Hare: Exploiting Inter-job and Intra-job Parallelism of Distributed Machine Learning on Heterogeneous GPUs

Fahao Chen
School of Computer Science and Engineering
The University of Aizu, Japan
d8232101@u-aizu.ac.jp

Peng Li
School of Computer Science and Engineering
The University of Aizu, Japan
pengli@u-aizu.ac.jp

Celimuge Wu
Department of Computer and Network Engineering
Graduate School of Informatics and Engineering
University of Electro-Communications, Japan
celimuge@uec.ac.jp

Song Guo
Department of Computing
The Hong Kong Polytechnic University & The Hong Kong Polytechnic University Shenzhen Research Institute
song.guo@polyu.edu.hk

ABSTRACT

Distributed machine learning (DML) has shown great promise in accelerating model training on multiple GPUs. To increase GPU utilization, a common practice is to let multiple learning jobs share GPU clusters, where the most fundamental and critical challenge is how to efficiently schedule these jobs on GPUs. However, existing works about DML job scheduling are constrained to settings with homogeneous GPUs. GPU heterogeneity is common in practice, but its influence on multiple DML job scheduling has been seldom studied. Moreover, DML jobs have internal structures that contain great parallelism potentials, which have not yet been fully exploited in the heterogeneous computing environment. In this paper, we propose *Hare*, a DML job scheduler that exploits both inter-job and intra-job parallelism in a heterogeneous GPU cluster. *Hare* has three novel designs. First, *Hare* optimizes GPU execution environment to reduce task switching overhead by exploiting unique features of DML scheduling. Second, *Hare* adopts a relaxed fixed-scale synchronization scheme that allows independent tasks to be flexibly scheduled within a training round. Finally, we propose a fast heuristic algorithm to minimize the total weighted job completion time by jointly considering job features and hardware heterogeneity. Its theoretical bound is derived. We evaluate *Hare* using a small-scale testbed and a trace-driven simulator. The results show that it can outperform the state-of-the-art by about 2x.

CCS CONCEPTS

• **Computing methodologies** → **Distributed computing methodologies**.

KEYWORDS

Distributed machine learning; heterogeneous GPUs; scheduling

ACM Reference Format:
Fahao Chen, Peng Li, Celimuge Wu, and Song Guo. 2022. *Hare*: Exploiting Inter-job and Intra-job Parallelism of Distributed Machine Learning on Heterogeneous GPUs. In *Proceedings of the 31st International Symposium on High-Performance Parallel and Distributed Computing (HPDC '22), June 27–30, 2022, Minneapolis, MN, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3502181.3531462

HPDC '22, June 27–30, 2022, Minneapolis, MN, USA.
2022. ACM ISBN 978-1-4503-9199-3/22/06…$15.00
https://doi.org/10.1145/3502181.3531462

1 INTRODUCTION

Background. The recent success of machine learning, especially deep learning, stems from the availability of big data and strong computational power brought by cutting-edge hardware (e.g., GPUs and TPUs). Facing massive computational loads, it is inefficient or sometimes impossible to train models on a single GPU, driving attention towards distributed machine learning on multiple GPUs. In the paradigm of distributed machine learning (DML), a learning job is divided into multiple tasks, which can run on multiple GPUs in parallel. The Parameter Server (PS) [25] scheme has been widely adopted to coordinate the training processes across multiple GPUs.

In practice, it is rare to assign a dedicated GPU cluster to each DML job, due to low resource utilization [26]. Instead, a common practice is to let multiple jobs share these GPUs. A critical research challenge is how to efficiently schedule these jobs on GPUs, which is particularly concerned by public or private cloud data centers that offer learning services while desiring high hardware resource utilization. Therefore, the learning job scheduling problem has attracted great research attention, and various solutions have been recently proposed with different objectives. For example, Gandiva [41] has studied GPU sharing among several jobs to improve GPU utilization. The fairness of learning jobs has been studied by Pollux [32]. Zhang et al. [47] have exploited both intra-job and inter-job parallelism and proposed online DML job scheduling algorithms to minimize job completion time.

Limitation of state-of-art approaches. However, the above works are all based on an assumption that GPUs are homogeneous. In practice, hardware heterogeneity commonly exists in computing clusters. For example, as the expansion of data centers, new GPUs are continuously added and they should work with existing ones to maximize resource utilization. Some recent works [9, 24, 29] have started to pay attention to the influence of GPU-heterogeneity, which motivates us to re-examine the DML job scheduling problem in such an emerging heterogeneous computing environment. We find that existing works with the homogeneity assumption cannot fully achieve their claimed goals in heterogeneous environment. That is because they expect that training tasks scheduled simultaneously on several machines have the same completion time. However, when these tasks actually run on heterogeneous machines, they complete at different time. Due to the task synchronization at the

end of each training round, the faster tasks need to wait for slower ones, which may lead to longer job completion time.

Hardware heterogeneity brings new challenges as well as opportunities to DML system design. We are excited to see the success of several preliminary studies. For example, Gandiva$_{\text{fair}}$ [9] is designed to ensure the user-level fairness while maximizing the efficiency of heterogeneous GPU clusters. Gavel [29] generalizes existing scheduling policies with consideration of GPU heterogeneity. Allox [24] efficiently schedules ML jobs in a heterogeneous cluster to improve the max-min fairness. These recent works have extensively studied inter-job parallelism in heterogeneous computing environment, but leaving intra-job parallelism unexplored. They treat each DML job as a unsplittable unit when making scheduling decisions. We are still facing open questions: how to exploit both inter-job and intra-job parallelism on heterogeneous GPUs? How much acceleration can be obtained? And is there strong theoretical support for such acceleration?

Key insights and contributions. In this paper, we propose *Hare* for heterogeneous GPU cluster scheduling, to answer the above questions. The basic idea of *Hare* can be illustrated using the example in Fig. 1. There are 3 jobs, and every job consists of several tasks, each of which responsible for training a data batch. We further assume that job J3 needs to synchronize for every two tasks. The single-batch training time on 3 different GPUs is shown in the table. By following the heterogeneity-oblivious strategies in [47], we get the scheduling results shown in Fig. 1(a), where the job J3 uses GPU2 and GPU3 to exploit intra-job parallelism and J2 takes the whole GPU1. When both jobs complete, we start J1 that runs on two GPUs in parallel. The total job completion time is 10.5 seconds and the makespan is 4.5 seconds. The results of job-level scheduling aware of GPU-heterogeneity, represented by Allox [24], are shown in Fig. 1(b). Each job gets a dedicated GPU and the total completion time of 9 seconds. An alternative scheduling result with better performance is shown in Fig. 1(c), where the idle time on GPU3 can be used by J1. This scheme reduces total job completion time to 8.5 seconds and makespan to 3 seconds. Note that although communication time and task switching overhead is ignored in this example for simplicity, we have similar observation even if they are considered.

Despite the promise of GPU-heterogeneity-awareness and intra-job parallelism, *Hare* needs to conquer several critical technical challenges to grasp the promised benefits. First, to use the idle time on some GPUs before model synchronization, as shown in Fig. 1(c), the scheduler needs to allow GPU preemption during learning job execution, which is forbidden by existing works [32, 41, 47]. Moreover, with such GPU preemption, switching between tasks belonging to different jobs should be quick. Otherwise, frequent task switching may happen in *Hare*, leading to ruinous overhead. To solve this challenge, *Hare* first enables fast task switching by optimizing task initialization and cleaning on GPUs, which has been identified as the major source of switching overhead. We use some methods, e.g., CUDA context sharing, that have been shown to be effective in reducing task switching overhead [8]. Moreover, we exploit the unique features of *Hare* to further improve performance by proposing early task cleaning and speculative memory management.

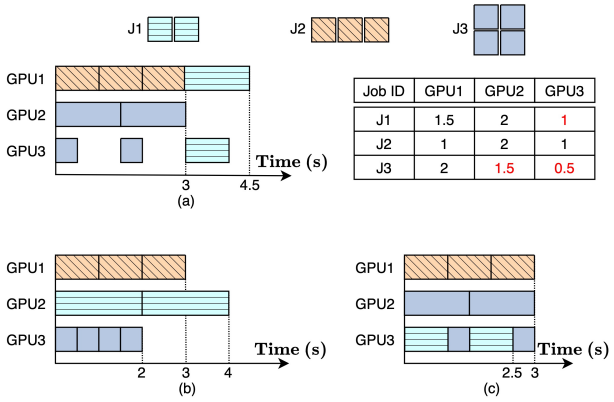

Figure 1: A toy example to show job scheduling results under different methods. (a) GPU-heterogeneity-oblivious scheduling result; (b) GPU-heterogeneity-aware scheduling result, but without exploiting intra-job parallelism; (c) A better scheduling result jointly considering GPU heterogeneity and intra-job parallelism.

Second, existing intra-job synchronization schemes are not flexible enough, which constrains the optimization space of *Hare*. Synchronization schemes decide how many independent tasks are launched in a training round and how to synchronize these tasks. Two synchronization schemes have been widely adopted. A scale-fixed scheme always launch the same number of tasks in each training round and schedule them when the same number of GPUs are available to maximize parallelism. Instead, a scale-adaptive scheme can adjust the number of parallel tasks according to available GPU resources. Although scale-adaptive scheme is more flexible but may lead to uncertainty in convergence. Comparison details of both schemes can be found in Section 2.2. Motivated by their pros and cons, we propose a relaxed scale-fixed synchronization scheme for *Hare* to maximize scheduling flexibility. It fixes the number of tasks in each rounds but relaxes resource requirement for scheduling, so that we can maintain convergence certainty while maximizing GPU utilization.

The final challenge is about scheduling algorithm design. We need to exploit the parallelism at both intra-job and inter-job levels while considering GPU heterogeneity. Different jobs may prefer different GPUs because their models and training datasets are diverse. A sophisticated scheduler should make careful decisions to optimize the overall performance. Thanks to our proposed fast task switching mechanism, the switching overhead is so tiny that we can ignore it in the scheduling algorithm design for simplicity. Even though, the scheduling problem is still NP-hard. We propose a fast heuristic algorithm to minimize the total weighted job completion time and derive an important theoretical approximation ratio to the optimal solutions.

Experimental methodology. We develop a prototype of *Hare* based on PyTorch 1.8.1 by adding about 2500 LoC of Python and C++. We build a testbed consisting of 15 heterogeneous GPUs (8 V100s, 4 T4s, 1 K80, and 2 M60s) for performance evaluation. The workloads contain 8 types of jobs, which train different deep learning models

respectively. The details of models and datasets are shown in Table 2. We consider 4 scheduling algorithms proposed by recent works as comparison baselines.

We also develop a simulator using Python to evaluate *Hare* under large-scale settings. The accuracy of our simulator has been verified by comparing its results with the one obtained from the testbed. They have no more than 5% difference. We collect tasks' running traces from our testbed and synthesize large-scale traces to feed this simulator. The results show that *Hare* outperforms baselines under various settings.

Limitations of the proposed approach. Since *Hare* currently uses an offline scheduling algorithm, it is short in handling dynamic jobs. This dynamic comes from two kinds of cases. First, jobs may change settings (e.g., hyper-parameters or parallelism levels) during running. For example, some jobs use the autoML technique to search for the best models over different hyper-parameters or even model structures. *Hare* needs to frequently profile task running time, which may cause high overhead. Second, jobs arrive in different time and we cannot accurately predict future job arrivals. Online algorithms are needed to address the dynamic in both cases.

Despite these limitations, we believe this paper still have important and sufficient contributions. We have strong theoretical results for offline scheduling. A basic scheduling system has been built and it can be easily extended to accommodate other scheduling algorithms for dynamic jobs, which is left for future work.

2 BACKGROUND AND MOTIVATION

In this section, we first give the background of distributed machine learning (DML), and then present the unique characters of DML jobs, which motivate us to design *Hare*.

2.1 Background

Distributed machine learning (DML) on GPUs has been widely adopted to accelerate model training on large datasets. The training goal is to minimize a loss function as follows:

$$\mathcal{L}(w) = \frac{1}{|\mathcal{P}|} \sum_{p_i \in \mathcal{P}} \ell(\mathbf{w}, p_i), \tag{1}$$

where p_i is a data point in the training dataset $\mathcal{P}$. The loss function $\ell(\cdot)$ is, typically cross-entropy for the classification problem or squared error for the regression problem. The trainable model parameters $\mathbf{w}$ are updated iteratively by using stochastic gradient descent (SGD).

In each iteration, training workloads are shared by $\mathcal{K}$ GPUs, which are also called workers. Each worker is assigned a fixed-size mini-batch $\mathcal{B}_k \subseteq \mathcal{P}$ and computes its local gradients g_k^t as:

$$g_k^t = \frac{1}{|\mathcal{B}_k|} \sum_{p_i \in \mathcal{B}_k} \nabla \ell(\mathbf{w}^t, p_i). \tag{2}$$

After the local training, workers send their gradient updates to a parameter server that creates a global model:

$$g^t = \frac{1}{|\mathcal{K}|} \sum_{k \in \mathcal{K}} g_k^t; \qquad \mathbf{w}^{t+1} = \mathbf{w}^t - \eta g^t, \tag{3}$$

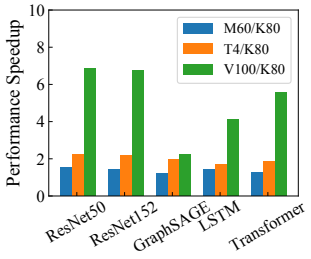

Figure 2: Training speedup of different jobs on different GPUs.

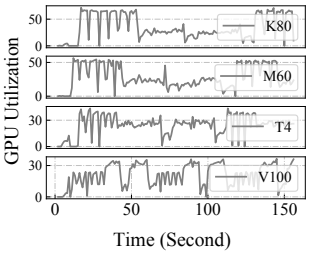

Figure 3: The GPU utilization of training GraphSAGE model.

where η is the learning rate. Then workers download the global model and move to the next iteration of training. The training process ends when a required number of rounds is achieved. Typically, the parallelism scale $|\mathcal{K}|$, the batch size $|\mathcal{B}|$, and the learning rate η are chosen by the user.

Typically, multiple DML jobs share GPU resources in a cluster. With the soaring size of the DML jobs, a sophisticated scheduler is needed to shorten the training time and improve GPU utilization. Existing works [9, 19, 26, 29, 41, 47] have made many efforts on scheduling algorithms design with an assumption that GPUs in the cluster are homogeneous. However, existing clusters usually accommodate different types of GPUs with various specifications, which implies the inefficiency of existing homogeneity-based schedulers.

2.2 Motivation

2.2.1 GPU heterogeneity and inter-job parallelism. We find that different GPUs provide different performance speedups for learning jobs, mainly because of the heterogeneity of model (such as model architecture) and hardware. As shown in Fig. 2, we use the training time per mini-batch on a K80 GPU as the baseline and evaluate the speedup for other GPUs. Training the ResNet50 model can be sped up by 2x on a T4 GPU, while with 7x significant speedup on a V100 GPU. However, the graph learning model GraphSAGE shows the heterogeneous performance on different GPUs. Specifically, GraphSAGE can only be sped up by about 2x, even on the most advanced V100 GPU. That is because the required FLOPS of GraphSAGE are much smaller than other models. Moreover, the data pre-processing speed is slower than the GPU computation speed. The GPU spends more time to wait for input data, resulting in low GPU utilization. As shown in Fig. 3, we find that utilization of GPU is less than 30% when we train GraphSAGE on a V100 GPU. There is a little improvement when training GraphSAGE on a V100 GPU. Therefore, giving a high priority for assigning V100 GPUs to the ResNet50 job is more efficient since it shows a high-performance speedup than other jobs.

This empirical study gives us important hints about accelerating learning jobs and increasing GPU utilization. On the other hand, it throws challenges about how to schedule jobs on GPUs, considering massive learning workloads and hardware resources in modern data centers. Moreover, the intra-job parallelism, which will presented in the following, further complicates this problem.

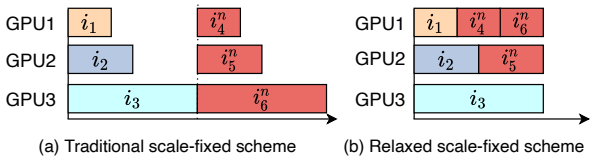

Figure 4: An example showing the benefit of relaxed scale-fixed synchronization scheme adopted by *Hare*.

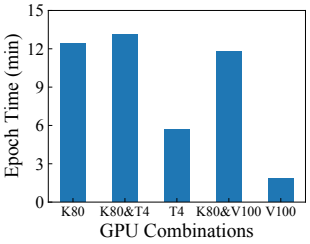

Figure 5: Epoch time of ResNet152 under different GPU combinations.

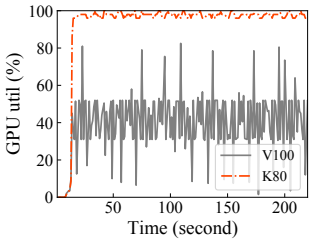

Figure 6: GPU utilization of V100 and K80 when training ResNet152.

2.2.2 GPU heterogeneity and intra-job parallelism. Each DML job consists of multiple tasks, which are periodically synchronized to share gradients, via a parameter server or exchanging gradients directly. Although a single advanced GPU can provide a performance speedup for gradients computing, the training speed of the whole DML jobs is constrained by the synchronization. We train the ResNet152 on five different distributed settings and show the epoch time in Fig. 5. We find that mixing different GPUs is not always helpful. For example, compared to a pure K80 cluster, adding faster T4 or V100 brings no acceleration. That is because the gradient synchronization impedes early completed GPUs to move to the next-round training. There is much idle time on V100 GPUs when they waits for the gradients update from K80 GPUs. This low efficiency can be also reflected by GPU utilization as shown in Fig. 6, where we can see that K80 is always busy while V100's utilization is rarely over 50%.

A straightforward idea to address this challenge is to schedule parallel tasks belonging to the same job on similar GPUs. However, it is hardly to have such a perfect allocation in practice because of limited GPU resources in the cluster. Since it is inevitable to use heterogeneous GPUs for intra-job parallelism, it is desired an algorithm that can well schedule fine-grained tasks to reduce idle time.

2.2.3 Scale-fixed synchronization versus scale-adaptive synchronization. Existing intra-job parallelism methods can be categorized into two types, scale-fixed and scale-adaptive, according to how many tasks are synchronized. Scale-fixed methods, adopted by Tiresias [19] and Gandiva [41], fix the number of synchronized tasks and always try to allocate the same number of GPUs so that they can achieve full parallelism. If the number of available GPUs is insufficient, all tasks need to wait until required GPU number is satisfied. In contrast, scale-adaptive methods [26, 29, 31, 42] dynamically change the number of synchronized tasks according to available GPU resources. Although these methods are flexible and tasks are not blocked by strict resource requirement, we may need more training epochs to achieve competitive accuracy of scale-fixed methods. Moreover, it is hard to build theories to predict how many epochs are needed. Due to this uncertainty, we do not use scale-adaptive design in *Hare*.

Motivated by the above analysis, we would like to follow the scale-fixed idea but relax the parallelism requirement. An example is shown in Fig. 4, where three tasks, i_1, i_2 and i_3, are running on 3 GPUs respectively. Now a new job n consisting of 3 tasks (i.e., synchronization scale is 3) comes. As illustrated in Fig. 4(a), traditional scale-fixed methods start job n after the completion of slowest task i_3, when 3 GPUs are available. We find that it is unnecessary to make 3 tasks strictly run in parallel. Two tasks can run sequentially on GPU1, as shown in Fig. 4(b), leading to earlier completion than traditional methods while maintaining the same level of parallelism.

Implementing such a relaxed scale-fixed synchronization method is not easy. We need to address challenge of changing the task assignment and synchronization modules. It also affects task scheduling algorithm design.

2.2.4 Task Switching Cost. As we are motivated above, exploiting intra-job parallelism on heterogeneous GPU environment is critical for accelerating training and increasing hardware resource utilization. It should be taken into consideration of scheduling algorithm design. We further find that such an algorithm inevitable generates results with frequent task switching on GPUs. To study task switching cost on GPUs, we conduct experiments to compare switching time and task time under 3 different settings. In the first setting, we alternatively run a GraphSAGE task and a ResNet50 task, each of which trains a mini-batch. We define a metric $\Omega = \frac{t_{sw}}{t_c^g + t_c^r}$ to evaluate the switching cost, where t_{sw} is the task switching time, t_c^g and t_c^r is the average batch training time of GraphSAGE and ResNet50, respectively. As shown in Fig. 7, the switching cost is about 9 times higher than training. Similar high cost can be observed under other two settings. We also show the real-time GPU utilization with and without task switching in Fig. 8. When training a single ResNet50 model on a V100 GPU, GPU resources are almost fully utilized. However, if we train GraphSAGE and ResNet50 alternatively, GPU utilization is no more than 50%, because much time is spent on CUDA environment cleaning and creation during task switching.

Even though we can continuously schedule the same-type tasks on a GPU, which is possible due to the proposed relaxed scale-fixed method, to amortize switching cost. However, the switching cost is still high and such a solution heavily relies on sophisticated design of scheduling algorithm that takes switching cost into consideration. In this paper, instead of struggling on amortizing switching cost, we propose to reduce it using novel system designs.

3 SYSTEM OVERVIEW

In this section, we give an overview about *Hare*'s design. First, we clarify three primary design goals of *Hare*.

- **High training efficiency:** Given a number of DML jobs, *Hare* needs to schedule them on a cluster of GPUs to finish

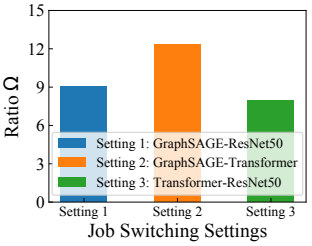

Figure 7: Ratio of switching time and training time under three settings.

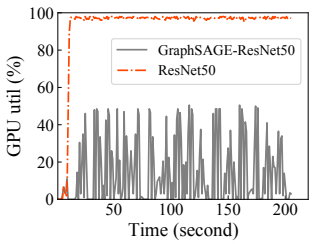

Figure 8: V100 GPU utilization with and without task switching.

the training as fast as possible. It depends on exploiting both intra-job and inter-job parallelism as well as GPU heterogeneity, as we shown in the motivation section. We achieve this goal by designing a high-efficient task scheduling algorithm with strong theoretical performance guarantee.

- **High GPU utilization:** Hardware utilization is always important for cluster providers. *Hare* is not only a job scheduler but also a resource manager in the GPU cluster. Therefore, *Hare* aims to improve the GPU utilization by reducing system cost and minimizing GPU idle time.
- **Starvation-free:** In real scenarios, learning tasks can not wait for arbitrarily long time or starve due to mutual exclusion resource requirement. The scheduling algorithm design should be starvation-free so that every task has a chance to run.

A system overview of *Hare* is shown in Fig. 9, where *Hare* is integrated into the existing PS-based distributed machine learning framework. *Hare* is not only a scheduling algorithm, but also a set of modules that optimize training processes across GPUs. It contains two main components: a logically centralized task scheduler, and executors running on training machines. All data are stored with HDFS [6]. The whole system running process contains two stages, an offline preparation stage and an online training stage. In the preparation stage, the task scheduler is fed by job information, e.g., job types, model description and training data size, from upper-layer applications. It also collects hardware information, e.g., GPU types, speed and memory, from the under-layer computing infrastructure. These information first goes to a module called profiler that trains a small piece of data to obtain expected task execution time on different GPUs, which will be the input of the task scheduling algorithm. We note that some jobs are usually repeatedly submitted to the training platform. For example, some models are periodically re-trained using latest collected datasets to adapt to emerging cases, which is particularly common in deep reinforcement learning. This observation motivates us to accelerate the profiling by maintaining a database that stores historical profiling results. We first search the database upon receiving job information. If corresponding results can be found, we skip profile training and directly feed searching results to the scheduling algorithm. We then run the scheduling algorithm (in Section 5) to generate a task running sequence for each GPU. Finally, these task sequences are sent to corresponding executors.

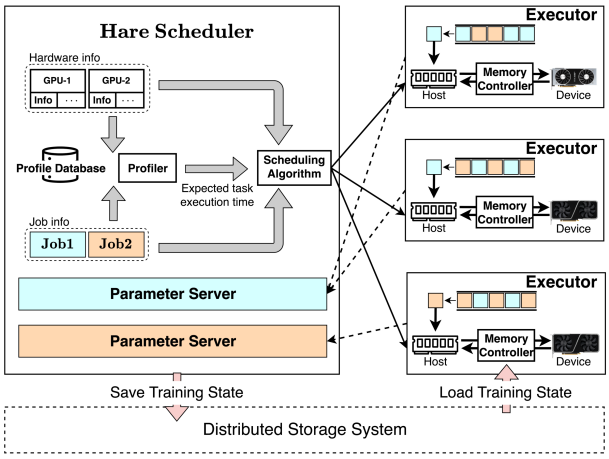

Figure 9: System Overview.

In the training stage, each executor schedules tasks and loads checkpoints on GPU according to their order given in the received task sequence. When a task completes, it sends updated gradients to the corresponding parameter server for aggregation. We follow the most of training designs in traditional distributed machine learning frameworks [26, 31], except the task switching mechanism. In existing works, since each job has exclusive use of assigned GPUs, several consecutive tasks on a GPU belongs to the same job and they share the same GPU context, leading to low switching overhead. In contrast, *Hare* allows GPU preemption by alternatively running tasks of different jobs, which involves frequent context switching with high overhead. Therefore, we design a fast task switching mechanism with a customized memory controller (in Section 4) to reduce this overhead.

4 FAST TASK SWITCHING

Given two tasks scheduled continuously on a GPU, a traditional task switching process contains two main steps. First, the predecessor needs to clean its GPU environment by freeing GPU memory occupations. Second, the successor initializes its environment by creating a new CUDA context, launching the process, allocating GPU memory and moving the model from main memory to GPU memory. Traditionally, the above two steps run sequentially, which incurs large overhead. Such overhead has been observed by [2, 8, 45] and our experimental results have also confirmed it. Therefore, reducing the task switching cost becomes a critical challenge that has to be addressed by *Hare*.

The fast task switching of *Hare* is mainly motivated by PipeSwtich [8]. To accelerate the model movement from main memory to GPU memory, PipeSwitch leverages the layered structure of neural networks and pipelines the model transmission and execution. Moreover, it finds that CUDA context creation is slow, and proposes to create multiple CUDA contexts in advance to hide the overhead of context creation during task switching. Other techniques, like NVIDIA Multiple Process Sharing (MPS) [2], allow multiple processes to share a single GPU, but they cannot be applied here

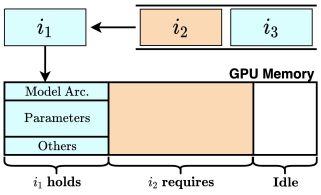

Figure 10: An example of speculative memory management.

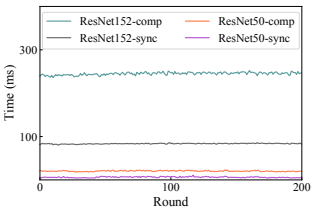

Figure 11: Training two popular models on 8 V100 GPUs.

because these processes need to be pre-loaded into GPU memory, which may exceed GPU memory limit.

Similar designs have been also adopted by PipeSwitch [8]. However, it is originally designed for maximizing GPU utilization by filling GPU idle time of an inference job with training or other inference jobs, and it misses many optimization chances in training job scheduling scenarios studied in this paper. To further reduce task switching cost, we propose the following two new designs by exploiting unique features of our task scheduling problem.

Early Task cleaning. When a task completes, PipeSwitch cleans its GPU environment by deleting GPU memory pointers only, leaving memory content unmodified, which may cause security issues [16, 44]. For example, an attacker can steal the private data by allocating the memory region the same as that used by the previous task. Instead, we propose early task cleaning that deletes intermediate data of each layer once its backward training completes. Early task cleaning has two benefits. First, we delete not only memory pointers but also memory content to avoid potential security concerns. Second, released GPU memory can be used for pre-loading data of the next task, so that it can start earlier.

Speculative Memory Management. PipeSwitch cleans a task by removing all its data stored in GPU memory. However, we find that it is not always necessary, especially when we know the tasks that will be scheduled on the same GPU. As an example shown in Fig. 10, we consider three tasks, i_1, i_2 and i_3, scheduled sequentially on a GPU. Tasks i_1 and i_3 belong to the same training round of a job, while i_2 is from a different job. We further suppose that three tasks do not occupy the whole GPU memory, which is common in practice [40]. In a traditional design, all data of i_1 are removed when it completes. An alternative method adopted by *Hare* is to keep the model data of task i_1, so that they can be re-used by i_3 scheduled later. Such a speculative memory management is feasible because *Hare* conducts an offline task scheduling and task running sequences as well as their GPU memory occupation can be known in advance. To decide which models and how long they can be kept in GPU memory, *Hare* uses a simple heuristic that always gives higher GPU memory priority to the next tasks, and greedily keeps models of latest completed tasks until they cannot be accommodated. Of course, we can formulate this memory management problem as an optimization problem and solve it to get the optimal solution. However, we find that the heuristic works sufficiently well in practice, and the resulted switching cost can be neglected.

Table 1: Notations

$\mathcal{N}$	set of training jobs	$\mathcal{M}$	set of heterogeneous GPUs
a_n	arrive time of job n	w_n	weight of job n
R_n	set of training rounds for job $n \in \mathcal{N}$		
D_r	set of parallel tasks in $r \in R_n$		
D	set of all tasks in $\mathcal{N}$		
$T^c_{i,m,r}$	training time of task $i \in \mathcal{N}$ on GPU m in the training round r		
$T^s_{i,m,r}$	synchronization time of task $i \in \mathcal{N}$ on GPU m in the training round r		
x_i	the start time of task $i \in \mathcal{N}$		
$y_{i,m}$	whether task $i \in \mathcal{N}$ is assigned to GPU m		
$\hat{x}_i$	the solution of x_i from the relaxed problem		
$\tilde{x}_i$	the solution of x_i from Algorithm 1		
$\tilde{y}_{i,m}$	the solution of $y_{i,m}$ from Algorithm 1		
$H_{i,m}$	the middle completion time of task $i \in \mathcal{N}$ on GPU m		
H_i	the maximum middle completion time of task $i \in \mathcal{N}$		
$\boldsymbol{\pi}$	a non-descending sequence according to H_i		
φ_m	current available time of GPU m		

5 TASK SCHEDULING ALGORITHM

In this section, we present the task scheduling algorithm, which is the core design of *Hare*. The system model is first presented, followed by algorithm details and theoretical performance analysis.

5.1 Problem Statement

We consider a set $\mathcal{N}$ of training jobs, running on a set $\mathcal{M}$ of heterogeneous GPUs. Each job $n \in \mathcal{N}$ consists of multiple training rounds, which are denoted by set R_n. Each job $n \in \mathcal{N}$ launches a set D_r of training tasks that can run in parallel in every training round, and each task is responsible for training a data batch. After local training, all tasks synchronize their gradients via the PS scheme to obtain an updated model for the next-round training. Moreover, we denote $\mathcal{D}$ as the set of all tasks in $\mathcal{N}$.

We let $T^c_{i,m,r}$ denote the training time of task i on GPU m in the training round r, and the corresponding synchronization time is denoted by $T^s_{i,m,r}$. As shown in Fig. 11, our experimental results about 2 popular models have shown that task training time and synchronization time is highly predictable and stable across training rounds. This fact allows us to use $T^c_{i,m}$ and $T^s_{i,m}$ by dropping the subscript r to simplify problem formulation. More importantly, it makes task scheduling with performance guarantee feasible.

Due to GPU heterogeneity, each task may have different training time on different GPUs. Similarly, it may have different synchronization time across GPUs because network condition changes. Besides, we assume that the training time is longer than the synchronization time. That is because GPUs are usually connected by high-speed networks (e.g., NVLink and InfiniBand) in data centers. Note that this is different from job-level non-preemption assumed in existing works [19, 31, 41], i.e., a job $n \in \mathcal{N}$ cannot be preempted once it starts to run on GPUs.

We consider the non-preemption setting for task running, i.e., a task's execution cannot be preempted once it is scheduled on a GPU. Thanks to the fast task switching mechanism, there is tiny task

switching cost, which is less than 5% of task training time according to experimental results. Therefore, we ignore the task switching cost in the problem formulation for simplicity. Note that our main theoretical results are still true even this cost is considered. We list the important notations in Table. 1

Each job $n \in \mathcal{N}$ is associated with arrive time a_n and a weight w_n. To formulate the problem, we define a variable x_i to denote the start running time of task i. In addition, a binary variable $y_{i,m}$ is defined to indicate the GPU assignment, i.e., $y_{i,m} = 1$ if task i is assigned to GPU m, and $y_{i,m} = 0$ otherwise. The completion time of job n is denoted by C_n. With the objective of minimizing the total weighted job completion time, we formulate the task scheduling problem as follows:

$$\textbf{Hare\_Sched:} \quad \min \sum_{n\in\mathcal{N}} w_n C_n, \quad \text{subject to:}$$

$$x_i \geq a_n, \forall i \in D_r, r \in R_n, n \in \mathcal{N}; \tag{4}$$

$$\sum_{m\in\mathcal{M}} y_{i,m} = 1, \forall i \in D_r, r \in R_n, n \in \mathcal{N}; \tag{5}$$

$$C_n \geq x_i + \sum_{m\in\mathcal{M}} y_{i,m}(T^c_{i,m} + T^s_{i,m}), \quad \forall i \in D_r, r \in R_n, n \in \mathcal{N}; \tag{6}$$

$$x_j \geq x_i + \sum_{m\in\mathcal{M}} y_{i,m}(T^c_{i,m} + T^s_{i,m}), \forall j \in D_{r+1}, i \in D_r, \quad r \in R_n, n \in \mathcal{N}; \tag{7}$$

$$|x_i - x_j| \geq y_{k,m} T^c_{k,m}, \forall i, j \in \mathcal{D}, m \in \mathcal{M}, \quad y_{i,m} = y_{j,m} = 1, k = \text{argmin}_{k=\{i,j\}}\{x_k\}. \tag{8}$$

Constraint (4) indicates that tasks of each job can not start before arrival. Each task can be assigned to at most one GPU, which is represented by (5). The job completion time is constrained by (6). Besides, due to the synchronized parameter update policy, the tasks of the $(r+1)-$th round must wait for the completion of all tasks of the $r-$th round, as shown in (7). Finally, we use constraint (8) to guarantee the non-preemption among tasks. Specifically, for any two tasks assigned to the same GPU, one cannot start before the completion of the other. The hardness of the above problem is given as follows.

Theorem 1. *Hare_Sched is NP-hard.*

Proof. The NP-hardness of Hare_Sched can be proved by reducing the well-known SS13[17] problem. The details are ignored due to length limit. □

5.2 Algorithm Design

By carefully examining the problem formulation, we find that the difficulty mainly stems from the non-linear constraint (8). Therefore, we are motivated to relax (8) and then design a heuristic algorithm based on the solution of the relaxed problem. This algorithm contains the following two steps and pseudo codes are shown in Algorithm 1.

Algorithm 1 Task Scheduling Algorithm in *Hare*

1: $\tilde{x}_i = 0, \tilde{y}_{i,m} = 0, \forall i \in \mathcal{D}, m \in \mathcal{M}$;
2: $\varphi_m = 0, \forall m \in \mathcal{M}$;
3: Solve the Hare_Sched_RL problem to obtain solutions $\hat{x}_i$ as well as H_i;
4: Sort tasks to generate a sequence π satisfying $H_{\pi(1)} \leq H_{\pi(2)} \leq ... \leq H_{\pi(|\mathcal{D}|)}$;
5: **for** $i \in \{\pi(1), \pi(2), ..., \pi(|\mathcal{D}|)\}$ **do**
6: Identify the corresponding job n and r where $i \in D_r, r \in R_n$;
7: **if** $r = 0$ **then**
8: $t_i = a_n$;
9: **else**
10: $t_i = \max_{j\in D_{r-1}} \{\tilde{x}_j + \tilde{T}^c_j + \tilde{T}^s_j\}$;
11: **end if**
12: Find $m^* = \text{argmin}_{m\in\mathcal{M}} \varphi_m$;
13: $\tilde{x}_i = \max\{t_i, \varphi_{m^*}\}$;
14: $\tilde{y}_{i,m^*} = 1$;
15: $\tilde{T}^c_i = \tilde{y}_{i,m^*} T^c_{i,m^*}, \tilde{T}^s_i = \tilde{y}_{i,m^*} T^s_{i,m^*}$;
16: $\varphi_{m^*} = \tilde{x}_i + \tilde{T}^c_i$;
17: **end for**
18: **return** $\tilde{x}, \tilde{y}$

Step 1: Problem Relaxation. We relax the original planning problem as follows:

$$\textbf{Hare\_Sched\_RL:} \quad \min \sum_{n\in\mathcal{N}} w_n C_n, \quad \text{subject to:}$$

$$\sum_{i\in\mathcal{D}} y_{i,m} T^c_{i,m}(x_i + T^c_{i,m}) \geq \frac{1}{2}\left[\Big(\sum_{i\in\mathcal{D}} y_{i,m} T^c_{i,m}\Big)^2 + \sum_{i\in\mathcal{D}} \Big(y_{i,m} T^c_{i,m}\Big)^2\right], \forall m \in \mathcal{M}; \tag{9}$$

$$(4) - (7).$$

Constraint (9) is the relaxation of (8) according to [33]. Note that (9) is independent of the running order of tasks scheduled on each GPU and it always holds for any feasible scheduling. Thus, Hare_Sched_RL serves as a lower bound of Hare_Sched. Although Hare_Sched_RL is a mixed-integer quadratic programming that is still with high theoretical complexity, we are fortunate to have fast solvers, e.g., CPLEX and Gurobi, which have been well optimized and work well in practice.

Step 2: Task Scheduling. The solution of Hare_Sched_RL is denoted by $\hat{x}_i$. The middle completion time of task i on GPU m can be calculated by $H_{i,m} = \hat{x}_i + \frac{1}{2} T^c_{i,m}$. Therefore, the maximum middle completion time of task i can be denoted by $H_i = \max_m\{H_{i,m}\}$.

We then sort tasks in a non-descending order according to H_i to generate a sequence $\boldsymbol{\pi}$, as shown in line 4. The j-th task in the sequence $\boldsymbol{\pi}$ is denoted by $\pi(j)$. Next, we iteratively schedule tasks according to the sequence π in the **for** loop in lines 5-17. Specifically, in each iteration, we deal with the task i by first identifying its associated job n and training round r, which is easy in practice by attaching such information in the task description. In the following line 7, we continue to check whether this task belongs to the first training round. If it is, i.e., $r = 0$, this task can logically start upon

arrival. We let t_i denote the task available time. In this case, we have $t_i = a_n$ as shown in line 8, where a_n is the arrival time of job n containing task i. Otherwise, task i needs to wait the completion of all tasks in the previous training round, and its available time t_i can be calculated in line 10. Note that $\tilde{T}_j^c$ and $\tilde{T}_j^s$ in line 10 is the real training and synchronization time of tasks in the $(r-1)$-th round and they are updated in line 15 in the previous algorithm iterations.

Up to now, we consider only task available time, which may not be identical to its real start time in practice because we haven't decided which GPU to run this task. We let φ_m denote the current GPU available time. Next, we will check the GPU availability and find a GPU where the task can be assigned. We adopt a greedy strategy, which always assigns the task to the GPU m^* with the earliest available time, as shown in line 12. After that, we can update the real task start time, denoted by $\tilde{x}_i$, as well as the task assignment $\tilde{y}_{i,m^*}$. The real task training time and synchronization time is updated in line 15. Finally, we update the GPU available time φ_{m^*} in line 16. Note that the synchronization time T_{i,m^*}^s is not considered in line 16 because the communication can overlap with the next task assigned on this GPU.

5.3 Theoretical Analysis

Before deriving the approximation ratio of the proposed algorithm, we prove the following two lemmas. Due to the space limitation, we put the complete proof in our technical report [7].

Lemma 2. *For any task $\pi(j) \in \boldsymbol{\pi}$ on GPU m, we have:*

$$\sum_{k=1}^{j} \tilde{y}_{\pi(k),m} T_{\pi(k),m}^c \le 2H_{\pi(j)}, \tag{10}$$

where $\tilde{y}_{\pi(k),m}$ indicates the GPU assignment by Algorithm 1.

Proof. For any task $\pi(j)$ as well as its predecessors $\{\pi(1), \pi(2), ..., \pi(j-1)\}$ in the sequence $\boldsymbol{\pi}$, constraint (9) always holds and we have:

$$\sum_{k=1}^{j} \tilde{y}_{\pi(k),m} T_{\pi(k),m}^c (\tilde{x}_{\pi(k)} + T_{\pi(k),m}^c) \ge \frac{1}{2}\Big[\Big(\sum_{k=1}^{j} \tilde{y}_{\pi(k),m} T_{\pi(k),m}^c\Big)^2 + \sum_{k=1}^{j} \big(\tilde{y}_{\pi(k),m} T_{\pi(k),m}^c\big)^2\Big]. \tag{11}$$

By substituting $H_{i,m} = \hat{x}_i + \frac{1}{2} T_{i,m}^c$ and eliminating $\frac{1}{2} \sum_{k=1}^{j} (\tilde{y}_{\pi(k),m} T_{\pi(k),m}^c)^2$ in the right side of (11), we can obtain:

$$\sum_{k=1}^{j} \tilde{y}_{\pi(k),m} T_{\pi(k),m}^c H_{\pi(k),m} \ge \frac{1}{2}\Big(\sum_{k=1}^{j} \tilde{y}_{\pi(k),m} T_{\pi(k),m}^c\Big)^2. \tag{12}$$

Because of $H_{\pi(k)} = \max_m\{H_{\pi(k),m}\}$ and $H_{\pi(1)} \le H_{\pi(2)} \le ... \le H_{\pi(j)}$, we have:

$$H_{\pi(j)} \sum_{k=1}^{j} \tilde{y}_{\pi(k),m} T_{\pi(k),m}^c \ge \frac{1}{2}\Big(\sum_{k=1}^{j} \tilde{y}_{\pi(k),m} T_{\pi(k),m}^c\Big)^2. \tag{13}$$

Canceling out $\sum_{k=1}^{j} \tilde{y}_{\pi(k),m} T_{\pi(k),m}^c$ in both sides of (13) leads to (10). □

Lemma 3. *We let π_m denote the task sequence on GPU m returned by Algorithm 1. The total idle time before task $\pi_m(j)$ on GPU m is $\delta(\pi_m(j), m)$, which satisfies:*

$$\delta(\pi_m(j), m) \le \alpha H_{\pi_m(j)}, \tag{14}$$

where $\alpha = \max_{i \in \mathcal{D}}\{T_i^{c,max}/T_i^{c,min}, T_i^{s,max}/T_i^{s,min}\}$.

Proof. Without loss of generality, we consider two tasks $\pi_m(j-1)$ and $\pi_m(j)$, which are not continuously scheduled on GPU m. Suppose task $\pi_m(j)$ belongs to the job n. Let us first check how such a case happens. According to Algorithm 1, when we schedule task $\pi_m(j)$ according to the task sequence π, GPU m has the earliest available time. However, task $\pi_m(j)$ cannot start immediately after $\pi_m(j-1)$ because of the synchronization barrier, i.e., there must exist some tasks, which belong to previous rounds of $\pi_m(j)$, running on other GPUs before the start of task $\pi_m(j)$. Otherwise, there would be no GPU idle time between $\pi_m(j-1)$ and $\pi_m(j)$. Note that there may be multiple training rounds of job n between tasks $\pi_m(j-1)$ and $\pi_m(j)$. In each round, there must be a bottleneck task whose running time is the same with duration of this round. Otherwise, some tasks would be scheduled on GPU m between tasks $\pi_m(j-1)$ and $\pi_m(j)$. We denote these bottleneck tasks as $\langle u(1), u(2), ..., u(l)\rangle$. We have the following relationship between tasks $\pi_m(j-1)$ and $u(0)$.

$$\tilde{x}_{u(0)} \le \tilde{x}_{\pi_m(j-1)} + \tilde{T}_{\pi_m(j-1)}^c; \tag{15}$$

$$H_{u(0)} \ge H_{\pi_m(j-1)}. \tag{16}$$

where $u(0)$ is a task that belongs to the previous round of $u(1)$ and satisfies $\tilde{x}_{u(0)} + \tilde{T}_{u(0)}^c + \tilde{T}_{u(0)}^s = \tilde{x}_{u(1)}$. By introducing $\alpha = \max_{i \in \mathcal{D}}\{T_i^{c,max}/T_i^{c,min}, T_i^{s,max}/T_i^{s,min}\}$ and adopting constraint (7), we can obtain:

$$\alpha(H_{\pi_m(j)} - H_{u(0)}) \ge \sum_{i=0}^{l} (T_{u(i)}^{c,max} + T_{u(i)}^{s,max}). \tag{17}$$

According to the property of the results $\{\tilde{x}_i\}$ returned by the Algorithm 1:

$$\tilde{x}_{\pi_m(j)} - \tilde{x}_{u(l)} = \tilde{T}_{u(l)}^c + \tilde{T}_{u(l)}^s; \tag{18}$$

$$\tilde{x}_{u(i)} - \tilde{x}_{u(i-1)} = \tilde{T}_{u(i-1)}^c + \tilde{T}_{u(i-1)}^s, \forall i = 1, 2, ..., l. \tag{19}$$

we final proof that:

$$\delta(\pi_m(j)) \le \alpha(H_{\pi_m(j)} - H_{\pi_m(0)}) = \alpha H_{\pi_m(j)}. \tag{20}$$

□

Theorem 4. *Algorithm 1 is $\alpha(2+\alpha)$-approximation.*

Proof. Since our algorithm always schedules tasks on machines with the earliest start time, we have:

$$\tilde{x}_{\pi(j)} \le \sum_{k=1}^{j-1} \tilde{y}_{\pi(k),m} T_{\pi(k),m}^c + \delta(\pi(j), m), \forall m \in \mathcal{M}. \tag{21}$$

By combining with the result of adding (10) and (14), we obtain:

$$\tilde{x}_{\pi(j)} + \tilde{T}^{c}_{\pi(j)} \leq (2+\alpha) H_{\pi(j)}. \tag{22}$$

We can proof Theorem 4 by taking scaling methods in (22).

□

6 IMPLEMENTATION

We have implemented a prototype of *Hare* by using roughly 2500 LoC of Python and C++. We use PyTorch 1.8.1 for DML job training. *Hare* primary maintains two components: a central scheduler and executors. The scheduler communicates with executors via controlling messages implemented using gRPC APIs [5].

Scheduler. The scheduler integrates a task profiler, a scheduling algorithm, and parameter servers. The scheduler first executes `task_profile()`, fed by job information, to predict task execution time. The task scheduling is executed to obtain the task sequence for each executor. According to job information, the scheduler instantiates a series of `Hare_Parameter_Server` to bind to each DML job for gradient synchronization. `Hare_Parameter_Server` saves the checkpoint of DML job by using PyTorch interface `save()` Also, the scheduler maintains a gRPC module to communicate the task sequence and gradients with executors.

Executor. The executor initializes several trainer processes to train tasks in the sequence received from the scheduler. In our implementation, we initialize three trainer processes in the executor. To hide CUDA context creation cost, we create a CUDA context for each process in advance by calling an implicit initialization `torch.randn(10, device='cuda')`. When a task needs to be trained, we assign it to a process (called the working process) and leave the rest on standby. Each working process initializes the task model locally and loads the checkpoint from storage by using PyTorch interface `load()`. Note that the model structure is small so that we can save it locally. We add hooks to the model to enable pipelined model transmission. Specifically, we use PyTorch interface `register_forward_pre_hook()` to change the `initialized` property of components (e.g., `torch.nn.Linear()`) in each layer. After that, the executor starts the training task and sends gradients to the parameter server using PyTorch and gRPC interfaces. We also add hooks to change the property `retain_graph` of all tensors in each layer, which aims to support early task cleaning. In native PyTorch, the gradients of intermediate variables are deleted. To avoid the deletion, we modify the gradients free mechanism in PyTorch. Moreover, we keep the model data in GPU according to switching algorithm results.

7 PERFORMANCE EVALUATION

In this section, we first introduce our experimental settings and then present the results of the testbed and simulations.

7.1 Experimental settings

Testbed. We build a testbed consisting of 15 heterogeneous GPUs (8 V100s, 4 T4s, 1 K80, and 2 M60s), which are deployed on 4 Amazon EC2 instances. All GPUs are equipped with PCIe-3×16 (15.75 GB/s). Each instance is powered by NVIDIA driver 418.21, CUDA 10.1 and cuDNN 8.0.4, running Ubuntu 18.04 with Linux kernel version 5.4. All instances are connected via the 25 Gbps Ethernet.

Table 2: Deep Learning Jobs Used in Our Experiments.

Type	Model	Dataset	BatchSize	
CV	VGG-19 [37]	Cifar10 [23]	128	25%
CV	ResNet50 [22]	Cifar100 [27]	64	
CV	Inception V3 [38]	Cifar100 [27]	32	
NLP	Bert_base [15]	SQuAD [34]	32	25%
NLP	Transformer [39]	WMT16 [1]	128	
Speech	DeepSpeech [21]	ComVoice [4]	8	25%
Rec.	FastGCN [11]	Cora [35]	128	25%
Rec.	GraphSAGE [20]	Cora [35]	16	

Simulator. We have developed a trace-driven simulator to evaluate *Hare* in large-scale settings. The simulator is built in Python, and emulates the execution of DML jobs using the traces collected from the testbed. The job arrival time is set according to the trace in Google cluster [3].

Workload. We create some DML jobs based on 8 popular models across domains of computer vision (CV), natural language processing (NLP), speech, and recognition (Rec.). The details of these models are shown in Table 2. In the default setting, each type of jobs accounts for 25% of the total workload. All jobs are implemented in PyTorch 1.8.1, and they are trained using synchronous PS scheme. Since the original datasets of SQuAD and WMT16 are too large and the corresponding training would run for days, we downscale them so that they can complete within hours.

Schemes for Comparison. We compare *Hare* with following schemes.

Gavel_FIFO: FIFO (First In First Out) is a default job scheduling algorithm in many traditional batch job processing systems [46]. It schedules jobs in an order according to their arrival time. Gavel [29] customizes FIFO for heterogeneous GPUs by assigning jobs to fastest available GPUs. If the number of idle GPUs is insufficient, this job needs to wait until demanded GPUs are available.

SRTF (Shortest Remaining Time First): SRTF has been widely adopted to minimize total job completion time. It always schedule jobs that could complete earlier.

Sched_Homo [47]: We denote a recent scheduling algorithm [47] designed for homogeneous GPUs by Sched_Homo. Similar to *Hare*, it aims to minimize the weighted ML job completion time by exploiting both inter-job and intra-job parallelism. However, job-level preemption is not allowed.

Sched_Allox [24]: We consider the ML job scheduling algorithm proposed by Allox [24]. The GPU heterogeneity has been fully exploited, but it does not consider the intra-job parallelism.

7.2 Resutls on Testbed

We first study the benefits of fast task switching by showing the average switching time of different jobs in Table 3. A default task switching scheme, without any optimization, needs more than 3000ms for all jobs. PipeSwitch can reduce the average switching time to 12.57ms for Bert_base and less for others. The maximum switching time of *Hare* is no more than 6ms. The proportion of task switching time to the total task time is also shown in the table. We can see that *Hare* constrains the task switching overhead within

Table 3: Average Task Switching Time of Different Jobs.

	VGG19	ResNet50	Inception V3	Bert_base	Transformer	DeepSpeech	FastGCN	GraphSAGE
Default	3288.94 ms (98.21%)	5961.16ms (97.37%)	7807.43 ms (96.99%)	9016.99 ms (93.95%)	5257.17 ms (95.41%)	5125.64 ms (94.15%)	5327.24 ms (98.47%)	5213.54 ms (98.29%)
PipeSwitch[8]	4.01 ms (2.40%)	4.75 ms (5.46%)	5.03 ms (2.39%)	12.57 ms (1.99%)	10.34 ms (2.03%)	8.91 ms (1.59%)	2.86 ms (7.56%)	2.42 ms (8.64%)
Hare	2.77 ms (1.82%)	2.04 ms (3.71%)	2.46 ms (1.43%)	5.03 ms (1.13%)	5.79 ms (1.36%)	4.27 ms (1.25%)	1.83 ms (4.53%)	0.96 ms (3.36%)

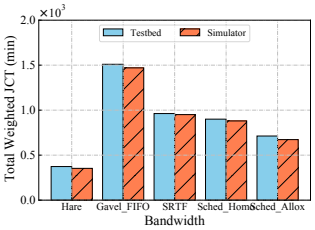

Figure 12: The results in testbed.

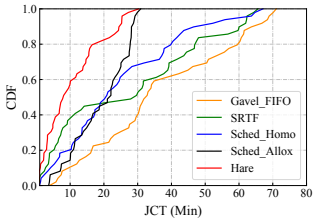

Figure 13: CDF of job completion time.

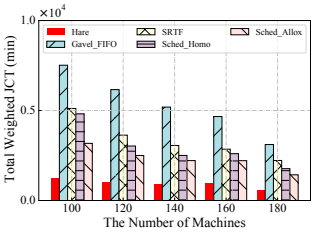

Figure 14: Performance under different number of GPUs.

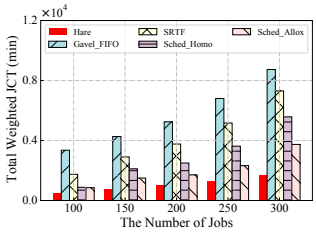

Figure 15: Performance under different number of jobs.

2% for most of models, and the largest overhead under FastGCN is no more than 5%. These results justify our assumption that task switching time is negligible in the scheduling algorithm design.

The total weighted job completion time (JCT) of several schemes running on the testbed and the simulator is shown in Fig. 12. Compared with other schemes, *Hare* can reduce total weighted JCT by 47.6% to 75.3%, significantly outperforming other schemes. Fig. 13 shows the cumulative distributed function (CDF) of JCT of all jobs. We can see that about 90.5% of jobs can complete within 25 minutes by *Hare*, while Sched_Allox and Sched_Homo can complete only 66.7% and 56.5%, respectively. That is because Allox misses the optimization chances brought by intra-job parallelism, and Sched_Homo is GPU-heterogeneity-oblivious, leading to low GPU utilization.

7.3 Results of Simulations

Large-scale experiments are conducted using the simulator. As we have shown in Fig. 12, the maximum performance gap between the testbed and simulator is only 5%, which demonstrates that the simulator can offer sufficient simulation accuracy. The gap is mainly because the error in prediction of training time and switching cost.

We study the influence of number of GPUs in Fig. 14. The number of ML jobs is set to 200. The weighted JCT of all schemes decreases as more GPUs are used. *Hare* always outperforms other schemes under all cases. Sched_Allox is slower than *Hare* by about 2x, but it is still significantly faster than others, thanks to its heterogeneity-aware design. Although Gavel_FIFO schedules jobs with the consideration of heterogeneity, it still has the largest weighted JCT since it has no optimization in scheduling.

We then consider 160 GPUs and change the number of jobs from 100 to 300 to see how it affects the performance. As shown in Fig. 15, as the number of jobs increases, the total weighted JCT grows under all schemes. Meanwhile, the performance gaps between *Hare* and other schemes become bigger. For example, *Hare* outperforms others by 54.6%-80.5% when processing 300 jobs. It demonstrates that *Hare* can use these GPUs in a more efficient way, to minimize the total weighted JCT.

We study the influence of GPU heterogeneity in Fig. 16. We consider 160 GPUs and 200 jobs. We set different heterogeneity levels by selecting a different combination of GPUs. For the low heterogeneity level, we only choose V100 GPUs for training. We select the combination of (V100×K80) GPUs as the middle heterogeneity level while selecting the combination of (V100×T4×K80×M60) GPUs as the high heterogeneity level. We find the gaps between *Hare* and other schemes become bigger as the increasing of heterogeneity level. The main reason is the higher heterogeneity level results in lower resource utilization in heterogeneity-oblivious schemes. Although Sched_Allox suffers a slight influence from the heterogeneity level, its performance still lags behind *Hare* by 2× since there is no consideration of intra-job parallelism optimization. We also find that *Hare* and Sched_Homo have close performance when there is a low-level heterogeneity, because intra-job parallelism optimization has the dominant influence in such scenarios.

We investigate how job type affects the performance by changing their proportions. The results are shown in Fig. 17. In the default setting, each type of jobs account for 25%. In each experiment, we then increase one of them and keep others the same. The x axis of Fig. 17 shows the ratio of different job types. When we increase the proportion of NLP jobs, the total weighted JCT of all schemes increases since NLP jobs involve heavier training workloads (i.e., more training rounds and more training time). On the other hand, all schemes have smaller weighted JCT when more recognition jobs are added, because they have less workloads. Although *Hare* is affected by the job proportion, it always achieves the best performance due to the sophisticated scheduling algorithm.

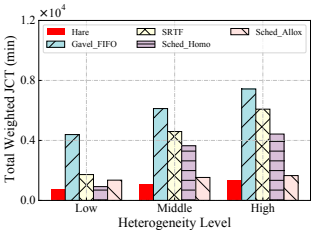

Figure 16: Performance under different heterogeneity levels.

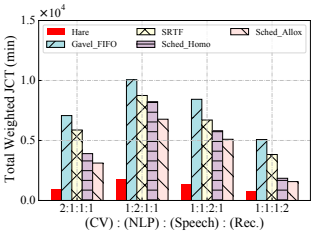

Figure 17: Performance under different fractions of jobs.

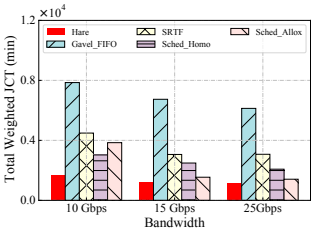

Figure 18: Performance under different bandwidth.

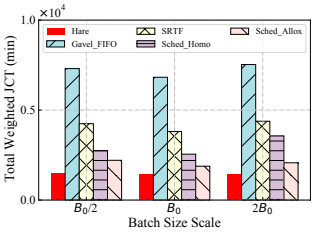

Figure 19: Performance under different batch sizes.

We change the speed of the network connecting GPUs and study its influence in Fig. 18. The results are in alignment with our intuition that faster networks can accelerate the ML training. However, such acceleration is not linear with the network speed since the training time becomes the main bottleneck as the decreasing of the synchronization time. For example, *Hare*'s weighted JCT decreases by only 31.2%, even though increase the network speed from 10Gbps to 25Gbps.

Fig. 19 shows the performance under different batch sizes, where B_0 stands for the default batch size configuration. We can see that batch size has no big influence to all schemes except Sched_Homo. That is because larger batch size leads to longer training time, and there is more GPU idle time in Sched_Homo.

8 RELATED WORK

Distributed Machine Learning. Distributed machine learning on GPUs has been widely adopted to accelerate model training on large datasets. Typically, We can assign and synchronize workloads on GPUs in two different ways, which are referred to as model parallelism [13, 28] and data parallelism [12, 25, 36]. In the model parallelism, each GPU trains a partition of the model with the entire dataset. In data parallelism, each GPU maintains a complete model and trains it using a subset of data. The model gradients are periodically synchronized across GPUs using All-Reduce [15, 18, 30] or Parameter Server (PS) [10, 14, 25, 43] scheme. In particular, the PS scheme is popular due to its simplicity, and we also use it in our work. Specifically, the training process contains multiple rounds. In each round, training workloads are shared by multiple GPUs, which are also called workers. Each worker computes its local gradients by using mini-batch stochastic gradient descent (SGD) method. Then they send gradients to the parameter server, which updates the model for the next-round training.

Job scheduling for machine learning. Job scheduling, which determines when and where each job should run, is the most fundamental and critical issue for distributed machine learning. Early studies follow the idea of traditional batch job scheduling by treating each job as an unsplittable unit and schedule them on different GPUs [46]. Later, some works have exploited the intra-job parallelism, i.e., tasks in the same training round of a job can run in parallel, which can significantly enhance learning performance. Optimus [31] allocates resources to ML jobs by learning a throughput model with respect to various resource allocation. Themis [26] introduces a notation of finish-time fairness to promote fair allocation, while improving the cluster utilization. Pollux [32] studies different resource allocation for ML jobs by observing the throughput and statistical efficiency during training. Zhang et al. [47] design an online algorithm that selects the amount of resources for each job to minimize the total job completion time. Although the above works have exploited both inter-job and intra-job parallelism, they consider homogeneous GPUs and forbid GPU preemption during job execution.

Recently, GPU-heterogeneity becomes popular as the expansion of data centers and it has attracted significant research attention. $\text{Gandiva}_{\text{fair}}$ [9] proposes an automated trading mechanism to support time-slicing resource sharing among different jobs while improving the cluster efficiency. Gavel [29] develops a heterogeneity-aware scheduler to generate different scheduling policies for different kinds of jobs. However, $\text{Gandiva}_{\text{fair}}$ and Gavel schedule jobs based on given time slice length. Such a coarse-grained scheduling manner leaves a large optimization space for performance improvement. Moreover, they ignore the task switching cost. Allox [24] transforms the job scheduling problem into a min-cost bipartite matching to provide dynamic fair allocation, but it conducts job-level scheduling and ignores the intra-job parallelism.

9 CONCLUSION

We present *Hare*, a system enabling efficient multiple DML job scheduling on heterogeneous GPU cluster. Considering frequent task switching may happen in *Hare*, we propose fast task switching optimization in *Hare* to reduce the overhead of task switching. Besides, we propose a heterogeneity-aware task scheduling algorithm to minimize the total weighted job completion time. We demonstrate the performance of *Hare* through experiments on both small-scale testbed and large-scale trace-driven simulator. *Hare* can significantly outperform existing works.

10 ACKNOWLEDGEMENTS

This research was supported by JSPS KAKENHI (No. 21H03424), the Key-Area Research and Development Program of Guangdong Province (No. 2021B0101400003), Hong Kong RGC Research Impact Fund (No. R5060-19), General Research Fund (No. 152221/19E, 152203/20E, and 152244/21E), the National Natural Science Foundation of China (61872310), and Shenzhen Science and Technology Innovation Commission (JCYJ20200109142008673). Peng Li is the corresponding author.

REFERENCES

[1] 2016. Wmt16 dataset. http://www.statmt.org/wmt16/

[2] 2019. CUDA Multi-Process Service. https://docs.nvidia.com/deploy/pdf/\CUDA_Multi_Process_Service_Overview.pdf

[3] 2019. Google Cluster Traces. http://github.com/google/cluster-data

[4] 2020. Common voice dataset. https://voice.mozilla.org/

[5] 2020. gRPC. https://grpc.io

[6] 2021. Apache Hadoop. http://hadoop.apache.org/.

[7] 2021. Technical Report. https://www.dropbox.com/s/vkdip7xbhh5ca06/Technical_report.pdf?dl=0

[8] Zhihao Bai, Zhen Zhang, Yibo Zhu, and Xin Jin. 2020. PipeSwitch: Fast Pipelined Context Switching for Deep Learning Applications. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. 499–514.

[9] Shubham Chaudhary, Ramachandran Ramjee, Muthian Sivathanu, Nipun Kwatra, and Srinidhi Viswanatha. 2020. Balancing efficiency and fairness in heterogeneous GPU clusters for deep learning. In *Proceedings of the Fifteenth European Conference on Computer Systems*. 1–16.

[10] Fahao Chen, Peng Li, Toshiaki Miyazaki, and Celimuge Wu. 2021. FedGraph: Federated Graph Learning With Intelligent Sampling. *IEEE Transactions on Parallel and Distributed Systems* 33, 8 (2021), 1775–1786.

[11] Jie Chen, Tengfei Ma, and Cao Xiao. 2018. FastGCN: Fast Learning with Graph Convolutional Networks via Importance Sampling. In *6th International Conference on Learning Representations, ICLR Conference Track Proceedings*. https://openreview.net/forum?id=rytstxWAW

[12] Tianqi Chen, Mu Li, Yutian Li, Min Lin, Naiyan Wang, Minjie Wang, Tianjun Xiao, Bing Xu, Chiyuan Zhang, and Zheng Zhang. 2016. Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems. *NIPS Workshop on Machine Learning Systems (LearningSys)* (2016).

[13] Trishul Chilimbi, Yutaka Suzue, Johnson Apacible, and Karthik Kalyanaraman. 2014. Project adam: Building an efficient and scalable deep learning training system. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*. 571–582.

[14] Wei Dai, Abhimanu Kumar, Jinliang Wei, Qirong Ho, Garth Gibson, and Eric Xing. 2015. High-performance distributed ML at scale through parameter server consistency models. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 29.

[15] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. In *NAACL*.

[16] Hongyu Fang, Miloš Doroslovački, and Guru Venkataramani. 2019. Eraseme: A defense mechanism against information leakage exploiting gpu memory. In *Proceedings of the 2019 on Great Lakes Symposium on VLSI*. 319–322.

[17] Michael R Garey and David S Johnson. 1979. Computers and intractability: A Guide to the theory of np-completeness. (1979).

[18] Andrew Gibiansky. 2017. Bringing HPC techniques to deep learning. https://andrew.gibiansky.com/blog/machine-learning/baidu-allreduce/

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

[20] William L. Hamilton, Zhitao Ying, and Jure Leskovec. 2017. Inductive Representation Learning on Large Graphs. In *Neural Information Processing Systems (NIPS)*. 1024–1034.

[21] Awni Hannun, Carl Case, Jared Casper, Bryan Catanzaro, Greg Diamos, Erich Elsen, Ryan Prenger, Sanjeev Satheesh, Shubho Sengupta, Adam Coates, et al. 2014. Deep speech: Scaling up end-to-end speech recognition. *arXiv preprint arXiv:1412.5567* (2014).

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

[23] Alex Krizhevsky, Geoffrey Hinton, et al. 2009. Learning multiple layers of features from tiny images. (2009).

[24] Tan N Le, Xiao Sun, Mosharaf Chowdhury, and Zhenhua Liu. 2020. AlloX: compute allocation in hybrid clusters. In *Proceedings of the Fifteenth European Conference on Computer Systems*. 1–16.

[25] Mu Li, David G Andersen, Jun Woo Park, Alexander J Smola, Amr Ahmed, Vanja Josifovski, James Long, Eugene J Shekita, and Bor-Yiing Su. 2014. Scaling distributed machine learning with the parameter server. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*. 583–598.

[26] Kshiteej Mahajan, Arjun Balasubramanian, Arjun Singhvi, Shivaram Venkataraman, Aditya Akella, Amar Phanishayee, and Shuchi Chawla. 2020. Themis: Fair and efficient GPU cluster scheduling. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*. 289–304.

[27] David Martin, Charless Fowlkes, Doron Tal, and Jitendra Malik. 2001. A database of human segmented natural images and its application to evaluating segmentation algorithms and measuring ecological statistics. In *Proceedings Eighth IEEE International Conference on Computer Vision (ICCV)*, Vol. 2. 416–423.

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

[29] Deepak Narayanan, Keshav Santhanam, Fiodar Kazhamiaka, Amar Phanishayee, and Matei Zaharia. 2020. Heterogeneity-Aware Cluster Scheduling Policies for Deep Learning Workloads. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. 481–498.

[30] Pitch Patarasuk and Xin Yuan. 2009. Bandwidth optimal all-reduce algorithms for clusters of workstations. *J. Parallel and Distrib. Comput.* 69, 2 (2009), 117–124.

[31] Yanghua Peng, Yixin Bao, Yangrui Chen, Chuan Wu, and Chuanxiong Guo. 2018. Optimus: an efficient dynamic resource scheduler for deep learning clusters. In *Proceedings of the Thirteenth European Conference on Computer Systems*. 1–14.

[32] Aurick Qiao, Sang Keun Choe, Suhas Jayaram Subramanya, Willie Neiswanger, Qirong Ho, Hao Zhang, Gregory R Ganger, and Eric P Xing. 2021. Pollux: Co-adaptive Cluster Scheduling for Goodput-Optimized Deep Learning. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*.

[33] Maurice Queyranne. 1993. Structure of a simple scheduling polyhedron. *Mathematical Programming* 58, 1 (1993), 263–285.

[34] Pranav Rajpurkar, Jian Zhang, Konstantin Lopyrev, and Percy Liang. 2016. SQuAD: 100,000+ Questions for Machine Comprehension of Text. In *Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing*. 2383–2392. https://www.aclweb.org/anthology/D16-1264

[35] Prithviraj Sen, Galileo Namata, Mustafa Bilgic, Lise Getoor, Brian Galligher, and Tina Eliassi-Rad. 2008. Collective classification in network data. *AI magazine* 29, 3 (2008), 93–93.

[36] Alexander Sergeev and Mike Del Balso. 2018. Horovod: fast and easy distributed deep learning in TensorFlow. *arXiv preprint arXiv:1802.05799* (2018).

[37] Karen Simonyan and Andrew Zisserman. 2015. Very Deep Convolutional Networks for Large-Scale Image Recognition. In *3rd International Conference on Learning Representations, ICLR Conference Track Proceedings*. http://arxiv.org/abs/1409.1556

[38] Christian Szegedy, Vincent Vanhoucke, Sergey Ioffe, Jon Shlens, and Zbigniew Wojna. 2016. Rethinking the inception architecture for computer vision. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2818–2826.

[39] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. In *Advances in neural information processing systems*. 5998–6008.

[40] Linnan Wang, Jinmian Ye, Yiyang Zhao, Wei Wu, Ang Li, Shuaiwen Leon Song, Zenglin Xu, and Tim Kraska. 2018. Superneurons: Dynamic GPU memory management for training deep neural networks. In *Proceedings of the 23rd ACM SIGPLAN symposium on principles and practice of parallel programming*. 41–53.

[41] Wencong Xiao, Romil Bhardwaj, Ramachandran Ramjee, Muthian Sivathanu, Nipun Kwatra, Zhenhua Han, Pratyush Patel, Xuan Peng, Hanyu Zhao, Quanlu Zhang, et al. 2018. Gandiva: Introspective cluster scheduling for deep learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. 595–610.

[42] Wencong Xiao, Shiru Ren, Yong Li, Yang Zhang, Pengyang Hou, Zhi Li, Yihui Feng, Wei Lin, and Yangqing Jia. 2020. AntMan: Dynamic Scaling on GPU Clusters for Deep Learning. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. 533–548.

[43] Eric P Xing, Qirong Ho, Wei Dai, Jin Kyu Kim, Jinliang Wei, Seunghak Lee, Xun Zheng, Pengtao Xie, Abhimanu Kumar, and Yaoliang Yu. 2015. Petuum: A new platform for distributed machine learning on big data. *IEEE transactions on Big Data* 1, 2 (2015), 49–67.

[44] Yaswanth Yadlapalli, Husheng Zhou, Yuqun Zhang, and Cong Liu. 2021. gGuard: Enabling Leakage-Resilient Memory Isolation in GPU-accelerated Autonomous Embedded Systems. In *2021 58th ACM/IEEE Design Automation Conference (DAC)*. IEEE, 817–822.

[45] Peifeng Yu and Mosharaf Chowdhury. 2019. Salus: Fine-grained gpu sharing primitives for deep learning applications. *arXiv preprint arXiv:1902.04610* (2019).

[46] Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker, Ion Stoica, et al. 2010. Spark: Cluster computing with working sets. *USENIX HotCloud* 10, 10-10 (2010), 95.

[47] Qin Zhang, Ruiting Zhou, Chuan Wu, Lei Jiao, and Zongpeng Li. 2020. Online scheduling of heterogeneous distributed machine learning jobs. In *Proceedings of the Twenty-First International Symposium on Theory, Algorithmic Foundations, and Protocol Design for Mobile Networks and Mobile Computing*. 111–120.