UNIVERSITY POLITEHNICA OF BUCHAREST

Doctoral School of Electronics, Telecommunications and Information Technology

Decision No. 965 **from** 16-11-2022

Ph.D. THESIS SUMMARY

George-Vlăduț POPESCU

ARHITECTURI ȘI STRUCTURI PENTRU CALCUL ETEROGEN - ÎMBUNĂTĂȚIRI ALE TRANSFERULUI DE DATE PENTRU UN SISTEM ETEROGEN DE CALCUL

ARCHITECTURES AND STRUCTURES FOR HETEROGENEOUS COMPUTING - IMPROVEMENTS IN DATA TRANSFER FOR A HETEROGENEOUS COMPUTING SYSTEM

THESIS COMMITTEE

Prof. Dr. Ing. Gheorghe BREZEANU University Politehnica of Bucharest	President
Prof. Dr. Ing. Gheorghe ȘTEFAN University Politehnica of Bucharest	PhD Supervisor
Prof. Dr. Ing. Corneliu BURILEANU University Politehnica of Bucharest	Referee
Prof. Dr. Ing. Aurel-Ștefan GONTEAN Politehnica University Timișoara	Referee
Prof. Dr. Ing. Dan NICULA Transilvania University of Brașov	Referee

BUCHAREST 2023

Table of contents

Chapter 1

Introduction

1.1 Parallel Computing Architectures

The increasing need for computing power imposed by applications that require the processing of large volumes of data, such as those that process sound or images or those used in artificial intelligence, has led to the development of parallel computing architectures, which, in addition to the problem of reducing the execution time, also try to provide reduced energy consumption.

Each parallel computing application is characterized by a computation pattern closely related to the mathematical mechanisms involved. In [1] and [2], the main computational patterns were identified, as well as their presence in several general applications. From the hardware perspective, each computation pattern is defined by similar behavior in terms of computation and data movement.

Several architectures have been proposed over time to provide hardware support for various applications that require parallel computing, each of them trying to improve the *number of operations/consumed power* ratio. Examples of such architectures are: The Many Integrated Core (MIC) Architecture, a solution proposed by Intel for highly parallel computation [3–6], Graphics Processing Units (GPUs) [7–10], which are processors with a high number of cores initially designed to accelerate tasks related to computer graphics, or Google's Tensor Processing Unit (TPU), an Application-Specific Integrated Circuit, designed for machine learning applications [11, 12].

Although the previously presented architectures have high theoretical performance, their actual performance when used for different tasks may be much lower. As demonstrated in [6, 13–15] and summarized in [16], depending on the task, the actual performance can drop below 50% of the theoretical one.

This decrease can be explained, among others, by the lower degree of software optimization and the inappropriate way in which the memory is accessed. The problem of not fetching the data in an optimized way, together with the discrepancy between the increase in performance of processors and that of memories [12], can cause important

delays. Moreover, it is difficult for an architecture to be optimized in terms of execution time and power consumption for a wide range of applications.

The need for a solution that offers both flexibility and good performance has led to the emergence of heterogeneous systems. These systems integrate CPUs and other processing elements specialized for certain tasks. To support the development of these systems, FPGA manufacturers have created systems that combine a CPU and an FPGA on the same chip, allowing the user to describe his own processing core, which can be configured and adapted to the target application.

1.2 Motivation and Objectives of the Thesis

The increasing demand for computing power at the lowest possible costs, for both manufacturing of the chips and energy required for their exploitation, as well as the important difference between the peak performance and the actual performance when they are used in certain applications, motivates research in the field of architectures for parallel computing.

Adding as many processing cores as possible is not always a solution, because it can end up in a situation where, due to several factors, a large part of the computing power is not activated for an important number of applications. Furthermore, focusing on optimization for a certain application will make that architecture inflexible and ineffective for other applications.

A solution could be heterogeneous computing systems, which can provide the needed efficiency for a larger number of applications. Moreover, a system that is also reconfigurable, such as those implemented using FPGAs, offers great flexibility, being able to adapt to the requirements of certain applications [17, 18].

The general objective of this research is to propose a fully functional heterogeneous computing system based on an improved architecture of a MapReduce Accelerator that provides an efficient and flexible alternative solution for applications that require parallel data processing.

To achieve this, the first major objective is that, starting from an already existing MapReduce architecture, a new architecture with improved I/O data transfer is proposed. Data transfer was chosen as the target of the improvement efforts because, as mentioned in the previous section, unoptimized data transfers represent an important factor that negatively influences the performance of a system.

In order to be functional, the Accelerator must be integrated into a system that will ensure its operation by transmitting instructions and data and by reading the results and transferring them to the main memory. Therefore, the second major objective is to propose an architecture for a heterogeneous computing system that integrates the MapReduce Accelerator and to implement it on a PYNQ-Z2 board containing the Zynq-7020 SoC.

For the system to be fully functional, the user must have access to its resources with the help of a software environment. Thus, the third major objective of the research is the development of a programming environment that allows the writing of programs and their execution on the heterogeneous computing system.

The last major objective of the thesis is the development of test environments, which will be used to test the correctness of the system's operation and characterize it in terms of execution time and consumed energy.

1.3 Thesis Overview

The thesis is structured into six chapters and successively describes the development stages of the heterogeneous system, from the understanding of the problem that motivates the current research, to the evaluation setup and results, general conclusions, and proposals for further development and improvement of the proposed system.

Chapter 1 presents an overview of the main currently used parallel computing architectures, but also some details about their shortcomings, highlighted in the literature following the evaluation of their performance when used in various applications. Starting from these, the motivation and objectives of the current research are outlined.

Chapter 2 describes the main components of the MapReduce Accelerator that represents the starting point of the research and the instruction set it supports. Then, the proposed heterogeneous system architecture and how it is implemented on the Zynq-7020 are presented. In addition to the overall picture, the reasons for the choices made during the development of the system, as well as implementation and operation details, are presented.

Chapter 3 is dedicated to the detailed description of the principles and changes brought by the new MapReduce Accelerator and how these changes lead to improved data transfer between the main memory and the Accelerator.

Chapter 4 concentrates on the description of the last element of the proposed heterogeneous system, a Python-based programming environment that allows the user to access the system resources, to write libraries of functions and programs using the Accelerator-specific assembly language, and to execute them. In addition, this environment can be used to evaluate the correctness of the system's operation.

In Chapter 5, the two components of the system evaluation are presented: the evaluation of the execution time using a simulation test environment and the evaluation of the correctness of the system's operation using a hardware test environment. Furthermore, an estimate of the consumed energy is obtained. This chapter also contains the results of the tests and the conclusions arising from them.

Finally, in the last chapter, Chapter 6, the main conclusions and results, the original contributions of the current research and the perspectives for the improvement and further development of the proposed heterogeneous computing system are synthesized.

Chapter 2

Heterogeneous Computing System

In this chapter, a heterogeneous, pseudo-reconfigurable system capable of efficient execution of linear algebra tasks is presented. It integrates a mono- or a multi-core CPU, called the Host processor, and an Accelerator, specialized in parallel computing. In this system, the complex part of the program runs on the Host while the intensive computation sequences are executed on the Accelerator.

A simplified structure of the heterogeneous computing system is presented in Figure 2.1.

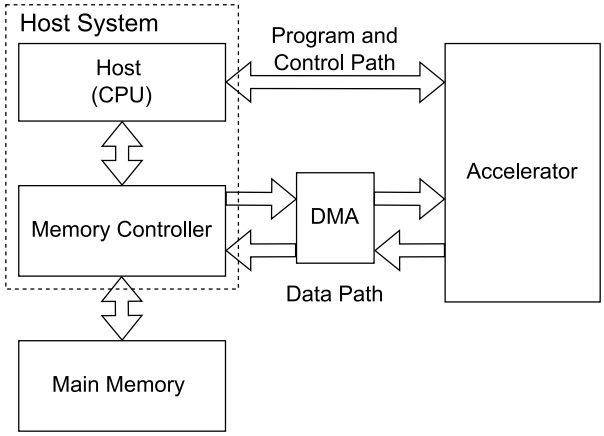

Fig. 2.1 The architecture of the heterogeneous computing system

The Accelerator is a MapReduce Accelerator which communicates with the Host System through the Program and Control Path and the Data Path. The Program and Control Path is used by the Host to access the Accelerator's status and control registers and to transmit commands. The Data Path is used by the Accelerator to communicate with the main memory of the system. To transfer data efficiently between the main memory and the Accelerator, the Data Path is controlled by a DMA.

Several architectures of this type of accelerator have been previously proposed, some of them being implemented on silicon, as in [19], and [20].

The current architecture is based on the Zynq-7020 SoC from Xilinx [21]. On this platform, the Host System is represented by the Processing System (PS), built around

a dual-core ARM Cortex-A9 processor, while the Accelerator, the DMA, and other modules needed for correct operation are implemented on the Artix-7 FPGA (PL).

2.1 MapReduce Accelerator

The MapReduce Accelerator is a parallel computing core that can process large amounts of data, making it suitable for the intensive computation part of a program.

The main components of the Accelerator are the Controller and the Parallel Processing Unit. The Controller is mainly used for coordinating the activity of the Accelerator and for performing operations on scalar data [22]. The Parallel Processing Unit is responsible for processing vector-type data and its main components are the Array of processing elements (cells), the Distribution Network, and the Scan-Reduce Network. An overview of the structure of the Accelerator is presented in Figure 2.2.

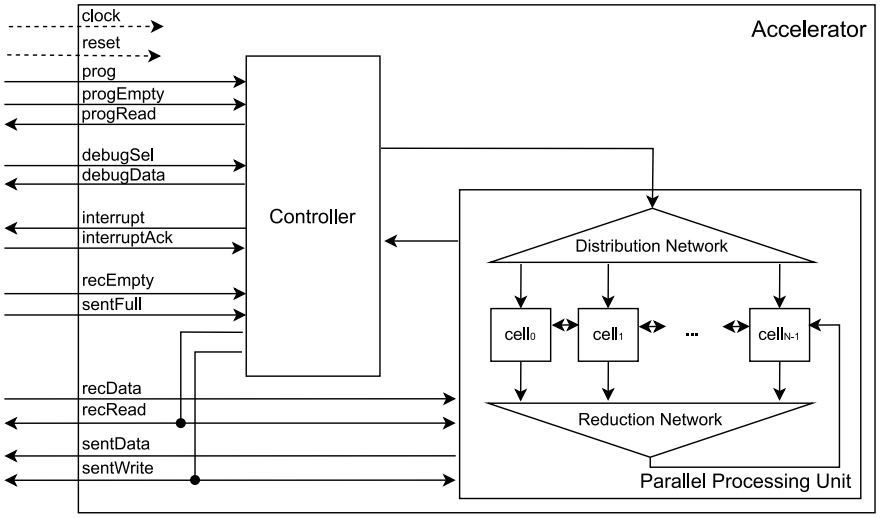

Fig. 2.2 The structure of the Accelerator [22]

2.1.1 The Controller

The main role of the Controller is to coordinate the execution process in the Accelerator. Its structure is divided into three main sections: a program section, responsible for the execution control, a data section, used to perform operations on scalars, having its own decode unit, scalar data memory, accumulator, and processing core, and a connector section, used to prepare and send the commands to the Parallel Processing Unit.

The program memory stores on each memory location two instructions, one for the Controller and one for the Parallel Processing Unit. The interaction with the Accelerator assumes that it executes functions already stored in its program memory on data received through the Data Input Path.

The controller also provides access to resources used for debugging and performance measurements, such as the current *PC* or the output of a clock cycle counter.

2.1.2 The Parallel Processing Unit

The Parallel Processing Unit is responsible for the processing of data organized as vectors or matrices. Its main components are the Distribution Network, the Array of processing cells, and the Scan-Reduce Network. A detailed view of the components of the Parallel Processing Unit and their interconnection is presented in Figure 2.5.

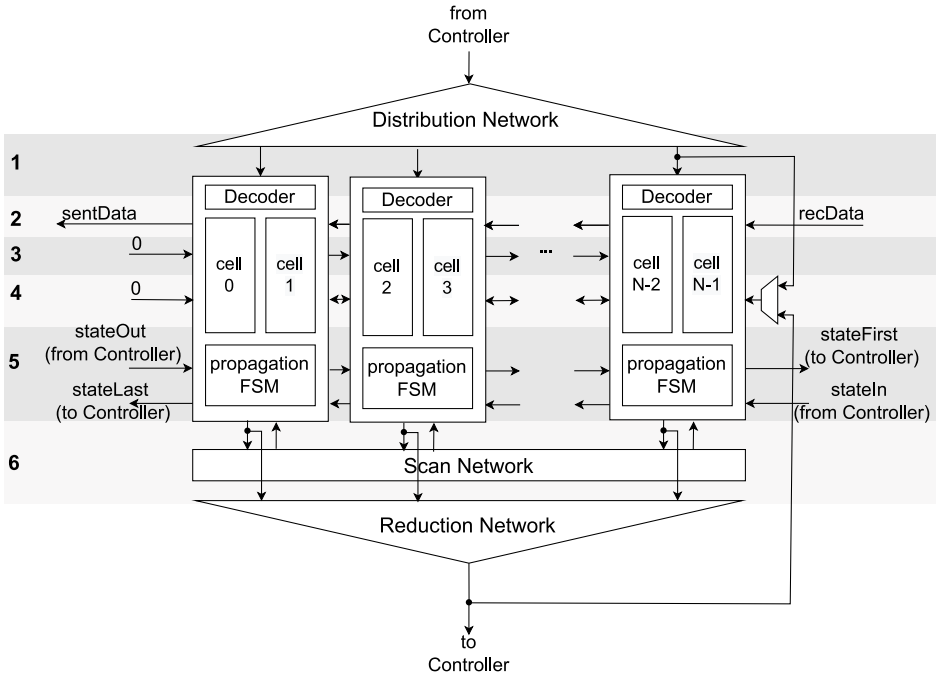

Fig. 2.5 The structure of the Parallel Processing Unit

The Distribution Network is a pipe-line structure used to send instructions, addresses, memory commands, or data from the Controller to the Array.

The central element in the Parallel Processing Unit is the Array of processing cells. Using this, a specific instruction can be executed on different data at the same time, each cell having its own controller, execution unit, and data memory [22].

Considering the presence of a local data memory in each cell and the layout of the Array, the internal data storage resources can be seen as a $2^V \times N$ matrix, where V is the address dimension and N is the number of cells.

In Figure 2.5, it can be observed that each cell in the Array communicates with other cells and elements in the Parallel Processing Unit through multiple command, data, and state channels. The distribution channel (1) ensures the distribution of commands from the Controller to each cell in the Array, the I/O data transfer (2), active state (3), data shift (4), and propagation state (5) channels ensure the communication of data and states between the cells, and the scan-reduce channel (6) ensures the communication between each cell and the Scan-Reduce Network.

Depending on how the Array is organized, a cell can share with other cells an instruction decode unit and a propagation finite state machine. In the current implementation,

the cells in the Array are organized into groups of two (double-cells), meaning that each decode unit and propagation finite state machine serves two cells.

The I/O data transfer channel is implemented as a chain of shift I/O registers, distributed in each group of cells. The data propagation is locally controlled by a finite state machine, associating a propagation state with each double-cell: *full*, if the cell cannot accept any data, or *empty*, otherwise.

A bidirectional shift channel connects the cells, providing the possibility to move data inside the Array. This is implemented as a serial-parallel register called the global shift register, distributed along the cells.

The Scan-Reduce Network has two components: the Reduction Network, which performs operations on vectors such as addition, finding the maximum value, or finding the minimum value, providing a scalar as a result, and the Scan Network, which performs operations on vectors and provides a vector as a result. The vector results are sent back to the Array and the scalar ones are sent back to the Array or to the Controller.

2.1.3 The Instruction Set

The Accelerator's instruction set includes instructions targeting the resources in the Controller or the ones in the Parallel Processing Unit. Furthermore, there are instructions that ensure the transfer of data and commands between the two main components of the Accelerator, or between the Accelerator and its interface.

Each 32-bit memory location in the Accelerator's program memory stores two instructions: one for the Controller and one for the Parallel Processing Unit. Almost all the instructions have a 16-bit format. The only exceptions are the jump, halt, and call instructions, which target the Controller, but they use all available bits on the Program and Control Path for the address, allowing access to a larger memory range.

2.2 System Architecture

In this section, the architecture of the heterogeneous computing system that integrates the MapReduce Accelerator will be presented. The way in which it is suitable for some of the applications that require parallel computing was analyzed in [23] and [24].

2.2.1 Overview of the Implementation Platform

Considering the architecture of the heterogeneous computing system, where both a Host processor and a custom Accelerator are needed, the Zynq-7020 SoC from Xilinx was chosen as the implementation platform. This integrates a Processing System (PS), built around a dual-core ARM Cortex-A9 processor and an Artix-7 FPGA (PL), which offers the user the flexibility of implementing any custom logic circuit. The architecture of the Zynq-7000 SoC family is presented in Figure 2.8.

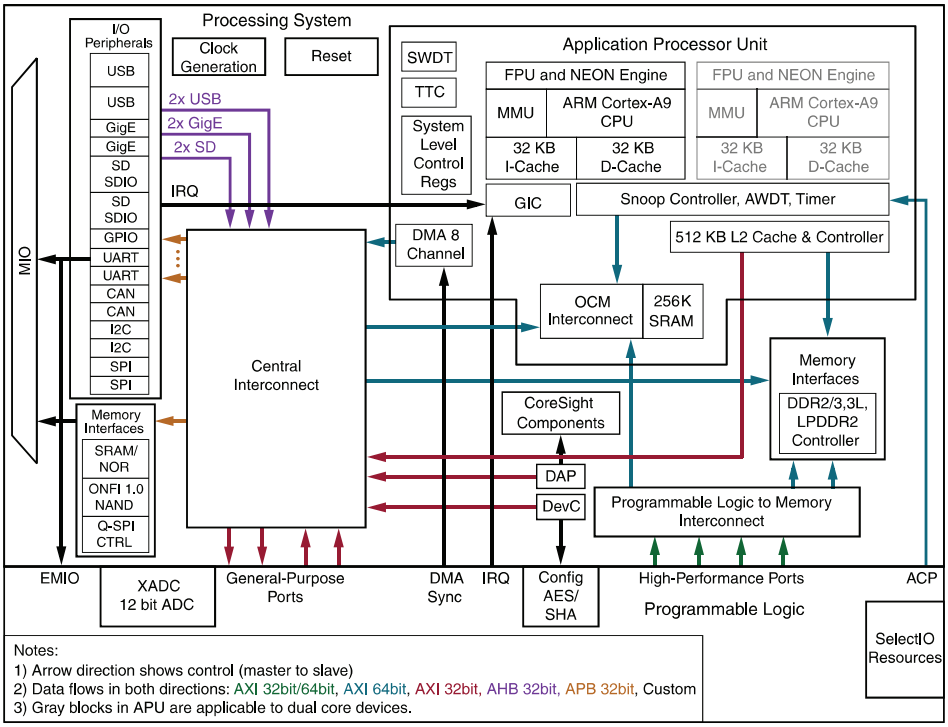

Fig. 2.8 The architecture of the Zynq-7000 SoC [21]

Besides the central element of the Processing System which is the Application Processing Unit (APU), other important components for the heterogeneous system are the Generic Interrupt Controller (GIC) which manages the interrupts from the peripherals and from the Programmable Logic, the DDR Memory Controller, which manages the interactions with the DDR Memory, and the AXI3 based PS-PL interfaces. Although the PS modules are AXI3 compatible, those in the PL usually use AXI4.

2.2.2 The Top Level Architecture of the System

Being implemented on the Zynq-7020 SoC, the Host processor of the heterogeneous system is represented by the Application Processing Unit in the PS, while the Accelerator is implemented in the PL section.

The system architecture, presented in Figure 2.9, takes into consideration the way the Accelerator must interact with the Host and the memory in order to function efficiently.

AXI4-Lite is a memory-mapped interface used in the current design to connect the Program and Control Path to the Host and to configure other components in the PL, such as the DMA and the Interrupt Controller. On the PS side, this bus is connected to one of the General-Purpose interfaces, which is further connected to the Central Interconnect.

The DMA is implemented using the Xilinx LogiCORE AXI DMA IP. The DMA data channels are connected to the Accelerator using two AXI4-Stream buses and to the PS using one of the four High-Performance ports, which implements the AXI4-Full

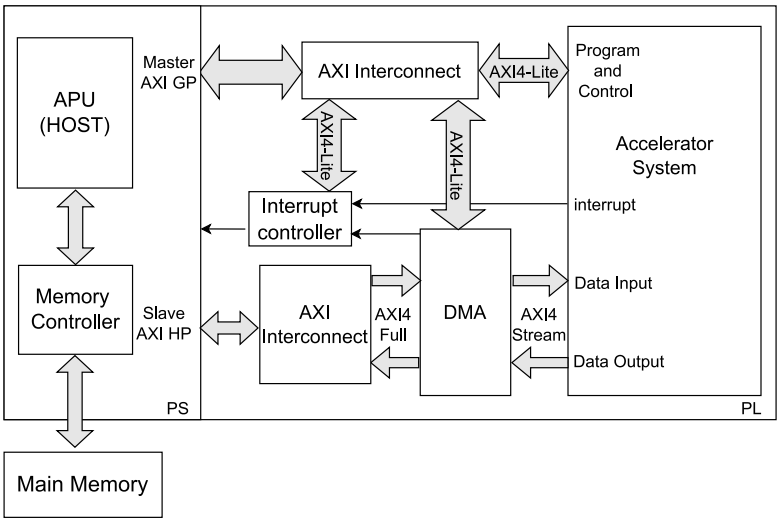

Fig. 2.9 The top level architecture of the heterogeneous computing system

protocol. The High-Performance bus is further connected inside the PS to the DDR Memory Controller, which ensures interaction with the main memory.

The Interrupt Controller is responsible for managing the interrupts from the DMA and the Accelerator and it is implemented using the Xilinx LogiCORE AXI Interrupt Controller IP. Its output signal is connected to the Generic Interrupt Controller (GIC).

As mentioned before, the modules in PS are AXI3 compatible. In order to ensure compatibility between the PS and the modules in the PL, additional interconnection modules are added (AXI Interconnect).

2.2.3 The Program and Control Path

The Program and Control Path is primarily used for sending instructions from the Host to the Accelerator and for accessing its control and status registers. Additionally, it is used to configure and control the DMA, responsible for data transfers, and the Interrupt Controller, responsible for managing the interrupts from the DMA and Accelerator.

The Program and Control Path is based on the AXI4-Lite bus [25]. The Processing System controls this bus through one of its General-Purpose AXI ports. The compatibility between the AXI3 PS port and the AXI4-Lite bus is ensured by an interconnection module (Xilinx LogiCORE AXI IP). The architecture of the Program and Control Path is presented in Figure 2.10.

The Accelerator interacts with the Program and Control Path using 3 groups of signals: the program group, the interrupt group, and the debug group. The program group is used to transfer the instructions from the Host to the Accelerator, the interrupt group is used by the Accelerator to issue an interrupt and receive an acknowledgement from the Host, and the debug group is used to transfer the debug information from the Accelerator to the corresponding register in the AXI-Lite Interconnect.

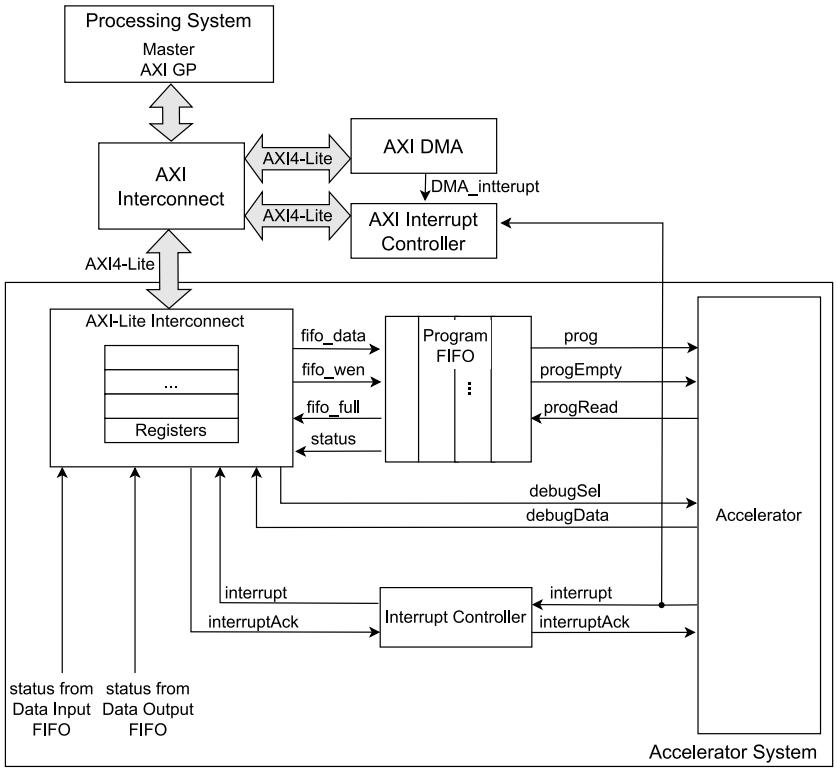

Fig. 2.10 The architecture of the Program and Control Path

AXI-Lite Interconnect is the module that interfaces the Accelerator with the AXI4-Lite bus. Its main components are the status and control registers and the logic that ensures the correct writing and reading of these registers.

The Program FIFO is used as a buffer for the instructions sent by the Host processor. The input data for this FIFO comes from the AXI-Lite Interconnect module. Each time the Host writes an instruction to address offset 0x00 on the AXI4-Lite interface, it will be forwarded to the FIFO.

The Interrupt Controller inside the Accelerator is an alternative way to handle the interrupts generated by the Accelerator.

2.2.4 The Data Path

The Data Path is used to transfer large amounts of data between the main memory and the Accelerator. It has a Data Input component, through which the Accelerator receives from the main memory the data to be processed, and a Data Output component, through which the results are read and stored into the main memory.

To be able to transfer large amounts of data in the most efficient way, the Data Path is controlled by a DMA. The architecture of the Data Path is presented in Figure 2.13.

The main memory is accessed by both the Host and the DMA using the DDR Memory Controller, integrated into the Processing System section of the Zynq-7020 SoC. The Data Path in the PL is connected to the DDR Memory Controller by one of the four slave High-Performance ports of the PS.

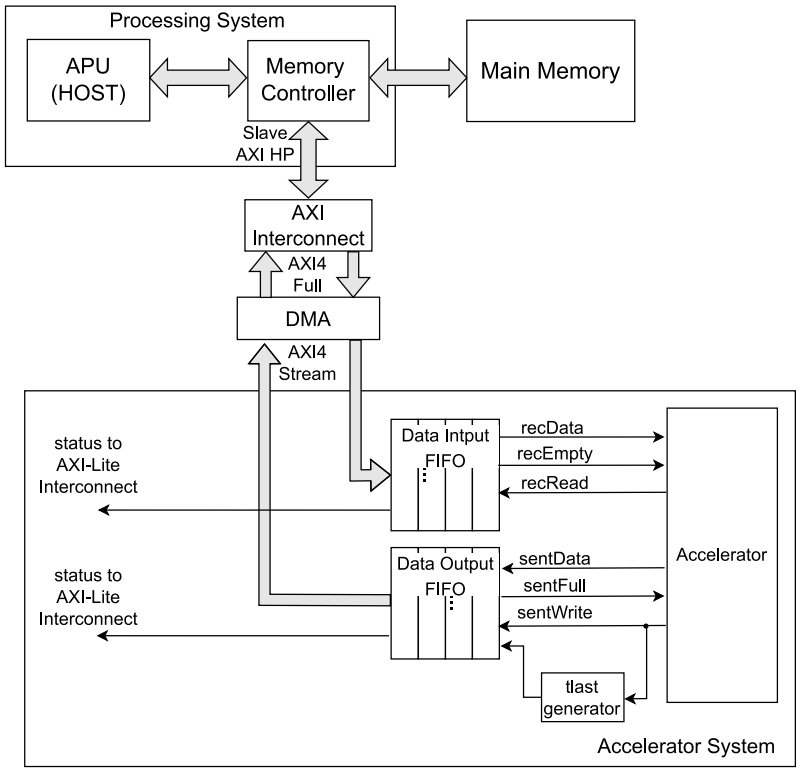

Fig. 2.13 The architecture of the Data Path

The operation of the AXI DMA module is based on two channels that operate independently: the Memory-Map to Stream (MM2S) channel, used to transfer data from the main memory to the Accelerator, and the Stream to Memory-Mapped (S2MM) channel, used to transfer data from the Accelerator to the main memory. The two independent channels of the DMA are responsible for protocol conversion from memory-mapped, which ensures the connection with the DDR Memory Controller, to AXI4-Stream [26], which ensures the connection with the Accelerator.

The interface of the Accelerator with the two AXI4-Stream channels from the DMA is done through two configurable FIFO structures.

The Data Input FIFO is used to temporarily store the data transferred from the main memory. The data will remain here until the Accelerator reads it. This is a synchronous FIFO with two different protocols for writing and reading. The input end of the FIFO is a simplified AXI4-Stream interface, and the output end is a basic FIFO interface, through which the Accelerator extracts data and moves it to its internal data memory.

The Data Output FIFO is used to store the results from the Accelerator until a read transfer is initiated by the DMA. The input end of the FIFO is a basic FIFO interface, and the output end is a simplified AXI4-Stream interface. In addition to the basic FIFO interface signals, the Data Output FIFO input requires a *tlast* signal, which will be used by the DMA to indicate the boundary of a read data packet.

Information such as the *full* and *empty* status, or the number of occupied locations in each FIFO is available to the Host through the control and status registers.

Chapter 3

Hardware Improvements in Data Transfer for the MapReduce Accelerator

3.1 Overview of the Weak Points of Data Transfer

In the process of improving data transfer, the current architecture of the Accelerator must be analyzed from two perspectives: identifying those elements that introduce delays on the Data Path, and analyzing the possibility of separating I/O data transfer and processing flows. If this separation is possible, the data can be brought into the Accelerator's memory at the right time, so that the time a processing sequence waits for the input data is as short as possible.

In the initial architecture, the data transfers between the Array and the Data Input and Output FIFOs are coordinated by the Controller. The propagation state machines in each double-cell used in the data shifting process insert a delay of one clock cycle at each shift step, considering that the data cannot go further until the next cell is in *empty* state. As an example, the data shift through the chain of I/O registers for an 8-cell Array is presented in Figure 3.1.

To achieve the separation of I/O data transfer and processing flows, the two must use different resources with separate control paths. Because the data shifting operation uses the chain of I/O registers, which is a separate storage resource from those used for data processing, its independence depends only on the implementation of a module that takes over the shift control from the Controller.

Furthermore, the data transfers between the I/O registers and the internal memories in the cells must pass through the Accumulator, as there is no direct connection between them in the current design. In the context of separating the data and processing flows, it is necessary for the data transfers between the I/O register and the internal memory to be carried out independently of the Accumulator. In addition, since simultaneous

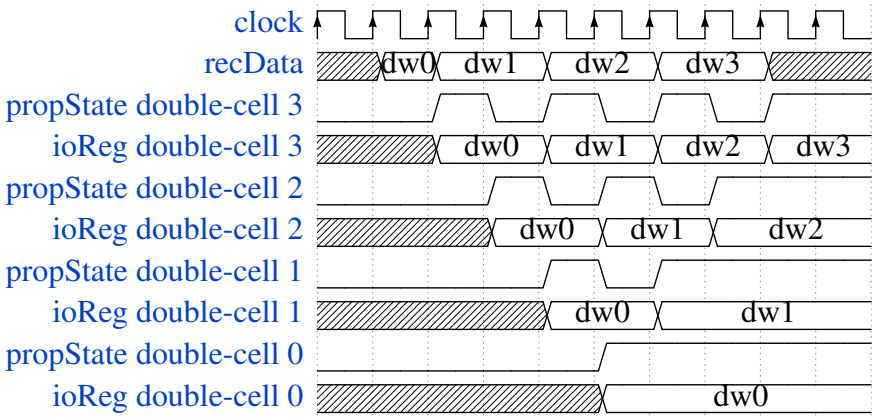

Fig. 3.1 Data shift through the chain of I/O registers for an 8-cell Array

requests to access the internal memory may occur from the I/O data transfer flow and the processing flow, it is necessary to implement additional arbitration logic [22].

Considering the previous observations, the execution flows in the current design of the MapReduce Accelerator and in the improved one are presented in Figure 3.3. P1, P2, and P3 are processing sequences, and D1, D2, and D3 are the I/O data transfer sequences of the corresponding input data. As it can be observed, in the initial architecture, new data can be transferred only after the current processing sequence has finished. If the I/O data transfer and the processing flows become independent, the transfer of new data can start immediately after the previous transfer has finished [22].

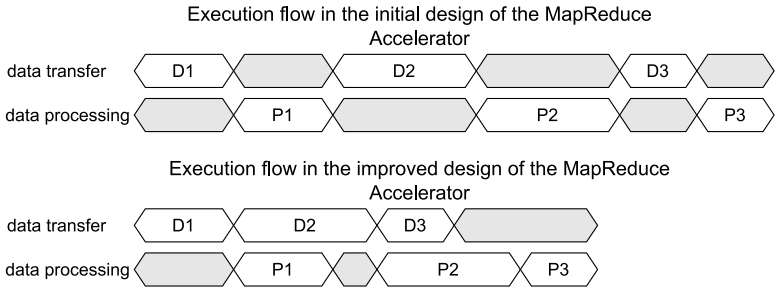

Fig. 3.3 Execution flows in the initial and improved designs of the MapReduce Accelerator [22]

3.2 Improved Array Architecture

One of the elements that slows down the propagation of data through the chain of I/O registers is the one clock cycle delay introduced by the propagation finite state machine at each shift step. This delay is caused by the fact that each double-cell waits for the next one to become available and be able to accept the data.

One way to eliminate the unwanted delay introduced by the propagation finite state machine at each shift step is to split the double-cells into two groups with different data propagation paths, that work alternately. In this way, new data will be accepted by the Array at each clock cycle.

An important observation is that the new way of organizing the cells only affects the structure of the I/O data transfer (2) and the propagation state (5) channels presented in Figure 2.5. From the perspective of functionality, the behavior will remain the same, this change being transparent to the user.

The new way of organizing the cells in the Array is presented in Figure 3.4. Only the new connections of the two channels involved in I/O data transfer are represented, as the way of connecting the other ones remains unchanged.

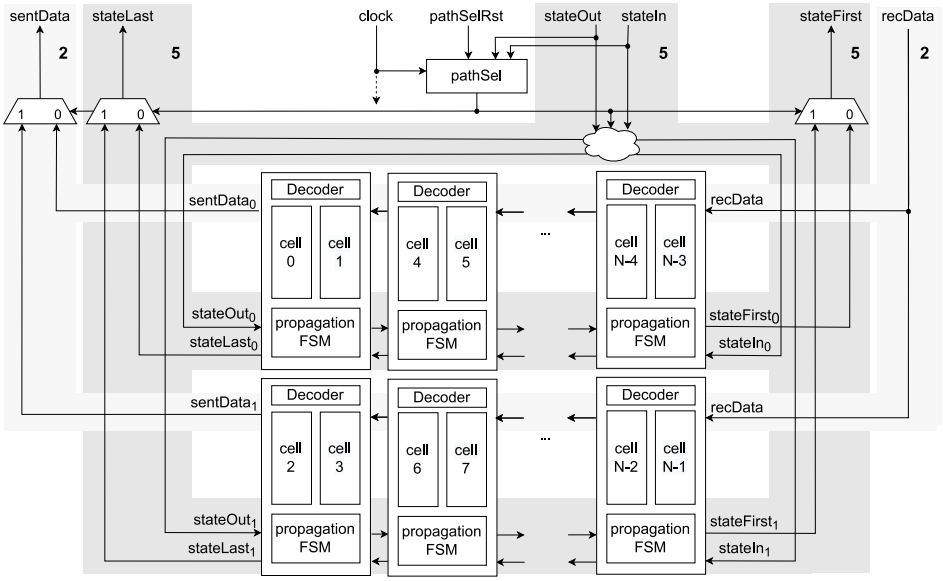

Fig. 3.4 The improved Array Architecture

Using this new architecture, it is possible to transfer data to and from the Array at each clock cycle, as presented in Figure 3.5, for the same 8-cell Array as in the previous section.

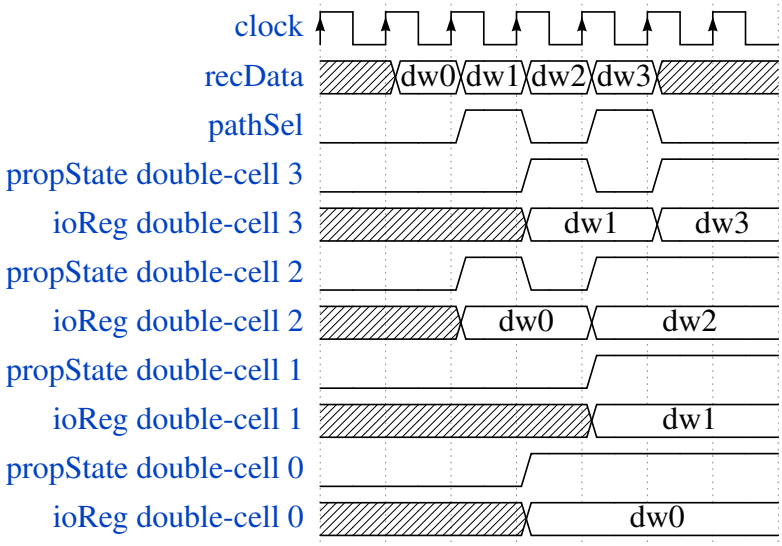

Fig. 3.5 Data shift through the chain of I/O registers for an 8-cell Array with improved architecture

3.3 The Data Transfer Engine

To achieve the requirements presented in the previous sections, a new module called the Data Transfer Engine was implemented [22]. It has the role of coordinating the data transfers between the Data Path FIFOs and the Accelerator, having a high degree of independence from the current states of the Controller and Parallel Processing Unit. In this way, the two modules can continue the processing of data already present in the internal data memory while data that will be used in subsequent processing is transferred. The Data Transfer Engine can access the internal memory for read and write transfers if the current running instructions do not initiate similar requests. Otherwise, it will wait for the higher priority access to finish.

So that the Data Transfer Engine can perform the I/O data transfers correctly and synchronize them with other modules when needed, the following new commands and instructions are introduced:

- TINRUN: Transfer Input Run: Data Transfer Engine command that performs the transfer of a data matrix in the data memory of the Array. This command needs to be followed by three parameters: the internal data memory address where the store will start, the number of lines, and the number of columns. If the number of columns is smaller than the number of cells in the Array, each data line will be padded with zeros.

- TOUTRUN: Transfer Output Run: Data Transfer Engine command that performs the transfer of a data matrix from the data memory of the Array to the Data Output FIFO. This command must be followed by three parameters: the internal data memory address where the read will start, the number of lines, and the number of columns. If the number of columns is smaller than the number of cells in the Array, the transfer of a data line will be considered finished after shifting out only the needed data.

- WAITRESREADY: Wait for Result to be Ready: Data transfer Engine command that tells the module to wait until the Controller marks a result as ready before starting its transfer to the output.

- cWAITMATW(scalar): Wait for Matrices to be Written: Instruction for the Controller that tells it to perform no operation until the Data Transfer Engine confirms that a number of *scalar* matrices needed for the processing sequence are transferred into the Array's data memory.

- cRESREADY: Result Ready: Instruction for the Controller to acknowledge the Data Transfer Engine that a processing sequence is finished and the result is ready to be read.

3.3.1 The Architecture of the Data Transfer Engine

The Data Transfer Engine will take over the responsibility of I/O data transfers to and from the Array's internal data memory and also ensure that the transfers are synchronized with the processing sequences for a correct operation of the Accelerator.

The module interface is composed of signals through which the Data Transfer Engine establishes connections with the other components of the Accelerator and with the rest of the heterogeneous computing system. The structure of the Accelerator integrating the Data Transfer Engine is presented in Figure 3.6.

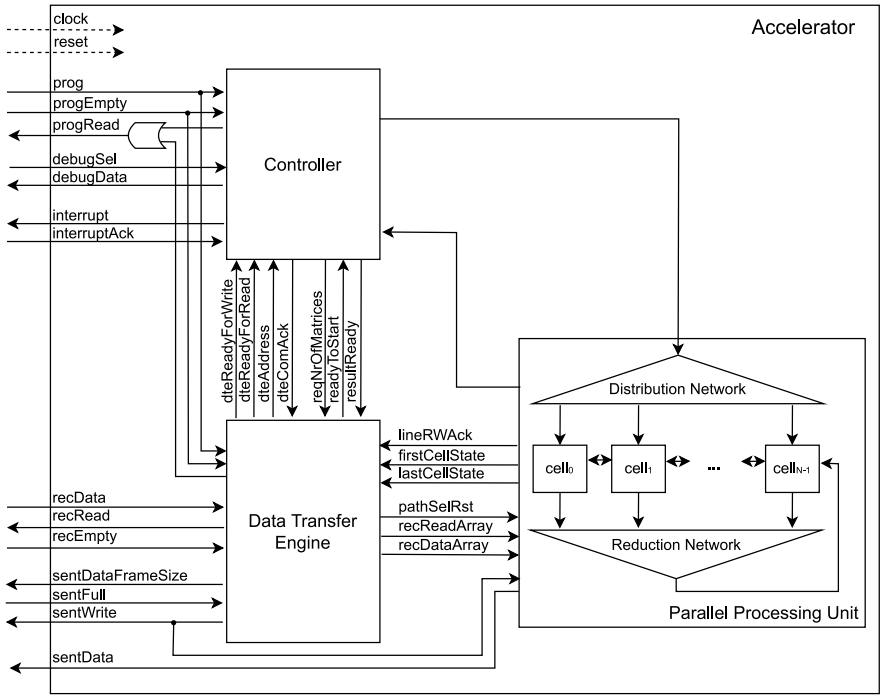

Fig. 3.6 The structure of the Accelerator integrating the Data Transfer Engine [22]

The internal structure of the Data Transfer Engine is presented in Figure 3.7. The core of the transfer engine is a finite state machine that coordinates and synchronizes the operation of all the other components.

The Command FIFO is a First-In, First-Out memory structure used to store the commands and their parameters. The write and read of commands and parameters are controlled by the Parameter Write Counter and the Parameter Read Counter.

The Command Register, Address Register, Line Counter, Column Counter, and the Padding Counter are used to hold the information about the transfer until it finishes. The Ready to Start Counter is used to synchronize the start of a processing sequence in the Parallel Processing Unit with the completion of the transfer of the needed matrices in the internal memory. The Result Ready Counter is used to synchronize the end of a processing sequence with the reading of the resulting data.

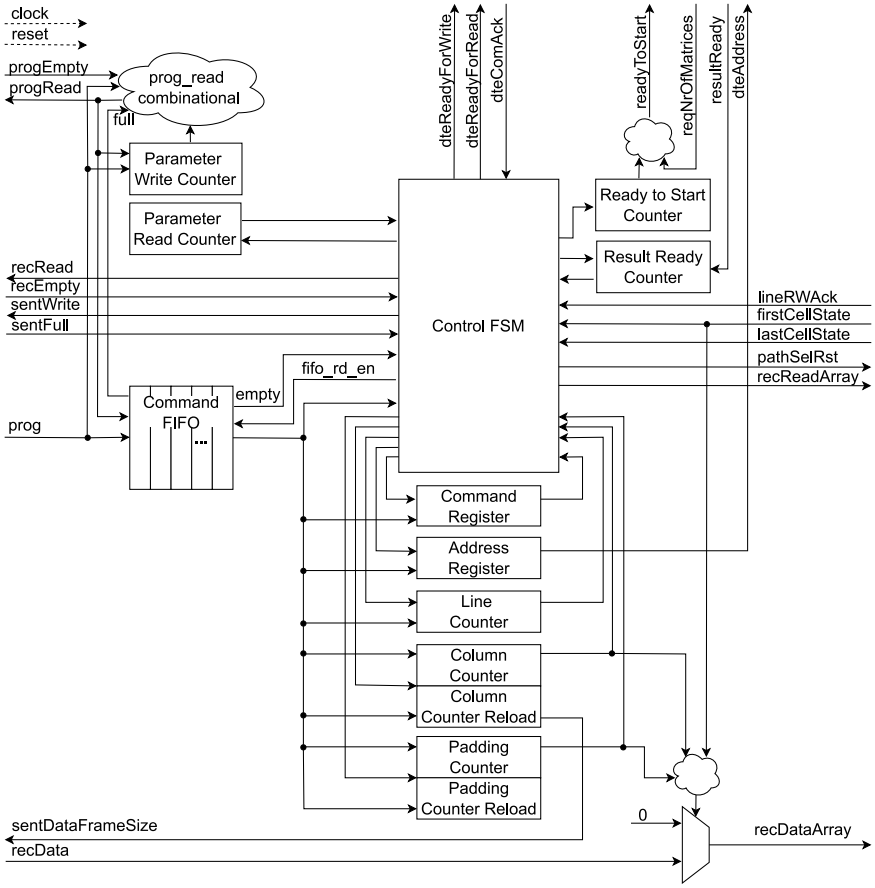

Fig. 3.7 The architecture of the Data Transfer Engine

3.3.2 Transferring Data Using the Data Transfer Engine

Some additional changes have been made in the Controller and Parallel Processing Unit for the Data Transfer Engine to operate properly.

To be able to manage the internal memory access requests coming from the two sources (the instructions sent by the Controller to be executed by the Array and the requests from the Data Transfer Engine), arbitration logic was added in the connector section of the Controller. Its architectural details are presented in Figure 3.9.

Because the data and command packets sent by the Controller to all the cells in the Array through the Distribution Network have separate fields for the information involved in internal memory transfers, the decision to approve the requests from the Data Transfer Engine is taken by testing these fields. If the current packet does not contain memory accesses but there are write or read requests from the Data Transfer Engine, the decision to approve them is made, and an acknowledgement is sent to the Data Transfer Engine.

The logic dealing with internal memory data input and output was modified to allow a direct connection between the I/O register and the memory. In this way, the data no longer has to pass through the Accumulator, allowing it to be used for other instructions while a data line is being written to or read from the Array's internal memory.

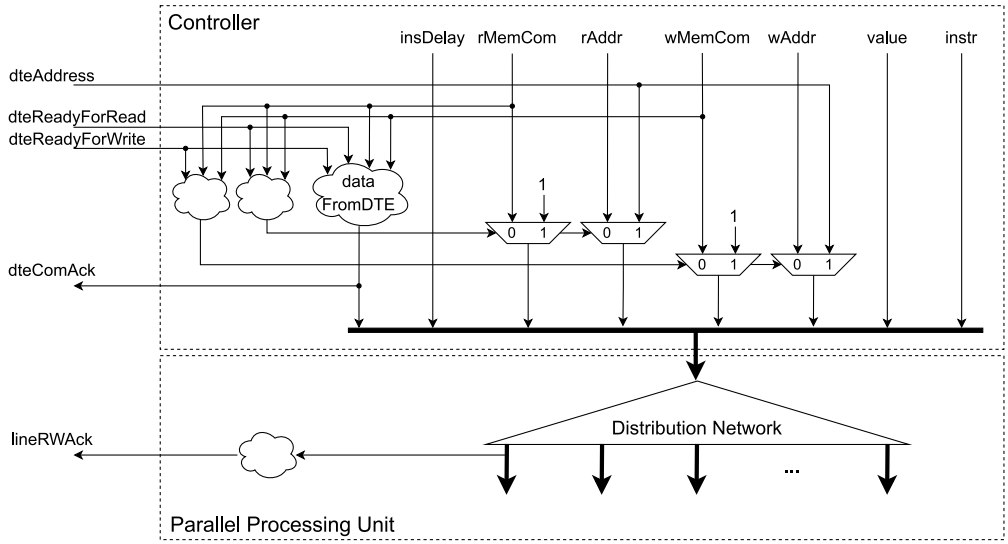

Fig. 3.9 Architecture detail of the Array's internal memory access arbitration logic

For starting a data transfer between the Data FIFOs and the data memories in the Array, the Host must transmit on the Program and Control Path a TINRUN or a TOUTRUN command, followed by its parameters. Both the commands and their parameters occupy all 32 bits available on the Program and Control Path.

The Data Transfer Engine monitors the Program FIFO output and, when a specific command is detected, it saves it to the Command FIFO if it is not *full*. Once reaching the output of the Command FIFO, if the Data Transfer Engine is in the IDLE state (no transfer is in progress), the command and its parameters are read, updating the Command Register, Address Register, Line Counter, Column Counter and its reload register, and the Padding Counter and its reload register.

In the case of a writing transfer, if the data is available in the Data Input FIFO, the Data Transfer Engine starts to generate read commands for the Data Input FIFO and shift commands for the chain of I/O registers in the Array. The shift commands will also be generated for the additional padding zeros when the number of columns is smaller than the number of cells in the Array. When a line is completely shifted through the chain of I/O registers and is in its final position, the Data Transfer Engine sends a write memory request to the Controller. If the Data Transfer Engine request is approved, it waits for the write to be performed and then starts to transfer the next matrix line. The process is repeated until all the lines are transferred to the Array's internal data memory.

In the case of a reading transfer, the Data Transfer Engine sends a read request to the Controller. If the request is approved, it waits for the read to be performed and then shifts the data to the Data Output FIFO. When the line is completely shifted, a new read request is sent, and the process is repeated until all the matrix elements are read from the Array's internal memory. An additional command, WAITRESREADY, keeps the Data Transfer Engine in the IDLE state until the Controller marks a result as ready. This mechanism helps to wait for the resulting matrix to be ready before starting to read it.

Chapter 4

Python-Based Programming Environment

The previously presented heterogeneous computing system was implemented on a Zynq-7020 SoC, integrated on a PYNQ-Z2 board. In order to be able to interact with it, a Python-based programming environment was designed and implemented [27], based on the PYNQ software package [28], preinstalled on the SoC.

Several similar programming environments for other MapReduce architectures have been developed in the past [29–31].

The programming environment runs on the Host and enables access to the resources of the system, managing the program and I/O data transfers between the Host System and the MapReduce Accelerator.

It is based on multiple Python classes, organized in such a way to allow fast adaptation when the target architecture is changed: Instruction class, used to generate the binary code of the low-level instructions targeting the Accelerator, Kernel class, used to generate the binary code of kernels (for the current environment, a kernel is defined as a group of low-level instructions that perform a certain task), Library class, used to generate the binary code of a library, which is a collection of kernels, and Machine class, used to configure the hardware and interact with the resources of the Accelerator.

The structure of the programming environment with its main classes is presented in Figure 4.2.

The use of the programming environment has two phases that correspond to the way the MapReduce Accelerator operates:

- Preparation phase: In this phase, the FPGA is configured with the bitstream corresponding to the implemented heterogeneous system. Furthermore, other elements that enable access to the Accelerator's resources are configured now: the base address of the Program and Control Path and the addresses of all the control and status registers, the names of the registers, the data dimensions, and the DMA and its communication channels.

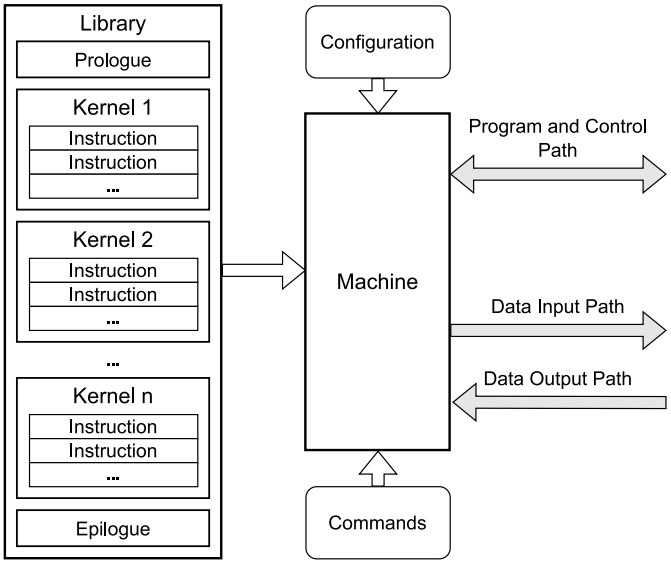

Fig. 4.2 The structure of the Python-based programming environment

- Run-time phase: Before starting to send commands to the Accelerator, a library of kernels must be loaded into the Controller's program memory. After this, the Host will interact with the Accelerator by sending on the Program and Control Path kernel call commands, or I/O data transfer commands that will start the reading and writing of data on the Data Input and Data Output Paths.

4.1 The Instruction Element

The instructions are the basic elements of the programming environment. An instruction can target either the Controller, or the Parallel Processing Unit. Because each time the Accelerator is in the running state it needs two instructions, one for each of the two modules, they must be assembled in pairs of two. Exceptions to this rule are the jump and program call instructions, for which, in order to increase the target memory range, it was decided to use for the address all the available bits in the Program and Control Path.

4.2 The Kernel Element

For the MapReduce Accelerator, a kernel represents a sequence of low-level Accelerator instructions used to perform a certain task. In other words, a kernel is a function written in a low-level custom language.

Because the Accelerator needs two instructions at each execution step, one for the Controller and one for the Parallel Processing Unit, the kernel groups them in pairs of two, forming a 32-bit Accelerator instruction. The exception is represented by the 32-bit instructions targeting the Controller, which are transmitted alone. For these instructions, the Controller internally generates NOP instructions for the Parallel Processing Unit.

A typical kernel structure is presented in Figure 4.4.

```
        cINSTR   aINSTR
        cINSTR   aINSTR
LB(n)   cINSTR   aINSTR
        cINSTR   aINSTR
        ...
        cINSTR   aINSTR
        JMP(n)
        HALT
```

Fig. 4.4 The typical structure of a kernel

4.3 The Library Element

A library is a collection of kernels loaded together into the program memory of the Accelerator. When constructing the library, the kernels are extracted from a previously written library file and packed together.

The Library class object is constructed based on the name of the library file, the starting address, and the list of kernels to be included in the resulting binary code. In addition to the user-defined kernels, each library has two predefined kernels called the Prologue and the Epilogue, which will be placed at the beginning and end of the library, respectively.

4.4 The Machine Element

The Machine class is the element of the Python-based programming environment that ensures interaction with the heterogeneous computing system. Its implementation is dependent on the architecture of the system and the resources provided by PYNQ [28].

The Machine class uses a configuration Python dictionary to extract the needed information and configure the environment. The Machine class resources mapping the hardware resources of the heterogeneous computing system are presented in Figure 4.6.

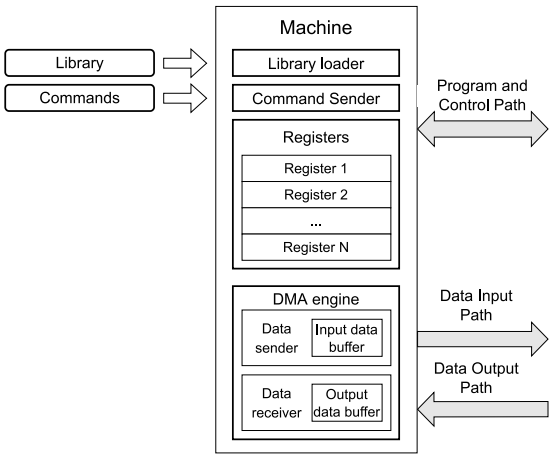

Fig. 4.6 Machine class resources

Chapter 5

Evaluation of System Performance

5.1 Basic Linear Algebra Library

In order to test both the correct functionality of the Accelerator and its performances, a basic linear algebra library was implemented. The library contains mathematical functions for matrices with a number of columns smaller or equal to the number of cells in the Array and other functions used to control the resources of the MapReduce Accelerator [22]:

- Start cycle counter
  START_CC(): starts the cycle counter in the Controller
- Stop cycle counter
  STOP_CC(): stops the cycle counter in the Controller.
- Send interrupt
  SEND_INT(): sets the Accelerator's interrupt flag. This can be used to signal the end of a task.
- Matrix-Matrix element-wise operation
  MM_EWO(*destination, source1, source2, linesNr, operation, waitMatricesNr*): performs an element-wise operation (ADD, SUB, MULT, AND, OR and XOR).
- Scalar-Matrix multiplication
  SM_MULT(*destination, scalar, source, linesNr, waitMatricesNr*): multiplies *scalar* with each element of matrix *source*.
- Matrix-Matrix multiplication
  MM_MULT(*destination, source1, source2, linesNr, waitMatricesNr*): multiplies matrix *source1* with *source2*, considering the latest already transposed.
- Matrix-Matrix multiplication and accumulation
  MM_MAC(*destination, source1, source2, linesNr, waitMatricesNr*): multiplies

matrices *source1* and *source2*, considering the latest already transposed, and accumulates them with *destination*.

In addition to the library of kernels, three commands used in I/O data transfer between the Data Input and Data Output FIFOs and the Parallel Processing Unit were implemented: WRITE_MATRIX, READ_MATRIX and WAIT_RES_READY.

5.2 Evaluation Algorithms

In order to evaluate the performance of the Accelerator and the improvements brought by the Data Transfer Engine, four algorithms were proposed to extend the linear algebra operations described above for large matrices. A large matrix is considered to be a matrix whose dimensions are larger than the number of cells in the Array (i.e., N).

The algorithms are based on the fact that a large matrix can be divided into smaller matrices that can be processed, thereby obtaining parts of the resulting large matrix that will be used to compose the final result. For simplicity, the algorithms consider all the large matrices (A, B, and R) to be squared, with dimensions equal to $2^x \cdot N$, where N is the number of cells and x is an integer number. In this particular case, a large matrix can be divided into n^2 $N \times N$ matrices, where $n = 2^x$.

- **Matrix-Matrix element-wise operations for large matrices**

The element-wise operation for large matrices is defined by (5.5) and (5.6):

$$R_{ij} = A_{ij} o B_{ij}, \qquad i = 1...n,\ j = 1...n \tag{5.5}$$

$$\begin{bmatrix} R_{11} & R_{12} & ... & R_{1n} \\ R_{21} & R_{22} & ... & R_{2n} \\ ... & ... & ... & ... \\ R_{n1} & R_{n2} & ... & R_{nn} \end{bmatrix} = \begin{bmatrix} A_{11} o B_{11} & A_{12} o B_{12} & ... & A_{1n} o B_{1n} \\ A_{21} o B_{21} & A_{22} o B_{22} & ... & A_{2n} o B_{2n} \\ ... & ... & ... & ... \\ A_{n1} o B_{n1} & A_{n2} o B_{n2} & ... & A_{nn} o B_{nn} \end{bmatrix} \tag{5.6}$$

, where R_{ij}, A_{ij}, and B_{ij} are $N \times N$ matrices, components of large matrices R, A, and B.

- **Scalar-Matrix multiplication for large matrices**

The Scalar-Matrix multiplication for large matrices is defined by (5.7) and (5.8):

$$R_{ij} = s \cdot A_{ij}, \qquad i = 1...n,\ j = 1...n \tag{5.7}$$

$$\begin{bmatrix} R_{11} & R_{12} & ... & R_{1n} \\ R_{21} & R_{22} & ... & R_{2n} \\ ... & ... & ... & ... \\ R_{n1} & R_{n2} & ... & R_{nn} \end{bmatrix} = \begin{bmatrix} s \cdot A_{11} & s \cdot A_{12} & ... & s \cdot A_{1n} \\ s \cdot A_{21} & s \cdot A_{22} & ... & s \cdot A_{2n} \\ ... & ... & ... & ... \\ s \cdot A_{n1} & s \cdot A_{n2} & ... & s \cdot A_{nn} \end{bmatrix} \tag{5.8}$$

, where R_{ij} and A_{ij} are $N \times N$ matrices, components of large matrices R and A.

- **Matrix-Matrix multiplication for large matrices**

The Matrix-Matrix multiplication for large matrices is defined by (5.9) and (5.10). For the current algorithm, it is considered that the transpose of matrix B (B^t) has already been computed.

$$R_{ik} = \sum_{j=1}^{n} A_{ij} \times B^t_{kj}, \qquad i = 1...n, k = 1...n \tag{5.9}$$

$$\begin{bmatrix} R_{11} & ... \\ R_{21} & ... \\ ... & ... \\ R_{n1} & ... \end{bmatrix} = \begin{bmatrix} A_{11} \times B^t_{11} + A_{12} \times B^t_{12} + ... + A_{1n} \times B^t_{1n} & ... \\ A_{21} \times B^t_{11} + A_{22} \times B^t_{12} + ... + A_{2n} \times B^t_{1n} & ... \\ ... & ... \\ A_{n1} \times B^t_{11} + A_{n2} \times B^t_{12} + ... + A_{nn} \times B^t_{1n} & ... \end{bmatrix} \tag{5.10}$$

, where R_{ij}, A_{ij}, and B^t_{ij} are $N \times N$ matrices, components of the large matrices R, A, and B^t. B^t is the transpose of matrix B.

- **Matrix-Matrix multiplication and accumulation for large matrices**

The Matrix-Matrix multiplication and accumulation for large matrices is defined by (5.11) and (5.12). For the current algorithm, it is considered that the transpose of matrix B (B^t) has already been computed.

$$R_{ik} = R_{ik} + \sum_{j=1}^{n} A_{ij} \times B^t_{kj}, \qquad i = 1...n, k = 1...n \tag{5.11}$$

$$\begin{bmatrix} R_{11} & ... \\ R_{21} & ... \\ ... & ... \\ R_{n1} & ... \end{bmatrix} = \begin{bmatrix} R_{11} + A_{11} \times B^t_{11} + A_{12} \times B^t_{12} + ... + A_{1n} \times B^t_{1n} & ... \\ R_{21} + A_{21} \times B^t_{11} + A_{22} \times B^t_{12} + ... + A_{2n} \times B^t_{1n} & ... \\ ... & ... \\ R_{n1} + A_{n1} \times B^t_{11} + A_{n2} \times B^t_{12} + ... + A_{nn} \times B^t_{1n} & ... \end{bmatrix} \tag{5.12}$$

, where R_{ij}, A_{ij}, and B^t_{ij} are $N \times N$ matrices, components of the large matrices R, A, and B^t. B^t is the transpose of matrix B.

5.3 Performance Figures

The performance of the Accelerator was analyzed both from the perspective of execution time and consumed power. For this, test environments were created for simulation and hardware measurements. The simulation environment was used to evaluate the execution time in terms of clock cycles. The hardware environment was used to test the correctness of the system's operation and also its power consumption.

In order to highlight the impact on the execution time and power consumption of the improvements brought to the I/O data transfer, three versions of the Accelerator were analyzed, each with their own simulation and hardware setups:

- MRA V0: The initial version of the MapReduce Accelerator, without any improvements (as it is described in Chapter 2). In this design version, the Controller is responsible for the I/O data transfers and the cells are organized on a single propagation path.
- MRA V1: An improved version of the MapReduce Accelerator having the cells reorganized in order to mitigate the delay introduced by the two-step propagation. In this version, the Controller is still responsible for the I/O data transfers.
- MRA V2: An improved version of the MapReduce Accelerator, in which both improvements described in Chapter 3 have been implemented: the cells are reorganized as in MRA V1, and the Data Transfer Engine is responsible for the I/O data transfers.

5.3.1 Execution Time Analysis

In order to analyze the variation of the execution time depending on the computing resources of the Accelerator for the algorithms proposed in the previous section, but also to test the correctness of its operation, a simulation environment capable of executing several test scenarios was created.

Two test scenarios were used to evaluate the variation of the execution time depending on the available resources and the required calculation volume, which is related to the size of the matrices. In the first scenario, a matrix operation on 16×16, 32×32, 64×64, and 128×128 matrices was executed on a MapReduce Acelerator with 16 processing cells. This test scenario will show how the execution time evolves as the calculation volume increases by increasing the size of the matrices, if the same computation resources are used. In the second test scenario, a matrix operation on 128×128 matrices is executed on MapReduce Accelerators with 16, 32, 64, and 128 processing cells. This test scenario will show how the execution time evolves as the computation resources increase, if the calculation volume is constant.

The two test scenarios were applied to Matrix-Matrix element-wise addition (Matrix-Matrix ADD), Scalar-Matrix multiplication (Scalar-Matrix MULT), Matrix-Matrix multiplication (Matrix-Matrix MULT), and Matrix-Matrix multiplication and accumulation (Matrix-Matrix MAC) operations, executed on all the three previously presented versions of the MapReduce Accelerator.

In order to have a clear picture of the Accelerator's performance, the starting point for measuring the number of clock cycles was considered to be the moment when the library is already in the program memory and the first data reaches the input of the Accelerator.

The results obtained for the Matrix-Matrix ADD operation for the two test scenarios are presented in Figure 5.7 and Figure 5.8.

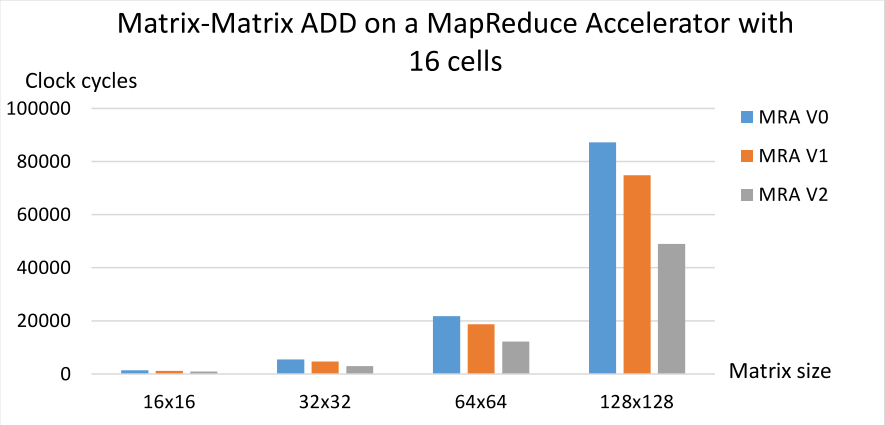

Fig. 5.7 Matrix-Matrix ADD on a MapReduce Accelerator with 16 processing cells

In Figure 5.7, it can be observed that the execution time is proportional to the number of elements in the matrices for all the three design versions: it increases by 4× when the number of elements increases by 4×. Furthermore, it can be observed that the best improvement is brought by the version that contains the Data Transfer Engine (MRA V2), for which the execution time decreases by approximately 44% for all operations with large matrices included in the test scenario.

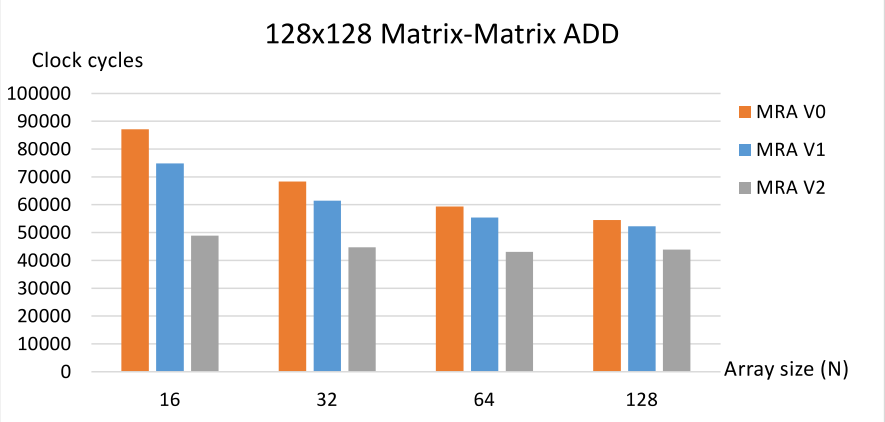

Fig. 5.8 128×128 Matrix-Matrix ADD on MapReduce Accelerators of various sizes

In Figure 5.8, the influence of the computation resources on the operation of adding two 128×128 matrices can be observed. This test scenario also reveals the improvement brought by the presence of the Data Transfer Engine on the total execution time, this being higher as the computing resources decrease: approximately a 43% improvement when adding two 128×128 matrices on an Accelerator with 16 processing cells. Furthermore, it can be observed that the presence of the Data Transfer Engine considerably reduces the difference between the execution times of the same operation on Accelerators with different numbers of processing cells.

The results obtained for the Scalar-Matrix MULT operation for the two test scenarios are presented in Figure 5.9 and Figure 5.10. For this operation, only one matrix must be transferred from the main memory to the Accelerator.

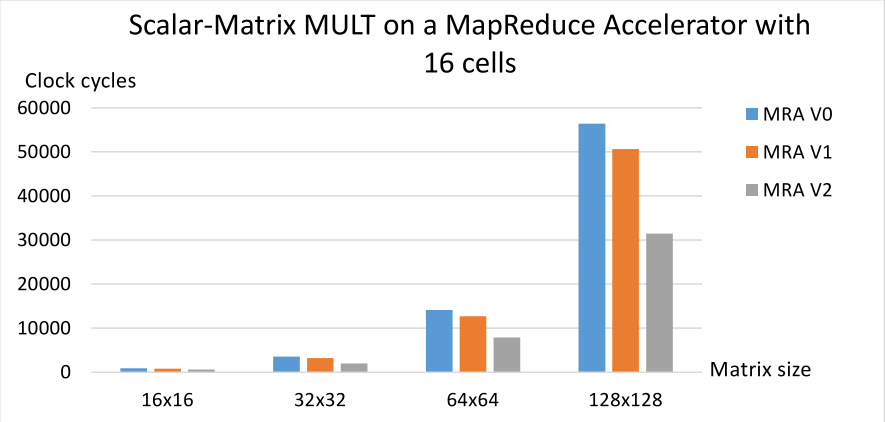

Fig. 5.9 Scalar-Matrix MULT on a MapReduce Accelerator with 16 processing cells

In Figure 5.9, it can be observed that, as in the case of the previous operation, the best improvement is brought by the version that contains the Data Transfer Engine (MRA V2), for which the execution time decreases by approximately 44% for all operations with large matrices included in the test scenario. However, since the operation requires only one matrix as an operand, the total execution time is smaller than the one obtained for the Matrix-Matrix ADD operation (e.g., the execution time is smaller by 35% when using MRA V2, compared to the Matrix-Matrix ADD operation).

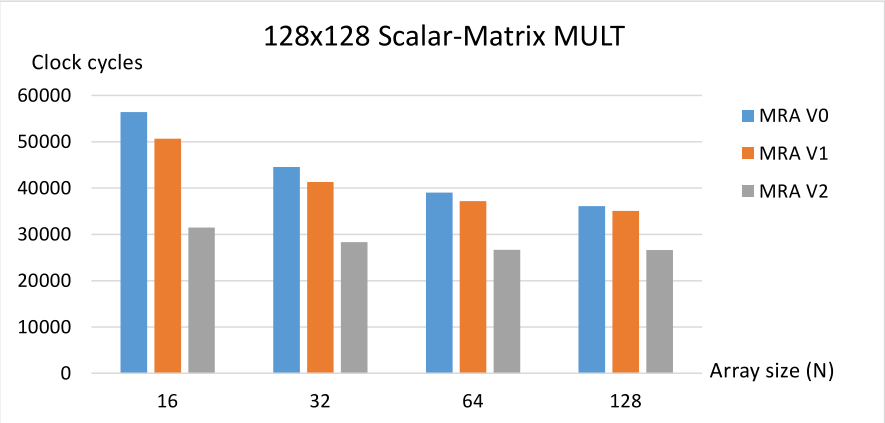

Fig. 5.10 128x128 Scalar-Matrix MULT on MapReduce Accelerators of various sizes

In Figure 5.10, the influence of the computation resources on the operation of multiplying each element of a 128×128 matrix with a scalar can be observed. As in the case of the Matrix-Matrix ADD operation, the improvement brought by the presence of the Data Transfer Engine combined with the reorganization of cells on the total execution time is higher as the computing resources decrease: approximately a 44% improvement when performing the operation on an Accelerator with 16 processing cells. Furthermore, it can also be observed that the presence of the Data Transfer Engine considerably reduces the difference between the execution times of the same operation on Accelerators with different numbers of processing cells, mitigating the impact on execution time when the computational resources are reduced.

The results obtained for the Matrix-Matrix MULT operation for the two test scenarios are presented in Figure 5.11 and Figure 5.12. Compared to the previously analyzed operations, an important increase in execution time is observed. This is explained by the significant increase in the number of I/O data transfers, determined by the fact that an $N \times N$ matrix must be transferred several times from the main memory to the internal memory of the Array during the algorithm execution.

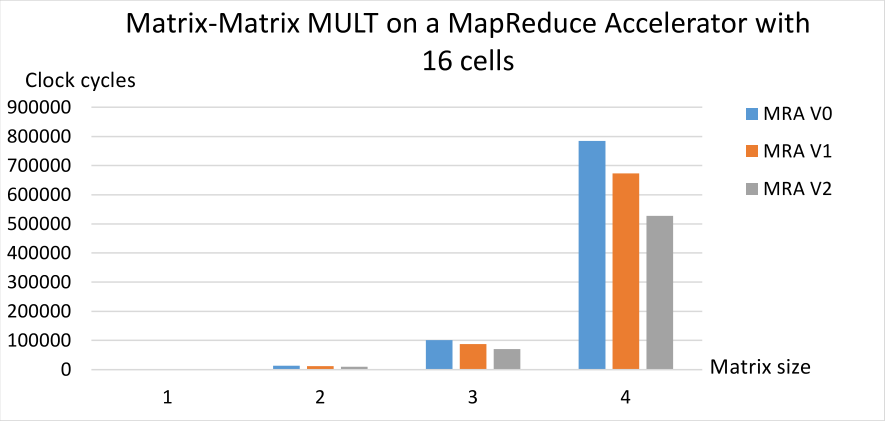

Fig. 5.11 Matrix-Matrix MULT on a MapReduce Accelerator with 16 processing cells

In Figure 5.11, it can be observed that the improvement in execution time increases with the increase in the number of elements in the matrix. The higher improvement (of about 32%) for the MapReduce Accelerator is obtained when multiplying two 128×128 matrices, therefore, when the number of required data transfers is the highest. The lower value of the execution time improvement compared to the ones obtained for the previously tested algorithms can be explained by the fact that, during the multiplication of two matrices, the internal memory is accessed more often and the requests with a lower priority from the Data Transfer Engine are accepted with a delay.

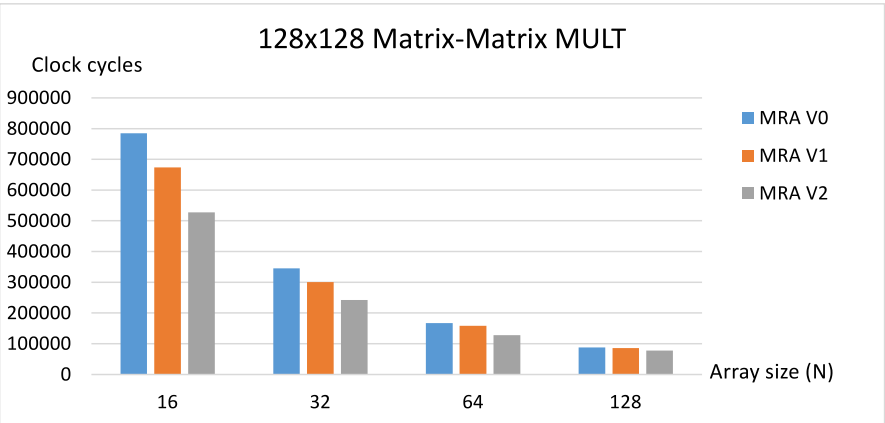

Fig. 5.12 128x128 Matrix-Matrix MULT on MapReduce Accelerators of various sizes

The influence of the computation resources on the operation of multiplying two 128×128 matrices is presented in Figure 5.12. The MRA V2 design version shows improvements in the execution times for all variations of the test scenario, the highest

being obtained when multiplying the matrices on a MapReduce Accelerator with 16 processing cells: approximately 32%. Nevertheless, the results suggest that the growth rate of the improvement tends to decrease as the computational resources decrease.

The results obtained for the Matrix-Matrix MAC operation for the two test scenarios are presented in Figure 5.13 and Figure 5.14. Compared to the multiplication operation, a slight increase in the execution time can be observed.

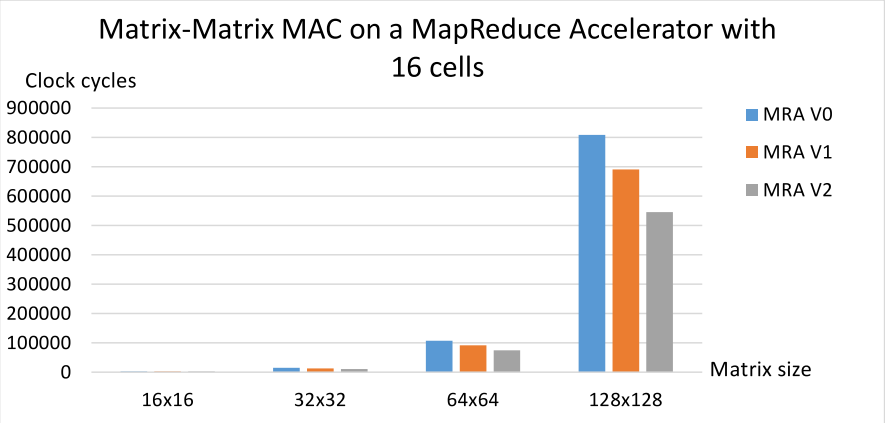

Fig. 5.13 Matrix-Matrix MAC on a MapReduce Accelerator with 16 processing cells

In Figure 5.13, it can be observed that the evolution of the execution time depending on the size of the matrices is similar to that obtained in the case of multiplication. As in the previous case, the highest improvement (of about 32%) is obtained when multiplying and accumulating two 128×128 matrices. From this, it can be concluded that the improvement increases as the number of I/O data transfers between the main memory and the Array increases.

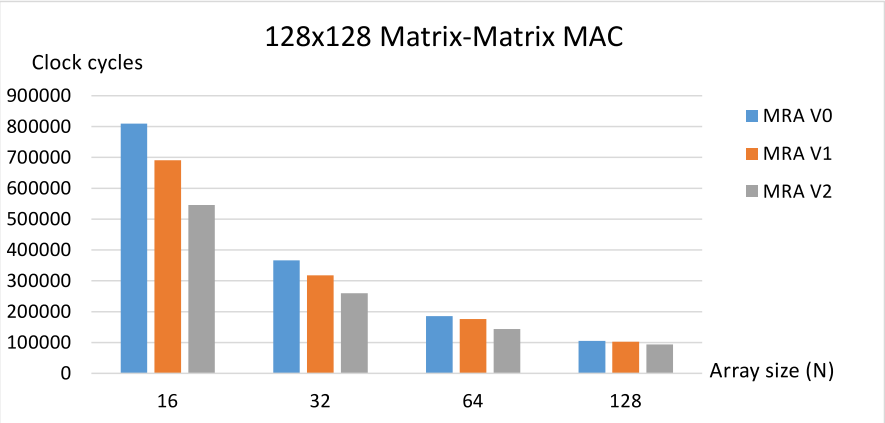

Fig. 5.14 128x128 Matrix-Matrix MAC on MapReduce Accelerators of various sizes

The influence of the computation resources on the operation of multiplying and accumulating two 128×128 matrices is presented in Figure 5.14. The results confirm the positive influence of the Data Transfer Engine on the total execution time but also the decrease in the improvement rate with the increase in the volume of data transfers between the main memory and the Array.

5.3.2 Hardware Implementation Analysis

In addition to the evaluations made using the simulation environment, a hardware test setup was implemented to test the correctness of the operation of the system and also to highlight the impact of the implementation platform (i.e., the PYNQ-Z2 board and the PYNQ software package) on the execution time. Due to the limitations given by the resources of the FPGA integrated in the ZYNQ-7020 SoC, Accelerators with 16, 32, and 64 processing cells could be synthesized, and 64×64 matrices were chosen for testing.

The correctness of operation was tested by comparing the results returned by the Accelerator with those calculated on Host using the Python *numpy* library.

In order to highlight the influence of the system, the execution times of two large matrix processing algorithms were measured for each of the three versions of the system design (MRA V0, MRA V1, and MRA V2). The obtained results were similar for the same combination of matrix sizes and number of cells for all three versions of the MapReduce Accelerator. This is explained by the large delays introduced by the interpretation of the code written in Python and by accessing the resources from the classes of the programming environment.

The current intensity and voltage were measured during the execution of the two algorithms on all the three design versions, but no notable differences were observed. This can be explained by the significant weight that the static power consumption of the PYNQ-Z2 board has in the total power consumption, thus masking the differences given by the variation of the design implemented on the FPGA. Using the execution time obtained in simulation and the currents and voltages measured on hardware, an estimate of energy consumption was computed for MRA_V2 of various sizes (Table 5.10).

Table 5.10 Estimation of energy consumed by the Accelerator for adding and multiplying 64×64 matrices

Accelerator	**Matrix-Matrix ADD**				**Matrix-Matrix MULT**			
	U [V]	**I [A]**	Δt_{sim} **[μs]**	**E [μJ]**	**U [V]**	**I [A]**	Δt_{sim} **[μs]**	**E [μJ]**
MRA V2 16	4.96	0.35	122	212	4.96	0.36	702	1253
MRA V2 32	4.96	0.37	112	206	4.96	0.39	344	665
MRA V2 64	4.96	0.37	116	213	4.96	0.39	200	387

As shown in the previous table, the biggest influence on the energy consumed by the current setup is the execution time, which means that the smaller the size of the Accelerator and the longer the operation with large matrices takes, the more energy is consumed. However, using the current setup, with many other elements that consume power, a definitive conclusion cannot be drawn on the influence of the Accelerator size on the total consumed energy. However, it is expected that the implementation of the heterogeneous computing system on silicon using an advanced technological node will result in much lower energy consumption.

Chapter 6

Conclusions

6.1 Objectives and Results

The research was dedicated to the development of a heterogeneous computing system capable of efficient parallel data processing. For the development of a fully functional system, three main elements were considered: the design and implementation of a new architecture for a MapReduce Accelerator having a much more efficient I/O data transfer, the design of the architecture of a heterogeneous computing system that integrates the Accelerator and its implementation on the support platform, and the development of both simulation and hardware programming environments, used to interact with the system and evaluate its performance.

The development of a new MapReduce architecture was made on the basis of an already existing Accelerator, which was analyzed to highlight its weak points that have a negative impact on the total execution time of a program sequence. Following this analysis, it was found that the I/O data transfer represents an important part of the total execution time, thus becoming the target of the improvement efforts. The principle that is the basis of the new architecture is the separation of the I/O data transfer and processing flows, so that the data transfers required by future operations can be carried out in advance, in parallel with the current data processing sequence. Furthermore, the processing cells in the unit responsible for parallel data processing have been reorganized so that the Accelerator can accept a new input data at each clock cycle.

Thus, two new architectures were implemented, each one progressively integrating the proposed improvement measures.

For performance evaluation, a simulation environment was developed, allowing the user to write programs in the assembly language specific to the Accelerator, interact with it, and evaluate the returned results.

In order to highlight the performance improvement due to the improved I/O data transfer, four evaluation algorithms for large matrices were proposed: Matrix-Matrix ADD, Scalar-Matrix MULT, Matrix-Matrix MULT, and Matrix-Matrix MAC. When

operating with large matrices, the number of columns is greater than the number of processing cells in the unit responsible for parallel data processing, thus exceeding the Accelerator's internal data storage and computing capacity. This forces the Accelerator to perform successive I/O data transfers in order to bring parts of the operand matrices and to send back parts of the result matrix.

The tests were used to characterize the dependency of the execution time on the resources of the Accelerator and the complexity of the algorithm. The results showed an improvement in execution time for both the new architectures compared to the initial one. The higher improvement was obtained for the architecture implementing both the reorganization of processing cells and the separation of I/O data transfer and processing flows: up to a 44% improvement for Matrix-Matrix ADD and Scalar-Matrix MULT and up to a 32% improvement for Matrix-Matrix MULT and Matrix-Matrix MAC. The lower value of improvement in execution time obtained for the last two operations compared to the first two can be explained by the fact that, during the multiplication of matrices, the internal memory of the Array is accessed more often by the processing flow, and the requests from the I/O data transfer flow are accepted with a delay, thus reducing the capacity to transfer new data.

Furthermore, the test results showed that, for the new architectures, the improvement in execution time compared to the initial one is higher as the dimensions of the matrices to be processed increase compared to the number of processing cells in the Accelerator. In other words, the new architecture shows its usefulness as the volume of data to be processed by the Accelerator is higher.

To be able to use the resources of the Accelerator, it has been integrated into a heterogeneous computing system. In such a system, the main program is executed on a general-purpose processor called the Host, while the Accelerator is responsible for the execution of the intensive computation sequences. The proposed architecture of the system takes into consideration the way the Accelerator must interact with the Host and the main memory in order to function as efficiently as possible. Moreover, the system architecture was adapted to the resources offered by the platform on which it was implemented (i.e., the Zynq-7020 Soc).

A programming environment based on the Python language was developed in order to allow the user to interact with the heterogeneous computing system implemented on the Zynq-7020 SoC. This environment, like the one developed for simulation, allows the description of a library of functions in the specific assembly language, the development of programs, and their execution on the Accelerator. However, although it is user-friendly and allows fast development and testing of programs, an important delay introduced by the software layers of the environment was observed, making it not suitable for intensive use.

A power consumption measurement setup was also developed in order to make a preliminary evaluation of the energy required by the system to perform various matrix

operations. Although the influence of the number of processing cells in the Accelerator on the total amount of energy consumed during an operation with matrices could not be highlighted due to the major influence of other components of the platform on which the heterogeneous computing system was implemented, an initial assessment of the total required energy could be made.

As a general conclusion, a fully functional heterogeneous computing system has been developed and improved, which, subject to further improvement and implemented on a suitable platform, can offer good performance, becoming competitive with the solutions already existing on the market.

6.2 Original Contributions

The major original contributions of this research can be classified as follows, based on the three main components of the developed system:

The architecture of the MapReduce Accelerator

- The analysis of the architecture of an existing MapReduce Accelerator and the proposal of a new architecture to improve the total execution time. The decisions regarding the elements that need to be improved were taken based on the literature study and the test setup, through which the executions of several algorithms were analyzed in order to highlight the weak points of the Accelerator.
- The design and implementation of a new way of organizing the processing cells in the Accelerator so that the propagation of I/O data is made more efficient without changing the functional behavior (this change is completely transparent to the user).
- The design and implementation of a control module called the Data Transfer Engine, which is responsible for coordinating I/O data transfers. This, together with the separation of the resources used in I/O data transfers, led to the separation of I/O data transfer and processing flows, which can now work independently.
- The design and implementation of a synchronization procedure between the I/O data transfer and processing flows.
- The design and implementation of structures responsible for arbitrating the requests for accessing the Array's internal memory coming from the two flows.

The architecture of the system

- The design of the architecture of a heterogeneous computing system that integrates the MapReduce Accelerator and its implementation considering the resources available on the Zynq-7020 SoC so that the interaction between its components is as efficient as possible.
- The design and implementation of a structure to interface the MapReduce Accelerator with the rest of the system, through which it can be configured and operated.

The software

- The implementation of a basic linear algebra library for operations with matrices using the Accelerator's specific assembly language.
- The proposal of a form of segmentation for the Array's internal memory so that the data transfer capabilities of the Data Transfer Engine are used as efficiently as possible and the adaptation of the algorithms for large matrices operations to the specific resources of the Accelerator.
- The implementation of a Python-based programming environment that supports the development and execution of software programs specific to the heterogeneous computing system.

6.3 List of Original Publications

1. **George-Vlăduț Popescu**, Improvements in Data Transfer for a MapReduce Accelerator, *Romanian Journal of Information Science and Technology*, 25(3-4), 2022 [22].
2. **George-Vlăduț Popescu** and Călin Bîră, Python-Based Programming Framework for a Heterogeneous MapReduce Architecture, *2022 14th International Conference on Communications (COMM)*, Bucharest, June 2022, pp. 1-6, DOI: 10.1109/COMM54429.2022.9817183 [27].
3. Mihaela Maliţa, **George-Vlăduţ Popescu**, Gheorghe M. Ştefan, Pseudo Reconfigurable Heterogeneous Solution for Accelerating Spectral Clustering, *2020 IEEE International Conference on Big Data (Big Data)*, December 2020, pp. 5138-5145, DOI: 10.1109/BigData50022.2020.9378150, WOS:000662554705026 [24].
4. Mihaela Maliţa, **George-Vlăduţ Popescu**, Gheorghe M. Ştefan, Heterogeneous Computing System for Deep Learning. In *Deep Learning: Concepts and Architectures*, Pedrycz, W., Chen, S.M., Eds.; Springer International Publishing: Cham,

Switzerland, 2020; pp. 287–319, ISBN: 978-3-030-31756-0, DOI: 10.1007/978-3-030-31756-0_10 [16].

5. Mihaela Maliţa, **George-Vlăduţ Popescu**, Gheorghe M. Ştefan, Heterogenous Computing for Markov Models in Big Data, *2019 International Conference on Computational Science and Computational Intelligence (CSCI)*, Las Vegas, NV, USA, Decembrie 2019, pp. 1500-1505, DOI: 10.1109/CSCI49370.2019.00279, WOS: 000569996300272 [23].

6. Alexandru Gheolbănoiu, **George-Vlăduț Popescu**, Radu Hobincu, Lucian Petrică, A Software-Defined FPGA Vector Processor with Application-Aware Reconfiguration, *University Politehnica of Bucharest Scientific Bulletin, Series C: Electrical Engineering and Computer Science*, 78(4), pp.43-56, 2016, ISSN 2286-3540, WOS:000393328400004 [18].

7. Mihaela Malița, **George Vlăduț-Popescu**, Gheorghe M. Ștefan, Hybrid System for Deep Learning, *ICON4N 2018: 1st International Conference on Neuroscience, Neuroinformatics, Neurotechnology and Neuro-Psycho-Pharmacology*, Bucharest, Romania, Nov 15-18, 2018, [unpublished].

8. **George Vlăduț-Popescu**, State of the Art in sound source recognition using neural networks, Technical Report No. 1, University Politehnica of Bucharest, 2018.

9. **George Vlăduț-Popescu**, Computational components of Deep Neural Networks used in sound space investigation, Technical Report No. 2, University Politehnica of Bucharest, 2018.

10. **George Vlăduț-Popescu**, Computer architectures for Deep Neural Networks, Technical Report No. 3, University Politehnica of Bucharest, 2019.

6.4 Perspectives for Further Developments

In order to develop the capabilities of the proposed heterogeneous computing system and to improve its performance so that it becomes competitive with the other systems currently on the market, further investigations and improvements must be made in all its main components: the architecture of the Accelerator, the architecture of the system, and the software used to access its resources.

The architecture of the MapReduce Accelerator can be further developed to improve both the computation capabilities and the I/O data transfer.

New modules that allow new operations can be added to improve computation capabilities, thereby expanding the field of target applications. For example, floating-point computation can be implemented either through specialized modules in each processing cell or through completely separate processing cells.

Additionally, I/O data transfer can be further improved by implementing design structures that allow both read and write transfers at the same time. One solution to accomplish this is to separate the input and output data flows in the Array of processing cells by implementing two chains of I/O registers with separate control sections, one for each I/O data transfer type. Another solution could be to implement read-write transfers, which first load the read data into the I/O registers and then shift it to the output while new data is shifted in at the input of the chain of I/O registers.

At the system level, several Accelerators can be added and grouped into clusters, either identical or with different computing capabilities but sharing the same data and program paths. In this way, the system's capacity to process data is considerably increased. For such a system, it is necessary to investigate the optimal number of processing cells in an Accelerator and how many Accelerators should be grouped together in order to obtain the best *number of operations / consumed power* ratio.

At the software level, although the proposed programming environment is user-friendly and offers a short development cycle for algorithms that use data organized as vectors and matrices, it introduces large delays, reducing the system's performance. Thus, an improvement to the interaction with the system could be the development of an environment that uses low-level drivers to control the resources of the heterogeneous computing system. Moreover, the current library of functions can be improved by including additional functions, allowing the Accelerator to be used for other applications as well.

References

[1] Krste Asanovic et al. *The Landscape of Parallel Computing Research: A View from Berkeley*. Tech. rep. UCB/EECS-2006-183. Electrical Engineering and Computer Sciences, University of California at Berkeley, Dec. 2006.

[2] Krste Asanovic et al. "A View of the Parallel Computing Landscape". In: *Commun. ACM* 52.10 (Oct. 2009), pp. 56–67. ISSN: 0001-0782. DOI: 10.1145/1562764.1562783.

[3] Alejandro Duran and Michael Klemm. "The Intel® Many Integrated Core Architecture". In: *2012 International Conference on High Performance Computing & Simulation (HPCS)*. 2012, pp. 365–366. DOI: 10.1109/HPCSim.2012.6266938.

[4] George Chrysos. "Intel® Xeon Phi coprocessor (codename Knights Corner)". In: *2012 IEEE Hot Chips 24 Symposium (HCS)*. 2012, pp. 1–31. DOI: 10.1109/HOTCHIPS.2012.7476487.

[5] *Intel® Xeon Phi™ Coprocessor System Software Developers Guide*. https://www.intel.com/content/dam/www/public/us/en/documents/product-briefs/xeon-phi-coprocessor-system-software-developers-guide.pdf. California, USA: Intel Corporation, 2014.

[6] Evangelos Georganas et al. "Anatomy of High-Performance Deep Learning Convolutions on SIMD Architectures". In: *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*. 2018, pp. 830–841. DOI: 10.1109/SC.2018.00069.

[7] *NVIDIA TESLA V100 GPU Architecture*. https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf. California, USA: NVIDIA Corporation, 2017.

[8] *NVIDIA TURING GPU Architecture*. https://images.nvidia.com/aem-dam/en-zz/Solutions/design-visualization/technologies/turing-architecture/NVIDIA-Turing-Architecture-Whitepaper.pdf. California, USA: NVIDIA Corporation, 2018.

[9] *NVIDIA A100 Tensor Core GPU Architecture*. https://images.nvidia.com/aem-dam/en-zz/Solutions/data-center/nvidia-ampere-architecture-whitepaper.pdf. California, USA: NVIDIA Corporation, 2020.

[10] *NVIDIA H100 Tensor Core GPU Architecture*. https://resources.nvidia.com/en-us-tensor-core. California, USA: NVIDIA Corporation, 2022.

[11] Norman P. Jouppi et al. "A Domain-Specific Supercomputer for Training Deep Neural Networks". In: *Commun. ACM* 63.7 (June 2020), pp. 67–78. ISSN: 0001-0782. DOI: 10.1145/3360307.

[12] John L. Hennessy and David A. Patterson. *Computer Architecture, Sixth Edition: A Quantitative Approach*. 6th. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2017. ISBN: 978-0-12-811905-1.

[13] Norman P. Jouppi et al. “In-Datacenter Performance Analysis of a Tensor Processing Unit”. In: *Proceedings of the 44th Annual International Symposium on Computer Architecture*. ISCA ’17. Toronto, ON, Canada: Association for Computing Machinery, 2017, pp. 1–12. ISBN: 9781450348928. DOI: 10.1145/3079856.3080246.

[14] Sharan Chetlur et al. “cuDNN: Efficient Primitives for Deep Learning”. In: *CoRR* abs/1410.0759 (2014). arXiv: 1410.0759. URL: http://arxiv.org/abs/1410.0759.

[15] Chao Li et al. “Optimizing Memory Efficiency for Deep Convolutional Neural Networks on GPUs”. In: *SC ’16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 2016, pp. 633–644. DOI: 10.1109/SC.2016.53.

[16] Mihaela Malița, George Vlăduț Popescu, and Gheorghe M. Ștefan. “Heterogeneous Computing System for Deep Learning”. In: *Deep Learning: Concepts and Architectures*. Ed. by Witold Pedrycz and Shyi-Ming Chen. Cham: Springer International Publishing, 2020, pp. 287–319. ISBN: 978-3-030-31756-0. DOI: 10.1007/978-3-030-31756-0_10.

[17] Gheorghe M. Ștefan. “Pseudo-Reconfigurable Computing”. In: *Romanian Journal of Information Science and Technology (ROMJIST)* 24.4 (2021), pp. 366–383. ISSN: 1453-8245.

[18] Alexandru Gheolbănoiu et al. “A Software-Defined FPGA Vector Processor with Application-Aware Reconfiguration”. In: *University Politehnica of Bucharest Scientific Bulletin Seris C* 78.4 (2016), pp. 43–56. ISSN: 2286-3540.

[19] Lazar Bivolarski et al. “The CA1024: A fully programmable system-on-chip for costeffective HDTV media processing”. In: *2006 IEEE Hot Chips 18 Symposium (HCS)*. 2006, pp. 1–26. DOI: 10.1109/HOTCHIPS.2006.7477854.

[20] Mihaela Malița, Gheorghe Ștefan, and Dominique Thiébaut. “Not Multi-, but Many-Core: Designing Integral Parallel Architectures for Embedded Computation”. In: *SIGARCH Comput. Archit. News* 35.5 (Dec. 2007), pp. 32–38. ISSN: 0163-5964. DOI: 10.1145/1360464.1360474.

[21] *Zynq-7000 SoC Technical Reference Manual*. https://docs.xilinx.com/v/u/en-US/ug585-Zynq-7000-TRM. California, USA: XILINX Corporation, 2021.

[22] George-Vlăduț Popescu. “Improvements in Data Transfer for a MapReduce Accelerator”. In: *Romanian Journal of Information Science and Technology (ROMJIST)* 25.3-4 (2022), pp. 368–380. ISSN: 1453-8245.

[23] Mihaela Malița, George-Vlăduț Popescu, and Gheorghe M. Ștefan. “Heterogenous Computing for Markov Models in Big Data”. In: *2019 6TH INTERNATIONAL CONFERENCE ON COMPUTATIONAL SCIENCE AND COMPUTATIONAL INTELLIGENCE (CSCI 2019)*. 2019, pp. 1500–1505. ISBN: 978-1-7281-5584-5. DOI: 10.1109/CSCI49370.2019.00279.

[24] Mihaela Malița, George-Vlăduț Popescu, and Gheorghe M. Ștefan. “Pseudo-Reconfigurable Heterogeneous Solution for Accelerating Spectral Clustering”. In: *2020 IEEE INTERNATIONAL CONFERENCE ON BIG DATA (BIG DATA)*. IEEE International Conference on Big Data. IEEE; IEEE Comp Soc; IBM; Ankura. 2020, pp. 5138–5145. ISBN: 978-1-7281-6251-5. DOI: 10.1109/BigData50022.2020.9378150.

[25] *AMBA AXI and ACE Protocol Specification*. https://developer.arm.com/documentation/ihi0022/e/AMBA-AXI3-and-AXI4-Protocol-Specification. Cambridge, UK: ARM Company, 2013.

[26] *AMBA AXI and ACE Protocol Specification*. https://developer.arm.com/documentation/ihi0051/a/Introduction/About-the-AXI4-Stream-protocol. Cambridge, UK: ARM Company, 2010.

[27] George-Vlăduţ Popescu and Călin Bîră. "Python-Based Programming Framework for a Heterogeneous MapReduce Architecture". In: *2022 14th International Conference on Communications (COMM)*. 2022, pp. 1–6. DOI: 10.1109/COMM54429.2022.9817183.

[28] Xilinx. *PYNQ: Python productivity for Xilinx platforms*. https://pynq.readthedocs.io/en/v2.7.0/index.html. Version 2.7.0.

[29] Călin Bîră, Lucian Petrică, and Radu Hobincu. "OPINCAA: A Light-Weight and Flexible Programming Environment For Parallel SIMD Accelerators". In: *Romanian Journal of Information Science and Technology (ROMJIST)* 16.4 (2013), pp. 336–350. ISSN: 1453-8245.

[30] Alexandru E. Şuşu. "A Vector-Length Agnostic Compiler for the Connex-S Accelerator with Scratchpad Memory". In: *ACM Trans. Embed. Comput. Syst.* 19.6 (Oct. 2020). ISSN: 1539-9087. DOI: 10.1145/3406536.

[31] Alexandru E. Șușu. "A C Compiler for the Wide, Low-Power CONNEX-S Vector Accelerator". In: *University Politehnica of Bucharest Scientific Bulletin Seris C* 82.2 (2020), pp. 143–156. ISSN: 2286-3540.