Large Scale Caching and Streaming of Training Data for Online Deep Learning

Jie Liu
jliu279@ucmerced.edu
University of California, Merced
Merced, USA

Bogdan Nicolae
bnicolae@anl.gov
Argonne National Laboratory
Chicago, USA

Dong Li
dli35@ucmerced.edu
University of California, Merced
Mercced, USA

Justin M. Wozniak
woz@anl.gov
Argonne National Laboratory
Chicago, USA

Tekin Bicer
tbicer@anl.gov
Argonne National Laboratory
Chicago, USA

Zhengchun Liu
zhengchun.liu@anl.gov
Argonne National Laboratory
Chicago, USA

Ian Foster
foster@anl.gov
Argonne National Laboratory
Chicago, USA

ABSTRACT

The training of deep neural network models on large data remains a difficult problem, despite progress towards scalable techniques. In particular, there is a mismatch between the random but predetermined order in which AI flows select training samples and the streaming I/O patterns for which traditional HPC data storage (e.g., parallel file systems) are designed. In addition, as more data are obtained, it is feasible neither simply to train learning models incrementally, due to catastrophic forgetting (i.e., bias towards new samples), nor to train frequently from scratch, due to prohibitive time and/or resource constraints. In this paper, we study data management techniques that combine caching and streaming with rehearsal support in order to enable efficient access to training samples in both offline training and continual learning. We revisit state-of-art streaming approaches based on data pipelines that transparently handle prefetching, caching, shuffling, and data augmentation, and discuss the challenges and opportunities that arise when combining these methods with data-parallel training techniques. We also report on preliminary experiments that evaluate the I/O overheads involved in accessing the training samples from a parallel file system (PFS) under several concurrency scenarios, highlighting the impact of the PFS on the design of the data pipelines.

CCS CONCEPTS

• **Computer systems organization** → **Caching systems**; *Streaming data*; Training data; • **Deep learning** → Online deep learning.

KEYWORDS

deep learning, distributed caching, data pipelines, reuse or training data

FlexScience '22, July 1, 2022, Minneapolis, MN, USA

ACM ISBN 978-1-4503-9309-6/22/07.
https://doi.org/10.1145/3526058.3535453

ACM Reference Format:
Jie Liu, Bogdan Nicolae, Dong Li, Justin M. Wozniak, Tekin Bicer, Zhengchun Liu, and Ian Foster. 2022. Large Scale Caching and Streaming of Training Data for Online Deep Learning. In *Proceedings of the 12th Workshop on AI and Scientific Computing at Scale using Flexible Computing Infrastructures (FlexScience '22), July 1, 2022, Minneapolis, MN, USA.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3526058.3535453

1 INTRODUCTION

Deep learning (DL) applications are rapidly gaining traction both in industry and scientific computing, driven by the accumulation of massive data. In science, for example, instruments that collect data at GB/s and 100+ TB/day present a wide range of learning opportunities. We thus see significant interest in deploying DL on high-performance computing (HPC) systems in order to enable rapid learning in various scientific areas, such as fusion energy science, computational fluid dynamics, lattice quantum chromodynamics, virtual drug response prediction, and cancer research.

Various approaches for training DL models on massive data have been proposed: coarse-grain parallelization on multiple nodes using data-parallel [5], model-parallel [3], pipeline-parallel [12], and hybrid techniques; fine-grain parallelization on many-core architectures by constructing and scheduling execution graphs at the tensor level; and low-level optimizations of operators [7] and communication primitives [2]. Most such work is targeted at alleviating the computational overhead needed to perform the forward and backward passes in DL training, as well as the communication costs associated with synchronizing subtasks across devices and nodes.

However, as computation and communication become highly optimized, another bottleneck begins to emerge that has seen comparatively less attention: the I/O operations needed to read training data and feed them to the computational pipeline. This bottleneck is particularly prominent at scale in large data centers and supercomputers that typically feature a large number of compute nodes connected to a storage repository of relatively limited I/O bandwidth (e.g., a parallel file system). Indeed, the limited I/O bandwidth of parallel file systems is known to cause bottlenecks in general [19], especially under concurrent access.

This I/O bottleneck is further exacerbated in the case of DL applications by a mismatch between the preferred access pattern of parallel file systems (large non-overlapping reads from a few files)

and the access pattern of DL training (small, pseudo-random reads from many files). In fact, popular reference datasets such as ImageNet [25], used to evaluate image classification DL models, feature millions of small images, each of size 100 KB or less. Retrieving these files is expensive because each file access incurs a double latency cost, both at the level of metadata (to obtain the location of a dataset) and data (I/O at offset within dataset). Unfortunately, parallel file systems are primarily optimized to deliver high throughput at the cost of high latency, which explains the mismatch regarding the preferred access pattern.

Once extracted from a remote repository, the raw input data is only the beginning of a complex data ingestion pipeline that involves decoding of training samples as tensors, augmentations (e.g., stretching or shifting the color spectrum of images), pseudo-random shuffling, grouping of samples into batches, etc. The data ingestion pipeline is typically implemented in asynchronous fashion though producer-consumer buffers that involve caching at each stage. Using this approach, each DL training iteration overlaps with the data ingestion, thereby hiding the overhead of I/O operations and successive transformations in the pipeline, which reduces the overall runtime.

In addition to data ingestion pipelines, there is an increasing for other data abstractions. For example, *continual learning* [16] involves updating a DL model in near-real time, by linking the data ingestion pipeline directly to a data stream rather than a storage repository. However, it is not possible to simply incrementally train the DL model with the new data, because this would lead to *catastrophic forgetting*: a bias of the DL models towards the most recent training samples at the expense of older ones, which effectively causes them to reinforce new patterns and forget old ones. A similar issue is also exhibited by *reinforcement learning* (RL) [8] approaches, because the samples are generated by sequential exploration of the states of an environment, which may lead to an excessive reinforcement of recent states at the expense of older ones. A common approach to address the problem of catastrophic forgetting is *rehearsal*, i.e., mixing new training samples with previously encountered, representative training samples in order to enable incremental training without bias. Thus, there is a need to transparently store, retrieve and mix representative training samples with regular training samples in the data ingestion pipeline.

In this context, there is a need for flexible data runtimes that are capable of addressing a variety of data ingestion patterns (irregular fine-grain I/O from storage repositories, direct streaming from data sources, rehearsal, augmentations and other post-processing). In addition, since the data ingestion pipeline is asynchronous, flexibility is also needed with respect to resource allocation, such as to dynamically respond to changes in the requirements of computational and I/O resources. This applies both to the interference between the data ingestion pipeline and the training, as well as to the interference between the different stages of the data ingestion pipeline itself. In this paper, we study the challenges, trade-offs and opportunities in the design of flexible data runtimes for both offline and continual DL model training We summarize our contributions as follows:

- We revisit state-of-art streaming techniques for feeding training samples to DL model training pipelines. We focus in particular on the *data pipeline* abstraction used by *TensorFlow* to enable transparent handling of prefetching, caching, shuffling, and data augmentation. These methods may have significant overheads, and thus data pipelines aim to overlap their execution with DL training as much as possible. However, this overlapping generates contention for resources (computational units, memory, network bandwidth) that introduces non-trivial trade-offs (Section 3).
- We discuss several considerations that arise when data pipelines are used at scale in conjunction with data-parallel and/or ensemble training techniques. In this context, the data pipelines constructed on each compute node are not independent of each other, but rather share input data, intermediate data, and/or resources (e.g., the data repository). This observation leads us to identify several challenges and opportunities to leverage synergies between the independent data pipelines for both offline training and continual learning; in the latter case, we advocate for extensions to facilitate transparent rehearsal (Sections 4).
- We evaluate the I/O overhead of serving training samples to DL models in several configurations (directly from a parallel file system or cached on local storage) using TensorFlow data pipelines for a representative DL application. Based on these preliminary results, we discuss the observed bottlenecks and their impact on the identified opportunities (Section 5).

2 RELATED WORK

The challenging nature of the pseudo-random I/O access patterns generated by DL training on small training samples is acknowledged by several studies [1, 11]. One approach to addressing this problem is to reorganize the training data. For example, the lightweight Lightning Memory-Mapped Database (LMDB) maps content directly into memory (thus taking advantage of OS-level I/O optimizations) and uses B+-trees to index it (thus reducing metadata overheads). However, Pumma et al. [17] have shown that this solution does not mitigate the problem sufficiently, as I/O overheads still dominate training (up to 90%) even for only a small degree of parallelism. Other approaches such as FanStore [26] provide a global cache layer on node-local burst buffers in a compressed format, allowing POSIX-compliant file access to the compressed data in user space. Further optimizations explore prefetching with perfect knowledge of future I/O based on fixing the seeds of pseudo-random number generators [4]. Such approaches are limited in their applicability to accelerating low-level I/O operations only.

Overlapping computations with I/O operations by prefetching and caching training samples in the background is another popular direction [14, 13]. However, generic I/O optimizations are not enough for this purpose, because specific operations such as shuffling the training samples with various ordering guarantees and/or complex transformations to augment the data are also needed before the data can be passed to training pipelines. To this end, abstractions such as data loader [24], data pipelines [11] and DALI [15] are becoming increasingly popular for both PyTorch and TensorFlow ecosystems. However, current implementations of such approaches provide only limited support for multi-node parallelism.

Several approaches deal with catastrophic forgetting [16]. Inspired by cognitive and neural science theories, rehearsal methods

date back three decades [20]. The most straightforward approach is to augment each minibatch with previously streamed training samples, called *exemplars*, that were selected randomly to be persisted to the storage repository for the purpose of rehearsal. To improve I/O performance, classification models such as iCARL [18] employ a limited memory buffer that stores a fixed number of exemplars for each class. However, this approach may lead to a decrease of accuracy when the number of classes increases, because exemplars from each class need to be dropped in order to make them all fit into the memory buffer. Approaches such as Naive Incremental Learning (NIL) [10] simply use a random replacement policy without considering classes. An alternative to storing exemplars is to train a generative adversarial network (GAN) from the stream in order to produce fake exemplars on demand that capture the patterns needed for rehearsal [9]. However, such approaches suffer from high overheads to train and keep the GAN up to date. Other approaches are based on regularization (i.e., estimating the relevance of the DNN parameters and penalizing those that show significant change from one task to another) or architectural changes that grow subnetworks dynamically [21]. However, rehearsal remains one of the most widely used techniques for mitigating catastrophic forgetting, and thus the problem of caching and storage of exemplars remains important.

In summary, state-of-art approaches are not sufficiently optimized to stream efficiently from a parallel file system or directly from an external data source into the DL training pipelines at scale on multiple nodes, nor do they offer support for rehearsal. To the best of our knowledge, we are the first to consider this combined problem in a distributed setting.

3 DATA PIPELINES

Modern DL runtimes such as TensorFlow are beginning to acknowledge the importance of optimizing the entire input data lifecycle, from reading the training samples all the way to feeding them to the training pipeline. To this end, abstractions such as *data pipelines* [11] are becoming increasingly popular. In this section, we revisit data pipelines, which will form the basis of our study.

In a nutshell, data pipelines abstract input data as a potentially infinite sequence of elements that can be either tensors or composite types (tuples, nested datasets, etc.). The elements can be accessed by means of an iterator, which offers sequential access through a simple API call: `get_next()`. The state of an iterator can be checkpointed to a file by using `save()` and `restore()`.

Since most DL training algorithms rely on mini-batch SGD (stochastic gradient descent), training samples are not accessed individually, but rather in groups called mini-batches that are returned by `get_next()`. The path from reading the input data all the way to generating the mini-batches creates a complex producer-consumer pipeline, as illustrated in Figure 1.

The entry point in the producer-consumer pipeline is a flexible *record enumeration abstraction* that is responsible for exposing new elements in a metadata queue. For example, in the case of a POSIX filesystem, the metadata queue will hold file names. These metadata are consumed by the *readers*, who are responsible for fetching the content of the elements. A framework like TensorFlow provides multiple types of readers, to support various data sources. The readers can optionally pass the content to a *pre-processing task* responsible for data augmentation, which can be defined by the user by using a `map()` transformation. From there, the elements are enqueued in a *shuffle queue*, from which the *prefetcher* extracts a pseudo-random subset of the elements that it then assembles into mini-batches. Finally, the mini-batches arrive in the *batch queue*, from where they are consumed by `get_next()`.

Data pipelines are highly customizable via composability. A basic pipeline creates a one-to-one correspondence between the elements of the dataset and the iterator. This version can be extended by adding more intermediate stages, and parameterized through primitives that return a new data pipeline, which in turn can be further optimized. For example, to combine two different data sources (or to parallelize I/O), two data pipelines `ds` and `other_ds` can be interleaved with the statement: `ds = ds.interleave(other_ds, num_parallel_calls=2)`. Similar primitives allow for the definition and parameterization (number of threads, buffer size, batch size, etc.) of map transformations, shuffling, prefetching, and batching.

This decoupled design allows data pipelines to achieve efficient overlapping of tasks, not only of a data pipeline with the training pipeline itself, but also of multiple intermediate stages within the data pipeline. However, such overlapping also leads to contention for resources (computational units, memory, network bandwidth), which introduces non-trivial trade-offs. For example, memory can be split between the data pipeline and training pipeline in different ways: metadata queue, shuffle queue, and prefetch queue. Computational units must similarly be shared between DL training and threads allocated to the data pipeline. Thus, fine-tuning a data pipeline is challenging.

To address this issue, data pipelines combine a series of optimizations that are applied both statically and dynamically. Static optimizations can benefit from the fact that composability allows the expression tree of a data pipeline to be inspected and updated at runtime to use more efficient methods where such exist. For example, if the iterator produces mini-batches, then it would be inefficient to apply a `map()` transformation individually to each training sample; instead, it can be applied in bulk after assembling an entire mini-batch. This is an example of a fusion operation.

Opportunities for dynamic arise because there are many variables that are known only at runtime: properties of the training samples (e.g., image size) and available resources (e.g., computational units, memory, network bandwidth). Thus, data pipelines implement an auto-tuning mechanism that redistributes resources among the intermediate stages so as to minimize the expected latency of producing an element that is obtained by `get_next()`. Specifically, the performance of each intermediate stage (obtained from the expression tree) is constantly monitored at runtime in order to build and update an analytical model for it. Analytical models are composed and optimized using gradient descent, in a manner similar to the actual DL training. Using this approach, the entire end-to-end pipeline can be optimized. This aspect is essential, because global resource contention inevitably leads to cases in which locally optimal decisions are suboptimal. For example, they may employ excessive parallelism and buffering, which in turn lead to inefficient thread scheduling, memory utilization, and caching locality.

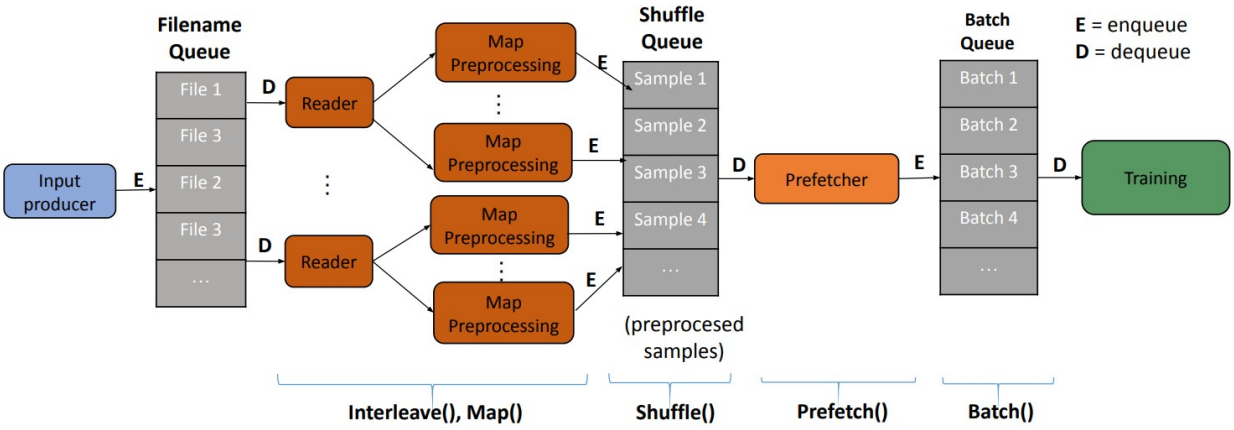


Figure 1: The data pipeline abstraction as implemented in TensorFlow. It exposes an iterator that returns the next mini-batch to be used by the training process with each invocation. The pipeline is implemented as an asynchronous producer-consumer workflow with intermediate buffers where the training samples are read from the repository, encoded as tensors, optionally transformed using custom augmentation functions, shuffled and finally assembled as mini-batches.

4 CHALLENGES AND OPPORTUNITIES

4.1 Collaborative producer-consumer queues

Data pipelines are designed to be used individually on each compute node. As a consequence, they are unaware of each other and may perform sub-optimally at large scale. For example, in a data-parallel training scenario in which many identical model replicas average their gradients, input data are typically shuffled and partitioned among multiple model replicas, each of which then visits a different set of training samples. To this end, each model replica could create a data pipeline for its assigned partition. However, after one epoch, a reshuffling and repartitioning is necessary in order to preserve the global randomness of sampling. Unaware that most input data are already cached, pre-processed, shuffled, and batched on other nodes, the data pipelines will start from scratch, thereby incurring redundant overheads (I/O contention to the parallel file system, resource contention due to overlapping with the training pipeline) that may slow DL training.

This effect is even more pronounced for ensemble learning, neural network architecture search and hyper-parameter optimization, where a large number of model variations need to be explored concurrently. In this case, the different data pipelines incur redundant overheads even during the same epoch, because the training data needed by one model variation may be cached by another model variation. An opportunity in this context is to allow the data pipelines to be aware of each other's producer-consumer queues and to share their content in order to optimize their performance and scalability as a group. Unlike state-of-art collaborative caching approaches, this approach would expose not only input data but also intermediate data (e.g., training samples after they were transformed by `map()`), which may further reduce redundant overheads. However, this approach also requires that data pipelines be able to discover new intermediate data dynamically as they become available, which introduces the need for advanced metadata management techniques.

4.2 Adaptive streaming with rehearsal support

When, as in continual learning, input data are streamed directly from a data source without being accumulated on a storage repository, it is not sufficient simply to minimize the time required to access data pipeline iterators. It is also important to adapt to the rate at which training samples arrive. Specifically, if the stream generates input data more slowly than are consumed by the iterator, then the overall training pipeline will be slowed down. On the other hand, if the stream generates data more rapidly than the can be consumed by the iterator, we encounter the difficult question of what to do with the excessive input data. Should we simply drop them at random, or alternatively buffer them under the assumption that the production/consumption rate will change in the future? If buffering is feasible, where can it be done? There are multiple choices, each with its own trade-offs: (1) a sufficiently large storage repository can be used as a buffer, with the caveat that subsequent reads will incur I/O overheads; (2) spare capacity in the queues of other data pipelines can be used for buffering, although this may interfere with their own optimizations.

A related problem is how to store the exemplars needed for rehearsal, as discussed in Section 1. If exemplars are persisted in the storage repository, then more can be accumulated for better diversity (especially when considering that the same exemplar can be augmented by the data pipeline in different ways). However, exemplars are then more costly to access and process. Another approach is to introduce an additional queue (e.g., rehearsal queue) to mitigate this problem, at the expense of complicating the data pipeline and introducing more resource contention.

4.3 Decoupled design

Efforts to build data pipelines are not limited to the Tensorflow ecosystem. For example, NVIDIA's *DALI* (The NVIDIA Data Loading Library) [15] aims to provide highly optimized building blocks for loading and processing image, video and audio data for DL applications. The key focus is on portability: it can be used as a drop-in

replacement for built in data loaders and data iterators of popular DL frameworks, including Tensorflow data pipelines. DALI supports a wide range of data formats and augmentations, while taking advantage of vendor-specific optimizations, such as leveraging a direct data path between storage and GPU memory (using GPUDirect). As a consequence, the native data pipelines provided by the DL frameworks, while offering better integration with the data-flow graphs, may not always be the preferred choice of users. Thus, optimizations such as collaborative producer-consumer queues and streaming with rehearsal support should ideally be abstracted as a separate layer that can integrate with more than a single data pipeline effort.

Fortunately, efforts such as DALI rely on a similar architecture as the Tensorflow data pipelines, sharing the same principle of chaining producer-consumer queues. Thus, an opportunity in this context is to provide a unified API to manage generic objects that are cached collaboratively in the queues of multiple compute nodes. For example, given the pseudo-random nature of I/O accesses and other operations in general, the interactions with the queues and caches are deterministic. Therefore, given a fixed caching algorithm and a fixed seed used by the pseudo-random number generator of each compute node, the availability of data and even memory offset on a remote compute node can be determined without additional metadata that needs to be exchanged between the compute nodes. Such capabilities can be encapsulated and exposed as a separate layer for use with Tensorflow pipelines, DALI and potentially other frameworks.

5 PRELIMINARY EVALUATION

5.1 Experimental Setup

Our experiments were performed on Argonne's *ThetaGPU* cluster, a testbed specifically optimized for training DNN models at scale. It comprises 24 NVIDIA DGX A100 nodes, each with eight NVIDIA A100 Tensor Core GPUs and two AMD Rome CPUs. Memory-wise, each node is equipped with 1 TB of DDR4 memory and 320 GB GPU memory, for a total of 24 TB DDR4 and 7.6 TB GPU memory. The nodes are interconnected using 20 Mellanox QM9700 HDR200 40-port switches wired in a fat-tree topology. External storage is provided by a Lustre parallel file system deployment with aggregate 250 GB/ bandwidth, mounted using POSIX (referred to as the *PFS*).

In terms of software, we use *TensorFlow* 2.5.0 and *Horovod* 0.20.3. Communication between nodes is facilitated by *OpenMPI* 4.1.1, which is used by Horovod.

5.2 Methodology

In order to quantify the limitations of the TensorFlow data pipelines, we study both the I/O overheads and the impact of those overheads on overall training duration under various data-parallel training scenarios. To enable data-parallelism, we rely on the *Horovod* [22] runtime library. Specifically, Horovod hides the details of parallelization by wrapping around the optimizer normally instantiated by a Keras DL application. Using this approach, Horovod augments the data-flow graph transparently to average the local gradients of all workers using an all-reduce collective communication pattern (e.g., as provided by MPI implementations) before proceeding with the weight updates.

To measure the I/O overheads, we use Darshan [23], an I/O profiling tool that transparently intercepts all POSIX system calls in order to generate statistics (number of calls, durations, sizes, etc.). The impact on the training duration is non-trivial to measure, because the data pipelines overlap the sample pre-pocessing with the mini-batch training. Thus, we rely on an indirect method that compares two data storage strategies: (1) the dataset is shared by all model replicas of the data-parallel training via the PFS; (2) the dataset is cached in-memory on each compute node by using the `/dev/shm` mount point, which features a `tmpfs` filesystem. In the case of (2), the I/O overheads are negligible. Thus, by measuring the difference between the end-to-end runtimes of (1) and (2), we can infer by how much time the data pipelines stall the training process due to the I/O overheads of the PFS. As we shall see, we find that the I/O overheads of the PFS and the duration of the stalls are high, thereby justifying the opportunities mentioned in Section 4.

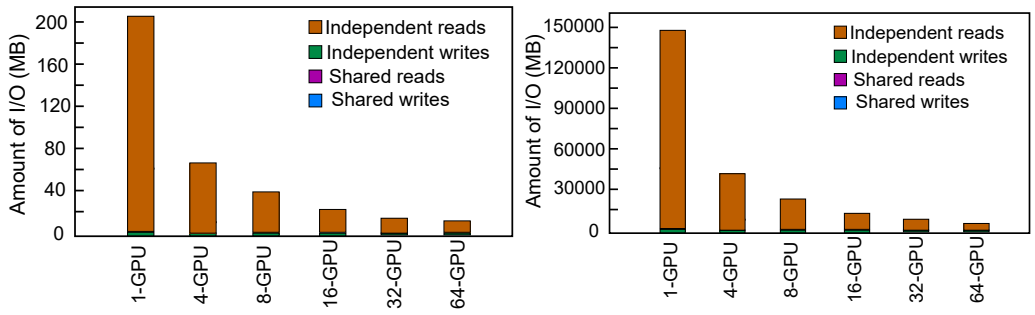
Application: ResNet-50. We use *ResNet-50* [6] as a representative DL model that is frequently used as a DL benchmark. Specifically, it is a family of DNN where the layers learn residual functions with reference to the input layers, instead of learning unreferenced functions. This allows ResNet to train extremely deep neural networks with 150+ layers. For the purpose of this work, we focus on the variant with 50 layers, which has a reference implementation for use with *Keras* and *Horovod*. Although ResNet-50 is deep, its overall size is relatively small (i.e., in the order of MiB). Therefore, the model and its intermediate computational states can be cached in the GPU memory and does cause any I/O related interference that may influence the study of the data pipelines.

Datasets: ImageNet and Tiny-ImageNet. As input to ResNet-50, we have chosen two standardized datasets. ImageNet is a widely used dataset in the evaluation of DL models for image classification. It comprises 1.28 million training images and 50,000 validation images, most with a size of between 10 KB and 100 KB, and each assigned to one of 1000 classes. Tiny-ImageNet is similar to ImageNet, but smaller, with just 200 image classes, a training dataset of 100,000 images, a validation dataset of 10,000 images, and a test dataset of 10,000 images. All images are of size 64×64.

5.3 Results

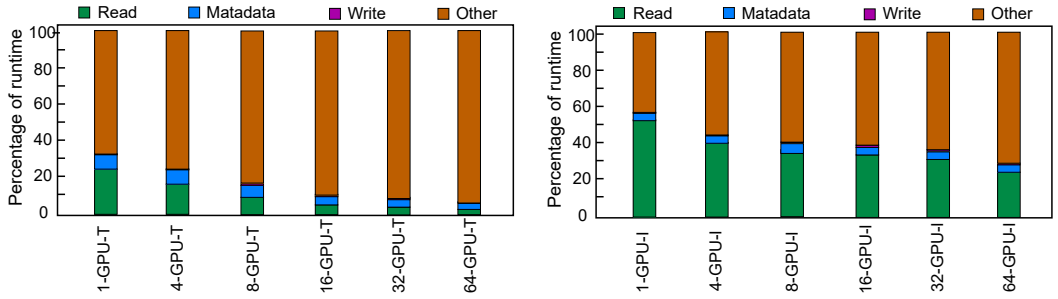
We first examine the I/O overheads captured by Darshan for the case when the dataset is shared via the PFS. We study strong scaling in two scenarios: (1) increasing number of GPUs per node (Figure 3a); and (2) increasing number of nodes with eight GPUs/node (Figure 3b).

Figure 2 confirms that DNN training is a read-intensive scenario in which each process predominantly performs independent read operations; the independent write operations incur only minimal overhead (cached Python JIT compiled code and temporary files). The figure also shows there are no shared read or write operations, which confirms the data pipelines do not produce intermediate temporary files that shared and potentially subject to concurrency control. Note that the read amount per process decreases with increasing number of GPUs, due to partitioning.



(a) Tiny-ImageNet. **(b) ImageNet.**

Figure 2: Amount of I/O (independent reads, independent writes, shared reads, shared writes) per process during ResNet-50 training. The Y axis aggregates all I/O amounts performed by all I/O operations.



(a) Tiny-ImageNet. **(b) ImageNet.**

Figure 3: Duration of PFS I/O operations (reads, writes, metadata queries) per process as a percentage of the total runtime for an increasing number of GPUs (up to eight per node).

Table 1: End-to-end runtime and cumulative duration of read and metadata I/O operations, in seconds, per process when training Resnet-50 on Tiny-Imagenet.

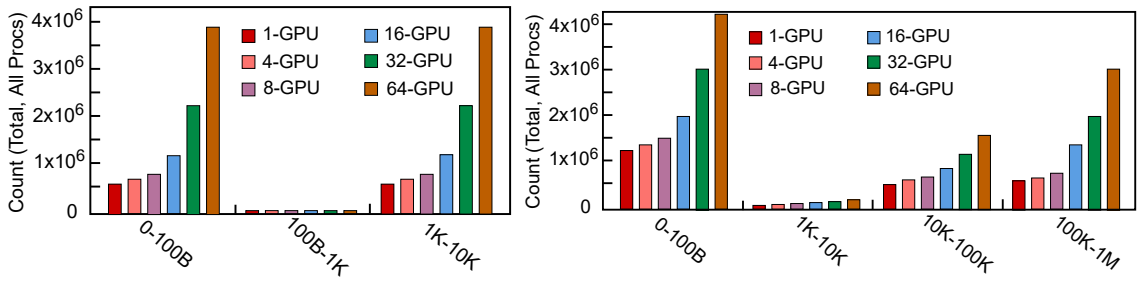
	GPUs					
	1	**4**	**8**	**16**	**32**	**64**
PFS reads	322	52.7	24.5	12.4	4.93	3.3
PFS metadata	121	31.6	15.6	10.5	6.8	5.0
PFS runtime	1473	421	275	221	171	160
Cached runtime	985	362	269	217	168	157

Table 2: End-to-end runtime and cumulative duration of read and metadata I/O operations, in seconds, per process when training Resnet-50 on Tiny-Imagenet.

	GPUs					
	1	**4**	**8**	**16**	**32**	**64**
PFS reads	19473	3441	1502	859	449	217
PFS metadata	1233	326.7	211.1	145.2	108.9	73.1
PFS runtime	36247	8345	4298	2709	1744	1209
Cached runtime	15263	3852	1975	1038	576	364

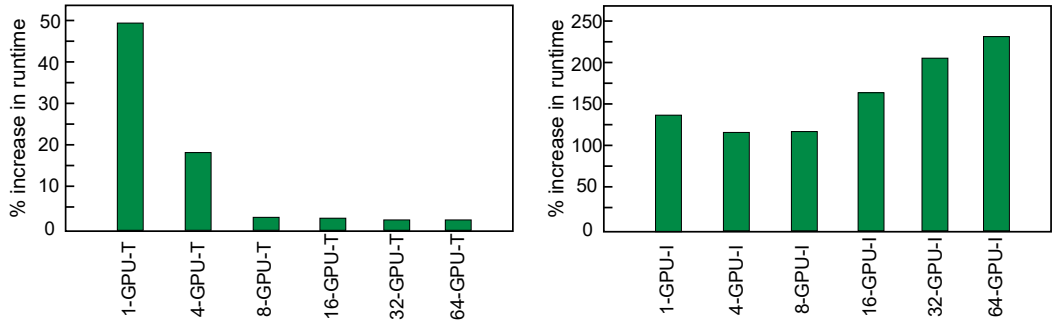
There are several interesting observations with respect to the read overheads, depicted in Figure 3. For a single node, despite no concurrent access to the PFS, the read overheads are high. For a single GPU, they reach up to 25% of the total runtime for Tiny-ImageNet and 50% for ImageNet. The metadata overheads are also non-trivial, reaching up to 5% of total runtime. With an increasing number of GPUs per node (4 and 8), both the read and metadata overheads begin to decrease due to OS-level caching. However, this is only possible because our nodes are equipped with 1 TB of main memory and ResNet-50 is a relatively small model. With increasing model complexity, there will be less free memory available for OS-level caching and therefore diminished benefits. For an increasing number of nodes (more than eight GPUs), the read and metadata overheads keep decreasing for Tiny-ImageNet, which is expected because different nodes can read from different I/O servers in parallel, therefore achieving a higher aggregated I/O bandwidth. However, the decrease of the I/O overheads is much slower for ImageNet, which means the PFS is experiencing I/O bottlenecks under concurrency, even for a small number of nodes.

To explain these findings better, we depict in Figure 4 the number of read operations broken down by size ranges. We see that the total number of read operations in the 0–100 bytes range equals the number of files for each dataset. This is due to the fact that the data pipeline needs to read the header of each image to determine its size and allocate the corresponding tensors before following up with a subsequent read of the content. Since there are many files



(a) Tiny-ImageNet. (b) ImageNet.

Figure 4: Bytes per read operation when training Resnet-50. Each label on the X axis represents a different size range, while the Y axis counts the number of read operations in each range.



(a) Tiny-ImageNet. (b) ImageNet.

Figure 5: PFS impact measured as percent increase of end-to-end runtime over in-memory caching. The X-axis labels refer to the number of GPUs used for training with the tiny (T) or full (I) ImageNet dataset. Lower is better.

in each dataset, the resulting aggregate metadata overhead is high. Furthermore, since each file is small, most read operations involve less than 10 KB in the case of Tiny-ImageNet and are in the order of 100 KB in the case of ImageNet. The PFS was not optimized for this pattern, which leads to an overall throughput in the order of a few MB/s per node—two orders of magnitude less than the peak I/O throughput.

For completeness, we show in Tables 1 and 2 the actual measurements of the end-to-end runtime and cumulative duration of read and metadata I/O operations. The cached runtime corresponds to the case when the full dataset is cached in memory on each compute node (on the `/dev/shm`).

Figure 5 depicts the percent increase in runtime due to reading from the PFS vs. the cached runtime. For Tiny-ImageNet, the increase in runtime is high when using a single GPU (up to 50%), but it drops sharply for an increasing number of GPUs. Thus, we can conclude that despite large read overheads, the data pipelines successfully mask them without causing stalls. However, the situation is reversed for ImageNet: the increase in runtime is much higher (up to 200%) and it only drops slightly from one GPU to four and eight GPUs, but then goes up again much faster. This confirm our hypothesis that a PFS is likely to become a significant bottleneck both at small scale and large scale.

Furthermore, another important observation can be made: the relative increase in runtime of PFS vs. cached is much higher than the PFS I/O overhead (reads and metadata operations) to PFS runtime ratio. Thus, we can conclude that the stalls caused by the slow PFS read operations are augmented further down the data pipelines, which means it will not suffice to make the read operations faster.

6 CONCLUSIONS

We have reviewed the challenges and opportunities of designing efficient data pipelines for data-parallel training of DNN models, which is essential in the design of flexible AI runtimes deployed on HPC systems. Based on a preliminary evaluation of Tensorflow data pipelines on several nodes and GPUs, we highlighted a significant I/O overhead due to both data ingestion and metadata operations that involve a parallel file system, which can cause up to 200% increase in execution time. Based on this preliminary evaluation, we plan to pursue future work that enables data pipelines to collaborate and cache training samples as a group, both with respect to raw input data and intermediate transformations. Such an approach aims to minimize expensive I/O interactions with parallel file systems and other external repositories, thereby reducing observed I/O overheads. Furthermore, streaming training data for continual learning and/or reinforcement learning often involves rehearsal of representative historic training samples and/or learning patterns, which can be implemented in a distributed fashion by leveraging collaborative caching. Finally, the rising popularity of a variety of data loading and preprocessing frameworks makes it important to focus on a decoupled design that expose a unified API that can be leveraged in such frameworks.

REFERENCES

[1] Fahim Chowdhury, Yue Zhu, Todd Heer, Saul Paredes, Adam Moody, Robin Goldstone, Kathryn Mohror, and Weikuan Yu. 2019. I/O characterization and performance evaluation of BeeGFS for deep learning. In *8th International Conference on Parallel Processing*. Kyoto, Japan, 80:1–80:10.

[2] Ching-Hsiang Chu, Pouya Kousha, Ammar Ahmad Awan, Kawthar Shafie Khorassani, Hari Subramoni, and Dhabaleswar K. Panda. 2020. NV-group: link-efficient reduction for distributed deep learning on modern dense GPU systems. In *34th ACM International Conference on Supercomputing*. Virtual, 1–12.

[3] Jeffrey Dean et al. 2012. Large scale distributed deep networks. In *25th International Conference on Neural Information Processing Systems*. Lake Tahoe, USA, 1223–1231.

[4] Nikoli Dryden, Roman Böhringer, Tal Ben-Nun, and Torsten Hoefler. 2021. Clairvoyant prefetching for distributed machine learning i/o. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 1–15.

[5] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. 2016. *Deep Learning (Adaptive Computation and Machine Learning series)*. MIT Press.

[6] K. He, X. Zhang, S. Ren, and J. Sun. 2016. Deep residual learning for image recognition. In *IEEE Conference on Computer Vision and Pattern Recognition*. Las Vegas, USA, 770–778.

[7] Andrei Ivanov, Nikoli Dryden, Tal Ben-Nun, Shigang Li, and Torsten Hoefler. 2021. Data movement is all you need: a case study on optimizing transformers. In *4th Conference on Machine Learning and Systems*. Virtual.

[8] Timothy P Lillicrap, Jonathan J Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. 2015. Continuous control with deep reinforcement learning. *arXiv preprint arXiv:1509.02971*.

[9] X. Liu, C. Wu, M. Menta, L. Herranz, B. Raducanu, A. D. Bagdanov, S. Jui, and J. van de Weijer. 2020. Generative feature replay for class-incremental learning. In *IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 915–924.

[10] David Muñoz, Camilo Narváez, Carlos Cobos, Martha Mendoza, and Francisco Herrera. 2020. Incremental learning model inspired in rehearsal for deep convolutional networks. *Knowledge-Based Systems*, 208, 106460.

[11] Derek G. Murray, Jiri Simsa, Ana Klimovic, and Ihor Indyk. 2021. Tf.data: a machine learning data processing framework. (2021). arXiv: 2101.12127 [cs.LG].

[12] Deepak Narayanan, Aaron Harlap, Amar Phanishayee, Vivek Seshadri, Nikhil R. Devanur, Gregory R. Ganger, Phillip B. Gibbons, and Matei Zaharia. 2019. PipeDream: generalized pipeline parallelism for DNN training. In *27th ACM Symposium on Operating Systems Principles*. Huntsville, Canada, 1–15.

[13] Bogdan Nicolae. 2020. DataStates: Towards lightweight data models for deep learning. In *SMC'20: The 2020 Smoky Mountains Computational Sciences and Engineering Conference*. Nashville, United States, 117–129. DOI: 10.1007/978-3-030-63393-6_8.

[14] Bogdan Nicolae, Adam Moody, Elsa Gonsiorowski, Kathryn Mohror, and Franck Cappello. 2019. VeloC: towards high performance adaptive asynchronous checkpointing at large scale. In *IEEE International Parallel and Distributed Processing Symposium*. Rio de Janeiro, Brazil, 911–920.

[15] [n. d.] NVIDIA Data Loading Library. https://developer.nvidia.com/DALI. ().

[16] German I. Parisi, Ronald Kemker, Jose L. Part, Christopher Kanan, and Stefan Wermter. 2019. Continual lifelong learning with neural networks: a review. *Neural Networks*, 113, 54–71.

[17] Sarunya Pumma, Min Si, Wu-chun Feng, and Pavan Balaji. 2019. Scalable deep learning via I/O analysis and optimization. *ACM Transactions on Parallel Computing*, 6, 2, 1–34.

[18] Sylvestre-Alvise Rebuffi, Alexander Kolesnikov, Georg Sperl, and Christoph H. Lampert. 2017. ICaRL: incremental classifier and representation learning. In *IEEE Conference on Computer Vision and Pattern Recognition*, 5533–5542.

[19] Daniel A Reed and Jack Dongarra. 2015. Exascale computing and big data. *Communications of the ACM*, 58, 7, 56–68.

[20] A. Robins. 1993. Catastrophic forgetting in neural networks: the role of rehearsal mechanisms. In *The First New Zealand International Two-Stream Conference on Artificial Neural Networks and Expert Systems*, 65–68.

[21] Andrei A. Rusu, Neil C. Rabinowitz, Guillaume Desjardins, Hubert Soyer, James Kirkpatrick, Koray Kavukcuoglu, Razvan Pascanu, and Raia Hadsell. 2016. Progressive neural networks. arXiv 1606.04671. (2016).

[22] Alex Sergeev and Mike Del Balso. [n. d.] Meet Horovod: uber's open source distributed deep learning framework for tensorflow. https://eng.uber.com/horovod. ().

[23] Shane Snyder, Philip Carns, Kevin Harms, Robert Ross, Glenn K Lockwood, and Nicholas J Wright. 2016. Modular HPC I/O characterization with Darshan. In *5th Workshop on Extreme-scale Programming Tools*. IEEE, 9–17.

[24] Chih-Chieh Yang and Guojing Cong. 2019. Accelerating data loading in deep neural network training. In *IEEE 26th International Conference on High Performance Computing, Data, and Analytics*, 235–245.

[25] Yang You, Zhao Zhang, Cho-Jui Hsieh, and James Demmel Kurt Keutzer. 2018. ImageNet training in minutes. In *47th International Conference on Parallel Processing*. Eugene, USA, 1–10.

[26] Z. Zhang, L. Huang, J. Pauloski, and I. T. Foster. 2020. Efficient I/O for neural network training with compressed data. In *IEEE International Parallel and Distributed Processing Symposium*, 409–418.