HAALO:
A cloud native Hardware Accelerator Abstraction with Low Overhead

Jeremy Facchetti

Computer Science and Engineering, master's level (120 credits)
2019

Luleå University of Technology
Department of Computer Science, Electrical and Space Engineering
Abstract

With the upcoming 5G deployment and the exponentially increasing data transmitted over cellular networks, off the shelf hardware won’t provide enough performance to cope with the data being transferred over cellular networks. To tackle that problem, hardware accelerators will be of great support thanks to their better performances and lower energy consumption. However, hardware accelerators are not a silver bullet as their very nature prevents them to be as flexible as CPUs. Hardware accelerators integration into Kubernetes and Docker, respectively the most used tools for orchestration and containerization, is still not as flexible as it would need. In this thesis, we developed a framework that allows for a more flexible integration of these accelerators into a Kubernetes cluster using Docker containers making use of an abstraction layer instead of the classic virtualization process. Our results compare the performance of an execution with and without the framework that was developed during this thesis. We found that the framework’s overhead depends on the size of the data being processed by the accelerator but does not go over a very low percentage of the total execution time. This framework provides an abstraction for hardware accelerators and thus provides an easy way to integrate hardware accelerated applications into a heterogeneous cluster or even across different clusters with different hardware accelerators types. This framework also moves the hardware specific parts of an accelerated program from the containers to the infrastructure and enables a new kind of service, OpenCL as a service.
Acknowledgements

First and foremost I would like to thank my family, who has always been supportive to me during my studies. Whether I was studying in France, USA or Sweden, they have always been there for me during highs and lows and I don’t think I could ever express my gratitude enough to you. Je vous aime, vous êtes tout pour moi.

“Le suprême bonheur de la vie, c’est la conviction qu’on est aimé; aimé pour soi-même, disons mieux, aimé malgré soi-même.” -Victor Hugo

I would also like to thank Université de Poitiers, where I got my bachelor from. I have been able to acquire a strong skill base and I have had the chance to study in the USA thanks to them. Also many thanks to LTU where I studied for two years for my master’s degree. It was two great years and my stay in the great Norrland will not be forgotten. Finally, I would like to thank my managers at Ericsson for their amazing knowledge and advises, Ana with whom I formed the ”grumpy duo” and Catalin who was always here when I needed insights, for allowing me to write my thesis with them, I have learnt so much during these six months.

Thank you all so much, I would not be where I am right now without you. May the road ahead be full of joy and success.
Contents

1 Introduction 1
  1.1 Background ............................................. 1
  1.2 Research motivation .................................. 3
  1.3 Thesis aim and research questions .................... 4
  1.4 Thesis contributions .................................. 4
  1.5 Thesis outline ......................................... 5

2 Background and related work 6
  2.1 Network functions virtualization ...................... 6
  2.2 Hardware acceleration for NFV ........................ 6
    2.2.1 ASICs ............................................. 7
    2.2.2 FPGAs ........................................... 7
    2.2.3 GPUs .............................................. 9
  2.3 Hardware accelerators virtualization .................. 9
    2.3.1 GPU virtualization ................................ 9
    2.3.2 FPGA virtualization ............................... 10
  2.4 5G objectives ......................................... 11
    2.4.1 Enhanced Mobile Broadband or eMBB .......... 11
    2.4.2 Massive Machine Type Communications or mMTC . 12
    2.4.3 Ultra Reliable and Low Latency Communications . 12
  2.5 5G cloud architecture .................................. 13
  2.6 Kubernetes and Docker .................................. 13
    2.6.1 Docker and hardware acceleration ............... 14
    2.6.2 Docker isolation .................................. 15
    2.6.3 Docker and IPC ................................... 16
    2.6.4 Resource management in Kubernetes ............ 16
    2.6.5 Heterogeneous clusters management ............. 17
    2.6.6 FPGAs in Kubernetes ............................. 18
  2.7 Platform awareness in Docker and Kubernetes ....... 18
  2.8 Hardware virtualization and hardware abstraction .... 19
    2.8.1 Limitation of virtualization for heterogeneous hardware accelerated clusters .......... 20
  2.9 Breaking free from hardware virtualization .......... 20
3 Haalo: an Hardware Accelerator Abstraction with Low Overhead

3.1 Context ................................................. 21
3.2 Requirements analysis .................................. 21

3.2.1 Performance ........................................... 22
3.2.2 Integration ............................................ 22
3.2.3 Footprint .............................................. 23
3.2.4 Summary of solution requirements ................. 23
3.3 Motivations .............................................. 24
3.4 Overview ................................................ 25
3.5 Protocol .................................................. 26

3.5.1 IPC availables to Docker ......................... 26
3.5.2 Choice of IPCs ........................................ 28
3.5.3 Instantiation ......................................... 30
3.5.4 OpenCL Kernel execution ......................... 30
3.6 Thread model ............................................ 32

3.6.1 Request listener ..................................... 33
3.6.2 Notifications thread ................................ 33
3.6.3 HAALO role .......................................... 35
3.7 Security .................................................. 35

3.7.1 User based isolation ............................... 35
3.7.2 Access to devices ................................... 37
3.8 Resources management .................................. 37

3.8.1 Memory management ............................... 38
3.8.2 Accelerator management ......................... 39
3.8.3 Classic resources management ................. 39
3.9 OpenCL Kernels repository ......................... 39

3.9.1 Repository structure ............................... 40
3.9.2 Database content ................................... 41
3.9.3 Kernel serving api ................................ 41
3.10 Kubernetes extensions .............................. 42

3.10.1 Scheduling extensions ......................... 42
3.10.2 Choice of extension ............................ 47
3.10.3 Dynamic labeling ................................. 47
3.10.4 Scheduling decisions ......................... 48
1 Introduction

This section is an introduction to the master’s thesis. It first describes the current state of 5G and its need for hardware accelerators. It then describes the state of the art hardware acceleration and then presents the thesis objectives and purposes. Finally, a thesis outline is made.

1.1 Background

Since the introduction of the 1G networks in 1979, mobile networks have evolved in many ways to adapt to the changing use of the network by consumers. The current state of mobile networks is at 4G, but 5G is being developed to adapt the network to users and their evolving usage of cellular networks.

Some of the main motivations behind 5G development are for example the extreme growth of cellular data being produced and consumed as we can see in figure 1 and in [1] which show an exponentially growing cellular traffic, the upcoming massive cellular based IoT deployment as we can see in figure 2 or just the possibility to deploy new services using cellular networks (e.g. self-driving cars). These changes in the usage of cellular networks made of 5G a priority for network providers. 5G aims to bring to the mobile networking ecosystem better performances, better flexibility and better reliability than the previous mobile networks generations. To enable these enhancements, a wide restructuration has been applied to the previous 4G infrastructure to solve these problems.

One of these modifications to the infrastructure is the introduction of application containers to replace virtual machines currently in use in the 4G networks because of their better performances [4]–[6]. Containers are much more lightweight than virtual machines and use fewer resources. Container orchestration allows the infrastructure to be much more modular and flexible than it used to be but adds complexity to the infrastructure. The most widely used containers nowadays are Docker containers [7] and the most...
Figure 1: Ericsson’s prediction of global cellular traffic\cite{2}

widely used orchestration framework is Kubernetes\cite{8}. These technologies are actively developed by their respective open-source communities. Being open-source is a major advantage for these frameworks’ users as it is possible to extend or modify the behavior of the software to meet exactly their needs.

To enable better performance, network providers use hardware accelerators. An analysis\cite{9} shows Hardware accelerators provide multiple advantages over classical CPUs. First, hardware accelerators being developed for specific purposes perform much better than general purpose hardware such as CPUs. Hardware accelerators provide both better energy consumption (watts per bit) and better price for similar performances than general purpose hardware. However, hardware accelerators have significant drawbacks such as the lack of flexibility, and poor integration into existing orchestration and virtualization frameworks.
1. INTRODUCTION

Figure 2: Ericsson’s prediction towards cellular IoT connections

1.2 Research motivation

With the rise of wireless communications during the past few years and the upcoming 5G, more data is going to be processed. Hardware accelerators are well known for their higher performances and lower energy consumption than CPUs. However, even though hardware accelerators provide performances an order of magnitude superior to CPUs, they have a significant drawback which is the lack of compatibility. Each hardware accelerator has its own architecture and thus going from one accelerator to another may require large modifications or extremely long recompilation time, moreover, each hardware accelerator requires specific libraries and optimizations. As each hardware accelerator is different and does not always have the same libraries or interfaces than other accelerators, application developers currently need to take care of these different architectures or devices present in the cluster which is against the principles of virtualization, which aims to provide a unified platform to the developers and the applications. As of today, cloud orchestration relies heavily on Docker and Kubernetes, and thus this is why this thesis will focus on these frameworks.
1. INTRODUCTION

1.3 Thesis aim and research questions

This thesis focuses on the integration of hardware accelerators into the existing orchestration and virtualization frameworks and the implementation of a system solving this issue which will be described later on. The aim of this thesis is to provide a hardware agnostic integration of hardware accelerators into an environment containing both Kubernetes and Docker. To solve this challenge in the most efficient way, we have identified the current use and the current technologies involved into the hardware accelerated functions in a 5G context and then built a system aiming to keep the same level of performances while also enabling a better and more flexible usage of the underlying infrastructure thanks to Kubernetes and Docker. As Kubernetes does not have a developed platform awareness for hardware accelerators, this thesis will also aim to enhance Kubernetes’ platform awareness for hardware accelerators. To validate the viability of the system that has been implemented, performances will also be tested.

1.4 Thesis contributions

The contribution of this thesis is a proof of concept for a new kind of framework that enables access to hardware accelerators in such a way that Docker containers don’t need any information about the hardware that is present on the host. This brings multiple side effects, such as providing hardware specific code as part of the infrastructure and enabling sharing of hardware accelerators which is not possible in the current Kubernetes and Docker environment. The proof of concept’s source code is Ericsson internal and thus cannot be part of this thesis. However, an analysis and a description of the architecture will be provided, explaining the choices and decisions that were made during the course of the thesis.
1. INTRODUCTION

1.5 Thesis outline

In this section will be presented the main aims of the thesis in a summarized way.

• Background and related work
  This chapter will introduce the state of current hardware accelerators technologies, 5G requirements, virtualization and orchestration tool. Multiple kinds of hardware accelerators will be analyzed and their respective use cases will be identified.

• Haalo: an Hardware Accelerator Abstraction with Low Overhead
  This chapter will analyze the needs and usage of hardware accelerators in the case of 5G and especially network function virtualization. Multiple factors will be analyzed and taken into account to define the road map and objectives of the solution that will be implemented. This chapter will also present the architecture and explain the choices made during the implementation in order to satisfy the requirements and needs. Choices from protocol to the threads model or choice of IPC used in this project will be covered. Moreover, security of the implemented system will be analyzed. Because the system is running outside of the classic scope of virtualization and orchestration, some security measures have been taken to keep the system isolated and secure.

• Results
  In this chapter will be analyzed the performances of the system that has been implemented. It will include a benchmark on a real system with different kinds of accelerators and different kinds of functions.

• Conclusion and future work
  In the final chapter of the thesis will be discussed the contribution and the limitations of this master’s thesis. Lastly, future work for the implemented system will be drawn.
2 Background and related work

This section aims to describe the current and available technologies in the field of 5G. It first describes the purpose of network function virtualization and then looks into hardware accelerators and their characteristics. Finally, this section will analyze the cloud architecture and the virtualization technologies currently available.

2.1 Network functions virtualization

Network functions virtualization is the process of using software abstractions to reproduce network function using off the shelf hardware rather than specialized hardware. This process was motivated by the need of flexibility and the increased possibilities for innovations and modifications of network functions. Network functions virtualization trades off performances for flexibility and adaptability. NFVs help to reduce the costs, by removing or reducing the need for specialized hardware and by making use of the resource in the most efficient way. Network functions that are time limited for example do not require to have specific hardware at all time anymore, which reduces costs of the network infrastructure.

2.2 Hardware acceleration for NFV

Due to the architecture of x86 processor found in off the shelf servers but also due to the fact that some of the VNFs applications were designed for purpose-specific hardware and were not rewritten specifically for this architecture, performances of virtualized network functions are in some case either not performant enough, either too demanding for a x86 CPU and thus reduce CPU time for other functions. To tackle this problem, some functions can be delegated to hardware accelerators which are more specialized than CPU, and thus provide better performances. The more specialized and rigid the accelerator is, the higher performance and en-
2. BACKGROUND AND RELATED WORK

Figure 3: Flexibility vs performances

energy efficiency are. Different kind of hardware accelerators have different performances and flexibility as we can see in figure 3.

2.2.1 ASICs

Asics (or Application Specific Integrated Circuit) are very rigid accelerators, they are built for one purpose only and cannot be reconfigured once they come out of the factory. They however provide outstanding performances for the purpose they have been designed for and have the best performance to price and performance to energy consumption ratio. As they are not reconfigurable, or only very small parts of it are reconfigurable, they are not the most popular choice for virtualization related topics. For static network functions, they have the best performance to price ratio, far ahead of their competitors (FPGAs or GPUs for example).

2.2.2 FPGAs

FPGAs (or Field Programmable Gate Array) are hardware accelerators that are reprogrammable while giving very high performances. The process of modifying the behavior of an FPGA is however complex and require a
significant amount of time because it needs to physically reconfigure the transistors inside the FPGA to create new functionalities. Compiling for an FPGA is much more complex than compiling for a CPU, it can take up to several days to compile a program. FPGA frameworks for NFV are being developed because of their high performances [9], [11], [12].

FPGAs have the ability to execute OpenCL kernels using specific development frameworks such as Xilinx’s SDAccel [13] (as we can see in figure 4) or Intel’s FPGA SDK for OpenCL. Once such a setup is done, the FPGA can be called from a classic program as an OpenCL device. Due to the architecture of FPGAs, they are good contenders to execute these kernels, as they can be massively parallel (they can execute multiple instances of a piece of code at the same time, but can also use what is called pipe-lined parallelism), thus providing good or excellent performances. FPGAs have however a disadvantage, compiling the OpenCL kernels is much longer than compiling a kernel for a more general device such as GPUs or CPUs.

Figure 4: A Xilinx fpga configured as an OpenCL device [14]

Some modern NICs (Network Interface Controller) include FPGAs in their architecture. By doing that, some network related functions can be offloaded directly to the NIC, freeing CPU from computation heavy func-
tions. By integrating accelerators directly into the network controller, it also reduces the data flow in the system.

2.2.3 GPUs

GPUs are the most widespread kind of accelerators, they provide higher performances than a CPU for certain workloads, are easily reprogrammable, and more affordable than ASICs and FPGAs. They however do not provide the same level of performances than the accelerators mentioned above. GPUs are very efficient are executing parallel code (parallel meaning independent and similar), typically using OpenCL or Cuda kernels to do so.

2.3 Hardware accelerators virtualization

Virtualization for CPU has been around for a significant time already. Virtualization for hardware accelerators is more complex because there are many different architectures and paradigms for these accelerators. Accelerators of the same kind (for example two FPGAs) might not follow the same paradigms. Some kind of accelerators do not have architectures that allow them to be virtualized (ASICS, for example, do not make good candidates for virtualization)

2.3.1 GPU virtualization

GPU virtualization is a relatively new concept compared to CPU virtualization. Different virtualization techniques exist and are most of the time linked to GPU vendors. Some vendors implement SR-IOV\cite{15, 16} (such as AMD\cite{17}) into their GPUs, to allow each GPU to appear as multiple GPUs, then physically splitting the memory and the compute units inside the GPU to provide different execution environment to the programmers. Some other vendors will use proprietary software implementation to provide
virtualization to their GPUs (such as NVIDIA[18]). These virtualization techniques have however the same limitations, once a virtual GPU is allocated, it cannot be dynamically resized. Differences between the two virtualization techniques are exposed in figure 5.

2.3.2 FPGA virtualization

FPGA virtualization is not yet to the point of CPU or GPU virtualization. While modern FPGAs implement the SR-IOV specifications, the very nature of FPGAs is somewhat slowing down the process of virtualization. SR-IOV enables the FPGA to present different partitions as different devices to the host device. Moreover, most modern FPGAs also implement dynamic reconfiguration, which provides a certain modularity to its partitions. Each partition can have one or more subpartitions that are reconfigurable on the fly enabling run time change of behavior for the device. However, each partition needs to have a non-reconfigurable part, so the possibilities stay somewhat limited. However, some proof of concepts of virtualization making use of SR-IOV and dynamic reconfiguration are being developed. These virtualization techniques will have the most generic static part possible, then will reconfigure on the fly the dynamic
2. BACKGROUND AND RELATED WORK

Figure 6: 5G requirements and enabled use cases \[21\]

subpartitions to change the behavior of the FPGA.

2.4 5G objectives

5G is the next generation of mobile networks. It comes after 4G LTE. 5G aims to improve data rate, latency, energy consumption, system load and capacity \[19\] \[20\], Figure 6 also shows some of the possible applications of 5G. Its full deployment is scheduled for 2020.

2.4.1 Enhanced Mobile Broadband or eMBB

eMBB requirements are the following:

1. Peak data rate: from 10Gb to 20Gbps
2. Minimum of 100Mbps at any time
3. High Mobility support: up to 500 Km/h

These requirements were defined to enable data-driven scenarios requiring high data rates across large areas such as 4k streaming, 3D videos, augmented reality or cloud-based workstations or gaming stations. To follow these requirements, networks will mainly require to improve the Watt per bit ratio and also take care of its scalability. A more detailed view of the possibilities enabled by 5g can be found at [22] and [23].

2.4.2 Massive Machine Type Communications or mMTC

Massive Machine Type Communications requirements were defined to improve networks where the density of network-connected devices is high (up to 1 Million per square kilometer). Handling this kind of density will enable new applications, such as smart cities, medical sensors, and many more but will also benefit traditional network connected devices.

2.4.3 Ultra Reliable and Low Latency Communications

Requirements for uRLLC are the following:

1. 1ms latency over the air round trip
2. 5ms end to end latency between terminal and enodeB
3. Availability of 99.9999%

Networks able to provide these requirements will allow applications where reliability and latency are critical including but not limited to self-driving cars, remote surgery, industrial automation, or collaborative robots. Improving latency to such an extent will require to maximize the packets per second ratio.
2. BACKGROUND AND RELATED WORK

2.5 5G cloud architecture

To make some of the 5G objectives possible (such as the ultra-low latency requirement for example), a specific cloud architecture has to be used. The cloud platform needs to be distributed to provide the best compromise between proximity and processing power. Services that require ultra-low latency will typically be provided from the closest cloud points of presence because network transit would make the latency too important. Services that require a moderate latency and a more important processing power will be directed to more distant points of presence. Typically, the more processing power needed, the further the computation will happen, and the smaller latency the closer the computation will happen. One of the challenges of 5G will be to properly dynamically schedule these services into these different points of presence.

2.6 Kubernetes and Docker

When it comes to managing cloud resources, Kubernetes and Docker are the preferred solutions for many use cases. Docker is a container technology, which isolates the application from the host system, providing a predictable environment to the application. A docker container is different from a virtual machine (see figure 7). Virtual machines are a whole operating system, which runs a kernel and includes all the files present in the operating system. A Docker container doesn’t include the kernel, but only the binaries and the libraries needed to run an application. While a virtual machine will run its kernel on top of the host’s OS kernel, Docker containers will directly run on top of the host’s kernel. This provides better performances, because the docker engine is much lighter than the virtual machine engine and the additional kernel, and also results in a more lightweight image (some lightweight images are only around 5MB). When using Docker, applications only depend on the container itself and thus can be launched on any device or operating system. This is specifically useful for cloud scheduling because operating systems or hardware might differ from one location to another.
Kubernetes is an open source scheduling application. Kubernetes is widely used for scheduling docker containers and managing clusters in the cloud. While it was designed to be used with Docker, it is now possible to use it without Docker, this gives users the choice of the containerization technology. Kubernetes is able to manage many aspects of clusters, from load balancing to resource aware scheduling. It is easily extendable via plug-ins and thus can manage a wide range of clusters with specific needs. Kubernetes architecture is such that it is also easy to extend or modify existing behavior by rewriting or modifying some modules. It is also possible to manage federated, distributed or hybrid clusters with Kubernetes. This framework fits perfectly with the most widely used cloud platforms (OpenStack, AWS or GCE for example) and can even manage clusters distributed among multiples of these aforementioned cloud platforms.

These characteristics make Docker and Kubernetes first-class citizens in the cloud era because it allows to have a very strong control over the infrastructure that can be programmatically defined. Deployments become much easier too because developers do not need to care about the actual infrastructure and underlying system being used to run their applications but can rely on an abstraction of it.

2.6.1 Docker and hardware acceleration

While Docker has the ability to run hardware accelerated programs, it can introduce some problems to the developers. First, the concept of ”package once, run everywhere” that Docker implements is somehow limited when using hardware accelerators. This is due to the fact that hardware accelerators, by their very nature, will require different setups, environments or drivers to work properly. This makes the use of Docker more complicated, as the application packager will have to take care of the hardware the container is running on, which will reduce the efficiency and utility of Docker. Some solutions have been developed by accelerator vendors, such as Nvidia, to tackle this problem. Nvidia developed a Docker container image optimized to run applications using their graphics cards [24], but also
2. BACKGROUND AND RELATED WORK

Figure 7: Differences between docker containers and virtual machines to integrate it into Kubernetes [25], which limit the impact of the limitation described earlier. While this solution solves many problems in the context of homogeneous clusters, this kind of solution is limited for heterogeneous contexts, as it limits the user to a specific type of hardware and a specific brand of hardware, which is problematic when dealing with heterogeneous cloud environments.

2.6.2 Docker isolation

Docker provides an isolated run-time to applications through the docker engine. To the difference of virtual machines, the docker does not contain a complete operating system but relies on the host operating system (more specifically on the host’s kernel). A container will only contain the necessary libraries and binaries to run the application. Relying on the host’s kernel does not only allow lighter images but also reduces the overhead there will be only one kernel will be running for each host, no matter how many containers are running. System calls will not go through multiple kernels and this allows better performances and an overall lighter charge on the host system.
2. BACKGROUND AND RELATED WORK

2.6.3 Docker and IPC

A Docker container can be launched with certain configuration parameters that allow for the container to access the host’s system through IPC namespaces, this is allowed by the fact that containers are using the host’s kernel, thus can interact with other processes. The container can either directly access the host IPC namespace or another container’s namespace. Docker containers can also be launched as users, this provides control over what the container can access on the host. To enable an easier policy management, containers can also be part of UNIX groups. IPCs include UNIX domain sockets, shared memory blocks, POSIX message queues and many more. Using IPCs in Docker containers enables containers to communicate with other containers or with the host, at the price of reducing the isolation provided by Docker.

2.6.4 Resource management in Kubernetes

Kubernetes allows to manage resources per node, default resources are CPU and memory. These resources ensure that each pod (a group of containers) gets a sufficient share of the resource to properly run. A node (server on which pods are scheduled) cannot be assigned pods if the total amount of resources that it has would be overflowed. Custom resources can be added to Kubernetes (referred to as extended resources in Kubernetes docs). Custom resources allow nodes to advertise resources that are not present in the default Kubernetes resources (default resources include for example CPU, Memory, whereas GPUs are not present as a built-in resource). Once these resources have been added to the cluster or to the nodes, they can be advertised, consumed or released exactly as built-in resources would be. There are two ways to advertise a new kind of resource, the first one being creating a device plug-in and the second one being a stand-alone program running on nodes that will send HTTP request to the API server to add or remove resources as they are used or not. Device plug-in is a way of managing vendor-specific resources without modifying Kubernetes code. Device plug-ins have however a limitation which is that it is not possible to
have or use decimal resources and that containers are not able to share the resource, which means that for example if the new resource is a GPU, only one container will be able to use that GPU. Device plug-ins were designed to be used by GPU, NICs or FPGAs. Both of these solutions have their advantages and drawbacks, the main factor to chose one over the other is the level of control needed over these resources. Device plug-ins are easy to use, already written and easily manageable, however, they are not as flexible as classic extended resources and are vendor specific while classic extended resources are much harder to configure, but can provide vendor agnostic support and a more granular control.

It is also possible to manage cluster-wide resources. These resources are usually managed by a scheduler extension, but can also be handled by the default one. Pods will be scheduled only if all the resource requirements are fulfilled. If a schedule extension is used, it will be up to this extension to chose on which node to launch the pod. These resources can be split and used just like classic resources such as CPU and memory.

2.6.5 Heterogeneous clusters management

Kubernetes and Docker (and, more generally, containers), allow the management of heterogeneous clusters. Heterogeneous clusters are clusters made of different kinds of appliances. Managing heterogeneous clusters means that the orchestrator (Kubernetes here), has to be context-aware, and schedule and launch containers on the right appliance to give the best overall performance to the managed cluster. Kubernetes is able to manage classic (CPUs, memory, etc.) but is also able to take other resources into account when scheduling containers across the cluster. However, when using extended resources (resources that are not present in the default Kubernetes implementation), Kubernetes does not know what the resources represent, which forces the cluster manager to define and advertise their own resources (for example, if a particular node has a dongle which should be available to the cluster, some extra steps need to be taken to advertise it as available to the scheduler).
2. BACKGROUND AND RELATED WORK

2.6.6 FPGAs in Kubernetes

While there is some support for FPGAs in Kubernetes thanks to device plug-in provided by the FPGA vendors\[^{26}\], there are still some limitations. Needs for elastic provisioning of hardware accelerators in NFVs environments have been identified\[^{27}\], but such a system is yet to be designed and implemented. Either each FPGA is advertised as one resource and thus can only be used by one container, even though the FPGA would only be partially used, either the FPGA’s partitions are static and cannot be reprogrammed which limits the flexibility of the platform. The current integration does not allow dynamic use of FPGAs and is vendor and device specific. The current state of Kubernetes integration for FPGAs does not allow any heterogeneous clusters. This is due to the fact that different FPGAs can be massively different from an architectural point of view, thus limiting the common parts. Moreover, even though major FPGA brands have SDK for common languages (C++, OpenCL, C, etc.), compiling this code into FPGA understandable format takes a significant amount of time (can often take several hours to several days) due to the fact that it needs to configure the hardware during the compilation, and this is a very complex problem to solve to have an effective configuration, so run-time compilation as it is the case with CPUs or GPUs is not possible as it would be massively inefficient. This compilation length forces users in dynamic environments to precompile the OpenCL kernel and include it inside of the containers if it needs to be run on different devices.

2.7 Platform awareness in Docker and Kubernetes

Kubernetes provides some platform awareness, however, this awareness is limited to "cloud native" resources such as CPUs, memory, virtual networks. Docker as such is just a virtualization tool and thus does not really have to implement platform awareness. Kubernetes’ platform awareness relies on an abstraction of resources but lacks a hardware accelerator abstraction. To make use of hardware accelerators in a Kubernetes cluster, Docker containers and Kubernetes nodes need to be aware of the exact kind
2. BACKGROUND AND RELATED WORK

of hardware accelerator that is present in that cluster. This is limiting because any change of hardware would force changes into the containers. All the driver libraries and device specific code have to be present into the containers and thus, if multiple kinds of hardware accelerators are present, it would require to either have multiple driver libraries into the container or have different containers to run on specific nodes. This introduces complexity to the infrastructure and in the development of applications. This thesis project aims to enhance platform awareness for hardware accelerators in Kubernetes to facilitate the use of hardware accelerators, both enabling an easier sharing of the resources and by abstracting the hardware accelerators.

2.8 Hardware virtualization and hardware abstraction

Virtualization and abstraction are heavily used in the context of cloud computing because it simplifies the interaction between the user and the hardware. While the two concepts are close, there are some notable differences. Virtualization provides virtual devices to the users, letting them use them as if they were physical devices. This allows users to share resources while not having to take care of resource management. Virtualization of hardware is often static, with the hardware device being split into a fixed number of virtual devices. Abstraction does not aim for the same goals. Abstraction tries to provide a layer on top of the hardware to simplify its interaction from the users’ perspective. Abstraction does not expose the device to the user, it provides an interface to interact with it, which allows a user to use hardware without having all the information or access to it.
2. BACKGROUND AND RELATED WORK

2.8.1 Limitation of virtualization for heterogeneous hardware accelerated clusters

While virtualization allows multiple virtual machines to share the same hardware (CPU and memory for example) in a native way, the very nature of heterogeneous hardware accelerated clusters creates a problem that SR-IOV or classic virtualization cannot address because the whole device is exposed to the virtual machine or container. When needing to access the device, the virtualized environment is required to know about the device exposed to it and to have the proper hardware specific libraries to be able to access it. It is not possible to use that approach in a wide cluster where hardware can be updated frequently, and where hundreds or thousands of virtualized environments are deployed. Updating the hardware in such a context would require to update every environment that could potentially run in it.

2.9 Breaking free from hardware virtualization

As the current technologies that are currently in use have significant drawbacks and limitations, a new solution needs to be designed to efficiently integrate hardware accelerators into a Kubernetes cluster. As we have seen in the precedent sections, hardware virtualization is not efficient enough nor flexible enough to make an efficient use of hardware accelerators in a Kubernetes cluster, and thus we will need a better solution for integrating all of these devices into a Kubernetes deployment. This solution will be described in the next section and choices that affected the architecture and the idea will be provided.
3. Haalo: an Hardware Accelerator Abstraction with Low Overhead

3.1 Context

To put things in context, we will briefly describe the overall state of the current integration of hardware accelerators in a Kubernetes cluster. First of all, all of the section assumes that we are evolving in a Kubernetes cluster running Docker containers (see Fig.8). The daemon will be running on what is referred to as minions on the figure. In our case, minion nodes will also have an OpenCL accelerator of some sort (FPGA or GPU for example). When referred, containers are containers that are scheduled by the master node and running on any of the minion nodes.

Figure 8: Overview of a Kubernetes cluster

3.2 Requirements analysis

In this section, we will analyze the requirements of a hardware abstraction for network functions virtualization. We will first analyze the performance
3. HAALO: AN HARDWARE ACCELERATOR ABSTRACTION WITH LOW OVERHEAD

requirements, the integration requirements and finally, the footprint requirements of a system that solve the problem described earlier.

3.2.1 Performance

Virtualized network functions brought a lot more flexibility than there previously was, however, the very fact that VNFs often run on off the shelf hardware and not specialized hardware reduces the overall performances and this is becoming a problem to network providers. In the case of 5G, hardware accelerators are used to increase bits and packets per second because of the increasing throughput needs but also to reduce latency and increase the packets per second rate. These needs define some of the requirements that the solution that we will implement needs to follow. First, it needs to add very low overhead to keep the packets per second as high as possible as we have defined in the requirements analysis. If not followed, this solution could not be used in 5G because then the 5G requirement of ultra-low latency would not be achieved. Second, the solution should provide very high throughput not to limit the capabilities and performance gains provided by the hardware accelerators.

3.2.2 Integration

There are many network providers and many different hardware accelerators that are being used. Sometimes these hardware accelerators are not compatible with each other and one executable or one driver library working with one hardware accelerator will not work with another one, even if they are from the same manufacturer. Moreover, hardware accelerators from multiple vendors and even multiple hardware accelerator types are often present in a cluster, if not on a single server. The solution that will be implemented needs to provide seamless use of any hardware accelerator in the cluster to provide the highest potential resource usage in the cluster. By doing this, it will not only enable more computing and thus more throughput in the cluster, but also simplify the resource management.
3. HAALO: AN HARDWARE ACCELERATOR ABSTRACTION WITH LOW OVERHEAD

Last but not least, if the solution we implement can handle multiple device types or vendors, it would highly simplify deployment and implementation for developers.

3.2.3 Footprint

Hardware accelerators are used not only because they provide better performances than classical CPUs, but also because they allow processing to be offloaded from the CPUs which frees computation time for CPUs specific workloads.

To keep this aspect of freeing CPU time, the solution that we will implement needs to have little to no impact on CPU. By doing this, the solution will allow the use of hardware accelerators offloading capability as well as their ability to increase the overall performance of the system.

3.2.4 Summary of solution requirements

To summarize what was said in the previous sections, the solution being implemented in this thesis has to meet the following requirements:

- Low overhead
- High throughput
- Hardware agnostic
- Low footprint

The design of the system will thus be influenced by the aforementioned requirements and will aim to meet each of them.

To solve the challenges described in section 1.2, an abstraction for hardware accelerators was implemented. The choice of building an abstraction instead of using existing virtualization techniques has been induced by the
3. HAALO: AN HARDWARE ACCELERATOR ABSTRACTION WITH LOW OVERHEAD

heterogeneous architectures of hardware accelerators. To the contrary of CPUs where only a few architectures are available (x86, x64, arm), hardware accelerators have a wide range of architectures (GPUs, ASICs, FPGAs, smart-NICs, etc...), programs compiled for one brand of an hardware accelerator is often not compatible with another brand even if the two hardware accelerators are of the same type. Containers in a cluster would have to know about every single device they can run on to be able to be run in a cluster. Moreover, a hardware accelerator with a new architecture being installed in a cluster would require every container to be rebuilt and adapted to this hardware accelerator. Because of this, a solution based on virtualization of the hardware is not a viable solution. Instead, the solution that I implemented implements a unified interface between the container and the hardware accelerator. With this solution, when a new hardware accelerator architecture is introduced in the cluster, only the interface backend needs to be updated, and the same containers can run on the new hardware.

3.3 Motivations

The motivations behind this interface are the following:

1. Updating an interface, or implementing the interface for a new device requires significantly less time than rebuilding and modifying all the containers in order to access a new accelerator.

2. The code inside the container will be much easier than having to handle multiple accelerators. With this interface, the same code is able to run on every kind of accelerator if the interface has been updated to handle these accelerators.

3. Orchestration is much easier when the logic is handled by the host directly, instead of being handled by the orchestrator and modifying command being run on the nodes by the orchestrator (this would have been an issue at run time to mount devices into the container from a command directly issued by the orchestrator and not the host.)
4. With such an interface, dynamic provisioning of resources to containers would be possible without giving containers total access to the host (privileged mode). Furthermore, the logic necessary for that would not have to be inside the container but on the host directly.

3.4 Overview

In this section, we will discuss the top level architecture of the daemon that has been implemented in this project. The daemon is the main part of this system has it allows the execution of code on an accelerator from the virtualized application. We will first describe the protocol used, then explain the thread model and finally explain the choice of IPCs.

The daemon that has been implemented acts as a man in the middle between the accelerator and the containers as shown in Fig 9. It will communicate with both the container and the accelerators and synchronize their actions. Fig 10 shows a top-level view of a server running HAALO.

![Diagram](image_url)

Figure 9: High level description of Haalo
3. HAALO: AN HARDWARE ACCELERATOR ABSTRACTION WITH LOW OVERHEAD

Figure 10: HAALO overview

3.5 Protocol

This section describes the protocol designed to enable the execution of code on the accelerator from the docker container. The protocol design is very important as this is the way for the container and the host to synchronize themselves and communicate essential information. Specific IPCs have been chosen, and the reason for their choices will be explained. Also, the overall protocol will be described. An overview of the execution of the protocol is given in Fig. 11.

3.5.1 IPC availables to Docker

There are multiple IPCs (inter-process communication) available to Docker containers to communicate with the host the container is running on.

- Unix sockets: Unix sockets are available to Docker container via volumes. As Unix sockets are represented by files in a Unix environment. To access such a socket, the socket file needs to be mounted
into the container. Once the file is mounted into the container, it is fully available to the container and a bidirectional connection is available between the container and the host.

- **Abstract Unix sockets**: Abstract Unix sockets are not represented by a file but rather by a name. This kind of socket cannot be used with the same procedure as a classic Unix socket. As there is no file representing the socket, one cannot mount it into a container. However, they are available to Docker containers if the host’s network stack is exposed to it. This is a significant drawback as exposing the host is considered unsafe. They however have a significant advantage over network sockets that we will see later on, data does not go all the way to the network stack, which gives better performances than classic network sockets.

- **Network sockets**: Network sockets are classical sockets (UDP and
TCP sockets). To have access to these sockets and communicate with the host, these sockets also require to have access to the host’s network stack.

- **Other IPCs**: The other IPCs are available with the ”IPC” option of docker, this option has multiple possible values, including the ”host” value which allows sharing different kinds of IPCs (e.g. shared memory, messages queues, semaphores) between the host and the container. Most of the classic IPCs will fall under this category.

### 3.5.2 Choice of IPCs

This section describes the process leading to the choice of IPCs (inter-process communication) used in this project. Given the performance constraints that there are on this project, every IPC has to be chosen carefully and multiple factors have to been taken into considerations before choosing an IPC.

- **Protocol instantiation**: To instantiate the protocol, a Unix namespace socket has been chosen. Unix namespace sockets provide multiple advantages over classical network sockets. First of all, they are described by a file and not by an IP address and a port. Because of this fact, it is possible to control the group or the user id of the processes which access the socket. Secondly, as the primary target for this system is containers, giving access to a file is more secure and has fewer constraints than exposing the host network stack to a container. When the client is launched, it needs to have access to the socket file to be able to start the protocol. The socket is of type *SEQ_PACKET* which allows having a defined size packets flow, but is also connection-oriented, which allows getting information about the corresponding process. This allows the host process to check whether the client trying to access the service is allowed or not.

- **Instructions sharing and notifications**: To share instructions and notifications between the host and the container, the daemon and
3. HAALO: AN HARDWARE ACCELERATOR ABSTRACTION WITH LOW OVERHEAD

the container are both using shared memory segments. Shared memory segments allow us to share information between processes (here, container and host) only using user-space. While other IPCs are more convenient and less error-prone, shared memory blocks are the only IPC which does not require the data to go through the kernel. As the classic Linux kernel is not a real-time kernel, using classic IPCs using the kernel can add up to 10ms. Also, shared memory is an order of magnitude faster than other IPCs. In the case of 5g and ultra-low latency networking, this kind of delay is not acceptable and performances and quality of service would not meet the expectations for 5g. To share instructions, an instruction block is created in the /dev/shm directory. /dev/shm is a special directory which is a tmpfs (temporary file system). Tmpfs are not meant to survive a reboot, and are not always located on disk as a typical file system would, but can also be in the RAM. Thus a file in /dev/shm, is a shared memory block. It can be accessed by any process having the rights over the file representing the segment. It is extremely important for latency sensitive application to avoid any kind of context switch because context switches are controlled by the kernel and thus threads might be paused and resumed, which is a process that takes time and adds latency for each context switch.

• Data sharing: To share results, shared memory has also been chosen. Shared memory has been chosen for that function to enable a zero copy and thus a high throughput. Avoiding copies is very important as the overall throughput might be extremely high, and copying data multiple times would introduce a performance hit to the system. Shared memory blocks allocated to receive bulk data and results are defined in the instruction block by a name, which represents a file in the /dev/shm directory. Once the name is retrieved, a memory map of the file is made into the application space. This allows the application to access the data as a classical array. Another advantage of using shared memory for that case is that the overall memory usage will stay very limited and will only be present once in the system. When high volumes of data are treated by the server,
this is especially important. Lastly, no other data transfer is needed, except a notification that the results are ready to the container and results will be instantaneously available.

3.5.3 Instantiation

First, the client (the container), instantiate a connection through a Unix socket, which is mounted into the container when the container is launched. This allows having a static path inside the container to access the socket. Once the connection is instantiated, the host (server), creates memory blocks in the shared memory segments. Once these blocks are created, the ownership is given to the client. The client also sends its name through the socket, so the host can retrieve the different OpenCL kernels corresponding to the best device (figure 12) from the kernels repository (figure 13) and compile them [39] if they are not already. Once all the kernels are retrieved and ready to be fed to the OpenCL device, the host sends to the client the id that it has assigned to the client. This way the client knows what files represent its instruction blocks and its notification blocks.

3.5.4 OpenCL Kernel execution

In this section, references to OpenCL kernels are made, an OpenCL kernel is the piece of code that is offloaded to an OpenCL accelerator (e.g. GPU or OpenCL enabled FPGA).

Once the instantiation is done, the client/server duo enters an infinite loop where the server waits for an instruction from the client, and the client fills an instruction block when it needs computation to be done by the hardware accelerator, before notifying the host, the client fills shared memory blocks with data. Once the host detects that a new instruction has been posted, it retrieves the OpenCL kernel’s name and executes the corresponding binary on the hardware accelerator. The hardware accelerator is given the memory addresses of the shared memory. The host then waits for the hardware accelerator to be done and then notifies the client.
3. HAALO: AN HARDWARE ACCELERATOR ABSTRACTION WITH LOW OVERHEAD

Figure 12: Device initialization
that the computation is over. Both client and server will loop through this process until the client is killed.

### 3.6 Thread model

This section describes the thread model chosen for the system. The thread system is very important for overall performances because it allows execution to be done concurrently and enables high-performance systems. However, threads introduce complexity and need to be synchronized to avoid
race conditions and erroneous results and thus need to be used carefully.

3.6.1 Request listener

Performance and footprint drove the choice of the thread model. The thread model that has been chosen for this abstraction needed to keep latency as low as possible to enable its use in a 5G context, while having a low footprint to keep as high as possible the resources available to the containers running on the servers. In order to minimize the latency, threads are polling for new requests in a loop. While this keeps one core busy 100% of the time, it avoids the use of interrupt-driven stack, which would add latency overhead\cite{40} (in a default configuration, Linux can add up to 10ms of latency when a thread is interrupted) which are not usable in an ultra-low latency context. Additionally, to avoid any kind of interruption, the threads are pinned to physical cores and given a real-time priority so that no other process can run on the same core. To limit the footprint, only a limited number of threads will be deployed on the server to poll for new requests, while this can increase the latency from host to container, this allows the aforementioned container to have more CPU available to them as these containers’ purpose are not only to run the hardware accelerator but also to run CPU applications. The request polling thread’s event is described in Fig. 14.

3.6.2 Notifications thread

To notify that the results of the OpenCL kernel are available to use, a thread is allocated to poll the device(s) for new results in each OpenCL queue. This polling is done asynchronously thanks to markers placed in the queue by the request listener. This marker is an event which will indicate whether all the previous commands in the queue where executed and completed. If a marker status is "CL_COMPLETED" the thread in charge of the notification will inform the container that the result is ready to be used. As for the listener thread, polling for new results is consuming
Figure 14: Listener thread event flow

a whole physical core, the process also having ultra-high priority level to guaranty an ultra-low latency between the results coming out of the accelerator and the container being notified of the presence of the results. The client process has multiple choices when it comes to getting the results back, either it polls for new results to get the results as soon as possible, either it can process in an asynchronous way to check for new results. The latter is to be preferred when the accelerator is used to offload the CPU from compute heavy parts, while the former is to be preferred when the accelerator is used to decrease the latency of a computation. The notifier thread’s event flow is described in Fig. 15.
3.6.3 HAALO role

As we have seen in the precedent subsections, the protocol that we have designed mainly focused on synchronizing the containers and the hardware accelerators. Fig. 16 shows the global overview of the different modules that are specified in HAALO and handle the synchronization between the different protagonists in a cloud-native hardware accelerated application.

3.7 Security

In this section will be discussed the security choices and the drawbacks of the system implemented. We will first review the user based isolation for shared memory blocks and access to the daemon. Then we will review the access to devices.

3.7.1 User based isolation

As the containers and the host are sharing the same IPC namespace, containers could potentially interfere with or read the content inside other container’s shared memory segments. To prevent this issue from happen-
3. HAALO: AN HARDWARE ACCELERATOR ABSTRACTION WITH LOW OVERHEAD

Figure 16: HAALO detailed overview

ing, each container is given a unique UNIX user UID. On creation, the host’s daemon will create all the required shared memory blocks to communicate with the container and then set the ownership to the container. The write/read permissions are set to 600 (read and write access for the owner, no access for the members of the group of the owner, no access for anyone else). As the host’s daemon is running as the user with UID 0 (root user or superuser), these permissions do not affect the host’s daemon possibility to read or write to the shared memory block. These settings prevent access to results or instructions blocks from any other entity than the container’s user and the root user. Another advantage of using user-based isolation is the ability to verify a block’s owner during run time and thus the fact that the host can check if a block has been created according to the rules defined by the host or not. For example, one could add a rule concerning the number of blocks allowed in the memory, or a rule to limit the amount of total memory that the containers can allocate in the shared memory. As the daemon is running as the root user, it has the ability to
control and remove blocks from the shared memory if and when needed. The daemon could also terminate containers if these rules are not followed so the system does not run out of memory.

On conflict over a block (e.g., two containers try to create blocks with the same name) the permission of a block will protect it from being overwritten and thus prevent data corruption.

### 3.7.2 Access to devices

As the devices are not accessed by the containers directly, we can have fine control over what is being fed to the device. Another point that is fixed by our implementation is the fact that the containers can dynamically have access to multiple devices without running in the privileged mode. When a docker container runs with the privileged mode, it can do everything that the host can do, including mounting and unmounting devices which is highly insecure. Avoiding running in privileged mode keeps the containers isolated and thus keeps the system secure. Moreover, keeping the devices’ libraries out of the containers makes it easier to update and control which libraries are being used and ensure that the library being used is a secure one. Thus a container without any hardware specific library nor specific initialization can run any device provided the infrastructure is aware of such a device.

Another point that increases security is the user checking when a container asks for a device. As the daemon can check for the user id before it allocates the device, it is possible to restrict access to devices that are to be isolated or used exclusively by one container.

### 3.8 Resources management

In this section we will describe the resources management system that has been chosen in this project, some are only described and would need to be implemented for a production-ready framework. Resources management is
3. HAALO: AN HARDWARE ACCELERATOR ABSTRACTION WITH LOW OVERHEAD

![Flowchart](image)

Figure 17: Watch of shared memory blocks

a key point in such an infrastructure because Kubernetes is not aware of these resources and there is no built-in way of controlling these.

3.8.1 Memory management

Containers running in the host IPC namespace introduces a problem: containers can allocate memory outside of their namespace and thus can allocate memory that is not limited by the docker engine. To prevent containers to allocate more memory than they should, we have to build a system to limit the amount of memory allocated by containers (see figure 17 for its event flow). As we have seen in the Security section, each container runs with its own UID. Thus, the owner of each block of memory assigned in the shared memory can be identified thanks to its owner field. To limit the amount of memory allocated in the shared memory by each container, a daemon needs to be implemented to watch the /dev/shm directory. This daemon would use inotify to watch file creation and modification and checks that the maximum amount of shared memory allocated by the containers is not over the limit. If a container allocates more memory than it should, the container is killed and the memory blocks are removed from the shared memory directory.
3.8.2 Accelerator management

Accelerators are allocated at runtime when the container first instantiate the connection with the host. When the container instantiates the connection, the host will access the database to retrieve information about the container and will allocate the devices accordingly. Note that the container is not aware of what kind of accelerator is allocated to it. Once the information has been retrieved by the host, the cluster metadata is updated so that the scheduler and other entities that might need it can know what resources are now available or in use on the host. The host also keeps an internal record of the container’s resource utilization so that when a container is killed, it can know which resources are to be reported as available into the database.

3.8.3 Classic resources management

While some resources have to be managed on the hosts because of the lack of control docker gives when it comes to extended resources (accelerators or shared memory for example). Other resources can, and should, be managed by Docker or Kubernetes. Resources like CPU and memory (non-shared memory), are left for Docker and Kubernetes to manage. This simplifies the implementation of the daemon host but also let the version of Kubernetes and Docker as close as possible to the original upstream versions.

3.9 OpenCL Kernels repository

To provide kernels for different platforms, we built an OpenCL kernels repository. This repository, combined with a database, holds precompiled kernels and OpenCL sources. The database holds extra information such as the preferred device for the container and what devices the container can make use of. A central repository reduces greatly the time spent on the compilation, as every kernel is compiled at most once for every device, which reduces the overall resources and footprint of compilation cluster-
3. HAALO: AN HARDWARE ACCELERATOR ABSTRACTION WITH LOW OVERHEAD

wide.

3.9.1 Repository structure

The repository structure is a nested structure. Each container has its own entry and directory. Because of this fact, when two containers request the same kernel (e.g., a vecadd kernel), the kernel loaded on the device will not necessarily be the same, depending on the container needs. The second level will contain the different kernel names and finally, the third level will contain the source of the kernel (a .cl file), and different binaries for each device the kernel can run on. When retrieving a kernel the container host will retrieve all the required files to provide an execution runtime for the container. The repository layout is also described in figure 18.

Figure 18: Repository layout
3.9.2 Database content

To hold more detailed information about the containers and devices relations, we have set up a database. The database and the corresponding rest API should be accessible with a fixed IP address across the cluster for ease of use and deployment. The database contains information about all containers that require hardware accelerators potentially scheduled in the cluster. It holds information on containers’ preferred devices, devices that the container can run on, but also information about the kernels corresponding to the containers, (e.g. if the kernel is already compiled for a specific device or not). The database also contains information about the nodes and their available devices, to provide insights on device utilization to the Kubernetes scheduler extender that we will describe in the following section. The database will be checked before and at the moment a container is scheduled, either to retrieve information about the cluster or retrieve the kernels, either to update available resources.

3.9.3 Kernel serving api

To serve the different precompiled or source kernels an API has been implemented to allow an easy retrieval of these objects. The API serves the following paths (parameters between ‘{”}’ are dynamic):

- GET /prefs/{name}: this path serves a JSON file, which contains the preferred type of device for the container, the devices it can run on.

- GET /specs/{name}/{device}: this path serves the specifications of the kernel for the specified device. It contains information about the compilation status of the kernel, the name and the URL we can get it from.

- GET /kernel/{containername}/{kernelname}/{device}: this path is used to fetch a precompiled kernel for the specified device and the
specified kernel. A container may have multiple precompiled kernel for each kernel entry.

- POST /kernelUpload/{containername}/{device}: compilation may be done at runtime, so the daemon should be able to access this path. This path is used to upload a precompiled kernel for a specific device. The container and kernel should already be present in the database for the request to be accepted.

- POST /kernelSourceUpload/{containername}/{kernelname}: if the source is needed, we can also provide it to the system, in which case the daemon could compile the kernel at run time. This could be done for devices such as GPU or CPU when compiled is fairly quick. For devices such as FPGAs, this is not recommended. If the source shall stay confidential, only precompiled binaries may be uploaded.

3.10 Kubernetes extensions

This section describes different ways to extend Kubernetes and then explain the choice of the way to extend Kubernetes for this project. Many ways are available directly from Kubernetes without needing to modify the main source code. Kubernetes is very modular and thus modules can just be "plugged” into designed interfaces.

3.10.1 Scheduling extensions

Kubernetes provides many ways to extend the behavior of a cluster. This section will review the ones that are available to extend the scheduling decisions\cite{41} and explain the choice that has been made.

- **Extended resources**: Kubernetes has the ability of advertising resources that are not present by default in its implementation. These resources are called extended resources. Extended resources are user-defined resources, and are totally opaque to Kubernetes. Kubernetes
will not trigger any actions when scheduling containers or pods requiring these kinds of resources but it is guaranteed that no more than the advertised number of extended resources will be scheduled on a node. These resources are useful when pods access external resources like devices, or special storage, or anything that is not defined by Kubernetes. In the case of our implementation, we are using two kinds of extended resources: shared memory and accelerator. Shared memory is the amount of shared memory that the containers will be allowed to use in the host’s IPC namespace. Shared memory will be handled by Kubernetes only when it comes to scheduling, as we do not need any fine-grained control over it, we just need the amount of shared memory potentially requested by containers not to exceed the amount of shared memory available on the host. An "accelerator" resource has also been created. This resource is an abstract resource that we need to check before scheduling containers on nodes. This resource being abstract means that it represents multiple kinds of accelerators at the same time and that we will take an extra step to remove accelerators that do not fit the containers’ requirements right before they are scheduled.

- **Device plugin**: To advertise the hardware accelerators to the Kubernetes cluster, a device plug-in[42] can be implemented. Device plugins enable device advertisement to the cluster without modifying Kubernetes core code. This is especially useful when the device being advertised corresponds to an extended resource. Device plug-ins have been implemented to enable hardware accelerators vendors to include device specific initialization and monitoring to Kubernetes.

  The device plug-in uses a gRPC[43] service present in the Kubelet module of Kubernetes to register itself and implements its own gRPC calls to list and watch the available resources (listAndWatch call), and to allocate the resources to the container.

  Every time a new device is available or taken, the device plugin notifies the Kubelet agent, and then the updated value of the extended resource will be advertised to the cluster.
3. **HAALO: AN HARDWARE ACCELERATOR ABSTRACTION WITH LOW OVERHEAD**

- **Scheduler extender**: To take care of the abstract ”accelerator” resource, a scheduler extender in the form of a webhook can be written. The purpose of this scheduler extender is to select a node from the list of nodes that have been pre-selected by the default Kubernetes scheduler. Because Kubernetes does not know about accelerators and the preferences of containers towards some accelerators, or their capabilities to run on certain types of hardware. A scheduler extender would be a helper which helps to manage resources out of Kubernetes’ scope. A scheduler extender can do two actions: first it can filter and then it can prioritize as we can see in figure 19 (filter and prioritize are optional, meaning that there could be only filter or only prioritize calls). The filter removes nodes that cannot run the specified pod, and prioritize returns a score for each node. A higher score means that the node should be prioritized.

- **Labels**: In a Kubernetes cluster, nodes can have labels which are made of a key and a value. Scheduling decisions can be made on these labels, for example when Kubernetes is scheduling a pod on a node, if specified by a pod definition, Kubernetes can restrict the set of nodes that are available to only the ones that have the said label set to the defined value. Kubernetes can take different decisions based on the options in the pod definition. It can either restrict the schedulability with the ”requiredDuringSchedulingIgnoredDuringExecution” field, which will filter out every node which does not satisfy the conditions that will be described later. It can also prioritize some nodes based on conditions with the ”preferredDuringSchedulingIgnoredDuringExecution”. The options that are taken into account are the following:

  - Gt: shorthand for ”greater than”. Is true only if the label value is strictly greater than the value in the pod definition
  - Lt: shorthand for ”less than”. Is true only if the label value is strictly lesser than the value in the pod definition
  - In: True if the value of the label is present in the set of values in the pod definition.
3. HAALO: AN HARDWARE ACCELERATOR ABSTRACTION WITH LOW OVERHEAD

Figure 19: Scheduler execution flow with a webhook extender

- NotIn: True if not present in the set
- Exist: True if a label with the specified name is set on the node
- NotExist: True if label not present

These conditions only affect the scheduling decision, if the labels on a node are changed during a pod execution, no action will be taken. It can thus be used to extend or modify Kubernetes scheduling behavior by modifying the labels present on nodes at run-time. Figure[20] shows the removing of a node from the possible selection because of its label.

To modify labels at run time there are multiple solutions provided by
3. HAALO: AN HARDWARE ACCELERATOR ABSTRACTION WITH LOW OVERHEAD

Figure 20: Node selection in function of labels [44]

Kubernetes

- Kubectl command line tool: Kubectl is a command line tool that is used by cluster administrators to deploy and apply configurations to a Kubernetes cluster. It can set or remove labels on nodes present in the cluster via the command "kubectl label nodes {node name} {label-key}={label-value}".

- Rest API: Kubernetes also provides a raw REST API which can be accessed by simple HTTP requests (with utilities such as curl or wget for example).

- Language specific client libraries: Kubernetes also provides different client libraries to interact with the cluster. These libraries are wrappers around the REST API but highly simplify interactions with the cluster and provide language-native object representation. Some of the supported languages are Python, Go, Perl and Java.
3.10.2 Choice of extension

The extension process that has been chosen is the dynamic labeling of nodes because it is the most adapted to the abstraction being built. Device plugins were created to expose the device directly to the container, which is not done in this project so it doesn’t make sense to use it here. Scheduler extender and dynamic labeling are achieving the same task, which is filter and prioritize nodes to execute pods. However, the scheduler extender requires to modify options for the Kubernetes scheduler and also requires a database to keep track of device allocation and available devices. On the other hand, the dynamic labeling of nodes does not require that because labels are stored in a distributed key-value store which means that it will directly be available in the cluster. Moreover, it lets more freedom when it comes to scheduling management via the deployments configuration files without having to define these ways into a new service.

3.10.3 Dynamic labeling

As the number of available devices is continuously changing during the lifetime of the cluster, there is a need to update labels when this number changes. To keep the number of nodes up to date, a watcher was implemented to keep track of which containers are running on each node. The watcher will target the Kubernetes API and check for labels corresponding to containers that need hardware accelerators. For each node, the watcher will count the number of hardware accelerator units and update the node label accordingly. If a pod is scheduled before the node label is updated the watcher will delete pods until the label becomes positive again.

There is no need to specify the node as the labeler is deployed in the same pod than the daemon through a daemon set (see figure 22), and thus can retrieve the node’s id easily. In the figure, ”Daemon” and ”Watcher” are containers. When the label for a specific device is updated, the label corresponding to the number of accelerators (generic term) is also updated to reflect the total number of accelerators. To limit to the maximum the
footprint of the pods, we can limit the amount of CPU used by the watcher, or wait a specified amount of time before rechecking the pods that are running.

### 3.10.4 Scheduling decisions

Scheduling decisions are let to Kubernetes. Indications are however given to it via the configuration files (either deployment, service, or pod configuration file). Configuration files have to include node affinity fields for labels corresponding to devices that containers can run on. For each device that the entity can utilize, a corresponding node affinity is defined, including prioritization and capacity to run as some containers might be able to run on certain devices but would prefer another type of device. Multiple device count can be assigned to each pod for different devices, which allows having device allocation based on performances of said devices.

When support for a new device is added, configuration files have to be updated, to include the new device. This could be automated when the kernel is updated to the kernels repository. As there is a need to specify the
device count that is required by the pod, some performance tests need to be done to provide an accurate count and thus have constant performances across the different nodes and devices.

Scheduling decisions could also be made in a “soft manner” (by only using the option “preferredDuringSchedulingIgnoredDuringExecution” and not the “requiredDuringSchedulingIgnoredDuringExecution”) provided that all containers can run exclusively on CPU, in which case it accelerators would be preferred if available, but containers could potentially be scheduled on CPU only nodes.

Figure 23 shows a pod configuration file so it is only scheduled on nodes with at least two GPUs or one FPGA available. If none are available, this pod will stay in a pending state until a device is available.

### 3.11 Summary

In this section we described the overall architecture and the explained the choices that were made to make the system as efficient as possible while
3. **HAALO: AN HARDWARE ACCELERATOR ABSTRACTION WITH LOW OVERHEAD**

```yaml
apiVersion: v1
class: Pod
metadata:
  name: opencl-client1
labels:
  acceleratorneeded: "1"
spec:
  affinity:
    nodeAffinity:
      requiredDuringSchedulingIgnoredDuringExecution:
        nodeSelectorTerms:
          - matchExpressions:
            - key: accelavailable
              operator: GT
              values:
              - "0"
hostIPC: true
containers:
  - name: client
    image: client_container
    volumeMounts:
    - mountPath: /socket
      name: socket
      command: ["/client"]
volumes:
  - name: socket
    hostPath:
      path: /socketDir/socket
      type: Socket
```

**Figure 23:** Configuration file for a pods requiring accelerators
3. HAALO: AN HARDWARE ACCELERATOR ABSTRACTION
WITH LOW OVERHEAD

having a clean architecture that leverages all of the advantages of docker
and Kubernetes and put them to the service of high performance computing
and networking. In the next section we will analyse the performance and
check if the architecture that was used and implemented actually solves
the problem that we aimed to fix.
4 Results and discussion

This section provides insights into the performance and capabilities of the framework that has been developed during the master’s thesis. Tests were run on a server with a NUMA architecture (two CPUs) and an Nvidia GPU (Tesla P100). The tests that were run aimed to provide a comparison between running plain OpenCL accelerated programs running bare metal and running container packaged OpenCL accelerated applications. The project aims to have performances as close as possible to bare metal performances to enable the use of such architecture in high performances environments. This project was first aimed at environments where latency is critical, tests that were run will thus compare end to end latency. The tests that were run do not aim to get results about the graphics card that was used during it but were run to provide a comparison between a bare metal execution and an execution using the framework. The process niceness (process priority, -20 being the highest priority and 19 being the lowest, 0 is the default) that have been used for this benchmark are the following:

- Daemon process niceness : -20
- Container process niceness : 0
- Bare metal process niceness : 0

Bare metal execution and the client container run with a niceness of 0 because it is very unlikely that they run with a lower priority in a deployment, while the daemon process has a niceness of -20 because each server where the daemon will have cores dedicated to that process.

4.1 Framework capabilities

The framework that has been implemented during the course of this master’s thesis provides a hardware agnostic run-time for OpenCL to Docker container and an integration to Kubernetes. The framework allows the
same container to use any OpenCL hardware accelerator without any required libraries nor knowledge of the actual accelerator that the container is relying on. Whether the container is running on a GPU, CPU, or FPGA the results will be the same and the code or the libraries inside the container do not need to be modified to take advantage of different hardware accelerators.

A side effect of this is that the hardware libraries are being moved to and provided by the infrastructure. This can increase the overall security as library updates can be deployed in the whole infrastructure without any container having to be modified. This increases the reactivity when a security update needs to be deployed for example.

4.2 Performances

To measure the performances of the system, we calculated the overhead introduced by the framework compared to bare metal execution. To calculate it, we ran an OpenCL kernel directly on the host without any kind of virtualization (it was not running inside docker) and compared the time needed to get the results back from the GPUs. Compilation time was excluded as the kernels would typically be precompiled in a real deployment. The timestamps are placed right before the event is placed in the OpenCL queue and right after the result is mapped back into the local memory. For the framework, the timestamps were not placed at the same place as what is interesting to the end user is the latency from the moment the code inside the container calls the framework and the moment the memory is filled with the results from the OpenCL kernel.

In this benchmark we will study the effects on performances of different parameters which are the following:

- Size of data batch
- Computation load
- Buffer reuse vs buffer creation
4. RESULTS AND DISCUSSION

Data size and computation loads were chosen to represent a different kind of applications which are the following:

- Packet inspection (from 100kB and 4MB (lower and upper limit))
- Hardware accelerated application (from 4MB and 40MB)
- Deep learning model (from 40MB and 4GB)

The different loads (1 instruction, 1000 instructions) were applied to each data set. Data sizes and computation loads were chosen based on Ericsson’s existing hardware accelerated solutions.

4.2.1 Performance comparison

In this section we will compare the end-to-end latency from the moment the instruction to use the hardware accelerator to the moment the result is put in the memory. The test program reuses the same shared memory buffers, because buffer allocation is a system call and would introduce extra non controllable delay. Figures 24, 25, 26, 27, 28, 28, 31, 32, 33, 34 show the comparison between running a bare metal application and an application running HAALO.

The results show that the total execution time when using HAALO is consistently higher than when running bare metal. However, the overhead introduced is fairly constant and stable except for the first iteration. This is due to the fact that the first iteration needs to allocate and create shared memory buffers, which is time-consuming. Once these buffers are allocated, the extra overhead disappears as we are reusing the buffers. We can also see that the standard deviation is both low and stable, which shows that the framework does not introduce a high level of uncertainty.
4. RESULTS AND DISCUSSION

4.2.2 Overhead vs data size

In this benchmark, we will look into the overhead introduced by the framework for different sizes of data. The kernel that will be run on each data size is the same. Results are shown in Figure 25. We can see that the overhead is dependant on the buffer size, but not of the computational load. The overhead can be expressed as a linear function of the buffer size. This is the
4. RESULTS AND DISCUSSION

Figure 26: Total execution time in function of iteration number for 4MB and workload of 1

![Graph showing execution time](image)

Figure 27: Total execution time in function of iteration number for 4MB and workload of 1000

![Graph showing execution time](image)

drawback of using shared memory as there needs to be some bookkeeping for the said memory to make it available to multiple processes at the same time. Given that fact, this kind of result was expected.
4. RESULTS AND DISCUSSION

Figure 28: Total execution time in function of iteration number for 40MB and workload of 1

Figure 29: Total execution time in function of iteration number for 40MB and workload of 1000

4.2.3 Overhead vs computation load

In this benchmark, we will take a look at the overhead introduced by the framework depending on the workload present in the OpenCL kernel. For each data load, the kernel will be modified to increase the load on the GPU. As we have seen before the absolute overhead does not depend on the computation load, however, if we look at the relative overhead (shown in
4. RESULTS AND DISCUSSION

Figure 30: Total execution time in function of iteration number for 40MB and workload of 1000

Figure 31: Total execution time in function of iteration number for 400MB and workload of 1

Fig 37 we can see that the relative overhead decreases we the computation load increases. The relative overhead then stabilizes around 2.5% when the workload is 1000 and 9% when the workload is 1. This difference is partly due to the fact that shared memory requires extra work for the operating system, and thus provide slightly less performance than classic memory.
4. RESULTS AND DISCUSSION

Figure 32: Total execution time in function of iteration number for 400MB and workload of 1000

Figure 33: Total execution time in function of iteration number for 4GB and workload of 1

4.2.4 Buffer reuse vs buffer creation

In this benchmark, we will take a look at the effect of reallocating buffer for every computation compared to reusing the same set of buffers over and over. Results are shown in figure 36. From the results, we can easily see that the buffer creation is time-consuming and that it increases the total execution time. For that reason, buffer reuse is the way to go to get the
4. RESULTS AND DISCUSSION

Figure 34: Total execution time in function of iteration number for 4GB and workload of 1000

Figure 35: Overhead (microseconds) in function of buffer size

best performances out of this framework.

4.3 Conclusions

The tests that were run show that the framework introduces some overhead compared to an execution directly on the host. However, the overhead is
4. RESULTS AND DISCUSSION

Figure 36: Absolute overhead in function of size and buffer creation or reuse

fairly small and represents only a small percentage of the total execution time. The overhead is mainly dependant on the size of the buffers sent to the GPU, however as the total time of execution gets longer as the data size increases, the extra time introduced by the framework stays to a relatively small percentage of the total execution time. We could say that the framework provides an efficient way to access the hardware accelerators because the performances are close or meet the requirements that we defined earlier in that thesis. Also, it allows using hardware accelerators in a generic way without harming the performances too much. With some more engineering, we could aim to reduce this overhead even more as the program is not yet totally optimized (some engineering could be done on the retrieval of binaries for example). These optimizations would however stay limited as the overhead also depends on the amount of memory being used and would not affect the linearly increasing overhead. Even though it was not possible on the server used to do the tests, it would also be great to see how the performances change when the framework is pinned to specific cores that are in the same physical CPU and on dedicated cores where no other process could be scheduled (the server used here had multiple physical CPUs and thus we could not control where threads were scheduled,
also we were not able to modify the kernel options to prevent scheduling on specific cores).

4.4 Recommendations

To get the best performances from the system, recommendations are the following:

- Reuse buffers, do not allocate a buffer for each operation but rather pre-allocate buffer reuse them as long as possible
- Prefer one large workload rather than multiple smaller workloads

These recommendations will allow to get the best performance but are not required to use the framework. These recommendations are merely a guideline to limit the overhead introduced by the framework.
4. RESULTS AND DISCUSSION

4.5 Summary

As we have shown with the performances measured above, the overhead introduced by the framework is very limited. This overhead being that low makes the advantages that are introduced with that framework even more impactful. Being able to use different kinds of accelerators in an heterogeneous cloud context is extremely useful and allows to both have a better use of the platform available to the system that is deployed but it also simplify the scheduling and development of services.
5 Conclusion and future work

This thesis aimed to develop and integrate hardware acceleration into a Kubernetes cluster running Docker, where the cluster runs different kinds and brands of hardware accelerators. The thesis presents a framework to include heterogeneous hardware accelerators in the widely used Kubernetes orchestrator. This chapter aims to present the thesis’ contribution, its limits and future work.

5.1 Contribution

This framework provides a way to abstract accelerators and provides the possibility for Docker containers to access hardware accelerators without any need to have the knowledge or exclusive access to the said accelerator. This framework also extends Kubernetes’ scheduling by adding dynamic labels to influence scheduling decisions made by Kubernetes.

To provide unified access to hardware accelerators, a unified interface to communicate between the host and the container has been implemented. This is the most important piece of this framework because it is this interface that abstracts the hardware accelerator to the container and thus enables one container to use different kinds and brands of hardware accelerators without any device awareness.

One of the benefits of this architecture is that it allows a high decoupling of the different modules in hardware accelerated application environment. Programming the hardware accelerators and the server side programs are now totally independent and thus allow a better development process.

The concept of abstraction is of first class in that framework, not only it allows to have a coherent integration with Docker and Kubernetes which heavily use abstraction, instead of providing direct access to the accelerators as it is done as of right now. With that framework hardware acceleration has become an abstracted concept instead of staying very close to the actual physical hardware.
Moreover, this framework brings OpenCL as a service, going one step further than the current state of the art hardware accelerator as a service provided by cloud platforms such as AWS, Azure or GCP. What these platforms provide still requires the users to have knowledge about the hardware accelerators and have to use specific images and compilers to make use of these.

5.2 Limitations

While this framework provides new ways of integrating hardware accelerators into Kubernetes cluster with ease and enables hardware agnostic use of hardware accelerator there are still some limitations. First of all, this framework requires to have 2 physical cores dedicated to it which will not be usable by other applications. Secondly, as of now, only OpenCL hardware accelerators are supported, which limits the possible use cases of this framework. Moreover, while the overhead that is introduced because of this framework is fairly low, ultra-low latency requirements might not be met without a rework, or with an addition of new functionalities which would reduce the amount of communications between the client containers and the daemon.

5.3 Future work

To make this framework a truly production ready framework, multiple things are to be considered. First, adding support for non-OpenCL devices would be a great addition to provide access to FPGA enhanced NICs for example. Following this point, adding support for streaming and per-packet devices would greatly enhance the usage of the framework for virtualized network functions. Moreover, as of now, the framework provides only a few configuration options for the accelerators, adding a more fine grain control over the device configurations would enable developers to get the best performances out of this framework. Also, as of right now, there is no control
over the time spent holding the accelerator for each container, adding this would be of great benefit to enable the deployment of a managed OpenCL as a service platform by cloud platforms. Finally, while not possible with the server I had access to during the benchmark, it would be of great interest to see the effect of a real CPU isolation (remove cores from scheduling and manually pin the polling and the notifier to these cores) instead of the high priority that was used during the tests.
References


REFERENCES


REFERENCES


[26] Xilinx, FPGA as a service, <https://github.com/Xilinx/FPGA-as-a-Service/tree/master/k8s-fpga-device-plugin/trunk> [Online; accessed 02-02-19],


[29] Docker storage documentation, <https://docs.docker.com/storage/> [Online; accessed 22-01-19],


REFERENCES


[43] google, grpc, https://grpc.io/, [Online; accessed 08-03-19],