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

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 reach where I am. 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 thank the lab members, and Vladi for his great 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 photography. These advancements have been made possible thanks to the specialized hardwares such as GPUs, TPUs[17], 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[6] 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.

Chapter 2 provides a background on how GPUs work and how they can be programmed. It also discusses various models that multiple tasks can be executed at the same time on the GPU.

Chapter 3 discusses the motivation for co-location of the DL jobs.

Chapter 4 provides in-depth profiling information. This profiling information help understand why co-location improves the performance in some cases while in other cases it may hinder the performance. In this chapter we study the co-location of training jobs.

Chapter 5 studies the co-location effect of inference jobs. In this chapter we study the applicability of the performance models created for co-location of training jobs for inference jobs.

Chapter 6 provides a survey of the related work in this area and how this work is different from them.

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

Chapter 2

Background

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

2.1 Graphical Processing Units (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 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[6]. 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

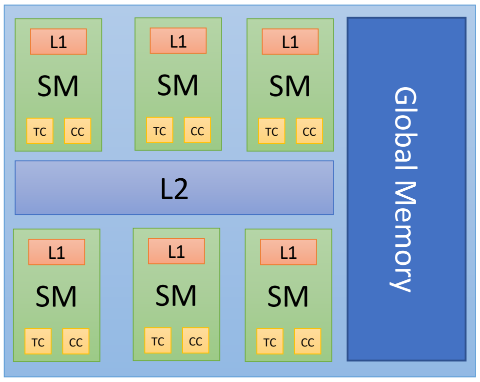


Figure 2.1: GPU Architecture

operations which are present in the Deep Learning workloads. They were introduced in the Volta[6] 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.

- **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.

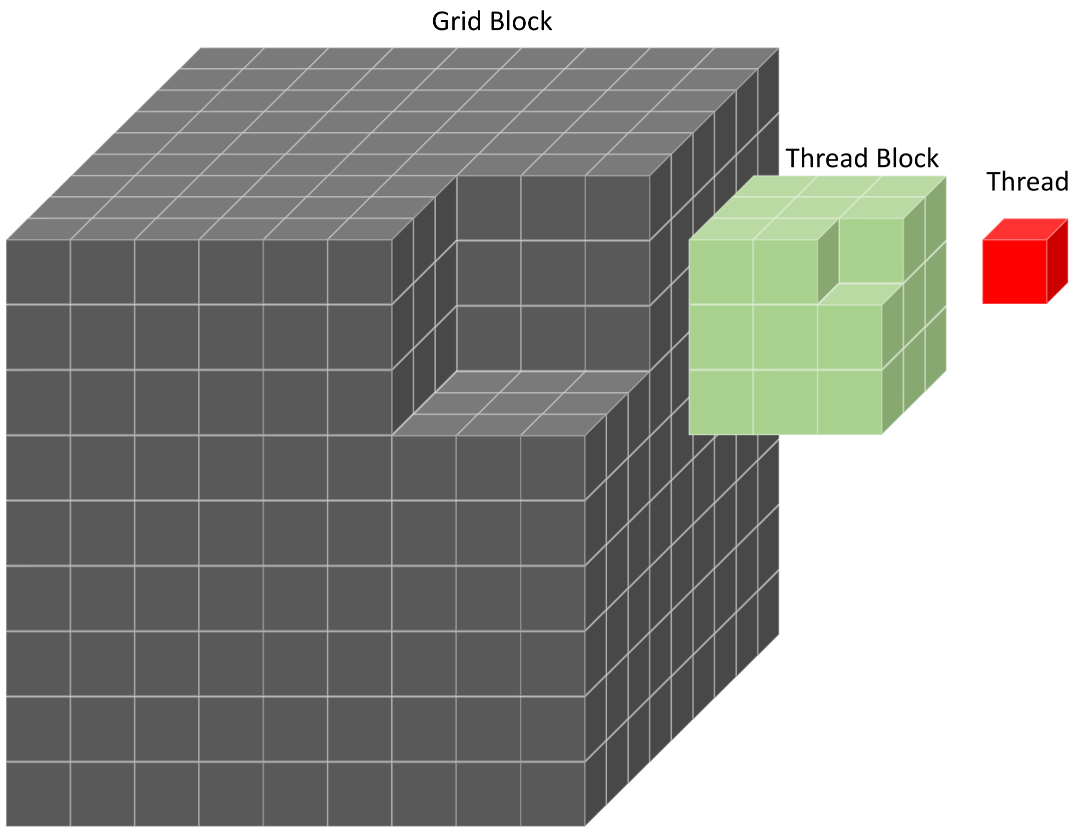


Figure 2.2: Grid Block vs Thread Block vs Thread

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*[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 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.

Life Cycle of a GPU accelerated Application

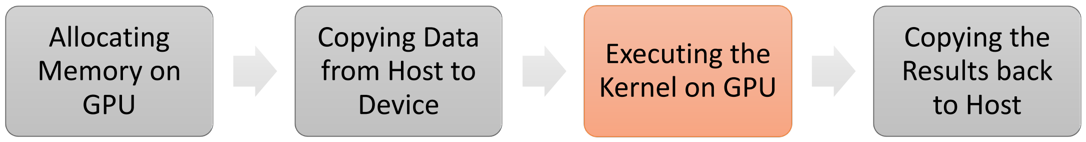

Allocating Memory on GPU
Copying Data from Host to Device
Executing the Kernel on GPU
Copying the Results back to Host

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 required to run the computation is executed. When the computation is complete, all the results are copied back into the host memory. All

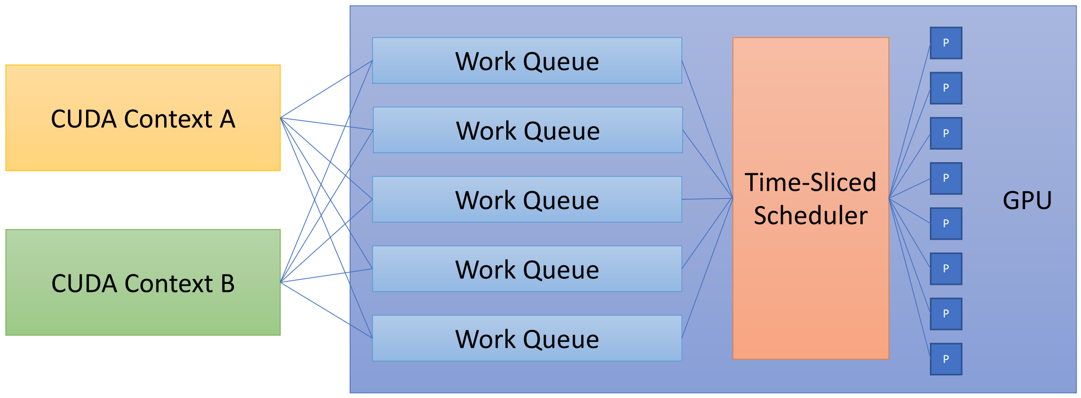


Figure 2.4: Interaction between CUDA contexts and Work Queues on a GPU

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 2.1.3.

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 MPS [2], 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.

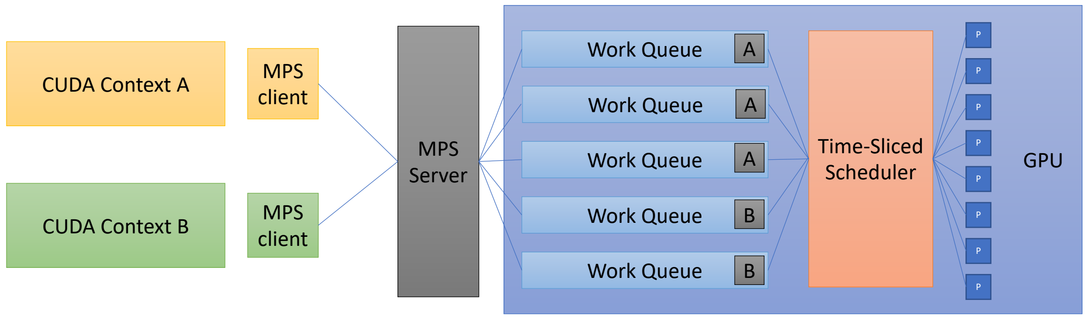


Figure 2.5: Interaction between CUDA contexts and Work Queues on a GPU

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 2.4 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.

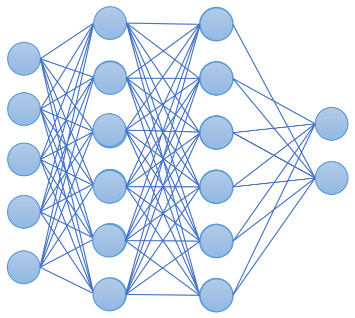


Figure 2.6: A Fully Connected Feed Forward Neural Network

2.2 Deep Learning

Deep learning [20] 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 [14]. It means that they can approximate any measurable function to any desired degree of accuracy. This feature combined with the advances in hardware enabled end-to-end learning algorithms that are able to generalize well in different domains of Natural Language Processing [8], Computer Vision [19], 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 performs 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 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 stateof-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 [7] 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 [24] 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 [16] have also taken a hybrid approach to maintain both a high developer productivity and optimized and fast execution.

2.3.3 MXNet

MXNet [9] 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 [16].

2.4 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

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 [4] 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 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 diminished returns [18, 30]. 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 [29], 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

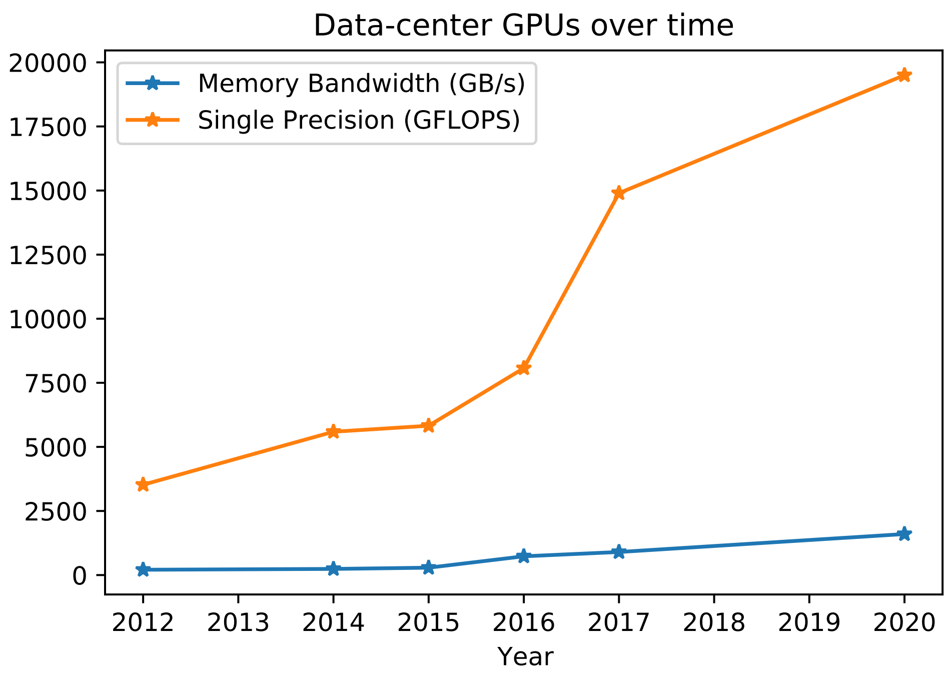


Figure 3.1: GPU Capabilities over time

about PyTorch hello world CPU task]

3.2 GPU Memory and Compute Capabilities

GPUs memory bandwidth and compute capabilities have increased by a tremendous amount over time. Figure 3.1 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.

3.3 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 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 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 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 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

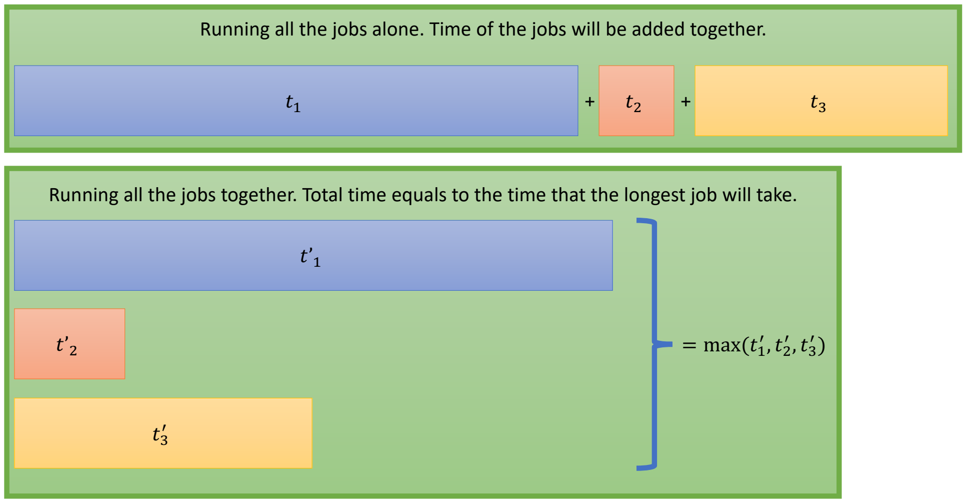

Running all the jobs alone. Time of the jobs will be added together.
t_1 + t_2 + t_3
Running all the jobs together. Total time equals to the time that the longest job will take.
t'_1
t'_2
t'_3
$= \max(t'_1, t'_2, t'_3)$

Figure 4.1: Co-location of Multiple Jobs

alone. In fig. 4.1 we are going to schedule 3 jobs and we want to find out whether co-location is suitable for them. 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 mentioned in eq. (4.1). The models we used here are two variations of ResNet [13] models, two variation of VGG [26] models, and a more recent image classification architectures named SE-Net [15]. 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 $\binom{5}{2} = 10$ when co-locating two jobs and $\binom{5}{3} = 10$

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 4.1: Description of the experiment numbers

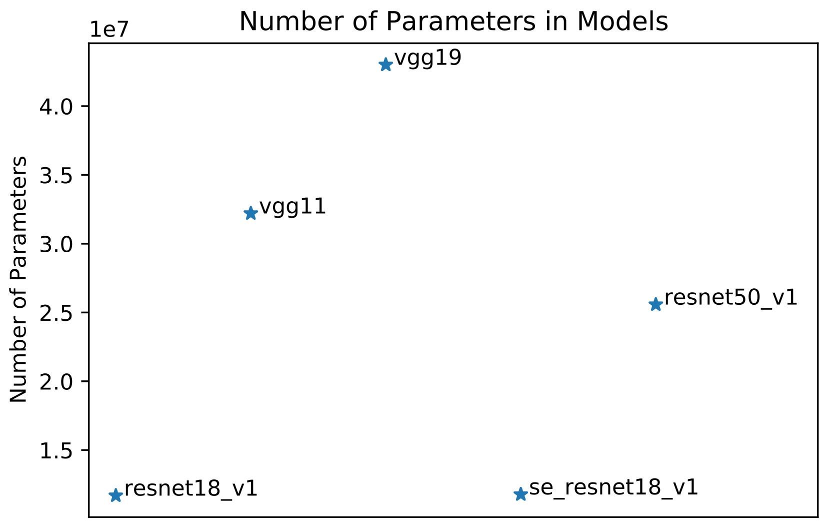


Figure 4.2: Number of Parameters in the Benchmark Models

when co-locating three jobs.

Figure 4.2 shows the number of parameters that the benchmarking 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 scale the number of parameters for each model. VGG models family require the largest number of parameters and ResNet family require fewer parameters compared to them.

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

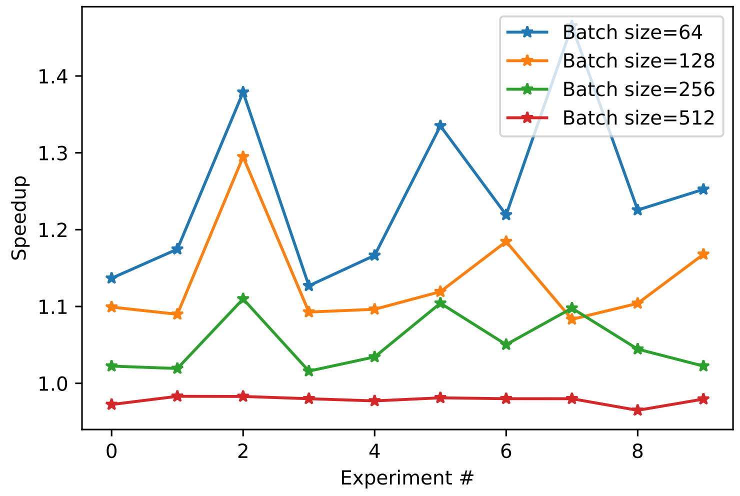


Figure 4.3: Co-locating two jobs together using various batch sizes. x-axis shows the experiment number described in table 4.1

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). Using 512 as the batch size, leads to minor slowdown in the models that we studied. 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 peeks and valleys seen in this figure.

Figure 4.4 shows the value of δ when two jobs are placed together. The difference with fig. 4.3 is that 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:

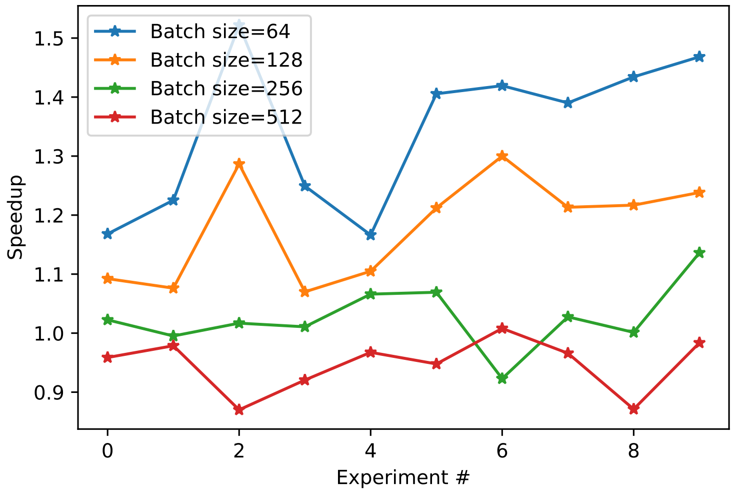
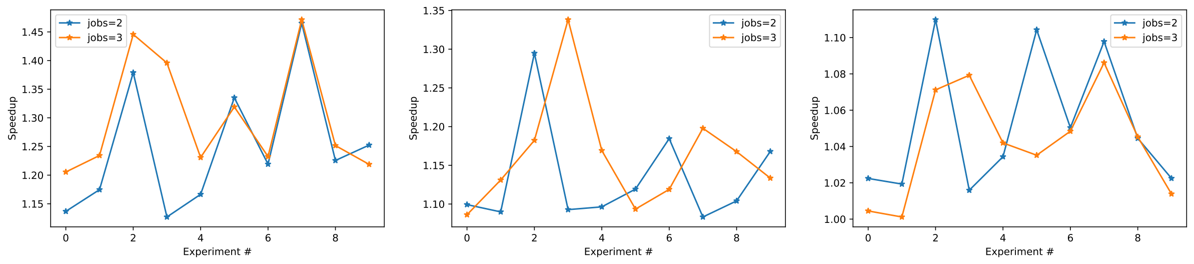
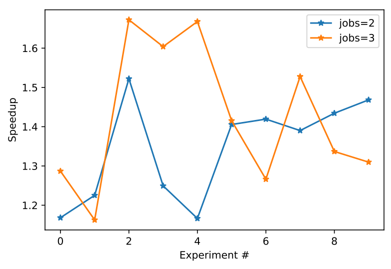


Figure 4.4: Co-locating two jobs together using various batch sizes. x-axis shows the experiment number described in table 4.1

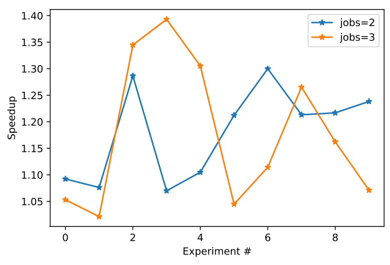


(a) Batch Size = 64 (b) Batch Size = 128 (c) Batch Size = 256

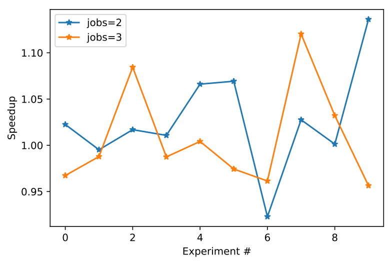
Figure 4.5: Co-location of two or three jobs together. x-axis shows the experiment number described in table 4.1. MPS is not enabled in these experiments.



(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.

- **Larger batch sizes lead to smaller speedup**. As we increase the batch size the largest speedup decreases.
- **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 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 [5]. [put a picture about job and number of kernels]

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. Histogram for p_i values is shown in fig. 4.7. Key takeaways from this figure are the following:

- ResNet-50 has a 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.

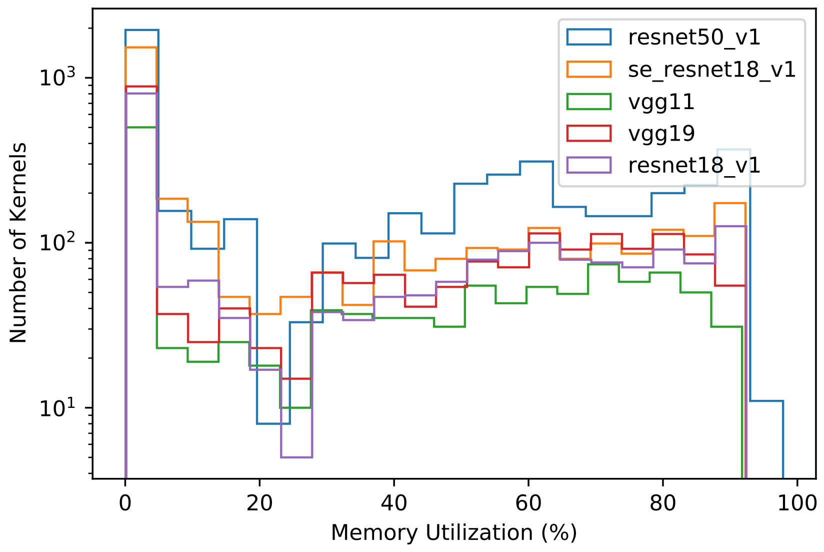


Figure 4.7: Histogram of the memory utilization of the kernels for models under study. The batch size is equal to 64. If a kernel is run multiple times, the results are not grouped together and is assumed as a separate kernel. The unit for x-axis is in percent and the y-axis is log based. This graph 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.

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

Equation (4.2) shows the formula for calculating the aggregate memory bandwidth utilization for a given job. This is a weighted average 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

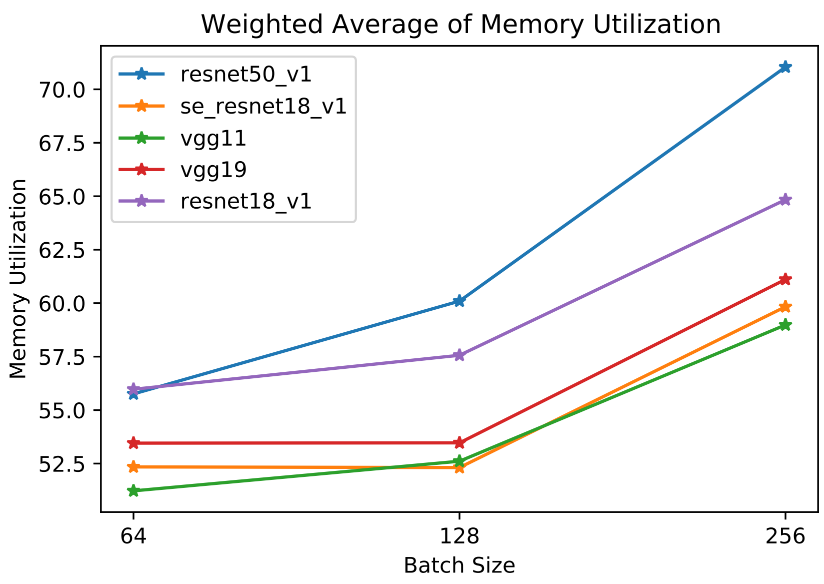


Figure 4.8: Weighted average of the memory utilization of the models using various batch sizes

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 the kernels used for training this model are highly optimized. Using this figure along with the fig. 4.4 can mostly explain why the peaks and valleys occur.

4.2.3 Compute Utilization

4.2.4 Identifying Relationship between Models and Speedup

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.

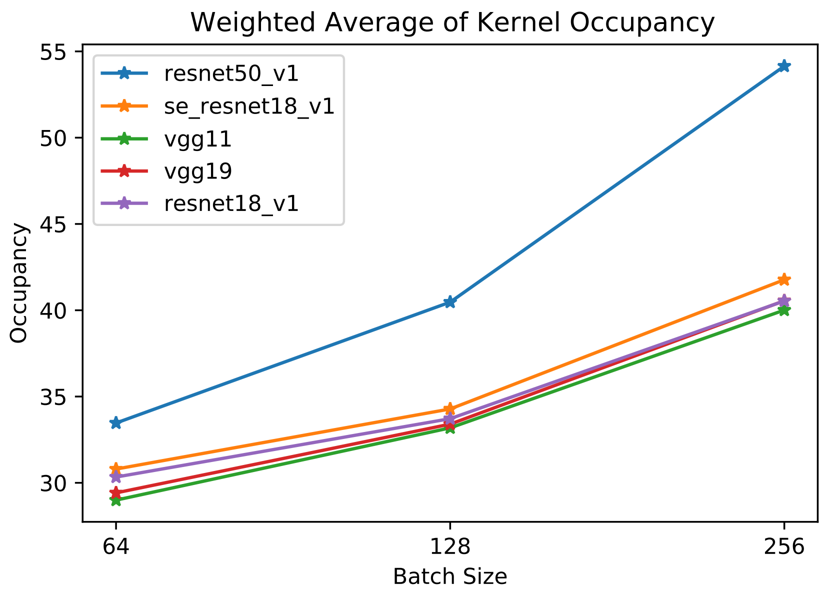


Figure 4.9: Weighted average of the kernel utilization of the models using various batch sizes

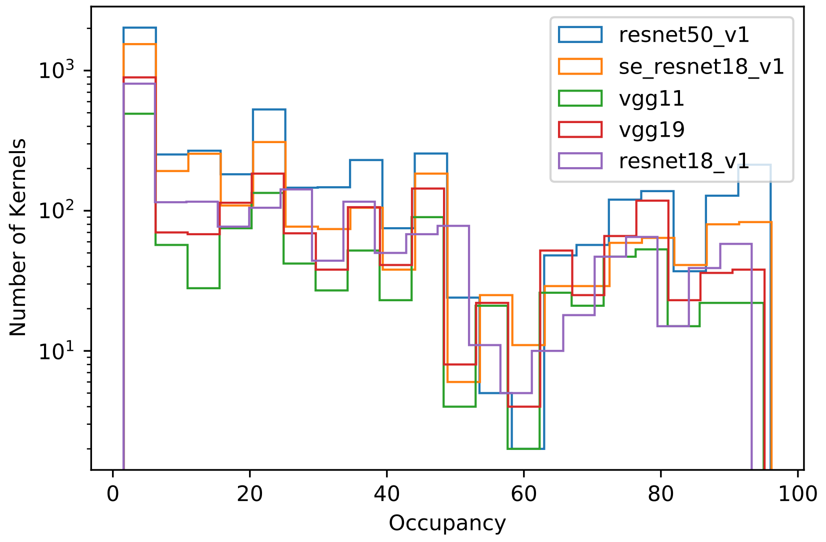


Figure 4.10: Histogram of the achieved occupancy of the kernels for models under study. The batch size is equal to 64. If a kernel is run multiple times, the results are not grouped together and is assumed as a separate kernel. The unit for x-axis is in percent and the y-axis is log based. This graph includes the results for two iterations of training.The kernels were analyzed individually without sharing the resources with any other application.

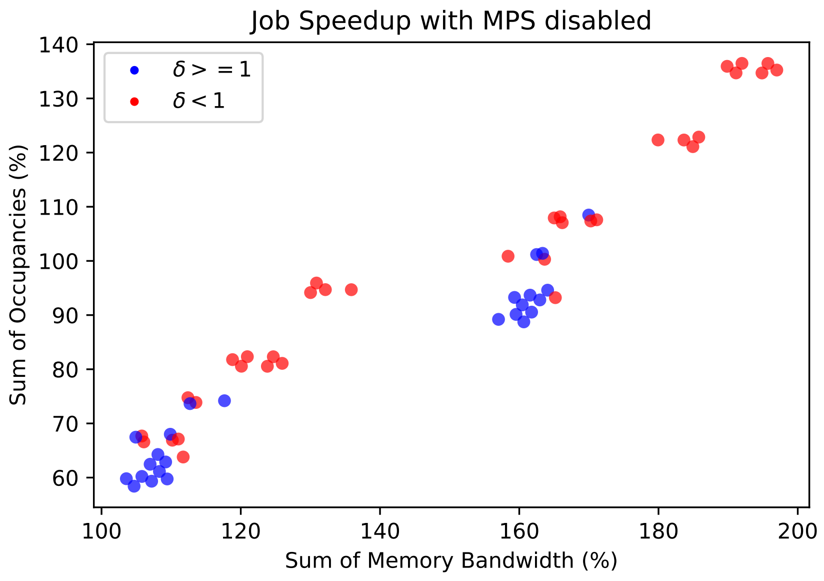


Figure 4.11

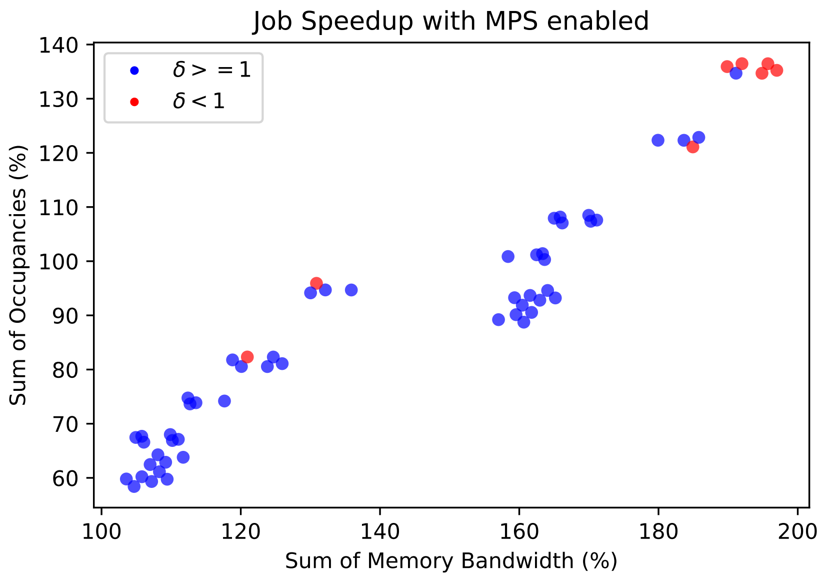


Figure 4.12

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 [25] 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 [22] 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 [12] 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

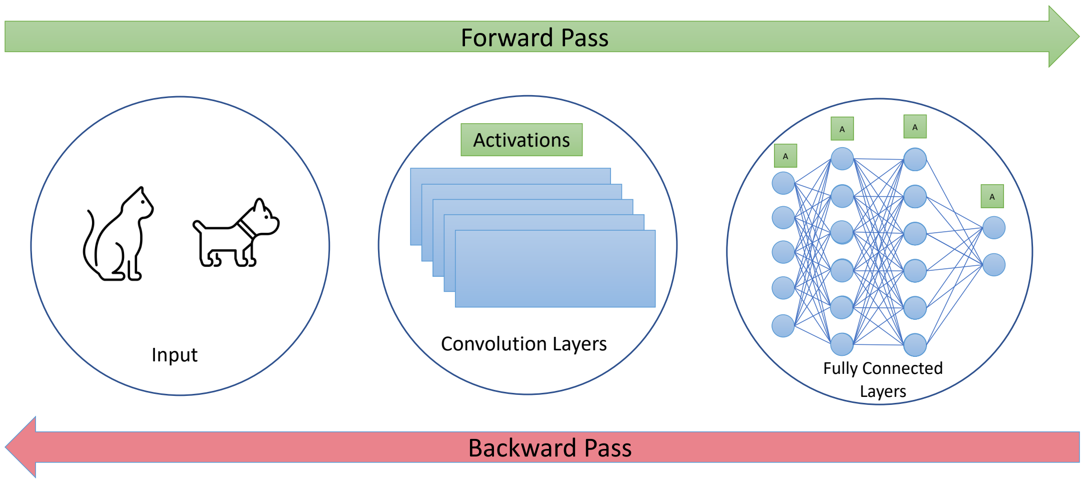


Figure 6.1: Memory Allocation in Forward and Backward Pass

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 [27] 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 [21]. 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 [23], this strategy is not sufficient for finding beneficial co-locations.

Chic [11] 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 [25]. However, they still didn’t consider the packing of multiple jobs into a single GPU.

Gavel [23] 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 Context Switching GPUs and GPU Sharing

Salus [28] provided a fast context switch mechanism by removing the need for checkpointing. Similar to Gandiva [27], they utilize different 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 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 where 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.

6.3 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 [10] 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 [23] 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.

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 simple metrics like weighted memory bandwidth utilization and weighted occupancy can accurately predict the speedup. 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 4.10, 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 a single V100 gpu and thus we could not verify whether these results hold for other Volta and Ampere GPU architectures too or not. Also, MPS has not yet been released for the recent Ampere [3] 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.

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-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).

[4] 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).

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

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

[7] 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.

[8] 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.

[9] 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.

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

[11] 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.

[12] 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.

[13] 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.

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

[15] 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.

[16] 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.

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

[18] 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.

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

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

[21] 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.

[22] 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.

[23] 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.

[24] 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.

[25] 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.

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

[27] 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.

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

[29] 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.

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