POLITECNICO DI MILANO
Corso di Laurea Magistrale in Ingegneria Informatica
Dipartimento di Elettronica, Informazione e Bioingegneria
Scuola di Ingegneria Industriale e dell'Informazione

POLITECNICO
MILANO 1863

BlastFunction: an FPGA-as-a-Service system for accelerated serverless computing

Advisor: Prof. Marco Domenico Santambrogio
Co-Advisor: Dott. Ing. Rolando Brondolin

Master of Science thesis of:
Marco Bacis
Matr. 873199

Academic Year 2018-2019

Alla mia famiglia e a Beatrice

Marco

Ringraziamenti

Questa tesi é il culmine di sei anni di impegno, fatica, ma anche vittorie e soddisfazioni. Durante questo percorso ho incontrato persone che mi hanno aiutato nell'università e al di fuori di essa, e senza le quali questo lavoro non esisterebbe.

Un primo grazie va il mio relatore Marco Santambrogio per avermi accolto al *NECSTLab* e avermi guidato in un lungo percorso che é andato ben oltre questo lavoro di tesi. Tutte le opportunità e le sfide che mi hai offerto negli ultimi quattro anni mi hanno fatto crescere e maturare, non solo nell'ambito accademico.

Un altro ringraziamento va a Rolando, colui che che ha seguito il mio lavoro, i miei dubbi e le mie ansie durante tutto questo anno passato a supervisionarmi. I tuoi suggerimenti, idee e "spinte" sono stati fondamentali durante questo anno, e mi hanno portato fin qui.

Grazie a tutti i ragazzi e ragazze del NECST, passati e presenti. Ognuno di voi mi ha mostrato che é con la grinta e la voglia di fare che si va avanti (senza contare il calcetto!). Grazie a voi il laboratorio é stato come casa per gran parte del mio percorso, e di questo vi sono grato.

Ringrazio i miei compagni di *dis*avventure al poli; chi ce l'ha fatta e chi ce la sta facendo, e chi ha cambiato strada: Antonio, Daniele, Maddalena, Federico, Carlo, Jacopo e tanti altri. Senza di voi questa avventura non sarebbe stata la stessa, tra lezioni, sushi e ore e ore di divertimento per fuggire alla noia e alla difficoltà degli studi.

Grazie inoltre a Monica, Francesca e Giulia, con cui ho condiviso difficoltà e felicità anche se in università diverse e con cui ho passato tanti momenti importanti assieme. Siete le migliori amiche che si possa avere.

Grazie ai miei genitori Bramina e Lorenzo, perché mi avete supportato (e sopportato) da sempre. Grazie per i vostri insegnamenti, consigli, conforto, per avermi capito nei miei momenti peggiori e per aver festeggiato con me in quelli migliori. Grazie a mia sorella Jessica, perché mi continua a mostrare che la tenacia e la testardaggine ripagano e che c'é sempre qualcosa di nuovo da fare nella vita.

Infine grazie alla mia fidanzata, Beatrice. Grazie per tutto l'affetto, la comprensione, il supporto, la pazienza, l'incoraggiamento, e per tutti i momenti passati assieme a te. Senza di te questo lavoro non sarebbe stato possibile.

Marco

Contents

List of Figures

List of Tables

Acronyms

AFI Amazon FPGA Image. 17, 18

ALU Arithmetic-Logic Unit. 16

API Application Programing Interface. 10, 11, 48

ASIC Application Specific Integrated Circuit. 17

AWS Amazon Web Services. 17, 18

BRAM Block RAM. 15, 16, 19, 54

CLB Configurable Logic Block. 14, 15, 16

CNN Convolutional Neural Network. 55, 74

CU Compute Unit. 18, 56

DFE Dataflow Engine. 26, 27

DSE Design Space Exploration. 54, 55

DSP Digital Signal Processor. 15, 16

FaaM FPGA-as-a-Microservice. 28

FPGA Field Programmable Gate Array. IX, X, XI, XII, 1, 2, 3, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 33, 34, 39, 42, 43, 48, 52, 53, 54, 55, 56, 58, 63, 69, 75, 76

HDL Hardware Description Language. 16

HLS High Level Synthesis. 16, 17

HTTP Hypertext Transfer Protocol. 51, 69

IaaS Infrastructure as a Service. 26

IC Integrated Circuit. 14

ICAP Internal Configuration Access Port. 16

IOB Input-Output Block. 14, 15

LRN Linear Response Normalization. 56

LUT Look-up Table. 14

MAC Multiply-Accumulate. 56

MM Matrix Multiply. IV, 59, 61, 62, 63, 74

OS Operating System. 48

PDR Partial Dynamic Reconfiguration. 23

PE Processing Element. 18

PLA Programmable Logic Array. 14

PLD Programmable Logic Device. 14

PR Partial Reconfiguration. 24

QoS Quality of Service. 26

RPC Remote Procedure Call. 43, 44, 46

RTL Register Transfer Level. 16, 17

RTT Round-Trip Time. 59

SaaS Software as a Service. 26

SJF Shortest Job First. 27, 28

SLA Service Level Agreement. IX, X, 1, 38

SLO Service Level Objective. 26

SM Switch Matrix. 14, 16

VFR Virtual FPGA Resource. 25

VM Virtual Machine. X, 1, 5, 7, 8, 9, 11, 17, 18, 20, 22, 23, 24, 25, 26, 27, 30

VMM Virtual Machine Monitor. 5, 6, 7

Abstract

The last decade saw the exponential growth of cloud computing as the primary technology to develop, deploy and maintain complex infrastructures and services at scale. Cloud computing allows to consume resources on-demand and designing web services following a cloud-native approach is fundamental to dynamically scale performance. However, some workloads require computing power that current CPUs are not able to provide and, for this reason, heterogeneous computing is becoming an interesting solution to continue to meet Service Level Agreement (SLA) in the cloud.

Field Programmable Gate Arrays (FPGAs) represent one of the possible ways to employ heterogeneous computing in cloud scenarios. Given that requests to services can come at unpredictable rates, the underlying FPGA may not be utilized for 100% of the time. From a cloud provider perspective, sharing would allow to improve time utilization of the FPGA, and the serverless computing paradigm represents a promising approach in this sense, as resources management is delegated to the cloud provider and each functionality is scaled depending on the exact need of the moment. Within this context, we propose that compute-intensive kernels should be accelerated with shared FPGAs handled transparently by the serverless infrastructure: this will maximize utilization while reaching near-native execution latency.

In this thesis work we propose BlastFunction, a distributed FPGA sharing system for the acceleration of microservices and serverless applications in cloud environments. BlastFunction provides a *transparent* and *scalable* system enabling *multi-tenancy* in the cloud FPGA scenario, with a *vendor-independent* and *reconfiguration-aware* allocation strategy integrated with an existing cloud orchestrator. The system includes a *Remote OpenCL Library* to access the shared devices transparently and with a known interface; multiple *Device Managers* which offer the underlying devices using a *time-sharing* approach and expose relevant metrics; a central *Accelerators Registry* which tackles the goal of allocating the available devices efficiently using runtime performance metrics, interacting with the Kubernetes orchestrator.

We evaluated the system with three experiments to observe first the introduced overhead, then the behaviour in a small cluster with a single scaled function and multiple functions. In all the experiments, BlastFunction was able to reach higher utilization and throughput thanks to the sharing of the device, with minimal differences in latency and requests drop given by the concurrent accesses and the additional I/O latencies.

Sommario

Gli ultimi anni hanno visto la crescita esponenziale del cloud computing come tecnologia primaria per lo sviluppo, la distribuzione e il mantenimento di infrastrutture complesse e servizi a larga scala. Il Cloud Computing permette il consumo di risorse on-demand, e il design di servizi web seguendo un approccio cloud-native é fondamentale per scalare dinamicamente le performance. Inoltre, i cloud provider offrono un accesso multi-tenant a un gruppo di risorse scalabili, che permettono ai servizi di crescere indefinitamente seguendo performance e costi. Per raggiungere questi obiettivi i servizi cloud si sono evoluti assieme ai corrispettivi sistemi di gestione. La tecnologia attuale comprende diverse tecnologie di virtualizzazione, come Virtual Machines (VMs) e Containers.

Purtroppo, alcuni workload richiedono una potenza di calcolo a cui le CPU non sono in grado di provvedere e, per questo motivo, l'*heterogeneous computing* sta diventando una possibile soluzione per soddisfare i SLAs. Il primo tentativo per risolvere questo problema é stato l'introduzione delle GPU nei datacenter e nelle istanze cloud per accelerare workload compute-intensive (come il training e l'inferenza nel Machine Learning e i sistemi di elaborazione video streaming). Le GPU offrono alte performance per workload e algoritmi che sono compatibili col loro pattern architetturale parallelo. Il problema é che le GPU eccellono nel processamento a batch, nel quale gruppi di record vengono processati da un acceleratore si ad alto throughput, ma con un'alta latenza nel caso di un singolo elemento [1]. Mentre molte applicazioni e framework cloud lavorano con una modalità batch (es. MapReduce o training nel Machine Learning), molte applicazioni cloud-native funzionano come *servizi*, nei quali la *latenza* é il fattore maggiormente considerato, poiché ogni richiesta inviata dall'utente dovrebbe essere processata nel minor tempo possibile. Inoltre, le GPU introducono dei problemi di efficenza energetica (poiché consumano molto più delle CPU), che nel contesto di un datacenter risulta essere un fattore di costo significativo.

In aggiunta alle istanze standard e equipaggiate con GPU, ultimamente sono state introdotte istanze basate su FPGA. Le FPGA offrono alte performance e un consumo ridotto di energia, grazie alla loro flessibilità a livello di architettura hardware. Infatti, esse possono essere configurate per implementare qualsiasi algoritmo in modo ottimale, poiché si può decidere a grana fine il livello di parallelismo di ogni componente dell'architettura risultante in base alla elaborazione che bisogna effettuare. Inoltre, la loro *riconfigurabilità* significa che il circuito fisico non cambia dopo ogni aggiornamento dell'implementazione (come negli ASIC, i quali devono passare da fasi di design e produzione per ogni aggiornamento dell'algoritmo accelerato). Questa funzionalità si adatta bene alle applicazioni

cloud-native, poiché sono continuamente sviluppate, aggiornate e distribuite nel loro ciclo di vita. Per questo, le FPGA hanno trovato uso in molteplici applicazioni. Contesti come la ricerca web [2], elaborazione di immagini [3], compressione [4], operazioni su database [5], inferenza di reti neurali [6] e molti altri possono trarre beneficio dall'uso di architetture e acceleratori specializzati come le FPGA per rispondere tempestivamente alle richieste degli utenti. Inoltre, l'introduzione delle istanze AWS F1 [7], oltre che ai progetti Catapult [8] e Brainwave [9] da parte di Microsoft dimostrano che questa tecnologia giocherà un ruolo chiave nei prossimi anni.

Per sfruttare al meglio le FPGA nel cloud, gli acceleratori hardware dovrebbero essere progettati per soddisfare i requisiti di latenza e ottimizzare il throughput [8]. Le richieste dalla rete esterna possono arrivare con una frequenza imprevedibile a di solito non possono essere messe in batch, quindi minimizzare la latenza risulta fondamentale. L'imprevedibilità delle richieste può portare a una sottoutilizzazione delle FPGA, quindi riservarne una per ogni servizio che ne richiede può risultare in uno spreco di risorse. Dalla prospettiva del cloud provider, la condivisione delle risorse permettebbe di migliorare l'utilizzo delle FPGA e in questo senso il paradigma del serverless computing é un approccio promettente [10]. Il Serverless computing é un pattern architetturale per applicazioni cloud nel quale la gestione dei server é delegata al cloud provider. Ogni funzionalità dell'applicazione é distribuita dall'utente come *funzione* e schedulata, eseguita, scalata e fatturata in base ai *on-demand*. In questo contesto, proponiamo che i kernel compute-intensive dovrebbero essere accelerati tramite FPGA condivise e gestite in modo trasparente dall'infrastruttura serverless: questo massimizzerà l'utilizzo pur raggiungendo una latenza vicina all'esecuzione nativa.

Allo stato attuale, un sistema completo per la condivisione e allocazione di FPGA in ambiente cloud (comprendente microservizi e piattaforme serverless) non é ancora stato realizzato. Inoltre, la maggior parte dei framework esistenti non sono *trasparenti* allo sviluppatore, e non sono integrati con orchestratori in commercio (come Kubernetes).

In questo lavoro di tesi proponiamo BlastFunction, un sistema distribuito per la condivisione di FPGA che permette di accelerare microservizi e applicazioni serverless in ambienti cloud. L'obiettivo del lavoro proposto é di offrire un sistema trasparente e scalabile per abilitare la *multi-tenancy* in uno scenario di FPGA cloud. Inoltre, offriamo una strategia di allocazione *vendor-independent* e *reconfiguration-aware* integrata con un orchestratore in commercio. Abbiamo deciso di focalizzarci su un approccio *time-sharing* per il nostro sistema, in modo da massimizzare l'utilizzo del dispositivo (a livello di tempo d'esecuzione dell'acceleratore) e ottimizzare l'uso dei dispositivi dalla prospettiva del cloud provider.

BlastFunction comprende tre componenti principali: una *Remote OpenCL Library*, multipli *Device Managers* e un *Accelerators Registry* centrale. La Remote OpenCL Library é un componente che permette alle applicazioni cloud (o funzioni serverless) di accedere alla FPGA condivisa nel cluster. Si tratta di un'implementazione customizzata dello standard OpenCL che astrae l'uso della connessione verso il dispositivo remoto e gli

altri componenti del sistema rispetto al codice host. Il Device Manager é un'applicativo server presente su ogni nodo del sistema e connesso all'FPGA sottostante. E' il componente che offre il meccanismo di *time-sharing*, esponendo un servizio tramite il quale funzioni e applicazioni possono accedere al dispositivo concorrentemente. Inoltre, ogni Device Manager espone metriche riguardanti il comportamento a runtime del dispositivo, consentendo agli altri componenti di agire di conseguenza. Infine, l'Accelerators Registry é il controllore centrale del sistema, che fa fronte all'obiettivo di allocare i dispositivi disponibili efficacemente usando le metriche raccolte a runtime. In particolare, il Registry traccia le metriche di utilizzo dei dispositivi e esegue un algoritmo di allocazione *online* tenendo conto della riconfigurazione, integrandosi con l'orchestratore per la modifica di applicazioni e deployment, in modo da decidere la loro posizione e integrarle col sistema.

Abbiamo valutato il sistema proposto usando benchmark disponibili al pubblico e integrandoli come funzioni serverless, in modo da testarle sotto tre aspetti diversi. Il primo aspetto é l'overhead introdotto dal sistema su un'applicazione eseguita su singolo nodo. In media, il nostro sistema aggiunge tra lo 0.27% e il 24% di overhead, in base al kernel eseguito e alla dimensione dei buffer, grazie a un efficiente meccanismo di trasferimento dei dati. Abbiamo inoltre eseguito un secondo set di esperimenti per testare il comporatmento del sistema in un piccolo cluster di tre nodi equipaggiati con FPGA Altera, su una singola applicazione. I risultati mostrano un miglioramento nel throughput (fino a 2.35 volte il numero di richieste processate per secondo) con lo stesso utilizzo dell'acceleratore e senza perdite in termini di latenza. Infine, abbiamo testato il sistema in uno scenario di non saturazione ma con più applicazioni (simulando la presenza di più tenants che condividono il dispositivo). In questi ultimi esperimenti, BlastFunction é stato in grado di raggiungere un utilizzo maggiore del dispositivo e un numero più alto di richieste processate grazie proprio alla condivisione dell'FPGA tra più applicazioni (5 invece che 3 col sistema nativo), con differenze minime in latenza e numero di richieste non soddisfatte.

Il testo é organizzato come segue:

- Il Capitolo 1 da un'introduzione generale al lavoro;
- Il Capitolo 2 offre una panoramica delle tecnologie e del contesto, oltre che del problema che viene affrontato;
- Il Capitolo 3 descrive i lavori presenti nello stato dell'arte;
- Il Capitolo 4 mostra e descrive il design del sistema proposto;
- Il Capitolo 5 dettaglia l'implementazione del sistema e dei casi di test;
- Il Capitolo 6 presenta i risultati sperimentali ottenuti per validare il sistema proposto;
- Infine, il Capitolo 7 riporta le conclusioni del lavoro e sottolinea i possibili lavori futuri.

Chapter 1

Introduction

The last decade saw the exponential growth of cloud computing as the primary technology to develop, deploy and maintain complex infrastructures and services at scale. Cloud computing allows to consume resources on-demand and designing web services following a cloud-native approach is fundamental to dynamically scale performance. Moreover, cloud systems provide multi-tenant access to a scalable pool of resources, which allows the services to scale indefinitely with measured performances and costs. To reach such goals, cloud systems have evolved, along with the underlying resource managers. The current technology comprises different virtualization techniques, such as VMs and Containers.

However, some workloads require computing power that current CPUs are not able to provide and, for this reason, heterogeneous computing is becoming an interesting solution to continue to meet SLAs. The first attempt in solving this issue was the introduction of GPUs into datacenters and cloud computing services, in order to accelerate compute-intensive workloads (such as Machine Learning training and inference and streaming video processing). GPUs offer relatively high performances for workloads and algorithms that fit into their architectural and parallel pattern. The issue is that GPUs excel at batch processing, in which batches of records are sent for processing in a high-throughput accelerator, but provide high latency when processing a single input record [1]. While many cloud applications and frameworks work in a batch fashion (e.g. MapReduce or Machine Learning training), many cloud-native applications work as *services*, in which *latency* is the major performance factor considered, as each request coming from a user of the service should be processed in the lowest time possible. Moreover, GPUs introduce power efficiency issues (as they consume more power than conventional CPUs) which in a datacenter scenario might be a significant cost factor.

In addition to standard and GPU-enabled instances, new resources available in cloud computing services are FPGA-based compute instances. FPGAs provide high level performances and low energy consumption, thanks to their flexibility in terms of hardware architecture. In fact, an FPGA can be configured to implement any given algorithm in an optimal way, as it is possible to finely tune the level of parallelism for any component of the resulting architecture based on the computation that has to be done. Moreover, their *reconfigurability* means that the physical chip does not change after every implementation update (as in ASICs, which need to pass through a design and manufacturing phase on

every algorithm change). This feature fits well with cloud-native applications, as they are continuously developed, updated and deployed through their lifetime. Thus, FPGAs found their use in a wide variety of applications. Workloads such as web search [2], image processing [3], compression [4], database operations [5], neural network inference [6] and many others can benefit from the use of specialized architectures and accelerators like FPGAs to timely react to the end users requests. Moreover, the introduction of the AWS F1 instances [7] as well as project Catapult [8] and project Brainwave [9] from Microsoft demonstrates that FPGAs will play a key role in the cloud in the next years.

To exploit FPGAs at their best in the cloud, hardware accelerators should be designed to meet latency requirements while optimizing throughput [8]. Requests from the outside network can come at unpredictable rates and they usually cannot be batched, thus minimizing latency becomes fundamental. The requests unpredictability can lead to an underutilization of the FPGAs, thus reserving one FPGA for each service that needs it can result in a waste of resources. From a cloud provider perspective, sharing allows to improve time utilization of the FPGA and the serverless computing paradigm can be a promising approach [10]. Serverless computing is an architectural pattern for cloud applications where server management is delegated to the cloud provider. Each application functionality is deployed by the user as a function and scheduled, executed, scaled and billed depending on the exact need of the moment. Within this context, we propose that compute-intensive kernels should be accelerated with shared FPGAs handled transparently by the serverless infrastructure: this will maximize utilization while reaching near-native execution latency.

To the best of our knowledge, a complete system for FPGA sharing and allocation in a cloud scenario (including microservices-based and serverless platforms) is still missing. Moreover, most of the existing frameworks and systems are not *transparent* to the application developer, and are not integrated with existing and known container orchestrators (such as Kubernetes).

In this thesis work we propose BlastFunction, a distributed FPGA sharing system for the acceleration of microservices and serverless applications in cloud environments. The goal of the proposed system is to provide a *transparent* and *scalable* system enabling *multi-tenancy* in the cloud FPGA scenario. Moreover, we aim at providing a *vendor-independent* and *reconfiguration-aware* allocation strategy integrated with an existing cloud orchestrator. We decided to focus on a *time sharing* approach for our system, in order to maximize the devices utilization (in terms of accelerator execution time) and optimize the use of devices from the cloud provider perspective.

BlastFunction is composed of three main components: a *Remote OpenCL Library*, multiple *Device Managers* and a central *Accelerators Registry*. The Remote OpenCL Library is a component which allows client applications or serverless functions to access the shared FPGAs in the cluster. The library is a custom OpenCL implementation which abstracts the use of the remote device access protocol and the communication to the other components of the system from the host code. The Device Manager is a

server application deployed on every node in the system and connected to each underlying FPGA board. It is the component providing the *time-sharing* mechanism, as it exposes a service through which functions and application can access the device concurrently. Moreover, each Device Manager exposes metrics about the runtime behaviour of the device, allowing the other components to act accordingly. Finally, the Accelerators Registry is the central controller of the system, which tackles the goal of allocating the available devices efficiently using runtime performance metrics. It does so by tracking the device utilization metrics from the Device Managers and performing an online device allocation algorithm which takes also care of reconfiguring the devices at runtime. In addition, it intercepts the deployment and removal of applications inside the cluster to integrate them with the system and perform the allocation algorithm.

We evaluated the proposed system using publicly available benchmarks and embedding them as serverless function, testing three different aspects. The first aspect is the overhead introduced by system on a single node and a single application. We show that our system adds a minimal overhead thanks to an efficient buffer transmission mechanism, between 0.27% and 24% depending on the executed kernel and on the size of the transferred buffers. We performed the second set of experiments to test the behaviour of the system in a small, three node cluster equipped with Altera FPGAs, on a single application. The results show major improvements in the throughput (up to 2.35x maximum requests processed per second) with the same devices utilization and without losses in the response latency. Finally, we tested the system in a non-saturated scenario but with multiple applications (simulating the presence of multiple tenants sharing the devices). In this experiment, BlastFunction was able to reach higher utilization and number of processed requests thanks to the sharing of the device with multiple functions (5 instead of 3 of the Native system), with minimal differences in latency and unprocessed requests.

This thesis is structured as follows:

- Chapter 2 provides an overview of the technologies and context, along with the problem that this work addresses;
- Chapter 3 describes the state of the art;
- Chapter 4 shows and details the proposed System Design;
- Chapter 5 digs into the implementation and deployment of the system and the test cases;
- Chapter 6 presents the experimental results gathered to validate our system;
- Chapter 7 draws the conclusions and presents the future directions of this work.

■

Chapter 2

Background and Problem definition

This chapter gives an overview of the main technologies and concepts on which this thesis is based, and defines the challenges and the goals of the proposed system. Section 2.1 provides an overview of cloud computing and related concepts, along with a description of orchestration and serverless systems that are used in the proposed work. Section 2.2 gives a background on FPGA technology and their use. Section 2.3 describes the OpenCL heterogeneous computing system. Finally, Section 2.4 defines the problem addressed by this thesis work and the goals of the proposed system.

2.1 Cloud Computing

As stated by NIST (National Institute of Standards and Technology) in [11], Cloud Computing is *"a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources that can be rapidly provisioned and released with minimal management effort or service provider interaction"*. By breaking down the definition and expanding it, we can highlight the main characteristics of a cloud computing system:

On-demand and Network Access The cloud users can access the resources offered by the cloud provider (e.g., servers, storage, application) over a broadband and high performance network connection, either through a public connection or a private/local one (in the case of a private cloud). In addition, the resources can be provisioned with a self-service automatic mechanism, without human intervention by the cloud provider. For example, servers and storage space can be requested through a web service or a command line, and their provisioning is performed without interacting with human operators.

Resource Pooling and Multi-Tenancy *Resource Pooling* means that multiple clients and applications access the same resources in a scalable way. The accessed resources quantity is chosen in order to fit the client's requirements, without requiring any changes in the application. *Multi-Tenancy* is an addition to the concept of resource pooling, as in this case multiple users access the same resources, but retain privacy and security over the informations. In addition, each *tenant* (application or user)

sees the system as its own, not noticing that the system is shared with multiple tenants. This technique allows applications and service to be easily deployed in the cloud system without changes in their behaviour, and allows to bill each tenant differently based on their actual resources usage.

Elastic Scalability The resources offered by the cloud system should scale up or down *rapidly* and *automatically*, based on the current service load or the user requests. This characteristic allows the user to pay only for the current resources which are used by its services, and to face unforeseen conditions (e.g., sudden spikes in the number of requests for a service) without keeping idle resources.

Measured Services The cloud provider measures all the resources which are offered by the system. The measurements are done at different abstraction levels (e.g., storage, number of users or services, total CPU load). This mechanism helps the provider to keep track of the offered services transparently and to provide a *"pay for use"* price model to the user.

To provide this kind of technology, different tools and concepts have been developed. In this section, we will highlight the main technologies related to the provisioning of resources in cloud computing (mostly processing resources).

Section 2.1.1 explains *virtualization* in the form of VMs, which was the first technology exploited to offer cloud services. In Section 2.1.2 we provide a description of *containerization* (in its most famous form, Docker Containers) and its main features. Then, Section 2.1.3 gives a brief overview over the main orchestration technologies employed in the last years used to manage VMs and containers in cloud scenarios. Finally, Section 2.1.4 describes the last technology introduced in cloud computing scenarios, which is *serverless computing*.

2.1.1 Virtualization technologies

Virtualization refers to the process of creating a *virtual* resource (e.g., processor, storage, network) instead of a physical resource. In the context of this thesis, we will refer to virtualization as *hardware virtualization*, which consists in creating a virtual hardware platform (VM) able to run an operating system.

Virtualization is performed on a given hardware platform by a *host* software (called Virtual Machine Monitor (VMM) or *Hypervisor*), which creates a simulated environment (the *VM*) for the *guest* software. The guest software will then run on the simulated platform as if it was running on the real hardware platform, with some limits introduced by different policies regarding the access to physical system resources. In fact, the VMM may limit the guest software's access to specific or partial physical resources such as storage or networking, depending on the virtualization policy defined on the host system. This mechanism allows to isolate the guest from the host system (or other guests) and to control the access to the underlying resources, which may be shared between multiple guests.

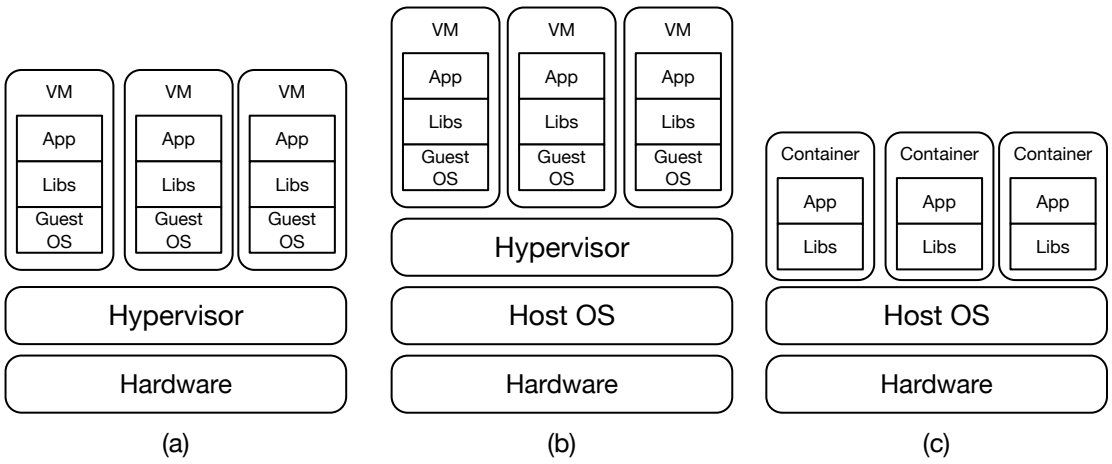

Figure 2.1: Overview of the major virtualization strategies: (a) Bare Metal hypervisor (b) Hosted hypervisor (c) OS-level virtualization.

Different kind of virtualization mechanisms can be found, based on the level at which the VMM runs and the virtualization method used. The first characterization of a virtualization system is given by the level at which the VMM/Hypervisor runs:

Type 1 - Bare Metal A bare-metal hypervisor (Figure 2.1a) runs directly on the host's hardware to control the hardware and the guest operating systems. In this way, is provides a high-performance access to the hardware from the guest machines and do not have the overhead of an additional host OS. Examples of bare metal hypervisors are Microsoft Hyper-V, VMWare ESXi or Xen.

Type 2 - Hosted A hosted hypervisor (Figure 2.1b) runs over an existing host operating system. The host OS manages the underlying hardware devices, so the hypervisor is more flexible in terms of compatible devices. Moreover, the VMM management is easier as it can be done from the host OS, which on the other hand requires to use part of the host resources (such as memory or processor). Examples of hosted hypervisors are VirtualBox, KVM or VMWare Player.

The second distinction that characterizes virtualization systems is the actual virtualization method used:

Full Virtualization With full virtualization, the VMM provides a full virtualization of the underlying hardware, including the full instruction set, I/O operations, interrupts and memory access. This mechanism allows to run unmodified guest OSes, but it requires the complete hypervisor mediation in order to work.

Paravirtualization In this case, the VMM and the guest OS collaborates, meaning that the VMM presents an interface similar (but not identical) to the underlying hardware to reduce the execution of expensive tasks for the virtualized environment.

Thus, the guest OS must be aware of the virtualization in order to use the VMM interface.

Operating System Virtualization OS-level (or kernel-level) virtualization (Figure 2.1c) allows to run private servers on top of the native host operating system. In this way, the host provides native performance and access to the underlying hardware. The issues are that the operating system is limited to the host OS and that the host kernel must be compatible with this virtualization method.

Hardware Assisted Virtualization This kind of virtualization relies on the underlying host processor capabilities to provide the virtualization interface used by the guest OSes. In particular, hardware assistance allows to run full and para virtualization at high performance without OS modifications. The issue is that this technique requires explicit support in the host CPU.

Virtualization plays an important role in cloud computing, as it allows the cloud provider to share the underlying hardware to multiple tenants. In fact, a VM can be created with a different share of resources from the host system, such as memory, storage or number of processors. This provides an implementation of the *elastic scalability* characteristic previously described at the beginning of this section. Virtualization also provides a standard and consistent development, building and deployment environment to the application developer and the system administrator, as all the application-specific configurations and dependencies can be encapsulated inside a VM, which can be moved across the cloud platform and among different servers in order to better manage the cloud system, without having to install the application every time. Finally, VMs provides an isolation mechanism to secure the deployed applications, as that the data processed by one application inside a VM cannot be accessed by other applications on different VMs. This allows to isolate the applications from different tenants, and to offer the cloud service to multiple organizations at the same time.

Virtualization presents also issues and challenges in addition to the advantages just described. A first issue related to virtualization (especially *full* virtualization) is the overhead introduced by the VMM. In fact, every time the guest OS performs a system call or requires to receive interrupts from the underlying hardware, the process has to pass through the hypervisor, which controls if the operations can be done by the guest. In case of a hosted hypervisor, the process latency may further increase because the host OS might also be involved. Para and OS-level virtualization mechanisms suffer less overhead, as the guest may communicate less with the hypervisor and more with the hardware. In any case, the performance of the virtualized application may be lower than the same application run on the host system, and the decrease in performance must be take into account when designing the service to run in a cloud environment. Moreover, the same issue applies to the virtualized devices (such as network devices), as the underlying hardware device may be shared among multiple applications which are using it concurrently without being aware of it. Another issue of virtualization is

related to the size of VMs. In fact, if the guest OS is fully virtualized, this means that each VM in the system must contain all the OS files, in addition to the application and its dependencies. If we take into account all the applications inside the cloud system, this mechanism introduces a significant overhead in terms of storage and memory, as each running application requires its OS and libraries to be loaded on disk and memory separately.

2.1.2 Docker Containers

As described in the previous section, *OS-level virtualization* allows to run private and isolated user-space instances of a given application or service on top of the host operating system. Many OS-level virtualization technologies has been created in the last years, starting from *chroot* (which allowed to partially isolate the filesystem between different applications) to LXC and Docker (which allows to run completely isolated *containers*). In this section, we are going to describe the main functionalities of the software we used to build and deploy our system's components in a virtualized environment, which is *Docker*.

Docker [12] is an OS-level virtualization platform first developed in 2013 by Docker Inc. It allows to develop and deploy software packages called *containers*. A container is essentially an isolated package running on the system which contains the application binary, its dependencies and all the needed software libraries and runtimes (including the OS distribution in some cases). The Docker runtime runs containers on the host OS without virtualizing the underlying hardware, thus avoiding a full virtualization overhead.

Docker offers multiple functionalities related to the creation and management of containers inside a system. The first distinction we need to make is between *images* and *containers*. A container *image* is the static definition of a container. The image contains a filesystem snapshot, which includes all the libraries and dependencies needed by the application to run. In addition, it defines useful metadata about the container which will be run, such as the environment variables needed and the exposed ports. Finally, it defines the *entrypoint* and the actual *command* which will be run when the container is started. Container images can be created and stored on a single host, or can be uploaded and downloaded from a *registry*, which is a storage service for container images. Registries can be public or private (internal to the organization, or with different authentication and authorization systems). A *container* is instead an instance of the image which is actually run on the host system. Docker allows the user to start multiple container instances on the same host or on different hosts, in order to scale the offered service based on the load that the application has to serve. When starting a container, the Docker engine takes a copy of the image and runs the entrypoint and the command indicated in that image. Being the container a new copy of the base image, all the changes made by the application during its lifetime are not saved on the image filesystem, meaning that the container storage is ephemeral. In addition to running containers, docker provides also a virtual networking environment and the ability to mount external filesystems and devices at runtime. This allows to expose the containerized application to the cluster or

the external network, and to store and update a non-ephemeral filesystem external to the image one. For example, a database engine running in a container will be able to write and read the database content on a stable storage.

Docker performs the virtualization process by using many underlying components, some of which are included in the Host OS kernel. If we look at the components used on the Linux kernel in particular, Docker relies on:

Namespaces Namespaces provide the first level of isolation, as they limit the access to certain resources and parts of the OS (e.g., processes, network, filesystem). They are used to provide a specific view of the system to the application, for example providing a limit on the PIDs that can be seen (and changing the virtual PID of the application) or on the filesystem (effectively creating a *chroot* environment for the containerized application).

Control Groups Control groups (cgroups) allow Docker to share available hardware resources to containers and optionally enforce limits and constraints. For example, the developer may want to limit the memory available to a specific container to avoid security issues (e.g., exhausting the host OS memory).

Union Filesystem Union filesystems are filesystems that work by creating layers. In this way, if multiple containers share any layers (e.g., the OS or library layers) these layers can be shared among them during storage (but not while running).

These components were first incorporated in a format called *libcontainer*. Starting from Docker 1.11, these components and other functionalities have been moved to new projects, in order to standardize the underlying containers technology. The lowest component in the stack is called *runc*, and it is a lightweight tool which is only meant to run containers. Thus, runc does not manage external resources such as network or filesystem volumes. Runc is then managed by another component called *containerd*, which exposes a service used by container tools such as Docker. When the Docker daemon wants to start a container, it sends the container image and other information (such as the virtual network interface and the external mounts) to containerd, which forwards the commands to runc. Finally, runc will start the container and notify the upper levels.

The main advantage of Docker w.r.t VMs is the reduced overhead, given by the fact that it does not perform hardware virtualization and does not intercept system calls and access to devices, as they are managed by the host OS. The absence of hardware virtualization means however that the applications must be compatible with the underlying hardware architecture, or that there must be a different version of the application for every different architecture. Most cloud computing environments use a homogeneous architecture, at least in terms of CPU ISA (usually x86/64), so the issue is mitigated.

Another advantage of Docker containers is the reduced size of a container image w.r.t a VM. In fact, a VM contains the entire guest OS, drivers, libraries and the application, while a container needs only the application and its dependencies, as the OS libraries and drivers are provided by the host.

The drawback is that there may be compatibility issues between containers and the host system, if the containerized application has been developed for a different OS. For example, at the beginning Docker allowed only Linux distributions to be run as containers, by running natively on linux hosts and using an underlying hypervisor on Mac OS and Windows systems. Nowadays, it is also possible to run Windows containers, but only on compatible Windows hosts as they require to use the underlying system DLLs and components. In general, as with every OS-level virtualization system, compatibility issues may arise if the application and the host system are not compatible.

2.1.3 Cloud Orchestration: Kubernetes

As stated in Section 2.1, one of the main characteristics of cloud computing is elastic scalability, which means that the offered services and infrastructure should scale *rapidly* and *automatically* based on the workloads. To enforce these requirements, multiple tools and systems have been created. The aim of this tools is to efficiently manage the resources offered by cloud platforms automatically and with less effort by the system administrator. In this section, we are going to describe a cluster management tool which is frequently used in cloud computing systems, called Kubernetes.

Kubernetes is a portable and extensible open-source platform for managing containerized workloads and services. Its aim is to facilitate both declarative configuration and automation of services and jobs inside a cluster (either on-premise or in a cloud environment). In particular, it offers a lot of features, such as deployment, scaling, load balancing, logging, and monitoring of services and workloads in general. These features can be removed or added based on the system requirements, as the platform can be extended with additional components.

Architecture Overview

Kubernetes allow to manage a cluster of physical and virtual machines using a shared network to communicate between each server. The cluster is the platform where all the components and capabilities are configured and managed. The main distinction in a kubernetes cluster is between *master* and *node* servers.

The *master* server in a kubernetes cluster acts as the primary control plane for the services and workloads deployed inside the cluster. The role of the master is, in particular, to expose an Application Programing Interface (API) for users and clients. This API is used by both external users and internal components of the system in order to change the cluster status and the deployments inside the cluster. Also, it's used by external components in order to extend the platform. The master server includes multiple components in order to perform its duties. These components may be installed on a single machine or on multiple machines in a distributed fashion, depending on the size of the cluster or the availability requirements. The master components are:

- *etcd* is a distributed key-value store which represents the main storage for the

cluster configuration data,

- *kube-apiserver* exposes the Kubernetes API to the users and applications,
- *kube-controller-manager* manages workloads and their lifecycle, along with specific controllers (for example for the replication of workloads),
- *kube-scheduler* assigns workloads to the available nodes in the cluster,
- *cloud-controller-manager* allows kubernetes to interact with the external cloud provider's system (if available) in a transparent way.

A *node* in a kubernetes cluster is a server that perform work by running containers. Node servers have components which are required to run the containers and to communicate with the master component. In particular:

- the *container runtime* (e.g., Docker) allows to run the containers on the node,
- *kubelet* is a service which acts as the main contact point with the master API and manages the workloads by controlling the container runtime,
- *kube-proxy* manages the virtual networking environment on the node and allows all components and workloads/containers to communicate with the rest of the cluster.

Kubernetes Resources and Objects

While containers represent the basic block managed in a cluster in addition to VMs, kubernetes provides additional layers on top of them. Moreover, additional resources and object types can be used to better represent the cluster configuration.

A *Pod* is the most basic unit which can be managed with kubernetes, as it's the unit which is actually assigned to nodes (it is not possible to assign a single container to a node). A Pod represents one or more tightly coupled containers that are seen as a single application by the system. In particular, the user can define a Pod template by listing the containers it is made with, along with specific metadata regarding additional requirements (such as resources limits and requests). All the containers inside a Pod will share the IP address, environment and volumes.

In general the user does not work with single Pods, but he will define a specific way in which the Pods should be deployed into the cluster. A group of identical Pods is controlled by a *Replica Set*. It is defined by including the pod template with additional parameters regarding the number of replicas and how the Pod should be replicated.

Deployments are one of the most common workloads, and give a higher degree of control over Replica Sets. A *Deployment* defines a desired state of the system which must be reached. For example, the user may define a number of desired replicas for the same Pod, or decide to scale up or down that number. The Deployment Controller will then notice the change in the Deployment definition and perform the required operations

in order to reach that state. In addition, Deployments allow to use selectors, labels and other metadata in addition to the Pod/ReplicaSet definition. Finally, a Deployment may perform rolling updates, in which older Pods are swapped with a new version one at a time (or with other policies) in order to keep the application online.

In addition to Deployments, there exist other kind of controllers that allow the user to define a different Pods management policy. For example, *StatefulSets* offer ordering and uniqueness guarantees to Pods that need persistent data handling (e.g., databases) or persistent naming. *DaemonSets* run a copy of the underlying Pod on every node in the cluster that meets a certain requirement. They are useful for application that must run on every node, such as monitoring or logging services. Finally, *Jobs* allow to define task-based workflows instead of long-running services. A Job may be any batch processing application which must run until completion. The jobs controller will schedule pods in order to meet the completion requirement, and can also be used to parallelize the computation on a coarse grain.

Apart from Pods and Controllers, kubernetes allows to define additional resources. A *Volume* represents the basic storage unit inside kubernetes. All the containers inside a Pod share the same Volume, which is however deleted when the Pod is removed. *Persistent Volumes* instead are not deleted, and can be shared among multiple Pods.

A *Service* is a component which acts as an internal load-balancer, and allows to expose the application served by multiple pods as a single entity, by providing a stable endpoint. Thanks to this, it is possible to scale up and down the number of Pods and face failures without changing the endpoint. A service can also be exposed outside the cluster as an external service using an *Ingress*, which acts as a router for multiple services. Additionally, the service can be connected to an external Load Balancer, such as the cloud provider's one.

2.1.4 Serverless Computing

Serverless computing is a cloud computing model in which the cloud provider manages the hardware infrastructure and provides automatic scaling of the compute resources based on the amount of resources consumed by the application. This model presents a series of advantages, the first being that the cost model is extremely elastic and follows almost linearly the resources usage of the application. In fact, the user does not need to reserve additional resources in order to face spikes in the application load. Moreover, the scaling features are automatically performed by the cloud provider, which can employ optimizations aimed at reducing the user and its own costs of service for the cloud infrastructure. One of the disadvantages for the user is instead that the total price for the service is difficult to estimate, as it means estimating the application resources in detail.

One of the most used severless paradigms is Function-as-a-Service (FaaS), in which the user provides the code for each function in the application in a separate way in order to have it automatically managed by the cloud provider. Multiple functions are then

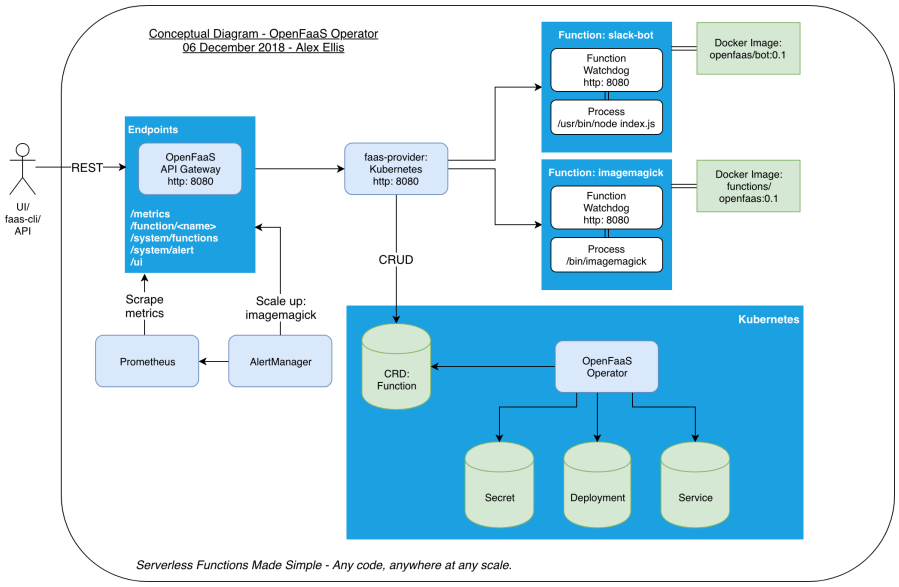

Figure 2.2: OpenFaaS architectural overview [13].

connected through standard interfaces and protocols (e.g., REST, JSON, gRPC) in order to create the entire application. This allows the developers to decouple the application and apply a differentiated scaling mechanism for different parts of the application, but also adds challenges in terms of distributed system debugging.

OpenFaaS

OpenFaaS [14] is an open-source framework for the deployment of serverless applications. The framework can be deployed in container-based clusters such as Docker Swarm or Kubernetes, and offers both a web UI and a CLI to build and deploy functions. A general overview of the framework can be seen in Figure 2.2. The most important component of the framework is the *Gateway*. The gateway is the main endpoint for the system, meaning that functions are invoked based on the requests arriving to it. It is also responsible for the automatic deployment and scaling of function instances based on the developer requirements and the functions load. The *Prometheus* and *AlertManager* components are external to the OpenFaaS system. The Gateway exposes metrics about the number of requests to each function and the mean latency, and pushes them to the Prometheus service, which aggregates them. Based on the aggregated metrics, the AlertManager notifies the Gateway when a function instance receives too many requests in a given time window, and the Gateway decides to scale up or down the number of deployed instances. Moreover, when a function remains idle (no incoming requests) for a given period of time, the Gateway scales the function deployment to zero instances in order to save capacity within the cluster.

A *Function* in the OpenFaaS system is defined as a handler process (written with any programming language) which integrates with the already present *Watchdog*. The

Watchdog is a process which receives the requests forwarded by the Gateway and sends them to the function handler in order to be processed. Handler process and Watchdog are bundled together in a docker container which is then managed by the orchestrator (Kubernetes in the case shown in Figure 2.2) under the Gateway conditions. Using this system, the developer must only define the handler process function and some additional metadata regarding it (e.g., the docker image name and additional details about the deployment, such as annotations and secrets). Moreover, the OpenFaaS maintainers offer a set of pre-made *templates*, which are skeletons for the handler function and the docker image useful when developing a serverless function.

2.2 Field Programmable Gate Array

In this section we are going to provide a general overview about FPGAs and their recent introduction in cloud environments. An FPGA is an Integrated Circuit (IC) that can be programmed (possibly) infinite times after manufacture. The idea of reconfigurable hardware does not refer only to FPGAs. In fact, many programmable technologies existed before that, such as Programmable Logic Devices (PLDs) and Programmable Logic Arrays (PLAs).

The issue with these kind of architectures is that they can not be exploited for more complex designs, as they present a fixed interconnection between components and provide few levels of reconfigurable components. FPGAs instead provide complex reconfigurable components and a fine-grained interconnection mechanism, allowing the designer to reconfigure them with complex circuits. This led to their utilization in various fields and with different applications.

2.2.1 FPGA architecture

The basic structure of an FPGA can be seen in Figure 2.3. It is composed of:

Configurable Logic Block (CLB) The CLB is the basic configurable block inside an FPGA, and provides different behaviours based on the given configuration. Internally, it is composed of logic cells containing a Look-up Table (LUT) (also called ALM for Intel FPGA), a Full adder to perform simple arithmetic operations, and a D-Type Flip-flop used to give sequential behaviour if needed. Each of these components can be configured and activated differently inside the CLB to give it a different configuration.

Switch Matrix (SM) The SM is the interconnection component of the FPGA, which allows to connect the CLBs and other components in different ways. The SM is a 2D matrix of interconnection signals that can be reconfigured at every crossing point, which allows it to be configurable as well.

Input-Output Blocks (IOBs) The IOBs are necessary for the communication between the FPGA and external components and circuits. Depending on the different FPGA

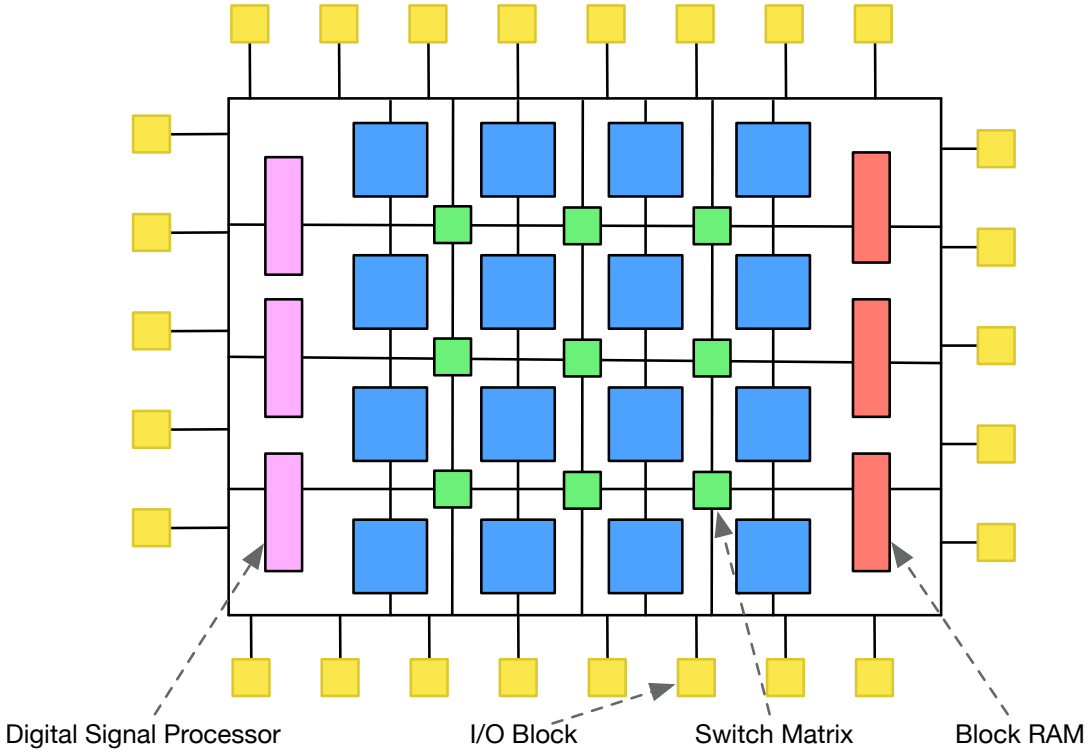

Figure 2.3: FPGA structure (simplified) with CLBs, IOBs, BRAMs and DSPs.

model and board, the IOB pins are used for different purposes (e.g., memory, network, video connections).

Digital Signal Processors (DSPs) DSPs are specialized component used to perform mathematical and logic operations. They are included in the FPGA because they allow to reduce the number of resources used for complex mathematical operations (such as floating point multiplication and addition) with a higher performance than CLBs.

Block RAMs (BRAMs) BRAMs are memory blocks directly instantiated on the FPGA plane, along with CLBs and other components. They provide a faster memory mechanism than external memories such as DDR because they are produced with a faster logic and are on the same fabric, so they have a lower latency.

2.2.2 FPGA Reconfiguration and Tools

The components described in the previous section can be configured in order to implement different circuits. In general, this makes FPGAs a turing-complete device, as in the simplest case the FPGA can be configured to implement a soft-core processor which is turing-complete.

The configuration by which the FPGA is programmed is called *bitstream*. The bitstream includes a list of bits that have to be written and stored on each component of the FPGA (e.g., CLBs, DSPs, SMs) in order to reconfigure it. When a bitstream is loaded onto the FPGA, a specialized component called Internal Configuration Access Port (ICAP) distributes the various configuration bits to all the components of the chip.

Bitstreams are created through a process called *synthesis*, in which a Register Transfer Level (RTL) representation of the circuit is transformed in the actual vendor-specific bitstream. To obtain the RTL representation, the hardware designer may use an Hardware Description Language (HDL) such as VHDL, which allows to define every logic component and behaviour. However, writing HDL may be a difficult and long task for complex or large circuits. To make the process easier and less error-prone, and to allow designers to create complex accelerators using FPGAs, vendors developed High Level Synthesis (HLS) tools. An HLS tool allows a software or hardware developer to synthesize hardware starting from high level languages (e.g., C++, Java) and using single *pragmas* and tool-specific configurations. The HLS tool compiles the high level specification and, using the given requirements and pragmas, translate the high level representation into a RTL representation which can be synthesized. Examples of HLS tools are Vivado HLS and SDAccel by Xilinx, and the FPGA OpenCL SDK from Intel (previously Altera OpenCL SDK).

The advantage of HLS tools is the increase in productivity for the hardware designer, as the tool abstracts detailed concepts like timing and pipelining, and allows the designer to focus on the architecture and the flow of data inside the accelerator. Moreover, HLS tools allows software developers to program FPGAs in addition to standard accelerators such as CPUs and GPUs. On the other hand, the simplification of the design process removes part of the designer freedom when creating complex or high-performance circuits, as some low-level details about the underlying architecture and constraints are abstracted away. Thus, for high-performance accelerators and strict timing-based circuits, hardware designers still prefer HDL languages.

2.2.3 Usages of FPGAs

Given their reconfigurability and the possibility to create a specific architecture for a given algorithm, FPGAs are widely used in many fields. The first advantage given by FPGAs is that they can employ a much higher level of parallelism w.r.t CPUs. In fact, while the execution of an algorithm on CPU is limited by the number of Functional Units available and the parallel access to memory, an FPGA can be configured with a variable number of units and memory blocks. For example, the design may decide to synthesize multiple Arithmetic-Logic Units (ALUs) based on the amount of parallel operations that should can be carried out for a given algorithm, or instantiate multiple BRAMs to enforce parallel access to local or cached data. Also, the application-specific architecture can employ multiple optimizations, like *pipelining* or *loop unrolling* which are different from their software counterpart, as they can be directly instantiated in hardware. In

addition to the performance improvement, FPGAs work at lower frequencies than most CPUs and GPUs, thus they enable *low power* designs.

Application Specific Integrated Circuits (ASICs) presents a challenge to FPGAs, as they present all the FPGAs advantages but with increased performance and lower power consumption. In fact an ASIC is a custom circuit which can run at higher frequencies than its FPGA implementation, without the area and timing overheads. However, ASICs can not be reconfigured. This means that if the implemented algorithm changes, the ASIC must be re-designed and printed again. Also, for specific algorithms and with small volumes, an ASIC might not be the perfect choice given the costs involved in its creation.

Many fields benefit from the advantages just described (high performance, low energy consumption and reconfigurability). An example is given by the telecommunications sector, in which FPGAs are used in switches and routers across the network, as they allow to update the switching behaviour without changing the network devices. Another sector which employs FPGAs is finance, where they are used to accelerate High Performance Trading algorithms working with low latencies. Finally, the last sectors which entered the FPGA scenario are *genomics* and *machine learning*, which benefit from the acceleration given by custom circuits and their reconfigurability as the algorithms keep changing.

2.2.4 FPGAs in Cloud Scenarios

In recent times, cloud providers started to include hardware accelerators in their cloud offering to target high performance computing workloads. Examples of this are Machine Learning, Genomics and Scientific computations, in which the algorithms require a high level of parallelism to perform at an acceptable level. In this context, the first devices to be offered in cloud scenarios have been GPUs, and later FPGAs. In particular, the cloud FPGAs offer can be distinguished in two types.

The first type of FPGA usage in the cloud is as an underlying technology for specific kinds of services. An example of this is Project Catapult [8] by Microsoft, in which FPGAs are used between the network interface and the server processor to offload certain workloads and provide an *acceleration plane* in addition to the standard compute plane of the cluster. The project has been employed to accelerate Machine Learning workloads with Project Brainwave [9], allowing Microsoft's cloud provider (Azure) to offer accelerated Machine Learning as a service.

FPGAs are also directly offered to the end user of the cloud service. The first cloud provider offering FPGA instances has been Amazon Web Services (AWS), with their *F1 instances* [7]. F1 instances provide multiple development environments to design and run FPGA-based applications, ranging from RTL design to HLS environments (such as SDAccel by Xilinx, which uses OpenCL and C++ as specification languages). In order to deploy an FPGA-based application, the developer needs to wrap its bitstream in an Amazon FPGA Image (AFI), then upload it to the AWS system and deploy it into a specific VM in order to reconfigure the connected FPGA. The FPGAs are connected on

the host machine through PCI-Express, and a pass-through link is given to the VM for the requested boards, providing full and private access to them. AWS allows to connect multiple FPGA boards to a single VM, and to instantiate multiple VMs using the same bitstream (connecting different FPGAs). The AFIs can also be sold or rented using the AWS store, which represents a meeting point for hardware designers and companies which need FPGA designs for their applications.

2.3 Heterogeneous Computing and OpenCL

In this context, multiple vendors (e.g. Intel, Nvidia, AMD) gathered to find a solution for the heterogeneous programming challenge. *OpenCL* is an open and royalty-free standard for heterogeneous parallel programming, which allows to greatly increase the speed and responsiveness of applications thanks to the use of parallel co-processors. The OpenCL standard defines a *memory model* and a *programming paradigm* which allows to run algorithms written in the same language for different hardware platforms, tuning the application parallelism based on the underlying architecture.

The computing model defines how to structure the application code in order to be compliant with the OpenCL standard. In fact, the device is divided in Compute Units (CUs), each subdivided in Processing Elements (PEs). The programming model defines instead three different concepts: *Kernel*, *Work-Group* and *Work-Item*. A Kernel is the function executed on the OpenCL device. It can be called multiple times and can run on multiple CUs and PEs, depending on the underlying hardware. Each invocation of the kernel is called a Work-Item. Finally Work-Items are grouped into Work-Groups to allow a finer control over the topology of the kernel invocations. In fact, given the input and output data, Work-Item and Work-Group can be used to define which portion of the data that particular invocation must process. This allows to execute the kernel in a parallel way (if the device allows it) without dependencies and concurrency issues. In more complex cases, the programmer can enforce *barriers* and synchronization points in order to better manage the concurrency between multiple Work-Items.

Kernels and Work Groups/Items are concepts related to device programming. In order to run Kernels and exchange data from and to the device, the programmer must also program the *host* application, which uses an underlying vendor-specific host library. Multiple vendor libraries can be installed on the same system. To face this issue Khronos [15], which is the maintainer of the OpenCL standard, released a framework called *Installable Client Driver (ICD Loader)*. The ICD loader allows multiple libraries to be on the same system, as they are dynamically loaded based on the used device.

The OpenCL standard defines additional concepts related to the host code. In particular, a *context* represents multiple devices which are being controlled by the application. It includes a *program* which consists of multiple kernel, and multiple *command queues*. A command queue is a queue used to send commands to the device, such as reading and writing from the device memory to the host memory and invoking kernels (giving also

the number of Work-Items and groups). The command queue sends commands to the device *in-order*, enforcing a order on the operations executed from that queue. However, there is no synchronization between multiple queues, meaning that multiple operations could be running at the same time on the device without any order between them (if they are sent from different queues). Finally, the host can use *Events* to check the status of operations and to allow *non-blocking* asynchronous operations and kernel invocations.

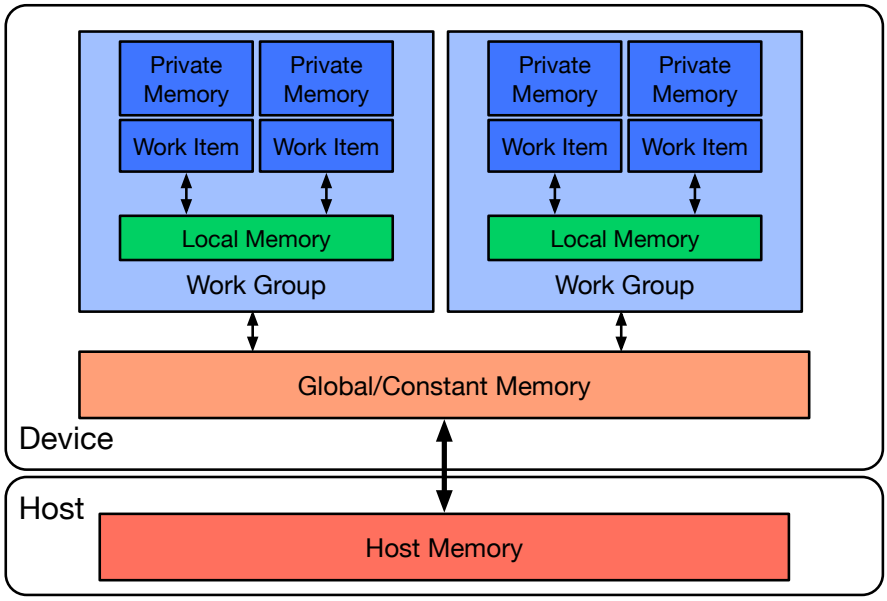

Figure 2.4: OpenCL standard memory hierarchy.

Regarding the OpenCL memory hierarchy, Figure 2.4 shows the different levels of memory defined in the standard and how they are accessed. The first difference in the memory model is between *host* and *device* memory, as they may be separated by a communication mean. For example, the host may be a standard cpu-based machine with a PCI-Express bus, while the device may be a GPU processor on a PCI-e board. In this case, the host memory is the host DDR connected on the bus, while the device memory is located on the GPU board. Inside the device, different memory locations are defined based on their level. The outer level is the *global memory*, which is visible and accessible to all the Work-Groups, items and kernels running on the device and can receive and send buffers from the host memory. The *Local Memory* is instead shared within a single Work-Group. Finally, the *Private Memory* is visible only by the single Work-Item. The different memory levels inside the device may be mapped to different components and locations. For example, on FPGAs the local memory is usually synthesized using local BRAMs, while the private memory is located on fast registers, depending on its size.

2.4 Problem definition and goals

In this section, we will present the problem addressed by this thesis work, based on the background concepts presented in the previous sections. In particular, Section 2.4.1 describes the problem addressed by this work with the related challenges and opportunities, while Section 2.4.2 provides the goals of the proposed system.

2.4.1 Problem definition

As described in the Section 2.2.4, FPGAs have been recently added to cloud computing systems in order to speed-up computations and services. However, the way in which FPGAs are offered is by adding them to the user VM by using a PCI-express pass-through link (e.g. AWS F1 [7], Nimbix cloud [16]). This means that only one user's VM has access to the underlying FPGA at a given time.

If the service is experiencing a high load (in terms of incoming requests), then using the entire FPGA is advantageous. However, when the number of requests is lower than the maximum throughput attainable, the FPGA remains underutilized. Here for FPGA *utilization* we mean *time* utilization, as in percentage of time in which the accelerator is actually executing meaningful work. Moreover, the current usage of FPGAs in the cloud is affected by underutilization in two different cases:

- Single user, multiple services with different accelerators
- Multiple users, multiple services with same accelerator

In the first case, if not all accelerators are used for 100% of the time, then the user is paying more than the needed resources. In the second case, multiple FPGAs are used even if the configured bitstream is the same. In general, the problem with current cloud FPGAs offering resides in the *non-linearity* of the service, as the allocation of devices to users follows a step function instead of a proportional line. Thus there is a resource optimization opportunity for the cloud provider, which could provide the same performances by using less physical devices by *sharing* them between multiple services and users.

The opportunity can be seen even more in the context of serverless computing, as no FPGA-based offer currently exists for serverless systems even if it represents a promising approach [10]. Also, given the automatic management of the underlying hardware resources needed by the serverless model, the devices allocation can be abstracted away from the user's perspective. The challenges in this context are providing an *optimal* allocation of resources given the devices utilization, and an *isolation mechanism* to allow multiple users to share the same device in a secure way.

2.4.2 Goals

Given the concepts and challenges in the previous sections, the goal of this thesis work consists in designing and developing a distributed system for FPGA sharing in cloud,

and in particular, serverless scenarios. The system is called BlastFunction, and it aims at meeting the following goals:

Multi-Tenancy The system should support multiple users deploying their FPGA-enabled serverless functions without impacting other users services and provide isolation between multiple applications using the same device.

Transparency The system should be transparent from the application point of view, allowing the developer to use the same host code to access the shared device as if it was a physical device directly connected to the application.

Scalability The system should not lose performances and allow to scale the number of deployed functions in relation to the number of physical servers and devices available.

Reconfiguration-aware and Accelerator-independent The system should allow the function instances to reconfigure the shared device with any given bitstream, without making assumptions about the used accelerator, and change the devices allocations based on the new state introduced by the reconfiguration procedure.

Cloud orchestrator integration The system should integrate with an external cloud orchestrator (e.g. Kubernetes) to perform the automatic allocation of the shared device to the requesting function instances, and to migrate the instances when the device configuration changes.

■

Chapter 3

State of the art

In this Chapter, we will present the analysis of the State of the Art related to this thesis work, focusing on the integration of FPGAs in cloud computing environments. We first provide a bird's eye view of the analyzed context in Section 3.1, by enumerating different classification criteria and the related works. Section 3.2 describes the works that share an FPGA device on a single host in the system. Section 3.3 describes multi-node system for FPGA sharing and allocation. Finally, Section 3.4 shows the works focused on *FPGA Pooling* and the related scheduling and management techniques.

3.1 Works classification

In this section, we are going to give a classification of the described works by diffent criteria.

The fist distinction among the analyzed works is the *communication method* used to access the shared or virtualized device:

PCIe-Passthrough This is the lowest level method available, as it works by directly connecting a single VM or container to the FPGA device. Notable works using this kind of communcation level are the AWS EC2 platform [7] and the work in [17].

Paravirtualization In this case, the requesting application VM is connected to a host device driver which virtualizes the access to the resources. This mechanism is used by *pvFPGA* [18] and *hCODE* [19].

API Remoting This mechanisms is the most used in the analyzed state of the art [20, 21, 22, 23, 24, 25, 26], and it works by defining a custom API to remotely access the device. It allows multiple applications to control the shared device and to perform both space and time sharing. A special API Remoting technique is represented by the work in [27], as in this case the system exposes a microservice for each accelerator, and not a general API for the entire system.

Direct Network Access Used by *Catapult* [28] and in [29], this method works by exposing the FPGA through its network interface, thus enabling a low-latency access and network filtering and offloading techniques.

We can make a second classification of the works analyzed, based on their sharing mechanism. In particular, we distinguish between:

Space-Sharing [30, 29, 21, 19, 24, 25] This sharing mechanism employs FPGA virtualiation through the use of *Partial Dynamic Reconfiguration (PDR)* or *Overlays* in order to run multiple different accelerators on the same FPGA, which are used by different applications. Space sharing allows to use the entire resources on the device (in terms of logic blocks), but requires careful handling of the accelerators in order to minimize the reconfiguration time and their interaction with shared memory interfaces on the board.

Time-Sharing [18, 20, 28, 22, 23, 24, 31] Time-Sharing works by *multiplexing* multiple requests from different applications on the same accelerator in the FPGA board (it can be coupled with *Space-Sharing* the board is split in multiple accelerators of regions). In this case, the challenge is to efficiently schedule the incoming requests in order to minimize the latency on the application side, and managing the memory accesses in order to fit in the available I/O bandwidth between the accelerator and the host system.

A final distinction between the described works is their *batch* or *service* based nature, where the batch/service terms are related to how the FPGAs are seen by the system:

Batch System [18, 30, 29, 21, 19, 23, 26, 24, 32, 25] In a batch-based system, the workloads (and the connection to the offered FPGA) are seen as limited in time. Therefore, in this kind of system the scheduling and allocation algorithms are decided based on the *lifetime* of each job. We include some *"as-a-Service"* systems in this category because, even if the authors considers the FPGA offering a service, the devices are not continuously serving requests, but are working on batch jobs.

Service-Based [20, 17, 28, 22, 27] In Service-Based systems, the FPGAs are continuously working by receiving and processing requests from the system or the application. Many *pooling* systems are included in this category because the device are *always online* and the challenges are related to balancing the load in the system and routing the requests to the offered devices. Moreover, works such as [27] directly expose the underlying FPGA accelerator as a standalone service.

3.2 Single Node FPGA Sharing

The first approach used to integrate FPGAs in cloud computing scenarios is virtualization, as it were (and still is) the most used technology to deploy applications and services. Within this context, each FPGA board in the cluster is attached to a VM on the host and can be accessed only by its applications or the applications on the same host.

An example of this kind of virtualization is *pvFPGA* [18]. In this work, the authors propose a para-virtualization mechanism on top of the Xen Hypervisor to connect the

FPGA board to the requesting VMs. The mechanism works at the device driver level, instead of intercepting and redirecting API calls to the privileged domain or the host user space. This means that the device is directly accessed by the VMs, and the system provides a close-to-zero overhead. Moreover, pvFPGA allows multiple domains on the same hypervisor to use the accelerator, by allocating a shared data pool to each unprivileged VM.

Another case of FPGA sharing on a single host through virtualization is the work proposed by *Asiatici et al.* [33], which describes a method and runtime to design and map accelerators onto FPGAs in cloud settings. In particular, the work proposes a runtime composed of a high-level API to interact with the accelerator, an execution model which support both hardware and software execution and a shared memory model able to isolate the different applications using the same board. In this case, the FPGA board is shared by using Partial Reconfiguration (PR) and a reconfiguration-aware controller and a runtime manager running on a static region of the board (over a MicroBlaze soft-core).

A similar work to [21] is *Vaishnav et al.* [24], which introduces a system to perform *resource elastic FPGA virtualization*. This approach allows multiple applications to program an elastic portion of the FPGA concurrently. The system uses a Resource Manager to schedule the different bitstreams reconfigured on the FPGA and the requests coming from the host applications. The Resource Manager scheduler employs both space and time *multiplexing* to decide which modules to shrink or grow based on a waiting queue of kernels to be executed, along with metadata concerning the kernels and bitstreams.

In *VineTalk* [22], the authors propose a *software layer* between application software and accelerator hardware which allows to share a single FPGA across multiple applications on the same host. The framework makes use of a shared memory communication layer and a software controller to schedule the tasks coming from different applications. Moreover, it allows hardware designers to easily incorporate new kernels by overloading two functions in the framework, and the system is integrated with the SDAccel environment by Xilinx. The authors tested the framework on a financial use-case, showing less than 4% overhead to the application execution time over a shared FPGA on a single node. However, some limitations of the work include the lack of dynamic reconfiguration to implement space-sharing, lack of transparency (the library presents a custom-made API) and lack of a cluster-level orchestrator (the system is limited to one node at the time of writing).

3.3 FPGA Sharing in cloud environments

Many works deal with the sharing and allocation of FPGAs on a multi-node setting, in which scheduling and allocation algorithms become more important, in addition to networking and processing constraints.

The first work in this category is *Byma et al.* [30]. The work raises the level of abstraction w.r.t previous works, as the authors propose a hardware and software framework to enable FPGAs virtualization in a cloud environment based on VMs and OpenStack. In particular, the hardware framework allows to define multiple Virtual FPGA Resources (VFRs) on every board. VFRs are then seen as separate resources by the OpenStack layer. The user decides which bitstream to use for its VM, and the OpenStack controller dialogates with a node-level *Agent* to program the allocated VFR and connect it to the booted VM. The system is evaluated by deploying different load balancer implementations to test the different between VM and FPGA, showing promising results.

In hCODE [19], the authors describe an open-source system to simplify the design, management and deployment of FPGA accelerators in cloud environments. The novelties of the proposed system include the addition of a resource-tracking mechanism, a cluster-level accelerator scheduler and a task execution controller, along with a central repository to store the different bitstreams and configurations.

A different method to share FPGAs in a large scale system is shown by *Weerasinghe et al.* [29], which decouples FPGAs from CPUs by offering them as standalone devices inside the cluster. The authors implement a network-attached FPGA architecture consisting of a *user logic* part and a *network service* and *management* layers. In particular, the user logic part allows to perform partial reconfiguration, thus sharing the FPGA board among multiple users. The management layer exposes the device to the cloud platform and allows to configure it remotely when requested. By implementing an additional service to manage the offered FPGAs, the proposed system is able to integrate with an OpenStack cluster used by the authors.

Catapult [28] represents another case of FPGA integration with cloud-scale architectures. In this work, Microsoft proposes an FPGA layer between the servers and the interconnection infrastructure. This is done by deploying an FPGA board on every server in the datacenter and use it as a *network-offloading* device. The FPGAs in the system are able to filter packets and workloads coming from the network plane before they are received by the server for processing. Moreover, workloads can be directly executed on the reconfigurable device. This allows the system to perform both *network* and *service* acceleration, as the device is able to execute accelerated tasks coming from the locally attached server or from any other host in the network. The authors finally describe a Hardware-as-a-Service (HaaS) model in which jobs can be sent and processed by FPGAs remotely and with a Resource Manager (RM).

Orellana et al. propose two works regarding the inclusion of FPGAs in heterogeneous private clouds. The first work [34] describes the architecture of the system, which allows two usage models: the first model is an Infrastructure as a Service (IaaS) model in which users require the exclusive access to an FPGA board, while the second model presents a Software as a Service (SaaS) system in which the user indicates an accelerated service and its Quality of Service (QoS) parameters (e.g. latency). Based on the requested model, service and parameters, the authors propose different allocation strategies and scheduling decisions for the SaaS model, evaluating it on a real testbed (based on OpenNebula). In the second work [17] the authors extend the previously proposed system with a multilevel cloud scheduling framework. In addition, they propose multiple node-level FPGA-aware scheduling strategies based on multiobjective metric aimed at providing QoS support and incrementing the number of requests serviced with their Service Level Objectives (SLOs) fulfilled.

A final work regarding the integration of FPGAs in VM-based cloud systems is *Tesfatsion et al.* [32]. The authors propose a system for efficient resource management in heterogeneous clouds including FPGA accelerators. The system employs an energy-aware technique that uses the applications performance and deadlines to allocate FPGAs to the most energy demanding applications. Once the system has allocated an FPGA to a VM, it optimizes the energy consumption while keeping the required performance of the remaining applications by employing frequency adaptation and CPU scaling. The authors measured a 32% improvement in the performance-energy ratio of data-intensive applications, thanks to the effectiveness of the allocation of heterogeneous resources in a cluster.

3.4 FPGA Pooling Mechanisms

In addition to FPGA sharing (in single or multi-node settings), some works in the State Of Art deal with the concept of resource *pooling*, in which multiple devices are seen as a single device by the application. In this way, the applications are not aware of the nature (in terms of size and number) of the underlying devices, and a more scalable service can be created without explicit care by the application developer.

The work by *Pogliani et al.* [23] proposes a dynamic resource manager to automatically assign reconfigurable devices (FPGA-based Dataflow Engines (DFEs)) to jobs with the goal of meeting application level QoS, in terms of deadline or throughput. The proposed system is composed by a *Job controller* which computes the individual job requests and constraints, and a centralized *Resource Broker* which computes the allocation based on the given request, constraints and available DFEs. Two system implements two scheduling policies: an *Earliest Deadline First* policy for offline scheduling, and an online *Throughput-based* policy based on the number of tasks performed by the job. Moreover,

the system presents a group of DFEs as a single virtualized DFE backed by a pool of one or more physical devices, thus implementing a *pooling* technique.

A similar work is represented by *FPGA Groups* [20] by *Iordache et al.* FPGA Groups is a system to aggregate multiple reconfigurable components (DFEs as in the previous work) with the same configuration as a single accelerator. Each group defines a different accelerator, and the kernels run on accelerators of the same group through *time-sharing*. The proposed system also performs autoscaling of the groups by first monitoring each device task queue, then estimating the runtime performance of the current kernels, and finally resizing the queues accordingly. As in [23], each group of devices is seen as a single virtualized accelerator by the application, so it implements resources pooling. However, the group configuration is static (the board-accelerator mapping is done at the beginning of the jobs allocation) and is not integrated with the current orchestration systems (even though it includes an orchestrator).

The work in *Zhu et al.* [25] proposes an FPGA Pooling system for offering FPGAs to the requesting VMs in the cloud. The system works by allocating an FPGA portion only when requested by a VM and releasing it afterwards, thus sharing the FPGA and reducing waste. Moreover, all FPGA accelerators are managed as a single resource *pool* shared among all VMs. To efficiently allocate the different reconfigurable portions, the authors implemented a central scheduler which employs two different algorithms (and a mix of the two). The first is a *resource-aware* algorithm, which categorizes the jobs based on their location and tries to avoid the creation of "hot-spots" inside the system, in terms of FPGA-available nodes and network connections. The second algorithm is *workload-aware*, as it employs multiple priority queues and allocates the workloads in their reverse order of size (following a Shortest Job First (SJF) policy). By conducting an extensive evaluation on a small testbed and a large-scale simulation, the authors conclude that the system is able to improve the job completion time by up to 7×, and 95% tail job completion time by 4×.

Another work related to FPGA pooling and scheduling is *Zhao et al.* [26], in which the authors describe scheduling policies to *"minimize the make-span of a given batch of requests"* employing FPGAs. In particular, they consider two major resource bottlenecks, computation and network, and propose different strategies to assign job requests to the FPGA pool. For the *compute-intensive* workloads, the algorithm runs a linear programming model and splits the workload accordingly, as the network bandwidth needed to perform the split is negligible. For the *network-intensive* case instead the algorithm determine (through a linear programming model) the maximum bandwidth available to each job portion, and allocate the jobs based on the processing nodes *ingress bandwidth*. Finally, a third algorithm is proposed which considers both transmission and computation. The authors evaluated the system on a real testbed and in a simulator to simulate differ-

ent workload and network patterns, showing a decreased make-span w.r.t SJF scheduling.

Finally, *Okija et al.* [27] proposes a high-performance architecture for FPGA-as-a-Microservice (FaaM) to allow decoupling between accelerators and microservices. The architecture is composed of *Workers* to manage each distinct accelerator. In addition, *Service Containers* perform load balancing and decouple the FPGA execution from the access to the service. The Accelerator Manager is responsible for data movement and controlling the kernels' execution on the single FPGA, while a node-level scheduler (acting with a first come first serve basis) schedules the different threads on the accelerator. The authors evaluate the system by deploying a compression service, showing minimum overhead w.r.t the native execution (thanks to the use of nonblocking buffer tranfers) and higher performance than a CPU implementation. However, the system is only compatible with JVM-based frameworks and lacks a cluster-level orchestrator (at the time of writing).

3.5 Closing remarks

In this chapter we presented the State of the Art which is of interest for this thesis work. In particular, we first showed a classification of the works considered, distinguishing them based on the *communication method* used to access the device, the *sharing mechanism* implemented and their *batch* or *service* nature.

We then analyzed three different approaches to the problem addressed. First, we described the single-node approach to sharing an FPGA device, based on *virtualization* and shared memory. Then, we presented approaches to share multiple FPGAs in a cloud scenario using distributed services and schedulers. Finally, we listed the latest works offering FPGA *pooling*, in which the devices are shared among multiple applications on multiple nodes and are abstracted as a single device.

■

Chapter 4

System Design

In this chapter we are going to describe the overall design of the proposed system for FPGA sharing in cloud environments. We will also provide a brief description of each component included in the system.

Section 4.1 provides an high-level overview of the proposed design. The chapter then describes each component and its working principles. In particular, Section 4.2 describes the Remote OpenCL Library, Section 4.3 shows the Device Manager component and explains the basic FPGA sharing mechanism. Section 4.4 gives an overview of the Accelerators Registry component.

4.1 BlastFunction Overview

BlastFunction is an FPGA sharing system for the acceleration of microservices and serverless applications in cloud environments. The system allows multiple applications to execute kernels on the same FPGA concurrently and without changing the underlying host code, providing a easy integration with existing applications. In addition, the system allocates the available devices as required by the applications based on runtime metrics collected by the system itself. This allocation is performed to utilize the accelerators at their maximum performance and capacity (in terms of time-slots in which it is executing tasks), without impacting on the applications performance.

For these reasons, the main goals of BlastFunction are:

1. To provide a sharing mechanism for FPGA accelerators, allowing concurrent access to the board from different applications,

2. To automatically manage the scheduling and allocation of devices and application containers in the cluster based on performance metrics,

3. To provide an easy integration of the system with existing applications and serverless and container-based systems, without changing the host or the hardware code provided by the application developer.

The goals previously defined are tackled by the combination of multiple components that we designed to create the proposed system. The components are the *Remote Library*,

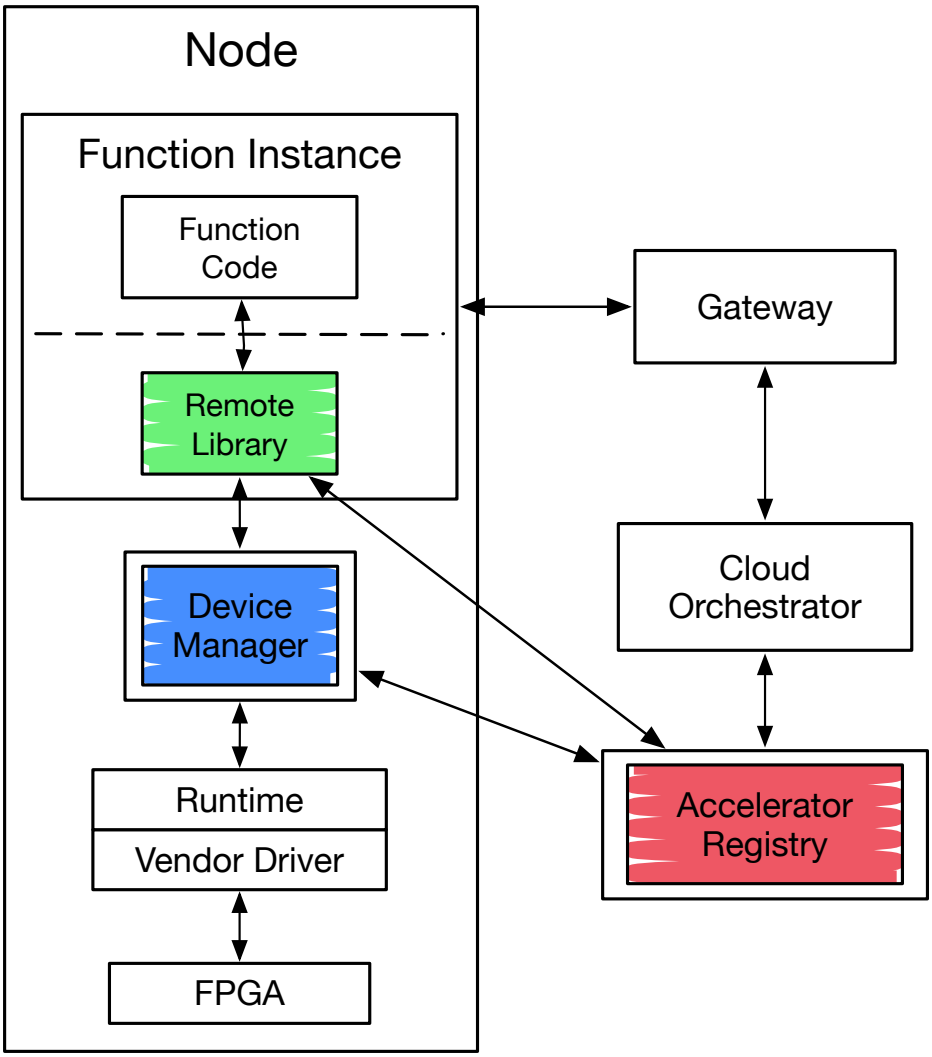

Figure 4.1: High Level Overview of BlastFunction components and their connections

the *Device Manager* and the *Accelerators Registry*, as shown in Figure 4.1.

The *Remote OpenCL Library* is the component of the system which allows the client application (or serverless function) to integrate with BlastFunction. The library is a custom OpenCL implementation which abstracts the use of the remote device access protocol and the communication to the *Accelerators Registry* and the *Device Managers* from the host code. In this way, the application can request and access an FPGA device as if it was on the host system (or VM), without caring about the underlying remote and shared access mechanisms.

The *Device Manager* is the component which is connected to each underlying FPGA in the system and provides the time-sharing mechanism. It exposes a service which is used to access the device functionalities remotely and without crossing memory boundaries on the system, in this way providing an isolated and secured access from multiple application

containers.

Finally, the *Accelerators Registry* is the central controller of the system, which tackles the goal of allocating the available devices efficiently using runtime performance metrics. It does so by tracking the device utilization metrics from the Device Managers and performing an online device allocation algorithm which takes also care of reconfiguring the devices at runtime. Finally, it intercepts the deployment and removal of applications inside the cluster to integrate them with the system and perform the allocation algorithm.

The system integrates with other external components in order to reach its goals. In particular, the *Cloud Orchestrator* depicted in Figure 4.1 (Kubernetes in our case) is used by the Accelerators Registry to control the cluster resources and their allocation to specific nodes. The *Gateway* is instead the serverless system's endpoint, which forwards external requests to the function instances and performs the autoscaling of functions.

4.2 Remote OpenCL Library

The *Remote OpenCL Library* is a custom implementation of an OpenCL host library, developed to integrate the applications with our system and to isolate them from the hardware accelerator execution. In particular, the *Remote Library* implements most of the methods used to control an FPGA accelerator. In addition, it can be easily integrated with the applications by overwriting the already installed library, or by updating the system OpenCL configuration to recognize it. If the application is linked dynamically, this allows to integrate it with our infrastructure without changing any line of code. Otherwise, the library can be directly used even with statically compiled applications.

The Remote Library implements a central router component, which acts as a singleton and it is responsible for keeping a list of the available platforms. In particular, it gets the address of the selected Device Manager (or managers if multiple addresses are provided as an environment variable) and creates a connection to it through gRPC. Also, for state-changing operations (any function call which does not just get information) the remote library calls the registration method on the manager to ensure that the connection is established and that the manager has reserved the resources for the client. After having checked the registration, the proper method is called based on the upper-level OpenCL call by the application.

The system allows for both synchronous/blocking requests to the remote runtime and asynchronous/non-blocking requests. Both the synchronous and asynchronous flows rely on *asynchronous events*. An event in the system is composed by a set of subsequent asynchronous calls to the device manager service, a state machine to control the steps that each event must follow and an OpenCL status for the event which is updated while the event is processed. In this way, the remote library supports event polling (e.g. clWaitForEvents, clGetEventInfo) like the standard OpenCL specification. We will explain the asynchronous flow by following Figure 4.2, which shows the main components of the Remote Library.

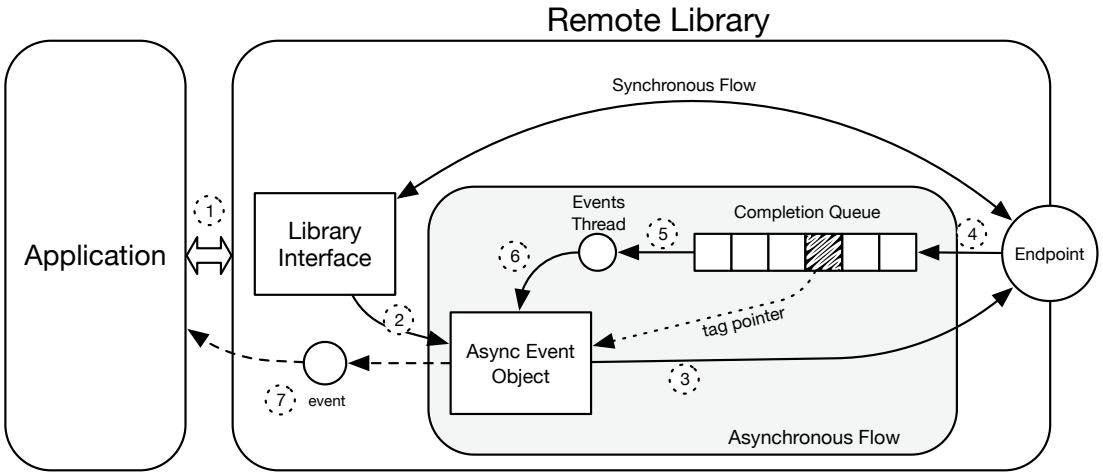

Figure 4.2: OpenCL Remote Library Architecture, highlighting the steps performed in the asynchronous flow.

When the Remote library receives an asynchronous OpenCL call from the application (step 1), it creates an event (step 2) and performs a first asynchronous request through the network stack (step 3). The request encapsulates a *tag*, which is the pointer to the newly created event. Whenever the device manager responds (step 4), the network runtime pushes the tag into the completion queue of the client. Then, the connection thread pulls the tag and retrieves the corresponding event (step 5). The thread calls the event state machine, which performs the needed operations and updates its state and the OpenCL event status (step 6). Finally, the application is notified when the event changes the OpenCL status (step 7). For example, to perform the clEnqueueReadBuffer function, the event's state machine contains 4 states, as shown in Figure 4.3. The INIT state sends the call metadata (buffer size, buffer id, offset); the FIRST step waits for the command to be enqueued by the manager; the BUFFER step actually sends the buffer data when the manager is available, and the COMPLETE step signals the call completion.

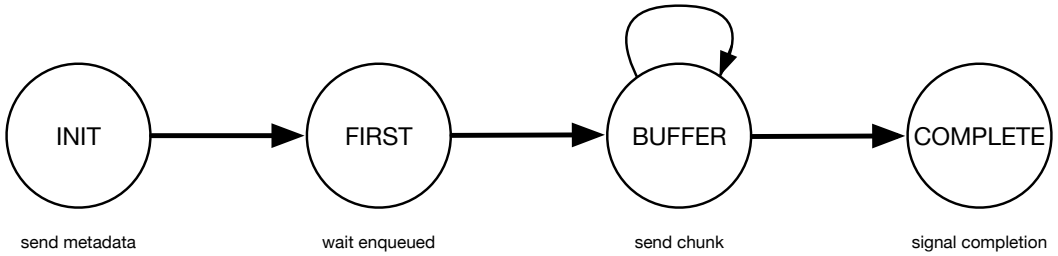

Figure 4.3: Example state machine for the read buffer operation

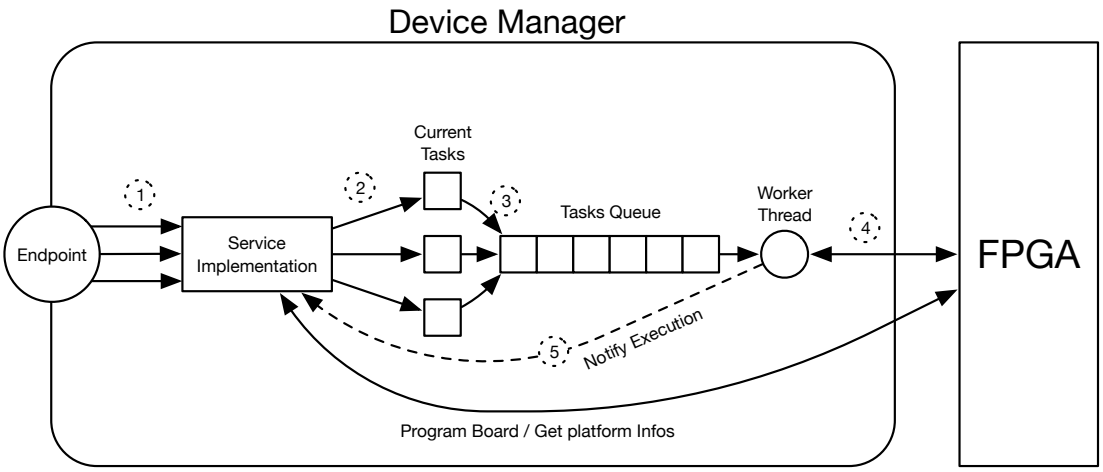

Figure 4.4: Device Manager Architecture, with the command queue methods flow highlighted

4.3 Device Manager

The *Device Manager* is the BlastFunction component responsible for the control and management of a single board inside the system. In particular, by exposing a service to access the FPGA concurrently, it's the basic block of the **sharing mechanism** presented in this work.

Figure 4.4 shows the key features and components inside the Device Manager, along with the flow followed by operations which are executed on the FPGA. The first component of the manager is the service endpoint, which receives all the method calls performed by the clients. There are two kind of methods exposed by the service: *context and information methods* and *command queue methods*.

The *context and information methods* are executed synchronously as they don't involve execution on the FPGA. This group of requests includes the creation (on the client side) of kernels, platforms and contexts, the request of informations related to the device and the buffer management requests. Regarding the creation of kernels and buffers management, the *Device Manager* controls each client resources pool separately, in order to enforce isolation between multiple clients. The board reconfiguration request represents an exception in this group, as it blocks the execution of other operations to reprogram the board with the given bitstream. We will describe in detail the reconfiguration while showing the *Accelerators Registry* in Section 4.4. All the other methods (platform, device, program and arguments information gathering, release of objects) don't involve the FPGA directly and can be executed synchronously.

The other group of requests is represented by *command-queue requests*, which are composed by operations that must be executed in the order decided by the client application and might require to use the FPGA exclusively. An example is the kernel execution request, which might be interleaved with buffers reads and writes on one or multiple queues. For this kind of requests, if any operation is received or executed in the

wrong order by the Device Manager, the results of the execution will change, and this would break the system consistency from the application point of view.

To ensure the in-order execution of *command-queue* requests, the *Device Manager* employs *multi-operation tasks*. We define a *task* as the atomic unit of execution of BlastFunction, composed of a sequence of operations that should executed atomically on the FPGA. Whenever the Device Manager receives a command-queue request (step 1), the requested operations are added to the task related to that particular client (step 2). After that, if the client sends a flush command (either by calling a blocking method or clFinish/Flush/EnqueueBarrier), the current task is sent to the central queue of the manager (step 3). This mechanism avoids conflicts between different clients and in the future could be used to lower the resource usage on the board (e.g. memory buffers can be deallocated while not used or when the data has to be overwritten by the next task). Once the task arrives to the central queue of the manager, a worker thread pulls and executes them on the FPGA in a First-In-First-Out order (step 4). Each operation in the task is linked with a OpenCL event and when the operation is completed the event is notified to the caller (step 5). In this way, the client is notified punctually, even if the operations are executed in groups.

4.4 Accelerators Registry

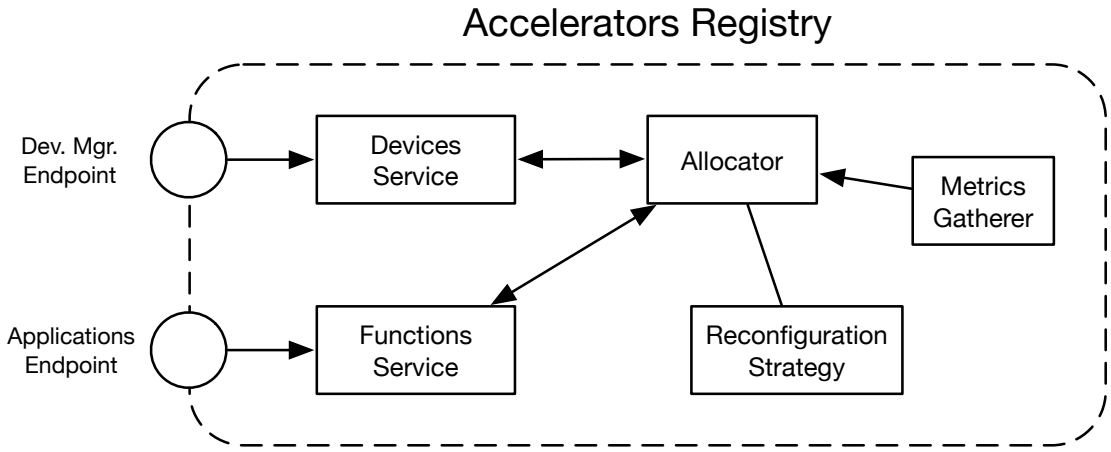

Figure 4.5: Accelerators Registry internal architecture

This section describes the *Accelerators Registry* component, which role is to gather information about the entire FPGA sharing system and to perform the admission control of the apps and the allocation of the devices to the deployed applications. We first give an overview of the Registry features and component. Then, Section 4.4.1 describes the devices allocation algorithm. Finally, we explain the reconfiguration flow coordinated by the Registry in Section 4.4.2.

The *Accelerators Registry* is the master component of the system, as it is the "bridge"

between the applications and the devices managed by BlastFunction. In fact, when an application is deployed, the *Remote Library* integrated with the application code asks to the Registry which devices to connect. In a similar way, whenever a *Device Manager* starts working, it notifies the Registry about the managed device. In this way, the Registry is able to track all the informations about the system which are required to perform an optimal allocation of devices to applications and function instances.

In particular, the features of the *Accelerators Registry* are:

Devices and Functions Registration The registry allows Device Managers and Applications (either containerized or implemented as serverless functions) to register their presence and information. Moreover, it is notified of their disconnection from the system to remove their information.

Devices Allocation Every time a new function or application is deployed in the system, the Registry allocates a variable number of devices to it (based on the application requirements). The allocation algorithm is explained in detail in Section 4.4.1.

Reconfiguration Validation Every time a Device Manager is asked to reconfigure its device, the Registry is notified and is able to validate the action or to block it. The reconfiguration flow is described in Section 4.4.2.

Metrics Aggregation The Registry tracks multiple runtime metrics about the system periodically, in order to enhance the allocation and reconfiguration mechanisms based on the current resources utilization and load.

We offer each functionality using different components inside the registry, as shown in Figure 4.5. The *Accelerators Registry* offers two main endpoints which external components can access: one endpoint is *device-related*, while the other is *application-related*. The two endpoints are backed by two different services which are responsible for the domain-specific data about the managed resources. The *Devices Service* collects and manage informations about the devices into the system. In particular, the informations stored about the devices are:

- Device identifier in the system,
- Device *OpenCL* Platform, Profile and Board Vendor,
- Position, in terms of cluster-level node name and *Device Manager* address,
- Current configured accelerator (name and bitstream hash),
- Function instances connected to it (references).

The *Functions Service* contains instead the informations about the different deployments and instances in the BlastFunction system. In particular, for *deployment* we mean the general specification of the application which is deployed in the system. An instance

is instead the single container or function replica of the application. Given that the allocation algorithm works with instance-level resolution, the Registry stores both the general specification of an application and the single instance information. The information stored in the Registry about the function specifications and instances are:

- Application/Deployment identifier in the system,
- Requested Device *Query* (e.g. board name or vendor-based filter, accelerator name/hash),
- Number of devices per instance,
- For each instance:
  - Instance container hostname,
  - Instance position in the cluster (node hostname),
  - List of assigned devices.

In general, the information contained in the *Device and Function service* components are updated from two channels: the external endpoints offered by the Registry and the Allocator component. When starting, both Device Managers and Function Instances register to the Registry through the endpoint, and remove their entry when they are disconnecting from the system (e.g. failing or re-deployed). The Allocator component also updates them when operating, such as on every new allocation.

The Accelerators Registry also tracks runtime metrics about the system, offering the Metrics Aggregation functionality through the *Metrics Gatherer* component. The Metrics Gatherer component connects to an external time-series database (e.g. Prometheus, as will be further described in Chapter 5) and fetches different metrics based on the queries coded into the component itself. When fetched, the resulting metrics are embedded in the device-related informations, and used by the Allocator component to enhance the resulting mapping between function instances and devices.

4.4.1 Allocation algorithm

The Devices Allocation functionality of the *Accelerators Registry* allows the system to automatically decide the how to connect Function Instances (each one requesting a device with a given configuration) and Device Managers (each one offering a time-shared device). To offer this functionality, the Registry performs an *online* allocation algorithm, meaning that it is not performed statically at startup, but it is performed whenever a new function instance is deployed. Also, the allocation depends on the runtime system configuration, in terms of already allocated accelerators, applications requests and runtime performance and utilization metrics gathered by the Registry.

The allocation mechanism tries to meet the following requirements:

- Optimize devices utilization in terms of time slices executed on each device by multiple services
- Avoid over-provisioning of the available resources, in order to not increase the services latencies
- Take into account the presence of multiple accelerators and bitstream (and allow the application to insert a new bitstream into the system if needed)

Algorithm 1 Devices allocation algorithm

```
1:  procedure ALLOCATE(instance, devices, metrics_order, metrics_filters)
2:      devices ← filterby_compatibility(devices, instance.devicequery)
3:      devices ← filterby_metrics(devices, metrics_filters)
4:      devices ← orderby_metrics_and_accelerator(devices, metrics_order)
5:
6:      i ← 0
7:      if not_compatible(devices(i)) then
8:          while not_redistributable(devices(i)) do
9:              i ← i + 1
10:
11:     if i < len(devices) then
12:         chosen_device ← devices(i)
13:     else
14:         raise error "device not found"
15:
16:     instance.devices ← {chosen_device}
17:     if instance.node == "" then
18:         instance.node ← chosen_device.node
19:     return
```

The allocation mechanism pseudocode is shown in Algorithm 1. Whenever the allocation procedure is called by the Accelerators Registry, it takes as input the function instance which must be matched, all the available devices in the system and a list of metrics to be taken into account. First, the procedure filters the devices based on their compatibility with the application requests. In particular, a device is *compatible* with the function instance if it complies with the function's device query. The *Device Query* embeds conditions about the device characteristics, such as vendor, OpenCL platform and profile, and the current configured bitstream or accelerator (either in form of hash or name). The most important parts of the Device Query from the allocation point of view are the vendor, platform and accelerator informations: vendor and platform are required to check if the board can be reconfigured by the application, while the accelerator hash is useful to check if the device has already been configured with the application-specific bit-

stream, thus not requiring a new reconfiguration for sharing it. The first filtering phase is performed based on the board and vendor information to remove from the allocation list the devices which can not be used by the application because they use a different underlying hardware.

Then, the devices are filtered again, this time based on the gathered or computed metrics. This is an additional functionality which is useful whenever the system designer (which can update the registry configuration) may want to keep some metrics in a specified range. For example, the utilization may be kept between 0% and 90%, in order to avoid overprovisioning of the device, or to take into account a possible variance in the device load by the sharing function instances. In general, if no device remains after any filtering phase, the Registry raises an error which is sent to the caller (application or orchestrator), and the function instance will not be connected to any device.

After all the filtering phases, the devices are sorted by metrics and by accelerator compatibility. The metrics priority can be chosen by the system maintainer, based on the system and applications SLA The *accelerator compatibility* refers to checking the current configured bitstream against the *Device Query* bitstream to see if the chosen device should be reconfigured in case of allocation. By employing this two ordering behaviors, the Registry can ensure an optimal and consistent allocation. For example, the default metrics sequence used by the Registry is $\{Utilization, Occupation\}$, meaning that the devices are first ordered by accelerators compatibility (default and first ordering), then time utilization and finally by basic occupation. In this way, the first compatible device with the lowest load (in terms of time-slices of execution) and occupation (number of instances connected) will be allocated to the instance.

It might happen that no compatible accelerator is found after ordering, meaning that no device in the cluster is configured with the required bitstream or has not been yet reconfigured. The algorithm notices this condition when the first device in the sorted list is not compatible (as the devices are sorted by compatibility first), then it tries to find a suitable device for *reconfiguration*. Starting from the sorted list of devices already found (the non compatible ones), the allocator checks which devices workloads can be redistributed to other compatible devices. In particular, the algorithm applies the following check to every device in the sorted list:

$$\exists D \in devices \mid \sum_{\substack{d \in devices, \\ d.hash = D.hash, \\ d \neq D}} (100\% - d.utilization) \leq D.utilization$$

If at least one device meets this requirement, then the device is flagged for reconfiguration and the registry allocates it to the requesting function instance. The process is approximated and takes into account only the total load for the reconfigured device, instead of the per-instance utilization. Thus, some reallocated function instances may not find enough time-slices on the new chosen device. However, in this case the system should react by scaling the function to other devices to solve the issue.

4.4.2 Reconfiguration Flow

The main feature of BlastFunction is that is leverages sharing FPGAs resources to accelerate cloud-native workloads. Using FPGAs in a distributed system poses different challenges w.r.t using conventional hardware such as CPUs. In fact, while CPUs provide a generic architecture able to process different operations and workloads without changing the underlying hardware, FPGAs must be reconfigured based on the target application, meaning that different workloads require different workload-specific accelerators and architectures.

The process of reconfiguring an FPGA board takes time to perform, and this time may impact on the execution of multiple applications. In addition, the new configured accelerator might not be compatible with the current applications using the device, which would need to be terminated or migrated to a different device in order to continue using the previous accelerator. In the context of this work, the system should then try to reconfigure the devices in an optimal way, meaning that reconfiguration should be performed the least number of times possible.

BlastFunction tackles these issues by relying on two mechanisms. The first mechanism has already been described in the previous Section (4.4.1), which deals with allocating already configured FPGAs to applications which require the same accelerator, or allocating them to different accelerators when the utilization or load of the current devices is too high. In this section, we describe the process performed by the components of the system when a function instance is connected to a device, which may be already configured with the required accelerator, or may need to be reconfigured.

Figure 4.6 describes the reconfiguration process flow from a system perspective, showing the different components involved and the messages that should be exchanged while processing the device reconfiguration. The first component involved is the *Remote OpenCL library* integrated with the function instance application code. When the application requests to reconfigure or program the remote device, the library sends a first request message to the *Device Manager* associated with the remote shared device. Figure 4.7 shows the flow of actions and choices made by the Device Manager regarding the reconfiguration flow. First, the Device Manager receives *hash* or any unique identifier of the required accelerator, then it checks if the reconfiguration can be performed. In particular, it first checks whether the device is already configured with the same accelerator bitstream or not. In case of a positive result, the device is already configured with the proper accelerator and there is no need for a reconfiguration. In the other case, the process requires to check whether the reconfiguration is valid in terms of the system configuration. This means that the Device Manager notifies the *Accelerators Registry* of the possible reconfiguration requested by the function instance. This notification is sent whether or not the reconfiguration is needed, in order to update the state of the system. As soon as it receives a notification, the Registry checks whether the function instance has been connected to the properly allocated device or not. If the allocation is verified (i.e. the device has actually been allocated to the requesting function instance), then

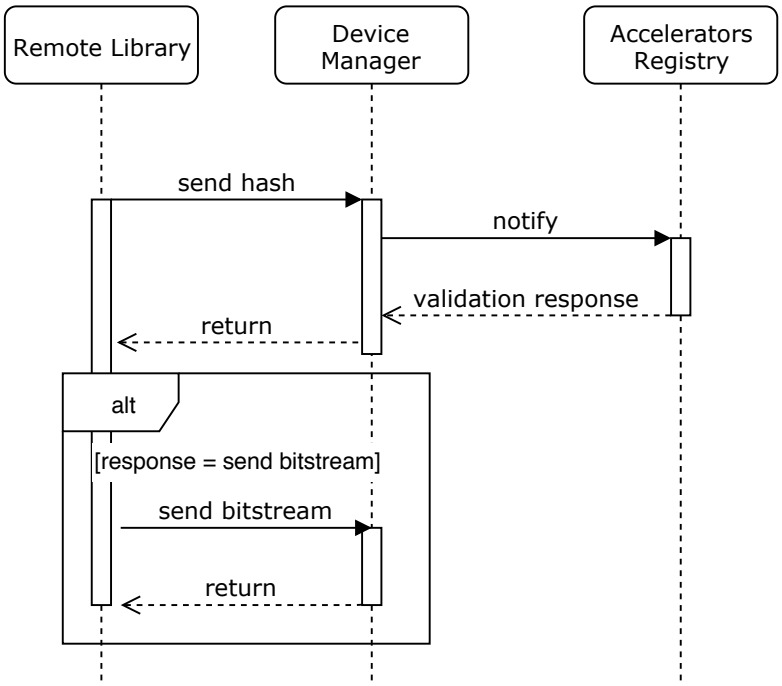

Figure 4.6: High-level view of the reconfiguration flow

the reconfiguration is valid and the Registry returns a positive response to the Device Manager. In addition, the Registry changes the system state by updating the allocation of already existing function instances connected to the device. In fact, if the new accelerator is different from the previous one, the current instances may not be compatible with it, so they must be migrated to a different device. This is performed by recreating the underlying application containers and allocating them to a different device with the proper configured accelerator. The process will be discussed in the next chapter as it is dependent on the cluster system implementation. After receiving the response from the Accelerators Registry, the Device Manager responds to the requesting function instance according to its current state. In particular, if the device is already configured with the requested accelerator it will stop the process; otherwise, it will ask the function instance to send the actual accelerator bitstream in order to reconfigure the device. The Remote Library receives the response from the Device Manager and acts accordingly: if the process ends it returns the control to the application host code (either with a success of failure code), otherwise it sends the accelerator bitstream. After receiving the bitstream the Device Manager needs to perform accessory operations in order to reset the status of the shared device: in particular, it flushes the tasks queue and stops receiving new operations regarding the previous accelerators from the Function Instances which are still connected (and will be stopped to perform their migration to a different device). Finally, the Device Manager performs the device reconfiguration and sends a final response to the Remote Library, which returns control to the application code.

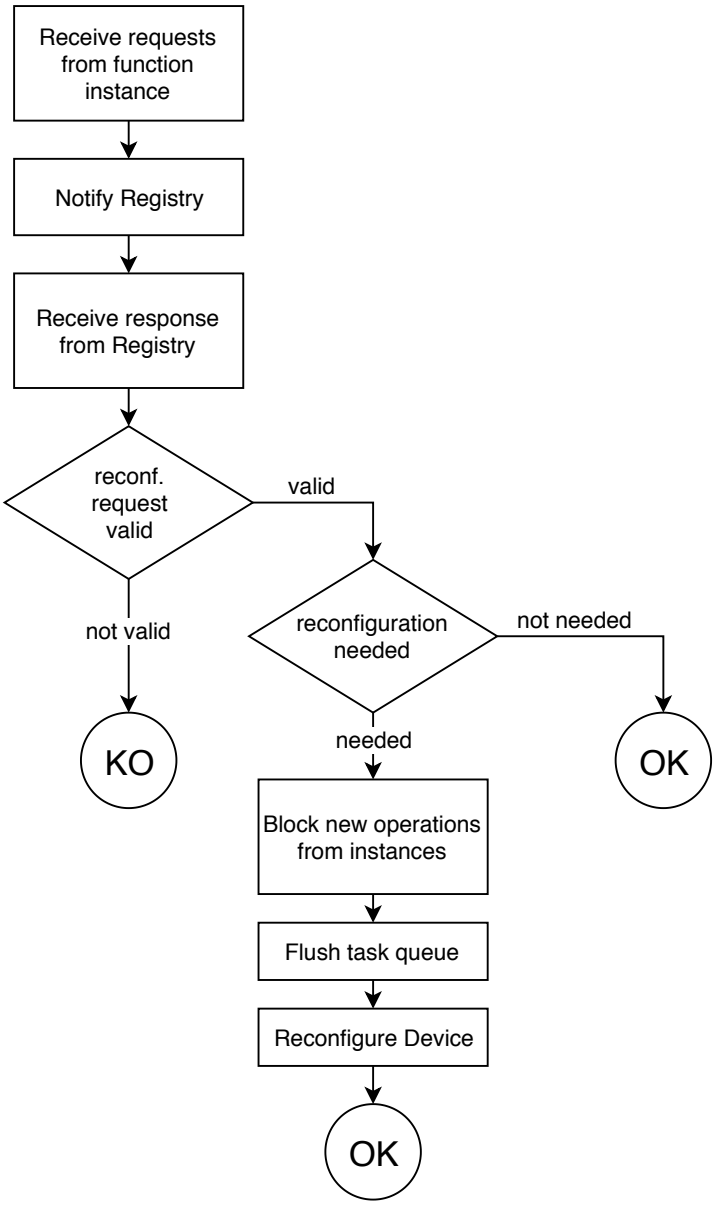

Figure 4.7: Reconfiguration Flow view from the Device Manager perspective

4.5 Closing remarks

In this chapter we described the design of the proposed system to tackle the FPGA sharing problem described in Section 2.4. BlastFunction is designed to offer a transparent and scalable system to share FPGAs among multiple tenant, providing a vendor-independent implementation.

The design includes 3 main components: a *Device Manager* for each device, a *Remote OpenCL Library* embedded in each Function Instance, and a central *Accelerator Registry*. The *Device Manager* provides the time-sharing mechanism to offer a single device to multiple instances, by receiving atomic tasks and executing them independently. The *Remote OpenCL Library* is a custom implementation of the OpenCL standard which allows the Function Instances to communicate with the system transparently and allows both synchronous and asynchronous operations. Finally, the *Accelerators Registry* gathers all the informations about the FPGA cluster from the Device Managers, receives the allocation requests from Function Instances and performs the allocation and admission control of them. The Registry employs an online allocation algorithm to match every Instance with the corresponding shared device, and takes care of managing the devices reconfiguration in accordance with the current state of the cluster.

■

Chapter 5

Implementation

In this chapter, we will show the implementation details of the proposed system, along with the description of the use cases integrated with it and its deployment. Section 5.1 explains the communication layer used by the BlastFunction system components. Section 5.2.2 shows the integrations that the Accelerators Registry employs with external services, and gives an overview of the final deployment of our system. Finally, Section 5.3 describes how we implemented the test case serverless functions, along with their packaging and deployment using OpenFaaS.

5.1 Communication layer implementation

In this section we describe the main communication mechanism employed by BlastFunction, highlighting the main features of the framework and protocol used. To implement a communication layer for our FPGA sharing system, some requirements should be met:

Low latency The protocol and framework employed should have a low overhead and provide low latency communication, in order to not impact on the execution time of the application.

Customizability / Extensibility The communication method should be customizable and extensible, in order to easily upgrade the system without losing compatibility with previous versions.

Standard interface On top of the other requirements, the framework should offer a standard interface for multiple languages, in order to not restrict it to only the system implementation languages (which in our case are C++ and Go).

We first show the gRPC-based communication system used for the majority of the operations. Then, we describe an improved mechanism to allow a fast buffer movement between the components, using shared memory segments.

5.1.1 gRPC-based communication system

Given the previously listed requirements, we found *gRPC* [35] to be a good fit for our system. gRPC is an open-source and high-performance Remote Procedure Call (RPC)

```
syntax = "proto3";
package opencl;

service OpenCL {

    rpc registerClient(ClientRegMsg) returns (ClientRegResp);
    rpc unregisterClient(ClientRegMsg) returns (ClientRegResp);

    rpc getPlatformInfo(PlInfoRequest) returns (PlInfoResponse);
    rpc getDeviceInfo(DevInfoRequest) returns (DevInfoResponse);

    rpc programWithHash(AccHash) returns (AccHashResponse);
    rpc programWithBinary(stream AccBinary) returns (AccResponse);

    rpc createBuffer(BufferRequest) returns (BufferResponse);
    rpc writeBuffer(stream WriteBuffRequest) returns (stream OCLResponse);
    rpc readBuffer(ReadBufRequest) returns (stream ReadBufResponse);
    rpc releaseMemObject(ReleaseMemRequest) returns (ReleaseMemResponse);

    rpc createKernel(KernelRequest) returns (OCLResponse);
    rpc getNumArgs(NumArgsRequest) returns (NumArgsResponse);
    rpc runKernel(RunKernelRequest) returns (stream RunKernelResponse);

    rpc flushCommandQueue(FlushRequest) returns (FlushResponse);

}
```

Listing 5.1: Main gRPC definitions for the communication between Remote Library and Device Manager

framework. The main characteristic of gRPC is that the protocol interface (in terms of RPCs and messages) can be defined using Protocol Buffers [36]. When the definition is given to the protobuf and grpc compilers, it generates a *stub* and a *service* implementation. The stub is a software component (generated with one of the languages supported such as Java, C/C++, Golang etc.) which can be easily integrated with an application to allow it to access the gRPC service. The service implementation sits instead on the server-side, and provides an interface that can be then implemented by the developer.

Listing 5.1 shows the protobuf definition of the OpenCL service, which the Remote Library and Device Manager use. Each RPC is defined by indicating an input and output message type (e.g. *ClientRegMsg* and *ClientRegResp* for the *registerClient* procedure) and the name of the method, which will be used by both the stub and the service interface. We also indicated some messages in the RPCs interface as *stream*. A stream allows the indicated part (client, server or both) to send and receive multiple messages during the execution of the RPC. This behaviour is useful when a procedure requires to send multiple data chunks, or when a more complex protocol (e.g. with multiple steps

```
message DataChunk {
    oneof payload {
         int32  totalSize = 1;
        bytes chunk = 2;
    }
}

message KernelArg {
     uint32  index = 1;
     uint32  size = 2;
    bool isMem = 3;
    oneof arg_type {
         int32  mem_id = 4;
        bytes arg = 5;
    }
}

message BufferHeader {
    string client_id = 1;
     uint32  queue_idx = 2;
     int32  mem_id = 3;
     int32  offset = 4;
    bool blocking = 5;
}

message WriteBufRequest {
    oneof payload {
        BufferHeader header = 1;
        DataChunk chunk = 2;
    }
}

message RunKernelRequest {
    string client_id = 1;
     uint32  queue_idx = 2;
    string kernel = 3;
     int32  dimensions = 4;
    repeated  uint32  offset = 5;
    repeated  uint32  global = 6;
    repeated  uint32  local = 7;
    repeated KernelArg karg = 8;
}
```

Listing 5.2: Example message definitions used in the Remote Library - Device Manager communication

and messages) is required. For example, the buffers-related RPCs use streaming messages to send the buffers to read/write.

Regarding the messages definition, Listing 5.2 shows some example messages that are exchanged in the system. A message is defined by its fields, which are numbered in order to keep the protocol retrocompatible (fields can be added, but it is better to not remove them afterwards from the definition). The protobuf compiler generates classes and methods based on the given definition, allowing a fast mechanism for serialization and deserialization. In addition to standard fields, protobuf allows to declare *oneof* conditions and *repeated* fields. A oneof definition (such as in the *DataChunk* message) allows only one field to be used at a time. This can be used in combination with streaming messages in order to enforce a given protocol. In our system, the oneof keyword is widely used, for example in the *DataChunk* and *WriteBufRequest* definitions. The *repeated* keyword indicates that a field may have multiple copies inside the same message. This is useful when sending a small array of elements (e.g. kernel arguments in the *RunKernelRequest* message). Finally, message definitions can be nested in order to create complex messages. We can see this feature in use in the *WriteBufRequest* and *RunKernelRequest* definitions.

Another gRPC characteristic that we leverage is the possibility to choose between *synchronous* and *asynchronous* calls, both for standard and streaming RPCs. When a synchronous RPC is performed, the underlying framework takes care of all the synchronization and thread patterns, without the intervention of the developer. In the *asynchronous* mode, instead, the developer must explicitly manage the *events* created by the framework. As already shown in the methodology chapter while describing the OpenCL Remote Library (Section 4.2), we make use of a *completion queue* which is part of the gRPC framework. Our implementation employs *tags* to recognize and process the incoming messages. A tag is a pointer to a specific event created in our system which then triggers an action using an embedded state machine. The asynchronous mode is needed on the client size to be OpenCL-compliant, as the original standard allows asynchronous operations. On the Device Manager side instead the service is implemented only in a synchronous way, in order to let gRPC manage the underlying thread pools.

5.1.2 Shared Memory mechanism for buffers movement

Even if it is a high performance communication framework, gRPC presents performance limitations related to the underlying serialization of Protobuffers. In fact, using the loopback interface (or the node-level docker virtual network) we noticed a throughput limit of roughly 1.3GB/s. This limit is confirmed by public benchmarking results for Protobuf shown in [37]. Moreover, the gRPC implementation of buffer-passing operations performs 3 transmissions of the same elements. For example, the `write` operation performs a first copy from the application buffer to gRPC messages, then a second copy to send the messages over the network to the Device Manager. The Device Manager copies the received messages to a local buffer, and finally sends the data to the device. All these copies represent a huge overhead in terms of operation latency.

In order to improve the performances, we decided to implement a shared memory mechanism in our communication system. The shared memory feature works in addition to the network (gRPC-based) mechanism for buffer passing, but only if the Function instance and the Device Manager are on the same node. By creating and using shared buffers instead of sending them through the network, we are able to avoid most of the copies, performing only one copy for each transfer (instead of 3 copies of the default mechanism).

The mechanism works by creating shared buffers through the use of the `open` and `mmap` POSIX primitives. When the client requests the creation of a buffer (through the `clCreateBuffer` function), the Remote Library checks if the server allows shared buffers and asks the Device Manager to create one. Then, the Device Manager creates a file in a directory shared with the application. Usually, the system tries to share the host's `/dev/shm` directory or one of its subdirectories among the applications and the Device Manager, as the `/dev/shm` mount point uses a memory-based volume. This allows to avoid the hard disk access overhead when manipulating the shared buffers. After creating the file, the Device Manager resizes it to the desired dimension (given by the requesting application) and uses `mmap` to map the file to a memory address, which will be used on the Device Manager side. The `mmap` function works by creating a virtual mapping between the file and the application, and allows different access modes for the mapped space. In particular, we use the `MAP_SHARED` flag to allow buffer visibility to the client application. Moreover, the mapping includes different flags (e.g. `PROT_READ` and `PROT_WRITE`) to protect the buffer from write or read operations, which is useful when mimicking the OpenCL flags for buffer access.

On the application side, the Remote Library receives a response from the Device Manager, containing the buffer id and the path of the created file (relative from the shared mount path). Then, the Remote Library tries to open and map the shared file and associates the mapped address to the created buffer, in order to use it in the buffer passing methods. When the applications requests a read or write operation (e.g. `clEnqueueReadBuffer`, `clEnqueueWriteBuffer`), the Remote Library starts the process in the same way as the gRPC-based mechanism, using asynchronous calls and an event-based state machine, using gRPC messages to advance the state of the operations. The difference when using shared memory is that the buffer passing method does not use gRPC messages to send/receive the buffer, but instantly uses the shared buffer in a single operation (performing a `memcpy`). The Device Manager also directly uses the shared buffer, passing it to the device-side OpenCL methods. The gRPC flow is maintained to synchronize the Remote Library and Device Manager through the use of ACKs, in order to avoid collision on the shared buffer.

The proposed mechanism also provides automatic deallocation procedures in case of failure to avoid memory leaks which could affect the host system. For example, if the client application can not open the shared file, or there is an error in the `clCreateBuffer` operation, the Device Manager will immediately delete the shared file and use the gRPC-

based mechanism.

Moreover, the shared memory mechanism is subject to compatibility of the host Operating System (OS), as it must allow the use of `mmap`. The Registry is responsible for providing the shared memory mount point and access permissions to both the Device Manager and the serverless instances. In addition, the Registry sets the environment variables related to the mechanism, activating it or not based on the host environment (more on this in Section 5.2.1).

Finally, we take into consideration the possibility of zero-copy buffer passing between the applications and the Device Manager. In fact, directly using the shared buffer for the client application would allow to avoid any copy apart from the host to device transmission, as every update to the buffer would be instantly visible to the Device Manager. Unfortunately, to allocate the shared buffer the client application would need to use a custom function which is not part of the OpenCL standard, breaking the transparency goal of our project. Moreover, the synchronization behaviour would presents a compatibility issue in the case of gRPC-based communication, as synchronizing the buffer transparently would introduce a high overhead (in terms of hash and differences computation, along with hidden buffer transfers over the network). For this reasons, we decided to not introduce a zero-copy mechanism using a custom allocation for the client buffers. In any case, we plan to introduce this feature as a custom extension of the library in future works.

5.2 System integration and Deployment

In this section, we show how the proposed system integrates with an existing cloud orchestrator (in this case, Kubernetes) to enable an easy and effortless deployment of applications using shared FPGAs.

5.2.1 Registry integrations

As already explained in Section 4.4, the *Accelerators Registry* is the main component of BlastFunction, responsible for the devices allocation and admission control of the client applications and function instances. Even if the Registry features should be agnostic in terms of cloud providers, we decided to provide a first implementation integrated with Kubernetes as a proof of concept. Moreover, we use Prometheus as our metrics timeseries database. Here we describe how the Registry communicates with the Kubernetes master API and a Prometheus deployment in order to offer its services.

Kubernetes WebHooks

The first feature requiring the Registry to integrate with Kubernetes is the registration and patching of function instances. When a function is deployed, we would like to know if it requires an accelerator, and in that case register it and assign it a unique iden-

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: example-deployment-blf
  annotations:
    blastfunction.io/vendor: "altera"
    blastfunction.io/device: "de5a_net_e1"
spec:
  template:
    metadata:
      labels:
        app: example
    spec:
      containers:
      - name: example-blf-application
        image: registry.blastfunction.io/exampleapp:v1
```

Listing 5.3: Example deployment highlighting the BlastFunction-related annotations

tifier. This can be done by using *WebHooks*. A Web Hook is a way for a service to notify a client application when a new event has occurred on the service. The concept of WebHook extends outside the context of Kubernetes to all web services, providing a callback mechanism using the HTTP protocol. For what concerns our work, we distinguish between two kind of WebHooks in Kubernetes, both related to resources admission control: *MutatingAdmissionWebHook* and *ValidatingAdmissionWebHook*. When a user or an application tries to create a new resource in the Kubernetes cluster, the master first calls all the associated Mutating Webhooks. Each application listening for this kind of callback can then send *patches* to the created resource in order to modify it before the creation. After all mutations have been performed to the resource definition, the Kubernetes master validate it through the Validation WebHook. In this way, multiple actors inside the system can decide which resources to admit and how to update them.

The Accelerators Registry listen a deployment and pod-level *MutatingAdmissionWebHook* inside the cluster to intercept the creation and removal of deployments (one for each function/application) and pods (for each function instance) and, if needed, update them. In particular, when a new deployment or pod creation request is intercepted, the Registry checks for some BlastFunction-related annotations inside the specification. The checked annotations are in the form `blastfunction.io/keyword`, with `keyword` being `accelerator`, `acceleratorhash`, `device` or also `vendor`, as shown in Listing 5.3. If any of the listed annotations is found, the Registry aggregates them as a unique *DeviceQuery*, which the Allocator uses to find the matching device. If a device is found, then the Registry issues a *patch* for the deployment. Listing 5.4 shows an example patch for a deployment, while Listing 5.5 shows a patch for a pod resource. The patch adds environment variables on the first container of the pod, in order for the Remote OpenCL

```
[
    {
        "op": "add",
        "path": "/spec/template/spec/containers/0/env/-",
        "value": {
            "name": "ROOT_CLIENT_ID",
            "value": "client0"
        }
    },
    {
        "op": "add",
        "path": "/spec/template/spec/containers/0/env/-",
        "value": {
            "name": "REGISTRY_ADDRESS",
            "value": "blfregistry.blastfunction.svc:50051"
        }
    }
]
```

Listing 5.4: Example Mutation patch issued by the Registry for a Deployment resource

```
[
    {
        "op": "add",
        "path": "/spec/containers/0/env/-",
        "value": {
            "name": "CLIENT_INSTANCE_ID",
            "value": "instance0"
        }
    },
    {
        "op": "add",
        "path": "/spec/nodeSelector",
        "value": {
            "kubernetes.io/hostname": "blfnode2"
        }
    }
]
```

Listing 5.5: Example Mutation patch issued by the Registry for a Pod resource

library to work. We do not add a volume for the actual library and assume that the container already includes it, but it would be a possible way to automatically load the Remote Library into application containers. The `REGISTRY_ADDRESS` variable indicates to the Remote Library how to connect to the Registry itself. The `ROOT_CLIENT_ID` and `CLIENT_INSTANCE_ID` are instead the unique identifiers of the function (and function instance) inside BlastFunction, and are then used by the Remote Library to authenticate within the system. Moreover, the patch mechanism is used to schedule a function instance on the specific node where the allocated device resides. This is done by adding a *nodeSelector* field to the pod specification (and not the deployment's), as shown in the example patch.

Kubernetes API integration

In addition to patching newly created instances, the Registry directly employs the Kubernetes API to manage existing function instance Pods. In particular, as already outlined in Section 4.4.2, whenever a device is reconfigured the Registry must reschedule the connected clients. Rescheduling of pods is not natively available in Kubernetes, so we implemented it in the Registry by employing a Golang client library for Kubernetes. When the registry receives a reconfiguration request for an already configured device, it checks whether the already connected instances would be compatible with the new accelerator. All the non-compatible instances are flagged for rescheduling, which is then performed by removing the associated Pods. When a Pod deletion request is sent to Kubernetes, it automatically creates a new Pod with the same specification before removing the previous one. This is done to maintain the service continuity and keep the application up. The Registry patches the new Pod in order to allocate a compatible device to it, considering also that the previous device has changed. In this way, the function instances can be migrated to a different device than the reconfigured one, without losing service uptime.

Prometheus integration

In order to allocate devices to function instances, the Accelerators Registry integrates with a central *Prometheus* server to gather system and devices-related *metrics*. Prometheus is an open-source *time series* processing and collection server, which works by collecting multiple metrics from endpoints inside the cluster. In particular, by *"scraping"* the endpoints inside the system for a specific Hypertext Transfer Protocol (HTTP) endpoint (usually `/metrics`), Prometheus collects the partial data collection (e.g. current sum of a variable or histogram) and stores every data series inside its database. Afterwards, applications and services can access and process the collected metrics by using *queries* in a specific language called *PromQL*. In BlastFunction, each Device Manager exposes a metrics endpoint, showing latencies, allocations and other information related to the underlying device and manager. The Registry periodically collects some of the exposed metrics by querying the central Prometheus server, and includes them in the allocator algorithm

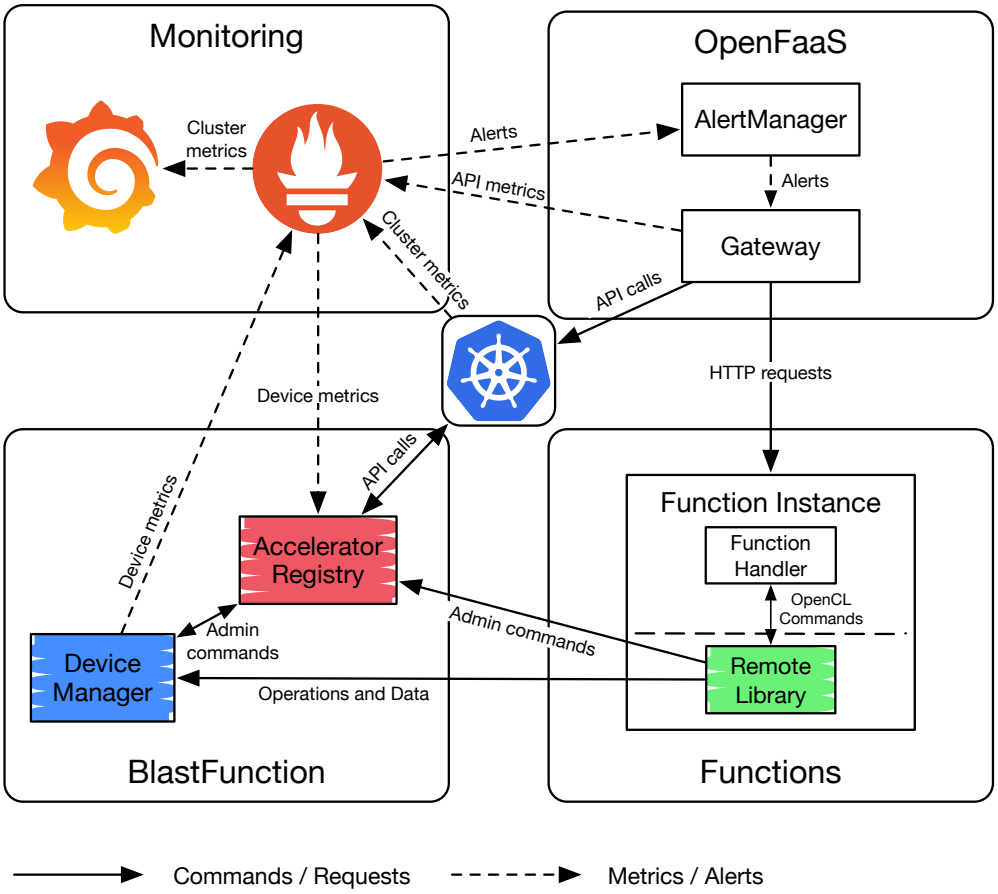

Figure 5.1: Overview of the system Deployment in a Kubernetes cluster, showing the main connections between Services and components.

(as described in Section 4.4.1). For example, the main metric used by the Allocator is called *FPGA utilization*, and it is computed with a prometheus query for each device in the system: `rate(fpga_task_latency_sum{app='device-manager'}[1m])`. This query instructs Prometheus to return a rate of change for the `fpga_task_latency_sum` variable in the last minute. In this way, the Registry obtains for each device the percentage of time (over the last minute) in which tasks were executed on the FPGA, giving a rough estimate of the device load.

5.2.2 Complete system Deployment

In this Section we describe the Deployment of BlastFunction and the related services and components inside a Kubernetes cluster.

Figure 5.1 shows an overview of the Deployments inside the system. The system is divided into four main namespaces, each one representing a different sub-system based on its functionalities.

The first Namespace is for *monitoring*. It contains the Prometheus and Grafana

deployments, which are used to pull metrics from all other components of the cluster and to query/show them. A standard Kubernetes Deployment is used to deploy both Prometheus and Grafana. Each component has an attached Volume in order to keep the metrics even if the Pods fail and to have a stateful behaviour. Moreover, each deployment is backed by a Service which exposes the application on all the nodes in the cluster. As shown in Figure 5.1 the Prometheus service receives metrics from multiple components: each Device Manager, the OpenFaaS gateway and the Kubernetes cluster itself, through the use of a node-exporter Pod running on every node. Moreover, Prometheus forwards metrics and alerts to other services, namely the Accelerators Registry and the OpenFaaS AlertManager.

Regarding the OpenFaaS namespace, we show only the Gateway and AlertManager deployments as they are its main components. Other deployments not shown are FaaS-idler (which scales function instances to zero when no requests are received), a streaming server and a QueueWorker for asyncronous requests. The OpenFaaS gateway is deployed either with one or multiple replicas depending on the load of the system, and it receives and forwards requests coming from the related service to the interested Function Instance. Moreover, it sends metrics related to the received requests to the Prometheus service. The AlertManager instead receives alerts from Prometheus and forwards them to the Gateway. Alerts are threshold related to particular queries, which are used by OpenFaaS to perform autoscaling decisions. Finally, the Gateway interacts with the Kubernetes API to create and update Deployments and Pods related to serverless functions.

The BlastFunction Namespace contains the Accelerators Registry and all the deployed Device Managers. Device Managers are deployed using a *DaemonSet* specification, which instantiates a Pod on every node. We label each node based on the installed FPGA board to deploy the proper Device Manager. In order to give the Manager informations about the node on which it is deployed, we add parametric values to the DaemonSet yaml file (e.g. `spec.nodeName` for the node and `status.podIP` for the Manager address). Moreover, we attach to the Manager container the `/dev` directory with privileged access to the FPGA device files, in order to give it access to the board. Regarding the interactions with the other parts of the system, each Device Manager pushes metrics to the Prometheus service and receives commands and data from the Registry and the connected Function Instances (through the Remote Library). The Registry instead pulls metrics from Prometheus, receives commands and requests from Device Managers and Function Instances, and interacts with the Kubernetes API in both ways, as explained in Section 5.2.1: it receives admission requests and sends rescheduling commands.

The last created Namespace is the Function's one (called `blastfunction-fn`), which acts as a simple container of Pods for the Function Instances created by the OpenFaaS gateway and patched/rescheduled by the Accelerators Registry. We created a separate Namespace in order to isolate the managed Pods and to give special permissions regarding them to the Accelerators Registry, OpenFaaS and Prometheus, which scans the namespace in case any of the Function would have a metrics endpoint exposed. To give

specific permissions we employ *Service Accounts* and *Roles* and *RoleBinding*. Each Role has associated operations and resources on which the operations can be performed. A role can then be bind to a given namespace or the entire cluster (for example, Prometheus has a cluster-wide `list` permission for Pods). Finally, a Service Account encapsulates multiple roles and can be associated with any Deployment or Service inside the cluster. For OpenFaaS and the Registry, we created two accounts and roles with similar permission, which are `get, list, watch, create, delete, update` of Pods and Deployments inside the functions namespace.

5.3 Use cases implementation

In this section we describe the kernels we selected to evaluate our system. We decided to find already existing hardware designs and host applications in the state of the art to demonstrate the transparency of our system w.r.t both the FPGA vendor and the host code. Moreover, we explain how each kernel was integrated in a serverless function and packaged using OpenFaaS.

5.3.1 Spector: Sobel and Matrix Multiplication

The first set of accelerators we found and integrated in the test applications belongs to the work of *Gautier et al.* [38] called *Spector*. Spector is an open-source OpenCL FPGA benchmark suite. It contains various tunable designs for the acceleration of algorithms through FPGAs. Moreover, the authors synthesized more than 8000 designs for the available kernels to provide a complete Design Space Exploration (DSE). This work is useful as it targets the same kind of FPGA board used in our evaluation phase and it provides an OpenCL implementation (both from the accelerator and the host point of view) of the applications, which allowed us to integrate the proposed designs easily in our system.

The fist design considered is the one performing a *sobel* edge detection algorithm (also called *sobel operator*). The algorithm works by applying a 3×3 kernel to an input image. This allows to compute an approximation of the derivatives for each point in the picture, then added together to compute the gradient *magnitude*. The magnitude indicates the level of *"change"* of the current pixel w.r.t neighboring pixels, such that if the value is high the pixel is considered as inside an edge. The authors in Spector employ *blocking* and internal *SIMD* operations to improve the algorithm performance on FPGA. In particular, a block of the input picture is first loaded into the kernel *local* memory (on BRAMs), then a sliding window is computed over the block. The parallelism is enhanced by employing multiple operators inside a compute unit, and by synthesizing multiple compute units on the board. By analyzing the DSE output data for this kernel, we ended up synthesizing a design using 32×8 blocks, 4×1 window with no SIMD applied and a single compute unit. In fact, the combination of this parameters result in the best performance in terms of latency, as the kernel is less memory-intensive. Moreover, the

lack of SIMD operations is balanced by the presence of a larger block and window, which are processed in parallel.

The second design we considered was a Matrix-Multiplication kernel, as it represents one of the most notable benchmarks for computing acceleration. In particular, the design given in Spector is based on the Altera OpenCL example. It allows to multiply two squared matrices of any size (with a maximum side length given by the resources available in the accelerator) by employing *tiling*, meaning that the resulting matrix is divided into blocks, each computed individually. Moreover, the design contains several knobs to tune the design to the specific board and FPGA used to obtain maximum performance. The knobs are the *block dimension*, the *subdimension* width and height (how many blocks are processed in a work-unit), number of compute units, and *SIMD* and *unrolling* parameters (enabling and factor). In our case, we analyzed the DSE results from [38] and found the best design with 1 compute unit, 8 work items for each unit, and a completely unrolled block of 16×16 elements.

5.3.2 PipeCNN: Neural Network acceleration

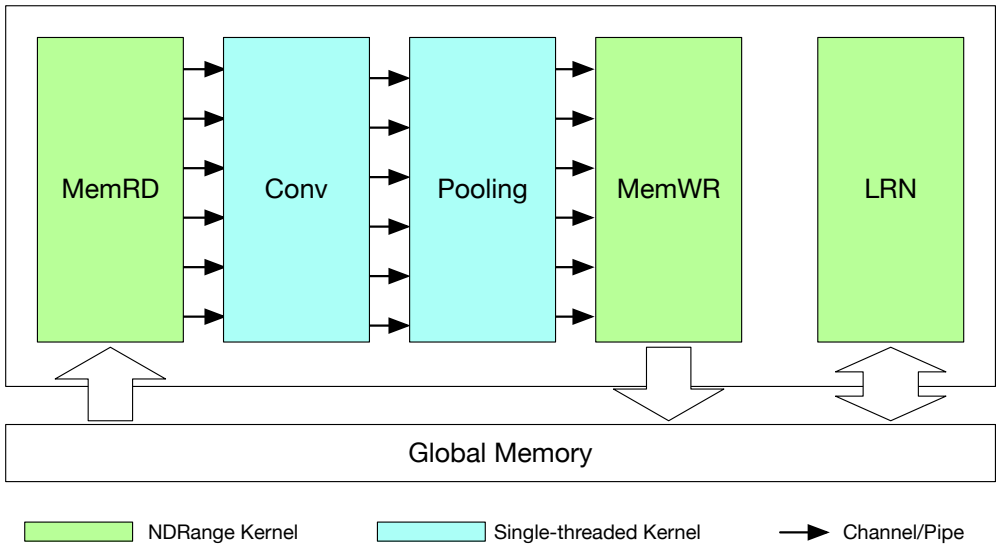

Figure 5.2: Top-level architecture of PipeCNN, showing the OpenCL kernels and their connection

Another work we used as a use case for our system is *PipeCNN* [39], which is an open-source implementation of an FPGA accelerator for Convolutional Neural Networks (CNNs). The accelerator allows to accelerate any kind of CNN using a set of pre-made OpenCL kernels for convolution/fully connected, pooling and normalization layers, which can be optimized with various parameters and options. Figure 5.2 shows the overall design structure of the accelerator, which is composed of multiple OpenCL kernels connected through *channels*. A channel is a connector between two kernels which allows to send and receive data in *streaming* fashion, meaning that the sender and receiver

exchange at most one data word for each clock cycle on the same channel. In this way, multiple cascaded kernels can run in parallel without using global memory, as each kernel reads from the previous one and writes to the next one. The two datamover kernels (MemRead and MemWrite) transfer feature and weight data from/to the global memory to the other kernels with high-throughput data streams. Moreover, cascading the Convolutional and Pooling kernels with the datamovers allow to reduce the inter-layer data which needs to be stored on global memory. The Convolution kernel is able to execute both convolutional and Fully Connected layers by employing a vectorized 3D Multiply-Accumulate (MAC) operation on multiple CUs (each one fed by a different channel coming from the MemRead datamover). In addition to convolution, other two kernels are implemented by the authors (Pooling and Linear Response Normalization (LRN)) in order to execute most of the network components on the FPGA board.

When the application wants to process an input image, it takes the network configuration in terms of number of layers and number of iterations to be executed for each layer. Then, the accelerator is called *iteratively*, meaning that each layer needs multiple runs to be processed. Moreover, all the kernels related to the current portion of the network are executed concurrently. For example, if the network definition requires convolution and pooling, the accelerator will enable 4 kernels: MemRead, Convolution, Pooling and MemRead. This represents a good test case for our system, as it uses multiple command queues in parallel and requires careful handling to not stall the entire accelerator. Moreover, given that the design is general enough to run multiple network at the same time, it represents a useful starting point to evaluate the benefits of accelerator sharing between multiple applications.

5.3.3 Integration and serverless implementation

Each of the previously described kernels was integrated into an OpenFaaS *serverless function*. In particular, in OpenFaaS a *function* can be created from any containerized application, by integrating it (and adding to the container) with the *Function Watchdog*. The Watchdog is a small Golang-based HTTP server which forwards external requests to the handler application, either by using the standard input/output (in the first version of the Watchdog) or HTTP forwarding. In this way, any application can be integrated with the serverless system easily and without custom APIs. Moreover, OpenFaaS provide pre-made *templates* to use for implementing a serverless function. A template includes the Watchdog and the basic code for the handler application (e.g. a small web server without a specific implementation) that can be filled by the developer.

In our case, we decided to use the *Dockerfile* template to implement from scratch our functions. In particular, starting from a standard CentOS containers we added our Remote Library in place of the already present OpenCL library. Then, we developed a basic HTTP handler application using the C++ REST framework called Pistache [40]. Finally, we packaged the application in a Docker Container by modifying the Dockerfile given by the OpenFaaS template, which adds the Watchdog binary and set up the related

```yaml
provider:
  name: openfaas
  gateway: http://blastfunction.io
functions:
  sobel:
    lang: dockerfile
    handler: ./sobel
    image: registry.gitlab.com/blastfunction/sobel:latest
    labels:
      com.openfaas.scale.min: 3
      com.openfaas.scale.max: 12
      com.openfaas.scale.factor: 10
    annotations:
      blastfunction.io/device: "de5a_net_e1"
    secrets:
      - gitlab-registry
```

Listing 5.6: Example stack.yaml configuration file for the sobel serverless function

environment variables.

In order to deploy a serverless function in a OpenFaaS-enabled cluster, the developer needs to create a yaml configuration file for that function. Listing 5.6 shows an example yaml file for the sobel function. The yaml file may contain multiple functions, and for each function it defines which container to use, the handler application and other variables. For example, the `com.openfaas.scale.*` labels are used by the OpenFaaS autoscaling mechanism. The function is first deployed with the `min` number of replicas. When the incoming requests rate surpasses a threshold, the function is progressively scaled by a `factor` ratio until it reaches the `max` number of instances. We also added other variables to the function definition file, such as the BlastFunction-related annotations and the name of the secret to access the docker registry.

When the function is deployed, it can be accessed afterwards through the OpenFaaS gateway address. For example, for the given sobel function definition the gateway is located at `http://blastfunction.io`, thus the function can be called by sending an HTTP request of any kind at `http://blastfunction.io/function/sobel`. This URL acts as the root address for the function, so if `http://blastfunction.io/function/sobel/apply` is called, the function handler will see `/apply` as the requested URL, as the Watchdog will strip the address portion related to the OpenFaaS gateway.

■

Chapter 6

Experimental results

In this chapter we will present and discuss the results of the experiments conducted to validate the proposed system design and implementation. In particular, we tested two different scenarios using the applications described in Section 5.3. Section 6.1 describe the first scenario, in which we measure the single-node *overhead* introduced by using our Remote OpenCL Library. Section 6.2 illustrates and the second scenario, which tests the system scalability in terms of accelerators usage in a small-scale experimental setup.

6.1 System Overhead Evaluation

In this section we describe and show the results of our first experiments concerning the system *overhead*. The goal of this first experimental campaign is to verify whether the proposed FPGA sharing system is introducing a limited overhead w.r.t a native execution. Considering a serverless scenario, the native execution represents the theoretical maximum performance that our system should achieve. Evaluating the system overhead in a controlled scenario allows to get a first measure of how BlastFunction performs, which we also use to produce estimates and assumptions for the other experiments.

6.1.1 Experimental Setup

To test our system, we run our experiments on a single node. The node is equipped with a single socket 3.40GHz Intel® Core™ i7-6700 CPU, with 8 total threads (4 cores) and 32GB of DDR4 RAM. The node is connected to the local network through a 1Gb/s Ethernet link. Moreover, we installed a Terasic DE5a-Net FPGA board on the node. The board includes an Intel®Arria 10 GX FPGA (1150K logic elements) along with 8GB RAM over 2 DDR2 SODIMM sockets and a PCI-Express x8 connector.

We run each test by deploying a single instance of the Device Manager using a Docker Container connected to the underlying FPGA board. Moreover, the tested applications were deployed on a different Docker Container on the same node, in order to have a fair evaluation between our system and the native execution mode (local virtual network stack + Shared Memory and PCI Express for our system, PCI Express only for the native FPGA execution).

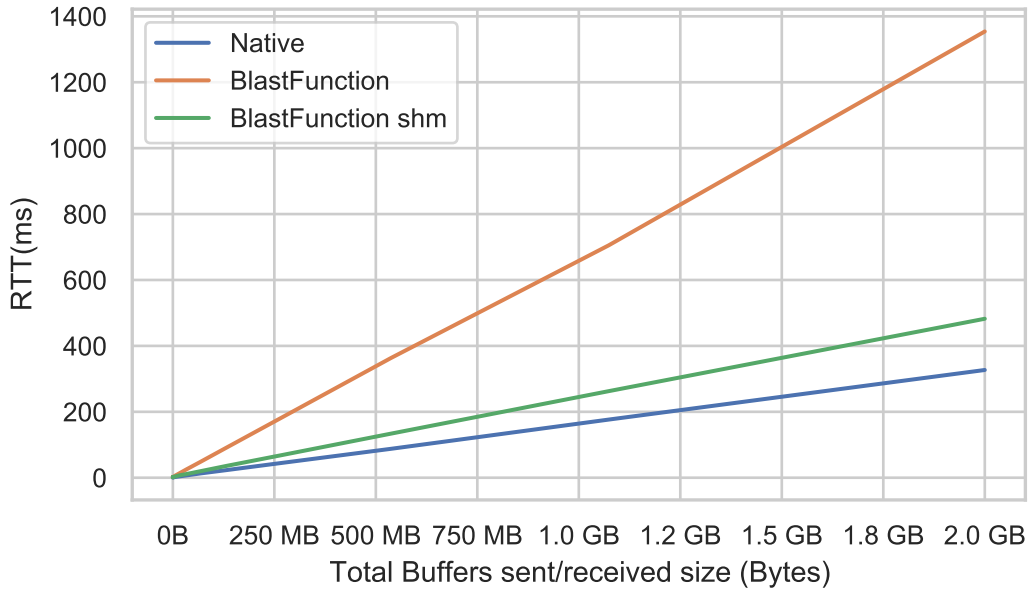

Figure 6.1: Latency overhead for read and write operations at increasing input and output sizes.

6.1.2 Overhead Evaluation Results

We tested the three accelerators described in Section 5.3 (Sobel, MM and VGG using PipeCNN) by copying each kernel specific host code in a test application written in C++ and linked dynamically with our *Remote OpenCL Library*. Moreover, we implemented a simple host application which reads and writes to/from the device in order to evaluate the communication overhead, without taking into account the kernel execution time. Each test application was deployed as a Docker container on the same node as the Device Manager.

To test the latency overhead, we run each test by increasing the input and output size to see the impact of the Remote Library serialization and communication mechanism (both gRPC-based and shared memory based) and the Device Manager queue. We tested each different size step 40 times to aggregate the results and get a better estimate of the average latency. Moreover, the accelerator calls are spaced by 200 milliseconds each, in order to get an independent measure of the execution without the interference given by multiple subsequent calls. Unfortunately, we were not able to change the input size to the CNN accelerator, as the implemented network has a fixed-size input.

We first discuss the results regarding the pure Read/Write performance. Figure 6.1 shows the Round-Trip Time (RTT) for a write-read operation (first write, then read synchronously). We tested the operation with an increasing buffer size, from 1KB to 2GB. To evaluate the performances, we measured the RTT latency for the native implementation and our system, testing both the gRPC-based and shared memory buffer

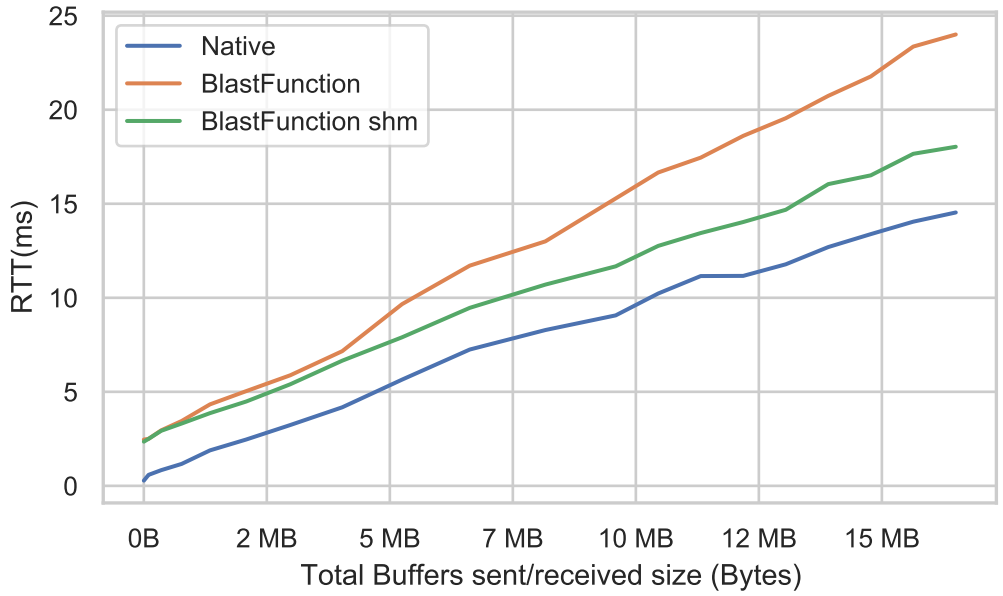

Figure 6.2: Latency overhead for Sobel operator accelerator at increasing input and output sizes.

passing mechanisms. In the gRPC-based implementation (denoted by the "BlastFunction" label in Figure 6.1), the total latency is more than four times the native execution time. For example, at 2GBs of transferred buffer size we measured a 1027ms overhead, which corresponds to 314% of the native execution time. This overhead is given by the serialization limit imposed by Protobuf and because the communication system performs 3 more copies of the same buffer (as explained in Section 5.1.2). The Shared Memory system (denoted by the "BlastFunction shm" label) removes the limitations of gRPC-based transfers, avoiding multiple copies of the same buffer. In fact, the results show an improvement in terms of latency and overhead, which is composed only of the `memcpy` latency (limited only by the system memory buffer). The maximum overhead measured is of 155ms when transferring 2GBs, which represents 47.55% of the native execution latency (which is 326.89ms with the same input sizes). Most of the overhead is composed by the memory copy operation, while a smaller part ($\sim 2ms$ for every input size) is given by the gRPC control signals, which are used even in the shared memory system.

The previous results show only the read and write overheads, which for the majority of accelerators are negligible w.r.t the kernel execution time, depending on the kernel memory/compute intensive nature. We now describe the results regarding different accelerators execution.

Figure 6.2 shows the latency measurements for the Sobel operator. In the native scenario, the kernel shows a linear behaviour based on the input size, as expected from a linear filter implementation. The Native execution time starts from 0.27ms with a 10×10 image (800 bytes sent and received), up to 14.53ms for the largest image ($1920 \times$

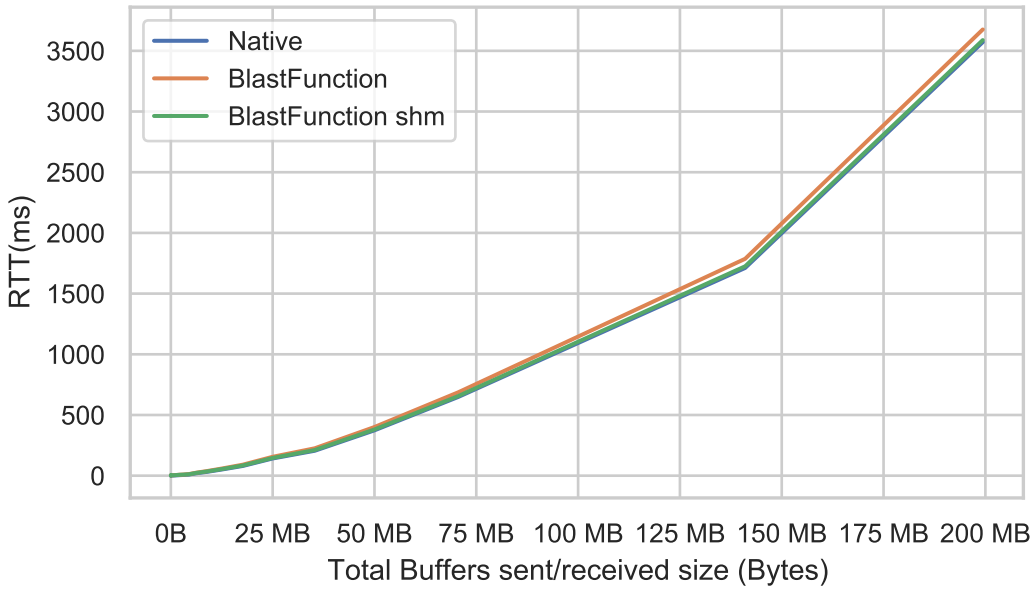

Figure 6.3: Latency overhead for MM accelerator at increasing input and output sizes (the "Native" latency line overlaps with the "BlastFunction shm" line).

1080 pixels, with a total read/write transfer of $\sim 8MB$). We can see that, in the BlastFunction results (both gRPC and shared memory) there is an initial 2ms overhead even at lower input sizes, given by the BlastFunction runtime and control messages (which travel through gRPC in both the gRPC and shared memory systems), as previously described in the I/O overhead results. The gRPC implementation shows a higher latency than the Native execution, starting from 2.46ms, then having an inflection point at about 1MB of buffers size and reaching a maximum of 24ms for the largest image. Instead, the shared memory implementation shows a mostly constant overhead (between 2-3ms) following the same linear behaviour of the Native implementation.

Figure 6.3 shows the latencies for the MM kernel. The MM accelerator is compute-intensive, meaning that the execution time is higher than the I/O times. In fact, the execution overhead between the native and remote execution is low for both communication systems (still remaining lower in the shared memory system). The Native runtime shows a minimum latency of 0.45ms for the smallest matrices (16×16 in both input and output matrices), but quickly rises for bigger matrices up to 3.571s (for 4076×4096 matrices). As in the Sobel accelerator results, both gRPC and shared memory implementations show a minimum latency of 2.46ms and 2.80ms respectively, given by the control signals. The gRPC-based runtime reaches a maximum of 3.675s, while the shared memory implementation reaches 3.588s for the largest case, which is only 17ms more than the Native execution.

Table 6.1 shows the minimum and maximum latencies observed in our tests using the

	Min (ms)	Max (ms)	Max (%)
Sobel	2.07	3.49	24.04
MM	2.35	17.70	0.27
PipeCNN - VGG	41.08	41.08	11.31
PipeCNN - AlexNet	15.79	15.79	31.27

Table 6.1: Latency Overhead results for the shared memory implementation (w.r.t native)

shared memory communication system. The minimum overhead is similar in both Sobel and MM (2.07ms and 2.35ms respectively), showing that this minimum latency is given by the already described factors (e.g. BlastFunction control messages and ACKs). The maximum latency overhead depends instead on the size of the buffer tested. For Sobel, we tested up until a Full HD image size (1920×1080 RGB pixels), with a minimum overhead of 2.07ms and a maximum of 3.49ms (about $\sim$ 24% w.r.t the native execution on the same input). The MM kernel was tested up to a 4096×4096 matrix, with 2 matrices in input and 1 in output. We show a minimum overhead of 2.35ms and a maximum of 17.7ms, which corresponds to $\sim 0.27\%$ of the native execution.

Finally, we tested the PipeCNN accelerator on two different network already provided by the authors, which are AlexNet and VGG. Given that each network takes a fixed-size image, we tested only that size and report the average latency in Table 6.1. For VGG acceleration we measured an average of 41ms of overhead w.r.t native execution, and 16ms for AlexNet. Considering the difference between the native execution time and the remote latency, we achieve $\sim 11.3\%$ overhead for VGG and $\sim 31.3\%$ for AlexNet.

Given the shown results for the overhead of our system when considering a single node setup, we can make some considerations. The first is that further studies need to be done with regards to the communication system. The current system introduces a small overhead because of the multiple copies done to the transferred data and the gRPC control signals. In fact, in all the tests we measured an initial 2-3ms step, given by this overhead. Moreover, in a synchronous scenario, multiple calls to parallel kernels may be delayed (as in the case of the PipeCNN accelerator).

A second consideration concerns the kernels which may be the best fit for this kind of system. In fact, the overhead results show that the latency in the read/write only scenario without executing a kernel is higher than when a kernel is running. This means that the overall impact of our system depends on the different complexity and operational intensity of the underlying accelerator. When the majority of the task execution time is passed in the kernel execution (as in the MM example, representing a highly operational intense kernel) the overall overhead is low. Instead, in a less complex or low operational intense accelerator the I/O latencies impact more on the task, even if the system tries to reduce the overhead through a shared memory system. This derives from the fact that the native system does not execute any copies of the data, while the isolation and transparency goals of our system require at least one copy. Given this facts, we

conclude that BlastFunction could target compute-intensive kernels in which the input and output data is low w.r.t the internally used data, such that the computation time exceeds the host-device transfer time. Examples of operational intense tasks are MM and Neural Networks, as the majority of the data remains inside the accelerator and the high complexity means that most of the execution time is passed computing internal values, without needing host-device communication.

6.2 Distributed System Evaluation

The previous section presented the results and considerations concerning the overhead introduced by our system in a single-node, single-application scenario. In this section, we turn the focus on the evaluation of the system in a small cluster. We first describe a single-application multi-node test to show how the FPGA utilization and the performances of our system differ from the native execution. Then, we present a multi-application scenario to show how our methodology allows to improve the current approach to FPGA usage in the cloud in terms of sharing and utilization.

6.2.1 Experimental Setup

To perform the distributed scenario experiments, we performed multiple tests in a local cluster deployed in our laboratory. The cluster is composed of three nodes, one master and two workers. The master node contains a single-socket 2.80GHz Intel® Xeon®W3530 CPU, with 8 threads (4 cores) and 24GB of DDR3 RAM. Each worker node is equipped with a single socket 3.40GHz Intel® Core™ i7-6700 CPU, with 8 total threads (4 cores) and 32GB of DDR4 RAM. Each node is connected to the local network through a 1Gb/s Ethernet link. Moreover, we installed a Terasic DE5a-Net FPGA board on every node in the cluster (including the master). The board includes an Intel®Arria 10 GX FPGA (1150K logic elements) along with 8GB RAM over 2 DDR2 SODIMM sockets and a PCI Express x8 connector (version 3 for worker nodes, version 2 for the master node).

To deploy the system, we followed the design described in Section 5.2.2. In particular, we deployed a central Prometheus server to collect metrics about the applications latency and the FPGAs devices utilization, and a OpenFaaS gateway to work as a Load Balancer for the tested serverless functions. In addition, for the BlastFunction tests we deployed the central Accelerators Registry and the Device Managers (the latters using a DaemonSet, resulting in one Device Manager running on each host in the cluster).

We tested the three use-cases described in Section 5.3, which are Sobel filter, Matrix Multiplication and AlexNet (using the PipeCNN accelerator). In particular, we developed a C++ HTTP server using the Pistache HTTP framework [40] to receive requests and call the accelerator with pre-defined data on each request. Then we wrapped the developed server into an OpenFaaS template based on Dockerfiles. The Native version of the Dockerfile links the server to a standard OpenCL library, which then directly accesses the host device through a bind mount on the container. The BlastFunction version of

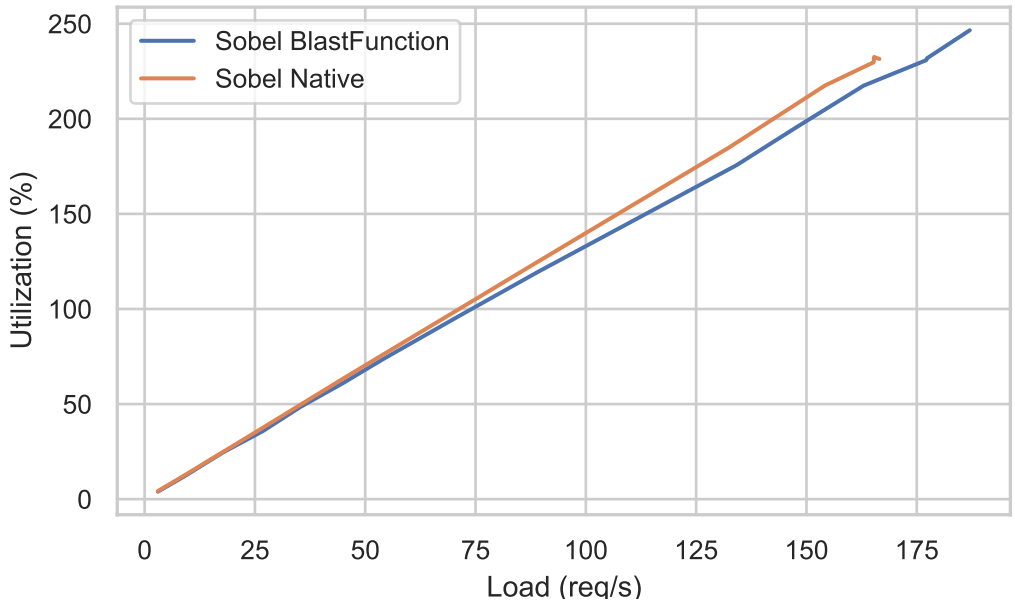

Figure 6.4: Device Utilization percentage (100% = One FPGA used 100% of the time) with an increasing number of processed requests per second for Sobel function using Native and BlastFunction runtimes.

the Dockerfile integrates instead with our Remote OpenCL Library, avoiding the direct access to the host devices. In addition, the Native application autonomously exposes the device utilization metrics using OpenCL profiling methods, while in the BlastFunction case all metrics are exposed by the Device Managers.

6.2.2 Single-application evaluation

In this first set of experiments, we decided to deploy and test a single application inside the cluster, to see how the system behaves under stress. Moreover, the experiments allow to test whether the introduced overhead impacts in a real-case scenario, in which multiple requests may arrive simultaneously and without idle periods. We tested each application using the Hey [41] tool for HTTP load testing. In particular, after deploying the application and the underlying system (e.g. Registry and Device Managers for the BlastFunction scenario), we executed the load test multiple times with an increasing number of requests sent per second, collecting the latencies (aggregated as average and percentiles) and the device utilization during the test.

Figure 6.4 shows the total devices utilization for the Sobel function. We can notice that both the Native and BlastFunction systems follow a linear utilization of the devices at increasing loads. At higher processed requests per second, the BlastFunction runtime

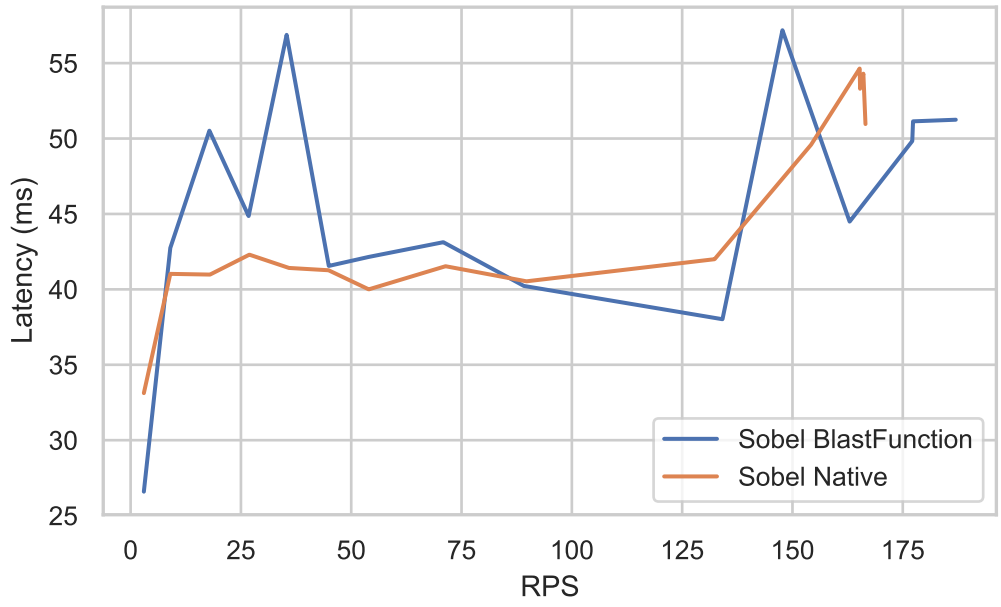

Figure 6.5: Sobel function latency with an increasing number of processed requests, using Native and BlastFunction runtimes.

is able to reduce the total utilization w.r.t the Native case. This allows the BlastFunction system to reach a higher throughput under load with the same utilization, thanks to the decoupling between the application and the device. For example, when the load tester sends 180 requests per second, the Native system is able to respond to 150 requests, resulting in a 207% utilization. Meanwhile, the BlastFunction serverless system reaches 163 requests processed and a utilization of 192.61%. A final consideration that can be made about this test is that both runtimes are not able to fully access the three devices in the cluster, due to additional overheads derived from processing multiple requests concurrently. In fact, connecting a *single* application per device does not remove the software overhead to the task execution. This happens because the function can execute only one task at a time due to the *serialization* employed by the runtime (single OpenCL queue in the native implementation, single task queue it the Device Manager for the BlastFunction system).

Figure 6.5 shows the average response latency of the function. The graph indicates that the function latency is higher than the corresponding kernel latency. In fact, in the overhead tests at the currently used input size (1920×1080 pixels) we measured an average execution time of 14.53ms for the Native runtime, and 18.03ms for the BlastFunction runtime (using the shared memory mechanism). The latencies measured in the distributed test are instead higher, in the 40-55ms range. This is due to the distributed fashion of the test which involves an additional latency observed by the client, and by the fact that multiple requests are processed at the same time. Indeed, as the function receives

requests from multiple connections, it enqueues multiple tasks to the runtime (either BlastFunction or Native) concurrently. However, both runtimes employ a single queue, meaning that each task waits for the previous ones to complete before being executed. This leads to an increase in the average response latency, as shown in the results.

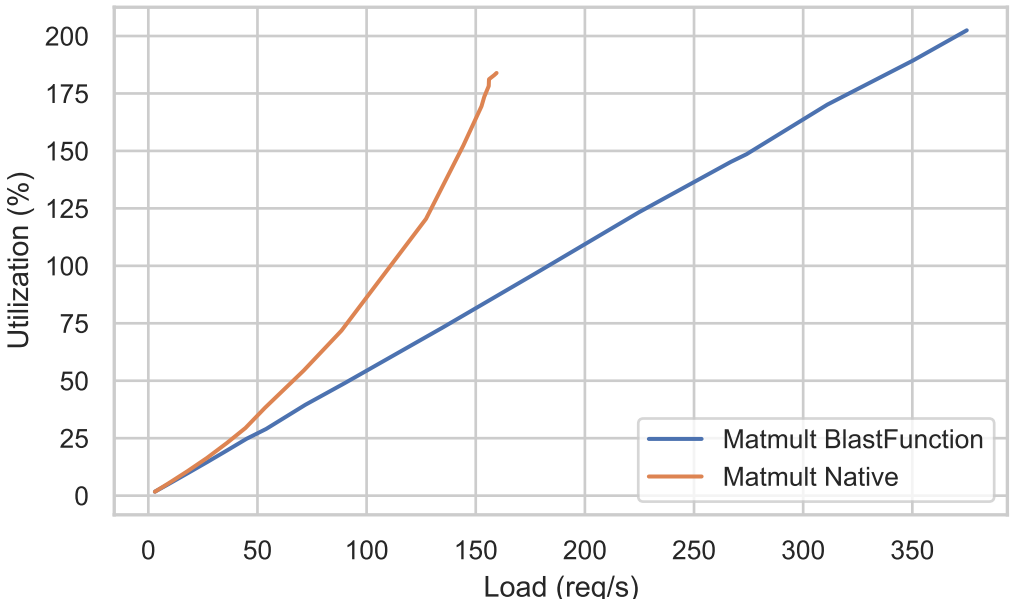

Figure 6.6: Device Utilization percentage (100% = One FPGA used 100% of the time) with an increasing number of processed requests per second for Matrix Multiplication function using Native and BlastFunction runtimes.

We now describe the results for the Matrix Multiplication function. Figure 6.6 shows the total devices utilization. As in the Sobel experiments, in both the Native and BlastFunction cases we were not able to fully utilize the available devices because of additional application overheads. The difference w.r.t the previously described experiments lies in the different utilization trend between the BlastFunction and Native runtimes. In fact, the BlastFunction runtime follows a linear behaviour when increasing the number of processed requests, while the Native runtime is not linear. This may be explained by the different nature of the Matrix Mutiply kernel w.r.t the Sobel kernel. The Sobel accelerator has a linear complexity as it is memory intensive. This means that, with a single accelerator synthesized on the device, the transfer times for the buffers can be used to interleave multiple kernel executions. The Matrix Multiplication kernel has instead a quadratic complexity, which means that the I/O latency doesn't interfere with the execution. The same results have already been observed in the overhead experiments, as we showed that the Matrix Multiplication kernel is not affected by our I/O overhead. This means that the system can not rely on transfer latencies to interleave kernel executions. Using BlastFunction instead the additional overhead introduced by the buffer copy op-

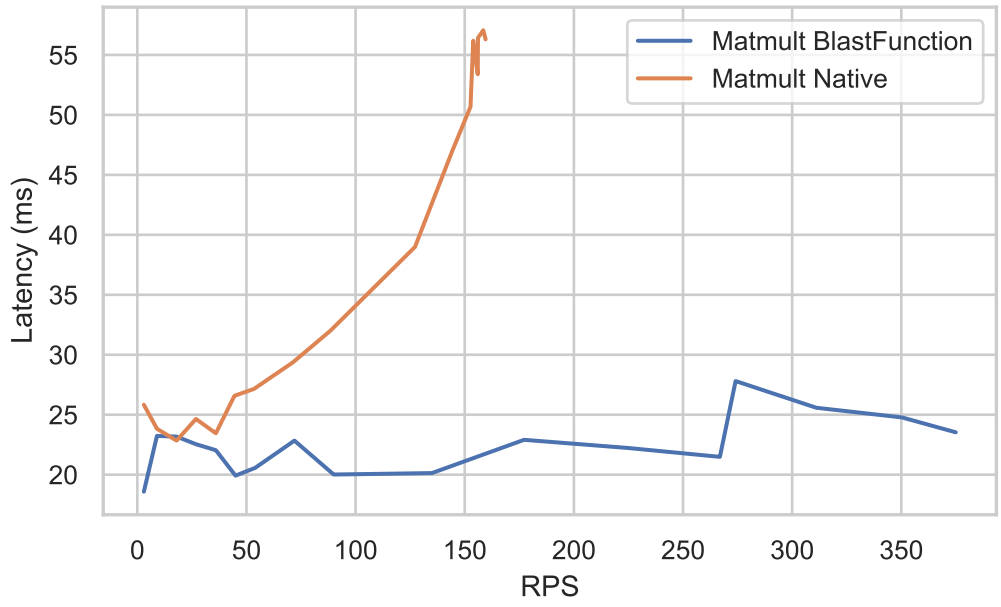

Figure 6.7: Matrix Multiplication function average latency with an increasing number of processed requests, using Native and BlastFunction runtimes.

eration allows to perform the interleaving, increasing the throughput. The results show that the Native runtime is able to reach a maximum of 159 requests per second, with a 183.91%, while the BlastFunction-based function was able to reach 374 requests per second with a 202.47% utilization. Moreover, as shown in Figure 6.7, the BlastFunction runtime latency is in the 20-30 ms range(min 18.58ms, max 27.80ms), while the Native latency grows from a minimum of 22ms to a maximum of 57.06ms.

We finally describe the results for the AlexNet function (which uses the PipeCNN accelerator). As shown in Figure 6.8 and Figure 6.9 the results are similar to the Matrix Multiplication experiment. The Native runtime reaches a maximum of 23.63 requests per second with a 242.16% utilization, while the BlastFunction runtime reaches 39.52 requests per second with a 249.40% utilization. Both runtimes latencies grow with the increasing number of requests issued, but the BlastFunction system is able to keep them under 300ms throughout the execution of the tests, while the Native runtime reaches a maximum of 390.2ms. In general, in both system the average latency is always higher than the latency measured in the overhead test (between 50-66ms) because of the network overhead between the testing node and the cluster and of the additional overhead given by the HTTP framework. Moreover, the PipeCNN accelerator presented additional challenges given by the presence of multiple OpenCL queues and multiple kernel executions in the same task, which increased the complexity and the latency for both runtimes. As the Matrix Multiplication accelerator, the PipeCNN kernel is compute-intensive, and this led to higher performances of the BlastFunction system w.r.t the Native runtime.

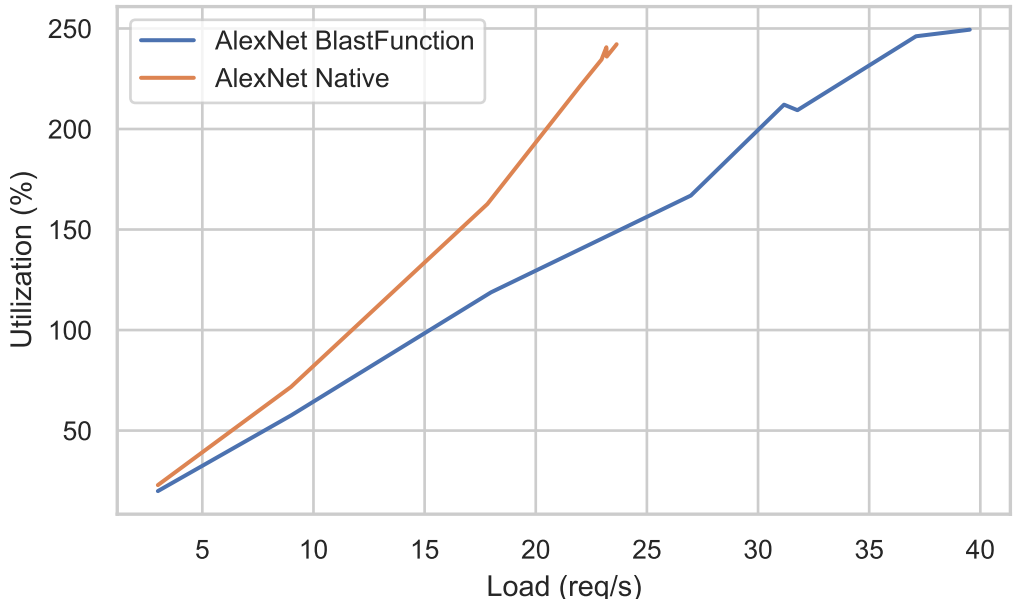

Figure 6.8: Device Utilization percentage (100% = One FPGA used 100% of the time) with an increasing number of processed requests per second for AlexNet function using Native and BlastFunction runtimes.

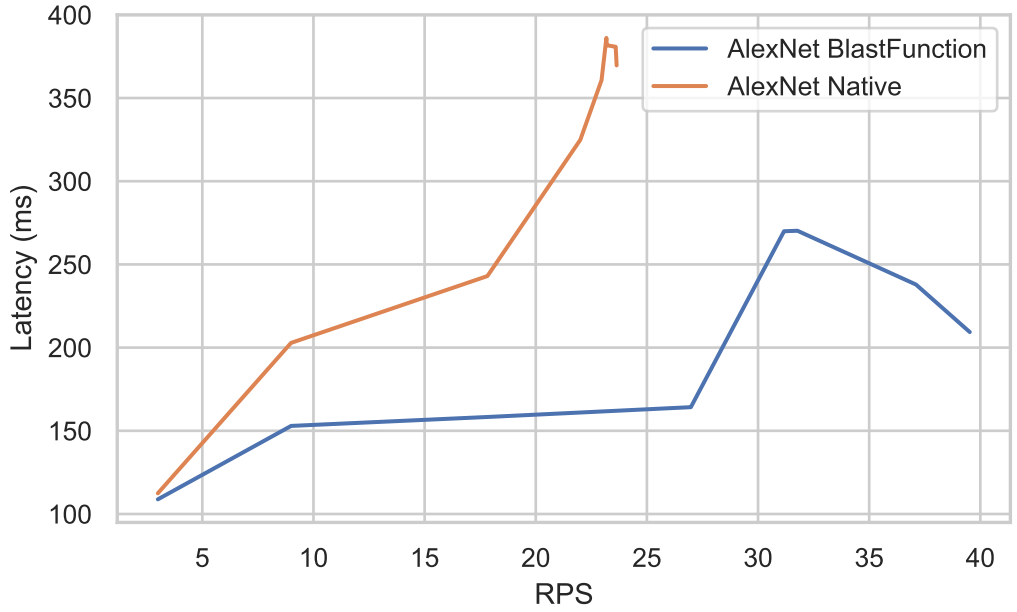

Figure 6.9: AlexNet function average latency with an increasing number of processed requests, using Native and BlastFunction runtimes.

Use-Case	Configuration	1st	2nd	3rd	4th	5th
Sobel	Low load	20 rq/s	15 rq/s	10 rq/s	5 rq/s	5 rq/s
	Medium Load	35 rq/s	30 rq/s	25 rq/s	20 rq/s	15 rq/s
	High Load	60 rq/s	50 rq/s	35 rq/s	30 rq/s	15 rq/s
Matmult	Low load	28 rq/s	21 rq/s	14 rq/s	7 rq/s	7 rq/s
	Medium Load	49 rq/s	42 rq/s	35 rq/s	28 rq/s	21 rq/s
	High Load	84 rq/s	70 rq/s	49 rq/s	42 rq/s	21 rq/s
AlexNet	Medium load	6 rq/s	3 rq/s	3 rq/s	3 rq/s	3 rq/s
	High Load	9 rq/s	9 rq/s	6 rq/s	6 rq/s	3 rq/s

Table 6.2: Tests configurations overview, showing how many requests per second were sent to each function for each use-case.

.

6.2.3 Multi-application evaluation

To properly test the assumption of this thesis work that sharing FPGAs represents an advantage in serverless cloud scenarios, we run a set of multi-application, multi-node experiments. In particular, we deployed multiple serverless functions using the same kind of accelerator in the previously described cluster, and tested them with different load configurations. The goal of these experiments is to verify that, under normal utilization (meaning that the system is not stressed), BlastFunction allows to increase the number of served tenants without major performance and latency losses.

For each use case (Sobel, Matrix Multiplication and CNN), we performed the experiments by deploying 5 functions with the same functionality (and using the same kind of accelerator) in the *BlastFunction* test, while we could deploy only 3 functions in the *Native* scenario (one for each device). Moreover, in order to test the system response and utilization without external influences given by the different allocation of the Function Instances, we opted for a fixed allocation to better show how sharing the FPGA allow to run more applications on the same device.

We then performed a load test using the Hey tool as in the previous tests, with different configurations regarding the number of requests per second to send to each function. The load test creates one HTTP connection to each deployed function, sending the chosen number of requests for each function sequentially. Table 6.2 show all the configurations used for the BlastFunction runtime. For the Native scenario we consider only the 1st, 2nd and 3rd columns, as there are only three functions deployed during the tests.

We show the per-function results for the Sobel use-case in Table 6.3. The results are divided in scenarios (BlastFunction vs Native), configuration and tested function. We also show the node on which each function was deployed in order to see the effect of sharing the device among multiple tenants. It can be seen that in the first configuration both runtimes allows the deployed functions to respond to almost all sent requests, with low

Type	Configuration	Function	Node	Utilization	Latency	Processed	Sent
BlastFunction	Low Load	sobel-1	B	21.95%	21.43 ms	17.25 rq/s	20.00 rq/s
		sobel-2	A	22.57%	24.23 ms	15.00 rq/s	15.00 rq/s
		sobel-3	C	13.22%	19.01 ms	10.00 rq/s	10.00 rq/s
		sobel-4	A	7.49%	31.98 ms	5.00 rq/s	5.00 rq/s
		sobel-5	B	6.48%	27.16 ms	5.00 rq/s	5.00 rq/s
	Medium Load	sobel-1	B	40.95%	19.45 ms	32.93 rq/s	35.00 rq/s
		sobel-2	A	39.40%	23.62 ms	26.30 rq/s	30.00 rq/s
		sobel-3	C	32.85%	18.28 ms	24.98 rq/s	25.00 rq/s
		sobel-4	A	29.85%	26.99 ms	19.98 rq/s	20.00 rq/s
		sobel-5	B	18.76%	22.94 ms	14.97 rq/s	15.00 rq/s
	High Load	sobel-1	B	60.31%	18.95 ms	49.58 rq/s	60.00 rq/s
		sobel-2	A	39.15%	32.05 ms	26.63 rq/s	50.00 rq/s
		sobel-3	C	45.75%	17.82 ms	34.96 rq/s	35.00 rq/s
		sobel-4	A	38.44%	22.56 ms	26.11 rq/s	30.00 rq/s
		sobel-5	B	18.39%	21.74 ms	15.00 rq/s	15.00 rq/s
Native	Low Load	sobel-1	A	30.41%	25.02 ms	19.49 rq/s	20.00 rq/s
		sobel-2	B	19.74%	21.50 ms	14.74 rq/s	15.00 rq/s
		sobel-3	C	13.73%	24.34 ms	9.75 rq/s	10.00 rq/s
	Medium Load	sobel-1	A	51.48%	26.04 ms	33.11 rq/s	35.00 rq/s
		sobel-2	B	37.19%	23.33 ms	27.95 rq/s	30.00 rq/s
		sobel-3	C	34.22%	23.48 ms	24.23 rq/s	25.00 rq/s
	High Load	sobel-1	A	58.10%	26.77 ms	38.36 rq/s	60.00 rq/s
		sobel-2	B	54.69%	23.95 ms	41.80 rq/s	50.00 rq/s
		sobel-3	C	44.81%	24.75 ms	32.61 rq/s	35.00 rq/s

Table 6.3: Multi-function test results for the Sobel accelerator, divided per System, Configuration and function.

latency (between 20-30ms, in line with the overhead results). In the second configuration the BlastFunction runtime (which represents a medium load) is sometimes not able to answer all requests, as the total load per node is near the limit of the accelerator in terms of throughput. For example, the `sobel-1` and `sobel-5` functions (both deployed on node B) receive a total of 50 requests per second and respond to 47.9 requests on average. The same happens between `sobel-2` and `sobel-4`, which respond to 46.2 requests over 50. In the last configuration the system is fully saturated, in fact the functions on nodes A and B do not respond to some requests Functions `sobel-1` and `sobel-5` processed 64.58 requests over 75, and the high stress put on the functions and the Device Manager leads to an underutilization of the board (78.7% average for node B). This derives from the acceleration contention between two tenants and a high number of requests. The Native scenario does not show the same behavior of the BlastFunction serverless system, because each function has an entire board for itself. However, we can see that in some cases the board utilization is lower, even w.r.t the single function utilization in the BlastFunction scenario. For example, in the third configuration, `sobel-1` experiences a considerable performance and utilization drop even if the board is not shared.

Table 6.4 shows the aggregate results of the Sobel use-case derived from the previous

Type	Configuration	Utilization	Latency	Processed	Sent
BlastFunction	Low load	71.70%	24.76 ms	52.24 rq/s	55 rq/s
	Medium Load	161.81%	22.25 ms	119.15 rq/s	125 rq/s
	High Load	202.03%	22.62 ms	152.27 rq/s	190 rq/s
Native	Low Load	63.87%	23.62 ms	43.98 rq/s	45 rq/s
	Medium Load	122.88%	24.28 ms	85.29 rq/s	90 rq/s
	High Load	157.60%	25.15 ms	112.77 rq/s	145 rq/s

Table 6.4: Multi-function test aggregate results for Sobel in terms of average latency, utilization and processed/sent requests.

table. We can see that the average response latencies (aggregated over the same test on multiple functions) are similar in both the Native and BlastFunction scenarios, even if the second case includes a concurrent access to the same device by multiple functions. Moreover, in the BlastFunction scenario we were able to fit more functions than in the Native scenario (5 instead of 3), and this is reflected by the total number of requests received and processed. This is the characteristic that allows BlastFunction to be a better system in medium load conditions, as it allows to improve the devices utilization by offering the accelerator to multiple tenants. Regarding the difference between sent and processed requests, the Native runtime presents an average of 2.25% in the low load configuration, and 5.23% and 22.22% for the medium and high load conditions respectively. The BlastFunction system has instead averages of 5.01%, 4.67% and 19.85% respectively. This means that in a medium load configuration the two systems are comparable, while in high load situations BlastFunction is slightly better (by 2.37%).

We now describe the sharing results regarding the Matrix Multiplication accelerator. Table 6.5 shows the per-function average measurements obtained. The BlastFunction implementation of the use-case shows a similar behaviour to the previous experiment for the Sobel accelerator. In fact, at low and medium load the function processes all requests and maintain a constant latency across all the functions (between 11 and 15ms). For example, at medium load function `matmult-2` is able to respond to 41.96 rq/s on average over the 42 rq/s sent by the load tester. In the same configuration, `matmult-4` responds to all the 28 rq/s sent, even if is shares the same device as `matmult-2` (on host A). Moreover, in the same example the two functions latency is the same (11.16ms) even if the device is shared. Even the "high load" configuration the functions are able to serve most of the requests. The Native runtime shows instead a behaviour similar to the single-node experiments previously described, as at a higher load the functions begin to drop requests at a high rate. For example, at high load the `matmult-1` functions do not respond to 43.85 rq/s, which is more than half the sent requests. In general, all the Native functions have a limit of 40/41 rq/s, showing a higher utilization that the same functions using BlastFunction.

Table 6.6 shows the aggregate results for the Matrix Multiplication use-case. We can

Type	Configuration	Function	Node	Utilization	Latency	Processed	Sent
BlastFunction	Low Load	matmult-1	B	15.79%	11.62 ms	27.98 rq/s	28.00 rq/s
		matmult-2	A	12.08%	11.61 ms	20.99 rq/s	21.00 rq/s
		matmult-3	C	7.61%	10.07 ms	14.00 rq/s	14.00 rq/s
		matmult-4	A	4.00%	13.96 ms	7.00 rq/s	7.00 rq/s
		matmult-5	B	4.01%	15.52 ms	7.00 rq/s	7.00 rq/s
	Medium Load	matmult-1	B	27.73%	11.70 ms	48.97 rq/s	49.00 rq/s
		matmult-2	A	24.00%	11.27 ms	41.96 rq/s	42.00 rq/s
		matmult-3	C	18.80%	9.59 ms	34.99 rq/s	35.00 rq/s
		matmult-4	A	15.99%	12.29 ms	27.99 rq/s	28.00 rq/s
		matmult-5	B	12.01%	13.00 ms	21.00 rq/s	21.00 rq/s
	High Load	matmult-1	B	43.27%	9.74 ms	81.33 rq/s	84.00 rq/s
		matmult-2	A	39.71%	11.16 ms	69.82 rq/s	70.00 rq/s
		matmult-3	C	26.07%	9.09 ms	48.98 rq/s	49.00 rq/s
		matmult-4	A	23.95%	11.96 ms	41.96 rq/s	42.00 rq/s
		matmult-5	B	11.19%	11.54 ms	20.65 rq/s	21.00 rq/s
Native	Low Load	matmult-1	A	21.55%	21.12 ms	26.73 rq/s	28.00 rq/s
		matmult-2	B	20.50%	24.18 ms	20.07 rq/s	21.00 rq/s
		matmult-3	C	8.82%	18.07 ms	13.70 rq/s	14.00 rq/s
	Medium Load	matmult-1	A	37.96%	24.45 ms	37.75 rq/s	49.00 rq/s
		matmult-2	B	39.49%	24.13 ms	36.31 rq/s	42.00 rq/s
		matmult-3	C	25.77%	19.87 ms	32.78 rq/s	35.00 rq/s
	High Load	matmult-1	A	41.20%	26.20 ms	40.15 rq/s	84.00 rq/s
		matmult-2	B	44.18%	24.86 ms	40.33 rq/s	70.00 rq/s
		matmult-3	C	37.59%	21.69 ms	41.37 rq/s	49.00 rq/s

Table 6.5: Multi-function test results for the Matrix Multiplication accelerator, divided per System, Configuration and function.

see that the numbers confirm the results we obtained in the single-application evaluation. In fact, the Native scenario presents a higher sent/processed requests difference than the BlastFunction system, with slightly higher latencies, and a similar utilization. The average difference for the BlastFunction functions is of 0.04%, 0.05% and 1.22% for the low, medium and high load configurations. Meanwhile, the Native functions reach a 3.97% difference with a low load, and 15.19% and 39.97% in medium and high load conditions.

Finally, we show the detailed and aggregate results for the AlexNet use case (using the PipeCNN accelerator). Because of the low number of requests that the accelerator is able to serve, we decided to test only two configurations, with medium and high load conditions. Table 6.7 contains the per-function results. As in the previous results, the BlastFunction runtime at low load conditions allows the deployed functions to respond to most of the requests sent by the load tester. However, the measured latency for the observed functions is in some cases more than double the latency observed in the overhead experiments. This derives from the long execution time of the tasks ran on the device, which create a higher probability of contention between multiple executions. Because the Device Manager serializes the arriving tasks on a single queue, most requests need to wait for the previous ones in order to start. We can see this behaviour in functions

Type	Configuration	Utilization	Latency	Processed	Sent
BlastFunction	Low Load	43.49%	12.55 ms	76.96 rq/s	77 rq/s
	Medium Load	98.53%	11.57 ms	174.90 rq/s	175 rq/s
	High Load	144.18%	10.69 ms	262.73 rq/s	266 rq/s
Native	Low Load	50.87%	21.12 ms	60.49 rq/s	63 rq/s
	Medium Load	103.22%	22.81 ms	106.84 rq/s	126 rq/s
	High Load	122.97%	24.25 ms	121.85 rq/s	203 rq/s

Table 6.6: Multi-function test aggregate results for Matrix Multiplication in terms of average latency, utilization and processed/sent requests.

Type	Configuration	Function	Node	Utilization	Latency	Processed	Sent
BlastFunction	Medium Load	pipecnn-1	B	42.18%	121.29 ms	5.99 rq/s	6.00 rq/s
		pipecnn-2	A	21.24%	152.65 ms	3.00 rq/s	3.00 rq/s
		pipecnn-3	C	17.45%	71.34 ms	3.00 rq/s	3.00 rq/s
		pipecnn-4	A	20.83%	153.59 ms	2.90 rq/s	3.00 rq/s
		pipecnn-5	B	22.97%	165.59 ms	3.00 rq/s	3.00 rq/s
	High Load	pipecnn-1	B	56.26%	97.63 ms	8.49 rq/s	9.00 rq/s
		pipecnn-2	A	51.53%	137.20 ms	7.13 rq/s	9.00 rq/s
		pipecnn-3	C	35.60%	70.50 ms	6.00 rq/s	6.00 rq/s
		pipecnn-4	A	37.97%	172.33 ms	5.21 rq/s	6.00 rq/s
		pipecnn-5	B	20.72%	144.93 ms	3.00 rq/s	3.00 rq/s
Native	Medium Load	pipecnn-1	A	49.42%	100.30 ms	5.92 rq/s	6.00 rq/s
		pipecnn-2	B	27.55%	105.49 ms	3.00 rq/s	3.00 rq/s
		pipecnn-3	C	19.25%	77.08 ms	3.00 rq/s	3.00 rq/s
	High Load	pipecnn-1	A	78.05%	102.57 ms	8.79 rq/s	9.00 rq/s
		pipecnn-2	B	71.82%	94.69 ms	8.80 rq/s	9.00 rq/s
		pipecnn-3	C	39.96%	77.97 ms	5.97 rq/s	6.00 rq/s

Table 6.7: Multi-function test results for PipeCNN (with AlexNet accelerator), divided per System, Configuration and function.

which are located on the same node, such as `pipecnn-2` and `pipecnn-4` in the low load configuration. In fact, while in the overhead tests we showed an average latency of 66ms for AlexNet execution, `pipecnn-2` responds with an average latency of 152.65ms, and `pipecnn-4` with an average latency of 153.59ms. In the same configuration instead, `pipecnn-1` and `pipecnn-5` show respectively 121ms and 165.59ms latencies. This is due to the high number of requests processed by `pipecnn-1`, which is double the requests sent to `pipecnn-5`. For every request, in fact, on average `pipecnn-5` needs to wait for two tasks enqueued by `pipecnn-1`. Finally, `pipecnn-3` is the only function deployed on node C, and shows a response latency similar to the one measured in the overhead tests, because it does not wait for other functions requests. The Native scenario shows lower latencies than the BlastFunction experiment, as every function is deployed without sharing the underlying device and without waiting for other functions tasks. However, as seen in the single-node experiments, it shows a higher device utilization, even if it able

Type	Configuration	Utilization	Latency	Processed	Sent
BlastFunction	Medium Load	124.68%	132.89ms	17.88 rq/s	18 rq/s
	High Load	202.08%	124.52ms	29.81 rq/s	33 rq/s
Native	Medium load	96.22%	94.29ms	11.91 rq/s	12 rq/s
	High Load	189.82%	91.74ms	23.57 rq/s	24 rq/s

Table 6.8: Multi-function test aggregate results for PipeCNN (with AlexNet accelerator) in terms of average latency, utilization and processed/sent requests.

to fulfill almost all sent requests.

Table 6.8 shows the aggregate results of the PipeCNN functions. The results show that the Native scenario has an average latency of 94.29ms for a medium load and 91.74ms for a high load, while the BlastFunction system presents a higher latency (132.89ms for medium and 124.52ms for high loads) as already shown in the previous detailed description. Regarding the unprocessed requests, the difference in the medium load configuration is small (0.63% for BlastFunction, 0.68% for Native), while in high load conditions BlastFunction shows a higher drop rate (9.64% vs 1.79%). However, in both configurations sharing the underlying devices allows the BlastFunction system to reach a higher utilization and number of processed requests, even with higher drops than in the Native scenario.

6.3 Closing remarks

In this Chapter, we evaluated the behavior of our system w.r.t the Native runtime implementation of three different use cases (Sobel, Matrix Multiplication and CNN).

We first measured the overhead introduced in a controlled environment, showing that our system adds between 0.27% and 24% overhead depending on the executed kernel and the size of the transferred data, thanks to an efficient buffer transmission mechanism.

We then performed the second set of experiments to test the behaviour of the system in a small, three node cluster, on a single application. In all cases we show major improvements in the throughput (up to 2.35x maximum requests processed per second for the MM and AlexNet cases) with the same devices utilization and without losses in the response latency.

Finally, we tested the system in a non-saturated scenario but with multiple applications sharing the devices, showing that BlastFunction is able to reach higher utilization and number of processed requests thanks to the sharing of the device with multiple functions (5 instead of 3 of the Native system), with minimal differences in latency and requests drop given by the concurrent access to the devices.

■

Chapter 7

Conclusions and Future work

In this thesis work we proposed *BlastFunction*, a distributed FPGA sharing system for the acceleration of microservices and serverless applications in cloud environments. The proposed system design is based upon the goals of *Multi-Tenancy* and *Scalability*, with a focus on the *Transparency* of the resulting software library, a *reconfiguration-aware* allocation of the devices and the integration with a widely used orchestrator.

In Chapter 2 and Chapter 3 we gave an overview of the background technologies and the State-of-Art works in the sharing of FPGAs and their integration in cloud environments. Most of the works handle the sharing challenge at the single-FPGA or single-node level, or try to offer entire FPGA in a cloud setting or a *Pool* of devices for batch systems. To the best of our knowledge, a complete system for FPGA sharing and allocation in a cloud scenario (including microservices-based and serverless platforms) is still missing. Moreover, most of the existing frameworks and systems are not *transparent* to the application developer, and are not integrated with existing and known container orchestrators (such as Kubernetes).

The system design described in Chapter 4 is composed of three main components: a *Remote OpenCL Library*, multiple *Device Managers* and a central *Accelerators Registry*. The Remote OpenCL Library allows serverless functions to access the shared FPGAs in the cluster. The component is custom OpenCL implementation which abstracts the use of the remote device access protocol and the communication to the other components of the system from the host code. The Device Manager is the server application deployed on every node in the system, which allows to perform *time-sharing* of the underlying device. Moreover, each Device Manager exposes metrics about the runtime behaviour of the device, allowing the other components to act accordingly. Finally, the Accelerators Registry is the central controller of the system, which tackles the goal of allocating the available devices efficiently using runtime performance metrics. It does so by tracking the device utilization metrics from the Device Managers and performing an online device allocation algorithm which takes also care of reconfiguring the devices at runtime. In addition, it intercepts the deployment and removal of applications inside the cluster to integrate them with the system and perform the allocation algorithm.

We described the system implementation, along with a description of the evaluated use cases, in Chapter 5. BlastFunction includes a common communication layer for

all the components implemented using gRPC for control messages and for out-of-node communications. Moreover, it provides a *shared memory* layer to perform buffer passing operations, which allows to reduce the overhead. The Accelerators Registry integrates with external components in three ways: for Kubernetes, with a set of *WebHooks* to intercept the Functions creation and deletion and its *API* to move them to the proper node; with the Prometheus metrics database to receive the runtime metrics needed for the allocation and reconfiguration algorithms.

We evaluated the system on three different use-cases to check our assumptions and verify our goals: Sobel, Matrix Multiplication and Convolutional Neural Networks. In all three cases, we used already available benchmarks to verify the host code transparency, and created serverless functions with the OpenFaaS system to perform our experiments. We first measured the overhead introduced by BlastFunction on a single node and a single application, which is between 0.27% and 24% depending on the executed kernel and on the size of the transferred buffers. Then, we performed a second set of experiments to test the behaviour of the system in a small, three node cluster equipped with Altera FPGAs, on a single application, showing major improvements in the throughput (up to 2.35x maximum requests processed per second) with the same devices utilization and without losses in the response latency. Finally, we tested the system in a scenario with multiple serverless functions. BlastFunction is able to reach higher utilization and number of processed requests thanks to the sharing of the device, with minimal differences in latency and processed requests given by the concurrent access to the devices.

Future Works We here describe the possible extensions/future works to BlastFunction, which are divided in two areas: cloud integration and methodology extensions. Regarding cloud integration, at the moment BlastFunction integrates with the Kubernetes orchestrator to trigger the allocation of devices to Pods and to enable the migration of pods when an FPGA is reconfigured. However, the system does not implement yet an autoscaling algorithm based on the current system utilization, given the limited number of devices used during the development. To this aim, future works will revolve around the development of an autoscaling methodology for cloud environments with more resources (such as AWS or Nimbix cloud).

Regarding the sharing methodology, a major limitation of BlastFunction is that it assumes a single accelerator per device, thus limiting the allocation to applications using the same kernels. Future works will include the integration of *space-sharing* techniques (such as FPGA *virtualization* or the use of *overlays*) alongside the main time-sharing methodology proposed in this thesis. This will allow to share the device among multiple functions which use different kernels, maximizing *area* utilization in addition to *time* slots. Finally, a possible extension of the methodology could include *pooling* techniques to offer multiple devices as a single one, as described in the works in Section 3.4.

■

Bibliography

[1] Yusuke Fujii, Takuya Azumi, Nobuhiko Nishio, Shinpei Kato and Masato Edahiro. Data transfer matters for gpu computing. In *2013 International Conference on Parallel and Distributed Systems*, pages 275–282. IEEE, 2013.

[2] Andrew Putnam, Adrian M Caulfield, Eric S Chung, Derek Chiou, Kypros Constantinides, John Demme, Hadi Esmaeilzadeh, Jeremy Fowers, Gopi Prashanth Gopal, Jan Gray et al. A reconfigurable fabric for accelerating large-scale datacenter services. *ACM SIGARCH Computer Architecture News*, 42(3):13–24, 2014.

[3] Shuichi Asano, Tsutomu Maruyama and Yoshiki Yamaguchi. Performance comparison of fpga, gpu and cpu in image processing. In *2009 international conference on field programmable logic and applications*, pages 126–131. IEEE, 2009.

[4] Bharat Sukhwani, Bulent Abali, Bernard Brezzo and Sameh Asaad. High-throughput, lossless data compresion on fpgas. In *2011 IEEE 19th Annual International Symposium on Field-Programmable Custom Computing Machines*, pages 113–116. IEEE, 2011.

[5] Philippos Papaphilippou and Wayne Luk. Accelerating database systems using fpgas: a survey. In *2018 28th International Conference on Field Programmable Logic and Applications (FPL)*, pages 125–1255. IEEE, 2018.

[6] Xuechao Wei, Cody Hao Yu, Peng Zhang, Youxiang Chen, Yuxin Wang, Han Hu, Yun Liang and Jason Cong. Automated systolic array architecture synthesis for high throughput cnn inference on fpgas. In *Proceedings of the 54th Annual Design Automation Conference 2017*, page 29. ACM, 2017.

[7] Amazon EC2 F1 instances. `https://aws.amazon.com/ec2/instance-types/f1/`. (Visited on 12/06/2019).

[8] Adrian M Caulfield, Eric S Chung, Andrew Putnam, Hari Angepat, Jeremy Fowers, Michael Haselman, Stephen Heil, Matt Humphrey, Puneet Kaur, Joo-Young Kim et al. A cloud-scale acceleration architecture. In *The 49th Annual IEEE/ACM International Symposium on Microarchitecture*, page 7. IEEE Press, 2016.

[9] Eric Chung, Jeremy Fowers, Kalin Ovtcharov, Michael Papamichael, Adrian Caulfield, Todd Massengill, Ming Liu, Daniel Lo, Shlomi Alkalay, Michael Haselman et al. Serving dnns in real time at datacenter scale with project brainwave. *IEEE Micro*, 38(2):8–20, 2018.

[10] Eric Jonas, Johann Schleier-Smith, Vikram Sreekanti, Chia-Che Tsai, Anurag Khandelwal, Qifan Pu, Vaishaal Shankar, Joao Carreira, Karl Krauth, Neeraja Yadwadkar et al. Cloud programming simplified: a berkeley view on serverless computing. *arXiv preprint arXiv:1902.03383*, 2019.

[11] Peter Mell, Tim Grance et al. The nist definition of cloud computing, 2011.

[12] Docker containers. `https://www.docker.com`. (Visited on 28/02/2019).

[13] Openfaas architecture image thanks to alex ellis. URL: `https://docs.openfaas.com/architecture/gateway/` (visited on 12/06/2019).

[14] Openfaas serverless framework. `https://www.openfaas.com/`. (Visited on 12/06/2019).

[15] Khronos group. `https://www.khronos.org/`. (Visited on 12/06/2019).

[16] Nimbix cloud fpga. URL: `https://www.nimbix.net/cloud-fpga/` (visited on 14/06/2019).

[17] Julio Proaño Orellana, Blanca Caminero, Carmen Carrión, Luis Tomas, Selome Kostentinos Tesfatsion and Johan Tordsson. FPGA-Aware Scheduling Strategies at Hypervisor Level in Cloud Environments. *Scientific Programming*, 2016(1), 2016. ISSN: 10589244. DOI: `10.1155/2016/4670271`.

[18] Wei Wang, Miodrag Bolic and Jonathan Parri. PvFPGA: Accessing an FPGA-based hardware accelerator in a paravirtualized environment. *2013 International Conference on Hardware/Software Codesign and System Synthesis, CODES+ISSS 2013*, 2013. DOI: `10.1109/CODES-ISSS.2013.6658997`.

[19] Qian Zhao, Motoki Amagasaki, Masahiro Iida, Morihiro Kuga and Toshinori Sueyoshi. Enabling FPGA-as-a-service in the cloud with hCODE platform. *IEICE Transactions on Information and Systems*, E101D(2):335–343, 2018. ISSN: 17451361. DOI: `10.1587/transinf.2017RCP0004`.

[20] Anca Iordache, Guillaume Pierre, Peter Sanders, Jose Gabriel de F. Coutinho and Mark Stillwell. High performance in the cloud with FPGA groups. *Proceedings of the 9th International Conference on Utility and Cloud Computing - UCC '16*:1–10, 2016. ISSN: 16130073. DOI: `10.1145/2996890.2996895`. arXiv: `arXiv:1603.07016v1`. URL: `http://dl.acm.org/citation.cfm?doid=2996890.2996895`.

[21] Mikhail Asiatici, Nithin George, Kizheppatt Vipin, Suhaib A. Fahmy and Paolo Ienne. Designing a virtual runtime for FPGA accelerators in the cloud. *FPL 2016 - 26th International Conference on Field-Programmable Logic and Applications*, 2016. DOI: `10.1109/FPL.2016.7577389`.

[22] Stelios Mavridis, Manolis Pavlidakis, Ioannis Stamoulias, Christos Kozanitis, Nikolaos Chrysos, Christoforos Kachris, Dimitrios Soudris and Angelos Bilas. VineTalk: Simplifying software access and sharing of FPGAs in datacenters. *2017 27th International Conference on Field Programmable Logic and Applications, FPL 2017*:2–5, 2017. DOI: `10.23919/FPL.2017.8056788`.

[23] Marcello Pogliani, Gianluca C. Durelli, Antonio Miele, Tobias Becker, Peter Sanders, Cristiana Bolchini and Marco D. Santambrogio. Quality of Service Driven Runtime Resource Allocation in Reconfigurable HPC Architectures. *Proceedings - 19th IEEE International Conference on Computational Science and Engineering, 14th IEEE International Conference on Embedded and Ubiquitous Computing and 15th International Symposium on Distributed Computing and Applications to Business, Engi*:16–23, 2017. DOI: 10.1109/CSE-EUC-DCABES.2016.156.

[24] Anuj Vaishnav, Khoa Dang Pham, Dirk Koch and James Garside. Resource Elastic Virtualization for FPGAs using OpenCL.

[25] Zhuangdi Zhu, Alex X. Liu, Fan Zhang and Fei Chen. FPGA Resource Pooling in Cloud Computing. *IEEE Transactions on Cloud Computing*, PP(c):1, 2018. ISSN: 21687161. DOI: 10.1109/TCC.2018.2874011.

[26] Yangming Zhao, Chen Tian, Zhuangdi Zhu, Jie Cheng, Chunming Qiao and Alex X. Liu. Minimize the Make-span of Batched Requests for FPGA Pooling in Cloud Computing. *IEEE Transactions on Parallel and Distributed Systems*, 29(11):2514–2527, 2018. ISSN: 15582183. DOI: 10.1109/TPDS.2018.2829860.

[27] D. Ojika, A. Gordon-ross, H. Lam, B. Patel, G. Kaul and J. Strayer. Using FPGAs as Microservices: Technology, Challenges and Case Study. *Bpoe*:0–5, 2018.

[28] Adrian M Caulfield, Others, Eric S Chung, Puneet Kaur, Joo-young Kim Daniel, Lo Todd and Massengill Kalin. A cloud-scale acceleration architecture. *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*:1–13, 2016. ISSN: 10724451. DOI: 10.1109/MICRO.2016.7783710.

[29] Jagath Weerasinghe, Francois Abel, Christoph Hagleitner and Andreas Herkersdorf. Enabling FPGAs in hyperscale data centers. *Proceedings - 2015 IEEE 12th International Conference on Ubiquitous Intelligence and Computing, 2015 IEEE 12th International Conference on Advanced and Trusted Computing, 2015 IEEE 15th International Conference on Scalable Computing and Communications, 20*:1078–1086, 2016. DOI: 10.1109/UIC-ATC-ScalCom-CBDCom-IoP.2015.199.

[30] Stuart Byma, J. Gregory Steffan, Hadi Bannazadeh, Alberto Leon-Garcia and Paul Chow. FPGAs in the cloud: Booting virtualized hardware accelerators with OpenStack. *Proceedings - 2014 IEEE 22nd International Symposium on Field-Programmable Custom Computing Machines, FCCM 2014*:109–116, 2014. DOI: 10.1109/FCCM.2014.42.

[31] David Ojika, Piotr Majcher, Wojciech Neubauer, Suchit Subhaschandra and Darin Acosta. SWiF: A simplified workload-centric framework for FPGA-based computing. *Proceedings - IEEE 25th Annual International Symposium on Field-Programmable Custom Computing Machines, FCCM 2017*:26, 2017. DOI: 10.1109/FCCM.2017.52.

[32] Selome Kostentions Tesfatsion, Julio Proaño, Luis Tomás, Blanca Caminero, Carmen Carrión and Johan Tordsson. Power and performance optimization in FPGA-accelerated clouds. *Concurrency Computation*, 30(18), 2018. ISSN: 15320634. DOI: 10.1002/cpe.4526.

[33] Mikhail Asiatici, Nithin George, Kizheppatt Vipin, Suhaib A. Fahmy and Paolo Ienne. Virtualized Execution Runtime for FPGA Accelerators in the Cloud. *IEEE Access*, 2017. ISSN: 21693536. DOI: 10.1109/ACCESS.2017.2661582.

[34] Julio Proaño Orellana, María Blanca Caminero and Carmen Carrión. On the provision of SaaS-level quality of service within heterogeneous private clouds. *Proceedings - 2014 IEEE/ACM 7th International Conference on Utility and Cloud Computing, UCC 2014*:146–155, 2014. DOI: 10.1109/UCC.2014.23.

[35] Grpc. URL: https://www.xilinx.com/products/design-tools/software-zone/sdaccel.html (visited on 26/06/2019).

[36] Protocol buffers. URL: https://developers.google.com/protocol-buffers/ (visited on 26/06/2019).

[37] Protobuffer serialization benchmarking results. URL: https://github.com/protocolbuffers/protobuf/blob/master/docs/performance.md (visited on 10/06/2019).

[38] Quentin Gautier, Alrie Althoff, Pingfan Meng and Ryan Kastner. Spector: An OpenCL FPGA benchmark suite. *Proceedings of the 2016 International Conference on Field-Programmable Technology, FPT 2016*:141–148, 2017. DOI: 10.1109/FPT.2016.7929519.

[39] Paul Dong Wang, Ke Xu and Diankun JiangBeckett. PipeCNN: An OpenCL-Based Open-Source FPGA Accelerator for Convolution Neural Networks.

[40] Pistache c++ rest framework. URL: http://pistache.io/ (visited on 26/06/2019).

[41] Hey load tester. (Visited on 25/08/2019).