# Microservice-Oriented Distributed framework for resource intensive workloads


By

**Muhammad Ehtisham Mubarik**

Fall2019MSCS900000318277

Supervisor

**Dr. Asad Waqar Malik**

Department of Computer Science

School of Electrical Engineering and Computer Science (SEECS)

National University of Sciences and Technology (NUST)

Islamabad, Pakistan

(December 2021)

# THESIS ACCEPTANCE CERTIFICATE

Certified that final copy of MS thesis written by Mr *Muhammad Ehtisham Mubarik*, (Registration No 0000318277), of SEECS has been vetted by undersigned, found complete in all respects as per NUST Statutes/Regulations, is free of plagiarism, errors and mistakes and is accepted as partial fulfillment for award of MS degree. It is further certified that necessary amendments as pointed out by GEC members of the scholar have also been incorporated in the said thesis.

Signature: _____

Name of Supervisor: _____

Date: _____

Signature (HOD): _____

Date: _____

Signature (Dean/Principal): _____

Date: _____

# Approval

It is certified that the contents and form of the thesis entitled "Microservice-Oriented Distributed framework for resource intensive workloads" submitted by *Muhammad Ehtisham Mubarik* have been found satisfactory for the requirement of the degree

Advisor: _____

Signature: _____

Date: _____

Committee Member 1: _____

Signature: _____

Date: _____

Committee Member 2: _____

Signature: _____

Date: _____

This thesis is dedicated to *my beloved parents* and *my lovely grandmother (LATE)*

# Certificate of Originality

I hereby declare that this submission titled "Microservice-Oriented Distributed framework for resource intensive workloads" is my own work. To the best of my knowledge it contains no materials previously published or written by another person, nor material which to a substantial extent has been accepted for the award of any degree or diploma at NUST SEECS or at any other educational institute, except where due acknowledgement has been made in the thesis. Any contribution made to the research by others, with whom I have worked at NUST SEECS or elsewhere, is explicitly acknowledged in the thesis. I also declare that the intellectual content of this thesis is the product of my own work, except for the assistance from others in the project's design and conception or in style, presentation and linguistics, which has been acknowledged. I also verified the originality of contents through plagiarism software.

Student Name: Muhammad Ehtisham Mubarik

Student Signature: _____

# Acknowledgments

In the name of Allah, the most beneficent and merciful. First of all, I would like to thank Allah Almighty for his countless blessings and my parents who were always there for me.

I would like to express my gratitude to my supervisor Dr. Asad Waqar Malik for his support and guidance through out my thesis work and my committee members for their valuable suggestions.

I would like to thank Dr. Jeremy Murray for his continuous involvement with my work his help and support has played an important role in completion of my thesis. I would also like to thank my brother and colleague Muhammad Husnain Mubarik who was always available whenever I needed him.

# Contents

# List of Figures

# List of Tables

# List of Abbreviations and Symbols

## Abbreviations

| | |
|---|---|
| **K8s** | Kubernetes |
| **K3s** | Light weight Kubernetes |
| **GPU** | Graphical Processing Unit |
| **VM** | Virtual Machine |
| **OS** | Operating System |
| **PCIe** | Peripheral Component Interconnect Express |
| **I/O** | Input/Output |

# Abstract

Fixed allocation of GPU resources to virtual machines increases idle time in utilization of GPU resources when the workloads are being executed on a different machine and increases the cost of hardware as it requires GPUs for every virtual machine. Recent solutions optimise scheduling algorithms in container orchestration environments to distribute workloads across machines having GPUs directly attached to them. However if the workloads are distributed across different machines but require GPU for short periods, GPU resources will stay idle on different machines for remaining time and that results in increased cost and under-utilization of the available resources. To address this under-utilization problem we present a framework to arrange available resources in a way that it allocates GPU to a machine only when required for processing and after processing that GPU can be shared with other machines for their workloads. The Key of our framework is to create a pool of all the available GPUs and then reserve a GPU for workload if it requests processing and add that GPU back to the pool of available resources once released from workload. Therefore, this framework assures the maximum utilization of GPUs with minimum available resources that results in significant decrease in cost as well. Furthermore this framework proposes integration container orchestration through kubernetes, provisioning resources and managing kubernetes clusters through Rancher. This provides an end to end infrastructure to deploy workloads in a containerized environment and improve utilization of available resources. Experiment results show that with our approach there's very little overhead with time but we do not need to directly attach GPU on each virtual machine to execute workloads.

**Keywords:** *Resource Under-utilization, GPU, Containers, Kubernetes, Rancher*

Chapter 1

# Introduction

With the increasing demand of cloud based deployments, containers and container orchestration platforms have gained immense popularity. Containers are lightweight virtualization platforms and are highly appreciated by microservices based architectures. Developers adapted to the microservices and containerization practices to develop applications and containers provided enhanced portability of application, speedy delivery and synchronization in the environment. Cloud infrastructures also demand containerized environments for cost, deployment, delivery and various other reasons. A lot of work is carried out on distribution of underlying resources to run the microservices and gain maximum throughput.

With the involvement of containers in almost every development life cycle, AI/ML workloads and applications that use GPUs to accelerate processing started adapting to microservices and containerization platforms very quickly. These workloads require GPUs to execute some operations. GPU has a key role in the cost management of any architecture and it is important to maximize the utilization of available GPU resources. With the approach of distribution frameworks, scheduling the available GPUs and sharing resources across different workloads became a challenge. Kubernetes clusters have different worker machines which are used to run containerized workloads and are responsible for the execution of jobs. But to run the workloads with the requirement of a GPU worker machine must have a GPU attached to it. Scheduling algorithms are highly effective and are required to share the GPU to the containers running on the same worker machine. But with orchestration clusters there are multiple machines where workloads are distributed and containers run. If applications are designed with the requirement

that one microservice requires GPU resources for a very limited amount of time and then GPU can be used by other microservices or some other application running in the same cluster, we cannot schedule workloads on machines having No GPUs. So we either have to buy more GPUs for other machines as well or schedule most of the workloads on limited machines compromising the efficiency, utilization and costs.

In our work, we present a solution that enables workloads running on different machines in the same cluster to utilize the same GPU resources at different times. We create a pool of all the GPUs available in our servers and then a centralized server is responsible to manage and execute GPU oriented tasks of each machine. Every workload distributed across different machines in a kubernetes cluster requests for a fixed number of GPUs from the GPU server and once the operation is completed and GPU is no more required server can add the GPU back to the available pool. Now when some other workload requires a GPU, since the GPU was released from the previous machine, the same GPU can be utilized for the new machine and a new GPU for the new machine is not needed. This way, by using a centralized pool of GPUs we can share GPUs on worker machines running in the same network and increase utilization of available GPU resources and significantly decrease cost and idle time.

## 1.1 Problem Statement

The superposition of artificial intelligence, data analytics, and IoT has aided the development of sophisticated, intelligent, and computationally complex algorithms. These algorithms require significant computing power to perform as they are desired to. Therefore, executing these applications using traditional design frameworks leads to performance bottlenecks. Thus, to alleviate the performance bottlenecks of these computationally extensive workloads, there is a need for a framework which efficiently maps these applications on the underlying hardware with the goal to achieve maximum efficiency and alleviate resource under-utilization.

## 1.2 Purpose

The main purpose of this research is to develop an approach to enhance the efficiency of resource intensive complex workloads and increase the utilization of available hard-

ware resources. Organizations who have distributed application clusters across regions in different time zones or require GPUs for certain operations across workloads in sequential or non-conflicting manners face a lot of GPU dead cycles when they have GPU for every machine that runs a GPU oriented workload. Therefore, this solution will help organizations to increase the utilization of GPU for those microservices and avoid idle time significantly. This will eventually help them significantly improve their cost of underlying hardware resources. The other main purpose of this framework was to provide an end to end deployment infrastructure from application and hardware provisioning aspects for the containerized AI/Ml workloads in a distributed environment.

## 1.3 Objectives

The objectives of the proposed framework are

1. Alleviate GPU under-utilization in distributed frameworks
2. Share GPU resources in distributed systems
3. Alleviate performance bottlenecks for resource intensive workloads
4. Cost effective hardware resource management

## 1.4 Background

Distributed system is an environment where the tasks are distributed across different machines or networks or devices[1]. Distributed systems plays an essential role avoiding single point of failures and helps with parallel computation or processing for complex applications. Distributed programming or computing frameworks provides an environment for developers where they have predefined steps and faster development process. Cloud computing and distributed systems made way for numerous other techniques and patterns for the development of applications to make them salable, cost effective, easy to manage and various other reasons. Microservices design pattern is one of the architectural design patterns which gained high popularity over the traditional big monolithic applications because of its various advantages. Microservices are decomposed sections of one big application which are executed separately and coordinate with each other to

form one big application or perform a main course of actions. These decomposed parts of an application gave high popularity to the concept of containers. Continuous delivery, reduced complexity, resource isolation and many other requirements which play an important role for the success of microservices architectures are successfully achieved by containers. Containerization is the concept of lightweight virtualization. Where a small package for a microservices is created which has all the dependencies required to execute the microservice and that is just about everything in that package. No individual host or kernel requirements. With this lightweight packaging, microservices became highly portable, it became easy to manage multiple environments for applications. This led to high success of containers in a short time and every organization started adapting to this new set of practices for ultimate success. With the high success of containers, container orchestration platforms e.g. Kubernetes took over the infrastructure designs for both on premises and public cloud deployments for distributed frameworks. Clusters of multiple machines are configured and containerized workloads are deployed on the machines with the required set of resources to execute those distributed workloads and communicate with each other to complete the functional requirements of the applications.

With the high success of containerized microservices architectures, heterogeneous programming systems also started adapting to the concept of containerization. AI/ML or other resource intensive applications are designed to use GPUs as accelerators in parallel with CPUs to perform complex operations. As container orchestration platforms are used to distribute containerized workloads across machines, availability of GPU resources for the architecture, optimized distributions of workloads across machines and resource sharing are some critical challenges for resource intensive applications. Orthodox or conventional methods can cause exhaustive cost increments and bad utilization of the available resources which in return will be a pathway for performance bottlenecks and bad management. Scheduling algorithms are introduced from time to time to improve sharing the GPU resources across multiple containers for workloads to increase efficiency and utilization. These algorithms are important for GPU sharing but cannot contribute to another problem that often gets neglected is that for these algorithms machines must have GPUs directly attached to them. But if the workloads are designed in a way that microservices require GPU for very short periods or certain tasks but in spare time other microservices require GPU for their operations organizations have to bear high costs of increased number of GPUs which are required for short patches of

time and are idle for remaining time.

To solve the problem of sharing GPU resources for complex architectures, there is also a need for pre-defined architecture to manage hardware resources and a set of components that can complement functional behaviour of other complements and enhance overall productivity of the architecture which in the end is essential for productivity of any design pattern. We carried this research to work on these two main problems and provide a comprehensive framework using Vmware components and Rancher server is used to provision resources for container orchestration and deploy microservices on that platform. In the following sections a comprehensive description is available on how multiple components are integrated for this framework and how applications running in a distributed system can access GPU resources from a pool instead of a locally attached GPU to accelerate their complex computations and achieve their results in an efficient and timely method.

## 1.5 Thesis Outline

Rest of the thesis document is divided into 5 chapters. In chapter 1 we have explained the introduction of the research domain, problem statement, purpose, objectives and background of the research. In chapter 2 we have included related work done on distributed frameworks, microservices and GPU resource utilization. Chapter 3 explains our approach for research, implementation and write-up of this thesis. Chapter 4 presents a comprehensive overview of all the required components, their integration with each other and an architecture of the framework formed as a result of the combination of all those components. Chapter 5 contains the experiments that we performed on conventional architectures and proposed framework their results and a short evaluation of framework based on those results. Chapter 6 provides the conclusion and future direction in this domain.

Chapter 2

# Literature Review

Key concept of virtualization is adding a layer of abstraction on underlying hardware resources. There are two kinds of virtualization, hardware virtualization and application virtualization. A manager or software component is used to provide virtualization and that could be considered as an additional complexity of virtualization but virtualization helps getting maximum output from limited underlying hardware resources[2].

Specific computing applications can cause mismatch with the algorithms provided by legacy operating systems. These mismatches cause significant performance degradation. The support for the virtualization in hardware architectures enables new methods to execute computing applications without losing the support of existing OS[3].

Virtualization is becoming a key variant for various IT enterprises. Containers is an emerging approach for virtualization. It is a lightweight virtualization where we do not need a complete host but the package required to run the microservices and multiple microservices can be managed on one machine without occupying too many resources and a short abstraction over the host OS. After performing various tests on microservices deployed on containers and VMs it was evaluated that containers are better in every aspect compared to virtual machines. These performance tests were carried out using different benchmarking tools including Sysbench, Phoronix and apache benchmarking[4].

The rise of complex architectures with microservices and the need for ever shorter deployment life cycles, continuous delivery and high cost of resources from different cloud environments gave rise to the need of containers[5]. Lightweight virtualization or containers provide near native performance. Docker, a platform for managing containers, provides abstraction between different containers and restricts those containers from

accessing host OS but containers still depend on the host OS kernel and could be vulnerable as they do not have any OS.

For edge devices where low cost low computation is required containers and microservices are beneficial in many ways. Autoscaling, enough computation and flexible deployment can be achieved even with all the challenges at hand[6]. Dynamic distribution or deployment of microservices is still a big challenge but autoscaling of the containers is feasible using a fuzzy controller.

Integration, feasible scaling and isolated nature of containers has made them popular in recent times and edge devices have also become powerful enough to run containerized workloads[7]. This calls for the need of optimized placement of containers on devices or underlying machines. Container orchestration tools are still resource intensive so FLEDGE, a container orchestration tool, is introduced. It is based on Kubernetes but could be considered as a lighter version for edge devices. Where networking orchestration and placement methods are customized and then evaluated against k3s and kubernetes clusters by deploying workloads on edge devices.

Sometimes native kubernetes algorithms schedule workloads unequally distributed across the environment. This [8] approach proposes a hybrid shared state scheduling framework. Main focus here is diverted towards unscheduled or unprioritized tasks and decisions are based on state management of complete clusters continuously updated through master. By employing a synchronized state in a kubernetes cluster conflicting jobs are avoided which leads to better scheduling and utilization of available resources in a kubernetes cluster compared to native algorithms.

The SLATE (Services Layer at the Edge) is a framework for monitoring and security purposes of container orchestration through kubernetes clusters[9]. It involved a combination of monitoring tools like prometheus prometheus operators and thanos. The purpose of SLATE is to provide visibility of resource utilization and deployment infrastructures of applications on independently running kubernetes clusters on different networks.

Kubernetes is one of the most widely used open source platform used for container orchestration even though orchestration of simple microservices is optimised to a good extent by now but scheduling GPUs is still a big challenge that requires optimisation[10]. Gaia is a topology based scheduling framework for kubernetes clusters. It is based on

traditional GPU scheduling algorithms. In Kubernetes GPU sharing a GPU can only be fully allocated and this results in waste of resources and high costs and low performance as well. In Gaia GPU sharing is converted into a resource-access cost tree. It helped achieve optimal scheduling of GPU clusters and has improved the performance by 10% compared to traditional sharing algorithms.

Applications nowadays want to make the best use of their parallel architectures by integrating use of GPUs[11]. Different Deep Learning or even simple video processing applications want to use GPUs to accelerate their processing and increase performance. Kubernetes, Docker and other platforms are continuously working on the optimized usage of available GPU resources inside containers. KubeCG is an algorithm based on heterogeneous kubernetes clusters to make use of CPUs and GPUs in parallel. This scheduler takes into account multiple matrices of kubernetes pods and containers and schedules them on available GPU machines accordingly. Tests have revealed approximately 64% less time required to perform tasks if KubeCG is used to schedule workloads on a kubernetes cluster compared to native kubernetes clusters.

Due to high performance requirements of different applications, maximum throughput of GPU resources from the underlying architectures is considered a key factor[12]. GPU sharing across multiple containers in kubernetes clusters is a challenge as native kubernetes scheduling algorithms do not allow sharing GPU resources between different pods which leads to the big problem of resource under-utilization. Kube-Share extends native kubernetes GPU sharing algorithms with fine-grained GPU allocation. Kube-Share allows to schedule GPUs as first class resources in a kubernetes cluster. GPU sharing through KubeShare increases the overall system throughput with a minimum overhead on the execution or initialization of containers inside a cluster making it a better approach to use compared to native kubernetes algorithms for GPU sharing inside a cluster.

Chapter 3

# Methodology

In our previous chapter we did literature review. We gathered information on the use of microservices and their role for proficient use of underlying resources. We discussed the role of Kubernetes and GPU's in compute sharing and acceleration. We have also discussed how critical optimised utilization of resources is in case of healthcare applications because of their critical nature.

This chapter presents a detailed methodology on how to create a framework which efficiently maps applications based on their underlying hardware resources to achieve maximum throughput.Fig. 3.1 provides technical workflow of framework implementation.

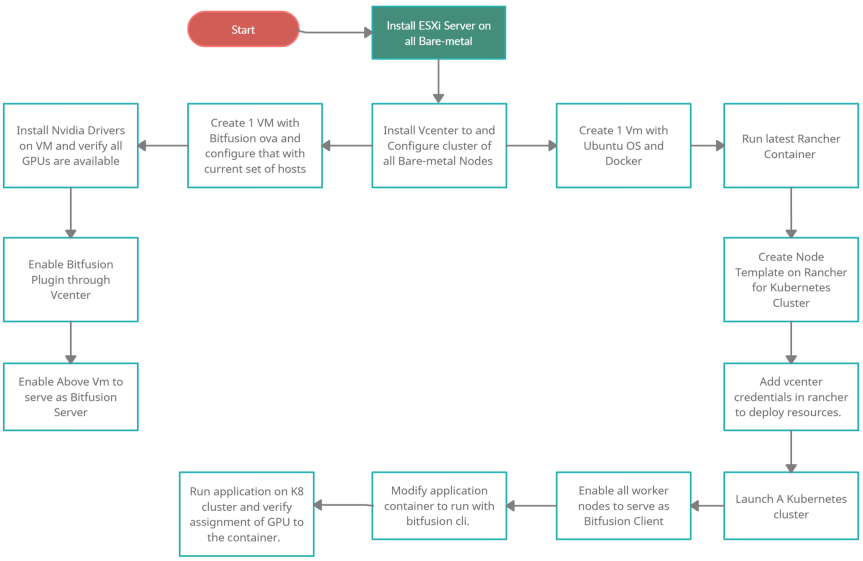


**Figure 3.1:** Flow Diagram for Framework

## 3.1 Literature

We started our work with literature review. In literature review we realized there are multiple frameworks which help GPU acceleration through Kubernetes but No particular framework is officially available to share GPUs across all Kubernetes nodes for applications to utilize available GPU resources if need be.

## 3.2 Hypervisors

After literature review, we evaluated two different hypervisors for the framework we planned to create. For this evaluation we worked on Xenserver Hypervisor and Vmware ESXi Hypervisor and opted for the later for it's unique property of allowing to share GPU resources across bare-metals through the network not to mention it's compatibility with other required tools. e.g. Rancher.

## 3.3 Deployment

After deciding to use Vmware Hypervisor for our framework we went through the following steps.

1. Install ESXi on all servers.
2. Install vcenter on one ESXi server.
3. Configure vcenter to create a cluster of all available ESXi servers.
4. Install and configure Bitfusion server to create a cluster of GPUs across all bare-metals.
5. Launch Kubernetes cluster from Rancher. add rancher details separately
   (a) Create 1 vm on ESXi 1.
   (b) Launch a minimum Rancher 2.0 container on that vm.
   (c) Create a Node template inside rancher.
   (d) Configure Cloud credentials for rancher to deploy k8 on vcenter.
   (e) Launch K8 cluster with above template and creds through rancher on vcenter.

6. Enable Bitfusion Client on all nodes created by rancher.

## 3.4 Verification

After creating the framework we installed Bitfusion CLI inside application containers and configured applications to run through Bitfusion CLI inside containers. Then deployed these modified applications on the Kubernetes cluster and verified through Vmware Vcenter that requested GPU resources are allocated to our application containers even if they are running on Worker nodes having No GPU resource attached to them.

## 3.5 Experiments and Results

After the installation and verification part was completed we executed cuda Matrix Mul and cuda Matrix Multiplication cuBLAS workloads on our framework and also on a conventional GPU allocated cluster. We compared the results and evaluated how this framework helps alleviate the problem of resource under utilization with negligble overhead.

## 3.6 Write-up

In our final step, we wrote down a detailed introduction for the set of tools and services being used. We provided a comprehensive literature review in Background. We wrote all our findings and detailed steps on how to recreate the proposed framework explained with Diagrams and then we verified successful allocation of GPU resources to Non-GPU Virtual Machines. After verification we executed cuda 11 provided scripts to run 2 experiments on vm having directly attached GPU and our framework. We compared our results and described how with slight overhead we can minimize resource under utilization problem. Then we concluded this thesis.

Chapter 4

# Framework Explanation

In the previous section, we discussed the methodology of the research work we carried out for this thesis. In this section we will discuss comprehensive details of the proposed framework.

Comprehensive details include the set of tools and technologies being used for this framework, where and how those tools and technologies are installed and how they communicate with each other to make this framework work.

## 4.1 Components

Set of components required to create this framework include VMware ESXi, VMware vcenter, VMware bitfusion, ubuntu, docker, rancher, kubernetes.

### 4.1.1 VMware ESXi

VMware ESXi is a hypervisor solution to completely manage processors, memory, storage and other resources available on bare-metal servers and their abstractions for the virtual machines. We can also create snapshots and store certain stages of those virtual machines as a backup in case those are required later on.

We installed VMware ESXi version 7.0 Update 2 on our bare-metals. This helps with the abstractions of resources for virtual machines we need to create in the later part of the framework.[13] [14] We use those machines to deploy applications and then utilize resources to get maximum output.

### 4.1.2 VMware Vcenter

VMware vcenter is a management tool provided by VMware. It helps manage resources across ESXi hosts and in some ways act as a single point of reference to manage multiple ESXi servers.[15] It also has additional plugins to manage some resources, monitor ESXi hosts for their various states and allows us to perform everything we need from a hypervisor.

We are using VMware vcenter server appliance version 7.0 update 2 to manage all our ESXi hosts. Once the deployment on VMware vcenter is completed we do not use ESXi hosts directly for any further creation of resources or anything else. We create resources, configure plugins, and connect to resources only through vcenter after first deployment.

### 4.1.3 VMware Bitfusion

VMware Bitfusion virtualized GPUs for AI and ML applications. It provides sharing of GPU resources across virtual machines over the network. It has three main components.[16]

- **Bitfusion Server**

  Bitfusion server is installed on the ESXi host. It is a pre configured virtual machine that requires direct access to local GPUs. These GPUs are connected through DirectPath I/O to this vm. It intercepts CUDA calls made by applications, processes them and returns to the application running on Bitfusion Client.

- **Bitfusion Client**

  Bitfusion Client is installed on virtual machines where we need to run our application which require GPU resources for their processing. These Virtual machines are also enabled to function as bitfusion clients through Bitfusion Plugin.

- **Bitfusion Plugin**

  The Bitfusion server registers bitfusion plugin to VMware vcenter. This plugin registers virtual machines acting as bitfusion servers or clients across ESXi hosts.

We can manage those machines and monitor utilization of GPU resources with this plugin.

We configured bitfusion server version 3.5.0 update 5 on one of our ESXi hosts and registered GPU resources attached to the hosts with this server. After this we installed nvidia drivers on this server and made sure all our GPUs are accessible.[17] Installation of GPUs added Bitfusion Plugin to vcenter. We enabled Bitfusion Plugin and from there we are able to manage and monitor utilization of available GPU resources.

### 4.1.4 Ubuntu

Ubuntu is an open source debian based Linux Operating system. We can manage resources attached with machines and install other third party tools required to proceed with framework deployment on virtual machines.

We created one virtual machine on ESXi hosts and installed Ubuntu version 20.04 as the operating system on that machine. Where we deploy other tools like docker and rancher which play an essential role in the deployment of complete framework.

### 4.1.5 Docker

Docker is an open source platform that helps with the development and delivery of application services completely isolated from our infrastructure. We can run application containers with the help of this platform isolated from our underlying operating system architecture.[4]

We installed docker version 20.10.8 on Ubuntu operating system installed on managing vm. We used docker to run rancher as a container on our virtual machine. Rancher plays an essential role in the management of kubernetes clusters that we need to deploy our application workloads eventually.

### 4.1.6 Rancher

Rancher is an open source set of tools that helps us deploy production scale kubernetes clusters on different cloud platforms e.g. digital ocean, google kubernetes engine, VMware vsphere etc. We can not only deploy kubernetes clusters with rancher but also

the application workloads on those clusters and have access to them all through a single platform.[18]

We deployed rancher as a single node container version 2.6 on our virtual machine to deploy a kubernetes cluster on VMware ESXi hosts that we need for the deployment of applications. We created a cloud credential in rancher with credentials for VMware vcenter that manages our ESXi hosts and a node template that contains specifications for the resources we need to create worker nodes for cluster. Then we used rancher to deploy a kubernetes cluster based on the above mentioned template and cloud credentials to create and deploy resources on our ESXi hosts.

### 4.1.7 Kubernetes

Kubernetes is an open source container orchestration platform. A kubernetes cluster contains a set of virtual machines that are also known as worker nodes. These worker nodes are being used to run containerized application workloads. We can handle auto-scaling of our application containers and also the number of worker nodes as well in a kubernetes cluster.

We deployed a group of kubernetes worker nodes on ESXi hosts through rancher. That's where we will deploy our workloads. Those worker nodes are registered as bitfusion clients and can request GPU resources from the bitfusion server if need be. [9] [18]

## 4.2 Framework

In this section the relationship of different components essential to the framework is discussed. First we break down architecture and describe the relation of components directly relating to each other. After the breakdown explanation, complete architecture and dependency of every component with every other component is discussed. It is also discussed how each component was configured to other and what essential part did those components play by connecting each other.

### 4.2.1 ESXi hosts and VMware vcenter

In this section we discuss where in the framework ESXi servers are configured. How those ESXi hosts are an integral part of the framework and how do we manage more than one ESXi hosts in our framework.

To add a layer of abstraction to the resources of any bare-metal server hypervisors play an important role. For the proposed framework we are using VMware ESXi hypervisor. We install VMware ESXi servers on bare-metal hardware so we can manage our processors, memory, storage and other hardware resources through those ESXi hosts. Once ESXi hosts are installed we can create or delete virtual machines, take snapshots and perform other required actions on underlying hardware resources.

To manage ESXi hosts at one place VMware vcenter server was installed on a virtual machine on one of the ESXi hosts. We create a cluster inside VMware vcenter and add all ESXi hosts connection details to that cluster. Once the VMware vcenter cluster is configured with all hosts we can manage all bare-metal nodes from a centralized solution virtual machines, plugins, services etc. all can be configured and managed across ESXi hosts from a single point of authentication with the help of VMware vcenter.

See Fig. 4.1 to understand how a VMware vcenter is used to manage resources across all ESXi hosts.

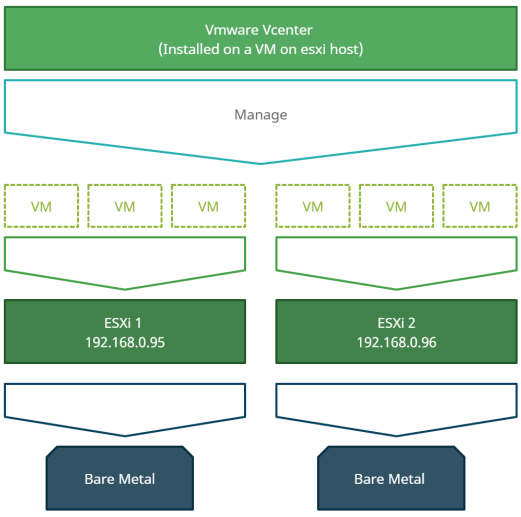


**Figure 4.1:** Vmware vcenter and ESXi hosts

### 4.2.2 Bitfusion in vcenter cluster

After the configuration of ESXi hosts cluster on a VMware vcenter, Bitfusion server can be installed. VMware bitfusion plays an integral role in sharing GPU resources across ESXi hosts and thus our application can use those GPUs even if running on virtual machines having no GPU resources directly attached to it.

Bitfusion Server is a pre-configured ova template provided by VMware. We load this template in our vcenter and create a virtual machine of this on one of ESXi hosts. After creation of the virtual machine it is required to find compatible Nvidia drivers and then install those drivers in the virtual machine. This virtual machine is now called bitfusion server vm. Once Bitfusion server vm is ready, it will generate a plugin in VMware vcenter. Once the bitfusion plugin is enabled in VMware vcenter it indicates successful configuration of bitfusion server inside a VMware vcenter.

Install the bitfusion client on virtual machines which require GPU resources to run AI or ML workloads and have no direct GPU attached to them. After installation of Bitfusion client in VMware vcenter we can enable those virtual machines as bitfusion clients. Once a bitfusion client is configured GPU resources can be allocated to those virtual machines using bitfusion cli commands to run workloads. Bitfusion server in a vcenter cluster is accessible to all bitfusion clients, no matter on which ESXi host that client is created on.

See Fig. 4.2 understand scope of bitfusion with Vmware vcenter.

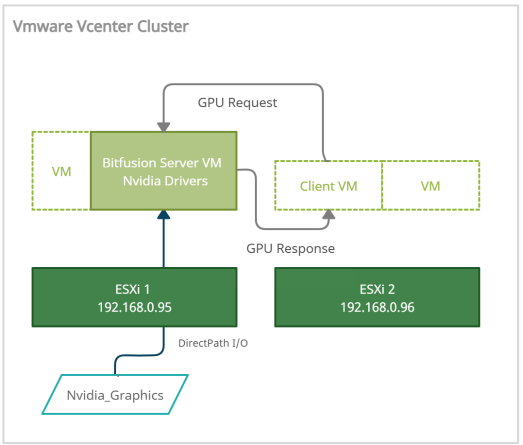


**Figure 4.2:** Vmware vcenter and ESXi hosts

### 4.2.3 VMware Bitfusion with Vmware vcenter

In the previous section we discussed the part where we configure the bitfusion server and what is the part played by VMware vcenter in bitfusion configurations. In this section the need of the VMware bitfusion and its role for the framework is discussed.

Once a VMware bitfusion server is configured in VMware vcenter, we can see the list of all available GPU resources, their utilization over the time and the clients for which servers were being used. To run the microservices inside docker containers and assign them GPU resources, we run AI and ML workload application containers with the bitfusion client installed in those containers and run those on a bitfusion client virtual machine. Once the container with the bitfusion client is started the bitfusion server intercepts the cuda call and then gets the data and remaining cuda calls from the bitfusion client container to process them with the requested number of GPU resources if available. After completion it returns the data back to the bitfusion client container.

After the bitfusion client container receives the processed data the bitfusion server adds the previously allocated GPU back to the GPU pool. This way whenever a GPU resource is required to run any workload on any container we do not need to shift them to some particular machine with GPU being directly attached to it. And this way we can make sure we can use available GPU resources for multiple application containers running across various ESXi hosts and do not need conventional ways to use dedicated GPU resources directly allocated to virtual machines to run AI or ML workloads.

See Fig. 4.3 to understand how the introduction of Bitfusion Server has enabled us to make better use of available GPU resources across ESXi hosts now.

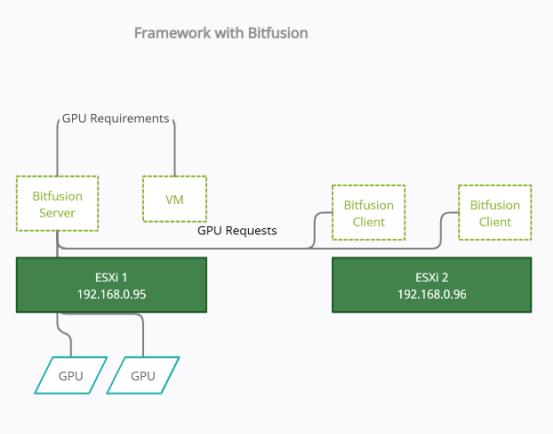


**Figure 4.3:** Bitfusion Server in Framework

### 4.2.4 Rancher VMware Vcenter and kubernetes

In the previous section we have discussed the role of bitfusion in sharing GPU resources and its configuration with VMware vcenter. In this section we will discuss how the rancher deploys kubernetes cluster in our ESXi hosts cluster in VMware vcenter.

Run a rancher container on a virtual machine that can be anywhere but has access to the vcenter network. In this framework we deployed it on one of the ESXi hosts. Once the rancher container is up and running login to the rancher portal on a mapped port through the browser. In the rancher portal add credentials of VMware vcenter in cloud credentials. Create a Node template and fill in all the details to create worker nodes where we want to run our AI or ML workloads. Fill in the form to complete specifications for worker nodes and save that node template to be used next.

Create a kubernetes cluster select VMware vsphere as the provisioner to provision kubernetes cluster. Select cloud credentials created earlier to connect with VMware vcenter and then use the node template configured to create virtual machines and configure them as worker nodes of kubernetes cluster. Select number of master and worker nodes to be created in VMware vcenter and proceed with deployment. Once the deployment process begins, Rancher will communicate with VMware vcenter to create virtual machines on ESXi hosts using VMware api and get its configurations from previously provided node template. After this rancher will create a user on those machines and start downloading required images to complete the deployment of the kubernetes cluster for future usage. Once virtual machines are created on ESXi hosts and the kubernetes cluster is deployed we can register these machines as Bitfusion clients in our VMware vcenter cluster.

Fig. 4.4 explains how a rancher communicates with VMware vcenter to create virtual machines by and then connect with those machines to deploy kubernetes.

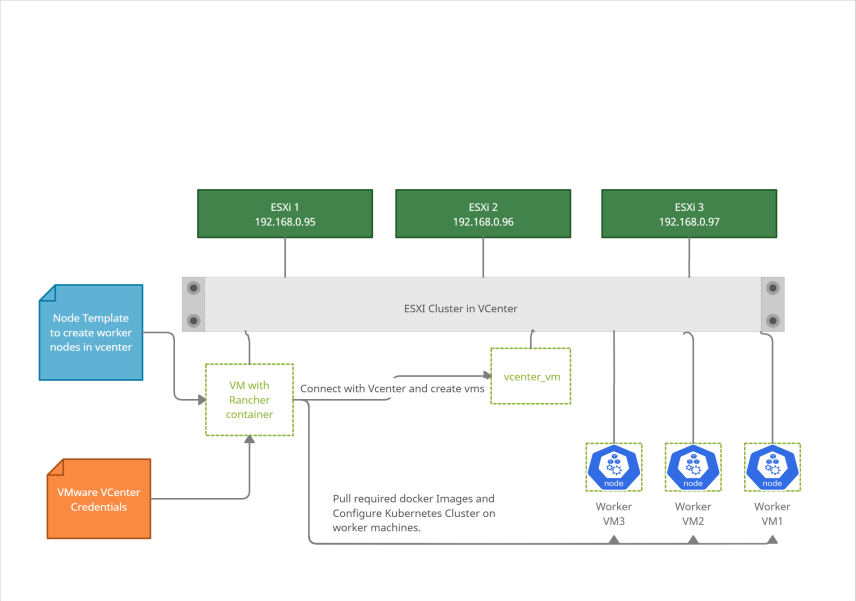


**Figure 4.4:** Rancher VMware vcenter and Kubernetes

### 4.2.5 Architecture

So far we have divided the framework in different sections and discussed how those components are connected with each other, the role of those components in the framework and explained the interaction between those components with a diagram in relative sections. In this section a brief explanation is provided on the complete architecture of the proposed framework. What is the role of all components when they all combine with each other and cumulative outcome of what do we get when all components are configured with each other.

Start with the deployment for VMware ESXi hypervisor on bare-metal servers. VMware ESXi provides an abstraction for the underlying resources to all the virtual machines being created in process. Every bare-metal with ESXi installed will serve as an ESXi host to create and deploy different resources required for the framework. After installation of ESXi hypervisor on bare-metal configure VMware vcenter on the primary ESXi host. Create a cluster in VMware vcenter and add connection details of all hosts in that cluster so all ESXi hosts can be managed through a centralized platform.

Once the cluster is ready and resources can be created across all ESXi hosts from a single point of authentication, deploy the VMware bitfusion server on the ESXi host

that contains GPU resources directly attached to id. Pass that GPU as Direct Path I/O to VMware bitfusion server. Install nvidia driver on bitfusion server machine. VMware bitfusion machine serves as a GPU server across all ESXi hosts inside the cluster where the parent ESXi host is configured. With VMware bitfusion, AI and ML applications can be deployed on virtual machines across all ESXi hosts in the cluster and can still get the GPU resource whenever they need to run the workloads or need GPU resources for any activity. With the configuration of VMware Bitfusion, GPU resources are no more reserved for their parent hosts only but now function as a pool and any client machine can request GPU resources whenever they need it.

Next step is towards the creation of the bitfusion client machines and creating a kubernetes cluster so we provide a platform to actually deploy the applications and utilize those resources. Create a virtual machine on any of the ESXi hosts with available resources, install Ubuntu as operating system and docker engine to run containerized rancher. Rancher is the platform that helps manage kubernetes clusters on VMware ESXi hosts and help manage the deployment of microservices. Once rancher is installed, create cloud credentials in rancher with the connection and user details of VMware vcenter. With cloud credentials rancher can connect with earlier deployed VMware vcenter while creating virtual machines for the deployment of kubernetes cluster. Second thing in rancher is to create a Node template. Node template, as the name suggests, is a template for rancher to create virtual machines. It contains the configuration of resources the virtual machines will be created with vcenter details where those will be created. Finally select the vsphere plugin of rancher from cluster creation form. Fill in the configurations for the vsphere plugin for which template to use, how many virtual machines to create , how many worker and master nodes and other essential resources and deploy the kubernetes cluster. Rancher will connect with VMware vcenter and start creating virtual machines to work as master and worker nodes of kubernetes. Once those machines are created, rancher will connect with those machines directly, download and configure resources and the kubernetes cluster will be provisioned for users to deploy their application resources directly to those virtual machines. Now from VMware vcenter, enable these newly created virtual machines as VMware bitfusion clients. Now the microservice resources that have bitfusion cli installed in them can request GPU resources whenever it needs any.

Fig. 4.5 explains the complete architecture of the framework, where the resources are

created or deployed and how these resources combine to complete the framework.

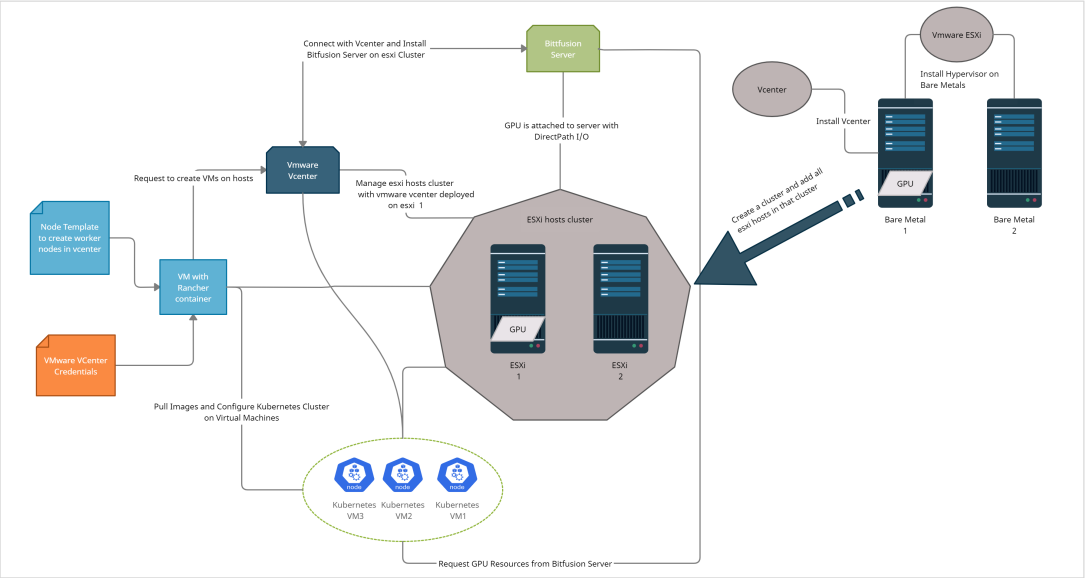


**Figure 4.5:** Architecture Diagram

Chapter 5

# Experiments and Results

In the previous section, we discussed the comprehensive architecture of the proposed framework. We discussed how one component is connected with another and how all those components work together to form the framework.

In this section we will discuss some experiments that we did on the virtual machine having a GPU directly attached to it, that is also a conventional way of allocating GPU resources. We repeated those same experiments with the virtual machines deployed through the framework and then we compared results retrieved from both cases to evaluate the effects of the framework and with how little overhead we can resolve a big challenge of resource under utilization.

## 5.1 Conventional Method

We created a virtual machine on one of the esxi hosts that also has GPU resources. We attached that GPU resource to the virtual machine through direct I/O in virtual machine configurations. Once the GPU was attached we installed Ubuntu 20.04 on that virtual machine as an operating system to perform experiments. [19]

Once the virtual machine was initiated with the operating system, we installed nvidia drivers and nvidia-docker2 to utilize the attached GPU in a containerized environment. We initiated a docker container with cuda-11 to run our experiments on the machine.

### 5.1.1 Matrix Multiplication

Once we have cuda 11 installed and gpu usable inside the container we used cuda sample provided by cuda 11 i.e. matrixMul to perform matrix multiplication utilizing GPU attached to virtual machine. Table Experiment 1 Conventional Method shows the sample set number of operations and time taken for those operations.[20] Table 5.1 shows the sample set number of operations and time taken for those operations in this experiment.

| Experiment | Cuda Ver. | Data set | No. of operations | Time |
|---|---|---|---|---|
| matrixMul | 11 | MatrixA(320,320), MatrixB(640,320) | 131072000 | 0.130 msec |

**Table 5.1:** Experiment 1 Conventional Method

### 5.1.2 Matrix Multiplication cuBLAS

After the results of first experiments were stored and GPU was released for next experiments we executed another sample provided by Cuda 11. For this experiment we used matrix multiplication through cuBLAS. cuBLAS library is optimized for performance on NVIDIA GPUs, and leverages tensor cores for acceleration of low and mixed precision matrix multiplication.[21] Table 5.2 shows the sample set number of operations and time taken for those operations in this experiment.

| Experiment | Cuda Ver. | Data set | No. of operations | Time |
|---|---|---|---|---|
| matrixMulCUBLAS | 11 | MatrixA(640,480), MatrixB(480,320), MatrixC(640,320) | 196608000 | 0.056 msec |

**Table 5.2:** Experiment 2 Conventional Method

## 5.2 Proposed Framework

Once we have performed both the experiments Matrix Multiplication and Matrix Multiplication through cuBLAS on virtual machine with GPU directly attached to it we will perform both operations on the proposed framework where we are sharing GPUs through a GPU pool and running experiments on kubernetes worker nodes having No GPU directly attached to them.

### 5.2.1 Matrix Multiplication

For this experiment we modified the cuda 11 container image and installed bitfusion-cli in that container. After that we launched the updated image in a kubernetes worker node and executed cuda 11 matrix multiplication scripts. Table 5.3 shows the sample set number of operations and time taken for those operations in this experiment by our framework.

| Experiment | Cuda Ver. | Data set | No. of operations | Time |
|---|---|---|---|---|
| matrixMul | 11 | MatrixA(320,320), MatrixB(640,320) | 131072000 | 0.132 msec |

**Table 5.3:** Experiment 1 Proposed Framework

### 5.2.2 Matrix Multiplication cuBLAS

After the first experiments results were generated GPU was released in the gpu pool for other machines to access, we deployed the same container on another worker node and this time we executed matrixMulCUBLAS available in cuda samples and got the execution time. Table 5.4 shows the sample set number of operations and time taken for those operations in this experiment by our framework.

| Experiment | Cuda Ver. | Data set | No. of operations | Time |
|---|---|---|---|---|
| matrixMulCUBLAS | 11 | MatrixA(640,480), MatrixB(480,320), MatrixC(640,320) | 196608000 | 0.059 msec |

**Table 5.4:** Experiment 2 Proposed Framework

## 5.3 Compare and Evaluate

So far we discussed the results we got by running matrixMul and matrixMulCUBLAS [21] by cuda 11 on a virtual machine having a GPU directly attached to it.[19] Then we executed the same experiments with the exact cuda version using bitfusion-cli on virtual machines in our framework which did not have any GPU resources attached to them but used them from a shared pool. Figure 5.3.a Comparison between conventional method and proposed framework shows the difference of time it took to perform those experiments on available GPU resources.

We can notice that for experiment 1 on conventional method matrixMul took 0.130 msec to run 131072000 of operations. But when exact matrixMul was performed in the proposed framework our framework took 0.132 msec to execute them fully. Our framework took slightly higher time (i.e. 0.02 msec) to execute 131072000 of operations. It's behavior was very similar in matrixMulCUBLAS experiment as well. Conventional method took 0.056 msec to execute 196608000 operations and our framework took 0.059

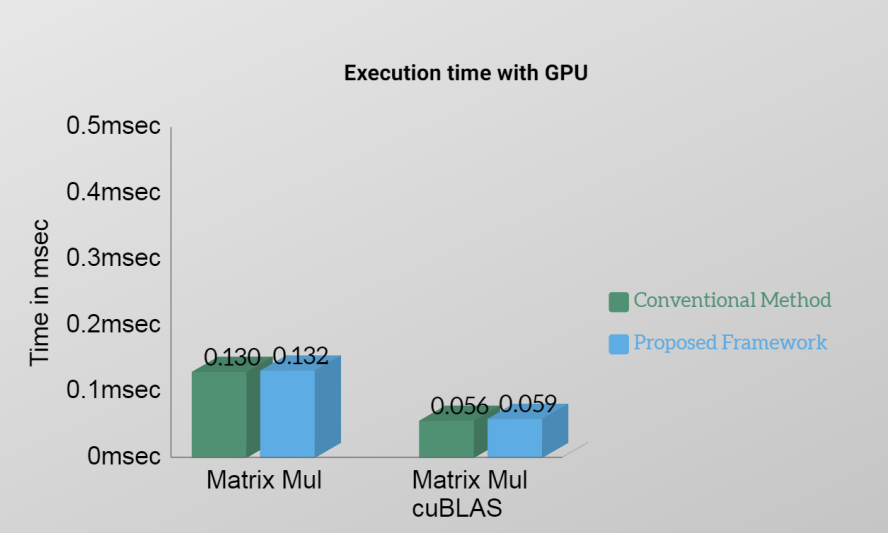


**Figure 5.1:** Comparison between conventional method and proposed framework

msec. We also had similar behaviour when we executed some proprietary algorithms in both cases.

GPU performance is not really in question as in our framework all the processing is performed on GPUs just like every other framework but we face slight delays because in our framework cuda calls and data pointers are transferred over the network to the GPU pool where any available GPU is allocated to perform the required set of actions. That's why we can label those delays as network delays and not the GPU ones.

Considering the fact that with our framework we do not have to allocate GPU resources to specific virtual machines and we can assign GPU resources to whatever virtual machines require on run time, we can avoid significant hardware under-utilization at a negligible time over-head. Which can also be reduced by using better network resources in our data centers to connect esxi hosts. Even in that case the cost of network resources is negligible compared to the cost of additional GPU resources.

Chapter 6

# Conclusions and Future directions

## 6.1 Conclusions

In our work we have developed a framework that allows GPU sharing for the workloads running in a distributed system from a centralized pool. For all native kubernetes GPU scheduling algorithms, GPU is scheduled between multiple containers but worker nodes should have GPUs directly attached to the machine if the user wants to deploy workloads with GPU requirements on that worker. But this direct binding of GPU makes that GPU unusable for other worker machines and workloads deployed on other machines must have their own GPU resources which leads to resource under-utilization and exponential increase in hardware costs. We used VMware ESXi hypervisors for virtualization and VMware vCenter to manage the data-center with all bare-metal servers to form a private cloud. After the configuration of the data-center we used VMware Bitfusion servers to create a pool of all the available GPU resources on all ESXi hosts included in VMware data-center. We then provisioned a Kubernetes cluster on our data-center with a defined template for worker machines and deployed containerized workloads on that cluster. We then enabled Bitfusion Client access for the worker machines in our cluster and installed Bitfusion cli for the workloads which require GPU resources for execution. Workloads whenever they require GPU for processing, request GPU resources from centralized Bitfusion server for GPU. Requested number of GPUs is reserved in the pool for the workloads which are used for all the processing requirements of the microservices and are added back to the pool once it is released from microservice which then can be allocated for other workloads on other machines in the cluster. This GPU

sharing mechanism alleviates GPU under-utilization in a distributed system for resource intensive or complex microservices.

## 6.2 Future Directions

In the future this framework can be extended to share GPUs with workloads running on machines in different clusters making it a multi-cluster platform. This can be further extended to share GPUs for workloads on multiple clouds which can be referred as multi-cloud GPU sharing framework.

# References


[1] Anitha Patil. Distributed programming frameworks in cloud platforms. *IJRTE: International Journal of Recent Technology and Engineering*, 7(2277-3878), 2019.

[2] Dmytro Ageyev, Oleg Bondarenko, Tamara Radivilova, and Walla Alfroukh. Classification of existing virtualization methods used in telecommunication networks. In *2018 IEEE 9th International Conference on Dependable Systems, Services and Technologies (DESSERT)*, pages 83–86, 2018. doi: 10.1109/DESSERT.2018.8409104.

[3] Mark F. Mergen, Volkmar Uhlig, Orran Krieger, and Jimi Xenidis. Virtualization for high-performance computing. *SIGOPS Oper. Syst. Rev.*, 40(2):8–11, April 2006. ISSN 0163-5980. doi: 10.1145/1131322.1131328. URL https://doi.org/10.1145/1131322.1131328.

[4] Amit M Potdar, Narayan D G, Shivaraj Kengond, and Mohammed Moin Mulla. Performance evaluation of docker container and virtual machine. *Procedia Computer Science*, 171:1419–1428, 2020. ISSN 1877-0509. doi: https://doi.org/10.1016/j.procs.2020.04.152. URL https://www.sciencedirect.com/science/article/pii/S1877050920311315. Third International Conference on Computing and Network Communications (CoCoNet'19).

[5] Theo Combe, Antony Martin, and Roberto Di Pietro. To docker or not to docker: A security perspective. *IEEE Cloud Computing*, 3(5):54–62, 2016. doi: 10.1109/MCC.2016.100.

[6] Claus Pahl, Pooyan Jamshidi, and Olaf Zimmermann. Microservices and containers. In Michael Felderer, Wilhelm Hasselbring, Rick Rabiser, and Reiner Jung, editors,

*Software Engineering 2020*, pages 115–116, Bonn, 2020. Gesellschaft für Informatik e.V. doi: 10.18420/SE2020_34.

[7] Tom Goethals, Filip De Turck, and Bruno Volckaert. Fledge: Kubernetes compatible container orchestration on low-resource edge devices. In *International Conference on Internet of Vehicles*, pages 174–189. Springer, 2019.

[8] Oana-Mihaela Ungureanu, Călin Vlădeanu, and Robert Kooij. Kubernetes cluster optimization using hybrid shared-state scheduling framework. In *Proceedings of the 3rd International Conference on Future Networks and Distributed Systems*, pages 1–12, 2019.

[9] Gabriele Carcassi, Joe Breen, Lincoln Bryant, Robert W. Gardner, Shawn Mckee, and Christopher Weaver. Slate: Monitoring distributed kubernetes clusters. In *Practice and Experience in Advanced Research Computing*, PEARC '20, page 19–25, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450366892. doi: 10.1145/3311790.3401777. URL https://doi.org/10.1145/3311790.3401777.

[10] Shengbo Song, Lelai Deng, Jun Gong, and Hanmei Luo. Gaia scheduler: A kubernetes-based scheduler framework. In *2018 IEEE Intl Conf on Parallel Distributed Processing with Applications, Ubiquitous Computing Communications, Big Data Cloud Computing, Social Computing Networking, Sustainable Computing Communications (ISPA/IUCC/BDCloud/SocialCom/SustainCom)*, pages 252–259, 2018. doi: 10.1109/BDCloud.2018.00048.

[11] Ghofrane El Haj Ahmed, Felipe Gil-Castiñeira, and Enrique Costa-Montenegro. Kubcg: A dynamic kubernetes scheduler for heterogeneous clusters. *Software: Practice and Experience*, 51(2):213–234, 2021. doi: https://doi.org/10.1002/spe.2898. URL https://onlinelibrary.wiley.com/doi/abs/10.1002/spe.2898.

[12] Ting-An Yeh, Hung-Hsin Chen, and Jerry Chou. Kubeshare: A framework to manage gpus as first-class and shared resources in container cloud. HPDC '20, page 173–184, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450370523. doi: 10.1145/3369583.3392679. URL https://doi.org/10.1145/3369583.3392679.

[13] ESXi. Vmware esxi: The purpose-built bare metal hypervisor, 2021. URL https://www.vmware.com/products/esxi-and-esx.html.

[14] Hiteshi. Analysis of system performance using vmware esxi server virtual machines. Master's thesis, 2012.

[15] vCenter Server. Simplified and efficient server management, 2021. URL https://www.vmware.com/products/esxi-and-esx.html.

[16] VMware. vsphere ai/ml solu, 2021. URL https://www.vmware.com/products/vsphere/ai-ml.html.

[17] VMware. Vmware + nvidia ai-ready enterprise platform, 2021. URL https://www.vmware.com/products/vsphere/ai-ml.html.

[18] Rancher. One enterprise platform for managed kubernetes everywhere, 2021. URL https://rancher.com/products/rancher/.

[19] Diana M. Naranjo, Sebastián Risco, Carlos de Alfonso, Alfonso Pérez, Ignacio Blanquer, and Germán Moltó. Accelerated serverless computing based on gpu virtualization. *Journal of Parallel and Distributed Computing*, 139:32–42, 2020. ISSN 0743-7315. doi: https://doi.org/10.1016/j.jpdc.2020.01.004. URL https://www.sciencedirect.com/science/article/pii/S0743731519303533.

[20] Robert Hochberg. Matrix multiplication with cuda — a basic introduction to the cuda programming model. Master's thesis, 2012.

[21] Junjie Li, Sanjay Ranka, and Sartaj Sahni. Strassen's matrix multiplication on gpus. In *2011 IEEE 17th International Conference on Parallel and Distributed Systems*, pages 157–164, 2011. doi: 10.1109/ICPADS.2011.130.

Appendix A

# Appendices

## A.1 vCenter Deployment requirements

To deploy a VMware vCenter with static IP address make sure you have a valid FQDN with forward and reverse lookup validity. Even though it is claimed to be optional for deployment but process cannot be completed without a valid FQDN.

Use the following command to make sure that you have a valid FQDN before starting the deployment process.

- nslookup -nosearch -nodefname FQDN_or_IP_address

If you use IP_address with above value make sure it returns FQDN and vice versa. Use a browser to initiate stage 2 of Vmware vCenter deployment instead of deployment manager. Management console should be available at url 'https://<fqdn_ip>:5480 which can lead to configuration tab, once in configuration tab replace url 'https://<fqdn_ip>:5480/configurev2 with 'https://<fqdn_ip>:5480/configure' as for both deployment manager and configurev2 are unstable and cause interrupts in deployment process.

## A.2 Bitfusion Deployment Modifications

Successful deployment of VMware Bitfusion server in a vCenter cluster requires few not so popular modifications and restrictions in the process.

When making configurations for the template make sure to use staticIP as Hostname of the bitfusion server it uses the value of hostname to look for monitoring health checks and other endpoints during installation process.

Second thing that has a critical impact during the deployment process is nvidia drivers installation. Do not select the 'download and install nvidia drivers' check box. It is unchecked by default but if selected can cause conflicting driver installations which does not allow GPU sharing in the cluster.

After successful creation and initialization of the Bitfusion server, copy compatible nvidia drivers NVIDIA-Linux-x86_xxx.run file to the Bitfusion server. Create a ssh connection with the server and use following commands to install drivers on Bitfusion server.

- sudo su
- cd /path/to/file/
- chmod +x NVIDIA-Linux-x86_xxx.run
- ./NVIDIA-Linux-x86_xxx.run –kernel-source-path /usr/src/linux-headers-xxx.ph3 –ui none –no-x-check

After successful installation of nvidia drivers, shutdown server and add the following parameter in the server configurations.

- hypervisor.cpuid.v0 = FALSE

When the server gets started, the Bitfusion plugin will be integrated in the cluster and can be managed through the plugins section in vCenter.