Accelerating Containerized Machine Learning Workloads

Ali Tariq*†, Lianjie Cao†, Faraz Ahmed†, Eric Rozner*, and Puneet Sharma†
*University of Colorado Boulder, *Hewlett Packard Labs

***Abstract*—To facilitate various Machine Learning (ML) training and inference tasks, enterprises tend to build large and expensive clusters and share them among different teams for diverse ML workloads. Virtualized platforms (containers/VMs) and schedulers are typically deployed to allow such access, manage heterogeneous resources and schedule ML jobs in these clusters. However, allocating resource budgets for different ML jobs to achieve best performance and cluster resource efficiency remains a significant challenge. This work proposes NEARCHUS to accelerate distributed ML training while ensuring high resource efficiency by using adaptive resource allocation. NEARCHUS automatically identifies potential performance bottlenecks for running jobs and re-allocates resources to provide optimized run-time performance with high resource efficiency. NEARCHUS's resource configuration significantly improves the training speed of individual jobs up to 71.4%–129.1% against state-of-the-art resource schedulers, and reduces job completion and queuing time by 35.6% and 67.8%, respectively.**

***Index Terms*—Machine Learning, Cloud Computing, Resource virtualization and management**

I. INTRODUCTION

Distributed training ML models is resource-intensive and time-consuming, especially as model sizes and training datasets grow. These distributed training techniques can run on dedicated enterprise clusters with heterogeneous resources (*e.g.,* CPU, GPU, TPU, FPGA, or SmartNICs [1]) or even on multi-tenant clouds. Container platforms and schedulers (*e.g.,* Kubernetes and OpenPAI) are increasingly adopted to manage ML clusters for better flexibility, scalability, and elasticity.

To reduce costs, clusters can be shared among tenants for running ML jobs with assigned resource budgets. However, assigning and managing resources for these jobs can be challenging. Consider a Parameter Server (PS)-based distributed ML training job that contains different types of training nodes (*e.g.,* chief, parameter servers, workers, and evaluators [2]). Each node in the PS architecture [3] is assigned a set of tasks, and hence training nodes may exhibit distinct resource requirements and resource usage. Simple resource configurations such as equal resource allocation may waste resources on some nodes, while overloading resources on other nodes, all leading to sub-optimal training performance and resource efficiency. Even training nodes of the same type may experience unbalanced task assignment [4], which further complicates the resource management problem. Configuring resources for ML training jobs is tedious and error-prone because the best resource configuration often changes for different ML jobs. As ML becomes more broadly applicable, users without enough domain knowledge may struggle to configure resources optimally. The problem becomes more challenging in multi-tenant container platforms, where sophisticated resource management and job scheduling [5], [6] are needed to deal with ever-increasing resource demands and varying resource availability.

We propose NEARCHUS to automatically find the best resource configuration for distributed ML training jobs. With PS distribution strategy, NEARCHUS determines the number of parameter servers and workers and their resource allocation (referred as *DT-config*) to achieve high resource efficiency and training performance. Instead of assigning the entire resource budget at once, NEARCHUS launches a training job with a subset of a resource budget and quickly and intelligently ramps up. This work-conserving approach improves overall resource utilization and reduces job queuing time. NEARCHUS continuously tracks training performance, compute and network utilization, and automatically adjusts DT-config to achieve better resource utilization and training performance.

We conduct performance benchmarking (§III) with multiple Deep Neural Networks (DNNs) using different DT-configs and observe resource utilization is the dominant indicator of training performance. We identify two performance bottlenecks (maximum utilization bottleneck and parameter server bottleneck) that can evaluate existing DT-configs. NEARCHUS automatically detects such performance bottlenecks and adapts in order to achieve better resource utilization and training performance(§IV). Compared to performance modelling approaches ([7], [8]), NEARCHUS does not require any prior knowledge or pre-training process, which eliminates significant modelling overhead and complements other workload-level optimizations (*e.g.,* PS load distribution [4] and pipeline parallelism [9], [10]) without modifications. We evaluate NEARCHUS using a diverse set of models (§V) to show the benefits against other solutions. This paper makes the following contributions:

- We present a measurement study of containerized ML training jobs showing how major bottlenecks can affect training performance. These bottlenecks are not addressed by the state-of-the-art modeling-based solution, Optimus [7].
- We design NEARCHUS that automatically detects and addresses various performance bottlenecks to achieve better training speed without incurring extra profiling overhead.
- We present system evaluation, NEARCHUS increases training speed up to 129%, and reduces job completion and queuing time by 35% and 67% in single-job and multi-job scenarios against Optimus and exhaustive search.

II. Related Work

Distribution strategies for ML training. There are several different ways to distribute ML training workload on multiple compute devices including data parallelism (*e.g.,* AllReduce [11] and Parameter Server [3]) and model parallelism (*e.g.,* pipeline parallelism [12] and tensor parallelism [13]). As a popular distribution strategy, significant efforts have been made to improve the parameter server distribution strategy. Previous works [3], [7], [14] have shown that the number of workers and parameter servers can directly impact the performance of model training and tried to determine the optimal distributed configuration based on various key metrics. Cruise [14] presents a cost-based optimization model to automatically find a good combination of workers and parameter servers along with model and data partitions for parameter servers. PLSD [4] and Parameter Service [15] focus on addressing run-time performance bottlenecks of parameter servers. These works use fixed resource allocation and only modify the number of workers and parameter servers. Optimus [7] and SLAQ [8] perform dynamic resource allocation and allocate resources among workers and parameter servers based on the quality improvements (*e.g.,* training loss) of ML models at run-time to achieve better resource efficiency. However, prior work has shown that loss-based prediction models can yield inaccurate results [16] which is further confirmed by our evaluation. In contrast, NEARCHUS performs dynamic resource allocation by monitoring resource utilization and does not depend on any pre-trained performance models.

Resource management for distributed ML training. As accelerators (*e.g.,* GPU and TPU) are increasingly adopted to accelerate ML workload, there has been a large body of work proposed to efficiently allocate accelerator resources for distributed ML training jobs. Tiresias [16] efficiently schedules deep learning jobs to reduce their job completion times (JCTs). Gandiva [17] exploits intra-job predictability to time-slice GPUs efficiently across multiple jobs, thereby delivering low latency and higher resource utilization. AntMan [18], Salus [19], HiveD [20], and GSLICE [21] are systems that enable sharing of GPU resources among training jobs by incorporating dynamic GPU resource sharing to maximize performance. Despite the wide adoption of GPU resources for running distributed ML jobs, dynamic and fine-grained GPU sharing on container platforms is still not mature.

Meanwhile, CPUs remain the preferred compute resource for many ML workloads, especially those that cannot be highly parallelized, require large memory, or have stringent cost limits [22]. Sparsh [23] highlights various benefits of using CPUs for DNN workloads and proposes various techniques for both inference and training, for optimizing model execution in the context of mobile, desktop/server, and distributed systems. CPUs can also enable wider and deeper network structures [24], [25], [26]. Training sparse DNNs can be highly inefficient on massively parallel processors [27], [28] because of their irregular memory accesses and inability to leverage optimizations such as cache tiling and vectorization, leading to inefficient GPU utilization. Meta adopted the parameter server architecture with CPU-based training for Deep Learning Recommendation Models in one of their production systems [29]. Furthermore, RNNs are difficult to parallelize [30] due to the dependencies between the steps. CPUs are preferred in such cases and may even outperform accelerators [31] with limited parallelism because of their advanced memory management. Due to this widespread CPU application and maturity of CPU virtualization, we restrict NEARCHUS to CPUs only.

Cluster	Servers	CPU	Memory	Network
A	9	2x Intel E5-2660v3 10-core	128 GB	1 Gbps
B	9	2x Intel Xeon 8375C 32-core	256GB	50 Gbps
C	4	2x Intel Xeon 6142 16-core	384 GB	10 Gbps
D	22	2x Intel E5-2660v3 10-core	128 GB	10 Gbps

TABLE I: Testbed configuration.

III. Measurement Study

We start with a measurement study that aims to answer the following questions: 1) what determines training speed (§III-A)?, 2) what factors affect the training speed (§III-B)?, 3) how to allocate the given resource budget (§III-C)?. For ease of discussion, we use (n_{ps}, n_{wk}) to denote a resource configuration of n_{ps} parameter server nodes and n_{wk} worker nodes for a training job, $n_{ps} + n_{wk}$ training nodes in total.

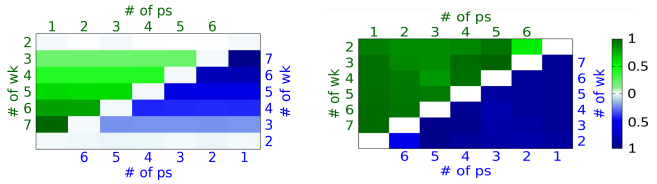
We use multiple clusters shown in Table I) for experiments. DNN models are developed using TensorFlow and training jobs are executed using Kubernetes and Kubeflow.

A. What determines training speed?

ML training jobs often use distributed strategies and previous works [7], [14] optimize resource configuration of PS-based distributed training jobs by changing the number of parameter server and worker nodes. For example, Optimus [7] demonstrates the relationship between resource configuration and training speed and aims to find an optimal combination for worker and parameter server nodes. We argue that such resource configuration may not accurately capture the performance characteristics of distributed training jobs. We first reproduce Optimus [7] results by training the ResNet-50 model with a fixed total number (*i.e.,* 8) of parameter server and worker nodes, each of which is allocated with 2.5 vCPUs.

The experiments are conducted on Cluster A, and each training node is placed on a dedicated server to avoid potential resource contention. We repeat the same training job by changing the relative number of parameter server and worker nodes while ensuring the total number of training nodes remains 8. Figure 1a shows the normalized training speed and corresponding CPU usage. We see linearly decreasing training speed as the number of worker nodes decrease or the number of parameter server nodes increase (green diagonal values), which aligns with Optimus' findings. We then extend experiments to cover more resource configurations with fewer than 8 worker and parameter server nodes in total (green upper triangle values). We observe configurations with different numbers of parameter server nodes but the same number of worker nodes yields similar training speeds. This confirms the aforementioned relationship between number of

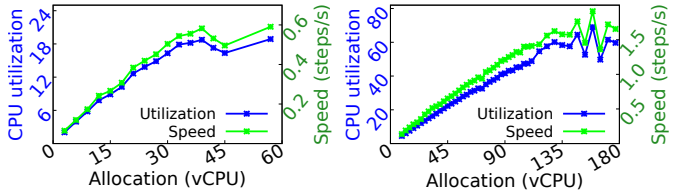
worker nodes and training speed. However, we also find a similar relationship between total CPU utilization and training speed (blue values in Figure 1a). Therefore, we modified the experiments to confirm which one, # of PS/WK nodes or CPU utilization, is the dominant factor of training speed.



(a) Pod allocation (b) Component allocation

Fig. 1: Normalized training speed (steps/sec) and CPU utilization.

We reuse the same set of resource configurations, *i.e.,* the total number of parameter server and worker nodes is less than 8, and keep the total resource allocation of 20 vCPUs. However, the resource allocation per training node is no longer a fixed value of 2.5 vCPUs. We vary the allocation of the 20 vCPUs among parameter server and worker nodes. For example, we tested 1 vCPU for parameter server nodes and 19 vCPUs for worker nodes, 5 vCPUs for parameter server nodes and 15 vCPUs for worker nodes, and 10 vCPUs for parameter server nodes and worker nodes respectively. vCPUs are evenly distributed among parameter server or worker nodes. In the case of 1 vCPU for parameter server nodes and 19 vCPUs for worker nodes and resource configuration $(2ps, 6wk)$, each parameter server node and worker node get 0.5 vCPU and 3.17 vCPUs respectively. Figure 1b shows the training speed and CPU utilization for the modified experiments with 1 vCPU for parameter server nodes and 19 vCPUs for worker nodes. With the same set of resource configurations, we observe that the training speed is no longer in a linear relationship with the number of workers as shown in Figure 1a. However, training speed still has a strong correlation with CPU utilization.



(a) $(1ps, 2wk)$ (b) $(10ps, 11wk)$

Fig. 2: Training speed and CPU utilization for two configurations.

To further confirm our observation, we scale the experiments and run them on a larger cluster — Cluster D with 22 servers. We tested two different resource configurations $(1ps, 2wk)$ and $(10ps, 11wk)$. We selected $(1ps, 2wk)$ as it is the smallest configuration and $(10ps, 11wk)$ as it is one of the largest configurations that fit on Cluster D while keeping one training node per server. Figure 2 shows how training speed increases linearly with CPU utilization. This confirms a strong correlation between CPU utilization and training speed of the same job irrespective of the resource configuration (*i.e.,* number of parameter server and worker nodes).

Our experiments show that training speed has a strong correlation with CPU utilization. Previous works improve training speed by finding the best resource configuration with fixed per-node CPU allocation. However, this approach only indirectly enhances training speed, as not all allocated resources can be efficiently utilized (as shown in Figure 1). Additionally, the resource configuration can only be adjusted by one training node per iteration, resulting in a long search process (e.g., Optimus adds one parameter server node or worker node with 5 vCPUs every 10 minutes [7]). Therefore, we need a new resource configuration solution that can directly improve resource utilization and accelerate the process of identifying the best resource configuration.

In summary, compute resource utilization is the dominant indicator of training speed. To improve the training speed of a given job, we must improve resource utilization.

B. What causes performance degradation?

In this subsection, we try to identify the performance bottlenecks that hinder parameter server-based ML training jobs from efficiently utilizing the allocated vCPUs. We use these insights as guidelines to design NEARCHUS in §IV.

Maximum Utilization Bottleneck. We start our analysis by running a single-worker training job of training ResNet-50 [32] on CIFAR-10 [33] dataset without any distribution strategy. We repeat the same training job on 20-core, 32-core, and 64-core servers and observe the CPU utilization increases with more compute resources allocated to the worker node. But it stagnates at 16 vCPUs, 21 vCPUs, and 37 vCPUs before reaching the maximum capacity as shown in Figure 3.

This limitation can be attributed to maximum parallelism limits [34] of single TensorFlow process [35] (recent work [36] details these topics). We identify this limitation as Maximum Utilization Bottleneck (MU-bottleneck) which limits the performance of individual workers in training jobs.

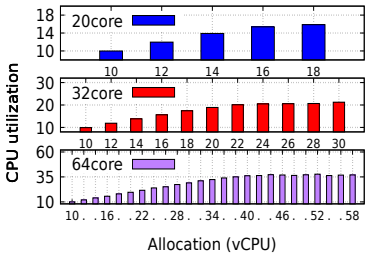


Fig. 3: Single-worker MU.

We then repeat the same ResNet-50 training job with parameter server distribution strategy on a 64-core server. We keep one parameter server node while increasing the number of worker nodes. The parameter server node is allocated with fixed 5 vCPUs to ensure that it does not become the bottleneck. All worker nodes evenly share the rest 59 vCPUs; hence each worker node is allocated with fewer vCPUs as we increase the number of worker nodes. Figure 4 shows how CPU utilization increases from 37 vCPU for a single worker to 46 vCPU for 2 workers and to 57 vCPU for 10 workers. This shows that MU-bottleneck is a relative value that is mainly caused by the multiprocessing overhead of ML frameworks (*e.g.,* Ten-

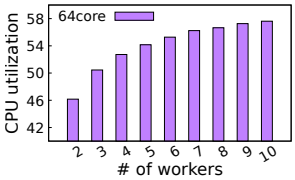


Fig. 4: Multi-worker MU.

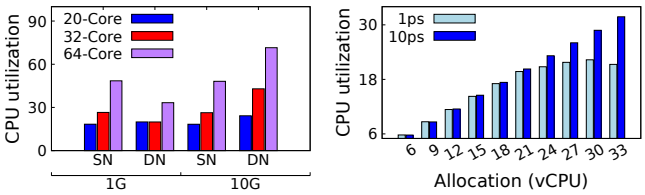
sorFlow) and, may further depend on several environmental factors such as server configuration, OS, dependent libraries, and the ML model. However, this can be avoided by adding more training nodes or dynamically allocating node resources.

Parameter Server Bottleneck. In distributed training, parameter server nodes may become overloaded when training large ML models, requiring up-scaling to achieve the desired training performance. We classify it into two categories, parameter server compute bottleneck (PSC-bottleneck) and parameter server network bottleneck (PSN-bottleneck).

PSC-bottleneck. For a given training job and compute resource budget, over-provisioning parameter server nodes can lead to inefficient resource utilization due to the MU-bottleneck while the idle resources could have been utilized by the worker nodes more efficiently. However, if parameter server nodes are under-provisioned, it not only slows down PS-related tasks but may also cause worker nodes to wait longer for the updated model parameters, which we identify as PSC-bottleneck. Both scenarios result in sub-optimal training speed. Striking the right balance is crucial, but also challenging. Furthermore, adding compute resources (*i.e.,* scaling up) to the same parameter server nodes may encounter the aforementioned MU-bottleneck, while adding parameter server nodes (*i.e.,* scaling out) may experience the parameter server straggler problem [4] that results in unequal resource requirements of different parameter server nodes.

PSN-bottleneck. Parameter server nodes pull and push gradients from/to worker nodes to update model parameters through communication channels across network links. To understand the impact of network links, we run ResNet-50 training job with $(1ps, 2wk)$ in two different scenarios: *SameNode (SN)* with all training nodes co-located on the same server and *DiffNode (DN)* with each training node allocated to a dedicated server. Since all training nodes are placed on the same server in SameNode, we ensure no network-related bottleneck. We repeat the experiments with 2 types of networks (*i.e.,* 1 Gbps and 10 Gbps Ethernet) and 3 server configurations (*i.e.,* 20-core, 32-core, and 64-core servers). Figure 5a shows the results. With restricted network bandwidth (*i.e.,* 1 Gbps), the resource efficiency of DiffNode is significantly lower than SameNode. Note that although the total CPU utilization of `SN-1G` and `DN-1G` may not be very different, the total CPU allocation of DiffNode is 3 $\times$ SameNode as each training node is assigned to a dedicated server. For example, in `DN-1G-64core`, only 33 vCPUs out of 196 vCPUs are utilized. Each worker node can only utilize 16 vCPUs due to the impact of both MU-bottleneck and PSN-bottleneck. With 10 Gbps network bandwidth, we observe increased resource utilization when comparing DiffNode to SameNode as PSN-bottleneck is no longer present. However, there is still a large amount of allocated resources not utilized due to the MU-bottleneck. For example, in `DN-10G-64core`, a worker node used 36 vCPUs, increased from 16 vCPUs in `DN-1G-64core`, which corresponds to our observation in the MU-bottleneck experiments. Since there is no external network traffic in SameNode, the results of `SN-1G` and `SN-10G` are identical on the three server sizes. However, when comparing `DN-10G` to `DN-1G`, we observe ~17.7%, ~53.5% and ~53.4% decrease in CPU utilization on 20-core, 32-core, and 64-core servers respectively, which is mainly caused by the PSN-bottleneck.

The parameter server can be scaled out when training large ML models and to understand the impact of increasing parameter server nodes, we run a new set of experiments in which we compare 1 parameter server node $(1ps, *wk)$ and 10 parameter server nodes $(10ps, *wk)$ on 1 Gbps network. We gradually increase the number of worker nodes from 2 to 11 and each of them is placed on a dedicated server, with 3 vCPUs allocated to conservatively avoid MU-bottleneck. Figure 5b shows the CPU utilization of all (ps, wk) pairs. As the number of worker nodes increases, the resource utilization of $(1ps, *wk)$ starts to stagnate at $(1ps, 8wk)$ while the resource utilization of $(10ps, *wk)$ shows continuous linear increments. It is mainly because the network traffic is spread across all parameter server nodes reducing per-node workload. This shows how we can reduce the impact of the PSN-bottleneck by carefully placing the parameter server nodes on different servers.



(a) SameNode and DiffNode using 1Gbps vs 10Gbps links.

(b) Single vs multiple parameter servers using 1Gbps link.

Fig. 5: Parameter server bottleneck.

In summary, ML training jobs can encounter various performance bottlenecks that reduce CPU utilization and training speed. Identifying and addressing such bottlenecks can help improve training performance.

C. How to allocate the given resource budget?

We consider the most common scenario of a containerized cluster shared by multiple tenants. Each tenant submits ML training jobs associated with resource budgets, which are stored in a job queue. The cluster resource manager determines when to launch a training job, how to allocate the given resource budget to parameter server and worker nodes, and where to place them. Most existing works [7], [37], [38], [39] address this problem by formulating optimization problems or building mathematical performance models. However, such model-based solutions may introduce significant modeling and profiling overhead and the performance models may not always provide accurate results during runtime. We will further illustrate this point using Optimus [7] as a specific example.

For a given job, Optimus requires a pre-training stage to fit a model for calculating the remaining steps, and a resource-speed model for calculating the duration of one training step regarding the number of parameter server and worker nodes. It requires running the training job with 10 different (ps, wk) configurations to obtain an accurate resource-speed

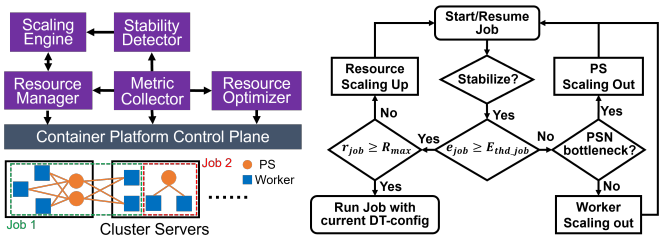
model. To evaluate Optimus, we run 5 jobs, each of which trains a unique ML model, *i.e.,* MobileNet, DenseNet-201, NasNetLarge, ResNet-50, and InceptionV3. To ensure the quality of the resource-speed model, we used 20 different (ps, wk) configurations in the profiling stage for each job. Then we run the training job with the fitted performance models and a resource budget of 100 vCPUs. We observe that MobileNet and DenseNet-201 achieved 82% and 86% resource efficiency, while NasNetLarge, ResNet-50, and InceptionV3 only achieved 63%, 43%, and 21% resource efficiency. Our analysis shows the three larger ML models encountered PSN-bottleneck in the middle of the training process as more training nodes were added. This was not accurately captured by the performance models because the profiling stage needs to make a trade-off between completeness and overhead; hence, performance models may fail to capture all possible situations during training, leading to sub-optimal configuration decisions.

In summary, avoiding performance bottlenecks is critical when allocating resource budgets. However, such bottlenecks can be dynamic and difficult to predict in multi-tenant clusters, which is why model-based solutions are unable to capture them despite additional profiling overhead.

IV. NEARCHUS DESIGN

To address the aforementioned performance bottlenecks and circumvent the limitations of model-based solutions, we propose NEARCHUS that detects various performance bottlenecks in real-time training and adaptively allocates compute resources to parameter server and worker nodes to achieve high resource utilization and desired training performance. NEARCHUS extends the definition of resource configuration to consider both the number of training nodes and their resource allocation, referred to as DT-config $(n_{ps}, n_{wk}, r_{ps}, r_{wk})$, where n_{ps}, n_{wk}, r_{ps}, and r_{wk} are the number of parameter server and worker nodes and the resources allocated to them respectively. As a result, the search space for the best DT-config becomes much larger. NEARCHUS aims to find the DT-config that yields the highest resource utilization through an iterative and directed search process. Similar to Optimus, NEARCHUS launches a training job with a subset of the given resource budget with an initial DT-config and iteratively adds more resources to the training job until the resource budget is reached. Therefore, NEARCHUS can launch a training job when only part of the resource budget is available, which reduces the job queuing time caused by gang scheduling [6].

Different from model-based solutions, NEARCHUS doesn't profile or train performance models for allocating resources. Instead, it detects different performance bottlenecks in real time and adjusts the current DT-config to resolve them (*i.e.,* scaling out). If no performance bottleneck is detected, the current DT-config is considered appropriate and NEARCHUS increases the resource allocation (*i.e.,* scaling up). Different from Optimus which adds one training node per iteration, NEARCHUS accelerates the scaling-up process by performing multiplicative increase. This process iterates until no further



(a) NEARCHUS architecture (b) NEARCHUS workflow

Fig. 6: NEARCHUS design.

improvements in resource utilization can be achieved and the last DT-config is used till the end of the training job. Figure 6 shows the architecture and workflow of NEARCHUS.

Resource Scaling Up. NEARCHUS launches a new training job with a proportion, R_0, of the resource budget, R_{max}. R_0 is uniformly distributed to each training node of $(1ps, 2wk)$; hence, the initial DT-config is $(1, 2, R_0/3, 2R_0/3)$. *Metric Collector* collects resource utilization metrics and training statistics of running jobs. For effective scheduling, it is crucial to ensure that the metrics are stable and reliable for detecting performance bottlenecks. When starting new jobs in a shared environment, it takes time for metrics to stabilize due to resource contention and cold starts [40]. *Stability Detector* applies a strict coefficient-of-variance ($CV < 5\%$) for $N_{comp} = 5$ consecutive samples to decide whether a training job is stabilized (5 to 8 minutes) based on two major metrics: compute resource utilization and training speed.

Once stabilized, the *Scaling Engine* aggregates the compute resource utilization of all training nodes and calculates the current overall compute resource efficiency of the training job. Let u_i denote the resource utilization of a training node $i \in \{1, 2, ..., n_{ps} + n_{wk}\}$ and let a_i denote the resource assignment of node i, then the resource efficiency is defined as $e_{job} = \frac{\sum_i u_i}{\sum_i a_i}$. If $e_{job} \geq E_{thd\_job}$ ($E_{thd\_job} = 0.95$ default threshold), we consider the current DT-config is not experiencing any performance bottleneck and all allocated resources are efficiently utilized by the training job. Hence, NEARCHUS increases the resource allocation by a configurable scale-up ratio R_{up} ($R_{up} = 50\%$ by default) that controls how aggressively resources scale. If the available resources in the cluster are less than the requested amount, the training job will only be scaled up with the available resources. Initially, the allocated resources are equally assigned among all training nodes, but the assignment may be adjusted later. The resources added by scaling-up decisions are proportionally added to each training node based on the previous resource assignments.

DT-config Scaling Out. Due to previously mentioned performance bottlenecks, the performance of a training job may not increase linearly with resource scaling up. For example, if worker nodes are assigned more resources than the MU-bottleneck, the training job is unable to efficiently utilize the allocated resources, leading to undesired training performance. This also wastes resources that potentially can be allocated to other training jobs. NEARCHUS detects the occurrence of performance bottlenecks by monitoring compute resource

efficiency, *i.e.,* $e_{job} < 0.95$, and addresses them by scaling out the current DT-config. DT-config scaling out is designed to handle the MU-bottleneck and PSN-bottleneck by adding worker and parameter server nodes respectively.

When a performance bottleneck occurs, the *Scaling Engine* needs to identify the type of the bottleneck. It first polls N_{net} ($N_{net} = 30$ by default) network utilization samples of each parameter node since the most recent scaling decision from *Metric Collector*. If more than 10% of the samples of a parameter server node are greater than 90% of the cluster network capacity, NEARCHUS considers PSN-bottleneck is detected. Hence, we scale out DT-config by adding a new parameter server node to re-balance the workload and network traffic to address this bottleneck. If the PSN-bottleneck is not detected, we consider the low compute resource efficiency is caused by the MU-bottleneck which means the worker nodes are allocated more resources than they can efficiently utilize. Hence, we scale out DT-config by adding a worker node and evenly redistributing the resource assignment among the same type of nodes, i.e., r_{ps}/n_{ps} or r_{wk}/n_{wk}, so each training node is assigned with less resources.

Resource Assignment Adjustment. Parameter server nodes use much less compute resources than worker nodes and their workload may not be evenly distributed depending on the tensor sizes of ML models. Additionally, ML framework (*e.g.,* TensorFlow) may adopt dynamic task scheduling on worker nodes at runtime. Therefore, the resource requirements of training nodes are not equal. For a given (n_{ps}, n_{wk}), the *Resource Optimizer* adjusts the assignment of the current resource allocation. The resource assignment adjustment happens at two levels: node level and component level (*i.e.,* parameter server and worker). *Resource Optimizer* computes resource efficiency of each training node periodically, $e_i = \frac{u_i}{a_i}$. If e_i is less than a predefined threshold E_{thd_opt} ($E_{thd\_opt} = 0.8$ by default), the node-level adjustment is triggered. The goal of node-level adjustment is to balance the resource efficiency of all training nodes of the same component by reclaiming idle resources from nodes with lower resource efficiency and redistributing them to other nodes proportionally. A successful node-level adjustment triggers component-level adjustment. Similarly, *Resource Optimizer* first computes the resource efficiency of the two components, e_{ps} and e_{wk}, and compares to E_{thd_opt}. If e_{ps} or e_{wk} is smaller than E_{thd_opt} while the other is larger than E_{thd_opt}, it means one component has idle resources that can be utilized more efficiently by the other component. Hence, we need to redistribute resources between parameter server nodes and worker nodes. If both e_{ps} and e_{wk} are smaller or larger than E_{thd_opt}, no component-level adjustment is needed. Idle resources are extracted from all nodes of the component with lower resource efficiency proportional to their resource assignment. The extracted resources are then proportionally added to the nodes of the other component. We assume the resource assignment of training nodes of the same component is already balanced during node-level adjustment, which is why node-level adjustments are executed first.

Model	# parameters	Network	Application Domain
MobileNet [41]	4.2M	CNN	Image classification
DenseNet-201 [42]	20M	CNN	Image classification
NasNetLarge [43]	84.9M	CNN	Image classification
ResNet-50 [44]	23M	CNN	Image classification
Inception-V3 [45]	24M	CNN	Image classification
CycleGAN [46]	28M	Transformer	Image generation
GNN application [47]	22K	GNN	Time series forecast
GPT using KerasNLP [48]	3.3M	Transformer	Text generation
Vision Transformer (ViT) [49]	21.7M	Transformer	Image classification
Bidirectional LSTM [50]	2.7M	RNN	Sentiment classification

TABLE II: Models used in evaluation.

System Implementation. We implement NEARCHUS in Python as a standalone job scheduler and resource manager, running on top of Kubernetes and Kubeflow. NEARCHUS uses a multi-queue system where the incoming training jobs first arrive in the *waiting queue*. Once the resources become available, *Resource Allocator* moves the job from *waiting queue* to *running queue* and launches the new job using R_0 compute resources equally assigned among the training nodes. *Metric Collector* keeps collecting the metrics for all jobs in the *running queue*. By default, *Resource Allocator* uses an FCFS policy to schedule the jobs in *waiting queue* and *running queue*, but it can be modified to use any other independent scheduling policy (*e.g.,* shortest job first) to better server to the needs of the incoming workloads. When a training job is completed, *Resource Allocator* moves it from *running queue* to the *finished queue*. Jobs are kept in *finished queue* along with its metrics until they can be moved to the logging archive.

V. EVALUATION

We evaluate NEARCHUS in two scenarios, single training job and multiple training jobs, with ML models listed in Table II. We compare NEARCHUS with two solutions, Optimus and exhaustive search, using job completion time (JCT) in hours and training speed in steps/sec. We also present microbenchmarks for understanding the impact of various configurable parameters in NEARCHUS. Cluster A (100 vCPUs compute resources and 1 Gbps network) and Cluster B (300 vCPUs and 50 Gbps network) from § III are used to represent resource-constrained and resource-abundant clusters.

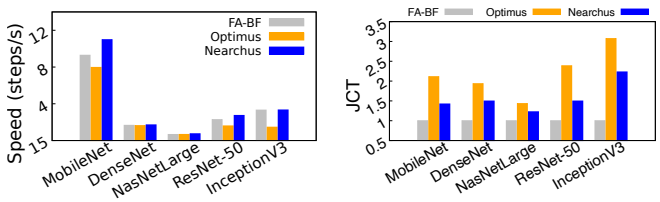
Optimus. We run profiling to derive the performance models for Optimus on both Cluster A and B using a maximum of 20 and 60 training nodes respectively. Each training node is allocated with fixed 5 vCPUs. Similar to § III-C, we use 20 configurations for profiling, more than Optimus [7] suggested.

Fixed-allocation brute force (FA-BF). FA-BF represents the best (n_{ps}, n_{wk}) identified through an offline exhaustive search. It uses 5 vCPUs per training node, the same as Optimus; hence, it can be considered the best achievable performance by Optimus. Note that FA-BF use the same static (n_{ps}, n_{wk}) the entire training process, which is different from the online search of Optimus and NEARCHUS.

A. Single-job Training

We first evaluate NEARCHUS when only one training job is executed at a time to limit resource contention between jobs. We use 5 different popular CNN models, each of which is allocated with the same resource budget of 100 vCPUs for

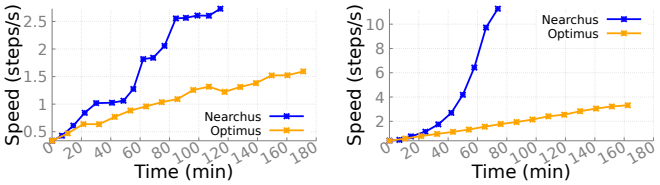
Cluster A and 300 vCPUs for Cluster B. Note that, we do not include the profiling stage for Optimus when comparing JCT.



(a) Training performance (b) Normalized JCT

Fig. 7: NEARCHUS evaluation in cluster A.

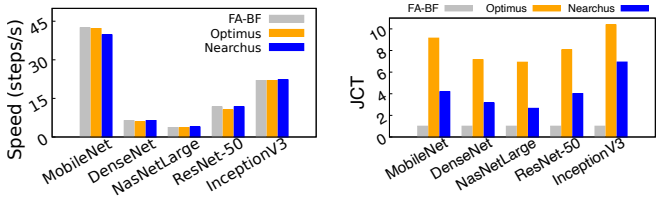
Figure 7a shows the best training speeds achieved by all solutions in Cluster A. NEARCHUS outperforms Optimus by up to 71.4% and 129.1% for ResNet-50 and InceptionV3. NEARCHUS also achieves 18.3% and 19.9% improvement compared to FA-BF for MobileNet and Resnet-50. Although FA-BF represents the best possible (n_{ps}, n_{wk}), NEARCHUS yields better training speed by performing finer-grained resource adjustments among all training nodes. For JCT shown in Figure 7b, FA-BF has the shortest JCT because it uses the total job budget from the beginning. In contrast, Optimus and NEARCHUS start with $(1ps, 2wk)$ and 15 vCPUs and ramp up the allocation while searching for the best DT-config. NEARCHUS achieves up to 37.2% reduction of JCT when compared to Optimus for ResNet-50.



(a) Cluster A (b) Cluster B

Fig. 8: Evolution of training speeds for ResNet-50.

NEARCHUS is able to effectively utilize its multiplicative scaling strategy because of the reactive approach and fine-grain resource assignment, that collectively allows training jobs to reach the resource budgets much faster without causing resource waste. Figure 8 compares how training speed improves over time for ResNet-50 between Optimus and NEARCHUS. NEARCHUS improves the training speed at a much faster pace which helps the job finish faster. This explains why NEARCHUS is able to achieve smaller JCT even for training jobs that have similar peak training speeds. Figure 8b shows the same job running on Cluster B. We see much smoother convergence curves in this case because of fewer performance bottlenecks in resource abundant environment.



(a) Training performance (b) Normalized JCT

Fig. 9: NEARCHUS evaluation in cluster B.

Figure 9 shows the training speeds and JCT for Cluster B. Since Cluster B has large resources, there are fewer performance bottlenecks for the profiling stage of Optimus and the training stage of all three solutions. Hence, the performance models of Optimus work well. Figure 9a shows similar peak training speeds achieved by FA-BF, Optimus, and NEARCHUS. However, as shown in Figure 9b, NEARCHUS outperforms Optimus for all 5 CNN models, reducing the JCT by up to 61.6% (NasNetLarge). This is mainly due to the additive increase of adding one training node per iteration by Optimus.

In addition to CNN models, we also evaluated NEARCHUS with other types of ML models against Optimus, including graph neural network (GNN), generative adversarial network (CycleGAN), language models (GPT), vision transformer (ViT), and RNN (LSTM). Figure 10 show NEARCHUS outperforms Optimus by a minimum of 14.4% for LSTM, and up to 55.6% for the GPT model, on Cluster A.

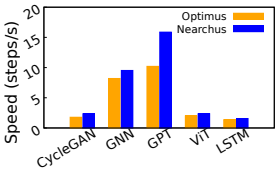


Fig. 10: Training speed of non-CNN models.

B. Multi-job Training

In production, ML training jobs are often submitted by different tenants sharing the same cluster. And those training jobs may run in parallel to maximize cluster resource efficiency. In this subsection, we evaluate NEARCHUS in such multi-job environment with synthetic job traces.

We create 6 variants of each of the 5 CNN models with different numbers of epochs and resource budgets to create a pool of 30 ML training jobs. We then create a job trace using Poisson distribution with an average arrival rate of 1 job per hour. The job trace contains 40 job arrivals and we randomly pick one training job from the training job pool for each arrival. We carefully design the job trace to avoid extreme overloading of clusters, especially for the resource-constrained Cluster A. We confirm this by analyzing the queued jobs over time, which varies from no queuing to at most 6 queued jobs. We repeat the same job trace on both Cluster A and B.

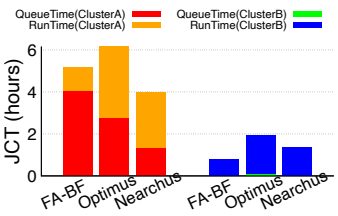


Fig. 11: Average JCT on Clusters A (left) and B (right).

Figure 11 summarizes the average JCT on both clusters. JCT in a multi-job environment consists of the job queuing time and the actual job training time (referred to as run time). In Cluster A, training jobs may need to wait for available resources more often than Cluster B which is why all three solutions show significant queuing time for individual jobs. Similar to single job experiments, FA-BF shows the shortest average job run time because it uses the whole budget from the beginning and does not require any online resource allocations. However, this also results in large queuing time as the individual jobs can only start when the entire resource budget

is available. Optimus and NEARCHUS, on the other hand, show much shorter queuing time, 32.6% and 67.8%, compared to FA-BF because the jobs can be launched with less available resources. NEARCHUS achieves the best average JCTs among the three: 23.6% and 35.6% less than FA-BF and Optimus. This is accomplished by allowing for slow starts which results in short queuing time and making efficient scaling decisions with faster resource allocation ramp-up which provides better job run time. In Cluster B, all three solutions experience very short queuing time due to sufficient available resources. For job run time, FA-BF again shows the shortest run time and NEARCHUS outperforms Optimus by a 24.6% shorter run time.

C. Microbenchmarks

We investigate overheads related to NEARCHUS via microbenchmarks for the core configuration parameters.

Restart overhead. Both NEARCHUS and Optimus modify resource configuration on the fly which incurs training restarts on a container platform. Figure 12 shows the number of restarts of NEARCHUS and Optimus on Cluster A and Cluster B. NEARCHUS on average requires fewer restarts compared to Optimus in both resource-constraint (Cluster A) and resource-abundant (Cluster B) environments. Unlike Optimus in which the number of restarts increases linearly with more training nodes added, NEARCHUS reduces subsequent restarts because of its multiplicative scaling strategy and reactive adjustment to performance bottlenecks, resulting in lower overall overhead.

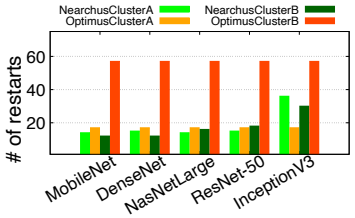
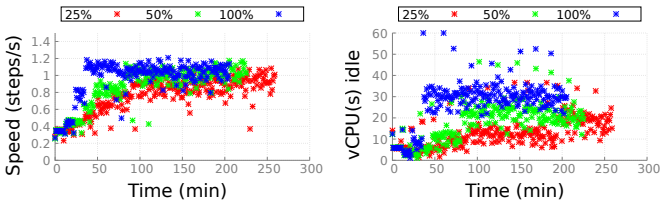


Fig. 12: Job restart analysis.



(a) Training Speed (steps/s) (b) Idle vCPU(s)

Fig. 13: Sensitivity analysis of multiplicative scaling.

Multiplicative scaling. NEARCHUS uses a multiplicative scaling ratio, R_{up}, to control how fast the resource allocation approaches the resource budget. An ideal scaling ratio should keep a proper trade-off between short convergence time (*i.e.,* how long it takes to find the best DT-config) and high CPU efficiency. A larger scaling ratio allows faster convergence to the resource budget, while a smaller scaling ratio enables a more thorough searching process for the best DT-config leading to high resource efficiency at the end. Figure 13a shows the training speeds over time, for ResNet-50 training jobs when using different scaling ratios and Figure 13b shows the relative idle CPUs over time. The results show higher training speeds on average for the larger scaling ratios but it also ends up having more idle CPUs. In practice, the resource requirement is a result of various performance bottlenecks in the environment which cannot be known beforehand. Therefore, we use 50% as the default scaling ratio.

VI. DISCUSSION AND CONCLUSION

Sub-optimal solutions. To achieve the best training performance, jobs require an optimal resource configuration. However, due to the vastness of the entire potential search space, identifying a globally optimal solution is a formidable challenge. NEARCHUS employs a heuristic-based approach to execute a locally directed search that expedites the search process. By permitting a degree of tolerance for suboptimal solutions, NEARCHUS effectively mitigates search overhead while preserving substantial performance improvements.

CPU vs. GPU. NEARCHUS focuses on CPU rather than GPU resource allocation. However, GPUs have similar resource efficiency problems that not all streaming multiprocessors (SM) can be efficiently utilized by ML workload [51]. However, fine-grained GPU allocation on container platforms is not available; we are unable to adjust GPU allocation to containerized training nodes as fine-grained as CPUs. There are recent efforts to enable such functionalities on container platforms [52], [53], [54], but none of them is mature enough or publicly available. However, NEARCHUS is applicable to GPU clusters with proper implementation modifications if fine-grained GPU management is available since it doesn't require prior knowledge of training jobs or underlying resources. NEARCHUS makes scaling decisions based on the performance bottlenecks and compute resource efficiency which are independent of the type of compute resource.

Parameter server vs. AllReduce. Different from parameter server approach, worker nodes in other distribution strategies and frameworks [55], [56] exchange model updates with each other directly. While NEARCHUS may not apply directly in such cases since the PS-worker architecture doesn't exist, the resource allocation of worker nodes including the number of nodes and per node resource assignment still needs to be fine-tuned for better training speed and resource efficiency. The basic idea of real-time performance bottleneck detection and dynamic resource adjustment of NEARCHUS can still be leveraged. For example, in a cluster with heterogeneous GPUs, the number of worker nodes and the resource allocation need to be carefully decided to avoid stragglers that slow down the training speed. This is particularly important when multiple different parallelisms are used together [36].

Conclusion. In this work, we first provide a detailed measurement study to understand the performance bottlenecks of distributed ML jobs. Inspired by our findings, we propose NEARCHUS to accelerate distributed ML training by making adaptive resource allocation decisions. NEARCHUS automatically detects performance bottlenecks when running ML training jobs and re-allocates resources to achieve optimized training speed. Our evaluation shows NEARCHUS can improve ML training speed up to 71.4% and 129.1%. It further reduces job completion time and queueing time by 35.6% and 67.8% respectively, when compared to a state-of-the-art solution.

REFERENCES

[1] Y. Li, I.-J. Liu, Y. Yuan, D. Chen, A. Schwing, and J. Huang, "Accelerating distributed reinforcement learning with in-switch computing," in *2019 ACM/IEEE 46th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2019, pp. 279–291.

[2] "Parameter server training with parameterserverstrategy," https://www.tensorflow.org/tutorials/distribute/parameter_server_training, 2022.

[3] M. Li, D. G. Andersen, J. W. Park, A. J. Smola, A. Ahmed, V. Josifovski, J. Long, E. J. Shekita, and B.-Y. Su, "Scaling distributed machine learning with the parameter server," in *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'14. USA: USENIX Association, 2014, p. 583–598.

[4] Y. Chen, Y. Peng, Y. Bao, C. Wu, Y. Zhu, and C. Guo, "Elastic parameter server load distribution in deep learning clusters," in *Proceedings of the 11th ACM Symposium on Cloud Computing*, ser. SoCC '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 507–521. [Online]. Available: https://doi.org/10.1145/3419111.3421307

[5] M. Jeon, S. Venkataraman, A. Phanishayee, J. Qian, W. Xiao, and F. Yang, "Analysis of Large-Scale Multi-Tenant GPU clusters for DNN training workloads," in *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. Renton, WA: USENIX Association, Jul. 2019, pp. 947–960. [Online]. Available: https://www.usenix.org/conference/atc19/presentation/jeon

[6] Q. Weng, W. Xiao, Y. Yu, W. Wang, C. Wang, J. He, Y. Li, L. Zhang, W. Lin, and Y. Ding, "{MLaaS} in the wild: Workload analysis and scheduling in {Large-Scale} heterogeneous {GPU} clusters," in *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, 2022, pp. 945–960.

[7] Y. Peng, Y. Bao, Y. Chen, C. Wu, and C. Guo, "Optimus: An efficient dynamic resource scheduler for deep learning clusters," in *Proceedings of the Thirteenth EuroSys Conference*, ser. EuroSys '18. New York, NY, USA: Association for Computing Machinery, 2018. [Online]. Available: https://doi.org/10.1145/3190508.3190517

[8] H. Zhang, L. Stafman, A. Or, and M. J. Freedman, "Slaq: Quality-driven scheduling for distributed machine learning," in *Proceedings of the 2017 Symposium on Cloud Computing*, ser. SoCC '17. New York, NY, USA: Association for Computing Machinery, 2017, p. 390–404. [Online]. Available: https://doi.org/10.1145/3127479.3127490

[9] Y. Huang, Y. Cheng, A. Bapna, O. Firat, D. Chen, M. Chen, H. Lee, J. Ngiam, Q. V. Le, Y. Wu *et al.*, "Gpipe: Efficient training of giant neural networks using pipeline parallelism," *Advances in neural information processing systems*, vol. 32, 2019.

[10] D. Narayanan, A. Harlap, A. Phanishayee, V. Seshadri, N. R. Devanur, G. R. Ganger, P. B. Gibbons, and M. Zaharia, "Pipedream: generalized pipeline parallelism for dnn training," in *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, 2019, pp. 1–15.

[11] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard *et al.*, "{TensorFlow}: a system for {Large-Scale} machine learning," in *12th USENIX symposium on operating systems design and implementation (OSDI 16)*, 2016, pp. 265–283.

[12] Y. Huang, Y. Cheng, A. Bapna, O. Firat, M. X. Chen, D. Chen, H. Lee, J. Ngiam, Q. V. Le, Y. Wu, and Z. Chen, *GPipe: Efficient Training of Giant Neural Networks Using Pipeline Parallelism*. Red Hook, NY, USA: Curran Associates Inc., 2019.

[13] M. Shoeybi, M. Patwary, R. Puri, P. LeGresley, J. Casper, and B. Catanzaro, "Megatron-lm: Training multi-billion parameter language models using model parallelism," *arXiv preprint arXiv:1909.08053*, 2019.

[14] W.-Y. Lee, Y. Lee, J. S. Jeong, G.-I. Yu, J. Y. Kim, H. J. Park, B. Jeon, W. Song, G. Kim, M. Weimer, B. Cho, and B.-G. Chun, "Automating system configuration of distributed machine learning," in *2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS)*, 2019, pp. 2057–2067.

[15] J. Gu, M. Chowdhury, K. G. Shin, and A. Akella, "Elastic model aggregation with parameter service," *arXiv preprint arXiv:2204.03211*, 2022.

[16] J. Gu, M. Chowdhury, K. G. Shin, Y. Zhu, M. Jeon, J. Qian, H. Liu, and C. Guo, "Tiresias: A GPU cluster manager for distributed deep learning," in *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*. Boston, MA: USENIX Association, Feb. 2019, pp. 485–500. [Online]. Available: https://www.usenix.org/conference/nsdi19/presentation/gu

[17] W. Xiao, R. Bhardwaj, R. Ramjee, M. Sivathanu, N. Kwatra, Z. Han, P. Patel, X. Peng, H. Zhao, Q. Zhang, F. Yang, and L. Zhou, "Gandiva: Introspective cluster scheduling for deep learning," in *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. Carlsbad, CA: USENIX Association, Oct. 2018, pp. 595–610. [Online]. Available: https://www.usenix.org/conference/osdi18/presentation/xiao

[18] W. Xiao, S. Ren, Y. Li, Y. Zhang, P. Hou, Z. Li, Y. Feng, W. Lin, and Y. Jia, "AntMan: Dynamic scaling on GPU clusters for deep learning," in *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, Nov. 2020, pp. 533–548. [Online]. Available: https://www.usenix.org/conference/osdi20/presentation/xiao

[19] P. Yu and M. Chowdhury, "Fine-grained gpu sharing primitives for deep learning applications," in *Proceedings of Machine Learning and Systems*, I. Dhillon, D. Papailiopoulos, and V. Sze, Eds., vol. 2, 2020, pp. 98–111. [Online]. Available: https://proceedings.mlsys.org/paper/2020/file/f7177163c833dff4b38fc8d2872f1ec6-Paper.pdf

[20] H. Zhao, Z. Han, Z. Yang, Q. Zhang, F. Yang, L. Zhou, M. Yang, F. C. Lau, Y. Wang, Y. Xiong, and B. Wang, "Hived: sharing a gpu cluster for deep learning with guarantees," in *Proceedings of the 14th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'20.

[21] A. Dhakal, S. G. Kulkarni, and K. K. Ramakrishnan, "Gslice: Controlled spatial sharing of gpus for a scalable inference platform," in *Proceedings of the 11th ACM Symposium on Cloud Computing*, ser. SoCC '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 492–506. [Online]. Available: https://doi.org/10.1145/3419111.3421284

[22] C. Zhang, M. Yu, W. Wang, and F. Yan, "Enabling cost-effective, slo-aware machine learning inference serving on public cloud," *IEEE Transactions on Cloud Computing*, vol. PP, pp. 1–1, 07 2020.

[23] S. Mittal, P. Rajput, and S. Subramoney, "A survey of deep learning on cpus: Opportunities and co-optimizations," *IEEE Transactions on Neural Networks and Learning Systems*, vol. PP, 04 2021.

[24] G. Li, M. Müller, G. Qian, I. C. D. Perez, A. Abualshour, A. K. Thabet, and B. Ghanem, "Deepgcns: Making gcns go as deep as cnns," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2021.

[25] G. Li, C. Xiong, A. Thabet, and B. Ghanem, "Deepergcn: All you need to train deeper gcns," 2020.

[26] M. Liu, H. Gao, and S. Ji, "Towards deeper graph neural networks," ser. KDD '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 338–348. [Online]. Available: https://doi.org/10.1145/3394486.3403076

[27] B. Chen, T. Medini, J. Farwell, s. gobriel, C. Tai, and A. Shrivastava, "Slide : In defense of smart algorithms over hardware acceleration for large-scale deep learning systems," in *Proceedings of Machine Learning and Systems*, I. Dhillon, D. Papailiopoulos, and V. Sze, Eds., vol. 2, 2020, pp. 291–306. [Online]. Available: https://proceedings.mlsys.org/paper/2020/file/65b9eea6e1cc6bb9f0cd2a47751a186f-Paper.pdf

[28] Z. Gong, H. Ji, C. Fletcher, C. Hughes, and J. Torrellas, "Sparsetrain: Leveraging dynamic sparsity in software for training dnns on general-purpose simd processors," 09 2020, pp. 279–292.

[29] M. Naumov, J. Kim, D. Mudigere, S. Sridharan, X. Wang, W. Zhao, S. Yilmaz, C. Kim, H. Yuen, M. Ozdal, K. Nair, I. Gao, B.-Y. Su, J. Yang, and M. Smelyanskiy, "Deep learning training in facebook data centers: Design of scale-up and scale-out systems," 2020.

[30] M. Zhang, S. Rajbhandari, W. Wang, and Y. He, "Deepcpu: Serving rnn-based deep learning models 10x faster," in *Proceedings of the 2018 USENIX Conference on Usenix Annual Technical Conference*, ser. USENIX ATC '18. USA: USENIX Association, 2018, p. 951–965.

[31] H. Li, Z. Wang, X. Yue, W. Wang, H. Tomiyama, and L. Meng, "An architecture-level analysis on deep learning models for low-impact computations," *Artificial Intelligence Review*, pp. 1–40, 06 2022.

[32] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016, pp. 770–778.

[33] A. Krizhevsky, V. Nair, and G. Hinton, "Cifar-10 (canadian institute for advanced research)." [Online]. Available: http://www.cs.toronto.edu/~kriz/cifar.html

[34] "Tensorflow:tf.config.threading," https://www.tensorflow.org/api_docs/python/tf/config/threading/set_inter_op_parallelism_threads, accessed: 2022-02-22.

[35] "Tensorflow:tf.config.threading," https://www.tensorflow.org/api_docs/python/tf/config/threading/set_intra_op_parallelism_threads, accessed: 2022-02-22.

[36] L. Zheng, Z. Li, H. Zhang, Y. Zhuang, Z. Chen, Y. Huang, Y. Wang, E. P. Xing, Y. Xu, D. Zhuo, J. E. Gonzalez, and I. Stoica, "Alpa: Automating inter- and intra-operator parallelism for distributed deep learning," 2022. [Online]. Available: https://arxiv.org/abs/2201.12023

[37] D. Didona, F. Quaglia, P. Romano, and E. Torre, "Enhancing performance prediction robustness by combining analytical modeling and machine learning," 01 2015.

[38] P. Malakar, P. Balaprakash, V. Vishwanath, V. Morozov, and K. Kumaran, "Benchmarking machine learning methods for performance modeling of scientific applications," in *2018 IEEE/ACM Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS)*, 2018, pp. 33–44.

[39] C. Witt, M. Bux, W. Gusew, and U. Leser, "Predictive performance modeling for distributed computing using black-box monitoring and machine learning," 2018.

[40] A. Verma, L. Pedrosa, M. R. Korupolu, D. Oppenheimer, E. Tune, and J. Wilkes, "Large-scale cluster management at Google with Borg," in *Proceedings of the European Conference on Computer Systems (EuroSys)*, Bordeaux, France, 2015.

[41] A. G. Howard *et al.*, "Mobilenets: Efficient convolutional neural networks for mobile vision applications," 2017.

[42] G. Huang *et al.*, "Densely connected convolutional networks," 2018.

[43] B. Zoph *et al.*, "Learning transferable architectures for scalable image recognition," 2018.

[44] K. He *et al.*, "Deep residual learning for image recognition," 2015.

[45] C. Szegedy *et al.*, "Rethinking the inception architecture for computer vision," 2015.

[46] A. K. Nain, "Cyclegan," https://keras.io/examples/generative/cyclegan/, 2020.

[47] A. Khodadadi, "Traffic forecasting using graph neural networks and lstm," https://keras.io/examples/timeseries/timeseries_traffic_forecasting/, 2021.

[48] J. Chan, "Gpt text generation from scratch with kerasnlp," https://keras.io/examples/gen-erative/text_generation_gpt/, 2022.

[49] K. Salama, "Image classification with vision transformer," https://keras.io/examples/vision/image_classification_with_vision_transformer/, 2021.

[50] fchollet, "Bidirectional lstm," https://keras.io/examples/nlp/bidirectional_lstm_imdb, 2020.

[51] A. Chowdhery, S. Narang, J. Devlin, M. Bosma, G. Mishra, A. Roberts, P. Barham, H. W. Chung, C. Sutton, S. Gehrmann *et al.*, "Palm: Scaling language modeling with pathways," *arXiv preprint arXiv:2204.02311*, 2022.

[52] "Nvidia multi-process service," https://docs.nvidia.com/deploy/mps/index.html, 2022.

[53] "Nvidia multi-instance gpu user guide," https://docs.nvidia.com/datacenter/tesla/mig-user-guide/, 2022.

[54] J. Cho, D. Zad Tootaghaj, L. Cao, and P. Sharma, "Sla-driven ml inference framework for clouds with heterogeneous accelerators," *Proceedings of Machine Learning and Systems*, vol. 4, pp. 20–32, 2022.

[55] A. Sergeev and M. Del Balso, "Horovod: fast and easy distributed deep learning in tensorflow," *arXiv preprint arXiv:1802.05799*, 2018.

[56] P. Moritz, R. Nishihara, S. Wang, A. Tumanov, R. Liaw, E. Liang, M. Elibol, Z. Yang, W. Paul, M. I. Jordan *et al.*, "Ray: A distributed framework for emerging {AI} applications," in *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, 2018, pp. 561–577.