Dataset Placement and Data Loading Optimizations for Cloud-Native Deep Learning Workloads

Zhuangwei Kang, Ziran Min, Shuang Zhou, Yogesh D. Barve, Aniruddha Gokhale
Vanderbilt University, Nashville, TN 37212, USA
Email: {zhuangwei.kang; ziran.min; shuang.zhou; yogesh.d.barve; a.gokhale}@vanderbilt.edu

***Abstract*—The primary challenge facing cloud-based deep learning systems is the need for efficient orchestration of large-scale datasets with diverse data formats and provisioning of high-performance data loading capabilities. To that end, we present DLCache, a cloud-native dataset management and runtime-aware data-loading solution for deep learning training jobs. DLCache supports the low-latency and high-throughput I/O requirements of DL training jobs using cloud buckets as persistent data storage and a dedicated computation cluster for training. DLCache comprises four layers: a control plane, a metadata plane, an operator plane, and a multi-tier storage plane, which are seamlessly integrated with the Kubernetes ecosystem thereby providing ease of deployment, scalability, and self-healing. For efficient memory utilization, DLCache is designed with an on-the-fly and best-effort caching mechanism that can auto-scale the cache according to runtime configurations, resource constraints, and training speeds. DLCache considers both frequency and freshness of data access as well as data preparation costs in making effective cache eviction decisions that result in reduced completion time for deep learning workloads. Results of evaluating DLCache on the Imagenet-ILSVRC and LibriSpeech datasets under various runtime configurations and simulated GPU computation time experiments showed up to a 147.49% and 156.67% improvement in data loading throughput, respectively, compared to the popular PyTorch framework.**

***Index Terms*—Deep Learning Training, Cache System, Cloud-native, Data Management, System Software**

I. Introduction

With the rapid development of heterogeneous computational devices and accelerators (e.g., GPU, FPGA, and TPU), computation is no longer the primary source of bottleneck for Deep Learning Training (DLT) tasks. However, these DL models must be trained using massive datasets, whose storage requirements can easily far exceed the storage capacity of individual physical disks. A network-based file system (e.g., NFS, HDFS) is a common solution to eliminate the need for large, centralized disk space and reduce storage costs. Unfortunately, contemporary distributed file systems are not designed to handle DL datasets and workloads, thus requiring users to make their own, *ad hoc* dataset placement and eviction decisions based on available resources and resource profiles of the DL workloads.

Alternatively, a centralized cloud bucket storage (e.g., AWS S3, Azure Blob, Alibaba Cloud OSS) provides on-demand data storage and access services. The cost of using cloud bucket storage is calculated based on the usage of storage space, API requests, and data transfer volumes. Since training a DL model needs to traverse the dataset multiple times and as every access to a data element corresponds to an API request, this cost could become prohibitively expensive (e.g., $\$4e^{-3}/1k$ requests for AWS S3 [*], and $\$1.9e^{-3}/1k$ read operations for Azure Blob [†]) particularly when the model needs to be trained on millions of small files and that too for several epochs.

Furthermore, the constraints on the API request rate enforced by the cloud providers limit the use of cloud buckets as direct-access storage facilities for local DLT jobs. For instance, AWS S3 supports a request rate of 5,500 GET requests per second per prefix in a bucket, whereas the constraint is 500 requests per second for a blob in Azure Blob. To ensure models can directly learn from datasets in buckets, data providers are forced to preprocess data files into a dedicated input format. It is also time-consuming and expensive to upload a massive number of small files to a bucket.

More importantly, since DL algorithms must wait for data to be available in GPU memory before training, the I/O bottleneck and inherent network latency in network-based file systems and cloud buckets can easily result in severe data stalls. As suggested in [1], since I/O takes as much as 85% of DL training time, ensuring efficient and robust data ingestion from data sources to DL applications so that the computational device can always be in a saturation state becomes the fundamental challenge. Motivated by the need for low-latency data access and high I/O throughput in memory, some efforts [2–5] build distributed cache systems by aggregating memory from a set of machines. These systems clone data from external storage and cache the raw or pre-processed data in RAM. Data elements are retrieved via local or remote cache hits during training. However, given that the memory space is far smaller than disk, building a cluster that can accommodate large-scale datasets purely in memory is rather expensive.

Widely used DL frameworks (e.g., PyTorch) leverage multi-process (i.e., workers) data-loading and prefetching mechanisms to reduce GPU waiting time. For these frameworks, I/O performance and concurrency level are the primary factors affecting training time, which can be improved using in-memory data storage. To that end, our investigations reveal that only those batches that the data-loading workers will load into memory in the near future must be cached. Further, the DLT task can run smoothly if the cache is refreshed on-the-fly based

979-8-3503-3902-4/23/$31.00 ©2023 IEEE

[*] https://aws.amazon.com/s3/pricing/

[†] https://azure.microsoft.com/en-us/pricing/details/storage/blobs/

on data access patterns. However, current DL frameworks and cache systems require users to decide the number of data-loading workers prior to training, which might degrade performance or waste memory under improper settings.

To address the challenges and limitations in current work, we present DLCache, a dataset management and runtime-aware data-loading solution that supports low-latency and high-throughput I/O requirements of DL training jobs in the setting where users utilize cloud buckets as persistent data storage and a dedicated computation cluster for training. DLCache is a cloud-native system that seamlessly integrates with the Kubernetes (K8s) ecosystem, allowing users and system maintainers to easily deploy, update, scale, and self-heal components.

DLCache was evaluated on the Imagenet-ILSVRC and LibriSpeech datasets using a hybrid of 3 batch size configurations and 6 simulated computation time settings. Empirical results demonstrate that compared to the PyTorch framework, DLCache exhibits a substantial improvement in data loading throughput, reaching up to 147.49% and 49.62% improvement for Imagenet-ILSVRC when there are 4 and 8 data-loading workers, respectively, and the presence of data stall. For the LibriSpeech dataset, DLCache achieved up to 156.67% and 39.70% improvement respectively under similar conditions. Due to the minor overhead of DLCache, when there is no data stall, DLCache's performance was found to be lower than that of PyTorch, with a maximum deviation of 3.03% and 7.74% for Imagenet-ILSVRC, and 0.12% and 5.89% for LibriSpeech.

In summary, this paper makes the following contributions:

1) Describe DLCache, a DL dataset orchestration system that enables efficient deployment and high-performance data loading for cloud-native DLT jobs. It uses a novel three-tier storage system and multiple data services, such as dataset preparation, placement, and a cost-aware eviction of the least recently/frequently used (CLRFU) data eviction strategy.
2) Present the design of a K8s Custom Resource Definition (CRD) controller and operator to streamline the DevOps process for DLT jobs in the cloud, while hiding the complexities of the underlying data services from user applications. This approach enables users to define DLT jobs in a descriptive language, such as YAML or JSON, following a pre-defined schema, and execute them in a special K8s pod we designed called DLTPod.
3) Analyze and identify the causes of PyTorch data stalls and implement a runtime-aware best-effort caching mechanism to mitigate the waiting time. Through its Cache Worker, the DLTPod monitors training processes and loads data dynamically from NFS into its in-memory file system, TMPFS.
4) Enhancement to the PyTorch DataLoader to dynamically determine and adjust the number of DataLoader Workers based on real-time training performance and data retrieval latency thereby improving memory utilization.

The rest of the paper is organized as follows: Section II provides background information; Section III describes the design of DLCache; Section IV provides results of empirically evaluating DLCache; Section V compares DLCache with related work; and finally Section VI provides concluding remarks alluding to future work.

II. Background

A. Deep Learning Training and Data Loading

In Deep Learning Training (DLT) jobs, a complex neural network model is trained by iterating through a dataset multiple time-scaled epochs. The dataset is divided into fix-sized mini-batches for each iteration, and the model updates its parameters through stochastic gradient descent [6] to minimize a loss function. Computation-intensive operations are performed using GPUs and accelerators, while CPUs handle data loading and preprocessing. DLT jobs can be categorized into supervised and unsupervised learning based on whether the data has labels or not. In supervised learning, the $\langle sample, label \rangle$ pairs are created by retrieving data and labels from an input file or a manifest file, and sent to the model using a custom loading function or a DL framework's utility. This paper focuses on the widely-used PyTorch DL framework [7], specifically the DataLoader [8] shown in Figure 1.

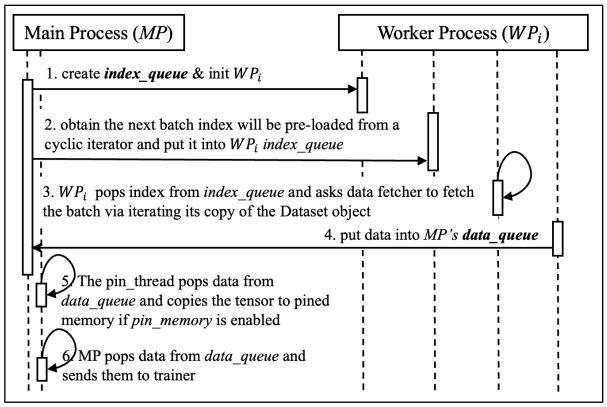


Fig. 1: PyTorch DataLoader Working Principle

To use the PyTorch DataLoader, users must create a subclass of the Dataset class where their dataset is stored. This subclass must implement the $\_\_getitem\_\_$ function, which returns a trainable $\langle sample, label \rangle$ tuple for a specific index in the dataset. The DataLoader is then initialized with this Dataset object. During the initialization process, the data item indexes are shuffled and divided into batches, and an iterator is created to iterate over these batch indexes and load the data. The DataLoader can load the data either sequentially in the main process or in parallel using multiple sub-processes (workers), depending on the DataLoader configuration.

The multi-process DataLoader from Figure 1 works as follows: each worker has its own copy of the Dataset object and its own index queue. The main process maintains a data queue where the tensors returned by the workers are stored. If the pin_memory feature, which accelerates data transfer between RAM and GPU memory, is enabled, then the main process also spawns a pin memory daemon thread. To retrieve a batch, the DataLoader selects a worker in a round-robin

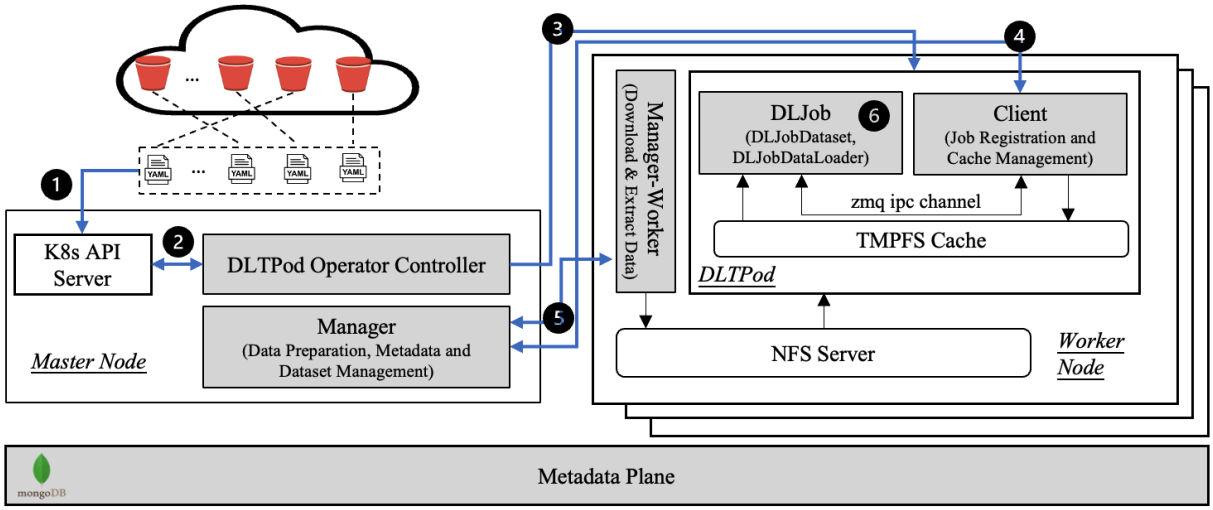


Fig. 2: DLCache Architecture (shaded blocks are our novel contributions to the K8s ecosystem)

manner and puts a batch index in the worker's index queue. The worker then retrieves the batch index, loads the data, converts it into a tensor, and places it in the data queue. If *pin_memory* is enabled, the pin memory daemon thread copies the tensor from the data queue to pinned memory. If not, the main process directly pops the tensor from the data queue and returns it, with the CPU then copying the tensor from RAM to GPU memory using traditional means.

Enabling *pin_memory* can improve GPU memory utilization by allowing more data from different workers to be pushed into GPU memory quickly. The GPU waiting time is influenced by the time it takes for the workers to load the data and place it in the data queue, as well as the number of mini-batches in the data queue when the main process tries to retrieve data. The DataLoader includes a $prefetch\_factor$ parameter that determines how many mini-batches will be preloaded into memory before the start of each epoch.

B. Kubernetes (K8s) Operator

In a K8s ecosystem, a custom resource like a DLT job is managed using a combination of Custom Resource Definition (CRD) and Operator Controller. A CRD is a custom resource added to the K8s API allowing users to store a collection of objects of a specific kind. The Operator Controller is a routine running within the K8s cluster that watches for events such as create, update, or delete on the custom resource and updates its status to the desired state. The Operator Controller is deployed as a separate pod in the cluster and is responsible for managing the custom resource. By wrapping a DLT job as a K8s Operator, users can focus on the application without having to worry about underlying support functions. For example, the Operator Controller can make placement decisions for the DLT job based on cluster resources and automatically add necessary built-in API objects to the CRD.

III. Design of DLCache

We now describe the design of DLCache in detail including its components and workflow. As shown in Figure 2, DLCache consists of four layers: control plane (in the master node), metadata plane, operator plane, and storage plane (which is distributed), all integrated within the K8s ecosystem, and where a DLT job is deployed as a new K8s CRD pod type we created called DLTPod. Users are required to provide a container image for the DL application and a YAML file with properties defined in the DLTPod specification. The data sources are assumed to be stored in external cloud storage, which can be accessed with security tokens and object keys provided by users. The system workflow is summarized in Algorithm 1.

Algorithm 1: System Workflow

1 Users send a DLTPod deployment request to the K8s API server (❶);
2 K8s API server invokes Operator Controller to process the request (❷);
3 **if** *Dataset size* $\leq$ *free + preemptible storage space* **then**
4 Deploy DLTPod (❸);
5 Client in DLTPod registers to Manager for dataset preparation (❹);
6 Manager-Workers execute preparation (❺);
7 Client notifies DLJob to start execution when dataset is cloned. During training, Client coordinates with the DLJob to transfer data from NFS to the TMPFS cache, release used data, and track the life stage of data chunks in NFS (❻);
8 **else**
9 Reject the deployment request;
10 **end**

The rest of this section delves in the details of the architecture explaining each component.

A. DLTPod: A Custom Pod Type for DL Jobs

In the K8s object model, Pod is the smallest unit representing a single instance of a running process in a cluster. DL jobs

require certain operations that are not natively supported by the traditional K8s Pod type. Consequently, we were required to define our own custom Pod type called DLTPod to support DL jobs, which also helps to hide complexities from the end user and enables them to use the new pod type seamlessly within the K8s ecosystem. DLTPod inherits the specifications of K8s native Pod; therefore, all fields for Pod are supported by DLTPod. We utilize K8s' Secret to store the user's credential information so that they do not need to be hard-coded or stored in the corresponding DLJob container, which reduces the risk of exposing confidential data. Containers in DLTPod are associated with a K8s ConfigMap, NFS Volumes, and a TMPFS (RAM-backed file system) Volume. The DLTPod comprises two containers: DLJob with the actual deep learning training logic, and a side car container called Client used in job registration and cache management.

In the DLTPod's YAML/JSON file, users must provide information about the data source, usually in the form of prefixes for the data object keys stored in the cloud storage bucket for training and validation. For supervised learning tasks where samples and targets are separate files (such as speech-to-text translation), users must supply a manifest file that outlines the mappings between the input samples and their corresponding targets. In other cases, it is optional to use the manifest field depending on whether it is needed for mapping $\langle sample, label \rangle$ tuples while loading data.

We designed the DLJobDataset and DLJobDataLoader primitives on top of PyTorch's Dataset and DataLoader utilities, respectively, for dataset abstraction and data loading. Users of DLCache must subclass from the DLJobDataset class and initialize a DLCJobDataLoader object in their DL application. DLJobDataset integrates the logic of mapping the user-known file path in the cloud bucket to hashed paths in our system, as well as handling data missing in cache and NFS. The DLJobDataLoader interacts with the Client through ZeroMQ IPC [9] to provide dynamic cache management.

Additionally, DLJobDataLoader does not need users to specify the number of data-loading workers but can automatically tune the value by comparing the training time and data-loading latency. Since we identify data blocks by their location and content hash value (explained later), the Client will generate a digest file that maps user-known data object keys in the cloud bucket to local file paths in the format: */nfs_server_ip/content_hash_value*. The DLJobDataset will automatically load this digest file as class attributes as users might need the information when retrieving data items.

B. DLTPod Operator Controller

We store the DLTPod metadata as structured data in the etcd key-value store of K8s when the CRD is created, but it becomes active only when associated with the DLTPod Operator Controller. This Controller ensures that the current state of the DLTPod always matches the desired state. The declarative API for DLTPod exposes fields related to the DLJob only, while other supporting components are created automatically by the Controller, which isolates users from the complexities of the underlying workflows and communication. Our Controller includes a job scheduler module, which is responsible for making placement decisions for DLTPods by analyzing the current dataset distribution and remaining resources across the cluster. The goal is to deploy an incoming DLTPod on a node that has (a) available computation resources (e.g. GPUs), (b) as many data blocks of the associated dataset as possible, and (c) as much free disk space as possible. This improves the probability of loading data from the local NFS server while meeting the computation resource requirements and reducing the network to disk I/O latency.

A score is computed for each node to represent how well these conditions are met. Assuming that node i contains k data blocks of the dataset belonging to job j, K of the D data blocks in the dataset already exist in the cluster, and the ratio of the available storage space of node i to the total available space is α, then we calculate the score of node i for job j as $score(i,j) = I(i,j) + \frac{k+\alpha(D-K)}{D}$, where $I(i,j)$ is a binary indicator signifying whether the available computation resources on node i can satisfy job j's requests. The job gets deployed on the node with the highest score. A sequence of nodes with decreasing scores is saved in a ConfigMap, which can later be used in the dataset placement strategy.

C. DLCache Manager

The DLCache's Manager is responsible for dataset preparation and its management. It is deployed as a K8s Deployment and exposed as a K8s Service with a ClusterIP. Once a DLTPod is deployed, the Client container is automatically started in it and connected to the Manager's gRPC server. The Client calls the job registration service to forward the cloud credentials secret, data source information, node scores, and resource profiles to the Manager. The Manager then saves these information into MongoDB as the job's metadata and notifies the Manager-Workers running on designated nodes to prepare the datasets. The Manager-Workers communicate with the Manager via gRPC and are responsible for downloading and extracting data files. They are deployed as a Pod on every node within the cluster (K8s Daemonset). Since each Manager-Worker is only responsible for downloading data on its local NFS server, there is no redundant data transmission within the cluster while preparing datasets.

1) Dataset Preparation: The Manager orchestrates Manager-Workers to perform data preparation according to node scores computed by the DLTPod Operator Controller. Before downloading a data file from the cloud bucket, the Manager will check if it is already available on the NFS cluster by checking its content hash value in MongoDB. If the data object's content hash is provided by the cloud bucket service, such as the Entity Tag (ETag) attribute for objects in S3, it will be used. Otherwise it will be generated using the hex digest of the SHA-256 hash of the first 1MB of the file's content. The reason for this constraint is to prevent the memory from overflowing and prolonging the data preparation time by directly reading large data files. The path of data files on the NFS cluster is in the $/nfs\_server\_ip/ETag$

format. Note that the Etag, which represents a single file or an uncompressed folder, is the smallest unit of manipulation.

NFS servers are selected sequentially following the descending order of node scores. A new NFS server is chosen if the current node's storage is exhausted. This locality-first strategy ensures that data resides on the same node as the DLJob as much as possible. If the dataset is (partially) available on the cluster and no active job is used, we redistribute the dataset using the node score sequence. If multiple DLJobs share the same dataset and cannot be deployed on the same machine due to resource constraints, we do not change the distribution of the dataset as it may cause disruption to the ongoing jobs.

Since data providers typically compress a dataset into several compressed files to reduce the data sharing cost and save uploading time, the Manager-Worker detects the compression format automatically, then downloads and extracts them using the pigz [10] parallel decompression technique. We preserve the file tree structure if the compressed file is initially a folder. If a data chunk is available on NFS but does not locate on the node decided by the Manager based on node scores and has no associated active job, the Manager will move it to the targeting node. We measure the total time of downloading and extracting a data object, called the cost of the data object, which will be an essential criterion when making the data eviction decision.

2) Cost-aware LRFU Data Eviction: After a job has completed, its dataset is not immediately deleted as it may still be needed by other jobs in the near future. Instead, a novel cost-aware data eviction strategy is used when the cluster runs out of storage. We first identify if a data block is eligible for deletion. A data block is considered eligible if it is no longer being used by any active jobs and has completed its cool-down period. The DLJobDataLoader uses a daemon process to initiate the cool-down of used data blocks after each epoch. This process terminates once a new epoch starts. Once all the epochs have finished and the cool-down period has expired, a data block becomes inactive and becomes eligible for deletion.

We designed the Cost-aware LRFU eviction strategy to address the limitations of traditional cache eviction policies such as Least Recently Used (LRU), Least Frequently Used (LFU), and Least Recently/Frequently Used (LRFU) [11] which only consider the historical references but ignore the cost of re-downloading and processing deleted data blocks thereby adversely impacting data preparation time. Traditional LRFU combines the LRU and LFU policies and considers the likelihood of a data block (represented by an ETag) being used in the future based on its combined recency and frequency (CRF) of past references.

At time t_{base}, the CRF value of data block b is calculated as $CRF_{t_{base}}(b) = \sum_{i=1}^{k} F(t_{base} - t_{b_i})$, where b_i indicates the i^{th} time reference of the block b before t_{base}, and $F(x)$ is the weighting function suggesting the contribution of a reference, which decreases with time passing. Let x be the difference between the current time and the time of a reference in the past, then $F(x) = \left(\frac{1}{p}\right)^{step \times x}$, where p is an attenuation factor that controls how fast the influence decays with time passing, and $step$ balances how much LRFU is close to LRU and LFU. LRFU becomes LFU if $step$ is 0, else LRU when $step$ is 1.

We improve upon the LRFU scheme as follows. In DLCache we assume all data files in a dataset are accessed at the same time when the job is registered because the random data-access nature in the DL training process requires all data to be available ahead of training. Therefore, for an incoming job, data chunks belonging to the same dataset have identical t_{base}. We further take the cost of each data block $C(b)$ into account when sorting the $CRF_{t_{base}}$ values. Simply put, the Cost-Aware LRFU policy evicts data blocks based on increasing order of their CRF values at t_{base} (first criteria) and data preparation cost (second criteria).

D. Cache Management

1) Best-effort Caching: From Section II, the PyTorch DataLoader improves I/O performance by overlapping the training time and the data-loading time, which attempts to have data reside in memory when the model needs them. Likewise, when using cache to improve I/O performance, we need to ensure the data have been pushed into the TMPFS cache when DLJobDataLoader workers need to load them. To that end, the DLJobDataLoader relays the succeeding index of the batch that will be immediately loaded by workers to the Client through ZMQ IPC. The Client inserts the received messages in a separate queue and utilizes a daemon process to load data into the cache asynchronously. A loader in the daemon process loads data in the inverse order of samples in the batch and is interrupted when a new batch index is pushed into the queue, when it starts processing the next batch. This ensures that as many samples as possible are forwarded to the cache and fully utilized. Specifically, we skip the first $prefetch\_factor \times num\_workers$ batches in each epoch if the dataset shuffle is enabled because loading them at once can prolong the launching time of each epoch. In contrast, consecutive epochs can be overlapped if the shuffle is disabled, and only the first $prefetch\_factor \times num\_workers$ batches in the first epoch are missed. If a data item in TMPFS is not available when reading, we implicitly redirect the target file path to that in NFS by wrapping the $\_\_getitem\_\_$ function in the PyTorch Dataset utility.

2) Release Cache: At the beginning of an iteration, the DLJobDataLoader sends the index of the last-used trained batch to the Client. Like the loading cache process, the Client maintains another daemon process to delete data from TMPFS asynchronously. Since the main process of DLJobDataLoader inserts a new index into a worker's index queue only after it consumes one batch from the data queue, theoretically there are at most $prefetch\_factor \times num\_workers$ batches of data residing in cache during the training time.

E. Adaptive Multiprocess Data Loading

The purpose of employing multiple workers for data loading in PyTorch's DataLoader utility is to parallelize the loading and preprocessing of data so that data can be directly accessed from RAM or GPU memory (if the pin_memory

is enabled). Increasing the number of workers can increase the memory footprint and CPU utilization of the loading process. In practice, users need to fine-tune the $num\_workers$ parameter manually in search of the optimal trade-off between data-loading speed and resource utilization. Our DLTJob-DataLoader incorporates an online approach to identify the optimal value of $num\_workers$ automatically based on real-time measurements as shown in Algorithm 2.

Assuming the request interval between consecutive training iterations is t_{req} and the fetching time (including the I/O time and data processing time) for a single batch is t_{fetch}, then to avoid data stalls, the number of workers ω should satisfy $\omega \geq \frac{t_{fetch}}{t_{req}}$. However, if ω is greater than the number of CPU cores, the CPU cores will be oversubscribed and the workers will compete for CPU time. This can lead to context switching, which can cause a decrease in performance. Therefore, we adopt the constraint with the theoretically calculated worker number (line 2). In addition, depending on the batch size, a high value of ω might overflow the RAM or GPU memory ($pin\_memory = True$). Therefore, the ultimate value of ω should also be subjected to CPU and memory constraints (lines 3-6). Finally, the updating decision is applied to the data loader by spawning new workers or pausing active workers, and synchronous the data pre-fetching index (lines 7-12). Note that paused worker processes have a grace period to be terminated because their index queue might have pending batches indexes. The algorithm tunes the number of workers every B batches starting from an initial value ω, and stops after N times.

Algorithm 2: Tune the $num\_workers$ Knob

Data: the initial value of $num\_workers$: ω, current average data fetching time: $\bar{t}_{fetch}$, current average interval between consecutive training iterations: $\bar{t}_{req}$, the number of CPU cores on the machine: num_cores, and the memory usage upper bound mem_{up};

Result: Add or remove active workers;

1 Shut down inactive workers;
2 $\omega' = min(\lceil \frac{\bar{t}_{fetch}}{\bar{t}_{req}} \rceil, num\_cores)$;
3 Average memory usage per worker $mem_{worker} = \frac{mem_{total}}{\omega}$;
4 **if** $mem_{worker} \times \omega' > mem_{up}$ **then**
5 $\quad \omega' = \lfloor \frac{mem_{up}}{mem_{worker}} \rfloor$
6 **end**
7 $\Delta = \omega' - \omega$;
8 **for** $i = 0;\ i < |\Delta|;\ i = i + 1$ **do**
9 $\quad$ Spawn a new worker process if $\Delta > 0$, pause a worker process (stop putting index into its index queue) if $\Delta < 0$;
10 **end**
11 Put $prefetch\_factor$ number of batch indexes into each new workers' index queue if $\Delta > 0$;
12 Adjust the dataloader prefetch index to ensure there are at most $\omega' \times prefetch\_factor$ outstanding batches in Cache.

IV. Experimental Evaluation

This section describes three experiments that evaluate DL-Cache along several dimensions using multiple datasets. Generally speaking, they answer the following questions:

- Compared to the baseline that uses the PyTorch framework for data loading, can DLCache improve the data loading throughput under heterogeneous hardware and workload configurations?
- Can DLCache automatically find the optimal number of data loader workers for a given workload and ensure throughput improvement at the same time?
- Can DLCache's cost-aware data eviction strategy reduce the data preparation time when executing a series of DLT jobs in a resource-constrained environment?

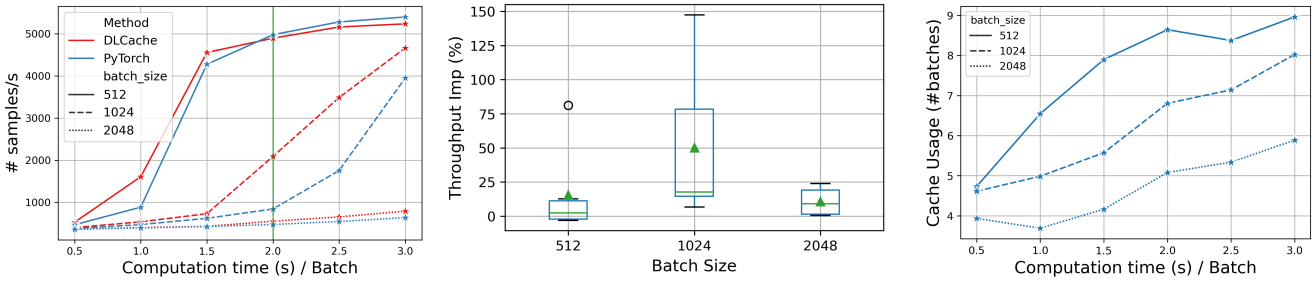
A. Experiment Setup

Hardware and Software: Our testbed comprises four AWS EC2 instances; each is equipped with 8 2.30GHz vCPUs, 32GB RAM, and a 512GB SSD. They are connected via a 1.0 Gbps LAN. NFSv4 servers are installed on the nodes without space limitation. We set up a K8s (v1.17.6) cluster across the nodes and used Flannel for the container network plugin. Since DLCache is geared to optimizing dataset placement and data loading and not the actual training process, we simulate the computation time using a sleep function. This setting allows us to verify whether DLCache is adaptive to different hardware, model configurations, and workloads. We employ PyTorch v1.12 for implementations of the test applications and the DLJobDataset and DLJobDataLoader bundle.

Workloads and Datasets: The simulated workloads for the first two experiments are an image classification model and a speech recognition model, which are trained on the ImageNet-ILSVRC [12] (136GB size) and the LibriSpeech [13](105GB size) datasets, respectively. Recent benchmarking [14] on a variety of modern GPU devices reveal the training throughput of the ResNet50 [15] model ranges from 500 to 4,000 images/s. Likewise, [16] reported spending 38 hours ($batch\_size = 256$) to train a speech recognition model on LibriSpeech for 100 epochs using 8 Nvidia V100 16GB GPUs. Based on these practical observations, we simulated a range of training times (from 0.5s to 3.0s) under different batch sizes (128 to 2048).

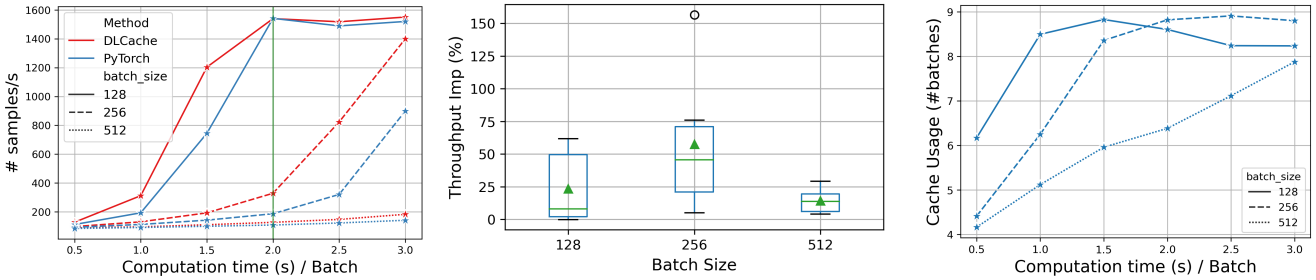
Due to storage capacity limitations, a dataset may be (partially) removed if it is not bound to any active job and new jobs are requesting resources. Consequently, if this dataset is used again in the future, the DLT job has to wait for the dataset to be re-downloaded. In the real world, different DLT jobs might have different preferences or frequencies for reusing a dataset.

We summarize four types of datasets: (1) *One-time-use datasets* are used by DLT models that are trained only once and then deployed, (2) *Reuse-oriented datasets* usually appear in use cases, such as fine-tuning pre-trained models, hyper-parameter tuning, or boosting algorithms, (3) *Partially-reused datasets* are common in scenarios where models are trained on a mixture dataset composed of part of the old dataset and some



(a) (b) (c)

Fig. 3: Compare DLCache and PyTorch performance on improving data loading throughput with $num\_workers = 4$ (no auto-tuner) using the **ImageNet-ILSVRC** dataset: (a) data loading throughput of DLCache and PyTorch; the green line indicates where the data stall eliminates when the batch size is 512; (b) average throughput improvement compared to PyTorch under different batch size settings; (c) cache usage of DLCache.



(a) (b) (c)

Fig. 4: Compare DLCache and PyTorch performance on improving data loading throughput with $num\_workers = 4$ (w/o auto-tuner) using the **LibriSpeech** dataset: (a) data loading throughput of DLCache and PyTorch; the green line indicates where the data stall eliminates when the batch size is 128; (b) average throughput improvement compared to PyTorch under different batch size settings; (c) cache usage of DLCache.

emerging new data, such as model drift correction, updating a classifier for new classes, rebalancing class distribution, and continuous learning models, and (4) *Temporary-shared datasets* are similar to one-time-use datasets but shared by multiple training jobs simultaneously. They are usually used in multi-task, multi-view, or ensemble learning scenarios.

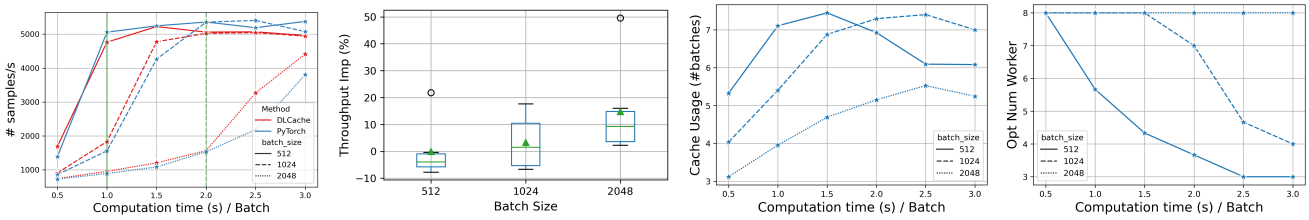
In the third experiment, we assume these four types of datasets map, respectively, to a private traffic mobility dataset TrafficInrix (96GB size), ImageNet-ILSVRC, a network traffic dataset CSE-CIC-IDS2018 [17] (6.89GB size), and LibriSpeech. We set the reuse times for ImageNet-ILSVRC to 1, 3, and 5, and the number of Jobs sharing LibriSpeech and the number of reused old data files in CSE-CIC-IDS2018 to 1, 3, 5, and 7. Moreover, we set 10 storage stress levels ranging from 160GB to 260GB. Therefore, there is a total of 480 combinations of system-level and job-level configurations. For each configuration, we randomly sample the sequence of the jobs 50 times, yielding 24,000 workload patterns.

B. Evaluation Results

In experiments 1 and 2, each test loaded 100 mini-batch data and was repeated three times on three worker nodes in our cluster. Since the size of both datasets is smaller than the disk capacity, DLTPod can load the data directly from the mount point of the local NFS server thereby avoiding the use of the network. We report the average data loading throughput (excluding the simulated computation time), the throughput improvement ratio, and the average number of samples resident in cache. The cache usage is determined by dividing the number of samples present in the cache prior to each cache release operation by the corresponding batch size; therefore, the maximum value would be $prefetch\_factor \times num\_workers + 1$, indicating the highest cache utilization during the test. For experiment 2, we additionally present how the computed $num\_workers$ changes over the simulated compute time under different batch size settings.

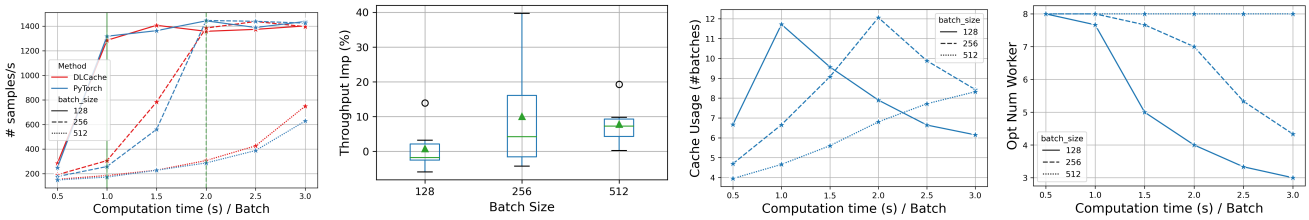
Recall that enabling pin_memory can speed up GPU data access and potentially increase training speed, but also increases GPU memory utilization when there are multiple data loading workers. In Experiment 1, we assume that the user has enabled the pin_memory feature, and due to limitations of GPU memory, the maximum num_worker is set to 4. Moreover, we set the $prefetch\_factor$ to 2, and disabled the automatic adaptation of num_worker feature in DLCache.

As shown in Figures 3a and 4a, the throughput decreases with the increase in batch size. The primary reasons are: (1) the disk I/O speed is not fast enough to keep up with the increasing demand for data; (2) loading and preprocessing the larger batch sizes increases the CPU overhead causing the data loading throughput to decrease. In addition, data stalls gradually diminish with an increase in simulated compute time as expected because the data loader has sufficient time to



(a) (b) (c) (d)

Fig. 5: Compare DLCache and PyTorch performance on improving data loading throughput with $num\_workers = 8$ (w/ auto-tuner) using the **ImageNet-ILSVRC** dataset: (a) data loading throughput of DLCache and PyTorch; the solid and dashed green lines indicate where the data stall eliminates when the batch size is 512 and 1024; (b) average throughput improvement compared to PyTorch under different batch size settings; (c) cache usage of DLCache; (d) auto-tuned $num\_workers$ for given workloads.



(a) (b) (c) (d)

Fig. 6: Compare DLCache and PyTorch performance on improving data loading throughput with $num\_workers = 8$ (w/ auto-tuner) using the **LibriSpeech** dataset: (a) data loading throughput of DLCache and PyTorch; the solid and dashed green lines indicate where the data stall eliminates when the batch size is 128 and 256; (b) average throughput improvement compared to PyTorch under different batch size settings; (c) cache usage of DLCache; (d) auto-tuned $num\_workers$ for given workloads.

prefetch data from disk into memory. Therefore, the upper bound of data loading throughput ultimately depends on the data processing time on CPUs. DLCache delivers similar throughput to PyTorch. On the other hand, if the computation speed is very fast (e.g., 0.5s), the time for DLCache to transfer data from disk to cache will be shorter; therefore, relatively fewer I/O operations can benefit from the cache. In contrast, DLCache significantly outperforms PyTorch in the use cases where I/O is the primary bottleneck of data loading.

As depicted in Figures 3b and 4b, when evaluated under the established batch size configurations, DLCache demonstrates an average increase in throughput for ImageNet-ILSVRC in the range of 10.68% to 50.27%, and for LibriSpeech from 14.26% to 57.48%. Figures 3c and 4c demonstrate that more batches can be transferred into cache with increase in computing time, but the usage is strictly limited under 9 batches. Also, workloads with large batch sizes transfer fewer batches into cache because the transfer process of a batch will get interrupted when a new data loading request arrives. These observations validate that DLCache uses cache efficiently.

Experiment 2 utilized 8 workers for PyTorch's $num\_workers$ parameter, aligning with the number of available CPU cores. This implies that both RAM and GPU memory is sufficient to support the training process regardless of whether the page-locked memory (pin_memory) is used. The number of data-loading workers was optimized by DLCache, starting at 8 and concluding after 20 batch cycles. As indicated in Figures 5a and 6a, the general trend of throughput is comparable to that of Experiment 1. The time required for data loading and processing for a given batch is reduced, and the throughput begins to converge at an earlier stage. When the computation time fully overlaps the data loading time (denoted by the green solid and dotted lines), DLCache's throughput is slightly lower than PyTorch. In this scenario, DLCache's online caching operations has no opportunity to improve performance and instead incurs additional overhead (e.g. CPU contention). However, DLCache constantly delivers higher throughput when the I/O bottleneck is non-negligible.

In summary, as shown in Figures 5b and 6b, DLCache exhibits an average increase in throughput for ImageNet-ILSVRC in the range of -0.02% to 14.81% and for LibriSpeech from 0.85% to 10.07% when evaluated under the established batch size and computation time configurations. Additionally, Figures 5d and 6d demonstrate that DLCache can adaptively adjust the number of workers to the optimal range, thereby providing the best performance and efficient resource utilization under the given configurations. The real-time cache loading mechanism of DLCache is capable of adapting to changes in the number of workers, thereby preventing the loading of unnecessary samples into the cache, as shown in Figures 5c and 6c. It is worth noting that a reduction in the number of workers does not suddenly result in a decrease in cache usage. This is because the utilization of cache is also contingent upon the number of samples that can be transferred into the cache per batch within a specified computation time constraint.

In the third experiment, which aims to assess the efficacy of DLCache's cost-aware LRFU strategy on data preparation time, a range of classical algorithms were considered, including Least Recently Used (LRU), Least Frequently Used (LFU), Least Recently Frequently Used (LRFU), and Random, as baselines for performance comparison. The data eviction

algorithms used in cache systems have a primary objective of determining which data should be evicted from the cache to accommodate new data while disregarding the cost of data preparation. In contrast, the Greedy algorithm adopts a unique approach by removing the data chunk with the lowest cost. DLCache's Cost-Aware LRFU algorithm takes into consideration not only the data access frequency and recency but also the data preparation cost thereby providing a comprehensive evaluation of our cost-aware LRFU strategy.

To investigate a wide range of workload patterns (24,000 in our case), this experiment was performed in an offline manner as a simulation. The diversity in the costs associated with different data chunks presents an opportunity to optimize data preparation time by making strategic eviction decisions. As depicted in Figure 7, the cumulative distribution function (CDF) curve of the Cost-Aware LRFU approach consistently surpasses those of the baseline algorithms without substantial variations demonstrating the superiority of Cost-Aware LRFU in performance.

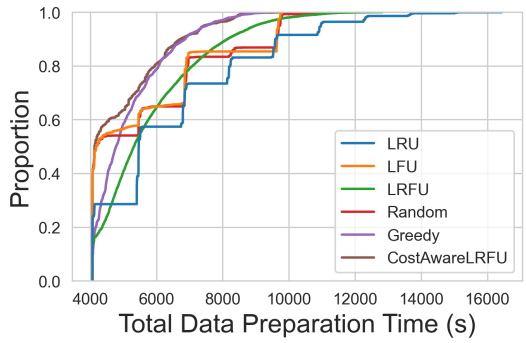


Fig. 7: CDF curve of the total data preparation time when executing a series of DLT workloads.

V. Related Work

Many prior efforts exist in the literature that address the storage management and data access challenges from different perspectives in DLT systems. We provide a sampling of these and compare DLCache with them.

Databases and file systems for DL applications: Research efforts exist that determine the best storage solution for DL applications including evaluations of existing database software and custom platforms. Some studies have compared object storage systems and key-value storage databases. For example, Cheng et al. [18] compared MinIO, Ceph, MongoDB, Redis, and Cassandra and found that the optimal storage choice depends on the specific workload and configurations. File-system-based solutions, such as FanStore [19] and Alluxio [20], have also been proposed. FanStore is a runtime shared file system for efficient and scalable DL training, while Alluxio serves as a middleware between a distributed file system and a computation framework. In contrast, our approach uses a lightweight and mature NFSv4 with cachfilesd and TMPFS for data sharing and caching, and a global MongoDB database to manage file metadata. Jobs obtain metadata from the database, generate a runtime manifest based on their data access sequence, and save it in RAM.

Caching mechanisms for DL training: Various caching strategies have been proposed to improve I/O performance in DL training. DELI [2] uses a disk-based MongoDB database for caching and a customized pre-fetching approach to maximize overlap between communication and computation. Quiver [3] uses cloud storage for input data and local SSD for caching, offering three options based on how much the job can benefit from caching. In our case, the dataset is stored as compressed files in a cloud bucket, so it's not suitable to use as the backend storage. DLCache is a three-level data orchestration system, with added support for batch-grained in-memory caching compared to Quiver and DELI, which enhances our ability to optimize I/O performance.

Prior work on data loading and caching for distributed training has focused on locality-aware techniques and aggregation of local caches. Yang et.al [21] proposed a shared in-memory cache pool built from the local caches of all nodes. The dataset is divided into disjoint subsets and stored in each node's cache, and learners on each node receive the same global mini-batch sequence and make agreements on how to load samples locally. Wang et.al [22] presented DIESEL, a task-grained distributed caching system that aggregates small data files into large data chunks and saves the mappings as metadata snapshots in memory. The caching process in DIESEL is integrated with a chunk-wise shuffle method that transforms random file read requests into chunk-based reads and caches visited chunks. In comparison to prior works, DLCache adopts an on-demand and best-effort caching strategy. This is due to the fact that, in real-world scenarios, the rapid growth of dataset size makes it costly to establish a distributed memory pool. DLCache can achieve high cache hit rates as long as the computation time is greater than the cost of transferring data from SSD to TMPFS. The cache will contain at most $prefetch\_factor \times batch\_size$ items throughout the training process, which is usually a tiny fraction of the entire dataset. Additionally, these previous studies do not account for the actual I/O requirements of a DLT job under specific runtime configurations (e.g., $batch\_size$ and $num\_workers$) and resource limitations (e.g., GPU computation speed and GPU memory size). As our experiments demonstrate, if the computation time coincides with the data loading time, the cache system would not be able to enhance performance. In such cases, maintaining a subset of the dataset in memory would be unnecessary.

VI. Conclusion

This paper describes DLCache, which provides comprehensive system optimizations for deep learning dataset placement and data loading in cloud-native deep learning training jobs. DLCache leverages a three-tier storage and implements multiple data services, including dataset preparation, placement, and cost-aware least recently/frequently used (CLRFU) data eviction strategy. A Kubernetes CRD controller and operator including a new Pod type were developed to simplify the DevOps process for DLT jobs. The system implements a runtime-aware best-effort caching mechanism that reduces the

waiting time caused by data stalls in the PyTorch framework. Additionally, DLCache enhances the PyTorch DataLoader to automatically decide the number of DataLoader workers based on real-time training performance and data retrieval latency. The performance of DLCache was evaluated through a systematic set of experiments under a wide range of runtime settings. The results demonstrate that DLCache can significantly improve DLT data loading throughput even when there is an I/O bottleneck and achieve comparable performance with the PyTorch framework when there is no data stall. The cost-aware LRFU data eviction policy was found to outperform a variety of commonly used policies.

In the future, we plan to (1) optimize the system and reduce operation overheads; (2) evaluate other types of network-based and parallel file systems (e.g., GPFS [23], Lustre [24], etc.); (3) port DLCache to GPU clusters; and (4) add more features to support distributed learning and federated learning in cloud.

REFERENCES

[1] N. Dryden, R. Böhringer, T. Ben-Nun, and T. Hoefler, "Clairvoyant prefetching for distributed machine learning i/o," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2021, pp. 1–15.

[2] N. Krichevsky, R. St Louis, and T. Guo, "Quantifying and improving performance of distributed deep learning with cloud storage," in *2021 IEEE International Conference on Cloud Engineering (IC2E)*. IEEE, 2021, pp. 99–109.

[3] A. V. Kumar and M. Sivathanu, "Quiver: An informed storage cache for deep learning," in *18th USENIX Conference on File and Storage Technologies (FAST 20)*, 2020, pp. 283–296.

[4] R. Gu, K. Zhang, Z. Xu, Y. Che, B. Fan, H. Hou, H. Dai, L. Yi, Y. Ding, G. Chen *et al.*, "Fluid: dataset abstraction and elastic acceleration for cloud-native deep learning training jobs," in *2022 IEEE 38th International Conference on Data Engineering (ICDE)*. IEEE, 2022, pp. 2182–2195.

[5] D. Graur, D. Aymon, D. Kluser, T. Albrici, C. A. Thekkath, and A. Klimovic, "Cachew: Machine learning input data processing as a service," in *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, 2022, pp. 689–706.

[6] H. Robbins and S. Monro, "A stochastic approximation method," *The annals of mathematical statistics*, pp. 400–407, 1951.

[7] G. Nguyen, S. Dlugolinsky, M. Bobák, V. Tran, Á. López García, I. Heredia, P. Malík, and L. Hluchỳ, "Machine learning and deep learning frameworks and libraries for large-scale data mining: a survey," *Artificial Intelligence Review*, vol. 52, no. 1, pp. 77–124, 2019.

[8] "torch.utils.data — pytorch 1.13 documentation," https://pytorch.org/docs/stable/data.html.

[9] "zmq_ipc(7) - 0mq api," http://api.zeromq.org/master:zmq-ipc.

[10] M. Adler, "pigz: A parallel implementation of gzip for modern multi-processor, multi-core machines," *Jet Propulsion Laboratory*, 2015.

[11] D. Lee, J. Choi, J.-H. Kim, S. H. Noh, S. L. Min, Y. Cho, and C. S. Kim, "Lrfu: A spectrum of policies that subsumes the least recently used and least frequently used policies," *IEEE transactions on Computers*, vol. 50, no. 12, pp. 1352–1361, 2001.

[12] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, "Imagenet: A large-scale hierarchical image database," in *2009 IEEE conference on computer vision and pattern recognition*. Ieee, 2009, pp. 248–255.

[13] V. Panayotov, G. Chen, D. Povey, and S. Khudanpur, "Librispeech: an asr corpus based on public domain audio books," in *2015 IEEE international conference on acoustics, speech and signal processing (ICASSP)*. IEEE, 2015, pp. 5206–5210.

[14] "Gpu benchmarks for deep learning — lambda," https://lambdalabs.com/gpu-benchmarks.

[15] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 770–778.

[16] K. Zhao, H. D. Nguyen, A. Jain, N. Susanj, A. Mouchtaris, L. Gupta, and M. Zhao, "Knowledge distillation via module replacing for automatic speech recognition with recurrent neural network transducer," 2022.

[17] I. Sharafaldin, A. H. Lashkari, and A. A. Ghorbani, "Toward generating a new intrusion detection dataset and intrusion traffic characterization." *ICISSp*, vol. 1, pp. 108–116, 2018.

[18] P. Cheng and H. S. Gunawi, "Storage benchmarking with deep learning workloads."

[19] Z. Zhang, L. Huang, U. Manor, L. Fang, G. Merlo, C. Michoski, J. Cazes, and N. Gaffney, "Fanstore: Enabling efficient and scalable i/o for distributed deep learning," *arXiv preprint arXiv:1809.10799*, 2018.

[20] H. Li, *Alluxio: A virtual distributed file system*. University of California, Berkeley, 2018.

[21] C.-C. Yang and G. Cong, "Accelerating data loading in deep neural network training," in *2019 IEEE 26th International Conference on High Performance Computing, Data, and Analytics (HiPC)*. IEEE, 2019, pp. 235–245.

[22] L. Wang, S. Ye, B. Yang, Y. Lu, H. Zhang, S. Yan, and Q. Luo, "Diesel: A dataset-based distributed storage and caching system for large-scale deep learning training," in *49th International Conference on Parallel Processing-ICPP*, 2020, pp. 1–11.

[23] F. B. Schmuck and R. L. Haskin, "Gpfs: A shared-disk file system for large computing clusters." in *FAST*, vol. 2, no. 19, 2002.

[24] P. Schwan *et al.*, "Lustre: Building a file system for 1000-node clusters," in *Proceedings of the 2003 Linux symposium*, vol. 2003, 2003, pp. 380–386.