2021 IEEE International Conference on Cluster Computing (CLUSTER) | 978-1-7281-9666-4/21/$31.00 ©2021 IEEE | DOI: 10.1109/Cluster48925.2021.00096

The Case for Storage Optimization Decoupling in Deep Learning Frameworks

Ricardo Macedo, Cláudia Correia, Marco Dantas, Cláudia Brito
INESC TEC & University of Minho
{ricardo.g.macedo, claudia.s.mendonca, marco.f.dantas, claudia.v.brito}@inesctec.pt

Weijia Xu
Texas Advanced Computing Center
xwj@tacc.utexas.edu

Yusuke Tanimura, Jason Haga
National Institute of Advanced Industrial Science and Technology
{yusuke.tanimura, jh.haga}@aist.go.jp

João Paulo
INESC TEC & University of Minho
joao.t.paulo@inesctec.pt

***Abstract*—Deep Learning (DL) training requires efficient access to large collections of data, leading DL frameworks to implement individual I/O optimizations to take full advantage of storage performance. However, these optimizations are intrinsic to each framework, limiting their applicability and portability across DL solutions, while making them inefficient for scenarios where multiple applications compete for shared storage resources.**

We argue that storage optimizations should be decoupled from DL frameworks and moved to a dedicated storage layer. To achieve this, we propose a new Software-Defined Storage architecture for accelerating DL training performance. The data plane implements self-contained, generally applicable I/O optimizations, while the control plane dynamically adapts them to cope with workload variations and multi-tenant environments.

We validate the applicability and portability of our approach by developing and integrating an early prototype with the TensorFlow and PyTorch frameworks. Results show that our I/O optimizations significantly reduce DL training time by up to 54% and 63% for TensorFlow and PyTorch baseline configurations, while providing similar performance benefits to framework-intrinsic I/O mechanisms provided by TensorFlow.

***Index Terms*—Software-Defined Storage, Application-specific storage, Deep Learning frameworks**

I. Introduction

Recently, the research and practical use of deep learning (DL) techniques have experienced an unprecedented growth. To provide accurate predictions, DL models must be trained with diverse datasets that range from a few MiB [1], [2] to several TiB [3]–[5] in size. Thus, DL training has become prohibitively expensive and time-consuming, and extensive work has been done to accelerate training performance, including specialized hardware [6], [7], schedulers [8], [9], and carefully engineered optimizations at the compiler [10], [11], communication [12]–[14], and GPU [15]–[17] layers.

With the advent of such optimizations, however, the training bottleneck has shifted to the storage layer [18]–[20]. Specifically, data is randomly read from backend storage, making it harder to leverage caching and data tiering mechanisms, while significantly impacting training performance. To address this issue, recent work has been focused on developing system-specific I/O optimizations over DL frameworks (*e.g.,* TensorFlow [21], PyTorch [22], Chainer [23]), such as caching and prefetching [19], [20], [24]–[26], storage tiering [26]–[28], and data sharding [29], resulting in improved data loading performance and overall training time.

This approach however, has two main drawbacks. First, current optimizations are single-purposed, as they are tightly integrated within the core of each DL framework and cannot be easily decoupled, making it challenging to port them to other frameworks that would equally benefit from such optimizations. This effect becomes further amplified when considering complex I/O stacks made of multiple layers throughout the I/O path, whose storage access is distributed over several types of nodes, which is the case of modern high-performance computing (HPC) infrastructures [30]–[32]. Second, in shared environments where multiple systems operate concurrently and compete for shared storage resources, framework-specific optimizations only have partial visibility of the overall I/O stack, which can lead to conflicting optimizations and misconfiguration [33], [34], I/O contention [32], [35], and performance variation [36]–[38].

To address these challenges, we argue that *I/O optimizations should be decoupled from DL frameworks and moved to a dedicated storage layer with system-wide visibility*. By moving I/O optimizations to a dedicated, adaptable, and extensible storage component, they become generally applicable and portable across DL frameworks, improving their applicability and adoption. Further, rather than operating in isolation, optimizations must have global visibility of the infrastructure to ensure coordinated and holistic control of all resources.

In conformity with these design directives, we propose a new Software-Defined Storage (SDS) [39] architecture for accelerating DL training performance. Following a software-defined approach, DL I/O optimizations are decoupled into two planes of functionality. The *control plane* is a logically centralized entity with global visibility that defines the control logic of I/O optimizations through user-defined policies, while the *data plane* implements the actual I/O logic to be employed over DL requests. Under this design, I/O optimizations are implemented as self-contained, generally applicable, and extensible building blocks (*data plane*) that can be adapted to cope with volatile system requirements and workload variations

978-1-7281-9666-4/21/$31.00 ©2021 IEEE
DOI 10.1109/Cluster48925.2021.00096

(*control plane*). Contrary to system-specific optimizations, our design is framework-agnostic and can be extended with ease, promoting portability and code-reuse.

Prior SDS work is ineffective to address these challenges, as it is designed for enforcing policies at a given layer (*e.g.,* hypervisor [40], [41], file system [42], object store [43]) or is only applicable over a specific storage context (*e.g.,* virtualization [40], [44], cloud-based environments [40], [45]).

We developed PRISMA, an early prototype of our SDS system, where the data plane implements a parallel data prefetching mechanism that reads training data in advance and stores it in an in-memory buffer to serve incoming I/O requests; and the control plane provides an auto-tuning control algorithm that automatically adjusts the number of threads reading from storage and the size of the in-memory buffer. We validate its applicability and portability by optimizing the training performance of two popular DL frameworks, namely TensorFlow and PyTorch, under different models and configurations. The integration of our solution only required adding 10 and 35 LoC to TensorFlow and PyTorch, respectively. Further, our approach does not require any manual input from users nor modifications to the internal DL framework workflow.

Preliminary results show that our SDS-enabled I/O optimizations can significantly decrease DL training time. Under PyTorch, our approach observes performance gains up to 63%, outperforming any manually optimized setup under 8 parallel I/O workers, while also performing constantly across different combinations of workers. For TensorFlow, PRISMA provides similar performance benefits to framework-intrinsic mechanisms while reducing training time up to 54% when compared to a vanilla configuration.

In summary, this paper makes the following contributions:

- **Redesign DL frameworks' storage optimizations.** We propose a new approach for implementing storage optimizations of DL frameworks. We propose a novel SDS architecture for accelerating DL training performance, that repurposes monolithic and framework-specific I/O optimizations into SDS-enabled ones (§III).
- **Middleware for accelerating training performance.** We implemented PRISMA, a framework-agnostic SDS-enabled middleware that improves the I/O performance of modern DL frameworks (§IV).
- **Integration of PRISMA with popular DL frameworks.** We integrated PRISMA with TensorFlow and PyTorch, validating its applicability and feasibility over different DL frameworks (§IV).
- **Experimental evaluation of our approach.** Experimental evaluation demonstrating the performance of PRISMA when compared to TensorFlow and PyTorch (§V).

This paper is organized as follows. §II details the background and limitations of traditional I/O optimizations in DL frameworks. §III proposes a new SDS system for DL frameworks, while §IV describes a preliminary implementation of our approach. §V demonstrates obtained experimental results. §VI surveys existing related work. §VII discusses future research directions and concludes the paper.

II. BACKGROUND AND MOTIVATION

For DL models to provide accurate predictions, they must be trained with large and varied datasets. During the training phase, data samples are continuously read from storage in random order to avoid overfitting [46]. This, however, incurs significant performance overhead to the training process [19], [20], [26], as conventional file systems are optimized for large and sequential accesses instead of small and random ones.

After being read from storage, samples are preprocessed in memory, typically by the CPU, and then batched and transferred to GPU to train the neural network. While ideally there would always be batches available to be consumed by the GPU, in I/O-bound models, computing is faster than data loading causing the GPU to sit idle most of the time [19], [47]. Thus, modern DL frameworks (*e.g.,* TensorFlow) provide an optimized data loading pipeline composed of specific I/O optimizations to accelerate DL training, including caching and data prefetching [20], [24]–[26], parallel I/O [24], [25], storage tiering [26]–[28], and auto-tuning mechanisms [48]. However, these optimizations have two main limitations:

Tightly coupled optimizations. Current DL I/O optimizations are framework-specific, as they are tightly integrated within the core of each framework. Implementing such optimizations requires significant system rewrite, reducing their portability and wider adoption over DL frameworks that would equally benefit from such optimizations. Further, this design makes fine-tuning and extending optimizations to cope with different system requirements an increasingly complex and time-consuming task, as it demands deep understanding of the framework's internal operation model.

For example, let us consider TensorFlow's autotuning optimization [48]. This mechanism enables the number of I/O threads and the buffer size to be automatically adapted during training. For this to be possible, TensorFlow periodically collects runtime information about each transformation of the input pipeline, while a background task uses this information to continually distribute the available CPU and memory across all parallel transformations. As this optimization is tightly integrated with TensorFlow's internal mechanisms, porting it to other frameworks that would also benefit from it, such as PyTorch and Chainer, is not trivial and requires profound system refactoring. This is equally true for optimizations that seek to improve storage backend performance (*e.g.,* optimized data formats [49], modified access patterns [50]).

Partial visibility. System-specific optimizations are single-purposed and applied over a subset of the system's scope. This leads optimizations to act in isolation, as they do not consider systems that operate concurrently over the same resources, resulting in conflicting optimizations and misconfigurations [33], [34], I/O contention, and performance variation. For instance, while computational resources (*e.g.,* CPU, GPU) can be fairly isolated in HPC infrastructures, through exclusive node allocation or resource isolation techniques (*e.g.,* Linux's cgroups, containerization), the same does not apply for I/O resources, including network and shared storage. As such, it

is extremely challenging to efficiently execute hundreds of DL jobs that compete for shared storage resources [32], [36]–[38], for example, at distributed storage backends such as Lustre [51], GPFS [52], and BeeGFS [53].

To address these challenges, we advocate for optimization decoupling, where I/O optimizations are moved to an independent layer, being generally applicable and portable to different DL frameworks. We also argue that optimizations should also have system-wide visibility, and ensure coordinated and holistic control of all storage resources.

III. Software-Defined Storage for Deep Learning Frameworks

We propose a new Software-Defined Storage system for accelerating DL training performance. Rather than designing monolithic and framework-specific optimizations, we decouple these into two planes of functionality, namely *control* and *data*. The control plane holds the control logic of I/O optimizations through user-defined policies (*e.g.,* caching and tiering policies, prioritize workloads) that orchestrate the overall system stack. The data plane contains the actual I/O mechanisms required to enforce such policies, which are implemented as self-contained and extensible building blocks, and provide the necessary tuning knobs to be adjusted upon workload and policy variations. Such a design allows to implement I/O optimizations with system-wide visibility that are generally applicable and portable across DL frameworks while ensuring coordinated and holistic control of storage resources.

A. Design

Figure 1 outlines the design of our approach. The **data plane** is designed as a framework-agnostic storage middleware that sits between the DL framework (*e.g.,* TensorFlow, PyTorch) and the backend storage system (*e.g.,* local or distributed storage). It is made of multiple stages that can be placed either locally (*i.e.,* single compute node) or distributed. Internally, each stage is organized into three main modules. It provides an *optimization object* abstraction that allows users to implement custom storage optimizations to apply over DL requests. Examples of such objects include data prefetching, parallel I/O, and storage tiering. Such an abstraction provides users with the means to develop new and reusable I/O mechanisms tailored for enforcing specific DL optimizations. It also exposes a *POSIX-compliant interface* that intercepts the DL framework's storage requests (*e.g.,* `read`, `pread`) and submits them to the *optimization objects* for the respective I/O logic to be applied. Finally, it implements a *control interface* that communicates with the control plane for internal stage management (*e.g.,* policies) and monitoring. Contrarily to traditional approaches, the I/O mechanisms implemented in the data plane focus solely on optimizing the I/O interaction between DL frameworks and the file system, being agnostic of the remainder optimizations of the training pipeline (*e.g.,* scheduling, GPU and memory management, networking).

The **control plane** is a logically centralized component with system-wide visibility that controls and coordinates the

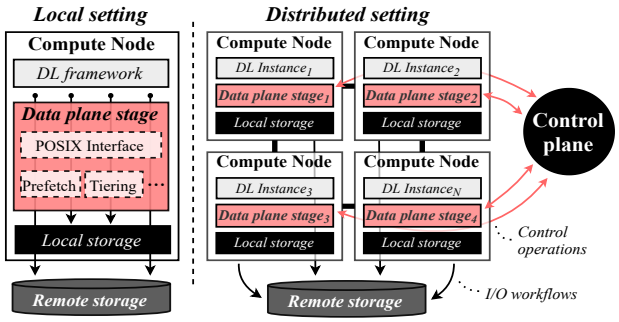

Fig. 1. SDS-enabled architecture for Deep Learning frameworks under local and distributed settings.

data plane stages and how I/O requests should be handled. It communicates with the data plane for collecting monitoring metrics (*e.g.,* cache hits, I/O rate) and enforcing storage policies to respond to workload variations. The control logic of I/O optimizations is implemented as centralized *control algorithms*, which are simpler and less error-prone than designing the corresponding decentralized versions [40]. This centralized control eases the development and maintenance of complex storage optimizations and enables holistic and coordinated control across the I/O stack. While logically centralized, the control plane is physically distributed and made of multiple controllers to meet the scalability and availability (in case of controller failures) requirements of large scale infrastructures.

Our design focuses on DL storage access optimizations, which could be used to complement existing compute or communication-based optimizations conducted at other phases of the training pipeline [13], [15], [54].

IV. Prisma

To demonstrate the feasibility of our approach we developed Prisma, a preliminary prototype that implements an SDS-enabled version of TensorFlow's [24] and PyTorch's [25] parallel I/O and data prefetching optimizations. Its main purpose is to always serve data from high-speed memory when the DL framework requests it. DL frameworks spawn one or more threads (or processes) to read training data, which are referred as *consumers*, and its number is oblivious to Prisma.

Data plane. The data plane implements a parallel data prefetching *optimization object* that proactively reads data from backend storage and holds it in an in-memory buffer. It performs parallel I/O using multiple threads (at most t), referred to as *producers*, that concurrently read data from storage. The order in which files are read is given by an internal FIFO queue that stores the filenames of dataset samples. A filenames list, populated by the DL framework at the beginning of the training phase, is shared with Prisma so it knows in advance which files will be requested. This file is created with a simple Python module that receives a list with the names

of all training files and shuffles them for each epoch.[1] Note that this process only requires changing the job's script, and does not change how files are shuffled and requested by the DL platform, which is important to avoid any impact on the accuracy of the trained model [46]. Each producer dequeues a filename from the queue and reads data from storage, which is then stored in the in-memory buffer so it can be later provided to the DL framework. The in-memory buffer stores at most N training samples. Since the training process only requires files to be read a single time for each epoch, the caching policy is quite straightforward — a training file is stored in the buffer whenever it is read by a producer and is evicted when a consumer requests it. As DL training is predominantly read-oriented [18], [19], PRISMA's *POSIX interface* exposes a single `read` method to intercept and service read requests to the DL framework.

Control plane. The control plane implements the control logic of our I/O optimization. For ease of development, it is implemented as a logical component of our middleware. Instead of delegating to the user the responsibility of finding the optimal combination of parallel reads (t) and buffer size (N) to use, we developed an auto-tuning control algorithm that automatically adjusts these parameters. The algorithm selects t and N to provide a balanced trade-off between performance and resource usage. To ensure this, the control algorithm uses a feedback control loop [55] that collects statistics from the data plane (*i.e.,* buffer usage) and continuously adjusts its parameters (*i.e., t* and *N*) until converging to an optimal configuration. This algorithm is similar to TensorFlow's auto-tuning mechanism [48], however, it is implemented as a control algorithm that can be easily extended and ported across different DL frameworks.

Integration with DL frameworks. To demonstrate the portability and applicability of our solution, we integrated PRISMA with TensorFlow and PyTorch. For TensorFlow, as it exposes different interfaces to interact with varied storage backends (*e.g.,* POSIX, HDFS, S3), we extended the existing POSIX file system backend and replaced the `pread` invocation with `Prisma.read`, which directly reads data from PRISMA rather than accessing the storage backend. This only required changing 10 LoC. For PyTorch, because it uses processes instead of threads, we implemented an inter-process communication client-server through UNIX Domain Sockets. For each spawned process, a PRISMA client instance is created to intercept all read invocations and submit them to the server to be handled. This required changing 35 LoC.

V. EVALUATION

We now conduct a set of experiments that seek to answer the following research questions:

- *Can* PRISMA *improve DL I/O performance?*
- *How does* PRISMA *compare with framework-intrinsic storage optimizations?*

[1]The filename shuffling process is performed identically to the original shuffle mechanism of the DL framework.

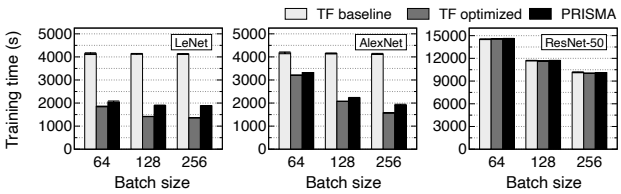

Fig. 2. Average training time of TensorFlow and PRISMA with the LeNet, AlexNet, and ResNet-50.

- *Are these benefits observable for different frameworks?*

Experimental setup. Experiments were conducted on a compute node of the AI Bridging Cloud Infrastructure supercomputer[2], equipped with two 20-core Intel Xeon processors, four NVidia Tesla V100 GPUs, 384 GiB of RAM, and a single 1.6 TiB Intel SSD DC P4600. Software-wise, it uses CentOS 7.5 with the Linux kernel v3.10 and XFS file system.

Dataset, models, and DL frameworks. We used the Imagenet dataset [56], that includes 1.28 million images ($\approx$ 138 GiB) for training and 50,000 images ($\approx$ 6 GiB) for validation. To ensure a comprehensive evaluation in terms of workload heterogeneity, and according to previous work [47], our experiments included I/O-bound models, namely LeNet [57] and AlexNet [58], and a compute-bound model, namely ResNet-50 [59]. We used TensorFlow v2.1.0 and PyTorch v1.7.0 to validate and compare our approach.

Methodology. We measured the elapsed training time of all experiments, each configured to run for 10 training epochs, and simultaneously use all 4 GPUs available in the compute node. Different batch size configurations were tested, namely 64, 128, 256. The results of each experiment concern the average and standard deviation of 5 runs.

A. TensorFlow

Three TensorFlow setups were used for the experiments.

- ***TF baseline*** provides a non-optimized deployment with single-threaded disk operations without data prefetching.
- ***TF optimized*** includes both disk I/O parallelism and prefetching optimizations, which are managed by TensorFlow's auto-tuning mechanism.
- **PRISMA** corresponds to the integration of PRISMA's data plane with the non-optimized TensorFlow.

Training time. Figure 2 depicts the training time for all setups. PRISMA significantly improves the performance of I/O-bound models, reducing training time by more than 50% for LeNet and 20% for AlexNet, when compared to *TF baseline*. For compute-bound models, similarly to *TF optimized*, PRISMA has no impact on training time.

Contrary to *TF baseline*, PRISMA and *TF optimized* improve training performance with larger batch sizes. With a batch size of 64 and the LeNet model, PRISMA executes over 2,047 seconds while *TF optimized* achieves 1,851 seconds,

[2][Online] Available: https://abci.ai.

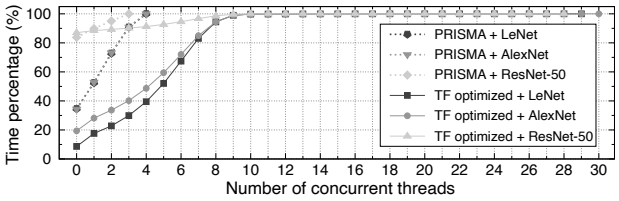

Fig. 3. Cumulative distribution function for the time percentage of each number of concurrent threads used by TensorFlow and PRISMA.

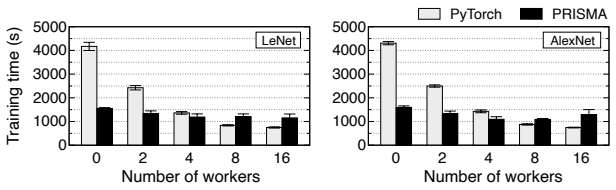

Fig. 4. Average training time of PyTorch and PRISMA with the LeNet and AlexNet models.

corresponding to a reduction in training time of 51% and 55%, respectively. When the batch size increases, *TF optimized* performs better than PRISMA. This is noticeable with LeNet under a batch size of 256. While PRISMA executes over 1,880 seconds, *TF optimized* achieves 1,363 seconds, which corresponds to a reduction in training time of 54% and 67% when compared to *TF baseline*, respectively.

PRISMA's prototype does not perform prefetching for validation files. In *TF optimized*, however, all read operations are backed by TensorFlow's I/O optimizations, explaining their difference in training performance. Nonetheless, contemplating the prefetching of validation files would be feasible and only require a few adjustments on the prototype, thus not affecting the design principles of our approach. Furthermore, as PRISMA's control algorithm is similar to the *TF optimized* auto-tuning mechanism (§IV), it is expected for both systems to perform similarly. The same may not hold true when considering other control algorithms.

Control algorithm. We also measured the percentage of time spent by I/O threads actively reading data from the backend storage. Figure 3 depicts the results for *TF optimized* and PRISMA setups. PRISMA only uses at most 4 concurrent threads (3 in the case of ResNet-50), while *TF optimized* uses 2-7x more threads for training. While *TF optimized* allocates the maximum number of threads (*i.e.,* 30) regardless of whether they are needed or not, PRISMA auto-tuning mechanism only allocates the necessary threads, avoiding overprovisioning while providing similar storage performance.

B. PyTorch

We evaluated a baseline PyTorch deployment with an increasing number of worker processes (0-16). Note that the number of workers must be chosen manually by users, while the optimal configuration may vary according to the targeted AI workload. PRISMA includes parallel I/O, prefetching, and auto-tuning mechanisms. Experiments were conducted using LeNet and AlexNet with a batch size of 256.

Training time. Figure 4 depicts the average training time for both setups under the LeNet and AlexNet models. PRISMA outperforms PyTorch with 0, 2 and 4 workers, presenting an absolute decrease in training time of 2,618, 1,085, and 176 seconds for LeNet and 2,710, 1,171, and 337 seconds for AlexNet. This improvement is justified by PRISMA starting prefetching samples before the epoch begins. PyTorch, however, decreases training time over PRISMA by 362 and 405 seconds for LeNet, and 211 and 542 for AlexNet with 8 and 16 workers, respectively. We observed that for 8+ workers, PRISMA presents a performance bottleneck upon the synchronization between consumer and producer threads accessing the in-memory buffer. This could be addressed by tuning PRISMA for PyTorch's operation model, specifically by improving the synchronization in multi-process scenarios.

Nonetheless, it is important to note that PRISMA performs similarly for different combinations of PyTorch workers. This is due to PRISMA's auto-tuning mechanism, which enables users to avoid running exhaustive and time-consuming preliminary experiments to find the best combination of workers.

VI. RELATED WORK

Software-Defined Storage. Existing SDS systems are designed for enforcing storage policies at a specific *storage layer* or *storage context*, thus not being suited for orchestrating modern DL frameworks [39]. Specifically, IOFlow [40], sRoute [44], and PSLO [41] tackle the virtualization layer. While suited for cloud-based environments, these solutions cannot be used over scenarios that require bare-metal access (*i.e.,* direct access to resources) such as HPC infrastructures and bare-metal cloud servers. Retro [45], Cake [60], and Crystal [43] implement performance and resource management policies over distributed file systems and object stores. SafeFS [42] provides a framework for implementing and stacking FUSE-based file systems on top of each other but introduces non-negligible overhead to most POSIX calls.

HPC-oriented SDS systems, namely Clarisse [30] and SIREN [31], implement resource management policies such as parallel I/O scheduling and bandwidth reservations to ensure isolation and QoS provisioning of HPC jobs. Thus, these are not suited for implementing the optimizations discussed in this paper (*e.g.,* data caching and prefetching).

PAIO [61] is a general-purpose SDS data plane framework that enables system designers to build custom-made data plane stages applicable over different I/O layers, including DL frameworks. While complementary to our contributions, the optimizations presented in this paper could be also implemented with PAIO. We leave this integration as future work.

I/O optimizations in DL. There has also been an increase in the research and proposal of I/O optimizations for DL frameworks. LMDBIO [18] introduces speculative parallel I/O and I/O staggering for improving DL workloads running on top of the LMDB key-value store. [62] extends Chainer to optimize

data loading under a shared storage deployment. Quiver [26] demonstrates how to efficiently cache data from remote to local storage. These optimizations are single-purposed and intrinsic to the core of each DL framework.

FanStore [27], DLFS [28], and DeepIO [50] focus on optimizing data fetching and distribution across local storage. Frameworks like DALI [54], CoorDL [19], and NoPFS [20] are also aligned with the reutilization principle, as they are designed to implement and reuse data preprocessing and I/O optimizations across DL frameworks. These optimizations operate in isolation with partial visibility, and thus cannot ensure coordinated and holistic control of the overall I/O stack.

HPC I/O libraries. Existing HPC-oriented I/O libraries are designed for optimizing the performance of HPC applications through optimized data formats and by redefining I/O workflows. NetCDF [63] and HDF5 [64] provide a set of libraries that support the creation, formatting, and access to portable data models that can represent complex data objects and metadata. ADIOS [65] is an I/O subsystem that offers a set of APIs that enable system designers to control the I/O workflows of scientific applications, including buffering, compression, and indexing. ADIOS2 [66] extends the ADIOS system to manage how data is produced and consumed in scientific applications in order to reduce the cost of integrating different data transport technologies. ROMIO [67] is a MPI-IO-based library that offers a two-phase I/O optimization scheme for efficiently accessing noncontiguous data. While these libraries could be used to implement the I/O optimizations proposed in our data plane, these would still operate in isolation and with partial visibility, and thus, would be unable to ensure coordinated and holistic control of the overall I/O stack.

VII. Conclusion and Discussion

We demonstrate the case for decoupling I/O optimizations from DL frameworks and moving them to a dedicated storage layer. Rather than following the traditional uncoordinated development model of implementing independent I/O optimizations, we follow an alternative approach. We propose a new SDS architecture for accelerating DL training performance and developed an early prototype that implements a parallel data prefetching optimization. We demonstrate its applicability and portability by integrating it with TensorFlow and PyTorch. Preliminary results show that our approach significantly outperforms baseline PyTorch and TensorFlow configurations, and achieves similar performance benefits as carefully engineered I/O optimizations in TensorFlow.

We believe our approach can be useful not only for accelerating the performance of existing DL frameworks but also to those currently being designed, saving developers from re-implementing custom and intrinsic I/O optimizations. We now discuss some open questions and future research directions.

Implementing other optimizations. There is a large scope of I/O optimizations that could be implemented under our design. For instance, it would be interesting to explore the impact of storage tiering policies under different datasets and models. Moreover, it would be interesting to use our approach to complement existing data pipeline frameworks, such as DALI and CoorDL, adding support for other I/O optimizations (*e.g.,* storage tiering) and system-wide visibility.

Distributed training settings. While we demonstrate the impact of SDS-enabled optimizations in a local setting, it would be interesting to explore their impact on large-scale DL deployments, that require tight coordination and holistic tunning of data plane stages.

Access coordination to shared datasets. Under shared storage infrastructures (*e.g.,* HPC centers), it is common to have multiple DL jobs (that are oblivious of each other) operating concurrently over the same dataset, leading to resource contention and performance variation [32], [37]. As such, it would be interesting to explore and introduce performance isolation and resource fairness policies to these deployments.

Control plane scalability and dependability. Either for controlling a large number of data plane stages (*e.g.,* hundreds to thousands of concurrent DL jobs) or enforcing storage policies under volatile environments, it is fundamental to investigate the control plane's scalability and dependability.

Availability

PRISMA user-level library, along with the TensorFlow and PyTorch integrations, are publicly available at https://github.com/dsrhaslab/prisma.

Acknowledgments

We thank the National Institute of Advanced Industrial Science and Technology for providing access to computational resources of AI Bridging Cloud Infrastructure (ABCI). This work was financed by National Funds through the Portuguese Foundation for Science and Technology (FCT) within project PAStor (UTA-EXPL/CA/0075/2019) and through PhD Fellowships SFRH/BD/146059/2019 (Ricardo Macedo) and SFRH/BD/146528/2019 (Cláudia Brito).

References

[1] A. Krizhevsky, G. Hinton *et al.*, "Learning Multiple Layers of Features from Tiny Images," Citeseer, Tech. Rep., 2009.

[2] Y. LeCun, C. Cortes, and C. J. Burges, "The MNIST Database of Handwritten Digits," 1999. [Online]. Available: http://yann.lecun.com/exdb/mnist/

[3] S. V. Lab, "ImageNet-22K," 2020. [Online]. Available: https://imagenet.stanford.edu

[4] S. Abu-El-Haija, N. Kothari, J. Lee, P. Natsev, G. Toderici, B. Varadarajan, and S. Vijayanarasimhan, "YouTube-8M: A Large-Scale Video Classification Benchmark," *CoRR*, vol. abs/1609.08675, 2016.

[5] "cvdfoundation/open-images-dataset: Open Images Dataset," 2017. [Online]. Available: https://github.com/cvdfoundation/open-images-dataset

[6] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers *et al.*, "In-Datacenter Performance Analysis of a Tensor Processing Unit," in *44th Annual International Symposium on Computer Architecture*. ACM, 2017, pp. 1–12.

[7] S. Markidis, S. W. Der Chien, E. Laure, I. B. Peng, and J. S. Vetter, "NVidia Tensor Core Programmability, Performance & Precision," in *2018 IEEE International Parallel and Distributed Processing Symposium Workshops*. IEEE, 2018, pp. 522–531.

[8] W. Xiao, R. Bhardwaj, R. Ramjee, M. Sivathanu, N. Kwatra, Z. Han, P. Patel, X. Peng, H. Zhao, Q. Zhang, F. Yang, and L. Zhou, "Gandiva: Introspective Cluster Scheduling for Deep Learning," in *13th USENIX Symposium on Operating Systems Design and Implementation*. USENIX Association, 2018, pp. 595–610.

[9] D. Narayanan, K. Santhanam, F. Kazhamiaka, A. Phanishayee, and M. Zaharia, "Heterogeneity-Aware Cluster Scheduling Policies for Deep Learning Workloads," in *14th USENIX Symposium on Operating Systems Design and Implementation*. USENIX Association, 2020, pp. 481–498.

[10] T. Chen, T. Moreau, Z. Jiang, L. Zheng, E. Yan, H. Shen, M. Cowan, L. Wang, Y. Hu, L. Ceze, C. Guestrin, and A. Krishnamurthy, "TVM: An Automated End-to-End Optimizing Compiler for Deep Learning," in *13th USENIX Symposium on Operating Systems Design and Implementation*. USENIX Association, 2018, pp. 578–594.

[11] Z. Jia, O. Padon, J. Thomas, T. Warszawski, M. Zaharia, and A. Aiken, "TASO: Optimizing Deep Learning Computation with Automatic Generation of Graph Substitutions," in *27th ACM Symposium on Operating Systems Principles*. ACM, 2019, pp. 47–62.

[12] N. Dryden, N. Maruyama, T. Moon, T. Benson, A. Yoo, M. Snir, and B. V. Essen, "Aluminum: An Asynchronous, GPU-Aware Communication Library Optimized for Large-Scale Training of Deep Neural Networks on HPC Systems," in *2018 IEEE/ACM Machine Learning in HPC Environments*. IEEE, 2018, pp. 1–13.

[13] D. Narayanan, A. Harlap, A. Phanishayee, V. Seshadri, N. R. Devanur, G. R. Ganger, P. B. Gibbons, and M. Zaharia, "PipeDream: Generalized Pipeline Parallelism for DNN Training," in *27th ACM Symposium on Operating Systems Principles*. ACM, 2019, pp. 1–15.

[14] A. Sapio, M. Canini, C.-Y. Ho, J. Nelson, P. Kalnis, C. Kim, A. Krishnamurthy, M. Moshref, D. Ports, and P. Richtarik, "Scaling Distributed Machine Learning with In-Network Aggregation," in *18th USENIX Symposium on Networked Systems Design and Implementation*. USENIX Association, 2021.

[15] S. Rajbhandari, J. Rasley, O. Ruwase, and Y. He, "ZeRO: Memory optimizations Toward Training Trillion Parameter Models," in *International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2020, pp. 1–16.

[16] L. Wang, J. Ye, Y. Zhao, W. Wu, A. Li, S. L. Song, Z. Xu, and T. Kraska, "Superneurons: Dynamic GPU Memory Management for Training Deep Neural Networks," in *23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. ACM, 2018, pp. 41–53.

[17] M. Rhu, N. Gimelshein, J. Clemons, A. Zulfiqar, and S. W. Keckler, "vDNN: Virtualized Deep Neural Networks for Scalable, Memory-Efficient Neural Network Design," in *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE, 2016, pp. 1–13.

[18] S. Pumma, M. Si, W.-C. Feng, and P. Balaji, "Scalable Deep Learning via I/O Analysis and Optimization," *ACM Transactions on Parallel Computing*, vol. 1, no. 1, pp. 1–34, 2019.

[19] J. Mohan, A. Phanishayee, A. Raniwala, and V. Chidambaram, "Analyzing and Mitigating Data Stalls in DNN Training," *Proceedings of the VLDB Endowment*, vol. 14, no. 5, pp. 771–784, 2021.

[20] R. Böhringer, N. Dryden, T. Ben-Nun, and T. Hoefler, "Clairvoyant Prefetching for Distributed Machine Learning I/O," *CoRR*, vol. abs/2101.08734, 2021.

[21] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard *et al.*, "TensorFlow: A System for Large-Scale Machine Learning," in *12th USENIX Symposium on Operating Systems Design and Implementation*. USENIX Association, 2016, pp. 265–283.

[22] A. Paszke, S. Gross, S. Chintala, G. Chanan, E. Yang, Z. DeVito, Z. Lin, A. Desmaison, L. Antiga, and A. Lerer, "Automatic Differentiation in PyTorch," 2017.

[23] S. Tokui, K. Oono, S. Hido, and J. Clayton, "Chainer: A Next-Generation Open Source Framework for Deep Learning," in *Proceedings of Workshop on Machine Learning Systems in the 29th Annual Conference on Neural Information Processing Systems*, vol. 5, 2015, pp. 1–6.

[24] "TensorFlow API: tf.data.Dataset," 2021. [Online]. Available: https://www.tensorflow.org/api_docs/python/tf/data/Dataset

[25] P. Contributors, "PyTorch Docs: torch.utils.data," 2019. [Online]. Available: https://pytorch.org/docs/stable/data.html

[26] A. V. Kumar and M. Sivathanu, "Quiver: An Informed Storage Cache for Deep Learning," in *18th USENIX Conference on File and Storage Technologies*. USENIX Association, 2020, pp. 283–296.

[27] Z. Zhang, L. Huang, U. Manor, L. Fang, G. Merlo, C. Michoski, J. Cazes, and N. Gaffney, "FanStore: Enabling Efficient and Scalable I/O for Distributed Deep Learning," *CoRR*, vol. abs/1809.10799, 2018.

[28] Y. Zhu, W. Yu, B. Jiao, K. Mohror, A. Moody, and F. Chowdhury, "Efficient User-Level Storage Disaggregation for Deep Learning," in *2019 IEEE International Conference on Cluster Computing*. IEEE, 2019, pp. 1–12.

[29] S. Nakandala, Y. Zhang, and A. Kumar, "Cerebro: A Data System for Optimized Deep Learning Model Selection," *Proceedings of the VLDB Endowment*, vol. 13, no. 12, p. 2159–2173, 2020.

[30] F. Isaila, J. Carretero, and R. B. Ross, "CLARISSE: A Middleware for Data-Staging Coordination and Control on Large-Scale HPC Platforms," in *IEEE/ACM 16th International Symposium on Cluster, Cloud and Grid Computing*. IEEE, 2016, pp. 346–355.

[31] S. Karki, B. Nguyen, and X. Zhang, "QoS Support for Scientific Workflows Using Software-Defined Storage Resource Enclaves," in *2018 IEEE International Parallel and Distributed Processing Symposium*. IEEE, 2018, pp. 95–104.

[32] G. K. Lockwood, S. Snyder, T. Wang, S. Byna, P. Carns, and N. J. Wright, "A Year in the Life of a Parallel File System," in *International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2018, pp. 931–943.

[33] S. Kim, H. Kim, J. Lee, and J. Jeong, "Enlightening the I/O Path: A Holistic Approach for Application Performance," in *15th USENIX Conference on File and Storage Technologies*. USENIX Association, 2017, pp. 345–358.

[34] H. Jo, S. hun Kim, S. Kim, J. Jeong, and J. Lee, "Request-aware Cooperative I/O Scheduling for Scale-out Database Applications," in *9th USENIX Workshop on Hot Topics in Storage and File Systems*. USENIX Association, 2017.

[35] T. Patel, Z. Liu, R. Kettimuthu, P. Rich, W. Allcock, and D. Tiwari, "Job Characteristics on Large-Scale Systems: Long-Term Analysis, Quantification, and Implications," in *International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2020, pp. 1186–1202.

[36] J. Lofstead, F. Zheng, Q. Liu, S. Klasky, R. Oldfield, T. Kordenbrock, K. Schwan, and M. Wolf, "Managing Variability in the IO Performance of Petascale Storage Systems," in *2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2010, pp. 1–12.

[37] O. Yildiz, M. Dorier, S. Ibrahim, R. Ross, and G. Antoniu, "On the Root Causes of Cross-Application I/O Interference in HPC Storage Systems," in *2016 IEEE International Parallel and Distributed Processing Symposium*. IEEE, 2016, pp. 750–759.

[38] A. Jokanovic, J. C. Sancho, G. Rodriguez, A. Lucero, C. Minkenberg, and J. Labarta, "Quiet Neighborhoods: Key to Protect Job Performance Predictability," in *2015 IEEE International Parallel and Distributed Processing Symposium*. IEEE, 2015, pp. 449–459.

[39] R. Macedo, J. Paulo, J. Pereira, and A. Bessani, "A Survey and Classification of Software-Defined Storage Systems," *ACM Computing Surveys*, vol. 53, no. 3, pp. 1–38, 2020.

[40] E. Thereska, H. Ballani, G. O'Shea, T. Karagiannis, A. Rowstron *et al.*, "IOFlow: A Software-Defined Storage Architecture," in *24th ACM Symposium on Operating Systems Principles*. ACM, 2013, pp. 182–196.

[41] N. Li, H. Jiang, D. Feng, and Z. Shi, "PSLO: Enforcing the X^{th} Percentile Latency and Throughput SLOs for Consolidated VM Storage," in *11th European Conference on Computer Systems*. ACM, 2016, pp. 28:1–28:14.

[42] R. Pontes, D. Burihabwa, F. Maia, J. Paulo, V. Schiavoni *et al.*, "SafeFS: A Modular Architecture for Secure User-space File Systems: One FUSE to Rule Them All," in *10th ACM International Systems and Storage Conference*. ACM, 2017, pp. 9:1–9:12.

[43] R. Gracia-Tinedo, J. Sampé, E. Zamora, M. Sánchez-Artigas, P. García-López *et al.*, "Crystal: Software-Defined Storage for Multi-tenant Object Stores," in *15th USENIX Conference on File and Storage Technologies*. USENIX Association, 2017, pp. 243–256.

[44] I. Stefanovici, B. Schroeder, G. O'Shea, and E. Thereska, "sRoute: Treating the Storage Stack Like a Network," in *14th USENIX Conference on File and Storage Technologies*. USENIX Association, 2016, pp. 197–212.

[45] J. Mace, P. Bodik, R. Fonseca, and M. Musuvathi, "Retro: Targeted Resource Management in Multi-tenant Distributed Systems," in *12th*

USENIX Symposium on Networked Systems Design and Implementation. USENIX Association, 2015, pp. 589–603.

[46] R. Caruana, S. Lawrence, and C. L. Giles, "Overfitting in Neural Nets: Backpropagation, Conjugate Gradient, and Early Stopping," in *Advances in Neural Information Processing Systems*, 2001, pp. 402–408.

[47] S. Sarkar, "A Scalable Artificial Intelligence Data Pipeline for Accelerating Time to Insight," 2019, SNIA Storage Developer Conference. [Online]. Available: https://www.snia.org/sites/default/files/SDC/2019/presentations/Machine_Learning/Sarkar_Sanhita_A_Scalable_Artificial_Intelligence_Data_Pipeline_for_Accelerating_Time_to_Insight.pdf

[48] "TensorFlow Auto-tuner: prefetch_autotuner.cc," 2017. [Online]. Available: https://github.com/tensorflow/tensorflow/blob/master/tensorflow/core/kernels/data/prefetch_autotuner.cc

[49] "TensorFlow Tutorial: TFRecord and tf.Example," 2021. [Online]. Available: https://www.tensorflow.org/tutorials/load_data/tfrecord

[50] Y. Zhu, F. Chowdhury, H. Fu, A. Moody, K. Mohror, K. Sato, and W. Yu, "Entropy-Aware I/O Pipelining for Large-Scale Deep Learning on HPC Systems," in *2018 IEEE 26th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems*. IEEE, 2018, pp. 145–156.

[51] P. Schwan, "Lustre: Building a File System for 1000-node Clusters," in *Proceedings of the 2003 Linux Symposium*, vol. 2003, 2003, pp. 380–386.

[52] F. B. Schmuck and R. L. Haskin, "GPFS: A Shared-Disk File System for Large Computing Clusters," in *1st USENIX Conference on File and Storage Technologies*. USENIX Association, 2002, pp. 231–244.

[53] F. Chowdhury, Y. Zhu, T. Heer, S. Paredes, A. Moody, R. Goldstone, K. Mohror, and W. Yu, "I/O Characterization and Performance Evaluation of BeeGFS for Deep Learning," in *48th International Conference on Parallel Processing*. ACM, 2019.

[54] "nvidia/dali: NVidia DALI," 2018. [Online]. Available: https://github.com/NVIDIA/DALI

[55] J. C. Doyle, B. A. Francis, and A. R. Tannenbaum, *Feedback Control Theory*. Courier Corporation, 2013.

[56] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein *et al.*, "Imagenet Large Scale Visual Recognition Challenge," *International Journal of Computer Vision*, vol. 115, no. 3, pp. 211–252, 2015.

[57] Y. LeCun, L. Bottou, Y. Bengio, P. Haffner *et al.*, "Gradient-based Learning Applied to Document Recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998.

[58] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet Classification with Deep Convolutional Neural Networks," in *Advances in Neural Information Processing Systems*, 2012, pp. 1097–1105.

[59] K. He, X. Zhang, S. Ren, and J. Sun, "Deep Residual Learning for Image Recognition," in *IEEE Conference on Computer Vision and Pattern Recognition*. IEEE, 2016, pp. 770–778.

[60] A. Wang, S. Venkataraman, S. Alspaugh, R. Katz, and I. Stoica, "Cake: Enabling High-level SLOs on Shared Storage Systems," in *3rd ACM Symposium on Cloud Computing*. ACM, 2012, pp. 14:1–14:14.

[61] R. Macedo, Y. Tanimura, J. Haga, V. Chidambaram, J. Pereira, and J. Paulo, "PAIO: A Software-Defined Storage Data Plane Framework," *CoRR*, vol. abs/2106.03617, 2021.

[62] K. Serizawa and O. Tatebe, "Accelerating Machine Learning I/O by Overlapping Data Staging and Mini-batch Generations," in *6th IEEE/ACM International Conference on Big Data Computing, Applications and Technologies*. ACM, 2019, pp. 31–34.

[63] R. Rew and G. Davis, "NetCDF: an interface for scientific data access," *IEEE Computer Graphics and Applications*, vol. 10, no. 4, pp. 76–82, 1990.

[64] M. Folk, G. Heber, Q. Koziol, E. Pourmal, and D. Robinson, "An Overview of the HDF5 Technology Suite and Its Applications," in *EDBT/ICDT 2011 Workshop on Array Databases*. ACM, 2011, p. 36–47.

[65] J. F. Lofstead, S. Klasky, K. Schwan, N. Podhorszki, and C. Jin, "Flexible IO and integration for scientific codes through the adaptable IO system (ADIOS)," in *6th International Workshop on Challenges of Large Applications in Distributed Environments*, 2008, pp. 15–24.

[66] W. F. Godoy, N. Podhorszki, R. Wang, C. Atkins, G. Eisenhauer, J. Gu, P. Davis, J. Choi, K. Germaschewski, K. Huck *et al.*, "ADIOS 2: The Adaptable Input Output System. A framework for high-performance data management," *SoftwareX*, vol. 12, 2020.

[67] R. Thakur, W. Gropp, and E. Lusk, "Data sieving and collective I/O in ROMIO," in *7th Symposium on the Frontiers of Massively Parallel Computation*. IEEE, 1999, pp. 182–189.