



CONFERENCE

—NORTH AMERICA 2026—

Keeping GPUs Busy

High-Speed Storage for PyTorch via fsspec

Speakers Intro



Ankita Luthra

Senior Software Engineer, Google

Ankita Luthra is a Software Developer at Google, focused on AI/ML storage infrastructure and scalable data pipelines. Her work with open-source tools like fsspec (gcsfs) and gcsfuse improves how frameworks such as PyTorch/JAX efficiently access data from Cloud Storage.



Trinadh Kotturu

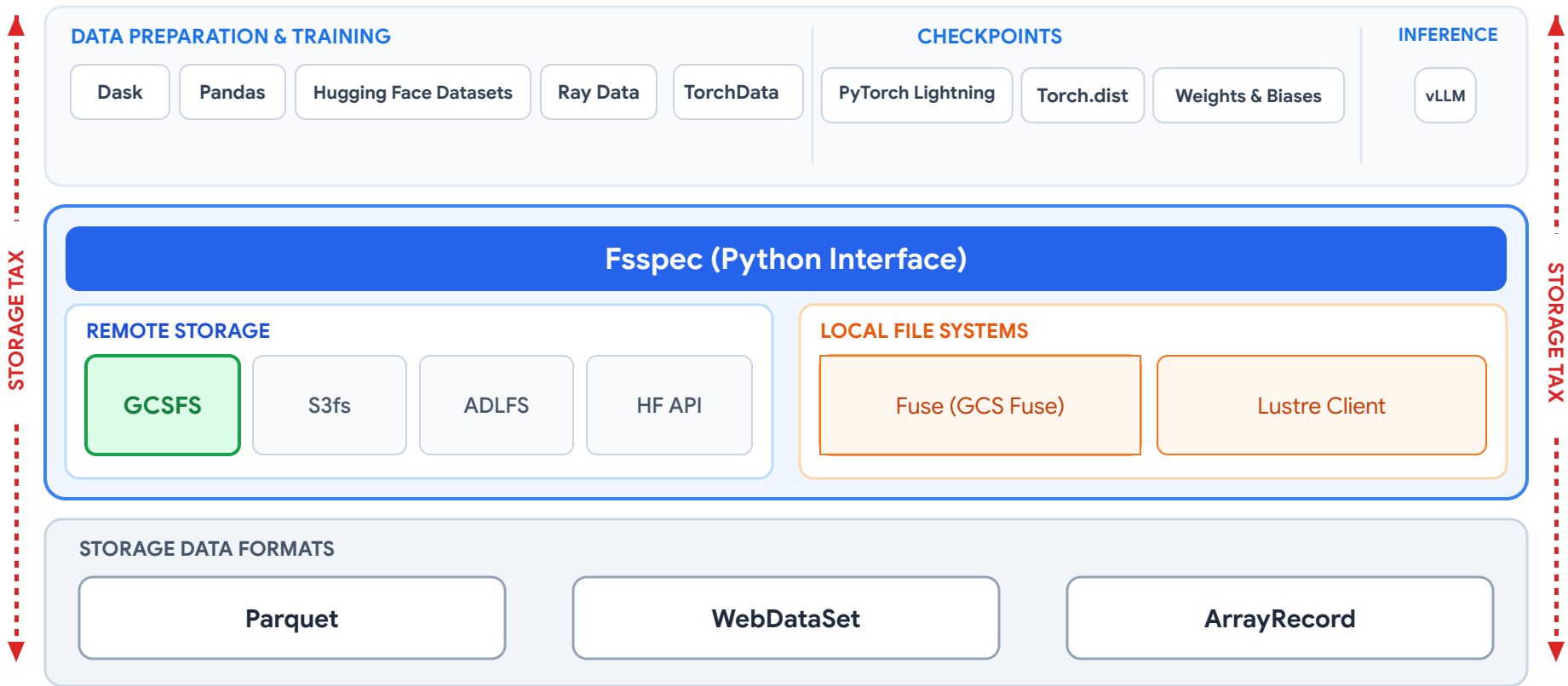
Senior Product Manager, Google

Trinadh Kotturu is a Senior Product Manager specializing in AI/ML and analytics client strategy at Google. An alumnus of IIM Bangalore with 12 years of experience, he has a proven track record of shipping v1 products and scaling them into robust platform services. His expertise spans large-scale distributed storage systems, autonomous driving, and system resiliency.



FSspec is pervasive in PyTorch

Industry standard Pythonic file interface across the AI development lifecycle



Optimising GCSFS

Getting past GIL contention and improving GCSFS

DATA PREPARATION & TRAINING

Dask

Pandas

Hugging Face Datasets

Ray Data

TorchData

CHECKPOINTS

PyTorch Lightning

Torch.dist

Weights & Biases

INFERENCE

vLLM

FSspec (Python Interface)

GCSFS

STORAGE DATA FORMATS

Parquet

WebDataSet

ArrayRecord

Rapid Storage

High performance object storage in a dedicated zonal bucket built on top of Google's colossus



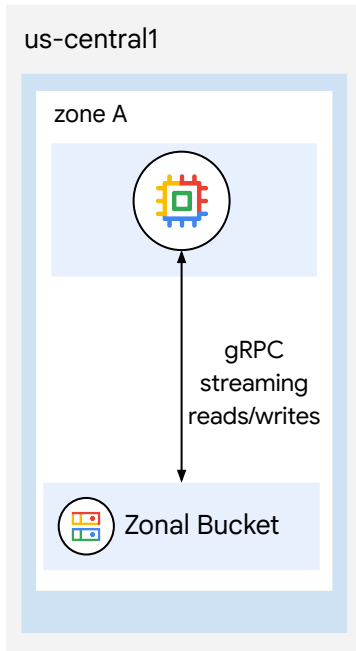
Colocated zonal bucket provides up to **15 TiB/s throughput** and **20M QPS**



<1 ms latency for random reads and append writes.



Optimized for AIML workloads with appendable objects + native Hierarchical Namespace



12x

faster sequential reads
(1MB IO)

9x

faster random reads
(64KB IO)

3x

faster writes
(16MB IO; single process)

[Microbenchmark results](#)
[Try it yourself](#)

Goal: Saturate the NIC

Leverage adaptive prefetcher to saturate the NIC with less processes and bypass GIL contention.

KEY CHALLENGE

When reading large files from cloud storage, GPUs can be often idle while waiting for data packets in between read calls due to network latency.

THE SOLUTION

Overlapped Background I/O

Dynamic Sizing

Uses actual I/O size provided by the application.

Linear Scaling

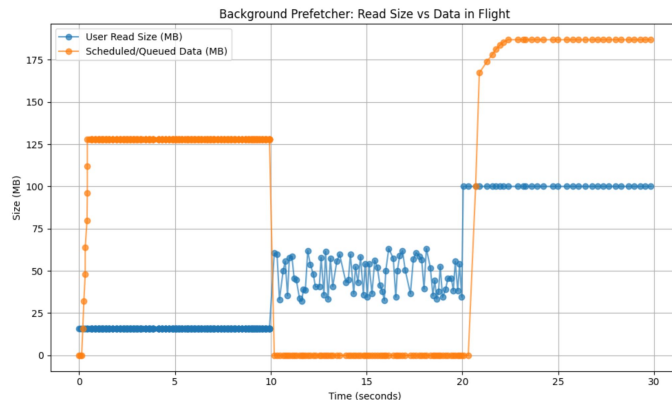
Prevents wasted calls and controls egress costs.

Concurrent Streams

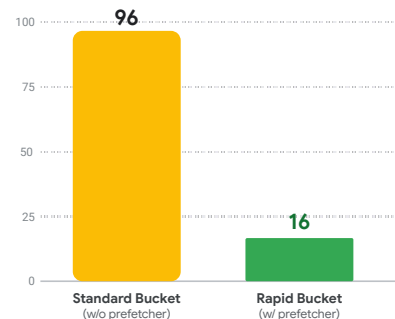
Parallel connections to maximize throughput.

Adaptive Behavior

Ramps for sequential; drops for random seeks.



Processes required to saturate the NIC



6x drop in process count to saturate the NIC

No-op migration to use best configuration

How to use Rapid Buckets in Pytorch workloads

Standard Buckets

```
# 1. Load Data (pseudo code)
ds = datasets.load_dataset("parquet",
    data_files=f"gs://my-standard-bucket/*.parquet",
    split=..., streaming=True)
train_loader = DataLoader(dataset,
    batch_size=..., collate_fn=...,
    workers=..., keep_workers_alive=True)

# 2. Configure Trainer
trainer = pl.Trainer(**trainer_config)

# 3. Train Model
trainer.fit(LlamaLitModel(model), train_loader,
    ckpt_path="gs://my-standard-bucket/ckpts")
```

Rapid Buckets + Prefetcher

```
# 1. Load Data (pseudo code)
ds = datasets.load_dataset("parquet",
    data_files=f"gs://my-rapid-bucket/*.parquet",
    split=..., streaming=True)
train_loader = DataLoader(dataset,
    batch_size=..., collate_fn=...,
    workers=..., keep_workers_alive=True)

# 2. Configure Trainer
trainer = pl.Trainer(**trainer_config)

# 3. Train Model
trainer.fit(LlamaLitModel(model), train_loader,
    ckpt_path="gs://my-rapid-bucket/checkpoints")
```

No-op Migration

Optimizing data formats

Recommendations for best storage performance

DATA PREPARATION & TRAINING

Dask

Pandas

Hugging Face Datasets

Ray Data

TorchData

CHECKPOINTS

PyTorch Lightning

Torch.dist

Weights & Biases

INFERENCE

vLLM

Fsspec (Python Interface)



GCSFS

STORAGE DATA FORMATS

Parquet

WebDataSet

ArrayRecord

WebDataset recommendations

Boost your data access for WebDataset

RAPID BUCKETS ADVANTAGE

Rapid buckets provides up to 2.5x throughput increase out-of-box without requiring architectural changes.

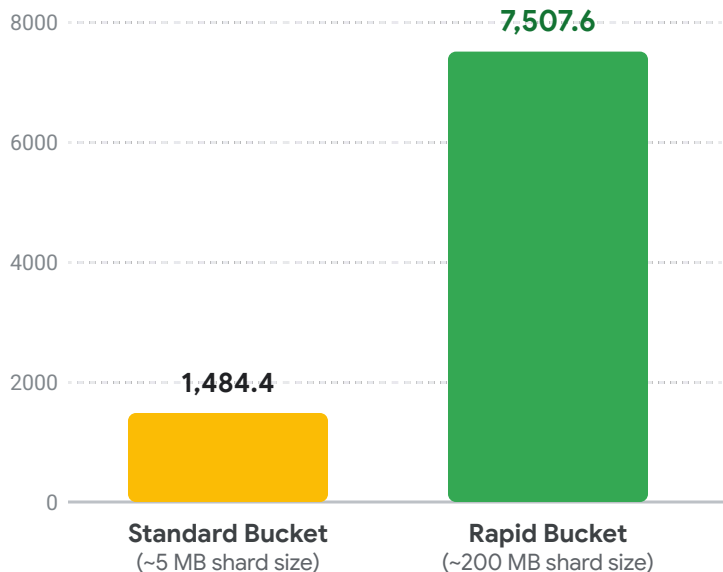
OPTIMAL CONFIGURATION PAIRING

Client-side buffering of 8-32MB or whole_object read mode to boost performance up to 4x.

OPTIMUM SHARD SIZE PAIRING

Moving from small shards to large shards increases throughput up to 3.8x and reduces network calls by ~92%.

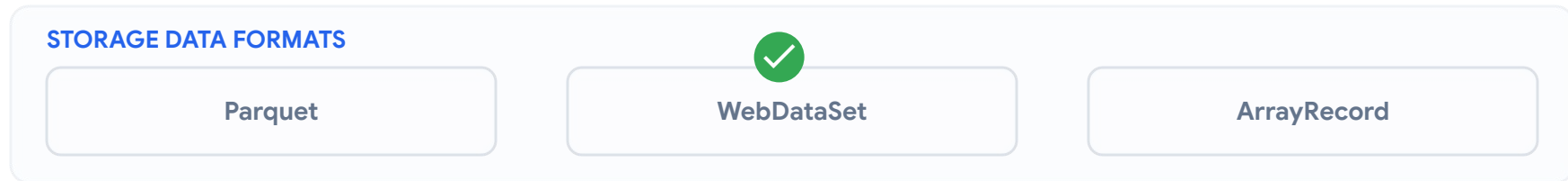
Sample Throughput (Images/s)



**+5x Samples/Read
Throughput Gain**

Optimising frameworks

Adjusting frameworks for high performance storage



Hugging Face Datasets optimisation

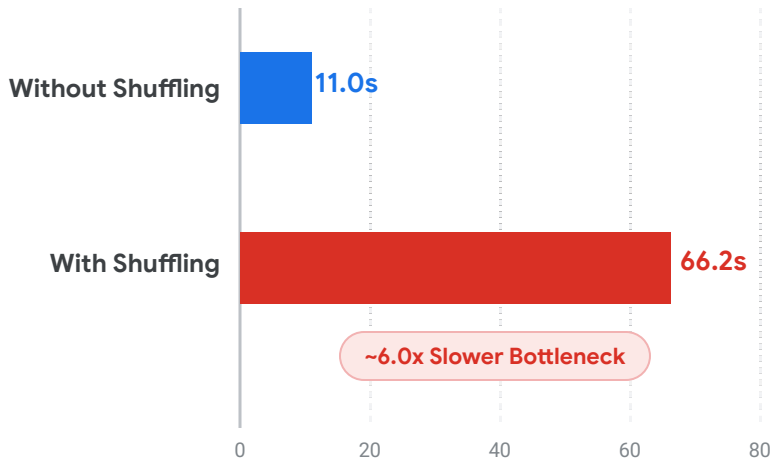
Throughput stalls identified during dataset shuffling and streaming checkpointing

SHUFFLING BOTTLENECK

6.0x execution time slowdown

- **Severe Throughput Penalty:** Enabling dataset shuffling increased execution time from 11.0s to 66.2s (~6.0x slower) compared to sequential reading.
- **GIL Contention Stalls:** Scaling worker threads yielded little to no throughput gains, as thread contention around the Python GIL prevented parallel execution.

Execution Time (seconds) — Lower is better



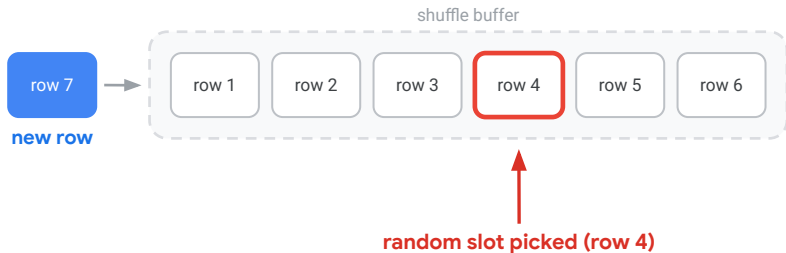
Hugging Face Datasets optimisation

Vectorized PyArrow C++ Shuffling & Dynamic Chunk Interleaving

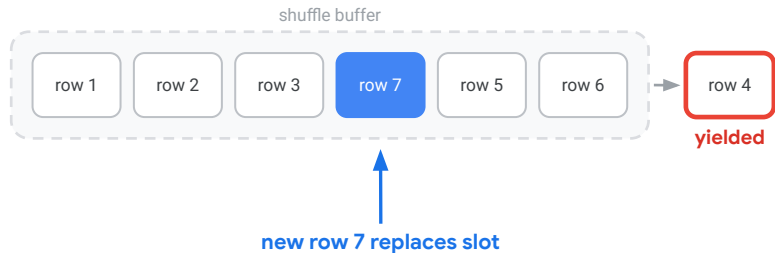
Bottleneck: one row at a time

- Every Arrow table was cut into **single rows** and the shuffle buffer swapped them **one by one in a Python loop**.
- Slow **DeepCopy** for **state_dict** executed on **each individual row**, stalling pipeline throughput.

1. New row arrives: random slot picked



2. Picked row is yielded: new row replaces it

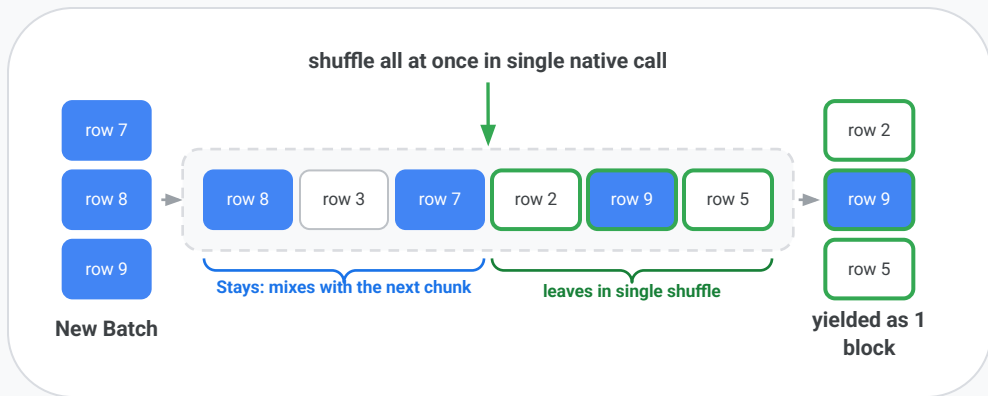


Hugging Face Datasets optimisation

Optimised Shuffling & Dynamic Chunk Interleaving

Performance optimisation

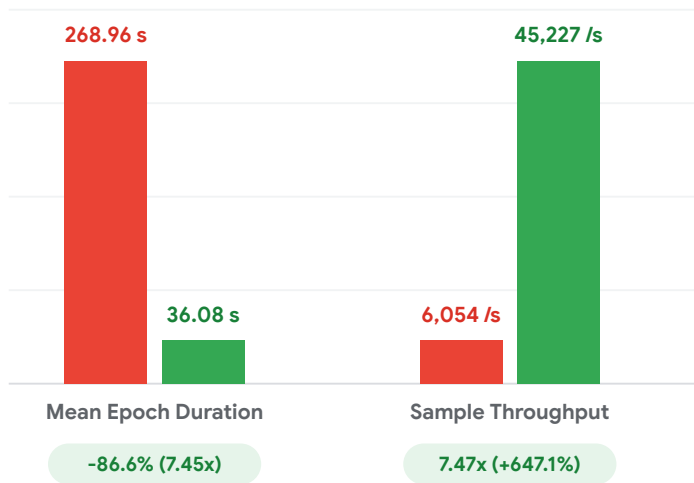
- Dynamic chunking (`buffer_size // max_buffer_input_shards`) avoids 1-row slicing and eliminates CPU overhead.
- Replaced expensive `deepcopy` with a lightweight custom `_fast_copy()` helper that directly clones acyclic lists/dicts and skips immutable primitives upfront.



Shuffled Streaming Benchmark (20 Shards, nw=4)

Epoch duration (lower is better) & throughput (higher is better)

Before After



Hugging Face Datasets Optimisations

**Quentin Lhoest** 🤖
@lhoestq

Congrats to the Google Cloud Storage team for the contribution !

Crazy speed improvement in `datasets` when streaming a dataset using approximate shuffling

Arrow batches are buffered, concatenated, and then rows are shuffled all using optimized Arrow C++ operations now

Code ▾

Optimize PyArrow block shuffling with dynamic rebatched chunk interleaving #8407

Merged

lhoestq merged 7 commits into `huggingface:main` from `yuxin00j:optimize-...` 

Ray Data optimisation

Boost your data access for Ray Data

PYTHON GCSFS WITH RAPID BUCKETS

Migrating datasets to Rapid buckets delivers upto 2.5x throughput boost out-of-the-box via DirectPath gRPC.

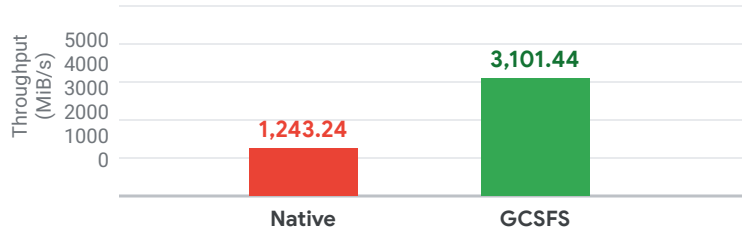
ADAPTIVE PREFETCHING FOR LARGE FILES

With multi-row-group large Parquet files, GCSFS Adaptive Prefetching downloads with CPU decoding parallelly shows upto 4x speedup over unbuffered reads.

NEAR-LINEAR WORKER SCALING

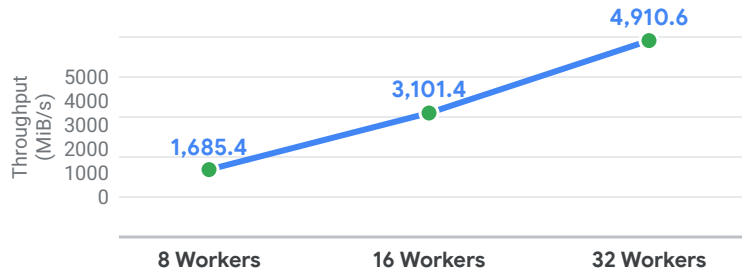
GCSFS lock-free asyncio architecture scales linearly with Ray workers - delivering a 2.91x throughput gain from 8 to 32 workers.

Rapid Throughput (16W, 256 Parquet files)



+2.5x Throughput Gain

Near Linear Worker Scaling (256 Parquet Files)



Near-Linear Throughput Scaling (2.91x Gain)

PyTorch Lightning Optimisation

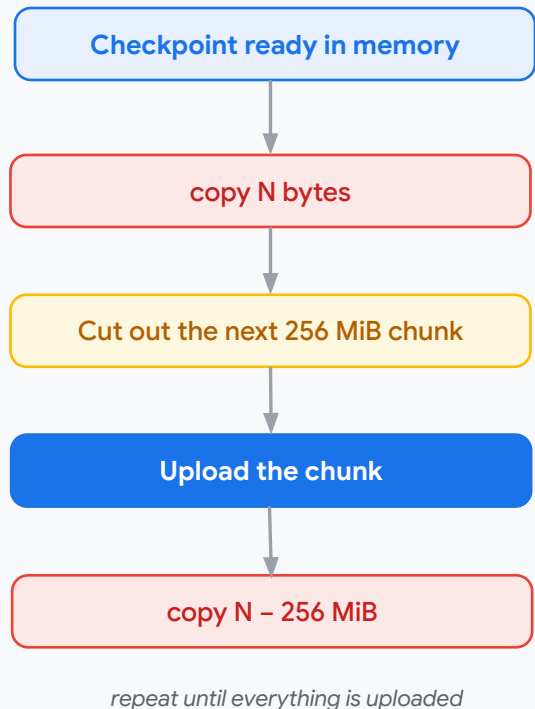
Improving performance for Pytorch Lightning

Problem Statement

Saving checkpoint using PyTorch Lightning's `_atomic` was severely bottlenecked at ~20 MB/s stalling GPU training. A single 48 GB checkpoint was taking nearly 40 minutes.

Root Cause Analysis

Redundant in memory copies across multiple upload chunks causes performance degradation. Plus, degradation linearly increases with increase in checkpoint sizes.



PyTorch Lightning Optimisation

Resolving the memory copying bottleneck in checkpoint saving



The Fix

Use `fs.pipe()` during checkpoint save



The checkpoint buffer is sliced with memoryview and sent as-is.



Bytes stream straight into one resumable upload.



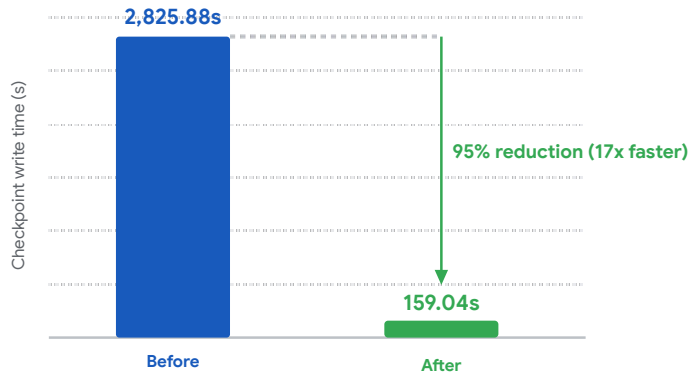
Copy cost is $O(N)$ instead of $O(N^2)$, so large checkpoints no longer stall the GPU.



`_atomic_save` internally calls `fs.pipe()`, no changes required in public interface.

Checkpoint Write Time (48 GB Checkpoint)

Execution time in seconds (available starting version 2.6.5)



Latency dropped from **~47 minutes** to **~2.6 minutes**
`fs.pipe()` + Rapid Buckets.

For more Details: <https://lightning.ai/blog/ptl-checkpoints-gcs-rapid-buckets>

Optimising frameworks

Adjusting frameworks for high performance storage



DATA PREPARATION & TRAINING

Dask

Pandas

Hugging Face Datasets

Ray Data

TorchTitan

PyTorch Lightning

Torch.dist

Weights & Biases

INFERENCE

vLLM

Fsspec (Python Interface)



GCSFS

STORAGE DATA FORMATS

Parquet



WebDataSet

ArrayRecord

What's next: Agentic Systems & TorchTPU

What agents and accelerators need from storage

AGENTIC FRAMEWORKS

What Agentic Systems Need

- **Artifact-centric:** Agents read & write files, tool outputs, checkpoints
- **Low latency:** Tool calls can't wait on mounts or cold starts
- **Framework-native:** Standard interfaces
- **GCSFS preview:** fsspec-based artifact access for ADK with Rapid support out-of-box.

PREVIEW: GCSFS + ADK

PYTORCH ON TPU

Hardware Optionality

- **Accelerator choice:** Same PyTorch code on GPU or TPU
- **Storage-agnostic:** Data & checkpoints stream from GCS to TPUs.
- **GCSFS preview:** GCSFS backed GCS access on TPU.

PREVIEW: GCSFS ON TPU

Thank You!

GET STARTED

Try PyTorch with GCS Rapid Buckets!

Boost your GPU/TPU efficiency & accelerate dataset throughput.