KSpeed: Beating I/O Bottlenecks of Data Provisioning for RDMA Training Clusters

Jianbo Dong[†], Hao Qi[‡], Tianjing Xu[§], Xiaoli Liu[†], Chen Wei[†], Rongyao Wang[†], Xiaoyi Lu[‡], Zheng Cao[†], Binzhang Fu[†]
[†] Alibaba Group, Beijing, China
[‡] Department of Computer Science, University of California, Merced, Merced, CA, USA
[§] China Construction Bank Operations Data Center, Beijing, China

***Abstract*—The rapidly-increasing computing power of GPUs has rendered the I/O subsystem a bottleneck for distributed deep learning (DL) training. Currently, substantial data preprocessing work (e.g., decoding) has to be conducted on CPUs for a wide range of training scenarios such as computer vision (CV) and audio. Unfortunately, the involvement of training nodes' host memory and/or CPUs on the critical path of loading data to GPUs incurs significant GPU stalls in modern RDMA training clusters, because CPUs are much slower than GPUs and the connection from PCIe switches to host memory tends to suffer from incast problems. Moreover, this also incurs high CPU usage and resource contention, which consequently causes data loading performance variation and stragglers.**

This paper presents *KSpeed*, a novel data provisioning framework for large-scale RDMA training clusters. As many data preprocessing tasks need to be done by CPUs, KSpeed organizes host memory and CPU resources in the cluster to build a disaggregated memory/CPU pool, where the nodes can read raw input data from backend storage to their host memory, preprocess the data by their CPUs if necessary, and write cached/preprocessed data (on demand) directly to the training workers' GPU memory to minimize GPU stalls. KSpeed leverages the multi-rail RDMA network to eliminate unnecessary memory copies, interference, and congestion. Evaluation on a 96-GPU cluster shows that KSpeed delivers 5.4$\times \sim$ 100$\times$ higher data loading performance over the state-of-the-art designs (DPP and Alluxio). KSpeed achieves near-linear scalability as the GPU number increases from 8 to 512.

***Index Terms*—I/O bottlenecks, RDMA, distributed deep learning**

I. Introduction

Deep Learning (DL) applications have gained significant success in many fields such as natural language processing (NLP) [38], [17], [46], computer vision (CV) [26], [21], [12], and audio. To meet the need for high accuracy and generality, the sizes of both input training datasets and DL models (i.e., deep neural networks or DNNs) have been increasing drastically in recent years [44] . For instance, the youtube-8M dataset is of about 1.5 terabytes (TB) [10], the Google OpenImages dataset has a total size of about 18 TB [28], and the OpenAI GPT-3 [38] language prediction model has 175 billion parameters and over 45 TB of input data. The recent GPT-4 has a context length of 8,192 tokens [7], $\sim$4$\times$ GPT-3's length.

The large sizes of datasets and models necessitate distributed DL training in GPU clusters [42], [40], [23]. During each training epoch, the complete dataset is divided into mini-batches [32]. These mini-batches are then trained in parallel on the training nodes, with each mini-batch being trained in one iteration. To ensure model synchronization, appropriate mechanisms such as parameter server (PS) [31], [24] or all-reduce [9], [13], [37], [41] are employed. The mini-batches of the input training dataset are typically read by the training nodes from a shared remote backend storage system.

Loading input training data from remote storage to GPUs is a severe bottleneck for distributed DL training, especially on those high-performance computing (HPC) or DL data center clusters [18], [14] equipped with expensive high-end GPUs. This is because the computing power of GPUs [36] is continuously increasing and the I/O speed of loading input training data is too slow to saturate the GPUs. For example, an NVIDIA A100 node in our high-end GPU cluster consists of eight A100 GPUs, each with 80 GB of high-bandwidth memory, achieving the floating-point (FP16) performance of a total of 5 PFLOPS (peta floating-point operations per second) [14]. Simple tests for training ResNet-50 [25] show that an A100 node can process about 17,000 200KB images per second [45], which imposes high I/O and preprocessing pressure on data provision to catch up with the GPUs' processing speed. The upcoming B100 GPU will be an order of magnitude faster than A100 [1].

Traditional distributed DL training systems usually have the *storage nodes* that store the original input data, and the *training nodes* that consume the input data with GPUs for DL training. For mainstream training frameworks like PyTorch [39], data loading and computation is tightly coupled, which requires the training nodes to read data from storage nodes, locally perform preprocessing using CPUs and host memory, and then asynchronously feed the preprocessed data into GPUs. As discussed in recent studies [27], [20], [47], this design is highly inefficient for the I/O path, as the training nodes have to temporarily hold the input data in host memory buffers. When the buffers are full, data loading must wait until the current mini-batch is completed, which significantly degrades the data loading speed [18].

Recent studies [27], [20], [47] have been focusing on the data loading problem, and found that (i) disaggregating data loading from model training is vital for high-performance DL training systems, and (ii) caching input data can effectively reduce the I/O bottlenecks. Therefore, researchers have introduced the intermediate layer of *cache nodes* (or I/O nodes) between the storage nodes and the training nodes, which perform data fetching, caching, and/or preprocessing in a dedicated manner. NVIDIA has also released the DALI library to support certain preprocessing operations (e.g., image resizing and rotation) to be done on GPUs.

From the perspective of GPU utilization, however, the state-of-the-art disaggregated data loading solutions are far from satisfactory, leaving GPUs suffering from costly stalls [20], [47]. As introduced by the NVIDIA DALI official website [8], the DALI library only

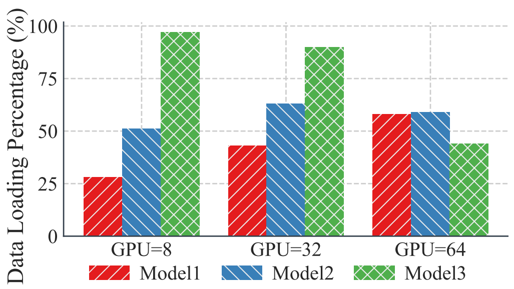


Fig. 1: Relative waiting time for loading data into GPUs.

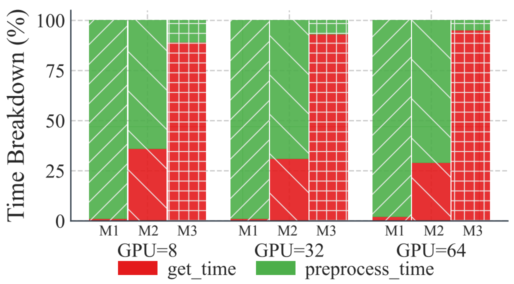


Fig. 2: I/O Bottleneck 1: The data loading time (get_time and preprocess_time) is significantly increased for all three models when the CPUs of training nodes are overloaded.

supports a small subset of the preprocessing operations to be done on GPUs, and thus substantial data preprocessing work (e.g., decoding) has to be conducted on CPUs for a wide range of training scenarios such as computer vision and audio. All existing DL training solutions adopt the DL frameworks' data loading architecture where training nodes transfer data from their host memory to the GPU memory, no matter whether data preprocessing is conducted by a disaggregated service (e.g., in DPP [47]), by the CPUs of training nodes (e.g., in most caching solutions like CoorDL [34], Quiver [27], DeepIO [48], and Alluxio [29], [30]), or partially by the GPUs (with DALI). However, the involvement of training nodes' host memory on the critical path of data loading still incurs severe GPU stalls in modern RDMA-enabled GPU clusters, because the CPUs are much slower than the GPUs and the connection from PCIe switches to host memory tends to suffer from incast problems. Moreover, as observed in many of our production training clusters, the involvement of host memory on the critical path often incurs high CPU usage and resource contention, which causes serious variation in data loading performance of the parallel GPU workers and consequently stragglers.

This study introduces KSpeed, a novel data provisioning framework for beating I/O bottlenecks of DL training. KSpeed disaggregates not only host memory but also CPU resources and offers two collaborative data provisioning acceleration services: data caching and data preprocessing. The data caching service organizes host memory resources to build an RDMA-based, disaggregated memory cache pool, while the data preprocessing service coordinates CPU and GPU resources in the cluster for data preprocessing tasks such as decoding, encoding, clipping, and rotating.

KSpeed's key innovations lie in three novel contributions: (1) leveraging the predictability of I/O patterns in DL workloads to prefetch data from remote storage into the memory pool of servers, (2) utilizing GPUDirect RDMA to proactively deliver data from servers directly to the GPUs of computing nodes, bypassing CPUs and host memory, and (3) optimizing multi-rail networks by binding each GPU to the NIC under the same PCIe switch to reduce congestion. By employing these strategies, KSpeed eliminates traditional request-response dependencies, alleviates system bottlenecks, reduces variations in data loading time, and fully overlaps data loading with computation, which ultimately leads to more efficient and scalable DL training.

KSpeed can be co-located with the training/storage nodes or be deployed as a standalone service. We have integrated KSpeed with PyTorch, and have deployed KSpeed in our production GPU cluster for various large-scale DL training workloads. Evaluation on a 96-GPU cluster shows that KSpeed delivers significantly higher I/O performance than the state-of-the-art data loading mechanisms equipped with memory-based storage and caching, for various training workloads. KSpeed can achieve over 100× improvement for fetching small samples and up to 5.4× improvement for fetching large samples compared to the state-of-the-art Alluxio [29] caching solution. Further, the data loading time of KSpeed is reduced by up to 5.4× and 20.7× for 3D-UNet [15] and CosmoFlow [33], respectively, compared to the state-of-the-art data loading solutions (DPP [47] and Alluxio). KSpeed achieves near-linear scalability as the number of GPUs increases from 8 to 512. KSpeed has been successfully deployed on our production cloud, proving its effective and efficient capabilities in various real-world deep-learning training applications.

II. Background and Motivation

Data loading from storage to GPUs is a performance bottleneck for distributed DL training. To quantify the effect of data loading on the overall performance of training tasks, we run three applications from our real businesses, where the models come from different fields and the datasets have different characteristics. The first model (Model1), whose backbone is based on ResNet50, is used for SKU (stock keeping unit) level product classification. The second model (Model2), 3D-UNet, is used for medical image segmentation. And the third model (Model3), Cosmoflow, is for climate prediction.

Fig. 1 shows the relative data loading time percentage in these tasks. All the tested applications spend a great portion of time (at least 28%) in waiting for the input data. Note that, the proportion of time spent on data loading is closely linked to the overall iteration time, which encompasses not only computing time but also communication time among the workers. For instance, as the system scale increases from 8 GPUs to 64 GPUs, the data loading overhead of Model1 increases from 28% to 58% but this overhead in Model3 decreases from 97% to 44%, as the running time of its entire training iteration increases more significantly, by 2.44× (not shown in this figure).

To understand the causes of data loading overhead, we further analyze the dynamics of the training system and running tasks in-depth. The results identify three major I/O Bottlenecks in large-scale distributed DL training systems.

I/O Bottleneck 1 – Excessive data transferring and preprocessing are required in DL training jobs, which rely heavily on the CPUs in training nodes. While RDMA NICs manage the physical data

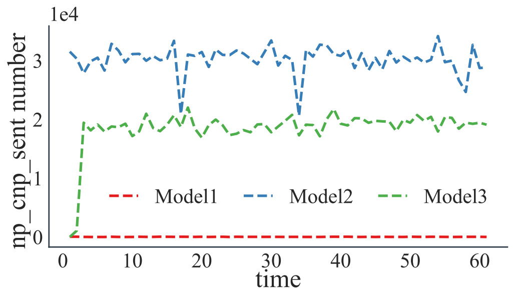


Fig. 3: I/O Bottleneck 2: CNPs indicate congestion caused by incast between PCIe switches and host memory.

transfer between nodes, the CPU is still heavily involved in managing preprocessing and coordination tasks. When the CPUs of the training nodes are overloaded, the data loading time is significantly increased. Fig. 2 demonstrates the breakdown of the data loading time into "get_time" (for transferring) and "preprocess_time" (for preprocessing). As shown in Fig. 2, the data loading time in Model1 is dominated by preprocessing ("preprocess_time"), while the data loading time in Model3 is dominated by data transferring ("get_time") and Model2 has comparable preprocessing time and data transferring time. Therefore, to serve various training applications with different characteristics, the data loading service should be able to minimize the cost in all cases.

I/O Bottleneck 2 – The state-of-the-art data loading mechanisms cannot fully utilize the underlying high-performance infrastructure. As shown in Fig. 3, we have observed a lot of CNPs (Congestion Notification Packets) sent from the training nodes in two tested cases (Model2 and Model3), indicating that the network traffic oversubscribes the local bus in the training nodes. This problem is because the high-end training nodes are equipped with multiple RDMA NICs to serve the intensively deployed GPUs, where the aggregated bandwidth for inter-node communication reaches 800 Gbps (8×100 Gbps), much higher than the total bandwidth (512 Gbps) of root ports in the host PCIe fabric (4 PCIe Gen3 ports). When the PCIe root ports are oversubscribed, the network interfaces would be back-pressured, which results in longer data loading time.

I/O Bottleneck 3 – The data loading time of different worker threads on the same training node has obvious variations. The data loading delay variations can depend on many factors such as I/O contention and CPU resource allocation. As shown in Fig. 4, the 32 workers (GPUs) have significantly varied data loading times when training Model2. The maximum data loading delay is about $1.8\times$ of the minimum. As most distributed training applications take the BSP (Bulk-Synchronous Parallel) model, the workers need to synchronize with each other at the end of each training iteration. Consequently, the completion time of a training iteration could be affected by the slowest worker (i.e., stragglers).

In this paper, we propose a data provision framework (called KSpeed) that addresses the aforementioned major I/O bottlenecks and other important issues, with the goal of achieving high performance and scalability in large-scale distributed DL training. To prevent the training system from being bottlenecked at CPUs of training nodes (preprocessing_time in I/O Bottleneck 1 in Fig. 2), KSpeed provides the data preprocessing service that can offload the heavy preprocessing tasks to the CPUs on the intermediate cache nodes. To mitigate the get_time in I/O Bottleneck 1 and address I/O Bottleneck 2 (Fig. 3), we design interference-free and contention-aware multi-rail GPU-Direct RDMA schemes to write the required data directly to the GPU memory, without making a detour to the host memory of the training nodes. This avoids the incast problem in PCIe root ports and makes full utilization of the high network bandwidth from modern high-end GPU clusters. By removing the involvement of compute nodes' CPUs and host memory from the critical path of data loading, KSpeed can also mitigate the variation in the data loading performance of GPU workers (I/O Bottleneck 3 in Fig. 4).

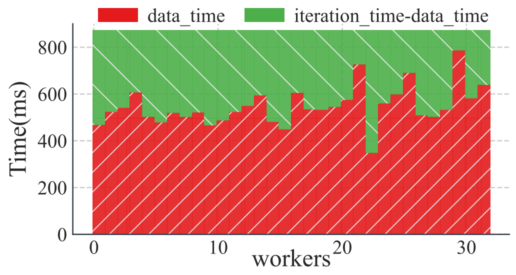
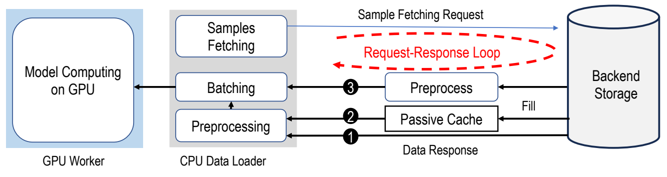
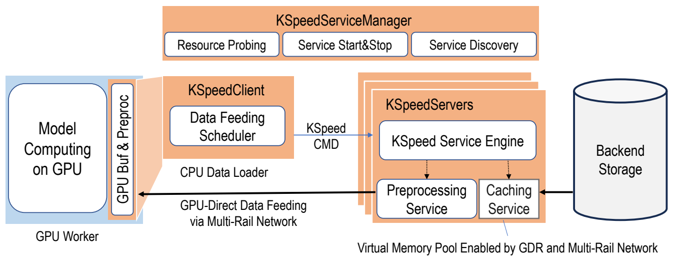


Fig. 4: I/O Bottleneck 3: The data loading time of different worker threads on the same training node has variations.



(a) Data loading pipelines of prior works



(b) KSpeed pipeline

Fig. 5: Comparison between KSpeed and prior studies.

III. Design

A. Overview

Fig. 5 compares the architecture of KSpeed with prior works. As shown in Fig. 5a, the default data loading pipeline in widely used frameworks like PyTorch is indicated by ❶, which reads the raw data from a remote storage and sends the data to GPU devices after preprocessing. The data loader with a cache layer design, such as Alluxio [2], Quiver [27], DeepIO [48], PacMan [11], Quartet [16], Clairvoyant [19], etc., is illustrated by ❷. The data loader with dedicated preprocessing techniques, like DPP [47], Cachew [20], MinIO [34], and scientific encoder/decoder [22], is denoted as ❸. In these prior works, the CPUs in training nodes are the central components in the data loading subsystem, which reads the data from the remote storage and delivers it to the GPU devices after preprocessing. It is crucial to note that traditional data loaders operate in a "Master-Slave" mode. Consequently, receive buffers and other data structures are allocated in host memory, which invariably means that responses from remote nodes are confined to being processed within the host memory. They inevitably create a request-response loop between the CPU data loaders and their passive respondents.

The KSpeed data loading pipeline is shown in Fig. 5b. Different from prior works, KSpeed realizes data loading under a "Client-Server" model, which breaks the conventional request-response dependency by enabling servers to deliver "replies" directly to the destination GPUs asynchronously. KSpeed removes the host memory in computing nodes from the critical path of data transferring, and provides precise data fetching scheduler, RDMA-enabled disaggregated memory-based I/O, flexible service configuration, and direct data feeding for GPUs.

The newly introduced modules in KSpeed include *KSpeedClient* that is deployed on the training nodes, *KSpeedServer* that can run on any nodes with available memory resources, and *KSpeedServiceManager* that is responsible for service and resources management. High-bandwidth multi-rail RDMA communication is implemented between the KSpeed components. Based on the efficient communication and memory management designs, we build the KSpeed system by jointly managing resources in authorized nodes in the GPU cluster.

KSpeedServers collect the CPU and memory resources from available nodes to form an intermediate disaggregated memory system enabled by RDMA, where the training dataset will be preloaded. They interact with GPU computing nodes by directly writing preprocessed data into GPU memory using GPU-Direct RDMA, avoiding the CPU and host memory in the critical data transfer path, and they optimize multi-rail networks by binding each GPU to the closest NIC, reducing PCIe congestion and improving data transfer performance. KSpeedServers provide the caching service and data preprocessing service based on the intermediate disaggregated memory based data storage layer. The data caching service responds to data cache registration, allocation of memory resources, and storing datasets in the intermediate disaggregated memory based data storage layer. The data preprocessing service is responsible for data preprocessing registration and runs user-specified preprocessing operations. It can proactively write the preprocessed data directly to the destination GPU memory. The data preprocessing service can be configured to pass or bypass depending on whether the dataset is registered for preprocessing operations. KSpeedServers are independent of each other and jointly form a data service cluster accessed by KSpeedClients.

KSpeedClients provide deep learning training jobs with a highly-optimized data provisioning service. The core functions of the data provisioning service include preloading from backend storage and proactive data feeding to computing components. During the data preloading stage, KSpeedClients first send memory allocation requests to the KSpeedServiceManager according to the training dataset size. It will then allocate memory from the disaggregated memory system to form a virtual contiguous addressing space. Then KSpeedClient will indicate the servers to read the training dataset from backend storage and write them to the virtual space. During training, KSpeedClients send the data load requests (preprocessing may be included) to servers with indications of the destinations for replies. The servers will write the data (possibly before/after preprocessing) to their designated devices directly (data plane), without involving host CPU/memory in training nodes.

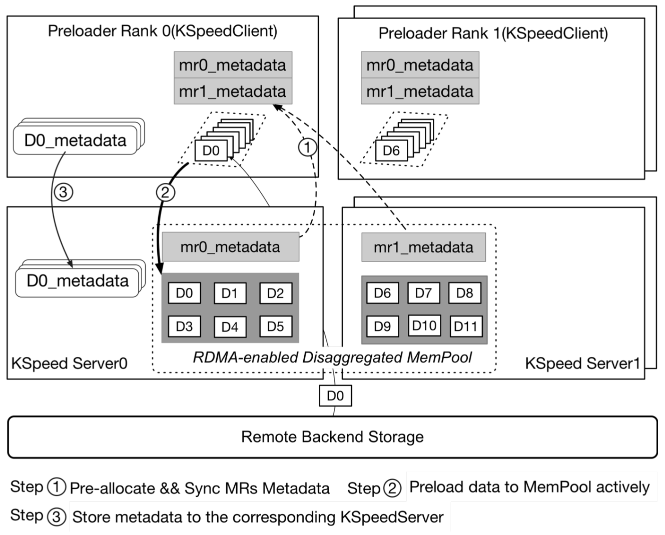

Preloader Rank 0(KSpeedClient)
Preloader Rank 1(KSpeedClient)
mr0_metadata
mr1_metadata
D0_metadata
D0
D6
①
②
③
mr0_metadata
mr1_metadata
D0 D1 D2
D3 D4 D5
D6 D7 D8
D9 D10 D11
KSpeed Server0
RDMA-enabled Disaggregated MemPool
KSpeed Server1
Remote Backend Storage
Step ① Pre-allocate && Sync MRs Metadata
Step ② Preload data to MemPool actively
Step ③ Store metadata to the corresponding KSpeedServer

Fig. 6: Proactive preloading based on predictable order.

The KSpeedServiceManager is the central control unit of KSpeed, in charge of service initialization, discovery, restart, termination, state tracking, and exception handling, etc. By default, an empty server is deployed in each node to collect information about available resources in the cluster. This information is then used to initialize services when new requests are received. Once resources have been allocated for the servers, their addresses are broadcast to the clients and connections are established between them. The KSpeedServiceManager also continuously monitors the running state of these servers and handles any exceptions that may occur.

B. Dataset Preloading

Fig. 6 shows the process of preloading a dataset, i.e., reading data from a remote backend storage to the RDMA-enabled disaggregated memory cache by KSpeedClient, including the following steps.

(1) Resource pre-allocation: In this step, KSpeedServer pre-allocates memory regions (MRs) for data caching and synchronizes MRs' metadata to KSpeedClient. In KSpeed, each memory region (MR) serves as the fundamental unit of management, and is designed with a fixed size, such as 128MB. Before the start of each training process, KSpeed initiates a preloading process. During the initialization phase of a training task, the Preloader Rank 0 (as shown in Fig. 6) associated with the Rank0 training process traverses the entire dataset to determine its size. Subsequently, KSpeedServer allocates a portion of available disaggregated memory based on the size of the dataset. The MRs pre-allocated to the same dataset are uniformly addressed to form the disaggregated memory pool, and the dataset is preloaded to MRs by calculating the data storage location based on the order.

(2) Data filling: In this step, KSpeedClient populates the memory pool with data. To expedite the preloading process, all metadata of the allocated memory regions (MRs), such as `mr0_metadata` and `mr1_metadata` in Fig. 6, are synchronized. This allows the source data to be efficiently written into the disaggregated memory via RDMA operations. Additionally, the MR-based design aggregates write operations of small data preloading in step 2, as illustrated in Fig. 6, to enhance overall preloading performance. The KSpeed Preloader packages the data retrieved from remote backend storage and writes it all at once. Each preloading process is assigned to a unique partition of the allocated MRs based on the rank number of the corresponding training process. If the allocated MRs can accommodate the entire dataset, KSpeed Preloaders will actively preload the entire dataset during the training initialization phase, according to the traversal order of the dataset, and preload it only once. If the allocated MRs are insufficient to accommodate the entire dataset, a preloading procedure will be executed in each training epoch. Each preloading process will preload the data in the exact order used in each training epoch, achieving a 100% hit rate, which overlaps with the prefetching procedure described later.

(3) Decentralized metadata management: In this step, KSpeed-Client stores the dataset's metadata (not the MR's metadata) back to the corresponding KSpeedServer. Unlike storing all metadata in a master metadata server, KSpeed uses a decentralized metadata management mechanism that stores metadata in the KSpeedServers that hold the data to avoid single-point bottlenecks. As depicted in Fig. 6, `D0_metadata` is located with `D0` in KSpeedServer0. Without central management, KSpeed adopts a broadcast-reply mechanism to fetch metadata during proactive data prefetching. KSpeedClient broadcasts the required data sample IDs to all KSpeedServers, and the corresponding KSpeedServer responds with the metadata (Step 1 in Fig. 7). This mechanism is efficient in DL training scenarios. As the samples to be read in each epoch are predictable, we can perform a precise sample ID broadcast.

C. Data Feeding and Preprocessing Offloading

Proactive data feeding involves transferring data from the disaggregated memory system to the training nodes and providing data provisioning services for DL training jobs. Compared to dataset preloading, data feeding performance plays a more crucial role, particularly when the datasets are entirely cached in the KSpeed disaggregated memory system. As a result, several crucial optimizations are implemented.

(1) Precise data feeding: In the first place, the training workloads typically exhibit predictable I/O requests due to the epoch and mini-batch-based data access patterns. For example, the PyTorch data loader shuffles the indices of the samples across the dataset at the beginning of each epoch. The shuffled index vector is then evenly divided into sub-vectors, and each training worker is assigned a sub-vector of indices to load the indexed samples for training, one batch after another. Therefore, we can obtain the access sequences of the samples during the training initialization phase, and even calculate the access sequences for the following epochs. With the known access sequences, the data feeding flow in KSpeed (as shown in Fig. 7) proceeds as follows: At the beginning of each epoch, the index list of required data is generated, and the corresponding metadata is synchronized back from KSpeedServers.

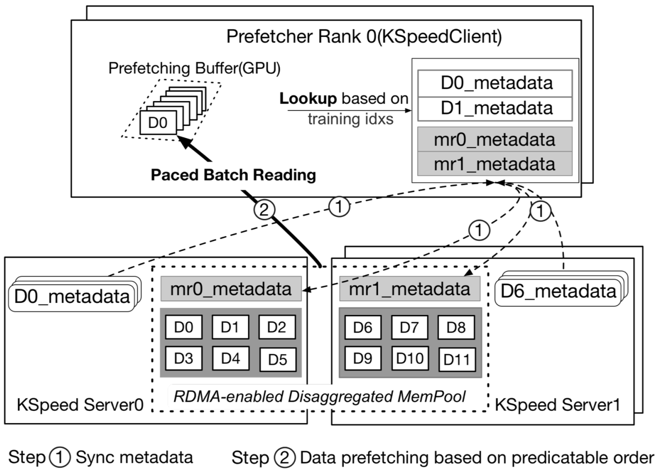


Fig. 7: Proactive feeding from disaggregated memory.

Then, KSpeed performs accurate data feeding by sending the data to the preset buffer in local computing devices (like GPU) using the addresses in the metadata (Step 2 in Fig. 7).

(2) Data preprocessing offloading: The unprocessed input data requires specific formatting before it can be utilized by a model. In KSpeed, a streamlined approach to this preprocessing service is to delegate it to KSpeedServer, allowing it to overlap with other stages in the data loading pipeline such as the I/O with backend storage and the data transmission to worker nodes. Moreover, the preprocessing service is configurable and can be initiated prior to and/or subsequent to the data caching. Users have the flexibility to choose whether to cache preprocessed data and/or preprocess cached data. Similarly, users have the flexibility to append GPU-based preprocessing operations to the data on the client side, if GPUs are not fully utilized. Finally, it is noteworthy that this data preprocessing service is not mandatory in KSpeed, as users can opt to enable or disable it based on the nature of the training tasks.

(3) Efficient data provisioning interface: If we solely offer a bitstream reading interface, users would frequently need to convert it into other formats such as `numpy` to perform data preprocessing, leading to additional memory copying. To mitigate the data format conversion overhead between KSpeedClient and the upper application, KSpeed presents the subsequent high-performance data provisioning interfaces:

(a) Cast-free data provisioning interface: KSpeed furnishes a data reading interface that offers an array memory view, which empowers user programs to use the array directly or transform it into a tensor for data preprocessing, without incurring any memory copy overhead. In addition, KSpeed also presents tailored interfaces for frequently employed image and voice datasets. For instance, for `wav` datasets [5], KSpeed provides an interface that yields a numpy-formatted speech data, with a 44-bit offset from raw data.

(b) GPU-Direct data provisioning interface: During a deep learning training task, the data is fetched from a remote storage and assembled into batch data following a sequence of preprocessing operations, before eventually being transferred to GPU memory for model training. As the PCIe interface has become the bottleneck compared to the modern high-bandwidth network, KSpeed incorpo-

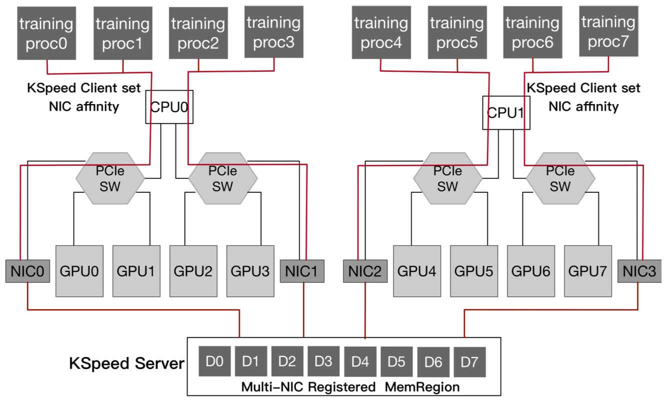


Fig. 8: Accelerating input I/O through multi-rail network

rates the GPU-Direct data feeding mechanism. Specifically, KSpeed proactively delivers data to GPU memory, offers users a view of the `cuda_array_interface`, connects to user-defined preprocessing operations, and minimizes the overhead of I/O.

D. Optimization for Multi-Rail Network

We have integrated KSpeed with PyTorch, and deployed it in our clusters with multi-rail RDMA network. In real-world large-scale distributed DL training tasks, each training process typically operates with 4 or even 10 I/O threads for data loading. In a common configuration of a single training node with 8 GPUs, dozens or even hundreds of data loading threads may execute a large number of concurrent data I/O accesses. Traditional data caching solutions can only employ *ONE* NIC for data I/O access and are unable to harness high-bandwidth and low-latency multi-rail GPUDirect RDMA networks in high-end GPU clusters. To address this, KSpeed designs the following data loading optimization schemes.

(1) Interference-free mapping: The mapping relationship between the NICs and the data loading processes for GPU training is critical for effectively utilizing a multi-rail RDMA network. One possible approach is to allow each GPU to use all network cards, while another is to restrict each GPU to a pre-specified NIC. While the former approach may seem to enable each GPU to access bandwidth from all NICs, our experiments have shown that the out-of-order mapping between GPUs and NICs introduces significant performance interference under the GPU server architecture where multiple GPUs coexist. The reasons are as follows. Firstly, due to the existence of the PCIe tree topology in the server, the network bandwidth that can be directly utilized by a single GPU is limited by the upstream PCIe port. Therefore, the highest achievable bandwidth is the same as that of a single PCIe port. Secondly, although the PCIe specification includes a QoS definition, the actual chip implementation lacks support for it. This results in a chaotic PCIe communication pattern that potentially causes performance degradation. Hence, based on the characteristics of PCIe topology, we bind a GPU with a specified NIC so that communication between the GPU and NIC is restricted to a PCIe switch chip, and the traffic flows among different GPUs do not interfere with each other.

As depicted in Fig. 8, in order to facilitate multi-rail RDMA transfers, the memory regions administered by each KSpeedServer are registered with all NICs during the RDMA registration procedure. Prior to initializing data loading, KSpeedClient proactively identifies the GPU number being utilized by the training process, selects the closest NIC to this GPU, sets NIC affinity, and subsequently employs the NIC for data loading. For example, in Fig. 8, the GPUs allocated to training proc0 and training proc1 are GPU0 and GPU1, respectively. Therefore, the KSpeedClients linked to proc0 and proc1 will employ NIC0 for data loading, given that NIC0 is in closest proximity to GPU0 and GPU1 within the PCIe topology. Meanwhile, to circumvent the introduction of remote non-uniform memory access across CPUs, KSpeedClient will designate the CPU core that is nearest to the GPU and set the CPU affinity of each training process.

(2) Congestion-aware requests batching: To optimize the data transfer performance of numerous small samples, KSpeedClient incorporates batch sending, which combines small requests into a batch request. However, we have observed that batch requests can cause multiple KSpeedServers to send data to the destination simultaneously. When the returned data surpasses the processing capacity, congestion backpressure may arise, leading to a dramatic degradation in GPU-Direct RDMA-based I/O performance.

In order to solve the congestion problem of batch requests, we propose to pace the batch data sending based on the proactive flow control at the request side. Its key idea is to control the speed of inflight data-sending requests within the processing capability of the multi-rail network. We first set several data size categories (e.g., 32KB, 256KB, and 1MB) and measure the transmission delay (latency) of each data size. Then, when the data feeding process is initialized, the limited number of online reading requests allowed for each process is calculated according to the measured delay, the actual data size, and the number of loading processes, which can represent the current processing capability. This allows KSpeedClient to dynamically split batch readings into multiple sub-batch readings, thereby keeping the number of inflight data requests within the processing capability limit. This contention-aware multi-rail GPU-Direct RDMA scheme can write the required data directly into the GPU memory, without making a detour to the host memory of the training nodes. Designing this request batching approach with the multi-rail network results in a significant reduction in congestion and a performance improvement in data transferring.

E. Discussion on Deployment Experience

KSpeed has been successfully deployed on our production cloud, proving its high-performance capabilities in various real-world deep-learning training applications. We demonstrate its effectiveness in data preloading acceleration within our large-scale AI Computing Service. Here, we discuss two pivotal real-world deployment experiences involving KSpeed in large-scale system implementations. This journey has been insightful.

Benefits of disaggregation: In our deployment of KSpeed, we found that resource disaggregation is vital to tackle I/O bottlenecks in DL training. KSpeed collectively organizes host memory and CPU resources in our production AI clusters where GPUs are much more precious. This strategy provides an excellent opportunity for KSpeed to minimize GPU stalls. The key lies in its proactive service agents, with predictable DL workload patterns, which preload raw input data, perform preprocessing, and seamlessly dispatch it to GPU memory. This proactive approach decouples the traditional request-response sequence, reducing variations in data loading time and optimizing DL training efficiency. The benefits observed in real-world deployments are twofold: improved resource utilization for cloud vendors and reduced training time and monetary cost for users. KSpeed organizes distributed resources into disaggregated caching and preprocessing services for smoother and more synchronized training sessions, ultimately enhancing both performance and cost-effectiveness.

High availability: KSpeedServers might crash or stop responding for many reasons, including hardware/software failures, resource shortage and contention, etc. To tackle this, we have integrated high availability mechanisms to KSpeed, making sure the training system keeps its stride. To keep the system simple and robust, we do not adopt data redundancy across servers. Instead, our system is crafted to recover lost data directly from the backend storage. If data corruption rears its head, KSpeed takes charge by reinitiating the data retrieval process, reading the necessary data from the backend.

In case a KSpeedServer encounters operational issues, we have implemented a rapid response system to address the situation. A replacement server on an alternative set of spare resources is promptly activated to ensure the uninterrupted flow of operations. Adding an extra layer of vigilance, the KSpeedManager regularly runs health checks on all KSpeedServers. This not only ensures optimal performance but also guarantees the high availability of the system.

IV. Evaluation

A. Experimental Setup

Our cluster features a diverse array of nodes, encompassing 332 nodes equipped solely with CPUs, 192 nodes powered by NVIDIA A100 GPUs, and 384 nodes outfitted with NVIDIA A10 GPUs (not used in this work). This hybrid configuration enables a wide range of computational capabilities to accommodate various workload requirements. The evaluations in this paper are conducted on two testbeds. The first testbed is a cluster of 12 A100 GPU nodes and 16 CPU nodes. The hardware architecture is shown in Fig. 8. Each GPU node is equipped with 8 NVIDIA A100 GPUs, dual-socket Intel(R) Xeon(R) Platinum 8269CY CPU @ 2.50GHz, and 1TB memory. All GPU nodes are connected to HDR InfiniBand interconnect with four ConnectX-6 NICs, which achieve an aggregate of 800 Gbps bandwidth for each node. The CPU nodes are used as storage nodes and installed with one NIC per node. The first testbed is primarily utilized for performance studies. The second testbed is similar to the first, but it is sourced from one of our real-world large-scale production GPU clusters. The second testbed is used for scalability studies, enabling experiments to be conducted with up to 512 GPUs.

State of the arts: We choose one representative state-of-the-art solution for comparison from each path discussed in Fig. 5a.

- PyTorch: The original PyTorch Pipeline, with built-in DataLoader where data input comes from backend GlusterFS [3].
- Alluxio: The original PyTorch Pipeline, with built-in DataLoader where data input comes from in-memory cache represented by Alluxio [30], [29] (the state-of-the-art open-source data orchestration framework for DL training in the industry).
- DPP: We also emulate the (closed-source) DPP framework described in [47] for comparison. DPP is a disaggregated preprocessing service for recommendation models. It is emulated with KSpeed by offloading the data preprocessing to KSpeed-Server and loading preprocessed data during the training loop.

Workloads: The tested applications are from real businesses.

(1) An in-house application for commercial products classification takes ResNet-50 [6] as the backbone. Its data-loading I/O pipeline consists of decoding jpeg image formats into pixels represented by 8-bit values, applying resizing, random cropping, flipping, to obtain a $3 \times 224 \times 224$ tensor. We train this model on a dataset with each image data size ranging from 200KB to 2MB, and a total size of more than 100GB.

(2) 3D-UNet [15] is a Convolutional Neural Network that has achieved promising results in medical image segmentation. Its data-loading I/O pipeline includes RandomCrop, Left-Right RandomFlip, Up-Down RandomFlip. We train 3D-UNet on the LITS 2017 dataset [4], with the size of a single sample is about 30MB. In the experiment, we use resampling and normalization to expand the dataset by 100 times, and the total size is about 380GB.

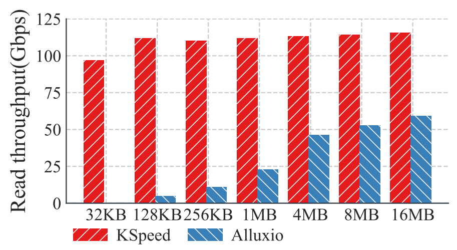
(3) CosmoFlow [33], the MLPerf-HPC CosmoFlow model, is a highly optimized and scalable deep learning application that predicts the cosmological parameters that describe the nature of the universe given the 3D matter distribution. Its data-loading I/O pipeline consists of only one log operation for smoothness. Here, we use its PyTorch implementation introduced in [19]. We train CosmoFlow on a simulated dataset with a size of 17MB per sample and 8.6TB in total.

The evaluations in this section are structured as follows. Initially, we assess the data loading performance of KSpeed using microbenchmarks, which do not require data preprocessing. Next, we evaluate the data loading performance of KSpeed when data preprocessing is implemented, by excluding the model training from the evaluated real-life tasks. Lastly, we evaluate KSpeed using end-to-end applications and illustrate the scalability of large-scale training systems when employing the full KSpeed service.

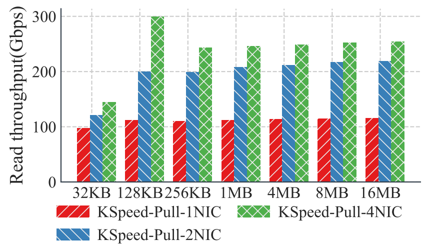
B. Data Loading without Preprocessing

As a microbenchmark, we executed a collection of data loading tasks to assess the throughput of KSpeed. We varied the sample sizes from 32KB to 16MB, which can represent typical data sample sizes in real-world business scenarios. As one key service provided by KSpeed is the caching service based on its high-performance disaggregated memory pool, we evaluate KSpeed against Alluxio [30], [29]. Throughout the experiments, we assigned 4 clients to each worker, resulting in 32 clients for an 8-GPU node. As the samples in DL training are read one at a time, the I/O depth of the read request during testing is set to 1. We designated 4 servers to provide caching service for both KSpeed and Alluxio.

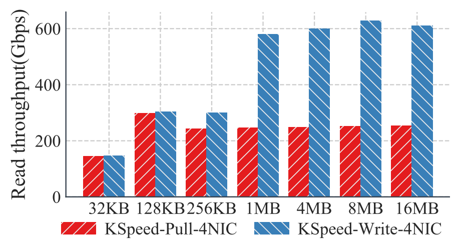
1) Improvement from the Optimizations in KSpeedClient and KSpeedServer: As mentioned before, some optimizations have been adopted in KSpeedClient and KSpeedServer, such as the decentrailized metadata and cast-free data provision interface. Here, we will evaluate the effectiveness of these optimizations with traditional data loading mechanism, which we call `Pull` operation. That is, we first load the data from KSpeedServer to local host memory, and



(a) KSpeed runs faster than Alluxio due to the optimization in KSpeedClinet and KSpeedServer

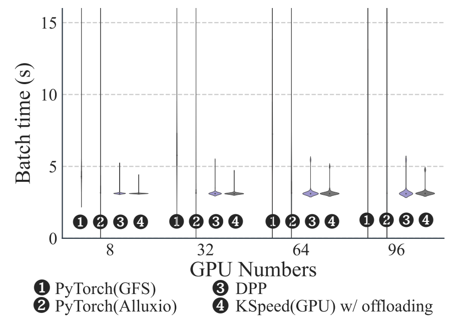


(b) KSpeed runs even faster when multi-rail network is fully utilized

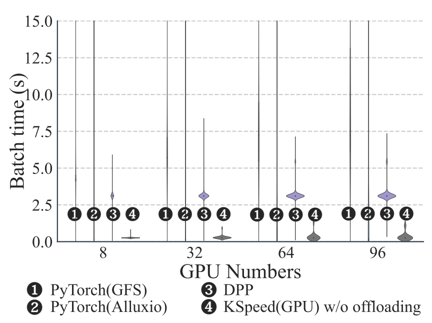


(c) KSpeed reaches its peak performance with GDR-based data writing

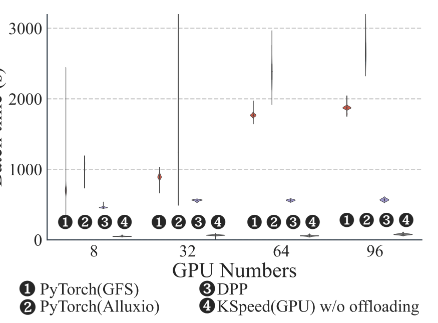
Fig. 9: Data loading performance without preprocessing.



(a) In-house application



(b) 3D-UNet



(c) CosmoFlow

Fig. 10: Data loading performance with preprocessing.

then send to GPUs. Since prior arts are not able to adopt multiple NICs in our evaluation system, we use only *one* NIC in this case.

Fig. 9a presents a performance comparison of the aggregate read throughput of multiple clients running on one node with one 200 Gbps NIC enabled. For small samples, whose sizes range from 32KB to 256KB, KSpeed achieves over 100× performance improvement over Alluxio. For large samples ($\geq$ 1MB), KSpeed still achieves up to 2× higher performance improvement.

2) Improvement from Multi-Rail Network: Realizing the limitation of prior arts, as illustrated in previous sections, KSpeed is designed to provide efficient support for multi-rail networks. In this experiment, we investigated the impact of the number of NICs on the performance with traditional data loading mechanism.

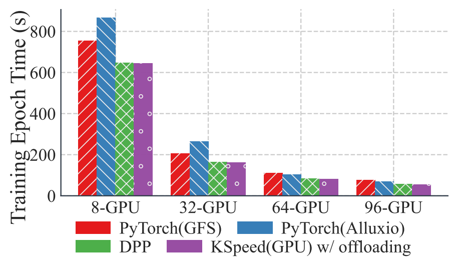
Fig. 9b demonstrates the aggregated read throughput of 32 clients with 1, 2, and 4 NICs. Our results indicate that the total throughput under 2-NIC is 1.25× to 1.9× higher than the 1-NIC configuration. However, the 4-NIC system does not exhibit significant improvement, and its performance is only 2.2× higher than the 1-NIC configuration. Upon profiling the NICs, we discovered that in the 4-NIC test, a large number of TX_PAUSE appeared on the NICs, indicating that the system I/O bus (PCIe) was oversubscribed. Once the upstream ports in PCIe fabric became congested, the downstream NICs experienced back-pressure, leading to significant bandwidth waste. This result further demonstrates the effectiveness and requirement of our proposed multi-rail design and optimizations (Sec. III-D).

3) Improvement from GPUDirect RDMA Data Write: In this case, we shall evaluate the efficacy of the unique `Write` operation supported in KSpeed. That is, the KSpeedServers write data mini-batches directly to the destination GPUs' device memory, without involving the CPUs and host memory in computing nodes. As shown in Fig. 9c, the `Write` operation displays a performance that is approximately 2.5× faster than that of without `Write`. Combined with the multi-NIC optimization, it gives about 5.4× improvement (compared to no `Write` under single-NIC configuration). As mentioned in Section III-C, the benefits of `Write` come from the awareness of multiple NICs in training nodes and avoidance of the congestion in the upstream ports of PCIe fabric. When the sample size is relatively small (32KB to 256KB), there is no notable traffic congestion in this test.

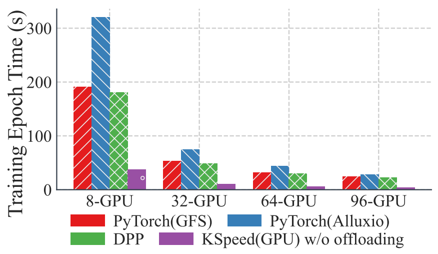
C. Data Loading with Preprocessing

In this subsection, we test the performance of data loading with preprocessing by bypassing the actual model training phase. We executed the tested applications, including our in-house application, 3D-UNet, and Cosmoflow, on different system configurations. We used the default data loader configuration: using 10 data loading clients for the in-house application, 4 for 3D-UNet, and 8 for CosmoFlow. We then ran the tested applications for 3 epochs while skipping the first epoch as it consistently exhibited high variance. We compare the data loading performance with preprocessing based on KSpeed (GPU), DPP, PyTorch (Alluxio), and PyTorch (GFS).

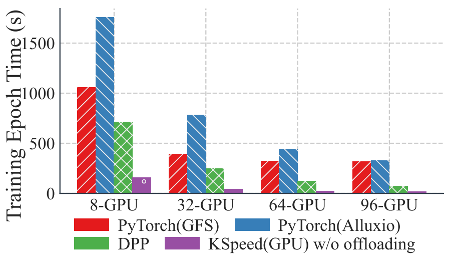
Fig. 10a depicts the evaluation results of the in-house application, which illustrates the distribution of the loading time for a batch of data. From the figure, we observed that KSpeed had the least variation for all the test cases. Additionally, DPP yielded similar results because, as mentioned earlier, the tested in-house application is preprocessing-dominated. If we employ the same offloading method, we expect KSpeed to perform similarly as DPP.



(a) In-house application

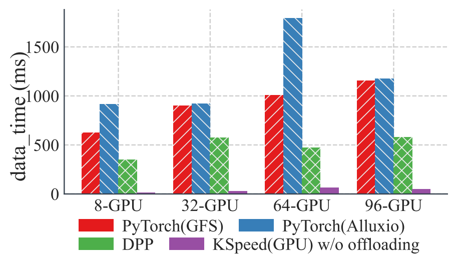


(b) 3D-UNet

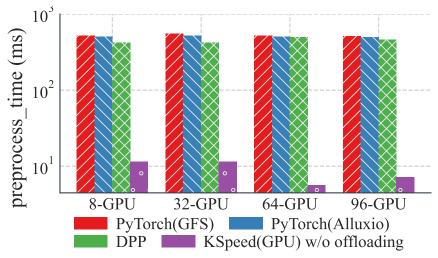


(c) CosmoFlow

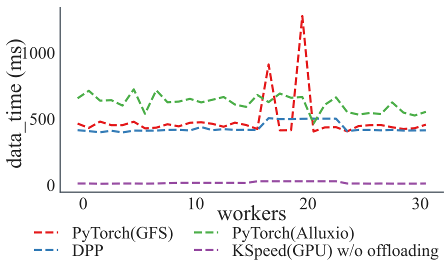
Fig. 11: Average end-to-end training epoch time.



(a) Average data waiting time of each job



(b) Average preprocessing time of each iteration



(c) Data waiting time variation for GPU workers

Fig. 12: The time breakdown for 3D-UNet jobs.

Fig. 10b and Fig. 10c demonstrate that the results for 3D-UNet and CosmoFlow follow a similar trend as the in-house application. That is, KSpeed exhibits the lowest loading delay and the least variation. Unlike Fig. 10a, KSpeed performs much better than DPP for these two sets of tests. This can be attributed to the fact that 3D-UNet and CosmoFlow require less preprocessing, making the benefits of KSpeed on data transferring more apparent. With proactive data feeding to the GPU, KSpeed can yield significant performance improvements for transferring-dominated applications.

D. *End-to-End Evaluations on Real-life Tasks*

In this subsection, we present a series of end-to-end evaluations on three real-life workloads. We compare the average epoch time still based on KSpeed, DPP, Alluxio, and PyTorch. For each workload, we conduct a distributed training job by varying the number of GPUs from 8 to 96. Both KSpeed and Alluxio are configured with a 2TB cache capacity. For the in-house application and 3D-UNet, the entire dataset is preloaded in KSpeed and Alluxio at the first epoch of training. For Cosmoflow, the dataset is too large (8.7TB) to be fully preloaded. Instead, it is preloaded in KSpeed in a predictable order. For Alluxio, the data is loaded and cached with its default cache replacement policy.

1) Performance of KSpeed Offloading: As the in-house application requires intensive data preprocessing, the KSpeed pipeline is configured to not only preload its dataset to the KSpeedServers, but also offload the data preprocessing to KSpeedServers. KSpeed proactively feeds preprocessed data to the GPUs during the training loop. Figure 11a depicts the average epoch time for the in-house application. According to the figure, KSpeed exhibits up to a 1.28× improvement in training performance over PyTorch (GFS) and a 1.63× improvement over the Alluxio caching mechanism. Meanwhile, KSpeed is about 9% faster than DPP in the training epoch time. These results demonstrate that the primary source of performance improvement in KSpeed is the offloading of data preprocessing, and the remaining 9% improvement is attributable to directly writing data to GPU memory. Furthermore, the Alluxio caching mechanism performs less efficiently in all test cases, which is mainly because of the inefficiency of accessing the centralized metadata.

2) Performance of Proactive Data Feeding to GPU: For 3D-UNet and CosmoFlow, KSpeed applies a different way to offload the data preprocessing. This time, KSpeed has the GPU to do data preprocessing after the raw data is fed into the device memory. As the preprocessing operation involves mainly tensor transformations, significant performance gains can still be achieved compared to prior methods that rely on CPUs in training nodes for preprocessing, without incurring additional computing resources. The evaluation results are shown in Fig. 11b and Fig. 11c. For 3D-UNet, KSpeed achieves up to 5.9× performance improvement over PyTorch(GFS), 8.5× over the Alluxio caching mechanism, and 5.4× over DPP. For CosmoFlow, KSpeed provides up to 20× performance improvement over PyTorch(GFS), 20.7× over the Alluxio caching mechanism, and 5.8× over DPP.

To manifest where the performance gain comes from, we review the dynamics of the 3D-UNet training task. First, we reexamine the data wait time of tested tasks. As shown in Fig. 12a, KSpeed has a considerably smaller data waiting time than others, including Pytorch, Alluxio, and DPP, in all the tested cases. Second, we reexamine the data preprocessing time in tested cases.

As shown in Fig. 12b, KSpeed delivers up to 70× data preprocessing performance improvement over others, which undoubtedly contributes significantly to the shorter data waiting time in workers. The unique data preprocessing offloading mechanism employed by KSpeed is the reason why it outperforms DPP. For tensor transformation, GPUs can be magnitude faster than CPUs. Lastly, we reex-

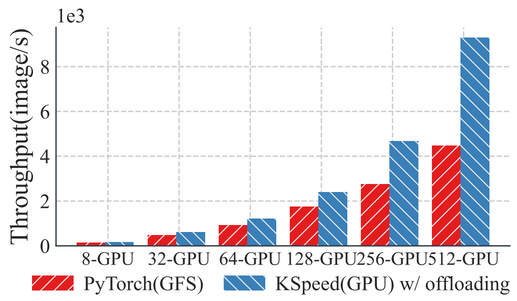


Fig. 13: Scalability of in-house applications.

amine the variation in data waiting time among different workers.

Fig. 12c shows the data waiting time of the 32 workers in the same training step. The X-axis shows the sequences of GPU workers while the Y-axis gives the data waiting time in that iteration. From the figure, we can find that the workers in KSpeed present minimal variations, and the values are much lower than other methods. In contrast, the system built with DPP experiences much longer waiting time and severe performance variations. Alluxio presents the highest data waiting time for most workers, while PyTorch (GFS) experiences the worst-case scenario for this training scenario, where the completion time of this training step is significantly prolonged compared to KSpeed.

E. Scalability

Finally, we evaluate the scalability of KSpeed by performing end-to-end training of in-house applications in a 512-GPU cluster. As the result shown in Fig. 13, KSpeed can achieve near linear scalability. The average number of images processed per second for the 8-GPU and 512-GPU settings are 154 and 9300, respectively, resulting in a scalability efficiency of 94.2%. Although there is minimal performance difference between PyTorch and KSpeed at smaller scales, the difference becomes more significant in large-scale systems due to the I/O contention of PFS. PyTorch's scalability drops sharply beyond 256 GPUs. In addition, KSpeed experiences fewer contentions owing to its disaggregated caching and preprocessing designs proposed in Section III-A.

V. Related Work

Data caching mechanisms [30], [29], [27], [48], [11], [43], [19], [34] have been proposed to accelerate the data loading performance from remote storage. Alluxio [30], [29] is an open-source data orchestration system for data analytics and AI workloads. Compared to native PyTorch I/O, it can provide a distributed in-memory file cache layer between deep learning frameworks and backend storage. DeepIO [48] is an entropy-aware I/O framework for TensorFlow. It improves the training efficiency by coordinating TensorFlow's memory, communication, and I/O resources. PacMan [11] explores data reconciliation-based caching schemes across different workers to extract the maximum benefit of data analytics performance. Similarly, intelligent scheduling of big data jobs is explored in Quartet [16] to maximize cross-job sharing of cached data. Clairvoyant [19] proposes a caching scheme for training based on the probability that some of the data would be reused by other workers in the same node. Quiver [27] is designed as a shared distributed cache for multi-tenant deep learning training for reusing input training data (with isolation) through content-addressing. In Quiver, the cache key is determined by the hash of the cached content.

Recently, preprocessing offloading has been proposed to mitigate the performance variations caused in trainer nodes. Zhao et al. present Meta's end-to-end data storage and ingestion pipeline called DPP [47], which offloads the data preprocessing tasks to separate nodes, between the backend storage nodes and the trainer nodes, to mitigate the loading on trainers and improve the system throughput. In addition to DPP [47], there are several other representative studies on preprocessing. Ibrahim et al. [22] develop domain-specific data encoding and decoding plugins to reduce the required data movement and accelerate data prepossessing execution. Graur et al. [20] further investigate the resource allocation mechanism including CPU time and expensive cache storage among different training tasks, and propose Cachew, a data pipelining service that dynamically scales and schedules distributed processing resources to match the speed of multiple training nodes. MinIO [34] can be viewed as a software cache that accelerates deep learning training, which mitigates data stalls based on the NVidia DALI library [35].

Since PyTorch, Alluxio, and DPP are the most mature designs originating from industry, we mainly compare their performance with that of KSpeed in this paper. The key difference between KSpeed and existing studies is that the I/O nodes can preprocess the input data and directly write the preprocessed data to the training nodes' GPU memory. In contrast, existing works have to perform preprocessing by local CPUs on the training nodes for certain models without copying the data to the host buffers in compute nodes. Moreover, KSpeed can fully utilize the multi-rail RDMA network.

VI. Conclusion and Discussion

This paper presents KSpeed, a novel data provisioning framework for beating I/O bottlenecks in deep learning training. In contrast to traditional host memory disaggregation approaches, KSpeed takes one step further by organizing not only host memory but also CPU resources into a disaggregated memory/CPU pool for training data caching and preprocessing services. We have integrated KSpeed with PyTorch, which realizes a high-performance deep learning training framework for many training applications on our production AI Cloud. In our future work, we will deploy KSpeed in the virtualized training environment and adapt proactive data provisioning to GPU virtualization.

Recently, LLMs have gained significant popularity, particularly in text-based applications. As they have evolved, they are increasingly becoming multimodal, requiring the integration of not only text but also videos and images. The shift toward multimodality also introduces data loading bottlenecks. In these scenarios, KSpeed also can provide benefits by optimizing data provisioning for the various modalities involved, ensuring efficient training performance.

Acknowledgment

We thank the ICNP anonymous reviewers and our shepherd, Prof. Hong Xu, for their valuable feedback.

References

[1] "Nvidia teases its most powerful GPU ever."

[2] "Alluxio," https://www.alluxio.io/, 2022.

[3] "GlusterFS," https://github.com/gluster/glusterfs, 2022.

[4] "LITS 2017 Dataset," https://paperswithcode.com/dataset/lits17, 2022.

[5] "WAV File Format," https://docs.fileformat.com/audio/wav/#wav-file-format, 2022.

[6] "Convolutional Network for Image Classification in PyTorch," https://github.com/NVIDIA/DeepLearningExamples/tree/master/PyTorch/Classification/ConvNets, 2023.

[7] "GPT-4," https://openai.com/research/gpt-4, 2023.

[8] "NVIDIA DALI User Guide," https://docs.nvidia.com/deeplearning/dali/user-guide/docs/supported_ops.html, 2023.

[9] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard *et al.*, "TensorFlow: A System for Large-Scale Machine Learning," in *12th USENIX symposium on operating systems design and implementation (OSDI 16)*, 2016, pp. 265–283.

[10] S. Abu-El-Haija, N. Kothari, J. Lee, P. Natsev, G. Toderici, B. Varadarajan, and S. Vijayanarasimhan, "YouTube-8M: A Large-Scale Video Classification Benchmark," 2016. [Online]. Available: https://arxiv.org/abs/1609.08675

[11] G. Ananthanarayanan, A. Ghodsi, A. Warfield, D. Borthakur, S. Kandula, S. Shenker, and I. Stoica, "Pacman: Coordinated Memory Caching for Parallel Jobs," in *9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*, 2012, pp. 267–280.

[12] J. Bao, D. Chen, F. Wen, H. Li, and G. Hua, "CVAE-GAN: Fine-Grained Image Generation Through Asymmetric Training," in *Proceedings of the IEEE International Conference on Computer Vision (ICCV)*, Oct 2017.

[13] Y. Bao, Y. Peng, Y. Chen, and C. Wu, "Preemptive All-Reduce Scheduling for Expediting Distributed DNN Training," in *IEEE INFOCOM 2020-IEEE Conference on Computer Communications*. IEEE, 2020, pp. 626–635.

[14] J. Choquette and W. Gandhi, "NVIDIA A100 GPU: Performance & Innovation for GPU Computing," in *2020 IEEE Hot Chips 32 Symposium (HCS)*. IEEE Computer Society, 2020, pp. 1–43.

[15] Ö. Çiçek, A. Abdulkadir, S. S. Lienkamp, T. Brox, and O. Ronneberger, "3D U-Net: Learning Dense Volumetric Segmentation from Sparse Annotation," in *International Conference on Medical Image Computing and Computer-Assisted Intervention*. Springer, 2016, pp. 424–432.

[16] F. Deslauriers, P. McCormick, G. Amvrosiadis, A. Goel, and A. D. Brown, "Quartet: Harmonizing Task Scheduling and Caching for Cluster Computing," in *8th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 16)*, 2016.

[17] J. Devlin, M.-W. Chang, K. Lee, and K. N. Toutanova, "BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding," 2018. [Online]. Available: https://arxiv.org/abs/1810.04805

[18] J. Dong, Z. Cao, T. Zhang, J. Ye, S. Wang, F. Feng, L. Zhao, X. Liu, L. Song, L. Peng, Y. Guo, X. Jiang, L. Tang, Y. Du, Y. Zhang, P. Pan, and Y. Xie, "EFLOPS: Algorithm and System Co-Design for a High Performance Distributed Training Platform," in *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2020, pp. 610–622.

[19] 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*, ser. SC '21. New York, NY, USA: Association for Computing Machinery, 2021. [Online]. Available: https://doi.org/10.1145/3458817.3476181

[20] 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)*, Carlsbad, CA, Jul. 2022, pp. 689–706. [Online]. Available: https://www.usenix.org/conference/atc22/presentation/graur

[21] 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 (CVPR)*, June 2016.

[22] K. Z. Ibrahim and L. Oliker, "Preprocessing Pipeline Optimization for Scientific Deep Learning Workloads," in *2022 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2022, pp. 1118–1128.

[23] Z. Jia, M. Zaharia, and A. Aiken, "Beyond Data and Model Parallelism for Deep Neural Networks," *Proceedings of Machine Learning and Systems*, vol. 1, pp. 1–13, 2019.

[24] Y. Jiang, Y. Zhu, C. Lan, B. Yi, Y. Cui, and C. Guo, "A Unified Architecture for Accelerating Distributed DNN Training in Heterogeneous GPU/CPU Clusters," in *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, Nov. 2020, pp. 463–479. [Online]. Available: https://www.usenix.org/conference/osdi20/presentation/jiang

[25] Kaiming, 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.

[26] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet Classification With Deep Convolutional Neural Networks," *Advances in neural information processing systems*, vol. 25, pp. 1097–1105, 2012.

[27] 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.

[28] A. Kuznetsova, H. Rom, N. Alldrin, J. Uijlings, I. Krasin, J. Pont-Tuset, S. Kamali, S. Popov, M. Malloci, A. Kolesnikov, T. Duerig, and V. Ferrari, "The Open Images Dataset V4: Unified image classification, object detection, and visual relationship detection at scale," *IJCV*, 2020.

[29] H. Li, "Alluxio: A virtual distributed file system," Ph.D. dissertation, EECS Department, University of California, Berkeley, May 2018. [Online]. Available: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2018/EECS-2018-29.html

[30] H. Li, A. Ghodsi, M. Zaharia, S. Shenker, and I. Stoica, "Tachyon: Reliable, Memory Speed Storage for Cluster Computing Frameworks," in *Proceedings of the ACM Symposium on Cloud Computing*, 2014, pp. 1–15.

[31] M. Li, D. G. Andersen, J. W. Park, A. J. Smola, A. Ahmed, V. Josifovski, J. Long, E. J. Shekita, and B.-Y. Su, "Scaling Distributed Machine Learning with the Parameter Server," in *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, 2014, pp. 583–598.

[32] M. Li, T. Zhang, Y. Chen, and A. J. Smola, "Efficient Mini-Batch Training for Stochastic Optimization," in *Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining*, 2014, pp. 661–670.

[33] A. Mathuriya, D. Bard, P. Mendygral, L. Meadows, J. Arnemann, L. Shao, S. He, T. Kärnä, D. Moise, S. J. Pennycook, K. Maschhoff, J. Sewall, N. Kumar, S. Ho, M. F. Ringenburg, Prabhat, and V. Lee, "CosmoFlow: Using Deep Learning to Learn the Universe at Scale," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis*, ser. SC '18. IEEE Press, 2018. [Online]. Available: https://doi.org/10.1109/SC.2018.00068

[34] 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. [Online]. Available: http://www.vldb.org/pvldb/vol14/p771-mohan.pdf

[35] NVIDIA, "DALI," https://developer.nvidia.com/dali, 2022.

[36] ——, "Inside Volta: The World's Most Advanced Data Center GPU," https://developer.nvidia.com/blog/inside-volta/, 2022.

[37] ——, "The NVIDIA Collective Communication Library (NCCL)," https://developer.nvidia.com/nccl, 2022.

[38] OpenAI, "GPT-3 Powers the Next Generation of Apps," https://openai.com/blog/gpt-3-apps/, 2022.

[39] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga *et al.*, "Pytorch: An Imperative Style, High-Performance Deep Learning Library," *Advances in neural information processing systems*, vol. 32, 2019.

[40] S. Rajbhandari, O. Ruwase, J. Rasley, S. Smith, and Y. He, "ZeRO-Infinity: Breaking the GPU Memory Wall for Extreme Scale Deep Learning," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '21. New York, NY, USA: Association for Computing Machinery, 2021. [Online]. Available: https://doi.org/10.1145/3458817.3476205

[41] A. Sergeev and M. D. Balso, "Horovod: Fast and Easy Distributed Deep Learning in TensorFlow," *ArXiv*, vol. abs/1802.05799, 2018.

[42] M. Shoeybi, M. Patwary, R. Puri, P. LeGresley, J. Casper, and B. Catanzaro, "Megatron-LM: Training Multi-Billion Parameter Language Models Using Model Parallelism," *arXiv preprint arXiv:1909.08053*, 2019.

[43] P. Stuedi, A. Trivedi, J. Pfefferle, R. Stoica, B. Metzler, N. Ioannou, and I. Koltsidas, "Crail: A High-Performance I/O Architecture for Distributed Data Processing," *IEEE Data Eng. Bull.*, vol. 40, no. 1, pp. 38–49, 2017.

[44] V. Sze, Y.-H. Chen, T.-J. Yang, and J. S. Emer, "Efficient Processing of Deep Neural Networks: A Tutorial and Survey," *Proceedings of the IEEE*, vol. 105, no. 12, pp. 2295–2329, 2017.

[45] TensorFlow, "Pushing the limits of GPU performance with XLA," https://medium.com/tensorflow/pushing-the-limits-of-gpu-performance-with-xla-53559db8e473, 2022.

[46] M. Yan, H. Xu, C. Li, J. Tian, B. Bi, W. Wang, W. Chen, X. Xu, F. Wang, Z. Cao *et al.*, "Achieving Human Parity on Visual Question Answering," *arXiv preprint arXiv:2111.08896*, 2021.

[47] M. Zhao, N. Agarwal, A. Basant, B. Gedik, S. Pan, M. Ozdal, R. Komuravelli, J. Pan, T. Bao, H. Lu, S. Narayanan, J. Langman, K. Wilfong, H. Rastogi, C.-J. Wu, C. Kozyrakis, and P. Pol, "Understanding Data Storage and

Ingestion for Large-Scale Deep Recommendation Model Training: Industrial Product," in *Proceedings of the 49th Annual International Symposium on Computer Architecture*, ser. ISCA '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 1042–1057. [Online]. Available: https://doi.org/10.1145/3470496.3533044

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