Synergy: Resource Sensitive DNN Scheduling in Multi-Tenant Clusters

Jayashree Mohan[†*], Amar Phanishayee[⋆], Janardhan Kulkarni[⋆], Vijay Chidambaram[†]
[⋆]*Microsoft Research* [†]*University of Texas at Austin*

Abstract

Training Deep Neural Networks (DNNs) is a widely popular workload in both enterprises and cloud data centers. Existing schedulers for DNN training consider GPU as the dominant resource, and allocate other resources such as CPU and memory proportional to the number of GPUs requested by the job. Unfortunately, these schedulers do not consider the impact of a job's sensitivity to allocation of CPU, memory, and storage resources. In this work, we propose Synergy, a resource-sensitive scheduler for shared GPU clusters. Synergy infers the sensitivity of DNNs to different resources using optimistic profiling; some jobs might benefit from more than the GPU-proportional allocation and some jobs might not be affected by less than GPU-proportional allocation. Synergy performs such multi-resource workload-aware assignments across a set of jobs scheduled on shared multi-tenant clusters using a new near-optimal online algorithm. Our experiments show that workload-aware CPU and memory allocations can improve average JCT up to 3.4× when compared to traditional GPU-proportional scheduling.

1 Introduction

The widespread popularity of Deep Neural Networks (DNNs) makes training such models an important workload in both enterprises and cloud data centers. Training a DNN is resource-intensive and time-consuming. Enterprises typically setup large multi-tenant clusters, with expensive hardware accelerators like GPUs, to be shared by several users and production groups [24, 42]. In addition to the model-specific parameters and scripts, jobs specify their GPU demand before being scheduled to run on available servers. Jobs are scheduled and managed either using traditional big-data schedulers, such as Kubernetes [7] or YARN [38], or using modern schedulers that exploit DNN job characteristics for better performance and utilization [8,20,26,27,31,34,41]. These DNN schedulers decide how to allocate GPU resources to many jobs while implementing complex cluster-wide scheduling policies to optimize for objectives such as average job completion times (JCT), makespan, or user-level fairness.

[*]Work done as a MSR intern in Project Fiddle.

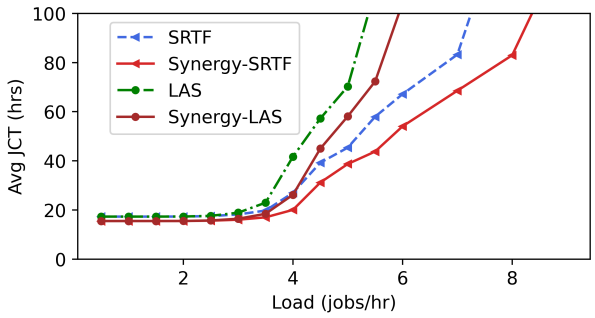


Figure 1: **Average JCT with Synergy**. Synergy is able to significantly reduce the average job completion time in a cluster for various scheduling policies

Current DNN cluster schedulers assume GPUs to be the dominant resource in the scheduling task [8, 20, 24, 26, 27, 31, 34, 41]; i.e., a user requests a fixed number of GPUs for her DNN job, and when the requested number of GPUs are all available, the job is scheduled to run. Other resources such as CPU and memory are allocated proportional to the number of GPUs assigned to the job (*GPU-proportional* allocation).

However, we identify an important property of DNN training jobs that GPU-proportional allocation is unable to exploit: DNNs exhibit varied sensitivity to the amount of CPU, memory, and storage bandwidth available to the job. For instance, some image and video recognition models achieve up to 3× speedup by overcoming data stalls (§2) when the CPUs allocated exceed their GPU-proportional share, while other models like GNMT are unaffected when the CPUs assigned are less than GPU-proportional share. Our main insight here is that allocating these auxiliary resources in a workload-aware fashion, rather than the traditional GPU-proportional allocation, can significantly improve performance by effectively utilizing cluster-wide resources. Figure 1 shows the average JCT in the cluster as we vary load for two scheduling policies with Synergy; Synergy's workload-aware allocation is able to significantly improve average JCT in the cluster.

Based on this insight, we propose Synergy, a workload and data-aware scheduler for multi-tenant, shared GPU clusters. Synergy profiles the sensitivity of DNNs to different auxiliary resources such as CPU, memory, and storage, and

allocates them disproportionately among jobs rather than using traditional GPU-proportional allocations. While doing so, Synergy ensures a job gets less than GPU-proportional auxiliary resources *only* if such an allocation does not degrade the job throughput compared to a GPU-proportional allocation. Our extensive experimentation on both physical and simulated GPU clusters show that resource-sensitive allocation and scheduling in Synergy can mitigate data stalls in several models, significantly increasing effective cluster throughput.

Efficiently exploiting the heterogeneity in resource sensitivity among DNN jobs raises two important problems which have not been tackled by prior work:

1. What is the ideal resource requirement for each job (with fixed GPU demand) and how can this be determined quickly?

2. How should we pack these jobs onto servers along multiple resource dimensions efficiently, especially when we can tune the job's demand for these resources?

Optimistic profiling. To identify the resource sensitivity of a job, Synergy exploits the predictability of DNN computation to measure the job throughput as we vary the amount of CPU and memory allocated to the job. This is performed offline by the Synergy scheduler, prior to job execution on the cluster. However, profiling all possible combinations of CPU, and memory values is computationally expensive. Therefore, Synergy introduces optimistic profiling; it only empirically profiles the job throughput for varying CPU allocations, assuming maximum memory allocation. It then *analytically* estimates the job throughput for all combinations of CPU and memory along with the respective storage bandwidth requirement for each allocation. A key insight that makes such analytical modelling feasible is the predictable nature of job performance to memory allocation when using DNN-aware caching like MinIO [29] that guarantees a certain cache hit rate. We show in §3.1 that our optimistically profiled model performance closely resembles the true empirical values, while significantly reducing profiling time (by up to 30×). From the profiles, Synergy identifies the ideal resource allocation as the least value of CPU and memory beyond which job throughput does not improve beyond a threshold (say 5%).

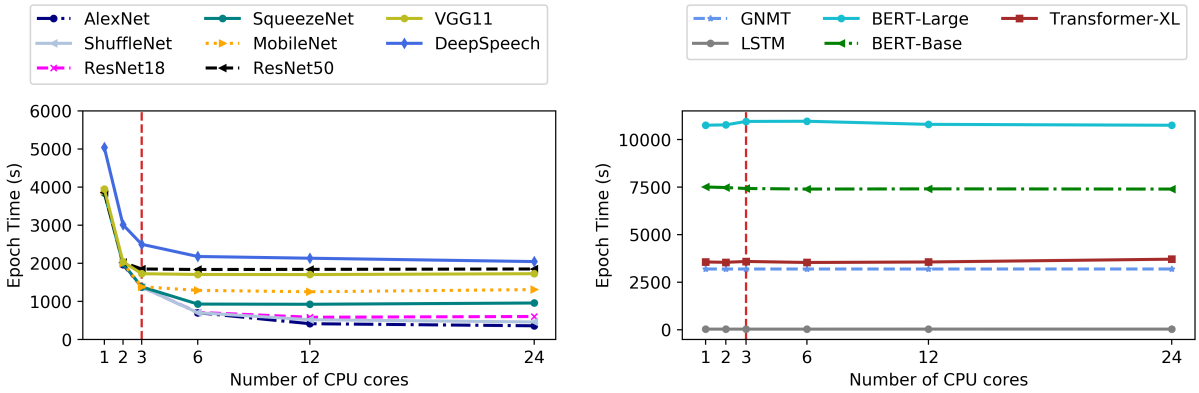
Scheduling mechanism. Synergy makes a round-based scheduling decision similar to prior DNN schedulers [31]. In each round (say 5 minutes), we identify the set of jobs that are runnable in the cluster using a given scheduling policy such as FIFO [38, 43], SRTF [9], LAS [20, 32], FTF [27], etc. This is the set of top n jobs as decided by the scheduling policy such that their collective GPU demand is less than the available GPUs in the cluster. Synergy's scheduling mechanism then packs these jobs among available servers in the cluster along all resource dimensions identified in the profiling phase. This is analogous to multi-dimensional bin-packing problem, which is NP-Hard [39], and hence requires approximate solutions. But unlike prior work in big-data scheduling which tackles the problem of multi-dimensional bin-packing (with fixed resource demands), *e.g.,* Tetris [17] and DRF [15], Synergy has to contend with the fungible resource demands. Algorithmically this introduces two challenges that need to be solved in tandem: First to find an optimal partition of CPU and memory among jobs to maximize throughput while ensuring fair allocations (every job's throughput is at least that of GPU-proportional allocation), and second, a feasible packing of these resources among jobs.

In this paper we propose two effective algorithms to enable such fungible multi-dimensional bin-packing. Our first algorithm, Synergy-OPT, is formulated as a linear program and enables determining an upper-bound on achievable throughput by an optimal solution for a given workload trace. However, we find that Synergy-OPT is impractical for two reasons: (1) it is computationally expensive as we scale cluster size (it has to run frequently within a single round duration), and (2) it produces fractional GPU, CPU, and memory allocations that cannot be achieved in real deployments. Nevertheless, its solution provides an aspirational optimal goal that we can use to measure the efficacy of any practical solution. The second algorithm, Synergy-TUNE, is fast and near-optimal (within 10% of Synergy-OPT in evaluation). If a job to be scheduled does not fit in the cluster along all the resource dimensions, we revert the job demands to GPU-proportional if its current ideal demands are above it. If the job's ideal demands are already GPU-proportional or below, then we find a suitable job in the cluster with higher than GPU-proportional allocation, which is then reverted to GPU-proportional. Synergy-TUNE also outperforms simpler greedy approaches (Synergy-GREEDY) that recursively pack jobs along multiple resource dimensions using a first-fit allocation strategy [14]; *i.e.,* place the job on a server that can fit it, else skip over the job.

We implement a prototype of Synergy and an accompanying event-driven simulator in Python. Synergy transparently communicates with the DNN job using a thin iterator API, that is a wrapper around the existing data iterators in these jobs, thereby requiring minimal code changes to the DNN job script. Across various scheduling policies, and workload traces, we show that the resource sensitive scheduling mechanisms used by Synergy can improve cluster objectives such as average job completion time (JCT) by up to 1.5× on a physical cluster of 32 GPUs. On a large simulated cluster, Synergy improves average JCT by up to 3.4×.

In summary, our paper makes the following contributions.

- We identify the importance of resource sensitivity of jobs in shared, multi-tenant GPU clusters, and the need for resource-aware scheduling (§2).
- We present Synergy, a workload-aware scheduler that optimistically profiles the DNN's sensitivity to resources and makes disproportionate allocations, ensuring no job achieves lower than GPU-proportional throughput (§3).



(a) DNNs sensitive to CPU allocation

(b) DNNs in-sensitive to CPU allocation

Figure 2: **CPU sensitivity**. This graph plots the epoch time for DNNs as we vary the number of CPU cores allocated for single-GPU training. Since the server has 8 GPUs and 24 CPUs in total, a GPU-proportional share is 3 CPUs. Some jobs such as Transformers need as few as 1 CPU to achieve maximum training speed; others like AlexNet need more than 12 CPUs per GPU.

- We present a heuristic scheduling mechanism Synergy-TUNE that maps the allocations calculated by the profiler onto the cluster, while better utilizing the resources compared to a GPU-proportional allocation (§4).
- In extensive experimentation on physical and simulated clusters, Synergy's techniques improve average JCT by up to 3.4×, allowing a higher input load (§5).

2 Background and Motivation

In this section, we briefly describe DNN scheduling and introduce the terminology used in the rest of the paper. We then motivate resource-sensitive DNN cluster scheduling using empirical analysis and examples.

Scheduling ML training jobs in a cluster. Training a ML model is a resource intensive and long-running task (order of hours to days). Collocating ML training workloads in a shared, multi-tenant cluster is a very common setup in several large organizations, for both research and production [20, 27, 31, 34, 41]. Our work targets state-of-the-art multi-tenant clusters similar to the ones published by prior large-scale studies by organizations like Microsoft [24] and Alibaba [42]. These clusters use on-premise servers or cloud VMs with pre-defined GPU, CPU, and memory resources. The cluster itself is shared by multiple users and jobs, and each server can host more than one job each with varying resource usage (some heavy on CPU side pre-processing, while others heavy on GPU computation). For example, a server with 8 GPUs can host 8 single GPU jobs from different users.

GPU-proportional allocation. In addition to the GPUs, each DNN job also requires CPUs to pre-process the dataset, and memory to cache the dataset during training. Existing DNN schedulers in literature [27, 31, 41], and those used in real-world GPU clusters [2, 24], allocate CPU and memory resources to a job using a GPU-proportional allocation. For instance, consider a server with 4 GPUs, 12 CPUs and 200 GB memory. The GPU-proportional allocation for 1 GPU-job is 3 CPUs and 50GB memory. If a job is allocated two GPUs on this server, then it is given 6 CPUs and 100GB memory.

Scheduling policy and mechanism When jobs are submitted to a scheduler, a scheduling policy such as First In, First Out (FIFO) [38, 43], Shortest Remaining Time First (SRTF) [9], Least Attained Service (LAS) [20, 32], or Finish Time Fairness (FTF) [27] decides the set of jobs (J) to be run on the cluster. A scheduling mechanism then identifies where job J should be run, and how much resources to allocate to the job.

2.1 Motivation : Resource sensitivity

Insight. The main insight that motivates our work is that DNNs co-scheduled on a cluster exhibit different levels of sensitivity to CPU and memory allocations during training. Therefore, it is possible to improve the overall cluster utilization and efficiency by performing resource-aware allocations instead of the state-of-the-art GPU-proportional allocation without any hardware changes. Prior work on characterization study of jobs in Microsoft's Philly cluster [24] shows that CPU cycles are under-utilized in multi-tenant clusters; we use this as motivation to show that we can *exploit the disparity in resource requirements* ***across jobs*** *to improve overall cluster utilization without hardware upgrades* (storage, CPU, or memory).

Figure 2 plots the per-epoch time for various DNNs when trained on 1 GPU (in isolation, one job at a time) by varying the number of CPUs allocated to the job (ensuring that the dataset is fully cached for each job). It is trained on a server whose GPU-proportional allocation per GPU is 3 CPUs and 62GB of memory.

Job	Model
J_1	ResNet18
J_2	Audio-M5
J_3	Transformer
J_4	GNMT

Table 1: Example jobs

Server	Job	GPU	CPU	Mem
S_1	$J1$	4	12	250
	$J2$	4	12	250
S_2	$J3$	4	12	250
	$J4$	4	12	250

Table 2: GPU-proportional allocation

Server	Job	GPU	CPU	Mem
S_1	$J1$	4	23	400
	$J3$	4	1	100
S_2	$J2$	4	12	450
	$J4$	4	12	50

Table 3: Resource-sensitive allocation

Figure 2a shows that most image and speech models are sensitive to CPU allocations; smaller models like ShuffleNet and ResNet18 require 9–24 CPU cores per GPU to pre-process data items. Increasing the CPU allocation from the GPU-proportional share 3 to 12 results in 3.1× faster training for AlexNet, and increasing it to 9 results in 2.3× faster training for ResNet18. On the other hand, most language models are insensitive to CPU allocations as shown in Figure 2b. This is because they have modest input data pre-processing requirements. Transformer models for example, unlike image classification models, do not perform several unique data augmentation operations for each data item in every epoch [29].

Next, to understand the importance of memory allocations, we train two models; an image classification model - ResNet18 on OpenImages [16] and a language model GNMT on WMT, with varying memory allocations on a server whose GPU-proportional share of memory per GPU is 62GB. We observe that GNMT is insensitive to memory allocation; even if only 20GB memory is allocated (which is the required process memory for training), the training throughput is unaffected. However, increasing the memory from 62GB to 500GB for ResNet18 speeds up training by almost 2×. This is because, language models like GNMT, and transformers are GPU compute bound. Therefore, fetching data items from storage if they are not available in memory does not affect training throughput. On the other hand, image and speech models benefit from larger DRAM caches. If a data item is not cached, the cost of fetching it from the storage device can introduce fetch stalls in training [29].

Takeaway. *When two jobs have to be scheduled on the same server, it is possible to co-locate a CPU-sensitive job with a CPU-insensitive one. This allows CPU allocation to be performed in a resource-sensitive manner rather than GPU-proportional allocation. Similarly, it is always beneficial to pack a memory-sensitive job with an insensitive one, allowing disproportionate resource-sensitive sharing of memory to improve the aggregate cluster throughput.*

Example We now show how resource-sensitivity aware scheduling can improve cluster efficiency using a simple example. Consider two servers each with 8 GPUs, 24 CPUs and 500GB DRAM. Let's say we have 4 jobs in the scheduling queue, each requesting 4 GPUs as shown in Table 1.

We consider two different schedules; (1) GPU-proportional allocation and (2) resource-sensitivity aware allocation. The results of these schedules are shown in Table 2 and Table 3.

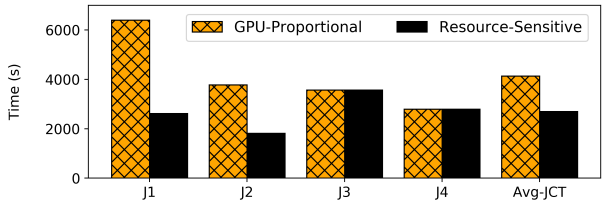


Figure 3: **Resource sensitive scheduling**. We compare the runtime of the jobs with two different schedules; GPU-proportional and resource-sensitive. By allocating resources disproportionately, CPU and memory sensitive jobs see increased throughputs which reduces the average JCT by 1.5×.

Figure 3 compares the epoch time of each of these jobs in the two scenarios. The increased resource allocation to CPU and memory sensitive jobs in Schedule 2 speeds up $J1$ and $J2$ significantly, while leaving the runtimes of $J3$ and $J4$ unaffected. The average job completion time in the cluster thus drops by 1.5× by performing resource-sensitivity aware allocations.

2.2 Synergy Scheduling Policies

Synergy is not constrained to one particular scheduling policy, but is instead general enough to improve a wide range of scheduling policies (*e.g.,* LAS, FIFO, SRTF, FTF, etc), creating Synergy-augmented variants for all of them. The main challenge that Synergy addresses is, finding an efficient partition of available cluster CPU and memory among jobs to maximize throughput while ensuring that every job's throughput is at least that of GPU-proportional allocation. Synergy's innovation thus lies in exploiting the differences in resource sensitivity across jobs in a cluster to improve overall cluster metrics.

3 Synergy: Design

Overview. Synergy is a round-based scheduler for GPU clusters that arbitrates multi-dimensional resources (GPU, CPU, and memory) in a homogeneous cluster among the set of runnable jobs in every round.Synergy augments existing scheduling policies with *resource sensitivity awareness* and it accomplishes this in two steps. First, it identifies the job's best-case CPU and memory requirements using an *optimistic profiling* technique (§3.1). Synergy then identifies a set of runnable jobs for the given round using an existing scheduling policy (*e.g.,* SRTF, FTF, FIFO, LAS, etc) such

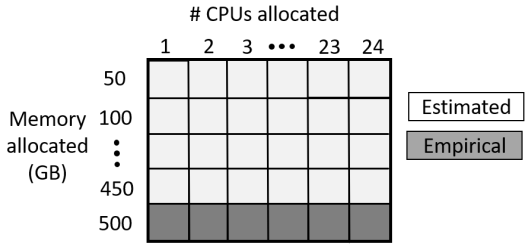
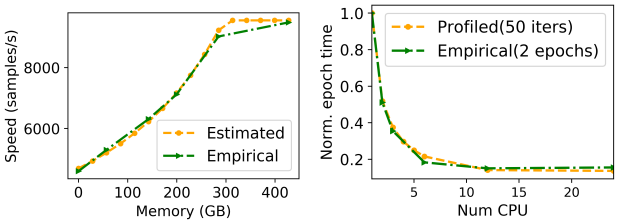


Figure 4: **Optimistic profiling** empirically evaluates the sensitivity of a model to varying # CPUs assuming a fully cached dataset; the rest of the matrix is completed using estimation



(a) Memory Validation (b) CPU validation

Figure 5: **Validating Synergy's optimistic profiling strategy**. The graphs compare the profiling results to empirical runs for CPU and memory demands in ResNet18

that their collective GPU demand is less than or equal to the GPUs available in the cluster. Then, using the profiled job resource requirements, Synergy packs these jobs on to the available servers along multiple resource dimensions using a near-optimal heuristic algorithm (§4). At the end of a round, the set of runnable jobs are updated using the scheduling policy, and their placement decisions are recomputed. We now discuss both the components of Synergy in detail.

3.1 Optimistic Profiling

A DNN job is profiled for its resource sensitivity once per lifetime of the job, i.e. on job arrival. Each incoming job is profiled by varying the CPU and memory allocated to the job. A *resource sensitivity matrix* is then constructed for discrete combinations of CPU and memory allocations as shown in Figure 4. Since DNN training has a highly predictable structure, empirically evaluating training throughput for a few iterations gives a fair estimate of the actual job throughput [29, 41].

It is easy to see that naively profiling different combinations of CPU and memory can be very expensive. For instance, if the cost of profiling one combination of CPU, and memory for a job is 1 minute, then to profile all discrete combinations of CPU and memory (assuming allocation in units of 50GB) on a server with 24 CPUs and 500GB DRAM takes about 24*10 = 240 minutes (4 hours)!

To tackle this problem, Synergy introduces an optimistic profiling technique that exploits the predictability in the relationship between job throughput and memory allocation. We observe that, with DNN-specific caches like MinIO [29], it is easy to model the job throughput behaviour as we vary the amount of memory allocated to a job at fixed CPU allocation. This is because, MinIO ensures that a job gets x% cache hits in every epoch if the memory allocated to the job holds x% of the dataset. For a given CPU allocation that determines the CPU pre-processing speed, and a known storage bandwidth, it is easy to analytically model the job throughput or varying memory allocation. Therefore, we only need to empirically profile the job for varying CPU values at full memory allocation, as indicated by the last row of the matrix in Figure 4. All the other entries can be estimated using the above technique. This leads to a 10× reduction in profiling time, bringing it down to 24 minutes! We experimentally validate this in Figure 5a. For a 8-GPU Resnet18 job, we compare the modeled job throughput using Synergy to the empirical results obtained by training the job for 2 epochs with varying memory allocations. As we see in Figure 5a, Synergy's estimations are within 3% of the empirical results, without having to actually run the model.

To further optimize profiling time, we observe that we do not require exact throughput values for a job with varying CPU allocations. We instead need a curve depicting the empirical job throughput. Therefore, instead of profiling the job for all possible CPU values, we pick discrete points for CPU profiling using the following algorithm. We start with the maximum CPU allocation and do a binary search on the CPU values to estimate job throughput. If the profiled point resulted in a throughput improvement that is less than a fixed threshold (say 10%), then we continue binary search on the lower half of CPU values, else we profile more points on the upper half. The idea here is to empirically profile CPU regions that show significant difference in job throughput, while skip those regions with little to no improvement in throughput. We experimentally show the efficacy of our CPU profiling technique in Fig 5b for a 1-GPU ResNet18 job. We compare the normalized job runtime (wrt 1 CPU) using empirical results averaged over 2 epochs of the job and Synergy's optimistic profiling averaged over 50 iterations (approximately, a minute per profile). Synergy is able to mimic the empirical job performance very closely, in under 8 minutes (using just 8 CPU profile points instead of 24). We believe that this is a reasonable overhead as it is incurred only once per lifetime of the job, which typically runs for hours.

After profiling a job on arrival, the job along with its resource sensitivity matrix is enqueued into the main scheduling queue, from which the scheduling policy picks a set of runnable jobs every round. When the job is scheduled and run on the cluster, its performance is monitored by a local agent in the cluster, which then updates the empirical performance

for a given CPU-memory allocation combination online, if it deviates from the value in the sensitivity matrix.

3.2 Scheduling mechanism

Synergy performs round-based scheduling. At the beginning of each scheduling round (a pre-defined time interval, say 15 minutes), Synergy identifies a set of runnable jobs from the scheduling queue that can be packed on the cluster in the current round duration. The set of runnable jobs is identified using a scheduling policy such as FIFO, SRTF, LAS, or FTF. Using the resource sensitivity matrix, Synergy packs these jobs onto the available servers in the cluster while satisfying the multi-dimensional resource constraints as opposed to simply performing a GPU-proportional allocation of CPU and memory resources.

Job demand vector. To pack the jobs onto servers, we first construct a job demand vector that indicates the GPU demand, and best-case CPU and memory requirements for the job. We identify the best-case values using the resource sensitivity matrix. We pick the minimum value of CPU and memory that saturates the job throughput.

Packing a job with multi-dimensional resource demands is analogous to multi-dimensional bin packing problem which is NP hard [39]. Therefore, we first evaluate the efficacy of a naive greedy scheduling mechanism as an approximation to tackle the multi-dimensional resource allocation problem.

3.3 Synergy-GREEDY: Greedy Scheduling

A naive greedy multi-resource packing algorithm translates to a first-fit approximation of the multi-dimensional bin packing problem [14]. Given a job demand vector populated as described in §3.2. The greedy algorithm picks the next runnable job decided by the scheduling policy, and places it on the server that can satisfy the jobs demands in all dimensions. If no such server exists, the job is *skipped* over for this round and the next available job is scheduled. This greedy approach has two main drawbacks

- This scheme can result in CPU and memory resources being exhausted by jobs, while leaving GPU resources underutilized. This results in GPU fragmentation in the cluster. We experimentally show that GPU fragmentation in Synergy-GREEDY severely degrades cluster objectives such as average job completion times (5.5). GPUs being the most expensive resource in a GPU-cluster, thus have to be utilized efficiently to run DNN jobs.
- This scheme also hurts the fairness of the scheduling policy as some jobs can be skipped over for a long time if their resource demands cannot be satisfied in the cluster.

Synergy-GREEDY thus introduces two major problems in the cluster - fragmentation and violation of scheduling fairness. The challenge ahead of us is to design a scheduling mechanism that eliminates GPU under-utilization due to fragmentation, and upholds the fairness properties of the given scheduling policy, while performing multi-dimensional resource allocation.

However, any greedy heuristic solution we come up with, is not optimal in our problem setting as greedy solutions do not make global repacking decisions when new jobs arrive; it is quite expensive to do so. Therefore, one pertinent question is to understand how good is the allocation produced by our heuristic when compared to an optimal solution

To this end, we first formulate a theoretical upper bound on the optimal throughput achieved by the cluster given a set of runnable jobs and their resource sensitivity profiles. We then discuss the challenges associated with materializing the optimal allocation on a physical cluster and introduce Synergy-TUNE, an empirically close-to-optimal heuristic solution.

4 Scheduling Algorithms

We first present our formulation of an optimal allocation that provides an upper bound on the achievable cluster throughput. This provides a basis for us to empirically evaluate how close to optimal the heuristic solution we design gets.

4.1 Synergy-OPT

Problem Definition Our goal is to allocate CPU and memory to each job so as to maximize the throughput, while guaranteeing that each job makes at least as much progress as it would do if we allocate its *GPU-proportional share*.

Notation

- s: The number of machines or servers.
- For each machine $i \in [s]$, we denote G_i, C_i, M_i as the total GPU, CPU, and memory available on machine i.
- We denote the total GPU available across all machines by G. That is, $G = \sum_i G_i$. Similarly, we denote C, M as the total CPU and Memory capacity across all machines.
- We denote jobs by indices j. The GPU requirement of job j is denoted by g_j.
- For each machine $i \in [s]$, we denote C_g, M_g as the GPU-proportional allocation of CPU and memory. That is, $C_g = C_i/G_i * g_j$ and $M_g = M_i/G_i * g_j$.
- J_t: The set of jobs that needs to be scheduled in each *round*. J_t is the set of runnable jobs for this round, identified by the scheduling policy such that the total GPU requirements of jobs in J_t is at most the total GPU capacity of the cluster. In notation, $\sum_{j \in J_t} g_j \leq G$.

- n: We denote the number of jobs in the set J_t by n. In notation, $n = |J_t|$.

- W_j: We assume that resource sensitivity matrix for each job j is given as input. $W_j[c,r]$ denote the amount of progress made on job j per round if c units of CPU and r units of (RAM) memory are allocated to job j.

- With a baseline GPU-proportional allocation strategy the progress a job makes in each round is equal to $W[Cg, Mg]$.

4.1.1 Throughput Upperbound in an Optimal Solution

It is not hard to show that our problem is NP-hard. So, we resort to finding approximate solutions. Towards that we first find an *upperbound* on the throughput achievable by an optimal solution. We achieve that by formulating our problem as a linear program (LP). Moreover, we assume an *idealized setting*: We assume that all the CPU and memory available across all the machines is present in one (super) machine. That is, there is only one machine with C units of CPU and M units of memory. Note that in reality C units of CPU and M units of memory are spread across s machines. This means that in our throughput allocation, we do not take into account the effect of network when resources are allocated to jobs across multiple machines. Therefore, the true optimal solution of our problem can only do worse than the idealized allocation.

4.1.2 An LP formulation

We get an upperbound on the optimal allocation via an LP formulation. The variables of our LP are denoted by $y_{\{c,m,j\}}$, which should be interpreted as follows. If for a job $j \in J_t$, $y_{\{c,m,j\}} = 1$, then it means that in the LP solution c units of CPU and m units of memory are allocated. We further note that for every job j, there is a variable $y_{\{c,m,j\}}$ for *for every possible* allocation of CPU and memory. We consider these variables in the discrete space as identified by our resource sensitivity matrix.

- Our objective function is to maximize the throughput. We formulate it as follows using our LP variables.

$$\text{Maximize} \quad \sum_{j \in J_t} \sum_{[c,m]} W_j[c,m] \cdot y_{\{c,m,j\}} \tag{1}$$

  Now, we enforce constraints such that LP solution is feasible in the idealized setting we talked about.

- First constraint we enforce is that the total CPU allocated to jobs is no more than the total capacity available:

$$\sum_{j \in J_t} \sum_{[c,m]} c \cdot y_{\{c,m,j\}} \leq C \tag{2}$$

- Similarly, we make sure that the total memory allocated to jobs is no more than the total capacity available:

$$\sum_{j \in J_t} \sum_{[c,m]} m \cdot y_{\{c,m,j\}} \leq M \tag{3}$$

- We want LP to allocate only one configuration of CPU and memory to each job.

$$\text{For each job } j \in J_t: \quad \sum_{[c,m]} y_{\{c,m,j\}} = 1 \tag{4}$$

- Finally, we want LP solution to be as good as the fair allocation.

$$\text{For each job } j \in J_t: \quad \sum_{[c,m]} W_j[c,m] \cdot y_{\{c,m,j\}} \geq W_j[C_g, M_g] \tag{5}$$

It is easy to verify that the optimal solution for our problem defines a feasible solution to our LP. On the other hand, as the LP solution can be fractional, in the sense $y_{\{c,m,j\}}$ variables can take fractional values, the optimum solution for LP can be no smaller than the true optimum solution, and thus always an upperboun on the throughput one can achieve for our problem. By enforcing the integrality constraints on $y_{\{c,m,j\}}$ variables one can getting a tighter upper bound. Indeed, in our experiments we solve this as a Integer Linear Program (ILP) where $y_{\{c,m,j\}}$ takes boolean values. For every job, we define the total CPU (c_j^*) and memory (m_j^*) allocated by the optimal ILP solution as follows.

$$\text{For each job } j, \text{ define} \quad c_j^* := c \quad \text{if} \quad y_{\{c,m,j\}==1}. \tag{6}$$

$$\textit{and} \quad m_j^* := m \quad \text{if} \quad y_{\{c,m,j\}==1}. \tag{7}$$

4.1.3 Feasible Allocation on Multiple Machines

Recall that in the LP(1-5), we assumed that all the resources are present on a single machine. However, in reality these resources are spread across multiple machines. So, now we need to make an allocation taking into account this fact. We achieve that by solving another linear program.

Now our goal is the following:

- For each job $j \in J_t$, allocate g_j units GPU, c_j^* units of CPU, and r_j^* units of memory across s machines such that each job is fully scheduled on a single machine. We call (g_j, c_j^*, r_j^*) as the demand vector of job j.

Again, the above problem is an instance of multi-dimensional bin packing problem, so it is NP-hard. Instead, we try to reduce the number of jobs that get fragmented. So, our new goal is:

- For each job $j \in J_t$, allocate g_j units GPU, c_j^* units of CPU, and r_j^* units of memory across s machines such that the number of jobs that get fragmented is minimized.

A Feasible Allocation via Second LP. The variables of the second LP are denoted by $x_{i,j}$. Here index i denotes the machine and j denotes the job. The variables $x_{i,j}$ are interpreted as follows: if $x_{i,j} = 1$, it means that resources of job j (that g_j units of GPU, c_j^* units of CPU, and r_j^* units of memory) are allocated on machine i.

Now we are ready to find a feasible allocation minimizing the number of fragmented jobs.

- First constraint we enforce is that the total GPU allocated to jobs is no more than the total capacity available on the machine:

$$\text{For each machine } i \text{ in } [s]: \quad \sum_{j \in J_t} g_j \cdot x_{i,j} \leq G_i \tag{8}$$

- Next, we make sure that the total CPU allocated is no more than the total capacity available on the machine:

$$\text{For each machine } i \text{ in } [s]: \quad \sum_{j \in J_t} \sum_{[c,m]} c_j^* \cdot x_{i,j} \leq C_i \tag{9}$$

- Similarly, we make sure that the total memory allocated is no more than the total capacity available on the machine:

$$\text{For each machine } j \text{ in } [s]: \quad \sum_{j \in J_t} \sum_{[c,m]} r_j^* \cdot x_{i,j} \leq M_i \tag{10}$$

- We make sure that every job is allocated all the resources it demands:

$$\text{For each job } j \in J_t \quad \sum_{i \in [s]} x_{i,j} \geq 1 \tag{11}$$

- Finally, We make sure that variables are positive.

$$\text{For each job } j \in J_t \text{ and } i \in [s] \quad x_{i,j} > 0 \tag{12}$$

Using linear programming theory, we now prove a structural property about our LP that states that most of the variables are integral.

Theorem 4.1. Suppose we assume that no job demands more CPU, GPU or memory available on a single machine. Then, the total number of jobs that get fragmented in the LP (8-12) is at most 3s.

Proof. Let $\{x_{i,j}^*\}_{i,j}$ denote an optimal solution to the LP(8-12). We know from linear programming theory that for every LP, there is an optimal solution which is a *vertex* of the polytope. Let P denote the set of positive variables in the LP solution. That is set of $x_{i,j}$ such that $x_{i,j} > 0$. A vertex solution is defined by a linearly independent family of tight constraints. A tight constraint means that in the LP solution a constraint is satisfied with an equality (=). A tight constraint of the form $x_{i,j} = 0$, only leads to variables not in P. Therefore, we only we need to consider tight constraints of the form (8), (9), (10), and (11). Therefore, number of variables taking positive values in P is bounded by

$$|P| \leq 3s + n \tag{13}$$

The above equation is true because there is only 1 constraint of the type (8), (9), and (10) for each machine and there are s machines. Further more there is one constraint of type (10) and there are n jobs.

Now let N_1 denote the number of jobs that got fragmented in the LP (8-12) solution. Now each such job contributes at least 2 variables to P. This implies,

$$2N_1 + (n - N_1) \leq |P| \leq 3s + n \tag{14}$$

Therefore, $N_1 \leq 3s$, and it concludes the proof. □

4.1.4 Challenges with operationalizing Synergy-OPT

While the allocations identified by Synergy-OPT provides an upper bound on the optimal cluster throughput, it is challenging to operationalize these allocations in the real world due to two main reasons;

1. Solving two LPs per scheduling round is a computationally expensive task. As cluster size and the number of jobs per round increases, the time to find an optimal allocation exponentially increases (§5.6)

2. The allocation matrix obtained with the second LP can result in fractional GPU allocations when jobs are split across servers; for instance, a valid allocation might assign 3.3 GPUs on server 1 and 2.7 GPUs on server 2 for a 6 GPU job. Realizing such an allocation requires a heuristic rounding off strategy to ensure non-fractional GPU allocations, as GPU time or space sharing, and its impact on job performance is considered beyond the scope of our work.

4.2 Synergy-TUNE

We now describe Synergy-TUNE, our heuristic scheduling mechanism.

Goal. Our goal is to design a scheduling mechanism that allocates multi-resource demand job onto available servers in the

cluster each round. In doing so, we want to ensure that (1) we do not affect the fairness properties of the scheduling policy used, (2) the expensive GPU resources are not fragmented and underutilized.

Allocation Requirements. Synergy-TUNE's allocation must satisfy the following requirements.

- The GPU, CPU, and memory resources requested by a single-GPU job must all be allocated on the same server.
- A multi-GPU job can either be consolidated on one server, or split across multiple servers. In the latter case, the CPU and memory allocations must be proportional across servers. For instance, if the job demands (2GPU, 12 CPU, 300GB DRAM), then while splitting it across two servers, we need to ensure that each server gets (1GPU, 6CPU, 150GB DRAM). This is because, multi-GPU jobs train on a separate process on each GPU, and synchronize at regular intervals, i.e., after one or many iterations. The job performance will vary across processes if each GPU does not get the same ratio of resources, and will eventually proceed at the speed of the process with the lowest allocation of CPU and memory.

In a shared cluster where users share the available resources across their jobs, it is import to enforce fairness in terms of throughput achieved by individual jobs. We need to ensure that no job runs at a throughput lower than what it would have achieved if we allocated a GPU-proportional share of CPU and memory resources. Additionally, we need to respect the priority order of jobs identified by the scheduling policy. For instance, a FIFO scheduling policy can be implemented using a priority queue sorted by job arrival times. Our mechanisms must schedule the top n jobs from this priority queue in every round that *fit* the cluster. As we mentioned, a DNN job can run as long as the requested number of GPUs are available; irrespective of the amount of memory and CPU available. Therefore, Synergy-TUNE identifies a set of runnable jobs as the top n jobs from the scheduling queue, whose GPU demands can be exactly satisfied by the available servers in the cluster. Synergy-TUNE picks this runable job set irrespective of the job's other resource demands - which are fungible. Note that, unlike Synergy-GREEDY, we do not skip over any jobs unless it cannot be scheduled (GPU demand cannot be met). Therefore, we never underutilize the GPUs when the cluster is at full load.

Next, Synergy-TUNE greedily packs each of these runnable jobs along multiple resource dimensions on one of the available servers, with the objective to minimize fragmentation. To achieve this, Synergy-TUNE sorts the runnable jobs by their GPU demands, followed by CPU, and memory demand. For each job j in order, Synergy-TUNE then picks the server with the least amount of free resources just enough to fit the demand vector of j. If it is a multi-GPU job, then we find a minimum set of servers with sufficient GPU availability that can fit the job's demands in entirety. However, it is possible that the job cannot fit in the cluster along all dimensions. In such a case,

1. We check if the job's demand vector is greater than proportional share of resources, In this case, we switch the job's demand to GPU-proportional share and retry allocation.

2. If the job still does not fit the cluster, or if the job's demand vector was less than or same as GPU proportional allocation in step (1), then to ensure fairness, we cannot further reduce the current job's demand. In this case, we do the following.

   (a) We repeat step (1) ignoring the job's CPU and memory requirements. We find a server that can just satisfy the job's GPU requirements. We know by construction that there is atleast one job on this server, which is allocated more than GPU-proportional share of resources. We identify the job or a set of jobs (J_s) on this server by switching whom to GPU-proportional share, we can release just as much resources required by the current job j. We switch the jobs in J_s to fair-share and by design, job j will fit this server.

   (b) We continue this recursively for all runnable jobs.

In the worst case, all the running jobs in a round could be allocated GPU-proportional share of resources. Therefore, Synergy ensures that its allocations never degrades a job throughput to less than GPU-proportional allocation.

In §5.6, we empirically compare Synergy-TUNE to Synergy-OPT showing that it is practical and near-optimal.

4.3 Implementation

We implement Synergy and an associated simulator in Python. Our scheduler is event-driven. There is a global event queue where job arrivals, schedule event, and deploy events are queued. These events are handled in the order of their arrival time. There is a priority job queue, where all the jobs arriving into the cluster are added. This queue is sorted by the priorty metric decided by the scheduling policy; for instance, SRTF sorts the jobs in the order of job remaining time.

When a schedule event occurs, the scheduler collects a list of runnable jobs from the job queue and identifies the appropriate placement for these jobs for the follwoing round, either using Synergy-GREEDY, Synergy-TUNE or Synergy-OPT. Then when a deploy event is triggered, these allocations are deployed on to the cluster. By default, every job requests for a lease update to continue running on the same server [31]. The scheduler then either grants a lease update or terminates the lease for the job, adding it back to the job queue.

The scheduler and the DNN jobs interact via a thin API provided by the Synergy data iterator. DNN job scripts must

Task	Model	Dataset
Image	Shufflenetv2 [44] AlexNet [25] Resnet18 [22] MobileNetv2 [36] ResNet50 [22]	ImageNet [35]
Language	GNMT [40] LSTM [] Transformer-XL [11]	WMT16 [6] Wikitext-2 [28] Wikitext-103 [28]
Speech	M5 [10] DeepSpeech [21]	Free Music [12] LibriSpeech [33]

Table 4: **Models used in this work**.

Policy (Metric)	Workload Split	Mechanism	Time (hrs) Deploy	Time (hrs) Simulate
FIFO (Makespan)	60-30-10	Proportional	16	15.67
		Tune	11.6	11.33
		Opt	-	11.01
SRTF (Avg JCT)	30-60-10	Proportional	4.81	4.52
		Tune	3.21	3.19
		Opt	-	3.06
SRTF (99 Percentile JCT)	30-60-10	Proportional	17.32	16.85
		Tune	8.59	8.54
		Opt	-	8.21

Table 5: **Physical cluster experiments**. This table sshows the comparison of makespan and average JCT, and 99th percentile JCT for two different traces; (1) a static trace with a workload composition of 60% image, 30% language and 10% speech models using FIFO scheduling (2) a dynamic trace with a workload composition of 30% image, 60% language and 10% speech models, using SRTF scheduling policy. Synergy-TUNE improves makespan by 1.4×, average JCT by 1.5 × and 99th percentile JCT by 2×.

be updated to call the Synergy iterator which is a wrapper around the default PyTorch [4] and DALI [3] iterators. The iterator handles registering the job with the scheduler, and appropriately sending lease updates. It also checkpoints the job to a shared storage if its lease is terminated. The iterator also synchronizes across GPU processes for a multi-GPU job to ensure that each process makes identical progress. We use `gRPC` [1] to communicate between the scheduler and the jobs.

We implement Synergy-OPT in `cvxpy` [13] for use in our simulator. The optimistic profiling module is also implemented in Python, and it profiles the incoming jobs hooked to the Synergy iterator, prior to the job's initial addition to the scheduling queue (a one time overhead for each job).

5 Evaluation

In this section, we use a number of microbenchmarks, trace-driven simulations from production cluster traces, and physical cluster deployment to evaluate the efficacy of Synergy's scheduling and profiling mechanism. Our evaluation seeks to answer the following questions.

- Does Synergy's data-aware scheduling mechanism improve objective metrics such as makespan and average JCT in a physical cluster (§5.2) and in trace-driven simulations of large-scale clusters (§5.3) ?
- How does Synergy-TUNE and Synergy-GREEDY perform with different workload splits and how well do they utilize available resources (§5.5)?
- How does Synergy compare to Synergy-OPT (§5.6) ?

5.1 Experimental setup

Clusters. Our experiments run on both a physical and a large simulated cluster. Our experiments are performed on state-of-the-art internal servers at Microsoft - these servers are part of larger multi-tenant research/production clusters. We run physical cluster experiments on a cluster with 32 V100 GPUs across 4 servers. Each server has 500GB DRAM, 24 CPU cores, and 8 GPUs. In all our experiments, fair-share CPU allocation is 3 cores per GPU and fair-share memory allocation is 62.5GB per GPU.

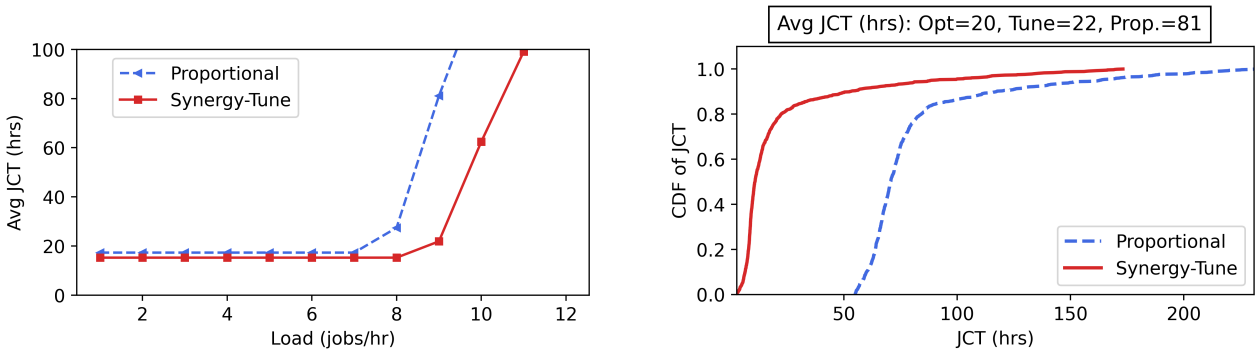
For simulated experiments, we assume two cluster sizes; a 128 GPU cluster across 16 servers and a 512 GPU cluster across 64 machines, where each machine resembles the physical server config mentioned above.

Models. Our experiments consider 10 different DNNs (CNNs, RNNs, and LSTMs) spanning across image classification (AlexNet, ResNet18, ShuffleNet, MobileNet, ResNet50), speech recognition (DeepSpeech), music classification (M5), language translation (GNMT), and language modeling (LSTM, Transformer-XL). We categorize these models by task (image, language, and speech) and assign a certain weight to these tasks in our traces. We call this a workload *split*. For instance, if the split for a given trace is (30,40,30), then the percentage of image, language, and speech models in the job trace is 30%, 40% and 30% respectively. All experiments are performed on PyTorch 1.1.0.

Traces. We run our physical and simulated experiments using both production traces from Microsoft Philly cluster [2] and traces derived using the Philly trace.

In the production trace, we use the job GPU demand, arrival time, and duration as is from the trace and fix an appropriate cluster size for simulation. We assign a model to each job from Table 4 based on a chosen workload *split*.

In the production-derived trace, we extract job GPU demand from the production trace and assign a model based on the chosen *split*. We then appropriately scale the job runtime and arrival for the chosen cluster size, while keeping the job duration distribution similar to the one in Philly trace. We achieve this as follows:



(a) FIFO (single)

(b) CDF of JCT at 9 jobs/hr

Figure 6: **Average JCT and CDF for FIFO**. Comparison of GPU-proportional allocation with Synergy.

- **Duration**. The duration of each job for GPU-proportional allocation is sampled from an exponential distribution: the job duration is set to 10^x minutes, where x is drawn uniformly from [1.5,3] with 80% probability, and from [3,4] with 20% probability similar to the trace duration used in prior work [31]. The iterations for this job is calculated using the assigned runtime and the per-iteration time obtained from the empirically profiled value.
- **Arrival**. We classify derived traces into two kinds based on the job arrival time : (1) a *static* trace where all the jobs arrive at the start of the workload, and (2) a *dynamic* trace, where the job arrival time is determined by a Poisson distribution at a rate λ.

Such a derived trace gives us the flexibility to vary the load on the cluster, the distribution of job duration and the composition of jobs in the workload. It thus helps evaluate our scheduling mechanism across a range of workload scenarios.

The experiments with the production Philly trace uses a 512 GPU cluster with a subrange of the trace containing 8000 jobs. The derived traces with varying job arrival rates uses a 128 GPU cluster. In both cases, we report the average metrics across a set of 1000 jobs in steady state.

For the physical cluster experiment, we choose a fixed arrival rate for the derived trace that keeps our cluster at *full load* (GPU demand of all runnable jobs > available GPUs in the cluster). For the simulated experiments, we vary the load λ on the cluster to evaluate its impact on cluster metrics. For the simulated experiments, we show results for two trace categories - (1) all jobs request single-GPU (2) multi-GPU jobs that request 1, 2, 4, 8, or 16 GPUs.

Policies and metrics. We evaluate Synergy against GPU-proportional for 4 different scheduling polices; FIFO, SRTF, LAS, and FTF. For a static trace, we measure makespan (time to complete all jobs submitted at the beginning of the trace) and for the dynamic job traces, we measure the average job completion time (JCT) of a subset of jobs in steady state (cluster at full load), and their CDF. This is same as the evaluation metrics used by prior related work [27, 31, 41].

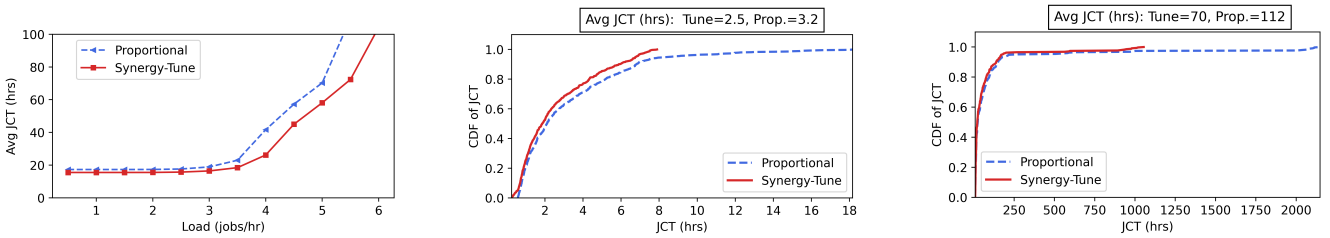
5.2 End-to-End Physical Cluster Experiments

For the physical cluster experiments, we run a Synergy-TUNE (*tune*) and GPU-proportional allocation (*proportional*) for two different workload traces. (1) A static production-derived trace of 100 jobs with a *split* (60,30,10), scheduled using FIFO and evaluated for makespan. (2) A dynamic production-derived trace with continuous job arrivals and a split of (30,60,10), scheduled using SRTF and evaluated for average and 99th percentile JCT. Both scenarios use an appropriately sized trace that keeps the cluster fully loaded. We compare the obtained results to that of the simulator by replaying the same trace. Additionally, we compare our metrics to the upper bound generated by the optimal solution, Synergy-OPT (*opt*). The results are shown in Table 5.

Synergy-TUNE reduces the makespan of static trace by 1.4× when compared to a data-agnostic GPU-proportional allocation mechanism. For the dynamic trace, Synergy-TUNE reduces average JCT of steady-state jobs by 1.5× while reducing the 99th percentile JCT of these jobs by 2× as shown in Table 5.

We compare the observed results from physical experiments to the same trace replayed on our simulator. As shown in Table 5, the difference between metrics in real and simulated clusters are less than 5%, demonstrating the fidelity of the simulator.

We also see from Table 5 that the cluster objectives achieved by Synergy-TUNE are within 4% of the optimal solution in this case. We do not deploy the optimal allocations due to the challenges enumerated in §4.1.4

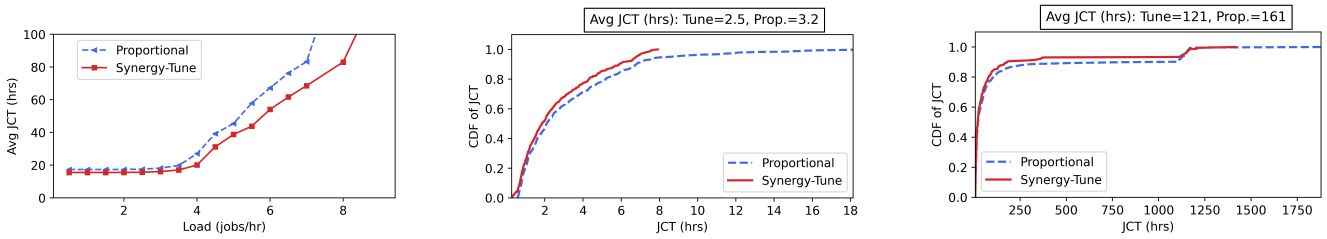


(a) LAS (multi)

(b) CDF of JCT at 4 jobs/hr (short)

(c) CDF of JCT at 4 jobs/hr (long)

Figure 7: **Average JCT and CDF for LAS policy**. This graph compares the baseline GPU-proportional allocation with Synergy-TUNE for LAS scheduling policy.



(a) SRTF (multi)

(b) CDF of JCT at 5.5 jobs/hr (short)

(c) CDF of JCT at 5.5 jobs/hr (long)

Figure 8: **Average JCT and CDF for SRTF**. This graph compares the baseline GPU-proportional allocation with Synergy-TUNE for SRTF scheduling policy.

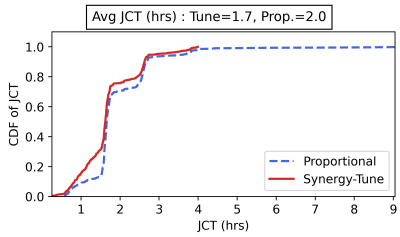
5.3 End-to-end results in simulation

We run simulated experiments on a cluster of 128 GPUs across 16 servers using production-derived traces. Each server is setup to mimic the real-world server used in our physical cluster experiments. We evaluate Synergy against GPU-proportional allocation mechanism for 4 different scheduling policies - FIFO, SRTF, LAS and FTF. We run dynamic workload traces, where jobs arrive continuously throughout the workload, at a rate *lambda*, as such a workload closely resembles real-world traces [2]. We show results for both single-GPU traces (where all jobs request 1 GPU) and multi-GPU traces (where jobs request multiple GPUs). Our metric of evaluation is the average Job Completion Time (JCT) of a set of 1000 jobs in cluster steady state.

Comparing Synergy-TUNE to GPU-proportional allocation. We plot the average JCT for a set of jobs in steady state cluster for varying cluster loads. We show the results for three scheduling policies, FIFO (single GPU trace) in Figure 6, LAS (multi-GPU trace) in Figure 7, and SRTF (multi-GPU trace) as shown in Figure 8. In all cases, we assume a workload split of (20,70,10). We plot both average JCT and the CDF of job completion times for a specific cluster load in all the scenarios described above. For the multi-GPU traces, we split the CDF into those for short and long jobs to distinctly differentiate the tail of the distribution. We make three key observations.

First, Synergy-TUNE improves average JCT by up to 3.4× in the single-GPU trace, and up to 1.6× in the multi-GPU trace by speeding up resource sensitive jobs with disproportionate allocation. The improvement in average JCT is higher as the load increases, because at low load the cluster is not at full capacity. As load increases, jobs start to get queued and incur queuing delay before being scheduled on the cluster. Since Synergy significantly speeds up individual jobs using disproportionate resource allocation, pending jobs can get scheduled faster, thereby reducing their queuing delays. Therefore Synergy improves cluster metrics by both reducing qeuing delays and speeding up individual jobs. Second, Synergy-TUNE is able to sustain a larger cluster load than GPU-proportional allocation. For multi-GPU scheduling with LAS, Synergy-TUNE reduced the 95th percentile JCT of long jobs by 2×. Third, the average JCT achieved with Synergy-TUNE is within 10% of the optimal solution in all cases.

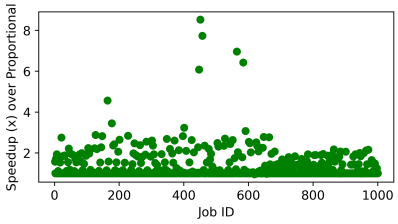
Similarly, for FTF scheduling policy, Synergy-TUNE observed 2.3× and 2× improvement in average JCT for a single-GPU and multi-GPU trace respectively.



(a) CDF of JCT for short jobs

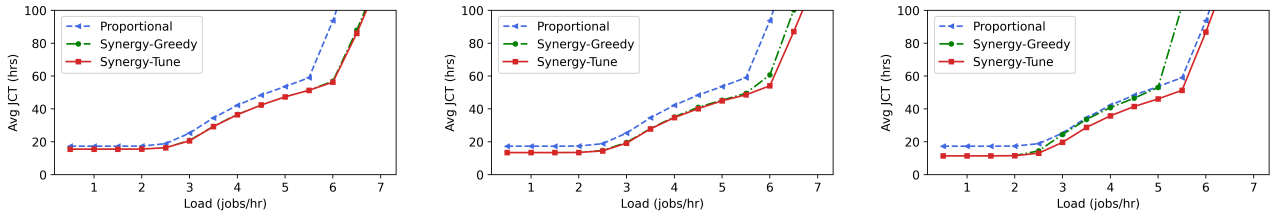
JCT (hrs)		Short	Long
Avg	Prop.	2	80
	Synergy	1.7	68
99p	Prop.	9	660
	Synergy	4	641

(b) Cluster metrics



(c) JCT speedup across jobs

Figure 9: **CDF for SRTF**. This graph compares the baseline GPU-proportional allocation with Synergy-TUNE for SRTF scheduling policy for long and short jobs using a subrange of real-world Philly trace. The JCT of individual jobs improves by upto 9× with Synergy.



(a) Split=(20,70,10) (b) Split=(33,33,33) (c) Split=(50,0,50)

Figure 10: **Varying workload split**. The graphs compares the Synergy-GREEDY with Synergy-TUNE and GPU-proportional allocation as we vary the workload split.

Policy	SRTF	LAS	FIFO
GPU-prop.	30	32	71
Synergy	26	28	62

Table 6: **Average JCT with Synergy**. This table shows the average JCT on a 512-GPU cluster using production trace with a (20,70,10) split across different scheduling policies

5.4 Simulation with production traces

We run simulated experiments on a cluster of 512 GPUs across 64 servers using a subrange of the Philly trace published by Microsoft [2]. We assume a workload split of (20,70,10) for this trail. Table 6 lists the average JCT with Synergy and GPU-proportional scheduling for three different scheduling policies. Across all policies, Synergy is able to reduce the average JCT compared to GPU-proportional scheduling due to better split of resources between jobs. We further plot the CDF of job completion time for one of the scheduling policies, SRTF as shown in Figure 9. We split the set of 1000 monitored jobs into short (JCT $<$ 4 hrs) and long jobs. Synergy reduces the tail of the distribution by 2.2× for short jobs and the average JCT of both long and short jobs by 15%. For each of the 1000 monitored jobs, we plot the individual job speedup with repect to GPU-proportional scheduling in Figure 9c. We see that Synergy speeds up jobs by upto 9× using better resource allocations.

5.5 Impact of workload split

The workload split decides the percentage of resource sensitive jobs in the workload. As the percentage of speech and image models increase in the trace, there may not be enough spare CPU and memory resources to perform disproportionate allocation, as they are mostly CPU- and memory-hungry. Figure 10 plots the average JCT with varying load for 3 different workload splits with FIFO scheduling for multi-GPU jobs. As the percentage of resource-sensitive jobs increase, we observe that Synergy-GREEDY breaks down, and ends up degrading JCTs significantly compared to a GPU-proportional allocation. This is because, the naive greedy technique results in resource fragmentation when the demand along CPU and memory dimensions are high, leaving several GPUs underutilized. Whereas, by the design of Synergy-TUNE, it allocates at least as many resources required to achieve the throughput of GPU-proportional allocation; therefore, even in the worst case workload split shown in Figure 10c, where all the jobs are CPU- and memory-sensitive, Synergy-TUNE performs as good as GPU-proportional allocation.

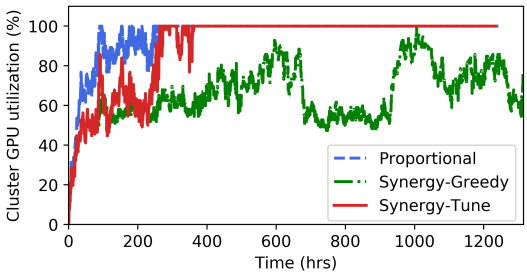


Figure 11: **Cluster GPU utilization**. This graph plots the GPU utilization in the cluster at high load. Synergy-TUNE matches GPU-proportional in terms of GPU utilization, while Synergy-GREEDY degrades cluster metrics due to GPU fragmentation and underutilization.

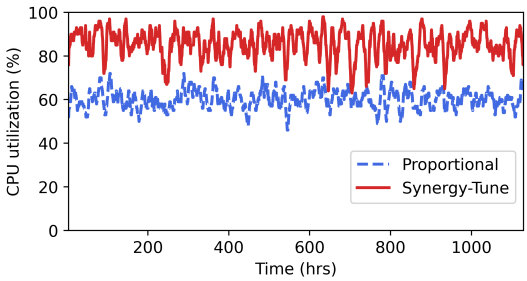


Figure 12: **CPU utilization**. This graph plots the cluster CPU utilization at low load. GPU-proportional scheduling underutilizes the available CPU resources while Synergy is able to efficiently utilize spare resources to improve cluster metrics.

Resource utilization. Figure 11 plots the GPU allocation over time for the workload in Figure 10c at a high load of 5.5 jobs/hr where the cluster GPU demand is higher than 100%. While Synergy-TUNE is able to sustain a higher load by finishing jobs faster at low cluster load, Synergy-GREEDY severely underutilizes GPU resources throughout the workload, trading it off for higher CPU and memory allocation. At low loads as shown in Figure 12, GPU-proportional allocation only utilized 60% of the available CPU resources, while Synergy-TUNE utilized it up to a 90%, resulting in up to 1.5× lower average JCT.

5.6 Comparison to Synergy-OPT

Calculating optimal allocations for every scheduling round with Synergy-OPT can be quite expensive, especially for large cluster sizes. We plot the time taken for per-round allocations for Synergy-OPT against that of Synergy-TUNE for varying cluster sizes in Figure 13. For Synergy-OPT, the time to solve the allocation problem quickly escalates, exponentially with increasing cluster sizes, while that for Synergy-TUNE is hardly a second. We also show experimentally that the allocations given by Synergy-TUNE are close to those estimated

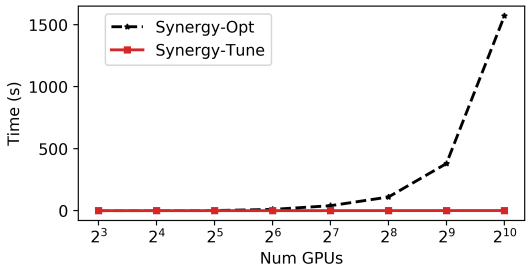


Figure 13: **Scalability of the scheduling mechanism**. This graph plots the scheduling overhead per round with Synergy-OPT and Synergy-TUNE. With Synergy-OPT the scheduling overhead increases exponentially with cluster size.

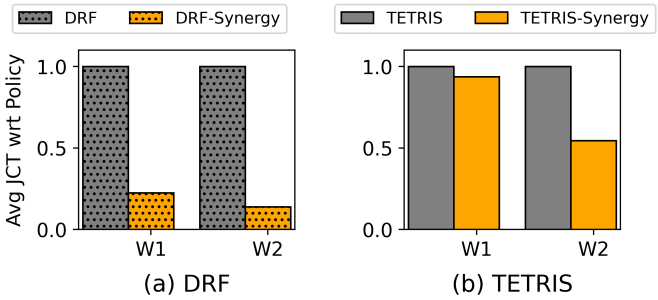


Figure 14: **Comparison to big data scheduling policies**. Synergy outperforms existing multi-resource allocation policies for big data workloads such as DRF and TETRIS evaluated here for two different workload compositions (lower JCT is better).

by Synergy-OPT in §5.2 and §5.3. At a cluster size of 128 GPUs used in our experiments, Synergy-TUNE converges at allocations that are within 10% of the optimal value, 200× faster than Synergy-OPT.

5.7 Comparison to DRF and Tetris

Big data schedulers like Dominant Resource Fairness (DRF) [15] and Tetris [17] have explored multi-dimensional resource allocation for map-reduce jobs. DNN jobs have different properties when compared to big-data jobs. DNN jobs are gang-scheduled, meaning they can run only when all the GPUs requested by them are available on the cluster at once. Further, the auxiliary resource requirements like CPU and memory are fungible unlike the GPU demand. DRF and Tetris assume resources to be statically allocated throughout the lifetime of a job, whereas Synergy assumes these resources to be fungible and could result in varied allocations throughout the lifetime of a DNN job. Furthermore, profiling the DNN job's resource demands is unique to Synergy; big data schedulers assume that the job request already encodes resource demands across all dimensions. To evaluate Synergy against these poli-

cies, we assume that the best-case resource requirement for CPU and memory is fed as input to the bigdata scheduling policies using Synergy's profiling mechanism.

On a cluster of 128 GPUs, we evaluate these policies on two different workload compositions : W1 (20,70,10), and W2 (50,0,50) and compare the naive policy with its Synergy-variant, which allows resource tuning. W1 represents a workload split with a good mix of resource-sensitive as well as resource-insensitive jobs. W2 is a workload dominated by resource-sensitive jobs, which is one of the worst-case scenarios for multi-dimensional scheduling as it could lead to GPU fragmentation (explained in §5.5)

We plot the results in Figure 14. Tuning resource allocation across jobs using Synergy reduced the average JCT of DRF by 7.2× and that of Tetris by 1.8× for the workload split W2. This is because Synergy is able to allocate auxiliary resources in a fungible-manner every round, whereas the big-data scheduler's static allocations performs similar to greedy techniques, resulting in GPU fragmentation, and thereby degrading the overall cluster metrics. Synergy performs the best in each scenario as it uses the best-case resource demands of jobs to perform fungible, disproportionate allocation.

6 Discussion and Future Work

Use of MinIO. Synergy assumes the use of MinIO [29] because it is a DNN-aware caching mechanism that outperforms traditional OS page caching and allows performance predictability. If we do not use MinIO, we will have to profile the model at discrete memory allocations which will increase the profiling costs, and also potentially change the trends in profiling matrix. However, the use of minIO in Synergy makes cache performance predictable and hence reduces Synergy 's profiling costs – allowing optimistic profiling.

Tradeoff between consolidation and allocation. When multi-GPU jobs are split across physical servers, they may incur a penalty due to network communication [30, 41]. DNN jobs therefore prefer consolidation. In this work, we assume that no more than a server's worth of CPU or memory resources can be allocated to a job if its GPU demands can be satisfied by one server. However, we find that some jobs may benefit from giving up consolidation if the throughput gain due to increased CPU or memory allocation is higher than the penalty due to splitting. It is an interesting future direction to explore the trade off between consolidation and allocation, while taking into account the network overhead.

Dataset locality. When DNN training jobs are submitted to a cluster scheduler, the input datasets for the job may reside on remote storage [5, 37]. When training begins, the dataset is first downloaded locally on the machine, and then loaded into the server memory. The former is constrained by the remote storage and network bandwidth, while the latter by the local storage bandwidth. In this work, we assume that the datasets are present locally on each server. Remote storage fetch will need the scheduler round duration to be inflated appropriately, a common concern for any round-based scheduler, and is not the focus of this work.

7 Related Work

DNN cluster schedulers A number of recent works target on cluster scheduling for DNN workloads each one focusing on improving a certain objective; Cluster utilization (Gandiva [41]), JCT (Tiresias [20]), and fairness (Themis [27], Gandiva-Fair [8]). Some works have also looked at exploting performance heterogeniety among accelerators to improve cluster objectives [26, 31]. All these schedulers assume GPU to be the dominant resource in the scheduling task; i.e., a user requests a fixed number of GPUs for her DNN job, and when the requested number of GPUs are all available, the job is scheduled to run. Other resources such as CPU and memory are allocated proportional to the number of GPUs requested by the job. Existing scheduler thus ignore *resource-sensitivity* of the DNN tasks to CPU, and memory. Synergy shows that resource-sensitivity allocation is an important factor to be considered for DNN workload scheduling on multi-tenant clusters, and can help achieve better cluster utilization.

Big data schedulers. Our work builds upon the insights drawn from the rich literature of schedulers for big data jobs [15, 17–19, 23, 38]. Big data schedulers like Tetris [17], and DRF [15] have looked at the problem of multi dimensional resource allocation for big data jobs. They propose new scheduling policies aimed at optimizing a specific cluster objective for jobs whose resource demands are prior known. Big data jobs come in with resource requirements along several dimensions, all of which is necessary to run that task. In contrast, the primary resource in a DNN job is the accelerator (GPU), whose requirement is specified by the job; other resources are fungible. Our work exploits this key insight to perform disproportionate allocations by profiling job resource sensitivity, and then appropriately packing them onto available servers.

Data stalls. A recent, deep characterization study explored the impact of CPU and memory on individual DNN jobs [29] Unlike prior work that focuses on individual jobs, the focus of our paper is on the tricks we can play when we schedule multiple jobs together in a cluster. Another way to look at this is: why should you pay for upgrading the number of CPUs, memory, or storage on your server when in fact you can cooperatively best use the existing resources across all the jobs being scheduled?

8 Conclusion

This paper introduces Synergy, a resource-sensitive DNN cluster scheduler. Synergy is based on the insight that, not all jobs exhibit the same level of sensitivity to CPU and memory allocation during DNN training, and hence breaking the shackles of GPU-proportional allocation can result in improved job and cluster-wide objectives. Our experiments on physical and large simulated clusters show that Synergy can reduce average JCT by upto 3.4× compared to GPU-proportional allocation.

References

[1] gRPC. https://grpc.io/.

[2] Microsoft philly traces. https://github.com/msr-fiddle/philly-traces.

[3] NVIDIA DALI. https://github.com/NVIDIA/DALI, 2018.

[4] Pytorch. https://github.com/pytorch/pytorch, 2019.

[5] Blobfuse. https://github.com/Azure/azure-storage-fuse, 2020.

[6] Wmt16. http://www.statmt.org/wmt16/, 2020.

[7] Brendan Burns, Brian Grant, David Oppenheimer, Eric A. Brewer, and John Wilkes. Borg, omega, and kubernetes. *Commun. ACM*, 59(5):50–57, 2016.

[8] Shubham Chaudhary, Ramachandran Ramjee, Muthian Sivathanu, Nipun Kwatra, and Srinidhi Viswanatha. Balancing efficiency and fairness in heterogeneous GPU clusters for deep learning. In *EuroSys '20: Fifteenth EuroSys Conference 2020, Heraklion, Greece, April 27-30, 2020*, pages 1:1–1:16. ACM, 2020.

[9] Mark Crovella, Robert Frangioso, and Mor Harchol-Balter. Connection scheduling in web servers. In *2nd USENIX Symposium on Internet Technologies and Systems, USITS'99, Boulder, Colorado, USA, October 11-14, 1999*. USENIX, 1999.

[10] Wei Dai, Chia Dai, Shuhui Qu, Juncheng Li, and Samarjit Das. Very deep convolutional neural networks for raw waveforms. In *2017 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 421–425. IEEE, 2017.

[11] Zihang Dai, Zhilin Yang, Yiming Yang, Jaime G. Carbonell, Quoc Viet Le, and Ruslan Salakhutdinov. Transformer-xl: Attentive language models beyond a fixed-length context. In *Proceedings of the 57th Conference of the Association for Computational Linguistics, ACL 2019, Florence, Italy, July 28- August 2, 2019, Volume 1: Long Papers*, pages 2978–2988. Association for Computational Linguistics, 2019.

[12] Michaël Defferrard, Kirell Benzi, Pierre Vandergheynst, and Xavier Bresson. Fma: A dataset for music analysis. *arXiv preprint arXiv:1612.01840*, 2016.

[13] Steven Diamond and Stephen P. Boyd. CVXPY: A python-embedded modeling language for convex optimization. *J. Mach. Learn. Res.*, 17:83:1–83:5, 2016.

[14] György Dósa and Jirí Sgall. First fit bin packing: A tight analysis. In Natacha Portier and Thomas Wilke, editors, *30th International Symposium on Theoretical Aspects of Computer Science, STACS 2013, February 27 - March 2, 2013, Kiel, Germany*, volume 20 of *LIPIcs*, pages 538–549. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2013.

[15] Ali Ghodsi, Matei Zaharia, Benjamin Hindman, Andy Konwinski, Scott Shenker, and Ion Stoica. Dominant resource fairness: Fair allocation of multiple resource types. In *Proceedings of the 8th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2011, Boston, MA, USA, March 30 - April 1, 2011*. USENIX Association, 2011.

[16] Google. Open images dataset. https://opensource.google/projects/open-images-dataset.

[17] Robert Grandl, Ganesh Ananthanarayanan, Srikanth Kandula, Sriram Rao, and Aditya Akella. Multi-resource packing for cluster schedulers. In *ACM SIGCOMM 2014 Conference, SIGCOMM'14, Chicago, IL, USA, August 17-22, 2014*, pages 455–466. ACM, 2014.

[18] Robert Grandl, Mosharaf Chowdhury, Aditya Akella, and Ganesh Ananthanarayanan. Altruistic scheduling in multi-resource clusters. In *12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016, Savannah, GA, USA, November 2-4, 2016*, pages 65–80. USENIX Association, 2016.

[19] Robert Grandl, Srikanth Kandula, Sriram Rao, Aditya Akella, and Janardhan Kulkarni. GRAPHENE: Packing and dependency-aware scheduling for data-parallel clusters. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 81–97, Savannah, GA, November 2016. USENIX Association.

[20] Juncheng Gu, Mosharaf Chowdhury, Kang G. Shin, Yibo Zhu, Myeongjae Jeon, Junjie Qian, Hongqiang Harry Liu, and Chuanxiong Guo. Tiresias: A GPU cluster manager for distributed deep learning. In *16th USENIX Symposium on Networked Systems Design*

and Implementation, NSDI 2019, Boston, MA, February 26-28, 2019, pages 485–500. USENIX Association, 2019.

[21] Awni Y. Hannun, Carl Case, Jared Casper, Bryan Catanzaro, Greg Diamos, Erich Elsen, Ryan Prenger, Sanjeev Satheesh, Shubho Sengupta, Adam Coates, and Andrew Y. Ng. Deep speech: Scaling up end-to-end speech recognition. *CoRR*, abs/1412.5567, 2014.

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

[23] Benjamin Hindman, Andy Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D. Joseph, Randy H. Katz, Scott Shenker, and Ion Stoica. Mesos: A platform for fine-grained resource sharing in the data center. In *Proceedings of the 8th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2011, Boston, MA, USA, March 30 - April 1, 2011*. USENIX Association, 2011.

[24] 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 *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 947–960, 2019.

[25] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in Neural Information Processing Systems*, pages 1097–1105, 2012.

[26] Tan N. Le, Xiao Sun, Mosharaf Chowdhury, and Zhenhua Liu. Allox: compute allocation in hybrid clusters. In Angelos Bilas, Kostas Magoutis, Evangelos P. Markatos, Dejan Kostic, and Margo Seltzer, editors, *EuroSys '20: Fifteenth EuroSys Conference 2020, Heraklion, Greece, April 27-30, 2020*, pages 31:1–31:16. ACM, 2020.

[27] Kshiteej Mahajan, Arjun Balasubramanian, Arjun Singhvi, Shivaram Venkataraman, Aditya Akella, Amar Phanishayee, and Shuchi Chawla. Themis: Fair and efficient GPU cluster scheduling. In *17th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2020, Santa Clara, CA, USA, February 25-27, 2020*, pages 289–304. USENIX Association, 2020.

[28] Stephen Merity, Caiming Xiong, James Bradbury, and Richard Socher. Pointer sentinel mixture models. In *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*. OpenReview.net, 2017.

[29] Jayashree Mohan, Amar Phanishayee, Ashish Raniwala, and Vijay Chidambaram. Analyzing and mitigating data stalls in DNN training. *Proc. VLDB Endow.*, 14(5):771–784, 2021.

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

[31] Deepak Narayanan, Keshav Santhanam, Fiodar Kazhamiaka, Amar Phanishayee, and Matei Zaharia. Heterogeneity-aware cluster scheduling policies for deep learning workloads. In *14th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2020, Virtual Event, November 4-6, 2020*, pages 481–498. USENIX Association, 2020.

[32] Misja Nuyens and Adam Wierman. The foreground-background queue: A survey. *Perform. Evaluation*, 65(3-4):286–307, 2008.

[33] Vassil Panayotov, Guoguo Chen, Daniel Povey, and Sanjeev Khudanpur. Librispeech: An ASR corpus based on public domain audio books. In *2015 IEEE International Conference on Acoustics, Speech and Signal Processing, ICASSP 2015, South Brisbane, Queensland, Australia, April 19-24, 2015*, pages 5206–5210. IEEE, 2015.

[34] Yanghua Peng, Yixin Bao, Yangrui Chen, Chuan Wu, and Chuanxiong Guo. Optimus: an efficient dynamic resource scheduler for deep learning clusters. In *Proceedings of the Thirteenth EuroSys Conference, EuroSys 2018, Porto, Portugal, April 23-26, 2018*, pages 3:1–3:14. ACM, 2018.

[35] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, et al. Imagenet large scale visual recognition challenge. *International journal of computer vision*, 115(3):211–252, 2015.

[36] Mark Sandler, Andrew Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. Mobilenetv2: Inverted residuals and linear bottlenecks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 4510–4520, 2018.

[37] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. The hadoop distributed file system. In Mohammed G. Khatib, Xubin He, and Michael Factor, editors, *IEEE 26th Symposium on Mass Storage Systems and Technologies, MSST 2012, Lake Tahoe, Nevada, USA, May 3-7, 2010*, pages 1–10. IEEE Computer Society, 2010.

[38] Vinod Kumar Vavilapalli, Arun C. Murthy, Chris Douglas, Sharad Agarwal, Mahadev Konar, Robert Evans, Thomas Graves, Jason Lowe, Hitesh Shah, Siddharth Seth, Bikas Saha, Carlo Curino, Owen O'Malley, Sanjay Radia, Benjamin Reed, and Eric Baldeschwieler. Apache hadoop YARN: yet another resource negotiator. In *ACM Symposium on Cloud Computing, SOCC '13, Santa Clara, CA, USA, October 1-3, 2013*, pages 5:1–5:16. ACM, 2013.

[39] Gerhard J. Woeginger. There is no asymptotic PTAS for two-dimensional vector packing. *Inf. Process. Lett.*, 64(6):293–297, 1997.

[40] Yonghui Wu, Mike Schuster, Zhifeng Chen, Quoc V. Le, Mohammad Norouzi, Wolfgang Macherey, Maxim Krikun, Yuan Cao, Qin Gao, Klaus Macherey, Jeff Klingner, Apurva Shah, Melvin Johnson, Xiaobing Liu, Lukasz Kaiser, Stephan Gouws, Yoshikiyo Kato, Taku Kudo, Hideto Kazawa, Keith Stevens, George Kurian, Nishant Patil, Wei Wang, Cliff Young, Jason Smith, Jason Riesa, Alex Rudnick, Oriol Vinyals, Greg Corrado, Macduff Hughes, and Jeffrey Dean. Google's neural machine translation system: Bridging the gap between human and machine translation. *CoRR*, abs/1609.08144, 2016.

[41] 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 *13th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2018, Carlsbad, CA, USA, October 8-10, 2018*, pages 595–610. USENIX Association, 2018.

[42] 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 *14th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2020, Virtual Event, November 4-6, 2020*, pages 533–548. USENIX Association, 2020.

[43] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. Spark: Cluster computing with working sets. In *2nd USENIX Workshop on Hot Topics in Cloud Computing, HotCloud'10, Boston, MA, USA, June 22, 2010*. USENIX Association, 2010.

[44] Xiangyu Zhang, Xinyu Zhou, Mengxiao Lin, and Jian Sun. Shufflenet: An extremely efficient convolutional neural network for mobile devices. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 6848–6856, 2018.