Technical University of Crete
School of Electrical and Computer Engineering

Mapping HPC accelerators on HARP2 platform using the DAER Decoupled Access-Execute Framework

Ioannis Morianos

Thesis Committee:
Professor Apostolos Dollas (supervisor) (ECE TUC)
Associate Professor Eftichios Koutroulis (ECE TUC)
Professor Dionisios N. Pnevmatikatos (ECE NTUA)

Chania, September 2020

Ioannis Morianos: Mapping HPC accelerators on HARP2 platform using the DAER Decoupled Access-Execute Framework

Abstract

In the latest years, the need to process large volumes of data in a short period and the need to limit the power consumption, have shifted the computing industry vendors to build High-Performance Computing (HPC) acceleration platforms. The hybrid CPU-FPGA (Field-Programmable Gate Arrays) system is one of the most promising HPC platforms because FPGAs provide reconfigurability to accelerate different applications, faster processing, and more power-efficient and lower latency service. In these platforms, the CPU and the FPGA are tightly coupled with each other and share the same DRAM for better communication. The platform used in this work is the Intel© Xeon© Scalable Platform with Integrated FPGA (HARP2 Platform).

The Decoupled Access/Execute framework is a new way of mapping algorithms efficiently on Reconfigurable platforms (DAER). This framework splits the application tasks into two parts, the data processing (Process Unit) and the data fetching (Fetch Unit). This division completely decouples access to memory by processing data, to achieve high performance by exploiting parallelism.

This Diploma Thesis aims to implement the Jacobi algorithm with the DAER framework in the HARP platform for the solution of Laplace equations. The Jacobi method belongs to the Structured Grid algorithms that fall into the list of thirteen (13) Dwarfs that represent active areas in parallel computing. In this work, two architectures have been mapped to try to exploit the advantages of the DAER framework. The experiments were conducted in the Academic Compute Environment (ACE) that is located on the vLabs of Intel. The results of those experiments show that using the DAER framework in the Hybrid CPU-FPGA platform achieves up to 2x speed-up compared to the CPU-based solution.

Keywords: DAER, Hybrid Platform, HARP, Jacobi Method, CCI-P, OPAE, AFU, ASE, HW, SW, CLs, Memory requests, Fetch Unit, Process Unit

Περίληψη

Τα τελευταία χρόνια, η ανάγκη επεξεργασίας μεγάλου όγκου δεδομένων σε σύντομο χρονικό διάστημα και η ανάγκη περιορισμού της κατανάλωσης ενέργειας, έχουν υποκινήσει τη βιομηχανία του κλάδου των υπολογιστών να δημιουργήσει πλατφόρμες επιταχυντών υψηλής απόδοσης (HPC). Τα υβριδικά συστήματα CPU-FPGA είναι από τις πιο ελπιδοφόρες πλατφόρμες υψηλής απόδοσης, επειδή οι FPGA παρέχουν δυνατότητα αναδιάταξης για την επιτάχυνση διαφορετικών εφαρμογών, ταχύτερη επεξεργασία και πιο αποδοτική και ταχύτερη μεταφορά δεδομένων. Σε αυτές τις πλατφόρμες, η CPU και η FPGA συνδέονται στενά μεταξύ τους και μοιράζονται την ίδια μνήμη DRAM επιτυγχάνοντας καλύτερη επικοινωνία. Η πλατφόρμα που χρησιμοποιείται σε αυτήν την εργασία είναι η Intel © Xeon © Scalable Platform με ενσωματωμένη FPGA (HARP2 Platform).

Η αρχιτεκτονική αποζευγμένης επεξεργασίας και πρόσβασης δεδομένων είναι ένας νέος, αποτελεσματικός τρόπος απεικόνισης αλγόριθμων σε αναδιατασόμενες πλατφόρμες (DAER). Αυτή η αρχιτεκτονική χωρίζει τις εργασίες της εφαρμογής σε δύο μέρη, την επεξεργασία δεδομένων (μονάδα επεξεργασίας) και την ανάκτηση δεδομένων (μονάδα ανάκτησης). Αυτός ο διαχωρισμός ελαχιστοποιεί την εξάρτηση της πρόσβασης στη μνήμη με την επεξεργασία των δεδομένων, επιτυγχάνοντας υψηλή απόδοση αξιοποιώντας τον παραλληλισμό.

Αυτή η εργασία, έχει ως στόχο την απεικόνιση του αλγόριθμου Jacobi στη πλατφόρμα HARP με την χρήση της αρχιτεκτονικής DAER για την λύση εξισώσεων Laplace. Η μέθοδος Jacobi ανήκει στους αλγόριθμους δομημένων πλεγμάτων που περιλαμβάνονται στη λίστα των 13 νάνων, οι οποίοι αντιπροσωπεύουν ενεργές περιοχές στον παράλληλο προγραμματισμό. Σε αυτή την εργασία, έχουν χαρτογραφηθεί δύο αρχιτεκτονικές που προσπαθούν να εκμεταλλευτούν τα πλεονεκτήματα της αρχιτεκτονικής DAER. Τα πειράματα διεξήχθησαν στο Academic Compute Environment (ACE) που βρίσκεται στο vLabs της Intel. Τα αποτελέσματα αυτών των πειραμάτων δείχνουν ότι η χρήση της αρχιτεκτονικής DAER στη υβριδική πλατφόρμα CPU-FPGA επιτυγχάνει μέχρι και 2x επιτάχυνση της απόδοσης σε σύγκριση με την λύση που βασίζεται μόνο στην CPU.

Λέξεις κλειδιά: DAER, Υβριδική πλατφόρμα, HARP, Μέθοδος Jacobi, CCI-P, OPAE, AFU, ASE, HW, SW, CLs, Memory requests, Μονάδα ανάκτησης, Μονάδα επεξεργασίας

Acknowledgments

First of all, I would like to thank Prof. Dionysios N. Pnevmatikatos (NTUA) for entrusting me with this assignment and Prof. Apostolos Dollas (TUC) for being my supervisor. I am thankful for their guidance and support during the work and writing of this thesis. I would also like to express my gratitude to Prof. Eftichios Koutroulis (TUC) of being the third member of the committee and for evaluating my thesis.

I would also like to express my deepest gratitude to Dr. Grigorios Chrysos for his significant contribution and guidance during this thesis. Also, I would like to thank ECE MSc Konstantinos Kyriakidis and ECE MSc Georgios Pekridis for their valuable input on Intel's HARP tools and documentation. The tools for this thesis were provided by Intel's Academic Compute Environment and I would like to express my gratitude to Intel's stuff for granting me access and guiding me, in order to develop and test on their environment.

Finally, I would like to thank my family and my friends for their support over the years. This thesis is dedicated to them.

Contents

List of Figures

List of Tables

Acronyms – Abbreviations

AFU	Accelerator Functional Unit
ASE	AFU Simulation Environment
BBB	Basic Building Blocks.
CCI-P	Core Cache Interface
CL	Cache Line
DAER	Decoupled Access/Execute for Reconfigurable platforms
EMIF	External Memory Interface
FIFO	First In First Out
FIM	FPGA Interface Manager
FIU	FPGA Interface Unit
FPGA	Field-Programmable Gate Arrays
HARP	Intel's Hardware Accelerator Research Program
HW	Hardware
MMIO	Memory Mapped I/O
OPAE	Open Programmable Acceleration Engine
QPI	Quick-Path Interconnect
RTL	Register Transfer Level
SW API	Application Programming Interface that is used to interface between an SW application and the AFU
UPI	Ultra-Path Interconnect

CHAPTER 1

Introduction

1.1 Motivation

The growing need for a more efficient process of large volumes of data leads the developers to use High-Performance Computing (HPC) acceleration platforms. One of the most promising types among the various heterogeneous acceleration platforms is the CPU-FPGA (Field-programmable gate arrays) systems. In these platforms, the CPU and the FPGA are in the same SoC (System on Chip). The developers use these platforms to implement SW-HW (software-hardware) applications with parallel computing. Therefore, many researchers develop algorithms for testing these platforms with different ways of parallel programming.

The Decoupled Access/Execute framework is a new way of mapping algorithms efficiently on Reconfigurable platforms (DAER) [1], which is motivated by the idea of Decoupled Access/Execute (DAE) architectures [2]. This way of mapping is suitable for parallel programming. The DAER framework has two units. The Fetch Unit for accessing the memory and Process Unit for data processing. This work aims to exploit all the advantages of the DAER framework in an HPC platform.

1.2 Thesis Contributions

This thesis implements an algorithm in hardware using the DAER framework for the optimization system performance. The application that is used in this work is the Jacobi method that is a Structured Grid instantiation. Structured grid is one of the algorithmic methods of the thirteen (13) Dwarfs [3]. The platform used in this work is the Intel© Xeon© Scalable Platform with Integrated FPGA. This platform was selected as it is an extremely promising piece of hardware due to its 12-core Xeon processor and integrated Arria 10 GX1150 FPGA connected with a QuickPath Interconnect (QPI) coherent link and 2 PCIe*8 links. With the advantages that provide the DAER framework, the objective of this work was the mapping of the application with two different architectures in the HARP platform and comparing them with a serial hardware implementation and a software-based solution that was implemented in the CPU of the HARP.

1.3 Thesis Outline

The remainder of this work is organized as follows:

- Chapter 2: Related work and analysis of the DAER framework
- Chapter 3: Analysis of the HARP platform and the tools that were used in this thesis.
- Chapter 4: Description of the application that was mapped to evaluate the architectures.
- Chapter 5: Presentation of all the architectures that were mapped in this work.
- Chapter 6: Description of the software and hardware implementations on the HARP platform.
- Chapter 7: Results and discussion.
- Chapter 8: Conclusions and future work.

CHAPTER 2

Related Work

2.1 The DAE architecture

In recent years, the developers use hardware accelerators to speed up applications and to compute large volumes of data. Most efforts are focused on computer systems that combine flexible software with high performing hardware. However, sometimes large volumes of data can cause serious delays and make it difficult to speed up the program with hardware accelerators. To solve this problem the developers need to optimize the subsystem of data access.

Thus was born the idea of the Decoupled Access/Execute (DAE) architecture that was presented for the first time by James E. Smith in 1982 [2]. In this approach, the system is separated into two major functional units, each with its instruction stream. These are the Access Processor (A-processor) and the Execute Processor (E-processor).

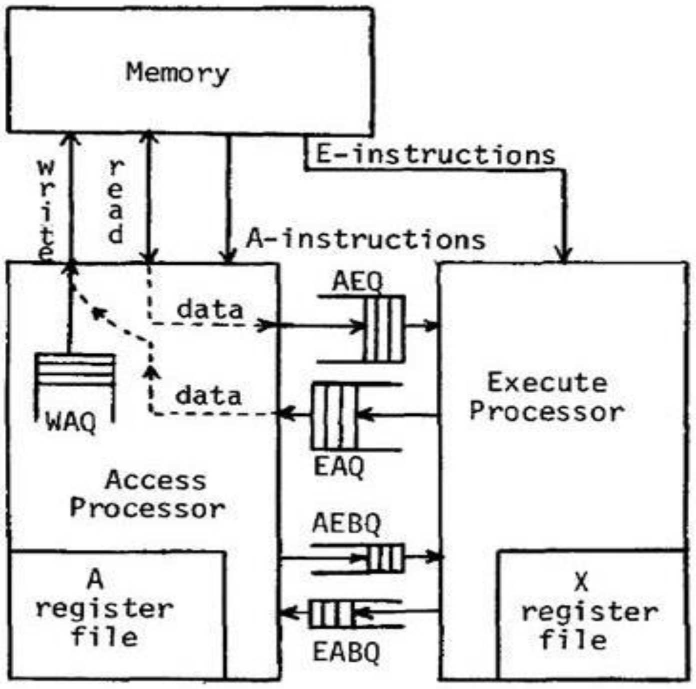

Figure 2.1 Decoupled Access/Execute Architecture (DAE)

The A-processor performs all operations necessary for transferring data to and from the main memory. It does all the address computation and performs all the memory read and write requests. Data fetched from memory is either used internally in the A-processor or it is placed in a FIFO queue and it is sent to the E-processor. This is the Access to Execute Queue (AEQ). The E-processor, process the data from the AEQ and stores the results into a second FIFO queue, the Execute to Access Queue (EAQ). The calculation of the addresses by the access processor and the results produced by the execute processor are performed in parallel. The access processor generates the addresses where the results should be stored without necessarily having received the results from the E-processor. These addresses are stored in parallel in the Write Address Queue (WAQ) and when the data arrives in conjunction with the first WAQ address they are sent to memory. This combination is done automatically as soon as the data arrives. A problem that arises, however, is that a load instruction might use the same memory location (address) as a previously issued, but not yet completed, store. The solution is to provide the programmer with interlocks to hold stores from issuing until data is available when there is a danger of a load bypassing a store to the same location. Another solution is to check which new address is loaded with the addresses stored in the WAQ. If there is one, then the loading of a new command must wait until the above check is no longer valid. However, it is a more expensive solution compared to the previous one.

It is worth noting that there is a third module, separate from the A and E processors, that handles the data and the addresses where they need to be written. EAQ can also be used to transfer data to the A-processor that is not to be written to memory but it is used to calculate addresses. In some cases, a double calculation may be required on both processors to prevent A from waiting for data from processor E. In order for the A- and E-processors to track each other, they must be able to coordinate conditional jumps or branches. It is proposed that FIFO queues also be used for this purpose. These are the E to A Branch Queue (EABQ) and A to E Branch Queue (AEBQ).

When producing software for a DAE architecture, the E- and A-processor programs have to be carefully coordinated so that data is placed into and taken out of the two data transmission queues in the correct sequence. In many cases, the accessing stream rushes ahead of the execute stream resulting in significantly less memory fetch delay.

2.2 The DAER framework

Taking into account the decouple access/execute architecture, the idea of capturing a high-performance architecture that exploits software and hardware was born, the DAER [1]. The DAER is a Decoupled Access/Execute framework for Reconfigurable accelerators. This framework has two main goals: generality and efficiency. This approach maps the code to be accelerated in two separate parts: the fetch unit and the process unit (Figure 2.2).

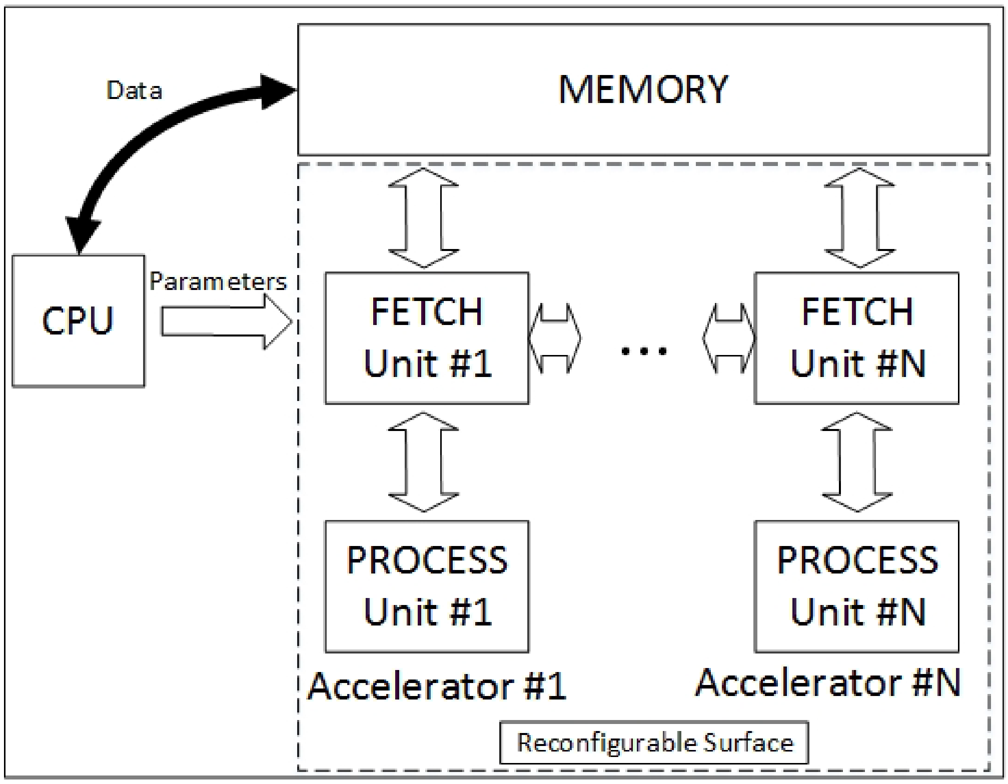

Figure 2.2 Decoupled Access/Execute Architecture for Reconfigurable accelerators (DAER)

The first part of DAER architecture is the fetch unit, which is connected to both the CPU and the memory. The CPU passes only the application's parameters. The fetch unit is responsible for fetching data to the accelerator and storing the results back in memory. DAER architecture provides the ability to map applications based on processes. Each process can be mapped to an accelerator where each accelerator can communicate with the others through the fetch unit. Therefore, sometimes the fetch unit can bypass receiving data from the main memory and receiving it directly from other fetch units.

The second part of the architecture is the Process Unit. This unit processes the fetched data in a streaming way. It communicates directly with the fetch unit through FIFO-based links. These links are used for passing data to the processing unit and sending back the results. This unit is responsible for all the logic and arithmetic operations.

This division of duty offers the user a structured and well-defined way of mapping the target application on an FPGA. Also, this approach bodes well with other hardware-based optimization techniques, e.g. pipelining, custom processing, and data prefetching, hide the memory data access latency, resulting in increased application performance. The DAER framework is a generic and platform-independent scheme. This framework, implemented with HLS mapping tools on five applications. The implementation was mapped in modern FPGA-based HPC platforms showing the performance advantages of the proposed solution for various problems on real-world platforms.

2.3 DAE in HPC Platforms

As HPC platforms are becoming an important research subject with scalability in mind, there has been a need for easiest developing in reconfigurable platforms. The generality that provides the DAE architecture has led more and more developers to map designs with these characteristics in modern FPGA-based platforms.

Tao Chen and G. Edward Suh [4] present an architecture framework to easily design hardware accelerators that can effectively tolerate long and variable memory latency using prefetching and access/execute decoupling. The proposed framework utilizes automated program analysis along with High-Level Synthesis (HLS) tools to enable prefetching and access/execute decoupling with minimal manual efforts. The framework adds tags to accelerator memory accesses so that hardware prefetching can effectively preload data for accesses with regular patterns. To handle irregular memory accesses, the framework generates an accelerator with decoupled access/execute architecture using program slicing. Experimental results show that the proposed optimizations can significantly improve the performance of HLS-generated accelerators (average speedup of 2.28x across eight accelerators) and often reduce energy consumption (average of 15%).

Tae Jun Ham et al [5] propose the Decoupled Supply-Compute (DeSC) approach that offers a way to attack communication latency bottlenecks automatically while maintaining good portability and low complexity. Their work expands prior Decoupled Access Execute techniques with hardware/software specialization. For a range of workloads, DeSC offers roughly 2x speedup, and additional specialized compression optimizations reduce traffic between decoupled units by 40%.

Konstantinos Koukos et al [6] work evaluates how much we can increase the effectiveness of DVFS (Dynamic Voltage Frequency Scaling) by using a software decoupled access/execute approach. Decoupling the data access from execution allows applying optimal voltage-frequency selection for each phase and therefore improves energy efficiency over standard coupled execution. The underlying insight of their work is that by decoupling access and execute they can take advantage of the memory-bound nature of the access phase and the compute-bound nature of the execute phase to optimize power efficiency while maintaining good performance. To demonstrate this, they built a task-based parallel execution infrastructure consisting of:

1. a runtime system to orchestrate the execution,
2. power models to predict optimal voltage-frequency selection at runtime,
3. a modeling infrastructure based on hardware measurements to simulate zero-latency, per-core DVFS, and
4. a hardware measurement infrastructure to verify their model’s accuracy.

Based on real hardware measurements they project that the combination of decoupled access/execute and DVFS has the potential to improve EDP by 25% without hurting performance. On memory-bound applications, they significantly improved performance due to increased MLP in the access phase and ILP in the execution phase. Furthermore, their method can achieve high performance both in the presence or absence of a hardware prefetcher.

José-María Arnau et al [7] research focuses on improving the energy efficiency of the GPU since graphical applications consist of an important part of the existing market. The trend towards better screens will inevitably lead to a higher demand for improved graphics rendering. They show that the main bottleneck for these applications is the texture cache and that traditional techniques for hiding memory latency (prefetching, multithreading) do not work well or come at a high energy cost. They proposed the migration of GPU designs towards the decoupled access/execute concept. Furthermore, they significantly reduced bandwidth usage in the decoupled architecture by exploiting inter-core data sharing. Using commercial Android applications, they show that the end design can achieve 93% of the performance of a heavily multithreaded GPU while providing energy savings of 34.

2.4 Architectures in HARP platform

One of the latest HPC platforms with a CPU-FPGA in one chip is the Intel's Hardware Accelerator Research Platform (HARP). It's one of the most promising platforms due to the advantages that provide the 12-core Xeon processor and integrated Arria 10 GX1150 FPGA. At this time the users can use this platform, developing with OpenCL environment or RTL environment. In this section, there are few attempts of mapping efficient architectures in both environments and evaluate the HARP platform.

de Souza Junior et al [8] studies focus on the acceleration of a chemical reaction simulation that relies on a system of a stiff ordinary differential equation (ODEs) targeting heterogeneous computing systems with CPUs and field-programmable gate arrays (FPGAs). Specifically, they target an essential kernel of the coupled chemistry aerosol-tracer transport model to the Brazilian developments on the regional atmospheric modeling system (CCATT-BRAMS). They focus on a linear solve step using the QR factorization, based on the modified Gram-Schmidt method, as the basis of the ODE solver in this application. The implementations took place in the HARP platform using the OpenCL programming environment. The results show that the hardware design is up to 4 times faster than the original iterative Jacobi method but even with hardware support, the overall performance of the QR-based hardware is lower than its original software version.

Kyriakidis, K's master thesis [9] describes the development process of a series of HW components for Intel's HARP CPU-FPGA hybrid platform that will be used to synthesize a Trace-Driven FPGA Accelerated Full-System Architectural Simulator. The developed modules facilitate high-performance HW components that can accurately and efficiently simulate a highly configurable L1 Cache and 3 highly configurable Branch Predictor HW structures. In this thesis are presented the SW and the HW implementations of the simulator. This simulator is designed to exploit the benefits of the HARP CPU-FPGA hybrid platform and the QPI interconnect to its max.

Pekridis, G's master thesis [10] goal was to evaluate Intel's HARP platform. The evaluation was done with an ARM many-core accelerator. The ARM core has a 3-stage pipeline, and it uses a 32-bit architecture and is implements the ARMv4 instruction set. Also, it implements a few basic floating-point instructions. The implementations took place in the HARP platform using the RTL environment. The RTL for the ARM core was written in Bluespec System Verilog (BSV). The hardware architecture has 16 ARM cores. Depending on the measurements of the read and write bandwidth of the system and the results of the benchmarks with the ARM accelerator, the conclusion that comes is that streaming applications would be preferable for this platform.

CHAPTER 3

Tools and Platform

3.1 The HARP Platform

The platform, in which the accelerators are mapped, is the Intel Xeon Scalable Platform with Integrated FPGA (HARP Platform) [11]. This platform has an Intel Xeon CPU with an FPGA in a single package that shares the same DRAM main memory by using the same Quick Path Interconnect (QPI) high-speed link and 2 PCIe 3.0 x8 links. The CPU is a Broadwell Xeon (E5-2600v4) with 12 cores (24threads) at 1.6-2.2GHz [12]. The FPGA model is an Arria 10 GX1150 FPGA (10AX115U3F45E2SGE3) [13]. The communication between the CPU and the FPGA is achieved with the Core Cache Interface (CCI-P).

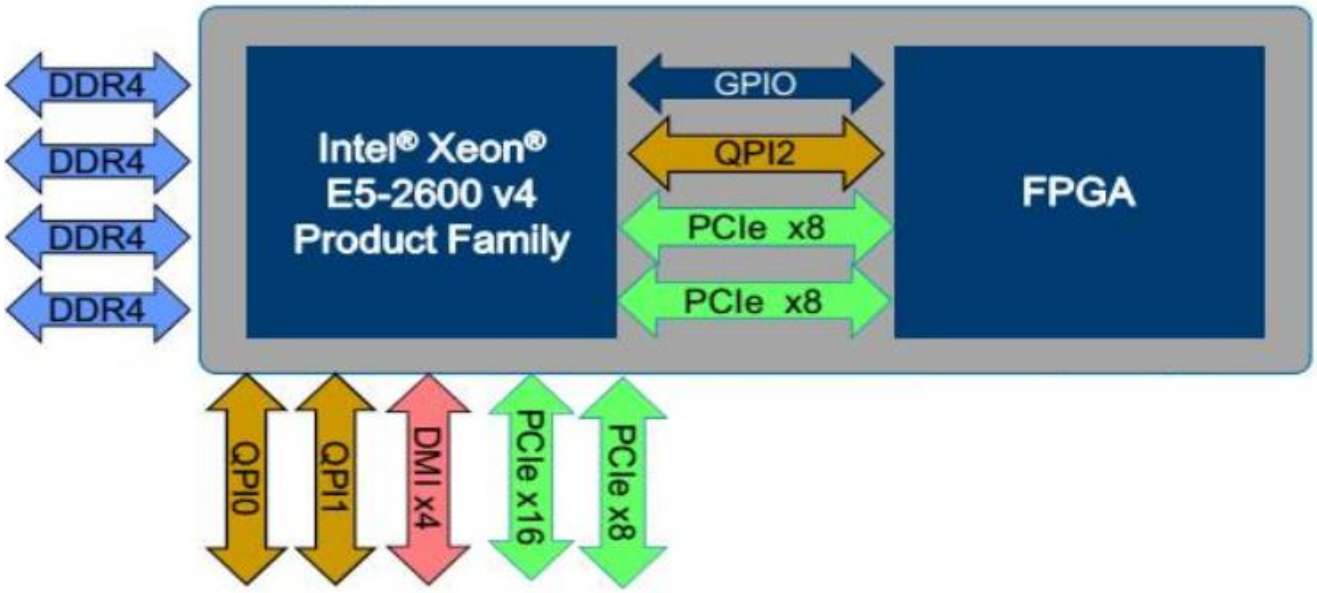

Figure 3.1. Intel Xeon-FPGA Hybrid Chip. ©Intel Corporation

3.2 FPGA Interface Manager (FIM)

The Intel FPGA Accelerator package consists of an FPGA Interface Manager (FIM) and an Accelerator Functional Unit (AFU).

The FIM consists of the following:

- FPGA Interface Unit (FIU)
- External Memory Interface (EMIF) for interfacing to external memory
- High-Speed Serial Interface (HSSI) for external transceiver interfacing.

The FIU acts like a bridge between the AFU and the platform. The FIM owns all hard IPs on FPGA (for example PLLs), partial reconfiguration (PR) engine, JTAG atom, IOs, and temperature sensors. The FIM is defined as a static region that configured first at boot up and persists until the platform power cycles, whereas the AFU is defined as a partial reconfiguration region that can be dynamically reconfigured.

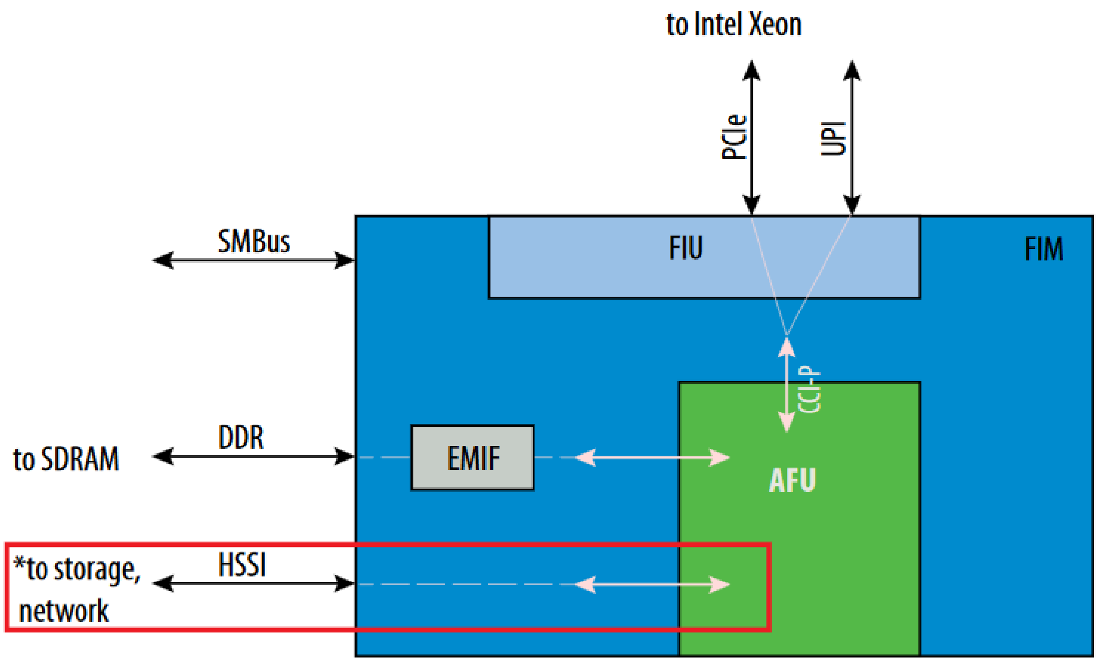

Figure 3.2. FPGA Interface Manage. ©Intel Corporation

3.3 FPGA Interface Unit (FIU)

The CPU is connected with the FPGA with a Quick Path Interconnect (QPI) high-speed link and 2 PCIe 3.0 x8 links. The FIU maps these three links to the CCI-P interface so that the AFU sees a single logical communication interface to the host processor with a bandwidth equal to the bandwidth of the three links in total. CCI-P maps the three physical links to four virtual channels: PCIe0 to VH0, PCIe1 to VH1, UPI to VL0, and all physical links to VA. VA implements a weighted de-multiplexer to route the requests to all of the physical links. The FIU uses the PCIe-0 for the communication of the AFU with the software. Therefore, the AFU is discovered and enumerated over PCIe-0.

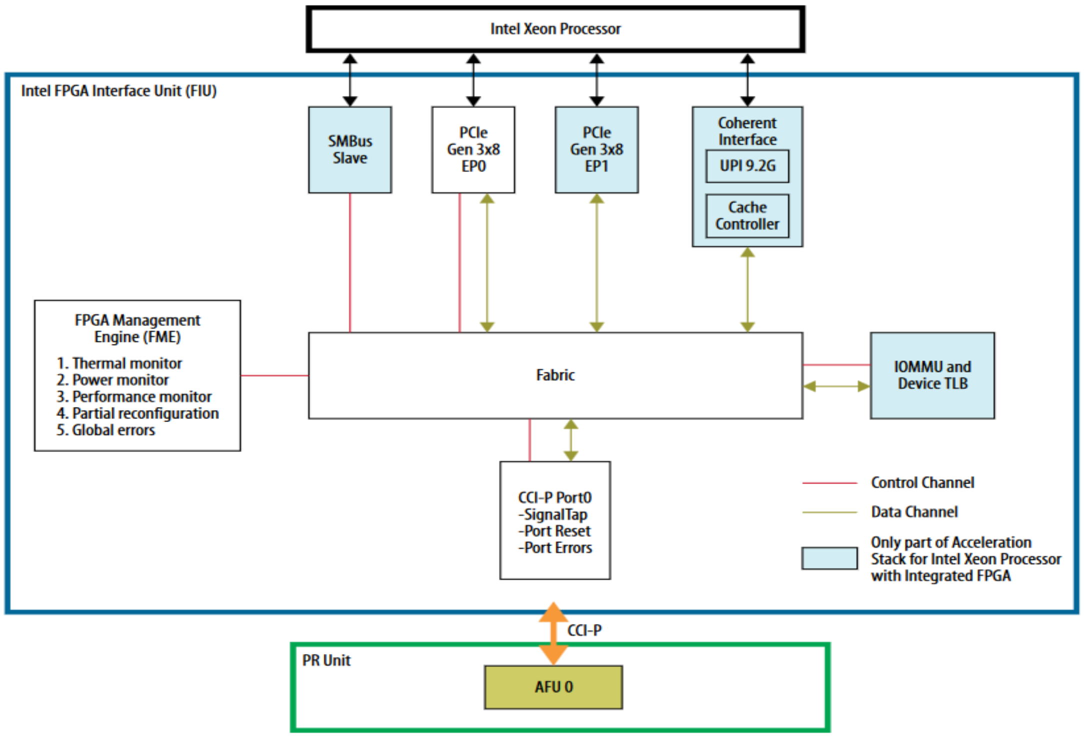

Figure 3.3. FIU for Intel Integrated FPGA Platform Block Diagram. ©Intel Corporation

3.4 Memory and Cache Hierarchy

Figure 3.4 shows the three-level cache and memory hierarchy seen by an AFU. A single processor Integrated FPGA Platform has only one main memory SDRAM (A.3). The first level is the FPGA Cache (A.1). An AFU memory request that hit in FPGA cache has the lowest latency. When all the requests hit the FPGA Cache, the design achieves the highest bandwidth. An AFU that uses VL0 virtual channel and VA requests looks up the FPGA cache first, and only upon a miss looks upon the other levels. The second level (A.2) is the CPU-side cache. The requests that hit at this level have higher latency than FPGA cache but lower latency than reading from SDRAM (A.3). The SDRAM is the third level that is used only in misses in the Processor-side cache (A.2). Most AFUs achieve maximum memory bandwidth by choosing the VA virtual channel that selects automatically witch of the three physical links will be used. The FIU selects among the VL0, VH0, and VH1 by taking account of the physical link latency and efficiency characteristics, physical link utilization, and traffic distribution.

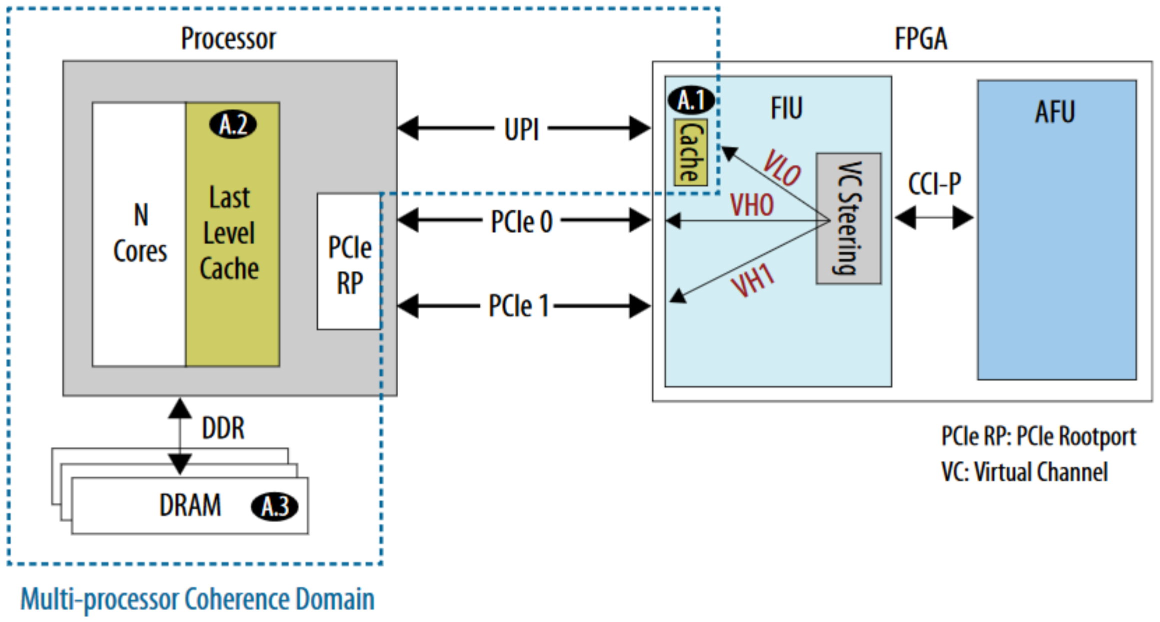

Figure 3.4. Integrated FPGA Platform Memory Hierarchy. ©Intel Corporation

3.5 The CCI-P Interface

The CCI-P [14] implements to Main Memory (CPU Cache and DDR) and Memory Mapped I/O (MMIO) address spaces. The main memory is the memory of the CPU-side that can be accessed by the AFU. The CCI-P defines a request format to access I/O memory using memory-mapped I/O (MMIO) requests. Requests from the AFU to the main memory are called upstream requests and to the MMIO space are called downstream requests. The AFU's MMIO address space is 256 kB in size and is used for communication between software and hardware. All CCI-P signals are grouped into three Tx channels, two Rx channels, and some additional control signals. The Tx requests flow from the AFU to the FIU and the Rx requests from the FIU to the AFU. All CCI-P signals must be synchronous to the same clock and same reset. The CCI-P signals are shown in figure 3.5 and the most important of them are analyzed in the description of the requests.

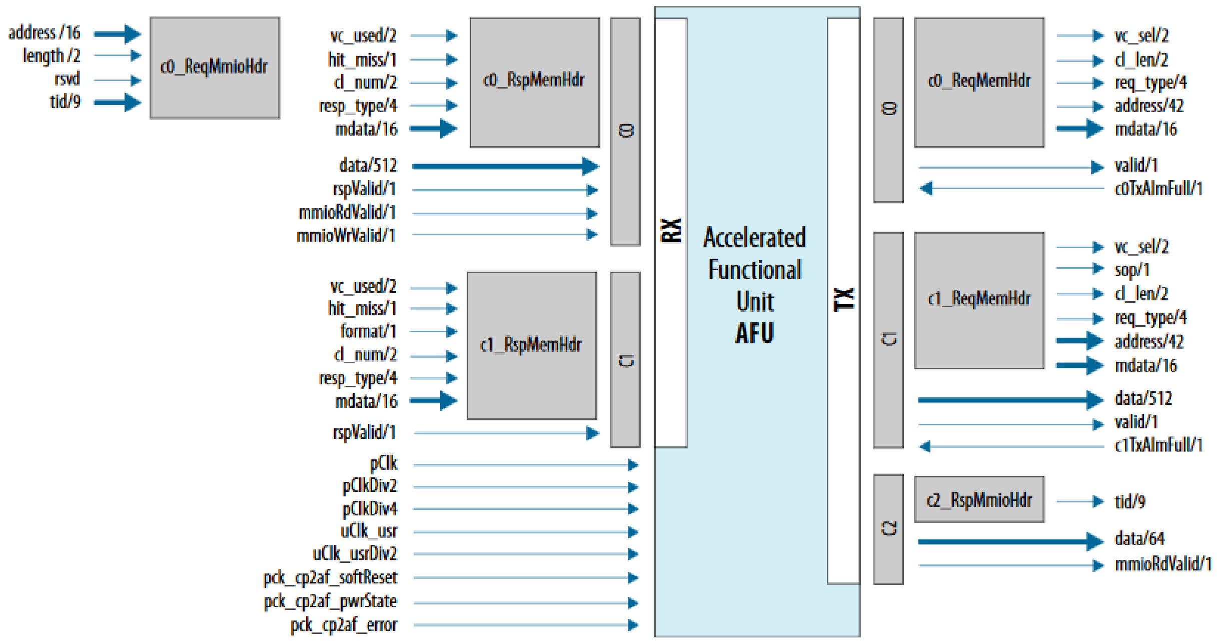

Figure 3.5. CCI-P signals. ©Intel Corporation

3.5.1 Read and Write to Main Memory

Read Requests:

The AFU sends a memory read request over CCI-P Channel 0 (C0), using pck_af2cp_sTx.c0 and receives the response over C0, using pck_cp2af_sRx.c0. The response is 1, 2, or 4 CL (Cache Lines), out of order, depending on the developer's choice (rd_hd.cl_len). The ID of every request is the value of the rd_hdr.mdata and the ID of every CL is the value of the sRx.c0.hdr.cl_num. The read request is sent through the link when the sTx.c0.valid signal is set and in response, the sTx.c0.rspValid is set for the successful read. The rd_hdr.address sets the DDR address that the AFU will read.

Write Requests:

The AFU sends memory write requests over CCI-P Channel 1 (C1), using pck_af2cp_sTx.c1 and receives write completion acknowledgement responses over C1, using pck_cp2af_sRx.c1. The AFU drives the request with the data (512 bit) when the pck_af2cp_sTx.c1.valid is set and the sRx.c1.rspValid is set for the successful write. The wr_hdr.address sets the DDR address that the AFU will write to.

3.5.2 MMIO - Accesses to I/O Memory

MMIO Reads:

The AFU receives an MMIO read request over pck_cp2af_sRx.c0. The CCI-P asserts mmioRdValid and drives the MMIO read request on hdr. The response is received via the pck af2cp sTx.c2. The data lengths supported are 4bytes and 8bytes.

MMIO Writes:

The AFU receives an MMIO write request over pck_cp2af_sRx.c0. The CCI-P asserts mmioWrValid and drives the MMIO write request header and data on hdr and data, respectively. The response is received via the pck af2cp sTx.c2. The data lengths supported are 4 bytes, 8 bytes, and 64 bytes.

3.6 The OPAE C API

The OPAE C library (*libopae-c*) [15] is a lightweight user-space library that provides abstractions for FPGA resources in a computing environment. The OPAE C library builds on the driver stack that supports the FPGA device, abstracting hardware- and OS-specific details. It provides access to the underlying FPGA resources as a set of features available to software programs running on the host. These features include the acceleration logic preconfigured on the FPGA and functions to manage and reconfigure the FPGA. The library enables applications to transparently and seamlessly benefit from FPGA-based acceleration.

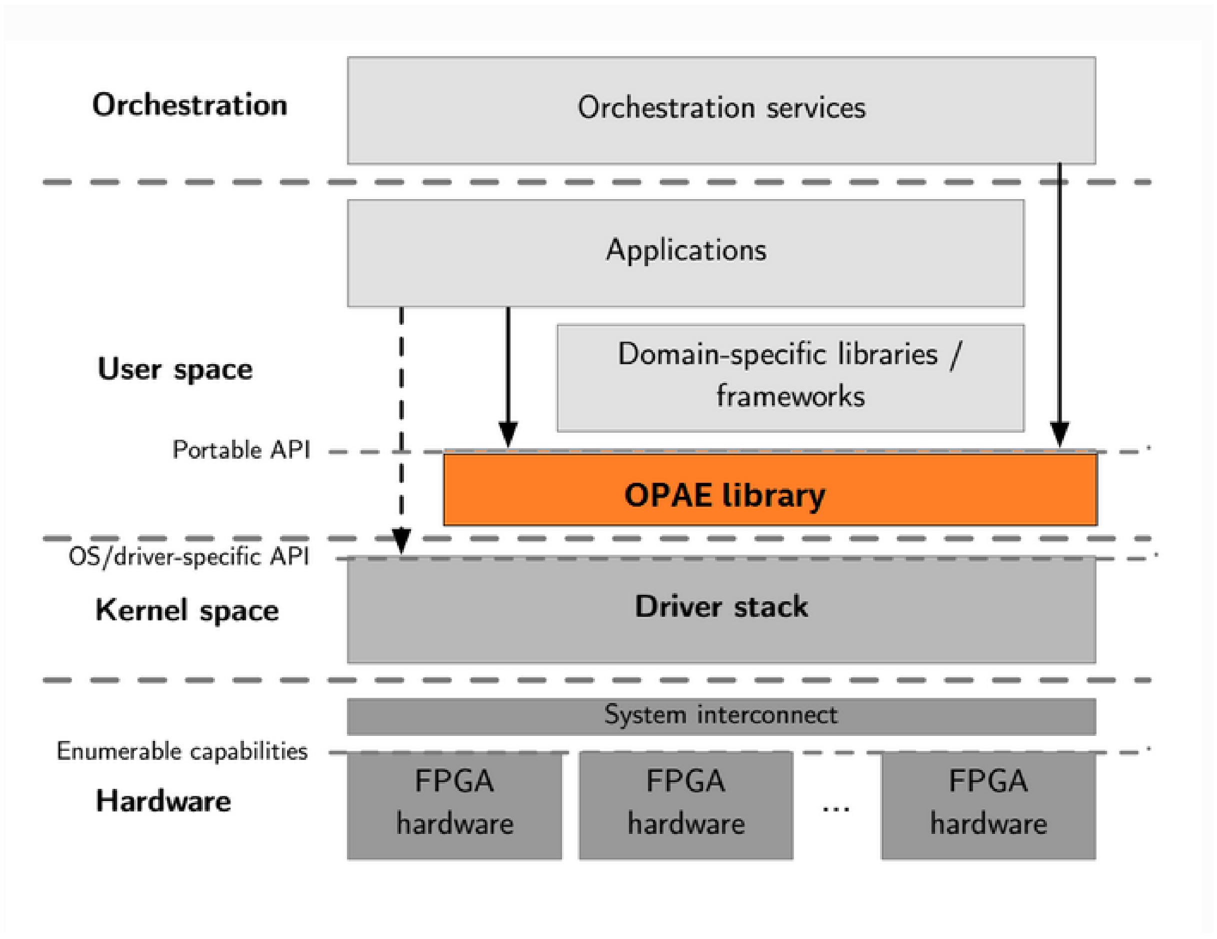

Figure 3.6 OPAE. ©Intel Corporation

Figure 3.7 shows the basic application flow from the viewpoint of a user-process.

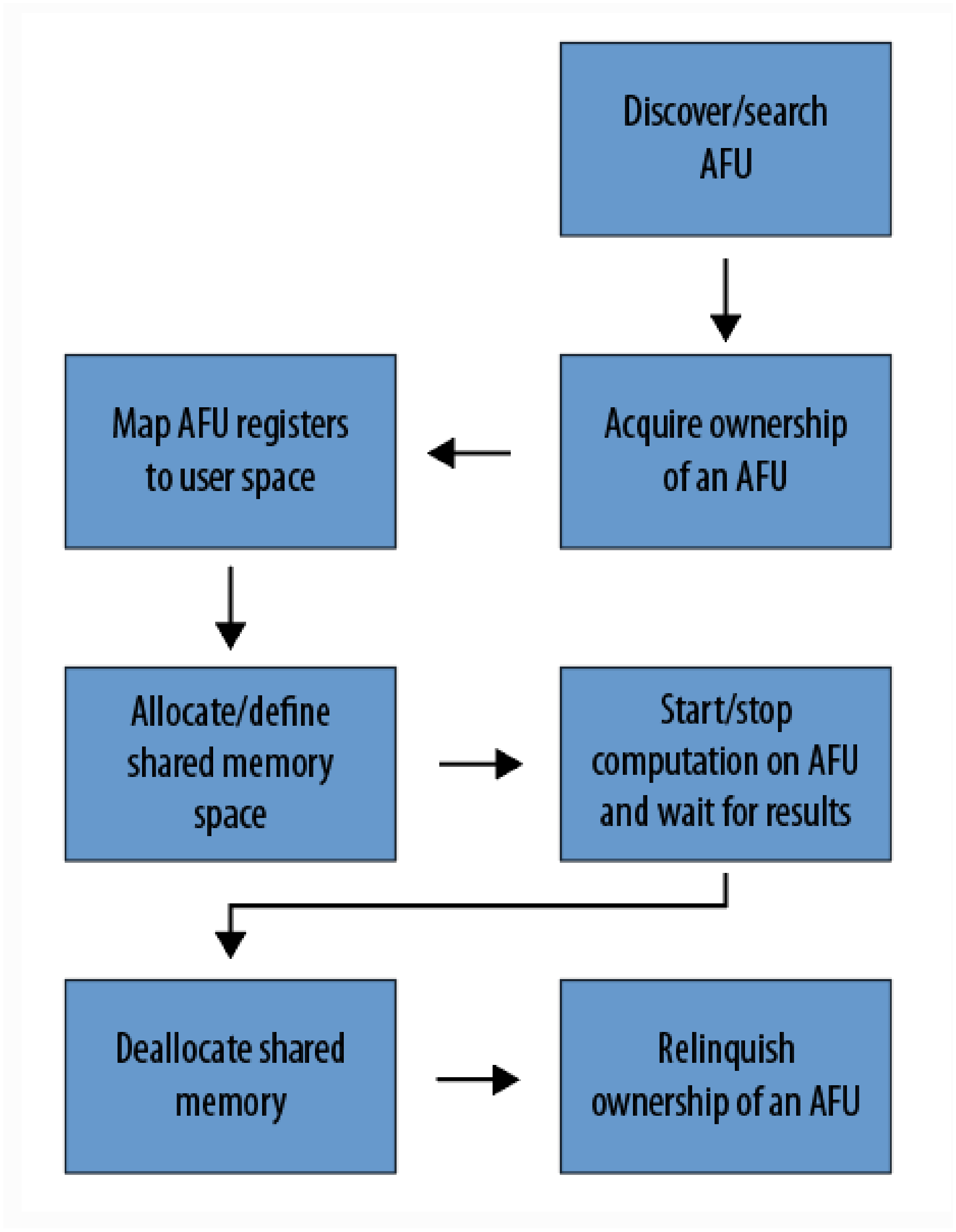
Discover/search AFU
Acquire ownership of an AFU
Map AFU registers to user space
Allocate/define shared memory space
Start/stop computation on AFU and wait for results
Deallocate shared memory
Relinquish ownership of an AFU

Figure 3.7 Application flow. ©Intel Corporation

3.7 The OPAE ASE simulation

The AFU Simulation Environment (ASE) provides a consistent transaction-level hardware interface and software API that allows users to develop a production-quality Accelerated Functional Unit (AFU) and host software application. To use the ASE Environment users must have source code in a language that RTL (Register Transfer Level) simulators can interpret. In this work the language of RTL is SystemVerilog.

ASE is a dual-process simulator. Process one is responsible for running the RTL simulation, while the other completes the AFU simulator by connecting the software with the RTL.

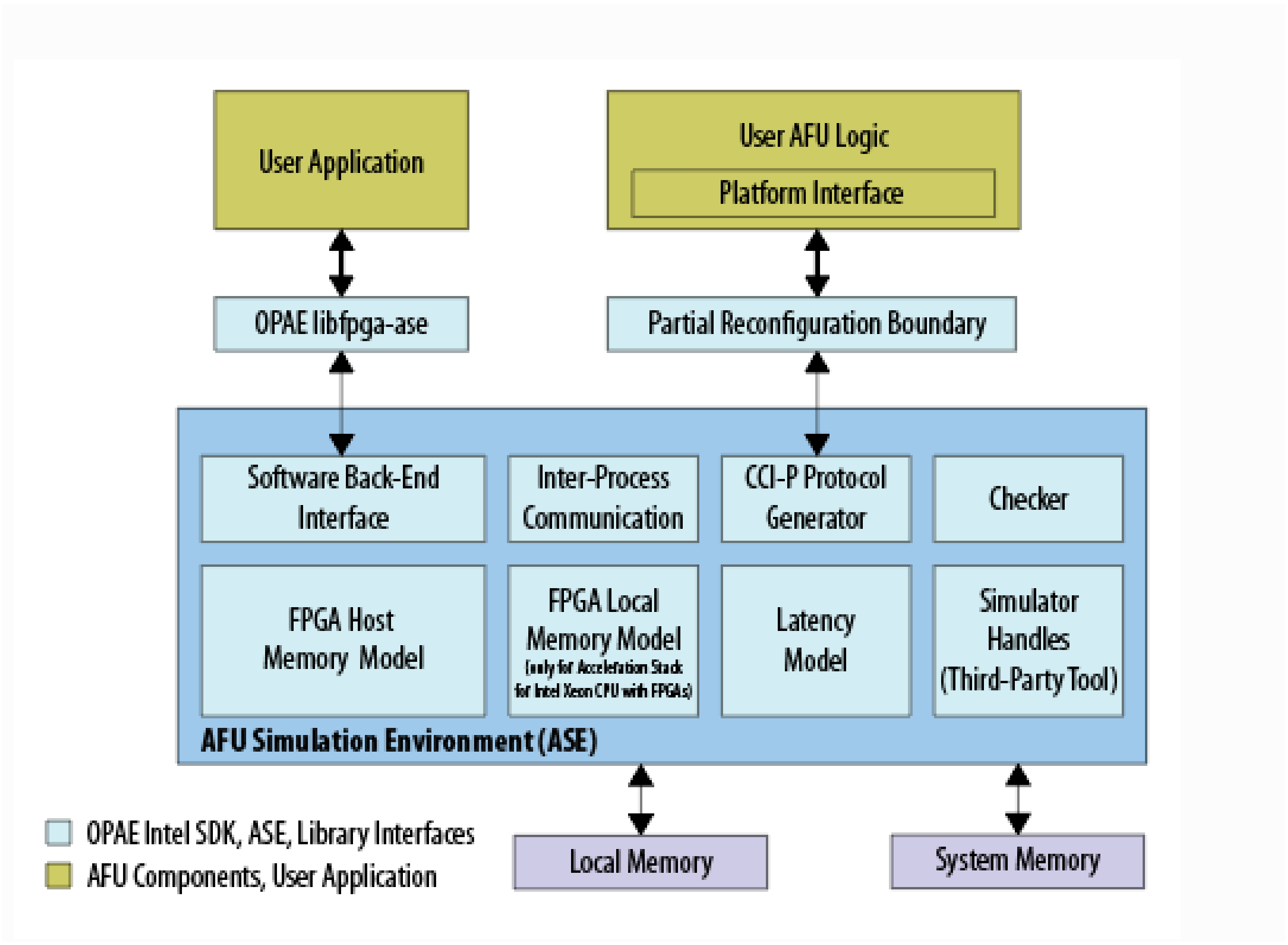
User Application
User AFU Logic
Platform Interface
OPAE libfpga-ase
Partial Reconfiguration Boundary
Software Back-End Interface
Inter-Process Communication
CCI-P Protocol Generator
Checker
FPGA Host Memory Model
FPGA Local Memory Model (only for Acceleration Stack for Intel Xeon CPU with FPGAs)
Latency Model
Simulator Handles (Third-Party Tool)
AFU Simulation Environment (ASE)
OPAE Intel SDK, ASE, Library Interfaces
AFU Components, User Application
Local Memory
System Memory

Figure 3 8 ASE dual-process simulator overview. ©Intel Corporation

The ASE provides two interfaces:

- Software: OPAE API is implemented in the C programming language.
- Hardware: Core Cache Interface (CCI-P) specification implemented in SystemVerilog.

The ASE is vital for developing in the SW and the HW (AFU). It includes a behavioral model of the FPGA Interface Manager (FIM) IP that provides immediate feedback on functionality during the development phase that flags errors or warnings in the CCI protocol, memory accesses, or simply syntax errors. The best way to start developing on HARP is by using the Basic Building Blocks (BBB). The BBB includes reference sample codes that developers can use or modify for their work. Several example AFUs are stored in the intel-fpga-bbb repository [16] that were useful in this work.

The ASE presents a memory model that can inform illegal memory transactions or requests to locations outside the memory spaces. All these information are evaluable to the developer in memory and transaction report files when an implementation in ASE is run. However, it must be stated that a correct ASE simulation does not guarantee a correct AFU synthesis. For example, the ASE runs at 100MHz and the hardware clock frequency can be pushed up to 400Mhz. The timing report will be available after the synthesis. For all this information and more there is the “Intel Accelerator Functional Unit (AFU) Simulation Environment (ASE) User Guide” [17]. In the Academic Compute Environment (ACE) [18], which is located on vLabs, the simulation can be started from a node with the command “qsub-sim”.

3.8 Synthesis - Quartus Prime Pro

Once AFU RTL and software are functionally correct the Bitstream Generation phase begins. The synthesis requires the Quartus Prime Pro Edition 16.0.0 [19]. The synthesis process creates the green bitstream (.gps) that will be combined with the blue one that is already synthesized by the OPAE during the synthesizing phase, to complete a design. In vLabs, the synthesis can be started from a node with the command “qsub-synth”.

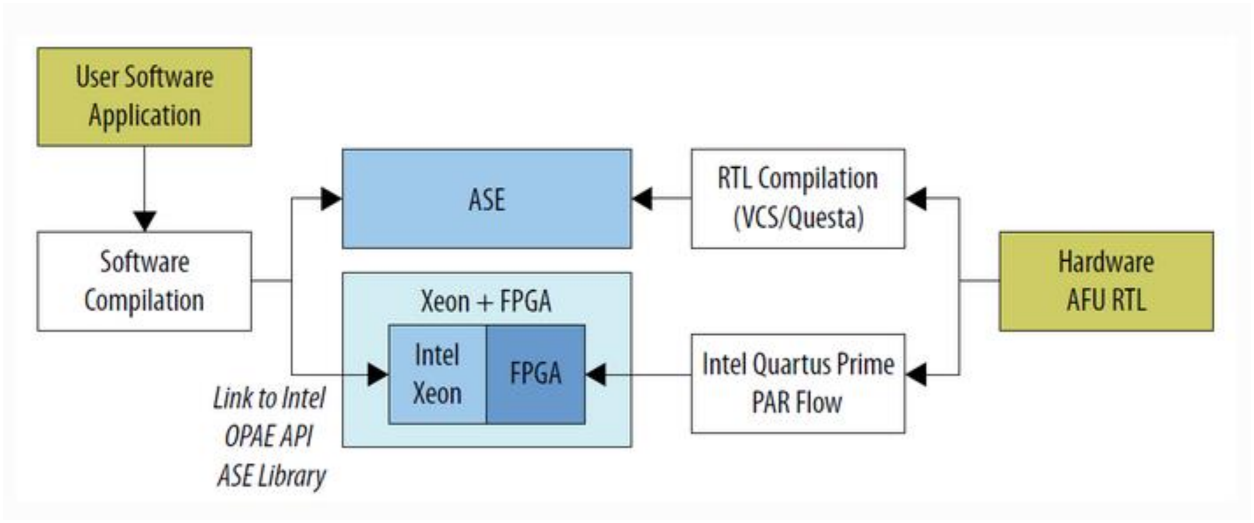
User Software Application
Software Compilation
ASE
RTL Compilation (VCS/Questa)
Hardware AFU RTL
Xeon + FPGA
Intel Xeon
FPGA
Intel Quartus Prime PAR Flow
Link to Intel OPAE API ASE Library

Figure 3.9 ASE simulation and Quartus Synthesis overview. ©Intel Corporation

3.9 Developing in vLabs

The vLabs [20] is a very efficient environment to make synthesis and ASE simulation because it provides all the previously mentioned scripts and utilities, as well as all the licenses. The first step is to develop the RTL files and test them with ASE in the simulation node (qsub-sim). The synthesis can take ours, so the best way of developing the code is with ASE. The synthesis is recommended to start only when the design is free of simulation errors. As soon as the synthesis is done, the developer can check all the report/log files for errors that the simulation couldn’t find. The developer can use these files to correct problems regarding the AFU RTL or even the SW API. The timing report that is evaluable only after the synthesis shows hold violations (slack) or clock closure (gated clocks). If an error is detected, the development phase should be restarted so that the model can be re-validated. At last, and when the AFU is error-free, the generated (green) bitstream can be downloaded in the FPGA. In vLabs the synthesis can be started from a node with the command “qsub-synth”, and the download of the AFU in the FPGA with the command “qsub-fpga”. At last, Quartus prime pro was also used to create the FIFOs that were used for the designs.

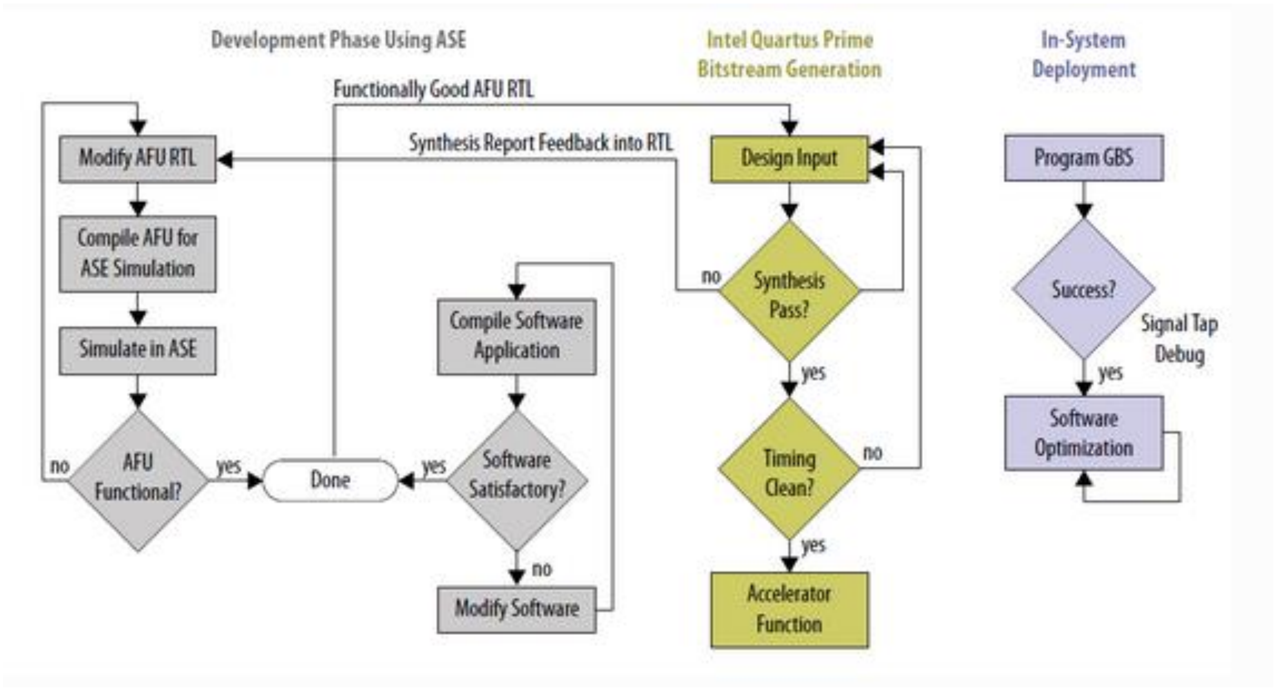

Figure 3.10 Workflow of HARP development. ©Intel Corporation

CHAPTER 4

Application

For the evaluation of the architectures, an application of the Dwarfs benchmark suit [3] was used. The Dwarfs are algorithmic methods that capture a pattern of computation and communication and are widely adopted by the research community. Many research works test them on reconfigurable HPC platforms and other high-parallel frameworks [21], [22]. More specifically the DAER framework that is mapped in this thesis was tested in platforms that have similar characteristics with HARP, using five of these applications [1]. The algorithm that was used in this work is the 2D Laplace equation with Dirichlet conditions that is a Structured Grid instantiation.

4.1 Structured Grids

Structured grid is one of the algorithmic methods of the thirteen (13) Dwarfs and has been created by researchers in the Par Lab at the University of California at Berkeley [3]. One problem with many dimensions can be described by a structured grid. Structured grids are identified by regular connectivity. This method has high space-efficiency since the neighborhood relationships are defined by the storage arrangement. In engineering, it is often necessary to present points of a grid and their values which are updated from their neighboring points. This value can simulate the temperature, the color of a pixel of an image, and more. Each exit point is calculated from the value of each element and the value of the nearest adjacent points. A major advantage of structured grids in hardware is the high spatial locality and temporal locality because of the use of adjacent cells and the use of the same piece of data frequently. The possible element choices are quadrilateral in 2D and hexahedra in 3D etc. In two dimensions a grid point can be specified by a pair of indexes (i,j) and adjacent cells can be identified by increasing or decreasing the indexes i or j ([i+1,j], [i-1,j], [i,j+1], [i,j-1]).

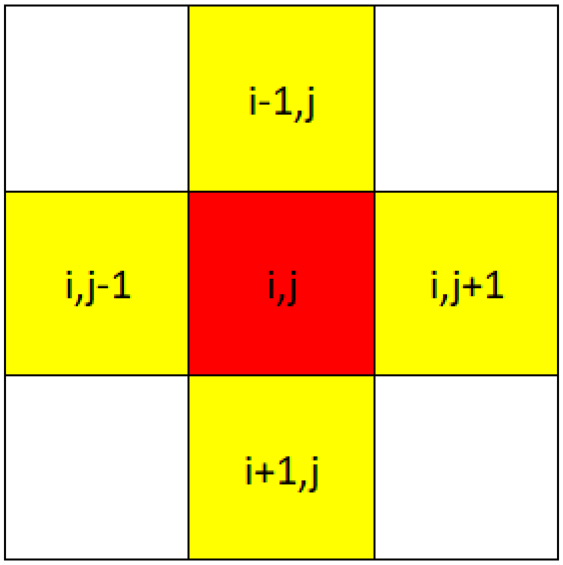

Figure 4.1 Nodes in a structured grid

4.2 Solving Laplace’s equation using Jacobi iteration

Laplace's equation in a regular grid, shown in Figure 4.1, is:

$$\Delta u = \frac{\vartheta^2 u}{\vartheta x^2} + \frac{\vartheta^2 u}{\vartheta y^2} = 0 \qquad (4.1)$$

The simplest iterative method for solving a system of linear equations (*Au = b with* $A \in R^{n\times n}$) is Jacobi’s Method**.**

This method requires two conditions:
(1) The given system has a unique exact solution
(2) The coefficient matrix A is a strictly diagonal dominant matrix, thus $|a_{ii}| > \sum_{j\neq i}|a_{ij}|$ for all i.

Each diagonal element is solved for, and an approximate value plugged in. The process is then iterated until it converges.

For example, **Au=b** is a square system of n linear equations, where [23]:

$$A = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{bmatrix}, \quad u = \begin{bmatrix} u_1 \\ u_2 \\ \vdots \\ u_n \end{bmatrix}, \quad b = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix} \qquad (4.2)$$

Then A can be decomposed into a diagonal component *D*, a strict lower triangular *L,* and a strict upper triangular *U*:

$$D = \begin{bmatrix} a_{11} & 0 & \cdots & 0 \\ 0 & a_{22} & \ddots & \vdots \\ \vdots & \ddots & \ddots & 0 \\ 0 & \cdots & 0 & a_{nn} \end{bmatrix}, \quad L = \begin{bmatrix} 0 & 0 & \cdots & 0 \\ a_{21} & 0 & \ddots & \vdots \\ \vdots & \ddots & \ddots & 0 \\ a_{n1} & \cdots & a_{nn-1} & 0 \end{bmatrix}, \quad U = \begin{bmatrix} 0 & a_{12} & \cdots & a_{1n} \\ 0 & 0 & \ddots & \vdots \\ \vdots & \ddots & \ddots & a_{n-1n} \\ 0 & \cdots & 0 & 0 \end{bmatrix}, \quad (4.3)$$

Then Jacobi's Method can be written in matrix-vector notation as

$$Du^{(k+1)} + (L+U)u^k = b \quad (4.4)$$

The solution is then obtained iteratively via:

$$u^{(k+1)} = D^{-1}[(-L-U)u + b]$$

Where ***u**^(k)^* is the ***k***th approximation or iteration of ***u*** and ***u**^(k+1)^* is the next or *k + 1* iteration of ***u*** [24].

The element-based formula is thus [23], [25]:

$$u_i^{(k+1)} = \frac{1}{a_{ii}}\left(b_i - \sum_{j=1}^{i-1} a_{ij}u_j^k - \sum_{j=i+1}^{n} a_{ij}u_j^k\right), \quad i = 1,2,3,\dots,n \ and \ k = 0,1,2,\dots \quad (4.5)$$

The computation of $u_i(k+1)$ requires each element in $u(k)$ except itself. In this method, we can't overwrite $u_i(k)$ with $u_i(k+1)$, as that value will be needed by the rest of the computation. The minimum amount of storage is two vectors of size n.

The iterative process is stopped at the minimum value of *k* such that $\left\| u_i^{(k+1)} - u_i^k \right\| < \varepsilon$ where *ε* is a fixed tolerance. Since the exact solution is not available, it is necessary to introduce suitable stopping criteria to monitor the convergence of the iteration [26]

The best way to write the Jacobi method for Laplace's equation is in terms of the residual defined (at iteration k) [27]:

$$u_{ij}^{(k+1)} = \frac{1}{4}(u_{i+1,j}^{(k)} + u_{i-1,j}^{(k)} + u_{i,j+1}^{(k)} + u_{i,j-1}^{(k)}) \quad (4.6)$$

Assuming we have an n×n table the Jacobi method is applied to the (n-2)×(n-2) space in the center of the table As seen in figure 4.2, the red blocks of the memory will change in every iteration of the algorithm. To proceed to the next iteration, this application must have the results of the previous one. That makes the Jacobi method quite slow compared to other algorithms. However, it is quite useful because it is the base for other methods. It's easy for a developer to use this algorithm as a base and create an algorithm that is distinguished for its flexibility and efficiency because he takes trivial steps to create an application with parallel programming standards. Another characteristic of the algorithm as a Dwarf is that it requires memory access for a large amount of data. That makes this algorithm perfect for testing parallel architectures like DAER. By splitting the instruction stream into two modules where the first is responsible for retrieving the addresses and the second for processing the results, it’s easy to understand that the DAER architecture improves the performance of the above algorithmic method since a command does not need to be executed to start processing the next one.

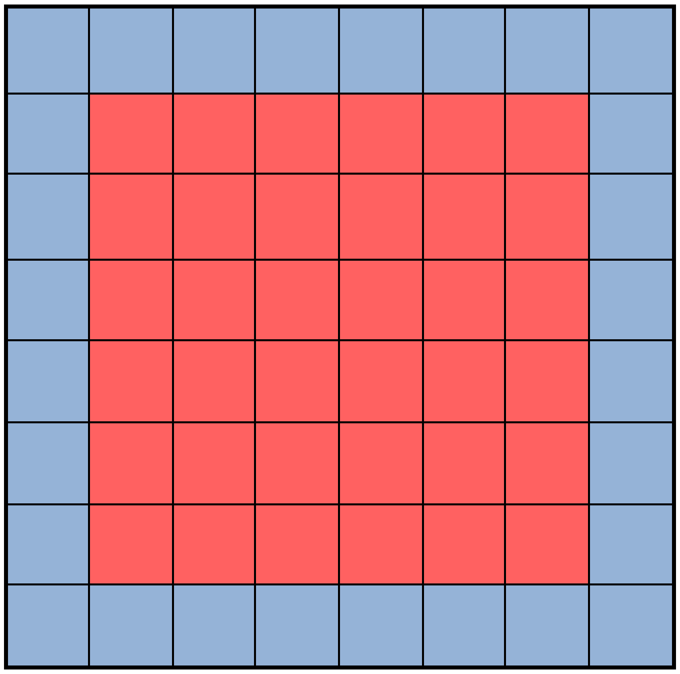

Figure 4.2 Implementation area of the Jacobi method

CHAPTER 5

Architectures

In this chapter, there is an extensive description of the hardware designs that were implemented in this thesis. They were mapped towards the development of an optimized DAER architecture that exploits the advantages of the CPU-FPGA platform.

5.1 Serial architecture

This is the first attempt of mapping an accelerator in HARP using the samples and the tools that Intel provides to the developers. The AFU makes four read requests, calculate the data, and then makes a write request serially. There is no parallelization of the data flow and the AFU must wait for all the responses of the requests and all the calculations of the data before making another four requests. It is difficult for this architecture to compete with a CPU because of that access pattern. This attempt will be the base for developing an optimized DAER architecture that can accelerate the application.

5.2 Decoupled Read/Execute architecture (Architecture v1.1)

This architecture is the first step to develop an optimized DAER architecture for structured grids. This architecture is trying to achieve parallelism by using two parallel units that will hide the latency of the fetched data. The goal was to map an architecture that focuses on hiding the memory read responses because in this application four data must be fetched for only one result. The two units that will be analyzed below are shown in figure 5.1.

<u>Fetch Unit:</u>

This unit is responsible for all the read requests. It uses the read channel to send a request (Tx.c0) to the memory and wait for the responses from the same channel (Rx.c0). All the responses are fetched in the form of a Cache Line (CL). In this application, for the calculation of one result data four input data must be fetched. Because of that, the Fetch Unit makes four read requests from different addresses and stores the responses in four Fetch to Process Queues (FPQ) FIFOs. The Fetch Unit recognizes the responses from the tag, which is a unique ID of every request. The calculations of the addresses are continuous and the Fetch Unit doesn't wait for the results or the writes. The prefetching of the data, that this unit provides, hides the memory data access latency resulting in increased application performance.

Process Unit:

This unit is responsible for all the logic and arithmetic operations and the memory writes. The Process Unit fetches all the data from the FPQ FIFOs and then processes them in a streaming way. There is a decoder in this unit that decides which data must be changed and which not. More specifically, in this algorithm, the first and the last column, and the first and the last line must have always the same values. For that reason, the decoder must check which of the data in the fetched CLs must be changed. When one CL of results is ready, the Process Unit makes a write request with the use of the write channel (c1), to the memory (Tx.c1) and waits for a response (Rx.c1). The address for this write has been already calculated during the data calculations.

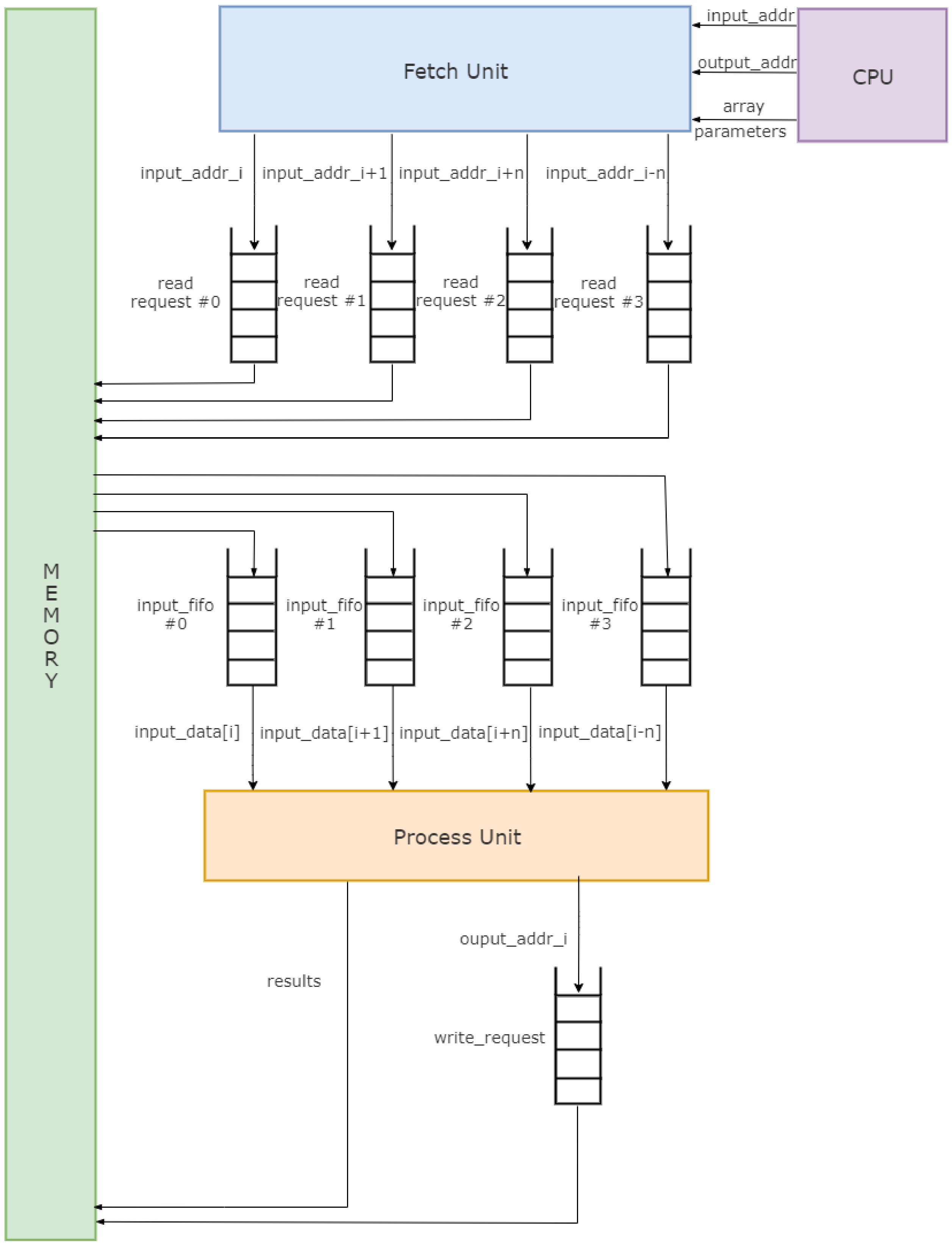

Figure 5.1 Block diagram of the Decoupled Read/Execute architecture v1.1

5.3 Decoupled Access/Execute architecture (Architecture v2.1)

As mentioned before the DAER framework's goals are generality and efficiency. For a more general design, a complete DAER architecture must be mapped. In this architecture, the Fetch Unit has two sub-units. One that makes only read memory requests and one that makes only write memory requests. So, the main difference with the previous architecture is that this design is trying to hide the latency of all the memory responses (read and write). The units of this architecture that will be analyzed below are shown in figure 5.2.

Fetch Unit:

This unit is responsible for the memory access. To make the reads and the writes independently, the Fetch Unit has two sub-Units that work in parallel.

a) *Read sub-Unit*:

This sub-unit makes only read requests and waits for the responses from the memory. It makes the same operations with the Fetch Unit of the previous architecture. For that reason, the FIFO links (FPQ) that it uses for feeding the Process Unit are the same.

b) *Write sub-Unit*:

The difference with the previous architecture is that this unit makes the writes in the memory and not the Process Unit. For receiving the results there is a FIFO link between this sub-unit and the Process Unit. This is the Process to Fetch Queue (PFQ). When there are data stored in this FIFO the Write sub-Unit makes a write request to the memory and then waits for another data to be stored in the FIFO. The address for this write has been already calculated before the data arrive in the FIFO.

Process Unit:

This Process Unit is responsible only for the logic and arithmetic operations unlike the Process Unit of the previous architecture that makes also the memory writes. Because this unit communicates with the Fetch Unit in both directions, there two cases that stops processing. The one is that the FPQ FIFOs are empty and there is no data to calculate and the other is when the PFQ FIFO is almost full.

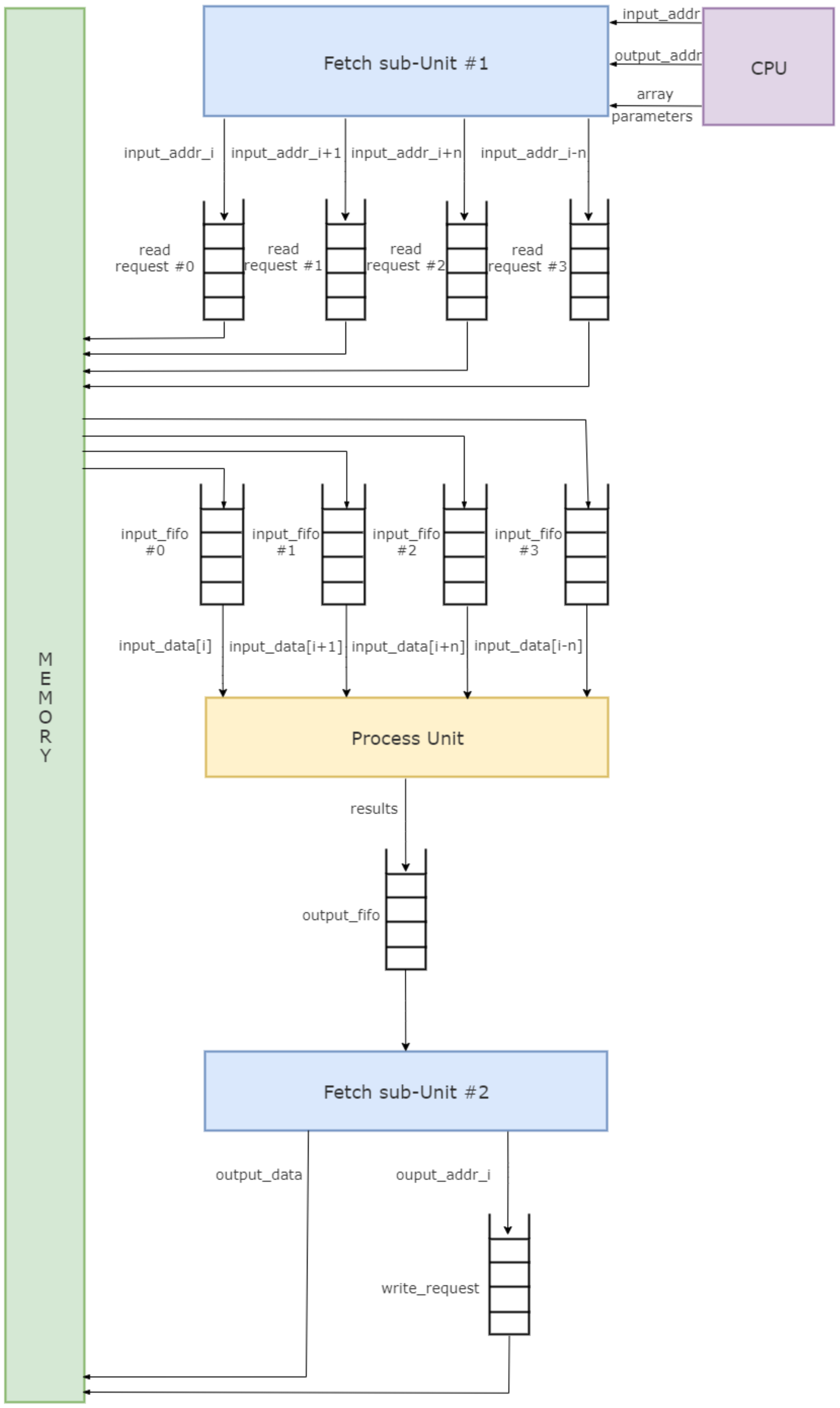

Figure 5.2 Block diagram of the Decoupled Access/Execute architecture v2.1

5.4 Optimized architectures for HARP platform (Architectures v1.2 and v2.2)

The primary advantage of the machines with FPGAs integrated into CPUs is the ability to address a large memory from the FPGA. The only way hardware architecture on FPGA will be an improvement over the CPU is to hide the memory latency by parallelizing the memory requests. To achieve maximum bandwidth the AFU must have an efficient Fetch Unit that uses the advantages of the HARP platform’s memory access system. The memory requests from the AFU are pipelined. This means that an efficient Fetch Unit must calculate the addresses and make memory requests without waiting for the responses of the previous ones. As mentioned before, the developer can make read and write requests of 1, 2, or 4 CLs. In this section, the architectures were designed for working with requests of four CL (2048 bits) at a time. With these requests, the Fetch Unit makes four times more requests at a time and then waits for the responses. This way of fetching the data exploits better the advantages of the DAER framework and the pipelined request system of HARP. The Process Unit also calculates four times more data in one cycle because of that.

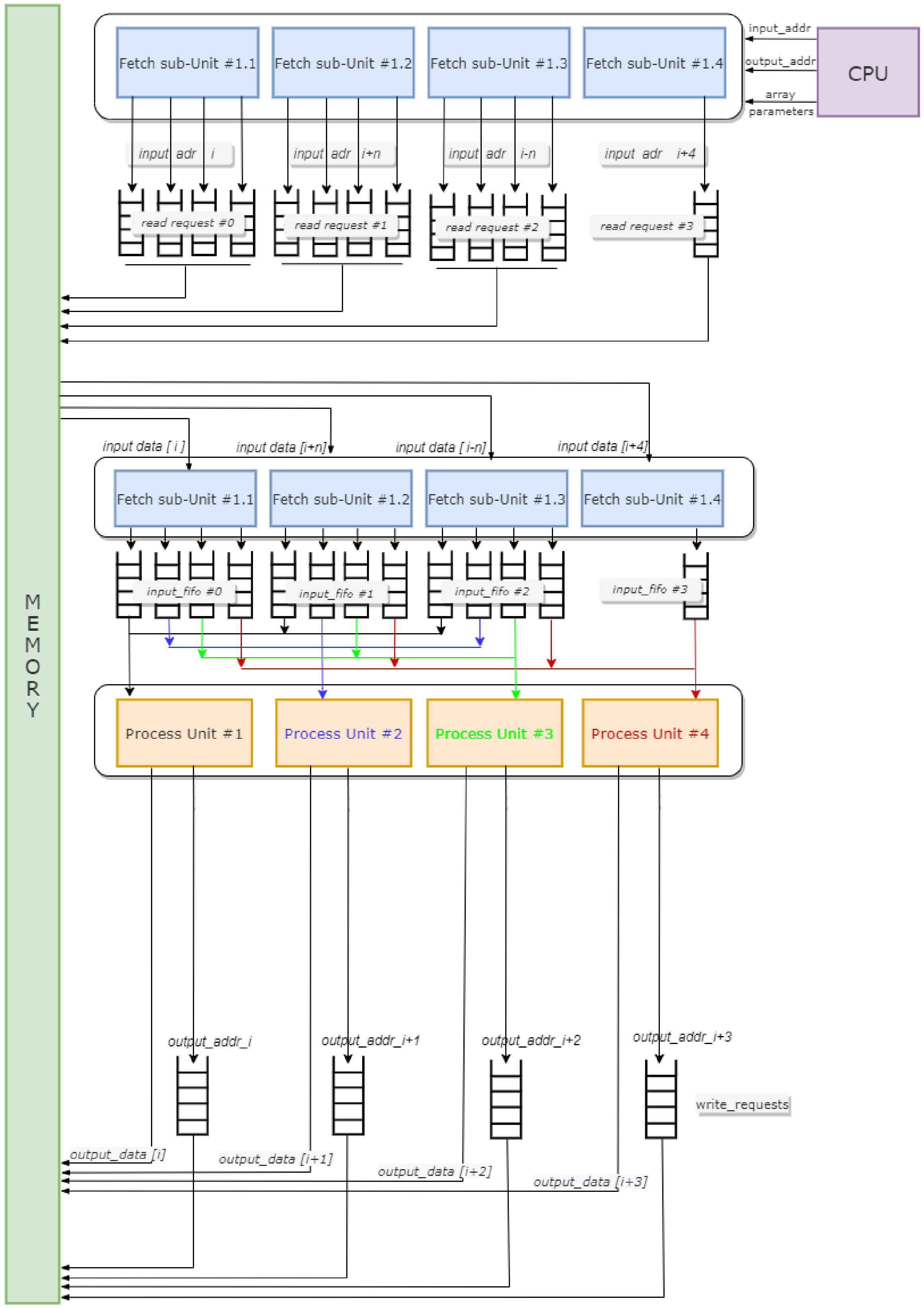

Figure 5.2 Block diagram of the optimized Decoupled Read/Execute architecture v1.2

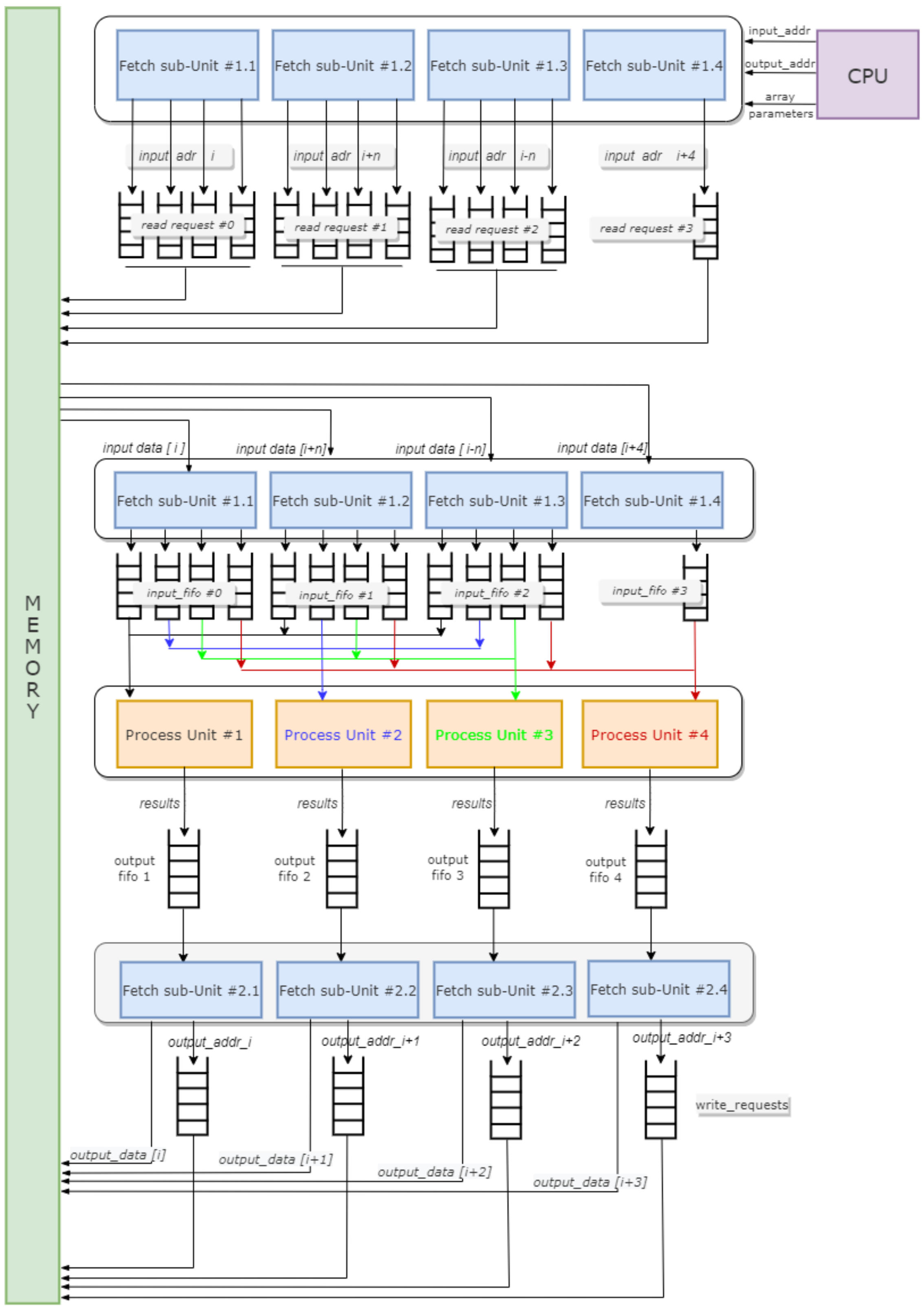

Figure 5.3 Block diagram of the optimized Decoupled Access/Execute architecture v2.2

CHAPTER 6

Software and Hardware Implementations

The SW APIs were written in C and were implemented in the CPU. They are responsible for addressing the HW-side and receiving the results. The Accelerator Functional Units (AFU) were written in System Verilog and were mapped in the FPGA of the platform. In the CPU-based solution, the SW API makes all the processes without addressing an AFU.

6.1 Software Implementation

In this section, there is an extensive description of the SW codes that were run on the CPU of the HARP platform. The CPU is a Broadwell Xeon (E5-2600v4) with 12 cores (24threads) at 1.6-2.2GHz [12].

6.1.1 Software-based solution (CPU)

For the Software-based solution of the Jacobi method, two arrays are needed. The INPUT array has initially all the data that are produced by the rand() function and are needed for the calculations. This array is a $n \times n$ array. Likewise, the RESULT array is the same size as the INPUT. At the end of the calculations, the RESULT has all the updated data and the INPUT has the data of the previous iteration. It is not possible to store the updated values in the INPUT because that will affect the calculations. The process repetitions stop based on a convergence criterion, which is the fixed tolerance ε (epsilon). The criterion is based on the difference of the vertices' values for the two latest iterations. When the calculation of each point of the INPUT table is finished, the difference between the resulting value and its initial value is compared with the ε. If it is greater than the ε, then the above calculations are performed again until the difference is less than the ε. For the next iteration to start, the INPUT array must be updated with the values of the RESULT array.

The SW-based solution was written in C code and was run on the CPU. It's a one-thread solution of the Jacobi algorithm as seen below:

```
while (start_again == 1)
{
        start_again = 0;
        for (i = 1; i < n-1; i++)
        {
                for (j = 1; j < n-1; j++)
                {
                        RESULT[i][j]=(INPUT[i-1][j]+ INPUT [i+1][j]+ INPUT [i][j-1]+ INPUT [i][j+1])/4;
                }
        }
        for (i = 1; i < n-1; i++)
        {
                for (j = 1; j < n-1; j++)
                {
                        if(abs(INPUT [i][j]-RESULT[i][j])>epsilon)
                        {
                                start_again = 1;
                        }
                }
        }
        for (i = 1; i < n-1; i++)
        {
                for (j = 1; j < n-1; j++)
                {
                        INPUT [i][j]=RESULT[i][j];
                }
        }
}
```

6.1.2 Software for hardware accelerator

For an accelerator to work an efficient SW must be designed. As seen in figure 6.1, the SW is responsible for addressing the HW and reading its responses. The application can be implemented for solving several problems, with different array sizes and tolerance ε. Therefore, in the first step, the parameters of the problem are defined. Then the SW is searching for an AFU to connect with. The connection is possible if the ID that the SW reads from the json file is matching with the accelerator's ID. Afterward, the AFU registers are mapped in the user space, and memory space for the shared buffers is allocated. The input buffer is filled with random numbers by using the rand() function. Then the CPU feeds the AFU with the parameters using writes on the MMIO address space. This MMIO write requests contain the address and the sizes of the buffers and the tolerance ε of the application. The SW triggers the accelerator to start with a write on a special register. The SW makes continuous MMIO reads on another register that informs the end of the AFU processes. With other similar MMIO reads the CPU can receive all the responses from the accelerator (cycle counter, rerun, etc.). The SW-side stops when all the data have the desired value difference with the instance of the previous iteration. In general, for the communication between CPU and FPGA, responsible is the SW-side of the design. The application flow of the SW-side is shown in figure 6.1.

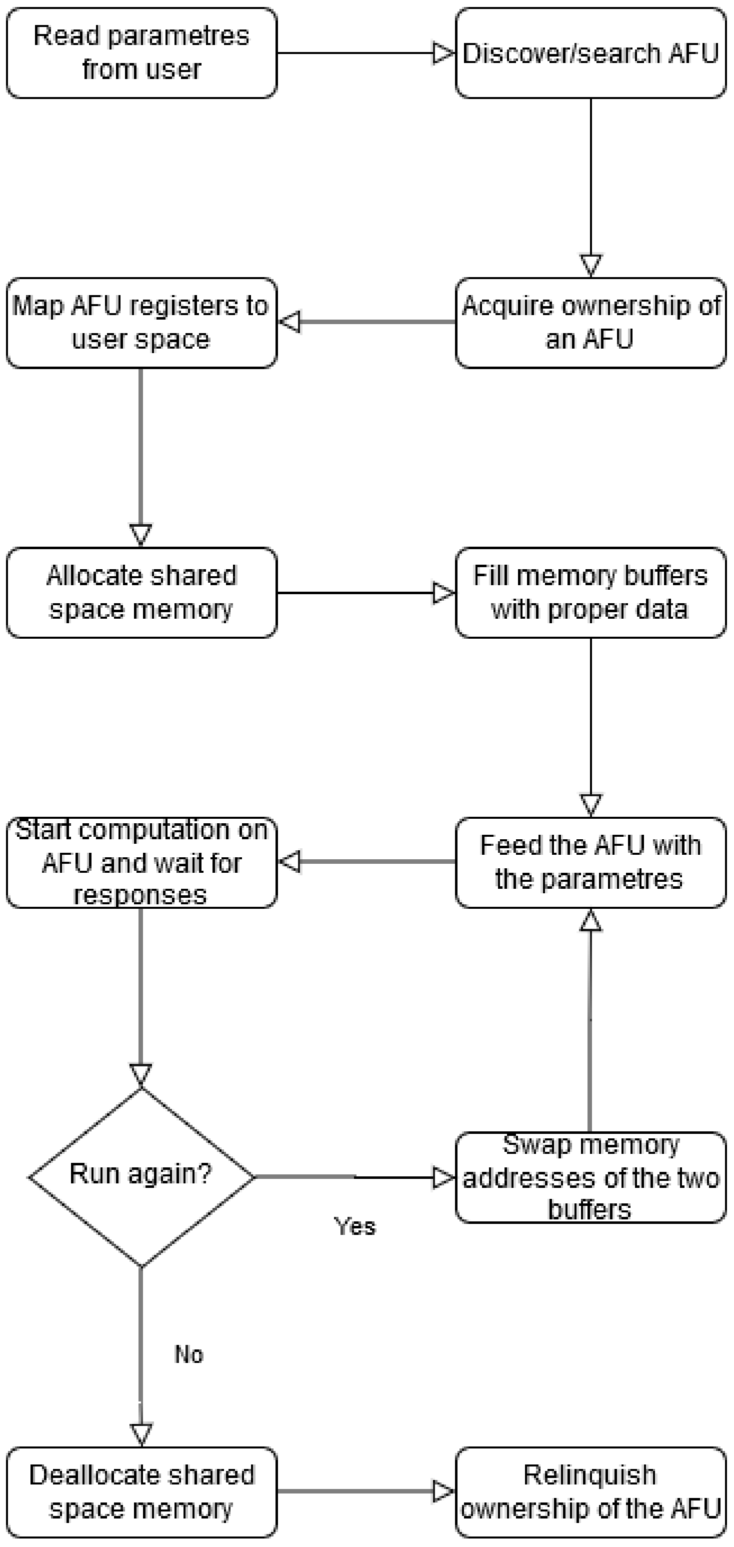
Read parametres from user
Discover/search AFU
Acquire ownership of an AFU
Map AFU registers to user space
Allocate shared space memory
Fill memory buffers with proper data
Feed the AFU with the parametres
Start computation on AFU and wait for responses
Run again?
Yes
Swap memory addresses of the two buffers
No
Deallocate shared space memory
Relinquish ownership of the AFU

Figure 6.1. Application flow of the SW-side

6.1.3 Double buffering

As mentioned before the Jacobi method needs two arrays $n \times n$, one array for the updated values, and one array for the values of the previous iteration. In this way, read address and write address are always different and there aren't any request conflicts that can delay the kernel. The data are fetched from the INPUT array with size $n \times n$. The RESULT array has the same size as the INPUT array. For the next iteration to start, the INPUT must be updated with the values of the RESULT. To avoid that delay, the SW-side of a HARP project swaps the addresses of the two buffers and the AFU uses the RESULT buffer of the previous iteration as INPUT buffer and the INPUT buffer as RESULT buffer alternately. The advantage that this swap of addresses provides is that the SW-side doesn't need to update the buffers because at the end of each iteration the RESULT buffer can be used as input in the next iteration.

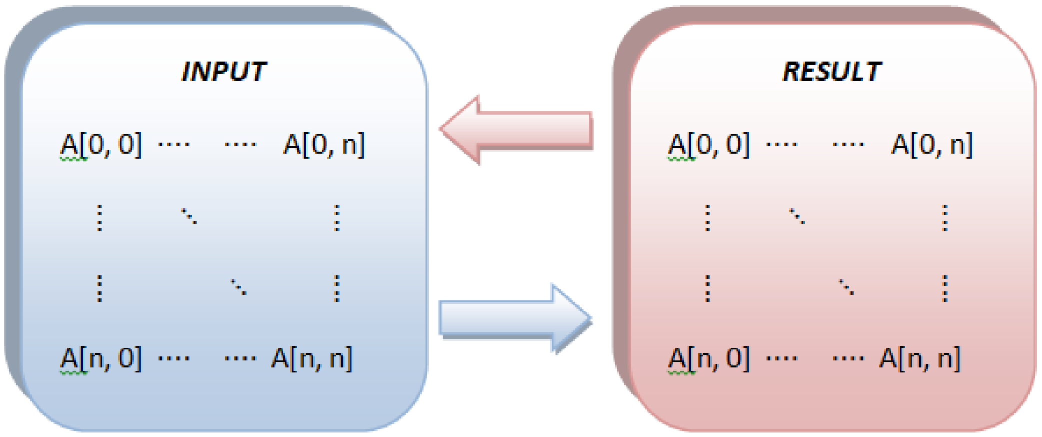
INPUT
A[0, 0] ···· ···· A[0, n]
A[n, 0] ···· ···· A[n, n]
RESULT
A[0, 0] ···· ···· A[0, n]
A[n, 0] ···· ···· A[n, n]

Figure 6.2. Double buffering memory interface

In this version, the Process Unit must decode witch of the data must be calculated. More specifically, the first and the last column and the first and the last line must not be changed by the Process Unit. In a compute-bound application, this can be a delay problem because the execution time of this decode can be proved crucial.

6.1.4 Multi buffering

This method is trying to remove the decode-part from the Process Unit that was mentioned before, by giving to the AFU only the data that are allowed to be updated. This can be achieved if the data needed for the calculations are stored in a different buffer. In this way, the AFU can read the data from the buffers and calculate them without checking if it is acceptable to update them. For this application, five arrays are needed. The first array INPUT has all the initial data and its size is $n \times n$. Instead of creating a copy of this array, the application creates another three arrays with size $n \times (n-2)$. The three extra arrays are:

LEFT: This array has all the values of the left side of A. (A[i,j] for 0<j<n-2)

CENTER: This array has all the values of the center of A. (A[i,j] for 1<j<n-1)

RIGHT: This array has all the values of the right side of A. (A[i,j] for 2<j<n)

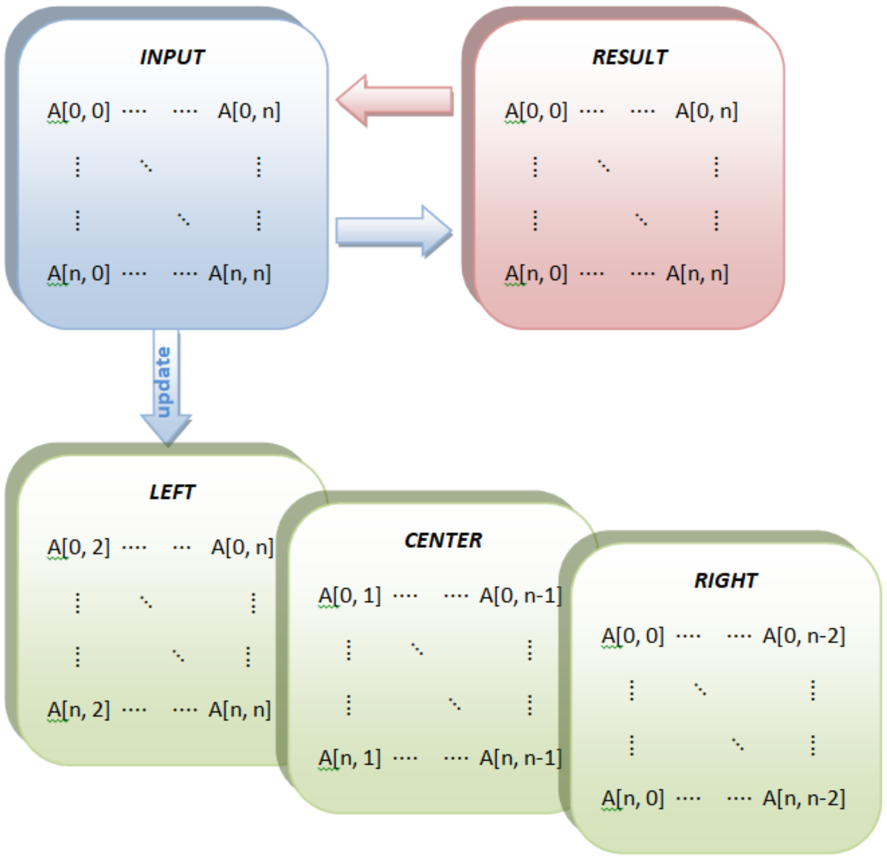
INPUT
A[0, 0] ···· ···· A[0, n]
A[n, 0] ···· ···· A[n, n]
RESULT
A[0, 0] ···· ···· A[0, n]
A[n, 0] ···· ···· A[n, n]
update
LEFT
A[0, 2] ···· ··· A[0, n]
A[n, 2] ···· ···· A[n, n]
CENTER
A[0, 1] ···· ···· A[0, n-1]
A[n, 1] ···· ···· A[n, n-1]
RIGHT
A[0, 0] ···· ···· A[0, n-2]
A[n, 0] ···· ···· A[n, n-2]

Figure 6.3. Multi buffering memory interface

These are the three input buffers of the AFU. In every iteration, the Process Unit calculates the results using the data from the three arrays without making any decoding that can delay the process. The disadvantages of that method compared to the double buffering are two. The first is that, in each iteration, the AFU makes one more read request for fetching the same amount of data. The second is that the SW-side has to update the three input buffers with the data of the RESULT buffer before the next iteration starts. In conclusion, double buffering has a more efficient SW-side and way of fetching the data, and the double buffering a more efficient way of computing.

6.2 Hardware Implementation

In this section, the development of the structural components of the architectures will be analyzed. Every component has a control Finite State Machine (FSM). The necessary information for these FSMs to work is passed from SW to HW through MMIO requests. With these requests, the AFU becomes aware of the addresses of the buffer that will read and write. Also, the components receive orders from the SW with such requests for example when to start/reset. All the architectures have the same 200Mhz clock.

6.2.1 Serial architecture

This architecture has a single FSM control. All the memory requests and the calculations are executed by the same component. As there aren’t any units that work in a parallel way, all the processes are serial. As mentioned before the SW-side is responsible for addressing the HW-side. So in the state IDLE, the AFU is waiting for a start signal from the SW. When that signal arrives the AFU makes four read requests. When read responses arrive, the AFU performs all the calculations. As soon as the results are ready, they are written back to the memory. All these procedures are serial and the read-part of the design is very slow. A memory request can’t be issued until the response of the previous request arrives.

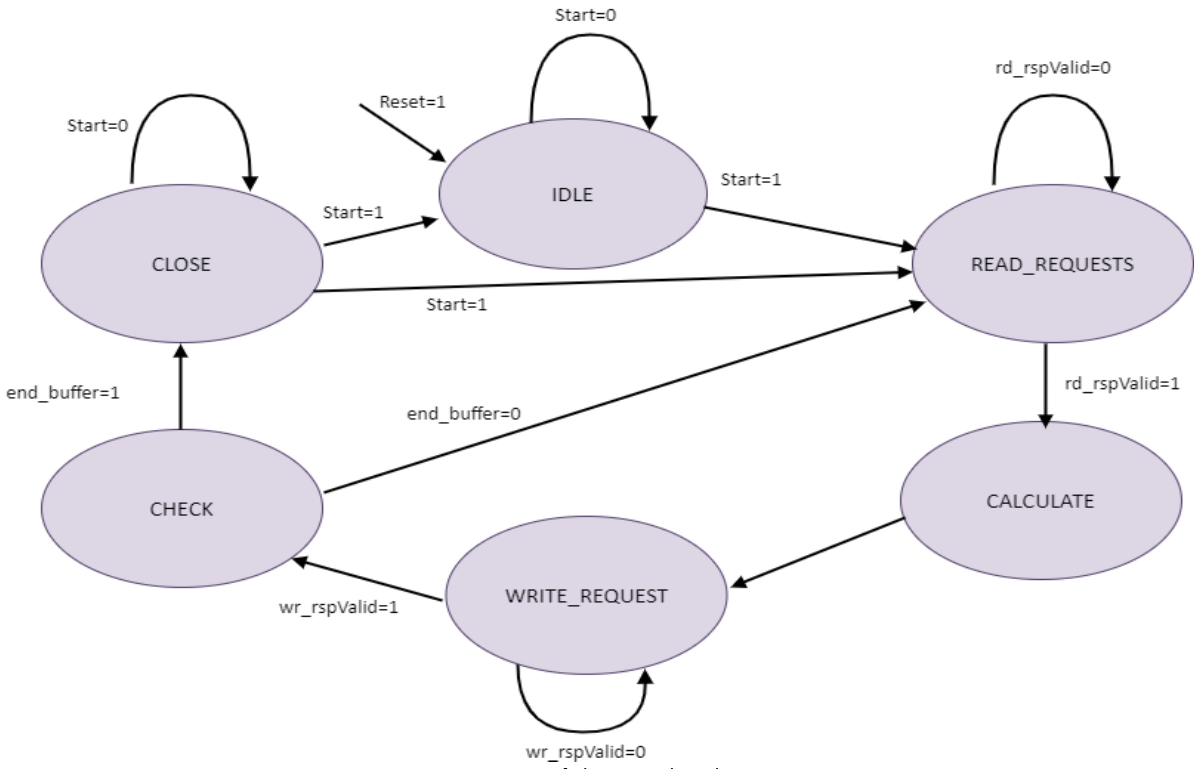

Figure 6.4. FSM of the Serial architecture

6.2.2 Decoupled Read/Execute architecture (Architecture v1)

For this architecture were designed two FSMs. The first FSM controls the Fetch Unit that reads all the data from the memory. The second FSM controls the Process Unit that calculates the results and writes them back to the memory. These to FSMs are parallel.

Fetch Unit FSM:

Initially, the Fetch Unit waits for the start signal from the SW and then jumps to the read-part. In that part, the Fetch Unit checks if Channel 0 is busy by using the sRx.c0TxAlmFull signal and issues read requests of it the channel is free. After issuing the read requests, the AFU has to wait for all the CLs to be received. The responses may come out of order, so with the use of the mdata (id of the request) and cl_num (id of the CL), the Fetch unit can reorder the responses and store them in the FIFOs. Then in CHECK_FIFO_FULL state, it checks if the FIFOs have space for issuing more requests with the signal fifo_alm_full. This unit stops making requests when the read buffer is exceeded.

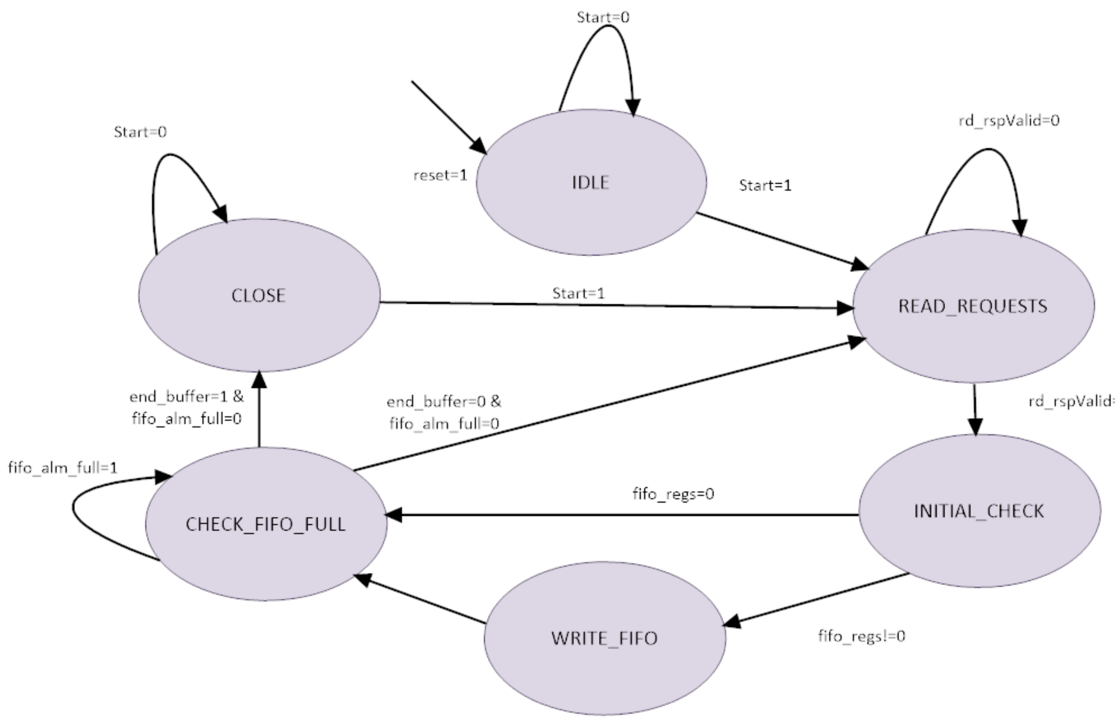

Figure 6.5. Fetch unit FSM of the Decoupled Read/Execute architecture v1.0

<u>Process Unit FSM</u>:

Initially, the Process Unit waits until the FIFO's aren't empty. When they aren't, the four read enable signals are set. Then, the four groups of CLs are ready for the calculations. In the CALCULATE state, the outputs of the four FIFOs are calculated with multiple controls. The first result (64bits) of the first CL (512bits) and the last result of the last CL stay the same. In every other case, all the results are calculated with the Jacobi formula. In the register "previous" the last node (64bits) from this run is saved. In the next run, the "previous" register will be used as the left node of the first node. In this version, this unit is responsible for writing the results to the memory before starting to calculate the next results. In every run of the FSM the CHECK_FIFO_EMPTY state, checks if the write buffer is exceeded for ending all the processes. If not, it checks if the FIFOs are empty and wait until they are not so it can proceed to the next reads. When this FSM stops the HW-side sends a signal to the SW API to inform that the application has ended.

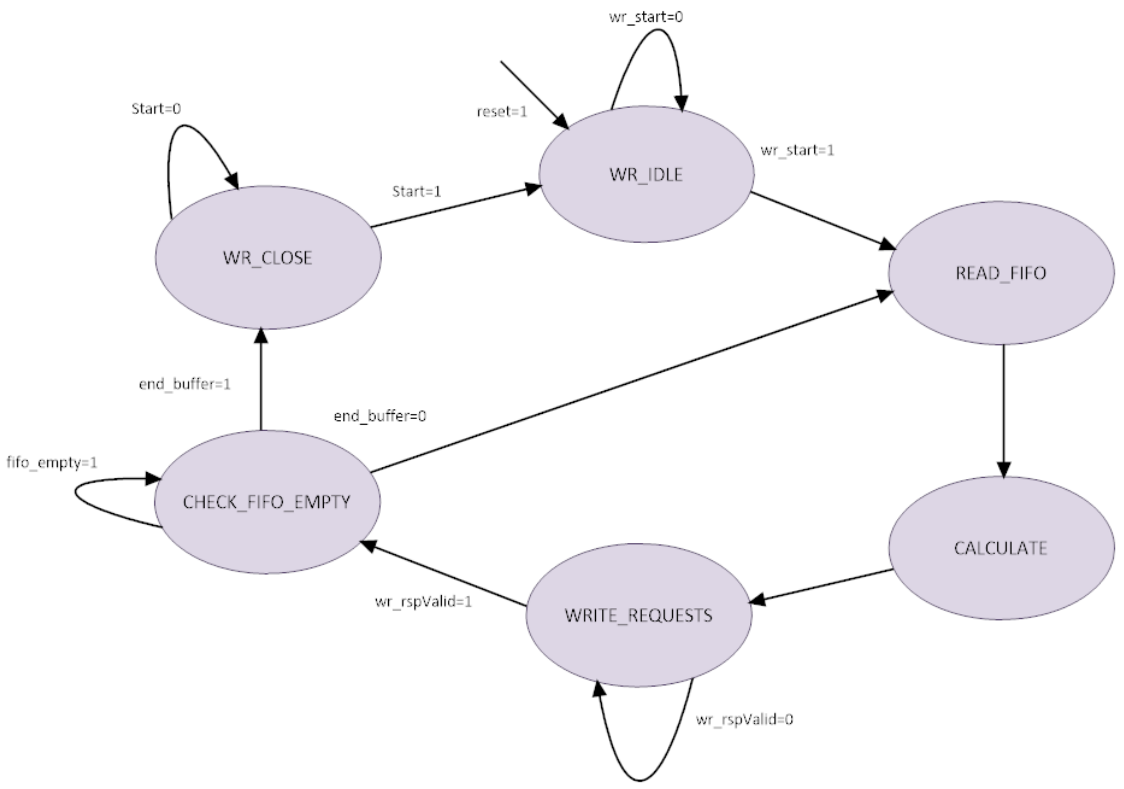

Figure 6.5. Process unit FSM of the Decoupled Read/Execute architecture v1.0

6.2.3 Decoupled Access/Execute architecture (Architecture v2)

In this architecture, the FSM of the Process Unit does not write the results to the DDR. The Fetch Unit has two sub-units. The FSM of Read sub-Unit reads from the DDR and the Write sub-Unit writes in the DDR the results. The Process Unit only calculates the results and writes them in a FIFO. Therefore the Process Unit isn't waiting for the response of the write request and proceeds to the next calculation.

Read sub-Unit FSM:

This sub-unit has the same FSM with the Fetch Unit of the previous architecture (Section 6.2.2).

Process Unit FSM:

In this version, the Process Unit calculates the result in the same way that the previous architecture did. The difference is that the results are written in a FIFO and the FSM can proceed to the next reads without waiting for memory writes. The CHECK_FIFO state must check both input and output FIFOs. The input FIFOs must have data end the output FIFO must not be full to proceed in the next FIFO reads.

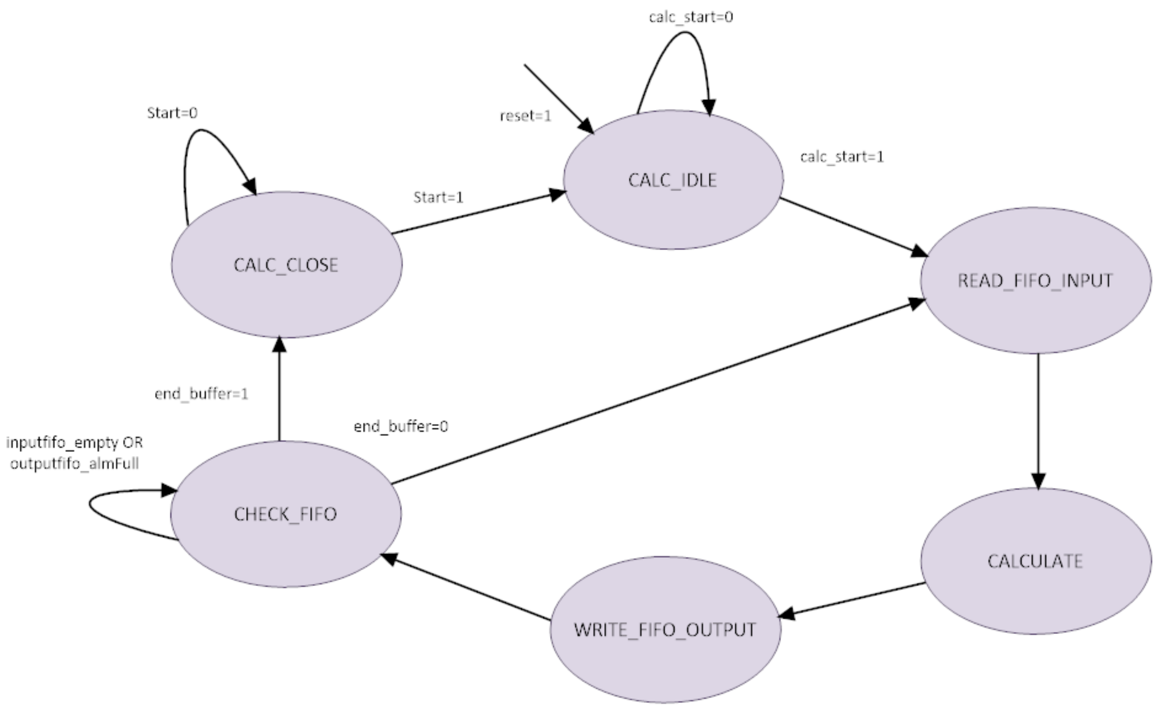

Figure 6.6. Process unit FSM of the Decoupled Access/Execute architecture v2.0

<u>Write sub-Unit FSM</u>:

This is the second sub-unit of the Fetch Unit. This unit checks if there are data in the output FIFO and then writes them in the memory. Before checking the FIFO for more writes, checks if the write buffer is exceeded. When this FSM stops the HW-side sends a signal to the SW API to inform that the application has ended.

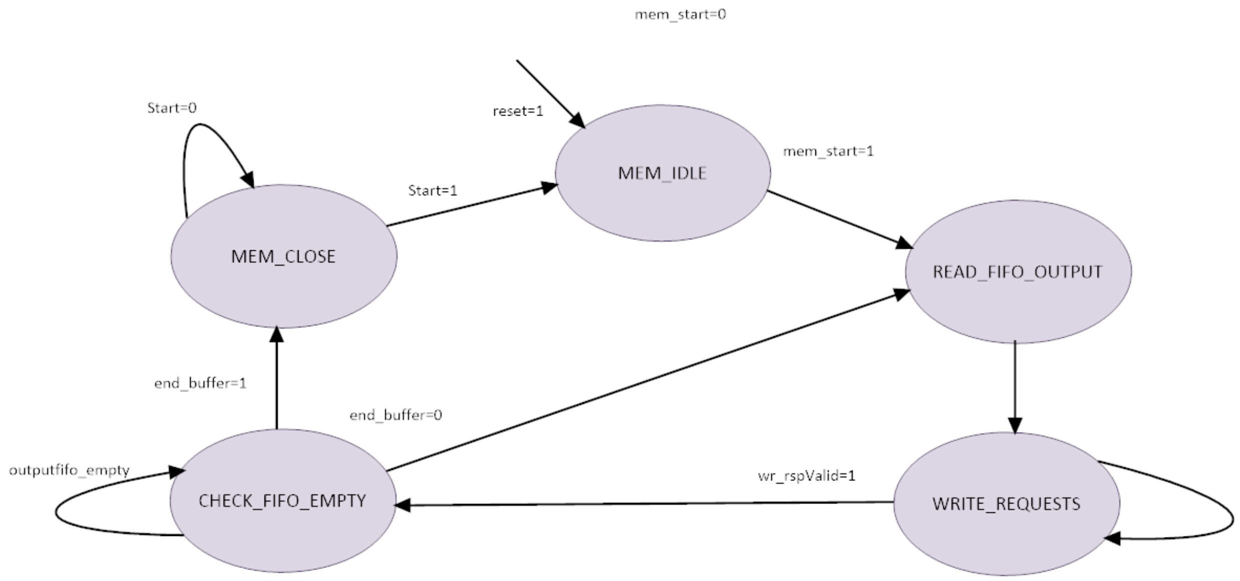

Figure 6.7. Write sub-unit FSM of the Decoupled Access/Execute architecture v2.0

6.2.4 Memory access with CL1 and CL4 requests

As mentioned before, the developer can make read and write requests of 1, 2, or 4 CLs. In this section, all the hardware implementations are tested with requests of CL1 (512 bits) and CL4 (2048 bits) at a time. When the CL4 requests are used the architectures are optimized as shown in Section 5.4.

<u>CL1 requests:</u>

For one calculation the Fetch Unit makes four CL1 read requests (one CL at a time) from different addresses of the DDR. In indexes [i-1,j], [i,j], [i+1,j] [i,j+1] the AFU issue read requests and then stores them in four FIFOs. The Process Unit reads the four FIFOS and uses the data to make the calculations. It uses all the data from the FIFOs [i-1,j], [i,j], [i+1,j], the first number of the FIFO [i,j+1] and the register with the name previous that contains the last number of the FIFO [i,j] of the previous iteration, as shown in Figure 6.9. The CL1 write requests are issued in the index [i,j] of the result buffer.

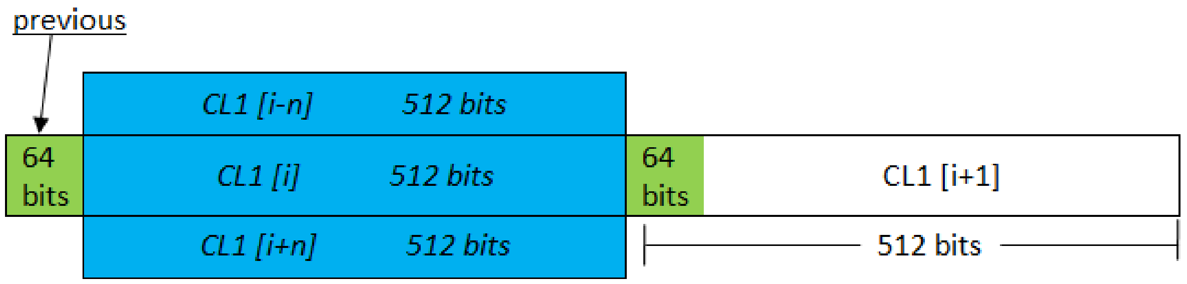

Figure 6.8. Fetching data with CL1 requests in the Jacobi method

<u>CL4 requests:</u>

The AFU makes 3 CL4 read requests and 1 CL1 read request. With these requests and the register with the name "previous" can make all the calculations, as shown in Figure 6.10. The CL4 write requests are issued in the indexes [i,j], [i,j+1], [i,j+2], [i,j+3] of the result buffer.

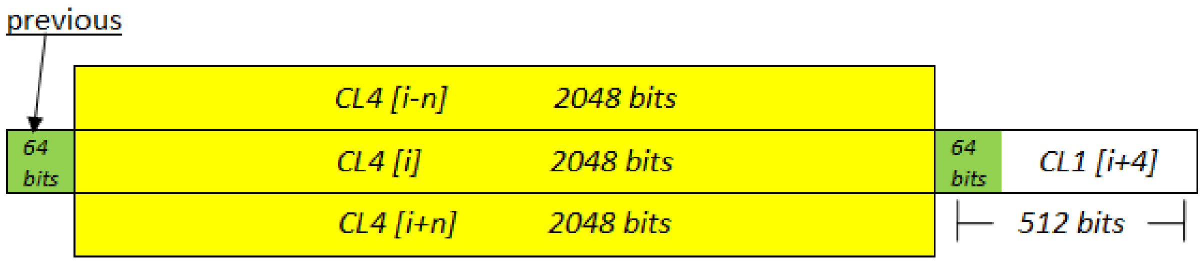

Figure 6.9. Fetching data with CL4 requests in the Jacobi method

CHAPTER 7

Results and Discussion

In this chapter, there is an extensive description and analysis of all the experiments that were conducted, with their results. These experiments were conducted in the Academic Compute Environment (ACE) that is located on the vLabs of Intel. This environment allows the developers to run their designs on CPU-FPGA chips in real conditions.

7.1 Dataset

For each run, the software that creates the INPUT array takes as variables:

• The array size $(n \times n)$	n
• The tolerance of the iterative algorithm (ε)	e
• The range of the values ({1, 10}, {1, 100}, {1, 1000} etc.)	w

All the tests had the same dataset for a better comparison of the results. The dataset was created with the rand() function and for each size of array $(n \times n)$ the elements stay the same. The tolerance of the iterative algorithm e and the range of the values w determine the number of iterations for the same size of data. When the size of the table $(n \times n)$ is changed, the amount of calculated data will be changed as well.

7.2 Software-based solution (CPU)

The experiments for the CPU version of the application were conducted with the same dataset as the hardware architectures, for better comparison. The frequency of the CPU is at 2.2GHz and the experiments were conducted in one thread. The results of the CPU version with $e = 300$ and $w = \{1, 1000\}$ are shown in table 7.1.

Table 7.1. Results of the Software-based solution (CPU)

Table size (n x n)	Execution time (ms)	Iterations	Execution time/Iterations (ms)
32 x 32	0.25	5	0.05
64 x 64	1.07	7	0.15
128 x 128	5.91	10	0.59
256 x 256	29.70	14	2.12
512 x 512	73.50	10	7.35
1024 x 1024	383.50	15	25.57

Then, for the same array sizes, were conducted experiments with different tolerance ε to examine the behavior of the CPU with a different number of iterations. The processing rate of the experiments with runs of one and ten iterations are shown in table 7.2. For these experiments, were measured the number of elements in (MB) that were processed by the algorithm per second.

Table 7.2. Processing rate of Software-based solution for different number of iterations

Table size (n x n)	Processing rate Iterations=1 (MB/s)	Processing rate Iterations=10 (MB/s)
32 x 32	1.1	3.1
64 x 64	2.2	3.2
128 x 128	2.7	3.3
256 x 256	2.7	3.6
512 x 512	3.4	4.4
1024 x 1024	4.4	4.7

When the application runs for multiple iterations the processing rate is bigger, because the CPU makes more hits in the cache when it reads the same cache lines repeatedly. When it reads them in the first iteration there are no useful data in the cache and the CPU makes more misses.

7.3 Efficiency of the SW-side

In this section, the two versions of the SW-side that run on the CPU and address the FPGA will be compared. In table 7.3, there is a comparison between the Double buffering and the Multi buffering version. It's clear that for this application the double buffering is way more efficient SW than the Multi buffering. The memory computations that the Multi buffering SW makes in every iteration to update the extra three buffers have a big impact on the execution time. Also, the SW of the Multi buffering allocates way more space. As expected, these results prove that in a memory-bound application like the Jacobi the double buffering is a better solution for feeding the AFU.

Table 7.3. Comparison of the Double buffering with the Multi buffering

Table size (n x n)	**Double buffering (ms/iteration)**	**Multi buffering (ms/iteration)**
32 x 32	0.08	0.17
64 x 64	0.25	0.59
128 x 128	1.00	2.20
256 x 256	3.97	8.41
512 x 512	15.39	33.07
1024 x 1024	59.76	131.46

7.4 Serial architecture

This is the first hardware architecture that was tested in the vLab's CPU-FPGA chip. It is created to examine how efficient is a hardware architecture that works without parallel units and will be compared later with the parallel architectures. The results of this architecture are shown in the following tables. As expected, an architecture that works in 200MHz and doesn't exploit the advantages of the HARP platform or making parallel calculations, it is difficult to beat a CPU that works in 2.2GHz. In that design, the processing rate is approximately 0.5MB/s. That didn't change when the design runs for many iterations because the cache of the AFU is very small and direct-mapped unlike the big cache of the CPU. So the best way to compete with the CPU in a memory-bound application like Jacobi is by creating a design that hits the maximum bandwidth of the platform.

Table 7.4. Results of the Serial architecture

Table size (n x n)	Execution time (ms)	Iterations	Execution time/Iterations (ms)
32 x 32	1.50	5	0.30
64 x 64	7.70	7	1.10
128 x 128	43.10	10	4.31
256 x 256	221.20	14	15.80
512 x 512	617.50	10	61.75
1024 x 1024	3604.90	15	240.33

Table 7.5. Comparison of the Serial architecture with the CPU

Table size (n x n)	Serial (ms/iteration)	CPU (ms/iteration)
32 x 32	0.08	0.17
64 x 64	0.25	0.59
128 x 128	1.00	2.20
256 x 256	3.97	8.41
512 x 512	15.39	33.07
1024 x 1024	59.76	131.46

In figure 7.1 the cycles that need every step of the serial implementation can be seen. This figure shows that the read-part that needs more improvement is the read-part.

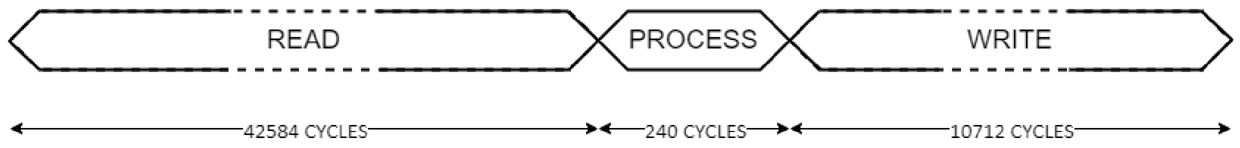

Figure 7.1. Time diagram of Serial architecture

7.5 Decoupled Read/Execute architecture

In this section, are presented the results of the DRER architecture with both CL1 and CL4 memory requests. As mentioned before the CL4 memory requests give the opportunity for optimization of the architecture. This architecture has the advantage of making the reads independently from the other processes of the AFU. This independence allows further optimizations of the read-part and it is easier to solve the problem of the slow reads. The results will be compared with the Serial architecture and the CPU results.

7.5.1 DRER with CL1 memory requests (Architecture v1.1)

In the following tables, the results of the DRER architecture with CL1 memory requests are presented. Comparing to the Serial architecture there is speed up because that way of reading is hiding one part of the delay from the slow read request system of the application. On the other hand, it has a worse processing rate compared to the CPU and it will be needed a more optimized architecture.

Table 7.6. Results of the DRER CL1 architecture v1.1

Table size (n x n)	Execution time (ms)	Iterations	Execution time/Iterations (ms)
32 x 32	0.39	5	0.08
64 x 64	1.77	7	0.25
128 x 128	9.96	10	1.00
256 x 256	55.6	14	3.97
512 x 512	153.9	10	15.39
1024 x 1024	896.35	15	59.76

Table 7.7. Comparison of the DRER CL1 architecture v1.1 with the CPU and the Serial architecture

Table size (n x n)	DRER CL1 (v1.1) (ms/iteration)	Serial (ms/iteration)	CPU (ms/iteration)
32 x 32	0.08	0.30	0.05
64 x 64	0.25	1.10	0.15
128 x 128	1.00	4.31	0.59
256 x 256	3.97	15.80	2.12
512 x 512	15.39	61.75	7.35
1024 x 1024	59.76	240.33	25.57

Table 7.8. Speedup of the DRER CL1 architecture v1.1 over the Serial architecture and the CPU

Table size (n x n)	Serial	CPU
32 x 32	3.8x	0.6x
64 x 64	4.4x	0.6x
128 x 128	4.3x	0.6x
256 x 256	4.0x	0.5x
512 x 512	4.0x	0.5x
1024 x 1024	4.0x	0.4x

7.5.2 DRER with CL4 memory requests (Architecture v1.2)

In the following tables, the results of the DRER architecture with CL4 memory requests are presented. Comparing to the CL1 memory requests there is a speedup because of the way that the CL4 requests exploit the advantages of the pipelined memory access system of HARP. Also achieves acceleration over the CPU as shown in figure 7.2.

Table 7.9. Results of the DRER CL4 architecture v1.2

Table size (n x n)	Execution time (ms)	Iterations	Execution time/Iterations (ms)
32 x 32	0.14	5	0.03
64 x 64	0.76	7	0.11
128 x 128	4.20	10	0.42
256 x 256	23.06	14	1.65
512 x 512	64.40	10	6.44
1024 x 1024	377.62	15	25.17

Table 7.10. Comparison of the DRER CL4 architecture v1.2 with the CPU and the DRER CL1 architecture v1.1

Table size (n x n)	DRER CL4 (v1.2) (ms/iteration)	DRER CL1 (v1.1) (ms/iteration)	CPU (ms/iteration)
32 x 32	0.03	0.08	0.05
64 x 64	0.11	0.25	0.15
128 x 128	0.42	1.00	0.59
256 x 256	1.65	3.97	2.12
512 x 512	6.44	15.39	7.35
1024 x 1024	25.17	59.76	25.57

Table 7.11. Speedup of the DRER CL4 architecture v1.2 over the CPU and DRER CL1 architecture v1.1

Table size (n x n)	DRER CL1 (v1.1)	CPU
32 x 32	2.8x	1.8x
64 x 64	2.3x	1.4x
128 x 128	2.4x	1.4x
256 x 256	2.4x	1.3x
512 x 512	2.4x	1.1x
1024 x 1024	2.4x	1.0x

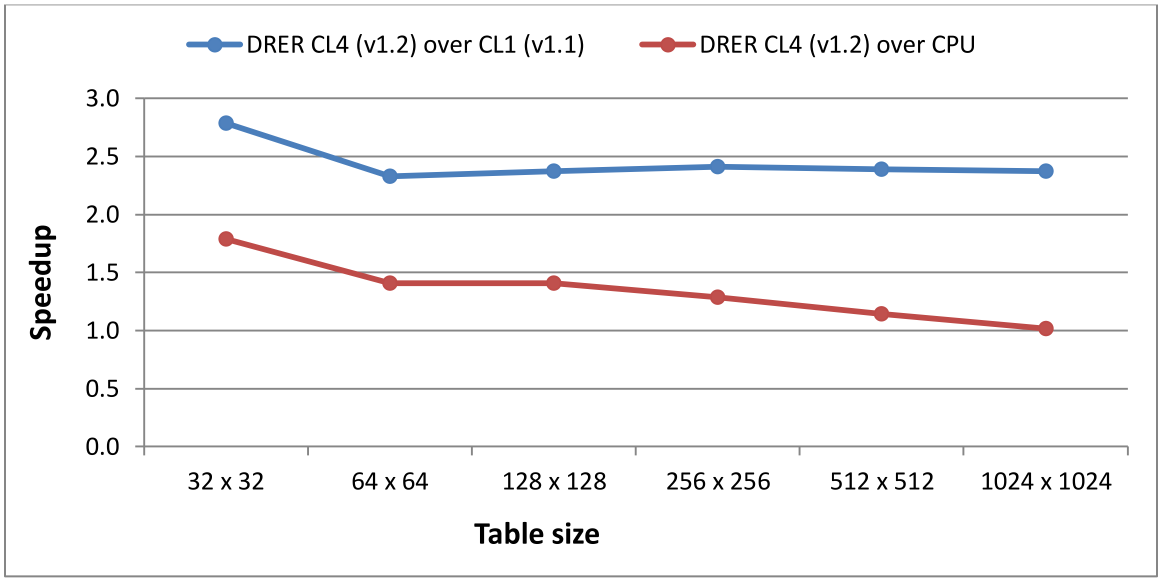

Figure 7.2. Speedup of the DRER CL4 architecture v1.2 over the CPU and DRER CL1 architecture v1.1

7.6 Decoupled Access/Execute architecture

In this section, are presented the results of the DAER architecture with both CL1 and CL4 memory requests. The results will be compared with the Serial architecture and the CPU results. The extra advantage of the DAER architecture over the DRER is the parallelization of the writes.

7.6.1 DAER with CL1 memory requests (Architecture v2.1)

In the following tables, the results of the DAER architecture with CL1 memory requests are presented. Comparing to the Serial architecture there is speed up and compared to the CPU has a worse processing rate for all the experiments.

Table 7.12. Results of the DAER CL1 architecture v2.1

Table size (n x n)	Execution time (ms)	Iterations	Execution time/Iterations (ms)
32 x 32	0.35	5	0.07
64 x 64	1.77	7	0.25
128 x 128	9.92	10	0.99
256 x 256	55.60	14	3.97
512 x 512	153.60	10	15.36
1024 x 1024	895.70	15	59.71

Table 7.13. Comparison of the DAER CL1 architecture v2.1 with the CPU and the Serial architecture

Table size (n x n)	DAER CL1 (v2.1) (ms/iteration)	Serial (ms/iteration)	CPU (ms/iteration)
32 x 32	0.07	0.30	0.05
64 x 64	0.25	1.10	0.15
128 x 128	0.99	4.31	0.59
256 x 256	3.97	15.80	2.12
512 x 512	15.36	61.75	7.35
1024 x 1024	59.71	240.33	25.57

Table 7.14. Speedup of the DAER CL1 architecture v2.1 over the Serial architecture and the CPU

Table size (n x n)	Serial	CPU
32 x 32	3.8x	0.6x
64 x 64	4.4x	0.6x
128 x 128	4.3x	0.6x
256 x 256	4.0x	0.5x
512 x 512	4.0x	0.5x
1024 x 1024	4.0x	0.4x

7.6.2 DAER with CL4 memory requests (Architecture v2.2)

In the following tables, the results of the DAER architecture with CL4 memory requests are presented. The two architectures behave in a similar way to the application's demands. Thus, the results of the DAER CL4 version achieve a similar speedup over the CL1 version and the CPU with the speedup that the results of the DRER CL4 version achieve, as shown in figure 7.3. This similarity in the results will be analyzed in section 7.7.

Table 7.15. Results of the DAER CL4 architecture v2.2

Table size (n x n)	Execution time (ms)	Iterations	Execution time/Iterations (ms)
32 x 32	0.12	5	0.02
64 x 64	0.76	7	0.11
128 x 128	4.15	10	0.42
256 x 256	22.90	14	1.64
512 x 512	64.20	10	6.42
1024 x 1024	374.80	15	24.99

Table 7.16. Comparison of the DAER CL4 architecture v2.2 with the CPU and the DRER CL1 architecture v2.1

Table size (n x n)	DAER CL4 (v2.2) (ms/iteration)	DAER CL1 (v2.1) (ms/iteration)	CPU (ms/iteration)
32 x 32	0.02	0.07	0.05
64 x 64	0.11	0.25	0.15
128 x 128	0.42	0.99	0.59
256 x 256	1.64	3.97	2.12
512 x 512	6.42	15.36	7.35
1024 x 1024	24.99	59.71	25.57

Table 7.17. Speedup of the DAER CL4 architecture v2.2 over the CPU and DRER CL1 architecture v2.1

Table size (n x n)	DAER CL1 (v2.1)	CPU
32 x 32	3.2x	2.1x
64 x 64	2.3x	1.4x
128 x 128	2.4x	1.4x
256 x 256	2.4x	1.3x
512 x 512	2.4x	1.1x
1024 x 1024	2.4x	1.0x

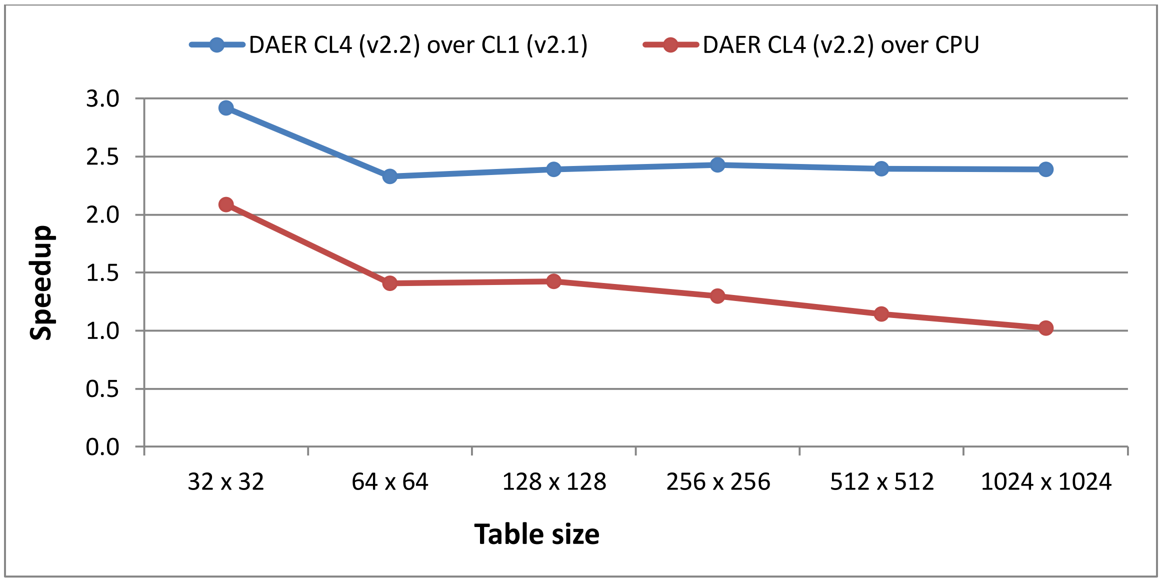

Figure 7.3 Speedup of the DAER CL4 architecture v2.2 over the CPU and DRER CL1 architecture v2.1

7.7 Overall comparison of the results

In the following figures 7.4, 7.5, the speedups that the two parallel architectures achieve over the Serial architecture are presented. As can be seen, both architectures achieve acceleration 4x over the Serial with the use of CL1 memory requests, but they manage 10x acceleration with CL4 requests. That proves that in this framework multiple requests must be used to hide more latency.

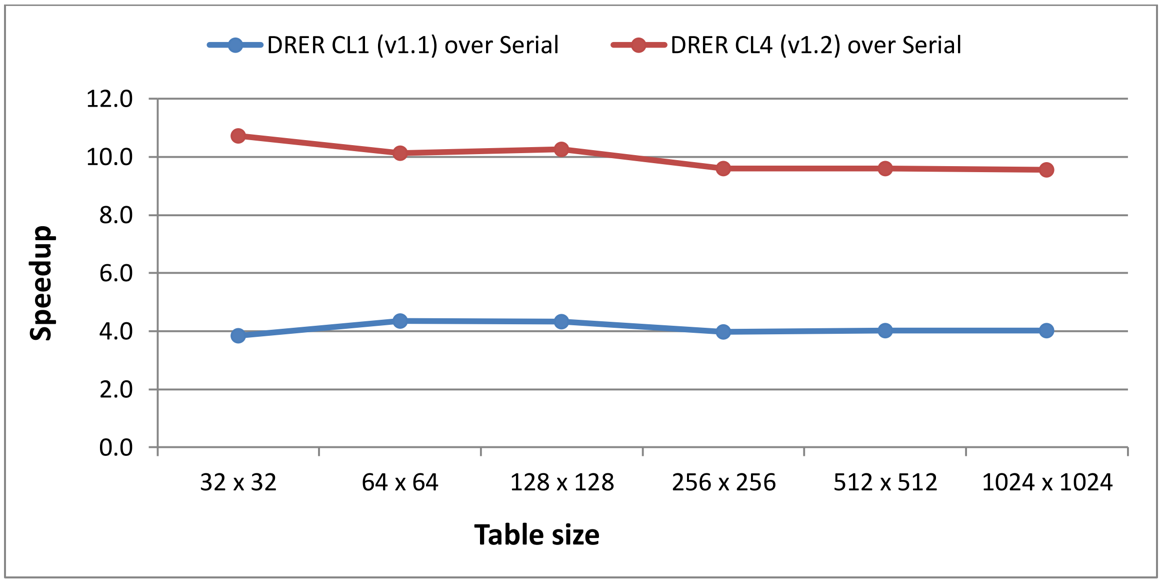

Figure 7.4. Speedup of the v1.1 and the v1.2 DRER architectures over the Serial architecture

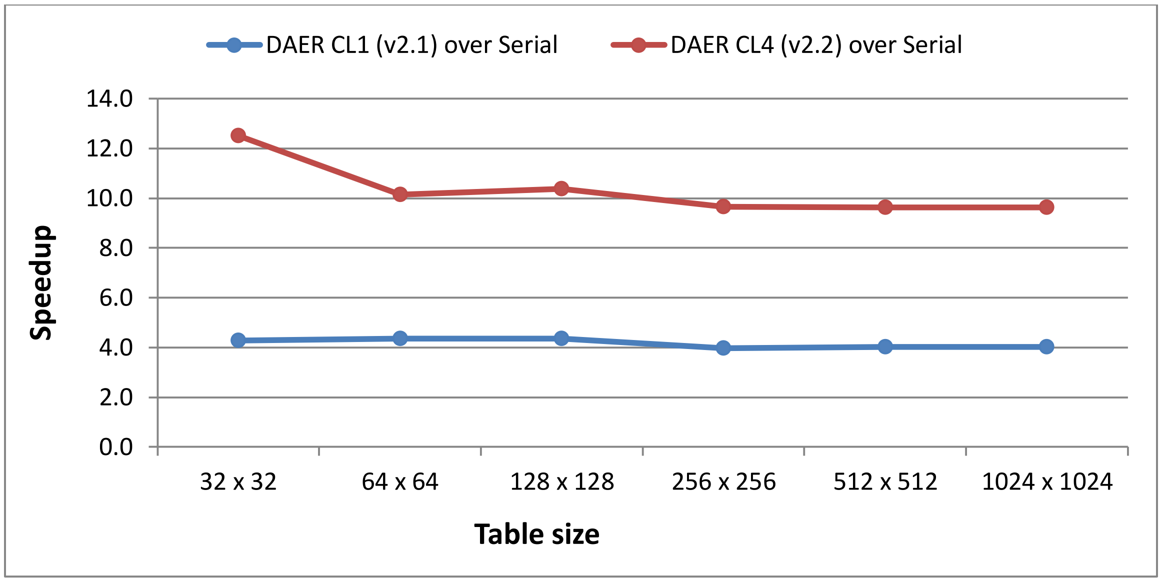

Figure 7.5. Speedup of the v2.1 and the v2.2 DAER architectures over the Serial architecture

In figures 7.6, 7.7 the speedups of the two architectures over the CPU are presented. Only the optimization with the CL4 requests leads to acceleration over the CPU. As mentioned before the CPU has a bigger and more effective cache than the small direct-mapped cache of the FPGA. That makes the CPU more and more effective as the iterations multiply.

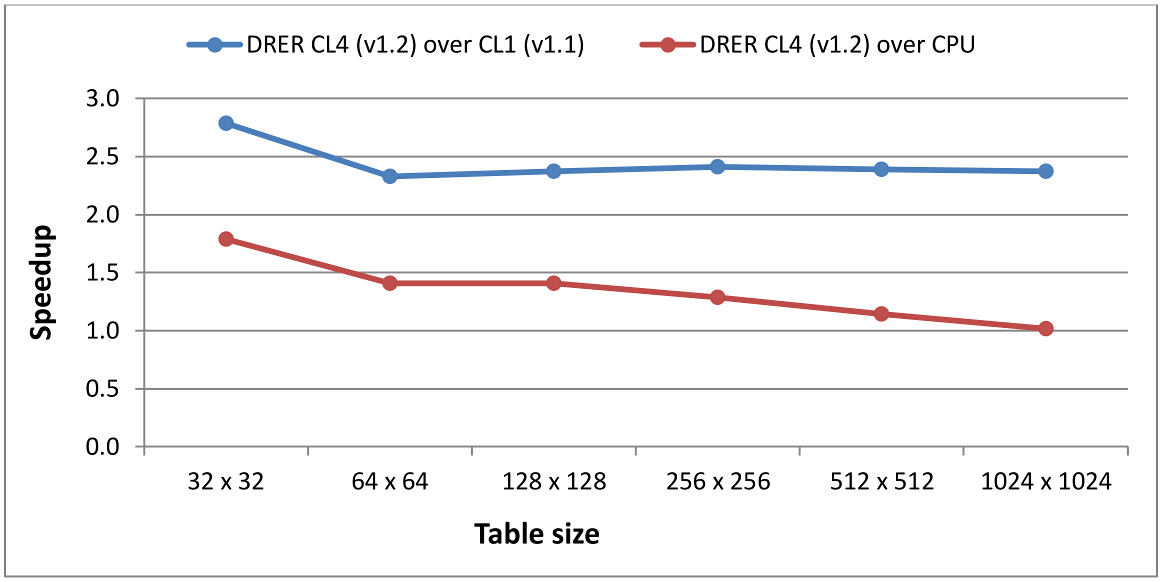

Figure 7.6. Speedup of the v1.1 and the v1.2 DRER architectures over the CPU

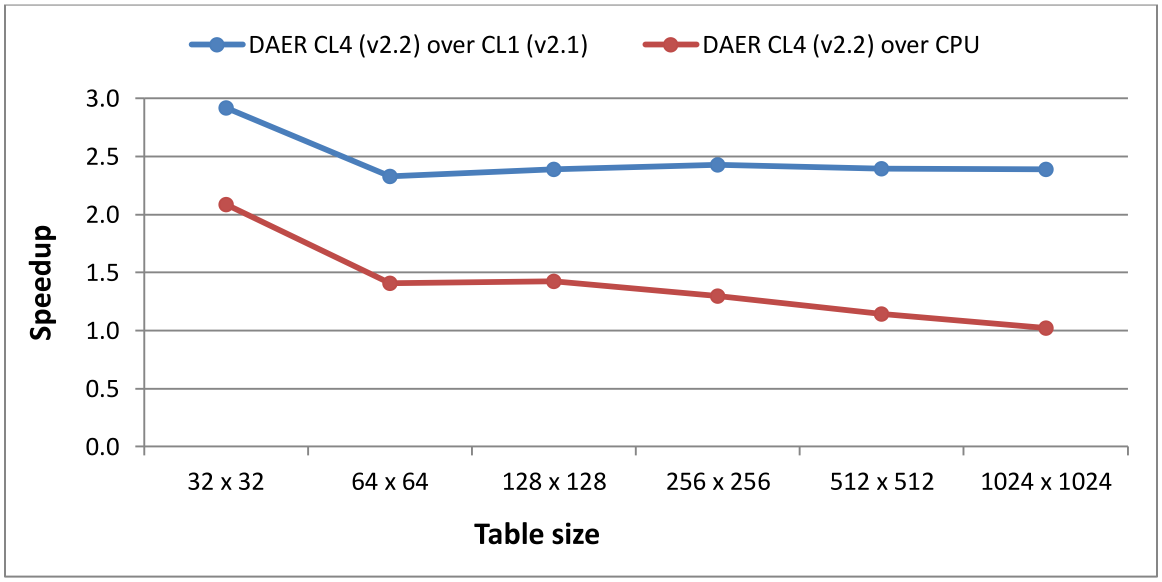

Figure 7.7. Speedup of the v2.1 and the v2.2 DAER architectures over the CPU

As shown in the time diagram of the Serial architecture (figure 7.1), the read-part of the application is the slowest part. With the DRER and the DAER architectures, there is speedup over that first attempt, but the read-part continuous to be the slowest part. In all the hardware implementations, the read-part determines the execution time. Because of that, the parallelization between the write-part and the calculation-part, that the DAER architecture provides over the DRER architecture, can't be seen clearly in the results. Finally, in figure 7.8 the processing rates of all the implementations are presented for better comparison and evaluation of this thesis's work.

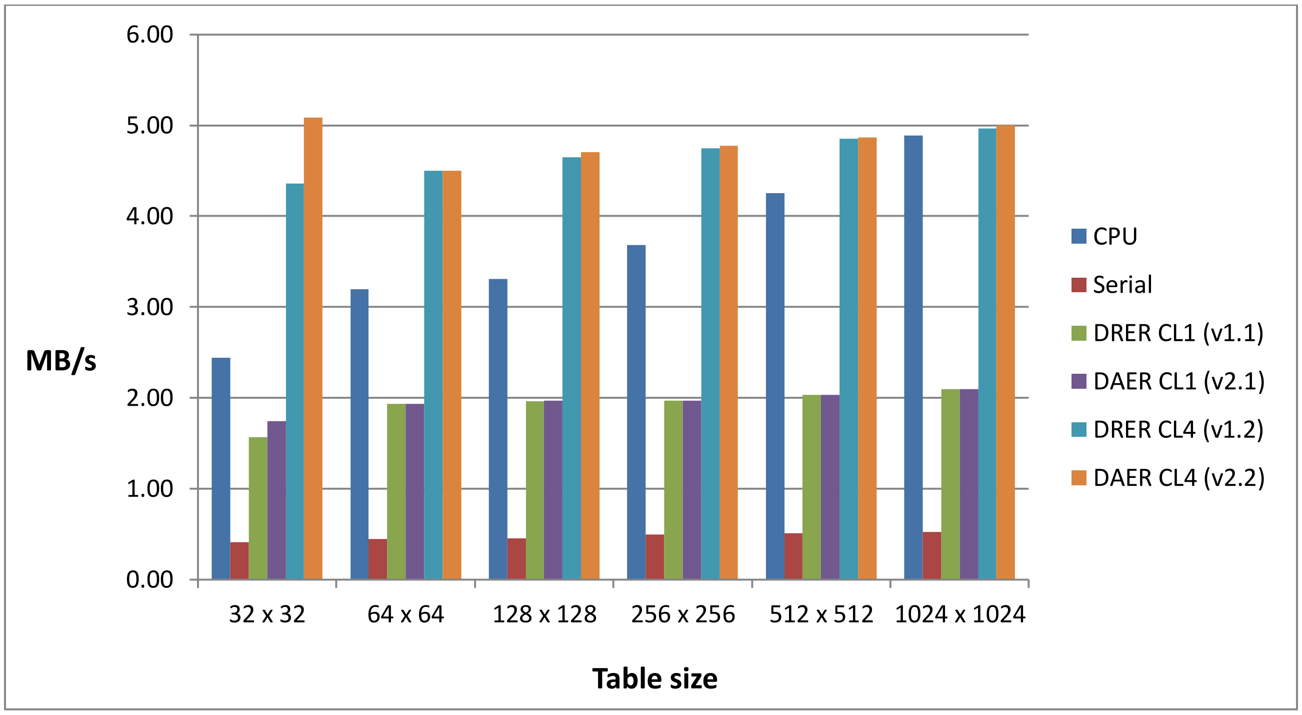

Figure 7.8. Overall comparison of the results

CHAPTER 8

Conclusions and Future work

8.1 Conclusions

This thesis aims to map the DAER framework and implement it on the HARP with an application suitable for testing HPC platforms. The algorithm that implemented was the Jacobi Method. The best way to fetch the data is with double buffering, based on DDR usage and software interference of this method. In this work, there is an extensive description of all the steps toward an optimized DAER architecture. From the results, it can be seen that the two parallel architectures (DRER and DAER) can achieve speedup over the CPU. The more these architectures exploit the advantages of the DAER framework by prefetching more data the more the performance improves. That can be seen by the speedup that the CL4 requests achieve over the CL1 requests. This leads to the conclusion that the memory access system must be improved furthermore to achieve maximum bandwidth. That can be achieved if the AFU waits for more responses of "in-flight" requests at a time. In this way, the pipelined system of the memory requests will be used better by the accelerator. This platform has one port for reading and one for writing. In a platform with more ports, the developer would have the opportunity to use the extra ports for the read-part (4 read ports and 1 write) and minimize the delay from the read requests.

The experiments that were conducted on vLabs with real CPU-FPGA chips, help on a better understanding of how HARP works. This thesis is a step for more efficient developments in this platform. The results of those experiments show that using the DAER framework achieves an order of magnitude performance speed-up compared to the unmodified application and the CPU. But, for achieving better performance, there have to be some improvements.

8.2 Future work

This thesis is an approach for evaluating the DAER framework but there is space for improvement with exploiting more of the advantages that the platform provides.

- In this work, the HARP hybrid platform's resources are heavily underutilized. This means that multiple Fetch and Process units can be load into the FPGA and make a parallel application with multiple parallel systems.
- FPGAs overcome a huge frequency disadvantage relative to CPUs through application-specific, spatial solutions. In this work, the HW clock is at 200MHz. To decrease the results of this disadvantage, the HW clock can be pushed to 400MHz (pClock) by using different development strategies. This clock was made for running CCI-P AFUs but, according to the complexity of an AFU, the HW-clock frequency may be lower to avoid timing errors.
- In this thesis, the environment that the experiments were made has an Integrated Xeon-FPGA Hybrid Chip. For future work, the architectures could be developed in an environment with an Intel FPGA Programmable Acceleration Card (Intel FPGA PAC) [28]. The PAC has easy-access local memory that can make iterative methods like Jacobi more efficient. The AFU will not make DDR requests in every iteration, which is time-consuming but only at the start and the end of the execution.

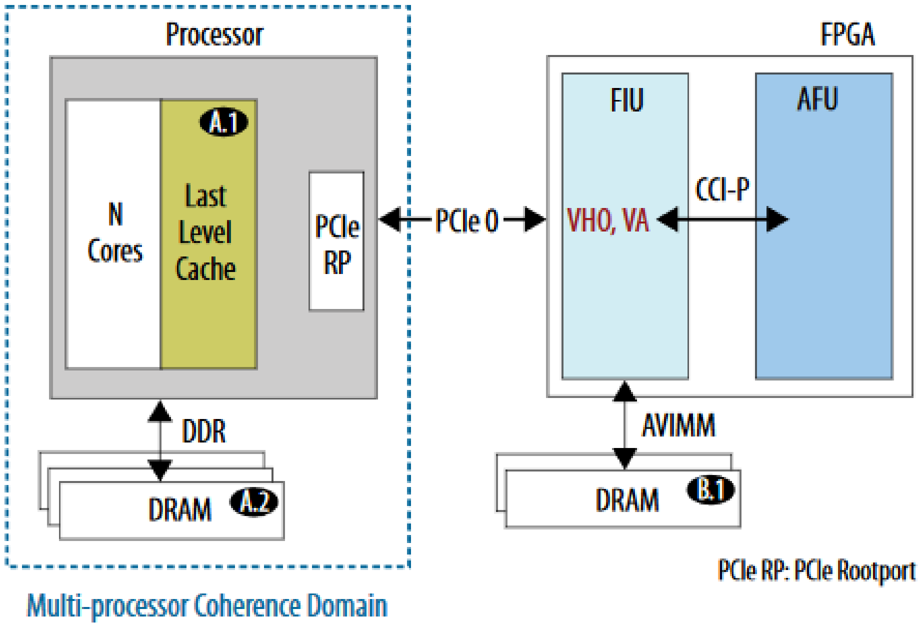

Figure 8.1. PAC Platform Memory Hierarchy. ©Intel Corporation

- The interface that was used in this work is the CCI (Core-Cache Interface). With this interface, the responses of the memory are out of order. That makes it difficult for an application that needs in order responses to prefects a big amount of data because of the need to reorder one group of responses before issue the next group of requests. The Memory Properties Factory (MPF) [29] provides a common collection of memory semantic extensions to CCI. One of these shims is the cci_mpf_shim_rsp_order that implements a reorder buffer (ROB) on read responses. With this shim can be designed an accelerator that achieves maximum read bandwidth by making read requests without waiting for the previous group of responses to be stored in order.

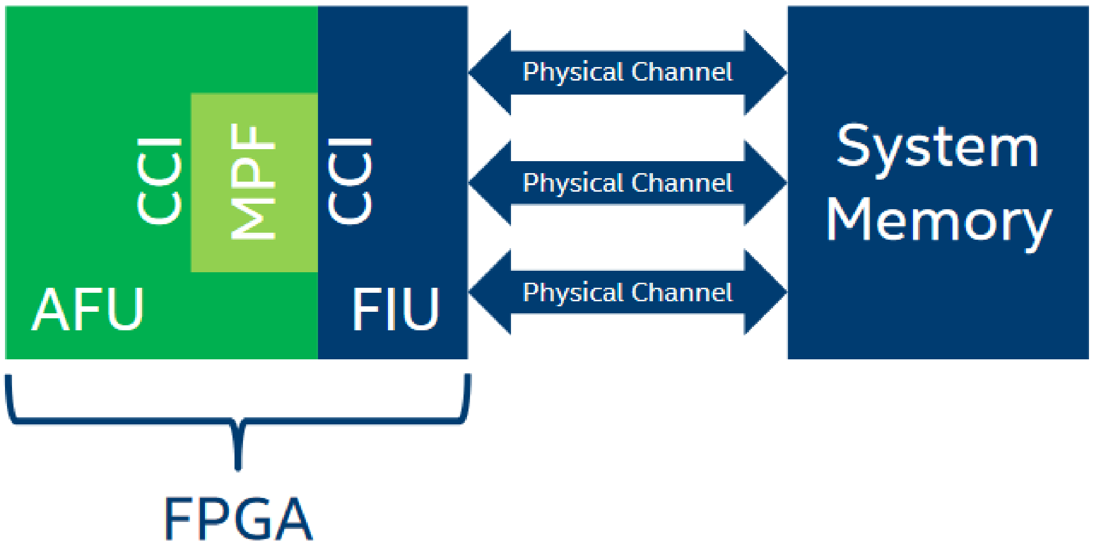

Figure 8.2. Addition of MPF to the FPGA. ©Intel Corporation

References

[1] Charitopoulos, G., Vatsolakis, C., Chrysos, G., & Pnevmatikatos, D. N. (2018). A decoupled access-execute architecture for reconfigurable accelerators. In Proceedings of the 15th ACM International Conference on Computing Frontiers (pp. 244-247). ACM.

[2] Smith, J. E. (1984). Decoupled access/execute computer architectures. ACM Transactions on Computer Systems (TOCS), 2(4), 289-308.

[3] Asanovic, K., Bodik, R., Catanzaro, B. C., Gebis, J. J., Husbands, P., Keutzer, K., ... & Yelick, K. A. (2006). The landscape of parallel computing research: A view from Berkeley. In Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley

[4] Chen, T., & Suh, G. E. (2016, October). Efficient data supply for hardware accelerators with prefetching and access/execute decoupling. In 2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO) (pp. 1-12). IEEE.

[5] Ham, T. J., Aragón, J. L., & Martonosi, M. (2017). Decoupling data supply from computation for latency-tolerant communication in heterogeneous architectures. ACM Transactions on Architecture and Code Optimization (TACO), 14(2), 1-27.

[6] Koukos, K., Black-Schaffer, D., Spiliopoulos, V., & Kaxiras, S. (2013). Towards more efficient execution: A decoupled access-execute approach. In Proceedings of the 27th international ACM conference on International conference on supercomputing (pp. 253-262). ACM.

[7] Arnau, J. M., Parcerisa, J. M., & Xekalakis, P. (2012). Boosting mobile GPU performance with a decoupled access/execute fragment processor. In ACM SIGARCH Computer Architecture News (Vol. 40, No. 3, pp. 84-93). IEEE Computer Society.

[8] de Souza Junior, A. O., de Souza, C. A. O., Bispo, J., Cardoso, J. M., Diniz, P. C., & Marques, E. (2020). Exploration of FPGA-Based Hardware Designs for QR Decomposition for Solving Stiff ODE Numerical Methods Using the HARP Hybrid Architecture. Electronics, 9(5), 843.

[9] Kyriakidis, K. (2019). Full system architectural simulation on the HARP integrated CPU-FPGA platform. In *Library of the School of Electrical and Computer Engineering, Technical University* of Crete.

[10] Pekridis, G. (2019). Evaluating the Intel HARP (tightly-coupled CPU-FPGA) platform with an ARM many-core accelerator. In *Library of School of Electrical and Computer Engineering, Technical University* of Crete.

[11] Integrated Xeon-FPGA Hybrid Chip. https://www.nextplatform.com/2018/05/24/a-peek-inside-that-intel-xeon-fpga-hybrid-chip/

[12] Xeon E5-2628L v4. http://www.cpu-world.com/CPUs/Xeon/Intel-Xeon%20E5-2628L%20v4.html

[13] Arria 10 FPGA Family. https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/pt/arria-10-product-table.pdf

[14] Intel Acceleration Stack for Intel®Xeon® CPU with FPGAs Core CacheInterface (CCI-P) Reference Manual. https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/manual/mnl-ias-ccip.pdf

[15] Open Programmable Acceleration Engine (OPAE) C API Programming Guide. https://opae.github.io/1.1.2/docs/fpga_api/prog_guide/readme.html

[16] Intel's FPGA BBB https://github.com/OPAE/intel-fpga-bbb

[17] Intel Accelerator Functional Unit (AFU) Simulation Environment (ASE) User Guide. https://opae.github.io/1.1.2/docs/ase_userguide/ase_userguide.html

[18] Intel's labs Academic Compute Environment (ACE) Documentation https://wiki.intel-research.net/index.html

[19] Quartus Prime. https://www.intel.com/content/www/us/en/software/programmable/quartus-prime/overview.html

[20] Intel Labs (vLabs). https://www.intel.com/content/www/us/en/research/overview.html

[21] Feng, W. C., Lin, H., Scogland, T., & Zhang, J. (2012, April). OpenCL and the 13 dwarfs: a work in progress. In Proceedings of the 3rd ACM/spec international conference on performance engineering (pp. 291-294).

[22] Krommydas, K., Feng, W. C., Antonopoulos, C. D., & Bellas, N. (2016). Opendwarfs: Characterization of dwarf-based benchmarks on fixed and reconfigurable architectures. Journal of Signal Processing Systems, 85(3), 373-392.

[23] Bakari, A. I., & Dahiru, I. A. (2018). Comparison of Jacobi and Gauss-Seidel Iterative Methods for the Solution of Systems of Linear Equations. *Asian Research Journal of Mathematics*, 1-7.

[24] Mathematical Association of America (MAA). https://www.maa.org/press/periodicals/loci/joma/iterative-methods-for-solving-iaxi-ibi-jacobis-method (pages 1-2)

[25] Barrett, R., Berry, M. W., Chan, T. F., Demmel, J., Donato, J., Dongarra, J., Eijkhout, V., Pozo, R., Romine, C. & Van der Vorst, H. (1994). Templates for the solution of linear systems: building blocks for iterative methods (Vol. 43). 2nd ed. Philadelphia, PA: SIAM

[26] Mathématiques et Interaction á Nice. Iterative Methods for Solving Linear Systems https://math.unice.fr/~frapetti/CorsoF/cours3.pdf (page 1)

[27] Arizona State University. Poisson's and Laplace's Equations. https://math.la.asu.edu/~gardner/laplace.pdf

[28] Accelerator Functional Unit (AFU) Developer's Guide for Intel FPGA Programmable Acceleration Card (Intel FPGA PAC). https://www.intel.com/content/www/us/en/programmable/documentation/bfr1522087299048.html

[29]Adler, M.(2017) Memory Properties Factory (MPF) https://filebox.ece.vt.edu/~athanas/HARP%20Tutorial/doc/201705_MPF_Overview.pdf