FPGA Virtualization

PARMISS FALLAH
Abstract

During recent years, the population of internet users has sharply increased and more applications are utilizing data centers. Telecommunication companies are constantly trying to make communication applications faster to provide faster cellular networks. Moreover, recent computation applications have been analysing very big data and need higher performance. On the other hand, transistor scaling has almost come to its end, which makes it difficult to provide higher performance and efficiency in processors. As a solution, application-specific hardware platforms are used to accelerate the applications and improve the performance, energy consumption, and latency. Field programmable gate arrays (FPGAs) are the most popular means of hardware acceleration since they are reprogrammable and consume relatively low power. Connecting FPGAs to servers through high-speed PCIe links is the most common way of deploying them in data centers. However, FPGA resources will not be efficiently used if they are all assigned to one specific task or user. One task may just need a fraction of the FPGA resources. Therefore, a single FPGA can be shared among different applications in terms of area and time. FPGA virtualization is done by partitioning the fabric into isolated regions programmed dynamically based on active applications.

This thesis proposes a hardware architecture that enables an FPGA board to be deployed on a server. In this design, two isolated reconfigurable regions have been provided on the FPGA to accommodate accelerators. Our design is configured to accelerate Low-Density Parity Check (LDPC) encoding and decoding applications for 5G. However, the accelerators can be replaced by any other user-defined designs by simply reconfiguring a region on the FPGA. Compared to the Xilinx SDAccel reference platform for the same board, this design provides isolated regions with 25% more logic resources to be used by the users. Also, the data transfer latency has been significantly decreased to make the platform more compatible with the communication systems applications.
Sammanfattning


# List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>XDMA Example Design - Streaming mode</td>
<td>11</td>
</tr>
<tr>
<td>2.2</td>
<td>Host to Card Transfer Performance (Polling VS Interrupt) for 1,2, and 4 channels, source: <a href="http://www.xilinx.com/support/answers/68049.html">www.xilinx.com/support/answers/68049.html</a> [17]</td>
<td>12</td>
</tr>
<tr>
<td>2.3</td>
<td>Card to Host Transfer Performance (Polling VS Interrupt) for 1,2, and 4 channels, source: <a href="http://www.xilinx.com/support/answers/68049.html">www.xilinx.com/support/answers/68049.html</a> [17]</td>
<td>13</td>
</tr>
<tr>
<td>2.4</td>
<td>Communication System Block Diagram</td>
<td>14</td>
</tr>
<tr>
<td>2.5</td>
<td>SDaccel platform kernel-to-memory and host-to-memory connectivity [22]</td>
<td>15</td>
</tr>
<tr>
<td>2.6</td>
<td>Device View of a Design Connecting an XDMA block to four DDR4 memory controllers</td>
<td>16</td>
</tr>
<tr>
<td>2.7</td>
<td>Four Logical Layers of FPGA Compute Node, source: Fei Chen et al. 2014 [14], page 3</td>
<td>18</td>
</tr>
<tr>
<td>3.1</td>
<td>Block Diagram of the Overall Design</td>
<td>22</td>
</tr>
<tr>
<td>3.2</td>
<td>Clocking System Block Diagram</td>
<td>25</td>
</tr>
<tr>
<td>3.3</td>
<td>XDMA Data and Control Paths Connectivity</td>
<td>26</td>
</tr>
<tr>
<td>3.4</td>
<td>LDPC Block</td>
<td>29</td>
</tr>
<tr>
<td>3.5</td>
<td>Extracting LDPC Input Data from Host to Card Channel in a Kernel-specific Wrapper</td>
<td>30</td>
</tr>
<tr>
<td>3.6</td>
<td>Packing LDPC Output Data in Card to Host Channel in a Kernel-specific Wrapper</td>
<td>30</td>
</tr>
<tr>
<td>3.8</td>
<td>Final FloorPlan</td>
<td>34</td>
</tr>
</tbody>
</table>
List of Tables

2.1 KCU1500 Specifications ........................................ 4
3.1 Clock Domains ..................................................... 24
3.2 Management Unit Register File Address Map ................. 27
3.3 Achievable Configuration Speeds using different PCIe Interfaces 28
3.4 Address Map of each of the Hardware Components on its Corresponding Bus ........................................... 33
5.1 Resource Utilization of the Static Part .......................... 39
5.2 Power Consumption of the Static Part’s Components .......... 40
5.3 Power Consumption of the LDPC IP core ....................... 40
5.4 A Summary of Available Resources in each Partition .......... 41
5.5 LDPC Encoder/Decoder Resource Utilization (in 5G mode) .... 41
# Contents

1 Introduction

1.1 Motivation ................................................. 1
1.2 Thesis Statement .......................................... 2
1.3 Thesis Contributions ................................. 2

2 Background and Related Works ............................. 3

2.1 FPGA Architectures ........................................ 3
    2.1.1 Stacked Silicon Interconnect (SSI) Technology .... 3
    2.1.2 Xilinx KCU1500 acceleration board ................. 4
    2.1.3 Bus Protocols ...................................... 4
2.2 FPGA Virtualization Concept ............................ 5
    2.2.1 Multitenancy ....................................... 5
    2.2.2 Security and Isolation ............................ 6
2.3 FPGA Virtualization Use Cases ........................... 7
    2.3.1 Hardware Acceleration ........................... 7
    2.3.2 Hardware Access through the Cloud .............. 7
2.4 FPGA Deployment ......................................... 7
    2.4.1 System Bus ....................................... 7
    2.4.2 Peripheral Component Interconnect express (PCIe) . 8
    2.4.3 Ethernet ......................................... 8
    2.4.4 Examples in Industry ........................... 9
2.5 Xilinx PCIe DMA (XDMA) IP core ......................... 10
    2.5.1 XDMA Performance ............................... 10
2.6 Low-Density Parity Check (LDPC) ......................... 13
2.7 Related Works ............................................ 14
    2.7.1 SDAccel Platform ................................ 14
    2.7.2 Enabling FPGAs in the Cloud [14] ............... 16
3 Design Architecture

3.1 Design Guidelines ........................................ 21
3.2 A General Overview of the Design ....................... 22
3.3 Static Part ................................................... 22
  3.3.1 Communication Link ...................................... 22
  3.3.2 Clocking System ......................................... 24
  3.3.3 Isolator FIFOs .......................................... 25
  3.3.4 Management Unit ........................................ 26
  3.3.5 Reconfiguration Subsystem .............................. 27
3.4 Partitions .................................................. 28
  3.4.1 Accelerators Interfacing ................................. 28
3.5 Floor-planning .............................................. 30
  3.5.1 PCIe DMA Subsystem ..................................... 31
  3.5.2 DRAMs .................................................. 31
  3.5.3 Partitions ................................................ 32
3.6 Software Platform .......................................... 33
  3.6.1 Address Map ............................................ 33
  3.6.2 Driver .................................................. 33

4 Verification Methodology .................................. 37

5 Results and Discussion .................................... 38

5.1 Platform Features ......................................... 38
5.2 Resource Utilization of the Static Part .................. 39
5.3 Power Consumption of the Static Part .................... 39
5.4 LDPC IP Core Power Consumption ......................... 40
5.5 Available Resources in the Partitions .................... 40
5.6 Limitations ................................................ 41

6 Conclusions .................................................. 43

6.1 Summary ................................................... 43
6.2 Future Works .............................................. 43

Bibliography ................................................ 44
Chapter 1

Introduction

1.1 Motivation

In recent years, the population of internet users is constantly increasing as mobile devices are becoming more widespread. Therefore, the amount of data that needs to be transferred has also increased. On the other hand, the industry is expected to make communications faster with the development of the technology. To make the data center speed and capacity to keep up with the increasing amount of data, more servers have been added to data centers. However, this approach is limited by many factors, such as high power consumption rates and the capacity of the cooling equipment. Another alternative is to increase the clock frequency in which the available Central Processing Units (CPU) work, but scaling the frequency has become very slow recently because of the semiconductor’s limits [1]. A more practical way is to utilize the already available servers more efficiently by accelerating the processing algorithms. Instead of running software applications on a CPU with a general hardware architecture, application-specific hardware platforms can be used to significantly improve the performance, energy consumption, and latency.

In the most popular hardware acceleration approaches, Application-Specific Integrated Circuits (ASICs) and Field-Programmable Gate Arrays (FPGAs) are used. While ASICs are more specific and deliver better performance and power in a smaller area, they do not provide any flexibility which is a crucial drawback for rapidly-changing network applications. FPGAs are re-programmable and can achieve relatively good performance. Therefore, FPGAs are mainly used for acceleration in data centers.

Servers are currently using FPGAs as slaves to accelerate applications.
However, the traditional way of deploying FPGAs to speed up only one specific task wastes precious resources available on the FPGA. One accelerator may just need a small part of the whole die area, leaving the remaining parts free to be used by other accelerators. In addition, not all accelerators are active at all times. When an accelerator is in idle state, the FPGA resources assigned to it can be used by another application. Therefore, an FPGA can be shared among multiple applications to increase the utilization rate.

To sum up, hardware acceleration is used to compensate for the lack of computing resources compared to the constantly growing demand for data processing. FPGAs are mostly used for this purpose due to their flexibility, good performance, and efficiency. Finally, FPGAs need to be virtualized and shared among users or applications to get the best utilization rate.

1.2 Thesis Statement

FPGAs are widely used to accelerate data centers. However, FPGA resources are not being used efficiently in the traditional deployment methods [2]. One solution is to split the FPGA fabric into different regions and assign them dynamically to the applications. Therefore, a single FPGA can be shared in terms of area and time in order to maximize the utilization rate. In this thesis, this approach has been followed to manage the FPGA resources and utilize them to the highest capacity. Multiple applications share a board by accessing the host server. Applications run in completely isolated regions enabling the design in one region to be replaced by another design without any interference with what is running in the other regions.

1.3 Thesis Contributions

An FPGA board has been connected to a server and is shared among two independent users. Our design is configured to accelerate Low-Density Parity Check (LDPC) encoding and decoding applications for 5G. However, the accelerators can be replaced by any other user-defined designs by simply sending a request to the host server to reconfigure a region on the FPGA with another design.
Chapter 2

Background and Related Works

2.1 FPGA Architectures

FPGAs are integrated circuits designed to be programmable after being manufactured. They mostly consist of arrays of configurable logic blocks (CLB), block RAMs, and interconnects. CLBs consist of look-up tables (LUT) to create any arbitrary logic like a truth table followed by a flip-flop which can be used to achieve a sequential block. The main companies that manufacture FPGAs are Xilinx and Intel. Different FPGA families have been introduced, each optimized for a specific application.

Hardware description languages (HDLs) are used to create a design for FPGAs, the same as for ASIC design. Synthesis tools receive the HDL design files as inputs and generate a bitstream which is used for programming the FPGAs.

2.1.1 Stacked Silicon Interconnect (SSI) Technology

In the manufacturing process, several die slices are extracted from the same silicon wafer. Higher defect density in a wafer causes more chips to become defected and unusable. If very large chips are going to be manufactured, the number of dies that fit on one wafer will decrease. It implies that large die sizes significantly reduce the die yield when defects exist. Therefore, the manufacturability is lowered and time for volume productions is increased. Xilinx has implemented a solution to mitigate this issue by introducing SSI technology to keep the FPGA size large with manufacturing advantages of small chips[3].

In devices that use SSI technology, each FPGA die is a Super Logic Region
(SLR) component. All SLRs are combined and placed on a single passive silicon interposer using through-silicon vias and micro bumps. SLR components are stacked vertically, such that the bottom die is SLR0. Super Long Line (SLL) routes are located in the silicon interposer and provide paths for the signals crossing the SLRs[4]. The key challenges of this technology are:

1. The connectivity between the FPGA die slices is limited due to the insufficient number of I/Os.
2. The latency of the inter-die signals might affect the performance.

2.1.2 Xilinx KCU1500 acceleration board

In this project, a KCU1500 board [5] is used. The FPGA device on this board uses SSI technology and contains two super logic regions (SLRs). A summary of the board’s specifications which are of importance for us is shown in Table 2.1.

<table>
<thead>
<tr>
<th>Device Family</th>
<th>Kintex UltraScale</th>
</tr>
</thead>
<tbody>
<tr>
<td>PCIe connector</td>
<td>Gen3 x16 bifurcated into two x8 interfaces</td>
</tr>
<tr>
<td>Memory</td>
<td>4x DDR4 of 4GB each</td>
</tr>
<tr>
<td>Configuration Memory</td>
<td>1Gb Dual Quad SPI Flash</td>
</tr>
</tbody>
</table>

2.1.3 Bus Protocols

FPGA vendors apply various bus protocols for interfacing the design modules to each other. Application of a standard interface protocol enhances productivity and allows the developers to master only a single protocol for IP integration. Xilinx uses AMBA AXI protocol for its IP core’s interfaces and connecting components of a SOC, e.g. processors and FPGA fabrics, together. Intel, on the other hand, depends on its Avalon bus protocol [6] for interfacing modules on a bus and the AXI protocol for its IP core interfaces. A summary of the AXI4 protocol is explained further.

AXI4 is part of ARM’s Advanced Microcontroller Bus Architecture (AMBA) specification [7]. This protocol enables different design blocks to communicate with each other. It is implemented between a master and a slave module and provides a point-to-point communication strategy. A single AXI4 compatible
master module can communicate with multiple AXI4 compatible slave modules through an AXI4 interconnect. AXI4 interconnect creates a dedicated communication channel between the master and requested slave based on the address of the slave. Data transmission between the modules takes place at the end of a successful handshake operation between the master and slave. During this handshaking process, the master module announces that it wants to transfer data to/from the slave and waits for the slave module to get ready for the transmission. The protocol has five data and control channels: read address, write address, read data, write data and write response channels. There are three types of AXI4 interfaces[7]:

1. AXI4 enables high-performance memory-mapped data transfers.

2. AXI4-Lite is a memory-mapped interface with a narrower bandwidth compared to AXI4. It is appropriate for non-critical low-volume data transfers such as control signals.

3. AXI4-stream enables high-speed streaming of data.

2.2 FPGA Virtualization Concept

The emergence of the acceleration of the software applications by running them on custom hardware led the data centers to use FPGAs as hardware accelerators. When this solution was first being deployed, a whole FPGA used to be dedicated to one user [2]. With increasing the FPGA sizes and the number of users, the lack of an efficient scheme for assigning the resources was sensed. Since some devices are very large, users might not need the whole FPGA fabric for their designs. Therefore, a limited portion of the device is used and the rest of it remains unused. Similarly, a user might not use the FPGA constantly, thus the device will be unused during the period that one specific user is not using it. Hence, FPGA virtualization was introduced to enable the sharing of FPGAs in terms of resources and time. Virtualizing FPGAs provides an abstract layer to simplify the interface and hide the complexity of the system. It also enables the following features:

2.2.1 Multitenancy

A single FPGA must be shared among multiple users so that all of its available hardware resources can be used. To implement the area sharing mechanism, an FPGA fabric can be partitioned into multiple independent regions to be used
by different users. Partial reconfiguration has to be applied such that it enables the users to reconfigure their own region at run-time without interrupting the other parts of the design. This enables sharing the partitions in terms of time with the help of a scheduler to maximize the utilization rate. Furthermore, the bandwidth of the interconnection must also be shared. Assigning the bandwidth to the partitions can be done statically or dynamically and highly depends on the use-case and other factors within the system.

### 2.2.2 Security and Isolation

The device must be securely shared among the users so that one user can not access another user’s data or design. Security in such systems falls into three main categories:

1. **Bandwidth Sharing Security:** Users must not be able to access the other user’s dedicated partitions. The host or a controller unit must verify that the user has permission to access a particular address corresponding to a partition’s interface.

2. **Memory Virtualization Security:** Users must not be able to access the other user’s dedicated memory, in case each user has access to a part of an external memory module. Besides, when a region is reconfigured, either all the data owned by that region must be removed or the new design, which is now in the same region, must be prohibited from illegally accessing data remained from the previous user.

3. **Partial Reconfiguration Security:** Re-configuring the FPGA with a malicious bitstream might damage the device or crash the system [2]. Moreover, users must simply be able to reconfigure their own region. Therefore, in most systems, the design codes are taken from the users and the implementation and bitstream generation is done in a trusted system. Otherwise, a control mechanism must be applied to detect the malicious bitstreams before allowing it to program the FPGA.
2.3 FPGA Virtualization Use Cases

FPGA virtualization is mainly done for the following two purposes:

2.3.1 Hardware Acceleration

As Moore’s law and Dennard scaling are coming to their end [1], it is becoming very difficult to speed-up processors further due to physical limits. Therefore, new ideas are needed to accelerate the applications within the same semiconductor technology. Hardware acceleration in data centers and clouds is one of these ideas that has become a major trend in recent years. Almost all popular IT vendors and Companies, e.g. IBM, Amazon, Microsoft, etc, have proposed their own solutions to bring in cloud computing. Hardware platforms are being used in order to accelerate computation-intensive applications, such as signal processing algorithms.

2.3.2 Hardware Access through the Cloud

The increasing need for computing and storage resources brought up the idea of using and sharing hardware resources over the internet. This makes resources more easily accessible and re-usable. Data centers, which are available to the users over the internet, are referred to as cloud [8]. Cloud computing services fall into three main categories: Software as a Service (SaaS), Platform as a Service (PaaS), and Infrastructure as a Service (IaaS). SaaS makes applications available to the customers on its own infrastructure. PaaS provides hardware and software tools for application development, and IaaS provides virtualized computing resources, such as servers, processors and storage, over the internet. This provides users the freedom to choose a suitable hardware platform for running their applications. Hardware accelerators, such as virtualized FPGAs, are also provided in the cloud servers to be used by the users.

2.4 FPGA Deployment

There are 3 main ways of connecting FPGAs to the servers in data centers [9]:

2.4.1 System Bus

FPGAs can be placed on the same chip or board as the CPUs. In such systems, processors are weak, e.g. an ARM, and mostly used for control purposes. Both
Xilinx and Intel provide a system on a chip (SoC) that integrates FPGAs with CPUs. In some cases, the interconnect is slightly looser such as in Intel’s work which attaches a Xeon processor to an FPGA over their coherent low-latency quick path interconnect (QPI). However, this method has not been used in many data centers, since it has poor scalability and homogeneity which are important properties in the case of data centers. Moreover, the FPGA will be unused in case of a CPU failure.

2.4.2 Peripheral Component Interconnect express (PCIe)

CPUs can be connected to the FPGA cards through a high-speed communication link, i.e. PCIe bus. This method is the fastest and most common way of FPGA deployment. Today, PCIe bus Gen3 provides a bandwidth of 1 GB/s per lane. Therefore, a PCIe Gen3 x16 interconnect offers 16 GB/s of bandwidth in each direction. Furthermore, this high-speed link is easily achievable as it is now supported by many FPGA cards and servers. Recently, PCIe Gen4, which gives a bandwidth of twice wider than Gen3, has also been used on Xilinx acceleration cards [10].

A drawback of using this interconnect is that the FPGA utilization rate is affected by the server’s workload. Moreover, in order to implement an elastic network of FPGAs, either a high number of PCIe slots per server must be available or a network must be implemented among the FPGAs to let them communicate without accessing the server.

2.4.3 Ethernet

FPGAs can be directly connected to the network through an Ethernet interconnect. In this method, the FPGA usage is not bonded with the server’s resources. This approach also increases the elasticity of the system by enabling the FPGAs to communicate with each other, meaning that FPGAs can be used independently or together in order to accelerate an arbitrary application instead of a specific application that is running in the corresponding host server.

However, today, this interconnect provides a maximum bandwidth of 100 Gb/s which is still slower than the PCIe bus. On the other hand, an Ethernet subsystem IP core utilizes more resources compared to the PCIe subsystem IP, thus wastes valuable resources of FPGA fabric which could be used for computation.
2.4.4 Examples in Industry

Each cloud service provider has its own way of deploying the FPGAs in the cloud. Mostly, PCIe and Ethernet interconnections are used in various combinations with some advantages and disadvantages.

**Amazon F1 Instance**

An Amazon Web Services (AWS) instance is a server connected to a switch with Ethernet cable[11]. A maximum of 8 FPGAs is then connected to the server through the PCIe bus. FPGA devices are also connected to each other in a ring topology.

**Microsoft Catapult**

In the first version of this system [12], each server hosts one FPGA and is connected to the network switch via an Ethernet cable. In addition, all FPGAs are connected to each other in a 2D torus topology using the Intel SerialLite Protocol.

In Catapult V2 [13], a network switch is connected to the FPGAs through Ethernet. FPGAs are then connected to the servers directly via a PCIe link. With the help of this innovation, each server is able to use more than one FPGA. The FPGA devices can also be connected in a user-defined topology. However, the main drawback of this design is that the server is connected to the network through FPGA. In case of a failure in the FPGA or reprogramming the static part of it, the server would be lost from the network.

**IBM**

This company started with the conventional idea of having one PCIe-attached FPGA as a local accelerator on each server[14]. FPGAs were not connected to each other and all servers were connected to the network switch via Ethernet. They continued with this idea that a server can request to use another server’s local FPGA to increase resource utilization rate and system flexibility[15]. In this approach, the data is needed to be sent to the other servers through the network. They have introduced many algorithms to efficiently assign the FPGA resources to the requester servers.
2.5 Xilinx PCIe DMA (XDMA) IP core

Xilinx UltraScale devices Integrated Block for PCIe IP core is a high-speed serial interconnect for interfacing an FPGA device to a CPU[16]. This IP enables data transfer between the host CPU and the FPGA fabric by providing a PCIe interface to the host server. It also provides interfaces inside the FPGA fabric to transfer data to the user applications.

A DMA subsystem is further appended to the PCIe IP core to perform direct memory transfers between the PCIe IP and the user logic. Xilinx DMA subsystem for PCIe (XDMA IP) [16] enables direct data transfers between host memory and the FPGA card. It supports up to four data channels for each data direction, i.e. Host to Card (H2C) and Card to Host (C2H). The DMA subsystem can be configured to have a memory-mapped or streaming interface. In memory-mapped mode, one AXI-MM interface is shared by all read/write channels. While in streaming mode, one AXI stream interface is assigned to each enabled channel providing up to four read and four write AXI-stream interfaces. Figure 2.1 shows an example design for the XDMA IP in streaming mode, with four H2C and four C2H channel enabled. The incoming data is looped back to the host. In addition to the DMA interfaces, XDMA IP has two other interfaces which bypass the DMA engine:

- AXI-Lite Master, which is a 32-bit port suitable for non-critical access to design.
- DMA-Bypass Master, which is 256 bits wide and is supposed to make high-bandwidth access possible. However, a high-bandwidth transfer is also dependant on the XDMA driver and the Xilinx reference driver does not support it.

In Figure 2.1, the AXI-Lite master and Bypass master are enabled and connected to block RAMs.

2.5.1 XDMA Performance

Xilinx has published performance numbers for its PCIe DMA subsystem [17] as represented below. These numbers are extracted using the XDMA IP in memory-mapped mode and Xilinx XDMA reference driver.
CHAPTER 2. BACKGROUND AND RELATED WORKS

Hardware Performance
This number simply represents the DMA subsystem data rate excluding the impacts of the kernel driver and software. In this experiment, XDMA IP is configured to be a Gen3 x8 PCIe endpoint with 4 channels of C2H and H2C. The maximum data rate in channels of both directions is 7 GBps, which is achieved for transfer sizes of more than 1 KB.

Software Performance
This measurement represents the ratio of the amount of transferred data to the total transfer time. Therefore, the processing time of the user application and kernel driver are included in the data transfer time in addition to the DMA hardware latency. In this case, several factors affect the data throughput such as the host operating system, the driver, and the interrupt processing. For
instance, much lower throughput is achieved when the XDMA driver is used in the interrupt mode compared to the polling mode. The reason is that, after a transfer completion, the host receives an interrupt signal from the DMA and it waits for the interrupt service routine (ISR) to process the status which can take an unpredictably long time. In contrast, the data throughput is higher when the driver is used in polling mode since it constantly monitors the data transfer completion and does not need to process any interrupt signals.

Figures 2.2 and 2.3 illustrate the H2C and C2H channels performance of the PCIe DMA subsystem in polling and interrupt mode of the driver. Data has also been transferred through different numbers of channels. This reveals that the best performance is achieved when all four channels are involved in the transfer.

![Graph showing performance comparison]

Figure 2.2: Host to Card Transfer Performance (Polling VS Interrupt) for 1, 2, and 4 channels, source: www.xilinx.com/support/answers/68049.html [17]

**XDMA Performance in Streaming Mode**

As mentioned, the above numbers show the XDMA performance in memory-mapped mode. Although the IP has the same performance for streaming mode, the Xilinx XDMA driver is not optimized for the C2H streaming transfers. The data rate in H2C transfers can reach a maximum of 7 GBps. However, according to our experiments on streaming mode, the maximum throughput of the C2H
Figure 2.3: Card to Host Transfer Performance (Polling VS Interrupt) for 1, 2, and 4 channels, source: www.xilinx.com/support/answers/68049.html [17]

channel is approximately 15 times lower than the H2C channel in streaming mode. Therefore, in order to use the full capacity of the resources, either the driver has to be used in memory-mapped mode or it has to be modified for the C2H transfers in the streaming mode as in [18].

2.6 Low-Density Parity Check (LDPC)

Forward Error Correction (FEC) is a technique to enable the transmission of data over a noisy channel [19]. In FEC approaches, redundant data is appended to each packet to help to detect and correct the errors. LDPC code is an error-correcting code used to reduce data errors. LDPC code, which is in the same class as Turbo code, has recently become a more preferable choice for higher code rate range [20]. Figure 2.4 shows a digital communication system that uses LDPC encoder and decoder. The data packets are first encoded using LDPC codes. Then they are modulated and transferred over a channel. After demodulation, the packets are finally decoded with the help of LDPC codes. The decoder uses the redundant data to correct the data errors caused by the unreliable channel. Forward error correction’s success rate is lower in very noisy channels.

Data decoding algorithms fall into two categories of soft-decision and
hard-decision. In hard-decision systems, in spite of soft-decision ones, the demodulator makes a hard decision on whether a transmitted bit has been zero or one without giving more information about the reliability of the decision to the decoder. Thus, a demodulator can have a hard or soft value output data. Xilinx LDPC decoder [21] is a soft-decision decoder, meaning that it needs to be fed with soft value log-likelihood ratio (LLR) data -defined in eq2.1- as input. Since the encoded data is a hard data, it cannot directly be used as the decoder’s input data and must be modulated and demodulated (in an LLR demodulator) beforehand.

$$\text{LLR}(x) = \ln \left( \frac{\text{Pr}(x = 1)}{\text{Pr}(x = 0)} \right)$$  \hspace{1cm} (2.1)

2.7 Related Works

Some related works and papers are summarized below to give the reader an overview of the system complexity. Also, advantages and disadvantages of the related works are discussed.

2.7.1 SDAccel Platform

Xilinx has provided a reference design [22] to be used for acceleration with the KCU1500 board. This design has a "base region" consisting of a PCIe DMA IP to keep the link active and some clocking resources and control interfaces. The remaining parts of the fabric are referred to as "expanded region" which is just one partition. Four DDR4 memory controller IPs, all the user kernels (accelerators) and the interconnections among them are placed in this partition. A user-defined interconnect is used to connect kernels to the DRAMs in a many-to-many fashion, which allows arbitrary access from kernels to the global memory. However, more connectivity imposes challenges.
in routing and increases resource utilization. Thus connecting all kernels to all four DRAMs is not an efficient way of using the resources.

Since all of the kernels are in one partition, it is not possible to reconfigure a kernel individually. To change one kernel, the whole partition with all the DRAM controllers and kernels need to be reconfigured and the data inside the DRAMs will be lost. This is fine when all the kernels are working together and the whole FPGA is assigned to one user. Thus, multi-tenancy is not supported by this platform in its general meaning.

Figure 2.5 shows how DDR memories work as mediums between the PCIe link and accelerators in the SDAccel platform. The input data that comes from host to card through PCIe is primarily written into the DRAM. Then the accelerators read the data from DRAM. Similarly, kernels write the output data into the DRAM and the host reads the data afterward from the DRAM. The data transfer mechanism increases the latency of transferring the data from the host to accelerators and vice versa. Each access to the DDR4 memory has a latency of 72-112 clock cycles. This high transfer overhead makes the design work best when kernels have a high ratio of computing time to input and output data volume. It also works well when one kernel uses the output data of another kernel so that the data does not need to be written back to the host unnecessarily.

Figure 2.5: SDAccel platform kernel-to-memory and host-to-memory connectivity [22]

Following the Xilinx approach, a design with a PCIe endpoint connected to 4 DDR memories has been implemented. The resulted floor plan, illustrated
in Figure 2.6, shows that the usable area for implementing the computation logic, i.e. accelerators, is shrunk to less than 50% of the whole FPGA area. Moreover, all kernel-to-DRAM interconnections are placed in this area, which reduces the computing resources even more.

Figure 2.6: Device View of a Design Connecting an XDMA block to four DDR4 memory controllers

### 2.7.2 Enabling FPGAs in the Cloud [14]

This work has been done in IBM and Microsoft research groups. It proposes the following methods to meet four fundamental requirements to enable FPGAs in the cloud:
1. Resource abstraction: By providing an FPGA as a pool of predefined accelerators, which can be requested by a tenant. A tenant can also submit designs and the cloud owner performs the compilation for all compatible slots and adds it into the accelerator list.

2. Sharing: A virtualization mechanism is implemented to guarantee isolation.

3. Compatibility for different FPGA ecosystems: It is addressed through defining unified software-hardware interfaces.

4. Security: It is enhanced by controlling the DMA engine by a trustable hypervisor.

A prototype is implemented based on OpenStack, Linux-KVM and PCIe-attached Xilinx FPGAs. It enables partial reconfiguration and is comprised of four logical layers shown in Figure 2.7:

1. The hardware layer consists of
   (a) user layer, which includes the re-configurable partitions.
   (b) service layer, which includes reconfiguration controller, security controller, job scheduler, DMA engine, error detection, and MMIO registers
   (c) platform layer, including PCIe FPGA card, ICAP, DRAM, etc.

2. The hypervisor layer runs on the hardware, provides accessibility to the FPGA and finds the suitable FPGA resources for each application.

3. A library layer is used to manage FPGA bit files and provide APIs.

4. The application layer runs the OpenStack drivers and allocates the source and result data buffers needed for executing the application.

### 2.7.3 FPGA Resource Pooling in Cloud Computing [15]

In [15], Z. Zhu et al add another novelty to the work presented in [14]. In this paper, all FPGA accelerators are managed as a single resource pool and shared among all virtual machines (VM). In this case, cloud users can request FPGA acceleration from VMs in all machines, not just FPGA-attached machines. After a VM finishes using the FPGA accelerator, it releases the FPGA accelerator
back to the pool. The design has a controller/scheduler node and 5 compute
nodes, each containing an FPGA pooling Service Layer. Only 3 computing
nodes have a locally-attached FPGA. The scheduler categorizes the jobs into
two categories:

1. Jobs from the same node as the available accelerator.

2. Jobs from a node without FPGA resources.

3. Jobs from another FPGA-available node that has no idle resources in the
meantime. For this kind of job, the algorithm tends to let it wait for a
local accelerator, instead of assigning it to a remote node.

The scheduler uses the following Workload-and-Resource-Aware algo-
rithms to assign each job to a suitable FPGA. The two following strategies are
used to make the algorithm aware of the resources:
1. **FPGA Contention Awareness**: If a third kind of job has waited for a long enough time, it is probably from a busy-node, i.e., a node busy with large jobs. In that case, this job will be assigned with a high priority and get launched immediately to alleviate FPGA contention in the hot spot.

2. **Network Contention Awareness**: If the remote job number on a node outnumbers a threshold, the scheduler will constrain more remote jobs from launching on this node to balance the load of remote jobs equally among all FPGA-available nodes.

In addition, the workload is distributed by a Workload-Aware algorithm which imitates Shortest Job First (SJF) for jobs among different queues and FIFOs for jobs in the same queue.

### 2.7.4 Enabling Flexible Network FPGA Clusters in a Heterogeneous Cloud Data Center [23]

This work provides a way to orchestrate scalable and elastic FPGA clusters for large multi-FPGA and heterogeneous applications, such as a database query application. Dynamic cluster topologies are created using the data center network. The FPGA in this design is a PCIe-connected device. The Xilinx SDAccel platform is programmed onto the on-board flash memory. OpenCL is used to communicate to and manage devices through the API. Using a low-speed Ethernet core and the absence of a partially reconfigurable region are limitations of this design.

The prototype system maps FPGA kernels to devices, maps multi-FPGA topologies onto the network, and provides ways to scale up by replicating nodes and inserting schedulers to communicate with the replicated nodes. Kernels within a single FPGA are directly connected, whereas kernel connections between FPGAs are implemented via the network.

### 2.7.5 Hypervisor Mechanisms to Manage FPGA Reconfigurable Accelerators [24]

This work proposes a framework for hardware virtualization with dynamic partial reconfiguration accelerators using CPU-FPGA systems. FPGA virtualization has been done for servers, but it is not appropriate to follow the same approach for embedded systems due to their limited resources.

Ker-ONE is introduced as a small hypervisor which provides para-virtualization on ARM processors. It allows multiple guest OSes to be hosted concurrently.
with different priority levels. Each guest OS runs in an isolated domain named virtual machine. Critical tasks must be hosted in a high-priority VM at a higher speed. The virtual machine monitor (VMM) provides fundamental functions to manage the guests, such as scheduling and inter-process communication (IPC).

Partial reconfiguration interfaces (IF) are implemented on the FPGA side and are mapped to the physical memory space of the processor. Each IF is simply connected to a specific VM. In order to map an accelerator to a VM, an IF is mapped to the VM address space and connected to the target FPGA partition that implements the desired function. Then, the VM can control this accelerator by manipulating IF registers.

The requests coming from VMs that want to use the virtual devices are handled on the software side. On the hardware side, the static part of the FPGA monitors reconfigurable accelerators and tries to find appropriate solutions for VMs’ requests.

2.7.6 Limitations of the Similar Works

There have been many efforts to enable FPGAs as accelerators in the cloud. The Xilinx SDAccel hardware design [22] has been publicly published. It lacks multi-tenancy and has a very high data transfer overhead. Moreover, projects which use SDAccel as their base platform, such as [23], inherit its drawbacks. Unfortunately, the other similar works, such as [14], have not explained the realization of their hardware design in detail. Besides, they are mainly focused on bringing novelty to the software and cloud side of the system, while our purpose is to optimize the hardware design and efficiently utilize the resources.
Chapter 3
Design Architecture

3.1 Design Guidelines

This design is created with the help of Xilinx IP cores and custom HDL codes. The Xilinx Vivado tool provides an IP catalog that consists of many useful IP cores, such as PCIe and memory controller IP cores. Complex designs can be created using the Vivado IP integrator tool. It provides a graphic canvas for instantiating different design blocks and highly simplifies interconnecting the standard AXI interfaces. A less productive alternative for the GUI approach is instantiating the IP cores using TCL scripts and manually connecting all the interfaces in HDL codes. A combination of graphic block diagrams and HDL codes has been used in this project. Each IP core needs to be configured according to the project needs. Xilinx IP cores used in this design and a summary of their configurations are discussed further in this chapter. The accelerator wrappers and the design’s top module are done using Verilog HDL codes, whereas our management unit has been generated as a custom IP using IP packager. Design constraints (.xdc files) are also added to the project in order to map IO signals to the board pins and constrain the location of some design blocks.

The partial reconfiguration flow has been used for implementing this design. Three implementation configurations are introduced to generate the partial bitstreams for reconfiguring the partitions with grey box design, LDPC encoder, and decoder. The bitstreams are generated using TCL scripts and they are all in compressed format. The partial bit files are then converted to bit-swapped .bin files to have an Internal Configuration Access Port (ICAP)-friendly format. On the other hand, the static design bit files are converted to .mcs files so that they can be loaded on the onboard SPIx8 flash memory. In this way, the FPGA
will be programmed by the primary static design whenever it is turned on.

### 3.2 A General Overview of the Design

A Xilinx board is attached to a Dell server through the PCIe bus. It has a static part and two re-configurable partitions that can be programmed in the run time by the host. A total on-board DRAM memory of 8 GB is provided that is dynamically shared between the two regions, depending on how much memory they need. Xilinx LDPC encoder and decoder IP blocks have been configured for both of the regions to accelerate LDPC encode/decode applications for 5G.

![Figure 3.1: Block Diagram of the Overall Design](image)

### 3.3 Static Part

#### 3.3.1 Communication Link

A PCIe link has been preferred to be used over an Ethernet link in this design, due to its higher speed and smaller static design area.

The FPGA card is connected to the server through a PCIe link. Changes in the card’s PCIe subsystem after the link training are not expected. The server reboots and might even fail to boot up again if the PCIe endpoint gets corrupted.
or lost. Therefore, a PCIe subsystem must always be available on the FPGA to keep the link up. In this design, the Xilinx PCIe Direct Memory Access (DMA) subsystem is used to speed up data transfers to and from the host.

There are two options for transferring the data between the server and the accelerators: 1) Putting a memory block in between as a medium, 2) Directly connecting the DMA channels to the accelerators. The first approach, which is used in SDAccel, adds an unwanted latency and is not suitable for the designs with a high volume of data transfer between the server and accelerators. Since this project mostly targets communication systems with high data rates, the second option is adopted.

Since most of our target accelerators, such as LDPC, have streaming interfaces, XDMA IP has been configured to work in the streaming mode in this project. Using the memory-mapped mode would add another layer of bus protocol conversion between the DMA channels and the partitions, for which Xilinx AXI-Stream FIFO IP [25] could be used. Although the streaming mode is applied for the XDMA block in this design, the memory-mapped mode is going to be used in the future as this mode is more compatible with the Xilinx XDMA reference driver and has a higher C2H performance as discussed in Section 2.5.1.

This thesis does not implement dynamic bandwidth assignment to the partitions. Each partition has one C2H and one H2C DMA channels of 256 bits each. Therefore, a single accelerator can send/receive a total data-width of 256 bits per cycle. For the accelerators that have more than one interface, data can be packed together on the host side and be separated in the accelerator-specific wrapper. When a higher data-width is needed, data can be split into blocks of 256 and transmitted in more than one cycle.

The KCU1500 board has a PCIe Gen3 x16 connector bifurcated into two PCIe x8. In order to use all the 16 lanes, two instances of XDMA x8 IP could be used. This would provide up to eight C2H and H2C channels. However, the host server assigned to this project does not support PCIe bifurcation which makes us loose half of the bandwidth. This means that although this design can only have up to four H2C and C2H channels, it can be extended to support up to eight input and output channels using a server with bifurcation support.

The number of available channels limits the number of partitions that can be implemented in this design to four. Increasing the number of partitions makes the floor-planning more challenging and slightly expands the static part. Besides, each partition will have fewer resources. It is important to do the partitioning based on the resource utilization of the candidate accelerators. This prototype has two relatively large partitions to meet the resource needs of
a wider range of accelerators and simplify the floor-planning at the same time.

### 3.3.2 Clocking System

Table 3.1 presents brief information about available clock domains in this hardware platform. Independent clocks have been used for the partitions to reduce the size of the clock trees.

Each DDR instance has a dedicated 300 MHz differential clock input from the board. A PCIe 100 MHz differential ref clk signal is also provided by the board.

Table 3.1: Clock Domains

<table>
<thead>
<tr>
<th>Clock Domain</th>
<th>Source</th>
<th>Frequency</th>
<th>Usage in Platform</th>
</tr>
</thead>
<tbody>
<tr>
<td>XDMA</td>
<td>XDMA IP core</td>
<td>250 MHz</td>
<td>XDMA IP core, its AXI interfaces, and the management unit</td>
</tr>
<tr>
<td>DDR4 Memory Controller Clock (one per instance)</td>
<td>ddrmem_0 and ddrmem_2</td>
<td>300 MHz</td>
<td>DDR IP instances and their AXI interfaces connected to the corresponding smartconnect IPs</td>
</tr>
<tr>
<td>Partition 1</td>
<td>Clocking Wizard IP instance Clk_RR1</td>
<td>250 MHz</td>
<td>RR1’s accelerator, wrapper, and Isolating FIFOs of each corresponding AXI interface</td>
</tr>
<tr>
<td>Partition 2</td>
<td>Clocking Wizard IP instance Clk_RR2</td>
<td>250 MHz</td>
<td>RR1’s accelerator, wrapper, and Isolating FIFOs of each corresponding Advanced Extensible Interface (AXI) interface</td>
</tr>
<tr>
<td>ICAP</td>
<td>Clocking Wizard IP instance Clk_ICAP</td>
<td>125 MHz</td>
<td>ICAP and its corresponding FIFO</td>
</tr>
</tbody>
</table>
CHAPTER 3. DESIGN ARCHITECTURE

Figure 3.2 demonstrates that the clocks for XDMA, reconfigurable regions, and ICAP are all derived from the PCIe ref_clk. A single clocking wizard including three Mixed-Mode Clock Managers (MMCM) is used to provide these three clocks. The ref_clk drives clock buffer util_ds_buf_1 and its outputs are then fed to XDMA IP and another clock buffer. The second buffer, util_ds_buf_0, further drives the clocking wizard IP to generate three output clocks used for the partitions and ICAP.

A more flexible clocking system, such as the system used in SDAccel [22], would apply independent clocking wizards for each partition with the AXI-Lite control interface to enable arbitrary changes in the clock frequency of accelerators. In this design, however, partitions receive a constant clock of 250 MHz, which suits our target accelerators.

3.3.3 Isolator FIFOs

In this prototype, reconfigurable modules are fed from different clock sources with the same frequency of 250 MHz. DMA channels are also configured to work with a clock frequency of 250 MHz. Although these clock signals have the same frequency, their phase might be different. In addition, clocks for the accelerators might have various frequencies in the later projects.

FIFOs are used between the XDMA IP and the partitions to enable signals to cross the clock domains and avoid data loss. These FIFOs send and receive the data on two different clocks. As shown in Figure 3.3, a FIFO needs to be used on each enabled C2H and H2C channel.

On the other hand, XDMA IP needs to be isolated from changes to the reconfigurable regions. Otherwise, the link might get interrupted while a
partition is being re-configured. The same FIFOs are used as decouplers. By asserting the reset signals of the FIFOs connected to one region, all the data on that region’s interface will be ignored by the PCIe subsystem. Thus, a decoupling signal is sent to the management unit through the XDMA Lite interface before reconfiguring a partition to reset the corresponding FIFOs. After the reconfiguration, the reset signals will be de-asserted. This mechanism would also make sure that the FIFOs are empty by the time a new accelerator starts transferring data.

![Figure 3.3: XDMA Data and Control Paths Connectivity](image)

### 3.3.4 Management Unit

A control path has been provided to deliver the management signals to the FPGA design. A register file receives the control signal coming from the host server through the XDMA AXI-Lite master interface. This prototype offers a simple management unit which includes decoupling and reset signals for the accelerators. It enables the isolation of reconfigurable regions from the static part in case of a region re-configuration. Other features can be added in the future design, such as control interfaces used for selecting an arbitrary clock for the accelerators. Table 3.2 shows the addresses mapped to each control signal. Since these control signals are active-low reset signals, they are asserted and de-asserted by writing 0 and 1, respectively, into their corresponding address.
3.3.5 Reconfiguration Subsystem

There are two main ways of re-configuring the partitions on the FPGA fabric. One way is to program the partial bitstreams onto the device using the JTAG port. In this method, the server must be connected to the FPGA through a USB port and have Vivado installed on it. Therefore, it is not a scalable and data-center-friendly solution. The other way is to do the partial reconfiguration over the PCIe bus and through the ICAP port. This approach is more compatible with data centers since FPGA boards are connected to the servers through PCIe. In addition, the Media Configuration Access Port (MCAP) is a dedicated link to the FPGA's configuration engine from one specific PCIe endpoint per ultrascale device. Although this link is very efficient in terms of resource utilization, it has a bandwidth of 3-6 MB/s which makes configurations undesirably slow.

In this project, the FPGA board is a peripheral on the host’s PCIe bus. The static design is saved in the onboard dual-SPI flash memory and is downloaded on the FPGA as soon as it turns on. Then, the host can reconfigure the partitions by sending the partial bit files to the ICAP port over the PCIe bus. Configuration speed is very important for a user who needs to switch between specific accelerators too often.

ICAP is the fastest interface to the reconfiguration engine in PCIe-attached systems. It has a data width of 32 bits and supports a maximum frequency of 200 MHz for the monolithic devices and 125 MHz for the devices with SSI technology, including the device used in this project [26]. Sending the bit files through the PCIe DMA IP channels makes it possible to saturate the ICAP interface, leading to a speed of 500 MB/s for our device. In this design, one H2C XDMA Channel is dedicated to ICAP to make the reconfiguration as fast as possible.

However, ICAP can also be connected to other interfaces if all of the DMA channels needed to be dedicated to the accelerators’ data. An AXI-lite master interface of XDMA IP, with a data width of 32 bits, can be connected to the ICAP. However, since this interface bypasses the DMA, it can not saturate the

---

### Table 3.2: Management Unit Register File Address Map

<table>
<thead>
<tr>
<th>Address Offset</th>
<th>Control Signal</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0000</td>
<td>RR1_resetn</td>
</tr>
<tr>
<td>0x0004</td>
<td>RR1_decoupler</td>
</tr>
<tr>
<td>0x0008</td>
<td>RR2_resetn</td>
</tr>
<tr>
<td>0x000C</td>
<td>RR2_decoupler</td>
</tr>
</tbody>
</table>
ICAP. It takes about 20 cycles for each 32-bit-data to be transferred, lowering the configuration speed to 25 MB/s. Due to high configuration time, this approach is suitable for applications that reprogram the partitions less frequently. In addition, XDMA has another AXI-MM bypass interface with a higher bandwidth of 256 bits. Offering a bandwidth of eight times wider than AXI-lite, it can support a configuration speed of 200 MB/s. However, writing 256-bits-data onto the bypass interface requires modifying the reference driver.

The maximum size of a partial bitstream file is proportional to the size of the partition and it is 29.6 MegaBytes for the partitions of this design. The bit file size can be decreased significantly by compressing the bitstreams, making the file size to be proportional to the design size instead of the size of the partition assigned to it. The maximum size of a compressed partial bitstream in this design is 18.2 MB. Table 3.3 shows the maximum achievable configuration speed and time for each XDMA interface assuming that the maximum size of a bit file is 30 MB.

Table 3.3: Achievable Configuration Speeds using different PCIe Interfaces

<table>
<thead>
<tr>
<th>Interface</th>
<th>Configuration Speed</th>
<th>Configuration Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>DMA</td>
<td>500 MB/s (saturated)</td>
<td>60 ms</td>
</tr>
<tr>
<td>Bypass</td>
<td>200 MB/s</td>
<td>150 ms</td>
</tr>
<tr>
<td>AXI-Lite</td>
<td>25 MB/s</td>
<td>1200 ms</td>
</tr>
<tr>
<td>MCAP</td>
<td>6 MB/s</td>
<td>5000 ms</td>
</tr>
</tbody>
</table>

3.4 Partitions

3.4.1 Accelerators Interfacing

Xilinx LDPC IP core [21] is used as a default accelerator for this design. This core is implemented in 5G encoding and decoding modes for both of the partitions, making four different combinations. In other words, each partition can be configured to work as an encoder or decoder independent of the other. The IP is configured to use 5G initialized tables, fixed words configuration of 16 words, and normalized min-sum algorithm. With the mentioned configurations, the LDPC block will have four AXI4-stream interfaces shown in Figure 3.4:

- Control Input AXI4-Stream Interface (CTRL): For each packet, a single control input is required describing the packet-specific parameters, such
as the number of the parity bits. This interface determines the packet’s length. CTRL data-width is 40 bits in 5G mode.

- Data Input AXI4-Stream Interface (DIN): The input data packets are transferred through the DIN interface. The data-width that is transferred on each cycle is fixed at 128 bits for this design.

- Status Output AXI4-Stream Interface (STATUS): For each packet, a single status output is generated describing the packet-specific parameters, such as parity check pass/fail. STATUS data-width is 40 bits in 5G mode.

- Data Output AXI4-Stream Interface (DOUT): The output data packets are transferred through the DOUT interface. The data-width that is transferred on each cycle is fixed at 128 bits for this design.

![Figure 3.4: LDPC Block](image)

Each partition has only one dedicated input and one output AXI4-stream channel of 256 bits wide. However, 5G initialized LDPC block has two input streaming interfaces, CTRL and DIN. Therefore, the channel’s bandwidth is shared between them. A wrapper is used, in which the protocol signals are mapped accordingly and the data are extracted as illustrated in Figure 3.5. Data are received by the accelerator when both input interfaces are ready. Valid bit of each interface, \( V_{DIN} \) and \( V_{CTRL} \), is also included in the data packet. This wrapper has been designed specifically for LDPC decoder and the same approach might not work for other accelerators. Users are responsible to design their own wrapper for their accelerators.

Similarly, the C2H channel is also shared between DOUT and STATUS output streams, shown in Figure 3.6. In this case, data will be transferred when at least one output interface has valid data. The valid bits of DOUT and
3.5 Floor-planning

Several factors are taken into consideration while floor-planning this design. The modules must be constrained to specific places in the fabric, such that all the timing paths are met. Besides, the static part has to be as small as
possible to maximize the resources available for the accelerators. Different combinations of constraints lead to different designs with various parameters, e.g. area, delay, and power consumption. Therefore, exploration is needed to find the right placement for each module.

Floor-planning a device with SSI technology, such as the FPGA device used in this design, is very challenging. The SLR crossing resources are limited and the SLR crossing paths are relatively long compared to the routes inside a single SLR. Therefore, the inter-SLR paths commonly need to be pipelined to meet the timing constraints. The issue is that pipelining uses the precious resources of the FPGAs and increases the routing congestion in dense designs. Therefore, the best strategy is to place different design parts in a way that reduces the number of paths, particularly the timing critical paths, that cross the SLRs[4].

Design parts and options for placing them are mentioned in the following sections.

3.5.1 PCIe DMA Subsystem

The GTH transceivers that are allocated to the PCIe lanes are in the lower-right corner of the device. Clock regions X5Y0 and X5Y1 are where the transceivers allocated to the PCIe lanes 0-7 are placed. Transceivers assigned to the lanes 8-15 are included in the clock regions X5Y2 and X5Y3.

The XDMA block needs to be placed close to the GTH transceivers that are allocated to the PCIe lanes and the transceivers are in fixed places. Since the Xilinx XDMA block for UltraScale devices can be configured to have up to eight lanes, the block can be connected to the lower or higher eight lanes of the link. Therefore, it can be placed in the lowest or the middle part of the right edge in SLR0, shown in Figure 3.7a.

In this design, the XDMA IP is placed in the lowest part, which is a corner, in order to maximize the free contiguous space to be assigned to the partitions.

3.5.2 DRAMs

There are four DRAM memories available on the KCU1500 board. The ports allocated to the DRAMs have a fixed place on the FPGA device and each memory controller instance takes approximately three clock regions, as shown in 3.7b. However, only two DRAMs are used in this project. This provides flexibility in floor-planning based on which memory is used.

Using two DRAMs in the same SLR might destroy the balance of resources between SLRs. This placement is not desired if one partition is completely in
another SLR and tries to access the memories by crossing the SLRs. Therefore, the placement of the other modules has to be considered in order to take this approach.

In this design, DDR0 and DDR2 have been used so that the SLRs are balanced and the remaining spaces can be divided into two identical contiguous partitions. Although each partition is simply connected to one DRAM in this design, the expansion of partitions throughout the SLRs allows us to connect both partitions to both DRAMs without crossing the SLRs. Selecting DDR1 or DDR3 would have decreased the design symmetry. Using DDR3 would have caused the clock regions to the right of the DDR3 to fall apart and be wasted.

### 3.5.3 Partitions

This project aims to offer two partitions with an equal number of resources. These partitions are fed with data coming from the PCIe DMA IP, which is placed in SLR0. Hence, a part of each partition needs to be in SLR0 in order not to have SLR crossing long paths. In addition, the portion of the fabric area which is not occupied by the static part has to be completely assigned to the
partitions to minimize waste of resources. Considering all the conditions, the final floor-planning is done as illustrated in Figure 3.8. The partitions 1 and 2 are highlighted with green and purple, respectively. The remaining parts, which are in blue, show the static design including the clocking resources, PCIe IP, isolating FIFOs, and DRAMs.

3.6 Software Platform

3.6.1 Address Map

The server can access the hardware platform by writing to particular memory spaces. In order to transfer data through memory-mapped interfaces, the address which is going to be accessed must be specified. Vivado block diagram IP integrator provides an address editor tab, in which a specific address space is assigned to each memory-mapped interface of the design. Table 3.4 shows the address map for IP cores used in this design. There are three buses and each IP core uses a different bus. DRAM memory AXI-Lite CTRL interfaces are not used and thus excluded from the address map. As the table shows, the access link of each partition to their DRAM modules have the same address value. However, they are on two different buses and can be accessed simultaneously.

Table 3.4: Address Map of each of the Hardware Components on its Corresponding Bus

<table>
<thead>
<tr>
<th>Master</th>
<th>Slave</th>
<th>Offset Address</th>
<th>Range</th>
<th>High Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>XDMA IP core - M_AXI_Lite</td>
<td>Management Unit</td>
<td>0x0000_0000</td>
<td>64K</td>
<td>0x0000_FFFF</td>
</tr>
<tr>
<td>RR1 - M_AXI</td>
<td>DDR4_S_AXI_0</td>
<td>0x0000_0000</td>
<td>4G</td>
<td>0xFFFF_FFFF</td>
</tr>
<tr>
<td>RR2 - M_AXI</td>
<td>DDR4_S_AXI_2</td>
<td>0x0000_0000</td>
<td>4G</td>
<td>0xFFFF_FFFF</td>
</tr>
</tbody>
</table>

3.6.2 Driver

The server sends computing data and the partial bitstream to the FPGA through PCIe DMA streaming channels. In addition, control data are sent via the XDMA AXI-Lite interface. The Xilinx XDMA reference driver [27] has been used to enable the host to access XDMA. Although polling mode has better performance in CPU-attached systems, interrupt mode is used at this stage.
Commands which are used for initiating host-to-card and card-to-host transactions are, respectively, *dma_to_device* and *dma_from_device*. Here is a command initiating a data transfer on a H2C DMA channel:

```
./dma_to_device -d DeviceName -f FileName -s TransferSize -c NumberOf-
```
The Xilinx driver applications are programmed such that they need to know the exact transfer size. A sender module asserts a tlast signal on the last cycle of transferring each packet. This optional signal is provided by the AXI-streaming protocol and is used by the XDMA IP core. A tlast signal is asserted by the sender, which can be host or card, at the end of a single packet transfer. By default, the reference driver applications fail to complete a transaction unless the tlast signal’s value complies with the transfer size.

Luckily, the accelerator used in this project ignores the tlast signal. This enables the transfers from host to card to be easily done in one transaction (c=1) of the whole file size (s=filesize). However, transfer size on the C2H channels are not always known, as incoming packets might have different sizes. This problem is solved by modifying the driver application, such that it recognizes TransferSize as the maximum packet size. Then, after receiving the first interrupt, it attempts to transfer data for a number of times referred to as NumberOfTimesToRepeat. This number represents the number of packets. Here is a command which tries to receive 5 packets from the FPGA card, each with a maximum size of 3000 bytes:

```
./dma_from_device -d /dev/xdma0_c2h_0 -f data/RR1_dataOut.bin -s 3000 -c 5
```

reg_rwlock command is used for accessing the AXI-Lite and DMA-Bypass interfaces. The following is a command which writes a value into a specific address via the AXI-Lite interface. This command initiates the transfer of only one data and does not use the DMA.

```
./reg_rwlock /dev/xdma0_user Address DataWidth Value
```

DataWidth can get the values of 8, 16, or 32 bits which are respectively shown with 'b', 'h', and 'w'. Here is an example which writes 32-bit-data of 0 in address offset 4:

```
./reg_rwlock /dev/xdma0_user 0x0004 w 0
```

As mentioned, reg_rwlock is used for sending the control signals over the XDMA AXI-Lite interface. Since this interface is memory-mapped, the to-be-accessed memory address has to be specified. In-use addresses have been mentioned in Table 3.2.

In the following example, RR1 is assumed to be programmed with an LDPC decoder design. Here is a scenario that reconfigures this partition to an LDPC encoder. In UltraScale devices, the old design must first be cleared from the region and then the new design can be configured onto it. The first command decouples RR1 from the static part by asserting the reset
signal of its corresponding FIFO. Next, the clearing bitstream and the new accelerator bitstream are sequentially downloaded onto RR1. At last, RR1 has been connected to the static part again.

```
./reg_rw /dev/xdma0_user 0x0004 w 0
./dma_to_device -d /dev/xdma0_h2c_2 -f pr1_dec_partial_clear.bin -s 552992
./dma_to_device -d /dev/xdma0_h2c_2 -f pr1_enc_partial.bin -s 5394680
./reg_rw /dev/xdma0_user 0x0004 w 1
```
Chapter 4

Verification Methodology

As mentioned, the Xilinx LDPC IP core is used as an accelerator in this project to test the whole system functionality with a telecommunication use-case.

The input data to LDPC encoder and decoder blocks are extracted from simulating a Xilinx processor-based example design [21], which follows the conventional architecture of communication systems illustrated in Figure 2.4. Besides, a Xilinx MicroBlaze soft processor core [28] is instantiated in this design to run a C code that generates the test data. This software code allows the users to set the system parameters, e.g. number of packets, channel’s signal to noise ratio (SNR), etc, according to their needs.

The input and output data of the LDPC encoder and decoder are extracted from this design by pulling the desired signals out of the block diagram into the testbench, simulating the design, and writing the generated data in text files.

As for generating the test data, we disabled the channel noise to have predictable output data. Testing the fault-tolerance of the LDPC block is not in the scope of this work. The number of packets is set to 100 to generate input data files, in the format mentioned in 3.4.1, with a size of 170 KB for LDPC decoder. Input files are large enough to achieve the maximum throughput of the PCIe DMA subsystem discussed in Section 2.5.1. Finally, the validity of the output data is proved by comparing the output of hardware accelerators and simulated IP cores. The generated output file is approximately 70 KB.
Chapter 5

Results and Discussion

5.1 Platform Features

This thesis provides an acceleration platform with the following features:

- The acceleration board is connected to a server through a PCIe bus.
- Two isolated reconfigurable regions with one dedicated read and write PCIe DMA channel. Each channel is 256 bits wide.
- The base platform has been configured so that each partition can be configured into an LDPC encoder or decoder. In addition, regions can be re-configured by any other user-defined design.
- One external DRAM memory of 4 GB is dedicated to each of the partitions.
- A fast reconfiguration system is provided by using PCIe DMA to transfer bit files to the ICAP primitive, which saturates its bandwidth. It takes less than 60 ms to do a complete reconfiguration.
- Management unit for transferring the control signals to the platform.
- The system has a low data transmission latency since data is directly streamed to the accelerators through PCIe DMA channels.

A user, who wants to program a partition with an arbitrary design, has to upload the Verilog/VHDL codes or block diagrams of a design on the host. Users must wrap their designs in a specific I/O interface that is defined by us (the service provider). In this way, the host will be able to implement their
design using Vivado and generate the partial bitstreams that are also compatible with the onboard static part of the design. Obviously, due to security issues, the users must not have access to the reconfiguration subsystem. To reprogram their partition, they would need to send a reconfiguration request to the host and the host will do so if the bit file is valid. The implementation of the above processes is not in the scope of this thesis.

5.2 Resource Utilization of the Static Part

Table 5.1 reports the resource utilization of the design. To extract these numbers, partitions are configured with a grey box reconfigurable module (RM) so that they utilize a negligible amount of resources. Approximately 70% of the CLB and BRAM resources have been dedicated to the partitions. A fraction of the unused resources is not available to be used by the partitions since each partition must have a contiguous shape. Details on the available resources in each partition are mentioned in Section 5.5.

Table 5.1: Resource Utilization of the Static Part

<table>
<thead>
<tr>
<th>Resource Type</th>
<th>Used</th>
<th>Available</th>
<th>Utilization%</th>
</tr>
</thead>
<tbody>
<tr>
<td>CLB LUTs</td>
<td>83,198</td>
<td>663,360</td>
<td>12.54</td>
</tr>
<tr>
<td>CLB Registers</td>
<td>90,764</td>
<td>1,326,720</td>
<td>6.84</td>
</tr>
<tr>
<td>BRAM Tiles</td>
<td>146.5</td>
<td>2,160</td>
<td>6.78</td>
</tr>
<tr>
<td>DSP</td>
<td>6</td>
<td>5,520</td>
<td>0.11</td>
</tr>
<tr>
<td>Global Clock Buffers</td>
<td>27</td>
<td>1,248</td>
<td>2.16</td>
</tr>
</tbody>
</table>

5.3 Power Consumption of the Static Part

The design consumes a total power of 9.6 W when both partitions are configured with grey box RMs. However, the numbers are slightly inaccurate since grey box RMs are not empty and they insert tie-off LUTs to make sure that the outputs do not float [29]. Table 5.2 represents the share of on-chip components in total power consumption. Note that DRAM controllers are part of the static design.
Table 5.2: Power Consumption of the Static Part’s Components

<table>
<thead>
<tr>
<th>On-Chip Component</th>
<th>Power (W)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Clocks</td>
<td>0.764</td>
</tr>
<tr>
<td>CLB Logic</td>
<td>0.727</td>
</tr>
<tr>
<td>Signals</td>
<td>0.495</td>
</tr>
<tr>
<td>Block RAM</td>
<td>0.973</td>
</tr>
<tr>
<td>MMCM</td>
<td>0.373</td>
</tr>
<tr>
<td>PLL</td>
<td>0.459</td>
</tr>
<tr>
<td>I/O</td>
<td>1.849</td>
</tr>
<tr>
<td>GTH</td>
<td>2.215</td>
</tr>
<tr>
<td>PCIe Hard IP</td>
<td>0.298</td>
</tr>
<tr>
<td>Static Power</td>
<td>1.453</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>9.607</strong></td>
</tr>
</tbody>
</table>

5.4 LDPC IP Core Power Consumption

Table 5.3 shows the power consumed by the LDPC 5G encoder and decoder IP cores implemented in our design.

Table 5.3: Power Consumption of the LDPC IP core

<table>
<thead>
<tr>
<th>IP core</th>
<th>Power (W)</th>
</tr>
</thead>
<tbody>
<tr>
<td>LDPC Encoder</td>
<td>0.372</td>
</tr>
<tr>
<td>LDPC Decoder</td>
<td>1.433</td>
</tr>
</tbody>
</table>

5.5 Available Resources in the Partitions

This design includes two partitions covering 70% of the whole FPGA area. Table 5.4 implies that the partitions are almost identical in terms of resources. In addition, each partition has access to 4GB of DRAM memory. Although DRAM controllers are placed in the static part, they are used by the accelerators. DRAM controllers take more than 10% of the total area. The core static part, which mainly consists of PCIe DMA block, FIFOs, and ICAP, takes roughly 10% of the area. Some regions are not used by the static part, but they cannot be included in the partitions since the reconfigurable regions need to be contiguous. This explains why the total area usage of the static part and partitions, altogether, is less than 100%.
Table 5.4: A Summary of Available Resources in each Partition

<table>
<thead>
<tr>
<th>Resource Type</th>
<th>Partition 1 (% of total)</th>
<th>Partition 2 (% of total)</th>
</tr>
</thead>
<tbody>
<tr>
<td>CLB LUTs</td>
<td>205920 (31%)</td>
<td>260440 (39.3%)</td>
</tr>
<tr>
<td>BRAM Tiles</td>
<td>672 (31.1%)</td>
<td>864 (40%)</td>
</tr>
<tr>
<td>DSP</td>
<td>1968 (35.6%)</td>
<td>1862 (33.7%)</td>
</tr>
</tbody>
</table>

In this design, the LDPC core has been used for accelerating 5G encode and decode functions. The resource utilization of these blocks is shown in Table 5.5 as a percentage of available resources within region 1. The wrapper also utilizes a few negligible resources. It can be seen that the Xilinx LDPC IP core uses only one-fourth of a partition in the worst-case scenario (Decoder) in 5G mode, showing that the partitions are big enough for our target use-cases. The number of partitions could be higher to waste the least amount of resources. However, the design is limited by floor-planning considerations as well, mentioned in section 3.5. Therefore, two partitions are used to implement a prototype in this thesis.

Table 5.5: LDPC Encoder/Decoder Resource Utilization (in 5G mode)

<table>
<thead>
<tr>
<th>Resource Type</th>
<th>LDPC Encoder (% of partition1 resources)</th>
<th>LDPC Decoder (% of partition1 resources)</th>
</tr>
</thead>
<tbody>
<tr>
<td>CLB LUTs</td>
<td>14776 (7.2%)</td>
<td>52013 (25.3%)</td>
</tr>
<tr>
<td>BRAM Tiles</td>
<td>23.5 (3.5%)</td>
<td>119 (17.7%)</td>
</tr>
<tr>
<td>DSP</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

5.6 Limitations

The hardware Platform designed during this thesis has the following limitations that can be addressed in the future works:

- The partitions need to be created at the start of a design and cannot be changed afterward.

- The PCIe bandwidth is equally divided between the regions. It implies that even if one partition is not using the bandwidth at all, almost half of the link is always reserved for it and the other partition cannot use a larger portion of the bandwidth even if it needs.
• The DRAM modules are statically assigned to the regions. Therefore, a DRAM module that is assigned to region A can never be used by region B, even when accelerator A does not need it.

• As for this thesis, the DRAM values are not erased or put into a protected memory when the corresponding design is replaced by another design. This has to be done in the future to increase security.

• The maximum speed that is offered by the XDMA driver is not achieved because of the bad driver design in the streaming mode. This problem can be tackled in the future by using the XDMA IP in the memory-mapped mode.
Chapter 6

Conclusions

6.1 Summary

FPGAs are deployed in data centers as hardware accelerators. The main purpose of this thesis was to develop a hardware platform to enable the sharing of an FPGA among multiple applications, through virtualization. Therefore, FPGA internal resources, e.g. CLBs and BRAMs, are virtualized by spatial partitioning of the fabric. Furthermore, PCIe bandwidth (data link between the host server and the FPGA card) has been statically shared among the reconfigurable regions, ICAP module, and a management unit. Virtualization is also done in terms of external DRAM memories, by providing separate memory modules and controllers for each partition. The resource utilization has been maximized and the data latency is kept low by using the DMA subsystem to transfer data directly to the accelerators.

6.2 Future Works

This project can be followed by providing the dynamic dedication of the PCIe bandwidth and off-chip memory resources. Also, increasing the number of partitions is another area to be explored. One way to achieve this idea is allowing the users to instantiate a memory controller in their region (if needed).
Bibliography


