Co-location of ML 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 ML Jobs in GPU Clusters

Iman Tabrizian

Master of Applied Science

Graduate Department of Electrical and Computer Engineering

University of Toronto

2020

I would like to thank my parents and my dear brother for their amazing support. Without their support, I would not been able to do anything. I want to sincerely thank my advisor Professor Alberto Leon-Garcia for giving me the ability to work independently. His insightful comments have greatly impacted this project. I would like to thank Kaveh Mahdaviani and Professor Bianca Schroeder for their time and insightful discussions.

Contents

1 Introduction **1**

2 Background **3**

2.1 GPUs 3

2.1.1 GPU Memory and Compute Capabilities 4

2.1.2 CUDA 5

2.1.3 Thread Block Scheduling 5

2.1.4 Life Cycle of a GPU accelerated Application 7

2.1.5 CUDA Streams 7

2.1.6 Multi-Process Service (MPS) 8

2.2 Machine Learning Frameworks 8

2.2.1 Tensorflow 9

2.2.2 PyTorch 9

2.2.3 MXNet 10

3 Motivation **11**

3.1 Deep Learning Job Characteristics 11

3.1.1 Heterogeneity of GPU Clusters 11

3.1.2 Statistical Constraints 12

3.1.3 Monitoring, Logging, and Data Loading 12

4 Training Job Co-location 14

4.1 Modeling Co-location Speed Up 14

4.1.1 Problem Formulation 14

4.1.2 Value of δ in the Co-location of Various Deep Learning Models . . 15

4.2 Profiling Training Jobs 16

4.2.1 Memory Utilization 19

4.2.2 Compute Utilization 19

5 Inference Job Co-location 23

6 Related Work 24

6.1 GPU Scheduling 24

6.2 Jobs Interference 26

6.3 Scheduling of Data Analytics Job 26

7 Conclusion and Future Work 27

Bibliography 28

List of Tables

4.1 Characteristics of the models used in the experiments 16

4.2 Description of the experiment numbers 16

List of Figures

2.1 GPU Architecture 3

2.2 GPU Capabilities over time 4

2.3 Grid Block vs Thread Block vs Thread 6

2.4 Lifecycle of a GPU accelerated application 7

2.5 Interaction between CUDA contexts and Work Queues on a GPU 8

4.1 Co-location of Multiple Jobs 15

4.2 Co-locating 2 jobs together using various batch sizes. x-axis shows the experiment number described in table 4.2 17

4.3 Co-location of 2 or 3 jobs together when the batch size is equal to 64. x-axis shows the experiment number described in table 4.2 17

4.4 Co-location of 2 or 3 jobs together when the batch size is equal to 128. x-axis shows the experiment number described in table 4.2 18

4.5 Co-location of 2 or 3 jobs together when the batch size is equal to 256. x-axis shows the experiment number described in table 4.2 18

4.6 Weighted average of the memory utilization of the models using various batch sizes 19

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 application. 20

4.8 Weighted average of the kernel utilization of the models using various batch sizes 21

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. 22

6.1 Memory Allocation in Forward and Backward Pass 25

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 photography. These advancements have been made possible thanks to the specialized hardwares such as GPUs, TPUs[14], 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 the 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 compute clusters that provide 100s to 1000s of GPUs. It will 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[5] architecture GPUs now include hardware support for execution of multiple processes together.

In this dissertation, 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 as it will decrease the training speed for these jobs. Additionally, we study the potential reasons behind the incompatibility of the workloads and how they can be used to predict the incompatible jobs.

The next chapter gives an overview of the scheduling algorithms in other domains and an introduction on how GPUs are used for general processing. Chapter 3 discusses the motivation for co-location of the DL jobs. **??** discusses the design of our proposed scheduler that employs co-location to improve the job completion time (JCT). **??** provides the experimental results for this design and how it compares to the schedulers that do not employ co-location. Finally, Chapter 7 concludes the thesis and provides future directions on how this work can be extended.

Chapter 2

Background

2.1 GPUs

GPUs were among the first accelerators to be introduced alongside with CPUs. GPUs are throughput optimized in contrast to the CPUs which are latency optimized. They consist of many cheap 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 in 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

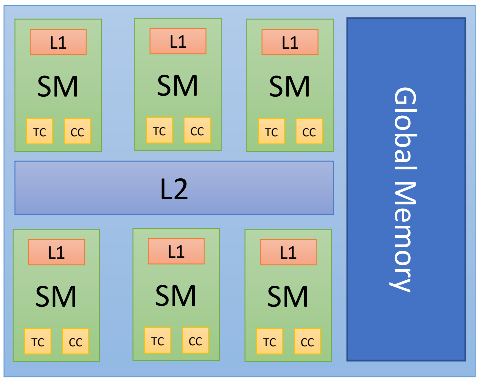

Figure 2.1: GPU Architecture

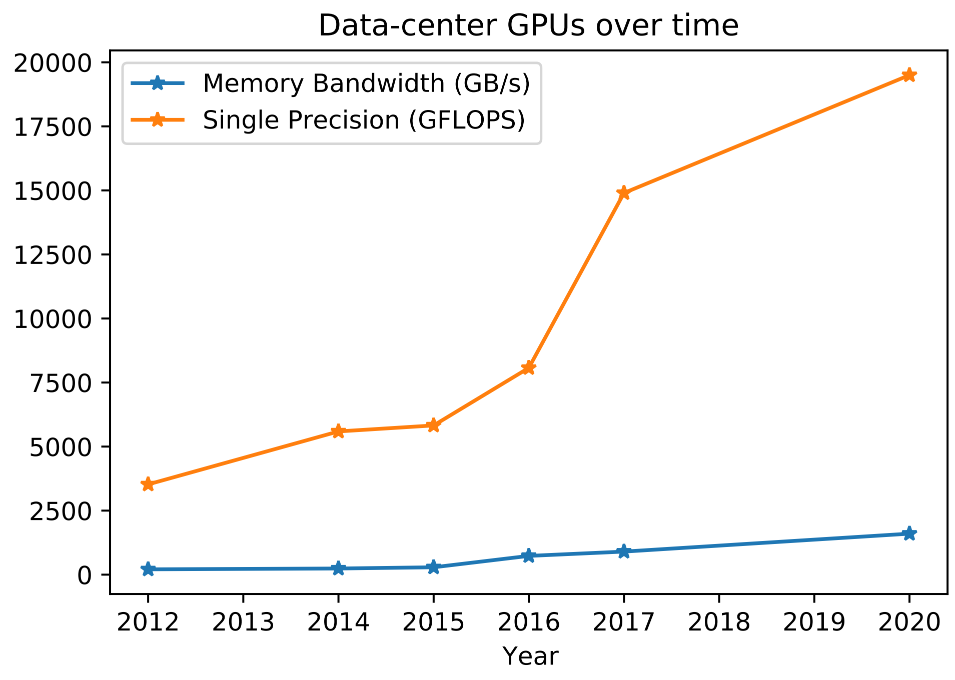

Figure 2.2: GPU Capabilities over time

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[5]. 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 operations which are present in the Deep Learning workloads. They were introduced in the Volta[5] microarchitecture in 2017.

2.1.1 GPU Memory and Compute Capabilities

GPUs memory bandwidth and compute capabilities have increased by a tremendous amount over time. Figure 2.2 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 they do not fully utilize the GPU individually.

2.1.2 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 2.3 shows the difference between a thread block, grid block, and a thread.

2.1.3 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*[1] 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

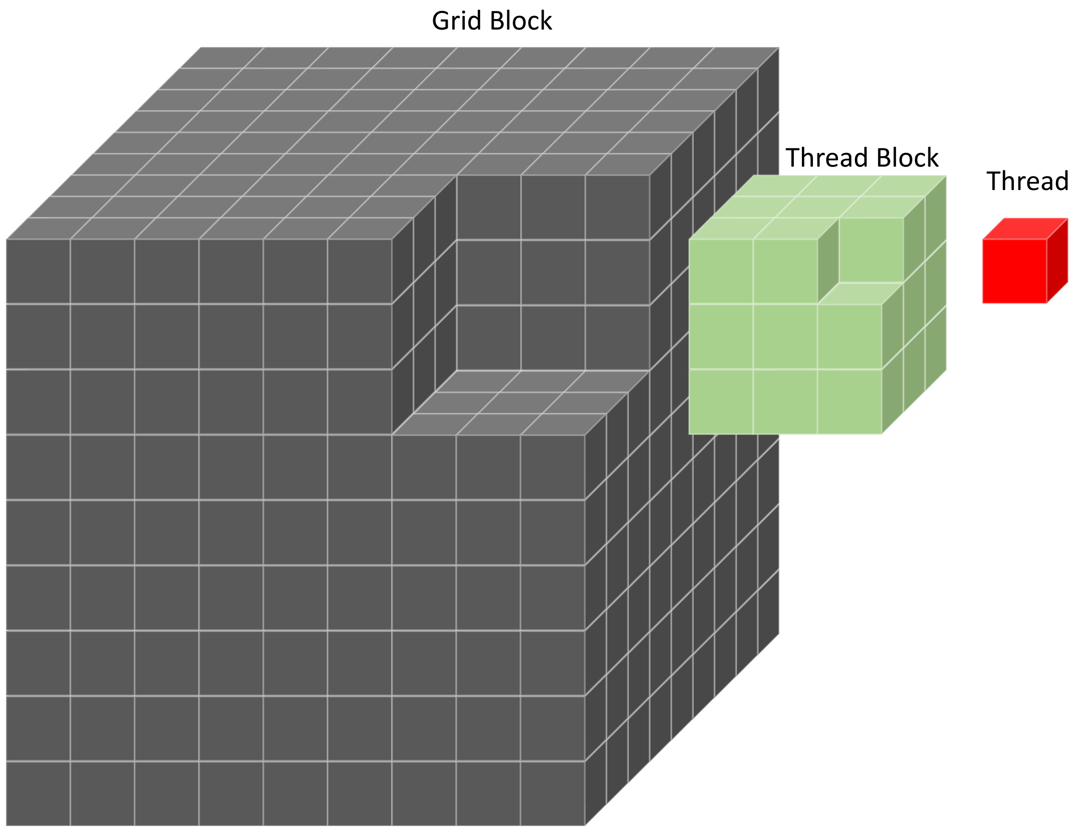

Figure 2.3: Grid Block vs Thread Block vs Thread

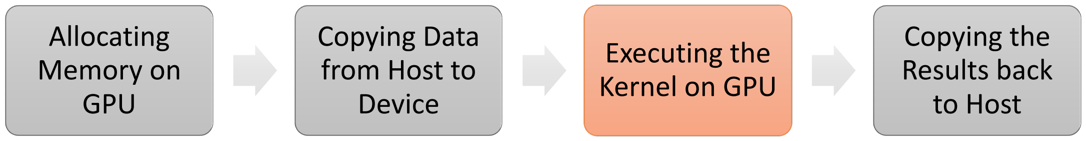
Allocating Memory on GPU
Copying Data from Host to Device
Executing the Kernel on GPU
Copying the Results back to Host

Figure 2.4: Lifecycle of a GPU accelerated application

in warp might we stalled on a memory load and is not yet ready to be scheduled. If there is not enough warps in flight ready to be scheduled instead of the stalled thread block, the achieved occupancy will be lower than the Theoretical Occupancy. Theoretical Occupancy serves as the upper bound for the Achieved Occupancy.

2.1.4 Life Cycle of a GPU accelerated Application

Figure 2.4 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 required to run the computation is executed. When the computation is complete, all the results are copied back into the host memory. All these operations must be executed 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 [2] allows multiple CUDA contexts to run applications on the GPU concurrently. This is explained in more details in section 2.1.6.

2.1.5 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 MPS [2], it is not possible to run commands from another CUDA context unless the work from the current CUDA context has finished. A

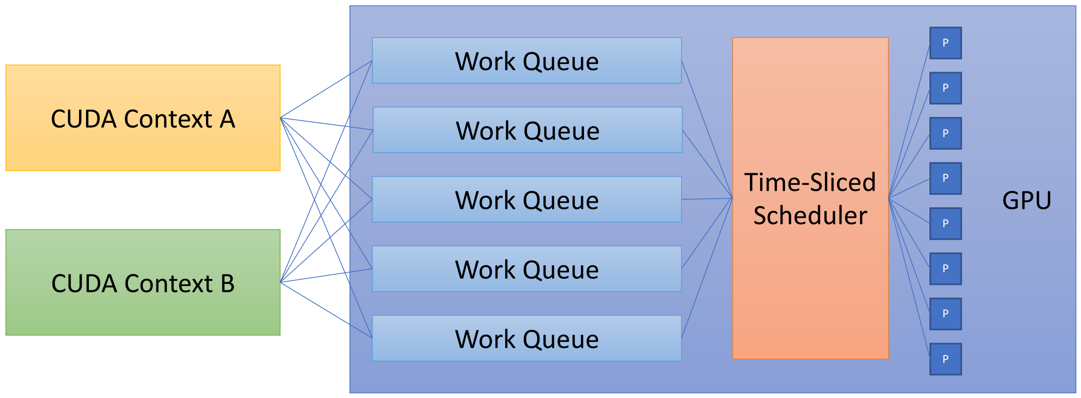

Figure 2.5: Interaction between CUDA contexts and Work Queues on a GPU

common use case for CUDA streams is overlapping computation and communication to speedup the Kernel execution.

2.1.6 Multi-Process Service (MPS)

MPS [2] 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 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 2.5 shows how CUDA contexts interact with the GPU to schedule work. GPUs have a hardware construct named *Work Queue*. Different CUDA contexts

2.2 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 where 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 in this field making sure that you have the best performing algorithm on the new hardware is not very 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.

2.2.1 Tensorflow

Tensorflow [6] 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.

2.2.2 PyTorch

PyTorch [19] 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 [13] have also taken a hybrid approach to maintain both a high developer productivity and optimized and fast execution.

2.2.3 MXNet

MXNet [7] 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 2.2.1 and 2.2.2. MXNet also supports a *hybrid* approach similar to Janus [13].

Chapter 3

Motivation

In this chapter we will discuss the unique characteristics of deep learning jobs that make them suitable for co-location. We will discuss why deep learning job are not usually able to saturate GPU resources in today's data centers.

3.1 Deep Learning Job 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 [3] 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 more scarce than the older GPUs due to the high demand and better performance. Users usually 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. [Put a figure about increasing batch size and training speed]

3.1.2 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, may be different from the hyperparamters 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 diminished returns [15, 24]. This constraint further leaves the GPUs underutilized.

3.1.3 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 [23], 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. [Put a figure about PyTorch hello world CPU task]

Chapter 4

Training Job Co-location

4.1 Modeling Co-location Speed Up

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

4.1.1 Problem Formulation

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 eq. (4.1) is true.

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

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

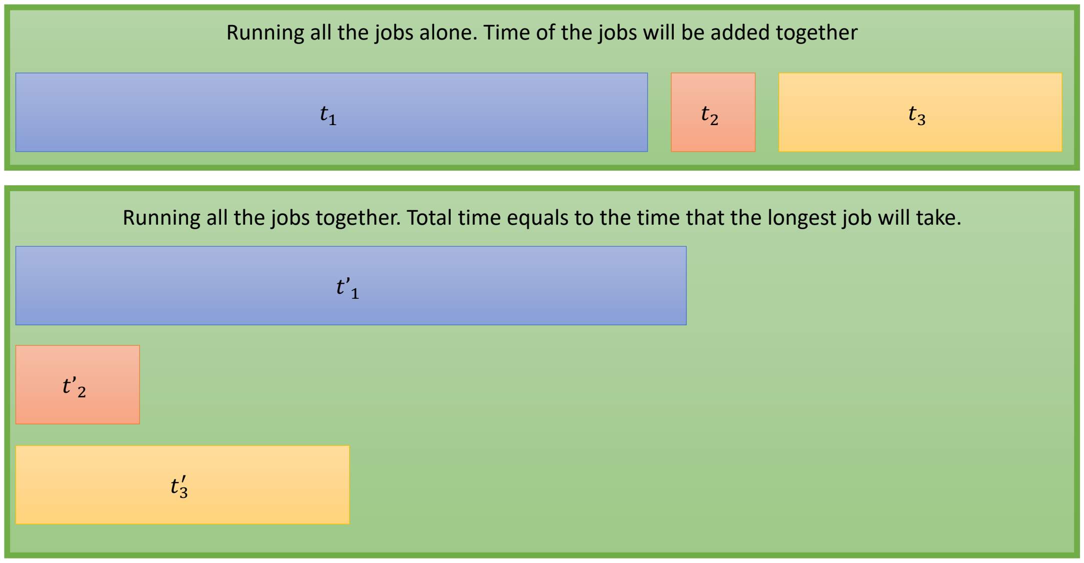
Running all the jobs alone. Time of the jobs will be added together
t_1
t_2
t_3
Running all the jobs together. Total time equals to the time that the longest job will take.
t'_1
t'_2
t'_3

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.

4.1.2 Value of δ in the Co-location of Various Deep Learning Models

We run a series of experiments to explore the value of δ mentioned in eq. (4.1). The models we used here are two variations of ResNet [11] models, two variation of VGG [21] models, and a more recent image classification architectures named SE-Net [12]. 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 equal to $\binom{5}{2} = 10$.

Figure 4.2 shows the value δ when 2 jobs are placed together. The X axis shows the experiment number for all the possible combinations of the models described in table 4.1. 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

Model Name	Number of Parameters
VGG-11	1
VGG-19	1
ResNet-18	1
SENet-18	1

Table 4.1: Characteristics of the models used in the experiments

Experiment Number	Models
0	vgg19, resnet50
1	se_resnet18, resnet50
2	vgg11, vgg19
3	resnet18, resnet50
4	vgg11, resnet50
5	resnet18, vgg11
6	vgg19, se_resnet18
7	vgg11, se_resnet18
8	resnet18, se_resnet18
9	resnet18, vgg19

Table 4.2: Description of the experiment numbers

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). In all of these experiments, MPS is **not** enabled. Using 512 as the batch size, leads to minor slowdown in the models that we studied.

Figures 4.3 to 4.5 show the results when the batch size is constant and we are running 2 or 3 jobs together. For small batch sizes, running 3 jobs together gives speedup compared to large batch sizes. Figure 4.3 shows 25 percent more speedup compared to running three batch sizes together. Note that the speedup values show the amount of speedup that you can gain on a **per iteration** basis.

4.2 Profiling Training Jobs

You might notice that even when co-locating a set of jobs with fixed batch sizes, some jobs are more compatible with each other leading to higher boost in the speedup, while other jobs are not compatible. We run a series of more detailed profiling to study how

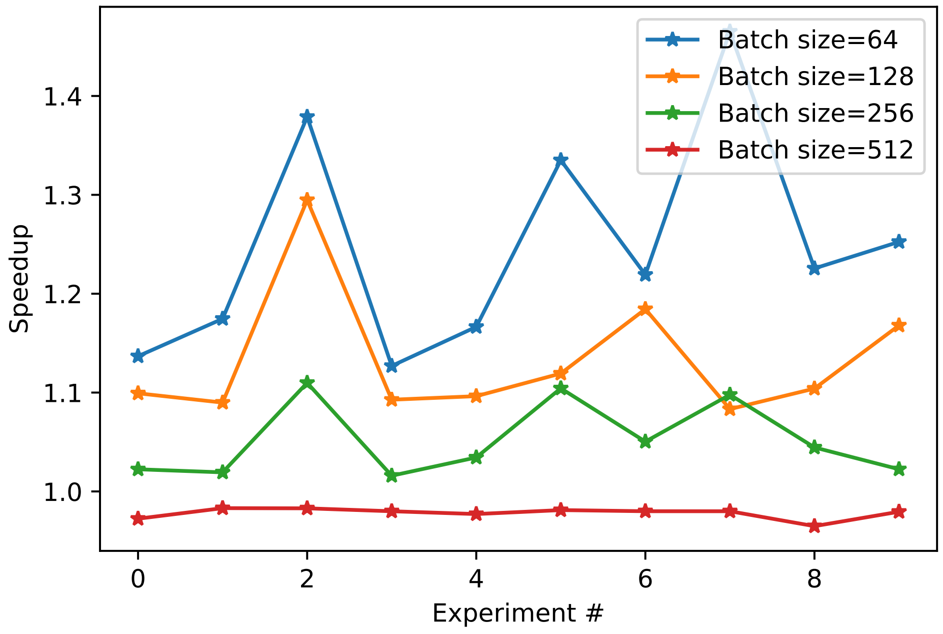

Figure 4.2: Co-locating 2 jobs together using various batch sizes. x-axis shows the experiment number described in table 4.2

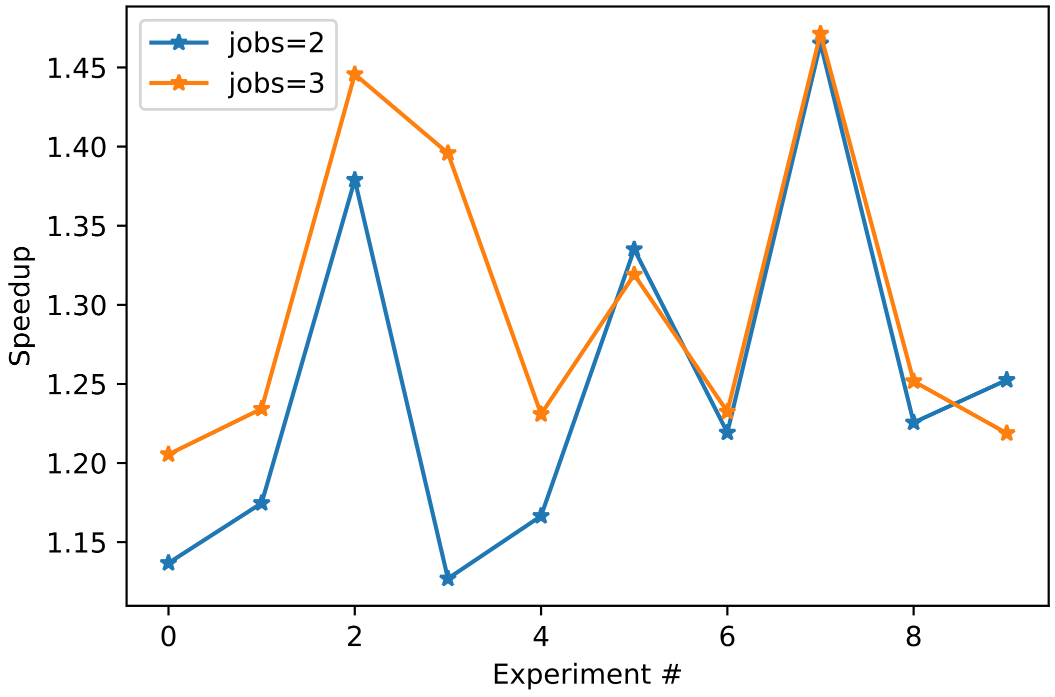

Figure 4.3: Co-location of 2 or 3 jobs together when the batch size is equal to 64. x-axis shows the experiment number described in table 4.2

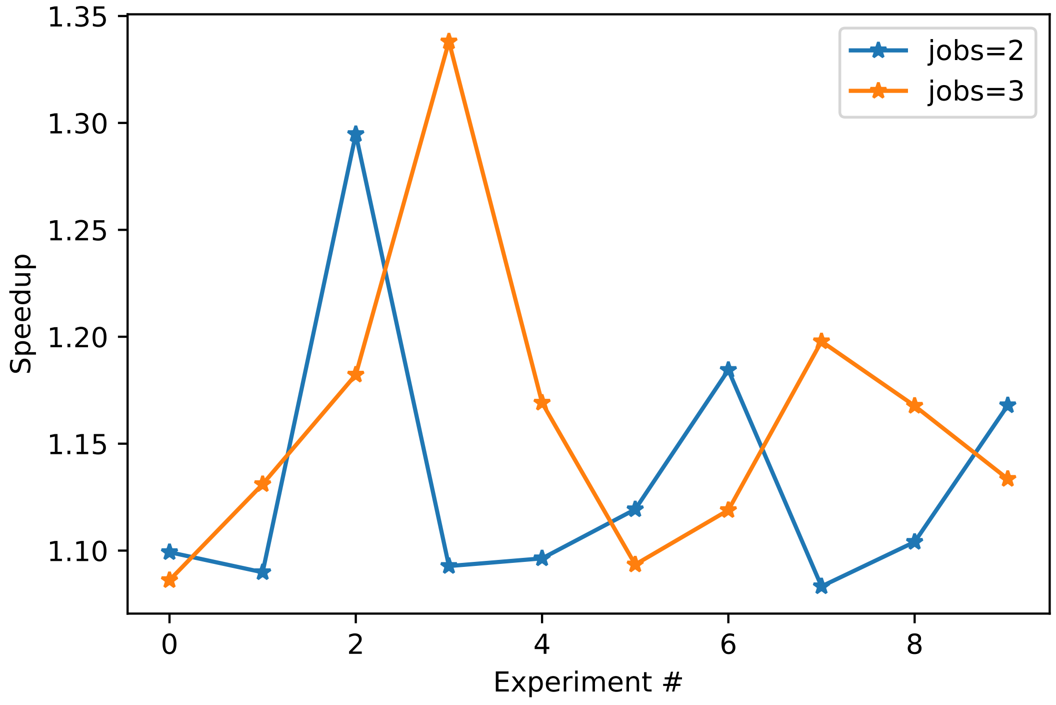

Figure 4.4: Co-location of 2 or 3 jobs together when the batch size is equal to 128. x-axis shows the experiment number described in table 4.2

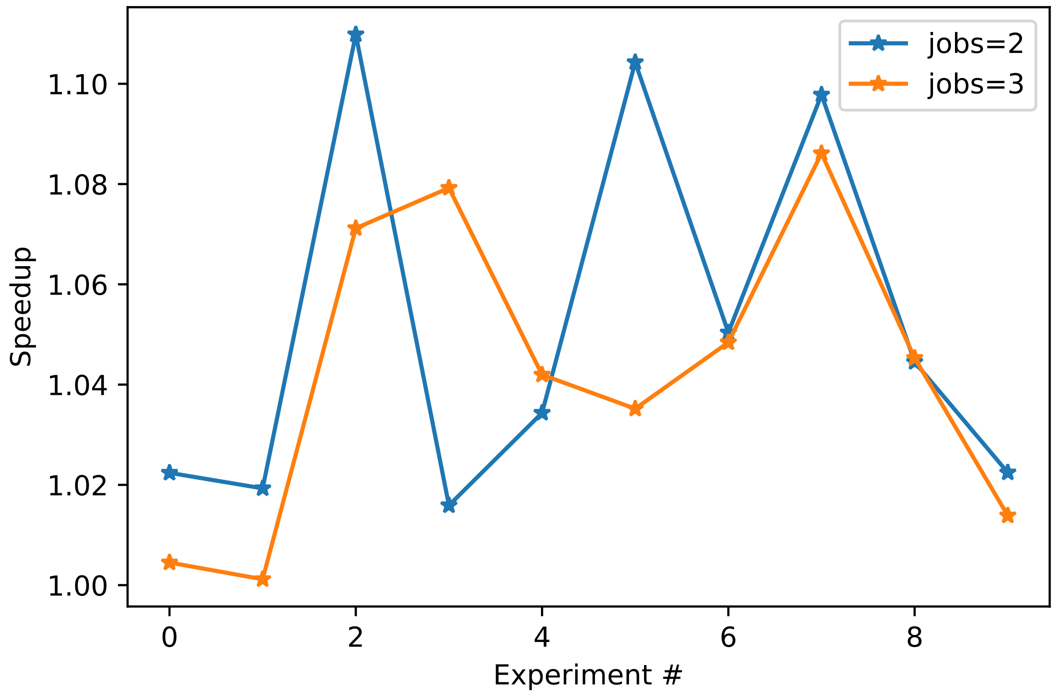

Figure 4.5: Co-location of 2 or 3 jobs together when the batch size is equal to 256. x-axis shows the experiment number described in table 4.2

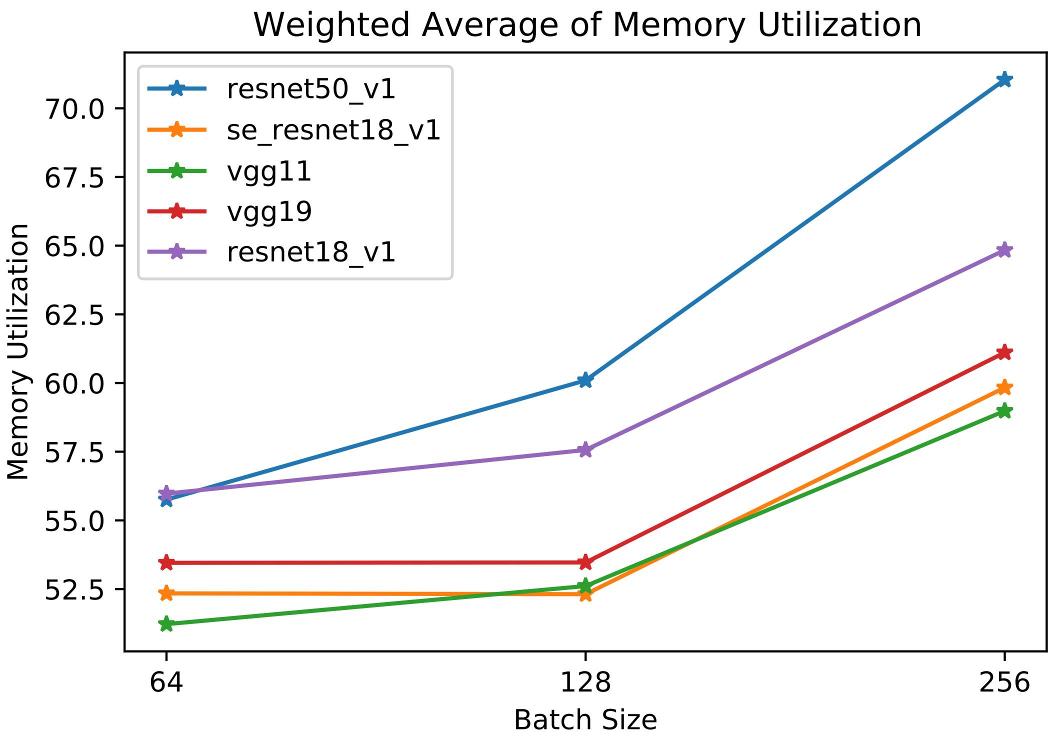

Figure 4.6: Weighted average of the memory utilization of the models using various batch sizes

individual kernels are using the GPU and finding where is the potential bottleneck. We used NVIDIA Nsight Compute [4] to perform these profiling experiments.

4.2.1 Memory Utilization

4.2.2 Compute Utilization

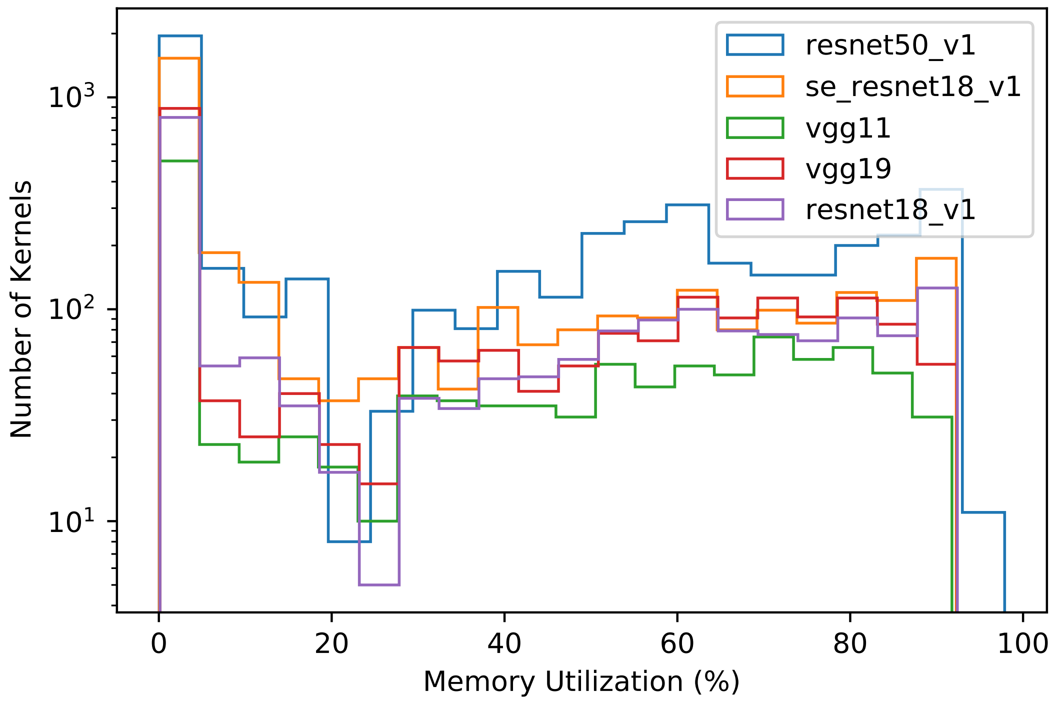

Figure 4.7: Histogram of the memory utilization of the kernels for models under study. The batch size is equal to 64. If a kernel is run multiple times, the results are not grouped together and is assumed as a separate kernel. The unit for x-axis is in percent and the y-axis is log based. This graph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

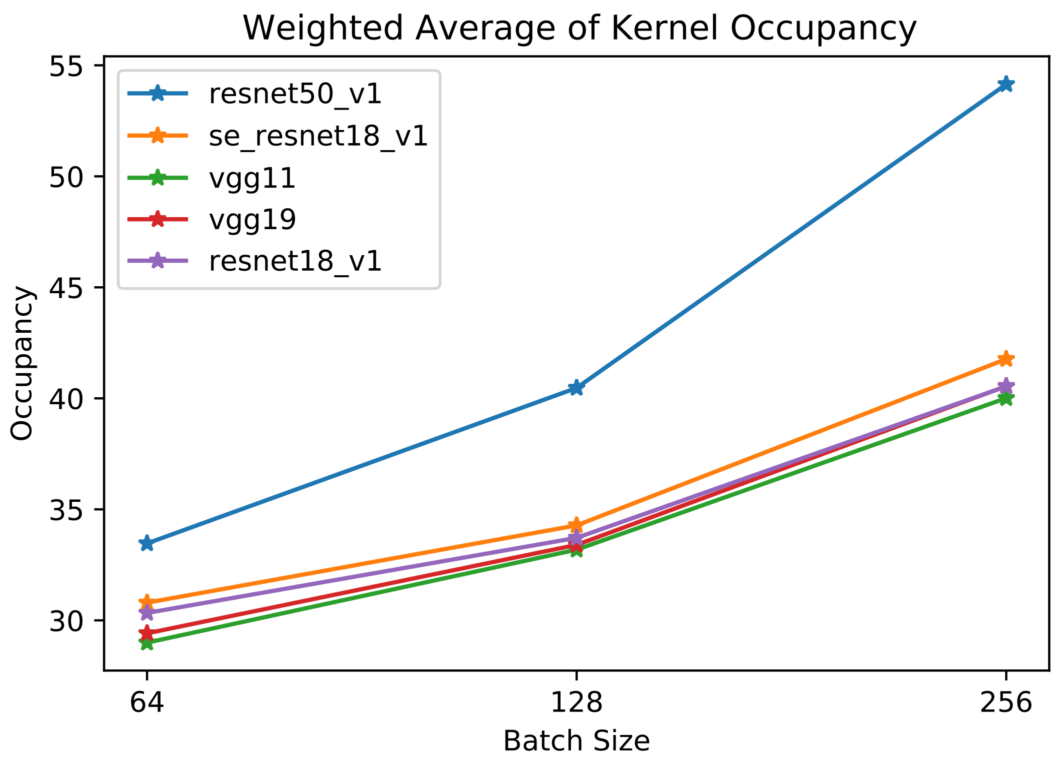

Figure 4.8: Weighted average of the kernel utilization of the models using various batch sizes

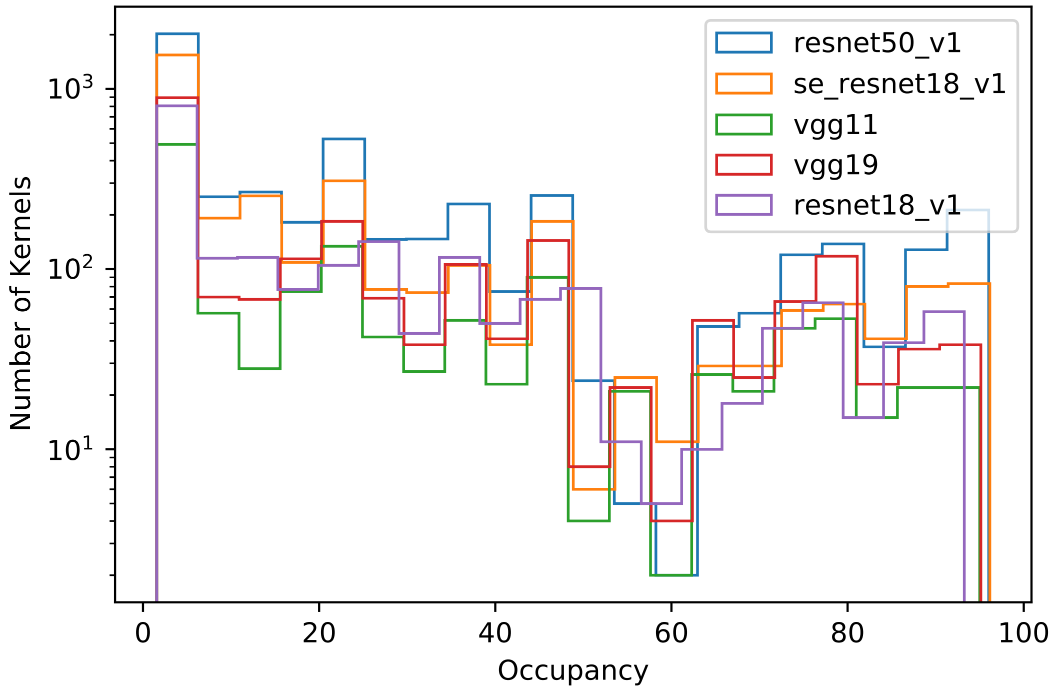

Figure 4.9: Histogram of the achieved occupancy of the kernels for models under study. The batch size is equal to 64. If a kernel is run multiple times, the results are not grouped together and is assumed as a separate kernel. The unit for x-axis is in percent and the y-axis is log based. This graph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

Chapter 5

Inference Job Co-location

Chapter 6

Related Work

6.1 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 [20] 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 [17] 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 [10] 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

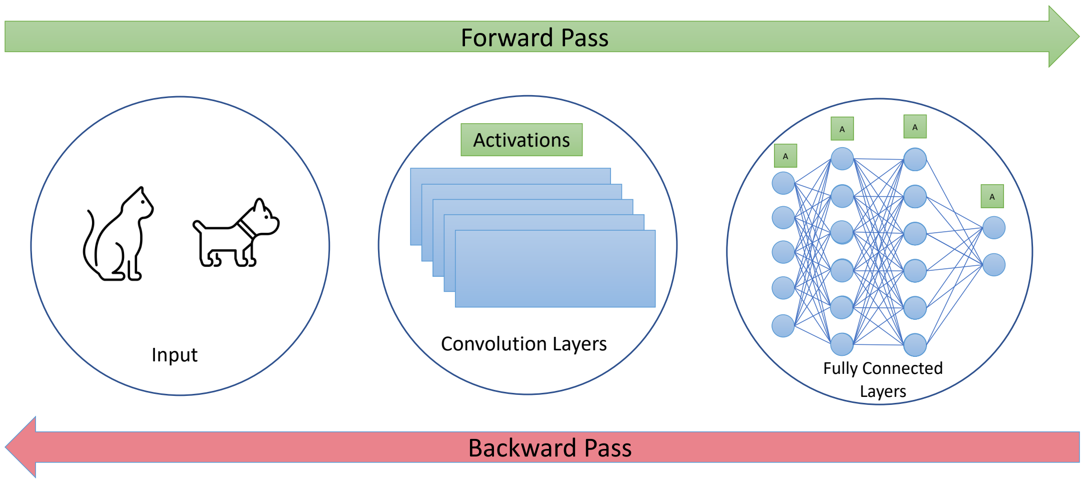

Figure 6.1: Memory Allocation in Forward and Backward Pass

hyperparameters. They do not consider the packing of multiple jobs on the same GPU.

$$w(t+1) = w(t) - \eta * \nabla Q(w(t)) \tag{6.1}$$

Gandiva [22] 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. While this is not true for PyTorch and Tensorflow frameworks, some frameworks like MXNet deallocate the memory when it is no longer needed. Figure 6.1 shows how memory allocation is performed during the training. In this figure, we are assuming the training of a convolutional neural network [16]. The weights associated with each layer is always present in the GPU. As the input traverses different layers of the neural network, it creates *activations*. Activations must be stored for each layer. The activations are required for calculating the gradients in eq. (6.1). 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 [18], this strategy is not sufficient for finding beneficial co-locations.

Chic [9] 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 [20]. However, they still didn't consider the packing of multiple jobs into a single GPU.

Gavel [18] 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.

6.2 Jobs Interference

Although interference of co-located jobs in GPU clusters has not been very explored, there is a significant body of work on the interference effect of co-located jobs in CPU clusters. Quasar [8] 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 [18] 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.

6.3 Scheduling of Data Analytics Job

Chapter 7

Conclusion and Future Work

Bibliography

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

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

[3] 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).

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

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

[6] 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.

[7] 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.

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

[9] 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.

[10] 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.

[11] 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.

[12] 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.

[13] 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.

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

[15] 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.

[16] 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.

[17] 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.

[18] 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.

[19] 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.

[20] 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.

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

[22] 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.

[23] 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.

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