UC Irvine

UC Irvine Electronic Theses and Dissertations

Title

Field Programmable Gate Array (FPGA) Accelerator Sharing

Permalink

https://escholarship.org/uc/item/3vh8k8nr

Author

Rezaei, Siavash

Publication Date

2020

Copyright Information

This work is made available under the terms of a Creative Commons Attribution License, availalbe at https://creativecommons.org/licenses/by/4.0/

Peer reviewed|Thesis/dissertation

eScholarship.org Powered by the California Digital Library
University of California

UNIVERSITY OF CALIFORNIA,
IRVINE

Field Programmable Gate Array (FPGA) Accelerator Sharing

DISSERTATION

submitted in partial satisfaction of the requirements
for the degree of

DOCTOR OF PHILOSOPHY

in Computer Science

by

Siavash Rezaei

Dissertation Committee:
Professor Eli Bozorgzadeh, Chair
Professor Alexander V. Veidenbaum
Professor Ian G. Harris

2020

© 2020 Siavash Rezaei

DEDICATION

I dedicate my dissertation to my family. A special feeling of gratitude to my amazing parents, Saeed Rezaei and Soror Sohrabi, whose love, supports, and words of encouragement have always been pushing me toward success.

TABLE OF CONTENTS

Page

LIST OF FIGURES **v**

LIST OF TABLES **vi**

ACKNOWLEDGMENTS **vii**

VITA **viii**

ABSTRACT OF THE DISSERTATION **xi**

1 Introduction **1**
- 1.1 FPGA-based acceleration . . . 6
- 1.2 Sharing FPGA accelerators and challenges . . . 9
- 1.3 Overview and Contributions of this dissertation . . . 10

2 Background and Related work **15**
- 2.1 Overview of FPGAs Architecture . . . 15
- 2.2 FPGA Software Tools . . . 17
  - 2.2.1 Software design tools for FPGAs . . . 17
  - 2.2.2 Interface Schemes for FPGA-based Accelerators . . . 19

3 Scalable Multi-queue Scheme for Various Applications Seeking FPGA-based Accelerators **24**
- 3.1 Introduction . . . 24
- 3.2 Multi-Queue Multi-Accelerator Interface (MQMAI) . . . 27
  - 3.2.1 MQMAI framework . . . 29
  - 3.2.2 MQMAI Protocol . . . 33
- 3.3 Evaluation . . . 36
  - 3.3.1 Experimental Setup . . . 36
  - 3.3.2 Experimental Results . . . 38
- 3.4 Conclusion . . . 42

4 Fpga-based Dynamic Accelerator Sharing **43**
- 4.1 Introduction . . . 43
- 4.2 UltraShare: Multi-accelerator Framework . . . 45
  - 4.2.1 Multiple Command Queues . . . 45

4.2.2 Dynamic Accelerator Allocation 46
4.2.3 Scatter-Gather (SG) 47
4.2.4 Accelerators Controller 48
4.2.5 Data Transfer 49
4.3 Experimental Results 50
4.3.1 Experimental setup 50
4.3.2 Benchmarks 50
4.3.3 Results 51
4.4 Conclusions 57

5 UltraShare Express: Bandwidth Aware Ultimate Sharing of FPGA-based Accelerators 58
5.1 Introduction 58
5.2 Motivation 60
5.3 UltraShare Express: Policies and Protocols 62
5.3.1 Congestion-free Multi-core access 63
5.3.2 Accelerator Sharing 64
5.3.3 Minimizing Accelerators Stall Time 66
5.3.4 Dynamic data-link Usage Balancing 67
5.4 UltraShare: Hardware and Software Design 68
5.4.1 Software Stack 68
5.4.2 Hardware controller 71
5.5 Experimental 78
5.5.1 Experimental Setup 78
5.5.2 The impact of deploying a single command based structure 80
5.5.3 The impact of dynamic accelerator allocation 82
5.5.4 The impact of different multi-queue architectures in the software stack 83
5.5.5 The impact of enabling accelerator grouping in the hardware controller 84
5.5.6 Balancing data-link usage 86
5.6 Conclusion 88

6 Conclusions and Future Directions 89
6.1 Conclusion 89
6.2 Directions for Future Work 91
6.2.1 Immediate extensions of this dissertation 92
6.2.2 Novel research directions 93

Bibliography 94

LIST OF FIGURES

Page

1.1 FPGAs being used for acceleration in cloud . . . 3
1.2 Multiple FPGA accelerators being used for acceleration in clouds and datacenters where multi-core processors trying to invoke accelerators . . . 9

2.1 Internal architecture of a typical FPGA. . . . 16

3.1 The multi-queue structure of MQMAI . . . 25
3.2 Block diagram hardware architecture of MQMAI. . . . 32
3.3 Timing diagram of UltraShare Express from a user application requesting an accelerator process until the request being processed . . . 35
3.4 Experimental platform. . . . 37
3.5 Pseudocode of the threads. . . . 38
3.6 Throughput for different request sizes in Single-Thread Single-Accelerator case. 39

4.1 UltraShare hardware parts and components. . . . 46
4.2 Accelerator controller and interleaved RX/TX SG manager . . . 48
4.3 UltraShare LUT utilization . . . 53
4.4 UltraShare BRAM utilization . . . 54
4.5 Exploiting parallelism in UltraShare . . . 55
4.6 AES accelerator sharing among different applications submitting three different video resolutions . . . 56
4.7 Normalized AES accelerators usage by three different applications submitting three different video resolutions . . . 56

5.1 High-level block diagram of UltraShare Express . . . 63
5.2 The throughput of different accelerator types with different processing delay (presented as clock cycle) for different number of instances implemented on a single FPGA. . . . 67
5.3 UltraShare hardware controller . . . 72
5.4 Detailed components of Accelerators controller and Data manager and their interconnections . . . 74
5.5 Accelerators throughput (req/sec) of submission queue per core vs. submission queue per accelerator type. . . . 83
5.6 Accelerators throughput for different number of accelerator instances . . . 86
5.7 Fair allocation of PCIe bandwidth among accelerator groups . . . 87

LIST OF TABLES

Page

1.1 Comparing different aspects of hardware accelerators [9]. 4

2.1 Bandwidth of different PCIe generations. 17

3.1 An UltraShare command structure 30
3.2 Average end-to-end delay in Multi-Thread Single-Accelerator case. 40
3.3 Average end-to-end delay in Multi-Thread Multi-Accelerator. 41
3.4 Resource utilization on a Xilinx Virtex 7 FPGA for 4 I/O channels. 42

4.1 Throughput of different accelerators for UltraShare vs. single-queue non-grouping structure 53

5.1 UltraShare Express software stack API table 70
5.2 The impact of single command-based invoking of FPGA accelerators 81
5.3 Dynamic allocation of accelerators vs. static allocation in previous works . . 82
5.4 The throughput multiple accelerators being invoked by multiple applications simultaneously 85

ACKNOWLEDGMENTS

First of all, I wish to express my deepest gratitude to my Ph.D. advisor, professor Eli Bozorgzadeh. Her invaluable guidance, encouragement, and support have accompanied me throughout my entire Ph.D. Journey. I would like to especially thank Dr. Kanghee Kim for his kind assistance and support.

I am grateful to my dissertation committee members Professor Alexander V. Veidenbaum and Professor Ian G. Harris, for their time, support, and invaluable advice.

I would like to express my sincere gratitude to my internship advisor at NGD Systems, Inc., Dr. Vladimir Alves. Dr. Alves, I have always been inspired by your personality, knowledge, and kindness. Also, a special thanks to all the team members at NGD Systems, Inc.

During my Ph.D. life, at the both University of California, Irvine, and NGD Systems, Inc., I was very fortunate to have the support of dearest colleagues and friends, Reza Asadi, Mahdi Torabzadehkashi, Ahmad Razavi, Saeed Mirzamohammadi, Ali HeydariGorji, Sajjad Taheri, Hsin-Yu Ting, Tootiya Giyahchi, Hossein Bobarshad, Shahir Khatami, Roozbeh Naderi, and Maryam Hassani.

I thank the Institute of Electrical and Electronics Engineers (IEEE) and the Association for Computing Machinery (ACM) for giving me permissions to include my previously published papers in this dissertation.

VITA

Siavash Rezaei

EDUCATION

Doctor of Philosophy in Computer Science **2020**
University of California, Irvine *Irvine, California*

Master of Science in Computer Engineering **2012**
Sharif University of Technology *Tehran, Iran*

Bachelor of Science in Computer Engineering **2010**
Shahid Beheshti University *Tehran, Iran*

RESEARCH EXPERIENCE

Graduate Research Assistant **2014–2020**
University of California, Irvine *Irvine, California*

Researcher and Engineer **2017–2020**
NGD Systems, Inc. *Irvine, California*

Researcher **2010–2013**
Sharif University of Technology *Tehran, Iran*

TEACHING EXPERIENCE

Teaching Assistant **2015–2017**
University of California, Irvine *Irvine, California*
Principle in System Design, Data Structures, Artificial Intelligence, Boolean Algebra

Teaching Assistant **2011–2012**
Sharif University of Technology *Tehran, Iran*
Logic lab

Teaching Assistant **2008–2010**
Shahid Beheshti University *Tehran, Iran*
Computer Architecture, Digital Electronics

REFEREED JOURNAL PUBLICATIONS

Cost-effective, Energy-efficient, and Scalable Storage Computing for Large-scale AI Applications **2020**
J. Do, V. C. Ferreira, H. Bobarshad, M. Torabzadehkashi, S. Rezaei, A. Heydarigorji, D. Souza, B. F. Goldstein, L. Santiago, M. S. Kim, P. M. V. Lima, F. M. G. França, and V. Alves, ACM Transaction on Storage (ToS) 16 (4), 1-37

Computational storage: an efficient and scalable platform for big data and HPC applications **2019**
M. Torabzadehkashi, S. Rezaei, A. HeydariGorji, H. Bobarshad, V. Alves, and N. Bagherzadeh, Journal of Big Data 6(1), 100.

REFEREED CONFERENCE PUBLICATIONS

HyperTune: Dynamic Hyperparameter Tuning For Efficient Distribution of DNN Training Over Heterogeneous Systems **2020**
A. Heydarigorji, S. Rezaei, M. Torabzadehkashi, H. Bobarshad, V. Alves, and Pai Chou, International Conference On Computer-Aided Design (ICCAD)

Stannis: Low-Power Acceleration of DNN Training Using Computational Storage Devices **2020**
A. HeydariGorji, M. Torabzadehkashi, S. Rezaei, H. Bobarshad, V. Alves, and P. H. Chou, ACM/IEEE Design Automation Conference (DAC)

UltraShare: FPGA-based Dynamic Accelerator Sharing and Allocation **2019**
S. Rezaei, E. Bozorgzadeh, and K. Kim, International Conference on ReConFigurable Computing and FPGAs (ReConFig)

Catalina: In-Storage Processing Acceleration for Scalable Big Data Analytics **2019**
Torabzadehkashi, Mahdi, Siavash Rezaei, Ali Heydarigorji, Hosein Bobarshad, Vladimir Alves, and Nader Bagherzadeh, Euromicro International Conference on Parallel, Distributed, and Network-Based Processing (PDP)

Accelerating HPC Applications Using Computational Storage Devices **2019**
Torabzadehkashi, Mahdi, Ali Heydarigorji, Siavash Rezaei, Hosein Bobarshad, Vladimir Alves, and Nader Bagherzadeh, International Conference on High-Performance Computing and Communications (HPCC)

Scalable Multi-Queue Data Transfer Scheme for FPGA-based Multi-Accelerators. **2018**
Siavash Rezaei, Kanghee Kim, and Eli Bozorgzadeh, International Conference on Computer Design (ICCD)

CompStor: An In-storage Computation Platform for Scalable Distributed Processing **2018**

M. Torabzadehkashi, S. Rezaei, V. Alves, and N. Bagherzadeh, IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)

Data-rate-aware FPGA-based acceleration framework for streaming applications **2016**

Rezaei, Siavash, Cesar-Alejandro Hernandez-Calderon, Saeed Mirzamohammadi, Eli Bozorgzadeh, Alexander Veidenbaum, Alex Nicolau, and Michael J. Prather, IEEE International Conference on ReConFigurable Computing and FPGAs (ReConFig)

SOFTWARE

UltraShare `https://github.com/siavashr/UltraShare.git/`

A hardware/software framework for ultimate sharing of FPGA accelerators among various applications

ABSTRACT OF THE DISSERTATION

Field Programmable Gate Array (FPGA) Accelerator Sharing

by

Siavash Rezaei

Doctor of Philosophy in Computer Science

University of California, Irvine, 2020

Professor Eli Bozorgzadeh, Chair

The high demand for addressing the required processing power of today's big-data and compute-intensive applications, and the cost of powerful processing units have led end-devices to move most of their data processing to clouds, edges, and datacenters. In clouds, edges, and datacenters, hardware accelerators are exploited to drastically augment the processing power of multi-core processors. Among hardware accelerators, Field Programmable Gate Arrays (FPGAs) have attracted significant attention due to their prominent performance and power efficiency. There are two main challenges in exploiting FPGAs as hardware platforms for acceleration: how to design efficient accelerators to gain high performance?, and how to efficiently enable software applications to access FPGA accelerators? In this dissertation, my focus is on the second challenge, where I show the importance of FPGA-based acceleration management on the total gained performance.

Unlike Graphics Processing Units (GPUs), FPGAs offer a heterogeneous environment for different types of accelerators. This unique feature allows FPGAs to serve various applications concurrently and makes them suitable to address the demands from diverse applications in clouds, edges, and datacenters. However, due to the limited number of resources and time-consuming process of reconfiguration, the importance of sharing FPGA accelerators among different applications arises. Current state-of-the-art accelerator management schemes do not

yet provide accelerator sharing among multiple applications concurrently. To achieve this goal, we need system software support as well as a hardware controller to enable seamless accelerator sharing. In this dissertation, I propose a scalable framework, called UltraShare Express, for the ultimate concurrent sharing of different accelerators among various applications. UltraShare Express provides software like function calls for the seamless FPGA acceleration virtualization. Unlike previous works that exploit a static accelerator allocation at the design-time, UltraShare Express offers a run-time dynamic accelerator allocation that enables maximum utilization of the available resources and accelerators on FPGAs.

In Chapter 1, I discuss the essential reasons behind huge attention toward FPGA acceleration in clouds, edges, and datacenters. I also address the challenges of using FPGAs as hardware acceleration platforms for general applications. In Chapter 2, I present the background and history of using FPGAs as acceleration platforms. I also discuss the advantages and disadvantages of previous works in deploying FPGAs for the acceleration. In Chapter 3, I focus on the system software to address the conflicts that happen among multiple applications requesting FPGA accelerators simultaneously. In this chapter, I present our proposed single-command-based framework, called MQMAI, that uses a multi-queue architecture in the software-stack to minimize conflicts among different applications to access different FPGA accelerators. In Chapter 4, I focus on the hardware accelerator controller, and specifically address the interleaved/concurrent sharing of multiple accelerators among multiple user applications. In this chapter, I propose our hardware controller, called UltraShare, that enables the ultimate sharing of FPGA accelerators among multiple applications. The proposed hardware controller introduces a dynamic accelerator sharing scheme through an accelerator grouping mechanism. In Chapter 5, I propose UltraShare Express, our full-fledged framework for ultimate sharing of several FPGA accelerators among multiple applications. UltraShare Express inherits the advantages of both MQMAI and UltraShare by combining them. In Chapter 5, I further investigate the opportunities for addressing and eliminating FPGA accelerators' stall times in different scenarios. Focusing on the software stack, I propose a novel

multi-queue architecture in the software-stack to avoid possible command blocking scenarios that can significantly degrade the performance of FPGA acceleration when accelerators are shared. I also propose a mechanism to fairly distribute data-link bandwidth among different accelerators concerning the accelerator grouping mechanism proposed in Chapter 4. This mechanism prevents accelerators with higher throughput be sacrificed in accessing data-link bandwidth.

Our Experimental results on various accelerator IP cores show significant improvements when the ultimate sharing of FPGA accelerators is enabled. I believe that UltraShare Express provides an important step toward the efficient and easy deploying of FPGAs in heterogeneous architectures. UltraShare Express is an open-source framework available on GitHub and can be used by other research groups.

Chapter 1

Introduction

Based on a report by the International Data Corporation (IDC), it is expected that the number of Internet of Things (IoT) devices be more than 41.6 billion in the next five years and they generate about 79.4 zettabytes of data [2]. Thanks to recent advancement in communication technology, there has been a huge improvement in data communication around the world [50, 77]. With the emergence of high-frequency data transfer technologies like 5G, the orchestration and coordination of end devices, data processing devices, and storage systems all around the world has become viable in IoT [6]. IoT provides innovative solutions for different issues and problems by putting together smart and intelligent devices and sensors. IoT technology has led to the emergence and growth of the cloud computing [55]. The cloud computing plays a role of collaboration in IoT and is used to store IoT data and process data when it is required. The cloud computing is the current trend in the next-generation application architecture [10].

The high demand for data processing and its required energy and cost lead to a vast growth of the cloud and edge computing to move most of the data processing to the shared powerful resources [38]. Although moving data to clouds is a performance and energy costly process,

there have been many works to accommodate this cost through the gaining performance from powerful processing units [29, 22]. Another proposed solution to address the costly data movement process is the *in-storage processing* technique that moves the processing of data inside datacenters and storage devices instead of moving data to user desktops for being processed [40, 27]. In the last few years, the concept of *edge computing* has also been introduced to avoid sending a huge amount of data to the far servers in clouds. Edge computing is defined as a distributed computing topology in which information processing is located close to where things and people produce or consume that information [32]. However, all these solutions rely on powerful processing units with the capability of serving different users. Regardless of the chosen technique, there is a common demand for powerful processing units in clouds, edges, and datacenters to address the seeking performance for processing data.

With reaching close to the end of the Dennard scaling and multi-core processors era [73, 47, 81], researchers have been looking for new techniques to address this demanding processing power. Among all the solutions, deploying heterogeneous architectures has shown promising performance improvements [53, 30]. A heterogeneous architecture is composed of multi-core processors accompanied by hardware accelerators for executing computational intensive kernels of different types of applications. These hardware accelerators are mainly custom-designed circuits for specific operations that gain much faster execution times comparing to multi-core processors [68]. Figure 1.1 depicts some IoT devices and their demands to servers and datacenters in the cloud, to benefit from their powerful processing powers that are augmented through deploying hardware accelerators.

Currently, the most attractive platforms for hardware acceleration in the cloud and edge computing are in-storage processing, Application-Specific Integrated Circuits (ASICs), Graphics Processing Units (GPUs), and Field Programmable Gate Arrays (FPGAs). Researchers in academia and industries have targeted a wide range of applications to accelerate using these

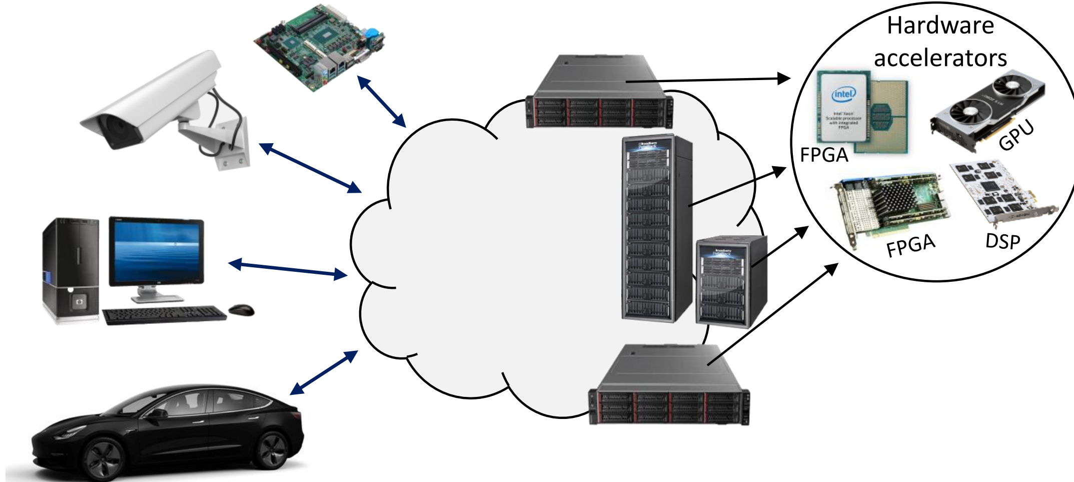


Figure 1.1: FPGAs being used for acceleration in cloud

hardware accelerators. For example, Google and IBM have been exploring their own for Artificial Intelligence (AI) and deep learning application with ASIC designs [33, 36]. Facebook uses ASIC accelerators for their AI inference, AI training, and video transcoding [57]. NVIDIA has introduced a huge range of optimized accelerators including data science and analytics, AI, media and entertainment, medical imaging, safety and security, and many more for GPUs to be used by its users [70]. Researchers at the University of Illinois at Urbana-champaign are working on the acceleration of molecular modeling applications on GPUs [69]. On FPGAs, algorithms including both training and inference phases of deep Learning [60, 88, 108, 97], image processing [90] and vision algorithms [94], Hadoop MapReduce [67], streaming applications [79], data analytics, financial analytics [54], and many other applications have been investigated.

Table 1.1 compares different aspects of hardware acceleration platforms. An ASIC is an integrated circuit chip customized for a special application, rather than intended for general-purpose use. To design and fabricate an ASICs design, designers often use a hardware description language (HDL), such as Verilog or VHDL, to describe the functionality of ASICs. ASIC designs are very efficient for the acceleration of specific applications. ASIC designs are *very* power and performance efficient, however, they are fully customized for specific

Table 1.1: Comparing different aspects of hardware accelerators [9].

	CPU	GPU	FPGA	ASIC
Overview	Sequential processor for general-purpose applications	Originally designed for graphics; now used in a wide range of computationally intensive applications	Flexible collection of logic elements and IP blocks that can be configured and changed in the field	Custom integrated circuit optimized for the end application
Processing	Single- and multi-core processor, plus specialized blocks: FPU, etc.	Thousands of identical processor cores	Configured for application	Application-specific: may include third-party IP cores
Programming	huge range of high-level languages (e.g., C, C++, Python, Java, Fortran); assembly language	OpenCL & Nvidia's CUDA API allow general-purpose programming (e.g., C, C++, Python, Java, Fortran)	Traditionally HDL (Verilog, VHDL); newer systems include C/C++ via OpenCL & SDAccel	---
Strengths	Versatility, multitasking, ease of programming	Massive processing power for target applications: video processing, image analysis, signal processing; Suitable for data parallel tasks	Configurable for specific application; configuration can be changed after installation; high performance per watt; accommodates massively parallel operation;	Custom-designed for application with optimum combination of performance and power consumption
Weaknesses	OS capability adds high overhead; optimized for sequential processing with limited parallelism	High power consumption, not suited to some algorithms; problems must be reformulated to take advantage of parallelism; Sequential execution	Difficult to program; second-longest development time; poor performance for sequential operations; not good for floating-point operations	Longest development time; high cost; cannot be changed without redesigning the silicon

operations and cannot be used for different purposes and applications.

A GPU is a processor that is specially designed for executing intensive rendering of 3D graphics. While CPUs consist of a few numbers (four to eight) of cores, GPUs are composed of hundreds of much smaller cores. GPUs have a Single Instruction Multiple Data (SIMD) architecture and provide massive parallelism to gain high compute performances [14]. GPUs are suitable for data parallel tasks, otherwise they have to run sequentially and very inefficient. GPUs are *very* programmable, however, *very* power-hungry. Due to their SIMD architectures, GPUs cannot support various accelerators and be shared among multiple tenants simultaneously [31]. However, as opposed to ASICs, FPGAs are flexible and re-configurable, and as opposed to GPUs, they are *very* power efficient, while through fine-grain parallelism they can gain considerable performance improvement. FPGAs are relatively in an order of magnitude more energy-efficient and they have higher performance per watt compared conventional processors, CPUs, and GPUs. Given the limitations of GPUs, in the last two

decades, FPGAs have attracted huge attention in heterogeneous architectures, to accelerate compute-intensive applications in both industry and academia [76, 101, 98, 37, 96, 64, 80]. In clouds and datacenters, the combination of diversity of applications and the importance of energy consumption has raised attention toward FPGAs for being deployed for acceleration. This dissertation focuses on FPGA based accelerators, and integration and sharing them among user applications.

In the last few years, the novel concept of in-storage processing has been introduced. In-storage processing allows data to be processed in the storage systems instead of being transferred to the processing engines and then being processed. "In-storage processing pushes the *bring the process to data* paradigm to its ultimate boundaries by utilizing processing engines inside the storage units to process data" [99]. This technique can considerably improve and accelerate the processing time of different applications (especially big-data applications) by removing the data transfer cost and delay [34]. As a joint research project, I collaborated with NGD Systems, Inc. on deploying in-storage processing for accelerating big-data applications. In our research, we focused on deploying embedded processors in storage systems for processing applications in place. Through our research, we developed a full-fledged Computational Storage Device (CSD) and showed a considerable improvement in the processing time of applications using our proposed platform [27, 39, 40, 100, 101, 103]. We also investigated the possibility of using FPGAs alongside embedded processors in storage systems to further accelerate big-data applications. We evaluated our work on a similarity search library, called Faiss [28], and showed a considerable performance improvement [102]. Despite the advantages of in-storage processing, it requires a new set of storage devices that support in-storage processing, and cannot be added to currently available computers and servers in clouds and datacenters as an extension.

1.1 FPGA-based acceleration

In 1984, Xilinx introduced the first Field Programmable Gate Arrays (FPGAs) [104]. FPGA is a reconfigurable integrated circuit that can carry out one or more logical operations. These operations can be as simple as a logic gate (an AND or OR function), or one or more complex functions as comprehensive as a multi-core processor. FPGAs are reconfigurable which means users can change the functionality of them at any time by loading their binary files called bitfile in the context of FPGA. To program an FPGA, a design needs to be coded in the Register-Transfer Level (RTL). RTL is a design abstraction which models a digital circuit through the flow of digital signals between registers and logical operations. There are three steps from an RTL design to generate a bitfile that are performed by vendors' design tools like the Xilinx Vivado design suite. These three steps are synthesizing, implementing, and writing configuration bitfile [52].

FPGAs have attracted huge attention for being explored in the cloud and edge computing and datacenters, due to their unique features. For example, IBM has deployed FPGAs for processing of its large volume and fast-growing NoSQL data stores [15]. Intel also has provided its quick path interconnect (Intel QPI) based Xeon+FPGA platforms for datacenters [89]. Amazon has equipped its Elastic Compute Cloud (EC2) with Xilinx UltraScale+FPGAs to allow its users to gain higher performance by accelerating their applications on FPGAs [8]. Such instances allow users to leverage their application design using Xilinx SDAccel and run them on the available FPGAs [26]. Besides, FPGAs are also being explored in in-storage processing systems by Samsung [49] and NGD Systems [102]. Predictably, in near future, many datacenters will deploy FPGAs for the acceleration [17].

Run-time reconfiguration allows FPGAs to accelerate various compute-intensive applications [95, 119]. Despite the attractive aforementioned features of FPGAs, some challenges make them difficult to be exploited easily. Designing an efficient FPGA accelerator is one of the

challenges in deploying FPGAs for the sake of acceleration. An efficient design of FPGA accelerators requires a strong knowledge of RTL design through hardware programming languages like VHDL and Verilog. Commonly, application developers are software programmers with none or very little knowledge of hardware programming. Thus, conventionally, it is very challenging to explore FPGA accelerators and it needs a team of software and hardware programmers. However, the advancement of high-level synthesis (HLS) tools, like Xilinx SDx HLS [110] and LegUp [16], has been eased the development of FPGA accelerators.

Relying on HLS tools to automatically and based on patterns and templates generate RTL designs would not necessarily lead to an optimum design for gaining performance from FPGAs. C/C++ codes are typically sequential, while performance improvement from FPGAs is achieved through parallelism. Thus, identifying the potential parts to be paralleled is very crucial but time-consuming. Altoyan [7] has shown that the HLS implementation of a design in term of performance-per-power can merely reach 36.4% of HDL implementation of that design. HLS tools require users to provide guidelines, as pragmas, for the compiler to generate effective RTL designs. With considering all the combinations of different pragmas with their multiple configuration possibilities and different parts of the code that each can be used, a very huge design space is ahead of users. This makes it very challenging and time-consuming for users to achieve an optimum and efficient design. A very common procedure in most of the proposed techniques is profiling the high-level description of the computing kernel and adding appropriate HLS optimization annotations to the code.

The other very important challenge of using FPGA accelerators is the integration of accelerators to user applications running on host CPU cores. For host application to invoke an accelerator on FPGAs, an *integration framework* is needed that essentially performs three basic tasks: 1) handshaking, 2) mapping requests to the relevant accelerators, and 3) handling data movements between hosts and the FPGA. The first requirement needs an infrastructure to transfer information and control data between hosts and FPGAs to establish the

connection. The second requirement can be as simple as having only one single accelerator on the FPGA bounded to a single application in the host and forwarding all requests to this single accelerator. In such a simple scenario, for calling accelerators, a set of *blocking functions* would be developed to avoid contention on accessing the accelerator. In the third requirement, handling data movements, the main goal is providing enough data transfer throughput to keep accelerators feeding as much as they can process the data. In the ideal scenario, the data transfer (for both *to* and *from* FPGAs) time is hidden by overlapping the data process and data transfer.

There are some frameworks and platforms that have been proposed to ease invoking FPGA accelerators from host computers and servers. A few numbers of these frameworks are mainly developed to get the whole high-level description of a design as the input and generate the whole design package including the software part -known as host application-, FPGA accelerators, and interfacing part to provide the communication between a host application and FPGA accelerators. Industrial platforms like Xilinx SDAccel and Intel SDK are providing such an environment [4, 1]. Through automating all the design/implementation process, this approach takes the burden of hardware design away from the designers; however, it has three main disadvantages: 1) the entire FPGA is assigned to a single application to accelerate some of its compute-intensive kernels, 2) accelerators on the FPGA cannot be used by other host applications, and 3) for each design, a timing (performance) cost of implementation of design and configuration of the FPGA has to be paid. In this approach, dedicating the entire FPGA resources to a single application leads to a significant under-utilization, and considering the fact that FPGAs are expensive devices it results in a significant inefficiency.

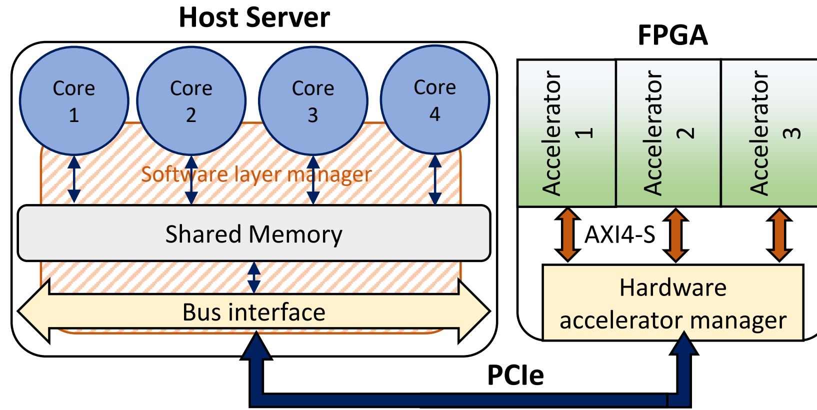


Figure 1.2: Multiple FPGA accelerators being used for acceleration in clouds and datacenters where multi-core processors trying to invoke accelerators

1.2 Sharing FPGA accelerators and challenges

Unlike GPUs and ASICs, FPGAs can host various accelerators at the same time. This opens up huge opportunities for benefiting cloud, edge, and datacenters. In complex systems like cloud, edge, and datacenters, many applications and threads are being executed simultaneously. Each application has many demands and requests for different types of compute-intensive operations that can take advantage of FPGA accelerators. As an example, a convolutional neural network algorithm needs many vector operations in each layer that are usually running on different processing cores simultaneously in a pipelined mechanism. These layers can share the same accelerators on the FPGA and issue requests to them. An efficient accelerator manager system needs to support multiple simultaneous access to accelerators in a congestion-free environment. However, in such systems, to enable concurrent access to accelerators and sharing them among various applications a sophisticated accelerator integration and management platform is required.

Figure 1.2 shows a typical block diagram of a host server, in the cloud, edge, or datacenters, with a multi-core processor and an FPGA that is connected to the host server through PCIe (Peripheral Component Interconnect Express). A comprehensive framework that manages applications to accelerator integration is composed of two main layers for handling the three

requirements of an integration framework: a software layer management and a hardware layer accelerator manager. In such frameworks (shown as *Hardware accelerator manager* and *Software accelerator manager* in Figure 1.2), these two layers are responsible for managing accelerator requests all the way from user applications on host servers to FPGA accelerators. The software layer manager and the hardware accelerator controller access the host's main memory to exchange data through programming a Direct Memory Access (DMA) and communicate (to exchange control data and information) to each other through the device registers using programmed I/O (PIO) mechanism.

1.3 Overview and Contributions of this dissertation

Although there are some frameworks and platforms for managing acceleration requests from host applications to FPGA accelerators, to the best of our knowledge, none of the available allow accelerator sharing in a non-blocking environment. OpenCL-based platforms like Xilinx SDAccel, SDSoC, and Intel SDK allow accelerator sharing, however, due to the specific programming model of OpenCL, these frameworks fail to support concurrent sharing of accelerators. In these platforms, a single application can request for an accelerator, however, after establishing the connection, any other request to the same accelerator from other applications will be rejected until the first application releases the accelerator. This full dedication mechanism leads to a huge under-utilization of an accelerator [109]. Multi-tenancy is a new approach that has been attracted researchers' attention in the last few years [43, 65, 12]. In this approach, the main focus is on sharing FPGAs resources among multiple applications simultaneously. Thus, by partitioning an FPGA area, they allow various concurrent applications to keep using FPGAs. Although these works overcome the issue of under-utilization of FPGA resources at some levels, they still fail to share the same accelerators on FPGAs among multiple applications/tenants. Thus, another level of under-utilization of resources

is still failed to address.

The main theme of this dissertation is studying and introducing *application to accelerator integration* platforms that enable sharing multiple FPGA accelerators among various user applications regarding the demands in clouds and datacenters. Through investigation of the available frameworks and platforms, we recognized their shortcomings and proposed our novel framework for sharing FPGA accelerators to gain the maximum performance. When we discuss sharing many questions need to be answered. Following are part of these questions that this dissertation aims to answer.

- How to efficiently manage acceleration requests from multiple applications to multiple accelerators?
- Is this possible to share accelerators among various applications concurrently?
- How to avoid conflicts among applications trying to access FPGA accelerators?
- Can we interleave execution of acceleration among applications?
- Can the data movement be time shared on the bus?
- Does it needed to prioritize data bus usage between accelerators?
- Should we prioritize acceleration requests?

To answer these questions and maximizing the performance efficiency of FPGA-based acceleration in cloud, edge, and datacenters, we proposed our framework called UltraShare Express. UltraShare Express is an open-source full-fledged framework that enables the full sharing of multiple FPGA accelerators among various applications. To develop UltraShare Express, in Chapter 3, we proposed a congestion-free, non-blocking mechanism for various user applications to access FPGA accelerators. For this purpose, we introduced our single-command based framework, called MQMAI, and our multi-queue mechanism in the software

stack to address the conflict among simultaneous threads/applications requesting FPGA accelerators. To allow fully sharing of FPGA accelerators among various applications, in Chapter 4 we introduce UltraShare. UltraShare uses an accelerator grouping mechanism at the hardware level to minimize accelerators' stall times and enables the full sharing of FPGA accelerators. Finally, in Chapter 5, we introduce UltraShare Express that benefits from the advantages of MQMAI and UltraShare by combining them. UltraShare also proposes a data link aware mechanism for a fair distribution of data link bandwidth among different accelerators. Through our experiments, we recognized that in the existence of various accelerator requests from each application, a blocking situation happens that drastically degrades the performance of accelerators with higher bandwidth. UltraShare Express also proposes a new multi-queue architecture to overcome this shortcoming.

In this dissertation, we specifically focus on streaming applications due to the lack of an efficient integration framework for this huge important category of applications. Streaming applications, such as image/video processing applications, real-time vision algorithms, and network packet encryption algorithms, are in the category of FPGA-friendly data-intensive applications [79, 48]. However, the multi-level memory hierarchy model that is used in OpenCL-based platforms fails to meet the high-demanding data throughput of streaming applications. Ruan et al. [84] have shown that a point to point data transfer from main memory to FPGA BRAMs can significantly improve the performance of the streaming accelerators comparing to an OpenCL hierarchy memory model. Although the current version of UltraShare Express only supports streaming applications, our proposed techniques and architectures are not limited to streaming applications and can be expanded to all types of accelerators. Following we summarize the major contributions of this dissertation:

- **Identifying the shortcomings of the available platforms for enabling and deploying FPGA accelerators:** In Chapter 2, through a comprehensive investigation of the available platforms and the previous research for FPGA acceleration, we rec-

ognized the shortcomings of the platforms and frameworks that are used to integrate FPGA accelerators to user applications. These shortcomings have significant influences on the gained performance.

- **Developing a full-fledged framework for sharing FPGA-based accelerators among multiple applications:** In this dissertation, we developed a fully functional framework, called UltraShare Express, from software stack to hardware controller for FPGA-based acceleration of streaming applications. UltraShare Express is an open-source framework that is compilable and synthesizable for different platforms.

- **Proposing and developing a congestion-free non-blocking software stack that resolves conflicts among multiple applications requesting for FPGA accelerators:** In Chapter 3, We proposed a novel single-command multi-queue based software stack for various applications to share multiple accelerators.

- **Proposing an accelerator grouping technique in hardware controller to enable a fully shared environment for FPGA accelerators:** In Chapter 4, focusing on the hardware controller of our accelerator manager framework, we propose an accelerator grouping mechanism that enables a fully shared non-blocking environment.

- **Proposing a data link aware mechanism to distribute the data link bandwidth among different accelerators and avoid data starving:** In Chapter 5, we show the challenge of data link bandwidth distribution among different types of accelerators with different processing times that leads to a data starving for accelerators with higher throughput. To address this issue, we proposed a mechanism to fairly distribute the data link bandwidth among different types of accelerators.

- *Eliminating accelerator stall times due to the fetch-conflicts that happen in the hardware/software communication:* In Chapter 5, through our experiments, we recognized a command fetch conflict that happens when every single application submits various

accelerator requests. We addressed this issue through our proposed software stack architecture.

Chapter 2

Background and Related work

2.1 Overview of FPGAs Architecture

A Field Programmable Gate Array (FPGA) is a semiconductor device built from the basic components of configurable logic blocks (CLBs). As Figure 2.1 shows, CLBs are connected to each other through interconnects and programmable switches. FPGAs are configurable through programming their CLBs and interconnects for carrying out different operations and functionalities. Unlike GPUs and ASICs, FPGAs can host different types of operations, while they are completely independent of each other, but, they can work simultaneously. FPGAs are very power efficient comparing to typical general-purpose processors and GPUs.

Comparing to ASICs, earlier FPGAs have been slower, less energy efficient, and for the same operation, FPGAs need a larger area. Later FPGAs from major industries like Xilinx and Altera (acquired by Intel Inc. in 2015) have come closer to ASICs by providing a significant reduction in power consumption, increasing their speed, and lowering materials cost [56, 85]. However, in comparison to ASICs, FPGAs are reconfigurable and it makes them suitable for prototyping since using them, debugging is possible and relatively easy,

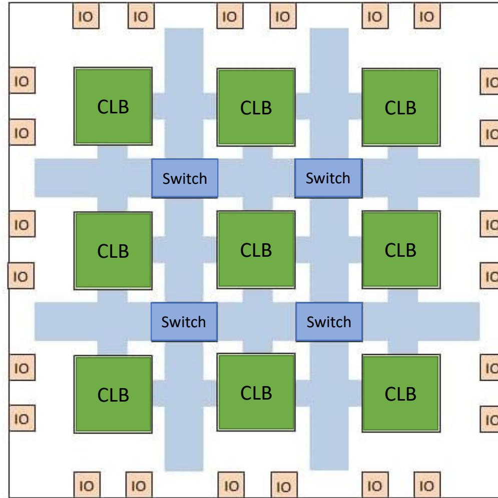


Figure 2.1: Internal architecture of a typical FPGA.

and often their time to market is short. Comparing to CPUs and GPUs, FPGAs work with lower frequencies. Thus, to gain performance improvement using FPGAs accelerator designs have to be very careful. Basically, through fine-grain parallelism, FPGAs have the potential to gain considerable performance improvement. Gaining performance efficiency from FPGAs seeks high skill designers.

In the currently available systems and platforms, there are two main platforms for interfacing FPGAs and multi-core processors: 1) using FPGA boards through PCIe, and 2) using System on a Chip (SoC) FPGAs. Despite the benefits of SoC FPGAs regarding their packaging, power, and direct access to the shared memory, most of the currently available computers and servers in cloud and datacenters do not support SoC FPGAs. Thus, still, the interface between FPGA accelerators and multi-core processors are mostly realized through PCIe-based I/O interface [5]. PCIe is a data link standard for connecting high-speed components. Every desktop computer has several PCIe slots on its motherboard that can be used to add different peripherals like GPUs, RAID cards, Wi-Fi cards, FPGA boards, or SSD (solid-state drive) cards. PCIe slots are available in different physical configurations with different numbers of lanes: x1, x4, x8, x16, x32. For example, a PCIe x4 has four physical lanes and can move data at four bits per cycle. A PCIe lane represents two paths for each direction;

Table 2.1: Bandwidth of different PCIe generations.

PCI 1.0	1 GB/s
PCI 2.0	2 GB/s
PCI 3.0	4 GB/s
PCI 4.0	8 GB/s
PCI 5.0	16 GB/s

thus, both directions are independent of each other and can happen at the same time. There are different generations of PCIe standard that the bandwidth of each generation is about doubled of its previous one [72]. Table 2.1 shows the bandwidth of different PCIe generations. Considering the high-bandwidth of PCIe, it is still a very common interface in the cloud and datacenters. In this dissertation, although the proposed frameworks are based on the PCIe interface, we are not limited to PCIe and our techniques can be used for other FPGA platforms.

2.2 FPGA Software Tools

While FPGAs are hardware platforms that originally need low-level hardware-friendly tools to be programmed, there have been tremendous efforts on making FPGAs programming closer to higher levels of abstraction. This section provides an overview of FPGAs programming and the advancements in this direction.

2.2.1 Software design tools for FPGAs

Register-Transfer Level (RTL) is a design abstraction that models a digital circuit. RTL designs for FPGAs are achieved through hardware description languages (HDLs) like Verilog and VHDL. The concepts in HDL programming is very different from software programming. While in software programming, everything is a sequence of logic and mathematical

operations, HDL programming has a nature of parallelism. This different nature brings significant difficulties for common software programmers, and they need a deep understanding of the hardware designs to be able to map their knowledge to HDL programming. HDL programming is the most hardware friendly language to design accelerators.

Moving towards a software-like environment to design FPGA accelerators, the High-Level Synthesis (HLS) process has been introduced. HLS that also is known as behavioral synthesis and algorithmic synthesis, is a design process in which a high-level description of a design is automatically compiled and transformed into an RTL design that meets certain user specified design constraints. HLS tools that perform the HLS process and generate RTL designs from a high-level description, e.g. C/C++, have been commercially available for over fifteen years [23]. Although HLS tools have taken away the complication of HDL programming, there are still many challenges in using these tools. HLS tools do not necessarily generate the most optimum design. In fact, due to the huge available design space, it is very complicated to achieve the best design from a high-level description of an algorithm [44]. On the other hand, software-level programs are mainly non-hardware aware, and using different memory constructs and data types that are not suitable for hardware can lead to a huge performance and resource utilization inefficiency [44].

Several works in the literature propose automated techniques and optimizations for using HLS tools to fasten their design space explorations [19, 91, 92, 20]. For example, Chi et. al focused on stencil kernels to explore the computation reuse patterns [18]. They propose an optimal algorithm for design space exploration of computation reuse for stencil accelerators with reduction operations. In this work, to shrink down the huge design exploration time, a heuristic beam search algorithm is proposed. There are many other works focusing on augmenting frameworks to help HLS tools being more efficient and effective [66, 59, 63].

There have been some efforts on optimizing high-level synthesis tools for nested loops to find the best combinations of pragmas for gaining the maximum performance [21, 74, 116, 58, 79].

We have proposed a framework to identify parts of the code with the potential of being paralleled or pipelined by applying loop transformation techniques on nested loops [79]. Guan et. al [35] proposed a framework for Deep Neural Networks (DNNs) that takes TensorFlow-described DNNs and generates the FPGA implementation. This work provides specific optimizations for DNN. The work in [54] and [75] generate optimized FPGA designs-based on the parallel patterns such as map and reduce. These works follow their predefined templates and the statistical performance model to achieve higher performances. Choi et. al [19] focus on loop optimization and propose an HLS-based FPGA optimization and design space exploration framework that produces a high-performance design even in the presence of variable loop bounds. Aladdin [87] and Lin-analyzer [118] accelerate the process of HLS through rapid design space exploration without a need to perform RTL implementation. COMBA [117] performs a comprehensive design space exploration on HLS optimization pragmas. By pruning the design space, they find the best configuration with the consideration of the aim of optimization. The aforementioned works are only a few examples comparing to the vast numbers of research that have been accomplished to assist HLS tools for generating RTL designs with closer characteristics to the optimum designs.

2.2.2 Interface Schemes for FPGA-based Accelerators

In cloud and datacenters, FPGA accelerators are mainly invoked by software applications are executed on the host computers and servers. A very important challenge to use FPGA acceleration is the integration of accelerators to the software applications. Several industrial and academic works have been introduced that investigate FPGA-based accelerator invocation and data transfer between a host and FPGAs. These works mainly target two different platforms of SoC FPGAs [62, 113, 24] and FPGAs on PCIe board [79, 13, 106, 83]. Mantovani et al. [62] present a hardware-software solution for memory mapping for on-chip FPGAs. This solution gains performance improvement by preventing an extra data

copy from user space to the special physical memory address allocated to the accelerator. However, for the PCIe-based platforms, this technique is not attractive since the extra data copy does naturally not happen using a pure software memory map [46]. SDSoC is a fully automated Xilinx commercial tool for accelerator exploration for SoC FPGAs. SDSoC environment is an Eclipse-based platform for implementing heterogeneous embedded systems on Zynq boards [113]. Intel has also proposed an open software platform, called OPAE, for accelerator integration with multi-core [24]. OPAE is a unified software framework for FPGA accelerators.

Using industrial platforms to implement and exploit PCIe-based FPGA accelerators is a very common approach among researchers and designers [93, 115]. OpenCL-based platforms from major industries in the field like Xilinx SDAccel and Intel SDK are basically designed to ease accelerating applications on FPGAs. Following the OpenCL programming model for heterogeneous architectures, these platforms transform user-specified accelerating kernels to RTL designs accompanying the necessary components -including the OpenCL memory hierarchy- to allow host software applications to invoke accelerators. Although these industrial platforms from Xilinx and Intel provide a user-friendly environment for FPGA acceleration, their fundamental concept of context considers the entire FPGA as the smallest schedulable unit. Thus, the entire FPGA can only be used by a single application that results in a severe under-utilization of FPGAs [111]. From the other perspective, the memory hierarchy exploited by OpenCL-based platforms is also very inefficient for streaming applications due to their multi-hub data movement that cannot be overlapped [84].

To overcome the conventional use of FPGAs for single applications, a multi-tenancy concept has been introduced for space-sharing of an FPGA among multiple accelerators that each is bound to a specific user/application [43, 65]. Multes [43] is a multi-tenancy framework built on top of Caribou [42], an in-storage processing platform for database engines. While Multes focuses on fairly sharing the bandwidth among different tenants, it is not considering

the accelerators/resources being shared/used simultaneously by different tenants. Mbongue et al. [65] have proposed a Network on Chip (NoC) architecture for FPGA multi-tenancy in the cloud to assign multiple FPGA regions to users. However, the same as Multes, this work does not support accelerator sharing among multiple users.

BlastFunction [12] is a recent multi-tenancy system for sharing FPGAs as micro-services in the cloud. BlastFunction allows multiple applications to execute kernels on the same FPGA concurrently. However, BlastFunction does not support sharing the same accelerators among different applications. Chen et al. [17] proposed an accelerator invocation framework that only allows different threads in a single application to share FPGA accelerator and does not support sharing among multiple applications. The proposed technique introduces the concept of FPGA-as-a-Service (FaaS) to share FPGA accelerators, however, FaaS comes with some shortcomings: the accelerator invocation is blocking; using a single shared memory region for data transfer prevents the overlapping of data movement and computation; On the hardware side, large buffers need to be allocated while BRAM resources are limited on FPGAs, while for most of the designs BRAMs are the bottleneck resources that prevent a higher number of accelerators; Last but not least, introducing a software layer accelerator manager keeps the processing core of the host in interaction while accelerators are processing requests.

Centaur [71] introduces an accelerator sharing environment for the specific MonetDB database application on the Intel's Xeon+FPGA platform. Centaur benefits from the shared memory spaces between processing cores and FPGA to enable accelerator invocation. Centaur uses a polling mechanism to fetch different jobs to the FPGA. Using the concept of *job*, Centaur is the closest framework to ours that can manage accelerator allocation dynamically. However, Centaur suffers from some shortcomings. It only allows sharing among multiple threads in a single application and fails to support sharing accelerators among multiple applications. Also, being restricted to the shared memory for data movement limits Centaur to perform concurrent data movements for multiple accelerators. Centaur is designed only for integrated

CPU+FPGA chips and cannot easily be extended to support PCIe; while due to the currently available platforms in the cloud and datacenters, adding FPGAs through PCIe boards is more common. Optimus [61] proposes a hypervisor for shared memory on the Intel HARP shared-memory FPGA platform. Exploiting Memory Mapped I/O (MMIO) to invoke an accelerator and keeping processing core in the interaction with the accelerators, Optimus does not allow concurrent requests being issued for an FPGA accelerator. These frameworks either do not support or fail to provide a seamless interface to multiple accelerators accessed simultaneously by various applications. On the other hand, deploying FPGA boards through PCIe is common due to the portability, extendability, and possibility of being added easily to the current systems.

Some other open-source frameworks have been developed that allow user applications to transfer data between host computers and FPGA accelerators [46, 105, 25]. These frameworks need software designers to manage hardware accelerators that require an understanding of hardware developments. fflink [25] and JetStream [105] provide a data transfer datapath between host and FPGA relies on Xilinx commercial IP cores. Using Xilinx commercial IP cores has made these frameworks non-synthesizable on the current versions of Xilinx synthesizing tools due to many changes and upgrades to the IP cores. Among these frameworks, Riffa [82] is synthesizable with the latest available tools like Xilinx Vivado design suite. Riffa [46] introduces multiple channels on the FPGA. Each channel has a pair of input/output buffers. Each accelerator can be connected to those buffers for the data transfer. To exploit an accelerator, the user application calls functions send_fpga() and receive_fpga() in order on a specific channel number that is passed as an argument to these functions. Specifying the destination accelerator by user application significantly degrades the performance of FPGA accelerators when multiple applications share the accelerators. fflink, RIFFA, and JetStream, all suffer from supporting simultaneous multi-core access to the accelerators. Due to managing each FPGA accelerator through a specific region of the PIO range, these frameworks also have a limitation on the maximum number of accelerators implemented on FPGAs.

In this chapter, we provided an overview of FPGAs architecture and their use for acceleration purposes. We investigated the most related previous works and research in FPGA acceleration, integration of applications to accelerators, and sharing FPGA accelerators. To summarize this chapter, the currently available frameworks for the integration of user applications and FPGA accelerators fail to provide a full shared environment for accessing FPGA accelerators. A full-shared environment allows various applications to access various accelerators without being locked by each other or other external limitations. The lack of such a framework motivated us to propose our novel mechanisms to enable sharing FPGA accelerators and develop our full-fledged framework UltraShare Express.

Chapter 3

Scalable Multi-queue Scheme for Various Applications Seeking FPGA-based Accelerators

3.1 Introduction

FPGAs have shown a viable solution toward addressing the demanding performance and power efficiency in the era of big-data applications. However, despite the promising improvements that all the research and works have achieved for specific applications using FPGA accelerators, still gaining the maximum performance from FPGAs in cloud and datacenters is very challenging. In near future, FPGAs will be common processing platforms in datacenters and cloud computing [41, 11]. In cloud and datacenters, many applications with different computational kernels need to use FPGA accelerators to achieve their demanding processing powers. To address this demand an interfacing framework is needed to integrate software applications to FPGA accelerators and manage FPGA requests. The management

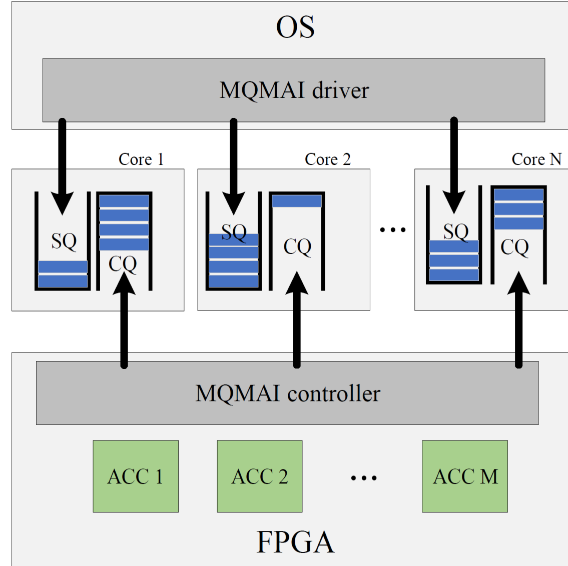


Figure 3.1: The multi-queue structure of MQMAI

of FPGA requests is further challenged by increasing the number of parallel threads requesting access to accelerators and exponential growth in the volume of data to send/receive to/from FPGA-accelerators [107]. As a result, deployment of FPGA-based accelerators in large software systems has been challenged by the lack of a scalable scheme from application software to the I/O interface to allow multi-core to access FPGA-based accelerators at a high data transfer rate.

As opposed to GPUs, multiple FPGA accelerators can be accessed from multi-core (See Figure 3.1) that can be time-shared among multiple threads. Such multiple accelerators are attractive to applications since it avoids the long wait time for threads to access accelerators. However, management of multiple requests to access multiple accelerators is more challenging and complex both at the FPGA side (accelerator controller) and at the CPU side (I/O software stack). Hence, we need an interface scheme that is scalable with increasing the number of host cores, accelerators, and the volume of I/O data.

Current FPGA-based design flows like Xilinx SDAccel and Intel Open Programmable Acceleration Engine (OPAE) [24] do not support sharing accelerators among multiple appli-

cations/threads. Sharing accelerators as proposed in previous works [105, 25, 46] brings resource contention issues in I/O software stack, i.e., a thread waiting for an accelerator when in use by another thread. To tackle this drawback, we present MQMAI, a multi-queue command-based mechanism, to reduce the contention over the access to multiple FPGA accelerators. Our proposed technique enhances I/O parallelism for access to multiple accelerators. To the best of our knowledge, this is the first effort to introduce an efficient multi-queue interface to allow multiple threads/applications access and share multiple FPGA-based accelerators [80].

The multi-queue data transfer mechanism in MQMAI is a non-blocking command-based that enables efficient and scalable access to multiple accelerators on FPGA via PCIe. MQMAI is assembled of 1) a software stack on the host side and 2) a hardware controller on the FPGA side. On the software side, a device driver manages the multi-queue software architecture and submits the commands when an access request is issued. It updates the queues when the host is notified about the completion of a command from FPGA. The role of the hardware controller starts by receiving a command. The controller is responsible for data movement and scheduling accelerator accesses without any interaction with the host CPU, hence, reducing CPU time spent on accelerator management and data transfer.

Compared to the state-of-the-art data transfer frameworks such as RIFFA [46], our proposed method reduces the communication traffic between CPU and FPGA for data movement as well as resource contention. This leads to a significant improvement in total latency (total access and process time) of parallel threads accessing single or multiple accelerators on FPGA. When compared to RIFFA, our experimental results show that, as the number of parallel threads accessing per accelerator increases, the total latency increases more scalable using the MQMAI framework. For example, for 4 parallel threads accessing FPGA accelerators in a system with 4 FPGA-based accelerators, the total latency is 18x lower compared to RIFFA. We summarize MQMAI contributions as follows:

- Developing a multi-queue command-based framework that enables a reduced conflict accelerator-sharing environment.
- Providing a C library with a user-friendly API to access FPGA-based accelerators.
- Developing an accelerator controller at the hardware level to allocate accelerators to multiple requests from parallel threads/applications.
- Providing MQMAI open-source framework from software stack to RTL design of accelerator controller.

3.2 Multi-Queue Multi-Accelerator Interface (MQMAI)

In accelerator-rich architectures, multiple threads can simultaneously request access to accelerators. Sharing accelerators among threads brings resource contention issues in the I/O software stack. Resource contention (i.e., a thread waiting for a resource while being used by another thread) is a drawback in multi-threaded programs. Managing resource contention is a key challenge when maximizing resource utilization. When a resource access request is issued at the application level, it must go through the system software stack to get access to the resource. Rejecting a request because of a contention results in performance degradation as the request needs to re-pass through the software stack. A resource contention also degrades the performance drastically because either the processing core would be locked, or the operating system will schedule another application to the core and the application needs to be rescheduled.

Introducing multi-queues to multi-core processors is a promising solution to manage resource contention in I/O intensive applications. In this solution, each core has its own dedicated queue to get access to an I/O resource. Thus, each core can issue a request without facing a contention from a software stack point of view. Deploying multi-queue architecture to issue

read/write requests to SSD drives such as NVMe has drastically improved the memory access challenge through the serial port of PCIe [13, 3]. NVMe uses multiple queues to offer higher levels of I/O parallelism. Once the CPU writes a command to the queue, the CPU finishes the I/O request and can proceed to another request. Otherwise, the CPU is responsible to pass the request through several functions before the command is finally issued to the device. Thus, the NVMe driver reduces CPU time to issue an I/O request. NVMe is a promising standard to enable scalable access to SSDs with high performance and throughput.

Comparing to NVMe, Multiple accelerators shared and accessed by multiple parallel threads of applications, are faced with similar challenges. Each accelerator on the FPGA can be viewed as a memory block in SSD for parallel access. The large I/O data size of accelerators for big-data applications and the increasing number of threads accessing multiple accelerators resemble I/O intensive applications from the CPU I/O software side. Adopting NVMe like standards and software stack for multi-queue access to multi-accelerators in FPGA is a scalable framework to enable more parallelism and high performance; which would be a promising solution toward resolving the lack of multi-core accelerator access in the previous data transfer frameworks.

In this chapter, we introduce MQMAI, a multi-queue command-based interface for the FPGA accelerator. Similar to NVMe, MQMAI bypasses the block layer and instead deploys a multi-queue mechanism. MQMAI allows different host applications to directly invoke and share FPGA accelerators through the PCIe bus (however the protocol is not limited to PCIe bus) with no contention. This interface defines a set of commands and registers to be used between the MQMAI driver on the host operating system and the MQMAI controller on the FPGA. It features multiple I/O queues shared between the driver and the controller, as shown in Figure 3.1, which is scalable in terms of the number of queues and the number of queue entries.

3.2.1 MQMAI framework

MQMAI is assembled of 1) a software stack and 2) a hardware controller. The software stack includes a device driver that manages requests from library calls, commands, submission/completion queues, and MSI interrupt handler from the MQMAI controller. The hardware controller role starts by receiving a signal (called doorbell) from software that shows a command is available. Then all the data management process is done in the hardware with no interaction with the host CPU. After writing back the result of the accelerator request to the host memory, the controller makes the host CPU aware of the command completion through an MSI interrupt.

Software stack on host CPU

The MQMAI software stack consists of two layers: a library layer and a driver layer. The library provides user-friendly APIs shown in section IV.D. The library layer also compresses and standardizes the arguments from API calls and passes them to the device driver. MQMAI driver is capable of handling simultaneous multi-thread (-core) access exploiting multiple command (submission/completion) queues. We explain the main components of MQMAI driver as follow:

Submission/Completion command: A command is a set of information related to each request/response. A submission command is generated by the device driver to specify a request to the hardware controller. The device driver pushes the submission command into the submission queue of the corresponding core. Table 3.1 shows different fields of a submission command.

Each submission command has 256 bits including a command ID and the ID of the core that is executing the corresponding thread. The input/output command type specifies if

Table 3.1: An UltraShare command structure

Name	Number of bits	Explanation
Cmd_id	10	Each command has a specific command ID
Core_id	5	The ID of the core that has submitted the command
Acc_type	5	The requested accelerator
In_sgl_addr	64	The SGL address for input data
In_sgl_len	10	The length of the input SGL
Out_sgl_addr	64	The SGL address for output data
Out_sgl_len	10	The length of the output SGL
Reserved	88	Reserved

the I/O address refers to the main data (physical region pages (PRP)) or the scatter-gather (SG) list of data. When the size of a data is larger than one memory page, the operating system stores each page in a different location. Thus, to read/write the data into the right location, hardware needs the address of all the pages using the SG list of all the addresses. Accelerator ID specifies the accelerator type. It guides the MQMAI controller to allocate the right accelerator type to each request (in the case of having multiple accelerator types). Input/output length specifies the length of data/SG to be transferred.

A completion command is generated by the hardware controller to make the driver aware of a completed request. The hardware controller writes the completion command into the completion queue of the corresponding core through a DMA request. A completion command only contains the command ID and a validation bit.

Submission/Completion queue: In the device driver, one submission queue and one completion queue are allocated to each core. The commands issued from each core are sent to the corresponding submission queue. These queues are shared between the device driver and the hardware controller to exchange commands. This multi-queue mechanism gives the capability to handle parallel I/O requests to access accelerators. The size of queues depends

on the load and the number of parallel requests.

Doorbell: Device driver makes hardware controller aware of the existence of commands by sending a doorbell register to the controller. This register contains the tail address of the corresponding submission queue through programmed I/O (PIO) mode. Each submission queue has its own dedicated doorbell. MQMAI hardware controller always keeps the track of the head addresses of all submission queues. Thus, by receiving the tail address, the hardware controller will be aware of available commands and issues a read request to DMA to fetch the commands. Transferring doorbell registers is the only PIO mode that MQMAI deploys. All the rest of the data transfer is through DMA. It makes the host CPU detached from the data transferring process. Previous open-source interfaces, like fflink [25] and RIFFA [46], keeps the host processor busy by exchanging handshaking information between host and FPGA in the PIO mode during the data transfer.

Interrupt handler: When the hardware controller writes a completion command to the completion queue, it notifies the device driver by issuing a message signaled interrupt (MSI). MSI interrupt specifies the core ID from which the access request was issued. Thus, the device driver starts reading all the completion commands from the corresponding completion queue.

Hardware Architecture

Figure 3.2 shows the block diagram of the MQMAI hardware architecture. The hardware design is constructed of 3 main modules: 1) PCIe interface, 2) MQMAI controller, and 3) Interface channels. For the PCIe interface, we used Xilinx PCIe integrated IP block. This IP block has a master-slave AXI interface that can be configured to send and receive data in both PIO mode and DMA mode [112].

MQMAI controller: It contains 4 main parts: 1) Doorbell registers management unit, 2) Command controller, 3) SG management unit (SGMU), and 4) Completion command man-

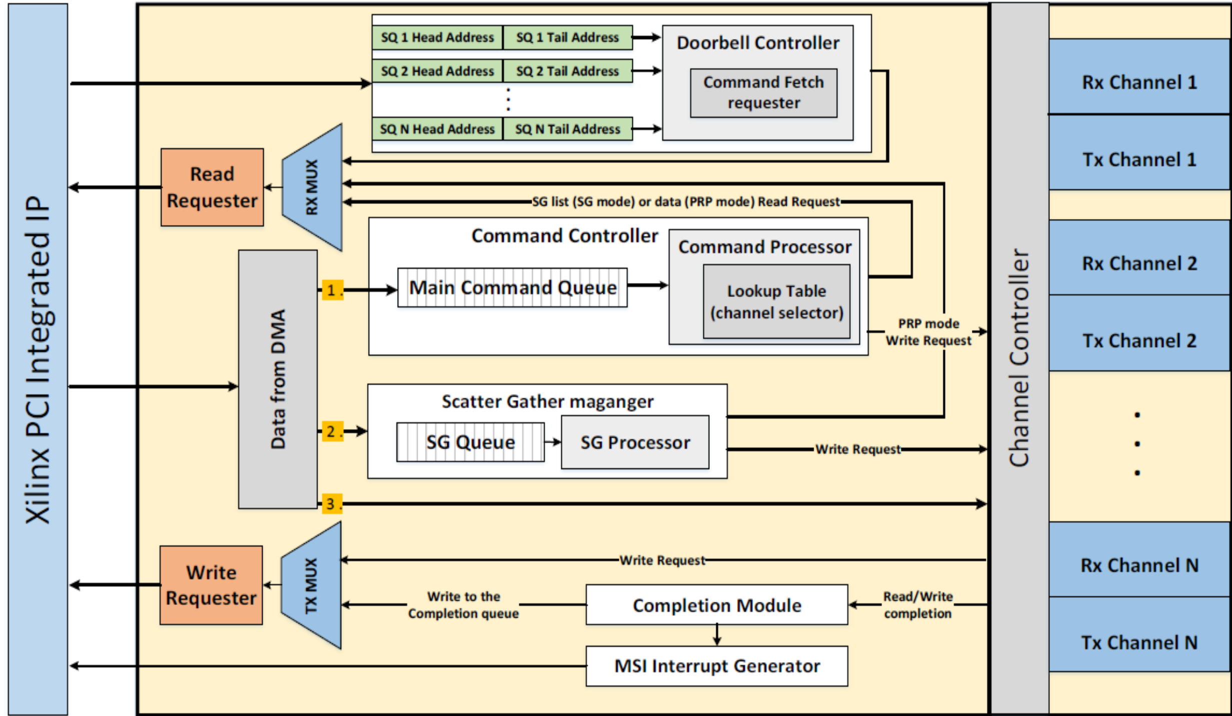


Figure 3.2: Block diagram hardware architecture of MQMAI.

agement. The doorbell registers management unit consists of doorbell registers and a doorbell controller. For each submission queue on the host side, two doorbell registers are keeping the head and tail DMA addresses of the submission queue on the hardware side. The doorbell controller starts providing a DMA request through the RX MUX module whenever the head address is not equal to the tail address for any of the queues. Received commands will be forwarded to the command controller and stored in a command queue to be processed in the command processor unit. The controller must allocate an accelerator to each command. If there are multiple identical accelerators on the FPGA, the scheduler has the flexibility to allocate the accelerators to the commands based on the availability of accelerators or in the order of the criticality of commands. In our current framework, the controller has a round-robin scheduler. This is the key feature that enables the virtualization of destination buffers at the application level and enhances accelerator utilization.

The SGMU unit receives the SG list. For each entity, based on the corresponding SG field

in the command, data addresses, and length from SG list:

- If it is an SG for input data: SGMU sends a read request to the memory through DMA. In this case, a control signal will be passed to the interface channels controller to make it ready for accepting incoming data.
- If it is an SG for output data: SGMU passes the request to the interfacing channels controller.

Interface channels have two main parts: 1) Rx/Tx channels, and 2) channel controller. MQMAI has one channel controller and as many as available accelerators Rx/Tx channels. Rx channel stores the input data for the accelerator and Tx data stores the result of the accelerator. Channel controller manages all reads/writes data from/to Rx/Tx channels. Channel controller can handle simultaneous read and write from/to host memory. Channel controller also receives write requests to the memory from the command controller and SGMU, keeps the request until the corresponding Tx channel has enough data, and then issues a write request to the memory through DMA.

3.2.2 MQMAI Protocol

MQMAI deploys a command based mechanism to submit accelerator requests from a host application to the corresponding accelerator on an FPGA. Being a single-command based enables the possibility of adding many features including fully sharing of accelerators among multiple host applications. To the best of our knowledge, among all the previous works only [71] uses a similar concept called *job*, however, as mentioned in Section 2, it suffers from other shortcomings that cannot support fully sharing of accelerators among multiple applications.

Figure 3.3 shows the basic timing diagram of the accelerator invocation protocol of Ultra-Share Express. As the figure shows, a request from a host application starts being processed

by going through UltraShare's software stack, i.e. UltraShare's library and UltraShare's device driver. UltraShare's library is an interface to translate user-friendly APIs to io controller commands interfacing the device driver. In the device driver, for each request, a command is built that includes all the required information that represents the request to be processed in an FPGA accelerator with no additional interaction with the host processing units. The generated command is pushed into a submission queue (SQ). Each SQ has a doorbell signal that makes UltraShare's hardware controller (implemented on the FPGA) aware of commands being queued in that SQ. From this point, host processing cores have no interaction with the issued request until receiving a completion interrupt and completion command from the FPGA side.

When a doorbell signal is received, the hardware controller fetches the queued commands in the corresponding SQ. Each fetched command is queued in a command queue in the hardware controller waiting for its turn to be processed by the appropriate accelerator. When an accelerator is available to serve a command, the first step is to fetch the associated input and output scatter-gather (SG) lists (SGLs) of the command. Then the input data based on the input SGL is fetched to feed the allocated accelerator. When the process is finished the result is written back to the host's main memory based on the output SGL. Then a completion command is written to the completion queue (QC) in the host main memory and an interrupt is issued to make the host processors aware of the completion. Receiving the interrupt, the host processor reads the completion command and makes the application aware of the completion to read back the result of the accelerator call.

Software Programming Model

To use MQMAI, a user need to follow our programming model in their C/C++ codes. The current version of MQMAI has 4 main APIs: fpga_open(), fpga_close(), fpga_acc(), and fpga_wait(). fpga_open() and fpga_close() are used to open/close the MQMAI device driver

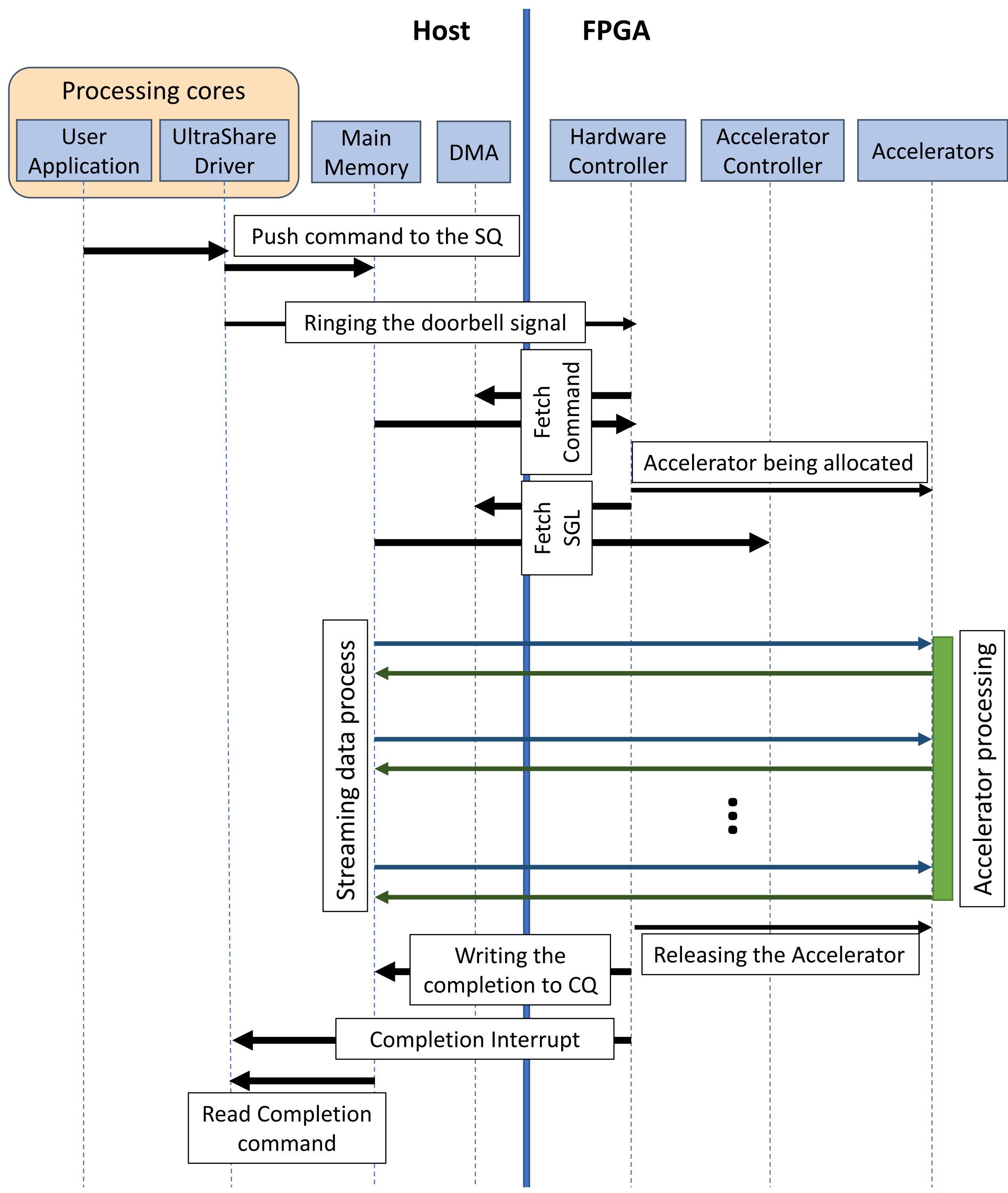


Figure 3.3: Timing diagram of UltraShare Express from a user application requesting an accelerator process until the request being processed

and start/finish the use of MQMAI. fpga_acc() is used to call an accelerator. It is a non-blocking API that allows applications with no data dependency to be able to issue many accelerator requests or execute the rest of their computations. fpga_wait() is used to wait for the issued accelerator requests.

Hardware Interface

To integrate an streaming accelerator to MQMAI, the accelerator needs to use our hardware interface signals. Hardware interface includes 7 different signals:

- **Input signals:** Reset, User_clock, Tx_data, Tx_data_valid, Rx_data_request.
- **Output signals:** Rx_data, Rx_data_valid.

To read data from an Rx channel, an accelerator needs to set the Rx_data_request signal to a 1 and wait for the Rx_data_valid signal to be 1. Valid data arrives in the Rx_data bus while Rx_data_valid signal is 1. To write data to the Tx channel, an accelerator only needs to put the data on the Tx_data bus and keep the Tx_data_valid high while Tx_data carries valid data.

3.3 Evaluation

3.3.1 Experimental Setup

To evaluate MQMAI, we used a host computer with a quad-core Intel processor i5-4590 running at 3.3GHz and 8GB memory, running the Linux operating system. We developed a user-level library for applications and a kernel-level device driver in the Linux kernel to

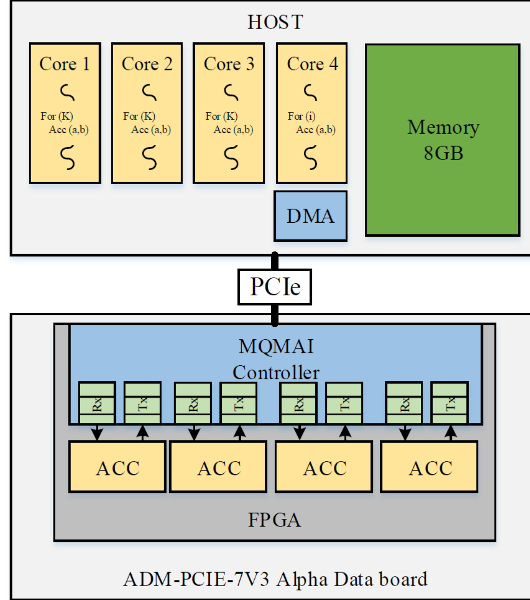


Figure 3.4: Experimental platform.

access our FPGA board through 4 lanes of PCI Express 3.0 bus. The board is an ADM-PCIE-7V3 alpha data board with a Xilinx Virtex 7 FPGA where we program the MQMAI controller and some accelerators. To generate bitfiles of the RTL design, we used Xilinx Vivado design suite tool 16.4. For comparison purposes, we also implemented RIFFA [46], which is a state-of-the-art open-source platform widely used in related works [82].

Figure 3.4 shows our platform. To evaluate the performance scalability of MQMAI, we varied the number of cores (C), the number of threads per core (T), and the number of accelerator requests iteratively issued as a batch in the for loop of each thread (K). Figure 3.5 shows the pseudocode of the threads. Different threads may call the same accelerator or different accelerators, respectively, which can help understand the effect of resource contentions.

We program the FPGA board with our MQMAI controller and 4 black box functionally identical accelerators. Each accelerator is assumed to have a pipeline architecture with streaming input/output data. This assumption is for a simple experimental setup and they can be programmed to have different functions. We also assume that the latency caused by each accelerator is 10 ns to process each accelerator request of 1 Mbyte data size from the

```
Void *thread_main () {
        buffer *A;
        buffer *B;
        …
        While () {
        // Start of measuring the delay
        For (i = 0, i < K, i ++)
                // A: input data      B: output data
                Ret = acc_fpga (fpga, acc_type, A, length, B, length);
        wait_fpga (fpga, acc_id, K);
        // End of measuring the delay
        }
}
```

Figure 3.5: Pseudocode of the threads.

threads. Each accelerator reads data from its dedicated Rx channel and writes the result back into its Tx channel (Figure 3.4). Note that our evaluation metric is the end-to-end delay (or latency) measured at the application level for each batch of K accelerator requests. Figure 3.5 shows the two timestamping points in the thread to get the end-to-end delay.

3.3.2 Experimental Results

Performance Scalability

Single-Thread Single-Accelerator Case: Figure 3.6 shows the throughputs achieved by MQMAI and RIFFA for different sizes of accelerator requests from the host to the FPGA board. In this case, we run a single thread that generates requests over the same accelerator. The results show that the throughput increases almost linearly to the size of accelerator request for both MQMAI and RIFFA. They give similar throughputs for each request size,

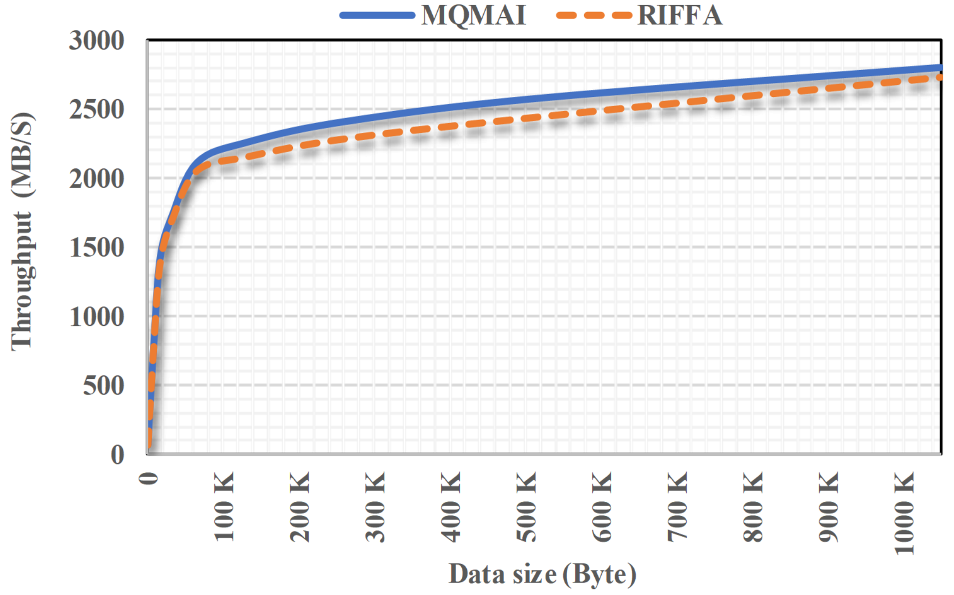


Figure 3.6: Throughput for different request sizes in Single-Thread Single-Accelerator case.

but MQMAI is always slightly better than RIFFA. For the request size of 1 Mbyte (MB), MQMAI achieves 2803 MB/s while RIFFA does 2727 MB/s. This improvement mostly comes from the less amount of interactions between CPU and FPGA for exchanging information to set up the data transfer and notify the completion. For the rest of the experiments, we set the request size to 1 MB.

MTSA (Multi-Thread Single-Accelerator) Case: To evaluate the performance scalability obtained from our "command-based interface" of MQMAI, we vary the number of threads (T) on a single core from 1 to 4 and the number of accelerator requests issued as a batch from each thread (K) from 1 to 20. All the threads are assumed to access the same accelerator. Table 3.2 shows the average end-to-end delay measured at the application level for both MQMAI and RIFFA.

From the table, we can see that MQMAI always gives a shorter delay than RIFFA for all combinations of T and K we test. When T = 4 and K = 20, the delay obtained with MQMAI is just 0.118 seconds, which is 18.5 times faster than that of RIFFA, i.e., 2.25 seconds. The reason for this improvement is that in MQMAI any request, possibly generated from

Table 3.2: Average end-to-end delay in Multi-Thread Single-Accelerator case.

Framework	# of threads (T)	End-to-end delay (second)				
		# of requests in a batch (K)				
		1	2	5	10	20
RIFFA	1	0.00211	0.00442	0.00905	0.01899	0.03720
	2	0.00882	0.03869	0.06087	0.10994	0.23968
	4	0.01728	0.02467	0.80174	1.6634	2.25089
MQMAI	1	0.00197	0.00368	0.00871	0.01721	0.03393
	2	0.00302	0.00667	0.01581	0.03035	0.06531
	4	0.00481	0.00996	0.02748	0.05536	0.11783

different threads, is not rejected in the submission queue addressed by the threads while it has available entries. Thus, it can benefit from the effect of pipelining all such outstanding requests between the host and the FPGA. On the contrary, RIFFA rejects any requests submitted to an accelerator that is busy servicing a request, thus causes the rejected request to be reissued by the application and to go through the entire software stack again until it succeeds. Thus, in RIFFA, the delay significantly increases when such contentions occur.

Recall that the numbers in Table 3.2 are the average delay. For 100 samples obtained for the case of $T = 4$ and $K = 20$, the standard deviation is 1.633 for RIFFA and 0.017 for MQMAI. That is, MQMAI gives much better performance even in terms of the worst delay.

MTMA (Multi-Thread Multi-Accelerator) Case: To evaluate the performance scalability obtained from our “multi-queue structure” of MQMAI, we distribute the threads over four cores. Table 3.3 shows the average end-to-end delay obtained with 4 threads and 20 requests in a batch. In this setup, we assume that each thread runs on its dedicated core for both MQMAI and RIFFA. However, in our MQMAI, all the threads have access to all the four accelerators of the same type (or function) because the MQMAI controller can dynamically schedule and dispatch every incoming request to an available accelerator

Table 3.3: Average end-to-end delay in Multi-Thread Multi-Accelerator.

Framework	# threads to each accelerator				Average Delay (s)
	ACC1	ACC2	ACC3	ACC4	
RIFFA	1	1	1	1	0.03722
	2	1	1	0	0.09141
	2	2	0	0	0.24017
	3	1	0	0	1.16931
	4	0	0	0	2.26002
MQMAI	Dynamic scheduling within the controller				0.03485

at the arrival time. This feature of virtualizing multiple accelerators of the same type into one single virtual accelerator unburdens application developers of the responsibility to decide which accelerator to use. On the contrary, RIFFA does not allow such virtualization. In RIFFA, each thread should be manually programmed so that it can be directed to a particular accelerator.

From the table, RIFFA achieves its shortest average delay, i.e., 0.0372 seconds, when each thread is directed to one dedicated accelerator. However, in our MQMAI, such user-defined guidance is not needed because the MQMAI controller dynamically schedules all requests over all the accelerators. In the worst case of directing requests in RIFFA, i.e., all the threads direct their request to the same accelerator, RIFFA imposes a 60x delay comparing MQMAI.

Resource Overhead

Table 3.4 compares the FPGA resource utilization of MQMAI and RIFFA when the request size for the accelerators is 1 MB. Also, resource utilization of JetStream is provided based on the numbers provided in its paper [105]. Compared to RIFFA, MQMAI uses 70% less Logic (LUTs), 81% less LUTRAM, and 69% fewer flipflops. Compared to JetStream, still, MQMAI uses slightly fewer LUTs, while uses slightly more flipflops and LUTRAMs. Consider that

Table 3.4: Resource utilization on a Xilinx Virtex 7 FPGA for 4 I/O channels.

	Total resources	JetStream	RIFFA	MQMAI
LUT	433200	8571 (1.97%)	26433 (6.1%)	7883 (1.82%)
LUTRAM	174200	392 (0.22%)	2464 (1.41%)	456 (0.26%)
FF	866400	6955 (0.8%)	39720 (4.58%)	12196 (1.41%)
BRAM	1470	NA	83 (5.6%)	95 (6.4%)
DSP	3600	0 (0%)	0 (0%)	0 (0%)

as we mentioned before, JetStream is not currently available and the numbers presented to give some ideas. On the other hand, these numbers will change by changing the maximum possible size of data transfer.

MQMAI uses 13% more BRAMs (memory blocks) compared to RIFFA. MQMAI requires more BRAMs for command queue implementation. As a result, with the MQMAI framework, there are more unused FPGA resources to be utilized for more (sophisticated) accelerators.

3.4 Conclusion

In this chapter, we present a multi-queue command-based data interface to access multi-accelerators on FPGAs. The proposed framework is a non-blocking interface with reducing resource contention due to multi-queue I/O software architecture to increase parallel I/O access. The accelerator controller, implemented on FPGA, manages the commands and allocates the accelerators using a round-robin policy. The overhead of the proposed framework both on the software stack and hardware controller is lightweight. Our proposed framework is more scalable and results in total latency improvement by 18x compared to state-of-the-art RIFFA.

Chapter 4

Fpga-based Dynamic Accelerator Sharing

4.1 Introduction

As we discussed in previous chapters, FPGAs have been assisting general processors as custom hardware accelerators with their fine-grain programmable hardware resources [76, 114, 86, 102]. When multiple accelerators on an FPGA are deployed to accelerate various applications, shared hardware resources incur more stringent constraints on high-throughput data movement between the FPGA and the main memory. As we discussed in chapter 3, to integrate FPGA accelerators to host user applications, an infrastructure is required that besides handling data movement, must be able to manage the allocation of FPGA accelerators.

Unlike GPUs, FPGAs provide an environment for various types of accelerators to be implemented simultaneously. This capability adds more value to FPGAs for being exploited in cloud, edge, computer servers, and datacenters for responding to various needs and requests

from various applications. In today's computer servers and datacenters with a vast variety of applications being executed, there are many applications with similar compute-intensive kernels. For example, a vast category of machine learning algorithms, and vision and image processing applications need vectors multiplications [51]. In such systems, sharing accelerators among different applications is a very important technique to avoid under-utilization of resources and gain maximum performance from the limited number of FPGAs' resources.

In this chapter, our focus is on improving the sharing mechanism in the simple hardware controller that is proposed in Chapter 3. Our proposed hardware controller, called UltraShare, provides a dynamic accelerator sharing scheme through an accelerators grouping mechanism. UltraShare allows software applications to fully utilize FPGA accelerators in a non-blocking congestion-free environment. UltraShare also reduces the idle time of the accelerators through deploying an accelerator grouping mechanism which results in a considerable improvement in the performance of FPGA accelerators. We briefly summarize the contributions of this chapter as follow:

- For the first time, we introduce a non-blocking FPGA-based accelerator sharing framework, called UltraShare, by proposing a hardware controller to enable dynamic sharing,
- We propose an accelerator grouping architecture to enable efficient access to accelerators shared among multiple streaming applications,
- We developed UltraShare in Verilog hardware programming language which makes it compatible with all the FPGA vendors and RTL synthesis tools. UltraShare is an open-source framework and can be used and contributed by other research groups,
- We evaluated UltraShare with the standard IP-cores interfacing standard AXI4-Stream protocol.

4.2 UltraShare: Multi-accelerator Framework

One of the most important requirements of a multi-accelerator system is the capability of *sharing* accelerators among different host applications. An efficient accelerator sharing mechanism addresses the following features: *1) Exploits dynamic parallelism:* All the requests from one application are distributed among the available accelerators. *2) Shares accelerators among multiple applications:* Multiple applications can share a single accelerator. *3) provides Non-blocking congestion-free access to accelerators.*

In this chapter, we propose an adaptable scalable hardware controller, called UltraShare, to address the three mentioned features of accelerator sharing along with an efficient data movement between a host and FPGAs. UltraShare is composed of five main parts (Figure 5.3): 1) multi-queue accelerator request, 2) dynamic accelerator allocation, 3) scatter-gather, 4) accelerators controller, and 5) data transfer. The inputs to the hardware controller are streams of commands from the host's main memory. In the following, we explain the different components of each part in detail.

4.2.1 Multiple Command Queues

Accelerator allocation is responsible for assigning commands to accelerators dynamically. In a single-queue non-grouping mechanism, always the command at the head will be processed. If there is no accelerator available for this command, it would block the rest of the commands to be processed. Thus, the single-queue mechanism may result in a severe accelerator under-utilization due to the blocking requests among multiple applications requesting accelerators of different types. To tackle this, UltraShare proposes an accelerator allocation based on an accelerator grouping mechanism. The accelerator allocation includes the following components.

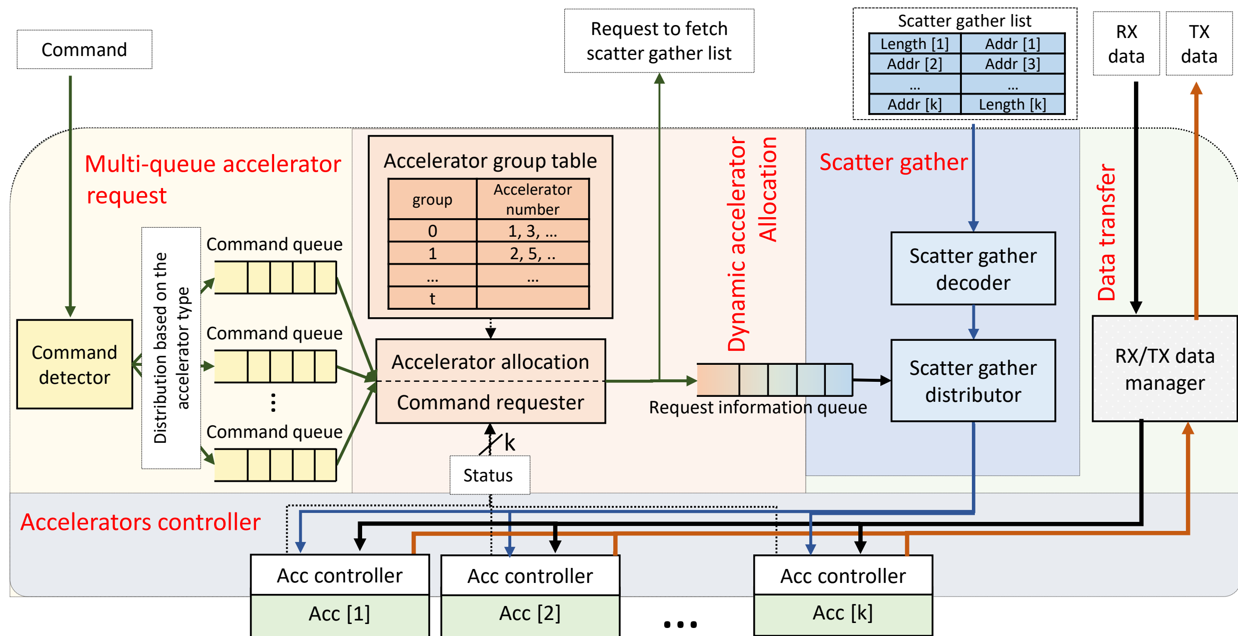


Figure 4.1: UltraShare hardware parts and components.

Command Detector: A command includes all the information required to process the associated request with no interaction with the host. When a command arrives, the command detector pushes the command into one of the command queues based on the requested accelerator type. In this chapter, we only consider a single-level accelerator grouping which is based on the accelerators types, however, the UltraShare framework allows more sophisticated strategies, e.g. a two-level priority-based grouping the first level of which is based on the priority of the accelerators and the second level is based on the accelerators types. In this regard, some of the accelerators can be reserved for high-priority requests.

Command Queues: a command queue is a simple FIFO implemented with BRAMs. For each group of accelerators, there is one dedicated command queue.

4.2.2 Dynamic Accelerator Allocation

Accelerator Allocation Unit (AAU): The main unit of the dynamic accelerator allocation part is AAU. This unit assigns an accelerator to the command which is at the head of a

command queue. AAU travels between the queues in a round-robin scheduling mechanism. If there is no accelerator available for a selected command, the next command queue will be selected. The inputs to AAU are: 1) the status of all accelerators, and 2) the output of *accelerator group table* that represents the mapping of accelerator numbers to the accelerator groups.

Accelerator Group Table: The Accelerator group table is a lookup table that provides the information of matching accelerators to the accelerator groups. This lookup table is reconfigurable through software APIs, and a user can regroup accelerators, remove an accelerator from a group, or add an accelerator to a group.

Command Requester: After allocating an accelerator to a command, *command requester* submits a request to the DMA to fetch input data (RX) and output data (TX) scatter-gather lists.

Request Information Queue: The information related to each processed command will be stored in a queue to be used when the associated scatter-gather lists arrive.

4.2.3 Scatter-Gather (SG)

The SG part is responsible for receiving and decoding SG Lists (SGLs), and distributing them into their associate accelerators. The SG part is composed of the following components.

SG Decoder: An SGL is constructed of a list of addresses of memory pages with their associated data lengths; while usually, the length of the first and last SG is less than a memory page size, the length of the other SGs is equal to the page size. To shorten the size of the SGL, we compact it by skipping the length of middle SGs. When an SGL arrives, *SG decoder* extracts pairs of addresses and lengths from an SGL. We call each pair of address and length an SG element.

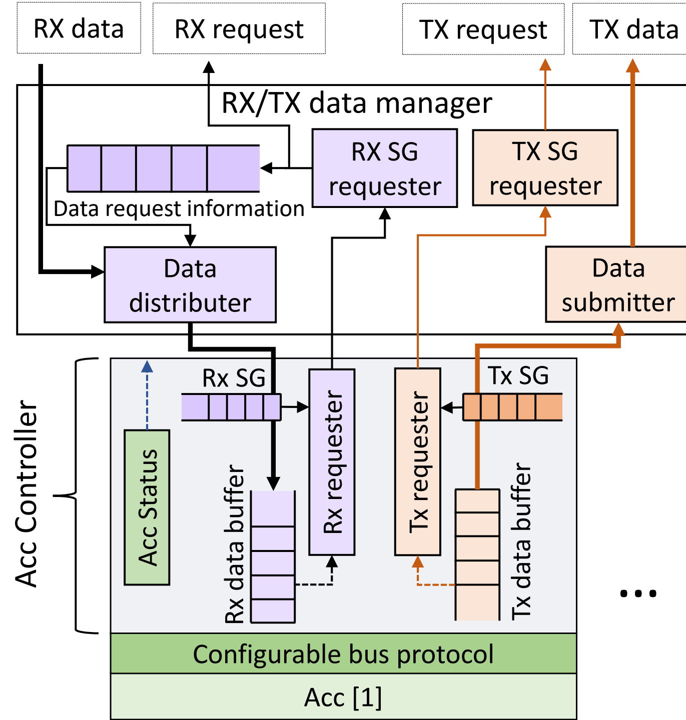


Figure 4.2: Accelerator controller and interleaved RX/TX SG manager

SG Distributor: The inputs to *SG distributor* are SG elements and the information from *request information queue*

4.2.4 Accelerators Controller

Acc Controller: In an accelerator controller, for each input, one RX data buffer, and for each output, one TX data buffer exists. Figure 5.4 shows a detailed block diagram of the accelerator controller and data transfer parts. Conventionally, RX/TX buffers must be large enough to store all the data for one accelerator request. Considering the rate of input/output data from PCIe and the rate of the data process in the accelerator, buffers could be more optimized. However, the optimization requires careful profiling of the accelerator's processing rate and the data rate transfer. On the other hand, still for most of the accelerators with a low processing rate the size of the buffers would be relatively large. Defining large buffers in the BRAMs for the accelerator invocation framework does not leave enough space for

designers to place more accelerators in an FPGA.

To overcome this problem, we define one RX SG queue and one TX SG queue for each input and output, respectively. Thus, each accelerator controller stores the whole list of SG elements. Then, an accelerator issues one RX request if there is enough space available in the RX data buffer, also issues one TX request if there is enough data available in the TX buffer. This mechanism allows defining RX/TX data buffers with much smaller sizes. This size must be at least equal to the size of one memory page of the host which is the maximum length of one SG element. To prevent an accelerator stall due to waiting for RX data or a free TX buffer, we define buffers with the size of a few numbers of memory pages.

4.2.5 Data Transfer

The data transfer part is responsible for providing RX/TX data for accelerators. Together with the accelerator controller part, the RX/TX data would be fetched/submitted from/to the DMA engine for each SG element. It is notable that the path for RX and TX data is completely separated and at the same time an RX and a TX request can be responded to and the data movement be accomplished. In the following different components of the data, transfer part are introduced.

RX/TX SG Requester: These components submit a request related to one SG element to the DMA. The request includes an address and a length.

Data Request Information: For each RX request, a data request information queue stores the information related to the request to be used when the corresponding data arrives. This information allows the *data distributor* component to submit the data to the correct accelerator.

4.3 Experimental Results

4.3.1 Experimental setup

To synthesize and implement UltraShare, we use Xilinx® Vivado® 2018.2 design tool. We exploit a 7v3-alpha-data board which has a Xilinx® Virtex 7 XC7VX690T with a PCIe Gen3 connector. Our host is a PC with an Intel® Core™ i5-4590 CPU @ 3.30GHz.

We have implemented UltraShare in the pure Verilog programming language. Thus, UltraShare is not limited to any specific vendor tool or platform. We have deployed a command-based data interface scheme in our software stack similar to [80], allowing multi-core parallel access to accelerators. UltraShare can be used with other available command-based software platforms like Xilinx SDAccel. It only requires the software stack to submit single commands that follow the UltraShare command structure including an accelerator type field for managing accelerators.

In our applications, we use two APIs, one for calling accelerators and one for waiting for their completions. We measure the throughput of the accelerators by measuring the end-to-end delay of processing requests. This delay is measured in the software application from the point that the accelerator is called until the completion is received.

4.3.2 Benchmarks

To evaluate UltraShare, we exploit two streaming accelerators: an image/video processing accelerator and a network packet encryption algorithm.

Image processing: We exploit RGB-to-YCbCr, a standard streaming image processing IP core from Xilinx® with the standard AXI4-Stream interface. This accelerator can be

configured, at the design time, for different resolutions of images/videos. We define three different types of RGB-to-YCbCr converter, resolution 240×180, resolution 480×320, and resolution 960×640. While for these accelerators the computation algorithm is the same, the input/output sizes and the computation latency over a single input are different.

Network packet encryption: Encryption is a streaming computation-intensive algorithm that can be a good candidate to be accelerated on FPGAs. We use an AES-128 encryption algorithm from MVD® cores over different videos with different resolutions. Unlike the RGB-to-YCbCr accelerator, the same AES accelerator can operate over different input sizes.

4.3.3 Results

Dynamic accelerator allocation

To explore the impact of dynamic versus static accelerator allocation, we compare UltraShare with Riffa [46]. Riffa is the only open-source framework that is available to be used. While ST-Accel, a recently proposed framework, has automated the process of generating and connecting accelerators to the applications, the mechanism of accelerator allocation and data transfer in ST-Accel is very similar to Riffa. Riffa is not capable of handling multiple requests from different applications to a single accelerator. Thus, to compare with Riffa, we use one multi-threaded application and use a semaphore mechanism to manage requests to the same accelerators. The chosen accelerator is an RGB-to-YCbCr 480×360. For Riffa, different scenarios of static accelerator allocations have been experimented with. For example, the worst-case scenario which is all the three threads are requesting only for one of the accelerators, and the best-case scenario that each thread requesting for a different accelerator, and so on. Comparing to the worst case of the static accelerator allocation in Riffa, we observe more than 3x improvement in the throughput. Notably, this is just a simple scenario to show the impact of a dynamic accelerator allocation. In a more complicated

scenario, a static accelerator allocation can drastically degrade the performance.

multi-queue grouping accelerators

To show the impact of multi-queue grouping accelerators on removing accelerators idle times, we implemented three types of accelerators: two from RGB-to-YCbCr converter, for resolutions 240×180 and 480×320, and one AES accelerator that we submitted video frames with the resolutions of 240×180 to it. From each of these accelerator types, we implemented 3 instances. Thus, a total of 9 accelerators are implemented on our FPGA.

Table 4.1 compares the throughput of UltraShare versus a non-grouping single-queue implementation. As we described in section 4.2.1, the multi-queue mechanism that is proposed by UltraShare decreases the idle times of accelerators and allows them to process a request when at least one request is available. In this experiment, we used three different applications, each requesting one of the accelerator types. In a single-queue non-grouping implementation, the slowest accelerators will block other accelerators to be assigned to the available requests. Thus, all the accelerators will be limited to the throughput of the slowest accelerator. It is notable that in our experiment, RGB-to-YCbCr 240×180 accelerator has only a slightly higher throughput. The reason is that for this accelerator, due to smaller input sizes, the associated user application can prepare and submit more requests comparing to the other applications. Thus, more requests from this application will be ended in the shared command-queue.

Resource utilization

To show the scalability of UltraShare, we measured the resource utilization with a various number of accelerators and a various number of accelerator groups. Among all the resources, only the utilization and variation of LUTs and BRAMs are considerable. Increasing the

Table 4.1: Throughput of different accelerators for UltraShare vs. single-queue non-grouping structure

	Throughput (frame/sec)	
Accelerator	Single-queue Non-grouping	UltraShare
RGB-to-YCbCr (video 240×180)	1039	8230
RGB-to-YCbCr (video 480×360)	847	2166
AES decryption (video 480×360)	812	856

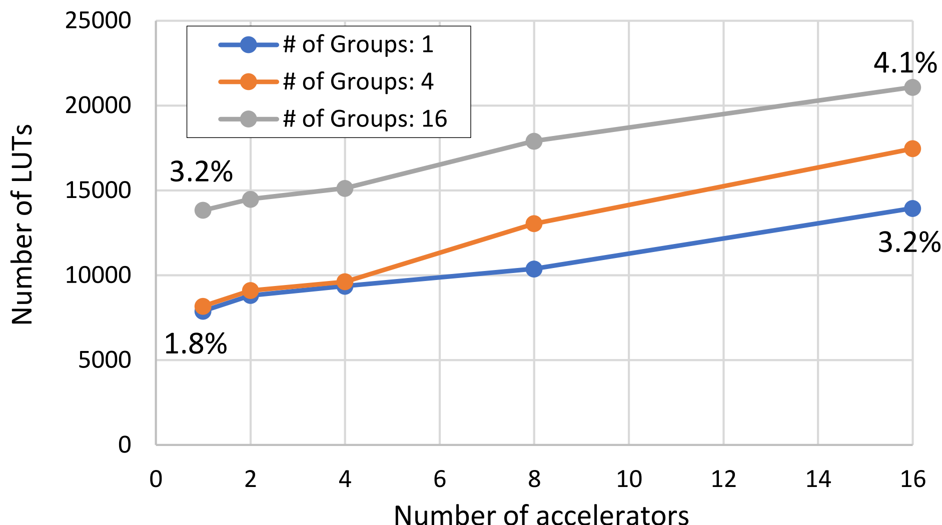


Figure 4.3: UltraShare LUT utilization

number of accelerators and accelerator groups have a linear impact on both the number of LUTs and BRAMs. Figure 4.3 and Figure 4.4 show the number of utilized LUTs and BRAMs, respectively (the given percentages are based on our Xilinx® Virtex 7 FPGA). For one group of accelerators, the number of LUTs increases from 700 to 1400 by increasing the number of accelerators from 1 to 16, and for BRAMs it increases from 55 to 230. While for 16 groups of accelerators these numbers are from 1400 to 2100 for LUTs and from 110 to 390 for BRAMs. It is notable that on our Virtex 7 XC7VX690T FPGA 2100 LUTs are 4.1% of LUTs and 390 BRAMs are 19.1% BRAMs.

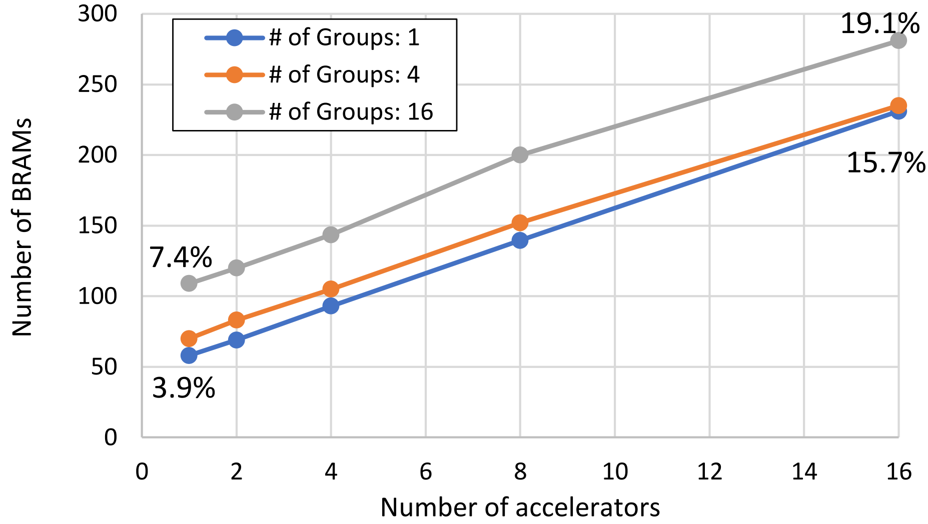


Figure 4.4: UltraShare BRAM utilization

Exploiting dynamic parallelism

To show the capability of supporting parallelism that UltraShare supports, we provided three separate experiments. In each experiment, we implemented three instances of only one type of accelerators. Figure 4.5 shows the end-to-end delay of submitting requests from a single application to the target accelerators. As Figure 4.5 shows, when the number of requests increases, requests will be distributed among the accelerators, and for each factor of three, there is a jump in the end-to-end delay. The reason is that there are no more accelerators available for a new request after all the three accelerators are processing three different requests. It means that the fourth request needs to wait until at least one accelerator is idle. Thus, the delay is increased after passing a factor of three requests which is the number of accelerators.

Accelerator sharing

To show the accelerator sharing feature of UltraShare among different applications, we implemented three instances of AES encryption accelerator on the FPGA. Then we provided three

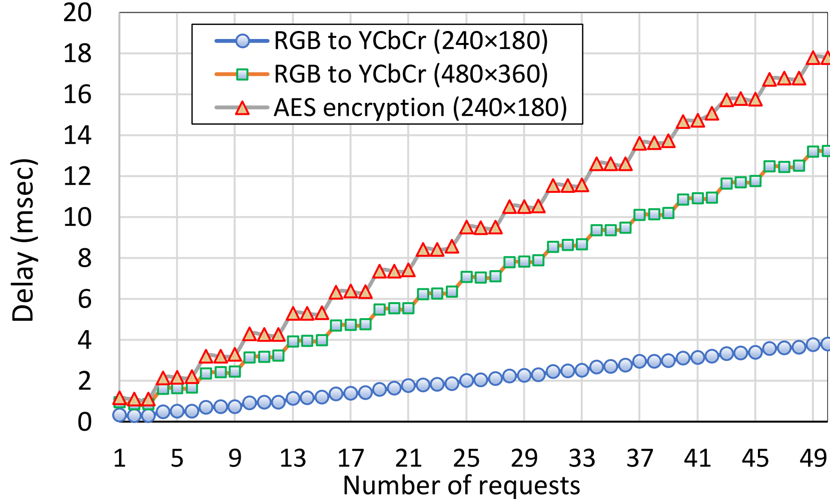


Figure 4.5: Exploiting parallelism in UltraShare

different applications: *application 1* that submits requests for a video with the resolution of 240×180, *application 2* that submits requests for a video with the resolution of 480×360, and *application 3* that submit requests for a video with the resolution of 960×640. Then we considered three different scenarios (Figure 4.6). In *scenario a*, we ran only one of the applications at a time and measured the throughput of the accelerators for each application. In *scenario b*, we ran two different applications simultaneously and we considered all the three possible combinations of two applications from three applications. In *scenario c*, we ran all the applications simultaneously. Considering the throughput of the processed frames for each application in these three scenarios, presented in Figure 4.6, we can see how the accelerators are evenly shared among the applications. Although due to the different input sizes the number of processed frames for the different applications are different (when the input size is larger, it takes longer time to be processed), Figure 4.7 *scenario c* shows that the accelerator usage for all the three applications is equal. In another word, the difference in throughput is due to the different request sizes which require different computation latency.

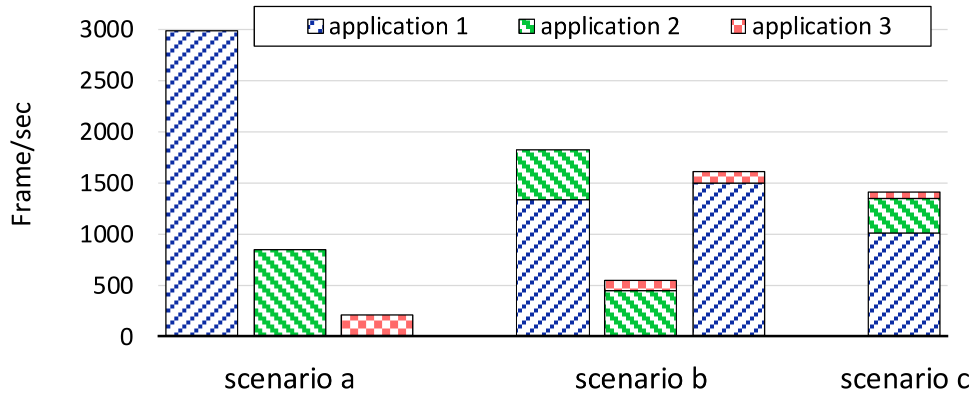


Figure 4.6: AES accelerator sharing among different applications submitting three different video resolutions

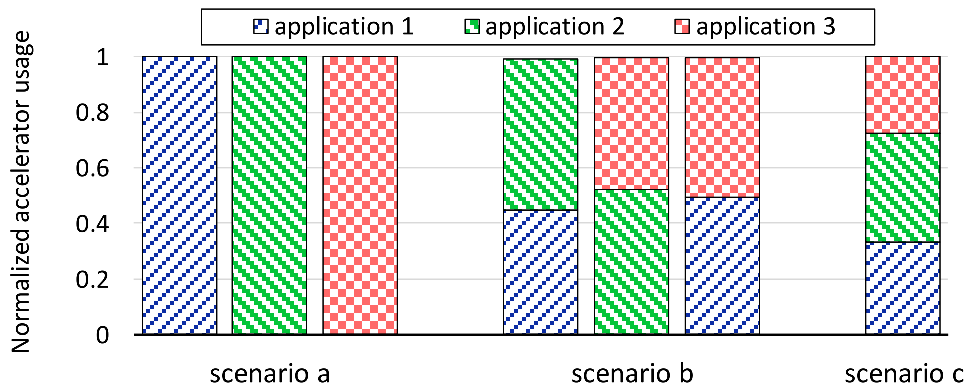


Figure 4.7: Normalized AES accelerators usage by three different applications submitting three different video resolutions

4.4 Conclusions

In this chapter, we proposed UltraShare, an FPGA-based accelerator hardware controller to enable dynamic accelerator sharing among multiple host applications. UltraShare provides a scalable dynamic accelerator allocation scheme to exploit dynamic parallelism for the requests from a single application. Using an *accelerator grouping* scheme, UltraShare removes accelerator idle times to improve the total throughput gained from a multi-accelerator system. UltraShare also deploys a single command-based request mechanism that addresses a non-blocking accelerator sharing environment for different host appellations to share FPGA accelerators. Experimental results show that in a simple scenario with 9 accelerators from 3 different types, UltraShare provides up to 8x throughput improvement for streaming applications comparing to a single-queue non-blocking implementation.

Chapter 5

UltraShare Express: Bandwidth Aware Ultimate Sharing of FPGA-based Accelerators

5.1 Introduction

In cloud, edge, and datacenters, various applications with different computational kernels are being executed. In many scenarios, these applications need to share the same accelerators while they still have requests for different types of accelerators. In previous chapters we discussed the advantages of FPGAs for accelerating cloud, edge, and datacenters applications. However, FPGAs have a limited number of resources. Thus, to be able to respond appropriately to the accelerator demands in cloud, edge, and data centers, accelerators need to be shared among various tenants and user applications. To address this demand, a software/hardware integration framework is needed to manage accelerator requests from user applications to FPGA accelerator [78].

In Chapter 3, we introduced MQMAI [80], our multi-queue software stack, that offers a congestion-free environment for different cores running different threads in a host computer and each thread requests for accelerators on an FPGA. MQMAI deploys the concept of single-command based for submitting each individual accelerator request. We also designed a hardware controller that processes commands sequentially and allocate accelerators to them if there are available accelerators from the same types. To evaluate this design we used mimicking accelerators with the same types that receive input data, add some delay, and forward back the input data to the host.

In Chapter 4, we focused on the hardware controller to provide a fully shared environment for accelerators being invoked from host applications [78]. In this chapter, we proposed an accelerator grouping mechanism that eliminates commands blocking situations. This blocking is caused by other commands that are waiting for other types of accelerators that none of their instances are free. This grouping mechanism is extendable for supporting commands with priorities and commands with a guaranteed processing rate.

In this chapter, by combining our proposed techniques in Chapter 3 and Chapter 4, we introduce UltraShare Express that inheritances the advantages of both frameworks. In Addition, through investigating the multi-queue structure in the software stack, we propose a new architecture for a multi-queue structure that removes additional existing conflicts and blocking among user applications. In this chapter, we also expand our experiments with different real accelerators and propose a scheduling algorithm to resolve data-link bandwidth starving. We show that due to the different processing times of different accelerator types, accelerators with lower processing times can suffer from data bandwidth starving while other accelerators are receiving higher portions of the data-link bandwidth. We propose a data-link bandwidth balancer mechanism to fairly distribute the bandwidth of the data-link among different accelerator types.

5.2 Motivation

Despite the promising performance and power efficiency of FPGAs, it is still very difficult and challenging to efficiently use them for different purposes and applications in datacenters and cloud computing. As described in Chapter 2, there have been attempts to ease using FPGAs. However, many of the available and proposed works fail to support unique features of FPGAs. There are essential expectations that the currently available frameworks do not or fail to fully support for efficient access to FPGA accelerators. Following we list these expectations.

The support for multiple user applications to access the same accelerators: To the best of our knowledge, none of the previous works support multiple user applications accessing and sharing the same accelerators. Due to the limited number of resources on FPGAs and the time-consuming reconfiguration of FPGAs, sharing same accelerators among different applications is very important toward minimizing accelerators' stall times, while they are on demand. Because of their fundamental architecture, in OpenCL based frameworks, accelerator allocation is handled by user applications, and the permission of using accelerators is held until the application releases the permission. During this time no other application is allowed to submit a request to the accelerator. This drastically degrades the efficiency and performance of using FPGA accelerators in a multi-core system that each core executes threads/applications with acceleration demands. The general open-source frameworks like Riffa [46], JetStream [105], and fflink [25] use memory mapped I/O mechanism to manage accelerators from the host device driver. This mechanism ends up with the policy of rejecting a new request for an accelerator that is allocated, and processing a previous request. The same as the OpenCL programming model, ST-Accel [84] requires a series of APIs to be called by a user application for each accelerator invocation. This programming model blocks requests to currently busy accelerators.

Benefit from the maximum acceleration capacity: In deploying any hardware accelerator, the maximum gain is achieved when all available accelerators can service their corresponding requests. Thus, an accelerator manager needs to be able to distribute requests among the available accelerator as much as possible. In the currently available PCIe based frameworks, managing FPGA accelerators including OpenCL based frameworks, regardless of the number of instances of each accelerator, users need to specify which instance of accelerators they want to use. This imposes huge performance degradation to the whole system due to the static allocation of accelerators to the applications.

Support for streaming applications: Despite the drastic increase of using FPGAs for the acceleration of streaming kernels in high-demand applications [84], the OpenCL programming model is not efficient for streaming applications. An accelerator invocation in OpenCL includes four main steps controlled by user applications: accelerator allocation, data transfer from host main memory to the on-board DDR, initiation of the process by the accelerator, and reading back the results from the on-board DDR to the host's main memory. Due to the hierarchical memory architecture of OpenCL, these four steps must be executed sequentially and cannot be overlapped. This makes OpenCL based frameworks very inefficient for streaming applications [84]. Besides the inefficiency of data movement in using the currently available frameworks, exploiting an available streaming accelerator (as an IP core) is challenging. For example, Xilinx SDAccel requires users to have hardware design knowledge since they are restricted to use AXI4 wrappers. Thus, users need to properly handle the controlling signals of AXI4 to AXI4-Stream to allow SDAccel components to equip the possibility of accessing accelerators. While this process is feasible, it is very time consuming, and it needs hardware design knowledge. Open-source frameworks like Riffa mainly focus on the data transferring part and leave the challenges of integrating accelerators and hardware managers to developers.

Dynamic distribution of data-link bandwidth: Various accelerators have different latencies. In

an FPGA, there are ideally different instances of different types of accelerators. However, the bandwidth of PCIe is limited and would be saturated when multiple streaming accelerators are being in use simultaneously. We show that when different numbers of instances of each accelerator type are implemented on an FPGA, some accelerators can face data starving due to the saturation of data-link bus by other accelerators. In this chapter, we show that with our proposed accelerator grouping mechanism, a fair bandwidth distribution is possible while different instances of each accelerator type are available.

5.3 UltraShare Express: Policies and Protocols

Figure 5.1 shows a high-level block diagram of UltraShare Express. UltraShare Express is a hardware/software integration framework. The current version of UltraShare Express only supports FPGA streaming accelerators while it can be extended with the same advantages to support other types of accelerators. Through comprehensive management of commands (acceleration requests) and data transfers, UltraShare Express introduces a fully shared conflict-free environment for user applications to benefit from FPGA accelerators. To the best of our knowledge, UltraShare Express is the first hardware/software integration framework for FPGA accelerators that deploys a single-command-based mechanism to submits accelerator requests from host applications to the corresponding accelerators. Being single-command-based enables the possibility of adding many features including fully sharing of accelerators among host applications that are explored in this chapter. A detailed description of UltraShare Express main protocol is presented in Section 3.2.2. In this section, we present UltraShare Express policies and protocols and introduce the contributions that it offers to the field of FPGA-based acceleration.

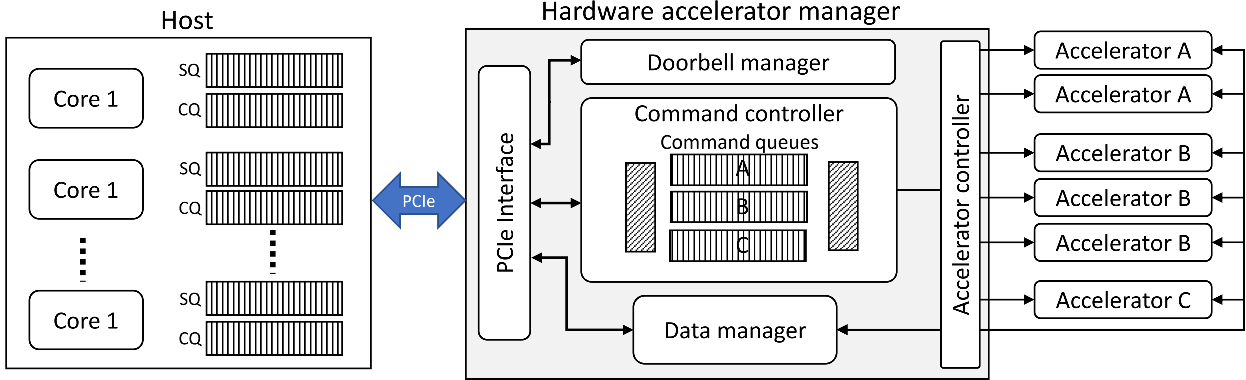


Figure 5.1: High-level block diagram of UltraShare Express

5.3.1 Congestion-free Multi-core access

There are two types of conflicts that happen among multiple cores of a multi-core system requesting external operations. The first conflicts happen in submitting requests and the second conflicts happen in receiving their completions. These conflicts are not limited to FPGAs but, we focus on FPGA accelerators and more specifically on UltraShare Express. In our proposed protocol in Section 3.2.2, conflicts on submitting requests happen when multiple threads try to access the same submission queue and its accessories to submit their accelerator requests. In such a scenario, to avoid data corruption, a locking mechanism is necessitated. This locking mechanism causes performance degradation. To address this issue, UltraShare Express deploys a multi-queue mechanism that dedicates at least one submission queue for each core in a multi-core processor [80]. In this case, threads that are being executed simultaneously on different cores would submit their acceleration requests to different submission queues with no conflict.

Conflict on completions happens when a single core receives interrupts for all the completions, and by running the interrupt routine, it reads the completion command. However, a single-core being responsible for handling interrupts imposes a considerable latency and performance degradation to specific applications being executed on that core [3]. Also, there should be communication mechanisms between cores to provide the required information re-

garding a completion command to the CPU core that has issued the corresponding request. These communication mechanisms also needs shared resources that a locking mechanism must protect them. To address this issue, UltraShare Express deploys Message Signalled Interrupts (MSIs) to interrupt the corresponding processing core for each request being completed. However, yet accessing a single completion queue by multiple processing cores causes performance degradation due to the locking mechanisms required to protect the shared resources. To overcome this issue, UltraShare Express dedicates a single completion queue for each processing core.

5.3.2 Accelerator Sharing

UltraShare proposes a harmonic mechanism to allow accelerator sharing among multiple applications executed simultaneously. The very basic concept that equips UltraShare to enable accelerators sharing is the single-command-based concept for submitting requests. UltraShare's single-command-based mechanism allows requests to be processed independently of the host applications. It means there is no interaction/handshaking between processing cores and hardware components on the FPGA right after a request is submitted to a submission queue as a submission command. After a command is submitted to a submission queue, the hardware controller interacts with the commands regardless of the source of requests. It enables the possibility that all commands be processed by all the available corresponding accelerators one after each other.

To eliminate the delay of fetching commands, multiple commands need to be fetched and queued to the hardware controller in a command queue while previous commands are being processed by the accelerators. Despite commands with the same accelerator requests have equal opportunities of being processed, a single command queue in the hardware controller can cause significant performance degradation when various types of accelerators exist. Mix-

ing commands with different types of requests in a single command queue imposes a huge stall time to the accelerators with faster processing time. The reason is that the commands with the requests to the slower accelerators block other commands to reach the top of the queue to be scheduled for being processed in the accelerators, while their corresponding accelerators are available.

There are two different approaches that can be considered to address this issue: 1) proposing a scheduler that travels in the queue and is not restricted to only process the command on top of the queue, and 2) dedicating a single command queue for each accelerator type. Despite the feasibility of the first mechanism, it requires a more complicated scheduler and also suffers from imposing a not negligible latency to the system for finding a command to match an available accelerator.

To address this issue, UltraShare proposes an accelerator grouping mechanism that is based on a multi-queue architecture. In this way, accelerators are classified into different groups and for each group of the accelerators, a single command queue is allocated. A simple grouping architecture is to classify accelerators with the same type in the same group. Thus, a single command queue will be allocated to each type of accelerator. This allows commands to be processed as soon as their corresponding accelerators are available.

The accelerator grouping mechanism equips UltraShare with other features as well including the partial reconfiguration of accelerators to change the accelerators, and the possibility of using the multi-layer grouping. UltraShare uses a lookup table programmable from the host to manage accelerator grouping. Using UltraShare, users can easily partially reconfigurable the accelerator's area and by submitting a control command program the lookup table to reflect the changes in the accelerator's arrangement. The multi-layer grouping can be used for many reasons including providing priority and guaranteed processing rate.

5.3.3 Minimizing Accelerators Stall Time

While in storage systems due to the similarity of requests (being a read or a write and both need the same resources to be accomplished) resolving conflicts between cores trying to reach resources is very important to gain the maximum throughput [103], in the FPGA acceleration with various types of accelerators another conflict arises that ends up blocking requests. In the FPGA acceleration, each command can only be processed by the accelerators with the same type that it is requested. Thus, as long as commands are not blocking each other to reach their requesting accelerators, the maximum throughput is achievable. However, in the FPGA acceleration mixing commands with different request types can ultimately block those that target accelerators with higher throughput. On the other side, since commands in the submission queues are fetched by the hardware controller, the hardware controller is not aware of the type of accelerators that commands are requesting. That means, to fetch new commands, the hardware controller needs to make sure that there are enough spaces available for the new commands in the hardware controller. Thus, the hardware controller always considers the minimum available space among all command queues in the hardware for fetching new commands. This ultimately results in a huge throughput degradation for faster accelerators and limit their throughput to the slowest accelerators. To address this issue and take advantage of no-conflict between cores trying to reach the same accelerators (explained in Section 5.3.1), UltraShare Express proposes a new multi-queue architecture. In this architecture, for each processing core, a single completion queue and multiple submission queues are introduced. The number of submission queues per each core is equal to the number of accelerator types that are implemented on the FPGA and each submission queue is allocated to an accelerator type.

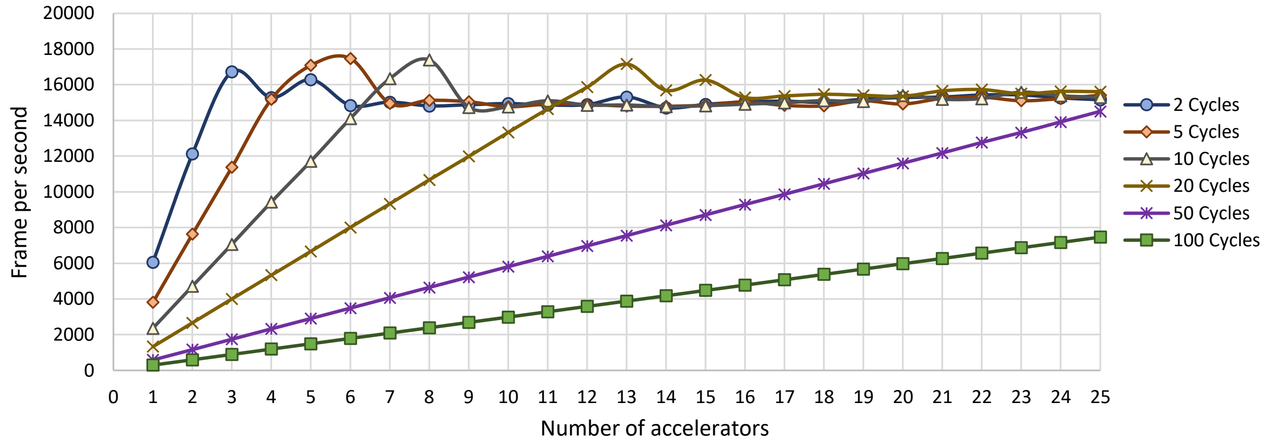


Figure 5.2: The throughput of different accelerator types with different processing delay (presented as clock cycle) for different number of instances implemented on a single FPGA.

5.3.4 Dynamic data-link Usage Balancing

Our motivation for this contribution comes from the fact that different accelerator types have different processing times. Therefore, to gain the best performance from each accelerator type, different instances of each need to be implemented. For more clarification, we conducted an experiment that presents a visual observation. In this experiment, we explore different accelerators with different processing times. To do so, we implemented a simple IP core based on the behavior of some real accelerators that we used in the experimental section from Xilinx® and MVD®. this IP core can be configured to introduce different delays as the processing time on each window of input data (each window has 128 bits of data). Figure 5.2 shows the throughput of this IP core with different configured delays for different numbers of instances. As the figure shows, when the number of instances of each accelerator increases the throughput increases linearly until saturating the bandwidth of the PCIe data-link. Figure 5.2 shows that accelerators with longer processing time, saturate the PCIe bandwidth with larger numbers of instances. Thus, as it can be observed, to gain the best performance from each accelerator, a different number of instances of that accelerator type needs to be implemented.

On the other side, when multiple accelerator types are implemented on an FPGA, due to the different number of instances, a round-robin scheduler fails to fairly share the data-link bandwidth among different type of accelerators, i.e. accelerator types with higher numbers of instances (slower accelerators) have more opportunities to receive higher portions of the bandwidth. To address this issue, UltraShare Express uses a dynamic bandwidth balancer mechanism using two modules performing Algorithms 2 and Algorithm 1 explained in detail in section 5.4. We show that by deploying the load balancer, UltraShare is capable of managing data-link access and allowing different types of accelerators to use it fairly.

5.4 UltraShare: Hardware and Software Design

To fulfill the policies and protocols presented in Section 5.3, we developed UltraShare Express in two main subsystems: a software stack and a hardware controller. These subsystems work together to integrate user applications and FPGA accelerators in a fully shared environment. In this section, we provide a detailed explanation of both subsystems. It can help users and researchers who are willing to use or expand UltraShare Express to have a better understanding of the way that UltraShare Express works.

5.4.1 Software Stack

UltraShare's software stack is composed of two parts: a library and a device driver called UltraShare driver. UltraShare driver is responsible for building commands and queuing them in the submission queues, pushing doorbells, receiving completions, and making applications aware of the completions. At the initiation, in the Linux kernel, UltraShare driver registers itself as a character driver and through I/O registers handshakes with the hardware controller. UltraShare driver is composed of three main parts: 1) command submission, 2)

interrupt and completion handler, and 3) wait completion.

Command submission part: For this part, the UltraShare driver allocates two main data structures at the initiation time: 1) submission queues and 2) record queues. Record queues keep the track of submitted commands. As we discussed in Section 5.3.1 and Section 5.3.3, the number of submission queues is the production of the number of host processing cores and the number of FPGA accelerator types. The number of FPGA accelerator types comes from the hardware controller at the handshaking time. The number of record queues is based on the number of host processing cores. *Command submission part* is responsible for generating and pushing commands to the corresponding submission queues and ringing their doorbells. Each submission command includes all the information that is needed for hardware accelerators to process the request including pointers to specify the input and output addresses on the host main memory.

For each submitted command, a *record command* is queued in the proper recording queue to be used at the completion time. The record command includes command id, process id, and accelerator type.

completion command and interrupt handler: Based on the number of host processing cores, the UltraShare driver allocates completion queues and message signaled interrupts (MSIs) and through handshaking reports them to the hardware controller. When a command is completed in the hardware controller, using the core id, the hardware controller writes the *completion command* to the corresponding completion queue through a DMA request. Then an interrupt is generated by the hardware controller to aware of the corresponding core of the completion. At this point, the corresponding core runs the interrupt routine and reads the *completion command*. Then by matching the *completion command* and *record command*, UltraShare driver keeps the track of the completed requests of a specific type for a specific process id. Due to the per processing core multi-queue structure of the completion handler, there is no contention on accessing resources, and therefore, no need for locking mechanisms.

Table 5.1: UltraShare Express software stack API table

API	Arguments	Return	Explanation
open_fpga	• int fpga_num	fpga* fpga	
close_fpga	• fpga* fpga	void	
acc_fpga	• fpga* fpga • int acc_type • char* input • int input_len • char* output • int output_len	bool success	Non-blocking function to submit a request for accelerator type of acc_type
wait_fpga	• fpga* fpga • int acc_type • int num_waiting	int num_succeed	Blocking function for waiting for the completion of num_waiting number of submitted requests
reset_fpga	• fpga* fpga	void	Reset the hardware controller
group_acc_fpga	• int group_number • int acc_number	bool success	To program the accelerator grouping table

wait completion: User applications essentially need to be informed of the finishing time of the accelerator requests that they submit. UltraShare introduces a waiting mechanism that allows users to wait for a specific number of requests that have been submitted. The waiting mechanism is blocking on a conditional kernel routine wait that wakes up by a kernel internal signal. The wake-up signal is issued when the condition of the number of received completion greater than the number of waiting completion is satisfied. Thus, *wait completion* part is directly actuated by the updates from *interrupt and completion handler* part.

UltraShare library interfaces UltraShare driver by introducing APIs to allow user applications accessing I/O controllers of the driver. Table 5.1 presents the main APIs of UltraShare. In the library, besides providing a set of user-friendly APIs, the process id of the threads invoking FPGA accelerators is extracted and forwarded to the device driver to be used for tracking the requests from individual threads.

5.4.2 Hardware controller

We designed UltraShare's hardware controller in the pure Verilog programming language. The hardware controller is composed of five main blocks: 1) memory mapped I/O controller, 2) doorbell controller, 3) command controller, 4) data manager, 5) accelerators controller. The command controller block is composed of two main parts: multi-queue grouping, and dynamic accelerator allocator. The data manager also block is composed of two main parts: scatter-gather manager, and data transfer. Following we provide a detailed discussion about each part and block.

Memory mapped I/O Controller (MMIC):

This part is responsible for handshaking and passing control signals between the hardware controller and host processing cores. MMIC uses the programmed I/O (PIO) mechanism for transferring data and signals. This mechanism is under device driver control through registers on the FPGAs. Doorbell signals are transferred through MMIC. In the handshaking, information like the number of accelerator types, and the number of accelerators are transferred from FPGAs to the host, and information like the base physical address of submission and completion queues, and length of submissions and completions queues are transferred from the host to FPGAs.

Doorbell controller:

The doorbell controller is responsible for receiving doorbell signals, and scheduling command fetch with regards to the available spaces for new commands in the hardware command queues. Each doorbell signal is a single physical address of the host's main memory that pints to the command of the top of a submission queue. In doorbell controller, for each submission

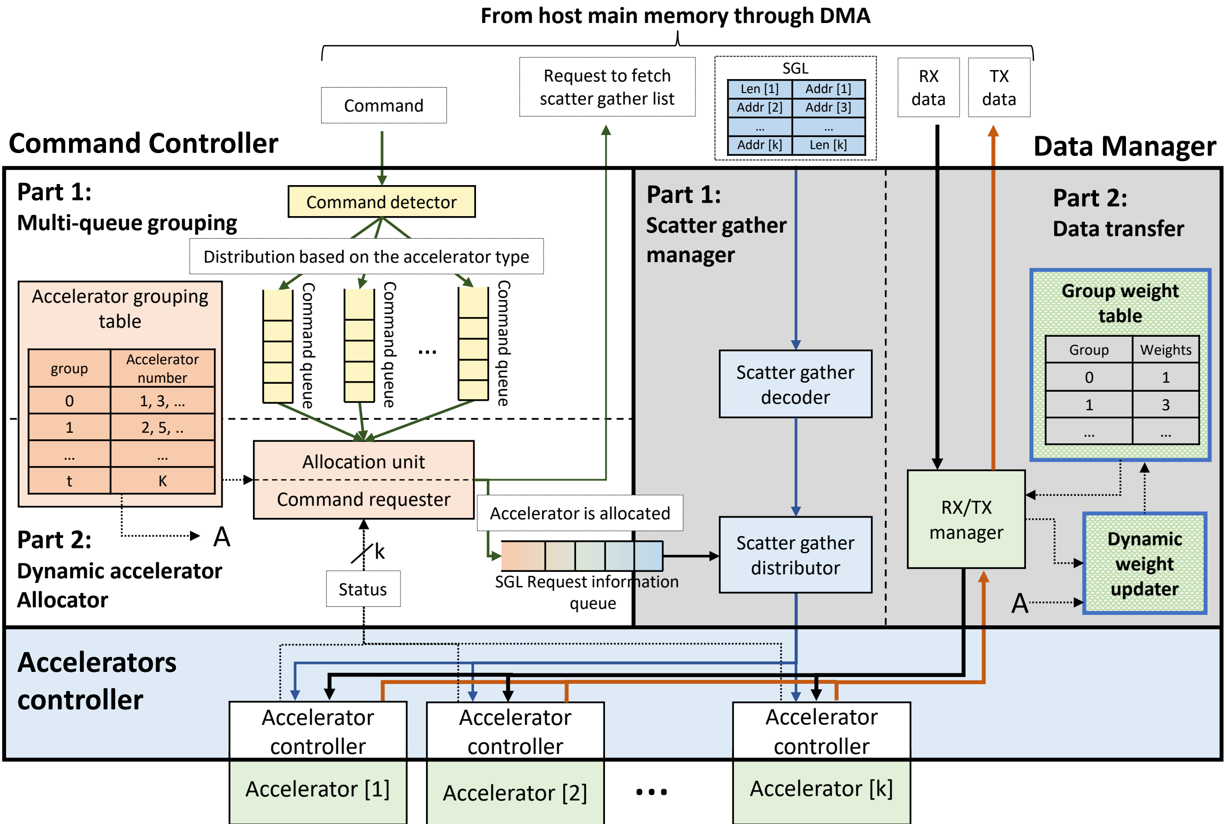


Figure 5.3: UltraShare hardware controller

queue, a module called *register_controller* exist that manages three registers. These registers respectively store the first address of the submission queue, the last address of the submission queue, and the current position of the submission queue. Each doorbell signal updates the value of the last register (the current position of the submission queue). Doorbell controller has a module called *fetch_command* that deploys a round-robin scheduler to circle around *register_controllers* and with the respect of the available spaces in the hardware command queues, initiates command fetches.

Multi-queue grouping:

Multi-queue grouping is responsible for managing and queuing commands to the corresponding commands queues based on their specified group. Figure 5.3 shows a high-level block

diagram of parts of the hardware controller that are related to the processing steps of a command. As it is showed in Figure 5.3, *Multi-queue grouping* is Part 1 of the command controller block, and it is composed of the following components.

- *Command detector:* Based on the information provided in a command, the command detector pushes the command into one of the command queues. In this chapter, this information only includes the accelerator type that the command is requesting, however, as we mentioned in section 5.3.2, UltraShare framework allows more sophisticated scenarios, for example, a multi-level priority-based grouping. In this way, some of the accelerators can be reserved for high-priority requests.
- *Command queues:* Each command queue is a FIFO implemented with BRAMs. For each group of accelerators, there is one dedicated command queue.

Dynamic accelerator allocator:

Along with *Multi-queue grouping*, *Dynamic accelerator allocator* is responsible for accelerator grouping and accelerator allocation to commands as explained in Section 5.3.2. This part is composed of the following units.

- *Allocation unit:* The main unit of the dynamic accelerator allocation part is the Allocation unit. This unit assigns an accelerator to the command on top of a command queue. Allocation unit circles around the queues in a round-robin scheduling mechanism. The inputs to the Allocation unit are accelerators' status and the output of *accelerator grouping table* that represents the mapping of accelerator numbers to the accelerator groups.
- *Accelerator grouping table:* Accelerator grouping table is basically a programmable lookup table that provides the mapping information of accelerators to the accelera-

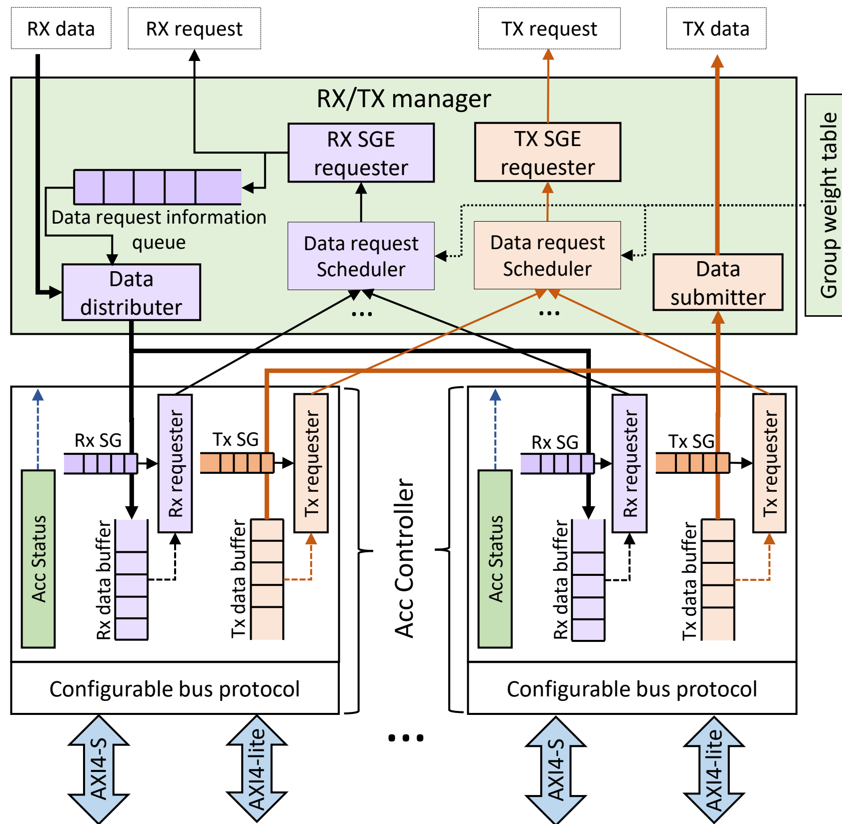


Figure 5.4: Detailed components of Accelerators controller and Data manager and their interconnections

tor groups. This lookup table is programmable from the host through a software API; thus, a user can rearrange the accelerator groups, removes an accelerator from a group, or adds an accelerator to a group. This allows users to benefit from the partial reconfiguration of accelerator regions, i.e. with a simple software API, users can rearrange the accelerator groups after replacing an accelerator with another.

- *Command requester:* After assigning an accelerator to a command, the *command requester* programs the DMA to fetch input data (RX) and output data (TX) scatter-gather lists.

- *SGL Request information queue:* The information related to each processed command will be stored in a queue to be used when the associated scatter-gather lists arrive.

Scatter-Gather (SG) manager

SG manager is responsible for receiving and decoding the arriving SG Lists (SGLs) and distributing them to the associate accelerators. *SG manager* is composed of the following components.

- *SG decoder:* A SGL is structured of a list of paged aligned memory addresses with their associated data lengths; while usually, the lengths of the first and last SGs are less than a memory page size, the lengths of the other SGs are equal to the page size. To shorten the size of the SGL, we compress it with not including the lengths of the middle SGs. When an SGL arrives, *SG decoder* extracts pairs of addresses and lengths from an SGL. We call each pair of address and length an SG element (SGE).
- *SG distributor:* This component uses the information queued in the *SGL Request information queue* to send SGEs to their assigned accelerator.

Accelerators controller

Accelerator controller: An accelerator controller allocates two main buffers: one RX data buffer and one TX data buffer. Figure 5.4 shows a detailed block diagram of the accelerator controller and data transfer parts. To optimize the size of RX and TX buffers, we define one RX SG queue and one TX SG queue for each input and output, respectively. Thus, each accelerator controller stores the whole list of SGEs. In this way, an accelerator issues one RX request if there is enough space available in the RX data buffer, and issues one TX request if there is enough data available in the TX buffer. This mechanism allows defining RX/TX data buffers with much smaller sizes. This size must be at least equal to the size of one memory page of the host (usually 4KB) which is the maximum length of one SG element. To prevent an accelerator stall due to waiting for RX data or a free TX buffer, we define

buffers with the size of a few numbers of memory pages.

Besides resolving the problem with large buffer sizes, our mechanism of fetching the data for each SGE allows providing a scheduling strategy for serving different accelerators. It allows serving SGEs from different accelerators with different weights that are explained in more detail in the data management part (section 5.4.2).

Interfaces: axi4 stream and axi4 lite

Data transfer

The *data transfer* part is responsible for moving RX and TX data through programming the DMA based on the SGEs from *Accelerator controller*s. Notably, the path for RX and TX data is completely separated and at the same time, an RX and a TX request can be responded to. *Data transfer* is also responsible for dynamic data-link usage balancing, as discussed in Section 5.3.4. Different components of the *Data transfer* are introduced in the following.

- *RX/TX manager:* This component responds to an accelerator and programs the DMA based on the SGE from that accelerator. Addressing the data-link usage balancing, *RX/TX manager* respond to the accelerator-based on their group weights coming from *Group weight table* using the algorithm presented in Algorithm 1. In this algorithm, based on the specified weights in *Group weight table*, each group of accelerators is served to transfer data through PCIe. In Algorithm 1, a circular FIFO is defined, called Responded_SGE_window, that keeps a window of last $C \times N$ accelerator types that have been served by *RX/TX manager* for a data transfer. By reporting this window to *Dynamic weight updater*, *Group weight table* is updated dynamically to balance the data-link bandwidth usage.

- *Data request information queue:* For each RX request, a data request information queue stores the information related to the request to be used when the corresponding data arrives. This information allows the *data distributor* component to submit the data to the correct accelerator.
- *Group weight table:* The *Group weight table* is a lookup table that represents the weights that the *RX/TX manager* unit uses for responding to the accelerator requests for data transfers.
- *Dynamic weight updater:* Working together with *Rx/Tx manager*, this component updates *Group weight table*. Algorithm 2 represents the main routine of *Dynamic weight updater* to update Group weight table based on the amount of bandwidth that each accelerator type receives. For simplicity, we keep the record of last $C \times N$ responded SGEs and based on that we dynamically update the weights and allow accelerator types with less being served to have higher weights in the next rounds.

Algorithm 1: Rx/Tx data manager

```
1  Inputs: weight[], NumAcc[]
2  Outputs: Responded_SGE_window[head], Responded_SGE_window[tail]
3  //N: Number of accelerator groups
4  //weight[i]: weights
5  //NumAcc[i]: Number of accelerators of type i
6  Define Responded_SGE_window[C × N]
7  for i : 0 to N do
8      if Req[i] == 1 then
9          for w : 0 to weight[i] do
10             for j : 0 to NumAcc[i] do
11                 Response to the request from accelerator[i][j]
12                 Responded_SGE_window[head] = i
13             end
14         end
15     end
16 end
```

Algorithm 2: Dynamic Weights Updating

```
1 Inputs: Responded_SGE_window[head], Responded_SGE_window[tail]
2 Output: weight[]
3 while True do
4     Updating Response_count[Responded_SGE_window[head]]
5     Updating Response_count[Responded_SGE_window[tail]]
6     for i : 0 to N do
7         weight[i] = 2 × C − Response_count[i]
8     end
9 end
```

5.5 Experimental

5.5.1 Experimental Setup

UltraShare's hardware controller is designed in Verilog programming language and we used Xilinx® Vivado design suite 2018.3 to synthesis it and generate its bitfile. To evaluate UltraShare, we used a 7v3-alpha-data board which has a Xilinx® Virtex 7 XC7VX690T with a PCIe Gen3. UltraShare's software stack that includes a library and a device driver is designed in C programming language. The device driver is a kernel module that is inserted into the kernel of the host Linux Ubuntu operating system. We use a host with an Intel Core i5-4590 3.30GHz CPU. Our host has an 8GB memory. In our experiments, to read video streams in host applications we use the OpenCV library.

To evaluate UltraShare, we use three different streaming accelerators: an image/video processing accelerator, an FIR filter, and a network packet encryption algorithm.

Image/video processing accelerator: We exploited an RGB-to-YCrCb image/video converter from Xilinx® ip libraries which is included in Xilinx® Vivado design suite 2018.3. RGB is not an efficient representation for images and videos for storage because it has a lot of redundancy. YCbCr is a practical approximation for images and videos that is suitable for being processed and stored. An RGB-to-YCrCb is an important operation that is on-demand

in datacenters and cloud to store streaming images and videos captured from end-devices [45].

The RGB-to-YCrCb image/video converter that we exploited is a streaming IP core with the standard AXI4-Stream input and output interface. This accelerator is configurable at the design time to be exploited for different resolutions of images/videos. For the evaluation, we use videos with resolutions of 240×180 and 480×320. Thus we configure two different accelerators with resolutions of 240×180 and 480×320 from the RGB-to-YCrCb converter.

FIR Filter: We exploited the Xilinx® FIR Filter compiler 7.2 IP core which is included in Xilinx® Vivado design suite 2018.3. We configured a single rate, one channel instance of the FIR Filter compiler. The input and output of this accelerator use the standard interface of AXI4-Stream. The conventional single-rate FIR version of the IP core computes the convolution sum which is shown in Equation 5.1, where N is the number of filter coefficients.

$$y(k) = \sum_{n=1}^{N-1} a(n) \times x(k-n), \quad k = 0, 1, \ldots \tag{5.1}$$

This IP core can process any size of input data with the same configuration. To make the analyses easier, in our experiments we supply this core with video streams with the resolution of 240×180.

Network packet encryption: Packet encryption/decryption is a streaming computation-intensive algorithm. This algorithm is a good candidate for being accelerated on FPGAs [84]. We exploited the AES-128 encryption/decryption IP core from MVD®. This core uses the AXI4-Stream interface for input/output data and an AXI4-lite interface for controlling the IP core. This core can operate over different input sizes with the same configuration. The same as FIR Filter, we only use video streams with the resolution of 240×180 to supply

this IP core.

The pseudocode presented in algorithm 3 is used to measure the data rate process reported in our experiments. In this algorithm, we use a while loop to submit FPGA acceleration requests for a period of time, and then using the wait_fpga() function we wait for the completion of all the submitted requests. The submission time is measured in the points that are shown in the algorithm 3 in line 7 and 16. In line 18, the throughput is measured by dividing the number of processed requests by the processing time.

Algorithm 3: Programming model and measuring the end to end delay

```
1  Function main():
2      buffer **A, **B
3      fpga_t *fpga
4      int i = 0 // i is the number of submitted requests
5      int fpga_id = 0
6      fpga = fpga_open(fpga_id)
7      ...
8      // Start of measuring the delay
9      while Time < MeasuringPeriod do
10         // A[i] : frame i of a video stream
11         // B[i] : Result of frame A[i] being processed
12         if acc_fpga(fpga, acc_id, A[i], length_A[i], B[i], length_B[i]) > 0 then
13             i++
14         end
15     end
16     wait_fpga(fpga, acc_id, i)
17     // End of measuring the delay
18     // Calculating the throughput
19     throughput = i/delay
20     fpga = fpga_close(_id)
21 End Function
```

5.5.2 The impact of deploying a single command based structure

Using a single-command-based structure allows accelerator requests from different applications to be queued without any interaction with the processing cores until they are processed

Table 5.2: The impact of single command-based invoking of FPGA accelerators

Framework	# of Threads	Throughput (frame/sec)	
		each thread average	Total processed by the accelerator
RIFFA	1	4560	4560
	2	648	1296
	4	75	300
UltraShare	1	4562	4562
	2	2288	4576
	4	1143	4572

by the corresponding accelerators. However, in the previous works including industrial and academic works, a one to one allocation of accelerators to the applications results in dropping requests from other applications to the same accelerators. To show the impact of this behavior we compared UltraShare with RIFFA since RIFFA is the closest framework to our framework considering the data path for moving data from/to FPGAs.

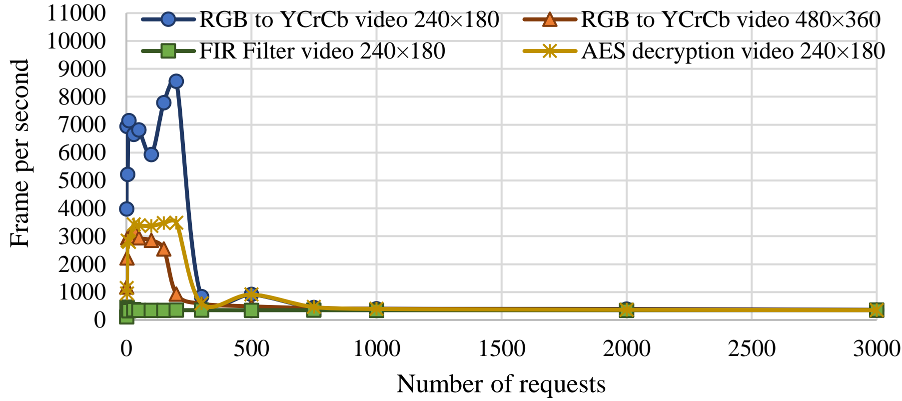
For this experiment, we implanted a single instance of the RGB-to-YCrCb converter with the resolution of 240×180. By increasing the number of applications requesting submitting video streams to this single accelerator we show the data process throughput that each application receives in Table 5.2. As Table 5.2 shows, using RIFFA with increasing the number of applications a huge performance degradation will occur due to failure to resolve the conflict among the applications. In this case for each accelerator call, the software stack is executed and when the target accelerator is busy the request is rejected. Thus, the user application requires to call the accelerator again rerun the software stack until the request is accepted. While due to the similarity of data path we compared UltraShare only with RIFFA, the other existing frameworks including industrial frameworks like SDAccel suffer from the same shortcoming.

Table 5.3: Dynamic allocation of accelerators vs. static allocation in previous works

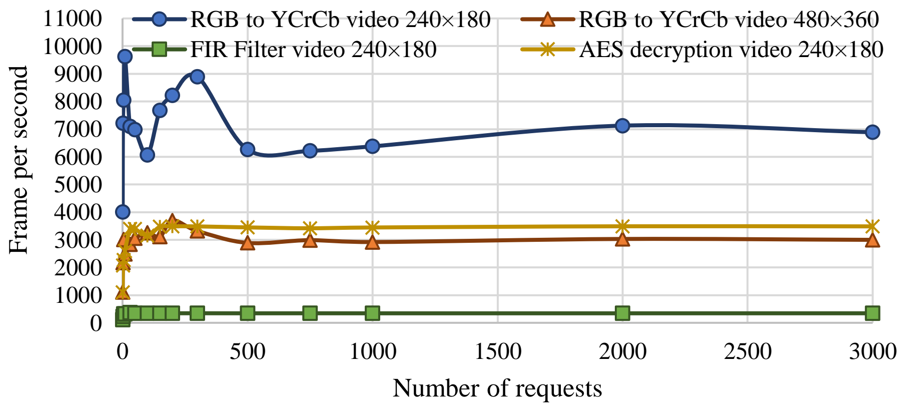
Framework	# of Threads To each accelerator				Throughput (frame/sec)
	AES Decryption 1	AES Decryption 2	AES Decryption 3	AES Decryption 4	
User defined allocation in previous works	1	1	1	1	4716
	2	1	1	0	2983
	2	2	0	0	2359
	3	1	0	0	2112
	4	0	0	0	1180
UltraShare	Dynamic allocation within the hardware controller				4721

5.5.3 The impact of dynamic accelerator allocation

To the best of our knowledge, all the previous works including industrial and academic works require users to determine the target accelerator in the application code. In an interactive system with multiple accelerator types with different numbers of instances, it is impossible or very difficult for users to be aware of all the applications' behavior requesting FPGA accelerators. In Table 5.3 we show the impact of dynamic allocation of available accelerators vs. a static allocation defined in the user applications. To simplify the experiment and avoid confusion we consider four different applications requesting AES encryption accelerator for video streams with the resolution of 240×180. We implemented four instances of the AES accelerator on the FPGA. In the first experiment, we use the static allocation of accelerators in the user applications and we explore all the possible scenarios. For example in Table 5.3, for RIFFA the first row reports that when each application requests for different AES accelerators a throughput of 4721 is achieved, while the second row reports that when two of the applications submits requests for AES number 1 and the other two applications submit their requests for AES number 2 and AES number 3, respectively, a throughput of 2983 is achieved. The reported throughput in this table is the summation of the throughput of all four accelerators. As Table 5.3 shows, for this simple scenario the throughput can degrade up to 4 times due to a static allocation of accelerators.



a) One submission queue per core



b) Submission queues based on the number of cores and accelerator types

Figure 5.5: Accelerators throughput (req/sec) of submission queue per core vs. submission queue per accelerator type.

5.5.4 The impact of different multi-queue architectures in the software stack

To show the impact of different architectures for multi-queue structure in the software stack we implemented two different scenarios: 1) defining a single submission queue for each core of the host processing unit; and 2) defining A submission queues for each processing core, while A is equal to the number of accelerator types.

In Figure 5.5, the throughput of 4 different accelerators is presented for different numbers of requests to each accelerator when each accelerator is invoked by a different application.

In this experiment for each accelerator type 3 instances are implemented. Figure 5.5.a represents the throughput for the structure of a single queue for each processing core and Figure 5.5.a represents the throughput for the structure of a single queue for each accelerator type. As Figure 5.5.a shows, with increasing the number of requests for each accelerator, the throughput of faster accelerators degrades drastically and saturated to the throughput of the slowest accelerator. The reason is that since the hardware controller is not aware of the types of accelerators that each command in the submission queues is requesting, the maximum number of commands that can be fetched are equal to the minimum spaces available in the command queues in the hardware controller. Thus, the rate of fetching commands from the submission queues is saturated to the rate of processing commands in the slowest command queue in the hardware controller. Consequently, the throughput of all the accelerators will be saturated to the throughput of the slowest accelerator when the number of requests is high enough to fill up the command queues in the host controller. However, by defining the number of submission queues based on both the number of processing cores and accelerator types, due to not having mixed commands in the submission queues, the hardware controller can fetch commands with respect to the accelerator types and the processing rate of those accelerators. Thus, as Figure 5.5.b shows, the throughput of accelerators remains the same for any type of accelerators.

5.5.5 The impact of enabling accelerator grouping in the hardware controller

Accelerator grouping is one of the main advantages of UltraShare that provides a fully shared environment for different applications to access FPGA-based accelerators. Despite all the benefits that accelerator grouping can propose, in this experiment, we only consider a single scenario of grouping based on accelerator types. For this purpose we consider two different architectures for the hardware controller: 1) hardware controller with a single command

Table 5.4: The throughput multiple accelerators being invoked by multiple applications simultaneously

	Throughput (frame/sec)	
Accelerator	Single queue None grouping	UltraShare
RGB to YCrCb video 240×180	670	6542
RGB to YCrCb video 480×360	651	2058
FIR Filter video 240×180	622	619
AES decryption video 240×180	632	2839

queue shared among all the accelerators which are our baseline; 2) hardware controller with a single command queue for each group of accelerators (more specifically for each type of accelerators). Deploying a single command queue in the hardware controller forces the queue to match the processing rate of the slowest accelerator. Thus, the throughput of all the accelerators would be limited to the throughput of the slowest accelerator. Table 5.4 shows the throughput of 4 different types of accelerators while 3 instances of each type are implemented on an FPGA. As the results show, while in a single queue none-grouping architecture all the accelerators are limited to the throughput of the FIR Filter accelerator (the slowest accelerator), UltraShare allows different types of accelerators to reach their ultimate processing power by removing the stall time of the accelerators resulted from the interaction of all accelerators with a single shared command queue. It worth mentioning that in this experiment, for example, for RGB-to-YCrCb accelerators with the resolution of 240×180 the ultimate processing power is more than 6542 frame/second, however, due to saturating the bandwidth of the PCIe when all the 12 accelerators (3 instances of 4 types of accelerators) are processing simultaneously, the throughput of these accelerators lowered down.

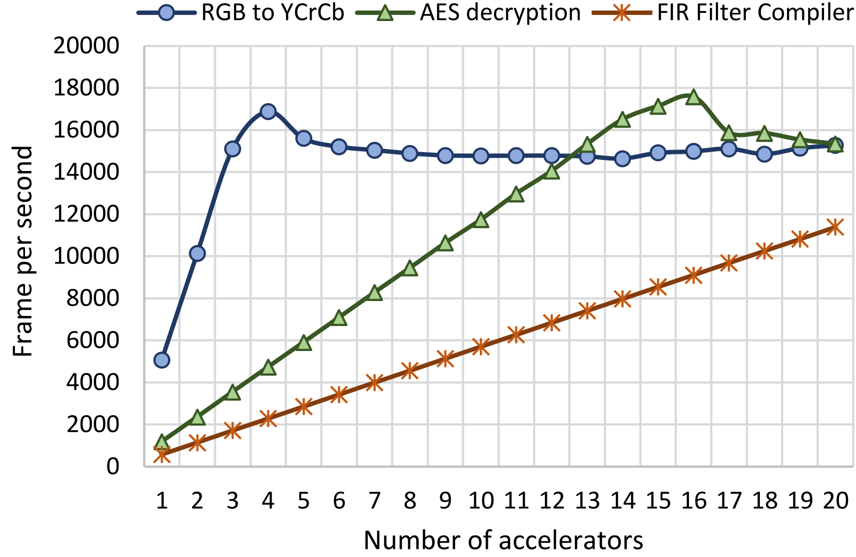


Figure 5.6: Accelerators throughput for different number of accelerator instances

5.5.6 Balancing data-link usage

Figure 5.6 shows the throughput for different instances of three different accelerator types when each type is examined alone with no existence of other accelerator types processing. It is notable that for each of these accelerators 240×180 video streams were used as inputs. In Figure 5.6, for the accelerators with higher processing delay, the maximum throughput appears for a larger number of the accelerator instances. For example, in our case study, an RGB-to-YCrCb accelerator will gain the maximum throughput with 4 instances, and deploying more instances won't help to increase the throughput. In fact due to adding more complexity to the system for accelerator allocation and data management, instantiating more instances slightly degrades the throughput. Thus, to gain the best performance one can find the best number of instances of a specific accelerator type. Using the information from Figure 5.6 we can find the optimum number of instances for each type of accelerators.

To evaluate the balancing data-link usage in UltraShare, we conducted an experimental result by exploiting two different types of accelerators. By considering Figure 5.6, we implemented the optimum instances of the faster accelerator among the two. Then we started

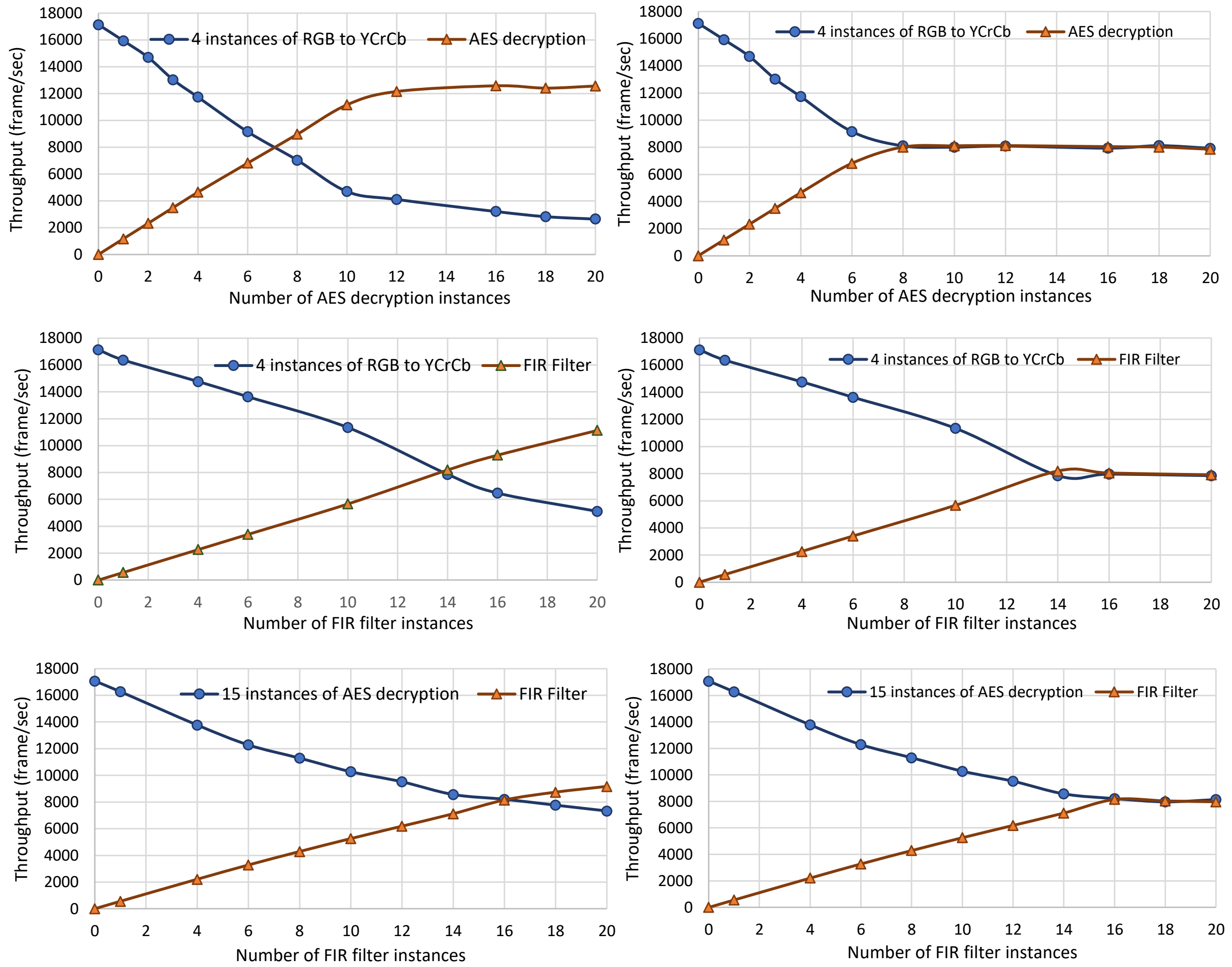


Figure 5.7: Fair allocation of PCIe bandwidth among accelerator groups

implementing the number of instances of the second accelerator from zero. We experimented this experiment with different combinations of choosing two accelerators from three different accelerator types. Figure 5.7 shows the throughput of two accelerator types implemented on a single FPGA. In this figure, the number of instances for the faster accelerator is the optimum number to gain the maximum throughput. For the slower accelerator, we increase the number of instances (the x-axis). In Figure 5.7, the left diagrams show the throughput using the none data-link usage balancing mechanism, and for the right diagrams, the data-link usage balancing mechanism is enabled. As this figure shows, with increasing the number of the slower accelerator, the throughput of the faster accelerator continues to decrease until from some points, it delivers less throughput comparing to the slower accelerator. It happens because of the saturation of data-link bandwidth by the higher number of instances of the slower accelerator. As the right diagrams show, our proposed data-link usage balancing mechanism does not allow the faster accelerator to face data starving in such scenarios.

5.6 Conclusion

In this chapter, we proposed UltraShare Express a scalable dynamic multi-application to multi-accelerator integration framework. UltraShare Express inherited the advantages of our previously proposed frameworks MQMAI [80] and UltraShare [78]. In addition, through our experiments, we detected a new blocking situation in the existence of mixed requests from each application. We proposing a new architecture for multi-queue structure in software stack to overcome this performance degrading issue. In this chapter, we also showed that in FPGAs, due to the diversity of accelerators, data starving can happen for accelerators with higher throughput. To overcome this problem, we introduced an algorithm to measure the data rate of different accelerators and fairly distribute data-link bandwidth among them.

Chapter 6

Conclusions and Future Directions

6.1 Conclusion

In the era of big-data applications with the high demand for processing power, clouds and datacenters are becoming the main places for processing data. Thus, in the new future, end-devices and personal and commercial computers send huge amounts of data continuously to clouds and datacenters for both processing and storing. Reaching close to the end of Dennard scaling and multi-core processors, heterogeneous architectures and hardware accelerators are playing an eminent role to accommodate the demanding processing powers.

Among hardware accelerators, FPGAs attracted the attention of researchers and industries due to being flexible, reconfigurable, energy-efficient, and potentially performance efficient. However, because of their different nature of programming comparing to software programming, FPGAs are complicated to be used by application designers. Also, as opposed to other peripherals, FPGAs are reconfigurable hardware devices that interfacing them is not as simple as the other hardware devices because, for each specific design, an appropriate way of communication is needed. To enable using FPGA accelerators, an interfacing framework

is needed to integrate host running applications to FPGA accelerators.

In clouds and datacenters, various applications with different computational intensive kernels are being executed. Unlike GPUs, since various independent accelerators can be implemented on FPGAs, a single FPGA has the potential of responding to the requests from multiple applications simultaneously. Due to the limited number of resources on FPGAs, it is very important to share FPGA resources among different applications to avoid under-utilization of resources. However, the currently available frameworks for the integration of software applications and FPGA accelerators either fail or cannot fully support efficient accesses to FPGA accelerators.

In this dissertation, we discussed the limitation of the available industrial platforms and research-based frameworks that provide application-to-accelerator integration. Then through realizing the major flaws, we introduced three major frameworks. In Chapter 3, we introduced Multi-Queue Multi-Accelerator Interface (MQMAI) that focuses on resolving the FPGA access conflicts among multiple applications running simultaneously on host computers. MQMAI deploys a single-command based protocol that enables two main characteristics: 1) accelerators are not bounded to applications (instead they are bounded to commands), and 2) removing additional interactions between FPGAs and host processing cores while a request is being processed (this avoid host processing core blocking or interrupting). MQMAI also introduces a multi-queue architecture in the software stack to avoid conflicts between cores submitting their FPGA accelerator requests. Our experimental results show huge improvements in FPGA acceleration comparing to the state-of-the-art frameworks in multi-core processors in the existence of multiple FPGA accelerators.

In Chapter 4, we introduced UltraShare to enable full-sharing of FPGA accelerators among various applications. There are three main features that an accelerator sharing scheme requires to support: 1) exploiting dynamic parallelism of multiple accelerators, 2) sharing accelerators among multiple applications, and 3) providing a non-blocking congestion-free

environment for multiple applications to call multiple accelerators. UltraShare proposes an accelerator grouping mechanism in the FPGA side of the integration framework (hardware controller) that eliminates commands to be blocked by accelerators with lower processing throughput. UltraShare also standardizes the connection of the hardware controller to accelerators by AXI4-Stream standard connection. Our Experimental results show a considerable performance improvement (on real accelerators from major companies in the field) comparing to MQMAI as the closest framework to UltraShare.

In Chapter 5, we introduced UltraShare Express that inherits the advantages of MQMAI and UltraShare through combining them. Also, we further improved the software stack multi-queue architecture of UltraShare Express to overcome the blocking issue that happens in the existence of various accelerator requests from single applications. UltraShare Express provides a fully shared non-blocking congestion-free environment for datacenters and cloud applications to exploit streaming FPGA accelerators. In Chapter 5, we also showed -through real acceleration scenarios- that in the existence of diverse accelerators a starving situation is very common to happen to accelerators with higher throughput. To overcome this issue, we proposed our data link usage balancing algorithm that is implemented in the UltraShare Express hardware controller. Our Experimental results show the performance advantages and fair distribution of data link bandwidth among various accelerators.

6.2 Directions for Future Work

UltraShare Express is an open-source framework with a huge potential for expansion. To develop UltraShare many obstacles have been passed. While HDL programming and operating system kernels design are very challenging and time-consuming, we developed UltraShare Express in a pure Verilog code with a rich supporting device driver. Other researchers can benefits from UltraShare express to either use it for FPGA accelerations of their specific

streaming applications or expand it for more general purposes. Following we suggest some opportunities and directions for the future of UltraShare Express to other researchers.

6.2.1 Immediate extensions of this dissertation

Some applications, like real-time object detection algorithms, require a guaranteed rate of data process that sharing accelerators can not necessarily assure that. In this case, the dedication of one or more accelerators to an application can resolve the problem while it raises the problem of under-utilization of accelerators. Using the accelerator grouping mechanism that we proposed in UltraShare in Chapter 4, we can provide the possibility of prioritizing commands. This can happen through a multi-layer grouping with the first layer of priority and the second layer of accelerator types. Using the multi-layer mechanism, besides the guaranteed rates of throughput, we can promote it to provide the possibility of prioritizing requests. Thus, each user can define the priority of their request in a static priority system.

While our proposed framework is originally developed for streaming accelerators, all types of accelerators can benefit from our proposing architectures and protocols. Another immediate extension to our proposed framework can be providing supports for general accelerators. To do so, some changes are required in the hardware controller to support address-based interfaces like AXI4. The general path is through control signals from the host; these control signals can be encapsulated in acceleration commands.

Our proposed accelerator grouping mechanism in Chapter 4, can open up an interesting path toward interactive use of FPGA accelerators in clouds and datacenters. We have proposed and implemented a grouping-based mechanism that allows host servers to have control over the grouping accelerators together. Taking advantage of this novel feature, by defining partially reconfiguration accelerator regions, it is possible to manage the throughput of each type of accelerators based on the demand. To do so, for reconfiguration of an accelerator

region, a control command is needed to be issued to eliminate that accelerator from the list of its current group, and after the reconfiguration, a control command needs to add that accelerator to the new group. All these processes happening while the rest of the accelerators are working with no interruption. After the changes also the gaining throughput changes dynamically for all the affected accelerator groups.

6.2.2 Novel research directions

The exploiting of FPGA accelerators remains a challenging process, until powerful platforms provide efficient automated mechanisms to implement complete enough libraries to support a vast range of operations. In this dissertation, we proposed a platform that is open-source and expendable. Expanding UltraShare Express to support general accelerators besides streaming accelerators, and providing a rich library of common accelerator IPs with automating their controlling signals can lead to a huge change in the area of FPGA acceleration.

Bibliography

[1] Intel® fpga sdk for opencl software technology. `https://www.intel.com/content/www/us/en/software/programmable/sdk-for-opencl/overview.html`. Accessed: 2020-10-02.

[2] Iot growth demands rethink of long-term storage strategies, says idc. `https://www.idc.com/getdoc.jsp?containerId=prAP46737220`. Accessed: 2020-11-01.

[3] Nvm express™ base specification. `https://nvmexpress.org/wp-content/uploads/NVM-Express-1_4-2019.06.10-Ratified.pdf`. Accessed: 2020-10-20.

[4] Xilinx® sdaccel: Enabling hardware-accelerated software. `https://www.xilinx.com/products/design-tools/software-zone/sdaccel.html`. Accessed: 2020-11-07.

[5] J. Ajanovic. *PCI Express*, pages 1487–1498. Springer US, Boston, MA, 2011.

[6] A. Alshehri and R. Sandhu. Access control models for virtual object communication in cloud-enabled iot. In *2017 IEEE International Conference on Information Reuse and Integration (IRI)*, pages 16–25. IEEE, 2017.

[7] W. Altoyan and J. J. Alonso. Investigating performance losses in high-level synthesis for stencil computations. In *2020 IEEE 28th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 195–203. IEEE, 2020.

[8] Amazon. Enable faster fpga accelerator development and deployment in the cloud. `https://aws.amazon.com/ec2/instance-types/f1/`. Accessed: 2020-11-15.

[9] Arrow. Fpga vs cpu vs gpu vs microcontroller: How do they fit into the processing jigsaw puzzle? `https://www.arrow.com/en/research-and-events/articles/fpga-vs-cpu-vs-gpu-vs-microcontroller`. Accessed: 2020-11-15.

[10] M. Attaran. Cloud computing technology: leveraging the power of the internet to improve business performance. *Journal of International Technology and Information Management*, 26(1):112–137, 2017.

[11] Avnet. Where is fpga in cloud computing today? `https://www.avnet.com/wps/portal/apac/resources/article/where-is-fpga-in-cloud-computing-today/`. Accessed: 2020-11-17.

[12] M. Bacis, R. Brondolin, and M. D. Santambrogio. Blastfunction: an fpga-as-a-service system for accelerated serverless computing. In *2020 Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 852–857, 2020.

[13] M. Bjørling, J. Axboe, D. Nellans, and P. Bonnet. Linux block io: introducing multi-queue ssd access on multi-core systems. In *Proceedings of the 6th international systems and storage conference*, pages 1–10, 2013.

[14] Boston. Gpu computing. `https://www.boston.co.uk/info/nvidia-tesla/default.aspx`. Accessed: 2020-11-17.

[15] B. BRECH, J. RUBIO, and M. HOLLINGER. Ibm data engine for nosql - power systems edition. In *Tech. rep., IBM Systems Group*, 2015.

[16] A. Canis, J. Choi, M. Aldham, V. Zhang, A. Kammoona, T. Czajkowski, S. D. Brown, and J. H. Anderson. Legup: An open-source high-level synthesis tool for fpga-based processor/accelerator systems. *ACM Transactions on Embedded Computing Systems (TECS)*, 13(2):1–27, 2013.

[17] Y.-T. Chen, J. Cong, Z. Fang, J. Lei, and P. Wei. When spark meets fpgas: A case study for next-generation DNA sequencing acceleration. In *8th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 16)*, Denver, CO, 2016. USENIX Association.

[18] Y. Chi and J. Cong. Exploiting computation reuse for stencil accelerators. In *2020 57th ACM/IEEE Design Automation Conference (DAC)*, pages 1–6. IEEE, 2020.

[19] Y. k. Choi and J. Cong. Hls-based optimization and design space exploration for applications with variable loop bounds. In *2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 1–8. IEEE, 2018.

[20] J. Cong, P. Wei, C. H. Yu, and P. Zhang. Automated accelerator generation and optimization with composable, parallel and pipeline architecture. In *2018 55th ACM/ESDA/IEEE Design Automation Conference (DAC)*, pages 1–6. IEEE, 2018.

[21] J. Cong, P. Zhang, and Y. Zou. Optimizing memory hierarchy allocation with loop transformations for high-level synthesis. In *Proceedings of the 49th annual design automation conference*, pages 1233–1238, 2012.

[22] R. A. Cooke and S. A. Fahmy. A model for distributed in-network and near-edge computing with heterogeneous hardware. *Future Generation Computer Systems*, 105:395–409, 2020.

[23] P. Coussy and A. Morawiec. High-level synthesis: from algorithm to digital circuit. 2008.

[24] F. Cross-Platform and A. Developers. Simplify software integration for fpga accelerators with opae. `https://01.org/sites/default/files/downloads/opae/open-programmable-acceleration-engine-paper.pdf`. Accessed: 2020-10-10.

[25] D. de la Chevallerie, J. Korinth, and A. Koch. ffLink: A lightweight high-performance open-source pci express gen3 interface for reconfigurable accelerators. *SIGARCH Computer Architecture News*, 43:34–39, 2015.

[26] L. Di Tucci, M. Rabozzi, L. Stornaiuolo, and M. D. Santambrogio. The role of cad frameworks in heterogeneous fpga-based cloud systems. In *2017 IEEE International Conference on Computer Design (ICCD)*, pages 423–426, 2017.

[27] J. Do, V. C. Ferreira, H. Bobarshad, M. Torabzadehkashi, S. Rezaei, A. Heydarigorji, D. Souza, B. F. Goldstein, L. Santiago, M. S. Kim, P. M. V. Lima, F. M. G. França, and V. Alves. Cost-effective, energy-efficient, and scalable storage computing for large-scale ai applications. *ACM Trans. Storage*, 16(4), 2020.

[28] Facebook research. Faiss. `https://github.com/facebookresearch/faiss`. Accessed: 2020-11-17.

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

[30] M. Ferdman, A. Adileh, O. Kocberber, S. Volos, M. Alisafaee, D. Jevdjic, C. Kaynak, A. D. Popescu, A. Ailamaki, and B. Falsafi. A case for specialized processors for scale-out workloads. *IEEE Micro*, 34(3):31–42, 2014.

[31] S. Garg, K. Kothapalli, and S. Purini. Share-a-gpu: Providing simple and effective time-sharing on gpus. In *2018 IEEE 25th International Conference on High Performance Computing (HiPC)*, pages 294–303. IEEE, 2018.

[32] Gartner.com. Hype cycle for emerging technologies 2017. `https://www.gartner.com/en/documents/3768572`. Accessed: 2020-11-17.

[33] Google. Available tensorflow ops — cloud tpu — google cloud. `https://cloud.google.com/tpu/docs/tensorflow-ops`. Accessed: 2020-11-17.

[34] B. Gu, A. S. Yoon, D.-H. Bae, I. Jo, J. Lee, J. Yoon, J.-U. Kang, M. Kwon, C. Yoon, S. Cho, et al. Biscuit: A framework for near-data processing of big data workloads. *ACM SIGARCH Computer Architecture News*, 44(3):153–165, 2016.

[35] Y. Guan, H. Liang, N. Xu, W. Wang, S. Shi, X. Chen, G. Sun, W. Zhang, and J. Cong. Fp-dnn: An automated framework for mapping deep neural networks onto fpgas with rtl-hls hybrid templates. In *2017 IEEE 25th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 152–159. IEEE, 2017.

[36] S. Gupta, A. Agrawal, K. Gopalakrishnan, and P. Narayanan. Deep learning with limited numerical precision. In *International Conference on Machine Learning*, pages 1737–1746, 2015.

[37] C. Hao, X. Zhang, Y. Li, S. Huang, J. Xiong, K. Rupnow, W.-m. Hwu, and D. Chen. Fpga/dnn co-design: An efficient design methodology for 1ot intelligence on the edge. In *2019 56th ACM/IEEE Design Automation Conference (DAC)*, pages 1–6. IEEE, 2019.

[38] I. A. T. Hashem, I. Yaqoob, N. B. Anuar, S. Mokhtar, A. Gani, and S. U. Khan. The rise of "big data" on cloud computing: Review and open research issues. *Information systems*, 47:98–115, 2015.

[39] A. HeydariGorji, S. Rezaei, M. Torabzadehkashi, H. Bobarshad, V. Alves, and P. H. Chou. Hypertune: Dynamic hyperparameter tuning for efficient distribution of dnn training over heterogeneous systems. *arXiv preprint arXiv:2007.08077*, 2020.

[40] A. HeydariGorji, M. Torabzadehkashi, S. Rezaei, H. Bobarshad, V. Alves, and P. H. Chou. Stannis: Low-power acceleration of dnn training using computational storage devices. In *2020 57th ACM/IEEE Design Automation Conference (DAC)*, pages 1–6. IEEE, 2020.

[41] Intel. Cloud computing. `https://www.intel.com/content/www/us/en/products/docs/storage/programmable/applications/cloud.html`. Accessed: 2020-11-17.

[42] Z. István, D. Sidler, and G. Alonso. Caribou: Intelligent distributed storage. *Proceedings of the VLDB Endowment*, 10(11):1202–1213, 2017.

[43] Z. István, G. Alonso, and A. Singla. Providing multi-tenant services with fpgas: Case study on a key-value store. In *2018 28th International Conference on Field Programmable Logic and Applications (FPL)*, pages 119–1195, 2018.

[44] J. Inkeles VP Product at Silexica. 5 challenges of high-level synthesis for fpga design. `https://www.silexica.com/blog/5-challenges-of-hls-for-fpga-design/`. Accessed: 2020-11-02.

[45] K. Jack. Video demystified: a handbook for the digital engineer. In *Elsevier*, 2011.

[46] M. Jacobsen, D. Richmond, M. Hogains, and R. Kastner. Riffa 2.1: A reusable integration framework for fpga accelerators. *ACM Trans. Reconfigurable Technol. Syst.*, 8(4), 2015.

[47] L. Johnsson and G. Netzer. The impact of moore's law and loss of dennard scaling: Are dsp socs an energy efficient alternative to x86 socs? In *Journal of Physics: Conference Series*, volume 762, page 012022. IOP Publishing, 2016.

[48] P. Jokic, S. Emery, and L. Benini. Binaryeye: A 20 kfps streaming camera system on fpga with real-time on-device image recognition using binary neural networks. In *2018 IEEE 13th International Symposium on Industrial Embedded Systems (SIES)*, pages 1–7, 2018.

[49] S.-W. Jun, M. Liu, S. Lee, J. Hicks, J. Ankcorn, M. King, S. Xu, and Arvind. Bluedbm: An appliance for big data analytics. In *Proceedings of the 42Nd Annual International Symposium on Computer Architecture*, ISCA'15, pages 1–13, New York, NY, USA, 2015. ACM.

[50] S. Karimi-Bidhendi, J. Guo, and H. Jafarkhani. Energy-efficient node deployment in heterogeneous two-tier wireless sensor networks with limited communication range. *IEEE Transactions on Wireless Communications*, 2020.

[51] S. Karimi-Bidhendi, F. Munshi, and A. Munshi. Scalable classification of univariate and multivariate time series. In *2018 IEEE International Conference on Big Data (Big Data)*, pages 1598–1605. IEEE, 2018.

[52] S. Kilts. *Advanced FPGA design: architecture, implementation, and optimization.* John Wiley & Sons, 2007.

[53] N. S. Kim, D. Chen, J. Xiong, and W. H. Wen-mei. Heterogeneous computing meets near-memory acceleration and high-level synthesis in the post-moore era. *IEEE Micro*, 37(4):10–18, 2017.

[54] D. Koeplinger, R. Prabhakar, Y. Zhang, C. Delimitrou, C. Kozyrakis, and K. Olukotun. Automatic generation of efficient accelerators for reconfigurable hardware. In *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, pages 115–127. Ieee, 2016.

[55] S. Kumar, P. Tiwari, and M. Zymbler. Internet of things is a revolutionary approach for future technology enhancement: a review. *Journal of Big Data*, 6(1):111, 2019.

[56] I. Kuon and J. Rose. Measuring the gap between fpgas and asics. *IEEE Transactions on computer-aided design of integrated circuits and systems*, 26(2):203–215, 2007.

[57] K. Lee and V. Rao. Accelerating facebook's infrastructure with application-specific hardware. `https://engineering.fb.com/2019/03/14/data-center-engineering/accelerating-infrastructure/`. Accessed: 2020-11-17.

[58] P. Li, P. Zhang, L.-N. Pouchet, and J. Cong. Resource-aware throughput optimization for high-level synthesis. In *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 200–209, 2015.

[59] J. Liu, J. Wickerson, S. Bayliss, and G. A. Constantinides. Polyhedral-based dynamic loop pipelining for high-level synthesis. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 37(9):1802–1815, 2018.

[60] C. Luo, M.-K. Sit, H. Fan, S. Liu, W. Luk, and C. Guo. Towards efficient deep neural network training by fpga-based batch-level parallelism. *Journal of Semiconductors*, 41(2):022403, 2020.

[61] J. Ma, G. Zuo, K. Loughlin, X. Cheng, Y. Liu, A. M. Eneyew, Z. Qi, and B. Kasikci. A hypervisor for shared-memory fpga platforms. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '20, pages 827–844. Association for Computing Machinery, 2020.

[62] P. Mantovani, E. G. Cota, C. Pilato, G. Di Guglielmo, and L. P. Carloni. Handling large data sets for high-performance embedded applications in heterogeneous systems-on-chip. In *Proceedings of the International Conference on Compilers, Architectures and Synthesis for Embedded Systems*, pages 1–10, 2016.

[63] J. Matai, D. Richmond, D. Lee, Z. Blair, Q. Wu, A. Abazari, and R. Kastner. Resolve: Generation of high-performance sorting architectures from high-level synthesis. In *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 195–204, 2016.

[64] J. M. Mbongue, F. Hategekimana, D. T. Kwadjo, and C. Bobda. Fpga virtualization in cloud-based infrastructures over virtio. In *2018 IEEE 36th International Conference on Computer Design (ICCD)*, pages 242–245. IEEE, 2018.

[65] J. M. Mbongue, A. Shuping, P. Bhowmik, and C. Bobda. Architecture support for fpga multi-tenancy in the cloud. In *2020 IEEE 31st International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, pages 125–132, 2020.

[66] V. Mirian and P. Chow. Ut-ocl: an opencl framework for embedded systems using xilinx fpgas. In *2015 International Conference on ReConFigurable Computing and FPGAs (ReConFig)*, pages 1–6, 2015.

[67] K. Neshatpour, M. Malik, A. Sasan, S. Rafatirad, and H. Homayoun. Hardware accelerated mappers for hadoop mapreduce streaming. *IEEE Transactions on Multi-Scale Computing Systems*, 4(4):734–748, 2018.

[68] M. A. Netto, R. N. Calheiros, E. R. Rodrigues, R. L. Cunha, and R. Buyya. Hpc cloud for scientific and business applications: taxonomy, vision, and research challenges. *ACM Computing Surveys (CSUR)*, 51(1):1–29, 2018.

[69] Nih center for macromolecular modeling bioinformatics. Gpu acceleration of molecular modeling applications. `http://www.ks.uiuc.edu/Research/gpu/`. Accessed: 2020-11-17.

[70] NVIDIA. Gpu-accelerated applications. `https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/tesla-product-literature/gpu-applications-catalog.pdf`. Accessed: 2020-11-17.

[71] M. Owaida, D. Sidler, K. Kara, and G. Alonso. Centaur: A framework for hybrid cpu-fpga databases. In *2017 IEEE 25th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 211–218, 2017.

[72] pcisig.com. Integrators list — pci-sig. https://pcisig.com/developers/integrators-list. Accessed: 2020-11-17.

[73] F. Peper. The end of moore's law: Opportunities for natural computing? *New Generation Computing*, 35(3):253–269, 2017.

[74] L.-N. Pouchet, P. Zhang, P. Sadayappan, and J. Cong. Polyhedral-based data reuse optimization for configurable computing. In *Proceedings of the ACM/SIGDA international symposium on Field programmable gate arrays*, pages 29–38, 2013.

[75] R. Prabhakar, D. Koeplinger, K. J. Brown, H. Lee, C. De Sa, C. Kozyrakis, and K. Olukotun. Generating configurable hardware from parallel patterns. *Acm Sigplan Notices*, 51(4):651–665, 2016.

[76] A. Putnam and et al. A reconfigurable fabric for accelerating large-scale datacenter services. In *Proceeding of the 41st Annual International Symposium on Computer Architecuture*, ISCA'14, pages 13–24, Piscataway, NJ, USA, 2014. IEEE Press.

[77] S. A. Razavi, E. Bozorgzadeh, and S. S. Kia. Communication-computation co-design of decentralized task chain in cps applications. In *2019 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1082–1087. IEEE, 2019.

[78] S. Rezaei, E. Bozorgzadeh, and K. Kim. Ultrashare: Fpga-based dynamic accelerator sharing and allocation. In *2019 International Conference on ReConFigurable Computing and FPGAs (ReConFig)*, pages 1–5. IEEE, 2019.

[79] S. Rezaei, C. Hernandez-Calderon, S. Mirzamohammadi, E. Bozorgzadeh, A. Veidenbaum, A. Nicolau, and M. J. Prather. Data-rate-aware fpga-based acceleration framework for streaming applications. In *2016 International Conference on ReConFigurable Computing and FPGAs (ReConFig)*, pages 1–6, Nov 2016.

[80] S. Rezaei, K. Kim, and E. Bozorgzadeh. Scalable multi-queue data transfer scheme for fpga-based multi-accelerators. In *2018 IEEE 36th International Conference on Computer Design (ICCD)*, pages 374–380, Oct 2018.

[81] S. Rezaei, S. G. Miremadi, H. Asadi, and M. Fazeli. Soft error estimation and mitigation of digital circuits by characterizing input patterns of logic gates. *Microelectronics Reliability*, 54(6-7):1412–1420, 2014.

[82] D. Richmond and M. Jacobsen. Riffa 2.2.2 documentation. https://github.com/KastnerRG/riffa/blob/master/docs/riffa_documentation.pdf, 2016. Accessed: 2020-11-15.

[83] L. Rota, M. Caselle, S. Chilingaryan, A. Kopmann, and M. Weber. A pcie dma architecture for multi-gigabyte per second data transmission. *IEEE Transactions on Nuclear Science*, 62(3):972–976, 2015.

[84] Z. Ruan, T. He, B. Li, P. Zhou, and J. Cong. St-accel: A high-level programming platform for streaming applications on fpga. In *2018 IEEE 26th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 9–16, April 2018.

[85] M. Sadeghi, S. A. Razavi, and M. S. Zamani. Reducing reconfiguration time in fpgas. In *2019 27th Iranian Conference on Electrical Engineering (ICEE)*, pages 1844–1848. IEEE, 2019.

[86] H. Sayadi and et al. 2smart: A two-stage machine learning-based approach for run-time specialized hardware-assisted malware detection. In *2019 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 728–733. IEEE, 2019.

[87] Y. S. Shao, B. Reagen, G.-Y. Wei, and D. Brooks. Aladdin: A pre-rtl, power-performance accelerator simulator enabling large design space exploration of customized architectures. In *2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*, pages 97–108. IEEE, 2014.

[88] H. Sharma, J. Park, D. Mahajan, E. Amaro, J. K. Kim, C. Shao, A. Mishra, and H. Esmaeilzadeh. From high-level deep neural models to fpgas. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1–12. IEEE, 2016.

[89] D. Sheffield. Ivytown xeon + fpga: The harp program. `https://cpufpga.files.wordpress.com/2016/04/harp_isca_2016_final.pdf`, 2016. Accessed: 2020-11-10.

[90] F. Siddiqui, S. Amiri, U. I. Minhas, T. Deng, R. Woods, K. Rafferty, and D. Crookes. Fpga-based processor acceleration for image processing applications. *Journal of Imaging*, 5(1):16, 2019.

[91] M. Siracusa and F. Ferrandi. Tensor optimization for high-level synthesis design flows. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 39(11):4217–4228, 2020.

[92] M. Siracusa, M. Rabozzi, E. Del Sozzo, M. D. Santambrogio, and L. Di Tucci. Automated design space exploration and roofline analysis for fpga-based hls applications. In *2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 314–314. IEEE, 2019.

[93] N. Suda, V. Chandra, G. Dasika, A. Mohanty, Y. Ma, S. Vrudhula, J.-s. Seo, and Y. Cao. Throughput-optimized opencl-based fpga accelerator for large-scale convolutional neural networks. In *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 16–25, 2016.

[94] S. Taheri. *Towards Engineering Computer Vision Systems: From the Web to FPGAs*. PhD thesis, UC Irvine, 2019.

[95] S. Taheri, J. Heo, P. Behnam, J. Chen, A. Veidenbaum, and A. Nicolau. Acceleration framework for fpga implementation of openvx graph pipelines. In *2018 IEEE 26th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 227–227. IEEE, 2018.

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

[97] H.-Y. Ting, T. Giyahchi, A. A. Sani, and E. Bozorgzadeh. Dynamic sharing in multi-accelerators of neural networks on an fpga edge device. In *2020 IEEE 31st International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, pages 197–204. IEEE, 2020.

[98] H.-Y. Ting, A. A. Sani, and E. Bozorgzadeh. System services for reconfigurable hardware acceleration in mobile devices. *2018 International Conference on ReConFigurable Computing and FPGAs (ReConFig)*, pages 1–6, 2018.

[99] M. Torabzadehkashi. *SoC-Based In-Storage Processing: Bringing Flexibility and Efficiency to Near-Data Processing*. PhD thesis, UC Irvine, 2019.

[100] M. Torabzadehkashi, A. Heydarigorji, S. Rezaei, H. Bobarshad, V. Alves, and N. Bagherzadeh. Accelerating hpc applications using computational storage devices. In *2019 IEEE 21st International Conference on High Performance Computing and Communications; IEEE 17th International Conference on Smart City; IEEE 5th International Conference on Data Science and Systems (HPCC/SmartCity/DSS)*, pages 1878–1885. IEEE, 2019.

[101] M. Torabzadehkashi, S. Rezaei, V. Alves, and N. Bagherzadeh. Compstor: An in-storage computation platform for scalable distributed processing. In *2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 1260–1267, 2018.

[102] M. Torabzadehkashi, S. Rezaei, A. Heydarigorji, H. Bobarshad, V. Alves, and N. Bagherzadeh. Catalina: In-storage processing acceleration for scalable big data analytics. In *2019 27th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP)*, pages 430–437. IEEE, 2019.

[103] M. Torabzadehkashi, S. Rezaei, A. HeydariGorji, H. Bobarshad, V. Alves, and N. Bagherzadeh. Computational storage: an efficient and scalable platform for big data and hpc applications. *Journal of Big Data*, 6(1), Nov 2019.

[104] S. M. S. Trimberger. Three ages of fpgas: A retrospective on the first thirty years of fpga technology. *IEEE Solid-State Circuits Magazine*, 10(2):16–29, 2018.

[105] M. Vesper, D. Koch, K. Vipin, and S. A. Fahmy. Jetstream: An open-source high-performance pci express 3 streaming library for fpga-to-host and fpga-to-fpga communication. *2016 26th International Conference on Field Programmable Logic and Applications (FPL)*, pages 1–9, 2016.

[106] K. Vipin and S. A. Fahmy. Dyract: A partial reconfiguration enabled accelerator and test platform. In *2014 24th international conference on field programmable logic and applications (FPL)*, pages 1–7. IEEE, 2014.

[107] J. Weerasinghe, R. Polig, F. Abel, and C. Hagleitner. Network-attached fpgas for data center applications. In *2016 International Conference on Field-Programmable Technology (FPT)*, pages 36–43, 2016.

[108] Xilinx. Fpgas in the emerging dnn inference landscape. `https://www.xilinx.com/support/documentation/white_papers/wp514-emerging-dnn.pdf`. Accessed: 2020-11-12.

[109] Xilinx. Sdaccel development environment help for 2019.1. `https://www.xilinx.com/html_docs/xilinx2019_1/sdaccel_doc/cuu1526001449959.html`.

[110] Xilinx. Sdx pragma reference guide. `https://www.xilinx.com/support/documentation/sw_manuals/xilinx2019_1/ug1253-sdx-pragma-reference.pdf`. Accessed: 2020-11-17.

[111] Xilinx. The xilinx sdaccel development environment. `https://www.xilinx.com/support/documentation/backgrounders/sdaccel-backgrounder.pdf`. Accessed: 2020-11-15.

[112] Xilinx. Virtex-7 fpga gen3 integrated block for pci express v4.1, 2015.

[113] Xilinx. Sdsoc development environment. `https://www.xilinx.com/publications/prod_mktg/sdnet/sdsoc-development-environment-backgrounder.pdf`, 2016. Accessed: 2020-11-10.

[114] C. H. Yu and et al. S2fa: An accelerator automation framework for heterogeneous computing in datacenters. In *Proceedings of the 55th Annual Design Automation Conference*, DAC '18, pages 153:1–153:6, New York, NY, USA, 2018. ACM.

[115] C. H. Yu, P. Wei, M. Grossman, P. Zhang, V. Sarker, and J. Cong. S2fa: an accelerator automation framework for heterogeneous computing in datacenters. In *2018 55th ACM/ESDA/IEEE Design Automation Conference (DAC)*, pages 1–6. IEEE, 2018.

[116] C. Zhang, P. Li, G. Sun, Y. Guan, B. Xiao, and J. Cong. Optimizing fpga-based accelerator design for deep convolutional neural networks. In *Proceedings of the 2015 ACM/SIGDA international symposium on field-programmable gate arrays*, pages 161–170, 2015.

[117] J. Zhao, L. Feng, S. Sinha, W. Zhang, Y. Liang, and B. He. Comba: A comprehensive model-based analysis framework for high level synthesis of real applications. In *2017 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 430–437. IEEE, 2017.

[118] G. Zhong, A. Prakash, Y. Liang, T. Mitra, and S. Niar. Lin-analyzer: a high-level performance analysis tool for fpga-based accelerators. In *2016 53nd ACM/EDAC/IEEE Design Automation Conference (DAC)*, pages 1–6. IEEE, 2016.

[119] S. Zhou and V. K. Prasanna. Accelerating graph analytics on cpu-fpga heterogeneous platform. In *2017 29th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, pages 137–144. IEEE, 2017.