Co-location of Deep Learning Jobs in GPU Clusters

by

Iman Tabrizian

A thesis submitted in conformity with the requirements
for the degree of Master of Applied Science
Graduate Department of Electrical and Computer Engineering
University of Toronto

Abstract

Co-location of Deep Learning Jobs in GPU Clusters

Iman Tabrizian

Master of Applied Science

Graduate Department of Electrical and Computer Engineering

University of Toronto

2020

Deep learning training jobs now constitute a large portion of the jobs in the GPU clusters. Following the success of deep learning in various domains such as natural language processing, image classification, and object detection GPUs have become the new member of the computing clusters. Due to various reasons, GPUs are highly underutilized in the production GPU clusters. In this thesis, we design a scheduler that uses co-location to improve the GPU utilization in these clusters. Using in-depth profiling of DL jobs, we provide metrics that guide us on the compatibility of different DL jobs. Using these profiling data we are able to achieve almost 2X speedup in the makespan when using co-location compared to the first-in-first-out baseline.

Acknowledgements

I would like to thank my parents and my dear brother for their amazing support. Without their support, I would have not been able to survive this journey. I want to sincerely thank my advisor Professor Alberto Leon-Garcia for giving me the chance to work independently. His insightful comments have greatly impacted this project. I want to also thank the Network Architecture Lab members for insightful discussions and feedback on this project. Also, I would like to thank Vladi for his great administrative support.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Machine Learning has transformed the world. Nowadays, there are various machine learning models used in production systems. Models that provide image classification, movie recommendations, and even driving. These advancements have been made possible thanks to the specialized hardwares such as GPUs, TPUs[22], and FPGAs. The process of training these models is a trial-and-error approach combined with intuition. It requires tuning a large number of hyperparameters to achieve a desired accuracy. To achieve the best accuracy in the shortest amount of time, usually many similar jobs are dispatched that only change a single hyperparameter. Research institutes and companies have access to large compute clusters. These compute clusters require a scheduler to map the jobs to the resources. The users of the jobs need to specify the resource requirements of the jobs too. The scheduler will take in these specifications and map it to the the available resources. GPUs with their unique characteristics add a new dimension to this problem. The main problem with the GPUs is that they were not originally designed for multi-tasking. Unlike conventional resources like CPUs and main memory, that have hardware support for resource sharing, GPUs were originally designed for single process use. The assumption was that the application is able to effectively use all the resources on a GPU and adding another application to the GPU will hinder the performance. However, starting from the Volta[8] architecture GPUs now include hardware support for execution of multiple processes together.

In this thesis, we explore the effect of co-locating multiple deep learning training jobs together. We devise metrics that guide the scheduler on which workloads can benefit from co-location and which workloads should not be placed together. Additionally, we study the reasons behind the incompatibility of the workloads and how we can predict the incompatible jobs.

This thesis is organized as follows. Chapter 2 provides a background on how GPUs work and how they can be programmed. It discusses how various tasks can be executed at the same time on the GPU. We also discuss and compare this work to the related work in this area.

Chapter 3 discusses the motivation for co-location of the DL jobs. We discuss the unique characteristics of DL jobs and GPU clusters that make co-location a good choice.

Chapter 4 provides in-depth profiling information. This profiling information helps us understand why co-location improves the performance in some cases while in other cases it may hinder the performance.

Chapter 5 discusses the design and implementation of the scheduler that uses co-location. We provide an algorithm that uses the profiling information from chapter 4 to identify co-locations that will lead to speedup.

Chapter 6 this chapter provides end-to-end evaluation of the co-location algorithm. We study the effect of co-location in two different scenarios.

Chapter 7 concludes the paper and discusses the potential future research directions that can be built on this work.

Chapter 2

Background and Related Work

In this chapter we will discuss some background information related to this work. We will discuss the basics of generic programming on GPUs and how scheduling multiple processes work on GPUs. We also discuss the related work to this project.

2.1 Graphical Processing Units (GPUs)

GPUs were among the first accelerators to be introduced as CPUs co-processors. GPUs are throughput optimized in contrast to the CPUs which are latency optimized. They consist of many inexpensive cores that can perform large number of operations in parallel. They also have a large memory bandwidth that helps them bring data into these cores efficiently.

Figure 2.1 shows a simple illustration of the GPU architecture. GPUs consist of several *Streaming Multiprocessors (SMs)*. SMs are the processing units in GPUs. Each of the SMs contains various cores designed for operations on different data types. For example, V100 GPU contains 84 SMs where each of them has 64 FP32 cores, 64 INT32 cores, 32 FP64 cores, and 8 tensor cores[8]. In fig. 2.1, *CC* refers to the CUDA cores which consists of all the cores present in each SM except the tensor cores. Tensor Cores are abbreviated using *TC*. They provide accelerated performance for the reduced precision

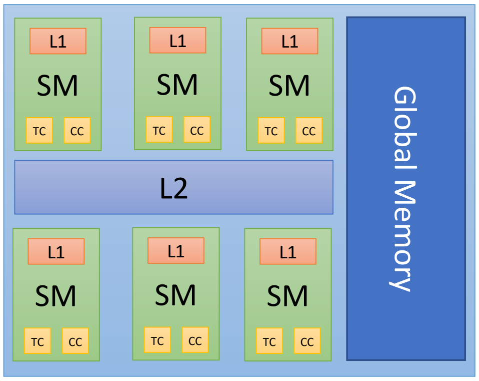


Figure 2.1: GPU Architecture

operations which are present in the Deep Learning workloads. They were introduced in the Volta[8] microarchitecture in 2017.

2.1.1 Memory Hierarchy

As shown in fig. 2.1, GPUs have a memory hierarchy similar to the CPUs. The main difference is including a shared memory that can be explicitly managed by the users program.

GPU memory hierarchy consists of:

- **Registers**: registers are allocated to each individual thread.
- **L1 Cache**: L1 cache is shared among all the thread blocks in a SM.
- **Shared Memory**: Shared memory is similar to L1 cache except that it is explicitly managed by the user.
- **L2 Cache**: L2 cache is a large cached memory that is shared among all the SMs.
- **Global DRAM Memory**: It is the slowest memory access compared to all the other memory levels described above. This memory is very large. Usually all the data required for a computation is stored in this memory.

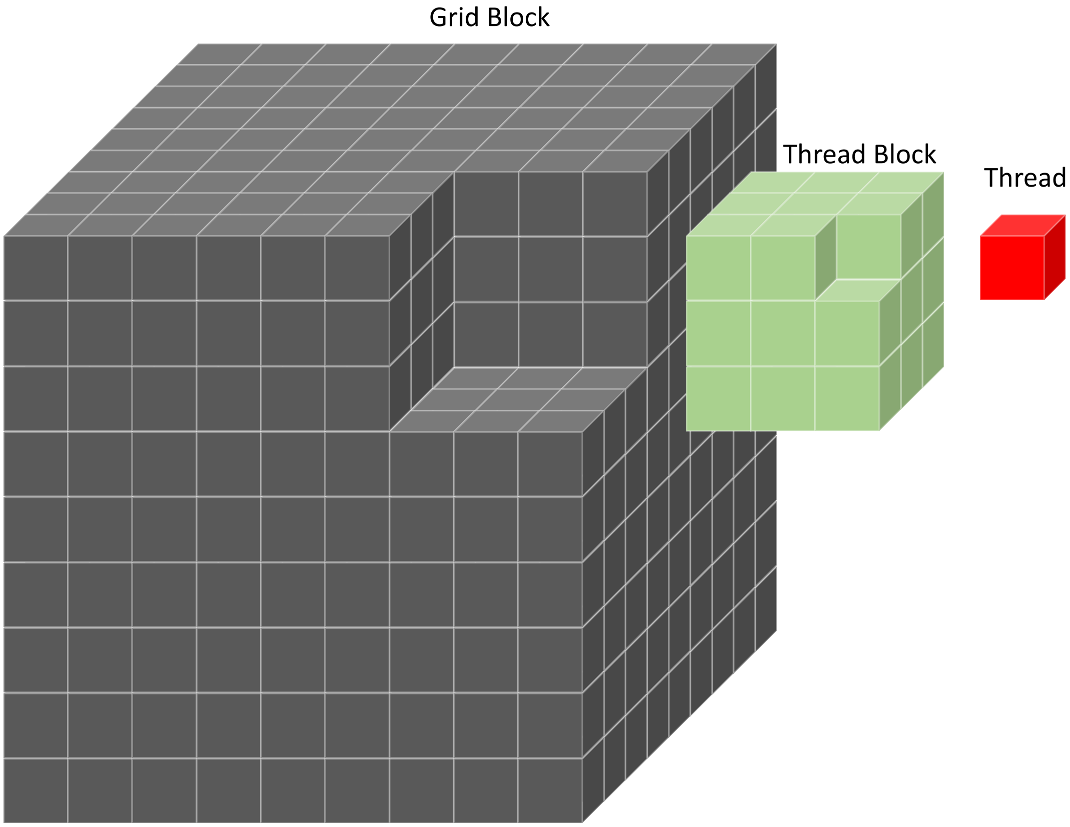


Figure 2.2: Grid Block vs Thread Block vs Thread

2.1.2 Programming Model

CUDA

CUDA is a set of extensions to C/C++ to enable easier application development in GPUs. CUDA also introduces a set of language abstractions that make it easier to think about GPU programs. GPU accelerated programs use many threads to perform the computation. CUDA groups the threads into *thread blocks* and *grid blocks*. A *thread block* is a group of threads which are guaranteed to be running on the same SM. A *grid block* is a group of thread blocks which contain all the processing necessary for the computation of a given *kernel*. A *kernel* is a function that runs on a GPU. Both thread blocks and grid blocks can be represented using three dimensions. Figure 2.2 shows the difference between a thread block, grid block, and a thread.

Thread Block Scheduling

There are various constraints that limit the scheduling of thread blocks into the SMs. Amount of the registers, shared memory, and number of threads inside a thread block are among the factors that limit the number of blocks that can be scheduled into a single SM. Since there is a limited amount of these resources available in each SM, the number of thread blocks will be limited to the available resources. Apart from that, different GPU architectures have hard limits on the number of thread blocks and threads that can be scheduled on a given SM. All these factors lead to a metric called *Theoretical Occupancy*[2] of a kernel. Theoretical Occupancy is a metric in percent which determines the percentage of active warps in comparison with the total warps that could be scheduled on a given GPU. There is another concept called *Achieved Occupancy*. Achieved Occupancy measures the scheduled number of warps when the kernel is actually running on the GPU. This can be different from the Theoretical Occupancy because a given thread in a warp might be stalled on a memory load and may not be ready to be scheduled. If there is not enough warps inflight ready to be scheduled, the achieved occupancy will be lower than the Theoretical Occupancy. Theoretical Occupancy serves as the upper bound for the Achieved Occupancy.

Life Cycle of a GPU accelerated Application

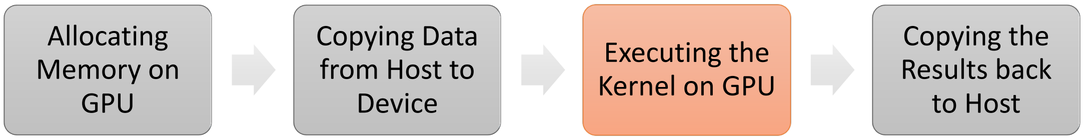

Allocating Memory on GPU
Copying Data from Host to Device
Executing the Kernel on GPU
Copying the Results back to Host

Figure 2.3: Lifecycle of a GPU accelerated application

Figure 2.3 shows lifecycle of a typical CUDA application. The application starts with allocating memory on the GPU. Then, it will copy data from the host memory into the GPU memory. After that, the kernel is executed. When the computation is complete, all the results are copied back into the host memory. All these operations must be executed

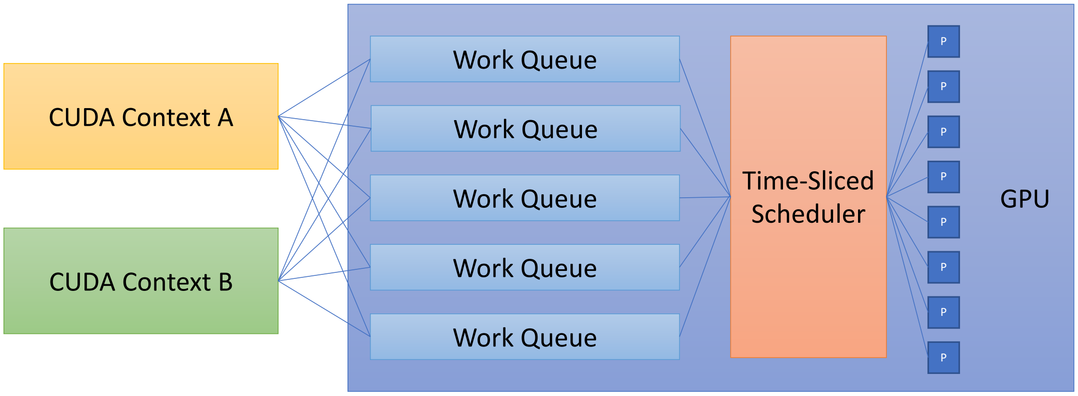


Figure 2.4: Interaction between CUDA contexts and Work Queues on a GPU

inside a *CUDA context*. Usually there is one CUDA context associated with each process. In the *Exclusive mode*, GPUs give exclusive access to a single CUDA context but in the *Default mode* work submitted from multiple CUDA contexts to the GPU will be scheduled in a time-sharing manner. MPS [3] allows multiple CUDA contexts to run applications on the GPU concurrently. This is explained in more details in section 2.1.3.

2.1.3 Concurrent Execution of Tasks on a GPU

CUDA Streams

CUDA Stream is a software construct containing a series of commands that must be executed in order. Work from different streams can be executed concurrently. Recent GPUs are capable of executing work concurrently from different CUDA streams belonging to the same CUDA context. Without Multi-Process Service(MPS) [3], it is not possible to run commands from another CUDA context unless the work from the current CUDA context has finished. A common use case for CUDA streams is overlapping computation and communication to speedup the Kernel execution.

Multi-Process Service (MPS)

MPS [3] is a mechanism that enables packing multiple processes together without having to time-share the GPU. MPS achieves this by using a client-server architecture. All the

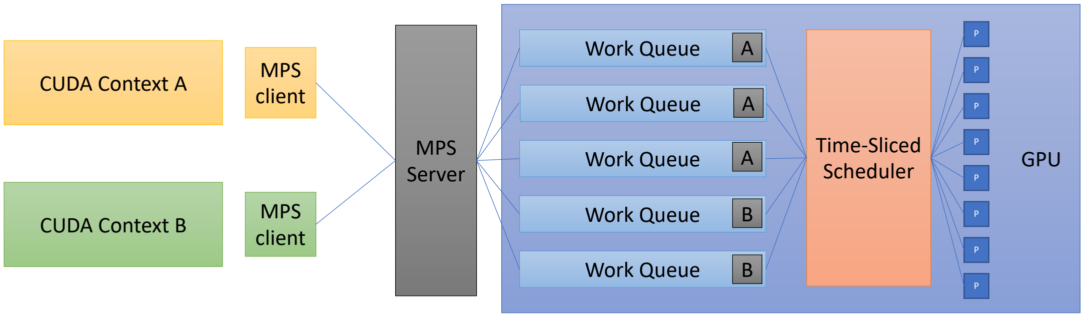


Figure 2.5: Interaction between CUDA contexts and Work Queues on a GPU

processes that want to run on the GPU are submitted to the the MPS server. MPS is useful when an individual job is not able to saturate all the GPU resources. Before Volta, MPS could not isolate the memory address of different CUDA contexts running on the same GPU. After Volta, MPS has improved the address space isolation along with improved performance through hardware support for MPS.

Figure 2.4 shows how CUDA contexts interact with the GPU to schedule work. GPUs have a hardware construct named *Work Queue*. Different CUDA contexts cannot have their work be executed simultaneously on the GPU. The GPU is time-shared between tasks coming from different CUDA contexts.

Figure 2.5 shows how CUDA contexts interact with the GPU when the MPS server is running on the GPU. MPS server acts as a middle-man that intercepts all the work that is being submitted to the GPU. MPS Server will then submit the work on behalf of the application to the GPU. The GPU now will schedule all the work from both of the CUDA contexts increasing the GPU utilization.

2.2 Deep Learning

Deep learning [27] is a machine learning paradigm that focuses on training of artificial neural networks with many layers. Artificial Neural Networks (ANNs) are function approximators that are proven to be universal [18]. It means that they can approximate any measurable function to any desired degree of accuracy. This feature combined with the

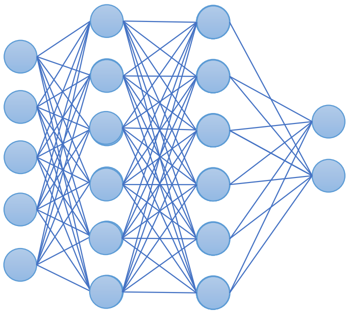


Figure 2.6: A Fully Connected Feed Forward Neural Network

advances in hardware enabled end-to-end learning algorithms that are able to generalize well in different domains of Natural Language Processing [11], Computer Vision [26], and Speech Recognition.

Figure 2.6 shows a simple illustration of a four-layer neural network. In this figure, there are four layers in total. The first layer contains the input data and the last layer is called the output layer. The intermediate layers in this network is called the hidden layers. Each of the nodes in these layers is called a neuron. Each neuron in every layer needs to use a function for activation. The ultimate goal of a neural network is to learn a set of weights that perform best on the test data set. The process of learning the optimal weights is referred to as *Training*. The neural network starts with random weights. During each iteration of the training a sample from the training data set is passed in the forward direction known as *forward pass*. In the output layer, neural network compares the calculated output with the expected output and calculates the difference between them using a metric known as *loss*. Loss is a scalar value, which will be propagated backward into the network and the weights are updated for each layer. The updates for each layer is calculated by computing the partial derivatives with respect to the previous layer. These updates along with a learning rate and an optimization algorithm will determine the weights for the next iteration.

2.2.1 GPUs and Deep Learning

GPUs were originally designed for graphic processing. Because of the enormous amount of vector computations that were required for graphics, GPUs were invented. Deep learning also involves a variety of different operations. The main operation is matrix by matrix multiplication which specifically GPUs are very good at. Matrix by matrix multiplication is a compute-bound operation which makes it the best fit for GPUs. Indeed, the peak compute performance of GPUs is calculated by measuring the matrix multiplication. Although deep learning techniques were introduced many decades ago, their feasibility remained questioned until recent years. The hardware advances enabled these computationally expensive workloads affordable and feasible in a timely manner.

2.3 Machine Learning Frameworks

Machine Learning frameworks where introduced to enable easier adaption of new deep learning techniques and help with democratization of research in this area. The mainstream frameworks are specifically designed to help with adoption of deep learning on specialized hardwares such GPUs or TPUs. These frameworks are usually implemented in C++ for better performance with APIs in the languages that the community prefers like Python. Designing a deep learning framework is a very challenging task. DL frameworks have to adapt very quickly with the latest versions of the accelerators. New accelerators are announced around every year. With multiple accelerators present in this field, making sure that you have the best performing algorithm on the new hardware is not easy. Also, there are new models being introduced on a daily basis. Making sure that all the state-of-the-art models produce the same result on this large spectrum of hardware of software versions is almost an impossible task. Because of this, mainstream frameworks are usually a result of collaboration between hardware vendors and big software companies.

2.3.1 Tensorflow

Tensorflow [10] is one of the first mainstream DL frameworks open sourced by Google. Tensorflow uses a *computation graph* to describe all the computation necessary to achieve a task. When using a computation graph, the user has to specify all the operations that it needs beforehand. After that, they can compile the computation graph and get the results. This model allows more optimizations to be performed during the compile time. However, this model may limit the users ability to debug the values when creating the model since the intermediate values are not available. To fix this problem, Tensorflow introduced an eager execution mode that uses an imperative model to make it easier to debug the models. This model is usually slower as it doesn't have the ability to have an omniscience view of the whole computation to perform the optimizations.

2.3.2 PyTorch

PyTorch [36] introduced in 2016 by Facebook with the focus on developer productivity. PyTorch uses autograd to automatically store the necessary information for calculating the gradient of arbitrary PyTorch code. Also, instead of using a computation graph, PyTorch uses an imperative programming model that increases the developer productivity. Some research frameworks [21] have also taken a hybrid approach to maintain both a high developer productivity and an optimized, fast execution.

2.3.3 MXNet

MXNet [13] was introduced in 2015. MXNet is an Apache project supported by many universities and companies. MXNet tries to support both of the programming paradigms discussed in sections 2.3.1 and 2.3.2. MXNet also supports a *hybrid* approach similar to Janus [21].

2.4 Related Work

2.4.1 GPU Scheduling

In the area of GPU schedulers for deep learning workloads there are various related works to the work presented in this thesis. Optimus [37] is one of the earlier works in this area. The main goal of this work is which job should be given more workers so that the total job completion time for a set of jobs is minimized. It is assumed that the jobs use the Parameter Server [31] architecture. They learn the convergence curve for a given job to predict the number of required iterations until the completion. Using this information, they can measure how long the job is going to last. Then, they create a heuristic algorithm that increases the number of workers or parameter servers for a job that gains more speed up compared to the other jobs. This work is complementary to our work. We can augment their techniques with our co-location algorithms to better utilize the GPUs.

Tiresias [16] presented a more realistic scheduling algorithm. Tiresias is able to use the historical data of the jobs to minimize the job completion time. They do not adaptively change the number of workers for a given job. This is a more realistic approach since increasing the number of workers affects the accuracy and may require retuning all the hyperparameters. They do not consider the packing of multiple jobs on the same GPU.

$$w(t+1) = w(t) - \eta * \nabla Q(w(t)) \tag{2.1}$$

Gandiva [39] introduced a fast context switch mechanism to avoid starvation of the jobs in deep learning clusters. They observed that the GPU memory usage of a job is not constant during the training and is minimum between the iterations. Figure 2.7 shows how memory allocation is performed during the training. In this figure, we are assuming the training of a convolutional neural network [28]. The weights associated with each layer is always present in the GPU. As the input traverses different layers of the neural

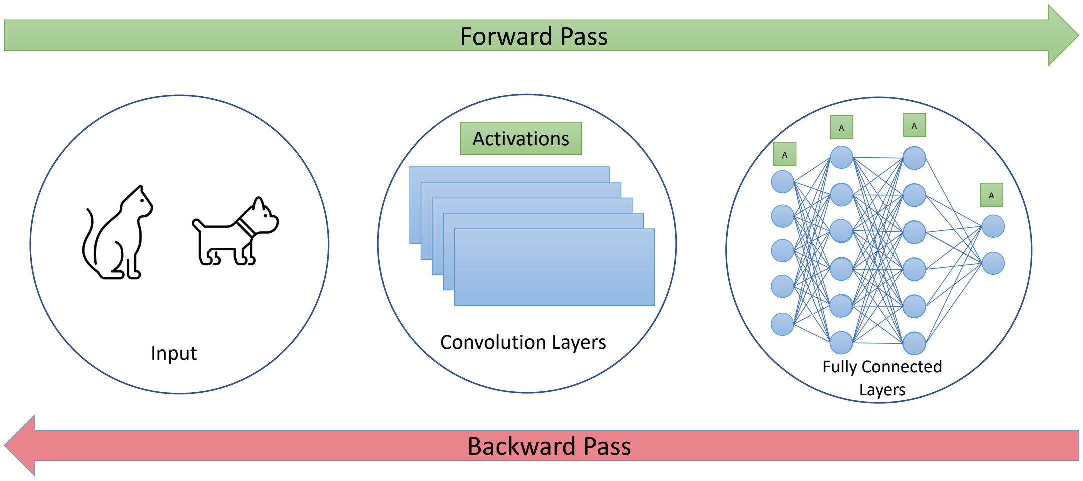


Figure 2.7: Memory Allocation in Forward and Backward Pass

network, it creates *activations*. Activations must be stored for each layer. The activations are required for calculating the gradients in eq. (2.1). These gradients will be calculated during the backward pass. In the backward pass, the activation values can be discarded and as a consequence the memory allocated for each of the layers may be freed. Gandiva leverages this pattern, and does not interrupt the job during the forward or backward passes to reduce the amount of data that needs to be copied during the checkpointing process. Gandiva also included a mechanism for ”packing” the jobs to reduce the queuing time and improve JCT. They employed random packing to find the matching job pairs. As mentioned in [34], this strategy is not sufficient for finding beneficial co-locations.

Chic [15] introduced the idea of adaptively changing the number of workers using a reinforcement learning algorithm. They showed that using reinforcement learning will lead to better results compared to Optimus [37]. However, they still didn't consider the packing of multiple jobs into a single GPU.

Gavel [34] was the first scheduler to design an scheduling mechanism that can use many different scheduling objectives. Gavel has support for hierarchical and multi-domain scheduling policies. Their modeling of the scheduling problem is able to take into account packing of multiple jobs into a single GPU if the appropriate profiling information is available. They also include a throughput estimator that is able to estimate the co-location throughput of unseen jobs.

2.4.2 GPU Sharing

Salus [40] provided a fast context switch mechanism by removing the need for checkpointing. Similar to Gandiva [39], they utilize DL job GPU memory allocation patterns to enable faster context switch. They divide the memory used by the deep learning frameworks into three types. Ephemeral memory which is allocated and deallocated at every iteration. Model memory which is the weights used by the model. The model memory is allocated and deallocated once during the training. And the memory required by the framework. This is usually smaller than the two other types of memory allocations. Using their library, multiple processes can run on the same GPU. They use ”GPU Lane” which is a logical component of Salus that tasks are dispatched to it. GPU lanes are implemented using CUDA streams. They also compared their implementation with MPS and achieved higher performance. However, their comparisons were based on the Pascal MPS which is not the MPS version that we used in this work. Volta MPS comes with greater hardware support and fixes some of the errors that was mentioned in the paper.

2.4.3 Jobs Interference

Although interference of co-located DL jobs in GPU clusters has not been well explored, there is a significant body of work on the interference effect of co-located jobs in CPU clusters. Quasar [14] employs PQ-reconstruction with Stochastic Gradient Descent, to fill the unspecified elements of a matrix. In this matrix, the rows are jobs and the columns are different platforms. Quasar first profiles a couple of jobs extensively on a number of platforms. For unseen jobs, it profiles the job on a limited set of platforms and then uses the PQ-reconstruction to predict the performance on unseen platforms. Gavel [34] used this technique in the ”Throughput Estimator” to predict the performance of co-location. They treated the co-located jobs as a new job and tried to fill in the matrix appropriately.

2.4.4 Performance Models for GPU Multitasking

HSM [45] provides a hybrid slowdown model that tries to predict the performance of the kernels using a hybrid model. The term ”hybrid” here refers to using both whitebox and blackbox models. For the compute-bound models increasing the number of SMs has a linear corelation with the speedup while that is not the case for memory-bound applications. The goal of this paper was to implement the QoS guarantees using these models. HSM was implemented in the GPUSim [23], a cycle simulator for GPUs.

[29] presented a performance model for CUDA streams for the compute-bound applications. The goal of this paper was to analyse the scheduling of the thread blocks on SMs when CUDA streams are enabled. They ultimately use these performance models to build a database that uses CUDA streams to increase the performance.

2.4.5 Scheduling CPUs in Big Data Clusters

The problems in CPU scheduling focus on different requirements for scheduling. The focus of the works in this area is mainly designing custom schedulers for Hadoop [1] and Spark [43] clusters. The tasks on these clusters are represented by a computation graph similar to DNNs. The difference with the DL jobs is that the data is usually dispersed into multiple nodes. The problem here is how we should decide on the scheduling of each of the subtasks on each node given the data distribution on the nodes in the cluster. [41] adds a small amount of delay between the scheduling of different jobs. The benefit of this approach is that this small delay will help with the availability of nodes where the data for a given task is present. Since each task takes a small amount of time, simply waiting a bit longer may help with the data locality of the tasks.

Since the latency requirements of scheduling are very important in these clusters, there have been some research trying to decentralize the scheduling decisions for smaller latency requirements. Sparrow [35] is one of the projects that focused on designing a decentralized scheduler for these clusters. They also provided a mathematical analysis of their scheduling mechanism.

There has also been recent papers that use Reinforcement Learning [33] techniques to make the scheduling decisions.

2.5 Summary

We provided an overview of the GPU computation and basics of running CUDA accelerated applications on GPUs. We also discussed how running multiple tasks on the same GPU work. Additionally, we discussed some of the mainstream ML frameworks and what are some of the programming paradigms in these frameworks.

Chapter 3

Motivation

Production GPU clusters suffer from low GPU utilization [20]. In a GPU cluster trace published by Microsoft more than 40% of the jobs only utilize 20% of the GPUs. In this chapter we discuss the potential reasons behind why the GPU utilization is low in production GPU clusters and how co-location can improve the utilization.

3.1 GPU Cluster Characteristics

3.1.1 Heterogeneity of GPU Clusters

GPU data centers are becoming more diverse with the introduction of new GPUs every year. The GPUs that are available in the data centers have many different capabilities. Some GPUs may have a couple of gigabytes of memory while others may have tens of gigabytes of memory. At the time of writing this thesis, A100 GPUs can support 80GBs [5] of memory. The amount of memory available on a GPU affects the batch size that the developer can choose. Larger batch sizes usually increase the utilization of a GPU. It may seem like an obvious choice to divide the GPU memory by batch size to fully utilize the GPU. However, not all the GPUs available in the cluster are A100 GPUs with 80 GBs of memory. In fact, these newer GPUs are in greater demand. Users usually

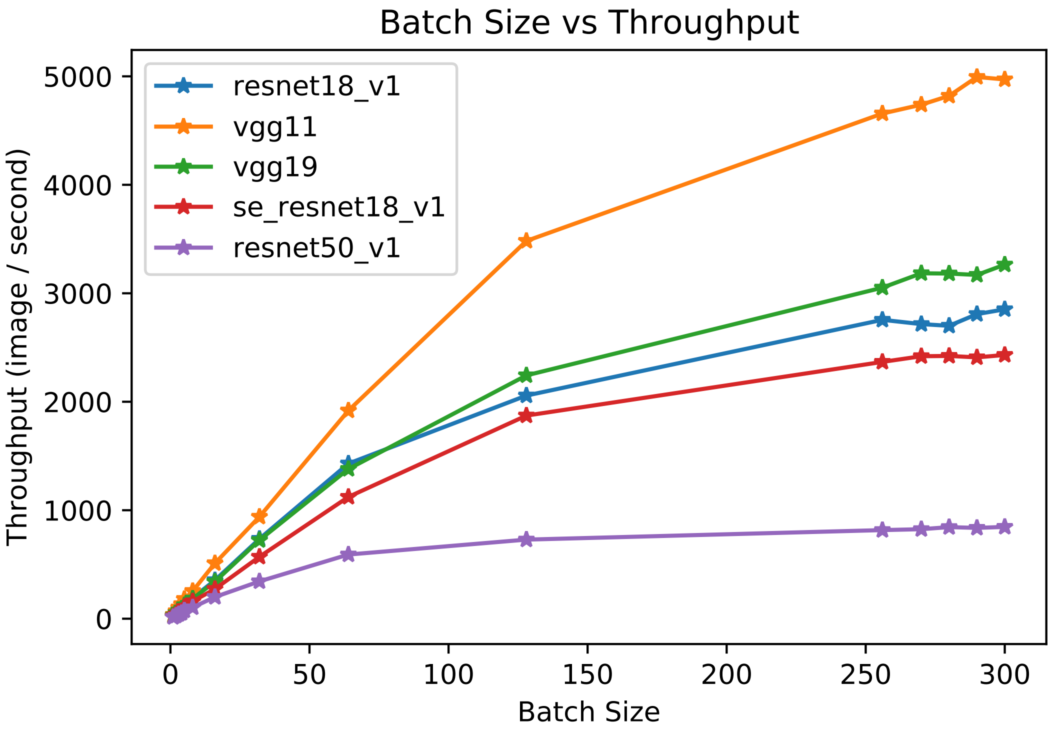


Figure 3.1: Batch Size vs Throughput

end up selecting a larger spectrum of GPUs in order to reduce the queuing time for their job to be scheduled. Since users cannot determine the effective batch size for their job when they are submitting the job to the cluster, they will end-up under utilizing the GPU and choosing smaller batch size.

Figure 3.1 shows the throughput of different image classification models. As we increase the batch size, the increase in throughput decreases. This point is different for various models. The takeaway from this figure is that apart from the GPU memory limit, the models see diminishing returns on increasing the batch size for training.

3.2 DL Job Characteristics

3.2.1 Statistical Constraints

Training a deep neural network involves launching many similar jobs with a single parameter change in order to find the best set of hyperparameters that achieves the best performance. One may argue that the batch size problem discussed in section 3.1.1 can be solved by querying the amount of GPU memory available by size of memory required for a single batch of data at the time the job is being scheduled. Since many different training jobs are launched simultaneously, the hyperparameters found for one batch size, is different from the hyperparameters for the other job. Because of this, batch size for all the jobs that training the same model should be the same.

Another issue is that some machine learning tasks are not able to use large batch sizes, and large batch sizes come with diminishing returns [24, 44]. This constraint further leaves the GPUs underutilized.

3.2.2 Monitoring, Logging, and Data Loading

Training a neural network does not only involve GPU computation. A common training task involves loading the data from the disk to the RAM, logging metrics such as accuracy to monitor the progress of training [42], and check pointing the trained model so that the progress is not lost in case of job preemption or power shutdown. Job preemption is implemented every hour in some academic clusters such as Vector Institute. Preemption helps avoiding starvation by giving everyone a fair share of the cluster. These tasks lead to leaving the GPU unutilized for many cycles. Using co-location, one job can use the GPU while the other job is busy monitoring metrics or check pointing. This way the GPU is better utilized.

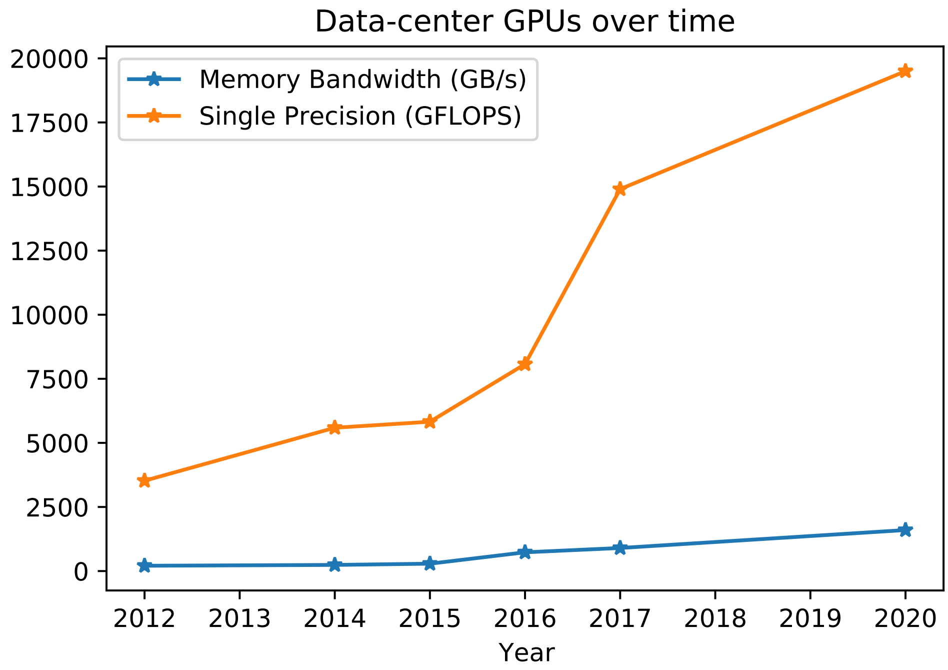


Figure 3.2: GPU Capabilities over time

3.3 GPU Memory and Compute Capabilities

GPU memory bandwidth and compute capabilities have increased by a tremendous amount over time. Figure 3.2 shows this trend in the Tesla GPU class. Tesla GPUs are NVIDIA's data center GPU class. The most recent data center GPU class as of writing this thesis is the A100 GPU with the ability to perform 312 TFLOPS half-precision operations. These increases in the compute and memory capabilities have made it harder for application developers to fully saturate the GPU resources. In this thesis, we use co-location to better utilize GPUs even if the individual jobs cannot fully utilize the GPU.

3.4 Summary

In this chapter we discussed why co-location of jobs can be beneficial when designing schedulers for deep learning jobs. We talked about the unique characteristics of deep learning jobs and GPU clusters and why they may not be able to fully utilize the resources in the cluster. These factors include the statistical constraints of the jobs, monitoring, logging or other CPU based tasks, and heterogeneity of GPU clusters. In the next chapter, we provide detailed profiling results to quantify the amount of speed up that can be gained using co-location.

Chapter 4

Training Job Co-location

4.1 Problem Formulation

Since the individual jobs may be slowed down when being co-located, we need to have an aggregate formulation for representing the speed up of multiple jobs on a single GPU and compare it with the case when they run individually. We propose a simple speedup factor that makes it easy to measure the amount of speedup, compared to the original case.

We are given n jobs that can be co-located all together on a single GPU. The time that it takes for a single iteration of job i to complete when running alone is t_i. $t'_{i,S}$ denotes the time for single iteration of job i to complete when being co-located with jobs that belong to $S \subset J$. J is the universal set containing all the jobs. We are interested in cases where the ratio of the sum of the jobs time running individually to the longest job time when being co-located is greater than 1, i.e. yields speedup. This is represented mathematically in eq. (4.1).

$$\delta = \frac{\sum_{j \in S} t_j}{\max_{j \in S} t'_{j,s}} \geq 1 \tag{4.1}$$

Figure 4.1 illustrates why this formulation is valid. As shown in fig. 4.1, an individual

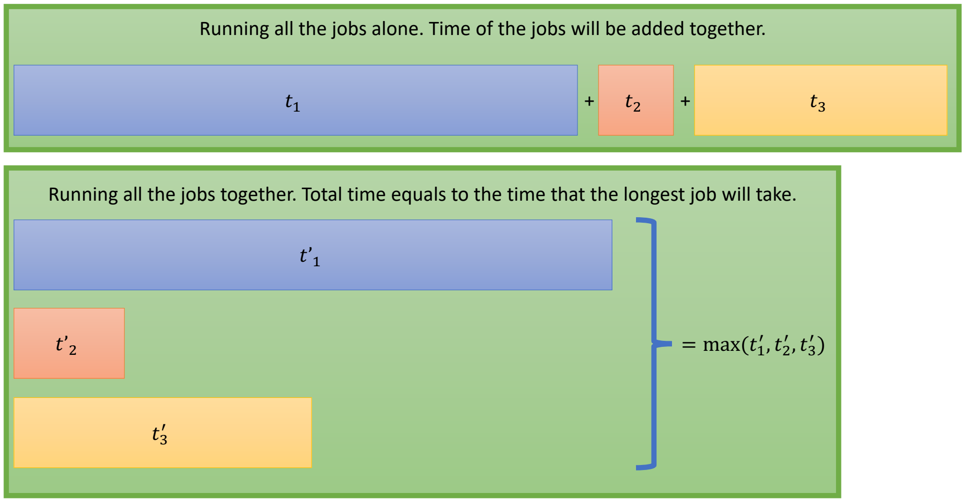

Running all the jobs alone. Time of the jobs will be added together.
t_1
+
t_2
+
t_3
Running all the jobs together. Total time equals to the time that the longest job will take.
t'_1
t'_2
t'_3
$= \max(t'_1, t'_2, t'_3)$

Figure 4.1: Co-location of Multiple Jobs

t'_i might be greater than t_i but because all the jobs have started together, as long as the longest job takes less than the sum of all the jobs, co-location is better than running alone. In fig. 4.1 we schedule 3 jobs and we want to find out whether co-location is suitable. We need to first calculate t_i values which is the time that each iteration of the job takes when running alone. Then, we need to calculate the values for t'_i which is the time that each iteration of the job takes when it is co-located with all the jobs in S. Next, we can use t_i and t'_i values and plug them in eq. (4.1). If the value for δ is larger than 1, these jobs will benefit from this co-location. It is worth noting that while every t'_i may be larger than t_i, it may be still worth the co-location as long as each iteration of the longest job does not take as much as the sum of all the individual iterations.

4.2 Profiling Training Jobs

We run a series of experiments to explore the value of δ in eq. (4.1). The models we used here are two variations of ResNet [17] models, two variations of VGG [38] models, and a more recent image classification architecture named SE-Net [19]. We run all the possible combinations of these models. These combinations include cases of running more than two jobs together. Since we are studying five models, the total number of models used is

Experiment Number	Models (Co-location of 2)	Models (Co-location of 3)
0	vgg19, resnet50	resnet18, se_resnet18, resnet50
1	se_resnet18, resnet50	resnet18, vgg19, resnet50
2	vgg11, vgg19	resnet18, vgg11, se_resnet18
3	resnet18, resnet50	resnet18, vgg11, vgg19
4	vgg11, resnet50	resnet18, vgg19, se_resnet18
5	resnet18, vgg11	vgg11, se_resnet18, resnet50
6	vgg19, se_resnet18	resnet18, vgg11, resnet50
7	vgg11, se_resnet18	vgg11, vgg19, se_resnet18
8	resnet18, se_resnet18	vgg11, vgg19, resnet50
9	resnet18, vgg19	vgg19, se_resnet18, resnet50

Table 4.1: Description of the experiment numbers

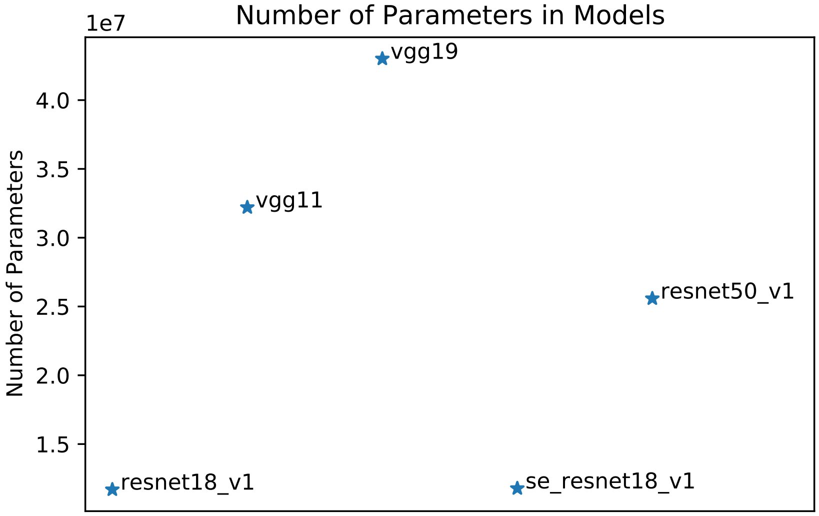


Figure 4.2: Number of Parameters in the Benchmark Models

equal to $\binom{5}{2} = 10$ when co-locating two jobs and $\binom{5}{3} = 10$ when co-locating three jobs.

Figure 4.2 shows the number of parameters that the benchmarked models need. The values presented in this figure are for the time that the model is using a batch size of one. Increasing the batch size will increase the number of parameters for each model. VGG models require the largest number of parameters and ResNet family require fewer parameters in comparison.

Figure 4.3 shows the value of δ when two jobs are placed together. The X axis shows the experiment number for all the possible combinations of the models described

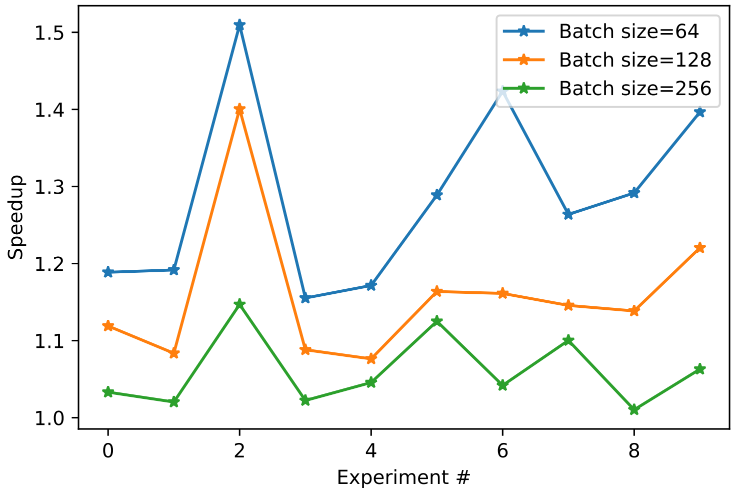


Figure 4.3: Co-locating two jobs together using various batch sizes. x-axis shows the experiment number described in table 4.1. MPS is not enabled in this figure.

in fig. 4.2. As expected, smaller batch sizes provide higher speedups compared to large batch sizes. These results are for a per iteration speedup of the models. We sampled 10 iterations of the training and calculated the average of these samples. We used the mean as the value for t' and t values in eq. (4.1). MPS is not enabled in any of the experiments shown in this figure. Later in this chapter we will present some other experiments explaining the cause for peaks and valleys seen in this figure.

Figure 4.4 shows the value of δ when two jobs are placed together when MPS is enabled. As expected, enabling MPS will lead to higher speedup compared to not enabling MPS. Gaining speed up when MPS is not enabled suggests that for certain batch sizes and certain models given the GPU that we used for benchmarking, there are time intervals that a GPU is not utilized at all and thus time-sharing leads to speedup.

Figures 4.5a to 4.5c show the results when the batch size is constant and we are running two or three jobs together. The following trends can be observed from these experiments:

- **Larger batch sizes lead to smaller speedup**. As we increase the batch size the largest speedup decreases.

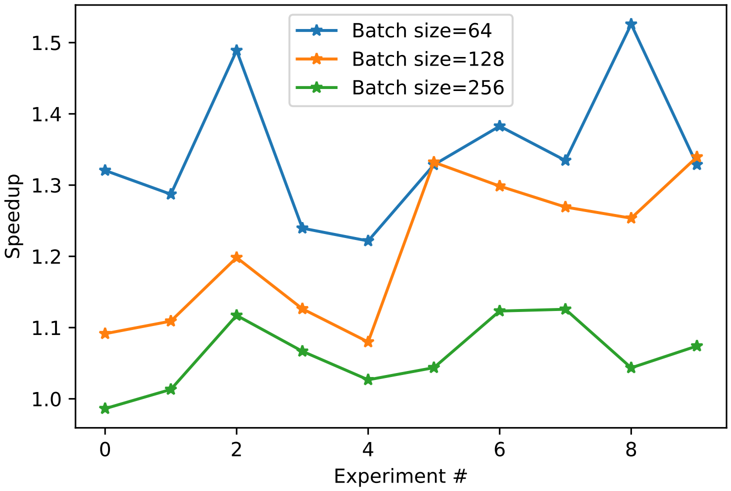
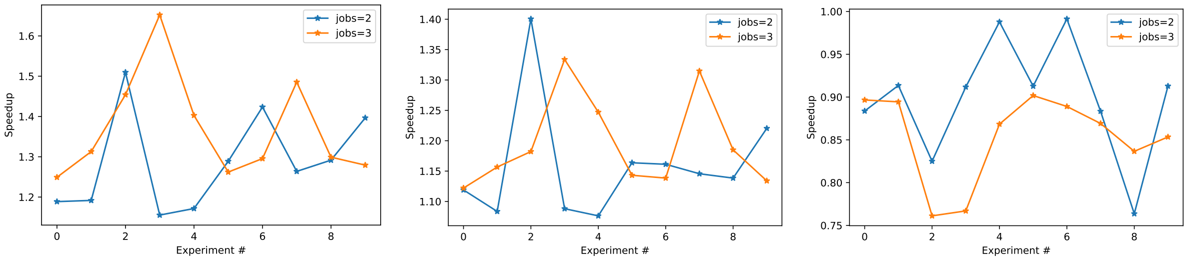


Figure 4.4: Co-locating two jobs together using various batch sizes. x-axis shows the experiment number described in table 4.1. MPS is enabled in this figure.

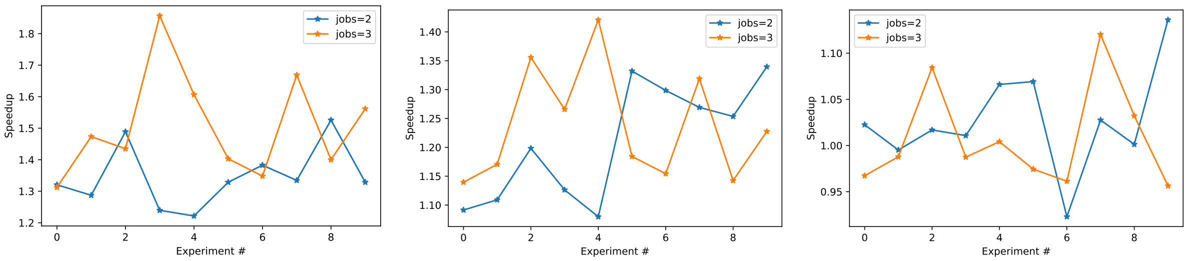


(a) Batch Size = 64

(b) Batch Size = 128

(c) Batch Size = 256

Figure 4.5: Co-location of two or three jobs together. x-axis shows the experiment number described in table 4.1. MPS is not enabled in these experiments.



(a) Batch Size = 64 (b) Batch Size = 128 (c) Batch Size = 256

Figure 4.6: Co-location of two or three jobs together. x-axis shows the experiment number described in table 4.1. MPS is enabled in these experiments.

- **More than two jobs leads to higher speedup when using small batch sizes.** As shown in fig. 4.5a, in most of the cases co-locating three jobs together gives higher speedup than co-locating two jobs together.

- **Number of parameters is not the source of slowdown.** If we use the information in table 4.1, we notice that the experiments that are slowest, are the ones that are being co-located with ResNet-50. In fig. 4.2, ResNet-50 is not the model that has the largest number of parameters in the models that we studied. Also, the model that has the best co-location behavior (i.e. leads to higher speedup) is the VGG11 model. The peaks occur when the job is being co-located with a VGG11 model. VGG11 neither has the fewest number of parameters or the largest number of parameters.

Figures 4.5a to 4.5c show the results when the batch size is constant and we are running two or three jobs together. In these experiments MPS is enabled. The same observations that we described for Figure 4.5 hold for these experiments too. In addition the speedup gained from co-location is larger when MPS is enabled. This is what we expected.

4.2.1 Kernel Analysis of the Jobs

We want to find the underlying reason for the different values of speedup when co-locating different models. As described previously, we noticed that number of parameters that each model uses is not a good heuristic for determining the compatibility between a set of jobs. Each deep learning training job is composed of many kernels that are used to perform computation for each stage of the DL training. As discussed in chapter 2, deep learning frameworks implement many of the operations required for DL training on the specialized hardwares such as GPUs. The user requests to calculate a backpropagation, and the DL framework will launch a series of kernels to perform that computation on the GPU. In the following subsections, we propose simple metrics that can attribute a number to the whole job and we will use these metrics to describe the speedup observed in the experiments when MPS is enabled. Since the kernels are either memory-bound or compute-bound, we will focus on the memory bandwidth utilization of the GPU and kernel occupancy for the models that we are studying. The kernel analysis is performed using NVIDIA Nsight Compute [6].

4.2.2 Memory Bandwidth Utilization

Assume that each kernel takes t_i time and utilizes p_i percent from the memory bandwidth of our V100 GPU. A histogram for p_i values is shown in fig. 4.7. Key takeaways from this figure are the following:

- ResNet-50 has more kernels in almost every utilization category. It is also the only kernel that has some kernels that are able to almost fully utilize the V100 bandwidth.
- VGG11 has fewer number of kernels in almost every utilization bucket.
- Kernels that utilize the memory bandwidth least constitute the largest number of kernels.

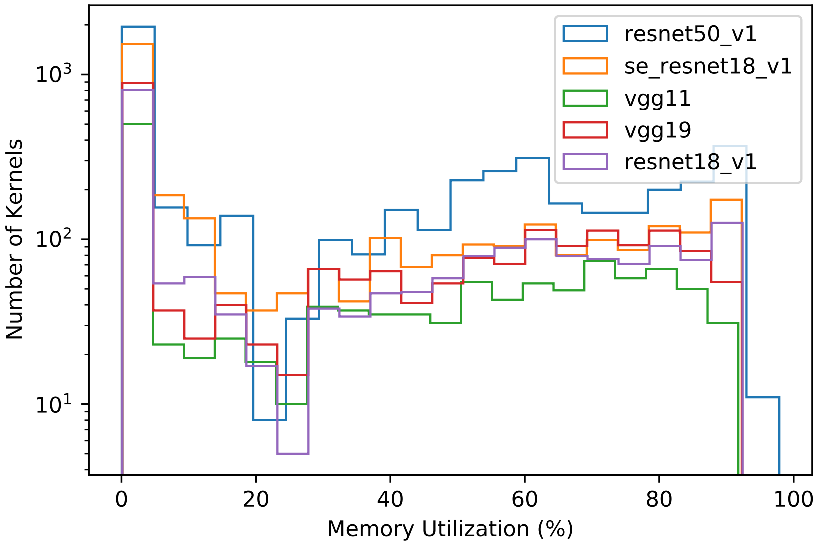


Figure 4.7: Histogram of the memory utilization of the kernels for models under study. The batch size is equal to 64. If a kernel is run multiple times, the results are not grouped together and is assumed as a separate kernel. The unit for x-axis is in percent and the y-axis is log based. This graph includes the results for two iterations of training. The kernels were analyzed individually without sharing the resources with any other job.

Using this profiling information, we aim to create a single number that represents the aggregate memory bandwidth utilization of each of these models.

$$M = \sum_{i=1}^{N} p_i t_i \tag{4.2}$$

Equation (4.2) shows the formula for calculating the aggregate memory bandwidth for a given job. This is a weighted sum of the memory bandwidth utilization. The weights are the duration of the individual kernels. If a kernel uses all the memory bandwidth but doesn't take a very long time, it shouldn't affect the speedup very much. Likewise if the kernel does not utilize the memory bandwidth significantly but takes a very long time it should not affect the speedup to a great extent either.

Figure 4.8 shows the value of M presented in eq. (4.2) for different models and different batch sizes. As we increase the batch size, the M value increases for all the jobs that we studied. ResNet-50 utilizes the most amount of memory bandwidth compared to other

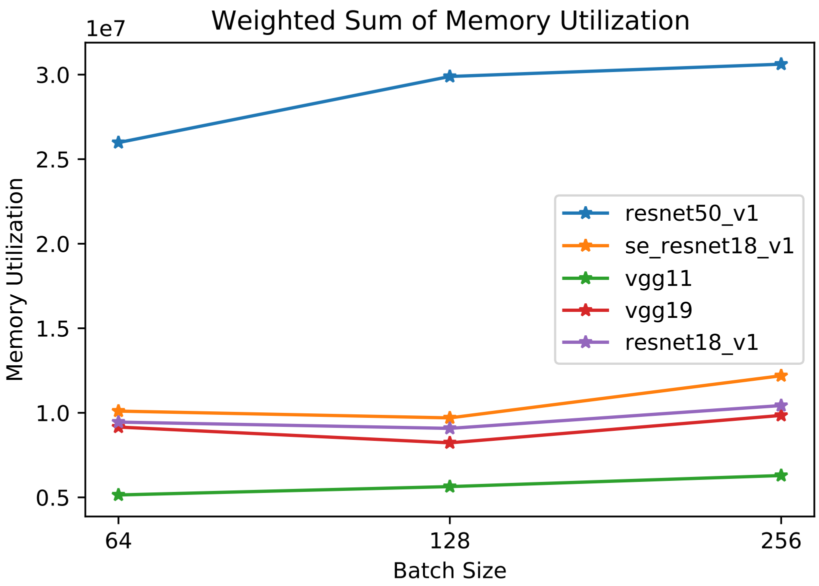


Figure 4.8: Weighted sum of the memory utilization of the models using various batch sizes

deep learning models. While ResNet-50 does not have the largest number of parameters, it is able to utilize the memory bandwidth more effectively compared to all the other models. We suspect that this may be due to the popularity of this model and as a result the kernels used for training this model are highly optimized.

4.2.3 Compute Utilization

Analogous metrics can be created for compute utilization. Assume that each kernel takes t_i time and q_i is the occupancy of the kernel on the V100 GPU. Histogram for q_i values is shown in fig. 4.9. Key takeaways from this figure are the following:

- ResNet-50 has a more kernels in almost every occupancy bucket.

- VGG11 has fewer number of kernels in almost every utilization bucket.

- Kernels that has the lowest occupancy constitute the majority of the observed kernels.

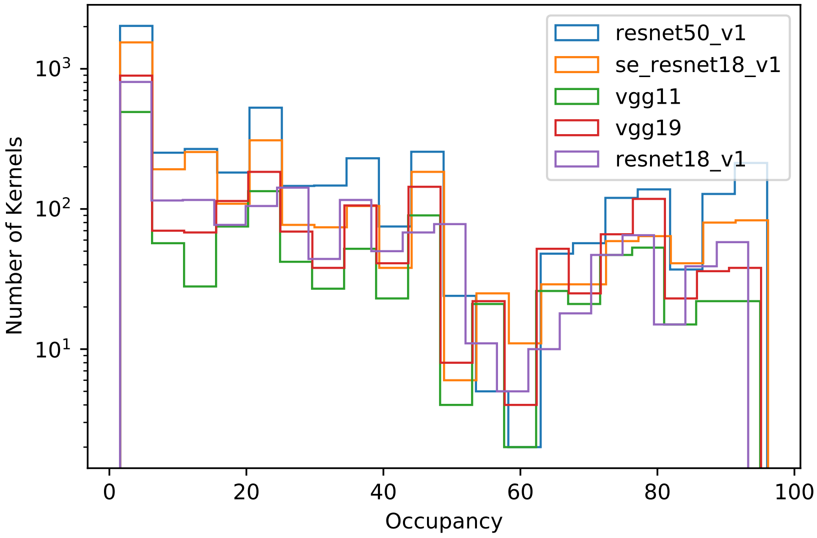


Figure 4.9: Histogram of the achieved occupancy of the kernels for models under study. The batch size is equal to 64. If a kernel is run multiple times, the results are not grouped together and is assumed as a separate kernel. The unit for x-axis is in percent and the y-axis is log based. This graph includes the results for two iterations of training.The kernels were analyzed individually without sharing the resources with any other application.

Using this profiling information, we aim to create a single number that represents the aggregate occupancy of each of these models.

$$C = \sum_{i=1}^{N} q_i t_i \tag{4.3}$$

Figure 4.10 shows the value of C presented in eq. (4.3) for different models and different batch sizes. As we increase the batch size, the C value increases for all the jobs that we studied. ResNet-50 has the highest occupancy compared to all the other deep learning models. While ResNet-50 does not have the largest number of parameters, it is able to achieve higher occupancy compared to all the other models. Since C and M are correlated with each other in the models we studied, both of them can be used for predicting the speedup of the co-location of the deep learning models.

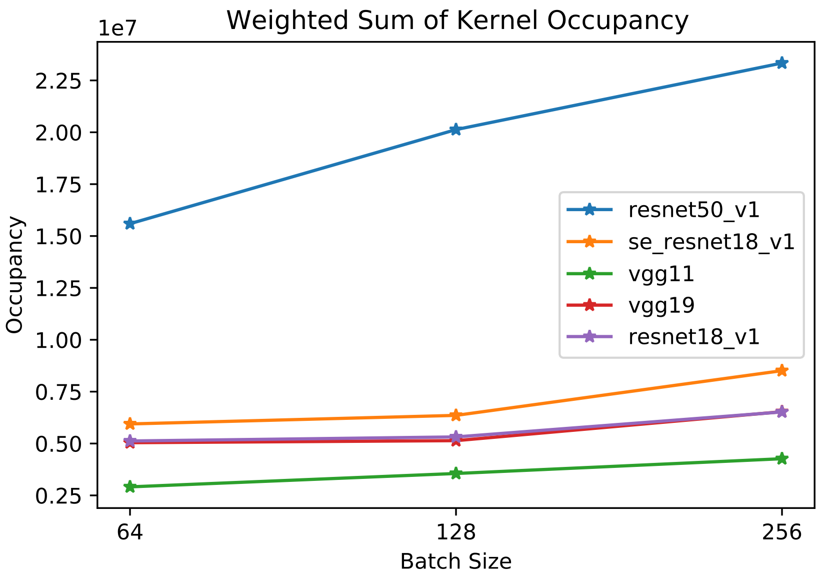


Figure 4.10: Weighted average of the kernel occupancy of the models using various batch sizes

4.2.4 Identifying Relationship between Models and Speedup

Now, we want to use the metrics described in the previous subsections, to build a classifier that is able to predict when the jobs will benefit from co-location and when they will suffer from it.

Figure 4.11 shows the job speedup vs the pair of metrics discussed in the previous subsections. In the experiments described in the figure, MPS is not enabled. Each dot represents a single co-location experiment. The experiments presented in this figure includes all the previous experiments described in this chapter. Each dot may represent an experiment with only two jobs or three jobs co-located. Also, the experiments contain all the batch sizes from 64 to 256 using all the models described. The y-axis is the sum of C values of the individual jobs and x-axis is the sum of M values.

As shown in Figure 4.11, speedup does not have a correlation with the sum of the C and M values when MPS is not enabled. This result is expected because when MPS is not enabled, a kernel with low occupancy that takes a long time to complete can block the execution of any other kernel. Thus, the speedup value when the MPS is not enabled

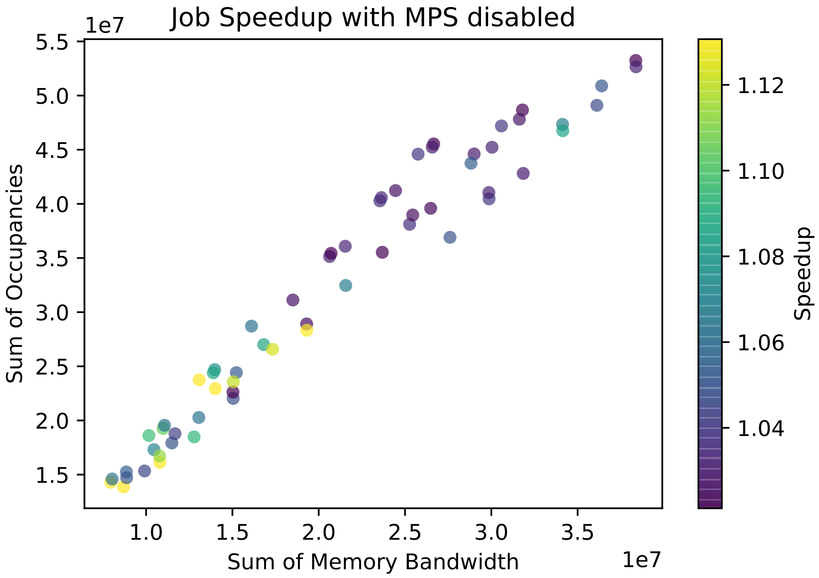


Figure 4.11: Relationship between models and jobs speedup when MPS is disabled.

does not have a linear correlation with either M or C values.

Figure 4.12 shows a similar figure but with MPS enabled. Contrary to fig. 4.12, the cases where speedup is larger than 1 is linearly separable. Using this graph we can create simple classifiers that are able to predict the speedup given two or three different models. For example, a simple criteria like "$\sum C_i \leq 4 * 10^7$" is sufficient for our models and settings. This criteria is represented using the red line. Since the speedup gain is not very significant for the cases above the red line we will not co-locate the jobs above the red line together. We also investigated the cases where the red dots happen. All of the red dots happen when the batch size is 256. Co-location is not recommended for large batch sizes as it may lead to slowdown.

Figure 4.13 shows the exact value of the speedup using colors. As you can see in this figure, jobs with smaller C and M values achieve higher speedups. We will use this heuristic for designing the scheduler that uses co-location in chapter 5.

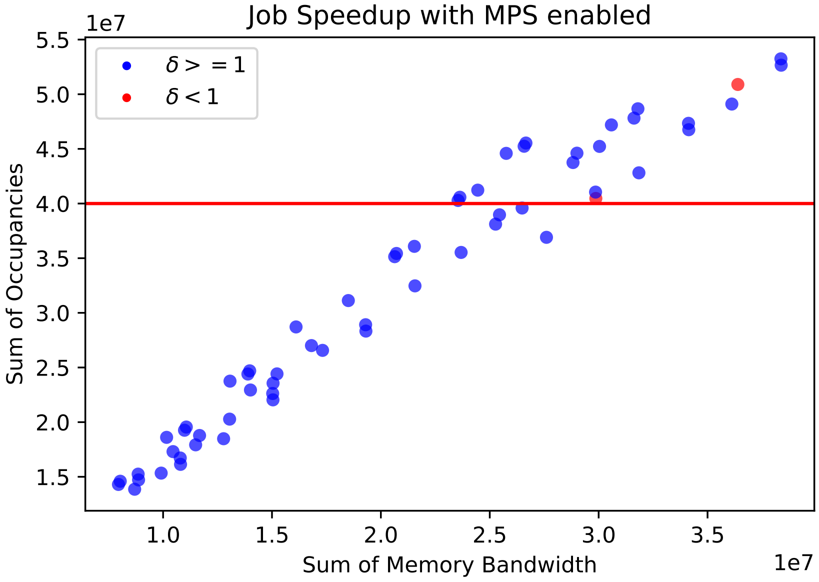


Figure 4.12: Relationship between models and jobs speedup when MPS is enabled.

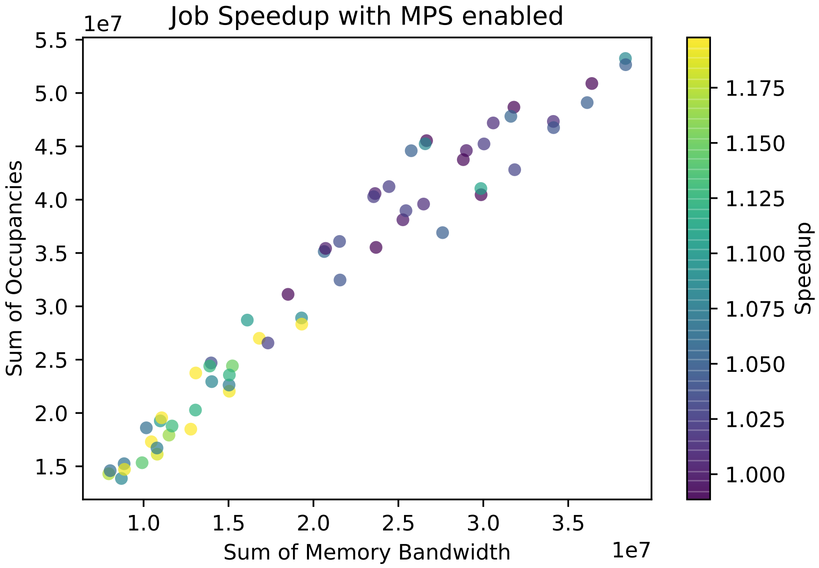


Figure 4.13: Relationship between models and jobs speedup when MPS is enabled.

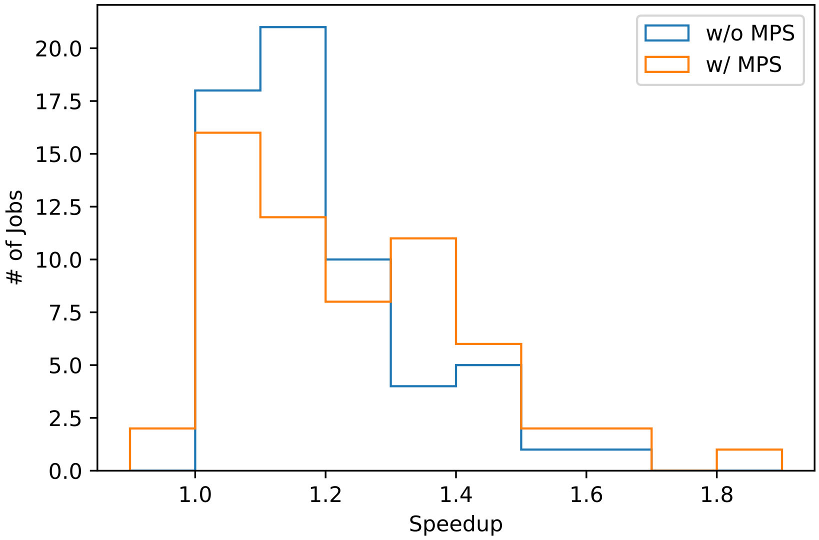


Figure 4.14: MPS effect on Speedup

4.2.5 MPS Effect

From the previous figures it is hard to argue whether MPS is helpful for the speedup or not. Figure 4.14 shows the histogram of the increase in speedup when using MPS. Using MPS leads to higher speedup in most cases. However, for large batch sizes (e.g. 256) using MPS may lead to resource contention and slowdown the jobs execution.

4.3 Summary

In this chapter we introduced the formulation for modeling the speedup. This formulation is able to generalize to co-location of more than two jobs on a single GPU and does not depend on the batch size. We presented detailed profiling of the models used in this study. We created metrics that described the overall performance of the job in terms of key utilization metrics. We also shown why the jobs speedup is not correlated with the values of C and M when MPS is not enabled. It was also shown how you can build classifiers for detecting the compatible jobs when deciding on which job should be co-located with a given job. In the next chapter, we will build on these insights to create a real scheduler that is able to utilize co-location to decrease the makespan and queuing time.

Chapter 5

Scheduler Design and Implementation

In this chapter we explain the design and implementation of a scheduler that uses the metrics described in chapter 4 to make better scheduling decisions. The main goal of this scheduler is to show that co-location decreases the makespan and job queuing time when job is not utilizing the GPU effectively.

5.1 Design

An overall architecture of the scheduler is depicted in fig. 5.1. Our scheduler contains two main components an *Agent* and a *Scheduler*. For each GPU cluster, there is a single scheduler required. The scheduler is the central point that makes scheduling decisions. Agent is the component that runs on every node. Each node has a number of GPUs. Similar to the Kubernetes [12] design, the scheduler only updates the state of the cluster in the database and it is the job of the agents to make sure that the state present in the database is realized. All the communication between the scheduler and the agents is conducted using the REST APIs.

The scheduler sequence diagram is shown in fig. 5.2. First, the user submits a batch

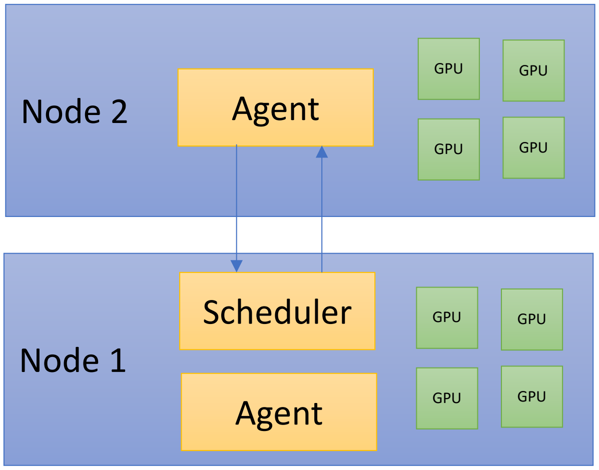


Figure 5.1: Scheduler Architecture

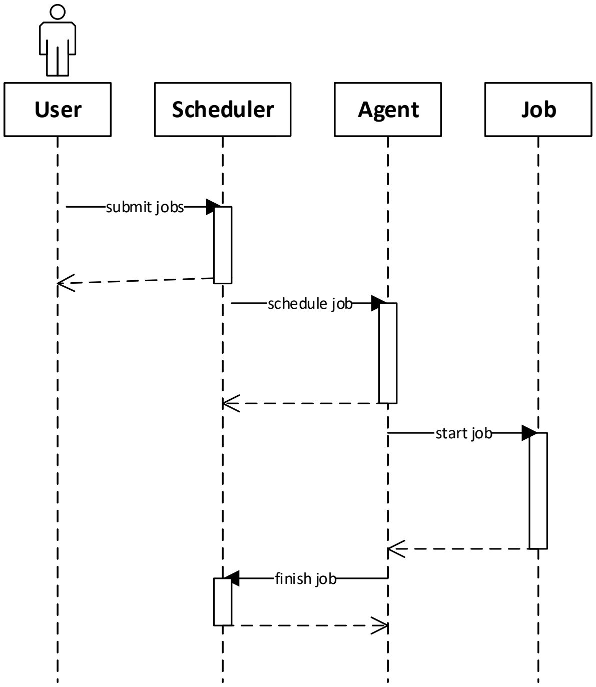


Figure 5.2: Scheduler Sequence Diagram

of jobs to the scheduler. Then, the scheduler decides on which jobs should be scheduled on which agent and updates the node object corresponding to that node. After this, the scheduler notifies the agent corresponding to that node so that it can start scheduling the job. Next, the job starts the execution on the assigned GPU. After the job finishes, it reports the finish time and the time each iteration took back to the scheduler.

Algorithm 1: Scheduler Loop

```
while True do
    jobs ← GetJobsInQueue();
    nodes ← GetAllNodes();
    if length(jobs) > 0 then
        scheduled_nodes,scheduled_jobs ← Scheduler(jobs,nodes);
        foreach node of scheduled_nodes do
            UpdateNode(node);
        end
        foreach job of scheduled_jobs do
            RemoveJobFromQueue(job);
        end
    end
    Sleep(1000);
end
```

Algorithm 1 shows the scheduling loop of the scheduler. The scheduler polls the queue for the list of all the jobs in the queue that are not currently scheduled. It also gets all the active nodes in the cluster. The list of active nodes contains the nodes that already have scheduled a job too. Then, the *nodes* and *jobs* arrays are passed to the algorithm responsible for assigning jobs to nodes. Note that by this design we can substitute the *Scheduler* function with any arbitrary function that we want. This function will return a list of scheduled jobs and the nodes that the jobs were scheduled on. The scheduler will then remove the scheduled jobs from the job queue and update the node object in the database to represent the new state. During the update of the node, the agent for that node is also notified about the new jobs being scheduled for it. This loop is executed every second to dispatch any new jobs to the agents.

Algorithm 2 describes the co-location algorithm that we used for scheduling the jobs.

Algorithm 2: Co-location Algorithm

```
Function GetPerfResult(Job)
    memutil_info ← LoadMemUtilInfoFromFile();
    occupancy_info ← LoadOccupancyInfoFromFile();
    job_occupancy ← occupancy_info [Job.model_name][Job.batch_size];
    job_memutil ← memutil_info [Job.model_name][Job.batch_size];
    return job_occupancy,job_memutil;
Function GetSumJobPerf(Node)
    total_occupancy ← 0;
    total_memutil ← 0;
    foreach Job of Node.Jobs do
        job_occupancy,job_memutil = GetPerfResult(Job);
        total_occupancy ← job_occupancy + total_occupancy;
        total_memutil ← job_memutil + total_memutil;
    return total_occupancy,total_memutil;
Function Colocation(Nodes,Jobs)
    NodePerf ← {},JobPerf ← {};
    scheduled_jobs ← [], scheduled_nodes ← [];
    foreach Node of Nodes do
        NodePerf [Node] = GetSumJobPerf(Node);
    foreach Job of Jobs do
        JobPerf [Job] = GetPerfResult(Job);
    SortByValueAscending(NodePerf);
    SortByValueAscending(JobPerf);
    foreach key,value of JobPerf.items() do
        (job_occupancy,job_memutil),Node ← NodePerf [0];
        if CalculateSpeedup(Node,job_occupancy,job_memutil) < 1 then
            break;
        Node.Jobs.append(Job);
        scheduled_nodes.append(Node);
        scheduled_jobs.append(Job);
        SortByValueAscending(NodePerf);
    return scheduled_nodes,scheduled_jobs;
```

Note that the *Colocation* replaces the *Scheduler* function in Algorithm 1. This function should return a list of nodes that a job is scheduled on and the list of jobs that are being scheduled. Using the weighted memory utilization and weighted kernel occupancy results presented in chapter 4, we try two schedule the jobs to achieve the largest amount of speedup. The heuristic that we use is to schedule the job with lowest weighted kernel occupancy on a node that has the lowest sum of kernel occupancies. This heuristic is based on fig. 4.13. As explained, the smaller values of weighted kernel occupancy and weighted memory utilization lead to higher speedup. In algorithm 2 there is also a threshold for when to stop co-locating jobs. In fig. 4.12 it fairly easy to use a linear regression algorithm to predict the speedup. We use that to avoid packing more jobs together on a single GPU that will cause a slowdown.

The complexity of the *Colocation* function is $O(m \log m) + O(n \log n) + O(mn)$ where m is the number of jobs that are going to be scheduled and n is the number of nodes. The $O(k \log k)$ terms are for the sorting performed in lines 20 and 22 of algorithm 2. We used $O(n)$ for sorting in line 30 of algorithm 2 since we are inserting into an already sorted array. Since this is an array we need to shift all the elements and thus we need $O(n)$. This loop is executed m times for each element in the array thus the overall complexity for the last section of the function is $O(mn)$.

5.2 Implementation

The whole platform for the scheduler is implemented using Python. The scheduler and the agent components are completely independent and communicate with each other over HTTP ReST. We used Redis [7] to store the state of the cluster. Redis is an in-memory key-value database. Redis is executed on the same node as the scheduler. We used Flask [9] for implementing the HTTP interfaces.

The first component that must be started is the scheduler. The scheduler will connect to the Redis database and start the scheduling loop to poll for any new submitted jobs.

Endpoint	Description	Method
/run	Refresh the node jobs and execute any jobs assigned to it	GET

Table 5.1: Agent API

5.2.1 Agent API

Table 5.1 shows the Agent API. The API of the agents is very simple and contains a single */run* endpoint. This endpoint is used to notify the agent about the new jobs being scheduled for it.

5.2.2 Scheduler API

Endpoint	Description	Method
/jobs	Create a batch of jobs	POST
/job	Create a single jobs	POST
/jobs	Get all the jobs inside the queue	GET
/jobs	Delete all the jobs inside the queue	DELETE
/node	Create a new node	POST
/nodes	Get all the nodes inside the cluster	GET
/node/<name>	Get the details for a node	GET
/node/<name>	Update the node status	PUT
/nodes	Delete all the nodes	DELETE
/finished_job	Record the status of a finished job	POST
/finished_jobs	Get all the finished jobs	GET
/finished_jobs	Delete all the finished jobs	DELETE

Table 5.2: Scheduler API

Table 5.2 shows the Scheduler API. These APIs are created for putting objects inside the Redis database. We serialize the Python objects for storing them inside the Redis. As mentioned earlier, Redis is a key-value database. The keys that we use for this project are ”submitted_jobs”, ”nodes”, and ”finished_jobs”.

The endpoints starting with ”/job” are used for managing the job queue. Every job posted to the queue using these APIs will place the job at the end of the queue. The jobs added using this endpoint will be placed in the *submitted_jobs* key of the database.

```
{
    "model_name": "vgg11",
    "batch_size": 128,
    "lr": 0.01,
    "dataset": "cifar10",
    "optim": "sgd",
    "iters": 100,
    "dataset_path": "./data",
    "workers": 2,
    "weight_decay": 0.0001,
    "momentum": 0.1,
    "num_class": 10
}
```

Listing 1: Sample job object

```
{
    "gpus": ["0"],
    "name": "node-1"
}
```

Listing 2: Sample node object

A sample JSON for the job submitted to the ”/apis” is shown in listing 1.

Endpoints starting with ”/nodes” are used for managing the nodes in the cluster. Whenever a new node joins the cluster, it will report the GPUs available on the machine and let the scheduler know about it using these APIs. These APIs will modify the ”nodes” key of the redis database. A sample JSON of the node object is shown in listing 2. In this listing, the node has a single GPU and the node name is ”node-1”.

”/finished_job” endpoints are used for reporting the statistics about the job completion. The statistics include the finish time, start time, and per iteration duration. These endpoints will modify the ”finished_jobs” key of the redis database. This endpoint is always triggered by the agents.

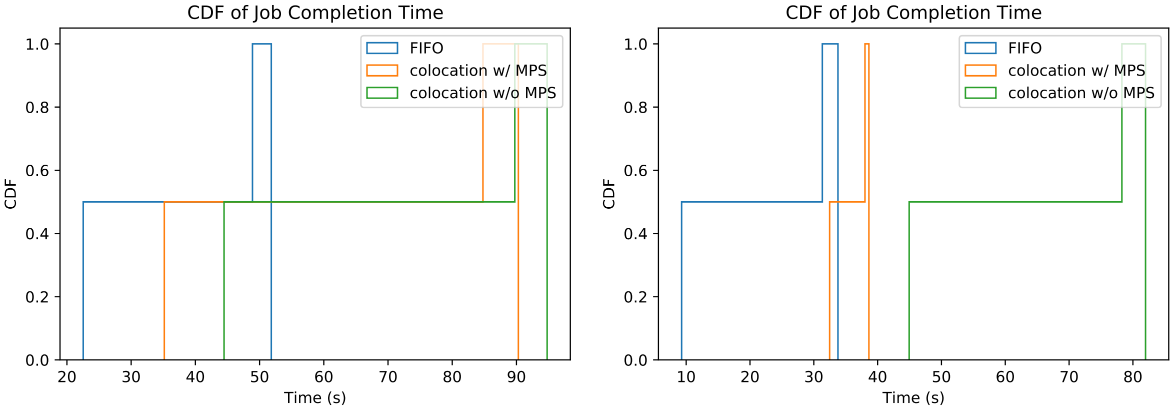
5.3 Summary

In this chapter we presented the algorithm for co-locating deep learning jobs. We also presented the design and implementation of a simple scheduler that leverages co-location as a method to decrease the queuing time of the jobs. Also, the scheduler is designed in a modular way to allow easy implementation of different scheduling algorithms.

Chapter 6

Evaluation

In this chapter we present the end-to-end evaluation of the algorithm 2. We use the metrics presented in chapter 4 to guide the scheduler to perform appropriate co-locations. The scheduling metrics that we are interested in are *makespan*, *job completion time (JCT)*, and *queuing time*. Makespan is a metric used to describe the time between the submission of the first job and the departure of the last job from the cluster. Makespan is a key indicator of the benefit that results from co-locating jobs. JCT is the time between the time that the job starts running and the time that the job leaves the cluster. Queuing time is the time between the job start time and the time that the job was submitted to the cluster. Because of diversity in the DL models and lack of a complete benchmark in this area we present the worst-case and best-case scenarios for the workloads that might occur for co-location. For each scenario we present two different set of results. One set of results when the number of data workers is 10 and another case where the number of data workers is 2. The data workers are used for loading the data from hard disk to the main memory and prepare them for training. The dataset used for training all the models is the CIFAR10 [25] dataset.



(a) Ten Data Workers

(b) Two Data Workers

Figure 6.1: CDF of job completion time for the best-case scenario.

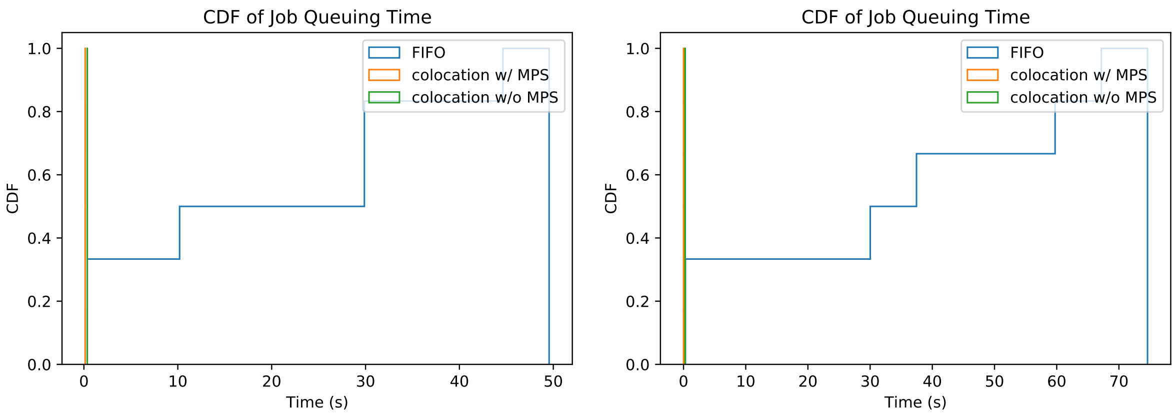
6.1 Experimental Setup

We evaluated algorithm 2 on a real system with two V100 GPUs running Ubuntu 18.04. Each of the V100 GPUs are placed in different machines connected to each other over the network. We used the PyTorch implementation of the models in the analysis. We compare algorithm 2 with First in First out (FIFO) algorithm in which jobs are scheduled in the order of arrival. FIFO serves as the baseline for running all the jobs alone. We also present our results with MPS both enabled and disabled.

6.2 Best-case Scenario

The best-case scenario is that all the jobs submitted to the cluster have small batch size. We use the models that have the largest gain from co-location in chapter 4. The models are VGG19, VGG11, and ResNet-18 with a batch size of 64. We submit two set of these jobs and train them for 1000 iterations.

Figure 6.1 shows the CDF of the job completion time when scheduling using FIFO and co-location. The histogram shows that FIFO attains a smaller job completion time for each individual job. The reason for this is that in FIFO jobs have exclusive access to



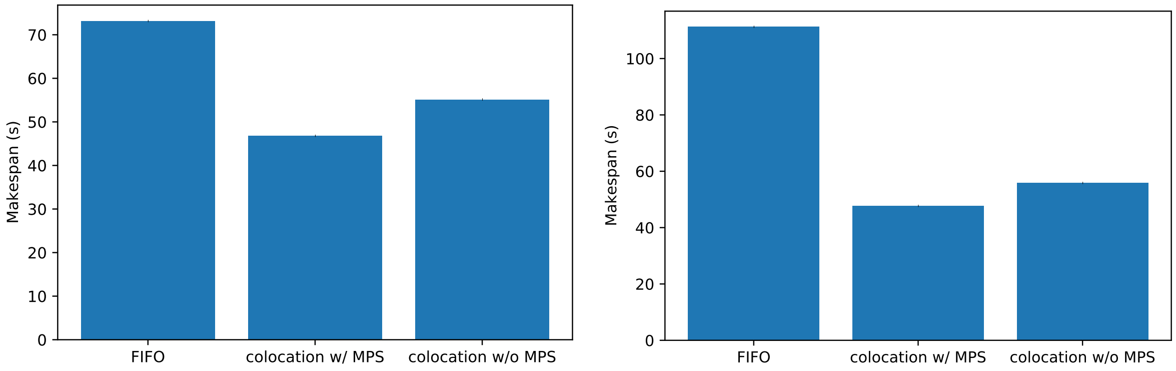
(a) Ten Data Workers (b) Two Data Workers

Figure 6.2: CDF of queuing time for the best-case scenario.

the GPU and the individual kernels are able to run faster. However, when the jobs are being co-located, the jobs do not have exclusive access to the GPU and they are running slower compared to the original case. Using eq. (4.1) we can determine whether they are running faster together or not. We submit the jobs in one batch altogether and it is the job of the scheduler to determine the best co-locations. Algorithm 2 is able to find the best co-location according to the profiling result in this scenario. When using smaller number of data workers, the JCT of the jobs increase for FIFO. This suggests that FIFO needs more data workers to load enough batches of data for efficient training.

Figure 6.2b shows the queuing time of the jobs under two different policies. As expected, the queuing time is significantly smaller compared to FIFO. This is because the jobs get scheduled much faster without needing to wait for the completion of the other jobs.

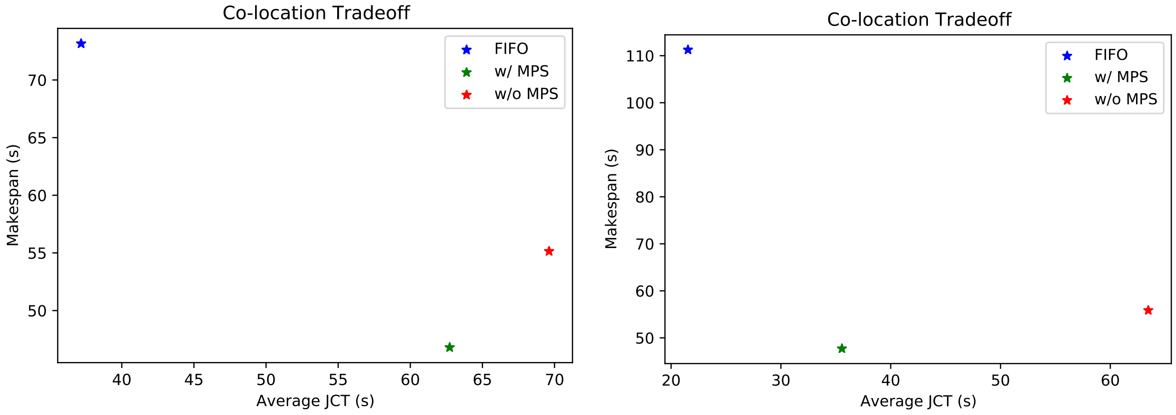
Small queuing time is especially important for the hyperparameter tuning phase of the DL training. AutoML [32, 30] techniques rely on the current progress of the jobs to determine the ones that are promising. Small queuing time helps with faster scheduling of more jobs. By monitoring the progress of more jobs, we can detect the non-promising jobs and stop their execution to avoid wasting resources.



(a) Ten Data Workers

(b) Two Data Workers

Figure 6.3: Makespan for the best-case scenario.



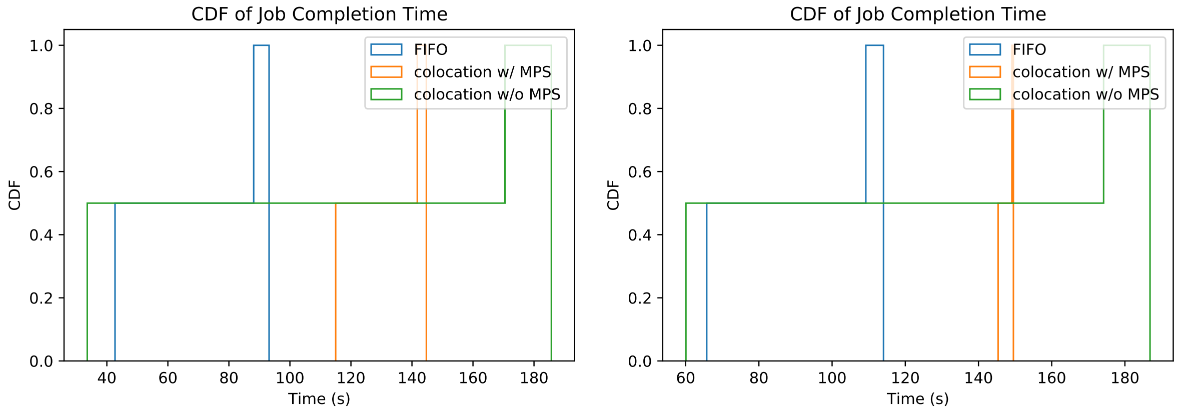
(a) Ten Data Workers

(b) Two Data Workers

Figure 6.4: Tradeoff in using co-location for the best-case scenario.

Figure 6.3 shows the makespan for different scheduling policies. Co-location gives a 2x decrease in the makespan for this specific workload when using two data workers. This figure highlights the importance of co-location specially when the utilization of the individual jobs is small. While the individual jobs see an increase in the JCT value, this increase is the price that the jobs pay for better utilization of the GPUs and smaller makespan values.

Figure 6.4 shows that makespan can be reduced by more than 50% in exchange for an increase in JCT of 25% in this scenario when using two data workers. When using ten data workers makespan can be reduced by more than 37% in exchange for an increase in



(a) Ten Data Workers

(b) Two Data Workers

Figure 6.5: JCT for the worst-case scenario.

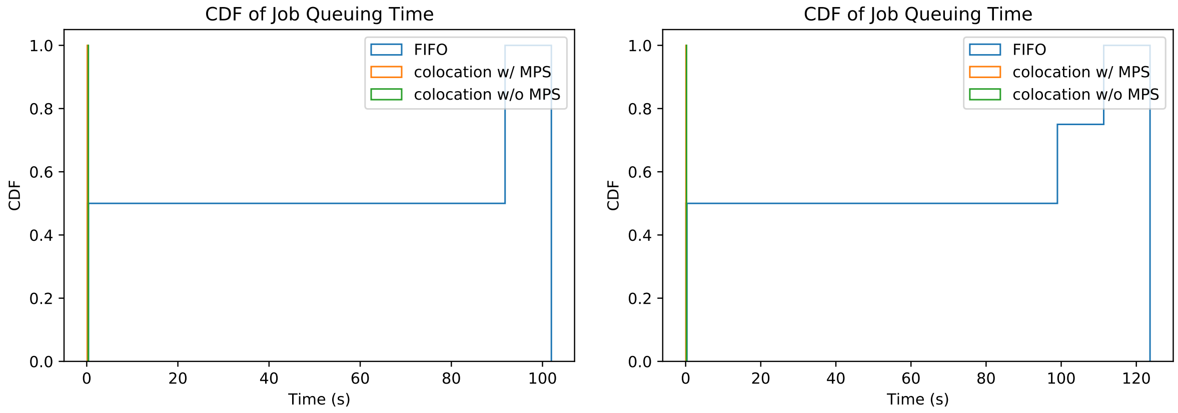
JCT of 61% in this scenario.

6.3 Worst-case Scenario

The worst-case scenario is that we make bad co-location decisions by putting the jobs that together lead to minor speedup. Also, we need to make sure that our classifier predicts that the jobs will benefit from the co-location. Based on our profiling data, the speedup value for co-location of SENet18 and ResNet18 is 1.02 when the batch size is 256. Like the previous section, we present the results for two data workers and ten data workers.

Figure 6.5 shows the CDF of job completion. Similar to the previous figure FIFO has the smallest JCT. The reason is that FIFO jobs are exclusively running on the GPU and hence they are faster. An important difference between fig. 6.5 and fig. 6.1 is that when the batch size is large, jobs' JCT never come close the FIFO case. The resource contention makes it hard to become close to the performance of job running alone.

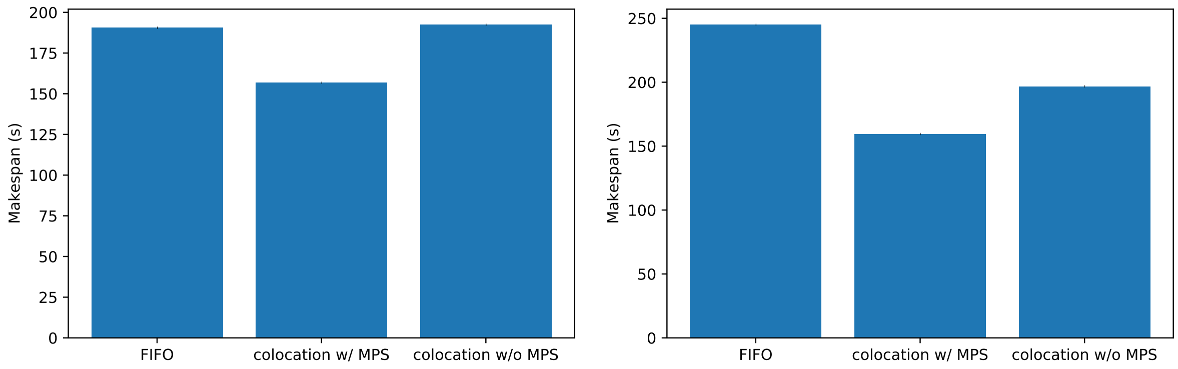
Figure 6.6 shows the queuing time for the jobs in this scenario. In this scenario two jobs are co-located together on a single GPU. Since we have a total of 4 jobs and 2 GPUs we experience almost no queuing delay compared to the FIFO baseline.



(a) Ten Data Workers

(b) Two Data Workers

Figure 6.6: CDF of queuing time for the worst-case scenario.



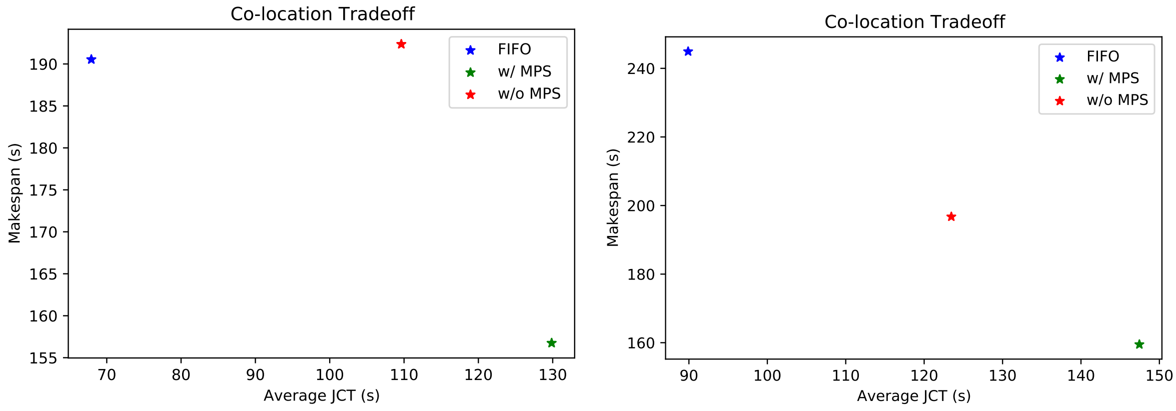
(a) Ten Data Workers

(b) Two Data Workers

Figure 6.7: Makespan for the worst-case scenario.

Figure 6.7 shows the result for makespan. The gains for this case are smaller compared to the fig. 6.3. When using two data workers, there is a significant decrease in makespan compared to FIFO. The reason is that FIFO requires more data workers to be able to use the GPU efficiently. Using small number of workers, affects the performance of individual jobs and leads to 25% increase in makespan for large batch sizes.

Figure 6.8 shows the tradeoff for using different scheduling policies. The difference between this figure and fig. 6.4 is that in this scenario using MPS leads to larger JCT compared to not using MPS. Still, MPS is able to attain lower makespan compared to not using MPS.



(a) Ten Data Workers

(b) Two Data Workers

Figure 6.8: Tradeoff in using co-location for the worst-case scenario.

We showed that co-location can work in different scenarios. As expected the decrease in makespan is larger when the batch size is small and when the models are not able to utilize the GPU effectively. Smaller number of data workers does not significantly affect the makespan when using co-location. Using MPS will also improve the performance in all the different scenarios. Additionally showed why co-location can significantly reduce the queuing time.

6.4 Discussion

We presented end-to-end analysis of the co-location algorithm using two different scenarios. We showed that when the batch size is small and model has a low memory bandwidth and SM utilization, co-location leads to an almost 2X speedup. We also showed that in the worst case scenario co-location is slightly better than FIFO. Also, if there are not enough CPUs available on the system for data workers, co-location can lead to significant decrease in makespan compared to the FIFO case. Co-location is a trade-off between the makespan and the JCT metrics. Co-location leads to smaller queuing times and larger JCT for individual jobs. If the jobs require early feedback (e.g. hyperparameter tuning) co-location becomes more important as it will give faster feedback to a larger number of jobs so that we can decide whether we want to continue running them or not. However, if the jobs require low latency and need to run to completion in the shortest amount of time, co-location is not a good solution. The other down side to co-location is requiring extensive profiling to find out about the SM and memory bandwidth utilization. However, having a database of the common kernels used in the deep learning frameworks and their profiling information can omit the need for the extensive profiling.

Chapter 7

Conclusion and Future Work

In this thesis we presented a detailed study on the performance impact of co-location of different deep learning models on a single GPU. We showed how metrics like weighted memory bandwidth utilization and weighted occupancy can help with selecting compatible co-locations. The key takeaways from this thesis are the following:

- Due to various reasons, individual jobs may not be able to fully utilize all the resources on a single GPU. Especially as the GPUs keep evolving, it becomes more difficult for the designers of the frameworks to optimize all the previous models in the new architecture. Co-location is a simple, yet effective, technique that allows the designers of cluster schedulers to improve the utilization of GPUs and reduce the queuing time.

- We modeled the speedup in case of co-location. Our modeling is able to generalize to co-location of more than two jobs and does not depend on the batch size of the models.

- We provided in depth profiling of the models on a V100 GPU. We showed that by profiling the individual models, we can predict whether the jobs will gain a speedup when packed together or not.

- Our method of co-location is independent of the deep learning framework and does not require any modification to the users' code.

7.1 Future Work

This work is one of the first works that studies the implications of using Volta MPS in deep learning jobs. Using the metrics and tools provided in this thesis, GPU cluster schedulers can be built that better predict which jobs will benefit from the packing and which jobs won't.

In this work we let the DL frameworks decide on how the kernels are dispatched. As shown in figs. 4.7 and 4.9, individual kernels can have different memory bandwidths and occupancies. Designing a kernel context manager that can decide which kernels execution can be overlapped with each other while other kernels will perform better when they run individually. This will further improve the performance of co-location by only allowing the execution of compatible kernels.

Evaluation of proposed approaches on GPUs other than V100 would be also useful. In this work we only had access to V100 GPUs and thus we could not verify whether these results hold for other than architectures other than Volta or not. Also, MPS has not yet been released for the recent Ampere [4] architecture. If the same results hold for other GPU architectures too, a one-time creation of the performance model would be sufficient to guide the scheduling decisions for all different GPU architectures.

Studying the security implications of co-location is also an interesting extension of this work. Co-located jobs pose a great security threat in the cloud environments that prevents co-location being implemented in the public cloud. Studying the address isolation and the side channels that co-location exposes can be a future step for this work.

Bibliography

[1] Apache hadoop. https://hadoop.apache.org/. (Accessed on 12/25/2020).

[2] Cuda warps and occupancy. https://on-demand.gputechconf.com/gtc-express/2011/presentations/cuda_webinars_WarpsAndOccupancy.pdf. (Accessed on 11/07/2020).

[3] Multi-process service. https://docs.nvidia.com/deploy/pdf/CUDA_Multi_Process_Service_Overview.pdf. (Accessed on 11/16/2020).

[4] nvidia-ampere-architecture-whitepaper.pdf. https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/nvidia-ampere-architecture-whitepaper.pdf. (Accessed on 12/06/2020).

[5] Nvidia doubles down: Announces a100 80gb gpu, supercharging world's most powerful gpu for ai supercomputing — nvidia newsroom. https://nvidianews.nvidia.com/news/nvidia-doubles-down-announces-a100-80gb-gpu-supercharging-worlds-most-powerful-g (Accessed on 11/22/2020).

[6] Nvidia nsight compute — nvidia developer. https://developer.nvidia.com/nsight-compute. (Accessed on 11/29/2020).

[7] Redis. https://redis.io/. (Accessed on 12/19/2020).

[8] Volta architecutre. https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf. (Accessed on 10/20/2020).

[9] Welcome to flask — flask documentation (1.1.x). https://flask.palletsprojects.com/en/1.1.x/. (Accessed on 12/20/2020).

[10] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. Tensorflow: A system for large-scale machine learning. In *12th {USENIX} symposium on operating systems design and implementation ({OSDI} 16)*, pages 265–283, 2016.

[11] Tom B Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. *arXiv preprint arXiv:2005.14165*, 2020.

[12] Brendan Burns, Brian Grant, David Oppenheimer, Eric Brewer, and John Wilkes. Borg, omega, and kubernetes. *Queue*, 14(1):70–93, 2016.

[13] Tianqi Chen, Mu Li, Yutian Li, Min Lin, Naiyan Wang, Minjie Wang, Tianjun Xiao, Bing Xu, Chiyuan Zhang, and Zheng Zhang. Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems. *arXiv preprint arXiv:1512.01274*, 2015.

[14] Christina Delimitrou and Christos Kozyrakis. Quasar: Resource-efficient and qos-aware cluster management. *SIGPLAN Not.*, 49(4):127–144, February 2014.

[15] Yifan Gong, Baochun Li, Ben Liang, and Zheng Zhan. Chic: Experience-driven scheduling in machine learning clusters. In *Proceedings of the International Symposium on Quality of Service*, IWQoS '19, New York, NY, USA, 2019. Association for Computing Machinery.

[16] Juncheng Gu, Mosharaf Chowdhury, Kang G. Shin, Yibo Zhu, Myeongjae Jeon, Junjie Qian, Hongqiang Liu, and Chuanxiong Guo. Tiresias: A GPU cluster manager for distributed deep learning. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 485–500, Boston, MA, February 2019. USENIX Association.

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

[18] Kurt Hornik, Maxwell Stinchcombe, Halbert White, et al. Multilayer feedforward networks are universal approximators. *Neural networks*, 2(5):359–366, 1989.

[19] Jie Hu, Li Shen, and Gang Sun. Squeeze-and-excitation networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 7132–7141, 2018.

[20] Myeongjae Jeon, Shivaram Venkataraman, Amar Phanishayee, Junjie Qian, Wencong Xiao, and Fan Yang. Analysis of large-scale multi-tenant {GPU} clusters for {DNN} training workloads. In *2019 {USENIX} Annual Technical Conference ({USENIX}{ATC} 19)*, pages 947–960, 2019.

[21] Eunji Jeong, Sungwoo Cho, Gyeong-In Yu, Joo Seong Jeong, Dong-Jin Shin, and Byung-Gon Chun. JANUS: Fast and flexible deep learning via symbolic graph execution of imperative programs. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 453–468, Boston, MA, February 2019. USENIX Association.

[22] Norm Jouppi. Google supercharges machine learning tasks with tpu custom chip. *Google Blog, May*, 18:1, 2016.

[23] M. Khairy, Z. Shen, T. M. Aamodt, and T. G. Rogers. Accel-sim: An extensible simulation framework for validated gpu modeling. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, pages 473–486, 2020.

[24] Alexandros Koliousis, Pijika Watcharapichat, Matthias Weidlich, Luo Mai, Paolo Costa, and Peter Pietzuch. Crossbow: Scaling deep learning with small batch sizes on multi-gpu servers. *Proc. VLDB Endow.*, 12(11):1399–1412, July 2019.

[25] Alex Krizhevsky, Geoffrey Hinton, et al. Learning multiple layers of features from tiny images. 2009.

[26] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. *Communications of the ACM*, 60(6):84–90, 2017.

[27] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *nature*, 521(7553):436–444, 2015.

[28] Yann LeCun, Bernhard Boser, John S Denker, Donnie Henderson, Richard E Howard, Wayne Hubbard, and Lawrence D Jackel. Backpropagation applied to handwritten zip code recognition. *Neural computation*, 1(4):541–551, 1989.

[29] Hao Li, Di Yu, Anand Kumar, and Yi-Cheng Tu. Performance modeling in cuda streams—a means for high-throughput data processing. In *2014 IEEE International Conference on Big Data (Big Data)*, pages 301–310. IEEE, 2014.

[30] Liam Li, Kevin Jamieson, Afshin Rostamizadeh, Ekaterina Gonina, Moritz Hardt, Benjamin Recht, and Ameet Talwalkar. Massively parallel hyperparameter tuning. *arXiv preprint arXiv:1810.05934*, 2018.

[31] Mu Li, David G. Andersen, Jun Woo Park, Alexander J. Smola, Amr Ahmed, Vanja Josifovski, James Long, Eugene J. Shekita, and Bor-Yiing Su. Scaling distributed

machine learning with the parameter server. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 583–598, Broomfield, CO, October 2014. USENIX Association.

[32] Richard Liaw, Romil Bhardwaj, Lisa Dunlap, Yitian Zou, Joseph E Gonzalez, Ion Stoica, and Alexey Tumanov. Hypersched: Dynamic resource reallocation for model development on a deadline. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 61–73, 2019.

[33] Hongzi Mao, Malte Schwarzkopf, Shaileshh Bojja Venkatakrishnan, Zili Meng, and Mohammad Alizadeh. Learning scheduling algorithms for data processing clusters. In *Proceedings of the ACM Special Interest Group on Data Communication*, pages 270–288. 2019.

[34] Deepak Narayanan, Keshav Santhanam, Fiodar Kazhamiaka, Amar Phanishayee, and Matei Zaharia. Heterogeneity-aware cluster scheduling policies for deep learning workloads. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 481–498. USENIX Association, November 2020.

[35] Kay Ousterhout, Patrick Wendell, Matei Zaharia, and Ion Stoica. Sparrow: Distributed, low latency scheduling. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, page 69–84, New York, NY, USA, 2013. Association for Computing Machinery.

[36] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library. In

Advances in Neural Information Processing Systems, volume 32, pages 8026–8037. Curran Associates, Inc., 2019.

[37] Yanghua Peng, Yixin Bao, Yangrui Chen, Chuan Wu, and Chuanxiong Guo. Optimus: An efficient dynamic resource scheduler for deep learning clusters. In *Proceedings of the Thirteenth EuroSys Conference*, EuroSys '18, New York, NY, USA, 2018. Association for Computing Machinery.

[38] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

[39] Wencong Xiao, Romil Bhardwaj, Ramachandran Ramjee, Muthian Sivathanu, Nipun Kwatra, Zhenhua Han, Pratyush Patel, Xuan Peng, Hanyu Zhao, Quanlu Zhang, Fan Yang, and Lidong Zhou. Gandiva: Introspective cluster scheduling for deep learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 595–610, Carlsbad, CA, October 2018. USENIX Association.

[40] Peifeng Yu and Mosharaf Chowdhury. Salus: Fine-grained gpu sharing primitives for deep learning applications. *arXiv preprint arXiv:1902.04610*, 2019.

[41] Matei Zaharia, Dhruba Borthakur, Joydeep Sen Sarma, Khaled Elmeleegy, Scott Shenker, and Ion Stoica. Delay scheduling: A simple technique for achieving locality and fairness in cluster scheduling. In *Proceedings of the 5th European Conference on Computer Systems*, EuroSys '10, page 265–278, New York, NY, USA, 2010. Association for Computing Machinery.

[42] Matei Zaharia, Andrew Chen, Aaron Davidson, Ali Ghodsi, Sue Ann Hong, Andy Konwinski, Siddharth Murching, Tomas Nykodym, Paul Ogilvie, Mani Parkhe, et al. Accelerating the machine learning lifecycle with mlflow. *IEEE Data Eng. Bull.*, 41(4):39–45, 2018.

[43] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauly, Michael J Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Presented as part of the 9th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 12)*, pages 15–28, 2012.

[44] Ce Zhang and Christopher Ré. Dimmwitted: A study of main-memory statistical analytics. *Proc. VLDB Endow.*, 7(12):1283–1294, August 2014.

[45] Xia Zhao, Magnus Jahre, and Lieven Eeckhout. Hsm: A hybrid slowdown model for multitasking gpus. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '20, page 1371–1385, New York, NY, USA, 2020. Association for Computing Machinery.