LCache: Log-structured SSD Caching for Training Deep Learning Models

Shucheng Wang⋆, Zhiguo Xu⋆, Zhandong Guo⋆, Jian Sheng‡, Kaiye Zhou⋆ and Qiang Cao†✉
⋆China Mobile (Suzhou) Software Technology Co., Ltd., Suzhou, China ‡Suzhou City University, Suzhou, China
†Huazhong University of Science and Technology, Wuhan, China

***Abstract*—Training deep learning models is computationally demanding and data-intensive. Existing approaches utilize local SSDs within training servers to cache datasets, thereby accelerating data loading during model training. However, we experimentally observe that data loading remains a performance bottleneck when randomly retrieving small-sized sample files on SSDs. In this paper, we introduce LCache, a log-structured dataset caching mechanism designed to fully leverage the I/O capabilities of SSDs and reduce I/O-induced training stalls. LCache determines the randomized dataset access order by extracting the pseudo-random seed from the training frameworks. It then aggregates small-sized sample files into larger chunks and stores them in a log file on SSDs, thus enabling sequential I/O requests on data retrieval and improving data loading throughput. Furthermore, LCache proposes a real-time log reordering mechanism that strategically schedules cached data to organize logs across different epochs, which enhances cache utilization and minimizes data retrieval from low-performance remote storage systems. Additionally, LCache incorporates an MetaIndex to enable rapid log traversal and querying. We evaluate LCache with various real-world DL models and datasets. LCache outperforms the native PyTorch Dataloader and NoPFS by up to 9.4× and 7.8× in throughput, respectively.**

***Index Terms*—Data Caching; Deep Learning; Solid State Disk**

I. Introduction

Deep learning (DL) has been extensively adopted in many applications, including language processing [1], [2] and image classification [3], [4]. The DL training tasks are inherently characterized by high computation and data demands; thereby, the efficiency of data storage and retrieval significantly impacts the end-to-end training performance [1], [7]–[9].

The DL training tasks are often trained on datasets that consist of a multitude of small files. For example, the widely used ImageNet-21K [3] for computer vision contains over 14.1 million images with less than 100KB average file size. Typically, datasets are stored on remote storage systems that have substantial capacity. During runtime, DL training frameworks retrieve data from remote systems to enable training, but this generally fails to meet the demands of computational throughput and leads to training stalls [8]–[11]. To overcome this problem, modern training frameworks such as DeepSpeed [12] and Megatron [13] overlap data loading with computation, which is still insufficient to eliminate I/O-induced training stalls caused by low-throughput remote data retrieval [8]–[10].

Previous studies have leveraged local memory and/or storage devices to cache datasets for enhancing data loading efficiency. The in-memory caching system [11], [14] struggles with the accommodation of TB-scale datasets due to limited memory capacity, necessitating frequent access to remote storage in runtime. In contrast, SSD-based caching systems exploit local SSD devices within training servers to expand cache capacity. However, our experiments (detailed in Section II-B) reveal that even with the entire dataset stored in local SSDs, the I/O-induced overhead with the mainstream PyTorch Dataloader [15] accounts for up to 45% of the training time. The main challenge in optimizing data loading is the randomized file assess pattern, which is generated by the widely used stochastic gradient descent (SGD) algorithm in model training [14], [16]. Our experimental analysis shows that the throughput of SSDs only reaches 25% of their maximum performance when randomly reading small-sized files.

To speed up data loading in DL training, we propose a log-structured SSD cache system, LCache for short, to fully exploit the I/O throughput and storage capability of local SSDs. LCache employs a log data-layout that deploys multiple Logfiles on SSDs to store cached sample files. It utilizes the predefined pseudo-random seeds employed by SGD to calculate the randomized data access list offline, then aggregates sample files into large-sized chunks and sequentially stores them in Logfiles according to the access list. In runtime, LCache facilitates reading chunks with sequential and large-sized I/O requests, thus preventing random file retrieval. Furthermore, LCache introduces a log reordering method that strategically schedules the accessed data to construct the Logfile for the next epoch, improving cache utilization and minimizing data prefetching from remote storage. To efficiently manage the metadata of sample files and chunks, LCache designs a two-level memory index to perform log traversal and file querying.

This paper makes the following contributions:

- We experimentally analyze the data loading overhead in DL training due to the random file access pattern on SSDs.
- We propose a log-structured caching mechanism LCache upon local SSDs. LCache aggregates sample files into large-sized chunks and stores them in the log manner, while strategically scheduling accessed data to construct logs for subsequent epochs.
- We evaluate LCache with a variety of datasets and deep learning models. LCache achieves up to 9.4× and 7.8× speedup over native PyTorch DataLoader and NoPFS in throughput, respectively.

The rest of the paper is organized as follows. Section II describes the background and performance analysis of data

✉ Qiang Cao is the corresponding author

978-3-9826741-0-0/DATE25/© 2025 EDAA

TABLE I
EVALUATION PLATFORM SPECIFICATIONS

Components	Configurations
Processor	Dual Socket Intel Xeon Platinum 8358, 64 Cores
Memory	256GB 2666MHz DDR4
GPU	8 * NVIDIA A800 (80GB)
Local Storage	8 * Intel P4610 NVMe SSD
Remote Storage	1 * Mellano ConnectX-4 (25 Gbps)
Operating System	Ubuntu 22.04 LTS with Linux kernel 5.15

loading in DL training tasks. Section III details LCache's design. We evaluate LCache in Section IV and describe related works in Section V. Section VI concludes this paper.

II. BACKGROUND AND MOTIVATION

A. I/O Patterns in Model Training

Training a DL model typically involves many epochs, each of which performs a traversal of the entire dataset. The stochastic gradient descent (SGD) method [17] is commonly employed to refine the model parameters during each epoch, as it requires less memory and achieves higher convergence efficiency [9], [14], [16]. Based on SGD, the DL training framework constitutes a *batch* by randomly selecting a subset of samples from the dataset, followed by iterative model training across multiple batches within an epoch. Additionally, the dataset is randomized to a different order between epochs to mitigate model overfitting. Previous studies [16], [18] have indicated that the full dataset randomization achieves better model training accuracy and stability.

In implementation, the DL training framework such as PyTorch [15] and TensorFlow [19] provides data loading utilities (*torch.utils.data.DataLoader* and *tf.data* respectively) for model training. They typically generate an index for each dataset, then shuffle and partition the index into multiple batches in each epoch. For small datasets such as CIFAR-10 [20] (i.e., including 60,000 images), it is feasible to cache the entire dataset into memory to mitigate I/O overhead. However, modern large-scale datasets, such as LAION [21] and ImageNet [3], contains more than tens of millions of files with storage capacity spanning an order of terabytes, which significantly exceed the available memory capacity. Therefore, random I/O requests to read data from storage are inevitable on training. The latest version of PyTorch DataLoader uses two methods to reduce I/O overhead: (1) utilizing multiple workers (i.e., subprocesses) to concurrently read data; (2) overlapping data loading and model computation through asynchronous prefetching. However, whether these methods can effectively improve data loading throughput on disks has not been fully studied.

B. Analysis of Data Loading

Experimental Setup. We start by measuring the data loading performance of DL training tasks running on a training server, whose configuration is detailed in Table I. The server contains 8 NVIDIA A800 GPUs and 8 Intel P4610 NVMe SSDs, each of which achieves up to 3GB/s and 3.2GB/s write and read throughput, respectively. We compose all SSDs into a RAID-0 array to achieve the best I/O performance in our platform.

TABLE II
MODEL SIZE AND COMPLEXITY

Model	Params (million)	FLOPs
AlexNet	61.10	7.1×10^8
SqueeseNet	1.25	8.2×10^8
ResNet-50	25.56	3.9×10^9
ViT	86.56	1.7×10^{10}

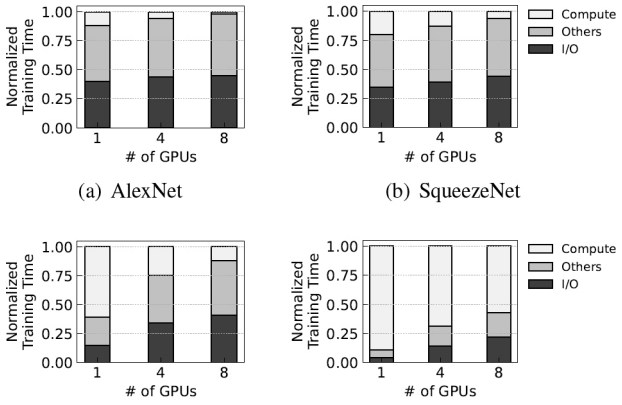
To ensure a comprehensive evaluation in terms of workload heterogeneity, this experiment includes two I/O-bound models AlexNet [22] and SqueezeNet [23], and two compute-bound models ResNet-50 [24] and Vision Transformer (ViT) [25], as detailed in Table II. We use the ImageNet-1K [4] dataset that consists of 1,000 classes and 1.28 million images (about 138 GB). We place the entire dataset into the SSD RAID and use the default dataset layout where one directory per class containing all image files of that class.

Performance of Data Loading. In experiments, we break down the overall training time into three parts: data loading (`I/O`), model computation (`Compute`) and additional time costs (`Others`) caused by index shuffling and decoding, etc. We use the PyTorch Dataloader and enable 16 workers for concurrently reading data from SSDs, while prefetching data for the next 2 batches (by setting the *prefetch_factor*) to overlap data loading with computation.

Fig.1 shows that reading data from SSDs is one of the major performance bottlenecks. The I/O overhead accounts for 22%-45% of the total training time across all four DL models. When training ViT with a single GPU, the prefetching mechanism in PyTorch efficiently pipelines data loading with computation, thus covering the I/O overhead. However, as the number of GPUs increases, the computation time decreases rapidly whereas the data loading time remains relatively constant or drops slowly. The training process has to stall when the data loading throughput falls below the computational speed. Therefore, the challenge of data loading performance becomes more severe with the use of more computational resources or advanced GPU devices.

C. Solid State Drive

Solid State Drives (SSDs) with increasing performance and capacity have been gradually replacing traditional Hard Disk Drives (HDDs) as the primary storage devices in many scenarios. To increase the density of SSDs, the number of bit per flash-cell has expanded from 1-bit SLC (Single-Level Cell) to 5-bits PLC (Penta-Level Cell) [26]. Modern DL training servers [27], [28] deploy high-performance NVMe SSDs as local storage for data caching and checkpoint saving. We further profile the file read performance on SSDs. We deploy the ext4 file system on the SSD RAID and invoke up to 8 user threads to randomly read 128KB-sized and 4MB-sized files. Results in Fig.2 show that the average read throughput for 4MB files is 5.2×-7.4× higher than that for 128KB files. There are mainly two reasons: (1) SSDs struggle to achieve maximum throughput with small-sized random reads [29]; (2) the overhead of file system operations, such as metadata retrieval and file opening/closing, becomes more substantial with small files [30]. For image-typed



(a) AlexNet (b) SqueezeNet

(c) ResNet-50 (d) ViT

Fig. 1. Time breakdown on training different DL models

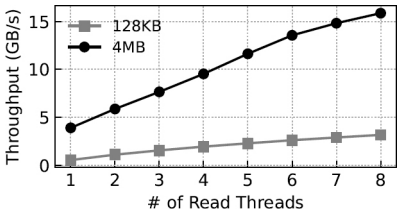


Fig. 2. SSD throughput on different access patterns

datasets such as ImageNet [3], [4], their average file sizes are generally smaller than 128KB. Therefore, it remains a challenge for DL training tasks to achieve desired data loading throughput even when the entire datasets are cached in local SSDs.

D. Clairvoyance Method

In most DL training tasks, the random shuffling feature of SGD complicates the optimization of data access performance. To address this problem, previous caching approaches such as NoPFS [16] and DeepIO [14] proposed the clairvoyance method to improve data loading performance. The key idea of clairvoyance is that the result of random shuffling is determined by a pre-defined random seed (e.g., using the *torch.manual_seed()* in PyTorch). Besides, parameters including the number of epochs, training servers and GPU devices are also set by users before initiating training tasks. Therefore, it is feasible to accurately predict and replicate the order of dataset access before shuffling. Leveraging this approach, NoPFS calculates the most frequently accessed samples and caches them in memory. DeepIO prefetches upcoming samples in the memory buffer to overlap data reading with model computation. However, these approaches cannot overcome the I/O bottleneck arising from random file accesses on SSDs. In this work, we explore how to utilize the clairvoyance method to optimize data layout in SSD caches and improve data loading throughput for training DL models.

III. Design

We introduce LCache that leverages local SSDs as a dataset cache to enhance data loading throughput in training tasks. LCache designs a log-based cache layout with a chunk aggregation mechanism to prevent random file reading on SSDs.

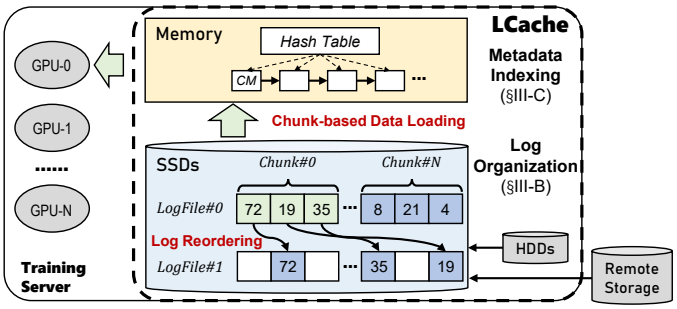


Fig. 3. Architecture of LCache

LCache further employs a log reordering method to efficiently maintain the log layout across different epochs by strategically scheduling accessed data.

A. Architecture

Fig.3 illustrates the LCache architecture, which utilizes local SSDs as a high-performance layer to cache datasets from HDDs or remote storage on training. It combines multiple SSDs into a RAID-0 array to maximize I/O throughput by exploiting the parallel processing capabilities among multiple SSDs. LCache introduces a log-structured data layout that partitions the global storage space into two *Logfiles* of equal capacity. Before training, LCache obtains the pseudo-random seeds and the number of GPUs/servers from the training framework, subsequently generating a *shuffled list* that indicates the file access order for the entire DL training task.

LCache aggregates a fixed number of sample files (e.g., defaulting to the batch size) into large-sized *chunks* and sequentially writes these chunks into a Logfile according to the access order in shuffled list (§III-B). During data loading, LCache reads samples into memory with a chunk-sized I/O request, thus fully leveraging SSD's sequential throughput while minimizing overhead on file system operations. LCache further proposes a log reordering mechanism to preserve the sequential access pattern across different epochs. In contrast to existing methods [8], [16] that focus on prefetching data from low-performance layers such as HDDs or remote storage system, LCache strategically schedules accessed data to the Logfile intended for the next epoch, thereby enhancing the efficiency of data prefetching. Furthermore, LCache proposes an in-memory MetaIndex (§III-C) to record the metadata and storage location for cached sample files. MetaIndex utilizes a two-level indexing structure to optimize query complexity and ensure high traversal performance.

B. Log Organization

To enhance data loading performance in DL training tasks, LCache transforms randomized small-sized files into sequential large-sized chunks on SSDs. Before training, LCache determines the number of sample files in a chunk based on the batch size setting, and retrieves the size of all files from remote storage system. Based on these information, LCache calculates the storage location of each sample file within the LogFile.

In the first epoch, LCache fetches sample files from remote storage for training, while writing them to the LogFile ac-

cording to the pre-calculated locations. However, each training epoch performs a different random data access order, preventing DL training frameworks from sequentially traversing the LogFile generated in the previous epoch. CoorDL [11] does not change the cache layout, thus leading to random file reads during training. NoPFS [16] evicts accessed data from local cache and prefetches other data from remote storage, which induces training stalls when the prefetching throughput is lower than the computation speed.

To address this issue, LCache proposes a real-time log reordering method as described in Algorithm 1, which strategically reuses cached data to maintain the log layout among epochs. During an epoch, DL training tasks traverse the dataset at the $batch$ granularity (line 1). LCache utilizes the MetaIndex (detailed in the next Section III-C) to query the storage location of chunks in current epoch's LogFile (line 5). Then it invokes a read request to fetch cached chunks from the Logfile (line 6-8), otherwise retrieving required files from remote storage (line 9-10). Then, LCache performs real-time log reordering after training (line 11). It queries the MetaIndex to obtain the storage location of used files in the next epoch's LogFile and writes them in the background (line 12-15). Finally, LCache removes the chunk in the current LogFile (line 16). Moreover, LCache employs the thread pool to reorder logs or release chunks in the background (line 15, 16), thus overlapping log reordering with model computation to prevent training stalls.

Modern DL training platforms such as NVIDIA DGX [28] and HGX [27] provide tens of terabytes of local SSD capacity; therefore, a single LogFile in LCache can accommodate most of the text-typed and image-typed datasets [3]. In this scenario, LCache could prevent remote data read after the first epoch through log reordering, improving cache utilization and data loading efficiency. Furthermore, when the LogFile is insufficient to contain the entire dataset, LCache would release accessed chunks from SSDs (line 16) and sequentially prefetch unused data from remote storage according to the shuffled list. Besides, LCache prioritizes reusing accessed files that will be loaded first in the next epoch (line 14), thus minimizing I/O-induced training stalls.

C. Metadata Indexing

LCache proposes an in-memory MetaIndex to record the metadata of all chunks and their contained sample files in the LogFile. MetaIndex uses a two-level indexing structure to provide efficient traversal and query capabilities. Specifically, the low-level utilizes linked list to record the exclusive metadata entry (i.e., *CM*) of each chunk according to the shuffled list within an epoch. Moreover, the high-level is a concurrent hash table that uses the file name field as the key to query the position of corresponding CM in the linked list, thus enhancing the efficiency of log reordering.

Each CM contains four fields: *CID* serves as a unique identifier for the chunk within an epoch; the *Offset* field records the storage location of the chunk in the LogFile; the *Size* field records the capacity of the chunk; the *Cached* flag indicates whether the chunk is cached in local SSDs. Additionally, the *Content* field records the metadata of all sample files contained in this chunk, including their original file name, file size and the storage offset within the chunk.

Algorithm 1: Log Reordering

Input: The number of epoches N; The number of batches M in each epoch; The number of dataset files C in each batch.

```
1  foreach epoch ∈ [0, N − 1] do
2      CurrLog = LCache.LogFiles[epoch%2]
3      NextLog = LCache.LogFiles[(epoch + 1)%2]
4      foreach batch ∈ [0, M − 1] do
5          CM = CurrLog.MetaIndex.next()
6          if CM.Cached == True then
7              chunk = read(curr_Log, CM.Offset, CM.Size)
8              files[C] = split(chunk)
9          else
10             files[C] = read(Remote Storage or HDDs)
11         Train DL model with files[C]
12         foreach file ∈ files[C] do
13             noffset = NextLog.MetaIndex.query(file).Offset
14             if noffset ≤ NextLog.capacity then
15                 Threadpool.write(NextLog, noffset, file)
16         Threadpool.evict(CurrLog, cmeta.offset, cmeta.size)
```

During the training process, LCache sequentially traverses the linked list to extract the *Offset* and *Size* fields, then fetches the entire chunk from LogFile by invoking a single read I/O request. After that, it splits the chunk into multiple sample files based on the emphContent metadata recorded in MetaIndex. LCache maintains MetaIndex instances for the current and the next epoch simultaneously, and releases the instance after epoch end to reduce memory usage. Our experiments show that generating MetaIndex for the TB-scale ImageNet-21K dataset (i.e., containing 14 million images) takes less than 15 second and requires about 10GB memory space.

D. Crash Consistency

The training servers may encounter system failures due to out-of-memory or software crashes [7], [31], thus losing the in-memory MetaIndex. For fast recovery, LCache saves essential information of currently used datasets, including the number of sample files, file names, and directories, in a specific location on the local SSDs. This approach prevents LCache from retrieving the low-performance layer during system recovery. Before training, LCache obtains the pseudo-random seed from the DL training framework to calculate the shuffled list and regenerate the MetaIndex instances. Additionally, users have to replace damaged servers in the event of hardware failures due to broken GPUs or network issues [32]. In such cases, all cached data on the server's SSDs will be lost. LCache fetches the datasets from remote storage and caches them into the local SSDs with a log-structured layout after training resumes.

IV. Evaluation

A. Experiment Setup

Platform. We implement LCache in the PyTorch framework and evaluate it on a training server with the configurations listed in Table I. Each server has 8 NVIDIA A800 GPUs, 8 Intel 4TB-sized P4610 SSDs as local caches and connects to a

TABLE III
DATASETS SPECIFICATIONS

Datasets	Sample Files (million)	Capacity (GB)	Average File Size (KB)
COCO-2017	0.11	25	165.2
ImageNet-1K	1.3	138	107.8
ImageNet-21K	14.1	1315	92.4

remote storage system via a 25Gbps NIC. The softwares used are CUDA-12.1 and PyTorch-2.1.

System setup. We evaluate `LCache` against the PyTorch builtin DataLoader (labeled as `PyTorch`) and `NoPFS` [16]. Specifically, PyTorch DataLoader maintains the default file and directory layout of datasets. NoPFS exactly predicts the data access order through the pseudo-random seed, then preferentially caches frequently accessed data into local SSDs and allocating other data in the low-performance layer. Besides, we simulate the case where no I/O-induced stalls occur during training and provide an upper performance bound (`NO I/O`), which is not realistic in practice.

Workloads. We evaluate LCache with four representative DL models as listed in Table II. We use three datasets and their detailed information is summarized in Table III. In most experiments, all the datasets can be entirely cached in the SSD. We also limit the SSD capacity to evaluate the performance of cache systems when a dataset is larger than local SSD's capacity.

B. Performance of Data Loading

Training Time. We first evaluate the training time in real DL training tasks with LCache and other approaches. We conduct training of all the four models across 100 epochs on the ImageNet-1K dataset. We use a batch size of 128 samples per GPU and activate 16 data loading workers. The chunk size is 13.3MB in average. Fig.4 reports the average training time per epoch and skips the first epoch that has consistently high variance due to remote data access. The epoch time with LCache is reduced by 47%, 45%, 39% and 19% than PyTorch with AlexNet, SqueezeNet, ResNet-50 and ViT, respectively.

We further break down the epoch time into two parts: data loading time (labeled as *I/O*), and other times including decoding and computation (labeled as *Others*). For the I/O-bound AlexNet and SqueezeNet models, LCache reduces data loading time by more than 91% and 80% compared to PyTorch and NoPFS respectively, thereby decreasing the I/O overhead from 45% to less than 6% of the total training process. Compared to the ideal NoIO, LCache generates less than 15 seconds of I/O-induced training stall per epoch. This is because LCache utilizes chunk aggregation to transform randomly small-sized file reads into sequentially large-sized chunk reads, fully leveraging SSDs' high throughput while reducing file system overhead.

Data Loading Throughput. We study the performance improvement of LCache compared to other approaches across different cache sizes, which is a major factor affecting data loading efficiency. Fig.5 illustrates the data loading throughput under fully and partial cached configurations. The former stores entire datasets on SSDs, while the latter allocates 12/70/600

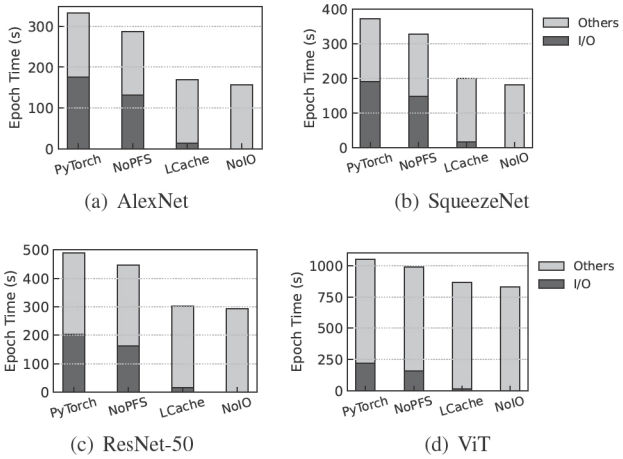


Fig. 4. Epoch time on training DL models with ImageNet-1K.

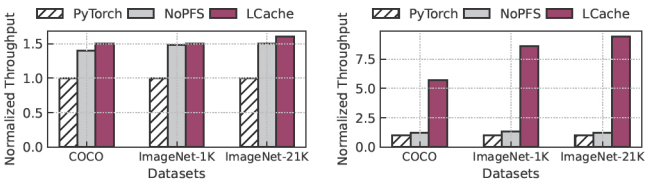
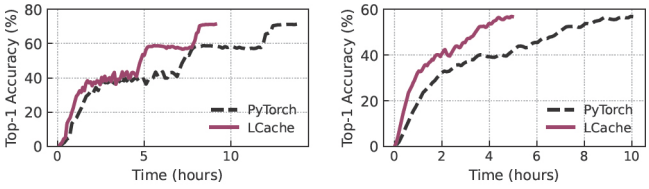


Fig. 5. Comparison of data loading performance with different cache capacity.

GB of SSD storage for COCO, ImageNet-1K and ImageNet-21K datasets respectively. We use the PyTorch DataLoader as the baseline. In the partial cached scenario, the throughput of LCache is 45% and 8% higher than that of PyTorch and NoPFS respectively. LCache significantly improves the read performance of cached sample files by 7.6× in average, but the Logfile for current training epoch only takes half of the total SSD capacity, thereby increasing the number of remote storage access.

Conversely, the fully cached system enables all sample files to be retrieved from the cache system after the first epoch. The data loading performance on NoPFS improves by less than 20% over PyTorch due to all the data being cached. In comparison, LCache fully leverages the advantages of log-structured data layout, achieving up to 9.4× and 8.6× more throughput than PyTorch and NoPFS, respectively. LCache employs the chunk-based data loading mechanism to retrieve Logfiles with sequential and large I/O requests, while performing real-time reordering to maintain the log layout across epochs. In modern DL training servers that incorporate tens of terabytes of SSD devices [27], [28], LCache can utilize the high capacity and I/O throughput of SSDs to efficiently accelerate data loading and reduce I/O-induced training stall.

End-to-end Training. Fig.6 shows the top-1 validation accuracy (i.e., the percentage of the model's answers exactly matching the correct answers) of ResNet-50 and AlexNet with LCache and PyTorch DataLoader, respectively. The dataset used is ImageNet-1K. Results first show that the training process performs exhibit similar learning curves because LCache does not change the random access order of the dataset, thus



(a) ResNet-50 (b) AlexNet

Fig. 6. Top-1 accuracy over training time

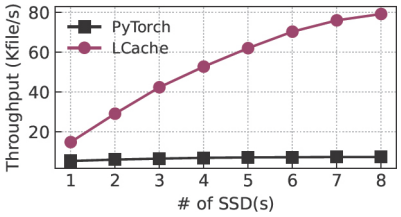


Fig. 7. LCache on different numbers of local SSDs

TABLE IV
PERFORMANCE OF LCACHE ON P40 GPUS WITH RESNET-50/IMAGENET-1K

	PyTorch	NoPFS	LCache
Average Epoch Time (seconds)	1851	1753	1443
Data Loading Throughput (GB/s)	0.65	0.78	6.8

not affecting the model accuracy or the training stabilization. Moreover, the total training time of ResNet and AlexNet under LCache is compressed by 39% and 49% on average compared with PyTorch due to the speedup in data loading.

C. Sensitive Study

Experiment with other GPUs. We evaluate the sensitivity of LCache on different types of GPU devices. We use a server that contains 8 NVIDIA Tesla P40 (24GB) GPUs, 8 Intel P4610 SSDs, and a 25Gbps NIC connecting to the remote storage system. We train ResNet-50 on ImageNet-1K in this experiment. Results in Table IV show that LCache on the P40 GPUs efficiently reduces the average epoch time by up to 22% compared with PyTorch, while incurring less than 4% I/O overhead on the training process. In addition, the performance gain of LCache on A800 GPUs is about 1.9× higher than that on P40 GPUs because the former platform has higher computational capability.

Experiment with different number of SSDs. Next, we evaluate the performance scalability of LCache built on different numbers of local SSDs, ranging from one to eight P4610 SSDs. We compose SSDs as a RAID-0 array. In this experiment, we invoke 16 workers to concurrently read sample files from the ImageNet-1K dataset on SSDs. Fig.7 shows that the data loading throughput of LCache and PyTorch under 8 SSDs is 5.4× and 1.2× than that with a single SSD, respectively. This is because LCache transforms random small file reads into sequential large chunk reads, efficiently leveraging the I/O capability among multiple SSDs while reducing file system overhead. Besides, LCache is also beneficial with other RAID configurations such as RAID-5.

V. RELATED WORKS

To enhance data loading throughput in model training, previous works [8], [9], [11], [16] leverage training server's memory and/or storage devices as local cache to reduce fetching data from remote storage systems. Quiver [8] employs a hash-based addressing mechanism to transparently reuse cached data across multiple model training tasks. DeepIO [14] deploys a memory buffer system between training servers and remote storage to prefetch and reshuffle sample files. Compared with these works, LCache utilizes local SSDs in each training server as a dataset cache and proposes a log-structured layout to minimize I/O overhead on reading cached data. Besides, DIESEL [33] packs sample files into large-sized chunks to reduce I/O overhead, but it must modify the shuffling algorithm to achieve the desired data retrieval order in different epochs. In contrast, LCache aggregates files according to the shuffled list and proposes the log reordering method to maintain the log layout across different epochs, which does not change the randomization performed by SGD and makes no impact on training efficiency.

Moerover, caching algorithms have been extensively studied in areas of storage systems. General cluster caching system like Alluxio [34] uses classic caching algorithms such as LRU and LFU to improve cache efficiency. For DL training tasks, CoorDL [11] populates local cache with a random set of sample files from the first epoch, and keeps these files unchanged during the training process. To improve cache hit rate, iCACHE [35] and SHADE [9] detect sample files that contribute more to model accuracy and preferentially place them in the cache system. SiloD [10] co-designs data caching and job scheduling to optimize cache allocation among multiple DL training tasks. Compared with these works, LCache provides an SSD-based cache system to improve data loading performance on model training, while strategically conducting log reordering to reduce remote data accesses and improve cache utilization.

VI. CONCLUSION

This paper introduces a log-structured caching mechanism LCache to accelerate data loading in model training. LCache utilizes the clairvoyance method to calculate the randomized file access order of SGD, then aggregates sample files into large-sized chunks and stores them in the log manner. In runtime, LCache retrieves cached data with sequential and large I/O requests, fully utilizing the I/O capability of SSD devices. Furthermore, LCache proposes a real-time log reordering method that strategically schedules accessed data to build log for the next epoch, thus accelerating data prefetching. Evaluations show that LCache achieves 9.4× and 7.8× data loading throughput improvement compared to PyTorch and NoPFS, respectively.

ACKNOWLEDGMENT

This work was supported by the Natural Science Foundation of Jiangsu Province Grant No. BK20240412, NSFC No. 62172175, No. 61821003 and Key Research and Development Project of Hubei No. 2022BAA042.

REFERENCES

[1] S. Zhang, S. Roller, N. Goyal *et al.*, "OPT: open pre-trained transformer language models," *CoRR*, 2022.

[2] OpenAI, "Chatgpt," https://openai.com/blog/chatgpt.

[3] T. Ridnik, E. B. Baruch, A. Noy, and L. Zelnik, "Imagenet-21k pretraining for the masses," in *Proceedings of the Neural Information Processing Systems Track on Datasets and Benchmarks 1, NeurIPS Datasets and Benchmarks 2021, December 2021, virtual*, J. Vanschoren and S. Yeung, Eds., 2021.

[4] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. S. Bernstein, A. C. Berg, and L. Fei-Fei, "Imagenet large scale visual recognition challenge," *Int. J. Comput. Vis.*, vol. 115, no. 3, pp. 211–252, 2015.

[5] J. Liu, C. S. Xia, Y. Wang *et al.*, "Is your code generated by chatgpt really correct? rigorous evaluation of large language models for code generation," in *NeurIPS 2023*, 2023.

[6] A. Ni, S. Iyer, D. Radev, and othersn, "LEVER: learning to verify language-to-code generation with execution," in *ICML 2023*, vol. 202, 2023, pp. 26 106–26 128.

[7] Z. Wang, Z. Jia, S. Zheng *et al.*, "GEMINI: fast failure recovery in distributed training with in-memory checkpoints," in *SOSP 2023*. ACM, 2023, pp. 364–381.

[8] A. V. Kumar and M. Sivathanu, "Quiver: An informed storage cache for deep learning," in *18th USENIX Conference on File and Storage Technologies, FAST 2020, Santa Clara, CA, USA, February 24-27, 2020*. USENIX Association, 2020, pp. 283–296.

[9] R. I. S. Khan, A. H. Yazdani, Y. Fu, A. K. Paul, B. Ji, X. Jian, Y. Cheng, and A. R. Butt, "SHADE: enable fundamental cacheability for distributed deep learning training," in *21st USENIX Conference on File and Storage Technologies, FAST 2023, Santa Clara, CA, USA, February 21-23, 2023*. USENIX Association, 2023, pp. 135–152.

[10] H. Zhao, Z. Han, Z. Yang, Q. Zhang, M. Li, F. Yang, Q. Zhang, B. Li, Y. Yang, L. Qiu, L. Zhang, and L. Zhou, "Silod: A co-design of caching and scheduling for deep learning clusters," in *Proceedings of the Eighteenth European Conference on Computer Systems, EuroSys 2023, Rome, Italy, May 8-12, 2023*. ACM, 2023, pp. 883–898.

[11] J. Mohan, A. Phanishayee, A. Raniwala, and V. Chidambaram, "Analyzing and mitigating data stalls in DNN training," *Proc. VLDB Endow.*, vol. 14, no. 5, pp. 771–784, 2021.

[12] J. Rasley, S. Rajbhandari, O. Ruwase, and Y. He, "Deepspeed: System optimizations enable training deep learning models with over 100 billion parameters," in *KDD '20: The 26th ACM SIGKDD Conference on Knowledge Discovery and Data Mining, Virtual Event, CA, USA, August 23-27, 2020*. ACM, 2020, pp. 3505–3506.

[13] D. Narayanan, M. Shoeybi, J. Casper *et al.*, "Efficient large-scale language model training on GPU clusters using megatron-lm," in *International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2021, St. Louis, Missouri, USA, November 14-19, 2021*. ACM, 2021, p. 58.

[14] Y. Zhu, F. Chowdhury, H. Fu, A. Moody, K. M. Mohror, K. Sato, and W. Yu, "Entropy-aware I/O pipelining for large-scale deep learning on HPC systems," in *26th IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems, MASCOTS 2018, Milwaukee, WI, USA, September 25-28, 2018*. IEEE Computer Society, 2018, pp. 145–156.

[15] A. Paszke, S. Gross, F. Massa *et al.*, "Pytorch: An imperative style, high-performance deep learning library," in *NeurIPS 2019*, 2019, pp. 8024–8035.

[16] N. Dryden, R. Böhringer, T. Ben-Nun, and T. Hoefler, "Clairvoyant prefetching for distributed machine learning I/O," in *International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2021, St. Louis, Missouri, USA, November 14-19, 2021*. ACM, 2021, p. 92.

[17] L. Bottou, F. E. Curtis, and J. Nocedal, "Optimization methods for large-scale machine learning," *SIAM Rev.*, vol. 60, no. 2, pp. 223–311, 2018.

[18] M. Gürbüzbalaban, A. E. Ozdaglar, and P. A. Parrilo, "Why random reshuffling beats stochastic gradient descent," *Math. Program.*, vol. 186, no. 1, pp. 49–84, 2021.

[19] M. Abadi, P. Barham, J. Chen, Z. Chen *et al.*, "Tensorflow: A system for large-scale machine learning," in *OSDI 2016*. USENIX Association, 2016, pp. 265–283.

[20] A. Krizhevsky and G. Hinton, "Learning multiple layers of features from tiny images," *Handbook of Systemic Autoimmune Diseases*, vol. 1, no. 4, 2009.

[21] C. Schuhmann, R. Beaumont, R. Vencu, C. Gordon, R. Wightman, M. Cherti, T. Coombes, A. Katta, C. Mullis, M. Wortsman, P. Schramowski, S. Kundurthy, K. Crowson, L. Schmidt, R. Kaczmarczyk, and J. Jitsev, "LAION-5B: an open large-scale dataset for training next generation image-text models," in *Advances in Neural Information Processing Systems 35: Annual Conference on Neural Information Processing Systems 2022, NeurIPS 2022, New Orleans, LA, USA, November 28 - December 9, 2022*, 2022.

[22] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in Neural Information Processing Systems 25: 26th Annual Conference on Neural Information Processing Systems 2012. Proceedings of a meeting held December 3-6, 2012, Lake Tahoe, Nevada, United States*, 2012, pp. 1106–1114.

[23] F. N. Iandola, M. W. Moskewicz, K. Ashraf, S. Han, W. J. Dally, and K. Keutzer, "Squeezenet: Alexnet-level accuracy with 50x fewer parameters and <1mb model size," *CoRR*, vol. abs/1602.07360, 2016. [Online]. Available: http://arxiv.org/abs/1602.07360

[24] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *2016 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2016, Las Vegas, NV, USA, June 27-30, 2016*. IEEE Computer Society, 2016, pp. 770–778.

[25] A. Dosovitskiy, L. Beyer, A. Kolesnikov, D. Weissenborn, X. Zhai, T. Unterthiner, M. Dehghani, M. Minderer, G. Heigold, S. Gelly, J. Uszkoreit, and N. Houlsby, "An image is worth 16x16 words: Transformers for image recognition at scale," in *9th International Conference on Learning Representations, ICLR 2021, Virtual Event, Austria, May 3-7, 2021*, 2021.

[26] A. Tavakkol, M. Sadrosadati, S. Ghose *et al.*, "FLIN: enabling fairness and enhancing performance in modern nvme solid state drives," in *ISCA 2018*. IEEE Computer Society, 2018, pp. 397–410.

[27] NVIDIA, "Nvidia hgx ai supercomputer," https://www.nvidia.com/en-us/data-center/hgx/.

[28] ——, "Nvidia dgx-2," https://www.nvidia.com/en-gb/data-center/dgx-2/.

[29] G. Lee, S. Shin, W. Song, T. J. Ham, J. W. Lee, and J. Jeong, "Asynchronous I/O stack: A low-latency kernel I/O stack for ultra-low latency ssds," in *Proceedings of the 2019 USENIX Annual Technical Conference, USENIX ATC 2019, Renton, WA, USA, July 10-12, 2019*. USENIX Association, 2019, pp. 603–616.

[30] X. Liao, Y. Lu, E. Xu, and J. Shu, "Max: A multicore-accelerated file system for flash storage," in *Proceedings of the 2021 USENIX Annual Technical Conference, USENIX ATC 2021, July 14-16, 2021*. USENIX Association, 2021, pp. 877–891.

[31] B. Nicolae, J. Li, J. M. Wozniak *et al.*, "Deepfreeze: Towards scalable asynchronous checkpointing of deep learning models," in *CCGRID 2020*. IEEE, 2020, pp. 172–181.

[32] J. Mohan, A. Phanishayee, and V. Chidambaram, "Checkfreq: Frequent, fine-grained DNN checkpointing," in *FAST 2021*. USENIX Association, 2021, pp. 203–216.

[33] 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 *ICPP 2020: 49th International Conference on Parallel Processing, Edmonton, AB, Canada, August 17-20, 2020*. ACM, 2020, pp. 20:1–20:11.

[34] H. Li, "Alluxio: A virtual distributed file system," Ph.D. dissertation, University of California, Berkeley, USA, 2018.

[35] W. Chen, S. He, Y. Xu, X. Zhang, S. Yang, S. Hu, X. Sun, and G. Chen, "icache: An importance-sampling-informed cache for accelerating i/o-bound DNN model training," in *IEEE International Symposium on High-Performance Computer Architecture, HPCA 2023, Montreal, QC, Canada, February 25 - March 1, 2023*. IEEE, 2023, pp. 220–232.