundreds of MB to a few GB each. Lustre's rule of thumb is to "stripe over as few objects as will meet those needs and no more"; wide striping is for when "the file size is very large and/or is accessed by many nodes at a time", and "small files do not benefit from striping" ([Lustre Operations Manual](#), [ch. 19](#)). Thousands of file-per-rank writers with low stripe counts spread across OSTs naturally. For single shared files, Lustre warns that several clients on one stripe "can lead to issues with lock exchange", which overstriping (Lustre 2.13+) addresses.

Metadata is a separate budget. One file per rank per checkpoint means tens of thousands of creates, `fsync` calls and closes in seconds, and retention deletes add the same again. ByteCheckpoint's team had to address a "QPS bottleneck of HDFS NameNode metadata requests" ([ByteCheckpoint](#)). Test create rates at full rank count, not just GB/s.

Separate the hot tier from retention. Keep recent checkpoints on flash near the cluster and age older ones to cheaper capacity. A node-local NVMe tier that drains to shared storage protects the stall budget, but a lost node takes its unsynced shard with it, so decide explicitly which copy is your recovery point. For the cloud cost side, see [AWS P5 storage cost](#).

The restart path is a read problem

After a failure every rank loads at once, and the cluster idles until the slowest rank has its state: 5.67 TB in the central 405B case, more if replicas read the same weights or resharded loads read overlapping ranges. ByteCheckpoint's 405B job took 129.49 s to load against 51.06 s to save ([ByteCheckpoint](#)). At Llama 3's interruption rate that load happens more than seven times a day, so qualify restart read bandwidth too. If restores go straight into GPU memory, the [GDSIO qualification runbook](#) covers the read path.

Worked example: 405B on 16,384 GPUs

Illustrative assumptions: dense 405B, bf16 with fp32 master and Adam, 14 Ψ , 2,048 nodes $\times$ 8 GPUs, Llama 3 interruption rate, checkpoint overhead $\leq$ 1%.

1. **Checkpoint size:** $405 \times 10^9 \times 14 \text{ B} = \mathbf{5.67 \text{ TB}}$. Even shard: $5.67 \text{ TB} / 16,384 \approx 346 \text{ MB}$ per GPU, about 2.8 GB per node.
2. **Interval:** a 10-minute interval holds expected lost work to about 2.7% of wall clock (300 s / 11,135 s).
3. **Synchronous option:** 1% of 600 s = 6 s, so $5.67 \text{ TB} / 6 \text{ s} = \mathbf{945 \text{ GB/s}}$ aggregate, about 462 MB/s per node ($945 / 2,048$). Per node that is easy; the aggregate burst plus about 16,000 file creates in the same seconds is not. Relaxing to a 60 s stall (10% overhead at this interval) still needs 94.5 GB/s.
4. **Asynchronous option:** blocking is the staging copy, published at sub-second (ByteCheckpoint, 405B) to 6 to 14 s (PyTorch blog, 7B to 13B); measure yours. Storage must persist 5.67 TB within 600 s: $\geq \mathbf{9.45 \text{ GB/s}}$, about **20 to 30 GB/s** with headroom. Host staging: about 2.8 GB per node per buffer.
5. **Restart read:** loading 5.67 TB in 2 minutes takes about 47 GB/s aggregate read.
6. **Capacity:** 10 retained checkpoints $\approx$ 57 TB on the hot tier, before protection overhead.

Async needs roughly 30 to 50 times less write bandwidth than 1%-overhead synchronous save; remaining risk is metadata and reads.

Test plan and checklist

Model the real pattern: thousands of writers, one file each, a burst, silence, then a mass read. IOR fits: -F makes each MPI process "perform I/O to a unique file", -b sets per-task block size, -t transfer size, -e will "perform fsync upon POSIX write close", and -C reorders tasks so readback avoids the client's own cache ([IOR options](#)).

1. Set -b to your measured per-rank shard size, including the largest shard.
2. Run one rank per GPU across as many nodes as you can spare with -F -e -w; record aggregate GB/s and the slowest rank's time.
3. Repeat while another job saturates reads, approximating async save under training.
4. Read back with -r -C for restart bandwidth.
5. Measure file create/close rates at full rank count separately.
6. On the real framework, time DCP or Megatron saves with async on and off, and check du against the 12 Ψ /14 Ψ /16 Ψ assumption.
7. Test a resharded restore onto fewer nodes, the restart you will actually perform after hardware loss.

Checklist: checkpoint bytes measured · interval derived from your failure rate · stall budget written down · sync vs async decided · persist bandwidth $\geq$ size / interval with headroom · metadata rate tested · restart read tested with resharding · retention tier sized.

GPU Smith helps teams work through this kind of storage sizing and its test plans when specifying training clusters.

Source links

Links cited in this article, in order of first appearance. Full addresses are provided for readers using extracted text.

1. <https://arxiv.org/pdf/1910.02054>
2. <https://arxiv.org/abs/2407.20143>
3. <https://arxiv.org/pdf/2407.21783>
4. <https://docs.pytorch.org/docs/2.15/distributed.checkpoint.html>
5. <https://pytorch.org/blog/reducing-checkpointing-times>
6. https://docs.nvidia.com/megatron-core/developer-guide/0.19.2/apidocs/core/core.dist_checkpointing_strategies.fully_parallel.html
7. <https://arxiv.org/abs/2010.08679>
8. https://doc.lustre.org/lustre_manual.xhtml
9. <https://ior.readthedocs.io/en/latest/userDoc/options.html>

THE PUBLISHER

About GPU Smith

GPU Smith provides independent engineering and research for private AI infrastructure. We help technical buyers and operators reason about GPU workload sizing, hardware selection, procurement, deployment and inference operations, from the compute system to the networking, power and cooling requirements around it.

Start with the workload

Useful infrastructure decisions begin with the models, data, throughput, latency and operating constraints a team actually has. GPU Smith helps connect those requirements with system design and deployment choices. The goal is a privately controlled AI environment whose capacity and operational demands are understood before a purchasing decision.

Hardware and supplier research

Our public reference library covers GPUs, complete systems and networking components. The server-vendor directory and manufacturing research help teams investigate suppliers, compare options and follow sources behind technical claims. We distinguish manufacturer specifications from measured benchmarks and advertised capabilities from tested configurations. Reference pages are research resources, not a live inventory listing or a binding equipment quote.

Deployment and operations

GPU Smith's engineering scope includes infrastructure integration, networking, power, cooling and the practical operation of inference workloads. Our published articles explain the tradeoffs behind deployment, procurement and ongoing operations so teams can ask better questions and document decisions.

Work with GPU Smith

Explore the [hardware library](#), [GPU references](#), [networking references](#) and [vendor research](#). [Contact GPU Smith](#) with your workloads and deployment constraints to discuss a project and confirm current scope, pricing and availability.

Inclusion of a manufacturer or product in our research does not imply a partnership, certification or endorsement.

Website: <https://gpusmith.com>

This article is educational. Verify consequential medical, legal, financial, regulatory and technical decisions with appropriate professionals and the cited primary sources. Company services are described separately from the article's research findings.