HiveD: How to Share a GPU Cluster for Deep Learning

Paper #91

Abstract

Deep learning training on GPUs is emerging to be a significant computation workload. Simply retrofitting a big-data infrastructure and scheduler, such as YARN, to share a GPU cluster for this workload leads to unexpected anomalies that disincentivize sharing, due to the stringent gang-scheduled strong affinity requirements, as well as the observed fragmentation as a result of non-uniformity in the workload.

We propose the notion of *Virtual Private Clusters* with multi-level cell structures to ensure a strict *sharing safety* condition. The multi-level cell structures reflect the natural hierarchy of affinity in a typical GPU cluster. Such multi-level cell structures can be managed and allocated with an elegant and efficient *buddy cell allocation* algorithm under the *hierarchical uniform composability* condition. We further introduce *cell exchange* to allow tenants to manage the fluctuations and diversity of their workloads by exchanging cells among them to increase the utilization of those cells. The buddy cell allocation algorithm can be naturally extended to manage jobs with multiple levels of priorities, where low-priority jobs can take advantage of unused resources, but might be preempted by regular jobs arriving later.

We analyze the trace from a production multi-tenant GPU cluster to reveal the unique characteristics of the deep learning training workload, as well as the resulting sharing safety violations. We further use a combination of trace-driven simulation and real deployment on a public cloud to validate our design and show how our mechanisms can be used to strike the right balance between reducing queuing delay and improving overall utilization with both regular and low-priority jobs, while ensuring sharing safety.

1 Introduction

Deep learning training on a GPU cluster is becoming a major computing workload and comes in a variety of forms including long-running, large jobs that require multiple GPUs with strong affinity (e.g., within the same node or even on the same PCIe switch), as well as a large number of single-GPU jobs, often for AutoML-like explorations [9, 40, 46, 57].

It is a common practice for an organization to train deep learning models in a multi-tenant GPU cluster, with resource quota (in number of GPUs, plus associated resources such as CPU and memory) providing resource guarantees for each tenant, coupled with a mechanism to allow a tenant to tap into unused resources whenever they are available [50].

Surprisingly, our study on a trace from a production multi-tenant GPU cluster reveals unexpected anomalies that cause tenants to prefer their own (small) private cluster over a (large) shared cluster. We first observe that, due to fragmentation, a tenant, even with sufficient unused GPUs in the quota, often fails to get enough GPUs that satisfy the required affinity and has to choose between degraded performance (when sacrificing affinity) and significant queuing delay (to wait for GPUs with the desirable affinity). This is the case even with an affinity-aware scheduling algorithm to schedule jobs, because a large number of (single-GPU) jobs get scheduled and run for unpredictable amounts of time. The situation becomes even worse in a shared cluster mainly due to an implicit type of interference: spreading of GPU fragmentation across tenants.

We aim to design a solution to sharing a GPU cluster for deep learning training that satisfies the *sharing safety* condition, which intuitively means that a tenant allocated the same amount of resource in a shared cluster should experience no worse than its own private cluster. Our study already shows that a simplistic quota system based on only the number of GPUs is fundamentally flawed as GPUs are not created equal. We therefore use *multi-level cell structures* to capture the different levels of affinity that a group of GPUs satisfies. Those cell structures naturally form a hierarchy in a typical GPU cluster; e.g., from GPUs in a pod, to GPUs in a node, then to GPUs connected to a CPU socket, and subsequently to GPUs attached to a PCIe switch, and so on.

In this paper, we present HiveD for sharing a GPU cluster targeting deep learning training workload. HiveD introduces the notion of a *virtual private cluster* (abbreviated as VC), which is defined not only in the number of GPUs, but also in the cell structures modeling a private cluster. HiveD then ensures sharing safety in that, if a sequence of requests for cells can be satisfied in a private cluster, then it should be satisfied in the corresponding virtual private cluster and the shared physical cluster. We further define the *hierarchical uniform composability* condition on cell structures, which allows an elegant and efficient *buddy cell allocation* algorithm for cell management and allocation. The algorithm resembles buddy memory management [53], but with key differences due to different characteristics of GPU hierarchy vs. memory regions, as well as our focus on the sharing safety condition.

To ensure high utilization despite load fluctuations across tenants and over time, HiveD further introduces mechanisms to facilitate explicit *cell exchanges* among tenants, which gives one tenant the right to use cells in a different VC when its VC cannot satisfy its current needs. HiveD further introduces *low-priority jobs*, which are preemptible jobs to scavenge unused resources opportunistically. The buddy cell allocation algorithm is naturally extended to manage jobs with multiple levels of priorities. Combined, HiveD achieves the best of a private cluster (for guaranteed availability of cells independent of other tenants) and a shared cluster (for

improved utilization and access to more resources when other tenants are not using them). The combination of these mechanisms also offers a large design space to customize policies to cater to the needs of specific deployments.

We evaluate HiveD using experiments in a 96-GPU real cluster and trace-driven simulations. The results show that HiveD eliminates sharing anomalies identified in all the three state-of-the-art deep learning schedulers we evaluate, under various cluster loads. It decreases the excessive queueing delay from 1,300 minutes to zero! Moreover, the cell exchange mechanism increases the guaranteed resource capacity of a tenant by up to 58%.

In summary, this paper makes the following contributions:

- We define the notion of sharing safety inspired by the observed anomaly from a production system and propose a new construct, virtual private cluster, with hierarchical cell structures for sharing safety.
- We develop an elegant and efficient buddy cell allocation algorithm to manage cells for sharing safety and to support low-priority jobs for improved utilization.
- We enable cell exchange for tenants to gain more guaranteed resource capacity by seizing the opportunities from load fluctuations and diversity.
- We perform extensive evaluations both in a cluster and through simulation, driven by a production trace, to show that HiveD achieves the design goals in terms of sharing safety, queuing delay, and utilization, while revealing the intricate interplay of multiple factors for future research.

2 Motivation

In this section, we show the motivating characteristics of deep learning training workload both from prior work and from the observations on a trace extracted from a production cluster.

Affinity matters. A GPU cluster demonstrates multiple levels of affinity. GPU affinity within the same PCIe switch, the same CPU socket, or the same node could have significant impacts on the performance of deep learning training. Even within a single node, a multi-GPU deep learning training workload like VGG16 may suffer up to 40% performance loss if placed across PCIe switches, compared to that in the same switch [58, 60, 87].

Sharing anomaly and fragmentation. Currently, tenants share a GPU cluster in a way similar to sharing a CPU cluster [1, 50, 94]: each tenant is allocated a number of *tokens* as its quota, with each token corresponding to the right to use a GPU. A tenant can consume unused GPUs by running low-priority jobs, but these low-priority jobs may get preempted by the regular (high-priority) jobs.

However, this seemingly natural practice leads to sharing anomalies in multi-tenant GPU clusters. Figure 1 highlights one tenant's queuing anomaly of a shared GPU cluster through a trace-driven simulation, with an affinity-aware

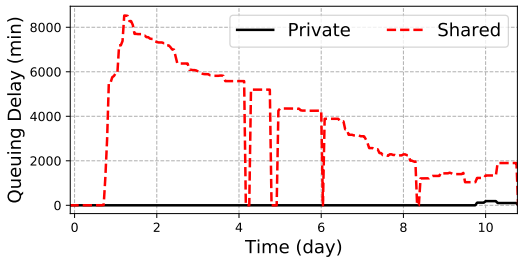


Figure 1. Sharing anomaly: a tenant suffers higher queuing delay in a shared cluster than in its own private cluster.

scheduler similar to the one described in [50]. During the 10 days, the tenant's average job queuing delay soars to 8,000 minutes in the shared cluster. While when running the jobs in a private cluster with the size of the tenant's quota, the queueing delay is approximately zero! This significantly impacts the sharing incentives: the tenant would rather run the (small) private cluster than share the GPUs in a larger cluster.

To understand the root cause, take the example shown in Figure 2, where tenants A and B share two GPU nodes, each with 4 GPUs evenly split in two PCIe switches. Both tenants get a quota of 4 GPUs. At time t_1, 4 1-GPU jobs a_1–a_4 from A run on GPUs 1 to 4. At t_2, a_2 and a_4 complete, and a 2-GPU job a_5 from A is scheduled to GPUs 5 and 6 for better affinity than GPUs 2 and 4. At t_3, a 4-GPU job b_1 from B cannot get the desired affinity in a single node even with 4 unused tokens due to allocations to A. In this case, tenant B would be better off having its own private cluster with a 4-GPU node. This example shows an implicit type of interference: *spreading of GPU fragmentation* across tenants, which often gets exacerbated by a large number of 1-GPU jobs. In our trace, we find more than 80% of jobs run on 1 GPU, with varying durations spanning from 1 minute to tens of hours.

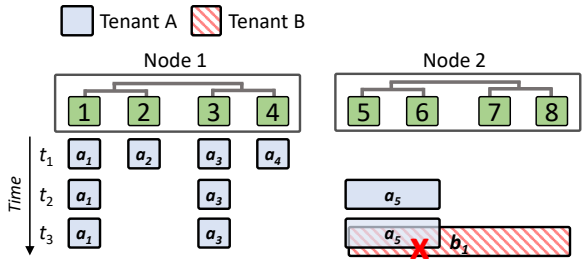


Figure 2. Example of inter-tenant fragmentation interference.

Load fluctuations. Figure 3 shows the fluctuating GPU demands of two tenants, extracted from our trace. On the 5th day, tenants A and B's GPU demands hit peak and valley, respectively. While on the 15th day, the two tenants again reach low and high demands but this time in an opposite way. A fixed quota cannot always meet the fluctuating demands. This motivates our design in § 3.3.

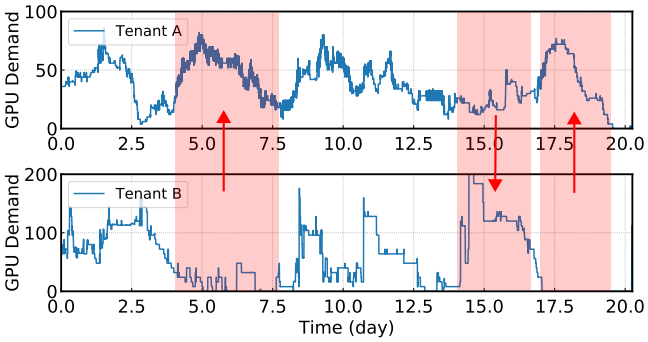


Figure 3. The variation of two tenants' GPU demand that exhibits interlaced peaks and valleys.

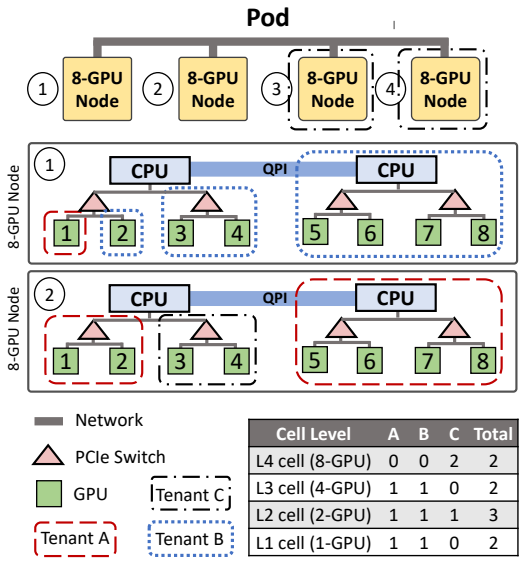
3 Virtual Private Cluster

With the observed sharing anomaly, a prerequisite to sharing a GPU cluster is to ensure *sharing safety*, in that a tenant's sequence of GPU requests (possibly with affinity constraints) is guaranteed to be satisfiable in a shared cluster if it can be satisfied on its private cluster.

To model a (private) GPU cluster, HiveD defines a *hierarchy of cell structures*, corresponding to different levels of affinitized GPU collections. A *cell* at a certain level is then the corresponding collection of affinitized GPUs with their interconnection topology. Each VC is then defined as numbers of cells at each level, modeled after the corresponding private cluster. Figure 4 shows an example, where there are 4 levels of cell structures: at the GPU, PCIe switch, CPU socket, and node levels, respectively. The pod consists of four 8-GPU nodes, shared by three tenants, *A*, *B*, and *C*. The cell assignment for each tenant's VC is summarized in the table in Figure 4. Tenants *A* and *B*'s VCs both reserve one level-3 cell (4 GPUs under the same CPU socket), one level-2 cell (2 GPUs under the same PCIe switch), and one level-1 cell (single GPU). Tenant *C* is a larger tenant, which reserves two level-4 cells (node level) and one level-2 cell.

In the cell hierarchy, a level-k cell c consists of a set S of level-$(k-1)$ cells. The cells in S are called *buddy cells* for each other; buddy cells can be merged into a cell at the next higher level. We assume cell demonstrates *hierarchical uniform composability*: (i) all level-k cells are equivalent in terms of satisfying a tenant request for a level-k cell, and (ii) how a level-k cell splits into level-$(k-1)$ cells is uniform. A heterogeneous cluster can be divided into multiple homogeneous ones satisfying hierarchical uniform composability.

A cluster provider must figure out the number of cells at each level to be assigned to each VC for all tenants. A VC assignment is *feasible* in a physical cluster if it can accommodate all cells assigned to all VCs; that is, there exists a one-to-one mapping from the "logical" cells in each VC to the "physical" cells in the physical cluster. A cluster might spare more physical resource than the assigned cells to handle hardware failure. HiveD advocates for a dynamic binding approach for flexibility, which could reduce job preemption and GPU fragmentation. Our evaluations (§ 5) confirm its benefits over a static binding. Each tenant then operates at two layers: it interacts with the cluster manager to allocate and deallocate cells dynamically within the constraints of its VC assignment. Within the boundary of its own VC, a tenant can adopt any scheduler of its choice to make scheduling decisions. Different tenants can even use different schedulers and impose different policies within their own VCs.



Cell Level	A	B	C	Total
L4 cell (8-GPU)	0	0	2	2
L3 cell (4-GPU)	1	1	0	2
L2 cell (2-GPU)	1	1	1	3
L1 cell (1-GPU)	1	1	0	2

Figure 4. Multi-level cell assignment for a pod: An example.

3.1 Buddy Cell Allocation Algorithm

HiveD must manage the dynamic binding between the logical cells in tenants' VCs and the physical cells in the physical cluster, and handle tenants' requests to allocate and release cells in their VCs. This is done by the *buddy cell allocation algorithm*. The buddy cell allocation algorithm maintains for each VC the information of the corresponding physical cell for each allocated logical cell, as well as a global free list at each cell level k to track all unallocated physical cells at that level. The algorithm always keeps available cells at the highest possible level: for example, if all the buddy cells at level-$(k-1)$ are available for a cell at level-k, only the cell at level-k is recorded. And the algorithm aims to keep as many higher-level cells available as possible. Algorithm 1 shows the pseudo-code of the algorithm.

To allocate[1] a level-k cell in a VC, the algorithm starts at level-k and goes up the level if needed: it first checks whether or not a free level-k cell is available and allocates a free level-k cell if one is available. If not, the algorithm will move up level-by-level, until a free level-l cell is available, where $l > k$. The algorithm will then split a free level-l cell recursively into multiple lower-level cells, until a level-k cell is available.

[1] We use the term cell mapping, binding, and allocation interchangeably.

Algorithm 1 Buddy Cell Allocation Algorithm

```
1:  // Initial state of free_cells: only top level has cells
2:  procedure AllocateCell(cell_level)
3:    if free_cells[cell_level].size() == 0 then
4:      c = AllocateCell(cell_level+1)
5:      cells = Split(c)                    ▷ Split cells are buddies
6:      free_cells[cell_level].extend(cells)
7:    Return free_cells[cell_level].pop()
8:
9:  procedure ReleaseCell(cell)
10:   if cell.buddies ⊆ free_cells[cell.level] then
11:     higher_cell = Merge(cell, cell.buddies)
12:     free_cells[cell.level].remove(cell.buddies)
13:     ReleaseCell(higher_cell)
14:   else
15:     free_cells[cell.level].add(cell)
```

Each split produces a set of buddy cells at the next lower level, which will be added to the free list at that lower level. One of those new low-level cells is again split until free level-k cells are produced.

The cell release process also works in a bottom-up manner. When a level-k cell c is released, the algorithm places c into a pool of free level-k cells and checks the status of c's buddy cells. If all of c's buddy cells are free, the algorithm will merge c and its buddy cells into a level-$(k+1)$ cell. The merge process continues recursively going up the levels, until no cells can be merged. In this way, the buddy cell allocation algorithm reduces GPU fragmentation and creates opportunities to schedule jobs that require higher-level cells.

Before processing an allocation request, the algorithm ensures the request is *legal* in that it is within the assigned quota for the VC at this cell level. HiveD stores the cell assignment in a table r, a tenant t's preassigned number for level-k cells is stored in $r_{t,k}$. The buddy cell allocation algorithm guarantees to satisfy all legal cell requests under a feasible initial VC assignment, which is formally stated in Theorem 1.

Theorem 1. *Buddy cell allocation algorithm satisfies any legal cell allocation, under the condition of hierarchical uniform composability, if the original VC assignment is feasible.*

Proof. Denote as $r_{t,k}$ the number of level-k cells reserved by tenant t, i.e., cell assignment for t. Denote as r_k the number of reserved level-k cells for all tenants, i.e. $r_k = \sum_t r_{t,k}$. Denote as $a_{t,k}$ the number of level-k cells that have already been allocated to t by the buddy cell allocation algorithm. Cell allocations that maintain $a_{t,k} \le r_{t,k}$ are legal. Denote as a_k the number of allocated level-k cells for all tenants (i.e., $a_k = \sum_t a_{t,k}$), and f_k the number of free level-k cells in the physical cluster, and h_k the number of level-$(k-1)$ buddy cells that a level-k cell can be split into (hierarchical uniform composability). Define F_k as the number of level-k cells that can be obtained by splitting the higher level cells while still satisfying the safety check for the cell assignment. F_k can be calculated by Eqn. (1).

$$F_k = \begin{cases} (f_{k+1} + F_{k+1} - (r_{k+1} - a_{k+1}))h_{k+1} & k < \hat{k}; \\ 0 & k = \hat{k}, \end{cases} \tag{1}$$

where $\hat{k}$ is the highest level.

To prove the theorem, we prove the following invariant:

$$r_k - a_k \le f_k + F_k \qquad \forall k = 1, 2, ..., \hat{k}. \tag{2}$$

The L.H.S. is the number of level-k cells all tenants yet to allocate, and the R.H.S. is the number of available level-k cells the cluster can provide.

We prove it by induction on discrete time slots. Denote as w the sequence number of time slots. A change of the cluster state will increase w by 1. When $w = 0$, $a_k = 0$, the invariant (2) holds as long as the original VC assignment is feasible. Assuming the invariant (2) holds at time $w = i$, we shall prove the invariant (2) still holds at time $w = i + 1$ after a tenant allocates a legal level-k cell.

Because the allocation is legal, $a_k < r_k$ should hold at time $i + 1$. In order to satisfy the invariant (2), either $f_k > 0$ or $f_k = 0$.

When $f_k > 0$, according to Algorithm 1, $a_k = a_k + 1$ and $f_k = f_k - 1$ after an allocation of level-k cell at time $i + 1$. The gap of both sides in the invariable (2) remains constant, thus it still holds.

When $f_k = 0$, i.e., no free cell at level-k, the algorithm will split a level-k' cell by finding the smallest k' where $k' > k$ and $f_{k'} > 0$. In this case, the invariant (2) remains true as in the $f_k > 0$ case, while the gap of the invariant (2) at level-k' will decrease by 1. If the invariant (2) at the level-k' breaks after cell splitting, this means $r_{k'} - a_{k'} = f_{k'} + F_{k'}$ at time $w = i$. By definition, F_k should be 0 at time $w = i$. But since $a_k < r_k$ (because the allocation request is legal), thus invariant (2) cannot hold true at level k. This leads to a contradiction. Therefore, the invariant (2) must hold at level k' after splitting a level-k' cell. Following the same step, we can prove the invariant (2) holds at level k'' when the algorithm recursively splitting a level-k'' cell, where $k'' \in [k+1, k'-1]$. Hence the invariant (2) holds on all levels when $f_k = 0$.

Merging the buddy cells can only either increase or keep the gap of the invariant (2) thus it still holds. Q.E.D. □

The buddy cell allocation algorithm has the time complexity of $O(\hat{k})$, where $\hat{k}$ is the number of levels, and can therefore scale to a large GPU cluster efficiently: $\hat{k}$ is usually 5, from the level of pods to the level of GPUs.

Hierarchical uniform composability ensures the algorithm's correctness and efficiency: it does not have to check explicitly after each split whether or not the subsequent legal allocation requests are satisfiable. Instead, it just needs to check whether every allocation request is legal. For the case where cells are heterogeneous (e.g., due to different GPU models or different inter-GPU connectivity), HiveD partitions the

cluster into several pools within which cells at the same level are homogeneous, and applies Algorithm 1 in each pool.

The algorithm resembles buddy memory allocation [53], hence the name. In buddy memory allocation, each memory block has an order n, with a size of 2^n. An order-n memory block is similar to a level-n cell in our case, with the same split and merge process, even though we generalize the memory block hierarchy slightly with the hierarchical uniform composability condition to model the natural GPU affinity hierarchy. But there are key differences: we have the concept of VCs with the guarantee that every legal request with respect to the VC is satisfiable. Buddy memory allocation does not have to worry about such correctness constraints. Also, our notion of buddies models the real GPU affinity hierarchy naturally, where the memory blocks in buddy memory allocation are mostly artificially created, limiting the possible choices.

We also find that it is straightforward to extend our algorithm to support low-priority jobs, whose allocated cells can be reclaimed by higher-priority jobs. Supporting such low-priority jobs helps improve overall GPU utilization, without compromising the guarantees provided by the VCs. In fact, our algorithm can easily support preemptible jobs at multiple priority levels. We elaborate next.

3.2 Allocating Low-Priority Cells

The cells assigned to each tenant's VC are the resources HiveD should guarantee with the highest priority. HiveD also allows tenants to submit low-priority jobs by consuming the free cells in the physical cluster. HiveD maintains two cell views, one for allocating the high-priority (guaranteed) cells, and the other for allocating the low-priority cells. Both cell views manage the same set of cells in the physical cluster using the same cell allocation algorithm (i.e., Algorithm 1). The high-priority cell requests are processed twice in both the high-priority view and the low-priority view. The low-priority cell requests are only processed in the low-priority view, but marks the corresponding cells in the high-priority view as "dirty" cells. A dirty cell in the high-priority view shows which GPUs are used by low-priority jobs. Note that a low-priority job will be preempted if the cell(s) it uses are allocated to high-priority jobs. Thus, when allocating cells for regular jobs, HiveD can choose the free high-priority cells with the least GPUs used by low-priority jobs for reducing unnecessary preemptions. HiveD adopts weighted max-min fairness [47] to decide the numbers of low-priority cells allocated to tenants. With a similar approach, we can easily extend HiveD to support multiple levels of priority.

3.3 Inter-Tenant Cell Exchange

In addition to low-priority cells, HiveD offers tenants an opportunity to obtain additional high-priority cells through *cell exchange* without worrying about being preempted.

In a shared cluster, a tenant's requirement to cells might fluctuate over time: pre-assigned cells might not always match the demand. For example, some research tenant might experience heavy load when being close to a conference deadline (e.g., NeurIPS) while the load is light otherwise. The tenant could exchange its pre-assigned cells with other tenants through *temporal exchange* for the right to use additional high-priority cells during the conference deadline.

Tenants also have incentives to exchange cells through *spatial exchange*. Sometimes a tenant needs to run many jobs to explore different neural architectures and hyperparameters that require many small cells. While sometimes a tenant needs to train a model with long wall-clock training time that requires fewer GPUs but a high-level cell with good GPU affinity [31], e.g., 8-GPU in the same node. When both types of tenants exist at the same time, the former type of tenant can offer a few high-level cells for exchanging more small cells with the later type of tenant.

Next, we discuss a general exchange mechanism to address the dynamic cell requirements across tenants. Based on the mechanism we further present an example cell exchange policy to implement the temporal exchange.

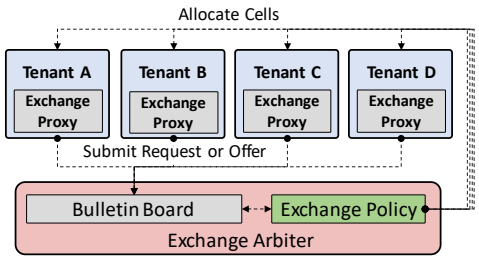


Figure 5. Architecture of the cell exchange mechanism.

A general exchange mechanism. Figure 5 illustrates HiveD's cell exchange mechanism. A tenant can post a request or an offer of cells through a per-tenant *Exchange Proxy*. A bulletin board in a centralized *Exchange Arbiter* collects requests/offers and exposes them to all tenants. A tenant could adopt some customized policy to match other's requests/offers through Exchange Proxy.

To motivate a tenant to make an offer, some kind of reward is desired. As a general cell exchange mechanism, HiveD relies on an *Exchange Policy* in the Exchange Arbiter to specify the exact form of reward. For example, a policy can reward the tenant who contributes the most, using a score-based credit scheme. It can also adopt an auction-based policy by asking tenants to bid the offers using a second-price auction [84]. Next, we present an example exchange policy based on the exchange mechanism. More design space of exchange policy will be discussed in § 4.

Example policy: balanced cell exchange. Algorithm 2 shows balanced cell exchange, an exchange policy used in HiveD.

It introduces a variable *surplus* to track the GPU hours that a tenant contributes (to other tenants from its VC) and consumes (on other tenants' cells). The surplus can be negative; in this case it becomes a deficit. When new offers or requests are posted, the policy matches the requests from the tenant who has the highest *surplus*, i.e., who contributes the most. For each request it retrieves the offers that could satisfy this request and chooses the one whose owner has the lowest *surplus*. This policy encourages tenants to contribute as many cells as possible to gain more high-priority cells.

Algorithm 2 Balanced Cell Exchange Algorithm

```
1:  // offers and requests are retrieved from Bulletin Board
2:  procedure DoMatching( )
3:    for tenant ∈ tenants do   ▷ Decreasing order of surplus
4:      for request ∈ requests[tenant] do
5:        qualified_offers = QualifiedOffers(request, offers)
6:        if qualified_offers.size() > 0 then
7:          offer = the offer whose tenant has lowest surplus
8:          offers.remove(offer)
9:          surplus[tenant] -= offer.gpu_hours
10:         surplus[offer.tenant] += offer.gpu_hours
11:         Return <request, offer>
12:   Return NULL
```

4 Discussion

VC assignment, pricing, tenant strategy, and cloud. We have focused on the mechanisms in HiveD, but there is a rich and interesting policy and strategy dimension from the perspectives of both the provider and the tenants. It is a fertile ground for future research and we will briefly discuss some of the opportunities here. As we have shown in § 5, the VC assignment to tenants, in terms of both the number of GPUs and their cell structures in a VC, has deep implications on the effective VC utilization, queuing delay, cell exchange opportunities, and fairness across tenants. Factors to consider in VC assignment include overall capacity, tenant demands, composition of tenant workload, workload variation over time, business priority, and budget constraints.

Pricing, together with a market-based mechanism, is one potential approach, as it has been studied in the context of resource management for the big-data workload. (Please see § 6 for a discussion of the related work on this subject.) With HiveD, a provider needs to consider the following dimensions in determining a pricing scheme: (i) the structure of a cell; e.g., an 8-GPU cell should ideally be priced higher than 8 1-GPU cells, (ii) guaranteed cells in a VC vs. low-priority cells, and (iii) time, or more precisely, the supply/demand at each particular time interval. Having a pricing scheme can also facilitate a wider range of cell exchange, even across different types of cells.

With a market-based mechanism, a tenant needs to develop its own strategy, to decide what type of VC to request (how many cells and what cell structures), based on its projected demand, its budget, and the pricing information for both VC cells and low-priority cells. A tenant's strategy should also explore opportunities for cell exchange: multiple tenants can do cell exchanges if their demands and supply match over time. Tenants can also do other types of trade based on their specific needs; for example, one tenant might exchange an 8-GPU cell for 16 single-GPU cells simply because it does not need the affinity for its workload. The variety of choices and their complex interplay make the problem particularly challenging, but also open up many new opportunities.

Major cloud providers are offering GPU VMs in the cloud. Our findings in HiveD are highly relevant even in the cloud setting and can shed light on the types of offering and pricing in the cloud. Our buddy cell allocation algorithm can also be used by the cloud providers to manage their reserved [2, 8, 19] and spot [6, 10, 77] GPU instances, as our VC cells are essentially reserved instances and our low-priority cells are essentially preemptible spot instances. It is easy to extend the algorithm to accommodate different GPU models, support pay-as-you-go on-demand instances [68], and handle expansion in capacity. For practical deployment, HiveD can use a hybrid strategy to leverage the cloud as an extension of the multi-tenant GPU cluster when the demand temporarily exceeds the capacity, or can be deployed entirely on a cloud using reserved resources at a lower price, with the options to (i) use spot instances, (ii) buy pay-as-you-go instances when needed, and (iii) purchase and sell reserved capacity in the marketplace [3].

Job migration. Migrating jobs between GPUs is a powerful mechanism that has been shown effective [87] in improving quality of GPU allocations dynamically. De-fragmentation via migration can in theory be used to resolve potential sharing safety violations, but our experience has shown that there are significant challenges in applying migration in production. First, fully transparent migration remains challenging in practice, due to implementation issues in different deep learning frameworks (e.g., inconsistent or limited use of certain programming APIs; challenges of multi-language, multi-framework, and multi-version support [14, 22, 62, 63]). Second, migration is not free. For example, a job often needs to warm up after migration; to hide the latency of job warm-up during migration, both the source and destination GPUs need to be used for a period of time. Finally, the choice of which jobs to migrate and where can be complex, with different conflicting objectives to balance and among a large search space. As shown in § 5.2.1, a greedy migration algorithm [87] can still violate sharing safety. In contrast, HiveD's buddy cell allocation algorithm is simple and effective. HiveD can also leverage migration, especially within each tenant—it will be a search space constrained to within a tenant to avoid any fairness issues, and under the sharing safety guarantee.

5 Evaluation

We evaluate HiveD using experiments in a real deployment on a public cloud and trace-driven simulations on a production workload. Overall, our key findings are:

- HiveD eliminates sharing safety violations observed in all of the tested schedulers. Excessive queuing delay in a shared cluster decreases from 1,300 minutes to zero!
- HiveD can incorporate the state-of-the-art schedulers by applying them inside VCs, and preserve sharing benefits with low-priority jobs, while ensuring sharing safety.
- HiveD's buddy cell allocation algorithm improves scheduling quality by reducing the number of preemptions by 55% and cluster fragmentation by up to 20%.
- HiveD's cell exchange mechanism helps seize further opportunities brought by load fluctuations and can increase tenants' guaranteed resource capacity by up to 58%.

5.1 Experimental Setup

Workloads. We collect a job trace that spans more than two months from a 2,232-GPU production cluster (279 8-GPU nodes). The trace contains 141,950 deep learning training jobs, each specifying its submission time, training time, number of required GPUs along with the affinity requirement, and the tenant who submitted it. The cluster is shared by 11 tenants. Table 1 shows the quota assignment of the tenants. Table 2 shows the distribution of GPU requirements of jobs in each tenant.

Cluster. To evaluate HiveD in a real environment, we conduct experiments in a GPU cluster deployed on a public cloud. The deployment consists of 24 virtual machines, each with 4 NVIDIA K80 GPUs (96 GPUs in total). We implement HiveD along with the schedulers on top of Kubernetes [21].

Tenant	Quota	Tenant	Quota
res-a	0.37%	prod-a	8.79%
res-b	0.73%	prod-b	10.62%
res-c	0.73%	prod-c	11.36%
res-d	1.47%	prod-d	15.75%
res-e	1.83%	prod-e	19.78%
res-f	28.57%		

Table 1. Quota assignment across tenants.

5.2 Sharing Safety

In this section, we examine the sharing safety of traditional quota-based scheme and HiveD, respectively, by experiments in a deployed cluster and large-scale simulations.

5.2.1 Cluster Experiment

Methodology. We collect a 10-day trace from the original 2-month production trace, and scale it down to our cluster size by random sampling. Due to privacy and security reasons, we cannot access the code and data of the jobs; therefore, we replace the jobs with 11 popular deep learning models

Tenant	1-GPU	2-GPU	4-GPU	8-GPU	≥16-GPU	Total
res-a	429	14	260	625	40	1,368
res-b	18,319	1,593	931	148	238	21,229
res-c	3,285	161	716	185	0	4,347
res-d	1,754	0	0	0	0	1,754
res-e	2,682	110	3,005	0	0	5,797
res-f	8,181	88	618	1,337	559	10,783
prod-a	227	54	23	1,132	138	1,574
prod-b	16,446	67	605	1,344	22	18,484
prod-c	4,692	301	1,905	4,415	1,206	12,519
prod-d	781	6	545	650	95	2,077
prod-e	58,407	532	2,118	959	2	62,018
Total	115,203	2,926	10,726	10,795	2,300	141,950

Table 2. Number of jobs with different GPU demands.

Type	Model	Dataset
NLP	Bi-Att-Flow [71]	SQuAD [66]
	Language Model [91]	PTB [61]
	GNMT [86]	WMT16 [13]
	Transformer [82]	WMT16
Speech	WaveNet [81]	VCTK [12]
	DeepSpeech [39]	CommonVoice [7]
CV	InceptionV3 [79]	ImageNet [25]
	ResNet-50 [41]	ImageNet
	AlexNet [54]	ImageNet
	VGG16 [74]	ImageNet
	VGG19	ImageNet

Table 3. Deep learning models used in the experiments [87].

in domains of Natural Language Processing (NLP), Speech, and Computer Vision (CV) from GitHub. We mix these models following a distribution of NLP:Speech:CV = 6:3:1, as reported in [87]. The models are summarized in Table 3.

In this experiment, we test three state-of-the-art scheduling policies for deep learning: YARN-CS [50], Gandiva [87], and Tiresias [35]. We obtained the source code of Gandiva and Tiresias and use the same implementation in our experiments. YARN-CS is a modified YARN Capacity Scheduler, which is optimized for GPU affinity by packing jobs, similar to that reported in [50]. We further refine the preemption policy of YARN-CS: instead of preempting the latest low-priority jobs, it preempts jobs based on the affinity requirement of the job to be scheduled. Otherwise, the baseline of YARN-CS will be much worse, due to excessive fragmentation. For each policy, we compare: (i) each tenant running its jobs in a private cluster with the capacity equals to its quota; (ii) tenants sharing the cluster using quota[2]; and (iii) tenants sharing the cluster using HiveD by applying the policies inside their VCs. In HiveD's experiments, we assign each tenant a set of node-level (8-GPU) cells with a total number of GPUs equal to its quota.

To model the cell hierarchy after the production cluster, during the scheduling we combine every two VMs into one

[2]We do not enforce quota in Tiresias as it does not support multi-tenancy.

8-GPU node. Similar to [87], to speed up replaying the 10-day trace, we "*fast-forward*" the experiment by instructing running jobs to skip a number of iterations whenever there are no scheduling events, including job arrival, completion, preemption, migration, etc. The time skipped is calculated by measuring job performance in a stable state.

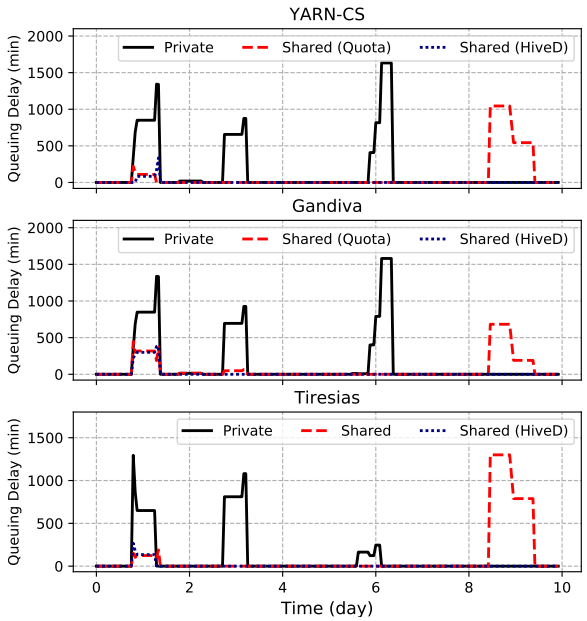


Figure 6. Queuing delay of prod-a in the cluster experiments.

Results. Figure 6 shows the queuing delay of tenant prod-a over time when using the three scheduling policies. The queuing delay is calculated with a moving average of jobs submitted in a 12-hour window.

Since sharing safety is not a design goal, we observe sharing anomalies in all the three scheduling policies when HiveD is not applied. For YARN-CS, from Day 8 to Day 10, the tenant suffers over 1,000 minutes higher queuing delay in a shared cluster than that in its private cluster. Although YARN-CS packs jobs as compactly as possible, a large number of 1-GPU jobs from other tenants with varying duration still make the available GPUs fragmented across nodes. As a result, the multi-GPU jobs have to wait for a long time for the desired affinity.

Similarly, in Gandiva, the tenant suffers up to 700 minutes higher queuing delay in the shared cluster. The queuing delay is lower than that in YARN-CS because Gandiva can mitigate cluster fragmentation by migrating jobs. However, due to the lack of the concept of cell, Gandiva's greedy algorithm may accidentally migrate jobs to make a tenant's GPU fragmentation worse, thus violating sharing safety.

In Tiresias, the tenant prod-a suffers the most. It experiences an excessive queuing delay up to 1,300 minutes. This is because the design goal of Tiresias is to minimize the overall job completion time. Thus it prefers short and small jobs over long-running and large ones. Without the sharing safety guaranteed by VC-based cell reservation, the multi-GPU jobs submitted by prod-a get fewer chances to run, when competing with the 1-GPU jobs from other tenants. The advantage of smaller jobs also exaggerates the fragmentation. Note that Tiresias does achieve the design goal, the overall job completion time is lower than other policies.

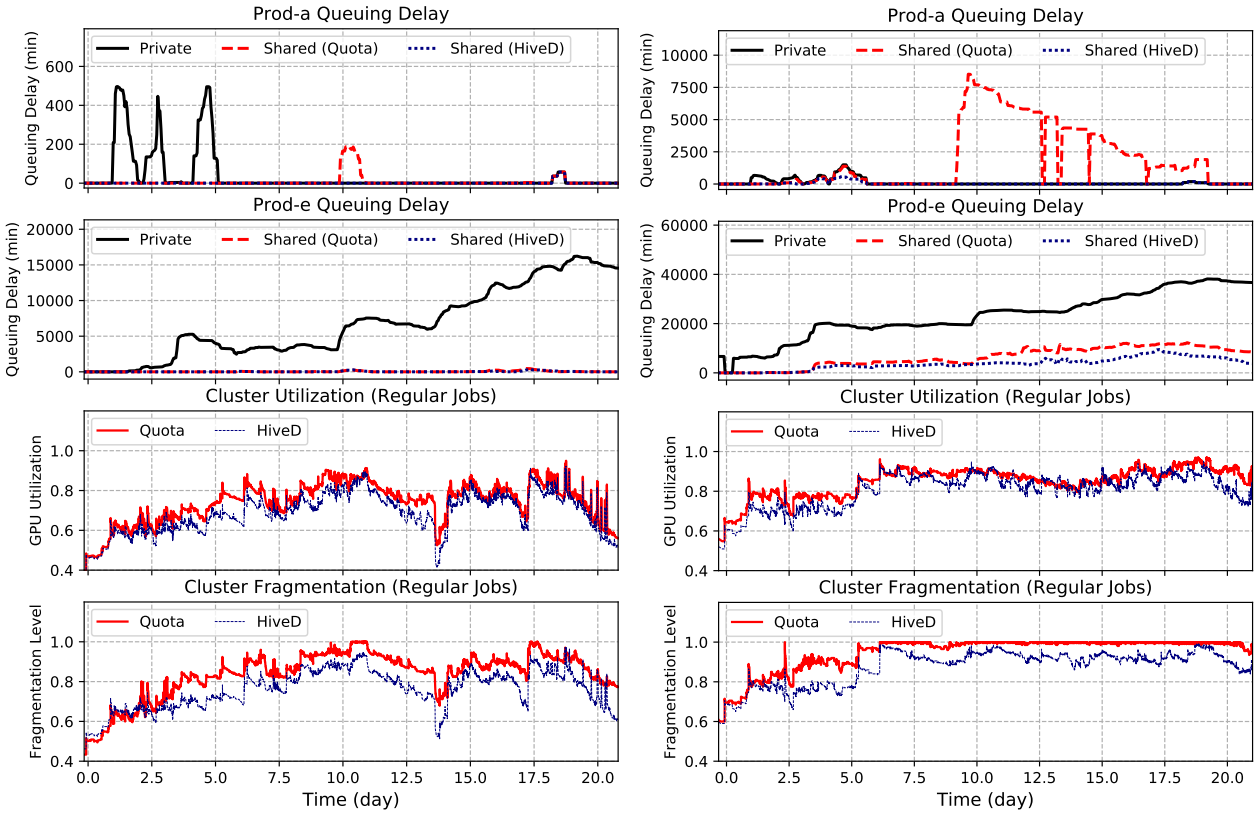
The experiments suggest that the evaluated schedulers are effective in their design objectives but they do not consider sharing safety, an important factor that impacts sharing incentive. HiveD complements them by ensuring sharing safety and at the same time preserving their effectiveness inside tenants' VCs. With HiveD, tenant prod-a (and all other tenants) never experiences an excessive queuing delay in the shared cluster, using each of the three policies. Even during Day 8∼10, the multi-GPU jobs are scheduled immediately as the tenant has enough 8-GPU cells in its VC (hence the reserved cells in the physical cluster). HiveD also provides the tenant with a significantly lower queuing delay in the shared cluster when it runs out of its own capacity in the private cluster (Day 1, 3, and 6), by giving it chances to run low-priority jobs.

In the experiments, we assume that each job always prefers the desired level of affinity. We also conduct experiments to evaluate the situation when a job can tolerate a relaxed level of affinity. In this case, some jobs experience excessive performance loss in training speed in a shared cluster, compared to those in the private cluster. Overall, we observe a similar trend with and without HiveD. Hence we omit the detailed report here.

5.2.2 Full Trace Simulation

We further use simulations on the 2-month production trace to reveal the factors that influence queuing delay and sharing safety. In the simulations (in this and the following sections), we use YARN-CS as the scheduling policy. To validate our simulator, we first replayed the cluster experiments in our simulator. We compare the average queuing delay of jobs in the simulation and that in the real experiment. The biggest difference among all the experiments was 7%. The difference is mainly due to factors such as performance variations of cloud VMs and job interference, which are not captured by our simulator. In the simulations we also observe similar sharing anomalies as shown in the real experiments, so we believe these variations do not affect our main conclusion.

Queuing delay with the original cluster size. Figure 7(a) shows the queuing delays of two representative tenants, prod-a and prod-e, during Day 1∼20 with the original size of our production cluster (279 nodes). We also show the GPU utilization and cluster fragmentation (of regular jobs, since the low-priority jobs do not affect scheduling decisions) over time to observe the correlation between queuing delay and cluster status. Specifically, at any time, the GPU utilization is defined as the proportion of GPUs occupied by regular jobs in the



(a) 279 nodes

(b) 200 nodes

Figure 7. Average queuing delay of prod-a and prod-e in a 12-hour window versus cluster utilization and fragmentation level.

cluster. The fragmentation level is defined as the proportion of nodes that are not available for an 8-GPU regular job.

HiveD achieves the lowest queuing delay in both tenants throughout the time span. For prod-a, it suffers a higher queuing delay in its private cluster in several time slots (e.g., the first 5 days), as the resource demands exceed its capacity. Since the shared cluster has a relatively low utilization from the regular jobs (at around 70%), both the quota-based scheduler and HiveD's scheduler reduce the queuing delay significantly. However, from Day 11 to Day 12, the tenant experiences a higher queuing delay of 200 minutes in the quota-based cluster than that in the private cluster. Note that prod-a has enough GPU quota in this period (since there is no queuing delay in its private cluster during the same period). However, the cluster-wide fragmentation level reaches 100%, which means the quota-based scheme cannot find even one node for scheduling prod-a's 8-GPU jobs. In comparison, with HiveD prod-a gains immediate access to the 8-GPU nodes with enough 8-GPU cells in its VC. Overall, the cluster fragmentation level in HiveD is lower than that in the quota-based scheme, because HiveD reserves cells for each tenant, preventing the spreading of fragmentation across tenants.

Queuing delay in a higher-load cluster. When a shared cluster is under-utilized, the sharing anomalies are less likely to happen due to sufficient GPUs on all affinity levels. To further understand the impact of cluster load on sharing safety, we run a simulation with the cluster size reduced to 200 8-GPU nodes (1,600 GPUs), where the GPU utilization reaches around 90%. The results are shown in Figure 7(b). In the quota-based scheme, prod-a experiences more sharing anomalies lasting for about 10 days (Day 9 to Day 19): its average queuing delay is thousands of minutes (over 8,000 minutes) higher than that in the private cluster. The reduced cluster size and higher load incur even higher fragmentation: the fragmentation level stays at 100% for most of the time, which delays the multi-GPU jobs and leads to head-of-line blocking of the following jobs.

Besides prod-a, we observe that res-f, the tenant owning the most quota in the cluster, also suffers from such sharing anomalies. Overall, 14% of res-f's multi-GPU jobs (365 in total) run with sufficient quota but experience excessive queuing delays. This number for prod-a is 23% (315 in total). While in HiveD there is no regular job suffering higher queuing delay.

For the other tenant (prod-e), the queuing delay is always lower in a shared cluster (for both quota and HiveD) than in the private cluster. This is because its workload is dominated by a large number of 1-GPU jobs (refer to Table 2),

which are immune to cluster fragmentation. HiveD can further lower prod-e's queuing delay by guaranteeing its multi-GPU affinities for its multi-GPU jobs.

In all the previous experiments, HiveD improves the cluster utilization (of both high- and low-priority jobs) over quota-based scheme by up to 20% in the 200-node case and 14% in the 279-node case, as a result of the reduced queueing delay.

5.3 Buddy Cell Allocation

In this section, we evaluate the buddy cell allocation algorithm through trace-driven simulations, to understand its effectiveness in reducing preemption and fragmentation, and its algorithm efficiency.

Reducing preemption with dynamic binding. In the buddy cell allocation algorithm, cells are bound to those in the physical cluster dynamically. This reduces unnecessary preemption of low-priority jobs when there are idle cells. Figure 8 shows the number of job preemptions when using buddy cell allocation's dynamic binding and static binding, respectively. This experiment uses the same setup of the 279-node experiment in § 5.2.2. In total, dynamic binding reduces the number of preempted GPUs by 55%, compared to that in static binding. We also measure the correlation between preemption (in a 12-hour window) and the proportion of bound cells in the time dimension. When there are more cells being bound to the physical cluster (e.g., Day 10, Day 20 in dynamic binding), there are also more GPUs being preempted. This is because we have fewer choices of physical cells to bind, hence fewer opportunities to reduce preemptions. This observation is also consistent with the fact that static binding, where this proportion is always 100%, incurs many more unnecessary preemptions.

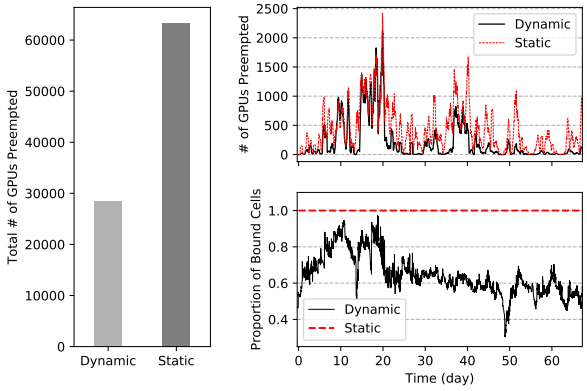


Figure 8. Preemption in dynamic and static bindings.

Reducing fragmentation with multi-level cells. The multiple levels of cells allow the buddy cell allocation algorithm to packs the cells at the same level across tenants to reduce fragmentation. For example, if two tenants both have a level-1 (1-GPU) cell, the algorithm prefers selecting two cells from the same physical node, i.e., buddy cells, to run a 1-GPU job. Instead, if both tenants only reserve level-4 cells, each of the two tenants has to use a level-4 (8-GPU) cell to run its 1-GPU job. Hence the two 1-GPU jobs will be placed on two different nodes, which increases fragmentation.

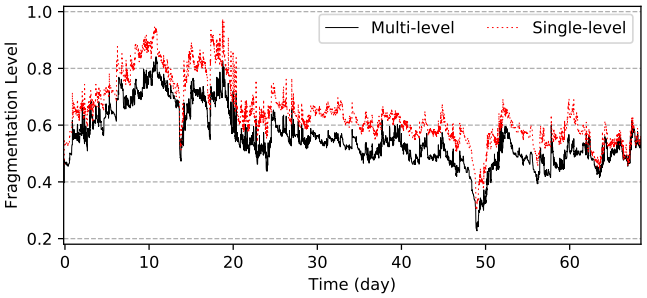


Figure 9. Fragmentation with multi- and single-level cells.

To demonstrate this, instead of assigning only level-4 cells, we assign cells from level-1 to level-4 while keeping the total number of GPUs assigned to each tenant the same as in the above 279-node simulation. This assignment for each tenant matches the distribution of its demands on each level of the cells. Figure 9 shows the fragmentation level over time when using multi-level and single-level (level-4) cells, respectively. The fragmentation level is always lower with multi-level cells. The gap is over 10% (up to 20%) for most of the time, which means we can spare roughly 30 more level-4 cells.

Despite the ability to reduce fragmentation, packing cells across tenants could potentially lead to inter-tenant job interference. A tenant could reserve higher-level cells if its jobs are sensitive to interference, so that the jobs will be less likely to be co-located with jobs from other tenants. Inside its VC the tenant can leverage scheduler like Gandiva to avoid interference among its own jobs.

Algorithm efficiency. We profile the performance of our implementation of buddy cell allocation in a setup of a 65536-GPU cluster that contains 8 pods, 256 nodes each with 8 GPUs. We issued 10,000 cell allocation requests at a random level. The average time to complete a schedule request is 2.18 milliseconds. The majority of the time cost comes from ordering cells according to low-priority jobs, which consists of 88% of the time. As the algorithm is clearly not the system bottleneck, we do not perform further optimization (e.g., lock-free operations).

5.4 Cell Exchange

Potential exchange opportunities. To investigate cell exchange opportunities in the production workload, we identify all the time slots in which an 8-GPU cell remains unused by regular jobs for a continuous interval (10 hours and 100 hours). Such time slots could be exchanged between the tenants.

Figure 10 shows the percentage of such slots out of all cells in the time dimension. Due to higher fragmentation, the quota-based shared cluster has much fewer 8-GPU cells than that in HiveD, which coincides with our observation in the

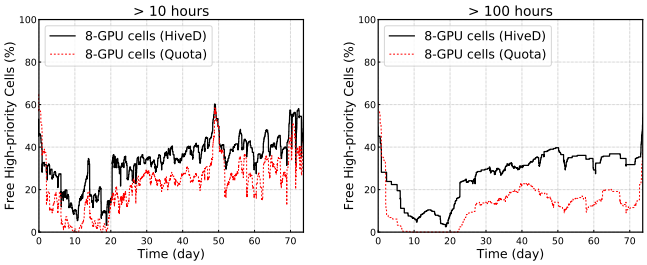


Figure 10. Potential cell exchange opportunities.

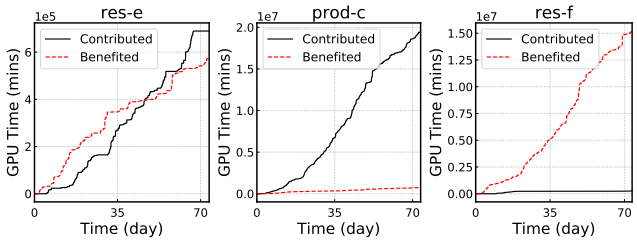


Figure 11. Contributed and benefited GPU time on the exchanged and low-priority cells in the default cell assignment.

previous experiments. Moreover, we observe that, in HiveD, about 22%-60% cells are not used by the owners for at least 10 hours after the 20-th day, which could be idle or used by low-priority jobs. Over 30% of the cells could even remain unused by the tenant for over 100 hours. HiveD brings more opportunities for reciprocal cell exchanges among tenants. Using HiveD's cell exchange mechanism, the tenants can trade these long-idle cells with other tenants for guaranteed cells in the future when they have demands exceeding their VCs' capacity.

Tenant	Effective Util. of High-Priority Cells	Increased Util. via Exchanges	Gain
res-e	52.3%	13.5%	25.8%
prod-a	25.3%	2.9%	11.5%
res-d	16.1%	1.5%	9.2%
prod-e	60.0%	5.3%	8.8%
res-c	74.9%	5.2%	7.0%
res-a	57.6%	4.0%	6.9%
prod-b	44.8%	2.5%	5.6%
prod-c	78.6%	2.9%	3.6%
res-b	74.8%	0.9%	1.2%
res-f	45.1%	0.4%	0.9%
prod-d	47.0%	0.4%	0.8%

Table 4. Increased high-priority cells via cell exchanges under the default cell assignment.

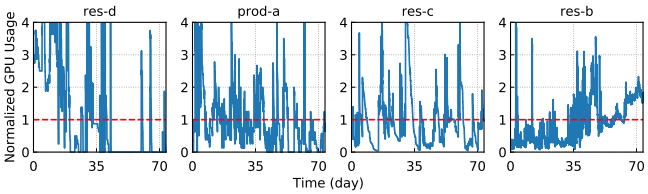
Cell exchange under the default cell assignment. We first evaluate HiveD's balanced cell exchange policy under the default cell assignment. Table 4 shows the effective utilization of each tenant's allocated cells, the increased utilization from cell exchanges, and the gain on the guaranteed utilization. The increased utilization only counts the cells that are exchanged successfully (i.e., when the cells a tenant contributed and those of others it benefit from are matched). Only two tenants achieve the improvement over 10%. The other tenants do not benefit much from cell exchange. The primary reason is the unbalanced cell assignment in the default setting, where some tenants' demands are always exceeding the assigned cells throughout the trace, while the other tenants' cells are heavily under-utilized. Figure 11 shows three tenants, and the variation of total GPU time each tenant contributed to and benefited from other tenants, via cell exchanges or low-priority jobs. Tenant res-e exhibits the most balanced contribution-benefit pattern, thus transforms the most exchange opportunities to the guaranteed cells (25.8% improvement in guaranteed utilization). Meanwhile, the other two tenants are either contributing or benefiting too much, and can hardly gain more guaranteed resources from exchange.

Cell exchange under on a balanced cell assignment. To experiment the cell exchange policy under a more balanced cell assignment, which should better reflect each tenant's demand, we reassign the cells according to the distribution of total GPU time requested by each tenant in the full trace. Table 5 shows each tenant's improvement on the guaranteed resource utilization under the balanced cell assignment. We observe significant improvement on all tenants on the increased guaranteed resource utilization via cell exchanges. More than half of the tenants achieve an improvement over 10% and up to 57.53%. This suggests that a more balanced cell assignment reflecting tenants' real demand can maximize the benefits of cell exchanges.

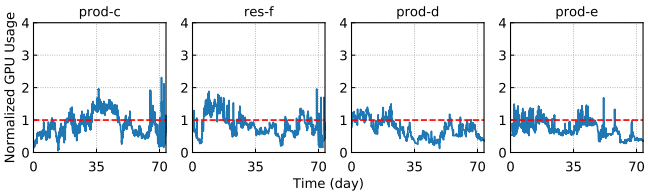
Tenant	Effective Util. of Guaranteed Cells	Increased Util. via Exchanges	Gain	Relative Fluctuation
res-d	24.80%	14.27%	57.53%	1.58
prod-a	48.39%	18.39%	38.00%	0.85
res-c	50.58%	13.17%	26.04%	0.82
res-b	55.03%	13.24%	24.06%	0.65
res-e	57.14%	9.77%	17.10%	0.7
res-a	60.49%	8.99%	14.87%	0.36
prod-b	62.53%	6.06%	9.69%	0.44
prod-c	68.32%	6.57%	9.61%	0.41
res-f	65.74%	6.07%	9.23%	0.38
prod-d	55.38%	4.10%	7.41%	0.41
prod-e	55.07%	2.64%	4.79%	0.38

Table 5. Increased high-priority cells via exchanges under the balanced cell assignment.

Intuitively, a tenant with a fluctuating resource usage pattern could benefit more from cell exchange, since it can contribute its cells during the usage valleys and take more guaranteed cells during the peaks. To quantify each tenant's usage fluctuation compared to their capacities, we define a metric called *Relative Fluctuation (RF)* by modifying Coefficient of Variation (CV) (shown in the last column of Table 5), which



(a) Top four tenants



(b) Last four tenants

Figure 12. GPU usage of the top and last four tenants on the improvement of guaranteed resource via cell exchanges (normalized to each tenant's number of GPUs in its VC).

is calculated as follows:

$$\text{Relative Fluctuation} = \sqrt{\frac{\sum_{i=1}^{N}(x_i - \overline{x})^2}{N}} \Big/ \overline{x}, \qquad (3)$$

where x_i is the i-th sample of the tenant's GPU usage over time and $\overline{x}$ is the number of GPUs in its VC (instead of the mean, which is used in CV). As expected, the tenant benefits the most (57.53%) exhibits the highest fluctuation (RF = 1.58).

We also show in Figure 12 the normalized GPU usage over time of the top and the last four tenants, in term of improvement from the exchanges. We observe a bursty resource usage in the top tenants, hence the higher improvement. While the last four tenants exhibit less fluctuation, with stable submissions of production training jobs. The experiment shows that, cell exchange can help HiveD handle fluctuating resource demands even under a static cell assignment by better matching resource peaks and valleys of different tenants over time.

6 Related Work

Affinity-aware schedulers for deep learning training. It has been recognized that a job scheduler should take affinity into account for deep learning workload [15, 35, 49, 50, 59, 64, 69, 87], as well as for other important (big-data) workload [27, 29, 90]. HiveD complements these schedulers by decoupling tenant-level fair sharing from job scheduling: existing schedulers can still be used in each virtual private cluster for improved scheduling quality, but our experiments show that they could lead to violations of sharing safety when used directly at the cluster level.

Cluster managers and fairness. There are numerous cluster managers and schedulers [4, 5, 11, 20, 32–34, 42, 43, 70, 78, 83, 92] that focus on different aspects such as fairness, cluster utilization, and scheduling efficiency. In multi-tenant shared clusters, fair sharing has always been an important research topic. In a CPU cluster, people extend max-min fairness [47] to address fair allocation of multiple resource types (DRF [28]), job scheduling with locality constraints [29, 30, 45, 90], and correlated and elastic demands (HUG [23]). These approaches all rely on quota, a scalar metric, to preserve the fair share of each tenant [52]. We have shown that simply relying on quota cannot achieve safe sharing, due to the spreading of fragmentation across tenants. Instead, we propose to leverage multi-level cells to achieve safe sharing. In this sense, cells can be viewed as the new units for fair sharing.

Performance isolation. Application performance in a shared cluster is sensitive to various sources of interference, including I/O, network, and cache. There are research works on performance isolation that include storage isolation [24, 36, 37, 80], appliance isolation [16, 73], network isolation [38, 56, 65, 72, 88], and GPU isolation [17, 18, 48, 51, 67, 89]. In HiveD, we identify a new source of interference: the spreading of fragmentation among tenants in a shared GPU cluster. To isolate such interference, HiveD adopts the notion of virtual private cluster that encapsulates the requirement in multi-level cells.

Market-based resource management. Some proposals advocate economics-based solutions to manage cluster resources based on real [26, 55, 76, 93] or virtual currency [44, 75, 85]. Stoica et al. [75] proposed to ask users to pay a virtual currency to run jobs; the more the user pays the better performance of the job. Stokely et al. [76] proposed to use a simulated clock auction for resource provisioning in heterogeneous clusters. Tycoon [55] is another a market-based resource allocation system to differentiate the economic value of jobs in the context of grid computing. We believe HiveD's cell exchange mechanism and low-priority jobs offer new dimensions for a market-based resource management scheme in a GPU cluster.

7 Conclusion

Motivated by observations from a real workload on a production cluster and validated through extensive evaluations, HiveD takes a new approach to the challenge of sharing a multi-tenant GPU cluster for deep learning by (i) defining a simple and practical guarantee, sharing safety, that is easily appreciated by tenants, (ii) developing an elegant and efficient algorithm, buddy cell allocation, that is naturally extended to handle multiple priorities, and (iii) devising a rich set of mechanisms such as cell exchange to offer flexibility to improve the system further. Combined, HiveD strikes the right balance between conflicting objectives such as low queuing delay, high utilization, and fairness, while offering a fertile ground for new research venues, such as market-based approaches.

References

[1] 2016. Hadoop: Fair Scheduler. https://hadoop.apache.org/docs/r2.7.2/hadoop-yarn/hadoop-yarn-site/FairScheduler.html.

[2] 2017. Announcing General Availability of Azure Reserved VM Instances. https://bit.ly/2jEFKHR.

[3] 2019. Amazon EC2 Reserved Instance Marketplace. https://aws.amazon.com/ec2/purchasing-options/reserved-instances/marketplace/.

[4] 2019. Apache Hadoop 2.9.2 – Hadoop: Fair Scheduler. https://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/FairScheduler.html.

[5] 2019. Apache Mesos – Quota. http://mesos.apache.org/documentation/latest/quota/.

[6] 2019. AWS Spot Instances. https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/using-spot-instances.html.

[7] 2019. Common Voice dataset. http://voice.mozilla.org/.

[8] 2019. Google Cloud: Committed Use Discounts. https://cloud.google.com/compute/docs/instances/signing-up-committed-use-discounts.

[9] 2019. Neural Network Intelligence (NNI). https://github.com/Microsoft/nni.

[10] 2019. Preemptible Virtual Machines. https://cloud.google.com/preemptible-vms/.

[11] 2019. Resource Quotas – Kubernetes. https://kubernetes.io/docs/concepts/policy/resource-quotas/.

[12] 2019. VCTK dataset. https://homepages.inf.ed.ac.uk/jyamagis/page3/page58/page58.html.

[13] 2019. WMT16 dataset. http://www.statmt.org/wmt16/.

[14] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. 2016. Tensorflow: A system for large-scale machine learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. 265–283.

[15] 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 (SC '17)*. ACM, New York, NY, USA, Article 17, 12 pages.

[16] Sebastian Angel, Hitesh Ballani, Thomas Karagiannis, Greg O'Shea, and Eno Thereska. 2014. End-to-end Performance Isolation Through Virtual Datacenters. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*. Broomfield, CO, 233–248.

[17] Rachata Ausavarungnirun, Joshua Landgraf, Vance Miller, Saugata Ghose, Jayneel Gandhi, Christopher J Rossbach, and Onur Mutlu. 2018. Mosaic: An Application-Transparent Hardware-Software Cooperative Memory Manager for GPUs. *arXiv preprint arXiv:1804.11265* (2018).

[18] Rachata Ausavarungnirun, Vance Miller, Joshua Landgraf, Saugata Ghose, Jayneel Gandhi, Adwait Jog, Christopher J Rossbach, and Onur Mutlu. 2018. Mask: Redesigning the gpu memory hierarchy to support multi-application concurrency. In *ACM SIGPLAN Notices*, Vol. 53. ACM, 503–518.

[19] Jeff Barr. 2009. Announcing Amazon EC2 Reserved Instances. https://aws.amazon.com/blogs/aws/announcing-ec2-reserved-instances/.

[20] Eric Boutin, Jaliya Ekanayake, Wei Lin, Bing Shi, Jingren Zhou, Zhengping Qian, Ming Wu, and Lidong Zhou. 2014. Apollo: scalable and coordinated scheduling for cloud-scale computing. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*. 285–300.

[21] Brendan Burns, Brian Grant, David Oppenheimer, Eric Brewer, and John Wilkes. 2016. Borg, Omega, and Kubernetes. *Commun. ACM* 59, 5 (2016), 50–57.

[22] Tianqi Chen, Mu Li, Yutian Li, Min Lin, Naiyan Wang, Minjie Wang, Tianjun Xiao, Bing Xu, Chiyuan Zhang, and Zheng Zhang. 2015. Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems. *arXiv preprint arXiv:1512.01274* (2015).

[23] Mosharaf Chowdhury, Zhenhua Liu, Ali Ghodsi, and Ion Stoica. 2016. HUG: Multi-Resource Fairness for Correlated and Elastic Demands. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*. Santa Clara, CA, 407–424.

[24] Asaf Cidon, Daniel Rushton, Stephen M Rumble, and Ryan Stutsman. 2017. Memshare: a dynamic multi-tenant key-value cache. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*. 321–334.

[25] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. 2009. Imagenet: A large-scale hierarchical image database. In *2009 IEEE conference on computer vision and pattern recognition*. Ieee, 248–255.

[26] Michal Feldman, Kevin Lai, and Li Zhang. 2005. A price-anticipating resource allocation mechanism for distributed shared clusters. In *Proceedings of the 6th ACM conference on Electronic commerce*. ACM, 127–136.

[27] Panagiotis Garefalakis, Konstantinos Karanasos, Peter Pietzuch, Arun Suresh, and Sriram Rao. 2018. Medea: Scheduling of Long Running Applications in Shared Production Clusters. In *Proceedings of the Thirteenth EuroSys Conference (EuroSys '18)*. ACM, New York, NY, USA, Article 4, 13 pages.

[28] Ali Ghodsi, Matei Zaharia, Benjamin Hindman, Andy Konwinski, Scott Shenker, and Ion Stoica. 2011. Dominant Resource Fairness: Fair Allocation of Multiple Resource Types.. In *Nsdi*, Vol. 11. 24–24.

[29] Ali Ghodsi, Matei Zaharia, Scott Shenker, and Ion Stoica. 2013. Choosy: Max-min fair sharing for datacenter jobs with constraints. *Proceedings of the 8th ACM European Conference on Computer Systems, EuroSys 2013*, 365–378.

[30] Ionel Gog, Malte Schwarzkopf, Adam Gleave, Robert NM Watson, and Steven Hand. 2016. Firmament: Fast, centralized cluster scheduling at scale. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. 99–115.

[31] Noah Golmant, Nikita Vemuri, Zhewei Yao, Vladimir Feinberg, Amir Gholami, Kai Rothauge, Michael W Mahoney, and Joseph Gonzalez. 2018. On the Computational Inefficiency of Large Batch Sizes for Stochastic Gradient Descent. *arXiv preprint arXiv:1811.12941* (2018).

[32] Robert Grandl, Ganesh Ananthanarayanan, Srikanth Kandula, Sriram Rao, and Aditya Akella. 2015. Multi-resource packing for cluster schedulers. *ACM SIGCOMM Computer Communication Review* 44, 4 (2015), 455–466.

[33] Robert Grandl, Mosharaf Chowdhury, Aditya Akella, and Ganesh Ananthanarayanan. 2016. Altruistic scheduling in multi-resource clusters. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. 65–80.

[34] Robert Grandl, Srikanth Kandula, Sriram Rao, Aditya Akella, and Janardhan Kulkarni. 2016. GRAPHENE: Packing and Dependency-Aware Scheduling for Data-Parallel Clusters. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. 81–97.

[35] 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)*. Boston, MA.

[36] Ajay Gulati, Irfan Ahmad, Carl A Waldspurger, et al. 2009. PARDA: Proportional Allocation of Resources for Distributed Storage Access.. In *FAST*, Vol. 9. 85–98.

[37] Ajay Gulati, Arif Merchant, and Peter J Varman. 2010. mClock: Handling Throughput Variability for Hypervisor IO Scheduling.. In *OSDI*, Vol. 10. 1–7.

[38] Chuanxiong Guo, Guohan Lu, Helen J Wang, Shuang Yang, Chao Kong, Peng Sun, Wenfei Wu, and Yongguang Zhang. 2010. Secondnet: a data center network virtualization architecture with bandwidth guarantees. In *Proceedings of the 6th International Conference*. ACM, 15.

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

[40] Kaiming He, Ross Girshick, and Piotr Dollár. 2018. Rethinking ImageNet Pre-training. *arXiv preprint arXiv:1811.08883* (2018).

[41] 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.

[42] Fabien Hermenier, Xavier Lorca, Jean-Marc Menaud, Gilles Muller, and Julia Lawall. 2009. Entropy: a consolidation manager for clusters. In *Proceedings of the 2009 ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*. ACM, 41–50.

[43] Botong Huang, Matthias Boehm, Yuanyuan Tian, Berthold Reinwald, Shirish Tatikonda, and Frederick R Reiss. 2015. Resource elasticity for large-scale machine learning. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. ACM, 137–152.

[44] David Irwin, Jeff Chase, Laura Grit, and Aydan Yumerefendi. 2005. Self-recharging virtual currency. In *Proceedings of the 2005 ACM SIGCOMM workshop on Economics of peer-to-peer systems*. ACM, 93–98.

[45] Michael Isard, Vijayan Prabhakaran, Jon Currey, Udi Wieder, Kunal Talwar, and Andrew Goldberg. 2009. Quincy: fair scheduling for distributed computing clusters. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*. ACM, 261–276.

[46] Max Jaderberg, Valentin Dalibard, Simon Osindero, Wojciech M Czarnecki, Jeff Donahue, Ali Razavi, Oriol Vinyals, Tim Green, Iain Dunning, Karen Simonyan, et al. 2017. Population based training of neural networks. *arXiv preprint arXiv:1711.09846* (2017).

[47] Jeffrey Jaffe. 1981. Bottleneck flow control. *IEEE Transactions on Communications* 29, 7 (1981), 954–962.

[48] Paras Jain, Xiangxi Mo, Ajay Jain, Harikaran Subbaraj, Rehan Sohail Durrani, Alexey Tumanov, Joseph Gonzalez, and Ion Stoica. 2018. Dynamic Space-Time Scheduling for GPU Inference. *arXiv preprint arXiv:1901.00041* (2018).

[49] Anand Jayarajan, Jinliang Wei, Garth Gibson, Alexandra Fedorova, and Gennady Pekhimenko. 2019. Priority-based Parameter Propagation for Distributed DNN Training. In *SysML*.

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

[51] Angela H Jiang, Daniel L-K Wong, Christopher Canel, Lilia Tang, Ishan Misra, Michael Kaminsky, Michael A Kozuch, Padmanabhan Pillai, David G Andersen, and Gregory R Ganger. 2018. Mainstream: Dynamic stem-sharing for multi-tenant video processing. In *2018 USENIX Annual Technical Conference (USENIXATC 18)*. 29–42.

[52] J. Kay and P. Lauder. 1988. A Fair Share Scheduler. *Commun. ACM* 31, 1 (Jan. 1988), 44–55.

[53] Kenneth C. Knowlton. 1965. A Fast Storage Allocator. *Commun. ACM* 8, 10 (Oct. 1965), 623–624.

[54] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. 2012. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*. 1097–1105.

[55] Kevin Lai, Lars Rasmusson, Eytan Adar, Li Zhang, and Bernardo A Huberman. 2005. Tycoon: An implementation of a distributed, market-based resource allocation system. *Multiagent and Grid Systems* 1, 3 (2005), 169–182.

[56] Jeongkeun Lee, Yoshio Turner, Myungjin Lee, Lucian Popa, Sujata Banerjee, Joon-Myung Kang, and Puneet Sharma. 2014. Application-driven bandwidth guarantees in datacenters. In *ACM SIGCOMM computer communication review*, Vol. 44. ACM, 467–478.

[57] Lisha Li, Kevin Jamieson, Giulia DeSalvo, Afshin Rostamizadeh, and Ameet Talwalkar. 2016. Hyperband: A novel bandit-based approach to hyperparameter optimization. *arXiv preprint arXiv:1603.06560* (2016).

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

[59] Hyeontaek Lim, David G Andersen, and Michael Kaminsky. 2018. 3LC: Lightweight and Effective Traffic Compression for Distributed Machine Learning. *arXiv preprint arXiv:1802.07389* (2018).

[60] Liang Luo, Jacob Nelson, Luis Ceze, Amar Phanishayee, and Arvind Krishnamurthy. 2018. Parameter hub: a rack-scale parameter server for distributed deep neural network training. In *Proceedings of the ACM Symposium on Cloud Computing*. ACM, 41–54.

[61] Mitchell Marcus, Beatrice Santorini, and Mary Ann Marcinkiewicz. 1993. Building a large annotated corpus of English: The Penn Treebank. (1993).

[62] Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, Melih Elibol, Zongheng Yang, William Paul, Michael I Jordan, et al. 2018. Ray: A distributed framework for emerging AI applications. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. 561–577.

[63] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. 2017. Automatic differentiation in pytorch. (2017).

[64] Yanghua Peng, Yixin Bao, Yangrui Chen, Chuan Wu, and Chuanxiong Guo. 2018. Optimus: An Efficient Dynamic Resource Scheduler for Deep Learning Clusters. In *Proceedings of the Thirteenth EuroSys Conference (EuroSys '18)*. ACM, New York, NY, USA, Article 3, 14 pages.

[65] Lucian Popa, Gautam Kumar, Mosharaf Chowdhury, Arvind Krishnamurthy, Sylvia Ratnasamy, and Ion Stoica. 2012. FairCloud: sharing the network in cloud computing. *ACM SIGCOMM Computer Communication Review* 42, 4 (2012), 187–198.

[66] Pranav Rajpurkar, Jian Zhang, Konstantin Lopyrev, and Percy Liang. 2016. Squad: 100,000+ questions for machine comprehension of text. *arXiv preprint arXiv:1606.05250* (2016).

[67] Christopher J Rossbach, Jon Currey, Mark Silberstein, Baishakhi Ray, and Emmett Witchel. 2011. PTask: operating system abstractions to manage GPUs as compute devices. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*. ACM, 233–248.

[68] Margaret Rouse. 2015. Pay-as-you-go cloud computing. https://searchstorage.techtarget.com/definition/pay-as-you-go-cloud-computing-PAYG-cloud-computing.

[69] Amedeo Sapio, Marco Canini, Chen-Yu Ho, Jacob Nelson, Panos Kalnis, Changhoon Kim, Arvind Krishnamurthy, Masoud Moshref, Dan RK Ports, and Peter Richtárik. 2019. Scaling Distributed Machine Learning with In-Network Aggregation. *arXiv preprint arXiv:1903.06701* (2019).

[70] Malte Schwarzkopf, Andy Konwinski, Michael Abd-El-Malek, and John Wilkes. 2013. Omega: flexible, scalable schedulers for large compute clusters. (2013).

[71] Minjoon Seo, Aniruddha Kembhavi, Ali Farhadi, and Hannaneh Hajishirzi. 2016. Bidirectional attention flow for machine comprehension. *arXiv preprint arXiv:1611.01603* (2016).

[72] Alan Shieh, Srikanth Kandula, Albert G Greenberg, Changhoon Kim, and Bikas Saha. 2011. Sharing the Data Center Network.. In *NSDI*, Vol. 11. 23–23.

[73] David Shue, Michael J. Freedman, and Anees Shaikh. 2012. Performance Isolation and Fairness for Multi-tenant Cloud Storage. In

Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation (OSDI'12). Berkeley, CA, USA, 349–362.

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

[75] Ion Stoica, Hussein Abdel-Wahab, and Alex Pothen. 1995. A microeconomic scheduler for parallel computers. In *Workshop on Job Scheduling Strategies for Parallel Processing*. Springer, 200–218.

[76] Murray Stokely, Jim Winget, Ed Keyes, Carrie Grimes, and Benjamin Yolken. 2009. Using a market economy to provision compute resources across planet-wide clusters. In *2009 IEEE International Symposium on Parallel & Distributed Processing*. IEEE, 1–8.

[77] Lee Stott. 2017. Microsoft Azure Low-priority Virtual Machines – take advantage of surplus capacity in Azure.

[78] Peng Sun, Yonggang Wen, Nguyen Binh Duong Ta, and Shengen Yan. 2017. Towards distributed machine learning in shared clusters: A dynamically-partitioned approach. In *2017 IEEE International Conference on Smart Computing (SMARTCOMP)*. IEEE, 1–6.

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

[80] Eno Thereska, Hitesh Ballani, Greg O'Shea, Thomas Karagiannis, Antony Rowstron, Tom Talpey, Richard Black, and Timothy Zhu. 2013. IOFlow: a software-defined storage architecture. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. ACM, 182–196.

[81] Aäron Van Den Oord, Sander Dieleman, Heiga Zen, Karen Simonyan, Oriol Vinyals, Alex Graves, Nal Kalchbrenner, Andrew W Senior, and Koray Kavukcuoglu. 2016. WaveNet: A generative model for raw audio. *SSW* 125 (2016).

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

[83] Abhishek Verma, Luis Pedrosa, Madhukar Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. 2015. Large-scale cluster management at Google with Borg. In *Proceedings of the Tenth European Conference on Computer Systems*. ACM, 18.

[84] William Vickrey. 1961. Counterspeculation, auctions, and competitive sealed tenders. *The Journal of finance* 16, 1 (1961), 8–37.

[85] Carl A Waldspurger and William E Weihl. 1994. Lottery scheduling: Flexible proportional-share resource management. In *Proceedings of the 1st USENIX conference on Operating Systems Design and Implementation*. USENIX Association, 1.

[86] Yonghui Wu, Mike Schuster, Zhifeng Chen, Quoc V Le, Mohammad Norouzi, Wolfgang Macherey, Maxim Krikun, Yuan Cao, Qin Gao, Klaus Macherey, et al. 2016. Google's neural machine translation system: Bridging the gap between human and machine translation. *arXiv preprint arXiv:1609.08144* (2016).

[87] 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.

[88] Di Xie, Ning Ding, Y Charlie Hu, and Ramana Kompella. 2012. The only constant is change: Incorporating time-varying network reservations in data centers. *ACM SIGCOMM Computer Communication Review* 42, 4 (2012), 199–210.

[89] Hangchen Yu and Christopher J Rossbach. 2017. Full Virtualization for GPUs Reconsidered. In *Proceedings of the Annual Workshop on Duplicating, Deconstructing, and Debunking*.

[90] Matei Zaharia, Dhruba Borthakur, Joydeep Sen Sarma, Khaled Elmeleegy, Scott Shenker, and Ion Stoica. 2010. Delay scheduling: a simple technique for achieving locality and fairness in cluster scheduling. In *Proceedings of the 5th European conference on Computer systems*. ACM, 265–278.

[91] Wojciech Zaremba, Ilya Sutskever, and Oriol Vinyals. 2014. Recurrent neural network regularization. *arXiv preprint arXiv:1409.2329* (2014).

[92] Haoyu Zhang, Logan Stafman, Andrew Or, and Michael J Freedman. 2017. SLAQ: quality-driven scheduling for distributed machine learning. In *Proceedings of the 2017 Symposium on Cloud Computing*. ACM, 390–404.

[93] Liang Zheng, Carlee Joe-Wong, Chee Wei Tan, Mung Chiang, and Xinyu Wang. 2015. How to bid the cloud. In *ACM SIGCOMM Computer Communication Review*, Vol. 45. ACM, 71–84.

[94] Jingren Zhou, Nicolas Bruno, Ming-Chuan Wu, Per-Ake Larson, Ronnie Chaiken, and Darren Shakib. 2012. SCOPE: Parallel Databases Meet MapReduce. *The VLDB Journal* 21, 5 (Oct. 2012), 611–636.