DESIGN AND DEVELOPMENT OF A PLANNING ARCHITECTURE FOR CPUs AND GPUs FOR CONTAINERS

Alberto González Cristobo

Master’s Thesis presented to the Telecommunications Engineering School
Master’s Degree in Telecommunications Engineering

Supervisors
Prof. Felipe Gil Castiñeira

2020
Every day we know more
and we understand less.
— Albert Einstein

To my family and friends, and all the people that the University gave me the pleasure to meet.
Abstract

Graphics Processing Units (GPUs) were originally designed as coprocessing units to accelerate the creation of images intended for being displayed in a screen. Nevertheless, over the years graphics chips became increasingly programmable and developers started to create applications that used the GPU parallel architecture to accelerate calculations. Initially, it was necessary to use programming languages designed for graphics, such as OpenGL, but NVIDIA created CUDA (a platform and an application programming interface or API) for general purpose processing. Nowadays, for certain applications a GPU can utterly surpass the performance of a Central Processing Unit (CPU).

Thus, many applications are being designed for its use in GPUs, but they still can use the CPU when the GPU is busy with another task.

Service virtualization techniques became also essential to develop new applications that are easy to deploy, maintain and scale. For example, the microservice architecture is usually built by using virtual machines or, more frequently, containers (where the OS is virtualized and not the hardware) to execute each service. Among the different containerization tools, Docker is one of the most popular, and the selected in this Master’s Thesis.

Such services may take advantage of the GPU processing power. Nevertheless, most GPUs have not been designed for time-sharing (CPUs are shared among several applications and users using this approach) so, in order to maximize the use of the GPU, it is necessary to have an orchestrator that assigns the applications to the GPU or just to the CPU of a systems, or even that chooses one node among the different nodes available in a cluster.

There are different alternatives that orchestrate Docker containers. One of the most famous is Kubernetes, an open source manager for Docker developed by Google.

One of the core components in a container orchestrator is the scheduler. After a user creates a Pod (the basic scheduling unit, which can consist of one or more containers) the scheduler uses the information regarding the nodes, decides to which one the Pod should be assigned. Scheduling is an optimization problem, where the scheduler determines which nodes meet the requirements, and then selects the best one.

By default, the Kubernetes scheduler is executed when a new Pod is created. Nevertheless, as stated before, Pods can be designed to take advantage of the processing power provided by GPUs, but they can still run in a CPU. Therefore, if when a new Pod is created there are not GPUs available, it will be assigned to a node with just a CPU. Clearly this operation can be improved, so we have studied how to reassign the resources, trying
to provide GPUs to the Pods that require them. We have analyzed the different viable alternatives to achieve this objective, and we decided to implement a new scheduler. Finally, we have studied its performance and compared it with the default Kubernetes scheduler.

Key words: Kubernetes, Docker, GPU, scheduling, containers, virtualization.
Contents

Abstract i

List of figures v

1 Introduction 1
  1.1 GPUs and computing ............................................. 2
  1.2 Kubernetes ...................................................... 3
    1.2.1 Kubernetes architecture ................................. 5
  1.3 Objectives ...................................................... 6

2 The Kubernetes scheduler 9
  2.1 Advanced scheduling features .................................. 12
  2.2 Descheduler for Kubernetes .................................... 13

3 A new GPU scheduler for Kubernetes 17
  3.1 Validating the scheduler ...................................... 19

4 Results 21
  4.1 Performance of Kubernetes with GPU support enabled ........ 22
  4.2 Performance improvement with the new scheduler ............. 22
    4.2.1 Matlab simulation .......................................... 23
    4.2.2 Validation with real experiments ......................... 23

5 Conclusions 31

A Installation 33
  A.1 Installation .................................................. 33
    A.1.1 First Steps ................................................. 34
    A.1.2 NVIDIA Soft ................................................ 34
    A.1.3 Docker install ............................................. 34
    A.1.4 Check and optimize Docker ................................ 36
    A.1.5 Kubernetes ................................................ 36
# List of Figures

1.1 GPU performance vs CPU performance ........................................... 3
1.2 Virtual Machine Stack ................................................................. 4
1.3 Containers Stack ...................................................................... 4
1.4 Kubernetes structure ................................................................. 6

2.1 Scheduler example: cluster ........................................................... 11
2.2 Scheduler example: filtered nodes ................................................ 11
2.3 Scheduler example: prioritized nodes ............................................ 12

4.1 Cluster example ....................................................................... 21
4.2 Default scheduler vs our modified version of Descheduler ............... 24
4.3 CPU node only ......................................................................... 24
4.4 Descheduler applied ................................................................. 25
4.5 Two new GPU nodes added ........................................................ 26
4.6 Cluster after the tests ............................................................... 27
4.7 Intermediate state with new GPU nodes ........................................ 27
4.8 State at the end of the execution with new GPU nodes .................... 28
4.9 Number of executions and time spend in forward benchmark ............. 28
4.10 Number of executions and time spend in forward-backward benchmark .. 29
4.11 Matlab Simulation with 5000 jobs in 2 GPUs and 3 CPUs ................ 29
4.12 Time to complete the jobs, which arrive following a Poisson process .... 30

A.1 NVIDIA-SMI ................................................................. 35
A.2 Docker runtime NVIDIA .......................................................... 36
A.3 Docker daemon ................................................................. 37
A.4 NVIDIA Docker ................................................................. 37
In recent years, cloud computing became an essential technology for providing services in the Information Technologies (IT) industry. Large data centers host computer system resources such as data storage and computing power, provided by CPUs (x86 or ARM, principally) or even GPUs. Services are executed dynamically in that diverse infrastructure, so it is necessary to deploy a control layer that can decide where applications should be executed to minimize the time to provide results while limiting the required resources.

Hardware virtualization proved to be essential in this scenario to separate one physical computing device into one or more virtual devices, facilitating the allocation and usage of idle computing resources and the isolation between services offered by different customers. Virtual Machines (VMs) provide an abstraction layer that allows executing any software compatible with the virtual machine without the need of a specific hardware. This can be achieved because the VM core emulates a physical host that executes the instructions. VMs offer isolation between processes thanks to the hardware abstraction layer.

Although VMs provide an excellent solution for implementing computing solutions as a service, more recently appeared another solution for executing applications in data centers that maximize the use of the hardware available: containers. Instead of virtualizing a system at the physical level, the container approach provides Operating-System (OS) virtualization. Multiple isolated user space instances (called containers, zones, virtual private servers, etc.) share the same OS system call interface to interact with the system resources (devices, filesystem, CPU power, network, etc.) without needing a complete virtualized computer, then reducing the consumption of resources in data centers, and being able to execute more applications at the same time.

GPUs have become critical for different applications. For example, its usage is essential for training deep learning models and neural networks. This is especially true for computer vision related models, which require to perform complex computations on tens of thousands of matrices. Nevertheless, container systems didn’t include GPU support until recently, when NVIDIA created a runtime for Docker called NVIDIA-Docker, which provisions a
Chapter 1. Introduction

container with the necessary dependencies to execute code on the GPU. Nevertheless, such runtimes don’t allow to share a GPU among several containers, so it is necessary to wait until a container has finished before launching another that will use the same GPU, or run that container using just the CPU.

If we want to maximize the performance of a cloud computing datacenter that allows containers to use GPUs and CPUs, we have to schedule the different applications and decide when to assign a container to a node with a CPU or a GPU. This is the focus of this Master’s Thesis.

1.1 GPUs and computing

Nowadays, when an application requires parallel processing, such as matrix calculations, it is necessary to think about GPUs. They are the most cost-effective hardware to obtain the best cost-performance ratio. The architecture of a GPU allows many applications to run in parallel as opposed to a CPU where the number of cores is smaller. There are many GPU manufacturers in the market, but the main competitors are NVIDIA and AMD, offering GPUs both for the consumer and professional markets. A professional grade GPU is very similar to a consumer grade one, although for the consumer market they are optimized for image processing in video games, and for the professional market they are more focused for their use in calculation, or 3D edition.

Today NVIDIA leads the discrete GPU market, and offers an SDK to develop software to be executed in GPUs. That SDK is known as CUDA, and enables GPUs to execute C++ code with some modifications. An application that was originally designed to be executed in a CPU can be easily adapted to use also the GPU. Figure 1.1 shows the difference of performance between GPUs and CPUs in terms of Gigaflops, and how the performance gap is growing with time.

GPU-accelerated computing is improving operations such as deep learning, data analysis or engineering applications that perform parallel computing operations. The use of GPUs in clusters has grown exponentially as shown in the TOP500 [1] list of the most powerful supercomputers of the world. The top place is occupied by a supercomputer with a lot of CPU cores, but the second position is occupied by a supercomputer which also includes NVIDIA GPUs.

But not only supercomputing applications use GPUs. Many consumer applications are also using GPUs to improve their performance. For example, photographic or video editors are using GPUs to reduce the time required to complete different tasks, especially during editing, since it facilitates editing in real time. Another popular use is for cryptomining, where the increasing complexity of calculations requires many resources and GPUs are the only ones that can get the result in a reasonable time.
1.2. Kubernetes

Virtualization is one of the key technologies that make Cloud Computing work, by helping to solve problems such as process isolation, scalability, or hardware optimization. Traditionally, VMs have been used to emulate a complete system in a physical host system (figure 1.2). This can be done multiple times, while the physical infrastructure has enough computing power. Nevertheless, each time a new VM is needed, it is necessary to deploy a complete computing stack (emulator, operating system, applications) which may be excessive for many purposes.

Furthermore, although some components of the hardware can be emulated, it is necessary to provide applications with direct access to the main components. There are plugins that can be installed in the virtual machine controller, which act as a bridge between the emulated OS and the hardware. This approach is typically used to provide applications access to the GPU. Thus, it is not possible to share a GPU among several VMs (except if we use GPUs specifically designed for this purpose, but they are very expensive).
Chapter 1. Introduction

Another option to share the same physical resources between isolated applications is the use of containers. This architecture does not provide a layer of virtualized hardware, as the virtualization, instead it is implemented at the OS layer (figure 1.3). This reduces the overload compared to a VM approach, and therefore also reduces the hardware that is required to run the same applications [2].

The inconvenience of using containers appears when we want to save information, because when a container is shut down, it loses all the information, unless we create an associated volume and save the information there before. In virtual machines that does not happen.

Docker is one of the main solutions for containers. Furthermore, NVIDIA has released NVIDIA-docker 1, a container runtime that includes GPU support.

\[1\text{https://github.com/NVIDIA/nvidia-docker}\]
Docker provides the tools for building, distributing, and running containers, but it is also necessary to have tools that make it possible to coordinate and schedule the different containers, or to facilitate the communication of the different containers in order to build complex services (e.g. a web service requires databases, frontends, backends, etc. that can be executed in different containers).

Containers orchestrators are the solution. Kubernetes (developed by Google) is one of the most popular in the last years. Docker swarm, developed by the creators of Docker, is another alternative. It is embedded with Docker, and makes it possible to create clusters without the need of other applications. Nevertheless, Kubernetes is considered better in tasks such as scaling or volume assignment.

Kubernetes “is an open-source system for automating deployment, scaling, and management of containerized application”, so it is possible to modify it to improve its characteristics.

1.2.1 Kubernetes architecture

Kubernetes is implemented with Go\(^2\), an evolution of C with an object oriented programming.

Kubernetes is designed to control a cluster of nodes. One of the nodes will be the master, and will control the others using a star topology. Figure 1.4 shows the main elements and how they are interconnected.

The master is the node that knows how Pods are distributed in the cluster and the state of the network (etcd). It executes control functions and performs the Pod scheduling algorithm. The master communicates with the rest of the nodes through an API. This communication is bidirectional, as nodes can communicate with the master to notify their state and the master can send orders to the different nodes using this interface.

In each node, the instructions received from the master are managed by the Kubelet app. It manages the communication and starts or stops the Pods. A node can run multiple Pods at the same time. cAdvisor provides information about the resources of the node to the master. Kube-proxy provides the interface with the users. They can use it to make petitions to the cluster and execute the required Pod.

Then, if we want to deploy a Pod that requires using GPUs, the master node will ask each node to report its free resources, and once that information is collected, it runs the scheduling algorithm to select the best node for executing the new Pod. Once the Pod is deployed it runs until the end, and is not moved or reallocated.

\(^2\)https://golang.org/

\(^3\)https://upload.wikimedia.org/wikipedia/commons/b/be/Kubernetes.png
Chapter 1. Introduction

Kubernetes offers a simple environment for managing Docker containers to create services ranging from small web applications to complex data processing architectures. Kubernetes is easily extended by deploying new applications that can also be managed by Kubernetes itself.

1.3 Objectives

First of all, GPUs have become essential to accelerate applications that can be parallelized. In addition, applications are being offered as services, following the cloud computing paradigm, and finally, Docker containers is one of the most popular solutions to virtualize services. Therefore, this Master’s Thesis looks for a mechanism to assign applications to nodes, using Docker containers, to optimize the use of a cluster of nodes by taking advantage of the existing GPUs to accelerate the compatible applications. Kubernetes is one of the most popular orchestrator for Docker, so we will use this platform to implement our solution.

So, one of the first objectives is to allow Docker and Kubernetes to manage GPUs, because by default, they are not configured to use them. Then, it is necessary to understand how Kubernetes assigns Pods (containers or sets of containers) to the different nodes.

Figure 1.4 – Kubernetes structure

Kubernetes offers a simple environment for managing Docker containers to create services ranging from small web applications to complex data processing architectures. Kubernetes is easily extended by deploying new applications that can also be managed by Kubernetes itself.

1.3 Objectives

First of all, GPUs have become essential to accelerate applications that can be parallelized. In addition, applications are being offered as services, following the cloud computing paradigm, and finally, Docker containers is one of the most popular solutions to virtualize services. Therefore, this Master’s Thesis looks for a mechanism to assign applications to nodes, using Docker containers, to optimize the use of a cluster of nodes by taking advantage of the existing GPUs to accelerate the compatible applications. Kubernetes is one of the most popular orchestrator for Docker, so we will use this platform to implement our solution.

So, one of the first objectives is to allow Docker and Kubernetes to manage GPUs, because by default, they are not configured to use them. Then, it is necessary to understand how Kubernetes assigns Pods (containers or sets of containers) to the different nodes.
available in a cluster. For example, Kubernetes uses a scheduler that allocates resources when a new task arrives, but we want to take advantage of the existing GPUs and assign the Pods that can make use of them in order to get better global results. Thus, we will have to modify how the Kubernetes scheduler operates. In our development we use Pods provided as a services that the cluster can run. We use that approach to be able to start and stop Pods when we want because if a container stop we lose the information that contains. If we use Pods as service we have the information stored outside it, so we can start or stop without the risk of losing information.

Finally, it will be necessary to measure the performance of the new scheduler, and compare it with the performance of the default one.

Thus, the main objectives for this Master’s Thesis are:

1. Add support for GPUs to Docker
2. Add support for GPUs to Kubernetes
3. Learn about how the Kubernetes scheduler works
4. Modify the scheduling algorithm to consider GPUs
5. Improve the default scheduler for Kubernetes
6. Compare results between the default scheduler and our proposal
In order to improve the assignment of GPU enabled nodes to the Pods that will take advantage of this physical resource, it is necessary modify the operation of the Kubernetes scheduler. Then, first of all it is necessary to understand how a scheduler works, and how they are being used to manage containers or even VMs, and how they can be modified to consider the application requirements and the hardware available.

A scheduler is an algorithm used to distribute the resources to various processes, users, applications etc. Usually, it works just as a load balancer, by distributing the tasks over the different nodes.

Scheduling is required in many diverse scenarios, even in computing. For example, to manage how CPUs, memory or disk are shared among different users. Schedulers must be fast and suppose a low overhead for the system, because if we spend a relevant amount of time scheduling the system would provide a poor global performance and may provide results too late.

A scheduler can be very simple, for example by using a Firs-In-First-Out (FIFO), where each incoming request is processed and assigned in order, or much more complex as a HASS [3] algorithm. This is characterized by using a search algorithm between the hardware and software to be used where both parties meet the requirements of the other.

HASS allows assigning applications to those cores that provide a better result. But there are lots of different applications, with different requirements. Then if an application has strict requirements, the scheduler will assign it to the faster core (GPU and/or CPU), and applications with lower needs are assigned to slower nodes. This also makes a more efficient use of the system, because applications that require more resources will finish earlier, avoiding keeping a slower core busy during a long period and avoiding the use of that core by another application. Nevertheless, to take advantage of HASS it is necessary to have a prior knowledge about the use of the hardware by the different applications. Thus, HASS needs more information than other schedulers to make decisions, making
Chapter 2. The Kubernetes scheduler

the scheduling slower, specially when a new type of application appears. Thus, HASS algorithm optimizes the hardware assignation, it requires more time to be executed, so its performance is worse than FIFO for low demanding applications.

Kubernetes uses by default a scheduler that satisfies the requirements for typical scenarios. The scheduling in Kubernetes is part of the his core, and operates when a new Pod appears in the system\(^1\). When a new Pod arrives, the scheduler determines the best node for the new Pod. To select the best node is necessary complete some steps before its assignation.

The master node maintains a register with information about the resources available in each node. Thus, the scheduler can make a fast decision to assign the Pod to a particular node, which is selected according to an algorithm.

The scheduler tries to find a node for each Pod, one at a time, as it notices these Pods. First, the scheduler filters out nodes that don’t satisfy the requirements, which will be defined in the Pod and may change. Such requirements include different hardware capabilities such as number of CPU cores, available memory, or number of GPUs to use. Figure 2.1 shows an example where a cluster composed of 8 different nodes and a master, decides where to deploy a new Pod in the system. When the master nodes try to create a new Pod, it asks the scheduler to provide a list of nodes satisfying the requirements.

Once this selection has been completed, we will have a smaller list of nodes (figure 2.2). Now the scheduler asks each node about the use of its resources to see if they meet the requirements. If these requirements are met, the scheduler will assign a weight to each node, and create a weighed vector, it uses formula 2.1:

\[
finalScore_{Node,A} = weight_1 \times priorityFunc_1 + weight_2 \times priorityFunc_2
\]  

(2.1)

The node that obtains the best score is selected (figure 2.3). In case there are several nodes with the same priority, one of them is chosen randomly.

Thus, we can summarize the steps completed to decide which node should be used:

1. A new Pod appears in the system.
2. The scheduler is notified about the creation of a new Pod.
3. The scheduler removes the nodes that don’t satisfy the requirements.
4. Once the list of available nodes is ready, then the scheduler asks each node about the usages of the system to assign a weight to the node.

\(^1\)https://sysdig.com/blog/kubernetes-scheduler/
Figure 2.1 – Scheduler example: cluster

Figure 2.2 – Scheduler example: filtered nodes
Chapter 2. The Kubernetes scheduler

5. With the weight calculated before, build an ordered vector where the best node have the max weight.

6. Scheduler informs the Master where create the new Pod, and the master acts over the selected node to deploy the pod.

There are different studies that analyze the performance of the Kubernetes scheduler with different loads and different Pods [4].

Kubernetes also allows to constrain Pods to run only on particular nodes, or to prefer to run in particular nodes\(^2\). To do this, when a Pod is defined it is necessary to add label selectors to make the selection. This approach is explained in detail in section 2.1.

Furthermore, the operation of the Kubernetes scheduler easily extended by choosing one of the predefined scheduling policies or by adding new ones\(^3\).

2.1 Advanced scheduling features

One of the simpler ways to modify the default operation of the Kubernetes scheduler is by including constraints in Pod definitions to force them to run in particular nodes [5]. Nevertheless, this requires the administrator intervention to manually include such restrictions in the Pod configuration file. Below is a brief explanation of the available options to configure the deployment of the Pod and hand how it affects the scheduler.

\(^2\)https://kubernetes.io/docs/concepts/configuration/assign-pod-node/

\(^3\)https://github.com/kubernetes/kubernetes/blob/release-1.1/docs/devel/scheduler_algorithm.md
2.2 Descheduler for Kubernetes

- **Node Affinity/Anti-Affinity:** This option allows to run a Pod only in certain nodes with/without certain characteristics, like having a SSD, or having enough RAM that will be required by the Pod etc.

- **Taints and Tolerations:** Allows a node to “repel” a set of Pods. When a node has this option enabled, it will only accept the type of Pods defined.

- **Pod Affinity/Anti-Affinity:** The Pod affinity/anti-affinity is like in the Node Affinity/Anti-Affinity, but applied to Pods. That is, makes it possible to define with which Pods should the labeled Pod executed or not.

- **Custom Schedulers:** With that option it is possible to delegate the responsibility for scheduling a Pod or a subset of Pods to a custom scheduler. Nevertheless, again, by using this option the scheduler has to be explicitly declared in the Pod configuration file.

Thus, by including the different constraints when a Pod is scheduled, it is possible to force its execution in a particular node. In our case, it is possible to force the execution of a Pod in a node that includes a GPU. Nevertheless, this requires a manual modification of the configuration of the Pods.

### 2.2 Descheduler for Kubernetes

One of the limitations in the default Kubernetes Scheduler resides in the fact that when a Pod is deployed in a node, it runs until the end, even when there is another better node. The scheduler is only executed when a new Pod arrives, and does not move already running Pods even when the cluster state changes.

Descheduler\(^4\) is a tool that is being developed (is part of the Kubernetes Incubator\(^5\)) to address this problem. Descheduler finds Pods that can be moved and evicts them, according to a predefined policy and according to different possible reasons such as:

- Some nodes are under or over utilized.
- The original scheduling decision does not hold true anymore, as taints or labels are added to or removed from nodes, Pod/Node affinity requirements are not satisfied anymore.
- Some nodes failed and their Pods are moved to other nodes.
- New nodes are added to clusters.

\(^4\)https://github.com/kubernetes-incubator/descheduler

\(^5\)https://github.com/kubernetes/community/blob/master/incubator.md
Chapter 2. The Kubernetes scheduler

The first thing that Descheduler analyzes is the node occupancy. It looks for nodes with low occupancy, and evicts Pods from other nodes with the hope that recreation of evicted Pods will be scheduled on these underutilized nodes. Finally, it analyzes all nodes, and checks if some Pods can be moved because of their special requirements.

This new tool for Kubernetes facilitates a better assignment of Pods to nodes. It can be executed manually or periodically by Cron or as a job inside a Pod.

Descheduler uses configuration files in the Master node. It is also necessary to define a new role with a new user, which will be used to run Descheduler, and limit its level of execution. The configuration file establishes the different threshold levels that set when we consider that a node as under-used or over-used. This configuration is considered a policy file for Descheduler and is used by Descheduler to determine what Pods should be evicted.

Descheduler can be also useful to delete other Pods deployments that not fit because they are violating some restrictions like Node Affinity and Pod Affinity.

This is an example of a policy configuration that will be applied by Descheduler to take into account if a node with an NVIDIA GPU is not being utilized:

```yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: descheduler
data:
  policy.yaml: |
    apiVersion: descheduler/v1alpha1
    kind: DeschedulerPolicy
    strategies:
      RemoveDuplicates:
        enabled: false
      LowNodeUtilization:
        enabled: true
        params:
          nodeResourceUtilizationThresholds:
            thresholds:
              cpu: 1
              memory: 10
              pods: 5
              nvidia.com/gpu: 0
            targetThresholds:
              cpu: 2
```
The project will study different scheduler implementations and their performance. In [6] we can observe how different schedulers act in big clusters that perform big data analytics. In this master’s thesis we study the number of processors that can be managed by a scheduler before reaching the maximum of the system. to reach this objective we use some metrics that can determine if one scheduler is better than another. For example, we consider the number of jobs that can be executed at the same time and the time spent in assigning a new Job to a Worker.

It is also necessary to understand how Kubernetes handles different types of hardware. In this work, we also defined a strategy to make this comparison. For example, as can be observed in [4], it is possible to define different tests to know how Kubernetes acts for heavy load applications that demand high amounts of resources and how Kubernetes can scale in clusters with a good amount of resources. The authors developed their own scheduler. In our case we use the tools provided by Kubernetes to modify the scheduler to assign Pods to CPUs or GPUs, in order to reduce the global time required to complete their execution.
3 A new GPU scheduler for Kubernetes

Scheduling the execution of Pods in different Kubernetes nodes is not a simple task, as many factors have to be considered. Although the latest versions of NVIDIA-Docker support containers enable GPU acceleration, a GPU can be used only by a container at a time. Thus, in order to globally optimize the system, it is necessary to assign Pods to the right nodes during the initial scheduling, or to move the Pods to the nodes that are free after other Pod finishes.

Some high performance professional GPUs can be shared among different VMs\(^1\). Nevertheless, such technology is still not available for containers, so it is necessary to reserve a physical GPU for each container that uses it. Then, it is necessary to create a scheduler that is specifically designed to manage how GPUs are assigned. This can be specially useful for containers that can use a CPU to complete their tasks, but that may also use a GPU to accelerate the calculations.

Moreover, the default Docker and Kubernetes installations don’t support GPUs. This is the first challenge that has to be addressed. Fortunately, the publication of NVIDIA-Docker and the latest Kubernetes versions provide documentation to enable GPU support in an easy way.

Once GPUs are supported by the Kubernetes installation, it is necessary to replace the default scheduler. The new one, completed in this Master’s Thesis, should be able to assign GPU nodes to the Pods that may take advantage of them, but having into account that the cluster is an entity where the state and the demands change with the time. The default scheduler only analyzes the state of the cluster when a Pod arrives. This approach is adequate for clusters with lots of available resources or for systems that only run Pods that require a short execution time, but in a cluster with a limited number of GPUs or with processes that run for long periods of time, it will be probably necessary to move Pods when the state of the cluster changes. Nevertheless, as introduced in chapter

\(^1\)https://blogs.nvidia.com/blog/2018/06/11/what-is-a-virtual-gpu/
2, when a new Pod arrives, the scheduler decides which node is the most adequate to execute it.

Our scheduler pays special attention to nodes with available GPUs, and includes that information to select the node that will be assigned to the new Pod. This is a novelty, as the default scheduler doesn’t consider the information regarding GPUs to prioritize nodes. We have included the information about the GPUs to assign a new weight to the node. So, if a Pod can use a GPU to accelerate its calculations, and a node includes one or several GPUs, we add the number of GPUs to the score assigned to that node.

Thus, our scheduler uses the following equation to assign a score to each node:

\[
finalScore_{\text{Node}: A} = weight_1 \times priorityFunc_1 + weight_2 \times priorityFunc_2 + \\
weight_3 \times priorityFunc_3 + weight_4 \times priorityFunc_4
\]  

(3.1)

The three first “priority functions” are the same used by the default scheduler: \( priorityFunc_1 \) is the percentage of CPU resources being used, \( priorityFunc_2 \) the percentage of memory being used, and \( priorityFunc_3 \) the hard disk space used.

Therefore, when a new Pod arrives, it should provide information about the required resources. Depending on the required resources, the scheduler will assign a weight to each “priority function”.

We have included a new “priority function”, \( priorityFunc_4 \), which detects if the node includes the desired hardware and adds that information to the final score.

The scheduling is performed when a new Pod arrives, but we have modified Descheduler to move Pods that can make use of GPU when a GPU enabled node becomes available.

The default Descheduler implementation still does not support GPUs, so it was necessary to modify its implementation and to define a new type for GPU resources. We decided to use the type \textit{nvidia-gpu}, and to modify the Descheduler source code to recognize and use such type.

Once Descheduler has the capacity to recognize GPUs, it can execute the Scheduler to move, or create new Pods in the nodes that have now GPUs available.

The implementation also considers assigning more than one Pod to a node, if that node has the necessary resources. Also, if a Pod was assigned before to a node, and that node is not being used when the same Pod request arrives, the scheduler assigns it to the same node.
Finally, a Pod may demand a node with a GPU (and not just “prefer” a node with a GPU). In that case the scheduler will wait until a node with a GPU is available.

### 3.1 Validating the scheduler

In order to validate the performance of the new scheduler it would be necessary to create a new cluster and perform different experiments to measure the results. Although this approach is completely valid, it would require a large amount of time and resources, so we decided to develop a simulator in Matlab.

Through different tests, we could check that our modified Scheduler and Descheduler versions improve the results provided by the default ones, being able to obtain a better throughput and achieving a better utilization of the resources, as shown in chapter 4. When we talk about the throughput of the system we talk about the number of petitions that can be finished in a period of time. When using GPUs instead just CPUs the time that we spend in each node is lower, thus in the same period of time we can manage more petitions.

Matlab simulator provides realistic results, but some simplifications were made. For example, it doesn’t consider the time used by the scheduler to calculate the best node for a Pod (this time is much lower than the time used by the Pod during its execution), nor the time that is required by Kubernetes to deploy the Pod in the selected node.
In this section we will explain the different test completed to check the performance of our proposal. The test will be divided in two sections. The first section will show how the use of GPUs doesn’t penalize the performance of Kubernetes to orchestrate Pods. The second section will describe how services can be accelerated thanks to the new scheduler, and will analyze the performance through real experiments with Kubernetes and a with simulations performed with Matlab.

We will describe the results obtained in a real implementation, in which we have deployed a cluster and used our scheduler to move Pods to the best available node. In that test we
use Tensorflow \[7\], a machine learning library. This is a good use case for our solution, because this library performs complex operations that can be accelerated with a GPU. To test the solution we selected a benchmark, more concretely we used a Convolutional Neural Network \[8\], as it offers a good benchmark to compare the GPU vs CPUs.

4.1 Performance of Kubernetes with GPU support enabled

We wanted to check if Kubernetes and Docker are penalized by adding support for managing GPUs. The different tests performed let us conclude that adding this support does not decrease the performance. The time to deploy applications that make use of GPU acceleration is the same as the time required before adding the support for GPUs. Nevertheless, such applications are benefited from the use of GPUs to accelerate calculations. That occurs because during the deployments the GPUs does not perform any action it only be represent as a available resource or not.

4.2 Performance improvement with the new scheduler

The main objective for this project was to add support for Kubernetes to take advantage of GPUs to accelerate the Pods that could use them to perform operations. This way, it would be possible to make a better use of the resources. We modified the default scheduling architecture, so in order to analyze if we achieved that objective, it is necessary to compare the default scheduler with our proposal. In both cases we have enabled the GPU support, but the default scheduler only takes into account if there is an available GPU when a new Pod arrives.

It is necessary to define a mechanism to compare both schedulers. It would be possible to include mechanisms in Kubernetes to provide logs, and perform several experiments, but this would require a large infrastructure and a large amount of time, to perform a complete set of experiments, or experiments with a large number of nodes. So we decided to implement a simulator in Matlab that modeled both schedulers. Then we performed different simulations, with Pods that can be executed in CPU or in GPU.

In our simulation we didn’t consider the time spent to assign a Pod to a node, as this time is much smaller than the time spent by a Pod in the system, and in any case, it would be the same for both schedulers. We have considered also that the performance provided by a GPU, for the simulated Pods, is $\rho$ times the performance provided by a CPU, that is: $\rho = \frac{\text{CPUthroughput}}{\text{GPUthroughput}}$. 

22
4.2. Performance improvement with the new scheduler

4.2.1 Matlab simulation

We have implemented a simulator in Matlab to compare the performance provided by the default scheduler and our proposal (based on Descheduler, with support for GPUs) with different types of jobs. Then, we completed different simulations. In the first test we simulated 1000 Pods arriving following a Poisson process. The cluster has ten nodes, five with GPU+CPU, and five just with CPU. The Pods are simulated with different requirements: some could use a GPU and a CPU and other only a CPU. With the default scheduler, if a Pod that could be accelerated with a GPU arrives when all the nodes with GPUs are busy, then will be assigned to the first free node.

Descheduler was modified to prioritize the selection of nodes with a GPU over nodes with just a CPUs, for Pods that can be accelerated with a GPU. When a new Pod appears in the system, if there is a node with a GPU available, then the Pod is assigned to that node. Nevertheless, if there is no any node available, the scheduler then will try to move a Pod that only requires a CPU to another node without GPU, and launch the new Pod in the node with the GPU.

With Descheduler we can make a better use of the nodes’ resources while maintaining the capacity to execute Pods. This is achieved because before evicting a Pod that doesn’t take advantage of a GPU and that is being executed in a GPU node, while there is another Pod that requires a GPU waiting, we need to wait until the first Pod finishes. Despite the waiting time, in our simulation, we achieved better results.

We measured the difference between both scheduler for the same scenario, to understand how the execution time varies for each solution. An increment in the number of Pods implies an increment in the time that is necessary to complete their execution, but with Descheduler and its capacity to select GPUs the time spent is lower than with the default scheduler. Each simulation is repeated several times to obtain a mean of time that is represented in figure 4.2 where we mark the worst and better result for the same number of Pods.

In the next section we explain how we have obtained this simulator from real data.

4.2.2 Validation with real experiments

In order to check if our simulations were realistic, we also performed several experiments where we checked the performance of Descheduler in a real cluster. We tested how Pods are created and removed in a cluster where new nodes are added from time to time. That test shows how the system acts when new nodes appear and how by using GPUs can be advantageous even when there is a CPU with more resources available.

In the first scenario we analyzed two nodes with a GPUs. We started the experiment
Chapter 4. Results

Figure 4.2 – Default scheduler vs our modified version of Descheduler

Figure 4.3 – CPU node only
4.2. Performance improvement with the new scheduler

Figure 4.4 – Descheduler applied

with just the master and one node. Later, the second node is included in the cluster. Thus, at the beginning all the tasks have to be completed in one node. The GPU cannot be shared so, if it is being used, a second Pod wanting to use the GPU has to wait until the Pod that is running finishes.

We add the second node, but if we are using the default scheduler it will remain idle until new Pods arrive. Nevertheless, when Descheduler is executed we can observe how a Pod that needs a GPU for a better performance is moved to this new Node, the other remains in the previous node.

In the second test we started with just a node without GPU and later we added a new node with a GPU to the cluster (figure 4.3). All the Pods that required a GPU are moved to this new node (figure 4.4) because the Descheduler selects it as the best.

In a third test, a node without GPU executes a Pod. Then, two nodes with GPU are added to the cluster (figure 4.5). In this case the Pod is not moved while it is being executed, although the performance in any of the GPU nodes would be much better. Nevertheless, to make it possible to move a running Pod to a better node, we would have to implement a mechanism able to save the context of the running Pod and restore it to continue with the execution in other node.
Chapter 4. Results

Figure 4.5 – Two new GPU nodes added

In 4.7 shows the intermediate state of the cluster when the new node instances the Pod. As the execution of the previous Pod on the CPU has not yet finished, this will continue in it until the end of the current execution 4.6. With this we want to verify that there is no requirement by the system if a Pod is in use.

Figure 4.8 shows how the cluster looks after the tests, and we can conclude that Descheduler works well with GPU support. We modified the scheduler by adding support for GPUs, and including an algorithm that selects a node by his capacity prioritizing the existence of a GPU once the support is added.

We also tested the performance of the cluster by executing Alexnet Benchmark [8] with nodes with just CPUs or with CPUs and GPUs. We used the results obtained in our Matlab simulator to model the performance of the different nodes, as shown in figures 4.9 and 4.10, and also to validate the simulator by comparing its results with the results provided by the real experiments.

With that test we obtain that 3/4 of our petitions to the cluster will be executed in GPUs and the rest are executed in the CPU. This is expected because GPUs can complete the tasks in 1/4 of the time required by a GPU. Mean time of execution in GPU = 5.06 s in whole benchmark this is forward plus forward-backward. Mean time of execution in CPU = 20.64 s in whole benchmark this is forward plus forward-backward.

To calibrate the simulator we completed 20 iterations in the real benchmark. To compare the real test with the Matlab simulation we studied to the maximum time of execution for the whole test. This is the time spent in 20 petitions to the cluster. That petitions were launched at the same time. Thus, when all the resources are being used, enqueued tasks have to wait.
4.2. Performance improvement with the new scheduler

Figure 4.6 – Cluster after the tests

Figure 4.7 – Intermediate state with new GPU nodes
Figure 4.8 – State at the end of the execution with new GPU nodes

Figure 4.9 – Number of executions and time spend in forward benchmark
4.2. Performance improvement with the new scheduler

Our simulator provides parameters that can be adjusted, for example the CPU or GPU power. We can also configure the number of nodes in the cluster or the number of jobs that should be executed. By setting the same values that we have in the real experiment, the results obtained in both environments is similar. For example, for 20 petitions:

Real Test: 92.10 s
Matlab Simulation: 84 s

Then we complete a test with 1000 request in the real cluster and we compared the results with the ones provided by the Matlab simulator. The difference is less than a 10%. As we suspected the workload is lineal, because if we have always the same job the time spent will be increased with new petitions. In figure 4.11 we can observe that in an execution of 5000 request.

Figure 4.10 – Number of executions and time spend in forward-backward benchmark

Figure 4.11 – Matlab Simulation with 5000 jobs in 2 GPUs and 3 CPUs
Once we have validated the simulator we completed a more realistic simulation because in real clusters the tasks arrive at random times, and have different duration. The arrival of new tasks was modeled as a Poisson process with a mean $\lambda = 300s$. Now, when a new petition arrives, the system may have nodes free, so the scheduler can assign the new task to one of the free nodes or move one of the Pods that was running to the free node. The results is shown in figure 4.12

![Graph showing total time max - min for different numbers of jobs with two lines: Descheduler and Scheduler.](image)
5 Conclusions

As we described, Kubernetes is a powerful tool that facilitates the management of the deployment of tasks using Docker. Due to its modularity and open-source concept, it is easy to modify it and add new capacities. During the development of this thesis Kubernetes was updated from version 1.08 to the current version 1.17. In the latest version, there is more than one scheduler that can be selected to be used.

The main objectives of the master’s thesis were completed:

1. Add support for GPUs to Docker
2. Add support for GPUs to Kubernetes
3. Learn about how the Kubernetes scheduler works
4. Modify the scheduling algorithm to consider GPUs
5. Improve the default scheduler for Kubernetes
6. Compare the performance obtained with the default scheduler and our proposal

Thanks to the use of NVIDIA-docker we could use GPUs in Docker. Nevertheless, this tool only supports NVIDIA hardware so, as future work, it would also be necessary to provide support and test the hardware from other manufacturers such as AMD or Intel.

In order to support GPUs in Kubernetes, we had do include information about this new resource. Once this was done, the scheduler could consider the existing GPUs when assigning Pods. To improve the performance of the system we had to understand how to take advantage of GPUs behaviour. Nevertheless, this was not an easy task, because of the lack of information about how to change the default scheduler operation. However, Kubernetes also provides Deschedulder, which was designed to allow rescheduling
Chapter 5. Conclusions

operations. We used this functionality to force the reassignment of the Pods suitable for its execution over the GPU resources.

The designed scheduler can improve the performance of a cluster by assigning the tasks that can run faster using GPUs to the nodes that include one. This mechanism can be used in different applications. For example, Google has recently presented Stadia, a game streaming service. With tools like the one presented in this Master’s Thesis, it would be possible to optimize the execution of the games launched by the different users.

As a future work, it would be necessary to find a mechanism to move a running Pod from a CPU node to a GPU node without the need to finish it. Nevertheless, this would require saving the context of the running Pod, and restoring it in the new node.

There are more options for the acceleration of tasks. For example, NVIDIA has a system based on vGPUs \(^1\), which allows the virtualization of several GPUs in one. However, this solution is expensive and only compatible with the most powerful NVIDIA graphic cards. Other solutions are based on specific hardware, optimized to perform a task or a group of tasks as fast as possible. As the hardware is designed to optimize a specific task, these kind of solutions are difficult to update. In another way, there exist some solutions based on FPGAs (Field-Programmable Gate Arrays), which are more flexible, but they have limited resources. These systems would allow us to obtain a better performance than using GPUs, since they will be more optimized to perform certain calculations. However, we would lose the versatility that a GPU gives to us. Therefore it is a point to consider in the future.

---

\(^1\)https://www.nvidia.com/es-la/data-center/graphics-cards-for-virtualization/
This appendix shows the steps to install Kubernetes in a node to develop

A.1 Installation

This section contains all the steps needed to install Kubernetes with GPU support.

The hardware that was used is:

1. Laptop HP ENVY 15 j033s
2. CPU: Intel® Core™ i7-4700MQ ¹
3. GPU: NVIDIA 740M (GK208M) ²
4. RAM: 2x4GB DDR3 1600MHz
5. SSD: Crucial M500 240GB

The software was:

1. OS: Ubuntu 16.04.4 64bits
2. CUDA: 8.0 ³
3. Graphic NVIDIA Driver: 384.130
4. Docker: 17.03.2

²https://videocardz.net/gpu/nvidia-gk208/
³https://developer.nvidia.com/cuda-zone
Appendix A. Installation

5. Kubernetes: v1.10
6. TensorFlow: v1.9

A.1.1 First Steps

The first step comprises a fresh Ubuntu installation. We downloaded and installed the image from the Ubuntu web site with the specific version defined. When we are installing the system is important to avoid configuring a swap partition due the incompatibility of Kubernetes with swap.

A.1.2 NVIDIA Soft

That section describes how to install the NVIDIA software needed to use the GPU. It was necessary to download the CUDA software from the official repository.

First, it is necessary to delete the previous versions from the official repository of Ubuntu and add only the new version downloaded from the NVIDIA web site with the best fit drivers.

```bash
$ sudo apt purge nvidia-* && sudo dpkg -i cuda-repo-ubuntu1604-8-0-local-ga2_8.0.61-1_amd64.deb
```

This step added the NVIDIA repository to our installation, so we can install the latest CUDA version:

```bash
$ sudo apt install cuda
```

Then, we need restart computer. If the driver and CUDA was installed then we see in the terminal something like the figure A.1

A.1.3 Docker install

If we want to including GPU support, it is necessary to install a compatible version. To accomplish the Kubernetes requirements should be the version 17.03 or newer.

Then the fist thing that is required is adding Docker repository to Ubuntu:

```bash
$ sudo apt install apt-transport-https ca-certificates curl software-properties-common
$ curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo apt-key add -

$ sudo apt install docker-ce=17.03.2~ce-0~ubuntu-xenial
```

Once Docker is installed it is necessary to enable GPU support. For NVIDIA devices it is
A.1. Installation

necessary to install NVIDIA-Docker⁴.

If we install the last version of the NVIDIA Docker, it can cause incompatibility with Kubernetes. For that reason below are the correct command to be executed to obtain a stable Kubernetes installation.

First, we include the Kubernetes repository:

```
1 $ curl -s -L https://nvidia.github.io/nvidia-docker/gpgkey | sudo apt-key add -
2 $ distribution=$(./etc/os-release;echo $ID$VERSION_ID)
   list | sudo tee /etc/apt/sources.list.d/nvidia-docker.list
4 $ sudo apt-get update
```

Then, we install the compatible version of NVIDIA-Docker.

```
1 $ sudo apt install nvidia-docker2=2.0.3+docker17.03.2-1 nvidia-container-runtime
   =2.0.0+docker17.03.2-1
```

Finally, it is necessary to give permissions to the user that is executing Docker.

```
1 $ sudo pkill -SIGHUP dockerd
2 $ sudo groupadd docker
3 $ sudo gpasswd -a ${ USER } docker
4 $ sudo service docker restart
```

⁴https://github.com/NVIDIA/nvidia-docker
A.1.4 Check and optimize Docker

Once Docker is installed, it is necessary to check if everything is working as expected and includes GPU support.

```
$ docker run --runtime=nvidia --rm nvidia/cuda nvidia-smi
```

In the previous command it is necessary to specify what run-time is needed to execute the container. Kubernetes requires the NVIDIA runtime to support GPUs. To set the default run-time it is necessary to change change the daemon of Docker `daemon.json` file and add the next expression to the json:

```
"default-runtime": "nvidia"
```

```
$ sudo systemctl daemon-reload
```

Some times Docker does not recognize correctly the driver, so we need to specific the path to the executable file to allow Docker use CUDA programs. We change linux environment to add in PATH the libraries installed with CUDA.

```
$ export PATH=/usr/local/cuda-8.0/bin:$PATH
```

A.1.5 Kubernetes

For the installation of kubernetes it is necessary to include the Kubernetes repository:
A.1. Installation

Figure A.3 – Docker daemon

Figure A.4 – NVIDIA Docker
Appendix A. Installation

1 $ curl -s https://packages.cloud.google.com/apt/doc/apt-key.gpg | apt-key add -
2 $ cat <<EOF >/etc/apt/sources.list.d/kubernetes.list deb http://apt.kubernetes.io/kubernetes-xenial main EOF
3 $ sudo apt-get update && sudo apt-get install -y apt-transport-https curl
4 $ curl -s https://packages.cloud.google.com/apt/doc/apt-key.gpg | apt-key add -
5 $ cat <<EOF >/etc/apt/sources.list.d/kubernetes.list
6 $ deb http://apt.kubernetes.io/kubernetes-xenial main
7 $ sudo curl -s https://packages.cloud.google.com/apt/doc/apt-key.gpg | apt-key add -
8 $ sudo bash -c 'apt-get update && apt-get install -y apt-transport-https
9 $ curl -s https://packages.cloud.google.com/apt/doc/apt-key.gpg | apt-key add -
10 $ cat <<EOF >/etc/apt/sources.list.d/kubernetes.list
11 $ deb http://apt.kubernetes.io/kubernetes-xenial main
12 $ EOF

1 $ sudo apt-get install -y --allow-unauthenticated kubeadm kubelet kubectl

To initiate Kubernetes in localhost and allow it to use the Master Node as a common node and deploy containers on it, is need to taint that node:

1 $ kubectl taint nodes --all node-role.kubernetes.io/master-

With all the previous instructions, it is possible to start deploying some containers in the localhost. Thus, the same computer will be able to work as a Master and a node.
Bibliography


