Design and implementation of a framework for efficient remote reconfigurable accelerator deployment in disaggregated environment

Pissadakis Emmanouil

Department of Electrical and Computer Engineering
Technical University of Crete

Supervisor
Professor Dionisios Pnevmatikatos

Committee
Assistant Professor Vasilios Samoladas
Professor Ioannis Papaefstathiou

In partial fulfillment of the requirements for the degree of Master of Science in Engineering in computer Science

JULY, 2019

Abstract

Cloud computing usage has drastically increased over the years, providing data security and privacy which are prime concerns these days. The scalability of the cloud capacity, as well as the accessibility of the provided services, constitute relevant factors for the cloud computing evolution. Data centers are mainly deployed as cloud computing resources to deal with large storage and computation requirements. The need for specialized hardware acceleration in this domain is well established and intensified by the insatiable demand for compute power. Hardware accelerators can also provide high energy efficiency for many application domains in comparison with current architectures based on general purpose processors. The fixed amount of the available resources constitutes the major disadvantage of traditional data centers. Resource disaggregation alleviates this issue while offering the opportunity to manage resources more efficiently. In disaggregated computing environments, where all data transfers between remote nodes are realized via packet exchanges over a rack-scale network, reducing communication and synchronization is a prerequisite to the effective employment of remote acceleration. To this end, this thesis presents ReFiRe [1] (Remote Fine-grained Reconfigurable acceleration), a generic deployment framework with native support for partial reconfiguration that allows considerable reduction of communication needs between a processor and remote accelerators. Custom instructions that encapsulate complex sequences of operations and their respective synchronization requirements deployed for shifting control flow and partial reconfiguration decisions to the remote side. Considering the high complexity of the instruction-initialization procedure, a source-to-source transformation framework based on the ReFiRe infrastructure was further implemented. Through this framework, these instructions are automatically generated according to application requirements transparently to the user level. To evaluate ReFiRe, three benchmark applications were employed. A 2D-FFT algorithm, a genomics application that detects positive selection in genomes and a Binarized Neural Network, demonstrate that offloading computations to remote accelerators using ReFiRe leads to superior aggregate performance on the same specialized hardware platform compared to using dedicated accelerator calls on a per-operation basis.

This work has been supported in part by EU H2020 ICT project dRedBox, contract #687632

Acknowledgments

Many people helped and inspired me in order to complete this thesis. First, I would like to thank my professor Dionisios Pnevmatikatos for giving me the opportunity to work in the field that I am interested in. Second, special thanks to Nikolaos Alachiotis for the cooperation and his constant technical support during this thesis, as well as to Dimitris Theodoropoulos for his helpful advises. Additional thanks to my friends for their vital help. Finally, I am indebted to my family for their mental support during my studies.

Publications

1. Emmanouil Pissadakis, Nikolaos Alachiotis, Panagiotis Skrimponis, Dimitris Theodoropoulos, Thanasis Korakis and Dionisios Pnevmatikatos *"ReFiRe: Efficient Deployment of Remote Fine-Grained Reconfigurable Accelerators"*, International Conference on Field-Programmable Technology (FPT), Okinawa, Japan December 2018

2. Panagiotis Skrimponis, Emmanouil Pissadakis, Nikolaos Alachiotis and Dionisios Pnevmatikatos *"Accelerating Binarized Convolutional Neural Networks with Dynamic Partial Reconfiguration on Disaggregated FPGAs"*, paraFPGA 2019

Contents

List of Figures

List of Tables

Nomenclature

ACI Advanced Co-processor Instructions

ANN Artificial Neural Network

API Application Programming Interface

AS Accelerator Slot

BNN Binarized Neural Network

CNN Convolutional Neural Network

CPU Central Processor Unit

DC Datapath Constructor

DFD Data Fetch and Dispatch

DMA Direct Memory Access

DPR Dynamic Partial Reconfiguration

$FPGA$ Field Programmable Gate Array

$ICAP$ Internal Configuration Access Port

MM Memory Mapped

PF Parameter File

PR Partial Reconfiguration

PRR Partial Reconfiguration Region

RM Reconfigurable Module

Chapter 1

Introduction

Over the past years, cloud computing usage has drastically raised since a pool of available resources are available over the internet anytime. The quality of services, alongside the descending costs and the verified reliability, have driven companies of all sizes to adopt this technology. Rather than having desktop software, cloud computing offers space for maintaining data over the network, while providing recovery services that handle data restoration reducing data loss possibility. Device diversity and security are also important parameters of cloud computing employment.

Dealing with storing and accessing large amount of data by companies or organizations increases the need of data center deployment in cloud services. Data centers are defined as locations where computing and networking equipment is concentrated for the purpose of processing, storing and collecting large amount of data. Traditional data centers usually consist of a static number of servers, each having a specific amount of memory and computational power. The major drawback that data centers have to cope with is the variance of the applications' requirements. Many applications may utilize a large amount of data while processing resources are under-utilized. Considering traditional data center architecture, the efficient resource utilization constitutes an important research field.

To alleviate this issue, resource disaggregation has been proposed. Separating data-center components into particular servers could perform optimal utilization. The traditional architecture which is based on the monolithic server approach should be replaced with a resource-centric architecture where pools of processors, memory and storage are interconnected through the network. Additionally, the rapid growth of hardware application acceleration while maintaining low energy consumption could derive FPGAs as an additional execution resource.

Exploiting the efficient resource utilization of the disaggregated data centers, as well as the improved performance of the hardware accelerators, this thesis introduces a framework for efficient remote reconfigurable accelerator deployment in disaggregated environment aiming to reduce the communication/synchronization ratio between the remote nodes.

1.1 Motivation

The end of Dennard scaling [6] in CMOS technology in combination with the increasing demand for computing power led to the implementation of domain-specific execution resources in today's computing platforms. The major conundrum that developers have to resolve in the hardware accelerator deployment, is the trade-off between efficiency and applicability. Which of these options should be the most critical factor in the accelerator development? Deploying an accelerator architecture with increased specialization that could derive better performance at the cost of reduced applicability and load imbalance risk (coarse-grain accelerators), or constructing accelerators with reduced specialization (fine-grained accelerators) to achieve better load balance at the cost of a higher synchronization requirements? The main performance principle is the same, and comes down to how to increase the time spent on computation (accelerator operation) while reducing the time spent on synchronization (host-accelerator communication). This naturally leads to larger, coarser-grained, less applicable custom architectures. Consequently, to broaden an accelerator's scope of application, it is a prerequisite to reduce synchronization, which will, in turn, pave the way for finer-grained and more generic accelerator architectures since the more basic the operation, the wider the application domain.

The main drawback of the traditional data centers was the problem of the fixed resource proportionality. This problem is addressed in disaggregated data centers which are based in distinct components that perform specific functionality (e.g. memory, compute, acceleration, etc). Thus, efficient resource utilization according to each application requirements could be accomplished. Hardware accelerators are important architectural components in the context of data center customization because they can achieve high performance while maintaining lower energy consumption.

In disaggregated computing platforms, where all the nodes are distributed through the network, the need of low-latency communication between the host-processor and the remote accelerator is a prerequisite. Especially, when hardware execution resources are deployed in a disaggregated environment, it is of paramount importance to alleviate the high communication and/or synchronization requirements for remote, fine-grained accelerators while providing flexibility during the deployment procedure.

1.2 Thesis contributions

In this thesis, we propose ReFiRe, a generic deployment framework that introduces various degrees of flexibility in reconfiguring at run time and orchestrating fine-grained accelerator cores on the reconfigurable fabric of a remote, FPGA-based multiprocessor system-on-chip (MPSoC). Particularly, the contributions of this thesis are:

- The proposal of a novel hardware architecture that dynamically interconnects partially reconfigurable regions (PRRs) to construct larger pipelines aiming to boost the performance of the deployed accelerator cores (reconfigurable modules (RM)) by increasing computation in-between synchronization and/or partial reconfiguration (PR) events.
- The design of Advanced Co-processor Instruction(ACI) for offloading operations to RMs. An ACI consists of variable length instructions that grant high degrees of flexibility in deploying remote accelerators.
- The implementation of high-level API that exposes all the ACI-construction functionality at the user level, enabling applications to deploy, interconnect, and reconfigure remote accelerators at run time.
- The implementation of a source-to-source transformation framework built on the high-level API to construct ACIs according to application requirements, transparently to the user level.
- The evaluation of a row-column 2D-FFT algorithm, a genomics application that detects positive selection in genomes and a Binarized Neural Network over ReFiRe, demonstrating that offloading computations to remote accelerators using ReFiRe leads to superior aggregate performance on the same specialized hardware platform compared to using dedicated accelerator calls on a per-operation basis.
- The evaluation of two alternative execution scenarios of the Binarized Neural Network, in Static and Partial Reconfigurable architectures exploiting DPR and intra-layer parallelism through dedicated features of the ReFiRe. More importantly, the proposed approach is highly generic and versatile, thus allowing to boost performance of existing CNN and/or BNN accelerators using DPR and parallelism, with negligible development effort.

1.3 Thesis outline

Chapter 2 provides a background of cloud computing while analyzing how FPGAs could be efficiently exploited in these environments. Furthermore, disaggregated data centers' architecture is analyzed and explained how it addresses the traditional data center restrictions. Chapter 3 presents Host - FPGA communication related work as well as FPGA-based implementations in data center environment. In chapter 4, the ReFiRe framework is going to be analyzed. Especially, Host and FPGA side components as well as the implemented communication method are going to be depicted. Furthermore, this chapter presents the ACI class hierarchy and the specialization principles

that ReFiRe relies on. Chapter 5 analyzes the prototype platform used for the evaluation. Chapter 6 contains experimental results for the 2D-FFT, a genomics application and a Binarized Neural Network implementations over ReFiRe, while chapter 7 concludes this thesis and suggests future improvements.

Chapter 2

Background

2.1 Cloud computing

Cloud computing could be defined as the delivery of on-demand computing services including servers, storage, databases, networking, software, analytics, and intelligence over the internet. With cloud computing, users can access files and use applications from any internet-connected device. Data in the cloud are usually stored stored on many physical and/or virtual servers that are hosted by a third-party service provider. Low-cost, high performance productivity and reliability are the main benefits of cloud computing.

Cloud computing service models describe how cloud services are made available to clients and divided into three categories.

1. **The Software as a Service (SaaS):** Provides online software solutions to the clients. Everything is available over the internet when clients log in to their account online. They can usually access the software from any device anytime. Google Apps, Dropbox, Salesforce and Cisco WebEx are the most common examples in this category.

2. **The Platform as a Service (Paas):** Hardware and software tools available over the internet. Usually an API is provided through this service that includes a set of functions for programmatic platform management and solution development. AWS [7], [8] Elastic Beanstalk, Heroku and Windows Azure are the most commonly used examples.

3. **The Infrastructure as a Service (Iaas):** Via this service, infrastructure components are provided to the clients. Thus, clients have access to the lowest-level software like virtual machines, storage, networks and firewalls. Amazon Web Services is one of the largest providers.

Cloud computing usage has immensely increased through the years because of the easy access in any infrastructure/software available over the network anytime. Furthermore, FPGAs (Field

Programmable Gate Arrays) acceptance for computation acceleration has also increased. High throughput, predictable latency while maintaining reprogrammabilty and low power consumption could be accomplished by deploying FPGAs as execution resources. Hence, the simultaneous rise in popularity for both the cloud and FPGAs grew the demand for deploying FPGA-based applications in cloud environments. Fei Chen et al. [9] proposed four fundamental requirements that had to be addressed in order to integrate FPGA resources into the cloud

1. **Abstraction:** FPGAs must be exhibited to the cloud via an easy to use API. FPGA programming is a difficult process that requires strong programming skills. Pursuant to this demand, if the provided interface isn't well established then deploying a FPGA in the cloud could be a very difficult and time consuming procedure.

2. **Sharing:** FPGA resource utilization among various users in the cloud is the most important requirement that had to be addressed. Sharing and isolating the available resources should be efficiently accomplished targeting to provide anytime the maximum available amount to the users.

3. **Compatibility:** Many different tool chains and applications are used for the FPGA programming in stand alone environments. Thus, the software development kit that is going to be provided in the cloud had to compatible to the most commonly used of them.

4. **Security:** As FPGAs were not designed for multi-user needs splitting FPGA resources in multiple users could derive multiple issues. The reliability of the system is a very important factor for a user who wants to deploy his system on the cloud. If there is no security in users transactions and a shared compute host could be easily brought down, then users aren't going to be attached in this cloud.

Knodel [2] defined three service models for FPGAs in a cloud environment. Their work mainly focused on enabling remote FPGAs for acceleration and also providing background acceleration for data centers with multiple users on the same physical FPGA. The main component of their system (Reconfigurable Common Cloud Computing Environment-RC3E) is the hypervisor RC3E. It manages the resources and provides access to the FPGA devices. The first model as illustrated in Figure 2.1 below named (The Reconfigurable Silicon as a Service – RSaaS) provides full access to the reconfigurable resource. Consequently, users can allocate a complete physical FPGA and can implement the hardware of their choice. Virtual machines with the appropriate FPGAs devices attached are allocated by the users for driver and hardware interface development.

In this model the whole development flow is provided as a cloud service. The concept can be compared to the cloud service models Platform as a Service (PaaS) and Infrastructure as a Service(IaaS).

The Reconfigurable Accelerators as a Service (RAaaS) constitutes the next model. According to this, the FPGA is accessible through a computing framework in order to be deployed as a hardware

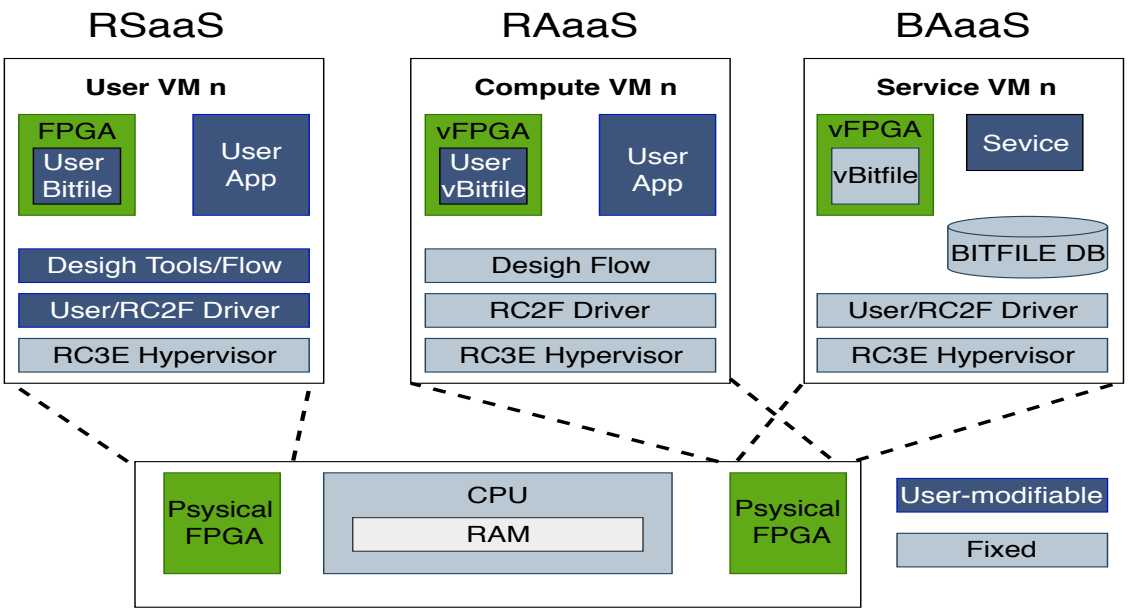

Figure 2.1: The three service models provided in the cloud environment as presented by Oliver Knodel [2].

accelerator. Memory interfaces on the FPGA as well as a communication API is provided in the host side. Users have to design the computation core in the vFPGA and a program to send/receive data. This way the system is safer than the RSaaS model. The RAaaS model can be compared to the PaaS model.

The third model (Background Acceleration as a Service – BAaaS) is suitable for applications and services running in common data centers. vFPGAs aren't accessible to the users. Several services and applications are running in the background in order to accelerate the application. A resource management system is responsible for resource allocation and vFPGA reconfiguration. Because of the transparency of that model, it could be compared to Service (SaaS) model.

2.2 Data center disaggregatation

The prompt growth of cloud computing nowadays improved the use of the data centers. Traditional data centers are usually defined as servers consisting of a fixed amount of storage, computing and memory resources as illustrated in Figure 2.2 below. CPU to memory I/O ration is predetermined and unchangeable [10]. This add barriers and limitations related to resource utilization and data management. Assigning a processing intensive task to conventional server will exploit all the processing resources while memory module will be idle. Nevertheless, servers couldn't access these resources because of the restricted current data center architecture. Similarly, for a memory intensive task the processing core would be unreachable leading to inefficient resource utilization.

The concept of disaggregation in computing servers has the intent to break the boundaries of

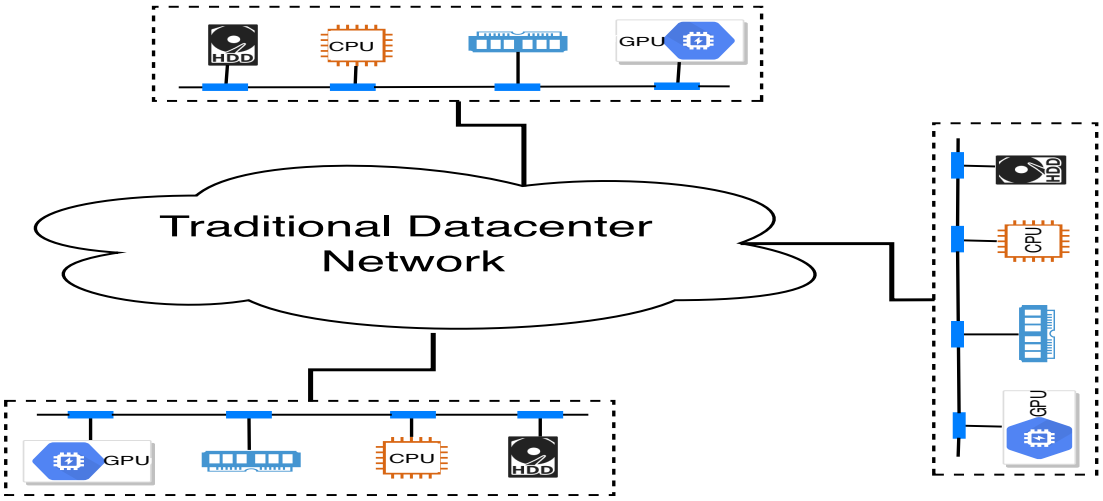

Figure 2.2: Traditional data center network.

current compute, memory, network and storage components built in a hard, unique and tightly connected unit. Next generation data centers will likely be based on the emerging paradigm of

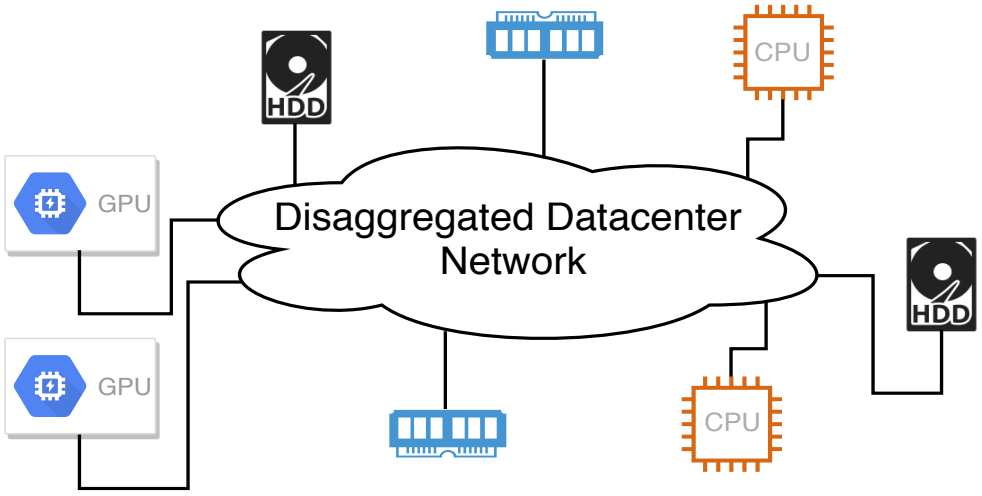

Figure 2.3: Disaggregated data center network.

disaggregated function blocks as a unit departing from the current state of main board as a unit. As shown in Figure 2.3 multiple functional blocks or bricks such as compute, memory and peripheral will spread through the entire system and interconnected together via one or multiply high speed networks. This new architecture brings various benefits that are desirable in today's data centers such as fine-grained technology upgrade cycles, fine grained resource allocation and access to a larger amount of memory and accelerators. Disaggregation of resources in the data center, especially at the rack-scale, offers the opportunity to use valuable resources more efficiently

Alachiotis [11] proposed dRedbox, a disaggregated architectural perspective for Data Centers. As above mentioned, traditional data centers usually consist of a fixed ratio of resources (mainboard tray along its hardware and software components). Hence, that restriction exposes several drawbacks. First, system-level upgrades are very restricted because of the basic mainboard tray. Upgrading a processor, for instance, usually leads to incompatibility problems related to other components e.g memory and peripherals. Second, VM initialization/allocation in data centers is also restricted due to the fixed amount of available resources. Finally, the costs of technological upgrades are rising significantly due to the hardware dependencies.

Figure 2.4 below illustrates the limitation of current data centers infrastructures regarding resource utilization for four different VMs in three separate servers. For the first VM 3 MEM Units and 1 CPU core are allocated in the 1st server, 2 MEM units and 1 CPU core for the second VM in the second server and 3 MEM and 1 CPU core for the third VM in the third server. According to the data center infrastructure each VM could be allocated only in a single server. As shown in figure 2.3 (b) in dRedbox resource allocation VMs could be allocated in different memory and CPU units. Thus, VM4 is also allocated in the free MEM and CPU space, exploiting resource disaggregation.

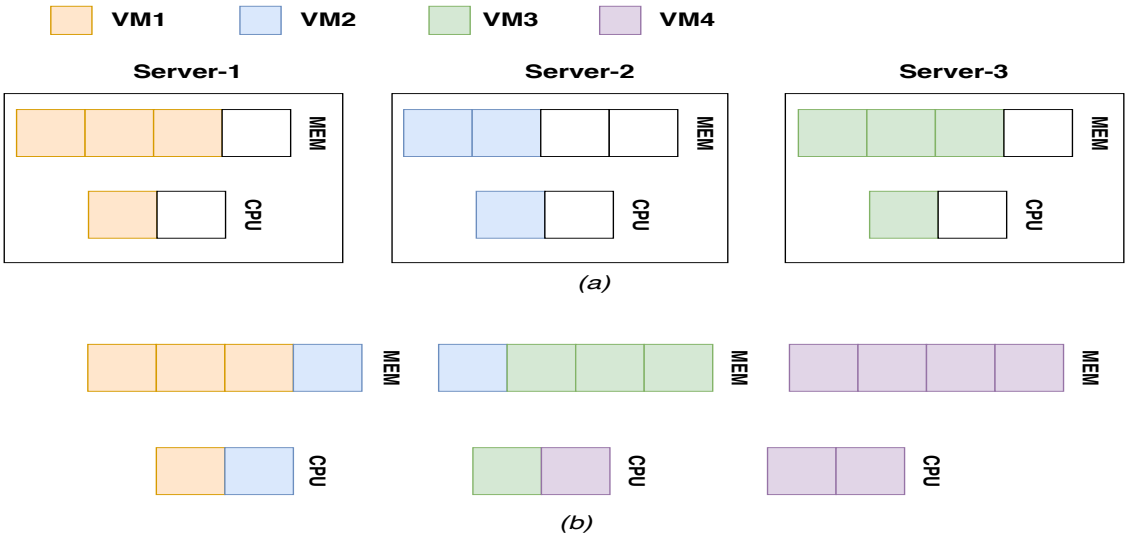

Figure 2.4: A limitation of current data center infrastructures regarding resource utilization (a), and the respective resource allocation scheme of dReDBox (b).

Overcoming the fixed architectural design of traditional data centers that are based on the monolithic block design and introducing novel data center infrastructure in order to achieve better resource allocation derives several challenges that had to be addressed. These challenges are related to memory, network and hardware/software platform requirements. Inter and intra-node communication paths between the blocks of the system had to be configured and designed efficiently in order

to reduce the communication overhead. Furthermore remote memory allocation had to maintain coherency and consistency while providing minimal remote-memory access latency. Finally, the appropriate software that defines resource typologies, ensures reliability while maintains correctness had to be configured properly to address system's problems that may occur.

The dReDBox approach is based on resource disaggregation of arbitrary types such as memories, processors and FPGA-based accelerators. These blocks defined as bricks, are constructing pools of resources with system software tools implementing software-defined virtual machines intending on better resource utilization according to each application needs. Figure 2.5 exhibits the expected resource allocation schemes, for current and dReDBox enabled infrastructure, for serving the requirements of memory intensive applications. According to traditional data centers (Figure 2.5 (a)) the application had to be dispatched into two different nodes because each of them could accommodate only 32 GB of memory. As a consequence, only 25% of processing resources per node is going to be utilized. In dRedBox infrastructure (Figure 2.5 (b)) three dbricks are going to be initiated. The first will handle the processing requirements of the application while the other ones the memory requirements. This way,the available dbicks are going to be fully utilized. Communication between bricks is achieved through high-speed, low-latency optical and electrical networks.

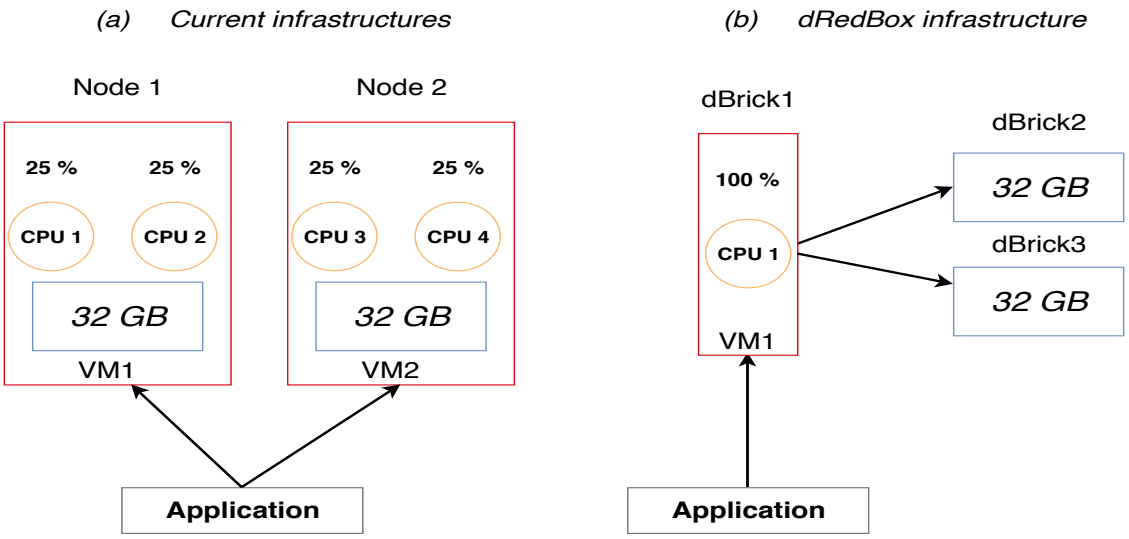

Figure 2.5: Example of a memory-intensive application and the respective resource allocation schemes in a current infrastructure (a) and dReDBox (b).

2.3 The need for hardware acceleration

Processor designers as time follows increase the number of processor cores exploiting Moore's Law [12] rather than focusing on single score performance. As single core scaling has been reduced, the

failure of Dennard scaling [6] will soon limit multicore scaling as well. Hadi Esmaeilzadeh et al. [3] combined device scaling, single-core scaling, and multicore scaling to model multicore scaling limits for by measuring the speedup for parallel workloads for next technology generations. For their approach three different models evaluated which in combination with empirical measurements projected multicore performance and chip utilization.

As illustrated in Figure 2.6 below the **Device scaling model** (DevM) provides the area, power, and frequency scaling factors at technology nodes from 45 nm to 8 nm, considering TRS Roadmap projections [13] and conservative scaling parameters from Borkar's recent study [14]. The **Core scaling model** (CorM) maintains the maximum performance per area for a single-core. Furthermore, it provides the appropriate power that had to be consumed to keep up the corresponding performance. Finally, for the **Multicore scaling model** (CmpM) two classes of multicore organizations (multi-core CPUs and many-thread GPUs) evaluated. For each of them considered four topologies: symmetric, asymmetric, dynamic, and composed.

Finally, the product of CmpM, DevM and CorM outcomes the multicore speedup estimation through the years. As shown if the figure above the amount of "dark silicon" (transistor under-utilization) gap continuously grows up. According to this assumption, the deployment of better architectured single-core processors could lead to better performance in comparison with multicore processors. To this end, the implementation of fine-grained accelerators seems to be promising for accelerating large scale applications.

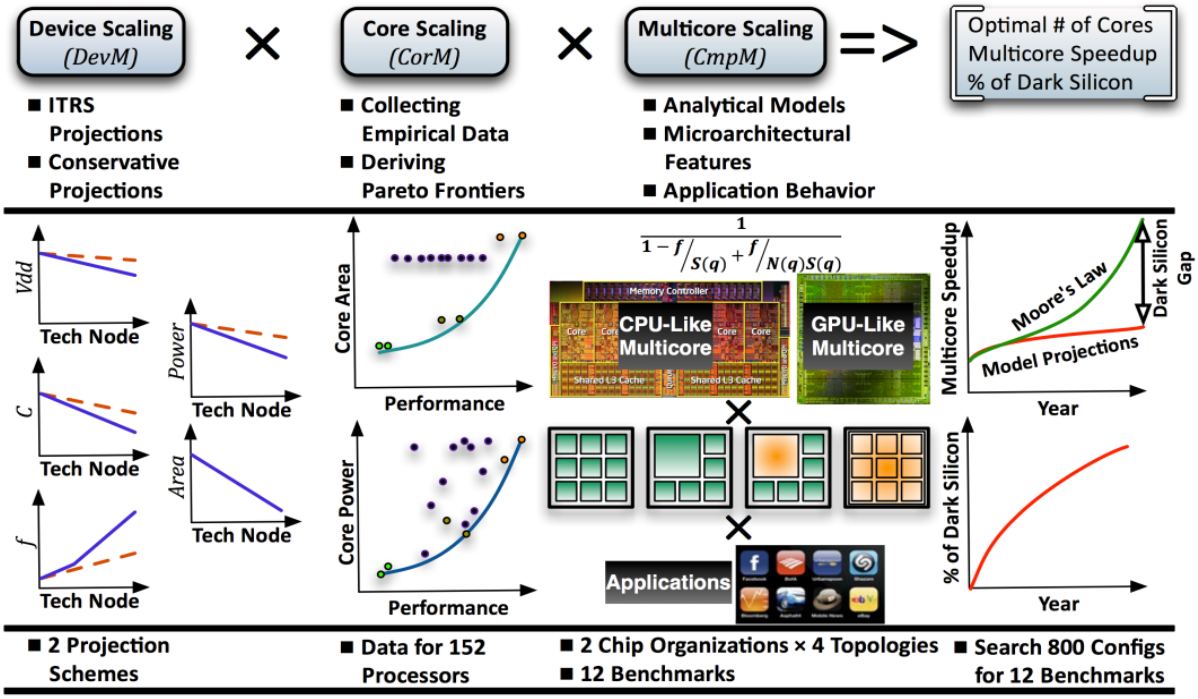

Figure 2.6: Dark silicon gap, as presented by Hadi Esmaeilzadeh et al. [3].

Michael Ferdman et al. [15] based on their previous work [16] proved that datacenters which use modern server hardware seems to be inefficient for scale-out workloads that need large-scale computational resources. According to their assumption modern datacenter architecture is designed for an extensive market. Current processors are build using increased clock speeds and (power and area) inexpensive transistors. However, the ending of Dennard scaling [6] made these two factors limited by power.

Scale-out workloads spend most of their time waiting for cache misses, instead of other workloads which are based on parallel execution of the processors. As a consequence, during scale-out workloads execution cores are mostly idle because of the high instruction-cache miss rates. Furthermore, using wide processors for these applications doesn't yield significant benefits. Generally, modern processors sacrifice efficiency in order to be able to accommodate different workloads. The need of specialized processors that can efficient handle specific applications is necessary for improving the throughput of the processor and the overall datacenter capability.

2.4 FPGA-CPU communication in data centers

Recently, FPGAs utilization into data centers (DC) has raised. They are mainly deployed to offload and accelerate specific services that intent low power consumption. As analyzed from Jagath Weerasinghe et al. [17] the cloud is housed in Data Centers which are based on on ever shrinking servers. FPGAs must be deployed as independent Data Center resources, so as to be accessible to the cloud users. They proposed to decouple the FPGA form the CPU and connect them as standalone resource in the data center network. This way, they could be handled from the users as a standard server. However, instantiating an FPGA in a data center infrastructure would be a difficult process. Figure 2.7 below shows the three different approaches for setting up a large number of FPGAs into a data center. For the first option the FPGA is incorporated into the same board as the CPU. That approach breaks the homogeneity of the compute module in an environment where server homogeneity is sought to reduce the management overhead. Moreover, if they both occur in the same server and a hardware problem exists in one of them, then the whole node had to be offline, making the other resource unusable. The most popular option is to implement the FPGA on a daughter-card and communicate with the CPU over a high-speed point-to-point interconnect such as the PCIe-bus Figure 2.7 (b) creating two separate data center nodes. However, this is the most popular type it comes with several drawbacks dealing when used in data center architecture. Initially the use of the FPGA is bonded to the workload of the CPU. Deploying a small amount of PCIe buses per CPU may lead to FPGA under-provision and vice versa. Additionally, the number of the FPGAs that an application may need to utilize isn't known a priori and usually cause under/over-utilization issues. This approach was also adopted by Microsoft [18] and going to be analyzed in more detail in the next section. According to the third method as shown in Figure 2.7 (c) the FPGA is directly hooked to the data center network. Joining a network

controller interface to an FPGA could enable the communication with other resources, such as disks and memory. Hence, FPGA modules can then be deployed in the data center independently of the number of CPUs overcoming the limitations of the two previous options.

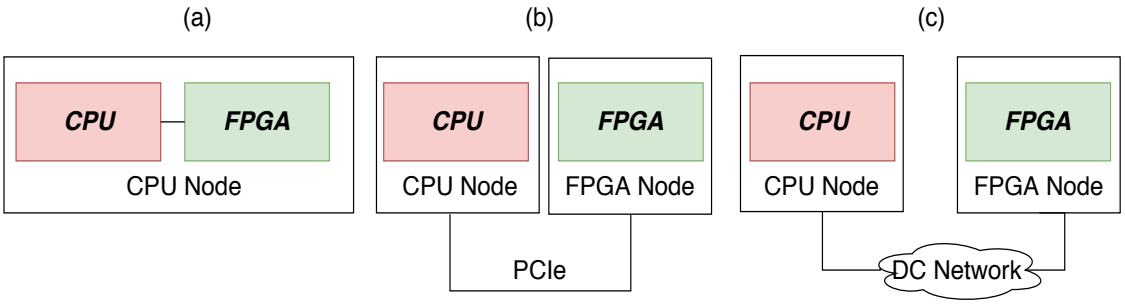

Figure 2.7: Options for Attaching an FPGA to a CPU.

Based on the third option Weerasinghe et al. [19] implemented a prototype architecture of network-Attached FPGAs for data center applications. This architecture was partitioned into two main layers: The application layer (vFPGA) and the network service layer (NSL). The vFPGA has one or more communication links through the NSL to the servers and to other vFPGAs over the data center network. The NSL provides the network connection for vFPGA to communicate with servers and other vFPGAs over the DC network. It consists of an application interface layer, a management layer, and a network protocol stack. The manager layer is responsible for listening FPGA-configuration-related commands from the external software service. The Application Interface Layer creates the appropriate data path between the FPGAs while the network protocol stack, contains a network interface and a protocol stack to connect the FPGA to the DC network.

Performance and power are two of the most important challenges dealing with cloud computing. While performance could be theoretically considered as a fixed factor, the scale of data centers makes power a particularly significant factor. As above mentioned, these two challenges are usually solved with accelerator (FPGAs and GPUs) deployment. Generally, accelerators are employed in master-slave architectures where a host processor controls how the accelerators are used. Naif Tarafdar et al. [20] proposed an architecture where any computing device could interact with any other computing device over the data center network. In cloud environments, where most of the clients aren't developers, the need of making Field-Programmable Gate Arrays (FPGAs) easier to use as computing devices is urgent. Further more, removing a layer of complexity from the programmers provides code portability which is very important. Consequently, the major challenge they had to deal with was to develop an infrastructure that could hide code complexity in order to make the acceleration transparent to the users while maintaining efficient resource utilization.

Initially they developed a Message-Passing Interface (MPI) [21] in heterogeneous system [22] for FPGA - processor communication. This was the basis for implementing an abstraction layer

between software applications and the hardware. When moving to data center environment they implemented even more layers of abstraction. The Resource Management and Resource Allocation layers used to keep track of the computing resources and handle requests for resources from the users. Openstack [23] services deployed for the networking. Via the interconnect layer the hardware processor layer of the software node is connected with the Board Support Package Hardware layer of the hardware node.

Depending on the layers of abstraction that they crated in order to deploy FPGAs on the cloud a specific software/hardware design flow had to be initiated. This flow is analyzed below :

1. Implementation of all parts of the design in software.
2. Implementation and testing each individual function as an FPGA-offloaded design.
3. Swapping the software-based function with the tested FPGA-based kernel.

Assuming these works the integration of FPGAs in data center environments constitutes a difficult procedure which prerequisites the implementation of various layers that provide the reconfiguration of the accelerators transparent to the users as well as a secure interconnection that ensures low synchronization requirements.

2.5 Remote-accelerator deployment challenges

Many FPGA-based accelerators need to communicate with other devices, either because the logic does not fit on a single FPGA or since the data calls for pre- or post- processing for which another platform is better suited. The most popular interconnection method as proposed in [24] [25] [26] [27] [28] [29] is the PCIe bus. Because there is no standard host-FPGA communication library, FPGA developers have to write significant amounts of PCIe related code. Furthermore, FPGA-related software drivers had to be implemented from the host side in order to deploy these accelerators. These efforts are heavy tasks for the developers and usually mislead them from their main task which is the FPGA accelerator deployment. Flexibility and efficiency are the main challenges that they have to deal with. The communication must maintain high data throughput while consuming a minimal amount of FPGA resources.

So far, FPGAs have mainly been used as static accelerators, designed once and used for a single function. In the cloud context, the ability to modify accelerator functions at runtime in a multi-user environment is essential. This requires new techniques in hardware design, interfacing, accelerator management, OS integration, and programming models. As analyzed from Suhaib A. Fahmy et al. [30] the key challenges that had to be addressed for a cloud-centric integration of FPGAs are :

- Support for dynamically reconfigurable accelerators to support changing application needs with low latency

- Maximizing communication throughput to multiple accelerators with fair, segregated sharing
- Maximized usage of FPGA resources at all times through efficient scheduling and allocation
- Easy integration of accelerated tasks within software applications

As extracted from the above mentioned, the most important challenge in the remote accelerator deployment is to minimize the synchronization requirements between the remote nodes while providing flexibility to the users via an easy to use Application Programming Interface

Chapter 3

Related work

Efficient communication between a host processor and accelerator devices is of paramount importance to the effective deployment of specialized solutions, which led to various frameworks been proposed in recent years.

Neves et al. [24] described a flexible interfacing framework to establish communication between a host processor and custom FPGA-based accelerators over PCI express interconnection. They proposed a framework that targets heterogeneous systems consisting of an FPGA device connected with a CPU through high speed interconnection.They evaluated their framework in a a Xilinx 7-Series FPGA device, connected to a host x86 CPU through a PCI Express interconnection. As shown in Figure above the reconfigurable fabric composed of two main modules. The Host Interface Bridge (HIB), that handles all communication with the PCIe and the accelerator module which is divided into the controller (module that manages the accelerator) , the Host Interface Bridge for the communication and a global memory. The software layer that takes place in the CPU is composed of two modules : i) the device driver and a low level Application Programming Interface which via high level routines provides to the user the ability to load data and code to the processing cores and manage the accelerator side.

Assuming various application-specific processing elements (PEs) mapped onto an FPGA, a set of control and execution commands, exposed to the host processor via the low-level API, allow to reset and initiate operations on the PEs, as well as transfer data to on-chip memory. The PEs are controlled by four registers that are written by the host processor on a per-command basis, introducing computation (via the PE identification register) and communication (via the broadcast register) flexibility. However, explicit synchronization is required at an operation-level granularity, since only a single command register is employed. ReFiRe can achieve higher computation-to-synchronization ratios since an ACI can contain an arbitrary number of operations to execute before the next synchronization event.

RIFFA 2.1 [31] is a low-level PCIe-based framework that establishes CPU-FPGA communication through dedicated channels between software threads and accelerator cores. Unlike its predecessor, RIFFA 1.0, it saturates the PCIe link for both upstream and downstream transfers. Furthermore,

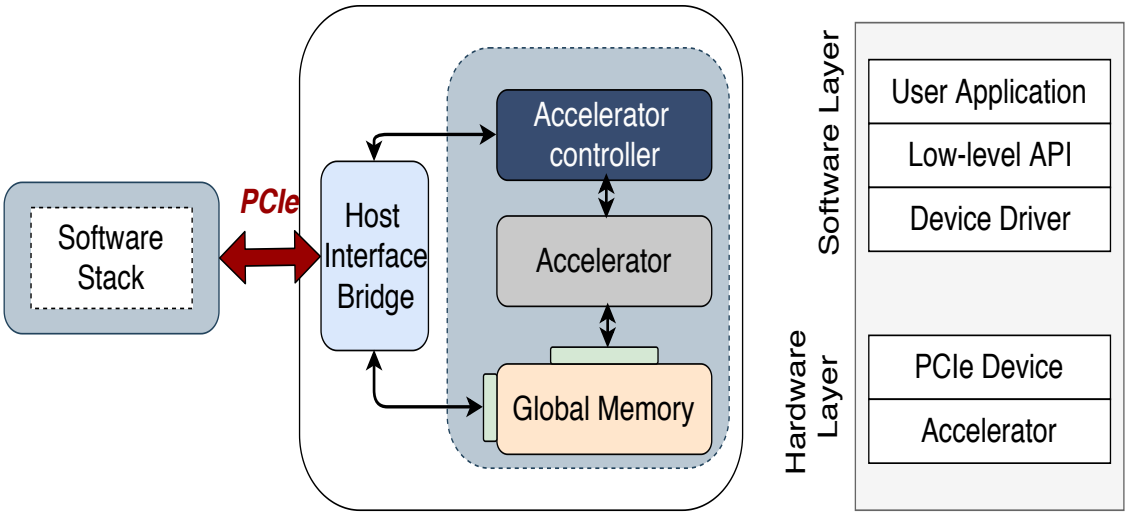

Figure 3.1: Heterogeneous computing framework overview and framework integrated stack, depicting both the software and the hardware layers.

it supports up to 5 FPGA devices, and provides bindings for C/C++, Java, and Python. A set of basic software functions allows to open/close a communication channel, as well as exchange data with the accelerator hardware, requiring a custom RIFFA-based implementation to introduce flexibility in deploying accelerator cores

Guillermo et al. [29], presented the MPRACE framework, an open-source stack for conducting data transfers between a host CPU and DDR memory on an FPGA board. The solution consists of a generic PCIe driver, a DMA engine, a hardware-abstraction library for IO, and a buffer-management library for data transfers. This framework is adopted in many FPGAs to Host CPU (through PCIe) connections because of the flexibility that is provided through the well established libraries. On the other hand its interrupt handling method outcomes a major problem. An interrupt could arrive at the Host before the Transaction Layer Package (TLP)s of the corresponding transfer had been received. Compared to our approach, through the Advanced Co-processor Instructions an efficient interrupt handling mechanism is implemented resolving this issue.

Paiágua et al. [27] described HotStream, a communication framework for stream-based architectures that exhibits a software and a hardware layer. The software layer consists of an API for deploying complex streaming patterns, and a device driver to map user-specified memory buffers to the physical address space. In the hardware layer, the Host Interface Bridge handles the data transfers between the host processor and the accelerator, while the Multi-Core Processing Engine (MCPE) manages the streaming of data through an interconnected array of processing elements (PEs). This significantly improves performance due to reduced communication requirements between the accelerator hardware and the host processor, which is achieved by supporting direct inter-PE data exchanges.

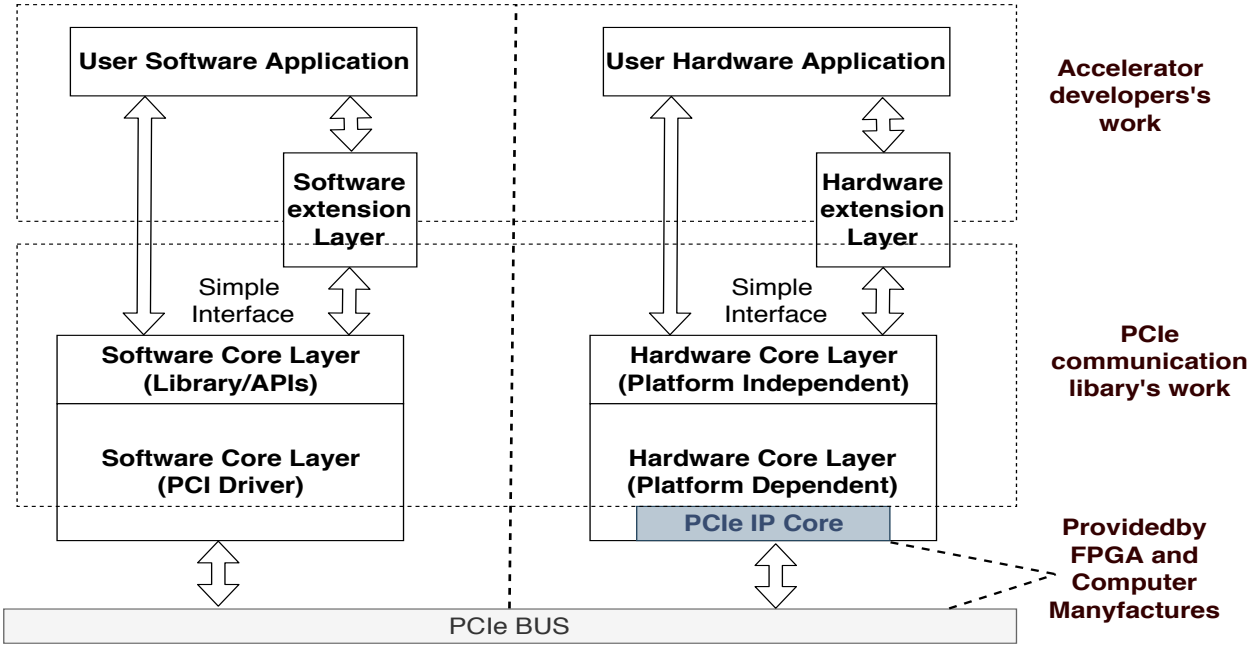

Figure 3.2: EPEE system overview. EPEE consists of a software component and a hardware component, each includes a core layer and a extension layer.

Jian Gong [28] et al. proposed EPEE , an efficient and flexible host-FPGA PCIe communication library. That library provided a collection of common functions that aid the communication between the FPGA and the Host computer. Through these functions, the control of registers for read/write device state registers, read/write memory transactions and PCIe user-defined interrupts are handled. They targeted to achieve high flexibility through balancing efficiency and functionality.

Figure 3.2 above, illustrates the overall system architecture that they proposed. It consists of a software and hardware component. The hardware side consists of a platform independent and a platform depended part. The fist part is responsible for handling host processors activities while the second one refers to each hardware platform requirements that had to be defined in order to port their framework in different hardware platforms. The software side, composed of a linux driver a software Core Layer and a Software extension layer. The driver is responsible for controlling the FPGA hardware via a set of specific system calls The software core layer provides an easy to use application programming interface for data transfer. Their implementation focused on satisfying two major design requirements. The easy to use APIs that is going to be provided to the users and the transparency of the system by hiding low-level details.

The above mentioned related work focused on PCIe based implementations for host - FPGA communication and synchronization. On the other hand, in the cloud computing field many worth-noticed works have already been implemented.

Amazon offers its F1 machine instance type [32], where developers can accelerate application tasks on either a single or eight Xilinx FPGA chips. Alibaba has already started providing cloud resources to its Chinese customers that utilize FPGAs towards improving performance of AI-based applications [33].

Andrew Putnam et al. proposed Catapult [18] targeting to face the problem of the high computational capabilities , power efficiency and flexibility that datacenter workloads request. The power limitations that servers have to deal with, have significantly slowed their improvement. FPGAs offers the potential of providing application acceleration with low power consumption. Reconfiguration and area utilization are the main challenges of deploying FPGAs in the cloud . To this end , Catapult alleviates these issues by providing a low power reconfigurable fabric (embedded into the servers) that could accelerate portions of large-scale software services.

FPGA and CPU communication is achieved through a custom PCIe interface that warranties low latency and multithreading safety. A low-level software library provided to the users in order to configure the fabric with the the appropriate function , for passing the corresponding bitstream in the FPGA . Additionally, they divided the programmable logic of the FPPGA in two main parts : the *shell* which constitutes the reusable part (DRAM controllers, router logic , DMA , high speed serial links) and a fixed region named *role* that corresponds to the application logic. They managed to achieve 90-95% improvement in ranking throughput for a fixed latency compared to the software approach.

Vineyard [34] mainly focused on the automatic accelerator utilization. To achieve that, they developed a energy-efficient, integrated platform for data centers that consisted of a software framework and FPGA-integrated servers. Vineyard mainly focused on six objectives: i)The development of hardware accelerators that could be coupled to server processors in heterogeneous data centers, ii) The incorporation of energy-efficient processors (ARM cores) with FPGAs targeting low energy consumption and high performance FPGA-accelerated servers , iii)the implementation of a programming framework that could hide the complexity of programming hardware accelerators compatible with multicore programming frameworks (e.g. Spark and MapReduce), iv) development of a run-time scheduler responsible for controlling accelerators utilization treating power efficiency and resources flexibility and v) hardware accelerator virtualization in a open source ecosystem facilitating innovative enterprises.

Finally, Nowatzki et al. [35] analyzed that efficiency is sacrificed for programmability through the implementation of domain specific accelerators (DSAs). Exploiting Dennard scaling and Moore's Law led to the design of hardware architectures that are capable of performing computations for a specific domain achieving high performance and energy efficiency. On the other hand, these architectures give up on programmability because of the constantly evolution of the algorithms which are based on. Additionally , the need of flexibility across workloads between different devices make DSAs inefficient. Build on this weakness they proposed a programmable architecture that contains a low-power core, a spatial architecture, scratchpad, and DMA (LSSD) [36]. Based on a set

of common specialization principles named the “5C’s “ concurrency, computation, communication, caching, and coordination specialization matched DSA performance with two to four times the power and area overhead while retaining reprogrammability. Based on their approach, in this thesis a generic framework for efficient remote accelerator deployment in disaggregated data centers is going to be implemented. It is based on a simplified LSSD-like hardware architecture on the remote platform and exhibits concurrency, computation, communication, caching, and coordination through a set of Advanced co-processor instructions (ACI).

Chapter 4

The ReFiRe framework

This thesis mainly focused on the implementation of the ReFiRe [1] framework. The ReFiRe allows to efficiently deploy remote/disaggregated accelerators by improving the computation-to-communication ratio between a host processor and an arbitrary number of accelerator devices. This is achieved by relying on complex instructions of variable length, henceforth referred to as Advanced Co-processor Instructions (ACIs), which describe partial reconfiguration events and the required flow of data among a set of remote partially reconfigurable accelerator cores. Figure 4.1 below illustrates the outline of the ReFiRe framework. It consists of three main parts which analyzed below:

1. The ACI support library: Provides an easy to use Application Programming Interface to the users in order to encapsulate software application instructions in (ACI) format.

2. Advanced Co-processor Instruction (ACI): A set of custom instructions for Host-Hardware accelerator communication, synchronization and remote accelerator deployment.

3. Advanced Co-processor Instruction decode control: A software implemented finite state machine (FSM) handler that decodes the ACI and controls the reconfigurable area.

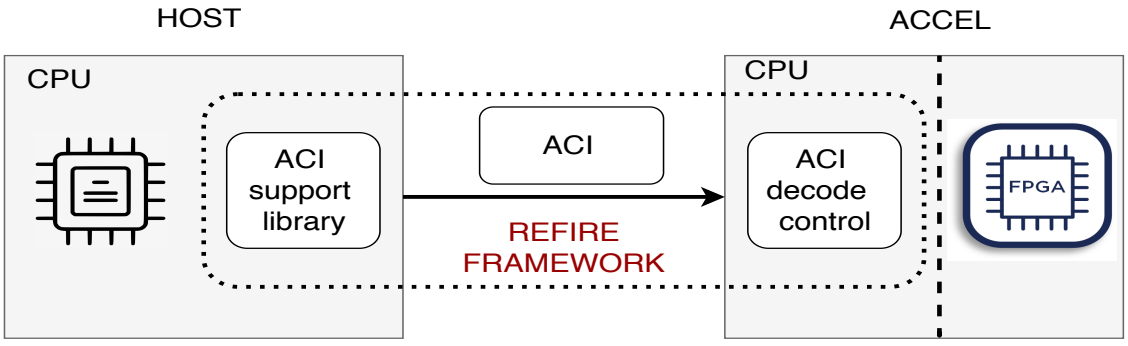

Figure 4.1: ReFiRe framework overview.

4.1 Hardware accelerator architecture

An overview of the ReFiRe hardware architecture that corresponds to the Field-Programmable Gate Array (FPGA) part of the ACCEL device (Figure 4.1) is illustrated in Figure 4.2. It consists of an array of S Reconfigurable Accelerator Slots (AS), with each AS being a partially reconfigurable region (PRR) that could accommodate the accelerator core. The datapath Constructor (DC) modules are responsible for interconnecting the ASs according to each application requirements. Each DC which provides AXI4-Stream interface includes a crossbar switch and FIFO blocks for data exchanges between the ASs. Thus, the output of each AS is going to be forwarded to the next one without the need of external memory accesses. Each accelerator core may be characterized from a set of parameters that remain the same during the execution. The Parameter File (PF) constitutes a memory block that accommodates these configuration parameters which are retrieved through the ACI. The data Fetch and Dispatch (DFD) unit handles the data transfer in between the main memory and the accelerator cores. For every constructed datapath the DFD unit is responsible for transmitting them from the memory at the point of entry and writing the output form the last accelerator's core point of exit to the appropriate memory space defined in the application. Each of these units corresponds to a distinct AS and consists of a DMA engine that controls input and output transactions. Depending on the application requirements, input data to each AS can derive either from on-chip storage or from external memory (through DFD units). An array of multiplexers are responsible for controlling the above mentioned input/output destination. The DC, AS, DFD and multiplexer arrays are configured and controlled by the processing system on the accelerator hardware based on information extracted from the ACI, which is stored in the ACIMEM memory block as illustrated below.

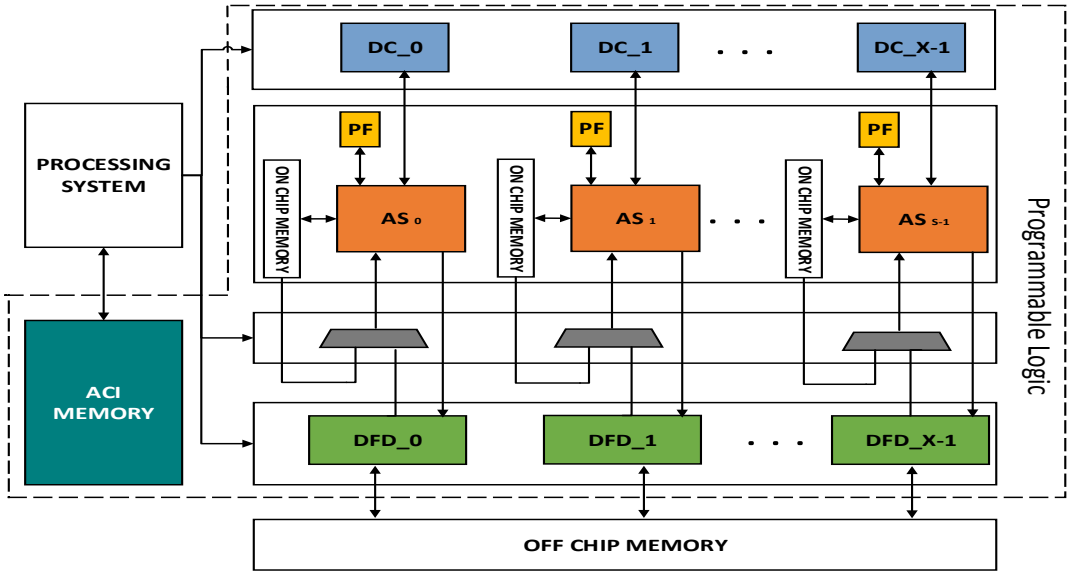

Figure 4.2: The ReFiRe architecture.

4.2 Advanced Co-processor Instruction (ACI)

An ACI is a complex instruction of variable length that encapsulates several degrees of flexibility in deploying a fixed number of fine-grained accelerator cores for an arbitrary number of tasks. It consists of three main parts, namely *Sync*, *Compute*, and *Param*. The *Sync* area facilitates host-accelerator synchronization via the exchange of basic control signals (e.g., start/stop), progress counters, and status codes. The *Compute* part includes the entire computational load that an ACI carries, adopting a hierarchical organization of basic elements of five class types: the `TASK`, the `LOOP`, the `THREAD`, the `PARSEC`, and the `WINDOW`, illustrated in Figure 4.3. The *Param* area contains parameter values and corresponding PF locations per task. The five *Compute*-specific classes are described below.

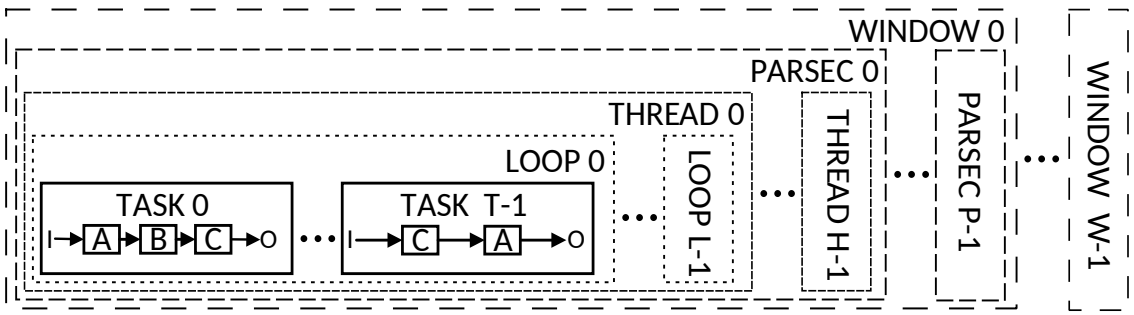

Figure 4.3: Hierarchy of ACI *Compute* classes, providing an example of two TASK configurations for a set of three accelerator cores, A, B, and C. I (input) and O (output) per task are served by a dedicated DFD.

`TASK:` This class represents a set of interconnected ASs. In a `TASK` class data are retrieved from a predefined specific memory area and when the acceleration procedure finishes the output is written back to memory. For every `TASK` class, unique opcodes initialize DFD, DC and multiplexer configuration. For the DFD unit, specific opcodes related to input/output start addresses and the data stream size according to each application requirements included in this class. The destination of the intermediate data as well as the datapath of the accelerators are also encapsulated in order to properly configure the DC and multiplexer units. Furthermore, for the PF initialization a set of address offsets points to a specific region in the `SYNC` area.

`LOOP:` This is a container class for `TASK` objects. This class accommodates `TASK` class objects that are going to be executed sequentially. `LOOP` class is characterized from the required number of iterations and a stride which corresponds to a particular memory access pattern that had to be applied per iteration.

`THREAD:` This is a container class for `LOOP` objects. In this class the number of `LOOP` class instances that execute sequentially are defined. Each `LOOP` class is defined by a unique id with the

corresponding offset that points to the memory area where LOOP characteristics are defined.

`PARSEC:` This is a container class for `THREAD` objects. Via `PARSEC` class, `THREAD` objects could execute concurrently. In this class a parallel section is initialized containing the number of the ASs that had to execute simultaneously according to each application demands. Next, for every AS a DFD and DC module is instantiated for the concurrent data transfer to the accelerator cores.

`WINDOW:` This is a container class for `PARSEC` objects. In a `WINDOW` class the partial reconfiguration of the available AS with the appropriate accelerator cores is established. These cores remain in the AS until all the objects of the `WINDOW` class are executed and a next WINDOW class relies on different partial reconfiguration of the ASs. Through this class the ASs could be reconfigured at acceleration run-time providing the capability on reducing the overall partial reconfiguration overhead by resizing the amount of the required computation into the appropriate `WINDOW` class objects.

4.2.1 ACI Specialization principles

As already stated, the proposed framework is build on the five specialization principles suggested by Nowatzki et al. [35]. Concurrency, computation, communication caching and coordination specialization are these five principles. Concurrency refers to the ability of the operations to be performed simultaneously. To achieve that the appropriate hardware architecture had to be designed and configured properly in order to accommodate the parallel execution. Usually many independent processing elements with their corresponding synchronization support mechanisms are deployed. Dealing with hardware architecture, this principle increases the total performance of the system for parallel workloads while broadens area and power demands. Computation principle refers to the district units that an algorithm is composed of. Problem specific Functional Units (FUs) used to cope with the specializing computation. It improves performance with low power demands when some commonality between domains exists. The communication specialization focuses on the data transmission between the FUs and the storage units. As proposed faster throughput to the FUs could be achieved by instantiating communication channels and buffers between hardware units. Data caching specialization points to the data reuse. In hardware acceleration where the access patterns are known apriori constitutes an important aspect for the efficient deployment. Finally, coordination is the management of hardware units and mainly involves the implementation of finite state machines.

In comparison with ReFiRe's infrastructure, data caching is achieved through the FIFO blocks in each DC module as well as the on chip memory where data could be stored there without the need for of chip memory accesses. Computation is exhibited in the `TASK` from the accelerator cores, while communication and coordination between the ASs is exposed with the DC modules by deploying the crossbar which initiates the appropriate datapath. Finally, concurrency is exposed

through the `PARSEC` and `THREAD` classes that enables the simultaneous execution of the accelerators cores.

The above proposed specialization principles could also be efficiently applied in different hardware architectures like GPGPUs or FPGAs. Likewise, the run-time partial reconfiguation that is transparently supported through the `WINDOW` class in ReFiRe constitutes a mechanism that fully exploit the reconfigurability feature which is a very important factor in such devices and could further improve the provided flexibility and capability.

4.2.2 ACI memory components

4.2.2.1 `SYNC, COMPUTE and PARAMETER` area

As described above, the main part of the ReFiRe framework is the Advanced Co-processor Instruction memory (ACI). The sync area constitutes a memory space in the accelerator side for the communication with the Host device. In this area, information related to accelerator status is represented. Recurrently, both `HOST` and `ACCEL` read from specific memory space in order to be informed about the current system's status. Accelerator's status states could be *ACTIVE, INACTIVE, RESERVED, BUSY, IDLE and FREE*. According to these states, the host writes in a predefined memory space the next desirable state of the accelerator according to each application requirements while informed about the current state if zero or one occurs in a specific area of the corresponding state.

The Compute area includes execution related information in a fixed format as illustrated in the Figure 4.4 below. The beginning and the end of each ACI class, as well as the appropriate values needed for the application execution is mapped in this area using specific opcodes in order to be recognized from the decode control in the accelerator side. The first line of the Compute area informs the ACI decode control about the available slots that are going to be reconfigured in a specific *WINDOW* class. Next, if one or more reconfigurable regions needed to be placed in the FPGA simultaneously and the number of the appropriate thread classes are defined. If one or more *THREAD* classes had to be initiated, the *thread offset* opcode notifies the accelerator side for the specific memory region that each thread corresponds to. Each thread consists of a fixed number of *LOOP* classes. Details relevant to the iteration number as well as the data stride are included afterwards. The input/output address and size of the remote data, as well as the configuration parameters required for the *TASK* class. Specifically, the configuration parameters are stored in the *param area* in a prearranged format as shown in the figure above. The *Parameter offset* informs the decode control where these task specific parameters are stored in consonance with the compute area region.

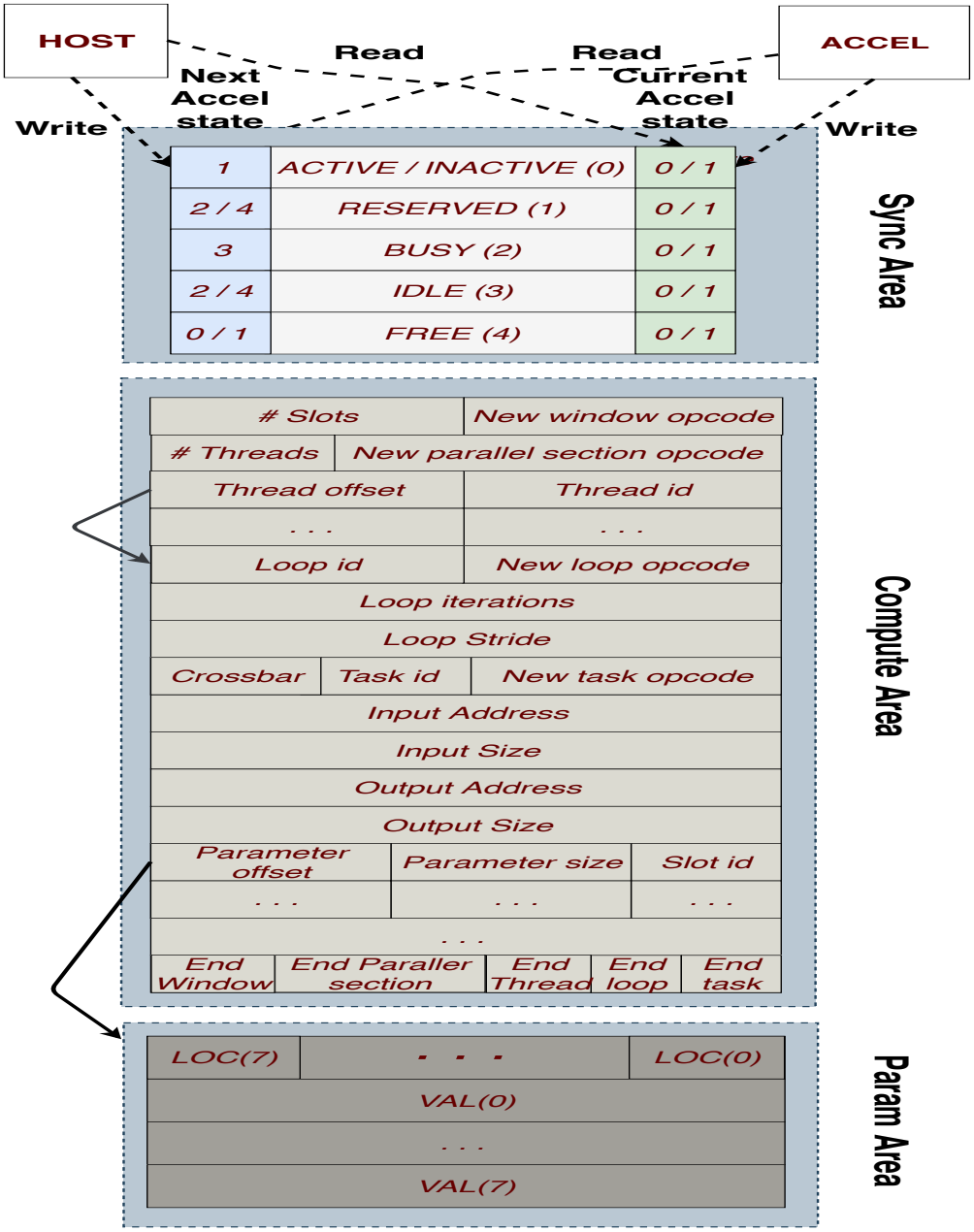

Figure 4.4: ACI memory component overview.

4.3 Host device

4.3.1 ReFiRe Application Programming Interface (API)

Writing a ACI based application often requires high complexity and excessive requirements from the programmers side. The ReFiRe framework provides an expressive and flexible Application Pro-

gramming Interface (API) for ACI class initialization according to each application requirements. Because of the high complexity of the instructions that had to be encapsulated in the compute area of the ACI memory, the implementation of an easy to use API was a major challenge. To achieve that a set of library routines implemented for the placement of the appropriate instructions in the ACI descriptor so as to be decoded and executed from the remote accelerator's decode control.

The pseudocde below displays the sequence of function calls that had to be executed in order to create an ACI descriptor that contains a single *WINDOW*, *THREAD*, *LOOP* and *TASK* class. Initially, five different pointer types that correspond to these classes are declared. In the *Create_Desc* function descriptor pointer is initialized. Next, each of the available hardware accelerators had to be placed in the appropriate available partial reconfigurable slot. *Place_Acc_In_As* constitutes the function for the above procedure. Thereafter, the *Create_Window* function initializes a window pointer for this descriptor. Similarly, *Create_Thread, Create_loop and Create_Task* functions used for *THREAD*, *LOOP* and *TASK* class initialization. *Create_loop* input arguments are also the iteration number of the loop as well as the data stride per iteration. Via the *Set_Input* function the input data address and size are declared in the descriptor. Likewise the location and the size of the output data are declared in the *Set_Output* function. The *Set_Crossbar* function is responsible for configuring the interconnect in order to create the appropriate accelerator datapath.

```
#include "ACI_support_lib.h"


int main(void)
{

Descriptor* Descriptor_ptr;//ACI Descriptor pointer initialization
Window* window_ptr;//Window pointer init.
Thread* thread_ptr;//Descriptor pointer init.
Loop* loop_ptr;//Thread pointer init.
Task*  task_ptr;//Task pointer init.
Place_Acc_In_As(Descriptor_ptr,slot_id,accel_id);//Accelerator placement in AS
Descriptor_ptr=Create_Desc(); // ACI descriptor generation
Window_ptr=Create_Window(Descriptor_ptr);//WINDOW class initialization
Thread_ptr=Create_Thread(Window_ptr); //THREAD class int.
loop_ptr=Create_Loop(Thread_ptr,loop_iteration,loop_stride);//LOOP class init.
task_ptr=Create_Task(hardware_or_software_task_id,loop_ptr);//TASK class init.
Set_Input(task_ptr,input_address,input_size);//TASK specific parameters
Set_Output(task_ptr,output_address,output_);
Set_Cros_Conf(task_ptr,crossbar_id);
Set_Task_Acc(task_ptr,accel_id);
Set_Acc_Param(accel_id,loc,vall);//Param area components


}
```

Listing 4.1: ACI application example

Finally, using the *Set_Task_Acc()* function the appropriate accelerator is placed in the corresponding *TASK* class. The parameters are forwarded to the accelerator side using the *Set_Acc_Param()* function. The fists argument corresponds to the accelerator id while the second and the third to position and the value of the parameter vector.

4.3.2 Source-to-source transformation framework

Although through ACI the synchronization between a Host CPU and the remote accelerators is reduced, it is rather difficult for the programmers to encapsulate their work in an ACI due to the complex hierarchical architecture. To this end, a framework that handles this task transparent to the programmers is necessary.

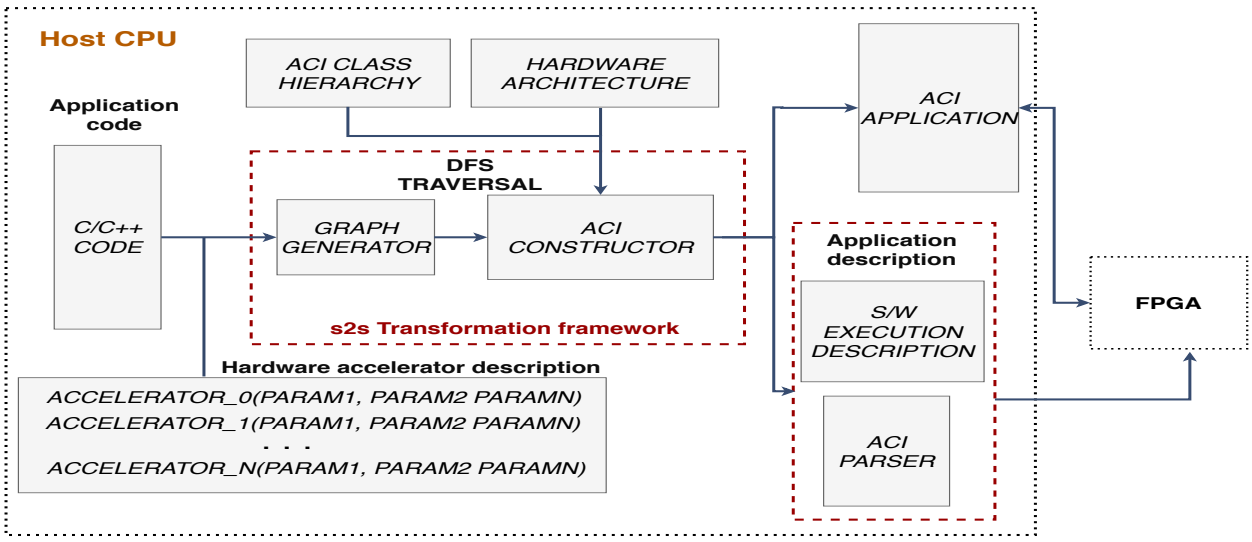

Figure 4.5: Source-to-source transformation framework overview.

As shown in Figure 4.5, the source-to-source (s2s) transformation framework executes on the Host CPU side. It has as input a description of the available hardware accelerators and the related application. The graph generator module of the s2s takes as input that application and the hardware accelerator description,(where the hardware accelerators are defined in the form of application related functions) and generates the corresponding graph. After parsing the graph using the Depth First Search traversal, the result is inserted in the ACI constructor. The ACI constructor constitutes a mechanism for mapping the available nodes of the graph to the compute part of an ACI. The construction of an ACI depends on the available hardware architecture and the ACI class hierarchy. During the implementation of the compute area, all the graph nodes are mapped into the appropriate ACI classes. The outcome of the framework is an ACI based Application and the application description that consists of software execution description and the ACI PARSER.

4.3.3 The ACI constructor

The ACI constructor (which executes on the host CPU) takes as input the result of the DFS traversal of the application graph and maps all the graph nodes into the appropriate classes of the ACI. Figure 4.6 below, exhibits the flowchart of the ACI constructor in order to initiate the appropriate TASK classes. As an assumption for the ACI constructor a terminal node is defined as node in a graph with no children. Moreover a node to an application graph is equal to a hardware accelerator or a software function that is between two hardware nodes. In order to encapsulate each node of the graph in the correct class, the ACI constructor performs a number of analysis steps. At first, regarding the TASK class, it checks whether a node or terminal or not. If a node is terminal, it means that the TASK class that has been assigned to, can't accommodate other accelerators. Furthermore, if a node corresponds to a software function then it has to be assigned in a unique TASK class. During the next step, the number of the children nodes is analyzed. If a node has more than one child, its output had to be written to the appropriate memory space in order to be used from its children nodes. If a node has only one child, then its child node (accelerator) can be assigned to the same TASK. The hardware architecture allows creating a pipeline with a specific number of accelerators, according to the available reconfigurable slots. In that case, the output of the first accelerator is going to be used as input to its child node without the need of external memory accesses. These steps are going to be applied in every node of the application graph. Similarly, according to each node properties the ACI constructor is going to initialize the LOOP class (based on the flowchart depicted in Figure 4.7 below). TASK nodes with the same iteration number are assigned to the same LOOP class. When all the available accelerators slots have been initialized, the next node is going to be attached to a new WINDOW class.

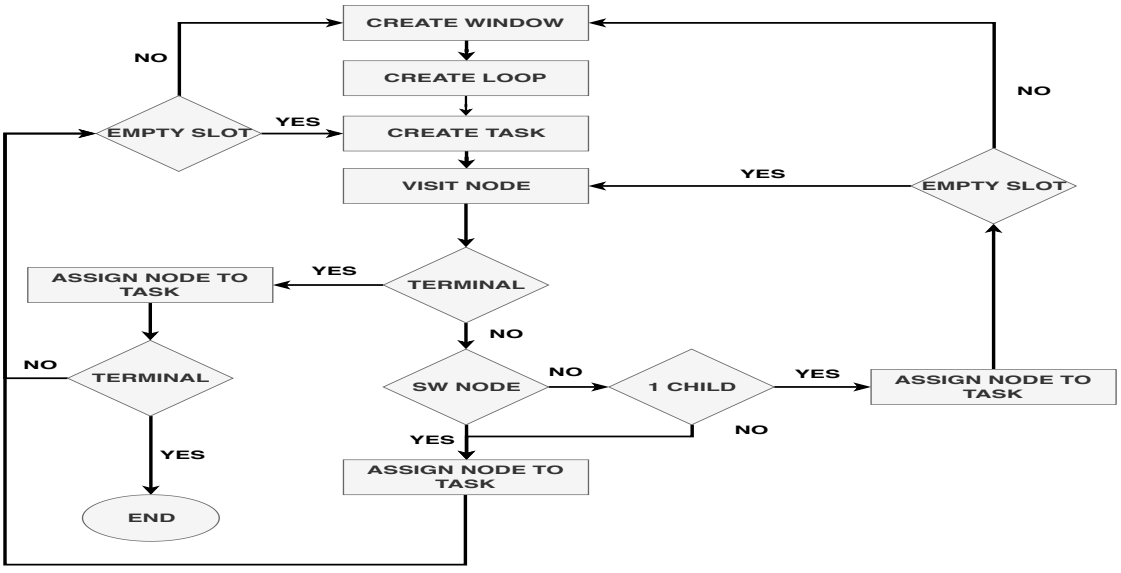

Figure 4.6: The ACI constructor flowchart (for the TASK class).

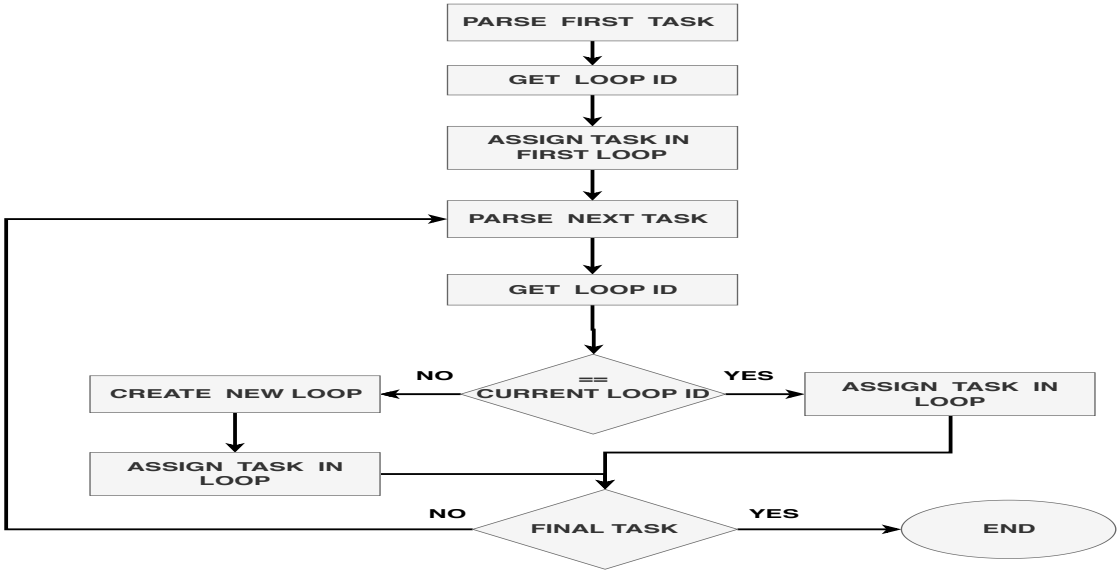

Figure 4.7: The ACI constructor flowchart (for the LOOP class).

Figure 4.7 above illustrates the ACI constructor flowchart for the LOOP class initialization. Aforementioned, the output of the first ACI constructor step is a TASK graph. Each node in the graph consists of a set of accelerators with a specific loop id. During the first application parsing, for each accelerator a unique loop id assigned according to the loop sequence they belonged to. If one or more accelerators belong to the same iteration block, they are marked with the same loop id. Next, based on this assumption, the ACI constructor parses that graph and gets the loop id of each node. If two continuously nodes had the same id then they are attached in the same LOOP class, otherwise new LOOP class had to be initiated to accommodate the next TASK.

4.3.4 The application description

The most important part of the s2s framework is the construction of the application description which contains the application source code that is going to be executed from the processing system of the remote FPGA. Initially, the graph generator creates the application graph according to each application properties. Consequently, the ACI constructor will generate the appropriate application description which consists of the ACI PARSER and the software execution description, as illustrated in Figure 4.8. The ACI PARSER is responsible for decoding the ACI that has been sent from the host CPU which contains all the instructions related to the application execution. As shown in Figure 4.8 *(ACI PARSER)*, initially the WINDOW of the ACI is going to be parsed. During that step, the partial reconfiguration regions are going to be configured.

If there are any parts of the application that are going to be executed simultaneously on the FPGA's ASs, they will be recognized while parsing the THREAD class. In the LOOP class,

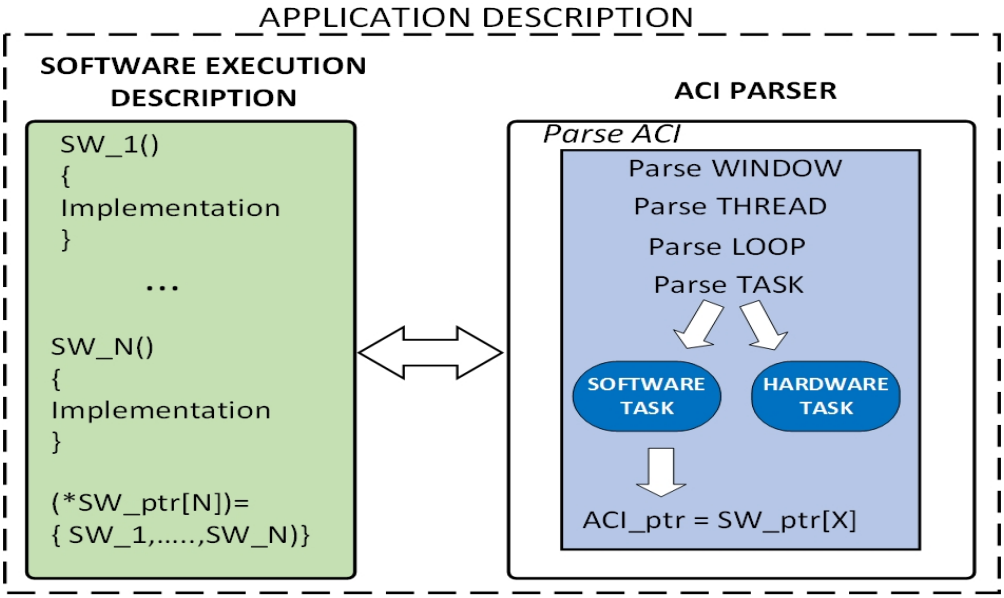

Figure 4.8: Application description components.

information related to the execution iterations of the containing tasks is included. This way, extra synchronization between host CPU and FPGA is redundant. Finally, in the TASK class information related to the accelerator pipeline is embodied. A TASK may contain an accelerator or a set of accelerators with the appropriate interconnect between them according to each application requirements. Without the software execution description in the remote FPGA, only code related to the hardware accelerators could be executed. Thus, if a software function was in between two accelerators, it has to be executed from the host CPU. According to the assumption that data are on the accelerator side, they had to be transferred on the HOST CPU side and after the software execution the outcome should be transferred in the acceleration side again. As a result, extra overhead related to data transfer time is added to the application execution. To overcome that, the s2s framework generates the software execution description. It contains all the function implementations that are going to be executed between the accelerators. Moreover, information related to the order of the software function execution is included in the ACI PARSER. A pointer to the related software function is initialized to the ACI PARSER from the s2s framework in the corresponding software TASK. When the ACI PARSER recognizes software TASK, the s2s transformation software had already initialized that pointer to the corresponding position of the function pointer table in the software execution description. Hence, the appropriate software function is going to be implemented as a software TASK in the ACI on the remote side.

4.3.5 ACI application mapping

The significance of the source-to-source transformation framework was depicted in the Figure 4.5 above. Having as input a software application that contains the hardware functions that could be executed from the accelerator device will construct the ACI hierarchy class representation which corresponds to sync, compute and param area application related instructions.

Pseudocode below represents an example application while Figure 4.9 below illustrates the ACI class hierarchy class exported from the source-to-source transformation framework. The first application is going to be executed from the programmable logic of the remote FPGA without the processing system invocation (for software function execution). Each *HW_accel_i()* function of the application pseudocode corresponds to a hardware accelerator call (e.g FFT, matrix multiplication etc.).

According to the source-to-source execution procedure, initially, the graph generator maps each hardware accelerator call to a graph node. Next, each of those nodes (as analyzed in the previous section) is going to be encapsulated in an ACI class. Assuming hardware architecture with four partial reconfiguration regions all of the available accelerators can be placed in a single WINDOW. Likewise, due to the same number of execution iterations are assigned to the same LOOP class.

1st Application Pseudocode

```
int main(){
HW_accel_0(int* in_0, int in_size_0, int* out_0, int out_size_0);
HW_accel_1(int* out_0, int out_size_0, int* out_1, int out_size_1);
HW_accel_2(int* out_1, int out_size_1, int* out_2, int out_size_2);
HW_accel_3(int* out_1, int out_size_1, int* out_3, int out_size_3);
}
```

Regarding TASK classes, three of them are instantiated. The first one contains the chaining between the Hw_accel_0 and the Hw_accel_1. Due to the fact that the output of the 2nd accelerator is used from the 3rd and the 4th accelerator it had to be written to the main memory. According to this technique, the 2nd and the 3rd TASK input data will be retrieved from the memory. Otherwise, the first two accelerators should be executed twice for the 3rd and the 4th accelerator individually, which would increase the overall execution time.

For the second application, which pseudocode is illustrated below software functions and accelerators with recursive calls evaluated. In more complex synthetic applications (pseudo code is shown in below) where more accelerators needed for the evaluation, in order to be encapsulated in an ACI, more than one WINDOW classes should be instantiated. A WINDOW can contain a fixed number of accelerator cores according to the available partial reconfiguration slots. In the example below, assuming hardware architecture with 4 partial reconfiguration slots may contain up to 4

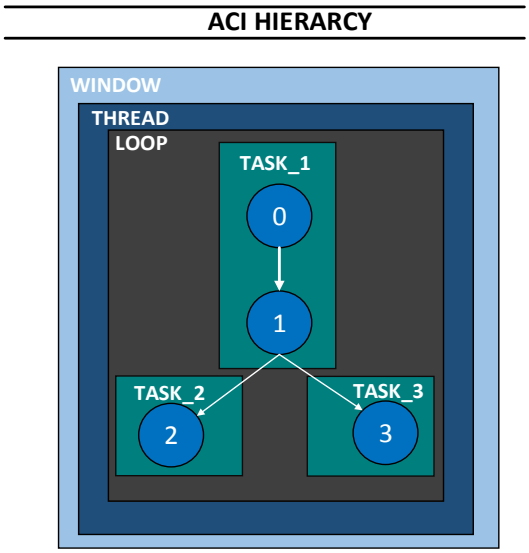

Figure 4.9: ACI class hierarchy for the 1st application.

accelerators. Moreover, the software functions between accelerators lead to different ACI hierarchy construction. Each software function had to be encapsulated in a single TASK class. For the first WINDOW, as shown in Figure 4.10, due to the same number of iterations, all the available TASKs are going to be attached to the same LOOP class. Each TASK contains only one accelerator, on the grounds that each them have more than one child or they are terminal nodes.

```
2nd Application Pseudocode
 int main(){
 HW_accel_0(int* in_0, int in_size_0, int* out_0, int out_size_0);
 HW_accel_1(int* out_0, int out_size_0, int* out_1, int out_size_1);
 HW_accel_2(int* out_0, int out_size_0, int* out_2, int out_size_2);
 HW_accel_3(int* out_2, int out_size_2, int* out_3, int out_size_3);
   for(i=0; i<N; i++){
 HW_accel_4(int out_2[i], int out_size_2, int out_4[i], int out_size_4);
 HW_accel_5(int out_4[i], int out_size_4, int out_5[i], int out_size_5);
   }
   for(j=0; i<M; j++){
 HW_accel_6(int out_5[j], int out_size_5, int out_6[j], int out_size_6);
 SW_func_0(int out_6[j], int out_size_6, int out_s0[j], int out_size_s0);
 HW_accel_7(int out_s0[j], int out_size_s0, int out_7[j], int out_size_7);
   }
  HW_accel_8(int* out_6, int out_size_6, int* out_8, int out_size_8);
}
```

For the second WINDOW class, the first two accelerators are placed in the same LOOP because they execute with the same iteration number. Moreover they are members in the same TASK class. This way, the output of the 4^{th} accelerator is going to be used from the 5^{th} immediately without the need of memory accesses. The same dispatch should be applied to the 6^{th} and 7^{th} accelerator cores. Because of the software application which is between them, they are assigned in different TASK classes. Similarly, the 8^{th} accelerator is assigned in a new WINDOW class due to the fact that all the available slots in the previous windows were reserved.

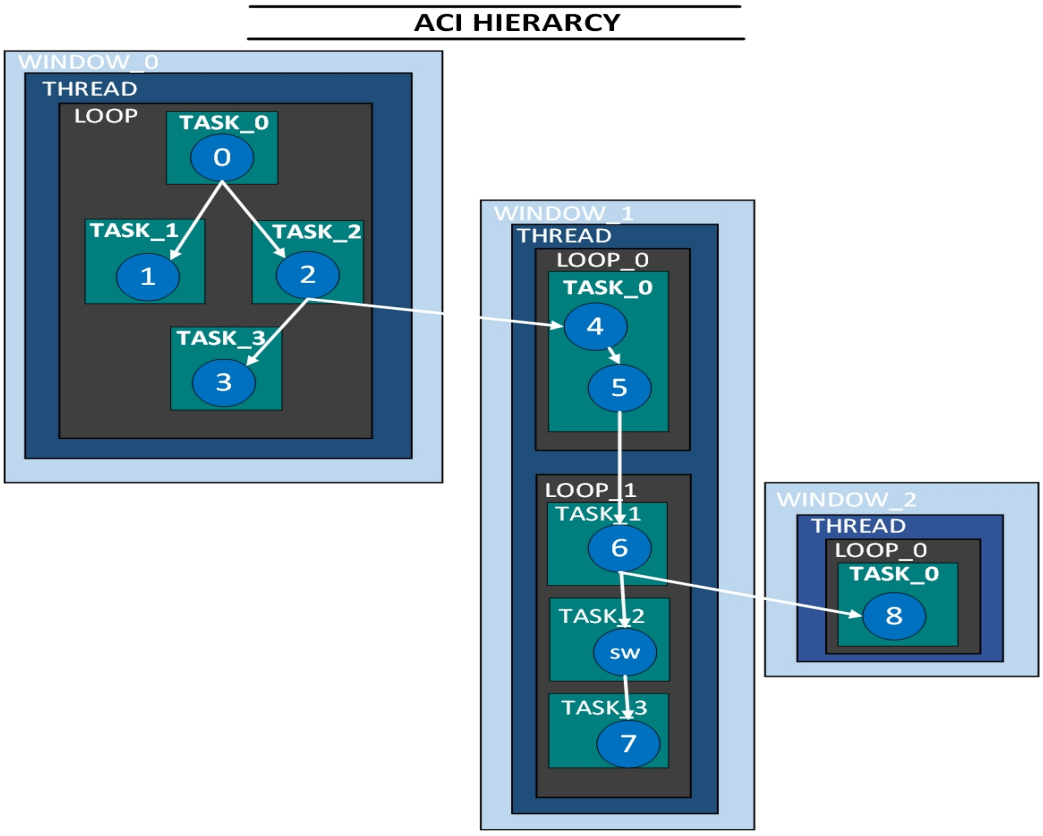

Figure 4.10: ACI class hierarchy for the 2^{nd} application.

4.4 Accelerator device

4.4.1 ACI decode control

The ACI decode control is the most important part of the accelerator device because it controls the remote accelerator states according to the received instructions from the Host CPU. As shown

in Figure 4.11 below, these states are divided into three different categories namely *Accelerator status, programmable logic configuration and accelerator execution* states. For the first category, the ACI decode control performs and endless sync area reading procedure in order to serve Host CPU commands regarding the desired accelerator's state. Initially, the accelerator device is in *INACTIVE* state, where the hardware part of the accelerator device hasn't been configured. In this state, it can be configured from any remote Host device. When a Host device wants to initiate a connection with the accelerator device forwards an ACTIVATE command. If the accelerator state changes form *INACTIVE* to *ACTIVE* then the connection is established. Thus, the Host device reserves the accelerator device. When it occurs in the *RESERVED* state it can't be used from other remote devices preventing any communication and synchronization deadlocks.

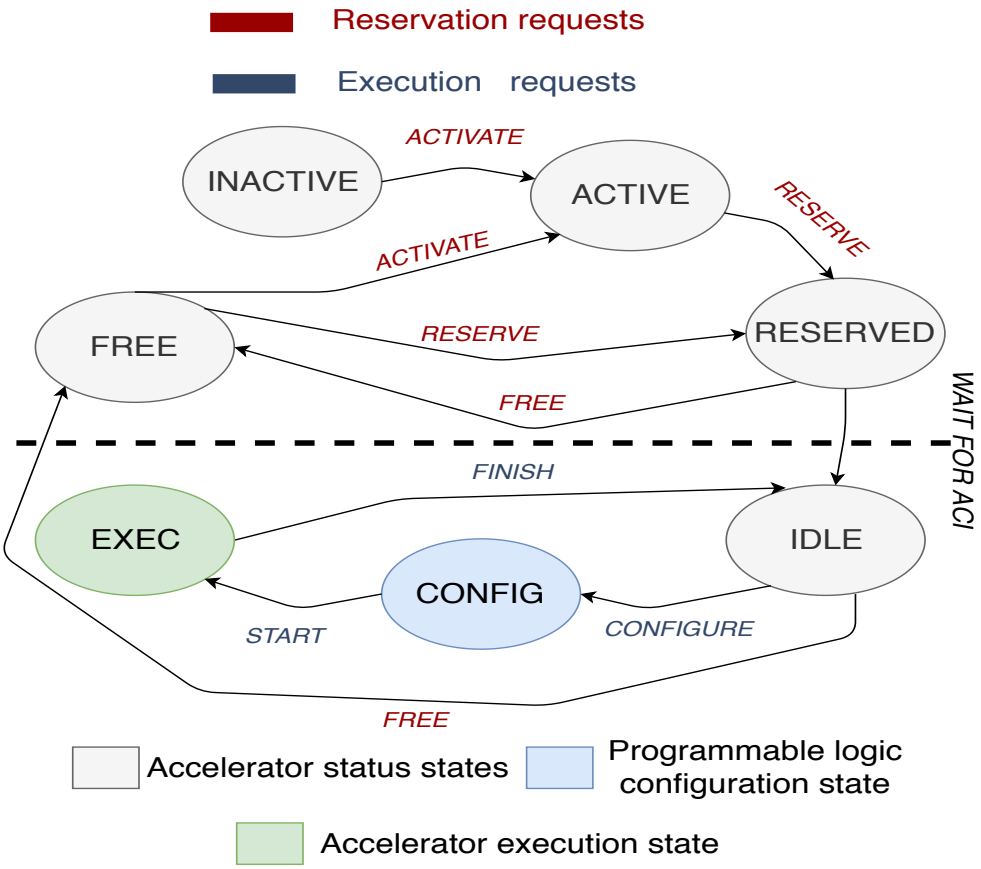

Figure 4.11: Remote accelerator state overview.

When the accelerator is reserved from a host CPU, it moves to the *IDLE* state where the ACI decode control waits for ACI in order to proceed in the *CONFIG* state. In this state, the ACI decode control decodes the Compute and Param area and according to the ACI instructions configures properly the programmable logic. In this state, a set of configuration steps had to be

implemented according to each application requirements before the execution start. These steps analyzed below:

1. Initialize ICAP[37] controller.
2. Place the appropriate partial bitstream in the corresponding partial reconfigurable region.
3. Initialize AXI4-Stream Switch.
4. Initialize DMA Interrupt Controller.
5. Start the DMA engine for the data transmission to the accelerator slots.
6. Retrieve the output data and place them in the destination source.

Following the application of the previous declared steps the accelerator device transmits to the *EXEC* state where the hardware accelerator execution occurs. When the execution completes the accelerator device carries on the *IDLE* state. When the Host device recognizes *IDLE* state, by applying the *FREE* command in the sync area, it releases the remote accelerator so as to be reconfigured for a different application or deployed from a different device. Otherwise, if a new ACI arrives from the same host device the above mentioned procedure is going to be repeated.

4.5 ACI sequence diagram

Figure 4.12 illustrates the required sequence of interactions for offloading a single `TASK` to a TASK-specific configured datapath (TCD), assuming a master-worker deployment scheme. Primarily, the user application creates and populates an ACI, using the API that the ACI Support Library exposes. A subset of the library routines are shown in figure 4.12. Input arguments and return values are omitted for the sake of clarity. The library implementation enforces a priority on the valid sequence of invoked functions, yielding correct-by-construction ACIs (note steps 1 to 6 in figure 4.12). The ACI-construction process completes with the `finalize_aci()` function (step 7), which ensures that ACI data are present in memory before firing up the remote DC. This is a prerequisite for correctness since the steps of populating the ACI and transferring ACI data can be interleaved, as is also the case in the disaggregated-computing emulation platform employed here for verification and evaluation purposes.

On the remote-accelerator side, processing begins with the finalization of the ACI. The ACI Decode Control, a light-weight software implementation that is structured as a hierarchy of inter-dependent FSMs, parses the ACIMEM memory space element by element, performing the required control/configuration actions per class type. `WINDOW`, `PARSEC`, `THREAD`, and `LOOP` elements only

generate configuration actions, whereas `TASK` elements additionally require synchronization and monitoring, as they comprise computation. For each `TASK`, the ACI Decode Control initially allocates and configures a DC, therefore constructing the required TASK-specific configured datapath. Thereafter, PF initialization data are retrieved from the *Param* area and stored in PF blocks. Next, a DFD is reserved and configured with input/output start addresses and the data size. Upon DFD configuration, input data are fetched from memory and directed to the TCD's entry point, i.e., the first AS in the datapath. Output data come out from the TCD's exit point, i.e., the last AS in the datapath, which the DFD writes into the designated memory space. Throughout ACI processing, the ACI Decode Control updates specific *Sync* area locations to facilitate progress monitoring from the host side, which happens transparently to the operation of the DC. Note that, the ACI Decode Control ignores PR requests for RMs already deployed in a AS.

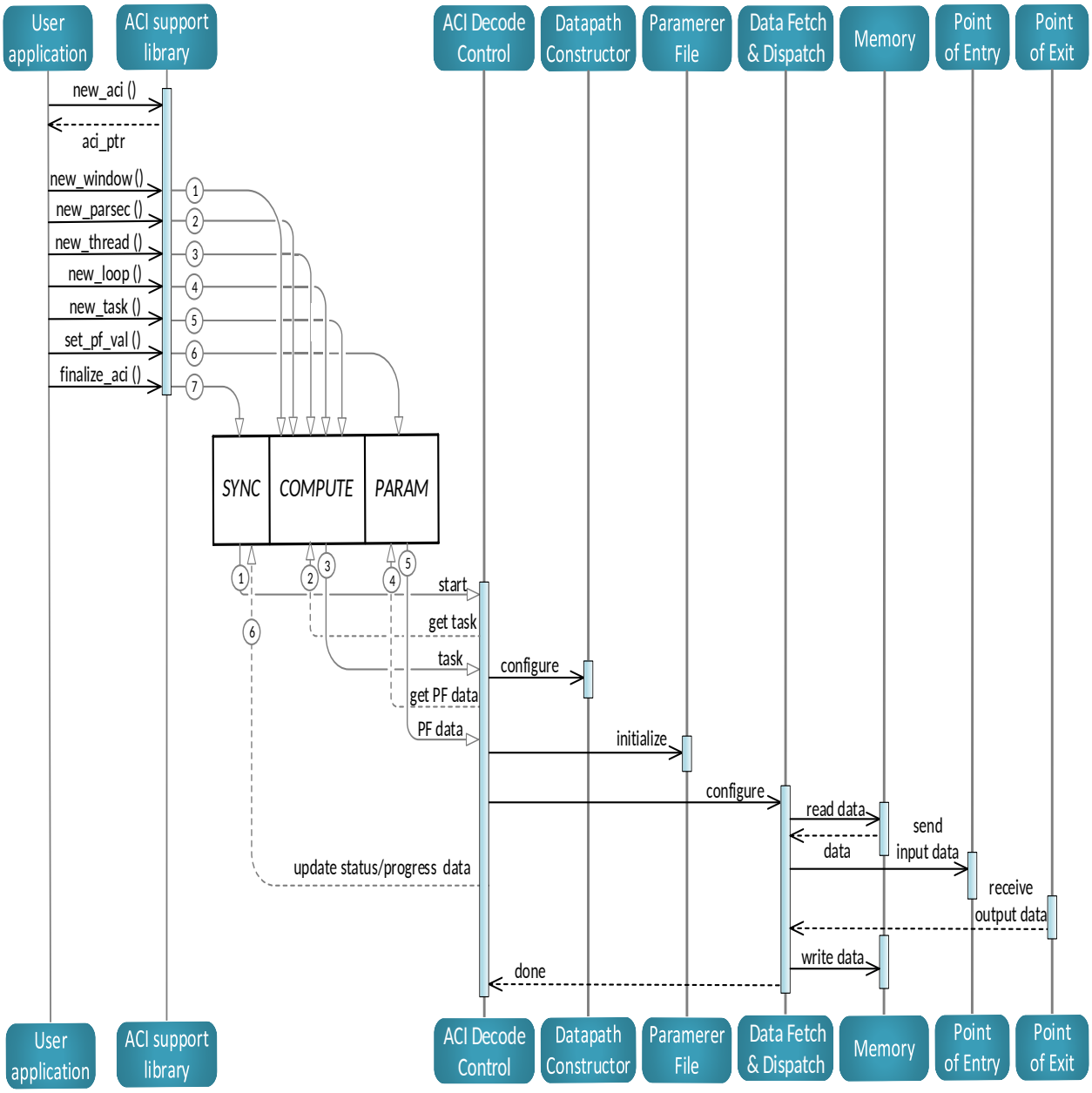

Figure 4.12: Sequence diagram of inter-node interactions for deploying remote hardware acceleration in a master-worker scheme using ACI.

Chapter 5

Evaluation

5.1 System implementation

In order to evaluate ReFire in a disaggregated environment, two ZCU 102 evaluation boards employed each hosting a Zynq Ultrascale+ MPSoC, interconnected over a Small Form-factor Pluggable (SFP) 10-Gbps link as shown in Figure 5.1 below. The disaggregated resource that provides the computing power will be referred as `HOST` and the hardware accelerator platform as `ACCEL`.

An ARM Cortex-A53 64-bit quad-core processor operating at 1.2 GHz is the Application Processing Unit (APU) on the `HOST` side running Ubuntu 16.4, while in the `ACCEL` side the ACI decode control runs as a baremetal application. The programmable logic of both devices constitutes of fixed amount of resources as depicted in table 5.1 below.

To facilitate data exchanges between `HOST` and `ACCEL` nodes, a set of extensions had to be implemented on the programmable logic of both nodes in combination with certain operating-system-level modifications [38]. Thus, data transmission is going to be accomplished transparent to the user level. This procedure requires the operating system mapping of logical physical addresses to remote memory segments, in order to make them available to user-space applications by altering its kernel's page tables to create a virtual map that associates local physical addresses to kernel virtual addresses. Consequently, the remote memory segments on the `ACCEL` subsist on the operating system's virtual memory management system and could be accessed at application level via system calls e.g. mmap().

In ReFiRe, the ACI memory block (Figure 4.4) is memory mapped, allowing the ACI to be transmitted transparently and in parts while it is being populated by the user application using the ACI Support Library (writing to corresponding virtual addresses). The observed, at the application level, memory bandwidth for remote write operations is 1,046 MB/sec, measured using the Sustainable Memory Bandwidth in High Performance Computers (STREAM) benchmark. Read and write operations of 64-byte chunks (Cortex-A53 cache line size) to remote memory locations introduce end-to-end latencies of 706 ns and 783 ns, respectively.

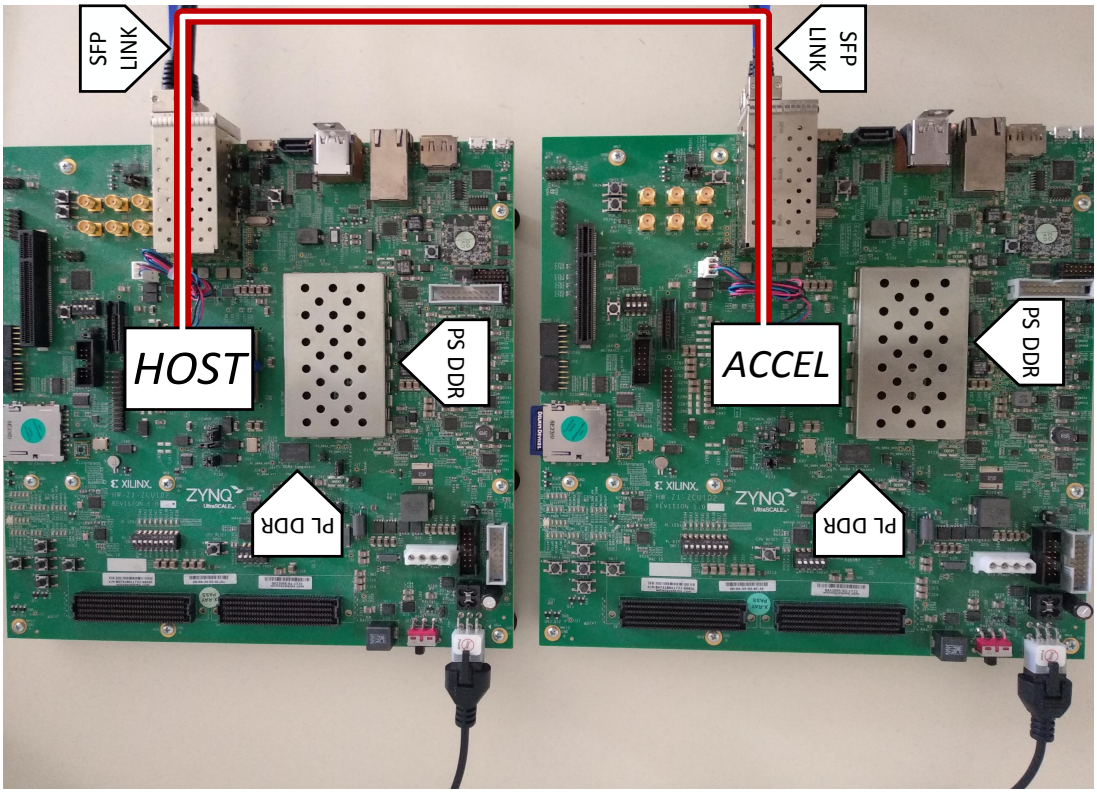

Figure 5.1: The prototype platform consisting of two ZCU102 boards interconnected through SFP-based link.

Table 5.1: Hardware resources of the ZCU102 Evaluation Board

Part:xczu9eg-ffvb1156-2-i		Size
Programmable functionality	System Logic Cells	600 (K)
	CLB Flip-Flops	548 (K)
	CLB LUTs	274 (K)
Memory	Max. Distributed RAM	8.8 (Mb)
	Total Block RAM	32.1 (Mb)
	UltraRAM	-

ReFiRe supports partial reconfiguration through the AXI Hardware Internal Configuration Access Port (HWICAP [39]) controller, which allows the `ACCEL` APU to read and write the FPGA configuration memory through the Internal Configuration Access Port (ICAP) primitive [37]. Prior to processing, the `HOST` transfers a set of partial bitstreams per AS/PRR to remote memory on the `ACCEL`, while at run time, the ACI Decode Control performs PR operations according to custom instructions received from the ACI memory.

5.2 2D-FFT accelerator for image processing

5.2.1 Application description

In image processing, the Fast Fourier Transform (FFT), an efficient implementation of the Discrete Fourier Transform, is used to transfer spatial-domain image data into the frequency domain, which facilitates several operations on images, such as noise removal, pattern recognition, and filtering. A filter, for instance, is applied to an image by a convolution in the spatial domain, whereas a multiplication is only required in the frequency domain. The 2D-FFT of an N -by-N image can be computed with an N number of N -point 1D-FFTs per dimension using the well-known row-column algorithm [3], i.e., by applying the N -point 1D-FFT first to every row and then to every column.

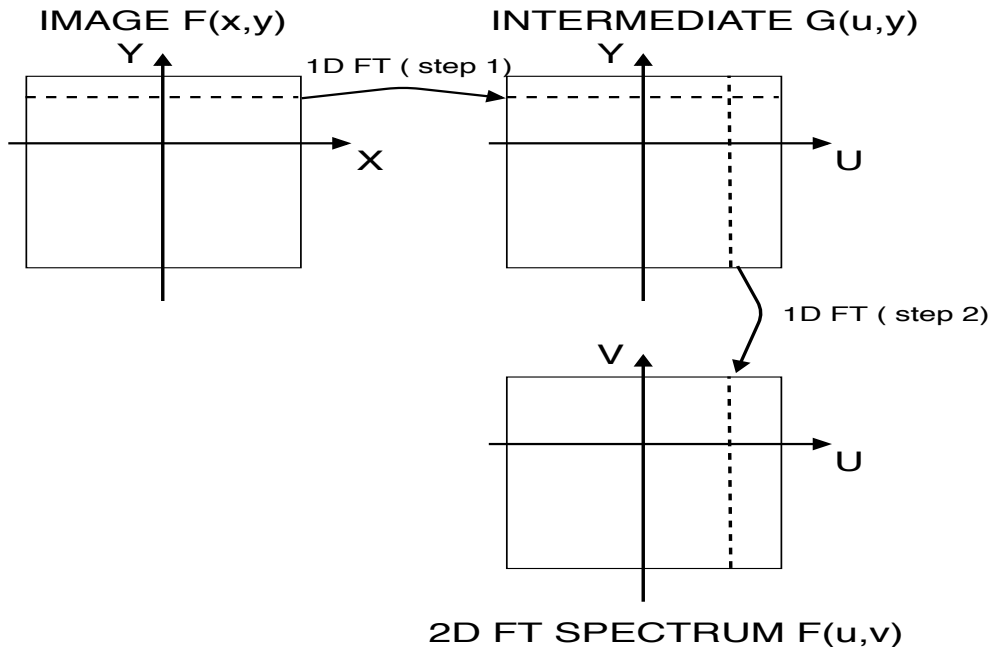

Figure 5.2: 2D FFT implementation using 1D FFTs.

As illustrated in Figure 5.2 above, a two dimensional transform can be separated into a series of one dimensional transforms. Initially, each horizontal line of the image is individually transformed and produces an intermediate form in which the horizontal axis is frequency (u) and the vertical axis is space (y). Hence, each vertical line of the intermediate image is also separately transformed in order to acquire each vertical line of the transformed image.

5.2.2 ACI application mapping

The aforementioned computation can be conveniently captured in a single LOOP object per dimension, where the iteration number and the data stride are set according to the image size N.

Figure 5.3 illustrates the ACI class hierarchy representation that had to be initiated in order to execute the FFT from the remote device. Initially, one *WINDOW* class containing one *THREAD* class initiated. Thereafter, two *LOOP* classes had to be designed, each containing a single *TASK* class where the accelerator execution is encapsulated. The first *LOOP* class relies on the first dimension accommodating 1D FFT related information (e.g iteration number and data stride) while the second class relies on the second dimension parameters. Without the required for-loop operations captured in an ACI, explicit synchronization (start/done) is required per iteration, i.e., the for-loops are implemented by the `HOST`. When the corresponding ACI functionality is exploited, via the `LOOP` class, a single ACI is created and transferred to the `ACCEL`, where the for-loops are implemented by the processing system.

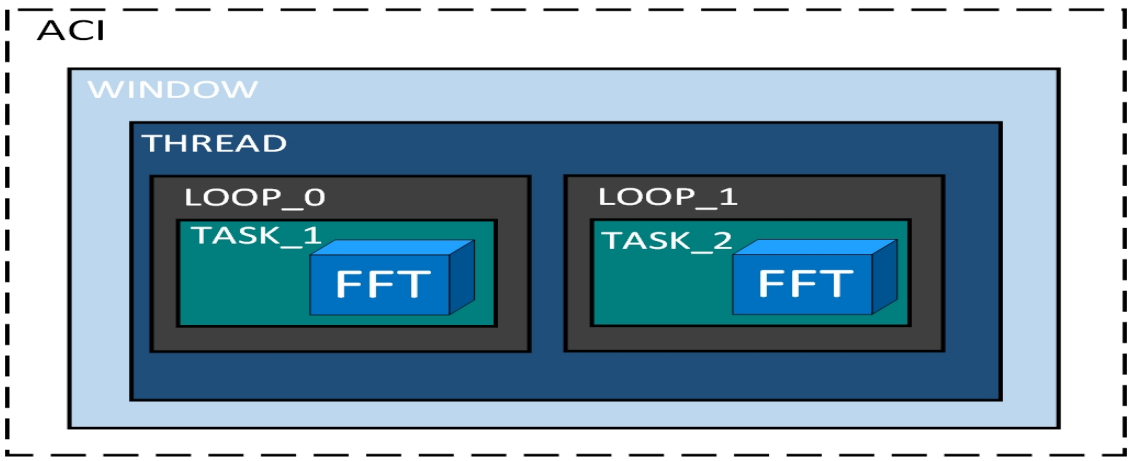

Figure 5.3: ACI class hierarchy representation that enapsulates 1D FFTs accelerator cores.

5.2.3 Experimental results

We create a ReFiRe design point with one DC, one DFD, and a two ASs each hosting a 1D-FFT core [21] for every dimension. Figure 5.4 below shows the attained performance improvement (speedup) for an increasing image (and FFT) size when the for-loops are described in the ACI in comparison with being implemented by the HOST. Different core throughput configurations are considered, i.e., 2 and 8 results per cycle, as well as local memory bandwidth rates for read/write operations, i.e., 2 GB/sec (the disaggregated-computing emulation platform described in Chapter 5) and 30 GB/sec (modelled considering an AC-510 SuperProcessor module with 4 GB of Hybrid Memory Cube and a Xilinx Kintex UltraScale XCVU060 FPGA serving as the ACCEL).

Figure 5.5 provides a time breakdown for the 1,024- point 2D-FFT, considering additional local memory bandwidth rates (16 GB/s) and core throughput configurations (4 and 16 results per cycle). The barplot reveals that accessing memory dominates execution time (case M2-T2, no ACI), with the percentage dropping as the memory bandwidth increases, and both computation (FFT

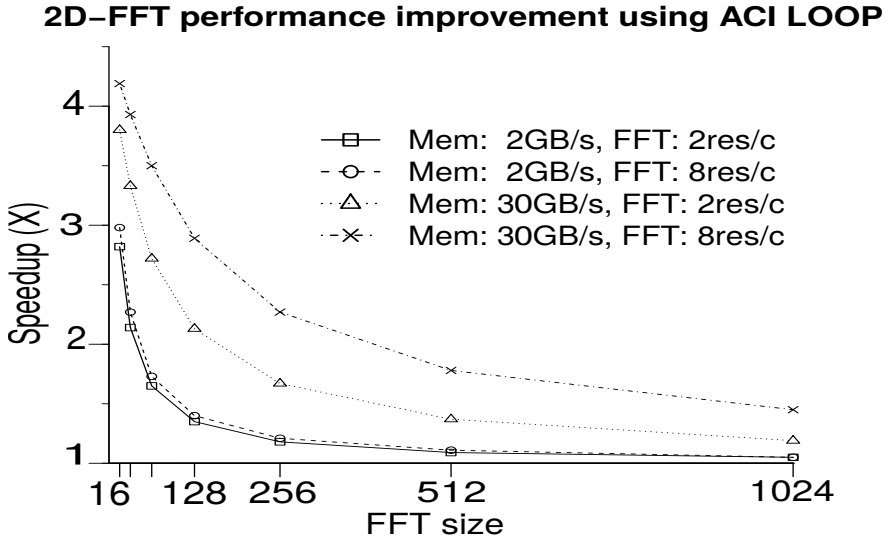

Figure 5.4: Attained performance improvement from encapsulating the for-loop operations in a single ACI via LOOP objects, rather than controlling them from the HOST via explicit per-iteration synchronization.

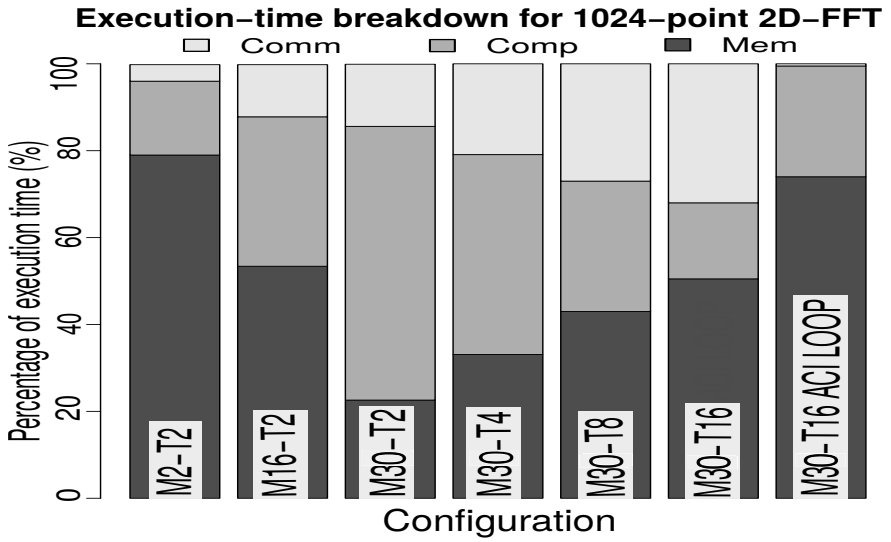

Figure 5.5: Execution time breakdown for 1024-point 2D-FFT, with and without the ACI LOOP functionality.

operation) and communication (synchronization) fractions increase (case M30-T2, no ACI). Core configurations with higher throughput decrease compute time (cases M30-T4 through M30-T16, no ACI), leading to synchronization eventually exceeding computation (case M30-T16). The final configuration (case M30-T16-ACI-LOOP) shows that exploiting the ACI LOOP class effectively eliminates the synchronization overhead ($\leq 0.5\%$) that would otherwise exceed computation and yield the deployment of the accelerator in a disaggregated environment impractical. Likewise, table

5.2 below, exhibits that ACI LOOP functionality could eliminate communication requirements for different 2D-FFT sizes considering core throughput equal to 4 results per cycle.

Table 5.2: Comparison between communication, computation and memory access percentage of execution time, for various 2D-FFT sizes, with and without the ACI LOOP functionality, assuming 4 results/cycle FFT core throughput.

2D-FFT	Percentage of execution time (%) Without ACI LOOP			Percentage of execution time (%) With ACI LOOP		
size	Communication	Computation	Memory	Communication	Computation	Memory
16	73%	2%	25%	2%	11%	87%
128	60%	14%	26%	0.15%	36.45%	63.4%
256	49%	24%	27%	0.15%	46.35%	53.5%
512	36%	34%	30%	0.1%	53.6%	46.3%

5.3 Detection of positive selection in genomes

5.3.1 OmegaPlus

5.3.1.1 Application description

Positive selection is a form of natural selection that is driven by beneficial mutations in a species population. A mutation is beneficial when it improves the chances of survival and reproduction for the carrier, leading to an increasing number of individuals having the beneficial mutation from generation to generation. When a beneficial mutation occurs, the amount of linked neutral mutations[1] diminishes, creating a so-called selective sweep [40]. Detecting selective sweeps, and thus positive selection, has practical applications such as detecting drug-resistant mutations in pathogens (e.g., HIV [41]) and improving the efficiency of drug treatments [42].

5.3.1.2 ACI application mapping

Here, we employ ReFiRe to accelerate OmegaPlus [4], a detection method that has been reported to outperform[2] other neutrality tests, e.g., SweepFinder [44] and SweeD [45], in terms of power to detect selection and reject neutrality [43]. OmegaPlus exhibits two major computational stages, the calculation of linkage disequilibria (LD) [46] and the calculation of ω-statistic values [46], with the contribution of each stage to the total execution time varying with input data and user parameters. An input dataset of S genomes and N polymorphisms (genomic locations with at least one mutation) is represented in memory as a $S \times N$ binary matrix D, where every non-zero entry indicates a mutation. The LD score between polymorphic columns i and j is calculated as:

[1] Mutations in the proximity of the beneficial mutation (linked) with no effect on the survival and reproduction of the carrier (neutral).

[2] Except for certain neutral non-equilibrium evolutionary models [43].

$$r_{ij}^2 = \frac{(p_{ij} - p_i p_j)^2}{p_i p_j (1 - p_i)(1 - p_j)}, \tag{5.1}$$

where p_i and p_j are calculated by dividing the number of mutations in columns i and j, respectively, with the number of genomes S, and p_{ij} is calculated by dividing the number of genomes with mutations in both i and j with S. OmegaPlus evaluates overlapping subgenomic regions in D, and computes an LD score for every pair of polymorphic columns per region. Since the enumeration of mutations (1's in D) can easily dominate the execution time when the number of genomes increases, OmegaPlus adopts a binary-vector representation scheme that allows to store each column as a number of 64-bit unsigned values. This allows to compute the number of mutations per column (or pair of columns) by iteratively performing population count operations, i.e., counting the set bits in a word. Thereafter, ω-statistic values are calculated based on LD scores per subgenomic region as follows:

$$\omega = \frac{(\binom{l}{2} + \binom{W-l}{2})^{-1}(\sum_{i,j \in L} r_{ij}^2 + \sum_{i,j \in R} r_{ij}^2)}{(l(W-l))^{-1} \sum_{i \in L, j \in R} r_{ij}^2}. \tag{5.2}$$

For this calculation, a subgenomic region than comprises W polymorphic columns is divided into two non-overlapping regions, L and R, with l and $W - l$ polymorphic colums, respectively. Eq. 5.2 is computed iteratively for all possible combinations of L and R sizes, and the maximum ω-statistic value per region is reported.

To evaluate ReFiRe, we modified the OmegaPlus source code to implement blocking in both LD and ω-statistic calculations in order to facilitate hardware design using the high-level synthesis (HLS) tool Vivado HLS. We created 3 reconfigurable accelerator modules, `ACCEL1`, `ACCEL2`, and `ACCEL3`, as well as a 3-RAS ReFiRe design point to host them. `ACCEL1` performs all the required population count operations for an 8×8 block of polymorphic columns, `ACCEL2` calculates LD scores (Eq. 5.1), and `ACCEL3` computes 32×32 ω-statistic values (Eg. 5.2). Table 5.3 provides resource utilization for the ReFiRe design point and the three accelerators.

As shown in Figure 5.6 below, for the ACI one `WINDOW` class that contains one `THREAD` class employed. Furthermore two distinct `LOOP` classes that contain two `TASK` classes (the first for the LD scores calculation and the second for the omega statistic values) initiated. The first one contains the pipeline between `ACCEL1` and `ACCEL2` while the second `TASK` class includes the `ACCEL3`.

.

5.3.1.3 Experimental results

The rationale behind designing two accelerator modules for computing Eq. 5.1, i.e., `ACCEL1` and `ACCEL2`, lies in the fact that the former's computational load increases with the number of genomes S, whereas the latter's increases with the number of polymorphic columns W in each subgenomic

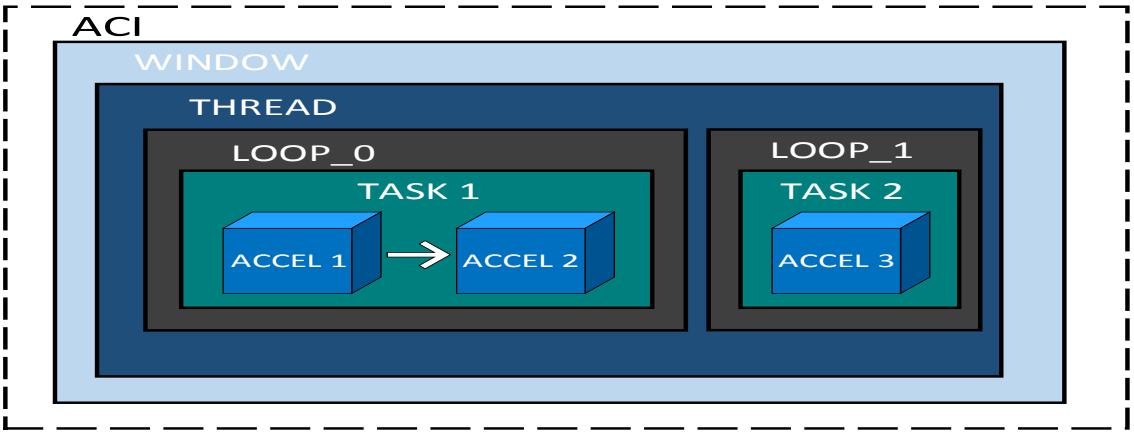

Figure 5.6: ACI representation for the calculation of LD scores and omega statistic values.

Table 5.3: Resource utilization for the 3-RAS design point and the three accelerators (ACCEL1-3) for OmegaPLus [4].

Hardware module		Resource type			
Name	Inst.	CLB LUTs	Register/SDR	BRAM18/36	DSP
ACCEL1	1	6,617	5,048	131/9	-
ACCEL2	1	5,546	4,139	1/2	13
ACCEL3	1	3,423	3,179	1/2	7
HWICAP	1	412	978	-/2	-
DFD (AXI_DMA)	3	2,650	3,318	-/18	-
AXIMM (PRR isolation)	4	208	-	-	-
AXIS-Inp. (PRR isolation)	4	147	-	-	-
AXIS-Out. (PRR isolation)	4	147	-	-	-
DC (crossbar)	2	1,505	2,589	-/18	-
PF (AXIBRAM Control)	8	305	209	-	-
PF (BRAM36 slices)	8	-	-	-/2	-
HOST logic (remote comm.)	1	14,399	27,034	15/112	-
ACCEL logic (remote comm.)	1	27,034	36,599	17/117	-

region. Thus, decoupling population count operations from LD calculations allows for more flexibility in accelerator deployment based on the characteristics of the input dataset, i.e., the number of genomes and the number of polymorphic columns (and their locations in the genome). Furthermore, this allows to inrcease the scope of usage for `ACCEL1` since representing elements as binary vectors and performing population count operations on them finds application in several domains. In chemical informatics, for instance, each bit in a so-called 2D fingerprint (binary vector) represents the existence or absence of a substructural fragment in a molecule. Enumerating the set bits

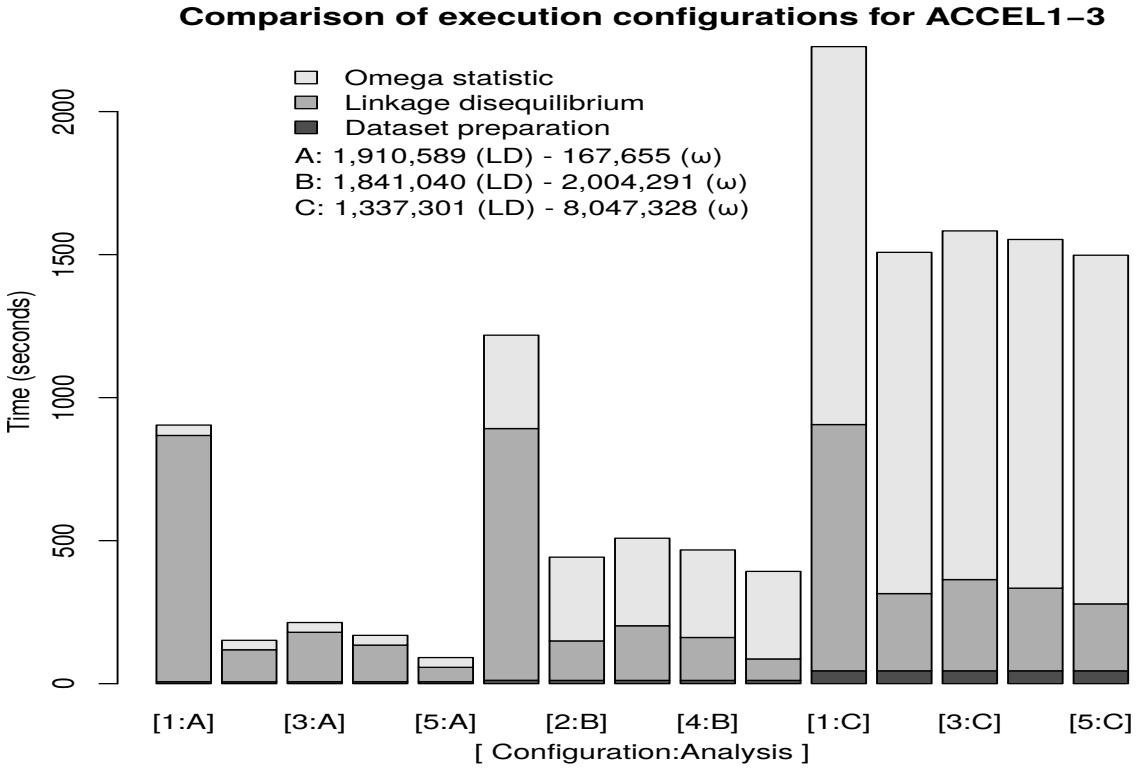

Figure 5.7: The effect of acceleration varies per configuration (table 5.4), demonstrating the performance role of the computation-to-synchronization ratio in remote accelerator deployment, and the ACI-enabled performance boost. Analyses A, B, and C are described in terms of total LD and omega values.

Table 5.4: Execution configurations for accelerators `ACCEL1-3`.

Configuration (Index: Name)	Environment (OS/Compiler)	Description
1: Reference	Ubuntu/gcc	OmegaPlus v. 3.0.3 (software only)
2: SDSoC	Pre-built*/sdscc	Invokes `ACCEL1-3` locally
3: ReFiRe	Ubuntu/gcc	Invokes `ACCEL1-3` remotely, creating one ACI per accelerator call
4: ReFiRe-A	Ubuntu/gcc	Invokes `ACCEL1-3` remotely, deploying `ACCEL1-2` in the same `TASK`
5: ReFiRe-B	Ubuntu/gcc	Invokes `ACCEL1-3` remotely, deploying `ACCEL1-2` in the same `TASK` in a `LOOP`

in 2D fingerprints is the basis for several chemical-similarity measures, e.g., Jaccard/Tanimoto and Dice.

We performed 3 genomics analyses (`A`, `B`, and `C` in Figure 5.7 with different OmegaPlus parameters based on a simulated dataset with 8,000 genomes and 20,000 polymorphisms (generated using

Table 5.5: Performance comparison between ReFiRe and SDSoC per algorithm stage (LD:`ACCEL1-2`, ω statistic:`ACCEL3`)

Application stage (metric)	Execution configuration [2]	[3]	[4]	[5]	Max. accel. performance
LD ($\times 10^3$/sec)	1,091.77	707.21	948.19	2,430.87	2,809.48
ω statistic ($\times 10^6$/sec)	5.218	5.032	-	-	49.089

the software ms [47]), and evaluated different accelerator-deployment scenarios with ReFiRe and Xilinx SDSoC (execution configurations, see Table 5.4). All configurations were compiled with all optimizations activated (-O3 for gcc and sdscc). Figure 5.7 illustrates execution times per algorithm stage for each of the execution configurations, whereas table 5.5 above provides a comparison in terms of throughput performance. As can be observed for LD, increasing the computation-to-synchronization ratio for remote accelerators by creating deeper pipelines (configuration [4], `ACCEL1` and `2` in a `TASK`) and offloading considerably more computations to a remote node (configuration [5], use of `LOOP` with 10,000 iterations) allows to expose near-peak accelerator performance at the application level (86.5%). The same does not hold for ω-statistic calculations, neither with ReFiRe (configuration [3]) nor with SDSoC (configuration [2]), due to the fact that random memory accesses, as required by `ACCEL3`, dominate processing and lead to poor accelerator performance (observed at the application level) of as low as 10% for both ReFiRe and SDSoC.

5.3.2 Linkage Disequilibrium calculation for DNA input data

5.3.2.1 Application description

Genetic variation is observed in the form of single-nucleotide polymorphisms (SNPs). A SNP results from one or more mutations at the same genetic location in a genome [48]. The allele encoding scheme is dictated by the mutation model, which can either be the Infinite Sites Model [49] or a Finite Sites Model [50]. The former assumes at most one mutation per site (1 bit per state), whereas the latter allows all possible DNA states at every site (4 bits per state). Linkage Disequilibrium (LD) calculation (equation 5.2), prerequisites binary input data. Dealing with DNA input data LD scores are calculated as follows according to [51].

$$r_{ij}^2 = \frac{(v_i - 1)(v_j - 1)v_{ij}}{v_{ij}} \sum_{si,sj \in S} r_{sisj}^2, \tag{5.3}$$

where v_i represents the number of existing states in SNP $i (v_i <= 4)$, v_j the number of existing states in SNP $j (v_j <= 4)$, and v_{ij} is the number of valid pairs of states $(s_i, s_{j \in S})$.

5.3.2.2 ACI application mapping

As referred above, `ACCEL1` and `ACCEL2` yield LD scores when input data are in binary format. Assuming there is not a hardware accelerator for LD score calculation with DNA input data, this process should be executed from the processing system of `HOST` or `ACCEL`. Without the extension that source-to-source transformation framework provides, the software process that is responsible for the LD calculation with DNA input data had to be executed in the `HOST` side. For the first step of the algorithm, based on the above assumption, LD scores (for binary input data) will be calculated in the `ACCEL` side using the pipeline of `ACCEL1` and `ACCEL2`. Thereafter, the `HOST` device would obtain the output through the remote segments and outcome LD scores for DNA input data. Exploiting source-to-source transformation framework's capability, the second step of the algorithm is going to be encapsulated in an ACI as a software TASK and be executed from the `ACCEL` device as shown in Figure 5.8 below. Based on the ACI class hierarchy, a single WINDOW class containing a TREAD class with two distinct LOOP classes instantiated. The first LOOP class contains a hardware TASK (pipeline of `ACCEL1` and `ACCEL2`) that corresponds to equation 5.1, while the second one accommodates a software implemented TASK (equation 5.3) for the LD score calculation with DNA input data.

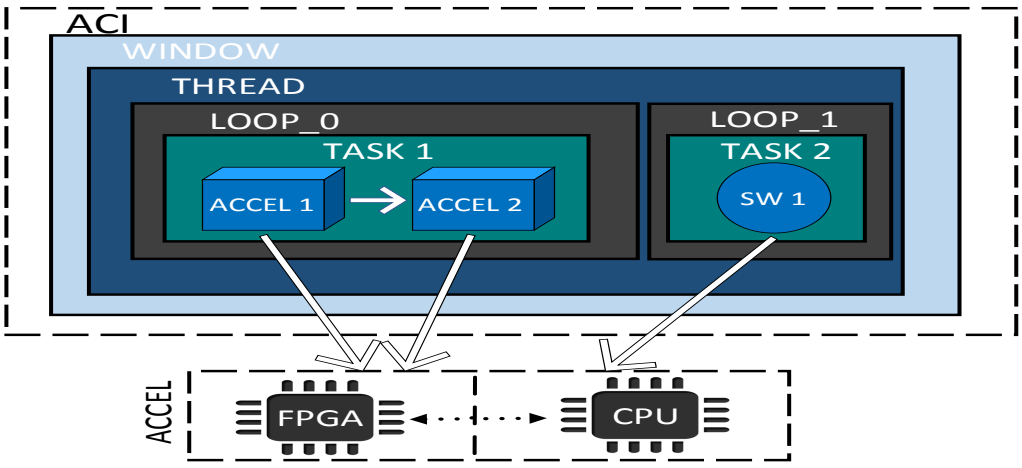

Figure 5.8: ACI hierarchy for calculating LD scores with DNA input data.

5.3.2.3 Experimental results

Figure 5.9 exhibits the overall execution time needed for deploying two different processing scenarios. For the first scenario (represented with a red line in the following plot), the software TASK is going to executed from the `HOST` side. Thus, for every algorithm iteration the synchronization between `HOST` and `ACCEL` is a prerequisite. On the other hand, for the second execution scenario

(represented with a blue line in the following plot), the software TASK is going to be encapsulated in an ACI and executed from the `ACCEL` among the hardware TASKs. Hence, regardless the required iterations, one ACI will be forwarded to the remote device containing the appropriate LOOP classes with the corresponding information, without the need for further synchronization with the `HOST` device. Thus, exploiting source-to-source framework's flexibility, the overall execution time could be boosted up to 2.5 times (according to the required algorithm iterations) in comparison with the first execution scenario where the software TASK is executed from the `HOST` device, as illustrated in figure below.

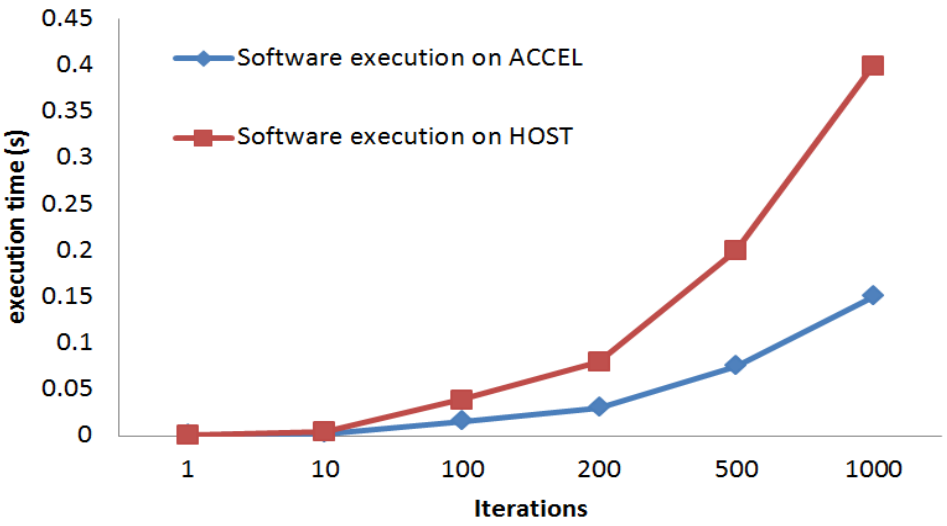

Figure 5.9: Comparison of software task execution on HOST and ACCEL device.

5.4 Binarized neural network

5.4.1 Application description

Artificial Neural Networks (ANNs) are computational models inspired by the way biological neural networks of the human brain process information. Within the domain of ANNs, there is an area frequently referred to as deep learning, where the employed neural networks typically exhibit between a few and more than a thousand layers. The most commonly used deep-learning network is the Convolutional Neural Network (CNN), which processes data in a grid-like topology. The high accuracy of CNNs was first demonstrated in the 2012 ImageNet recognition challenge [52]. Nowadays, CNNs are widely used in various domains, such as computer vision [53, 54] and artificial intelligence [55], leading to considerable advancements in speech recognition [56] for machine translation [57] and natural language processing [58], among others. In fact, CNNs have several practical applications, from self-driving cars [59] and autonomous aerial vehicles [60], to detecting

cancer [61, 62, 63] and playing complex games [64]. The superior performance of CNNs comes from their ability to extract high-level features from input data after using statistical learning over a large amount of data.

Considerable improvements in the development of high-performance systems for neural networks using multi-core technology have been proposed in recent years [65]. However, various challenges in power, cost, and performance scaling remain, due to the ever increasing model sizes (e.g., 50MB for GoogLeNet [66], 200MB for ResNet-101 [67], 250MB for AlexNet [52], or 500MB for VGG-Net [68]) that inevitably introduce prohibitively high computational costs, steadily raising the need for accelerated solutions using FPGAs and/or GPUs. The need for models with low memory and compute requirements is imperative.

Several works have been introduced to address the aforementioned challenges, and reduce the resource utilization requirements of CNNs, e.g., by exploiting the sparsity of the network connections [69], or by narrowing the data width [70, 71, 72]. Another promising method is binarization, which relies on a considerably more compact data representation for the network weights and the neuron values than the one employed by regular CNNs. The underlying idea is to constrain each value to be either +1 or -1. Consequently, this reduces storage and memory bandwidth requirements and allows to replace floating-point operations with binary operations, thereby paving the way for efficient deep learning using FPGA technology.

Binarized Convolutional Neural Networks (BNNs) was first presented by Courbariaux et al. [73], who introduced a method to train BNNs with the permutation invariant MNIST, CIFAR-10, and SVHN [74] datasets, achieving state-of-art accuracy. Rastegari et al. [75] successfully trained a BNN with ImageNet models, reportedly improving accuracy, boosting performance, and reducing the model size, when compared with a full-precision AlexNet [52] implementation. Existing implementations of CNNs on FPGAs face several challenges due to their prohibitively high requirements for storage, memory bandwidth, and compute capacity. This problem exacerbates with more complex state-of-art models, such as the VGG model [70] that has 16 layers and 138 million weights.

A typical CNN classifier consists of a parameterized pipelined multi-layer architecture. Layers require configuration of their parameters, often called weights, which must be determined by training the CNN offline on pre-classified data. Once the parameters are determined, the CNN can be deployed for the classification of new data points. The first layer takes as input a multi-channel input image and outputs a set of feature maps (fmaps). Each of the following layers read the fmaps, performs some computation on them, and produces a new set of fmaps to be fed into the next layer. Finally, a classifier produces the probability of that image belonging to each output class. The layer types are the following:

Convolutional layers realize a filter-like process, convolving the input fmaps with a $K \times K$ weight kernel. The results are summed, added with a bias, and passed through a non-linearity function to produce a single output fmap. This process is given in Equation 5.4 below,

$$y_n = f(\sum_{m=1}^{M} x_m * w_{n,_m} + b_n). \tag{5.4}$$

Pooling layers map each input fmap to an output fmap where every pixel is the max/mean of a $K \times K$ window of input pixels. They are inserted through a CNN to reduce the size of the intermediate fmaps.

Fully-Connected layers apply a linear transformation on the input 1-D vectors with a weight matrix. A bias is applied on the result, which is then passed through a non-linearity function to produce a single 1×1 output. This process is given in Equation 5.5 below,

$$y_n = f(\sum_{m=1}^{M} x_m * w_{n,_m} + b_n). \tag{5.5}$$

A BNN is essentially an extremely quantized, reduced-precision CNN model where weights and fmap pixels are binarized using the sign function. Positive weights are mapped to +1 and negative weights to -1, using a compact single-bit representation. Therefore, BNNs require significantly less storage than standard CNNs. The binarization of the neural networks can either be partial of full. In order to be consider as full, it has to encompass the following aspects: binary input activations, binary synapse weights and binary output activations. Due to the quantization effect, there is no need for biasing since it does not compromising the accuracy. However, in order to enhance the accuracy and scale down the error, a new layer type has to be introduced:

The **Batch normalization** [76] layer reduces the quantization error of the binarization by linearly shifting and scaling the input distribution to have zero mean and unit variance. The transformation is given in Equation 5.6 below,

$$y = \frac{x - \mu}{\sqrt{\sigma^2 + \epsilon}}\gamma + \beta. \tag{5.6}$$

The CIFAR-10 dataset [77] contains sixty thousand 32×32 3-channel images consisting of photos taken of real world vehicles and animals. For the experiments, out of the 60,000 images, 50,000 images were chosen for training and 10,000 images for testing. Training of the CIFAR-10 BNN model was done using open-source Python code provided by Courbariaux et al. [73], which uses the Theano and Lasagne deep learning frameworks.

5.4.2 ACI application mapping

The architecture of the BNN consists of nine layers, with the first six being convolutional layers while the next three are fully connected layers, as illustrated in Figure 5.10. The first layer (L0 in Fig. 5.10) receives fixed-point input data and binary weights, whereas the rest of the layers (L1 through L8 in Fig. 6.9) operate only on binary data. The convolutional layers rely on 3×3 filtering and edge padding, while the fully connected layers apply batch normalization prior to pooling, and

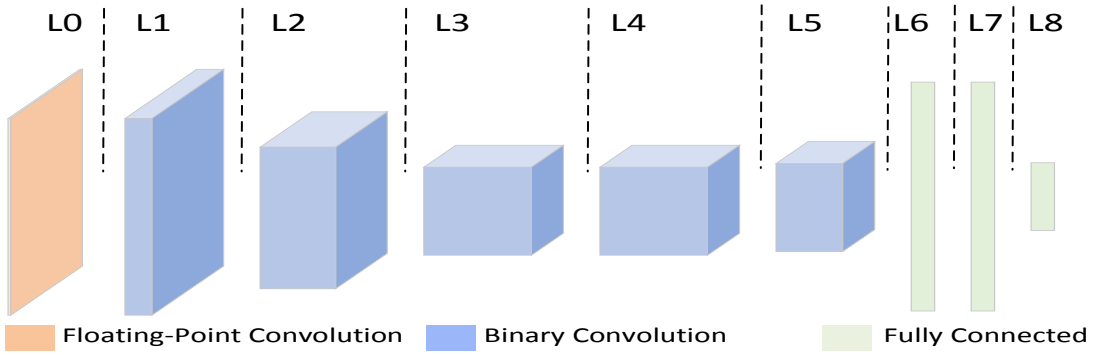

Figure 5.10: Binarized Neural Network architecture.

binarization before writing data out to the buffers. The accelerator system presented by Zhao et al. [5] designed three accelerators, which are employed as-is in our disaggregated accelerator systems. The *FP_CONV* core implements the L0 layer of the BNN. The *BIN_CONV* core is employed for the following five binary-only convolution layers (L1 through L5). Finally, the *BIN_FC* core accelerates the last three BNN layers (L6 through L8).

To map the required BNN computations to an ACI, we place each accelerator call in a dedicated TASK class, which also contains the respective core's configuration parameters and input/output address and sizes. The number of images that are processed in-between PR events is defined as the number of iterations of a LOOP class, with the stride being the image size. The THREAD and PARSEC classes allow to expose parallelism per layer by partitioning processing over multiple AS that host the same accelerator core. Finally, the WINDOW class performs one PR event per AS to deploy a different accelerator core to serve the needs of the next BNN layer. Due to the fact that there are three accelerator cores, the final ACI that implements the BNN consists of three WINDOW classes, one per accelerator core. Figure 5.11 below, illustrates alternative execution scenarios based on different ACI structures for the BNN. The *Static_Architecture* is identical to the reference execution scenario that is implemented on a software-programmable FPGA by Zhao et al. [5]. Due to the fact that ReFiRe is a native partially reconfigurable architecture, the *Static_Architecture* involves the initial deployment of the three accelerators in three AS. This is achieved by placing all nine TASK classes (one per BNN layer) in the same WINDOW class. The *PR_Architecture* exploits PR at run time and exposes intra-layer parallelism through the PARSEC/THREAD classes. Therefore, thee WINDOW classes are required, one per accelerator core, and multiple ACI-based iterations are performed.

5.4.3 Experimental results

All three accelerator cores deployed in ReFiRe are retrieved from *https://github.com/cornell-zhang/bnn-fpga*. Table 5.6 provides resource utilization per accelerator on the ZCU102 evaluation platform.

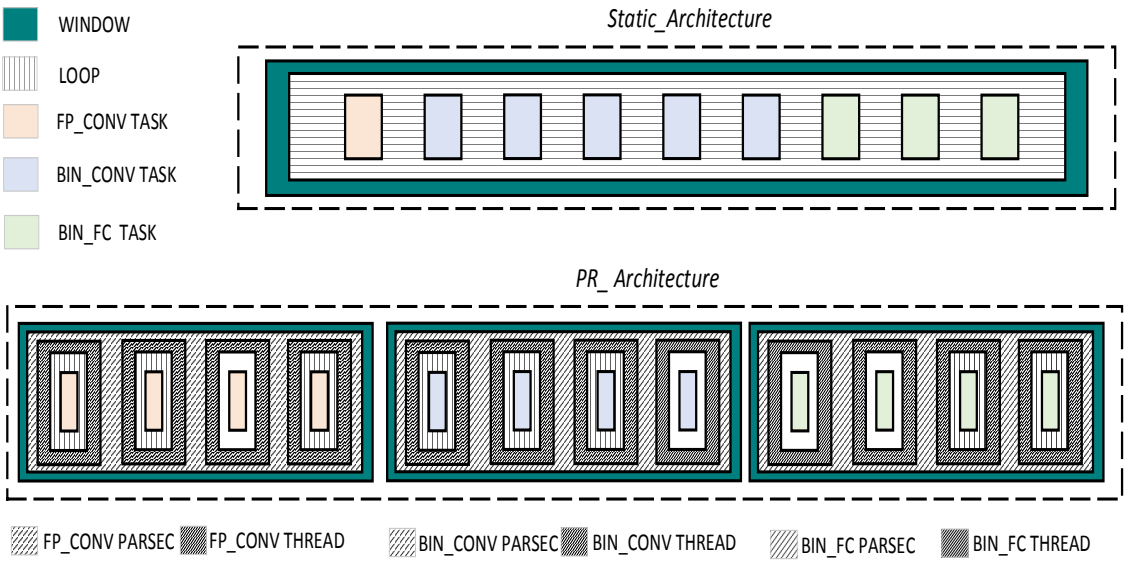

Figure 5.11: Illustration of the ACI format for the *Static_Architecture* and the *PR_Architecture* for FPGA-based BNN acceleration.

We evaluate two alternative execution scenarios, the *Static_Architecture* and *PR_Architecture* (illustrated in Figure 5.11) using the CIFAR-10 dataset.

5.4.3.1 *Static_Architecture*

We initially reproduce, using ReFiRe, the same static execution scenario that was evaluated by Zhao et al. [5] using SDSoC. Thus, we first deploy the accelerator cores FP_CONV, BIN_CONV and BIN_FC through an initial configuration WINDOW. In this scenario, all 10,000 images we used for evaluation are processed sequentially, directing the each layer's output to the next, as dictated by the BNN architecture. This approach required 128 sec to complete. As a reference, we note that the SDSoC-based approach [5] using the exact same accelerators and amount of images required 103.1 sec. The observed delay is due to data exchanges between remote FPGAs for ACI transfers and synchronization.

5.4.3.2 *PR_Architecture*

Next, we evaluate the DPR-based execution scenario by populating all AS with the same accelerator core and rely on the PARSEC and THREAD classes to invoke them in parallel per layer. The DPR overhead per AS (using ICAP [37]) is 7 ms (2.5 MB bitstream sizes). Note that, Zhao et al. [5] report 5.7 ms per image without using DPR. Thus, to yield a beneficial computation-to-PR ratio to exploit DPR using ReFiRe, we organize processing in batches. Figure 5.12 illustrates how performance improves with the batch size. As can be observed in the figure, DPR allows to outperform the

Table 5.6: Resource utilization for the three BNN accelerator cores on the Zynq Ultrascale+ MPSoC

ACCEL.	LUTs	FFs	BRAMs	DSPs	Power (W)
FP_CONV	11609	13802	16	0	0.112
BIN_CONV	13208	5849	86	2	0.050
BIN_FC	4432	6148	20	2	0.086

fully static architecture when the batch size exceeds 25 images/batch. Evidently, processing a single image in-between DPR events yields the worst-case performance, requiring 917 seconds in total for the 10,000 images, when the static design with 1 instance per accelerator finishes in 128 seconds. When the batch size exceeds the 300 images, DPR allows up to 3.1x faster execution, due to the four accelerator instances per layer. Note that, aggregate system performance increases almost linearly with the number of disaggregated FPGAs used, due to the beneficial computation-to-synchronization ratio that the ACI offers. The overhead to create and transfer an ACI to a remote FPGA is as low as 1.33 sec.

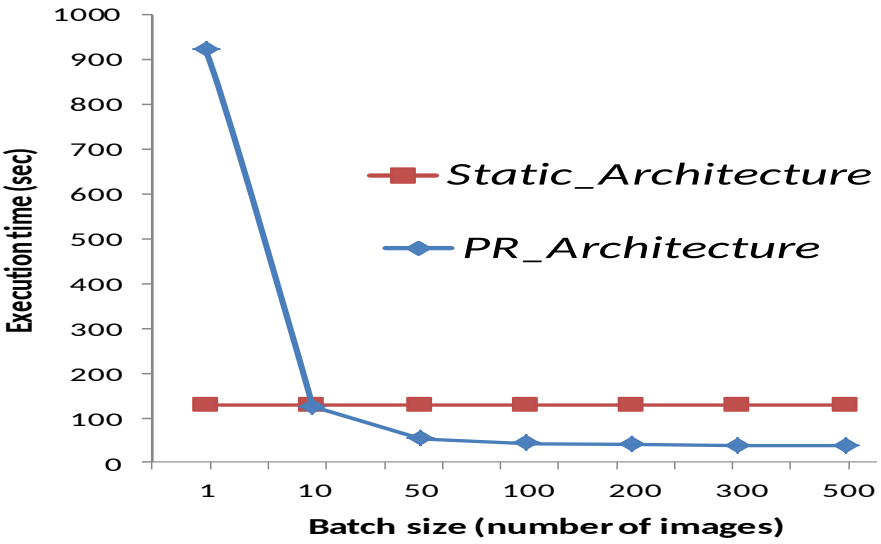

Figure 5.12: Execution time to process 10,000 images using the *Static_Architecture* and the *PR_Archtitecture* when the batch size (number of images in-between PR events) grows up to 500.

5.4.3.3 Comparison with other works

A comparison with previous FPGA accelerator designs for CNN and BNN models is provided in Table 5.7. Suda et al.[78] and Qui et al.[70] reported 117 and 136 GOPS/s, respectively, significantly lower than the performance attained through ReFiRe. Li et al.[79] achieved 594 GOPS/s, with 22.5

GOP/s/W efficiency, due to the increased power consumption of the design. Our work outperforms the reference approach proposed by Zhao et al. [5], achieving about 3.1 times higher performance and efficiency for the exact same set of accelerators. Umuroglou et al. [80] and Liang et al. [81] report considerably high performance than all other approaches. Therefore, we intend to employ ReFiRe to further improve the performance of these accelerators by transparently introducing DPR and deploying disaggregated FPGAs.

Table 5.7: Performance comparison with other FPGA-based CNN/BNN accelerators. The presented accelerator system employs the same set of accelerator cores as Zhao et al. [5].

	Zhao et al.[5]	**This work**	Suda et al.[78]	Qiu et al.[70]	Li et al.[79]	Umuroglu et al.[80]	Liang et al.[81]
Platform	Zynq	**ZynqMP**	Stratix-V	Zynq	Virtex-7	Zynq	Stratix-V
	XC7Z020	**XCZU9EG**	5SGSD8	XC7Z045	VX690T	XC7Z045	5SGSD8
Clock(MHz)	143	**150**	120	150	156	200	150
Precision(bit)	Input: 8	**Input: 8**	8-16	16	16	Input: 8	Input: 8
	Weight: 1	**Weight: 1**				Weight: 1	Weight: 1
Model size (OPs)	1.24 G	**1.24 G**	30.9 G	30.76 G	1.45 G	112.5 M	1.23 G
Performance (GOP/s)	207.8	**667**	117	136	565.94	2465.5	9396.41
Power(W)	4.7	**5.97**	25.8	9.63	30.2	11.7	26.2
Efficiency (GOP/s/W)	44.2	**111.73**	4.57	14.22	22.15	210.72	358.64

5.5 ReFiRe performance versus primitive remote calls

As stated in previous sections, ReFiRe [1] encapsulates sequences of operations in Advanced Coprocessor Instructions in order to reduce the computation-to-communication ratio between a host processor and the accelerator device. Table 5.8 below, illustrates the required number of primitive remote calls that had to be accomplished for three different workload types in comparison with the corresponding ACI calls. Furthermore, it exhibits the relation between the workload type and the corresponding ACI class. Through a primitive call, the host processor could handle fundamental remote device's operations e.g start/stop accelerator, initialize input/output data addresses, perform one loop iteration with the appropriate parameters, initialize ICAP controller etc.

Table 5.8: Comparison between the required ACI and primitive calls, for three distinct workload types.

Workload		ACI-based offloading		Primitive calls
Type	Size	ACI class	#ACIs	Number
Accelerator calls	T	TASK	1	T
Loop iterations	L	LOOP	1	L
Partial Reconfiguration Events	W	WINDOW	1	W

Assuming a fixed number of operations that had to be implemented in the remote device,

for every accelerator call, loop iteration and Partial Reconfiguration event, one primitive call is required. On the other hand, regardless the number of the required operations, the appropriate information could be embodied in one ACI, reducing the synchronization requirements between the remote nodes. It is worth noticing that exploiting the Datapath Constructor of the ReFiRe, one TASK class could accommodate more than one accelerator calls depending on the appropriate pipeline depth.

Table 5.9 illustrates the size of the ACI memory that is going to be transferred on the remote side for different workload sizes. In order to deploy T accelerator calls and W Partial reconfiguration events remotely, the overall size of the ACI memory is equal to `Sizeof(WINDOW)+Sizeof(LOOP) + T*Sizeof(TASK)` and `W*Sizeof(WINDOW)+Sizeof(LOOP)+ Sizeof(TASK)`, respectively. Contrarily, each LOOP's iteration information could be embodied in one LOOP class reducing the overall transfer size. Thus, the size of the ACI memory is equal to `Sizeof(WINDOW) + Sizeof(LOOP) + Sizeof(TASK)`.

Table 5.9: ACI memory size for distinct workload sizes.

ACI		Workload	ACI memory
Class type	Class size	Size	Total size (Bytes)
TASK	64(BYTES)	T	Sizeof(WINDOW) + Sizeof(LOOP) + (T)*Sizeof(TASK)
LOOP	32(BYTES)	L	Sizeof(WINDOW) + Sizeof(LOOP) + Sizeof(TASK)
WINDOW	32(BYTES)	W	W*Sizeof(WINDOW) + Sizeof(LOOP) + Sizeof(TASK)

Figure 5.13 below, shows the attained performance improvement for an increasing operation number when deploying ACI instead of performing primitive remote calls. By encapsulating the accelerator calls in TASK classes rather than performing primitive remote calls, the overall performance could be boosted up to 5 times. Likewise, by including the appropriate loop iteration information in a LOOP class will improve the performance up to 3.5 times. However, considering the increased WINDOW class decode time, both methods achieve the same performance. That occurs because it lies in the top of the ACI class hierarchy and includes all the appropriate information related to the contained ACI classes.

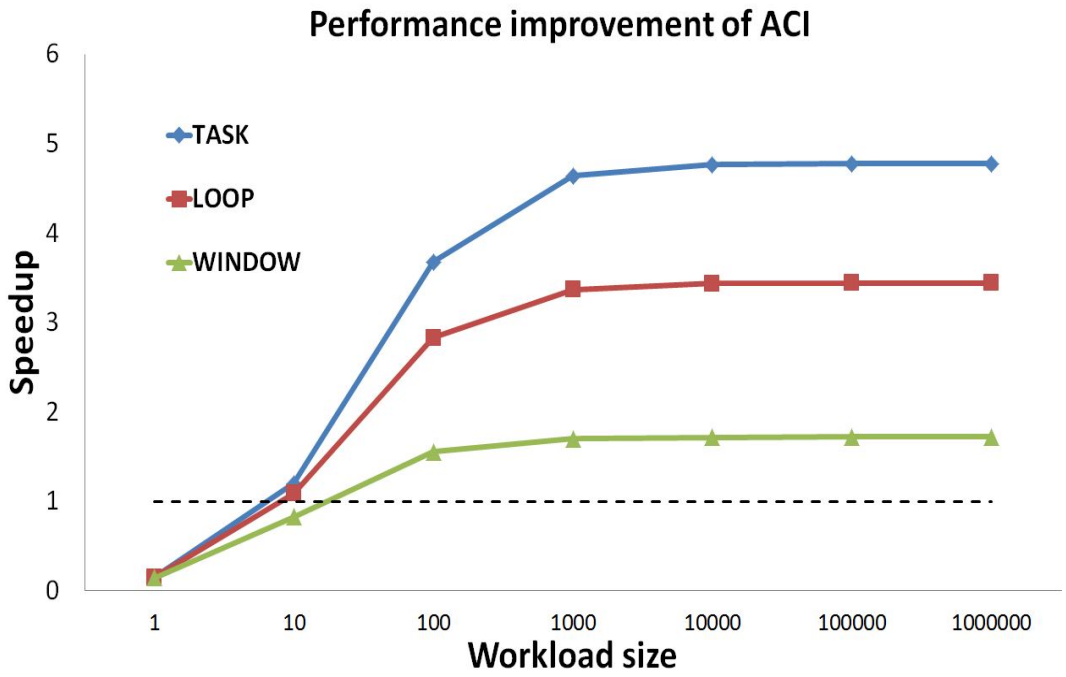

Figure 5.13: Attained performance improvement from adopting ReFiRe for remote accelerator deployment, rather than having primitive remote calls.

Chapter 6

Conclusions and future work

6.1 Conclusions

In this thesis, a generic acceleration framework with support for dynamic partial reconfiguration that improves performance of remote accelerators by encapsulating complex sequences of operations in arbitrarily long instructions called ACIs presented. Through this framework, the control flow of the execution procedure is shifted in the remote side. Thus, it reduces the synchronization requirements between the host processor and the accelerator device. This way, near-peak accelerator performance is exhibited at the application level.

The procedure of mapping a sequential application in Advanced Co-Processor Instructions in order to be processed from the ACI decode control of the ReFiRe framework was a difficult and time consuming process. To alleviate this issue, this thesis presents a source-to-source transformation framework which takes as input the available hardware architecture alongside the corresponding application and yields all the appropriate parts of the framework needed for the remote hardware accelerator deployment. Furthermore, it encapsulates software tasks that are in between hardware accelerators in order to be executed from the ACCEL side reducing the synchronization requirements with the Host device.

In order to evaluate ReFiRe, four distinct applications deployed. For the first application a 2D FFT (implemented by a sequence of 1D FFTs) evaluated. Exploiting LOOP class feature, we managed to reduce the synchronization requirements to 0.5% of the total execution time. For the second application taking advantage of the accelerator pipeline deployment inside a TASK class as well as LOOP's class functionality, near-peak accelerator performance exposed for the LD score calculation. Attributed to source-to-source's capability, for the third application (which calculates LD scores for DNA input data), a software TASK executed in ACCEL (s2s outcome) and in HOST device, representing two different evaluation scenarios. The source-to-source outcome over-performed about 2.5 times the software TASK execution in the HOST side. Finally, a Binarized Convolutional Neural Network evaluated over the ReFiRe framework. Assuming three hardware

BNN accelerator cores, using ReFiRe we encapsulated the required sequence of operations in order to execute the BNN on the remote device. The evaluation results show that disaggregation offers an attractive solution, because the aggregate system performance increases almost linearly with the number of disaggregated FPGAs, due to the beneficial computation-to-synchronization ratio that the ACI offers. That allows to expose near-peak accelerator performance at the application level, despite performing computations on remote node.

To conclude, the ReFiRe framework exploits the five specialization principles, concurrency, computation, communication, caching, and coordination. Based on these principles it constitutes a flexible and efficient framework for remote accelerator deployment in disaggregated environment targeting to maximize the remote accelerator's performance.

6.2 Future work

As aforementioned, the ReFiRe framework minimizes the synchronization requirements between a HOST and ACCEL device, while maximizes accelerator's efficiency. Dealing with more than one ACCEL device a scheduler daemon could be additionally implemented aiming on the efficient ACI dispatch in the remote devices according to a set of scheduling parameters e.g. the reliability of the remote node, the available resources etc. Furthermore, the increased number of the remote devices will increase the communication requirements between the HOST and the ACCEL nodes. Thus, in the host side, a process that will handle communication/synchronization requirements in order to avoid deadlock generation should be implemented. Finally, further custom sequences of operations related to different remote processing resources e.g. GPUs could be also encapsulated in the current infrastructure, maximizing ReFiRe's field of usage.

To summarize, the ReFiRe framework in combination with the above referred daemons could initiate a very reliable, flexible and efficient infrastructure for deploying variable remote accelerator devices in disaggregated data centers handling the major drawbacks that traditional data centers cope with.

References

[1] E. Pissadakis, N. Alachiotis, P. Skrimponis, D. Theodoropoulos, T. Korakis, and D. Pnevmatikatos, "ReFiRe: efficient deployment of Remote Fine-grained Reconfigurable accelerators," *ICFPT*, 2018.

[2] O. Knodel and R. G. Spallek, "Computing Framework for Dynamic Integration of Reconfigurable Resources in a Cloud," *Euromicro Conference on Digital System Design*, 2015.

[3] H. Esmaeilzadeh, E. Blem, R. S. Amant, K. Sankaralingam, and D. Burger, "Dark Silicon and the End of Multicore Scaling," *38 th International Symposium on Computer Architecture (ISCA)*, 2011.

[4] N. Alachiotis *et al.*, "OmegaPlus: a scalable tool for rapid detection of selective sweeps in whole-genome datasets," *Bioinf.*, vol. 28, no. 17, pp. 2274–2275, 2012.

[5] R. Zhao, W. Song, W. Zhang, T. Xing, J.-H. Lin, M. Srivastava, R. Gupta, and Z. Zhang, "Accelerating Binarized Convolutional Neural Networks with Software-Programmable FPGAs," *FPGA*, 2017.

[6] R. Dennard, F. Gaensslen, V. Rideout, E. Bassous, and A. LeBlanc, "Design of ion-implanted MOSFET's with very small physical dimensions," *IEEE Journal of Solid-State Circuits*, 1974.

[7] J. Polzehl and V. Spokoiny, "Propagation-separation approach for local likelihood estimation," *Probab. Theory Related Fields*, vol. 135, no. 3, pp. 335–362, 2006.

[8] J. Polzehl, K. Papafitsoros, and K. Tabelow, "Patch-wise adaptive weights smoothing," WIAS Berlin, Tech. Rep. 2520, 2018.

[9] F. Chen, Y. Shan, Y. Zhang, Y. W. 2, H. Franke, X. Chang, and K. Wang, "Enabling FPGAs in the Cloud," *Proceedings of the 11th ACM Conference on Computing Frontiers*, 2014.

[10] A. Barret, ""A disaggregated server proves breaking up can be a good thing" https://searchdatacenter.techtarget.com/feature/A-disaggregated-server-proves-breaking-up-can-be-a-good-thing," *SearchDataCenter , Online*, 2016.

[11] N. Alachiotis, A. Andronikakis, D. T. Orion Papadakis, D. Pnevmatikatos, D. Syrivelis, A. Reale, K. Katrinis, G. Zervas, H. Y. Vaibhawa Mishra, I. Syrigos, I. Igoumenos, T. Korakis, M. Torrents, and F. Zyulkyarov, "dReDBox: A Disaggregated Architectural Perspective for Data Centers," *Hardware Accelerators in Data Centers*, 2019.

[12] G. E. Moore, "Cramming more components onto integrated circuits," *Electronics, Volume 38, Number 8*, 1965.

[13] "TRS. International technology roadmap for semiconductors, 2010 update, 2011. URL http://www.itrs.net."

[14] S. Borkar, "The exascale challenge." *Keynote at International Symposium on VLSI Design, Automation and Test (VLSI-DAT)*, 2010.

[15] M. Ferdman, A. Adileh, O. Kocberber, S. Volos, M. Alisafaee, D. Jevdjic, C. Kaynak, A. D. Popescu, A. Ailamaki, and B. Falsafi, "A CASE FOR SPECIALIZED PROCESSORS FOR SCALE-OUT WORKLOADS," *IEEE Micro*, 2014.

[16] ——, "Clearing the Clouds : A Study of Emerging Scale-out Workloads on Modern Hardware ," *In Proceedings of the 17th Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2012.

[17] J. Weerasinghe, F. Abel, C. Hagleitner, and A. Herkersdorf, "Enabling FPGAs in Hyperscale Data Centers," *15 IEEE 12th Intl Conf on Ubiquitous Intelligence and Computing*, August 2015.

[18] A. Putnam, A. M. Caulfield, E. S. Chung, D. Chiou, K. Constantinides, J. Demme, H. Esmaeilzadeh, J. Fowers, G. P. Gopal, J. Gray, M. Haselman, S. Hauck, S. Heil, A. Hormati, J.-Y. Kim, S. Lanka, J. Larus, E. Peterson, S. Pope, A. Smith, J. Thong, P. Y. Xiao, and D. Burger, "A Reconfigurable Fabric for Accelerating Large-Scale Datacenter Services," *Proceeding of the 41st Annual International Symposium on Computer Architecuture, ser. ISCA '14. Piscataway, NJ, USA: IEEE Press, 2014, pp. 13–24.*

[19] J. Weerasinghe, R. Polig, F. Abel, and A. Hagleitner, "Network-Attached FPGAs for Data Center Applications," *International Conference on Field-Programmable Technology (FPT)*, 2016.

[20] N. Tarafdar, T. Lin, D. Ly-Ma, D. Rozhko, A. Leon-Garcia, and P. Chow, "Building the Infrastructure for Deploying FPGAs in the Cloud," *Hardware Accelerators in Data Centers*, August 2019.

[21] G. W, L. E, D. N, and S. A, "A high-performance, portable implementation of the MPI message passing interface standard," *Parallel Comput 22(6):789–828. https://doi. org/10.1016/0167-8191(96)00024-5*, 1996.

[22] N. Tarafdar, T. Lin, E. N, L. D, L.-G. A, and P. Chow, "Heterogeneous virtualized network function framework for the data center." *Field programmable logic and applications (FPL)*, 2017.

[23] "OpenStack (2018) OpenStack. https://www.openstack.org/."

[24] N. Neves, P. Tomás, and N. Roma, "Host to Accelerator Interfacing Framework for High-Throughput Co-Processing Systems," 2015.

[25] M. Vesper, D. Koch, K. Vipin, and S. A. Fahmy, "JetStream: An Open-Source High-Performance PCI Express 3 Streaming Library for FPGA-to-Host and FPGA-to-FPGA Communication," *26th International Conference on Field Programmable Logic and Applications (FPL)*, 2016.

[26] M. Jacobsen, Y. Freund, and R. Kastner, "RIFFA: A Reusable Integration Framework for FPGA Accelerators," *IEEE 20th International Symposium on Field-Programmable Custom Computing Machines*, 2012.

[27] S. Paiágua, F. Pratas, P. Tomás, N. Roma, and R. Chaves, "HotStream: Efficient Data Streaming of Complex Patterns to Multiple Accelerating Kernels," *25th International Symposium on Computer Architecture and High Performance Computing*, 2013.

[28] J. Gong, T. Wang, J. Chen, H. Wu, F. Ye, S. Lu, and J. Cong, "An Efficient and Flexible Host-FPGA PCIe Communication Library," *International Conference on Field Programmable Logic and Applications (FPL)*, 2014.

[29] G. Marcus, W. Gao, A. Kugel, and R. Männer, "THE MPRACE FRAMEWORK: AN OPEN SOURCE STACK FOR COMMUNICATION WITH CUSTOM FPGA-BASED ACCELERATORS," *VII Southern Conference on Programmable Logic (SPL)*, 2011.

[30] S. A. Fahmy, K. Vipin, and S. Shreejith, "Virtualized FPGA Accelerators for Efficient Cloud Computing," *International Conference on Cloud Computing Technology and Science (CloudCom)*, 2015.

[31] M. Jacobsen *et al.*, "RIFFA 2.1: A reusable integration framework for FPGA accelerators," *ACM TRETS*, vol. 8, no. 4, p. 22, 2015.

[32] Amazon Web Services, "Amazon EC2 F1 Machine Instance," https://aws.amazon.com/ec2/instance-types/f1/, [Online; accessed 02-Jun-2019].

[33] Alibaba Cloud, "Machine Learning Platform For AI," https://www.alibabacloud.com, [Online; accessed 02-Jun-2019].

[34] C. Kachris, D. Soudris, G. Gaydadjiev, H.-N. Nguyen, D. S. Nikolopoulos, A. Bilas, N. Morgan, C. Strydis, C. Tsalidis, J. Balafas, R. Jimenez-Peris, and A. Almeida, "The VINEYARD approach: Versatile, Integrated, Accelerator-based, Heterogeneous Data Centres," *5th International Conference on Modern Circuits and Systems Technologies (MOCAST)*, 2016.

[35] T. Nowatzki *et al.*, "Domain specialization is generally unnecessary for accelerators," *IEEE Micro*, vol. 37, no. 3, pp. 40–50, 2017.

[36] ——, "Pushing the limits of accelerator efficiency while retaining programmability," in *HPCA 2016*. IEEE, 2016, pp. 27–39.

[37] Xilinx, "UltraScale Architecture Configuration," https://www.xilinx.com/support/documentation/user_guides/ug570-ultrascale-configuration.pdf, [Online; accessed 05-Jul-2018].

[38] D. Theodoropoulos, A. Reale, D. Syrivelis, M. Bielski, N. Alachiotis, and D. Pnevmatikatos, "REMAP: Remote mEmory Manager for disAggregated Platforms," *IEEE 29th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, 2018.

[39] Xilinx, "AXI Hardware ICAP," https://www.xilinx.com/products/intellectual-property/axi_hwicap.html, [Online; accessed 05-Jul-2018].

[40] J. Maynard Smith and J. Haigh, "The hitch-hiking effect of a favourable gene." *Genetical research*, vol. 23, no. 1, pp. 23–35, Feb. 1974. [Online]. Available: http://www.ncbi.nlm.nih.gov/pubmed/4407212

[41] M. T. Alam *et al.*, "Selective sweeps and genetic lineages of plasmodium falciparum drug-resistant alleles in ghana," *J. of Infectious Diseases*, vol. 203, no. 2, pp. 220–227, 2011.

[42] N. G. de Groot and R. E. Bontrop, "The HIV-1 pandemic: does the selective sweep in chimpanzees mirror humankind's future?" *Retrovirology*, vol. 10, no. 1, p. 53, Jan. 2013. [Online]. Available: http://www.retrovirology.com/content/10/1/53

[43] J. L. Crisci, Y.-P. Poh, S. Mahajan, and J. D. Jensen, "The impact of equilibrium assumptions on tests of selection," *Frontiers in genetics*, vol. 4, 2013.

[44] R. Nielsen *et al.*, "Genomic scans for selective sweeps using SNP data," *Genome Research*, vol. 15, no. 11, pp. 1566–1575, Nov. 2005.

[45] P. Pavlidis *et al.*, "SweeD: likelihood-based detection of selective sweeps in thousands of genomes," *Molecular biology and evolution*, p. mst112, 2013.

[46] Y. Kim and R. Nielsen, "Linkage disequilibrium as a signature of selective sweeps," *Genetics*, vol. 167, no. 3, pp. 1513–1524, Jul. 2004. [Online]. Available: http://www.ncbi.nlm.nih.gov/pubmed/15280259

[47] R. R. Hudson, "Generating samples under a Wright-Fisher neutral model of genetic variation." *Bioinformatics (Oxford, England)*, vol. 18, no. 2, pp. 337–8, 2002. [Online]. Available: http://www.ncbi.nlm.nih.gov/pubmed/11847089

[48]

[49] M. Kimura, "The number of heterozygous nucleotide sites maintained in a finite population due to steady flux of mutations," *Genetics, vol.61, no. 2, pp. 57-86*, 1969.

[50] S. Tavaré, "Some probabilistic and statistical problems in the analysis of dna sequences," *Lectures on mathematics in the life sciences, vol. 17, no. 2, pp. 57–86*, 1986.

[51] Z. DV, P. A, and W. BS, "orrelation-based inference for linkage disequilibrium with multiple alleles."

[52] A. Krizhevsky *et al.*, "ImageNet Classification with Deep Convolutional Neural Networks," in *Advances in Neural Information Processing Systems (NIPS)*, 2012, pp. 1097–1105.

[53] O. Russakovsky *et al.*, "ImageNet Large Scale Visual Recognition Challenge," *International Journal of Computer Vision*, vol. 115, no. 3, pp. 211–252, 2015.

[54] R. B. Girshick *et al.*, "Rich Feature Hierarchies for Accurate Object Detection and Semantic Segmentation," in *2014 IEEE Conference on Computer Vision and Pattern Recognition, CVPR*, 2014, pp. 580–587.

[55] Y. LeCun, Y. Bengio, and G. Hinton, "Deep Learning," *Nature*, vol. 521, pp. 436–44, 05 2015.

[56] G. Hinton *et al.*, "Deep Neural Networks for Acoustic Modeling in Speech Recognition," *Signal Processing Magazine*, 2012.

[57] L. Deng *et al.*, "Recent advances in deep learning for speech research at Microsoft," in *IEEE International Conference on Acoustics, Speech and Signal Processing, ICASSP 2013, Vancouver, BC, Canada, May 26-31, 2013*, 2013, pp. 8604–8608.

[58] R. C. et al, "Natural Language Processing (Almost) from Scratch," *Journal of Machine Learning Research*, vol. 12, pp. 2493–2537, 2011.

[59] C. Chen, A. Seff, A. Kornhauser, and J. Xiao, "DeepDriving: Learning Affordance for Direct Perception in Autonomous Driving," in *Proceedings of the 2015 IEEE International Conference on Computer Vision (ICCV)*, 2015.

[60] T. Zhang, G. Kahn, S. Levine, and P. Abbeel, "Learning Deep Control Policies for Autonomous Aerial Vehicles with MPC-Guided Policy Search," *CoRR*, vol. abs/1509.06791, 2015.

[61] A. Esteva *et al.*, "Dermatologist-level classification of skin cancer with deep neural networks," *Nature*, vol. 542, no. 7639, pp. 115–118, 2017.

[62] M. Jermyn *et al.*, "Neural networks improve brain cancer detection with Raman spectroscopy in the presence of light artifacts," in *Clinical and Translational Neurophotonics; Neural Imaging and Sensing; and Optogenetics and Optical Manipulation*, vol. 9690, 2016.

[63] D. Wang *et al.*, "Deep Learning for Identifying Metastatic Breast Cancer," *CoRR*, vol. abs/1606.05718, 2016.

[64] D. Silver *et al.*, "Mastering the game of Go with deep neural networks and tree search," *Nature*, vol. 529, no. 7587, pp. 484–489, 2016.

[65] E. Nurvitadhi, D. Sheffield, , A. Mishra, G. Venkatesh, and D. Marr, "Accelerating Binarized Neural Networks: Comparison of FPGA, CPU, GPU, and ASIC," in *ICFPT*, 2016, pp. 77–84.

[66] C. Szegedy *et al.*, "Going Deeper with Convolutions," *CoRR*, vol. abs/1409.4842, 2014.

[67] K. He, X. Zhang, S. Ren, and J. Sun, "Deep Residual Learning for Image Recognition," in *2016 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2016, Las Vegas, NV, USA, June 27-30, 2016*, 2016, pp. 770–778.

[68] K. Simonyan *et al.*, "Very Deep Convolutional Networks for Large-Scale Image Recognition," *CoRR*, vol. abs/1409.1556, 2014.

[69] X. Xie *et al.*, "Exploiting Sparsity to Accelerate Fully Connected Layers of CNN-Based Applications on Mobile SoCs," *ACM Trans. Embedded Comput. Syst.*, vol. 17, no. 2, pp. 37:1–37:25, 2018.

[70] J. Qiu *et al.*, "Going Deeper with Embedded FPGA Platform for Convolutional Neural Network," *FPGA*, 2016.

[71] S. Han *et al.*, "Deep Compression: Compressing Deep Neural Network with Pruning, Trained Quantization and Huffman Coding," *CoRR*, vol. abs/1510.00149, 2015.

[72] F. N. Iandola *et al.*, "SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and <1MB model size," *CoRR*, vol. abs/1602.07360, 2016.

[73] M. Courbariaux, Y. Bengio, and J. David, "BinaryConnect: Training Deep Neural Networks with binary weights during propagations," *CoRR*, vol. abs/1511.00363, 2015.

[74] Y. Netzer, T. Wang, A. Coates, A. Bissacco, B. Wu, and A. Y. Ng, "Reading Digits in Natural Images with Unsupervised Feature Learning," in *NIPS Workshop on Deep Learning and Unsupervised Feature Learning*, 2011.

[75] M. Rastegari, V. Ordonez, J. Redmon, and A. Farhadi, "XNOR-Net: ImageNet Classification Using Binary Convolutional Neural Networks," *CoRR*, vol. abs/1603.05279, 2016.

[76] S. Ioffe and C. Szegedy, "Batch normalization: Accelerating deep network training by reducing internal covariate shift," *CoRR*, vol. abs/1502.03167, 2015.

[77] A. Krizhevsky, "Learning Multiple Layers of Features from Tiny Images," 2009.

[78] N. Suda *et al.*, "Throughput-Optimized OpenCL-based FPGA Accelerator for Large-Scale Convolutional Neural Networks," *FPGA*, pp. 16–25, 2016.

[79] H. Li *et al.*, "A High Performance FPGA-based Accelerator for Large-Scale Convolutional Neural Networks," *International Conference on Field Programmable Logic and Applications (FPL)*, pp. 1–9, 2016.

[80] Y. Umuroglu *et al.*, "FINN: A Framework for Fast, Scalable Binarized Neural Network Inference," *FPGA*, 2017.

[81] S. Liang *et al.*, "FP-BNN: Binarized neural network on FPGA," *Neurocomputing*, 2017.