Hops: Fine-grained heterogeneous sensing, efficient and fair Deep Learning cluster scheduling system

Qinghe Wang
School of Artificial Intelligence,
Anhui University
Hefei, Anhui, China
3167360788@qq.com

Futian Wang
Anhui University
Hefei, Anhui, China
wft@ahu.edu.cn

Xinwei Zheng*
Institute of Artificial Intelligence,
Hefei Comprehensive National
Science Center
Hefei, Anhui, China
xwzheng@ustc.edu.cn

ABSTRACT

In recent years, the number of clusters and cloud platforms dedicated to deep learning acceleration has increased, and research on multi-tenant deep learning (DL) cluster scheduling systems has also advanced quickly. However, we have observed several shortcomings in these systems. Firstly, resources exhibit heterogeneity, but even the most advanced heterogeneity-aware schedulers can only reach the GPU-type level. In addition, most scheduling systems cannot perform well in balancing efficiency and fairness, which leads to unfair resource allocation and reduced user satisfaction. Moreover, we have noticed the phenomenon of cluster fragmentation and job starvation.

In this paper, we propose a new scheduling architecture: Hops, which includes (1) fine-grained heterogeneity awareness and accurate throughput estimators, which allows for heterogeneity awareness at the server entity level. (2) Hops performs resource allocation by executing prior weighted integer linear programming (ILP) for specific placement locations, effectively balancing fairness and efficiency. (3) Hops introduces "latency ratio fairness" (LRF) as a user fairness criterion, which helps reduce starvation and enhance user experience. (4) To address cluster fragmentation, Hops intentionally uses low-sensitivity jobs to fill fragments. The final experimental results show that, in physical experiments, compared with the state-of-the-art scheduling architectures: Sia [17] and Gavel [32], Hops reduces cluster completion time by 18.5% to 34.2%, shortens average job completion time (JCT) by 27.4% to 45.9%, lowers waiting latency by 35.4% to 54.9%, significantly reduces cluster fragmentation, and performs significantly better in fairness metrics compared to Sia and Gavel. In the 512-GPU simulation experiments, Hops not only improves system efficiency but also reduces the maximum job latency ratio by over 21× and decreases cluster fragmentation to less than 1 GPU per round on average.

*Corresponding author.

ACM ISBN 979-8-4007-1286-9/24/11…$15.00
https://doi.org/10.1145/3698038.3698515

CCS CONCEPTS

• **Computer systems organization → Cloud computing**;
• **Software and its engineering → Scheduling**.

KEYWORDS

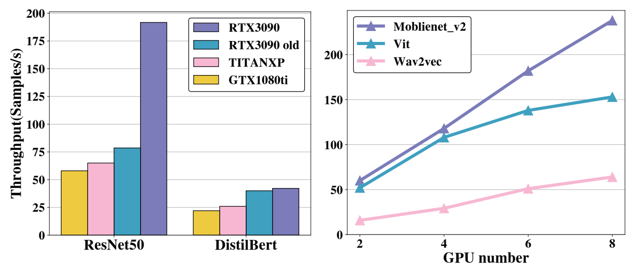
Deep learning, multi-tenant cluster, scheduling system, heterogeneous awareness, fairness

ACM Reference Format:
Qinghe Wang, Futian Wang, and Xinwei Zheng. 2024. Hops: Fine-grained heterogeneous sensing, efficient and fair Deep Learning cluster scheduling system. In *ACM Symposium on Cloud Computing (SoCC '24), November 20–22, 2024, Redmond, WA, USA.* ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3698038.3698515

1 INTRODUCTION

With the development of artificial intelligence, the number of DL tasks has surged, including image tasks [7, 14, 46], text tasks [6, 39, 41], multimodal tasks [34], and interactive tasks [31, 45] and so on. DL tasks usually require significant hardware resources to accelerate training. Consequently, large-scale distributed clusters and cloud platforms dedicated to deep learning have emerged. These clusters usually consist of many servers and a large number of hardware acceleration resources, such as CPUs and GPUs. At the user level, cloud platforms provide an interface through which users can submit resource requests to apply for acceleration resources to run DL tasks. The scheduling system serves as a comprehensive architecture bridging the user side and the cluster hardware. Its design goals need to consider not only how to allocate resources among different users to maximize overall cluster efficiency but also the user experience.

As the scale of clusters expands and hardware acceleration resources such as GPUs continue to evolve, these acceleration resources exhibit heterogeneity. This results in different throughput performance for DL tasks running in the cluster under different resource allocation configurations, as shown in Figure 1a. Having awareness of the throughput of tasks can guide scheduling decisions. With the development of multi-tenant deep learning cluster scheduling systems, some advanced schedulers have achieved a certain degree of heterogeneity awareness, such as the state-of-the-art scheduling systems: Gavel [32] and Sia [17] (which will be detailed in Section 2.1). They consider the GPU type as a heterogeneous resource. However, sometimes even with the same GPU type, the job throughput may vary significantly. For instance, the throughput of the ResNet job in Figure 1a shows a significant difference when running on RTX3090 and RTX3090old (RTX3090 installed in a less performant server). For the sake of scheduling convenience, designers sometimes do not distinguish such differences. Even for completely identical types of acceleration resources, physical-level differences may arise due to their different locations, interconnection methods with other servers, or switch performance differences.



(a) resource heterogeneity **(b) DL jobs heterogeneity**

Figure 1: In Figure 1a, we ran the ResNet [14] model (using the ImageNet [5] dataset) and the DistilBert [44] model (using the IMDb [26] dataset) on various types of accelerators, showing their throughput differences. In Figure 1b, various jobs exhibit different throughput growth as the number of GPUs increases.

Additionally, DL jobs also exhibit heterogeneity. For instance, different jobs exhibit different speedup ratios as acceleration resources increase, as shown in Figure 1b. Measuring each job under all resource configurations incurs significant time overhead. Therefore, designing a throughput estimator that is both time-efficient and highly accurate is a challenge.

Besides, the scheduling system should have an efficient and fair scheduling algorithm for resource allocation. Currently, there are already many kinds of scheduling algorithms. For example, some works utilize heuristic algorithms [13, 27, 38, 49], which mostly iteratively approach the theoretically optimal scheduling strategy. Other works use reinforcement learning and other deep learning methods to make scheduling decisions [4, 28, 42], which require a large amount of training data. Furthermore, some works employ operations research-based planning algorithms to achieve optimal scheduling decisions [17, 32]. planning algorithms can theoretically achieve the optimal scheduling strategy, but the time cost for searching for the best placement strategy increases exponentially with the cluster size. Therefore, it generally requires reducing the search space to lower the decision-making time cost. However, most of these scheduling strategies cannot perform well in balancing fairness and efficiency. While improving system efficiency, it is also important to ensure fairness and attach importance to user experience.

Additionally, queuing strategies in scheduling systems are crucial, as they usually determine the priority at which users receive resources, reflecting fairness. We find that many scheduling algorithms, such as Least-Attained Service (LAS) [35], shortest-job-first (SJF) [37], and shortest-remaining-time-first (SRTF) [13] are known to minimize the average JCT [11, 12], often overly favor short jobs, leading to starvation for relatively longer jobs (shown in Figure 13 in Section 4.5). To address the starvation issue, some schedulers have taken measures, such as using a threshold in Tiresias [13] to determine if a job is starving and then elevating it to the highest priority queue if it is. However, the length of user-submitted jobs varies, and they have different tolerance levels for waiting latency in the queue. Shorter jobs usually have stricter latency requirements for scheduling. A simple threshold for all jobs as a criterion for starvation makes it hard to ensure a good user experience.

Furthermore, resource fragmentation in clusters is a common issue [19, 48]. After the scheduling system allocates resources, there may be fragmented resources left. Since these GPUs are distributed across different servers, some high-demand jobs may not be scheduled or, even if they are scheduled, may also suffer from poor cross-server performance due to placement-sensitivity [27]. This phenomenon not only leads to the waste of GPU resources but also causes latency for jobs with high resource demands. To address the resource fragmentation issue, the scheduling system must consider how to allocate these fragmented resources effectively while ensuring job throughput as much as possible.

To address the above challenges, we propose a new DL scheduling system called Hops. The contributions of Hops are as follows: (1) A high-accuracy and low-overhead throughput estimator. We use a new throughput estimation formula that extends throughput estimation with high precision to any configuration. We implemented a multi-layer database to store historical data, which can correct potential biases in the throughput formula. As historical data accumulates, the granularity of heterogeneity awareness can reach the server entity level. (2) A scheduling algorithm based on a new ILP formula to solve specific allocation positions, incorporating service window and priority weights strategies to balance fairness and system throughput. (3) Reduction of cluster resource fragmentation. We propose a "sensitivity adjustment" algorithm and intentionally generate more configurations for low-sensitivity tasks to fill resource fragments, effectively solving the resource fragmentation problem. (4) A new fairness metric called "latency ratio fairness". For users sharing the cluster, the latency ratio reflects the degree of starvation. Ensuring latency ratio fairness can effectively enhance user experience.

2 BACKGROUND

2.1 Previous Schedulers

Since distributed deep learning clusters have become increasingly common, many schedulers designed for deep learning tasks have emerged [13, 15, 17, 27, 32, 38, 49].

Gandiva [49] improves cluster utilization through job packing and uses a custom suspend-resume mechanism to safely suspend jobs. It also employs a migration mechanism to handle cluster fragmentation and conflicting jobs. However, its scheduling approach based on packing is greedy and lacks heterogeneity awareness.

Tiresias [13] uses multi-level discrete queues to implement preemptive job execution. It also introduces the Two-Dimensional Attained Service-Based Scheduler (2DAS), which has a better perception of job duration, thus reducing the average JCT. It determines whether a job needs aggregated placement based on its placement sensitivity.

Gavel [32] is the first scheduler to consider different GPU types as distinct resource types. It measures job throughput to generate a throughput matrix and formulates different objectives as a linear programming problem of wall-clock time allocation. This approach determines the optimal time allocations for tasks on various resources, fitting the optimal time allocations through iterative time rounds. However, Gavel does not support elastic job requirements.

Pollux [38] supports elastic resource scaling and can dynamically adjust batch sizes. Pollux uses a genetic algorithm to consider the vast search space, which may lead to significant decision-solving time. Besides Pollux also lacks heterogeneity awareness.

Sia [17] is currently the most advanced scheduler supporting both heterogeneity awareness and elastic scaling, as well as batch size adjustment. To reduce profiling time, Sia uses a crude bootstrapped throughput model for the distributed throughput of some jobs. However, initial crude guidance can prevent the scheduler from ever obtaining opportunities to run in certain configurations, thereby failing to refine the initial crude bootstrapped throughput in the future. Sia uses an ILP formulation to plan the configuration of tasks and employs several measures to reduce the search space, significantly improving the throughput of cluster jobs. However, Sia's configuration generation assumes that job elastic demands can expand arbitrarily, which might not be supported by users in many scenarios. Moreover, we observed that Sia does not effectively guarantee queuing logic. Even tasks at the end of the waiting queue with high effective throughput acceleration ratios can have a greater chance of being scheduled than tasks at the front with lower throughput acceleration ratios.

Regarding heterogeneity awareness, the most advanced scheduler, Sia and Gavel, can only detect GPU type. However, as shown in Figure 1a, even the same type of GPU can result in significant throughput differences. From a finer-grained perspective, each server entity can yield different acceleration effects due to factors such as its location and variations in switch performance.

In balancing efficiency and fairness, although Pollux and Sia provide a knob to represent the balance between throughput and fairness, they achieve this by "reducing throughput optimization" to indirectly increase the "equal access to resources" for different jobs. However, "equal access to resources" does not equate to the fairness that clusters are interested in, which is usually reflected in the queuing strategy. This method cannot explicitly satisfy queuing logic, thus failing to effectively ensure fairness.

Concerning the cluster fragmentation issue, among the above schedulers, only Gandiva explicitly employs a migration mechanism [49] to handle cluster fragmentation. However, migration still incurs overhead.

2.2 User selectable requirements

Currently, state-of-the-art schedulers like Sia and Pollux support elastic resource requirements, but they assume that user requirements can be continuously and infinitely expanded, without considering the user's perspective. While an infinite and continuous demand space allows for allocating substantial resources to long-running jobs when resources are abundant, users may not be able to afford such a loose elastic expansion due to leasing costs and the potential impact on training outcomes. For example, when the number of GPUs is excessively large, an overly large global batch size may reduce randomness, leading to model overfitting and consequently affecting the quality of model validation performance. Moreover, a larger number of GPUs also incurs greater communication overhead, which potentially impacts training time and costs. Therefore, we give users the option to specify their requirements. Users can define rigid or elastic requirements. We express user requirements as a limited set:

$$R = \{R_1, R_2, R_3, \ldots, R_n\}$$

where R represents the set of user requirements, and each element in the set represents a specific number of GPUs requested by the user.

If users provide only maximum and minimum requirement boundaries $[R_{min}, R_{max}]$, cluster operators can set a maximum number of requirements R_{num}. This allows the generation of discrete requirements at equal intervals between the given boundaries. (Otherwise, the continuous range of demand levels could be immense, resulting in a huge search space). When increasing the number of GPUs, we keep the batch size within a single GPU constant. Thus, expanding the number of GPUs increases the global batch size, necessitating the adjustment of hyperparameters such as learning rate to maintain training effectiveness. Therefore, this paper supports users in specifying learning rates for each demand quantity. Alternatively, if users only provide one learning rate for one of the requirements, we support linear scaling rules [10] or square root scaling rules [23] to automatically adjust the learning rate (also allowing users to choose the most suitable scaling rules for specific tasks).

3 DESIGN OF THE SCHEDULING SYSTEM

In this section, we will introduce a comprehensive system design of Hops, including the overall processing logic of the system and detailed descriptions of various important modules. This includes the implementation of the profiler, the design of the multi-layer database, a detailed representation of the throughput formula, and the internal implementation details of the scheduler, among others.

3.1 System architecture

As shown in Figure 2, our system first processes incoming jobs by using a profiler (1) to run the job once on different types of resources (2) (each time for a few iterations). After obtaining the runtime data, the profiler writes this data into a multi-level database (3). The job is then placed into a waiting queue to await scheduling (4). The scheduler conducts round-based scheduling based on the current queue status and available resources (5). It generates certain configurations for jobs and estimates the throughput for each configuration using the throughput estimator (6), which accesses data from the multi-level database to achieve high accuracy (7 , 8). The scheduler then solves the optimal allocation within a round using an ILP formulation based on the estimated throughput. Finally, the placer executes the resource allocation (9). While jobs run in the cluster, throughput-related data is continuously generated and stored in the database. When generating configurations in the next round, the system predicts the throughput of the configurations based on the updated data, allowing for dynamic adjustments. Additionally, the cluster includes a monitor (not shown in Figure 2) that continuously checks if a job has failed or ended, releasing the occupied resources immediately upon failure or end.

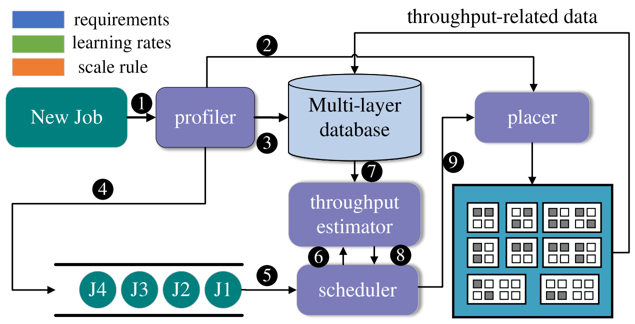


Figure 2: System architecture

3.2 Implementation of the profiler

Once the profiler detects that a task has arrived in the system, it will run the task on each type of GPU for approximately 10-50 iterations to obtain the basic timing modules for that task. Since tasks can be executed in parallel on different types of GPUs and the number of iterations is relatively small, the overhead of profiling is very low. In our tests, most models can be completed within one minute. Additionally, the profiler's analysis function is implemented as a plugin within each job. This plugin operates in two modes: profiling mode and running mode. In profiling mode, which is used for measuring new jobs, the profiler measures several basic time components of the job: data loading time, forward propagation time, backward propagation time, model update time, and process synchronization latency, and writes these measurements to the database. In running mode, the plugin only records data loading time and process synchronization latency, whose values tend to fluctuate significantly, requiring long-term measurement to obtain statistical probabilities, which we will explain in detail in Section 3.5.

To avoid conflicts between profiling tasks and tasks in the waiting queue, we designed a dedicated profiling queue. The scheduling logic for this queue is straightforward: at the start of each round, all tasks in the profiling queue are executed first, with any remaining idle resources then allocated to tasks that require actual execution.

3.3 Multi-layer Throughput Database

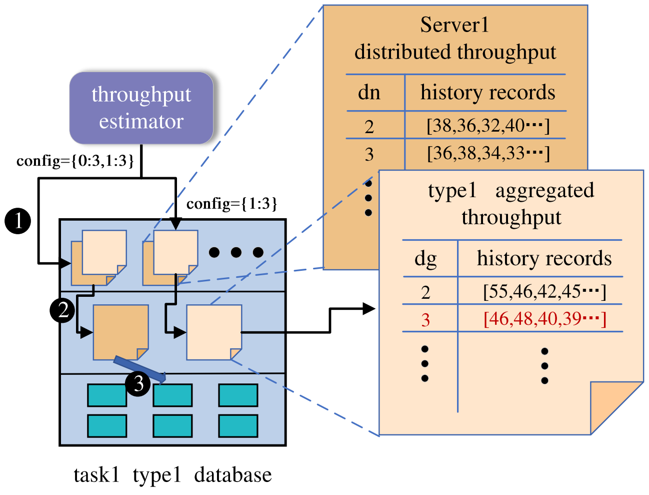


Figure 3: Muti-layer database. The multi-layer database is divided into three layers. The first layer is the entity layer, which stores the throughput for each entity. The second layer is the type layer, which stores throughput under a certain type of resource. The third level stores the basic time modules of a training iteration

We implemented a multi-layer throughput database, as illustrated in Figure 3. This database establishes a separate database object for each job and each resource type. Each database object is divided into three layers: the server entity layer at the top, the type throughput layer in the middle, and the theoretical throughput layer at the bottom.

In the server entity layer, the throughput data for each job running on a specific server is recorded. In the type throughput layer, the running data for each job on a resource type is stored. In the theoretical throughput layer, several basic times of training measured by the profiler are saved.

Data in the top two layers is recorded in two table formats, as shown in Figure 3: an aggregated throughput table, which records historical throughput data of the job running with a certain number of GPUs, and a distributed throughput table, which records the job's historical throughput data running with different numbers of nodes.

All records from individual server entities can be found in the type throughput layer, but individual server entities can provide more detailed predictions.

3.4 Configuration expression

We use a dictionary containing multiple key-value pairs to represent a configuration of a job:

$$config = \{s_1 : d_{g1}, s_2 : d_{g2}, \ldots, s_t : d_{gt}\}$$

The configuration is location-specific. Here, the keys represent the global IDs of the nodes, and the values represent the allocated number of GPUs.For example, config={0:2, 2:4}, which means that we assign 2 GPUs to the job on node 0 and 4 GPUs to the job on node 2 (a total of 6 GPUs). This is a cross-node configuration, which we call distributed configuration/placement, and oppositely, only placement within one node, which we call aggregate configuration/placement.

3.5 Throughput Estimator

When the generated configuration is a distributed one, better-performing nodes will be affected by worse performance, so the total throughput of a configuration depends on the slowest GPU. Specifically, we need to multiply the per-GPU throughput of the slowest GPU by the total number of GPUs:

$$config\_throughput = \min_i(throughput_i) \cdot \sum_{i=1}^{t} d_{gi}$$

where $throughput_i$ represents the throughput per GPU of the job on each node, and d_{gi} represents the number of GPUs allocated to each node. To estimate the aggregated and distributed per-GPU throughput of a node, we access a multi-layer database following a "top-down" approach. The throughput estimator first accesses the server entity layer (Figure 3 ❶). If this is not available, it proceeds to the type throughput layer (Figure 3 ❷). If it still misses, it accesses the theoretical throughput layer (Figure 3 ❸) to obtain the basic time modules for training. Then, using the throughput formula to predict. Below, we will detail the throughput prediction formula.

Since most DL jobs use synchronous data parallelism (DP), we focus on data parallelism. Through observing the training time overhead of numerous DL tasks using DP, we have the following conclusions: (1) The training time per mini-batch generally consists of data time, forward propagation (FP) time, backward propagation (BP) time, communication time, and process synchronization waiting time. (2) Many distributed training frameworks (e.g., PyTorch, TensorFlow) can overlap computation and communication, as well as parallel execution of CPU processes and CUDA processes. (3)Each mini-batch training cycle has two synchronization points: before each FP, the job needs to wait for data processing and synchronization of the previous round's gradients; before each BP, the job must wait for FP to complete (as shown in Figure 4). (4) Communication time is proportional to the model's parameter size.

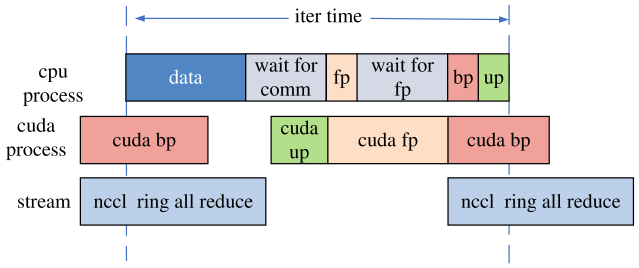


Figure 4: iter time breakdown. We decompose the training time of each iteration into three levels: CPU level, CUDA level, and communication stream. Within each level, we further break down the training time into different components.

Using the above conclusions, we can formulate the training time per round of min_batch as follows:

$$T_{iter}=\begin{cases}\max(T_d,\max(T_b+\widetilde{T_c},T_c)+T_u)+T_f+T_w & \text{if pattern}=1\\ T_d+\max(T_b+\widetilde{T_c},T_c)+T_u)+T_f+T_w & \text{if pattern}=2\end{cases} \tag{1}$$

where T_d represents the data time (including data loading and data processing), T_f and T_b represent the FP and BP time, respectively, T_w represents the training synchronization waiting time caused by speed differences between processes, T_c represents the communication time, $\widetilde{T_c}$ denotes the portion of communication time that does not overlap with the BP, which we empirically set to $0.2 * T_c$, and T_u represents the time to update the model weights.

In our experiments, we found jobs might follow two distinct patterns. One pattern overlaps computation and communication, as well as BP and data time. The other pattern overlaps only computation and communication. An example of the latter is seen in Hugging Face's Trainer class, which triggers gradient synchronization immediately after BP by checking if the gradient is zero or infinite, causing BP to complete before data processing. In our scheduling architecture Hops, profiler analysis automatically determines which pattern the training job follows.

T_f, T_b, and T_u in equation (1) are relatively stable and can be obtained by running a few iterations once. The communication time T_c depends on the resource type, the number of resources, and the communication algorithm. In mainstream training frameworks, the ring all-reduce algorithm [36] is often used within a single machine, with the theoretical communication time scaling as (N-1)/N, where N is the number of GPUs, which has been detailed in the previous studies [47]. For multi-machine scenarios, communication time primarily depends on network bandwidth. The ring all-reduce algorithm can result in high communication latency due to the potentially large ring size. Recently, new communication algorithms [18, 20, 21, 30, 33] have emerged, reducing communication latency. Therefore, estimating latency can be challenging, we approximate it by considering only network bandwidth in this work. Combining the above conclusion (4), we model T_c as follows:

$$T_c=\begin{cases}\dfrac{params}{B_{link}} * \dfrac{N-1}{N} & \text{if } d_n=1\\[2ex] \dfrac{params}{B_{net}} & \text{if } d_n>1\end{cases} \tag{2}$$

where $params$ is the number of parameters of the job, B_{link} is the intra-node link bandwidth of a certain resource type (The unit is parameters/s in this context), N represents the number of GPUs, d_n represents the number of nodes in configuration and B_{net} is the network bandwidth for the resource type. In this paper, B_{link} and B_{net} are measured offline by running different parameter quantities jobs under each resource to measure their communication time.

T_d is a random variable that can fluctuate. In our experiments, we observed that the data loading process for some jobs is very stable, while for others, it fluctuates significantly (Figure 7a). The variability in unstable data loading times tends to cluster around two distinct points, roughly forming two normal distributions. We hypothesize that this instability is related to the underlying hardware characteristics. The CPU continuously loads data from the disk into memory for the GPU to process. Since forward computation requires the data to be loaded, if the main process is slower than the data-loading process, the data is already preloaded into memory by the time the forward pass begins. In such cases, transferring the data from memory to the GPU is very fast. However, if the data is not ready in memory, the GPU process must wait for it to load from the disk, which takes much longer.

Thus, we categorize data loading times into two cases: when data hits the cache (in memory) and misses the cache. The Central Limit Theorem (CLT) [9] states that when many independent random variables with finite means and variances are summed, their total tends to follow a normal distribution. When a data load hits/misses the cache, its time is influenced by multiple factors such as CPU clock speed, I/O device read/write speeds, data sample size, and memory bandwidth, whose combined effects can be modeled as a normal distribution based on the CLT. Therefore, we model the data loading time as two conditional probabilities, each following a normal distribution.

$$T_d=\begin{cases}T_{cache}\sim N(\mu_c,\sigma_c) & \text{with probability } p_{cache}\\ T_{load}\sim N(\mu_l,\sigma_l) & \text{with probability } p_{load}\end{cases} \tag{3}$$

Over the long term, the means of the two normal distributions for data loading are essentially determined by data characteristics and hardware properties. Therefore, the profiler continuously records data loading times, allowing the model to gradually approach its true distribution.

T_w arises due to the differences in training speeds. We observed that, apart from T_d, there are no significant differences in other basic times, so usually T_w stems from inconsistencies in data loading. Therefore, T_w is highly correlated with T_d and can be expressed as different normal distributions under the same conditional probability.

Since T_w and T_d follow conditional probability distributions, the T_{iter} will also follow a distribution under two conditional probabilities. When estimating the throughput, we must represent the time with a single value, so we calculate the expectation of T_{iter}.

$$E(T_{iter}) = p_{cache} * E(T_{iter_c}) + p_{load} * E(T_{iter_l}) \tag{4}$$

$$throughput = \frac{batch\_size}{E(T_{iter})} \tag{5}$$

where T_{iter_c} and T_{iter_l} denote the training time per iteration when data loading is hit or miss.

When historical data is missing, the top two layers of the database are empty, and the job throughput will be predicted based on the above formula. In Section 4.1, we have demonstrated the accuracy of our throughput estimates for various configurations. However, for jobs with unstable data loading, due to the small initial iterations, it is difficult to fully characterize the mean and variance of a normal distribution. Therefore, we continue to record data loading times during subsequent runs. As historical data accumulates, the probability of jobs hitting the top two layers of the multi-layer database will gradually increase, and the throughput data obtained from actual measurements will be more accurate than the theoretical analysis values.

Prediction for other workloads. Hops leverages the characteristics of deep learning training to predict throughput, but we believe it can also handle other batch workloads by using throughput estimators tailored to each workload type. For instance, a new throughput prediction formula could be provided for batch deep learning inference jobs. Since inference tasks only involve data loading and the forward process, we can modify the above formula slightly to support inference tasks.

3.6 Scheduler

Our scheduler (as shown in Figure 5) maintains a queue that adheres to Latency Ratio Fairness (LRF) and refreshes periodically (every 30 seconds). The scheduler executes round scheduling. At the beginning of each round, we perform an Integer Linear Programming (ILP) to select the optimal configurations, which is precise down to the specific location since our configuration representation includes server IDs. If there are still available resources after the initial planning, we continue to perform planning to avoid wasting idle resources due to cluster fragmentation or job termination. Our scheduler aims to maximize cluster throughput while ensuring latency ratio fairness.

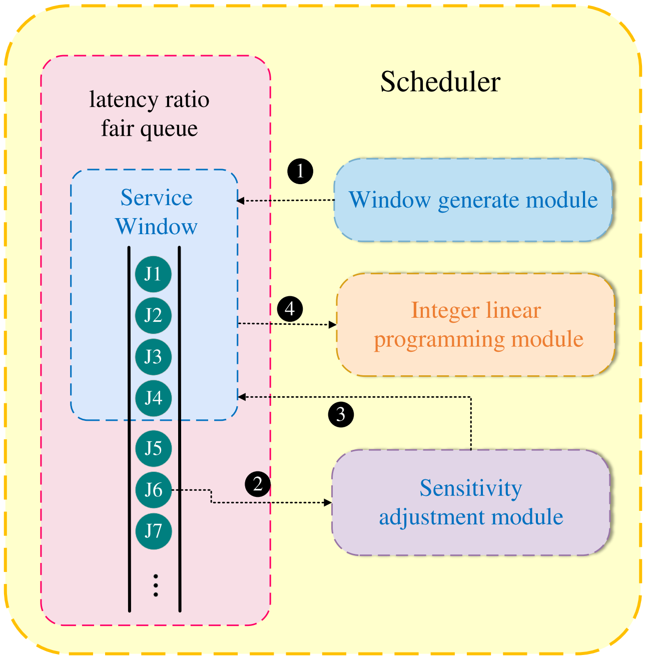


Figure 5: schedule process

Latency Ratio Fairness (LRF). In conversations with nearly 20 cluster users from various artificial intelligence fields, including autonomous driving, distributed systems, natural language processing, and computer vision, as well as interdisciplinary areas such as medical imaging and materials science, we found that users often experience reduced satisfaction with cluster usage due to prolonged waiting times for their jobs, leading to a loss of interest in using the cluster. This finding aligns with insights from previous scheduling systems Themis [27]. Notably, users submitting jobs of varying lengths have different tolerances for the waiting latency they encounter [50]. Typically, the shorter the job's lifespan, the lower its tolerance for the same amount of waiting time. For instance, a 1-hour wait is unacceptable for a job with a lifespan of 10 minutes, as it waits six times its duration. However, for a job with a lifespan of 10 hours, a 1-hour wait only accounts for 10% of its total lifespan, which is more acceptable. To measure the tolerance of different jobs to waiting latency and ensure that users do not lose interest in using the cluster due to starvation, we propose a metric to quantify job starvation, called the latency ratio (LR).

$$latency\_ratio_i = \frac{wait_i}{age_i} \tag{6}$$

where $wait_i$ represents the waiting latency of the i-th job in the waiting queue, and age_i represents the age of the i-th job, which is not equivalent to the job completion time (JCT); rather, it is the expected run time without waiting, estimated before the job starts running. This distinction is made because JCT includes waiting time, ensuring that the latency ratio does not exceed 1 and preventing the degree of starvation from increasing linearly with waiting time. $latency\_ratio_i$ represents the proportion of a job's waiting delay relative to its own lifespan. A higher latency ratio indicates that a job finds the wait time more intolerable and is experiencing greater starvation. Hence, we propose the **"Latency Ratio Fairness"** (LRF): In the process of cluster scheduling, the latency ratio for each job should be kept as low as possible and should tend to be similar across jobs. This reflects that the system does not excessively favor either short or long jobs, thereby preventing job starvation.

Additionally, we propose a metric to measure system busyness: the Recent Average Queue Latency Ratio (RAQLR).

$$queue\_latency\_ratio = \frac{\sum_i^m latency\_ratio_i}{m} \tag{7}$$

where m is the number of jobs that have died within h hours, $latency\_ratio_i$ represents the latency ratio of the m dead jobs. $queue\_latency\_ratio$ is the average latency ratio of dead jobs over the past h hours, which reflects the recent level of queue congestion. The higher the value, the busier the system has been in the last h hours. As for the value of h, the cluster personnel can adjust it to measure the system busyness metrics at different time scales. The metric can not only indicate the busyness of the system for the cluster personnel but also provide users with a rough judgment of how much time their jobs need to wait for their jobs to run in the cluster.

To satisfy the LRF, we try to design a priority that can fairly allocate resources to long and short jobs, we use the Latency Ratio (LR) priority:

$$prior_i = \frac{wait_i}{age_i} \tag{8}$$

Here, $wait_i$ represents the waiting latency of queuing, and age_i represents the expected running time of the job without waiting in the cluster. Since the job can change the heterogeneous resource and GPU number when the system runs, we use the following formula to estimate the expectation of age_i under the heterogeneous resource:

$$age_i = \sum_{type} P(type = j) \cdot \frac{total\_samples_i}{throughput_{ij} * R_{avg}} \tag{9}$$

where $type$ represents the resource type, and $P(type = j)$ is the probability that a job will select the j-th type of resource. We calculate this as $N_{type}/N_{cluster}$, where N_{type} represents the total number of GPUs of a certain type and $N_{cluster}$ represents the total number of GPUs in the cluster. $total\_samples$ represents the total number of execution samples for a job. Since the total number of execution epochs for a job is usually fixed, and each epoch requires a complete traversal of the dataset, the $total\_samples$ is typically determined. $throughput_{ij}$ is the per-GPU throughput of job i on the j-th type of resource (which can be measured by the throughput estimator). R_{avg} represents the average requirement of the job requirements set.

In Hops, the queue continuously updates each job's wait time in the system and promptly adjusts the latency ratio priority for each job, thereby updating the queue order (in this paper, the update frequency is set to once every 30 seconds). In the subsequent linear programming, higher-priority jobs will be prioritized for scheduling.

Plan multiple times within a round. Unlike Sia and Gavel, which only plan once at the beginning of each round, we allow that after the first plan, as long as there are available resources, Hops can continue to use ILP programming to allocate the idle resources generated by fragmented resources and job deaths, but the end time of these smaller-scale plans must be the same as the end time of the first plan, to ensure that more new resource adjustment schemes can be generated in the next round.

Service window. If there are many tasks in the queue and the queuing follows a certain priority order, planning the entire queue could increase the search space and violate fairness. This is because high-throughput jobs would have a greater chance of being allocated resources regardless of queue order. Therefore, we create a service window (Figure 5 ❶) at the head of the queue, where only tasks within this window have a chance to be scheduled (Figure 5 ❹). The size of the window is determined by the total cluster resources and the minimum resource requirement of each job in the queue: starting from the queue head, we traverse and accumulate the minimum resource requirements of each task until the total is greater than or equal to the total system resources. The last job in this traversal marks the boundary of the window, and tasks beyond this point in the queue do not have a chance to get resources. When a job's priority is high enough, it will appear within the window. While jobs within the window are not guaranteed to be allocated resources in a given round, being within the window significantly increases their chances of allocation in subsequent rounds.

Sensitivity. We quantify jobs' placement sensitivity as the aggregate placement throughput per GPU divided by the distributed placement throughput per GPU.

$$\rho_i = \frac{throughput_{agg}}{throughput_{dis}} \tag{10}$$

where $throughput_{agg}$ denotes the aggregate placement throughput, $throughput_{dis}$ denotes the distributed placement throughput. Here, we set $throughput_{agg}$ as the throughput of config={0:1}, which represents the task's single GPU throughput in server 0, which can be estimated by the throughput estimator. Similarly, we set $throughput_{dis}$ to the per-GPU throughput value for config={0:1, 1:1}, where hosts with IDs 0 and 1 are of the same resource type. ρ_i represents the placement sensitivity of a job. The higher the sensitivity of a job, the greater the negative impact of distributed placement on the job, primarily due to communication overhead. We set a threshold σ (here we set it to 1.4, which can be adjusted by the cluster operator); if the sensitivity is higher than this threshold, we consider the job to be overly sensitive, otherwise, we consider it to be a low-sensitivity job. Low-sensitivity jobs, compared to high-sensitivity jobs, are more advantageous for filling fragmented resources because their performance is not significantly affected by cross-server allocation.

Configuration generation. For each resource type in the system, assuming there are total N_{type} servers for each type, and the number of idle GPUs on each server t is S_t. And the requirements set of the job is $R = \{R_1, R_2, R_3, \ldots, R_n\}$. Our method of generating configurations is as follows: first, we compare each requirement R_k with the current number of idle GPUs on each server of the current type S_t. If $R_k < S_t$, an aggregated configuration $\{t : R_k\}$ is generated. After this step, we assess the job's sensitivity. If it's a high-sensitivity job, we only allow the job to create aggregated configurations (except its minimum requirement R_1 is less than the maximal number of GPU of servers in this type of resource). If it's a low-sensitivity job, we allow it to generate distributed configurations across adjacent node numbers, and each node must be filled as much as possible, this ensures the search space remains $O(N_{type})$. For each R_k, once there are fewer nodes to satisfy its distributed configuration, we won't generate configurations with more nodes.

Sensitivity adjustment algorithm. If the plan is for fragmented resources, the number of tasks screened out by the window may be small and may cause head-of-line (HOL) blocking, or the jobs at the front is placement-sensitive, making it impossible to allocate fragmented resources efficiently. Therefore, we propose a heuristic queue adjustment algorithm—Sensitivity adjustment algorithm. We determine whether the current round of planning is for fragmented resources (by checking if this planning is the first one of the round), if it is for fragmented, we allow a certain number of low-sensitivity jobs (fig5. ❷) outside the window to enter the window (fig5. ❸). These low-sensitivity tasks serve as filling in resource fragments. And the rules of adjustment are as follows: we calculate the average sensitivity of a subset of jobs in the window, denoted as ρ_{set}, and we calculate the average sensitivity of the entire queue ρ_{queue}, if $\rho_{set} > \rho_{queue}$, it means that the jobs in the windows is too sensitive, the planning for the fragmented resources may be less effective. We judge each job from the window boundary backward according to the queue order, If the sensitivity of the job $\rho_i < \rho_{queue}$ and its minimum requirement is less than the total amount of currently idle resources, add it to the window, otherwise continue to traverse backward until $\rho_{set} <= \rho_{queue}$ or the queue traversal ends. Although the algorithm violates fairness to a certain extent because it brings low-priority jobs into the window, we consider it worthwhile. Because fragmented resources are likely to cause low throughput of sensitive jobs and waste of idle resources, these allocation opportunities will be more efficient for low-sensitive jobs. And when the next round starts, the low-priority jobs are brought back to their original locations again.

Prior weighted integer linear programming. After generating the service window and the sensitivity adjustment algorithm, we get a subset of jobs that are allowed to be scheduled, and then we plan this subset to solve the specific allocation location. To make the throughput of different jobs comparable, we follow the practice in Sia [17], divide the throughput rate of each job under various configurations by its minimum throughput rate for normalization, and generate a throughput gain matrix G:

Table 1: Throughput gain matrix G. The symbol "/" indicates that the job does not have this configuration.

	config 1	config 2	config 3	config 4
job 1	1	1.8	2.6	/
job 2	2	1	3	4.1
job 3	1.2	1.6	1	/
job 4	1	1.8	3.2	/

each row of G represents the various configurations of a job and the throughput rate acceleration ratio brought by the configuration. We need to solve an allocation matrix X of the same size as G, $X_{ij} = 1$ means that the configuration is selected, and $X_{ij} = 0$ means that it is not selected. The GPU consumption of each configuration for a server t is recorded as $Comsume_{ijt}$. The plan must meet the constraints: (1) each job can only select a maximum of one configuration, and (2) each server has a limited number of GPUs and cannot be overallocated.

We will represent the plan as follows:

$$\max_{X} \quad \sum_i \sum_j (prior_i + bias)^{\lambda} \cdot G_{ij} \cdot X_{ij} \tag{11}$$

$$s.t. \quad \sum_j X_{ij} \le 1 \quad i = 1, 2, \ldots, n \tag{12}$$

$$\sum_j X_{ij} \cdot Consume_{ijt} \le S_t \quad t = 1, 2, \ldots, s \tag{13}$$

To ensure fairness, as the fairness that cluster operators care about is often reflected in the queue logic, tasks that are further ahead in the queue should be scheduled sooner. Therefore, in our objective, the configuration throughput is assigned a priority weight of the job. Here, $prior_i$ represents the priority of the i-th job in the waiting queue, and $bias$ is an offset term, which is to prevent zero or negative priorities from causing the optimization to select no configuration for a task. If all priorities are positive, then $bias = 0$; otherwise, $bias = abs(min(prior_i)) + 0.01$ (The 0.01 ensures the lowest-priority job has a non-zero weight to avoid starvation). The higher the priority of a task in the queue, the greater the weight assigned to its throughput. In optimizing this objective, for two tasks with the same throughput acceleration ratio, the task with the higher priority will be preferred by the integer linear programming and will be assigned to better resources for acceleration. λ is a knob that balances efficiency and fairness and can be adjusted by the cluster operators. When λ is set to 0, all task priority weights become 1, explicitly optimizing effective throughput. As λ increases, the differences in priorities are magnified due to the exponential function, making the optimization more inclined towards fairness. When λ approaches infinity, the optimization becomes equivalent to a greedy approach: allocating resources in descending order of priority, and each task selects the feasible configuration with the maximum throughput acceleration ratio until resources are insufficient for further allocation. In this paper, we set λ to 1, making the weights equal to the queue priorities.

Based on this planning formula, we calculate the allocation matrix X and allocate jobs resources, because the constraint (13) is satisfied, so we can ensure that the allocation is effective, and there will be no unallowable situation when $X_{ij} = 1$. In this plan, we solve the problem based on the GUROBI solver of the CVXPY package, and the solution can be completed within 1s for the total number of GPU cards less than 100. For larger clusters, we set the maximum tolerance gap=0.05%, that is, when the difference between the obtained value and the optimal value is less than 5%, it can jump out, and the cluster personnel can choose the maximum tolerance, although, for larger scales, the suboptimal solution may be obtained, because of the existence of tolerance, it will not lead to an obvious difference from the optimal solution.

4 EXPERIMENTAL RESULT

4.1 Prediction Accuracy Experiments

To validate the accuracy of our throughput estimator, we conducted several prediction experiments. For tasks with stable data loading, Hops demonstrated very high prediction accuracy. Figure 6 shows the throughput predictions for various configurations of the VIT and VGG models, with accuracy exceeding 95%.

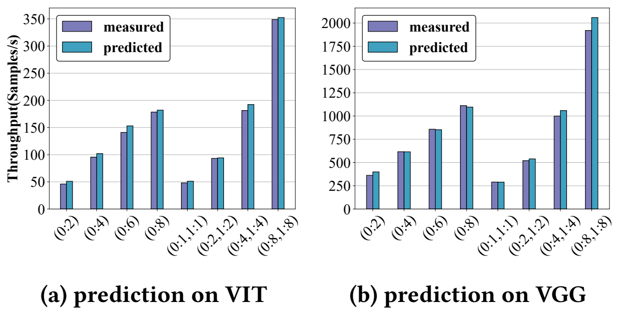
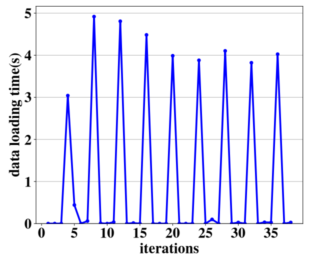


Figure 6: The prediction accuracy. The "predicted" means the predicted performance by our throughput estimator, and the "measured" denotes the directly measured throughput." (0:8, 1:8)" is the expression of a configuration, which means allocate 8 GPUs on both nodes of ID 0, 1

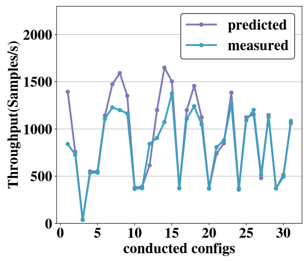
However, for tasks with unstable data loading, as mentioned in Section 3.5, their data loading times exhibit significant fluctuations (as shown in Figure 7a). Consequently, it is challenging to accurately capture the normal distribution and mean of these tasks' data loading times through initial measurements of a few iterations. This can lead to considerable initial estimation bias. Nonetheless, as historical data accumulates, the error gradually diminishes. In Figure 7b, we tested a set of jobs with unstable data loading and tracked the comparisons between our throughput estimator's predictions and the actual values across various configurations. It is evident that with increasing execution counts, the gap between the predicted and actual values consistently narrows.

4.2 Physical Experiments on Cloud

We applied for physical nodes on a cloud platform specifically designed for deep learning acceleration (https://bitahub.ustc.edu.cn) to conduct physical experiments. We compare the state-of-the-art heterogeneous-aware schedulers: Sia [17] and Gavel [32], and evaluate from five aspects: completion time, average JCT (Avg.JCT), average waiting latency (Avg.WT), maximum latency ratio (Max. LR), and an average number of fragments per round (Avg.frag). All our time



(a) unstable data loading



(b) predict unstable jobs

Figure 7: Unstable data loading time and prediction on jobs with unstable data loading time. Figure 7a traces the unstable data loading time generated by ResNet18 [14] on Imagenet [5]. Figure 7b compares the predicted and true throughput for these unstable data loading tasks

measurement metrics included the overhead of each system, such as profiling time and decision-making time. We count the fragments of each round as the number of idle GPUs in the cluster when there are unassigned jobs after each round of scheduling. Our physical experimental cluster includes 64 cards in 10 servers, 3 different GPU types, and 2 different server types:

- 2*8 Sugon W780-G GTX1080Ti with 192G memory size, 100 Gbps InfiniBand, and 2 Intel (R) Xeon (R) Gold 5118 CPU.
- 4*8 Sugon W780-G TitanXP with 192G memory size, 100 Gbps InfiniBand, and 2 Intel (R) Xeon (R) Gold 5118 CPU.
- 2*4 Sugon W780-G RTX3090 with 192G memory size, 100 Gbps InfiniBand, and 2 Intel (R) Xeon (R) Gold 5118 CPU.
- 2*4 TIGERWAY YK8100-8G RTX3090 with 384G memory size, 100 Gbps InfiniBand, and 2 Intel (R) Xeon (R) Gold 5218 CPU.

Workflow: We used a training workflow that contains mixed job types, which include classic network models such as ResNet [14] and VGG [46], as well as popular large model architectures like ViT [7], BERT [6], and GPT [39]. The datasets used included text, images, and audio. The workflow is shown in Table 2.

We categorize them based on their GPU duration into the following four categories: Small (S, 0-1h), Medium (M, 1-10h), Large (L, 10-50h), and Extra-large (XL, >50h). The distribution histogram is shown in Figure 8, more detailed categories of each task are shown in Table 2. There are a total of 25 tasks, and we have set a submission window of 2.5 hours, and we used the same workflow arrival order for all three schedulers.

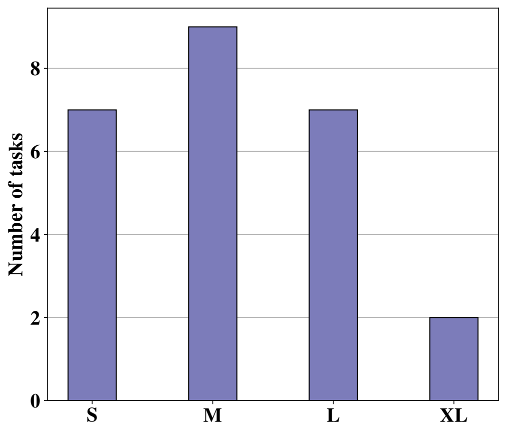


Figure 8: Job duration category classification

Baseline Settings: Since our tasks support user-selectable requirement sets, we suppose the GPU requirements are between 1 and 16 GPUs, with each user able to select up to 8 different requirements. For Gavel, since it does not support elastic requirements, we set the median requirement of the task as the rigid requirement for each job. Sia assumes an infinite elasticity space for users and uses powers of 2 for requirements to compress the space. However, this approach may lead to empty configurations in our proposed user-selectable requirements. Therefore, we ensure the effective implementation of the Sia algorithm by generating configurations for each requirement of a job. To control variables, we use the same latency ratio priority as Hops for the queuing priorities of both the Sia and Gavel.

Physical Experiment Results: The results are shown in Table 3. Compared to Sia, Hops reduced the makespan by 18.5% and the average JCT by 27.4%. When compared to Gavel, Hops shortened the makespan by 34.2% and the average JCT by 45.9%. Hops perform better in wait time compared with Sia and Gavel, reducing the wait time by 35.4% and 54.9% respectively. Additionally, Hops outperforms Sia and Gavel in the latency ratio, keeping the latency ratio for each job below 37%, meaning the wait time does not exceed 37% of the job's lifetime. Hops has a smaller average resource fragments, reducing the average number of fragments from 5.89 in Sia and 3.36 in Gavel to 0.15. The CDF plots of JCT and latency ratio are shown in Figure 9. It can be observed that Hops outperforms both Sia and Gavel in the two aspects.

4.3 Simulation Experiments

To conduct repeated experiments on larger-scale clusters and test some special settings, we conducted simulation experiments.

Table 2: Models and Datasets used in experiments.

Task	Models	Datasets	Category	Batch size	Optimizer
Image Classification	Resnet18/Resnet50 [14]	ImageNet/Cifar10 [5, 22]	XL/S	128/256	SGD
	Vgg11/Vgg13 [46]	ImageNet/Cifar10 [5, 22]	L/S	32/128	SGD
	Convnext_base [25]	ImageNet/Cifar10 [5, 22]	L/S	32/64	SGD
	Densenet161 [16]	ImageNet/Cifar10 [5, 22]	L/S	32/64	SGD
	Mobilenet_v2 [43]	ImageNet/Cifar10 [5, 22]	M/S	64/64	SGD
	Regnet [40]	ImageNet/Cifar10 [5, 22]	L/M	32/32	SGD
Text Classification	DistilBert [44]	IMDb [26]	L	128	AdamW
Image Classification	Vit [7]	Food-101 [2]	M	48	AdamW
Text Classification	MacBert [3]	C3 [24]	M	8	AdamW
Audio Classification	Wav2vec [1]	Common language	M	16	AdamW
LLM Finetuning	GPT-2 Medium [8]	Wikitext [29]	L	8	AdamW

Table 3: Comparison of Hops, Sia snd Gavel in the Heterogeneous physical cluster. Max.LR is the maximum latency ratio in jobs. Avg.frag is the average resource fragments per round

Policy	Makespan	Avg.JCT	Avg.WT	Max.LR	Avg.frag
Hops	9.51h	3.2h	0.95h	37%	0.15
Sia	11.68h	4.41h	1.47h	103%	5.89
Gavel	14.46h	5.92h	2.11h	131%	3.36

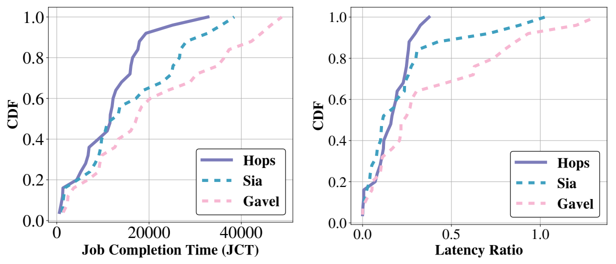


Figure 9: CDF of JCT and latency ratio compared with Sia and Gavel

To demonstrate the effectiveness of our simulation experiments, we replicated the configurations of the physical cluster and conducted experiments in the simulated cluster. The results of these experiments are shown in Figure 10, which are similar to those obtained from the physical cluster.

Afterward, we scaled up our simulation experiments: 64 servers, totaling 512 GPUs. We set up five types of resources:

- 16*8 Sugon W780-G GTX1080Ti with 192G memory size, 100 Gbps InfiniBand, and 2 Intel (R) Xeon (R) Gold 5118 CPU.
- 16*8 Sugon W780-G TitanXP with 192G memory size, 100 Gbps InfiniBand, and 2 Intel (R) Xeon (R) Gold 5118 CPU.

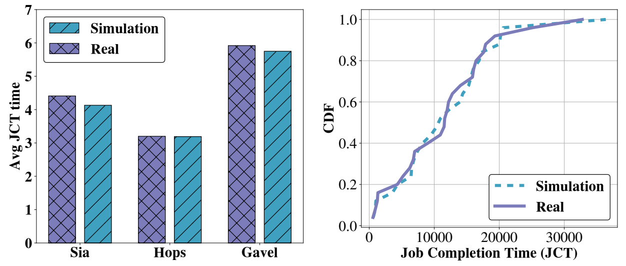


Figure 10: Avg JCT and CDF of JCT in both Physical experiments and simulation experiments

- 16*8 Sugon W780-G RTX3090 with 192G memory size, 100 Gbps InfiniBand, and 2 Intel (R) Xeon (R) Gold 5118 CPU.
- 8*8 TIGERWAY YK8100-8G RTX3090 with 384G memory size, 100 Gbps InfiniBand, and 2 Intel (R) Xeon (R) Gold 5218 CPU.
- 8*8 TIGERWAY YK8100-10G RTX3090 with 384G memory size, 100 Gbps InfiniBand, and 2 Intel (R) Xeon (R) Gold 5320 CPU.

Table 4: Comparison of Hops, Sia, and Gavel in the simulation experiment.Max.LR is the maximum latency ratio in jobs. Avg.frag is the average resource fragments per round

Policy	Makespan	Avg.JCT	Avg.WT	Max.LR	Avg.frag
Hops	10.48h	2.24h	0.87h	3.22	0.45
Sia	12.36h	3.79h	1.68h	78.5	56
Gavel	15.42h	4.04h	2.24h	70.43	18

The workflow uses 500 tasks, arriving according to a Poisson process, with an average of 100 tasks arriving per hour.

The submission window is set to 5 hours. We chose this arrival rate because our experiments found that it represents a relatively high job arrival rate, which maintains a larger number of tasks in the waiting queue. Under high-load conditions, it better tests the optimization performance of each architecture in scheduling. As shown in Table 4, compared to Sia and Gavel, Hops reduced the makespan by 15.2% and 32.0%, respectively, and reduced the average JCT by 40.9% and 44.5%, respectively. The average JCT of Sia and Gavel in the simulation experiment is closer because Sia's degree of elastic scaling is limited in scenarios where the user can choose requirements. Although Gavel only supports rigid requirements, the dense arrival of tasks did not lead to many resource vacancies.

To more clearly display the scheduling effects of each algorithm, we plotted the cluster throughput change curve by detecting the total cluster throughput every 10 seconds, as shown in Figure 11. It can be seen that the throughput of the Hops is significantly higher than that of the Sia and Gavel due to its finer-grained heterogeneity awareness, more precise position allocation, and handling of cluster fragmentation.

It is worth noting that in this simulation experiment, due to the dense arrival of tasks, queuing latency is inevitable. However, Hops significantly outperforms Sia and Gavel in the latency ratio fairness. The maximum job latency ratio for Hops is 3.22, while the maximum job latency ratio for Sia reached 78.5 and for Gavel reached 70.43 (shown in Table 4), meaning that some jobs have to wait for over 70 times their original runtime, which could lead users to abandon the cluster due to poor user experience. The CDF graph of the latency ratio for this simulation experiment is shown in Figure 12, where it can be seen that the latency ratio growth curve for Hops is almost vertical compared to Sia and Gavel. The reason Hops can significantly outperform Sia and Gavel in fairness is that Hops' planning explicitly considers queuing priority. Jobs with high waiting latency generate high weights in the next round, enabling timely scheduling, while Sia and Gavel lack consideration of queuing priority, resulting in poor fairness.

Regarding cluster fragmentation, Hops reduces the average resource fragmentation per round to 0.45 (shown in Table 4). This is a significant reduction compared to Sia and Gavel, which have 56 and 18 respectively. This improvement is attributed to our use of a sensitivity adjustment algorithm and intentionally generating more configurations for low-sensitivity jobs. These low-sensitivity jobs can help fill resource fragments during the scheduling process. Additionally, Hops plans the specific placement, which allows for better awareness of the GPU card limits of each server. In contrast, Sia and Gavel cannot fully account for the GPU limits of each server due to discrepancies between the computed allocation scheme and the actual placement.

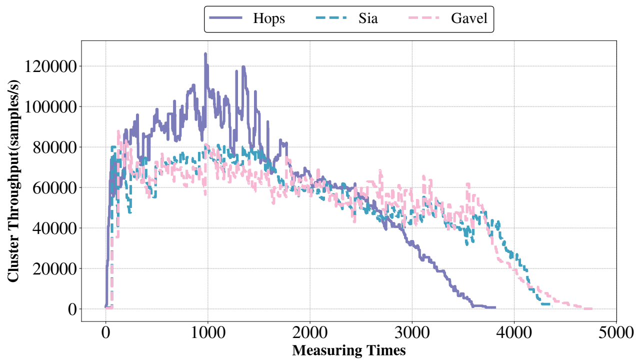


Figure 11: cluster throughput comparison of Hops, Sia, and Gavel

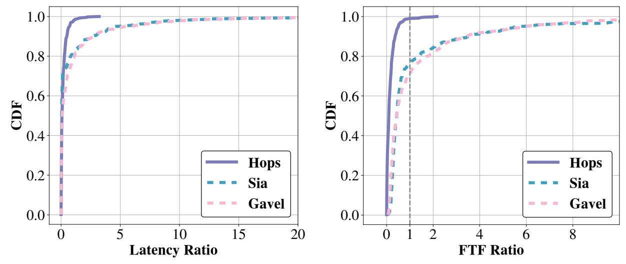


Figure 12: CDF of latency ratio (LR) and Finish-Time Fairness ratio ρ (FTF) in simulation experiments

4.4 Finish-Time Fairness (FTF)

This paper also supports other priorities. Mahajan et al. [27] introduced Finish-Time Fairness (FTF) in Themis. FTF requires that the completion time of each job in the cluster should not be lower than the time it would take to run if it were allocated 1/N of the cluster's resources, where N is the number of jobs in the cluster. The FTF ratio, denoted as ρ, equals T_{sh}/T_{id}, where T_{sh} represents the job's shared time in the cluster and T_{id} represents the time it would take to run if allocated 1/N of the cluster's resources independently. To adhere to the principle of fair resource sharing, each job in the cluster should have ρ values as low and as equal as possible, $\rho> 1$ indicates unfair execution: the job would complete faster on its own than when using the scheduler's strategy. In this paper, ρ in the heterogeneous cluster follows the same definition as in Sia [17]. After testing, Hops performs best when λ is set to 2, while in Sia, we set its parameter λ value to 1.1 and parameter p value to -0.5 as mentioned in their paper. Finally, under the same configuration as the 512-card simulated cluster mentioned above, the CDF graph of Finish-Time Fairness, as shown in Figure 12, was obtained. Hops has a maximum ρ value of 2.15, whereas Sia and Gavel have values of 28.27 and 32.01, respectively. Additionally, it can be observed that Hops' FTF ratio CDF curve is more vertical, and the proportion of unfair job allocations (where $\rho > 1$) has been significantly reduced. Hops significantly outperforms Sia and Gavel in FTF because Hops assigns a priority to each job that aligns with FTF. When a job is treated unfairly, its priority increases, thereby increasing its throughput weight and ensuring the job is scheduled in a timely manner. Conversely, jobs that have been excessively favored have lower priority and therefore reduced weights, decreasing their chances of being scheduled.

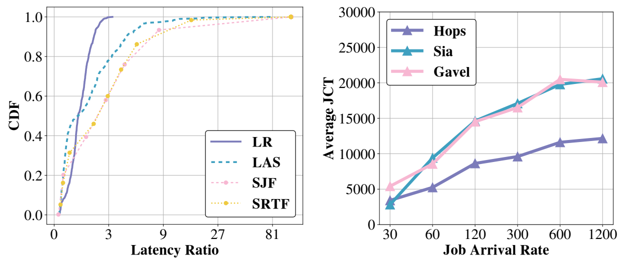


Figure 13: Ablation Experiment on Priorities and JCT Comparison under Different Job Arrival Rates

4.5 Ablation Experiment on Priorities

To demonstrate that some classical priority strategies may cause the system to overly favor small jobs, leading to the starvation of long jobs, we conducted an ablation experiment on priority strategies. In the same 512-card simulation setup mentioned above, we kept the scheduling architecture as Hops and only changed the priority strategy. We compared LAS, SJF, SRTF, and our LR priority. The results in Figure 13 show that LAS, SJF, and SRTF tend to favor short jobs, causing many jobs to wait more than 80 times their expected completion time. In contrast, the LR priority results in a maximum job latency ratio of 3 and a more vertical CDF curve.

4.6 Workload Intensity

To simulate the cluster results under different task arrival rates, we compared arrival rates of 30/h, 60/h, 120/h, 300/h, 600/h and 1200/h. We found that The advantages of the Hops algorithm become more significant (shown in Figure 13), as the difference in the average JCT between Hops and Sia and Gavel gradually increases with the growing density of task arrivals. This is due to the benefits of Hops' fine-grained heterogeneous-awareness and better alignment with the queuing logic.

4.7 Scalability

The overhead of Hops is divided into profiling time and the solving time of the ILP. Since each task only needs to run once on different types of GPUs (with 10-50 iterations), and profiling can be executed in parallel on different types of GPUs, the profiling time overhead of Hops is quite small. Most of the tested models in the experiments can be completed within one minute. Regarding the ILP solving time, we conducted several related experiments using the GUROBI solver in Hops to explore the scalability of the ILP solving overhead with respect to tolerance gap, task quantities, and cluster sizes.

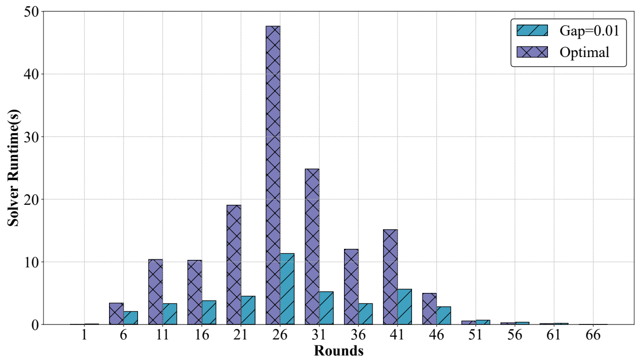


Figure 14: Comparison of search time with gap=0.01 and without gap

Firstly, many solvers now support the setting of tolerance gaps. By using tolerance gaps, solvers based on certain optimization algorithms can more quickly locate suboptimal solutions that are close to the optimal value, rather than traversing the entire space to find the optimal solution. Figure 14 shows the variation in solving time with and without a gap (i.e., finding the optimal solution) in the aforementioned simulation experiment with 512 cards. Here, we set the tolerance to 0.01, sampling every 5 rounds. It can be observed that the solving time is significantly reduced when the tolerance is set, with the vast majority of solving times remaining within 10 seconds. Additionally, we found that the actual gap is less than 0.005 in most cases, well below the upper limit.

We explored how the ILP solving time changes as the number of incoming tasks increases. Still, under the scale of 512 cards, the number of incoming tasks starts at 100 and increases up to 2000, we maintain a relatively high job arrival rate of 100 jobs per hour. In Figure 15, it can be observed that the solving time stabilizes when the number of tasks reaches 1000. This is because of the setting of our service window, which allows the ILP solver to handle only a number of tasks

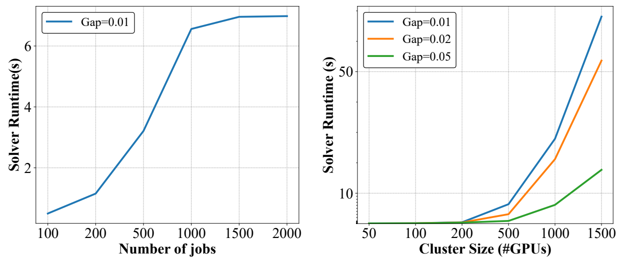


Figure 15: Scalability experiments on number of tasks and GPUs

suitable for the cluster size at one time, thus imposing an upper limit on the number of tasks the ILP can process.

Next, we explored how the ILP solving time changes with the cluster size. Since the solving time of the ILP tends to increase exponentially with the number of variables, the number of solving variables X is proportional to the configuration space. Our method of generating configurations keeps the complexity of the configuration space at $O(N_{type})$, which is within a controllable range. We tested the overhead of ILP solving time as the cluster size expanded. Considering the increased computing power of larger clusters, we linearly scaled the job arrival rate based on the arrival rate in the 512-GPU simulation above. Our experimental results are shown in Figure 15. For scales of 500 cards and below, the ILP solving time can be completed within 10 seconds. When the cluster size continues to grow and exceeds 1000 cards, we can set a larger gap. For example, when setting the gap to 0.05, the solving time for a scale of 1000 cards is about 6 seconds, and for 1500 cards, the solving time is around 17 seconds, which is approximately 3 times faster than the gap set to 0.01.

Overall, Hops can easily achieve low scheduling overhead for clusters with thousands of GPUs, meeting the scheduling requirements of many medium to large-scale clusters. For even larger clusters, the solving time may become longer, which could impact cluster performance. A feasible solution is to overlap the ILP solving time with job execution in each round. If the scheduling plan for the current round cannot be solved in time, we can reuse the solution from the previous round and apply the current solution in the next round. Since scalability optimizations for ultra-large clusters were not thoroughly addressed in this paper, we leave this for future work.

5 CONCLUSION

Theoretical and experimental validation has shown that the Hops scheduling architecture outperforms state-of-the-art scheduling algorithms in terms of improving cluster throughput and fairness metrics. Compared to Sia and Gavel, Hops reduces the makespan by approximately 18% to 34%, improves the average JCT metric by about 27% to 46%, and shortens the maximum job latency ratio by around 21 ×. Additionally, it also significantly outperforms the baseline in finish-time fairness (FTF).

6 ACKNOWLEDGEMENTS

We thank the anonymous reviewers and our shepherd Dr. Lukas Rupprecht for their valuable comments that improved this paper. We gratefully acknowledge the Institute of Artificial Intelligence and the Anhui Province Key Laboratory of Biomedical Imaging and Intelligent Processing, Hefei Comprehensive National Science Center, for providing both resources and funding for this research.

REFERENCES

[1] Alexei Baevski, Yuhao Zhou, Abdelrahman Mohamed, and Michael Auli. 2020. wav2vec 2.0: A framework for self-supervised learning of speech representations. *Advances in neural information processing systems* 33 (2020), 12449–12460.

[2] Lukas Bossard, Matthieu Guillaumin, and Luc Van Gool. 2014. Food-101–mining discriminative components with random forests. In *Computer Vision–ECCV 2014: 13th European Conference, Zurich, Switzerland, September 6-12, 2014, Proceedings, Part VI 13.* Springer, Springer, Zurich, Switzerland, 446–461.

[3] Yiming Cui, Wanxiang Che, Ting Liu, Bing Qin, Shijin Wang, and Guoping Hu. 2020. Revisiting Pre-Trained Models for Chinese Natural Language Processing. In *Findings of the Association for Computational Linguistics: EMNLP 2020*, Trevor Cohn, Yulan He, and Yang Liu (Eds.). Association for Computational Linguistics, Online, 657–668. https://doi.org/10.18653/v1/2020.findings-emnlp.58

[4] Hayssam Dahrouj, Ahmed Douik, Oussama Dhifallah, Tareq Y Al-Naffouri, and Mohamed-Slim Alouini. 2015. Resource allocation in heterogeneous cloud radio access networks: advances and challenges. *IEEE Wireless Communications* 22, 3 (2015), 66–73.

[5] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. 2009. ImageNet: A large-scale hierarchical image database. In *2009 IEEE Conference on Computer Vision and Pattern Recognition.* IEEE, Miami, Florida, USA, 248–255. https://doi.org/10.1109/CVPR.2009.5206848

[6] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. 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, Volume 1 (Long and Short Papers)*, Jill Burstein, Christy Doran, and Thamar Solorio (Eds.). Association for Computational Linguistics, Minneapolis, Minnesota, 4171–4186. https://doi.org/10.18653/v1/N19-1423

[7] Alexey Dosovitskiy, Lucas Beyer, Alexander Kolesnikov, Dirk Weissenborn, Xiaohua Zhai, Thomas Unterthiner, Mostafa Dehghani, Matthias Minderer, Georg Heigold, Sylvain Gelly, et al. 2020. An image is worth 16x16 words: Transformers for image recognition at scale.

[8] Gabriel Fabricius and Alberto Maltz. 2020. Exploring the threshold of epidemic spreading for a stochastic SIR model with local and global contacts. *Physica A: Statistical Mechanics and its Applications* 540 (Feb. 2020), 123208. https://doi.org/10.1016/j.physa.2019.123208

[9] Larry Goldstein. 2009. A Probabilistic Proof of the Lindeberg-Feller Central Limit Theorem. *The American Mathematical Monthly* 116, 1

(2009), 45–60. https://doi.org/10.1080/00029890.2009.11920908

[10] Priya Goyal, Piotr Dollár, Ross Girshick, Pieter Noordhuis, Lukasz Wesolowski, Aapo Kyrola, Andrew Tulloch, Yangqing Jia, and Kaiming He. 2017. Accurate, large minibatch sgd: Training imagenet in 1 hour.

[11] Robert Grandl, Mosharaf Chowdhury, Aditya Akella, and Ganesh Ananthanarayanan. 2016. Altruistic scheduling in multi-resource clusters. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation* (Savannah, GA, USA) *(OSDI'16)*. USENIX Association, USA, 65–80.

[12] Robert Grandl, Srikanth Kandula, Sriram Rao, Aditya Akella, and Janardhan Kulkarni. 2016. Graphene: packing and dependency-aware scheduling for data-parallel clusters. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation* (Savannah, GA, USA) *(OSDI'16)*. USENIX Association, USA, 81–97.

[13] 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)*. USENIX Association, Boston, MA, 485–500. https://www.usenix.org/conference/nsdi19/presentation/gu

[14] 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*. IEEE, Las Vegas, NV, USA, 770–778.

[15] Qinghao Hu, Meng Zhang, Peng Sun, Yonggang Wen, and Tianwei Zhang. 2023. Lucid: A Non-intrusive, Scalable and Interpretable Scheduler for Deep Learning Training Jobs. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2* (Vancouver, BC, Canada) *(ASPLOS 2023)*. Association for Computing Machinery, New York, NY, USA, 457–472. https://doi.org/10.1145/3575693.3575705

[16] Gao Huang, Zhuang Liu, Laurens Van Der Maaten, and Kilian Q. Weinberger. 2017. Densely Connected Convolutional Networks. In *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. IEEE, Honolulu, HI, USA, 2261–2269. https://doi.org/10.1109/CVPR.2017.243

[17] Suhas Jayaram Subramanya, Daiyaan Arfeen, Shouxu Lin, Aurick Qiao, Zhihao Jia, and Gregory R. Ganger. 2023. Sia: Heterogeneity-aware, goodput-optimized ML-cluster scheduling. In *Proceedings of the 29th Symposium on Operating Systems Principles* (Koblenz, Germany) *(SOSP '23)*. Association for Computing Machinery, New York, NY, USA, 642–657. https://doi.org/10.1145/3600006.3613175

[18] Sylvain Jeaugey. 2019. Massively Scale Your Deep Learning Training with NCCL 2.4. NVIDIA Developer Blog. https://developer.nvidia.com/blog/massively-scale-deep-learning-training-nccl-2-4

[19] Myeongjae Jeon, Shivaram Venkataraman, Amar Phanishayee, Junjie Qian, Wencong Xiao, and Fan Yang. 2018. *Multi-tenant GPU Clusters for Deep Learning Workloads: Analysis and Implications (TR)*. Technical Report MSR-TR-2018-13. Microsoft.

[20] Xianyan Jia, Shutao Song, Wei He, Yangzihao Wang, Haidong Rong, Feihu Zhou, Liqiang Xie, Zhenyu Guo, Yuanzhou Yang, Liwei Yu, et al. 2018. Highly scalable deep learning training system with mixed-precision: Training imagenet in four minutes.

[21] De-chun Kong and Hui-bin Shi. 2015. Research and Design of 2D-Mesh Network-on-Chip Based on Multi-core System. *Microelectronics & Computer* 32, 4 (2015), 75–78.

[22] Alex Krizhevsky. 2012. *Learning Multiple Layers of Features from Tiny Images*. Technical Report. University of Toronto. https://www.cs.toronto.edu/~kriz/learning-features-2009-TR.pdf

[23] Alex Krizhevsky. 2014. One weird trick for parallelizing convolutional neural networks.

[24] Wenda Li, Lei Yu, Yuhuai Wu, and Lawrence C. Paulson. 2021. IsarStep: a Benchmark for High-level Mathematical Reasoning. arXiv:2006.09265 [cs.LO]

[25] Zhuang Liu, Hanzi Mao, Chao-Yuan Wu, Christoph Feichtenhofer, Trevor Darrell, and Saining Xie. 2022. A ConvNet for the 2020s. In *2022 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*. IEEE, New Orleans, LA, USA, 11966–11976. https://doi.org/10.1109/CVPR52688.2022.01167

[26] Andrew L. Maas, Raymond E. Daly, Peter T. Pham, Dan Huang, Andrew Y. Ng, and Christopher Potts. 2011. Learning word vectors for sentiment analysis. In *Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies - Volume 1* (Portland, Oregon) *(HLT '11)*. Association for Computational Linguistics, USA, 142–150.

[27] 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)*. USENIX Association, Santa Clara, CA, 289–304. https://www.usenix.org/conference/nsdi20/presentation/mahajan

[28] Hongzi Mao, Mohammad Alizadeh, Ishai Menache, and Srikanth Kandula. 2016. Resource Management with Deep Reinforcement Learning. In *Proceedings of the 15th ACM Workshop on Hot Topics in Networks* (Atlanta, GA, USA) *(HotNets '16)*. Association for Computing Machinery, New York, NY, USA, 50–56. https://doi.org/10.1145/3005745.3005750

[29] Stephen Merity, Caiming Xiong, James Bradbury, and Richard Socher. 2016. Pointer Sentinel Mixture Models. arXiv:1609.07843 [cs.CL]

[30] Hiroaki Mikami, Hisahiro Suganuma, Yoshiki Tanaka, Yuichi Kageyama, et al. 2018. Imagenet/resnet-50 training in 224 seconds. , 770–778 pages.

[31] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. 2013. Playing atari with deep reinforcement learning.

[32] Deepak Narayanan, Keshav Santhanam, Fiodar Kazhamiaka, Amar Phanishayee, and Matei Zaharia. 2020. Heterogeneity-aware cluster scheduling policies for deep learning workloads. In *Proceedings of the 14th USENIX Conference on Operating Systems Design and Implementation (OSDI'20)*. USENIX Association, USA, Article 27, 18 pages.

[33] Deepak Narayanan, Mohammad Shoeybi, Jared Casper, Patrick LeGresley, Mostofa Patwary, Vijay Korthikanti, Dmitri Vainbrand, Prethvi Kashinkunti, Julie Bernauer, Bryan Catanzaro, Amar Phanishayee, and Matei Zaharia. 2021. Efficient large-scale language model training on GPU clusters using megatron-LM. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (St. Louis, Missouri) *(SC '21)*. Association for Computing Machinery, New York, NY, USA, Article 58, 15 pages. https://doi.org/10.1145/3458817.3476209

[34] Jiquan Ngiam, Aditya Khosla, Mingyu Kim, Juhan Nam, Honglak Lee, and Andrew Y. Ng. 2011. Multimodal deep learning. In *Proceedings of the 28th International Conference on International Conference on Machine Learning* (Bellevue, Washington, USA) *(ICML'11)*. Omnipress, Madison, WI, USA, 689–696.

[35] Misja Nuyens and Adam Wierman. 2008. The foreground–background queue: a survey. *Performance evaluation* 65, 3-4 (2008), 286–307.

[36] 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.

[37] James L Peterson and Abraham Silberschatz. 1985. *Operating System Concepts*. Addison-Wesley Longman Publishing Co., Inc., Reading, MA, USA.

[38] 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). USENIX Association, Santa Clara, CA, USA, 1–18. https://www.usenix.org/conference/osdi21/presentation/qiao

[39] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, Ilya Sutskever, et al. 2019. Language models are unsupervised multitask learners. *OpenAI blog* 1, 8 (2019), 9.

[40] Ilija Radosavovic, Raj Prateek Kosaraju, Ross Girshick, Kaiming He, and Piotr Dollár. 2020. Designing network design spaces. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*. IEEE, Seattle, WA, USA, 10428–10436.

[41] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J Liu. 2020. Exploring the limits of transfer learning with a unified text-to-text transformer. *Journal of machine learning research* 21, 140 (2020), 1–67.

[42] Rehenuma Tasnim Rodoshi, Taewoon Kim, and Wooyeol Choi. 2020. Deep reinforcement learning based dynamic resource allocation in cloud radio access networks. In *2020 International Conference on Information and Communication Technology Convergence (ICTC)*. IEEE, IEEE, Jeju Island, South Korea, 618–623.

[43] Mark Sandler, Andrew Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. 2018. MobileNetV2: Inverted Residuals and Linear Bottlenecks. In *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition*. IEEE, Salt Lake City, UT, USA, 4510–4520. https://doi.org/10.1109/CVPR.2018.00474

[44] Victor Sanh, Lysandre Debut, Julien Chaumond, and Thomas Wolf. 2019. DistilBERT, a distilled version of BERT: smaller, faster, cheaper and lighter.

[45] David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, et al. 2016. Mastering the game of Go with deep neural networks and tree search. *nature* 529, 7587 (2016), 484–489.

[46] Karen Simonyan and Andrew Zisserman. 2014. Very deep convolutional networks for large-scale image recognition.

[47] Rajeev Thakur, Rolf Rabenseifner, and William Gropp. 2005. Optimization of collective communication operations in MPICH. *The International Journal of High Performance Computing Applications* 19, 1 (2005), 49–66.

[48] Qizhen Weng, Lingyun Yang, Yinghao Yu, Wei Wang, Xiaochuan Tang, Guodong Yang, and Liping Zhang. 2023. Beware of Fragmentation: Scheduling GPU-Sharing Workloads with Fragmentation Gradient Descent. In *2023 USENIX Annual Technical Conference (USENIX ATC 23)*. USENIX Association, Boston, MA, 995–1008. https://www.usenix.org/conference/atc23/presentation/weng

[49] 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. 2018. Gandiva: Introspective Cluster Scheduling for Deep Learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. USENIX Association, Carlsbad, CA, 595–610. https://www.usenix.org/conference/osdi18/presentation/xiao

[50] Yihao Zhao, Xin Liu, Shufan Liu, Xiang Li, Yibo Zhu, Gang Huang, Xuanzhe Liu, and Xin Jin. 2023. Muxflow: Efficient and safe gpu sharing in large-scale production deep learning clusters.