Exploiting CXL-based Memory for Distributed Deep Learning

Moiz Arif*
Rochester Institute of Technology
Rochester, NY, USA
ma3890@cs.rit.edu

Kevin Assogba
Rochester Institute of Technology
Rochester, NY, USA
kta7930@cs.rit.edu

M. Mustafa Rafique
Rochester Institute of Technology
Rochester, NY, USA
mrafique@cs.rit.edu

Sudharshan Vazhkudai
Micron Technology, Inc.
Austin, TX, USA
svazhkudai@micron.com

ABSTRACT

Deep learning (DL) is being widely used to solve complex problems in scientific applications from diverse domains, such as weather forecasting, medical diagnostics, and fluid dynamics simulation. DL applications consume a large amount of data using large-scale high-performance computing (HPC) systems to train a given model. These workloads have large memory and storage requirements that typically go beyond the limited amount of main memory available on an HPC server. This significantly increases the overall training time as the input training data and model parameters are frequently swapped to slower storage tiers during the training process. In this paper, we use the latest advancements in the memory subsystem, specifically Compute Express Link (CXL), to provide additional memory and fast scratch space for DL workloads to reduce the overall training time while enabling DL jobs to efficiently train models using data that is much larger than the installed system memory. We propose a framework, called *DeepMemoryDL*, that manages the allocation of additional CXL-based memory, introduces a fast intermediate storage tier, and provides intelligent prefetching and caching mechanisms for DL workloads. We implement and integrate *DeepMemoryDL* with a popular DL platform, TensorFlow, to show that our approach reduces read and write latencies, improves the overall I/O throughput, and reduces the training time. Our evaluation shows a performance improvement of up to 34% and 27% compared to the default TensorFlow platform and CXL-based memory expansion approaches, respectively.

KEYWORDS

Deep Learning; Prefetching; Caching; TensorFlow; Data Pipeline

ACM Reference Format:
Moiz Arif, Kevin Assogba, M. Mustafa Rafique, and Sudharshan Vazhkudai. 2022. Exploiting CXL-based Memory for Distributed Deep Learning. In *51st International Conference on Parallel Processing (ICPP '22), August 29-September 1, 2022, Bordeaux, France.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3545008.3545054

*A part of this work was done while Moiz Arif was doing an internship at Micron Technology, Inc.

ICPP '22, August 29-September 1, 2022, Bordeaux, France

ACM ISBN 978-1-4503-9733-9/22/08…$15.00
https://doi.org/10.1145/3545008.3545054

1 INTRODUCTION

Modern enterprise, scientific, and high-performance computing (HPC) applications [14] are distributed, highly scalable, and consume huge amount of data during execution. Many of these applications run deep learning (DL) models to bring innovations in the fields of healthcare [38], finance [28], autonomous vehicles [45], etc. Hardware technologies are continuously bringing innovations to support the compute and complex I/O requirements of DL applications. Therefore, the execution runtime of DL workloads must also evolve to support and leverage advancements in the underlying hardware and data center infrastructure not only to reduce the time-to-answer of time-sensitive applications but also to improve the utilization of available data center resources.

The latest data centers and HPC servers are equipped with large memory and storage resources to meet the ever-increasing requirements of DL applications. To improve application performance and reduce the training time, these servers should have enough memory to fit the entire model and training data in the main memory. However, for complex models and large datasets [13], the memory footprint of DL jobs [9] exceeds the available system memory leading to undesirable swapping of excessive application data to local disks. This leads to reduced performance of DL jobs and increased training time as the I/O operations can take up to 85% of the total training time for large input datasets [15]. To address this challenge, high-capacity memory modules are installed that eventually lead to occupying all available memory slots on a server [11]. This increases the overall cost of each server to the extent that the main memory account for about 50% of the total acquisition cost in leading HPC centers. Therefore, investigating solutions to augment the direct-attached memory with PCIe-based memory and storage solutions can reduce the total cost of ownership (TCO) [5]. However, the amount of training data continues to outgrow the available system memory [44] requiring the underlying storage subsystem to develop efficient data management techniques to load data from the I/O subsystem to the main memory.

Several new interconnect technologies, such as Compute Express Link (CXL) [41] and Gen-Z [2], have been developed to sustain higher I/O and network throughput. CXL is a cache-coherent interconnect for processors, memory expansion, and accelerators that run over the PCIe interface [41]. It maintains memory coherency

between the main memory space and CXL attached memory to achieve higher performance and lower overall system cost. While there are no production CXL devices available yet, the emulation of CXL-based storage devices has shown promising performance improvements over the standard PCIe-based devices [39]. The CXL devices can be used in a variety of configurations, e.g., memory caching, pooling, and expansion, to provide high-speed memory access [39]. However, it is not clear how these configurations can be used to improve the performance of DL workloads.

In this paper, we explore the use of CXL-based memory and storage to improve the performance of DL workloads. Specifically, we analyze data staging and placement in the popular TensorFlow [7] platform and enable the use of CXL-based memory and storage to improve its data flow path. We address the following limitations in the data flow path of the TensorFlow platform: (i) TensorFlow is unaware of heterogeneity in the underlying memory and storage subsystems [22]; (ii) TensorFlow does not leverage additional memory and storage tiers to optimize data input pipelines [8]; and (iii) the default caching and prefetching mechanisms of TensorFlow are insufficient for processing large datasets on servers with limited memory and storage resources [29]. These limitations increase the training time of DL jobs as they lead to I/O stalls and sub-optimal utilization of data center resources. Several techniques, such as informed prefetching and caching [37], look ahead buffers [17], and data sharding [34], have been explored to address these limitations and bridge the performance gap between computation and I/O resources. However, these techniques must be adapted for using CXL-based memory devices in the TensorFlow platform.

We propose a framework, called *DeepMemoryDL*, to address the challenges of data staging and placement in the TensorFlow platform for distributed settings. The key idea behind *DeepMemoryDL* is to enable TensorFlow to identify and leverage various available tiers of memory and storage devices and develop proactive prefetching and caching techniques to avoid I/O stalls by exploiting the I/O access patterns of DL workloads. This reduces I/O wait time and the overall training time of DL jobs. *DeepMemoryDL* minimize the latency to access the training data by ensuring that the required data is already available for the compute threads in the fastest storage tier. To achieve this, we first classify different available memory and storage devices into tiers based on their performance profiles and then prefetch and cache the required data at the appropriate tier on each worker node. To the best of our knowledge, our work is the first to explore how CXL can be effectively and seamlessly used to improve the performance of DL jobs while reducing the complexity in the I/O subsystem and storage configurations. Specifically, we make the following contributions in this paper.

- We propose a novel framework, called *DeepMemoryDL*, that utilizes fast memory and storage tiers to prefetch and cache data to ensure deterministic performance of DL workloads while minimizing I/O stalls.
- We develop a prototype implementation of *DeepMemoryDL* for data center environments and integrate it with the TensorFlow platform to demonstrate its effectiveness.
- We evaluate and compare the performance of *DeepMemoryDL* with the default TensorFlow platform using representative DL workloads. Our evaluation shows that our proposed *DeepMemoryDL* framework achieves up to 34% performance improvement as compared to the default TensorFlow platform and up to 27% as compared to the CXL-based memory expansion approaches.

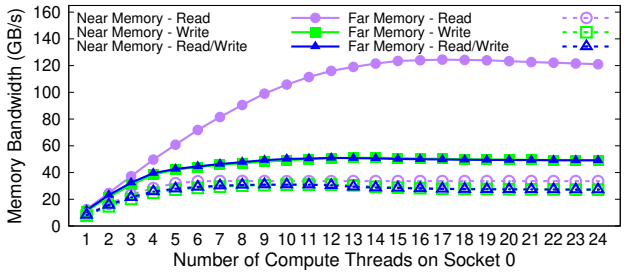

Figure 1: Impact of increasing Socket 0 thread count on near and far memory bandwidth.

2 CHALLENGES AND MOTIVATION

2.1 Slow Storage Tiers and DL Workloads

Heterogeneity is inevitable in modern data centers due to continuous upgrades and advancements in compute, memory, storage, and network technologies. Each memory and storage tier has a distinct throughput and access latency resulting in unpredictable application performance. Distributed DL workloads, which require fast memory, low-latency I/O pipelines, and a large storage medium to store huge datasets, are executed over multiple servers with varying memory and storage capacity and performance. The large input dataset is divided into shards and each shard is divided into small batches [26] and each batch is sent to the computing unit to train the given model. Therefore, the performance of DL workloads heavily depends on data transfer speed and how quickly the required data is made available to the processing threads. Typically, the processing threads process the data at a much higher rate than staging the data into the system memory. We observe that as the memory allocation moves farther away, e.g., to other NUMA nodes, from the compute threads, the performance starts to drop due to the impact of latency associated with accessing memory and storage resources over the respective interconnections. The result is shown in Figure 1. We ran STREAM Triad [31] and observe that the memory bandwidth is maximum when the data is accessed on the same node as the compute threads. Efforts have been made to predict both optimal core allocation and memory bandwidth usage with high accuracy and low overhead for memory-intensive multi-threaded applications on large-scale clusters [43]. However, such optimizations do not directly apply to distributed DL jobs over heterogeneous memory and storage resources.

The memory footprint of DL workloads increases over time [30], which leads to excessive swapping for servers with limited memory. The swap space is configured over the local storage, e.g., a solid-state disk (SSD), and access to which is significantly slower than the main memory. For a dual-socket system, a thread on socket 0 can access memory on socket 1 at the cost of additional latency. However, this additional latency is significantly less than the swap space. The idle

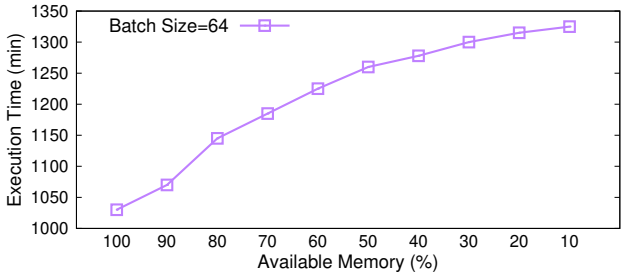

Figure 2: Impact of available system memory on DL workload performance with MobileNetv2 and ImageNet dataset with batch size of 64 and 3 epochs.

or unloaded latency is the latency when the memory subsystem is idle and represents the best case latency, whereas, the loaded latency is the latency when the memory subsystem is saturated with memory requests and represents a more realistic measurement for applications. For the thread on socket 0, we observe an idle latency of 70 *n*s for local memory access and 134.5 *n*s for accessing memory on socket 1 for an Intel processor. Similarly, the loaded latency for such a system is 228.59 *n*s for read-only traffic. Meanwhile, the read latency to swap space is 305.5 μs.

The memory access patterns of a workload determine the impact of using swap space on its performance. Most modern DL applications are read-intensive and perform write operations at regular intervals. The performance of a DL application drops when the memory footprint involves swap storage resulting in increased time per epoch and causing an increase in overall training time as shown in Figure 2. The memory footprint of the DL job is about 166 GB and fits entirely into the memory when 100% memory is available to the DL workload. The configured system swap space is 200 GB and the increased execution time is attributed to increased reads and write to swap. To mitigate the impact of using swap space, it is critical to explore the use of high-capacity and low-latency alternatives, such as CXL-based memory modules.

2.2 I/O Challenges in DL Data Pipelines

Executing a DL workload requires processing large datasets to achieve the desired training accuracy of the given DL model. The growing size of datasets emphasizes the importance of designing highly efficient I/O pipelines, especially in distributed DL environments where the dataset is distributed across multiple workers for processing. DL workloads contain various I/O stages, e.g., data loading, caching, prefetching, model fitting, and checkpointing. DL platforms provide methods and APIs to perform parallel I/O operations and improve the performance of data staging and placement. For example, TensorFlow provides *tf.data* [33] whereas PyTorch provides *DataLoader* [36] APIs to improve the performance of data staging. However, these built-in methods and APIs for data staging do not incorporate different memory tiers for storing large datasets.

During model training, the collection of data samples called batches is shuffled randomly [27] before each epoch for model convergence and to prevent overfitting. Typically, the input data is cached into memory during the first epoch to speed up the read operations in subsequent epochs. However, the cache hit rate is severely impacted when the entire dataset cannot fit into the available memory. The cached data is evicted after being processed during an epoch to load new batches into the main memory, causing thrashing and forcing the workers to fetch data from slower storage devices. Prefetching reduces I/O stalls by bringing data to a lower storage tier and moving the next batch into the main memory before the next iteration. This becomes challenging for large datasets as prefetching the next batch takes longer than processing the current batch [15]. Therefore, a data staging strategy is required to incorporate the heterogeneity of underlying memory and storage tiers to orchestrate the data pipeline and reduce I/O stalls.

2.3 Limitations of Data Pre-processing in DL Platforms

DL platforms provide APIs to load datasets into the main memory of workers for training. The most common approach involves reading individual data samples recursively from the raw data files. TensorFlow provides a high-level data abstraction in the form of wrapped data sequences known as *TFRecords*. TFRecords are stored in a sequential layout to speed up the process of loading the dataset into the main memory. This abstraction is tightly integrated with TensorFlow's data pre-processing stage to provide efficient batch processing while maximizing the use of available system memory.

The size and location of the dataset govern the creation of the data pipeline, pre-processing, and data loading into the main memory. Typically, large datasets cannot fit into the memory subsystem of a single worker due to limited memory and storage on each worker. A DL job fails to execute when available system memory is not enough to hold the entire training dataset. To avoid such failures due to limited resources, data is placed into pipelines and prefetching and caching techniques are used to efficiently manage memory and storage resources. TensorFlow caches the dataset in memory for improving I/O, however, its caching mechanism becomes ineffective if the memory is not large enough to host the entire dataset and it does not yield any performance benefit because the dataset cannot be cached in memory. TensorFlow also allows caching to disk, which becomes beneficial when caching to local NVMe devices as compared to reading the batch from network-attached storage. Another important factor that improves performance is prefetch, which ensures that the dataset is loaded in the main memory before the training job has finished processing the previous batch of data.

We analyze the impact of caching and prefetching techniques for two scenarios. First, the training data is greater than the available system memory resulting in excessive swapping, and second, the training data is smaller than the available system memory and it can fit entirely into the memory without swap utilization. We used TensorFlow's optimized data pipelines with limited memory and observed performance degradation due to frequent disk accesses. Figure 3 shows the results. We observe that prefetching and caching into memory yield better performance as compared to using the swap storage. Moreover, the impact of data prefetching reduces when the available system memory is less than the dataset size. With limited memory available, disk I/O becomes a bottleneck as compute units consume data at a much higher rate. We observe this during data loading, pre-processing, and training by analyzing

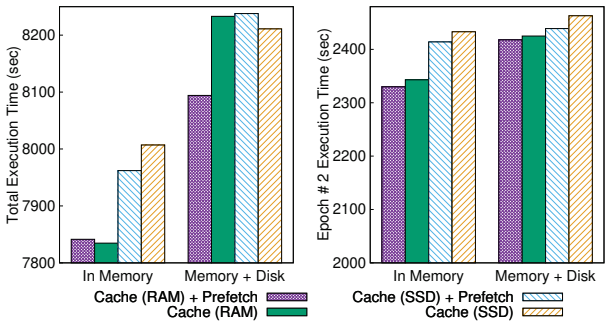

Figure 3: Impact of caching and prefetching on DL workload performance with MobileNetv2 and a subset of ImageNet with batch size of 64 and 3 epochs.

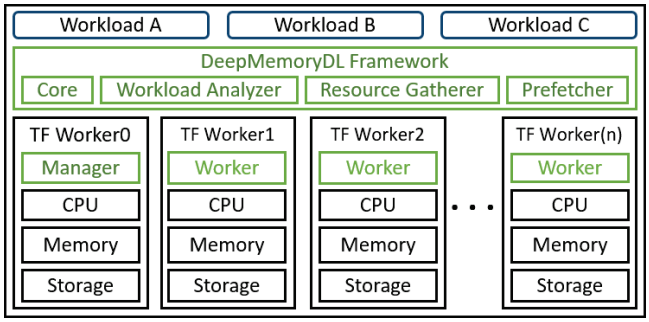

Figure 4: Proposed architecture of *DeepMemoryDL*.

the DL job's footprint, available system memory, and disk utilization with the help of PCM and SYSSTAT monitoring tools. We ran several experiments and observe that the data loading throughput is ~320 MB/s when loaded from the SSD in our evaluation setup. Prefetching consumes additional memory to store data batches and speeds up the training process. Therefore, less available memory significantly increases application execution times and reduces throughput. This impact is amplified with large batch sizes despite TensorFlow's caching and prefetching mechanisms. For a worker where less amount of memory is available, caching to the local storage is useful since it reduces access latency as compared to the network storage to fetch the same data for subsequent epochs. Moreover, caching to the local storage is beneficial if faster storage, such as NVMe [23] and CXL devices are available at the worker nodes. Our experiments show a throughput of ~5284 MB/s when reading data from CXL-based storage which is ~16.5× higher than the throughput observed when reading from the local storage.

3 *DeepMemoryDL* DESIGN

3.1 Design Objectives

The main goal of *DeepMemoryDL* framework is to improve the performance of DL applications by leveraging CXL-based memory devices to proactively stage the training data at the worker node and reduce I/O stalls. The key objectives of *DeepMemoryDL* are:

(1) Enable access to the additional memory space over the CXL interface for DL workloads. This will allow the DL applications to train the given model using a large dataset and reduce the training time.
(2) Avoid throughput bottlenecks in the I/O pipelines and ensure minimal I/O response time by intelligently prefetching the data required by the DL workloads and placing it close to the processing threads.
(3) Provide fast CXL-based scratch space to store the training data, thus enabling high bandwidth access to data and eliminating I/O access to slower storage tiers.

3.2 Design Overview

In this paper, we extend the popular TensorFlow platform to ensure that the training data is available for the processing threads in the fastest storage tier by enabling TensorFlow to request additional memory resources using *DeepMemoryDL*. Figure 4 shows the high-level architecture of the *DeepMemoryDL* framework. We develop a lightweight *Resource Gatherer Module* that collects compute, memory, network, and storage resources of all servers included in the cluster. We implement a *Workload Analyzer Module* that analyzes the submitted DL workload and breaks down the job in I/O and compute phases. Moreover, we develop a *Prefetcher Module* that prefetches data and loads it in the main memory before it is required by the processing thread to minimize I/O stalls.

3.3 *DeepMemoryDL* Architecture

3.3.1 Resource Gatherer Module. The *Resource Gatherer Module* captures information about compute, memory, network, and storage resources of all servers in the cluster. This module is responsible for the following tasks:

- **Maintain System Specifications:** The *Resource Gatherer Module* maintains a list of hardware specifications of the system including CPU make, model, caches, UPI/IF, installed memory, memory channels, memory controllers, supported memory speeds, local storage, and system mount points.
- **Maintain Available System Resources:** The *Resource Gatherer Module* maintains up-to-date information about the available system resources on all servers in the cluster.
- **Memory and Storage Classification:** The *Resource Gatherer Module* uses the system specifications on each server to classify them into tiers based on their performance statistics, i.e., the total achievable bandwidth, IOPS, and latency. This is obtained by running a set of micro-benchmarks, i.e., LMBench [32], FIO [1], and STREAM [31], to classify the available servers into tiered memory and storage subsystem that is used for prefetching and caching the training data.

3.3.2 Workload Analyzer Module. The *Workload Analyzer Module* analyzes the submitted DL jobs to isolate the I/O operations from the computation phases. It is responsible for the following tasks:

- **Analyze DL Job:** The *Workload Analyzer Module* analyzes the submitted DL job to capture model-specific information, i.e., the DL model, parameters, dataset, epochs, batch size, data pre-processing stage, and model training steps. It also identifies if TensorFlow's native checkpoint or data caching option is enabled for the submitted job.

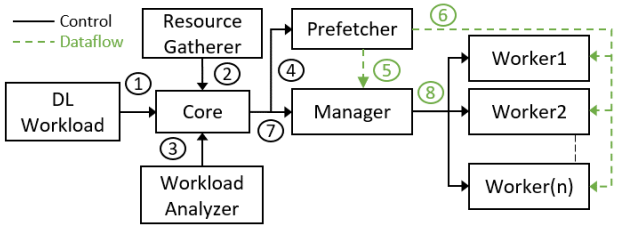

Figure 5: Control flow between *DeepMemoryDL* components.

- **Separate Data Processing from Execution:** The *Workload Analyzer Module* divides the submitted DL job into two main phases namely data processing and model execution. The data processing phase includes dataset loading and preprocessing operations. The model execution phase consists of the model training, validation, and evaluation phases.
- **Analyze Dataset and Batches:** The *Workload Analyzer Module* tracks dataset shards assigned to each TensorFlow worker at the start of the training process. This data is used to estimate the memory and storage allocations on each worker and to accurately determine the completion time for prefetching the required data in memory tiers.

3.3.3 Core Module. The *Core Module* is the main component of *DeepMemoryDL*. It is divided into two parts, i.e., manager and workers. The manager resides on the same node as the master node in TensorFlow and supervises all operations of *DeepMemoryDL*. Its main responsibilities are: 1) compute the batching schedule for each worker; 2) share the batching schedule with each worker along with metadata that specifies the location of each batch for the corresponding epoch; 3) inform each worker about when to launch the prefetching threads; and 4) inform each worker about the storage tier where a particular batch should be prefetched based on the memory and storage tiers available at a worker node. This information is sent to each worker along with the prefetching schedule so that the *Prefetcher Module* can proactively prefetch the batches and load them in the memory tiers. The workers reside at the worker nodes and perform tasks such as monitoring local buffers, prefetching and caching the required data as specified by the manager. The *Core Module* uses the information provided by other modules to issue control instructions to the workers. The *Core Module* exposes an API that can be used by the DL workloads to request memory space. Once a request is received, *DeepMemoryDL* fetches the latest resource availability data, computes the ideal execution strategy, and services the request. Figure 5 shows the interactions between different components of *DeepMemoryDL* and is explained as follows. ① A DL job is submitted by the user to the *Core Module*. ② The *Core Module* collects the details about the existing memory and storage tiers from the *Resource Gatherer Module*. ③ Concurrently, the *Workload Analyzer Module* analyzes the submitted DL workload to identify the operations involved in the data processing and training phases. ④ The *Core Module* forwards all the information on the DL job to the prefetcher module to create a prefetching and caching schedule. ⑤ The *Prefetcher Module* executes the schedule on the manager node which resides alongside the TensorFlow master node. ⑥ The *Prefetcher Module* module executes the schedule on all the worker nodes and ensures that the data is available on the fastest memory and storage tiers for I/O optimization. ⑦ The *Core Module* forwards the DL job to the manager for execution. ⑧ The manager shares the batching schedule with the workers and coordinates the execution of the DL job with all the worker nodes.

Algorithm 1: Data prefetching and caching scheduling.

```
for each worker in cluster do
    determine location for n batches
    if tier t avail. buffer space ≥ space for n batches then
        prefetch n batches from location x to tier t
    else
        prefetch (n − k) batches from location x to tier t
        prefetch k batches from location x to tier t − 1
    end if
    if buffer space in tier t ≥ util. threshold at tier t then
        if batch b is needed in upcoming i iterations then
            cache batch b to tier t − 1
        else
            evict batch b
        end if
    end if
end for
```

The *Core Module* defines prefetching and caching buffer sizes at each memory and storage tier on the worker nodes. The allocated buffers are adaptive to handle batches with varying sizes depending on the available system resources. A training batch contains n elements of width x_i, height y_i, and depth z_i stored in memory as arrays of d_i bytes objects. The memory size in bytes of one element is the product of the width, height, depth, and the number of bytes consumed per pixel. Therefore, the size S_b of a batch b is computed as $S_b = \sum_{i=1}^{n} x_i \times y_i \times z_i \times d_i$. The manager works closely with the *Prefetcher Module* on all the workers to ensure that appropriate buffers are allocated. *DeepMemoryDL* starts with a reservation of 20% of the available space at the memory or storage tier. The *Core Module* adjusts the allocation of buffer S_t based on the available space at tier t and the total number of batches B scheduled to be loaded onto t such that the total number of elements at the memory or storage tier does not exceed the space allocated to the buffer ($B \times S_b \leq S_t$). It is important to note that the size of buffers increases as we traverse from the fastest to the slowest tier based on the assumption that the fastest tier is the most expensive storage with less amount of available storage space.

Proactive Data Prefetching and Scheduling. The manager is tightly integrated with TensorFlow's core and coordinates the dataset preparation and pre-processing, and ensures that each training batch is loaded into the memory before the next iteration is ready for execution. The manager gets information about the DL job from the *Workload Analyzer Module* and determines a schedule and deadlines for I/O operations to stage the required data in the main memory of the worker nodes. Initially, the dataset resides in a cold storage tier, e.g., network-attached storage which is mounted at each server. The manager locates the dataset and creates a schedule to ensure that the initial dataset for pre-processing phase is loaded into the main memory to minimize I/O stalls. The data prefetching

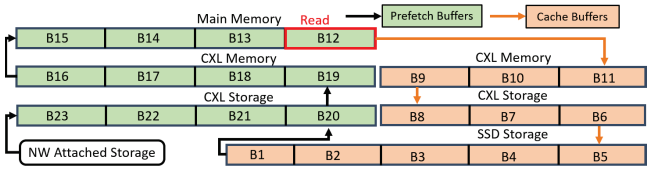

Figure 6: Dataflow for prefetching and caching in *DeepMemoryDL* using CXL-based memory and storage subsystem.

and caching approach in *DeepMemoryDL* is shown in Algorithm 1. Given the dataset, the target batch size, and available memory and storage space at each tier, it determines the initial location of each data batch and defines a prefetching and caching schedule. The schedule includes instructions to stage the pre-processed data in the memory subsystem. If the pre-processed data is larger than the available system memory, the additional data is cached in the CXL-based memory instead of slower local storage.

Figure 6 shows the flow of data to the prefetching and caching buffers. The prefetching schedule follows priority rules for storing the prefetched data. The priority is: 1) main memory; 2) CXL-based memory; 3) storage tier 0, i.e., CXL scratch storage; and 4) storage tier 1. The prefetching priority starts from the fastest storage tier and moves through the slower tiers. The prefetching schedule works in conjunction with the caching mechanism to coordinate data evictions to slower memory and storage tiers. The caching priority traverses the prefetching priority in reverse order to ensure that the required data is always available in the main memory. In the cases where the dataset, shard, or even a batch is much larger than the main memory, then the eviction coordination is halted and only prefetching buffers consume the entire memory and storage tiers. DL jobs can access the CXL-based memory directly to improve I/O performance despite the additional latency, which is lower than the latency to prefetch the data from local storage.

Allocation of CXL-based Memory. The manager is responsible for keeping track of CXL-based memory allocations on all worker nodes. The manager increases the memory allocation of a workload in chunks of 512 MB when a request for additional memory is received. This memory is allocated from the CXL-based memory space and allows the workload to avoid using the swap space after consuming the entire system memory. During the training phase, all worker nodes allocate memory to store the given model parameters and the training dataset. The available system memory on each worker varies depending upon the number of jobs running at a particular instance. This leads to memory starvation for DL jobs and slows down the training process. The manager ensures that enough memory is available for the model to grow over the specified training epochs.

Allocation of Fast Scratch Storage. The manager handles the allocation of fast scratch storage space over CXL-based storage. This is particularly critical for reducing the I/O wait due to slower storage tiers in the data processing workflow. Workloads can also benefit from using CXL-based storage for storing intermediate data when enough memory is not available to store the cached data. This is beneficial for typical DL workloads where the dataset is much larger than the available system memory and also when the processed data must be written back to the local storage.

3.3.4 Prefetcher Module. The *Prefetcher Module* is a part of the manager and worker components. It takes instructions from the *Core Module* and ensures that the data is prefetched and made available to the DL workloads before their execution. The schedule contains information about the worker nodes, assigned chunks of the dataset, memory and scratch space allocation, and a resource map for data placement. For extremely large datasets, the size of a single batch becomes substantially large causing the *Core Module* to define large buffer sizes and launch the prefetching threads ahead of schedule. The *Prefetcher Module* executes the prefetching schedule and reports the prefetching latency back to the *Core Module*. This information is used to dynamically adjust prefetching buffer sizes and the number of prefetching threads to further improve the I/O throughput. The *Prefetcher Module* tracks the memory footprint of each sample and the size of the entire training batch. This gives the *Prefetcher Module* the total size of a single prefetch block which is used to estimate the time it takes to prefetch a batch. For data transferred over the network, *DeepMemoryDL* incorporates the available link bandwidth and the latency to compute an estimated time to prefetch a given batch. This information is subsequently used to launch the prefetching threads and execute the prefetching schedule. The prefetching threads execute the schedule concurrently with the DL training job to ensure that the batch required in the next iteration is prefetched in the main memory.

The manager also defines a caching policy that is followed by all worker nodes to ensure quick access to the training data that is not available in the main memory. Figure 7 illustrates the caching policy of *DeepMemoryDL*. The policy uses the resource map provided by the *Prefetcher Module*. Once the main memory is fully used, the workers run the eviction policy to free up space from the main memory. The I/O buffers at each memory and storage tier hold the prefetched data. The size of I/O buffers are dynamic to incorporate the variations in the size of each batch. The manager defines and uses an eviction strategy to evict data from these buffers to make space for new data for the next iterations. The eviction strategy in *DeepMemoryDL* works closely with the TensorFlow training schedule and the *Prefetcher Module*. The policy is based on the following rules: 1) data is evicted in FIFO order; and 2) samples within a batch that are marked for prefetching will be cached to a lower memory tier. The eviction policy in *DeepMemoryDL* ensures that enough space remains available in the buffers of each tier and unnecessary expansion of a buffer is avoided at each tier. The worker nodes cache the evicted data to a lower memory and storage tier if the data is required by subsequent training iterations. The caching policy of *DeepMemoryDL* ensures that: 1) data is cached until the buffers are full; 2) the cached data is evicted in the first-in-first-out (FIFO) order; 3) data is always cached from a higher (faster) tier to a lower (slower) tier based on the prefetching schedule. The data that is needed first by the *Prefetcher Module* is kept in the CXL-based memory. Once the buffers in the CXL-based memory are full, the lower priority batches are cached into the storage tiers. *DeepMemoryDL* is more effective for workloads with high data reuse, such as DL jobs, due to prefetching and caching policies that ensure data to be prefetched is available in the fastest tier. However,

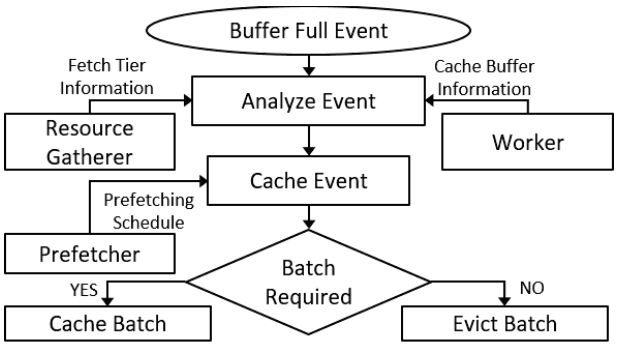

Figure 7: Caching workflow in *DeepMemoryDL*.

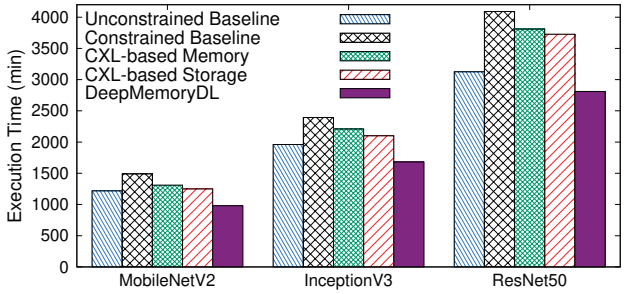

Figure 8: Total execution time of DL job with batch size of 64 and 3 epochs.

for workloads with low data re-use, the prefetcher ensures that the required data is available in local/CXL-based memory before it is required for processing. For such workloads, *DeepMemoryDL*'s caching policy will avoid aggressive caching to lower memory and storage tiers because data is not re-used by the workload.

By default, Linux operating system caches application data that is read from the local storage by assuming that the accessed data will soon be read again. The subsequent reads are served from the buffer cache instead of the disk to reduce I/O times. Once the memory footprint of a DL job grows and more memory is required, data from the buffer cache is evicted to fetch the required data. Thus, *DeepMemoryDL*'s caching policy is critical for keeping the required data in main memory as Linux caching policy will evict the data that is required in the next training iterations. *DeepMemoryDL*'s caching policies minimize reading the training batches from cold storage and improve the I/O throughput.

4 PERFORMANCE EVALUATION

In this section, we present the evaluation of *DeepMemoryDL* framework. We explain our experimental setup, the performance metrics, and the representative DL models that we use to evaluate the effectiveness of the proposed framework.

4.1 Evaluation Setup

Our evaluation setup consists of eight servers running Ubuntu 20.04 LTS server operating systems. Each server has two 2.40 GHz Intel Xeon Gold 6240R processors, with 192 GB main memory, out of which 96 GB of the main memory is reserved for emulating CXL-based memory and CXL-based storage scenarios, and 10 Gbps Ethernet between servers. We emulate the provisioning of CXL-based memory by allocating memory from the remote NUMA domain and CXL-based storage devices by creating a RAMDisk [16] on the remote NUMA domain.

In our evaluation, we stress the memory and storage subsystem with large datasets. Smaller batch sizes and models can be fit into the server's memory, however, our evaluation focuses on realistic scenarios where HPC servers have less memory available to store the entire dataset. To evaluate *DeepMemoryDL*, we use ImageNet [13] dataset and ResNet50 [20], Inception-V3 [40], and MobileNetV2 [21] models. We use Intel PCM [3], and `sysstat` [6] to monitor the memory, disk, swap, NUMA domains, and network activity during the execution of DL workloads. We develop a memory hogger to hog system memory on the worker nodes to mimic the behavior of background jobs in production data centers. We investigate the impact of limited system memory, the availability of CXL-based memory and storage tiers, and the impact of proactive prefetching and caching on the performance of DL jobs. Existing state-of-the-art prefetching and caching approaches [15, 29] are either developed for a single server or do not incorporate the characteristics of additional memory and storage tiers, specifically, CXL-based devices. Therefore, we analyze the performance of *DeepMemoryDL* using five realistic TensorFlow environments depending on the availability of memory and storage subsystems. These environments are:

(1) ***Unconstrained Baseline Environment*:** This environment represents an ideal scenario with no resource constraints or sharing between DL jobs.
(2) ***Constrained Baseline Environment*:** This environment represents a realistic scenario where resources are shared and limited memory is available for DL jobs.
(3) ***CXL-based Storage Environment*:** This environment has CXL-based storage available with limited system memory.
(4) ***CXL-based Memory Environment*:** This environment has CXL-based memory available with limited system memory.
(5) ***CXL-based Memory and Storage Environment*:** This environment has CXL-based memory and CXL-based storage available with limited system memory.
(6) ***DeepMemoryDL*:** This environment has our proposed framework, *DeepMemoryDL*, integrated with TensorFlow, which manages system resources including CXL-based memory and CXL-based storage to run DL jobs.

4.2 Performance Results

In this section, we execute DL workloads on the studied execution environments and compare their performance. We repeat each experiment five times and report the average execution time for each experiment. Overall, we experience negligible variance, i.e., less than 2%, between different executions of the same experiment.

4.2.1 Total Execution Time of the DL Job. We evaluate the effectiveness of *DeepMemoryDL* in reducing the overall training time and compare it with the studied environments. Figure 8 shows the performance comparison of *Unconstrained Baseline Environment*, *Constrained Baseline Environment*, *CXL-based Memory Environment*,

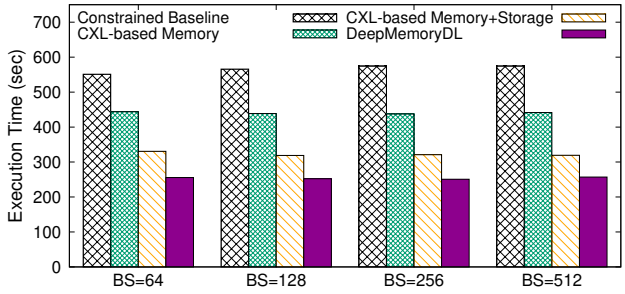

Figure 9: Data pre-processing time for ImageNet dataset with limited main memory.

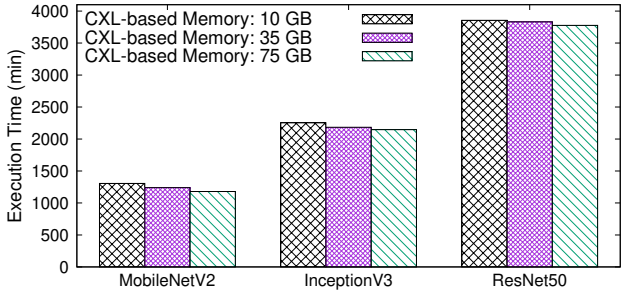

Figure 10: Impact of CXL-based memory allocation on DL job with batch size of 64 and 3 epochs.

CXL-based Storage Environment, and *DeepMemoryDL* when training a DL job over 3 epochs with a batch size of 64. For all models, we observe that *Constrained Baseline Environment* takes the longest time due to limited memory availability on worker nodes and excessive swapping of pages to the underlying SSD-based storage. The *CXL-based Memory Environment* enables DL jobs to train using a larger working set by allowing access to CXL-based memory, however, the additional latency of the CXL-based memory increases the training time by 14% on average as compared to the *Unconstrained Baseline Environment*. The *CXL-based Storage Environment* provides fast storage space to DL jobs to read the input data from CXL-based storage which increases the performance significantly over reading data from the SSD. However, due to the limited system memory, it increases the training time by 10% on average as compared to the *Unconstrained Baseline Environment* but reduces the training time by 9% as compared to the *Constrained Baseline Environment*. Overall, we observe that *DeepMemoryDL* reduces the training time by up to 20%, 34%, 27%, and 25% as compared to the *Unconstrained Baseline Environment*, *Constrained Baseline Environment*, *CXL-based Memory Environment*, and *CXL-based Storage Environment*, respectively. The performance improvement of *DeepMemoryDL* is attributed to the allocation of CXL-based resources, prefetching of data batches to main memory, caching data into CXL-based storage instead of the underlying SSD-based storage, and tailored data eviction policies.

4.2.2 Data Pre-Processing Phase. Optimizing the performance of the input data pipeline is crucial to the performance of DL jobs. In this section, we evaluate the impact of *DeepMemoryDL* on pre-processing stage by pre-processing 40 GB of images from the ImageNet dataset. The pre-processing phase consisted of downloading, extracting, generating training and validation data, shuffling, and reshaping images. Figure 9 shows the results. Before pre-processing, *DeepMemoryDL* proactively prefetches the input dataset to CXL-based storage to significantly reduce loading time and then caches the data in CXL-based memory to move data closer to compute threads allowing for faster prefetching onto main memory. These policies defined by the *Core Module* of *DeepMemoryDL* yield better I/O performance as compared to *Constrained Baseline Environment* and reduce overall data pre-processing time by 56%, 43%, and 23% as compared to *Constrained Baseline Environment*, *CXL-based Memory Environment*, and *CXL-based Memory and Storage Environment*, respectively. The impact of *DeepMemoryDL* is further observed

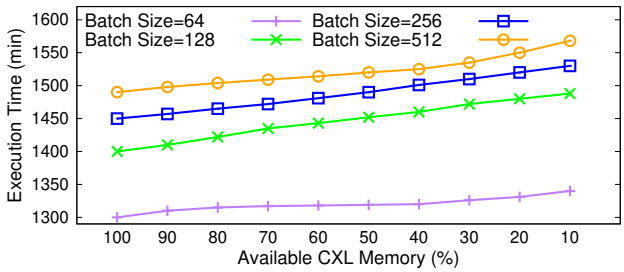

Figure 11: Impact of CXL-based memory on total execution time of DL job with 3 epochs.

while training a model with the pre-processed data as discussed in Section 4.2.1 as data batches are prefetched to reduce the training time for subsequent iterations. We note that the experiment results shown in Figure 9 only involve data pre-processing, therefore, the change in the batch size does not impact pre-processing time.

4.2.3 Impact of Using CXL-based memory on DL Job. Large memory systems allow DL jobs to train large models and datasets without running out of system resources. The scarcity of available memory leads to premature termination of DL job resulting in loss of computation. In such a scenario, a DL job is either restarted or resumed from the last checkpoint. We conduct experiments with varying CXL-based memory allocated to the DL job to study the impact on the execution time for the studied models using a batch size of 64. Figure 10 shows the results. As the memory footprint of DL jobs increases, *DeepMemoryDL* allocates CXL-based memory to expand its working set size to the CXL-based memory. We observe that as the CXL-based memory footprint of a DL job grows, the total training time is reduced due to the usage of a faster memory tier as compared to the SSD-based swap storage.

We evaluated the impact of available CXL-based memory and dynamic buffer sizes on the total execution time of DL jobs. Figure 11 shows that as the buffer size increases, the execution time reduces due to the increased prefetching and caching capacity at the CXL-based memory tier. We observe that as the batch size increases the execution time increases proportionally, however, *DeepMemoryDL* adjusts the I/O buffer sizes based on the footprint of a data batch for prefetching and caching. *DeepMemoryDL* mitigates the impact

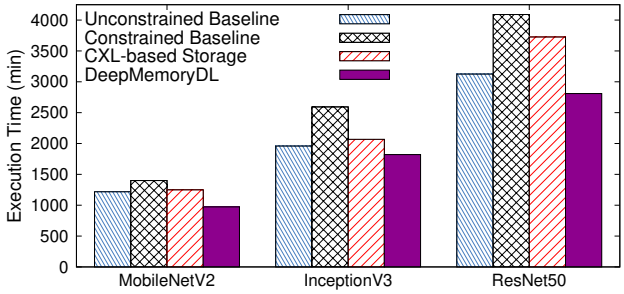

Figure 12: Impact of data staging storage on DL job with batch size of 64 and 3 epochs.

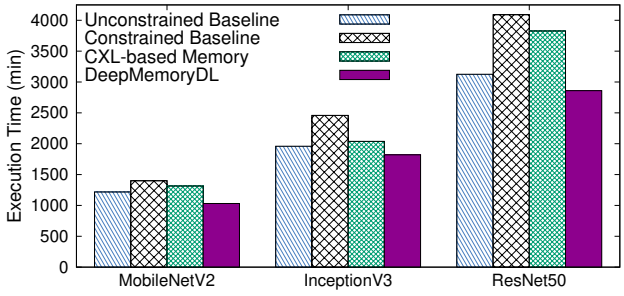

Figure 13: Impact of proactive prefetching and caching on DL job with batch size of 64 and 3 epochs.

of using large batch sizes on the execution time of a DL job and also enables TensorFlow to manage a much larger working set size.

4.2.4 Impact for Using CXL-based Storage for Staging Data on DL Job. Staging data in a fast storage tier improves I/O performance for DL jobs that read and write data to local storage devices, e.g., HDD or SSD. We conduct experiments to study the impact of using CXL-based storage on read and write operations for staging large datasets. Figure 12 shows the results. We observe that *CXL-based Storage Environment* reduces the execution time by up to 30% as compared to the *Constrained Baseline Environment* due to the improved I/O performance of the CXL-based storage. Overall, *DeepMemoryDL* reduces the execution time by up to 20%, 32%, and 24% as compared to the *Unconstrained Baseline Environment*, *Constrained Baseline Environment*, and *CXL-based Storage Environment*, respectively. *DeepMemoryDL* stages data in the CXL-based storage resulting in improved performance as compared to the other execution environments since storing the entire dataset in the staging area ideally yields higher read bandwidth and IOPS.

4.2.5 Impact of Proactive Data Prefetching and Caching on DL Job. Proactive prefetching of data minimizes I/O wait during model training. We study the impact of proactive data prefetching and caching on the execution time of DL jobs with a batch size of 64 and compare the results of *DeepMemoryDL* with the other execution environments. Figure 13 shows the performance variation while training MobileNetV2, InceptionV3, and ResNet50 models using the ImageNet dataset over 3 epochs. We observe that by effectively

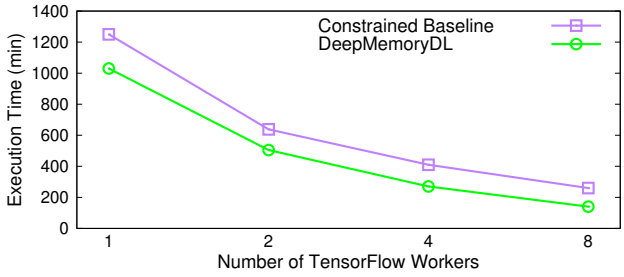

Figure 14: Scalability analysis on total execution time of DL job with batch size of 64 and 3 epochs.

managing the CXL-based memory, *DeepMemoryDL* reduces the execution time of DL jobs by 15%, 30%, 25% as compared to the *Unconstrained Baseline Environment*, *Constrained Baseline Environment*, and *CXL-based Memory Environment*, respectively. Moreover, the *Unconstrained Baseline Environment* performs better than the *CXL-based Memory Environment* when enough space is available in the main memory by an average overhead of up to 10%. The *CXL-based Memory Environment* provides additional CXL-based memory to DL jobs that reduces the execution time by up to 17% as compared to the *Constrained Baseline Environment*. Data eviction contributed to the worst performance of the *Constrained Baseline Environment*. The manager in *DeepMemoryDL* optimizes the caching mechanism by preparing a caching policy that is implemented on all worker nodes running the training job. Additional memory is pooled from CXL-based memory to cache processed data to ensure that the required data is always available in the faster available tier. In tandem with prefetching, caching improved the performance of *DeepMemoryDL* and optimized memory resource utilization.

4.2.6 Scalability Analysis of DeepMemoryDL *on DL Job Performance.* We ran several experiments using a combination of real-world use cases by varying the available system resources and the number of worker nodes to evaluate the performance of *DeepMemoryDL*. Single worker training is resource-intensive since the entire dataset has to be processed on a single node requiring more memory and storage resources. Multi-worker training requires the dataset to be divided between the available worker nodes reducing the resource requirement on each worker node. However, for large datasets, memory and I/O remain the bottlenecks. We studied the impact of increasing the number of workers on *DeepMemoryDL* by training MobileNetV2 and compare its execution time with *Constrained Baseline Environment*. Figure 14 shows the result. We observe that *DeepMemoryDL* outperforms the *Constrained Baseline Environment*, and the respective performance gap remains similar as we increase the number of worker nodes. Moreover, we observe that the performance of a DL job is improved linearly using *DeepMemoryDL* as we increase the number of worker nodes.

5 RELATED WORK

Many recent efforts have focused on improving the I/O performance of the TensorFlow platform. In this section, we provide an overview of efforts that are closely related to the contributions of this paper.

Several other efforts have been made to optimize the I/O path in DL, such as, [12] which studies the impact of the BeeGFS filesystem on DL workload performance. NoPFS [15] predicts data access patterns and performs prefetching and caching based on these patterns. It provides a distributed caching policy using local and distributed memory to improve the I/O performance of DL jobs. However, NoPFS does not support CXL-based memory or storage devices that introduce additional tiers in the memory and storage layer. In [10], the authors study the impact of multi-threading on the I/O pipelines on improving the performance of DL jobs. Recent efforts also explore optimizing data loading in the I/O pipelines and pre-processing to accelerate DL applications by utilizing Nvidia's Data Loading Library (DALI) [4]. Prisma [29] decouples storage I/O optimization using software-defined storage that is composed of a control plane that maintains user-defined caching policies and a data plane that implements parallel data prefetching. However, it does not leverage CXL-based memory and storage devices.

Informed Prefetching Data Loader (IPDL) [37] prefetches data from remote data stores to reduce the I/O wait times in PyTorch based DL and edge computing environments. Similarly, in [25] the authors employ caching and prefetching techniques to improve the performance of DL training in cloud environments. Quiver [27] is an informed storage cache designed to improve the performance of DL jobs on GPU-enabled clusters using secure hash-based addressing to reuse cached data across jobs and avoid cache thrashing. The approaches are tailored for GPUs and cannot be applied to all stages of DL jobs. Moreover, they do not explore the use of CXL-based memory and storage tiers. PreFAM [24] improves the performance of fabric-attached memory architectures by predicting future data access and prefetching data blocks from fabric-attached memories to node-local memory resulting in improved access latency. While this approach is similar to ours in leveraging the latest advancements in memory subsystem to provide additional memory and optimize data access, however, our approach reduces the uncertainty in predicting future data access and improves the accuracy of prefetching by integrating with the DL frameworks. We also propose a caching mechanism to maintain prefetched blocks that will be accessed in the future in the closest memory tier.

Distributed remote memory accesses can be performed by using fast low latency networks and protocols involving RDMA, NVMEoF [19] and SEMERU [42]. Remote memory paging system over RDMA called Infiniswap [18] that provides memory disaggregation. In [23], the authors use prefetching over NVRAM and DRAM to bridge the I/O gap between hard disk to RAM. RAMCloud [35] aggregates server memories into a single coherent key-value store and provides low-latency access to large-scale datasets enabling faster access to large datasets for various applications including DL workloads. Fanstore [46] provides a runtime file system to optimize DL I/O on existing hardware and software architecture by distributing datasets to all compute nodes, and maintains a global namespace. DLFS [47] provides I/O services on top of an emerging industrial standard NVMeOF leveraging storage disaggregation.

Previous efforts have explored different aspects of DL I/O to improve training and optimize input pipelines by introducing middleware, runtimes, file system abstractions, and utilizing caching and prefetching techniques. However, these efforts do not holistically incorporate multiple memory and storage tiers and do not leverage emerging technologies to optimize the data pre-processing and input pipelines in DL platforms. In this paper, we propose a holistic framework that improves the performance of DL workloads by incorporating and utilizing CXL-based memory and storage devices in the TensorFlow platform.

6 CONCLUSION

Modern deep learning (DL) applications are resource-intensive and have dynamic compute, memory, and storage requirements. The resource requirements of these workloads typically exceed the hardware resources available on the latest high-performance computing (HPC) servers. Currently, popular DL platforms do not incorporate I/O characteristics of the underlying memory and storage subsystems. In this paper, we present a framework, *DeepMemoryDL*, that improves the performance of DL workloads by efficiently leveraging storage and memory tiers to proactively prefetch and cache the training data. We emulate the latest advancement in memory subsystems, i.e., Compute Express Link (CXL), to provide additional memory and fast scratch storage space to DL workloads and reduce the overall training time. Moreover, we integrate *DeepMemoryDL* with the popular TensorFlow platform to improve the performance of its I/O requests for data pre-processing and training stages. Our evaluation using representative DL workloads shows that *DeepMemoryDL* reduces the overall training time of a DL job by up to 34% and 27% as compared to the default TensorFlow and CXL-based memory expansion approaches, respectively. In our future work, we will extend *DeepMemoryDL* to support other DL platforms, specifically the PyTorch platform, to improve its performance by eliminating I/O stalls. Moreover, we will explore accelerator-based systems, specifically GPUs, for utilizing their High Bandwidth Memory (HBM) interconnects to provide an additional storage tier for prefetching and caching the training data and improving the performance of DL workloads in distributed settings.

ACKNOWLEDGMENTS

This work is supported in part by the National Science Foundation under Award 2106635. Results presented in this paper are obtained using the Chameleon and CloudLab testbeds supported by the National Science Foundation.

REFERENCES

[1] 2022. FIO - flexible i/o tester synthetic benchmark. https://github.com/axboe/fio
[2] 2022. Gen-Z. https://genzconsortium.org/
[3] 2022. Intel PCM. https://www.intel.com/content/www/us/en/developer/articles/technical/performance-counter-monitor.html
[4] 2022. NVIDIA Data Loading Library (DALI). https://developer.nvidia.com/DALI
[5] 2022. OakRidge Learning Computing Facility (OLCF). https://www.olcf.ornl.gov/
[6] 2022. SYSSTAT. http://sebastien.godard.pagesperso-orange.fr/
[7] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. 2016. Tensorflow: A system for large-scale machine learning. In *Proceedings of the 12th USENIX symposium on operating systems design and implementation (OSDI)*. USENIX Association, 265–283.
[8] Moiz Arif, M. Mustafa Rafique, Seung-Hwan Lim, and Zaki Malik. 2020. Infrastructure-Aware TensorFlow for Heterogeneous Datacenters. In *Proceedings of the 28th IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*. IEEE, 1–8. https://doi.org/10.1109/MASCOTS50786.2020.9285969
[9] Kevin Assogba, Moiz Arif, M. Mustafa Rafique, and Dimitrios S. Nikolopoulos. 2022. On Realizing Efficient Deep Learning Using Serverless Computing. In *Proceedings of the 22nd IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*. IEEE.

[10] Steven W. D. Chien, Stefano Markidis, Chaitanya Prasad Sishtla, Luis Santos, Pawel Herman, Sai Narasimhamurthy, and Erwin Laure. 2018. Characterizing Deep-Learning I/O Workloads in TensorFlow. In *Proceedings of the 3rd IEEE/ACM International Workshop on Parallel Data Storage & Data Intensive Scalable Computing Systems (PDSW-DISCS)*. IEEE, 54–63. https://doi.org/10.1109/PDSW-DISCS.2018.00011

[11] Zeshan Chishti and Berkin Akin. 2019. Memory System Characterization of Deep Learning Workloads. In *Proceedings of the International Symposium on Memory Systems (MEMSYS)* (Washington, District of Columbia, USA). ACM, New York, NY, USA, 497–505. https://doi.org/10.1145/3357526.3357569

[12] Fahim Chowdhury, Yue Zhu, Todd Heer, Saul Paredes, Adam Moody, Robin Goldstone, Kathryn Mohror, and Weikuan Yu. 2019. I/O Characterization and Performance Evaluation of BeeGFS for Deep Learning. In *Proceedings of the 48th ACM International Conference on Parallel Processing (ICPP)* (Kyoto, Japan). ACM, New York, NY, USA, Article 80, 10 pages. https://doi.org/10.1145/3337821.3337902

[13] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. 2009. ImageNet: A large-scale hierarchical image database. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. IEEE, 248–255. https://doi.org/10.1109/CVPR.2009.5206848

[14] Li Deng and Dong Yu. 2014. Deep Learning: Methods and Applications. *Foundations and Trends in Signal Processing* 7, 3–4 (2014), 197–387. https://doi.org/10.1561/2000000039

[15] Nikoli Dryden, Roman Böhringer, Tal Ben-Nun, and Torsten Hoefler. 2021. Clairvoyant Prefetching for Distributed Machine Learning I/O. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)* (St. Louis, Missouri). ACM, New York, NY, USA, Article 92, 15 pages. https://doi.org/10.1145/3458817.3476181

[16] Michail D Flouris and Evangelos P Markatos. 1999. The network RamDisk: Using remote memory on heterogeneous NOWs. *Cluster computing* 2, 4 (1999), 281–293.

[17] Gaddisa Olani Ganfure, Chun-Feng Wu, Yuan-Hao Chang, and Wei-Kuan Shih. 2020. DeepPrefetcher: A Deep Learning Framework for Data Prefetching in Flash Storage Devices. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 39, 11 (2020), 3311–3322. https://doi.org/10.1109/TCAD.2020.3012173

[18] Juncheng Gu, Youngmoon Lee, Yiwen Zhang, Mosharaf Chowdhury, and Kang G. Shin. 2017. Efficient Memory Disaggregation with Infiniswap. In *Proceedings of the 14th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. USENIX Association, Boston, MA, 649–667.

[19] Zvika Guz, Harry (Huan) Li, Anahita Shayesteh, and Vijay Balakrishnan. 2017. NVMe-over-Fabrics Performance Characterization and the Path to Low-Overhead Flash Disaggregation. In *Proceedings of the 10th ACM International Systems and Storage Conference (SYSTOR)* (Haifa, Israel). ACM, New York, NY, USA, Article 16, 9 pages. https://doi.org/10.1145/3078468.3078483

[20] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep Residual Learning for Image Recognition. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. IEEE.

[21] Andrew G. Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. 2017. MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications. *CoRR* abs/1704.04861 (2017). arXiv:1704.04861

[22] Jiawei Jiang, Bin Cui, Ce Zhang, and Lele Yu. 2017. Heterogeneity-Aware Distributed Parameter Servers. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD)* (Chicago, Illinois, USA). ACM, New York, NY, USA, 463–478. https://doi.org/10.1145/3035918.3035933

[23] Wenbin Jiang, Pai Liu, Hai Jin, and Jing Peng. 2020. An Efficient Data Prefetch Strategy for Deep Learning Based on Non-volatile Memory. In *Green, Pervasive, and Cloud Computing*, Zhiwen Yu, Christian Becker, and Guoliang Xing (Eds.). Springer, Cham, 101–114.

[24] Vamsee Reddy Kommareddy, Jagadish Kotra, Clayton Hughes, Simon David Hammond, and Amro Awad. 2020. PreFAM: Understanding the Impact of Prefetching in Fabric-Attached Memory Architectures. In *Proceedings of the ACM International Symposium on Memory Systems (MEMSYS)* (Washington, DC, USA). ACM, New York, NY, USA, 323–334. https://doi.org/10.1145/3422575.3422804

[25] Nicholas Krichevsky, Renee St Louis, and Tian Guo. 2021. Quantifying and Improving Performance of Distributed Deep Learning with Cloud Storage. In *Proceedings of the IEEE International Conference on Cloud Engineering (IC2E)*. IEEE, 99–109. https://doi.org/10.1109/IC2E52221.2021.00024

[26] Andreas Krisilias, Nikodimos Provatas, Nectarios Koziris, and Ioannis Konstantinou. 2021. A Performance Evaluation of Distributed Deep Learning Frameworks on CPU Clusters Using Image Classification Workloads. In *Proceedings of the IEEE International Conference on Big Data (Big Data)*. IEEE, 3085–3094. https://doi.org/10.1109/BigData52589.2021.9671461

[27] Abhishek Vijaya Kumar and Muthian Sivathanu. 2020. Quiver: An Informed Storage Cache for Deep Learning. In *Proceedings of the 18th USENIX Conference on File and Storage Technologies (FAST)*. USENIX Association, Santa Clara, CA, 283–296.

[28] Sang Il Lee and Seong Joon Yoo. 2020. Multimodal deep learning for finance: integrating and forecasting international stock markets. *The Journal of Supercomputing* 76, 10 (2020), 8294–8312.

[29] Ricardo Macedo, Cláudia Correia, Marco Dantas, Cláudia Brito, Weijia Xu, Yusuke Tanimura, Jason Haga, and João Paulo. 2021. The Case for Storage Optimization Decoupling in Deep Learning Frameworks. In *Proceedings of the IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE, 649–656. https://doi.org/10.1109/Cluster48925.2021.00096

[30] Ruben Mayer and Hans-Arno Jacobsen. 2020. Scalable Deep Learning on Distributed Infrastructures: Challenges, Techniques, and Tools. *ACM Comput. Surv.* 53, 1, Article 3 (feb 2020), 37 pages. https://doi.org/10.1145/3363554

[31] John D McCalpin et al. 1995. Memory bandwidth and machine balance in current high performance computers. *IEEE computer society technical committee on computer architecture (TCCA) newsletter* 2, 19-25 (1995).

[32] Larry W McVoy, Carl Staelin, et al. 1996. lmbench: Portable Tools for Performance Analysis.. In *Proceedings of the USENIX annual technical conference*. San Diego, CA, USA, USENIX Association, 279–294.

[33] Derek Gordon Murray, Jiri Simsa, Ana Klimovic, and Ihor Indyk. 2021. tf.data: A Machine Learning Data Processing Framework. *CoRR* abs/2101.12127 (2021). arXiv:2101.12127

[34] Bogdan Nicolae, Justin M. Wozniak, Matthieu Dorier, and Franck Cappello. 2020. DeepClone: Lightweight State Replication of Deep Learning Models for Data Parallel Training. In *Proceedings of the IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE, 226–236. https://doi.org/10.1109/CLUSTER49012.2020.00033

[35] John Ousterhout, Arjun Gopalan, Ashish Gupta, Ankita Kejriwal, Collin Lee, Behnam Montazeri, Diego Ongaro, Seo Jin Park, Henry Qin, Mendel Rosenblum, Stephen Rumble, Ryan Stutsman, and Stephen Yang. 2015. The RAMCloud Storage System. *ACM Trans. Comput. Syst.* 33, 3, Article 7 (aug 2015), 55 pages. https://doi.org/10.1145/2806887

[36] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. 2019. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In *Proceedings of the Advances in Neural Information Processing Systems*, Vol. 32. Curran Associates, Inc.

[37] Xiaojun Ruan and Haiquan Chen. 2021. Informed Prefetching in I/O Bounded Distributed Deep Learning. In *Proceedings of the IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE, 850–857. https://doi.org/10.1109/IPDPSW52791.2021.00127

[38] Andrew W Senior, Richard Evans, John Jumper, James Kirkpatrick, Laurent Sifre, Tim Green, Chongli Qin, Augustin Žídek, Alexander WR Nelson, Alex Bridgland, et al. 2020. Improved protein structure prediction using potentials from deep learning. *Nature* 577, 7792 (2020), 706–710.

[39] Prateek Shantharama, Akhilesh S. Thyagaturu, and Martin Reisslein. 2020. Hardware-Accelerated Platforms and Infrastructures for Network Functions: A Survey of Enabling Technologies and Research Studies. *IEEE Access* 8 (2020), 132021–132085. https://doi.org/10.1109/ACCESS.2020.3008250

[40] Christian Szegedy, Vincent Vanhoucke, Sergey Ioffe, Jon Shlens, and Zbigniew Wojna. 2016. Rethinking the Inception Architecture for Computer Vision. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. IEEE.

[41] S. Van Doren. 2019. Abstract - HOTI 2019: Compute Express Link. In *Proceedings of the IEEE Symposium on High-Performance Interconnects (HOTI)*. IEEE, 18–18. https://doi.org/10.1109/HOTI.2019.00017

[42] Qing Wang, Youyou Lu, and Jiwu Shu. 2022. Sherman: A Write-Optimized Distributed B+Tree Index on Disaggregated Memory. In *Proceedings of the International Conference on Management of Data (SIGMOD)* (Philadelphia, PA, USA). ACM, New York, NY, USA, 1033–1048. https://doi.org/10.1145/3514221.3517824

[43] Wei Wang, Jack W. Davidson, and Mary Lou Soffa. 2016. Predicting the memory bandwidth and optimal core allocations for multi-threaded applications on large-scale NUMA machines. In *Proceedings of the IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 419–431. https://doi.org/10.1109/HPCA.2016.7446083

[44] Wei Wang, Meihui Zhang, Gang Chen, H. V. Jagadish, Beng Chin Ooi, and Kian-Lee Tan. 2016. Database Meets Deep Learning: Challenges and Opportunities. *SIGMOD Rec.* 45, 2 (sep 2016), 17–22. https://doi.org/10.1145/3003665.3003669

[45] Yuan Wang, Dongxiang Zhang, Ying Liu, Bo Dai, and Loo Hay Lee. 2019. Enhancing transportation systems via deep learning: A survey. *Transportation Research Part C: Emerging Technologies* 99 (2019), 144–163. https://doi.org/10.1016/j.trc.2018.12.004

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

[47] Yue Zhu, Weikuan Yu, Bing Jiao, Kathryn Mohror, Adam Moody, and Fahim Chowdhury. 2019. Efficient User-Level Storage Disaggregation for Deep Learning. In *Proceedings of the IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE, 1–12. https://doi.org/10.1109/CLUSTER.2019.8891023