A WORKFLOW FOR OFFERING HARDWARE ACCELERATORS AS A CLOUD COMPUTING SERVICE: IMPLEMENTATION AND EVALUATION

A THESIS SUBMITTED TO
THE GRADUATE SCHOOL OF NATURAL AND APPLIED SCIENCES
OF
MIDDLE EAST TECHNICAL UNIVERSITY

BY

ANIL TIRLIOĞLU

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR
THE DEGREE OF MASTER OF SCIENCE
IN
ELECTRICAL AND ELECTRONICS ENGINEERING

FEBRUARY 2022

Approval of the thesis:

A WORKFLOW FOR OFFERING HARDWARE ACCELERATORS AS A CLOUD COMPUTING SERVICE: IMPLEMENTATION AND EVALUATION

submitted by **ANIL TIRLIOĞLU** in partial fulfillment of the requirements for the degree of **Master of Science in Electrical and Electronics Engineering Department, Middle East Technical University** by,

Date: 10.02.2022

ABSTRACT

A WORKFLOW FOR OFFERING HARDWARE ACCELERATORS AS A CLOUD COMPUTING SERVICE: IMPLEMENTATION AND EVALUATION

Tırlıoğlu, Anıl
M.S., Department of Electrical and Electronics Engineering
Supervisor: Prof. Dr. Şenan Ece Güran Schmidt

February 2022, 77 pages

Cloud computing and hardware accelerators are two paradigm changes in the field of information technologies and computers. Accordingly, this thesis proposes a workflow for offering users hardware accelerators implemented on FPGA as computing resources in a heterogeneous cloud data center.

To this end, we perform the virtualization of FPGA resources as reconfigurable regions (RRs) and provide these resources through OpenStack, an open-source cloud resource management platform. Our workflow is designed for SoC FPGA platforms with a processor. The OpenStack module in the SoC processor is implemented as embedded software that works with other OpenStack modules. An accelerator image selected by the user can be programmed to an RR through OpenStack. The FPGA platform in our architecture features 40 Gbps Ethernet IP Cores and an on-chip switch that enables the communication of the RRs with each other, the SoC processor and the 40 Gbps Ethernet. To this end, distributed accelerator implementations can be realized on the same FPGA, and data received from the users can be processed and delivered back to the user. We explore OpenCL-based accelerator realization, which defines the data exchange between the accelerators and the CPU software. Furthermore, we demonstrate the implementation of the accelerators as stand-alone IP cores together with their wrappers in a more custom design flow. We implement the entire workflow on the Xilinx ZC706 board. The functional correctness and performance experiments are conducted throughout the thesis work. The experiments cover the effects of the Ethernet interface on the performance in accordance with the cloud data center operation.

Keywords: Hardware accelerator, hardware accelerated cloud data center, partial reconfiguration, FPGA virtualization

ÖZ

DONANIM HIZLANDIRICILARININ BULUT BİLİŞİM SERVİSİ OLARAK SUNULMASI İÇİN BİR İŞ AKIŞI: GERÇEKLEŞTİRİM VE DEĞERLENDİRME

Tırlıoğlu, Anıl
Yüksek Lisans, Elektrik ve Elektronik Mühendisliği Bölümü
Tez Yöneticisi: Prof. Dr. Şenan Ece Güran Schmidt

Şubat 2022 , 77 sayfa

Bulut bilişim ve donanım hızlandırıcıları, bilgi teknolojileri ve bilgisayar alanındaki iki paradigma değişikliğidir. Bu kapsamda, bu tez, kullanıcılara heterojen bir bulut veri merkezinde bilgi işlem kaynakları olarak FPGA üzerinde gerçeklenen donanım hızlandırıcılarını sunmak için bir iş akışı önermektedir.

Bu amaçla FPGA kaynaklarının yeniden yapılandırılabilir bölgeler (RR'ler) olarak sanallaştırılmasını gerçekleştiriyor ve bu kaynakları açık kaynaklı bir bulut kaynak yönetim platformu olan OpenStack üzerinden sağlıyoruz. İş akışımız, işlemcili SoC FPGA platformları için tasarlanmıştır. SoC işlemcisindeki OpenStack modülü, diğer OpenStack modülleriyle çalışan bir gömülü yazılımdır. Kullanıcı tarafından seçilen bir hızlandırıcı imajı OpenStack üzerinden RR üzerine yazılabilir. Mimarimizdeki FPGA platformu, 40 Gbps Ethernet IP Çekirdekleri ve RR'lerin birbirleriyle, SoC işlemcisi ve 40 Gbps Ethernet ile iletişimini sağlayan bir çip üzerinde anahtar içerir. Bu amaçla aynı FPGA üzerinde dağıtık hızlandırıcı uygulamaları gerçekleştirilebilir ve kullanıcılardan alınan veriler işlenerek kullanıcıya geri iletilebilir. Hızlandırıcılar ve CPU yazılımı arasındaki veri alışverişini tanımlayan OpenCL tabanlı hızlandırıcı gerçekleştirmeyi araştırıyoruz. Ayrıca, hızlandırıcıların daha özel bir tasarım akışında kabuk tasarımlarıyla birlikte bağımsız IP çekirdekleri olarak uygulanmasını gösteriyoruz. Tüm iş akışını Xilinx ZC706 kartında uyguluyoruz. Tez çalışması boyunca işlevsel doğruluk ve performans deneyleri yapılmıştır. Deneyler, bulut veri merkezi çalışmasına uygun olarak Ethernet arayüzünün başarıma etkilerini de kapsamaktadır.

Anahtar Kelimeler: Donanım Hızlandırıcılar, donanım hızlandırıcılı bulut veri merkezi, kısmi yeniden yapılandırma, FPGA sanallaştırılması

To my family

ACKNOWLEDGMENTS

First of all, I would like to thank my supervisor Prof. Dr. Şenan Ece Güran Schmidt for her exceptional efforts. Her inspiring and friendly supervision has helped me tremendously in cultivating and maintaining enthusiasm in our research. I am very grateful to her for giving me an academic mindset.

I would also like to thank my colleague and mentor Alper Yazar, who contributed greatly to the development of the work presented in this thesis, for his constant support, constructive suggestions and patience.

A special thank to my colleagues Yunus Esergün and Fatih Yazıcı, whom we worked with on the ACCLOUD project, I am grateful for their contributions and cooperation in the work.

This thesis was supported by the Scientific and Research Council of Turkey (TUBITAK) [Project Code 117E667-117E668].

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

FIGURES

LIST OF ABBREVIATIONS

ABBREVIATIONS	
2-D	Two Dimensional
ACCLOUD	Accelerated Cloud
ALM	Adaptive Logic Module
AMQP	Advanced Message Queuing Protocol
API	Application Programming Interface
ASIC	Application Specific Integrated Circuit
AXI	Advanced Extensible Interface
AWS	Amazon Web Service
BRAM	Block RAM
CK	Canny Kernel
CPU	Central Processing Unit
DCP	Design Checkpoint
DDR	Double Data Rate
DFX	Dynamic Function Exchange
DMA	DirectMemory Access
DSA	Domain Specific Architecture
DSP	Digital Signal Processor
DTS	Device Tree Source
ETK	Edge Tracing Kernel
FAC	FPGA Accelerator Card
FF	Flip Flop
FFT	Fast Fourier Transform

FIFO	First In First Out
FPGA	Field Programmable Gate Array
FPS	Frame Per Second
GbE	Gigabit Ethernet
GMII	Gigabit Media Independent Interface
GPU	Graphics Processing Unit
HA	Hardware Accelerator
HACDC	Hardware Accelerated Cloud Data Center
HaaS	Hardware as a Service
HAK	Hardware Accelerator Kernel
HBM	High Bandwidth Memory
HDL	Hardware Description Language
HLS	High Level Synthesis
IaaS	Infrastructure as a Service
IDE	Integrated Development Environment
II	Input Image
IP	Internet Protocol / Intellectual Property
KNN	K-Nearest Neighbours
LUT	Look Up Table
MAC	Media Access Control
MC	Main Code
MII	Media Independent Interface
MPSOC	Multiprocessor System-on-Chip
MTCNN	Multi-task Cascaded Convolutional Networks
MQ	Message Queue
OI	Output Image
OpenCL	Open Computing Language

PC	Personal Computer
PCAP	Processor Configuration Access Port
PCIE	Peripheral Component Interconnect Express
PR	Partial Reconfiguration
QOS	Quality of Service
QSFP+	Quad Small Form-Factor Pluggable Plus
RAM	Random Access Memory
RDMA	Remote Direct Memory Access
RM	Reconfigurable Module
PL	Programmable Logic
RPC	Remote Process Communication
RR	Reconfigurable Region
PS	Processing System
QEMU	Quick Emulator
RTL	Register Transfer Level
SOC	System-on-Chip
SSR	Super Sample DataRate
SW	Software
TCL	Tool Command Language
vFPGA	Virtual FPGA
VHSIC	Very High Speed Integrated Circuit Program
VHDL	VHSIC Hardware Description Language
VM	Virtual Machine
XBAR	Crossbar
XRT	Xilinx Runtime

CHAPTER 1

INTRODUCTION

The traditional approach to computer hardware, software, application and computation functions is to be managed by the users and in the environment where the users are located. In recent years, this traditional approach is increasingly replaced by cloud computing, and it is preferred to offer these functions as a service over the Internet in accordance with user demands. For this purpose, cloud service providers establish cloud data centers (CDC) with thousands of servers and a a high-speed networking infrastructure.

The most critical technology that makes cloud computing possible is *virtualization*, which enables using of hardware resources such as the number of processor cores, memory and disk as independent and abstracted partitions. Virtualization is accomplished by adding an abstraction software called the *hypervisor* between the physical machine hardware and the operating system.

With the application of this technology on servers, virtual machines are created and offered to users as infrastructure as a service (IaaS). To this end, OpenStack [6] is an open source platform used in cloud-based data centers and private cloud systems for offering IaaS services. OpenStack maintains a database of available computing resources together with *virtual machine* (VM) configurations. It instantiates the IaaS requests by fetching VM images from a repository and instantiating them on physical servers on demand.

Recently, cloud computing services include heterogeneous hardware resources including GPUs or FPGAs as well as servers [7, 8, 9]. This trend is in line with the prediction pointed out by [10] that hardware accelerators specially designed for a given computation will outperform general-purpose processors in terms of power and performance. FPGA platforms are preferred for accelerator implementation as they offer software tools that facilitate the implementation of different applications on the hardware. In this new *heterogeneous cloud computing architecture*, FPGA resources must be *virtualized* and presented to the user seamlessly with server resources. Within the scope of IaaS, the user should be able to realize the accelerators that he or she designed or bought as a service on the FPGA. SaaS requests are fulfilled under the control of the cloud service manager and independent of the user. In this case, hardware accelerators can also be used as less power-consuming alternatives for SaaS requests.

FPGA accelerators can be set up as PCIe attached accelerator cards to a traditional x86 processor or standalone network-attached devices tightly coupled with an SoC processor. A single FPGA can host multiple hardware accelerators within its dynamically programmable reconfigurable regions (RR) [11].

Constructing VMs with conventional cloud server resources such as CPU, memory and disk is a well-established procedure that is supported by well-known Operating Systems and smooth integration of the hypervisors. However, virtualization of FPGA resources and offering them to the users transparently with the server resources require hardware-specific implementations and extension of the cloud service management infrastructure. FPGA virtualization should provide partitions of FPGA resources that can be independently used. Furthermore, a seamless workflow is needed to implement hardware accelerators on the virtualized FPGA resources. Such workflow should produce accelerator bitstreams that can be instantiated on the virtualized FPGA resources similar to VM images for conventional cloud server resources.

This thesis makes the following contributions to address these problems that are presented as a workflow for offering hardware accelerators located in FPGA accelerator cards as computing resources as a part of IaaS in cloud computing platforms. The FPGA platform in this thesis is an SoC that features a CPU.

First, a specific implementation of the OpenStack software component, which we call Nova-Zynq, to run on the FPGA SoC ARM processor is carried out. This implementation is compatible with the rest of the OpenStack software by sending and receiving messages in the format defined by the OpenStack APIs. Nova-Zynq enables users to include the FPGA resources in their VMs and instantiate their accelerators by downloading and programming bitstream from a repository in the OpenStack platform. The virtualization of the resources in this thesis is done by partial reconfiguration. To this end, Nova-Zynq works together with other software components that can program the hardware accelerator bitstream into the reconfigurable regions (RRs). We provide the design of the static logic and the RMs. We demonstrate the functional correctness of Nova-Zynq. Second, we explore two different accelerator implementation flows. The first flow features the OpenCL programming model, which incorporates the software code that runs on the CPU, offloading the process to the hardware and collecting the output. The second flow implements the accelerator as an IP core with a custom-designed wrapper to provide the interface to the data exchange with the CPU or remote application. The first flow is meaningful to speed up the main software application function, where the data to be processed is located close to the CPU and hardware accelerator, with a hardware tightly coupled to the CPU. On the other hand, the second flow; In cases where a main application on the user side is accelerated or data is streamed by the user, it is more suitable for accelerating it without any extra CPU intervention in hardware accelerators located in the Cloud environment. Both of these flows are demonstrated with different accelerator applications. These applications are selected to demonstrate different scenarios. The first scenario covers a distributed accelerator implementation with two sub-components realized on two different RRs. The components communicate by an on-chip switch that resides in the static logic. The second scenario demonstrates the use of the hardware accelerator as a bump-in-the-wire network packet processor. All the implementations except for the static logic are carried out within the scope of this thesis. We demonstrate the functional correctness of the accelerators together with their performance, including resource use, throughput and latency with a series of experiments. In accordance with the thesis focus of offering the hardware accelerators as a cloud service, all experiments involve sending and receiving data for the accelerators through a 40 Gbps Ethernet Interface. We investigate the effects of various Ethernet frame sizes. All the tests include software written in the scope of this thesis that packs or extracts data to/from Ethernet frames. We emphasize that designing very efficient or high-performance accelerators is not in the scope of this thesis.

We use Xilinx tools and the ZC706 Evaluation Kit [12] SoC evaluation board during the thesis work. An FPGA tool ecosystem of Xilinx consists of Vivado, Vitis, Vitis HLS, Petalinux but not limited to these. Vivado is a tool that synthesizes, simulates, implements, and generates hardware binaries for programmable logic resources of FPGA. HDL codes like Verilog and VHDL and built-in functional IP blocks are the way to describe the hardware in the Vivado tool. Vitis is a software development platform. Vitis compiler compiles the applications written in C/C++ targeting the processing system of SoC. Vitis can also produce complete FPGA and ARM CPU binaries when used in conjunction with Vivado artifacts. Vitis HLS compiler compiles HLS codes, a subset of C/C++ language, into HDL codes or the Vivado RTL/block design IP. Petalinux creates custom GNU Linux distributions for Xilinx SoC hardware embedded systems. It can automatically generate bootloaders, kernel, device tree, and root file system for custom hardware generated by Vivado.

The SoC concept is used by many electronic component manufacturers for different hardware chips. Basically, the SoC can be formed by connecting the main CPU and other components such as GPU, FPGA, Signal Processors, analog-digital modem, interface units, memory units with a common bus within a chip. The Zynq-7000 SoC on the ZC706 Evaluation Kit used in this thesis consists of a processor and FPGA structure called PS and PL, respectively. In this system, the application processor cores are connected together with various peripherals as well as the PL domain with the industry-standard AXI bus structure. ARM A series processors are GNU Linux capable and can access peripherals, memories and custom hardware in the PL area from the same address space via AXI bus.

SoCs with FPGAs generally have a similar structure to the Zynq-7000. Zynq-Ultrascale+ [13] SoCs have a more advanced structure than Zynq-7000. Zynq-Ultrascale+ features A and R family ARM processors, GPU, advanced peripherals, transceivers and increased logic resources. AXI bus structure is similar to the Zynq-7000 series. Intel has a similar architecture. For example, this can be seen in Agilex SoCs [14]. This SoC structure also has ARM processors, AXI bus

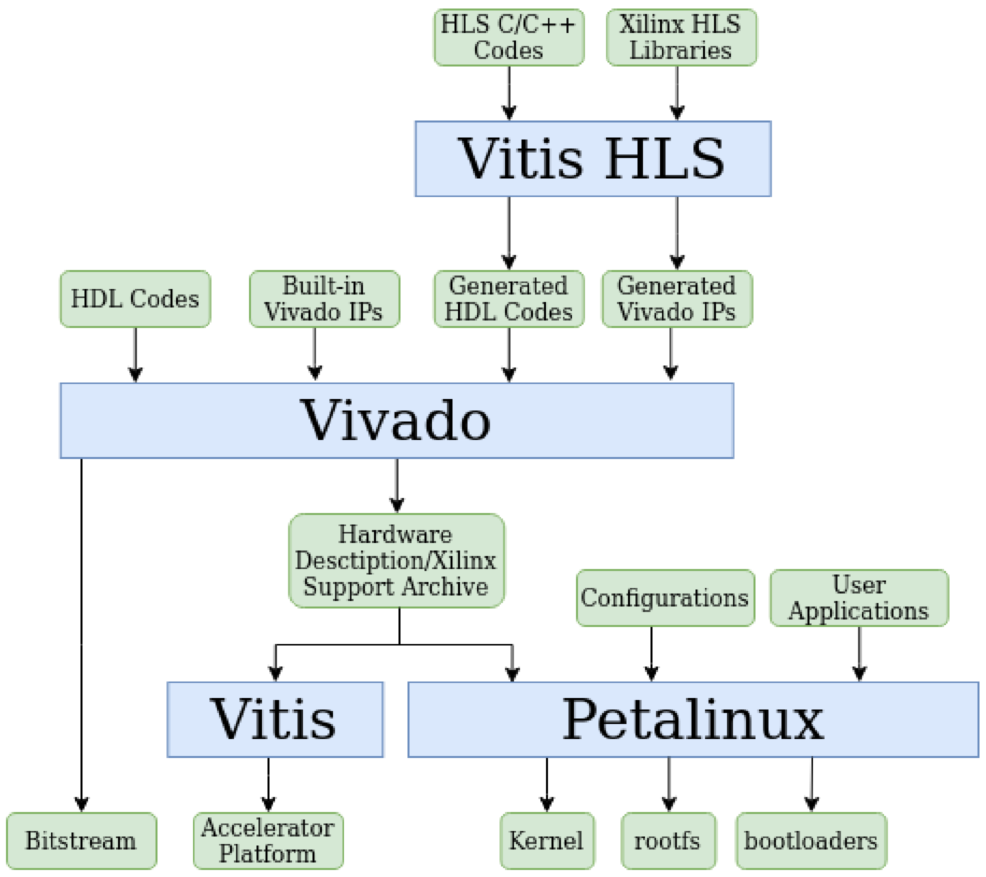


Figure 1.1: Overview of Xilinx Tools Flow in Thesis Work

structure and FPGA area. The Microchip has a Polarfire [15] product family with RISC-V processors and FPGA field structure connected by AXI bus. RISC-V soft processor IPs [16] that can be implemented in FPGA areas are offered by vendors on hardware that does not have a hard processor and only has FPGA domains. In this way, it is possible to create similar SoC structures with different FPGA vendor hardware.

The remainder of this thesis is organized as follows. Chapter 2 provides the background on FPGA virtualization and the specifications of selected fundamental hardware accelerators. We then introduce the workflows and tools for FPGA hardware accelerator realization in accordance to Figure 1.1. The implementations and tests are carried out on an FPGA platform that is designed in the scope of a Heterogeneous Cloud Data Center Architecture ACCLOUD. To this end, we introduce the relevant features and components of the ACCLOUD architecture.

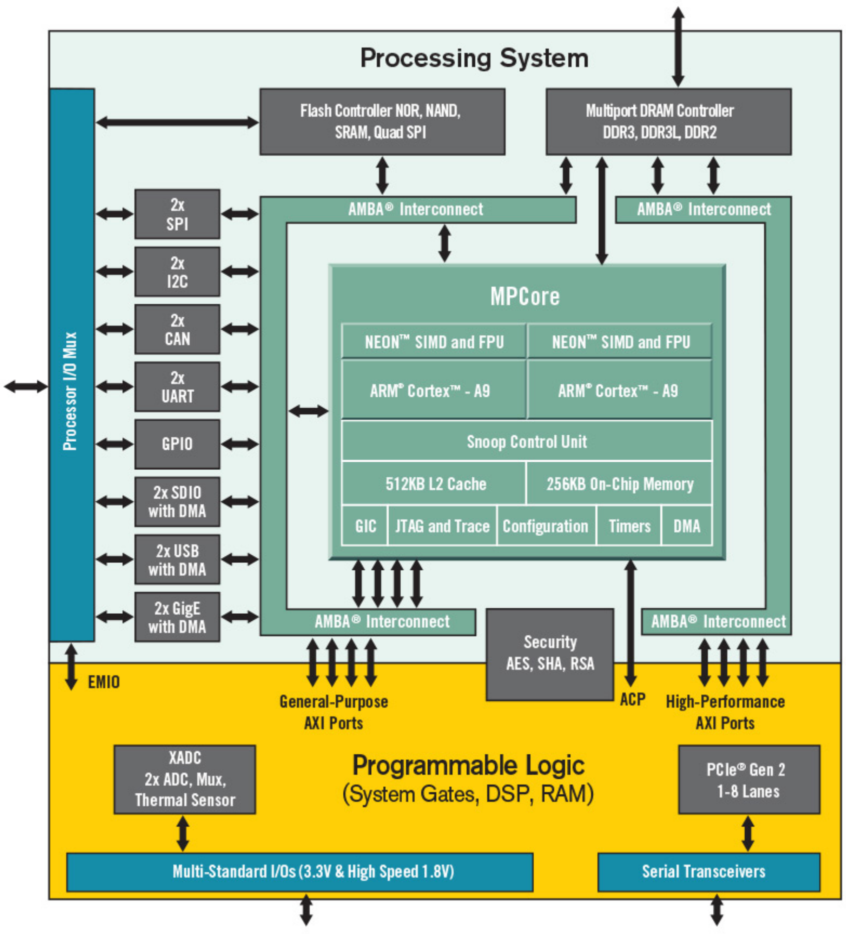


Figure 1.2: ZYNQ-7000 SoC Structure [2]

Chapter 3 presents our proposed workflow for offering hardware accelerators as cloud resources. To this end, this chapter presents Nova-Zynq and other software tools that present the RRs on FPGA as standalone programmable hardware resources to OpenStack. We describe the OpenCL based accelerator development with the Canny Edge Detector application that is implemented in two kernels in two RRs. We then provide a Gaussian Filter and a Sobel Filter implemented as IP cores together with their wrappers on two RRs. These two filters construct an image processing pipeline. We provide the implementation of the Ethernet Packet Processor Application. Chapter 4 describes the experiments that we conduct to demonstrate the functional correctness of our workflow. We further conduct experiments to measure the data processing performance of the overall architecture, complete with the frames received and sent through 40 Gbps Ethernet Interface as a part of cloud service. Chapter 5 summarizes our findings and future works.

CHAPTER 2

HARDWARE ACCELERATORS, HARDWARE VIRTUALIZATION, AND HARDWARE ACCELERATORS AS A CLOUD SERVICE

Moore's Law [17] and Dennard's Scaling [18] have been the defining rules of hardware architectures for computing are for many years. There is a roughly 15-fold gap between Moore's prediction and the current capability as of 2018. The prediction of constant power per mm^2 of silicon stated in Dennard scaling does not hold anymore since 2012. Domain-specific architectures (DSA) are "tailored to a specific problem domain and offer significant performance (and efficiency) gains for that domain". [10] propose DSA as a remedy for the gradual loss of relevance of these rules and to continue improving performance and energy efficiency. Performance improvement can be achieved by accelerating a component of an application by a DSA with respect to executing the entire application on a general-purpose CPU. Accordingly, DSA's including graphics processing units (GPUs) or IP Cores on FPGA's are called *accelerators*.

The focus of this thesis is Hardware Accelerators that are realized on FPGA because of their capability to realize different types of applications. Applications that use operands with custom data widths, combinational logic problems, finite state machines, and parallel MapReduce problems are particularly fit for FPGA implementation [19].

2.1 FPGA Virtualization and Hardware Accelerator implementations On FPGA

FPGAs are competitive on energy efficiency compared to CPUs and GPUs for machine learning and video processing problems. These problems feature integer arithmetic, Hamming Distance, KNN voting, dot product, scalar multiplication, vector addition, and binarized 2D convolution compute kernels [20]. Furthermore, FPGAs can efficiently realize sort and search computations by parallel networks [21].

[22] categorizes FPGA virtualization at the resource level, node level, and multi-node level. Resource level virtualization includes overlays. "An FPGA overlay is a virtual reconfigurable architecture that overlays on top of the physical FPGA configurable fabric [23]." To this end, an FPGA overlay is between the user application and the underlying physical FPGA. Rather than implementing the application on the physical FPGA directly, the application is targeted toward the overlay architecture independent of the physical FPGA features. The compilation has a second step to translate this overlay architecture, together with the application that runs on it, to the physical FPGA. Node level virtualization support at the node level is hardware and software infrastructure to manage the resources related to a single FPGA. The *shell* is the static part of the FPGA system/bitstream and works as a Hypervisor for vFPGAs. The shell provides the shared infrastructure required for different applications including the I/O virtualization support, resource management, and the drivers required to program the applications. These applications on a single FPGA are realized on partial reconfigurable regions (RRs). Even though virtualization with RRs provides more flexible resource management, a part of FPGA needs to be used for the shell. Multi-node level virtualization involves acceleration jobs across multiple FPGAs.

Virtualization can be done by presenting FPGA resources as partial reconfigurable regions (RRs) and deploying the application on multiple FPGAs [9, 22, 24]. In such an arrangement each RR acts as a single virtual FPGA (vFPGA).

FPGA accelerator realizations are evaluated with benchmark studies. [20] presents a benchmark and performance evaluation on two different platforms. New generation FPGAs [25] for accelerator development for direct cloud computing, which are increasingly used today, are not discussed in this study. In [26], the accelerator is implemented on different cards. Some of the applications discussed are not real applications, they are for measurement purposes only. In this study, a matrix multiplication application is presented.

Here we would like to focus on the realization of three representative applications realized as hardware accelerators on FPGA [27]. We present a comparative summary of these realizations in Table 2.1.

Matrix Multiplication: The matrix multiplication discussed in [26] multiplies 4096x4096 matrices as 8x8 blocks in the single-precision floating-point data type used in [28]. There are 3 pipelines in the accelerator core that are used. Two different implementations were made using DDR and HBM2 memory types. The HBM2-weighted implementation shows higher performance because the memory used is faster. Throughput increased proportionally with frequency.

Face detection: Face detection is performed from 320x240 grayscale images with the Viola-Jones algorithm in [20]. The used Viola-Jones Cascade classifier significantly affects the performance. Existing resources are not sufficient to implement all classifiers in parallel. The first three most invoked stages are performed in parallel and they keep the data in registers. The remaining stages are pipelined and the throughput is increased. The image is processed by dividing it into sub-windows at each clock cycle. The MTCNN (Multi-task Cascaded Convolutional Networks) inputs and outputs use three sequential neural networks in [29]. In their parallelization method, the input and neural network weight values are kept in two different buffers working with the ping-pong method, thereby reducing the memory access delay. The computation time varies with the number and size of faces in the input image. In the reported results, the UltraScale+ VU9P platform used in Amazon Web Services (AWS) F1 type achieves higher performance than the Xilinx ZC706 XC7Z045 platform.

FFT: 1-dimensional FFTs of different sizes are calculated with pipeline configuration for a total of 2 GB data in [26]. Accordingly, eight values are written to the global memory in each clock cycle, and floating-point multiplication is performed as complex as the logarithm of the FFT size (S). The calculation can be made with more than one core. Two memory banks are required per core. The shift register size used affects performance. The DSP library [30] provides a fully synthesizable Super Sample Data Rate (SSR) FFT implementation. The systolic architecture used processes multiple samples in a single clock cycle. The number of samples (P) processed in a clock cycle depends on the SSR factor. FFT is implemented as a C++ template function synthesized into a streaming data-processing structure.

In [26], for the same size FFT, in the HBM2-weighted implementation, using 15 core copies, the throughput increased more than 7 times compared to the DDR-weighted implementation, which is a single-core copy. Implementations of [30] are made for FFT sizes of 1024 and 4096 for different P values. Here, when P increased from 2 to 8, the frequency value did not decrease much, but the throughput increased with the increasing amount of parallel processing. As the value of P increases to 16, the throughput stays almost the same while the frequency drops a lot.

2.2 Workflows and Tools for FPGA Accelerator Realization

There are streamlined workflows for FPGA accelerator realization and interfacing with the CPU. We focus on two workflows in this thesis that we call OpenCL-based flow and IP core generation flow. Both flows feature HLS (High-level Synthesis) tools.

2.2.1 High-level Synthesis (HLS)

High-level synthesis (HLS) is a process that enables hardware-level definitions to be made with high-level languages such as C/C++. HLS translates behavioral codes written in high-level languages into Verilog/VHDL languages with cycle-accurate digital design definitions. HLS is ideal for quickly implementing complex algorithms and DSP applications on hardware that is more difficult to do with HDL. It stands out with its rapid prototyping and easy simulation.

Table 2.1: Specifications of the Selected Hardware Accelerators

Year, Reference	Platform	Workload	Resource Use (LUT, FF: x1000)	Throughput	Frequency
MatriX Multiplication					
2020,[26]	Xilinx Alveo U280 DDR	4096 × 4096 matrix, Block size:8 [28]	LUT(569), FF(442), BRAM(666), DSP(7,683)	266.9 GFLOP/s	100 MHz
	Xilinx Alveo U280 HBM2		LUT(499), FF(920), BRAM(666), DSP(7,683)	603.9 GFLOP/s	236 MHz
Face Detection					
2018,[20]	Xilinx ZC706	320x240, gray-scale, Viola Jones	LUT(63), FF(84), BRAM(121), DSP(79)	30.3 frames/s	140 MHz
	AWS F1 Xilinx Virtex UltraScale+ VU9P		LUT(48), FF(54), BRAM(92), DSP(72)	46.5 frames/s	250 MHz
2019, [29]	Xilinx ZC706	High resolution, color, MTCNN algorithm [31]	LUT(134), FF(222), BRAM(196), DSP(880)	11.9 frames/s	200 MHz
1 dimensional floating point FFT (S: FFT size, K: Number of kernels P: Number of parallel samples processed)					
2020,[26]	Xilinx Alveo U280 DDR	K: 1, S:512 [28]	LUT(83), FF(168), BRAM(39), DSP(672)	78.3 GFLOP/s	248 MHz
	Xilinx Alveo U280 HBM2	K: 15, S:32 [28]	LUT(602), FF(941), BRAM(405), DSP(5,280)	576.2 GFLOP/s	254 MHz
2020,[30]	Xilinx Alveo U250	P:2, S:1024	LUT(45.3), FF(91.3), BRAM(24), DSP(554)	49.9 GFLOP/s	500 MHz
		P:2, S:4096	LUT(66), FF(110), BRAM(50), DSP(670)	57.1 GFLOP/s	476 MHz
		P:8, S:1024	LUT(129.8), FF(247), BRAM(52), DSP(1424)	145.9 GFLOP/s	367 MHz
		P:8, S:4096	LUT(158), FF(268), BRAM(78), DSP(1606)	154.5 GFLOP/s	322 MHz
		P:16, S:1024	LUT(283), FF(520), BRAM(91), DSP(3072)	151.4 GFLOP/s	192 MHz
		P:16, S:4096	LUT(380), FF(592), BRAM(46), DSP(3664)	95.6 GFLOP/s	100 MHz

2.2.2 OpenCL

OpenCL (Open Computing Language) is a framework for writing programs for heterogeneous platforms consisting of central processing units (CPUs), graphics processing units (GPUs), digital signal processors (DSPs), field-programmable gate arrays (FPGAs). OpenCL specifies C/C++-based languages and application programming interfaces (APIs). OpenCL provides a standard interface for parallel computing [32]. The executed functions are called *kernels*. The OpenCL computing model for Xilinx platforms can implement one or more compute kernels on the FPGA.

For the FPGA platform, the data to be processed is first transferred from the host memory to the global FPGA DDR memory. Then, the CPU triggers the compute kernels on the FPGA to start processing. The kernel reads the data from the device memory, processes it, and writes the results back to the device memory. If there is more than one kernel, they can exchange data. Then, the results are transferred from the FPGA global memory to the host memory and are available for the CPU [27, 21].

Xilinx provides an open-source HLS accelerator library called Vitis Libraries together with a unified software platform. This platform supports both the embedded software development flow and the Vitis application acceleration development flow.

Vitis application acceleration development adopts this OpenCL flow [33]. A Vitis accelerated application has a software program and an FPGA binary containing hardware-accelerated kernels provided in the library as shown in Figure 2.1. Here, we note that the kernels in the Vitis Library can be implemented as IP Cores using the IP Core Generation Method.

The software program is written in C/C++ and runs on a conventional CPU. The software program uses user-space APIs implemented by the Xilinx Runtime library (XRT) to interact with the acceleration kernel in the FPGA device. The hardware-accelerated kernels can be written in C/C++ or RTL (Verilog or VHDL) and run within the programmable logic part of the FPGA device. The kernels are integrated with the Vitis hardware platform using standard AXI interfaces.

Host applications can offload functions that create performance bottlenecks in the main application to the FPGA hardware accelerators. Data exchange between hardware accelerator and CPU takes place via DDR memory. There is also an additional direct interface path between CPU and hardware accelerator for control information passing. PCIe attached FPGAs and embedded SoC platforms are suitable for Vitis acceleration development flow.

Vitis acceleration flow requires GNU Linux operating system and Xilinx Runtime Library (XRT) [34] Linux driver to handle high-level OpenCL communication seamlessly between CPU and its couple hardware accelerators. When chosen Vitis acceleration development flow, Vitis HLS generates Xilinx object file (.xo) from HLS C/C++ codes. Xilinx object files include compiled hardware accelerators. Vitis compiler then links *xo* file with target hardware platform to a fully functional platform binary.

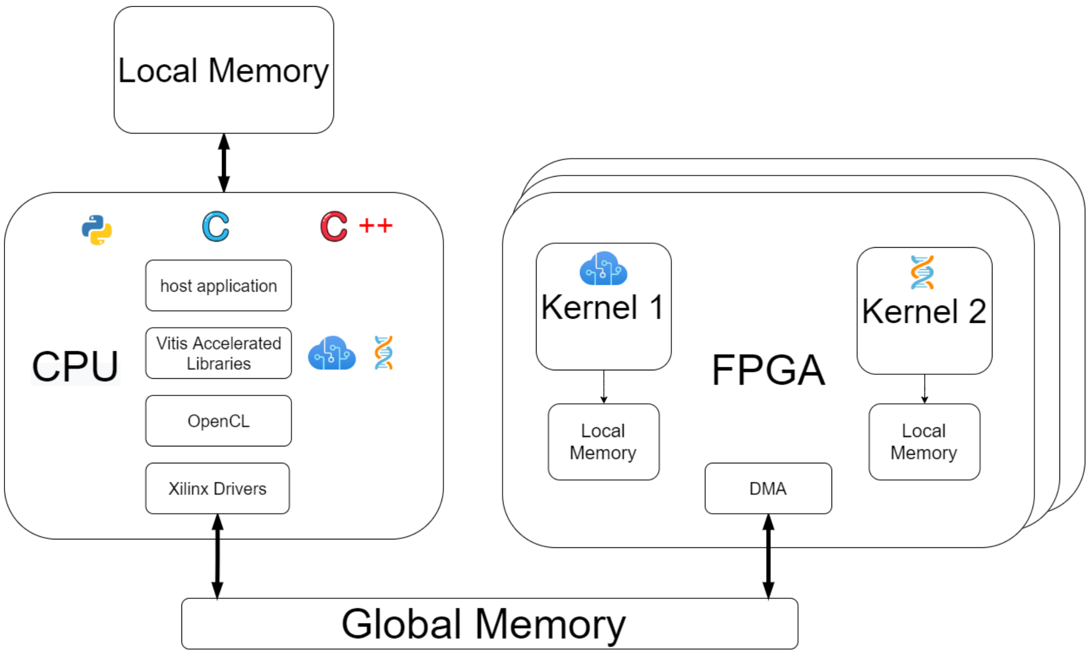


Figure 2.1: Vitis Acceleration Flow [3]

2.2.3 IP Core Generation

Vivado IP generation flow is another hardware accelerator development flow. When configured to generate Vivado IP, Vitis HLS can build standalone hardware accelerators just like a regular Vivado IP that does not require high-level software components like a Linux driver or OpenCL framework. Access to Vivado IPs written in HDL or HLS is usually provided through the AXI interface of the IPs. For Vivado IPs, communication with the CPU or any other FPGA interface is via the AXI interface. Both hardware acceleration flows, Vivado IP generation flow and Vitis application acceleration flow are suitable for hardware kernel development with Xilinx-provided HLS based libraries. Xilinx supports hardware accelerator development on FPGA with Vitis Libraries [35] and aims to streamline and speed up the FPGA design process [36, 37]. Due to the novelty of the libraries, accelerator realizations using Vitis Libraries are limited in the literature. We refer to this method as Vivado IP flow as our development is based on Xilinx ecosystem.

2.3 Hardware Accelerators as Cloud Service

Hardware Accelerated Cloud Data Centers (HACDC) offer hardware accelerators (HA) as computing resources in addition to memory, processor (CPU) and disk [38, 9]. To this end, FPGA Accelerator Cards (FAC) can be installed in the servers or FACs with an SoC processor can be directly connected to the HACDC network. Multiple HAs can be instantiated on partially Reconfigurable Regions (RR) on the same FPGA [11, 24, 39, 40]. Such organization requires high-speed data exchange among FAC components and *Quality of Service (QoS)* support for both satisfying the requirements of the applications and enabling bandwidth allocation for the Virtual Machines [41].

In the study [9], an Altera Stratix V-based hardware accelerator card was added to each compute node. All the network traffic of the compute node is routed from the NIC (Network Interface Card) to the FPGA with a 40 Gbps link. The FPGA is directly connected to the data center switch. There is also a PCIe connection between the CPU and the FPGA in the compute node. In the study, the FPGA can locally accelerate an application running on the machine it is connected to, can be used as a network appliance without putting a load on the CPU thanks to the direct network switch connection, or it can run distributed applications on the FPGA by communicating with other FPGAs to which it is connected with the network switch.

In this study, FPGA partial reconfiguration and FPGA working alone without a server are not considered.

[42] empirically quantifies the FPGA instance performances of three major cloud providers, namely, Amazon AWS, Alibaba and Huawei clouds over one year. They observe that on average, the two compute-intensive workloads can achieve up to 40 times speed-up with respect to CPU implementations. This speed-up decreases if the application is communication intensive. Here we provide the specifications of these instances. Amazon EC2 F1.2xlarge (AWS F1), Alibaba Cloud ECS.f1-c8f1.2xlarge (Ali F1) and Alibaba Cloud ECS.f2-c8f1.2xlarge (Ali F2) have Intel Xeon E5-2686v4 processor and and Xilinx VU9P, Intel Arria 10 and Xilinx KU115 FPGA cards respectively. Alibaba Cloud ECS.f3-c8f1.2xlarge (Ali F3) and Huawei Cloud fp1c.2xlarge.11 (HW F1) both have Xilinx VU9P FPGA Card with Intel Xeon Platinum 8163 and Intel Xeon E5-2697v4 processors respectively.

The abstraction, virtualization, and distribution of cloud resources to users involve many sub-problems. For this purpose, cloud computing control platforms have been developed. These platforms act as an operating system and offer cloud resources to users. Among these platforms is Openstack [6], an open-source platform used in cloud-based data centers and private cloud systems for Infrastructure as a Service (IaaS) type services. OpenStack software runs distributed on the controller and compute nodes. Accordingly, the OpenStack controller node is responsible for the messaging infrastructure and database management. Computing nodes are also responsible for providing cloud services to the user by communicating with the controller node. OpenStack consists of sub-software components called projects.

Keystone, Glance, Placement, Nova-Controller projects run on the controller node. The Keystone project is used as an authorization service in almost all of the OpenStack architecture. In order to make any API call within the OpenStack architecture, a current and valid token is taken from the Keystone service, and the validity of the relevant token is checked at the place where the call is received. The Glance project is a project that manages the image files of the virtual machines to be created. Virtual machines to be created must be initialized according to the relevant image and this capability is provided by Glance. Neutron project provides network management in OpenStack architecture. It can create virtual networks on top of physical networks for created virtual machines. Placement project is a project responsible for resource management of OpenStack architecture. Recognition of heterogeneous physical resources such as Placement project FPGA and GPU in OpenStack allows different physical resources to be used. Placement also provides an API that allows incoming virtual machine requests to be initiated on explicitly requested servers without forwarding them to Nova Scheduler. Nova-Controller is responsible for the creation and maintenance of physical machines. A component of Nova-Compute and Neutron is running on the compute node. According to the orders from the Nova-Controller in the controller node, the relevant physical resources are assigned for the creation of virtual machines. Virtual machines are created and initialized according to their respective Glance images.

[43] utilizes the multiple reconfigurable regions on the FPGA and assigns them to the user as IaaS/HaaS (Hardware as a Service) using OpenStack. For this purpose, it runs on a software computing node with a function similar to the hypervisor and communicates with OpenStack to configure the RRs according to the requests from the user. Users can write their bitstreams on the FPGA. Implementation is done on Xilinx Virtex V-based NetFPGA. In this study, the FPGA board's independent operation from the CPU and performing network functions are not investigated. The work is focused on OpenStack-managed IaaS applications. It is not explained how FPGA resources are defined and assigned in the OpenStack nova scheduler. A method that decides which resources to use by evaluating the virtual machine and FPGA resources in the cloud computing system has not been defined.

[44] suggests hypervisor-like application implementation for FPGA in Xilinx SDAccel environment. Partial reconfiguration is not studied. This work implements only pre-designed hardware accelerators on FPGA via OpenStack. There is no offering of RRs as IaaS/HaaS to the user to run its bitstream. Implementation is done on Xilinx Kintex-7 XC7K325T FPGA with 5Gbps network speed. Both [43] and [44] use the cloud server's NIC for network access which slows down the CPU.

2.4 ACCLOUD FPGA Platform

The work in this thesis is developed in the scope of a research project titled "ACCLOUD (Accelerated Cloud): A Novel, FPGA-Accelerated Cloud Architecture" which offers users hardware accelerators implemented on FPGA as computing resources with an innovative cloud computing approach [11]. ACCLOUD project proposes the virtualization of FPGA resources as reconfigurable regions (RRs) and providing them through the OpenStack. The OpenStack module in the SoC processor is implemented as embedded software that works with other OpenStack modules. An accelerator image selected by the user can be written on the RR. The FPGA platform in this architecture also acts as Network Interface with 40 Gbps Ethernet IP Cores. An on-chip switch is developed that provides 40 Gbps communication between the RRs on the FPGA, PCIe, DDR, SoC processor, and 40 Gbps Ethernet. The entire architecture shown in Figure 2.2 is implemented on the ZC-706 board with Xilinx XC7Z045 FPGA and tested in the laboratory environment with cloud servers and a 40Gbps Ethernet switch for functional correctness and performance.

FPGA Switch, is an on-chip switch [45, 46, 47] that is developed within the scope of ACCLOUD architecture. The FPGA Switch consists of 7 input/output ports. Each port can carry 256-bit wide data at 156.25 MHz (40 Gbps). Figure 2.2 shows the number of each port (0-6).

RR0 and RR1, reconfigurable regions are the physical areas where accelerator applications run. RR content(RM) can be changed (with partial reconfiguration or dynamic function exchange) without requiring a whole system reset or affecting the static FPGA shell during its operation. RR areas and RM accelerators are designed within the scope of this thesis. RMs are connected to the switch with Partial Region Decoupler blocks. These blocks are available on Vivado, the IP blocks offered by Xilinx; they are configured to be working suitable with RM interfaces. The purpose of the decoupler blocks, which have a fundamental and straightforward function, is to prevent the formation of "spurious" packets from RR to the switch or from the switch to RMs in the "transient" period. At the same time, the contents of the RRs are changed during the reconfiguration. When the specialized Nova-compute

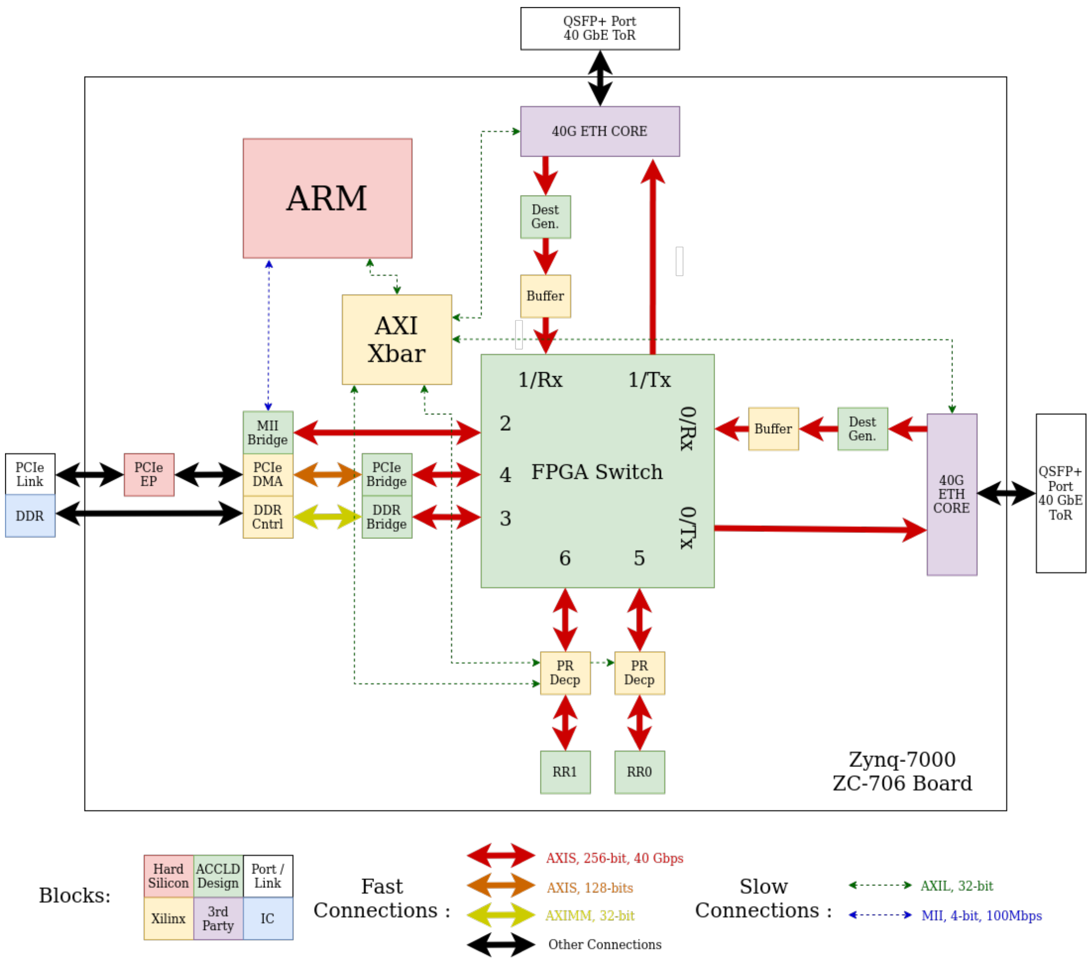


Figure 2.2: ACCLOUD FPGA Platform.

software running on ARM CPU configures the RR, it isolates the connections between the RMs and the switch by taking the Decoupler blocks to the "decouple" mode from the AXIL interface. The links are re-established when the configuration process completes by switching to the "couple" mode from the same interface.

In Figure 2.2, one interface of the ARM Block comes out of the FPGA and is directly connected to the QSFP+ port, where 40 Gbps Ethernet cables are inserted. The other side is going to the switch. This block consists of sub-blocks (7-8 pieces). In order not to complicate the drawing, the details are not shown. Each sub-block is an independent design. Sub-blocks such as MAC, PCS, PMA, PMD, Statistics Monitor are basic blocks. Each sub-block is configured according to the project's requirements with the software we prepared on the ARM processor from the AXI Lite interface (settings such as Ethernet Padding, CRC Stripping).

The buffer memory on the receiving side of the 40G Ethernet Core is relatively tiny. For this reason, data buffers are placed to reduce the possibility of packet dropping on the receiving side. For this part, we used the AXI Stream FIFO IP core offered by Xilinx, and we adjusted the IP Core buffer depth settings according to the design needs. Another module called Dest Generator was designed on the receiving side between Buffer and 40G Ethernet Core. In the ACCLOUD system, each Switch port in the FPGA has a MAC address. The purpose of this module is to determine which port of the Switch the incoming packet will be forwarded to by examining the destination MAC address of the incoming Ethernet packets (generating the AXI Stream interface tdest signal).

One of the interfaces connected to the switch is from DDR memory. Xilinx offers DDR Cntrl as a Vivado IP block as shown in the Figure 2.2 named as DDR controller block. Memory Interface Generator (MIG) 7 IP Core, established as the part of this IP core that talks to RAM chips implement the DDR protocol, while the other interface is AXIM (AXIMM). The module implements the DDR protocol and acts as a "memory controller", it sorts the requests coming from the AXIM memory-mapped interface and schedules the responses (write/read). This IP core card offered by Xilinx has been adjusted according to the DDR ICs used and the project requirements. The user interface of this block is AXIM, but the commonly used interface in ACCLOUD is AXIS. For this reason, a module called DDR Bridge is designed. The primary function of the DDR Bridge is to transform AXIS <-> AXIM interfaces following the ACCLOUD protocol. Since the module works following the specified protocol, it allows direct access to DDR from 40 Gpbs Ethernet ports (read/write). This feature also works as a simple Remote DMA (Direct Memory Access) Engine (RDMA Engine). With all this infrastructure, DDR memory content (write/read) can be accessed and modified directly from RMs without using any extra resource, directly by the host computer, if any, via PCIe, and from any of the two 40 Gbps Ethernet ports with RDMA logic without using any other intermediary.

In the ACCLOUD architecture, it is recommended to use FPGA cards standalone network-attached or by connecting them to a processor as an accelerator over PCIe. The PCIe connection is an interface that makes sense when the FPGA card is attached to a workstation. There is a hard-IP design embedded in silicon in the Zynq-7000 series for PCIe end-point functionality. This IP silicon is called PCIe E.P. as shown in Figure 2.2. The PCIe DMA module offered by Xilinx, using this module and working in harmony with the PCIe driver on the computer side, has been used in the design. PCIe DMA IP Core is also configured to project design requirements. These two IP blocks provide double-sided data flow between the FPGA and the computer CPU with the OS driver support on the computer side. Software drivers provided by Xilinx are used on the computer side. Although the user interface of the PCIe DMA module is AXIS, which is the primary interface, the bus width (128-bit) and data structure are not directly compatible with the AXIS structure we designed (256-bit, ACCLOUD protocol) for the ACCLOUD FPGA project. For this reason, the PCIe Bridge module is designed. The main function of this module is to match two AXIS interfaces with different features. All other ports (6 pcs) connected to the PCIe interface and Switch can send and receive data. Therefore, like the Dest Generator on the Ethernet side, this module multiplexes the traffic from all ports before sending it over PCIe and demultiplexes the packets coming from the computer according to the packet contents allowing the Switch to exit from the correct port. Like DDR, thanks to the ACCLOUD protocol, the PCIe interface can be accessed from all RMs and two 40 Gbps Ethernet ports (accessing a remote computer via Ethernet without any extra modules).

There is an ARM processor on the Zynq-7000 as hard silicon. We developed a GNU Linux distribution with the Xilinx Petalinux tool and named Zynqos runs on this processor. In this project, the main task of the processor is to talk to the Openstack Controller with the help of Nova-Zynq and configure RRs according to incoming hardware accelerator requests from Cloud Interface. In addition, the processor can change the configuration settings of all the IP modules that can be configured with the AXIL control interface. For AXIL connections, the infrastructure provided by Xilinx, shown as AXI Xbar in the Figure 2.2, made up of sub-blocks such as AXI Interconnect and Protocol Converter on Xilinx Vivado is used. The ARM processor needs to talk to the OpenStack Controller. For this, the network interface connection in the MII standard supported by ARM is included in the FPGA. To keep the implementation simple and while the speed is sufficient, MII operating at 100 Mbps was preferred instead of the GMII interface, which increased to 1 Gbps. ARM does not need higher bandwidth as only control messages going in and out from the Controller forwarded. The MII interface includes a 4-bit wide bus running at 25 MHz in both reception and transmission directions. Since our FPGA design uses 256-bit wide, 40 Gbps data connections built on AXIS, we have developed the MII Bridge module to harmonize these two interfaces.

2.4.1 ACCLOUD On-chip Message Protocol

FPGA design processes include also the ACCLOUD message protocol design. This message protocol ensures that the internal switch, HDL blocks, and even other boards or host applications designed ACCLOUD message protocol in mind works in harmony. ACCLOUD message protocol enables, for example, the DDR memory of the card can be accessed directly from a remote computer via the Ethernet connection of an FPGA card, and cloud jobs can be sent and received to the hardware accelerators running on the FPGA in RRs.

In FPGA, messages are transmitted with AXI Stream. Therefore, Ethernet frames and PCIe packets coming from physical external interfaces to the FPGA should be converted into AXI Stream data and AXI Stream side-channel signals. The Internal Switch does not care about the message content, it performs forwarding by looking at the value of the AXI Stream tdest signal. The "Dest Gen.", "DDR Bridge", and "PCIe Bridge" modules, located between the internal switch and Ethernet, PCIe and DDR, make sense of the relevant parts of the message packets and extract the AXI Stream tdest signal from the packet content. For example, if an Ethernet frame contains the destination MAC address "0x04, 0x00, 0x00, 0x05, 0x00, 0x00", "Dest Gen." It detects tdest as 5 by looking at the 4th digit of the MAC address and the internal switch forwards these message packets to RR0. If the destination MAC address is "0x04, 0x00, 0x00, 0x06, 0x00, 0x00" then "Dest Gen." It detects tdest as 6 and the internal switch forwards these message packets to RR1. Similarly, in PCIe and DDR messages, relevant signals and information are embedded in predetermined fixed fields in the message content as shown in Figure 2.3.

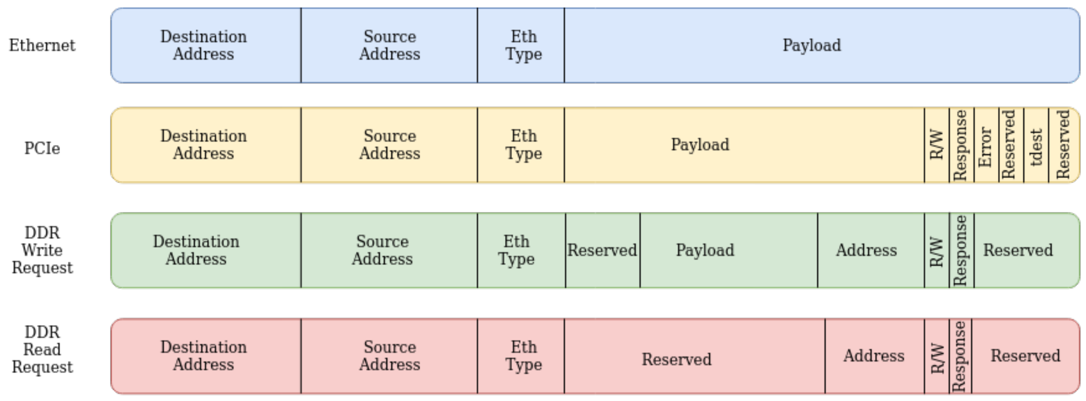


Figure 2.3: ACCLOUD Protocol Packet Structures [4].

2.4.2 OpenStack Nova Compute Redesigned for Heterogeneous Resources

[11] proposes that FPGA accelerators can work alone apart from the server. In this case, OpenStack Nova Compute must run on the processor on the SoC. The capabilities of this processor and the special operating system running on it may not be compatible with the operating system and hypervisor software used in standard cloud servers. To this end, [48, 5, 49] propose an extension of Nova Compute to incorporate FPGA hardware accelerators in cloud resources. In this work, a new lightweight project called Nova-G Compute is designed to replace the standard Nova Compute for such heterogeneous hardware platforms. Nova-G Compute can work with standard OpenStack projects by sending and receiving messages correctly. The implementation of Nova-G is in Python and is OS-independent. In this way, different hardware platforms can be supported.

Nova-G Compute can work with standard OpenStack projects by sending and receiving messages correctly. The implementation of Nova-G is in Python and is OS independent. In this way, different hardware platforms can be supported. The block diagram of Nova-G Compute is presented in Figure 2.4.

Nova-G Compute communicates with other OpenStack projects using Rabbit Message Queue as in the standard Nova application. The Nova-G Compute Message Encoder module properly formats the messages and sends them to Rabbit-MQ. Similarly, the Message Decoder module receives Messages from Rabbit-MQ and parses them properly. Then, necessary operations are performed by Nova-G

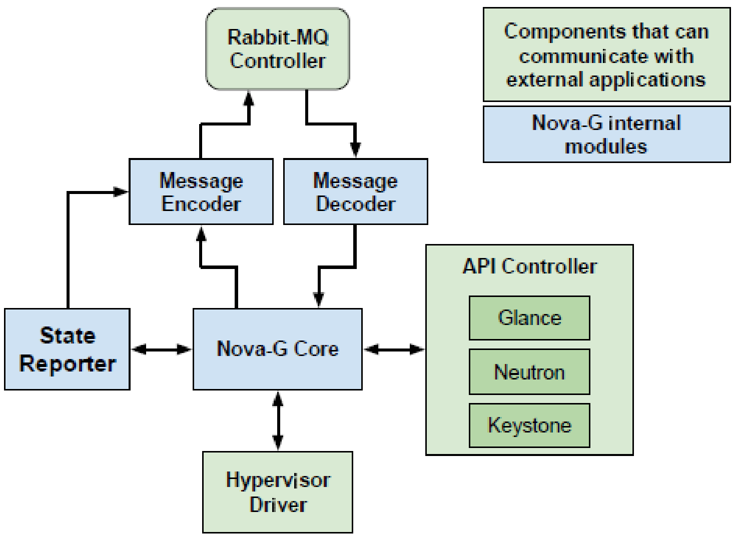


Figure 2.4: Nova-G Compute [5].

Compute according to the content of the message. Standard OpenStack messages are supported with extensions flavor data structures of the source database. OpenStack communication is generally based on RPC (Remote Process Communication). The Nova-G Compute core module can respond to these RPC requests and make RPC requests to other OpenStack components.

The Nova-G Compute module can use APIs of other OpenStack components. Nova-G Compute uses the Glance API to get the required VM (Virtual Machine) images.

Standard Nova Compute supports predefined hypervisors running on standard server hardware. Nova-G Compute is designed to support non-standard hardware. To this end, proper support is required for virtualizing this hardware with custom hypervisors. For this purpose, a hypervisor driver module has been developed to abstract the Nova-G Compute application from the custom hypervisor. The Nova-G Compute kernel collects all the information and provides it to the hypervisor driver. The hypervisor driver issues commands to the hypervisor specific for the particular hardware type to start the desired VM. Also, the VM gets its state information from the custom hypervisor.

CHAPTER 3

PROPOSED WORKFLOW FOR OFFERING HARDWARE ACCELERATORS ON FPGA AS CLOUD COMPUTING SERVICE

The proposed workflow in this thesis consists of offering the FPGA Partial Reconfigurable Regions (RRs) as hardware resources to the cloud users through the OpenStack resource management platform and realization of the hardware accelerators on the RRs that are suitable to offer as cloud computing resources. To this end, we develop the OpenStack Nova Compute project on the Zynq ARM SoC processor and the hypervisor that enables virtualization of the FPGA RRs and offers them to the cloud users. We then present accelerator development flows by realizing example applications. To this end, we propose the development of Reconfigurable Modules (RMs) residing in the RRs and an access method to RMs. We select image processing applications that can be distributed over two RMs to demonstrate how accelerators on the same FPGA chip can work together by communication through the FPGA switch. To this end, we implement a Gaussian Filter and a Sobel Filter using the IP Core Generation Method. We also demonstrate the bump-in-the-wire packet processing by implementing a representative Ethernet Packet Processor. We then implement a Canny Edge Detector which is composed of two compute kernels realized on two RRs using the OpenCL API Based Method. We describe each software and hardware component in the workflow in the remaining part of this section.

3.1 Integrating FPGA Accelerators in OpenStack as a Computing Resource

In the scope of this thesis, we implemented a specific Nova Compute to run on Zynq FPGA that we call *Nova-Zynq Compute* for integrating FPGA accelerators as cloud-computing resources. The development of Nova-Zynq Compute follows the Nova-G architecture proposed in [5].

Nova-Zynq Compute requires extensions to Nova-G to run on real hardware. Like other OpenStack services, Nova Compute exchanges messages via RabbitMQ message queues. Messages are always published to "exchanges" by services and exchanges push messages to queues. The services than grabs the messages by reading from their queues. In this context, the ability to create a "queue" and connect with the relevant "exchange" is added to Nova-Zynq so that it can communicate with RabbitMQ [50] in cases where it starts before the Openstack controller. In cases where the connection with the Nova controller is not found and disconnected, the errors that occur in Nova-Zynq are stabilized and Nova-Zynq is kept in a standby state until the controller or the connection is active again.

Nova-Zynq Compute requires a particular image that have been given the hypervisor that can communicate with the hypervisor drivers according to the generalized OpenStack resource databases. In this thesis, FPGA is defined as a generalized resource. Accordingly, a custom hypervisor called *FPGAvisor* is developed to demonstrate the features of our proposed extensions and Nova-Zynq Compute. To this end, partially reconfigurable FPGA regions (RRs) are considered a single generalized resource unit. Each RR can be programmed separately by FPGAvisor.

Nova-Zynq Compute's hypervisor driver controls FPGAvisor. Nova-Zynq Compute presents the unique image for the reconfigurable region of the FPGA to the FPGAvisor via the hypervisor driver. FPGAvisor is the closest software component to the FPGA hardware of Nova-Zynq software. Its connection and location are indicated in Figure 3.1. Next, FPGAvisor programs the selected region with the FPGA image provided by Glance via the FPGA Manager kernel driver. The programmed image can provide data exchange with the static design of the FPGA to which it is connected via the AXI Stream interface. AXI Stream interface of the programmed image; thanks to the MAC, IP and AXI Stream address it contains, it can work in harmony with PCIe and Ethernet protocols.

In our prototype system, we have the ZC706 SoC FPGA board with Xilinx Zynq-7000 [2]. Zynq has a dual-core ARM Cortex-A9 processor embedded in silicon. In addition to the processor, there is an FPGA area with Xilinx 7 series architecture.

The main task of the ARM processor on the FPGA is to communicate with the OpenStack controller and fulfill the incoming requests for RRs. The software running on the processor configures the RRs with the appropriate bitstreams (which can be thought of as an image file for the FPGA). At the same time, it also adjusts the settings of static designs, like IPs with AXI Lite control and configuration interfaces.

Nova-Zynq compute runs on a GNU Linux distribution installed in the Zynq-ARM processor. Considering the software features running on the Zynq ARM processor and the development environment, we consider it appropriate to run an embedded Linux on the ARM processor. Xilinx recommends its tool called Petalinux [51] to developers who want to create an embedded Linux distribution on their products. Petalinux provides opportunities such as creating a device tree (DTS) suitable for embedded Linux in accordance with the FPGA design, adding the relevant drivers to the distribution, easily configuring the related components to the file system (i.e., rootfs) in the created Linux image. Along with Petalinux comes a component called *FPGA Manager*, which is developed by Xilinx, which also provides the opportunity to upload images (bitstream) to the FPGA area on Zynq. For these reasons, the Petalinux tool is preferred to create a GNU Linux distribution in this project.

Nova-Zynq is developed in Python language. Its main task is to respond to the requests of the OpenStack controller and configure the FPGA side accordingly. Its architectural drawing is given in Figure 3.1. “FPGA Visor” is located in Nova-Zynq code and communicates with the FPGA Manager. “FPGA Manager” [52] performs dynamic reconfiguration of RR’s with PCAP interface.

OpenStack components use a communication data structure called RabbitMQ among themselves. Messages are transmitted in JSON format. When Petalinux is used to compile an embedded Linux with default settings, RabbitMQ and JSON-related components are not available in the rootfs. The python-JSON library is added to the Linux distribution to process messages in JSON format. The JSON library is a library that can be selected and distributed with the Petalinux configuration. We use Pika [53], a Python library that implements the AMQP protocol to realize RabbitMQ communication. This library is downloaded from the source code and added to the Linux distribution. We tested that Openstack messages can be interpreted with the added libraries. Petalinux uses a compilation tool called BitBake in the background, which is a component of the Yocto project [54]. With this tool, the components in the custom GNU Linux distribution are compiled and packaged by properly resolving their dependencies. Nova-Zynq software is also added in the final distribution via BitBake recipes in the Linux compilation process.

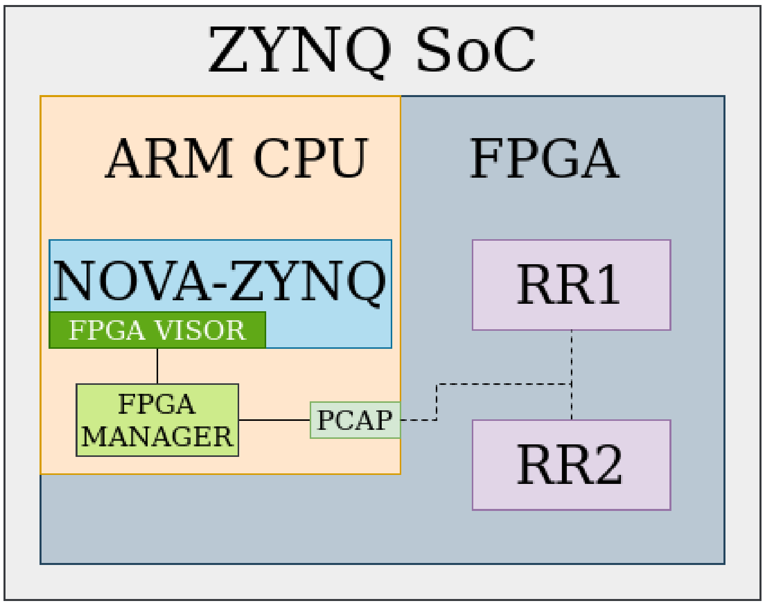


Figure 3.1: Nova-Zynq and FPGA Software architecture.

The RabbitMQ messaging structure, which supports the AMQP protocol used by OpenStack, is first created, virtualized and emulated very close to the actual system to be installed. The test setup is shown in Figure 3.2. There is an Embedded Linux built with Petalinux on ARM Cortex A9 processor virtualized on QEMU and Nova-Zynq software on it. The only deviation from the actual physical system is the FPGAVisor. Except for the FPGAVisor component, all Nova-Zynq components that communicate with Openstack are tested. The test environment verified that the controller computer running Nova Compute and the ARM processor on the Zynq running the embedded Linux distribution we created could talk to each other on the network according to the standard OpenStack APIs [55].

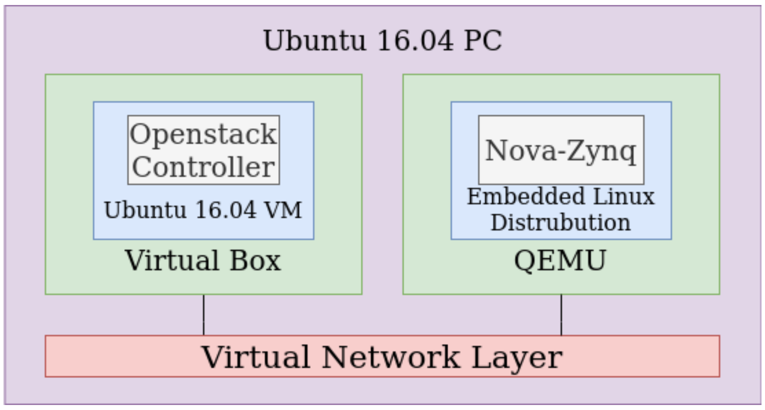


Figure 3.2: Nova-Zynq and Openstack Controller Test on Virtual Environment

FPGA-Visor software is a module that is created as a `HyperVisorDriver.py` file inside the Nova-Zynq code and consists of shell commands. FPGA Manager performs the command transfer to the Linux kernel driver for the configuration of RR via the following basic Python code snippet in Listing 1.

These commands provide the FPGA Manager's RR configuration by passing the commands in parentheses to the Linux Shell. PR Decoupler IPs are at addresses "0x43c20000" and "0x43c30000" in the PS address map. Therefore, PR Decouplers are activated during partial reconfiguration. A waiting time of 5 seconds is set to ensure that the Decoupling process is completed correctly. There is particular FPGA partial configuration commands passed to the FPGA Manager driver in the remaining part.

```
os.system('devmem 0x43c20000 w 1 && devmem 0x43c30000 w 1 && sleep 5')
os.system('echo 1 > /sys/class/fpga_manager/fpga0/flags')
os.system('mkdir -p /lib/firmware && cp image_received.bin /lib/firmware/
↪  && sync && echo image_received.bin >
↪  /sys/class/fpga_manager/fpga0/firmware')
os.system('sleep 5 && devmem 0x43c20000 w 0 && devmem 0x43c30000 w 0')
```

Listing 1: Python Code Snippet for Hypervisor

3.2 Accelerator Development with IP Core Generation Method

3.2.1 FPGA Static Shell

The static FPGA shell design includes core functionalities to provide a data forwarding mechanism between reconfigurable module blocks, ARM CPU, 40 G Ethernet interface, DDR memory, and PCIe interface. FPGA switch in Figure 2.2 has a data width of 256-bits. The switch has seven ports with AXI Stream interfaces. All the message traffic in the FPGA passes through the internal switch.

RRs are connected to switch through Partial Decoupler Modules. Partial Decouplers isolate the unstable AXI Stream data interface of RRs during reconfiguration. Section 2.4 discusses static shell components in more detail.

3.2.2 Reconfigurable Regions and Modules

A Reconfigurable Region (RR) [56] defines FPGA resources bounded with a user-defined square box in the physical placement of FPGA resources. Reconfigurable Modules (RMs) are behavioral descriptions of hardware logic resources within a RR. RRs that exist in complete FPGA design are runtime reconfigurable. While the rest of the static design operates, RRs can be reconfigured with compatible RMs. Hardware accelerators are physically located in RRs, and their netlists terminologically correspond to RM.

In this thesis, hardware accelerator modules have standard interfaces to the rest of the FPGA static design. Accelerator modules are wrapped with an AXI Stream interface. Hardware accelerator samples have been determined and developed to be used in evaluation for two different RRs defined on the FPGA. The hardware accelerator instances are developed to have a standard interface to be programmed into the identical RRs to replace each other. Since the ports of the internal switch structure in the FPGA project to which the RMs will be connected supports the AXI Stream protocol, the RM and the hardware accelerator interfaces are AXI Stream. The simplified structure of the RM is shown in Figure 3.3. Activation of the HLS hardware accelerator while the complete FPGA project is running, assignments such as clock assignment, AXI Stream source, and destination addresses are in the RM shell written in the Verilog/HLS language shown as “RM Wrapper“ and hardware accelerators written in C/C++ with the HLS concept are Vivado IP.

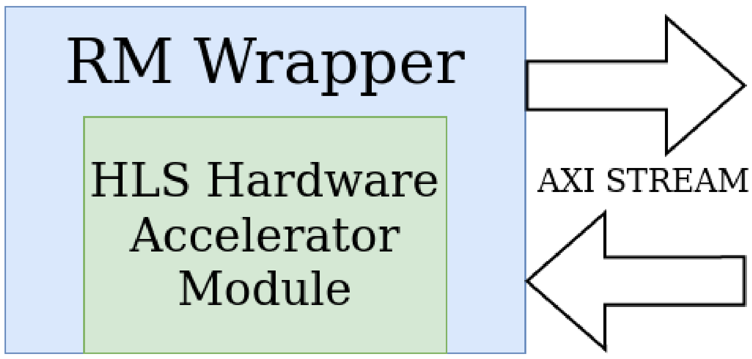


Figure 3.3: Reconfigurable Module Structure

3.2.3 The Sobel and the Gaussian Filter Examples

We implement Gaussian Filter and Sobel Filter accelerators which are hardware accelerator kernels in Xilinx’s Vitis Libraries [35]. We choose these two image filters to represent a distributed hardware accelerator application that is implemented on more than one RRs. The frames that are received onto the FPGA go through an image processing pipeline consisting of these two filters consecutively and then transmitted out of the FPGA. The interconnection between the Input Ethernet interface, the Gaussian Filter, the Sobel Filter and the Ethernet Interface is realized by the FPGA switch in Figure 2.2.

We use the HLS C/C++ language with reference to Vitis Libraries. Wrapper function prototypes of two hardware accelerator examples are given in Listing 2 below. They have an AXI Stream interface as in Figure 3.3, which is compatible with the FPGA switch. On the standalone Vitis HLS IDE, design and simulations are conducted. Vitis HLS IDE is a standalone tool to develop hardware accelerator kernels that comes with Vitis installation similar to other Vivado, Petalinux, Vitis tools as shown in Figure 1.1.

Gaussian filter and Sobel filter are examples of edge detection applications widely used in image processing. Usually, the picture is first blurred and smoothed through a Gaussian filter and then passed through a Sobel filter for edge detection. These

```
void gaussian_filter_accel(hls::stream<ap_axiu<_WA, 1, 4, 4> >& img_inp,
 ↪  hls::stream<ap_axiu<_WA, 1, 4, 4> >& img_out)

void sobel_filter_accel(hls::stream<ap_axiu<_WA, 1, 4, 4> >& img_inp,
 ↪  hls::stream<ap_axiu<_WA, 1, 4, 4> >& img_out)
```

Listing 2: Function Prototypes for the Wrappers of Gaussian and Sobel Filters in HLS

accelerators are selected to demonstrate that two RRs can both exchange data and work independently. To this end, they are realized as RMs in separate RRs. The filters are used to support single-channel image processing. The processed images are grayscale.

Function prototypes of Gaussian and Sobel Hardware Accelerator Modules are given in Listing 3. After the wrapper function receives the data from the FPGA switch, it extracts the pixel values from the data and transmits them to the HLS accelerator module. The Sobel Filter accelerator module calculates a frame's gradients in the x-axis and y-axis. Then, the gradients created with the Sobel filter for both the x-axis and the y-axis were combined with the `addWeighted` function in equal weights to form the single processed frame.

Table 3.1: Parameter Definitions for the Gaussian and Sobel Filter HLS Accelerators [1]

Parameter	Description
FILTER_SIZE, FILTER_TYPE	Filter size. Filter size of 3 (XF_FILTER_3X3), 5 (XF_FILTER_5X5) and 7 (XF_FILTER_7X7) are supported.
BORDER_TYPE	Border type supported is XF_BORDER_CONSTANT
SRC_T	Input and Output pixel type. Only 8-bit, unsigned, 1 and 3 channels are supported (XF_8UC1 and XF_8UC3)
DST_T	Output pixel type. Only 8-bit unsigned, 16-bit signed,1 and 3 channels are supported (XF_8UC1, XF_16SC1,XF_8UC3 and XF_16SC3)
ROWS	Maximum height of input and output image.
COLS	Maximum width of input and output image (must be a multiple of 8, for 8-pixel operation)
NPC	Number of pixels to be processed per cycle; possible values are XF_NPPC1 and XF_NPPC8 for 1 pixel and 8 pixel operations respectively.
USE_URAM	Enable to map storage structures to UltraRAM
src	Input image
dst	Output image
_dst_matx	X gradient output image.
_dst_maty	Y gradient output image.
sigma	Standard deviation of Gaussian filter
Alpha	Weight applied on first image
Beta	Weight applied on second image
gamma	Scalar added to each sum

```
template<int FILTER_SIZE, int BORDER_TYPE, int SRC_T, int ROWS, int COLS,
↪  int NPC = 1>
void GaussianBlur(xf::cv::Mat<SRC_T, ROWS, COLS, NPC> & src,
↪  xf::cv::Mat<SRC_T, ROWS, COLS, NPC> & dst, float sigma)

template<int BORDER_TYPE,int FILTER_TYPE, int SRC_T,int DST_T, int ROWS,
↪  int COLS,int NPC=1,bool USE_URAM=false>
void Sobel(xf::cv::Mat<SRC_T, ROWS, COLS, NPC> &
↪  _src_mat,xf::cv::Mat<DST_T, ROWS, COLS, NPC> &
↪  _dst_matx,xf::cv::Mat<DST_T, ROWS, COLS, NPC> & _dst_maty)

template< int SRC_T , int DST_T,   int ROWS, int COLS, int NPC=1>
void addWeighted(xf::cv::Mat<SRC_T, ROWS, COLS, NPC> & _src1, float alpha,
↪  xf::cv::Mat<SRC_T, ROWS, COLS, NPC> & _src2, float beta, float gamma,
↪  xf::cv::Mat<SRC_T, ROWS, COLS, NPC> & _dst)
```

Listing 3: Function prototypes of Gaussian and Sobel Hardware Accelerator Modules in HLS OpenCV

3.2.4 Ethernet Frame Processor Accelerator Example

The FPGA board with the 40 Gbps Ethernet interfaces provides the Network Interface to the cloud server. To this end, network packet processing functions can be implemented as accelerators on the FPGA without slowing down the traffic with CPU interference. The second accelerator example is designed to show that Ethernet frames can be processed transparently in hardware accelerators on FPGA and implemented using HLS. In this example, the payload has a number. The hardware accelerator modifies this number in the payload by multiplying it by four and adding five to it without interfering with the Ethernet destination, Ethernet source, and protocol addresses.

```
void eth_pack_proc(hls::stream< ap_axiu<256,1,4,4> > &src, hls::stream<
↪  ap_axiu<256,1,4,4> > &dst)
```

Listing 4: Function Prototype for Ethernet Frame Processor in HLS

3.3 Accelerator Development with OpenCL Based API: Canny Edge Detector

Hardware acceleration aims to increase the data throughput and reducing the response time. Vitis acceleration flow provides a framework to accelerate overall C/C++ host code by offloading its suitable functions to FPGA. Code profiling tools can evaluate C/C++ host code to find appropriate functions that create performance bottlenecks, have the potential for concurrent execution, require data access locally more, and are compute-intensive. These functions then can be implemented to work on the FPGA accelerator kernel with HLS or HDL. The host application uses OpenCL API [57] calls or Xilinx Runtime (XRT) [34] to communicate and control hardware accelerator kernels running on the FPGA regions. XRT combines GNU Linux user and kernel-level drivers to connect PL and application processors. Therefore, Vitis acceleration flow requires GNU Linux-based operating system to run the host application. XRT can communicate between the host and the kernel through PCIe or AXI interfaces. PCIe interface is used when the hardware platform is a PCIe-attached FPGA to CPU. The communication channel is the AXI interface when the hardware platform is Zynq SoC. Data flow between hardware accelerator kernels, and application processors occur through global memory. On the other hand, the control flow occurs through access ports and the AXI Lite interface of hardware accelerator kernels. A typical acceleration follows that the host application first transfers data into global memory and initiates the kernel to operate on the data; the hardware accelerator kernel then stores results back into the global memory. Upon kernel completion, the host fetches the results back.

Our third accelerator implementation is the Canny Edge Detector on Xilinx ZC706 [27]. The CED we implemented is derived from the open-source code in the HLS language available in Vitis Library [58].

The CED application is a system application on a chip that is compiled with two streams, the main code (MC) to run on the ARM processor on the ZC706 platform and the hardware logic design to run on the FPGA area. The application developed with C code to run on ARM is compiled with the "arm-linux-gnueabihf-g++" cross compiler. The hardware that runs on the FPGA area is created with the Vitis v++ compiler. As in other hardware accelerators, the data flow control is done by the application running on the processor while the calculations are made on the FPGA.

CED comprises two Kernels; hence similar to the Gaussian and Sobel Filters, it is a representative application for multiple accelerators distributed over different FPGAs or FPGA RRs for more scalable cloud services.

The CED implementation is performed with two hardware accelerator kernels (HAK) running on the FPGA. The Canny Kernel (CK) performs edge detection, while the Edge Tracer Kernel (ETK) connects the edges between strong pixels. A 3x3 Gaussian mask and Sobel Filter are applied to the input image (II) for edge detection. Weak pixels found by bidirectional gradient calculation are removed from the image using "Non-maximal suppression". Then, in the ETK, the connection of the edges of strong pixels is completed.

CK and ETK work by communicating with each other and with the MC running on the processor using OpenCL over AXI interfaces, as seen in Figure 3.4.

Using the write and read functions of the OpenCV library, MC reads n images to be processed from the hard disk into a matrix array with `imread`, and the images processed in HAKs are written to the processor global memory as a matrix array, and then sent to the hard disk using the `imwrite` method.

Each input image in the matrix array $II_i, i = 1, \cdots, n$ is written by MC with $\mathtt{W}_i$ over the AXI Stream interface to the global memory of the FPGA. MC transmits the II_i start address, output image OI_i start address in the global memory, image size, and filter parameters to HAK over the AXI Lite interface. The MC triggers the HAKs to start working over the AXI Lite interface. CK processes the II_i in the global memory and transmits the generated II_i^{CK} over the AXI Stream interface to ETK for processing. ETK writes the OI_i generated after the processing to the global memory and transmits the completion of the process to MC over the AXI Lite interface. MC reads OI_i from FPGA's global memory with $\mathtt{R}_i$ and writes it into processor global memory.

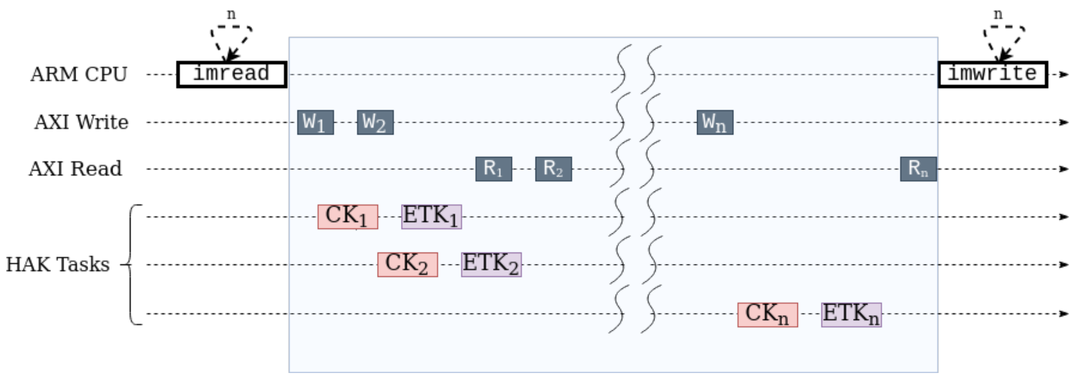


Figure 3.4: Workflow for CED.

3.4 Discussion on Vitis Acceleration Flow and the Realization of Vivado IP Flow in this Thesis

Vitis acceleration flow and Vivado IP generation flow have positive and negative aspects. Both of these flows have similar and distinctive features. Support for Dynamic Function Exchange (DFX), formerly partial reconfiguration, is available for both flows enabling them as cloud services. It is important to note that different hardware accelerator modules can be created to be compatible with static FPGA shell and configured interchangeably. In both flows, hardware accelerators can be developed with HDL or HLS. In the Vivado IP flow, communication with hardware accelerators can take place over a custom FPGA static shell and a custom message protocol. In Vitis acceleration flow, CPU and hardware accelerators exchange data with AXI Memory Mapped interface over a memory. Here, we note that in our Vivado IP flow, the FPGA static shell and hardware accelerator interface are created as AXI Stream. An accurate and efficient system design with the AXI Memory Mapped interface is usually more complex. AXI Memory Mapped consists of 5 stream channels. These channels are Write Response Channel, Write Data Channel, Write Address Channel, Read Address Channel and Read Response Channel. So it consumes more routing resources. In order to minimize the overhead of this interface, the communication should be performed in burst and the dataflow in hardware accelerators should be configured in a pipelined manner. In the AXI Stream interface, data can continuously be sent and received in burst form. Hardware accelerators with AXI Stream interface should be implemented with pipelined data flow to increase throughput. The application to be CPU accelerated developed in Vitis acceleration flow has OpenCL, XRT and Embedded Linux dependencies. On the other hand, our implementations in Vivado IP flow only expect the application to be accelerated on the user's side to provide data flow following the ACCLOUD messaging protocol, thanks to the Nova-Zynq and FPGA static shell infrastructure. The Vitis acceleration flow makes more sense to be preferred where the data to be processed is local to the CPU and hardware accelerator. On the other hand, our realization with custom shell, FPGAvisor and Vivado IP flow generated hardware accelerators; makes sense for the cases such as where the user streams data are processed in hardware accelerators, without any extra CPU intervention, located in the cloud-computing infrastructure.

CHAPTER 4

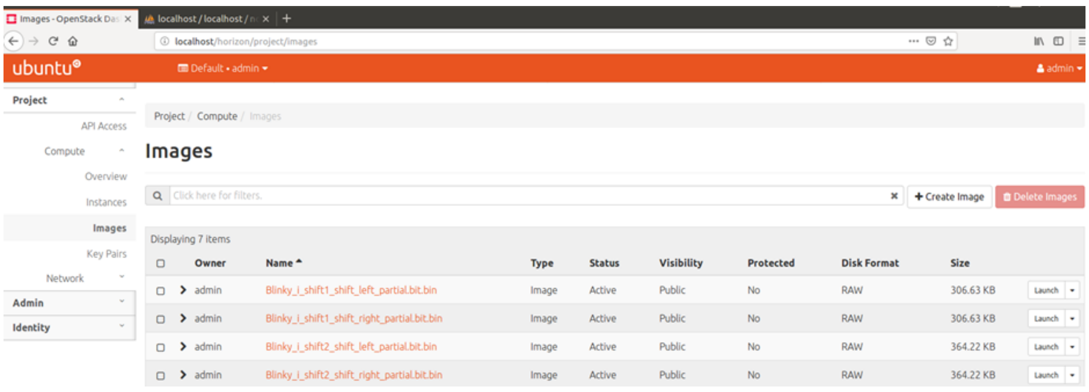
EVALUATION

OpenStack manages the cloud computing resources with Nova Project. Accordingly, the Nova Conductor on the controller node communicates with the Nova Compute instances on the compute nodes in the cloud over a RabbitMQ messaging interface. Nova-Zynq is the hardware-specific Nova-Compute that is developed in the scope of this thesis to run on the ARM processor of the Zynq SoC platform. In Section 4.1 we demonstrate the functional correctness of Nova-Zynq. In the remaining sections of this chapter, we provide performance evaluations of the accelerators developed and implemented within the scope of this thesis. To this end, we investigate metrics including reconfiguration speed, FPGA resource consumption, power, latency and throughput.

4.1 Nova-Zynq Functional Correctness

Nova-Zynq works as a daemon application in the background after GNU Linux is booted on the SoC board. It waits for messages from Nova Conductor to carry out specific instance launching tasks. OpenStack provides the Horizon Project that enables the cloud users to see an interface similar to Figure 4.1. Hardware accelerator image is selected by user sent to a Nova-Zynq application through OpenStack's RabbitMQ messaging tool, to be configured on an available FPGA reconfigurable region.

OpenStack graphical user interface then shows the launched instances as shown in Figure 4.2.

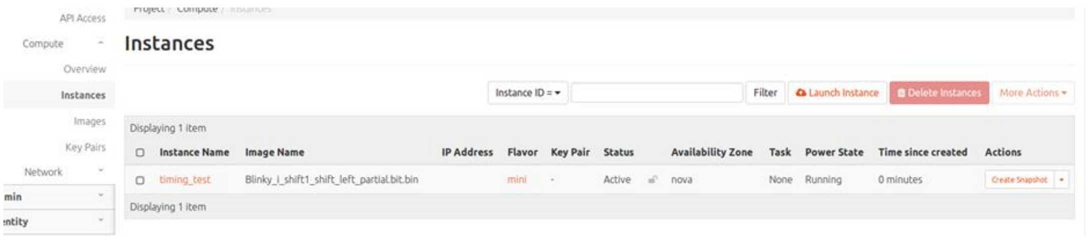


Images

Displaying 7 items

Owner	Name	Type	Status	Visibility	Protected	Disk Format	Size
admin	Blinky_i_shift1_shift_left_partial.bit.bin	Image	Active	Public	No	RAW	306.63 KB
admin	Blinky_i_shift1_shift_right_partial.bit.bin	Image	Active	Public	No	RAW	306.63 KB
admin	Blinky_i_shift2_shift_left_partial.bit.bin	Image	Active	Public	No	RAW	364.22 KB
admin	Blinky_i_shift2_shift_right_partial.bit.bin	Image	Active	Public	No	RAW	364.22 KB

Figure 4.1: Horizon GUI OpenStack Module



Instances

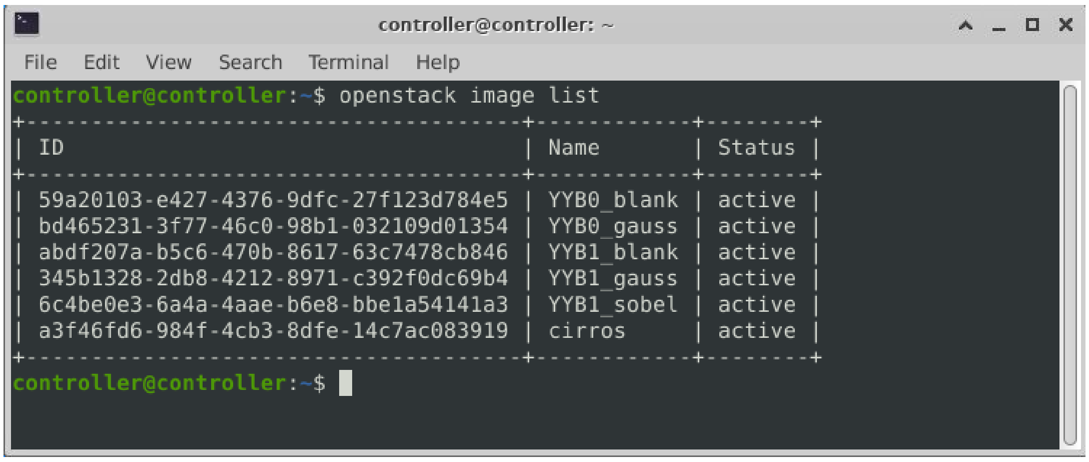
Displaying 1 item

Instance Name	Image Name	IP Address	Flavor	Key Pair	Status	Availability Zone	Task	Power State	Time since created	Actions
timing_test	Blinky_i_shift1_shift_left_partial.bit.bin		mini	-	Active	nova	None	Running	0 minutes	Create Snapshot

Displaying 1 item

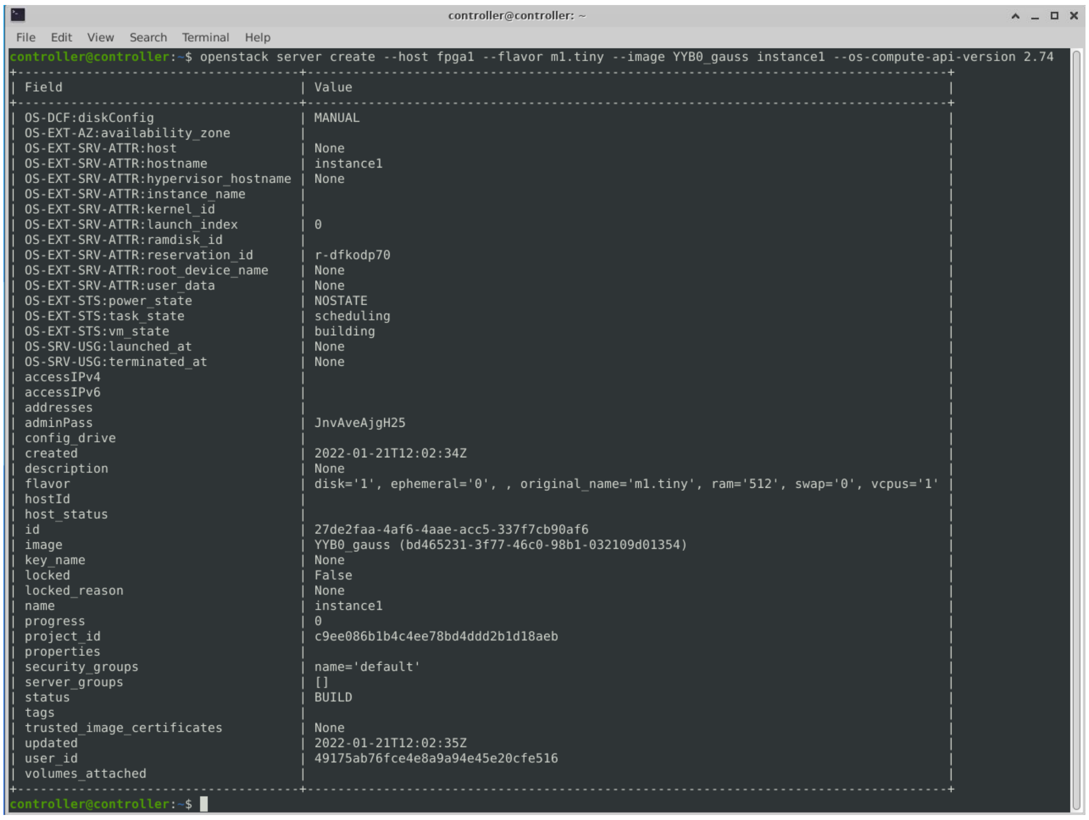
Figure 4.2: Horizon, Launched Instances View

In a similar manner, OpenStack client commands can be used to view FPGA hardware accelerator images and launch instances. Resource management of OpenStack compute nodes via the GNU Linux command line using the OpenStack Compute API [59] is shown in Figure 4.3 and Figure 4.4.



```
controller@controller:~$ openstack image list
+--------------------------------------+------------+--------+
| ID                                   | Name       | Status |
+--------------------------------------+------------+--------+
| 59a20103-e427-4376-9dfc-27f123d784e5 | YYB0_blank | active |
| bd465231-3f77-46c0-98b1-032109d01354 | YYB0_gauss | active |
| abdf207a-b5c6-470b-8617-63c7478cb846 | YYB1_blank | active |
| 345b1328-2db8-4212-8971-c392f0dc69b4 | YYB1_gauss | active |
| 6c4be0e3-6a4a-4aae-b6e8-bbe1a54141a3 | YYB1_sobel | active |
| a3f46fd6-984f-4cb3-8dfe-14c7ac083919 | cirros     | active |
+--------------------------------------+------------+--------+
controller@controller:~$
```

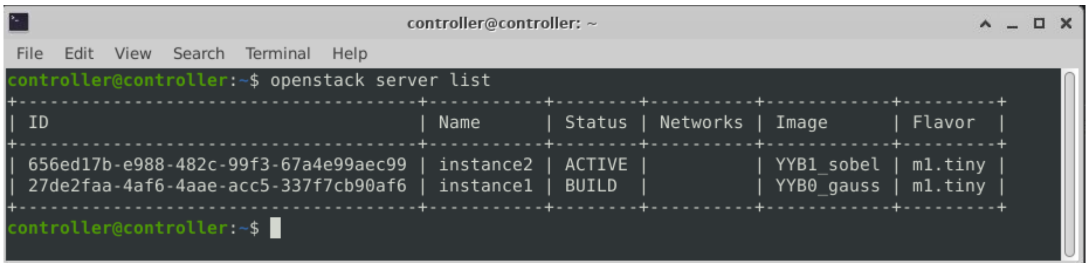
Figure 4.3: OpenStack CLI, Image List



```
controller@controller:~$ openstack server create --host fpga1 --flavor m1.tiny --image YYB0_gauss instance1 --os-compute-api-version 2.74
+-------------------------------------+------------------------------------------------------------------------------------+
| Field                               | Value                                                                              |
+-------------------------------------+------------------------------------------------------------------------------------+
| OS-DCF:diskConfig                   | MANUAL                                                                             |
| OS-EXT-AZ:availability_zone         |                                                                                    |
| OS-EXT-SRV-ATTR:host                | None                                                                               |
| OS-EXT-SRV-ATTR:hostname            | instance1                                                                          |
| OS-EXT-SRV-ATTR:hypervisor_hostname | None                                                                               |
| OS-EXT-SRV-ATTR:instance_name       |                                                                                    |
| OS-EXT-SRV-ATTR:kernel_id           |                                                                                    |
| OS-EXT-SRV-ATTR:launch_index        | 0                                                                                  |
| OS-EXT-SRV-ATTR:ramdisk_id          |                                                                                    |
| OS-EXT-SRV-ATTR:reservation_id      | r-dfkodp70                                                                         |
| OS-EXT-SRV-ATTR:root_device_name    | None                                                                               |
| OS-EXT-SRV-ATTR:user_data           | None                                                                               |
| OS-EXT-STS:power_state              | NOSTATE                                                                            |
| OS-EXT-STS:task_state               | scheduling                                                                         |
| OS-EXT-STS:vm_state                 | building                                                                           |
| OS-SRV-USG:launched_at              | None                                                                               |
| OS-SRV-USG:terminated_at            | None                                                                               |
| accessIPv4                          |                                                                                    |
| accessIPv6                          |                                                                                    |
| addresses                           |                                                                                    |
| adminPass                           | JnvAveAjgH25                                                                       |
| config_drive                        |                                                                                    |
| created                             | 2022-01-21T12:02:34Z                                                               |
| description                         | None                                                                               |
| flavor                              | disk='1', ephemeral='0', , original_name='m1.tiny', ram='512', swap='0', vcpus='1' |
| hostId                              |                                                                                    |
| host_status                         |                                                                                    |
| id                                  | 27de2faa-4af6-4aae-acc5-337f7cb90af6                                               |
| image                               | YYB0_gauss (bd465231-3f77-46c0-98b1-032109d01354)                                  |
| key_name                            | None                                                                               |
| locked                              | False                                                                              |
| locked_reason                       | None                                                                               |
| name                                | instance1                                                                          |
| progress                            | 0                                                                                  |
| project_id                          | c9ee086b1b4c4ee78bd4ddd2b1d18aeb                                                   |
| properties                          |                                                                                    |
| security_groups                     | name='default'                                                                     |
| server_groups                       | []                                                                                 |
| status                              | BUILD                                                                              |
| tags                                |                                                                                    |
| trusted_image_certificates          | None                                                                               |
| updated                             | 2022-01-21T12:02:35Z                                                               |
| user_id                             | 49175ab76fce4e8a9a94e45e20cfe516                                                   |
| volumes_attached                    |                                                                                    |
+-------------------------------------+------------------------------------------------------------------------------------+
controller@controller:~$
```

Figure 4.4: OpenStack CLI, Launching an Accelerator Instance

Launched accelerator instances are shown in Figure 4.5.



```
controller@controller:~$ openstack server list
+--------------------------------------+-----------+--------+----------+------------+---------+
| ID                                   | Name      | Status | Networks | Image      | Flavor  |
+--------------------------------------+-----------+--------+----------+------------+---------+
| 656ed17b-e988-482c-99f3-67a4e99aec99 | instance2 | ACTIVE |          | YYB1_sobel | m1.tiny |
| 27de2faa-4af6-4aae-acc5-337f7cb90af6 | instance1 | BUILD  |          | YYB0_gauss | m1.tiny |
+--------------------------------------+-----------+--------+----------+------------+---------+
controller@controller:~$
```

Figure 4.5: OpenStack CLI, Display Lauched Instances

Nova-Zynq receives the message for launching the hardware accelerator instance with the image address. It downloads the image from the OpenStack Glance module and dynamically exchanges the hardware accelerator function in an RR within the FPGA. Here, we emphasize that the partial reconfiguration is now called Dynamic Function Exchange by Xilinx and this capability allows for exchanging an existing accelerator with another one. The flow is demonstrated on a video [60]. Nova-Zynq output logs on Embedded GNU Linux console for debugging purposes are shown in Figure 4.6. In the log, it states the received image and configuration to RR.

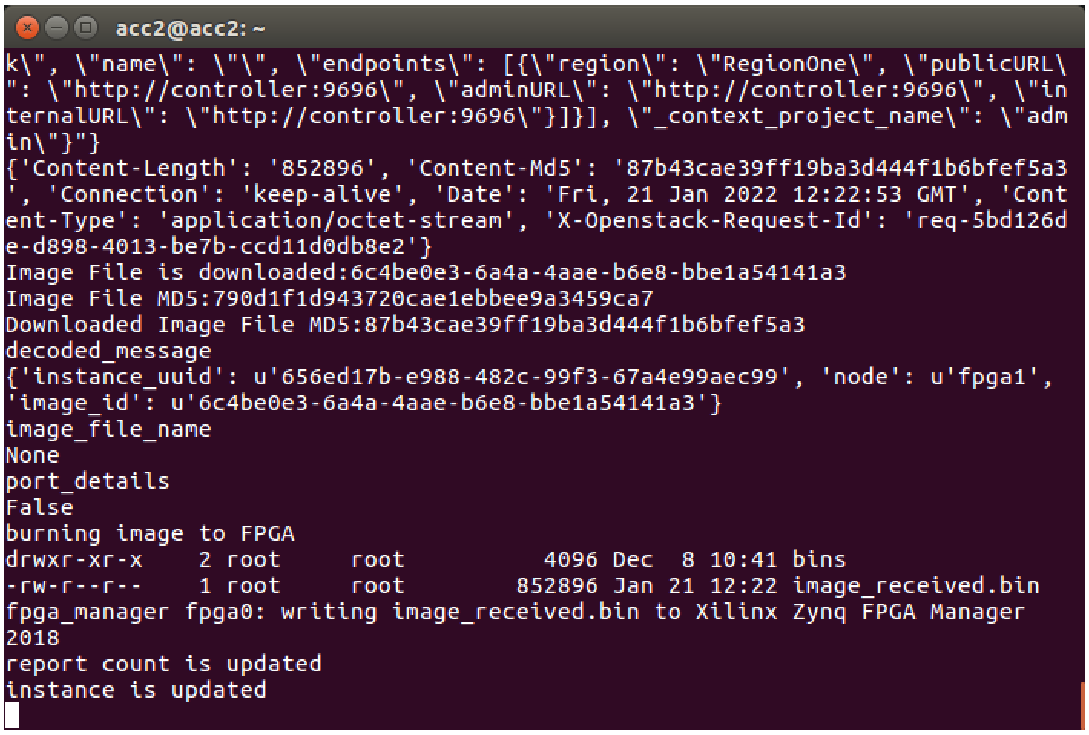

```
acc2@acc2: ~
k\", \"name\": \"\", \"endpoints\": [{\"region\": \"RegionOne\", \"publicURL\
": \"http://controller:9696\", \"adminURL\": \"http://controller:9696\", \"in
ternalURL\": \"http://controller:9696\"}]}], \"_context_project_name\": \"adm
in\"}"}
{'Content-Length': '852896', 'Content-Md5': '87b43cae39ff19ba3d444f1b6bfef5a3
', 'Connection': 'keep-alive', 'Date': 'Fri, 21 Jan 2022 12:22:53 GMT', 'Cont
ent-Type': 'application/octet-stream', 'X-Openstack-Request-Id': 'req-5bd126d
e-d898-4013-be7b-ccd11d0db8e2'}
Image File is downloaded:6c4be0e3-6a4a-4aae-b6e8-bbe1a54141a3
Image File MD5:790d1f1d943720cae1ebbee9a3459ca7
Downloaded Image File MD5:87b43cae39ff19ba3d444f1b6bfef5a3
decoded_message
{'instance_uuid': u'656ed17b-e988-482c-99f3-67a4e99aec99', 'node': u'fpga1',
'image_id': u'6c4be0e3-6a4a-4aae-b6e8-bbe1a54141a3'}
image_file_name
None
port_details
False
burning image to FPGA
drwxr-xr-x    2 root     root          4096 Dec  8 10:41 bins
-rw-r--r--    1 root     root        852896 Jan 21 12:22 image_received.bin
fpga_manager fpga0: writing image_received.bin to Xilinx Zynq FPGA Manager
2018
report count is updated
instance is updated
```

Figure 4.6: Nova-Zynq Debug Logs

In order to observe the relationship between the reconfiguration time and the bitstream size, experiments were conducted with different-sized RRs with different-sized bitstreams. In the experiments, the configuration or reconfiguration speed was calculated from the time measurements taken on the Embedded Linux with "time" Linux command. Regardless of the size of the RR and the region in which it is placed in the FPGA, the speed values of the RR configuration were measured as 75 ± 5 Mbps over the PCAP interface on-chip. This shows the dominance of hardware abstraction overhead with Kernel driver and OS among FPGA configuration speed parameters. According to Zynq TRM [61], maximum transfer rate through the PCAP for bitstream configuration with ARM is approximately 1160 Mbps.

The size of an RMs, hardware accelerator FPGA images depends on the size and shape of the RRs. Therefore, all the accelerators that is implemented for RR0 have an uncompressed size of 845568 bytes. All the accelerators that is implemented for

RR1 have an uncompressed size of 577312 bytes. These results are obtained from implementations of Gaussian and Sobel Filters as partial bitstream binaries. The full bitstream has 13321408 bytes in size. Calculations of reconfiguration durations at full and measured bandwidths are given in Table 4.1. Here we note that the average VM boot time representing the hardware accelerator instantiations without actually writing the bitstream image was measured more than 3 seconds in [5]. Hence, these durations are very small compared to the entire OpenStack control overhead.

Table 4.1: FPGA Image Reconfiguration Duration

		Configuration Time(ms)	
Region	**Size(Bytes)**	**@75Mbps**	**@1160 Mbps**
RR0	845568	90.19	5.83
RR1	577312	61.58	3.98
Full FPGA	13321408	1420.95	91.87

4.2 Canny Edge Detector Performance Evaluation

As seen in Table 4.2, the resource utilization of CED was found to be low. The rendering processes 512x512 color JPEG images and operates at 142 MHz. Transfer times between the global memory areas were measured to be 1,845 ms from the processor to the FPGA side and 1.934 ms from the FPGA to the processor side. The throughput is measured as 118.480 frames/sec. This throughput can be improved with the better multi-threaded and pipe-lined MC architecture. In the results we obtained from Vivado's power analysis tool, it was observed that the system chip with a total consumption of 2,162 W, the majority of the power consumption was made by the ARM processor on the FPGA with 1.576 W. Latency here is the time elapsed within kernel between the first input pixel and the last output pixel in a frame.

Table 4.2: Canny Hardware Accelerator Kernel Realization

HAK	LUT	BRAM	DSP	Power	Latency
Canny	6,800 (3.11 %)	9 (1.65 %)	10 (1.11 %)	0.060 W	4.647 ms
Edge Tracing	5,655 (2.59 %)	14 (2.57 %)	9 (1.00 %)	0.054 W	2.578 ms

4.3 Gaussian and Sobel Filter Evaluation

The designed Gaussian Filter HLS performance prediction results are given in Table 4.3 According to HLS reports, the target frequency for the Gaussian Filter hardware accelerator was 156.25 MHz clock. This frequency was selected compatible to the 40 Gbps Ethernet IP Core of the ACCLOUD platform. We reached that frequency with 0.133 ms latency. BRAM, DSP, LUT and Flip Flop resource usage predictions are noted on the table. Latency here is the time elapsed between the first input pixel and the last output pixel in a frame.

Table 4.3: Gaussian Hardware Accelerator Realization

LUT	BRAM	DSP	FF	Power	Latency
13026(3.43 %)	7(0.64 %)	35(3.89 %)	9314(1.40 %)	0.098 W	0.133 ms

Sobel Filter HLS performance prediction results are given in Table 4.4 According to HLS reports, Sobel Filter hardware accelerator runs at 156.25 MHz clock and the latency is 113.00 ms. BRAM, DSP, LUT, and Flip Flop resource usage predictions are noted on the table.

Table 4.4: Sobel Hardware Accelerator Realization

LUT	BRAM	DSP	FF	Power	Latency
6819 (3.12 %)	3 (0.28 %)	0 (0.00 %)	3990 (0.91 %)	0.043 W	0.130 ms

4.4 Ethernet Frame Processor Evaluation

Table 4.5 shows the synthesis result of the Ethernet frame processor hardware accelerator synthesis report. Ethernet frames are fed with 156.25 MHz clock. According to HLS reports, the frame processor hardware accelerator IP has a latency performance of 12.80 ns. This accelerator accepts and processes Ethernet frame data in 32-byte flits. BRAM, DSP, LUT and Flip Flop resource usage projections are noted.

Table 4.5: Ethernet frame Processor Hardware Accelerator Realization

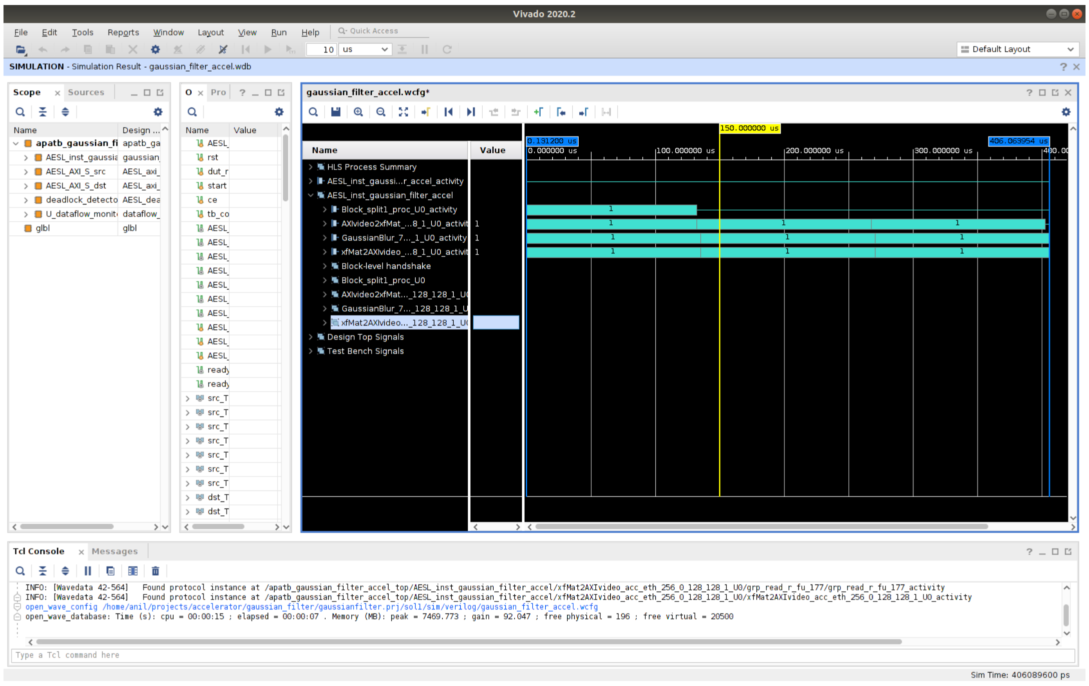
LUT	BRAM	DSP	FF	Power	Latency
667 (0.31 %)	3 (0.28 %)	0 (0.00 %)	841 (19 %)	0.029 W	0.013 ms

4.5 Gaussian and Sobel Filters Co-Simulation

Vitis HLS compiler compiles C/C++ HLS hardware accelerator modules to produce the HDL codes of Gaussian and Sobel hardware accelerators. We wrote a testbench in C/C++ and used it in Vitis HLS co-simulation. With this testbench, the inputs of Gaussian and Sobel Filter functions are created and their outputs are checked with the test code in testbench. During co-simulation, HDL codes are simulated in the background with the Vivado simulation tool.

Figure 4.7 shows the waveform of events when the 128x128 pixels (bytes) image is fed to Gaussian Filter hardware accelerator three times within a for loop (i.e. Listing 5 executed 3 times) from the hardware kernel point of view.

The data obtained as a result of the Gaussian Filter simulation are as follows.



```
INFO: [Wavedata 42-564]   Found protocol instance at /apatb_gaussian_filter_accel_top/AESL_inst_gaussian_filter_accel/xfMat2AXIvideo_acc_eth_256_0_128_128_1_U0/grp_read_r_fu_177/grp_read_r_fu_177_activity
INFO: [Wavedata 42-564]   Found protocol instance at /apatb_gaussian_filter_accel_top/AESL_inst_gaussian_filter_accel/xfMat2AXIvideo_acc_eth_256_0_128_128_1_U0/xfMat2AXIvideo_acc_eth_256_0_128_128_1_U0_activity
open_wave_config /home/anil/projects/accelerator/gaussian_filter/gaussianfilter.prj/soll/sim/verilog/gaussian_filter_accel.wcfg
open_wave_database: Time (s): cpu = 00:00:15 ; elapsed = 00:00:07 . Memory (MB): peak = 7469.773 ; gain = 92.047 ; free physical = 196 ; free virtual = 20500
```

Sim Time: 406089600 ps

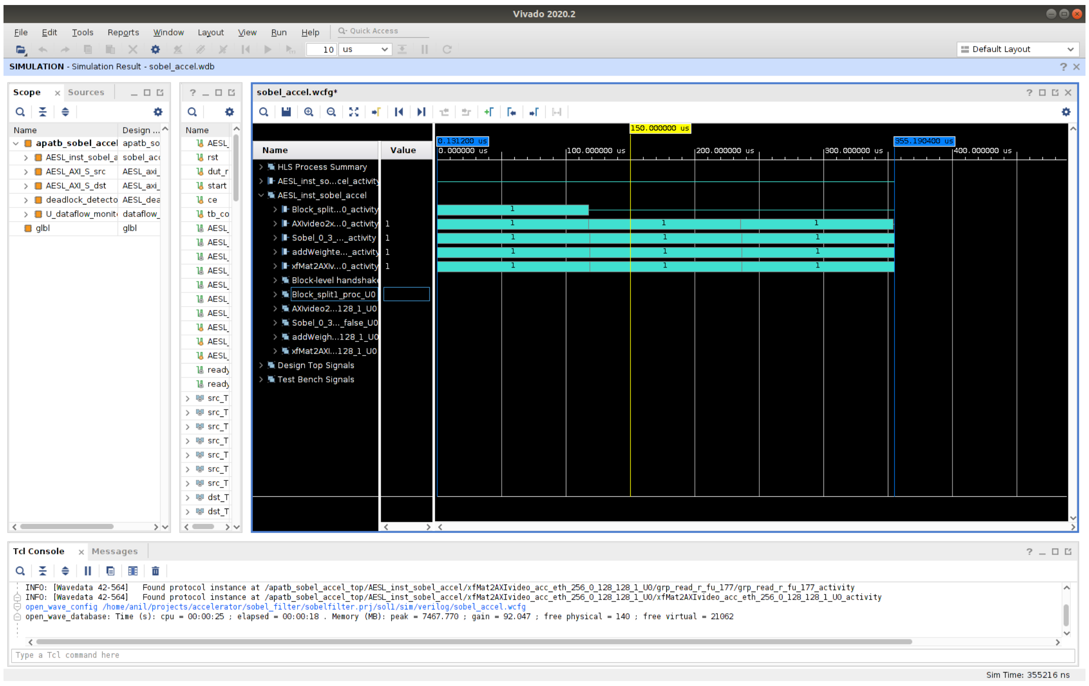
Figure 4.7: 128x128.png Gaussian Filter

```
//turn image frame into AXIS
Mat2Axivideo_acc_eth<XF_NPPC1, _W>(in_img, _src_gauss);
// Call the kernel function
gaussian_filter_accel(_src_gauss, _dst_gauss);
//turn AXIS back to image frame
AXIvideo2cvMatxf_acc_eth<XF_NPPC1, _W >(_dst_gauss, ou_img_gauss_0);
```

Listing 5: C/C++ Testbench Code Snippet for Gaussian Filter

Clock: 156.25 MHz ([HLS 200-789] **** Estimated Fmax: 194.16 MHz) Starting Marker : 0.131200 us Finishing Marker: 406.046000 us Avg Latency: 42285 Max Latency:63425 Min Latency:21145(from Vitis HLS Cosim Report) Throughput calculation@156.25Mhz: 128x128x8x3= 393216 bits three frames 406.046000 us - 0.131200 us = 405.9148 us = 0.0004059148 sec 393216 bits / 0.0004059148 sec = 968715602.387 bps = <u>968.715602387 Mbps</u>

A similar simulation was also applied for the Sobel Filter. The Vivado simulator waveform, which is formed when the Sobel Filter is run 3 times with a 128x128 frame, is as in Figure 4.8.



```
INFO: [Wavedata 42-564]   Found protocol instance at /apatb_sobel_accel_top/AESL_inst_sobel_accel/xfMat2AXIvideo_acc_eth_256_0_128_128_1_U0/grp_read_r_fu_177/grp_read_r_fu_177_activity
INFO: [Wavedata 42-564]   Found protocol instance at /apatb_sobel_accel_top/AESL_inst_sobel_accel/xfMat2AXIvideo_acc_eth_256_0_128_128_1_U0/xfMat2AXIvideo_acc_eth_256_0_128_128_1_U0_activity
open_wave_config /home/anil/projects/accelerator/sobel_filter/sobelfilter.prj/sol1/sim/verilog/sobel_accel.wcfg
open_wave_database: Time (s): cpu = 00:00:25 ; elapsed = 00:00:18 . Memory (MB): peak = 7467.770 ; gain = 92.047 ; free physical = 140 ; free virtual = 21062
```

Sim Time: 355216 ns

Figure 4.8: 128x128 frame Sobel Filter

The data obtained as a result of the Sobel Filter simulation are as follows.

Clock: 156.25 MHz

([HLS 200-789] **** Estimated Fmax: 194.16 MHz)

Starting Marker : 0.131200 us

Finishing Marker: 406.046000 us

Avg Latency: 36997 Max Latency:55476 Min Latency:18518(from Vitis HLS Cosim Report)

Throughput calculation@156.25Mhz:

128x128x8x3= 393216 bits three frame 355.190400 us - 0.131200 us = 355.0592 us = 0.0003550592 sec

393216 bits / 0.0003550592 sec = 1107466022.57 bps = 1107.46602257 Mbps

4.6 Reserving RR Resources on FPGA Floor-planning for Hardware Accelerators

In Figure 4.9, the placement of the existing FPGA resources on the Vivado GUI screen and their clock zones is shown. A structure showing the separated areas appears. These areas were created by drawing a rectangular area with the computer mouse interactively on the Vivado floorplanning window. How much and what kind of resources are encapsulated automatically during drawing is indicated by the tool. The parts reserved for two RR areas covers suitable number of RAM, DSP, BRAM, LUT resources to host the designed hardware accelerators in this thesis. The optimal construction of RR areas, regarding their resource amounts and homogeneity is not in the scope of this thesis. An optimization can be made by collecting the statistical data of the requests coming to a cloud computing management system and the desired hardware accelerator types.

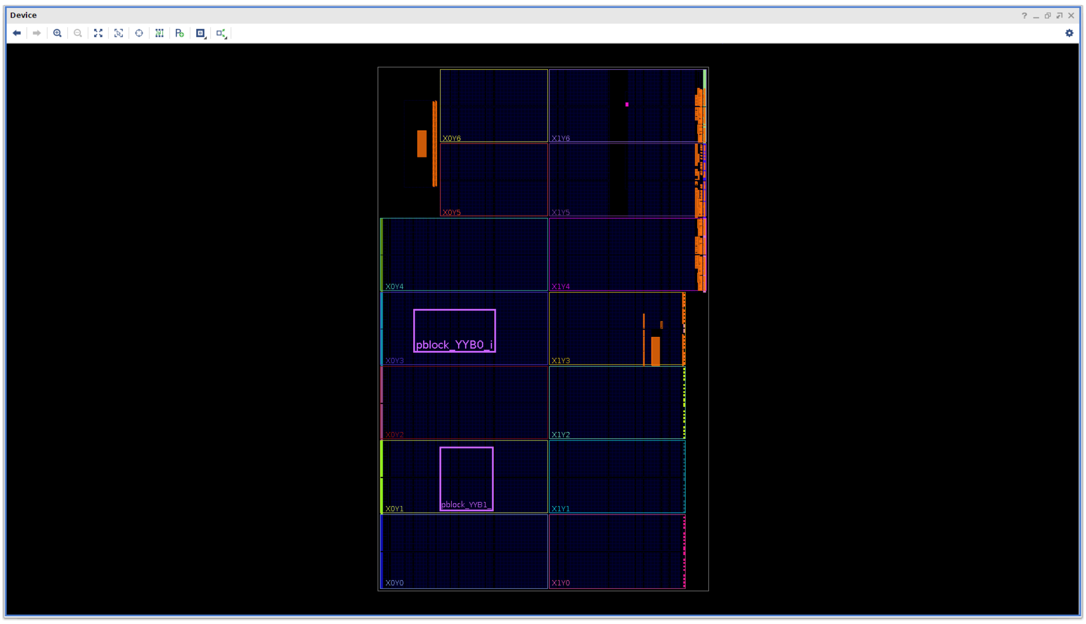


Figure 4.9: Vivado Floor-planning View, Reconfigurable Regions

The available FPGA resources in the areas reserved for the RRs are shown in Table 4.6.

Table 4.6: Available Resources in Reconfigurable Regions

	Available	
Site Type	**RR0**	**RR1**
Slice LUT	8800	5200
LUT as Logic	8800	5200
LUT as Memory	3600	2000
Slice Registers	17600	10400
Register as Flip Flop	17600	10400
Register as Latch	17600	10400
F7 Muxes	4400	2600
F8 Muxes	2200	1300
Block RAM Tile	30	10
RAMB36/FIFO	30	10
RAMB18	60	20
DSPs	60	40

4.7 Generating Hardware Accelerator Modules

Vitis HLS can export hardware accelerators as Vivado IP or as HDL codes. In Vivado, HDLs or IPs can be synthesized out of context, that is, independently of other FPGA parts. In OOC synthesis, I/O buffers, global clocks and other chip-level resources are not inserted. OOC designs can be saved as a design checkpoint(DCP) file for design reuse. DCPs archive synthesized modules, preserving the logical design and constraints, and can be restored as a part of another complete FPGA design for later reuse. Synthesized hardware accelerators are packaged as DCPs in Vivado with the commands in Listing 6.

```
synth_design -top <hw_accelerator_top> -part <fpga_part_number> -mode
↪  out_of_context
write_checkpoint -cell <RRx> <RMx_synth.dcp>
```

Listing 6: Synthesize and Save as DCP

DCPs are tool version dependent files. If design reuse is to be made between different versions, the synthesized hardware accelerator design should be saved as a netlist. In Vivado this can be done with the "write_edif <netlist_file>" TCL command. Netlist is a definition according to the FPGA architecture, as it only defines the connections in the design and the logical resource components used, it is independent of the tool version. Then, the netlists are converted back to DCP files with the commands as in Listing 7 in the tool version where the full FPGA design will be realized.

```
read_edif <netlist_file>
link_design -mode out_of_context -top <hw_accelerator_top> -part
↪  <fpga_part_number>
write_checkpoint <RMx_synth.dcp>
```

Listing 7: Generate DCP from Netlist

4.8 Assigning Hardware Accelerator Modules to Reserved RR Resources

Figure 4.10 shows the hardware accelerator modules assigned to the two RRs. Hardware accelerator modules are added to the project as Xilinx Vivado-specific DCP (design checkpoint) format. Hardware accelerator modules Gaussian Filter and Sobel Filter can be placed to the desired RR area in the fully routed and placed static FPGA design with the following TCL command in Listing 8.

```
read_checkpoint -cell <RRx> <RMx_synth.dcp>
```

Listing 8: Assigning RM to RR

Figure 4.11 shows logic resources after completing placement and implementation stages on the FPGA.

The resources consumed by the hardware accelerators placed in the RR areas in Table 4.7. Behaviorally identical Gaussian Filter IP designs were placed in both RRs.

The resources consumed by the hardware accelerators are placed in the RR areas in Table 4.8. In this case, Gaussian Filter design placed in both RR0 and Sobel Filter design placed in RR1.

Table 4.7: Resource Utilizations on Reconfigurable Regions

	RR0			**RR1**		
Site Type	**Available**	**Used**	**%Util**	**Available**	**Used**	**%Util**
Slice LUT	8800	1809	20.56	5200	2849	54.79
LUT as Logic	8800	1720	19.55	5200	2760	53.08
LUT as Memory	3600	89	2.47	2000	89	4.45
Slice Registers	17600	2684	15.25	10400	3740	35.96
Register as Flip Flop	17600	2684	15.25	10400	3740	35.96
Register as Latch	17600	0	0.00	10400	0	0.00
F7 Muxes	4400	48	1.09	2600	48	1.85
F8 Muxes	2200	0	0.00	1300	0	0.00
Slice	2200	854	38.82	1300	1169	89.92
SLICEL	1300	498	38.31	800	712	89.00
SLICEM	900	356	39.56	500	457	91.40
Unique Control Sets	2200	52	2.36	1300	84	6.46
Block RAM Tile	30	3.50	11.67	10	3.50	35.00
RAMB36/FIFO	30	0	0.00	10	0	0.00
RAMB18	60	7	11.67	20	7	35.00
DSPs	60	20	33.33	40	20	50.00

Table 4.8: Available Resources in Reconfigurable Regions

	RR0			RR1		
Site Type	**Site Type**	**Used**	**%Util**	**Available**	**Used**	**%Util**
Slice LUT	8800	1802	20.48	5200	1639	31.52
LUT as Logic	8800	1713	19.47	5200	1639	31.52
LUT as Memory	3600	89	2.47	2000	0	0.00
Slice Registers	17600	2684	15.25	10400	1693	16.28
Register as Flip Flop	17600	2684	15.25	10400	1693	16.28
Register as Latch	17600	0	0.00	10400	0	0.00
F7 Muxes	4400	48	1.09	2600	14	0.54
F8 Muxes	2200	0	0.00	1300	0	0.00
Slice	2200	910	41.36	1300	666	51.23
SLICEL	1300	547	42.08	800	399	49.88
SLICEM	900	363	40.33	500	267	53.40
Unique Control Sets	2200	52	2.36	1300	73	5.62
Block RAM Tile	30	3.50	11.67	10	1.50	15.00
RAMB36/FIFO	30	0	0.00	10	0	0.00
RAMB18	60	7	11.67	20	3	15.00
DSPs	60	20	33.33	40	0	0.00

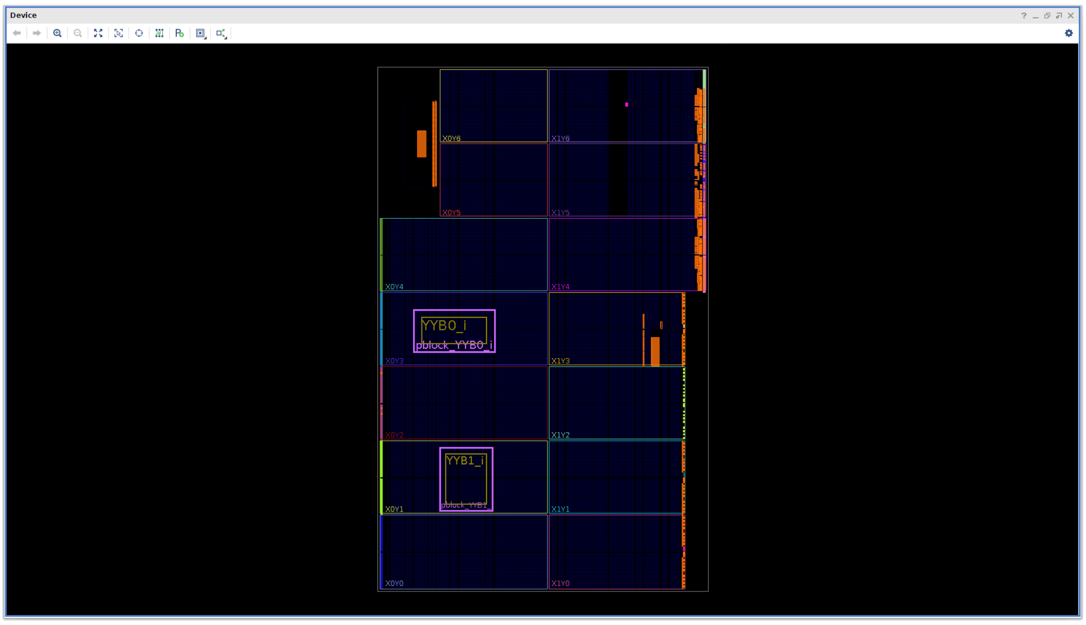


Figure 4.10: Vivado Floor Planning View Reconfigurable Modules Reside in Reconfigurable Regions

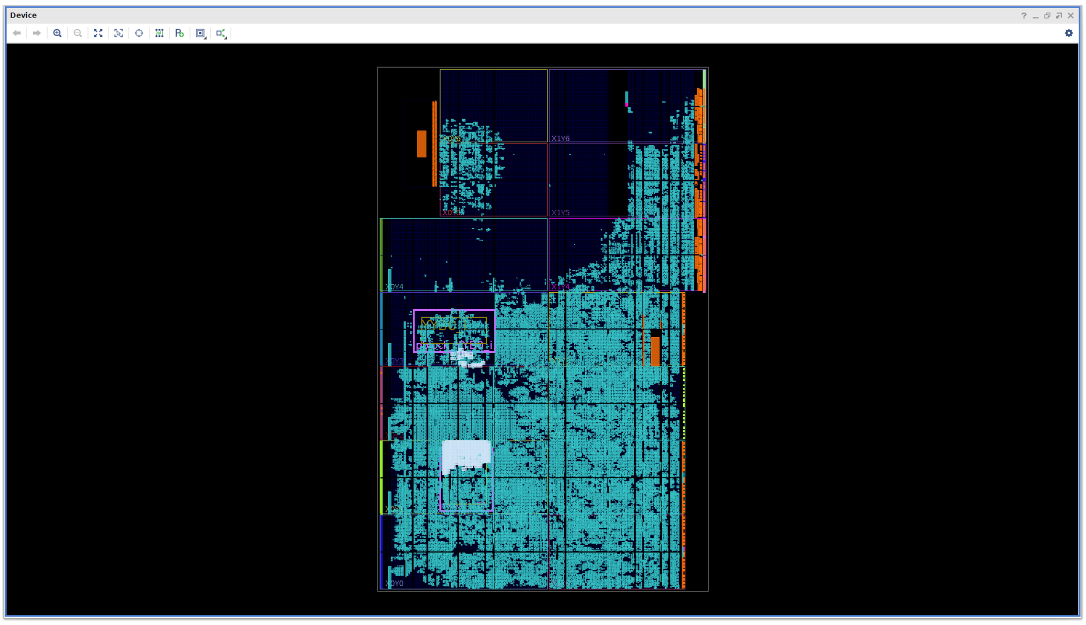


Figure 4.11: Vivado Floor Planning View of Implemented Full Design

4.9 Reusability of Reconfigurable Region Resources

It can be seen with these two implementation examples that sufficient resources are allocated for Gaussian and Sobel Filters. To this end, the size of the RR field will be sufficient for similar hardware accelerators such as image processing applications in Xilinx Vitis Vision libraries. When we look at the resource consumption of Vitis Vision libraries, it is seen that the resources allocated for RRs mostly be sufficient for the use of other functions in this area. In order to be able to develop partial RRs with different hardware accelerator modules, FPGA resources and connections outside the RRs are fixed and the static FPGA design is preserved. FPGA binaries can be created in a similar way by replacing the RRs with different hardware accelerator DCPs without touching the static FPGA areas. In this way, more than one hardware accelerator application in RR0 and RR1 can be configured in place of each other while the FPGA is operating. As an example, the RR1 is constructed with this method in such a way that it can be configured with both Gaussian and Sobel Filter binaries instead of each other, while the FPGA is running. Any user who has a static FPGA design DCP can also create a configuration binary for a RR without needing the source codes of the static FPGA project or sharing the source code of their hardware accelerator with the FPGA service provider. After hardware accelerator design is ready as HDL or HLS, the flow a user should follow is described in Section 4.7 and Section 4.8.

4.10 Gaussian-Sobel Hardware Accelerators Functional Correctness

Figure 4.12 shows the setup that we use to test the image processing pipeline, which consists of Gaussian and Sobel image filters. RM0 implements Gaussian image filter, RM1 implements Sobel image filter. Here note that these to RMs exchange Ethernet frames through the FPGA Switch shown in Figure 2.2 that resides in the FPGA Shell. The video data stream is started by the application created with C++ code called "video_stream_app". "video_stream_app" sends Ethernet frames to RM0's MAC address with a custom Ethernet protocol. The wrapper in RM0 extracts the video frame out of the Ethernet frame payload. Then the wrapper forwards the video frame to the Gaussian Filter IP and retrieves the processed video frame back. Subsequently, it regenerates the Ethernet frame with the processed video frame in the payload and sends it to the FPGA switch with a destination address of AXI Stream ID address of RM1. RM1 works similarly to RM0. The wrapper in RM1 receives the Ethernet frame from the FPGA switch. It extracts the video frame from the payload, passes it to the Sobel Filter and retrieves it. The fully processed video frame is packed in an Ethernet Frame and sent to the FPGA switch with a destination address of the "Workstation PC" MAC address. The FPGA switch forwards this Ethernet frame to the "Workstation PC". The video data stream captured by the "Workstation PC" is displayed on the monitor via the "video_display_app" application written in C++. These applications are written in the context of this thesis to conduct a demonstration of the setup and hardware accelerators.

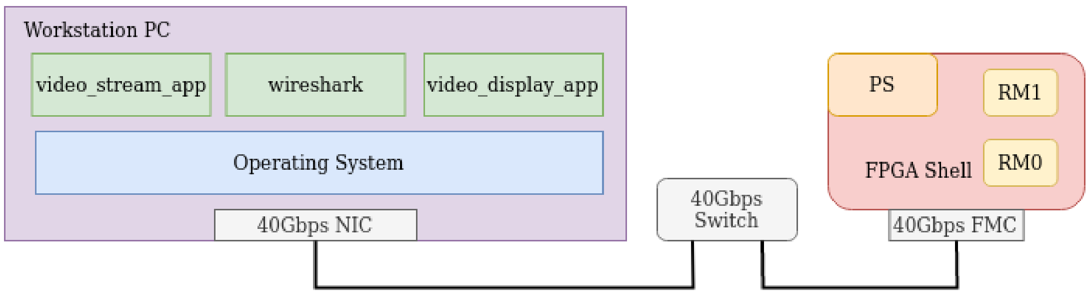


Figure 4.12: Gaussian and Sobel Filter Image Processing Pipeline in FPGA

The steps of the "video_stream_app" application operation are as follows: Using the OpenCV library [62], it reads a video file on the "Workstation PC" frame by frame. Each frame read is converted into a single channel image frame in grayscale. It is scaled to the sizes supported by the filters in the FPGA. Each pixel in the frame is written to the Ethernet frame payload area in a loop. After the Ethernet frame reaches a certain length, it is sent via the "40Gbps NIC" as a raw Ethernet frame with the API provided by the GNU Linux socket library [63].

The steps of the "video_display_app" application, which is used to visually play the processed video frames on the monitor of the "Workstation PC", are as follows: It sniffs the "40Gbps NIC" Ethernet traffic using socket library API. It creates a frame by filtering the Ethernet frames according to the Ethernet MAC addresses, reading the pixels of the video frames in the Ethernet frames in a loop, filling the pixel areas of a frame one by one. After the reading process of an Ethernet frame and the reading of the pixels in this frame are completed, it sniffs the Ethernet traffic again and waits for the next packet. After a frame is created with all its pixels, it displays this frame with the help of OpenCV library functions. These steps are repeated for each frame.

The path followed by the video stream data in the "FPGA shell" is as follows: Ethernet frames are received with "40G ETH CORE", which is a 3rd party IP. "Dest. Gen." With this, the destination MAC address of the Ethernet frame is parsed and the AXI Stream destination ID in the FPGA is determined from this address. Video data entering from the first port of "FPGA Switch" is switched to the fifth port where RR0 is located. The video data sent from RR0 back to "FPGA Switch" is then switched to the sixth port where RR1 is located. The video data processed in RR1 is sent back to the first port of the "FPGA Switch" and thus exits the FPGA. There are also Xilinx provided PR decoupler IPs between "FPGA Switch" and RRs. The end-to-end path is as shown in Figure 4.13.

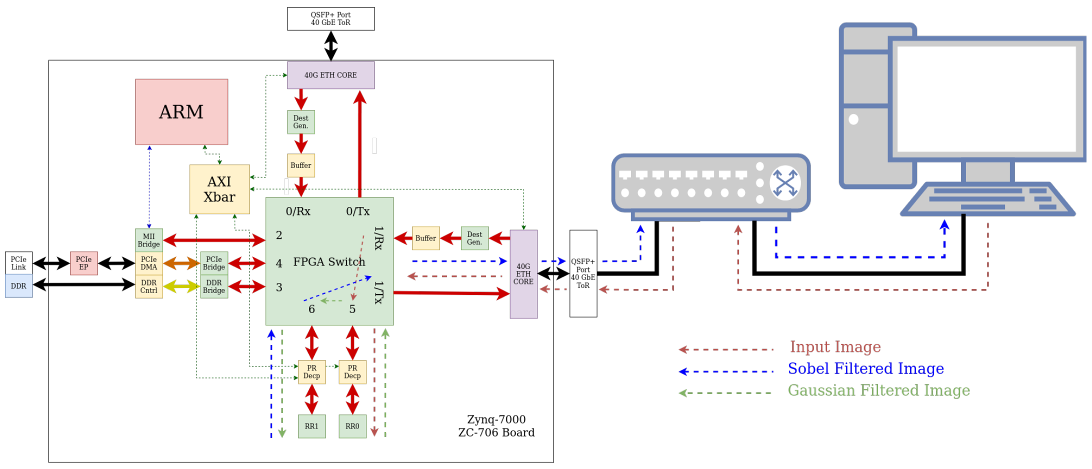


Figure 4.13: End-to-End Image Processing Pipeline Path

The measurements shown in Table 4.9 and Table 4.10 were calculated from Ethernet frames captured with Wireshark [64]. Each experiment is repeated 5 times. According to "Workstation PC" in Wireshark, the statistics of outgoing packets are shown with "Send" columns and the statistics of incoming packets with "Receive" columns. The Wireshark application is set to show the time differences of the packets with respect to each other in the "Time" column with nanosecond resolution. Each video frame fits into a certain number of Ethernet frames. An Ethernet frame does not contain pixels of two consecutive frames. The time elapsed between reading two consecutive frames with the OpenCV function was not taken into account in the measurements. Since Ethernet frames are created with the "video_stream_app" application, the speed of sending and receiving the packets is dependent on the packet creation rate of this application. "video_display_app" application does not affect metrics.

Ethernet frames have the structure shown in Figure 4.14.

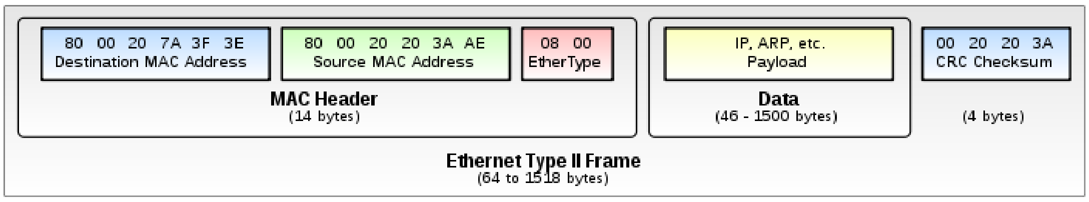


Figure 4.14: Ethernet Type II Frame Structure

In addition to these fields, there are 7-octet preambles and 1 octet SFD (Start of Frame Delimiter) data before each frame. There is also a minimum of 12-octet IFG (Inter Frame Gap) space between any two consecutive frames. The AXI Stream Data bus structure used in the FPGA consists of 256 bit flits. Therefore, each Ethernet frame is transmitted in 32-byte chunks within the FPGA. In order to simplify the frame processing structure in hardware accelerators, the number of pixels in a frame is equally distributed to each Ethernet frame. Apart from that, the 18-byte Data field that comes after the 14-byte Ethernet header field and creates a single flit in the FPGA is padded with ’0x00’. In this case, the efficiency calculation is done as follows.

$$\eta = \frac{Pixel\_Count\_in\_an\_Ethernet\_Frame}{Total\_Ethernet\_Frame\_Length + Preambles + IFG + SFD}$$

For the Ethernet frames of 1056, 544, 288, 160, 96, 64 bytes, the efficiency is calculated as 0.95, 0.91, 0.83, 0.71, 0.55, 0.38 respectively. The frame sizes are selected to have equal number of pixels per frame. As an example 1056 Byte Ethernet Frame has a payload of 1024 Bytes (pixels) and the 128x128=16384 Byte frame is equally divided into 16 such frames. As we show in Table 4.9, smaller frames have smaller latency with smaller efficiency and data rate.

The timestamp values shown in Wireshark are retrieved using the libpcap library [65]. The libcap library gets the time information from the networking stack in the OS. In this case, the accuracy of the measurements taken is dependent on the OS and hardware specifications. Ubuntu 20.04 LTS, Intel(R) Xeon(R) Bronze 3104 CPU @ 1.70GHz processor and Mellanox MCX313A-BCCT NIC were used in the measurements. For a more detailed explanation, see the reference [66].

264 frames of video with a resolution of 128x128 pixels were used in the measurements. The send rate and receive rate are calculated separately from the time difference of the first and last incoming and outgoing Ethernet Frames. The Send_and_Receive rate is calculated from the time difference of the first outgoing and last incoming packets. For latency measurements, the time difference between the last outgoing Ethernet frame of a video frame and the last incoming Ethernet frame of the same video frame is used.

"video_stream_app" and "video_display_app" applications were implemented with Python to shorten application development time before C++. These measurements are also included in Table 4.10 to show the dependence of the measurements on the performance of the video streaming application.

In order to test the performance of the image processing pipeline in the RRs on the FPGA at the maximum possible throughput, the Ethernet frame stream application running on a single core and having a low throughput value was not sufficient. Instead, we attempted to use open source applications that can create raw Ethernet frame traffic. In this context, two applications were examined. PackETH [67] is a very popular Ethernet frame generator tool. It can be used to generate packages easily with its GUI. Packets can be produced with the desired bandwidth and delay. But this tool did not work in a multi-threaded way. As a second alternative, the trafgen [68] tool was tried. By using all processor cores, trafgen can produce packages at much higher speeds. Many settings such as the sending speed and delay setting of the packets can also be made with this tool. It reads the settings and the package to be sent from a unique trafgen language configuration file created by user. It can only be used via the GNU Linux command line. In the Figure 4.15, it is shown running trafgen tool. It is configured to output 1056 bytes number of 15206400

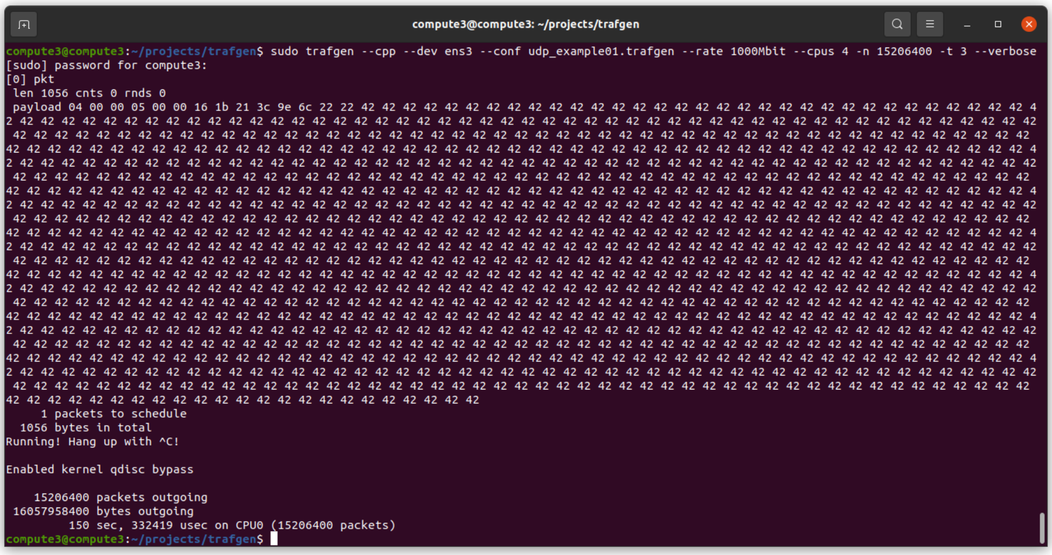
Table 4.9: 264 frames 128x128 Video Measurement CPP

		Frame Rate(fps)			Latency(us)
		Send	Receive	Send_and_Receive	
Measurement 1056 byte Ethernet frames	1	4344.54	6163.94	3768.02	35.22
	2	4673.79	5370.08	4104.03	29.70
	3	4472.23	5052.21	3966.93	28.48
	4	4635.88	5169.99	4066.02	30.23
	5	4682.20	5200.94	4141.39	27.89
Measurement 544 byte Ethernet frames	1	3678.13	4074.98	3395.57	22.62
	2	3771.15	4085.82	3482.80	21.95
	3	3863.14	4167.08	3597.12	19.14
	4	3832.58	4114.48	3586.54	17.90
	5	3831.10	5008.19	3549.99	20.67
Measurement 288 byte Ethernet frames	1	2599.77	3155.04	2469.05	16.55
	2	2615.16	2799.57	2502.80	17.08
	3	2626.40	2789.24	2521.28	11.71
	4	2648.82	2792.21	2527.52	18.12
	5	2658.44	3196.28	2534.79	18.35
Measurement 160 byte Ethernet frames	1	1502.55	1733.29	1462.78	18.09
	2	1530.66	1747.17	1495.08	15.55
	3	1505.28	1698.14	1475.08	13.60
	4	1494.75	1685.27	1464.10	14.00
	5	1495.30	1552.52	1457.62	17.29
Measurement 96 byte Ethernet frames	1	899.10	934.77	883.52	19.61
	2	1018.28	1093.50	997.35	20.60
	3	1015.76	1090.22	996.77	18.76
	4	1013.02	1085.16	985.80	27.25
	5	1001.80	1035.47	981.06	21.11
Measurement 64 byte Ethernet frames	1	481.38	507.38	470.48	48.12
	2	476.92	499.96	465.77	50.19
	3	488.73	513.51	478.98	41.61
	4	464.10	483.23	454.14	47.22
	5	476.11	496.06	466.26	44.36

Table 4.10: 264 frames 128x128 Video Measurement Python

		Frame Rate(fps)			Latency(us)
		Send	Receive	Send_and_Receive	
Measurement 1056 byte Ethernet frames	1	42.09	51.25	41.95	75.82
	2	38.32	46.05	38.21	38.21
	3	38.21	47.36	36.26	75.97
	4	39.24	47.18	39.13	73.58
	5	41.96	51.72	41.90	32.50
Measurement 544 byte Ethernet frames	1	31.92	35.08	31.89	27.24
	2	34.48	38.36	34.45	31.16
	3	33.74	37.50	33.65	80.24
	4	33.29	36.73	33.26	21.03
	5	34.03	37.49	34.00	23.07
Measurement 288 byte Ethernet frames	1	38.98	41.80	38.95	17.46
	2	36.35	38.67	36.33	17.34
	3	43.38	46.67	43.33	22.80
	4	42.93	45.70	42.90	20.62
	5	40.14	43.00	40.11	21.63
Measurement 160 byte Ethernet frames	1	35.53	37.09	35.50	19.93
	2	28.09	29.25	28.07	23.97
	3	27.92	29.08	27.91	19.41
	4	28.50	29.78	28.49	14.85
	5	28.80	30.06	28.78	20.86
Measurement 96 byte Ethernet frames	1	22.43	23.48	22.42	47.06
	2	21.69	22.69	21.68	37.20
	3	22.43	23.54	22.42	30.63
	4	19.79	20.47	19.76	30.88
	5	21.92	22.93	21.89	58.01
Measurement 64 byte Ethernet frames	1	14.48	15.08	14.46	92.00
	2	14.41	15.01	14.39	75.20
	3	14.32	14.90	14.30	53.90
	4	14.25	14.84	14.24	79.90
	5	14.49	15.11	14.47	62.44

Ethernet frames at a rate of 1 Gbps. However, at these frame rates Wireshark misses frames.

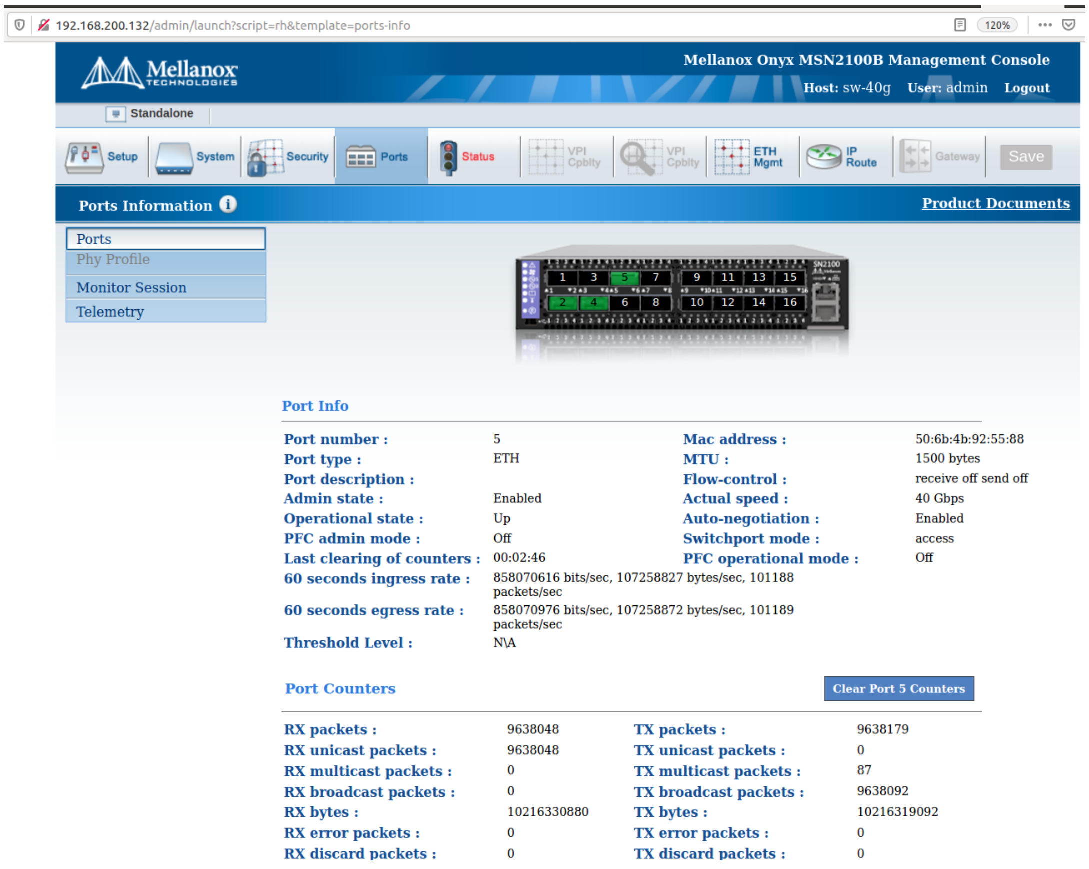


```
compute3@compute3:~/projects/trafgen$ sudo trafgen --cpp --dev ens3 --conf udp_example01.trafgen --rate 1000Mbit --cpus 4 -n 15206400 -t 3 --verbose
[sudo] password for compute3:
[0] pkt
 len 1056 cnts 0 rnds 0
 payload 04 00 00 05 00 00 16 1b 21 3c 9e 6c 22 22 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 4
2 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42
 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42
42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 4
2 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42
 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42
42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 4
2 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42
 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42
42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 4
2 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42
 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42
42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 4
2 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42
 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42
42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 4
2 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42
 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42
42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 4
2 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42
 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42
42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42
     1 packets to schedule
  1056 bytes in total
Running! Hang up with ^C!

Enabled kernel qdisc bypass

    15206400 packets outgoing
 16057958400 bytes outgoing
         150 sec, 332419 usec on CPU0 (15206400 packets)
compute3@compute3:~/projects/trafgen$
```

Figure 4.15: trafgen Packet Generator Tool Execution on Command Line

By looking at the port statistics of the "Workstation PC" from the web server interface of the 40 Gbps switch, the amount and rate of incoming and outgoing packets were measured. The web server interface of the 40 Gbps switch is as shown in Figure 4.16. Statistics on the web server interface show the average rate values over a one-minute period. Therefore, the packet generator was run for at least one minute and the number at which the data rate was saturated was found and recorded.

During the measurements, it was observed that when frames are sent at a high rate and low IFG are sent to the FPGA, the "40G ETH CORE" MAC, which is a 3rd party IP in the FPGA, crashes and cannot pass the packets any more till the next FPGA reconfiguration. The reason for this is thought to be that the buffer of the MAC IP core is very small and there is no flow control mechanism. It cannot handle the packet data rate and inter-frame time gap, which can change instantaneously for a small duration. Due to these reasons, the system could not be operated with maximum performance. Luckily, trafgen tool provides inter-frame time gap value setting. In this way, no problem is observed on the system when there is at least 3 us between Ethernet frames.



Port Info

Port number :	5	Mac address :	50:6b:4b:92:55:88
Port type :	ETH	MTU :	1500 bytes
Port description :		Flow-control :	receive off send off
Admin state :	Enabled	Actual speed :	40 Gbps
Operational state :	Up	Auto-negotiation :	Enabled
PFC admin mode :	Off	Switchport mode :	access
Last clearing of counters :	00:02:46	PFC operational mode :	Off
60 seconds ingress rate :	858070616 bits/sec, 107258827 bytes/sec, 101188 packets/sec		
60 seconds egress rate :	858070976 bits/sec, 107258872 bytes/sec, 101189 packets/sec		
Threshold Level :	N\A		

Port Counters

Clear Port 5 Counters

RX packets :	9638048	TX packets :	9638179
RX unicast packets :	9638048	TX unicast packets :	0
RX multicast packets :	0	TX multicast packets :	87
RX broadcast packets :	0	TX broadcast packets :	9638092
RX bytes :	10216330880	TX bytes :	10216319092
RX error packets :	0	TX error packets :	0
RX discard packets :	0	TX discard packets :	0

Figure 4.16: 40Gbps Melanox Switch Web Server Interface

4.11 Gaussian-Sobel Filters Workstation PC Implementation

The Gaussian Filter and Sobel Filter image processing pipeline are also run on the workstation CPU as a performance reference to the FPGA implementation. OpenCV functions are compiled with the g++ compiler for x86 architecture, passing the same parameters as the Gaussian and Sobel Filters implemented with Vitis HLS OpenCV.

```
void cv::GaussianBlur (InputArray src, OutputArray dst, Size ksize, double
↪ sigmaX, double sigmaY=0, int borderType=BORDER_DEFAULT)
```

Listing 9: Gaussian Filter OpenCV Function prototype

Measurements were taken by calling the image processing pipeline 1000 times in the for loop using the C++ Chrono library. In this way, reading and writing the frames from the file directory are not included in the measurements. Measurements were

```
void cv::Sobel (InputArray src, OutputArray dst, int ddepth, int dx, int
↪ dy, int ksize=3, double scale=1, double delta=0, int
↪ borderType=BORDER_DEFAULT)

void cv::addWeighted( InputArray src1, double alpha, InputArray src2,
↪ double beta, double gamma, OutputArray dst, int dtype = -1)
```

Listing 10: Sobel Filter and addWeighted OpenCV Function Prototypes

taken by calling the image processing pipeline 1000 times in the for loop using the C++ Chrono library. In this way, reading and writing the frames from the file directory are not included in the measurements. For 128x128 frames, an average of 2020.97 fps on Intel(R) Xeon(R) Bronze 3104 CPU @ 1.70GHz processor. For 128x128 frames, an average of 14705.88 fps measured on Intel(R) Core(TM) i7-4700HQ CPU @ 2.40GHz.

Here we note that the focus of this thesis is not design and implementation of accelerators that will outperform the same application that is implemented in software. Since the code is written as a single-threaded application, performance increases with CPU frequency, multi-threaded implementation results may increase performance in a better way. On the other hand, performance can be increased by repeating the hardware accelerator functionality on the FPGA in a much larger number of RRs. Also, accelerators can operate at higher frequencies when implemented on FPGAs produced with low nanometer FPGA fabric production process and can reach high F_{max} with state-of-the-art FPGA architecture technologies to get better performance.

Table 4.11: 128x128 Video Measurement on CPU

CPU	**Frequency(GHz)**	**Throughput(FPS)**
Intel(R) Xeon(R) Bronze 3104 CPU	1.70	2020.97
Intel(R) Core(TM) i7-4700HQ	2.40	14705.88

CHAPTER 5

CONCLUSION AND FUTURE WORK

This thesis addresses two critical technological trends: cloud computing and hardware acceleration and proposes a workflow for transparently offering hardware accelerators as cloud computing resources. To this end, we propose the virtualization of FPGA resources as partial reconfigurable regions and offer these regions in an independently programmable fashion. The RRs can be programmed by bitstream files that realize the hardware accelerators similar to Virtual Machine images for conventional cloud servers. Providing hardware accelerators as a part of Infrastructure as a Service (IaaS) is carried out through a well-known open-source cloud management platform, OpenStack.

Accordingly, the first part of the workflow in this thesis is implementing hardware-specific OpenStack software components on the SoC processor on the FPGA that we call Nova-Zynq. Nova-Zynq runs on the embedded Linux and can communicate with the rest of the OpenStack software through the defined APIs. Furthermore, we design and implement software components to run on the SoC processor that enables programming with the bitstream files received from the image database of OpenStack.

Providing hardware accelerators as a cloud computing service to users requires network connectivity to bring the user data for processing to the accelerator and then delivering it back to the user. Hence, the FPGA platform that we realize the workflow features high-speed Ethernet Interfaces. The static logic on the FPGA includes an on-chip switch that interconnects the accelerators to the Ethernet interfaces, to each other for distributed applications and to the SoC processor for service management. To this end, the realization of the hardware accelerators is explored with this communication infrastructure. We explored two accelerator design workflows. The first workflow is OpenCL-based with a well-defined interface between the application that runs on the CPU and its data exchange with the accelerator through shared memory. The second workflow is implementing the accelerators as standalone IP cores and remotely providing data exchange with the user or application through FPGA interfaces and custom-designed hardware accelerator shell. We implement the entire workflow on the Xilinx ZC706 board. The functional correctness and performance experiments are conducted throughout the thesis work. The experiments cover the effects of the Ethernet interface on the performance in accordance with the cloud data center operation.

This workflow has been validated on an actual hardware prototype demo setup and demonstrated in the thesis. Hardware acceleration development steps, simulation and implementation on real hardware are realized. There are generally GNU Linux, RTL synthesis and implementation tools in the design and simulation stages, and multiple of our custom Python C/C++ software written for testing. Similar design and simulation steps may be followed with tools provided by other FPGA vendors. The integrated operation of the Nova-Zynq embedded system software with the Openstack Controller components and also with the underlying hardware that includes a Linux kernel driver and hardware accelerators is described and shown. It shows how closely coupled FPGAs and processors can be utilized on the workflow of using hardware accelerators in cloud computing systems. Two different hardware accelerator development workflows are emphasized, and their advantages over each other are evaluated. We examined that OpenCL-based hardware acceleration flow has a structure that facilitates communication with processors connected to the FPGA via PCIe or AXI MM interface. We showed how a high-level application in processors is offloaded to hardware accelerators with this flow. In addition, we examined how AXI Stream-based hardware accelerators and the structure created with an FPGA shell can process high-speed stream data coming from the Ethernet interface to the FPGA. While realizing this, we showed the computer test software that generates the stream data and the effect of these software performances on the overall system performance. We have seen the effects of the differences in the Ethernet frame sizes and the Ethernet frame generator software implementation. When using the multi-threaded Ethernet frame generator tool, it was observed that the throughput from accelerators also increased. Higher throughput can be achieved by maintaining parallelism by iterating accelerators multiple times in RRs in the Cloud. The measurements taken together with the tests on the prototype system reveal that the system is scalable and has potential for improvement. Future studies include demonstrating the performance of hardware accelerator functions addressing different domains on this workflow, evaluating the usability of novel high-end and low-end FPGA hardware, and methodologically examining the optimum reconfigurable region resource allocation for accelerator functions.

REFERENCES

[1] "Vitis_Libraries_Vision." https://xilinx.github.io/Vitis_Libraries/vision/2021.2/api-reference.html, 2022.

[2] "Xilinx_ZYNQ-7000." https://www.xilinx.com/products/silicon-devices/soc/zynq-7000.html, 2021.

[3] E. G. Schmidt, "Cloud computing and hardware accelerated clouds," 2020.

[4] "Paket protokolleri - accloud dokümantasyon." `https://accloudmetu.gitlab.io/acc-doc/fpga/protocol/`. (Accessed on 01/28/2022).

[5] A. Erol, A. Yazar, and E. G. Schmidt, "Openstack generalization for hardware accelerated clouds," in *2019 28th International Conference on Computer Communication and Networks (ICCCN)*, pp. 1–8, IEEE, 2019.

[6] "Open Source Cloud Computing Platform Software - OpenStack." `https://www.openstack.org/software/`. (Accessed on 01/06/2022).

[7] B. Varghese and R. Buyya, "Next generation cloud computing: New trends and research directions," *Future Generation Computer Systems*, vol. 79, pp. 849–861, 2018.

[8] "Amazon ec2 instance types – amazon web services (aws)." https://aws.amazon.com/ec2/instance-types/, Amazon, Accessed: 2020-08-20.

[9] A. M. Caulfield, E. S. Chung, A. Putnam, H. Angepat, J. Fowers, M. Haselman, S. Heil, M. Humphrey, P. Kaur, J.-Y. Kim, *et al.*, "A cloud-scale acceleration architecture," in *The 49th Annual IEEE/ACM International Symposium on Microarchitecture*, p. 7, IEEE Press, 2016.

[10] J. L. Hennessy and D. A. Patterson, "A new golden age for computer architecture," *Communications of the ACM*, vol. 62, no. 2, pp. 48–60, 2019.

[11] A. Yazar, A. Erol, and E. G. Schmidt, "Accloud (accelerated cloud): A novel fpga-accelerated cloud archictecture," in *2018 26th Signal Processing and Communications Applications Conference (SIU)*, pp. 1–4, IEEE, 2018.

[12] "Xilinx_ZC-706."
https://www.xilinx.com/products/boards-and-kits/ek-z7-zc706-g.html, 2021.

[13] "Zynq_Ultrascale+."
https://www.xilinx.com/products/silicon-devices/soc/zynq-ultrascale-mpsoc.html, 2022.

[14] "Intel_Agilex."
https://www.intel.com/content/www/us/en/products/details/fpga/agilex.html, 2022.

[15] "Microsemi_PolarFire."
https://www.microsemi.com/product-directory/soc-fpgas/5498-polarfire-soc-fpga, 2022.

[16] "Lattice_Risc-V_IP."
https://www.latticesemi.com/products/designsoftwareandip/intellectualproperty/ipcore/ipcores04/riscvsı
2022.

[17] G. E. Moore, "Cramming more components onto integrated circuits," *Proceedings of the IEEE*, vol. 86, no. 1, pp. 82–85, 1998.

[18] R. H. Dennard, F. H. Gaensslen, H.-N. Yu, V. L. Rideout, E. Bassous, and A. R. LeBlanc, "Design of ion-implanted mosfet's with very small physical dimensions," *IEEE Journal of Solid-State Circuits*, vol. 9, no. 5, pp. 256–268, 1974.

[19] M. Vestias and H. Neto, "Trends of cpu, gpu and fpga for high-performance computing," in *2014 24th International Conference on Field Programmable Logic and Applications (FPL)*, pp. 1–6, IEEE, 2014.

[20] Y. Zhou, U. Gupta, S. Dai, R. Zhao, N. Srivastava, H. Jin, J. Featherston, Y.-H. Lai, G. Liu, G. A. Velasquez, *et al.*, "Rosetta: A realistic high-level synthesis benchmark suite for software programmable fpgas," in *Proceedings of the 2018*

ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, pp. 269–278, 2018.

[21] M. Owaida, G. Alonso, L. Fogliarini, A. Hock-Koon, and P.-E. Melet, “Lowering the latency of data processing pipelines through fpga based hardware acceleration,” *Proceedings of the VLDB Endowment*, vol. 13, no. 1, pp. 71–85, 2019.

[22] A. Vaishnav, K. D. Pham, and D. Koch, “A survey on fpga virtualization,” in *2018 28th International Conference on Field Programmable Logic and Applications (FPL)*, pp. 131–1317, IEEE, 2018.

[23] H. K.-H. So and C. Liu, “Fpga overlays,” in *FPGAs for Software Programmers*, pp. 285–305, Springer, 2016.

[24] L. M. Al Qassem, T. Stouraitis, E. Damiani, and I. A. M. Elfadel, “Fpgaaas: A survey of infrastructures and systems,” *IEEE Transactions on Services Computing*, 2020.

[25] “Xilinx_Alveo.” https://www.xilinx.com/products/boards-and-kits/alveo.html, 2021.

[26] M. Meyer, T. Kenter, and C. Plessl, “Evaluating FPGA accelerator performance with a parameterized OpenCL adaptation of selected benchmarks of the HPCChallenge benchmark suite,” in *2020 IEEE/ACM International Workshop on Heterogeneous High-performance Reconfigurable Computing (H2RC)*, pp. 10–18, IEEE, 2020.

[27] A. Tırlıoğlu, O. B. Demir, A. Yazar, and E. G. Schmidt, “Hardware accelerators for cloud computing: Features and implementation,” in *2021 29th Signal Processing and Communications Applications Conference (SIU)*, pp. 1–4, 2021.

[28] P. Luszczek, J. J. Dongarra, D. Koester, R. Rabenseifner, B. Lucas, J. Kepner, J. McCalpin, D. Bailey, and D. Takahashi, “Introduction to the HPC challenge benchmark suite,” tech. rep., Ernest Orlando Lawrence Berkeley NationalLaboratory, Berkeley, CA (US), 2005.

[29] C. Fu and Y. Yu, "Fpga-based power efficient face detection for mobile robots," in *2019 IEEE International Conference on Robotics and Biomimetics (ROBIO)*, pp. 467–473, IEEE, 2019.

[30] "Xilinx Vitis 1-Dimensional(line) FFT L1 FPGA module." https://xilinx.github.io/Vitis_Libraries/dsp/2020.2/user_guide/L1.html#l1-performance-benchmarks-and-qor, 2021.

[31] S. Yang, P. Luo, C.-C. Loy, and X. Tang, "Wider face: A face detection benchmark," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 5525–5533, 2016.

[32] J. E. Stone, D. Gohara, and G. Shi, "Opencl: A parallel programming standard for heterogeneous computing systems," *Computing in science & engineering*, vol. 12, no. 3, p. 66, 2010.

[33] "https://xilinx.github.io/vitis-tutorials/2020-2/docs/getting_started/vitis/part1.html." `https://xilinx.github.io/Vitis-Tutorials/2020-2/docs/Getting_Started/Vitis/Part1.html`. (Accessed on 01/26/2022).

[34] "Xilinx_XRT." https://www.xilinx.com/products/design-tools/embedded-software/petalinux-sdk.html, 2021.

[35] "Vitis libraries." `https://www.xilinx.com/products/design-tools/vitis/vitis-libraries.html#overview`. (Accessed on 01/28/2022).

[36] "Xilinx_Vitis." https://www.xilinx.com/products/design-tools/vitis.html, 2021.

[37] N. Brown, "Weighing up the new kid on the block: Impressions of using vitis for hpc software development," in *2020 30th International Conference on Field-Programmable Logic and Applications (FPL)*, pp. 335–340, IEEE, 2020.

[38] "Amazon_EC2_F1_Instances." https://aws.amazon.com/ec2/instance-types/f1/, Amazon, Accessed: 2020-08-20.

[39] D. Korolija, T. Roscoe, and G. Alonso, "Do {OS} abstractions make sense on fpgas?," in *14th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 20)*, pp. 991–1010, 2020.

[40] S. A. Fahmy, K. Vipin, and S. Shreejith, "Virtualized fpga accelerators for efficient cloud computing," in *2015 IEEE 7th International Conference on Cloud Computing Technology and Science (CloudCom)*, pp. 430–435, IEEE, 2015.

[41] N. U. Ekici, K. W. Schmidt, A. Yazar, and E. G. Schmidt, "Resource allocation for minimized power consumption in hardware accelerated clouds," in *2019 28th International Conference on Computer Communication and Networks (ICCCN)*, pp. 1–8, IEEE, 2019.

[42] X. Wang, Y. Niu, F. Liu, and Z. Xu, "When fpga meets cloud: A first look at performance," *IEEE Transactions on Cloud Computing*, 2020.

[43] S. Byma, J. G. Steffan, H. Bannazadeh, A. Leon-Garcia, and P. Chow, "Fpgas in the cloud: Booting virtualized hardware accelerators with openstack," in *2014 IEEE 22nd Annual International Symposium on Field-Programmable Custom Computing Machines*, pp. 109–116, IEEE, 2014.

[44] N. Tarafdar, T. Lin, E. Fukuda, H. Bannazadeh, A. Leon-Garcia, and P. Chow, "Enabling flexible network fpga clusters in a heterogeneous cloud data center," in *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pp. 237–246, 2017.

[45] F. Yazıcı, A. S. Yıldız, A. Yazar, and E. G. Schmidt, "An on-chip switch architecture for hardware accelerated cloud computing systems," in *2020 28th Signal Processing and Communications Applications Conference (SIU)*, pp. 1–4, 2020.

[46] F. Yazıcı, A. S. Yıldız, A. Yazar, and E. G. Schmidt, "A novel scalable on-chip switch architecture with quality of service support for hardware accelerated cloud data centers," in *2020 IEEE 9th International Conference on Cloud Networking (CloudNet)*, pp. 1–4, 2020.

[47] F. Yazıcı, "A novel flexible on-chip switch architecture for reconfigurable hardware accelerators," Master's thesis, Middle East Technical University, 2021.

[48] A. Erol, "Generalized resource management for heterogeneous cloud data centers," Master's thesis, Middle East Technical University, 2019.

[49] A. Erol, A. Yazar, and E. G. Schmidt, "A generalization of openstack for managing heterogeneous cloud resources," in *2019 27th Signal Processing and Communications Applications Conference (SIU)*, pp. 1–4, IEEE, 2019.

[50] "Rabbitmq."
https://www.rabbitmq.com/, 2022.

[51] "Xilinx_Petalinux."
https://www.xilinx.com/products/design-tools/embedded-software/petalinux-sdk.html, 2021.

[52] "Fpga_Manager."
https://www.kernel.org/doc/html/v5.0/driver-api/fpga/fpga-mgr.html, 2022.

[53] "GitHub - pika/pika: Pure Python RabbitMQ/AMQP 0-9-1 Client Library." `https://github.com/pika/pika`. (Accessed on 01/26/2022).

[54] "Yocto."
https://www.yoctoproject.org, 2022.

[55] "Openstack_API."
`https://docs.openstack.org/api-quick-start/`, 2022.

[56] "Xilinx_UG909."
https://www.xilinx.com/content/dam/xilinx/support/documentation/sw_manuals/xilinx2021_2/ug909-vivado-partial-reconfiguration.pdf#nameddest=VivadoSoftwareFlow, 2021.

[57] "Opencl."
https:https://www.khronos.org/opencl/, 2022.

[58] "Xilinx Vitis Canny Edge." https://github.com/Xilinx/Vitis_Libraries/tree/master/vision/L2/examples/canny, 2021.

[59] “Openstack_Compute_API.”
https://docs.openstack.org/api-ref/compute/, 2022.

[60] “ACCLOUD_Hardware_Accelerator_Flow_Demonstration.”
https://www.youtube.com/watch?v=ty5erUSYHwU, 2021.

[61] “Zynq_7000_TRM.”
https://www.xilinx.com/support/documentation/user_guides/ug585-Zynq-7000-TRM.pdf, 2022.

[62] “Opencv.”
https:https://opencv.org/, 2022.

[63] “socket.”
https:https://man7.org/linux/man-pages/man2/socket.2.html, 2022.

[64] “Wireshark.”
https:https://www.wireshark.org/, 2022.

[65] “libpcap.”
https:https://www.tcpdump.org/, 2022.

[66] “tcpdump.”
https:https://www.tcpdump.org/manpages/pcap-tstamp.7.html, 2022.

[67] “packeth.”
https:https://github.com/jemcek/packETH, 2022.

[68] “trafgen.”
https:https://man7.org/linux/man-pages/man8/trafgen.8.html, 2022.