GoldMiner: Elastic Scaling of Training Data Pre-Processing Pipelines for Deep Learning

HANYU ZHAO*, Alibaba Group, China and Peking University, China
ZHI YANG†, Peking University, China
YU CHENG*, Peking University, China and Alibaba Group, China
CHAO TIAN*, Peking University, China and Alibaba Group, China
SHIRU REN, Alibaba Group, China
WENCONG XIAO, Alibaba Group, China
MAN YUAN, Alibaba Group, China
LANGSHI CHEN, Alibaba Group, China
KAIBO LIU*, Peking University, China and Alibaba Group, China
YANG ZHANG, Alibaba Group, China
YONG LI, Alibaba Group, China
WEI LIN, Alibaba Group, China

Training data pre-processing pipelines are essential to deep learning (DL). As the performance of model training keeps increasing with both hardware advancements (*e.g.*, faster GPUs) and various software optimizations, the data pre-processing on CPUs is becoming more resource-intensive and a severe bottleneck of the pipeline. This problem is even worse in the cloud, where training jobs exhibit diverse CPU-GPU demands that usually result in mismatches with fixed hardware configurations and resource fragmentation, degrading both training performance and cluster utilization.

We introduce GoldMiner, an input data processing service for stateless operations used in pre-processing data for DL model training. GoldMiner decouples data pre-processing from model training into a new role called the *data worker*. Data workers facilitate scaling of data pre-processing to anywhere in a cluster, effectively pooling the resources across the cluster to satisfy the diverse requirements of training jobs. GoldMiner achieves this decoupling in a fully *automatic* and *elastic* manner. The key insight is that data pre-processing is inherently *stateless*, thus can be executed independently and elastically. This insight guides GoldMiner to automatically extract stateless computation out of a monolithic training program, efficiently disaggregate it across data workers, and elastically scale data workers to tune the resource allocations across jobs to optimize cluster efficiency. We have applied GoldMiner to industrial workloads, and our evaluation shows that GoldMiner can transform unmodified training programs to use data workers, accelerating individual training jobs by up to 12.1×. GoldMiner also improves average job completion time and aggregate GPU utilization by up to 2.5× and 2.1× in a 64-GPU cluster, respectively, by scheduling data workers with elasticity.

*Work done while Hanyu Zhao, Yu Cheng, Chao Tian, and Kaibo Liu are interns at Alibaba Group.
†Zhi Yang is the corresponding author (yangzhi@pku.edu.cn).

Authors' addresses: Hanyu Zhao, Alibaba Group, China and Peking University, China; Zhi Yang, Peking University, China; Yu Cheng, Peking University, China and Alibaba Group, China; Chao Tian, Peking University, China and Alibaba Group, China; Shiru Ren, Alibaba Group, China; Wencong Xiao, Alibaba Group, China; Man Yuan, Alibaba Group, China; Langshi Chen, Alibaba Group, China; Kaibo Liu, Peking University, China and Alibaba Group, China; Yang Zhang, Alibaba Group, China; Yong Li, Alibaba Group, China; Wei Lin, Alibaba Group, China.

2836-6573/2023/6-ART193 $15.00
https://doi.org/10.1145/3589773

CCS Concepts: • **Computer systems organization** → **Cloud computing**; • **Computing methodologies** → **Machine learning**.

Additional Key Words and Phrases: deep learning, data pre-processing, disaggregation, scheduling

ACM Reference Format:
Hanyu Zhao, Zhi Yang, Yu Cheng, Chao Tian, Shiru Ren, Wencong Xiao, Man Yuan, Langshi Chen, Kaibo Liu, Yang Zhang, Yong Li, and Wei Lin. 2023. GoldMiner: Elastic Scaling of Training Data Pre-Processing Pipelines for Deep Learning. *Proc. ACM Manag. Data* 1, 2, Article 193 (June 2023), 25 pages. https://doi.org/10.1145/3589773

1 INTRODUCTION

Deep learning (DL) training is an increasingly important workload in the cloud, owing to its vital role in a wide range of products, *e.g.*, social networks, E-commerce platforms. Companies are therefore building or purchasing GPU clouds to acquire higher compute power to accelerate the training, where the efficiency of utilizing the GPU resources becomes the key to maximizing the revenue and the return on investments. Many systems have been proposed to improve the efficiency from various aspects, *e.g.*, training acceleration and cluster scheduling [13, 27, 28, 32, 33, 40, 44, 49, 52, 54, 60, 61]. Although existing works mainly center around training computation and GPU accelerators, which indeed are critical factors, DL training is still essentially a multi-phase and multi-resource workload: it requires not only more advanced model architectures trained on GPUs, but also *data pre-processing pipelines* which produce high-quality input data for the models and typically run on CPUs. Data pre-processing pipelines have been proven a necessary step towards better models for a variety of workloads [15, 43].

As the training performance keeps increasing due to more powerful GPUs and various optimizations, the data pre-processing pipelines are becoming more resource-consuming and throttling the performance [23, 36]. We observe that a training job can demand CPUs far beyond the configured on cloud GPU VMs (§2), *e.g.*, 64 CPU cores are required per GPU worker, while only 8 CPU cores are provided. As a result, how to manage DL data prep-processing pipelines in the cloud has become a significant problem in data systems for machine learning and artificial intelligence and gained increasing attention in the data management community [24, 36, 38].

A major obstacle to improving job performance and cluster utilization is the tightly coupled data pre-processing (with APIs like `tf.data` [38]) and model training in user's monolithic programs. These programs will run in containers with GPUs and CPUs allocated from certain machines of a cluster. At the job level, this coupling prohibits jobs from scaling out the data pre-processing pipelines when the local CPUs are insufficient and finally compromises training performance. At the cluster level, the heterogeneous requirements of CPUs and GPUs across jobs can induce resource fragmentation, *i.e.*, unbalanced CPU-GPU usage of the machines, which further exacerbates the problem. Such fragmented resources will be more likely to be wasted because insufficiency of either type of resource will prevent the other type from being allocated, degrading cluster utilization. Our observation from a 5,600-GPU cluster at Alibaba shows that 40% of the total job queuing delay is caused by insufficient CPUs, wasting expensive GPUs.

Recent works explore the direction of separating training and data pre-processing into different processes to address this problem [9, 18, 59]. However, it is non-trivial to achieve this separation transparently and efficiently. First, we would like to make this separation non-intrusive to user code; however, just relying on existing dataset APIs like `tf.data` for this separation can possibly miss some of such computations, limiting its full potential. Moreover, even with the separation, it is still unclear how to exploit it during scheduling to maximize cluster efficiency.

GoldMiner is a fully-managed service that automatically disaggregates and elastically schedules data pre-processing to improve training performance and cluster efficiency. GoldMiner introduces a new role called the *data worker* that is designed to be independent from the training, thus facilitating horizontal and elastic scaling of data pre-processing to anywhere in a cluster. This scaling not only can accelerate data pre-processing, but also effectively pools the CPU resources throughout a cluster, improving resource utilization.

GoldMiner can automatically extract the computations that can be offloaded to data workers from unmodified training programs, even for those not encapsulated in the dataset APIs. The key insight here is that the essential distinction of data pre-processing is its statelessness — that is, it does not depend on states like model parameters that need to be updated repeatedly, and hence can be executed asynchronously. Leveraging the dataflow graph abstraction for DL training, GoldMiner adopts a graph partitioning strategy to identify the stateless computations for data workers. It then rewrites the connecting structures across subgraphs, with accompanying runtime mechanisms, to enable disaggregated execution. GoldMiner also supports an advanced version of partitioning to incorporate specially optimized data pre-processing operations on GPUs, while maintaining the benefit of CPU pooling.

Data workers also exhibit great elasticity, thanks to the statelessness, which brings opportunities for schedulers to improve cluster efficiency. To reap such benefits, GoldMiner further provides a new scheduler that continuously tunes resource allocations for data workers across jobs leveraging the elasticity. By monitoring the queues for transferring data, our scheduler can evaluate whether and how much the jobs benefit from data workers and allocate data workers across jobs to satisfy their respective needs while optimizing cluster-wide scheduling objectives.

We implement GoldMiner on top of TensorFlow and Kubernetes and have applied it to real production training jobs. We show, for a variety of micro-benchmark workloads, that GoldMiner can transform unmodified training programs to use data workers with high scalability. For individual jobs, GoldMiner achieves speedup by up to 12× and reduces monetary cost by up to 88% by scaling out data workers. For cluster-wide efficiency, GoldMiner improves average job completion time (JCT) by up to 2.5× and aggregate GPU utilization by up to 2.1× in a 64-GPU cluster.

The contributions of this paper are summarized as follows.

- We devise graph partitioning and disaggregation mechanisms to enable decoupled execution of data and training workers automatically and efficiently.
- We develop a cluster scheduler that exploits the elasticity of data workers to tune resource allocations to meet individual jobs' needs and improve cluster efficiency.
- We deploy GoldMiner to production training jobs and show the benefits to training performance and cluster efficiency with extensive evaluations.

Moving forward. With the features and advantages above, GoldMiner can be used for optimizing real-world industrial applications. For example, we are conducting an initial deployment of GoldMiner to optimize a production model from one of the most important customers of us. The model is used in the ranking stage of click-through rate (CTR) prediction, and serves daily recommendation requests for multiple major business lines. Our evaluation shows that GoldMiner accelerates the training by 1.43× and reduces the cost by 13% for our customer. We believe that more significant savings can be achieved when a wide variety of industrial models are deployed, as our evaluation with benchmark workloads showing up to 12× speedup. We are also deploying GoldMiner to internal GPU clusters and training jobs at Alibaba.

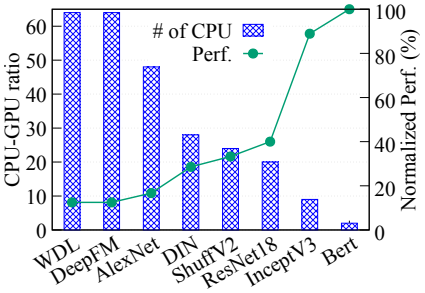

Fig. 1. CPU-GPU ratios and slowdowns using 8 cores of DNN models.

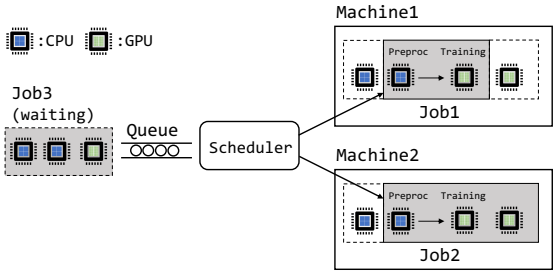

Fig. 2. Multi-resource fragmentation: an example.

2 MOTIVATION

In this section, we first briefly overview the DL training data pipelines as background. We then analyze how CPU allocation for data pipelines affect job performance and cluster efficiency to motivate the design of GoldMiner. The observations are collected from a *5,600-GPU production cluster* at Alibaba supporting daily training jobs. Finally, we summarize the limitations of existing solutions and discuss the opportunities to exploit.

DL training and data pre-processing. We focus on the DL computation workload which becomes increasingly important. In this kind of computation, input data is commonly pre-processed on CPUs and then fed into GPUs for gradient computations. Specifically, DL training jobs are usually expressed as dataflow graphs (DFGs) [8], where each node indicates a computational operator executed on CPU or GPU. The training workflow often involves millions of iterations, called *mini-batches*. Each mini-batch loads data from storage, feeds the data through the DFG, and applies gradients to refresh the model state. Although expressed as a DFG, the end-to-end training pipeline is composed of two major steps. The data pre-processing step takes raw data as inputs and produces tensors for model training. The data pre-processing is highly customized according to the workload requirements. Examples include data decompressing, image cropping and flipping, feature format transformation, *etc.*. These operations are CPU-intensive, and must make efficient use of available CPU resources to maximize input pipeline throughput. The model training step utilizes the processed data for model parameter updates, using gradients derived from the *forward-backward* passes, which are often conducted on GPUs. The common practice today processes the DL training pipeline in a worker process monolithically, with a bunch of CPU and GPU resources allocated.

Monolithic pipelines limit job performance. To understand the CPU requirement for GPU training, we measure the best-fit CPU-GPU ratios of representative workloads, by increasing the CPU allocations until the jobs reach the maximal training throughput* or the machine limit. Surprisingly, many workloads are in a high CPU demand, up to 64 cores per GPU (Figure 1). However, the hardware (*i.e.*, cloud GPU instances) are usually not equipped with sufficient cores. For example, `p3` and `ecs.gn6v`, as typical V100 GPU instance series on AWS and Alibaba Cloud, offer 8 cores per V100 GPU. Therefore, only Bert can fit into the cloud GPU instances. The other workloads suffer suboptimal performance, because of lacking sufficient CPUs for data pre-processing. Our measurement on an 8-CPU 1-V100 GPU instance shows that the performance of typical DL workloads can be slower by more than 90% compared to that of executing using sufficient CPUs, which not only

*WDL, DeepFM, and DIN are tested using GoldMiner, because their original TensorFlow versions show poor scalability with increasing numbers of CPUs. The other models are tested using TensorFlow.

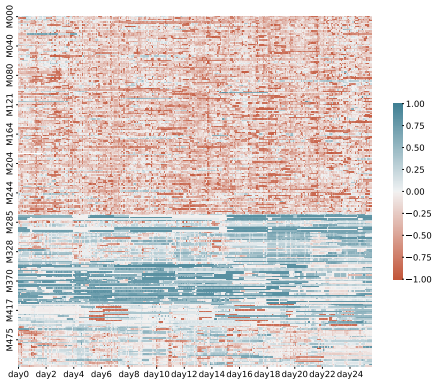

Fig. 3. Heatmap of multi-resource fragmentation.

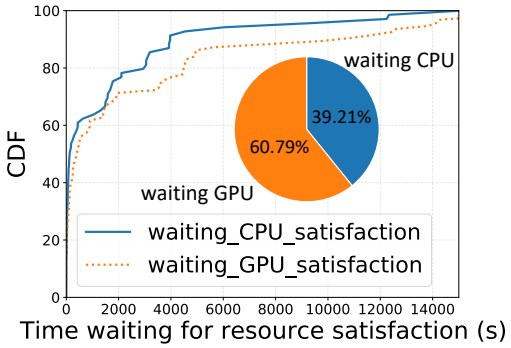

Fig. 4. Queuing delays for CPU and GPU satisfaction.

hurts job performance but also wastes expensive GPU resources. What's more, the new generation H100 GPU is about 16× in FP16 FLOPS compared to V100 [4]. The trend of growing performance gap between CPU and accelerators further shifts the bottleneck of DL workloads toward the data pre-processing on CPUs.

Fragmentation lowers cluster efficiency. To adapt to the heterogeneous CPU requirements while ensuring job performance, users submit GPU training jobs with different numbers of CPUs in shared GPU clusters. In this way, DL jobs come with different CPU-GPU ratios, which might introduce *multi-resource fragmentation* during job scheduling. Specifically, the term "multi-resource fragmentation" refers to the case where a machine cannot accommodate an incoming job, despite enough resources for certain types, due to insufficiency of other types. As illustrated in Figure 2, the cluster has scheduled job1 (1-GPU/1-CPU) and job2 (1-CPU/2-GPU) on a cluster with two machines (each contains 2 GPUs and 2 CPUs). Then job 3 (1-GPU/2-CPUs) has to wait despite enough resources over the cluster, as the data processing and training computations of the job need to execute on the same machine in DL frameworks.

We analyze the multi-resource fragmentation issue in a 26-day production trace of the 5,600-GPU cluster. We quantify the fragmentation for each machine as the difference between the proportion of free CPUs and the proportion of free GPUs (*i.e.*, $\frac{FreeCPU}{TotalCPU} - \frac{FreeGPU}{TotalGPU}$). The physical meaning of this indicator can be interpreted as the remaining CPU or GPU resources that cannot be allocated to other jobs, assuming future jobs have the same resource shape as the machine's. For example, if a machine has 25% of the CPUs and 50% of the GPUs left, then the fragmentation value 25% indicates that there would be 25% of the GPUs that remain unallocated when the CPUs are used up.

Figure 3 shows a heatmap at the machine level to demonstrate the multi-resource fragmentation. The color of each square implies the degree of multi-resource fragmentation on the corresponding machine. Deeper color indicates more serious fragmentation. Red color indicates GPU fragmentation, which means GPUs might remain unallocated when CPUs are used up. Blue color indicates CPU fragmentation. Resource fragmentation is commonly observed in the shared GPU cluster, for both CPU and GPU. Specifically, *31% of the machines have allocated all CPUs while still holding more than 25% of GPU resources (2 GPUs) unallocated.* On the other hand, 18% of the machines have allocated all GPUs while still holding more than 25% of CPU resources (24 cores) unallocated.

Multi-resource fragmentation can lead to significant queuing delays and resource waste. A machine with insufficient resource of any type (*e.g.*, CPU) will have to reject a container, waste

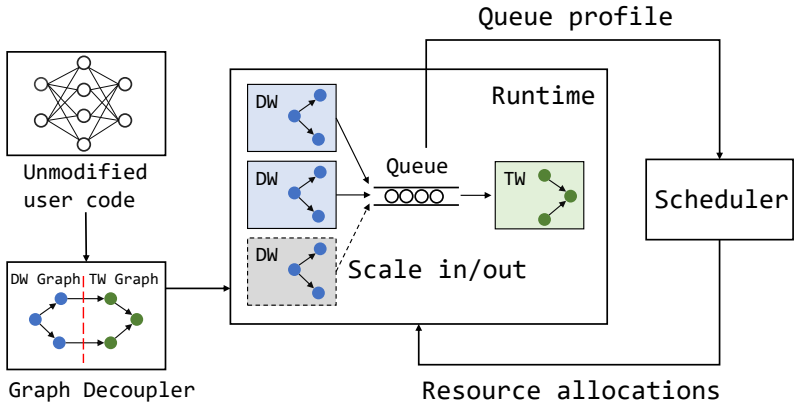

Fig. 5. Overall workflow of GoldMiner.

the other type (GPU), and prevent the whole job from being scheduled. Figure 4 illustrates the CDF of queuing delays in our production cluster caused by CPUs and GPUs, respectively (the case where both are not enough is included in GPU). Surprisingly, the queuing delay for CPUs accounts a significant fraction (39.2%) of the total, even though GPUs have been generally recognized as the primary resource for DL jobs. Therefore, CPU is a main factor that limits the utilization of GPU and becomes a major bottleneck.

Limitations of existing work. Recent researches [7, 53] propose to allocate CPUs based on profiling and adopt multi-resource packing algorithms to reduce fragmentation. However, the aforementioned fragmentation remains an inevitable problem in an online environment as a result of the diverse job requirements. Meanwhile, scheduling a DL training job in a shared GPU cluster should also consider factors like gang scheduling, fairness, interference, locality, priorities [12, 25, 52, 54, 57], making the decision space limited for considering multi-resource job packing. GoldMiner explores a different direction of disaggregating CPU computation from DL GPU training automatically, efficiently, and elastically.

3 DESIGN

This section presents how we architect and optimize our end-to-end input data pipelines in industry-scale environment at Alibaba. As illustrated in Figure 5, GoldMiner consists of three components — a graph decoupler and a runtime for each job, and a scheduler for cluster-wide resource allocation. We first provide an overview of GoldMiner, and then detail each of its components.

3.1 Overview

System goals. Although there is an extensive literature on parallel query processing and on dataflow graphs (*e.g.*, Spark [55] and Naiad [37]), they cannot be seamlessly integrated with DL training frameworks (e.g., TensorFlow [8] or PyTorch [41]), forcing DL pipelines to compose multiple systems would increase complexity and lead to unnecessary data movement and duplication. Therefore, a new input data processing service on the top of DL frameworks is needed, to enable a single system to address the entire DL computation flow.

GoldMiner builds upon the central idea of *decoupling data pre-processing from training at the DL framework*, executing and scheduling them in separate *data workers* (DWs) and *training workers* (TWs). This decoupling is beneficial at both job and cluster levels. First, GoldMiner accelerates individual jobs by allowing data workers to acquire extra resources than originally allocated for

the training workers, which is impossible when data pre-processing and training are in the same process. This works even for some operations that are typically treated as part of training but in fact can be executed in parallel with the training.

Second, GoldMiner improves cluster efficiency by leveraging the scheduling flexibility of data workers to tune the resource allocations via dynamic scaling. Moreover, data workers separate a major part of CPU usage from the training, which effectively pools the CPU resources across a cluster and significantly reduces the chance of job queuing and resource waste due to resource fragmentation. This scheduling flexibility and increased utilization are beneficial to *all* jobs in a cluster, even to those that cannot be accelerated directly (whose data pre-processing is fast enough).

Workflow. GoldMiner is an input data processing system for machine learning and artificial intelligence, rather for generic data processing. It achieves the above system goals in a *user-transparent and automated* manner by exploiting domain-specific intelligence that simplifies data pre-processing management in DL.

Given an unmodified training program, the graph decoupler automatically produces new computation graphs for data and training workers out of the original one. The key characteristic that GoldMiner exploits is that data pre-processing operators in the DL training DFG, are *stateless*, *i.e.*, independent on the per mini-batch stateful model update. GoldMiner leverages such opportunities to disaggregate stateless data pipelines from DL training completely. It identifies the maximum boundary of stateless computation, and performs a proper cut on the DFG for the efficient overlapping between pre-processing and training. In this way, we can distribute data processing across remote CPU hosts. Such resource allocation flexibility naturally enables scaling CPU independently from GPUs, which significantly reduces resource fragmentation.

The scheduler leverages the stateless nature of DL input data processing to adjust the resource allocation during runtime. Determining the right resource allocation for DL input data processing is non-trivial as each model and input data pipeline combination has unique requirement. We harnesses the iterative characteristic of DL training and leverage domain-specific metrics collected at DWs and TWs to minimize training time and cost. During execution, the runtime continuously profiles jobs to understand their behaviors when using data workers, *e.g.*, whether and how much they benefit from it. This information guides the scheduler to explore a best-fit resource configuration for each job and optimize cluster-wide objectives, exploiting the scheduling flexibility.

Advancements. Although systems like tf.data service [38] and Meta's Data PrePreprocessing (DPP) Service [58] disaggregate data processing, GoldMiner achieves *effective disaggregation* by leveraging the stateless nature and dataflow graph abstraction for DL data processing. Also, to the best of our knowledge, it is the first work that considers and evaluates *cluster-wide* data pre-processing scheduling for concurrent DL jobs.

3.2 Automatic Graph Partitioning

The critical first step towards the computation decoupling is to partition a monolithic, user-defined dataflow graph (DFG) to produce subgraphs to be executed by data and training workers. GoldMiner first partitions the graph into two subgraphs, and then replaces the crossing edges with structures for data transfer across subgraphs. Figure 6 presents a simple example of the partitioning.

Problem. An intuitive partitioning strategy is to split the original DFG at the boundary of standard dataset APIs like `tf.data` [5], illustrated as *simple partition* in Figure 6. However, such a graph partition sometimes is suboptimal because other CPU-intensive time-consuming operators might be left behind in training workers, such as feature transformations (*i.e.*, StringHash, unique, and feature validation) in recommendation models. Ideally, data workers should be responsible for all data pre-processing to maximize the benefits of parallelization. A natural question here is how to find a *maximum range* of "data pre-processing" from a DFG's point of view. The essential

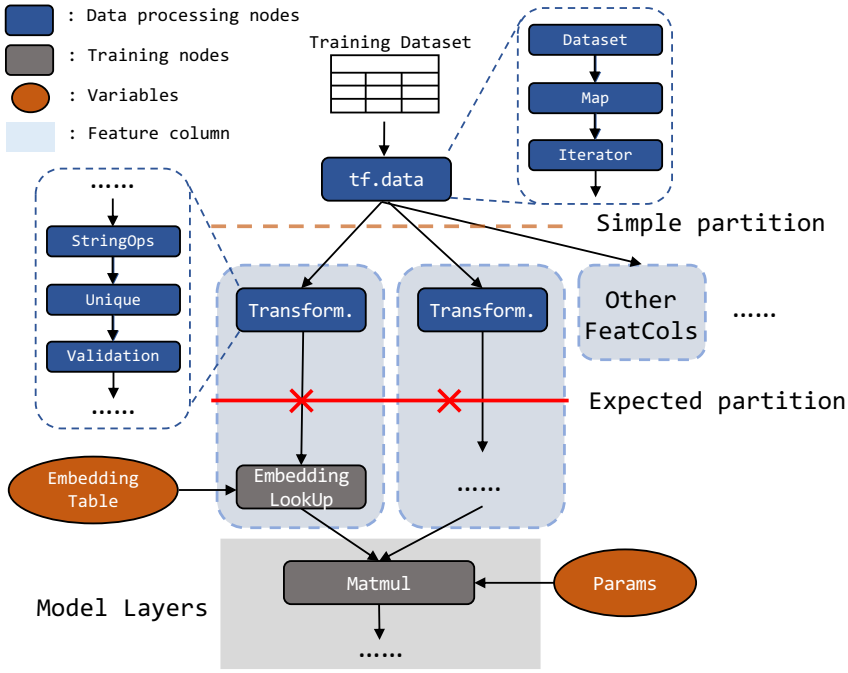

Fig. 6. A simplified dataflow graph of a recommendation model and possible partitioning strategies.

requirement here is making the two stages asynchronous to overlap them and hide the cost of data pre-processing. However, it has been well recognized that it is difficult to enable efficient asynchronous execution of different stages of a model, because of the dependency between the forward and backward passes: in each step, the forward and backward passes are executed in inverse order of layers, prohibiting the layers from proceeding to the next step until the backward pass of all layers completes (unless with intrusive changes to the semantics of model updates [39]).

State-aware partitioning. In this sense, the key distinction of the so-called "data pre-processing" is no involvement of model update. From a dataflow graph perspective, this property can be interpreted as *no dependency on stateful nodes* that will be updated during backward passes (*i.e.*, those for maintaining model parameters). For example, in TensorFlow, the forward and backward pass nodes of a training operation take a `Variable` node as input to access and modify the model parameter this operation uses. As long as a node locates in the upstream side of all the stateful nodes, it can execute asynchronously without waiting for the states to be updated. More formally, we want to partition the computation graph $G = (V, E)$ into $V - S$ and S, where S does not contain any stateful node and its successors. This observation leads us to a simple but effective graph partitioning strategy, as shown in Algorithm 1. It starts from collecting all the stateful nodes managing model parameters, and then finds all their downstream nodes via a breadth-first search. All these nodes belong to the training worker subgraph, and the other upstream nodes can build the data worker subgraph (the "data_stages" in Algorithm 1 will be explained later). In this way, the two subgraphs can be easily pipelined with no dependency.

Recall the recommendation model illustrated in Figure 6, GoldMiner can not only identify the computation in the standard `tf.data` API, but can also further expand the area to the a series of subsequent feature transformation operations encapsulated in a variety of feature columns,

Algorithm 1 Automatic Graph Partitioning

```
1: procedure PARTITION(dfg)
2:     stateful_nodes = dfg.get_stateful_nodes()
3:     train_nodes = bfs_from(stateful_nodes)
4:     data_nodes = dfg.exclude_nodes(data_nodes)
5:     data_stages = []
6:     while !data_nodes.empty() do
7:         stage = bfs_within_device(data_nodes)
8:         data_stages.append(stage)
9:         data_nodes.exclude_nodes(stage)
       return train_nodes, data_stages
```

which could be very time-consuming but can be parallelized across data workers (*i.e.*, *expected partition* in Figure 6). For feature columns with embeddings, GoldMiner marks an end of the data pre-processing when it encounters an `EmbeddingLookUp` operation which takes as input a stateful node, *i.e.*, a `Variable` for the embedding table, thereby avoiding being involved in backward passes. This example also shows that GoldMiner can partition the graph in the middle of a high-level operation, which one can hardly achieve with manual effort. For other feature columns, this search continues until the data is fed into operations like `MatMul` which depend on `Variables` of model parameters.

We do find the automatic graph partitioning especially advantageous for recommendation models, which commonly rely on such transformation operations to increase the quality of the features. These models are crucial to the business of Alibaba and many of our customers, and our evaluation shows that the automatic graph partitioning significantly accelerates the training of such models.

Multi-stage partitioning. In addition to separating the execution, the partitioning strategy also implicitly decouples the usage of CPU and GPU resources as data workers only need CPUs in most cases, which greatly reduces the chance of jobs waiting for multiple types of resources due to fragmentation. In practice, however, we do observe that sometimes data pre-processing also uses GPUs. For example, we have developed GPU versions for some CPU-heavy operations in certain production scenarios (see the evaluation). Using the partitioning strategy above, data workers become yet another role that need both CPUs and GPUs and may suffer from resource fragmentation, reducing the scheduling flexibility.

We support multi-stage partitioning as an advanced strategy to preserve the scheduling flexibility of data workers in such cases. A data worker subgraph using multiple types of devices will be further partitioned to different stages by grouping the operations for each device type (typically a CPU stage and a GPU stage for our production models), as shown in lines 6-9 in Algorithm 1. These stages will be executed by different sets of data workers. Multi-stage partitioning ensures that data workers also do not bundle large amounts of multiple resources, maintaining the advantage of resource decoupling and CPU pooling.

Enabling data transfer. GoldMiner inserts communication operations to enable data transfer between the subgraphs. For horizontal and elastic scaling, each subgraph can have a dynamically changing number of replicated source subgraphs (*i.e.*, data worker subgraphs). To this end, GoldMiner introduces `DWSend/Recv` operations, which are similar to `Send/Recv` in traditional dataflow engines but do not require explicit one-to-one `Send-Recv` mappings. As illustrated in Figure 7, for each crossing edge, there is always a single `DWRecv` regardless of the number of sources; it can connect to arbitrary `DWSends` from upstream workers at runtime. Note that it is possible to use traditional `Send/Recv` to implement the data transfer, but that would need to restart workers to

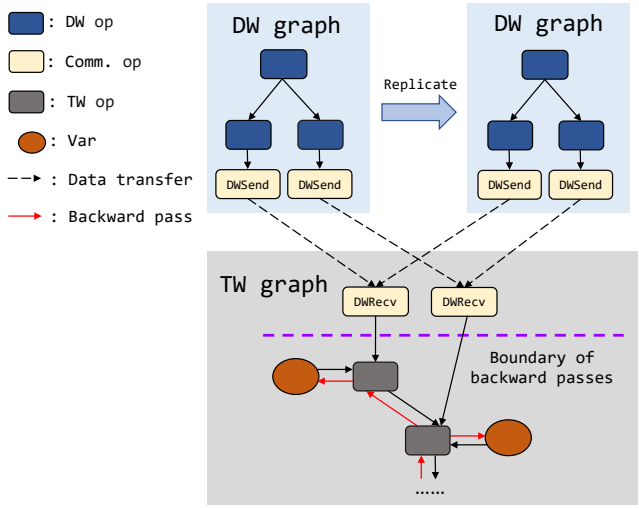

Fig. 7. Graph decoupling in GoldMiner.

rebuild the whole graph whenever scaling data workers. GoldMiner eliminates this overhead and helps the scheduler scale data workers more efficiently.

3.3 Disaggregated Execution

For better scalability and elasticity, data workers operate in a data-parallel model, *i.e.*, each run of a data worker produces a whole mini-batch for a training iteration. We do not split the same mini-batch across data workers (which can be regarded as model parallelism), because the data processing in data workers might contain non-element-wise operators, such as Unique, which requires complicated operator transformation to ensure consistent behavior. To enable data workers to compute non-overlapping mini-batches of data in an elastic manner, GoldMiner adopts a dynamic data dispatching approach where input data indices are continuously dispatched in small subsets to data workers by the runtime. Compared to statically sharding the whole dataset across data workers, this approach keeps data workers purely stateless and avoids data resharding during scaling. The `DWRecv` operation implements a data queue for fetching data from all the data workers in the background to enable asynchronous execution, as shown in Figure 5.

Scaling data workers is easy and transparent to other workers. Every new data worker first registers with the runtime, then the runtime dispatches a replica of the subgraph to it and starts dispatching data indices. It also notifies the training workers to start fetching processed data. Each data worker by default is multiplexed across multiple training workers, so that the number of data workers does not have to be a factor of that of training workers. Scaling in is similar — the runtime stops dispatching new data indices to a data worker and waits for the outstanding data to be fetched, then the data worker can be terminated. With multi-stage data workers, each data worker is assigned to one stage. Raw data indices are only dispatched to the first stage. Data workers of two consecutive stages work as if they were data and training workers as described before.

3.4 Data Worker Scheduling

GoldMiner's scheduler exploits the scheduling flexibility and elasticity of data workers to improve cluster efficiency. The scheduler optimizes resource allocations at both job and cluster levels in

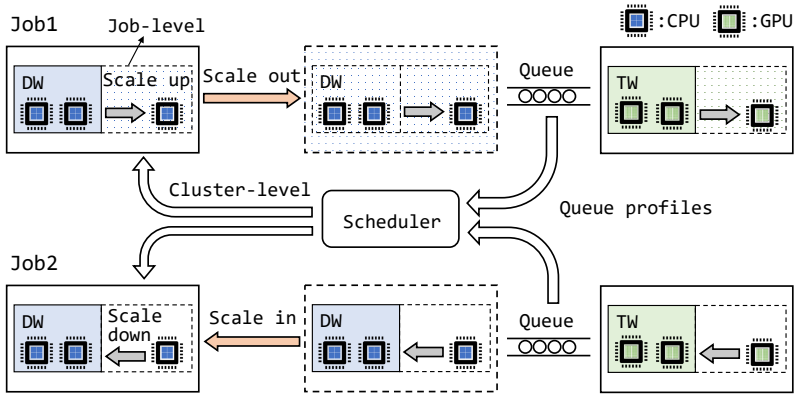

Fig. 8. Data worker scheduling in GoldMiner.

a collaborative manner, as shown in Figure 8. At the job level, it dynamically tunes data worker-related configurations for each job to find a best-fit. At the cluster level, it continuously (re-)allocates data workers across jobs according to the benefits they gain from data workers to optimize global scheduling objectives.

These two levels of scheduling are connected via a unified performance indicator, the state of the data queue. The frequencies of enqueuing and dequeueing of the data queue clearly shows the respective performance of data and training workers, and the performance gap also indicates the potential improvement with more resources for data workers. The scheduler leverages this information to tune resource allocations at both levels, as elaborated next.

3.4.1 Job-level Scheduling. It typically cannot be known a priori how much resource for data pre-processing benefits a job the most, which depends on the ratio of the data pre-processing and training computation and varies greatly across different jobs as shown before. Currently systems rely on users to set the CPU-GPU ratios for their jobs, or simply follow the hardware ratios, which are usually far from optimal without substantial profiling efforts.

Moreover, it is challenging to predict the performance and resource bottleneck of a training job statically during compile time, because of many factors that can hardly be known without profiling: *e.g.*, the compute power and communication bandwidth the operators can utilize (dependent on the batch size), the complex overlapping between computation and communication (dependent on the model structure), the interference among data pre-processing and other CPU training operators. This is also why existing scheduling systems mainly rely on runtime profiling to acquire performance information of training jobs [40, 42, 44].

GoldMiner therefore adopts a profiling-based approach to automatically determine the data worker-related resource configurations for each job based on its performance characteristic. There are three dimensions of configurations, as described below: (i) whether to use data workers; (ii) number of CPUs for each data worker; and (iii) number of CPUs for the training worker.

Whether to use data workers. GoldMiner by default enables data workers for all jobs to maximize the scheduling benefits. However, data workers may introduce extra communication overhead which could sometimes degrade job performance. For example, in computer vision (CV) models, the decoded image data is typically bigger than the raw JPEG files. In general, the transfer of the pre-processed data could be hidden by prefetching, but we do observe that in the worst case the data transfer can visibly degrade performance (*e.g.*, for some simple CV models with very high training throughput). To avoid such cases, GoldMiner first profiles a new job without using data

workers to get a baseline performance. If GoldMiner finds that the job does not get a comparable performance using data workers after tuning (see below) for a threshold of time, the job will fallback to the original mode.

Data worker size. GoldMiner first tunes the number of CPUs for each data worker after a job is launched. Larger data worker sizes can reduce overheads that increase with the number of data workers (*e.g.*, processes, RPCs, locks, memory), but on the other hand some operations might have worse scalability inside a large worker than data parallelism across multiple workers (*e.g.*, some non-parallel feature transformations). GoldMiner searches this CPU number to strike a balance between overheads and scalability. Each job is initially assigned a single data worker container per training worker, with a baseline number of CPUs (the CPU-GPU ratio of the machine by default). Inside the data worker container, GoldMiner gradually increases the CPU number of the process from 1, with the following termination conditions. (1) If the queue has reached the ideal state, *i.e.*, the data worker is as fast as the training worker, then the current CPU number is picked and no more data worker scaling is needed. (2) If the data worker begins to show sublinear speedup, or uses up the current CPUs, then the job chooses the highest CPU number with linear speedup as the maximum CPU number for each data worker (the scheduler is allowed to allocate smaller data workers to use fragmented CPUs).

Training worker size. The training worker also uses CPUs for some computations and fetching data, and the CPU requirement can be different from when a single worker handles both data pre-processing and training. GoldMiner seeks to find a minimum number of CPUs satisfying the requirements for the training worker. When a job finds its queue already in the ideal state, it will try to use fewer CPUs for the training worker, until it finds the minimum number maintaining the current performance. The excessive CPUs are returned to the cluster lazily, *i.e.*, only when the scheduler needs to allocate them to other jobs. On the other hand, if a job sees diminishing returns of more data workers, it is possibly due to insufficient training worker CPUs for fetching the data. To reduce the chance that it is actually caused by other bottlenecks, *e.g.*, network, the job also checks if the CPU utilization of the training worker is higher than a threshold. The job will then request the scheduler for more CPUs for the training worker. In fact, our evaluation shows that for most workloads the initial 8 CPUs are already sufficient for the training worker and no scaling up is needed.

If a job uses multi-stage data workers, the data workers are tuned stage-by-stage, by monitoring the queues between stages. We do not tune GPUs for data workers for simplicity and use 1 GPU for each data worker.

3.4.2 Cluster-level Scheduling. Based on the resource configurations and performance characteristics collected from the job level, GoldMiner employs a scheduler to further allocate data workers across jobs dynamically to optimize certain cluster-wide objectives. The GoldMiner scheduler works in tandem with existing job schedulers for GPU allocation and focuses on data worker allocation. Although it is possible to develop a complex scheduler that jointly considers multiple resources, the separation-of-concerns approach of GoldMiner is simpler and compatible with existing schedulers.

Heuristics. The scheduler continuously allocates data workers to jobs that still have room for improvement, *i.e.*, the enqueuing is slower than dequeuing. The scheduler assumes the speedup of the next scaling step the same as that of the previous step. This optimistic assumption of linear speedup is based on the fact that data workers are data-parallel with no synchronization needed. The scheduler can fix misestimates after it gets real feedbacks. Following a certain global objective, the scheduler allocates a data worker to a job with the largest improvement to a corresponding metric. In this way, the scheduler can address a variety of concerns, and we exemplify a few useful metrics for different objectives here.

Throughput. To maximize cluster throughput, we define the scaling efficiency metric as $\frac{\triangle t}{\triangle c} * g$, where t is the training throughput, c is the resources for data workers, and g is the number of GPUs for training. This metric prefers jobs with higher scaling efficiency of data workers and using more GPUs.

Remaining time. Shortest Remaining Time First (SRTF) is a classic policy to minimize job completion time by preferring jobs with the shortest remaining time (usually multiplied by the resource usage). The principle is to release more resources as soon as possible, thereby reducing job queuing delays.

Using this policy, GoldMiner assumes that each job specifies a number of training steps to calculate the remaining time, which typically can be parsed from the dataflow graph. The scaling metric is defined as the remaining time multiplied by the resource, *i.e.*, $\frac{n*(c+\triangle c)}{t+\triangle t}$, where n is the number of remaining steps. The job with the smallest value will be chosen for allocation.

Fairness. Fairness is also an important factor in scheduling to avoid certain jobs being allocated unfairly much resource and impacting other jobs. From the perspective of data workers, we deal with fairness by equalizing jobs' normalized performance, *i.e.*, relative to the ideal performance with sufficient data workers. The scaling metric is hence defined as $\frac{t+\triangle t}{t_i}$, where t_i is the throughout of training workers derived from the queue state, which representes the ideal throughput of data workers. The job with the smallest value of the metric will be chosen.

Data worker placement. The scheduler also optimizes the placements of data workers to reduce communication overhead. The scheduler prefers allocating local resources of training workers to data workers to reduce network traffic. It also prefers machines running fewer distributed training jobs or other remote data workers to reduce network interference. In particular, the scheduler exploits the elasticity of data workers to continuously optimize their placements by transparently migrating them to better machines.

4 IMPLEMENTATION

We implement GoldMiner on top of TensorFlow [8] and Kubernetes [11]. In particular, our TensorFlow implementation is based on DeepRec [1], an Alibaba-optimized version of TensorFlow v1.15. The implementation includes roughly 4,500 lines of C++ code for the graph decoupler and the runtime. We implement the cluster-level scheduling policies in a custom Kubernetes scheduler with 2,500 lines of Python code.

As usual, users submit their training jobs as containers without any modifications, and these containers now become the training workers in GoldMiner. The graph decoupler rewrites the graph when it starts the training (*i.e.*, when creating `tf.Session`) and then waits for data workers to join. Data worker containers run a generic client process provided by GoldMiner that requests the training worker for the subgraph and executes it when invoked.

Data dispatching and fault tolerance. We implement a data dispatcher as part of the runtime colocated with the training worker. When rewriting the graph, GoldMiner replaces the data reader operator in the data worker subgraph with a custom version that can receive data indices from the dispatcher. When a data worker receives an out-of-range signal from the dispatcher, it will forward the signal to all the training workers requesting for data. A training worker throws an out-of-range exception, as in existing dataset APIs, after it receives the signal from every connected data worker.

GoldMiner provides different levels of fault tolerance depending on the data format. For data where the indices can be deterministically mapped to mini-batches, *e.g.*, small files and table rows, GoldMiner provides the exactly-once guarantee (in each epoch). This is achieved by efficiently tracking and synchronizing the data dispatching and consumption states as step IDs, *i.e.*, the latest steps whose data has been dispatched/consumed, respectively. Data workers label the produced

mini-batches with the assigned step IDs, and the training worker consumes the data in order of the labeled step IDs. When a data worker fails, it may trigger timeouts for certain mini-batches; then the dispatcher will re-dispatch the data indices for those mini-batches to other data workers. Such re-dispatching may lead to some data being produced multiple times; the training worker can simply discard any data with a step ID no greater than the last consumed one.

To tolerate training workers' failures, we require that the data dispatching order be deterministic despite failures, by fixing the random seeds in operations like shuffling. Therefore, the data consumption state can be reproduced from a step ID. Training workers record the dispatched/consumed step IDs in their checkpoints, so that they can retrieve them after a restart by rollback to the last checkpoint. In distributed training, a single training worker's failure does not necessarily lead to a rollback, *e.g.*, when the model parameters are still alive in parameter servers. To avoid a global rollback in this case, we let training workers update the data consumption state along with the gradients to other workers during each step. It can then easily retrieve the state after a restart as long as there are still workers alive.

For other data formats like big files (*e.g.*, TFRecords), where the mapping from mini-batches to data indices inside files can be non-deterministic due to optimizations like interleaving across multiple files, GoldMiner provides an at-most-once guarantee by treating dispatched data as consumed.

Support for Python functions. TensorFlow allows users to write pure Python code for implementing customized data pre-processing operations (via the `tf.py_function` API). Such Python functions are not serialized in the dataflow graphs; instead, the system registers a Python function with a token and records the token in the graph, which is used for invoking the function object at runtime. To support Python functions for data workers, GoldMiner dispatches the registered function objects to the data worker client processes with the same tokens. Data workers can then retrieve the functions with the tokens recorded in the graph.

Fused data transfer. In TensorFlow distributed training, each tensor is transferred via a dedicated gRPC call. We find this approach could be inefficient, because some models may have a large number of tensors to transfer and need a lot of data workers. For example, the DeepFM model has 141 tensors to transfer from each data worker due to processing operations on different feature columns, and it needs up to 256 data workers to satisfy the training, namely 36,096 RPC calls in each training step. GoldMiner chooses to fuse all the tensors produced by a data worker in a step using a single RPC call, which significantly reduces excessive RPC overhead and improves training performance by up to 25%.

Support for PyTorch. From the design perspective, GoldMiner uses the dataflow graph abstraction of DL training, providing a separation from specific DL frameworks. Although we implement GoldMiner on top of TensorFlow, which provides a static graph abstraction, this is not unique to TensorFlow — for example, PyTorch, another popular DL framework, is also exploring compilation techniques (*e.g.*, TorchDynamo [6]) for building static graphs. We therefore believe the idea of GoldMiner can be extended to other DL frameworks by incorporating such techniques. In particular, we have developed a simplified PyTorch implementation that disaggregates PyTorch DataLoader computations (without further graph analysis) to make GoldMiner available to our PyTorch users. The evaluations below focus on the TensorFlow version to show GoldMiner's full potential.

5 EVALUATION

We evaluate GoldMiner with both single-job performance tests and cluster scheduling experiments using various and realistic workloads. Overall, the key findings include:

- Data workers exhibit near-ideal scalability with low overhead in most cases, making this architecture general enough to benefit various types of workloads.

Type	Model	Dataset
Recommendation	DeepFM[21]	Internal
	DIN[62]	Amazon Dataset[35]
	WDL[14]	
	Production	Internal
CV	AlexNet[31]	ImageNet[16]
	ResNet18[22]	
	ResNet50[22]	
	ShuffleNet_v2[34]	
	VGG16[47]	
	Inception_v3[48]	
NLP	Bert[17]	SQuAD[45]

Table 1. Deep learning models used for the evaluation.

- By scaling out data workers to multiple machines, GoldMiner accelerates model training by up to 12× and reduces monetary cost by up to 88% by using pure CPU machines.
- The automatic graph partitioning outperforms partitioning based on standard dataset APIs by up to 1.6×, and the multi-stage partitioning accelerates training by 1.43× and saves cost by 13% for real production model training.
- By exploiting the elasticity of data workers, GoldMiner reduces average job completion time by up to 2.5×, eliminates queuing delays caused by fragmentation, and improves aggregate GPU utilization by up to 2.1× in a 64-GPU cluster.

Experimental setup. We select 11 representative deep learning models from different domains for the evaluation, as shown in Table 1. The model "Production" is an internal recommendation model from one of the customers of Alibaba Cloud's AI infrastructure. The model has a similar architecture to DCN's [51] (details omitted due to privacy reasons). The CV models use a state-of-the-art data augmentation technique, RandAugment [15], with two layers of random augmentation, which shows to produce the best results on ImageNet in their paper. Except the Production model, we run all the other experiments on a 64-GPU cluster on Alibaba Cloud. The cluster is comprised of 8 VMs, each with 8 NVIDIA V100 GPUs, 64 vCPUs (Intel Xeon Skylake Platinum 8163), 256 GB memory, and 20 Gb/s network bandwith, running CentOS 7.9, CUDA 11, and cuDNN 8. Part of the experiments also use pure CPU machines with 64 vCPUs and 128 GB memory. The Production model runs in a real deployment on VMs with 4 NVIDIA A10 GPUs, 128 vCPUs, 752 GB memory, and 64 Gb/s network bandwidth, and smaller VMs with the same ratio of resources. We calculate speedups by measuring the average throughput over iterations. We also compute standard deviations for the averages. We elide error bars in our plots as the standard deviations are negligible.

5.1 Performance and Overhead

In this section, we examine the performance and scalability of data workers and analyze the associated overheads using different deep learning models and setups. This section does not test the Production model as it is more complex with GPU data pre-processing operations. We will discuss it later.

Figure 9 shows the performance of the 10 deep learning models with increasing numbers of CPUs. For GoldMiner, we run each model using a training worker with one GPU and a number of CPUs that can support the maximum performance of data workers. We measure the training

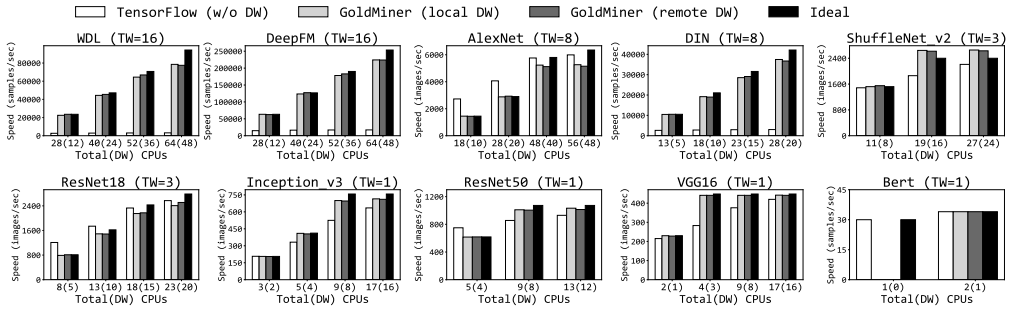

Fig. 9. Training speeds with increasing numbers of CPUs. Training worker CPUs are shown in the subfigure titles.

speeds with increasing numbers of CPUs for data workers. The CPU numbers labeled on the figure show both the total and that for data workers. We compare data workers on the local machine of the training worker and those on remote machines to observe the difference. We also evaluate the vanilla TensorFlow, *i.e.*, non-decoupled execution. For each CPU number, we also show the ideal performance, estimated with the following rule. We first measure the upper bound when data pre-processing is fast enough (by skipping the pre-processing and feeding the model with fake data). For each CPU number, we calculate the speed with linear scaling relative to the smallest CPU number. The ideal is the minimum of this speed and the upper bound.

Scalability. Figure 9 shows that increasing data workers brings near-ideal scalability in most cases. Specifically, for all models and all CPU numbers, we observe performance gaps larger than 10% only for the maximum CPU numbers of DIN (11%), ResNet18 (14%), and WDL (17%). GoldMiner can recognize and react to such decreasing scalability during the dynamic scaling of data workers. Meanwhile, local and remote data workers show nearly the same performance in all cases, because the network communication can be overlapped with the training. We also observe that for some models the vanilla TensorFlow shows limited scalability in the single worker. The most notable examples are the three recommendation models, which exhibit nearly zero speedup when using more CPUs, because they involve a series of single-threaded operations like data parsing in Python and some string transformations. In comparison, GoldMiner can parallelize such computations across data workers.

Overhead. Using data workers may introduce overheads due to the communication between data and training workers, which can consume visible CPU usage of the training worker, especially when the data is large. We compare GoldMiner to the vanilla TensorFlow to analyze the overheads. Overall, using data workers introduces overheads for only two models, AlexNet and ResNet18, with 12.7% and 6.3% slowdown at the peak performance, respectively. Compared to the other models, these two achieve higher training throughputs and have larger amounts of data to transfer between data and training workers, hence the higher overheads. We measure their data transfer throughput (*i.e.*, the bandwidth consumption) to quantify the overhead. As shown in Table 2, the overhead

Model	AlexNet	ResNet18	ResNet50	DeepFM
Overhead	12.7%	6.3%	0%	0%
Throughput (Gb/s)	12.5	6.9	2.7	0.15

Table 2. Overhead vs. throughput.

Model	# extra CPU VMs	Speedup	Cost Saving
ResNet18	1	**1.2×**	**9%**
DeepFM	4	**12.1×**	**88%**

Table 3. Scaling data workers to multiple CPU VMs, compared to using one GPU VM with no data workers.

increases with the data throughput, and becomes zero when the throughput is sufficiently low. GoldMiner compares the performance to the baseline measured without data workers at runtime to identify high-overhead cases and disable data workers accordingly. For the other models, GoldMiner shows better or similar performance, which means that most models can enjoy the flexibility and elasticity of data workers with low overheads.

5.2 Scaling to Multiple Machines

We have shown that some models have a CPU-GPU ratio larger than the hardware ratio, which means the model may need CPUs more than the physical limit of a machine when using multiple GPUs. GoldMiner allows the job to use CPUs on multiple machines, even on pure CPU machines, which can reduce monetary costs.

We train ResNet18 and DeepFM on 8 GPUs to demonstrate this benefit. The baseline is using a whole 8-GPU VM training the models without data workers. For ResNet18, we use 32 CPUs on the GPU VM for the training worker, and the other 32 CPUs plus another 64-CPU VM for data workers. For DeepFM, where the input data size is much smaller, we can use up to 4 extra CPU VMs to achieve higher speedup. The price of the GPU VM is rougly 8.9× that of the CPU VM when we do this experiment. Taking DeepFM as an example, we use 4 CPU VMs to achieve 12.1% speedup, which translates to 8.26% of the time required for training the same number of steps and 44.9% extra price (per hour). Therefore, the total cost is (1+0.449)*0.0826=12%, namely 88% cost reduction.

5.3 Automatic Graph Partitioning

Going beyond standard dataset APIs. It is an intuitive strategy to partition the original graph at the boundary of the standard data pre-processing APIs like `tf.data`. This is in fact how `tf.data.service` [9] disaggregates data pre-processing. But this approach cannot capture some operations that cannot be written with such APIs (*e.g.*, a lot of feature transformations are encapsulated in high-level feature column APIs in TensorFlow). Note that we cannot compare GoldMiner

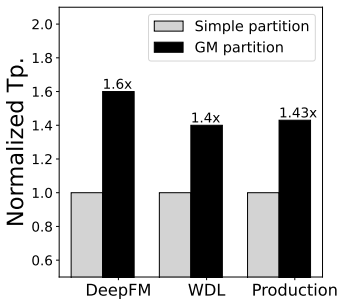

Fig. 10. GoldMiner graph partitioning vs. simple partitioning.

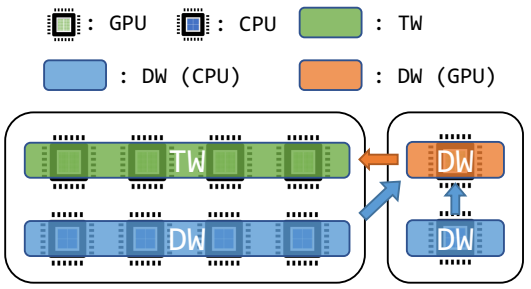

Fig. 11. Multi-stage partitioning: production example.

directly with `tf.data.service` because it is for TensorFlow v2 but GoldMiner is based on TensorFlow v1, the version used by our company and our customers. We therefore compare the automatic graph partitioning strategy against the simple strategy using GoldMiner, focusing on this key difference instead of other implementation differences.

Figure 10 shows the peak performance of three models using the automatic and simple partitioning, normalized by that of the latter. DeepFM and WDL gain 1.6× and 1.4× speedup, respectively, because some heavy feature transformations (*e.g.*, StringHash) get parallelized and overlapped with the training. With the simple partitioning, such operations have to be executed synchronously with other training operations. The Production model also gets 1.43× speedup because the automatic graph partitioning overlaps the Unique operation with the training. Note that the Production model uses GPU Unique, and we will discuss this case in detail next.

Multi-stage partitioning: production case. This section presents a case study of our real production model to show the effect of our multi-stage partitioning. Specifically, this model uses some basic data loading operations on CPUs in the `tf.data` API, followed by a series of feature transformations. In particular, it uses the Unique operation to deduplicate features, which can be extremely slow on CPUs. We therefore have developed a GPU version that is faster by orders of magnitude (as part of the HybridBackend library [2]). This model is trained using 4 A10 GPUs (the largest VM size for A10 from the cloud provider). Running the Unique using data workers on additional GPUs can ideally further overlap its execution. We add more GPUs with VMs of corresponding sizes.

Table 4 compares the single-stage and multi-stage partitioning for the Production model, in terms of normalized performance and cost savings relative to training on 4 GPUs without data workers. Surprisingly, the single-stage partitioning makes it even *worse* than the baseline, *e.g.*, with one additional GPU, it achieves only 74% of the baseline performance. This is because the single-stage data worker has to perform both CPU and GPU computations on the additional 1-GPU VM, and it has only 1/4 of the CPUs on the original 4-GPU VM, making the CPUs the bottleneck. This is exactly the same problem of mismatch between resource requirements and hardware configurations, because the data worker also bundles CPUs and GPUs together. With more GPUs, the single-stage partitioning can achieve speedup, but the cost savings are still negative because the speedup is sublinear.

With the multi-stage partitioning, GoldMiner further decouples the CPU and GPU usage of data workers into different stages, enabling data workers to use all the CPUs, including those on the 4-GPU VM, as illustrated in Figure 11. A single additional GPU for a data worker already achieves the maximum speedup of 1.43×, which needs 4 GPUs with single-stage partitioning, and saves cost by 13%. This is because a single GPU is enough for running Unique for training on 4 GPUs, and the CPUs are no longer a bottleneck. Note that this configuration also outperforms using the additional GPU for training, where the maximum speedup would be only 1.25× (1/4 extra resource for training), and the real speedup is only 1.17× due to communication of distributed training.

GPUs	4+1 (TW+DW)		4+2 (TW+DW)		4+4 (TW+DW)	
	Perf.	Cost Sav.	Perf.	Cost Sav.	Perf.	Cost Sav.
Single	0.74×	-70%	1.13×	-33%	1.43×	-40%
Multi	**1.43×**	**13%**	1.43×	-4.9%	1.43×	-40%

Table 4. Single-stage vs. multi-stage partitioning over the Production model using GPUs for data workers.

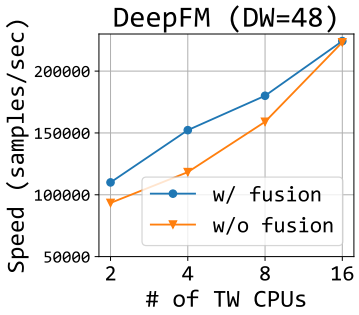

Fig. 12. Fused data transfer.

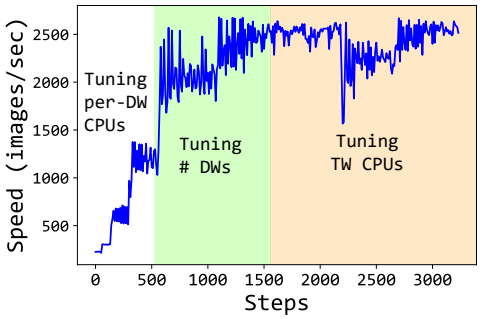

Fig. 13. Dynamic scaling of data and training workers for a ResNet18 job.

5.4 Fused Data Transfer

GoldMiner fuses tensors to transfer between workers to reduce overheads of excessive RPC calls. This is especially helpful for recommendation models with a lot of tensors to transfer as we discussed before. Figure 12 shows the training speeds of DeepFM when using 48 CPUs for data workers with varying numbers of training worker CPUs. The benefit of data fusion is more obvious when the training worker CPUs are more scarce (up to 1.3×), because the training worker is too overloaded to handle the RPCs.

5.5 Dynamic Scaling

GoldMiner dynamically tunes the resource allocation to each job leveraging the elasticity of data workers and queue profiles. We illustrate this process by tuning a one-GPU ResNet18 job and show the training speed over time in Figure 13. For clarity, we increase the resource tuning interval (30 seconds by default) to 2 minutes here. This tuning proceeds in three steps. The job starts with an initial allocation, a training worker container with 8 CPUs and a data worker container with 8 CPUs. It first increases the CPUs of the data worker from 1, and observes linear speedup until the data worker uses all the 8 CPUs. The scheduler then starts tuning the total data worker CPUs by allocating more data workers. This tuning ends up with three data workers in total, as implied by the two times of speed increase in this step shown in Figure 13. In the last step, the job tries to decrease the number of CPUs of the training worker. The job observes stable performance until it decreases the CPU number to 2 (the performance drop in the figure); hence the training worker CPU number rollbacks and finally fixed at 3.

5.6 Cluster Scheduling

Trace construction. We evaluate the scheduling efficiency of GoldMiner on the 64-GPU cluster using a synthetic job trace consisting of the models in Table 1 (except the Production model due to different deployment setups). The submission times, numbers of GPUs, and durations of jobs are sampled from an 8-hour period of the trace published by Microsoft [25]. We observe that in our production cluster there are roughly 50% of jobs whose CPU-GPU ratios are higher than 12, the hardware ratio of the cluster. Therefore, we classify the models in Table 1 into two categories based on if the CPU-GPU ratio is larger than 12: CPU-intensive (WDL, DeepFM, AlexNet, DIN, ShuffleNet_v2, ResNet18) and non-CPU-intensive (Inception_v3, ResNet50, VGG16, Bert). Each category accounts for 50% of the jobs in the trace, and models are uniformly chosen inside each

category. We translate the original job duration to a number of training steps according to the ideal training speed.

Policies. Our scheduler runs the FIFO scheduling policy of the YARN Capacity Scheduler, where jobs are scheduled in order of submission. When placing the jobs, the scheduler tries to pack them compactly to reduce resource fragmentation, *i.e.*, jobs are placed to machines with less free resource. We compare the following three types of approaches for CPU allocation.

No data workers. Each job is specified with a CPU-GPU ratio and is scheduled only when the required resources of all its workers are satisfiable on certain machines. The CPU-GPU ratios are configured in two ways: (1) *manually tuned* ratios that achieve the maximum performance (denoted in Table 5 as "No DW (tuned C/G)"), and (2) the hardware ratio in our testbed (1:8) (denoted as "No DW (HW C/G)"). Note that for the three recommendation models which scale poorly inside a single worker, we allow them to use data workers for higher performance, but we still restrict them to using local data workers with specified amounts of resources.

Fixed data workers. The jobs are allowed to use data workers to disaggregate the CPU computations, but still are scheduled only when it can acquire a *manually tuned* number of TWs to achieve the maximum performance. The scheduler does not adjust the data workers for running jobs.

GoldMiner. We use three policies for our runtime adjustment as described in Section 3.4.2 to show the generality of our scheduling architecture: (1) throughput maximization (MaxTP), (2) job completion time minimization (SRTF), (3) fair allocation (Fair).

Results. Table 5 shows the average job completion times (JCT) and queuing delays in the experiments. Somewhat surprisingly, the baseline of manually tuned CPU-GPU ratios performs the worst, even worse than simply using hardware CPU-GPU ratios. This is because the diverse CPU-GPU ratios fragment the cluster and result in severe queuing delays — the queuing delay accounts for 63% of the JCT (338/540) for this baseline. The baseline with hardware CPU-GPU ratios has near-zero queuing delay due to zero resource fragmentation, but most jobs run at suboptimal performance or use excessive resources, hence the poor JCT. This comparison also shows that although it is possible to hand-tune data pipeline configurations for individual jobs (usually with substantial manual effort), it still needs data-pipeline-aware scheduling policies to improve cluster efficiency. GoldMiner automates this process and improves job and cluster performance simultaneously (see below).

Using data workers, albeit with fixed resource configurations, has visibly reduced queuing delays compared to using tuned CPU-GPU ratios and brings 1.5× JCT gain, showing the increased scheduling flexibility. However, the potential is not fully unleashed because it fails to leverage the elasticity of data workers.

GoldMiner exhibits great advantage over all the baselines, achieving up to 2.5× JCT improvement and reduces queuing delay to zero. This improvement comes from vastly increased resource utilization by elastic scheduling of data workers. Figure 14 shows the resource utilization of two

Scheduler	JCT (mins)	JCT gain	Queuing (mins)
No DW (tuned C/G)	540	**1x**	338
No DW (HW C/G)	495	**1.1x**	4
Fixed DW (tuned)	356	**1.5x**	160
GoldMiner + Fair	352	**1.5x**	14
GoldMiner + MaxTP	231	**2.3x**	0
GoldMiner + SRTF	213	**2.5x**	0

Table 5. Cluster experiment results.

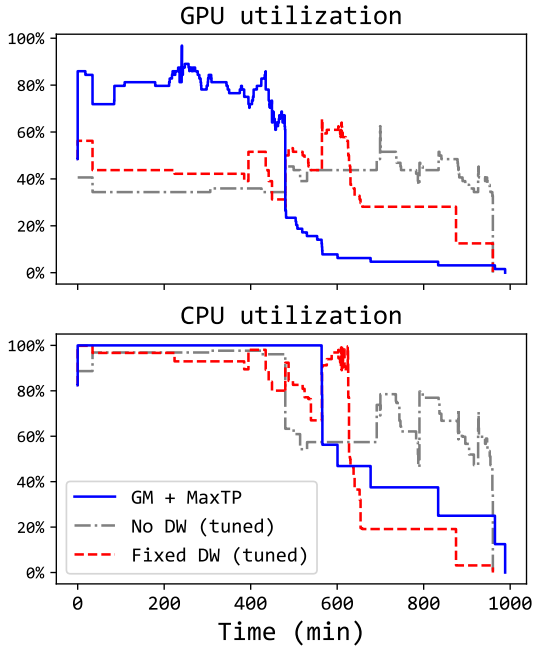

Fig. 14. GPU and CPU utilization over time. Aggregate GPU utilization during 0-500 min: 76% for GM + MaxTP, 35.5% for No DW (tuned), 43.8% for Fixed DW (tuned).

baselines and the MaxTP policy in GoldMiner, where the utilization is defined as the proportion of allocated resources. Due to the resource constraints, the two baselines severely under-utilize GPUs as many jobs are waiting for CPUs. GoldMiner achieves significantly higher GPU utilization and keeps 100% CPU utilization for a long period, finishing most of the jobs much earlier. Figure 14 also shows the aggregate utilization during 0-500 min, *i.e.*, when the cluster is busy for all the three cases, and GoldMiner achieves up to 2.5× improvement.

Among the three policies used by GoldMiner, SRTF achieves additional 8% JCT improvement over MaxTP by preferring jobs that are shorter and require fewer data workers. The fair allocation policy shows the longest JCT because it tries to equalize jobs' slowdown and does not explicitly optimize global performance.

6 RELATED WORK

Training data pipeline acceleration. There are a few concurrent works with GoldMiner that also propose to disaggregate data pre-processing. The `tf.data` service [9] is a new feature of the `tf.data` library [38] that uses a cluster of workers to prepare training data. However, only extracting the computations encapsulated in the `tf.data` API limits its potential, and GoldMiner can automatically identify the maximum scope for the disaggregation. On top of `tf.data` service, Cachew studies auto-scaling policies to dynamically scale the number of `tf.data` workers for individual jobs [18]. GoldMiner takes an important step further towards the cluster level: GoldMiner not only determines the worker configurations of data and training workers (size and number), but also optimizes cluster efficiency using global allocation policies. The Data PreProcessing Service [59] disaggregates computations in the PyTorch Dataset API, sharing similar limitations to `tf.data` service.

DALI [3] is a data loading library that offloads some pre-processing operations to GPUs. DALI is mostly for CV models and usually requires significant rewriting of the code. GoldMiner is a more general and user-transparent architecture and can also incorporate such GPU data pre-processing. CoorDL [36], OneAccess [30], and Isenko et al. [23] also provide a series of useful techniques to accelerate data pre-processing, *e.g.*, caching, compression, data sharing. In general, GoldMiner can incorporate various optimizations for data pipelines and complement them with automatic disaggregation and elastic scheduling. In particular, CoorDL and OneAccess allow a group of hyperparameter search jobs using the same dataset to reuse pre-processed mini-batches by all concurrent jobs. This inspires us to share data workers across jobs as future work.

Dataflow graph partitioning and optimization. Deep learning compilers and systems improves training performance by optimizing dataflow graphs and generating high-efficiency operators or GPU kernels [8, 13, 27, 28, 32, 33, 49, 56, 60, 61]. Besides, dataflow graph partitioning and optimization are also used to generate distributed training plans. Tofu [50] and SOAP [29] automatically partitioning and split tensor operators through searching. Whale [26] further performs resource-aware partition in dataflow graphs to support pipeline and tensor-model parallelism. The works above focus more on the dataflow graph of model training, however, GoldMiner partitions the graph from a new perspective, *i.e.*, partitions and places data pre-processing and CPU-intensive operators into data workers by automatically identifying the boundary by utilizing the characteristic of model updates.

Deep learning cluster schedulers. Many objectives have been explored in DL scheduling to improve the cluster effciency, such as utilization [20, 42], fairness [12, 40, 57]. Performance characteristics of training jobs are utilized in cluster scheduling in many ways. Gandiva [52] performs time-slicing, packing, and migration according to predictability of DL job performance. Antman [54] coordinates GPU memory allocation among jobs. Pollux [44] leverages statistical efficiency to (re-)assign GPUs for elastic training jobs. Recent workss, AutoPS [19], Singularity [46], and Pathways[10], have also explored elastic resource allocations in different ways, including model aggregation in the parameter server architecture, CUDA calls analytics, and heterogeneous interconnects. These cluster schedulers mainly assume GPU as the dominant resource in DL training. GoldMiner is an attempts at addressing the multi-resource fragmentation problem in DL scheduling by the decoupling and elastic scaling of data pre-processing pipelines.

7 CONCLUSION

GoldMiner represents an important step towards resource pooling for training data pipelines of deep learning in the cloud. GoldMiner exploits the characteristics of deep learning training to achieve automatic decoupling, efficient disaggregation, and elastic scheduling of data pre-processing computations. All these combined, GoldMiner brings vastly improved training efficiency, greater resource elasticity, and new scheduling opportunities.

ACKNOWLEDGMENTS

We thank the anonymous SIGMOD reviewers for their insightful feedback and suggestions. We thank Tongxuan Liu and Tao Peng for their help during the implementation and performance optimization of GoldMiner. This work was supported by the National Key Research and Development Program of China (No. 2021ZD0110202) and Alibaba Group through Alibaba Innovative Research Program.

REFERENCES

[1] DeepRec. https://github.com/alibaba/deeprec.
[2] HybridBackend. https://github.com/alibaba/HybridBackend.

[3] NVIDIA DALI. https://github.com/NVIDIA/DALI.

[4] NVIDIA H100. https://www.nvidia.com/en-us/data-center/h100/.

[5] tf.data.experimental.service. https://www.tensorflow.org/api_docs/python/tf/data/experimental/service.

[6] TorchDynamo. https://pytorch.org/docs/master/dynamo/.

[7] Looking beyond GPUs for DNN scheduling on Multi-Tenant clusters. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, Carlsbad, CA, July 2022. USENIX Association.

[8] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. TensorFlow: A System for Large-Scale Machine Learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, volume 16, pages 265–283. USENIX Association, 2016.

[9] Andrew Audibert, Yang Chen, Dan Graur, Ana Klimovic, Jiri Simsa, and Chandramohan A Thekkath. A case for disaggregation of ml data processing. *arXiv preprint arXiv:2210.14826*, 2022.

[10] Paul Barham, Aakanksha Chowdhery, Jeff Dean, Sanjay Ghemawat, Steven Hand, Dan Hurt, Michael Isard, Hyeontaek Lim, Ruoming Pang, Sudip Roy, Brennan Saeta, Parker Schuh, Ryan Sepassi, Laurent El Shafey, Chandramohan A. Thekkath, and Yonghui Wu. Pathways: Asynchronous distributed dataflow for ml. *CoRR*, abs/2203.12533, 2022.

[11] Brendan Burns, Brian Grant, David Oppenheimer, Eric Brewer, and John Wilkes. Borg, Omega, and Kubernetes. *ACM Queue*, 14:70–93, 2016.

[12] Shubham Chaudhary, Ramachandran Ramjee, Muthian Sivathanu, Nipun Kwatra, and Srinidhi Viswanatha. Balancing efficiency and fairness in heterogeneous gpu clusters for deep learning. In *Proceedings of the Fifteenth European Conference on Computer Systems*, EuroSys '20, New York, NY, USA, 2020. Association for Computing Machinery.

[13] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. TVM: An automated end-to-end optimizing compiler for deep learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 578–594, Carlsbad, CA, October 2018. USENIX Association.

[14] Heng-Tze Cheng, Levent Koc, Jeremiah Harmsen, Tal Shaked, Tushar Chandra, Hrishi Aradhye, Glen Anderson, Greg Corrado, Wei Chai, Mustafa Ispir, et al. Wide & deep learning for recommender systems. In *Proceedings of the 1st workshop on deep learning for recommender systems*, pages 7–10, 2016.

[15] Ekin D Cubuk, Barret Zoph, Jonathon Shlens, and Quoc V Le. Randaugment: Practical automated data augmentation with a reduced search space. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition Workshops*, pages 702–703, 2020.

[16] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. Imagenet: A large-scale hierarchical image database. In *Computer Vision and Pattern Recognition, 2009. CVPR 2009. IEEE Conference on*, pages 248–255. IEEE, 2009.

[17] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: pre-training of deep bidirectional transformers for language understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL-HLT 2019, Minneapolis, MN, USA, June 2-7, 2019, Volume 1 (Long and Short Papers)*, pages 4171–4186. Association for Computational Linguistics, 2019.

[18] Dan Graur, Damien Aymon, Dan Kluser, Tanguy Albrici, Chandramohan A Thekkath, and Ana Klimovic. Cachew: Machine learning input data processing as a service. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, pages 689–706, 2022.

[19] Juncheng Gu, Mosharaf Chowdhury, Kang G. Shin, and Aditya Akella. Elastic model aggregation with parameter service. *CoRR*, abs/2204.03211, 2022.

[20] Juncheng Gu, Mosharaf Chowdhury, Kang G Shin, Yibo Zhu, Myeongjae Jeon, Junjie Qian, Hongqiang Liu, and Chuanxiong Guo. Tiresias: A GPU cluster manager for distributed deep learning. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 485–500, 2019.

[21] Huifeng Guo, Ruiming Tang, Yunming Ye, Zhenguo Li, and Xiuqiang He. Deepfm: A factorization-machine based neural network for CTR prediction. In *Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence, IJCAI 2017, Melbourne, Australia, August 19-25, 2017*, pages 1725–1731. ijcai.org, 2017.

[22] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[23] Alexander Isenko, Ruben Mayer, Jeffrey Jedele, and Hans-Arno Jacobsen. Where is my training bottleneck? hidden trade-offs in deep learning preprocessing pipelines. In *Proceedings of the 2022 International Conference on Management of Data*, SIGMOD '22, page 1825–1839, New York, NY, USA, 2022. Association for Computing Machinery.

[24] Alexander Isenko, Ruben Mayer, Jedele Jeffrey, and Hans-Arno Jacobsen. Where is my training bottleneck? hidden trade-offs in deep learning preprocessing pipelines. In *Proceedings of the 2022 International Conference on Management of Data (SIGMOD)*, 2022.

[25] Myeongjae Jeon, Shivaram Venkataraman, Amar Phanishayee, Junjie Qian, Wencong Xiao, and Fan Yang. Analysis of large-scale multi-tenant GPU clusters for DNN training workloads. In *2019 USENIX Annual Technical Conference*

(USENIX ATC 19), pages 947–960, 2019.

[26] Xianyan Jia, Le Jiang, Ang Wang, Wencong Xiao, Ziji Shi, Jie Zhang, Xinyuan Li, Langshi Chen, Yong Li, Zhen Zheng, Xiaoyong Liu, and Wei Lin. Whale: Efficient giant model training over heterogeneous GPUs. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, Carlsbad, CA, July 2022. USENIX Association.

[27] Zhihao Jia, Oded Padon, James Thomas, Todd Warszawski, Matei Zaharia, and Alex Aiken. Taso: optimizing deep learning computation with automatic generation of graph substitutions. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pages 47–62, 2019.

[28] Zhihao Jia, James Thomas, Tod Warszawski, Mingyu Gao, Matei Zaharia, and Alex Aiken. Optimizing dnn computation with relaxed graph substitutions. *SysML 2019*, 2019.

[29] Zhihao Jia, Matei Zaharia, and Alex Aiken. Beyond data and model parallelism for deep neural networks. In A. Talwalkar, V. Smith, and M. Zaharia, editors, *Proceedings of Machine Learning and Systems*, volume 1, pages 1–13, 2019.

[30] Aarati Kakaraparthy, Abhay Venkatesh, Amar Phanishayee, and Shivaram Venkataraman. The case for unifying data loading in machine learning clusters. In *11th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 19)*, Renton, WA, July 2019. USENIX Association.

[31] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet Classification with Deep Convolutional neural Networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.

[32] Chris Lattner, Jacques A. Pienaar, Mehdi Amini, Uday Bondhugula, River Riddle, Albert Cohen, Tatiana Shpeisman, Andy Davis, Nicolas Vasilache, and Oleksandr Zinenko. MLIR: A compiler infrastructure for the end of moore's law. *CoRR*, abs/2002.11054, 2020.

[33] Lingxiao Ma, Zhiqiang Xie, Zhi Yang, Jilong Xue, Youshan Miao, Wei Cui, Wenxiang Hu, Fan Yang, Lintao Zhang, and Lidong Zhou. Rammer: Enabling holistic deep learning compiler optimizations with rtasks. In *14th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 20)*, pages 881–897, 2020.

[34] Ningning Ma, Xiangyu Zhang, Hai-Tao Zheng, and Jian Sun. Shufflenet v2: Practical guidelines for efficient cnn architecture design. In *Proceedings of the European conference on computer vision (ECCV)*, pages 116–131, 2018.

[35] Julian McAuley, Christopher Targett, Qinfeng Shi, and Anton Van Den Hengel. Image-based recommendations on styles and substitutes. In *Proceedings of the 38th international ACM SIGIR conference on research and development in information retrieval*, pages 43–52, 2015.

[36] Jayashree Mohan, Amar Phanishayee, Ashish Raniwala, and Vijay Chidambaram. Analyzing and mitigating data stalls in dnn training. In *VLDB 2021*, January 2021.

[37] Derek G Murray, Frank McSherry, Rebecca Isaacs, Michael Isard, Paul Barham, and Martín Abadi. Naiad: a timely dataflow system. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 439–455, 2013.

[38] Derek G. Murray, Jiří Šimša, Ana Klimovic, and Ihor Indyk. Tf.data: A machine learning data processing framework. *Proc. VLDB Endow.*, 14(12):2945–2958, jul 2021.

[39] Deepak Narayanan, Aaron Harlap, Amar Phanishayee, Vivek Seshadri, Nikhil R Devanur, Gregory R Ganger, Phillip B Gibbons, and Matei Zaharia. Pipedream: generalized pipeline parallelism for dnn training. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pages 1–15, 2019.

[40] Deepak Narayanan, Keshav Santhanam, Fiodar Kazhamiaka, Amar Phanishayee, and Matei Zaharia. Heterogeneity-aware cluster scheduling policies for deep learning workloads. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 481–498. USENIX Association, November 2020.

[41] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. Pytorch: An imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems*, pages 8024–8035, 2019.

[42] Yanghua Peng, Yixin Bao, Yangrui Chen, Chuan Wu, and Chuanxiong Guo. Optimus: An Efficient Dynamic Resource Scheduler for Deep Learning Clusters. In *Proceedings of the Thirteenth European Conference on Computer Systems*. ACM, 2018.

[43] Luis Perez and Jason Wang. The effectiveness of data augmentation in image classification using deep learning. *arXiv preprint arXiv:1712.04621*, 2017.

[44] Aurick Qiao, Sang Keun Choe, Suhas Jayaram Subramanya, Willie Neiswanger, Qirong Ho, Hao Zhang, Gregory R. Ganger, and Eric P. Xing. Pollux: Co-adaptive cluster scheduling for goodput-optimized deep learning. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*, pages 1–18. USENIX Association, July 2021.

[45] Pranav Rajpurkar, Jian Zhang, Konstantin Lopyrev, and Percy Liang. Squad: 100, 000+ questions for machine comprehension of text. In *Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing, EMNLP 2016, Austin, Texas, USA, November 1-4, 2016*, pages 2383–2392. The Association for Computational Linguistics, 2016.

[46] Dharma Shukla, Muthian Sivathanu, Srinidhi Viswanatha, Bhargav Gulavani, Rimma Nehme, Amey Agrawal, Chen Chen, Nipun Kwatra, Ramachandran Ramjee, Pankaj Sharma, et al. Singularity: Planet-scale, preemptible, elastic scheduling of ai workloads. *CoRR*, abs/2202.07848, 2022.

[47] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. In *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, 2015.

[48] Christian Szegedy, Vincent Vanhoucke, Sergey Ioffe, Jon Shlens, and Zbigniew Wojna. Rethinking the inception architecture for computer vision. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 2818–2826, 2016.

[49] Haojie Wang, Jidong Zhai, Mingyu Gao, Zixuan Ma, Shizhi Tang, Liyan Zheng, Yuanzhi Li, Kaiyuan Rong, Yuanyong Chen, and Zhihao Jia. {PET}: Optimizing tensor programs with partially equivalent transformations and automated corrections. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*, pages 37–54, 2021.

[50] Minjie Wang, Chien-chin Huang, and Jinyang Li. Supporting very large models using automatic dataflow graph partitioning. In *Proceedings of the Fourteenth EuroSys Conference 2019*, EuroSys '19, New York, NY, USA, 2019. Association for Computing Machinery.

[51] Ruoxi Wang, Rakesh Shivanna, Derek Cheng, Sagar Jain, Dong Lin, Lichan Hong, and Ed Chi. Dcn v2: Improved deep & cross network and practical lessons for web-scale learning to rank systems. In *Proceedings of the Web Conference 2021*, pages 1785–1797, 2021.

[52] Wencong Xiao, Romil Bhardwaj, Ramachandran Ramjee, Muthian Sivathanu, Nipun Kwatra, Zhenhua Han, Pratyush Patel, Xuan Peng, Hanyu Zhao, Quanlu Zhang, Fan Yang, and Lidong Zhou. Gandiva: Introspective cluster scheduling for deep learning. In *13th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2018, Carlsbad, CA, USA, October 8-10, 2018*, pages 595–610. USENIX Association, 2018.

[53] Wencong Xiao, Zhenhua Han, Hanyu Zhao, Xuan Peng, Quanlu Zhang, Fan Yang, and Lidong Zhou. Scheduling CPU for gpu-based deep learning jobs. In *Proceedings of the ACM Symposium on Cloud Computing, SoCC 2018, Carlsbad, CA, USA, October 11-13, 2018*, page 503. ACM, 2018.

[54] Wencong Xiao, Shiru Ren, Yong Li, Yang Zhang, Pengyang Hou, Zhi Li, Yihui Feng, Wei Lin, and Yangqing Jia. Antman: Dynamic scaling on GPU clusters for deep learning. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 533–548. USENIX Association, November 2020.

[55] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, and Ion Stoica. Resilient Distributed Datasets: A Fault-tolerant Abstraction for In-memory Cluster Computing. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, NSDI'12. USENIX, 2012.

[56] Yuanxing Zhang, Langshi Chen, Siran Yang, Man Yuan, Huimin Yi, Jie Zhang, Jiamang Wang, Jianbo Dong, Yunlong Xu, Yue Song, Yong Li, Di Zhang, Wei Lin, Lin Qu, and Bo Zheng. Picasso: Unleashing the potential of gpu-centric training for wide-and-deep recommender systems. In *2022 IEEE 38th International Conference on Data Engineering (ICDE)*. IEEE, 2022.

[57] Hanyu Zhao, Zhenhua Han, Zhi Yang, Quanlu Zhang, Fan Yang, Lidong Zhou, Mao Yang, Francis C.M. Lau, Yuqi Wang, Yifan Xiong, and Bin Wang. Hived: Sharing a GPU cluster for deep learning with guarantees. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 515–532. USENIX Association, November 2020.

[58] Mark Zhao, Niket Agarwal, Aarti Basant, Bugra Gedik, Satadru Pan, Mustafa Ozdal, Rakesh Komuravelli, Jerry Pan, Tianshu Bao, Haowei Lu, et al. Understanding and co-designing the data ingestion pipeline for industry-scale recsys training. *arXiv preprint arXiv:2108.09373*, page 4, 2021.

[59] Mark Zhao, Niket Agarwal, Aarti Basant, Bugra Gedik, Satadru Pan, Mustafa Ozdal, Rakesh Komuravelli, Jerry Pan, Tianshu Bao, Haowei Lu, Sundaram Narayanan, Jack Langman, Kevin Wilfong, Harsha Rastogi, Carole-Jean Wu, Christos Kozyrakis, and Parik Pol. Understanding and co-designing the data ingestion pipeline for industry-scale recsys training. *CoRR*, abs/2108.09373, 2021.

[60] Lianmin Zheng, Chengfan Jia, Minmin Sun, Zhao Wu, Cody Hao Yu, Ameer Haj-Ali, Yida Wang, Jun Yang, Danyang Zhuo, Koushik Sen, et al. Ansor: Generating high-performance tensor programs for deep learning. In *14th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 20)*, pages 863–879, 2020.

[61] Zhen Zheng, Xuanda Yang, Pengzhan Zhao, Guoping Long, Kai Zhu, Feiwen Zhu, Wenyi Zhao, Xiaoyong Liu, Jun Yang, Jidong Zhai, Shuaiwen Leon Song, and Wei Lin. Astitch: Enabling a new multi-dimensional optimization space for memory-intensive ml training and inference on modern simt architectures. ASPLOS 2022, page 359–373, New York, NY, USA, 2022. Association for Computing Machinery.

[62] Guorui Zhou, Xiaoqiang Zhu, Chenru Song, Ying Fan, Han Zhu, Xiao Ma, Yanghui Yan, Junqi Jin, Han Li, and Kun Gai. Deep interest network for click-through rate prediction. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pages 1059–1068, 2018.

Received November 2022; revised February 2023; accepted March 2023