Horus: Interference-Aware and Prediction-Based Scheduling in Deep Learning Systems

Gingfung Yeung, Damian Borowiec, Renyu Yang, Adrian Friday, Richard Harper, Peter Garraghan

Abstract—To accelerate the training of Deep Learning (DL) models, clusters of machines equipped with hardware accelerators such as GPUs are leveraged to reduce execution time. State-of-the-art resource managers are needed to increase GPU utilization and maximize throughput. While co-locating DL jobs on the same GPU has been shown to be effective, this can incur interference causing slowdown. In this paper we propose Horus: an interference-aware and prediction-based resource manager for DL systems. Horus proactively predicts GPU utilization of heterogeneous DL jobs extrapolated from the DL model's computation graph features, removing the need for online profiling and isolated reserved GPUs. Through micro-benchmarks and job co-location combinations across heterogeneous GPU hardware, we identify GPU utilization as a general proxy metric to determine good placement decisions, in contrast to current approaches which reserve isolated GPUs to perform online profiling and directly measure GPU utilization for each unique submitted job. Our approach promotes high resource utilization and makespan reduction; via real-world experimentation and large-scale trace driven simulation, we demonstrate that Horus outperforms other DL resource managers by up to 61.5% for GPU resource utilization, 23.7–30.7% for makespan reduction and 68.3% in job wait time reduction.

Index Terms—Distributed Systems, Deep Learning, Interference, GPU Utilization, Cloud computing, Workload prediction.

1 Introduction

Deep Learning (DL) is an increasingly important type of machine learning positioned to impact many fields. Innovation in DL architectures and growth in data volume has led to increased practitioner demand, resulting in the establishment of clusters of machines equipped with computer accelerators such as Graphical Processor Units (GPUs). These DL systems—comprising distributed systems at both small and large-scale—are leveraged to enable vast amounts of computation throughput and reduce total model training time [1], [2].

Cloud providers deploy and execute *DL workloads* (encapsulated as *DL jobs*) by provisioning resources as part of their service model [3], [4], [5]. An important goal for such DL systems is their ability to satisfy Service Level Agreements (SLA) and Quality of Service (QoS) criteria in a resource-efficient manner [6], [7]. Efforts to ensure such SLA and QoS guarantees are challenged due to GPU under-utilization [8], [9], [10]. This is due to existing resource managers such as Kubernetes [11] and YARN [12] *prohibiting the explicit use of GPU sharing* (i.e. only allowing a single DL job to be assigned to each GPU). Such under-utilization *decreases* performance, resource-efficiency, and service availability incurring longer queuing times [9], requiring additional GPU devices to satisfy demand.

The ability to *co-locate* DL jobs (i.e., execute on the same GPU) has been identified as a means to address under-utilization [13], [14], [15], [16]. The effectiveness of such co-location is based on a good understanding of DL workload GPU utilization patterns [8], [17], [18]. For providers, this enables high-quality DL system scheduling and co-location decisions that reduce GPU resource under-utilization. For consumers, this allows greater insight into potential GPU costs[1]. Understanding and exploiting DL workload utilization to improve co-location is critical for designing resource-efficient DL systems [19], [20], [10].

However, established approaches for characterizing GPU utilization from DL workloads leverage *online profiling* during execution. Online profiling entails executing each unique DL job on an isolated GPU (or dedicated machine) to ensure accurate metric collection [21], [22]. Such online profiling results in reduced service availability and resource-efficiency due to the need for reserved GPU devices: a growing problem given the increasing number of different model architectures and configurations [8]. Whilst co-location can improve GPU utilization, it also can incur *performance interference* (which we refer to as interference) resulting in an average DL job slowdown of 18% for different co-location combinations [8]. While DL resource managers now exist that allow for co-location [8], [6], [10], less attention has been paid to actively addressing interference between DL jobs sharing the same GPU during placement decisions. Poor DL job placement results in a higher makespan, increased Job Completion Time (JCT), job eviction, and job failures from GPU out-of-memory (OOM) errors [9].

In this paper we present Horus: a prediction-based interference-aware resource manager for DL systems. In contrast to existing approaches, Horus *proactively* predicts the GPU utilization of *unseen* DL jobs based on their model features, which are exploited by our scheduler to determine suitable DL job co-location combinations to minimize in-

- *G. Yeung, D. Borowiec, A. Friday, R. Harper and P. Garraghan are with the School of Computing & Communications, Lancaster University, UK. Email: {g.yeung1, d.borowiec, a.friday, r.harper, p.garraghan}@lancaster.ac.uk*
- *R.Yang is with the School of Computing, University of Leeds, Leeds, UK. Email: r.yang1@leeds.ac.uk*

Manuscript received XX XX 2021 (corresponding author: Renyu Yang)

1. AWS p3.16xlarge instance (8x NVIDIA V100 GPUs): $24.48/h (https://aws.amazon.com/ec2/pricing/on-demand/) [04/01/2021]

terference. Our approach avoids the need to profile kernel patterns [13], [21], [22], [10], modification of the underlying DL framework, nor require extensive online profiling of job execution requiring an isolated GPU at scheduler runtime—all of which are expensive and time consuming. We offer three specific research contributions:

- *Characterization of DL workload interference resulting from co-location.* We have characterized interference profiles of over 600 unique combinations of co-located DL jobs across heterogeneous GPU hardware architecture. Findings demonstrate that DL job co-location interference results in up to 2.4x–3.4x slowdown, and is comparable to network locality for distributed training.
- *GPU utilization analysis and prediction engine for DL workloads.* Through a series of benchmarks, we analyze and identify the key DL model features and their relationship to GPU utilization. These include Floating Point Operations Per second (FLOPs), input data size and DL computation graph structure such as number of convolution layers. Our proposed prediction engine allows for sub-second DL job GPU utilization prediction without a need for online profiling.
- *An interference-aware DL resource manager*. Exploiting our prediction engine, we propose an interference-aware resource manager supporting co-location and minimizing GPU over-commitment. Our approach offers two alternative scheduling algorithms that prioritize minimizing job makespan or improving fairness to avoid job starvation—lowering median job wait time at the expense of a marginal degradation to makespan and utilization. The resource manager was integrated into Kubernetes and deployed within a DL cluster, and evaluated at scale via trace-driven simulation of a production DL cluster. Results demonstrate that our approach achieves a 32–61.5% increase in GPU cluster utilization and up to 23.7–30.7% makespan reduction over existing approaches.

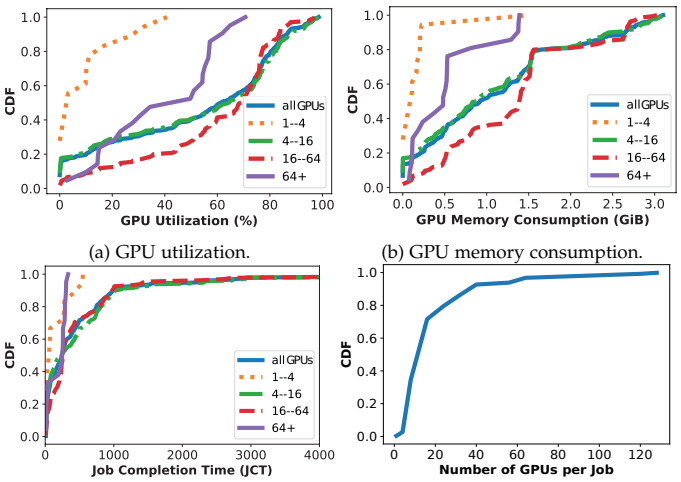
We expand upon our previous work [23], by increasing the scope of the DL workload characterization study from 81 to 292 models; capture additional GPU architectures and 600 more co-location profiles for analysis and modelling, improved GPU prediction model accuracy; and evaluate Horus at scale via trace-driven simulation of a production cluster. The Horus framework has also been redesigned to include a refined fair queuing scheduling algorithm to minimize a cost objective. Finally, the evaluation has been conducted with an additional set of workload compositions and an additional co-location algorithm for comparison [8].

The paper is structured as follows: §2 and §3 present the research background and job characterization study, respectively. §4 outlines design and implementation of the Horus system. §5 and §6 discuss experiment setup and results. §7 provides related work and §8 the conclusions.

2 Motivation

2.1 Background

Deep Learning (DL) are Deep Neural Networks (DNN) represented as a Directed Acyclic Graph (DAG) or computation graphs in execution. Each graph node is an operation (i.e. layer or combination of layers), containing parameter



(a) GPU utilization. (b) GPU memory consumption.

(c) JCT truncated at 4000 mins. (d) Number of GPUs per job.

Fig. 1: Job utilization and JCT from a production DL cluster (1 month).

information with access to its predecessor and successor. Model parameters are stored as floating point values, hence larger models in execution often result in a higher number of Floating Point Operations (FLOPs), and an increased requirement for GPU device memory. This is important as recent research demonstrate increasing DNN model depth and width can improve accuracy [24]. DNNs are frequently executed on GPUs due to the high performance capability to perform matrix multiplication on thousands of cores. Each operation is often expressed as computation or memory *kernels* on GPUs [25]. Hence a DL model with a large number of layers requires more kernels, resulting in greater GPU load driven by the number of FLOPS and the intermediate outputs (activations) of the network.

Deep Learning systems (DL systems) are clusters of machines containing one or more accelerators – predominately GPUs – employed to execute DL workloads. Users submit workloads into the DL system as *jobs* with various configurations (e.g. batch size, model, dataset). Jobs are then allocated onto the machines via the *resource manager*. Recent studies of production DL systems have identified the challenge of GPU under-utilization reflected by an average GPU utilization of 52% [9], and long queuing time for DL jobs of between 4,000s–8,000s due to head-of-line blocking [26]. We are able to corroborate such findings from conducting an analysis of a month-long trace of a 398 production DL jobs scheduled to a 500+ machine GPU cluster operated by a large global e-commerce company. As shown in Fig 1, we observed that half of the jobs have GPU Utilization less than or equal to 60% and JCT less than or equal to 300 minutes, averaging at 51% and around 500 minutes, respectively.

A primary cause of such under-utilization is the reliance on traditional, non-preemptive schedulers [11], [12], requiring each DL job to hold exclusive access to a GPU device. This is problematic due to its negative impact on job throughput, system availability, and resource-efficiency. Existing approaches have demonstrated a positive increase to DL system GPU utilization by enabling *co-location* of DL jobs on the same GPU [8], [6], [10], [18]. The effectiveness of co-location is dependent on two inter-related concepts: accurate GPU profiling and minimizing interference.

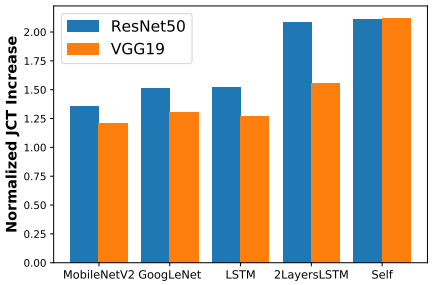


Fig. 2: DL job interference (Nvidia GTX 1080, Cifar10 dataset).

GPU profiling is used to ascertain the GPU utilization for jobs, and is known to be non-trivial to calculate [27]. In this context, GPU utilization is defined as the percentage of time in a given sample interval where one or more kernels executed on a GPU. It is important to note that this measurement is *not the actual utilization* of the processing elements core (chip area containing the floating-point, integer, tensor units), nor relates to the bytes read/written from device memory and cache. It is however, *a good estimate* of the amount of load required to keep the GPU busy within the measurement period. Profiling[2] is performed by collecting metrics related to a DL job on an isolated GPU or machine [14], [21], [22]. Profiling can be categorized into two types: *Coarse-grained profiling* obtains the number of kernels, kernel configuration, GPU/memory utilization, and kernel execution time, and usually takes several minutes to complete depending on the job. *Fine-grained profiling* requires accessing hardware performance counters including Achieved Occupancy and byte read/write throughput from DRAM for each observed kernel. Whilst more accurate when compared to coarse-grain profiling, this method is more intensive and takes longer to complete (minutes to hours), depending on the metrics measured and workload complexity. Whilst GPU profiling is used in existing DL resource managers for co-location decisions, such co-location also incurs performance degradation from interference.

Interference is a system phenomena occurring when multiple processes compete for the same limited set of resources on the same machine [28], [29], [30], [31]. GPU interference occurs with the same reason. Specifically, the limited set of processing elements and memory then causes queuing delays of the jobs' kernels [13], [22], [21], [16]. These kernels are launched by the GPU kernel scheduler, which follows policy similar to round-robin fashion [32]. Interference of co-located DL jobs has been shown to result in an 18% JCT degradation [8]. Our initial experiments of job co-location on an Nvidia GeForce GTX 1080 GPU depicted in Fig. 2, show that ResNet50 and VGG19 models experience up to 2.1x JCT slowdown when co-located with various other models. Such slowdown is problematic considering that DL jobs may perform model training in the region of hours to days. Hence, in order for DL systems to fully exploit co-location, maximizing resource utilization and minimizing makespan, DL resource managers should consider the effects of interference when performing DL job co-location during placement.

2. Nvidia tools: NSight Systems, NSight Compute and NVProf

TABLE 1: Micro-benchmark hardware setup.

Feature	System A	System B
CPU	Intel i7-6850K	AMD Ryzen 1920X
GPU	Nvidia GeForce GTX 1080	Nvidia GeForce RTX 2080
RAM	32GB	128GB

TABLE 2: Analyzed DL models. Datasets (CV): Cifar10 [33], batch sizes: {8, 16, 32, 64}. (NLP): WikiText2 [34] & News Commentary v14-en-zh [35]. NLP: sentence length: 200, vocab. 10000, batch sizes: {16,32,64}.

● - CV, ■ - NLP, ■ - FC

Architecture	Permutations
● MobileNet [36]	[default]
● MobileNetV2 [37]	[default, channel: $2^{i0\ldots n}$]
● MobileNetV3 [38]	[075, 100]
● GoogLeNet [39]	[default]
● ResNet[40]	[18, 18 - bottleneck 34, 50],
● VGGNet[41]	[11, 11 - bottleneck, 19]
● SqueezeNetV1[42]	[1.0]
● DenseNet[43]	[121, 161, 169]
● ShuffleNetV2[44]	[0.5, 1.0, 1.5, 2.0]
● MNASNet[45]	[0.5, 1.0, 1.3],
● DualPathNetwork[46]	[92, 26], blocks: [2,2,2,2]
● ProxyLess [47]	[cpu, gpu, mob, mob-14]
● PyramidNet[48]	depth: [48,84,270], alpha: [66,110]
● ResNeXt[49]	[11,29] cardinality: 2, width: [16,64]
■ LSTM [50]	ParaDNN implementation [20]
■ Gated Recurrent Unit [51]	ParaDNN implementation [20]
■ Fully Connected (custom)	ParaDNN implementation [20]

2.2 DL Utilization and Interference

Existing GPU and DL resource managers [13], [21], [16], [14], [8], [10], [15] alleviate interference effect by profiling kernel characteristics and GPU Utilization at runtime to orchestrate kernels scheduling order or opportunistically co-locate jobs. However, profiling DL job kernels at runtime to infer interference by creating suitable performance profiles may extend DL job training from minutes to hours. Moreover, profiling must be performed for every new job (DL model) submitted into the system, resulting in additional overhead in the system. Since GPU Utilization is correlated to the load of the GPU, we therefore turn to investigate the relationship between GPU Utilization and interference. It is imperative to understand how different DL model configurations affect the GPU Utilization, and exploit such information to ascertain co-location profiles with minimal interference.

Particularly, we conducted a set of relationship study to address the following questions: **[Q1]** Can we leverage GPU Utilization as a general proxy metric to estimate the interference level, i.e., JCT slowdown, without fine-grained profiling? **[Q2]** If so, can we exploit DL job characteristics and extract useful information to predict GPU Utilization?

3 Co-location Relationship Study

3.1 Profiling Setup

Environment. Micro-benchmarks were conducted using two different DL systems (A & B), described in Table 1. Leveraging methods established in the literature [15], [21], [22], DL model profiling was conducted using isolated GPUs, and by co-locating different combinations of DL jobs within the same GPU. Each micro-benchmark was repeated multiple times to ensure metric consistency. Both systems

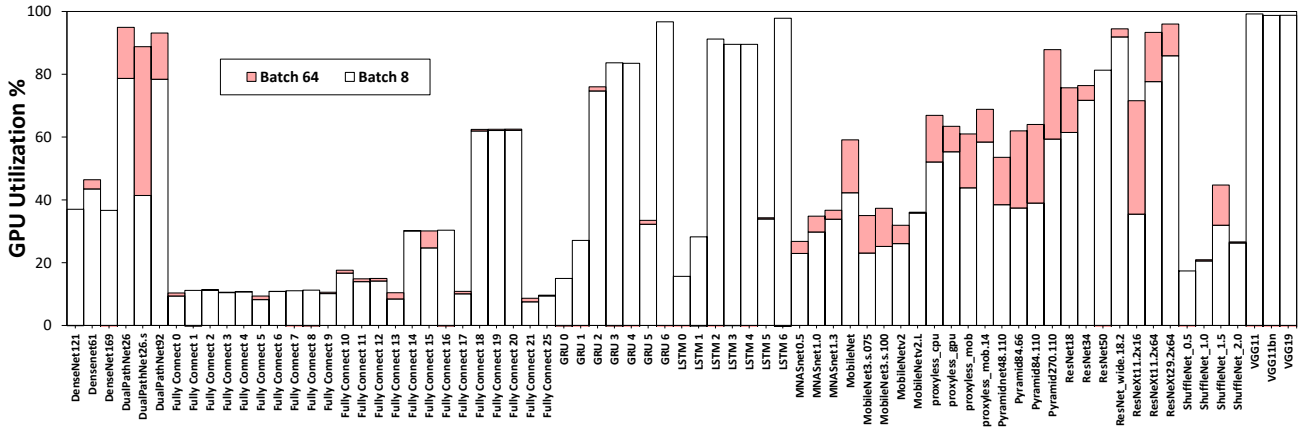


Fig. 3: Overview of DL workload GPU utilization differences (Nvidia RTX 2080).

used an Nvidia container runtime, CUDA Toolkit 10.2 and PyTorch 1.5 DL Framework [52].

DNN Models. We selected a wide variety of representative DL job types: 14 prominent computer vision (CV) models, 2 Natural Language Processing (NLP) and 1 custom Fully Connected (FC) model architecture, encompassing convolution neural networks (CNNs) and recurrent neural networks (RNNs), comparable to prior works [53], [26], [15], [54]. Each model architecture were then further refined into several different configurations by varying mini-batch size, hidden dimensions and number of layers to create a number of model permutations, as shown in Table 2 and Fig. 3. Within the memory constraints of GPU devices, this resulted in 292 unique configurations profiled in isolation with further 600 co-location combinations. We run these models with the highest capacity on our systems until they encountered OOM.

Metrics. In order to understand the impact of interference, we extracted several key metrics of interest including: GPU utilization, Job Completion Time (JCT) and kernel access patterns. Metrics were collected using `nvidia-smi`, `nvml-golang bindings`, and `Nvidia Nsight Systems`. We measured the impact of interference by analysing the corresponding JCT slowdown in each co-located execution case by comparing with the isolated execution case. JCT slowdown T_{deg} is measured as:

$$T_{deg} = \frac{|T_{colo} - T_{solo}|}{T_{solo}} \quad (1)$$

where T_{colo} is the time taken for a co-located DL job to reach a fixed time epoch, and T_{solo} is the time taken for the same DL job executing in isolation.

3.2 Relationship between GPU Util. and JCT Slowdown

In response to **[Q1]**, we observe that co-located DL jobs, with each requiring high GPU utilization, result in greater JCT slowdown due to more significant levels of resource contention by the scheduled kernels. This is intuitive as GPU utilization is driven by the degree in which kernels engage GPU's processing elements and memory.

As shown in Fig. 4, co-located job combinations with increasing levels of *GPU over-commitment* (i.e. the cumulative GPU utilization requirement greater than 100%) results in a JCT increase between 1.5x–3x; In contrast, pairs of co-located DL jobs which individually require less than 50% utilization are less likely to exhibit severe performance degradation, with an increase in JCT between 1x – 1.5x. The correlation of increased over-commit utilization with increased JCT suggests that utilization can be used as a proxy metric for determining job interference levels.

Without fine-grained profiling of individual DL job kernels, one can leverage GPU utilization w.r.t. each DL job as such proxy, when co-locating jobs with high load and determine scenarios which are likely to result in performance degradation. We observe an *approximate linear relationship* between accumulative GPU utilization and resultant JCT slowdown *before* GPU over-commitment manifests. In comparison, GPU over-commitment results in a *non-linear relationship* – a quadratic polynomial fits well to the data with the lowest R-squared difference indicated by 0.88 and and 0.84 for Nvidia 1080 [3] and Nvidia RTX 2080 [4], respectively.

Additionally, we investigate the impact of hardware heterogeneity on JCT slowdown. Fig. 4 reveals that on average the interference severity when running identical DL jobs is lower on the Nvidia RTX 2080 architecture than Nvidia 1080 due to additional processing elements, increased cache size, and larger memory bandwidth with the coefficient of the best-fit relationship differing for heterogeneous hardware.

3.3 Relationship between FLOPs and GPU Utilization

In response to **[Q2]**, we investigate the DNN computation graph and Fig. 5 illustrates a positive correlation between FLOPs and GPU utilization. This is because the DNN model has a larger number of parameters, and the number of activations will lead to more computation and memory kernels launched in the GPU device, further causing an increased GPU load.

In reality, both the number of matrix multiply and memory transaction increase when the batch size is increased due to the number of FLOPs and memory transactions are correlated to the number of elements within a batch of inputs, i.e., $B \times X$ where B is the batch size and X is the DNN inputs. Hence, by looking into the DNN computation graph, we can extract meaningful information to quantitatively determine the GPU utilization, which in turn allow us to accurately infer the most suitable job co-location scheme for reducing the performance interference in a deep learning cluster.

3. Function: $x^2(1.32198) + x(-0.00728) + 6 \times 10^{-5}$
4. Function: $x^2(1.16664) + x(-0.00302) + 4 \times 10^{-5}$

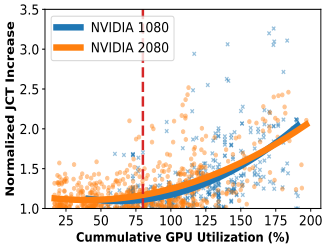


Fig. 4: Cumulative GPU Utilization and JCT slowdown.

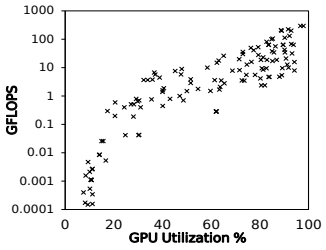


Fig. 5: Relation of GFLOP and GPU utilization.

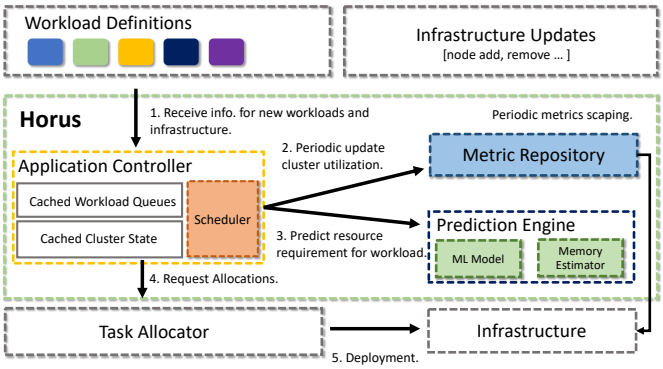


Fig. 6: Horus architecture – GPU utilization prediction engine and co-location scheduler deployed within a DL system resource manager.

4 Proposed Approach: Horus

Horus is a prediction-based interference-aware DL system resource manager, and has been designed as a set of components that can be deployed as part of existing cluster resource manager frameworks such as Kubernetes. Fig. 6 depicts Horus architecture and it comprises three main components: the *Prediction Engine*, the *Metric Repository* and the *Application Controller*. Upon job submission, the application controller sends a request to the prediction engine to estimate DL job GPU usage, i.e., GPU Utilization and GPU Memory Utilization by inspecting the workload definition (§4.1). Specifically, the prediction engine requires a way to access the DNN graph and dry run the model (e.g. downloading the .pth file). Cluster view is maintained through infrastructure updates and monitoring agents, to collect infrastructure data from each node including GPU usages and system usages (host memory usages, and CPU utilization). An agent is deployed on each individual node reporting application and system utilization metrics, which are eventually collected into the metric repository (§ 4.3).

The scheduler then assigns DL jobs to GPUs by computing their suitability—minimizing a cost function objective to support co-location w.r.t. cached cluster state. Our approach aims to maximize GPU utilization and minimize makespan via de-prioritizing co-location placement decisions that would result in JCT slowdown from severe interference and communication delays (§4.2).

4.1 Prediction Engine

4.1.1 Estimating GPU Utilization

Overview. The prediction engine extracts key DL workload features as described in Table 3 by iterating over the Open Neural Network Exchange (ONNX)[5] graph representation of the DL model. We can obtain aggregate features such as FLOPs by iterating each operators and calculate based on its inputs, output shape, and parameters. Features are then normalized and used as numerical inputs to a machine learning model in order to predict the GPU utilization ($\texttt{GUtil}_j$) of a given job j. We train the prediction model in an offline training stage based on a set of historical DL workload profile micro-benchmarks similar to existing prediction based approaches [14], [29]. These profiles are nominally acquired via developers running micro-benchmarks or by monitoring existing non co-located DL workloads on isolated GPUs. Critically, after successful prediction model training, there is no need for isolated profiling for unique DL workloads entering the system. It is worth noting that such an approach can also be combined with reactive approaches [8], [10], [15], [17], [18]. The machine learning model can be periodically retrained after collecting additional profiles (e.g. when new models are discovered).

5. https://onnx.ai/ [04/01/2021]

TABLE 3: ONNX model features.

Features
FLOPs, Memory Parameters, Batch Size, Memory Activations, Exponentials, Split, Constant, GlobalAveragePool, ReduceMean, MaxPool, GRU, Reshape, LSTM, Concat, Gather, Squeeze, Pad, BatchNormalization, AveragePool, Conv,Slice, Transpose, Flatten, Relu, Gemm

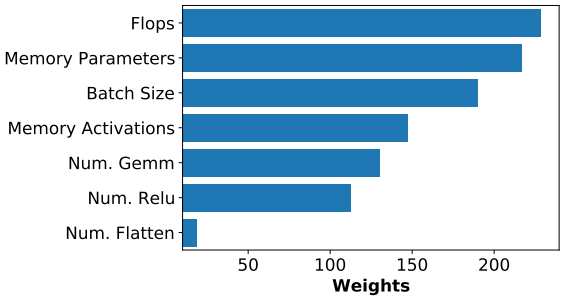


Fig. 7: Most important identified regressor features.

Feature Importance. To understand further what contributes towards GPU utilization, we investigate each of the tree based regressor feature weights by extracting the weights and averaged across them as shown in Fig. 7. These features are clear indicators and follow the existing literature of model compression and neural architecture search where reducing the number of parameters and intermediate activations can save computation and memory consumption of the hardware [37], [45]. Surprisingly, we found that the number of convolution and the remaining features had minimal to no impact on the regressor, and we plan to look into leveraging compiler intermediate representation for more hardware feature characteristics [25].

Model evaluation. Model accuracy was determined via measuring regressor Root Mean Square Log Error (RMSLE)—an established measure of regression accuracy when the under-prediction error is enlarged. This approach is useful for utilization prediction: whilst overestimating

TABLE 4: Regressors Root Mean Square Log Error (RMSLE) for GPU utilization prediction.

	Linear	LightGBM[55]	XGBoost[56]	Random Forest [57]
RMSLE	0.188	0.193	0.133	0.150

GPU utilization is not ideal in terms of maximizing resource efficiency, it is preferable to underestimation which could lead to unintended GPU over-allocation and interference that we attempting to avoid. Table 4 shows that all prediction models achieve a relatively low RMSLE score of 0.133.

4.1.2 Estimating GPU Memory Utilization

Compared with GPU utilization, estimating GPU memory utilization is more complex since total job memory size (MiB) is governed by initialization and optimization of individual DL libraries. Without looking into the kernels implementation, it is possible to estimate the minimum expected memory usage in bytes by considering the following four factors in both forward M^f and backward passes M^b: (i) the batch size of data B, (ii) the number of activations A, (iii) number of gradients G and (iv) the number of parameters P. In addition to an initialization overhead δ, the overall estimated memory requirement for a given DL job j will be:

$$\texttt{GMem}_j = M_j^f + M_j^b + \delta = (B * A + P) + B * G + \delta \quad (2)$$

The estimated GPU utilization ($\texttt{GUtil}_j$) and GPU memory ($\texttt{GMem}_j$) will be used for node capacity check in the scheduler in case of tackling an incoming job.

4.2 Interference-Aware Job Scheduling

Gandiva [8] placement strategy monitors application throughput, a job is killed or migrated to another node randomly upon slowdown detection using an undefined threshold value and time period. In such an approach, it is possible for random job migration to be allocated with another incompatible job leading to equal or greater performance slowdown. Antman [10] enables co-location by monitoring DL jobs, employing a local coordinator and modified the underlying DL frameworks to allow fine-grained control of DL jobs kernels, injecting idle time on a GPU to alleviate interference between co-located jobs. This approach, however, requires understanding and profiling of the kernels execution order at runtime to determine the appropriate idle time.

At the core of our interference-aware scheduling is to understand the compute resource requirement *prior* to job execution, and perform job placement with as least amount of cost as possible w.r.t the corresponding resources to the job. This is in contrast to existing DL system schedulers which *react* after obtaining workload utilization patterns.

4.2.1 Job Scheduling Plan

Problem formulation. Our objective is to find a job placement onto a cluster of nodes with GPU capacities that minimizes the cost value of all possible solutions. In this context, we use a decision variable X_{jng} to represent the node n's GPU g is allocated to the job j at the decision time, and $Cost_{jng}$ denotes the cost variable in this placement. The optimization problem can be therefore defined as the following Integer Linear Programming (ILP) problem:

TABLE 5: Notation definition.

Symbol	Description
J, j	Jobs awaiting scheduling, a job
N, n	Cluster node collection, a node
G_n	Available GPUs on node n
ω_i	Component weights in the objective function
R	Enumerated resource types: CPU(0), RAM(1),GMem(2)
r	a given resource type in R
CP_n^r	Capacity of resource r on node n
UR_n^r	Used resource r on node n
CP_{ng}^τ	Capacity of resource τ (GMem) on GPU g on node n
UR_{ng}^τ	Used resource τ (GUtil, GMem) on GPU g on node n
X_{jng}	1 if job j is allocated to GPU g on node n; 0 otherwise
RQ_j^r	Requested resource of job j for resource r
RQ_j^{GPU}	Requested GPU number of job j
$\texttt{GUtil}_j$	Estimated GPU utilization of a job j
$\texttt{GMem}_j$	Estimated GPU memory usage of a job j
β	the number of jobs considered in each scheduling round
k	queue numbers in the scheduler

$$\min \sum_{j\in J}\sum_{n\in N}\sum_{g\in G_n} Cost_{jng}\cdot X_{jng} \quad (3)$$

$$\text{s.t.} \sum_{n\in N}\sum_{g\in G_n} X_{jng} = RQ_j^{GPU}, \forall j \in J \quad (4)$$

$$\sum_{j\in J}\sum_{g\in G_n} RQ_j^r \cdot X_{jng} \leqslant CP_n^r - UR_n^r, \forall r \in R, \forall n \in N \quad (5)$$

$$X_{jng} = \{0,1\}, \forall j \in J, \forall n \in N, \forall g \in G_n \quad (6)$$

The constraints ensure at every time all GPU requests of each job can be satisfied (Constraint (Eq. 4) and the sum of any type of resources (i.e., CPU, memory, and GPU memory) requested by all jobs on any node must be within the bound of node free resource (Eq. 5). Subject to these constraints, we aim to minimize the involved cost of the overall GPU allocation among co-located jobs (Eq. 3). For clarity, notations used in this paper are summarized in Table 5.

Cost breakdown. To accurately capture the incurred cost and the impact of GPU co-location onto the DL job performance, we further break down the overall cost into two independent portions: GPU memory usage and GPU utilization increase:

$$Cost_{jng} = \omega_1 C_{jng}^{GMem} + \omega_2 C_{jng}^{GUtil} \quad (7)$$

wherein ω_i is a customized weight that indicates the performance impact and we set all weights equally by default.

Since higher GPU memory usage has a higher chance of OOM errors and JCT slowdown, the cost of GPU memory C_{jng}^{GMem} is inherently referred to as a proportion of GPU memory usage as a result of placing the job j (Eq. 8):

$$C_{jng}^{GMem} = \frac{UR_{ng}^{GMem} + \texttt{GMem}_j}{CP_{ng}^{GMem}} \quad (8)$$

where CP_{ng}^{GMem} is a fix number, i.e., the total GPU memory of the GPU device, while $\texttt{GMem}_j$ is the estimated GPU memory usage of job j and UR_{ng}^{GMem} is the used GPU memory within UR_{ng}^τ. Due to the relationships between increased GPU utilization of co-located DL jobs and JCT slowdown w.r.t hardware outlined in §3, we penalize the combinations of co-located DL jobs when over-commitment manifests. Specifically, let $\mathcal{F}$ be a set of functions that we trained and fitted on the JCT slowdown and cumulative GPU Utilization w.r.t GPU device g. f_τ^- and f_τ^+ are two function instances in $\mathcal{F}$ where f_τ^- represents the function when the targeting device is of τ type and cumulative GPU

Algorithm 1 Weighted Fair Queuing Based Job Scheduling

Input: (J, S, k, β) // Pending jobs, current cluster state, k queues and β jobs to consider into the buffer for each scheduling round.

1: *// Cluster the similar jobs into multi-tiered queues*
2: $\mathcal{Q} \leftarrow$ Put pending jobs into k queues via k-means (J, k)
3: **while** queues in $\mathcal{Q}$ is not empty **do**
4: $\widetilde{J} \leftarrow$ Pick β jobs into scheduling buffer via weighted fairness
5: **for** j in $\widetilde{J}$ **do**
6: **if** the cluster has allocatable resources (S) **then**
7: *// capacity check (CPUs, Mems, GPU Mems)*
8: $\mathcal{N} \leftarrow$ filter all nodes passing capacity check (j, S)
9: $\lambda \leftarrow j.requestedGPU$
10: $\sigma \leftarrow \lceil \lambda / \#GPUperNode \rceil$
11: **if** LEN($\mathcal{N}$) $< \sigma$ **then**
12: **continue**
13: *// calculate the cost of placing a job onto GPUs on the nodes*
14: $\mathcal{C}_j \leftarrow$ Eq. 7, ($j, \forall g \in G_n, \forall n \in \mathcal{N}$)
15: *// shortlist a collection of GPUs with min costs*
16: $\mathcal{G} \leftarrow$ select top-λ from $\mathcal{C}_j$ in ascending order
17: *// resource allocation*
18: SCHEDULE(j, $\mathcal{G}$)

utilization is not yet over-committed while f_τ^+ is used when the cumulative GPU utilization surpasses 100%.

Hence, the GPU cost can be expressed as:

$$C_{jng}^{GUtil} = \begin{cases} \mathrm{f}_\tau^+(\mathrm{GUtil}_{jng}), & \text{if } GUtil_{jng} > 100 \\ \mathrm{f}_\tau^-(\mathrm{GUtil}_{jng}), & \text{if } GUtil_{jng} \leq 100 \end{cases} \tag{9}$$

where $GUtil_{jng}$ is the estimated GPU utilization if job j is placed onto node n's GPU g:

$$GUtil_{jng} = UR_{ng}^{GUtil} + \mathtt{GUtil}_j \tag{10}$$

As our functions $\mathcal{F}$ was fitted against the JCT slowdown and GPU Utilization, we can directly use the outcome of the function as an estimated cost when these jobs are packed onto the GPU device. Therefore, the scheduling probability of the node would be inversely correlated to the JCT slowdown estimate. As this ILP problem is NP-hard and due to the heterogeneity of job resource requirements [26], [9], we have modified a cost based algorithm leveraged from [58] to greedily solve this scheduling problem.

4.2.2 Runtime Job Scheduling with Weighted Fair Queuing

As we observed in Fig. 1 that the JCTs vary substantially among jobs, it is critical to avoid head-of-line blocking and any forms of resource starvation – particularly incurred by long jobs with large resource requests. To schedule different jobs in a fair manner, we borrow the ideas from [59], [60]; (1) cluster similar jobs into several groups to individually manage the jobs in a group and (2) at each scheduling round, we fairly pick up a certain number of jobs from different queues, primarily considering the waiting time and queue length, and then assign the most suitable GPU resources to launch them in the GPU cluster. Alg. 1 outlines the algorithm details.

Job clustering. Before jobs are actually scheduled, we carry out a clustering procedure for all jobs. Specifically, the L1 Distance metric is used to identify similar jobs considering the following features: (1) Number of tasks; (2) GPU Utilization predicted; (3) GPU per task; and (4) GPU memory estimated. These features outline per-job resource requirements and can be obtained by adopting the method in §4.1. In practice, we run the *k-means* algorithm on all yet-to-execute jobs to identify similar jobs and put them into the corresponding queues, i.e., $\mathcal{Q} = [\mathcal{Q}_1, \ldots, \mathcal{Q}_k]$ (Line 2). We set k = 3 as we found that from Fig. 1a, the utilization patterns have 3 distinct CDFs.

Picking jobs based on weighted fair queuing. Presumably, β jobs are allowed, as a batch, into each scheduling round. To be fair, Horus picks up a certain number of pending jobs from each queue according to the queue weight, i.e., the degree of job pending (Line 4). more jobs are expected to be selected and processed from a queue with longer waiting time and larger queue length, we measure the weight as the product of job's median waiting time per queue and the queue length, i.e., $w_x = \mathtt{max}\{Len(\mathcal{Q}_x), Med(\mathcal{Q}_x) \times Len(\mathcal{Q}_x)\}, x = \{0 \ldots k\}$, where the $\mathtt{max}$ operation is to guarantee a non-zero value once median waiting time is zero when all jobs are new arrivals on the system. Median has a statistical property that is less affected by skewed data, thus can more accurately reflect the queuing time for a class of jobs. Eventually, the number of jobs picked from $\mathcal{Q}_x$ can be calculated by $\frac{w_x}{\sum_i^k w_i}\beta$. This design can actively avoid job *starvation* of any particular class of jobs – whenever a class of jobs start to starve, an increased number of jobs will be selected. We also allow reservation for starving jobs as the weighted fair queuing algorithm will select the jobs with the longest waiting time.

Resource allocation. Because all pending jobs are now well ordered into $\widetilde{J}$ according to the weighted fairness, the scheduler will try its best effort to allocate available resources to each job in turn whist minimizing the performance interference. Specifically, for each job, we check the resource capacity and select all the nodes ($\mathcal{N}$) that can satisfy all requirements of job j in terms of CPU, memory and GPU memory (Lines 8). The GPU memory requirement is inferred by using Eq. 2 in § 4.1. Based on the total number of GPUs required by the job and the number of GPUs per node, we calculate the minimal number of nodes that can meet the needs of job j (Lines 9-12). By using Eq. 7, we can then calculate the cost of scheduling a job onto each GPU of each node in $\mathcal{N}$ (Line 14) and pick the top-λ GPUs ($\mathcal{G}$) with the minimal costs (Line 16) before the final resource allocation and job scheduling (Line-18).

4.2.3 Other Considerations and Discussion

Job Failover and rescheduling. It is possible for our approach (as well as other DL resource managers) to encounter issues associated with OOM errors due to co-located DL jobs exceeding the total GPU memory capacity stemming from incorrect memory requirement estimation. We address this issue by using a separate thread to monitor job progress, and in the event of failure, jobs are resubmitted onto the scheduling queue similar to prior works [8], [29]. The scheduler will then update the DL job request with necessary GPU memory requirements, where GPU memory must be equal or greater than the memory available previously placed based on periodic infrastructure profiles.

Locality-based calibration. This work primarily tackles the JCT slowdown due to interference stemming from job co-location while optimizing the distributed job training is not the focus of this paper. The current job placement scheme assumes high-speed connection across-nodes, hence

the data transfer time during training is not the dominating factor in current algorithm design. The GPU interference aware scheduling is most suitable for jobs which do not have high frequent transfer of gradients and parameters. For jobs requiring 8 or more GPUs, the cost model in the algorithm frame can be integrated with the locality-based placement so that GPUs on the same nodes or same racks can be prioritized before the cost-based GPU filtering to reduce the large amount of data transfer [8], [26], [61].

Timing constraints. Horus factors in the waiting time in multiple queues job selection, however like many other DL cluster managers, does not consider the timing constraint in terms of completion time in the placement/planning phase [8], [10], [26], [61]. This is because a DL job's convergence rate is often non-linear, depends on hardware/software parameters and does not correlate to the number of iterations [9], so normally DL cluster managers cannot rely on the job's (remaining) execution time, which is used by generic algorithms such as shortest-job-first (SJF) and shortest remaining time first (SRTF), etc or other optimization problem formulation based on timing constraints. Existing scheduling approaches, particularly in HPC and Grid computing, based on the estimation of execution time rely on a strong assumption, that is, workloads are pre-known, e.g., periodic jobs with same datasets, hyperparameter, and model architecture. This assumption does not hold in DL clusters due to constant model evaluation with different datasets [8]. Considering this constraint is, however, beyond the scope of this paper.

4.3 System Implementation

Horus Application Controller is approximately 5k+ lines of code written in Go. The prediction engine is written in Python, and operates as a separate process within the DL system i.e. in Kubernetes, our prediction engine is a pod. Both the prediction engine and our application controller communicate via remote procedure calls (RPC). We leverage the gRPC[6] library as the underlying RPC implementation to perform data serialization and de-serialization during data transfer, allowing our scheduler to request predicted information upon job submission. It is worth noting that our approach requires *no modification* to any underlying DL libraries such as TensorFlow or PyTorch.

Monitoring. Monitoring is the key to application aware optimization [10], [26], [53], [62], [17], [63]. In order to obtain a fine-grained view of the infrastructure, Horus leverages cAdvisor[7], a container monitoring framework. These infrastructure information is then aggregated into a centralized time series database, which our application controller can query and make decision based on the job's historical usage.

Fault tolerance. Using a Network File System (NFS) is often necessary in DL training jobs due to a large amount of training data and memory limitation [9], [64], [26]. In addition to efficient retrieval of training data, a checkpoint file or miscellaneous event files can be persisted across nodes by using NFS. This allows DL job recovery after a failure and, more importantly, enables job preemption due to GPU over-commitment. In Horus, the over-commitment threshold can be configured based on the number of co-located jobs or device memory usage by DL system operators. Apart from failures, stragglers can be present in the cluster and elastic training regime is a practical way of addressing the issue [65]. However, it is not the core focus of this work.

6. https://github.com/grpc/grpc, [01/07/2020]
7. https://github.com/google/cadvisor

5 Experiment Setup

5.1 Hardware and Software

Horus was deployed onto a 12-GPU cluster with each node containing 4 x Nvidia 2080 GPUs, an AMD Ryzen 1920X 12 Core Processor (2 threads per core) with a 10Gb Ethernet network, and 128GB DDR4 memory. Each node was installed with Ubuntu Disco 19.04 and uses Nvidia driver version 430.50. In our experiments, the DL library and CUDA toolkits responsible for DL job instantiation and execution were packaged in a container. Our cluster uses Kubernetes 1.15.2 due to its prominence in the distributed systems community. cAdvisor and DCGM were configured to extract data at 1s and 250ms intervals, respectively, as initial trial runs indicated that these parameters resulted in effective job throughput given our cluster configuration. Our large-scale simulation forms a 128-node cluster with each node containing 8 GPUs, 128 CPU cores and 512GB memory comparable to existing work [53], [26].

5.2 Methodology

We have evaluated Horus using two production traces, one from [26] and another from our collaborator with 398 jobs shown in Fig. 1, using experiments and large-scale trace driven simulation. The main highlights are:

- In testbed cluster experiments, Horus reduces job makespan by up to 30.7% and increases the average cluster GPU utilization by up to 61.5% in comparison to FIFO, Opportunistic Bin Packing and Performance-aware Bin Packing.
- In trace-driven simulation, Horus performance also holds where our approach outperforms other scheduling approaches in makespan, cluster GPU utilization and average job waiting time.

Comparative algorithms. To evaluate the Horus scheduling algorithm described in §4.2.2, we have designed and implemented additional scheduling algorithms for comparison:

- ▷ ***First in First Out (FIFO)***: Emulating slot-based approaches established in big data cluster schedulers such as Kubernetes and YARN. FIFO assigns the incoming DL job onto an idle GPU without job co-location.
- ▷ ***Performance-aware Bin Packing (PAB)***: Leveraging techniques found in [8], [53], PAB schedules DL jobs based on leveraging job characteristics – detected iteration slowdown. The scheduler measures the difference in average steps per second vs. the previous state. After new job placement, if performance drops by 50%, the job is simply re-queued. We allow a warm-up period between 0-60s so all DL jobs achieve stable resource patterns.
- ▷ ***Opportunistic Bin Packing (OBP)***: Assigns DL jobs based on available information – GPU memory availability, via the memory estimation model described in Eq. 8. During job submission time, if a GPU has more memory available

than estimated memory requirement, the scheduler opportunistically schedules jobs to the GPU similar to least-loaded approach.

In the testbed cluster, we conducted experiments with Horus (k=1), as Horus-f (k=3) did not result in significant difference, which instead improved at greater system scale as shown in the large-scale trace driven simulation.

Workload. Experiments were conducted by using a mixture of DL jobs generated from Table 1, as well as new DL model configurations and models such as Transformer, resulting in Horus being exposed to 50% new DL jobs *not* used in predictor training. The selected models and datasets leveraged in our experiments are well established in micro-benchmarking DL cluster schedulers [8], [26]. All algorithms were evaluated with two different workload submission patterns *W-Small* & *W-Large*. W-Small and W-Large use job type distributions between 3 minutes to 2 hours following DL job sizes derived from production systems [26].

Job characteristics. Jobs are characterized as short/long (<=800s or >800s) and light/heavy (<60% or>60% GPU utilization). JCT was controlled by terminating jobs at specified epoch numbers to emulate JCT patterns of production systems. Note that over 50% of total production DL jobs have been shown to require a single GPU in [26], [9], and hence for our test bed experiments, we focused on DL jobs requiring a single GPU for training. Second, our objective is to study changes in workload makespan and JCT due to interference from DL job co-location. Locality–a focus in prior DL cluster schedulers [8], [53], [26]–introduces further JCT heterogeneity, making it difficult to fairly measure potential trade-off gains between resource utilization against JCT increase when co-locating DL jobs.

Experiment runs. Each algorithm scheduled 100 DL jobs for each workload pattern five times each, successfully training a total of 2,500 DL jobs; equivalent to approximately to 7.5 days of continuous DL cluster execution. For test bed experiments, Horus was configured to operate with buffer size $\beta = 15$, and $k = 1$ to demonstrate throughput. In order to evaluate fairness at scale, we conduct the fairness measure in large-scale production trace driven simulation.

Metrics. Algorithm effectiveness was measured using the following metrics: *Cluster GPU Resource Utilization*: average aggregate GPU utilization of all GPUs, *Job Completion Time (JCT)*: the end-to-end completion time for a DL job, commencing from the start of job execution and finishing at job completion. *Workload Makespan*: the total span time to complete all DL jobs from en-queuing through to completion. *Job waiting time*: average job waiting time measured from point at arrival to being scheduled and placed by our scheduler.

6 Experiment Results

6.1 Testbed Cluster

JCT. Fig 8 shows the comparison of average JCT of each scheduling approach. We observe that FIFO achieves the fastest JCT, due to exclusive GPU access, hence having no interference. In contrast, we observe that all co-location algorithms experience JCT slowdown of between 8.2%–21.8%, and 10.3%–30.3% for W-Large and W-Small when compared to FIFO, respectively. All co-location approaches suffer greater performance degradation in W-Small. This is because a higher proportion of short and small jobs allows for a more frequent and varied co-location within GPUs, as opposed to longer and heavier jobs that claim a large portion (or the entire GPU). Although FIFO achieve the fastest average JCT, it has resulted in the largest makespan and lowest GPU Utilization due to longer queuing times.

TABLE 6: DL cluster makespan statistics.

Workload	Algorithm	Avg.(mins)	St. Dev.(mins)	Gain
W-Large	FIFO	306.9	1.15	–
	PAB	277.6	1.72	9.5%
	OBP	238.6	4.9	22.2%
	Horus	**212.8**	**5.04**	**30.7%**
W-Small	FIFO	267.3	1.32	–
	PAB	250.4	2.02	6.3%
	OBP	225.3	5.38	15.7%
	Horus	**204.0**	**8.5**	**23.7%**

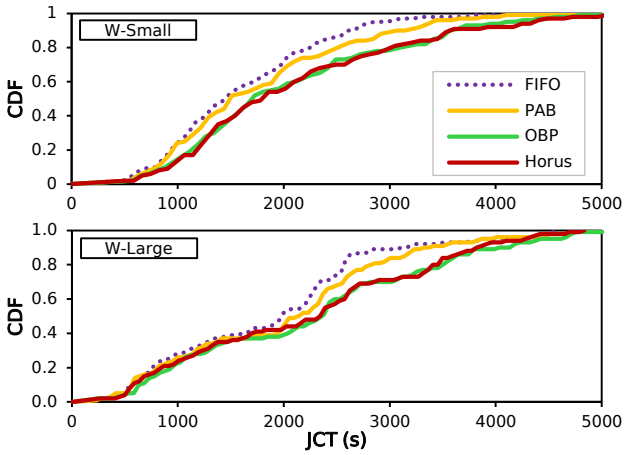


Fig. 8: Job Completion Time (JCT) in testbed cluster experiments.

Makespan. As shown in Table 6, Horus successfully schedules all DL jobs with the lowest makespan of 204 and 212 minute across W-Small and W-Large, respectively, and is equivalent to a 30.7% and 23.7% improvement against FIFO, and a 10.8%–23.3% improvement over OBP and PAB in W-Large. We observe that OBP has the second lowest makespan (238 and 225 minutes), outperforming PAB due to the latter algorithm incurring additional overhead, when determining whether performance were impacted after initial co-location decision. As co-location gains are more effective when workloads are long with diverse utilization, the effectiveness of co-location algorithms (particularly OBP and Horus) is therefore slightly lower in W-Small.

Utilization. Horus is also able to achieve high overall cluster resource utilization in all experiment runs as shown in Fig 9, reflected by an average 69.6% GPU utilization. We observed that in some experiments runs of W-Small, both Horus and OBP can experience up to 30 minutes of DL cluster resource utilization of only 3–5%. This is due to our generated DL jobs may have long epoch times, yet exhibit a low GPU utilization. When omitting such tailing behavior in W-Small, cluster resource usage of OBP and Horus algorithms increases by a further 5.2% and 11.9%, respectively. While OBP and PAB both achieve higher utilization compared to that of FIFO due to their ability to perform co-location,

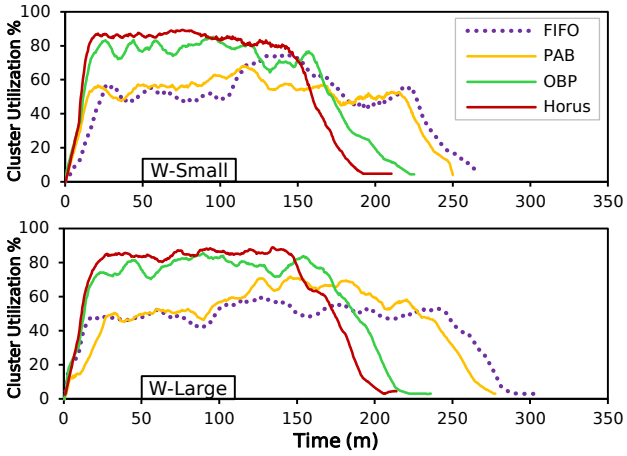
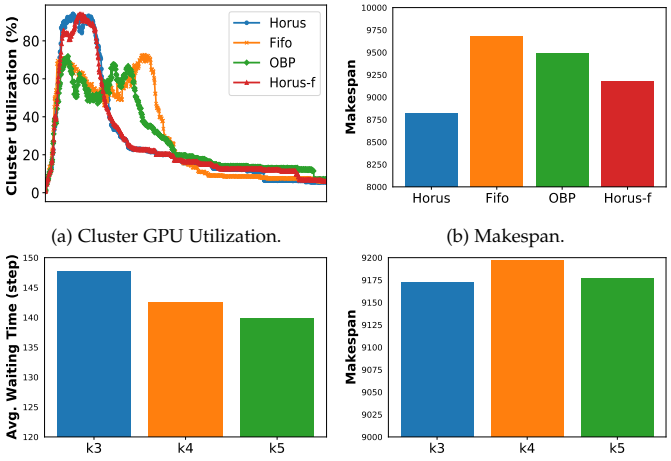


Fig. 9: Average cluster GPU util in testbed cluster experiments.



(a) Cluster GPU Utilization. (b) Makespan.

(c) Avg. waiting time sensitivity. (d) Makespan sensitivity.

Fig. 10: Summary of trace-driven DL cluster simulation at scale.

OBP is able to achieve higher utilization as a result of its rapid scheduling cycle. In contrast, PAB incurs additional scheduling waiting time in order to profile scheduled job's stable performance, this results in a total of n jobs multiply by T_{wait} where T_{wait} is the time it takes for a job to reach stability. Interestingly, Horus's ability to effectively co-locate, achieving higher DL job throughput and GPU Utilization will paradoxically expose to greater interference and consequent JCT slowdown. Horus does however still achieve a lower JCT in comparison to OBP, and when considering our gains to resource utilization and makespan, we view this as an acceptable trade-off.

6.2 Large-scale Trace Driven Simulation

To demonstrate scalability, we evaluated Horus's performance in simulation using 398 jobs from a production trace shown in Fig. 1. Derived from the production trace, our simulation captures both the number of GPUs allocated and execution time required for DL job execution. We executed scheduling algorithms OBP, FIFO, Horus (*k=1*) and Horus-f (*k=3*) configured identically to testbed experiments. Since the simulator does not capture the precise effects of kernel-level characteristics or internal job progress, PAB was not included. We assume interference overheads scale linearly w.r.t. the sum of the jobs GPU Utilization. The simulation runs in super-real time as the full trace duration is a month.

TABLE 7: Job waiting time (steps) for large-scale DL cluster.

Algorithm	Avg.	Med.	St. Dev	Reduction
FIFO	466.2	463.1	327.7	–
OBP	347.8	351.4	248.9	25.4%
Horus	156.5	150.1	130.4	66.4%
Horus-f	147.6	148.5	126.2	68.3%

Improvements. Similar to the testbed experiments, both Horus approaches now utilize the cluster GPU at higher values as shown in Fig. 10a and resulted in fastest makespan up to hundreds of scheduling decisions steps. Compared to Horus, Horus-f has almost the same makespan and utilization, however Horus-f results in a approximately 6% lower median job waiting time as shown in Table 7, showing that Horus-f is desirable when fairness between multiple tenants and jobs category should be considered as it is a common practice for production cluster to be shared by multiple tenants. Overall, both of our Horus scheduling approach utilizes the expensive GPUs effectively both in research scale and large scale cluster, thus enabling faster turn around time, increasing productivity and resource efficiency.

Impact of queue number k. We conduct sensitivity analysis by examining Horus's sensitivity to the configurable number of queues. We evaluate Horus with various values – 3, 4, and 5. The large-scale simulations are ran and averaged over three runs. We observe that the number of queues does not significantly affect Horus as shown in Fig. 10c. When k=5, the waiting time is reduced by $\tilde{1}$% when compared to k=3. However, the makespan performance has degraded slightly when the number of queues is increased as shown in Fig. 10d. There is a known trade-off between fairness and throughput and we view this as an acceptable trade-off.

7 Related Work

Understanding and achieving high resource utilization for heterogeneous workloads—including DL—in cloud computing is an important topic [30], [28], [21], [22], [14], [62], [8], [6], [17], [18], [10].

GPU profiling. Many existing DL systems profile workloads to improve resource-efficiency, these metrics includes training progress [53], communication patterns [26], [66], kernels scheduling patterns [10] and inference execution time [6]. In terms of GPU utilization profiling, Gandiva [8] focuses on time-sharing, leverages online profiling in isolated machines to determine suitable co-location and migration strategies. Thinakaran et al. [17] also perform online profiling on machines in isolation to harvest under-utilized resources. Xu et al. [15] leverage virtualized GPU metrics and vCPU in isolation to propose an approach to predict slowdown from co-located DL workloads. Wang et al [19] obtain DL workload and infrastructure features to determine suitable training regime. Antman [10] also leverages GPU Utilization to first identify jobs that maybe suitable for co-location. Qi et al [67] predict training time via model features, device features, and profiling per-layer execution time.

Interference-Aware resource managers. Studying GPU interference is an established area of research – various solutions have been proposed to mitigate kernel interference in GPU kernel scheduling [13], [14], [16], [21], [22], [32]. These GPU resource schedulers operate between the GPU device driver and the application framework, hence cannot effectively orchestrate and optimize with the global view of the cluster. [14] proposed to profile job's GPU hardware utilization patterns for only a few seconds, this is insufficient for DL jobs that typically require pre-processing of the data when each mini-batch is pulled from the DFS, which can be in region of tens of minutes [10]. For the same reason, various cluster schedulers which reduce performance interference of heterogeneous workloads in cloud environments [29], [30], [31], [58] are not designed to effectively handle DL cluster scheduling. In addition, they do not consider job's locality which is also a key driver in DL job's performance [8], [26], [61]. Thus, differences in hardware architecture, workload, and long queue times [9] drive a need for DL specific cluster schedulers.

DL resource managers. Gandiva [8] is the first co-location enabled DL resource manager, and focuses on improving time-sharing by introducing context-switch mechanism in DL jobs. Tiresias [26], focuses on improving average JCT and job starvation time. It does so by profiling network latency, consolidating distributed DL jobs and implementing a multi-level feedback queue, which adjusts job priorities. Optimus [53] implements a performance predictor model, which at runtime, adjusts the number of required parameter servers or workers. It assumes job convergence is predictable, which in many cases is difficult to ascertain [26]. Recently proposed DL resource manager - Antman [10] introduces modification to the underlying DL framework to allow fine-grained kernel scheduling for co-location to alleviate interference, however still requires profiling at runtime. All of the above DL system resource managers are complimentary to our work, as they focus on addressing various challenges and scheduling objectives. Horus builds upon prior works, and focuses on improving DL system overall makespan and GPU utilization by *automatically* predicting GPU utilization and estimate memory requirements without manually specifying placement decisions, and complements other interference-aware resource managers.

8 Conclusion

In this paper we have presented Horus, a prediction-based interference-aware resource manager for DL systems that achieves high job throughput and increased resource efficiency. Horus avoids the need for lengthy online profiling and one or more dedicated GPUs as favored by existing approaches, by predicting GPU utilization from computation graph features extracted from the DNN model and an offline trained resource predictor. Our approach requires no modifications to DL libraries nor expensive kernel profiling at scheduler run-time. In our analysis we have shown that interference between co-located DL jobs causes on average a JCT slowdown of 19%–42%—comparable to latency increases due associated with distributed learning. Horus is currently integrated into Kubernetes and is suitable for integration into existing DL system resource managers. We have demonstrated that Horus is capable of reducing makespan by up to 23.7–30.7%, achieving a cluster utilization of 69.6%, and 68.3% mean waiting time, representing a considerable increase in DL system resource-efficiency. We also offer Horus-f which lowers median job waiting time and avoids starvation among queued jobs, desirable when fairness between multiple tenants should be considered.

Acknowledgments

This work is supported by the EPSRC (EP/P031617/1).

References

[1] X. Jia *et al.*, "Highly scalable deep learning training system with mixed-precision: Training imagenet in four minutes," *arXiv*, 2018.
[2] E. Chung *et al.*, "Serving dnns in real time at datacenter scale with project brainwave," *IEEE Micro*, vol. 38, no. 2, pp. 8–20, 2018.
[3] Google. (2020) Cloud gpus — google cloud. [Online]. Available: https://cloud.google.com/gpu
[4] Amazon Web Services Inc. (2020) Amazon EC2 P3 – Ideal for Machine Learning and HPC - AWS. [Online]. Available: https://aws.amazon.com/ec2/instance-types/p3/
[5] Microsoft Corporation. (2020) Azure VM sizes - GPU - Azure Virtual Machines. [Online]. Available: https://docs.microsoft.com/en-us/azure/virtual-machines/sizes-gpu
[6] H. Shen *et al.*, "Nexus: a gpu cluster engine for accelerating dnn-based video analysis," in *ACM SOSP*, 2019.
[7] C. Zhang *et al.*, "Mark: Exploiting cloud services for cost-effective, slo-aware machine learning inference serving," in *USENIX ATC*, 2019.
[8] W. Xiao *et al.*, "Gandiva: Introspective cluster scheduling for deep learning," in *USENIX OSDI*, 2018.
[9] M. Jeon *et al.*, "Analysis of large-scale multi-tenant gpu clusters for dnn training workloads," in *USENIX ATC*, 2019.
[10] W. Xiao *et al.*, "Antman: Dynamic scaling on GPU clusters for deep learning," in *USENIX OSDI*, 2020.
[11] K. Hightower, B. Burns, and J. Beda, *Kubernetes: up and running: dive into the future of infrastructure*. " O'Reilly Media, Inc.", 2017.
[12] V. K. Vavilapalli *et al.*, "Apache hadoop yarn: Yet another resource negotiator," in *ACM SoCC*, 2013.
[13] R. Phull *et al.*, "Interference-driven resource management for gpu-based heterogeneous clusters," in *ACM SC*, 2012.
[14] Y. Ukidave *et al.*, "Mystic: Predictive scheduling for gpu based cloud servers using machine learning," in *IEEE IPDPS*, 2016.
[15] X. Xu *et al.*, "Characterization and prediction of performance interference on mediated passthrough gpus for interference-aware scheduler," in *USENIX HotCloud*, 2019.
[16] S. Kato *et al.*, "Timegraph: Gpu scheduling for real-time multi-tasking environments," in *USENIX ATC*, 2011.
[17] P. Thinakaran *et al.*, "Kube-knots: Resource harvesting through dynamic container orchestration in gpu-based datacenters," in *IEEE CLUSTER*, 2019.
[18] T.-A. Yeh *et al.*, "Kubeshare: A framework to manage gpus as first-class and shared resources in container cloud," in *ACM HPDC*, 2020.
[19] M. Wang *et al.*, "Characterizing deep learning training workloads on alibaba-pai," *arXiv*, 2019.
[20] Y. Wang *et al.*, "A systematic methodology for analysis of deep learning hardware and software platforms," in *MLSys*, 2020.
[21] Q. Chen *et al.*, "Prophet: Precise qos prediction on non-preemptive accelerators to improve utilization in warehouse-scale computers," in *ACM ASPLOS*, 2017.
[22] Q. Chen, H. Yang, J. Mars, and L. Tang, "Baymax: Qos awareness and increased utilization for non-preemptive accelerators in warehouse scale computers," *ACM SIGPLAN Notices*, 2016.
[23] G. Yeung *et al.*, "Horus: An interference-aware resource manager for deep learning systems," in *IEEE ICA3PP*, In Press.
[24] M. Tan and Q. V. Le, "Efficientnet: Rethinking model scaling for convolutional neural networks," *arXiv*, 2019.
[25] T. Chen *et al.*, "Tvm: An automated end-to-end optimizing compiler for deep learning," in *USENIX OSDI*, 2018.
[26] J. Gu *et al.*, "Tiresias: A gpu cluster manager for distributed deep learning," in *USENIX NSDI*, 2019.

[27] Z. Guz *et al.*, "Many-core vs. many-thread machines: Stay away from the valley," *IEEE Computer Architecture Letters*, 2009.
[28] J. Mars *et al.*, "Bubble-up: Increasing utilization in modern warehouse scale computers via sensible co-locations," in *IEEE/ACM MICRO*, 2011.
[29] C. Delimitrou and C. Kozyrakis, "Paragon: Qos-aware scheduling for heterogeneous datacenters," *ACM SIGPLAN Notices*, 2013.
[30] C. Delimitrou and C. Kozyrakis, "Quasar: resource-efficient and qos-aware cluster management," *ACM SIGPLAN Notices*, 2014.
[31] D. Novakovic *et al.*, "Deepdive: Transparently identifying and managing performance interference in virtualized environments," in *USENIX ATC*, 2013.
[32] A. Jog *et al.*, "Application-aware memory system for fair and efficient execution of concurrent gpgpu applications," in *ICPS GPGPU-7*. ACM, 2014.
[33] A. Krizhevsky, G. Hinton *et al.*, "Learning multiple layers of features from tiny images," Citeseer, Tech. Rep., 2009.
[34] S. Merity *et al.*, "Pointer sentinel mixture models," *arXiv*, 2016.
[35] A. M. T. (WMT19). Shared task: Machine translation of news. [Online]. Available: http://www.statmt.org/wmt19/translation-task.html
[36] A. G. Howard *et al.*, "Mobilenets: Efficient convolutional neural networks for mobile vision applications," *arXiv*, 2017.
[37] M. Sandler, A. Howard *et al.*, "Mobilenetv2: Inverted residuals and linear bottlenecks," in *IEEE ICCV*, 2018.
[38] A. Howard *et al.*, "Searching for mobilenetv3," in *IEEE ICCV*, 2019.
[39] C. Szegedy *et al.*, "Going deeper with convolutions," in *IEEE CVPR*, 2015.
[40] K. He *et al.*, "Deep residual learning for image recognition," in *IEEE CVPR*, 2016, pp. 770–778.
[41] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," *CoRR*, 2014.
[42] F. N. Iandola *et al.*, "Squeezenet: Alexnet-level accuracy with 50x fewer parameters and¡ 0.5 mb model size," *arXiv*, 2016.
[43] G. Huang *et al.*, "Densely connected convolutional networks," in *IEEE ICCV*, 2017, pp. 4700–4708.
[44] N. Ma *et al.*, "Shufflenet v2: Practical guidelines for efficient cnn architecture design," in *ECCV*, 2018.
[45] M. Tan *et al.*, "Mnasnet: Platform-aware neural architecture search for mobile," in *IEEE CVPR*, 2019.
[46] Y. Chen *et al.*, "Dual path networks," in *NeurIPS*, 2017.
[47] H. Cai, L. Zhu, and S. Han, "Proxylessnas: Direct neural architecture search on target task and hardware," *arXiv*, 2018.
[48] D. Han, J. Kim, and J. Kim, "Deep pyramidal residual networks," in *IEEE CVPR*, 2017.
[49] S. Xie *et al.*, "Aggregated residual transformations for deep neural networks," in *IEEE CVPR*, 2017.
[50] F. A. Gers, J. Schmidhuber, and F. Cummins, "Learning to forget: Continual prediction with lstm," in *ICANN*, 1999.
[51] K. Cho *et al.*, "Learning phrase representations using rnn encoder-decoder for statistical machine translation," *arXiv*, 2014.
[52] A. Paszke *et al.*, "Pytorch: An imperative style, high-performance deep learning library," in *NeurIPS*, 2019.
[53] Y. Peng, Y. Bao *et al.*, "Optimus: an efficient dynamic resource scheduler for deep learning clusters," in *ACM EuroSys*, 2018.
[54] S. Wang *et al.*, "Overlapping communication with computation in parameter server for scalable dl training," *IEEE TPDS*, 2021.
[55] G. Ke *et al.*, "Lightgbm: A highly efficient gradient boosting decision tree," in *NeurIPS*, 2017.
[56] T. Chen and C. Guestrin, "Xgboost: A scalable tree boosting system," in *ACM KDD*, 2016.
[57] A. Liaw, M. Wiener *et al.*, "Classification and regression by randomforest," *R news*, vol. 2, no. 3, pp. 18–22, 2002.
[58] J. Mars and L. Tang, "Whare-map: Heterogeneity in "homogeneous" warehouse-scale computers," in *ACM SIGARCH*, 2013.
[59] H. Wang *et al.*, "S-cda: A smart cloud disk allocation approach in cloud block storage system," in *ACM DAC*, 2020.
[60] A. K. Parekh and R. G. Gallager, "A generalized processor sharing approach to flow control in integrated services networks: the multiple node case," *IEEE ACM Transactions on Networking*, 1993.
[61] L. Luo *et al.*, "Plink: Discovering and exploiting locality for accelerated distributed training on the public cloud," *MLSys*, 2020.
[63] P. Dube, T. Suk, and C. Wang, "Ai gauge: Runtime estimation for deep learning in the cloud," in *SBAC-PAD*, 2019.
[62] R. Yang, C. Hu *et al.*, "Performance-aware speculative resource oversubscription for large-scale clusters," *IEEE TPDS*, 2020.
[64] D. Zhang *et al.*, "Agl: a scalable system for industrial-purpose graph machine learning," in *VLDB Endowment*, 2020.
[65] Y. Chen *et al.*, "Elastic parameter server load distribution in deep learning clusters," in *ACM Symposium on Cloud Computing*, 2020.
[66] Y. Peng *et al.*, "A generic communication scheduler for distributed dnn training acceleration," in *ACM SOSP*, 2019.
[67] H. Qi, E. R. Sparks, and A. Talwalkar, "Paleo: A performance model for deep neural networks," in *ICLR*, 2017.

Gingfung Yeung is a PhD Student in the Evolving Distributed Systems Laboratory (EDS Lab) at Lancaster University. He has industrial experience building Machine Learning systems at scale. His research interests include Machine Learning systems, distributed systems, and resource scheduling.

Damian Borowiec is a PhD student in the Evolving Distributed Systems Laboratory (EDS Lab) at Lancaster University. He received a bachelor degree in Computer Science at the Lancaster University. His research interests include Deep Learning systems, energy-adaptive computing, and Neural Network compilation methods.

Renyu Yang is a Research Fellow at the University of Leeds. He previously worked in Alibaba Group China and Edgetic Ltd. UK, having industrial experience in building large-scale resource scheduling systems. His research interests include reliable resource management, distributed systems and applied machine learning. He is a member of IEEE.

Adrian Friday is a Professor of Computing and Sustainability at Lancaster University. He is interested in the role of computational systems in helping us understand the energy and carbon footprint of socio-technical systems, and in finding more sustainable ways of living. His current work focuses on the role of energy data in smart cities, and using statistical and ML techniques to identify new opportunities for energy savings.

Richard Harper is a Professor of Computer Science and Co-Director for the Institute of Social Futures (ISF) at Lancaster University. Richard Harper has written 18 books and collections, including The Myth of the Paperless Office (2003), Texture: human expression in the age of communications overload (2010) and Skyping the Family (2019).

Peter Garraghan is a Lecturer (Assistant Professor) in Distributed Systems and EPSRC Fellow at Lancaster University. He is the leader of the Evolving Distributed Systems Laboratory (EDS Lab). Peter has industrial experience building production distributed systems at scale. His research interests include Machine Learning systems, Cloud computing, green computing, resource management, and system security.