Technical University of Crete

Diploma Thesis

# Methodology and Experiments for Large Scale Distributed Computation on Reconfigurable Logic – Based Platform

*Author:*
Christini Tzortzaki

*Thesis Committee:*
Prof. Apostolos Dollas
Assoc. Prof. Vasilis Samoladas
Asst. Prof. Vasileios Papaefstathiou (UoC)

*A thesis submitted in fulfillment of the requirements for the diploma of Electrical and Computer Engineer*

*in the*

School of Electrical and Computer Engineering
Microprocessor and Hardware Laboratory

February 8, 2024

TECHNICAL UNIVERSITY OF CRETE

# *Abstract*

School of Electrical and Computer Engineering

Electrical and Computer Engineer

**Methodology and Experiments for Large Scale Distributed Computation on Reconfigurable Logic – Based Platform**

by Christini Tzortzaki

Hardware accelerators have become crucial due to their superior performance and energy efficiency. One noteworthy example is the application of hardware accelerators to Convolutional Neural Networks (CNNs), which are computationally intensive and highly parallelizable. Recent research has demonstrated significant performance improvements when implementing CNNs with hardware accelerators. This study builds upon the CNN hardware accelerator developed by G. Pitsis [1] and C. Loukas [2] for the Xilinx ZCU102 and the QFDB multi-FPGA prototype board, respectively, and aims at the migration of the accelerator to the Alveo U50 Data Center Card and investigates opportunities for further scaling.

Through a series of experiments, the performance of the migrated CNN architecture on the Alveo U50 is evaluated and compared with its implementations on the Xilinx ZCU102 and the QFDB. Not only were the tools changed to enable the architecture's execution on the Alveo platform, but modifications were also necessary for the architecture itself. As a result, using a similar FPGA, a 22% improvement in throughput was achieved. The migration to the Alveo U50 is shown to result in improved computational efficiency, showcasing the platform's enhanced capabilities for large-scale distributed computation. Furthermore, the utilization of multiple compute units is explored as a means of achieving parallelization, leading to enhanced throughput and overall better performance.

TECHNICAL UNIVERSITY OF CRETE

# *Abstract*

School of Electrical and Computer Engineering

Electrical and Computer Engineer

**Methodology and Experiments for Large Scale Distributed Computation on Reconfigurable Logic – Based Platform**

by Christini Tzortzaki

Οι επιταχυντές υλικού έχουν γίνει κρίσιμοι λόγω της υπερέχουσας επίδοσης και της ενεργειακής τους απόδοσης. Ένα αξιοσημείωτο παράδειγμα είναι η εφαρμογή επιταχυντών υλικού σε συνελικτικά νευρωνικά δίκτυα (CNN), τα οποία είναι υπολογιστικά εντατικά και εξαιρετικά παραλληλίσιμα. Πρόσφατες έρευνες έχουν δείξει σημαντικές βελτιώσεις στην απόδοση κατά την εφαρμογή συνελικτικού νευρωνικού δικτύου (CNN) με επιταχυντές υλικού. Αυτή η μελέτη βασίζεται στον επιταχυντή υλικού CNN που αναπτύχθηκε από τους Γ. Πίτση [1] και Χ. Λουκά [2] για την Xilinx ZCU102 και την πρωτότυπη πλακέτα πολλαπλών FPGA QFDB, αντίστοιχα. Αυτή η διατριβή στοχεύει στην μετεγκατάσταση του επιταχυντή στην Alveo U50 και διερευνά ευκαιρίες για περαιτέρω κλιμάκωση.

Μέσα από μια σειρά πειραμάτων, αξιολογείται η απόδοση της αρχιτεκτονικής CNN στο Alveo U50 και συγκρίνεται με τις υλοποιήσεις της στο ZCU102 και το QFDB. Δεν χρειάστηκε μόνο να αλλάξουν τα εργαλεία για να καταστεί δυνατή η εκτέλεση της αρχιτεκτονικής στην πλατφόρμα Alveo, αλλά και τροποποιήσεις ήταν απαραίτητες για την ίδια την αρχιτεκτονική. Ως αποτέλεσμα, χρησιμοποιώντας FPGAs παρόμοιας τεχνολογίας, επιτεύχθηκε 22% βελτίωση στην απόδοση. Η μετάβαση στο Alveo U50 φαίνεται να έχει ως αποτέλεσμα βελτιωμένη υπολογιστική απόδοση, επιδεικνύοντας τις βελτιωμένες δυνατότητες της πλατφόρμας για κατανεμημένους υπολογισμούς μεγάλης κλίμακας. Επιπλέον, η χρήση πολλαπλών υπολογιστικών μονάδων διερευνάται ως μέσο για την επίτευξη παραλληλισμού, που οδηγεί σε βελτιωμένη απόδοση και συνολικά καλύτερη απόδοση.

# *Acknowledgements*

First of all, I would like to thank my supervisor, Prof. Apostolos Dollas, for his constant support and guidance both during my work on this thesis and throughout the course of my studies.

This thesis could of course not have been completed wihtout the valuable help of the staff of the MHL laboratory in TUC. I would also like to thank Prof. Vasilis Samoladas and Prof. Vasileios Papaefstathiou for being members of my committee, and evaluating my work Most of all I would like to thank Dr. Pavlos Malakonakis for valuable advice and his insightful input on my work.

Last but not least I would like to express my deepest gratitude to the people who have always been there for me, my family and friends.

Christini Tzortzaki,
Chania, 2024

# Contents

# List of Figures

# List of Tables

# List of Algorithms

# List of Abbreviations

| | |
|---|---|
| **ALU** | **A**rithmetic **L**ogic **U**nit |
| **ASIC** | **A**pplication **S**pecific **I**ntegrated **C**ircuit |
| **BRAM** | **B**lock **R**andom **A**ccess **M**emory |
| **CPU** | **C**entral **P**rocessor **U**nit |
| **CS** | **C**omputer **S**cience |
| **DDR4** | **D**ouble **D**ata **R**ate type texbf4 memory |
| **DRAM** | **D**ynamic **R**andom **A**ccess **M**emory |
| **DSP** | **D**igital **S**ignal **P**rocessor |
| **FF** | **F**lip **F**lops |
| **FPGA** | **F**ield **P**rogrammable **G**ate **A**rray |
| **GDDR6** | **G**raphics **D**ouble **D**ata **R**ate type **6** memory |
| **GPU** | **G**raphic **P**rocessor **U**nit |
| **HBM** | **H**igh **B**andwidth **M**emory |
| **HDL** | **H**ardware **D**escription **L**anguage |
| **HLS** | **H**igh **L**evel **S**ynthesis |
| **HPC** | **H**igh **P**erformance **C**omputing |
| **LUT** | **L**ook **U**p **T**able |
| **MPSoC** | **M**ulti **P**rocessor **S**ystem **o**n **C**hip |
| **PL** | **P**rogrammable **L**ogic |
| **PS** | **P**rocessing **S**ystem |
| **RAM** | **R**andom **A**ccess **M**emory |
| **SDK** | **S**oftware **D**evelopment **K**it |
| **SIMD** | **S**ingle **I**nstruction **M**ultiple **D**ata |
| **SSE** | **S**treaming **S**IMD **E**xtensions |
| **SSD** | **S**olid **S**tate **D**rive |
| **TDP** | **T**hermal **D**esign **P**ower |
| **URAM** | **U**ltra **R**andom **A**ccess **M**emory |
| **USD** | **U**nited **S**tates **D**ollar |

*Dedicated to my family and friends...*

# Chapter 1

# Introduction

In recent years, the field of computational research has witnessed a paradigm shift towards the utilization of reconfigurable logic-based platforms for large-scale distributed computation. This transition has been fueled by the increasing demand for efficient and high-performance solutions to address complex computational problems, particularly in the domain of deep learning. Convolutional Neural Networks (CNNs), a pivotal component in the realm of artificial intelligence, have proven to be instrumental in tasks such as image recognition, natural language processing, and autonomous systems.

Work in [1], [3] presents the acceleration of a Convolutional Neural Network (CNN) architecture on a multi-FPGA platform, given strict limitations regarding bandwidth and energy consumption. More specifically, it provides different ways to minimize the impact of such hardware limitations, accelerating the inference of the neural network on an FPGA platform. In this work, we make use of these results, and - building upon the work of Loukas in [2] - we propose an architecture that aims to further accelerate the inference of the CNN on a larger scale platform. In particular, we propose a parallel architecture utilizing the resources of the Alveo U50 Data Center accelerator card[4].

While the integration of reconfigurable logic promises enhanced computational capabilities, the process of migrating existing designs to such platforms is not without challenges. The complexity lies not only in adapting the design to the architecture of the Alveo U50 but also in leveraging the full potential of multiple compute units to achieve efficient parallelization. This research addresses these challenges head-on, aiming to provide insights into the intricacies of large-scale distributed computation on reconfigurable logic-based platforms.

The primary objectives of this research are as follows:

- Develop a comprehensive methodology for migrating a Convolutional Neural Network (CNN) design from ZCU102 and QFDB to the Alveo U50 reconfigurable logic-based platform.
- Implement and optimize the migrated CNN design on the Alveo U50, utilizing the parallel processing capabilities offered by multiple compute units.
- Conduct a thorough evaluation and comparison of the performance of the CNN design on the different platforms, considering factors such as execution time, resource utilization, and energy efficiency.

This research focuses on the migration and optimization of CNN designs specifically, exploring the challenges and opportunities presented by large-scale distributed computation on reconfigurable logic-based platforms. The experimentation involves quantitative analysis and comparison of results obtained from ZCU102, QFDB, and Alveo U50, providing valuable insights into the potential advantages and limitations of each platform.

The outcomes of this research will contribute to the broader understanding of the feasibility and effectiveness of large-scale distributed computation on reconfigurable logic-based platforms. The findings are expected to be valuable for researchers and practitioners in the fields of deep learning, FPGA-based computing, and distributed systems.

## 1.1 Thesis Outline

In this section we present an outline of the organization of this thesis:

- **Chapter 2 - Related Work:** We describe in detail the related work in the field of multi-FPGA and the high performance computing.
- **Chapter 3 - Design Flow:** We describe in detail the design flow in Xilinx Tools, Vivado Design Suite and Vitis Unified Software Platform
- **Chapter 4 - Methodology for transforming Vivado to Vitis:** We describe step-by-step the systemic work that was done in order to develop the design into Vitis and Alveo U50.
- **Chapter 5 - FPGA Implementation:** We develop an architecture for three single FPGAs (ZCU102, ALVEO U50 and VCU118) and for a Quad FPGA (QFDB).
- **Chapter 6 - Results:** We present the resource utilization and performance results of this work.
- **Chapter 7 - Conclusions and Future Work:** We conclude this thesis, and we provide directions for future work and possible extensions to our work.

# Chapter 2

# Related Work

In this chapter, we will present related work in the field of Convolutional Neural Networks (CNN), as well as the tools and design flows for FPGA implementation thereof. Due to the very large body of knowledge and published work, especially in CNN and their FPGA implementations, this chapter and the associated references will be largely on works related to the present thesis.

## 2.1 Convolutional Neural Networks (CNNs)

CNNs [5] are deep neural network architectures that took research by storm during the last decades, due to their efficiency in machine learning applications that involve images. Their main advantage compared to more traditional architectures is that they require minimal information regarding the features of the machine learning task, since the feature extraction is part of the training phase.

More specifically, a typical CNN consists of two different parts. The first one contains the so-called **Convolutional Layers** and the second one contains a traditional **Fully Connected feed forward neural network**. A CNN example is depicted in Fig. 2.1.

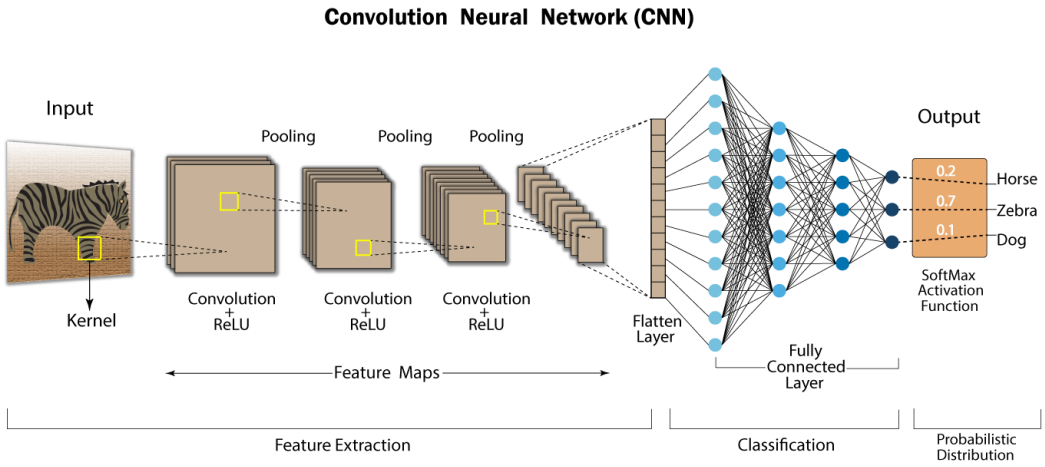


FIGURE 2.1: A CNN architecture. Source: [6]

### 2.1.1 Convolutional Layers

As the name suggests, convolutional layers involve the mathematical operation that is called convolution. More specifically, each layer consists of a certain number of randomly initialized **kernels-filters**, followed by a nonlinear function called **activation**, and optionally a **pooling layer**, which aims to reduce the dimensionality for computational efficiency. As it can be observed in Fig. 2.1, the input image is convolved with each filter of the first convolutional layer, and the resulting (filtered) images are passed to the next layer, after being passed through the activations and (possibly) the pooling operation.

What should be highlighted is that during the training phase, the network is able to compute the values of each kernel-filter of each layer that are best for the machine learning task of interest; those kernels are called **feature maps**. This is crucial, since handcrafting features, especially in datasets that involve images, is a task of immense difficulty.

### 2.1.2 Fully Connected Layers

After the Convolutional Layers, the architecture of the network typically involves a **Fully Connected feed forward neural network**. In particular, the output of the last Convolutional Layer, after being flattened utilizing a so-called **Flatten Layer** (meaning that the resulting kernels of the convolutions are stacked as a vector), is passed as an input to the Fully Connected layer, whose output depends on the machine learing task. For instance, in Fig. 2.1, since the CNN is utilized for a classification task, the output of the Fully Connected

layer involves a softmax activation and the output refers to the probability of an image belonging to a certain class.

## 2.2 Accelerations on FPGAs

In recent years, researchers have explored various approaches to address the computational complexity challenges associated with Convolutional Neural Networks (CNNs) on Alveo cards. The following literature review provides an overview of relevant studies in this domain, focusing on three key papers that propose novel solutions for enhancing the performance and efficiency of CNN inference accelerators on Alveo cards.

### 2.2.1 CNNs on ZCU102

Work in [7] proposes an FPGA-based CNN acceleration architecture using a mixed on-chip/off-chip memory strategy that achieves 10x speedup over CPU for VGG-16 and FCN models. The authors in [8] propose an object detection model called Agilev4 that achieves high accuracy and energy efficiency by optimizing the neural network architecture to balance processing speed, power consumption, and mean average precision. In [9], the authors present the design and implementation of a convolutional neural network accelerator that exploits sparsity and features hierarchical memory organization to achieve 1 TOPS throughput. In [10], the authors present an FPGA-based CNN accelerator that achieves 160 G-op/s peak performance and 96% resource utilization through optimizations for image edge processing, seamless channel switching, and reduced shift register chain length. In [11], the authors present an FPGA accelerator for sparse convolutional neural networks. The accelerator uses a weight-oriented dataflow which performs element-matrix multiplication as the core computation and employs architecture optimizations including a tile look-up table and channel multiplexer. Work in [12] proposes power efficient FPGA-SoC design techniques such as optimized FIFO blocks and reconfigured hardware resources for a CNN-based object detection accelerator, achieving 10% total power reduction. It also shows the potential for traditional low power RTL techniques to reduce power in CNN hardware accelerators, demonstrating a 25.9% FIFO power reduction. In [13], the authors present an automated framework to generate optimized FPGA-based hardware accelerators for deep neural networks: the tool takes a user's specifications and DNN model as input and outputs synthesizable HDL code, with optimizations in terms of latency, resource utilization,

and power efficiency. In [14], the authors propose an uninterrupted processing technique for a CNN accelerator, allowing simultaneous PE operations and data fetching, and present a low latency VLSI architecture using a random access line buffer PE array, achieving 587.52 GOPS throughput and 142.95 GOPs/W energy efficiency on an FPGA prototype.

### 2.2.2 CNNs on Alveo U50

In [15], the authors propose an FPGA-based accelerator for randomly wired neural networks (RWNNs) that uses multiple separable convolution engines to process the layers in parallel, leveraging the intrinsic parallelism of the RWNN structure. Their design utilizes HBM2 memory and a crossbar switch to enable concurrent access to feature maps, and allocates layers to compute units and HBM channels using heuristic scheduling and graph coloring algorithms Figure 2.2. In [16], the authors propose an FPGA CNN accelerator using Number Theoretic Transform that attain 2859.5 GOPS throughput, surpassing existing FFT and Winograd-based accelerators by 9.6x.

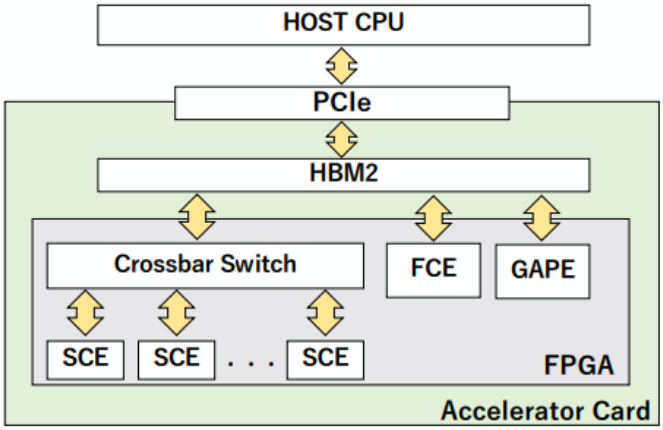


FIGURE 2.2: Architecture Design. Source: [15]

In [17], OpenMDS is presented as an open-source framework, proposed for optimizing high-performance designs on Xilinx multi-die FPGAs, using customized techniques to address timing issues from die-crossing signals. Work in [18] proposes techniques to optimize the utilization of High Bandwidth Memory (HBM) on FPGAs when using high-level synthesis (HLS) tools. Specifically, it introduces methods called Batched Inter-Channel Arbitrator (BICA) and Batched Inter-PE Arbitrator (BIPA) to improve memory bandwidth when accessing multiple HBM channels per processing element or allowing access to a channel from

multiple processing elements, respectively. In [19], the authors present machine learning algorithms based on convolutional neural networks and autoencoders for selecting events with long-lived particle decays using FPGA acceleration cards. Finds acceptable performance degradation after model quantization and that the tested CPU, GPU and FPGA setups fit latency constraints.

### 2.2.3 Descroption of the CNN Model Used in the Present Thesis

The central theme of this thesis pertains to the deployment of a previously developed Convolutional Neural Network (CNN) architecture across various platforms. This selection of platforms serves as the basis for our investigation and evaluation of the CNN's efficacy and versatility within diverse hardware contexts. Subsequently, we will proceed to furnish a comprehensive exposition of the architecture, elucidating its intricacies. This model is described in detail in two Technical University of Crete, School of ECE Theses [1, 2].

The work [3] presents a comprehensive study on implementing Convolutional Neural Networks (CNNs) on Field Programmable Gate Array (FPGA) platforms for signal analysis tasks, particularly focusing on space data classification. The motivation stems from the need for onboard processing in satellite-based remote sensing platforms, where bandwidth and energy constraints are critical. The researchers explore various weight compression techniques to address the limitations of FPGA hardware, including weight pruning, fixed-point operations, and weight clustering. They demonstrate significant reductions in memory footprint while maintaining competitive classification accuracy, paving the way for efficient realization of CNN-based inference architectures in FPGA platforms.

The most significant accomplishment of this work lies in its successful demonstration of highly efficient CNN implementation on FPGA platforms for space data classification. By effectively compressing model parameters through sophisticated weight compression techniques, such as pruning and clustering, the researchers achieve substantial reductions in memory requirements without sacrificing classification accuracy. This breakthrough enables the deployment of CNN-based inference directly onboard satellites, overcoming the limitations of traditional ground-based processing paradigms. Moreover, the comparative evaluation against Graphics Processing Units (GPUs) highlights the superior energy efficiency and competitive performance of FPGA-based implementations,

underscoring the potential of this approach for real-world space applications with strict bandwidth and energy constraints.

# Chapter 3

# Analysis of Design Vitis/Vivado Flow

Vitis Unified Software Platform and Vivado Design Suite are both tools provided by Xilinx for developing and accelerating applications on Xilinx devices. While Vivado is primarily used for hardware design and implementation, Vitis is a higher-level development platform that allows you to create and optimize applications for accelerated computing. The following is an overview of each tool and the process involved in programming or enhancing a device.

## 3.1 Vivado

### 3.1.1 Vivado Tools

#### Vivado High Level Synthesis (HLS)

The Xilinx Vivado High Level Synthesis (HLS) tool [20] is used to transform a C program into a register transfer level (RTL) implementation that can be easily syntesized into a Xilinx FPGA. The main advantage of utilizing this technology is the fact that both the development and the testing of the algorithms occur at the C-level, allowing designers to work at a higher level of abstraction and, thus, more quickly than using traditional hardware description languages (HDLs). In addition, it provides optimization directives that can be used for exploring the design space and finding an optimal implementation.

The HLS main phases are:

- **Scheduling**: Determines which operation occurs during on each clock cycle based on both the clock frequency and the time for the operation to

complete, as well as the optimization directives that are specified by the user.

- **Binding**: Determines which hardware resource implements each scheduled operation.
- **Contro logic extraction**: Extracts the control logic to create a finite state machine (FSM) that sequences the operations in the RTL design.

In order to determine whether a design meets the requirements, HLS provides designers with some performance metrics that can be used to refine the implementation. Those metrics are:

- **Area**: The amount of hardware resources required to implement the design based on the resources available in the FPGA.
- **Latency**: Number of clock cycles required for the function to compute all the output values.
- **Initiation interval (II)**: Number of clock cycles before the function can accept new input data.
- **Loop iteration latency**: Number of clock cycles it takes to complete one iteration of the loop.
- **Loop initiation interval**: Number of clock cycles before the next iteration of the loop starts to process data.
- **Loop latency**: Number of cycles to execute all iterations of the loop.

### Vivado IDE

The Vivado IDE is the GUI for the Vivado Design Suite.

Vivado IDE [21] can compile, synthesize, implement, place and route FPGA hardware designs written in high-level languages such as C/C++, and HDLs such as VHDL and Verilog.

In addition, using the IP Integrator tool, hardware systems can be designed by graphically connecting IP blocks and configuring them through their GUI, with no coding involved, hence, accelerating the design process.

After the design process is completed, a bitstream can be created and then downloaded to the target FPGA device to run as a standalone hardware device or in combination with firmware running on the FPGA's integrated ARM cores.

The device's firmware is developed, compiled, and deployed using the Xilinx Software Development Kit (SDK) tool.

All of the Vivado design Suite tools are written with a native Tcl interface, and all of those commands are available through the IDE either through the GUI or through the Tcl console. Tcl commands can be entered in the Tcl Console in the Vivado IDE or using the Vivado Design Suite Tcl shell. You can run analysis and assign constraints throughout the design process. Timing and power estimations are provided after synthesis, placement, and routing.

#### Xilinx SDK and Xilinx Vitis IDE

Xilinx SDK (Software Development Kit) is a comprehensive software development environment provided by Xilinx, a leading provider of programmable logic devices. It is part of the Xilinx Vivado Design Suite, which is a suite of tools used for designing, developing, and programming Xilinx FPGAs (Field-Programmable Gate Arrays) and SoCs (System-on-Chips).

Xilinx SDK allows developers to create software applications for Xilinx devices, including embedded processors like ARM Cortex-A9 and MicroBlaze. It provides a set of tools, libraries, and APIs that facilitate the development process, enabling software engineers to write, compile, debug, and deploy software on Xilinx platforms.

Key features of Xilinx SDK include:

- Integrated Development Environment (IDE): Xilinx SDK includes an Eclipse-based IDE tailored for FPGA and SoC development. It offers features like code editing, project management, and debugging capabilities.
- Cross-Compilation Toolchain: Xilinx SDK includes a toolchain that supports cross-compilation, allowing developers to write code on their host machine and compile it for the target Xilinx device.
- Board Support Packages (BSPs): Xilinx provides pre-configured BSPs for various Xilinx development boards and platforms. BSPs include device drivers, libraries, and example code specific to a particular board, simplifying the development process.
- Debugging and Profiling: Xilinx SDK supports various debugging and profiling features, including source-level debugging, breakpoints, and performance analysis tools. It allows developers to track and diagnose issues in their software applications.

- Libraries and APIs: Xilinx SDK provides a collection of libraries and APIs that enable developers to leverage hardware-accelerated features and interfaces provided by Xilinx devices. This includes libraries for communication, signal processing, and other specialized tasks.

### 3.1.2 Vivado Design Flow

Below is presented detailed description of the process of creating an accelerator for an FPGA using Vivado HLS, Vivado IDE and Vivado-Vitis SDK.

**Vivado-Vitis HLS**: Use Vivado HLS to convert the C/C++ function into RTL (Register Transfer Level) code. Vivado HLS automatically generates RTL code that describes the behavior of the function. This process is called high-level synthesis. First, import the C/C++ function as the top-level function. Specify synthesis options like the target FPGA device and performance goals. Next, run synthesis to generate the RTL code. Vivado HLS optimizes the code, adds pipelining, and generates detailed RTL implementation.

Subsequently, export the generated RTL code from Vivado HLS and create an IP (Intellectual Property) core for integration into Vivado. Export the RTL code as an IP core from Vivado HLS. This encapsulates the accelerator functionality as a reusable component.

**Vivado Integrated Design Environment (IDE)**: Consequently, proceed to transition to the Vivado IDE. First, Launch Vivado and create a new project targeting the FPGA platform. Import the desired IPs into the project and import the generated IP core or more IP cores, that were generated in Vivado HLS. Configure project settings like the target device, I/O interfaces, and clocking. Connect the IP cores to the appropriate interfaces. Finally perform synthesis, implementation, and bitstream generation. Vivado generates a bitstream file containing FPGA configuration.

**Vivado SDK**: The final tool is the Vivado SDK. Vivado SDK is to provide a complete software development environment for designing, implementing, and debugging embedded software applications targeting Xilinx FPGA devices. First, launch Vivado SDK and create a new software project. Select the hardware platform corresponding to the FPGA device. Then, develop software to interact with the accelerator running on the FPGA by creating software components to communicate with the accelerator. Compile and link the software project to generate the executable. Finally, download the bitstream onto the FPGA and run the software on the host CPU to control the accelerator and exchange data.

Vivado SDK provides libraries and APIs to facilitate communication between the host CPU and the FPGA accelerator.

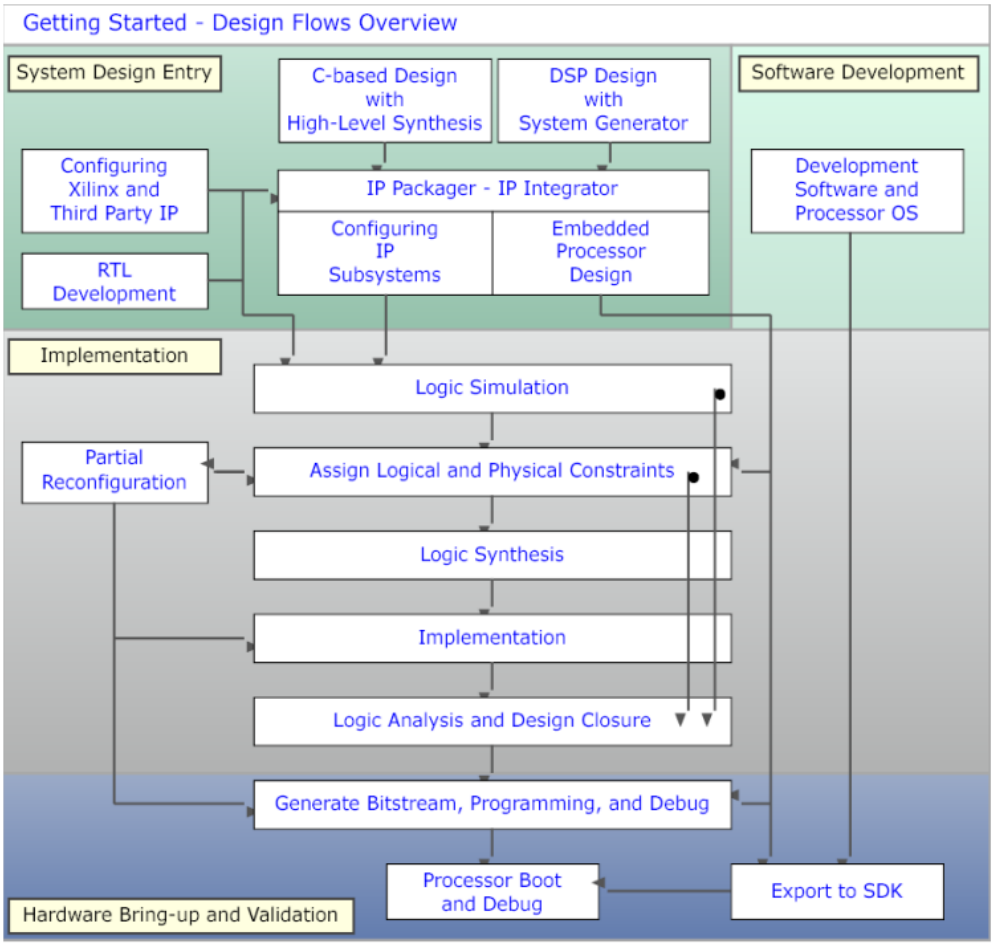


FIGURE 3.1: Vivado Design Suite design flow. Source: [21]

## 3.2 Vitis

### 3.2.1 Vitis Tools

#### Vitis Unified Software Platform

The Vitis Unified Software Platform is a software that integrates all features of Xilinx hardware and software development into one unified environment using C/C++ for both hardware and software components [22].

More specific, by using the Vitis unified software platform, we get access to a development environment for heterogeneous applications. In the aforementioned environment, heterogeneous systems include software applications running on x86 host processors or Arm embedded processors, compute kernels running in

programmable-logic (PL) regions or Versal AI Engine arrays, and extensible platform designs that provide the foundation for building and running the heterogeneous systems.

The Vitis unified software platform consists of the following elements:

- The software development tool stack, such as compilers and cross-compilers to build your software application.
- Debuggers that gives the capability to locate and fix any problems in system design.
- Program analyzers that offer profiling and analysis of the application's performance.
- Xilinx Runtime (XRT) that provides an API and drivers for your software program, in order to make the connection with the target platform possible, as well as handle transactions and data transfers between the software application and the hardware design.
- Vitis accelerated libraries that provide performance-optimized hardware functions with minimal code changes, and with the lack of need to re-implement your algorithms, so that they can reap the benefits of Xilinx adaptive computing. Vitis accelerated libraries are available for common functions of math, statistics, linear algebra and DSP, as well as for domain specific applications, like vision and image processing, quantitative finance, database, data analytics, and data compression.

The Vitis tools provide compilation, linking, profiling and debug capabilities for heterogeneous systems in a number of different design flows including Data Center application acceleration, RTL kernel design, Embedded System design, and traditional embedded hardware and software design.

### 3.2.2 Vitis Design Flow

Below is an overview of the design flow to create an accelerator using the Vitis Unified Software Platform:

*Project Initialization:* Create a new project in the Vitis IDE or Vitis command-line tools and specify the project details, such as the project name, target platform, and hardware platform (FPGA, ACAP, or SoC).

*Hardware Platform Specification:* Define the hardware platform details, including the target device, system configuration, memory interfaces, and connectivity. Select the appropriate platform description file (XSA or XPFM) provided by the hardware vendor.

*Application Development:* Write or import your host application code that will interact with the accelerator. This code will run on the CPU and control the accelerator's execution. You can use C, C++, OpenCL, or RTL (VHDL or Verilog) languages, depending on your requirements.

*Accelerator Design:* Design the accelerator component that will be implemented in hardware. This can be done using Vitis High-Level Synthesis (HLS) or RTL design methodologies or directly in the Vitis environment. Vitis HLS allows you to describe the functionality of the accelerator in C/C++ and automatically generates RTL code.

*Software-Hardware Interface:* Define the interface between the software and hardware components. This includes defining the data transfer mechanisms and specifying the data formats and memory locations used by the accelerator.

*Accelerator Optimization:* Optimize the accelerator design for performance and resource utilization. Utilize the HLS or RTL design tools to refine the hardware implementation. Perform optimizations such as loop unrolling, pipelining, and memory access optimizations to achieve the desired performance.

*Platform Integration:* Integrate the accelerator into the software application if you chose to develop the accelerator in Vitis HLS. Use the Vitis platform to interface with the accelerator, including data transfer, control signals, and synchronization. Vitis provides libraries and APIs to facilitate this integration.

*System Emulation:* Run system-level emulation to verify the functionality and performance of the accelerator. Use the Vitis emulation tools to simulate the entire system, including the software and hardware components. Emulation helps in identifying and fixing issues early in the development cycle.

*Accelerator Debugging:* Use the Vitis debug tools to identify and fix any issues in the accelerator or software code. Perform both host-based and target-based debugging, using techniques such as breakpoints, variable inspection, and profiling.

*System Validation:* Validate the accelerator on the target hardware platform. Deploy the application on the FPGA or ACAP, measure its performance, and

verify its functionality against the expected results. Vitis provides tools for performance measurement and verification.

*Deployment and Packaging:* Package the accelerator for distribution or deployment. Generate the necessary files and artifacts for deployment on the target hardware platform. This may include software executables, libraries, or system images.

*Performance Monitoring and Analysis:* Monitor and analyze the performance of the accelerator on the target hardware platform. Use the Vitis profiling tools to measure metrics such as execution time, memory bandwidth, and resource utilization.

By following this design flow, you can effectively create an accelerator using the Vitis Unified Software Platform. The platform offers a range of tools and utilities to streamline the development, optimization, debugging, and deployment processes.

Fig 3.2:

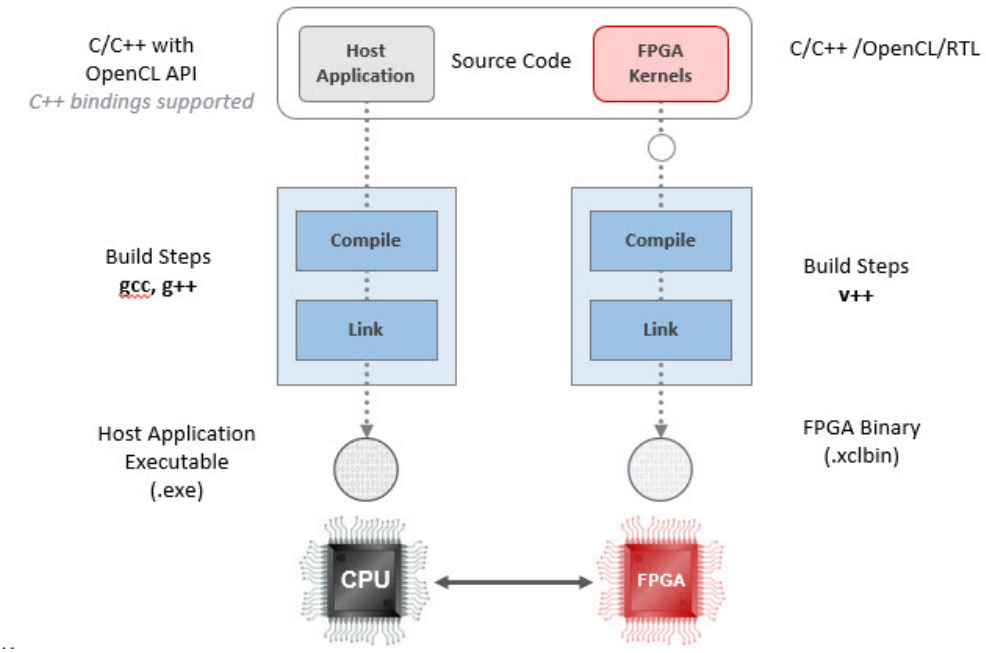


FIGURE 3.2: Vitis design flow. Source: [23]

## 3.3 Discussion

Vivado HLS, the predecessor in high-level synthesis (HLS) solutions, had its final official release in 2020.1. The subsequent iteration, Vitis HLS, represents an

advancement with notable improvements over Vivado HLS. A key enhancement as included in [24] is the utilization of an updated LLVM compiler standard, enabling support for C/C++ in compilation and simulation processes.

Hence, migrating a kernel module or IP from Vivado HLS to Vitis HLS requires a comprehensive understanding of the differences between these versions and their impacts on design. These disparities can be classified into three main categories:

**Key Behavioral Differences:** Vitis HLS introduces modifications in tool behavior that influence the overall design process, necessitating adaptation for seamless migration.

**Deprecated Commands:** Certain commands previously used in Vivado HLS are now either unsupported or discouraged in Vitis HLS, prompting users to transition to alternative approaches.

**Unsupported Features:** Vitis HLS may lack support for features that were available in Vivado HLS, requiring users to make adjustments to their designs during the migration process.

The most common category that was used in that design and it was needed to deal with was unsupported features. More specific, several features are not supported in Vitis HLS, and it is crucial to be aware of these limitations to avoid encountering issues during the compilation and synthesis processes. One notable restriction involves the use of pragmas, specifically with the HLS dependence pragma. When this pragma is employed on an argument that also features an m_axi INTERFACE pragma specifying a bundle with two or more ports, it is not supported, and the compiler may issue a warning or error.

Moreover, there have been changes in pragma support, specifically regarding the ap_bus mode, which is no longer supported. Developers are advised to use the m_axi interface instead. Also, certain directives and pragmas, such as "RESOURCE," are considered deprecated or unsupported. To address this, it is essential to eliminate or adjust them, following the guidelines outlined in [25] specific tutorial. Specifically, the outdated "RESOURCE" directive/pragma in Vivado HLS has been substituted with "BIND_OP" and "BIND_STORAGE" pragmas and directives in Vitis HLS. The recommended practice involves utilizing the "INTERFACE" pragma or directive along with the "storage_type" option to specify arguments for the top function. This migration guarantees compatibility and conforms to the updated syntax and features introduced in

Vitis HLS It's worth noting that the maximum width for C++ arbitrary precision types in Vitis HLS is 4096 bits, a reduction from the 32K bits supported by Vivado HLS. These limitations underscore the importance of revising code and adapting to the supported features in Vitis HLS to ensure a smooth and successful compilation process.

# Chapter 4

# Methodology to Transform Vivado Designs to Vitis Designs

This chapter describes in detail the main design which was developed in this Thesis. As mentioned in previous chapters the main goal of this thesis was to adapt the architecture in state-of-art tools and a larger FPGA in order to enhance the design by utilizing effectively the given resources.

The first part of this thesis concern the modifications which was needed to carry out in order to adapt the design in newest versions of tools. Particularly, the accelerator was developed in Xilinx ZCU102 and QFDB platforms from George Pitsis and Charisis Loukas in Xilinx Vivado Tools 2017.1 and 2017.2 versions. In this thesis it was adopted in Xilinx Vivado 2019.2. A detailed description of tools and platforms are presented in chapter 3. To be accomplished the adaptation there was not needed to proceed in crucial modifications.

The main and crucial section of this thesis regarding the implementation of architecture in Alveo Accelerator Card. As already is mentioned the card can be used with Xilinx Vitis Unified Software Platform, hence the adjustment to the Vitis platform was necessary.

The company offers numerous tutorials, such as the one on designing for the same platform and the migration from SDK tool to the Vitis platform. Specifically, tutorial [26] focuses on updating hardware using the Vivado Design Suite and importing SDK code into the Vitis platform. However, tutorial [27] reveals that in the transition to the Vitis environment, certain tasks, typically managed implicitly by the compiler and runtime, must now be explicitly handled by the application developer. Consequently, turorial [27] emphasizes the necessity of creating the design within the Vitis Unified Software Platform from the outset, rather than relying solely on updating hardware and importing SDK code.

## 4.1 Structure of Design

Besides focusing on the specific neural network, we have presented the structure of the host code as well as the kernel code, both coded in a high-level language, C/C++, and we have also visualized how the language for heterogeneous systems, OpenCL, is used and integrated.

Provided is a summary of the architectural design 4.1, as depicted below:

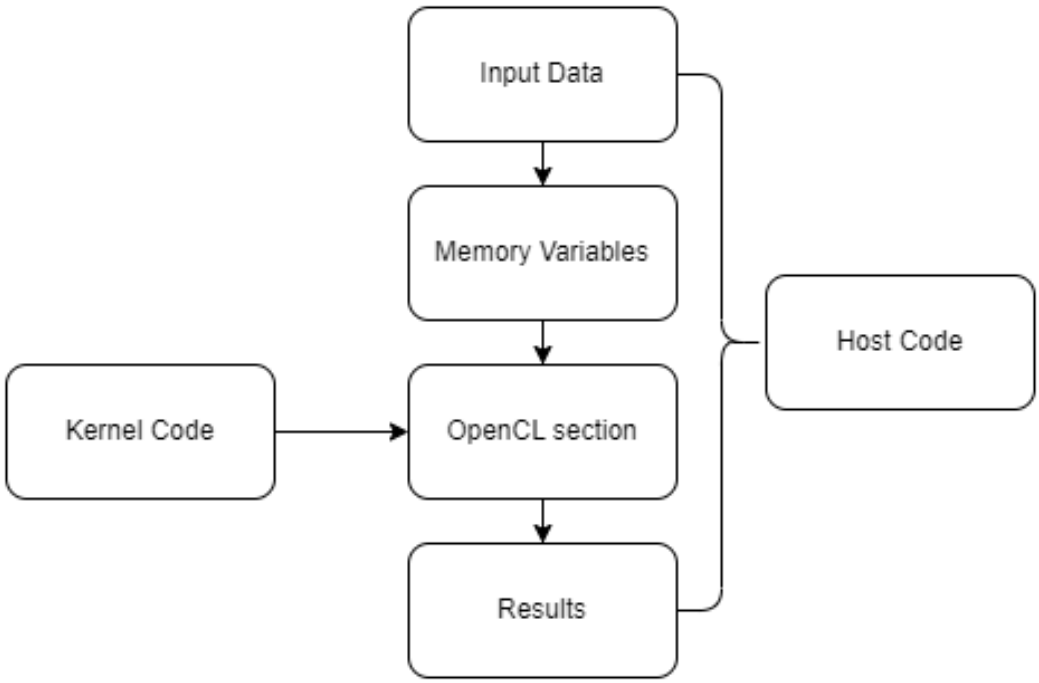


FIGURE 4.1: Architecture Design

1. Input data. In this stage, the input reads through the images, bias weights that is required for neural network.
2. Memory variables. The reservation of memory for variables in this type of design is relevant because, otherwise, the system derives from out-of-range dimension errors.
3. OpenCL section. This section prepares the data and calls the kernel, or kernels as the case may be, that runs on the FPGA. It consists of the following steps:
   (a) Detect the accelerator device. Since this methodology is compatible with multiple Xilinx platforms, the first step is to detect how many accelerator devices are available, their information (name and supplier) and which one will be used.

(b) Create and configure the context. This step initializes the application and defines the memory it needs, which will be used to run kernels on the platform or to transfer data.

(c) Create the input/output buffers. The buffers are memory spaces of a determined size, designed to be available to both the host and the kernel (platform).

(d) Configure kernel arguments.

(e) Configure the data transfer from the input buffers to the device.

(f) Call the kernel to be executed.

(g) Configure the data transfer from the output buffers to the host.

(h) Wait for the kernel to finish its task. When that happens, you will get the result of the hardware acceleration.

4. Results. Print the results from the kernel and valuate them.

## 4.2 Host Code

In the Vitis Unified Software Platform, the host code refers to the code that runs on the CPU of the system and controls the execution of the programmable logic in the FPGA. The host code is typically written in a high-level programming language such as C++ or Python and communicates with the FPGA through an API provided by the Vitis runtime libraries. In this thesis, the host code is written in C++ programming language.

The host code plays a vital role in managing and controlling the interaction between the software running on the host processor and the hardware accelerator implemented on the FPGA and the adaptive SoC device. This interaction is facilitated through the use of **OpenCL buffers**, which enable efficient data transfer and communication between the host and the accelerator as mentioned before.

### 4.2.1 CL buffers

CL buffers in Vitis refer to OpenCL buffers that are used for passing data between the host CPU and an FPGA accelerator. The usage of CL buffers in Vitis involves the following steps:

- Allocate memory for the buffer on the host: The host CPU allocates memory for the buffer that will be used to store data to be passed to the accelerator.
- Create a CL buffer object: The host CPU creates a CL buffer object and specifies the size of the buffer and the memory location of the buffer on the host.
- Copy data to the CL buffer: The host CPU copies data to the CL buffer using OpenCL APIs.
- Copy the results back to the host: The host CPU retrieves the results from the FPGA accelerator by copying the data from the result CL buffer back to the host memory using OpenCL APIs.
- Pass the CL buffer to the FPGA accelerator: The host CPU passes the CL buffer object to the FPGA accelerator, which can then access the data in the buffer.
- Process the data on the FPGA accelerator: The FPGA accelerator processes the data in the CL buffer and stores the results in a different CL buffer.
- Release the CL buffers: Once the processing is complete, the host CPU releases the memory used by the CL buffers.

The usage of CL buffers in Vitis enables efficient data transfer between the host CPU and FPGA accelerators and can significantly improve application performance.

Overall, cl buffers are an essential part of Vitis programming, as they provide a way to manage memory resources on FPGA accelerators efficiently. By using cl buffers, we can develop FPGA applications that take full advantage of the device's parallel processing capabilities while minimizing data transfer overhead.

**OpenCL buffers** serve as containers for data that can be shared between the host and the accelerator as seen in the figure 4.2. The host code allocates these buffers in the host memory, and they are used to store the input data that will be processed by the accelerator, as well as the output data generated by the accelerator. The buffers provide a unified and standardized interface for accessing and manipulating data, regardless of whether it resides in the host or the device memory.

To utilize OpenCL buffers effectively, the host code needs to establish a context for the accelerator device. The context encapsulates the resources and state necessary for the execution of OpenCL operations. It enables the host code to allocate and manage memory resources for the buffers and create the necessary command queues for issuing commands to the accelerator.

Once the context is established, the host code can create OpenCL buffers using the allocated memory resources. These buffers can be configured to hold input, output and intermediate data for the accelerator. The host code is responsible for initializing the input buffer with the required data and specifying the size and layout of the buffers.

Data transfer between the host and the accelerator is a key aspect of the host code's role. The host code uses OpenCL commands, such as clEnqueueWrite-Buffer and clEnqueueReadBuffer, to transfer data between the host and the device memory. These commands ensure the correct synchronization and memory consistency between the host and the accelerator, allowing for seamless data movement.

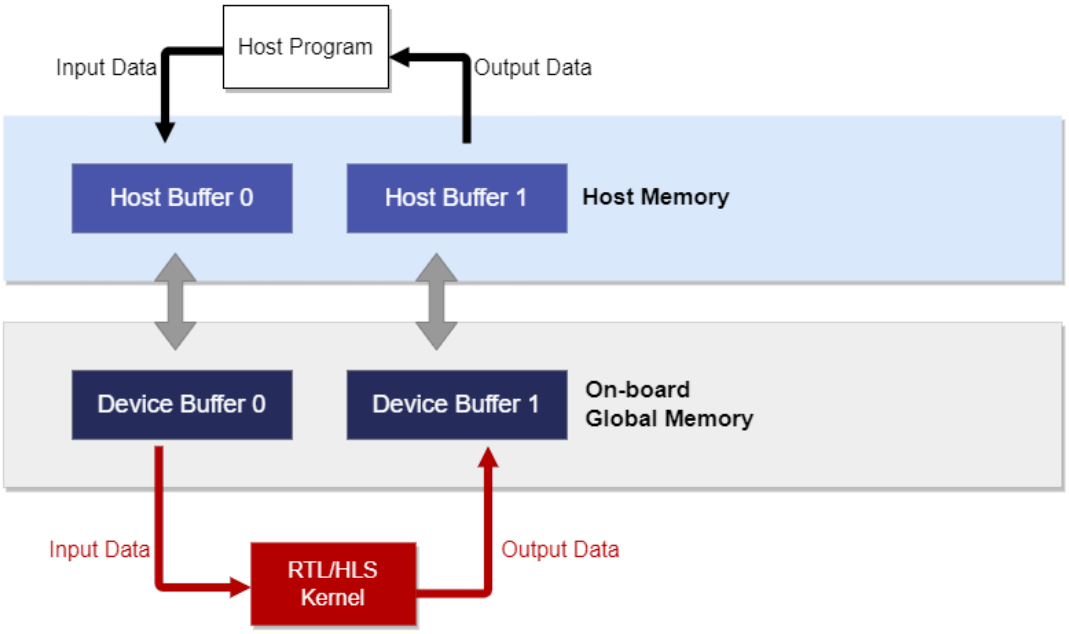


FIGURE 4.2: Data movement through CL buffers

After we have set aside the memory buffers and filled them with initial test data, the subsequent step to accelerate the process is to transfer these buffers to the FPGA global memory. This is achieved by establishing OpenCL buffer objects with the **CL_MEM_USE_HOST_PTR** flag, indicating to the API that we're supplying our own pointers instead of having it allocate its own buffers.

However, it's important to note that if we have not carefully managed the allocation of these pointers, it could negatively impact our system's performance.

The code snippet below demonstrates how are associated allocated buffers with OpenCL buffer objects.

**Algorithm 1** Allocate Buffers in Global Memory

$$
\begin{aligned}
&OCL\_CHECK(err, cl :: Bufferbuffer\_bias1(context,\\
&\qquad CL\_MEM\_USE\_HOST\_PTR|CL\_MEM\_READ\_ONLY,\\
&\qquad (BUFSIZE) * sizeof(float), bias\_1.data(), \&err));\\
&OCL\_CHECK(err, cl :: Bufferbuffer\_bias2(context,\\
&\qquad CL\_MEM\_USE\_HOST\_PTR|CL\_MEM\_READ\_ONLY,\\
&\qquad (BUFSIZE) * sizeof(float), bias\_2.data(), \&err));\\
&OCL\_CHECK(err, cl :: Bufferbuffer\_bias3(context,\\
&\qquad CL\_MEM\_USE\_HOST\_PTR|CL\_MEM\_READ\_ONLY,\\
&\qquad (BUFSIZE) * sizeof(float), bias\_3.data(), \&err));\\
&\qquad\qquad .\\
&\qquad\qquad .\\
&\qquad\qquad .\\
&OCL\_CHECK(err, cl :: Bufferbuffer\_results(context,\\
&\qquad CL\_MEM\_USE\_HOST\_PTR|CL\_MEM\_WRITE\_ONLY,\\
&\qquad (BUFSIZE) * sizeof(uint32\_t), results.data(), \&err));
\end{aligned}
$$

In this point to emphasize the fact that data which buffers are going to store and transfer is necessary to be aligned to 4 KiB boundaries as Alveo cards, are designed to work with the Alveo DMA (Direct Memory Access) engine. The Alveo DMA engine requires data to be aligned to 4 KiB boundaries for efficient data transfer and processing.

Once the buffers are created, the host code using the **enqueueMigrateMemObjects** function pass the data to device global memory. enqueueMigrateMemObjects function is typically used to move data between different memory regions in OpenCL, between the host and device memory. As we can see in Algorithm 2 below, the enqueueMigrateMemObjects take a second argument the number 0, that means that the input data are transmitted from the host toward device memory.

Then, the host code enqueues a command to launch the execution on the accelerator. This is achieved using the **enqueueTask**, which specifies the dimensions and workgroup sizes for executing the kernel function. The host code can also

provide synchronization points using OpenCL events to ensure that the execution on the accelerator is properly sequenced and coordinated with other operations.

After the execution is complete, the host code can retrieve the output data from the output buffer using **enqueueMigrateMemObjects**. The OpenCL buffer abstraction simplifies the retrieval of output data, allowing the host code to seamlessly access the results generated by the accelerator.

**Algorithm 2** Manage Data Movement using OpenCL

```
//Copy input data to device global memory
OCL_CHECK(err, err = q.enqueueMigrateMemObjects(all_input_buffers, 0));

//Launch the Kernel
OCL_CHECK(err, err = q.enqueueTask(kernel));

//Copy Result from Device Global Memory to Host Local Memory
OCL_CHECK(err, err = q.enqueueMigrateMemObjects(buffer_results,
CL_MIGRATE_MEM_OBJECT_HOST));
```

In summary, the host code in the Vitis Unified Software Platform leverages OpenCL buffers to manage the interaction between the host processor and the FPGA or adaptive SoC accelerator. It establishes a context for the accelerator, creates and configures the OpenCL buffers, transfers data between the host and the device memory, launches the execution on the accelerator, and retrieves the output data. The use of OpenCL buffers streamlines the data transfer and communication process, enabling efficient and seamless hardware acceleration within the Vitis Unified Software Platform.

## 4.3 Kernel Code

As described above, the Vitis Unified Software Platform comprises of two main components, the kernel and the host program. The kernel program is the design aspect for which the Vitis HLS accelerator was developed in the present thesis. In Vivado HLS Tool there are three IPs. First of all, there is the Conv Accel IP that is implementing the Convolutional Layer. Secondly, there is the FC Accel IP that is the Fully Connected Layer and finally there is the Find Max IP.

The accent of this thesis is to integrate that accelerators on Vitis Unified Software Platform. Thus, this thesis required the consolidation of these three different IPs into a single entity and at first the three IPs are considered as black boxes in order to be focused to the interface. Therefore, it was necessary to modify the method by which data is transmitted between IPs.

### 4.3.1 Streaming data

The three IPs, accept hls::streams of input data and return the results in hls::streams also. Convolutional layers accepts an input stream that include bias, kernel and the image. Then read that stream, store the bias and kernel in BRAM and execute the convolution. Convolution is composed of 5 functions. Convolution1, 2x Shifted Fifo , 2x Convolution2 and Packed Fifo. There is use of HLS dataflow as the Conv Accel was the top function of IP. Consequently, the FC Accel is the Fully Connected layer, that accepts the data streamed by 128bits and return a stream of 32bits. In that accelerator FC Accel is used 2 instances. Finally there is the Find Max IP that accepts the 2 hls::stream from FC Accel IPs and returns a stream with the position and the value of maximum element of these two streams.

To combine these three IPs into a single one, a new top-level function was created Figure 4.3, incorporating the functionalities from all Vivado HLS IPs. The main goal was to consolidate all three IPs by integrating their top-level functions. In the kernel's top function, the biases and kernels that are used in convolution are stored in BRAM in order to can be executed under dataflow and can accessed in parallel. Images data are passed directly into Convolution layer, where the convolution operation is performed. Additionally, FC Accel and Find Max functions are added. At the same way, data is passed directly to the FC Accel components. All of these operations, except the read and store data in BRAM, are organized under an HLS dataflow 4.5 to enable parallel processing.

Once the kernel has finished executing, the OpenCL runtime can copy the results from the buffer back to host memory for further processing.

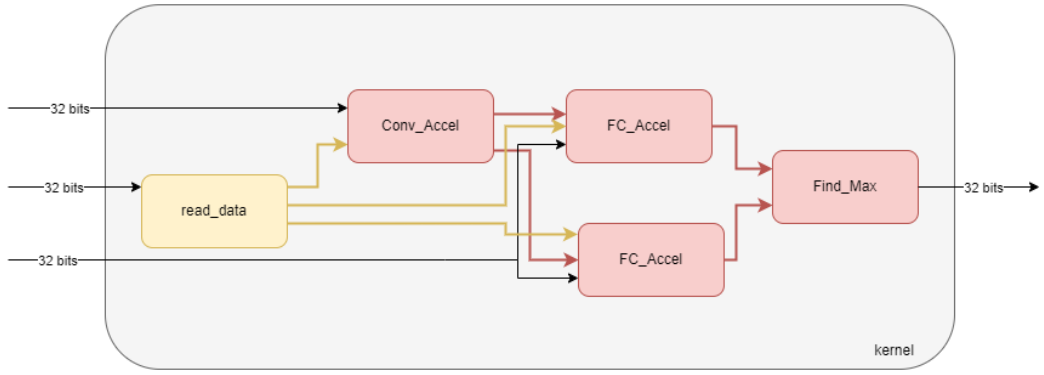


FIGURE 4.3: Block Diagram of kernel - input 32bits

### HLS DATAFLOW

It is crucial to note that we incorporate the call of all functions utilized in the top functions of each Vivado HLS into the kernel's top function. By employing HLS DATAFLOW for all functions, the dataflow configuration aligns with the illustrated Figure 4.5. This configuration clearly demonstrates that the design is capable of concurrently executing functions. As soon as the Convolutional layer produces results, triggering the PackedFifo function, the execution of the fully connected layer commences. Subsequently, when it yields its results, the Find Max function initiates its execution.

This approach ensures a parallel execution of functions, enhancing the efficiency of the overall design. As the Convolutional layer progresses, other layers seamlessly follow suit, creating a streamlined and parallelized dataflow that optimizes the processing capabilities of the design. The orchestration of functions in this manner promotes a more efficient and parallelized execution of tasks within kernel.


FIGURE 4.4: Block Diagram of kernel without dataflow

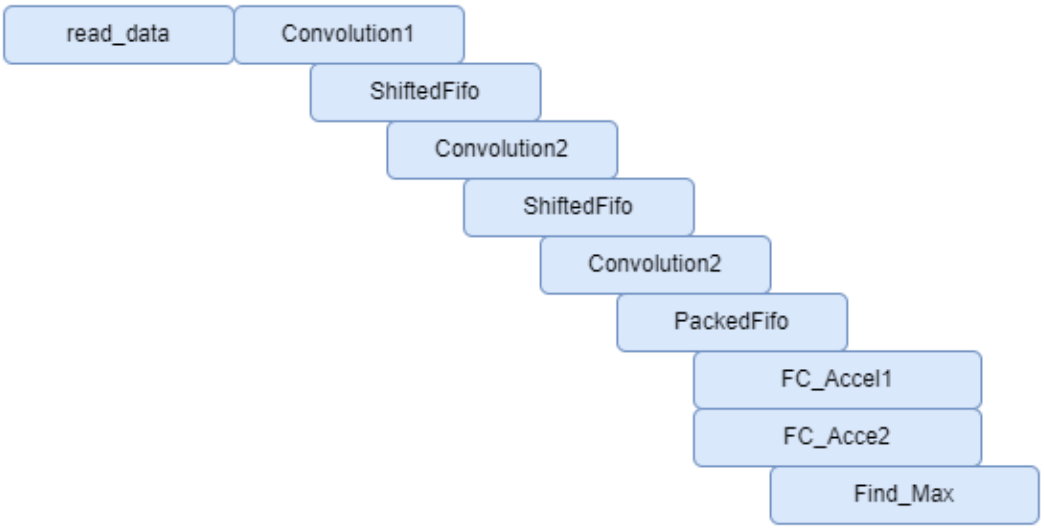


FIGURE 4.5: Block Diagram of kernel with dataflow

### 4.3.2 Changes in Kernel

The Xilinx Vitis development environment is designed to enable hardware acceleration on a variety of Xilinx devices, including both FPGAs and adaptable SoCs. When migrating a design from one target device to another, such as moving from the ZCU102 evaluation board to the Alveo U50 accelerator card, it is common to encounter design-specific issues and differences between the devices.

While attempting to adapt the code, II violations occurred as a consequence. **"II violations"** typically refer to initiation interval violations, which occur when the loop iterations in kernel cannot be fully unrolled or pipelined to meet the target device's timing requirements. The initiation interval represents the minimum number of clock cycles between successive loop iterations. Violating the initiation interval constraint can result in a suboptimal or non-functioning design.

When migrating from the ZCU102 to the Alveo U50, several factors can contribute to II violations:

1. Clock frequency: The Alveo U50 have a different maximum clock frequency compared to the ZCU102. If your design relies on a specific clock frequency, as in our case, that cannot be met by the new device, it may lead to II violations.

2. Resource limitations: The Alveo U50 has a different FPGA architecture and resource availability compared to the ZCU102. If your design uses

resources that are scarce or not available on the U50, it may require alternative implementations that introduce II violations.

3. Memory access patterns: The memory hierarchy and bandwidth characteristics differ between the ZCU102 and Alveo U50. If your design heavily relies on specific memory access patterns that are not well-suited for the U50, it may introduce II violations due to memory contention or suboptimal memory access.

The Alveo U50 has a higher clock frequency compared to the ZCU102. More specific, the design was implemented in ZCU102 with clock set at 150MHZ and Alveo U50 run at 300MHz. Consequently, tasks or procedures can be executed in fewer clock cycles. Therefore, it becomes necessary to modify certain sections of the code in order to perform these procedures efficiently and complete them before the next procedure requires their results.

To provide a scientific explanation, clock frequency refers to the speed at which a computer's processor operates. It is measured in cycles per second, or Hertz (Hz). A higher clock frequency means that the processor can perform more instructions or tasks in a given amount of time.

In this context, the Alveo U50, being equipped with a higher clock frequency than the ZCU102, can execute operations at a faster rate. This can potentially result in shorter execution times for procedures. Consequently, to take advantage of this increased processing speed, it may be necessary to optimize the code by rewriting specific sections to ensure that procedures are completed in a timely manner before their results are needed for subsequent tasks. By doing so, the overall efficiency and performance of the system can be improved.

A typically II Violation that we were called to face is located in Find Max function and is presented below:

**Algorithm 3** Original Kernel code

**if** $float1 > float2$ **then**
  $tempMax \leftarrow float1$
  $tempPos \leftarrow i$
**else**
  $tempMax \leftarrow float2$
  $tempPos \leftarrow i + DIM1$
**if** $tempMax > max$ **then**
  $max \leftarrow tempMax$
  $position \leftarrow tempPos$

In this scenario, the calculation of "res1" will begin in the first clock cycle, and in the subsequent clock cycle, it will be checked in the "if" statement. However, since the "sub" operation is not completed in the second clock cycle, it cannot be compared as required. This leads to a violation known as II (Initiation Interval) violation, indicating that there is an issue with the timing of the operations. Therefore, it is necessary to address this problem and make the appropriate corrections to ensure proper execution.

As result, the part of code must be rewrited in order to remove II violation as:

**Algorithm 4** Amended Kernel code

$res1 \leftarrow float1 - float2$
**if** $res1 > 0$ **then**
  $tempMax \leftarrow float1$
  $tempPos \leftarrow 4 * i + j$
**else**
  $tempMax \leftarrow float2$
  $tempPos \leftarrow 4 * i + j + DIM1$
**if** $tempMax > max$ **then**
  $max \leftarrow tempMax$
  $position \leftarrow tempPos$

## 4.4 Optimization

To harness the capabilities of the Alveo Accelerator Card's resources effectively, certain actions were taken. These cards are specifically engineered to deliver

high-performance computing across diverse applications. As part of this optimization, the Alveo card enables the reading of 512 bits of data, thereby minimizing memory access and consequently improving overall performance. Achieving this required modifications to the kernel code.

### 4.4.1 Wide Memory Access

In Vitis, wide memory access refers to accessing memory elements in a wide fashion, typically using vectorized or burst access methods. It allows for processing multiple data elements simultaneously, which can significantly improve memory bandwidth utilization and overall performance. To achieve wide memory access, a modification was made to the input reading mechanism in the kernel. Some code was implemented while every function read inputs, which reads inputs in chunks of 512 bits and then converts them into 32-bit elements, as accelerator specifically operates on 32-bit elements.

**Algorithm 5** Wide Memory Access

```
BUFFER_SIZE = 128
VECTOR_SIZE = 512 / 32 = 16
procedure FUNC(uint512_t *in, ..., size)
    uint512_t v1_local[BUFFER_SIZE];
    size_in16 ← (size − 1)/VECTOR_SIZE + 1
    for size_in16 do
        chunk_size ← BUFFER_SIZE
        for chunk_size do
            v1_local[j] ← in[i + j]
        for chunk_size do
            uint512_ttmpV1 ← v1_local[j]
            uint32_t val1
            for VECTOR_SIZE do
                val1 ← tmpV1.range(DATATYPE_SIZE ∗ (i + 1) − 1, i ∗
DATATYPE_SIZE)
                // your code using val1
```

The arguments of that function 5 are a type unsigned int 512 bits pointer, and the size of elements that we desire to read. Before start, it is necessary to define **BUFFER_SIZE** and **VECTOR_SIZE** that represent the elements that can be read with one memory access. As mentioned with one memory access

can be read 512bits from the memory. In that case, the data type that accelerator handled is unsigned 32 bits. Consequently, 512bits per access / 32 bits every element is equal to 16. Hence, with one memory access can be read 16 elements. Let's analyze the algorithm of function. First it is defined a uint512_t local buffer **v1_local**, where read data from the every access will be saved temporarily. Also, it is defined the access size, the times that it is going to access memory **size_in16**, in order to read the desired size of elements. The times of access memory calculating by the size of elements that is needed to read from memory divided by the vector size, in that case **size_in16=(size - 1)/VECTOR_SIZE + 1**. For example, if the elements that are desired to read from memory are 1000, the size_in16 will be 1000 divided with 16, elements that can be read in one access, size_in16=1000/16=62.5 +1 = 63.5. Thus, with 63 memory accesses can be read 1000 elements. Finally, ranging data that are saved in uint512_t local buffer v1_local passed into uint32_t output. The line **val1 = tmpV1.range(DATATYPE_SIZE * (i + 1) - 1, i * DATATYPE_SIZE);** uses the **range()** method on tmpV1 to extract a subrange of 32 bits from temporary **tmpV1**, where is stored 512 bits of one access. The starting position of the subrange is calculated as DATATYPE_SIZE * (i + 1) - 1, and the ending position is i * DATATYPE_SIZE. The extracted subrange of 32 bits is then stored in the **val1** variable, and finally, val1 can be used.

The block diagram that occurs by implementing that wide memory access is depicted below:

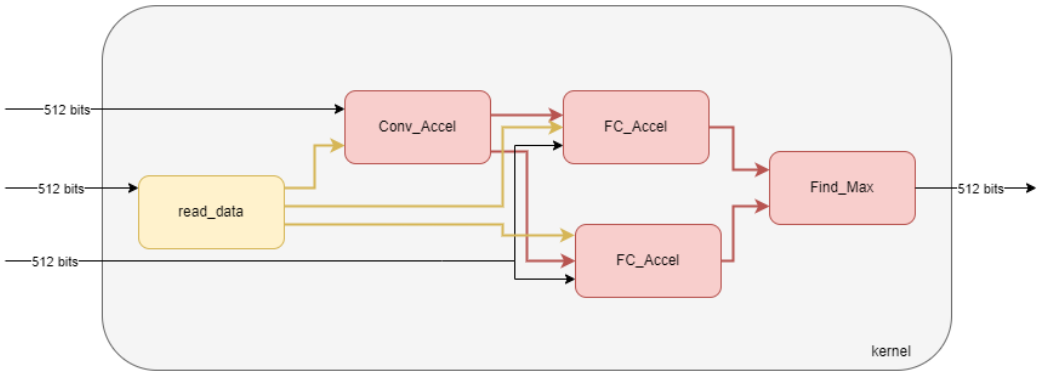


FIGURE 4.6: Block Diagram of kernel - input 512bits

### 4.4.2 Multiple Compute Units

Multiple compute units (MCUs) are a key feature of modern high-performance computing systems, including FPGA-based accelerators such as the Alveo U50.

MCUs enable parallel execution of compute-intensive tasks, which can significantly improve performance compared to a single compute unit. In this section, we will explore the concept of MCUs and how they can be implemented using the Vitis Unified Software Platform.

Each individual instance of a kernel is often referred to as a compute unit (CU). Increasing the number of CUs enhances parallelism within a host-kernel system. This means that a host program can execute the same kernel multiple times with various datasets. In such scenarios, it is beneficial to generate multiple CUs of the kernel to enable concurrent execution, which significantly enhances the overall system's performance. This approach allows the host program to fully leverage the available computing resources, improving the efficiency of data processing and computation. Essentially, by increasing the number of CUs, you're optimizing the system to perform tasks more swiftly and efficiently by parallelizing the workload across multiple compute units, thereby taking full advantage of modern multi-core processors and GPUs for enhanced computational performance.

Using Multiple Compute Units it was needed to modify the host code of application. More specific, it is defined the number of units in which the application will be executed. Practically, compute units represent instances of the kernel that calculate the total result. Using a single compute unit, the kernel accept as input 2500 images and process all images them own. Using more compute units divided the number of input images in compute units. Firstly, it was used 2 compute units, that means that each compute unit accepts and process 1250 images. The key of multiple compute units is that gives the opportunity to execute them concurrently. In effect, the time that it is needed to execute the application is decreased in half. Later we increased the number of compute units in 4. Practically, each compute unit accepts and process 2500/4=650 images. That means 650 images are processing concurrently instead of 2500 that was calculated by one compute unit. In that point, it is necessary to mention that the Alveo Data Center Cards give us the advantage of use multiple units with many resources that they provide. More details about the Alveo u50 which is used in this thesis provided in chapter 5.

**Algorithm 6** Multiple Compute Units

$num\_cu \leftarrow 4$
$chunk\_size\_image \leftarrow images/num\_cu$
$chunk_size\_results \leftarrow results/num\_cu$
$std :: vector < cl :: Buffer > buffer\_result(num\_cu)$
$std :: vector < cl :: Buffer > buffer\_input(num\_cu)$
**for** num_cu **do**
    $buffer\_input[i] = cl :: Buffer(context,$
$CL\_MEM\_USE\_HOST\_PTR|CL\_MEM\_READ\_ONLY, chunk\_size\_image *$
$sizeof(float), input.data() + i * chunk\_size\_image, \&err)$
    $buffer\_result[i] = cl :: Buffer(context,$
$CL\_MEM\_USE\_HOST\_PTR|CL\_MEM\_WRITE\_ONLY, chunk\_size\_result *$
$sizeof(uint32\_t), results.data() + i * chunk\_size\_result, \&err)$
**for** num_cu **do**
    // Setting kernel arguments
    $kernel[i].setArg(\#arg, buffer\_bias)$
                .
                .
                .
    $kernel[i].setArg(\#arg, buffer\_result[i])$
    $kernel[i].setArg(\#arg, buffer\_result[i])$
**for** num_cu **do**
    // Launch the kernel
    $q.enqueueTask(kernel[i])$
$q.finish()$
**for** num_cu **do**
    $q.enqueueMigrateMemObjects(buffer\_result[i],$
$CL\_MIGRATE\_MEM\_OBJECT\_HOST)$

The implementation of multiple compute units in the code does not deviate significantly from the previous version. Critical differences lie in how memory is allocated in host local memory and the execution of compute units. The code makes use of multiple compute units (num_cu) to concurrently execute the same kernel, thereby enhancing overall throughput.

Initially, the buffers used by the convolutional neural network for data remain unchanged, as they are essential for execution. The only necessary modifications involve the buffers storing the input images and the results of the convolutional

neural network. Specifically, the input data is divided among the employed compute units, requiring the allocation of one additional buffer for each compute unit dedicated to images and another for results. This results in an unchanged memory allocation process, with the size of allocated buffers now divided by the number of compute units.

**Data Flow and Data Interface**

One important consideration when implementing MCUs in the Vitis Unified Software Platform is the data flow and data interface between the different compute units. In order to achieve maximum performance, it is important to carefully design the data flow and data interface to minimize data movement between the different compute units.

Alveo Cards offer 32 memory ports for data processing. In our initial approach, we distributed the data across multiple ports to maximize parallel data processing performance. To be more precise, we allocated 13 ports for each individual compute unit. When utilizing 2 compute units, all 26 ports are fully utilized, leaving no additional ports for adding more compute units. To accommodate additional compute units, it becomes essential to merge input data into buffers, thereby reducing the consumption of memory ports. The code below demonstrates that we consolidate all the bias and kernel data for the CNN into a single buffer. Similarly, for the dense data, which is quite extensive, we use another buffer. We also maintain separate buffers for images and results because these will be divided to be processed by different computing units.

**Algorithm 7** Kernel Interface MCU

```
procedure KERNEL(inputs, images, data1, data2, results)
    #pragma HLS INTERFACE m_axi port=inputs bundle=gmem0
    #pragma HLS INTERFACE m_axi port=images bundle=gmem1
    #pragma HLS INTERFACE m_axi port=data1 bundle=gmem2
    #pragma HLS INTERFACE m_axi port=data2 bundle=gmem3
    #pragma HLS INTERFACE s_axi port=results bundle=gmem4
```

This approach allows each individual compute unit to utilize 5 memory ports.

In order to optimize the data flow and data interface, it is important to have a deep understanding of the characteristics of the input data and the computation being performed. This can be achieved through careful profiling and analysis of the application, which can help identify bottlenecks and areas for optimization.

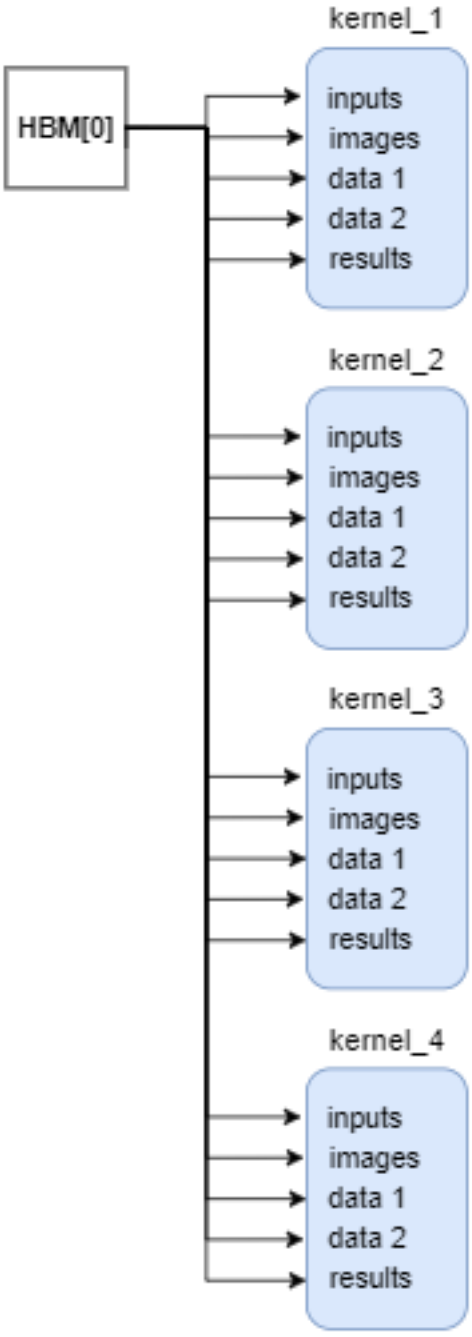


FIGURE 4.7: Platform Design - 4 Compute Units

In conclusion, multiple compute units are a key feature of modern high-performance computing systems, and are essential for achieving high levels of performance in FPGA-based accelerators such as the Alveo U50. The Vitis Unified Software Platform provides a set of APIs and tools that can be used to implement

MCUs in FPGA-based accelerators, enabling developers to achieve maximum performance through parallel execution of compute-intensive tasks.

### High Bandwidth Memory

HBM, or High Bandwidth Memory, utilizes advanced chip fabrication methods to offer increased bandwidth and efficiency per watt compared to conventional DDR implementations. In this Alveo implementation, the memory manufacturer employs stacked die and through-silicon via chip fabrication techniques to combine multiple smaller DDR-based memories into a larger, faster memory stack. The FPGA package integrates two 16-layer HBM stacks, adhering to the HBM2 specification, which are linked to the FPGA fabric through an interposer.

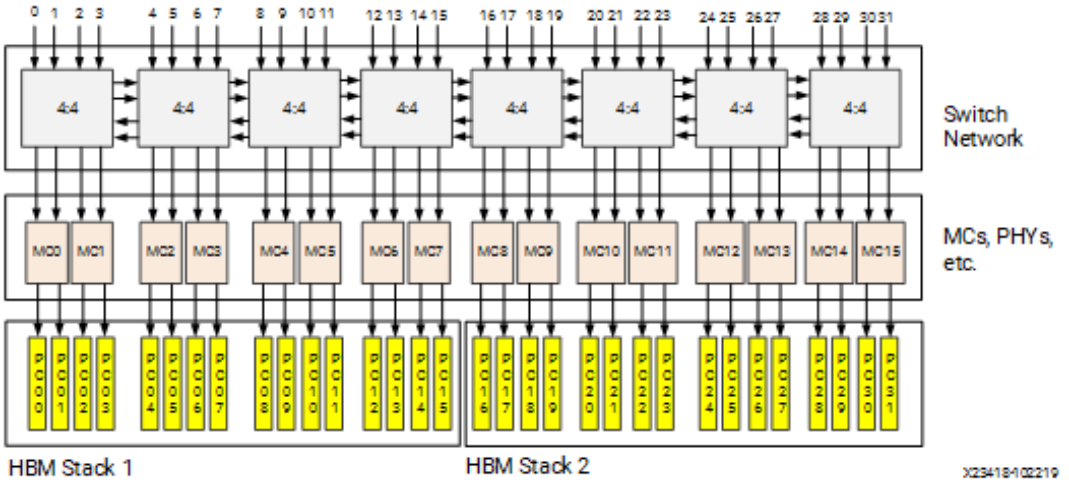


FIGURE 4.8: High-Level Diagram of Two HBM Stacks [28]

## 4.4.3 Mapping Kernel Ports to Memory

By default, when kernels are linked to the platform the memory interfaces from all the kernels are connected to a single default global memory bank, as shown in Figure 4.7. As a result, only a single compute unit (CU) can transfer data to and from the global memory bank at one time, limiting the overall performance of the application. If the device contains only one global memory bank, then this is the only option. However, if the device contains multiple global memory banks, you can customize the global memory bank connections by modifying the memory interface connection for a kernel during linking. The method for performing this is discussed in detail in Mapping Kernel Ports to Memory. Overall performance is improved by using separate memory banks for different kernels

or compute units, enabling multiple kernel memory interfaces to concurrently read and write data.

The Vitis compiler can automatically connect compute units (CU) to global memory resources, but it's also possible to manually designate which global memory bank each kernel argument or interface should connect to, using a configuration file 8. It's crucial to properly configure the kernel to memory connectivity in order to optimize data transfers, maximize bandwidth, and improve overall application performance. Even if there's only one compute unit in the device, performance can be improved by mapping its input and output arguments to different global memory banks, which allows for simultaneous access to input and output data. As we already mentioned, Alveo Cards provide 32 ports to memory, thus every input corresponds to a different global memory bank to achieve parallel access to memory and higher performance.

**Algorithm 8** HMB Configuration file

```
[connectivity]
sp=kernel1.inputs:HBM[0]
sp=kernel1.images:HBM[1]
sp=kernel1.data1:HBM[2]
sp=kernel1.data2:HBM[3]
sp=kernel1.results:HBM[4]
...
sp=kernel4.inputs:HBM[15]
sp=kernel4.images:HBM[16]
sp=kernel4.data1:HBM[17]
sp=kernel4.data2:HBM[18]
sp=kernel4.results:HBM[19]
```

## 4.5 Review of The Process

In the process of integrating a design from an FPGA to a larger one, several challenges were encountered and addressed. One prominent issue stemmed from the differing clock frequencies between Alveo and ZCU, leading to initiation interval (II) violations. This necessitated the rewriting of certain sections of the code to ensure compatibility and proper functionality.

Another noteworthy challenge arose from the utilization of hls::stream for communication between IPs through dataflow. The inherent depth of hls::streams was found to be insufficient in the Alveo design, where the higher clock frequency resulted in rapid data accumulation and subsequent waiting periods for data to be read. This scenario led to an unending cycle, preventing the program from concluding effectively. To overcome this, the depth of the hls stream had to be increased, providing sufficient space for data storage, thereby enabling the program to progress as intended.

In the context of design scaling, the original design incorporated multiple memory gates to facilitate parallel memory access for enhanced performance. However, adapting the design for Alveo, which offered 32 memory ports, required a redesign of the kernel interface. This involved minimizing the number of ports in memory from 11 to 5 for each kernel, a strategic decision enabling the inclusion of a larger number of design copies. Ultimately, this adjustment allowed for the creation of a design with four copies, optimizing resource utilization and overall system efficiency.

In summary, the integration process involved addressing clock frequency disparities, optimizing hls::stream depth, and reconfiguring the kernel interface for efficient scaling. These adaptations were crucial in ensuring seamless operation and performance improvement when transitioning the design to a larger FPGA platform.

# Chapter 5

# FPGA Implementation

This chapter will introduce the platforms where design has been applied, showcasing the results to comprehensively illustrate the capabilities that have been demonstrated in the field of design.

## 5.1 FPGA Platforms

The architectures implemented for the purposes of this thesis were targeted to three platforms, namely the Xilinx ZCU102, ALVEO U50 and the QFDB node prototype. In this section we will present a few basic facts about the platforms and a set of important figures to get a quantitative view of their capabilities.

### 5.1.1 Xilinx Zynq UltraScale+ MPSoC ZCU102

The ZCU102 [29], depicted in figure 5.1, is an evaluation board designed for rapid prototyping purposes. It is based on the Zynq UltraScale+ XCZU9EG-2FFVB1156E MPSoC, offering versatility for various applications. It features high-speed DDR4 SODIMM and component memory interfaces, FMC expansion ports, multi-gigabit per second serial transceivers, and a range of peripheral interfaces. Additionally, it incorporates FPGA logic, allowing users to customize their designs effectively. In terms of processing power, the board includes a Quad-core Cortex-A53 MPCore serving as an Application Processing Unit (APU), a Dual-core Arm Cortex-R5 functioning as a Real-Time Processing Unit (RPU), and an Arm Mali-400 MP2 serving as a Graphics Processing Unit (GPU).

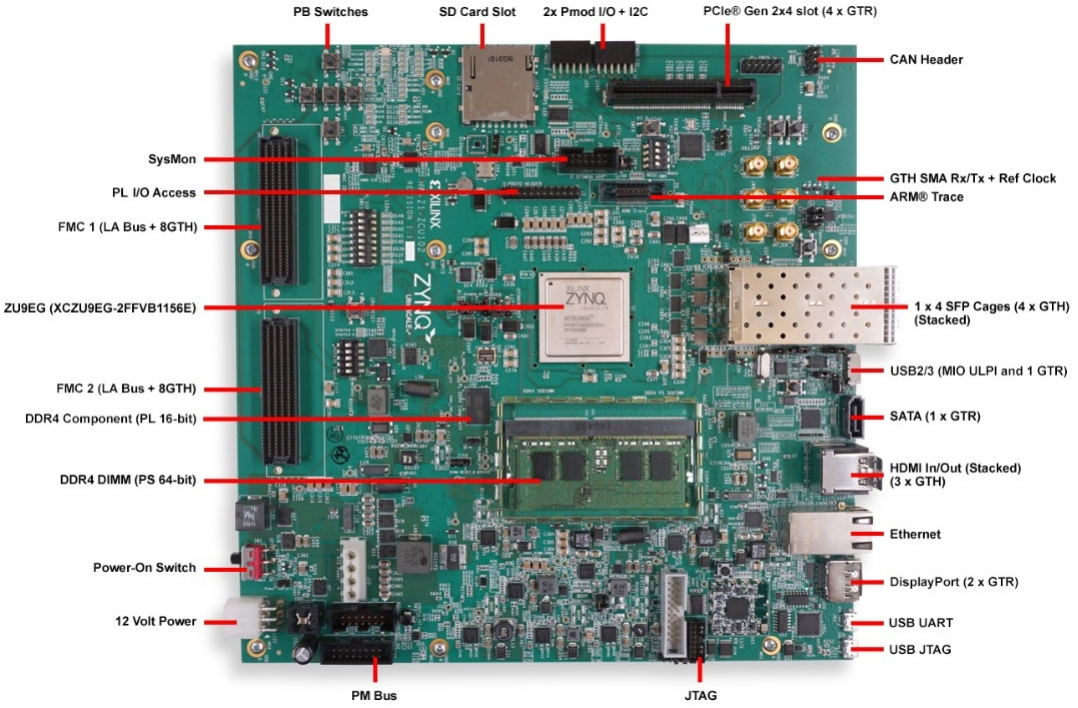


FIGURE 5.1: The ZCU102 evaluation kit. Source: [29]

### 5.1.2 Quad FPGA Daughter Board

The QFDB [30], depicted in figure 5.2, is a prototype HPC testbed for the EuroExa research project, which includes 4 Zynq UltraScale+ XCZU9EG-FFVC900-2-e MPSoCs, similar to those found in the ZCU102 board. Each FPGA node on the board is connected to a 32MB QSPI memory and a 16GB DDR4 SODIMM, allowing for data transfer rates of up to 160 Gbps and a total memory capacity of 64 Gb. The board also features a 256 GB SSD/NMVe, with larger 2 TB devices available. The FPGA nodes are interconnected in an all-to-all topology using 2 High-Speed Serial Links (HSSL) with GTH transceivers and 24 Low-voltage differential signaling (LVDS) pairs. Additionally, one of the MPSoCs is connected to the outside world using 10 HSSLs at 10.3125 Gbps and GTH transceivers. The hardware design can easily be transferred between the ZCU102 board and the QFDB due to their similar architectures.

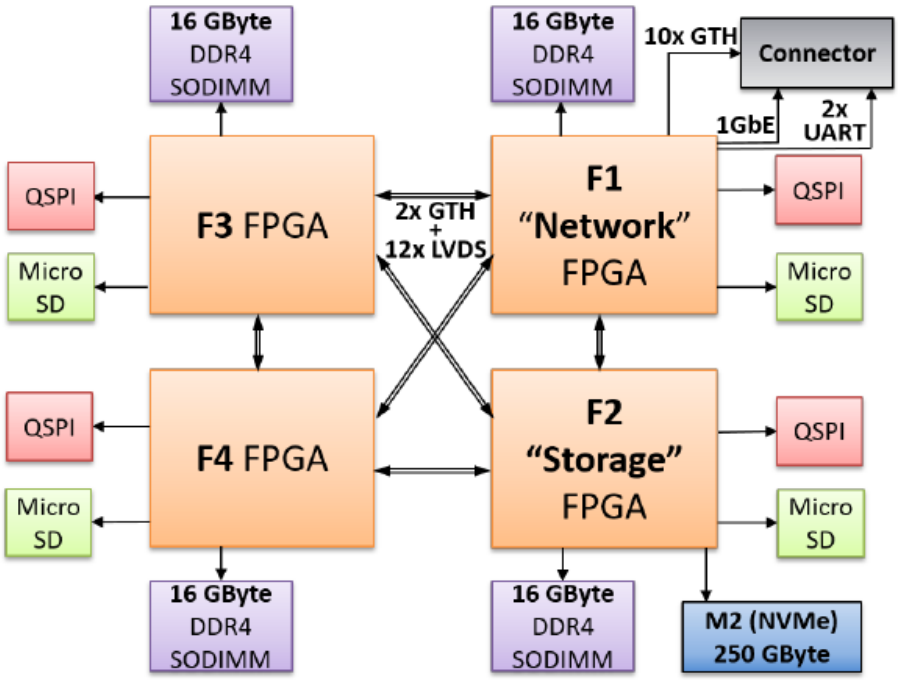


FIGURE 5.2: Block Diagram of the QFDB board: Source: [30]

Figure 5.3 illustrates the block diagram for both the ZCU102 and QFDB platforms, both of which incorporate the Zynq Ultrascale+ processor. The Zynq UltraScale+ processor utilizes the AXI4 protocol for communication and data streaming with IP cores. The AXI4 protocol is a high-speed on-chip interconnect standard that enables efficient data transfer.

By using three DMA IP cores, one for the input data and two for the weight values, the system can efficiently stream data to the different layers of the neural network. More specifically, one for the kernel and input data streamed to the IP of convolutional layers and two for the weight values to the 2 IP cores, each containing 400 nodes of the fully connected layer. The use of SmartConnect IPs allows for a more flexible connection between the DMAs and the PS. Finally, the AXI Interconnect IP core is used in the master port of the PS to improve communication and data transfer efficiency by routing data.

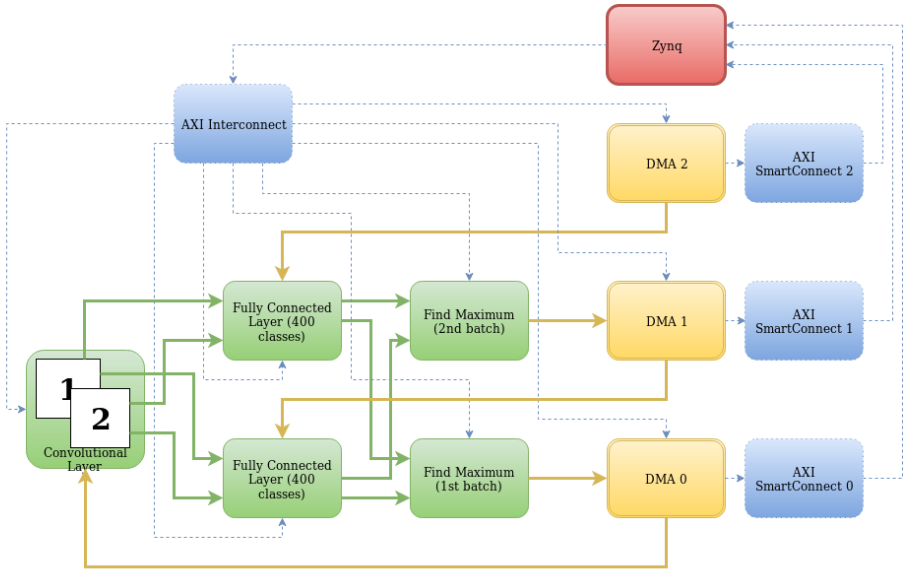


FIGURE 5.3: Zynq Architecture. Image from Loukas's Thesis [2]

### 5.1.3 AMD Virtex UltraScale+ FPGA VCU118 Evaluation Kit

The VCU118 evaluation board [31] serves as a robust hardware platform designed for the development and evaluation of designs targeting the AMD Virtex™ UltraScale+™ FPGA, specifically featuring the XCVU9P-L2FLGA2104 device. This board incorporates essential features commonly found in evaluation systems, including DDR4 and RLD3 component memory, dual small form-factor pluggable (QSFP+) connectors, a sixteen-lane PCI Express® interface, an Ethernet PHY, general-purpose I/O capabilities, two UART interfaces, and a FireFly™ Optical x4 28 G connector. Additionally, the VCU118 board is equipped with flexibility for expanding its functionality through modules compatible with the VITA-57.1 FPGA mezzanine card (FMC) and VITA-57.4 FPGA mezzanine card plus high serial pin (FMC+ HSPC) connectors. This versatility allows developers to leverage a wide range of peripherals and functionalities, making it an ideal platform for comprehensive FPGA-based design exploration and evaluation..

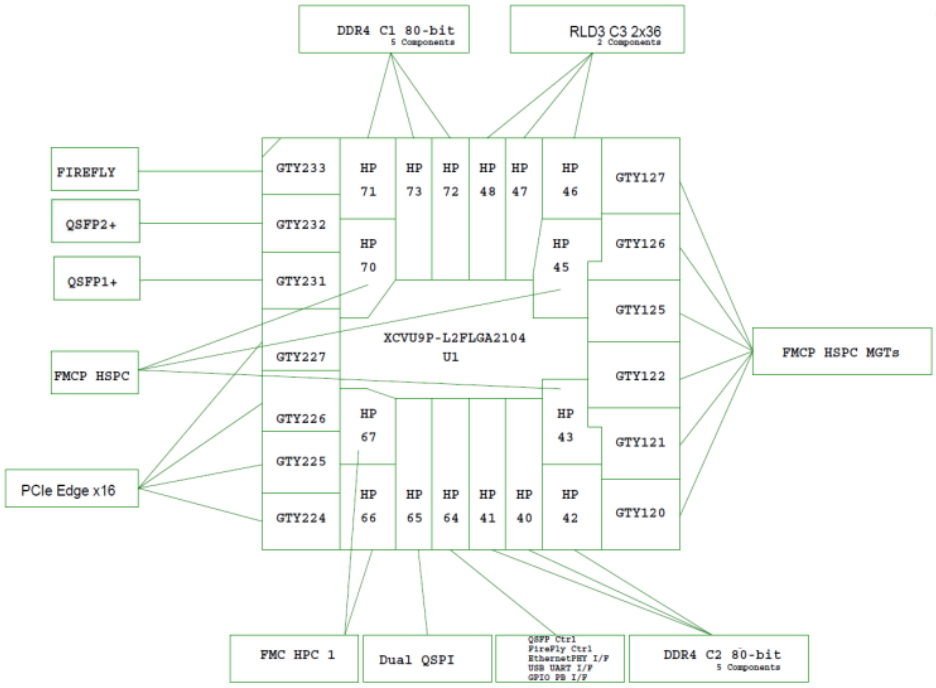


FIGURE 5.4: Block Diagram of the VCU118. Source: [31]

Initially, the architectural modifications for the Virtex UltraScale+ FPGA VCU118 Evaluation Kit involved the substitution of the Zynq UltraScale processing unit with a Microblaze core. Subsequently, it became imperative to replace the SmartConnect intellectual property (IP) components with AXI Memory-Mapped to Stream Mapper (mm2_mapper) instances. Additionally, two more AXI Memory-Mapped to Stream Mapper components were introduced to facilitate communication between the master port of the PS and the AXI Interconnect IP.

The AXI Memory-Mapped to Stream Mapper is used to bridge the gap between components that use memory-mapped interfaces (AXI) and those that use stream-based interfaces. It translates read and write requests on the AXI bus into a stream of data.

The AXI Memory-Mapped to Stream Mapper takes this request and converts it into a stream of data or extracts data from a stream, depending on the direction of the transaction.The data is then transferred between the two components using the stream interface.

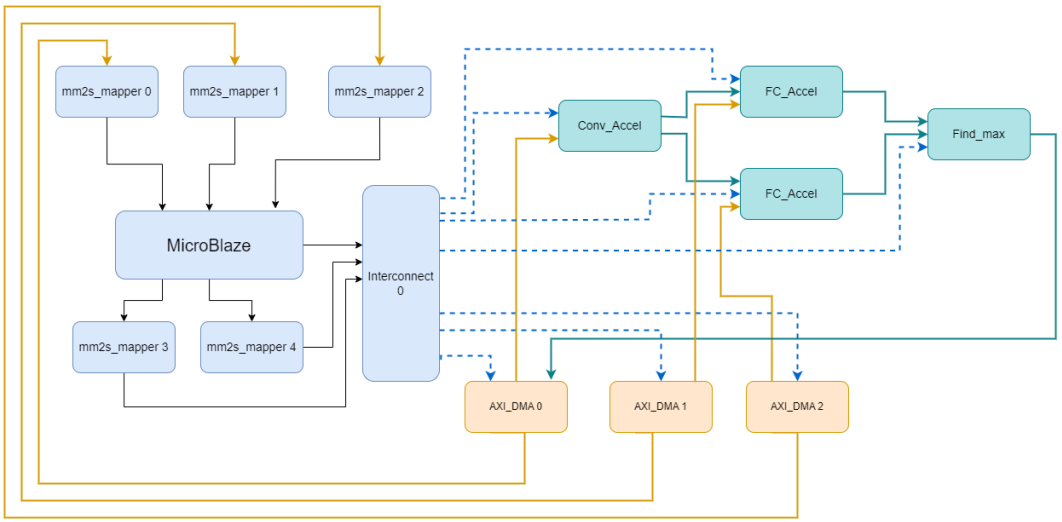


FIGURE 5.5: VCU118 architecture: clock, reset and interrupt signals were omitted, to emphasize the most structurally important connections of the AXI protocol

To incorporate multiple instances of accelerators, three AXI Interconnect IPs had to be integrated. Each AXI Interconnect IP is designated for a specific DMA (Direct Memory Access) module. These DMAs are responsible for handling streaming data for each IP, one for Convolutional layers and two for fully connected layers. This arrangement ensures seamless communication of the streamed data within each IP Figure 5.6.

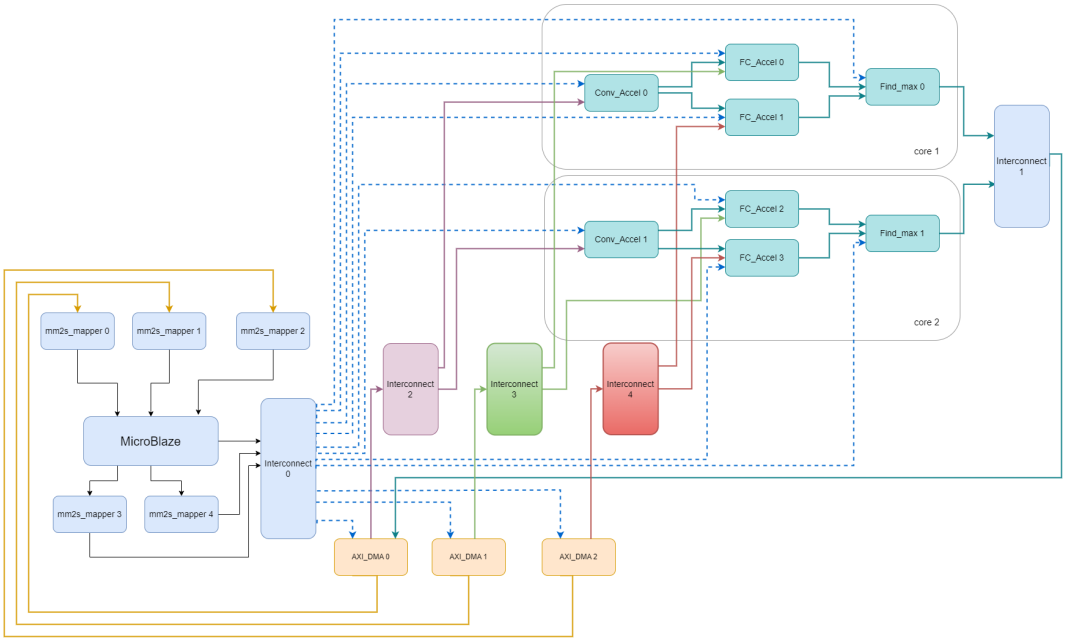


FIGURE 5.6: VCU118 architecture using 2 instances of accelerator: clock, reset and interrupt signals were omitted, to emphasize the most structurally important connections of the AXI protocol

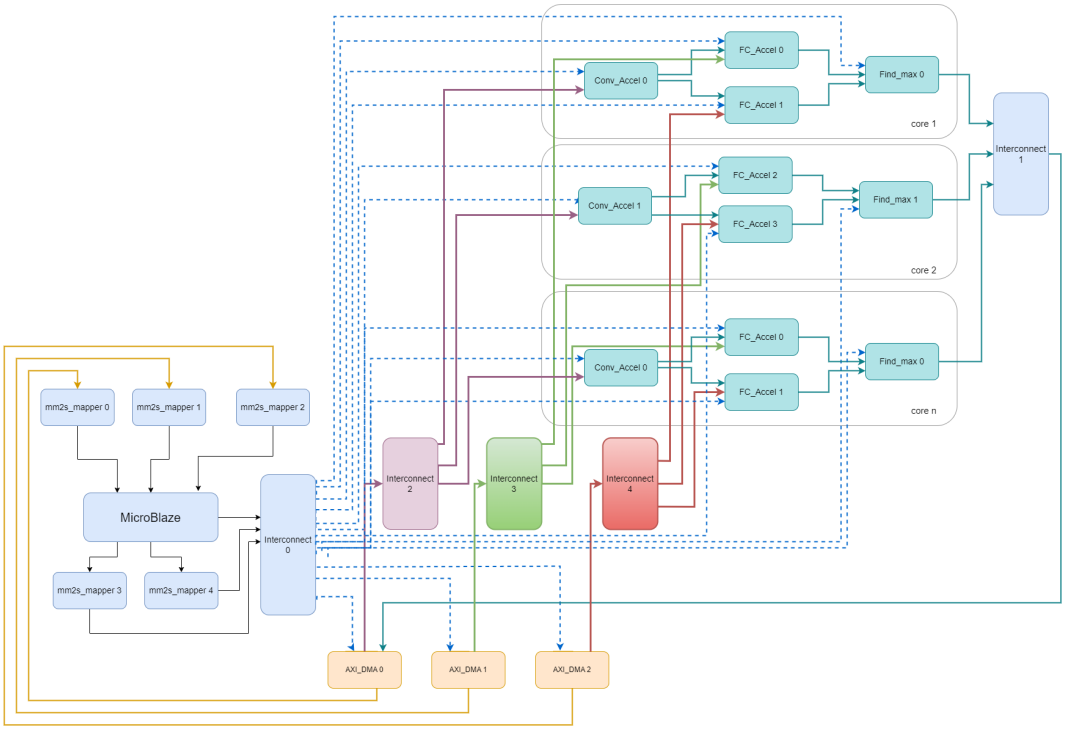


FIGURE 5.7: VCU118 architecture using n instances of accelerator: clock, reset and interrupt signals were omitted, to emphasize the most structurally important connections of the AXI protocol

### 5.1.4 Alveo U50 Data Center Accelerator Card

The Xilinx Alveo U50 accelerator card for data centers is a passively-cooled, low profile card that occupies a single slot and operates within a maximum power limit of 75W. It can support either PCIe Gen3 x16 or dual Gen4 x8 and comes with 8GB of high-bandwidth memory (HBM2) and Ethernet networking capabilities 5.8. The Alveo U50 card is designed to speed up compute-intensive applications that are memory-bound, such as financial computing, data search, and analytics. The Alveo U50 LV is specifically recommended for accelerating machine learning inference workloads. Both the U50 and U50 LV cards are identical, except for the core operating voltage, with the U50 operating at VNOM and the U50 LV at VLOW.

The Xilinx Alveo U50 accelerator card can be utilized with the Xilinx Vitis unified software platform and target platform, which streamlines the design process and enables the use of high-level programming languages such as C, C++, and OpenCL. A platform allows for configuration of the card from onboard flash memory, and upgrades can be performed through PCI Express.

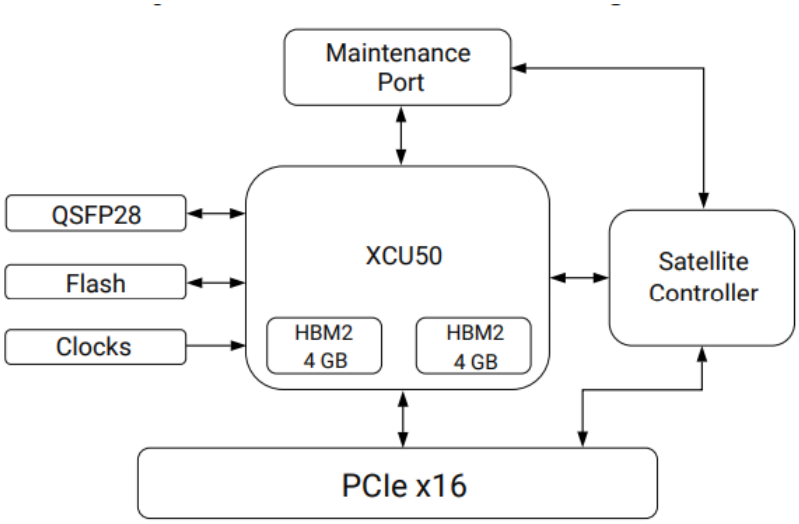


FIGURE 5.8: Block Diagram of Alveo U50

In Figure 5.9 depict the block diagram of multiple compute units. The data flow begins with the host system, which is connected to the Alveo U50 accelerator card via the PCI Express (PCIe) interface. PCIe is a high-speed serial interface that facilitates the transfer of data between the host and the FPGA-based accelerator card. The data received through PCIe is then loaded into the HBM. HBM's high bandwidth allows for quick and efficient storage and retrieval of data during computation. After the kernel has generated the data stored in HBM.

The kernels (kernel_1 through kernel_4) are processing data, and each has specific data types associated with them (e.g., inputs, images, data1, data2, and results). The data for each type is stored in a designated HBM interface. When a kernel needs access to a particular type of data (e.g., inputs or images), it communicates with the corresponding HBM interface.

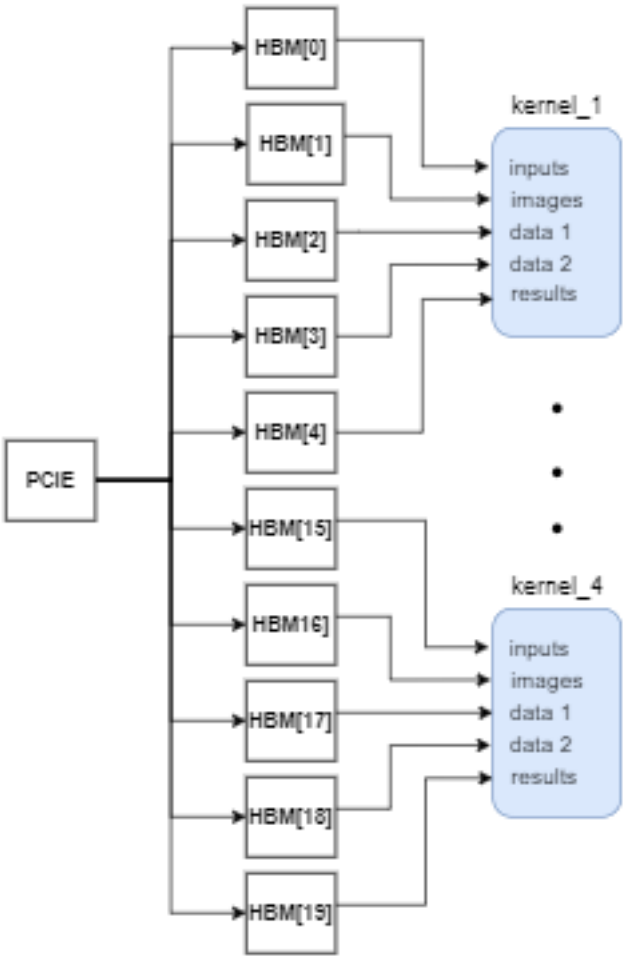


FIGURE 5.9: Block Diagram - 4 Compute Units

Each kernel is depicted in Figure 5.10. The kernel as we already mentioned, have five memory interfaces, consisting of four inputs (inputs, images, data0, data1) and one output. The kernel processes inputs by initially reading and passing them to various components. Conv_Accel receives an image, performs convolution, and transfers the results to FC_Accel, responsible for executing the fully connected layer of the CNN. Notably, two instances of the fully connected layer are employed to concurrently calculate. The results from FC_Accel are then forwarded to the Find_Max component, whose result constitutes the kernel's output.

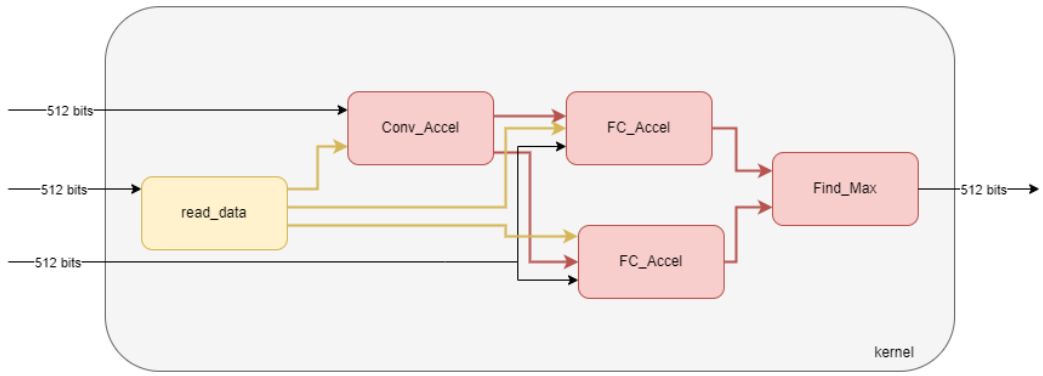


FIGURE 5.10: Block Diagram of kernel - input 512bits

## 5.2 Discussion

In the preceding sections, it became evident that certain platforms, like QFDB, boast numerous memory ports, resulting in higher overall bandwidth. However, this platform typically incorporate four FPGAs, necessitating internal communication. On the other hand, platforms such as Alveo exhibit fewer memory ports, but their advantage lies in the absence of communication challenges between multiple different FPGAs. It is crucial to acknowledge that the design must be tailored to optimize the memory subsystem for each specific platform, leading to the identification of a distinct design flow.

The variation in memory port abundance between platforms like QFDB and Alveo highlights the need for a platform-specific approach. While QFDB's extensive memory doors contribute to increased total bandwidth, the interconnected nature of its four FPGAs introduces complexities in internal communication. Conversely, Alveo's streamlined design with fewer memory ports mitigates communication challenges, capitalizing on the efficiency of a single

FPGA. Adapting the design to leverage the strengths of the memory subsystem on each platform becomes imperative, underscoring the significance of a nuanced and tailored design flow.

In essence, the observed differences underscore the necessity for an adaptable design strategy that can harness the unique characteristics of the memory subsystem on diverse platforms. Whether it is optimizing for the multitude of ports in QFDB or capitalizing on the faster single-port configuration of Alveo, designers must navigate a nuanced design flow to ensure optimal performance based on the intricacies of each platform's architecture.

# Chapter 6

# Results

This chapter presents the results obtained in the present thesis, and compares these results to previous ones from the same CNN, as described in the theses of G. Pitsis and C. Loukas. It should be noted that the previous designs were built again during this thesis on newer versions of the tools, and hence the performance and design footprint (resource utilization, etc.) which are presented in this chapter are from new runs on new tools. We will detail the resource utilization for each design and the performance.

## 6.1 Resource Utilization

| | **ZCU102 (%)** | **QFDB (only F2 FPGA) (%)** | **Alveo U50 (%)** |
|---|---|---|---|
| **LUT** | 40.0 | 40.0 | 24.23 |
| **LUTRAM** | 2.0 | 2.0 | 3.57 |
| **FF** | 39.0 | 39.0 | 18.58 |
| **BRAM** | 12.0 | 12.0 | 21.39 |
| **DSP** | 19.0 | 19.0 | 8.32 |
| **IO** | 0 | 0 | 2.40 |
| **GT** | 0 | 0 | 80.0 |
| **BUFG** | 6 | 0 | 6.10 |
| **PLL** | 0 | 0 | 6.25 |

TABLE 6.1: Utilization comparison between platforms

Table 6.1 shows a comparison between the resource utilization of the ZCU102-targeted design, the the QFDB-targeted standalone design and the Alveo U50-targeted design. As we can see, migrating from the ZCU102 to the QFDB

does not affect the resource utilization at all, and migrating to Alveo U50 there are significant differences. However, the difference between the ZCU102-target, QFDB-target designs and the Alveo U50-target design is obvious and is of course to be expected as Alveo Card is larger FPGA with more resources. We mark an decrease in Look-Up Tables (LUT) utilization by 16%, in Flip-Flops (FF) by 20% and in Digital Signal Processor by 10%.

| % | **LUT** | **LUTRAM** | **FF** | **BRAM** | **DSP** | **IO** | **BUFG** |
|---|---|---|---|---|---|---|---|
| single CU | 24.23 | 3.57 | 18.58 | 21.39 | 8.32 | 2.40 | 6.10 |
| 2 CUs | 37.21 | 5.16 | 30.33 | 29.54 | 16.57 | 2.40 | 7.59 |
| 4 CUs | 63.17 | 8.34 | 53.83 | 45.84 | 33.07 | 2.40 | 10.57 |

TABLE 6.2: Utilization Comparison between Compute Units

Table 6.2 presents the resource utilization after the use of the CNN architecture multiple instances, with the exception of the Alvep u50. Regarding these results we can see that one instance of CNN makes use of 13% LUTs, 2& LUTRAM 7% BRAM and 8% DSP. Also we can see that IO remains the same, as it is mentioned before, that kernel is using different memory port for each input/output, such in this way bottleneck is avoided by adding more compute units.

## 6.2 Power Consumption and Energy Consumption

### Power Consumption

Power consumption is defined as the amount of energy consumed per unit time to perform a specific task. It is usually measured in Watts (W) or kilo-Watts (kW). The power consumption of a system is extremely important and should be kept as low as possible. The battery life of portable electronic de-vices such as cell phones and laptops is limited by power consumption. Low power consumption leads to higher energy efficiency and lower building costs. Using a simplified and smaller architecture for a design can increase energy efficiency.

### Energy Consumption

Energy consumption refers to the energy required for accomplishing a particular task in a specific amount of time. It is commonly measured using Joule (J) or

kilo Joule (kJ). This metric value should also remain at the lowest level possible.

$$E = P \times T$$

where E represents the energy in Joules, P indicates the required power for the device to function and T is the time needed to execute the task. The Images/Joule metric can be calculated as follows:

$$\frac{Images}{Joule} = \max\left(\frac{Throughput}{Power}, \frac{1}{Power \times Latency}\right)$$

## 6.3 Throughput and Latency Speedup

### Latency

Latency, is the required time to complete a single task. In this work, latency can be the time taken to process a single image, a batch of images, a dataset of images, etc.

### Throughput

Throughput is generally referred to as the quantity of tasks completed in a given amount of time. The rate at which something is processed increases with throughput. Throughput in this work is referred to as the number of images processed per second.

$$Throughput = \frac{Images}{Time}(sec)$$

## 6.4 Overall Performance Comparison

Table 6.3 illustrates that Alveo operates at a clock frequency of 300MHz, surpassing the F2 design, which runs at 100MHz, resulting in improved latency. In the Alveo U50 design, latency is reduced to 2.485ms with a total runtime of 6.2 seconds for a dataset containing 2500 input values, as opposed to the 7.45 seconds required by the F2 design.

Utilizing multiple compute units in parallel execution further diminishes execution time significantly. Specifically, with 4 compute units, each responsible for calculating 625 images, the individual execution time per compute unit is reduced to 1.6 seconds. As previously mentioned, these compute units operate

concurrently, resulting in a total runtime of 1.6 seconds for the entire 2500 input value dataset when employing 4 compute units.

| | Clock Frequency (MHz) | Power (Watts) | Throughput (Images/s) | Latency (ms) |
|---|---|---|---|---|
| F2 | 100 | 9.523 | 329 | 3.037 |
| Alveo U50 | 300 | 20.966 | 402 | 2.485 |

TABLE 6.3: Measurements for 2500 input value dataset

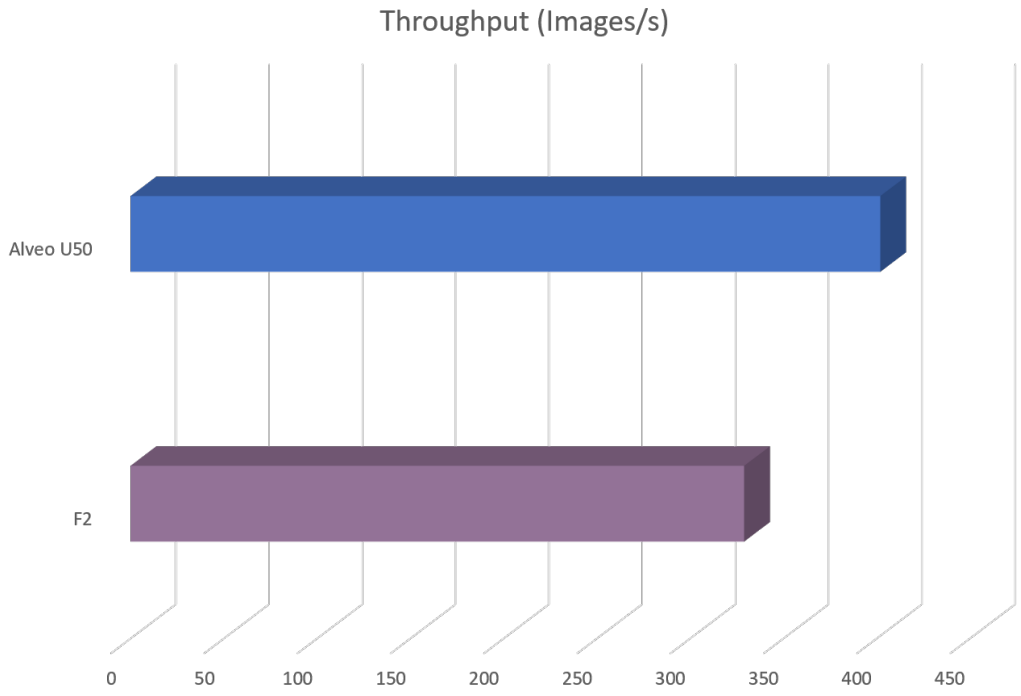


FIGURE 6.1: Total Throughput

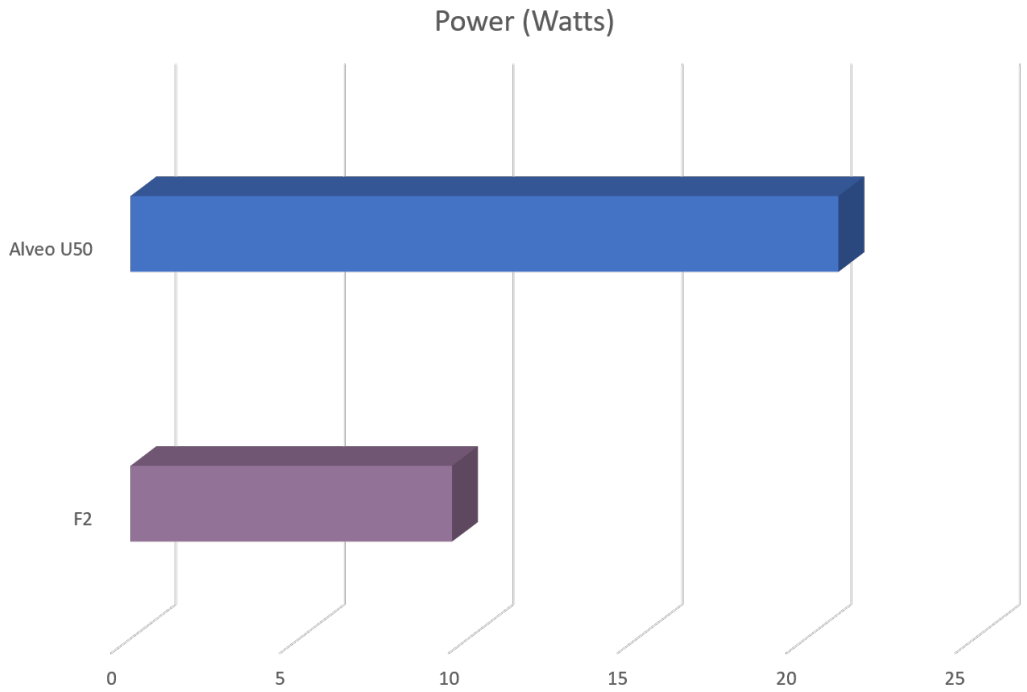


FIGURE 6.2: Total Power Consumption

Figure 6.1 depicts the throughput of each design, confirming our earlier observation that the Alveo U50 design operates more efficiently due to its higher clock frequency, resulting in faster results.

Simultaneously, Figure 6.2 illustrates that the Alveo U50 FPGA demonstrates higher power consumption compared to the F2 design, which is in line with expectations due to the Alveo U50's larger size requiring more power. Specifically, the Alveo U50 consumes 20.966W with an execution time of 6.2 seconds per compute unit, resulting in an energy consumption of 130 Joules. Conversely, the F2 consumes 9.523W with an execution time of 7.45 seconds per compute unit, leading to an energy consumption of 71 Joules. This indicates that per compute unit, the F2 is more energy-efficient.

However, when considering the utilization of four compute units, the Alveo U50 demonstrates a total execution time of 1.6 seconds with a power consumption of 42.599 Watts, resulting in a total energy consumption of 68 Joules. Consequently, the Alveo U50 emerges as more energy-efficient in this scenario.

In summary, while the F2 proves to be more energy-efficient per compute unit, the Alveo U50 surpasses it in energy efficiency when utilizing multiple compute units simultaneously.

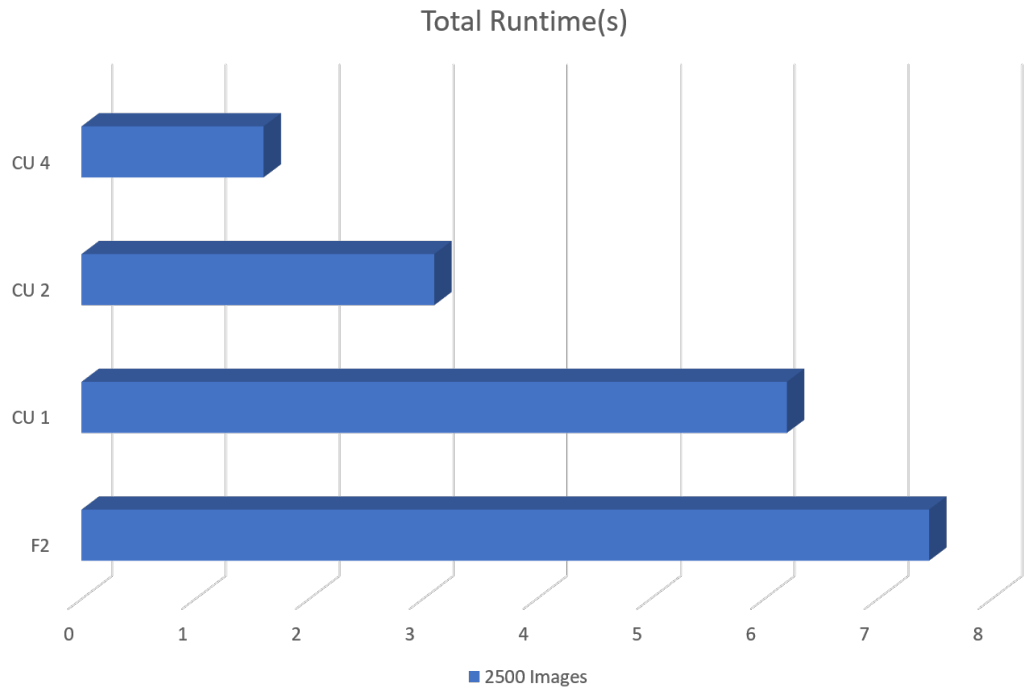


FIGURE 6.3: Execution runtime

Lastly, in Figure 6.3, the execution time is depicted concerning the number of utilized compute units. Notably, employing 4 compute units results in a performance improvement that is four times faster than utilizing a single compute unit. This acceleration is attributed to each compute unit handling the calculation of 2500/4 images. As discussed in the previous chapter, the concurrent operation of all compute units, coupled with the advantageous configuration of Alveo memory, allows for parallel access to memory without introducing additional latency to read and write in memory.

# Chapter 7

# Conclusions and Future Work

In summary, the results presented in this study provide valuable insights into the performance characteristics of Alveo U50 and ZCU102 FPGAs in the context of image processing. Alveo U50 demonstrated superior latency, lower runtime, and the ability to harness parallelism effectively, showcasing its potential for applications demanding high computational throughput. Conversely, ZCU102 exhibited competitive performance but with slightly higher latency and runtime.

The trade-off between FPGA size and power consumption was a key consideration in this study. While Alveo U50's larger size allowed for the integration of more compute units and concurrent execution, it came at the cost of higher power consumption. Although this led to increased power consumption, leveraging multiple compute units ultimately resulted in lower energy consumption despite the higher power usage and execution time.

Migrating a design from the Vivado design flow to the Vitis design flow is a time-intensive process involving numerous steps. Additionally, there is a need for redesign when dealing with variations in the memory subsystem or clock configurations.

The tools continue to evolve, and there are inconsistencies even within High-Level Synthesis (HLS). Maintaining an IP core over time is challenging, even within the same design flow. The complexity increases when transitioning from Vivado to Vitis.

In conclusion, this study contributes to the understanding of FPGA performance and highlights the nuanced decision-making process in FPGA selection. The results underscore the potential benefits of utilizing larger FPGAs for specific applications while emphasizing the importance of managing associated power consumption. Ultimately, the insights gained from this research can

inform the design and implementation of FPGA-based systems, paving the way for improved efficiency and performance in image processing applications.

# References


[1] Antonios-Georgios Pitsis. "Design and implementation of an FPGA-based convolutional neural network accelerator". Diploma Work. Chania, Greece: School of Electrical and Computer Engineering, 2018.

[2] Charisios Loukas. "Large Scale Design and Implementation of Convolutional Neural Networks based on Large FPGA Arrays". Diploma Work. Chania, Greece: School of Electrical and Computer Engineering, 2020.

[3] George Pitsis et al. "Efficient Convolutional Neural Network Weight Compression for Space Data Classification on Multi-fpga Platforms". In: *ICASSP 2019 - 2019 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. 2019, pp. 3917–3921. DOI: 10.1109/ICASSP.2019.8682732.

[4] Xilinx. *Alveo U50 Data Center Accelerator Card*. https://www.xilinx.com/products/boards-and-kits/alveo/u50.html.

[5] Yann LeCun and Yoshua Bengio. "Convolutional Networks for Images, Speech, and Time Series". In: *The Handbook of Brain Theory and Neural Networks*. Cambridge, MA, USA: MIT Press, 1998, 255–258. ISBN: 0262511029.

[6] *Convolution Neural Network*. URL: https://developersbreach.com/convolution-neural-network-deep-learning/.

[7] Xuepeng Chang et al. "A Memory-Optimized and Energy-Efficient CNN Acceleration Architecture Based on FPGA". In: *2019 IEEE 28th International Symposium on Industrial Electronics (ISIE)*. 2019, pp. 2137–2141. DOI: 10.1109/ISIE.2019.8781162.

[8] Kuan-Hung Chen, Chun-Wei Su, and Jen-He Wang. "Energy-efficient and Accurate Object Detection Design on an FPGA Platform". In: *2022 IET International Conference on Engineering Technologies and Applications (IET-ICETA)*. 2022, pp. 1–2. DOI: 10.1109/IET-ICETA56553.2022.9971590.

[9] Hong Wang et al. "Convolutional Neural Network Accelerator on FPGA". In: *2019 IEEE International Conference on Integrated Circuits, Technologies*

*and Applications (ICTA)*. 2019, pp. 61–62. DOI: 10.1109/ICTA48799.2019.9012821.

[10] Xin Li et al. "A high utilization FPGA-based accelerator for variable-scale convolutional neural network". In: *2017 IEEE 12th International Conference on ASIC (ASICON)*. 2017, pp. 944–947. DOI: 10.1109/ASICON.2017.8252633.

[11] Liqiang Lu et al. "An Efficient Hardware Accelerator for Sparse Convolutional Neural Networks on FPGAs". In: *2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. 2019, pp. 17–25. DOI: 10.1109/FCCM.2019.00013.

[12] Heekyung Kim and Ken Choi. "Low Power FPGA-SoC Design Techniques for CNN-based Object Detection Accelerator". In: *2019 IEEE 10th Annual Ubiquitous Computing, Electronics & Mobile Communication Conference (UEMCON)*. 2019, pp. 1130–1134. DOI: 10.1109/UEMCON47517.2019.8992929.

[13] Masoud Shahshahani et al. "An Automated Tool for Implementing Deep Neural Networks on FPGA". In: *2021 34th International Conference on VLSI Design and 2021 20th International Conference on Embedded Systems (VLSID)*. 2021, pp. 322–327. DOI: 10.1109/VLSID51830.2021.00060.

[14] Md Najrul Islam, Rahul Shrestha, and Shubhajit Roy Chowdhury. "An Uninterrupted Processing Technique-Based High-Throughput and Energy-Efficient Hardware Accelerator for Convolutional Neural Networks". In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 30.12 (2022), pp. 1891–1901. DOI: 10.1109/TVLSI.2022.3210963.

[15] Ryosuke Kuramochi and Hiroki Nakahara. "An FPGA-Based Low-Latency Accelerator for Randomly Wired Neural Networks". In: *2020 30th International Conference on Field-Programmable Logic and Applications (FPL)*. 2020, pp. 298–303. DOI: 10.1109/FPL50879.2020.00056.

[16] Prasetiyo et al. "Accelerating Deep Convolutional Neural Networks Using Number Theoretic Transform". In: *IEEE Transactions on Circuits and Systems I: Regular Papers* 70.1 (2023), pp. 315–326. DOI: 10.1109/TCSI.2022.3214528.

[17] Gyeongcheol Shin, Junsoo Kim, and Joo-Young Kim. "OpenMDS: An Open-Source Shell Generation Framework for High-Performance Design on Xilinx Multi-Die FPGAs". In: *IEEE Computer Architecture Letters* 21.2 (2022), pp. 101–104. DOI: 10.1109/LCA.2022.3202016.

[18] Young-kyu Choi et al. "When hls meets fpga hbm: Benchmarking and bandwidth optimization". In: *arXiv preprint arXiv:2010.06075* (2020).

[19] Andrea Coccaro et al. "Fast neural network inference on FPGAs for triggering on long-lived particles at colliders". In: *Machine Learning: Science and Technology* 4.4 (2023), p. 045040.

[20] Xilinx. *Vivado Hls Level Synthesis*. URL: https://docs.xilinx.com/v/u/en-US/ug902-vivado-high-level-synthesis.

[21] Xilinx. *Vivado Design Suite User Guide*. https://users.ece.utexas.edu/~mcdermot/arch/articles/Zynq/ug892-vivado-design-flows-overview_2016.pdf.

[22] Xilinx. *Introduction to Vitis Flows*. URL: https://docs.xilinx.com/r/en-US/ug1393-vitis-application-acceleration/Introduction-to-Vitis-Flows.

[23] Xilinx. *Vitis Unified Software Platform Documentation*. https://xilinx.eetrend.com/files/2019-10/wen_zhang_/100045696-82978-vitis-application-acceleration.pdf.

[24] Xilinx. *Migrating Vivado to Vitis*. URL: https://docs.xilinx.com/r/en-US/ug1399-vitis-hls/Migrating-from-Vivado-HLS-to-Vitis-HLS.

[25] Xilinx. *Deprecated and Unsupported Features*. URL: https://docs.xilinx.com/r/en-US/ug1399-vitis-hls/Deprecated-and-Unsupported-Features.

[26] Xilinx. *Migrating from SDK to Vitis*. URL: https://www.xilinx.com/video/software/migrating-from-sdk-to-vitis.html.

[27] Xilinx. *vitis-application-acceleration*. URL: https://docs.xilinx.com/v/u/2019.2-English/ug1393-vitis-application-acceleration.

[28] Xilinx. *HBM Configuration and Use*. URL: https://docs.xilinx.com/r/en-US/ug1393-vitis-application-acceleration/HBM-Configuration-and-Use.

[29] Xilinx. *ZCU102 Evaluation Board User Guide*. URL: https://docs.xilinx.com/v/u/en-US/ug1182-zcu102-eval-bd.

[30] Fabien Chaix et al. "Implementation and Impact of an Ultra-Compact Multi-FPGA Board for Large System Prototyping". In: *2019 IEEE/ACM International Workshop on Heterogeneous High-performance Reconfigurable Computing (H2RC)*. 2019, pp. 34–41. DOI: 10.1109/H2RC49586.2019.00010.

[31] Xilinx. *VCU118 Evaluation Board User Guide*. https://docs.xilinx.com/v/u/en-US/ug1224-vcu118-eval-bd.