UNIVERSITAT POLITÈCNICA DE VALÈNCIA

etsinf

Escola Tècnica Superior d'Enginyeria Informàtica

Escola Tècnica Superior d'Enginyeria Informàtica
Universitat Politècnica de València

Adaptation of High Performance and High Capacity Reconfigurable Systems to OpenCL Programming Environments

MASTER DEGREE FINAL WORK

Master Degree in Computer Engineering

Author: Davide Russo

Tutor: Flich Cardo, José

First External Tutor: Cilardo, Alessandro

Director Experimental: Tornero Gavilá, Rafael

Course 2019-2020

Resumen

En este trabajo se va a realizar la adaptación de un sistema reconfigurable de cómputo basado en tecnologías de FPGAs hacia entornos de programación en OpenCL. El sistema reconfigurable forma parte de un prototipo de cálculo del proyecto Europeo MANGO que incluye 96 FPGAs. Con el fin de optimizar el uso y de obtener sus máximas prestaciones, se hace imprescindible una adaptación a entornos de programación de sistemas heterogéneos como OpenCL, lo cual simplifica su programación y uso. En este trabajo se realizarán todas las actividades necesarias para una correcta implementación de la capa software y hardware necesaria para su uso en OpenCL así como una evaluación de las prestaciones obtenidas y de la flexibilidad ofrecida por la solución aportada.

Este trabajo se ha llevado a término durante una estancia de cinco meses en la Universitat Politécnica de Valéncia. Esta estancia está vinculada a un acuerdo entre la Universitat Politécnica de Valéncia y la Università degli Studi di Napoli Federico II.

Palabras clave: Sistema de cómputo heterogéneos; PCI express; OpenCL; Xilinx runtime; Custom platform FPGA

Abstract

In this work, we adapt a reconfigurable computer system based on FPGA technologies to OpenCL programming environments. The reconfigurable system is part of a compute prototype of the MANGO European project that includes 96 FPGAs. To optimize the use and to obtain its maximum performance, it is essential to adapt it to heterogeneous systems programming environments such as OpenCL, which simplifies its programming. In this work, all the necessary activities for correct implementation of the software and hardware layer required for its use in OpenCL will be carried out, as well as an evaluation of the performance obtained and the flexibility offered by the solution provided.

This work has been performed during an internship of 5 months. The internship is linked to an agreement between UPV and UniNa (Università degli Studi di Napoli Federico II).

Keywords: Heterogeneous computing system; PCI express; OpenCL; Xilinx runtime; Custom platform FPGA

Contents

List of Figures

List of Tables

Acronyms

AER Advanced Error Reporting.

AI Artificial Intelligence.

ALM Adaptive Logic Module.

ALUT Adaptive LUT.

API Application Programming Interface.

APM AXI Performance Monitor.

ASIC Application Specific Integrated Circuit.

AXI Advanced eXtensible Interface.

BAR Base Address Register.

BD Big Data.

BLE Basic Logic Element.

BRAM Block RAM.

C2H DMA Write Channel.

CAD Computer-Aided Design.

CIC Cascaded Integrator-Comb.

CLB Configurable Logic Block.

CLI Command-Line Interface.

CMOS Complementary Metal-Oxide Semiconductor.

CPU Central Processing Unit.

CU Compute Unit.

DDR Double Data Rate.

DL Deep Learning.

DLL Delay-Locked Loop.

DMA Direct Memory Access.

DMBI Device Message Box Interface.

DNN Deep convolutional Neural Network.

DSA Device Support Archive.

DSP Digital Signal Processing.

EEPROM Electrically Erasable Programmable Read-Only Memory.

EPROM Erasable Programmable Read-Only Memory.

FF Flip-Flop.

FFT Fast Fourier Transform.

FIFO First In First Out.

FIR Finite Impulse Response.

FPGA Field Programmable Gate Array.

FSM Finite-State Machine.

GN General-purpose Node.

GPU Graphics Processing Unit.

GUI Graphical User Interface.

H2C DMA Read Channel.

HAL Hardware Abstraction Layer.

HDL Hardware Description Language.

HLS High-Level Synthesis.

HN Heterogeneous Node.

HPC High-Performance Computing.

I/O Input/Output.

IBUF Input Buffer.

IC Integrated Circuit.

IDE Integrated Design Environment.

IP Intellectual Property.

LAB Logic Array Block.

LUT Look-Up Table.

MAC Multiply-Accumulate.

MIG Memory Interface Generator.

MPSoC Multiprocessor SoC.

OBUF Output Buffer.

OCL Region OpenCL Region.

OpenCL Open Computing Language.

Pblock Physical Block.

PCIe PCI Express.

PE Processing Element.

PLL Phase-Locked Loop.

PR Partial Reconfiguration.

PROM Programmable Read-Only Memory.

RAM Random Access Memory.

RM Reconfigurable Module.

RP Reconfigurable Partition.

RPR Regular Partial Reconfiguration.

RTL Register Transfer Level.

SDC Synopsys Design Constraints.

SDRAM Synchronous Dynamic Random Access Memory.

SIMD Single Instruction Multiple Data.

SLR Super-Logic Region.

SoC System-on-a-Chip.

SPI Serial Peripheral Interface.

SPMD Single Program Multiple Data.

SRAM Static Random Access Memory.

SRL Shift Register Logic.

SSI Stacked Silicon Interconnect.

SysMon System Monitor.

System ILA System Integrated Logic Analyzer.

Tcl Tool Command Language.

TSV Through-Silicon Vias.

USB Universal Serial Bus.

WA Write Address.

WE Write Enable.

xbinst Xilinx Board INSTallation.

xbsak Xilinx Board Swiss Army Knife.

XCL HAL Xilinx OpenCL Hardware Abstraction Layer.

xclbin Xilinx OpenCL Compute Unit Binary.

XDC Xilinx Design Constraints.

XDMA Xilinx DMA Subsystem for PCIe.

xocc Xilinx OpenCL Compiler.

XPR Expanded Partial Reconfiguration.

CHAPTER 1
Introduction

1.1 Heterogeneous system for HPC application

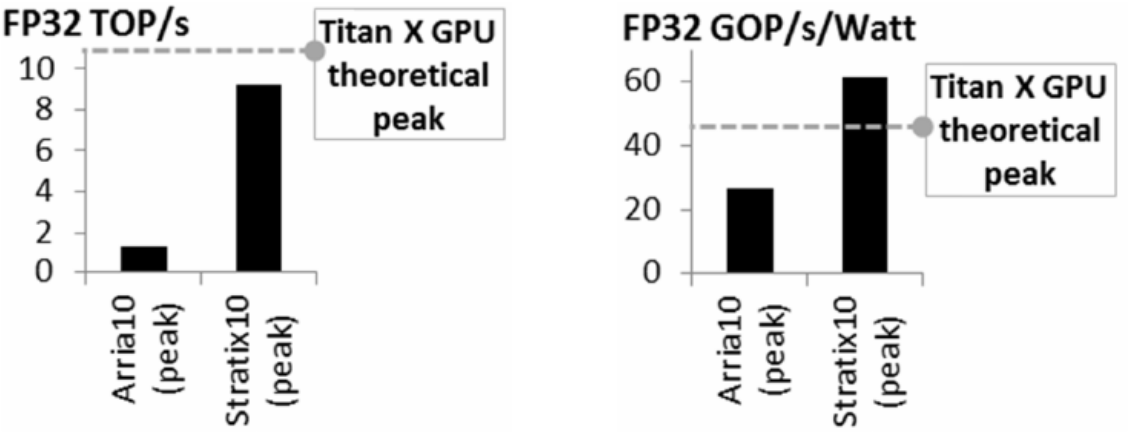
Traditional computing systems always relied on the use of Central Processing Units (CPUs) for the computation of applications. This was a valid solution while power and energy were not a concern. However, as performance and size of these systems grew the influence of the power consumption and energy dissipation became apparent and CPUs were no longer regarded as the only one computing components in the system.

This led to the emergence of heterogeneous computing. In such a system, different types of computing devices, each with different power consumption values and performances can be used all together. They rely on different architectures and therefore they are better suited for specific and particular programming approaches and problems definitions.

The first boost we had in High-Performance Computing (HPC) systems was the adoption of Graphics Processing Units (GPUs) as computing devices, typically regarded to be used in desktop systems for graphics processing. The computing capabilities of GPUs as they have many small cores with floating point capabilities, made those devices perfect for massive parallel applications being efficiently computed on them. Although power consumption of GPUs can be on par with CPUs they exhibit one or two orders of magnitude (if not more) when compared with CPUs.

Indeed, nowadays, new applications from the recent Artificial Intelligence (AI) boost with Deep convolutional Neural Network (DNN), and the fusion of Big Data (BD) and HPC led the GPUs to be seen as commodity components and key players in such large installations.

While the CPU and GPUs are combined and they are the elected components, we can see the emergence of other heterogeneous components which can be used for more specific application domains or where power and energy are of real concern. This is the case of manycores (one clear example is the Intel Xeon Phi which later was discontinued) and the Field Programmable Gate Arrays (FPGAs). FPGAs are flexible devices where its architecture can be programmed and thus, the hardware inside the FPGA can be perfectly adapted to the algorithm used



(a) Performance **(b)** Performance/Watt

Figure 1.1: Results for Classic DNNs

to solve a problem. Indeed, FPGAs are massively used in embedded systems domains and critical application domains such space, avionics, and automotive.

For HPC systems there are new efforts and early adoptions of FPGAs-based systems for HPC computation problems. Although FPGAs are flexible, they are not so powerful as GPUs in floating point operations. However, FPGAs can be very profitable in specific application domains for HPC where floating point is not of vital need and reduced precision algorithms can be used instead. **In this project we bet for the use of FPGAs as a equal partner in the set of computing devices and focus on their adoption for HPC systems.**

in favour of what has been said, a study carried out by Intel is reported [5]. This study compares FPGAs and GPU, supported by several algorithms of DNN. In this thesis work, the results of what is called *Classic DNNs* in the study are reported in the following Figures 1.1.

As we can see, the threshold in the case of performance, represented by the GPU, is not reached by the two FPGAs. While in the case of Strafix 10, the power consumption is much higher.

The use of FPGAs in HPC systems is increasingly widespread due to this power consumption advantage: this advantage is multiplied by the huge number of FPGAs that can replace GPUs in a heterogeneous HPC system, drastically reducing power consumption.

1.2 Standard vs custom FPGAs platform

Once we adopted the use of FPGAs for HPC systems, we need to care about the flexibility offered by these systems for the average programmer. Indeed, it is well known the complexity and the difficulties that FPGA programming has imposed to the end users. This is out of question as is clear the average programmer does not have any expertise on computer architecture and any design expertise. Indeed, an FPGA can be programmed easily but the resulting performance will be quite poor as the flexibility of components connection inside the FPGA are not exploited.

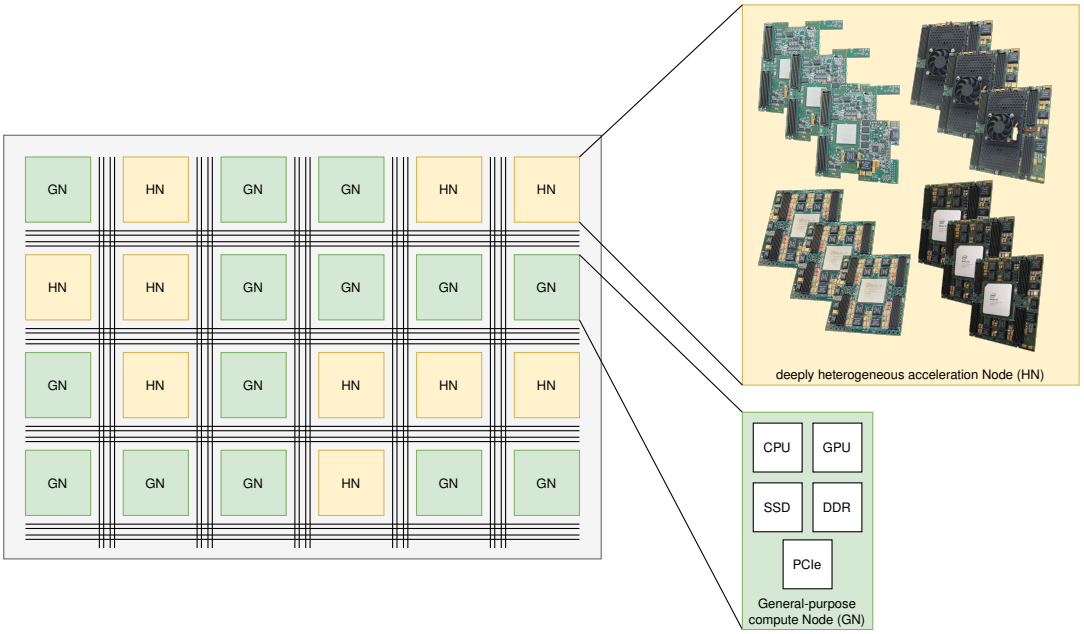


Figure 1.2: MANGO Prototype

To solve this bridge gap between FPGA programming and end user expertise, the Open Computing Language (OpenCL) programming paradigm came into play (other solutions exist also). OpenCL defines a programming approach where FPGAs can be programmed easily and with the required abstraction of its complexities and optimization opportunities. In Chapter 2.3 we fully describe the OpenCL framework.

OpenCL can be directly used by most FPGA boards available in the market. Indeed, the FPGA manufacturing companies (mostly Xilinx and Intel/Altera) offer their boards with current software and hardware support for the use of OpenCL programming frameworks. Xilinx and Intel/Altera market different FPGA-based acceleration platforms, but the flow used is very similar. The key goal of the flow lays in enabling a hardware platform and a set of software libraries to diminish programming complexities and facilitate the application of acceleration to software developers. However, and here is where our project is focused to, other FPGA systems implemented by third party companies do not have such support for OpenCL as they are focused on other fields, mostly Application Specific Integrated Circuit (ASIC) prototyping. This is the case of the products offered by the proDesign company, which offers an interesting solution where one can build a complete large FPGA solution with components as if they were part of a LEGO game. FPGA boards, Double Data Rate (DDR) memories, and Input/Output (I/O) components can be plugged in different ways based on the computing needs.

1.3 MANGO infrastructure

The MANGO prototype is shown in Figure 1.2. As we can see there is two types of node:

- **Heterogeneous Node (HN)**: to instantiate different hardware architectures coded directly in Hardware Description Language (HDL) programming languages. Each HNs consist of 12 FPGAs and 22GB of DDR-3 and DDR-4 memory mounted on top of 4 proFPGA motherboards from ProDesign GmbH. This cluster is also heterogeneous, since it is composed of different types of FPGAs: Xilinx Kintex Ultrascale KU115, Xilinx Virtex 7 Series V2000T, Xilinx Zynq 7000 SoC Z100 and Intel Stratix 10 SG280.

- **General-purpose Node (GN)**: to run a low-level runtime library and extended a runtime management system to manage both the different computational units implemented on a given architecture and the architecture itself. Each GNs consist of an high-end Intel Xeon E5 V3, a GPU, 64 GB of DDR4 of memory, 1 TB SSD hard disk and a PCI Express (PCIe) connectivity.

Every HN is connected to a GN through PCIe Gen3 x8 lanes, thus every GN can access two different HNs, for a total of 24 FPGAs and 44 GB of DDR3/4 memory.

The MANGO prototype was originally made of 16 clusters each with 12 FPGAs and 8 DDR memories. However, after project end, the prototype was split and UPV received half of the prototype, which still has reasonable performance capabilities. In this section we briefly describe the MANGO prototype located at UPV premises and which is the target of our project.

1.4 Goals and Motivations of this Work

Contrary to the Intel and Xilinx approach, the FPGAs delivered within the MANGO prototype are not equipped with any kind of circuitry to provide a high-performance communication link with the host or to facilitate the programming of these devices to software developers.

The goal of this thesis work is to create a Platform, including both hardware and software, on a FPGA implemented by a third-party company. The Platform will be used by the MANGO project and will therefore use its infrastructure. In particular the FPGA target is the **Xilinx Kintex UltraScale KU115** provided by proDesign.

To reach the final goal, several sub-objectives will be achieved, including enabling the OpenCL and High-Level Synthesis (HLS) high-level programming models for the MANGO prototype.

1.5 Structure of the Document

The project of this thesis is to solve a purely technical problem. Therefore, in the Chapter 2 will be exposed all the knowledge useful to a reader to understand what has been addressed. The chapter is extremely substantial because, as also

said in Chapter 1, the proposed problem embraces many technologies, including FPGA, OpenCL and Platform concept.

In the Chapter 3 the whole process of development of the custom Platform and all the design choices made and the why of certain choices will be explained. In addition, it will be explained in the same Chapter, the tests carried out to evaluate the correct functioning of the Custom Platform and evaluate its performance.

In Chapter 4, the final conclusions are presented, along with some suggestions on how the results could be further improved.

CHAPTER 2
Background

As said, several technologies are used in this thesis work. This chapter introduce the knowledge about these technologies, therefore most of reported information are taken from technical official documentation.

2.1 FPGA introduction

FPGAs are pre-fabricated silicon devices that can be electrically programmed to become almost any kind of digital circuit or system. Compared to ASIC technology, FPGAs provide advantages in terms of reduced reprogramming time and, consequently, cost. In fact, while for ASICs reprogramming time takes months and millions of dollars, for FPGAs it takes only a few moments. The flexible nature of an FPGA has a significant cost in terms of area, delay and power consumption.

2.1.1. Architecture

As shown in figure 2.1, FPGAs consist of a series of programmable logic blocks of different types: general logic blocks, memory blocks and multipliers. These elements are interconnected by a programmable routing fabric. The matrix is surrounded by programmable I/O blocks that connect the chip to the outside world. The term programmable indicates the ability to program a function in the chip after silicon manufacturing is completed.

2.1.2. Taxonomy

2.1.2.1. Static Memory Programming Technology

The static memory cells, represented in Figure 2.2, are the basis of the Static Random Access Memory (SRAM) programming technology. Nowadays this technology is the most used by Xilinx, Lattice and Altera. In these devices, static memory cells are distributed throughout the FPGA to provide configurability. The SRAM cells allow to

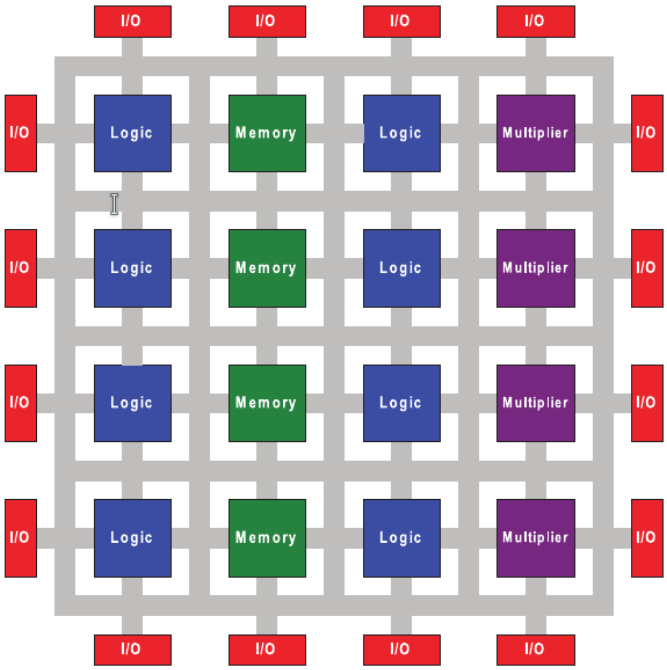


Figure 2.1: FPGA architecture

- select lines to multiplexers that steer interconnect signals, by the circuit shown in Figure 2.3a.
- store the data in the Look-Up Tables (LUTs) that are typically used in SRAM-based FPGAs to implement logic functions, by the circuit shown in Figure 2.3b.

The main advantages of SRAM programming technology are reprogrammability and the use of standard Complementary Metal-Oxide Semiconductor (CMOS) process technology. The first is guaranteed by the fact that a SRAM cell can be programmed an indefinite number of times. A dedicated circuit on the FPGA initializes all SRAM bits at power-up and configures the bits with a user-supplied configuration. The second advantage is given by the fact that the use of SRAM

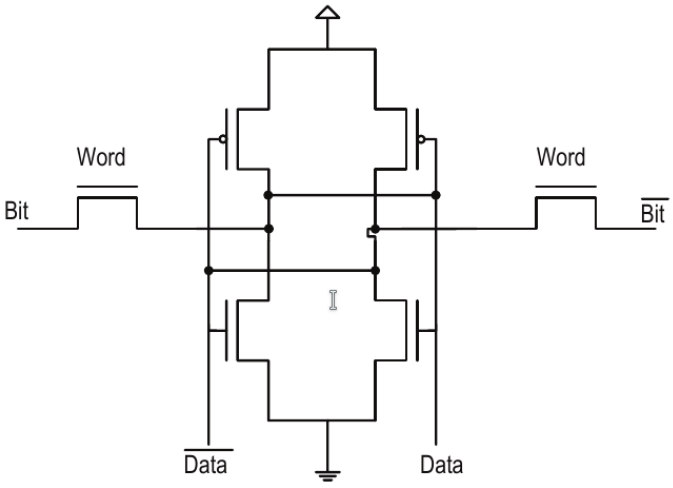
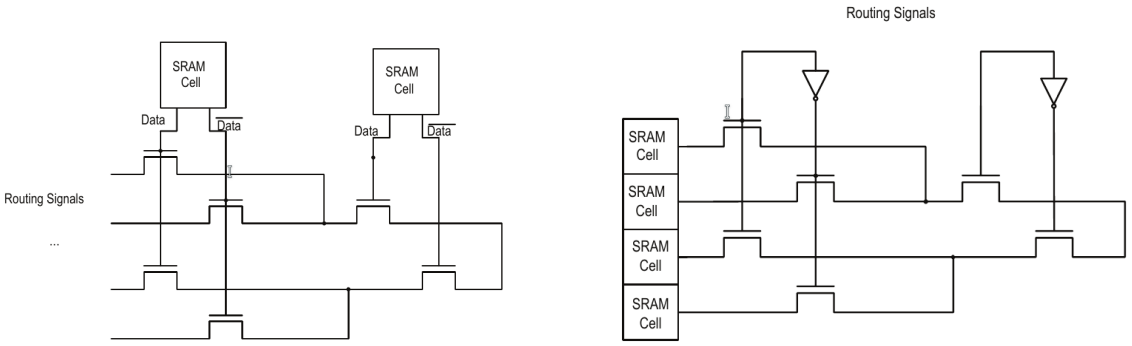


Figure 2.2: SRAM cell



(a) Multiplexer with Static Memory Cell

(b) Static Memory Cells and Lookup Table

Figure 2.3: Use of static memory cells

cells does not require particular phases of Integrated Circuit (IC) processing beyond the standard CMOS. As a result, SRAM-based FPGAs can use the latest CMOS technology available and therefore benefit from the increased integration, higher speeds and lower dynamic power consumption of new processes with smaller minimum geometries. However, SRAM-based programming technologies has the following disadvantages:

- The SRAM cell requires 5 or 6 transistors and the programmable element used for signal interconnection requires at least a single transistor causing size problems.
- Use of external devices, such as flash or Electrically Erasable Programmable Read-Only Memory (EEPROM), to permanently store configuration data when the device is powered down.
- Because the configuration information must be loaded into the device when the power is turned on, there is a possibility that the configuration information may be intercepted and stolen for use in a competing system.
- Multiplexers are implemented on pass transistor. However, they are not ideal switches as they have significant ignition resistance and an appreciable capacitive load. The problem increases as the FPGAs switches to smaller devices.

2.1.2.2. Flash/EEPROM Programming Technology

This technology is based on the use of floating gate programming technologies, shown in figure 2.4, which inject the charge onto a gate that *floats* over the transistor. This approach is used in flash or EEPROM memory cells.

The operation of the scheme is as follows: the smaller programming transistor is used to program the floating gate (injecting a charge that remains even when the power is switched off) while the larger switching transistor serves as a programmable switch. The switching transistor must also be used to cancel the device.

The advantages offered by this technology are: non-volatility and efficient use in terms of area. Compared to SRAM-based programming technology, non-

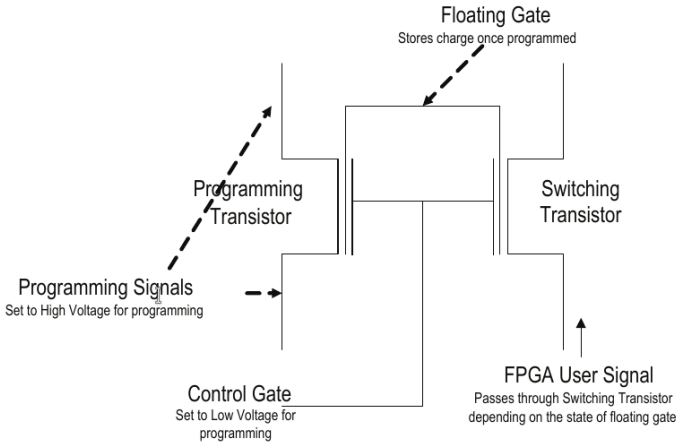


Figure 2.4: Floating gate transistor

volatility eliminates the need for external resources needed to store and load configuration data and therefore also eliminates the need to wait for configuration data to load when the device is powered on. The flash memory cells approach is also more area efficient than SRAM-based technology due to the reduced number of transistors to build the cell.

One of a disadvantages of flash-based devices is that they can not be reprogrammed an infinite number of times. The accumulation of charge in the oxide eventually prevents a flash-based device from being erased and programmed correctly after a certain number of reprogramming. For example the **Actel ProASIC3** is only guaranteed for 500 programming cycles. Another significant disadvantage of flash devices is the need for a non-standard CMOS process. Furthermore, like SRAM-based programming technology, this programming technology suffers from relatively high resistance and capacitance due to the use of transistor switches.

Nowadays, there are devices on the market that use flash memory in combination with SRAM programming technology. In particular, on-chip flash memory is used to provide non-volatile storage, while SRAM cells are still used to control programmable elements in the design. This solves problems associated with the volatility of pure SRAM approaches, such as the cost of additional storage devices or the ability to intercept configuration data, while maintaining the infinite reconfigurability of SRAM-based devices.

With the exception of very low capacity devices, such approaches are no longer commonly used because of the static power dissipation inherent in such schemes.

2.1.2.3. Anti-fuse Programming Technology

Anti-fuse programming technology is based on anti-fuse as programmable element. This structure exhibits very high-resistance under normal circumstances but a low resistance link can be created. Unlike SRAM or floating gate programming technologies, this link is permanent. Two approaches have been used to implement anti-fuses: dielectric anti-fuses and metal-to-metal-based anti-fuses.

The primary advantage of anti-fuse programming technology is its low area. With metal-to-metal anti-fuses, no silicon area is required to make connections, decreasing the area overhead of programmability. Anti-fuses have an additional advantage: they have lower on resistances and parasitic capacitances than other programming technologies. The low area, resistance, and capacitance of the fuses means it is possible to include more switches per device than in other technologies. Non-volatility also means that the device works instantly once programmed. This lowers system costs since additional memory for storing the programming information is not required and it also allows the FPGA to be used in situations that require operation immediately upon power up. Finally, since programming, and hence transmitting the bitstream to the FPGA, need only be done once, this can be done in a secure environment which improves the security of the design on the FPGA. To further enable this security, current devices offer security modes which disable accesses through the programming interface once the device is programmed.

There are also significant disadvantages to this programming technology. In particular, since anti-fuse-based FPGAs require a non-standard CMOS process, they are typically well behind in the manufacturing processes that they can adopt compared to SRAM-based FPGAs. Finally, the one-time programmability of anti-fuses makes it impossible for manufacturing tests to detect all possible faults. Some faults will only be uncovered after programming and, therefore, the yield after programming will be less than SRAM or floating-gate devices.

2.1.3. Logic Block Architecture

The purpose of a logic block in an FPGA is to provide the basic computing and storage elements used in digital logic systems.

In addition to a basic logic block, modern FPGAs contain a heterogeneous mix of different blocks, some of which can only be used for very specific functions, such as dedicated memory blocks or multipliers. These structures are very efficient in implementing specific functions, but are wasted if not used.

In general, published research on logic block architecture tends to model and explore relatively simple basic logic elements taking into account area, speed and power. In contrast, commercial logic blocks have undergone an evolution that typically has led to the development of more complex blocks in an attempt to gain more functionality.

The most relevant trade-off in the study of logical blocks is the area. The efficiency of the FPGA area is a key metric because the size of the FPGA dictates a significant part of its cost, especially for devices with a large logical capacity.

In general, logic blocks are also called Configurable Logic Blocks (CLBs). Each logic block contains a group of Basic Logic Elements (BLEs), shows in Figure 2.5, where each BLE is formed by a LUT and an Flip-Flop (FF), as shown in figure 2.6.

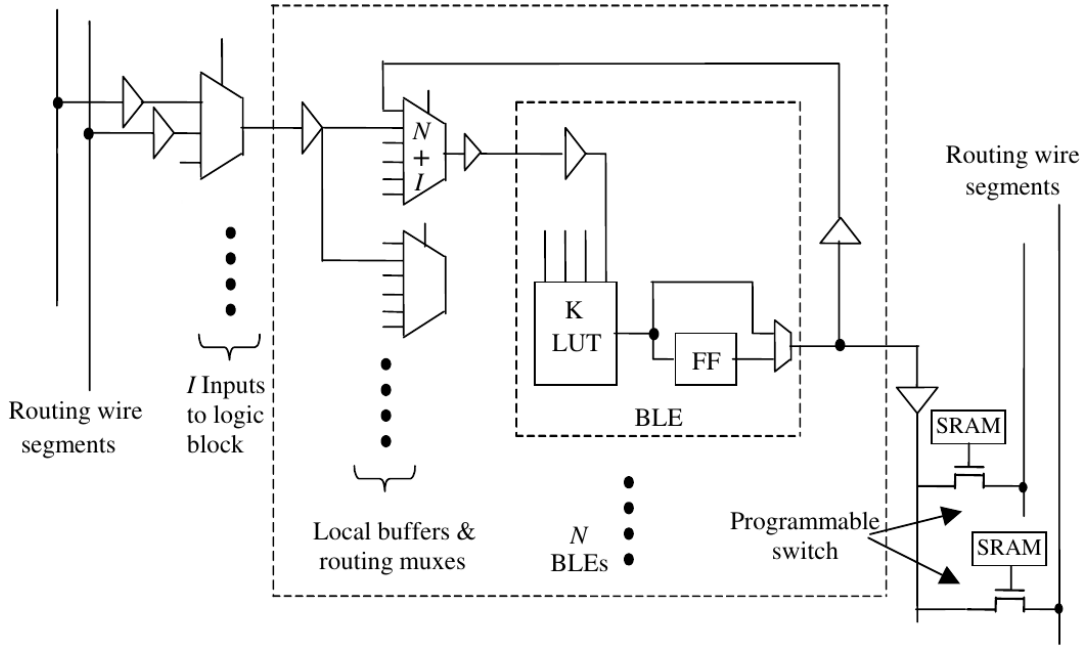


Figure 2.5: Configurable Logic Block

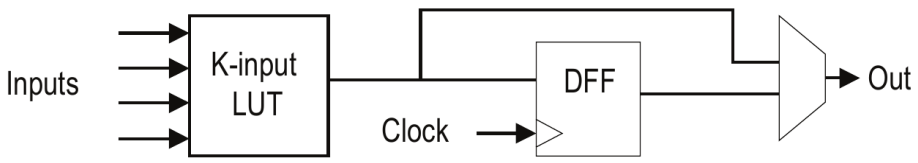


Figure 2.6: Basic Logic Element

2.1.3.1. Heterogeneity

As said, modern FPGA include different types of logical blocks. We give two definitions:

- **soft logic fabric**: the array of combinational logic elements, each consisting of a logic function implemented as a gate or LUT, that is connected through a programmable routing fabric.
- **hard circuit structure**: any other circuitry employed in the device which we define as a structure that allows the implementation of a specific logic function, such as a dedicated FF inside a logic block.

Modern commercial FPGAs contain dedicated logic within each general purpose block to support arithmetic carry and sum functions and some memory functions.

We distinguish between two kinds of heterogeneity:

- **soft logic fabric**: it refers to the heterogeneity within the soft logic fabric such as the FF and dedicated carry logic that appear alongside the combinational logic in every logic block that makes up the soft logic fabric.
- **tile-based heterogeneity**: distinct tiles containing dedicated hard circuit structures are added to the array of tiles. For example, multi-bit Block RAMs (BRAMs) or Multiply-Accumulate (MAC) blocks that appear in modern FPGAs are common hard circuit structures.

Figure 2.7 illustrates an FPGA with a mixture of different blocks with tile-based heterogeneity.

Soft Fabric Heterogeneity As said Soft Fabric Heterogeneity contains a set of heterogeneous elements. Examples of elements are:

- **FF**: all commercial FPGAs have included FFs in their basic logic elements. Modern FPGA FFs are typically edge-triggered and include a variety of set, reset, load, enable and clocking capabilities. Research investigated the area efficiency of FPGAs with and without dedicated FFs, and clearly established the significant benefits of including FF circuits within logic elements.
- **circuitry for arithmetic operations**: Many modern FPGAs include explicit circuitry for addition/subtraction/carry logic to make adders and subtraction units smaller and faster.

Memory Since different applications will need memory configured in many different ways, basic memory blocks must be flexible and configurable. All contemporary FPGAs include memory blocks and they have grown to cover a significant fraction of the FPGA die area.

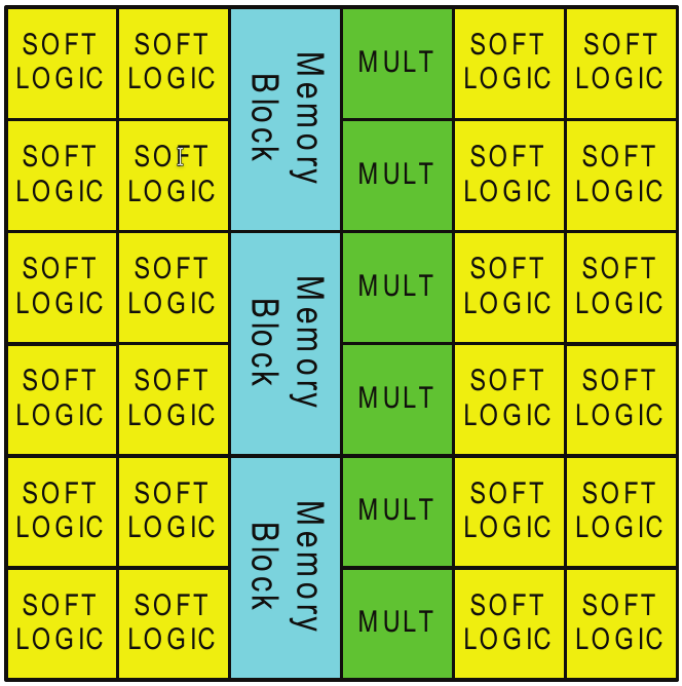


Figure 2.7: Illustration of tile based heterogeneity

Contemporary BRAMs provide the dual port functionality, i.e. they allow simultaneous read and write operations.

Others recent FPGAs have special features such as the ability to support First In First Out (FIFO) configurations. The cost of this added flexibility requires additional memory ports and internal memory complexity. Complex memory operation can be supported with control circuits implemented in soft logic. Often, it can be costly to combine memory blocks to perform large, complicated memory functions due to logic and routing overheads.

Computation-Oriented Tiles A popular industrial example of this type of block is MAC, it is also called Digital Signal Processing (DSP).

If multipliers are not needed by an application, the multiplier tiles located inside a target FPGA provide little benefit. One way to deal with this issue is to create multiple sub-families within a device family that use different ratios of soft logic to hard-logic. A device family typically consists of a set of FPGAs with the same basic architecture that contain differing amounts of resources. A designer can select the device with the most appropriate ratio, minimizing wasted computational tiles.

Microprocessors Microprocessors are vital components in many digital systems. Since they are often used in conjunction with FPGA logic, it makes sense to consider their integration into an FPGA logic fabric.

A challenging aspect of including a hard processor on an FPGA is the development of the interfaces between the processor, memory system, and the soft fabric. Nowadays, the most used communication protocols is Advanced eXtensible Interface (AXI).

The alternative to a hard processor is a soft processor, built out of the soft fabric and other hard logic. The latter is generally slower in performance and larger in terms of area. However, the soft processor can often be customized to exactly suit the needs of the application to gain back some of the lost performance and area-efficiency.

2.1.4. Routing Architecture

The programmable routing in an FPGA provides connections among logic blocks and I/O blocks to complete a user-designed circuit. It consists of wires and programmable switches that form the desired connections. Additionally, circuits also contain a number of signals such as clocks and resets that must be widely distributed across the FPGA. Modern FPGAs all contain dedicated interconnect networks that handle the distribution of these signals. Typically, these networks are carefully designed to be low skew for use in distributing clock signals

We distinguish between two kinds of routing:

- **global routing**: defines the relative position of routing channels in relation to the positioning of logic blocks, how each channel connects to other channels, and the number of wires in each channel.
- **detailed routing**: specifies the lengths of the wires, and the specific switching quantity and patterns between and among wires and logic block pins.

FPGA global routing architectures can be characterized as either hierarchical or island-style. Currently, most commercial SRAM-based FPGA architectures use island-style architectures.

2.1.4.1. Hierarchical-style

Hierarchical routing architectures, shows in Figure 2.8 separate FPGA logic blocks into distinct groups. Connections between logic blocks within a group can be made using wire segments at the lowest level of the routing hierarchy. Connections between logic blocks in distant groups require the traversal of one or more levels (of the hierarchy) of routing segments.

The advantage of this placement of logical elements is that the predictability of block delays is higher. The disadvantages, however, are present in design mapping and scalability.

For these reasons, most recent commercial FPGA routing architectures do not use this type of global routing architecture and, instead, use only one level of hierarchy to create a flat, island-style global routing architecture.

2.1.4.2. Island-style

As shown in Figure 2.9, island-style FPGAs logic blocks are arranged in a two dimensional mesh with routing resources evenly distributed throughout the mesh.

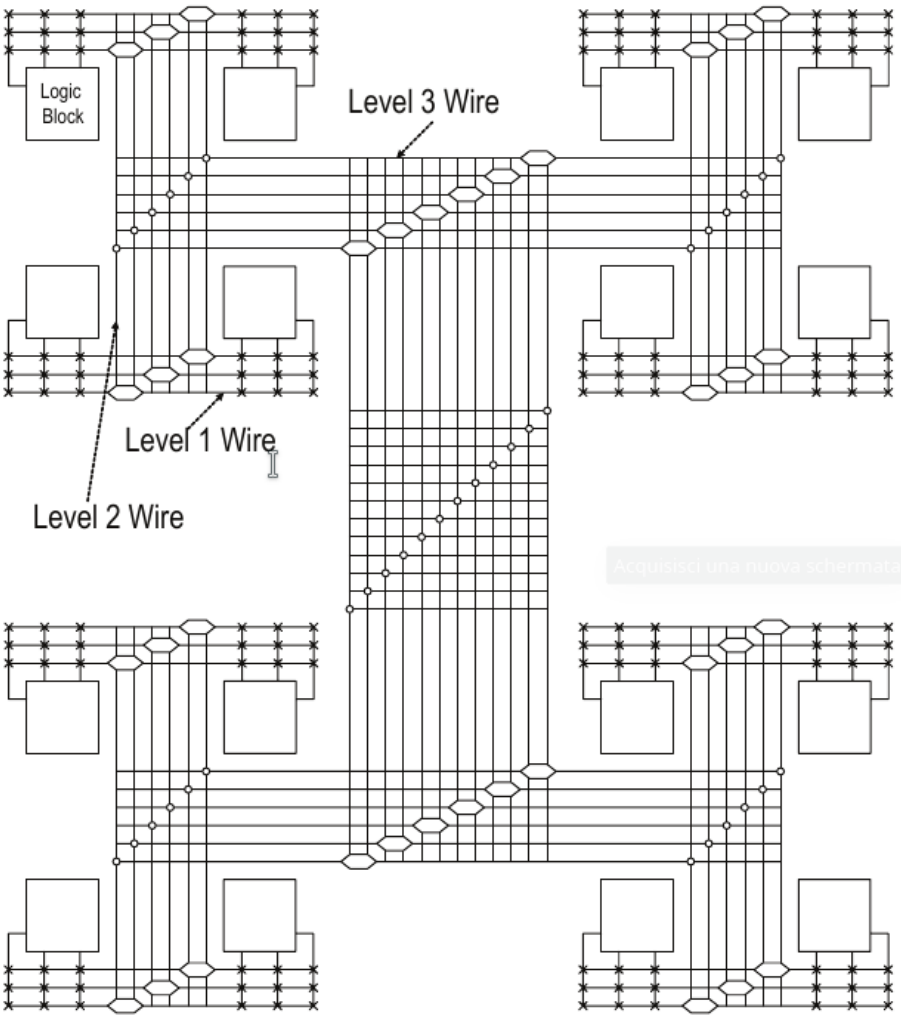


Figure 2.8: Hierarchical-style FPGA

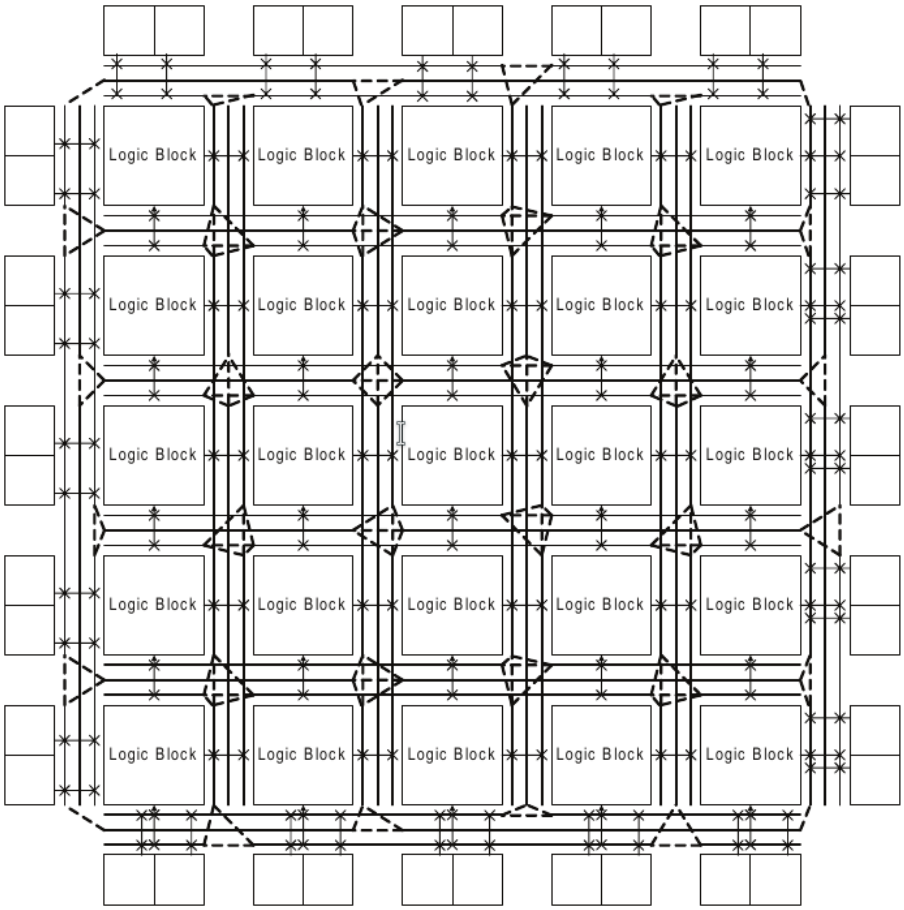


Figure 2.9: Island-style FPGA

The logic blocks have routing channels on all four side. The number of wires contained in a channel is pre-set during fabrication and generally lengths of wire segments are different in an attempt to provide the most appropriate length for each given connection.

The advantage of wires of different lengths near the logic blocks is that efficient connections can be formed. The physical layout for each logic block and surrounding routing channels can be optimized to form a single tile. This combined logic and routing tile can be replicated in two dimensions to form the FPGA array. As a result of this regularity, the minimum feasible routing delay between logic blocks can quickly be estimated.

Others component in this type of routing architecture are:

- **switch block**: connects wires in adjacent channels through programmable switches such as pass-transistors or bi-directional buffers.

- **connection block**: connects the wire segments around a logic block to its inputs and outputs, also through programmable switches.

2.1.5. I/O Architecture

In order to communicate with the outside world, a FPGA needs I/O cells. Given the huge number of communication protocols, these blocks are extremely heterogeneous to meet voltage and speed requirements.

The I/O pad and surrounding supporting logic and circuitry as an I/O cell. These cells, along with their supporting peripherals, consume a significant portion of an area of FPGA.

2.1.5.1. Basic I/O Standards

The major challenge in I/O architecture design is the great diversity in I/O standards. For example, different standards may require different input voltage thresholds and output voltage levels. To support these differences, different I/O supply voltages are often needed for each standard. They may also require a reference voltage to compare against the input voltages. Other standards require clamping diodes which allow specific abnormally high or low voltages to be tolerated. Many standards also rely on differential signaling to improve noise immunity and enable increased data transmission speeds. Proper termination is also essential for maintaining signal integrity but different standards have different termination requirements.

2.1.5.2. High-Speed I/O Support

When we talk about high-speed I/O we mean both the interfacing with the memory and the communication between two entities. In both cases, additional circuitry is required to facilitate this high-speed transfer. At a minimum, standards that make use of differential signaling require two I/O cells to be paired together with differential transmitters/receivers.

High-speed communication requires more than the high-speed signaling that these analog features enable. Therefore, contemporary FPGAs frequently include dedicated digital circuits to support higher-level protocols, such as PCIe

High-speed memory interfaces also need special-purpose hardware to accurately capture data flowing between memory chips and the FPGA. Delay-Locked Loops (DLLs) and Phase-Locked Loops (PLLs) are used to adjust the phase of a transfer clock to ensure that data from the external memory is sampled when the data is valid.

2.1.6. Programming

Nowadays, most FPGA vendors provide a set of design tools that allows automatic synthesis and compilation from design specifications in hardware specification languages, such as Verilog or VHDL, all the way down to a bit-stream to program FPGA chips. Examples of tool are **Xilinx Vivado** and **Intel Quartus Prime**.

2.1.6.1. Inputs

Inputs to the design flow typically are:

- **HDL specification of the design**: the most widely used design specification languages are Verilog and VHDL at the Register Transfer Level (RTL) which specify the operations at each clock cycle. Nowadays, there is a general trend to use a high-level languages like C or SystemC, or domain-specific languages, such as MatLab or Simulink to simplify the programming of FPGAs. Using these languages, one can specify the behavior of the design without going through a cycle-accurate detailed description of the design. A behavior synthesis tool is used to generate the RTL specification in Verilog or VHDL.

- **design constraints**: design constraints typically include the expected operating frequencies of different clocks, the delay bounds of the signal path delays from input pads to output pads (I/O delay), from the input pads to registers (setup time), and from registers to output pads (clock-to-output delay). In some cases, delays between some specific pairs of registers may be constrained. Design constraints may also include specifications of so-called false paths and multi-cycle paths. False paths will not be activated during normal circuit operation, and therefore can be ignored; multi-cycle paths refer to signal paths that carry a valid signal every few clock cycles, and therefore have a relaxed timing requirement. Finally, the design constraints may include physical location constraints, which specify that certain logic elements or blocks be placed at certain locations or a range of locations. These location constraints may be specified by the designer, or inherited from the previous design iteration (for making incremental changes), or generated automatically by the physical synthesis tools in the previous design iterations.

- **specification of target FPGA devices**: each FPGA vendor typically provides a wide range of FPGA devices, with different performance, cost, and power trade-offs. The selection of target device may be an iterative process. The designer may start with a low capacity device with a nominal speed-grade. if synthesis effort fails to map the design into the target device or if the synthesis result fails to meet the operating frequency, the designer has to, respectively, either upgrade to a high-capacity device or upgrade to a device with higher speed-grade. In both the cases, the cost of the FPGA device will increase. This clearly underscores the need to have better synthesis tools, since their quality directly impacts the performance and cost of FPGA designs.

2.1.6.2. Fow

A typical FPGA design flow includes the steps and components shown in Figure 2.10.

We now briefly describe each step in the design flow shown in Figure 2.10:

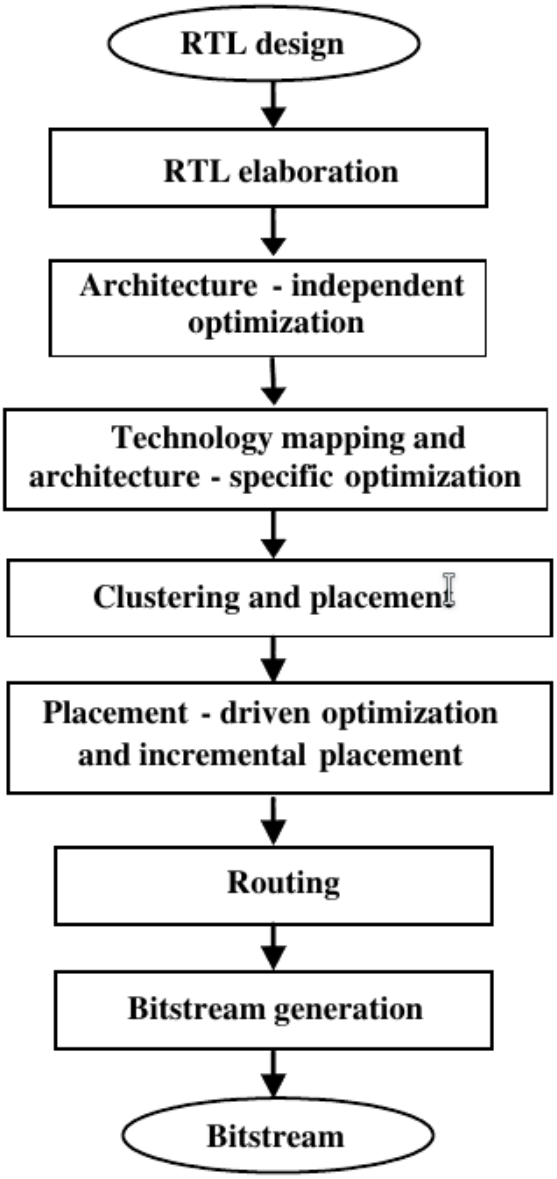


Figure 2.10: A typical FPGA design flow starting from RTL specifications

- **RTL elaboration**: identifies and/or infers *datapath* operations, such as sums, multiplications, register files, and/or memory blocks, and *control logic*, which is elaborated into a set of Finite-State Machine (FSM) and/or generic Boolean networks. It is important to recognize the datapath elements due to special architectural support in modern FPGAs, such as adders with dedicated fast-carry chains and embedded multipliers.

- **Architecture-independent optimization**: this includes both *datapath optimization*, using techniques such as constant propagation, strength reduction, operation sharing, and expression optimization; and *control logic optimization*, which includes both sequential optimization, such as FSM encoding/minimization and retiming, and combinational logic optimization, such as constant propagation, redundancy removal, logic network restructuring and optimization, and don't-care based optimization.

- **Technology mapping and architecture-specific optimization**: maps: (i) the optimized datapath to on-chip dedicated circuit structures, such as on-chip multipliers, adders with dedicated carry-chains, and embedded memory blocks for datapath implementation, and (ii) the optimized control logic to BLEs. Note that datapath operations can be mapped to BLEs as well if the dedicated circuit structures are not available or not convenient to use.

- **Clustering and placement**: determines the location of each element in the mapped netlist. With hierarchical FPGAs a separate clustering step may be performed prior to placement to group BLEs into logic blocks.

- **Placement-driven optimization and incremental placement**: once placement is available, interconnects are defined and may become a performance bottleneck. Further optimization may be carried out in the presence of interconnect delays, including logic restructuring, duplication, rewiring. After such operations, an incremental placement step is needed to validate the placement again. The step of placement-driven optimization is optional, but may improve design performance considerably.

- **Routing**: global routing and detail routing will be performed to connect all signal paths using the available programmable interconnects on-chip.

- **Bit-stream generation**: the final step of the design flow. It takes the mapped, placed, and routed design as input and generates the necessary bit-stream to program the logic and interconnects to implement the intended logic design and layout on the target FPGA device.

2.1.7. Xilinx vs Intel Altera FPGAs

2.1.7.1. Xilinx

The huge variety of FPGAs offered by Xilinx cannot be fully explored in this Section. Therefore, since a **Kintex UltraScale** has been used in this thesis work, the main features of FPGAs based on Ultrascale architecture will be exposed.

The UltraScale architecture is one of the architecture proposed by Xilinx for its FPGAs. UltraScale architecture-based devices address a vast spectrum of high-bandwidth, high-utilization system requirements by using technical innovations, including next-generation routing, ASIC-like clocking, 3D-on-3D ICs, Multiprocessor SoCs (MPSoCs) technologies, and new power reduction features. The devices share many building blocks, providing scalability across process nodes and product families to leverage system-level investment across platforms.

Kintex UltraScale is family devices for packet processing in networking and data centers applications as well as DSP-intensive processing needed in next-generation medical imaging, 8k4k video, and heterogeneous wireless infrastructure. Kintex UltraScale devices is a mid-range device that provides signal processing bandwidth, next-generation transceivers, and low-cost packaging for an optimum blend of capability and cost-effectiveness.

There are two types of slices in the UltraScale architecture, with different ratios of the two types by device: **SLICEL** and **SLICEM**.

SLICEL The **SLICEL** has the LUT and storage element resources, along with the carry logic and wide multiplexers. One SLICEL is included in a CLB. The UltraScale architecture CLBs provide advanced, high-performance, low-power programmable logic with:

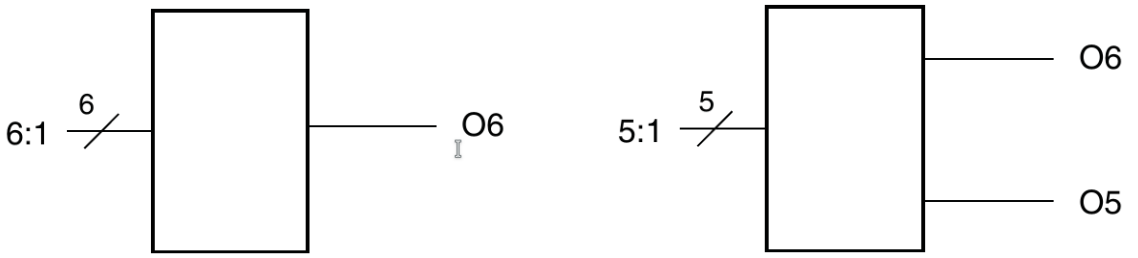
- Real 6-input LUT capability.
- Dual 5-input LUT option.
- Distributed memory and Shift Register Logic (SRL) ability.
- Dedicated high-speed carry logic for arithmetic functions.
- Wide multiplexers for efficient utilization.
- Dedicated storage elements that can be configured as FFs or latches with flexible control signals.

Each slice provides eight 6-input LUTs and sixteen FFs. The slices and their CLBs are arranged in columns throughout the device, with the size and number of columns increasing with density. The UltraScale architecture LUTs can be configured as shown in Figure 2.11.

Each LUT output can connect to slice outputs, or optionally be registered in a FF or latch. The storage elements can also be driven by direct inputs to the slice (X and I), or by the results of the internal carry logic or wide multiplexers, as shown in Figure 2.12. The storage elements have a clock enable input, along with an initialization signal that can be programmed as either synchronous or asynchronous, and as set or reset.

Carry logic consists of dedicated carry-lookahead gates, multiplexers, and routing that are independent of the general-purpose logic resources while providing both higher density and increased performance. Carry logic is often inferred for smaller arithmetic functions.

Finally, a SLICEL is represented in Figure 2.13.



(a) One Function of 6 Inputs **(b)** Two Functions of 5 Inputs

Figure 2.11: LUT configuration

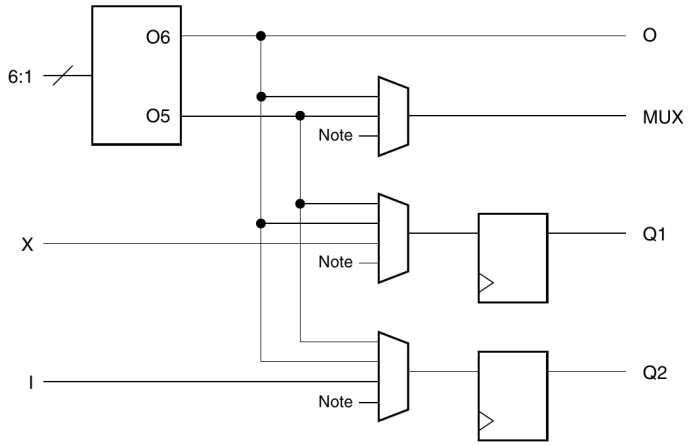


Figure 2.12: Xilinx SLICEL I/O connection

SLICEM A **SLICEM** use the LUTs as distributed 64-bit Random Access Memory (RAM), by adding a separate Write Address (WA), Write Enable (WE), and clock signal. The LUT can be configured as either a $64x1$ or $32x2$ memory, as shown in Figure 2.14. The direct inputs X and I serve as the data inputs.

The distributed RAM can be combined across the eight LUTs in the SLICEM to create memories of up to 512 bits. The SLICEM shares a common write address and write clock across all 8 LUTs. The SLICEM write enable is also shared but can be used in combination with three other slice inputs for more flexibility.

Each LUT in a SLICEM can also be used as a 32-bit shift register (SRL32). Combining the LUTs allows up to a 256-bit shift register in one SLICEM, compared to the 16 dedicated FFs per slice.

DSP block The UltraScale devices have many dedicated low-power DSP slices, combining high speed with small size while retaining system design flexibility. The DSP resources enhance the speed and efficiency of many applications beyond digital signal processing, such as wide dynamic bus shifters, memory address generators, wide bus multiplexers, and memory-mapped I/O registers. The DSP slice in the UltraScale architecture is defined using the **DSP48E2** primitive and the slice is referred to as either DSP or **DSP48E2** in the Xilinx tools. The scheme of the DSP48E2 slice is shown in Figure 2.15.

The DSP slice functionality include:

- $27x18$ two's complement multiplier with dynamic bypass

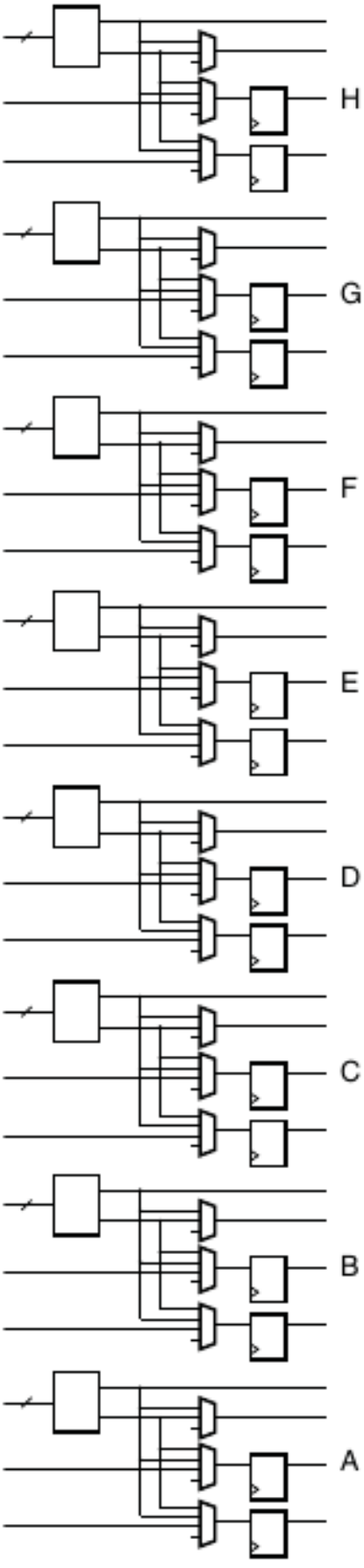
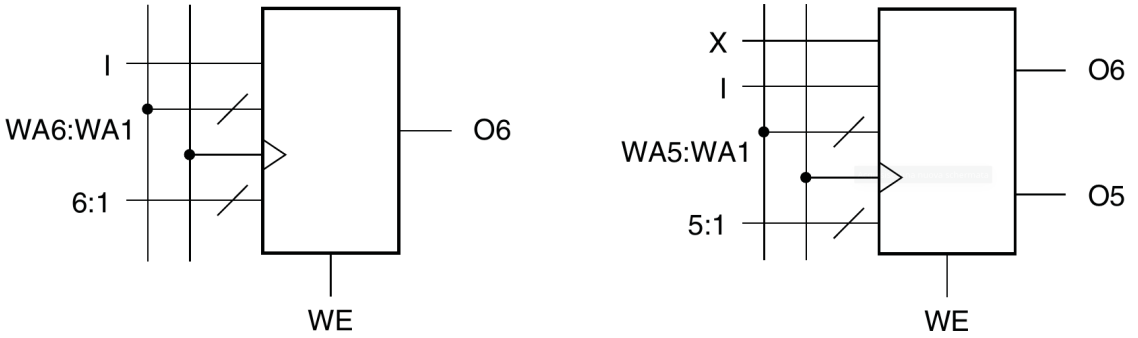


Figure 2.13: Xilinx SLICEL architecture



(a) 64x1 Distributed RAM **(b)** 32x2 Distributed RAM

Figure 2.14: LUT configuration

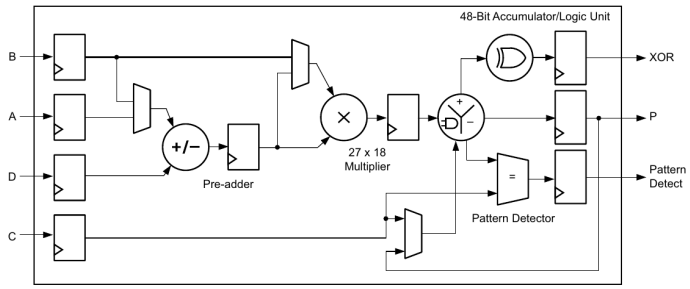


Figure 2.15: Xilinx DSP architecture

- Power saving 27-bit pre-adder: optimizes symmetrical filter applications and reduces DSP logic requirements
- 48-bit accumulator that can be cascaded to build 96-bit and larger accumulators, adders, and counters
- Single Instruction Multiple Data (SIMD) arithmetic unit: dual 24-bit or quad 12-bit add/subtract/accumulate
- 48-bit logic unit: bitwise AND, OR, NOT, NAND, NOR, XOR, and XNOR
- Pattern detector: terminal counts, overflow/underflow, convergent/symmetric rounding support, and 96-bit wide AND/NOR when combined with logic unit
- Optional pipeline registers and dedicated buses for cascading multiple DSP slices in a column for hierarchical/composite functions like Systolic Finite Impulse Response (FIR) filters

Applications of the DSP slice include:

- Fixed and floating point Fast Fourier Transform (FFT) functions
- Systolic FIR filters
- MultiRate FIR filters
- Cascaded Integrator-Comb (CIC) filters
- Wide real/complex multipliers/accumulators

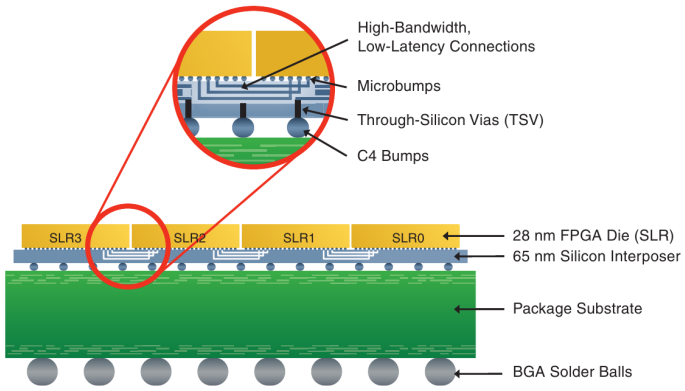


Figure 2.16: Xilinx FPGA enabled by SSI Technology

SSI technology Since the FPGA chosen for this thesis work uses Stacked Silicon Interconnect (SSI) technology, we introduce what this technology is.

The growing demand for FPGAs with high capacity and high bandwidth has led Xilinx to define a new technology to meet these needs with the second generation of the pioneering 3D SSI technology.

SSI technology uses passive silicon interposers with microbumps and Through-Silicon Vias (TSV) to combine multiple highly manufacturable FPGA die slices, referred to as SRLs, in a single package. The technology also allows die of different types or silicon processes to be interconnected on the interposer. This type of construction is referred to as a heterogeneous FPGA. Figure 2.16 shows the side view of the die stack-up with four FPGA SRLs, silicon interposer, and package substrate.

2.1.7.2. Intel Altera

Logic fabric The logical fabric of Intel Altera FPGAs consists of logical blocks called Adaptive Logic Module (ALM). As we can see in Figure 2.17, it consists of combinational logic, two registers, and two adders. The combinational portion has eight inputs and includes a LUT that can be divided between two Adaptive LUTs (ALUTs) using Altera's patented LUT technology. To implement an arbitrary six-input function, ALM is needed, but because it has eight inputs to the combinational logic block, one ALM can implement various combinations of two functions.

Building LUTs Since the type of FPGA produced by Intel Altera is SRAM-based programming technology, a LUT consists of SRAMs to hold the the configuration memory LUT-mask and a set of multiplexers to select the configuration bit to drive the output. Remember, to implement a k-input LUT, i.e. a LUT that can implement any k-input function, 2^k SRAM bits and a 2^k : 1 multiplexer are needed. Figure 2.18 shows an example of 4-LUT, which consists of 16 bits of SRAM and a 16 : 1 multiplexer implemented as a tree of 2 : 1 multiplexers.

DSP block On Stratix II-IV devices the block DSP, shown in Figure 2.19, consists of four 18x18 bit multipliers (signed or unsigned) and an adder tree with

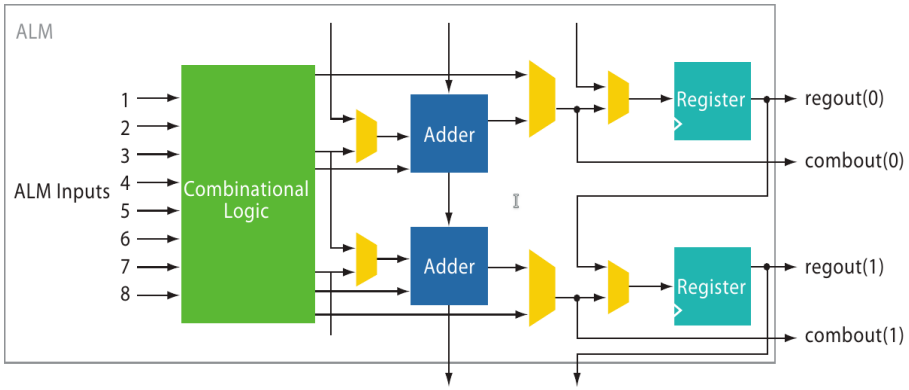


Figure 2.17: ALM Block Diagram

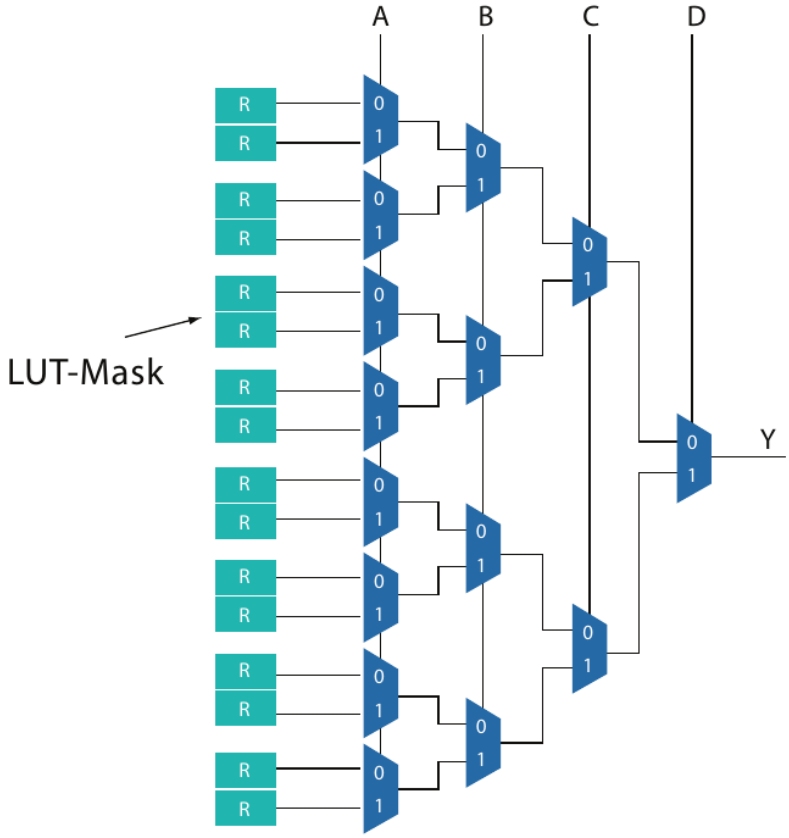


Figure 2.18: Example of 4-LUT

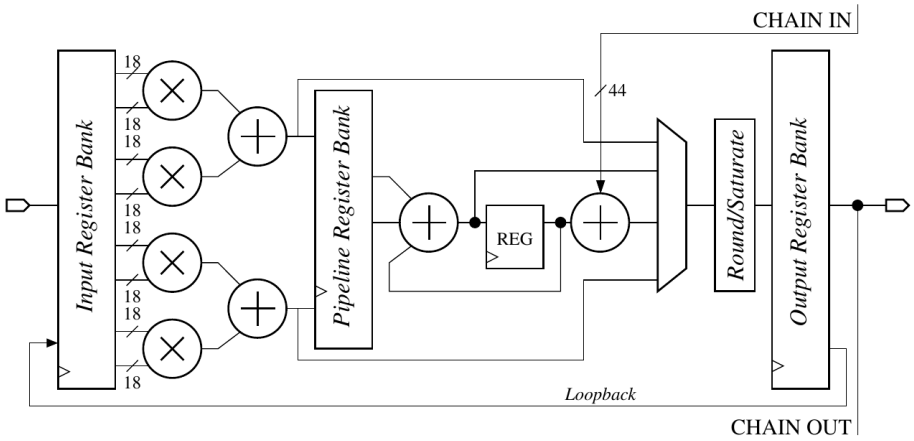


Figure 2.19: Intel Altera DSP block

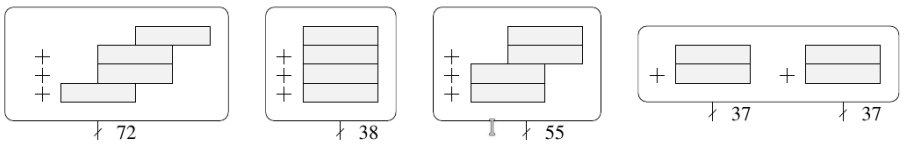


Figure 2.20: Intel Altera DSP configuration

different possible configurations, represented in Figure 2.20. Stratix III-IV calls these DSPs half-DSPs, and pack two of them into a block DSP. In these devices, the limiting factor in terms of configurations is the number of I/O in the DSP block. In addition, all DSPs allow various sum-of-two/four modes for greater versatility. Here also, adjacent DSP blocks can be cascaded, internal registers allow high-frequency pipelining and a loopback path allows for accumulation. These cascading chains reduce resource consumption, but also latency: a sum-of-two 27-bit multipliers can be clocked at the nominal DSP speed in just 2 cycles.

Routing architecture The routing architecture provides the connectivity between different clusters of logic blocks, called Logic Array Blocks (LABs), and can be measured by the number of *hops* required to get from one LAB to another. The fewer the number of hops and more predictable the pattern, the better the performance and the easier it is for Computer-Aided Design (CAD) tool optimization.

The Stratix and Stratix II families use a three-sided routing architecture as shown in Figure 2.21. This means that a LAB can drive or listen to all of the wires on one horizontal channel above it and two vertical channels to the left and right side of it. The channels contain wires of length 4, 8, 16, and 24, and signals can get off at any LAB along the length of the wire.

2.2 Partial Reconfiguration

Partial Reconfiguration (PR) takes the flexibility offered by FPGA one step further, allowing the modification of an operating FPGA design by loading a partial configuration file, usually a partial bitstream file. After a full bitstream file configures the FPGA, partial bitstream files can be downloaded to modify reconfig-

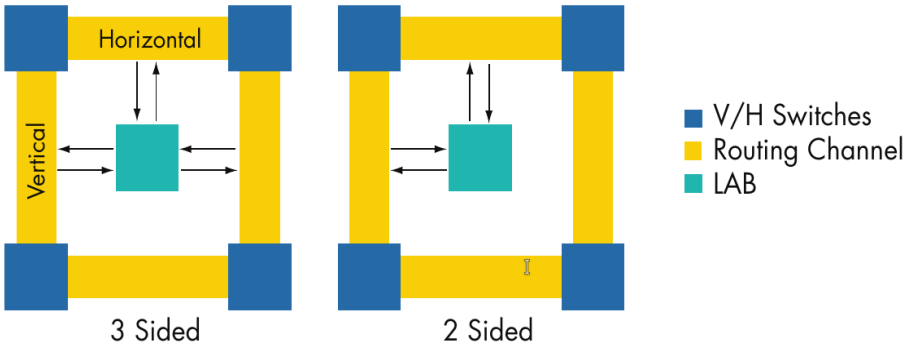


Figure 2.21: Rounting of Intel Altera FPGAs

urable regions in the FPGA without compromising the integrity of the applications running on those parts of the device that are not being reconfigured.

There are many reasons why the ability to time multiplex hardware dynamically on a single FPGA is advantageous. These include:

- Reducing the size of the FPGA needed to implement a given function, with consequent reductions in cost and power consumption.
- Providing flexibility in the choices of algorithms or protocols available to an application.
- Enabling new techniques in design security
- Improving FPGA fault tolerance
- Accelerating configurable computing

In addition to reducing size, weight, power and cost, PR enables new types of FPGA designs that would be otherwise impossible to implement.

We can find examples of partial reconfiguration application in networks, cryptography and HPC fields.

2.2.1. Vivado Partial Reconfiguration design flow

The **Vivado Partial Reconfiguration design flow** is similar to a standard design flow, with some notable differences. The implementation software automatically manages the low-level details to meet silicon requirements. It is needed to guide to define the design structure and floorplan. The following steps summarize processing a PR design:

1. Synthesize the static and Reconfigurable Modules (RMs) separately.
2. Create physical constraints (Physical Blocks (Pblocks)) to define the reconfigurable regions.
3. Set the *HD.RECONFIGURABLE* property on each Reconfigurable Partition (RP).

4. Implement a complete design (static and one RM per RP) in context.
5. Save a design checkpoint for the full routed design.
6. Remove RMs from this design and save a static-only design checkpoint.
7. Lock the static placement and routing.
8. Add new RMs to the static design and implement this new configuration, saving a checkpoint for the full routed design.
9. Repeat Step 8 until all RMs are implemented.
10. Run a verification utility (*pr_verify*) on all configurations.
11. Create bitstreams for each configuration.

2.3 OpenCL introduction

Most information in this Section has been extracted from [12].

OpenCL is an open royalty-free standard for general-purpose parallel programming across CPUs, GPUs and other processors, giving software developers portable and efficient access to the power of these heterogeneous processing platforms.

OpenCL supports a wide range of applications, ranging from embedded and consumer software to HPC solutions, through a low-level, high-performance, portable abstraction. By creating an efficient, close-to-the-metal programming interface, OpenCL will form the foundation layer of a parallel computing ecosystem of platform-independent tools, middleware and applications.

OpenCL is particularly suited to play an increasingly significant role in emerging interactive graphics applications that combine general parallel compute algorithms with graphics rendering pipelines. OpenCL consists of an Application Programming Interface (API) for coordinating parallel computation across heterogeneous processors; and a cross-platform programming language with a well-specified computation environment.

2.3.1. Platform Model

The Platform Model for OpenCL is defined in Figure 2.22. The model consists of a host connected to one or more OpenCL devices. An OpenCL device is divided into one or more Compute Units (CUs) which are further divided into one or more Processing Elements (PEs). Computations on a device occur within the PEs.

An OpenCL application runs on a host according to the models native to the host platform. The OpenCL application submits commands from the host to execute computations on the processing elements within a device. The processing elements within a CU execute a single stream of instructions as SIMD units (execute in lockstep with a single stream of instructions) or as Single Program Multiple Data (SPMD) units (each PE maintains its program counter).

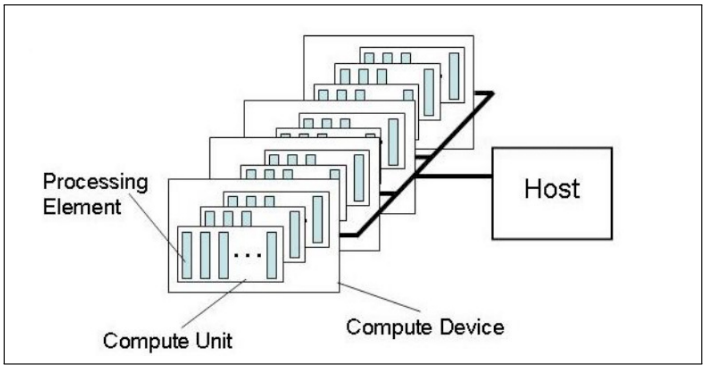


Figure 2.22: The logical view of OpenCL Platform Model

2.3.2. Memory Model

Work-item(s) executing a kernel, have access to four distinct memory regions:

- **Global Memory**: this memory region permits read/write access to all work-items in all work-groups. Work-items can read from or write to any element of a memory object. Reads and writes to global memory may be cached depending on the capabilities of the device.
- **Constant Memory**: a region of global memory that remains constant during the execution of a kernel. The host allocates and initializes memory objects placed into constant memory.
- **Local Memory**: a memory region local to a work-group. This memory region can be used to allocate variables that are shared by all work-items in that work-group. It may be implemented as dedicated regions of memory on the OpenCL device. Alternatively, the local memory region may be mapped onto sections of the global memory.
- **Private Memory**: a region of memory private to a work-item. Variables defined in one work-item's private memory are not visible to another work-item.

The memory regions and how they relate to the platform model are described in Figure 2.23.

The application running on the host uses the OpenCL API to create memory objects in global memory and to enqueue memory commands that operate on these memory objects.

The host and OpenCL device memory models are, for the most part, independent of each other. This is necessary given that the host is defined outside OpenCL. They, however, at times need to interact. This interaction occurs in one of two ways: by explicitly copying data or by mapping and unmapping regions of a memory object.

To copy data explicitly, the host enqueues commands to transfer data between the memory object and host memory. These memory transfer commands may be blocking or non-blocking. The OpenCL function call for a blocking memory

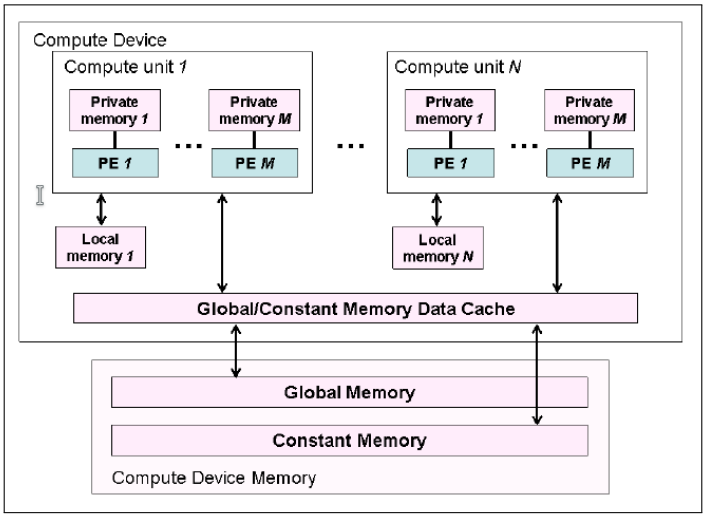


Figure 2.23: The logical view of OpenCL Memory Model

transfer returns once the associated memory resources on the host can be safely reused. For a non-blocking memory transfer, the OpenCL function call returns as soon as the command is enqueued regardless of whether host memory is safe to use.

The mapping/unmapping method of interaction between the host and OpenCL memory objects allows the host to map a region from the memory object into its address space. The memory map command may be blocking or non-blocking. Once a region from the memory object has been mapped, the host can read or write to this region. The host unmaps the region when accesses (reads and/or writes) to this mapped region by the host are complete.

2.3.2.1. Memory Consistency

OpenCL uses a relaxed consistency memory model. In other words, the state of memory visible to a work-item is not guaranteed to be consistent across the collection of work-items at all times.

Within a work-item memory has load/store consistency. Local memory is consistent across work-items in a single work-group at a work-group barrier.

Global memory is consistent across work-items in a single work-group at a work-group barrier, but there are no guarantees of memory consistency between different work-groups executing a kernel.

Memory consistency for memory objects shared between enqueued commands is enforced at a synchronization point.

2.3.3. Execution Model

Execution of an OpenCL program occurs in two parts: kernels that execute on one or more OpenCL devices and a host program that executes on the host. The host program defines the context for the kernels and manages their execution.

The core of the OpenCL execution model is defined by how the kernels execute. When a kernel is submitted for execution by the host, an index space is defined. An instance of the kernel executes for each point in this index space. This kernel instance is called a work-item and is identified by its point in the index space, which provides a global ID for the work-item. Each work-item executes the same code but the specific execution pathway through the code and the data operated upon can vary per work-item.

Work-items are organized into work-groups. The work-groups provide a more coarse-grained decomposition of the index space. Work-groups are assigned a unique work-group ID with the same dimensionality as the index space used for the work-items. Work-items are assigned a unique local ID within a work-group so that a single work-item can be uniquely identified by its global ID or by a combination of its local ID and work-group ID. The work-items in a given work-group execute concurrently on the processing elements of a single compute unit.

The index space supported in OpenCL is called an **NDRange**. An NDRange is an N-dimensional index space, where N is one, two or three. An NDRange is defined by an integer array of length N specifying the extent of the index space in each dimension starting at an offset index F (zero by default). Each work-item's global ID and local ID are N-dimensional tuples. The global ID components are values in the range from F, to F plus the number of elements in that dimension minus one.

Work-groups are assigned IDs using a similar approach to that used for work-item global IDs. An array of length N defines the number of work-groups in each dimension. Work-items are assigned to a work-group and given a local ID with components in the range from zero to the size of the work-group in that dimension minus one. Hence, the combination of a work-group ID and the local-ID within a work-group uniquely defines a work-item. Each work-item is identifiable in two ways; in terms of a global index, and terms of a work-group index plus a local index within its work-group.

2.3.3.1. Context and Command Queues

The host defines a context for the execution of the kernels. The context includes the following resources:

- **Devices**: the collection of OpenCL devices to be used by the host.
- **Kernels**: the OpenCL functions that run on OpenCL devices.
- **Program Objects**: the program source and executable that implement the kernels.
- **Memory Objects**: a set of memory objects visible to the host and the OpenCL devices. Memory objects contain values that can be operated on by instances of a kernel.

The context is created and manipulated by the host using functions from the OpenCL API. The host creates a data structure called a command-queue to co-

ordinate execution of the kernels on the devices. The host places the commands into the command-queue which are then scheduled onto the devices within the context. These include:

- **Kernel execution commands**: execute a kernel on the processing elements of a device.
- **Memory commands**: transfer data to, from, or between memory objects, or map and unmap memory objects from the host address space.
- **Synchronization commands**: constrain the order of execution of commands.

The command-queue schedules the commands for execution on a device. These execute asynchronously between the host and the device. Commands execute relative to each other in one of two modes:

- **In-order Execution**: commands are launched in the order they appear in the command-queue and complete in order. In other words, a prior command on the queue completes before the following command begins. This serializes the execution order of commands in a queue.
- **Out-of-order Execution**: commands are issued in order, but do not wait to complete before following commands execute. Any order can be enforced by the programmer through explicit synchronization commands.

Kernel execution and memory commands submitted to a queue generate event objects. These are used to control execution between commands and to coordinate execution between the host and the devices.

It is possible to associate multiple queues with a single context. These queues run concurrently and independently with no explicit mechanisms within OpenCL to synchronize between them.

2.3.3.2. Categories of Kernels

The OpenCL execution model supports two categories of kernels:

- **OpenCL kernels** are written with the OpenCL C programming language and compiled with the OpenCL compiler. All OpenCL implementations support OpenCL kernels. Implementations may provide other mechanisms for creating OpenCL kernels.
- **Native kernels** are accessed through a host function pointer. Native kernels are queued for execution along with OpenCL kernels on a device and share memory objects with OpenCL kernels. For example, these native kernels could be functions defined in application code or exported from a library. Note that the ability to execute native kernels is an optional functionality within OpenCL and the semantics of native kernels are implementation-specific. The OpenCL API includes functions to query capabilities of a device and determine if this capability is supported.

2.3.4. Programming Model

The OpenCL execution model supports data-parallel and task-parallel programming models, as well as supporting hybrids of these two models. The primary model driving the design of OpenCL is data-parallel.

2.3.4.1. Data-Parallel Programming Model

A data-parallel programming model defines a computation in terms of a sequence of instructions applied to multiple elements of a memory object. The index space associated with the OpenCL execution model defines the work-items and how the data maps onto the work-items. In a strictly data-parallel model, there is a one-to-one mapping between the work-item and the element in a memory object over which a kernel can be executed in parallel. OpenCL implements a relaxed version of the data-parallel programming model where a strict one-to-one mapping is not a requirement.

OpenCL provides a hierarchical data-parallel programming model. There are two ways to specify the hierarchical subdivision. In the explicit model, a programmer defines the total number of work-items to execute in parallel and also how the work-items are divided among work-groups. In the implicit model, a programmer specifies only the total number of work-items to execute in parallel, and the division into work-groups is managed by the OpenCL implementation.

2.3.4.2. Task Parallel Programming Model

The OpenCL task parallel programming model defines a model in which a single instance of a kernel is executed independently of any index space. It is logically equivalent to executing a kernel on a CU with a work-group containing a single work-item. Under this model, users express parallelism by:

- using vector data types implemented by the device,
- enqueuing multiple tasks, and/or
- enqueuing native kernels developed using a programming model orthogonal to OpenCL.

2.3.4.3. Synchronization

There are two domains of synchronization in OpenCL:

- Work-items in a single work-group
- Commands enqueued to command-queue(s) in a single context

Synchronization between work-items in a single work-group is done using a work-group barrier. All the work-items of a work-group must execute the barrier before any is allowed to continue execution beyond the barrier. Note that the

work-group barrier must be encountered by all work-items of a work-group executing the kernel or by none at all. There is no mechanism for synchronization between work-groups.

The synchronization points between commands in command-queues are:

- Command-queue barrier. The command-queue barrier ensures that all previously queued commands have finished execution and any resulting updates to memory objects are visible to subsequently enqueued commands before they begin execution. This barrier can only be used to synchronize between commands in a single command-queue.
- Waiting for an event. All OpenCL API functions that enqueue commands return an event that identifies the command and memory objects it updates. A subsequent command waiting on that event is guaranteed that updates to those memory objects are visible before the command begins execution.

2.3.5. OpenCL Framework

The OpenCL framework allows applications to use a host and one or more OpenCL devices as a single heterogeneous parallel computer system. The framework contains the following components:

- **OpenCL Platform layer**: the platform layer allows the host program to discover OpenCL devices and their capabilities and to create contexts.
- **OpenCL Runtime**: the runtime allows the host program to manipulate contexts once they have been created.
- **OpenCL Compiler**: the OpenCL compiler creates program executables that contain OpenCL kernels. The OpenCL C programming language implemented by the compiler supports a subset of the ISO C99 language with extensions for parallelism.

2.3.6. OpenCL Devices and FPGAs

In the context of CPU and GPU devices, the attributes of a device are fixed and the programmer has very little influence on what the device looks like. On the other hand, this characteristic of CPU/GPU systems makes it relatively easy to obtain an off-the-shelf board, i.e. this devices are ready to use without any type of changes. The major limitation of this type of device is that there is no direct connection between system I/O and the OpenCL kernels. All transactions of data are through memory-based transfers.

An OpenCL device for an FPGA is not limited by the constraints of a CPU/GPU device. By taking advantage of the fact that the FPGA starts off as a blank computational canvas, the user can decide the level of device customization that is appropriate to support a single application or a class of applications. In determining the level of customization in a device, the programmer needs to keep in

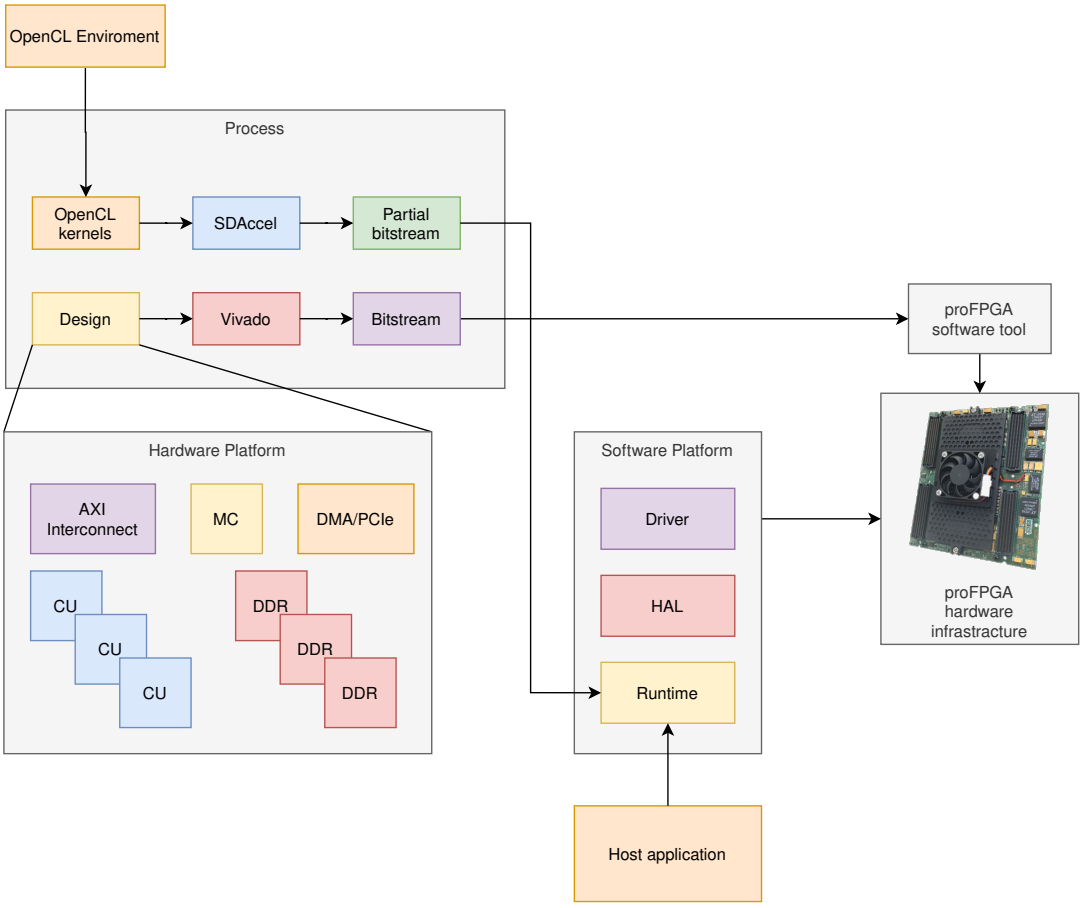


Figure 2.24: Entire system

mind that kernel compute units are not placed in isolation within the FPGA fabric.

FPGA devices capable of supporting OpenCL programs can include, but are not limited to, the following components:

- Direct Memory Access (DMA) engines
- I/O peripherals such as PCIe and Ethernet
- Memory controllers
- Custom interconnects
- OpenCL compute units
- RTL-based accelerators

2.4 SDAccel Platform

Platform, in this thesis work, is understood as the infrastructure, both hardware and software, capable of accelerating an application. Figure 2.24 shows the entire system useful to accelerating an application.

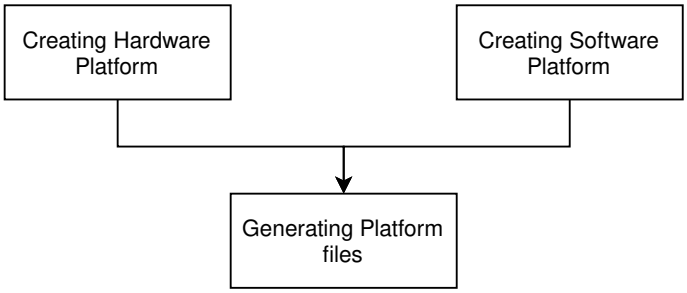


Figure 2.25: Creation Platform flow

In Figure 2.25 we show the flow for creating a Platform. In this Section, concepts for both hardware and software will be introduced.

2.4.1. Hardware Platform

The term **Hardware Platform** represents the physical board interface and the Programmable region. The Hardware Platform consists of a Vivado Intellectual Property (IP) Integrator design with a target device and all interface IPs configured and connected to the device I/Os and the Programmable region. It also contains the interface representation of the Programmable region. Figure 2.26 shows an example of the Hardware Platform. as we can see, the components used are:

- **DMA/PCIe Controller**: perform direct memory transfers, both DMA Read Channel (H2C), and DMA Write Channel (C2H). It is used to send command and data to our design.
- **AXI Interconnect**: allows any mixture of AXI master and slave devices to be connected to it, which can vary from one another in terms of data width, clock domain and AXI sub-protocol (AXI4, AXI3, or AXI4-Lite).
- **Memory Controller**: manages the flow of data going to and from the DDR.
- **DDR**: is the memory of the device. The DDR number can be chosen during the design phase.
- **Compute Unit**: is the calculation unit dedicated to a single kernel.

The Programmable region and Static region is programmed by different tools, as shown in Figure 2.27.

The Static Region represents the fixed logic portion of the programmable device that manages the design state before, during, and after partial reconfiguration of the Programmable region. This logic is not re-implemented with the Programmable region.

The Programmable region describes the partition region that accepts the hardware functions from the SDAccel Development Environment. The term also describes the physical resources available on the programmable device to implement the hardware functions. Special parameters and design considerations are

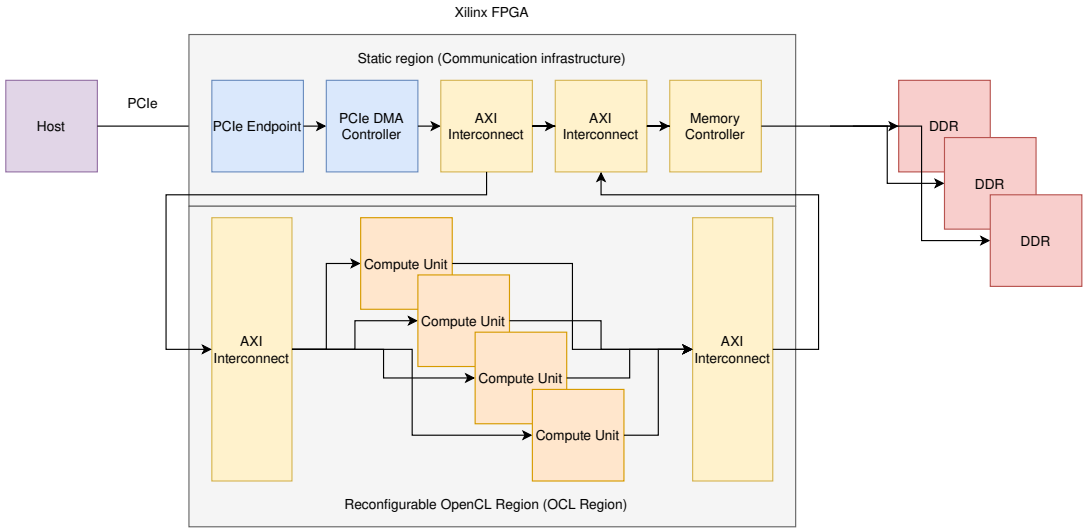


Figure 2.26: Logical view of SDAccel Hardware Platform

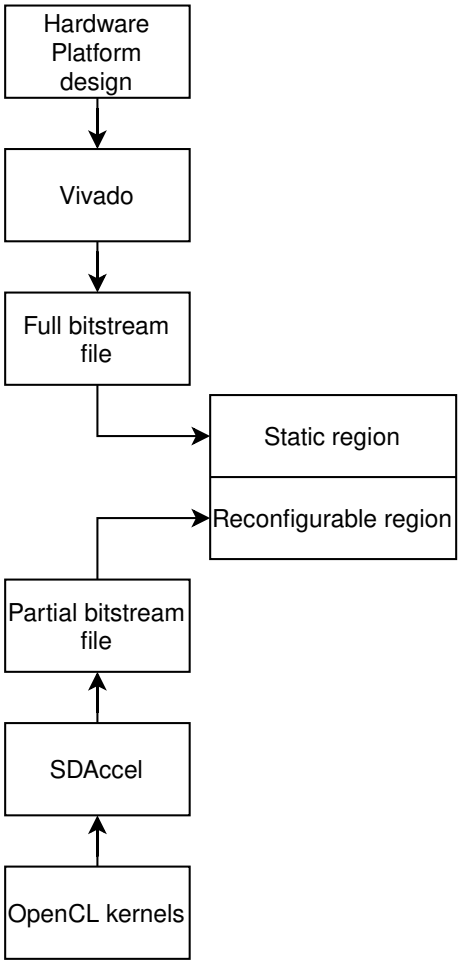


Figure 2.27: Tools used to manage the SDAccel Hardware Platform

required for signals that cross between the Static region and the Programmable region.

Building Xilinx FPGA-based OpenCL devices requires FPGA design expertise and is beyond the scope of SDAccel itself. Devices for SDAccel are created using the Xilinx Vivado design suite for FPGA designers. SDAccel provides pre-defined devices as well as allows users to augment the tool with third party created devices. The devices available in SDAccel are for Virtex-7, Kintex-7, and Kintex-UltraScale devices from Xilinx. These devices are available in a PCIe form factor. The PCIe form factor assumes that the host processor is an x86 based processor and that the FPGA is used for the implementation of compute units.

Nowadays, Xilinx advises against Platform development, both on FPGAs of Xilinx and third-party FPGAs, and has removed any kind of support, including various documentation, due to the introduction of **Alveo** acceleration boards.

2.4.1.1. OpenCL Region

An SDAccel device contains a customization area called the OpenCL Region (OCL Region). Although not defined in the OpenCL standard, the OCL Region is an important concept in SDAccel. The compute units generated from user kernel functions are placed in this region. These compute units are highly specialized to execute a single kernel function and internally contain parallel execution resources to exploit work-group level parallelism. By placing multiple compute units of the same type in the OCL Region, developers can easily scale the performance of single kernels across larger NDRange sizes. By placing multiple compute units of different types in the OCL Region, developers can leverage task parallelism between disparate kernels. In this way, the massive amounts of parallelism available in the FPGA device can be customized and harnessed by the SDAccel developer. This is different from CPU and GPU implementations of OpenCL which contain a fixed set of general purpose resources.

The OCL Region contains the customized compute units which implement the user-defined accelerator kernels. SDAccel automatically adds the necessary interconnects for these compute units to communicate with the rest of the Platform. Also contained on the FPGA device is a Static region including all the necessary circuitry for communication between host, compute units, and off-chip global memory. This Static region is a pre-defined base Platform which can be flashed onto an Erasable Programmable Read-Only Memory (EPROM) on the board. The FPGA would then be configured with this base Platform upon power-up and is always there and accessible for the user. As shown in Figure 2.26, communication to the host is performed over PCIe, a fast, standard interface used to connect and link with boards.

2.4.1.2. Partial Reconfiguration in SDAccel Hardware Platform

There are two methods for using PR in Hardware Platform designs: Regular Partial Reconfiguration (RPR) and Expanded Partial Reconfiguration (XPR). The Hardware Platform development process varies based on which of these methods is used, as it affects the logic hierarchy and design preparation techniques.

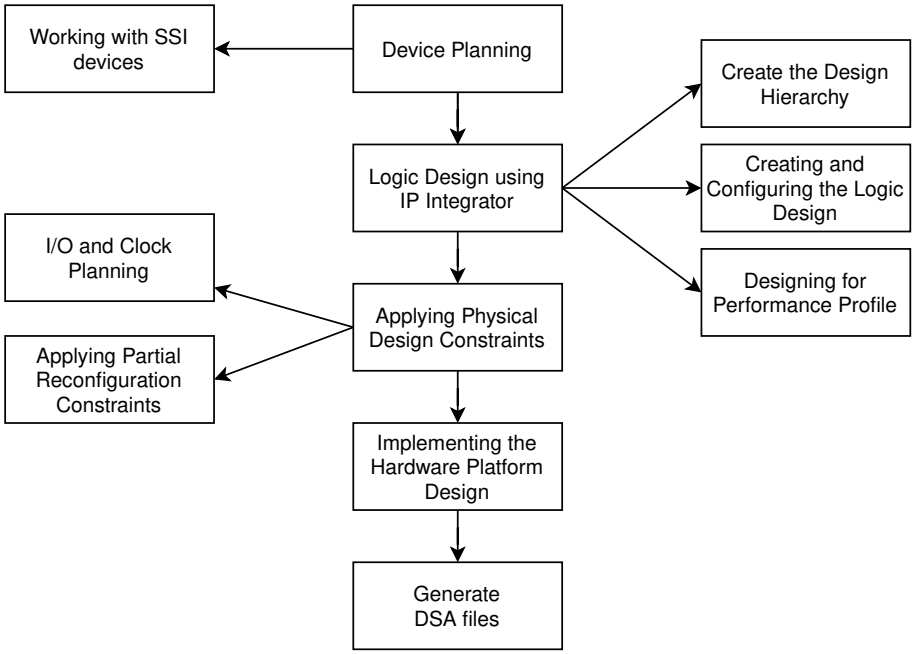


Figure 2.28: Hardware Platform creation and validation flow diagram

The XPR method includes board interface logic and DDR memory that are a part of the Hardware Platform design as part of the reconfigurable module. Implementing these along with the kernel from the SDAccel Environment enables the most effective use of the programmable device resources across the entire design. This is primarily done to improve the quality of the results, and maximize the performance of the overall design.

The smaller Static region contains the minimum logic needed to keep the Hardware Platform online and connected while waiting to be reconfigured with the hardware function of the Programmable region.

Typically with the XPR method, only the Xilinx DMA Subsystem for PCIe, basic control interfaces, and clock sources are contained within the Static region. This region is floorplanned to use as little of the device area as possible in order to maximize the available area for kernel resources.

This thesis work uses the XPR, so this chapter will focus on this technique.

2.4.1.3. Hardware Platform Design Flow

Figure 2.28 illustrates the Hardware Platform design and validation flow.

Device Planning

Working with SSI Devices If the target device is a SSI device, the designer must also analyze and plan the data flow across Super-Logic Region (SLR) boundaries. Although the increased availability of device resources in SSI devices is beneficial, the delay penalty for routes crossing between SLRs can be significant, and make timing closure especially difficult in highly connected designs.

AXI data paths used in many Platforms are wide and generally include combinatorial paths to implement the **AXI-4 Memory Mapped** protocol. It is imperative that these data paths are well designed and contain as few logic levels as possible when crossing SLR boundaries. I/O interfaces, IP logic distribution across SLRs, and clock isolation should also be studied carefully. As the Hardware Platform design is assembled and implemented, multiple iterations of the floorplan to achieve the required results may be needed.

I/O and Clock Planning The designer needs to pay special attention to the I/O interfaces and clocking topology used with SSI devices. Clocks within a single SLR need to be isolated whenever possible. The logic associated with various I/O interface IP should be floorplanned to reduce SLR crossings. The designer should also assign I/Os with the objective of optimizing data flow, and to account for the Programmable region logic coming from the SDAccel Development Environment.

Logic Partitioning across SLRs To avoid unnecessary SLR boundary crossing, floorplanning techniques are needed to assign specific IP cores to specific SLRs. The designer needs to plan the intended logic flow while paying attention to clock distribution, the expected logic size of the interfaces, and the utilization of hardened device resources such as PCIe, BRAMs and DSPs.

When planning logic content for the design, the total resources available in SSI devices should not be the only factor. Each SLR has a specific logic count, and over packing a single SLR can lead to poor performance for the whole design. Care has to be taken to balance the resource utilization across all SLRs as much as possible.

The logic for the Programmable region needs to be planned as well. Optimally, the Programmable region imported from SDAccel would be implemented within a single SLR. Consistent timing performance is more challenging when the Programmable region spans more than one SLR.

As previously commented, using an XPR Hardware Platform is preferred to maximize device resources and to allow the implementation tools flexibility when placing timing sensitive logic across the SLR boundaries.

Adding Pipeline Registers across SLRs If a critical interface must cross an SLR boundary, the designer should consider adding pipeline registers to the interconnect to help prevent performance degradation. These registers can be floorplanned to assign them to a specific SLR resource, or allowed to float to let the Vivado tool place them.

Figure 2.29 shows a simplified representation of the logic partitioning described above. The red horizontal line represents the boundary between the two SLRs. The small white squares represent pipeline stages that are floorplanned for assignment to a specific SLR. The small gray squares in the figure represent pipeline stages connecting to the Programmable region that are not floorplanned, and may be placed on either side of the SLR boundary as determined by the Vivado placer.

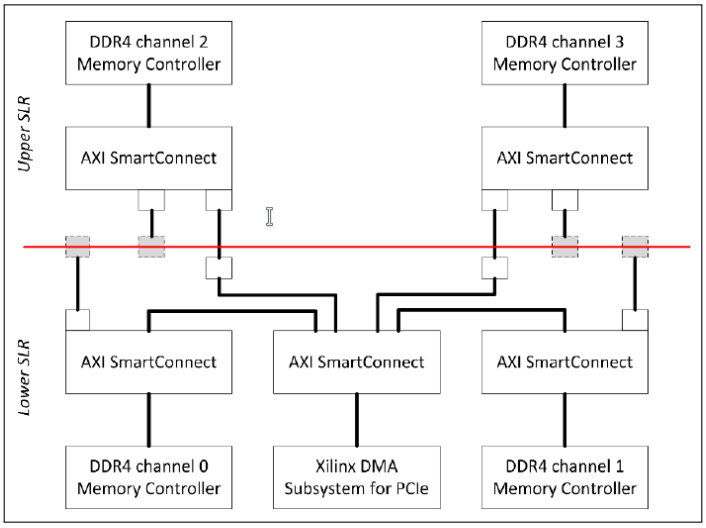


Figure 2.29: Simplified representation of IP partitioning for controlled SLR crossing

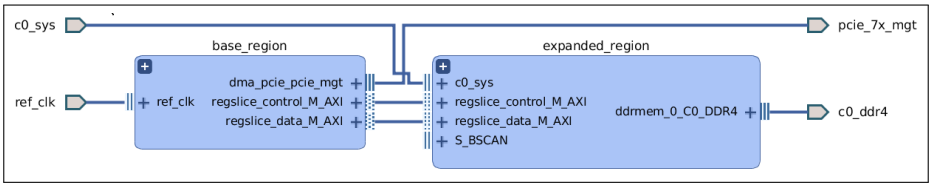


Figure 2.30: Top-Level Logic Hierarchy for Expanded Region

The Static region logic interconnect has to be defined with careful floorplanning of IP logic and signal interfaces and creating any necessary AXI Register SLR crossings in order to effectively utilize both SLRs.

Logic Design using IP Integrator Vivado IP Integrator feature will use to create the hardware platform logic design. This feature of the Vivado Design Suite provides the necessary board information, interface IP, and the interface to the programmable region which will be imported from the SDAccel Development Environment.

Creating the Design Hierarchy The logic hierarchy of the Hardware Platform must be carefully planned to properly support the partial reconfiguration flow (RPR or XPR) for the Programmable region of the design, to define the static base region, and to plan the signal flow across SLRs of SSI devices, and to other elements of the Hardware Platform, such as components on the Platform board or IP cores in the subsystem design. Proper levels of hierarchy need to be established within the Vivado IP Integrator block design.

When using XPR, the expanded region and the Static region logic must be separately partitioned in the hierarchy of the Hardware Platform. Figure 2.30 shows the top-level view of the IP Integrator block diagram of an example Hardware Platform showing the static base region and the reconfigurable expanded region.

With partial reconfiguration, all routing must be contained within the Programmable region. If the floorplan for the Static region includes any I/O ports that are used by signals within the expanded region, an Input Buffer (IBUF) or

Output Buffer (OBUF) needs to be placed in the Static region logic design to connect it through the Static region to the expanded region.

Creating and Configuring the Logic Design Use the features of the Vivado IP Integrator to instantiate IP from the Vivado IP catalog, configure and connect all of the interface IP associated with the Hardware Platform.

Configuring the Programmable region The Programmable region is represented by the SDAccel OpenCL Programmable region IP core in the block design. The designer should configure this module to properly represent the interfaces between the Programmable region and the static logic design. The OpenCL compiler uses the IP integrator framework to swap out the default Programmable region logic (for the training or placeholder kernel) and replace it with the compiled kernel generated from the OpenCL device code, and the interconnect topology it requires.

The SDAccel OpenCL Programmable region IP is a container for compiled OpenCL kernels from the SDAccel Development Environment. The number of training kernels the core contains can be specified. When used with SDAccel, the OpenCL code and the Xilinx OpenCL Compiler (xocc) determines the number of kernels in the Programmable region. The IP has one AXI slave interface which is used to send commands to the OpenCL kernels. It also has one or more AXI master interfaces which are connected to device RAM. A clock and a reset signal are also wired to the SDAccel OpenCL Programmable region.

The designer can customize the address and data bit width for the AXI master interface, but the maximum value that the memory controller can support in order to maximize bandwidth and minimize data width converters on the datapath should be used.

The SDAccel OpenCL Programmable region can support up to 16 compiled kernels on an FPGA device, located between the slave and master AXI interconnects. These are essentially placeholders to later accept compiled SDAccel kernels.

The designer must set the Master ID width on the Re-customize IP dialog box of the SDAccel OpenCL Programmable region in order to support multiple kernels. The Master ID width field supports values from 1 to 16.

If direct connections are required from the Programmable region to external FPGA I/Os, either an IBUF or OBUF must be instantiated into the static logic region.

Working with AXI Interconnect The Programmable region has two AXI Interconnects. One AXI interconnect is used to connect the SDAccel OpenCL Programmable region slave control port and to provide a path to read/write DDR from the PCIe bus. The other one is used to connect two bus masters to the Memory Controller: the PCIe DMA controller and SDAccel OpenCL Programmable region IP. The two AXI interconnections provide a path that allows bypassing the programmable region and directly access the map's resident RAM.

Configuring the AXI Address Space The device relies on AXI memory mapped bus for addressing the OpenCL kernel, the device dedicated RAM, and any other peripherals. There are two AXI masters in the design: the PCIe Endpoint/DMA Controller and the OpenCL kernel. There are two AXI slave end points in the design: The AXI-Lite control port of the Programmable region (*ocl_block_0*) and the Memory Interface Generator (MIG) controller.

Configuring the Design for SDAccel The Platform design should contain a DMA Subsystem master IP, memory controller interfaces (DDR3, DDR4 IP) for global memory, clocking via the Clocking Wizard IP and resets via the Processor System Reset IP.

Use the Xilinx DMA Subsystem for PCIe (XDMA) IP, or use a board vendor specific IP for the DMA functionality. This IP provides PCIe DMA functionality between the card resident RAM and host RAM.

The XDMA IP requires the addition of two master AXI interfaces:

- One to master data to the Memory Controller.
- One to control the Programmable region (to configure the OCL Region kernels) and to control other peripherals in the Static region.

Using the Decoupler IP The PR Decoupler IP may also be used in the Static region to hold the design in a safe state while the device is partially reconfigured with the hardware function. A single reset within the IP will put the entire Programmable region into reset. A pair of registers supply back pressure on the AXI interfaces to ensure they maintain state. The Decoupler IP can be instantiated and configured into the Static region design using the IP Integrator.

Configuring Debug Logic Debug logic can be added to the design to enable hardware validation and debug using the Vivado Hardware Manager. The System Integrated Logic Analyzer (System ILA) IP should be configured to add debug logic to the design. It can be used to monitor and validate the running Hardware Platform on the Xilinx device.

Designing for Performance Profiling The SDAccel OpenCL runtime has integrated device profiling features to enable real-time performance analysis of the board as they are being utilized. Performance statistics from the board can be unified with statistics from the host to get a clearer picture of the performance of the entire system. This allows the designer to analyze the performance of the device and then quickly optimize the host code and kernel source.

To take advantage of this capability, the Hardware Platform must have the following:

- A performance monitor framework must be added to the Hardware Platform.

- The framework must follow a specified address mapping.
- The specified format for the data must be followed.
- A shim for the Xilinx OpenCL Hardware Abstraction Layer (XCL HAL) driver API must be implemented.

A performance monitor framework can be inserted into the Hardware Platform. Xilinx provides a standard IP: AXI Performance Monitor (APM). An APM is the IP core used to monitor the AXI transactions and provide performance statistics and events of every monitored AXI connection.

The APM contains two modules: profile metric counters, and trace. The profile metric counters are aggregated counters used to calculate average throughput and latency for both write and read channels. A single AXI-Lite interface is used to configure the APM and read the metric counters. Xilinx recommends that using the sampled metric counters, which are guaranteed to provide samples on the same clock cycle. This is done by first reading from the sample interval register, then reading the appropriate sampled metric counters.

The APM trace module provides timestamps of AXI transaction events. These events are captured in a collection of individual flags and are output by the APM using an AXI Stream in one of the following ways:

- **AXI Lite**: slower offload but easier to close timing
- **Full AXI**: faster offload as it offloads trace at kernel clock rate

Thesis thesis work uses the Full AXI way. In this case, which uses AXI Full and a single FIFO, APM cores are not contained within their own level of hierarchy. However, the following IP cores logically constitute an *apm_sys*, which monitors transactions for the purposes of recording and reporting system performance details: *xilmonitor_apm*, *xilmonitor_broadcast*, *xilmonitor_fifo0*, *xilmonitor_subset0*.

Applying Physical Design Constraints

I/O and Clock Planning One of the key considerations in the design of a Device Support Archive (DSA) is to identify the I/Os necessary for the board requirements. The Static I/Os are expected to operate and remain live during the device reprogramming of OpenCL kernels. The physical I/O locations will influence performance and must be considered as part of the floorplanning process, especially when SSI devices are used.

It is recommended that the static I/Os are assigned physically outside of the Programmable region. There may be trade-offs made between the size and shape of the Programmable region to accommodate an area free of static I/Os and overall system performance.

If the floorplan for the Static region includes any I/O ports that are used by signals within the Programmable region, an IBUF or OBUF needs to be placed in the Static region logic design to connect it through the Static region to the Programmable region.

Applying Partial Reconfiguration Constraints The use of Xilinx PR technology requires the use of several types of physical constraints. These include:

- Definition of the reconfigurable region (Partition Definition)
- Floorplanning with Pblock
- Partition Pin Placement (partPin)

Definition of the Reconfigurable Region The Programmable region of the Hardware Platform must be identified in order for the tools to process the logic correctly with PR and in order to create a partial bitstream file. This process is done differently depending on whether the RPR flow or the XPR flow to define the Hardware Platform are using.

For the XPR flow, the designer will directly assign the *HD.RECONFIGURABLE* property on the hierarchical module that defines the expanded region. The *HD.RECONFIGURABL* property is set to *TRUE* in the Xilinx Design Constraints (XDC) constraints file on the expanded region hierarchical block.

Notice the *DONT_TOUCH* property is also set on the expanded region hierarchical module to prevent optimization across the boundary of the module.

Floorplanning When using Xilinx PR technology, floorplanning is required to separate the Static and Programmable regions of the design. The floorplanning strategy can dramatically affect the performance of the design. It can be an iterative process to find the optimal floorplan for the specific design and device.

Floorplanning is performed by assigning logic modules or cells to Pblock ranges on the device canvas. Floorplanning can be performed interactively by opening the Synthesized design in the Vivado Integrated Design Environment (IDE) or with constraints in the XDC file. Occasionally, in order to optimize performance or device resources, multiple Pblocks are used.

Pblocks can be defined as rectangular regions, or by combining multiple rectangles to define a non-rectangular shape. Pblocks can also be nested to enable lower levels of logic to be further constrained to specific regions of the device. If nested Pblocks are desired, ensure the lower level Pblocks region is completely within the upper level Pblocks region.

When designing an XPR Hardware Platform, only a small Static region is needed for the PCIe/DMA base logic. The rest of the Hardware Platform logic, including the Programmable region, gets implemented as part of a single XPR region hierarchy. Multiple Pblock rectangles should be used for the expanded region logic to reserve and use as much of the device as possible. Ensure as many of the DSP and BRAM device resources are included in the expanded region Pblock area. The designer will adjust the Pblock shapes to make the Static region as small as possible near the I/O banks assigned to it. Multiple iterations may be required to see how tightly the Static region can be packed.

Partition Pin Assignment Partition pins need to be assigned for the Programmable region interface pins to act as anchor points for the interface signal routing. Two ways can be followed to do this:

- manually define placement ranges for partition pin assignments setting the property *HD.PARTPIN_RANGE* for a set of slice.
- automatically assigned by the implementation tools within the Programmable region, along the border of the Programmable and Static regions.

Implementing Hardware Platform design The design should be synthesized and implemented to ensure desired performance is achieved. It is often required to iterate on floorplanning and implementation strategies to ensure optimal performance.

It is often important to implement, analyze, and iterate on the Hardware Platform design to ensure that it continues to meet timing during kernel implementation. Using a test kernel, implement the design and then check that the design meets timing by opening the Implemented Design.

Generating DSA files After completing Hardware Platform design using the Vivado Design Suite, the designer is ready to create a DSA file for use with the SDAccel Development Environment.

The DSA contains all of the data required to enable the design and programmability of the OpenCL C or C++ kernel directly with the running FPGA.

The DSA is output from the Hardware Platform design project in the Vivado Design Suite for use with the SDAccel Development Environment.

Setting Vivado Properties for Generating a DSA Prior to creating a DSA file for the hardware platform, the designer must define various properties in the hardware platform design so that they are included in the DSA. These include metadata properties to help identify the DSA, as well as design properties to define system configuration.

The following three properties are required by the *write_dsa* Tcl command, and must be defined prior to generating a DSA. If these properties are not found, the *write_dsa* command will issue an error and stop. The values shown are example values.

```
set_property dsa.vendor "vendor_name" [current_project]
set_property dsa.name "dsa_name" [current_project]
set_property dsa.boardId "board_id" [current_project]
```

The following property can be used to override the Serial Peripheral Interface (SPI) Flash type associated with a DSA, and influences the generation of the MCS file from the bitstream.

```
set_property dsa.flash_interface_type "spix8" [
   current_project]
```

The following properties have default values but can be defined to specify the DSA file version, enable the PR flow, or capture the synthesis checkpoint within the DSA.

```
set_property dsa.version "1.0" [current_project]
set_property dsa.uses_pr true [current_project]
set_property dsa.static_synth_checkpoint false [
   current_project]
```

The following properties can be used to set the PCIe ID and Board attributes for the Board section if a board file is not yet available.

```
set_property dsa.pcie_id_vendor "vendor_id" [
   current_project]
set_property dsa.pcie_id_device "device_id" [
   current_project]
set_property dsa.pcie_id_subsystem "subsystem_id" [
   current_project]
set_property dsa.board_name "board_name" [
   current_project]
set_property dsa.board_interface_type "gen3x8" [
   current_project]
set_property dsa.board_memories {{ddr3 8GB} {ddr4 16GB}
    ...} [current_project]
set_property dsa.board_interface_name "PCIe" [
   current_project]
set_property dsa.board_vendor "board_vendor_name" [
   current_project]
```

If defined, these property values will take precedence over the values that come from the board files. The default value for these properties is an empty string. Using a local board repository can be avoided by using these properties.

When using an expanded partial reconfiguration hardware platform, the following parameter is also required:

```
set_param dsa.expandedPRRegion 1
```

Following property enables debug IP capabilities with PR in coordination with Vivado.

```
set_param chipscope.enablePRFlow true
```

Generating the DSA Once the design has been implemented, and the required properties have been set, DSA file can be generated using the *write_dsa* command.

The designer can use the *validate_dsa* command to validate a custom DSA file to ensure it contains the proper content and metadata needed to support the Hardware Platform in the SDAccel environment.

Notice that the *write_dsa* command should be called after implementation phase and before the bitstream generation phase.

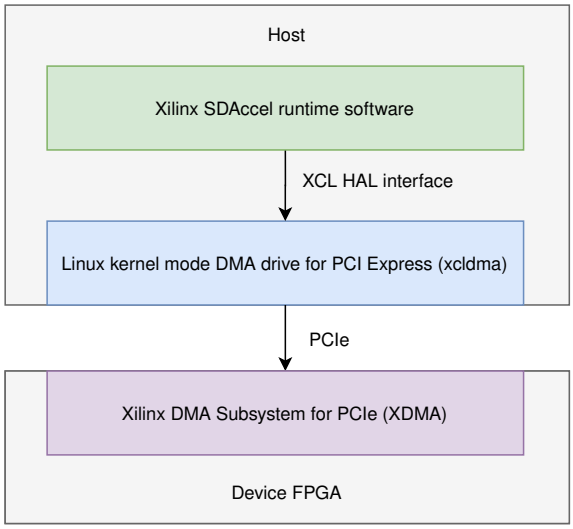


Figure 2.31: Layers of the Software Platform

2.4.2. Software Platform

Software Platform consists of the runtime, drivers, and software stack that are needed to enable interaction with the Hardware Platform.

The SDAccel runtime software is layered on top of a common low-level software interface called the Hardware Abstraction Layer (HAL). The HAL driver provides APIs to runtime software which abstracts the *xcldma* driver details. The *xcldma* driver is a kernel mode DMA driver which interfaces to the memory-mapped Platform over PCIe. Figure 2.31 shows the layers of the Software Platform.

2.4.2.1. xcldma

The *xcldma* kernel mode driver is used to manage the device (the accelerator), and provides essential services such as:

- DMA transfers from host to device and device to host.
- Clients can memory map the device registers which are exposed on PCIe Base Address Register (BAR).
- Bitstream download capabilities.
- Kernel clock scaling.
- Access to sensors on the device via System Monitor (SysMon) (temperature, voltage, etc.).
- MIG calibration status reporting.
- Resetting and rebooting the board from Programmable Read-Only Memory (PROM).

When the Hardware Platform is implemented as a DSA and used with the SDAccel Development Environment, the HAL driver insulates the SDAccel runtime software from the implementation details of the *xcldma* driver.

Since the Platform developed in this thesis work is completely customized and derived from known reference design, the Device ID and Subsystem ID of the PCIe IP have also been changed. Therefore, the drivers have been changed in order to make them compatible with new Device ID and Subsystem ID. The meaning of each bit of the Device ID and Subsystem ID is described in the Section 2.4.2.3.

2.4.2.2. XCL HAL

XCL HAL Driver API is required by the OpenCL runtime to communicate with the Hardware Platform. It is used for downloading Xilinx FPGA bitstreams, allocating and de-allocating OpenCL buffers, migrating OpenCL buffers between host memory and Hardware Platform memory, and communicating with the OpenCL kernel on its control port.

The API supports address spaces which may be used for accessing device peripherals with their own specific memory mapped ranges.

2.4.2.3. Meaning of bits of Device ID and Subsystem ID

The Vendor ID identifies the vendor of the device. The Device ID identifies a specific device from that manufacturer/vendor. The Subsystem ID is assigned by the subsystem vendor from the same number space as the Device ID. The Subsystem Vendor ID–Subsystem ID combination identifies the card, which is the kind of information the driver may use to apply a minor card-specific change in its operation.

The Vendor ID is a fixed number for Xilinx devices and it is 0x10EE.

The Device ID and Subsystem ID are two hexadecimal numbers that can be set and changed in the XDMA IP. These are massively used by drivers to extract Platform information, such as memory size. Each digit therefore has a different meaning.

The meaning of each bit of the Device ID and Subsystem ID depends on the version of the DSA and therefore the drivers that are used. In this thesis work, the DSA version is 3.3 and the corresponding drivers are included in version 2017.1 of Xilinx SDAccel.

To know the meaning, therefore, all the components of the drivers have been studied by doing a static analysis of the code. As an example, Device ID and Subsystem ID 0x8238 and 0x4432 respectively were used for the analysis. Table 2.1 and 2.2 reports the result of this analysis.

Other meanings are given by the set of bits:

- if Ultrascale or Ultrascale+ is used and DSA is greater than 32 then SysMon is supported by the driver and will be enabled all related features.

Device ID	Meaning
8	if FPGA architecture is Ultrascale based (8) or Ultrascale+ based (9)
2	
3	
8	family serie of FPGA

Table 2.1: Meaning of Device ID bits

Subsystem ID	Meaning
4	if XPR (4) or RPR (other number) is used.
4	number of DDR
3	DSA version
2	

Table 2.2: Meaning of Subsystem ID bits

- if Device ID is 0x8238 or first two digit of Subsystem ID are 44 and DSA version is greater than 31 then driver will support multiple clock and reclocking feature.

2.4.3. Generating Platform files

Once the DSA file has been created for the Hardware Platform, it needs to be associated with the Software Platform configuration files and drivers. Xilinx recommends using the same directory structure as used with the supplied example Hardware Platforms.

2.4.3.1. Platform Directory Structure

Each Platform should be contained in a directory with a name that corresponds to the DSA name. Naming the folder after the DSA file is just a recommendation, but it does help identify the Hardware Platforms from the directory names.

The Platform folder is structured as shown in figure 2.32.

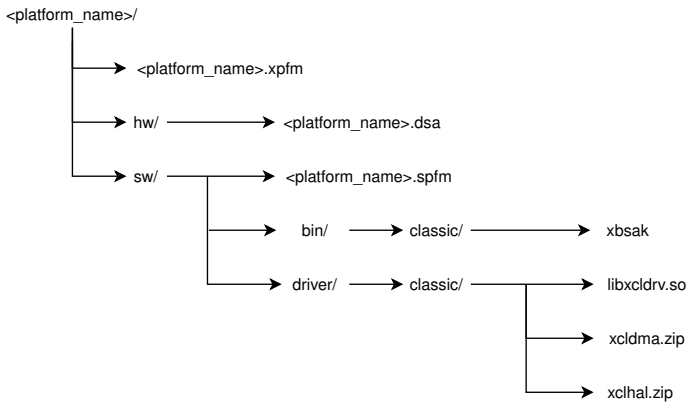


Figure 2.32: Platform folder structure

2.4.3.2. Configuring the Hardware Platform .xpfm File

The *<platform_name>.xpfm* file defines the location of the hardware and software components of the Hardware Platform.

2.4.3.3. Populating the Hardware Platform Directory

The Hardware Platform is defined in the DSA file. It is needed to move the DSA into ./hw folder. The DSA file contains the following information:

- **dsa.xml**: contains DSA Hardware Platform settings and configuration from when the write_dsa Tcl command was run, and the DSA file was created.
- **<platform_name>.bit**: full bitstream file for initial system configuration.
- **<platform_name>_clear.bit**: for UltraScale architecture only, this is a file for clearing the full bitstream from the device.
- **<platform_name_pblock_name>.bit**: placeholder partial bitstream of the test kernel for the Programmable region logic. The partial bitstream for the Programmable region is dynamically created by the SDAccel Development Environment and will replace the one included in the DSA file.
- **<platform_name>.dcp**: routed version of the Hardware Platform design.
- **<platform_name>.png**: optional picture of the board to display in Vivado and SDx projects.

2.4.3.4. Populating the Software Platform Directory

The Software Platform is included in the ./sw folder of the Platform directory structure. The ./sw folder contains the *<platform_name>.spfm* file and a ./driver subdirectory, as shown in 2.32.

The *<platform_name>.spfm* file defines information for the Software Platform.

The ./driver subdirectory contains the driver configurations defined for the Software Platform. The *libxcldrv.so* is a compiled OpenCL library file. To use it, in Linux environment, the location of this file should be extracted in *LD_LIBRARY_PATH* in environment variable. This file depends on the hardware and for a custom platform is needed generate it. How to generate it will explain in section 3.2.2.2.

The ./bin folder contains the Xilinx Board Swiss Army Knife (xbsak) utility that fits the platform.

Once the xbsak utility and the *libxcldrv.so* was generated, the designer must move them to the /sw/bin/classic and /sw/driver/classic folder respectively.

2.5 Tools

2.5.1. Xilinx Vivado Design Suite

Vivado Design Suite is a software suite produced by Xilinx for synthesis and analysis of HDL designs with additional features for System-on-a-Chip (SoC) development and HLS.

Vivado Design Suite is an IDE and it provides an intuitive Graphical User Interface (GUI). All of the tools and tool options are written in native Tool Command Language (Tcl) format, which enables use both in the Vivado IDE or Vivado Design Suite Tcl shell. Analysis and constraint assignment is enabled throughout the entire design process. For example, user can run timing or power estimations after synthesis, placement, or routing. The features offered by Vivado Design Suite are:

- RTL design in VHDL, Verilog, and SystemVerilog
- IP integration for cores
- Behavioral, functional, and timing simulation with Vivado simulator
- Vivado synthesis
- Vivado implementation for place and route
- Vivado serial I/O and logic analyzer for debugging
- Vivado power analysis
- Synopsys Design Constraints (SDC)-based XDC for timing constraints entry
- Static timing analysis
- High-level floorplanning
- Detailed placement and routing modification
- Bitstream generation

Vivado Design Suite uses a concept of opening designs in memory. Opening a design loads the design netlist at that particular stage of the design flow, assigns the constraints to the design, and then applies the design to the target device. This provides the ability to visualize and interact with the design at each design stage.

2.5.1.1. Use within our Project

Vivado Design Suite was used first to create and modify the circuit design, modify the physical constraints of the board and then synthesize, implement the new circuit design and generate the DSA. This tool was used through the GUI.

2.5.2. Xilinx SDAccel Development Environment

The **Xilinx SDAccel Development Environment** is part of the **SDx Development Toolchain**. This toolchain allows user to create FPGA accelerated designs using C/C++, OpenCL C, or RTL programming languages. User can create these designs in the SDx GUI environment or through a Makefile flow.

As said, a kernel can be implemented in RTL and developed using the Vivado IDE tool suite. RTL kernels offer potentially higher performance with lower area and power, but require development using RTL coding, tools, and verification methodologies. Existing RTL based IP and algorithms can be wrapped and migrated to the SDAccel framework enabling those HDL based algorithms to be callable by the runtime and application program. RTL kernels must use the correct interfaces, protocols, and packaging to be recognized by the SDAccel tool flow and runtime library. Therefore, implementing and developing a kernel in RTL requires high expertise in digital design and HDL. To increase the flexibility of using the custom platform, we can rely on use high-level programming languages, such as C/C++ and OpenCL. In this way, any developer with high-level programming know-how, regardless of his or her digital design knowledge, can accelerate any kernel using the custom platform presented in this thesis. The reason of this is that the compilation of OpenCL applications into binaries for execution on an FPGA does not assume nor require FPGA design knowledge. A basic understanding, however, of the capabilities of an FPGA is necessary during application optimization in order to maximize performance. The SDAccel environment handles the low-level details of program compilation and optimization during the generation of application specific CUs for an FPGA fabric. Therefore, using the SDAccel environment to compile an OpenCL program does not place any additional requirements on the user beyond what is expected for compilation towards a CPU or GPU target.

2.5.2.1. SDAccel Compilation flow

In contrast to a CPU or a GPU, an FPGA can be thought of as a blank computing canvas onto which a compiler generates an optimized computing architecture for each kernel in the system.

This inherent flexibility of the FPGA allows the developer to explore different kernel optimizations and CU combinations that are beyond what is possible with a fixed architecture. The only drawback to this flexibility is that the generation of a kernel specific optimized compute architecture takes a longer time than what is acceptable for just-in-time compilation. The OpenCL standard addresses this fundamental difference between devices by allowing for an offline compilation flow. This allows the user to generate libraries of kernels that can be loaded and executed by the host program.

SDAccel Development Environment uses an offline compilation flow to generate kernel binaries. To maximize efficiency in the host program and allow the simultaneous instantiation of kernels that cooperate in the computation of a portion of an application, Xilinx has defined the Xilinx OpenCL Compute Unit Bi-

nary (xclbin) format. The xclbin file is a binary library of kernel CUs that will be loaded together into an OpenCL Context for a specific device.

The xclbin file is created using the SDx IDE or the xocc command line utility. It provides a mechanism for command line users to compile their kernels, which is ideal for compiling host applications and kernels using a Makefile.

Since a Makefile flow was used is this thesis work, we explain the xocc command line utility. Most important options of xocc are:

- **– platform <arg>**: to select an acceleration device suppported by Xilinx and third-party platform providers.
- **–target sw_emu | hw_emu | hw**: to select the target of compilation.

Software emulation (sw_emu) or CPU emulation target The main goal of CPU emulation is to ensure functional correctness and to partition the application into kernels. Although partitioning and optimizing an application into kernels is integral to OpenCL development, performance is not the main goal at this stage of application development in the SDAccel environment.

For CPU-based emulation, both the host code and the kernel code are compiled to run on an x86 processor.

Hardware emulation (hw_emu) target The SDAccel development environment generates at least one custom CU for each kernel in an application. This means that while the CPU emulation flow is a good measure of functional correctness, it does not guarantee correctness on the FPGA execution target.

The SDAccel environment has a hardware emulation flow, which enables the programmer to check the correctness of the logic generated for the custom CUs. This emulation flow invokes the hardware simulator in the SDAccel environment to test the functionality of the logic that will be executed on the FPGA compute fabric.

Hardware (hw) or system target The SDAccel development environment generates custom logic for every CU in the binary container. Therefore, it is normal for this build step to run for a longer period of time than the other steps in the SDAccel application compilation flow. The steps in compiling CUs targeting the FPGA fabric are as follows:

1. Generate a custom CU for a specific kernel.
2. Instantiate the CUs in the OpenCL binary container.
3. Connect the CUs to memory and infrastructure elements of the target device.
4. Generate the FPGA programming file.

The generation of custom CUs for any given kernel code uses the production proven capabilities of the Xilinx Vivado HLS tool, which is the CU generator in the SDAccel environment. Based on the characteristics of the target device in the solution, the SDAccel environment invokes the CU compiler to generate custom logic that maximizes performance while at the same time minimizing compute resource consumption on the FPGA fabric. Automatic optimization of a CU for maximum performance is not possible for all coding styles without additional user input to the compiler.

After all CUs have been generated, these units are connected to the infrastructure elements provided by the target device in the solution. The infrastructure elements in a device are all of the memory, control, and I/O data planes which the device developer has defined to support an OpenCL application. The SDAccel environment combines the custom CUs and the base device infrastructure to generate an FPGA binary which is used to program the Xilinx device during application execution.

2.5.2.2. Use within our Project

The Xilinx SDAccel Development Environment tool was used for three purposes:

- to generate all necessary files for the platform support package for the FPGA card. In particular, the FPGA acceleration card plugged into the host machine needs to have the associated Linux kernel driver, firmware and runtime libraries installed before it can be used for running user applications. SDAccel provides Xilinx Board INSTallation (xbinst) tool. It also generates an installation script to compile and install the driver, firmware and runtime libraries. The commands launched will be explained in Section 3.2.4.

- to perform the following board administration and debug tasks independent of SDAccel runtime library:

  - Board administration tasks:
    - Flash PROM.
    - Reboot boards without rebooting the host.
    - Reset hung boards.
    - Query board status, sensors and PCIe Advanced Error Reporting (AER) registers.
  - Debug operations:
    - Download SDAccel binary (.xclbin) to FPGA.
    - DMA test for PCIe bandwidth.
    - Show status of CUs.

  The utility is named xbsak. It was used mainly to test the PCIe bandwidth, to write and read a string of bit to/from DDR and to query the custom platform and obtain information about it. The commands used will be explained in Section 3.3.1.

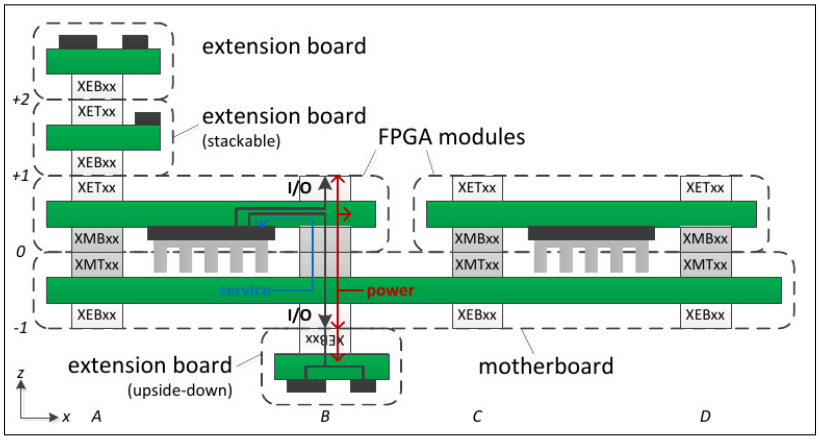


Figure 2.33: The logical view of the proFPGA prototyping system hardware-side

xbsak utility is strongly linked to the hardware and for a custom platform is needed generate it. In the 3.2.2.2 section will be explained how to generate the xbsak utility that fits the custom hardware.

- to compile a set of kernels for the custom platform developed in the thesis work and to optimize these kernels through directives to the pre-compiler, also called **pragma**, offered by Xilinx, in order to improve the performance of the kernel, the data throughput, reduce the latency, or reduce the resources utilized by the kernel.

2.5.3. proFPGA prototyping system

The **proFPGA prototyping system** consists of a set of modular building blocks. This allows highly customized prototyping solutions to match the project-specific resource requirements with a minimum of system complexity.

In order to clearly describe the tools under examination, we will explain the hardware and the software part separately.

2.5.3.1. Hardware side

The proFPGA prototyping system consists of a set of modular hardware units. In figure 2.33 we can see a logical view of system.

As we can see, several component are contained:

- **Motherboards**: provide the proFPGA system infrastructure. They offer mechanical fixture, power supply, I2C-based system management, clocking infrastructure, and MMI-64 communication for multiple FPGA modules. Also, motherboards have FPGA module connectors (carrying user I/O, power supply, service) on the top side and Extension board connectors (carrying user I/O, power supply) on the bottom side. The user I/O pins of top-side and bottom-side connectors are directly connected with each other, providing a transparent connection from the FPGA module on the top-side to the extension board on the bottom side.

- **FPGA Modules**: contain the user design. They offer up to 8 connectors to extension sites (4x top, 4x bottom). Each FPGA module has access to MMI-64 communication from the motherboard. FPGA modules have 4 FPGA module connectors (user I/O, power supply, service) on the bottom side and up to 4 Extension board connectors (user I/O, power supply) on the top-side. Because the motherboard transparently converts the FPGA module connector into an Extension board connector, each FPGA module can access up to 8 extension boards.
- **Extension Boards**: provide hardware functions to user designs inside the FPGA modules, e.g. Synchronous Dynamic Random Access Memory (SDRAM) memory, user PCIe connection, debug access. One extension board occupies one or more extension board connectors of one FPGA module, giving the user design inside the FPGA module exclusive access to the extension board. The Extension board connectors of the FPGA module are located on the bottom side. Some extension boards (e.g. the user PCIe adapter) are stackable. Unused I/O pins from the FPGA module are mapped to a top-side connector, allowing further extension boards to be added.
- **Interconnects**: are special extension boards to connect I/O pins of different FPGA modules. They are available as boards and cables. Interconnect cables connect two extension sites. Interconnect boards connect two or more extension sites. Connections are either broadcast (e.g. the 4-way interconnect board) or point-to-point (e.g. all two-way interconnect boards and cables).
- **System Extension Boards**: the system functionality may be extended by special hardware, such as motherboard PCIe adapter board or motherboard-to-motherboard connector cable. This hardware uses dedicated connectors on the motherboard.

The modules of the proFPGA prototyping system are assembled into a 3-dimensional structure. Any location of the extension boards can be identified by the coordinate systems.

2.5.3.2. Software side

The proFPGA system is delivered with the following applications:

- **profpga_run**: command line tool to power-up, power-down and to configure the system.
- **profpga_builder**: GUI to create board configuration setups and to perform runtime accesses to the system.
- **profpga_brdgen**: command line tool to generate various VHDL/Verilog toplevel files, constraint files, board description files and a design based self-test to test FPGA interconnections at speed.
- **profpga_selftest2**: command line tool which executes the design based self-test on the hardware.

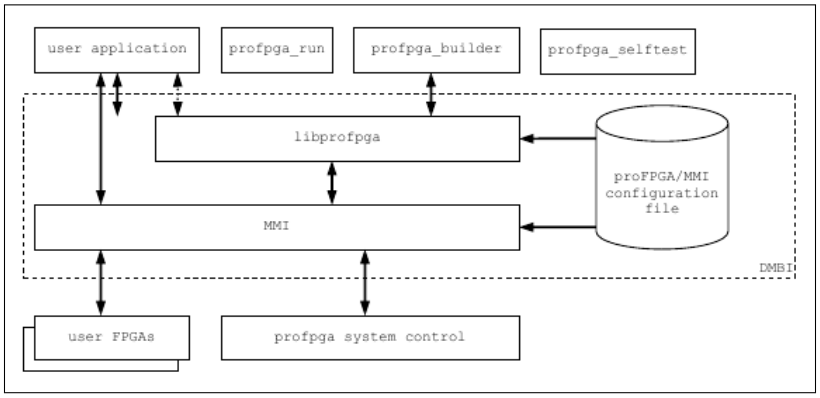


Figure 2.34: The software architecture of the proFPGA prototyping system

- **profpga_freq**: command line tool to determine proFPGA clock settings to generate a specific clock frequency or to achieve a desired data transfer rate with the proFPGA mux/demux modules.

The communication with the proFPGA system is done through the *libprofpga* library where all application above based on.

There are many options how to access the system (e.g. Universal Serial Bus (USB), Ethernet, PCIe) and there are many parameters and configuration options (FPGA binary images, clock and reset settings, etc.). All these options and settings are combined in a configuration file which is used by the applications and libraries.

There are two parts for the communication with the system:

- The transport mechanism between a Host-PC and the proFPGA system, which is called MMI. This mechanism abstracts the different communication ways like PCIe, Ethernet and USB and is used for user design communication and to control the system.
- The *libprofpga* provides all functionality to configure and control the system and to observe status information. *libprofpga* communicates with the hardware via MMI.

Both parts together form the proFPGA communication package which is called Device Message Box Interface (DMBI). In the figure 2.34, is shown the software architecture provided by proFPGA.

2.5.3.3. Use within our Project

The software presented was used to scan the infrastructure used, program the FPGA and generate the physical constraints. The commands used will be explained in the section 3.2.5.

CHAPTER 3
Contribution

3.1 Specification

In Section 1.4, we described the goal of this thesis work. As said, the goal is to realize an acceleration Platform for a third-party FPGA. Since Xilinx offers the source code of different reference design of own Platform with a portion of the software stack, which provides device drivers and a HAL compatible with them, we can follow two approaches: create a design from a scratch and use a reference design as a baseline to extend or adapt to other target requirements. The way chosen is the second one. The reference design is available here [25].

The specification of our Platform is different from the reference design.

3.1.1. Reference design specification

Below are the specifications of the reference design:

- 4 GB DDR4.
- 4 kernels.
- Flash memory.
- System Management.
- Standard Device ID and Subsystem ID.
- DDR clock frequency at 250 MHz.
- XPR.

3.1.2. Our design specification

Below are the specifications of our design:

- 2 GB DDR4.

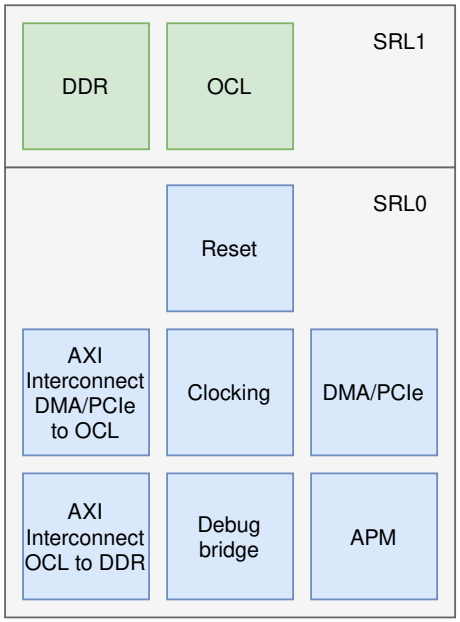


Figure 3.1: Logical view of position of components in the FPGA

- 16 kernels.
- No Flash memory.
- No System Management.
- Custom Device ID and Subsystem ID.
- DDR clock frequency at 125 MHz.
- XPR.

3.2 Design

To create the custom Platform, the flow shown in figure 2.25 was followed.

3.2.1. Hardware Platform

In order to create the Hardware Platform, the flow described in section 2.4.1.3 was followed.

3.2.1.1. Device Planning

Working with SSI Devices As already said, the Kintex UltraScale 115 used in this thesis use the SSI technology. The number of SLRs of this device is 2, as has been extracted from [22]. In Figure 3.1, we can see how the various components are positioned in the FPGA, emphasizing which SLR the different components belong to. The SLRs are called **SLR0** and **SLR1**.

Contrary to what the documentation recommends, that the Programmable region must be isolated in a single SLR, from Figure 3.1 we can see that the DDR and SDAccel OpenCL Programmable region IP have been located in the *SRL1* SLR, while other components that logically belong to the Reprogrammable region, such as AXI OCL Region to DDR, APM, Debug bridge, have been located in the SRL0 SLR. The reason for this is that each SLR has a limited number of resources and therefore we want to offer as many resources as possible to the kernel implementation. The DDR is leased to SRL1 SLR because of the physical constraints of the proFPGA components.

I/O and Clock Planning As we can see from the previous pictures, the I/O and clock are planned in order to reduce as much as possible the SLR crossing. Specifically, the circuitry used to provide the clock source is located and isolated in SRL0. PCIe and DDR are the only I/O way of design. As we can see, both are isolated and located, respectively, in SRL0 and SRL1.

Logic Partitioning across SLRs To avoid unnecessary SLR crossing, specific IP cores is assigned to specific SLRs by floorplanning technique that will be explain later. Since XPR was used, most assignments is performed by Vivado Design Suite.

Adding Pipeline Registers across SLRs In our design, the DDR and the PCIe are located in different SLR, so a SLRs crossing exist. To prevent performance degradation, a SmartConnect IP is added. The crossing is represented by Figure 3.2.

As we can see, the elements highlighted in pink is the DDR, the PCIe and the SmartConnect IP. The path is used to send the data from the XDMA IP to DDR and/or APM and crosses both the Static and the Reconfigurable region. SmartConnect IP is plugged to end-point IP, i.e. APM and DDR MIG.

3.2.1.2. Logic Design using IP Integrator

Creating the Design Hierarchy Since XPR is used in this thesis work, new hierarchy is created as shown Figure 3.3. The Figure show the top-level module of our design. As we can see, the hierarchy is divide in *base_region*, in other words the Static region, and *expanded_region*, i.e. the Programmable region.

Creating and Configuring the Logic Design Due to Vivado IP Integrator, the reference design has been modified to remove the DDRs, Flash memory and other useless component in our hardware infrastructure. Also, IP core configuration has been modified to fit our specification.

Configuring the Programmable region The number of kernel is set to 16, compared to the reference design in which it was set at 4. To fit the specification, number of slave DDR is 1. In reference design this value was 4. The address and data bit width for the AXI master interface. It is the maximum value that the

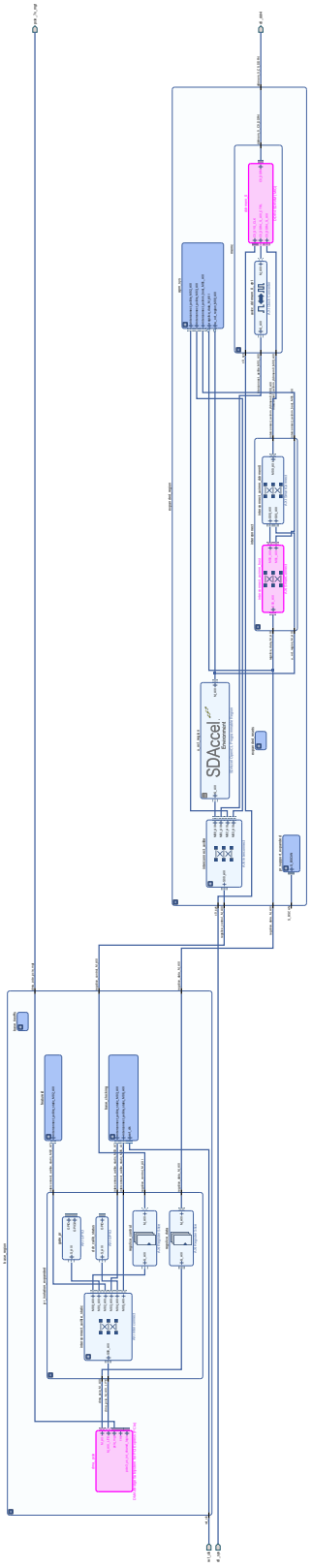


Figure 3.2: Use of SmartConnect IP

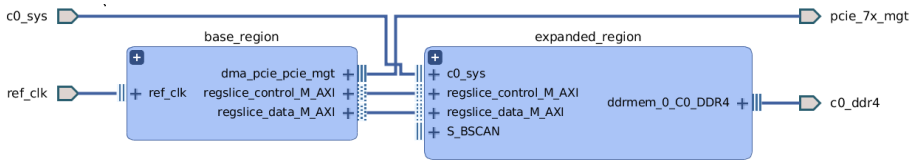


Figure 3.3: Top-Level Logic Hierarchy of our design

memory controller can support, i.e. 31. A very important option is Use Partial Reconfiguration. This option enable the RPR flow. It is clear that in our case it is disabled because the project specifications require the use of XPR.

Working with AXI Interconnect As the flow describe, two AXI Interconnects was used. The first was used to connect DMA/PCIe to SDAccel OpenCL Programmable region IP, as shown Figure 3.4. The second AXI Interconnect, shows in Figure 3.5, was used to:

- provide a path that bypasses the Programmable region, linking directly the DDR.
- link the Programmable region and DDR

The entities highlighted in pink in Figure 3.4 are the PCIe and the SDAccel OpenCL Programmable region IP.

The entities highlighted in pink in Figure 3.5 are the PCIe and the DDR MIG. in this case the IP Interconnect IP carries out the Programmable Region bypass. In fact, the IP Interconnect IP has two inputs: one coming from SDAccel OpenCL Programmable region IP and the other from DMA/PCIe.

Configuring the AXI Address Space For addressing OpenCL kernel, AXI Interconnect use a memory map methodology. This means that all the registers of the various components of our design are mapped in memory, i.e. in the same address space but at different addresses for each IP. Therefore, writing or reading to one of these addresses means communicating with that particular component. Figure 3.6 shows the current address space ranges for a custom Platform.

As we can see, this address space is divided in two main groups:

- **expanded_region/u_ocl_region**: allows user kernels to access 2GB of DDR memory, starting from offset 0x0.
- **base_region/dma_pcie**: allows the host to communicate with different components in the design. This group is also divided in two categories:
  - **base_region/dma_pci/M_AXI**: thanks to the DMA IP core, 64-bit address space is used to allow efficient memory data transfers to the host from and to the device memory. The DMA IP core can access 2 GB of DDR device memory. In this way, both the kernel and host can access the same memory address space, implementing what is called *Global Memory* in the OpenCL Memory Model.

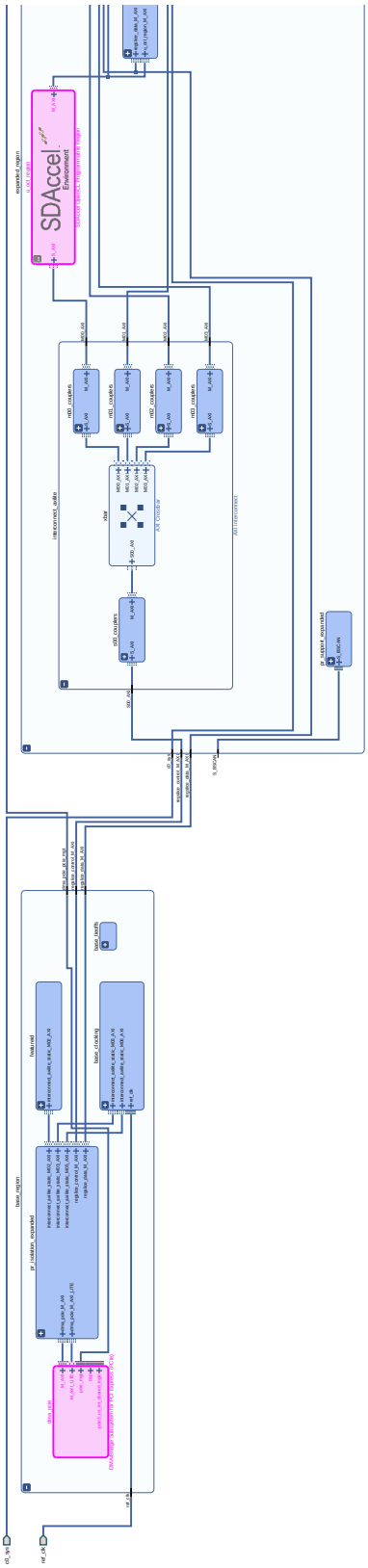


Figure 3.4: AXI Interconnect DMA/PCIe to SDAccel OpenCL Programmable region IP

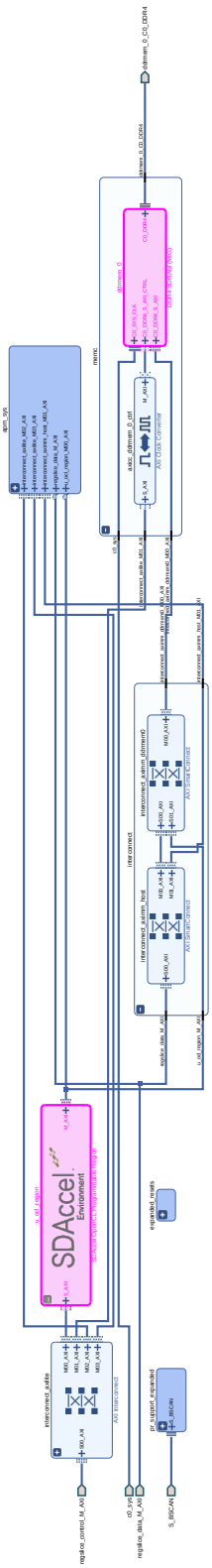


Figure 3.5: AXI Interconnect SDAccel OpenCL Programmable region IP to DDR and to bypass

Cell	Slave Interface	Base Name	Offset Address	Range	High Address
expanded_region/u_ocl_region					
M_AXI (31 address bits : 2G)					
expanded_region/memc/ddrmem_0	C0_DDR4_S_AXI	C0_DDR4_ADDRESS_BLOCK	0x0000_0000	2G	0x7FFF_FFFF
base_region/dma_pcie					
M_AXI (64 address bits : 16E)					
expanded_region/memc/ddrmem_0	C0_DDR4_S_AXI	C0_DDR4_ADDRESS_BLOCK	0x0000_0000_0000_0000	2G	0x0000_0000_7FFF_FFFF
expanded_region/apm_sys/xilmonitor_fifo0	S_AXI_FULL	Mem1	0x0000_0020_0000_0000	2G	0x0000_0020_7FFF_FFFF
M_AXI_LITE (32 address bits : 4G)					
base_region/base_clocking/clkwiz_kernel2	s_axi_lite	Reg	0x0005_1000	4K	0x0005_1FFF
base_region/base_clocking/clkwiz_kernel	s_axi_lite	Reg	0x0005_0000	4K	0x0005_0FFF
base_region/pr_isolation_expanded/ddr_calib_status	S_AXI	Reg	0x0003_2000	4K	0x0003_2FFF
expanded_region/memc/ddrmem_0	C0_DDR4_S_AXI_CTRL	C0_REG	0x0006_0000	128K	0x0007_FFFF
base_region/pr_isolation_expanded/gate_pr	S_AXI	Reg	0x0003_0000	4K	0x0003_0FFF
base_region/featureid/gpio_featureid	S_AXI	Reg	0x0003_1000	4K	0x0003_1FFF
expanded_region/u_ocl_region	S_AXI	Reg0	0x0000_0000	8K	0x0000_1FFF
expanded_region/u_ocl_region	S_AXI	Reg1	0x0000_2000	8K	0x0000_3FFF
expanded_region/u_ocl_region	S_AXI	Reg2	0x0000_4000	8K	0x0000_5FFF
expanded_region/u_ocl_region	S_AXI	Reg3	0x0000_6000	8K	0x0000_7FFF
expanded_region/u_ocl_region	S_AXI	Reg4	0x0000_8000	8K	0x0000_9FFF
expanded_region/u_ocl_region	S_AXI	Reg5	0x0000_A000	8K	0x0000_BFFF
expanded_region/u_ocl_region	S_AXI	Reg6	0x0000_C000	8K	0x0000_DFFF
expanded_region/u_ocl_region	S_AXI	Reg7	0x0000_E000	8K	0x0000_FFFF
expanded_region/u_ocl_region	S_AXI	Reg8	0x0001_0000	8K	0x0001_1FFF
expanded_region/u_ocl_region	S_AXI	Reg9	0x0001_2000	8K	0x0001_3FFF
expanded_region/u_ocl_region	S_AXI	Reg10	0x0001_4000	8K	0x0001_5FFF
expanded_region/u_ocl_region	S_AXI	Reg11	0x0001_6000	8K	0x0001_7FFF
expanded_region/u_ocl_region	S_AXI	Reg12	0x0001_8000	8K	0x0001_9FFF
expanded_region/u_ocl_region	S_AXI	Reg13	0x0001_A000	8K	0x0001_BFFF
expanded_region/u_ocl_region	S_AXI	Reg14	0x0001_C000	8K	0x0001_DFFF
expanded_region/u_ocl_region	S_AXI	Reg15	0x0001_E000	8K	0x0001_FFFF
expanded_region/apm_sys/xilmonitor_apm	S_AXI	Reg	0x0010_0000	64K	0x0010_FFFF
expanded_region/apm_sys/xilmonitor_fifo0	S_AXI	Mem0	0x0011_0000	4K	0x0011_0FFF

Figure 3.6: Memory address in the design

– **base_region/dma_pcie/M_AXI_LITE**: a 32-bit address space is used to access the other design components. In this way, the host can access the kernel control port by using 128KB address range when writing at I/O memory address 0x0. The whole space address dedicated to user OpenCL kernels is 128 kB. As we can see, since the number of kernel is 16, each kernel has a space address of 8 kB. Also, the host can access the kernel clock configuration ports through 4KB of allocated space for performing frequency scaling on the two available kernel clocks when writing at I/O memory address 0x0005_0000 and 0x0005_1000, respectively. In the same way that the one described here, the rest of components of the design have their own memory range and offsets to access them in the memory space defined.

Configuring the Design for SDAccel The other important IP cores for this kind of project are PCIe and DDR. The most important options of PCIe configuration is:

- third-generation.
- location set to *XOYO*.
- lane width set to 8.
- link speed set to 8 GT/s
- PR over PCIe.
- Device ID and Subsystem ID set to, respectively, 8237 and 4133.

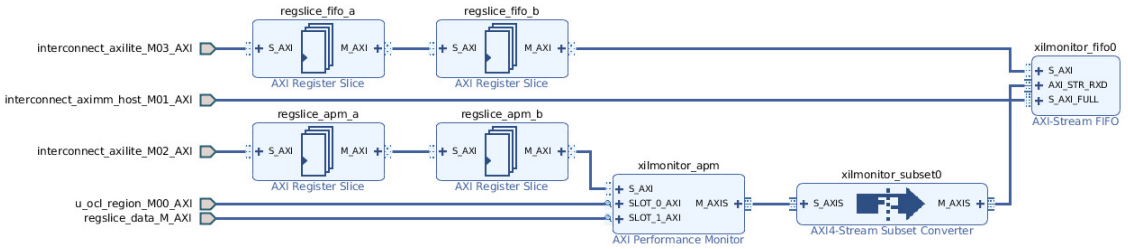


Figure 3.7: Logic module for profiling

- AXI Lite Master Interface size set to 4 MB, AXI Lite Master Interface address set to 0x0.
- number of H2C set to 2.
- number of C2H set to 2

The most important options of DDR MIG configuration is:

- he DDR provided by proDesing has a reference clock input of 125 MHz. Therefore the reference input clock speed was set to 125 MHz.
- memory part set to EDY4016AABG-DR-F.
- data with set to 512.
- address with set to 31

Using the Decoupler IP In the reference design, Decoupler IP is not used. Therefore in our design, Decoupler IP is not used.

Configuring Debug Logic In our design, a debug bridge is used. In order to use the debug bridge in Programmable region, therefore with XPR, we did same changes of reference design, described in [14]. In particular, an external interface was created in top level module *expanded_region* of type *xilinx.com:interface:bscan_rtl:1.0*. This interface was linked to debug bridge IP core.

Designing for Performance Profiling To profiling the system performance, an APM IP core was added. Figure 3.7 shows logical module which include the IP.

As we can see, the APM IP is linked to SDAccel OpenCL Programmable region IP and to DDR.

3.2.1.3. Applying Physical Design Constraints

I/O and Clock Planning As already said about PCIe location, we was forced to use *X0Y0* location to use the PR. In fact, as it is written in [24], the only location that allows the use of PR is *X0Y0*. This area is outside the Programmable region.

Applying Partial Reconfiguration Constraints

Definition of the Reconfigurable Region To use the XPR flow, the *HD.RECONFIGURABLE* PROPERTY is set *TRUE* and the *DONT_TOUCH* property was used to prevent optimization of the *expanded_region*. Below is reported the part of code of XDC file.

```
set_property DONT_TOUCH true [get_cells xcl_design_i/
   expanded_region/u_ocl_region]
set_property DONT_TOUCH true [get_cells xcl_design_i/
   expanded_region]
set_property HD.RECONFIGURABLE true [get_cells
   xcl_design_i/expanded_region]
```

Floorplanning Floorplanning is performed to assign logic modules to Pblock. Below is reported the part of code of XDC file.

```
# Expanded region pblock
create_pblock pblock_expanded_region
add_cells_to_pblock [get_pblocks pblock_expanded_region
   ] [get_cells [list xcl_design_i/expanded_region]]
resize_pblock [get_pblocks pblock_expanded_region] -add
    {SLICE_X0Y300:SLICE_X142Y599
   SLICE_X0Y180:SLICE_X133Y299
   SLICE_X0Y120:SLICE_X122Y179
   SLICE_X97Y60:SLICE_X122Y119 SLICE_X0Y0:SLICE_X95Y119
    SLICE_X97Y30:SLICE_X118Y59
   SLICE_X97Y0:SLICE_X117Y29}
resize_pblock [get_pblocks pblock_expanded_region] -add
    {BITSLICE_CONTROL_X0Y8:BITSLICE_CONTROL_X1Y15}
resize_pblock [get_pblocks pblock_expanded_region] -add
    {BITSLICE_RX_TX_X0Y52:BITSLICE_RX_TX_X1Y103}
resize_pblock [get_pblocks pblock_expanded_region] -add
    {BITSLICE_TX_X0Y8:BITSLICE_TX_X1Y15}
resize_pblock [get_pblocks pblock_expanded_region] -add
    {DSP48E2_X22Y12:DSP48E2_X22Y239
   DSP48E2_X0Y0:DSP48E2_X21Y239}
resize_pblock [get_pblocks pblock_expanded_region] -add
    {GTHE3_CHANNEL_X1Y20:GTHE3_CHANNEL_X1Y39
   GTHE3_CHANNEL_X0Y8:GTHE3_CHANNEL_X0Y39}
resize_pblock [get_pblocks pblock_expanded_region] -add
    {GTHE3_COMMON_X1Y5:GTHE3_COMMON_X1Y9
   GTHE3_COMMON_X0Y2:GTHE3_COMMON_X0Y9}
resize_pblock [get_pblocks pblock_expanded_region] -add
    {IOB_X2Y104:IOB_X2Y519 IOB_X0Y0:IOB_X1Y519}
resize_pblock [get_pblocks pblock_expanded_region] -add
    {LAGUNA_X22Y240:LAGUNA_X23Y479
   LAGUNA_X20Y120:LAGUNA_X21Y479
   LAGUNA_X0Y0:LAGUNA_X19Y479}
resize_pblock [get_pblocks pblock_expanded_region] -add
    {MMCME3_ADV_X0Y1:MMCME3_ADV_X1Y1}
```

```
14 resize_pblock [get_pblocks pblock_expanded_region] -add
       {PCIE_3_1_X0Y1:PCIE_3_1_X0Y5}
15 resize_pblock [get_pblocks pblock_expanded_region] -add
       {PLLE3_ADV_X0Y2:PLLE3_ADV_X1Y3}
16 resize_pblock [get_pblocks pblock_expanded_region] -add
       {PLL_SELECT_SITE_X0Y8:PLL_SELECT_SITE_X1Y15}
17 resize_pblock [get_pblocks pblock_expanded_region] -add
       {RAMB18_X15Y72:RAMB18_X17Y239
      RAMB18_X0Y0:RAMB18_X14Y239}
18 resize_pblock [get_pblocks pblock_expanded_region] -add
       {RAMB36_X15Y36:RAMB36_X17Y119
      RAMB36_X0Y0:RAMB36_X14Y119}
19 resize_pblock [get_pblocks pblock_expanded_region] -add
       {RIU_OR_X1Y4:RIU_OR_X1Y7 RIU_OR_X0Y0:RIU_OR_X0Y7}
20 resize_pblock [get_pblocks pblock_expanded_region] -add
       {SYSMONE1_X0Y1:SYSMONE1_X0Y1}
21 resize_pblock [get_pblocks pblock_expanded_region] -add
       {XIPHY_FEEDTHROUGH_X0Y1:XIPHY_FEEDTHROUGH_X7Y1}
22 set_property CONTAIN_ROUTING 1 [get_pblocks
      pblock_expanded_region]
23 set_property EXCLUDE_PLACEMENT 1 [get_pblocks
      pblock_expanded_region]
24 set_property SNAPPING_MODE ON [get_pblocks
      pblock_expanded_region]
25
26 # Lower SLR pblock
27 create_pblock pblock_lower
28 add_cells_to_pblock [get_pblocks pblock_lower] [
      get_cells [list xcl_design_i/expanded_region/
      u_ocl_region]]
29 add_cells_to_pblock [get_pblocks pblock_lower] [
      get_cells [list xcl_design_i/expanded_region/
      interconnect_axilite]]
30 add_cells_to_pblock [get_pblocks pblock_lower] [
      get_cells [list xcl_design_i/expanded_region/
      interconnect/interconnect_aximm_host]]
31 add_cells_to_pblock [get_pblocks pblock_lower] [
      get_cells [list xcl_design_i/expanded_region/
      interconnect/interconnect_aximm_ddrmem0/inst/
      s00_axi2sc]] -quiet
32 add_cells_to_pblock [get_pblocks pblock_lower] [
      get_cells [list xcl_design_i/expanded_r/egion/
      interconnect/interconnect_aximm_ddrmem0/inst/
      s00_entry_pipeline]] -quiet
33 #add_cells_to_pblock [get_pblocks pblock_lower] [
      get_cells [list xcl_design_i/expanded_region/
      interconnect/interconnect_aximm_ddrmem0/inst/
      s00_nodes]] -quiet
```

```
34 #add_cells_to_pblock [get_pblocks pblock_lower] [
      get_cells [list xcl_design_i/expanded_region/
      interconnect/interconnect_aximm_ddrmem0/inst/
      m00_exit_pipeline]] -quiet
35 #add_cells_to_pblock [get_pblocks pblock_lower] [
      get_cells [list xcl_design_i/expanded_region/
      interconnect/interconnect_aximm_ddrmem0/inst/
      m00_nodes]] -quiet
36 #add_cells_to_pblock [get_pblocks pblock_lower] [
      get_cells [list xcl_design_i/expanded_region/
      interconnect/interconnect_aximm_ddrmem0/inst/
      m00_sc2axi]] -quiet
37 add_cells_to_pblock [get_pblocks pblock_lower] [
      get_cells [list xcl_design_i/expanded_region/
      interconnect/interconnect_aximm_ddrmem0/inst/
      s01_nodes/s01_ar_node/inst/inst_mi_handler]] -quiet
38 add_cells_to_pblock [get_pblocks pblock_lower] [
      get_cells [list xcl_design_i/expanded_region/
      interconnect/interconnect_aximm_ddrmem0/inst/
      s01_nodes/s01_aw_node/inst/inst_mi_handler]] -quiet
39 add_cells_to_pblock [get_pblocks pblock_lower] [
      get_cells [list xcl_design_i/expanded_region/
      interconnect/interconnect_aximm_ddrmem0/inst/
      s01_nodes/s01_b_node/inst/inst_si_handler]] -quiet
40 add_cells_to_pblock [get_pblocks pblock_lower] [
      get_cells [list xcl_design_i/expanded_region/
      interconnect/interconnect_aximm_ddrmem0/inst/
      s01_nodes/s01_r_node/inst/inst_si_handler]] -quiet
41 add_cells_to_pblock [get_pblocks pblock_lower] [
      get_cells [list xcl_design_i/expanded_region/
      interconnect/interconnect_aximm_ddrmem0/inst/
      s01_nodes/s01_w_node/inst/inst_mi_handler]] -quiet
42 #add_cells_to_pblock [get_pblocks pblock_lower] [
      get_cells [list xcl_design_i/expanded_region/
      interconnect/interconnect_aximm_ddrmem0/inst/
      switchboards]] -quiet
43 resize_pblock [get_pblocks pblock_lower] -add {
      SLICE_X123Y180:SLICE_X133Y299
      SLICE_X119Y60:SLICE_X122Y299
      SLICE_X118Y30:SLICE_X118Y119
      SLICE_X97Y0:SLICE_X117Y119}
44 resize_pblock [get_pblocks pblock_lower] -add {
      DSP48E2_X22Y12:DSP48E2_X22Y47
      DSP48E2_X18Y0:DSP48E2_X21Y47}
45 resize_pblock [get_pblocks pblock_lower] -add {
      LAGUNA_X16Y0:LAGUNA_X19Y119}
46 resize_pblock [get_pblocks pblock_lower] -add {
      RAMB18_X15Y72:RAMB18_X17Y119
      RAMB18_X12Y0:RAMB18_X14Y47}
```

```
47 resize_pblock [get_pblocks pblock_lower] -add {
      RAMB36_X15Y36:RAMB36_X17Y59
      RAMB36_X12Y0:RAMB36_X14Y23}
48 resize_pblock [get_pblocks pblock_lower] -add {
      CLOCKREGION_X0Y2:CLOCKREGION_X4Y4
      CLOCKREGION_X0Y0:CLOCKREGION_X3Y1}
49 set_property SNAPPING_MODE ON [get_pblocks pblock_lower
      ]
50
51 # Upper SLR pblock
52 create_pblock pblock_upper
53 add_cells_to_pblock [get_pblocks pblock_upper] [
      get_cells [list xcl_design_i/expanded_region/memc/
      axicc_ddrmem_0_ctrl]]
54 add_cells_to_pblock [get_pblocks pblock_upper] [
      get_cells [list xcl_design_i/expanded_region/
      interconnect/interconnect_aximm_ddrmem0/inst/
      m00_exit_pipeline]] -quiet
55 add_cells_to_pblock [get_pblocks pblock_upper] [
      get_cells [list xcl_design_i/expanded_region/
      interconnect/interconnect_aximm_ddrmem0/inst/
      m00_nodes]] -quiet
56 add_cells_to_pblock [get_pblocks pblock_upper] [
      get_cells [list xcl_design_i/expanded_region/
      interconnect/interconnect_aximm_ddrmem0/inst/
      m00_sc2axi]] -quiet
57 add_cells_to_pblock [get_pblocks pblock_upper] [
      get_cells [list xcl_design_i/expanded_region/
      interconnect/interconnect_aximm_ddrmem0/inst/
      s00_nodes/s00_ar_node/inst/inst_mi_handler]] -quiet
58 add_cells_to_pblock [get_pblocks pblock_upper] [
      get_cells [list xcl_design_i/expanded_region/
      interconnect/interconnect_aximm_ddrmem0/inst/
      s00_nodes/s00_aw_node/inst/inst_mi_handler]] -quiet
59 add_cells_to_pblock [get_pblocks pblock_upper] [
      get_cells [list xcl_design_i/expanded_region/
      interconnect/interconnect_aximm_ddrmem0/inst/
      s00_nodes/s00_b_node/inst/inst_si_handler]] -quiet
60 add_cells_to_pblock [get_pblocks pblock_upper] [
      get_cells [list xcl_design_i/expanded_region/
      interconnect/interconnect_aximm_ddrmem0/inst/
      s00_nodes/s00_r_node/inst/inst_si_handler]] -quiet
61 add_cells_to_pblock [get_pblocks pblock_upper] [
      get_cells [list xcl_design_i/expanded_region/
      interconnect/interconnect_aximm_ddrmem0/inst/
      s00_nodes/s00_w_node/inst/inst_mi_handler]] -quiet
62 add_cells_to_pblock [get_pblocks pblock_upper] [
      get_cells [list xcl_design_i/expanded_region/
```

```
   interconnect/interconnect_aximm_ddrmem0/inst/
   s01_nodes/s01_ar_node/inst/inst_mi_handler]] -quiet
63 add_cells_to_pblock [get_pblocks pblock_upper] [
   get_cells [list xcl_design_i/expanded_region/
   interconnect/interconnect_aximm_ddrmem0/inst/
   s01_nodes/s01_aw_node/inst/inst_mi_handler]] -quiet
64 add_cells_to_pblock [get_pblocks pblock_upper] [
   get_cells [list xcl_design_i/expanded_region/
   interconnect/interconnect_aximm_ddrmem0/inst/
   s01_nodes/s01_b_node/inst/inst_si_handler]] -quiet
65 add_cells_to_pblock [get_pblocks pblock_upper] [
   get_cells [list xcl_design_i/expanded_region/
   interconnect/interconnect_aximm_ddrmem0/inst/
   s01_nodes/s01_r_node/inst/inst_si_handler]] -quiet
66 add_cells_to_pblock [get_pblocks pblock_upper] [
   get_cells [list xcl_design_i/expanded_region/
   interconnect/interconnect_aximm_ddrmem0/inst/
   s01_nodes/s01_w_node/inst/inst_mi_handler]] -quiet
67 add_cells_to_pblock [get_pblocks pblock_upper] [
   get_cells [list xcl_design_i/expanded_region/
   interconnect/interconnect_aximm_ddrmem0/inst/
   switchboards]] -quiet
68 resize_pblock [get_pblocks pblock_upper] -add {
   SLICE_X119Y300:SLICE_X142Y599}
69 resize_pblock [get_pblocks pblock_upper] -add {
   RAMB18_X15Y120:RAMB18_X17Y239}
70 resize_pblock [get_pblocks pblock_upper] -add {
   RAMB36_X15Y60:RAMB36_X17Y119}
71 resize_pblock [get_pblocks pblock_upper] -add {
   CLOCKREGION_X0Y5:CLOCKREGION_X4Y9}
72 set_property SNAPPING_MODE ON [get_pblocks pblock_upper
   ]
73
74 # APM pblock
75 create_pblock pblock_apm
76 add_cells_to_pblock [get_pblocks pblock_apm] [get_cells
    [list xcl_design_i/expanded_region/apm_sys/
   xilmonitor_apm/inst/
   GEN_PROFILE_Trace_Mode.profile_trace_mode_inst]]
77 add_cells_to_pblock [get_pblocks pblock_apm] [get_cells
    [list xcl_design_i/expanded_region/apm_sys/
   xilmonitor_subset0]]
78 add_cells_to_pblock [get_pblocks pblock_apm] [get_cells
    [list xcl_design_i/expanded_region/apm_sys/
   xilmonitor_fifo0]]
79 resize_pblock [get_pblocks pblock_apm] -add {
   CLOCKREGION_X3Y0:CLOCKREGION_X3Y1}
80 set_property SNAPPING_MODE ON [get_pblocks pblock_apm]
```

```
set_property PARENT pblock_lower [get_pblocks
   pblock_apm]

# DDR4 IP channel 0 pblock
create_pblock pblock_ddrmem_0
add_cells_to_pblock [get_pblocks pblock_ddrmem_0] [
   get_cells [list xcl_design_i/expanded_region/memc/
   ddrmem_0]]
resize_pblock [get_pblocks pblock_ddrmem_0] -add {
   CLOCKREGION_X2Y6:CLOCKREGION_X2Y9}
set_property SNAPPING_MODE ON [get_pblocks
   pblock_ddrmem_0]
set_property PARENT pblock_upper [get_pblocks
   pblock_ddrmem_0]
```

Partition Pin Assignment Partition Pin Assignment was used to assigned the Partition pins to Programmable region interface. We used the manually way to do it, therefore below is reported the part of code of XDC file

```
set_property HD.PARTPIN_RANGE {
   SLICE_X119Y60:SLICE_X122Y299
   SLICE_X123Y180:SLICE_X133Y299} [get_pins {
   xcl_design_i/expanded_region/*}]
```

3.2.1.4. Implementing Hardware Platform design

To meet all timing requirement, different strategy of synthesis and implementation was used. Strategy that didn't give timing issue was *Flow_PerfOptimized_high* for synthesis and *Performance_ExtraTimeOpt* for implementation.

3.2.1.5. Generating DSA files

Setting Vivado Properties for Generating a DSA Tcl script was created to set all necessary properties. Below is reported the *prepare_dsa.tcl* script.

```
# Set DSA project properties
set_property dsa.vendor "UPV" [current_project]
set_property dsa.board_id "prod-accel-profpga-ku115" [
   current_project]
set_property dsa.name "1ddr-xpr" [current_project]
set_property dsa.version "3.3" [current_project]
set_property dsa.uses_pr true [current_project]
set_property dsa.flash_interface_type "spix8" [
   current_project]
set_property dsa.flash_offset_address "0x4000000" [
   current_project]
```

```
set_property dsa.description "This platform targets the
    Prodesign Development Board for Acceleration with
   Kintex UltraScale KU115 FPGA. This high-performance
   acceleration platform features one channel of
   DDR4-2400 SDRAM, the expanded partial
   reconfiguration flow for high fabric resource
   availability, and Xilinx DMA Subsystem for PCI
   Express with PCIe Gen3 x8 connectivity." [
   current_project]

set_property dsa.pcie_id_vendor "0x10ee" [
   current_project]
set_property dsa.pcie_id_device "0x8237" [
   current_project]
set_property dsa.pcie_id_subsystem "0x4133" [
   current_project]
set_property dsa.board_interface_type "gen3x8" [
   current_project]
set_property dsa.board_memories {{ddr4 2GB}} [
   current_project]
set_property dsa.board_interface_name "PCIe" [
   current_project]
set_property dsa.board_vendor "prodesign-europe.com" [
   current_project]

set_param dsa.expandedPRRegion 1

set_param chipscope.enablePRFlow true
```

Generating DSA Once the circuit was implemented and the properties was set, the DSA was generated. To generate the DSA, we opened the implemented design and we didn't start the generation bitstream task. This because during the generating process, *write_bitstream* command was called. The following command was used to

- generate the DSA:

```
write_dsa -include_bit
```

  where *-include_bit* is used to include the bitstream for the current design in the DSA.

- validate the DSA:

```
validate_dsa
```

Device ID	Meaning
8	FPGA architecture is Ultrascale based
2	
3	
7	family series of FPGA

Table 3.1: Meaning of Device ID bits of custom Platform

Subsystem ID	Meaning
4	XPR is used.
1	1 DDR is used
3	DSA version
3	

Table 3.2: Meaning of Subsystem ID bits of custom Platform

3.2.2. Software Platform

Since the new custom Platform was created, Device ID and Subsystem ID was changed. The values chosen are 0x8237 and 0x4133 respectively. These values are compliant with the specification of this thesis work. Tables 3.1 and 3.2 explain the meaning of these values.

In order to make compatible the drivers, appropriate changes have been made.

3.2.2.1. xcldma

In this section, will be explain how *xcldma* driver was changed. For each change made, you can see the code in the original version, of course provided by Xilinx, and in the modified version.

All the changes were made on the *xdma-core.c* file.

Add custom Platform Device ID To make the connection between the driver and the custom platform, the Device ID has been added to the kernel mode driver in the *pci_device_id* data structure.

```
static const struct pci_device_id pci_ids[] = {
  { PCI_DEVICE(0x10ee, 0x7134), },
  { PCI_DEVICE(0x10ee, 0x7138), },
  { PCI_DEVICE(0x10ee, 0x8134), },
  { PCI_DEVICE(0x10ee, 0x8138), },
  { PCI_DEVICE(0x10ee, 0x8238), },
  { PCI_DEVICE(0x10ee, 0x8237), },
  { PCI_DEVICE(0x10ee, 0x8234), },
  { PCI_DEVICE(0x10ee, 0x8338), },
  { PCI_DEVICE(0x10ee, 0x8438), },
  { PCI_DEVICE(0x10ee, 0x8638), },
  { PCI_DEVICE(0x10ee, 0x923f), },
  { PCI_DEVICE(0x10ee, 0x943f), },
```

```
14   { PCI_DEVICE(0x10ee, 0x963f), },
15   { 0, }
16 };
```

System Monitoring support Since the custom platform does not include the hardware useful for providing platform contour information, SysMon features are not enabled.

```
1  bool is_sysmon_supported(const struct xdma_dev *lro)
2  {
3    if(lro->pci_dev->device == 0x8237 && lro->pci_dev->
       subsystem_device == 0x4133) return false;
4    u16 series = lro->pci_dev->device;
5    u16 dsanum = lro->pci_dev->subsystem_device;
6    series >>= 12;
7    series &= 0xf;
8    dsanum &= 0xff;
9    printk(KERN_DEBUG "SYSMON: Series: %u, dsanum: 0x%x.\
       n", series, dsanum);
10   return (series > 7) && (dsanum >= 0x32);
11 }
```

Multiple clock supported Since the custom platform has two clocks, one 250 MHz and the other 500 MHz, the driver must enable the functionality to support both clocks.

```
1  bool is_multiple_clock(const struct xdma_dev *lro) {
2    if ((lro->pci_dev->device != 0x8238) && (lro->pci_dev
       ->device != 0x8237))
3      return false;
4    if (((lro->pci_dev->subsystem_device & 0xff00) != 0
       x4400) && ((lro->pci_dev->subsystem_device & 0
       xff00) != 0x4100))
5      return false;
6    return ((lro->pci_dev->subsystem_device & 0xff) >= 0
       x31);
7  }
```

3.2.2.2. XCL HAL

In this section, will be explain how XCL HAL driver was changed. The changes concern the memory size and the name of the platform.

Add Platform name In this case the name of the platform and its Device ID-Subsystem ID pair has been added.

```
std::string XDMAShim::getDSAName(unsigned short
   deviceId, unsigned short subsystemId)
{
    std::string dsa("xilinx:?:?:?");
    const unsigned dsaNum = (deviceId << 16) |
       subsystemId;
    switch(dsaNum)
    {
    case 0x71380121:
        dsa = "xilinx:adm-pcie-7v3:1ddr:2.1";
        break;

    //cut

    case 0x82374133:
        dsa = "UPV:prod-accel-profpga-ku115:1ddr-xpr
           :3.3";
        break;

    //cut

    default:
        break;
    }
    return dsa;
}
```

Modify DDR size As stated in the specifications, the memory size is 2 GB. In this part of the code, you can see how the Device ID 0x8237 is associated with the value 0x080000000, which means 2147483648 bits and therefore 2 GB.

```
int XDMAShim::xclGetDeviceInfo2(xclDeviceInfo2 *info)
{
    std::memset(info, 0, sizeof(xclDeviceInfo2));
    info->mMagic = 0X586C0C6C;
    info->mHALMajorVersion = XCLHAL_MAJOR_VER;
    info->mHALMajorVersion = XCLHAL_MINOR_VER;
    info->mMinTransferSize = DDR_BUFFER_ALIGNMENT;
    info->mDMAThreads = mDataMover->channelCount();

    //cut

    if( (info->mDeviceId == 0x8238) ||
      (info->mDeviceId == 0x8638) ||
      (info->mDeviceId == 0x923F) ||
      (info->mDeviceId == 0x943F) ||
      (info->mDeviceId == 0x963F))
    {
```

```
18          info->mDDRSize = 0x100000000;
19      } else if(info->mDeviceId == 0x8237) {
20          info->mDDRSize = 0x080000000;
21      } else {
22          info->mDDRSize = 0x200000000;
23      }
24
25      //cut
```

How to generate the xbsak utility The XCL HAL driver folder is organized as shown in figure 3.8.

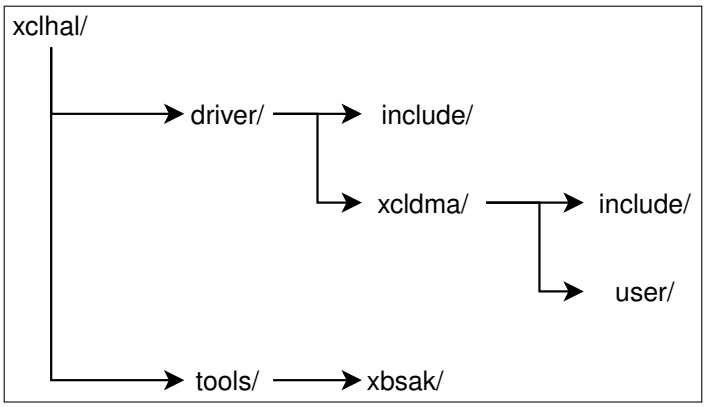


Figure 3.8: XCL HAL folder structure

The driver subfolders contain both the source files of the driver itself and the source files of the xbsak utility. The dependency between these two objects is given by the *libxcldrv.a* file. So first you need to generate the *libxcldrv.a* file. To do this just run the Makefile contained in the /driver/xcldma/user/ folder after modifying the driver to make it compatible with your custom platform. In this way, will be generated also the *libxcldrv.so* file. Now just run the Makefile in the /tools/xbsak/ folder.

3.2.3. Generating Platform files

As flow shown in figure 2.25 explain, the last step to create the Platform is to generate the Platform files in the right structured folder. In that case, the Platform folder was created as figure 3.9 shows.

The DSA file was copied in /hw folder, while the drivers was copied into /sw/driver folder.

3.2.3.1. Configuring the .xpfm File

The *UPV_prod-accel-profpga-ku115_1ddr-xpr_3_3.xpfm* file was copied from an available example platform in SDAccel and was changed in order to make it compatible with the custom Platform. The result is:

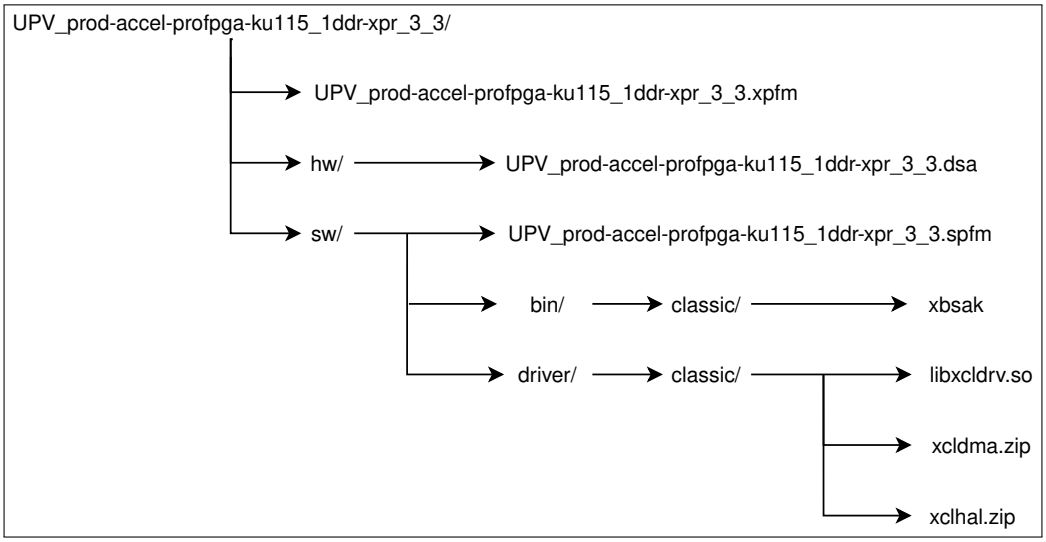


Figure 3.9: Custom Platform folder structure

```
<?xml version="1.0" encoding="UTF-8"?>
<sdx:platform sdx:vendor="UPV"
              sdx:library="prod-accel-profpga-ku115"
              sdx:name="1ddr-xpr"
              sdx:version="3.3"
              sdx:schemaVersion="1.0"
              xmlns:sdx="http://www.xilinx.com/sdx">
  <sdx:description>
  This platform targets the Xilinx Development Board
     for Acceleration with Kintex UltraScale KU115 FPGA
     . This high-performance acceleration platform
     features four channels of DDR4-2400 SDRAM, the
     expanded partial reconfiguration flow for high
     fabric resource availability, and Xilinx DMA
     Subsystem for PCI Express with PCIe Gen3 x8
     connectivity.
  </sdx:description>

  <sdx:hardwarePlatforms>
    <sdx:hardwarePlatform sdx:path="hw" sdx:name="
       UPV_prod-accel-profpga-ku115_1ddr-xpr_3_3.dsa"/>
  </sdx:hardwarePlatforms>

  <sdx:softwarePlatforms>
    <sdx:softwarePlatform sdx:path="sw" sdx:name="
       UPV_prod-accel-profpga-ku115_1ddr-xpr_3_3.spfm"/
       >
  </sdx:softwarePlatforms>

</sdx:platform>
```

Besides the Platform attribute, such as name and description, ti is possible to see the DSA path and the Software platform path in the Platform folder.

3.2.3.2. Configuring the .spfm File

To create the *UPV_prod-accel-profpga-ku115_1ddr-xpr_3_3.spfm* was followed the same flow as the previous file. The result is:

```
<?xml version="1.0" encoding="UTF-8"?>
<sdx:platform sdx:vendor="UPV"
              sdx:library="prod-accel-profpga-ku115"
              sdx:name="1ddr-xpr"
              sdx:version="3.3"
              sdx:schemaVersion="1.0"
              xmlns:sdx="http://www.xilinx.com/sdx">
    <sdx:description>
    This platform targets the Xilinx Development Board
       for Acceleration with Kintex UltraScale KU115
       FPGA. This high-performance acceleration
       platform features four channels of DDR4-2400
       SDRAM, the expanded partial reconfiguration flow
        for high fabric resource availability, and
       Xilinx DMA Subsystem for PCI Express with PCIe
       Gen3 x8 connectivity.
    </sdx:description>
    <sdx:systemConfigurations>
        <sdx:configuration sdx:name="linux_x86"
                           sdx:displayName="Linux on
                              x86">
            <sdx:description>Linux on x86</
               sdx:description>
        </sdx:configuration>
    </sdx:systemConfigurations>
</sdx:platform>
```

3.2.4. Installing the Platform

Once created the Platform, to use it you need to install it, i.e. to compile and install the driver, firmware and runtime libraries. In order to do that xbinst utility was used and the following command has been launched in the platform folder:

```
sudo xbinst -f UPV_prod-accel-profpga-ku115_1ddr-
   xpr_3_3.xpfm -d .
```

where:

- **-f UPV_prod-accel-profpga-ku115_1ddr-xpr_3_3.xpfm**: to point to the .xpfm file generated previously.

- **-d .**: to specify the output directory. In this case is the folder where the command was launched.

This phase generate a folder, called *xbinst*, which includes driver, firmware and runtime libraries. Also, it contains an installation script. To run this script, the command launched was:

```
sudo ./install.sh
```

Once the script finishes, a new script was generated. Its name is *setup.sh*. This script is used to set the environment variables of OpenCL and the path of library. You have to source this script every time you want to use the Platform.

3.2.5. Programming the FPGA

To program the FPGA, proFPGA tools was used. In particular, the following commands was performed:

- **profpga_builder**: to scan the hardware infrastructure and generate the file configuration which contains all used components. The tool is not used by GUI, but trough the command

```
profpga_builder --config-only --scan=169.254.0.2
```

  where the options have the following meaning:

  - **–config-only**: to generate the file configuration.
  - **–scan=169.254.0.2**: to scan the system with the specified IP address.

- **profpga_run**: to program the FPGA with a specified bitstream file. The bitstream file must be added to the configuration file by editing it in the part concerning the target FPGA. The used command was

```
profpga_run IP_169.254.0.2.cfg --up
```

  where the IP_169.254.0.2.cf argument is the configuration file name and the **–up** option is needed to initialize and configure the system based on the configuration file.

- **profpga_brdgen**: to generate the right psychical constraint about the hardware system. This software is used to understand where is mapped PCIe and DDRs. The used command was

```
profpga_brdgen IP_169.254.0.2.cfg --xdc
```

  where the IP_169.254.0.2.cf argument is the configuration file name and the **–xdc** option is needed to generate the .xdc constraint file used by Vivado.

3.3 Testing

3.3.1. Testing the correctness of the Platform installation

In order to test if the Platform was installed correctly, the xbsak utility was used. Particularly, the following commands was performed:

- to display a list of connected device:

```
./xbsak list
```

- since the custom platform is the only one connected to host, to get information about it:

```
./xbsak query -d 0
```

  where **-d 0** means that the first one of the list of device is queried.

- to test the PCIe bandwidth:

```
./xbsak dmatest -d 0 -b
```

  where **-b** specifies the test block size in KB. Default value is 65536.

- to write and read to/from DDR:

```
./xbsak mem --write -a 0x1000 -i 256 -e 0xAA
```

  where:

  - **–write**: to perform a write.
  - **-a 0x1000**: specifies the starting address.
  - **-i 256**: specifies the size in bytes.
  - **-e 0xAA**: specifies what to write.

```
./xbsak mem --read -a 0x1000 -i 256 -o read.out
```

  where:

  - **–read**: to perform a read.
  - **-a 0x1000**: specifies the starting address.
  - **-i 256**: specifies the size in bytes.
  - **-o read.out**: specifies the file name where direct the output.

- since the custom platform has a APM, to display the status of any APM and returns the value of the APM counters:

```
./xbsak status --apm
```

3.3.2. Partial Reconfiguration workaround

During the testing of custom Platform, the Partial Reconfiguration doesn't work. Particularly, the problem is about the calibration signal of DDR: after the downloading of partial bitstream, the calibration signal is low.

To workaround this problem, SDAccel Development Environment was modified. Specifically, SDAccel Development Environment generates the full bitstream and no the partial bitstream. To do so, the script *ocl_util.tcl* located in */path/to/Xilinx/folder/Xilinx/SDx/2017.1/scripts/ocl* has been modified. Below is reported the changes made (row $\sim$ 3350).

```
# pass -cell to write_bitstream to only generate the
   partial bit files for expanded pr
set more_option [get_property {
   STEPS.WRITE_BITSTREAM.ARGS.MORE OPTIONS} [get_runs
   impl_1]]
set_property -name {STEPS.WRITE_BITSTREAM.ARGS.MORE
   OPTIONS} -value"$more_option -cell
   $parent_rm_inst_path" -objects [get_runsimpl_1]
```

```
# pass -cell to write_bitstream to only generate the
   partial bit files for expanded pr
set more_option [get_property {
   STEPS.WRITE_BITSTREAM.ARGS.MORE OPTIONS} [get_runs
   impl_1]]
#set_property -name {STEPS.WRITE_BITSTREAM.ARGS.MORE
   OPTIONS} -value "$more_option -cell
   $parent_rm_inst_path" -objects [get_runs impl_1]
set_property -name {STEPS.WRITE_BITSTREAM.ARGS.MORE
   OPTIONS} -value"$more_option -no_partial_bitfile"
   -objects [get_runs impl_1]
```

3.3.3. Area performance evaluation

The most important advantage of using XPR is the percentage of resources used by the Static region. The reason why you want the percentage of resources used by the Static region to be low is that you want to offer as many resources as possible to the Reconfigurable region and SDAccel so that you can have more space for kernel implementation and optimization. Table 3.3 shows the values, taken from the Vivado utilization report (*report_utilization* command), available on Kintex UltraScale 115 FPGA and those used by the static region of the proposed solution. Figure 3.10 reports the chart of resource utilization.

An estimate of the total percentage of resources used by the Static region has been made by making the ratio between the sum of total resources and the sum of resources occupied by the Static region. The result is that about 2.44% of the resources are used by the Static region.

Resource	Utilization	Available	Utilization %
CLB LUTs	29502	663360	4.45%
CLB Registers	30126	1326720	2.27%
CARRY8	459	82920	0.55%
F7 Muxes	648	331680	0.20%
F8 Muxes	18	165840	0.01%
F9 Muxes	0	82920	0.00%
CLB	4722	82920	5.69%
LUT as Logic	26799	663360	4.04%
LUT as Memory	2703	293760	0.92%
LUT Flip Flop Pairs	12341	663360	1.86%
Block RAM Tile	49	2160	2.27%
DSPs	0	5520	0.00%
Bonded IOB	0	702	0.00%
HPIOB	0	598	0.00%
HRIO	0	104	0.00%
HPIOBDIFFINBUF	0	480	0.00%
HPIOBDIFFOUTBUF	0	480	0.00%
HRIODIFFINBUF	0	96	0.00%
HRIODIFFOUTBUF	0	96	0.00%
BITSLICE_CONTROL	0	192	0.00%
BITSLICE_RX_TX	0	1248	0.00%
BITSLICE_TX	0	192	0.00%
RIU_OR	0	96	0.00%
GLOBAL CLOCK BUFFERs	14	1248	1.12%
PLLE3_ADV	1	48	2.08%
MMCME3_ADV	2	24	8.33%
GTHE3_CHANNEL	8	64	12.50%
GTHE3_COMMON	2	16	12.50%
IBUFDS_GTE3	1	32	3.13%
OBUFDS_GTE3	0	32	0.00%
OBUFDS_GTE3_ADV	0	32	0.00%
PCIE_3_1	1	6	16.67%
SYSMONE1	0	2	0.00%
LAGUNA Registers	0	34560	0.00%
BSCANE2	0	8	0.00%
DNA_PORTE2	0	2	0.00%
EFUSE_USR	0	1	0.00%
FRAME_ECCE3	0	1	0.00%
ICAPE3	0	2	0.00%
MASTER_JTAG	0	2	0.00%
STARTUPE3	0	1	0.00%

Table 3.3: Static region resource utilization value

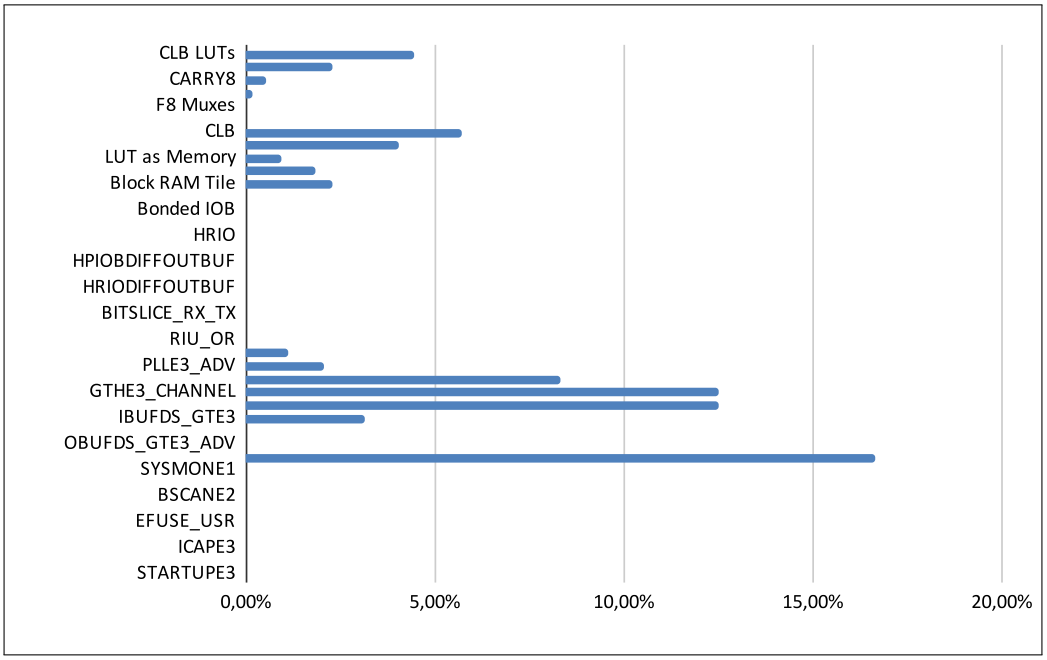


Figure 3.10: Static region resource utilization chart

As for the Reconfigurable region, this type of analysis is useless because the datapath of the Reconfigurable region depends on the number and type of kernel we want to accelerate.

3.3.4. PCIe speed performance evaluation

In order to testing the speed of PCIe communication from the host to the custom Platform, the *xbsak* tool was used. In particular, the command used is

```
./xbsak dmatest -d 0 -b xxx
```

where *xxx* is from 0x10 to 0x100000, which means that we started form a block size of 16 KB and ended with a block size of 1048576 KB.

The used command tests all channel in the design. In our design we have 2 channel for writing and 2 channel for reading. In Figure 3.11 and 3.12, we report the result of this evaluation.

We can deduce from the charts presented that the maximum performance of the PCIe, both in reading and writing and for all channels, can be achieved with a block size greater than 8192 KB.

This analysis is useful to try to optimize communication between host and device through the PCIe: a hypothetical kernel, in order to achieve good performance, must perform block transfers larger than 8192 KB.

The theoretical maximum bandwidth for communication via a third-generation 8-line PCIe is 15.8 GB/s for full-duplex communication [28]. Therefore, for half-duplex communication the theoretical maximum bandwidth is 7.9 GB/s. The reason for the difference between the theoretical value and the value collected in this test phase may be different, including:

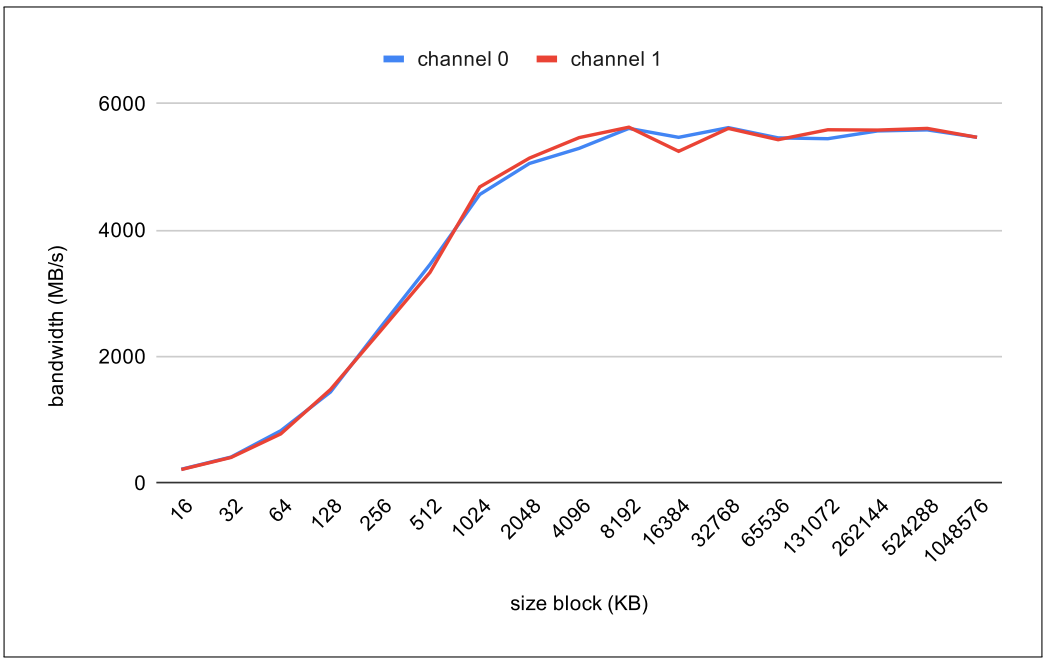


Figure 3.11: Results of test of reading

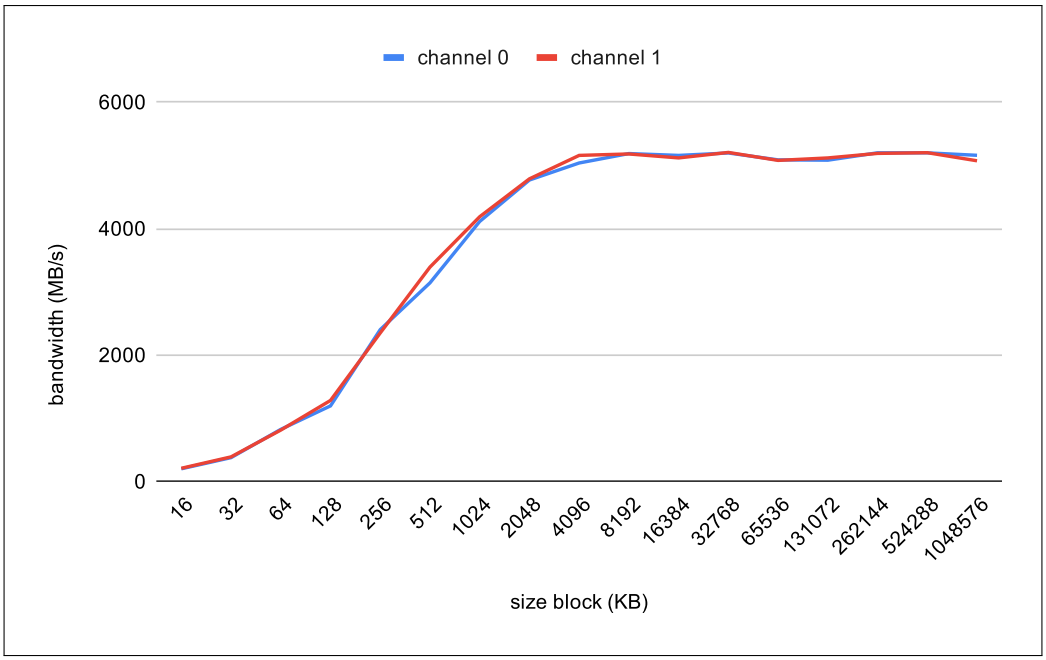


Figure 3.12: Results of test of writing

- kernel mode driver is set to work with interrupt. Xilinx recommends to prefer the polling mode to increase performance.
- limited number of DMA channel.
- Maximum Payload Size is set to 512 bytes. This number should be as large as possible. To check this value, the following command is performed:

```
$ lspci -d :8237 -xxxvvv
```

3.3.5. A real application: nnsim

nnsim is a neural network framework written in C, which allows to define, train, and infer neural network models. Its main purpose is to serve as a tool for teaching both machine learning and Deep Learning (DL) concepts and computer architecture.

nnsim includes kernel written in OpenCL can be accelerate with the custom Platform developed in this thesis work. Examples of function written in OpenCL are matrix multiplication, softmax, relu, maxpooling and demaxpooling.

nnsim's software architecture is abstractable as shown in Figure 3.13. We can see two levels: the first level is front-end to the end user, while the second level is interfacing with various computing units such as GPU, CPU and FPGA. In this case, the front-end represents the host, which manages the data transfer, the creation of OpenCL. Context, and also, from a Command-Line Interface (CLI), it is possible to choose which computing unit to use.

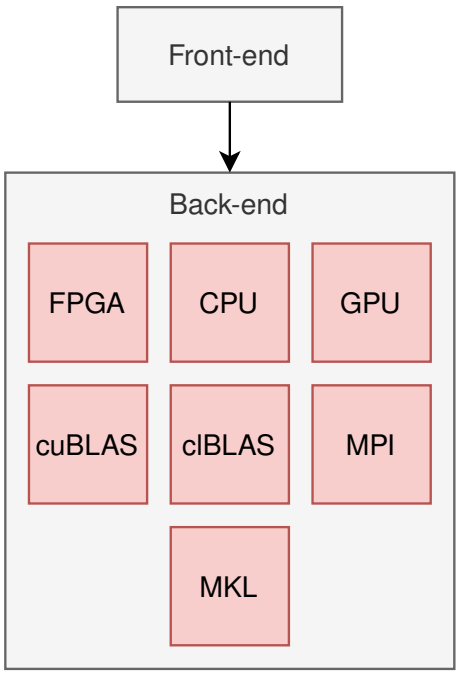


Figure 3.13: Logical view of nnsim's software architecture

What has been done, then, is to use the flow and the tools offered by Xilinx, specifically SDAccel Development Environment, and described in this thesis

work to deploy hardware accelerators capable of executing the functions written in OpenCL compliant with our custom Platform. To do this it needs to create the xclbin file and the bitstream file.

To easily create the xclbin file and bitstream file, a Makefile was written, shown below, which takes the kernel source code and, using the xocc compiler, produces the xclbin file and some estimates of resources used by kernels. The bitstream was used to program the FPGA by proFPGA tools and the xclbin was used to create the OpenCL Context in the host.

```
DEVICE := upv_prod_accel_profpga_ku115_1ddr
SDX_PLATFORM = UPV:prod-accel-profpga-ku115:1ddr-xpr
   :3.3
NUMBER_OF_DEVICES := 1
EMCONFIG_FILE = ./emconfig.json

XCLBIN := ./$(DEVICE)/$(TARGET)/binary
BUILD_DIR := ./$(DEVICE)/$(TARGET)
BUILD_DIR_KERNELS := $(BUILD_DIR)/kernels

#common tools
RM = rm -f
RMDIR = rm -rf

# compiler tools
XOCC ?= $(XILINX_SDX)/bin/xocc

#emulation tools
EMCONFIGUTIL = $(XILINX_SDX)/bin/emconfigutil --od .

# kernel compiler global settings
XOCC_OPTS = -t $(TARGET) --platform ./UPV_prod-accel-
   profpga-ku115_1ddr-xpr_3_3/UPV_prod-accel-profpga-
   ku115_1ddr-xpr_3_3.xpfm --save-temps  --report
   system -O3 --jobs 8
LDCLFLAGS += --xp prop:solution.hls_pre_tcl=synth.tcl

#
# OpenCL kernel files
#

BINARY_CONTAINERS += $(XCLBIN)/nnsim_binary_container.
   xclbin
BINARY_CONTAINER_KERNEL_OBJS += $(XCLBIN)/nnsim_kernels
   .xo

#
# host files
#

```

```
#
# primary build targets
#

.PHONY: all clean cleanall
all: $(BINARY_CONTAINERS) $(EMCONFIG_FILE)

clean:
  -$(RM) $(BINARY_CONTAINERS) $(EMCONFIG_FILE)
  -$(RM) *.rpt *.xtxt
  -$(RMDIR) _xocc*
  -$(RMDIR) .Xil

cleanall:
  -$(RMDIR) $(DEVICE)

.PHONY: incremental
incremental: all

#
# binary container: binary_container_$(DEVICE).xclbin
#

$(XCLBIN)/nnsim_kernels.xo: fpga_kernels.cl
  @mkdir -p $(@D)
  -@$(RM) $@
  $(XOCC) $(XOCC_OPTS) --temp_dir $(BUILD_DIR_KERNELS)
     -c -I"$(<D)" -o"$@" "$<"
  -@$(RMDIR) .Xil

$(XCLBIN)/nnsim_binary_container.xclbin: $(
   BINARY_CONTAINER_KERNEL_OBJS)
  @mkdir -p $(@D)
  -@$(RM) $@
  $(XOCC) $(XOCC_OPTS) --temp_dir $(BUILD_DIR_KERNELS)
     -l $(LDCLFLAGS) -o"$@" $(+)
  -@$(RMDIR) .Xil

#
# emulation configuration file
#

$(EMCONFIG_FILE):
  $(EMCONFIGUTIL) --platform ./UPV_prod-accel-profpga-
     ku115_1ddr-xpr_3_3/UPV_prod-accel-profpga-
     ku115_1ddr-xpr_3_3.xpfm --nd $(NUMBER_OF_DEVICES)
  -@$(RMDIR) TempConfig .Xil
```

Below, in Table 3.4 and 3.5, are the area and frequency estimates calculated by SDAccel Development Environment.

Kernel name	FF	LUT	DSP	BRAM
k_hadamard_product	3904	3230	27	2
k_im2col	31201	22874	64	2
k_maxpooling	10936	8094	60	2
k_demaxpooling	19197	15670	38	2
k_matmul	5339	4300	41	2
k_matadd_col	5920	5053	62	2
k_matmul_elwise	4320	3213	43	2
k_vect_scalar_prod	3817	3762	39	2
k_matsub	6269	4359	62	2
k_matadd	4208	3279	29	2
k_mat_reduce_rows	4483	4133	42	2
exp_generic_float_s	1899	1782	6	1
k_matrix_softmax	8695	8451	56	3

Table 3.4: nnsim's kernels performance estimate part 1

Kernel name	Total resources	Utilization %	Estimated frequency (MHz)
k_hadamard_product	7163	0,16%	342,4658
k_im2col	54141	1,23%	342,4658
k_maxpooling	19092	0,43%	342,4658
k_demaxpooling	34907	0,79%	342,4658
k_matmul	9682	0,22%	342,4658
k_matadd_col	11037	0,25%	342,4658
k_matmul_elwise	7578	0,17%	342,4658
k_vect_scalar_prod	7620	0,17%	342,4658
k_matsub	10692	0,24%	342,4658
k_matadd	7518	0,17%	266,6667
k_mat_reduce_rows	8660	0,20%	266,6667
exp_generic_float_s	3688	0,08%	342,4658
k_matrix_softmax	17205	0,39%	322,5807

Table 3.5: nnsim's kernels performance estimate part 2

To test the correct operation of the tool with the support of the Custom Platform, the following command has been launched.

```
./nnsim -net nets/mnist/mlp_tiny -dataset datasets/
   mnist/ -lr 0.01 -m 0.9 -mbs 64 -ne 1 -trs 64 -tes 64
    -opencl_fpga -fit_decoupled
```

The number of iterations, in this case, is 1. The output is shown below.

```
Start init
Platform name: Xilinx
```

```
Device name: UPV:prod-accel-profpga-ku115:1ddr-xpr:3.3
End init
WARNING: label text file not found, not using label
   texts...
Configuration (training):
Gradient descend method : minibatch
Minibatch size : 64
Learning rate : 0.0100 (constant)
Momentum : 0.9000
Gradient value clipping : 0.0000 not used
Loss function : cross entropy (ce)
L1 regularization : no
L2 regularization : no
Lambda : 0.0000
Block pruning lambda : 0.0000
Block pruning row size : 1
Block pruning col size : 1
Truncation threshold : 0.0000
Synthetic optimization function : None
Training dataset size : 64
Test dataset size : 64
Number of epochs : 1
Dataset directory : datasets/mnist/
Plot : None
Fused im2col : no
Load model : --
Save model : --
Generate confusion matrix image : no
Fit decoupled : yes
Preload images : no
Broadcast images : no
Disjoint training set : no
Chunk size (in items) : 64

# cut

Node: peak1.gap.upv.es epoch: 1/ 1 TRAIN: 64/ 64
   [**********]-> Cost: 2.2464 Acc: 10.9375 | VAL: 0/
   64 -> Cost: 0.0000Acc: 0.0000 | 0.02 MFLOPS - ETF:
   00:00:00 - batches 1 - time: process 670 us |
   compute 152120617 us
Node: peak1.gap.upv.es epoch: 1/ 1 TRAIN: 64/ 64
   [**********]-> Cost: 2.2464 Acc: 10.9375 | VAL: 64/
   64 -> Cost: 2.2532Acc: 23.4375 | 1.72 MFLOPS - ETF:
   00:00:00 - batches 0 - time: process 519425 us |
   compute 583441 us --fit time stats:
Shuffling : 6
Forward : 1167127
Loss : 73
```

```
Back : 44319
Update : 151492612
Bcast : 0
AllReduce : 0
Reduce : 0
Test : 0
Compute accuracy : 8
Total : 152705523

Time Init : 486 ( 0.00)
Time Process : 520095 ( 0.34)
Time Compute : 152704058 (100.00)
Time Stats : 519526 ( 0.34)
Total : 152705523 (100.0)
Timing stats. Node: peak1.gap.upv.es
--timing stats for functions:
matmul ( fpga): 5 calls, 70266 us ( 0.02), 14053.2002 us/call
matadd_col ( fpga): 3 calls, 1490304 us ( 0.49), 496768.0000 us/call
matmul_elwise ( fpga): 2 calls, 52 us ( 0.00), 26.0000 us/call
vect_scalar_prod ( fpga): 8 calls, 282659453 us ( 92.74), 35332432.0000 us/call
matsub ( fpga): 5 calls, 20372212 us ( 6.68), 4074442.5000 us/call
matadd ( fpga): 4 calls, 189 us ( 0.00), 47.2500 us/call
mat_reduce_rows ( fpga): 2 calls, 1100 us ( 0.00), 550.0000 us/call
matrix_relu ( fpga): 3 calls, 698 us ( 0.00), 232.6667 us/call
matrix_relu_der ( fpga): 2 calls, 405 us ( 0.00), 202.5000 us/call
matrix_softmax ( fpga): 3 calls, 192432 us ( 0.06), 64144.0000 us/call
xentropy ( cpu): 2 calls, 73 us ( 0.00), 36.5000 us/call
mat_copy ( cpu): 2 calls, 463 us ( 0.00), 231.5000 us/call
init_params ( cpu): 1 calls, 1267 us ( 0.00), 1267.0000 us/call
zero_vec ( cpu): 6 calls, 323 us ( 0.00), 53.8333 us/call
total time : 304789237 us

Parameter statistics:
Layer 1 weight statistics
```

```
77 max = 0.126193 min = -0.139922 histogram = [2, 44, 313,
       1051, 1943, 2204, 1519, 607, 139, 18, ] sparsity 0
      %
78 Layer 1 bias statistics
79 max = 0.000000 min = 0.000000 histogram = [7, 0, 0, 0,
      1, 0, 0, 1, 0, 1, ] sparsity 70 %
80 total sparsity 0 %
81
82 --fit time stats:
83 Shuffling : 0
84 Forward : 583422
85 Loss : 31
86 Back : 0
87 Update : 0
88 Bcast : 0
89 AllReduce : 0
90 Reduce : 0
91 Test : 0
92 Compute accuracy : 4
93 Total : 585227
94
95 Time Init : 742 ( 0.13)
96 Time Process : 840 ( 0.14)
97 Time Compute : 583422 ( 99.69)
98 Time Stats : 154 ( 0.03)
99 Total : 585227 (100.0)
100 ========================================
101 TEST RESULT
102 total cost = 2.2532 accuracy = 23.4375 %
103 ========================================
104
105 Global timing stats
106 Prep time : 152087142 us
107 Training time: 152705534 us (00:02:32)
108 Test time (final): 585240 us
109 Quantization time: 0 us
110 Total time : 305377916 us
111 ----------------------------
112 Total runtime : 306253603 us
```

As we can see, a complete execution takes about 5.1 minutes where the training time is about 2.32 minutes. An attempt was made to optimize performance, and it was thought to modify the kernels by adding the *pragma* provided by SDAccel Development Environment to add pipeling and unrolling, but the generic nature of the tool does not allow to know the exact size of the data at compilation time, or rather at kernel sisnthesis time, and for this reason the circuit can not be optimized.

CHAPTER 4
Conclusions

In this thesis work, an FPGA platform useful for the acceleration of kernels written in OpenCL, a high-level programming language, has been implemented. The goal in particular was to use the Xilinx OpenCL runtime on FPGA provided by a third party company as proFPGA.

The proposed solution allows the synthesis in an electronic circuit of any type of written kernel both with high level languages, such as C, C++ and OpenCL and with RTL languages. The choice of high-level languages also allows an average programmer to use this infrastructure. The main contribution of this thesis has been presented in Chapter 3. In this chapter the whole infrastructure development cycle has been described, starting from the Hardware Platform to the kernel mode driver modification. A reference design was used for the implementation of the custom platform, and modified to meet the required specifications. Also, Chapter 3 also includes several tests done on the platform. In particular, the correct functioning of the platform, the occupied area, the PCIe communication speed and the integration with a neural network simulator have been tested.

Bibliography

[1] TOP500. https://www.top500.org/.

[2] GREEN500. https://www.top500.org/green500/.

[3] Intel Corporation, Eriko Nurvitadhi, Ganesh Venkatesh, Jaewoong Sim, Debbie Marr, Randy Huang, Jason Gee Hock Ong, Yeong Tat Liew, Krishnan Srivatsan, Duncan Moss, Suchit Subhaschandra, Guy Boudoukh. *Can FPGAs Beat GPUs in Accelerating Next-Generation Deep Neural Networks?*. ACM/SIGDA International Symposium, February 2017

[4] MANGO project. http://www.mango-project.eu/.

[5] Ian Kuon, Russell Tessier, Jonathan Rose. *FPGA Architecture: Survey and Challenges*. Foundations and Trends in Electronic Design Automation Vol. 2, 2008

[6] Deming Chen, Jason Cong and Peichen Pan. *FPGA Design Automation: A Survey*. Foundations and Trends in Electronic Design Automation Vol. 1, 2006

[7] Xilinx. *UltraScale Architecture - Configurable Logic Block User Guide*. UG574 (v1.5) February 28, 2017

[8] Xilinx. *UltraScale Architecture - DSP Slice User Guide*. UG579 (v1.9) September 20, 2019

[9] Xilinx. *Xilinx Stacked Silicon Interconnect Technology Delivers Breakthrough FPGA Capacity, Bandwidth, and Power Efficiency*. WP380 (v1.2) December 11, 2012

[10] Intel Altera. *FPGA Architecture White Paper*. July 2006, ver. 1.0

[11] Florent De Dinechin, Bogdan Pasca. *Reconfigurable arithmetic for HPC*. High-Performance Computing using FPGAs, Springer, 2013.

[12] Khronos OpenCL Working Group. *The OpenCL Specification*. Version: 1.2 Document Revision: 19 Last Revision Date: 11/14/12

[13] Xilinx. *SDAccel Environment - Optimization Guide*. UG1207 v2017.1 June 20, 2017

[14] Xilinx. *Vivado Design Suite - User Guide Partial Reconfiguration*. UG909 v2017.1 April 5, 2017

[15] Xilinx. *Vivado Design Suite - User Guide Using the Vivado IDE*. UG893 v2017.1 April 5, 2017

[16] Xilinx. *SDAccel Environment - Tutorial Introduction*. UG1021 v2017.1 June 20, 2017

[17] Xilinx. *SDAccel Environment - User Guide*. UG1023 v2017.1 June 20, 2017

[18] PRO DESIGN Electronic GmbH. *proFPGA Hardware User Manual*. Version 3.6 / 2018-03-02 Documented release 2018A

[19] PRO DESIGN Electronic GmbH. *proFPGA Software User Manual*. Version 3.14 / 2018-03-15 Documented release 2018A

[20] PRO DESIGN Electronic GmbH. *proFPGA Builder User Manual*. Version 1.15 / 2018-03-08 Documented release 2018A

[21] Xilinx. *SDAccel Environment - Platform Development Guide*. UG1164 v2016.4 March 9, 2017

[22] Xilinx. *UltraScale Architecture and Product Data Sheet: Overview*. DS890 (v3.12) June 25, 2020

[23] Xilinx. *Vivado Design Suite - Tcl Command Reference Guide*. UG835 v2017.1 April 5, 2017

[24] Xilinx. *UltraScale and UltraScale+ FPGAs Packaging and Pinouts - Product Specification*. UG575 (v1.14) March 18, 2020

[25] KU115BOARD_SDAccel. `https://github.com/zakinder/SDAccel/tree/master/KU115BOARD_SDAccel`.

[26] Xilinx. *AXI Interconnect v2.1 - LogiCORE IP Product Guide*. PG059 December 20, 2017

[27] Xilinx. *DMA/Bridge Subsystem for PCI Express v3.1 - Product Guide*. PG195 June 7, 2017

[28] Jason Lawley, Xilinx. *Understanding Performance of PCI Express Systems*. WP350 (v1.2) October 28, 2014