Cuckoo: Deadline-Aware Job Packing on Heterogeneous GPUs for DL Model Training

Yuzheng Zhang[1], Renyu Yang[1†], Junhong Liu[1], Weihan Jiang[1], Tianyu Ye[1], Yiqiao Liao[2], Penghao Zhang[2], Tiezi Zhang[2], Kun Shang[2], Tianyu Wo[1], Chunming Hu[1], Chengru Song[2], Jin Ouyang[2]
[1]Beihang University [2]Kuaishou Inc.

ABSTRACT

The growing scale and heterogeneity of GPU clusters pose new challenges to deep learning (DL) job scheduling. While existing schedulers primarily focus on GPU utilization, they often ignore multi-dimensional resource demands of DL workloads and lack precise execution time estimation for co-located jobs. While Muri pioneered the use of interleaving execution to improve resource efficiency, it simplified interference when jobs using one resource simultaneously and is agnostic to the deadline constraints. The job grouping also comes to suboptimal when heterogeneous GPU devices are taken into account. In this paper, we propose Cuckoo, a scheduling system that packs deep learning jobs with stringent deadline requirements over a set of heterogeneous GPU devices where multi-dimensional resources are interleaved and shared by a group of jobs. Specifically, the interleaving execution of simultaneous jobs is characterized and modeled through stage-grained execution time estimation considering the runtime performance interference and the impact of GPU heterogeneity on the job performance. The job packing is formulated as a multi-objective optimization problem which is then solved by the maximum weight matching algorithm. Cuckoo then allocates heterogeneous resources to the packed job groups through a graph-based maximum flow and minimum cut algorithm. Experiments show that Cuckoo improves deadline satisfaction by 2.38x and reduces average job completion time (JCT) by 1.81x compared with the state-of-the-art approaches. Cuckoo is implemented based on Kubernetes and has been deployed in Kuaishou to serve thousands of model training jobs that can be interleaved on shared heterogeneous GPU clusters.

CCS CONCEPTS

• **Computer systems organization → Cloud computing**; • **Software and its engineering → Distributed systems organizing principles**.

KEYWORDS

GPU Scheduling, Job Deadline, Deep Learning Training

Corresponding Author: Renyu Yang (renyuyang@buaa.edu.cn).

SoCC '25, November 19–21, 2025, Online, USA

ACM ISBN 979-8-4007-2276-9/25/11…$15.00
https://doi.org/10.1145/3772052.3772266

ACM Reference Format:
Yuzheng Zhang, Renyu Yang, Junhong Liu, Weihan Jiang, Tianyu Ye, Yiqiao Liao, Penghao Zhang, Tiezi Zhang, Kun Shang, Tianyu Wo, Chunming Hu, Chengru Song, Jin Ouyang. 2025. Cuckoo: Deadline-Aware Job Packing on Heterogeneous GPUs for DL Model Training. In *ACM Symposium on Cloud Computing (SoCC '25), November 19–21, 2025, Online, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3772052.3772266

1 INTRODUCTION

Recent advancements in deep learning (DL) have boosted intelligent applications and services. Building multi-tenant GPU clusters at scale has become the de facto means to train DL jobs. The resource scheduler typically needs to schedule hundreds of thousands of DL jobs on thousands of GPU accelerators, with specific scheduling objectives, such as improving scheduling throughput and reducing job training time. Prevailing GPU scheduling strategies [8, 11, 16, 19, 28, 33, 35, 38] predominantly focus on GPU allocation where GPU is sole resource dimension to schedule. However, they are increasingly inadequate given the diversity of modern DL models, whose varying sizes and computational characteristics create significant demand for other resources, including CPU, disk I/O, and network bandwidth, throughout the training process [37, 40].

In modern GPU clusters, both jobs and accelerating devices exhibit diversity and heterogeneity. The ever-increasing advancement in GPU architectures boosts the holistic compute capabilities, memory bandwidth, and communication performance. Model training jobs become even diverse and have noticeable stage-based resource usage patterns. For instance, the performance of reinforcement learning models such as A2C [20] are bounded by data loading, while Transformer-based models such as GPT-4 [24] are GPU-intensive. Existing works such as Gavel [22] and Sia [13] have been proposed to schedule DL workloads in heterogeneous environments. However, they typically incur additional overheads stemming from time-slice switching and are unaware of job-specific QoS requirements.

In practice, jobs have distinct timing requirements – execution deadline – as per business types and priorities. For example, over 90% jobs in Kuaishou are specified with expectation of execution deadline. High-priority jobs such as production-grade recommendation models usually have such stringent deadline requirements that the cluster scheduler needs to first guarantee against other jobs. While some schedulers [7, 15] are designated to guarantee deadline satisfaction, it is intricate to integrate them with resource-sharing strategies, due to the huge combinatorial explosion in scheduling space. Therefore, it is highly desirable to co-optimize the GPU allocations taking into account the deadline requirement and GPU heterogeneity when packing and scheduling multiple jobs that share interleaved multiple dimensional resources.

Muri[40] is among the first attempts in interleaving multiple dimensional resources and grouping DL jobs with complementary resource demands, thereby making the best use of all types of idle resources across different stages of model training. Nevertheless, Muri oversimplifies the execution pattern – each training iteration partitioned into multiple resource-exclusive stages and individual resource is simultaneously used without interference. In reality, interference arising from asynchronous resource invocation and overlapping execution of stage usually leads to non-negligible inaccuracy in runtime estimation and thus undermines its effectiveness of job packing and utilization improvement.

In this paper, we present Cuckoo, a scheduling system that cam effectively pack DL training jobs, in the form of job groups, and schedule them on heterogeneous resources to improve overall system efficiency whilst conforming to deadline constraint of job completion time (JCT). The key insight is to estimate the execution efficiency of a colocated group of jobs through elaborating an estimation model that accounts for runtime interference. Based on this estimation, we then resolve an optimization problem to elaborate a concrete plan for job packing and execution, ensuring that the heterogeneous GPUs are utilized optimally.

To do so, Cuckoo employs a profiling-based approach to calculating the execution time of interleaved job groups, delicately considering the overlapping resource usage and the resultant interference derived from asynchronous resource invocation when jobs are co-executed. Based on the timing information, the deadline-aware resource scheduling encompasses two phases. i) in the *job packing phase*, we resolve a maximum weight matching problem to generate high-efficiency job grouping plan. A weighted graph is established where nodes represent jobs and edge weights indicate the estimated execution efficiency of potential job groups. ii) in the *resource scheduling phase*, We leverage bipartite matching mechanism for mapping job groups to the available GPU-time slots considering GPU heterogeneity. Each edge is weighted based on the predicted execution efficiency and deadline satisfaction of a job group on a specific GPU type and time slot. A maximum flow minimum cut algorithm is then adopted to pinpoint the optimal solution that enables efficient and deadline-aware job execution on heterogeneous GPU devices.

Cuckoo is implemented based on Kubernetes and has been deployed in Kuaishou to serve thousands of model training jobs on shared heterogeneous GPU clusters. We conduct end-to-end experiments on a heterogeneous cluster. Experimental results show that Cuckoo can reduce the job completion time (JCT) by 1.81x and improve the deadline satisfaction rate by 2.38x, compared with the state-of-the-art scheduling strategies.

This paper makes the following contributions:

- A Kubernetes-based GPU scheduling architecture that underpins multi-job interleaving and resource sharing.
- A model to quantify the throughput and execution efficiency of a group of jobs simultaneously executed on interleaving resources.
- A two-phased scheduling policy for packing suitable jobs into job groups and allocating heterogeneous GPUs and other resource dimensions. Cuckoo aims to accelerate the job execution, by fully exploiting the deadline constraint and the performance sensitivity of job groups on different GPUs.

Organization. The paper is organized as follows. §2 presents the background and motivation and §3 presents the system overview and the enabling techniques. §4 and §5 discuss the experimental results and industrial experience, respectively. We present general discussion of system design and related work in §6 and §7, before concluding the paper in §8.

2 BACKGROUND AND MOTIVATION

2.1 DL Job Scheduling

Recent studies aim to improve resource utilization by co-locating DL jobs [2, 4, 9, 14, 26, 34, 38, 40]. Nevertheless, efficiently packing and scheduling DL jobs in multi-tenant clusters is challenging due to the significant diversity in model architecture, iteration length, and communication-to-computation ratio. DL jobs consume multiple types of resources, such as disk I/O, CPU, GPU, and network in a training iteration. Each iteration is typically divided into multiple stages according to the dominant resource. Although this allows for simultaneous usage of different resources across jobs, jobs compete for multiple resources (CPU, GPU, memory, I/O, network), and the interleaving degree across multi-stages leads to intricate interference patterns that are difficult to accurately predict. Existing scheduling algorithms that aim to satisfy both resource sharing and deadline constraints typically assume a homogeneous setting and lack awareness of heterogeneity.

2.2 Characteristics and Challenges

Job packing – the practice of co-locating multiple jobs on a single compute node to share multi-dimensional resources – is a well-established technique for improving cluster utilization in HPC and data centers [37]. However, applying this method to industry-scale AI infrastructures, hosting diverse workloads such as model training and serving, presents significant yet unresolved challenges.

Heterogeneity of GPU Devices. Production-grade clusters often consist of multiple heterogeneous GPUs and different GPU types vary in compute capability, memory bandwidth, and interconnect performance. As illustrated in Fig. 1, there is a noticeable deviation in performance sensitivity across heterogeneous devices when running different DL jobs on different GPU types. Such heterogeneity introduces additional complexity of scheduling jobs, as GPU heterogeneity unleashes a huge potential for optimizing resource allocation by matching the diverse workloads and the multi-staged tasks with the most suitable and cost-efficient accelerators.

As shown in Fig. 2, consider a simple example of scheduling two jobs VGG (1000 steps) and ResNet (400 steps) onto two GPU types NVIDIA A800 and RTX 4090. Placing VGG on A800 and ResNet on RTX 4090 can accelerate the job completion by 20%. Omitting the heterogeneous nature can lead to low utilization of heterogeneous hardware and reduced overall system efficiency.

Job-Specific Execution Deadline. In real-world production environments, DL jobs often come with job-specific QoS requirements such as the execution deadline, where training tasks must be completed and deployed before a specific time to avoid monetary losses. According to a recent survey [6], more than half of the participants expressed expectations about the completion time of training jobs. In our industrial cases, jobs often have distinct deadline requirements as per business types and priorities. For example, in

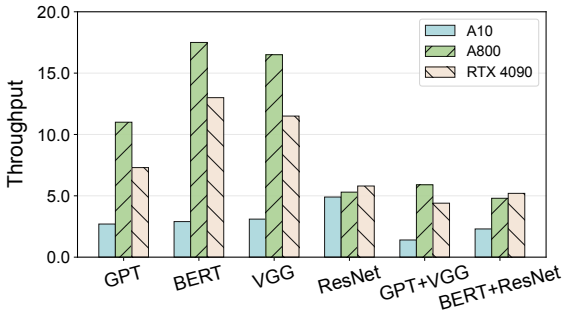


Figure 1: Efficiency of different jobs on different GPU types.

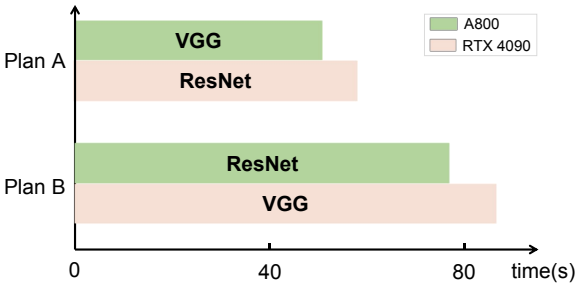


Figure 2: Exploiting heterogeneity to reduce execution time.

Table 1: Performance comparison using different packing strategies.

Job Packing Strategy	Avg. JCT (s)	Deadline Satisfaction Rate
Non-packing	3,953	49.14%
Muri	3,689	39.16%

Kuaishou, over 90% jobs are specified with an expectation of execution deadline. Jobs with high priority such as production-grade recommendation models usually have even more stringent deadline requirements that the cluster scheduler needs to first guarantee against other jobs.

Fig. 3 shows an intuitive example of considering deadline constraints in the job packing and scheduling. Assume a user specifies job A and job B with early deadline, job C and job D with late deadlines. Packing A with C and B with D yields higher execution efficiency, compared with the packing plan of A-B and C-D. However, an efficiency-focused packing strategy would cause job B to miss its deadline due to a hard timing constraint. As shown in Table 1, we conduct experiments to show how efficiency-aware approaches perform in terms of deadline misses. We choose Muri as the representative approach that prioritizes the job packing efficiency and compare against the naive approach without job packing (i.e., scheduling a job at a time). Obviously, deadline satisfaction is compromised for reduced JCT reduction and improved utilization. This issue is further exacerbated by taking into account the sensitivity of job execution to GPU heterogeneity.

Time Estimate for Interleaved Execution. Some recent work [40] makes an attempt in delivering interleaved execution model to reduce resource fragmentation and improve overall utilization. As shown in Fig. 4a, the existing solution schedules the same stage

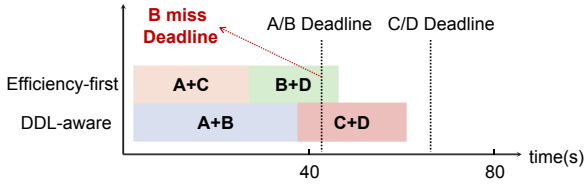
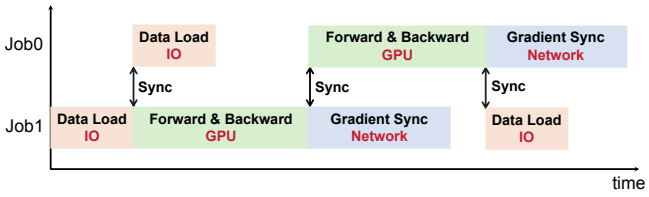
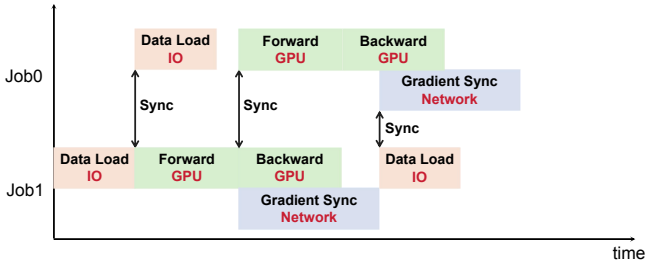


Figure 3: Efficiency-first vs. Deadline-aware Job Packing Example.



(a) The existing solution



(b) Real-world observation

Figure 4: Example of interleaving two jobs in ideal/realistic settings.

Table 2: The error of duration estimate by Muri on different GPUs.

GPU Type	A800	A10	4090	Total
Avg. Error	31.8%	13.9%	14.8%	20.2%

of different jobs in different time periods, and the duration is predetermined by the maximum resource occupancy of each corresponding stage. This is based on a simple assumption that the running stages are completely independent, without any time overlapping or resource sharing. However, this assumption departs from the actual observation when running multiple co-running jobs. As depicted in Fig. 4b, in real-world scenarios, communication begins right after forward propagation, concurrently at the event of backward propagation, due to the asynchronous nature of resource invocation and the inherent overlap between computation and communication. One single resource can be used across multiple stages and concurrent resource usage during the same period leads to interference, which in turn leads to much longer time duration.

Table 2 summarizes Muri's execution time estimation errors when running a mix of DL models, such as DQN [21], GPT-2 [24], etc., in the heterogeneous GPU cluster. We evaluate Muri's accuracy by comparing its predicted job iteration durations against actual measurements collected from real executions. The error is quantified using normalized absolute error. Across multiple job combinations and GPU types, the average estimation error of Muri reaches 20.2%, with some cases exceeding 30%. This indicates that the execution model of multiple job interleaving must take into

account the concurrent resource usage and the resultant resource contention across jobs.

Research Requirements. The emerging challenges necessitate a proper job packing scheme that can co-optimize deadline satisfaction and multi-dimensional resource utilization, and make the best use of device heterogeneity. Joint optimization, however, inevitably leads to an exponentially growing search space as the job number and the cluster scale increase. Therefore, it is paramount to design an efficient and practical scheduling framework in the face of heterogeneity and QoS satisfaction.

3 OUR APPROACH

3.1 System Overview

Cuckoo is designed to enable efficient scheduling of DNN training jobs on a shared GPU cluster. To reduce the JCT and promote the deadline satisfaction ratio, taking into account the temporal-spatio factors such as deadline constraints at per job level and the resource efficiency at cluster level. Figure 5 depicts an overview of Cuckoo design. Cuckoo advances Muri [40], the state-of-the-art multi-resource cluster scheduler for DL workloads. The jobs submitted by the users into job groups, which will be then assigned to, and interchangeably utilize a certain set of resources, including CPU, GPU, network bandwidth, and disk IO. At the core of Cuckoo is to select the most appropriate GPU type for a given job group and fulfill multi-resource sharing through interleaved job execution and exploiting heterogeneous GPUs. This enables DL jobs to share the same set of resources and use the correct GPUs depending on the characteristics of the jobs which improves resource utilization, meets job deadline requirements, and reduces JCT.

The key components of Cuckoo are as follows.

- **Job Profiler**. When a user submits the job to Cuckoo, Job Profiler will conduct job dry-runs for several steps. It records the duration in one iteration of various resources (e.g., disk IO, CPU, GPU, and network communication) on different GPUs, and the overlapping duration of communication and calculation. Jobs that complete the profiling process are enqueued, waiting for scheduling. To reduce the system overheads, if the profiling record of the same job exists, the historical data can be directly reused. Duration of resource usages will be provisioned to Job Packer for estimating the running efficiency of a pack of jobs that interleavingly co-utilize the same set of multiple resources. Technical details will be given in §3.2.
- **Job Packer**. Job Packer pops up jobs from the waiting queue and group those jobs into several packs (groups) by jointly considering execution efficiency, deadline constraints, and hardware heterogeneity. It estimates the potential throughput of a job group on various GPU types using the resource usage duration collected by Job Profiler. Job Packer combines jobs with complementary resource demands to maximize overall throughput, whilst ensuring that the job group conform the deadline requirements and avoid combinations likely to cause deadline violations. It also considers the performance differences across heterogeneous GPU types to generate job groups that are both efficient and heterogeneity-sensitive. We detail the key technique in §3.3.3.
- **Resource Scheduler**. Resource Scheduler allocates cluster resources to the the packs of jobs generated by Job Packer, to minimize the scheduling cost whilst meeting job deadlines. It evaluates each job group by estimating its completion time under various placement options and computes a holistic cost encompassing the expected execution time and the penalty for potential deadline violations. The scheduler selects the optimal placement for each group, according to the cost calculation. The decision-making takes into account all available GPU types, resource availability over time, and each job group's executing efficiency on different hardware. If a job group can be only scheduled by violating constraints, the associated jobs will be repopulated into the queue for future scheduling attempts. The details are given in §3.3.3.

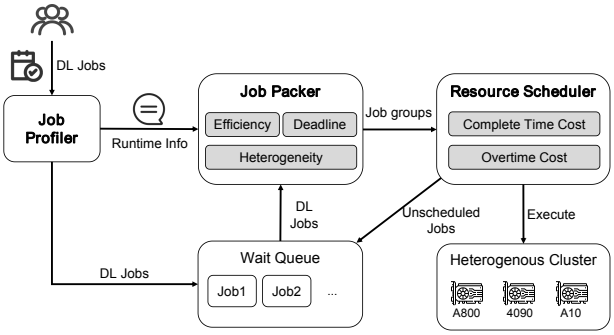


Figure 5: Overview of Cuckoo.

3.2 Executing Time and Efficiency

This section details how to estimate the interleaving execution of a job pack, considering the characteristic of asynchronous invocation and intrinsic performance interference.

3.2.1 Asynchronous Behavior in DL Execution. In actual DL training process, DL frameworks implement mechanisms such as asynchronous invocation of resources and overlapping between communication and computation. The GPU calculating during forward and backward propagation, as well as the network communication for gradient synchronization, are asynchronous with the main thread. Usually the main thread calls the GPU to perform calculation operations when starting perform forward propagation. When the backward propagation starts, the network communication begins via the callback of backward propagation to complete the gradient synchronization process. Besides, affected by the model characteristics and implementation, GPU execution and network communication only need to be synchronized in specific stages.

For example, some models (e.g., BERT, GPT) will synchronize the communication process before the gradient synchronization stage, and other models (e.g., VGG, ResNet) only need to synchronize the GPU execution and network communication process before the forward propagation in the next iteration. Since the GPU execution and network communication are not completely consistent with the corresponding stages of main thread, they can be carried out across multiple main thread stages. As discussed in § 2.2, Muri simply assumes that only one resource is used in each stage (e.g., only GPU is used in forward and backward propagation) of a training iteration. Resources considered to be used only in one stage may

Algorithm 1 Calculate One Iteration Duration

```
Function EstimateIterationDuration(F, B, C, S)
f, b, c, R ← 0
for i do
    R_i ← GetStageDuration(f, b, c, S)
    // If two consecutive iterations are sufficiently close
    if isCloseToLastIteration(R) then
        return R_i + R_{i-1} + R_{i-2} + R_{i-3}
    end if
end for
```

appear in other stages so that the actual execution time should be better profiled and modeled.

3.2.2 Inter-Job Interference. Under these conditions, estimating the execution time of a group of interleaved jobs requires additional consideration of the interference caused by the same resources used by different jobs in the same stage. The resource interference will cause the execution time of some resources to be longer. If different jobs use the same resource during the same stage, it is necessary to multiply the time periods existing overlapping resource usage by an interference factor λ to obtain an estimate of corresponding resource time. As CPU and disk I/O resources are typically sufficient, we don't need to consider interference of these two resources. Usually, data loading and preprocessing is synchronized with the corresponding main thread stage. Hence, we focus on the interference caused by the competition of GPU and network resources.

3.2.3 Estimating Execution Time. To resolve this problem, the key insight is to exploit the characteristic of asynchronous invocation model (AM) for estimating the temporal usage patterns of various resources and capturing the performance interference caused by their asynchronous and overlapping invocation. To model the execution time of interleaved jobs, we divide an iteration of DL training into three stages: data loading, GPU execution, and network communication based on the DL training process.

We employ dynamic programming algorithms to simulate the time that the job group spends on each stage. Dynamic programming (DP) is chosen because the interleaving of multiple jobs introduces complex temporal dependencies between stages. For example, overlapping communication and computation across different jobs. DP allows us to efficiently explore these dependencies. Compared with heuristic models, DP provides a balance between modeling accuracy and computational efficiency: while exact simulation of all interleaving combinations would be intractable, DP captures the major interaction patterns without incurring prohibitive computational cost.

Algorithm Details. Once the duration of the same stage between two iterations is stable, we can take the duration of three consecutive stages as an estimate for the execution time of the job group in one iteration.

Alg. 1 shows the pseudocode of the algorithm. To estimate the execution time of the one stage, we define the job set J and R_i as the total duration of stage i. We defined F_j, B_j, C_j respectively as the duration of forward propagation, backward propagation and network communication of job j in one iteration. We define $S_{i,j}$

Algorithm 2 Calculate One Stage Duration

```
Function GetStageDuration(f, b, c, S)
// Find the maximum duration in this stage
for j ∈ J do
    R ← max(R, S_j)
    // If in GPU exec stage, update the remaining time for f, b, c
    if isGPUExecutionStage(i, j) then
        f_{i,j}, b_{i,j}, c_{i,j} ← f_{i,j} + F_j, b_{i,j} + B_j, c_{i,j} + C_j
    end if
    // If resources require synchronization with the main thread, calculate
    the sync time
    if isSynchronized(i, j) then
        R ← max(R, syncTime(f, b, c, i, j))
    end if
end for
// Durations of concurrent resource usage are adjusted by a contention
coefficient
f_i, b_i, c_i, R_i ← resourceCompetition(f_i, b_i, c_i, R_i)
for j ∈ J do
    // Add the resource time that cannot be completed in the current stage
    to the next stage
    f_{i+1,j}, b_{i+1,j}, c_{i+1,j} ← NextStage(f_{i,j}, b_{i,j}, c_{i,j}, R)
end for
return R
```

Table 3: Job configurations and resource usage time (unit: ms).

	Model	Order	S	F	B	C
Job 0	ResNet	0	10	37	76	98
Job 1	BERT	1	10	72	61	363

as the duration of function invocation for job j in stage i. In the data loading stage, resource usage time is considered as $S_{i,j}$. $f_{i,j}$, $b_{i,j}$, $c_{i,j}$ represents the forward propagation, backward propagation, communication time that job j should spend in stage i.

When estimating the time R_i for a particular stage of job group J, we iterate through the job set J to search for R_i. At minimum, the time required for this stage includes the duration of main thread function invocation. If job j is in the GPU execution stage, the resource time for F_i, B_i, and C_i need to be appended to f_i, b_i, and c_i for this round, respectively. If job j requires synchronizing a certain resource within the stage i, we calculate the duration required to synchronize it and select the maximum value as the execution time for the current stage. If there is resource interference in the current stage, it is necessary to multiply the interference period by the interference coefficient and update the time of resources. After R_i is estimated, we limit the time of f_i, b_i, c_i within the range of R_i. The time of $f_{i+1,j}$, $b_{i+1,j}$, and $c_{i+1,j}$ for the subsequent stage is computed as the remaining execution time of last iteration. pseudocode is shown in Alg. 2.

Working Example. We provide an example involving two interleaved jobs to showcase how the algorithm works. Table 3 summarizes the characteristics of each job, including the model type and the measured duration of various resource stages. We suppose the interference coefficient of kernel and communication are 2. Forward propagation needs to be synchronized during the GPU computing stage, BERT needs to complete gradient synchronization before the

Table 4: Resource usage per stage in interleaved execution (unit: ms).

i	f_0	b_0	c_0	f_1	b_1	c_1	R
0	0	0	0	0	0	0	10
1	0	0	0	72	0	0	72
2	74	152	0	0	122	363	363
3	0	0	98	0	0	0	98
4	0	0	0	0	0	0	10
5	0	0	0	72	0	0	72

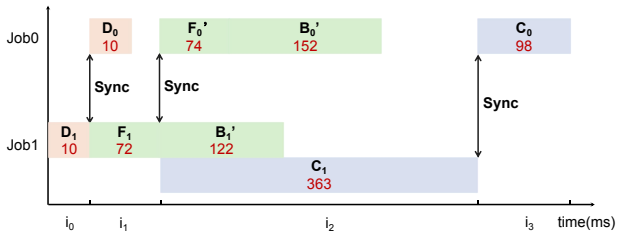


Figure 6: Execution Timeline. F_0', B_0', B_1' indicate stages with resource interference, where overlapping usage of GPU resources prolongs the actual duration.

communication stage, while ResNet needs to complete before the GPU computing stage of next iteration. We list the resource usage time of each stage in Table 4 to show the progress.

In stage i_0, j_1 execute data load, and other resources are not used. Therefore, R_0 is 10ms, same as S_0. In stage i_1, j_1 performs GPU computing, and job 0 starts data loading. j_1 needs to wait 72ms for the forward propagation computing completed. Due to no resource interference between jobs, R_1 is the larger time between $S_{1,1}$ and $f_{1,1}$. In stage i_2, j_1 performs back propagation and gradient synchronization and j_0 performs forward propagation. j_0 needs to complete the forward propagation while j_1 is gradient synchronization in this stage. Since both jobs need to use GPU and communication resources, duration that resource are used simultaneously needs to multiply by the interference coefficient. The communication used by j_1 is the longest in this stage and we obtain R_2 of 363ms. In stage i_3, j_0 performs gradient synchronization without resource contention. Thus, R_3 is simply the communication time of job j_0, which consumes 98ms. Similarly, we can get R_4 of 10ms, R_5 of 72ms. Observably, the execution time converges and one iteration time is 543ms in total. The timeline is illustrated in Fig. 6.

3.2.4 Estimating Execution Efficiency. We measure the execution efficiency eff_value = of a job group interleaved execution:

$$eff\_value = \frac{\sum_{j \in jobs} Duration(j)}{InterleavingDuration(jobs)} \tag{1}$$

It is defined as the ratio between the sum of the standalone execution durations of the individual jobs and the total required duration when the jobs are executed interleavingly. A higher eff_value indicates better utilization of multi-dimensional resources during interleaved execution.

3.3 Job Packing and Resource Scheduling

3.3.1 Problem Definition. We formulate a multi-objective optimizing problem to achieve the optimal solution in terms of resource usage, JCT, deadline satisfaction rate, etc:

$$\textbf{minimize } V \tag{2}$$

$$\textbf{s.t. } \forall j_1, j_2 \in J: \ g_{j_1} = g_{j_2} \rightarrow m_{j_1} = m_{j_2} = m_g \tag{3}$$

$$\forall g \in G, \forall k \in K: \ X_{gk} = \{0, 1\} \tag{4}$$

$$\forall t, \forall k \in K: \left(\sum_{g \in G, s_g < t < e_g} m_g X_{gk} \right) \leq M_k \tag{5}$$

$$\forall j \in J: \ a_j \leq s_{g_j} \tag{6}$$

$$\forall g \in G, \forall k \in K: \ s_g + T_{gk} X_{gk} \leq e_g \tag{7}$$

$$V_1 = \frac{1}{N} \sum_{j \in J} \left(e_{g_j} - a_j \right) \tag{8}$$

$$V_2 = \frac{1}{N} \sum_{j \in J} \left[e_{g_j} \geq d_j \right] \tag{9}$$

$$V = \alpha V_1 + \beta V_2 \tag{10}$$

Specifically, j,g,k represent a job, a job group and a kind of GPU respectively. Correspondingly, J, G, K are the sets that contain such elements. The size of job set J is N. Job j have the following attributes: arriving time a_j, deadline d_j, requested GPU number m_j. Job group g has these properties: start time s_g and end time e_g. M_k denotes the collection of GPU devices belonging to the type k in the cluster. T_{gk} represents the elapsed time of group g on GPU type k. The holistic objective consists of average job complete time V_1 and job overtime rate V_2 with weight. α, β are their weight coefficient.

The problem has the following constraints – Eq. 3 restricts that all jobs in a group request the same number of GPU. Eq. 4 uses a matrix to express GPU allocation. Each group can only execute on one kind of GPU. Eq. 5 restricts executing groups in cluster can not use more GPUs than that the cluster provides. Eq. 6 restricts jobs must arrive before executing. Eq. 7 describes jobs' executing time equals the executing time of group they belong to. Eq. 8 and Eq. 9 calculate the average job complete time and job overtime rate.

The algorithmic solution is with two stages – *job packing* to generate executable job groups and *resource scheduling* to make execution planning and allocate resources.

3.3.2 Job Packing. While following similar steps as Muri[40], we use different means to process the input data. Specifically, unlike Muri, which relies solely on interleaving efficiency, our approach jointly accounts for efficiency and deadline constraints in heterogeneous environments, dynamically determining the optimal packing strategy.

Balancing Execution Efficiency and Deadline. We propose to group jobs with similar deadlines to improve scheduling flexibility. To quantify the temporal proximity of deadlines within a group, we define a metric called ddl_value, which represents the ratio of overlapping time between the current moment and the job deadlines. For a set of jobs, ddl_value is computed as follows:

$$ddl\_value = \frac{\min_{j \in jobs} D_j - now}{\max_{j \in jobs} D_j - now} \tag{11}$$

where D_j denotes the deadline of job j, and now represents the current system time. A higher ddl_value indicates that the jobs have much closer deadlines, making them more suitable to be packed together, from the scheduling perspective.

Algorithm 3 Job Packing Algorithm

```
// Part 1
G ← newGraph()
// J includes all jobs to be packed
for j ∈ J do
    G.addNode(j)
end for
for each pair (u, v) ∈ G.nodes() do
    weight ← ComputeTotalValue(u, v)
    G.addEdge(u, v, weight)
end for
// Part 2
M ← G.ComputeMaximumWeightMatching()
pack_result ← newList()
// Pack each matched job pair into a group
for each pair (u, v) ∈ M do
    p ← PackJob(u, v)
    pack_result.add(p)
end for
// Pack unmatched jobs into individual groups
for unmatched node u ∈ M do
    p ← PackJob(u)
    pack_result.add(p)
end for
return pack_result
```

To take deadline into account in job packing, we extend Muri's original efficiency-only equation by introducing a combined scheduling value $total\_value$, which integrates both efficiency and deadline constraint.

$$total\_value = w \cdot eff\_value + (1 - w) \cdot ddl\_value \quad (12)$$

Herein, $w \in [0, 1]$ is a tuneable weight that balances the importance of execution efficiency and deadline constraint. The resultant $total\_value$ serves as the edge weight during maximum weight matching in the job grouping, enabling the system to form job packs that are not only efficient in terms of resource usage, but also more likely to meet their deadline constraints.

Algorithm 3 shows the main procedure. We transform the jobs needed to pack into a graph, where each node stands for a job. The weight of edges will be calculated by Eq. 12. We resolve the problem of the maximum weighted matching by using Blossom Algorithm. If a node in the matching, we pack this job and its paired job in a group; Otherwise, the job will be packed in a group in a singular way.

Heterogeneity-Aware Packing. Compared with clusters composed of a single GPU type, job packing becomes significantly more complex in heterogeneous environments. This complexity arises primarily from the inconsistency in group execution efficiency across different GPU types. Moreover, even if GPU types are assigned during the job packing phase, there is no guarantee that the group will eventually be scheduled on the expected GPU type during execution. As a result, tightly coupling job packing with execution planning becomes impractical. Instead, we separate these two stages and implement heterogeneity-sensitive scheduling separately in the job packing and execution planning phases.

During job packing, we adopt the following practical strategy: If only one GPU type is available in the cluster at the moment, the job group is assumed to be scheduled on that GPU type, and its interleaving efficiency is calculated accordingly. If multiple GPU types are available, we assume the group will be placed on the GPU type that yields the highest estimated efficiency. In real-world deployments, GPU resources are typically under heavy contention, and it is common that only a single GPU type is available during scheduling. Therefore, although our estimation introduces an approximation in selecting the GPU type during efficiency evaluation, this uncertainty rarely materializes in high-load environments. Empirical results show that the impact of this approximation on overall evaluation accuracy is negligible.

In the subsequent execution planning stage, each job group is explicitly assigned to a specific GPU type. This assignment is determined by comparing the estimated execution time of the group across all available GPU types, and selecting the one that minimizes the overall schedule cost. This decision establishes the actual GPU type on which the group will run.

Table 5: Executing time and deadline for group.

	Time on GPU A	Time on GPU B	Deadline
Group 1	2	3	5
Group 2	3	4	5
Average	2.5	3.5	

3.3.3 Resource Scheduling and Execution Planning. We propose a scheduling algorithm to generate the execution planning for the job groups. As jobs with the same GPU number requirement can be packed together, we only consider situation that all of jobs only request one GPU. The execution plan can be represented the number of GPUs allocated to the job group and the execution order on the set of resources. Only two factors can influence a group's complete time – the GPU type the group uses and the time when the group of jobs starts. We assume all job groups spend the same execution time, the factors will be simplified to GPU type and the execution order of groups on the GPU. In this way, we can use the average execution time of all job groups to represent each group's real executing time.

The problem can be regarded as a matching problem, i.e., mapping the job groups to the pairs of (GPU number, execution order). If we assign each match a cost related to the scheduling objective such as deadline or JCT, we can get an ideal matching result by resolving a minimum cost matching problem. This matching problem can be described as a bipartite graph. Nodes in the bipartite graph can be split into two parts. One part includes group nodes. The other part includes GPU nodes (including execution order). All of edges in the graph span from group node to GPU node. For example, there are 2 groups (Group 1, Group 2) and 2 GPU (GPU A, GPU B). Execution order has range $\{1, 2\}$ for there will be 2 groups executing on one GPU at most. Shown in Figure 7 and Table 5, there are 2 nodes for groups and 2×2 nodes for GPUs.

The weight of edges in the graph is the cost, including *complete time* cost and *deadline* cost. We use the complete time of groups as complete time cost. This cost will have an impact on the average job complete time of the whole cluster. Herein is an edge from Group i to (GPU X, Order n). The average time running GPU X is T. Group

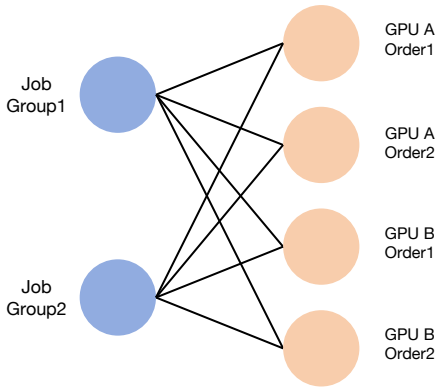


Figure 7: An Example of Bipartite Graph for Execution Planning.

Table 6: Weight for each edge.

Group	(GPU, Order)	C_1	C_2	$C_1 + C_2$
Group 1	GPU A, Order 1	2	0	2
	GPU A, Order 2	4.5	0	4.5
	GPU B, Order 1	3	0	3
	GPU B, Order 2	6.5	1.5	8
Group 2	GPU A, Order 1	3	0	3
	GPU A, Order 2	5.5	0.5	6
	GPU B, Order 1	4	0	4
	GPU B, Order 2	7.5	2.5	10

i need time t on GPU X. The complete time cost C_1 will be:

$$C_1 = (n-1) \times T + t \tag{13}$$

Second, we use "Time over deadline" as deadline cost. If a group finishes in time with a specific GPU-Order config, the deadline cost on this edge is 0. If the group described before has a deadline D, the deadline cost C_2 for the edge discussed before will be:

$$C_2 = \max(0, (n-1) \times T + t - D) \tag{14}$$

Example. The edges' weight for Figure 7 is shown in Table 6. Each edge in matching stands for a group's execution plan. Obviously, allocating Group A to GPU A Order 1 and Group 2 to GPU B Order 1 is the best matching, which can best satisfy our requirement.

3.3.4 Anti-Overpacking Mechanism. Job packing is not always beneficial to resource utilization. If total GPU demand of groups is less than the number of GPUs in the cluster, GPUs in the cluster may be wasted. In Muri's origin design, if scheduler encounters the problem, it will unpack groups with low effciency to use idle GPUs in the cluster. For example, there are 100 jobs and 70 GPUs. The scheduler will pack 100 jobs into 50 groups, with two jobs each group, and find 20 GPUs idle. Then it will unpack 20 groups into 40 new group with one jobs each group only. Finally there are 70 groups and 70 GPUs and no GPU left in the cluster.

However, this strategy fails in a heterogeneous cluster where GPU performance varies significantly—some GPUs can be two or even three times faster than others. Simply counting the number of GPUs cannot capture this performance disparity. For example, consider two groups running on two GPUs, A and B. On GPU A, Group 1 and Group 2 tak2 se and 3 s to complete, respectively, while on GPU B they take 6 s and 8 s. In this case, the optimal plan is to execute both groups on GPU A, even though the total GPU demand equals the number of available GPUs. As a result, GPU B remains idle, illustrating that "overpacking" can occur when GPU heterogeneity is ignored.

Table 7: GPU configuration of the experimental cluster.

# of Nodes	GPU Type per Node	# of GPUs per Node
1	NVIDIA A800	2
1	NVIDIA RTX 4090	8
3	NVIDIA A10	2

To handle the problem under the heterogeneous situation, we modify the method by adding weight for each kind of GPU. Set weight of GPUs with the worst performace to 1 and weight of other GPUs to the ratio compared with the worst one. Then get the sum of weight as the total GPU demand that groups need to reach. According to the execution time, if we set weight of GPU A to 3 and GPU B to 1 in example, we can get 4 groups in total. Three of them will execute on GPU A and the last one executes on GPU B. Although effciency through job packing is lower, the utilization of the whole cluster is higher.

4 EXPERIMENTS

4.1 Experimental Setup

Hardware and Software. The testing cluster consists of five physical servers, each equipped with an Intel Xeon Platinum 8352Y CPU @2.20GHz, 1TB of RAM, and GPU accelerators (detailed in Table 7). Cuckoo is implemented based on Kubernetes 1.17.4, PyTorch 1.11.0 with CUDA 11.5 support.

Methodology and Baselines. We conduct a comprehensive evaluation of Cuckoo from multiple perspectives. First, we compare it with Muri, a representative baseline system that supports job interleaving, to demonstrate the substantial performance of Cuckoo across overall metrics (§4.2). At a finer granularity, we evaluate the precision of interleaved job execution time prediction (§4.3), and further highlight the effectiveness of job packing algorithm (§4.4) and execution planning algorithm (§4.5). We also validated the effectiveness of the heterogeneity-sensitive design (§4.6). As fine-grained comparisons have different specific goals, we detail the related baselines in each subsection.

Workloads. We adopt the job submission pattern from the Helios trace [12] and randomly sample the model and batch size from Table 8, which includes a diverse mix of model types (CV, NLP, and RL) with different proportions of computation and communication. As illustrated in Figure 8, approximately 100 jobs are submitted within the first 1,000 seconds to fully utilize all GPUs, simulating a resource-constrained cluster environment. Thereafter, 10–40 new jobs are submitted every 1,000 seconds, resulting in a total of 521 jobs submitted within 30,000 seconds. The job deadline is generated in the following way:

$$\frac{deadline - arriving\_time}{executing\_time} \sim N(8, 2) \tag{15}$$

Table 8: Models and batch sizes used in evaluation.

Model Name	Type	Batch Size(s)
A2C [20]	RL	8, 16, 32
DQN [21]	RL	8, 16, 32
BERT [5]	NLP	2, 4, 8
GPT-2 [24]	NLP	2, 4
VGG19 [27]	CV	8, 16, 32
VGG16 [27]	CV	8, 16, 32
ResNet18 [10]	CV	16, 32, 64
ResNet50 [10]	CV	16, 32, 64

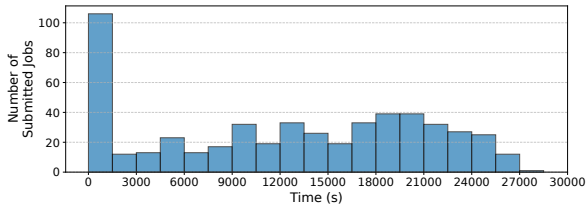


Figure 8: Job submission pattern derived from the Helios trace.

where *execution_time* refers to the fastest execution time of the job on all types of GPU. Each job is assumed to run on 2 GPUs.

Evaluation Metrics. To comprehensively evaluate the performance of Cuckoo, we adopt the following metrics:

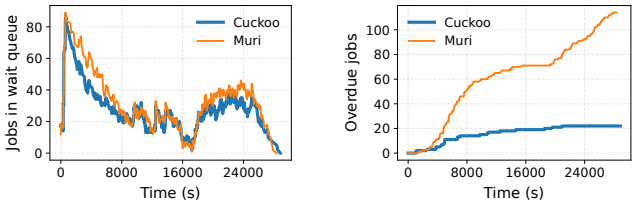
- **Job Completion Time (JCT) and Throughput**: The elapsed time from a job's submission to its completion. This metric reflects the overall efficiency and responsiveness of the scheduling system, encompassing both scheduling latency and actual execution time. We also report **throughput**, defined as the number of jobs completed per unit time, to facilitate comparison across different prediction models.
- **Deadline Satisfaction Rate**: The percentage of s that complete before their specified deadlines. This metric is critical for evaluating the system's ability to meet job deadline and ensure QoS requirements.
- **Tail JCT (99th percentile)**: The job completion time below which 99% of jobs fall. This tail latency metric captures how well the system handles the slowest jobs.
- **Queue Length**: The number of jobs currently waiting for resource allocation. This metric offers an intuitive measure of the system's load and resource availability.

4.2 Overall Performance of Cuckoo

We evaluated the overall performance of Cuckoo against Muri [40]. Our assessment centered on key performance indicators, including Job Completion Time (JCT), deadline satisfaction rate, tail JCT, queue length, and the number of overtime jobs. As presented in Table 9, Cuckoo significantly outperforms Muri across crucial metrics, demonstrating a 1.81× improvement in JCT and a 2.38× increase in deadline satisfaction rate. While Cuckoo exhibits a slightly inferior tail JCT, this is a considered trade-off for achieving superior overall efficiency and robust deadline adherence. These performance gains stem from Cuckoo's advanced job packing and scheduling strategies, which effectively manage system load. As further illustrated in Figure 9, Cuckoo maintains significantly shorter queue lengths and

Table 9: Performance comparison between Cuckoo and Muri.

Method	Norm.JCT	Deadline Satisfaction Rate	Norm.Tail JCT
Cuckoo	**1.00**	**77.16%**	1.00
Muri	1.81	32.44%	**0.61**



(a) Queue length **(b) Number of overtime jobs**

Figure 9: Comparisons of queue length and overtime jobs.

fewer overtime jobs than Muri during high job submission periods, particularly in the intervals [1000, 10000] and [20000, 25000]. These results unequivocally demonstrate Cuckoo's superior ability to optimize resource utilization and job execution in heterogeneous environments.

4.3 Effectiveness of Efficiency Estimate

Prediction Accuracy Comparison. Compared to the naive prediction model used in Muri(denoted as Muri-P), our proposed interleaved job execution model Asynchronously-invoked Muri (AM) explicitly accounts for asynchronous resource invocation and performance interference arising from concurrent resource usage. To demonstrate that AM achieves higher prediction accuracy for interleaved job throughput, we evaluate the estimation error between predicted and actual throughput under identical conditions. Specifically, we construct approximately 1,600 job group samples by interleaving 23 distinct DL jobs in various execution orders across heterogeneous GPUs. The actual throughput of these job groups is measured on three GPU types: A10, A800, and RTX4090. We profile the runtime characteristics of individual jobs on different GPUs, and then use the two models to predict the throughput of all possible job groups. Figure 10 compares the throughput estimation errors of Muri-P and AM on heterogeneous GPU. On all devices, AM consistently achieves lower average estimation errors and exhibits fewer extreme outliers. The average estimation error is 13.5% for AM, compared with 20.2% for Muri-P, showing AM's superior modeling of asynchronous interference in interleaved execution.

Real Cluster Experiments. We implement both prediction algorithms and conduct experiments by submitting jobs to a real cluster, ensuring that the job grouping and scheduling algorithms are kept consistent for fair comparison. As shown in Table 10, compared to Muri-P, our AM-based prediction improves JCT by 1.07×, deadline satisfaction rate by 1.01×, and reduces tail JCT by 1.23×. In addition, we analyze the number of jobs predicted to complete on time (positive, P) and those predicted to miss their deadlines (negative, N) for both methods. We further compute the counts of true positives (TP), false positives (FP), true negatives (TN), and false negatives (FN), as summarized in Table 11. AM significantly reduces the false positive

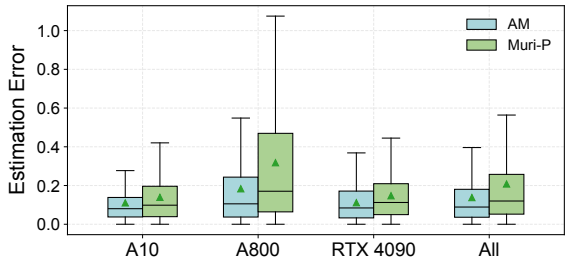


Figure 10: Throughput estimation error comparison.

Table 10: Performance comparison between AM and Muri-P

Method	Norm.JCT	Deadline Satisfaction Rate	Norm.Tail JCT
AM	**1.00**	**77.16%**	**1.00**
Muri-P	1.07	76.58%	1.23

Table 11: Predicted outcome breakdown.

Method	P	TP	FP	N	TN	FN
AM	90.21%	89.06%	1.15%	9.79%	6.82%	2.97%
Muri-P	91.17%	83.69%	7.48%	8.83%	5.10%	3.73%

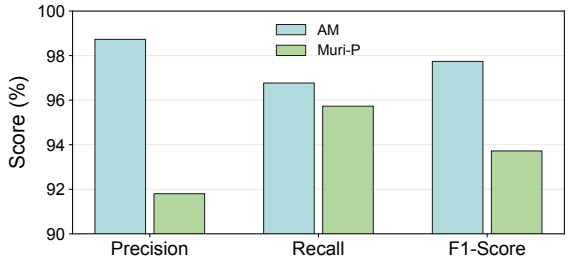
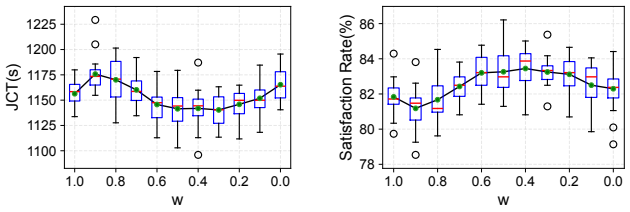


Figure 11: Precision, recall, and F1-score comparison.

rate, indicating more accurate and reliable positive predictions. As shown in Figure 11, although both methods exhibit similar recall, AM significantly outperforms Muri-P in precision and F1-score. Specifically, AM achieves a precision of 98.73% and a F1-score of 97.74%, compared to Muri-P's precision of 91.80% and F1-score of 93.72%. These results indicate that AM produces more reliable and accurate predictions by substantially reducing false positives.

4.4 Effectiveness of Job Packing

Matching Weight. As mentioned in Eq. 12, our job packing strategy introduces $total\_value$, a weighted combination of ddl_value (deadline adherence) and eff_value (packing efficiency), governed by parameter w. This w balances efficiency and deadline constraints during group matching. Varying w from 1 to 0 revealed a trade-off: lower w (prioritizing deadlines) can reduce resource utilization but expands the Available Deadline Window Size (ADWS), aiding feasible scheduling. As Figure 12 shows, JCT and deadline satisfaction rate exhibit non-monotonic trends, with optimal performance achieved when $w \in [0.5, 0.7]$. Consequently, we set $w = 0.6$ for all subsequent experiments.



(a) JCT **(b) Deadline Satisfaction Rate**

Figure 12: Impact of varying w on JCT and deadline satisfaction rate.

Table 12: Comparison of different job packing strategies.

Method	Norm.JCT	Deadline Satisfaction Rate	Norm.Tail JCT
DAB	**1.00**	**77.16%**	**1.00**
Blossom	1.32	47.02%	1.11
None	1.79	48.56%	1.77

Comparable Methods. We choose three representative job packing schemes for comparison.

- **DAB:** Deadline-Aware Blossom mentioned in §3.3. If there is no deadline requirement, DAB acts the same as Blossom.
- **Blossom:** Blossom algorithm used in Muri[40].
- **None:** Scheduling approach without job packig.

Result. Table 12 presents the performance comparison of different job packing strategies in terms of JCT, deadline satisfaction rate, and tail JCT. Among the three strategies, the None strategy performs the worst in all metrics. Due to the lack of job packing, jobs accumulate in the queue, resulting in longer job completion times and severe deadline violations.The Blossom strategy offers some improvement in JCT through basic job packing, but yields only limited gains in deadline satisfaction rate.

In contrast, our proposed DAB strategy outperforms both baselines by a significant margin. Compared to Blossom, DAB improves JCT by 1.32x, deadline satisfaction rate by 1.64x, and reduces tail JCT by 1.11x. These results demonstrate that DAB can more effectively balance interleaving efficiency and deadline constraints.

4.5 Effectiveness of Execution Planning

Comparable Methods. We also choose three representative execution planning schemes for comparison.

- **MCHP** denotes min cost heterogeneous placement discussed in §3.3.
- **FIFO** denotes the first in first out strategy. The job group submitted early will be executed first.
- **EDF** denotes the early deadline first. The job group which has an early deadline will execute first.

Result. Table 13 presents the performance comparison of different execution planning strategies in terms of JCT, deadline satisfaction rate, and tail JCT. Our proposed method, MCHP, achieves the best overall performance, improving JCT by 1.30× and deadline satisfaction rate by 1.86× compared to FIFO. Compared to EDF, it improves JCT by 1.22×, deadline satisfaction rate by 1.19×, and reduces tail JCT by 1.35×.

Table 13: Comparison of different execution planning strategies.

Method	Norm.JCT	Deadline Satisfaction Rate	Norm.Tail JCT
MCHP	**1.00**	**77.16%**	1.00
FIFO	1.30	41.46%	**0.79**
EDF	1.22	64.88%	1.35

Table 14: Comparison of heterogeneity-aware and homogeneous scheduling.

Method	Norm.JCT	Deadline Satisfaction Rate	Norm.Tail JCT
Cuckoo	**1.00**	**77.16%**	1.00
Cuckoo-Homo	1.16	71.29%	1.36
Muri-Hetero	1.60	38.20%	1.56
Muri	1.81	32.44%	**0.61**

FIFO preserves scheduling fairness by executing jobs in submission order, resulting in the shortest tail JCT. However, its poor resource efficiency leads to the highest overall JCT and the lowest deadline satisfaction among all methods. MCHP trades off a small degree of fairness to achieve high satisfaction rate and system efficiency, making it particularly effective under heterogeneous GPU settings with tight deadline constraints.

4.6 Effectiveness of Heterogeneity Awareness

To further evaluate the effectiveness of our heterogeneity-sensitive design, we introduce a variant, Cuckoo-Homo, which assumes all GPUs are NVIDIA RTX4090 during job packing and execution planning, thus disabling heterogeneity sensitivity. We also compare against two versions of Muri: the original Muri, which lacks heterogeneity-sensitive execution, and Muri-Hetero, an enhanced variant equipped with our heterogeneity-sensitive scheduling mechanism.

As shown in Table 14, Cuckoo significantly outperforms all baselines. Compared to Cuckoo-Homo, it improves normalized JCT by 1.16×, deadline satisfaction rate by 1.08×, and reduces tail JCT by 1.36×, demonstrating the benefits of heterogeneity sensitivity. Furthermore, even when Muri is enhanced with heterogeneity-sensitive scheduling (Muri-Hetero), its performance still lags far behind Cuckoo, indicating that heterogeneity sensitivity alone is not sufficient. Cuckoo 's joint design of job packing, scheduling, and resource modeling plays a crucial role in its overall effectiveness.

5 INDUSTRIAL CASE STUDY

5.1 Production-Grade Deployment

Cuckoo has been deployed in Kuaishou to serve thousands of model training jobs that can be interleaved on shared heterogeneous GPU clusters. Short video services and the pertaining recommendation models are among the most important workloads in Kuiashou's production systems. The traffic usually exhibits strong periodic fluctuation and noticeable peak and off-peak patterns in user behaviors (e.g. user clicks, app usage, transaction volume, etc.). For example, in the early morning, e.g., between 0am to 8am, GPU devices reserved for video recommendation model serving is extremely low-utilized, merely accounting for 20% of the entire clusters.

To tackle this, we co-located DL training jobs onto such inference clusters during the off-peak periods. It is thus desirable for such jobs to complete ahead of the huge amount of data samples and peak-time training jobs arrive. Specifying a concrete deadline turns out to be a constraint so that all the off-peak-time jobs can finish as soon as possible. We have deployed the core algorithms and multi-job interleaving mechanism presented in Cuckoo in our production-grade training services, and let multi-tenants from different business units set their own timing preference. The production monitoring system shows that the cluster-wide GPU utilization can be increased to 60% on average whilst the deadline satisfactory ratio can reach 90%.

5.2 Handling Jobs w/o Deadline Constraints

The deployed version of Cuckoo can also support the training jobs without specified deadline requirements, without additional modification or manual intervene. It can be simply done by setting the deadline to infinitely large and Cuckoo's scheduling algorithm will inherently prioritize to group jobs with similar deadlines and allocate available resources, before considering other low-priority or non-deadline jobs such as best-effort jobs.

5.3 Stability and Scalability

For stability, the native Kubernetes scheduling algorithms could be used as the fallback mechanism for launching non-deadline jobs or jobs that are unsuitable or uncertain for interleaving execution. Cuckoo's scalability can be guaranteed due to the inheritance of multiple components, such as API servers, from Kubernetes. In reality, the job packing and planning algorithm can make decisions of scheduling over 2,000 jobs every a few seconds. Considering the DL training jobs usually come into the system and are processed in a batch mode, such scheduling overhead is acceptable even in production-grade systems.

6 DISCUSSION

Applicable Models. The representative models in the experiment are listed in Table 9, including 8 diverse types such as A2C, DQN, BERT, VGG, ResNet, each of which has different batch sizes. Such models are used to better compare, in a fair manner, against Muri's algorithm that uses similar models when evaluating the effectiveness of handling interleaved multi-resources. Notably, Cuckoo supports more workloads in industrial production-grade environments, including recommendation models such as SIM, vision models such as ViT, and LLMs such as LLaMA.

Accuracy of Prediction Model and Its Impact. The accuracy of prediction model has direct impact on the deadline satisfaction rate. Statistically, jobs that miss their deadlines under asynchronized invocation typically exceed the deadline by less than 10%, whereas Muri jobs exceed it by around 30%. In the production cluster, to mitigate potential losses caused by prediction errors, we adopt two complementary measures. First, we continuously refine the prediction model with fresh profiling data collected from real executions, which reduces drift and improves accuracy over time. Second, we perform online monitoring of job progress and dynamically adjust scheduling plans whenever the observed runtime significantly deviates from the predicted values. These mechanisms ensure the robustness of Cuckoo even when prediction accuracy fluctuates.

Complexity and Overhead. The complexity and overhead of Cucкoo mainly stem from two aspects: *profiling* and *scheduling*. For profiling, after jobs are submitted, they are profiled across available GPUs. Since runtime information is cached, each job type only needs to be profiled once. The profiling overhead therefore grows linearly with the number of job types and heterogeneous GPU types. Regarding the scheduling overhead, our scheduling algorithm is based on the Blossom algorithm with a theoretical complexity of $O(n^3)$. In practice, however, the runtime is much smaller due to pruning and caching strategies. For example, with roughly 100 jobs in the queue in a testbed cluster, Cucкoo can complete scheduling within 1 s; with over 2,000 jobs in a production-grade cluster, the decision time is less than 3 s. Considering that typical DL training jobs run for hours or even days, such overhead is acceptable.

Fault Tolerance. Cucкoo incorporates several mechanisms to tolerate failures during job execution and scheduling. At the job level, each training task periodically checkpoints model states and intermediate metrics, allowing recovery from transient GPU or network failures with minimal progress loss. If a job fails or becomes unresponsive, the scheduler detects the failure and resubmits the job to another available GPU, leveraging cached profiling data to avoid repeated profiling overhead. At the group level, since job groups are scheduled independently, a failure within one group does not affect others, ensuring fault isolation and stable cluster throughput. When a GPU node crashes or temporarily disconnects, Cucкoo dynamically reclaims affected jobs and resubmits the workload across healthy devices. Overall, the fault tolerance design ensures that transient hardware or software failures have negligible impact on long-running DL training workloads.

7 RELATED WORK

DL cluster scheduler. A body of studies [11, 16–18, 31, 38, 41] leverage heuristic or learning-based scheduling to improve cluster utilization and reduce JCT. They use historical traces or early training signals to predict job runtime or resource demands. While effective in homogeneous and exclusively allocated GPU settings, they generally assume non-overlapping execution among jobs and neglect resource contention during concurrent GPU sharing. As a result, they cannot accurately model or optimize interleaved job execution scenarios, which are the focus of this study.

Deadline and heterogeneity-aware scheduling. A distinct line of research addresses deadline aware and heterogeneity aware scheduling. Works like Elasticflow [7] and Hypersched [15] focus on deadline-constrained resource allocation. Others, including TetriSched [30], 3sigma [23], Chronus [6], and Hydra [36], employ optimization techniques such as MILP or branch-and-bound to find deadline-guaranteed allocations. For heterogeneity, Gavel [22] and Sia [13] explore performance-driven scheduling on diverse GPUs, while Ada-SRSF [32], Topo [1], and MAPA [25] consider communication topology to mitigate contention and meet SLA requirements for multi-GPU jobs. However, these efforts often target big data jobs or treat DL training jobs as monolithic units, overlooking their inherent multi-stage characteristics. Cucкoo distinctively integrates deadline constraints with GPU heterogeneity, explicitly considering the multi-stage nature of DL jobs to fully leverage heterogeneous devices and enhance deadline adherence.

Multi-Job GPU Sharing. Early systems such as Gandiva [34], Antman [35], and Salus [39] explored time-sharing and suspend–resume mechanisms to enable fine-grained GPU sharing, improving utilization through rapid job switching and preemption. To further mitigate performance interference, several works [3, 9, 14, 38] developed predictive models—based on analytical methods, reinforcement learning, or random forests—to estimate interference and guide co-location decisions. Building on these efforts, Muri [40] introduced an interleaved execution model that captures stage-wise resource bottlenecks in DL training and performs fine-grained scheduling to improve utilization. Hardware-level isolation such as NVIDIA Multi-Instance GPU (MIG) [4, 26] has enabled interference-free GPU partitioning, and systems like MIG-Serving [29] have leveraged dynamic reconfiguration to optimize resource allocation for mixed workloads. Cucкoo utilizes stages to enable multi-resource sharing and improve performance by modeling the interleaved job packing.

8 CONCLUSION

Efficiently packing and scheduling DL jobs in multi-tenant clusters is increasingly challenging due to the significant heterogeneity of workloads and hardware accelerators. In this paper, we present Cucкoo, a deadline aware and heterogeneity sensitive scheduling system designed for DL workloads in multi-tenant GPU clusters. Cucкoo addresses the limitations of existing schedulers by accurately modeling the throughput of interleaved job groups, capturing the effects of asynchronous resource invocation and stage interference. It further incorporates job deadline requirements and GPU heterogeneity into a unified scheduling framework, jointly optimizes job grouping and resource allocation. Experiments show that Cucкoo improves deadline satisfaction by 2.38x and reduces average job completion time (JCT) by 1.81x compared with the state-of-the-art approaches. Cucкoo has been deployed at Kuaishou's large-scale production cluster systems, supporting various types of training workloads and effectively improving both job completion time and deadline satisfaction rate in real-world scenarios. In the future, we plan to investigate how to group latency-sensitive DL or LLM inference jobs to maximize the QoS satisfaction whilst improving the cost efficiency over heterogeneous clusters. We also plan to co-locate training and inference jobs in a shared GPU cluster.

ACKNOWLEDGMENT

We would very much like to thank our Shepherd Dr. Abel Souza and the anonymous reviewers for their insightful and constructive comments. Special thanks also must go to the members in AI platform team at Kuaishou Inc. and RAIDS Lab at Beihang University, for their collaborative contribution and countless technical discussion.

This work is supported in part by National Key R&D Program of China (Grant No. 2024YFB4505604), in part by the National Natural Science Foundation of China (Grant No. 62402024), in part by the Fundamental Research Funds for the Central Universities, and, last but not the least, by Kuaishou Research Fund. For any correspondence, please refer to the project lead and coordinator Prof. Renyu Yang (renyuyang@buaa.edu.cn).

REFERENCES

[1] Marcelo Amaral, Jordà Polo, David Carrera, Seetharami Seelam, and Malgorzata Steinder. 2017. Topology-aware gpu scheduling for learning workloads in cloud environments. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–12.

[2] Yixin Bao, Yanghua Peng, and Chuan Wu. 2019. Deep learning-based job placement in distributed machine learning clusters. In *IEEE INFOCOM 2019-IEEE conference on computer communications*. IEEE, 505–513.

[3] Yixin Bao, Yanghua Peng, and Chuan Wu. 2022. Deep learning-based job placement in distributed machine learning clusters with heterogeneous workloads. *IEEE/ACM Transactions on Networking* 31, 2 (2022), 634–647.

[4] Jack Choquette, Wishwesh Gandhi, Olivier Giroux, Nick Stam, and Ronny Krashinsky. 2021. Nvidia a100 tensor core gpu: Performance and innovation. *IEEE Micro* 41, 2 (2021), 29–35.

[5] 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)*. 4171–4186.

[6] Wei Gao, Zhisheng Ye, Peng Sun, Yonggang Wen, and Tianwei Zhang. 2021. Chronus: A novel deadline-aware scheduler for deep learning training jobs. In *Proceedings of the ACM Symposium on Cloud Computing*. 609–623.

[7] Diandian Gu, Yihao Zhao, Yinmin Zhong, Yifan Xiong, Zhenhua Han, Peng Cheng, Fan Yang, Gang Huang, Xin Jin, and Xuanzhe Liu. 2023. ElasticFlow: An elastic serverless training platform for distributed deep learning. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*. 266–280.

[8] 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.

[9] Rong Gu, Yuquan Chen, Shuai Liu, Haipeng Dai, Guihai Chen, Kai Zhang, Yang Che, and Yihua Huang. 2021. Liquid: Intelligent resource estimation and network-efficient scheduling for deep learning jobs on distributed GPU clusters. *IEEE Transactions on Parallel and Distributed Systems* 33, 11 (2021), 2808–2820.

[10] 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.

[11] Qinghao Hu, Peng Sun, Shengen Yan, Yonggang Wen, and Tianwei Zhang. 2021. Characterization and prediction of deep learning workloads in large-scale gpu datacenters. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–15.

[12] Qinghao Hu, Peng Sun, Shengen Yan, Yonggang Wen, and Tianwei Zhang. 2021. Characterization and prediction of deep learning workloads in large-scale GPU datacenters. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (<conf-loc>, <city>St. Louis</city>, <state>Missouri</state>, </conf-loc>) *(SC '21)*. Association for Computing Machinery, New York, NY, USA, Article 104, 15 pages. https://doi.org/10.1145/3458817.3476223

[13] 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*. 642–657.

[14] Sejin Kim and Yoonhee Kim. 2020. Co-scheML: Interference-aware container co-scheduling scheme using machine learning application profiles for GPU clusters. In *2020 IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE, 104–108.

[15] Richard Liaw, Romil Bhardwaj, Lisa Dunlap, Yitian Zou, Joseph E Gonzalez, Ion Stoica, and Alexey Tumanov. 2019. Hypersched: Dynamic resource reallocation for model development on a deadline. In *Proceedings of the ACM Symposium on Cloud Computing*. 61–73.

[16] Liu Liu, Jian Yu, and Zhijun Ding. 2022. Adaptive and efficient gpu time sharing for hyperparameter tuning in cloud. In *Proceedings of the 51st International Conference on Parallel Processing*. 1–11.

[17] Ziyang Liu, Renyu Yang, Jin Ouyang, Weihan Jiang, Tianyu Ye, Menghao Zhang, Sui Huang, Jiaming Huang, Chengru Song, Di Zhang, et al. 2024. Kale: Elastic GPU Scheduling for Online DL Model Training. In *Proceedings of the 2024 ACM Symposium on Cloud Computing*. 36–51.

[18] Yunteng Luan, Xukun Chen, Hanyu Zhao, Zhi Yang, and Yafei Dai. 2019. SCHED^2: Scheduling Deep Learning Training via Deep Reinforcement Learning. In *2019 IEEE Global Communications Conference (GLOBECOM)*. IEEE, 1–7.

[19] 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.

[20] Volodymyr Mnih, Adria Puigdomenech Badia, Mehdi Mirza, Alex Graves, Timothy Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. 2016. Asynchronous methods for deep reinforcement learning. In *International conference on machine learning*. PmLR, 1928–1937.

[21] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. 2013. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602* (2013).

[22] 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.

[23] Jun Woo Park, Alexey Tumanov, Angela Jiang, Michael A Kozuch, and Gregory R Ganger. 2018. 3sigma: distribution-based cluster scheduling for runtime uncertainty. In *Proceedings of the Thirteenth EuroSys Conference*. 1–17.

[24] 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.

[25] Kiran Ranganath, Joshua D Suetterlein, Joseph B Manzano, Shuaiwen Leon Song, and Daniel Wong. 2021. Mapa: Multi-accelerator pattern allocation policy for multi-tenant gpu servers. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–14.

[26] Ties Robroek, Ehsan Yousefzadeh-Asl-Miandoab, and Pınar Tözün. 2024. An Analysis of Collocation on GPUs for Deep Learning Training. In *Proceedings of the 4th Workshop on Machine Learning and Systems*. 81–90.

[27] Karen Simonyan and Andrew Zisserman. 2014. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556* (2014).

[28] Abeda Sultana, Li Chen, Fei Xu, and Xu Yuan. 2020. E-LAS: Design and analysis of completion-time agnostic scheduling for distributed deep learning cluster. In *Proceedings of the 49th International Conference on Parallel Processing*. 1–11.

[29] Cheng Tan, Zhichao Li, Jian Zhang, Yu Cao, Sikai Qi, Zherui Liu, Yibo Zhu, and Chuanxiong Guo. 2021. Serving DNN models with multi-instance gpus: A case of the reconfigurable machine scheduling problem. *arXiv preprint arXiv:2109.11067* (2021).

[30] Alexey Tumanov, Timothy Zhu, Jun Woo Park, Michael A Kozuch, Mor Harchol-Balter, and Gregory R Ganger. 2016. TetriSched: global rescheduling with adaptive plan-ahead in dynamic heterogeneous clusters. In *Proceedings of the Eleventh European Conference on Computer Systems*. 1–16.

[31] Haoyu Wang, Zetian Liu, and Haiying Shen. 2020. Job scheduling for large-scale machine learning clusters. In *Proceedings of the 16th International Conference on emerging Networking EXperiments and Technologies*. 108–120.

[32] Qiang Wang, Shaohuai Shi, Canhui Wang, and Xiaowen Chu. 2020. Communication contention aware scheduling of multiple deep learning training jobs. *arXiv preprint arXiv:2002.10105* (2020).

[33] Qizhen Weng, Wencong Xiao, Yinghao Yu, Wei Wang, Cheng Wang, Jian He, Yong Li, Liping Zhang, Wei Lin, and Yu Ding. 2022. {MLaaS} in the wild: Workload analysis and scheduling in {Large-Scale} heterogeneous {GPU} clusters. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*. 945–960.

[34] 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.

[35] 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.

[36] Zichao Yang, Heng Wu, Yuanjia Xu, Yuewen Wu, Hua Zhong, and Wenbo Zhang. 2023. Hydra: Deadline-aware and efficiency-oriented scheduling for deep learning jobs on heterogeneous gpus. *IEEE Trans. Comput.* (2023).

[37] Zhisheng Ye, Wei Gao, Qinghao Hu, Peng Sun, Xiaolin Wang, Yingwei Luo, Tianwei Zhang, and Yonggang Wen. 2024. Deep learning workload scheduling in gpu datacenters: A survey. *Comput. Surveys* 56, 6 (2024), 1–38.

[38] Gingfung Yeung, Damian Borowiec, Renyu Yang, Adrian Friday, Richard Harper, and Peter Garraghan. 2021. Horus: Interference-aware and prediction-based scheduling in deep learning systems. *IEEE Transactions on Parallel and Distributed Systems* 33, 1 (2021), 88–100.

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

[40] Yihao Zhao, Yuanqiang Liu, Yanghua Peng, Yibo Zhu, Xuanzhe Liu, and Xin Jin. 2022. Multi-resource interleaving for deep learning training. In *Proceedings of the ACM SIGCOMM 2022 Conference*. 428–440.

[41] Pan Zhou, Xinshu He, Shouxi Luo, Hongfang Yu, and Gang Sun. 2020. JPAS: Job-progress-aware flow scheduling for deep learning clusters. *Journal of Network and Computer Applications* 158 (2020), 102590.