Proactive Interference-aware Resource Management in Deep Learning Training Clusters

Ging Fung Matthew Yeung

School of Computing and Communications

Lancaster University

This thesis is submitted for the degree of

Doctor of Philosophy

June, 2022

I dedicate this thesis to my family.

Declaration

I declare that the work presented in this thesis is, to the best of my knowledge and belief, original and my own work. The material has not been submitted, either in whole or in part, for a degree at this, or any other university. This thesis does not exceed the maximum permitted word length of 80,000 words including appendices and footnotes, but excluding the bibliography.

Ging Fung Matthew Yeung

June, 2022

Proactive Interference-aware Resource Management in Deep Learning Training Clusters

Ging Fung Matthew Yeung, BSc Computer Science.

School of Computing and Communications, Lancaster University

A thesis submitted for the degree of *Doctor of Philosophy.* June, 2022

Abstract

Deep Learning (DL) applications are growing at an unprecedented rate across many domains, ranging from weather prediction, map navigation to medical imaging. However, training these deep learning models in large-scale compute clusters face substantial challenges in terms of low cluster resource utilisation and high job waiting time. State-of-the-art DL cluster resource managers are needed to increase GPU utilisation and maximise throughput. While co-locating DL jobs within the same GPU has been shown to be an effective means towards achieving this, co-location subsequently incurs performance interference resulting in job slowdown.

We argue that effective workload placement can minimise DL cluster interference at scheduling runtime by understanding the DL workload characteristics and their respective hardware resource consumption. However, existing DL cluster resource managers reserve isolated GPUs to perform online profiling to directly measure GPU utilisation and kernel patterns for each unique submitted job. Such a feedback-based reactive approach results in additional waiting times as well as reduced cluster resource efficiency and availability.

In this thesis, we propose Horus: an interference-aware and prediction-based DL cluster resource manager. Through empirically studying a series of micro-benchmarks and DL workload co-location combinations across heterogeneous GPU hardware, we demonstrate the negative effects of performance interference when co-locating DL workload, and identify GPU utilisation as a general proxy metric to determine good placement decisions. From these findings, we design Horus, which in contrast to existing approaches, proactively predicts GPU utilisation of heterogeneous DL workload extrapolated from the DL model computation graph features when performing placement decisions, removing the need for online profiling and isolated reserved GPUs. By conducting empirical experimentation within a medium-scale DL cluster as well as a large-scale trace-driven simulation of a production system, we demonstrate Horus improves cluster GPU utilisation, reduces cluster makespan and waiting time, and can scale to operate within hundreds of machines.

Publications

Contributing Publications

The following publications have been produced while developing this thesis, and to an extent has guided the thesis into what it has become:

- **Yeung, G.**, Borowiec, D., Yang, R., Friday, A., Harper, R. and Garraghan, P., 2021. Horus: Interference-Aware and Prediction-Based Scheduling in Deep Learning Systems. IEEE Transactions on Parallel and Distributed Systems.

  I designed, implemented and evaluated the design of Horus, via experimental testbed and large-scale simulation. Damian Borowiec supported data collection for co-location experiments. Dr. Renyu Yang, Dr. Peter Garaghan, Professor Richard Harper and Professor Adrian Friday helped with the paper writing, provided feedback and technical support on methods.

- **Yeung, G.**, Borowiec, D., Yang, R., Friday, A., Harper, R. and Garraghan, P., 2020. Horus: An interference-aware resource manager for deep learning systems. Springer International Conference on Algorithms and Architectures for Parallel Processing.

  I designed, implemented and evaluated the profiling engine including the GPU memory estimator, and the scheduler experiments. The implementation of the ONNX profiling engine is a joint effort with Damian Borowiec. Dr. Peter Garaghan, Professor Richard Harper and Professor Adrian Friday helped with the paper writing and provided feedback on methods.

- **Yeung, G.**, Borowiec, D., Friday, A., Harper, R. and Garraghan, P., 2020. Towards GPU Utilization Prediction for Cloud Deep Learning. USENIX 12th Workshop on Hot Topics in Cloud Computing.

  I designed, implemented and evaluated the GPU utilisation predictor, scheduler design, experiments, and evaluation. The implementation of the PyTorch profiling engine is a joint effort with Damian Borowiec. Dr. Peter Garaghan, Professor Richard Harper and Professor Adrian Friday helped with the paper writing and provided feedback on methods.

Acknowledgements

Thank you to all who has supported me during my PhD journey through any forms of encouragement, both academically and otherwise. I will thank some of them in text, and no doubt apologise profusely to anyone I forget.

My first and foremost gratitude goes to my supervisors, Dr. Peter Garraghan and Professor Adrian Friday, for their invaluable, endless help and support over the course of the past four years. Peter's work ethic, attention to detail and enthusiasm have impacted my journey towards being a qualified researcher massively. I am grateful to both Peter and Adrian, taking the time to comments and providing insightful feedback on countless drafts of this research work.

I would like to thank current and former members of the Experimental Distributed System Lab B38, they have supported me in numerous ways. I am grateful to Damian Borowiec and Dominic Lindsay for comments that have improved this research work, our close collaboration on various projects and the continual battle with the cluster. Thank you to James Bulman, who assisted with equipment installation and setup. Thank you to Petter Terenius, who broaden my understanding on data centre waste heat and sustainability research. Finally, thank you to all members who took the time to participate in social activities that make the PhD journey even more enjoyable.

I would also like to thank immensely my collaborator, Dr Renyu Yang, who gave me career advice and his excellent feedback on this research work.

Thank you to colleagues of other groups in SCC, you have all made this journey much more enjoyable. Thank you to Dr. Gengshen Wu, Dr. Xiaowei Gu, Dr. Yao Zhang, Dr. Peng Cheng, Dr. Binbin Su, Dr. Carmen Štikonas, Dr. Sukhpal Singh Gill, Dr. Muhammad Umair Chaudhary, Charles Wu, Daria Smirnova, Yumqi Miao, Chengcheng Qu, Alexander Wild, and Zhang Shuo for their constant encouragement and helpful discussion in my early researcher days. Thank you to Dr. Andrew Scott and Dr. Yehia Elkhatib, who made my time as a TA delightful. Thank you to Professor Richard Harper, who supported my application for Microsoft AI summer School, which deepens my interest in Deep Learning. I am grateful to my colleagues during my internships at Alibaba DAMO Academy, ARM Research and Huawei Edinburgh Research Centre. I have gained valuable insights during my time at all of the above teams, which made me realise the opportunities for supporting cluster optimisation and ML for systems. Thank you to Lancaster University for providing me with a PhD studentship.

Thank you to all my friends, especially Carl Yau, Joshua Davis and Stefan Gill who kept me sane over the four years and I have enjoyed our weekend calls.

I want to thank my family who supported me during my journey. Thank you to my cousins Sing-Kai Chiu, Shirley Chiu, Joshua Yeung, Jaren Yeung, and Jasmin Chan for their support and keeping me entertained over the years. Thank you to my parent-in-law. Thank you to my mum, dad, and brother, Ging Hao Yeung supporting my decision to pursue a PhD and who are there whenever I need them.

Finally, I thank Yiyin Zhang, who accompanied my PhD journey – from my PhD early days to the final draft of this document – with loving support.

Contents

List of Figures

List of Tables

Nomenclature

Glossary

Co-location	The act of placing multiple workloads within a single location.
Convergence	Achieving a satisfactory result where the error is close to the minimum.
Feature	A distinctive attribute that can be incoporated into a data point.
Gradient	The derivative calculated with respect to a parameter, calculated for updating the parameter towards convergence.
Hyperparameter	A tunable parameter that is not part of a model, and is set before the training process begins.
Inference	The action of a learning model making a prediction given an input data.
Interference	Performance degradation due to resource sharing between workloads.
Locality	The network topology requirement of a job.
Loss	The error measure of a learning model with respect to an objective function.

Makespan	The total span time of cluster scheduling, measured from the arrival of the first job till the last job finishes its execution.
Training	The process of iteratively updating a learning model's parameters with respect to training data until convergence.
Weight	A parameter within a learning model.

Acronyms / Abbreviations

ANN	Artifical Neural Network
CNN	Convolutional Neural Network
CPU	Central Processing Unit
DL	Deep Learning
DNN	Deep Neural Network
FIFO	First In First Out
FLOPS	Floating Point Operations per Second
FLOPs	Floating Point Operations
FPGA	Field Programmable Gate Array
GNN	Graph Neural Network
GPU	Graphical Processing Unit
GRU	Gated Recurrent Unit
IR	Intermediate Representation
LSTM	Long-Short Term Memory

MAC	Multiply-Accumulate
MAPE	Mean Absolute Percentage Error
MLP	Multi-Layer Perceptron
ML	Machine Learning
MSE	Mean Square Error
OOM	Out-of-memory
QoS	Quality of Service
RL	Reinforcement Learning
RMSE	Root Mean Square Error
RMSLE	Root Mean Square Log Error
RNN	Recurrent Neural Network
SOTA	state-of-the-art
TPU	Tensor Processing Unit
XGBoost	eXtreme Gradient Boosting (Machine Learning model)

Chapter 1

Introduction

1.1 The Rise of Deep Learning Clusters

Deep Learning (DL) has made a significant impact in various domains from medical imaging [130], environment modelling [158], to DNA sequencing [258]. However, accomplishing production-ready Deep Neural Network (DNN) models requires significant compute and financial investment. The compute systems required for training and deploying DNN models for a large number of users are typically composed of large clusters of machines equipped with heterogeneous accelerator hardware devices such as Graphical Processing Units (GPUs) [79], Tensor Processing Units (TPUs) [113] and Field Programmable Gate Arrays (FPGAs) [40]. Given these large-scale *DL clusters* are expensive to design and operate[1] due to the significant demand for electrical power [200, 257] and computing resource [247, 266], it is necessary and desirable to maximise the underlying cluster resources more effectively (CPU, GPU, memory, network bandwidth, etc.) in order to improve cost efficiency, i.e., maintaining cluster resources consistently at a high level of utilisation.

[1]For example, assuming a NVIDIA V100 32GB GPU cost $3000 with bulk purchase discount (listed on Amazon for $11,999 [172] in 2022), the costs of 5,000 GPUs [247] is 15 million dollars.

1.2 Underutilisation in Deep Learning Clusters

Leveraging DL clusters and petabyte-scale data to produce highly accurate and domain-specific DL models is now the norm. For example, Meta in 2019 reported that it is common to perform frequent re-training of their News Feed, Ads, and Community Integrity models within their DL clusters to avoid prediction inaccuracy [86]. However, ensuring high resource utilisation and model iteration time performance guarantees within DL clusters is challenging to achieve with existing *DL resource managers* [81, 187, 246, 247, 266], that are primarily responsible for resource allocation and workload deployment, whereby a workload comprising many jobs [78, 195].

These challenges are mainly caused by: (i) hardware heterogeneity [162], whereby changes to iteration time performance from differing hardware processing power results in additional complexity in workload placement and choosing an appropriate input size [208]; (ii) unpredictable job completion time [81], resulting in less effective workload schedules within deadline-aware scheduling approaches designed to optimise system utilisation [95]; (iii) task distribution skew [81, 181], causing difficulties in estimating the number and resources required to execute a task within a job; and (iv) engineering limitations [95, 246, 247], whereby the maturing technology underpinning the majority of existing production DL cluster resource managers can not yet effectively share a GPU device.

Underutilisation is a systemic issue which frequently manifests within compute clusters, primarily due to their requirement to provision service for peak user demand. Therefore, compute clusters are designed and constructed to operate under both maximum and average resource usage scenarios [107, 199], therefore it is inevitable that these clusters often experience resource underutilisation [16, 52] because it is rare for cluster workload to exhibit peak demand constantly [263]. This is especially the case for DL training clusters [95, 108, 246, 247] that mainly hosts DL jobs, i.e. (i) *training* workload, the process of producing an accurate DL model; and (ii) *inference* workload, utilising the trained DL model to provide the predictions; the average cluster GPU utilisation of production DL clusters has been reported to between 15% to 55% [95, 108, 247].

An effective means to increase cluster resource utilisation is via workload *co-location*, whereby multiple DL tasks from the DL jobs, are simultaneously executing on shared physical hardware [51, 52, 148, 263]. However, naively co-locating these tasks can lead to performance interference (*interference*) due to resource contention [31, 148, 159], resulting in user-facing application quality of service (QoS) degradation [136, 146] and therefore, loss in financial revenue.

1.3 Performance Interference in Deep Learning Training Clusters

Performing effective workload co-location decisions that minimises interference requires understanding how the workload will behave on shared hardware, and attaining an estimate of the workload's *utilisation profile* (i.e. resource consumption usage patterns) during resource scheduling *assignment* time.

To mitigate interference in compute clusters, existing cluster resource managers require a dedicated GPU to conduct isolated profiling of workload resource utilisation and resource access patterns [169, 180], as well as performing online interference modelling for placement decisions [31, 51, 148, 180, 231]. However, these approaches are not directly applicable to DL clusters, primarily due to hardware architectural differences, since they only model CPU, memory, disk and bandwidth utilisation profiles, and not the GPU accelerators required for DL computation. This is because GPU interference is a result of the resource contention manifesting from resources such as the number of *processing elements*, global memory bandwidth and

PCIe bandwidth [29]. Second, the GPU resource utilisation profiles do not directly translate to their resource active usage by definition [174], and therefore it is unclear how workload slowdown can be inferred by these utilisation profiles. Furthermore, existing interference mitigation approaches are known to *reactively* migrate jobs upon performance degradation detection [31, 263], which could incur additional time overhead from re-scheduling and re-execution.

The majority of production DL clusters are incapable of co-locating workload because they are currently not designed to support accelerator hardware sharing (e.g., GPUs) [95, 246]. In recent years, a number of DL resource management frameworks have been proposed that allow for co-location to address underutilisation [246, 247]. However, co-location strategies are not considered at resource allocation scheduling time, which leads to mitigation approaches that are reactive, incurring re-scheduling time and resource overhead. For instance, Gandiva [246] requires isolated resource utilisation profiling to obtain their utilisation profile in order to enact workload co-location decisions. Antman [247] also requires isolated resource access patterns profiling to obtain both utilisation profiles and workload GPU operation kernel patterns to determine throttle and co-location decisions. These isolated resource utilisation and access patterns profiling approaches have to reserve dedicated GPUs, which reduces resource availability, increases waiting time and degrades cluster resource utilisation.

While there exist GPU-specific interference-aware cluster scheduling approaches [184, 227, 230], they all require isolated profiling of GPU resource consumption and hardware resource access patterns, which reduces the availability of cluster resources. This is particularly challenging within DL clusters, as obtaining the exact utilisation profile and GPU operation kernel patterns of a DL training workload can take between tens of minutes to hours to complete [247] due to the workload often undergoing many phases, including data loading, pre-processing, and training.

To increase resource utilisation and mitigate the profiling and re-scheduling overheads within DL training clusters, it is necessary to design a new workload co-location approach that is specially tailored to DL models. However, it is unclear how to interpret DL training workload utilisation profiles in the context of co-location, i.e., how much slowdown would a DL training workload experience. Therefore, there is a need to understand the system behaviour during DL workload co-location, the relationship between interference and their respective utilisation profiles; as this would allow us to ascertain these utilisation profiles efficiently and finally, enable us to determine safe co-location scenarios, improving DL clusters resource utilisation.

1.4 Research Questions

The objective of this work is to improve DL training cluster resource utilisation and cost efficiency. To achieve this, we propose Horus, a DL training cluster resource manager that can proactively determine safe DL workload co-location decisions by minimising interference, removing the need to reserve isolated hardware. Doing so requires understanding workload utilisation profiles. This work aims to explore how to determine the workload utilisation profiles on heterogeneous hardware efficiently, without reserving isolated hardware. From these profiles, it would then be possible to design resource managers capable of performing efficient workload co-location within a DL training cluster. This thesis postulates the hypothesis:

Hardware consumption patterns of DL training workload are dictated by the DNN's architecture, therefore extracting features from the DNN's architecture can allow us to estimate the resource utilisation on particular hardware, given historical data. Using these predictions, we can create DL cluster resource managers that improve cluster average resource utilisation, makespan and lower job waiting time.

This hypothesis can be validated by decomposing into three key research questions:

[RQ1] *When co-locating DL training workload onto shared hardware to improve utilisation, what are the relationships between their respective utilisation profiles and interference?*

Answering this question allows us to infer the likely interference effect given DL training workload utilisation profiles. This is important as it can enable us to consider the interference effect with respect to the actual resource utilisation during resource allocation scheduling time.

[RQ2] *How to determine DL training workload utilisation profile efficiently without online profiling?*

Answering this question first requires us to determine the key DL training workload features that drive their respective utilisation profile, thus enabling us to derive accurate utilisation profiles without reserving isolated resources.

[RQ3] *Can DL cluster resource management frameworks leverage this resource utilisation estimate to provide better co-location scheduling decisions than existing frameworks, thus improving average cluster utilisation and lower job waiting time?*

To answer this question, it will require us to design new components that can interact with resource management frameworks, in order to provide the resource utilisation estimate.

1.5 Major Contributions

The major contributions of this work are:

- An approach to profile and extract DL model intrinsic characteristics, in-depth analysis of these characteristics with its relationship to resource utilisation on specialised hardware. Existing DL resource managers are unable to determine resource utilisation based on DL model characteristic.

- Analysis of the performance degradation severity caused by DL job interference and identification of an efficient way to quickly determine potential time and resource cost when co-locating jobs. Existing DL resource managers are unable to proactively mitigate interference in DL clusters due to treating the underlying Deep Learning model as a blackbox.
- An empirical analysis of a production DL training cluster that demonstrate a systemic issue of low cluster resource utilisation, and this phenomena manifests across production DL training clusters of similar scale.
- A data-driven approach to rapidly determine the likely hardware resource consumption and identify the relevant features of a DL workload that directly influence the resource consumption. Based on the identification, we present a solution that extracts DNN architecture information and utilises historical resource consumption data to predict its resource consumption on hardware.
- A proactive interference-aware cluster scheduler that maximises resource utilisation while minimising performance degradation. We present how data-driven techniques can aid cluster schedulers in co-location decision-making within DL clusters. This is critical for providers to maximise cost efficiencies in terms of maximizing their resource utilisation.

1.6 Thesis Organisation

The thesis is structured as follows:

Chapter 2 introduces the topics of Machine Learning, Deep Learning Systems, and Cluster computing. The background is organised to understand deep learning compute clusters, from algorithmic intuition to software implementation, hardware execution and resource management. This is followed by a comprehensive overview of the state-of-the-art (SOTA) DL cluster schedulers, Interference-aware schedulers in clusters, Machine Learning for Systems, and Machine Learning for Clusters. Finally, the chapter concludes with the limitation of existing DL cluster schedulers.

Chapter 3 presents the empirical analysis of production DL training cluster, the co-location study on DL jobs, and relationship analysis between DL models and GPU resource consumption. This analysis demonstrates the key challenges in existing production DL training cluster, discovering the relationship between performance degradation and co-location, and presents the opportunity for data-driven estimation.

Chapter 4 presents Horus, a proactive cluster resource manager, and its approach to DL training job scheduling. Horus proactively improves cluster resource utilisation and minimises performance degradation caused by interference. Leveraging findings from chapter 3, we present the Horus GPU resource prediction engine and the final scheduling system architecture.

Chapter 5 presents the experimentation setup and evaluation of an empirical study in a university testbed DL cluster and a large-scale simulation based on a production cluster. We present the methodology used for evaluating the proposed approach. Specifically, this chapter quantifies and shows the improvements in cluster resource utilisation, makespan, job waiting time and job completion time with respect to different configurations of the experiments.

Chapter 6 summarises the contributions of the thesis, provides conclusions, and outlines potential future research directions for this research.

Chapter 2

Background

Many modern Deep Learning (DL) powered applications depend on large quantity of high-quality data to train the underlying models, which often take days to months to converge to a satisfactory accuracy. Large-scale DL compute clusters containing accelerators to speed up training [81, 162, 182, 187, 246, 247], are expensive to operate and require extensive effort to optimise.

This chapter presents the broader context of this research. In Section 2.1, the background of Machine Learning is discussed, presenting the main concepts behind prediction and Deep Neural Networks (DNNs). Section 2.2 presents the background of DNNs and their respective architectures. Section 2.3 then introduces software and hardware systems behind Deep Learning, the DNNs resource consumption, and the need for large-scale compute clusters to accelerate both training and inference. Section 2.4 first describes the need for compute clusters, then discusses resource management for large-scale compute clusters, followed by expressing the several challenges in both tradition compute clusters and in DL compute clusters context. Finally, we survey both non-DL-based cluster resource managers and DL cluster resource managers to identify gaps that this thesis addresses.

2.1 Machine Learning

The problem of learning can be defined as searching for patterns in data. Machine Learning (ML) is defined as machines automatically discovering regularities in data x through the use of a computer algorithm (a model f), with parameters θ; leveraging a model f with learned parameters θ to make predictions $\hat{y}$. Formally, this is defined as mapping x and θ into $\hat{y}$, $\hat{y} = f(x, \theta)$ [20]. The act of making a prediction is called inference. Majority of ML applications operate on *processed* data in the hope that the computer algorithms can discover regularities to make prediction better than heuristic approaches. For instance, a computer vision problem of digit recognition may pre-process the images by scaling the size of the image. This pre-processing stage is sometimes called *feature extraction*. The precise process of determining the function f and parameters θ is called the *training* or *learning* phase, on the basis of utilising the pre-processed or original training data. A model is defined as the function f and parameters θ, the training procedure usually involves splitting the data set D into training, validation, and test sets. Once the model is trained (i.e., the best parameters θ and the form of function f are determined), the model can be leveraged to determine the results of new validation and testing data. The end goal for ML training is to produce a trained model that can achieve satisfactory performance with respect to production measures such as the number of customers in the next hour, the travelling time between cities and the resource consumption of a workload.

Machine Learning can be categorised into three major learning paradigms: (i) Supervised Learning, (ii) Unsupervised Learning and (iii) Reinforcement Learning. Although each learning algorithm's inner working mechanisms are different, they can be combined into a larger system to solve complex problems. This work utilises both supervised and unsupervised learning as part of the larger system to make effective data-driven decisions. To evaluate how well the learning algorithms perform, it is essential to define an error or loss function in regard to the key performance indicators.

Loss function. An *error* or *loss* function is a function that evaluates how well the underlying models performed relating to production aspects such as accuracy, system costs, distances, and density [20]; in order to determine the proper set of parameters θ of a model. For instance, evaluating the accuracy between predicted CPU utilisation $\hat{y}$ with the true CPU utilisation y in the next hour can be evaluated with absolute error $\hat{y}-y$. Similarly, generating N images that are close to the original images can be evaluated by comparing each pixel for each image y with mean-squared-error (MSE) function as shown in Equation 2.1 [69]:

$$\frac{1}{N}\sum_{i=1}^{N}(\hat{y}-y)^2 \tag{2.1}$$

In contrast, for a multi-class classification across C classes, a common loss function is the cross entropy loss [77]:

$$-\sum_{i=1}^{C}\hat{y_i}\log y_i \tag{2.2}$$

where $\hat{y_i}$ is the predicted probability for class i. Distance measures such as the Manhattan Distance may be leveraged for clustering problems [105], i.e. grouping objects into individual groups. There are many more loss function such as Huber loss [99], Mean Average Percentage Error [48], Root Mean Square Error (RMSE) [102] and Root Mean Square Log Error (RMSLE) [215]. The loss function is the most important component for training a machine learning model, as it guides the learning model to make accurate predictions. The rest of the learning paradigms in this section all utilise one or more loss functions.

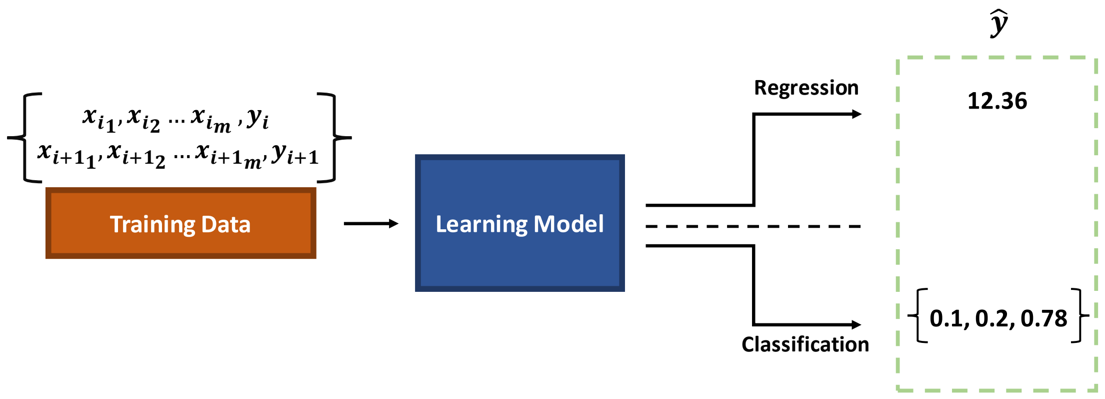

$x_{i1}, x_{i2} \dots x_{im}, y_i$
$x_{i+1_1}, x_{i+1_2} \dots x_{i+1_m}, y_{i+1}$
Training Data
Learning Model
Regression
Classification
$\hat{y}$
12.36
{0.1, 0.2, 0.78}

Figure 2.1: Simplified view of the supervised learning paradigm.

2.1.1 Supervised Learning

Supervised learning is a subcategory of machine learning, where the algorithm leverages ground truth input-label data pairs x, y to adjust its internal parameters θ through the training process [57]; where x is the input data and y is the ground truth label, in dataset $D = \{(x_i, y_i)\}, (|D| = n)$. The input data x can consist of m features that describe each data point, $x_i \in \mathbf{R}^m$. For instance, an input instance for predicting the CPU resource consumption in the next hour may involve current CPU utilisation, memory consumption and number of requests, i.e., a vector consisting of $\{x_{CPU}, x_{memory}, x_{num}\}$. The learning model f with parameters θ can learn to provide an accurate prediction $\hat{y}$, $\hat{y}_i = f_\theta(x_i)$, by adjusting its internal parameters according to an error metric that exploit a loss function l , $e = l(y_i, \hat{y}_i)$.

Supervised Learning problems can be categorised into either (1) classification – classifying a data point into a discrete class, or (2) regression – predicting a continuous value given a data point. Considering an image and asking the question whether this image has a cat, a dog, or a bird, is a classification task. Alternatively, predicting the housing prices for an area is a regression task. The simplified supervised learning paradigm is shown in Figure 2.1. Common algorithms that leverage supervised learning include polynomial regressions [20], linear regressions [20], decision trees [20, 23, 32, 65, 117] and *deep learning* [77]. In many complex and large-scale problems, obtaining accurate labels require in huge amount of manual labor efforts, and therefore *unsupervised learning* is exploited instead.

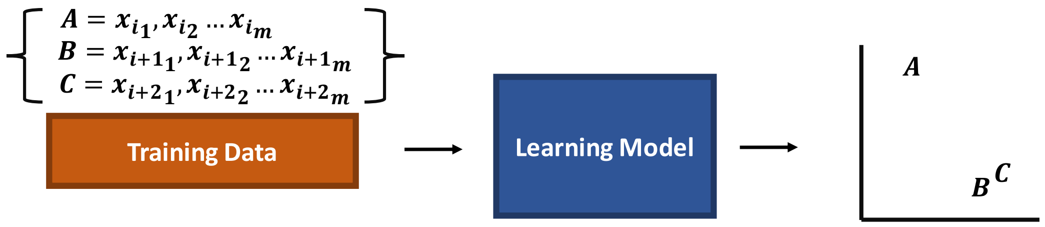

$A = x_{i_1}, x_{i_2} \dots x_{i_m}$
$B = x_{i+1_1}, x_{i+1_2} \dots x_{i+1_m}$
$C = x_{i+2_1}, x_{i+2_2} \dots x_{i+2_m}$
Training Data
Learning Model
A
B
C

Figure 2.2: Simplified view of the unsupervised learning paradigm.

2.1.2 Unsupervised Learning

In contrast, an unsupervised learning algorithm analyses input data without ground truth labels, i.e. $D = \{(x_i)\}, (|D| = n)$. The objective of unsupervised learning is to discover hidden patterns or data groupings according to a predefined metric [58], this is shown in Figure 2.2. The data groupings can reveal interesting insights among massive data sets, whereas discovering hidden patterns can lead to data dimension reduction, and association, such as a customer that bought a film also bought this other film. The above use cases are very common in production settings such as recommendation systems [88], vocabulary similarity [183] and anomaly detection [202]. However, unsupervised learning models can occasionally result in inaccurate results, and manual human intervention is needed to validate before deployment. Recently, a subcategory of learning algorithms that combines both supervised and unsupervised learning has emerged, semi-supervised learning. Semi-supervised learning uses a dataset that contains both labelled and unlabelled data, in the hope that a learning model can learn features that are relevant from the labelled data combined with pattern discovery to predict unseen data more accurately. This is particularly useful in settings where a portion of the data has been labelled, and obtaining ground truth labels for the remaining data is expensive. Apart from semi-supervised learning, *reinforcement learning* (RL) also contains elements of both supervised and unsupervised learning.

2.1.3 Reinforcement Learning

Reinforcement Learning (RL) is defined as learning how to map observations from the environment to actions given a cost function or a *reward* function [221]. Formally, the RL objective is to learn an optimal policy π with parameters θ. Under the policy π, an *action* a is performed given an observed *state* s from the *environment*. By interacting with the environment, an agent observed the new state s' and the reward r. The tuple (s, a, s', r) from each interaction is collected and served as training data. RL can be applied into various complex domains such as robotics manipulation [133], network congestion control [28] and process scheduling [143].

2.1.4 Applications

ML algorithms provide a computational approach that leveraged data, in order to learn the underlying patterns of the world [158]. There are many impactful applications that can leverage ML today, for example, gaining insights of customers behaviours [42], autonomous driving [12] and contributing towards biomedical engineering efforts [8]. Supervised learning is particularly useful, since ground truth labelled data can guide the algorithms into making accurate predictions. Unsupervised learning on the other hand contributes greatly to fields where data generation are fast and obtaining labelled data are expensive. They can be leveraged to discover clusters effectively. Finally, reinforcement learning is concerned with learning from interaction and has grown due to the success of beating human players in GO recently [212].

The field of machine learning has grown in an accelerated rate due to the re-emergence of *Deep Learning* (DL), and DL is an effective learning model algorithm that can be leveraged by all three paradigms for making inference.

2.2 Deep Learning

DL is defined as machine learning models that are based on artificial *neural networks* (ANNs) with many layers (greater than or equal to three), which enables learning complicated concepts by building them out of simpler ones [77]. As mentioned in Section 2.1, machine learning algorithms require engineers to carefully extract important features in the pre-processing stage to present a good *representation* of the data for the problem at hand. In contrast, DNNs do not require explicit handcrafted features and are generally well suited for high dimensional problems such as image and natural language domains. DNNs can discover not only a mapping from the representation to outputs, but also the representation itself from the data [77, 198]. However, the effort on handcrafted features has now shifted to carefully designing the DNNs architecture. Common architectures include Feed Forward Neural Network (Multi-Layer Perceptrons – MLPs), Convolutional Neural Networks (CNNs), Recurrent Neural Networks (RNNs), Graph Neural Networks (GNNs) and auto encoders. In the following section, we present MLPs, CNNs and RNNs due to their prominent use in the DL community and are the fundamental building blocks of larger DNN architectures. A comprehensive background of DL can be found in [77].

2.2.1 Artificial Neural Networks

Artificial Neural Networks (ANNs) are modelled after neuron connections within the human brain, where a single neuron receives signals from nearby connections and activates, if it crosses a threshold. Computationally, a neuron (node) contains its own set of inputs, weights θ and a bias value. When inputs x are received from nearby signals, a weighted sum is then performed between the weights and the inputs within the neuron. The bias term b is then added after. Finally, the neuron activates based on a predefined activation function f_a (i.e., crossing the threshold) and emits an output h. This is shown in the Figure 2.3 and mathematically, each node computes the following:

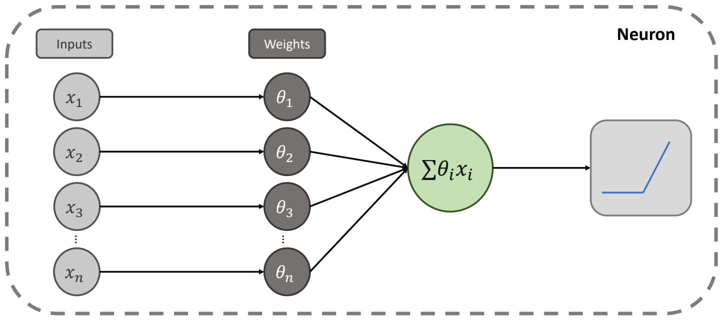


Figure 2.3: Internal of a single neuron, comprising a set of weights, performing a weighted sum and one addition, followed by an activation function.

$$h = f_a(\sum_{i=1}^{N} \theta_i * x_i + b) \tag{2.3}$$

A single neural network layer can consist of one or more nodes, as shown in Figure 2.4. DNN is defined as an ANN that comprises at least one input layer, and one or more intermediate (hidden) layers and output layers. Specifically, the input layers (the layers of which inputs are feeding in) do not consist of any weights. Outputs $\hat{y}$ depends on the previous hidden layers' outputs, which depends on the initial inputs x. Formally, let a DNN consists of $1...l \in \mathcal{L}$ hidden layers then:

$$\hat{y} = f_l(f_{l-1}(f_{l-2}(...f_1(x)))), \forall l \in \mathcal{L} \tag{2.4}$$

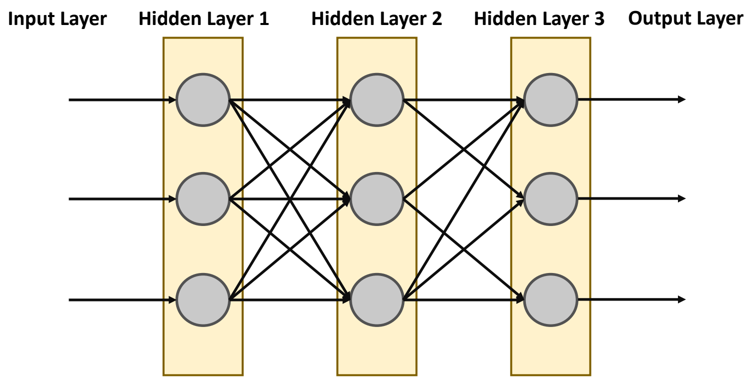


Figure 2.4: Example of a *Feed Forward Neural Network*.

DNNs can also be viewed as a graph, where vertices v are computation operations and edges e represent the connections. In certain application-specific architectures, the final outputs $\hat{y}$ and each individual hidden layer h_i can depend on many previous hidden layers h_j, where $j < i$, resulting in many inward edges to a single vertex [87]. The ability to learn useful representations not only depends on high-quality data and the training regimes, but also comes from the architecture.

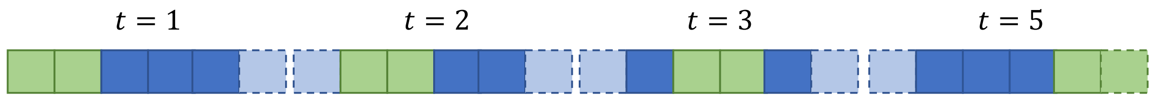
2.2.1.1 Neural Network Architectures

Feed Forward Neural Networks (Multi-Layer Perceptrons - MLPs), Convolutional Neural Networks (CNNs), and Recurrent Neural Networks (RNNs) are the most prominent architectures leveraged for solving different complex problems. All of these common architectures are utilised within this research.

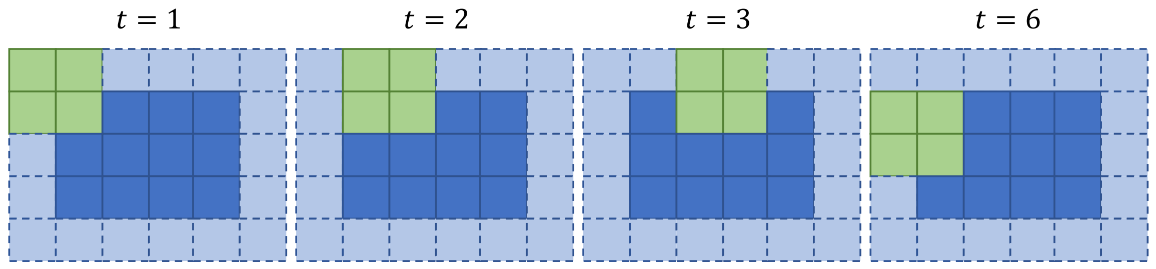
Feed Forward Neural Networks. This is the simplest DNN architecture, where it mainly consists of Fully Connected (Linear) layer. It is defined as a NN without any cycles, i.e., a Directed Acyclic Graph (DAG). The linear layer performs matrix multiplication, a dot product (weighted sum) between the inputs and the weights. The dot product outputs are followed by a non-linear activation function layer such as Rectified Linear Unit (ReLU), Gaussian Error Linear Unit (GELU) [89], or Scaled Exponential Linear Unit (SELU) [121], in order to introduce a non-linear relationship between the inputs and the outputs. An example of a Feed Forward neural network is shown on Figure 2.4. Although, feed forward network can capture non-linear relationships between inputs and outputs, there are architectures that can outperform feed forward networks in spatial or temporal problem domains. Nevertheless, research in feed forward networks is still ongoing [232] and recent research indicates that combining a feed forward network with other layer types can improve performance (e.g., the convolutional layer [56]).

Convolutional Neural Networks. For complex spatial problem domains such as object recognition, spatial grouping of features can provide stronger signals than scanning through an individual feature (i.e., a pixel). For example, an image usually has a dimension of width w, height h and channel C_{in} data represented in multidimensional arrays, where the channel dimension specifies the colors of the image, and grouping of these data can provide signals indicating an object is present. This is because the nearby pixels within a patch of an image, can have better representation than pixels that are at the edge of an image. For instance, an image patch that is near an object versus a patch that is of nothing but background. In these domains, CNNs can outperform vanilla feed forward networks by a large margin [123].

Instead of only using linear layers as the main feature extraction layers, CNNs utilise *convolutional layers* in which cross correlation is performed on a subset of the inputs through *filters* (kernels). The number of filters (kernels) in a convolutional layer dictates the output channel dimension C_{out}. Each filter has its width k_w, height k_h and channel k_c dimensions, and slides across the inputs from left to right, top to bottom, over the channels, according to (i) a stride s parameter, i.e., distance between two consecutive positions of the filter and (ii) a dilation d parameter, i.e., where the convolutional operation is performed on the expanded filter [254]; thus capturing the spatial relationship between input features. Engineers can specify *padding* p_j parameter where the features are padded by zeroes along j dimension before going through a convolutional layer in order to retain the feature dimensions. Retaining the spatial sizes by padding the inputs can have performance benefits, allowing the CNNs to retain information and propagate further into the network. Figure 2.5 shows 1-Dimension and 2-Dimensions convolution examples with padding.



(a) Convolution 1D with padding of 1, stride of 1. A filter of size 2 (green) is sliding across the inputs (blue).



(b) Convolution 2D with padding of 1×1, stride of 1×1. A filter of size 2×2 (green) is sliding across the inputs (blue).

Figure 2.5: Examples of Convolution Arithmetic for channel of size 1.

Similar to the fully connected layer, the convolutional layer's output goes through non-linear activation function to produce intermediate features. The output size of the intermediate features can be calculated from the following equation:

$$O_j = \lfloor \frac{I_j + 2 \times p_j - d_j \times (k_j - 1) - 1}{s_j} + 1 \rfloor \tag{2.5}$$

where O_j denotes the output size of dimension j, I_j denotes the input of dimension j, and j can be either the width w, height h or channel C dimension.

Pooling Layer is another layer that contributes towards the success in CNNs for the spatial problem domain. One can think of the pooling layer as a special type of convolutional operator where the function applied is typically one of two types: average or max, instead of a weighted sum. Average pooling layers compute the average of all input features under the filters, and max pooling layers compute the max of all features under the filters similarly.

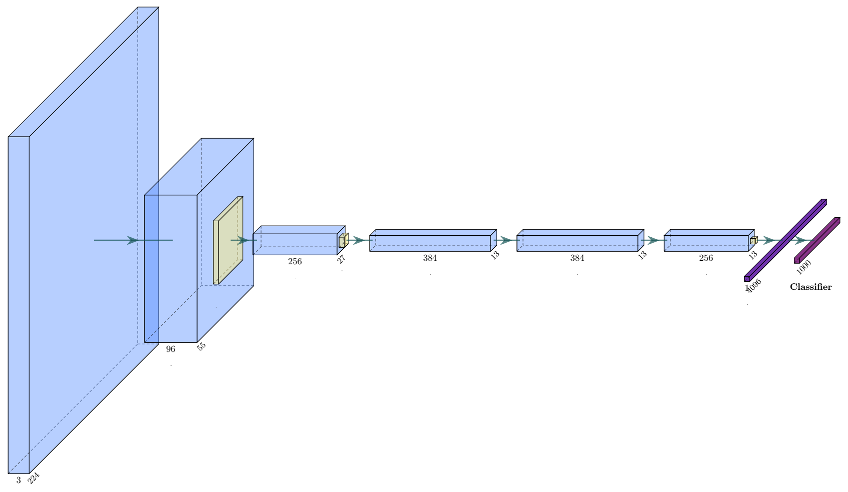


Figure 2.6: AlexNet [123] – Convolution Neural Network that consists of convolution layers (light blue), pooling layers (light yellow), fully connected layers (purple) and the final output fully connected layer (red).

An example of a fully constructed convolutional network is shown in Figure 2.6. Although convolution layer is typically applied to spatial problem domain, temporal dependence problems such as time series forecasting are also applicable. However, for long term temporal dependence problems such as language modelling, a recurrent neural network is better suited.

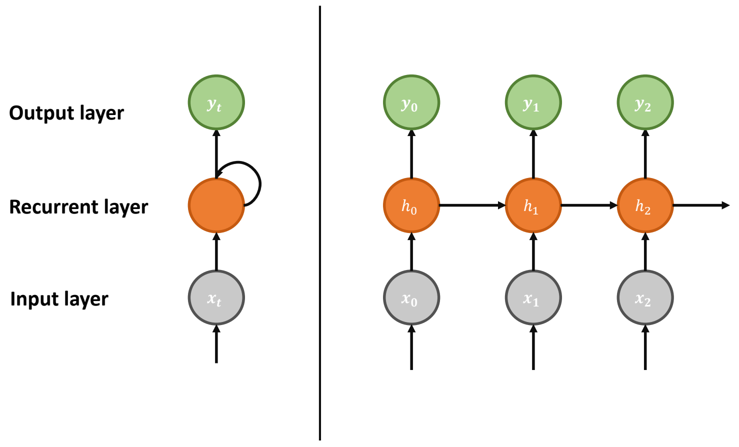


Figure 2.7: An example of a 1 layer Recurrent Neural Network. Weights are shared across each time step. Representation of rolled RNN (Left) and unrolled RNN (Right).

Recurrent Neural Networks. RNNs are better suited for long term sequential dependence problems because each input x_t in the sequence can take information *bidirectionally*, i.e., from either prior step x_{t-1} or future step x_{t+1} depending on the direction, hence having similar behaviour as "memorising" the features across time steps. Note that, unlike CNN and MLP, instead of having different weights in each layer, RNN layer share weights across each time step. Additionally, each time step takes a feature vector from the previous layer h_{t-1} to produce h_t in order to retain information[1]. Figure 2.7 shows an example of a recurrent neural network.

Although, an RNN architecture is good for modelling a temporal dependence problem, having a long sequence of time steps in RNNs can result in diminishing performance [91]. Nevertheless, two approaches proposed to address the aforementioned problem towards regular RNNs resulted in Long Short Term Memory (LSTM) neural networks and Gated Recurrent Unit (GRU) neural networks.

[1]The initial h_0 can be initialised randomly or zeros [196]

LSTM neural networks use *gates* to control the information flow, and outperform vanilla RNNs. Instead of having recurrent hidden layers, LSTM comprise recurrent modules (cells), where a cell has three gates, an input, output, and a forget gate, as shown in Figure 2.8. Each gate has its own set of weights to learn to activate based on the provided information. First, the forget gate is applied to the previous cell state c_{t-1} based on previous hidden state h_{t-1} and the input x_t, producing a filtered cell state c'_{t-1}. Second, the input gate is responsible for deciding how much information should get updated to the filtered cell state c'_{t-1}. Finally, the output gate decides the output for h_t and c_t feature vectors. For the interested reader on the inner workings of the gates, we refer the reader to [77, 91]. While LSTM outperforms vanilla RNN, the introduced gates added additional computation and memory parameters.

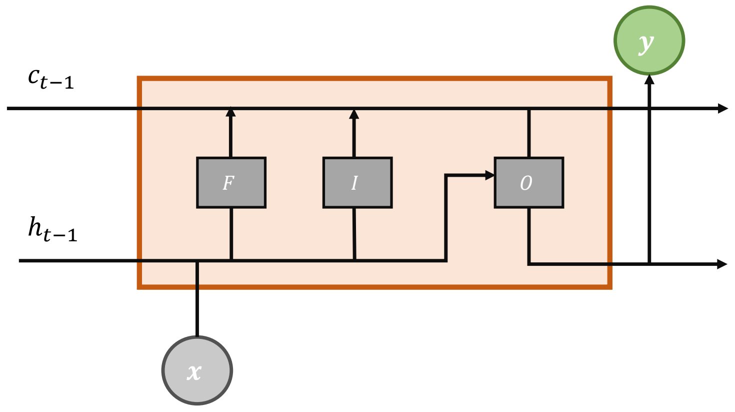


Figure 2.8: An overview of LSTM recurrent module, F, I, O represent Forget, Input and Output gate respectively.

GRU neural networks are a simplification of LSTM networks. GRU networks also have recurrent cells throughout the network, but instead of consuming and producing two feature vectors along with the input x_t, GRU produces only one feature vector h_t. We refer the interested reader to [77] for more details. Both LSTM and GRU utilise learned gates to determine the information flow during each time step x_t.

Although, recurrent neural networks outperform vanilla feed forward neural networks in long term temporal dependence domain problems, recent research has indicated that by carefully constructing a feed forward neural network (Transformer [232]), feed forward neural network can outperform recurrent networks even in applications such as language modelling [189].

The design of the above architectures all comes from understanding the target applications at hand. For instance, CNN LeNet-5 [127] is inspired by previous research on understanding a cat's visual system [98]. Hence, it is important to design a DNN architecture according to the target application. However, designing a DNN architecture requires domain expertise and takes a large amount of experimentation to find the optimal architecture.

All of the above neural network architectures have been shown to be successfully applied in DL applications [12, 129].

2.2.2 Deep Learning Applications

A wide range of complex problems are now leveraging DNNs to make accurate decisions such as DNA sequencing prediction [60], environment modelling [158], and map navigation [125]. While many applications can leverage DNNs to tackle problems, this thesis focuses on two specific applications domains for micro-benchmarking and analysis, (i) image classification and (ii) language modelling. These two application domains are common micro-benchmark targets due to the ease of accessibility and wide adoption [81, 182, 246].

Image Classification. Given image data, the aim is to assign a suitable label for the image content. Such classification is useful in vision applications such as cancer or illness detection [59, 135, 197, 252]. It was shown that DNNs can significantly outperform other machine-based vision algorithms [123]. Many highly accurate proposed DNN architectures consist of the previously presented layers [87, 224]. Importantly, training such DNNs and discovering effective architectures can take days to months of computation and experimentation.

Language Modelling. Remarkable results are shown within language modelling domain, where DNNs can generate sentences for documents of text given inputs related to a particular topic [54, 166, 190, 232]. GPT [190] is one of the most advanced language models, containing billions of parameters. It was shown that training such a language model requires petabytes of data and compute clusters operating for months to achieve state-of-the-art (SOTA) results. Although this thesis does not leverage GPT-based models, this thesis utilises Transformer [232], an architecture that is leveraged by GPT-based models and recent SOTA computer vision models.

Training and discovering accurate DNN models not only require efficient algorithms but also both software and hardware systems. Software systems should ensure applications are scheduled and executed efficiently on physical hardware with respect to resource utilisation, availability, and performance.

2.3 Deep Learning Systems

As research on DL continues to innovate, the underlying systems for DL also require research effort to cope with performance constraints and increased demands. In this section, DNN execution on hardware and software systems, the underlying principles for both DL training and inference are presented; the DL system key performance metrics are discussed, and pertaining to the proposed approach in this thesis.

2.3.1 Deep Learning Program Execution

DL program execution usually takes matrices as the inputs of the program. Tensors are the inputs to a DL program, a generalisation of matrices with dimensions greater than or equal to two. DL programs have one or two major execution phases (i) forward phase, where tensors flow forward along DNN layers producing an output, and (ii) backward phase, where the learning step of DNN happens [77].

2.3.1.1 Execution Phase

As mentioned in § 2.2.1, each DNN is represented as a graph, has its own sets of weights θ, the objective of training a DNN is to achieve satisfactory results by adjusting the weights iteratively according to one or more loss functions, until convergence $Err <= \epsilon$, where $Err = l(y, \hat{y})$ is an error measure. An iteration of training step involves two operations, (i) forward, this is also called inference, and (ii) backward, on the basis of *backpropagation* [126, 198], which is defined as leveraging chain rule to calculate the contribution of each weight towards making the error, thus adjusting the weights towards the *derivatives* direction that minimise the error [77]. The step of updating the weights is called *Gradient Descent* [44] and is a common method for optimisation problems. Iterating over an entire training dataset once is called an *epoch*. The training phrase can consist of thousands to millions of epochs. It is hard to predict the number of epochs for a DNN to converge (see Table 2.1). This is because DNN architecture, size of data and the complexity of the problem can affect the rate of learning and convergence.

Additionally, the parameters for training such as batch size, learning rate and optimiser configuration [111, 120, 253] that do not affect the DNN architecture are called *hyperparameters*, however, they all contribute towards the learning process, *statistical efficiency* [79, 187, 253], i.e., the amount of progress made per training

DNN Type	Parameters (MB)	Examples to Convergence	FLOPs per Example
MLP0	225	1 trillion	353 M
MLP1	40	650 billion	133 M
LSTM0	498	1.4 billion	29 G
LSTM1	800	656 million	126 G
CNN0	87	1.64 billion	44 G
CNN1	104	204 million	34 G

Table 2.1: Six DNN models that were used at Google in 2018 [16]

example processed. For the inference phase, the underlying DNN weights are frozen (fixed) and will not be updated when serving online queries, and therefore inference typically has less computation per iteration when compared to training.

The execution of a DL program can cross many layers of the stack, including both software, such as DL frameworks, and hardware accelerators.

2.3.1.2 Hardware Systems & Frameworks

The majority of the DNN computation comes from matrix manipulation, i.e., multiplication and addition; the parameters of a DNN are usually stored in the machine's memory as Single Precision Floating Point (FP32), hence the larger the model, the more *Floating Point Operations (FLOPs)* are performed. To accelerate DNN execution, hardware must execute these FLOPs efficiently. Central Processing Units (CPUs) are general purpose and built with tens to hundreds of cores. Although they can execute DNN operations with reasonable latency in the inference phase, they can not execute DNN training in a reasonable time. There are much better suited hardware devices for DNN execution phases. *Devices* such as Graphical Processing

Units (GPUs), Tensor Processing Units (TPUs) and Field Programmable Gate Arrays (FPGAs) are devices that consist of hundreds to thousands of processing cores. Therefore, they are better suited for high throughput data intensive operations, due to the ability to process large number of FLOPs, through higher *hardware efficiency*, i.e., number of examples processed per wall-clock time [187]. These devices are often referred to as *accelerators* as they speed up computation. TPUs version 1 to version 3, and FPGAs are predominantly leveraged for the inference phase, due to the ability to customise the underlying hardware topology to specifically execute matrix multiplication efficiently, resulting in much better energy efficiency [40, 113, 222]. In contrast, GPUs have a fixed topology and are designed to be general purpose in comparison to TPUs and FPGAs, they allow for more flexibility to run bespoke DL GPU operations and run DL training in large batches of data at the cost of more energy consumption. GPU accelerators are the main hardware focused on in this thesis due to academic constraints. As DL research accelerate, DL frameworks

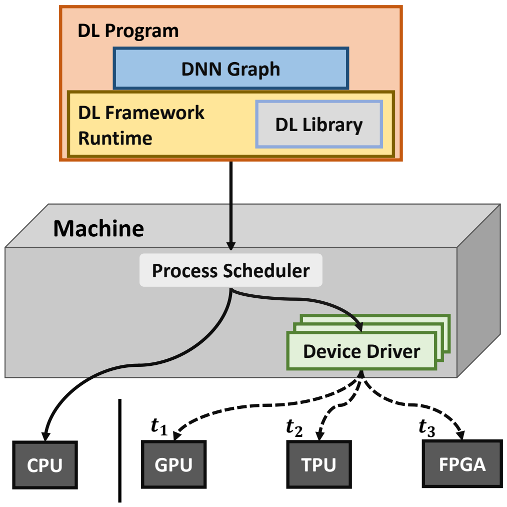


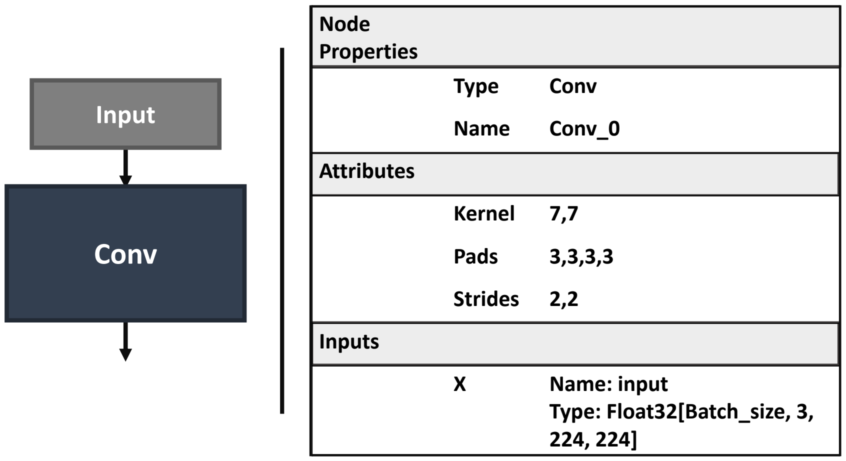
Figure 2.9: DL program execution in machine and devices. For a program that exploits accelerators, a DL framework's runtime will request program tasks to schedule onto them via device drivers.

such as PyTorch [178], TensorFlow [4] and MxNET [33], were created to provide high-level APIs for DL engineers to write DL programs efficiently (easily define DNN architectures and automatically perform backpropagation). To execute DL programs on machines and devices, DL frameworks parse the DNN graph and schedule each operation onto the devices for execution, as shown in Figure 2.9.

Interposed between a plethora of DL frameworks and hardware accelerators, intermediate representation (IR) for DL programs were developed to enable smooth transition and execution between systems.

2.3.1.3 Intermediate Representation

The parsing of the DNN graph generates a high-level IR of the DNN architecture, in which the IR is often referred to an object instance that represent the DNN graph programmatically [35, 45], without specifying how the underlying operation must be implemented. For instance, a convolution IR specifies the common parameters for a convolutional layer such as kernel size, stride, and padding as shown in Figure 2.10.



Node Properties		
	Type	Conv
	Name	Conv_0
Attributes		
	Kernel	7,7
	Pads	3,3,3,3
	Strides	2,2
Inputs		
	X	Name: input Type: Float32[Batch_size, 3, 224, 224]

Figure 2.10: Convolution Operator IR, adopted from Open Neural Network Exchange (ONNX) [74].

IR is especially useful due to interoperability [74], for instance, ML engineers in a training team can train a model in a specific framework and convert the same model into a common IR, in order to collaborate with deployment engineers that utilise another framework. In certain inference frameworks [173], the IR file is all that is needed for executing the DL program as it captures the dependencies between operations and control flows of the DNN program. Additionally, IR provides an interface and opportunities for high level hardware-independent (graph level) and low-level hardware dependent optimisation to accelerate executions. There are numerous DL compilers that focus on both high and low-level optimisation such as Halide [191], TVM [35] and PET [240]. Their objective is to generate highly optimised machine code, including scheduling order for execution. Nevertheless, the scheduling order and execution of a DL program largely depends on the high-level IR.

Scheduling execution on machines and devices is important, as scheduling policies can affect various production aspects within systems. To schedule DL program execution on an actual hardware, a process scheduler is needed for scheduling program execution. Additionally, custom device drivers and device schedulers can also affect the scheduling efficiency of the program, depending on the accelerators.

2.3.1.4 Execution Scheduling

Before the GPU kernel operations are scheduled to execute on the GPU, the DL program process is first scheduled onto the CPU by the process scheduler.

Process Scheduling. When a Process Scheduler such as the Linux Completely Fair Scheduler (CFS) or a custom-built microkernel scheduler [66, 101] decides to run a DL program among other processes in the same machine, the process scheduler makes the scheduling decision according to a *policy*, where policy dictates the scheduling order with respect to various business aspects such as *makespan* (the total time it takes to finish all jobs), *fairness* (waiting time) and resource utilisation (resources used over available resources) and latency (time taken for a task to finish its execution).

For example, consider the following scenario, assuming only one resource unit is available, no sharing of resources and pre-emption is not permitted (i.e., once a job is started, it cannot be stopped unless the job is finished or encounters errors). With the Shortest Job First (SJF) scheduler, the objective is to reduce the average job completion time, which it does so by scheduling job with the shortest computation time, however, longer jobs can be starved in such systems [81]. First In First Out (FIFO) schedules jobs according to the arrival order. FIFO, the objective is to maximise fairness by scheduling jobs and their waiting time; however, resource utilisation could be decreased due to long and small jobs blocking up the resources for sharing [246]. Similarly, Linux CFS's objective is to maximise fairness with respect to tasks priority; therefore, a red-black tree data structure is exploited in order to execute the highest priority thread based on CPU timeshare and task priority [26, 175]. Determining a suitable policy depends on the scheduling scenario.

Although a process scheduler manages which process gets executed next, it does not manage the actual execution on the external devices such as accelerators, due to these devices having their own separate vendor-specific device scheduler [156].

Device Scheduling. When a DL program executes on multicore machines with accelerators such as GPUs, since host CPUs and GPUs are different devices, there are different vendor-specific device drivers installed in order to aid with scheduling work onto accelerators. For example, NVIDIA GPUs contain a proprietary device driver that schedules work onto the device [156]. Note that devices may have different definitions of resource utilisation. For instance, NVIDIA GPUs report *GPU utilisation* as *the percentage of time in a given sample interval where one or more kernels executed on a GPU* [174]; the definition does not measure the actual cores' utilisation but only whether there is work that is currently being carried out in the given sampling period. This is not useful to determine how many cores exactly are being utilised.

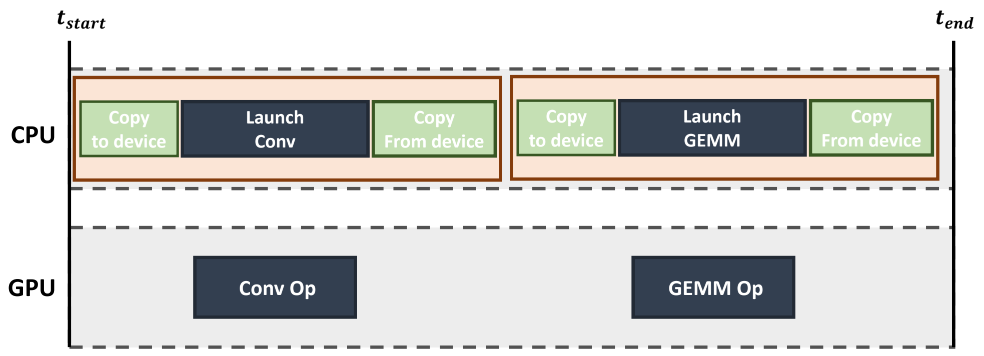


Figure 2.11: An example of launching DL operations sequentially (time-share), convolution operation follows by a GEMM operation onto the GPU. Assuming the sampling period is between the start and end of the timeline, the GPU utilisation would be shown as 50%.

In Figure 2.9, a DL program using a framework such as NVIDIA Triton Inference Server [173] parses the DNN graph in IR form, then schedules the DNN operations through the device driver; the implementation of the DNN operations are implemented in a hardware-aware DL library [38, 71, 237]. The scheduling order is usually in topological order to guarantee correctness but can be in out-of-order within the ready operations [109]. The scheduling of the DNN operations first follow the framework's policy for scheduling onto the device [109, 247], then follow the device scheduler's policy for actual execution [110, 156]. Figure 2.11 shows an example of the devices and their corresponding execution order, the accelerator starts executing the DL operations (*GPU kernels*) after CPUs initiate the execution through invoking device-specific APIs. For GPUs, time-sharing occurs when cores or memory on the GPU do not satisfy an individual GPU Kernel, but when GPU Kernels can "fit" on the GPU, they can be space-shared.

Since, the hardware efficiency and operational cost are proportional to hardware usage [16, 244], it is important to utilise these expensive accelerators well in terms of resource consumption. DL model resource consumption largely depends on the memory and computation it needs.

2.3.1.5 DL Model Hardware Resource Consumption Factors

Memory Consumption. Accelerator memory capacity is the major limiting factor to be considered when executing DNNs in DL training. Consider a DNN with M parameters, $\mathcal{L}$ layers, and inputs with batch size of B, the total inference phase's memory consumption Mem_{infer} is approximated with the following:

$$Mem_{infer} = M + (\sum_{i=0}^{\mathcal{L}} h_i \times B) + \epsilon \tag{2.6}$$

where h denotes the per layer intermediate outputs and ϵ denotes the frameworks or library memory overhead including, for example, temporary memory for computations. For the total training phrase's memory consumption Mem_{train}, G gradients, optimisers' states O_s are needed for the backpropagation calculation where G and O_s are both $\simeq M$ [43, 77].

$$Mem_{train} = M + G + O_s + (\sum_{i=0}^{\mathcal{L}} h_i \times B) + \epsilon \tag{2.7}$$

Therefore, the memory limitation mainly scales with the batch size B and the model's parameters M, and so does the computation.

Computation. The larger the DNN, the more FLOPs the accelerators have to perform. For the inference phrase, the total inference phase's computation Com_{infer} is given as:

$$Com_{infer} = \sum_{i=0}^{\mathcal{L}} m_i \times h_i \times B \tag{2.8}$$

where m denotes per layer parameters count.

For the training phrase Com_{train}, the additional backward phrase and optimiser calculation should be taken into account. Calculating the derivatives for each parameter may require the other parameters' derivatives due to chain rule [126, 198].

Additionally, the optimiser's gradient descent algorithm is computed regarding each parameter [253]. Therefore, the total computation required is shown as:

$$Com_{train} \simeq 2 \times Com_{infer} + \sum_{i=0}^{\mathcal{L}} l_i(m) \tag{2.9}$$

Discussion. Memory consumption estimation (2.6 and 2.7) of a DL program is difficult. This is due to the various independent optimisation efforts conducted across the individual DL frameworks and DL libraries [34, 192, 256], causing difficulties to account for the memory savings. For example, DeepSpeed ZeRO partitions the model parameters, gradient and optimizer states to avoid memory redundancy in large-scale distributed data-parallel training regime [256]. In our estimation modelling, we do not account for these memory saving techniques.

Nevertheless, the number of FLOPs scale with the input data and DNN parameters. To saturate the accelerators' compute capability, engineers can choose to scale the batch size, optimise scheduling policies, or redesign the DNN model.

2.3.2 Deep Learning Training System Limitations

There are many successful DNN-powered applications today that achieve SOTA results in domains such as DNA sequencing [8], games [212], and autonomous driving [12]. However, discovering an accurate and production-ready DNN model often requires a large number of machines due to the following reasons:

Complex Problem Domains. All of these SOTA DNN models require a huge number of computations, iterating over terabytes of data repetitively in order to learn and produce useful predictions. For instance, OpenAI-Five [17] leverages 51,200 CPUs and 1024 GPUs for multiple months in order to train a single layer LSTM DNN.

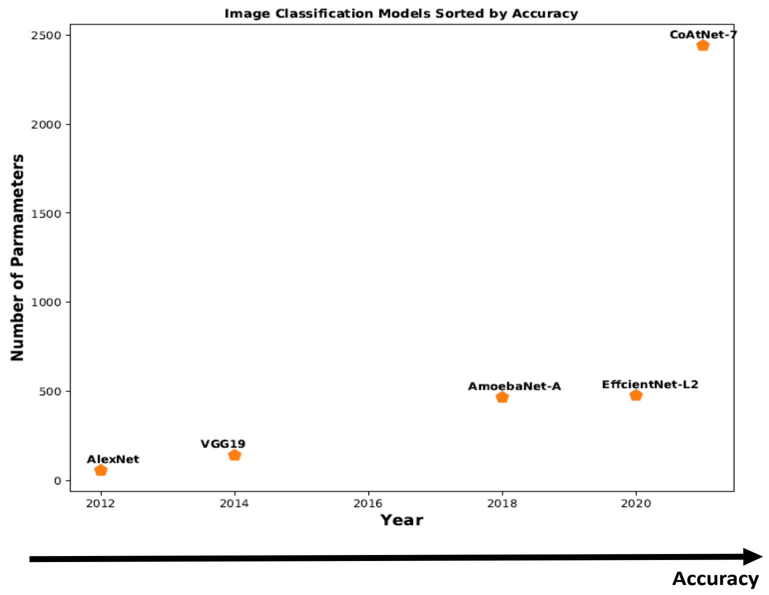


Figure 2.12: The growth of the model parameters in the past years [46, 123, 193, 213, 248]. The higher number of parameters, the higher accuracy obtained in ImageNet ILSVRC 2012 [53].

Large Models. Solving computationally complex problems [12, 158, 217] often requires more memory parameters, as it has been shown that larger models can outperform smaller ones [224]. Figure 2.12 shows selected models the numbers of parameters. The Alibaba M6 [134] model contains 100 billion parameters, which requires roughly 372 GB of memory to host the model, exceeding the capacity of a single accelerator, even at the 80 GB NVIDIA A100 [171].

Large Datasets. To cope with the ever-increasing scale of data, and newly generated data each day, distributed storage systems spanning across many machines are leveraged [131, 211]. For example, the size of the July 2019 Common Crawl uncompressed text dataset was 242 TB [114], exceeding the capacity of a commodity hard disk in a single machine. The frequent retrieval of training data from many distributed machines requires new innovation in these storage systems.

Large Design Space. Searching for an optimal DNN architecture not only involves carefully constructing the DNNs but also tuning relevant hyperparameters (hyperparameter tuning), and rapid turn-around experiments [95]. For example, to search for an optimal DNN, over 12,800 experimental DNNs were trained using Neural Architecture Search (NAS) at Google [269]; DeepMind evaluated over 1,500 relevant hyperparameter settings for a LSTM model [151]; calling for both efficient searching methods and parallelisation across machines.

New & Frequent Data Generation. Coupled with the above challenges, data is generated each day that may affect previously trained DNN models, and therefore frequent re-training of the DNN models and data preprocessing is required. For example, the Meta Community Integrity team require frequent re-training of their models to avoid potentially misclassified offensive content [86]; also calling for parallelised training of individual models across machines to keep up with new data.

A large-scale cluster of machines is the ideal solution to tackle the above challenges in order to support searching and training DNNs using large datasets repetitively across multiple organisations and teams, at the same time in a cost-efficient manner.

2.4 Deep Learning Training Cluster Management

DL researchers and engineers alike desire (i) a quick turn-around time for training and evaluating the performance metrics of a DNN in the search for an optimal DNN architecture, and (ii) high availability of their prediction services with quick response time. A large-scale cluster of machines can provide the infrastructure and necessary resources (i.e., the computation, storage, and ease of management [16]) needed for these services to operate.

2.4.1 Compute Clusters

A *cluster* is defined as multiple machines co-located and interconnected with network capabilities to communicate and provide resources for one and another [16].

There are many forms of large-scale compute clusters, some comprising hundreds and thousands of machines, across various paradigms such as grid computing [64], cloud computing [64], fog computing [21] and edge computing [210]. Resource sharing and problem-solving are the primary drivers for these cluster use cases. Many large-scale applications today, such as web search [15], social media platform [177, 226] and video sharing platform [47] run on clusters due to the ability to scale resources dynamically responding to user traffic demand in a timely manner [207], providing large amounts of storage and compute resources across multiple tenants (i.e., multiple organisation and teams) cost efficiently [16].

Compute clusters exist to simplify the deployment of applications and use the underlying hardware resources efficiently. One of the key technologies that enable deployment of applications easily across such machines is virtualisation [14, 152]. Virtualisation enables isolation of applications and allows multiple guests operating systems to co-exist on the same physical hardware. Popular variants of virtualisation are containers [18], as they allow sharing of host OS kernel and associated binaries and libraries, executing workloads on the host operating system without hosting a full guest OS on top of the existing OS.

Jobs & Tasks. Workloads consists of many jobs. A job is defined as the grouping of one or more tasks to ease management [24, 234]. Consider a DL application with four containers, where each container requests a GPU to accelerate training of a DNN. We can coincide the DL application as a DL job with four tasks. An example of a DL job is given in Figure 2.13.

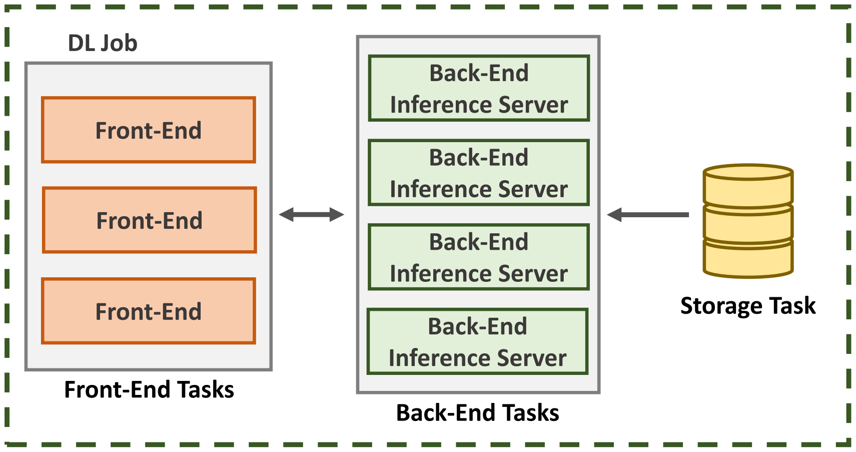

DL Job
Front-End
Front-End
Front-End
Front-End Tasks
Back-End Inference Server
Back-End Inference Server
Back-End Inference Server
Back-End Inference Server
Back-End Tasks
Storage Task

Figure 2.13: A DL inference job with three task types and their replicas (copies).

DL training jobs are usually offline, do not require immediate return of results, and can tolerate delays or failure of machines [81, 86, 108, 140, 162, 246]. In contrast, online DNN-powered user-facing jobs must respond in a timely manner even under high numbers of user requests [82, 177, 208].

Managing deployment and assigning resources for these jobs require a layer of software components to orchestrate systems and machines. Resource management framework is the layer responsible for the orchestration.

2.4.2 Resource Management

Resource management in clusters can be defined as controlling the mapping of tasks to hardware resources, enforcing priorities of these tasks for execution and managing quotas between tenants [16]. Managing clusters of machines and placement of tasks, via a layer of software components can ease management overheads. A resource management framework (*resource manager*) is needed, and it provides a software interface to automate resource management processes [16]. The two key responsibilities of cluster resource management are (i) *resource allocation* – allocate the resources to the tasks and (ii) *resource assignment* – assigning the tasks to the best machines according to the scheduling policy [52, 76].

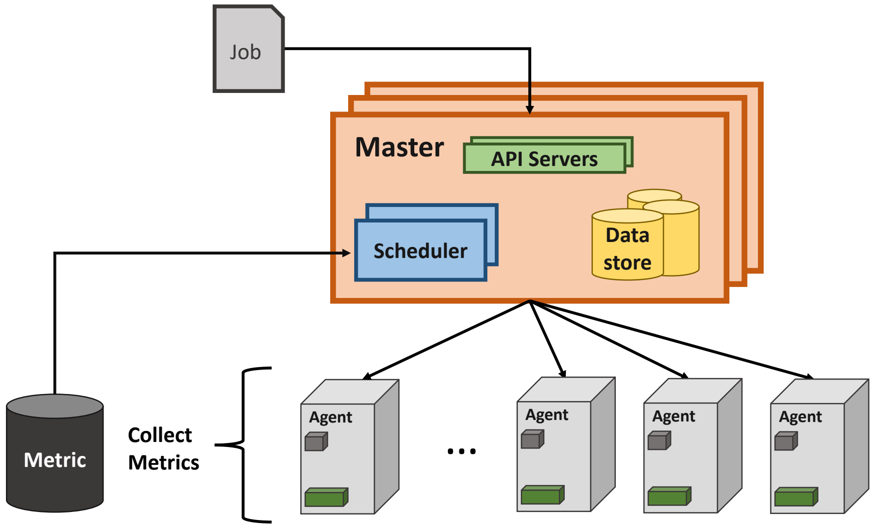


Figure 2.14: Major components and roles in cluster resource management framework.

These resource managers [24, 90, 226, 233, 234] consist of the following major roles: master and agent. An overview of resource manager is shown in 2.14. A master is a locally centralised controller comprises the following software components: (i) *cluster scheduler* – a crucial part of resource management within compute clusters responsible for allocation and assignment decisions, (ii) persistent database – responsible for storing tasks allocation and assignment decisions, and (iii) API server – a server for communicating state information and job submission. An agent runs on each machine, a software component that receives decisions from the master and manages execution of tasks, communicating machines' state information back to the master.

Quality of Service (QoS). One of the key responsibilities of the resource manager is to ensure tasks' QoS requirement are met [31, 51, 244, 264]. The QoS requirement are maintained by defining one or more Service Level Agreements (SLAs) between the cluster operator and the task's owner [112, 179]. A SLA is defined as a contract that includes the consequences of meeting or missing the task's objectives, whereby the objectives can be a target value or range of values [112, 179]. Therefore, cluster scheduler decisions can have a significant impact on task QoS.

Cluster Scheduler. A cluster scheduler makes the resource assignment decision by selecting appropriate machines based on the scheduling policy, which is similar to process scheduling mentioned in Section 2.3.1. A cluster scheduling policy can make these assignments based on a series of rules heuristically [24, 83, 234] or formulate a problem in multiple dimensions and solve them through optimisation approaches [68, 76, 80, 167]. An optimal policy can increase the allocation rate of a cluster, which in turn increases resource utilisation [229], boosting financial sales (i.e., able to sell more rentable compute resources by harvesting unused resources. For instance, Spot Instances [10] and Harvest VM [11]) and cost efficiency; thus, the cluster scheduler is one of the most important components within the resource manager.

Operating a large-scale cluster of machines and scheduling tasks comes with many challenges. These compute clusters require a lot of power to operate [124, 132, 200, 244], encounter failures [49, 140, 163, 203], have to abide by agreements set between cluster operators and users [112, 179], and have to consider the effects of different hardware generations [16, 51, 268].

2.4.2.1 Hardware Heterogeneity

Compute clusters are created by purchasing the underlying hardware in bulk to benefit from economies of scale [16]. In practice, these compute clusters have various generations of hardware due to long lifetime and application demands [51, 95, 147, 162, 164]. For instance, Microsoft recently indicated their compute clusters currently consist of twenty generations of hardware, thus requiring automated tuning of applications to achieve satisfactory performance [268]. Similarly, when scheduled tasks depend on each other, stragglers (slow workers) manifest from tasks scheduled onto older generation hardware [268]. Thus, it is important to be aware of the underlying hardware when scheduling tasks.

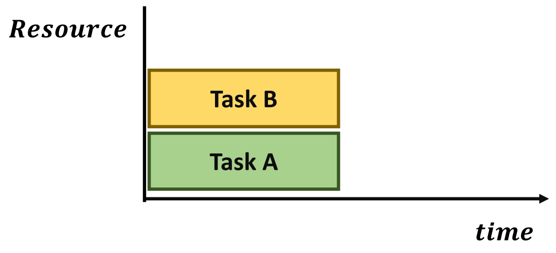
2.4.2.2 Resource Utilisation

A resource manager's key objective is to make high-quality assignments of tasks to improve cluster resource utilisation [229], i.e., binding a task to an appropriate node for the task execution. A core element of the resource assignment decision is making sure the candidate machines have adequate resources to run the tasks. Formally, this can be expressed as for each machine m, the resource type r_i, the requested resources of task t_j must be able to be adequately accommodated by the machine capacities C_m, $\forall r_i \in \mathcal{R}, \sum_j \sum_i t_j \times r_i \leq \sum_m C_m$. *Resource allocation* is a metric for the allocated resources, and one of the goals of a cluster resource manager is to increase the resource allocation rate to improve financial sales [83, 118], however, a high resource allocation rate does not necessarily mean high resource utilisation.

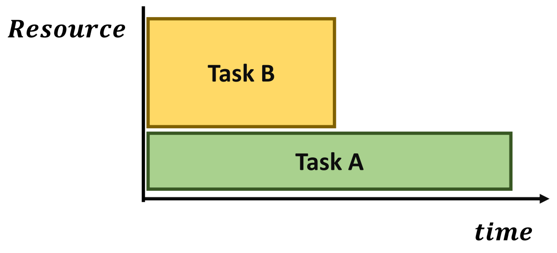
Resource utilisation is a key metric to measure whether resources are used effectively for task execution. The resource utilisation metric can be measuring any resources available such as power, CPU, memory, disk I/O, and accelerators. For instance, a task can request 4 CPUs, but the majority of the task execution may only ever utilise 1 CPU core. Therefore, underutilised machines, i.e., commonly defined as machines that are utilising <80% of their resources [229], contribute to undesirable financial and operational expenses. It is often undesirable to utilise machines to the maximum capacity due to unstable performance, lower system availability, and thus violating SLAs [41]. Resource underutilisation often occur due to application engineers do not necessarily understand their application performance on different generations of hardware and their resource consumption; therefore, it is difficult for engineers to estimate the resources required to execute their tasks, leading to overestimate resource requirements [7, 52]. In addition, cluster operators may reserve machines to account for machine failures [234], peak request loads [52] and power delivery interruption [167]. As a result, compute clusters usually operates at low CPU utilisation, ranging from 20% to 60% [7, 226, 229], which greatly reduces cost-effectiveness. A common approach to improve cluster resource utilisation is by assigning multiple tasks onto the same physical hardware, this is called *co-location*.

2.4.2.3 Co-location & Interference

Co-location can improve resource utilisation in large-scale compute clusters, thus boosting cost efficiency, and power usage ratios [51, 124, 132, 167, 234]. However, interference (performance degradation) arises when multiple tasks are contending for resources [51, 66, 148, 263], leading to ineffective execution of tasks, prolonged task execution time, and security risks [160]. Figure 2.15 shows the comparison between safe and unsafe co-location.



(a) Safe co-location without interference.



(b) Unsafe co-location, interference manifest due to resource contention.

Figure 2.15: Example of safe and unsafe co-location.

Unfortunately, achieving safe co-location of tasks is difficult due to task QoS constraints [31, 247] and hardware heterogeneity [51, 52]. To achieve safe co-location, it is crucial to understand how resources are shared between tasks. Fundamentally, interference is a system phenomenon in resource sharing computing systems, and can manifest in any component of a computing system, ranging from power [132] to hardware resources [29, 51, 66, 160, 263]. Particularly, the following resources often gained attention in system performance optimisation:

CPU & Memory. One of the primary shared resources in machines are CPU cores in one or more CPU sockets; resource contention in CPUs arises when tasks do not acquire the necessary number of CPU cores for execution, one of the primary causes of which is workload spiking (i.e., a particular task suddenly needs to consume all the resources due to computational demand) [51, 66, 263]. The introduction of hyperthreading [242] in CPUs also introduced more risks of interference due to CPU dividing physical core resources (e.g., the instruction micro-op queue) between tasks, when tasks are running on sibling cores [66]. The same form of interference can also happen in physically shared resources such as L1/L2 caches [169], memory bandwidth [159] and Last Level Cache (LLC) [104, 148]. For instance, LLC interference manifests when many tasks are reading and writing data to the shared LLC, causing data needed by a task to be evicted quickly, leading to cache misses when the task requests the now evicted data.

GPU & PCIe bandwidth. As of the writing of this thesis, the majority of GPUs do not provide fine-grained physical resources partitioning except, for example, the NVIDIA A100 GPU [170], i.e., the ability to physically (spatially) share the GPU hardware resources for task execution instead of time-sharing. Therefore, for the earlier GPU hardware generations, the primary reason for interference is queuing latency for accessing GPU hardware resources such as the processing elements, on chip memory bandwidth and the PCIe interconnects [29, 138, 247, 251]. PCIe bandwidth interference manifests similarly when bandwidth is not sufficient for frequent and high volume of data transfer between the host machine and the PCIe attached GPUs [29].

Table 2.2 shows examples of different levels of interference between two DL workloads across two GPU architectures.

Model	Batch Size	Device	Isolation Time	Co-location Time	Slowdown
VGG19	64	**V100**	350s	487s	1.39X
MNASNet 1.3	64	**V100**	227s	281s	1.24X
VGG19	64	**2080**	556s	688s	1.24X
MNASNet 1.3	64	**2080**	262s	302s	1.15X

Table 2.2: Examples of DL workloads interference when co-locating MNASNet and VGG19 on GPUs. Each workload was set to train for 10 epochs.

Naively co-locating tasks can lead to severe interference and hence performance degradation. Matching suitable pairs of tasks and respecting specific resource requirements is difficult [31]. For example, Zhang et al. [263] co-locates offline processing tasks with user-facing tasks and discovered interference can cause performance degradation up to two times. Mystic [230] observed interference stemming from co-locating two tasks from thirty plus applications can range from 45% to more than 250% on GPUs. Thus, resource managers need to consider the possible effect of interference, understand the tasks' resource access patterns to determine the acceptable performance degradation level and schedule tasks accordingly [31, 52].

2.4.2.4 Interference-aware Resource Managers

Interference is an important and fundamental problem to be addressed in compute clusters [31, 51, 107, 148, 231]. There are two types of data-driven approaches (see Table 2.3) that alleviate interference at scheduling time (i) reactive – involves profiling, detecting, and action, and (ii) proactive – leverage historical data to make predictions without profiling and detection.

Resource Manager	Resources	Approaches	Assumptions
Bubble-Up [148]	Memory	Reactive, stress testing.	Isolated stress testing.
CPI^2 [263]	CPU, Cycles Per Instruction	Reactive, statistical outlier detection.	Repetitive Jobs.
Whare-Map [146]	CPU, Instruction Per Second	Reactive, analytical model, random trial-error.	Repetitive Jobs.
DejaVu [231]	vCPU, vMemory, Hardware performance counters	Reactive, classify tasks into classes, monitor performance ratio.	Isolated monitoring (days to week).
Smite [265]	Memory, CPU Functional Units	Reactive, stress testing, analytical model.	Profile both co-located applications.
Paragon [51]	Memory, CPU	Reactive, ML classification after profiling.	Profiling duration of ∼4 mins, Workload phases do not deviate.
Heracles [136]	Memory, CPU, Network, Power, Disk	Reactive, feedback-based control.	No significant change in workload.
Janus et al. [107]	CPU	Reactive, statistical estimation.	all tasks have same priority & All machines have equal capacities.
PARTIES [31]	CPU, Memory, Disk, Network	Reactive, stress test workload, reactive co-location	Fine grained monitoring.
Resource Central [41]	CPU	Proactive, predict $P95^{th}$ CPU utilisation.	Task utilisation correlated to users.
FIRM [188]	CPU, Memory, Network, Disk	Proactive, RL to control resources.	Offline simulation available to train RL model.
Mystic [230]	GPU	Reactive, ML classification after profiling.	Profiling duration of ∼10 seconds.
KubeKnot [227]	GPU	Reactive, online modelling of running tasks future usage.	Task usage does not fluctuate.
Phull et al. [184]	GPU	Proactive, discrete time graph of access patterns.	Repetitive Jobs.
Xu et al. [250]	GPU	Reactive, collect features of each job and predict interference levels to co-locate VMs.	Homogenous GPUs.

Table 2.3: Overview of interference-aware cluster resource manager approaches.

Reactive, CPU-Based. Bubble-Up [148] carefully stress tests an application with potential co-runners, deriving a memory pressure score to drive co-location. DejaVu [231] profiles an application for at least a day to a week, then leverages unsupervised learning to cluster tasks to determine resource allocation for each task class, driving co-location scheduling decisions. Smite [265] characterises application individually by stress testing and leveraging linear regression to predict the tail latency of the target application when co-located to drive co-location decisions. Similarly, Paragon [51] combines both hardware heterogeneity and interference slowdown into a machine learning problem, it first profiles incoming tasks for a few minutes, uses Singular Value Decomposition (SVD) to perform collaborative filtering to identify similarities between tasks to drive scheduling decisions. Heracles [136] and PARTIES [31] characterised interference on multiple resources such as CPU, network, Disk, and memory, and developed a feedback-based heuristic algorithm to drive co-location decisions. Janus et al. [107] conducted statistical analysis on Google Cluster data [194], realising CPU requirements distribution vary significantly. Yet, the high percentiles of total CPU requirements can be approximated reasonably well by a Gaussian distribution; thus it is sufficient to co-locate tasks based on the distribution prediction, to increase resource utilisation and achieve satisfactory performance.

Reactive, GPU. Similar to Paragon [51], Mystic [230] profiles GPU tasks in an isolated machine to extract specific GPU hardware features such as Stream Multiprocessor (SM) efficiency, texture cache hit rate and reads to feed into SVD, in order to derive an interference score, driving co-location decisions. Kube-Knot [227] predicts GPUs resource consumption based on execution patterns after tasks have been executing for some time to drive live migration and co-location decisions.

The above works explore offline analytics, online profiling and reactively mitigate interference and maintain high resource efficiency. While these works are effective, now that application behaviour, resource consumption and cluster scheduler decisions are recorded and collected, resource management techniques can become even more effective by being able to predict the value for relevant metrics of interest [19]. Prior history may be an accurate predictor of the future behaviour for periodic tasks, tasks with well-defined behaviour and cluster characteristics.

Proactive, CPU-Based. Resource Central [41] identifies and extracts relevant key features (of up to hundreds) of submitted jobs, using ML algorithms to predict average and 95th-percentile CPU utilisation to co-locate tasks for improving resource utilisation. FIRM [188] leverages offline simulation and labelling in order to train (i) Support Vector Machine (SVM) models for classifying tasks for resource re-provisioning, and (ii) individual RL model for candidate tasks in order to scale up/down based on estimated interference levels.

Application-agnostic. While the above interference-aware cluster resource managers are effective given sufficient time for online profiling to obtain metrics of interest. They are mainly application-agnostic, i.e., they do not leverage any application-specific knowledge that is readily available in the job submission request or access job-specific application performance metrics. For example, Decima [144] is a cluster scheduler for large-scale data-processing, Spark [255] specifically; it extracts information from the job's DAG at submission time and leverages RL to learn to schedule tasks. Sharad et al. [207] proposed an approach to leverage historical serverless function's idle time to build a predictor to forecast the idle time for scheduling functions, resulted in less resource idleness and thus cost savings. Hence, it is beneficial for cluster resource managers to leverage application-specific domain knowledge for tasks scheduling.

Scheduling algorithm type. The majority of the interference-aware resource manager scheduling algorithms are based on heuristics, with the exception of FIRM [188], which leverages RL to automatically adjust resources when interference is detected. For example, Resource Central [41] avoids interference by heuristically avoiding over-subscribing the CPUs, using the predicted 95-th percentile utilisation. Xu et al. [250] heuristically minimises interference by packing jobs on least-loaded GPUs. PARTIES [31] derives a heuristic feedback-based algorithm to adjust the application resources. Smite [265] estimates the performance slowdown after profiling and heuristically migrate jobs that can caused high interference. Paragon [51] samples servers and derives a co-location score to select the best servers for each incoming job. There is a reason why resource managers mainly leverage heuristic-based algorithms instead of optimisation algorithms such as integer linear programming, and network-flow; this is due to the interpretability of the results and lower computational costs. Therefore, interference-aware resource managers should opt for an approach that is easily understandable and computation efficient.

Numerous data-driven methods are presented, and their common goal is to utilise resources efficiently, and able to mitigate performance degradation stemming from interference. However, there is a lack of accelerator-based, specifically, GPU cluster resource schedulers that are proactive and able to mitigate interference at scheduling decision time. Furthermore, application-aware cluster resource management can bring additional benefits to general cluster resource managers due to domain knowledge on data inputs, computation, communication patterns and jobs characteristics [109, 140, 161, 162, 187]. In recent years, there have been numerous resource management approaches developed for deep learning training clusters.

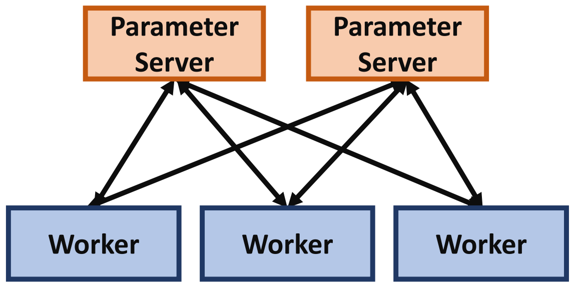
2.4.3 Deep Learning Training Clusters

Large-scale DL training in compute clusters requires distributed machines to frequently communicate due to the iterative nature of training with large-scale datasets. To minimise communication overhead and provide accelerators for computation, compute clusters specifically built for DL training are now common practice [17, 81, 95, 108, 113]. This thesis defines DL training clusters as compute clusters specifically for DL training.

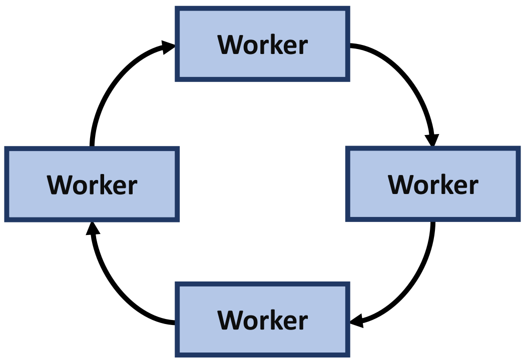
In a recent production DL training cluster study, Hu et al. [95] indicates the majority of the jobs are single-GPU ($> 50\%$) and their average GPU utilisation can range from 10% to $\leq$50%. A potential reason to the phenomena could be due to the application constraints requiring DNN models to be small, in the region of $\leq$10GB. Therefore, training a lot of small models on a single GPU on many machines is sufficient during early development. Additionally, evaluating a set of hyperparameters due to newly generated data [95] could be another potential reason for the many single-GPU jobs. For large DL models ($>$10GB), training models across many machines can accelerate training and in general have higher GPU utilisation when compared to jobs using a single GPU, with utilisation ranging from 40% to $>$60% [95, 108]. Both categories of jobs are equally important in producing a high-quality DL model.

2.4.3.1 Distributed Training

The time needed to train a DNN model using a single machine can range from minutes, hours, or days to months; it is now a common practice to deploy tasks across many machines to accelerate models training [50, 108]. In particular, there are various distributed learning paradigms, data parallel, model parallel, and pipeline parallel.



(a) Parameter Server and Worker strategy [50]



(b) Ring ALLREDUCE strategy [72]

Figure 2.16: Data parallel training strategies.

Data Parallel tasks are defined as each task has its own copy of DNN model and performs training in isolation. There are usually two training strategies in data parallel training, where the gradients are communicated in centralised or decentralised fashion, as shown in Figure 2.16. In centralised strategy, where the gradients g calculated by each task typically are sent to one or more dedicated master tasks (parameter server) synchronously or asynchronously, and perform statistical optimisation to get the processed gradients $\hat{g}$ [4, 50, 168]. Finally, the processed gradients $\hat{g}$ are broadcasted back to each training task in order to update the local model. In decentralised strategy, there are no dedicated master tasks to communicate gradients, and training tasks communicate with one another partially or fully to obtain $\hat{g}$. A popular implementation of decentralised strategy is Ring ALLREDUCE, where each training task communicates with its right most neighbour in a ring, after two passes of the ring, each training task will have obtained the full set of gradients.

Major bottlenecks in Data Parallel training are stragglers [27] and communication overhead [72, 109]. Communication overhead scales with the number of tasks participating in the distributed training due to bandwidth used when communicating g. This is further complicated in shared multi-tenant clusters where network

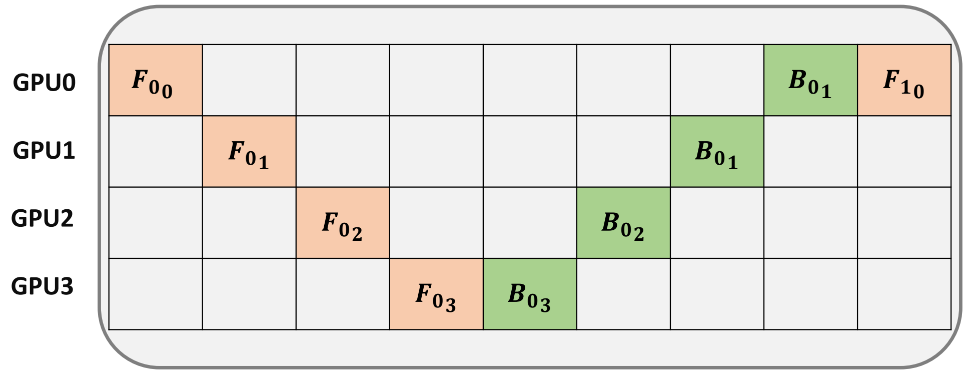


Figure 2.17: Model Parallelism, F_{ij} and B_{ij} indicate the forward and backward phase at ith step with jth stage (model part).

traffic is high. The system research community focuses on alternate hardware [81, 113], topology arrangement [9, 165] and communication procedures [55, 209, 262] to tackle the aforementioned problems, whereas the machine learning research community focuses on statistically efficient optimisation to side step communication overhead [111, 253]. As recent research suggests, larger models can express and are able to learn complicated functions, which poses more memory resource requirements and consequently leads to hardware being unable to host a single large model [50]. This is addressed via model parallelism.

Model Parallel tasks are defined as each task occupying a part of a DNN model, which is then executed in stages, illustrated in Figure 2.17. This is useful when total memory consumption for a model exceeds the capacity of the accelerator, it is therefore necessary to split the model from n into k parts for training. The amount of data communicated between machines is much smaller than with the data parallel counterpart, as only intermediate activation and gradients are communicated. The major downside of this paradigm is that at any given time, only the parts that are ready to execute can be active, leading to machines that host the other $n - k_{active}$ tasks idle, degrading cluster resource utilisation [161]. Second, the burden of deciding how a model should be partitioned falls on engineers, which often is suboptimal and time-consuming [6, 157]. One solution is to leverage pipeline parallelism.

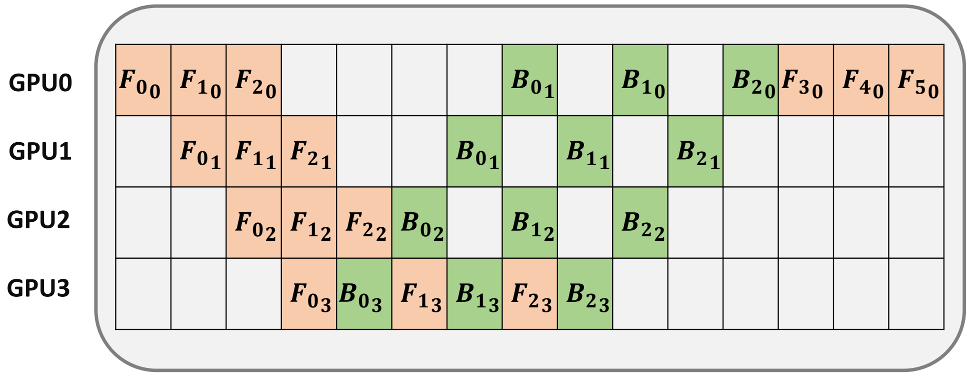

GPU0
F_{0_0} F_{1_0} F_{2_0} B_{0_1} B_{1_0} B_{2_0} F_{3_0} F_{4_0} F_{5_0}
GPU1
F_{0_1} F_{1_1} F_{2_1} B_{0_1} B_{1_1} B_{2_1}
GPU2
F_{0_2} F_{1_2} F_{2_2} B_{0_2} B_{1_2} B_{2_2}
GPU3
F_{0_3} B_{0_3} F_{1_3} B_{1_3} F_{2_3} B_{2_3}

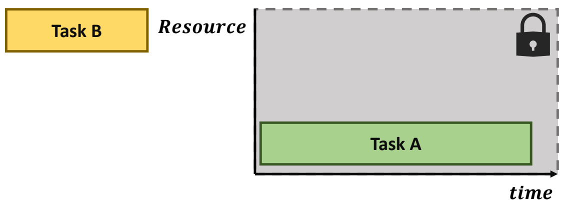
Figure 2.18: Pipeline parallel strategy.

Pipeline Parallel is defined as executing operational stages independently, as shown in Figure 2.18. Similar to model parallel, pipeline parallel execute different stages of a model, but overlapping the communication and computation with different inputs, i.e. operating on different batches of data and parameters; this technique can effectively addresses the problem of machines being idle waiting for an iteration to be completed. In the extreme case of only one stage, the pipeline training becomes model parallel training [97]. For large models, pipelining can scale to more machines in comparison to both data and model parallel. However, naive pipelining has to carefully accounted for stale parameters, this is because the B_t batches executing are using $t - m$ parameters, leading to stale gradient updates, and could hinder model learning. Additionally, pipelining requires careful placement of the model stages on machines similar to model parallelism, this is because stages executing at different speeds can lead to stragglers; while excessive numbers of model stages can lead to over communication, both problems can lead to underutilisation.

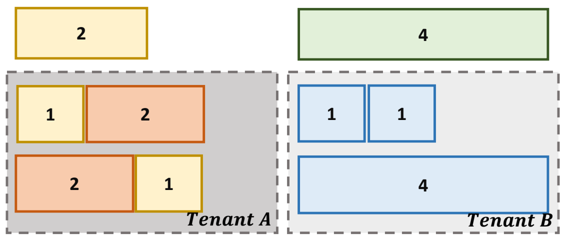
2.4.3.2 Resource Management Limitations for Deep Learning

As with traditional compute clusters, DL training clusters require a resource manager to assign tasks to machines, ensure fairness and improve cluster resource utilisation. However, there are challenges in leveraging existing cluster resource managers such as Kubernetes [24], YARN [233] and Mesos [90], for DL training clusters.

Exclusive Lock & Coarse-grained Cluster State. At the time of writing, current cluster resource managers are not built to cater for DL training and considering the actual load in clusters [227, 236]; there is no explicit sharing mechanism for DL training jobs in many existing DL training clusters, including co-location [95, 108, 227, 246]. Note that, currently, NVIDIA A100 GPUs do provide Multi-Instance GPU technology for hardware-level physical resource sharing, however many existing DL training clusters do not yet contain these newly released GPUs [95, 108, 247]. Although training jobs are often compute and communication intensive, there are many reasons for dedicated GPUs training jobs to experience underutilisation [108]. For example, inefficiencies in data processing code [247] and exclusive locks on GPU resources [95, 246], restricting co-location. Analysis by Microsoft [108] has shown that average GPU utilisation is around 55%, and this could be due to dedicated GPU training; Similarly, Alibaba [247] reports an average of 15% GPU utilisation. Figure 2.19a illustrates the entire GPU resource is dedicated to one task only.



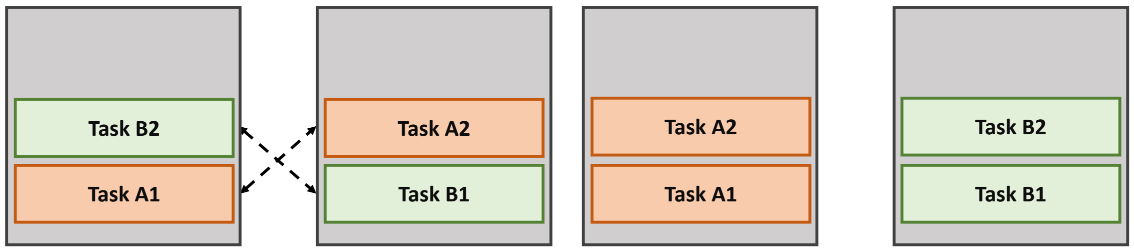
(a) Exclusive lock of GPU resource.



(b) GPUs fragmentation & Multi-tenancy.

Figure 2.19: Illustration of exclusive lock and resource fragmentation in DL training cluster.

Placement Sensitivity & Locality. Since DL training requires large amount of data and its training process is iterative, the number of jobs situated on a machine matters (see Figure 3.8), as the sharing of resources is not only limited to GPUs but to network, PCIe bandwidth and disk I/O [95, 108, 142, 246]. Additionally, jobs that are spread across machines perform worse due to locality. For example, instead of eight DL training tasks spread across eight nodes, each occupying a GPU, it is often desirable to place them all on one node with eight GPUs [81, 266], a simple example is shown in Figure 3.8. DL training jobs can specify locality requirement during job submission time, however, if there is no satisfactory condition, the job must wait, contributing to high waiting time, low cluster utilisation and resource fragmentation.



(a) Tasks have to communicate across machines causing performance degradation.

(b) Tasks only communicate internally.

Figure 2.20: Illustration of the importance of locality within DL training cluster.

Multi-tenancy & Resource Fragmentation. Compute clusters are usually multi-tenanted, providing services for many organisations or teams. Multi-tenant DL training clusters typically have more users submitting jobs than the clusters' resources can execute [81, 95, 108, 246, 266]. In production cluster analysis at Microsoft [81, 108, 246], job waiting time (*starvation*) can range from tens of minutes to hours due to exclusive locks and long execution times. This is further complicated when jobs have diverse resource requirements and each tenant reserves resource quotas with specified limits, i.e., the number of devices (CPUs and GPUs) and memory [95, 266].

For instance, Zhao et el. [266] identifies that a tenant's DL training jobs have to wait significantly longer for GPUs than they would in a private cluster of size at least equal to the resource quota. Even when there are enough resources for a particular tenant, it is desirable to reserve adequate resources in the case of job's submission burst and maintain fairness, lead to low utilisation.

As a result, cluster resource managers are proposed that are specifically tailored to DL training clusters, aiming to optimise DL training jobs performance, achieving faster turn around time and increasing cluster resource efficiency.

2.4.4 Deep Learning Training Cluster Resource Managers

DL training cluster resource managers are proposed with varying motivations to address the aforementioned challenges in Section 2.4.3.2:

Job Completion Time (JCT) & Locality. Speeding up DL JCT to effectively increase system throughput and reduce job waiting time. Many DL-aware cluster schedulers adopt approaches that aim to leverage DL-specific workload characteristics to optimise the training progress. For instance, Optimus [182] accelerates training by adaptively scaling tasks (parameter servers/workers) based on fitted regression to training progress and improve locality by consolidating tasks evenly on each machine, with regard to the type of the task. Amaral et al. [9] proposes a topology-aware approach to minimise cross-socket and cross-machine communication to mitigate communication latency. Tiresias [81] and ByteScheduler [181] speed up JCT by understanding the communication patterns of DL training jobs, proposing network-specific prioritisation approaches. KungFu [142] adapts the batchsize, communication strategy and number of workers based on monitored performance metrics such as throughput and gradient noise scale [149], to speed up JCT. Pollux [187] derives an analytical model, based on gradient noise scale [149], to tune the batch size and leverage genetic algorithms to migrate jobs to improve JCT. Tiresias [81] improves locality by consolidating jobs with large communication demands. Gavel [162] and Themis [141] on the other hand speed up JCT by improving the selection of underlying hardware leveraged for training jobs.

Fairness. Improving fairness can be achieved by one or more of the following, allocate equal resource share, timeshare, waiting time and equivalent computation. HiveD [266] adopts the buddy allocation algorithm to manage multi-tenant fairness, ensuring tenants can request GPUs when they need them (equal resource share). Gandiva [246] schedules jobs in a round-robin manner to ensure fairness for each job and in the hope that short jobs finish as soon as possible; improving job turn-around time for quick DNN architecture experimentation (equal time on resources). Tiresias [81] leverages a multi-level feedback queue with the least attained service to promote jobs with high waiting time (equal waiting time). Gavel [162] and Themis [141] maintain fairness by switching hardware when they experience slowdown due to hardware and machine assignment (equivalent computation).

Resource utilisation. Improving resource utilisation in DL cluster is important. One effective means to do so is by making modifications to existing DL cluster resource manager frameworks to enable co-location. This is especially important given the current state of GPU utilisation is low in multi-tenant DL compute clusters. Gandiva [246] is the first DL training cluster scheduler to enable co-location; however, it reactively monitors the training progress of a job and migrates the job after detecting performance interference. Antman [247] introduces a DL GPU operation manager to control co-located jobs execution; the manager sits between the application and device driver to allow effective throttling of lower priority jobs when interference is detected between co-located jobs. Both the Gandiva and Antman approaches require modification of the DL frameworks, which is of high maintenance cost.

Scheduler	Objectives	Approaches	Assumptions & Evaluations
Amaral et al. [9]	JCT, Locality	Reactive, analytical model for communication cost and utility.	Jobs' graphs available. | ⋆ – 2 GPUs, 6 models. | ♣
Optimus [182]	JCT, Locality	Reactive, add or remove task based on predicted training progress via regression model learned online.	Modify DL Frameworks for job progress. Data parallel jobs only. | ⋆ – 12 GPUs, 4 models. | ♣
Gandiva [246]	Fairness, Resource utilisation	Reactive, enable co-location, migrate jobs when progress slows down.	Modification of DL Frameworks. Profile jobs in isolation. | ⋆ – 180 GPUs, 10 models. | ♣
Tiresias [81]	JCT, Locality, Fairness	Reactive, consolidate jobs to maximise locality via network profiling and promote jobs based on the Gittins index [3].	Profile jobs in isolation. Data parallel jobs only. | ⋆ – 60 GPUs, 10 models. | ♣
ByteScheduler [181]	JCT	Reactive, prioritise jobs with small tensors size to speed up communication.	Integrate a new communication layer for DL frameworks. Modification of DL frameworks and training code. | ⋆ – 128 GPUs, 3 models. | ♣
Themis [141]	JCT, Fairness	Reactive, exchange hardware type and number of resources with respect to fairness between jobs.	Modification of DL frameworks. | ⋆ – 64 GPUs, 11 models. | ♣
Antman [247]	Resource utilisation	Reactive, throttle low-priority jobs via DL frameworks.	Modify DL frameworks to coordinate with cluster scheduler. Profile jobs in isolation. | ⋆ – 64, 5000 GPUs, 9 models. | ♣
HiveD [266]	Fairness	Reactive, leverage buddy allocation algorithm to improve fairness due to resource fragmentation.	– | ⋆ – 96, 2232 GPUs, 11 models. | ♣
KungFu [142]	JCT	Reactive, scale batchsize, communication strategy and workers based on monitored performance metrics.	Modification of training code.| ⋆ – 32/16 GPUs, 4 models.
Pollux [187]	JCT	Reactive, scale task based on analytical model and meta-heuristic algorithm; tune batch size of each job.	Modification of training code. | ⋆ – 64 GPUs, 6 models. | ♣
Gavel [162]	JCT, Fairness	Reactive, exchange hardware type based on system throughput.	Modification of training code. | ⋆ – 36 GPUs, 7 models. | ♣

Table 2.4: DL training cluster resource managers and their design goals.

⋆ and ♣ denote empirical system and large-scale simulation evaluation, respectively.

2.4.5 Discussion

Many DL training resource managers focus on accelerating training, but there is lack of support for improving resource utilisation and fairness. They are all reactive, i.e., decisions are based on online monitoring and profiling in isolation, which ties up GPUs and reduces resource efficiency.

Gandiva [246] and Antman [247] are the only resource managers designed to improve resource utilisation as one of their main objectives. They improve resource utilisation by modifying existing resource management and DL frameworks to allow sharing of GPUs. However, interference manifests as a result of GPU sharing. Gandiva [246] reactively mitigates interference by migrating tasks after interference is detected. Antman [247] reactively throttles low-priority tasks similarly upon interference detection. Importantly, both approaches require online monitoring and profiling in isolation on dedicated GPUs before deployment to obtain correct utilisation patterns or GPU kernel patterns, which does result in overheads.

A few DL training resource managers exploit DNN model characteristics to accelerate training [9, 81, 181, 187], usefully DNN model characteristics may be able to provide hints on resource utilisation, allowing DL training resource managers to make proactive scheduling decisions; however, there is a lack of analysis between DNN model characteristics and heterogeneous GPU utilisation. We focus on this potential in Chapter 3. None of the existing DL training cluster schedulers address the hardware heterogeneity, co-location, and interference challenges in a proactive manner while exploiting the intrinsic features from the DNN models. Yet, in order to improve DL training cluster GPU utilisation, makespan and job waiting time, an application-aware cluster scheduler should:

- understand the utilisation relationship with the submitted DL model, in order to predict the utilisation of a DL model likely to have given the hardware.

- avoid interference between co-located DL jobs that share the underlying hardware by co-locating suitable jobs that likely to fit the hardware and execute well together; and
- improve fairness by scheduling DL jobs likely to finish quickly while avoiding large DL job starvation.

2.4.6 Summary

As discussed in Section 2.4.2, efficient usage of cluster resources requires scheduling decisions that consider application characteristics and the underlying infrastructure. Within the same section, the thesis also explored how proactive scheduling approaches can be beneficial to cluster schedulers making high-quality decisions at resource assignment scheduling time in comparison to reactive approaches. This section has surveyed many existing DL training cluster schedulers that adopt various techniques to improve DL training jobs performance. The section started with the current challenges in DL training compute clusters (§ 2.4.3.2), and then discussed the relevant cluster schedulers (§ 2.4.4) that focus on one or more of the following: JCT, Fairness, and Resource utilisation. Finally, we identified the need for a co-location enabled, proactive interference-aware, DL-specific training cluster scheduler that understands the hardware consumption patterns, make predictions and maximises resource utilisation. To achieve this, we need to understand the relationships between co-located DL training workload and interference.

In Chapter 3, the thesis presents analysis that enables understanding of the relationship between DL models and GPU utilisation; a large-scale production GPU DL training cluster analysis that provide motivation for the need of a DL training cluster scheduler that is proactive and interference-aware. In Chapter 4, the thesis presents Horus, a proactive DL training cluster scheduler that support these goals.

Chapter 3

Towards a Proactive Approach to DL Training Scheduling

This chapter presents analysis and justification for a proactive approach in DL training compute clusters. First, to illustrate that DL training cluster resource management challenges exist, we present an analysis of DL workload data (§ 3.1). We show that low GPU utilisation persists across existing large-scale industrial production DL training compute clusters. Section 2.4.2.3 explains why co-location is important to boost average cluster GPU utilisation; we then ascertain the relationship between GPU utilisation and interference for co-located DL jobs by conducting a series of co-location experiments(§ 3.2). Heterogeneous GPUs are also presented within the co-location experiments, identifying the effects of hardware generation to GPU utilisation (§ 3.2.1). To understand the relationship between GPU utilisation and DL model characteristics, we conduct an in-depth analysis between the features that can be extracted from the DL models' IR (§ 3.3). We observe that DL model characteristics have a positive correlation with GPU utilisation and GPU utilisation is also correlated with interference levels.

3.1 DL Training Cluster Underutilisation

In this section, we analyse a production DL training cluster and observe that GPU underutilisation exists and is a systemic issue across production DL clusters with similar scale. Additionally, we observe that a large number of DL training jobs in production DL clusters are single-GPU, presenting opportunities for co-location.

3.1.1 Cluster Characteristics

Cluster Environment. The cluster studied is a production DL training cluster, which we refer to as Apollo. At the time of the study, Apollo is a 500+ GPU cluster, with each machine containing 4 GPUs (NVIDIA V100 or NVIDIA P100). Apollo serves multiple teams in the research department of a major Internet company, including both research and production engineers. Each team has its own resource quota. When the demands of a team increases, new machines are purchased and allocated to the team's quota. Apollo supports various applications in the DL training pipeline, e.g., data preprocessing, model training and hyperparameter tuning as outlined in Section 2.3.2. These jobs are submitted by research groups exploring and developing viable commercially valuable models. The majority of the models fall into the DL domain of computer vision. Jobs submitted include both single-GPU and multi-GPU jobs; for multi-GPU jobs, distributed training strategies discussed in Section 2.4.3.1 are leveraged, utilising PyTorch [178] and TensorFlow [4].

Job Management Software. Kubernetes [24] is leveraged as the resource manager for Apollo. To interact with Kubernetes, a management software framework is implemented to allow individual users to submit their DL training jobs, similar to Figure 2.14. The management framework allows users to track their experiments and allow administration to manage resource quotas. Resource allocation decisions are

Metric	Unit	Type	Source
Size	per task	discrete	Relational Database
Duration	second	aggregate	Relational Database, Time Series Database
GPU utilisation	percentage	sampled	DCGM, Time Series Database
GPU memory	byte	sampled	DCGM, Time Series Database

Table 3.1: DL jobs metrics collected and their corresponding source in Apollo.

made following three steps: (i) A user submits their job to the management web UI layer, specifying the resource demands, typically by instance type, i.e., pre-configured resource combination. (ii) the management software then polls each team's queue for pending jobs; upon submission to Kubernetes, jobs are tagged with relevant labels such as job name and unique identifier. (iii) Kubernetes allocates jobs to machines following the default policy. After the job is scheduled, it keeps running until completion, terminated by the user, killed by the system, or fails due to errors. No sharing of GPUs or preemption mechanisms are enabled.

Performance Metrics. Since the study's aim is to measure the performance of current DL training resource managers, the metrics of interests are: (i) the number of DL jobs, (ii) the size of DL jobs, (iii) the duration of DL jobs and (iv) the average GPU utilisation for each job, as reported by NVIDIA GPUs.

The dataset [1] was collected via a custom aggregation process that interacts with (i) a time series database component similar to [75] and (ii) a relational database. The time series database contained the actual resource utilisation and memory consumption per GPU, reported from the NVIDIA Data Center GPU Manager (DCGM) daemon [73]. Table 3.1 lists the precise set of metrics collected in Apollo. The relational database contained job request information, such as, number of requested GPUs, and the submitted, start of execution and finish times for the training jobs. For GPU utilisation and memory consumption, we aggregated these per DL task and calculated the corresponding mean average per DL job via the labels attached to the metrics time series.

3.1.2 The Importance of Co-location

To emphasise co-location is important in DL training clusters, we analysed and compared Apollo with previously reported DL training clusters of comparable size and characteristics (see Table 3.2): Helios [95], a production DL cluster in SenseTime; Philly [108], a production DL cluster in Microsoft, where it has been leveraged for various DL training resource manager research evaluation and analysis [81, 187, 246, 247]; and AntMan [247], a production DL cluster in Alibaba Group, leveraged for both public and private use. For the rest of the analysis, we omitted jobs that have average GPU utilisation of 0% in Apollo due to management or machine errors, leading to jobs entering the zombie state.

Requested GPUs. Table 3.2 shows both the average and maximum number of requested GPUs across production DL training clusters. We observe that Apollo's average number of requested GPUs (18.1) are much higher than the other clusters reported in the literature (<5) and have a much lower proportion of single-GPU jobs

[1] The dataset is a month-long trace consisting of 300+ jobs from September 2020.

	Helios [95]	Philly [108]	AntMan [247]	**Apollo**
Trace Duration	6 months	83 days	1 week	**1 month**
Average # of GPUs	3.72	1.75	Not Reported	**18.1**
Maximum # of GPUs	2048	128	Not Reported	**128**
Single-GPU Jobs	∼70%	∼80%	Not Reported	**∼10%**
Average Duration	6652s	28329s	Not Reported	**37494s**
Maximum Duration	50 days	60 days	Not Reported	**12 days**
CDF 50% GPU utilisation	Not Reported	∼50%	∼15%	**∼65%**
Average Queuing Delay	Medium	High	Medium	**Not Reported**

Table 3.2: Comparisons between the Helios, Philly, AntMan clusters and Apollo. Medium and high queuing delay indicate minutes and hours, respectively.

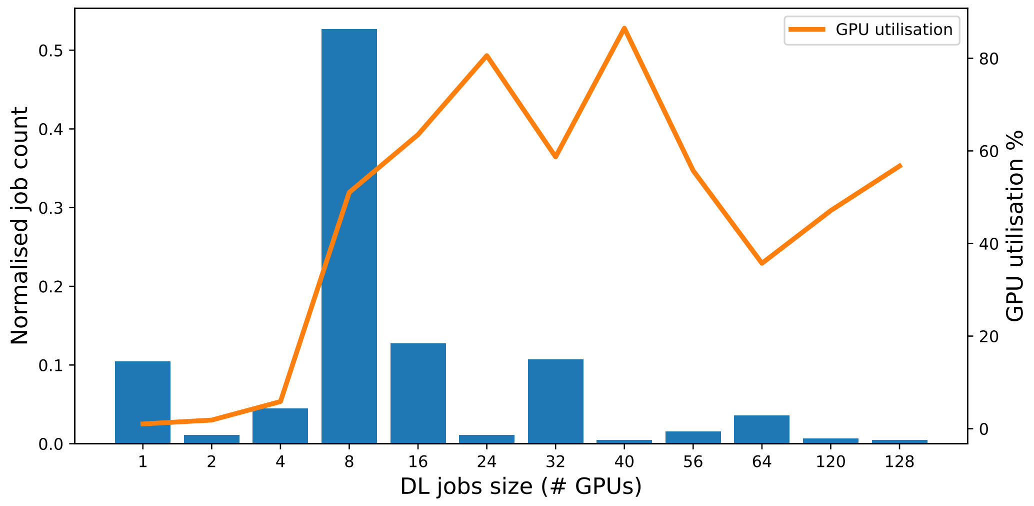
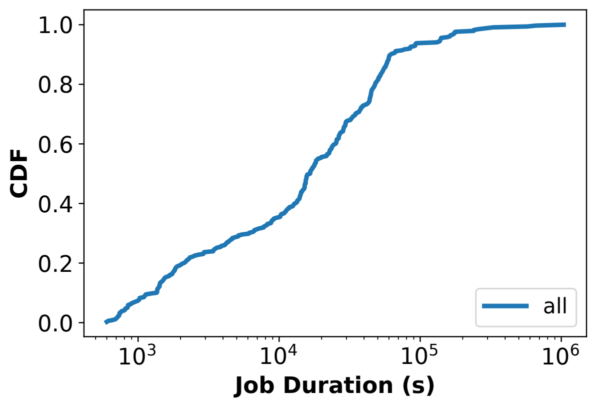


Figure 3.1: Normalised jobs count (%) and GPU utilisation (%) in Apollo.

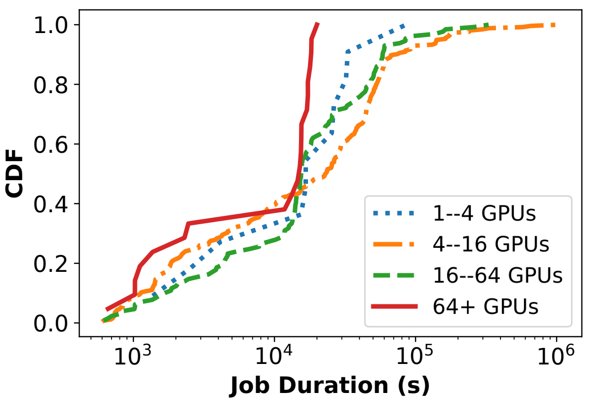
(∼10%), indicating that the submitted jobs mainly consist of distributed training jobs (55% of jobs require 8-GPUs) as shown in Figure 3.1. In contrast, both Helios and Philly consist of ∼70% and ∼80% single-GPU jobs respectively. Potential reasons are (i) more users sharing Helios and Philly, therefore limiting resource usage; (ii) users at the early stages of developing a DNN model architecture, therefore utilising a single-GPU to search for a promising DNN architecture; (iii) users at the final stage of training, hyperparameter tuning – searching for optimal parameters for their chosen DNN architecture. We speculate the differences are mainly due to Apollo's role in serving a research department for innovation and production, in contrast both Helios and Philly that may have restrictions on the resources each engineer can request.

JCT. Table 3.2 shows the average and maximum JCT in Apollo, 37942s and 12 days, respectively. Similar to Philly, Apollo's jobs have longer duration due to its training models from scratch to convergence. One of the reasons indicated in Philly was locality-agnostic scheduling, leading to higher synchronisation time for distributed jobs. The scheduling policy in Apollo did not consider locality, we speculate this is the reason for similar average duration since a large majority of jobs in Apollo were multi-GPUs DL training jobs. In contrast, Helios mainly consists of shorter jobs with an average JCT of 6652s, this is due to the majority of jobs being of program debugging and model evaluation rather than model training [95].

Figure 3.2 illustrates the Cumulative Distribution Function (CDF) of job duration in Apollo. We observe that half of the jobs have a duration of 2.5 hours or lower. In contrast, Helios and Philly traces indicated that half of the jobs have duration ranging from minutes to less than an hour. Our hypothesis is that the shorter duration are due to frequent experimentation, model evaluation or frequent error events as indicated in [108], which may also explains the large number of single-GPU jobs and the JCT distribution. Interestingly, Figure 3.2b shows that jobs in Apollo have different JCT



(a) DL job duration CDF for all jobs.



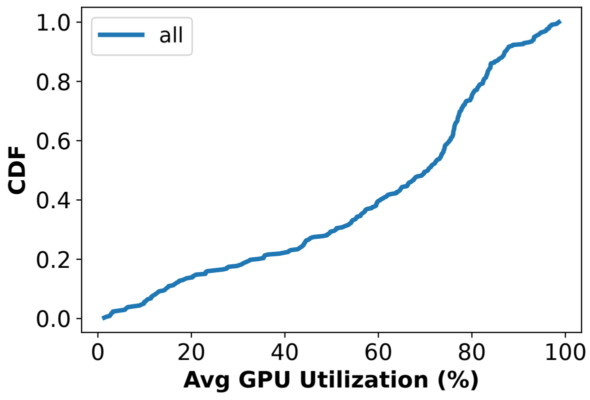
(b) DL job duration CDF for individual job size.

Figure 3.2: DL job duration CDFs in Apollo.

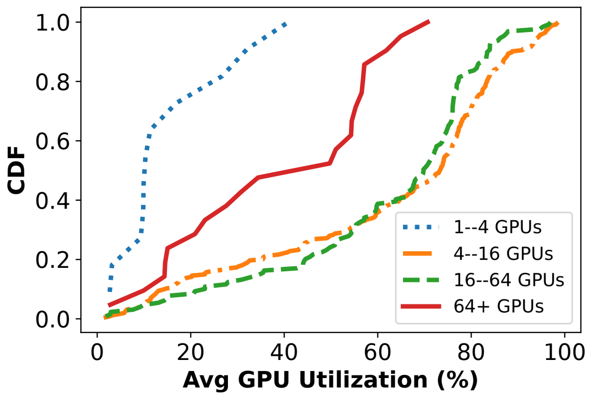
distribution for individual job sizes, where jobs with 64+ GPUs finished earlier than jobs with 4 to 64 GPUs. This is intuitive as data parallel jobs scale with number of GPUs, therefore 64+ GPUs finished training faster than 4 to 64 GPUs. Jobs with 1 to 4 GPUs finished earlier than 4 to 64 GPUs, this is likely as a result of models and hyperparameter exploration, which is often killed or stopped early.

GPU Utilisation. Figure 3.3 depicts the CDF of job average GPU utilisation in Apollo. We observed half of the jobs in Apollo have average GPU utilisation of ≤65%. Half of the jobs in Apollo for both small (1 to 4) and large (64+) DL jobs, both exhibit low GPU utilisation ranging from 15% to 50% respectively as shown in both Figure 3.3b and Figure 3.1. In contrast, AntMan's cluster manages only 15%. The mean GPU utilisation in Apollo is 60% which is slightly higher than Philly's 52%. A potential reason for the low GPU utilisation phenomena is due to exclusive locking of the GPUs, this is mentioned in both Philly's and Antman's analysis [108, 247].

A major reason for such low GPU utilisation in larger jobs is because network communication bottlenecks delayed the actual execution on the GPUs. For medium size jobs (4 to 64), they achieve mean GPU utilisation of 70% or lower. The observations corroborate previously reported statistics where average cluster GPU utilisation is at the lower end (≤60%) [108, 247]. The primary causes of such low average GPU utilisation are due to applications being unable to keep the GPU at high load, coupled with exclusive GPU locking and scheduling without considering locality, further degrading average GPU utilisation as discussed in Section 2.4.3.2.



(a) Average GPU utilisation CDF for all jobs.



(b) Average GPU utilisation CDF for individual job size.

Figure 3.3: DL jobs average GPU utilisation CDFs in Apollo.

3.1.3 Discussion

We summarise the key similarities across large-scale production DL training clusters.

Single-GPU Jobs. We observed that the majority of production DL training clusters comprise many single-GPU training jobs, ranging from ∼70% to ∼80%. Moreover, these single-GPU jobs are often underutilising GPUs, with utilisation ranging from ∼5% to ∼50%. The reasons could be due to the majority of these single-GPU DL jobs being for program debugging and model evaluation. This indicates that cluster GPU resource utilisation can be improved if co-location is enabled, especially for single-GPU DL jobs.

Low Average GPU Utilisation. We observed the average GPU utilisation across DL training compute clusters is lower than 60%. This corroborates the reported limitations of existing resource managers such as Kubernetes [24] and YARN [233], mentioned in Section 2.4.3.2. The main reason being due to application-agnostic scheduling, which does not consider both locality and actual GPU load.

3.2 Improving DL Training Cluster Utilisation Using Co-location

One key strategy for mitigating low utilisation and exploiting the fact that the majority of jobs are single-GPU, is co-location. Existing GPU device managers and DL training cluster resource managers that allow co-location [29, 115, 184, 230, 246, 247, 250] alleviate interference effects, by profiling GPU workload characteristics such as resource access patterns (see Figure 2.11) and GPU utilisation at runtime, to orchestrate GPU kernel scheduling order or to opportunistically co-locate GPU compute jobs respectively. However, it is possible to extend DL job training time from minutes to hours, due to the additional overheads of profiling DL GPU resource access patterns and hardware statistics at runtime. For instance, Xiao et al. [247] indicates that a DL job can go through many phases at the start of the training job and could take up to tens of minutes to reach stable resource access patterns.

Additionally, profiling must be performed for every new job (DL model) submitted to the system, resulting in additional time overhead in the system, further increasing the high job waiting times (see § 2.4.3.2). To optimise the co-location decision for DL training jobs, interference between DL jobs should be analysed. However, there is no in-depth analysis for co-location of DL jobs to date that can enable fast co-location decisions. We therefore make a contribution by conducting an interference analysis and co-location study. The aim of this section is to answer [**RQ1**] and confirm whether there are high-level metrics to be leveraged as a general proxy to estimate the interference level, i.e., JCT degradation, without fine-grained profiling.

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 3.3: Micro-benchmark hardware setup.

● - CV, 🟩 - NLP, 🟦 - FC

Architecture	Permutations
● MobileNet [94]	[default]
● MobileNetV2 [201]	[default, custom(§D.2)]
● MobileNetV3 [93]	[075, 100]
● GoogLeNet [223]	[default]
● ResNet[87]	[18, 18 - bottleneck, 34, 50],
● VGGNet[214]	[11, 11 - bottleneck, 19]
● SqueezeNetV1[103]	[1.0]
● DenseNet[96]	[121, 161, 169]
● ShuffleNetV2[139]	[0.5, 1.0, 1.5, 2.0]
● MNASNet[225]	[0.5, 1.0, 1.3],
● DualPathNetwork[37]	[92, 26], blocks: [2,2,2,2]
● ProxyLess [25]	[cpu, gpu, mobile, mobile-14]
● PyramidNet[84]	depth: [48,84,270], alpha: [66,110]
● ResNeXt[249]	[11,29] cardinality: 2, width: [16,64]
🟩 LSTM [70]	ParaDNN implementation [241](§D.1)
🟩 Gated Recurrent Unit [39]	ParaDNN implementation [241](§D.1)
🟦 Fully Connected (custom)	ParaDNN implementation [241](§D.1)

Table 3.4: Analyzed DL models. Datasets (CV): Cifar10 [122], batch sizes: {8, 16, 32, 64}. (NLP): WikiText2 [154] & News Commentary v14-en-zh [1]. NLP: sentence length: 200, vocab. 10000, default batch sizes: {16,32,64}.

3.2.1 Co-location Study

To answer the above question, we conducted a co-location study of DL systems. A wide range of DL models were selected to be included in the micro-benchmarks that contain both computer vision and natural language processing models. Specifically, we selected 14 CNN, 2 RNN and 1 MLP model (see Table 3.4), which is of comparable size to previous system research [81, 153, 182, 246, 250, 259]. Each model architecture was then further refined into different configurations by varying mini-batch size, hidden dimensions and number of layers, to create a number of model permutations. Within the memory constraints of GPU devices (see Table 3.3), this resulted in 292 unique configurations profiled in isolation with further 638 co-location combinations. We ran these models on our systems until they encountered out-of-memory (OOM). Figure. 3.4 shows an example of 72 DL models.

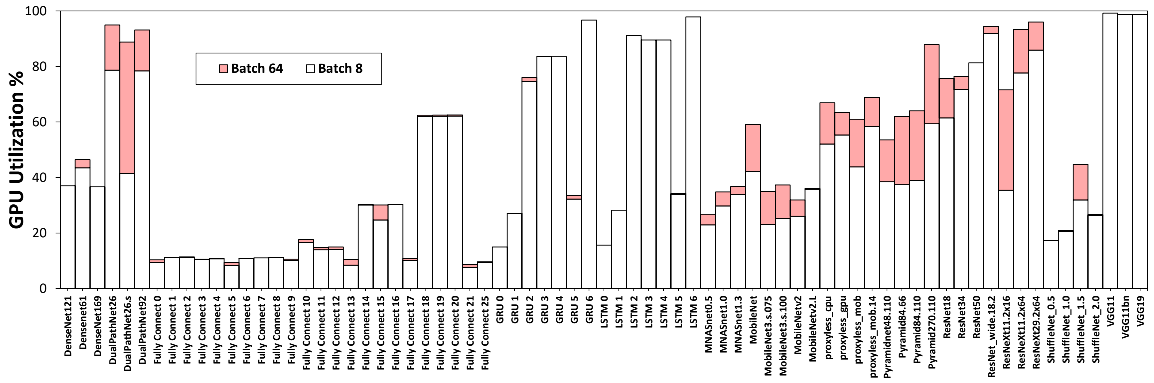


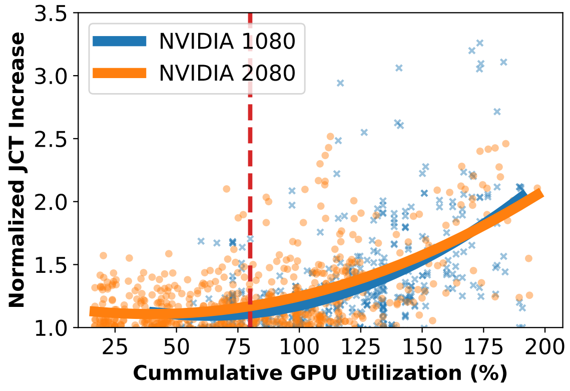
Figure 3.4: Overview of sampled DL workload GPU utilisation differences (NVIDIA RTX 2080).

Benchmarks Environment. Micro-benchmarks were conducted using two different DL systems (A & B), described in Table 3.3. Leveraging methods established in the literature [29, 30, 250], 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 five times to ensure metric consistency. Both systems used an Nvidia container runtime, CUDA Toolkit 10.2 and PyTorch 1.5 DL Framework [178].

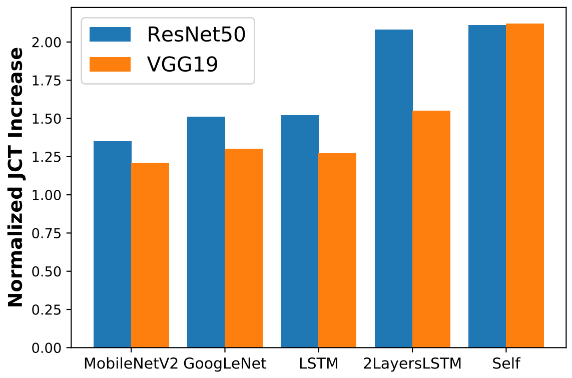
Performance Metrics. To understand the impact of interference, several key metrics of interest are extracted: GPU utilisation, Job Completion Time (JCT) and GPU kernel access patterns. Each DL job is configured to run for five epochs in order to set a baseline for isolated execution and to limit the time spent on training. 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 degradation or slowdown T_{deg} is measured as:

$$T_{deg} = \frac{T_{colo}}{T_{solo}} \tag{3.1}$$

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.



(a) Cumulative GPU utilisation and JCT slowdown.



(b) Co-location impacts on JCT for Resnet50 and VGG19.

Figure 3.5: Co-location study results.

3.2.2 GPU Utilisation as a Proxy for Interference

GPU over-commitment. The results shown in Figure 3.5a indicate that co-located job combinations with increasing levels of GPU over-commitment, i.e., the cumulative GPU utilization requirement for DL jobs greater than 100%, results in a JCT increase of between 1.5x–3x; For instance, co-locating two VGG19 models (each requiring 90%+ utilization in isolation) results in over three times JCT increase as shown in Figure 3.5b. 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. This finding is similar to previous co-location studies in CPU-based co-location enabled cluster resource managers, confirming that *GPU utilisation can be used as a proxy metric for determining job interference levels.* This is intuitive, as GPU utilisation is driven by the degree to which GPU kernels engage the GPU's processing elements and memory; thus, co-located jobs with high GPU over-commitment experienced greater JCT slowdown due to more significant levels of resource contention by the scheduled GPU kernels. Figure 3.6 shows a snapshot of the competing GPU kernels for VGG11 and MNASNet1.3. We further investigate the utilisation patterns when co-locating jobs with varying computational demands by changing the batch size.

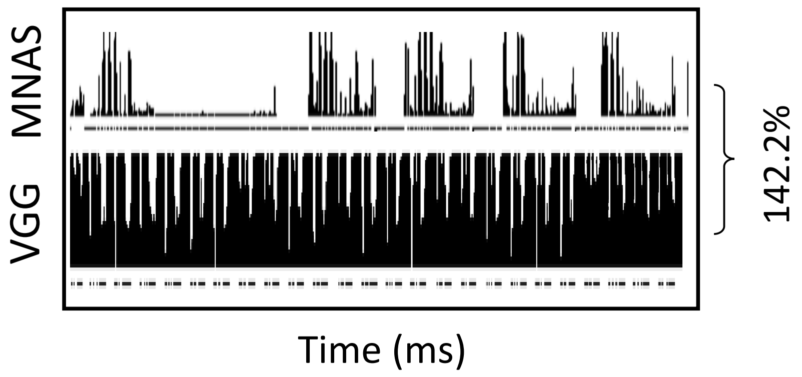
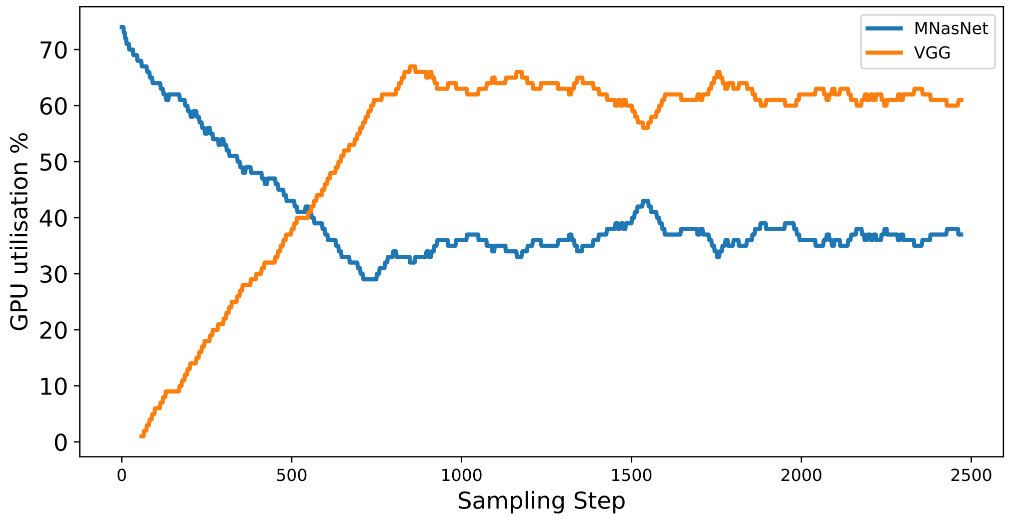
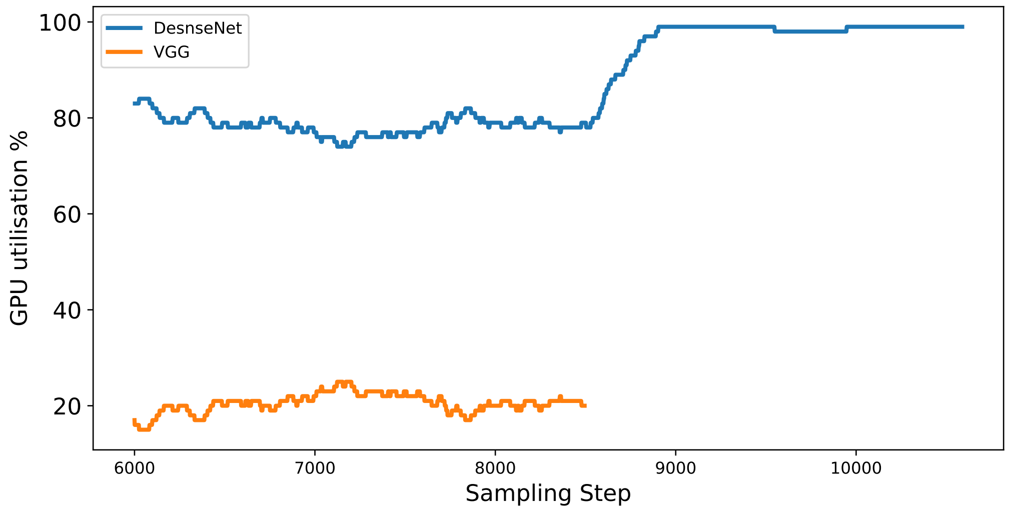


Figure 3.6: Competing GPU Kernels. VGG11 is the main source of contention on NVIDIA 1080.



(a) GPU utilisation patterns between VGG11 and MNASNet1.3, both with batch size 32.



(b) GPU utilisation patterns between DenseNet169 (batch size 64) and VGG11 (batch size 8).

Figure 3.7: GPU utilisation patterns between DL jobs when co-located: VGG11 (15.2 GFLOPs with Batch Size of 1), MNASNet1.3 (1.06 GFLOPs with Batch Size of 1) and DenseNet169 (6.8 GFLOPs with Batch Size of 1).

Computation Demand. Computation is mainly driven by the number of FLOPs and the inputs size to the model [13, 220]. Figure 3.7 depicts the resulting GPU utilisation patterns between three models when co-locating on to the same GPU.

Figure 3.7a shows the utilisation patterns between VGG11 and MNASNet1.3 with the same batch size, it is observable that VGG11 consumed more resources in comparison to MNASNet1.3 due to larger amount of computation. Similarly, Figure 3.7b shows that a model with fewer FLOPs but large input size – DenseNet169 with a batch size of 64, can dominate the GPU computation consumption in comparison to a model with a great number of FLOPs but small input size – VGG11 with batch size of 8. This demonstrates that computational requirements such as FLOPs and input size for DL jobs can be indicative of the resultant interference, and that GPU utilisation clearly reflects the computational patterns and correlates with performance interference.

3.2.3 Modelling Interference

It is observable that GPU over-commitment manifests, when the cumulative GPU utilisation is >75% as shown in Figure 3.5a. This is intuitive as GPU resources are in heavy contention when both DL training jobs are resource hungry as indicated in Figure 3.7. Therefore, a *non-linear relationship* is observed as a whole, and a quadratic polynomial fits well to the data with the lowest R-squared difference, indicated by 0.88 and 0.84 for NVIDIA 1080 (Equation 3.2) and Nvidia RTX 2080 (Equation 3.3), respectively (closer to 1 the better). We utilised NumPy [85] package for the polynomial regression fitting technique.

$$y = x^2(1.32198) + x(-0.00728) + 6 \times 10^{-5} \tag{3.2}$$

$$y = x^2(1.16664) + x(-0.00302) + 4 \times 10^{-5} \tag{3.3}$$

Impact of Hardware. The interference impact is not significantly different between our DL systems as shown on Figure 3.5a. However, when running identical DL jobs, the coefficient of the lines of best fit reveals that the severity of interference is slightly lower on the NVIDIA RTX 2080 architecture than on the NVIDIA 1080. This is due to additional processing elements, increased cache size, and memory bandwidth.

Impact of Locality. While the aim of this study was to measure interference impact from co-locating DL jobs within the same GPU, we have also studied how locality impacts DL job JCT to demonstrate that interference has the same order of impact. This was conducted by using data parallelism for distributed training, and the workers were placed in GPUs on separate machines as described in Section 2.4.3.1. Communication primitives within the NVIDIA NCCL library was used to allow for multi-node communication, as frequently used by the distributed DL community [81, 95, 238]. We observed that locality increased DL job JCT ranging from 2x to 5.5x when executing distributed DL jobs across multiple machines, as shown in Figure 3.8. This demonstrates interference does result in comparable performance degradation as locality-agnostic placement.

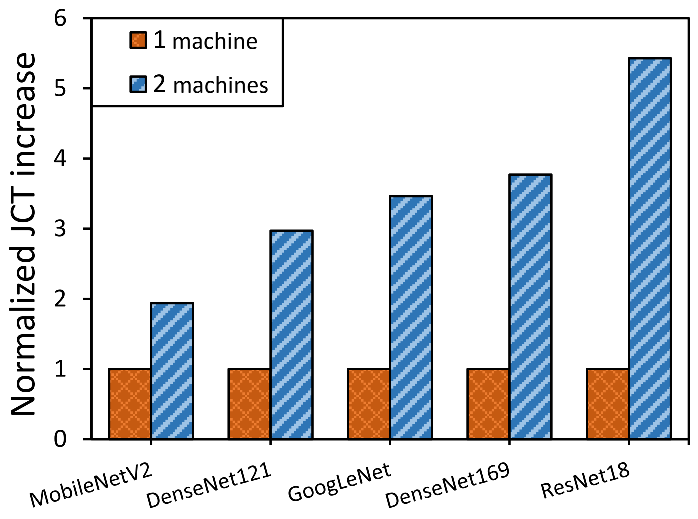


Figure 3.8: JCT increase by locality across five distributed DL jobs with 10GB Ethernet

It is observable that distributed DL job performance patterns share similarities to DL job interference in terms of performance degradation severity and heterogeneity.

Such results reinforce the notion that network latency is a sizeable contributor to JCT slowdown in DL clusters, and is a major focus of prior DL cluster schedulers [9, 81, 182, 246]. However, it is apparent that interference can also contribute similar levels of JCT slowdown and is in some cases, a greater concern considering the majority of DL jobs are single-GPU as shown in Section 3.1.

3.2.4 Summary

Our method for conducting the co-location study on DL training jobs on GPUs is general and presents confirmation that interference is manifested when each of the co-located DL training jobs is of a high GPU utilisation. The interference can contribute to JCT increase, ranging from 1.5x to 3x. We identify GPU overcommitment is highly correlated with performance interference. We further characterised the impact of hardware heterogeneity on interference level, and observe that there are minor differences between our DL systems via fitted regression coefficients.

The observations answer our [**RQ1**] that GPU utilisation does indeed contribute towards DL training jobs interference, and one should leverage GPU utilisation to determine safe co-location at scheduling time. However, the mean for determining the GPU utilisation of a particular DL training job remains unclear. Answering the question can allow us to determine safe co-location in DL training job scheduling with minimal overhead, improving GPU utilisation in DL training clusters.

3.3 Deep Learning models and GPU utilisation Relationship

We turn to investigate the relationship between DL models and GPU utilisation and attempt to answer our [**RQ2**]. As discussed in Section 2.3.1.5, the number of FLOPs and memory parameters of a model are the main contributors toward computation intensity and memory consumption during DNN execution on compute devices. As we confirmed the relationship between GPU utilisation and interference level, we turn to investigate the relationship between DL models and GPU utilisation, confirming safe co-location can be enabled by inferring GPU utilisation from DL model characteristics.

DL model execution consists of a series of matrix operations including both multiplication and addition, which are floating point arithmetic. The greater the number of FLOPs a GPU must execute, the higher load on the GPU, as indicated in Section 3.2.2. Since the load on the GPU correlates to FLOPs and it to GPU utilisation, we first investigate whether the number of FLOPs, which can be determined by traversing the DL model's IR, has a direct influence on GPU utilisation (§3.3.2). We then investigate whether other features extracted from DL model's IR exhibit a similar relationship to GPU utilisation (§3.3.3), confirming that a relationship exists between DL models and GPU utilisation.

3.3.1 Profiling Method

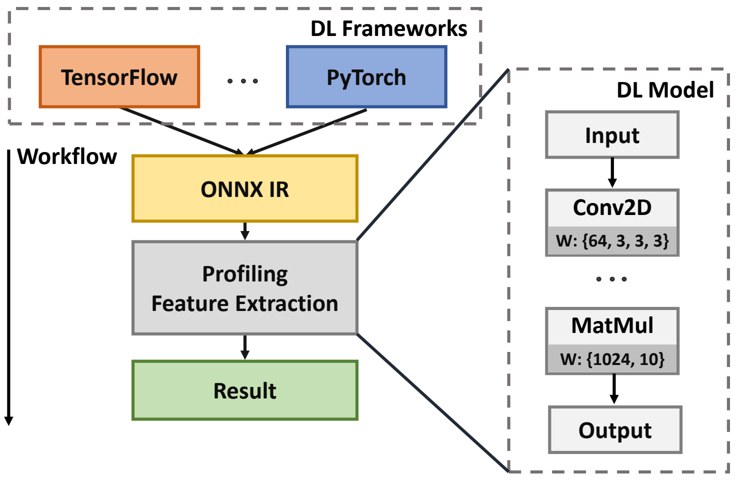


Figure 3.9: Extraction process via ONNX model's IR.

Algorithm 1 Pseudocode for profiling a DL model

```
Input: ($f$, $X$) // DL model file, example inputs
Output: $\mathcal{S}$ // statistic results
1: $\mathcal{M} \leftarrow$ loadOrConvert($f$, $X$) // load or Convert DL model
2: $\mathcal{N} \leftarrow$ topoSort($\mathcal{M}$.graph.node)
3: $\mathcal{H} \leftarrow$ initHookTable() // supported operations
4: for $n$ in $\mathcal{N}$ do
5:     $fn \leftarrow \mathcal{H}$.get(n.opType) // get profiling function
6:     $s_n \leftarrow$ fn(n) // profile
7:     $\mathcal{S}$.write($s_n$)
```

In order to obtain the number of FLOPs for each individual model (Table 3.4 and Figure 3.4), a custom profiler is built using Python. Specifically, our profiling process operates on the ONNX model's IR as it maintains good interoperability across DL frameworks (see § 2.3.1.3). Our profiling process traverses the DL model's IR and calculates the number of FLOPs required for each operation encountered, based on the required inputs, output shape, and parameters. For example, for a standard general matrix multiplication, the total number of FLOPs between $A = \mathbb{R}^{m\times n\times k}$ and $B = \mathbb{R}^{k\times p}$ is calculated from the following:

$$F_{gemm} = \mathbf{2} \times k \times p \times n \times m \tag{3.4}$$

where the multiplication of two [2] comes from the fact that addition is performed after multiplication [228]. To facilitate the profiling of individual operations, we register each operation within an operation table, the calculation for the number of FLOPs is based on our own understanding of how the matrix manipulation is performed. Specifically, the list of operations we support is shown in Table 3.5, and

[2] *Multiply-Accumulate* (MAC) [92] is defined as the number of FLOPs divided by two.

they are common DL operations in most prominent DL models. Figure 3.9 and Algorithm 1 describe the process of our profiling engine and shows the pseudocode for obtaining FLOPs from each operation, respectively. We verify our profiling fidelity via comparing our outputs with numbers reported in literature.

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

Table 3.5: ONNX model operation, DL model's characteristics and matrix manipulation counts, i.e., count of the operations, memory parameters and FLOPs.

Model	FLOPs (ours)	FLOPs (reported)	MACs (ours)	MACs (reported)	FLOPs Difference	MACs Difference
DenseNet121	**5.7B**	5.5B [96]	**2.8B**	NR	3.6%	–
DenseNet169	**6.8B**	7.1B [96]	**3.4B**	NR	4.4%	–
MNASNet1.0	**634M**	680M [225]	**314M**	340M [225]	7.2%	8.2%
NASNetMobile	**1.1B**	1.1B [225]	**563M**	564M [225]	–	0.1%
ResNet50	**7.8B**	3.9B [87]	**3.9B**	NR	∼2X	–
VGG19	**39.3B**	19.6B [213]	**19.6B**	NR	∼2X	–

Table 3.6: Comparison across profiled and reported models FLOPs count. NR – Not Reported. M – Million. B – Billion.

Profiling fidelity. Table 3.6 shows six examples of the FLOP count calculated by our profiling engine against reported the FLOP count in literature. Our FLOP calculation for Resnet50 and VGG19 differ by approximately a factor of two, this could be due to the reported count omitting the number of addition as indicated by the comparison between MACs (ours) and reported FLOPs. In contrast, our calculation for the other DL models are within 10% of the reported values. This indicates that our approach to FLOP calculation approach is within satisfactory accuracy and unless stated otherwise, we report the FLOP count including the number of additions, as the majority of FLOP calculations reported in literature include addition [4].

3.3.2 FLOP Analysis

As discussed previously in Section 3.2.2, computation demand drives GPU utilisation and FLOPs drive computation, we then investigate the relationship between FLOPs and GPU utilisation.

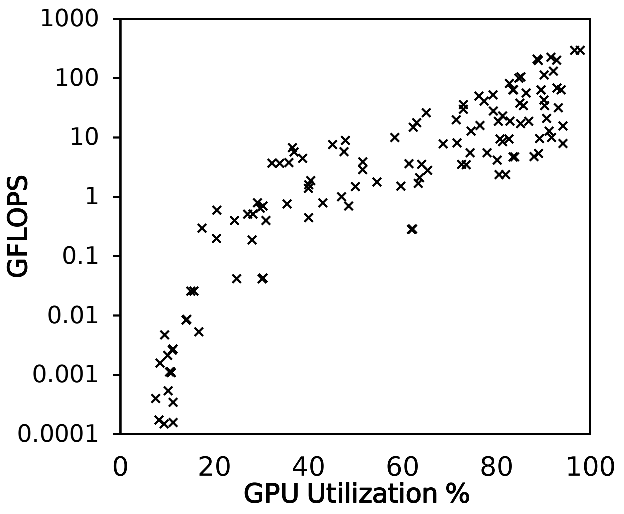


Figure 3.10: Relation of GFLOPs to GPU utilization.

Figure 3.10 shows that the FLOP count does correlate to GPU utilisation, and this confirms our understanding of the limit between matrix manipulation and hardware execution as explained in Section 2.3.1.2. However, there exist several models that exhibit low FLOP counts and high GPU utilization, which we postulate is likely a result of model architectures producing a high number of intermediate feature maps leading to a large number of memory accesses. These findings are supported by prior studies in the neural architecture search literature [201, 224].

Since the relationship between FLOPs and GPU utilisation is not entirely linear, this may indicate that there are other features that may contribute toward GPU utilisation for DL models.

3.3.3 Identifying additional Features

Data-driven methods often depend on a set of features that are correlated with the target variable. It is understood that the inputs to a DL model and its DL operations are the main drivers toward the number of FLOPs and memory consumption, as explained previously in Section 2.3.1.5 and in the literature [35, 67, 224, 243]. Additionally, a DL model with a large linear layer can occupy a large amount of GPU memory [218]. Therefore, to quantify the correlation between features and GPU utilisation, we investigate the remaining features extracted via traversing the DL model's IR, shown in Table 3.5.

Batch Size. In reality, the number of FLOPs and memory transactions increase when the batch size is increased. This is due to the number of FLOPs and memory transactions being 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 number of DL model inputs. Therefore, the number of computations scales with the increase in batch size, leading to higher GPU load as shown in Figure 3.7b. Batch size is one of the major parameters to be considered in DL models inference [82, 177, 208].

Model Parameters. DL models with a large number of parameters come with greater capacity to learn from data as shown previously in Figure 2.12, leading to higher accuracy but also higher memory consumption [224]. Additionally, a larger model usually comes with larger computation. For instance, larger convolutional filter size, and thus larger number of parameters and greater number of FLOPs computed [94]. Thus, more GPU computation and memory kernels launched in the GPU device, causing increased GPU load. It is observable that models with large numbers of parameters seem to exhibit higher GPU utilisation, for example, VGG19 with 98.8% as shown in Table 3.7.

DL operations. DL operations are composed of one or many GPU kernels that are either computationally intensive or memory intensive, i.e., most of the time spent on a GPU is computation or memory transactions [155]. Notably, DL models that have a significant number of both computational or memory operations should result in lower execution latency due to memory and computation GPU kernels overlapping. However, DL models that result in an uneven number of the kernels may result in high latency due to GPU kernels queuing for required inputs and execution [155], thus lower GPU utilisation. Table 3.7 shows that ResNet50 exhibits higher GPU utilisation in comparison to MNasNet1.0 with higher number of DL operations, 68 and 46, respectively. To identify and study these features that can contribute to GPU utilisation, we instruct our profiling engine to extract relevant features, specifically, count both the number of DL operations and matrix operations. Table 3.7 shows a sample of changes to DL model utilisation that include the number of convolution and linear operations, and model parameters; notably the number of convolution and memory parameters appear to contribute to GPU utilisation.

	MNasNet1.0	VGG19	ResNet50
Number of convolution layer	52	16	53
Number of linear layer	1	3	1
Number of parameters	4.3M	143M	25M
FLOPs	633M	39.3B	8.1B
Other modules (e.g. Batchnorm, AveragePool)	46	25	68
Utilization	**29.8**	**98.8**	**80.9**

Table 3.7: Sample of changes to DL model utilisation at batch size 8 on NVIDIA 2080.

3.3.4 Quantifying the Relationships

In the previous Sections(§3.3.2 and §3.3.3), we identify the set of related features and explained the relationships between them and GPU utilisation. We then conduct correlation analysis to quantify the relationships between all extracted features, using Pandas [150], a Python package for data manipulation.

In order to do so, we conduct two correlation analysis by using the Pearson [205] and Spearman-rank [204] methods to measure the strength of linear and monotonic relationships, shown in Figure 3.11 and Figure 3.12. These correlation methods are often leveraged in the system research community [36, 137, 265]. A correlation measure of 1 implies a positive relationship, whereas -1 implies a negative relationship.

Correlation between features. It is observable that features themselves have interesting correlation relationship. For example, the number of convolution layer correlates strongly with the number of ReLU activation operations in both pearson correlation and spearman rank correlation; this is due to CNNs typically being composed of individual blocks [87, 94], where each block usually contains at least one convolutional layer followed by an activation layer (e.g., ReLU). Similarly, the number of convolution layer and number of batch normalization layer correlate strongly for the same reason. Importantly, memory parameters correlate strongly with the number of FLOPs in both pearson and spearman rank correlation, this confirms our analysis that larger models with a large number of memory parameters contributes to more computation as explained in Section 2.3.1.5. Finally, we observe that FLOPs do correlate in both pearson and spearman rank correlation, as expected.

GPU utilisation correlation. It is observable that the last row of Figure 3.11, matrix manipulations (FLOPs, MACs), the number of parameters, pooling operations (MaxPool, AveragePool) and memory intensive operations (Flatten) correlates linearly ($\geq$0.6) with GPU utilisation using the pearson correlation. This confirms the moderate linear relationship that we speculated. Interestingly, by looking into the last row of Figure 3.12, the majority of the relevant features we extracted seem to exhibit a positive relationship to GPU utilisation ($\geq$0.5). In particular, the number of matrix manipulation, memory activations and memory parameters exhibit a strong positive increasing relationship ($\geq$0.7). Our findings indicate that all individual features we extracted have moderate to significant importance towards contributing to GPU utilisation.

Inspecting the DL model's computation graph, extracting and leveraging features such as the number of matrix manipulation, memory parameters, memory activations and operation types, may allow us to infer the likely resultant GPU utilisation given both the linear and monotonic relationships discovered, answering our [**RQ2**]. Paving the way for a proactive approach to estimating GPU resource consumption given a DL model computation graph.

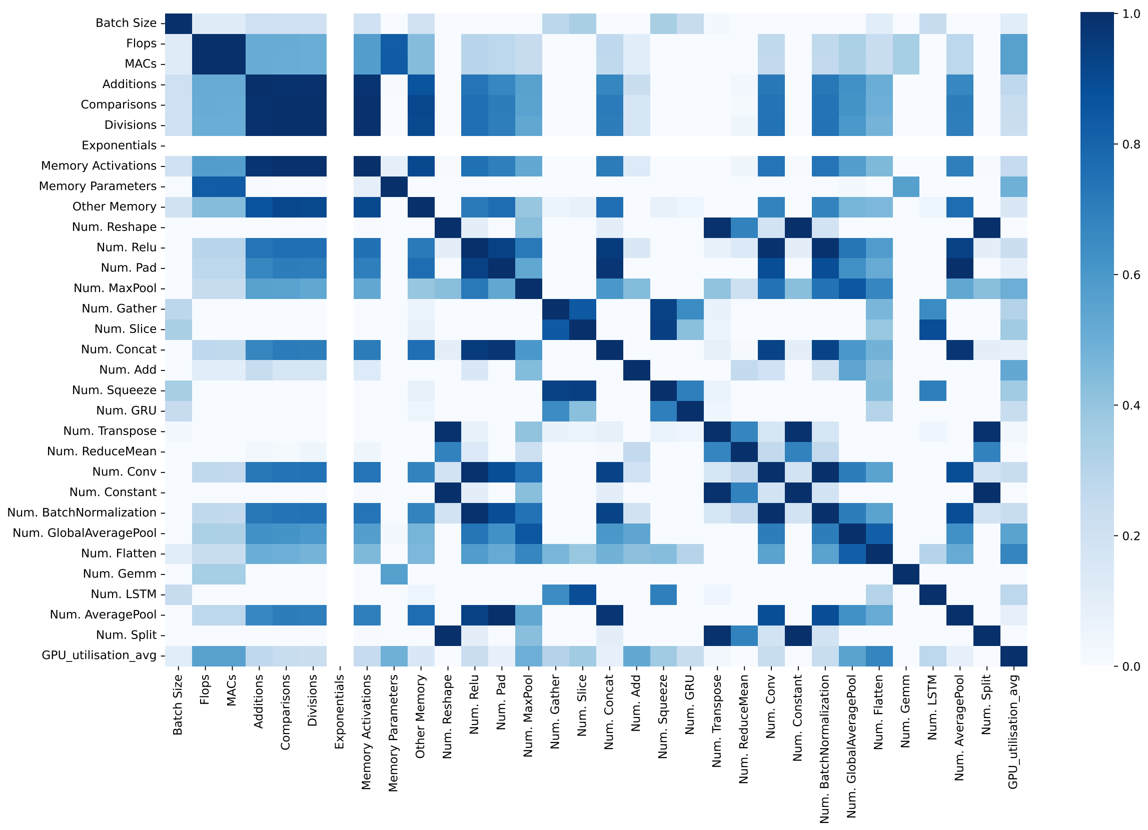


Figure 3.11: Pearson correlation between extracted features and GPU utilisation.

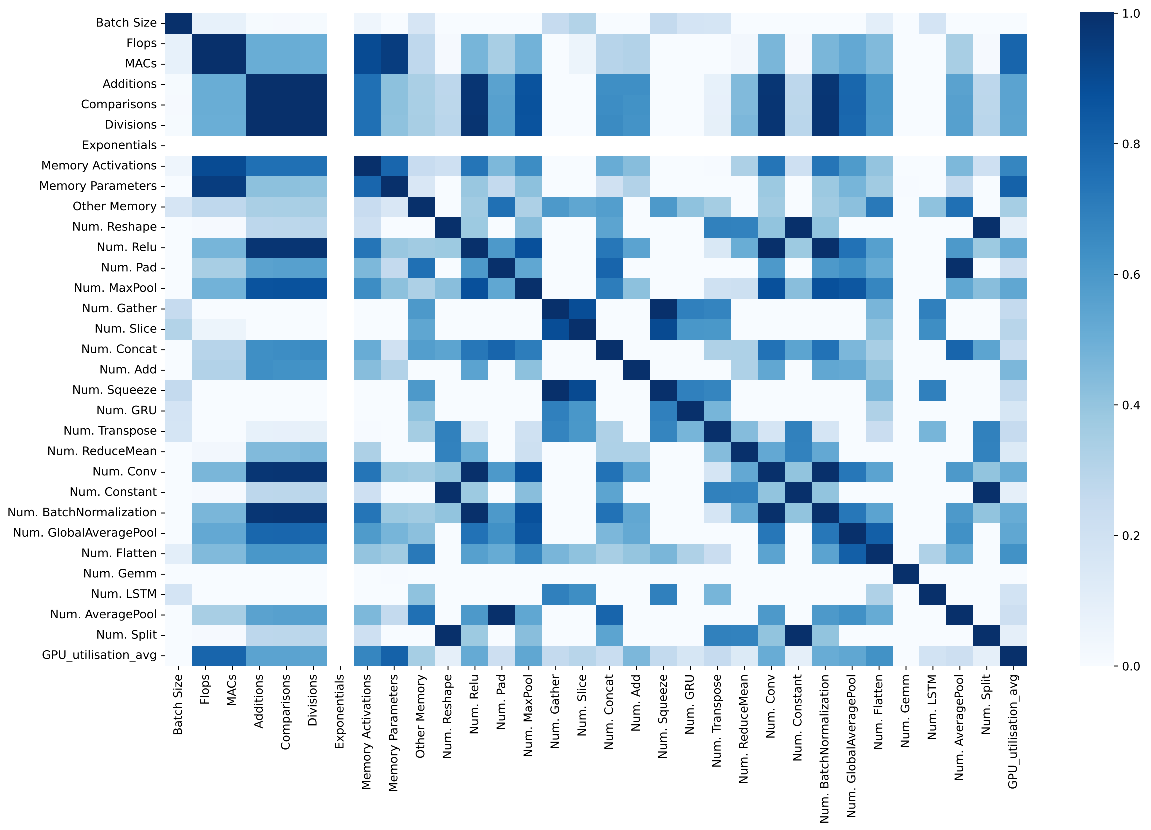


Figure 3.12: Spearman Rank correlation between extracted features and GPU utilisation.

3.4 Summary

In this chapter, we have presented an in-depth analysis of a large-scale DL training cluster, interference analysis between DL training jobs and analysed the relationship between DL models against GPU utilisation (during the training phase). We observe that low GPU utilisation is a systemic issue in large-scale DL training clusters and the majority of production DL training clusters comprise single-GPU jobs; justifying a need for proposing new DL training resource managers that enable co-location. The interference analysis enables the study and characterisation of DL training job co-location, and identifies the relationship between GPU utilisation and interference (JCT increase); the result of this analysis quantifies the interference effects and enable a minimal overhead approach to safely co-locate DL training jobs with respect to hardware. Finally, the results from analysis between DL model characteristics and GPU utilisation demonstrate both linear and monotonic relationships exist, and suggest that one can leverage features readily available from DL model IRs to infer GPU utilisation.

The DL training cluster analysis indicates exclusive locks have a huge impact on average cluster GPU utilisation, and low GPU utilisation persists across large-scale production DL training clusters, averaging at 60% or lower. Moreover, a large proportion of jobs (70 to 80%) are single-GPU, presenting significant opportunities for co-location to address low cluster GPU utilisation.

The interference analysis studies and models interference for GPU over-commitment with respect to GPU hardware with polynomial regressions. Notably, interference manifests when the cumulative GPU utilisation is greater than 75%. These polynomial regression coefficients are presented, so they can be leveraged by other researchers. This analysis demonstrates that by understanding GPU utilisation and the level of

GPU over-commitment, one can derive a safe co-location scheme.

The relationship between DL model characteristics and GPU utilisation is presented and discussed in detail. From the analysis, it is observable that the number of matrix operation, memory consumption and DL operations correlate positively with GPU utilisation in both pearson and spearman correlation methods ($\geq$0.6 and $\geq$0.7 respectively), presenting an opportunity to infer GPU utilisation by extracting features from a DL model's IR.

In the next chapter, we present a data-driven interference-aware cluster scheduling approach that leverages our findings to improve co-location decisions and DL training cluster resource utilisation.

Chapter 4

Proactive DL Training Job Scheduling with Horus

The efficient co-location of tasks on GPUs resources to improve cluster GPU utilisation are important and hard to achieve in cluster scheduling, according to analysis presented in Section 3.1.

In a DL training compute cluster, the cluster resource manager is responsible for maintaining state about machine load, scheduling and the allocation of tasks. The objectives of existing DL cluster resource managers mainly focus on accelerating training time and respecting job fairness, which affect cluster GPU utilisation. In this chapter, we present our proactive DL training cluster resource manager, Horus.

4.1 Overview

Horus is a prediction-based interference-aware DL training cluster resource manager that focuses on improving cluster GPU utilisation and fairness, while minimising training time. Horus is proactive, which means it anticipates potential performance interference effects at scheduling time, by predicting the likely resource consumption of an incoming DL job. Horus has been designed as a set of components that can be deployed alongside of existing cluster resource manager frameworks such as Kubernetes. Figure 4.1 depicts how Horus fits into a cluster resource manager framework and its main components.

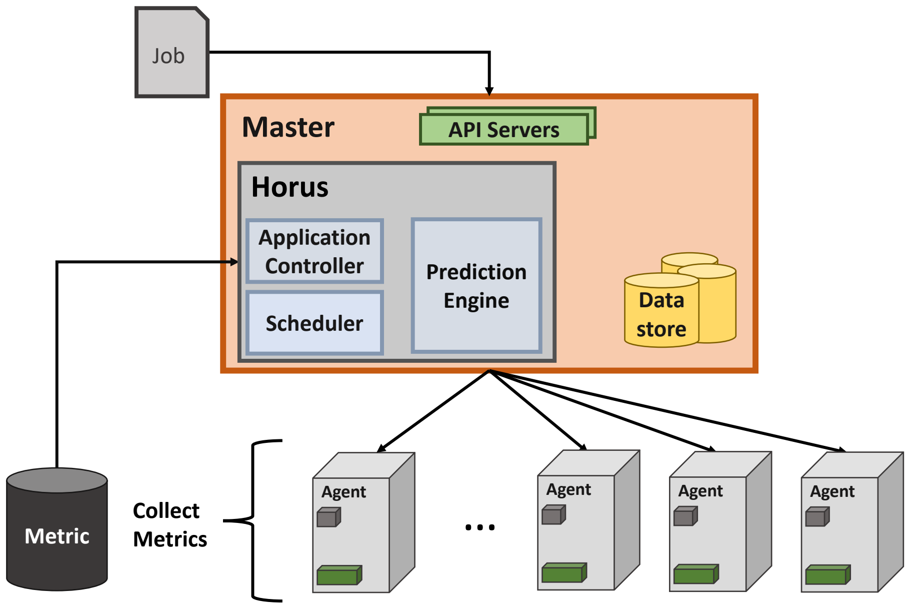


Figure 4.1: Horus overview, comprising (i) an application controller, (ii) a scheduler and (iii) a prediction engine.

To achieve safe co-location of DL jobs while scheduling, we need to co-locate DL jobs with suitable GPU utilisation patterns and GPU memory consumption, as indicated in Section 3.2.1. To this end, we have designed three main components: (i) a GPU resource prediction engine, (ii) a scheduling cost model and (iii) an application controller and resource scheduler. The GPU prediction engine estimates: (i) GPU utilisation and (ii) GPU memory consumption, so they can be fed into the scheduling cost model (§4.3.2) for the cluster resource scheduler, in order to make safe co-location decisions that avoid performance interference.

We first discuss the methods leveraged within the proposed GPU resource prediction engine (§4.2), followed by the overall Horus proactive scheduling approach that integrate with estimated GPU resource consumption (§4.3).

4.2 GPU Resource Prediction Engine

4.2.1 Overview

The GPU resource prediction engine is composed of two models, (i) a trained ML model for predicting the average GPU utilisation of a DL job, and (ii) an analytical model for calculating the likely GPU memory consumption. Our prediction engine extracts key DL model features as described in Table 3.5, by iterating over the ONNX DL model's IR. Obtaining aggregate features such as FLOPs, memory parameters and memory activations, by calculating them based on their inputs, output shape, and parameters. The inputs to the prediction engine are the following: (i) DL model definition, it can be ONNX IR or one of any publicly supported ONNX IR DL model's file (checkpoint file), and (ii) input definition, the expected input including the shape and type. These extracted features are normalised and used as numerical inputs to a ML model in order to predict the GPU utilisation ($\texttt{GUtil}_j$) 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 [41, 51, 230]. These profiles are nominally acquired by 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 incoming DL workloads entering the system. To accommodate with new innovation, the machine learning model can be periodically retrained after collecting additional profiles (e.g., when new models are discovered) or after the detection in accuracy decreased.

Apart from the GPU utilisation, GPU memory consumption ($\texttt{GMem}_j$) is also calculated following Equation 2.7 mentioned in Section 2.3.1.5; since the inputs are already determined, the GPU memory calculation can be obtained following the FLOPs calculation process. The GPU memory calculation can be conducted in advance to provide hints to cluster resource managers to avoid potential OOM error, minimise error overhead, re-scheduling time, and improve overall scheduling efficiency. The machine learning model training process is described (§4.2.2).

4.2.2 Prediction Model Training

Our ML model's aim is to predict GPU utilisation for a given DL job, and therefore the ML objective is modelled as a supervised regression task, i.e., predicting a continuous value as discussed in Section 2.1.1. In the following section, we show the prerequisites for training the GPU utilisation predictor.

Dataset Collection. Training a ML model requires a well-prepared dataset. In the pursuit of an accurate GPU utilisation predictor, a dataset composed of 145 rows of unique DL model features were collected by running the profiling process described in our co-location study analysis (§3.3). We collected 145 rows of data entries based on one DL system. Our dataset size is comparable to prior work [250]. Table 4.1 shows example rows and columns from the dataset we collected. The ground-truth label in the dataset is the GPU utilisation column.

Data Preprocessing. It is common for ML pipelines to normalise data to the same scale to avoid inefficient learning as regularisation often has the assumption of a zero mean and unit variance [216]. Therefore, before training the ML model for predicting the GPU utilisation, features are normalised via the python package scikit-learn [216]

Model	Batch Size	FLOPs	MACs	...	Num. Parameters	GPU utilisation
DenseNet121	32	1.8e+11	9.1e+10	...	32M	34.2%
DesneNet169	32	2.2e+11	1.1e+11	...	57M	35.4%
MobileNetV2	32	1.9e+10	9.6e+9	...	14M	35.9%
ResNet18	32	2.3e+11	1.2e+11	...	46M	76.6%
ShuffleNet 1.5	32	1.9e+10	9.5e+09	...	14M	42.7%

Table 4.1: Example rows and columns from the DL models' features dataset.

using the standard scaler, which works by following equation 4.1.

$$z = \frac{x - \mu}{\sigma} \tag{4.1}$$

where z is our normalised value, x is the original value, μ is the mean for the column feature and σ is the standard deviation for the column feature.

Model Selection. The standard technique to choose the best performing model across a range of ML models is by training and evaluating each individual model on a metric of interest [20, 250, 259]. Given it is a regression task (predicting GPU utilisation), we picked four prominent regression models for evaluation: (i) Linear Regression, (ii) LightGBM [117], (iii) XGBoost [32] and (iv) Random Forest [23]. These models are commonly used by system research community for regression tasks [153, 259, 267].

Training Procedure. To properly evaluate models and ensure the predictor can predict GPU utilisation for unseen models (i.e., a new DL model architecture), we have to incorporate validation and testing into our training procedure, which consists of commonly conducted steps [22, 123, 259, 261]: (i) Random shuffling and splitting samples into 80% training and 20% testing sets. (ii) Applying the chosen regression models on the training set to investigate prediction effectiveness. Additionally, we perform a grid search on each regressor to train the model with five-fold cross validation using the training dataset to find the best parameters, i.e., n estimators and max depth for tree-based regressors. (iii) Testing our models with 20% testing set to ensure the model can generalise to unseen data.

Metric Selection. The predictive performance must be measured using an appropriate error metric. Since the predicted GPU utilisation is leveraged for safe co-location, an under-predicted value is undesirable. In contrast, an over-predicted value is acceptable, i.e., taking the worst-case in calculating the cost of co-location. While the Mean Absolute Percentage Error (MAPE) is a common regression error metric leveraged in prior work [116, 119], it does not capture or penalise under-predicted values. Therefore, we opted to leverage the Root Mean Square Log Error (RMSLE) as shown in Equation 4.2. RMSLE is an established measure of regression accuracy that penalises under-prediction [2].

$$\sqrt{\frac{1}{n}\sum_{i=1}^{\mathcal{N}}(\log(\hat{y}_i+1)-\log(y_i+1))^2} \tag{4.2}$$

where $\hat{y}_i$ is the predicted value and y_i is the ground-truth value. The log function is an inverse of the exponential function, meaning values grow exponentially, and therefore the error calculated from the equation would appear higher for a lower range of predicted values with respect to the ground-truth value.

4.2.3 Prediction Model Evaluation

Accuracy. Table 4.2 shows the relative prediction accuracy achieved by different predictors. We observe that XGBoost outperforms other models consistently across the five-fold cross validation and similarly on test dataset. XGBoost achieves both low

	Linear Regression	LightGBM[117]	XGBoost[32]	Random Forest [23]
MAPE	0.169	0.145	0.077	0.087
RMSLE	0.188	0.193	0.133	0.158

Table 4.2: MAPE and RMSLE for GPU utilisation prediction on test dataset, lower is better.

0.077 MAPE and 0.133 RMSLE error, and has a 45% – 70% decrease in MAPE and RMSLE, when compare to a Linear Regression baseline, respectively. As discussed previously for metric selection, XGBoost has the lowest RMSLE error, meaning it is less likely to under-predict values when compared to other regressor models, therefore XGBoost is chosen as our GPU utilisation predictor.

Feature Importance. It is often important to conduct feature analysis after ML model training to determine whether our trained models learnt the correct features. Understanding what features are important could help to improve the design of the DNN architecture and understand GPU resource consumption. Therefore, to understand further what contributes towards GPU utilisation, we investigate the importance of each tree-based regressor feature by extracting the weights associated with each feature per tree and taking the mean across them. For example, random forest feature importance are computed by the accumulation of the impurity decrease within each tree [61]; XGBoost feature importance is computed by accumulation of the `gain` within each tree [32, 245], where the `gain` is computed based on gradient scores. Both of these importance values are calculated via the scikit-learn and XGBoost Python packages, respectively.

Figure 4.2 shows the most important seven features across the tree-based ML models. We observe that the FLOPs metric was the most important, this is intuitive as FLOPs encapsulates the number of computations that must be performed, and we showed that FLOPs correlated with GPU utilisation in Section 3.3.2. Furthermore, it is not surprising that batch size is one of the most significant features, as it was discussed in Section 3.2.2; batch size can drive a higher amount of computation, and thus GPU load. Generalised Element Matrix Multiplication (GEMM) expressed the number of matrix multiplications to be performed, and it is intuitive that the greater the number of GEMM operations, the more computations the GPU must perform. Memory related features such as a number of parameters and activations also resulted in the top two and four most important features. As discussed previously in Section 3.3.3, a greater number of parameters could result in a higher number of computations, and thus activations. These features are clear indicators and follow existing literature on model compression and neural architecture search, where reducing the number of parameters and intermediate activations can save computation and memory consumption on the hardware [201, 225]. The number of Flatten operations is the seventh of the top most important features, this is interesting as the flatten operation does not perform computation but memory manipulation. This confirms that when the amount of memory manipulation is large, the GPU needs to frequently launch memory transactions, increasing GPU load.

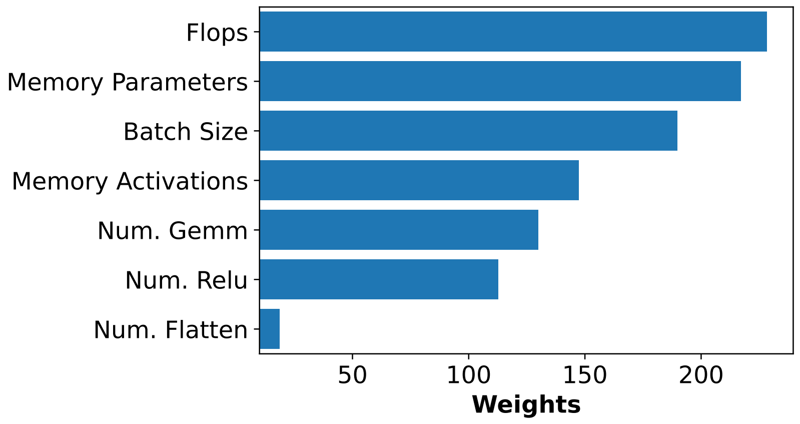


Figure 4.2: Most important identified regressor features.

Discussion. We chose to evaluate the four simpler regression models mentioned in Section 4.2.2, due to the consideration of computational cost. It is often a concern in cluster resource scheduling to be mindful of long decision time due to a large number of job submissions [63]. There are other regression models that can be leveraged when we have a large dataset and a softer constraint in prediction time. For instance, DNN-based regression models leveraged in the DL compiler community [116, 259] can be evaluated in addition to the four regression models. We leave this for future work.

Re-training prediction models are important in any practical setting. The de-facto approach is to pick a decision threshold or confidence score to evaluate whether to use the predicted value [206]. A re-training trigger could be activated when many predictions are below the confidence score n times. Alternatively, by actively monitoring the absolute error between predicted values and actual GPU utilisation, a re-training can be triggered when the MAPE fell above a certain threshold.

Limitation. The GPU utilisation prediction method within our prediction engine operates on the assumption that GPU utilisation does not differ massively between our DL systems (Table 3.3). However, to increase the accuracy of GPU utilisation prediction, it is possible to incorporate readily available hardware features such as core count, bandwidth and clock speed. Note that, one could also train a GPU utilisation predictor per hardware device, similar to the approach taken in [268].

4.2.4 GPU Memory Estimation

Reducing the likelihood of GPU OOM error is important, as OOM errors interrupt the training progress of a DL job. However, estimating GPU memory consumption is a complex task [260]. Our estimation method operates on the high-level IR layer, however, this misses information on (i) the underlying gradient optimiser's state (backpropagation calculation), and (ii) implementation details of DL operation.

Nevertheless, Equation 2.7 provides a coarse grained approximation for the GPU memory consumption and our estimation results in 48% average error across the 145 rows dataset. Table 4.3 shows the estimated GPU memory consumption against actual GPU memory consumption on our DL system B.

Model	Batch Size	Estimated (MiB)	Actual (MiB)	Difference
DenseNet121	16	1592.6	1003.4	58%
DenseNet169	8	1069.1	1035.5	3%
Resnet34	64	1621.9	1195.5	36%
MNASNet1.3	16	717.7	953.8	-25%
ResNet50	32	2250.8	4118.8	-45%

Table 4.3: Examples comparison between estimate GPU memory consumption against actual.

Limitation. Our GPU Memory estimation currently operates on the DL models' IR, fine-grained details on how the DL library implements the GPU computation and memory kernels are not available. Obtaining such implementation details may give clearer and more accurate memory consumption usage patterns due to the ability to account for temporary memory allocation and memory re-use mechanisms in DL frameworks [4]. As mentioned in Section 2.3.1.5, accounting for memory optimisation techniques for DL libraries [256] are difficult, and our memory estimation technique currently does not handle them. One potential way to address this challenge is to scan through the code to detect DL library optimisations; however, this is outside the scope of this thesis.

Although, our training dataset does not cover the whole spectrum of DNN architectures, the features obtained are common DL operations that exist across most prominent DL models, covering the entire spectrum is not the core focus in this thesis. Importantly, the core focus of our prediction engine is to provide two critical and relevant key features of DL jobs in order to enable safe co-location within a DL training cluster, (i) the predicted GPU utilisation and (ii) GPU memory consumption. These values are useful and provide hints for the DL training resource manager when making scheduling decisions.

4.3 Interference-aware DL training Job Scheduling

4.3.1 Overview

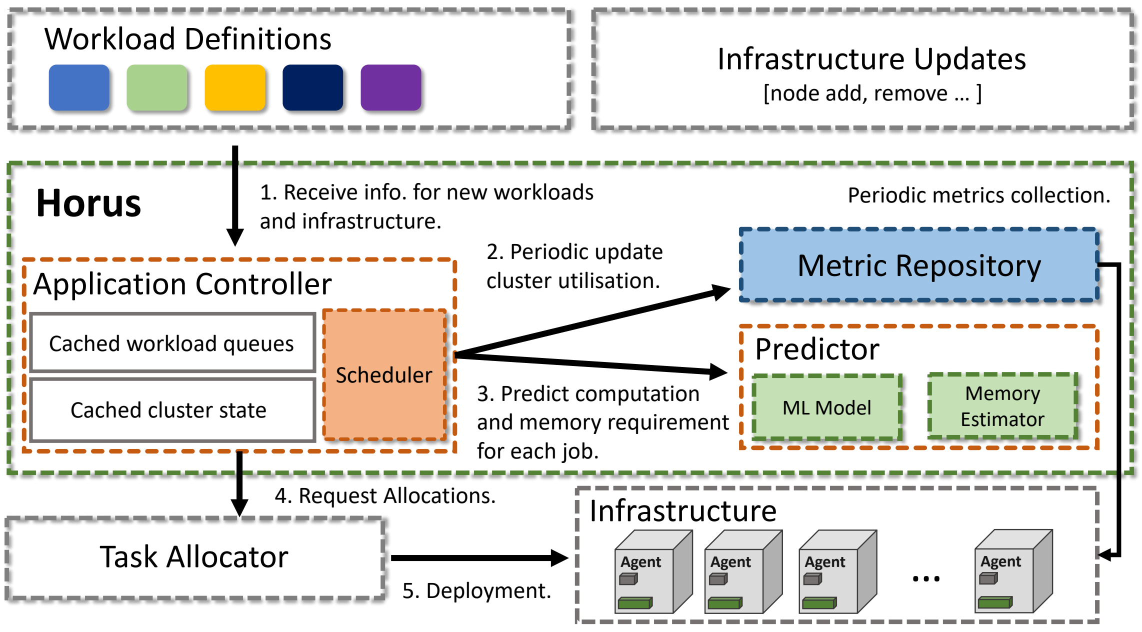


Figure 4.3: The Horus architecture and scheduling steps.

Fig. 4.3 depicts the Horus architecture, which comprises two main components: the *GPU resource prediction engine* (§4.2), and the *interference-aware application controller*. On job submission, the application controller first receives the job definition from the API server, and sends a request to the prediction engine to estimate DL job GPU usage, i.e., both GPU Utilisation and GPU memory consumption by inspecting the DL model's IR. The application controller then labels the job with the predicted GPU resource consumption, selected nodes and GPUs that minimise interference, based on the slowdown functions (§3.2) and a derived cost function (§4.3.2). Finally, the existing cluster resource scheduler (task allocator) then assigns resources accordingly. We believe the application level scheduling logic such as custom costs calculation, task labelling and spawning task groups, should be separated from task allocation logic, hence the design. Note that, this architectured design was adopted in the hyperscale datacenter after Horus was published [226].

The entire cluster state is maintained through infrastructure updates and monitoring agents via existing resource manager frameworks, to collect infrastructure data from each node including GPU and system usage (host memory usage, and CPU utilisation). Additionally, we employed a metric repository to track real GPU usage, allowing the application controller to periodically update its internal cached cluster state, enabling scheduling decisions with respect to real GPU load.

To account for GPU utilisation misprediction and inaccurate slowdown estimation, we combined a reactive migration mechanism to move jobs away from overutilised GPUs ($\geq$ three jobs on the same GPU) by preempting the newest job, so we can preserve the job progress on jobs that been running for a while. Horus combines both proactive and reactive techniques to provide safe co-location and thus boosting resource utilisation.

4.3.2 Interference-aware Cost Model

The key to our interference-aware scheduling is understanding the compute resource requirement *prior* to job execution, and performing job placement with the lowest cost possible to the corresponding resources for the job. This contrasts with existing DL cluster resource managers that either *react* after obtaining workload utilisation patterns or heuristically schedule onto the least-loaded resources (§ 2.4.4).

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	the number of queues

Table 4.4: Notation definition.

Problem formulation. Our objective is to find a job placement schedule 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 for this placement. The optimization problem is therefore defined as the following Integer Linear Programming (ILP) problem:

$$\min \sum_{j\in J}\sum_{n\in N}\sum_{g\in G_n} Cost_{jng} \cdot X_{jng} \tag{4.3}$$

$$\text{s.t.} \sum_{n\in N}\sum_{g\in G_n} X_{jng} = RQ_j^{GPU}, \forall j \in J \tag{4.4}$$

$$\sum_{j\in J}\sum_{g\in G_n} RQ_j^r \cdot X_{jng} \leq CP_n^r - UR_n^r, \forall r \in R, \forall n \in N \tag{4.5}$$

$$X_{jng} = \{0, 1\}, \forall j \in J, \forall n \in N, \forall g \in G_n \tag{4.6}$$

The constraints ensure at every time all GPU requests for each job can be satisfied (Constraint 4.4) and the sum of any type of resource (i.e., CPU, memory, and GPU memory) requested by all jobs on any node must be within the bound of that node's free resources (Constraint 4.5). Subject to these constraints, we aim to minimise the cost of the overall GPU allocation among co-located jobs (Constraint 4.3). For clarity, notations used in this paper are summarised in Table 4.4.

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 utilisation:

$$Cost_{jng} = \omega_1 C_{jng}^{GMem} + \omega_2 C_{jng}^{GUtil} \tag{4.7}$$

wherein ω_i is a customised 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. 4.8):

$$C_{jng}^{GMem} = \frac{UR_{ng}^{GMem} + \texttt{GMem}_j}{CP_{ng}^{GMem}} \tag{4.8}$$

where CP_{ng}^{GMem} is a fixed 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 utilisation of co-located DL jobs and JCT slowdown with respect to hardware outlined in Section 3.2.1, we penalise 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 utilisation with respect to GPU device g. f_g is the function instance in $\mathcal{F}$ and represents the function when the targeting device is of g type.

Hence, the GPU cost can be expressed as:

$$\begin{aligned} C_{jng}^{GUtil} &= f_g(GUtil_{jng}) \\ GUtil_{jng} &= UR_{ng}^{GUtil} + \texttt{GUtil}_j \end{aligned} \tag{4.9}$$

where $GUtil_{jng}$ is the estimated GPU utilization if job j is placed onto node n's GPU g. Our functions $\mathcal{F}$ was fitted against the JCT slowdown and GPU utilisation, 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 [81, 108], we took a heuristic cost-based approach to greedily solve this scheduling problem.

4.3.3 Weighted Fair Queuing Scheduling

Overview. As we observed in Section 3.1.2, JCTs vary among DL training jobs, and it is important to avoid head-of-line blocking and any form of resource starvation – particularly incurred by long jobs with large resource requests as mentioned in Section 2.4.3.2. To schedule different DL training jobs in a fair manner, we borrow the ideas from [176, 239]; (i) cluster similar jobs into several groups to individually

Algorithm 2 Weighted Fair Queuing-Based Job Scheduling

```
Input: ($J$, $S$, $k$, $\beta$) // Pending jobs, current cluster state, $k$ queues and $\beta$ 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 $\beta$ 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. 4.7, ($j, \forall g \in G_n, \forall n \in \mathcal{N}$)
15:             // shortlist a collection of GPUs with min costs
16:             $\mathcal{G} \leftarrow$ select top-$\lambda$ from $\mathcal{C}_j$ in ascending order
17:             // resource allocation
18:             SCHEDULE($j$, $\mathcal{G}$)
```

manage the jobs in a group and (ii) at each scheduling round, we fairly pick a certain number of jobs from k queues, considering job waiting time and the length of the queue, finally assigning the most suitable GPU resources to launch them in the GPU cluster. Algorithm 2 outlines the algorithm details.

Job clustering. Before jobs are scheduled, at each scheduling round, we carry out a clustering procedure for all pending jobs. Specifically, the L1 Distance metric is used to identify similar jobs considering the following features: (i) Number of tasks; (ii) predicted GPU utilisation; (iii) GPUs per task; and (iv) estimated GPU memory. These features outline per-job resource requirements and can be obtained by adopting the method in Section 4.2. In practice, we run the *k-means* algorithm on all yet-to-executed jobs to identify similarities and put them into the corresponding queues, i.e., $\mathcal{Q} = [\mathcal{Q}_1, \dots, \mathcal{Q}_k]$ (Line 2). We set $k = 3$ as we found that from Figure 3.2b, the utilisation patterns have three distinct CDFs.

Picking jobs based on weighted fair queuing. Backfilling has been shown to improve cluster utilisation by identifying suitable jobs in the queue to jump ahead for execution [62, 219]. Combined with backfilling, β jobs are allowed, as a batch, into each scheduling round. Horus picks up a number of pending jobs from each queue according to the queue weight, i.e., the degree of pending jobs within each queue (Line 4). We measure the weight as the product of the median waiting time for jobs per queue and the queue length, i.e., $w_x = \texttt{max}\{Len(\mathcal{Q}_x), Med(\mathcal{Q}_x) \times Len(\mathcal{Q}_x)\}, x = \{0 \dots k\}$, where the `max` operation is to guarantee a non-zero value once median waiting time is zero when all jobs are new arrivals on the system. The median has a statistical property that is less affected by skewed data [100], and thus can more accurately reflect the queuing time for a class of job. Finally, the number of jobs picked from $\mathcal{Q}_x$ can be calculated by $\frac{w_x}{\sum_i^k w_i}\beta$.

The rationale behind this queuing weight is, jobs are expected to be selected and processed from a queue with longer waiting time and larger queue length. This design can actively avoid job *starvation* of any particular class of jobs, whenever a class of jobs starts to starve, an increased number of jobs will be selected. Thus, reservation is implicitly implemented for starving jobs, as the weighted fair queuing algorithm will select the jobs from the longest waiting time queue.

Resource allocation. Because all pending jobs are now well-ordered into $\widetilde{J}$ according to the weighted fairness, the scheduler will try its best to allocate available resources to each job in turn whist minimising 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 Equation 2.7 in Section 2.3.1. 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 the job j (Lines 9-12). By using Equation. 4.7, 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). Figure 4.4 depicts the overall Horus scheduling approach.

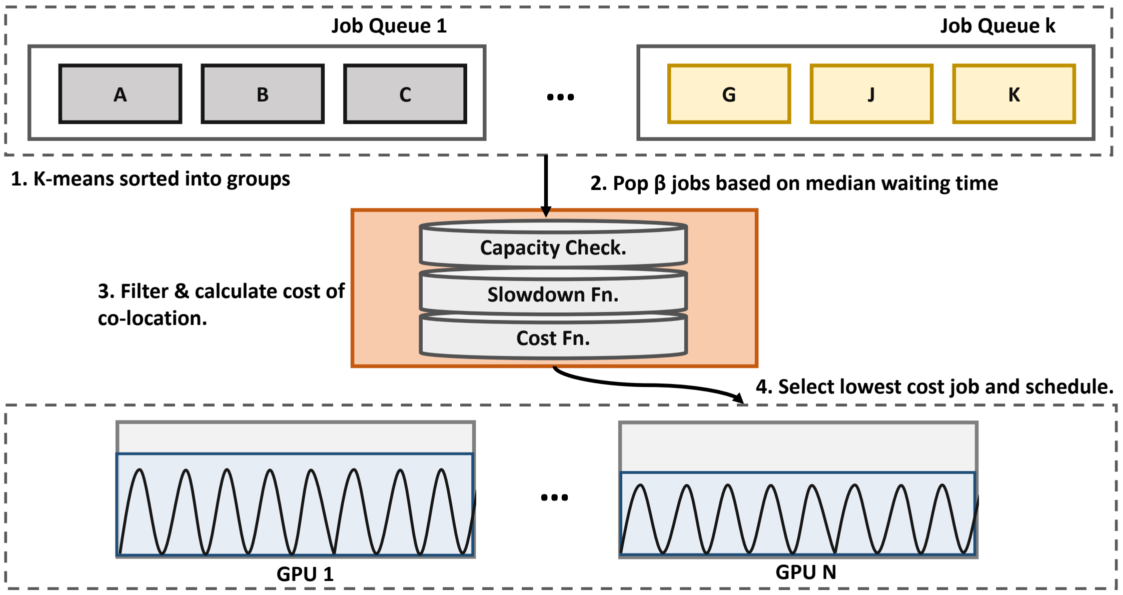

Job Queue 1
A
B
C
Job Queue k
G
J
K
1. K-means sorted into groups
2. Pop β jobs based on median waiting time
3. Filter & calculate cost of co-location.
Capacity Check.
Slowdown Fn.
Cost Fn.
4. Select lowest cost job and schedule.
GPU 1
GPU N

Figure 4.4: The Horus scheduling approach.

4.4 Interference-aware Scheduler Implementation

Horus was implemented as a set of components that can be deployed into existing resource manager frameworks, and was shown to integrate into Kubernetes 1.15. The 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 training cluster, i.e., in Kubernetes. Both our application controller and prediction engine are pods within the Kubernetes ecosystem, and they communicate via remote procedure calls (RPC). We leverage the gRPC[1] library as the underlying RPC implementation to perform data serialisation and de-serialisation 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.

Application Controller. The application controller is written as a single process, and maintains its own view of the cluster state in memory by watching any infrastructure update through the API server. Our implementation of the cluster state requires each machine node to have a map of GPU devices for ease of querying, whereas existing open-source resource manager frameworks do not associate GPUs with machines. Listing 2 shows the type definition of the machine node structure. On startup, the Horus application controller registers with the resource manager to build up its cluster state cache. Within the application controller, a gRPC communication channel is opened between the controller and the prediction engine. By default, the key GPU metrics are not collected by existing resource managers, and therefore an extra monitoring framework must be deployed to collect GPU load information.

[1]https://github.com/grpc/grpc, [01/07/2020]

Monitoring Monitoring is the key to application-aware optimisation [81, 182, 227, 247]. In order to obtain a fine-grained view of the infrastructure, we modified cAdvisor[2], a container monitoring framework to accommodate GPU utilisation per process instead of per GPU. The metrics are then exported, collected and aggregated into a centralised time series database, Prometheus, which our application controller can query via a thin client, and make decisions based on the job's historical usage. Horus collected the same metrics shown in Table 3.1. Listing **??** shows the function to build the machines cache. Horus also periodically updates its scheduling cache, as stale information on GPUs utilisation and memory consumption can result in a significant performance degradation.

Fault tolerance. Using a Network File System (NFS) is often necessary in DL training jobs due to the large volume of training data and limited on-host memory storage [81, 108]. 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 and migration. In Horus, upon selecting the best candidate GPU for placement, a persistent volume request is submitted to the resource manager.

Migration. Apart from periodic cluster state updates, the Horus application controller spawns an additional thread for speculative migration. The Migration thread periodically scans the current cluster state. The reason migration is considered is due to accounting for inaccurate GPU utilisation prediction and stale information from cluster state cache. Our conditions for migration must satisfy the followings: (i.) there must be at least an underutilised GPU, i.e., free or $<45\%$ utilised; and (ii.) we have GPUs that are overallocated, i.e., $\geq$ three DL jobs co-located on the same

[2]https://github.com/google/cadvisor

GPU. We found that GPUs with less than 45% utilisation can at least accommodate one or two more smaller DL jobs. Listing 5 shows the migration procedure in Horus.

OOM Failure. It is possible for our approach (as well as other DL training 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. Upon an OOM failure, Horus proactively queries its current GPU cache to obtain the last used GPU memory in order to update the DL job memory requirement, i.e., GPU memory must be equal or greater than the memory currently used by the GPU. Listing 4 shows the GPU memory update procedure.

Prediction Engine. The prediction engine is implemented as a Python class with two objects, the memory estimator and GPU utilisation predictor. We assumed that the prediction engine has access to the underlying distributed storage system to access the DL model's checkpoint file, i.e., accessible via storage clients or mounting a file system. Upon receiving a job request, the application controller then sends the request to the prediction engine via gRPC along with the storage path. The memory estimation then follows the procedure described in algorithm 1. The GPU utilisation predictor then receives the features extracted from the memory estimation process and emits a utilisation prediction value. We allow the prediction engine to update the ML model through a file system, as shown in listing 6.

4.4.1 Design Assumptions

Locality considerations. This work primarily tackles the JCT slowdown due to interference stemming from job co-location, while optimising the distributed job training is not the focus of this paper. The current job placement scheme assumes high-speed connection across machines, hence the data transfer time during training is not the dominating factor in the current algorithm design. The GPU interference aware scheduling is most suitable for jobs that do not have high frequently transfer of gradients and parameters. As indicated in Section 3.1.3, this is particularly useful for DL training cluster with a large number of single-GPU jobs. For jobs requiring GPUs across machines, the cost model in the algorithm can be integrated with locality-based placement [81, 182, 266], so that GPUs on the same machines or racks can be prioritised before the cost-based GPU filtering to reduce data transfer bottlenecks.

Timing constraints. Horus factors in the waiting time in multiple queue job selection, however like many other DL training cluster resource managers, Horus does not consider the timing constraint in terms of completion time in the placement/planning phase [162, 182, 187, 247, 266]. 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 [81, 108, 187], thus DL cluster managers cannot rely on the DL 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 optimisation problem formulation based on timing constraints. Estimation of execution time relies on a strong assumption, that is, workloads are pre-known, e.g., periodic jobs with same datasets, hyperparameters, and architecture. This assumption does not always hold in DL training clusters due to constant model evaluation with different datasets [86, 95]. Considering this constraint is beyond the scope of this thesis.

4.4.2 Comparing Horus with Existing Co-location Approaches

There are existing DL training cluster resource managers that enable co-location of DL jobs. Table 4.5 highlights the key differences between co-location enabled DL training cluster resource managers. The Gandiva [246] placement strategy entails monitoring of DL jobs throughput, a DL job is then killed or migrated upon detecting a slowdown, using an undefined threshold value given a time period. In such an approach, it is possible for the DL job being migrated to be allocated with another incompatible DL job, leading to equal or greater performance slowdown. Antman [247] enables co-location by monitoring DL jobs, scheduling jobs onto the least-loaded GPUs, employing a local coordinator and modifing 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 maintenance of the modified DL frameworks and allowing communication between cluster resource manager and the running DL job.

	Gandiva [246]	AntMan [247]	**Horus**
Objectives	High utilisation Time-share	High utilisation Time-share	**High utilisation** **Makespan**
Co-location	Trial-and-Error	Least-Loaded	**Prediction** **cost**
Locality	Consolidation	Consolidation	Consolidation
Scheduling input	Waiting time	Current Utilisation	**Predicted utilisation** **Predicted memory** **Waiting time**
Hardware awareness	No	No	**Regression based**
Interference mitigation	Reactive Migrate	Reactive Throttle	**Proactive** **Migrate**
DL frameworks modification	Emit training progress	GPU kernel operation manager	**No**

Table 4.5: Co-location enabled DL training cluster schedulers comparison.

Unlike existing co-location enabled DL training resource managers, Horus proactively assigns DL training jobs to GPUs by computing their suitability—minimising a cost function objective to support co-location with respect to cached cluster state at scheduling time without online profiling. Our approach aims to maximise GPU utilisation, minimise makespan and job waiting time by de-prioritizing co-location placement decisions that would result in JCT slowdown from severe interference.

4.5 Summary

In this chapter, we presented the Horus proactive interference-aware approach and discussed Horus main scheduling objectives, resource utilisation and fairness(§4.3.1).

Horus combines both proactive and reactive approaches, it does not require online isolated profiling, Horus predictive approach can minimise scheduling overhead at runtime. Importantly, Horus leverages application characteristics and DL model structure to estimate GPU resource consumption (§4.2), a key innovative approach to DL training cluster scheduling.

Finally, we described the Horus architecture, and how it schedules jobs with infrastructure metrics according to the scheduling algorithm (§4.3.2).

In the following chapter, we demonstrate Horus performance using an experimentation testbed cluster and large-scale simulation, and find that it compares favourably to other systems.

Chapter 5

Horus Evaluation

To demonstrate the effectiveness and applicability of Horus in a DL training cluster, two evaluations are conducted: (i) an empirical study in an experimental testbed (§5.2) and (ii) a large-scale trace-driven simulation (§5.3). The combination of empirical plus simulation based evaluation is consistent with that previously reported in the literature [81, 95, 182, 187]. In our evaluation, we focus on answering the following questions:

- What improvements does Horus bring to DL training compute clusters in terms of GPU utilisation, makespan and job waiting time?
- How well does Horus scale in a large DL training cluster with hundreds of machines?
- How sensitive is Horus regarding its configurable parameters?

5.1 Experimental Setup

5.1.1 Environment

Experimental testbed cluster. Empirical experimentation is carried out on 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. Our cluster was resource-managed using the Kubernetes 1.15.2 resource management framework. Ubuntu Disco 19.04 was used as the operating system on each node, supported by NVIDIA driver (version 430.50). In our experiments, the software used for DL training was identical to the ones we used in our co-location study (§3.2.1). The DL framework, library and CUDA toolkits responsible for DL job instantiation and execution were packaged in a container for ease of deployment within our Kubernetes cluster. cAdvisor and DCGM were configured to extract data at 1s and 250ms intervals, respectively, as initial trial runs indicated that these parameters resulted in appropriate system state monitoring for the purposes of Horus, given our cluster configuration.

Large-scale trace-driven simulation. We have also designed and implemented a discrete-time cluster simulator to evaluate Horus within larger scale systems. The simulator is written in Python, and simulates a 512-GPU cluster where each machine consists of 8 GPUs. The simulation operates by replaying the full 398 job traces described in Section 3.1. We fast-forward the trace by dividing the normalised time by 10000, effectively fast-forwarded it by 10 seconds per step (the trace timestamp is in milliseconds), as the full trace duration is a month with idle time period. When each schedule loop is conducted, the time step is incremented by one. Listing 1 shows the functions for generating jobs in each time step.

Listing 1 Jobs Generator function.

```
class JobGenerator:
    #...
    def generate_jobs(self, delta_time):
        '''output jobs that are not scheduled and already passed the delta time'''
        def _gen(df, delta):
            # not yet generated + passed the time offset
            logging.info("delta-time: %d" % delta)
            cond = (df["generated"] == 0) & (df["normalized_time"] <= delta)
            return df[cond]
        # Read Jobs from trace data frame
        to_be_generated = _gen(self.trace_df, delta_time)
        self.trace_df.loc[to_be_generated.index, "generated"] = 1
        return to_be_generated

class JobsManager:
    #...
    def gen_jobs(self, delta_time, scale_factor=1):
        # Read jobs from csv that should be in the system by this time
        samples = self.job_generator.generate_jobs(delta_time)
        # put into the queue or queues.
        logging.info("generated: %d" % len(samples))
        converted_jobs = []
        for idx, row in samples.iterrows():
            j = Job(idx, row.minutes * scale_factor, row.normalised_time, row.gpu_per_container,
                    gpu_utilization_avg=row.gpu_utilization_avg, gpu_utilization_max=row.gpu_utilization_max,
                    gpu_memory_max=util.convert_bytes(row.memory_max, unit="MiB"),
                    gpu_memory_avg=util.convert_bytes(row.memory_avg, unit="MiB"),
                    total_gpus=row.used_gpus)
            converted_jobs.append(j)
        # insert to Jobs Queue Manager.
        self.insert(converted_jobs)
        return len(samples)
```

We replay the scheduling trace in resemblance to the real production workflow as shown in Figure 5.1. A job is submitted when the scheduling time step is greater than or equal to the job's submission time, $t_{sched} \geq t_{submitted}$. A job is in a queuing state when there is no available machine to execute the job, and its waiting time is incremented by one at each scheduling round. Allocated jobs will start executing and will be in a finished state when their execution time is up. Each job is only scheduled when the resources are satisfied according to its resource request, similar to production scheduling scenarios. Table 5.1 shows the fields available in the job trace.

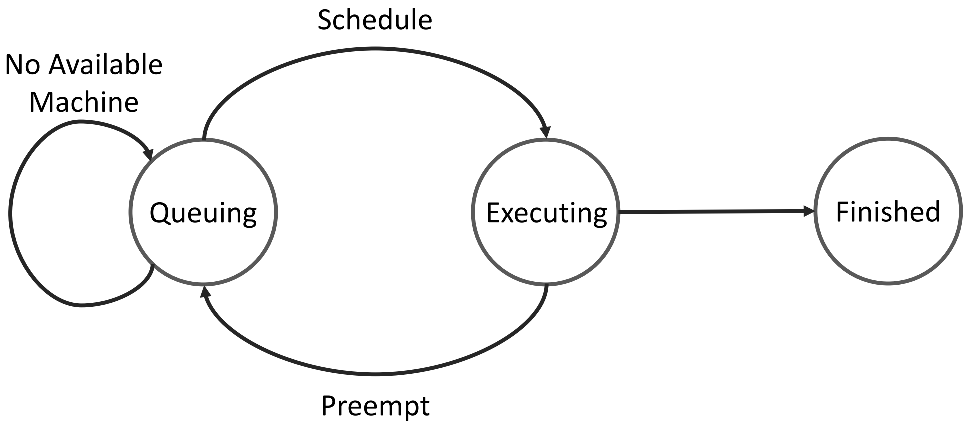


Figure 5.1: DL job state machine diagram in simulation.

Fields	Unit	Description
Submission time	second	The normalised submission time of a DL job.
GPUs (task)	unit	Number of GPUs per task.
GPUs (Job)	unit	Total number of GPUs per job.
Average GPU utilisation	percentage	Average GPU utilisation aggregated per job.
Maximum GPU utilisation	percentage	Maximum GPU utilisation per job.
Average GPU memory consumption	MiB	Average GPU memory consumption aggregated per job.
Maximum GPU memory consumption	MiB	Maximum GPU memory consumption per job.

Table 5.1: Available DL job information in Apollo's trace.

Since GPU resources are a major evaluation consideration, our simulation only considers DL training GPU jobs, i.e., no CPU jobs. Each GPU in the simulated infrastructure has 32 GiB memory capacity based on NVIDIA V100 GPU. The scheduling cost calculation is drawn from the sum of the average GPU utilisation per job per machine (capped at 100%) and GPU memory per job per machine (capped at full GPU memory capacity), similar to [41]. We model the interference effect by using the polynomial function derived in Equation 3.3, multiplied by the sum of the GPU utilisation of the device. Average cluster GPU utilisation is modelled by aggregating GPU utilisation per machine and dividing by the number of GPUs.

5.1.2 Evaluation Methodology

Metrics. Scheduler performance was measured using the following:

- **Cluster GPU Utilisation**: The cluster average GPU utilisation of all GPUs at a single point in time. The metric is used to characterise the GPU resource utilisation of a cluster.
- **JCT**: The end-to-end completion time for a DL job is calculated via the start of the executing state to the finished state, excluding the submission time and queuing time of the job. This metric is used to quantify whether the job performance is affected by interference.
- **Makespan**: The total span of time to complete all DL jobs from enqueuing through to completion. The metric is derived by determining the difference between the completion time of the last job and the submission time of the first job.
- **Job waiting time**: Job waiting time is particularly important in large-scale clusters because the resources within a single multi-tenant cluster are typically insufficient to serve all users at a single point in time. Hence, we measure the job waiting time in our simulation in addition to the above metrics. The metric is measured from a job's point of arrival to being placed and executed by our scheduler.

Comparative algorithms. Note that, our core focus is to evaluate our proactive bin packing scheduling policy against the reactive ones. Thus, we are only focusing on implementing the scheduling policies. We evaluate the Horus scheduling approach described in §4.3.2, with existing DL training cluster scheduling algorithms for comparison:

- ***First in First Out (FIFO)***: FIFO assigns the DL job according to arrival time without priority consideration. FIFO by design does not share GPUs among DL jobs. This algorithm is leveraged in production DL training clusters [95, 186] and traditional cluster schedulers such as Kubernetes [24] and the YARN Capacity scheduler [233].

- ***Performance-aware Bin Packing (PAB)***: PAB schedules DL jobs based on job progress characteristics – detected iteration slowdown. The scheduler measures the difference in the average steps per second vs. the previous state. After a new job placement, if performance drops by 50%, the job is simply re-queued. We picked 50% due to the threshold not being specified in previous approaches, and we consider a slowdown of 50% unacceptable, similar to prior interference-aware resource manager work [52]. We allow a warm-up period between 0-60s, so all DL jobs achieve stable resource patterns due to most DL jobs having a preprocessing stage and our DL jobs on average take 60s to preprocess and start the dataloader [247]. PAB mimics recent proposed DL training cluster schedulers packing policy [182, 246]. In comparison to both Gandiva and Optimus, our approach does not modify the underlying DL frameworks to directly retrieve the training iteration time metrics, we instrument the code to expose the step time metrics to the training logs and instruct our scheduler to parse the logs at every scheduling loop. Furthermore, we do not throttle jobs upon slowdown detection, as we focus only on evaluating the bin packing algorithms.

- ***Opportunistic Bin Packing (OBP)***: OBP assigns DL jobs based on cached GPU resource information. During job submission time, if a GPU has more memory available than the DL job estimated memory requirement, the scheduler opportunistically schedules the DL job onto the minimally loaded GPUs. This approach is taken in one of the recent DL training resource manager [247]. In comparison to Antman, since we are only focusing on the scheduling bin packing policy, we did not model their throttling mechanism in the OBP. We utilised the same migration approach in Horus for OBP, to make sure we only compare least-loaded scheduling policy effectiveness.

Workload. We generated two types of workload for our empirical study and large-scale trace-driven simulation:

- **Empirical Study Workload.** Empirical study experiments were conducted using a mixture of DL jobs generated from the set of jobs presented in Table 3.3, as well as new DL model configurations and models (PyramidNet, LSTM and transformer) resulting in Horus being exposed to 30 new DL jobs *not* used in predictor training. All algorithms were evaluated with two different workload job duration patterns *W-Small* & *W-Large* to demonstrate the effectiveness of a utilisation-based packing approach. W-Small comprises DL jobs ranging from 1 minute to 1 hour with 60% of jobs less than 1800s, whereas W-Large has jobs ranging from 3 minutes to 1 hour with only 40% of jobs less than 1800s, therefore W-Large has longer job duration in comparison to W-Small. Additionally, we assigned 20% higher probability to W-Large jobs to utilise a higher batch size, $\geq$ 128, in order to produce DL jobs with >60% utilisation. W-Small and W-Large are adopted and modified from the job distributions derived from prior work [81]. JCT was controlled by terminating jobs at specified epoch numbers to emulate JCT patterns of production systems. Note that over 70% to 80% of total production DL jobs have been shown to require a single-GPU (§3.1.3), and hence for our testbed experiments, we focused on DL jobs requiring a single-GPU for training. Secondly, 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 [81, 182, 246, 266]–introduces further JCT heterogeneity, making it difficult to fairly measure potential trade-off gains between resource utilisation against JCT increase when co-locating DL jobs, therefore distributed DL jobs were not included.

- **Large-scale Evaluation Workload.** To demonstrate scalability, we evaluated Horus's performance in simulation by replaying the 398 jobs from Apollo's production trace (§3.1.2). The jobs have (i) duration ranging from 10 minutes to 25,654 minutes, (ii) GPU utilisation ranging from 1% to 98.6% and (iii) GPU memory consumption ranging from 655MiB to 31.7GiB. Each DL job is assumed to consume 12 CPU and 60 GiB memory per task for simplicity. A job is finished when its execution time count is up, as discussed previously in §5.1.1.

Parameters & experiment configuration. The job waiting time is more significant in large-scale multi-tenant clusters [108, 266], thus in our experimental testbed experiments, Horus was configured to operate with the number of queues $k = 1$ and a backfilling buffer size $\beta = 15$ to maximise the scheduling throughput and cluster utilisation. This was set to demonstrate scheduling throughput. In our large-scale simulation, we additionally evaluate job waiting time improvement by including *Horus-f* with $k = 3$ in order to evaluate fairness. Moreover, we added the Gandiva [246] time slicing approach to OBP in order to fully evaluate the overhead of context-switching DL jobs and trade-off between job waiting time and makespan in large-scale cluster. Our Python-based simulator does not capture the precise effects of GPU kernel-level characteristics or internal job progress, therefore, PAB was not included in the simulation. We repeat each experiment five times each, and the results are reported using the arithmetic mean.

5.2 Empirical Study Results

5.2.1 Makespan Analysis

As shown in Table 5.2, Horus successfully schedules all DL jobs with the lowest makespan of 204 and 212 minutes across W-Small and W-Large, respectively, and is equivalent to a 23.7% – 30.7% improvement against FIFO, an 18.4%–23.3% improvement over PAB, and a 9.4% – 10.8% improvement over OBP.

Workload	Algorithm	Avg.(mins)	St. Dev.(mins)	Gain
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%**
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%**

Table 5.2: Makespan statistics.

We observe that OBP has the second lowest makespan, achieving 225 and 238 minutes across W-Small and W-Large as shown in Figure 5.2 and Table 5.2. OBP outperforms PAB due to the latter algorithm incurring additional overhead when determining whether slowdown occurred (threshold violation) after the initial co-location decision. Since W-Large have 20% of jobs with longer duration in comparison to W-Small, the early execution of these longer jobs from co-location would enable makespan improvement. We observe the worst makespan Horus achieved still outperforms the mean makespan achieved by other algorithms as shown in Figure 5.2.

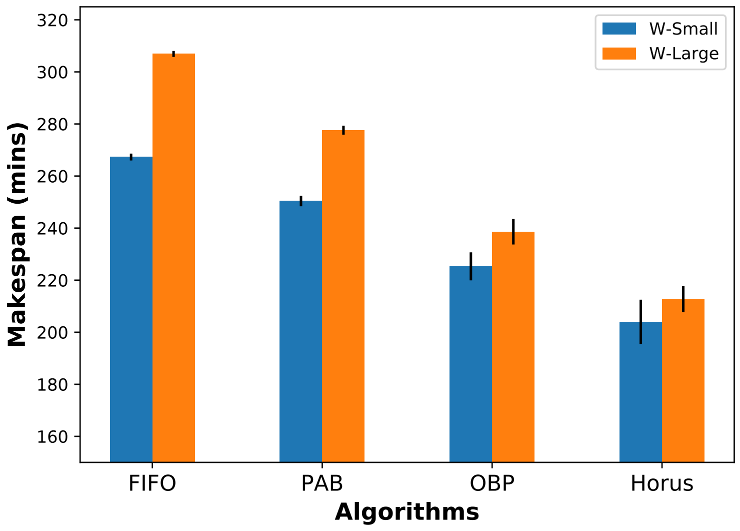


Figure 5.2: Makespan comparison, lower is better.

5.2.2 JCT Analysis

All co-location approaches introduce slowdown with PAB having the smallest slowdown (8.2% – 10.3%), Horus achieving the second-smallest slowdown (17.3% – 28.1%) and OBP achieving the highest slowdown (21.8% – 30.3%). Figure 5.3 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.

All co-location approaches suffer greater slowdown in W-Small in comparison to W-Large as shown in Table 5.3. 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 of (or the entire) GPU. Although FIFO achieves the fastest average JCT, it has resulted in the largest makespan and lowest GPU Utilisation due to longer queuing times and isolation of the GPUs. We observe that the JCTs standard deviation between Horus and OBP are similar as shown in Figure 5.3, however makespan achieved by Horus is lower, this is because the backfilling and prediction-based scheduling approach allows better co-location of

Workload	Algorithm	Avg. (s)	St. Dev (s)	Slowdown
W-Small	FIFO	1618.1	874.5	-
	PAB	1784.8	1047.5	10.3%
	OBP	2108.9	1236.8	30.3%
	Horus	**2073.6**	**1200.0**	**28.1%**
W-Large	FIFO	1869.7	1054.3	-
	PAB	2024.1	1121.0	8.2%
	OBP	2277.5	1293.1	21.8%
	Horus	**2193.8**	**1307.3**	**17.3%**

Table 5.3: JCT statistics.

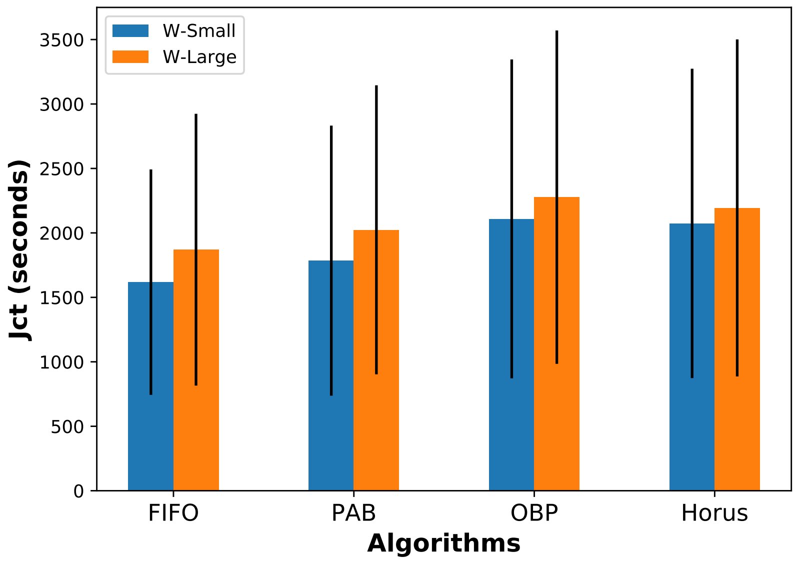


Figure 5.3: JCT comparison, lower is better.

suitable jobs. When considering the majority of DL jobs are for experimental and debugging purposes [95, 108], we believe that this is an acceptable tradeoff as engineers can obtain quicker feedback due to earlier execution of their training jobs as shown in the improved makespan results. Figure 5.4 depicts the JCT CDF between the algorithms.

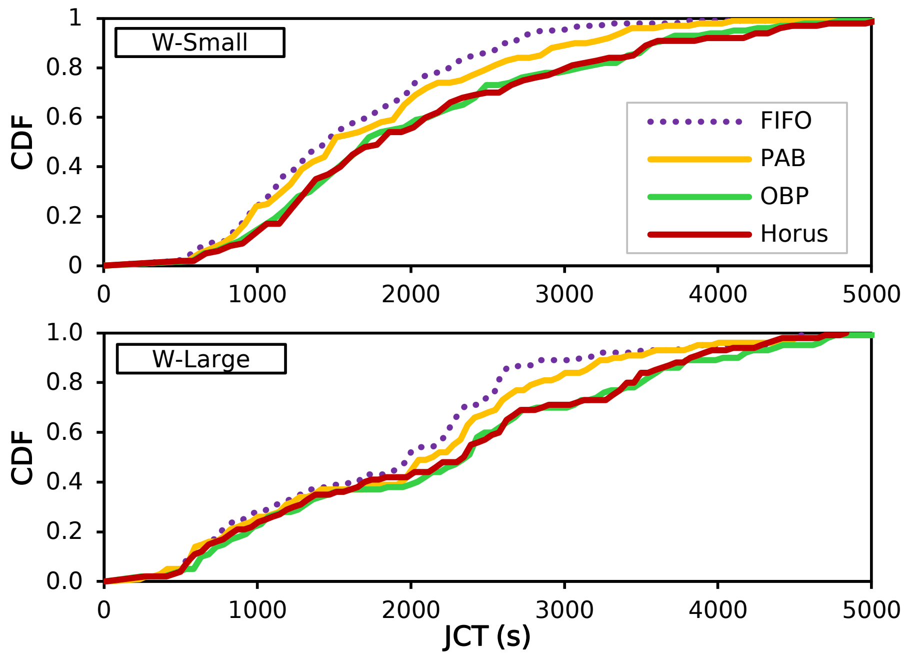


Figure 5.4: Job Completion Time (JCT) in testbed cluster experiments.

5.2.3 GPU Utilisation Analysis

Horus achieves the highest overall cluster resource utilisation in all experiment runs as shown in Figure 5.5 and Table 5.4, reflected by an average 60.1% GPU utilisation and 69.6% GPU utilisation across the experiments. We observed that in some experiments runs of W-Small, both Horus and OBP can experience up to 30 minutes of DL cluster resource utilisation of only 3–5%. This is because a small portion of our generated DL jobs have long epoch times, yet exhibit low GPU utilisation, according to JCT distribution presented in [81] and findings in Section 3.1. When omitting such tail 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 utilisation compared to FIFO due to their ability to perform co-location, OBP is able to achieve higher utilization as a result of its rapid scheduling cycle. In contrast, PAB incurs additional scheduling

Workload	Algorithm	Avg.	Std. Dev	Gain
W-Small	FIFO	45.2	14.3	-
	PAB	49.5	17.5	9.5%
	OBP	56.8	21.1	25.7%
	Horus	**60.1**	**19.9**	**33.0%**
W-Large	FIFO	43.1	16.7	-
	PAB	47.1	21.4	9.3%
	OBP	59.7	27.2	38.5%
	Horus	**69.6**	**26.9**	**61.5%**

Table 5.4: Cluster GPU utilisation statistics, higher is better.

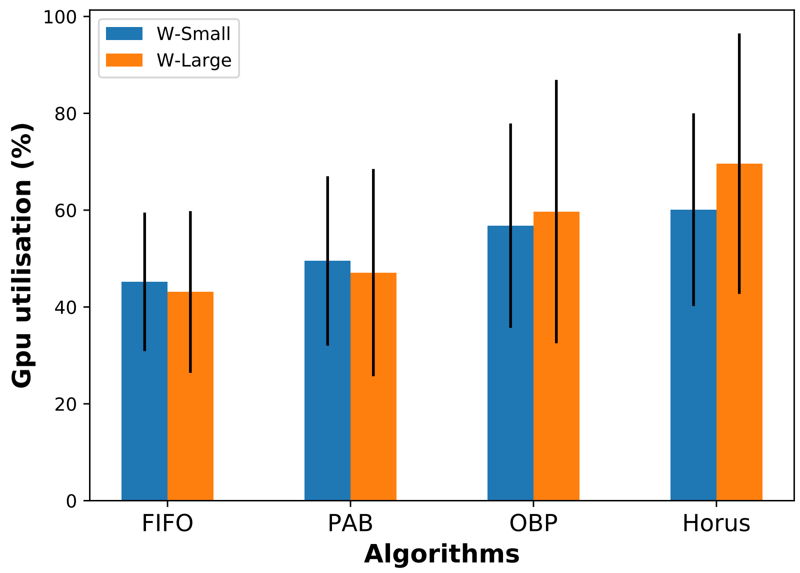


Figure 5.5: GPU utilisation comparison, higher is better.

waiting time in order to profile a scheduled job's stable performance, this results in a total of $n \times T_{wait}$ waiting time, where T_{wait} is the time it takes for a job to preprocess data and start its iterative training loop [247]. Interestingly, Horus's ability to effectively co-locate jobs, achieving higher DL job throughput and GPU Utilisation

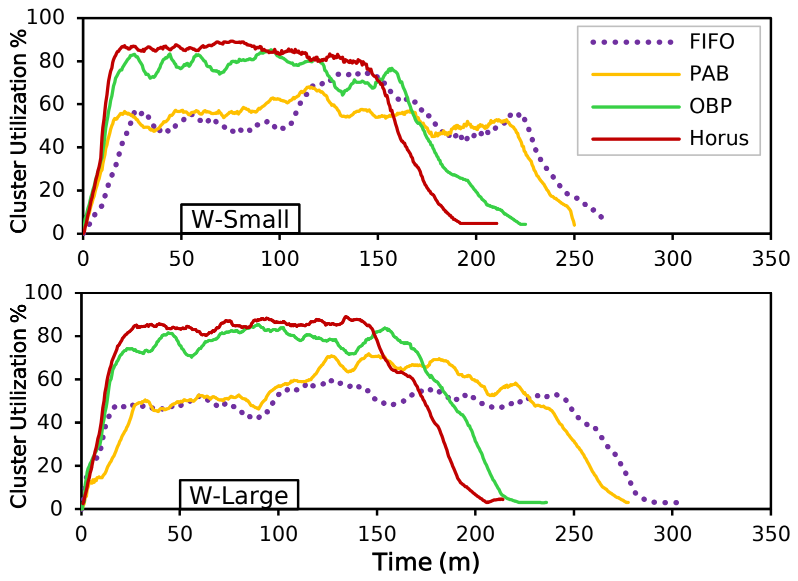


Figure 5.6: Average cluster GPU utilisation in empirical study experiments.

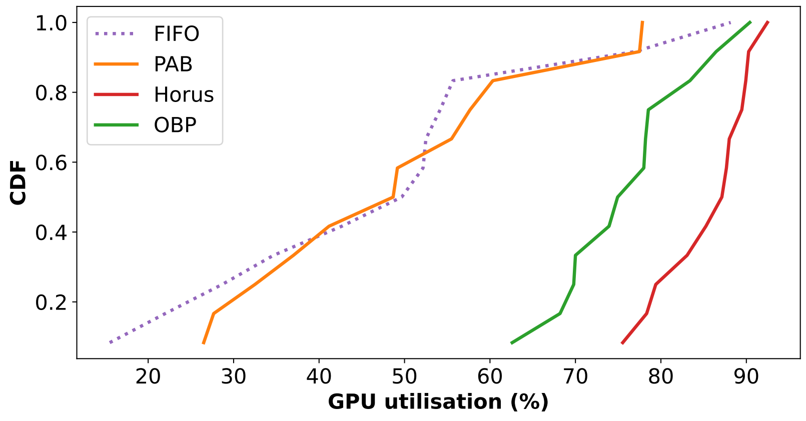


Figure 5.7: Sampled GPU utilisation CDF for W-Small workload.

will paradoxically induce interference and consequent JCT slowdown. Horus, does however, still achieve a lower JCT in comparison to OBP, and when considering our gains to resource utilisation and makespan, we view this as an acceptable trade-off. Figure 5.6 depicts the mean GPU utilisation of the testbed cluster during the experiments, and Figure 5.7 depicts the CDF of the GPU utilisation at the time when the cluster was highly utilised for W-Small. We observe that Horus successfully achieves the highest utilisation usage across the GPUs during the experiments.

5.2.4 Sensitivity Analysis

In order to explore how Horus performs under various configurations and parameter choices, we study how Horus operated under different configurations in the experimental testbed cluster for its predictor accuracy and queue size buffer β.

Prediction Error. Since Horus's approach is based on accurate GPU utilisation prediction, it is beneficial to evaluate the impact of scheduling decisions when prediction error increases and decide when to re-train the GPU utilisation prediction model [185]. Thus, we investigated how Horus operated under different predictor accuracy by introducing additional error, $\pm e$, onto the predicted GPU utilisation of each DL job, where 0% is the original prediction.

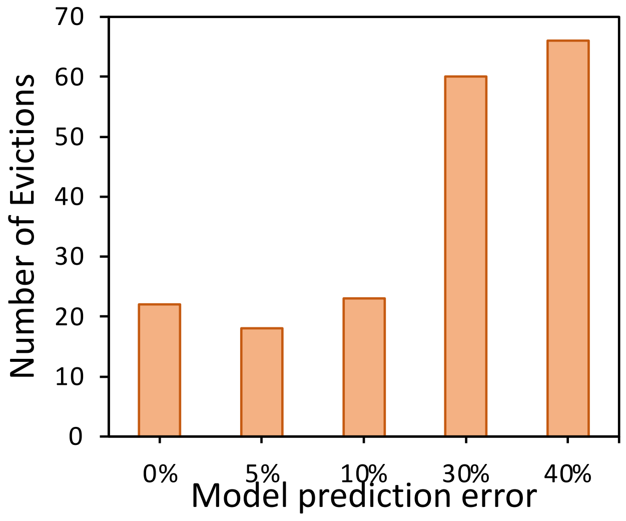


Figure 5.8: GPU utilisation prediction error and evictions statistics

When introducing artificial error e into the prediction output, we observe that the number of DL job failures increases by up to three times when the prediction error is high as shown in Figure 5.8. This is likely resultant from our cost calculation in Section 4.3.2, when GPU utilisation cost C^{GUtil} appears to be of medium cost, the total cost likely neglected the GPU memory cost C^{GMem} due to the weighted parameters ω (Equation 4.7), causing job failures due to inaccurate GPU memory estimation, thus causing OOM errors. Such results show that during a warm-up phase where large prediction errors are observed, it may be beneficial to assign a higher weighting on C^{GMem} until the ML predictor is trained to a satisfactory accuracy or when the predicted quality degrades after a period of continuous online evaluation. Finally, periodic monitoring of GPU utilisation error should be conducted and trigger a re-training step when prediction error starts to deviate more than $\pm$ 10%.

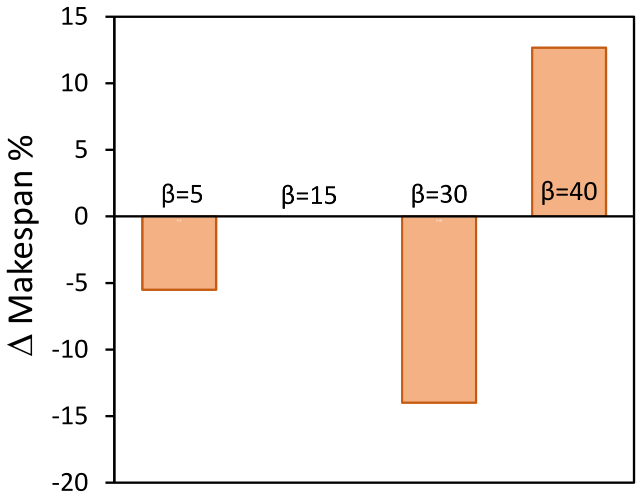


Figure 5.9: Beta configurations and makespan changes.

Queue Buffer Size. Altering queue buffer size β affects workload makespan, ranging from a makespan decrease of 14% to an increase of 12% as shown in Figure 5.9. We observe that β=30 performs the most effectively in experiment runs. While it is intuitive to assume that a larger queue size allows a scheduler to determine better co-location combinations, we found that across experiment runs, β=45 resulted in a 12% makespan increase. Because Horus prioritises making the best placement for co-location, we found that DL jobs with very high or very low GPU utilisation and memory consumption requirements were de-prioritised as placement candidates for co-location, and were forced to execute near the end of experiment runs. In comparison, β=5 indicates a selection of fewer candidates results in less effective placement for co-location, thus indicating that the queue buffer size β is likely affected by DL cluster configuration and DL jobs size.

5.3 Large-scale Trace-driven Simulation Results

5.3.1 Makespan Analysis

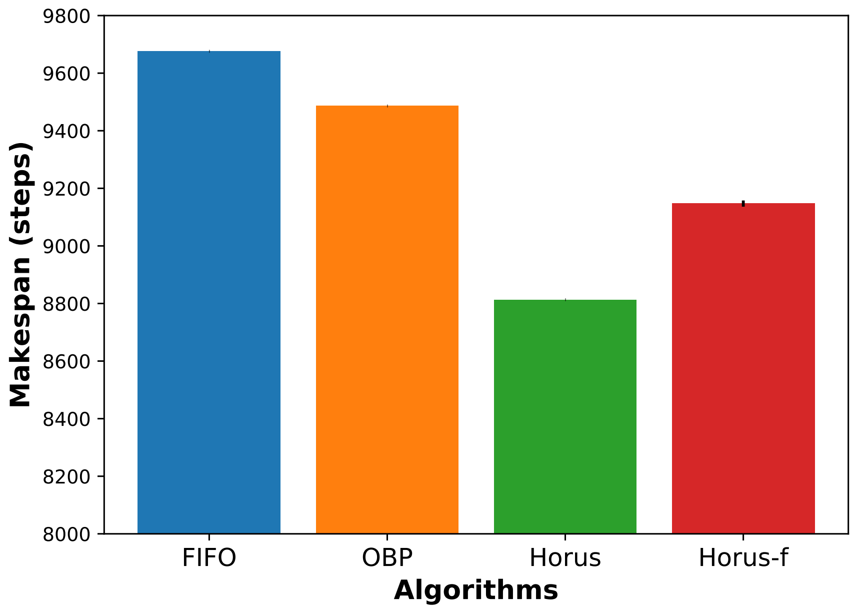


Figure 5.10: Makespan comparison, lower is better.

Similar to the testbed experiments, both Horus and Horus-f approaches resulted in the fastest makespan up to hundreds of scheduling decision steps, 8933 and 9121, respectively, as shown in Figure 5.10. FIFO resulted in the longest makespan (9676 steps) as expected due to dedicated GPUs, followed by OBP (9486 steps). Horus has improved the makespan by 8% compared to FIFO. Interestingly, by adding the fairness consideration into Horus, Horus-f in simulation resulted in a slightly slower makespan, a 2% makespan increase. This is due to the overhead of clustering DL jobs at each scheduling step. The results demonstrate that in larger-scale DL training clusters, both Horus approaches still successfully scheduled jobs, whilst minimising interference and outperforming other co-location approaches in terms of makespan.

5.3.2 JCT Analysis

In large-scale simulation, both Horus approaches incurred a performance slowdown on impacted jobs over 1.39x – 2.29x in comparison to OBP, as shown in Table 5.5. This is due to the large number of jobs in Apollo having ≥50% GPU utilisation as discussed in Section 3.1 and co-locating these jobs incurred slowdown due to GPU over-commitment. Although both Horus approaches degrade JCT higher than OBP, both Horus approaches still achieve faster makespan due to backfilling and co-locating suitable jobs that minimise interference. This phenomenon is similar to previous JCT observations in our empirical testbed study.

Algorithm	Avg.	Med.	St. Dev	Reduction
OBP	162.12	144.0	213.72	–
Horus	225.43	147.0	481.78	1.39x
Horus-f	371.69	157.0	737.83	2.29x

Table 5.5: JCT statistics for impacted DL jobs in simulation.

When considering the entire JCT distribution, there is only a minor degradation when compares to FIFO, 0.2% as shown in Table 5.6. The reasons for the phenomena

Algorithm	Avg.	Med.	St. Dev	Reduction
FIFO	270.60	119.29	614.56	–
OBP	271.06	121.47	614.58	0.2%
Horus	271.19	121.15	614.62	0.2%
Horus-f	271.18	119.29	614.8	0.2%

Table 5.6: JCT statistics in simulation.

are because Apollo jobs' arrival patterns were sparse, and therefore the DL jobs did not experience high GPU interference in the cluster.

5.3.3 GPU Utilisation Analysis

It is observable that both Horus and Horus-f achieve the highest cluster GPU utilisation, as shown in Figure 5.11. At scheduling step 800, Horus and Horus-f achieve 60.85% and 60.78% average GPU utilisation respectively, while FIFO and OBP are averaging 53.6% and 49.96% respectively. OBP is lower than FIFO in comparison to FIFO, this could be due to the time slicing for DL jobs can introduce additional scheduling overhead, i.e., frequent switching and re-scheduling of jobs. The period of re-scheduling and migration contributes to lower average cluster GPU utilisation.

When including the standard deviation, we observe that Horus and Horus-f can achieve average cluster GPU utilisation of $\geq$80%, outperforming FIFO and OBP significantly when compared against FIFO and OBP. This is due to Horus's efficient packing, which leads to high GPU utilisation early in the large-scale simulation and tail jobs contributing to the low GPU utilisation as shown in Figure 5.12. We speculate there are more jobs in the early scheduling rounds, could be due to working and holidays difference. The tail job behaviour manifests due to few but long-running jobs existing in the Apollo trace as discussed in §3.1, and in empirical study (§5.2.3).

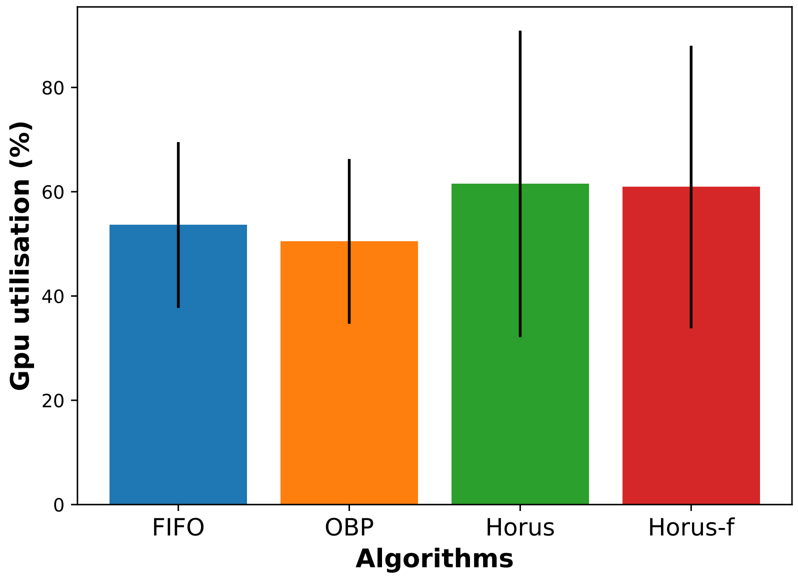


Figure 5.11: GPU utilisation comparison, higher is better.

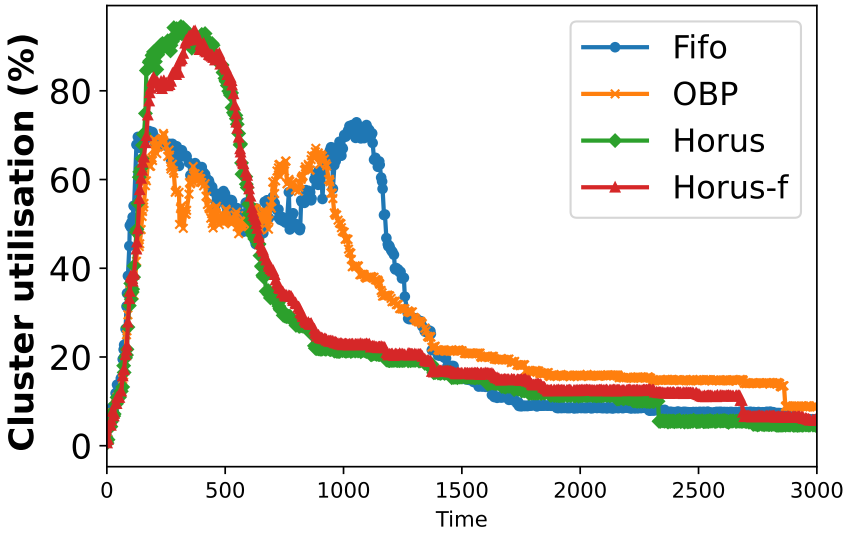


Figure 5.12: Cluster GPU utilisation in simulation.

5.3.4 Waiting Time Analysis

Combining fair queueing and co-location, Horus-f achieves the lowest average waiting time (132.9 steps) across all scheduling approaches, as shown in Figure 5.13. Table 5.7

shows Horus-f achieves a 71.5% job waiting time reduction when compared with FIFO.

Algorithm	Avg.	Med.	St. Dev	Reduction
FIFO	466.2	463.1	327.7	–
OBP	347.8	351.4	248.9	25.4%
Horus	142.3	119.3	120.3	69.5%
Horus-f	132.9	106.7	116.6	71.5%

Table 5.7: Job waiting time (steps) in simulation.

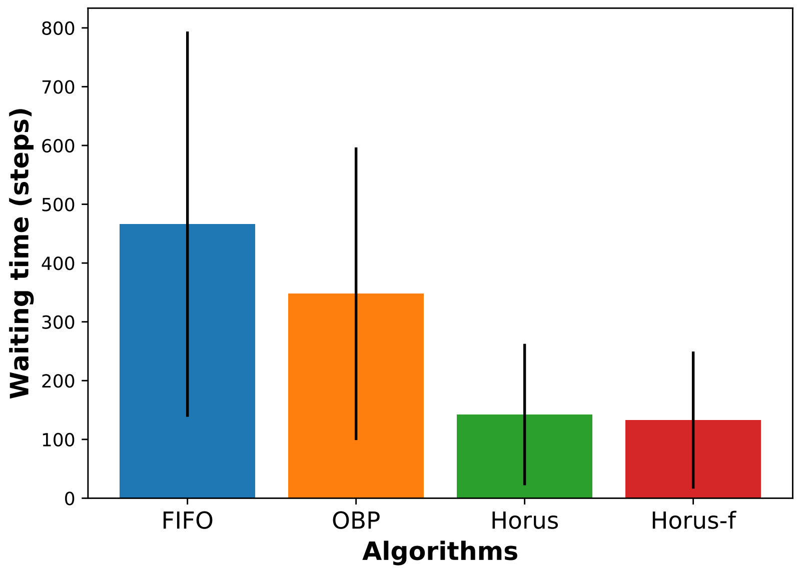


Figure 5.13: Job waiting time comparison, lower is better.

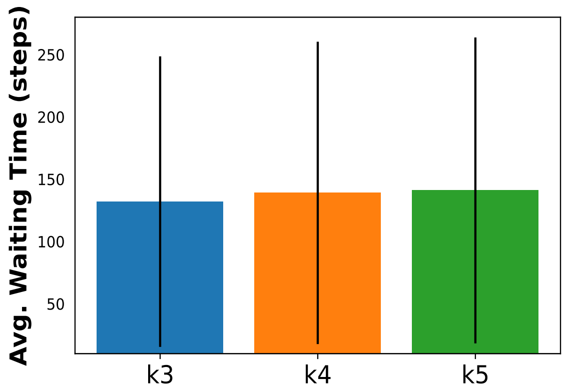
Moreover, We observe that Horus achieves the second-lowest average waiting time (142.3 steps). The fair queuing mechanism alone results in an approximately 10% lower median job waiting time when compared to Horus with $k = 1$, 106.7 and 119.3 respectively. Although OBP is co-location enabled, the mean waiting time only improves 25.4% when compared against FIFO. Since jobs in Apollo have long job duration, time slicing will have to go through many rounds for each job, leading to high makespan, i.e., when a large queue of jobs is waiting for their respective time slice t, they still have to wait for previous jobs to have obtained their $t-1$ time slices. The results demonstrate that Horus-f is desirable when the cluster is divided into multiple tenants and when jobs have distinct classes of duration [95].

5.3.5 Sensitivity Analysis

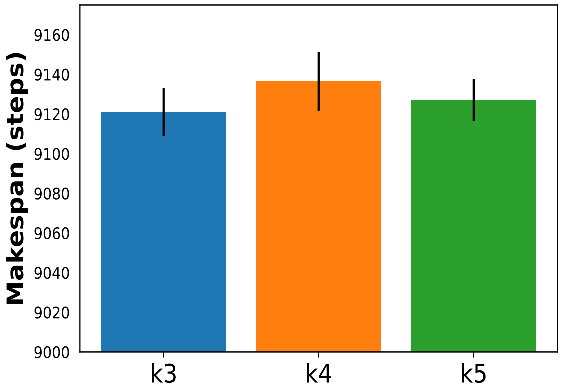
Similar to the empirical testbed experiments, we conduct sensitivity analysis in simulation by examining (i) the configurable number of queues with various values – 3, 4, and 5; and (ii) the queue buffer size β.

Number of queues. We observe that the number of queues does not significantly affect Horus scheduling performance, as shown in Fig. 5.14a. When k=5, the mean waiting time is degraded by 6% when compared to k=3, averaging 141.3 and 132.8, respectively. The reason for higher waiting time with higher number of queues could be due to when job clustering overhead. Similarly, the makespan performance has degraded slightly when the number of queues increased, as shown in Figure. 5.14b. Configuring the number of queues is not the focus in this thesis, however, we speculate this is workload and cluster size dependent.

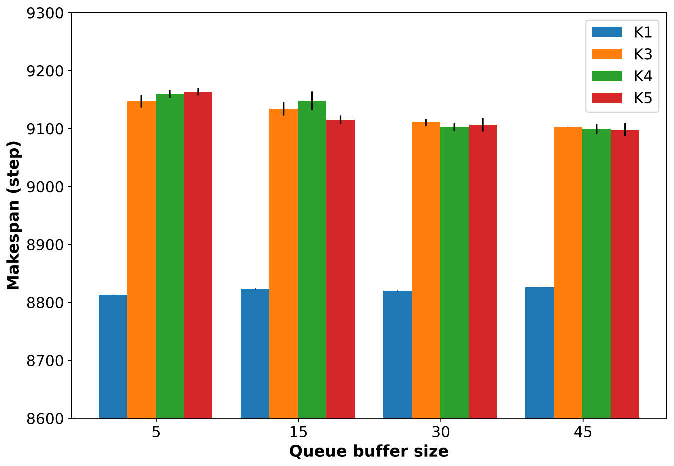
Queue buffer size. Figure 5.14c depicts the makespan achieved across different queue buffer size and queue settings. We observe that Horus with k=1 achieves the lowest makespan across all queue buffer size configurations. While the queue buffer size β does not affect Horus (k=1) makespan, we can observe that when β is increasing, Horus-f ($k \geq 1$) makespan improved slightly. This is due to backfilling scheduler can consider more variations of DL jobs and pack them according to GPU utilisation.



(a) Average waiting time sensitivity.



(b) Makespan sensitivity.



(c) Queue buffer size sensitivity.

Figure 5.14: Sensitivity analysis in simulation.

5.3.6 Simulation validity

Compared with our empirical testbed experiments, it is clear to see that the Horus scheduling approach performs similarly in simulation settings with a production trace, where Horus makes tradeoffs between JCT and resource efficiency, outperforms both FIFO, and OBP, in terms of makespan and GPU utilisation. In simulation, we also focus on the bin packing scheduling approaches rather than modelling performance tracking and job throttling. Horus uses the derived interference models (Eq. 3.2 and 3.3) to capture and be faithful to the real-world effect of job slowdown.

5.4 Discussion

5.4.1 Benefits

The above experiments have shown that Horus can bring several benefits to DL training clusters.

- **Improved cluster average GPU utilisation.** Horus has demonstrated that its prediction-based and cost-based approach can achieve high average GPU utilisation at $\geq$60% across empirical study and large-scale simulation, a sizable improvement over existing DL resource managers. This is due to the ability to leverage *apriori* knowledge of GPU utilisation before execution, and backfilling to enable safe co-location of suitable jobs without head-of-line blocking. This contrasts with reactive approaches that have to mitigate interference after initial placement decisions due to GPU over-commitment.
- **Improved scheduling makespan.** Makespan are improved by 8% to 30% across large-scale simulation and empirical study, respectively, when compared against other approaches. Improving makespan is equivalent to efficient utilisation of GPU resources [51]. This is due to safe co-location of suitable jobs (prediction) and considering a range of jobs (backfilling) allowing them to execute early while minimising interference overhead.
- **Reduced job waiting time.** As a result of early execution of suitable jobs, job waiting time is reduced, demonstrated by Horus improving waiting time by 71.5% when compared to FIFO. A benefit of our approach is that it provides earlier execution of scheduled DL models, and therefore ML engineers can detect poor DNN architecture designs or hyperparameter configurations quicker, leading to highly efficient usage of GPU resources and engineering effort.

5.4.2 Limitations

Horus demonstrated three key benefits in DL training cluster scheduling, however, there are several limitations:

- **Large-scale monitoring:** The Horus approach requires accurate monitoring of cluster resources to schedule jobs based on utilisation. Although Horus scales well when the cluster size is within the region of hundreds of machines, when cluster size is larger (thousands of GPU devices), frequent and detailed metric collection can introduce high CPU load to the monitoring framework, DL jobs and Horus application controller. One way to mitigate this challenge is to introduce additional components for monitoring and temporary blacklist GPUs for co-location when utilisation information is not available. Additionally, the GPU cluster state can be partitioned for distributed monitoring [5, 63].

- **Periodic predictor evaluation** – Inaccurate GPU utilisation prediction could worsen scheduling performance, and therefore reduce cluster GPU utilisation and cluster resource availability. In order to trigger a re-training step, keeping track of GPU utilisation prediction error rate is necessary. However, this approach could momentarily introduce high CPU load at run time, in the region of minutes, and therefore, an additional component could be introduced in the Horus architecture, in order to re-evaluate the prediction performance periodically. Specifically, one way to safeguard inaccurate prediction is by maintaining a sliding window of inaccuracy measurement. When the inaccuracy is greater than or equal to a threshold, Horus can disable the prediction engine code path, fallback to default scheduling approach without prediction [145].

- **Backfilling configuration:** Backfilling can enable higher resource utilisation when suitable jobs exist [62]. However, as demonstrated in both sensitivity analysis across empirical study and large-scale simulation, this requires careful tuning of the buffer size parameter. This is further complicated when cluster size and job size distribution can impact the effect of backfilling. It is common for scheduling approaches to be evaluated using a cluster simulator [76, 144]. Therefore, one way to tune these parameters is to adapt a feedback-based approach [106] with the use of a trace-based cluster simulator.

- **Locality consideration:** 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 work, 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. For jobs requiring multiple GPUs, our approach can be complementary to locality-based approaches that currently exist [81, 266], i.e., leveraged the cost model in our algorithm 2, and a potential future avenue is to calculate the tradeoff between locality and interference.

5.4.3 Scalability

The Horus prediction engine inference time overhead is under a second, we consider this overhead as acceptable, as most production DL training cluster schedulers have a soft constraint of making a scheduling decision under a few seconds due to the majority of GPU jobs being longer than 1-minute [95, 246, 247]. When there are bursts in job submission, if the prediction engine becomes the bottleneck of the DL training cluster, it is possible to deploy replicas of the prediction engine and load-balance between the prediction engines to cope with scheduling throughput. Since our scheduling cost model can be implemented as a plugin to cluster schedulers that follow the filter+cost heuristic policy [24, 83, 118], our approach would be bounded by $O(Ndk)$, where N is the number of sampled nodes, d is the number of GPUs and k is the number of backfill jobs. The precise configuration of sampling ratios and number of backfill jobs can be tuned according to the job submission throughput. The complexity is lowered to $O(Nd)$ when backfilling is disabled, results in similar time complexity to production cluster schedulers that employ sampling approach [83].

5.5 Summary

In Chapter 2, we detailed several goals for improved DL training cluster scheduling (§2.4.3.2). Within a set of experiments, Horus has been able to achieve these goals:

- Horus has been shown to successfully schedule DL training jobs in a GPU-load aware manner and avoid exclusively locking a GPU, addressing challenge 1 – exclusive lock and load-agnostic scheduling. Horus improved GPU utilisation by up to 61.5% when compared against FIFO in experimental testbed cluster and 15% in large-scale simulation. Horus can achieve the highest GPU utilisation when compared against existing co-location enabled approaches.

- By proactively predicting GPU utilisation and inferring the likely interference effect, Horus is able to mitigate severe performance degradation between DL jobs, addressing challenge 2 – placement sensitivity. Horus improved makespan by up to 30.7% when compared with FIFO in an experimental testbed cluster and up to 8% in large-scale simulation. Horus is able to achieve the fastest makespan when compared against existing co-location enabled approaches.

- Horus can lead to improved waiting time, addressing challenge 3 – multi-tenancy and high waiting time. Horus improved waiting time by 69.5% – 59.0% when compared to FIFO and OBP. Moreover, when including a weighted fair queuing approach, Horus-f improved waiting time by 71.5% and 61.8% when compared to

FIFO and OBP. Horus can achieve the lowest job waiting time against existing co-location enabled approaches.

Finally, Horus is a proactive DL resource manager that can be complementary to existing approaches due to its extensibility, i.e., without modifying the underlying DL frameworks and core Kubernetes components.

Chapter 6

Conclusion

DL applications are here to stay. They are being increasingly integrated into real-world applications, from satnav mapping and weather prediction to protein synthesis. There are a plethora of DL models, both small (<100MiB) and large ($\geq$1GiB). These are largely trained by organisations using large-scale compute clusters for effective sharing of hardware between multiple-tenants to increase cost efficiency [108, 177, 266]. DL training compute clusters are unique due to job characteristics, hardware differences and organisational behaviours; The scale of these clusters is only likely to increase in the future. Therefore, it is important to use these large-scale DL training clusters as efficiently as possible.

However, as observed and presented in Chapter 2, existing resource managers deployed to manage DL training compute clusters are facing several challenges: (i) exclusive lock and load-agnostic scheduling, leading to low average cluster resource utilisation; (ii) placement sensitivity and locality-agnostic scheduling, decreasing job performance; and, (iii) multi-tenancy and resource fragmentation, leading to high job waiting times. Recently proposed DL training specific resource managers that bypass exclusive locks to enable job co-location lead to interference from poor co-location decisions resulting in performance slowdown. A common approach to mitigate interference in cluster resource managers is to employ isolated profiling and reactive DL job migration. A better way to tackle the interference challenge without involving profiling is to improve co-location decisions at scheduling run time. Existing DL training resource managers mainly focus on improving JCT and fairness, and there is a lack of innovation in addressing low average cluster resource utilisation.

6.1 Summary of Contributions

To address these challenges, we proposed a proactive approach to DL training resource scheduling that enables co-location and minimises interference by making better co-location decisions, while focusing on improving makespan, cluster GPU utilisation and respecting fairness. The key insight in this thesis is that DL computational architectures can provide relevant information for the cluster resource manager to infer the likely GPU memory consumption and GPU utilisation of a DL job, thus making better co-location decisions at scheduling run time.

- In Chapter 3, we presented an in-depth analysis of a large-scale DL training compute cluster to demonstrate the severity of low average cluster resource utilisation and production DL training job characteristics. We demonstrated the key information in existing DL training clusters, namely a large amount of single-GPU jobs (>70%) and low cluster GPU utilisation (<60%). We conducted an interference study for DL jobs and identified that GPU over-commitment (cumulative GPU utilisation) correlates with performance interference for DL jobs, and modelled the relationship between interference and pver-commit GPU utilisation with respect to GPU hardware by fitting the polynomial regression models. We characterised and discovered DL model features such as FLOP count, number of model parameters and GEMM, correlate with GPU utilisation, presenting opportunities for predicting GPU utilisation.

- Chapter 4 contributes by describing our predictive approach to workload scheduling in DL training compute cluster. Based on analysis from Chapter 3, we proposed predicting GPU utilisation using ML techniques and estimating the likely GPU memory consumption of a DL training job through an analytical model. We trained a ML model that can predict GPU utilisation to a satisfactory level (0.133 RMSLE and 0.077 MAPE), and extracted the most important features from these predictors (FLOPs, Parameters, Batch Size, Activations, GEMM, ReLU and Flatten). Our findings show that these features should be considered when designing DNN architectures that minimise hardware load. Finally, we presented Horus, a new DL training resource manager that is proactive, GPU load-aware, waiting-time-aware and interference-aware that is based on a cost-model.

- Finally, Chapter 5 evaluated Horus through empirical testbed experimentation and a large-scale trace-driven simulation. We found that Horus improves the quality of co-location decisions over existing and state-of-the-art approaches since Horus considers GPU over-commitment behaviour, and proactively estimates GPU utilisation and memory consumption, to avoid co-location interference where possible. Horus successfully achieves the highest cluster average resource utilisation (up to 69.6%), lowest makespan and lowest job waiting time across co-location enabled approaches. Moreover, Horus can be complementary to existing DL training cluster resource managers that focus on various objectives due to its extensibility. Horus can also scale to DL training clusters with hundreds of machines.

6.2 Review of Research Questions

The work presented in these chapters collectively answers our research questions introduced in Chapter 1.

[**RQ1**] *When co-locating DL training workload onto shared hardware to improve utilisation, what are the relationships between their respective utilisation profiles and interference?*

In Section 3.2.1, we have shown that performance interference introduces slowdown for DL jobs. The performance slowdown induced is correlated with GPU over-commitment, i.e., the cumulative GPU utilisation of the co-located DL jobs. Furthermore, we showed that GPU hardware heterogeneity affects the performance slowdown by fitting regression models, and are reflected by the coefficients of the models. Thus, when DL job GPU utilisation is known prior to execution, resource management frameworks can leverage this information to improve co-location decisions.

[**RQ2**] *How to determine DL training workload utilisation profile efficiently without online profiling?*

In Section 3.3, we have demonstrated that DNN architecture and its inputs provide crucial information that are correlated with GPU utilisation, such as FLOPs, parameters, batch size, and activations, by looking through the Pearson and Spearman-rank correlation. We identified that there are moderate linear (≥ 0.4) and monotonic relationships (≥ 0.7) between the DL operations (e.g., matrix manipulation, GEMM, ReLU and Flatten) within common DNN architectures and GPU utilisation. Based on our findings, we trained a machine learning model that can predict GPU utilisation efficiently given these features.

[RQ3] *Can resource management frameworks leverage this resource utilisation estimate to provide better co-location scheduling decisions than existing frameworks, thus improving average cluster utilisation and lower job waiting time?*

In chapter 4, we proposed Horus, a ML-integrated approach (prediction engine) in DL training cluster resource scheduling that utilises a cost model to decide on DL job co-location. The prediction engine comprises a ML model for predicting GPU utilisation and an analytical model for estimating GPU memory consumption. We have shown that Horus achieves the highest average cluster utilisation, fastest makespan and lowest job waiting time in our empirical experimentation and in a large-scale trace-driven simulation. We discussed that this approach requires re-training of the ML model and a monitoring system in order to allow Horus to work effectively.

With Horus, we demonstrate that a proactive approach to DL training jobs scheduling with co-location that minimises interference without dedicated profiling can be achieved. This is done via (i) discovering DL training job interference can be estimated from GPU over-commitment degree, (ii) understanding the DL training job characteristics that drive GPU resource consumption, i.e., DNN models and their operations, and (iii) collect relevant data and train an ML model to estimate GPU resource consumption. We demonstrate that ML's integrated approach to cluster scheduling is not only easily trainable, not only limited to CPU-based workload; but can also scale to a large DL training compute clusters.

6.3 Future Work

There is, however, ample opportunity for future work to extend Horus in DL cluster resource scheduling. Horus has demonstrated that a modern ML-driven approach to DL training job scheduling is both practical and appealing. This is only an initial step in the direction of making DL cluster scheduling more aware of its infrastructure and applications. To make Horus a favourable DL compute cluster resource manager, there are five possible avenues for future research.

6.3.1 Alternative Machine Learning Predictor

An accurate predictor can further enhance co-location scheduling decisions. In the ML system community, utilising a DNN-based model to predict tensor programs has gained great attention due to the increase in prediction accuracy over traditional ML models [13, 220]. For instance, Gao et al. [67] leverages GNN to predict the end-to-end runtime latency of a DL model. While our XGBoost model provides a satisfactory level of prediction accuracy, an interesting idea would be to leverage a DNN-based model to predict the GPU utilisation. A potential obstacle to overcome is the source of a sufficiently large training dataset for the DNN-based model, as additional components are needed to acquire such a dataset.

6.3.2 Hardware Feature Integration

Accelerators and ecosystems for DL have increased in recent years. For example, NVIDIA A100 [171] can statically partition its compute and memory resources for different workloads. Therefore, the ability to determine the compute and memory resources needed (i.e., the partition size) for a particular DL workload and space share between jobs could dramatically increase resource utilisation. By integrating hardware features directly into the prediction engine and DL workload characteristics, the cluster operator might be able to draw on the predictive benefits without having to manually profile the best configuration for hardware partitioning and tuning the parameters of the DL models for a resource efficient DL cluster resource management.

6.3.3 Low-level IR Features Integration

While the high-level DL model's IR emits information about the DL models, the key implementation details are missing such as the number of loops, tiling structure and temporary memory allocation. This is because low-level IRs are emitted via DL compilers that give information on the number of threads needed to execute, number of loops and hardware intrinsic API used [35]. Incorporating these features into the prediction engine can provide further hints on resource consumption patterns. This is already a well-explored area in the DL compiler community [13, 220]. An interesting next step would be to study whether these features have a direct relationship with resource consumption and whether co-located DL jobs with similar number of threads, loops and intrinsic calls will manifest in high interference. Rammer [138], an approach that leverages low-level IR features including the number of threads and the number of processing elements, has indicated that a co-ordinated device scheduling approach would work at the device level. Making it applicable to cluster scheduling might be fruitful to explore.

6.3.4 Energy-aware scheduling

To further extend the hardware resource selection challenge, energy has now become one of the major concerns for hyperscale compute cluster operators [124, 132, 244]. The ability to select the right resource configuration for a particular energy budget is a crucial challenge to improve cost efficiency for DL jobs. Moreover, energy demand correlates to user request demand and the ability to deliver energy per DL job. It is essential to deliver the right energy level to a DL job to satisfy their service level objectives while minimising energy consumption. However, different ML models could behave differently with different energy levels [235], this becomes much more complex when DL jobs are co-located.

It would be interesting to implement an energy-aware proactive scheduling approach. To explore some of these questions, an additional energy selection ML model could be trained and implemented. For example, an implementation using reinforcement learning similar to [143] would allow the ML model to dynamically evaluate hardware configuration according to cluster state (including power).

6.3.5 Inference Scheduling

Horus is a DL training cluster resource manager, however, further extensions can be made to support inference jobs. This is both timely and important. Meta has indicated that 200 trillion inferences are performed per day [128]. We suspect that major hyperscale cluster operators that provide DL-driven decision-making services perform a similar number of inferences. Moreover, Mixture-of-Expert (MoE) models, i.e., many models combined into one, are gaining traction due to its ability for multitasks inference. Accommodating MoE models in inference can be crucial in improving resource efficiency. For example, can the number of inferences for the intermediate models within MoE be evaluated to dynamically adjust the resources assigned; trading-off accuracy with resources? For example, must a model be invoked for a request during a high cluster contention period? These questions would be both interesting and practical to explore.

Horus already has some of these elements, modelling interference with respect to hardware and DL training jobs. It would be interesting to broaden it to take those extra dimensions into account.

6.4 Summary

As applications are increasingly powered by DL models, the training infrastructure for DL models must be supported by better cluster resource management software.

Existing DL training clusters exhibit low average GPU utilisation and a common way to improve utilisation is by co-locating workload. However, interference is a fundamental challenge in all resource sharing compute systems, and it manifests when naively co-locating unsuitable workload.

In this dissertation, we have made the case for new proactive DL training cluster resource management that addresses performance interference. We have shown interference-aware proactive DL scheduling possible via extracting common features from a DL model's high-level IR and predicting its resource consumption, informing the cluster resource scheduler allowing it to make better co-location decisions without dedicated profiling. We proposed Horus as part of the resource management framework for DL training clusters.

With these contributions, Horus can make DL training compute clusters more resource efficient without incurring excessive maintenance.

References

[1] ACL Machine Translation (WMT19). *Shared Task: Machine Translation of News*. (Accessed on 01/07/2020). URL: `http://www.statmt.org/wmt19/translation-task.html`.

[2] *3.3. metrics and scoring: quantifying the quality of predictions*. 2021. URL: `https://scikit-learn.org/stable/modules/model_evaluation.html#mean-squared-log-error` (visited on 12/10/2021).

[3] Samuli Aalto, Urtzi Ayesta, and Rhonda Righter. "On the Gittins index in the M/G/1 queue". In: *Queueing Systems* 63.1 (2009), pp. 437–458.

[4] Martın Abadi et al. "Tensorflow: A system for large-scale machine learning". In: *12th USENIX symposium on operating systems design and implementation (OSDI 16)*. 2016, pp. 265–283.

[5] Colin Adams et al. "Monarch: Google's planet-scale in-memory time series database". In: *Proceedings of the VLDB Endowment* (2020), pp. 3181–3194.

[6] Ravichandra Addanki et al. "Placeto: Efficient progressive device placement optimization". In: *NIPS Machine Learning for Systems Workshop*. 2018.

[7] *Alibaba Cluster Data: using 270 gb of open source data to understand Alibaba data centers*. 2021. URL: `https://www.alibabacloud.com/blog/594340` (visited on 10/23/2021).

[8] *AlphaFold: a solution to a 50-year-old grand challenge in biology*. 2021. URL: `https://deepmind.com/blog/article/alphafold-a-solution-to-a-50-year-old-grand-challenge-in-biology` (visited on 10/28/2021).

[9] Marcelo Amaral, Jordà Polo, et al. "Topology-aware GPU scheduling for learning workloads in cloud environments". In: *ACM SC*. 2017.

[10] *Amazon EC2 Spot Instances*. 2022. URL: `https://aws.amazon.com/ec2/spot/` (visited on 01/19/2022).

[11] Pradeep Ambati et al. "Providing SLOs for Resource-Harvesting VMs in Cloud Platforms". In: *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, 2020. URL: `https://www.usenix.org/conference/osdi20/presentation/ambati`.

[12] *Artificial Intelligence Autopilot*. 2021. URL: `https://www.tesla.com/AI` (visited on 10/25/2021).

[13] Riyadh Baghdadi et al. "A Deep Learning Based Cost Model for Automatic Code Optimization". In: *Proceedings of Machine Learning and Systems* (2021).

[14] Paul Barham et al. "Xen and the art of virtualization". In: (2003). DOI: `10.1145/945445.945462`.

[15] L.A. Barroso, J. Dean, and U. Holzle. "Web search for a planet: The Google cluster architecture". In: *IEEE Micro* 23.2 (2003), pp. 22–28. DOI: `10.1109/MM.2003.1196112`.

[16] Luiz André Barroso, Jimmy Clidaras, and Urs Hölzle. "The datacenter as a computer: An introduction to the design of warehouse-scale machines". In: *Synthesis lectures on computer architecture* 8.3 (2013), pp. 1–154.

[17] Christopher Berner et al. "Dota 2 with large scale deep reinforcement learning". In: *arXiv preprint arXiv:1912.06680* (2019).

[18] David Bernstein. “Containers and cloud: From lxc to docker to kubernetes”. In: *IEEE Cloud Computing* 1.3 (2014), pp. 81–84.

[19] Ricardo Bianchini et al. “Toward ML-Centric Cloud Platforms”. In: *Commun. ACM* 63.2 (Jan. 2020), pp. 50–59. ISSN: 0001-0782. DOI: `10.1145/3364684`. URL: `https://doi.org/10.1145/3364684`.

[20] Christopher M Bishop. “Pattern recognition”. In: (2006).

[21] Flavio Bonomi et al. “Fog computing and its role in the internet of things”. In: *Proceedings of the first edition of the MCC workshop on Mobile cloud computing*. 2012, pp. 13–16.

[22] Mirela Madalina Botezatu et al. “Predicting disk replacement towards reliable data centers”. In: *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. 2016, pp. 39–48.

[23] Leo Breiman. “Random forests”. In: *Machine learning* 45.1 (2001), pp. 5–32.

[24] Brendan Burns et al. “Borg, omega, and kubernetes”. In: *Communications of the ACM* 59.5 (2016), pp. 50–57.

[25] Han Cai, Ligeng Zhu, and Song Han. “ProxylessNAS: Direct Neural Architecture Search on Target Task and Hardware”. In: *International Conference on Learning Representations*. 2018.

[26] *CFS Scheduler — The Linux Kernel Documentation*. 2021. URL: `https://www.kernel.org/doc/html/latest/scheduler/sched-design-CFS.html` (visited on 11/02/2021).

[27] Jianmin Chen et al. “Revisiting distributed synchronous SGD”. In: *arXiv preprint arXiv:1604.00981* (2016).

[28] Li Chen et al. “Auto: Scaling deep reinforcement learning for datacenter-scale automatic traffic optimization”. In: *Proceedings of the 2018 conference of the ACM special interest group on data communication*. 2018, pp. 191–205.

[29] Quan Chen et al. “Prophet: Precise QoS prediction on non-preemptive accelerators to improve utilization in warehouse-scale computers”. In: *ACM ASPLOS*. 2017.

[30] Quan Chen et al. “Baymax: QoS awareness and increased utilization for non-preemptive accelerators in warehouse scale computers”. In: *ACM SIGPLAN Notices* (2016).

[31] Shuang Chen, Christina Delimitrou, and José F Martınez. “Parties: QoS-aware resource partitioning for multiple interactive services”. In: *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*. 2019, pp. 107–120.

[32] Tianqi Chen and Carlos Guestrin. “Xgboost: A scalable tree boosting system”. In: *Proceedings of the 22nd acm sigkdd international conference on knowledge discovery and data mining*. 2016, pp. 785–794.

[33] Tianqi Chen et al. “Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems”. In: *arXiv preprint arXiv:1512.01274* (2015).

[34] Tianqi Chen et al. “Learning to optimize tensor programs”. In: *Advances in Neural Information Processing Systems* (2018).

[35] Tianqi Chen et al. “TVM: An automated end-to-end optimizing compiler for deep learning”. In: *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. 2018, pp. 578–594.

[36] Yujun Chen et al. “Outage prediction and diagnosis for cloud service systems”. In: *The World Wide Web Conference*. 2019, pp. 2659–2665.

[37] Yunpeng Chen et al. “Dual path networks”. In: *NeurIPS*. 2017.

[38] Sharan Chetlur et al. “cudnn: Efficient primitives for deep learning”. In: *arXiv preprint arXiv:1410.0759* (2014).

[39] Kyunghyun Cho et al. “Learning Phrase Representations using RNN Encoder–Decoder for Statistical Machine Translation”. In: *ACM EMNLP*. 2014.

[40] Eric Chung et al. “Serving DNNs in Real Time at Datacenter Scale with Project Brainwave”. In: *IEEE Micro* 38.2 (2018), pp. 8–20.

[41] Eli Cortez et al. “Resource Central: Understanding and Predicting Workloads for Improved Resource Management in Large Cloud Platforms”. In: *Proceedings of the 26th Symposium on Operating Systems Principles*. SOSP ’17. 2017.

[42] Paul Covington, Jay Adams, and Emre Sargin. “Deep neural networks for YouTube recommendations”. In: *Proceedings of the 10th ACM conference on recommender systems*. 2016, pp. 191–198.

[43] *CS231n convolutional neural networks for visual recognition*. 2021. URL: `https://cs231n.github.io/convolutional-networks/#case`.

[44] Haskell B Curry. “The method of steepest descent for non-linear minimization problems”. In: *Quarterly of Applied Mathematics* 2.3 (1944), pp. 258–261.

[45] Scott Cyphers et al. “Intel ngraph: An intermediate representation, compiler, and executor for deep learning”. In: *arXiv preprint arXiv:1801.08058* (2018).

[46] Zihang Dai et al. “CoAtNet: Marrying Convolution and Attention for All Data Sizes”. In: *arXiv preprint arXiv:2106.04803* (2021).

[47] James Davidson et al. “The YouTube Video Recommendation System”. In: *Proceedings of the Fourth ACM Conference on Recommender Systems*. RecSys ’10. Barcelona, Spain, 2010.

[48] Arnaud de Myttenaere et al. “Mean Absolute Percentage Error for regression models”. In: *Neurocomputing* (2016). Advances in artificial neural networks, machine learning and computational intelligence.

[49] Jeffrey Dean and Luiz André Barroso. “The tail at scale”. In: *Communications of the ACM* 56.2 (2013), pp. 74–80.

[50] Jeffrey Dean et al. “Large scale distributed deep networks”. In: *Advances in neural information processing systems* 25 (2012), pp. 1223–1231.

[51] Christina Delimitrou and Christos Kozyrakis. “Paragon: QoS-Aware Scheduling for Heterogeneous Datacenters”. In: *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS ’13. 2013.

[52] Christina Delimitrou and Christos Kozyrakis. “Quasar: Resource-efficient and QoS-aware cluster management”. In: *ACM SIGPLAN Notices* 49.4 (2014), pp. 127–144.

[53] Jia Deng et al. “ImageNet: A large-scale hierarchical image database”. In: *2009 IEEE Conference on Computer Vision and Pattern Recognition*. 2009, pp. 248–255. DOI: 10.1109/CVPR.2009.5206848.

[54] Jacob Devlin et al. “Bert: Pre-training of deep bidirectional transformers for language understanding”. In: *arXiv preprint arXiv:1810.04805* (2018).

[55] Jianbo Dong et al. “Eflops: Algorithm and system co-design for a high performance distributed training platform”. In: *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE. 2020, pp. 610–622.

[56] Alexey Dosovitskiy et al. “An image is worth 16x16 words: Transformers for image recognition at scale”. In: *arXiv preprint arXiv:2010.11929* (2020).

[57] IBM Education. *What is Supervised Learning?* 2021. URL: `https://www.ibm.com/cloud/learn/supervised-learning#toc-unsupervis-Fo3jDcmY` (visited on 10/28/2021).

[58] IBM Education. *What is Unsupervised Learning?* 2021. URL: `https://www.ibm.com/cloud/learn/unsupervised-learning` (visited on 10/28/2021).

[59] Andre Esteva et al. “Dermatologist-level classification of skin cancer with deep neural networks”. In: *Nature* 542.7639 (2017), pp. 115–118.

[60] Richard Evans et al. “Protein complex prediction with AlphaFold-Multimer”. In: *bioRxiv* (2021).

[61] *Feature importances with a forest of trees*. 2021. URL: `https://scikit-learn.org/stable/auto_examples/ensemble/plot_forest_importances.html` (visited on 12/10/2021).

[62] Dror G Feitelson and Ahuva Mu’alem Weil. “Utilization and predictability in scheduling the IBM SP2 with backfilling”. In: *Proceedings of the First Merged International Parallel Processing Symposium and Symposium on Parallel and Distributed Processing*. IEEE. 1998, pp. 542–546.

[63] Yihui Feng et al. “Scaling large production clusters with partitioned synchronization”. In: *2021 USENIX Annual Technical Conference (USENIX ATC 21)*. 2021, pp. 81–97.

[64] Ian Foster et al. “Cloud Computing and Grid Computing 360-Degree Compared”. In: *2008 Grid Computing Environments Workshop*. 2008, pp. 1–10. DOI: `10.1109/GCE.2008.4738445`.

[65] Yoav Freund and Robert E Schapire. “A Decision-Theoretic Generalization of On-Line Learning and an Application to Boosting”. In: *Journal of Computer and System Sciences* 55.1 (1997).

[66] Joshua Fried et al. "Caladan: Mitigating interference at microsecond timescales". In: *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. 2020, pp. 281–297.

[67] Yanjie Gao et al. *Runtime Performance Prediction for Deep Learning Models with Graph Neural Network*. Tech. rep. Microsoft, Feb. 2021. URL: `https://www.microsoft.com/en-us/research/publication/runtime-performance-prediction-for-deep-learning-models-with-graph-neural-network/`.

[68] Panagiotis Garefalakis et al. "Medea: Scheduling of Long Running Applications in Shared Production Clusters". In: *Proceedings of the Thirteenth EuroSys Conference*. EuroSys '18. Porto, Portugal: Association for Computing Machinery, 2018.

[69] Leon A Gatys, Alexander S Ecker, and Matthias Bethge. "Image style transfer using convolutional neural networks". In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2016, pp. 2414–2423.

[70] Felix A Gers, Jürgen Schmidhuber, and Fred Cummins. "Learning to forget: Continual prediction with LSTM". In: *ICANN*. 1999.

[71] *GitHub - Arm-software/ArmNN: Arm NN ML software*. 2021. URL: `https://github.com/Arm-software/armnn` (visited on 10/25/2021).

[72] *GitHub - baidu-research/baidu-allreduce*. 2017. URL: `https://github.com/baidu-research/baidu-allreduce` (visited on 11/09/2021).

[73] *GitHub - nvidia/dcgm: nvidia data center gpu manager (dcgm) is a project for gathering telemetry and measuring the health of nvidia gpus*. 2021. URL: `https://github.com/NVIDIA/DCGM` (visited on 11/27/2021).

[74] *GitHub - onnx/onnx: open standard for machine learning interoperability*. 2021. URL: `https://github.com/onnx/onnx` (visited on 11/01/2021).

[75] *GitHub - prometheus/prometheus: the prometheus monitoring system and time series database.* 2017. URL: https://github.com/prometheus/prometheus (visited on 11/18/2021).

[76] Ionel Gog et al. "Firmament: Fast, centralized cluster scheduling at scale". In: *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. 2016, pp. 99–115.

[77] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep learning*. MIT press, 2016.

[78] *Google Cluster Data*. 2010. URL: https://ai.googleblog.com/2010/01/google-cluster-data.html.

[79] Priya Goyal et al. "Accurate, large minibatch sgd: Training imagenet in 1 hour". In: *arXiv preprint arXiv:1706.02677* (2017).

[80] Robert Grandl et al. "Multi-resource packing for cluster schedulers". In: *ACM SIGCOMM Computer Communication Review* 44.4 (2014), pp. 455–466.

[81] Juncheng Gu et al. "Tiresias: A GPU cluster manager for distributed deep learning". In: *USENIX NSDI*. 2019.

[82] Arpan Gujarati et al. "Serving DNNs like clockwork: Performance predictability from the bottom up". In: *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. 2020, pp. 443–462.

[83] Ori Hadary et al. "Protean: VM Allocation Service at Scale". In: *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. 2020, pp. 845–861.

[84] Dongyoon Han, Jiwhan Kim, and Junmo Kim. "Deep pyramidal residual networks". In: *IEEE CVPR*. 2017.

[85] Charles R Harris et al. “Array programming with NumPy”. In: *Nature* 585.7825 (2020), pp. 357–362.

[86] Kim Hazelwood et al. “Applied machine learning at facebook: A datacenter infrastructure perspective”. In: *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE. 2018, pp. 620–629.

[87] Kaiming He et al. “Deep residual learning for image recognition”. In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2016, pp. 770–778.

[88] Xiangnan He et al. “Neural collaborative filtering”. In: *Proceedings of the 26th international conference on world wide web*. 2017, pp. 173–182.

[89] Dan Hendrycks and Kevin Gimpel. “Gaussian error linear units (gelus)”. In: *arXiv preprint arXiv:1606.08415* (2016).

[90] Benjamin Hindman et al. “Mesos: A platform for fine-grained resource sharing in the data center.” In: *NSDI*. Vol. 11. 2011. 2011, pp. 22–22.

[91] Sepp Hochreiter and Jürgen Schmidhuber. “Long short-term memory”. In: *Neural computation* 9.8 (1997), pp. 1735–1780.

[92] E. Hokenek, R.K. Montoye, and P.W. Cook. “Second-generation RISC floating point with multiply-add fused”. In: *IEEE Journal of Solid-State Circuits* 25.5 (1990), pp. 1207–1213. DOI: `10.1109/4.62143`.

[93] Andrew Howard et al. “Searching for mobilenetv3”. In: *IEEE ICCV*. 2019.

[94] Andrew G Howard et al. “Mobilenets: Efficient convolutional neural networks for mobile vision applications”. In: *arXiv preprint arXiv:1704.04861* (2017).

[95] Qinghao Hu et al. “Characterization and Prediction of Deep Learning Workloads in Large-Scale GPU Datacenters”. In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. SC ’21. St. Louis, Missouri: Association for Computing Machinery, 2021.

[96] Gao Huang et al. “Densely connected convolutional networks”. In: *IEEE ICCV*. 2017, pp. 4700–4708.

[97] Yanping Huang et al. “Gpipe: Efficient training of giant neural networks using pipeline parallelism”. In: *Advances in neural information processing systems* 32 (2019), pp. 103–112.

[98] David H Hubel and Torsten N Wiesel. “Receptive fields, binocular interaction and functional architecture in the cat’s visual cortex”. In: *The Journal of physiology* 160.1 (1962), pp. 106–154.

[99] Peter J Huber. “Robust estimation of a location parameter”. In: *Breakthroughs in statistics*. Springer, 1992, pp. 492–518.

[100] Peter J Huber. *Robust statistics*. Vol. 523. John Wiley & Sons, 2004.

[101] Jack Tigar Humphries et al. “ghOSt: Fast Flexible User-Space Delegation of Linux Scheduling”. In: *ACM SOSP*. 2021.

[102] Rob J Hyndman and Anne B Koehler. “Another look at measures of forecast accuracy”. In: *International journal of forecasting* 22.4 (2006), pp. 679–688.

[103] Forrest N Iandola et al. “SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and¡ 0.5 MB model size”. In: *arXiv:1602.07360 [cs.CV]* (2016).

[104] *Improving real-time performance by utilizing Cache Allocation Technology*. 2015. URL: `https://www.intel.com/content/dam/www/public/us/en/documents/white-papers/cache-allocation-technology-white-paper.pdf` (visited on 11/15/2021).

[105] Anil K Jain and Richard C Dubes. *Algorithms for clustering data*. Prentice-Hall, Inc., 1988.

[106] Philipp K Janert. *Feedback control for computer systems: introducing control theory to enterprise programmers.* " O'Reilly Media, Inc.", 2013.

[107] Pawel Janus and Krzysztof Rzadca. "SLO-aware colocation of data center tasks based on instantaneous processor requirements". In: *Proceedings of the 2017 Symposium on Cloud Computing* (Sept. 2017).

[108] Myeongjae Jeon et al. "Analysis of large-scale multi-tenant GPU clusters for DNN training workloads". In: *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. 2019, pp. 947–960.

[109] Yimin Jiang et al. "A Unified Architecture for Accelerating Distributed DNN Training in Heterogeneous GPU/CPU Clusters". In: *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, Nov. 2020.

[110] Adwait Jog et al. "Application-aware memory system for fair and efficient execution of concurrent gpgpu applications". In: *Proceedings of workshop on general purpose processing using GPUs*. 2014, pp. 1–8.

[111] Tyler Johnson et al. "AdaScale SGD: A user-friendly algorithm for distributed training". In: *International Conference on Machine Learning*. PMLR. 2020, pp. 4911–4920.

[112] Chris Jones et al. *Google - Site Reliability Engineering*. 2017. URL: `https://sre.google/sre-book/service-level-objectives` (visited on 10/31/2021).

[113] Norman P Jouppi et al. "In-datacenter performance analysis of a tensor processing unit". In: *Proceedings of the 44th annual international symposium on computer architecture*. 2017, pp. 1–12.

[114] *July 2019 – Common Crawl*. 2019. URL: `https://commoncrawl.org/2019/07/` (visited on 11/06/2021).

[115] Shinpei Kato et al. "TimeGraph: GPU scheduling for real-time multi-tasking environments". In: *USENIX ATC*. 2011.

[116] Sam Kaufman et al. "A Learned Performance Model for Tensor Processing Units". In: *Proceedings of Machine Learning and Systems* 3 (2021).

[117] Guolin Ke et al. "Lightgbm: A highly efficient gradient boosting decision tree". In: *Advances in neural information processing systems* 30 (2017).

[118] Xiaodi Ke et al. "Fundy: A Scalable and Extensible Resource Manager for Cloud Resources". In: *2021 IEEE 14th International Conference on Cloud Computing (CLOUD)*. 2021.

[119] Young Geun Kim and Carole-Jean Wu. "Autoscale: Energy efficiency optimization for stochastic edge inference using reinforcement learning". In: *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE. 2020.

[120] Diederik P Kingma and Jimmy Ba. "Adam: A method for stochastic optimization". In: *arXiv preprint arXiv:1412.6980* (2014).

[121] Günter Klambauer et al. "Self-normalizing neural networks". In: *Proceedings of the 31st international conference on neural information processing systems*. 2017, pp. 972–981.

[122] Alex Krizhevsky, Geoffrey Hinton, et al. *Learning multiple layers of features from tiny images*. Tech. rep. Citeseer, 2009.

[123] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. "Imagenet classification with deep convolutional neural networks". In: *Advances in neural information processing systems* 25 (2012).

[124] Alok Kumbhare et al. “Prediction-based power oversubscription in cloud platforms”. In: *arXiv preprint arXiv:2010.15388* (2020).

[125] Oliver Lange and Luis Perez. *Traffic prediction with advanced Graph Neural Networks*. 2021. URL: `https://deepmind.com/blog/article/traffic-prediction-with-advanced-graph-neural-networks` (visited on 10/25/2021).

[126] Yann LeCun et al. “A theoretical framework for back-propagation”. In: *Proceedings of the 1988 connectionist models summer school*. Vol. 1. 1988.

[127] Yann LeCun et al. “Gradient-based learning applied to document recognition”. In: *Proceedings of the IEEE* 86.11 (1998), pp. 2278–2324.

[128] Kevin Lee, Vijay Rao, and William Arnold. *Accelerating Facebook’s infrastructure with application-specific hardware*. 2021. URL: `https://engineering.fb.com/2019/03/14/data-center-engineering/accelerating-infrastructure/` (visited on 10/25/2021).

[129] Sebastien Levy et al. “Predictive and Adaptive Failure Mitigation to Avert Production Cloud VM Interruptions”. In: *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. 2020.

[130] Feng Li et al. “Deep learning-based automated detection for diabetic retinopathy and diabetic macular oedema in retinal fundus photographs”. In: *Eye* (2021).

[131] Haoyuan Li. “Alluxio: A Virtual Distributed File System”. PhD thesis. EECS Department, University of California, Berkeley, May 2018. URL: `http://www2.eecs.berkeley.edu/Pubs/TechRpts/2018/EECS-2018-29.html`.

[132] Shaohong Li et al. “Thunderbolt: Throughput-Optimized, Quality-of-Service-Aware Power Capping at Scale”. In: *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. 2020.

[133] Timothy P. Lillicrap et al. “Continuous control with deep reinforcement learning”. In: *International Conference on Learning Representations*. 2016.

[134] Junyang Lin et al. “M6: A chinese multimodal pretrainer”. In: *arXiv preprint arXiv:2103.00823* (2021).

[135] Saifeng Liu et al. “Prostate cancer diagnosis using deep learning with 3D multiparametric MRI”. In: *Medical imaging 2017: computer-aided diagnosis*. Vol. 10134. International Society for Optics and Photonics. 2017.

[136] David Lo et al. “Heracles: Improving resource efficiency at scale”. In: *Proceedings of the 42nd Annual International Symposium on Computer Architecture*. 2015, pp. 450–462.

[137] Chuan Luo et al. “Correlation-aware heuristic search for intelligent virtual machine provisioning in cloud systems”. In: *Proceedings of the AAAI Conference on Artificial Intelligence*. Vol. 35. 14. 2021.

[138] Lingxiao Ma et al. “Rammer: Enabling Holistic Deep Learning Compiler Optimizations with rTasks”. In: *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. Nov. 2020.

[139] Ningning Ma et al. “Shufflenet v2: Practical guidelines for efficient cnn architecture design”. In: *ECCV*. 2018.

[140] Kiwan Maeng et al. “Understanding and Improving Failure Tolerant Training for Deep Learning Recommendation with Partial Recovery”. In: *Proceedings of Machine Learning and Systems* 3 (2021).

[141] Kshiteej Mahajan et al. “Themis: Fair and efficient GPU cluster scheduling”. In: *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*. 2020, pp. 289–304.

[142] Luo Mai et al. “KungFu: Making Training in Distributed Machine Learning Adaptive”. In: *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. 2020, pp. 937–954.

[143] Hongzi Mao et al. “Resource management with deep reinforcement learning”. In: *Proceedings of the 15th ACM workshop on hot topics in networks*. 2016.

[144] Hongzi Mao et al. “Learning scheduling algorithms for data processing clusters”. In: *Proceedings of the ACM Special Interest Group on Data Communication*. 2019, pp. 270–288.

[145] Hongzi Mao et al. “Towards safe online reinforcement learning in computer systems”. In: *33rd conference on neural information processing systems*.

[146] Jason Mars and Lingjia Tang. “Whare-map: heterogeneity in” homogeneous” warehouse-scale computers”. In: *Proceedings of the 40th Annual International Symposium on Computer Architecture*. 2013, pp. 619–630.

[147] Jason Mars, Lingjia Tang, and Robert Hundt. “Heterogeneity in “homogeneous” warehouse-scale computers: A performance opportunity”. In: *IEEE Computer Architecture Letters* 10.2 (2011), pp. 29–32.

[148] Jason Mars et al. “Bubble-up: Increasing utilization in modern warehouse scale computers via sensible co-locations”. In: *Proceedings of the 44th annual IEEE/ACM International Symposium on Microarchitecture*. 2011, pp. 248–259.

[149] Sam McCandlish et al. “An empirical model of large-batch training”. In: *arXiv preprint arXiv:1812.06162* (2018).

[150] Wes McKinney et al. “pandas: a foundational Python library for data analysis and statistics”. In: *Python for high performance and scientific computing* (2011).

[151] Gábor Melis, Chris Dyer, and Phil Blunsom. "On the state of the art of evaluation in neural language models". In: *arXiv preprint arXiv:1707.05589* (2017).

[152] Daniel A Menascé. "Virtualization: Concepts, applications, and performance modeling". In: *Int. CMG Conference*. 2005, pp. 407–414.

[153] Daniel Mendoza et al. "Interference-Aware Scheduling for Inference Serving". In: *Proceedings of the 1st Workshop on Machine Learning and Systems*. EuroMLSys '21. Online, United Kingdom, 2021, pp. 80–88.

[154] Stephen Merity et al. "Pointer sentinel mixture models". In: *ICLR* (2016).

[155] Paulius Micikevicius. *Analysis-Driven Optimization*. 2010. URL: `https://www.nvidia.com/content/PDF/sc_2010/CUDA_Tutorial/SC10_Analysis_Driven_Optimization.pdf` (visited on 11/21/2021).

[156] Paulius Micikevicius. *Performance Optimization: Programming Guidelines and GPU Architecture Reasons Behind Them*. 2013. URL: `https://on-demand.gputechconf.com/gtc/2013/presentations/S3466-Programming-Guidelines-GPU-Architecture.pdf` (visited on 11/02/2021).

[157] Azalia Mirhoseini et al. "Device placement optimization with reinforcement learning". In: *International Conference on Machine Learning*. PMLR. 2017, pp. 2430–2439.

[158] S Mostafa Mousavi et al. "Earthquake transformer—an attentive deep-learning model for simultaneous earthquake detection and phase picking". In: *Nature communications* 11.1 (2020), pp. 1–12.

[159] Sai Prashanth Muralidhara et al. "Reducing memory interference in multicore systems via application-aware memory channel partitioning". In: *2011 44th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE. 2011, pp. 374–385.

[160] Onur Mutlu and Jeremie S. Kim. "RowHammer: A Retrospective". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 39.8 (2020), pp. 1555–1571.

[161] Deepak Narayanan et al. "PipeDream: generalized pipeline parallelism for DNN training". In: *Proceedings of the 27th ACM Symposium on Operating Systems Principles*. 2019, pp. 1–15.

[162] Deepak Narayanan et al. "Heterogeneity-aware cluster scheduling policies for deep learning workloads". In: *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. 2020, pp. 481–498.

[163] Iyswarya Narayanan et al. "SSD failures in datacenters: What? when? and why?" In: *Proceedings of the 9th ACM International on Systems and Storage Conference*. 2016, pp. 1–11.

[164] Ripal Nathuji, Canturk Isci, and Eugene Gorbatov. "Exploiting platform heterogeneity for power efficient data centers". In: *Fourth International Conference on Autonomic Computing (ICAC'07)*. IEEE. 2007, pp. 5–5.

[165] Maxim Naumov et al. "Deep learning training in facebook data centers: Design of scale-up and scale-out systems". In: *arXiv preprint arXiv:2003.09518* (2020).

[166] Pandu Nayak. *Understanding searches better than ever before*. 2021. URL: `https://www.blog.google/products/search/search-language-understanding-bert/` (visited on 10/25/2021).

[167] Andrew Newell et al. "RAS: Continuously Optimized Region-Wide Datacenter Resource Allocation". In: *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*. SOSP '21. New York, NY, USA, 2021.

[168] Feng Niu et al. "Hogwild!: A lock-free approach to parallelizing stochastic gradient descent". In: *arXiv preprint arXiv:1106.5730* (2011).

[169] Dejan Novaković et al. "Deepdive: Transparently identifying and managing performance interference in virtualized environments". In: *2013 USENIX Annual Technical Conference (USENIX ATC 13)*. 2013, pp. 219–230.

[170] *NVIDIA A100 Tensor Core GPU Architecture*. 2021. URL: `https://images.nvidia.com/aem-dam/en-zz/Solutions/data-center/nvidia-ampere-architecture-whitepaper.pdf` (visited on 10/30/2021).

[171] *NVIDIA A100 Tensor Core GPU Datasheet*. 2021. URL: `https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/a100/pdf/a100-80gb-datasheet-update-nvidia-us-1521051-r2-web.pdf` (visited on 11/06/2021).

[172] *NVIDIA Tesla Volta Accelerator Graphics*. 2022. URL: `https://www.amazon.com/NVIDIA-Tesla-Volta-Accelerator-Graphics/dp/B07JVNHFFX` (visited on 01/16/2022).

[173] *NVIDIA Triton Inference Server*. 2021. URL: `https://developer.nvidia.com/nvidia-triton-inference-server` (visited on 11/01/2021).

[174] *nvidia-smi - NVIDIA System Management Interface program*. 2016. URL: `https://developer.download.nvidia.com/compute/DCGM/docs/nvidia-smi-367.38.pdf` (visited on 11/29/2021).

[175] Chandandeep Singh Pabla. "Completely fair scheduler". In: *Linux Journal* 2009.184 (2009), p. 4.

[176] Abhay K. Parekh and Robert G. Gallager. "A generalized processor sharing approach to flow control in integrated services networks: the multiple node case". In: *IEEE ACM Transactions on Networking* (1993).

[177] Jongsoo Park et al. "Deep learning inference in facebook data centers: Characterization, performance optimizations and hardware implications". In: *arXiv preprint arXiv:1811.09886* (2018).

[178] Adam Paszke et al. “Pytorch: An imperative style, high-performance deep learning library”. In: *Advances in neural information processing systems* 32 (2019), pp. 8026–8037.

[179] Pankesh Patel, Ajith H Ranabahu, and Amit P Sheth. “Service level agreement in cloud computing”. In: (2009).

[180] Tirthak Patel and Devesh Tiwari. “Clite: Efficient and QoS-aware co-location of multiple latency-critical jobs for warehouse scale computers”. In: *International Symposium on High Performance Computer Architecture (HPCA)*. IEEE. 2020, pp. 193–206.

[181] Yanghua Peng et al. “A generic communication scheduler for distributed DNN training acceleration”. In: *ACM SOSP*. 2019.

[182] Yanghua Peng et al. “Optimus: an efficient dynamic resource scheduler for deep learning clusters”. In: *Proceedings of the Thirteenth EuroSys Conference*. 2018, pp. 1–14.

[183] Jeffrey Pennington, Richard Socher, and Christopher D Manning. “Glove: Global vectors for word representation”. In: *Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP)*. 2014, pp. 1532–1543.

[184] Rajat Phull et al. “Interference-driven resource management for GPU-based heterogeneous clusters”. In: *ACM SC*. 2012.

[185] Neoklis Polyzotis and Matei Zaharia. *What can Data-Centric AI Learn from Data and ML Engineering?* 2021. arXiv: `2112.06439 [cs.LG]`.

[186] Junjie Qian, Taeyoon Kim, and Myeongjae Jeon. “Reliability of Large Scale GPU Clusters for Deep Learning Workloads”. In: *Companion Proceedings of the Web Conference 2021*. 2021, pp. 179–181.

[187] Aurick Qiao et al. “Pollux: Co-adaptive Cluster Scheduling for Goodput-Optimized Deep Learning”. In: *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*. 2021, pp. 1–18.

[188] Haoran Qiu et al. “FIRM: An Intelligent Fine-grained Resource Management Framework for SLO-Oriented Microservices”. In: *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. 2020, pp. 805–825.

[189] Alec Radford et al. “Improving language understanding by generative pre-training”. In: (2018).

[190] Alec Radford et al. “Language models are unsupervised multitask learners”. In: *OpenAI blog* 1.8 (2019), p. 9.

[191] Jonathan Ragan-Kelley et al. “Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines”. In: *Acm Sigplan Notices* 48.6 (2013), pp. 519–530.

[192] Samyam Rajbhandari et al. “Zero-infinity: Breaking the gpu memory wall for extreme scale deep learning”. In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 2021, pp. 1–14.

[193] Esteban Real et al. “Regularized evolution for image classifier architecture search”. In: *Proceedings of the aaai conference on artificial intelligence*. Vol. 33. 01. 2019, pp. 4780–4789.

[194] Charles Reiss, John Wilkes, and Joseph L Hellerstein. “Google cluster-usage traces: format+ schema”. In: *Google Inc., White Paper* (2011), pp. 1–14.

[195] Zujie Ren et al. “Workload characterization on a production Hadoop cluster: A case study on Taobao”. In: *2012 IEEE International Symposium on Workload Characterization (IISWC)*. 2012, pp. 3–13.

[196] Sean Robertson. *NLP From Scratch: Translation with a Sequence to Sequence Network and Attention — PyTorch Tutorials 1.10.0+cu102 documentation.* 2021. URL: `https://pytorch.org/tutorials/intermediate/seq2seq_translation_tutorial.html` (visited on 10/29/2021).

[197] Olaf Ronneberger, Philipp Fischer, and Thomas Brox. “U-net: Convolutional networks for biomedical image segmentation”. In: *International Conference on Medical image computing and computer-assisted intervention.* Springer. 2015, pp. 234–241.

[198] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. “Learning representations by back-propagating errors”. In: *Nature* 323.6088 (1986), pp. 533–536.

[199] Krzysztof Rzadca et al. “Autopilot: workload autoscaling at Google”. In: *Proceedings of the Fifteenth European Conference on Computer Systems.* 2020, pp. 1–16.

[200] Varun Sakalkar et al. “Data center power oversubscription with a medium voltage power plane and priority-aware capping”. In: *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems.* 2020, pp. 497–511.

[201] Mark Sandler, Andrew Howard, et al. “Mobilenetv2: Inverted residuals and linear bottlenecks”. In: *IEEE ICCV.* 2018.

[202] Thomas Schlegl et al. “Unsupervised anomaly detection with generative adversarial networks to guide marker discovery”. In: *International conference on information processing in medical imaging.* Springer. 2017, pp. 146–157.

[203] Bianca Schroeder and Garth A Gibson. “A large-scale study of failures in high-performance computing systems”. In: *IEEE transactions on Dependable and Secure Computing* 7.4 (2009).

[204] *scipy.stats.mstats.spearmanr — scipy v1.7.1*. 2021. URL: https://docs.scipy.org/doc/scipy/reference/generated/scipy.stats.mstats.spearmanr.html?highlight=spearman%5C%20rank (visited on 11/22/2021).

[205] *scipy.stats.pearsonr — scipy v1.7.1*. 2021. URL: https://docs.scipy.org/doc/scipy/reference/generated/scipy.stats.pearsonr.html (visited on 11/22/2021).

[206] David Sculley et al. "Hidden technical debt in machine learning systems". In: *Advances in neural information processing systems* 28 (2015).

[207] Mohammad Shahrad et al. "Serverless in the wild: Characterizing and optimizing the serverless workload at a large cloud provider". In: *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. 2020, pp. 205–218.

[208] Haichen Shen et al. "Nexus: a GPU cluster engine for accelerating DNN-based video analysis". In: *Proceedings of the 27th ACM Symposium on Operating Systems Principles*. 2019, pp. 322–337.

[209] Shaohuai Shi et al. "Towards Scalable Distributed Training of Deep Learning on Public Cloud Clusters". In: *Proceedings of Machine Learning and Systems*. Vol. 3. 2021, pp. 401–412.

[210] Weisong Shi et al. "Edge computing: Vision and challenges". In: *IEEE internet of things journal* 3.5 (2016), pp. 637–646.

[211] Konstantin Shvachko et al. "The hadoop distributed file system". In: *2010 IEEE 26th symposium on mass storage systems and technologies (MSST)*. Ieee. 2010, pp. 1–10.

[212] David Silver et al. "Mastering the game of go without human knowledge". In: *Nature* 550.7676 (2017), pp. 354–359.

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

[214] Karen Simonyan and Andrew Zisserman. “Very Deep Convolutional Networks for Large-Scale Image Recognition”. In: *ICLR*. 2015.

[215] *sklearn.metrics.mean*$_s$*quared*$_l$*og*$_e$*rror*. 2021. URL: `https://scikit-learn.org/stable/modules/generated/sklearn.metrics.mean_squared_log_error.html` (visited on 10/28/2021).

[216] *sklearn.preprocessing.standardscaler*. 2021. URL: `https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.StandardScaler.html` (visited on 11/23/2021).

[217] Casper Kaae Sønderby et al. “Metnet: A neural weather model for precipitation forecasting”. In: *arXiv preprint arXiv:2003.12140* (2020).

[218] Liuyihan Song et al. “Large-Scale Training System for 100-Million Classification at Alibaba”. In: *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery amp; Data Mining*. KDD ’20. New York, NY, USA, 2020, pp. 2909–2930.

[219] Srividya Srinivasan et al. “Characterization of backfilling strategies for parallel job scheduling”. In: *Proceedings. International Conference on Parallel Processing Workshop*. IEEE. 2002, pp. 514–519.

[220] Benoit Steiner et al. “Value learning for throughput optimization of deep learning workloads”. In: *Proceedings of Machine Learning and Systems* 3 (2021).

[221] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. MIT press, 2018.

[222] *System Architecture*. 2022. URL: `https://cloud.google.com/tpu/docs/system-architecture-tpu-vm`.

[223] Christian Szegedy et al. “Going Deeper with Convolutions”. In: *IEEE CVPR*. 2015.

[224] Mingxing Tan and Quoc Le. “Efficientnet: Rethinking model scaling for convolutional neural networks”. In: *International Conference on Machine Learning*. PMLR. 2019, pp. 6105–6114.

[225] Mingxing Tan et al. “Mnasnet: Platform-aware neural architecture search for mobile”. In: *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 2019, pp. 2820–2828.

[226] Chunqiang Tang et al. “Twine: a unified cluster management system for shared infrastructure”. In: *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. 2020, pp. 787–803.

[227] Prashanth Thinakaran et al. “Kube-Knots: Resource Harvesting through Dynamic Container Orchestration in GPU-based Datacenters”. In: *IEEE CLUSTER*. 2019.

[228] Ryan Tibshirani. *Lecture 19: November 5*. 2015. URL: `https://www.stat.cmu.edu/~ryantibs/convexopt-F18/scribes/Lecture_19.pdf` (visited on 11/19/2021).

[229] Muhammad Tirmazi et al. “Borg: The next Generation”. In: *Proceedings of the Fifteenth European Conference on Computer Systems*. EuroSys ’20. Heraklion, Greece, 2020. ISBN: 9781450368827.

[230] Yash Ukidave et al. “Mystic: Predictive scheduling for gpu based cloud servers using machine learning”. In: *IEEE IPDPS*. 2016.

[231] Nedeljko Vasić et al. "DejaVu: Accelerating Resource Allocation in Virtualized Environments". In: *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems.* ASPLOS '12. 2012.

[232] Ashish Vaswani et al. "Attention is all you need". In: *Advances in neural information processing systems.* 2017, pp. 5998–6008.

[233] Vinod Kumar Vavilapalli et al. "Apache hadoop yarn: Yet another resource negotiator". In: *Proceedings of the 4th annual Symposium on Cloud Computing.* 2013, pp. 1–16.

[234] Abhishek Verma et al. "Large-scale cluster management at Google with Borg". In: *Proceedings of the Tenth European Conference on Computer Systems.* 2015, pp. 1–17.

[235] Chengcheng Wan et al. "ALERT: Accurate Learning for Energy and Timeliness". In: *2020 USENIX Annual Technical Conference (USENIXATC 20).* 2020, pp. 353–369.

[236] Chen Wang. *Improving resource efficiency for Kubernetes clusters via load-aware scheduling — IBM Research Blog.* 2020. URL: `https://www.ibm.com/blogs/research/2020/11/resource-efficiency-kubernetes/` (visited on 11/10/2021).

[237] Endong Wang et al. "Intel math kernel library". In: *High-Performance Computing on the Intel® Xeon Phi™.* Springer, 2014, pp. 167–188.

[238] Guanhua Wang et al. "Blink: Fast and generic collectives for distributed ml". In: *Proceedings of Machine Learning and Systems* 2 (2020), pp. 172–186.

[239] H. Wang et al. "S-CDA: A Smart Cloud Disk Allocation Approach in Cloud Block Storage System". In: *ACM DAC.* 2020.

[240] Haojie Wang et al. “PET: Optimizing Tensor Programs with Partially Equivalent Transformations and Automated Corrections”. In: *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*. USENIX Association, July 2021, pp. 37–54.

[241] Yu Wang et al. “A Systematic Methodology for Analysis of Deep Learning Hardware and Software Platforms”. In: *MLSys*. 2020.

[242] *What is hyper-threading?* 2021. URL: `https://www.intel.com/content/www/us/en/gaming/resources/hyper-threading.html` (visited on 11/15/2021).

[243] Bichen Wu et al. “Fbnet: Hardware-aware efficient convnet design via differentiable neural architecture search”. In: *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 2019, pp. 10734–10742.

[244] Qiang Wu et al. “Dynamo: Facebook’s data center-wide power management system”. In: *ACM SIGARCH Computer Architecture News* 44.3 (2016), pp. 469–480.

[245] *xgboost/sklearn.py*. 2021. URL: `https://github.com/dmlc/xgboost/blob/01152f89ee585c42523add286a78ab9101a5a40a/python-package/xgboost/sklearn.py#L1143` (visited on 12/10/2021).

[246] Wencong Xiao et al. “Gandiva: Introspective cluster scheduling for deep learning”. In: *USENIX OSDI*. 2018.

[247] Wencong Xiao et al. “AntMan: Dynamic Scaling on GPU Clusters for Deep Learning”. In: *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. 2020, pp. 533–548.

[248] Qizhe Xie et al. “Self-training with noisy student improves imagenet classification”. In: *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 2020, pp. 10687–10698.

[249] Saining Xie et al. "Aggregated residual transformations for deep neural networks". In: *IEEE CVPR*. 2017.

[250] Xin Xu et al. "Characterization and prediction of performance interference on mediated passthrough GPUs for interference-aware scheduler". In: *11th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 19)*. 2019.

[251] Gingfung Yeung et al. "Horus: Interference-Aware and Prediction-Based Scheduling in Deep Learning Systems". In: *IEEE Transactions on Parallel and Distributed Systems* 33.1 (2022), pp. 88–100.

[252] Jason Yim et al. *Using AI to predict retinal disease progression*. 2021. URL: https://deepmind.com/blog/article/Using_ai_to_predict_retinal_disease_progression (visited on 10/25/2021).

[253] Yang You et al. "Large batch optimization for deep learning: Training bert in 76 minutes". In: *arXiv preprint arXiv:1904.00962* (2019).

[254] Fisher Yu and Vladlen Koltun. "Multi-scale context aggregation by dilated convolutions". In: *arXiv preprint arXiv:1511.07122* (2015).

[255] Matei Zaharia et al. "Spark: Cluster computing with working sets." In: *HotCloud* 10.10-10 (2010), p. 95.

[256] *zero deepspeed: new system optimizations enable training models with over 100 billion parameters*. 2020. URL: https://www.microsoft.com/en-us/research/blog/zero-deepspeed-new-system-optimizations-enable-training-models-with-over-100-billion-parameters/.

[257] Chaojie Zhang et al. "Flex: High-Availability Datacenters With Zero Reserved Power". In: *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*. 2021, pp. 319–332. DOI: 10.1109/ISCA52012.2021.00033.

[258] Jinny X Zhang et al. "A deep learning model for predicting next-generation sequencing depth from DNA sequence". In: *Nature communications* 12.1 (2021), pp. 1–10.

[259] Li Lyna Zhang et al. "nn-Meter: Towards Accurate Latency Prediction of Deep-Learning Model Inference on Diverse Edge Devices". In: *The 19th ACM International Conference on Mobile Systems, Applications, and Services (MobiSys 2021).* June 2021.

[260] Ru Zhang et al. "An Empirical Study on Program Failures of Deep Learning Jobs". In: *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE).* 2020, pp. 1159–1170.

[261] Shenglin Zhang et al. "Prefix: Switch failure prediction in datacenter networks". In: *Proceedings of the ACM on Measurement and Analysis of Computing Systems* 2.1 (2018), pp. 1–29.

[262] Sixin Zhang, Anna Choromanska, and Yann LeCun. "Deep learning with elastic averaging SGD". In: *arXiv preprint arXiv:1412.6651* (2014).

[263] Xiao Zhang et al. "CPI2: CPU performance isolation for shared compute clusters". In: *Proceedings of the 8th ACM European Conference on Computer Systems.* 2013, pp. 379–391.

[264] Yanqi Zhang et al. "Sinan: ML-based and QoS-aware resource management for cloud microservices". In: *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems.* 2021, pp. 167–181.

[265] Yunqi Zhang et al. "Smite: Precise QoS prediction on real-system smt processors to improve utilization in warehouse scale computers". In: *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture.* IEEE. 2014, pp. 406–418.

[266] Hanyu Zhao et al. "Hived: sharing a GPU cluster for deep learning with guarantees". In: *14th USENIX symposium on operating systems design and implementation (OSDI 20)*. 2020, pp. 515–532.

[267] Lianmin Zheng et al. "Ansor: Generating high-performance tensor programs for deep learning". In: *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. 2020, pp. 863–879.

[268] Yiwen Zhu et al. "KEA: Tuning an Exabyte-Scale Data Infrastructure". In: *Proceedings of the 2021 International Conference on Management of Data*. 2021.

[269] Barret Zoph and Quoc V Le. "Neural architecture search with reinforcement learning". In: *arXiv preprint arXiv:1611.01578* (2016).

Appendix A

Application Controller Listings

A.1 Cluster State Implementation

The cluster state is constructed by watching the infrastructure updates and building an internal cache as mentioned in Section 4.3.1.

Listing 2 Machine node struct definition.

```
// NodeInfo is an instance of Machine Node with resources info
type NodeInfo struct {
        name string
        uid  types.UID
        node *v1.Node
        nonGPUPods []*v1.Pod
        numGpus     int
        devices     map[int]*accelerator.DeviceInfo
        mu          sync.RWMutex

        requestedResource *Resource
        nonzeroRequest    *Resource
        allocatableResource *Resource

        // double linked list.
        next *NodeInfo
        prev *NodeInfo
}
```

The cache contains individual node information such as the allocatable and requested resources (i.e., CPU, Memory, and GPUs). To achieve efficient querying of GPU hardware consumption-aware scheduling, We add any additional accelerator information within the node cache as shown in Listing 2 and Listing 3.

Listing 3 Function to build up the machines and GPUs cache in Go.

```
// BuildNodeInfo returns pointer to node info
func (cache *SchedulerCache) BuildNodeInfo(name string, node *v1.Node) (*NodeInfo, error) {
        _, ok := node.Labels["gpushare"]
        if !ok {
                return nil, fmt.Errorf("not responsisble for this node, %s", node.Name)
        }
        // Add a new node, since we just got one.
        klog.V(8).Infof("Going to build new node info: %v \n", name)
        var internalIPInCluster string
        addresses := node.Status.Addresses
        for i := 0; i < len(node.Status.Addresses); i++ {
                if addresses[i].Type == v1.NodeInternalIP {
                        internalIPInCluster = addresses[i].Address
                }
        }
        devices, err := cache.acceleratorManager.GetDevicesByNode(name, internalIPInCluster)
        if err != nil {
                klog.Errorf("Getting device for Node failed: %v \n", err)
                return nil, err
        }

        n, err := NewNodeInfo(node, devices)

        n.allocatableResource = NewResource(node.Status.Allocatable)

        // get the the pods currently residing in this node
        podList, err := cache.podLister.Pods(utils.ExperimentNameSpace).List(labels.Everything())
        if err != nil {
                klog.Errorf("getting pods from cache failed: %v \n", err)
                return nil, err
        }

        if err != nil {
                klog.Errorf("Building new Node info failed: %v \n", err)
                return nil, err
        }
        klog.Infof("Finish built new node info: %v \n", name)
        return n, nil
}
```

A.2 Migration Implementation

To cope with GPU OOM, we add a mechanism to update the GPU memory consumption for a DL training job who recently failed. The DL training job expected memory will be set to the currently used GPU memory to ensure it will obtain adequate amount of memory. This is shown in Listing 4.

Listing 4 GPU memory update due to OOM.

```go
// SetUsedMemoryToJob set currently used memory to the job
func (c *MyController) SetUsedMemoryToJob(nodename string, gpus []string, jobCopy *jobs.JobRequest) {
	nodeInfo, err := c.schedulerCache.GetNodeInfo(nodename)

	if err != nil {
		klog.Errorf("%v", err)
	}
	devices := nodeInfo.DevicesCopy()

	for _, g := range gpus {
		gpu, err := strconv.Atoi(g)
		if err != nil {
			continue
		}
		dev, ok := devices[gpu]
		if !ok {
			continue
		}
		currentUsedMem := float64(dev.UsedMem())
		klog.Infof("set fail request %v to use this much memory %v", jobCopy.JobName(), currentUsedMem)
		// so then the job will keep track of the largest memory we need for this job across all the pods
		jobCopy.SetExpectedMemory(currentUsedMem)
	}
}
```

To allow migration to happen, we add a co-routine within the application controller to periodically scan the cluster and migrate DL training jobs from overloaded devices to underutilised devices. This is shown in Listing 5.

Listing 5 Horus migration thread written in Go.

```go
// CheckUnderutilisedAccelerator discover where there is both underutilised and overutilised device
func (c *MyController) migrationCheck() {
	if c.scheduleQueue.NumWaitingJobs() < 1 {
		return
	}
	// we check through our cache.
	klog.Infof("Start check underutilized")
	defer klog.Infof("Finish check underutilized")
	c.schedulerCache.MU().RLock()
	nodes := c.schedulerCache.GetNodeInfos()
	c.schedulerCache.MU().RUnlock()
	if !c.HaveUnderutilisedDevice(nodes) {
		return
	}
	dev := c.FindOverloadedDevice(nodes)
	if dev == nil {
		return
	}
	c.EvictPodOnDev(dev)
	c.metricCollector.AddEviction()
}
```

A.3 GPU Utilisation Predictor Implementation

As mentioned in Section 4.4, our GPU utilisation predictor implementation leverage file system to load the trained model into memory for prediction. This is to allow efficient periodic update due to re-training. Our implementation assumes the model is packed into a tar file as shown in Listing 6. Finally, our prediction extract the features passed from the profiling method mentioned in Section 3.3.1 and feed to the predictor as shown in the defined method *predict_from_job*.

Listing 6 GPU utilisation Predictor class in Python.

```python
class GPUUtilsPredictor(object):
    def __init__(self, logger, file_path, pred_dir=None, model_name="xgb_model.sav"):
        self.file_path = file_path
        self.logger = logger
        if pred_dir is None:
            self.pred_dir = os.path.dirname(self.file_path)
        else:
            self.pred_dir = pred_dir

        if self.file_path.endswith("tar.gz") and tarfile.is_tarfile(self.file_path):
            t = tarfile.open(name=self.file_path, mode="r:gz")
            t.extractall(path=self.pred_dir)
            t.close()

        if not os.path.exists(os.path.join(self.pred_dir, model_name)):
            self.logger.warning("No model %s exist for prediction" % (model_name))
            self.model = None
        else:
            self.logger.info("Model %s found" % (model_name))
            self.model = joblib.load(os.path.join(self.pred_dir, model_name))
            self.logger.info(self.model)

    # predict GPU utilisation from features extracted from profiling.
    def predict_from_job(self, job_request):
        features = job_request.dnn_features
        features_arr = np.array(
            [features.batch_size, features.flops, features.memory_activations,
             features.memory_parameters, features.num_reshape, features.num_relu,
             features.num_pad, features.num_maxpool, features.num_gather,
             features.num_slice, features.num_concat, features.num_add,
             features.num_squeeze, features.num_gru, features.num_transpose,
             features.num_reducemean, features.num_conv, features.num_constant,
             features.num_batchnorm, features.num_globalavgpool, features.num_flatten,
             features.num_gemm, features.num_lstm, features.num_averagepool,
             features.num_split]).astype(np.float)
        return self.model.predict([features_arr])[0]
```

Appendix B

DNN Profiling Method Implementation

Listing 7 DNN model profiling method.

```
class ModelStats(object):
    # ...
    def count(self):
    for n in self.model.graph.node:
      hook_fn = hook.get(n.op_type, None)
      assert hook_fn != None, f"Unsupported operation from op: {n}"
      _inputs = []
      _outputs = []
      _attributes = {}
      for n_inp in n.input:
        io = self.get_io_identifier(n_inp, True)
        _inputs.append(io)
      for n_out in n.output:
        io = self.get_io_identifier(n_out, False)
        _outputs.append(io)
      for a in n.attribute:
        if a.type == AttributeProto.TENSOR:
          _attributes["t"] = a
        else:
          _attributes[a.name] = helper.get_attribute_value(a)
      # the _outputs, will get updated!
      footprint = hook[n.op_type](_inputs, _outputs, _attributes)
      self.add_footprint(n.op_type, footprint)
```

Appendix C

Experiment Configuration

Variable	Description	Value
cAdvisor log period	How frequently to collect metrics from the host machine	100ms
Prometheus scrape period	How frequently to scrape target metrics	250ms
Backfill buffer	The buffer size for backfilling	15
Number of queues	Number of queues to sort jobs into	1
Underutilisation threshold	Threshold to indicate whether a GPU is underutilised	40%
Accelerator synchronisation period	How frequently the application controller to update its cache for the accelerators	350ms

Table C.1: Empirical study experiment configuration.

Appendix D

Additional DL Models Configuration

D.1 ParaDNN Implementation

Listing 8 Feed Forward Neural Network implementation.

```python
class FC(nn.Module):
  def __init__(self, batch, layer, hidden, input_size, output_size):
    super(FC, self).__init__()
    batch_mult = 2 ** batch
    layer_mult = 2 ** layer
    hidden_mult = 2 ** hidden
    # batch size is overridable
    self.batch_size = min(64 * batch_mult, 16384)
    self.num_layers = min(4 * layer_mult, 128)
    self.nodes = min(32 * hidden_mult, 8192)
    self.input_size = min(2000 + ( 2000 * input_size), 8000)
    self.input = nn.Linear(self.input_size, self.nodes)
    self.output_size = min(200 + ( 200 * output_size), 1000)
    self.output = nn.Linear(self.nodes, self.output_size)
    self.linears = nn.ModuleList()
    for i in range(self.num_layers):
      self.linears.append(nn.Linear(self.nodes, self.nodes))

  def forward(self, x):
    x = self.input(x)
    for m in self.linears:
      x = m(x)
    return self.output(x)
```

Listing 8 and Listing 9 shows our implementation of ParaDNN benchmark models.

Listing 9 RNN implementation.

```
class RNN(nn.Module):
  def __init__(self, batch, layer, embed, length, vocab, type="LSTM"):
    super(RNN, self).__init__()
    self.type=type
    # batch size is overridable
    self.batch_size = min(16 * (4 ** batch), 1024)
    self.num_layers = min(1 + (4 * layer), 13)
    self.embed_size = min(100 + (400 * embed), 900)
    self.length = min(10 + (40 * length), 90)
    self.vocab = min(2 * (4 ** vocab), 1024)
    self.output = nn.Linear(self.embed_size*self.length, 1)

    self.embedding = nn.Embedding(self.vocab, self.embed_size)
    if self.type == "LSTM":
      self.rnn = nn.LSTM(self.embed_size, self.embed_size, self.num_layers, batch_first=True)
    else:
      self.rnn = nn.GRU(self.embed_size, self.embed_size, self.num_layers, batch_first=True)

  def forward(self, x, hidden_states=None):
    x = self.embedding(x)
    if hidden_states is not None:
      o, _ = self.rnn(x, hidden_states)
    else:
      o, _ = self.rnn(x)
    return self.output(o.reshape((self.batch_size, -1)))

  def init_hidden(self, batches, device=None):
    weight = next(self.parameters()).data
    h = Variable(weight.new(self.num_layers, self.batch_size, self.embed_size).zero_())
    if device is not None:
      h.to(device)

    if self.type == "LSTM":
      c = Variable(weight.new(self.num_layers, self.batch_size, self.embed_size).zero_())
      if device is not None:
        c.to(device)
      return (h, c)
    else:
      return (h)
```

Finally, Table D.1 and Table D.2 show the parameters we passed to generate these ParaDNN models.

Model	Num. Layers	Hidden Size	First Hidden Layer	Output Size
FC_0	4	32	2000	200
FC_1	8	32	2000	200
FC_2	8	64	2000	200
FC_3	8	64	4000	200
FC_4	8	64	4000	400
FC_5	16	32	2000	200
FC_6	16	128	2000	200
FC_7	16	128	6000	200
FC_8	16	128	6000	600
FC_9	32	64	2000	200
FC_{10}	32	256	2000	200
FC_{11}	32	256	8000	200
FC_{12}	32	256	8000	800
FC_{13}	64	64	4000	200
FC_{14}	64	512	4000	200
FC_{15}	64	512	8000	200
FC_{16}	64	512	8000	1000
FC_{17}	128	64	4000	400
FC_{18}	128	64	4000	400
FC_{19}	128	1024	8000	400
FC_{20}	128	1024	8000	1000
FC_{21}	128	32	2000	200
FC_{22}	128	2048	2000	200
FC_{23}	128	2048	8000	200
FC_{24}	128	2048	8000	1000
FC_{25}	128	128	2000	200

Table D.1: FC models configuration

Model	Num. Layers	Embedding Size	Sentence Length	Num. Vocabs
GRU_0	1	100	10	2
GRU_1	5	100	10	2
GRU_2	5	500	10	2
GRU_3	5	500	50	2
GRU_4	5	500	50	8
GRU_5	9	100	10	2
GRU_6	9	900	10	2
$LSTM_0$	1	100	10	2
$LSTM_1$	5	100	10	2
$LSTM_2$	5	500	10	2
$LSTM_3$	5	500	50	2
$LSTM_4$	5	500	50	8
$LSTM_5$	9	100	10	2
$LSTM_6$	9	900	10	2

Table D.2: RNN models configuration

D.2 MobileNetV2 Large Configuration

Table D.3 shows our configuration for a larger version of MobileNetV2.

t	c	n	s
1	16	1	1
6	32	2	2
6	64	3	2
6	96	4	2
6	128	4	1
6	256	2	2
6	512	2	1

Table D.3: MobileNetV2 Large inverted residual settings.