Beware of Fragmentation: Scheduling GPU-Sharing Workloads with Fragmentation Gradient Descent

Qizhen Weng[†*] Lingyun Yang[†*] Yinghao Yu[§†] Wei Wang[†]
Xiaochuan Tang[§] Guodong Yang[§] Liping Zhang[§]

[†]Hong Kong University of Science and Technology [§]Alibaba Group
{qwengaa, lyangbk, weiwa}@cse.ust.hk, {yinghao.yyh, xiaochuan.txc, liping.z}@alibaba-inc.com, luren.ygd@taobao.com

Abstract

Large tech companies are piling up a massive number of GPUs in their server fleets to run diverse machine learning (ML) workloads. However, these expensive devices often suffer from significant underutilization. To tackle this issue, GPU sharing techniques have been developed to enable multiple ML tasks to run on a single GPU. Nevertheless, our analysis of Alibaba production traces reveals that allocating partial GPUs can result in severe *GPU fragmentation* in large clusters, leaving hundreds of GPUs unable to be allocated. Existing resource packing algorithms fall short in addressing this problem, as GPU sharing mandates a new scheduling formulation beyond the classic bin packing.

In this paper, we propose a novel measure of fragmentation to statistically quantify the extent of GPU fragmentation caused by different sources. Building upon this measure, we propose to schedule GPU-sharing workloads towards the direction of *the steepest descent of fragmentation*, an approach we call *Fragmentation Gradient Descent* (FGD). Intuitively, FGD packs tasks to minimize the growth of GPU fragmentation, thereby achieving the maximum GPU allocation rate. We have implemented FGD as a new scheduler in Kubernetes and evaluated its performance using production traces on an emulated cluster comprising more than 6,200 GPUs. Compared to the existing packing-based schedulers, FGD reduces unallocated GPUs by up to 49%, resulting in the utilization of additional 290 GPUs.

1 Introduction

Graphics Processing Units (GPUs) are widely deployed in production clusters to accelerate machine learning (ML) tasks for a plethora of AI applications [14, 16, 17, 21, 31, 39]. Compared to CPUs and other resources, GPUs are considerably more expensive but often underutilized in production clusters, with the reported utilization rates ranging from 25% to below 50% [16, 19, 22, 31].

The primary reason for low GPU utilization is that a large number of ML tasks, mostly inference, cannot fully utilize the capacities of modern GPUs, which have seen exponential performance improvements in recent years. This trend is expected to continue in the foreseeable future [36]. To address this issue, GPU sharing techniques have been developed to enable multiple ML tasks to safely run on a single GPU with guaranteed isolation, where each task is allocated *partial resources* by means of virtualization [9, 13, 27, 33, 35] or the Multi-Instance-GPU (MIG) feature supported in NVIDIA's Ampere architecture [3].

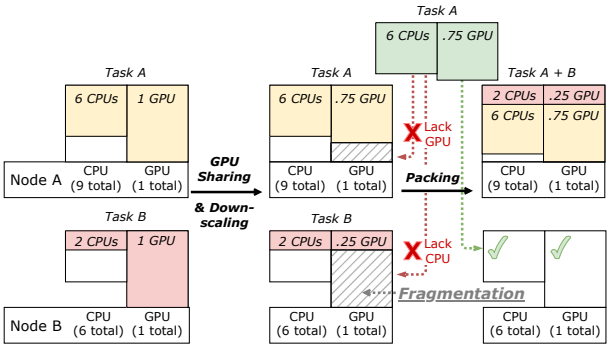


Figure 1: The allocation of partial GPUs results in fragmentation, which can be addressed with packing.

However, simply enabling GPU sharing does not necessarily lead to high utilization. In many cases, allocating partial GPUs results in *fragmentation*, preventing the remaining GPU resources from being allocated. Figure 1 illustrates this problem in a toy example. Consider a cluster of two nodes A and B with ⟨9 CPUs, 1 GPU⟩ and ⟨6 CPUs, 1 GPU⟩, respectively. There are two tasks A and B running on the two nodes, each demanding ⟨6 CPUs, 0.75 GPU⟩ and ⟨2 CPUs, 0.25 GPU⟩, respectively. Without GPU sharing, both tasks are allocated an entire GPU even though they cannot fully utilize it (Figure 1, left). This problem can be addressed by allocating partial GPUs, using the GPU sharing technique (Figure 1, middle). Now supposing another instance of task A arrives, it cannot run on either node even though the cluster has sufficient aggregate GPU resources ($0.25 + 0.75 = 1$ GPU).

GPU fragmentation has been widely observed in our production clusters that support GPU sharing. Figure 2 shows a 7-day trace collected from a 1280-GPU cluster. On average, the aggregate GPU allocations account for 77.6% of the total

*Equal contributions.

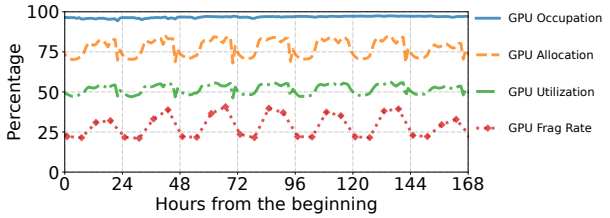


Figure 2: A 7-day trace from a large GPU-sharing cluster. GPU occupation measures the number of GPU devices that are not fully idle; GPU allocation measures the total amount of allocated GPU resources; GPU utilization refers to the proportion of actual GPU resources used by tasks; GPU frag rate is the percentage of unallocated GPU resources that become unusable due to fragmentation (defined in §3.2).

capacity (orange dashed line). These allocations, many being partial GPUs, are distributed across almost all GPU devices (blue solid line), turning 21–42% of the unallocated GPU resources into fragments (red dotted line) that cannot be utilized by the current workload. In general, higher allocations result in more severe fragmentation. In our operational experience, complaints about scheduling failures usually surge in rush hours, although the cluster still has sufficient aggregate idle GPUs, indicating severe fragmentation.

An effective approach to addressing fragmentation is to perform *packing*. Returning to the previous example, the scheduler can instead pack the two original tasks to node A, thereby leaving the entire node B to the new instance of task A (Figure 1, right). A large body of work formulates workload scheduling as a multi-dimensional bin packing problem, in which tasks and nodes are respectively modeled as balls and bins with sizes in multiple resource dimensions, and the goal is to pack balls to the fewest number of bins. Many heuristics have been proposed to schedule cluster workloads, such as best-fit [11, 21, 24, 30], vector alignment scoring [8, 23, 24], and "GPU Packing" [31, 33].

However, our experiments show that none of these heuristics work well in scheduling GPU-sharing workloads (§6). The fundamental reason, we believe, is that the problem is *intrinsically different* from the classic bin packing when a node has multiple GPUs. Consider two natural bin packing formulations. The first is to model a server's multiple GPUs as one large device with the aggregate capacity. This formulation pools together all unallocated GPUs, and the fragments on individual GPUs become irrelevant, which is not the case in reality. Alternatively, one can treat a server's multiple GPUs as different "resource dimensions". Yet, unlike other resources such as CPU and memory, these GPUs are not independent but *interchangeable* for a task that can run on any of them, mandating a new formulation beyond classic bin packing.

In this paper, we develop a novel fragmentation-aware scheduling approach for GPU-sharing workloads. Central to our approach is a new analytical framework that quantifies statistically the degree of GPU fragmentation in a cluster. Given a task, we identify the GPUs on each node that cannot be used to run the task (e.g., lacking sufficient GPU or other resources). These GPUs are *fragmented* from the view of that task as none of their remaining resources can be utilized. Now, consider the *target workload*, which consists of a set of tasks that are of interest (e.g., ML inference and training). We quantify the degree of GPU fragmentation as *the expected number of GPUs that cannot be allocated to a task which is randomly sampled from the target workload*. Intuitively, it measures the expected GPU resources that cannot be utilized by the target workload. We can further break down the fragmentation analysis into different causes, such as the node having insufficient or stranded GPUs, or the mismatch between the workload and the node spec. This analysis provides more insights for the operator to reason about the cluster state (§3).

Based on the GPU fragmentation analysis, we propose a simple, yet effective heuristic to schedule workloads towards the direction of *the steepest descent of fragmentation*, which we call Fragmentation Gradient Descent or FGD. For each GPU task submitted, FGD chooses a node and the available GPU(s) on it to run the task so that the growth of GPU fragmentation caused by this decision is minimized (§4). By doing so, FGD can minimize GPU fragmentation, saving a large amount of expensive resources for more workloads.

We have implemented FGD as a new scheduler in Kubernetes [1] (§5) and evaluated its performance with production and synthesized workload traces on an emulated cluster consisting of more than 1,200 nodes and 6,200 GPUs (§6). FGD consistently outperforms existing packing-based scheduling algorithms in various settings: it reduces the unallocated GPUs by up to 49%, allowing 290 GPUs to be utilized in a large production cluster. Our implementation, including the scheduler and the emulator[1], as well as the trace data[2] used in the evaluation, are available as open-source software.

2 Background and Motivation

In this section, we briefly introduce the GPU sharing technique and illustrate the resulted fragmentation problem through production trace analysis. We discuss the unique scheduling challenge brought by GPU sharing that invalidates the classic bin packing formulation.

2.1 GPU Sharing

Underutilized GPUs. GPU underutilization has been widely observed in production clusters that run diverse ML workloads. Many tech companies report the low GPU utilization averaging between 25% to below 50% [16, 19, 22, 31], which has become a thorny pain point in reducing the total cost of ownership of large GPU clusters.

[1] https://github.com/hkust-adsl/kubernetes-scheduler-simulator
[2] https://github.com/alibaba/clusterdata

There are multiple factors that contribute to the low GPU utilization. Most importantly, thanks to the exponential improvement of GPU performance in recent years, many ML tasks, especially inference, cannot saturate the compute capacity of a modern GPU. Taking the latest A100 GPU as an example, the peak inference speed of a ResNet50 [15] model reaches over 36k images per second [36], far exceeding the usual throughput requirement of an object detection application. In fact, even for training tasks, increasing evidences show that many of them cannot fully utilize a GPU [5, 6, 28, 31, 33, 34, 37].

GPU Sharing. GPU sharing techniques have recently been developed to enable multiple tasks to run on a single GPU with guaranteed performance isolation, where each task is allocated a partial GPU. In production systems, GPU sharing can be implemented at three levels.

1) Framework-level: This approach adds new dynamic scaling mechanisms and sharing primitives to the existing ML frameworks (e.g., TensorFlow, PyTorch, JAX) to allocate each task the exact amount of required GPU memory and compute units [6, 33, 34, 37]. The benefit of this approach is that it can achieve fine-grained sharing between tasks by leveraging their semantics information (e.g., training accuracy and loss), which is available to the framework. On the other hand, it requires users to switch to the modified framework to enable GPU sharing – not all users are willing to do so.

2) Device runtime-level: This approach uses the API remoting technique to implement GPU sharing and virtualization [9, 13, 27, 28, 32]. It deploys a GPU manager on each host. The manager intercepts compute- and memory-related runtime APIs (e.g., `cuLaunchKernel` and `cuMemAlloc` in CUDA Library) to keep track of the compute and memory resources requested by each task. A task's memory allocation request is accepted only when the its allocation is within the specified limit. The manager also controls kernel scheduling to enforce the specified allocation of compute capacity by time-multiplexing the device's compute units between tasks (e.g., a task with 0.1 GPUs is guaranteed to receive at least 10% of GPU time). This approach requires no framework changes or user cooperation.

3) Hardware-assisted: Starting with Ampere architecture, NVIDIA GPUs support the Multi-Instance GPU (MIG) feature. MIG can partition a GPU into as many as seven separate instances [3]. Compared with the software approaches, MIG provides the strongest isolation guarantee as each GPU instance has dedicated resources for compute, memory, and memory bandwidth. On the downside, MIG only supports resource sharing at a coarse granularity and is available exclusively to A100, A30, and H100 GPUs at the moment.

Production Deployment in Our Clusters. At Alibaba, we have developed our own GPU sharing solution based on CUDA Runtime API interception and deployed it in production clusters that run a mixture of training and inference tasks. A task can have one or multiple instances, each running in a container and requesting multiple resources such as CPUs, memory, and GPUs. We observe the GPU requests to be either a partial GPU or full GPU(s), but rarely the combination of both (e.g., 1.5 or 2.3 GPUs). In our implementation, the minimum GPU allocation unit is 0.01 GPUs, in which a task instance is allocated 1% of the GPU memory and *at least* 1% of the GPU time.[3] Tasks and their instances are orchestrated using a customized Kubernetes system [1] with many new features tailored to the production needs. In the following discussions, unless otherwise specified, *we do not differentiate between tasks and instances* as their meanings are usually clear from the context.

2.2 The Prevalence of GPU Fragmentation

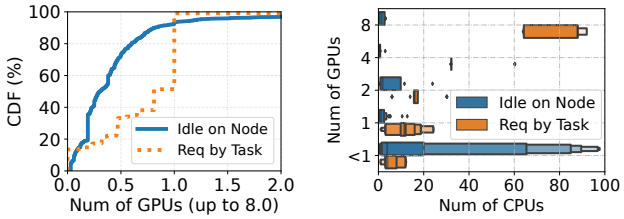
Operational Experience. GPU sharing greatly reduces the number of allocated GPUs for a workload, allowing more tasks to run in a cluster than before. However, as we consolidate more workloads, an increasing number of users complained about the long task wait time or even scheduling failures due to the timeout of pending tasks, although they still have sufficient GPU quotas to spare. In many clusters, the GPU allocation rate can reach 85–90% maximum, leaving hundreds of GPUs unable to utilize. In some extreme cases, pending tasks start to build up when the GPU allocation rate reaches above 80%. All these indicate heavy fragmentation.

Trace Analysis. We perform trace analysis to confirm the fragmentation problem, which occurs when a GPU has *insufficient resources* or becomes *stranded* as the host runs out of the other resources, such as CPU and memory. We choose a highly crowded ML cluster H consisting of 1.2k nodes with over 6.2k GPUs and 107k CPU cores. After scheduling over 7.6k tasks, the GPU (CPU) allocation ratio reaches 92% (75%), which is among the highest in all clusters. At this point, cluster H is fully packed and cannot accommodate new tasks despite having a total of 500 GPUs unallocated, causing a significant resource waste. Figure 3a depicts the distributions of unallocated GPUs on each node and the requested GPUs of each pending task. Around 92% of nodes have < 1 GPU left unallocated (blue solid line), whilst 49% of the pending tasks request ≥ 1 GPU (orange dashed line). Figure 3b gives the boxen plot of the nodes' unallocated CPUs and the tasks' requested CPUs, grouped by their GPU resources. Among the nodes with ≥ 1 unallocated GPU, over 75% have ≤ 10 CPUs left (blue boxes), which are *insufficient* to allocate to most tasks (orange boxes), leaving the unallocated GPUs stranded.

2.3 Inapplicable Bin Packing Formulation

While fragmentation is not a novel problem in cluster management, GPU fragmentation is noteworthy because it (1)

[3] In our system, a task instance can opportunistically use more GPU time if the computing capacity of the said GPU is not exhausted by the other tasks.



(a) CDF of idle and requested GPUs. (b) Boxen of idle and requested CPUs.

Figure 3: Illustration of GPU fragmentation in a fully packed cluster H. (a) Most nodes have insufficient GPUs. (b) Nodes with abundant GPUs are usually in short of CPUs.

inherently differs from other resource fragmentation problem and (2) is aggravated by partial-GPU allocation (Figure 1).

GPU Fragmentation Cannot be Handled Similarly as Other Resources. Fragmentation is a common problem in resource allocation. Taking file allocation as an example, when the disk has no enough contiguous space to store a file, the fragmentation occurs [4], and the solution is to chunk the file into blocks for non-contiguous allocation. However, GPU allocation must be *contiguous*: Consider a task requesting one full GPU, it is not possible to allocate it two partial GPUs (e.g., 0.3 GPUs + 0.7 GPUs).

Partial-GPU Allocation Invalidates Bin Packing Formulation. Bin packing is known effective to address the fragmentation problem [23, 24]. In the standard formulation, tasks and nodes are respectively modeled as balls and bins of sizes in $\mathbb{R}^d$, where d is the number of concerned resources, such as CPU, memory, and GPU. The goal is to pack balls to as fewest bins as possible. However, unless a node has a single GPU, this formulation does not apply to GPU-sharing tasks, which we illustrate through two formulation attempts.

Attempt-1: Treating Multiple GPUs as a Unified Logical Device. This formulation pools together all the available GPU resources of a node into a large logical GPU, leading to a node resource vector such as $\langle 16\text{ CPUs}, 24\text{ GiB memory}, 1.3\text{ GPUs}\rangle$. However, this formulation is problematic as it ignores the *allocation boundary* of physical GPUs, making it unable to differentiate the fragmentation state on each individual device. For example, consider a 2-GPU node with the remaining capacity of 0.4 GPUs and 0.9 GPUs. For a task that requests one full GPU, although in total the node has 1.3 GPUs, neither of its two GPUs has sufficient capacity to run the task, to which both GPUs are fragmented.

Attempt-2: Treating Each GPU as an Independent Resource Dimension. An alternative formulation is to model each GPU on a node as an independent resource dimension, just like CPU and memory. This formulation is also problematic as GPUs are *interchangeable* as long as they have sufficient capacity to run a task. Returning to the previous example and assuming that the node has 16 CPU cores and 24 GiB memory available, its resource vector is $\langle 16\text{ CPUs}, 24\text{ GiB memory}, 0.4\text{ GPUs}, 0.9\text{ GPUs}\rangle$. For a task that requests 2 cores, 8 GiB memory, and 0.3 GPUs, it can run on the node with any of the two GPUs. The task hence has two interchangeable demand vectors $\langle 2\text{ CPUs}, 8\text{ GiB memory}, 0.3\text{ GPUs}, 0\text{ GPUs}\rangle$ or $\langle 2\text{ CPUs}, 8\text{ GiB memory}, 0\text{ GPUs}, 0.3\text{ GPUs}\rangle$. This invalidates the classic bin packing formulation as the balls are now "deformable" and can transform to various sizes.

To summarize, we stress that the inapplicability of bin packing formulation stems from the contiguous GPU allocation requirement and the partial-GPU allocation practice, in which each GPU has its own allocation boundary (Attempt 1) but is also interchangeable to one another (Attempt 2). Through extensive experiments in §6, we will show that none of the existing packing heuristics, including best-fit [11, 21, 24, 30], vector alignment scoring [8, 23, 24], and "GPU Packing" [31, 33], work well in scheduling GPU sharing workload.

3 The Fragmentation Measure

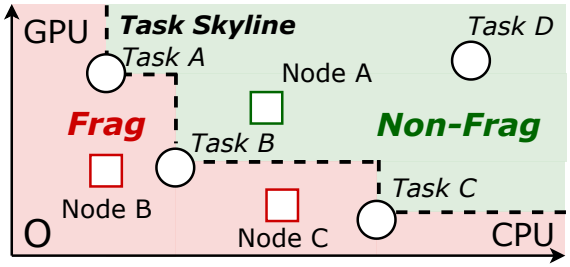
"You keep using that word. I do not think it means what you think it means." — Inigo Montoya, *The Princess Bride*

While the term fragmentation has been frequently mentioned in the existing cluster scheduling works [8, 11, 29, 30, 33, 34, 39, 40], its formal definition and quantitative measure remain unclear. In this section, we answer *what fragmentation is and how can it be measured*. We start with a conventional measure defined in absolute terms and discuss its ineffectiveness (§3.1). We next present a new measure that statistically quantifies the fragmentation degree in a cluster (§3.2). We show in case studies that the new measure can help operators better reason about the cluster state (§3.3).

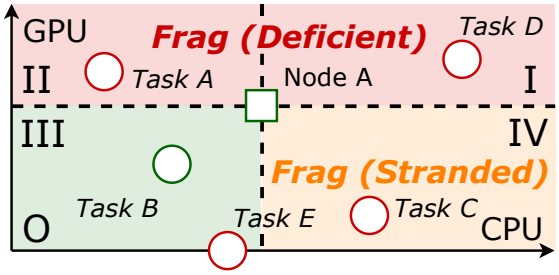
3.1 Fragmentation in Absolute Terms

Basic Assumption. In general, *whether a node has fragmented resources depends on the target workload*. For example, a node with 0.6 GPUs is considered fragmented by a task that requests one full GPU, but not by the one that demands 0.5 GPUs. In this paper, we assume that the target workload, which contains a set of tasks with popularity (e.g., number of instances) following a certain distribution, is known to the cluster. This is reasonable as production ML workloads consist of a large number of recurring tasks [31]. We also assume that each task can request either a partial GPU or full GPU(s), but not both, as mentioned in §2.1.

An Absolute Fragmentation Measure. Although not formally defined, it is commonly accepted that a node is fragmented *if its remaining resources cannot be allocated to run any tasks* in the target workload. In this definition, fragmentation is measured in absolute terms: regardless of task scheduling, the fragmented resources cannot be utilized anyway and are inevitably wasted.



(a) Absolute fragmentation defined by task skyline is defective.



(b) Statistical fragmentation in expectation of randomly sampled tasks.

Figure 4: Fragmentation definition in the example cluster with nodes of various unallocated resources (squares) and tasks of various requested resources (circles).

Figure 4a gives a pictorial illustration. For simplicity, we only consider CPU and GPU resources in a two-dimensional plane. A task is depicted as a circle (ball) with x- and y-coordinates being the requested CPUs and GPUs, respectively. Similarly, a node is depicted as a box (bin) with the two coordinates being the unallocated CPUs and GPUs. In case that a node has multiple GPUs, we map their unallocated capacity (a vector) to a scalar number as follows. Let f be the number of *fully-unallocated GPUs* and p the *maximum unallocated partial GPU*. We map the vector of unallocated GPUs to a scalar $u = f + p$.[4] For example, a 4-GPU node with an unallocated capacity of $\langle 1, 1, 0.5, 0.25 \rangle$ is considered having $u = 2.5$ unallocated GPUs. Under this mapping, a node has sufficient GPUs to run a task that requests g GPUs *if and only if* $u \geq g$, provided that the task requests either a partial GPU or full GPU(s), i.e., $g \in [0,1) \cup \mathbb{Z}^+$.

With nodes and tasks depicted in the resource plane, we see that a node has sufficient GPUs and CPUs to run a task if it is located above and on the right of the task (e.g., node A and task B). We call this region the *non-fragmentation region* of the task. Taking the union of the non-fragmentation regions of all tasks gives the non-fragmentation region of the target workload (the green area in Figure 4a). It encompasses all tasks, with the "innermost" ones (e.g., tasks A, B, and C) located on the region's border which we call the *skyline* [38]. The area below the skyline is the *fragmentation region*, within

[4]The mapping is not unique. For example, alternatively one can map the capacity vector to $u = \max\{f, p\}$, which serves the same purpose.

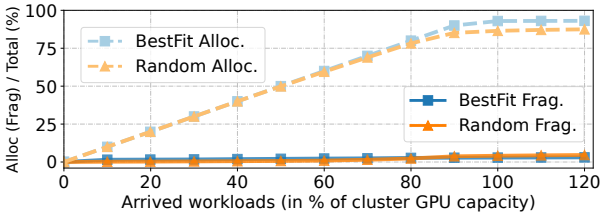


Figure 5: Trace-driven emulations with two scheduling policies (details in §6.1). The x-axis is the cumulative GPU demands of arrived tasks, divided by the total cluster capacity. Fragmentation in absolute terms (lower-right corner) stays at a low level ($< 5\%$) throughout the allocation of arrival workloads, failing to provide useful feedback to the scheduler.

which a node cannot run any task due to insufficient resources and is thus fragmented (e.g., nodes B and C).

The Inefficiency of the Absolute Measure. Under the absolute measure, resource fragmentation is identified in a rather biased manner. Whether a node is fragmented solely depends on if it has sufficient resources to run a task that is located on the skyline (i.e., the skyline task), whereas the other tasks are irrelevant. Yet, skyline tasks can rarely represent the entire workload. Compared with the other tasks, they request fewer CPUs and/or GPUs, and usually have a small population. In our clusters, only 0.06% of instances belong to the skyline tasks. On average, a skyline task requests $\langle$3.2 CPUs, 0.07 GPUs$\rangle$, as opposed to the average demand of $\langle$9.4 CPUs, 0.9 GPUs$\rangle$. As a result, even if a node has a small amount of resources that cannot be allocated to run the majority of the tasks in the target workload, it may still be considered non-fragmented as long as it can run a tiny skyline task.

For the reasons above, the absolute fragmentation measure cannot be used as a good metric to guide task scheduling. To see this, we run trace-driven emulations with two different scheduling policies (details in §6.1) and depict in Figure 5 the fragmented and allocated GPUs in percentage of the total capacity. The fragmentation measure stays at a low level ($< 5\%$) throughout the process regardless of the scheduling logic and its placement decisions, failing to provide useful feedback to the scheduler. Increased fragmentation is only observed when the cluster starts crowded, by which it is too late for the scheduler to take actions. In fact, even when the cluster becomes fully packed (i.e., the allocation curve flattens when the cumulative GPU demands reach over 100% of the cluster capacity), over 50% of unallocated GPUs are still deemed non-fragmented under the absolute measure.

3.2 A Statistical Fragmentation Measure

We believe a good fragmentation measure should not be defined against a small subset of tasks, but a joint calibration of the entire workload. We hence use a *statistical measure* to quantify the degree of fragmentation. Formally, let M be the target workload in which each task m has popularity p_m.

Without loss of generality, we assume normalized popularity where $\sum_{m \in M} p_m = 1$. Given a node n, $F_n(M)$ denotes the GPU fragmentation measured by workload M, and $F_n(m)$ is the fragmentation measured by a certain task m in the workload. We define the node-level measure as the weighted sum of the task-level, i.e.,

$$F_n(M) = \sum_{m \in M} p_m F_n(m). \tag{1}$$

One can interpret Eqn. (1) as *the expected fragmentation* measured by a task that is *randomly sampled* from the target workload. We next describe how $F_n(m)$ can be computed.

Pictorial Interpretation. From the view of a task, it considers a GPU of a node being fragmented if it cannot be allocated to the task. As a pictorial interpretation, we refer to Figure 4b and consider node A. In case that the node has multiple GPUs, we map their unallocated capacity (a vector) to a scalar representation following the approach described in §3.1. We partition the resource plane into four quadrants with node A at the origin. The node has insufficient resources to run any of the tasks that are located in Quadrants I, II, and IV, among which those in Q-I are in short of CPUs and GPUs (e.g., task D), Q-II in short of GPUs (e.g., task A), and Q-IV in short of CPUs (e.g., task C). From the point of these tasks, the unallocated GPUs are all fragmented, as they either have insufficient capacity (Q-I and Q-II) or become *stranded* due to the lack of CPU resource on the node (Q-IV).

Things become a bit more complex when it comes to Q-III. While the node has sufficient resources to run a task in that quadrant, not every unallocated GPU has enough capacity. For example, on a 4-GPU node with an unallocated capacity of $\langle 1, 1, 0.5, 0.25 \rangle$, the two partial GPUs cannot be assigned to a task that requests 2 GPU, even though the node has sufficient GPU resources. In this case, the two partial GPUs should be counted as being fragmented for the task. More generally, given a task in Q-III, we check each unallocated GPU, and those with insufficient capacity are considered fragmented.

In addition to the four quadrants, tasks can also locate on the x-axis if they request no GPU resource (e.g., task E). For these tasks, all the unallocated GPUs are considered fragmented as none of them can be utilized.

Formal Description. We now give a formal description of the computation of $F_n(m)$, where we consider only GPU and CPU resources. That being said, the fragmentation measure can be easily generalized to a high-dimensional space with more resources such as memory and network.

We start with a few notations. Given a node n with G_n GPUs, denote by $R_n = \langle R_n^{\text{CPU}}, R_{n,1}^{\text{GPU}}, \cdots, R_{n,G_n}^{\text{GPU}} \rangle$ the unallocated resource vector, where $0 \le R_{n,g}^{\text{GPU}} \le 1$ for all GPU g. Let R_n^{GPU} be the scalar representation of the unallocated GPU vector, which is defined as the number of fully unallocated GPUs plus the maximum partial GPU ($u = f + p$ in §3.1), i.e., $R_n^{\text{GPU}} = \sum_g \lfloor R_{n,g}^{\text{GPU}} \rfloor + \max_g (R_{n,g}^{\text{GPU}} - \lfloor R_{n,g}^{\text{GPU}} \rfloor)$. For each task m, denote by $D_m = \langle D_m^{\text{CPU}}, D_m^{\text{GPU}} \rangle$ the resource demand vector, where $D_m^{\text{GPU}} \in [0, 1) \cup \mathbb{Z}^+$. We compute $F_n(m)$ in the following three cases.

Case-1 (Q-I, Q-II, and Q-IV): Task m cannot run on node n due to the lack of CPU or GPU resources, i.e., $D_m^{\text{CPU}} > R_n^{\text{CPU}}$ or $D_m^{\text{GPU}} > R_n^{\text{GPU}}$. In this case, all the unallocated GPUs are considered as fragments by task m. We have

$$F_n(m) = \sum_{1 \le g \le G_n} R_{n,g}^{\text{GPU}}. \tag{2}$$

Case-2 (Q-III): Task m can run on node m and has a GPU request, i.e., $D_m^{\text{CPU}} \le R_n^{\text{CPU}}$ and $0 < D_m^{\text{GPU}} \le R_n^{\text{GPU}}$. In this case, we check each unallocated GPU and those with insufficient capacity are identified as fragment in the view of task m. Note that if m requests one or more GPUs, all partial GPUs cannot be allocated and are hence fragmented. We have

$$F_n(m) = \sum_{1 \le g \le G_n} R_{n,g}^{\text{GPU}} \mathbb{1}(R_{n,g}^{\text{GPU}} < \min\{D_m^{\text{GPU}}, 1\}), \tag{3}$$

where $\mathbb{1}(\cdot)$ is an indicator function that returns 1 if the given condition holds, and 0 otherwise.

Case-3 (x-axis): Task m requests no GPU, i.e., $D_m^{\text{GPU}} = 0$. In this case, all the unallocated GPUs are deemed fragments by task m, as none of them can be utilized. We have $F_n(m)$ computed the same way as in Eqn. (2).

In essence, $F_n(m)$ measures the amount of available GPUs on node n that cannot be allocated to task m. Taking the expectation of $F_n(m)$ with respect to the popularity distribution of tasks (see Eqn. (1)), we obtain $F_n(M)$, which measures the unallocated GPU resources on node n that are *expected to be fragmented* (hence wasted) from the viewpoints of the entire workload M.

Fragmentation Rate. Once $F_n(M)$ is obtained, we compute the node's *fragmentation rate* as the ratio between the amount of fragmented GPUs and the unallocated GPUs, i.e.,

$$f_n(M) = \frac{F_n(M)}{\sum_{1 \le g \le G_n} R_{n,g}^{\text{GPU}}}. \tag{4}$$

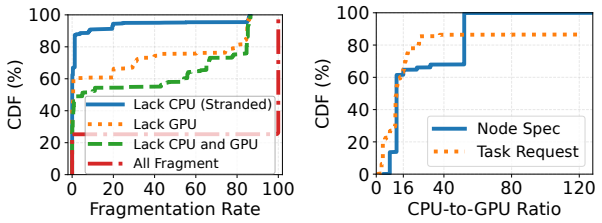
Intuitively, Eqn. (4) measures the severity of GPU fragmentation on a node.

Cluster-Level Fragmentation Measure. Given a cluster N and the target workload M, the cluster-level GPU fragmentation, denoted by $F_N(M)$, is the aggregate fragmentation of all nodes n in N, i.e.,

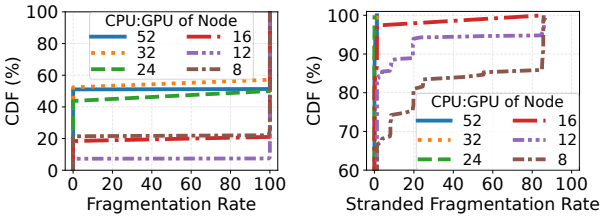
$$F_N(M) = \sum_{n \in N} F_n(M). \tag{5}$$

Normalizing $F_N(M)$ by the unallocated GPUs in the cluster gives the fragmentation rate of the cluster, i.e.,

$$f_N(M) = \frac{F_N(M)}{\sum_{n \in N} \sum_{1 \le g \le G_n} R_{n,g}^{\text{GPU}}}. \tag{6}$$



(a) CDF of node fragmentation rate. (b) CPU:GPU of nodes and tasks.



(c) Frag rate among various nodes. (d) Stranded frag rate among nodes.

Figure 6: Distribution of node fragmentation and CPU-to-GPU ratio of node and task in the fully packed cluster *H*.

3.3 Fragmentation Analysis in Action

The fragmentation measure described above can help operators better reason about the cluster state. To demonstrate its practical utility, we perform fragmentation analysis in various production clusters.

High Fragmentation Blocks Further Allocation. We reexamine a production cluster *H* previously described in §2.2. For each node, we measure its GPU fragmentation rate (Eqn. (4)) and depict the distribution in Figure 6a (red dash-dotted line). We observe that 25% of the nodes have GPUs fully allocated and are free of fragmentation, while the other 75% nodes measure over 99% fragmentation rate. This explains why the cluster cannot run any tasks even if it still has a capacity of 500 unallocated GPUs.

Breakdown Analysis of the Fragmentation Causes. Using the quadrant interpretation descried in §3.2, we can break down the fragmentation of a node into different causes: (1) having insufficient CPUs and GPUs to run tasks in Q-I (Figure 4b); (2) having insufficient GPUs to run tasks in Q-II, some GPU-sharing tasks in Q-III, and tasks that request different types of GPUs (not shown in Figure 4b); (3) having insufficient CPUs to run tasks in Q-IV (stranded GPUs); (4) running non-GPU tasks (x-axis) on a GPU node.

Figure 6a attributes the fragmentation rate to different causes and depicts their distributions. We see that the high fragmentation is primarily caused by the node lacking sufficient CPU and GPU resources (green dashed line), followed by lacking GPUs only (orange dotted line). This suggests that in cluster *H*, the allocation of CPUs and GPUs are relatively balanced. But still, a small number of nodes (4.6%) attribute stranded GPUs as the dominant factor (over 80%) of fragmentation (blue solid line).

Impact of CPU-to-GPU Ratio. In our analysis, we are interested in knowing which nodes are more likely to become fragmented and find the CPU-to-GPU ratio a good indicator. Figure 6b compares the CPU-to-GPU ratio of the node specs and the task requests. On one hand, 65% nodes (tasks) have (request) $\leq$ 16 CPUs per GPU, for which it is a good match between the node specs and workload demands. On the other hand, the cluster workload also contains 13% non-GPU tasks (with an infinite CPU-to-GPU ratio), and they account for 23% of all CPU requests. The existence of non-GPU tasks renders nodes with low CPU-to-GPU ratios more likely to become fragmented, especially with stranded GPUs.

Figure 6c correlates the fragmentation rate with the node's CPU-to-GPU ratio. Among the nodes with CPU-to-GPU ratio $\leq$ 16, 80% of them measure high fragmentation rate over 98% (dark red lines near the bottom). This proportion drops to only 47% when it comes to the nodes with 52 CPUs per GPU (light blue lines in the center). As for the fragmentation caused by stranded GPUs, Figure 6d shows that they are more commonly observed on the low-CPU nodes (e.g., 8 or 12 CPUs per GPU). We therefore recommend adding more high-CPU nodes to the cluster for reduced fragmentation and improved utilization.

Fragmentation in a Less-Crowded Cluster. Fragmentation is not a unique problem to fully packed clusters. Referring back to Figure 2, we measured the the cluster-wide fragmentation rate (Eqn. (5)) between 21% and 42% in a 1280-GPU cluster in a normal (off-peak) period with 77.6% average GPU allocation rate. Breakdown analysis further shows that 60% of the fragmentation was caused by GPU shortage. This highlights the importance of reducing fragmentation by packing existing GPU tasks, even in periods where the cluster is less crowded. Additionally, non-GPU tasks contributed only 13% to fragmentation in this cluster.

Running CPU Workloads in GPU Clusters. Our fragmentation analysis has also identified a pathological case. We have observed in a large cluster *B* consisting of 6.4k GPUs and 750k CPU cores, there were 84% of tasks requesting no GPU, resulting in many pending GPU tasks when the GPU and CPU allocation ratios reached only 80% and 86%, respectively. This led to the waste of 1.2k GPUs, with 95% of them considered being stranded. The main cause of this issue was *the lack of CPU reservation on GPU nodes* when scheduling non-GPU tasks. To address this problem, we recommend migrating non-GPU tasks to CPU nodes or even CPU clusters in order to decrease fragmentation and improve resource utilization. However, implementing this solution requires coordination and collaboration in areas such as scheduling, quota design, admission control, and capacity planning.

4 Fragmentation Gradient Descent

The fragmentation measure not only helps operators reason about the cluster state, but can also be used to guide task scheduling. In this section, we present the Fragmentation Gra-

dient Descent (FGD) algorithm that schedules tasks towards the direction of the steepest descent of fragmentation (§4.2), thereby achieving the maximum GPU allocation rate. We start with a description of the scheduling problem (§4.1).

4.1 Online Task Scheduling

We consider a GPU cluster managed by a container orchestration system such as Kubernetes [1] and Borg [30], in which tasks are submitted as pods and maintained in a queue. In its simplest form, tasks are scheduled in a first-come-first-served (FCFS) manner. For each task pod, the scheduler finds the best node and, if necessary, GPU(s) for it to run on. If no node can be assigned, the pod remains unscheduled and is pending for another scheduling attempt (e.g., placed to the end of the queue after a certain timeout). This formulates an *online scheduling problem*, in which tasks are revealed sequentially to the scheduler for placement decision making.

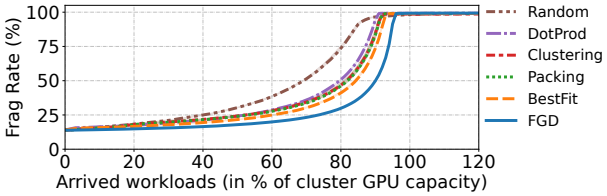
Fragmentation as a Metric. Each time a pod is assigned to a node, the remaining resources decrease, moving the node closer to the origin in the resource plane (Figure 4b). As a result, the fragmentation region (Q-I, Q-II, and Q-IV) expands and the fragmentation rate grows. Figure 7a depicts the growing fragmentation rate as the arriving tasks are scheduled under different policies in our trace-driven emulations (details in §6.1). Compared with the absolute measure (§3.1), the fragmentation rate is more sensitive to the placement decision and can be used as an indicator of the scheduling quality: higher fragmentation rate suggests a poorer scheduling decision.

Unlike the fragmentation rate, the fragmentation amount has no clear trend of growing or decreasing, as shown in Figure 7b. This is because by Eqn. (5), it is the product of the fragmentation rate and the unallocated GPUs – whilst the former grows as more tasks are scheduled, the latter decreases. Also note that the fragmentation starts with 13% of the total GPU capacity – all attributed to non-GPU tasks – but ends up at different degrees under different policies when the cluster is fully packed. At that point, all unallocated GPUs are fragmented (100% fragmentation rate in Figure 7a).

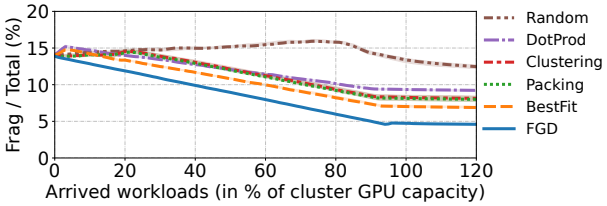
4.2 FGD Algorithm

Key Insight. From the previous discussions, we see that an effective approach to minimizing fragmentation is to *suppress the growth of fragmentation rate as much as possible*. This can be achieved by scheduling tasks towards the steepest descent of fragmentation, a heuristic called Fragmentation Gradient Descent or FGD.

Algorithm Description. Algorithm 1 formalizes the description of FGD scheduling. Given a task (pod) to schedule, the scheduler first filters out all the unavailable nodes with insufficient resources or unsatisfied placement constraints (lines 3–4), such as the lack of requested GPU types. The scheduler



(a) Fragmentation rate grows to 100% as more resources are allocated.



(b) Percentage of fragmented GPUs to total resources under our measure. It reveals the differences between various strategies from the early stage.

Figure 7: FGD pursues the lowest fragmentation among various policies in scheduling production workloads (more results under the same experiment settings are shown in Figure 9).

then *hypothetically* assigns the task to each node and calculates the increment (can be negative) of fragmentation that would be caused by each assignment. In case a task requests a partial GPU, the hypothetical assignment needs to try each GPU as well (line 6). Note that the hypothetical assignment can be performed *in parallel* for acceleration (lines 2–7). The scheduler finally assigns the task to the node (and the GPU) that causes the minimum increment of fragmentation (line 9).

Complexity and Scalability. FGD has a low computational complexity and scales to a large cluster. For a cluster with N nodes, the score of each node can be evaluated *in parallel* (Algorithm 1 line 2) and the scheduler simply selects the minimum. Besides, the scale of M is also limited since it only represents the number of *distinct* task resource requirements (e.g., 80 among 7.6k tasks). In our evaluation (§6), each scheduling decision can be made in *hundreds of milliseconds* on a cluster of $N = 1.2k$ nodes.

A Running Example. To better illustrate the scheduling process of FGD, we refer to Figure 8 for a running example. Consider a node with $\langle 0.5, 1\rangle$ unallocated GPUs. The target workload has three tasks with equal popularity, each requesting 0.3 GPUs (task-A), 0.5 GPUs (task-B) , and 0.7 GPUs (task-C). Originally, only GPU-A is considered fragmented (by task-C only). It thus measures the fragmentation of $0.5 \times 1/3 = 1/6$ GPUs. Assume that task-A arrives first. Assigning it to GPU-A increases the fragmentation to 0.2 GPUs whereas assigning it to GPU-B results in no increase. FGD hence assigns task-A to GPU-B, leaving the node with $\langle 0.5, 0.7\rangle$ unallocated GPUs. Next for task-B, FGD assigns it to GPU-A, reducing the fragmentation to 0. In comparison, assigning it to GPU-B would increase the fragmentation by 0.2 GPUs. Finally for task-C, it has no choice but to run on GPU-B, the only GPU with the sufficient capacity.

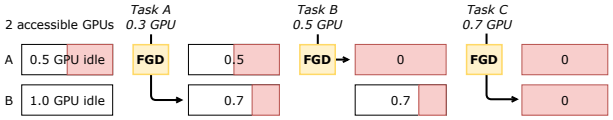


Figure 8: A running example of FGD scheduling. The target workload contains three tasks with equal popularity.

Algorithm 1: Task scheduling of FGD

Input : Cluster N, incoming task m, target workload M
Output : Assigned node n^*

Initialize node score set $\mathcal{S} \leftarrow \varnothing$, and output $n^* \leftarrow \varnothing$.
parallel for *node* $n \in N$ **do**
  if *Insufficient resources* $\|$ *constraints not met* **then**
    Return ▷ Filter out unavailable nodes
  $n^- \leftarrow \text{AssignTaskToNode}(m,n)$ ▷ Hypothetically
  $\Delta \leftarrow F_{n^-}(M) - F_n(M)$ ▷ Fragmentation increment
  $\mathcal{S} \leftarrow \mathcal{S} \cup (n,\Delta)$
if $\mathcal{S} \neq \varnothing$ **then**
  $n^* \leftarrow \arg\min_{n \in \mathcal{S}} \Delta$ ▷ pick the node with the least Δ.

As FGD considers both the resource availability and task distribution, it exhibits distinct behavior compared to other scheduling algorithms. Back to the provided example, for task-A scheduling, the best-fit (and similar bin packing policies) would prioritize GPU-A (0.5 GPUs idle) over GPU-B (1 GPU idle), whereas for task-B scheduling, the worst-fit (and other load-balancing policies) would choose GPU-B (0.7 GPUs idle) over GPU-A (0.5 GPUs idle). These policy preferences diverge from the scheduling choices made by FGD.

5 System Implementation

We have implemented a prototype scheduling system on top of Kubernetes v1.25.0 [1] in over 10k lines of Go codes. Our system consists of two main components: a standalone scheduler and an event-driven emulator.

A Standalone Scheduler. Kubernetes provides a pluggable architecture called the *scheduling framework* [2] for developers to implement a customized scheduler. Following this standard approach, we have implemented FGD and many other scheduling policies as individual score plugins. The scheduler listens for task creation events from the Kubernetes API server and maintains a queue to cache submitted tasks which are scheduled on a first-come-first-served basis.

Also, to enable fine-grained GPU allocation, which is not supported by native Kubernetes, we have implemented a *GPU-sharing* plugin to manage GPU resources. It filters out nodes with insufficient GPUs or mismatched GPU types, assigns tasks to the node with the *highest* scheduling score, and keeps track of the allocatable GPU resources on each node.

Event-Driven Emulator. The event-driven emulator interacts with the API server to manage the creation and deletion of nodes and tasks. It supports two modes: *high-fidelity simulation*, which can receive production traces as input and simulate the scheduling process in a large cluster consisting of tens of thousands of GPUs within a few hours; and *real deployment*, which can take over a production cluster with valid certificates and create task pods on real nodes. The goal of our system is to study the packing efficiency and resource fragmentation of different scheduling policies, which are critical in large clusters. Although it is not possible to perform experiments on real clusters with thousands of nodes, we use the *high-fidelity simulation* mode in our evaluation. This mode uses real traces as input and simulates task placements and resource allocations based on the scheduling logic, which yields the same scheduling results as real deployment.

6 Evaluation

In this section, we conduct extensive experiments to demonstrate the effectiveness of our scheduling policy FGD. We first compare FGD with several state-of-the-art mechanisms in scheduling production workloads (§6.2). Further, we examine the generality of FGD in various scenarios, covering a variety of traces featured by GPU-sharing (§6.3), multi-GPU (§6.4), GPU-type-constrained (§6.5), and non-GPU tasks (§6.6).

6.1 Methodology

Baselines. We compare FGD with five state-of-the-art heuristic policies for scheduling GPU workloads:

1. Best-fit (BestFit) [11,21,24] assigns tasks to the node with the least remaining resources, computed as the weighted sum of all resource dimensions.
2. Dot-product (DotProd) [8, 23, 24] allocates tasks to the node with the smallest dot-product value between the node's remaining resources and the task demands.
3. GPU Packing (Packing) [31] prioritizes task assignment to occupied GPUs, followed by idle GPUs on occupied nodes, and finally to fully idle nodes. The intuition is to reserve available resources for multi-GPUs tasks.
4. GPU Clustering (Clustering) [33] packs the tasks of the same GPU request together (GPU-sharing tasks are packed together). It avoids heterogeneous distribution of task resource requirements on the same node.
5. Random-fit (Random) distributes the task randomly to any node that meets the requirements for load balancing.

Since most of the policies above provide no native support of GPU-sharing workloads, we made some simple extensions as follows: 1) GPU resources of multi-GPU nodes are summed up as one dimension; 2) GPU-sharing tasks scheduled to a node are placed on the available GPU with the least remaining resources (i.e., BestFit); 3) multi-resource vectors are normalized by the maximum node capacity in the cluster.

GPU Request per Task	0	(0, 1)	1	2	4	8
Task Population (%)	13.3	37.8	48.0	0.2	0.2	0.5
Total GPU Reqs. (%)	0	28.5	64.2	0.5	1.0	5.8

Table 1: Distribution of tasks in the traces of cluster H.

Monte-Carlo Workload Inflation and Metrics. To assess the cluster's ability to accommodate workload under a given scheduling policy, we employ the Monte-Carlo *workload inflation* approach [29]. Specifically, we repeatedly submit tasks to the cluster for scheduling until they no longer fit. The tasks are randomly sampled from the traces with replacement [12], along with their resource requests and scheduling constraints. We repeatedly conduct each experiment 10 times and report the average and standard deviation of the key metric—the percentage of unallocated GPUs in the cluster when cumulative GPU requests reach 100% of the cluster capacity[5].

6.2 Allocation of Original Production Traces

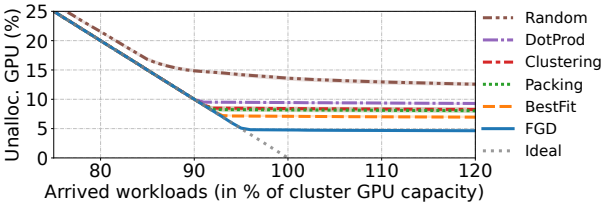
To evaluate the performance of different scheduling policies, we emulate the scheduling of over 8k tasks on the heterogeneous cluster H (§2.2) with 6.2k GPUs. As shown in Table 1, the majority tasks are the 1-GPU and GPU-sharing ones. Despite the small population of 8-GPU tasks, they still occupy a non-negligible portion in terms of requested GPUs (5.8%).

FGD Saves More Unallocatable GPUs. Figure 9a illustrates how the unallocated GPUs in the cluster decrease as tasks arrive under different scheduling policies. Ideally, the arrived tasks should be scheduled successfully until all GPUs are allocated (gray dotted line). However, in practice, the fragmentation rate also increases with the allocation of arrived tasks (Figure 7a). After a certain point when the fragmentation rate reaches 100%, all unallocated GPUs become fragmented and the allocation rate plateaus at a certain level, depending on the used scheduling policy (e.g., DotProd at 90%). Compared to classic scheduling policies, FGD achieves the highest allocation rate and reduces wasted GPUs by 33–49%, which translates to the additional allocation of 150–290 GPUs.

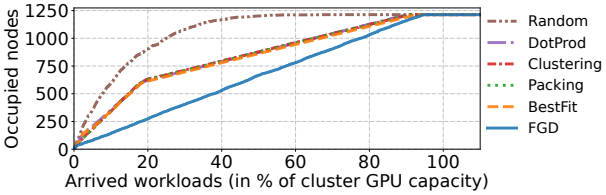
FGD Occupies Fewer Nodes in Scheduling. Figure 9b shows how the number of nodes that have at least one task running (occupied nodes) grows during the scheduling process under different policies. We observe that FGD packs tasks onto nodes whenever possible, leading to the fewest occupied nodes among all policies. Especially in early stages when the GPU allocation rate is 20%, FGD requires 55–70% fewer nodes to host all the tasks compared to other policies. As a result, FGD can enable significant savings in energy consumption (on premises) or operational cost (on clouds).

FGD Schedules More GPU-Sharing and One-GPU Tasks.

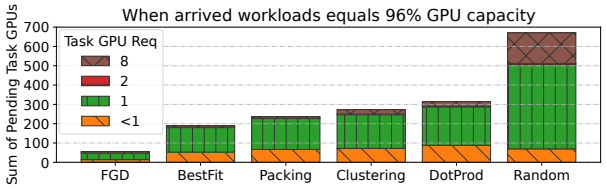
[5] Although a slight decrease in unallocated GPUs may still occur thereafter (thanks to few tiny tasks), the cluster is oversubscribed and rejects the majority of incoming tasks; this is not a desirable state to be considered in our metric.



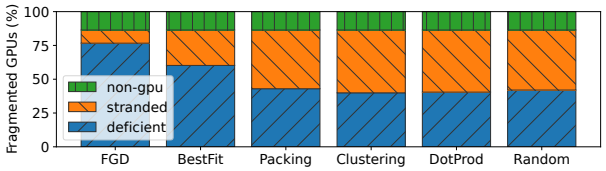
(a) The percentage of unallocated GPUs given arriving workloads.



(b) The number of GPU nodes occupied during the scheduling.



(c) GPU requests of failed tasks when the cluster is almost full (i.e., cumulative GPU requests reach 96% of the cluster capacity).



(d) The breakdown of GPU fragmentation into three causes.

Figure 9: Performance comparison of FGD and various scheduling policies. FGD outperforms all baselines with fewer unallocated GPUs and failed tasks.

Figure 9c depicts the distribution of GPU requests of unscheduled tasks under different policies when the cluster is almost full (i.e., the cumulative GPU requests of arrived tasks reach 96% of the cluster capacity). Except for Random, all policies schedule multi-GPUs tasks well. Compared to classic policies, FGD schedules up to 5.9× GPU-sharing tasks and 6.6× one-GPU tasks while reserving multi-GPU slots.

Early Detection of Fragmentation. Being able to detect fragmentation early on is the key to improving the GPU allocation rate. Referring back to Figure 7a, we see that FGD consistently achieves the lowest fragmentation rate among all policies, indicating better scheduling quality. As tasks arrives, the advantage of FGD becomes larger. When the cumulative requests grow to 90% of the total capacity, although most policies can still successfully schedule tasks, FGD outperforms alternative policies with 24–44% lower fragmentation rate. Figure 9d further decomposes the fragmented GPUs generated by each scheduling policy. It shows that $< 10\%$ of the fragmented GPUs by FGD come from stranded resources,

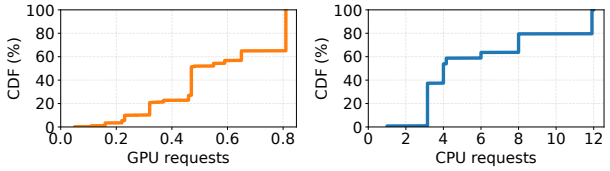


Figure 10: Distribution of resource requests of GPU-sharing tasks in cluster *H*.

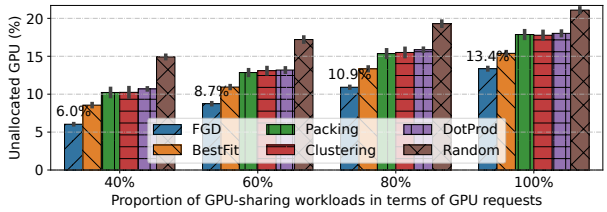


Figure 11: Scheduling results when the traces contain a varying number of GPU-sharing tasks. Y-axis shows the percentage of unallocated GPUs when the overall GPU requests of the arrived tasks reach 100% of the cluster's GPU capacity (same for the following figures).

which is 63–79% fewer than other policies. This confirms that reducing the amount of stranded resources provides better packing efficiency, which is in line with Borg's insights [30].

6.3 Allocation of More GPU-Sharing Tasks

GPU-sharing techniques enable finer-grained resource allocation. For GPU-sharing tasks, their resource requests are typically based on the actual usage of model execution. Figure 10 depicts the CDF of resource requests of GPU-sharing tasks. We observe that around 35% of GPU-sharing tasks request 0.8 GPUs, while no more than 5% tasks request less than 0.2 GPUs. This indicates that fractional GPUs (i.e., GPUs with partial resources allocated), after scheduling 0.8-GPU tasks, will mostly become fragmented. Nonetheless, there are still many packing combinations of tasks that can effectively maximize the use of shared GPUs. For example, 0.32-GPU tasks and 0.65-GPU tasks account for nearly 10% tasks, respectively—they can be placed together to reduce fragmentation.

FGD Tailors Resources for GPU-Sharing Tasks. To evaluate the impact of GPU-sharing tasks on scheduling, we construct different experimental settings based on production traces (§6.2), increasing the proportion of GPU-sharing tasks while keeping their resource requests in line with the original distribution. Fragmented GPUs increase as the proportion of GPU-sharing tasks rises (Figure 11). FGD considers the chances that remaining resources after packing can be utilized and tailors proper resources for subsequent tasks (Figure 8). It outperforms the other policies in all cases; even when the cluster has only GPU-sharing tasks, FGD reduces the unallocated resources of 125–290 GPUs.

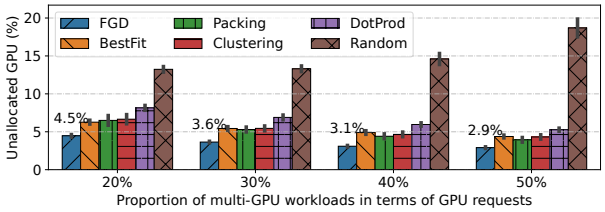


Figure 12: Various proportions of multi-GPU tasks.

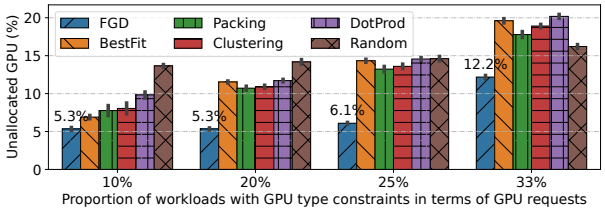


Figure 13: Proportions of task with GPU-type constraint.

6.4 Allocation of More Multi-GPUs Tasks

To better understand the scheduling impact of multi-GPUs tasks, we artificially adds more of them to the input workload, increasing their GPU requests from 20% to 50% of the overall demands. Figure 12 shows the scheduling results given by different policies. We observe that all policies except Random perform well when the workloads contain more multi-GPUs tasks. The reasons are two-fold: (1) As the population of multi-GPUs tasks increases, the scheduling impact caused by GPU-sharing tasks becomes less significant, making the classic multi-resource bin packing algorithms more effective. (2) Reserving multi-GPUs slot is desirable due to the presence of multi-GPU tasks. Compared to other scheduling policies, FGD avoids stranded resources and judiciously reserves GPU cards on nodes, reducing unallocated GPUs by 26–45% even when the multi-GPU tasks account for 50% of GPU requests.

6.5 Allocation of Tasks with GPU Constraints

In previous experiments, we have mainly considered fragmentation caused by *mismatches of resources*. Yet, production tasks also have placement constraints of desired GPU type. They typically request high-end GPUs for better performance; models are sometimes optimized for a specific generation of GPU. In our clusters, around 33% of GPU tasks have specified desired GPU types, while the rest can run on any GPUs. Heterogeneous GPUs in a cluster are often unevenly distributed, posing a challenge to resource scheduling.

Figure 13 shows the performance of different policies with varying numbers of tasks that specify GPU-type constraints. Classic heuristic policies only consider the alignment of resource demands, which is likely to cause severe fragmentation. In comparison, FGD reduces 32–40% of fragmented GPUs when tasks with GPU-type constraints account for 33% of the total GPU requests. FGD selectively reserves popular GPUs to avoid fragmentation caused by mismatches of GPU types.

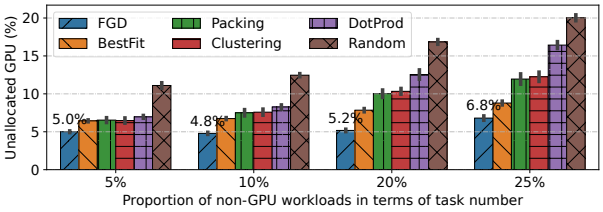


Figure 14: Various proportions of non-GPU tasks.

6.6 Allocation of More Non-GPU Tasks

In GPU clusters, there are often many tasks that request no GPU but other resources, such as CPU, memory, and disk. Examples include parameter servers and data processing tasks. These non-GPU tasks are usually not resource-intensive. Yet, if scheduled unwisely, they may cause some GPUs to become stranded. To evaluate their impact to scheduling, we vary the number of non-GPU tasks and compare the performance of different scheduling policies in Figure 14. FGD consistently maintains the unallocated GPUs at a low level, demonstrating its strong capability of avoiding stranded resources when making scheduling decisions. In fact, as the proportion of non-GPU tasks increases from 5% to 20%, fragmented GPUs caused by FGD increase by $< 3\%$ while other policies increase fragmentation by 18–45%.

7 Discussion

Scheduler Independence of Fragmentation Metrics. Our proposed fragmentation metric quantifies how fragmented the current cluster is considering only the next incoming task. This narrow focus ensures *scheduler independence*. Specifically, if we consider two or more incoming tasks, each node would need to determine the likelihood that the first task has consumed its resources when assessing the fragmentation measured by the second task. This determination would involve the scheduling policy. In contrast, our one-step fragmentation metric determines if a node is fragmented based solely on whether its remaining resources can be fully utilized by the next task, agnostic to the scheduler or other nodes.

Combining with Other Heuristics. Guided by the one-step fragmentation metric, schedulers may perform suboptimally in early stages when the cluster has abundant resources and little fragmentation is observed. However, the fragmentation-guided scheduling heuristic can be combined with other heuristics to navigate this initial period. For example, the scheduler could fall back to a best-fit approach if no increase in fragmentation is detected.

8 Related Work

Resource Fragmentation. Many research works address resource fragmentation in clusters [8,29,30,39,40]. For example, Tetris [8] shows that fair schedulers usually result in fragmented resources and delayed job completion, and proposes to combine them with an alignment-based policy. Borg [30] uses a hybrid scoring model to reduce the amount of stranded resources. HiveD [39] eliminates external fragmentation across multiple tenants by constructing virtual clusters. However, these works do not give a formal definition of fragmentation. We propose a statistical measure to quantify the degree of resource fragmentation and use it to guide task scheduling.

Multi-Resource Bin Packing. Resource allocation is often formulated as a multi-dimensional bin packing problem, which has been extensively studied [24]. Many heuristic policies, such as best-fit, vector alignment scoring (dot-product), are proven effective with high packing efficiency for scheduling big data analytics workloads [8] and virtual machines consolidation [11, 23]. However, it can be observed from our experiments that they do not work well in GPU sharing scenarios. Our work advocates resource allocation from the perspective of mitigating fragmentation, which can further reduce wasted resources.

GPU Cluster Scheduling. Recent works on GPU scheduling concern various objectives, such as cluster utilization [21, 32–34, 37, 40], job completion time [10, 18, 25, 26], job performance [22], and fairness [7, 20]. Although some research efforts (e.g., Gandiva [33], Salus [37], AntMan [34], TGS [32]) exploit GPU sharing techniques to improve resource *utilization*, they do not address GPU fragmentation. Our work complements them by reducing fragmentation and improving the GPU allocation rate.

9 Conclusion

In this paper, we have identified the significant fragmentation problem caused by GPU-sharing workloads that are increasingly deployed in production clusters. To address this problem, we have proposed a novel metric to statistically quantify the degree of GPU fragmentation caused by various sources. Based on this measure, we have proposed a simple, yet effective scheduling approach, named Fragmentation Gradient Descent (FGD), that schedules tasks towards the direction of the steepest descent of GPU fragmentation. Large-scale trace-driven emulations show that FGD substantially achieves higher GPU allocation rate compared to existing packing-based heuristics, saving hundreds of GPUs.

10 Acknowledgment

We thank our shepherd Feng Zhang and anonymous reviewers for their valuable comments. We also thank colleagues from Alibaba Group for their feedback and assistance in the early stage of this work. This work was supported in part by the Alibaba Innovative Research (AIR) Grant and RGC GRF Grants (#16213120 and #16202121). Qizhen Weng was supported in part by the Hong Kong PhD Fellowship Scheme.

References

[1] Kubernetes: Production-grade container orchestration. https://kubernetes.io, 2023.

[2] Kubernetes scheduling framework. https://kubernetes.io/docs/concepts/scheduling-eviction/scheduling-framework/, 2023.

[3] NVIDIA multi-instance GPU, seven independent instances in a single GPU. https://www.nvidia.com/en-us/technologies/multi-instance-gpu/, 2023.

[4] Remzi H. Arpaci-Dusseau and Andrea C. Arpaci-Dusseau. *Operating Systems: Three Easy Pieces*. 2018.

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

[6] Zhihao Bai, Zhen Zhang, Yibo Zhu, and Xin Jin. PipeSwitch: Fast pipelined context switching for deep learning applications. In *Proc. USENIX OSDI*, 2020.

[7] Shubham Chaudhary, Ramachandran Ramjee, Muthian Sivathanu, Nipun Kwatra, and Srinidhi Viswanatha. Balancing efficiency and fairness in heterogeneous GPU clusters for deep learning. In *Proc. ACM EuroSys*, 2020.

[8] Robert Grandl, Ganesh Ananthanarayanan, Srikanth Kandula, Sriram Rao, and Aditya Akella. Multi-resource packing for cluster schedulers. In *Proc. ACM SIGCOMM*, 2014.

[9] Jing Gu, Shengbo Song, Ying Li, and Hanmei Luo. GaiaGPU: Sharing GPUs in container clouds. In *Proc. IEEE ISPA/IUCC/BDCloud/SocialCom/SustainCom*, 2018.

[10] Juncheng Gu, Mosharaf Chowdhury, Kang G. Shin, Yibo Zhu, Myeongjae Jeon, Junjie Qian, Hongqiang Liu, and Chuanxiong Guo. Tiresias: A GPU cluster manager for distributed deep learning. In *Proc. USENIX NSDI*, 2019.

[11] Ori Hadary, Luke Marshall, Ishai Menache, Abhisek Pan, Esaias E. Greeff, David Dion, Star Dorminey, Shailesh Joshi, Yang Chen, Mark Russinovich, and Thomas Moscibroda. Protean: VM allocation service at scale. In *Proc. USENIX OSDI*, 2020.

[12] John Hammersley. *Monte carlo methods*. Springer Science & Business Media, 2013.

[13] Mingcong Han, Hanze Zhang, Rong Chen, and Haibo Chen. Microsecond-scale preemption for concurrent GPU-accelerated DNN inferences. In *Proc. USENIX OSDI*, 2022.

[14] Kim Hazelwood, Sarah Bird, David Brooks, Soumith Chintala, Utku Diril, Dmytro Dzhulgakov, Mohamed Fawzy, Bill Jia, Yangqing Jia, Aditya Kalro, James Law, Kevin Lee, Jason Lu, Pieter Noordhuis, Misha Smelyanskiy, Liang Xiong, and Xiaodong Wang. Applied machine learning at Facebook: A datacenter infrastructure perspective. In *Proc. IEEE HPCA*, 2018.

[15] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proc. IEEE/CVF CVPR*, 2016.

[16] Qinghao Hu, Peng Sun, Shengen Yan, Yonggang Wen, and Tianwei Zhang. Characterization and prediction of deep learning workloads in large-scale GPU datacenters. In *Proc. ACM/IEEE SC*, 2021.

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

[18] Tan N. Le, Xiao Sun, Mosharaf Chowdhury, and Zhenhua Liu. AlloX: Compute allocation in hybrid clusters. In *Proc. ACM EuroSys*, 2020.

[19] Jiamin Li, Hong Xu, Yibo Zhu, Zherui Liu, Chuanxiong Guo, and Cong Wang. Lyra: Elastic scheduling for deep learning clusters. In *Proc. ACM EuroSys*, 2023.

[20] Kshiteej Mahajan, Arjun Balasubramanian, Arjun Singhvi, Shivaram Venkataraman, Aditya Akella, Amar Phanishayee, and Shuchi Chawla. Themis: Fair and efficient GPU cluster scheduling. In *Proc. USENIX NSDI*, 2020.

[21] Jayashree Mohan, Amar Phanishayee, Janardhan Kulkarni, and Vijay Chidambaram. Looking beyond GPUs for DNN scheduling on multi-tenant clusters. In *Proc. USENIX OSDI*, 2022.

[22] Deepak Narayanan, Keshav Santhanam, Fiodar Kazhamiaka, Amar Phanishayee, and Matei Zaharia. Heterogeneity-aware cluster scheduling policies for deep learning workloads. In *Proc. USENIX OSDI*, 2020.

[23] Rina Panigrahy, Vijayan Prabhakaran, Kunal Talwar, Udi Wieder, and Rama Ramasubramanian. Validating heuristics for virtual machines consolidation. Technical report, Microsoft Research, 2011.

[24] Rina Panigrahy, Kunal Talwar, Lincoln Uyeda, and Udi Wieder. Heuristics for vector bin packing. Technical report, Microsoft Research, 2011.

[25] Yanghua Peng, Yixin Bao, Yangrui Chen, Chuan Wu, and Chuanxiong Guo. Optimus: An efficient dynamic

resource scheduler for deep learning clusters. In *Proc. ACM EuroSys*, 2018.

[26] Aurick Qiao, Sang Keun Choe, Suhas Jayaram Subramanya, Willie Neiswanger, Qirong Ho, Hao Zhang, Gregory R. Ganger, and Eric P. Xing. Pollux: Co-adaptive cluster scheduling for goodput-optimized deep learning. In *Proc. USENIX OSDI*, 2021.

[27] Lin Shi, Hao Chen, Jianhua Sun, and Kenli Li. vCUDA: GPU-accelerated high-performance computing in virtual machines. *IEEE Trans. Comput.*, 61(6):804–816, 2012.

[28] Dharma Shukla, Muthian Sivathanu, Srinidhi Viswanatha, Bhargav Gulavani, Rimma Nehme, Amey Agrawal, Chen Chen, Nipun Kwatra, Ramachandran Ramjee, Pankaj Sharma, Atul Katiyar, Vipul Modi, Vaibhav Sharma, Abhishek Singh, Shreshth Singhal, Kaustubh Welankar, Lu Xun, Ravi Anupindi, Karthik Elangovan, Hasibur Rahman, Zhou Lin, Rahul Seetharaman, Cheng Xu, Eddie Ailijiang, Suresh Krishnappa, and Mark Russinovich. Singularity: Planet-scale, preemptive and elastic scheduling of AI workloads. *arXiv preprint arXiv:2202.07848*, 2022.

[29] Abhishek Verma, Madhukar Korupolu, and John Wilkes. Evaluating job packing in warehouse-scale computing. In *Proc. IEEE CLUSTER*, 2014.

[30] Abhishek Verma, Luis Pedrosa, Madhukar Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. Large-scale cluster management at Google with Borg. In *Proc. ACM EuroSys*, 2015.

[31] Qizhen Weng, Wencong Xiao, Yinghao Yu, Wei Wang, Cheng Wang, Jian He, Yong Li, Liping Zhang, Wei Lin, and Yu Ding. MLaaS in the Wild: Workload analysis and scheduling in large-scale heterogeneous GPU clusters. In *Proc. USENIX NSDI*, 2022.

[32] Bingyang Wu, Zili Zhang, Zhihao Bai, Xuanzhe Liu, and Xin Jin. Transparent GPU sharing in container clouds for deep learning workloads. In *Proc. USENIX NSDI*, 2023.

[33] Wencong Xiao, Romil Bhardwaj, Ramachandran Ramjee, Muthian Sivathanu, Nipun Kwatra, Zhenhua Han, Pratyush Patel, Xuan Peng, Hanyu Zhao, Quanlu Zhang, Fan Yang, and Lidong Zhou. Gandiva: Introspective cluster scheduling for deep learning. In *Proc. USENIX OSDI*, 2018.

[34] Wencong Xiao, Shiru Ren, Yong Li, Yang Zhang, Pengyang Hou, Zhi Li, Yihui Feng, Wei Lin, and Yangqing Jia. AntMan: Dynamic scaling on GPU clusters for deep learning. In *Proc. USENIX OSDI*, 2020.

[35] Ting-An Yeh, Hung-Hsin Chen, and Jerry Chou. KubeShare: A framework to manage GPUs as first-class and shared resources in container cloud. In *Proc. ACM HPDC*, 2020.

[36] Fuxun Yu, Di Wang, Longfei Shangguan, Minjia Zhang, Xulong Tang, Chenchen Liu, and Xiang Chen. A survey of large-scale deep learning serving system optimization: Challenges and opportunities. *arXiv preprint arXiv:2111.14247*, 2021.

[37] Peifeng Yu and Mosharaf Chowdhury. Salus: Fine-grained GPU sharing primitives for deep learning applications. In *Proc. MLSys*, 2020.

[38] Yidong Yuan, Xuemin Lin, Qing Liu, Wei Wang, Jeffrey Xu Yu, and Qing Zhang. Efficient computation of the skyline cube. In *Proc. VLDB*, 2005.

[39] Hanyu Zhao, Zhenhua Han, Zhi Yang, Quanlu Zhang, Fan Yang, Lidong Zhou, Mao Yang, Francis C. M. Lau, Yuqi Wang, Yifan Xiong, and Bin Wang. HiveD: Sharing a GPU cluster for deep learning with guarantees. In *Proc. USENIX OSDI*, 2020.

[40] Yihao Zhao, Yuanqiang Liu, Yanghua Peng, Yibo Zhu, Xuanzhe Liu, and Xin Jin. Multi-resource interleaving for deep learning training. In *Proc. ACM SIGCOMM*, 2022.