TorchFS: A machine learning oriented file-system

Daniel Pereira
University of Minho
pg55928@alunos.uminho.pt

Duarte Ribeiro
University of Minho
pg6969@alunos.uminho.pt

Filipe Pereira
University of Minho
pg55941@alunos.uminho.pt

Abstract

As deep learning and machine learning advance, the need for efficient data storage and retrieval has become crucial due to the exponential growth of training datasets. We introduce TorchFS, a distributed file system tailored for storing and retrieving data used in training deep learning models. Designed for integration with the PyTorch framework, TorchFS prioritizes performance and resilience to meet the demands of modern AI workflows.

Keywords: typst, acm

1. Introduction

With the rapid evolution of deep learning (DL) and machine learning (ML), the demand for a filesystem that can handle the intense data ingestion requirements of ML training pipelines has become paramount. As the volume of training data continues to grow exponentially, efficient data storage and retrieval are crucial. Traditional distributed filesystems like Ceph, HDFS, and Lustre [1], [2], [3] offer reliable throughput and durability but are not optimized for the predominantly sequential, read-heavy access patterns typical of ML data workflows.

In this paper, we introduce TorchFS, a distributed filesystem that addresses this gap by providing a filesystem designed specifically for ML workloads. TorchFS plugs directly into the PyTorch ecosystem, exposing a familiar POSIX-style interface while retooling the backend for training-centric performance and resilience.

Our design centers on three key ideas:

1. **Epoch aware caching & pre-fetch** — utilizes training-step hints to keep frequently accessed training data on the host device, reducing latency and minimizing data transfer overhead.
2. **Horizontal scalability** — object-striped data servers that add bandwidth and capacity linearly as you scale out.
3. **Metadata/data separation** — decoupling namespace operations from bulk I/O (inspired by GFS, GPFS and BeeGFS [4], [5], [6]), so lookups never contend with tensor streaming.

This design ensures that TorchFS can handle the unique demands of DL workloads, providing a robust and efficient solution for data storage and retrieval.

2. Background and Challenges

We aimed to develop a filesystem that was optimized for the specific needs of ML workloads, particularly for use with the PyTorch framework. Therefore, we needed to consider the unique characteristics of these workloads and how they differ from traditional file systems.

Traditional file systems are designed to handle a wide variety of workloads, including random access patterns, small file sizes, and a mix of read and write operations. However, ML workloads often involve large datasets that are accessed in a more predictable manner, with the bulk of the load coming from reading training files. This means we can tailor our filesystem to better suit these workloads using optimizations.

2.1. PyTorch Access Patterns

To know how to optimize our filesystem to suit PyTorch workloads, we first needed to understand how PyTorch accesses data.

Therefore, we have to first profile the PyTorch data access patterns.

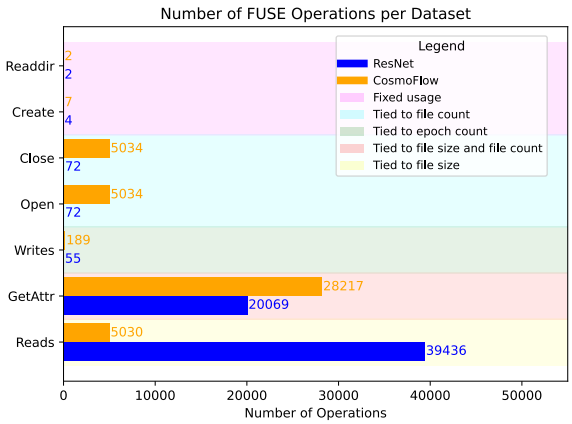
We profiled the data acess patterns of PyTorch using a passthrough implementation of FUSE that logged the accesses made to files during training. We evaluated the access patterns with two different datasets: resnet and cosmoflow, with 1 and 4 GPUs each. Resnet is a composed of a few large files (80MB each), while cosmoflow is composed of many small files (64KB each). These four configurations let us evaluate the access patterns of PyTorch in different scenarios.

After evaluation, we found the PyTorch access pattern to be as follows:

1. Epoch Starts
2. The GPU opens a few files, and reads a few chunks[1] of data from each file immediately (presumably as a pre-fetch cache)
3. The GPUs then start reading from the first file sequentially, in a first-come-first-serve manner, until the end of the file is reached.
   3.1 When the file is a chunk long, each GPU uses a whole file by themselves
4. When a file is fully read, it is closed. A new one (that is after the first few that were opened at the start) is then opened and its first chunks read, and the next file in the order is fully read.
5. When all files are fully read, the epoch ends.
6. After each epoch, a small number of files are opened and read sequentially, presumably for validation purposes.
7. A new epoch starts and the process repeats.

Below is a bar chart that plots the operation count for each dataset. We discovered that the acess pattern varies accross operations:

- `Readdir` and `Create` are used a fixed (and small) ammount of times
- `Open` and `Close` are proportional to the number of files in the dataset (and therefore the number of epochs)
- `Write` is tied mostly to epoch count, but also to the number of files
- `Getattr` is tied both to the file size and file count, as it was called for every open, close and also every two reads
- `Read` is tied to the file size (or count if the file is smaller than a chunk)



2.1.1. Insights

This behaviour brings us some insights into how we can optimize our filesystem to better suit PyTorch workloads:

1. The access pattern is predictable, with a few files being opened and read sequentially, and the next file in the order is read.
   - We can accurately pre-fetch files and chunks into a in-memory cache before they are neeeded, reducing latency and increasing GPU usage.
2. After each epoch, the first few files are opened and read sequentially.
   - We can prefetch these files into the same cache, again reducing latency.

3. The TorchFS design

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magnam aliquam quaerat voluptatem. Ut enim aeque doleamus animo, cum corpore dolemus, fieri tamen permagna accessio potest, si aliquod aeternum et infinitum impendere malum nobis opinemur. Quod idem licet transferre in voluptatem, ut postea variari voluptas distinguique possit, augeri amplificarique non possit. At etiam Athenis, ut e patre audiebam facete et urbane Stoicos irridente, statua est in quo a nobis

[1] Chunks are about 130KB in size

philosophia defensa et collaudata est, cum id, quod maxime placeat, facere possimus, omnis voluptas assumenda est, omnis dolor repellendus. Temporibus autem quibusdam et.

4. Implementation

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magnam aliquam quaerat voluptatem. Ut enim aeque doleamus animo, cum corpore dolemus, fieri tamen permagna accessio potest, si aliquod aeternum et infinitum impendere malum nobis opinemur. Quod idem licet transferre in voluptatem, ut postea variari voluptas distinguique possit, augeri amplificarique non possit. At etiam Athenis, ut e patre audiebam facete et urbane Stoicos irridente, statua est in quo a nobis philosophia defensa et collaudata est, cum id, quod maxime placeat, facere possimus, omnis voluptas assumenda est, omnis dolor repellendus. Temporibus autem quibusdam et.

5. Conclusion

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magnam aliquam quaerat voluptatem. Ut enim aeque doleamus animo, cum corpore dolemus, fieri tamen permagna accessio potest, si aliquod aeternum et infinitum impendere malum nobis opinemur. Quod idem licet transferre in voluptatem, ut postea variari voluptas distinguique possit, augeri amplificarique non possit. At etiam Athenis, ut e patre audiebam facete et urbane Stoicos irridente, statua est in quo a nobis philosophia defensa et collaudata est, cum id, quod maxime placeat, facere possimus, omnis voluptas assumenda est, omnis dolor repellendus. Temporibus autem quibusdam et.

Bibliography

[1] S. A. Weil, S. A. Brandt, E. L. Miller, D. D. E. Long, and C. Maltzahn, "Ceph: A Scalable, High-Performance Distributed File System," in *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation (OSDI '06)*, Seattle, WA, USA, Nov. 2006, pp. 307–320. [Online]. Available: https://ceph.io/assets/pdfs/weil-ceph-osdi06.pdf

[2] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, "The Hadoop Distributed File System," in *Proceedings of the 26th IEEE Symposium on Mass Storage Systems and Technologies (MSST '10)*, Lake Tahoe, NV, USA, May 2010, pp. 1–10. doi: 10.1109/MSST.2010.5496972.

[3] P. Schwan, "Lustre: Building a File System for 1,000-node Clusters," in *Proceedings of the Ottawa Linux Symposium*, Ottawa, Ontario, Canada, Jul. 2003, pp. 380–386. [Online]. Available: https://www.kernel.org/doc/ols/2003/ols2003-pages-380-386.pdf

[4] S. Ghemawat, H. Gobioff, and S.-T. Leung, "The Google File System," in *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP '03)*, Bolton Landing, NY, USA, Oct. 2003, pp. 29–43. doi: 10.1145/945445.945450.

[5] F. Schmuck and R. Haskin, "GPFS: A Shared-Disk File System for Large Computing Clusters," in *Proceedings of the 1st USENIX Conference on File and Storage Technologies (FAST '02)*, Monterey, CA, USA, Jan. 2002, pp. 231–244. [Online]. Available: https://www.usenix.org/events/fast02/full_papers/schmuck/schmuck.pdf

[6] J. Heichler, "An Introduction to BeeGFS," Kaiserslautern, Germany, Nov. 2014. [Online]. Available: https://www.beegfs.de/docs/whitepapers/Introduction_to_BeeGFS_by_ThinkParQ.pdf