DIGITAL ACCESS TO SCHOLARSHIP AT HARVARD
DASH.HARVARD.EDU

HARVARD LIBRARY
Office for Scholarly Communication

Minimizing NUMA Effects on Machine Learning Workloads in Virtualized Environments

Citation
Broestl, Sean. 2021. Minimizing NUMA Effects on Machine Learning Workloads in Virtualized Environments. Master's thesis, Harvard University Division of Continuing Education.

Permanent link
https://nrs.harvard.edu/URN-3:HUL.INSTREPOS:37367717

Terms of Use
This article was downloaded from Harvard University's DASH repository, and is made available under the terms and conditions applicable to Other Posted Material, as set forth at http://nrs.harvard.edu/urn-3:HUL.InstRepos:dash.current.terms-of-use#LAA

Share Your Story

The Harvard community has made this article openly available.
Please share how this access benefits you. Submit a story .

Accessibility

Minimizing NUMA Effects on Machine Learning Workloads in Virtualized Environments

Sean Broestl

A Thesis in the Field of Software Engineering

for the Degree of Master of Liberal Arts in Extension Studies

Harvard University

May 2021

Copyright © 2021 Sean Broestl

Abstract

This thesis is an investigation into the performance ramifications of making specialized component reservations for machine learning workloads in virtualized environments. The reliance of machine learning on floating-point operations makes graphics processing units an important part of processing these workloads quickly. While virtualization is one of the most widely-used consolidation techniques used in data centers of all sizes, compatibility issues between graphics processing units and virtualization have slowed the adoption of virtualization for machine learning workloads.

To that end, this paper discusses the motivations and history behind virtualization and the application-specific acceleration devices used and how they are applied to machine learning on various public and private computing platforms. This is followed by a presentation of an experimental framework for testing the impact of controlling for non-uniform memory access when running machine learning workloads.

Using this framework, a series of experiments were performed and documented in this thesis that test multiple placement configurations for graphics processing units in a virtualized system and how throughput of data from the host system to the device was affected. Current virtualization platforms offer recommendations to use these settings, but do not talk about the specific impacts of implementing them.

Based on the results of the experiments, configuration parameters and placement recommendations are presented along with information about how these settings can help optimize the machine learning pipeline and the potential pitfalls to their use.

Acknowledgements

I offer the most sincere thanks to my thesis director Dr. James L. Frankel, whose guidance and feedback helped mold this thesis into a much greater work than I imagined it could be at the outset. Over the past nine months our regular meetings were a welcome source of both normalcy and laughter during a very difficult year for everyone. Of course, that step would not be possible without the efforts of my research advisor Dr. Hongming Wang and Dr. Sylvain Jaume, whose feedback, guidance, and persistence were instrumental in getting my idea transformed into a real proposal, and for that I am very grateful.

I also wish to thank my family, whose contributions of encouragement, assurance, and advice were oftentimes just the little nudge I needed to get through a bout of writer's block. Finally, thanks goes out to my colleagues at Brown University, who in addition to their affirmations, were able to support me with a loan of the computing resources necessary to complete this thesis.

Contents

Table of Contents **vi**

List of Figures **viii**

List of Tables **x**

Chapter I. Introduction **1**

1.1 Problem Statement 3

1.2 Prior Work 5

Chapter II. Requirements **8**

2.1 High-level Requirements 8

2.1.1 GPU 9

2.1.2 CPU and Hardware Platform 10

2.1.3 Software Requirements 12

2.2 System Components 13

2.2.1 Orchestration 13

2.2.2 Hypervisor 14

2.2.3 GPU-backed VM instances 15

Chapter III. Implementation **18**

3.1 Non-Uniform Memory Access and Multi-Processor Design 20

3.2 Test Environment Details 23

3.3 IOMMU 25

3.4 Machine Learning Workflow Sample 26

3.4.1 Code and Language Choices 27

3.5 Testing PCIe Performance 29

3.5.1 PCIe Bandwidth 34

3.5.2 Multiple GPU scenarios 35

3.6 Simulating non-ML workloads 36

Chapter IV. System Documentation **38**

4.1 Basic VM instance 38

4.2 GPU-backed VM instances 40

4.3 Non-GPU Load Generation 41

Chapter V. Experimental Results **42**

5.1 Baseline Results 43

5.2 Workload spanning multiple NUMA nodes 46

5.3 ML Workload on Mixed-use System (CPU-bound Load) 47

5.4 ML Workload on Mixed-use System (CPU and Memory-bound Load) 49

5.5 ML Workload on Mixed-use System (Multiple NUMA-aware VMs) . . 51

5.6 ML Workload vs ML Workload 52

5.6.1 Experiment Setup 52

5.6.2 Results 54

5.6.3 Results with two GPU-backed VM instances 55

5.6.4 Results with four GPU-backed VM instances 57

Chapter VI. Summary and Conclusions **64**
6.1 Knowledge Gained 64
6.2 Configuration Recommendations 66
6.3 Limitations and Known Issues 67
6.4 Lessons Learned 68
6.5 Future Considerations 70

References **73**

Appendix A. Glossary **77**

Appendix B. Configuration Settings **82**
B.1 VM Configuration 82
B.2 Hypervisor Configuration 82
B.3 Hardware Configuration 82

List of Figures

Figure 1 GPU VM instance with 8 CPUs 22
Figure 2 GPU VM instance with 16 CPUs, host has 40 cores (80 logical CPUs) 22
Figure 3 GPU VM instance with 24 CPUs, host has 40 cores (80 logical CPUs) 22
Figure 4 GPU VM instance with 16 CPUs and 256 GB of RAM 23
Figure 5 GPU VM instance with 24 CPUs and 256 GB of RAM 23
Figure 6 Notable python libraries used in experiment code 27
Figure 7 Main method spawning child processes 30
Figure 8 Python method used to move data from system memory to GPU 31
Figure 9 Python method for moving data to main memory 33
Figure 10 Load Generator Workload Types 37
Figure 11 Required packages for benchmark VM instance 39
Figure 12 Commands used to install CUDA 11 on Ubuntu 20.04 40
Figure 13 Baseline data transfer rate for single mode 44
Figure 14 Baseline time elapsed for single mode 45

Figure 15 Data transfer rate for single mode when VM instance sees multiple NUMA nodes 47
Figure 16 Time elapsed for single mode when VM instance sees multiple NUMA nodes 48
Figure 17 Data transfer rate sampling when loading CPU and RAM . . . 50
Figure 18 Baseline data transfer rate and time elapsed for multiple ML workloads (single mode) 55
Figure 19 Baseline data transfer rate and time elapsed for multiple ML workloads (batch mode) 56
Figure 20 Transfer rate and time elapsed for 4 GPU-backed ML workloads (single mode), scenario 4 58
Figure 21 Transfer rate and time elapsed for 4 GPU-backed ML workloads (single mode), scenario 5 59
Figure 22 Transfer rate and time elapsed for 4 GPU-backed ML workloads (batch mode), scenario 5 59
Figure 23 Transfer rate and time elapsed for 4 GPU-backed ML workloads (single mode), scenario 6 61
Figure 24 Transfer rate and time elapsed for 4 GPU-backed ML workloads (batch mode), scenario 6 61
Figure 25 Transfer rate and time elapsed for 4 GPU-backed ML workloads (single mode), scenario 7 62
Figure 26 Transfer rate and time elapsed for 4 GPU-backed ML workloads (batch mode), scenario 7 63

List of Tables

Table 1 Version constraints for automation components 14

Table 2 Version constraints for hypervisor 15

Table 3 Hardware configuration for test systems 24

Table 4 Command line switches for the benchmark application 29

Table 5 ML vs. ML workload experiment scenarios 53

Chapter I.

Introduction

Due to the increased utilization of machine learning in applications and the strong demands placed upon computing systems, machine learning is a driving force of innovation of computing components. Significant research goes into ways to optimize the different layers of these workloads in terms of speed, cost, algorithmic complexity, and data structure size. Machine learning, or ML for short, is a field of computing broadly referring to the use of neural network or similar algorithms that can be trained using a small initial dataset, then classify and identify data that has not been previously seen. ML workloads can utilize many or all components of the computing system, but one of the most important pieces of that system today is the electronic pathways used to move data across the system bus to a Graphics Processing Unit, or GPU, where floating point operations can be performed with a level of parallelism much greater than would be available on the CPU (Keckler et al., 2011).

This thesis will present a set of recommendations and configuration parameters that help optimize ML workloads that will be running on servers employing Virtualization. These configuration parameters will be applicable at multiple levels, both in how VM instances are sized and balanced on Hypervisor systems and how memory can be best allocated at the guest OS level.

The system bus of the x86 architecture has always been important from the

standpoint of moving data from the CPU to its various peripheral devices like storage and networking controllers. For most of its history, the Peripheral Component Interconnect Express, referred to as PCI and PCI Express bus have done a fine job of meeting the I/O needs of system components. It is the responsibility of processor developers to implement the PCI Express standard, which calls for a number of electrical lanes on the motherboard dedicated to the interface. Modern computing peripherals using the PCI Express, PCIe for short, have embraced the potential speed of this interface, bringing to market powerful high-speed devices like 40 Gigabit Ethernet adapters, non-volatile memory for fast storage, and Graphics Processing Units. PCIe slots are divided up among the limited number of electrical lanes provided by the computer system, using as few as one lane, or more commonly four, eight, or sixteen lanes. This topic is discussed in detail in Chapter 3.5.1. Today's GPUs most commonly use 16x slots in order to quickly move data to the high-performance processors and large amounts of memory located on the cards.

These cards have become so important to ML work that cloud providers like Google are taking steps to develop their own accelerators for this type of work in order to compete with GPU developers on price and performance (Jouppi et al., 2017). Cloud providers are also some of the heaviest users of virtualization technology. Virtualization is a technique in computing where a complete hardware computing system can be abstracted in software and as a result, many of these systems that were previously using an entire piece of physical hardware can be colocated on a single computer. The upside of this is that it reduces the physical footprint required to run massive computing operations while also enabling more flexible usage and payment models for only those resources that are needed for an application to run. If there is a downside, it is that all these virtual machines, or VMs, must share the limited physical resources of the computer on which they reside. The intersection of

GPU use and virtualization is worth exploring in order to analyze the effects of the heavy PCIe bus consumption of GPU when applied to a shared computing model like Virtualization.

We will briefly cover some current designs in machine learning system systems as well as trends and challenges relevant to data center and cloud computing operators. This overview will present some commonly understood definitions for various jargon and official terms in the realm of system design. This introduction will serve as a way to set context for why optimizing execution environment is important to machine learning and the benefits that can be derived from doing so. We will also discuss the state of research and some prior work in this area.

Following the introductory chapter, this thesis will present some requirements relevant to both generically running ML workloads and to implement the environment used to prove the thesis. These details will include a summary of a common model used in processing of data for machine learning, as well as discussion on requirements in terms of hardware and software for that work. Before moving onto results, the implementation chapter will cover details on how experiments were run and the parameters used for each scenario.

The conclusion chapter will present sample VM configurations and code considerations to use when building ML workflows. Finally, the thesis will close with some brief discussion of some future considerations which may alter the importance of GPUs for this work in the future.

1.1 Problem Statement

Not all workloads are so large as to need exclusive access to dedicated hardware and four or more GPUs. Many models and datasets exist that are smaller scale and still return useful results. For example, there are methods for training one of the

most popular data sets, ImageNet, using down sampled versions of the images that produced results on par with algorithms using the full-resolution images (Chrabaszcz et al., 2017). Chrabaszcz et al.'s approach to working with a data set at a smaller scale gives support to the idea that more detail in data and training accuracy are not strictly tied to each other. Knowing this, it becomes possible to propose that the newest, fastest multiple GPU setups are not necessarily required for acceptable ML training. Another point worth considering is that GPUs are becoming popular to help speed up inference workloads on already trained models. One set of researchers recently presented research on how they used Nvidia V100 GPUs to speed up inference in COVID-19 infection simulations (Kulkarni et al., 2020). The conclusion to draw given these points is that the use cases for GPU are expanding into smaller use cases even as powerful GPUs are becoming more common. These smaller workloads are easily able to be run on virtualization platforms with a single GPU backing them, and it's worth exploring how to optimize those as well.

However, the solution isn't as simple as heading to the console of a favorite cloud provider, provisioning a VM, and training our new model. Most current configurations for GPUs on VM instances use a virtualization technique called pass-through to grant exclusive GPU access to a VM. Once pass-through is involved, it complicates the utilization of resources for the hypervisor.

The first problem is that management of a hardware component is no longer the exclusive domain of the hypervisor, and this has security, performance, and resource sharing implications. The second issue is that of Non-Uniform Memory Access, known commonly as NUMA. NUMA refers to the amount of time needed to access memory that is non-local to the processor where an instruction is being executed. Each CPU in a multi-processor system gets their own set of PCIe lanes and system RAM, and when you use pass-through the VM instance employing pass-through be-

comes strongly associated with one of those processors, the processor's PCIe lanes, and the processor's RAM. NUMA therefore has the potential to harm performance for the VM instance using pass-through because every memory-related operation could end up taking longer as the VM has to move data from one processor's memory to a device in another NUMA domain.

1.2 Prior Work

In their paper about OpenCL and CUDA performance on four different hypervisors using pass-through, Walters, et al. acknowledged that virtualization effects on NUMA is "interesting in its own right" (Walters et al., 2014). Briefly, CUDA is a programming language which uses C-style convention and is designed to make it easier to harness the power of a GPU for floating point arithmetic. Their deep dive into GPU pass-through provides sufficient evidence that it can be reliably expected to get performance approaching the levels of an OS installed on bare metal rather than a hypervisor.

Of note, in the time since Walters et al. paper was published, the hypervisor landscape has shifted quite a bit. Their research involved performance comparisons between the major hypervisors of the era — ESXi, KVM, and Xen. VMware's ESXi is still a popular choice for private data centers. Xen has faded somewhat in use — Citrix remains the primary commercial vendor supporting Xen in their Citrix Hypervisor product. KVM has a tremendous amount of momentum in 2020, with development support from Google, and an active fork used by Amazon Web Services for their Nitro hypervisor platform, (Honig & Porter, 2017), (AWS, 2020).

AWS announced the change to KVM in 2018, and Google Cloud Platform came to market with KVM as their hypervisor of choice. It can be difficult to determine quite where ESXi stands in terms of installed base and usage. As a publicly-traded

company, VMware sales are a matter of public record and from all appearances, ESXi is a well-regarded hypervisor for private datacenters. Its high regard, however, has done little to improve ESXi's standing in the rankings of Hypervisors used for public cloud companies. For what it's worth, KVM is free and open-source, where as ESXi is licensed commercial software.

Denneman makes the case that in world with high-speed peripherals like GPU, PCIe devices are the primary units of data processing, (Denneman, 2020). Denneman rightly points out that in an x86 architecture, breaking NUMA locality for a VM instance means that data must traverse a central interconnect between processor/memory domains when trying to access a PCIe device owned by another processor. In the case of a system that is heavily utilized, this could become a limiting factor on performance, especially when dealing with many gigabytes or even terabytes of data. Fortunately, the bandwidth available to this central interconnect is greater than any single PCIe slot, so it may not be an issue to utilize it. The potential parallelism gains by utilizing another GPU may outweigh the costs of moving this data across the central interconnect.

Nvidia has introduced multiple solutions for virtualizing GPU access, the first being Nvidia Grid in 2013. Grid is a set of kernel modules for hypervisors and a specialized fork of Nvidia's display drivers that allowed for fractional access to a GPU, not unlike virtualizing a CPU. Cards designed for Grid, such as the Tesla M60, Tesla V100, and Tesla A100 contain multiple GPU cores on a single card, thereby routing around some issues related to low PCIe lane count in some previous CPU designs. While a promising and developing technology, much of its evolution has been focused on 3D accelerated application and desktop delivery rather than ML workload support.

A technology startup called Bitfusion (acquired by VMware in 2018) created

a GPU virtualization layer that allows for a user to split GPUs into logical blocks similarly to how CPU time can be split by a hypervisor. Bitfusion's solution was unique in the market because it not only allowed the user to slice up a GPU, but it allowed these devices to be accessed over an Ethernet transport. In other words, despite the advertised need of GPU hardware for the highest PCIe speeds available, it is conceivable that bandwidth needs can be met using a slower transport than PCIe.

Chapter II.

Requirements

This chapter specifies the requirements of the system.

2.1 High-level Requirements

This paper is primarily concerned with producing recommendations that help optimize workload placement on computing hardware that uses hypervisors. Previously, when combining older generations of processors with a hypervisor, the type of throughput and CPU utilization pattern desired for high-performance computing meant accepting trade-offs. Either the HPC-style workload wouldn't run well while general-purpose workloads did, or vice versa. Recent advances in processors from both Intel and Advanced Micro Devices (AMD) along with improvements in hypervisor software have made the general-purpose workloads and high-performance computing-style workloads much better tenants when sharing a piece of host hardware. As such, when designing and building virtualization hosts, there are several minimums that should be taken into account and are explained in more detail in this chapter.

2.1.1 GPU

Chapter 1.2 discussed some options for GPU virtualization or abstraction such as Nvidia Grid and VMware's Bitfusion product. These products either add a proprietary layer or shift I/O to a different part of the computer. Additionally, they add additional cost and setup complexity to a virtualized ML environment that is unnecessary. As a result, this thesis presents its solution using a pass-through configuration for the GPU due to its low cost and common deployment in the industry. This allows the VM to use the theoretical maximum performance of that specific hardware at the expense of removing the ability of another VM to be able to share that resource. Pass-through mode is a configuration option offered by all current hypervisors and needs to be employed to maximize ML workload performance. This is the method that is employed by most major cloud vendors, and as such, has the most real-world applicability.

When generating the conclusions of this paper, Nvidia GPUs were employed, specifically Nvidia Tesla M60 GPUs. The M60 is a GPU built specifically for use in hypervisor environments, as it colocates two distinct GPU units on a single PCIe slot in order to more densely pack GPUs into a single Hypervisor server. Nvidia publishes software and drivers that allow an administrator to fractionally divide the GPU core along various memory boundaries, such as 512 MB, 1 GB, 4 GB, and so on. This feature was expanded upon in later cards such as the V100, which employ a technique called multi-instance GPU (MIG), which allows an administrator to fractionally divide the GPU core up to seven times and present each as a distinct GPU to a VM, container, or application instance. The card can also be addressed as a single-core GPU card with 8 GB of RAM and passed through to a VM, which is the configuration used here. Due to limitations with this line of GPU, it is not possible

to test the fractional usage features noted above as CUDA can only function with the card in a full 8 GB configuration. This limitation is due to the Tesla's use of CUDA cores rather than the Tensor cores used on more recent cards, and this limitation exists regardless of whether a pass-through or virtual GPU solution is employed.

Fortunately, GPU choice is more or less inconsequential to the methods used to reach the paper's conclusions. While library and hardware choice play a large part in the overall performance of ML workloads, here the focus is on verifying the effects of virtualization and NUMA locality when moving data across the PCIe bus. As such, there is less concern about what happens with the data once it is on the GPU and more about how fast data can be moved to the GPU. Nvidia's CUDA libraries expose a simple interface for copying data across the PCI bus from host memory to device memory. While CUDA is used here, OpenCL exposes a similar set of functionality, as does Google's TensorFlow. This mean that the methods described here could also be applied to AMD GPUs or ASICs.

2.1.2 CPU and Hardware Platform

While virtualization can be used in a variety of system CPU configurations, this paper is concerned primarily with applications for ML workload optimization in data center scenarios. Using advertised system sizes from major cloud providers such as AWS and Amazon as a guide it can be observed that the overwhelming majority of systems advertised for GPU utilize multi-socket systems with tens if not hundreds of cores. Large-scale systems are also an area of interest for GPU developers and cloud providers. For example, Nvidia recently entered the custom computer market with a line of data center systems packed with GPUs to do large-scale floating-point calculations for the data center with their DGX line of computer systems. Competitive to that approach, major cloud providers like Google and Amazon are

developing their own application-specific integrated circuits to accelerate floating-point operations in their products.

Therefore, the recommendations of this paper are tailored around the use of multi-socket systems that employ two or more CPU sockets, high CPU core counts, and multiple PCIe interfaces. The conclusions presented here assume that processors from generations released in 2016 to present are used, such as Intel's Sandy Bridge or AMD Zen series. These CPU revisions guarantee the presence of features that are required for good performance on ML workloads and are enumerated here. First, PCIe 3.0, commonly available since 2010, is certain to be present on these platforms. PCIe 4.0 improves upon the standard, doubling the transfer rate of its predecessor, but the standard was only ratified in 2017. Peripherals using the 4.0 standard only became available in 2020. Second, these systems feature a high-speed bus interconnect between the processors which facilitates higher connectivity speeds between component devices on the PCIe bus and for processes that must cross NUMA domains. Intel's Ultra Path Interconnect and AMD's Infinity Fabric are examples of these interconnects, which has been available under various names since 2003. The speed of these devices is a contributor to the throughput that can be achieved for a workload, especially once system memory sizes start to get large enough to cross CPU memory domains.

Along with the high-speed interconnect, the system also needs to have native support for virtualization. Specifically, the processor needs to support CPU virtualization extensions like AMD-Vi and Intel VT-d, as well as memory and I/O virtualization for the PCIe pass-through features like function level reset and memory mapping. Note that while a processor may support virtualization extensions, it also needs explicit enablement in the system BIOS/EFI interface, as some vendors may not enable it by default. Without these features enabled, hypervisor software will

either run poorly or not at all. Most processors released since 2008 should have these features, but the more recent the processor, the better. In the case of Intel, recent high-profile CPU cache exploits and the subsequent mitigation techniques have come with some costs to hypervisor performance under load. More recent CPU releases have lessened the impact of these exploits, but it is worthy of note since cloud and virtualization vendors who use Intel processors will still have some of these affected CPUs in their datacenters for years to come.

2.1.3 Software Requirements

As it serves as the layer between the computer's hardware and the VM instances, the hypervisor plays an extremely important role in the level of performance that our VMs can achieve. Using a recent version is important so that the execution environment can make effective use of pass-through devices, large memory allocations at the hardware and VM layer, and optimizations in NUMA techniques. As discussed in earlier chapters, hypervisors passing control of devices to a VM have been around for a while. Newer GPUs can include memory banks that rival system memory and utilize low-level PCIe features. Using recent hypervisor versions ensures that there is support for these features. For example, in the test environment used here, advanced VM configuration had to be set in order to support large memory allocations on the PCIe bus. This functionality is available in KVM versions running on Linux 3.9 or higher, and on ESXi 4.0 or higher.

The VMs themselves should be built using UEFI boot and whichever drivers are provided by your hypervisor vendor to enable paravirtualized hardware. As of 2018, current versions of operating systems already include drivers or modules that allow seamless operation in a virtual environment regardless of hypervisor vendor.

The VMs built for the GPU-backed VM instances in the test environment

used in this thesis used Ubuntu Linux 20.04, though the methods and configuration presented here are OS-agnostic and should perform the same under other Linux distributions or Windows, given that similar versions of the software and drivers are employed. VMs used for load simulation used Linux and the stress-ng package due to its high flexibility in terms of targeting specific parts of the data processing path (King, 2017). Stress-ng was written with Linux in mind, though recent additions to Windows may make it possible to use for the load generation VM instances (this was not tested).

2.2 System Components

This section describes in detail the versions of the various components that were used to create an environment to test ML workload

2.2.1 Orchestration

For provisioning both the load generation VM instances and the GPU-backed VM instances in the experiment environment, automation scripts are provided and their usage is detailed in Chapter IV. These can be used to quickly stand up the initial environment as well as supporting components like networking. Since the experiment environment involves a hypervisor, it is also an ideal target for automation of these types of operations.

For this experiment, Terraform was chosen as the automation tool (Hashicorp, 2021). Terraform is an open source infrastructure-as-code tool with commercial support from Hashicorp. It's an ideal fit for environments where building from scratch every time you want to change parameters in undesirable. Since Terraform tracks the state of the components you deploy, it is fast to quickly deploy an environment, test, and then change things and redeploy. Terraform itself consists of a few small

components, a parser for Hashicorp Configuration Language (HCL), a framework for creating, updating, and deleting the state of your infrastructure objects, and an engine for using first and third party providers to perform the actual work of creating or changing infrastructure. A provider in this context is a bundle of code which implements all the Terraform functions for create, read, update, and delete for a list of objects from a particular vendor. For provisioning the test environment used in this thesis Terraform's vsphere provider was used.

Of note, during the experimentation phase the author had a vCenter instance available for ESXi. vCenter is an enterprise-grade management product for one or more instances of their vCenter is commercial software and VMware restricts a significant portion of API functionality behind it. ESXi is available as a free version for personal use. As much as is possible, a version of the automation scripts will be available for the free edition of ESXi.

Component	Version
Terraform	$\geq$ v0.13
vsphere provider	$\geq$ v1.24.3

Table 1: Version constraints for automation components

2.2.2 Hypervisor

The hypervisor is an x86 architecture server with enough resources to run multiple VMs simultaneously. It must also support PCIe device pass-through so that distinct GPU cards can be reserved by distinct VMs. The experiment environment used ESXi 6.7, the most current stable release of the 6.0 branch of VMware's hypervisor. It provides an industry-standard level of performance and stability, while providing all the features needed to run the type of ML workloads that are of interest today.

Component	Version
VMware ESXi	$\geq$ v6.7

Table 2: Version constraints for hypervisor

2.2.3 GPU-backed VM instances

GPU-backed VM instances are VMs which run ML workloads on the hypervisor. As discussed in the introduction, machine learning really doesn't have many constraints on what we can try to teach a neural network to do. Specifically though, these workloads can be simplified as different implementations of neural networks that do things like object recognition, sentiment analysis, or transformation tasks.

Rather than try to figure out parameters specific to each of type of workload, this thesis concerns itself with the hardware common to the GPU processing pipeline. Certainly, if one were interested in the hyperparameters or algorithms that could best process a specific type of ML workload, there would probably have been a new paper proposing a novel solution published in the time it took to read this paragraph.

Since this experiment is interested in the NUMA effects and virtualization bus-sharing factors of multiple GPU-equipped VMs, the experiment environment will need to have at least two VMs dedicated to simulating an ML training environment. Furthermore, each GPU will need to be attached to the PCIe bus of a distinct CPU socket, and then to a VM using a specific set of CPU cores. The number of cores and the allocation from each physical NUMA domain will be changed several times in order to demonstrate the various scenarios that can occur as VM processes are scheduled. Specific configurations for this will be discussed in the implementation chapter.

The GPU-backed VM instances use a baseline of 16 virtual CPUs and a total of 256 GB of RAM to match instance sizes used by common GPU instance sizes on

GCP and AWS. This CPU configuration was also chosen because the systems used for the thesis had enough physical cores that we could guarantee that a VM instance would only see a single NUMA domain. In large-scale training scenarios, more RAM may be desirable and this paper makes no specific recommendations for CPU to RAM ratios. For the sake of generating this paper's conclusions, this size made sense as an instance size larger than commonly available for desktop processors while having a RAM size that allowed for multiple GPU-backed VM instances to be loaded at once on the test hardware without resorting to memory over-commitment techniques.

Some recent Nvidia GPUs implement a technology known as NVLink, which chains together multiple GPU cards via a proprietary private bus in order to increase I/O speed and parallelism between GPUs by bypassing the PCIe bus. These cards were not taken into consideration during testing, due to their relatively high cost and lower applicability to the small-to-medium sized hypothetical ML workloads that are part of the scenario being tested for. It is the opinion of the author that NVLink is an impressive technology in terms of adding parallelism, increasing the available GPU memory space, and reducing the number of times that workloads must return to the CPU for additional data. However, this solution does not take away the fact data must still traverse the PCIe bus at least twice, once on load, and once on return of results. Therefore, it is still worth sizing VM instances in such a way as to avoid virtualization and NUMA effects, regardless of whether additions to the system bus such as NVLink are utilized.

A final consideration that is discussed briefly in the implementation chapter is that rack mount and blade server platforms can and do make use of interfaces that are uncommon in desktop and laptop systems. In particular, PCIe riser or other port multiplier solutions are used to increase the number of expansion slots, or for cooling considerations. These riser cards will make use of 32x PCI slots and divide those

lanes among multiple slots. This would be unremarkable, however it does appear to interfere with the reporting of PCIe link speed by `lspci` tool. In the case of one of the test servers used in the experiment environment, the GPU would be reported as being plugged into a 5 GT/s slot with a 32x width. This led to some considerable investigation time to understand why the link speed was reported in this manner. The lesson learned here is that relatively uncommon hardware plus virtualization interactions can cause strange reporting errors. In some cases, verification by physical examination may be preferable.

Chapter III.

Implementation

The original goal of this thesis was to produce recommendations around building configurations for how to best generically allocate computing resources in a mixed-use data center. The author envisioned a deep-dive into virtualized system architecture and alignment of virtual CPUs against a physical architecture. This study would investigate the effects of higher CPU utilization, non-uniform memory access effects, and high VM density vs. high physical CPU use. This was driven by a need to understand how to best allocate shifting consumption of compute workloads from databases and web applications to ML and other research workloads. Historically, applications like web servers were easy to virtualize because those applications had CPU usage patterns that involved low sustained use or short bursts of high utilization.

Instead of low utilization that can be easily shifted to idle processors or even other physical nodes based on overall system load, number crunching for research and machine learning uses a constant stream of I/O from local/remote storage, to the CPU for preprocessing, then onto the PCIe bus for GPU processing. Without going into detail about the nuances of CPU scheduling, virtualization has functioned best when resource utilization is short bursts of high activity vs. sustained use. Since virtualization is a foundational technology in cloud computing and data centers, it's important to think about the impact that many VMs running at high resource

utilization means for scaling and consolidation ratios in data centers.

During the period spent researching this topic, it became overwhelmingly clear that while there are projects that use CPU for ML training and inference work, a majority of this work takes advantage of the ability of GPUs to do fast floating-point arithmetic. However, this doesn't make the CPU obsolete or remove it from the ML workload equation. GPUs do one the one task of floating point arithmetic and do it extremely well, but cannot perform the common tasks expected of a computer. As the central component of a computer the CPU is required for allocating memory and moving data across the system buses to the GPU, and this vital role makes it ideal for preprocessing data. When considering the full data ingestion path and how many components are utilized, an ML workload doesn't appear to be something that should be difficult to use a hypervisor for. However, one of the most important factors for ML workloads is how much I/O the system can provide when it has to move data between system memory, GPU memory, network, and disk, all which share a common bus. That lead to the investigation of how much it matters to properly align VMs to particular CPU/Memory/PCI bus boundaries and if doing so truly provides optimization in the data center-size computing space.

A central question this thesis seeks to answer was what, if any, performance effects would be encountered due to I/O limitations on the PCIe bus and system bus. In virtualization scenarios that do not involve ML, this wasn't necessarily an issue, or at least it wasn't one to worry about over sustained periods. With the greater prevalence of GPU in servers, and not just one GPU per server, there now needs to be consideration given to what the cost will be to move data from disk, to main memory, and then onto the GPU and back. GPU and other application-specific accelerators depend on the use of 16x PCIe slots and bandwidth provided by the PCIe 3.0 and higher specifications. This places GPUs at the top of the priority chart in terms of

the bandwidth they can demand from the system bus.

3.1 Non-Uniform Memory Access and Multi-Processor Design

For a variety of reasons, designs for x86 processors changed in the early 2000s to place more processing cores on each physical CPU chip. These chip designs are referred to be as multi-core processors. This innovation made multi-CPU processing much more accessible by bringing a level of computing power to the desktop that was previously only possible in x86 multi-socket systems, which at the time used multiple single-core processors.

Today, desktop multi-socket processors can have as many as 10 cores per processor chip. Server-class processors are capable of having two more times that number of cores depending on CPU generation and manufacturer.

The most differentiating factor for systems with multiple processors over more common single-socket systems is that each socket gets its own set of memory pathways and system bus channels. This enables more effective density of VMs using GPUs because you can place more VMs on a system each with exclusive access to its own pass-through GPU without concerns that multiple VMs will consume all the PCIe bandwidth and slow operations for every consumer. Extra sets of PCIe lanes and memory channels come with their own set of concerns though.

Non-Uniform Memory Access is a design consideration in multi-processor system architecture that divides system memory into domains based on how close the memory is to each distinct processor. Each processor is designated as a single domain and can access its own memory without penalty. Processes running in a NUMA domain are able to access any part of the system memory, but accessing memory outside the execution domain incurs an access time penalty.

There are various implementations of NUMA depending on CPU architecture.

The major processor design companies using the x86 architecture implement NUMA using a central interconnect between all system CPUs. Intel refers to this component as the Ultra Path Interconnect or Quick Path Interconnect depending on processor generation. AMD uses a technology called HyperTransport for their interconnects.

Modern interconnect designs have greatly increased the speed of this component, but it cannot match the speed of a dedicated CPU to memory lane. NUMA designs enable a much higher density of processing power available in a single computer, though as described here there are additional parameters to take into consideration when writing software for these platform.

Multi-processor systems also get a full set of PCIe lanes allocated to each processor, greatly increasing the density of peripheral devices that can be attached to a single computer. Peripheral devices attached to a processor are also subject to the effects of NUMA in the same manner that memory is.

In a virtualized system, NUMA becomes a concern because VMs can be told that they have more than one processor and then be scheduled and executed on multiple physical CPUs when it may not be advantageous to do so. Knowing the conditions under which the hypervisor will make a VM instance aware of the host system's multiprocessor architecture is an important part of maximizing performance for a specific workload.

For this thesis, VMware's ESXi hypervisor was chosen as the platform. As an example, ESXi's rules for NUMA awareness are as follows: Any VM with an eight CPU configuration will be not be NUMA-aware, beyond that, any VM with a CPU count greater than the core count on a single physical CPU will be scheduled into multiple NUMA nodes. Figure 1 shows the output of the `lscpu` command on a VM instance with eight virtual CPUs.

As CPU count increases, NUMA node count for the VM instance will also

Figure 1: GPU VM instance with 8 CPUs

```
user@gpu-node01:~$ lscpu | grep NUMA
NUMA node(s):       1
NUMA node0 CPU(s):  0-7
```

Figure 2: GPU VM instance with 16 CPUs, host has 40 cores (80 logical CPUs)

```
user@gpu-node01:~$ lscpu | grep NUMA
NUMA node(s):       1
NUMA node0 CPU(s):  0-15
```

increase depending on the physical CPU socket and core count. If a VM can be scheduled onto a single physical CPU, ESXi will continue to constrain the VM instance to a single NUMA node. This can be validated on the test system, which has a total of two physical CPUs with 20 cores each. Note that NUMA calculations happen based on core count without hyperthreading or related thread-splitting techniques being considered. These processor features split a single CPU core into two execution threads, allowing the platform to report itself as having twice the physical cores, referred to as logical cores. To the untrained eye, this can be misleading, as other parts of the ESXi interface will report available CPUs as the value of cores * 2 with this processor feature enabled. Figures 2 and 3 illustrate this difference.

Likewise, NUMA controls where a VM instance's RAM is allocated even if the entire RAM allocation could fit entirely in the memory banks of a single NUMA node. See 4 and 5 for examples of the allocations.

Figure 3: GPU VM instance with 24 CPUs, host has 40 cores (80 logical CPUs)

```
user@gpu-node01:~$ lscpu | grep NUMA
NUMA node(s):       2
NUMA node0 CPU(s):  0-11
NUMA node1 CPU(s):  12-23
```

Figure 4: GPU VM instance with 16 CPUs and 256 GB of RAM

```
    user@gpu-node01:~$ cat /sys/devices/system/node/node*/meminfo | grep
MemTotal
    Node 0 MemTotal:       263948196 kB
```

Figure 5: GPU VM instance with 24 CPUs and 256 GB of RAM

```
    user@gpu-node01:~$ cat /sys/devices/system/node/node*/meminfo | grep
MemTotal
    Node 0 MemTotal:       131974098 kB
    Node 1 MemTotal:       131974098 kB
```

3.2 Test Environment Details

To test the ideas being proposed in this paper, two server-class computers with multiple GPUs were procured — A HP ProLaint DL380 Gen9 rack mount server and an HP ProLaint WS460c Gen9 blade server. Both systems meet the high-level requirements presented in chapter II, though they differ slightly in terms of CPU and GPU. System Configurations for these two systems are summarized in the following table.

The GPUs available in the test systems are not an exact match in terms of model. However, they are the same in terms of processor generation and installed RAM. Both utilize the Nvidia GM204 GPU core, which is one of the last chips to use Nvidia's Maxwell architecture. The following table summarizes the differences between the two in order to establish that the test systems are close enough to be able to consider results together.

It is worth noting that System 2 is a blade form factor, and does not use the traditional PCIe form factor for its GPUs. This server utilizes the Mobile PCI Express, which uses the acronym MXM, interface for its Tesla M6 cards. This interface

	System 1	System 2
CPU	Intel Xeon(R) E5-2698 v4	Intel Xeon(R) E5-2699 v3
CPU Family	Broadwell	Haswell
CPU Clock	2.20 GHz	2.30 GHz
System RAM	512 GB	512 GB
PCIe Revision	3.0	3.0
Max PCI Lanes	40	40
Installed GPU	Nvidia Tesla M60	Nvidia Tesla M6
Storage	None installed	None installed
GPU	Tesla M60	Tesla M6
GPU Card Interface	PCI Express	MXM
PCIe Slot Speed	16x	16x
GPU Architecture	Maxwell	Maxwell
GPU RAM	16 GB (2x 8 GB)	8 GB
GPU Core Type	CUDA	CUDA
GPU Core Count	4096 (2x 2048)	1536

Table 3: Hardware configuration for test systems

may be unfamiliar to those who haven't worked with blade systems or PC laptops that integrate high-performance GPUs over the past decade. MXM interface slots resemble laptop RAM slots, in that it uses a flat connector with the device entering at a 45-degree angle and then lowered until the back of the card is flat against the motherboard surface. Retaining clips and multiple screws are used to keep the card in place. As it is a form-factor and not a different bus, standard PCIe slot speeds are determined by number of pins utilized. While laptops will generally only have space for a single MXM form-factor card, other devices without the same space constraints can have multiple MXM slots. Indeed, this is the case with the blade server used here, as it comes equipped with four MXM slots, all of which are populated in the test environment.

For the MXM/blade configuration, it should be stated that the Tesla M6 is one of only a few Nvidia GPUs that utilized the MXM interface and were not designated mobile GPUs intended for laptop use. The Tesla M6 was released in September 2015

and has not seen a followup since. Given the lack of new product in this line, the comparatively low core count vs the PCIe version, and the absence of other servers that include GPU using the blade form factor it can be assumed that this was not a direction the data center market decided to go. Fortunately, the test parameters as defined here are decoupled from the actual GPU performance, as we are only controlling for PCIe congestion and NUMA effects. The MXM form factor does provide us with equivalent PCIe speeds and functionality, and therefore should still be a valid test bed here.

3.3 IOMMU

On the x86 platforms involving virtualization an additional system component comes into play, the input-output memory management unit (IOMMU). This interposer provides memory address virtualization and management functions for the PCIe bus. It also serves as a cache for translated addresses and a way to add a compatibility layer for devices that are not designed with virtualized environments in mind. In a hypervisor or bare-metal OS, use of the IOMMU is elective based on need. Enabling or disabling it is a configuration toggle in the kernel configuration. Use of the IOMMU is in fact a common configuration parameter when attempting to use desktop or gaming GPUs in a hypervisor environment. However, it has been shown that using the IOMMU device can cause appreciable latency for certain types of network workloads, (Neugebauer et al., 2018).

For the sake of completeness several benchmarks using this option were tested during the research phase of this thesis. In contrast to Neugebauer, there was not an observed difference in completion times of the benchmark with IOMMU disabled or enabled. This is likely a result of the NVIDIA Tesla cards in use being specifically built for use in a hypervisor environment. That said, based on other findings, it is likely

a best practice to enable IOMMU only if it is required in order to successfully pass-through a GPU to a VM. Adding additional PCIe latency is obviously not optimal, even if it was not observed here.

3.4 Machine Learning Workflow Sample

In order to test how both virtualization and NUMA can impact the execution of ML workloads, it's important to understand how these workloads are structured. The steps involved in a common ML workload will be summarized here.

All workloads will start with data resting on a storage volume. The closer the data is to the system bus, the better, so high-speed SSD or non-volatile memory express (NVMe) disks residing on a SATA 3.x or PCIe bus are preferred. These transports provide the highest speeds possible for data, with the lowest potential for transport latency that might be incurred by an Ethernet or fiber-based storage solution. That said, it is worth considering that because NVMe also shares the PCIe bus with other devices like Ethernet or GPU devices, that could potentially slow movement of data to a GPU depending on how the workload is structured.

Once the data is in memory, some preprocessing of the data element may be performed on the CPU for tasks that a GPU doesn't do well. Following this, data can be moved to the GPU over the PCIe bus. In the case of CUDA, which was used here, the required memory is allocated, and the object is moved from system memory across the PCIe bus into GPU memory.

Objects in memory on the GPU can then be processed by whatever GPU program is written for the workload and returned to system memory across the PCIe bus. Depending on complexity and data set size, all the objects may not fit into GPU memory at one time. In these cases, data should be moved in batches and returned to the system in batches.

Figure 6: Notable python libraries used in experiment code

```
numa==1.4.6
numpy==1.19.4
psutil==5.8.0
pycuda==2020.1
```

3.4.1 Code and Language Choices

Fortunately, analyzing the effects of virtualization and NUMA doesn't require a replication of the entire workflow described in the previous chapter. By focusing exclusively on an area of the process that utilizes a shared system resource, we can write tests that stress that component and see which, if any, parameters can be used to balance or ensure that a chosen workload gets 100% of the resources it requests.

Before going into further detail about the testing workflow, figure 6 lists the important external libraries used in the experiment code. `numpy`, `mp`, and `pycuda` are core to the functionality of the code (Harris et al., 2020), (McKerns et al., 2012), (Klöckner et al., 2012). `numa` and `psutil` are helpful in data collection about where and how processes ran (Smirnov, 2020), (Rodola, 2021). This chapter will discuss the specific application and rationale for each package's inclusion.

The experiment environment skips the loading of data from disk, opting instead to use the `numpy` library to create in-memory `numpy.float32` arrays using random data of a user-defined size at runtime. This process simulates the workflow steps where data would be loaded from disk, preprocessed by the CPU, and allocated in system memory.

`numpy.float32` is a reasonable default to choose for this example. The reasons being is that `numpy.float32` represents a single-precision number and is a reasonable default to choose given performance and support configurations in Nvidia's card lineup. Using double-precision numbers also requires a much larger memory footprint, re-

sulting in longer times to transfer data to the GPU for not much benefit. There is evidence that even lower levels of precision can still yield well-trained models, and that the costs of using higher precision don't outweigh the benefits of being able to do more training operations in the same time frame (Gupta et al., 2015). In the interest of running on as many devices as possible, and avoiding unnecessary processing and memory overhead, single-precision is an ideal choice in these experiments as a reasonable trade off between the speed of lower precision, and the possibility of using higher precision as future hardware innovation allows for it.

PyCUDA uses the most recent version, 2020.1, though the functions employed here are all stable portions of the code base, and it is not expected that they will be significantly altered in the future. PyCUDA is a python interface to the CUDA API that allows the programmer to make use of the power and speed of CUDA while also maintaining the convenience, readability, and breadth of the python language. Using python as a hook into CUDA enables the programmer to make use of popular and powerful data processing libraries like matplotlib, numpy, scipy, and more while still being able to write GPU code and execute from the python environment with few speed penalties. In this manner, the example can be more relevant to how machine learning code is written now than being a one-off example.

The experiment code also uses the python multiprocessing library. This allows the user to spawn any number of worker processes to split up the data across them for movement onto the GPU. During the design phase of the experiment environment, it would sometimes be observed that a single process could not fully saturate the PCIe bus. Running multiple workers ensures that the bus is completely consumed, as well as providing flexibility in the code should it be desired to run against multiple GPUs. Multiple GPU behavior is configurable at runtime, please see chapter IV for complete instructions on how to do this. By default, the code tries to account for

multiple GPUs if available and map them to workers in a round-robin fashion in order to emulate the behavior of popular libraries that take advantage of multiple GPUs like TensorFlow, though that is not a specific end goal.

Finally, the `numa` and `psutil` libraries are used to help with results collection. `psutil` is handy because it can provide all sorts of stats about the execution of a process. Most important here is that psutil has knowledge of what CPU a process executes on, which is important for being able to determine if a process ran in a non-local NUMA node. Due to the way python processes are scheduled by the OS, we can have certainty that all of a process' instructions will execute on the same CPU core, even when more than one processor is available. Since the multiprocessing library is used, it becomes even more important to track which CPU a process executes on, since child processes are likely to execute on other CPUs.

3.5 Testing PCIe Performance

In the benchmark code, there are a few key functions that are used to test the parts of the system that can tell us how load is being processed. This chapter will discuss specific details about the benchmark code that has been implemented.

The following table details all the switches available in the benchmark application.

Switch	Default	Description
–help, -h	N/A	Displays the help dialog
–hwinfo DEVICE_ID	0	Displays some diagnostic info
–single SIZE	8	Size in MB of an element
–batch SIZE	512	Size in MB of a batch of elements
–elements, -e N	1	Number of elements to use for –single or –batch
–num_devices, -d N	1	Number of CUDA devices to use in benchmark
–iterations, -i N	4	Number of iterations to run the benchmark
–workers, -w N	1	Number of concurrent processes to run

Table 4: Command line switches for the benchmark application

Figure 7: Main method spawning child processes

```
np_list = [args.single for x in range(args.elements)]
pool = Pool(processes=args.workers)
for i in range(args.iterations):
    print("Run {}:".format(i))
    total_size = args.single * args.elements
    res = pool.map(process_single, np_list)
```

The benchmark application has three major parts: the main method which ingests the command line parameters, the data to host memory mover, and the host memory to device mover. These correspond to various parts of the ML workflow, and it makes most sense to explain them in order starting with the main method. As Table 4 illustrates, there are a number of configurable run-time behaviors for the application. The main method is responsible for ingesting the command line parameters and dispatching those to the python multiprocessing module, which determines how many processes will run the load.

Figure 7 covers the majority of the program loading logic. The application creates a python multiprocessing Pool object based on the number of workers requested. Then for the number of iterations requested, the pool will spawn the number of workers specified by the `--workers` flag and pass a list of elements to the pool. The pool remains as the parent process and passes each element of `np_list` off to a child worker process to complete the load.

There are two ways in which to run the application code, single or batch mode. Single mode allocates and moves data once per element, while batch mode is intended to model the behavior of allocating and moving data as one large block. Moving memory in batches is the recommended method in order to optimize execution time and batch mode is implemented in a manner to mimic this behavior (Harris et al., 2020). However, due to the way the `--batch` switch is implemented, only a

Figure 8: Python method used to move data from system memory to GPU

```
def test_child(size):
    # Init the CUDA context
    cuda.init()
    # Create numpy array of n size
    np_array = np.random.randn(int(size * (10**6)/4)).astype(np.float32)
    # Allocate memory on GPU
    mem_gpu = cuda.mem_alloc(np_array.nbytes)
    # Allocate memory on host
    mem_host = cuda.register_host_memory(np_array)
    # Copy np array to system memory
    np_to_hmem(np_array,mem_host)
    # Copy system memory to GPU memory
    cuda.memcpy_htod(mem_gpu, mem_host)
    # Clean up CUDA instance
    mem_host.base.unregister()
    mem_gpu.free()
```

single worker process is possible due to restrictions in how CUDA manages execution context. Reasons behind this will be explained further along in this section when analyzing the data to host memory mover. Both modes have been implemented as a way to test if batch transfers still perform better when there's contention for bandwidth on the PCIe bus.

Before explaining this piece of code, it's important to understand that CUDA does not support an execution model where CUDA is initialized in a parent process and then accessed from a child process. Therefore, all child processes must initialize their own context when they start up. This is unfortunate from the standpoint that context creation and destruction does come with a time and resource cost in order to set up and tear down. Managing context is an important part of production code for this reason, but for the purposes of this experiment, we control for it by not starting the timer until after the context creation is complete. In production code, a library such as Tensorflow or similar would use a threaded approach and shared memory.

Once CUDA is initialized, the full power of PyCUDA is available to the child

process. This function takes in the size of the desired numpy array and creates it using the `numpy.randn()` function and of type `np.float32`. As discussed in 3.4.1, float32 is the most commonly-employed precision level as well as being the one best supported by Nvidia GPUs and therefore a natural choice for the experiment. With the array created, the space required for it can be registered in memory using the `cuda.register_host_memory()`function and then copied to system memory using the `np_to_hmem()` function.

This leads to the question of why this step is needed if the numpy object already exists in memory. In order to move data across the system bus, a memory region on the GPU device must map to a block of system memory. CUDA has two techniques to accomplish this, pageable or page-locked (pinned) transfers. `cuda.memcpy_htod()` permits the user to pass it a GPU memory allocation and the numpy object or a host memory allocation as the second argument. The first case executes the copy as a pageable transfer. However, in the first scenario, CUDA performs its own host memory allocation and copy to host memory anyway. With system memory allocation having a small cost in terms of execution time, it is therefore advantageous in production code to pre-allocate a block of page-locked memory and use that during ML workload execution.

For the experiment environment, the code uses the explicit allocation of memory for a page-locked transfer because it allows the isolation and timing of only the step where data is moved across the PCIe bus from system memory. Otherwise, it would be difficult to determine if latency is being encountered at the system memory or PCIe level once outside load is added to the system. In production code, this would be controlled for by designing code for the device and system RAM size and pre-allocating all the required memory ahead of time.

The code in Figure 9 just needs a brief explanation. Since the experiment code

Figure 9: Python method for moving data to main memory

```
def np_to_hmem(src, dest):
    source = src.ctypes.data_as(ctypes.POINTER(ctypes.c_float))
    destination = dest.ctypes.data_as(ctypes.POINTER(ctypes.c_float))
    size = src.size * ctypes.sizeof(ctypes.c_float)
    ctypes.memmove(source,destination,size)
```

is isolating the movement of data over the PCIe bus, the copy to system memory needs to be manually implemented. This is accomplished using the python ctypes functionality, specifically with the `ctypes.memmove()` function, which is required to ensure that the memory allocated by PyCUDA will match what is seen by the CUDA API when it executes. Effectively, `ctypes.memmove()` will populate the memory it is told to with C efficiency and ruthlessness: immediately and without regard for what is there already.

Finally, in terms of specifying `--single` vs. `--batch`, there are some implementation details that bear explaining. Single mode grants more flexibility in terms of the amount of variance between individual element size, number of workers, and number of elements. Starting the application with `--single 128 -w 2 -e 100` for example, would create a list of 100 elements of 128 MB in size, use two workers, and run for one iteration (the default). Due to the limitations of this implementation in terms of using the multiprocessing library, in `--batch` mode, the application can only run using a single worker. This is a result of not being able to share a memory pool among child processes. When in batch mode, the worker flag is set to one, even if a larger number of workers is specified. This means that when wishing to test batch mode, it is recommended to pick a relatively high element size, say a minimum of 1 gigabyte.

3.5.1 PCIe Bandwidth

PCIe bandwidth is traditionally expressed in GT/s, or gigatransfers per second. A transfer refers to the number of operations performed per second by a particular device across a data channel. While this might seem unusual for a computing device, it's important to remember that PCIe is a multi-layered standard with multiple points of interaction. The PCIe standard defines a transaction layer, a data link layer, and a physical connection layer. In other words, the maximum speed of a PCIe device is a function of its PCIe revision, and the number of PCIe lanes that a device uses.

The most important consideration for this thesis is the number of lanes that a device uses. A PCIe device will be classified as a 1x, 2x, 4x, 8x, 16x, or uncommonly, a 32x device. Each PCIe lane has two pairs of wires used for sending and receiving data, and as the number of lanes on a device increases so too does its throughput. With PCIe 3.0, each lane can operate at a transfer rate of 8 GT/s, or 2.0 GB/s total bandwidth per lane for send and receive. That is a bit of an important detail as well — bandwidth is the total data transfer for both directions, so the maximum throughput for send is 1.0 GB/s and receive is also maximum 1.0 GB/s. This works out to a specification maximum bandwidth of 15.8 GB/s in a 16x slot.

There is also transactional overhead involved with PCIe data movement. In benchmarking this during the development phase, it was observed that in general 6.1 GB/s is the average speed at which data can be sent, and 6.5 GB/s for receiving data from a PCIe device when the total data sent was greater 1024 MB. This works out to average about 12.6 GB/s total, a number that becomes more stable as the size of the data transfer increases. Results here correlate with Intel's PCI Express High Performance Reference design, which indicates that the expectation is that

throughput should increase and stabilize as the size of the payload increases (Intel, 2018). Please see Chapter V, specifically the baselines chapter for a more complete picture of what this looks like.

As a final point on the PCIe specification — while the PCIe 4.0 specification has been ratified and was released in its final form in summer 2017, the first systems from AMD and Intel to use it were only released in January 2019 and mid 2020 respectively. Few GPUs support the 4.0 specification at the publishing date of this paper, and it may be a couple more years before support is widespread.

There are a few final things to say about PCIe that have been a part of the specification from its very first version. The total number of PCIe lanes that a computer has is a sum of how many lanes are provided by the CPU and how many are provided by the motherboard chipset via electrical switches. What this means is that there is a limit to how many lanes are provided for peripheral devices, and furthermore, available lanes will be divided among the various slot types. Take for example Intel's Haswell chip, which defines a maximum of 32 PCIe lanes provided by the CPU. If a motherboard vendor decides to divide the PCIe lanes in an arrangement such as 16x, 8x, 4x, 2x, 1x, 1x, the ability to have multiple GPUs running at their rated speed is limited. The PCIe specification allows for a slot to accept larger cards than actually connected lanes, and therefore it is important to validate that a card is connected to the correct slot in order to maximize its performance. In the example above, only one GPU would be able to run at the full 16x speed, and any subsequent devices added would have their throughput halved.

3.5.2 Multiple GPU scenarios

One of the switches for the benchmark application, `-d or --num_devices`, allows the user to control the number of CUDA devices the application will use. This option

is available as a convenience for environments where PCIe throughput cannot be reached with a single device. As an example, these may be environments that have PCIe 4.0, but older GPUs that are not capable of saturating the bus. When this option is in use, the application will assign GPUs to workers in a round-robin scheme. In limited testing, this functions fine and definitely ensured that the test system was always completely saturating the PCIe bus. The default is set to one, and the application by design does not respect the system environment variable `CUDA_DEVICES`, which can be used to mask or enable devices as needed.

For environments that want to use a specific CUDA device, pass the argument `-c cuda_device_id`. CUDA devices are 0-indexed by the NVIDIA driver and can be identified by using the `--hwinfo` flag from the benchmark application. This will limit the application to using the single specific GPU, the application does not implement a way to control for more than one specific device.

3.6 Simulating non-ML workloads

Since the experiment environment is concerned with simulating general purpose virtualization environments, some benchmarks need to be run in an environment that has non-ML workloads running. In the interest of trying to keep the scope manageable the experiment environment does not try and set up numerous various workloads like databases or web servers. Rather, stress-ng is used to tax VMs at various CPU/memory/bus levels. These will generally try to adhere to some high-level usage patterns, such as databases having high disk I/O and memory transactions, or web servers that run at a constant CPU utilization and will be noted as such in results.

Non-ML workloads can be deployed in batches or clusters running various workloads types. Terraform can be used with the various template files provided to deploy different swarms of generic traffic nodes. Once they are provisioned, use the

Description	Switches
CPU stress only	–cpu-load 50 –cpu-load-slice 0
Add memory	–cpu-load 50 –vm-bytes 1G –cpu-load-slice 0 -vm 4 –cpu 24
Add NUMA stress	–cpu-load 50 –vm-bytes 1G –cpu-load-slice 0 -vm 4 –cpu 24 –numa 24

Figure 10: Load Generator Workload Types

bash scripts to issue commands to them. Since an unknown number VM instances may be spawned for this purpose, the test automation does not depend on networking being available. Rather, all the VM instances can be controlled using an open source component written in the Go language called `govc`. `govc` is part of the `govmomi` library, which is an implementation of the VMware API using Google's Go language (MacEachern, 2021). `govc` allows the user to run commands on a VM without network access by making use of the hypervisor to guest interface via VM support tools. The bash scripts are tuned to use the Terraform outputs and dispatch commands to a group of these non-ML workloads VMs using `govc`.

Figure 10 summarizes all the stress-ng command-line switches used during the experiments.

Chapter IV.

System Documentation

This chapter specifies how to reproduce the execution environment.

4.1 Basic VM instance

This thesis uses Ubuntu Linux 20.04 as the guest OS for all VM instances. Ubuntu Linux 20.04 is a long-term support version released April 23, 2020, which means it values stability and reliability and will be supported for a period of five years. Ubuntu releases are popular platforms for ML researchers as well, and enjoy frequent updates from third-party vendors like Nvidia, making the OS a natural choice for both the GPU-backed VM instances and the load generation VM instances.

Before taking any other step, download an installation image of the Ubuntu installer from `https://ubuntu.com` for Ubuntu LTS 20.04. Ubuntu Linux is distributed in both desktop and server editions, but these instructions can be used to reproduce the benchmark environment with either edition. The benchmark environment does not use any GUI-based tools that are included with Ubuntu Desktop, so the smaller download size of the server edition may be preferable.

The next step is to create a template VM instance that can be used as a base for the other resources that must be created. If desired, the user can build the VM instance manually using the GUI. However, `github.com/broestls/virtual-numa-test`

includes code using Terraform to help manage these steps.

Start by cloning the thesis repo, `github.com/broestls/virtual-numa-test`, and entering the `environment/globals` folder. This folder contains configuration parameters that need to be set, such as the ESXi host to use, the virtual network to use, and other common parameters. The comments in that file will also cover the necessary steps to set up a credential set for future steps. Once those configuration parameters are complete, change to the `terraform/bootstrap` director, which contains the Terraform code necessary to build the VM instance template that can be used as a base for every component in the experiment environment. Building this portion of the experiment of the environment will require the user to run the commands `terraform plan` and `terraform apply` in the `bootstrap` directory.

The result of running the bootstrap code is that there is a new VM instance ready for guest OS installation. From the ESXi interface, open a console for the new VM and attach the installation media downloaded in an earlier step and power on the VM. Complete the installation using the defaults, unless you know a different configuration is needed for your situation. The installation is followed by a reboot, at which point the VM instance should be ready for use. Figure 11 lists the common packages that can be installed using `apt` using the console or the Terminal application if using Ubuntu desktop.

Figure 11: Required packages for benchmark VM instance

```
openssh-server
python3-dev
python3-setuptools
jq
stress-ng
```

At this point, shut down the VM and from the ESXi interface, take a snapshot of the VM with the name `ubuntu-2004-stress-ng`. This will be the base for the load

Figure 12: Commands used to install CUDA 11 on Ubuntu 20.04

```
wget <NVIDIA Driver download link>
sudo sh NVIDIA-Linux-x86_64-450.57.run
wget https://developer.download.nvidia.com/compute/cuda/11.2.1/
local_installers/cuda_11.2.1_460.32.03_linux.run
sudo sh cuda_11.2.1_460.32.03_linux.run
git clone https://github.com/broestls/thesis-code
```

generation VM instances. Start the VM back up for additional software installs.

The next steps will require the retrieval of a download link from an external site and will be used for the first command in figure 12. Nvidia's Linux driver is required and the download link can be retrieved from `https://www.nvidia.com/en-us/drivers/unix/`. The latest version in the 460.x series of Nvidia drivers is appropriate for CUDA 11. Copy the link and use it where specified in Figure 12. With the steps in Figure 12 complete, once again shut down the VM and take another snapshot, using the name `ubunutu-2004-gpu`.

4.2 GPU-backed VM instances

From the experiment environment repo, switch to the `environment/gpu-instance` directory. Terraform provisioning steps (`terraform plan; terraform apply`) can be used to deploy a total of four GPU-backed VM instances. This process makes use of the concept of linked clones, where each of the VM instances use one of the snapshots taken of the template VM made in the previous section. In this manner, VMs can quickly be spun up and destroyed while using very little disk space.

Terraform will power up the VMs automatically. Once that is complete, it is up to the user to log onto each node to execute the benchmark application as needed. Please see the section 3.5 for details on how to tailor the benchmark application to specific scenarios.

4.3 Non-GPU Load Generation

From the experiment environment repo, switch to the `environment/load-generation` directory. The file `terraform.tfvars` file controls how many load generators will be spawned. Alter the `num_load_generators` and `load_profile` variables to set the number of VM instances to spawn and what stress-ng load will be executed on each one. When `terraform apply` is run, the VM instances will be quickly spawned and begin running their specified stress-ng command until a `terraform destroy` command is issued.

Chapter V.

Experimental Results

This chapter discusses in detail the results of the experiments set up to test where machine learning workloads performed best or worst. The beginning of the chapter will present baseline numbers for different common scenarios with GPU and no other workloads. That will be followed by a section that presents results when running multiple GPU-backed VM instances in parallel. Next, some results involving single GPU-backed VM instances and non-ML workloads will be presented. The final section will present results that involve multiple GPU-backed VM instances and non-ML workloads, *i.e.*, a scenario designed to model real-world load.

Each experiment result will include a table with the configuration used, some contextual information about what is intended with that particular configuration, and what result was expected. Where appropriate, charts or graphs will be included to visually explain results. Each result section will conclude with a summary of what was observed along with notes about the run and comments on how it differed from the expectation, if applicable. In some cases, figures may be compiled into a single chart and referenced in the text depending on the size of the chart.

Prior to each experiment group each system was rebooted and allowed 15 minutes to perform startup tasks and return to an idle state. Also, unless otherwise noted, each GPU-backed VM instance runs with a total of four processes running the

benchmark application running in a batch mode and static/single processing mode. Initial sample runs indicated that PyCUDA and PCIe 3.0 on the test hardware server was able to consistently max out the roughly 6 GB/sec bandwidth available on the bus. While it was observed that each process could move 6.1 GB/sec of data on their own, in practice running four processes insures that each benchmark run is fully saturating the bus with data at every interval possible.

5.1 Baseline Results

This set of experiments exist in order to establish baseline results. For each experiment in this group, a single GPU-backed VM instance is run using different CPU/PCI placement scenarios: no CPU affinity, local CPU affinity, and non-local CPU affinity. Each GPU-backed VM instance will have unfettered access to system resources, bounded only by the fences placed on CPU and RAM utilization by the hypervisor. System resource contention is minimal as there are no other VMs running that could cause even the slightest speed bump for the GPU-backed VM instance.

Predictions for this set of experiments is that GPU-backed VM instance will be able to saturate the entire bandwidth available to a 16x PCIe card. This actually isn't a bold prediction, since the baseline scenario is also the target parameters used during the development phase. During development of the application, there were plenty of examples of seeing the application use all the available bandwidth regardless of CPU affinity.

For these experiment runs the default GPU-backed VM instance size for each test system was used, 16 virtual CPU and 256 GB or 64 GB of RAM with no other VMs running on the hypervisor host. On the test machines, this configuration was guaranteed to fit in a single physical NUMA node because it did not extend over a core or memory boundary. `lscpu` output confirmed that a single NUMA node was

available in the guest OS. It can be cleanly scheduled on either CPU socket and should remain there for the duration of its execution.

For this batch of experiments, the command used was `python bwt.py -d 1 -i 3 -w 1 -e 10 -n <run_name> --single <element_size>` for the single transfer process and `--batch <element_size>` for the batch transfer mode. This translates to using a single device, repeat the run for three iterations, and use one worker process. Single mode used 10 elements, whereas batch mode is a single element per iteration.

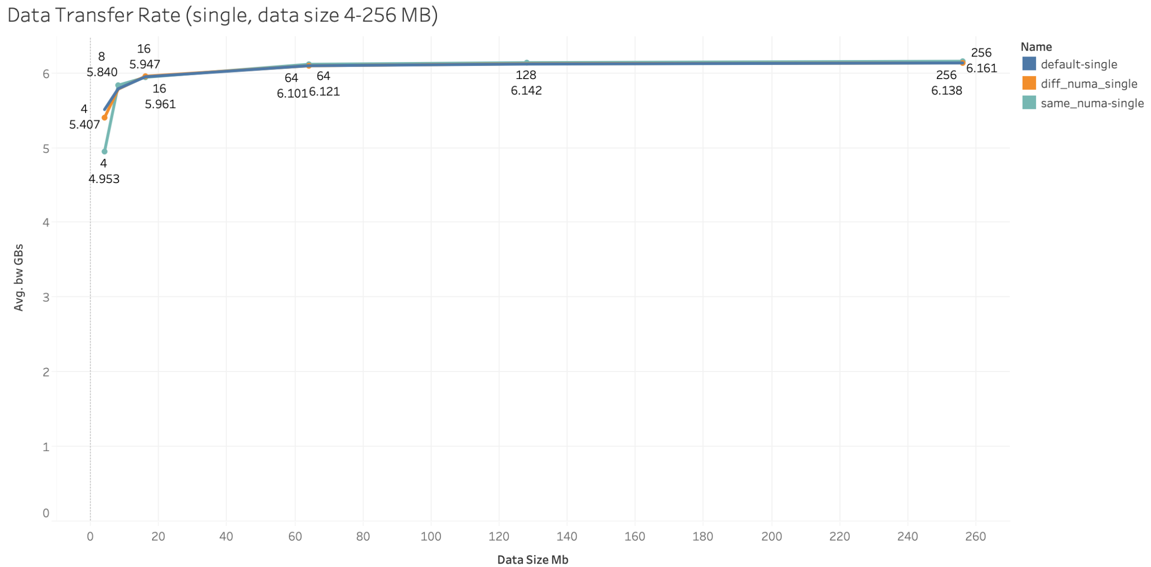

Figure 13: Baseline data transfer rate for single mode

Figures 13 and 14 illustrate the results, and the first thing to note from these results is that we can more or less confirm the findings from Walters et al. In pass-through mode, we can match or come within a couple percentage points of bare-metal GPU performance. The test program is easily able to reach the PCIe 3.0 data transfer rates that were seen during the development phase and were detailed in 3.5.1. Therefore, a couple conclusions can be drawn — the first being that a single VM can easily consume all the physical bandwidth of a 16x PCIe slot by itself, and that this can be accomplished using a fraction of total system resources available at the

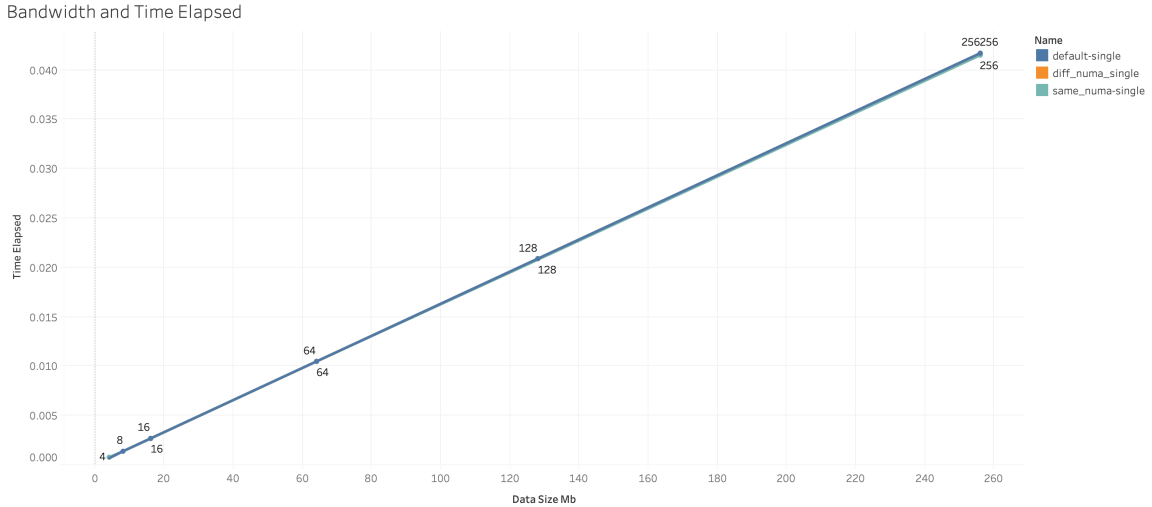

Figure 14: Baseline time elapsed for single mode

hypervisor level. Indeed, even in smaller test cases using four or eight virtual CPUs, these results remained unchanged.

Several more observations are worth noting. It is apparent that the systems' CPU interconnect between NUMA domains contains plenty of bandwidth to be able to continue to saturate a PCIe slot, as bandwidth values remained consistent across all baseline scenarios. So it can be said that in a situation with no other load on the hypervisor host, NUMA effects are non-existent for moving data across the PCIe bus.

As stated in the intro paragraph to this chapter, this CPU configuration is always able to be scheduled on a single CPU socket from the standpoint of the hypervisor. Indeed, even when CPU affinity is not set, the VM instance stayed constrained to a single physical CPU and was never scheduled across two physical CPUs. The benchmark application also displays which CPU a worker node is executed on, and during these tests it reported using at least two distinct CPUs. On the hypervisor side, this behavior was reflected by the hypervisor assigning a different physical CPU core when the single worker node swapped to a new CPU in the VM. Curiously, even when not using any CPU pinning for the VM, the hypervisor preferred to assign CPUs

on the second CPU socket regardless of the location of the PCI card in the system.

To confirm this behavior, a new experimental run was enqueued using 10 iterations per run. The results of those runs confirmed that without any CPU affinity being set at the hypervisor level, the GPU-backed VM instance continued to be scheduled on the second physical CPU socket in all but the case where CPU affinity was constrained to physical CPU 1. There was not a good explanation for this behavior, but at least in this case there is not a penalty for it.

5.2 Workload spanning multiple NUMA nodes

In this scenario, a VM instance is large enough that the hypervisor must schedule it across multiple physical NUMA domains. The rules for the hypervisor used in the tests are reviewed in section 3.1. Here, the VM CPU configuration is set to use 24 CPUs and 384 GB of RAM. This ensures that the hypervisor will advertise to the VM instance that multiple NUMA domains are available. This can be validated using the lscpu command, which is shown in figure 3.

As in the previous experiment, no other load has been added to the system yet. The GPU-backed VM instance should be able to utilize 100% of its resource assignment without any contention. Based on the results from the baseline batch of experiments, it is not expected that this scenario will yield significantly different results.

As expected and shown in figures 15 and 16, the results did not differ for this experiment. If anything, it reinforces the fact that throughput improves with data size and that in an unburdened system, NUMA effects simply don't appear to be an issue. In terms of scheduling at the hypervisor level, it was indeed observed that CPUs from both NUMA nodes were utilized. This set of experiment runs did demonstrate an interesting part of the hypervisor scheduler though that couldn't be

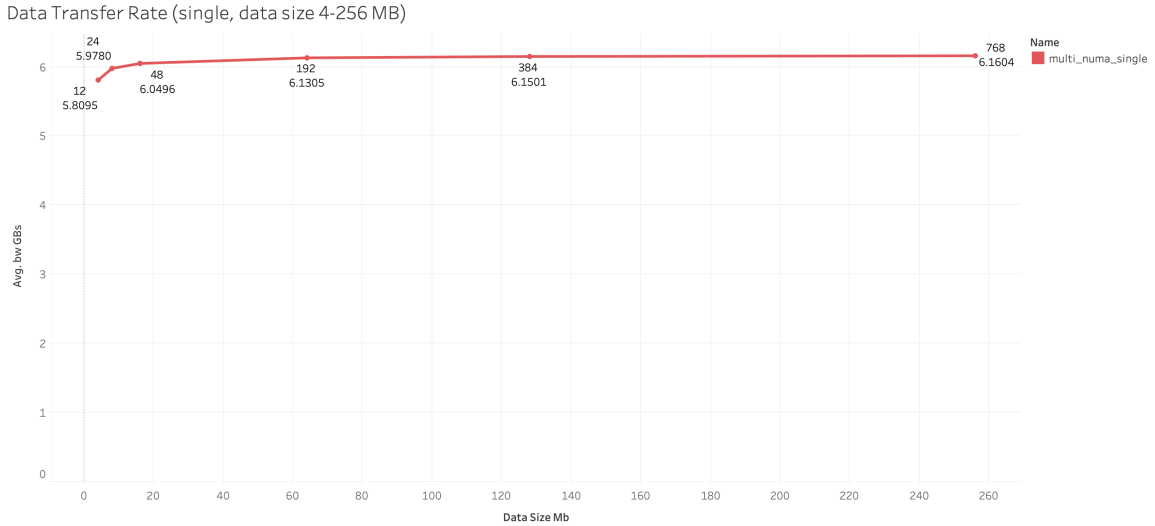

Figure 15: Data transfer rate for single mode when VM instance sees multiple NUMA nodes

quite quantified during the baseline experiments.

During this experiment, it was much more clear that from the guest OS point of view, a worker can keep executing on the same CPU over and over. The hypervisor takes a much different view though. During a run in batch mode using a data size of 4096, it was observed that the benchmark application used the same CPU for eight of the 10 elements in the set. However, the hypervisor used a total of six distinct physical CPUs to do the work. This presents an interesting circumstance for the user who might use the python library numa to try and control which cores a workload executes on. The actual execution core is obfuscated from the user, and as a result it is fruitless to try and control for NUMA at the guest OS level.

5.3 ML Workload on Mixed-use System (CPU-bound Load)

With baseline numbers established, the next step is to make it more difficult for the GPU-backed VM instance to get access to resources. This set of experiments will start with a single load generation node, perform a run of the benchmark program,

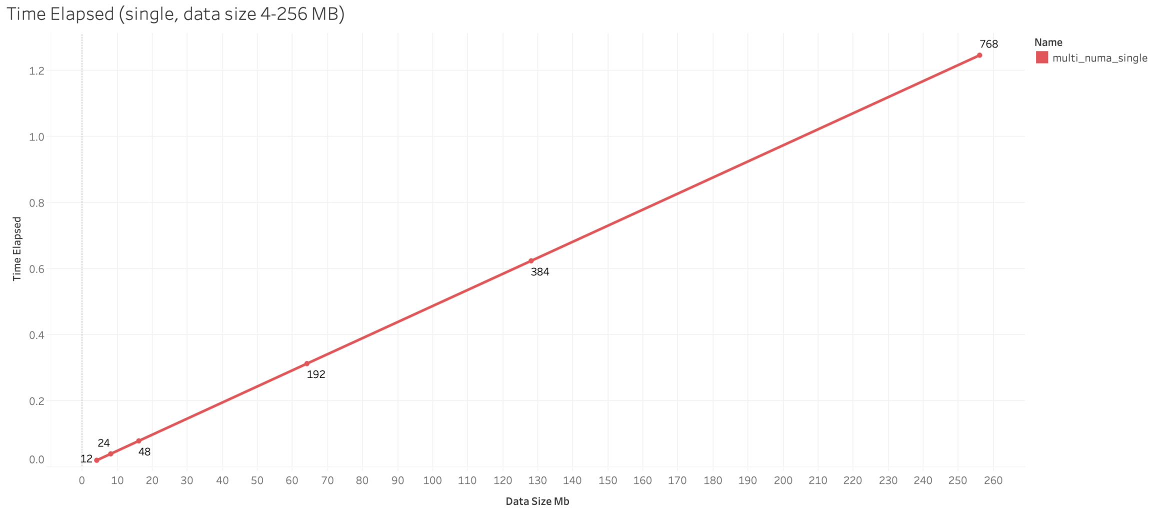

Figure 16: Time elapsed for single mode when VM instance sees multiple NUMA nodes

then add more load generation nodes, and repeat until total system utilization has reached 70%.

For this set of experiments, the load generator nodes will initially be limited to an eight CPU configuration. This is to ensure load generator nodes will not expand past a NUMA boundaries. The load generation will use `stress-ng --cpu-load 50 --cpu-load-slice 0 --cpu 8` to simply set the utilization to 50% across on all load generation nodes.

The expectation is that the GPU-backed VM instance will continue to run full speed until a large enough number of load generation nodes come online and cause CPU contention for the GPU-backed VM instance. In the baseline scenarios, it should be noted that getting full speed on the PCIe data movement required 100% CPU. That is likely to be the only thing that will cause any slow down until the nodes that are bigger than a single NUMA node are added into the mix.

Results from this experiment were a bit surprising in that a total of 10 load generators were brought online and using a pattern where they iterated between 0 and

50% utilization over a random gap of time between 0 and 0.5 seconds. At 10 running load generators with eight CPUs each plus the GPU-backed VM instance, the CPU utilization for the test system was at 100%, each individual CPU core had an average utilization of 50%. The experiment began with a single load generator, then two, and then they were doubled every pass until 10 load generators were running. Only one run displayed any significant change to the throughput that the GPU-backed VM instance was able to achieve. That run was one where 10 load generators were running, the application was in single mode, and using an element size of 64 MB. The bandwidth on that iteration was 5.6 GB/s . 6.1, a roughly 8% performance penalty.

Total experiment time was over a period of 45 minutes, though not fully loaded that entire time. It is a good sign that in an environment where the non-ML workload is constrained to just uncomplicated CPU operations, a single ML workloads can run with little encumbrance.

5.4 ML Workload on Mixed-use System (CPU and Memory-bound Load)

This scenario builds on the previous one by continuing to stress the CPU at 50% and also adding memory allocation and freeing to the mix. Load generation VMs will continue to use the same VM hardware profile as in the previous scenario of eight CPUs and 32 GB of RAM. Memory allocation will run using four workers, which will allocate and free 1 GB of RAM each time they run. The command passed to load generators is `stress-ng --cpu-load 50 --vm-bytes 1G --cpu-load-slice 0 -vm 4 --cpu 8`

For this set of parameters, expectation continues to be that until the system becomes heavily loaded there won't be much of a negative effect on PCIe speed, if at all. Memory allocation operations don't touch the PCIe bus. In this case, the potential for slowdown for the GPU-backed VM instance is if a large single payload

or multiple smaller payloads need to traverse between NUMA domains at the same time as a memory transfer operation for a GPU-backed VM instance that is also trying to traverse NUMA domains.

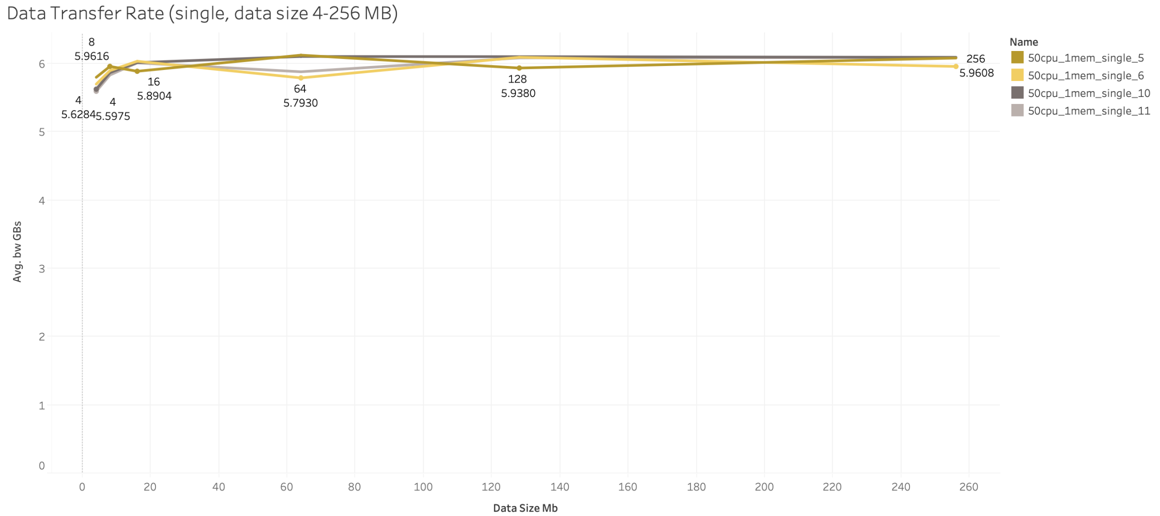

Figure 17: Data transfer rate sampling when loading CPU and RAM

Figure 17 shows the results of this particular scenario. Results were a surprise on the whole. Even though the expectation was to see greater penalties for data movement as total system utilization increased, it never came to pass. There were some benchmark runs that yielded slightly slower data throughput to the GPU and Figure 17 highlights some of these. The worst case scenario here was with the 64 MB element size, where multiple iterations reported a data transfer speed of 5.79 GB/sec, or about a 5% penalty. In aggregate the results were slightly worse than when just CPU was being stressed, overall there is only a 1-2% slowdown over the GPU-backed VM instance running its workload as the only tenant on a hypervisor system.

These experiments are focused on analyzing potential effects on PCIe data movement speeds, but it is worth noting that it was observed that total execution time for each benchmark run (*i.e.,* context creation, memory allocation, transfer, memory freeing, and context destruction) were slightly longer (about 1 second) at

higher levels of hypervisor system utilization. This could become a factor depending on the types of workloads being mixed together. To be fair though, stress-ng is likely allocating and freeing memory more frequently than a real-world application would be doing.

5.5 ML Workload on Mixed-use System (Multiple NUMA-aware VMs)

This scenario uses the CPU and memory stressors of the previous experiment, but the load generating VM instances use a virtual CPU configuration that is certain to be scheduled across multiple physical CPU sockets. `stress-ng --numa` will cause the worker processes to spread move data in such a way that they stress the CPU interconnect and CPU caches. The load generator nodes also need to be resized in order to use the –numa switch for stress-ng, as it requires the system to have more than one NUMA node. Command used was stress-ng `--cpu-load 50 --vm-bytes 1G --cpu-load-slice 0 -vm 4 --cpu 24 --numa 24`

Expectation in this experiment that some lag on the PCIe bus will be observed here as the GPU-backed VM instance is also configured with no CPU affinity and is free to be scheduled on any CPU.

The results here were surprising in the sense that they didn't really match expectation. Due to the higher virtual CPU configuration of the load generation VM instances, the hypervisor system's resources became constrained much more quickly. Just the GPU-backed VM instance and one load generator alone was enough to cause the hypervisor to report that 81% of the CPU was consumed. With two load generator VM instances running, CPU consumption quickly rose to 100%.

Once three or more load generators were running, more of the 4-5% penalties started to appear in the results, though batch mode results stayed consistent as they have throughout the experiments so far. The results do confirm a hypothesis

put forward in the previous section, that more CPU and memory resources that are allocated on non-local NUMA domains increases the chance that the GPU-backed VM instances may experience lower performance. That said, that chance is also fairly low, especially in batched transfers that don't need to allocate/free memory constantly. At this point, it is looking clear that if one is seeking the most consistent results, allocating a large block of memory and using that throughout model training is a more reliable methodology to adopt.

5.6 ML Workload vs ML Workload

This section focuses on test of how PCIe performance is affected when multiple ML workloads are running by using multiple GPU-backed VM instances in different configurations. Based on the results of the previous experiments, it appears as though workloads that only traverse the CPU caches, system memory, and between NUMA domains is not able to strain the system in such a way that movement of data to a GPU device is severely affected. For this set of experiments, only ML workloads run on GPU-backed VM instances will be used. The focus is on maximizing utilization of the PCIe bus to the greatest degree possible with the hardware available on the test system.

5.6.1 Experiment Setup

Table 5 presents a total of seven different arrangements of GPU-backed VM instances using the four available GPUs in the test systems. These different alignments of GPU-backed VM instances test the various ways in which GPUs can be allocated to VM instances to run ML workloads. Scenario one and four best resemble what would be considered to be best practice, where the GPU-backed VM instances are placed in the same NUMA domain as their backing device.

Scenarios two, three, five, and six use configurations which place greater numbers of VM instances in a different NUMA domain than their backing GPU device. It should be noted that in these scenarios, the configurations are intentionally set to induce contention between the GPU-backed VM instances. Seeing these in a real-world situation would be unusual, but it is necessary in order to be able to observe and measure any possible contention.

Scenario seven is intended to model a real-world configuration made without consideration of NUMA effects. For this scenario, all the VMs use 24 virtual CPUs so that they are NUMA-aware and can be scheduled on either physical NUMA domain. The GPU-backed VM instances also do not have any CPU affinity configuration in place, the physical CPU core they execute on will be scheduled at the hypervisor's discretion. This details of this configuration and its implications are discussed in sections 5.2 and 5.5.

Except for scenario seven, which uses a specific virtual CPU configuration noted in the previous paragraph, all other GPU-enabled VM instances use 16 CPUs and 64 GB of RAM.

Scenario	Number of Nodes	Condition
1	2	GPUs in a local domain to owning VM
2	2	Both GPUs are in a non-local domain to owning VM
3	2	GPUs are both non-local to owning VM
4	4	GPUs are all local to owning VM
5	4	no GPU is local to its owning VM
6	4	half the GPUs are local and half are not
7	4	All nodes are NUMA-aware, no pinning

Table 5: ML vs. ML workload experiment scenarios

This experiment group differs from previous ones because of the involvement of more than a single GPU-backed VM instance and therefore has more complex interactions of hardware components. Scenario one is there simply to set a baseline

set of results for two GPU-backed VM instances. There is a clear expectation that throughput should be affected in every other scenario, starting with scenario two. This expectation became evident during development, where unoptimized testing against multiple GPUs was a reliable way to see negative effects on PCIe throughput.

One other factor that bears mentioning is that there may be some speed penalties due to the way the Tesla M60 cards in the primary test system are architected. Since the M60 places two GPUs on a single PCIe expansion card, there is a very high chance that there will be immediate speed penalties upon using two GPU-backed VM instances that access the same PCIe expansion card. In those cases, the more interesting result will be if the load is balanced fairly by the hypervisor and system hardware. This should provide some insight of how fair the sharing of bus resources is once they start to become constrained.

5.6.2 Results

This section will review results from all seven scenarios presented in Table 5. Results are broken down into two subsections, section 5.6.3, which covers results using 2 GPU-backed VM instances (scenarios 1-3) and section 5.6.4, which covers results using 4 GPU-backed VM instances (scenarios 4-7).

Figures 18 and 19 for the experimental runs that end in _1. These correspond the experimental profile in line 1 of table 5. Those are for two VM instances with GPU executing on a single physical CPU with the PCIe devices residing on the same PCIe lanes owned by that processor. It can be observed that with two GPUs in use, the VMs can still sustain around 6.0 GB/s of throughput each to the GPUs.

5.6.3 Results with two GPU-backed VM instances

This section will discuss the results from scenarios that used only two GPU-backed VM instances. Scenario 1 (Table 5) serves as the baseline for all multiple GPU experiments. In scenario 1, two GPU-backed VM instances were benchmarked, each with CPU affinity for the VM instance set to a separate physical CPU socket and backing GPUs located on PCIe lanes owned by that same processor. As shown in Figures 18 and 19 (Scenario 1), this set of parameters allowed both GPU-backed VM instances to operate at full speed.

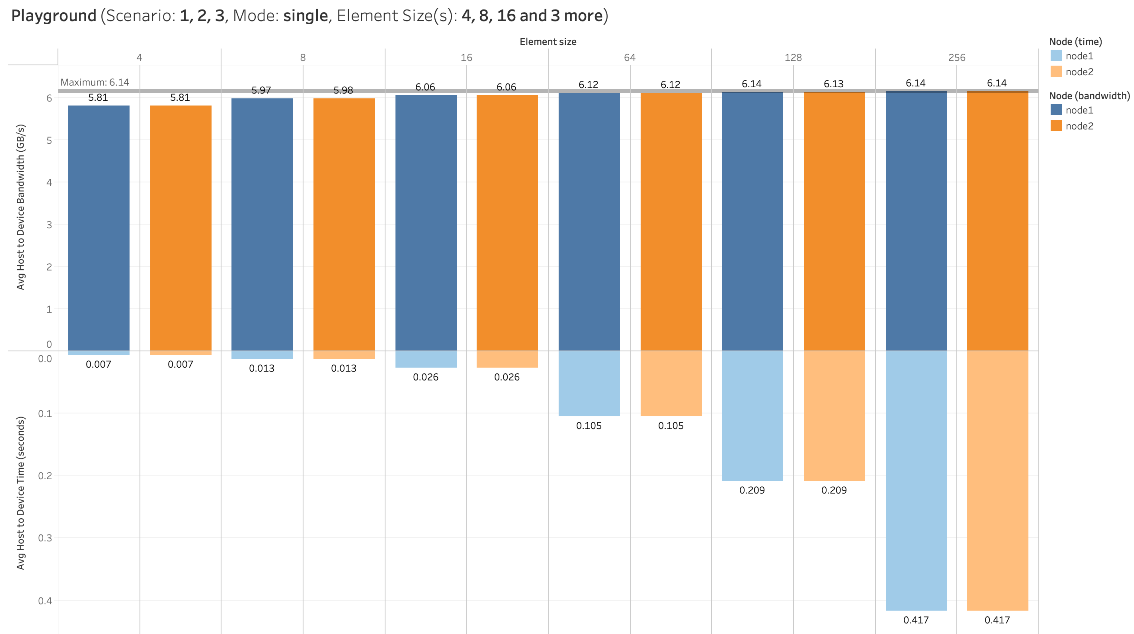

Figure 18: Baseline data transfer rate and time elapsed for multiple ML workloads (single mode)

In addition to the baseline configuration, scenarios two and three also demonstrated full speed for all GPU-backed VM instances in both single and batch mode. Scenario two involved two GPU-backed VM instances in different NUMA domains with both their backing devices being non-local to their NUMA domain. No performance drop was observed, even though both devices had to send data from non-local

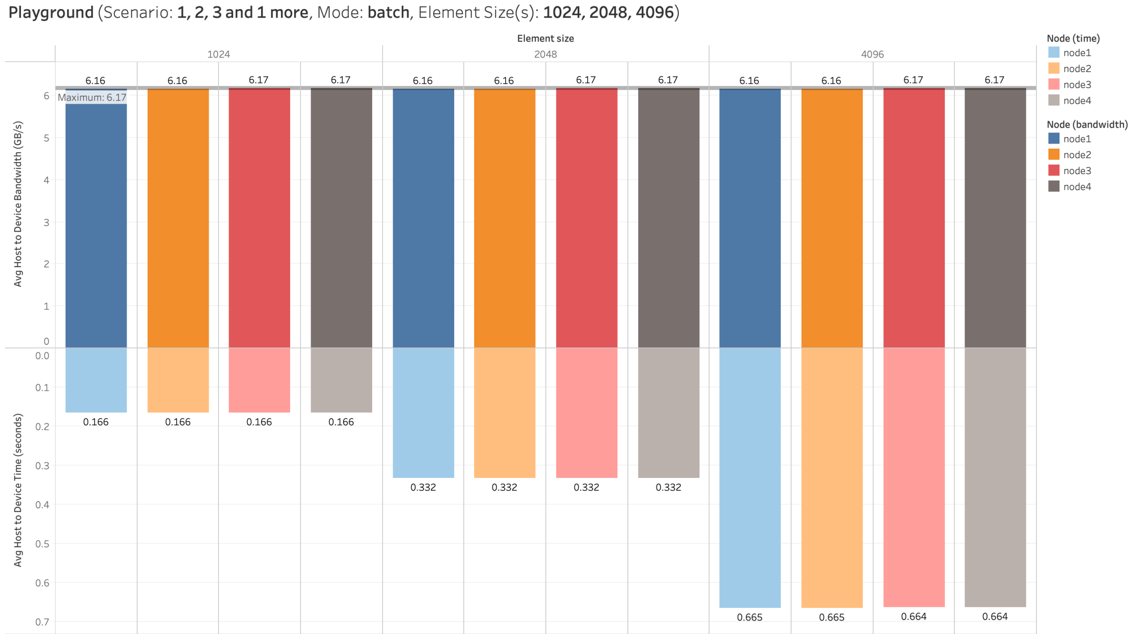

Figure 19: Baseline data transfer rate and time elapsed for multiple ML workloads (batch mode)

memory. Likewise, scenario three (two GPU-backed VM instances in the same NUMA domain with GPU devices non-local) showed no drop in performance regardless of data size.

There is one difference between the results in single and batch mode that merits further explanation. In all instances of the benchmark in single transfer mode regardless of whether two or four GPUs are used, it can be observed (and shown on the graph in Figures 18, 20, 21, and 25) that transfers with data element sizes of four and eight MB ran slightly slower. This is a function of PCIe data transfer overhead and data encoding. As Intel details in their PCIe reference design documents, as data size increases, the overall effects of the overhead involved data encoding and connection negotiation are minimized, leading to greater throughput.

As a result, it is worth taking into consideration that smaller element sizes should at least be batched into 64 if not 128 MB chunks. Doing so reliably provides

around a 5% speedup in overall transfer speed over the entire transfer.

The results with two GPU-backed VM instances were encouraging in the sense that they provide confirmation that there is plenty of bandwidth, even crossing NUMA domains for two machine learning workloads. As the previous experiments have shown, a single GPU-backed VM instance working in its own NUMA domain never saw any penalty to its performance. It is safe to say that this is also true for two GPU-backed VM instances.

5.6.4 Results with four GPU-backed VM instances

This section will discuss the results of scenarios that used four GPU-backed VM instances. Scenarios four through seven in Table 5 are those that involve four GPU-backed VM instances. These results make clear that at least four GPU-backed VM instances are needed in order to see any negative effects on throughput and will be discussed below.

The biggest takeaway from scenarios four through seven is that penalties to throughput were seen by one or more GPU-backed VM nodes in multiple scenarios. However, this wasn't true of both modes, as batch mode performed well in two of the four scenarios where single mode only performed well in a single scenario.

The differences between single and batch mode in scenario four are worth examining closer, as the parameters were not expected to cause any contention. Scenario four is one that doesn't have any GPU traffic crossing into non-local NUMA domains, and it had a poor result for single mode anyway. After some review of the results, it appears as though the slowdown was likely the result of some competition for CPU time between the two GPU-backed VM instances. In the case of single mode, because each new child process that is spawned could be executed on another CPU by the guest OS, it increases the chance that two processes could accidentally be

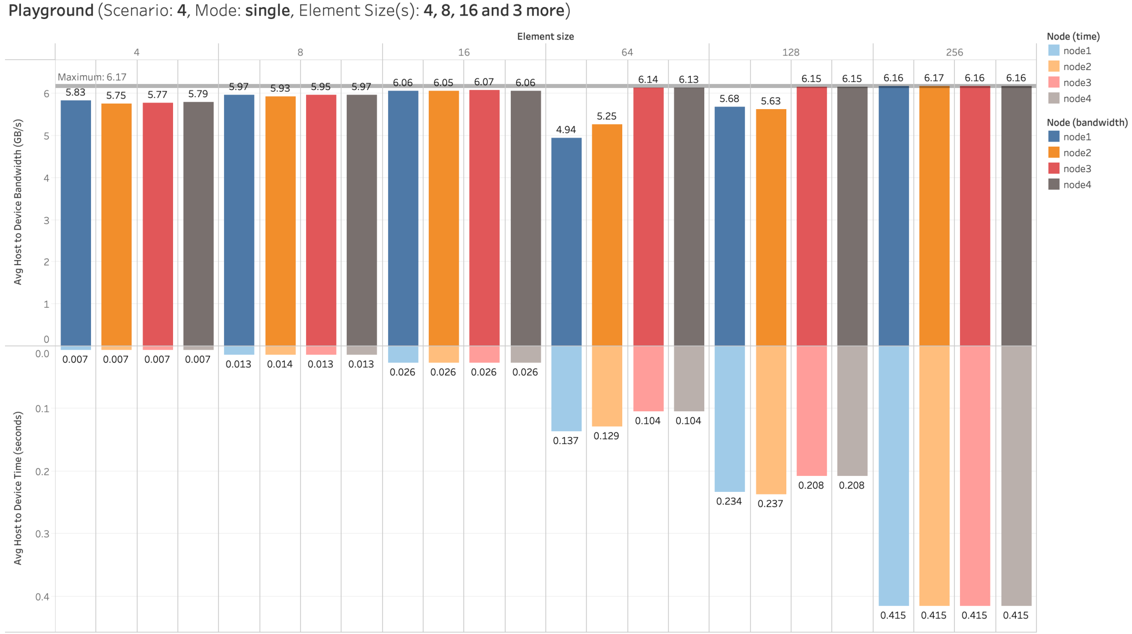

Figure 20: Transfer rate and time elapsed for 4 GPU-backed ML workloads (single mode), scenario 4

scheduled on the same physical CPU core. This effect is more likely to appear when using CPU affinity settings for each VM, which is the case for all scenarios except seven. Ultimately, single mode in scenario four suffered a 12% throughput penalty at a 64 MB element size and a 7% penalty at the 128 MB element size.

scenario five highlights the worst-case scenario, where no GPU-backed VM instance is in the same NUMA domain as its backing GPU device. Any CPU time or memory allocation has to cross between NUMA domains, and this must happen for all four GPU-backed VM instances In both modes, it can be seen that two of the four the GPU-backed VM instances are relegated to splitting the available bandwidth and the implications of this become worse as element size increases. Earlier in the results chapter, it was mentioned that it was important to determine not just if resources would be constrained, but how they would be split when those conditions occurred. Based on the results here and in scenario seven, it would appear that those GPU-

Playground (Scenario: **5**, Mode: **single**, Element Size(s): **4, 8, 16 and 3 more**)

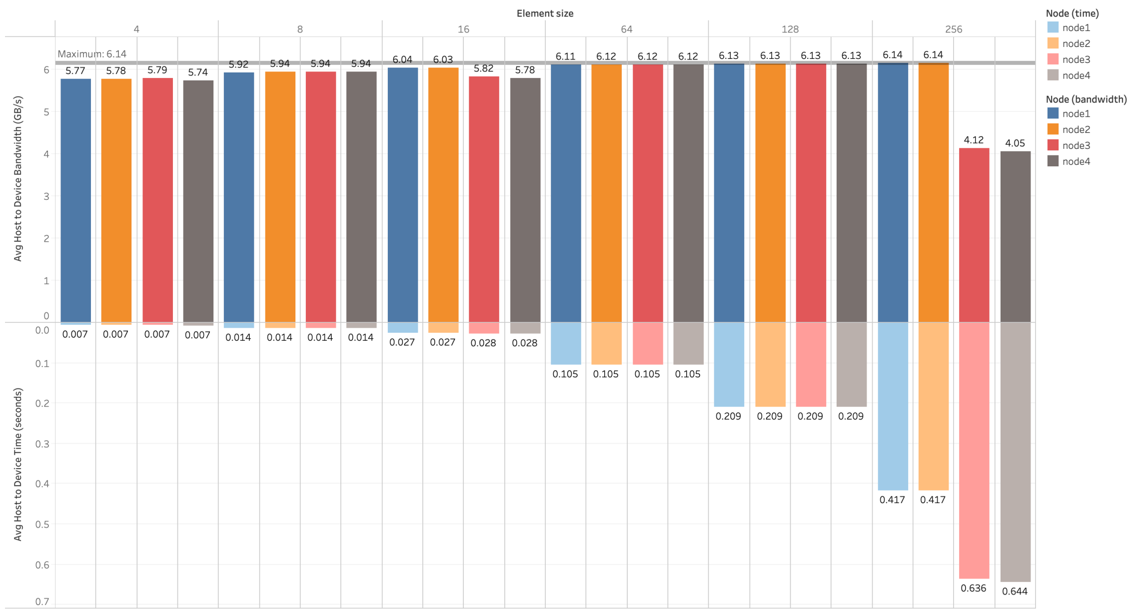

Figure 21: Transfer rate and time elapsed for 4 GPU-backed ML workloads (single mode), scenario 5

Playground (Scenario: **5**, Mode: **batch**, Element Size(s): **1024, 2048, 4096**)

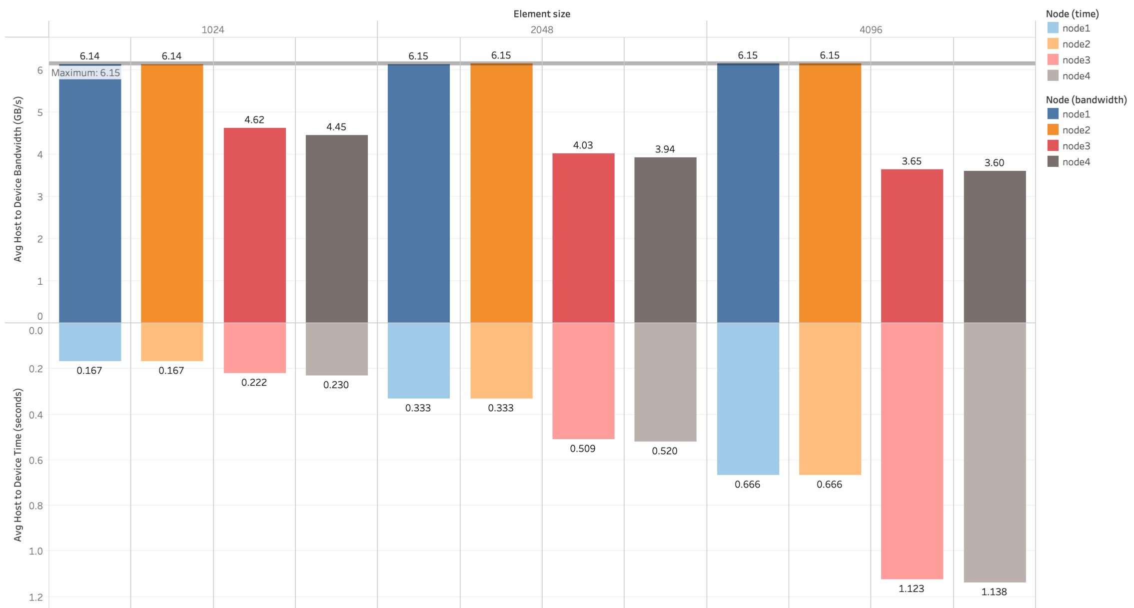

Figure 22: Transfer rate and time elapsed for 4 GPU-backed ML workloads (batch mode), scenario 5

backed VM instances that started first (even if it was only seconds earlier) will get their resource request fully fulfilled. Those VM instances that are left to compete end up splitting the remaining resources evenly. While it could not be shown with the equipment available, the expectation is that this effect would continue this trend as the number of VM instances as the number of GPU-backed VM instances increased.

Scenario six appears to be an arrangement of system resources that is easier for the hypervisor to handle, as all GPU-backed VM instances were able to complete their benchmarks at full speed. As only two of the four GPU-backed VM instances need to traverse into another NUMA domain, demand on the system bus is much lower than in scenario five. This scenario gives support to the notion that there are two configuration parameters that need to be taken into consideration when sizing these workloads. The first point is one that this thesis aims to prove: that taking measures to avoid breaking NUMA locality are worthwhile and improve outcomes for ML workloads. A second point is that while using CPU affinity to control for NUMA effects with GPUs, it is also important to take into consideration that using CPU affinity also makes work harder for the hypervisor CPU scheduling. This is shown when comparing outcomes from scenario four to six -— scenario four should have run at full speed, but NUMA isolation did not always lead to the expected outcome.

This chapter will close with discussion of scenario seven. In scenario seven, all CPU affinity for the VM instances was disabled and furthermore, each VM was assigned 24 virtual CPUs. As in previous scenarios on this hypervisor host, configuring 24 virtual CPUs for a VM instance causes the hypervisor to assign a NUMA architecture to the VM instance. Doing so allows the VM to be scheduled on either physical processor and NUMA domain, though the VM instance has no actual control over which physical CPU processes will be executed on. Scenario seven is intended to represent an unoptimized real-world scenario, where the GPU-backed VM instances

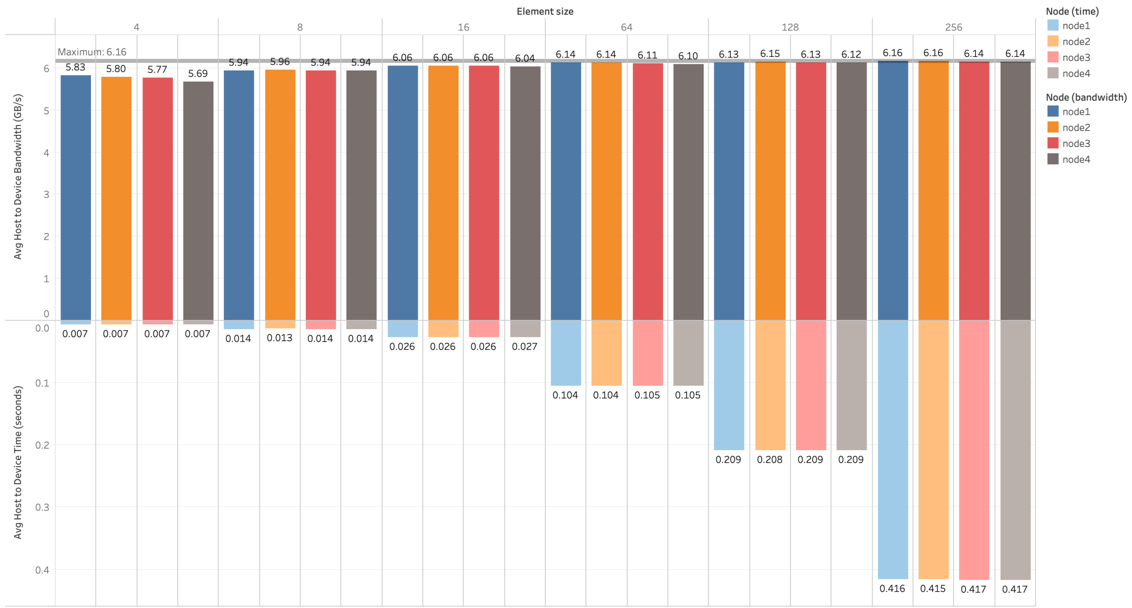

Figure 23: Transfer rate and time elapsed for 4 GPU-backed ML workloads (single mode), scenario 6

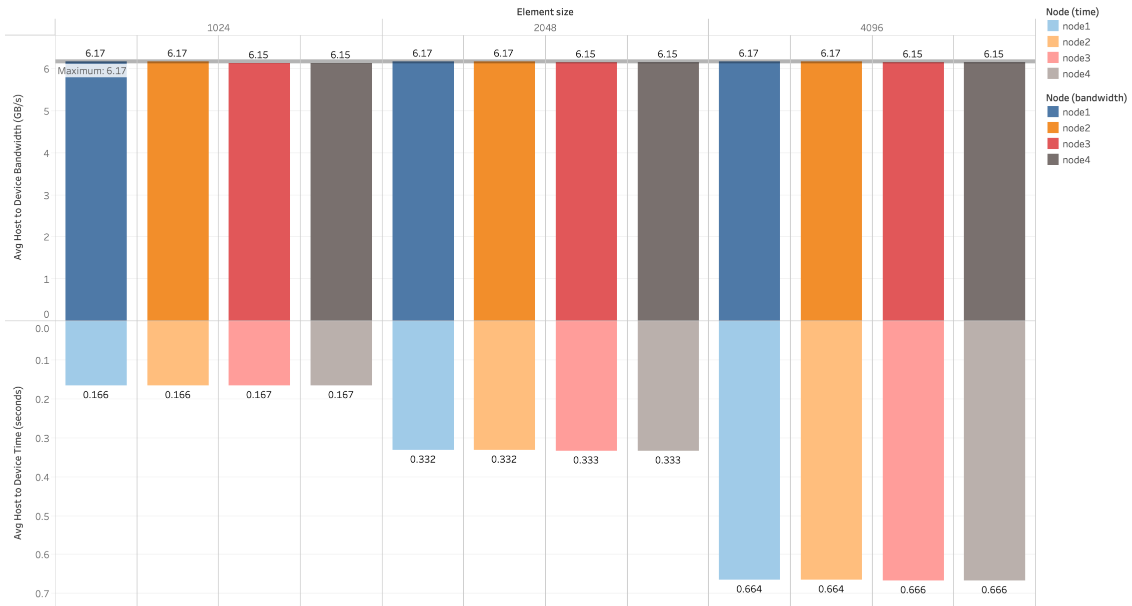

Figure 24: Transfer rate and time elapsed for 4 GPU-backed ML workloads (batch mode), scenario 6

can be scheduled and allocate memory in either NUMA domain, no CPU affinity is set on the VM instances, and as is the case in other scenarios — the hypervisor does not have visibility into the I/O requests to a peripheral device by a VM instance using pass-through mode.

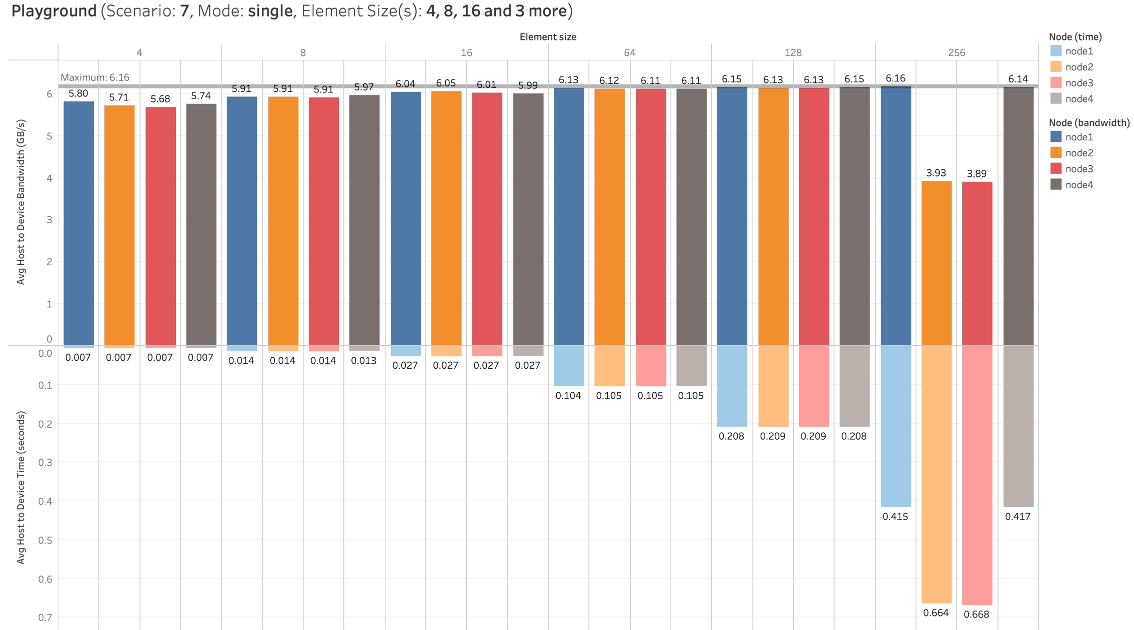

Figure 25: Transfer rate and time elapsed for 4 GPU-backed ML workloads (single mode), scenario 7

It would appear that in addition to being an unoptimized real-world scenario, scenario seven could also be considered a worst-case scenario. Results match those that were seen in scenario five, which was by design supposed to be a difficult arrangement of resources for the hypervisor to handle. Resource collisions occurred, and first consumer to those resources won while the other VM instances had to split what is left over.

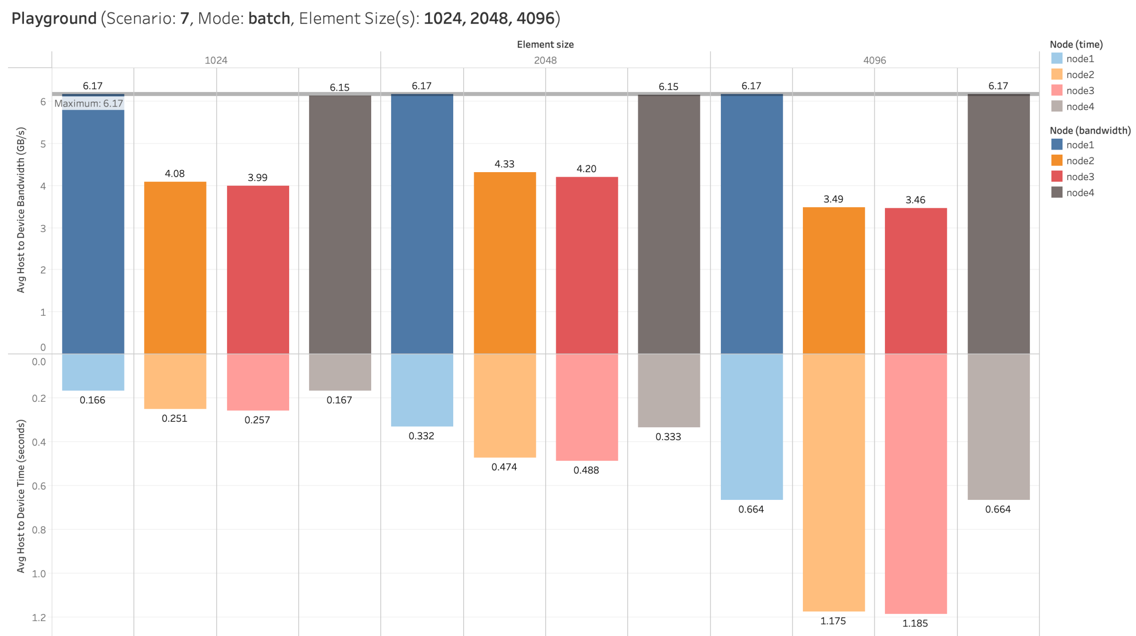

Figure 26: Transfer rate and time elapsed for 4 GPU-backed ML workloads (batch mode), scenario 7

Chapter VI.

Summary and Conclusions

This chapter provides conclusions based on the data from chapter V and recommendations for how to optimize workloads to avoid negative effects on throughput. The first section will summarize the results from the various experiments and contextualize why results occurred in a particular way. After the results summary, there is a section that goes into detail about the specific configuration parameters to use in order to minimize NUMA effects on ML workloads. These findings will also be listed in Appendix B, though without as much explanation. Finally, the chapter closes with a short summary of aspects to consider when conducting future research in this area.

6.1 Knowledge Gained

One of the surprises here is how little overall system load played a part in the results observed. System traffic directed to and from peripheral devices is the single biggest factor affecting PCIe speeds. It is only within the most dire of CPU usage circumstances that performance appeared to be affected, and the only real penalty was a slight ($\leq$ 0.5 sec) extra latency in system memory allocation times and/or CUDA context init/destruction tasks.

Furthermore, in the scenarios where only two GPU devices were in use it was difficult to see any significant effect on average PCIe bandwidth. In the cases where

bandwidth speeds were less than the maximum, the throughput was still only impacted by a few percentage points and may be flattened out over the timescale of a much larger data transfer. Once four GPUs were involved, there were more occurrences of 50% throughput penalties for half the GPU-backed VM instances. This gives support to the notion that as the number of devices utilizing the PCIe bus increases, the potential for impact to the performance of that bus increases in kind. The only situation where this wasn't observed was the alignment profiles where GPU-backed VM instances were constrained to the same NUMA domain where their backing GPU device resided. Another manageable situation is when least some instances were constrained (as shown in alignment profile five).

This effect is likely to be magnified as the number of GPUs installed in the system increases. The full extent of this factor was likely not able to be observed with the test systems available here due to the relatively low number of installed GPUs. These conditions are explained in the limitations and known issues section. With that said, the conclusion to be drawn here is that controlling for NUMA locality in devices installed in a pass-through context is important in order to maximize performance for all clients. As discussed in section 5.6.4, when pass-through devices are used by a VM instance the hypervisor loses visibility for that device. On the flip side of that, the condition in which the VM instance doesn't have any knowledge of which physical CPU its processes are executing on leads to the potential for poor performance when using pass-through devices. Manual intervention through setting NUMA domain affinity is likely the only tool available to administrators of hypervisor systems to mitigate this shortcoming at this time.

6.2 Configuration Recommendations

Armed with the knowledge gained from this thesis, the following are some recommended configuration parameters to apply to virtualized systems that are going to use GPUs. The first place to apply optimizations is at the system level.

In the system firmware management, power management should be set to off or maximum performance in order to limit CPU throttling and keep system thermal management from throttling down and putting extra thermal load on the GPUs. Virtualization extensions should be set to on, otherwise the hypervisor will run very poorly or not at all. Node interleaving should be set to off, as it allows each NUMA domain to be responsible for its own memory and devices rather than having the system create a unified memory map without regard to the memory's distance from its owning processor. The last required setting is that Single Root I/O Virtualization (SR-IOV) must be set to enabled or true. Briefly, SR-IOV allows the system to differentiate between the various PCIe functions being executed in the system and is necessary in order for a VM to take sole ownership of a pass-through device. Finally, if the system firmware provides an option to enable IOMMU, that should be enabled for the benefit on the hypervisor even if the GPU card is hypervisor-aware. The implications of this option are discussed in section 3.3.

At the hypervisor level, the GPUs to be used in pass-through mode need to be marked for pass-through. In ESXi, this can be set from the Host Configuration menu under Hardware, PCI Devices, Configure Passthrough. Changing this configuration setting requires a hypervisor host reboot.

VM instances require both some common and advanced configuration parameters. The first is that all VM instance memory must be reserved. This is a boolean option in the `vsphere` API, `memoryReservationLockedToMax = true`, but it can also be

toggled via a VM's configuration in the GUI. In order to use a pass-through device with a VM, the following options must also be set: `pciPassthru.use64bitMMIO = true`, and VM instance firmware should be set to use EFI. Finally, there is an advanced setting: `pciPassthru.64bitMMIOSizeGB`, whose value depends on how many cards are being used in pass-through mode and how much RAM they have. To determine the correct value for `pciPassthru.64bitMMIOSizeGB`, multiply the number of pass-through devices being attached to the VM instance by the amount of RAM on each card, rounded to the next power of two. The experiment environment used a value of 16, since multiple cards were used during development. There does not appear to be a penalty for over-sizing.

Appendix B has specific configuration parameters to be applied to the hardware system itself, the hypervisor, and the VM instances.

6.3 Limitations and Known Issues

There is one primary limitation with this thesis that bears mention. That is the fact that while the experiments used servers that have GPUs in them, there may not have been enough GPUs to truly enumerate the full extent of what was intended to be shown. With only four GPUs installed, there were situations missed in which most or all the PCIe lanes are in use, and it would've been a chance to further investigate how reliable the CPU, system management controller, and hypervisor are when a larger portion of the PCIe bus is under heavy use.

This thesis also limited the scope to just PCIe-based GPU adapters and ignored the impact of PCIe-based local storage or Ethernet adapters, both of which would likely play a part in a real-world application. It is expected that once PCIe-based network adapters or PCIe-based storage solutions are part of the workflow, bandwidth will be much more constrained. Non-volatile memory express, or NVMe,

disks are rapidly becoming the disk of choice for high-performance applications. NVMe disks, as they sit on the PCIe bus and take a 4x slot per disk can quickly add load to the bus. Including these devices in a comprehensive benchmark would likely lead to a better understanding of where other optimization points may exist in this entire workflow.

Finally, one other limitation in the data gathering process is that it was difficult to identify which CPUs VM instances were executing on due to the mismatch between VM instance CPU IDs and the actual physical CPU core seen at the hypervisor level. During the baseline-type scenarios it was easy to identify generally where VM instances were executing, and indeed, doing so led to the discovery in Chapter 5.1 (Baseline Results) section where it appeared as though the hypervisor was always favoring certain high-ID cores regardless of GPU location for a particular GPU-backed VM instance. When working through the scenarios with a more than two or three load generating VM instances, it quickly became overwhelming to track where a single VM instance was executing and eventually this data was too chaotic to be of use. Ultimately, during the development phase it was proven that one is able to trust CPU affinity settings to act as advertised on the hypervisor used in the experiment and this issue wasn't too much of a hindrance.

6.4 Lessons Learned

While the findings in this thesis give support to the idea that controlling for NUMA effects in ML workloads is an important configuration parameter to control for, there are ways in which this research can be taken further.

One of the primary limitations of the experiments performed is that the age of the GPUs made it impractical to run many contemporary ML benchmarks. It is unfortunate that Nvidia introduced the tensor core for their GPUs a mere generation

after the Maxwell-core GPUs used in these experiments. The Maxwell-based GPUs are not particularly adept at CUDA compute operations compared to devices that are available today. As a result, the decision was made to focus on the parts of a GPU which would be common to multiple generations of devices: PCIe 3.0.

Another limitation related to hardware is that the test systems did not feature any NVMe or even SATA-based local storage. These types of devices both use PCIe lanes as well and as a result, could be used as additional load-generating devices in order to further ascertain the degree of system bus congestion caused by loading data from disk to memory, and then onto the GPU.

With these types of modifications to the test criteria, it would be easier to generate results that could be compared to other platforms using common benchmarks. Benchmarks like those developed by the MLCommons project would have results comparable using modern hardware, likely even running on a hypervisor (Mattson, 2021). When attempting to run some more demanding MLCommons benchmarks on the test system, estimated completion times were estimated to be in excess of 24 hours in some cases.

Another angle worth investigating is how these results apply to other hypervisors and possibly to cloud-provider forks of KVM. As KVM is likely the most widely-used open source hypervisor, it is a high-value platform to investigate to see if the results observed here are reproducible on any hypervisor. Likewise, since KVM is a common hypervisor for major cloud providers, it is of interest to see if they perform this type of optimization and if not, see how the potential performance penalties could be avoided.

6.5 Future Considerations

Improving the capacity of the system buses and the amount of bandwidth that can be provided to peripherals is always an area of innovation in computing. Systems used in this thesis provide a mere 40 lanes for PCIe devices to use. Two 16x GPUs would consume over half of those available slots, and leaves few expansion slots available for high-speed Ethernet or fast local storage. These conditions have made it difficult for system integrators and data center administrators to achieve the consolidation ratios they desire, never mind ML developers who want to use lots of GPU and load data from network-based volumes.

The PCIe 4.0 standard was ratified in 2016, and the first motherboards and devices using it were released in 2020. PCIe 4.0 doubles the bandwidth available to the bus and should help push forward the adoption of technologies like 40 and 100 Gigabit Ethernet. Nvidia's 3000-series GPUs support PCIe 4.0 and the higher data transfer rate should help the card realize its potential for all sorts of applications. A specification has already been proposed for PCIe 5.0, which should bring a similar jump in capability in a few years.

Another development of note from the chip developers is the massive expansion of available PCIe lanes in CPU and motherboard designs. For example, two of the latest processors in AMD's Ryzen 3000-series CPUs not only have an industry-leading number of cores, but also include a massive number of on-CPU PCIe lanes. These new CPUs include 64 or even 128 PCIe lanes on certain models, more than doubling the number available from other x86 CPU vendors. The potential for being able to colocate multiple GPUs, Terabytes of fast PCIe-based storage and fast Ethernet without having to compromise the throughput of any single device is a compelling prospect for computing in general and machine learning specifically.

There are also many researchers who are developing new algorithmic approaches to machine learning that make use of the CPU power that's been ignored as GPUs and neural networks for ML have been at the forefront. A group of researchers at Rice University designed an algorithmic approach to neural network training called SLIDE (Sub-LInear Deep learning Engine) (Chen et al., 2020). SLIDE does depend on the presence of some vector-math enhancing instructions present in contemporary Intel CPUs, but they claim to achieve training and inference performance several times better than when using a single Nvidia V100 GPU. Algorithmic advances have the potential to swing the pendulum away from the reliance on GPU that have dominated machine learning and neural network design since AlexNet first appeared 2012 (Russakovsky et al., 2015).

The bottom line is that there is a march of hardware advancement as well as algorithmic research that is pushing the level of utilization in computing systems for machine learning. The future of this space will likely involve two different approaches to increase the overall utilization of hardware. First, innovation in the general purpose CPU space to compete with the GPU will likely lead to development of additional specialized CPU instructions for ML purposes or specially tuned algorithms that take advantage of both CPU and GPU at the same time. The second approach, which is already a reality today, are custom chip designs built with ML-like operations in mind. For example, Apple's M1 chip is more than just a general-purpose CPU -— it is built as a system-on-a-chip, which moves components like the CPU, GPU and system storage much closer together. Apple praises this design as being responsible for greatly enhanced ML workload performance, and while some custom CPU instructions come into play, there is no doubt that moving system components closer together has a high potential to minimize the effects observed in this thesis. Developments such as these are the kind of innovation that will likely be responsible for significant leaps

forward in ML workload performance, rather than the iterative steps that come with a new GPU chip or PCIe revision. Future research directions based on this thesis would do well to focus on studying how the system-on-a-chip design improves data movement to a GPU-like processor optimized for vector operations and what that means for both existing algorithms and those being written for this specific type of architecture.

In closing, the stated goal for this thesis was to test and demonstrate the actual cost for not properly taking NUMA into account in virtualized environments. A secondary goal was to also provide a set of configuration parameters for optimized performance of ML workloads that takes NUMA into account. As Chapter V showed, there is indeed a performance cost of to unoptimized alignments of GPU-backed VM instances ranging from a 5-10% penalty all the way up to a 50% throughput penalty. Fortunately, there are methods to mitigating these penalties, and those were presented in Chapter VI. The conclusions chapter offers both placement scenarios and configuration parameters for multiple components to help optimize placement of GPU-backed VM instances. Finally, it is clear that while the conclusions presented here help provide concrete evidence for why accounting for NUMA for GPU use in hypervisors is important, there is still work to be done, especially in systems with a greater number of GPUs and to see the potential benefits of system-on-a-chip designs. Additional paths forward can be found in 6.5.

References

AWS (2020). Amazon EC2 FAQs - Amazon Web Services. `https://aws.amazon.com/ec2/faqs/`, retrieved June 2020.

Chen, B., Medini, T., Farwell, J., Gobriel, S., Tai, C., & Shrivastava, A. (2020). SLIDE : In Defense of Smart Algorithms over Hardware Acceleration for Large-Scale Deep Learning Systems. *arXiv:1903.03129 [cs]*. arXiv: 1903.03129.

Chrabaszcz, P., Loshchilov, I., & Hutter, F. (2017). A Downsampled Variant of ImageNet as an Alternative to the CIFAR datasets. *arXiv:1707.08819 [cs]*. arXiv: 1707.08819.

Denneman, F. (2020). PCIe Device NUMA Node Locality. `https://frankdenneman.nl/2020/01/10/pcie-device-numa-node-locality/`, retrieved September 2020.

Gupta, S., Agrawal, A., Gopalakrishnan, K., & Narayanan, P. (2015). Deep Learning with Limited Numerical Precision. *arXiv:1502.02551 [cs, stat]*. arXiv: 1502.02551.

Harris, C. R., Millman, K. J., van der Walt, S. J., Gommers, R., Virtanen, P., Cournapeau, D., Wieser, E., Taylor, J., Berg, S., Smith, N. J., Kern, R., Picus, M., Hoyer, S., van Kerkwijk, M. H., Brett, M., Haldane, A., del Río, J. F., Wiebe, M., Peterson, P., Gérard-Marchant, P., Sheppard, K., Reddy, T., Weckesser, W., Ab-

basi, H., Gohlke, C., & Oliphant, T. E. (2020). Array programming with NumPy. *Nature*, 585(7825), 357–362.

Hashicorp (2021). Introduction to Terraform. `https://www.terraform.io/intro/index.html`, retrieved Feburary 2021.

Honig, A. & Porter, N. (2017). 7 ways we harden our KVM hypervisor at Google Cloud: security in plaintext. `https://cloud.google.com/blog/products/gcp/7-ways-we-harden-our-kvm-hypervisor-at-google-cloud-security-in-plaintext/`, retrieved June 2020.

Intel (2018). PCI Express High Performance Reference Design. `https://www.intel.com/content/www/us/en/programmable/documentation/nik1412473924913.html#nik1412473910415`, retrieved Jan 2021.

Jouppi, N. P., Young, C., Patil, N., Patterson, D., Agrawal, G., Bajwa, R., Bates, S., Bhatia, S., Boden, N., Borchers, A., Boyle, R., Cantin, P.-l., Chao, C., Clark, C., Coriell, J., Daley, M., Dau, M., Dean, J., Gelb, B., Ghaemmaghami, T. V., Gottipati, R., Gulland, W., Hagmann, R., Ho, C. R., Hogberg, D., Hu, J., Hundt, R., Hurt, D., Ibarz, J., Jaffey, A., Jaworski, A., Kaplan, A., Khaitan, H., Koch, A., Kumar, N., Lacy, S., Laudon, J., Law, J., Le, D., Leary, C., Liu, Z., Lucke, K., Lundin, A., MacKean, G., Maggiore, A., Mahony, M., Miller, K., Nagarajan, R., Narayanaswami, R., Ni, R., Nix, K., Norrie, T., Omernick, M., Penukonda, N., Phelps, A., Ross, J., Ross, M., Salek, A., Samadiani, E., Severn, C., Sizikov, G., Snelham, M., Souter, J., Steinberg, D., Swing, A., Tan, M., Thorson, G., Tian, B., Toma, H., Tuttle, E., Vasudevan, V., Walter, R., Wang, W., Wilcox, E., & Yoon, D. H. (2017). In-Datacenter Performance Analysis of a Tensor Processing Unit. *arXiv:1704.04760 [cs]*. arXiv: 1704.04760.

Keckler, S. W., Dally, W. J., Khailany, B., Garland, M., & Glasco, D. (2011). Gpus and the future of parallel computing. *IEEE Micro*, 31(5), 7–17.

King, C. (2017). Ubuntu Manpage: stress-ng - a tool to load and stress a computer system. `https://manpages.ubuntu.com/manpages/artful/man1/stress-ng.1.html`, retrieved November 2020.

Klöckner, A., Pinto, N., Lee, Y., Catanzaro, B., Ivanov, P., & Fasih, A. (2012). PyCUDA and PyOpenCL: A Scripting-Based Approach to GPU Run-Time Code Generation. *Parallel Computing*, 38(3), 157–174. arXiv: 0911.3456.

Kulkarni, S., Krell, M. M., Nabarro, S., & Moritz, C. A. (2020). Hardware-accelerated Simulation-based Inference of Stochastic Epidemiology Models for COVID-19. *arXiv:2012.14332 [cs]*. arXiv: 2012.14332.

MacEachern, D. (2021). vmware/govmomi. `https://github.com/vmware/govmomi`, retrieved December 2020.

Mattson, P. (2021). mlcommons/training. `https://github.com/mlcommons/training`, retrieved October 2020.

McKerns, M. M., Strand, L., Sullivan, T., Fang, A., & Aivazis, M. A. G. (2012). Building a Framework for Predictive Science. *arXiv:1202.1056 [cs]*. arXiv: 1202.1056.

Neugebauer, R., Antichi, G., Zazo, J. F., Audzevich, Y., López-Buedo, S., & Moore, A. W. (2018). Understanding PCIe performance for end host networking. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication* (pp. 327–341). Budapest Hungary: ACM.

Rodola, G. (2021). giampaolo/psutil. `https://github.com/smira/py-numa`, retrieved December 2020.

Russakovsky, O., Deng, J., Su, H., Krause, J., Satheesh, S., Ma, S., Huang, Z., Karpathy, A., Khosla, A., Bernstein, M., Berg, A. C., & Fei-Fei, L. (2015). ImageNet Large Scale Visual Recognition Challenge. *arXiv:1409.0575 [cs]*. arXiv: 1409.0575.

Smirnov, A. (2020). smira/py-numa. `https://github.com/smira/py-numa`, retrieved December 2020.

Walters, J. P., Younge, A. J., Kang, D. I., Yao, K. T., Kang, M., Crago, S. P., & Fox, G. C. (2014). GPU Passthrough Performance: A Comparison of KVM, Xen, VMWare ESXi, and LXC for CUDA and OpenCL Applications. In *2014 IEEE 7th International Conference on Cloud Computing* (pp. 636–643). Anchorage, AK, USA: IEEE.

Appendix A.

Glossary

Advanced Micro Devices AMD is a microprocessor development company for the x86 architecture. Notably, AMD competes not only in the CPU space with Intel, but also with Nvidia by selling designs to vendors for their Radeon line of GPUs. 8

Amazon Web Services Cloud computing provider wholly owned by Amazon and launched in 2006. Provides a full suite of computing services such as VM instances, storage, databases, networking, containers and more. 5

Citrix Citrix provides application and desktop virtualization via an ever-changing matrix of product and service lines. Citrix is notable to this thesis as it was the steward of the Xen hypervisor for many years, having acquired the rights to the hypervisor from XenSource in 2007. 5

CUDA CUDA is a C-like language used to write programs, or kernels, for execution on GPUs. The CUDA language is exclusive to Nvidia devices. 5, 28

ESXi A type-1 hypervisor developed, maintained, and sold by VMware, Inc. Used heavily in the private cloud and datacenter space. VMware, Inc. is generally considered to be first to market with an x86 server virtualization product in

2001 with VMware GSX (which eventually was renamed ESX). "ESXi" is an acronym, elastic sky X integrated, though it is not used in any product documentation or marketing.. 5, 6, 21, 66

Google Cloud Platform Cloud computing provider operated by parent company Alphabet, launched in 2008. Provides a full suite of computing services such as VM instances, storage, databases, networking, containers, etc. 5

GPU See Graphics Processing Unit. 2–6, 9, 19, 20

Graphics Processing Unit A peripheral device that contains one or more processors designed to perform floating point calculations at high speed. GPUs initially came to market as a device for accelerating computer graphics for gaming and 3D applications. Their specific aptitude for vector math made them a popular tool for training neural networks and performing blockchain-related operations. 1, 2, 78

hypervisor Software that provides acts as the governor of I/O passing between logical and physical resources in a computer system. There are two recognized schemes – Type-1 (bare-metal) and Type-2 (hosted) hypervisors. Type-1 is effectively the operating system installed on the hardware and has direct management of the system resources. Type-2 is a process running in another operating system and must manage I/O using the OS-provided methods. 1, 4–6, 8, 9, 12

Intel A historically dominant CPU design company. Intel is one of the primary vendors developing microprocessors using the x86 architecture. 8

KVM KVM: Also referred to as "kernel-based virtual machine", KVM is considered a type-1 hypervisor, though it exists as a module for the Linux kernel (though

it has ports to many other kernels and platforms). KVM is distributed under an open-source license. KVM is the hypervisor of choice for a number of cloud platforms such as GCP and AWS. 5, 6

machine learning Machine Learning – an area of research that aims to have computers aid in decision making by applying statistical models and ever-larger sets of training data. The aim is not necessarily to have a pre-determined solution for every question, but rather for the computer be able to use inference to answer a question based on knowledge of previous situations. This makes it an ideal mechanism for processing streams of data that can evolve over time or have large variation, such as facial recognition or spam filtering. 1, 79

ML See Machine learning. 1, 3, 9, 19

Mobile PCI Express An implementation of the PCIe standard that defines a different form factor for peipheral cards that makes it easier to integrate them into smaller computers such as laptops or blade servers. 23, 79

MXM See Mobile PCI Express. 23, 24

neural network Neural networks are data systems that model the arrangement of neurons in the brain, with the intention of emulating the ability of biological nervous system to process inputs and learn how to act, respond, or recognize future unknown inputs. 1

Non-Uniform Memory Access A design component used in multiprocessor systems to define memory access rules between different processors. 4, 20, 79

NUMA See Non-Uniform Memory Access. 4, 5, 11, 12, 20, 21

Nvidia Perhaps the most prominent name in GPU development today. Nvidia produces and sells GPU designs to vendors for gaming cards while also manufacturing and selling its own line of maching learning and artificial intelligence accelerators for data center use. The company develops and maintains the CUDA language, sponsors ML research as well as retaining an in-house research division, and contributes to other software projects that use its accelerators. 6, 9, 16, 23, 27, 32, 38, 40, 70, 71

OpenCL A language used to provide a standard interface for writing programs, or kernels, that execute on GPUs. OpenCL is an alternative to CUDA and is available for a far greater number of devices, though at the cost of not being able to take advantage of hardware-specific optimizations. 5

paravirtualized Paravirtualization is a design technique used for VM instances where rather than emulating all the device functions of a peripheral device such as an Ethernet adapter or storage controller, the hypervisor provides a piece of kernel software for the VM instance that offloads processing of those types of commands to the hypervisor itself, reducing CPU load on the VM instance. 12

PCIe See Peripheral Component Interconnect Express. 4–6, 9, 11

Peripheral Component Interconnect Express An electrical and physical connection specification for adding expansion devices to a computer. PCIe defines multiple slot sizes (commonly 1x-16x) that determine the throughput of the connected device. PCIe is the most common connection standard today and is used for everything including storage, networking, and graphics expansion devices. 2, 80

virtual machine A virtual machine is an abstraction of a complete operating system install. In other words, a VM is a bundle of files that describe the hardware resources of the VM and contains its entire filesystem. In concept, VMs should be abstract enough to be portable to different platforms and hypervisors. In practice, a VM is strongly tied to the hypervisor that created it, especially the more a VM takes advantage of hypervisor-specific optimizations for accessing the underlying hardware. 2, 81

virtualization Generally meant to describe a scheme that divides a concrete resource into one or more logical instances of the concrete one. For the purposes of this document, virtualization refers to the splitting of physical computing resources into multiple logical copies of that system. This allows the user to execute multiple instances of an operating system on a single piece of hardware as virtual machines. 1–5

VM See Virtual machine. 2–5, 12, 20

Xen Xen: A type-1 hypervisor used by multiple cloud vendors (notably AWS) and in commercial hypervisor distributions, such as Citrix's XenApp and XenDesktop. Xen is freely available as an open-source project, but commercial derivatives are sold by Citrix due to them having acquired XenSource, the company that holds the various technology patents for Xen in 2007. 5

Appendix B.
Configuration Settings

B.1 VM Configuration

The following configuration parameters are added/changed in the VM instance's `vm_name.vmx` config file.

```
memoryReservationLockedToMax = true
pciPassthru.use64bitMMIO = true
pciPassthru.64bitMMIOSizeGB = 16
firmware = efi
sched.cpu.affinity = "all" or ["0-39", "40-79"]
```

B.2 Hypervisor Configuration

The following configuration parameters are added/changed in the ESXi configuration.

```
GPU Device(s) = Configuration, Hardware, PCI Devices, Configure Passthrough (
checked)
```

B.3 Hardware Configuration

The following configuration parameters are set in the System Configuration tool.

```
Node Interleaving = false/disabled
Single Root I/O Virtualization = true/enabled
IOMMU (if present) = true/enabled
Power Management = Off, Maximum Performance
Virtualization Support = enabled
```