Horus: An Interference-Aware Resource Manager for Deep Learning Systems

Gingfung Yeung[1], Damian Borowiec[1], Renyu Yang[2(✉)], Adrian Friday[1], Richard Harper[1], and Peter Garraghan[1]

[1] School of Computing and Communications, Lancaster University, Lancaster, UK
{g.yeung1,d.borowiec,a.friday,r.harper,p.garraghan}@lancaster.ac.uk
[2] School of Computing, University of Leeds, Leeds, UK
r.yang1@leeds.ac.uk

Abstract. Deep Learning (DL) models are deployed as jobs within machines containing GPUs. These DL systems - ranging from a singular GPU device to machine clusters - require state-of-the-art resource management to increase resource utilization and job throughput. While it has been identified that co-location - multiple jobs co-located within the same GPU - is an effective means to achieve this, such co-location incurs performance interference that directly debilitates DL training and inference performance. Existing approaches to mitigate interference require resource intensive and time consuming kernel profiling ill-suited for runtime scheduling decisions. Current DL system resource management are not designed to deal with these problems. This paper proposes Horus, an interference-aware resource manager for DL systems. Instead of leveraging expensive kernel-profiling, our approach estimates job resource utilization and co-location patterns to determine effective DL job placement to minimize likelihood of interference, as well as improve system resource utilization and makespan. Our analysis shows that interference cause up to 3.2x DL job slowdown. We integrated our approach within the Kubernetes resource manager, and conduct experiments in a DL cluster by training 2,500 DL jobs using 13 different models types. Results demonstrate that Horus is able to outperform other DL resource managers by up to 61.5% for resource utilization and 33.6% for makespan.

Keywords: Machine learning systems · Performance interference · Deep Learning · GPU scheduling · Cluster resource management

1 Introduction

Deep Learning (DL) is an increasingly important type of machine learning algorithm with significant potential for touching many aspects of society [12]. The rapid growth in the number of DL practitioners and the data they require in their computations has created a necessity for both individually powerful, as well as large-scale clusters of machines equipped with GPUs - specialized hardware accelerators - to facilitate the vast amounts of computation DL entails at

© Springer Nature Switzerland AG 2020
M. Qiu (Ed.): ICA3PP 2020, LNCS 12453, pp. 492–508, 2020.
https://doi.org/10.1007/978-3-030-60239-0_33

reduced training time. These systems, which we refer to as *DL systems*, use resource management frameworks to perform DL scheduling and job placement (i.e allocation of DL jobs onto GPUs). An important goal for DL systems is to maximize the effective utilization of these expensive resources through minimizing *makespan*, *job waiting*, and *job completion time (JCT).*

DL systems, particularly clusters, experience issues associated with GPU underutilization and long queuing times [19]. One cause of such underutilization is that DL resource managers disallow *co-location* of multiple DL jobs within the same GPU [13,25]; a characteristic shared within other resource managers such as Kubernetes and Yarn that were originally designed for CPU-based workloads [16,35]. Instead, the majority of DL resource managers focus on reducing network latency and locality [3,13,25]. This inability to co-locate DL jobs within the same GPU results in reduced resource utilization, longer queuing times and reduced cost efficiency within DL systems.

Recent DL resource managers have been proposed that make placement decisions by consolidating DL jobs onto fewer machines to minimize workload and JCT [3,13,37]. However, while DL resource managers now exist that allow for co-location [29,37], there has been little attention drawn to the *performance interference* which arises between multiple DL jobs training within the same GPU. Performance interference (which we refer to as *interference*), results in slower training step time and overall epoch time. Previous work has demonstrated the existence of interference within DL systems, resulting in an 18% JCT slowdown [37]. Furthermore, as no existing DL resource manager considers the impact of interference in DL job co-location decisions, this can lead to poor placement of unsuitable jobs resulting in a higher makespan, increase in JCT, job eviction and job failures from GPU out-of-memory (OOM) errors [19].

We propose Horus, a DL system resource manager that maximises resource utilization and minimizes makespan whilst attempting to reduce JCT performance degradation by anticipating interference due to co-location. By leveraging DL model application features ascertained from [40], Horus is able to estimate the GPU utilization of DL jobs *prior to execution*, and make better placement decisions to determine suitable co-location combinations with the lowest interference. Our approach avoids the need to profile kernel patterns [6,26], modification of the GPU thread-block scheduler (commonly proprietary to GPU hardware manufacturers), and extensive online profiling of job execution in an isolated GPU at scheduler runtime; all of which are expensive and time consuming processes in terms of system development and job placement. Our core contributions are: **(1) Analysis of DL job co-location patterns**. We empirically measured 276 unique combinations of interference patterns from co-locating 13 prominent types of DL jobs comprising both vision and language models (Sect. 5.2). Our results demonstrate that co-locating DL jobs with high GPU utilization requirements leads to a 1.5X–3.2X JCT increase stemming from interference. Moreover, we observe that interference patterns significantly vary between different DL job combinations, and that equivalent GPU utilization can exhibit dissimilar JCT degradation patterns.

(2) Co-location DL resource manager. Via leveraging GPU utilization estimation, we construct a cost-based best-fit-decreasing model that determines suitable placement for DL job co-location (Sect. 4). Horus was integrated into Kuberentes [16] - an open source cluster manager, and was evaluated by submitting 2,500 DL jobs from different application domains into a GPU cluster under different workload patterns (Sect. 5). Results demonstrate that Horus achieves high resource utilization and scheduler performance, and outperforms various cluster resource managers - including other co-location approaches [35,37]) - with improvements up to 61.5% and 33.6% for resource utilization and makespan.

2 Background

2.1 Deep Learning

Deep Learning is a type of machine learning algorithm based on neural networks. DL models are formed by *deep neural networks* (DNNs), consisting of input, hidden, and output layers [12] and have two phases: *training* and *inference*. DL model training requires significant volumes of data [15] to iteratively minimize an error objective. Larger numbers of layers and units per layer results in higher number of floating point operations (FLOPs) to execute [32]. GPUs are frequently used to accelerate DL model training due to their ability to rapidly perform FLOPs execution using thousands of processing cores. For example, Nvidia GPUs define abstraction over a group of cores as streaming multiprocessors (SMs), which are used to execute GPU kernels. A complex DL model (i.e expressed by model depth and width) will result in greater number of FLOPs, and thus a higher GPU utilization in comparison to simpler DL models that leverage the same batch size and GPU architecture [1].

2.2 Deep Learning Resource Managers

Due to the growing number and scale of DL jobs that require training on TB-scale data, researchers and businesses leverage *DL systems*: powerful machines or clusters of machines to accelerate training. Users submit jobs to a DL system via a web portal or command line interface with specified job configurations (e.g. batch size, model, dataset) comprising one or more tasks that execute within a container. DL jobs are assigned resources and allocated onto machines through use of a *resource manager* to increase system resource efficiency to satisfy a specified Service Level Agreement (SLA). Existing DL system resource managers[1] have focused on a specific sub-set of objectives including minimizing makespan, JCT, as well as maximizing system resource utilization and energy-efficiency [4].

Recent studies of production DL systems have identified several challenges: low utilization of system resources reflected by an average GPU utilization of

[1] Which we refer to as DL resource managers.

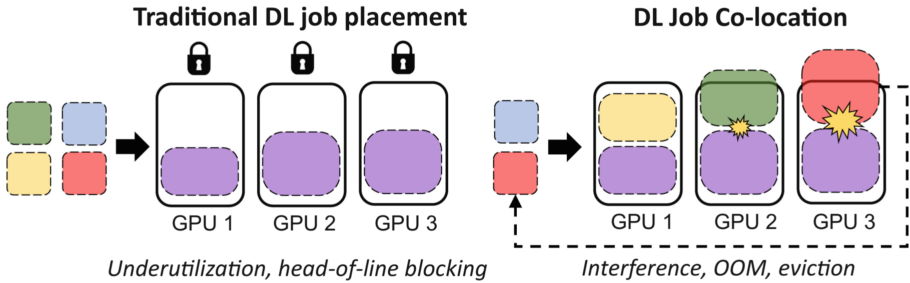


Fig. 1. Difference between traditional vs. DL resource managers: DL job co-location manifesting interference.

52% [19], and long queuing time for DL jobs between 4000 s–8000 s due to head-of-line blocking [13]. These challenges are exacerbated by DL systems leveraging non-preemptive traditional schedulers [16,35] that require DL jobs to hold exclusive access to a given GPU. This is particularly problematic in the context of schedulers due to its negative impact upon job throughput and makespan, system availability, and cost efficiency.

Improving upon established approaches, recent DL resource managers have been designed to address challenges of under-utilization and long queue times via improvements to network locality and bandwidth [3,13,25]. Another approach demonstrated to be effective for DL resource managers is enabling the *co-location* of DL jobs within the same GPU to execute simultaneously, improving overall system resource utilization [29,37]. However, few DL resource managers consider or capture the drawbacks of co-location when performing DL job placement decisions, including the manifestation of interference that might result (Fig. 1).

2.3 Deep Learning Interference

Interference occurs when multiple processes compete for limited resources within the same machine [8,23]. Interference of DL jobs co-located within the same GPU has been shown to result in 18% JCT degradation [37]. This is problematic when considering that DL jobs may train in the region of hours to days. Hence, in order for DL systems to fully exploit co-location, DL resource managers should consider the effects of interference when performing DL job placement.

Profiling DL job resource usage (notably GPU utilization) allows DL resource managers to minimize interference resultant from co-location. GPU interference differs from that of CPU interference because of their processor architectures. GPUs use a thread-block scheduler to schedule compute or memory intensive kernels to streaming multiprocessors, and leverage *single instruction multiple data* parallelism on many cores, whereas CPU uses *multiple instruction multiple data* parallelism on fewer cores. Precisely calculating interference for different co-located DL job combinations is challenging due to the diversity in model types and kernels that are implemented in different DL libraries [2].

Table 1. Studied DL models in Vision (Cifar10 [21]) and NLP (WikiText-2 [24], News-Commentary v14-en-zh [36]

Model	Type	Batch size
MobileNetV2 [27], GoogLeNet [31], ResNet [15], VGGNet [30], DenseNet [17], SqueezeNetV1 [18], ShuffleNetV2 [22], ResNeXt [38], MNASNet [33], PyramidNet [14], DualPathNetwork [7]	CNN	64, 128, 256
LSTM [11], Transformer [34]	RNN	16, 32, 64

While there exist several GPU resource managers that minimize interference [6,26], such approaches require extensive online profiling of job kernel access patterns at scheduling runtime. This is challenging as profiling DL job resource usage (e.g. instruction per cycle, DRAM throughput, compute efficiency) to infer interference by creating suitable performance profiles may extend DL job training within the regions of minutes to hours. Additionally it must be performed for every new DL model type submitted into the system. This results in considerable resource overhead when using profiling tools such as `nvprof` and `nv-nsight-cu-cli`, as well as increased DL system makespan. We believe an alternative approach is to understand how DL model types and model configurations result in varying DL job resource usage patterns, in order to infer interference for different co-location combinations. This would allow for DL resource managers to co-locate DL jobs more effectively through considering the impact of interference in placement decisions.

3 Deep Learning Interference Study

This section presents our analysis of different interference profiles for co-located DL jobs. While there have been prior studies into GPU interference [6,39], the majority of works use few or relatively simple DL models types and configurations (i.e. LeNet, Multilayer perceptron, MNIST). We conduct a micro-benchmark of different co-location combinations with heterogeneous configurations of prominent DL model types, and study their influence upon resource utilization and interference profiles.

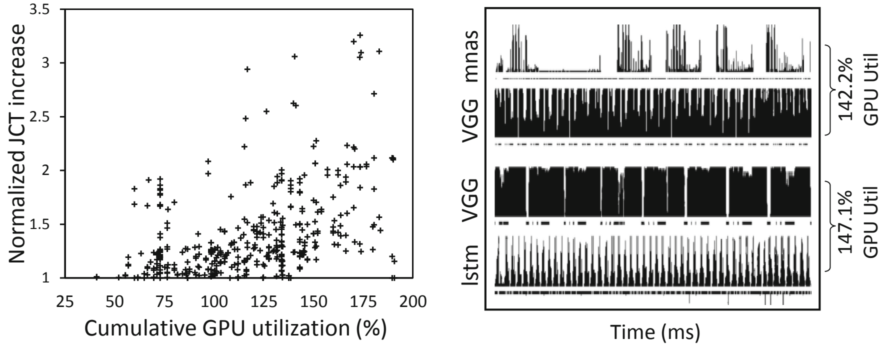
3.1 Setup

A wide variety of DL jobs were deployed within a DL system (4 x Nvidia Geforce 1080, Intel i7-6850k, Nvidia Docker 2, CUDA Toolkit 10.0), and using the DL library frameworks AllenNLP [10] and Pytorch 1.1. Leveraging methods established within prior studies of GPU interference [6,39], micro-benchmarking was conducted by co-locating paired combinations of DL jobs within the same GPU device, and then measuring the corresponding JCT performance degradation

T_{deg} from interference during DL job training. Performance degradation is calculated as

$$T_{deg} = \frac{|T_{colo} - T_{solo}|}{T_{solo}} \tag{1}$$

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 training in isolation.



(a) Interference profiles from different DL job co-location combinations

(b) Kernel co-located jobs patterns with equiv. GPU utilization

Fig. 2. GPU utilization interference patterns

The DL job micro-benchmark comprises 13 unique DL model architectures including both convolution neural networks (CNNs) and recurrent neural networks (RNNs) from computer vision domains and natural language processing as shown in Table 1. Each model is then further modified with different model configurations such as residual blocks, projection dimensions, and scale parameters. DL models were selected due to their usage within previous DL resource managers [13,25,37] and prominence in the machine learning community. This provided a total of 24 unique DL model configurations, and 276 unique co-location combinations (300 when including DL jobs in isolation) for profiling. Each DL job is trained for a fixed set of five epochs to capture a stable performance profile. Analysis metrics were collected from monitoring the DL system, using `nvidia-smi` to collect statistics for GPU utilization, PCIe bandwidth and GPU memory usage.

3.2 Analysis

We found that co-located DL jobs that require high GPU utilization results in greater JCT slowdown from higher interference. For example, co-locating two VGG19 models (each requiring 90%+ utilization in isolation) results in over a 3.2X JCT increase. Figure 2a shows that for all co-located job combinations,

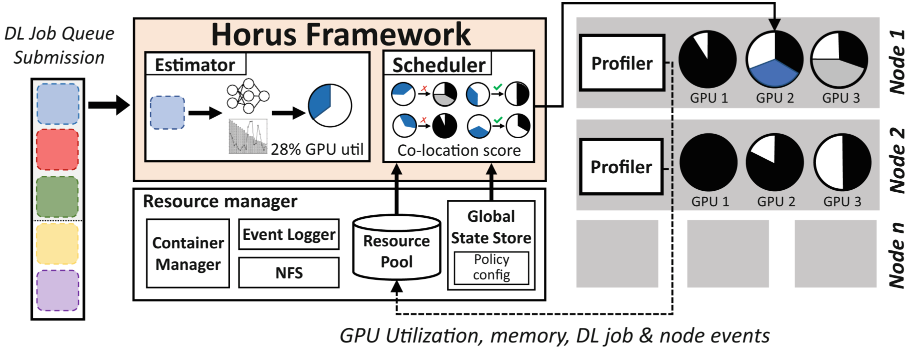


Fig. 3. Horus scheduler framework overview

GPU overcommitment (i.e. the cumulative GPU utilization requirement greater than 100% of a GPU device) results in an average JCT increase of 42%. In contrast, pairs of co-located DL jobs that individually require less than 50% utilization are less likely to experience overcommitment, and, as a result, can be co-located with similar sized jobs with minimal interference JCT increases of 1–10%. Moreover, we observed that the degree of JCT increase stemming from interference varies quite substantially, even for jobs with similar DL job utilization levels in isolation. This phenomena can be seen in kernel access patterns shown in Fig. 2b, whereby co-locating VGG with either LSTM (1 layer, hidden dimensions 128) or MNASNet (Depth 1.3) results in an equivalent GPU utilization of approximately 147%, however results in a 1.3X and 3.03X JCT increase. The reason for this behaviour is due to convolution kernels being intrinsically more compute intensive in contrast to GEMM kernels, as well as different memory transfer and compute patterns when contending for resources [6].

4 The Horus Framework

4.1 Overview

Horus is designed to operate within DL resource managers that leverage GPUs, and comprises two main components as shown in Fig. 3: the *Resource Estimator*, and *Resource Scheduler*. At submission time, the estimator calculates the GPU utilization of an executing, or incoming DL job, ascertained via online metric collection or prediction [40] (detailed in Sect. 4.2). The scheduler assigns DL jobs to GPUs by ranking their suitability to support co-location. Our approach attempts to greedily maximize GPU utilization to minimize makespan whilst attempting to avoid placement decisions that lead to severe interference causing JCT slowdown (detailed in Sect. 4.3).

Our resource manager can be deployed within a single DL system, or within a cluster integrated within existing resource managers such as Kubernetes [16]. In the context of clusters, Horus uses a shared-state, centralized architecture

due to its suitability to handle long jobs (an common characteristic of DL job training), as well as providing a view of overall global cluster for high quality scheduling decisions [9,28]. A centralized cluster view is maintained through a shared centralised repository, with monitoring agents deployed in each machine reporting application and system utilization metrics for placement decisions.

4.2 Resource Estimator

Estimation of Expected GPU Utilization: While our framework does not modify any underlying DL libraries to function, it does require DL model utilization to be provided. Such utilization patterns can be provided via execution, or prediction. For Horus we have leveraged the prediction technique from Yeung et al. [40] to avoid executing each and individual incoming job j, which operates by traversing the computation graph, extracting model features and eventually estimating its GPU utilization, i.e. $\mathbb{E}(GUtil_j)$. Such features are well known and frequently modified by Machine Learning researchers and developers [27,32,33], hence it is relatively straight forward for practitioners to manually extract, or automatically collect such metrics via graph analysis tools such as TensorFlow profiler[2] and TorchScript[3].

Estimation of Expected GPU Memory: The only exception for straight forward metric collection the total job memory size (MiB) due to initialization and optimization of individual DL libraries. However, it is possible to estimate the minimum expected memory usage in bytes by considering the following four factors involved in both forward M^f and backward passes M^b [12]: (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 can be expressed as:

$$\mathbb{E}(GMem_j) = \mathbb{E}(M_j^f) + \mathbb{E}(M_j^b) + \delta = (B * A + P) + B * G + \delta \qquad (2)$$

The expectation of both GPU utilization and GPU memory will be used for node capacity check in the scheduler in case of tackling an incoming job.

4.3 Resource Scheduler

This section describes our scheduling approach to effectively co-locate DL jobs and handle potential placement issues from interference. We observe that in order to maximize GPU utilization, it is necessary to allow co-location of DL jobs onto the same GPU. Gandiva [37] employs a random trial-and-error strategy to co-locate DL jobs. In their approach, by monitoring the job in isolation and application throughput, a job is killed or migrated to another node randomly using an undefined threshold value and time period. In such an approach, it is

[2] https://github.com/tensorflow/tensorflow/tree/master/tensorflow/core/profiler.
[3] https://pytorch.org/docs/stable/jit.html.

possible for random job migration to be allocated with another incompatible job leading to equal or greater performance slowdown.

To alleviate the performance degradation stemming from co-location interference, the core of resource scheduling is to understand the compute resource requirement *prior* to job execution and incur as less overhead (cost) as possible. We design the cost to reflect the selection preference of a node mainly considering GPU memory usage and GPU utilization, respectively. We can therefore determine the most suitable placement based on per-node cost inference.

Cost Inference: We break down the cost of scheduling job j onto an individual GPU k (denoted by $j \rightarrow k$):

$$Cost_{j \rightarrow k} = C^{GMem}_{j \rightarrow k} + C^{GUtil}_{j \rightarrow k} + \epsilon^{jobType}_{j} \tag{3}$$

We add up the incurred cost of running job j regarding GPU memory usage and GPU utilization increase, followed by a calibration ϵ due to DL jobs type.

In particular, the cost of GPU memory $C^{GMem}_{j \rightarrow k}$ is referred to as a weighted proportion of GPU memory usage (Eq. 4), in light of the key implication – higher current GPU memory usage causes a higher chance of OOM and JCT slowdown.

$$C^{GMem}_{j \rightarrow k} = \omega * \frac{GMem^{used}_{k}}{GMem^{total}_{k}} \tag{4}$$

where $GMem_{total}$ is the total GPU memory of the device and ω is used to customize and indicate the performance impact.

As there exists a relationship between increased GPU utilization of co-located DL jobs and JCT slowdown (Sect. 3.2), we penalize the combinations of co-located DL jobs when over-commitment manifests – the total forthcoming GPU utilization (i.e., current GPU utilization $GUtil^{curr}_{k}$ and the estimated increment $\mathbb{E}(GUtil_j)$) exceeds 100%. Likewise, we add a penalty hyperparameter $\phi \in [1, 2]$ to tweak the slowdown impact due to GPU over-commitment:

$$\begin{aligned} GUtil_{j \rightarrow k} &= GUtil^{curr}_{k} + \mathbb{E}(GUtil_j) \\ C^{GUtil}_{j \rightarrow k} &= \begin{cases} \phi * |GUtil_{j \rightarrow k} - 100|, & \text{if } GUtil_{j \rightarrow k} > 100 \\ 100 - GUtil_{j \rightarrow k}, & \text{if } GUtil_{j \rightarrow k} \leq 100 \end{cases} \end{aligned} \tag{5}$$

The implication behind this *piecewise* cost setup is to pack a job so that the host node can approach to 100% GPU utilization without resource over-commitment. For instance, higher remaining GPU capacity results in a higher cost. On the other hand, once over-commitment occurs, the cost will be increasingly augmented, which indicates a reduced scheduling probability of the node, considering job's performance.

Similarly, as observed in Fig. 2b, different DL model architectures exhibit various degrees of JCT slowdown from interference, thus a *niceness* hyperparameter ν is added to accommodate this behaviour (Eq. 6. Our scheduling approach will favour a particular DL job types based on ν selection.

$$\epsilon^{jobType}_{j} = \begin{cases} \nu, & \text{if } C_{jobtype} \text{ is CNN} \\ 0, & \text{otherwise} \end{cases} \tag{6}$$

Algorithm 1. Best Fit Decreasing Job Scheduling Algorithm

```
Input: (J, S) // k jobs in the queue and current cluster state
Output: (scheduleStatus, NodeBindingResult res)
 1: 𝒥 ← DescendSort(J) // a job collection via descend sort by jobs' GPU util
 2: res ← dict()
 3: for j in 𝒥 do
 4:     if hasAllocatableResources(j, S) then
 5:         𝒩 ← getFeasibleNodes(j, S) //capacity check(CPUs, Mems, GPU Mems)
 6:         𝒢 ← getAllGPUs(𝒩)
 7:         λ ← j.requestedGPU; σ ← GetMinRequiredNodeNum(j),
 8:         if LEN(𝒩) < σ then
 9:             continue
10:         𝒞_{𝒢→j} ← GetCosts(𝒢, j) // use Eq. 3 to calculate cost for each GPU
11:         𝒢⁺ ← AscendSort(𝒞_{𝒢→j}).topK(λ)
12:         res ← Put(j, 𝒢⁺)
13: if Len(res) > 0 then
14:     BindNodes(res, j)
15:     return ScheduleResults.SCHEDULED, res
16: else
17:     return ScheduleResults.EMPTY, Nil
```

Cost-Based Best-Fit-Decreasing Job Scheduling: Algorithm 1 outlines our scheduling solution. At each scheduling time, the scheduler will collect the current DL system state, and fetch job collection J from the queue. We adopt a best-fit-decrease like algorithm to prioritize jobs with larger GPU requests, avoiding their long-time starvation. To do so, we firstly sort the job collection $\mathcal{J}$ according to the pertaining GPU utilization (Line 1). We then iteratively attempt to find and bind resources for each job. Specifically, we firstly check the resource capacity and filter out available candidate nodes $\mathcal{N}$ that can satisfy all requirements of job j in terms of CPU, memory and GPU memory (Lines 4–9). GPU collection $\mathcal{G}$ is obtained from the pertaining candidate nodes. For instance, the GPU memory requirement is inferred by using Eq. 2 discussed in Sect. 4.2. We can further assess the incurred cost stemming from running the job j on each GPU of $\mathcal{G}$, separately, via the cost estimation in Eq. 3 (Line 10). To reduce possible interference, the best fit is to select the nodes with minimized impact in case of placing j. Hence the scheduler prefers nodes that host the selected GPUs ($\mathcal{G}^+$) with λ least costs (Lines 11–12), and finalizes placement decisions by sending binding requests to the scheduler (Lines 13–17).

Job Failover and Rescheduling: In some scenarios, 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 resulting from incorrect 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. The scheduler will then update the DL

job request with necessary GPU memory requirements, where GPU memory must be equal or greater than the job to be included in consideration.

In a worst case scenario whereby each DL job already fully utilizes GPU resources (i.e. no memory available for packing), our algorithm will pend waiting jobs until existing jobs terminate and release resources and so act similarly to traditional DL resource managers. However, we can leverage other priority or time-sharing primitives to reduce the waiting time.

Complexity Analysis: It is worth noting that at each scheduling cycle, the scheduler re-considers all available GPUs and nodes, given resource requirements are satisfied, including the number of GPUs, and the available GPU memory against the least expected memory calculated in Eq. 4. The time complexity of the capicity check procedure (Lines 4–5) is $\mathcal{O}(kNG)$ where k denotes the size of pending jobs, while N and G represent the number of nodes and GPUs in the DL system, respectively. In addition to $\mathcal{O}(NlogN)$ used for sorting and asymptotically $\mathcal{O}(N)$ in top-K selection, the overall complexity remains $\mathcal{O}(kNG)$. The scheduling can therefore scale well with the increment of either node or GPU scaling-out. Since our scheduler uses a placement algorithm for co-location is greedy, we do not guarantee algorithm optimality. Nevertheless, we find it is less invasive and less time consuming in improving scheduling effectiveness.

5 Evaluation

5.1 Experiment Setup

System: Horus was deployed onto a 12-GPU cluster with each node containing 4 x Nvidia RTX 2080 Ti GPUs, an AMD Ryzen 1920X 12 Core Processor (2 threads per core) with 10Gb Ethernet network, and 128 GB DDR4 memory. Each node was installed with Ubuntu Disco 19.04 and Nvidia driver 430.50. In our experiments, the DL libraries (AllenNLP [10], Pytorch 1.1) and CUDA toolkits responsible for DL job instantiation and execution were stored in a Docker container. Our cluster uses the Kubernetes 1.15.2 resource management framework due to its prominence within the resource management community. cAdvisor and DCGM were configured to extract data at 1 s and 250 ms intervals, respectively, as initial trial runs indicated that these parameters resulted in effective job throughput given our cluster configuration.

Comparative Algorithms: To evaluate the Horus scheduling co-location algorithm described in Sect. 4.3, we have designed and implemented two additional scheduling algorithms for comparison:

- ▷ ***First in First Out (FIFO)***: Emulating slot-based approaches established in big data cluster schedulers such as Kubernetes [16] and YARN [35], FIFO assigns the next incoming DL job onto an idle GPU without job co-location. As threshold values are not defined in prior work, the timing period was set at 0–60 s so that all DL jobs achieve stable performance patterns, and we configure the performance threshold to 50% informed by interference patterns (Fig. 2a).

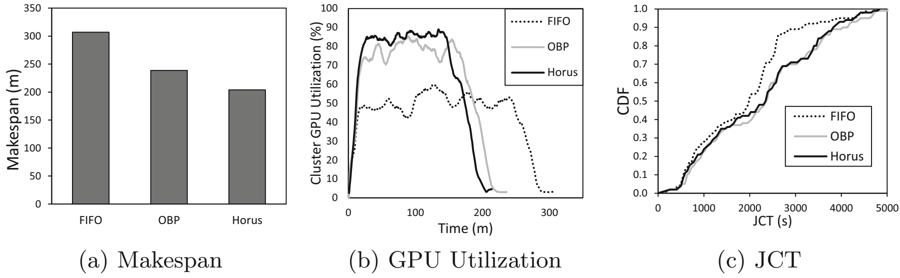
▷ ***Opportunistic Bin Packing (OBP)***: Assigns DL jobs based on GPU memory availability. The algorithm assigns and co-locates DL jobs based on estimated memory requirements prior to submission via exploiting our memory estimation model described in Eq. 4. During job submission time, if a GPU with higher memory is available than estimated memory, then the scheduler will opportunistically allocate a schedule for that job to that GPU.

With the exception of FIFO, preemption was enabled for all algorithms. Our preemption strategy is triggered when a GPU is overcommitted (threshold defined by operator) or an idle GPU. This preemption allows previously co-located DL jobs experiencing interference to be rescheduled to another GPU. Horus have configured to operate job collection size $J = 15$ based on initial experiment runs to provide a sufficient number of candidate DL jobs for placement.

Workload: Experiments were conducted using a mixture of DL job types generated from Table 1, as well as new DL model configurations and model types (LSTM, Transformer), resulting in Horus being exposed to approximately 50% new DL jobs not used in predictor training. Selected models and datasets leveraged in our experiments are well established in micro-benchmarking DL resource managers [13,25,37]. Submitted DL jobs use a distribution between 3 min to 2 h following DL job sizes derived from JCT of production systems [13]. Jobs are characterized as short/long ($<$800 s or $\geq$800 s) and light/heavy ($<$60% or $\geq$60% GPU utilization). JCT was controlled by terminating jobs at specified epoch numbers to emulate JCT patterns of production systems, as well as train sufficient DL jobs in a reasonable time frame. For our experiments we focused on DL jobs requiring a single GPU for training, following established practice from other co-location DL schedulers [29]. This is because between 50%–86% of total production DL jobs have been shown to require a single GPU [5,13,19] and hence we have attempted to capture a broad spectrum of different job and model types. Our objective is to study changes in workload makespan and JCT due to interference from DL job co-location. Furthermore, locality—a key focus within prior DL cluster schedulers [13,25,37]—introduces a non-intuitive dimension of JCT heterogeneity even for jobs running in isolation, making it difficult to fairly measure potential trade-off gains between resource utilization against JCT increase when co-locating DL jobs.

Metrics: Algorithm effectiveness was measured using the following metrics: *Cluster GPU Resource Utilization*: GPU utilization of all devices, *Job Completion Time (JCT)*: 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. Moreover, we have also measured parameters for general cluster resource utilization (CPU, memory, disk).

Experiment Runs: Each algorithm scheduled 100 DL jobs for each workload pattern five times each, successfully training a total of 1,500 DL jobs; equivalent to approximately 66 days of DL system GPU computation.



(a) Makespan (b) GPU Utilization (c) JCT

Fig. 4. Experiment results for comparing DL resource manager operation.

5.2 Results

Makespan: In all experiment runs, Horus was able to successfully schedule all DL jobs with the lowest makespan as shown in Fig. 4a. This is demonstrated by a makespan of 204 min, and is equivalent to a 33.6% improvement against FIFO, and a 22.2% improvement over OBP as shown in Table 2. The reason for a lower makespan is due to Horus being able to perform better placement decisions for co-locating DL jobs, by leveraging our GPU utilization estimator to avoid underutilization and OOM errors from overcommitment. We observe that OBP has the second lowest makespan for all experiments. FIFO has the highest makespan at 306.9 min due to longer queueing times for DL jobs waiting to acquire exclusive access to an idle GPU for training.

Table 2. Workload makespan, GPU utilization and JCT statistics.

Objective	Algorithm	Avg.	St. dev	Change
Makespan (mins)	FIFO	306.9	1.15	-
	OBP	238.6	4.9	22.2%
	Horus	**204.0**	**8.5**	**33.6%**
Utilization (%)	FIFO	43.1	16.7	-
	OBP	59.7	27.2	38.5%
	Horus	**69.6**	**26.9**	**61.5%**
JCT (s)	FIFO	1869.7	1054.3	-
	OBP	2277.5	1293.1	21.8%
	Horus	**2193.8**	**1307.3**	**17.3%**

Utilization: Horus is able to achieve high overall cluster resource utilization in all experiment runs as shown in Table 2 and Fig. 4b, reflected by an average 69% utilization in comparison to FIFO (43%) and OBP (60%). This is resultant of the Horus algorithm determining better co-location combinations for DL job placement to maximize GPU utilization from predicted memory and utilization

requirements of DL jobs described in Sect. 4.3. While OBP achieve relatively high utilization compared to FIFO due their ability to perform co-location, OBP is able to achieve higher utilization as a result of its rapid scheduling cycle.

JCT: Figure 4c and Table 2 (JCT) shows the average JCT for DL jobs. We observe that FIFO achieves the fastest JCT at 1869.7 s, due to DL jobs acquiring exclusive GPU access disallowing co-location and thusly no interference. In contrast, we observe that all co-location algorithms experience JCT slowdown between 17.3%–21.8%. A note of particularly interest is Horus's ability to effectively co-locate and reduce makespan to achieve higher DL job throughput will paradoxically expose the DL cluster 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.

Scalability: Horus has been evaluated through empirical means in a DL cluster of comparable scale found in recent works [25,29]. As discussed in Sect. 4.2, our algorithm complexity is a linear combination between the nodes and GPUs. Moreover, Horus was evaluated using established communication mechanisms in the Kubernetes framework for orchestrating job deployments and file system mounting, which has been demonstrated to scale to thousands of machines [16].

6 Related Work

DL Resource Managers: Gandiva [37] focuses primarily on improving the time-sharing, by enabling DL job *context-switching*, and extracting job throughput (e.g. minibatch per second) thus allowing 'random-and-trial' job co-location placement and eviction upon performance slowdown. Tiresias [13], 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 [25] 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 [13]. All of the above DL cluster schedulers are complimentary to our work as they focus on addressing various challenges and scheduling objectives, related to locality, time-sharing and average JCT. Horus focuses on DL workload makespan and GPU utilization, as well as making placement decisions based on interference between co-located DL jobs in GPUs.

Interference-Aware Resource Managers: The study of GPU interference is an established area [6,26], whereby approaches leverage heavy static profiling of GPU kernel access patterns within isolated machines to classify the workload types at the job submission time, identifying suitable placements. There also exist various cluster schedulers which reduce performance interference of heterogeneous CPU workloads [8,9,20]. As discussed in Sect. 2, these cluster schedulers are not designed to effectively handle GPU scheduling, particularly DL clusters due to differences in hardware, workload, and long queuing times

[19]. Horus builds upon these ideas, proposing a prediction discussed in Sect. 4.2 which complements other GPU interference-aware resource managers [29,39], by using lightweight profiling to characterize DL job utilization patterns.

GPU Interference Analysis: Interference analysis and fine-grained GPU kernel scheduling is an established field of research within the hardware architecture community. Researchers have proposed various solutions to mitigate kernel interference in GPU kernel scheduling [6]. All established DL cluster schedulers do not incorporate GPU kernel scheduling characteristics during job scheduling due to heterogeneous DL system hardware and DL frameworks. As an alternative to threadblock scheduling and kernel prediction, Horus uses application features to ascertain job utilization to alleviate GPU interference in DL systems, and provides a study focused on DL job interference due to co-location.

7 Conclusions

In this paper we have presented Horus, a resource management framework for Deep Learning systems that achieves high job throughput and resource efficiency via effective DL job co-location. Horus performs intelligent placement and co-location of DL jobs in GPUs by estimating job utilization patterns using model features without requiring DL library modification or heavy kernel profiling at scheduler runtime. From our analysis, we have empirically shown the diversity of interference profiles manifesting between co-located DL jobs, and can result in up to 3.2X increase within their completion time. We have created a resource estimator and scheduler integrated into existing DL cluster resource managers. Through experiments we have shown that Horus is capable of reducing makespan by up to 33.6% and improving cluster GPU utilization by 61.5%.

References

1. Nvidia Deep Learning Performance Guide, https://docs.nvidia.com/deeplearning/sdk/dl-performance-guide/index.html
2. Pytorch, https://pytorch.org/
3. Amaral, M., Polo, J., Carrera, D., Seelam, S., Steinder, M.: Topology-aware GPU scheduling for learning workloads in cloud environments. In: ACM SC (2017)
4. Bhuiyan, A., Guo, Z., Saifullah, A., Guan, N., Xiong, H.: Energy-efficient real-time scheduling of DAG tasks. ACM TECS **17**, 1–25 (2018)
5. Chaudhary, S., et al.: Balancing efficiency and fairness in heterogeneous GPU clusters for deep learning. In: ACM EuroSys 2020 (2020)
6. Chen, Q., Yang, H., et al.: Prophet: precise QoS prediction on non-preemptive accelerators to improve utilization in warehouse-scale computers. In: ACM SIGOPS Operating Systems Review (2017)
7. Chen, Y., Li, J., Xiao, H., Jin, X., Yan, S., Feng, J.: Dual path networks. In: Advances in Neural Information Processing Systems, pp. 4467–4475 (2017)
8. Delimitrou, C., Kozyrakis, C.: Paragon: QoS-aware scheduling for heterogeneous datacenters. In: ACM SIGPLAN Notices. ACM (2013)

9. Delimitrou, C., Kozyrakis, C.: Quasar: resource-efficient and QoS-aware cluster management. In: ACM ASPLOS (2014)
10. Gardner, M., Grus, J., Neumann, M., Tafjord, O., et al.: AllenNLP: a deep semantic natural language processing platform (2017)
11. Gers, F.A., Schmidhuber, J., Cummins, F.: Learning to forget: continual prediction with LSTM (1999)
12. Goodfellow, I., Bengio, Y., Courville, A.: Deep Learning. MIT Press, Cambridge (2016)
13. Gu, J., Chowdhury, M., Shin, K.G., Zhu, Y., et al.: Tiresias: a {GPU} cluster manager for distributed deep learning. In: USENIX NSDI (2019)
14. Han, D., Kim, J., Kim, J.: Deep pyramidal residual networks. In: IEEE CVPR, pp. 5927–5935 (2017)
15. He, K., Zhang, X., Ren, S., Sun, J.: Deep residual learning for image recognition. In: IEEE CVPR (2016)
16. Hightower, K., Burns, B., Beda, J.: Kubernetes: Up and Running: Dive into the Future of Infrastructure. O'Reilly Media Inc., Sebastopol (2017)
17. Huang, G., Liu, Z., Van Der Maaten, L., Weinberger, K.Q.: Densely connected convolutional networks. In: IEEE CVPR, pp. 4700–4708 (2017)
18. Iandola, F.N., Han, S., et al.: Squeezenet: alexnet-level accuracy with 50x fewer parameters and <0.5 mb model size. arXiv preprint arXiv:1602.07360 (2016)
19. Jeon, M., et al.: Analysis of large-scale multi-tenant GPU clusters for DNN training workloads. arXiv preprint arXiv:1901.05758 (2019)
20. Kambatla, K., Yarlagadda, V., Goiri, Í., Grama, A.: UBIS: utilization-aware cluster scheduling. In: IEEE IPDPS (2018)
21. Krizhevsky, A., Hinton, G., et al.: Learning multiple layers of features from tiny images. Tech. rep, Citeseer (2009)
22. Ma, N., Zhang, X., Zheng, H.T., Sun, J.: Shufflenet v2: practical guidelines for efficient CNN architecture design. In: ECCV (2018)
23. Mars, J., Tang, L., et al.: Bubble-up: increasing utilization in modern warehouse scale computers via sensible co-locations. In: IEEE/ACM MICRO (2011)
24. Merity, S., Xiong, C., Bradbury, J., Socher, R.: Pointer sentinel mixture models. arXiv preprint arXiv:1609.07843 (2016)
25. Peng, Y., Bao, Y., Chen, Y., Wu, C., Guo, C.: Optimus: an efficient dynamic resource scheduler for deep learning clusters. In: ACM EuroSys (2018)
26. Phull, R., et al.: Interference-driven resource management for GPU-based heterogeneous clusters. In: Proceedings of HDPC. ACM (2012)
27. Sandler, M., Howard, A., Zhu, M., Zhmoginov, A., Chen, L.C.: Mobilenetv 2: inverted residuals and linear bottlenecks. In: IEEE CVPR, pp. 4510–4520 (2018)
28. Schwarzkopf, M., Konwinski, A., Abd-El-Malek, M., Wilkes, J.: Omega: flexible, scalable schedulers for large compute clusters. In: ACM EuroSys (2013)
29. Shen, H., et al.: Nexus: a GPU cluster engine for accelerating DNN-based video analysis. In: ACM SOSP (2019)
30. Simonyan, K., Zisserman, A.: Very deep convolutional networks for large-scale image recognition. CoRR arXiv:1409.1556 (2014)
31. Szegedy, C., et al.: Going deeper with convolutions. In: Computer Vision and Pattern Recognition (CVPR) (2015)
32. Tan, M., Le, Q.V.: Efficientnet: rethinking model scaling for convolutional neural networks. arXiv preprint arXiv:1905.11946 (2019)
33. Tan, M., et al.: MNASNet: platform-aware neural architecture search for mobile. In: IEEE CVPR, pp. 2820–2828 (2019)

34. Vaswani, A., Shazeer, N., Parmar, N., et al.: Attention is all you need. In: NIPS (2017)
35. Vavilapalli, V.K., et al.: Apache hadoop yarn: yet another resource negotiator. In: ACM SoCC (2013)
36. (WMT19), A.M.T.: Shared task: machine translation of news. http://www.statmt.org/wmt19/translation-task.html
37. Xiao, W., et al.: Gandiva: introspective cluster scheduling for deep learning. In: USENIX OSDI (2018)
38. Xie, S., Girshick, R., Dollár, P., Tu, Z., He, K.: Aggregated residual transformations for deep neural networks. In: IEEE CVPR (2017)
39. Xu, X., et al.: Characterization and prediction of performance interference on mediated passthrough GPUs for interference-aware scheduler. In: HotCloud (2019)
40. Yeung, G.F., Borowiec, D., Friday, A., Harper, R., Garraghan, P.: Towards GPU utilization prediction for cloud deep learning. In: USENIX HotCloud (2020)