Efficient User-Level Storage Disaggregation for Deep Learning

Yue Zhu† Weikuan Yu† Bing Jiao† Kathryn Mohror‡ Adam Moody‡ Fahim Chowdhury†
†Florida State University ‡Lawrence Livermore National Laboratory
{yzhu, yuw, jiao, fchowdhu}@cs.fsu.edu {kathryn, moody20}@llnl.gov

***Abstract*—On large-scale high performance computing (HPC) systems, applications are provisioned with aggregated resources to meet their peak demands for brief periods. This results in resource underutilization because application requirements vary a lot during execution. This problem is particularly pronounced for deep learning applications that are running on leadership HPC systems with a large pool of burst buffers in the form of flash or non-volatile memory (NVM) devices. In this paper, we examine the I/O patterns of deep neural networks and reveal their critical need of loading many small samples randomly for successful training. We have designed a specialized Deep Learning File System (DLFS) that provides a thin set of APIs. Particularly, we design the metadata management of DLFS through an in-memory tree-based sample directory and its file services through the user-level SPDK protocol that can disaggregate the capabilities of NVM Express (NVMe) devices to parallel training tasks. Our experimental results show that DLFS can dramatically improve the throughput of training for deep neural networks on NVMe over Fabric, compared with the kernel-based Ext4 file system. Furthermore, DLFS achieves efficient user-level storage disaggregation with very little CPU utilization.**

I. Introduction

With the popularity of microprocessors and scale-out system architectures, many large-scale high performance computing (HPC) systems are built from a collection of compute servers, with an identical set of resources such as CPU, memory and storage. For improved performance, applications submitted to these systems are provisioned to meet their peak demands. Since the periods of peak demands can be short-lived, this can result in significant resource underutilization [21] when a diverse set of applications with varying requirements are running on a leadership computer system. Similar problems have been observed in large-scale data centers hosted by HPE [34], Intel [35], and Facebook [62]. Thus an emerging paradigm for resource provisioning called resource disaggregation is quickly gaining popularity. With resource aggregation, the resources are physcially separated from the compute servers and accessed remotely, which allows each resource technology to evolve independently and supports increased configurability of resources. In the context of HPC, several projects have provided mechanisms for disaggregating GPU accelerators [52], [50], [29].

As discussed above, resource provisioning can result in inefficient resource utilization. This is particularly challenging for storage on leaderhip class HPC systems that utilize non-volatile memory (NVM) devices, because the massive performance of NVM devices with respect to low latency and high bandwidth can be severely underutilized under heavy CPU and memory workloads [39]. This problem is particularly pronounced for deep learning (DL) applications that are running on large scale leadership systems with a large pool of burst buffers in the form of flash or non-volatile memory devices. In this environment, the flash or NVMe storage devices are capable of low-latency and high-bandwidth I/O services, but such capabilities are significantly hampered by the complex software stack for I/O processing in the kernel [60]. It is hard to make a one-size-fit-all decision for different resource demands.

In this paper, we explore the use of NVMe storage disaggregation for support of deep neural networks (DNNs), where we utilize a collection of local and/or remote storage devices to serve the DNN job for improved performance. Deep neural networks such as AlexNet have demonstrated capabilities for image classification, speech recognition and signal processing. To leverage the computation power of large systems, stochastic gradient descent (SGD) as a model optimizer usually requires loading a batch of training samples for every iteration. Such iterative loading of samples imposes nontrivial I/O pressure on storage systems [10], [3]. In addition, to avoid overfitting caused by predefined input patterns, each batch of sample files must be selected randomly. This presents another challenge to the conventional I/O services that are highly optimized for large sequential I/O.

Many studies have explored storage disaggregation to improve storage resource utilization on large-scale data centers [41], [58], [43], [51], [49]. Burst buffers with NVM and SSD devices have become an integral part of HPC leadership facilities. The NVMe over Fabrics [9] protocol supports the access of a remote NVMe SSD device, with only a few microseconds of added latency, blurring the difference between local and remote devices. However, to the best of our knowledge, there has been no effort to leverage user-level disaggregation for efficient I/O in deep learning applications on large-scale HPC systems.

In view of the performance impedance caused by the kernel-based file systems such as Ext4 to burst buffers, we have undertaken an effort to develop a specialized user-level, read-optimized file system for deep learning applications. Our

978-1-7281-4734-5/19/$31.00

solution – Deep Learning File System (DLFS) – is designed as a thin layer of file I/O services on top of an emerging industrial standard, NVMe over Fabrics. DLFS provides efficient and convenient I/O services for deep learning on HPC systems with storage disaggregation. Our experimental results show that DLFS can dramatically improve the throughput of training samples for deep neural networks on NVMe over Fabric, compared with the kernel-based file systems such as Ext4.

Specifically, we make the following contributions:

- We designed a specialized Deep Learning File System (DLFS) that supports user-level storage disaggregation and provides a thin set of APIs for deep learning applications.
- We designed fast metadata management in DLFS through an in-memory tree-based sample directory and its I/O services through the SPDK protocol. To the best of our knowledge, this is the first study on enabling DL applications through user-level disaggregation of NVMe devices.
- We introduced opportunistic optimizations that can batch the movement of data at both the sample level and the chunk level through cross-layer coordination.
- We conducted an extensive performance evaluation to validate our design of DLFS. Our results show that DLFS can achieve efficient user-level storage disaggregation with very little CPU utilization.

II. Background and Motivation

In this section, we review the background of resource disaggregation (§II-A), discuss I/O patterns in deep learning applications (§II-B), and then motivate storage disaggregation for deep learning (§II-C).

A. Storage Disaggregation

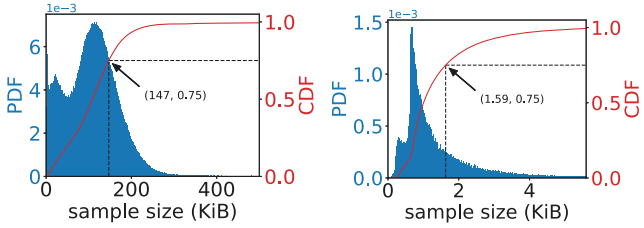
Compute servers with aggregated resources (CPU, Memory and storage) are the typical building blocks of warehouse-scale computers (WSC) and HPC centers. Such aggregation can result in inefficient resource utilization because it is difficult to determine an exact amount of resources on every server to meet different applications' unique behaviors [21]. Many efforts have been carried out recently on *resource disaggregation* in both academia [20], [40], [28] and industry [7], [8], [11]. The goal of disaggregation is to decouple different resources or disaggregate a large amount of a single resource so that fine-grained allocations can be made to meet the dynamic demands of applications at run-time.

The provisioning of storage resources can be particularly hard because the actual demand may be a complex combination of required capacity, throughput in terms of I/O operations or files per second, bandwidth (bytes per second), latency, etc. For example, leveraging PCIe based flash devices through a complex kernel-based I/O stack can incur various software overheads due to memory allocation, data movement, compression, scheduling, and network transmissions [39]. Such software overheads can saturate CPU cycles before the PCIe bus bandwidth, resulting in underutilization of flash storage. Baidu reported that their storage system can only achieve half of the raw device bandwidth [51]. Disaggregating storage resources can help amortize the total system building cost and maximize the performance-per-dollar of computing infrastructure.

Disaggregation via NVMe over Fabrics: Recently, NVMe SSD devices have been popular targets of storage disaggregation for their powerful bandwidth and throughput capacity. A number of research studies [42], [22], [69] have worked to support storage disaggregation for NVMe SSD devices. NVMe over Fabrics [9] is an emerging industrial standard supporting disaggregation. It mounts a remote NVMe SSD device locally and allows NVMe devices to be accessible in a fabric within 10 μsec. However, the original NVMe over Fabrics solution in the Linux kernel 4.8 or higher only supports an end-to-end connection between two nodes, which limits the data sharing between multiple nodes. A recent protocol developed by Intel called SPDK (Storage Performance Development Kit) [71] provides a user-level solution for NVMe over Fabrics [13], [14]. We refer to the SPDK-based user-level NVMe over Fabrics target as *NVMe-oF Target*. An NVMe-oF Target allows data on an NVMe SSD device to be directly accessible to all connected remote clients through remote direct memory access (RDMA) [48]. Through the use of RDMA in the user-level SPDK protocol, NVMe-oF clients and targets can perform zero-copy data transfers in an OS-bypass manner.

B. I/O in Deep Neural Networks

Gradient descent is one of the most popular algorithms to optimize parameters of the DNNs. Mini-batched gradient descent, often referred to as SGD [1], is commonly used because of its fast training speed and low memory requirements. However, there are a number of challenges for the effective utilization of storage resources on large-scale HPC systems.



(a) ImageNet dataset (b) IMDB movie review dataset

Fig. 1: Sample size distribution for different dataset.

Increased mini-batch size: The popularity of SGD has led to many recent optimizations to leverage increasing compute power. For example, it is important to batch more input samples, i.e., increasing the mini-batch size, in each training iteration. Many studies[33], [30], [57], [19], [37], [47], [70] have reported that they can finish the training of ResNet-50 [33] with ImageNet [25] at fast speeds. In these studies, the batch size has increased from 256 [33] to 80K [70] over the past few years. Compared to the conventional demands on high bandwidth and low latency, the increased mini-batch size

requires much higher throughput so that more samples can be trained in each iteration.

Many small random samples: In deep learning training, a large number of training samples are expected to arrive in a random order to speed up the convergence speed and reduce the training biases caused by fixed input sequences. Many popular datasets contain small samples. As shown in Fig. 1, the ImageNet dataset consists of many small samples (every raw image file is an individual sample), about 75% of samples are less than 147 KB. A similar trend is also shown in [23]. In the case of the IMDB dataset, 75% of samples are less than 1.6 KB. Working with these datasets result in many random reads to the storage system. This is a challenging I/O pattern because it cannot benefit from the traditional storage systems (e.g., a parallel file system) that are typically designed for large sequential I/O patterns. Although we can preprocess small samples into large batched files (e.g., TFRecord format and CIFAR10 format) to avoid small random I/O, the existing sample shuffling method cannot support global data shuffling, and the size of shuffle buffer limits the shuffling result. For example, when using TFRecord files in TensorFlow, every TFRecord file is sequentially read in a fixed size shuffle buffer. However, if the size of the shuffle buffer is not large enough, the learner only obtains partially shuffled samples, which reduces the training accuracy.

C. Storage Disaggregation for Deep Learning

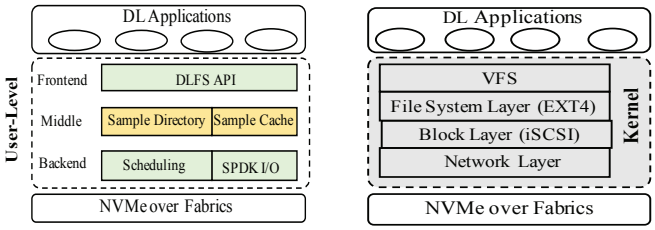
In addition to the challenges discussed above, the dynamic workload exhibited by deep learning training leads to underutilization of traditional storage resources on HPC systems. For instance, (1) there are not enough CPU cycles for storage processing in computation-intensive DNNs; (2) due to accuracy concerns, the storage capacity and the performance of on-node storage devices on the nodes allocated to the training job are not sufficient to store an entire dataset and match the training speed, respectively. To address these challenges, we developed a user level method to disaggregate storage devices that matches the needs of training applications.

There have been storage disaggregation solutions to improve storage resource utilization on large-scale data centers [41], [58], [43], [51], [49]. These proposed solutions aim to solve the problem at the system level. To the best of our knowledge, there has been no effort to leverage user-level disaggregation for efficient I/O in deep learning applications on large-scale HPC systems at the user level. The SPDK-based NVMe over Fabrics protocol can address the need of high throughput and solve the challenges imposed by many small samples. Thus we propose to leverage the SPDK protocol for storage disaggregation of NVMe devices and develop a special user-level read-optimized file system for deep learning applications.

III. Design and Implementation of DLFS

Today, large-scale leadership computing facilities are equipped with a pool of burst buffers composed of SSD devices. On these systems, DL applications typically load the training datasets into the burst buffers at the beginning of their execution from the persistent file system. The overarching goal of DLFS is to provide a temporary and efficient substrate for DL applications to buffer and exchange the samples across the NVMe SSD devices for efficient training.

While using DLFS, a set of NVMe devices is allocated to DLFS in a job. Our design is flexible such that the allocated NVMe devices may be local or remote with respect to the compute nodes in the job. During initialization, DLFS builds the connection between allocated NVMe devices, uploads the training dataset from backend persistent file system to the NVMe devices, and creates the in-memory sample directory based on the uploaded datasets. When reading samples, DLFS locates data through the in-memory sample directory and directly accesses local/remote data via the SPDK protocol in user-space. To support high throughput and low latency for data access services, we have designed DLFS with three layers, as shown in Fig. 2. The first *frontend* layer provides a small set of API functions. This layer directly interacts with DL applications to receive requests for services and deliver metadata or data as a response. The *middle* layer hosts an in-memory sample directory for metadata and a sample cache for data retrieved from local or remote storage targets. The *backend* layer is equipped with a complex set of data structures created for queuing and scheduling the requests and then dispatching them to the NVMe devices. This backend layer leverages the SPDK protocol to access both local or remote storage targets in an OS-bypass manner.



(a) DLFS I/O Stack (b) Conventional I/O Stack

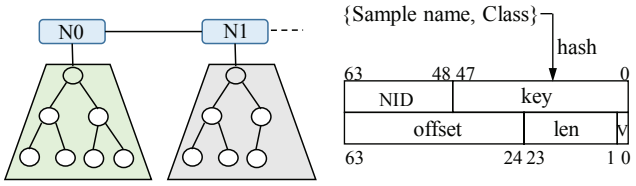
Fig. 2: Architecture Comparison between DLFS and Generic File System

To shed light on the benefits of DLFS, we provide a comparison with a conventional I/O stack that accesses remote network-attached storage. As shown to the right of Fig. 2(b), file I/O requests have to travel through the user-kernel boundary to reach the generic VFS layer, the actual file system layer for indexing and caching, the block layer such as iSCSI, and the network layer before reaching remote storage targets. Along this deep kernel-based stack, multiple context switches and data copies are incurred for the I/O requests. In contrast, DLFS serves storage resources to many clients in a disaggregated manner without the involvement of the kernel.

A. DLFS API

To facilitate the integration of disaggregated storage to various applications, we offer a set of functions for using DLFS.

dlfs_mount: This API function takes a parameter specifying the dataset(s) on the HPC parallel file system. Each process



(a) Partitioned Directory (b) Sample Entry

Fig. 3: Design of In-Memory Sample Directory

then loads its portion of the files to its local NVMe device. This is similar to the convention of creating a file system before I/O services. However, in our case, the mount call is a collective call from all processes in a DL application. It initializes an in-memory sample directory that tracks the location of all samples (§III-B). A DLFS instance, particularly its sample directory, is alive only during the lifetime of an application.

dlfs_open, dlfs_read, dlfs_close: Similar to the POSIX APIs, we enable these conventional APIs for deep learning. All *open*, *read*, *close* functions called on the training datasets are redirected towards its DLFS counterpart. In turn, these DLFS functions locate the targeted file through the directory initialized by *dlfs_mount* and access the data via the SPDK protocol (§III-C).

dlfs_sequence and dlfs_bread: To cater to the use of mini-batches in SGD neural networks, we add two APIs, *dlfs_sequence* and *dlfs_bread*, in our DLFS implementation for efficient loading random samples. The former specifies a random seed for DLFS to generate a global sample sequence list, and the latter reads data for a mini-batch. These functions are supported through our optimizations in §III-D.

B. In-Memory Tree-Based Sample Directory

In a conventional file system, a file I/O request starts with a lookup operation to retrieve metadata and validate the access rights before the actual I/O. Metadata management is a complex problem for distributed file systems. This is also true for file systems across distributed burst buffers [56], [65], [73]. While previous solutions for distributed burst buffers have been designed for write-intensive I/O patterns such as checkpoint and restart from scientific applications, DLFS is designed to read many batches of small samples in gigantic DL datasets. We will elaborate on the directory organization and its construction.

1) Organization of a Sample Entry and the Directory Tree: In mini-batch based DNNs, each batch is formed by random samples from the training dataset. As mentioned in §II-B, many random small samples present a challenging I/O pattern to the burst buffers and parallel file system on HPC systems. To speed up the lookup time for mini-batch creation, we create an in-memory tree-based sample directory during the initialization of DLFS.

As shown in Fig. 3(a), the entire directory is partitioned into an array of balanced AVL trees, according to the file name and the number of storage nodes. Each node creates an AVL tree to hold the sample entries on the node, each with the actual location of a sample. Allowing the index management at sample-level helps us imposing full randomization with opportunistic batching (§III-D) over different dataset formats. For instance, we are able to have direct access to any samples in a TFRecord file. Note that there is also an entry taking by the batched file for file-oriented access.

Fig. 3(b) illustrates the composition of a sample entry. Each entry is composed of 128 bits, divided into two 64-bit units. The first unit contains a 16-bit Node ID (NID) and a 48-bit key. The key is generated by hash value of a file/sample name and other attributes such as its class. The second unit contains a 40-bit offset field and a 23-bit length (len) field. These two tuples together provide the actual location of a sample on an NVMe device. Furthermore, the second unit of a sample entry provides a V field of one bit, which is used to track the presence of a data copy in local sample cache. This partitioned directory draws its inspirations from the directory protocols [24], [27] for cache coherence on distributed memory systems. An important distinction is that the training files in DL applications are read-only. Thus there is no need to track any changes made by parallel tasks across different NVMe devices. Together, this tree-based sample directory not only maps the input data to their location in distributed NVMe devices but also tracks the presence of data in the local sample cache.

2) Construction and Aggregation of Sample Directory: During the initialization, DLFS is designed to partition and load the entire training set into distributed burst buffers. All nodes in a DLFS instance will load their share of files into the local NVMe device(s) from an HPC persistent file system. This is realized as part of the functionalities in the *dlfs_mount* call (§III-A). Accordingly, each node creates an entry for each sample according to the format described above and populates its AVL tree.

After the construction of their local AVL tree, all nodes then invoke a collective communication to gather all AVL trees, forming an identical copy of in-memory sample directory at every node. This distributed generation of AVL trees speeds up the creation of the in-memory sample directory. With a complete directory in memory for all samples in the training dataset, DLFS prevents any bottleneck on a centralized metadata store and avoids cross-node communication for sample lookup. Furthermore, this in-memory sample directory enables local metadata retrieval for the NVMe clients and relieves the NVMe-oF Targets of any metadata pressure from many parallel training processes on large-scale HPC systems.

Memory Consumption: A legitimate concern is the memory consumption at each compute node. We provide a quick calculation to show its negligible impact. For a training dataset with 50 million samples, since we only need 16 bytes for every sample entry, the memory consumption is only 0.8 GB (50×16 MB), for the entire sample directory. For contemporary HPC systems with more than 100 GB per compute node such as Sierra [12] (256 GB/node) and Summit [15] (512 GB/node), this memory consumption is much less than 1% and quite affordable for the overall performance of DL

applications.

C. SPDK-Based User-Level Disaggregation

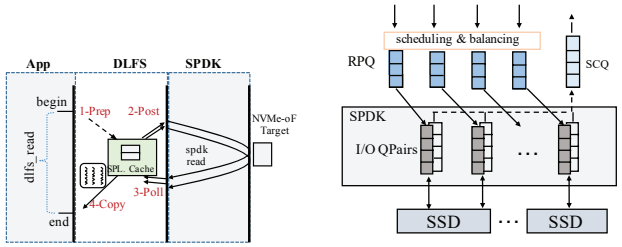
The SPDK protocol for NVMe over Fabrics offers a user-level direct I/O technique without going through the kernel. In addition, it supports disaggregation of storage resources to both local and remote clients through multiple concurrent I/O queue pairs. However, the SPDK protocol does impose a couple of restrictions because of its limitation on the accessible memory areas, and its queuing and polling semantics. DLFS must leverage SPDK powerful features and mitigate its limitations for efficient disaggregation. We first describe our design of user-level direct I/O and then balanced scheduling and completion queues in DLFS.

1) DLFS User-Level I/O Operations on SPDK: With the assistance of RDMA and the SPDK user-space driver, the SPDK NVMe-oF Target offers the capability of accessing remote SSD as if it were local. However, an NVMe device has to be unbound from the kernel before direct memory-mapped I/O operations are allowed for the current process [14]. In addition, the SPDK protocol mandates that all I/O requests have their memory allocated on huge pages, which typically fall outside of the memory area with application data.

We allocate the sample cache on huge pages to store the data read from local/remote NVMe devices. This allows DLFS to perform zero-copy data transfer between its sample cache and the NVMe devices. For flexible management, the cache is divided into many fixed-size chunks (256 KB by default but configurable). A read request is associated with one or multiple chunks. Upon the completion of reads, data are available at specified chunks in the sample cache. We also create a pool of copy threads to assist the data copying from the sample cache to the general application buffers.

As shown in Fig. 4(a), in a DLFS read operation, we have four stages: *prep*, *post*, *poll*, and *copy*. As mentioned earlier, each sample entry tracks the presence of a local copy in the sample cache. When a *dlfs_read* operation is received at the DLFS, we first check the sample entry and return the data if the V field is on. If a local copy is not present, we prepare and schedule this read operation to the targeted NVMe device in the *prep* stage. By default, each sample will be converted into one SPDK request and be allocated with one data chunk to receive data. A data request larger than a chunk will be disassembled into multiple requests and allocated with more chunks. In the *post* stage, each SPDK request will be posted to the SPDK *I/O queue pair* (I/O QPair), which will be delivered to the correct NVMe-oF target. In the third *poll* stage, the completion of a SPDK request will be detected by polling the completion queue (CQ) associated with the I/O QPair. At the completion of all outstanding chunks for the current request, we set the V field in its sample entry to mark the presence of a local copy. In the final *copy* stage, we copy the data in the cache to the destination memory in the application.

2) Balanced Scheduling and Completion Queues: The SPDK protocol supports lightweight concurrent I/O queue



(a) Flow of a DLFS Request (b) Management Queues

Fig. 4: Design of User-Level I/O Disaggregation in DLFS

pairs (QPair) with a predefined queue depth, i.e., the threshold of outstanding SPDK commands per QPair. Each QPair consists of a submission queue for posting block requests to the NVMe target and a completion queue for the notification of request completion. To allow the capabilities of an NVMe device to be fully disaggregated, we allocate a request posting queue (RPQ) in DLFS for every NVMe SSD device. Each RPQ is associated with a submission queue of an I/O QPair as shown in Fig. 4(b). When a read request is received from a RPQ, DLFS analyzes and prepares one or more SPDK requests to the targeted NVMe device accordingly.

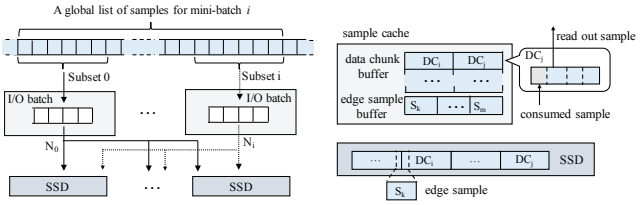
In contrast, we allocate a shared completion queue (SCQ) in DLFS to poll the completion queues of all I/O QPairs. Since SPDK currently supports only busy polling, an SCQ helps balance the progress management across all QPairs and consolidate the amount of computation required.

When a SPDK request is completed, we add it to the SCQ in DLFS from SPDK completion queue for moving arrived data on sample cache to application buffer. To deal with this CPU-intensive movement, we use the pool of copy threads to process all completed requests in the SCQ. The copy threads are shown in Fig. 4(a). Even though each completed request may carry a different amount of data, a shared queue helps balance the workload distribution to all copying threads. Note that, due to the additional copy step, the current implementation of data transfers in DLFS is not zero-copy from the NVMe target to the application. True zero-copy transfers would require the application buffers to be mapped on the huge pages, which we plan to investigate in future studies.

D. Opportunistic Batching Optimizations

In the basic design of DLFS I/O flow (§III-C) an application has to synchronously wait for the completion of its *dlfs_read* operation. We have noticed that the basic design cannot fully exploit the bandwidth and throughput potential of NVMe devices. Therefore, we introduce a couple of opportunistic optimizations that can batch more data transfers together through the coordination of frontend and backend layers inside DLFS. Accordingly, two functions, *dlfs_sequence* and *dlfs_bread*, are introduced in the DLFS API to allow a set of samples to be read together, and mitigate the detrimental performance impact caused by individual synchronous *dlfs_read* operations.

1) Frontend Sample-Level Batching: As mentioned in §II-B, in SGD DNNs, the input data are grouped in batches and trained in iterations. Each batch is divided among the training tasks. Since the accuracy of DL applications



(a) Frontend sample-level batching (b) Backend chunk-level batching

Fig. 5: Design of Opportunistic Batching

is improved with a fully randomized sequence of training samples, it does not matter how and where the sample sequence is generated as long as the sequence does not lead to an accuracy degradation. Thus we grant DLFS the liberty to determine the sample sequence for each iteration.

In sample-level batching, the frontend layer of DLFS determines the sample sequences in mini-batches. To ensure consistency across all training tasks, we use the same seed to generate a global random sample sequence when the frontend layer receives a *dlfs_sequence* call as shown in Fig. 5(a). This reduces the inter-node overhead for synchronization and communication for an agreement on the global list. While accessing samples, every node only reads its assigned portion on the list for current mini-batch. As also shown in Fig. 5(a), an individual node, N_0 or N_i, only reads its own subset of sample from the global list. With multiple samples batched for reading, the DLFS frontend can then submit as many requests as allowed by the queue depth of SPDK I/O QPairs. Such sample-level batching can maintain a high utilization of the NVMe devices, thereby avoiding the performance impact of synchronous *dlfs_read* operations.

2) Backend Chunk-Level Batching: While sample-level batching can overlap the transfer of many samples, the processing overhead is largely determined by the number of samples. If the sample size happens to be small, the processing overhead can dominate over the cost of transferring.

To optimize the performance of small samples, we introduce backend chunk-level batching by aggregating the small sample in to a data chunk (DC) as shown in Fig. 5(b). In this optimization, DLFS still determines the sample access sequence. Different from the sample-level batching, the backend layer aggregates small samples and accesses them in large chunks.

For chunk-level batching, we divide a dataset into multiple fixed-size data chunks during DLFS initialization. Every data chunk contains multiple samples. There are also some samples crossing the boundary of data chunks. We refer those samples to as *edge sample*s. For example, S_k in Fig. 5(b) crosses the boundary between DC_i and the chunk before DC_i. DLFS also constructs a data chunk access list and an edge sample access list to assist the sample lookup and access. The data chunk access list holds the chunk IDs and the corresponding key of the first complete sample in the chunks.

Upon a *dlfs_bread* call, the backend layer in DLFS fetches data chunks and edge samples into the sample cache according to the two lists. The copy threads then select samples randomly from the sample cache to application buffer. As shown in Fig. 5(b), a copy thread randomly picks DC_j in the sample cache and reads the first valid sample inside DC_j to the application buffer. The next valid sample inside DC_j then becomes the candidate for the next selection. Similarly, reading an edge sample will leave the next edge sample to be a candidate. DLFS repeats this selection until a sufficient number of samples have been identified for a *dlfs_bread* call. The copy threads at the frontend layer then copy the sample content to the application buffer.

The key strength of chunk-level batching is to avoid frequent SPDK read requests for many small samples. Depending on how many times the chunk size (256 KB by default) is greater than the average sample size, conceptually it can reduce the processing overhead by the same amount. While randomness is important for DL applications, there have been earlier efforts in adjusting the randomization of input samples for performance purposes [17], [2], [75]. We also emphasize the efficacy of randomization in our batching optimizations. We have evaluated our training accuracy in Section IV-E.

IV. Experimental Evaluation

As the SPDK library requires root privileges for configuration, all our experiments are conducted on an in-house cluster. Each node is equipped with 10 dual-socket Intel Xeon(R) CPU E5-2650 cores and 64 GB memory, and connected through FDR InfiniBand switch via ConnectX-3 adaptors. We have one 480 GB Intel Optane NVMe SSD device on one of the nodes. We compare the performance of DLFS against the Ext4 file system and Octopus [45].

Ext4: Ext4 is a kernel-based local file system. It is commonly used over SSDs to store datasets for compute nodes in parallel training [30], [37]. We use it in both single node and multi-node tests. In single node tests, we mount Ext4 over an NVMe SSD device. Due to limited NVMe SSD devices on our system, we can only run real tests on one SSD device. For multi-node tests, we leverage RAMdisk to emulate NVMe SSD devices by adding a delay when accessing the data, similar to other studies [45], [36], [68].

Octopus: Octopus is an RDMA-enabled distributed persistent memory file system. Although there are a few burst buffer file systems (§V), to best of our knowledge, they are not open-sourced or do not support NVMe over Fabrics for data transfer. Also, due to limited NVMe devices on our system, Crail[59], a storage middleware for resource disaggregation, is not able to be directly configured for testing. Since Octopus has proven its capability over Crail, we use Octopus with memory emulating backend NVMe devices as a comparison target. In the emulation, we add a delay when reading data from remote memory via RDMA, which is similar to the Ext4 test case. In addition, we do not include Octopus in single node tests since it is designed for distributed systems.

DLFS: We conduct two groups of tests, single-node and multi-node tests, to evaluate DLFS. The single-node tests are on the node with an Intel NVMe SSD device. We run multi-node tests from 2 to 16 nodes through memory-based NVMe-oF Targets. In memory-based NVMe-oF Targets, we introduce delays when accessing data to emulate using NVMe devices.

To measure the read performance of Octopus, Ext4, and DLFS, we use a dummy dataset with random values as the sample content. As random reads are the most common data read pattern in deep learning applications, we introduce our micro-benchmarks for measuring the sample throughput. When not specified otherwise, the number of batched samples is 32, and all our results are reported as an average from five measurements.

A. Performance of Local NVMe Device

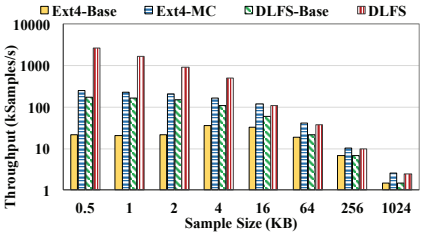
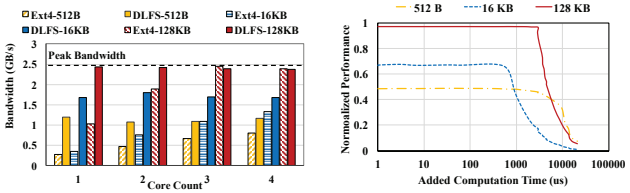


Fig. 6: Random read sample throughput on single node

1) Sample Throughput with Different Sample Sizes: We first measure the throughput of Ext4 and DLFS for random samples access on a real NVMe device. Octopus is not included for its requirement on multiple NVMe devices. In our experiments, we vary the sample size from 512 B to 1 MB. As shown in Fig. 6, for Ext4 tests, we denote the baseline case (one thread on one core) as *Ext4-Base* and the optimized case (multiple threads with multiple cores) as *Ext4-MC*. We also denote the baseline performance of DLFS as *DLFS-Base* and the performance with batching optimizations as *DLFS*. For small sample sizes (i.e., less than 16 KB), Ext4-Base is much lower than DLFS-Base and DLFS. DLFS-Base outperforms Ext4-base by at least 1.82× when the sample size is less than or equal to 4 KB. Although Ext4-MC outperforms DLFS-Base, it requires the use of more threads and more CPU cores. Nonetheless, its performance is 3.35× lower than DLFS for small sample sizes. The performance improvement of DLFS comes from our techniques such as fast metadata management, SPDK-based direct I/O, and opportunistic batching optimizations. Unlike the metadata management scheme in Ext4, our in-memory sample directory enables efficient metadata management. The SPDK-based direct I/O bypasses OS kernel and avoids the associated overhead of context switches and multiple memory copies. Our cross-layer opportunistic batching overlaps the transfer of multiple small samples.

When the sample size increases to more than 16 KB, the performance of Ext4-Base comes close to that of DLFS-Base. However, the sample throughput of Ext4-Base is still 43.8% lower than DLFS.

2) CPU Utilization: We also measure the impact of CPU resources in DLFS and Ext4. We have used a varying number of cores with one thread per core to read samples. Fig. 7(a) shows the total bandwidth with different sample sizes and core counts. DLFS saturates the peak NVMe bandwidth for all sample sizes with as few as only one core. In contrast, Ext4 needs three or more cores to reach the peak bandwidth. These results suggest that DL applications can benefit much more



(a) Core count to saturate SSD bandwidth

(b) CPU intensity

Fig. 7: DLFS CPU Utilization

from DLFS for small samples. In addition, using more I/O threads on multiple cores may add to the contention, which leads to a slight performance drop both in Ext4 and DLFS at high core counts. Since DLFS requires very little CPU cycles, contention from more cores can interfere more on the peak performance of DLFS compared to Ext4.

In light of the reduced requirements from DLFS on CPU cores, we have also devised a test to measure the potential overlap of I/O and computation. Note that the current SPDK protocol mandates the use of busy polling for progress checking and completion notification of I/O requests. Thus we measure the amount of CPU computation that can be added to the polling loop without affecting the performance of DLFS. We also convert the CPU cycles to time. Several different sample sizes are used in this test. As shown in Fig. 7(b), for samples of 128 KB, the performance of DLS stays unaffected until the added computation time is close to 2 ms. Since the DLFS I/O thread polls the completion for a batch of samples, this means that DLFS on SPDK can overlap the progress of 32 128KB samples with up to 2 ms of concurrent computation. As shown by the performance curve of 16 KB samples, for smaller samples, the amount of computation that can be overlapped reduces because of their fast completion. For very small samples such as 512 B, the amount of overlapped computation is similar to that of 128 KB samples, due to the batching optimizations in DLFS for small samples. The actual I/O requests we issue to the SPDK's I/O QPair are mostly the chunk size, which is 256 KB by default. Thus it allows more computation and I/O overlap.

B. Throughput of Networked NVMe Devices

We measure the sample throughput with DLFS, Octopus, and Ext4. DLFS is designed for loading data to NVMe SSD based burst buffers on large-scale systems. Ext4 is commonly used when DL training applications read datasets from local file systems on top of burst buffers. DLFS and Octopus can read data from remote NVMe devices, but Ext4 reads data locally. Due to the performance difference between DLFS-base and DLFS showed in Fig. 6, we do not include DLFS-base in the rest of the section. Besides, according to our results in §IV-A2 and the fact that DL applications are computation-intensive, we use only a single core for issuing I/O requests in DLFS and Ext4 in this test and the rest of performance evaluation.

1) Sample Throughput with Different Sample Sizes: We run the throughput tests with different sample sizes over 16 nodes, each with one emulated NVMe SSD device. Fig. 8 shows the

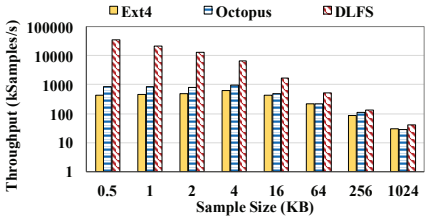
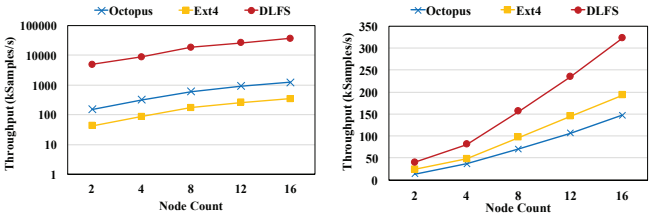


Fig. 8: Aggregated read throughput over 16 nodes

aggregated throughput for reading random samples. Clearly, DLFS outperforms Octopus and Ext4 in all cases.

For small samples ($\leq 4KB$), the throughput of DLFS is 9.72× and 6.05× higher than that of Ext4 and Octopus, respectively. For samples greater than or equal to 16 KB, DLFS outperforms Ext4 and Octopus, respectively, by 1.31× and 1.12× on average. The performance advantages of DLFS demonstrate that our techniques such as fast metadata management, user-level direct I/O and opportunistic batching optimizations are effective. Compared to DLFS, Octopus is designed as a general file system and leverages RDMA, but it does not offer any optimizations specialized for random small samples. Its use of RDMA seems to help reduce the number of memory copies, thus it outperforms Ext4 for small samples. But compared to DLFS, Octopus suffers from frequent inter-node communication for sample lookup. In the case of Ext4, its kernel implementation causes additional memory copies across different layers of I/O stack and frequent context switches for small samples.



(a) Sample Size = 512 Byte (b) Sample Size = 128 KB

Fig. 9: Aggregated throughput on networked NVMe devices

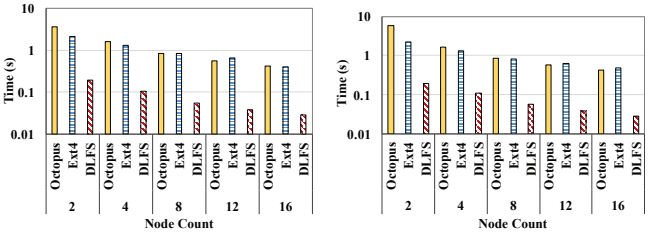
2) Scalability: We measure the scalability of DLFS across a varying number of nodes, each with an emulated NVMe device, and compare its performance to Ext4 and Octopus. Fig. 9 shows the aggregated sample throughput across 2 to 16 NVMe SSD devices. We use 128 KB and 512 B as the representative sample sizes.

With very small samples of 512 B, DLFS delivers the best performance among the three cases as shown in Fig. 9(a). Its sample throughput is 28.45× higher than that of Ext4 and 104.38× higher than that of Octopus, on average. In addition, we can observe a linearly increasing curve for DLFS with an increasing number of NVMe devices.

Based on the throughput measurement in Fig. 9(b), DLFS can outperform Ext4 by 65.1% on average while leveraging the disaggregated throughput from multiple NVMe devices. The throughput of Octopus is about 1.37× lower than that of DLFS. This is mostly due to its distributed metadata management and its lack of specialized optimizations for random small samples. As shown in Fig.9(a), we also achieve near-linear scalability for small samples with an increasing number of NVMe devices.

C. Sample Lookup Time

We further measure the sample lookup time of DLFS to evaluate the performance of our in-memory tree-based metadata management.



(a) Sample Size = 512 B (b) Sample Size = 128 KB

Fig. 10: Sample lookup time of DLFS on NVMe devices.

We conduct a test to evaluate the performance of metadata management for 1 million samples on a different number of nodes. We have chosen 512 B and 128 KB as the representative sample sizes. For DLFS and Octopus, we measure the sample lookup time directly due to their user-level implementations. In the case of Ext4, we measure the file open time as an equivalent of the lookup time to evaluate its metadata management. Fig. 10(a) shows the sample lookup time for the 512 B sample test. Due to its complex inode and block management, the sample lookup time of Ext4 is higher than DLFS by two orders of magnitude. Therefore, Octopus has the longest sample lookup time among three as shown in Fig. 10(a).

Moreover, only DLFS's sample lookup time decreases linearly with the increasing number of nodes, since every participating node reads fewer amount samples. However, with an increasing number of nodes, Octopus' and Ext4's sample lookup times do not decrease linearly. As mentioned previously, Octopus is limited by its cross node communication, and Ext4 is restricted by its complex inode and block management. Both of them cannot fully benefit from the parallelism when having more nodes. We also observe the same trend when testing with 128 KB samples shown in Fig. 10(b) These results demonstrate that our in-memory tree-based metadata management delivers the best sample lookup time. Note that, based on our measurements, the lookup time for 128-KB samples in DLFS takes only 1% of the sample reading time.

D. Effectiveness in Utilizing Disaggregated NVMe Devices

We have devised two tests to evaluate the effectiveness of DLFS in utilizing the storage capabilities from disaggregated NVMe devices. In the first test, we measure the sample throughput of DLFS running on one compute node while accessing storage from an increasing number of disaggregated NVMe devices. Its performance is shown by the curve *DLFS-1C* in Fig. 11. In the second test, we measure the sample throughput from 16 DLFS clients when accessing storage

from disaggregated NVMe devices. This case is similar to the scalability test shown in Fig. 9(b). In both tests, the sample size is 128 KB. The performance is shown by the curve *DLFS-16C* in Fig. 11. We include it here to analyze the effectiveness of storage access from many DLFS clients to an increasing number of disaggregated NVMe devices.

For a comparative analysis, we calculate the ideal sample throughput of an increasing number of NVMe devices with one or 16 DLFS clients. The results are included in Fig. 11 as two additional curves: *NVMe-1C* and *NVMe-16C*. The throughput of NVMe-1C is calculated as the quotient of the total NVMe bandwidth from all devices divided by the sample size, when the number of NVMe devices is ≤ 2. For more than two NVMe devices, the network bandwidth to the single client is smaller than the aggregated device bandwidth and becomes the bottleneck. Thus we use the network bandwidth to calculate the ideal throughput for more than 2 NVMe devices. With 16 clients, the network bandwidth is no longer the bottleneck. So the throughput numbers for NVMe-16C are directly calculated based on the aggregated bandwidth from the actual number of NVMe devices.

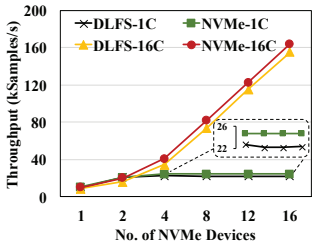


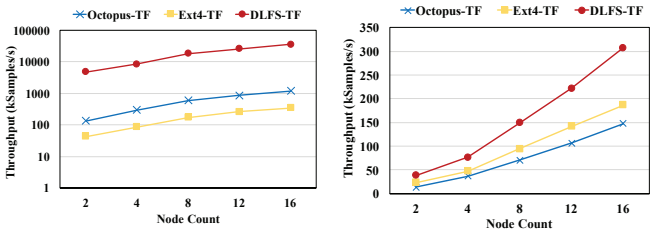
Fig. 11: Effective throughput on disaggregated NVMe Devices

As shown in Fig. 11, compared to NVMe, one DLFS client is very effective to leverage the capabilities of NVMe devices. Despite the bottleneck imposed by a single client's network bandwidth, DLFS achieves 93.4% of the ideal throughput allowed by a pool of disaggregated NVMe devices. On the other hand, with 16 DLFS clients, the sample throughput of DLFS increases linearly with an increasing number of NVMe devices. These DLFS clients can achieve up to 88% of the sample throughput that is theoretically possible from a pool of disaggregated NVMe devices. Taken together, these results demonstrate that DLFS clients can leverage storage resources effectively from a pool of disaggregated NVMe devices.

E. Performance of DLFS for DNNs

To evaluate the performance of DLFS for DNNs, we have enabled TensorFlow on top of DLFS, Octopus and Ext4 by designing a customized TensorFlow API [16]. A similar API can also be deployed to other training DNN training frameworks based on their dataset importing features. In our test, we then train AlexNet with the ImageNet dataset, and compare the validation accuracy between fully randomized sequences and the DLFS determined sequences.

1) Sample Throughput with TensorFlow: Fig. 12 shows the data importing performance of TensorFlow on top of Octopus, Ext4, and DLFS across a varying number of compute



(a) Sample Size = 512 B (b) Sample Size = 128 KB

Fig. 12: Aggregated throughput for TensorFlow on top of DLFS

nodes (referred to as Ext4-TF, Octopus-TF, and DLFS-TF). Similar to the tests in Section IV-B2, we use two representative sample sizes: 512 B and 128 KB. The performance trend in Fig. 12 is similar to the trend in Fig. 9. DLFS-TF outperforms Octopus-TF and Ext4-TF by 29.93× and 102.07× on average for 512-B samples as shown in Fig. 12(a). Fig. 12(b) shows the data importing performance of TensorFlow for 128-KB samples. DLFS-TF delivers the highest throughput. Its throughput is 1.25× and 61.4% higher than Octopus-TF and Ext4-TF, respectively. These performance results demonstrate the performance strength of DLFS for deep learning workloads with TensorFlow.

2) Training Accuracy with AlexNet: In our opportunistic batching optimizations, DLFS is allowed to determine the sequence of input samples for DL applications. We need to evaluate how such delegation of randomization may affect the training accuracy of DL applications. We train AlexNet with the ImageNet dataset over 100 epochs with different sample sequence decision mechanisms. In the case of application-driven full randomization (*Full_Rand*), we completely shuffle the sample names in AlexNet and let the AlexNet model access samples based on the shuffled order. In the case of DLFS-based randomization (*DLFS*), we call *DLFS_bread* and let DLFS determine the sequence of samples. The accuracy results from these tests are shown in Fig. 13. Clearly, there are no observable differences in the training accuracy between the application-drive randomization and the DLFS-based randomization. These tests demonstrate that, with opportunistic batching optimizations, DLFS is effective to achieve high-performance disaggregation without affecting the training accuracy.

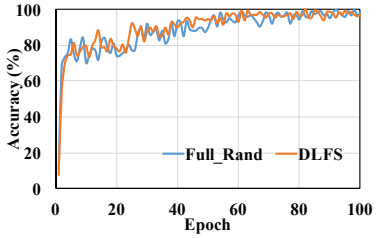


Fig. 13: AlexNet training accuracy with the CIFAR10 dataset.

V. Related Work

Storage disaggregation: Since we focus on disaggregating SSD (especially NVMe SSD) from computation resources,

we compare DLFS with a few SSD-based storage disaggregation studies. Kilmovic et al. [41] discuss a few fine-grained system tuning methods for improving the remote SSD access performance via iSCSI and provide insights about how flash disaggregation leads to resource utilization. Different from this work, our work focuses on storage disaggregation at application-level and leverages the emerging NVMe over Fabrics technique. Guz et al. [31], [32] evaluate the cost of using different protocols (DAS, NVMe-oF, iSCSI) for accessing SSD and the performance of applying these protocols on disaggregated database. Besides leveraging NVMe over Fabrics for storage disaggregation, DLFS is designed as a user-level file system that makes use of disaggregated NVMe devices on large-scale systems for efficient resource utilization. Crail [59] also supports resource disaggregation over NVMe over Fabrics. Contrary to Crail's centralized metadata management, DLFS maintains metadata locally which reduces the potential bottlneck during sample lookup. Also, DLFS offers specialized sample access pattern which is introduced based on the features of DL workloads. Methods to share the use of remote SSD devices have been attempted in ReFlex[42] and Flexdriver[46]. These methods do integrate NVMe over Fabrics in their design and are applicable for storage disaggregation. However, their design is at the block level compared to DLFS's file level disaggregation. Overall, compared to the prior studies, our work provides the first user-level file system that enables storage disaggregation for large-scale DL training.

Burst Buffer File Systems: Burst buffer has shown its importance by its inclusion on many supercomputers. Many works have been done on studying the potential benefits of different burst buffer architectures. DataWarp[4], IME[6] and aBBa[5] are software packages that only support remote burst buffers. IBIO[55], Burstmem[67], and [44] are efforts that also explore different aspects of remote burst buffer support. Also, many works have been done on node-local burst buffers (e.g., BurstFS[66], DataElevator[26], GekkoFS[64], [61]). Different from all these works that only focus on either remote or local burst buffer, we propose DLFS as a user-level storage disaggregation solution which can be applied to both remote and node-local burst buffers. Also, we leverage the Intel SPDK library and the emerging technique of NVMe over Fabrics in our design to explore the full performance of NVMe SSDs and match the system need of deep learning workloads. In addition, the optimizations in those works are introduced based on I/O features of scientific applications, which is different from the read-only feature in data importing procedures of deep learning workloads. In the metadata management, we propose the in-memory sample directory to avoid inter-node communication. In the data management, we allow direct data access to remote/local NVMe devices while batching and relaxing the data access order to accelerate the data importing speed for deep learning workloads.

I/O for Deep Learning Applications: Deep learning frameworks (e.g., TensorFlow [18], Caffe [38], LBANN [63]) contain their own input methods. For example, TensorFlow introduces Dataset API for importing dataset from different formats. However, their data reading performance depends on underlying conventional file systems (e.g., standard POSIX file system) that are designed for general usage instead of being optimized for large-scale training. Contrary to the conventional file systems, DLFS considers deep learning's dataset importing features and provides a very lightweight metadata and data stack. Besides improving the sample throughput at file system level, other efforts have been taken to improve dataset importing speed. Zhang et al. [72] build a user-level file system over a native file system on node-local storage for deep learning with the assistance of MPI send/receive. Contrasting from their design, DLFS completely bypasses the OS kernel and works with the emerging NVMe over Fabrics technique. Zhu et al. [75], [74] proposed DeepIO to preload data into a fixed-size memory, which does not support storage disaggregation for remote clients. Its performance is also limited by the total available memory. DLFS provides a fast in-memory metadata management scheme and enables flexible disaggregated storage access across compute nodes. Pumma et al. [54], [53] study the performance bottleneck when using Caffe with LMDB on multi-node training. However, the proposed solution is only applicable when using LMDB with Caffe. DLFS is designed as a more general data importing file system and can be incorporated in any training frameworks for deep learning.

VI. Conclusion

In this work, we present the design and implementation of DLFS, a user-level, read-optimized file system for deep learning applications which require a large number of small random reads during training. DLFS is designed on top of NVMe over Fabrics [9], including a set of techniques to enable storage disaggregation as efficient and convenient I/O services. Specifically, we have designed an in-memory tree-based sample directory for fast metadata management and its SPDK-based user-level direct I/O, along with opportunistic batching optimizations. Our performance evaluation demonstrates that DLFS can achieve efficient user-level storage disaggregation across a pool of NVMe devices with very little CPU utilization.

Acknowledgment

This work is performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344. LLNL-CONF-771643. This work is also supported in part by the National Science Foundation awards 1561041, 1564647, 1744336, 1763547, and 1822737. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation. [1].

REFERENCES

[1] An Overview of Gradient Descent Optimization Algorithms. http://ruder.io/optimizing-gradient-descent/index.html#minibatchgradientdescent.

[2] Caffe Database Layer. http://caffe.berkeleyvision.org/tutorial/layers/data.html.

[3] Data Storage Keeping Pace for AI and Deep Learning. https://medium.com/predict/data-storage-keeping-pace-for-ai-and-deep-learning-ad3e75e1c67a.

[4] Datawarp. https://www.cray.com/products/storage/datawarp.

[5] EMC: ABBA. https://www.theregister.co.uk/2012/09/21/emc_abba/.

[6] Infinite Memory Engine (IME). https://www.ddn.com/products/.

[7] Intel Rack Scale Design. https://www.intel.com/content/www/us/en/architecture-and-technology/rack-scale-design-overview.html.

[8] Introducing Data Center Fabric, The Next-Generation Facebook Data Center Network. https://code.fb.com/production-engineering/introducing-data-center-fabric-the-next-generation-facebook-data-center-network/.

[9] NVMe over Fabrics. https://nvmexpress.org/wp-content/uploads/NVMe_Over_Fabrics.pdf.

[10] Removing The Storage Bottleneck for AI. https://www.nextplatform.com/2018/03/29/removing-the-storage-bottleneck-for-ai/.

[11] SeaMicro SM10000 System Overview. https://www.tiger-optics.ru/download/seamicro/SM_TO02_v1.4.pdf.

[12] Sierra. https://computation.llnl.gov/computers/sierra.

[13] SPDK: NVMe over Fabrics Target. https://spdk.io/doc/nvmf.html.

[14] SPDK: User Space Drivers. https://spdk.io/doc/userspace.html.

[15] Summit. https://www.olcf.ornl.gov/summit/.

[16] TensorFlow: Adding a New Op. https://www.tensorflow.org/versions/master/extend/adding_an_op.

[17] TensorFlow Dataset API: tf.data.FixedLengthRecordDataset.shuffle(). https://www.tensorflow.org/api_docs/python/tf/data/FixedLengthRecordDataset#shuffle.

[18] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, et al. Tensorflow: Large-Scale Machine Learning on Heterogeneous Distributed Systems. *arXiv preprint arXiv:1603.04467*, 2016.

[19] Takuya Akiba, Shuji Suzuki, and Keisuke Fukuda. Extremely Large Minibatch SGD: Training Resnet-50 on Imagenet in 15 Minutes. *arXiv preprint arXiv:1711.04325*, 2017.

[20] Krste Asanovic and David Patterson. Firebox: A Hardware Building Block for 2020 Warehouse-Scale Computers. In *USENIX FAST*, volume 13, 2014.

[21] Luiz André Barroso, Jimmy Clidaras, and Urs Hölzle. The atacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines. *Synthesis lectures on computer architecture*, 8(3):1–154, 2013.

[22] Wei Cao, Zhenjun Liu, Peng Wang, Sen Chen, Caifeng Zhu, Song Zheng, Yuhui Wang, and Guoqing Ma. PolarFS: an Ultra-Low Latency and Failure Resilient Distributed File System for Shared Storage Cloud Database. *Proceedings of the VLDB Endowment*, 11(12):1849–1862, 2018.

[23] Fahim Chowdhury, Yue Zhu, Todd Heer, Saul Paredes, Adam Moody, Robin Goldstone, Kathryn Mohror, and Weikuan Yu. I/O Characterization and Performance Evaluation of BeeGFS for Deep Learning. In *Proceedings of the 48th International Conference on Parallel Processing*, page 80. ACM, 2019.

[24] David Culler, J.P. Singh, and Anoop Gupta. *Parallel Computer Architecture: A Hardware/Software Approach*. Morgan Kaufmann, 1st edition, 1998. The Morgan Kaufmann Series in Computer Architecture and Design.

[25] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. Imagenet: A Large-scale Hierarchical Image Database. In *Computer Vision and Pattern Recognition, 2009. CVPR 2009. IEEE Conference on*, pages 248–255. IEEE, 2009.

[26] Bin Dong, Suren Byna, Kesheng Wu, Hans Johansen, Jeffrey N Johnson, Noel Keen, et al. Data Elevator: Low-Contention Data Movement in Hierarchical Storage System. In *2016 IEEE 23rd International Conference on High Performance Computing (HiPC)*, pages 152–161. IEEE, 2016.

[27] Huansong Fu, Manjunath Gorentla Venkata, Ahana Roy Choudhury, Neena Imam, and Weikuan Yu. High-performance key-value store on openshmem. In *Proceedings of the 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, pages 559–568. IEEE Press, 2017.

[28] Peter Xiang Gao, Akshay Narayan, Sagar Karandikar, Joao Carreira, Sangjin Han, Rachit Agarwal, Sylvia Ratnasamy, and Scott Shenker. Network Requirements for Resource Disaggregation. In *OSDI*, volume 16, pages 249–264, 2016.

[29] Giulio Giunta, Raffaele Montella, Giuseppe Agrillo, and Giuseppe Coviello. A GPGPU Transparent Virtualization Component for High Performance Computing Clouds. In *European Conference on Parallel Processing*, pages 379–391. Springer, 2010.

[30] Priya Goyal, Piotr Dollár, Ross Girshick, Pieter Noordhuis, Lukasz Wesolowski, Aapo Kyrola, Andrew Tulloch, Yangqing Jia, and Kaiming He. Accurate, Large Minibatch SGD: Training Imagenet in 1 Hour. *arXiv preprint arXiv:1706.02677*, 2017.

[31] Zvika Guz, Harry Huan Li, Anahita Shayesteh, and Vijay Balakrishnan. NVMe-Over-Fabrics Performance Characterization and the Path to Low-Overhead Flash Disaggregation. In *Proceedings of the 10th ACM International Systems and Storage Conference*, page 16. ACM, 2017.

[32] Zvika Guz, Harry Huan Li, Anahita Shayesteh, and Vijay Balakrishnan. Performance Characterization of NVMe-over-Fabrics Storage Disaggregation. *ACM Transactions on Storage (TOS)*, 14(4):31, 2018.

[33] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep Residual Learning for Image Recognition. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 770–778, 2016.

[34] HP. Moonshot System: The Worlds First Software-Defined Servers. http://h10032.www1.hp.com/ctg/Manual/c03728406.pdf.

[35] Intel. Intel RSA. https://www.intel.com/content/www/us/en/architecture-and-technology/rack-scale-design-overview.html.

[36] Nusrat Sharmin Islam, Md Wasi-ur Rahman, Xiaoyi Lu, and Dhabaleswar K Panda. High Performance Design for HDFS with Byte-Addressability of NVM and RDMA. In *Proceedings of the 2016 International Conference on Supercomputing*, page 8. ACM, 2016.

[37] Xianyan Jia, Shutao Song, Wei He, Yangzihao Wang, Haidong Rong, Feihu Zhou, Liqiang Xie, Zhenyu Guo, Yuanzhou Yang, Liwei Yu, et al. Highly Scalable Deep Learning Training System with Mixed-Precision: Training Imagenet in Four Minutes. *arXiv preprint arXiv:1807.11205*, 2018.

[38] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. Caffe: Convolutional Architecture for Fast Feature Embedding. In *Proceedings of the 22nd ACM International Conference on Multimedia*, pages 675–678. ACM, 2014.

[39] Svilen Kanev, Juan Pablo Darago, Kim Hazelwood, Parthasarathy Ranganathan, Tipp Moseley, Gu-Yeon Wei, and David Brooks. Profiling a Warehouse-Scale Computer. In *ACM SIGARCH Computer Architecture News*, volume 43, pages 158–169. ACM, 2015.

[40] Kostas Katrinis, Dimitris Syrivelis, D Pnevmatikatos, Georgios Zervas, Dimitris Theodoropoulos, Iordanis Koutsopoulos, K Hasharoni, Daniel Raho, Christian Pinto, F Espina, et al. Rack-Scale Disaggregated Cloud Data Centers: The dReDBox Project Vision. In *Proceedings of the 2016 Conference on Design, Automation & Test in Europe*, pages 690–695. EDA Consortium, 2016.

[41] Ana Klimovic, Christos Kozyrakis, Eno Thereska, Binu John, and Sanjeev Kumar. Flash Storage Disaggregation. In *Proceedings of the Eleventh European Conference on Computer Systems*, page 29. ACM, 2016.

[42] Ana Klimovic, Heiner Litz, and Christos Kozyrakis. Reflex: Remote Flash Local Flash. *ACM SIGOPS Operating Systems Review*, 51(2):345–359, 2017.

[43] Sergey Legtchenko, Hugh Williams, Kaveh Razavi, Austin Donnelly, Richard Black, Andrew Douglas, Nathanaël Cheriere, Daniel Fryer, Kai Mast, Angela Demke Brown, et al. Understanding Rack-Scale Disaggregated Storage. In *9th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 17), Santa Clara, CA*, 2017.

[44] Ning Liu, Jason Cope, Philip Carns, Christopher Carothers, Robert Ross, Gary Grider, Adam Crume, and Carlos Maltzahn. On the Role of Burst Buffers in Leadership-Class Storage Systems. In *012 IEEE 28th Symposium on Mass Storage Systems and Technologies (MSST)*, pages 1–11. IEEE, 2012.

[45] Youyou Lu, Jiwu Shu, Youmin Chen, and Tao Li. Octopus: an RDMA-enabled Distributed Persistent Memory File System. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 773–785, 2017.

[46] Krishna T Malladi, Manu Awasthi, and Hongzhong Zheng. Flexdrive: a Framework to Explore NVMe Storage Solutions. In *2016 IEEE 18th International Conference on High Performance Computing and Communications; IEEE 14th International Conference on Smart City; IEEE 2nd International Conference on Data Science and Systems (HPCC/SmartCity/DSS)*, pages 1115–1122. IEEE, 2016.

[47] Hiroaki Mikami, Hisahiro Suganuma, Yoshiki Tanaka, Yuichi Kageyama, et al. ImageNet/ResNet-50 Training in 224 Seconds. *arXiv preprint arXiv:1811.05233*, 2018.

[48] Radhika Mittal, Alexander Shpiner, Aurojit Panda, Eitan Zahavi, Arvind Krishnamurthy, Sylvia Ratnasamy, and Scott Shenker. Revisiting network support for rdma. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '18, pages 313–326, New York, NY, USA, 2018. ACM.

[49] Mihir Nanavati, Jake Wires, and Andrew Warfield. Decibel: Isolation and Sharing in Disaggregated Rack-Scale Storage. In *NSDI*, pages 17–33, 2017.

[50] Minoru Oikawa, Atsushi Kawai, Kentaro Nomura, Kenji Yasuoka, Kazuyuki Yoshikawa, and Tetsu Narumi. DS-CUDA: A Middleware to Use Many GPUs in the Cloud Environment. In *High Performance Computing, Networking, Storage and Analysis (SCC), 2012 SC Companion:*, pages 1207–1214. IEEE, 2012.

[51] Jian Ouyang, Shiding Lin, Song Jiang, Zhenyu Hou, Yong Wang, and Yuanzheng Wang. SDF: Software-Defined Flash for Web-Scale Internet Storage Systems. In *ACM SIGARCH Computer Architecture News*, volume 42, pages 471–484. ACM, 2014.

[52] Antonio J Peña, Carlos Reaño, Federico Silla, Rafael Mayo, Enrique S Quintana-Ortí, and José Duato. A Complete and Efficient CUDA-Sharing Solution for HPC Clusters. *Parallel Computing*, 40(10):574–588, 2014.

[53] Sarunya Pumma, Min Si, Wu-chun Feng, and Pavan Balaji. Parallel I/O Optimizations for Scalable Deep Learning. In *2017 IEEE 23rd International Conference on Parallel and Distributed Systems (ICPADS)*, pages 720–729. IEEE, 2017.

[54] Sarunya Pumma, Min Si, Wu-chun Feng, and Pavan Balaji. Towards Scalable Deep Learning via I/O Analysis and Optimization. In *2017 IEEE 19th International Conference on High Performance Computing and Communications; IEEE 15th International Conference on Smart City; IEEE 3rd International Conference on Data Science and Systems (HPCC/SmartCity/DSS)*, pages 223–230. IEEE, 2017.

[55] Kento Sato, Kathryn Mohror, Adam Moody, Todd Gamblin, Bronis R De Supinski, Naoya Maruyama, and Satoshi Matsuoka. A User-Level InfiniBand-Based File System and Checkpoint Strategy for Burst Buffers. In *Cluster, Cloud and Grid Computing (CCGrid), 2014 14th IEEE/ACM International Symposium on*, pages 21–30. IEEE, 2014.

[56] Xuanhua Shi, Ming Li, Wei Liu, Hai Jin, Chen Yu, and Yong Chen. Ssdup: a Traffic-Aware SSD Burst Buffer for Hpc Systems. In *Proceedings of the International Conference on Supercomputing*, page 27. ACM, 2017.

[57] Samuel L Smith, Pieter-Jan Kindermans, Chris Ying, and Quoc V Le. Don't Decay the Learning Rate, Increase the Batch Size. *arXiv preprint arXiv:1711.00489*, 2017.

[58] Murray Stokely, Amaan Mehrabian, Christoph Albrecht, Francois Labelle, and Arif Merchant. Projecting Disk Usage Based on Historical Trends in a Cloud Environment. In *Proceedings of the 3rd workshop on Scientific Cloud Computing*, pages 63–70. ACM, 2012.

[59] Patrick Stuedi, Animesh Trivedi, Jonas Pfefferle, Radu Stoica, Bernard Metzler, Nikolas Ioannou, and Ioannis Koltsidas. Crail: A High-Performance I/O Architecture for Distributed Data Processing. *IEEE Data Eng. Bull.*, 40(1):38–49, 2017.

[60] S. Swanson and A. M. Caulfield. Refactor, Reduce, Recycle: Restructuring the I/O Stack for the Future of Storage. *Computer*, 46(8):52–59, August 2013.

[61] Kun Tang, Ping Huang, Xubin He, Tao Lu, Sudharshan S Vazhkudai, and Devesh Tiwari. Toward Managing HPC Burst Buffers Effectively: Draining Strategy to Regulate Bursty I/O Behavior. In *2017 IEEE 25th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*, pages 87–98. IEEE, 2017.

[62] J. Taylor. Facebook's data center infrastructure: Open compute, disaggregated rack, and beyond. In *2015 Optical Fiber Communications Conference and Exhibition (OFC)*, pages 1–1, March 2015.

[63] Brian Van Essen, Hyojin Kim, Roger Pearce, Kofi Boakye, and Barry Chen. LBANN: Livermore Big Artificial Neural Network HPC Toolkit. In *Proceedings of the Workshop on Machine Learning in High-Performance Computing Environments*, page 5. ACM, 2015.

[64] Marc-André Vef, Nafiseh Moti, Tim Süß, Tommaso Tocci, Ramon Nou, Alberto Miranda, Toni Cortes, and André Brinkmann. Gekkofs-a temporary distributed file system for hpc applications. In *2018 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 319–324. IEEE, 2018.

[65] T. Wang, A. Moody, Y. Zhu, K. Mohror, K. Sato, T. Islam, and W. Yu. MetaKV: A Key-Value Store for Metadata Management of Distributed Burst Buffers. In *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 1174–1183, May 2017.

[66] Teng Wang, Kathryn Mohror, Adam Moody, Kento Sato, and Weikuan Yu. An Ephemeral Burst-Buffer File System for Scientific Applications. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, page 69. IEEE Press, 2016.

[67] Teng Wang, Sarp Oral, Yandong Wang, Brad Settlemyer, Scott Atchley, and Weikuan Yu. Burstmem: A High-Performance Burst Buffer System for Scientific Applications. In *2014 IEEE International Conference on Big Data (Big Data)*, pages 71–79. IEEE, 2014.

[68] Kai Wu, Yingchao Huang, and Dong Li. Unimem: Runtime Data Managementon Non-Volatile Memory-Based Heterogeneous Main Memory. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, page 58. ACM, 2017.

[69] Qiumin Xu, Manu Awasthi, Krishna T Malladi, Janki Bhimani, Jingpei Yang, and Murali Annavaram. Performance Analysis of Containerized Applications on Local and Remote Storage. In *Proc. of MSST*, 2017.

[70] Masafumi Yamazaki, Akihiko Kasagi, Akihiro Tabuchi, Takumi Honda, Masahiro Miwa, Naoto Fukumoto, Tsuguchika Tabaru, Atsushi Ike, and Kohta Nakashima. Yet Another Accelerated SGD: ResNet-50 Training on ImageNet in 74.7 seconds. *arXiv preprint arXiv:1903.12650*, 2019.

[71] Ziye Yang, James R Harris, Benjamin Walker, Daniel Verkamp, Changpeng Liu, Cunyin Chang, Gang Cao, Jonathan Stern, Vishal Verma, and Luse E Paul. SPDK: A Development Kit to Build High Performance Storage Applications. In *2017 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*, pages 154–161. IEEE, 2017.

[72] Zhao Zhang, Lei Huang, Uri Manor, Linjing Fang, Gabriele Merlo, Craig Michoski, John Cazes, and Niall Gaffney. FanStore: Enabling Efficient and Scalable I/O for Distributed Deep Learning. *arXiv preprint arXiv:1809.10799*, 2018.

[73] Qing Zheng, Kai Ren, Garth Gibson, Bradley W Settlemyer, and Gary Grider. DeltaFS: Exascale File Systems Scale Better Without Dedicated Servers. In *Proceedings of the 10th Parallel Data Storage Workshop*, pages 1–6. ACM, 2015.

[74] Yue Zhu, Fahim Chowdhury, Huansong Fu, Adam Moody, Kathryn Mohror, Kento Sato, and Weikuan Yu. Multi-Client DeepIO for Large-Scale Deep Learning on HPC Systems.

[75] Yue Zhu, Fahim Chowdhury, Huansong Fu, Adam Moody, Kathryn Mohror, Kento Sato, and Weikuan 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 (MASCOTS)*, pages 145–156. IEEE, 2018.