NTNU – Trondheim
Norwegian University of
Science and Technology

# Improving Energy Efficiency with Special-Purpose Accelerators

Alexandru Fiodorov

Embedded Computing Systems
Submission date: June 2013
Supervisor: Magnus Jahre, IDI

Norwegian University of Science and Technology
Department of Computer and Information Science

## Abstract

The number of transistors per chip and their speed grows exponentially, but the power dissipation per transistor is decreased slightly with each process generation. This leads to increased power density and heat generation, meaning that only a fraction of the chip can be active at any given time. To attack this problem, heterogeneous systems-on-chip are developed. They consist of multiple specialized cores, each optimized to perform a particular set of tasks. Delegating parts of the application to run on specific, energy-efficient cores, allows more computations to execute within the given power budget, increasing the overall performance of the system.

This thesis proposes a methodology for developing a special-purpose accelerator for a given application to create an energy-efficient heterogeneous system-on-chip based on the Xilinx Zynq platform. This work introduces the Xilinx tool suite used during development and defines the complete design work flow for implementing the accelerator and running the application on the accelerated system. This work evaluates the optimization techniques which lead to the most energy-efficient implementation. The simulations show that pipelining, separate ports for reading and writing data and a small, fast, local memory improves the performance of the accelerator by a factor of 44.4x and the energy-efficiency by 379x.

The accelerator is physically implemented on the Xilinx Zynq SoC and acts as a co-processor for the ARM CPU available on the system. This work proposes a methodology for evaluating the physical power consumption and performance of various configurations of the system. For the given application, the system with the accelerator running at 125 MHz is 1.5x faster and 2.15x more energy-efficient compared to the application executing only on the CPU at 666 MHz. If the clock frequencies are matched at 100 MHz, the accelerated system is 3.6x faster and 3x more energy-efficient.

# Contents

# List of Figures

# List of Tables

# List of Acronyms

| | |
|---|---|
| **ACC** | Accelerator |
| **ACP** | Accelerator Coherency Port |
| **ASIC** | Application-Specific Integrated Circuit |
| **ASIP** | Application-Specific Instruction-set Processor |
| **ASSP** | Application-Specific Standard Product |
| **BRAM** | Block RAM |
| **BSP** | Board Support Package |
| **DRC** | Design Rule Check |
| **EDP** | Energy Delay Product |
| **ED2P** | Energy Delay Squared Product |
| **EEMBC** | Embedded Microprocessor Benchmark Consortium |
| **FPGA** | Field-Programmable Gate Array |
| **FSM** | Finite-State Machine |
| **HLS** | High-Level Synthesis |
| **HP** | High-Performance slave port |
| **IP** | Intellectual Property |
| **LBM** | Lattice Boltzmann methods |
| **OCM** | On-Chip Memory |
| **PL** | Programmable Logic |
| **PLL** | Phase Lock Loops |
| **PS** | Processing System |

| | |
|---|---|
| **RTL** | Register-Transfer Level |
| **SCPI** | Standard Commands for Portable Instruments |
| **SCU** | Snoop-Control Unit |
| **SDK** | Software Development Kit |
| **SLCR** | System Level Control Registers |
| **SoC** | System-on-Chip |
| **SPEC** | Standard Performance Evaluation Corporation |
| **Tcl** | Tool Command Language |
| **VHLS** | Vivado HLS |
| **XPS** | Xilinx Platform Studio |

# Chapter 1

# Introduction

## 1.1 Motivation

Modern computing is largely influenced by power limitations. Following Moore's Law [16], the number of transistors per chip and their speed continues to increase. However, the per-transistor switching power cannot be decreased by the same factor any more, due to the limits of threshold voltage scaling [22]. The result is the increased power density and heat generation, which cooling systems fail to remove completely.

With fixed power and area budget, integrating more transistors on a die results in need for under-clocking or under-utilizing a part of the chip. Ahn et al. [1] introduced the term *utilization wall* to refer to the limit on the fraction of the chip that can be used at full speed at any time. The remaining, passive silicon areas are referred to as *dark silicon* [13].

Figure 1.1 illustrates the problem of the utilization wall. It is an example of the current-generation 45nm-scale chip, which would shrink to a quarter size at 22nm and a sixteenth at 11nm. These chips would consume the same power for 22nm with 60% increase in peak frequency, and even drawing 40% less power with 2.4 times the original frequency. Future microcontrollers would be more efficient in terms of performance, power and area. Keeping the original area constant, one could pack 4 times the transistors at 22nm and 16 times at 11nm, thus increasing the processing power. However the power constraint of the original 45nm chip limits the 22nm and 11nm chips to use only 25% and 10% respectively. This means that at any given time, up to 90% of the silicon will be "dark".

Recently, a number of researches [7, 22, 23] showed that developing heterogeneous multi-core Systems-on-Chip (SoCs), containing specialized hardware, is an effective solution to the problem of using the area budget to improve performance. Choosing a heterogeneous architecture is driven by the fact that in most embedded systems, there are a number of computationally intensive algorithms that can be easily mapped to an individual processor that is highly tuned to running that one particular type of algorithm. Such processors are generally referred to as

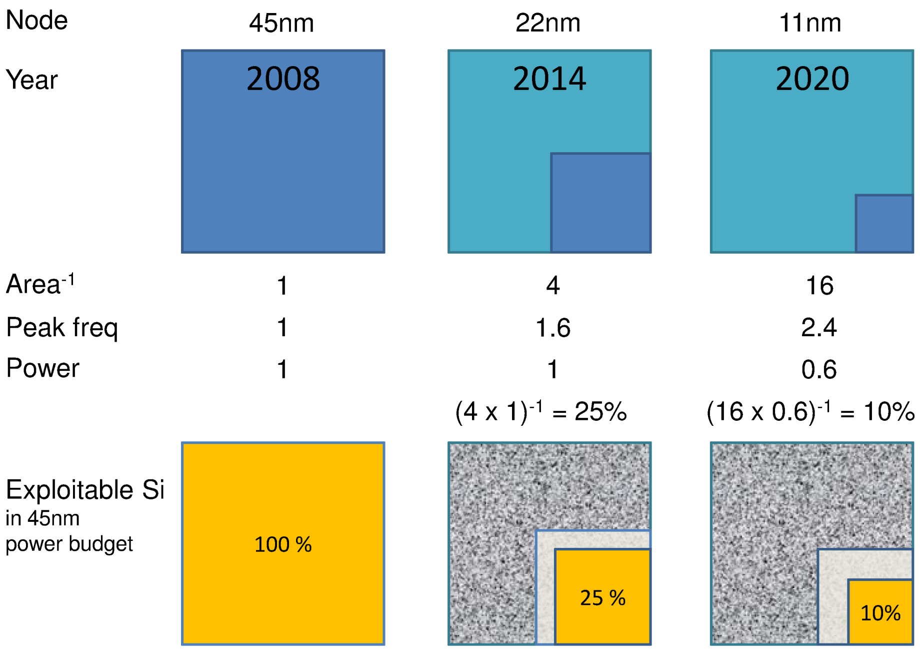


Figure 1.1: Dark silicon increase with technology scaling [3]

*Application-Specific Instruction-set Processors (ASIPs), co-processors* or *accelerators*. Instead of running the whole application on a single, big, general-purpose core, different parts of the workload are delegated to specific co-processors. These hardware units optimize per-computation power requirements, allowing more computations to execute within the given power envelope [23]. The energy-efficiency of these accelerators improve the overall performance of the system without violating the power constraint. The main challenge with such systems is the software application, which should be aware of the heterogeneous nature of the underlying hardware and partition its code accordingly.

## 1.2 Research Questions

This thesis addresses one of the most discussed problems in modern computing:

*"How to improve the energy-efficiency of the system?"*

It tries to experimentally discuss one of the aspects of this issue:

*"Does moving to heterogeneous multicore systems improve energy-efficiency?"*

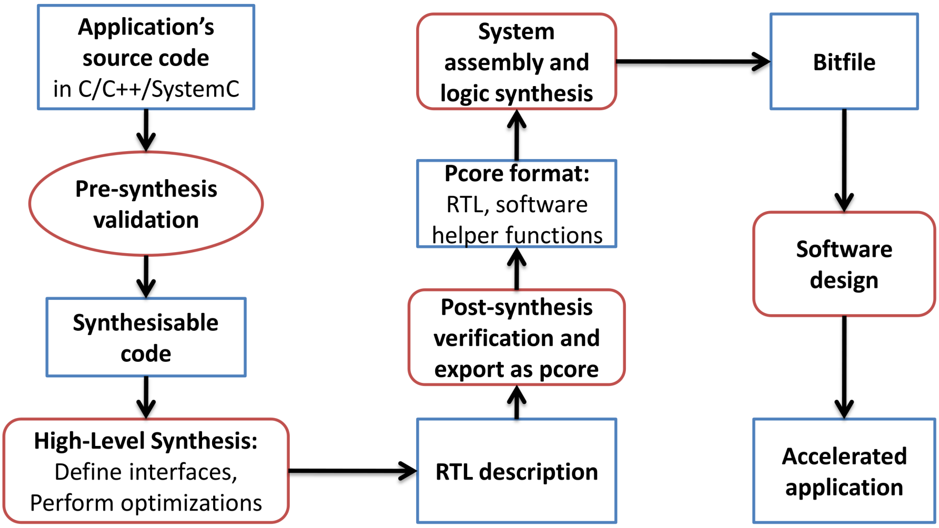

Application's source code in C/C++/SystemC
Pre-synthesis validation
Synthesisable code
High-Level Synthesis: Define interfaces, Perform optimizations
RTL description
Post-synthesis verification and export as pcore
Pcore format: RTL, software helper functions
System assembly and logic synthesis
Bitfile
Software design
Accelerated application

Figure 1.2: Design work flow

For this purpose a specialized accelerator is designed. During its development the following questions arise:

1. How can Xilinx tools assist in developing a energy-efficient hardware accelerator?
2. How can different optimization and architectural choices affect the performance and energy-efficiency of the accelerator?
3. How can the performance and energy-efficiency be evaluated on Zynq?
4. How can the custom accelerator improve the energy-efficiency of the complete system?

## 1.3 Accelerating Applications on Xilinx Zynq

This work presents the process of designing a hardware accelerator to improve the energy-efficiency of a given application. It explores the effects of different architectural techniques, like pipelining, caches and dynamic clock gating, to achieve the optimal balance between performance, area and power consumption, which leads to the most energy-efficient solution. This work also introduces a methodology for measuring power and performance, deriving metrics for energy-efficiency and comparing those for different system architectures.

The complete design flow is illustrated in Figure 1.2. It is based on High-Level Synthesis (HLS) – an automatic process of creating a Register-Transfer Level (RTL) hardware representation of an algorithm, given its behavioural description. The behavioural specifications are generally represented by a synthesisable subset of ANSI C, C++ and SystemC programming languages. The first steps in the design flow are choosing the right application and identifying the so called *hot* regions – the most computationally intensive blocks of code, which will serve as input to the HLS process. These code segments will then pass the pre-synthesis validation, to make sure that all the constructs can be synthesised into a hardware representation. After running the synthesis process, the resulting hardware description is verified against the initial source code, to make sure it correctly implements the expected behaviour. The work flow then continues with *logic synthesis* – a process of transforming the RTL representation into a design implementation in terms of logic gates. The accelerator is prototyped on a Field-Programmable Gate Array (FPGA) platform, so the logic synthesis actually outputs a bitstream to program the device. Finally, the application's source code is adapted to make use of the hardware accelerator.

In order to find a suitable candidate for acceleration, I referred to the "Berkeley Dwarves" classification, by Asanovic et al. [6]. Dwarves are equivalence classes of applications that are believed to be the common computational patters of current and future scientific computing [6]. This makes the result of current work applicable for important, real-world scenarios. I chose the target application from the "Structured Grids" dwarf.

In the applications belonging to the "Structured Grids" dwarf, data is arranged in a regular multidimensional grid (most commonly 2D or 3D, sometime 4D, but rarely higher). Computation is a sequence of grid update steps. In each iteration, every node is updated using the values from a small neighbourhood. This algorithm is highly vectorizable. The points can be visited in an order that provides spatial locality to make good use of long cache lines and temporal locality to allow cache reuse. Due to high spatial locality and predictable addressing pattern, hardware or software pre-fetching can be used effectively. Temporal locality is limited and depends on the size of the neighbourhood, as each data value is accessed once by each neighbourhood that contains it [10].

In my previous paper [14] I identified the SPEC2006 implementation of Lattice Boltzmann methods (LBM) to be the best candidate for acceleration out of all Standard Performance Evaluation Corporation (SPEC) benchmark members of the Structured Grids dwarf. Its method `performStreamCollide` takes more than 99% of the execution time. Due to its structured data organization, spatial and temporal locality, application's performance can potentially benefit from hardware acceleration.

The accelerator is developed on a Xilinx Zynq System-on-Chip [31]. Zynq is composed of a dual-core ARM-A9 CPU and a Field Programmable Gate Array (FPGA), tightly integrated on a single die. The FPGA provides a flexible and powerful environment for rapid prototyping and exploring different design decisions. Coupled with the ARM cores, it creates a system well-suited for developing high-performance accelerator applications.

## 1.4 Contributions

This thesis proposes improving the system's energy-efficiency by implementing the most computationally intensive part of an application as a hardware accelerator. In the process of developing the accelerator the following is achieved:

1. The complete tool suite for developing a hardware module, given the behavioural description of a program in C is defined (Chapter 3).

2. Different optimization techniques to improve accelerator's energy-efficiency are evaluated (Section 3.3).

3. The specific details of developing such an accelerator on Xilinx Zynq are explained (Section 3.5).

4. The methodology for measuring power and latency is defined. These metrics are used to compute system's energy-efficiency (Chapter 4).

5. The results of the experiments are presented and analysed (Chapter 5).

## 1.5 Thesis Organization

The current thesis starts with Chapter 1, that presents the motivation, research questions and contributions. Chapter 2 presents the background on energy-efficient hardware accelerators, the Berkeley Dwarves and the Xilinx Zynq. Chapter 3 describes the design flow for implementing the accelerator, followed by Chapter 4 which presents the methodology for power measurements and describes how to configure the system to explore the energy-efficiency of different setups. Chapter 5 summarizes the results of the experiments. Chapter 6 draws the conclusions and discusses the future work. The Appendix lists the design files attached to this work.

# Chapter 2

# Background

## 2.1 Accelerating for Energy-Efficiency

At first, accelerators were concerned only about increasing the performance of the system. Most of them are strongly application-specific and cover domains such as cryptography [25], signal processing [11], vector processing [2] and computer graphics [17]. However, of particular interest for this paper are the works of Venkatesh et al. that focus on improving the energy-efficiency, while keeping the performance constant [22,23] or increasing it [18]. In contrast to the above mentioned examples, their approach is general and can target different applications. These papers are discussed in more detail in the following.

To attack the *utilization wall*, Venkatesh et al. [22] present the *conservation cores* or *c-cores*. C-cores have a different purpose than conventional accelerators and focus primarily on energy reduction rather than performance. Conservation cores that achieve a better performance are also possible, but the paper describes those with similar performance. The focus on energy-efficiency allows the c-cores to address a broader range of applications. Codes with large amounts of parallelism and predictable memory access patterns map naturally to conventional accelerators. On the other hand, their results show that irregular applications with little parallelism and very poor memory behaviour are excellent candidates for conservation cores. Their paper describes the tool chain for automatically synthesising c-cores for any given C code base (Figure 2.1). First the tool analyses the input C code and generates a set of c-cores. The custom compiler uses the descriptions of the c-cores available on the chip and produces assembler code that utilizes them. This means the C code should not be adapted to use the c-cores. The conservation cores support load-time reconfigurability using *patches*. These introduce a limited amount of flexibility, hence the applications should experience a high level of maturity to be accelerated using c-cores. The results show an improvement in energy-efficiency between 3.3x and 16.0x for targeted functions and up to 2.1x for the full application.

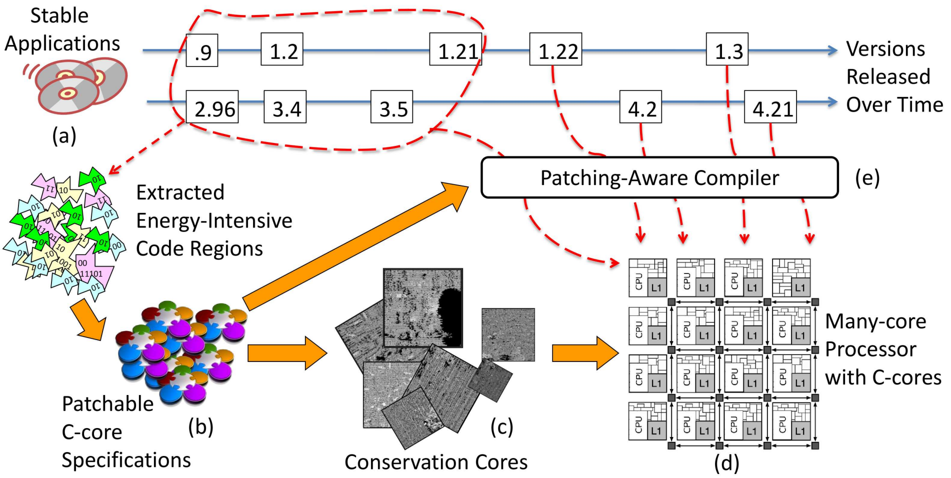

Stable Applications
(a)
.9
1.2
1.21
1.22
1.3
2.96
3.4
3.5
4.2
4.21
Versions Released Over Time
Extracted Energy-Intensive Code Regions
Patching-Aware Compiler
(e)
Patchable C-core Specifications
(b)
(c)
Conservation Cores
CPU
L1
Many-core Processor with C-cores
(d)

Figure 2.1: Automatic synthesis and compilation of c-cores [22]

The next paper by Venkatesh et al. [23] describes the *Quasi-specific Cores* (QsCores), specialized processors capable of executing several general-purpose computations, optimized for energy-efficiency. When carefully implemented, a relatively small number of QsCores can potentially support a significant part of the computation. The design flow (Figure 2.2) is based on the observation that similar code patterns exist within and across applications. QsCores exploit this similarity to reduce the hardware redundancy of specialized cores by implementing a generalized functionality which is customized by parametrization. The authors develop heuristics that analyse the energy-area trade-off to ensure that the QsCores fit the area budget while providing a high degree of energy-efficiency. They evaluate their solution by implementing the main operator functions, *find, insert, delete* of commonly used data structures like linked-list, binary and AA trees, hash table etc. Using just 4 QsCores, they achieve 13.5x energy savings. On a general-purpose workload consisting of different benchmarks, the energy-efficiency is improved by a factor of 18.4x while reducing the number of specialized cores by 50% and area by 25% compared to fully-specialized implementation. At the system level, the energy-delay product is improved by 46% compared to general-purpose processors.

In the last paper [18], the authors optimize their earlier solution of c-cores with two techniques to improve performance and energy-efficiency even further, Selective Depipelining (SDP) and Cachelets. These enhanced c-cores are called *Efficient Complex Operation Cores* (ECOcores). SDP is a novel pipelining technique that reduces both unnecessary clock power and time wasted due to poorly aligned operators within cycles. Several operations, including dependent memory accesses can be performed in a single logical clock cycle. Selective depipelining exploits the fact that datapath and memory operations have different needs. The datapath is inexpensive to replicate by adding more functional units, while the memory accesses are centralized around a single bus. SDP allows the memory to run at a much higher

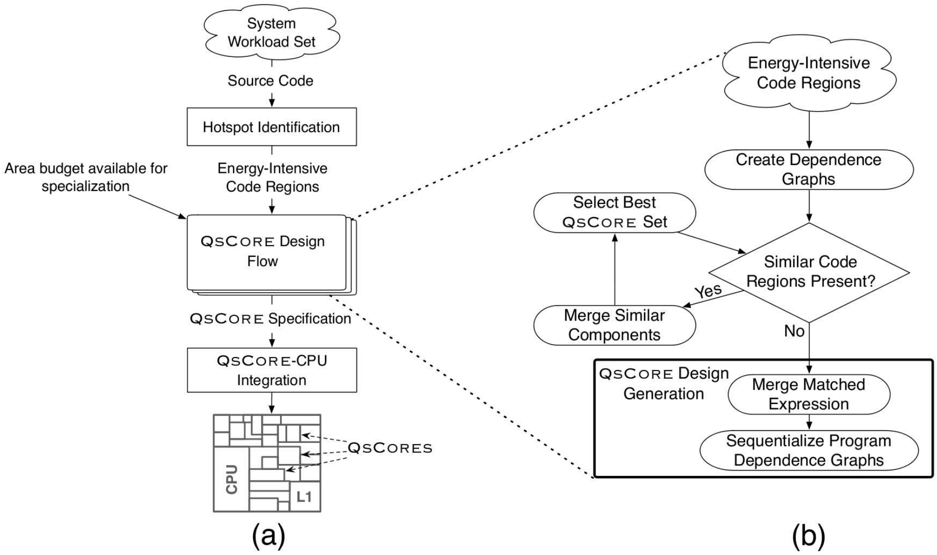


Figure 2.2: **Overview of QsCore-enabled system** The figure overviews the high-level design flow of a QsCore-enabled system (a) as well as the design flow for generating QsCores (b) (Source: [23])

clock rate, which effectively replicates the memory interface *in time*. Slowing down the datapath clock saves power and leverages ILP by replicating the computation *in space*. The second technique, *cachelets* is a type of small, distributed, coherent L0 cache that optimizes common load-store operations to reduce the memory latency by up to 83%. Cachelets provide sub-cycle memory accesses, 6x faster than L1 cache. Each cachelet serves a particular set of static operations. ECOcores achieve a reduction of 2x for Energy Delay Product (EDP) and 35% for area relative to c-cores. Compared to an efficient MIPS processor, they achieve on average a speed-up of 1.5x for targeted functions and 1.3x for the whole application, with EDP reduction of 7.1x and 2.9x respectively.

## 2.2 Berkeley Dwarves

For many years the evaluation of architectural innovations was based on benchmark suites like Standard Performance Evaluation Corporation (SPEC) and Embedded Microprocessor Benchmark Consortium (EEMBC) [12, 20]. These are very well suited for uni-processor systems, but there are no such standard benchmarks for evaluating parallel applications on multi- and many-core architectures, hence a higher level of abstraction is needed.

Researchers at Berkeley came up with the idea of “dwarves”, algorithmic methods that capture a pattern of computation and communication [6]. The initial “Seven Dwarves” are based on the work of Phil Colella [8], who identified seven numerical methods that he believed will be important for science and engineering for at least the next decade. Later, six more dwarves were identified. Table 2.1 is adopted from [6, 10] and presents a brief description of the 13 Berkeley Dwarves.

*"Dwarves are equivalence classes, where membership is defined by the similarity in computation or data movement."* [6]. Dwarves define a higher level of abstraction for a broad range of applications. Asanovic et al. believe that *"although implementation of these programs may vary, the underlying patterns have persisted through generations of changes and will remain important in the future."* [6]

Members of the dwarf "Structured Grids" are good candidates for acceleration, due to their structured data organization, spatial and temporal locality. The accelerator implementation described in this thesis targets one of the members of this dwarf – the SPEC2006 implementation of Lattice Boltzmann methods algorithm.

Table 2.1: The 13 Berkeley Dwarves

| **Name** | **Description** |
|---|---|
| ***Dense linear algebra*** | Data are dense matrices or vectors, having little zero values, typically laid out as a contiguous array. Computations are performed on elements, rows, columns or partitions of matrices, usually addition and multiplication. |
| ***Sparse linear algebra*** | Data sets include vectors and matrices with few non-zero values. To reduce space and computation, they are stored in a compressed form, as indexed lists. |
| ***Spectral methods*** | Data is operated on in the spectral domain, often transformed from either a temporal or spatial domain. During a transformation, spectral methods typically use multiple stages, where the dependencies within a stage for a set of butterfly patterns. Each butterfly operation has two inputs and two outputs (each typically a complex number) and perform a set of multiply and add operations. |
| ***N-body methods*** | Depends on interactions between many discrete points. Variations include particle-particle methods, where every point depends on all others, leading to an O(N$^2$) calculation, and hierarchical particle methods, which combine forces or potentials from multiple points to reduce the computational complexity to O(N log N) or O(N). |
| ***Structured grids*** | Data is arranged in a regular multidimensional grid. Computation proceeds as a sequence of grid update steps. At each step, all points are updated using values from a small neighbourhood around each point. These codes have a high degree of parallelism, and data access patterns are regular and statically determinable. |

Continued on next page

Table 2.1 – continued from previous page

| Name | Description |
|---|---|
| ***Unstructured grids*** | An irregular grid where data locations are selected, usually by underlying characteristics of the application. Data point location and connectivity of neighbouring points must be explicit. The points on the grid are conceptually updated together. Updates typically involve multiple levels of memory reference indirection, as an update to any point requires first determining a list of neighbouring points, and then loading values from those points. |
| ***MapReduce (Monte Carlo)*** | This dwarf was originally called “Monte Carlo”, after the technique of using statistical methods based on repeated random trials. The patterns defined by the programming model MapReduce are a more general version of the same idea: repeated independent execution of a function, with results aggregated at the end. Nearly no communication is required between processes. |
| ***Combinational Logic*** | exploits bit-level parallelism to achieve high throughput. Workloads dominated by combinational logic computations generally involve performing simple operations on very large amounts of data. |
| ***Graph Traversal*** | Visits many nodes in a graph by following successive edges. These applications typically involve many levels of indirection, and a relatively small amount of computation. |
| ***Dynamic Programming*** | is an algorithmic technique that compute solutions by solving simpler overlapping subproblems. It is particularly applicable for optimization problems where the optimal result for a problem is built up from the optimal result for the subproblems. |
| ***Backtrack and Branch-and-Bound*** | These algorithms work by the divide-and-conquer principle: the search space is subdivided into smaller subregions (this subdivision is referred to as branching), and bounds are found on all the solutions contained in each subregion under consideration. Suboptimal solutions are discarded. |
| ***Graphical Models*** | A graphical model is a graph in which nodes represent variables, and edges represent conditional probabilities. Graphical models include Bayesian networks (also known as belief networks, probabilistic networks, causal network, and knowledge maps). Hidden Markov models and neural networks are also graphical models. |
| ***Finite State Machine*** | capture a system whose behaviour is defined by states, transitions defined by inputs and the current state, and events associated with transitions or states. These applications are mostly sequential. |

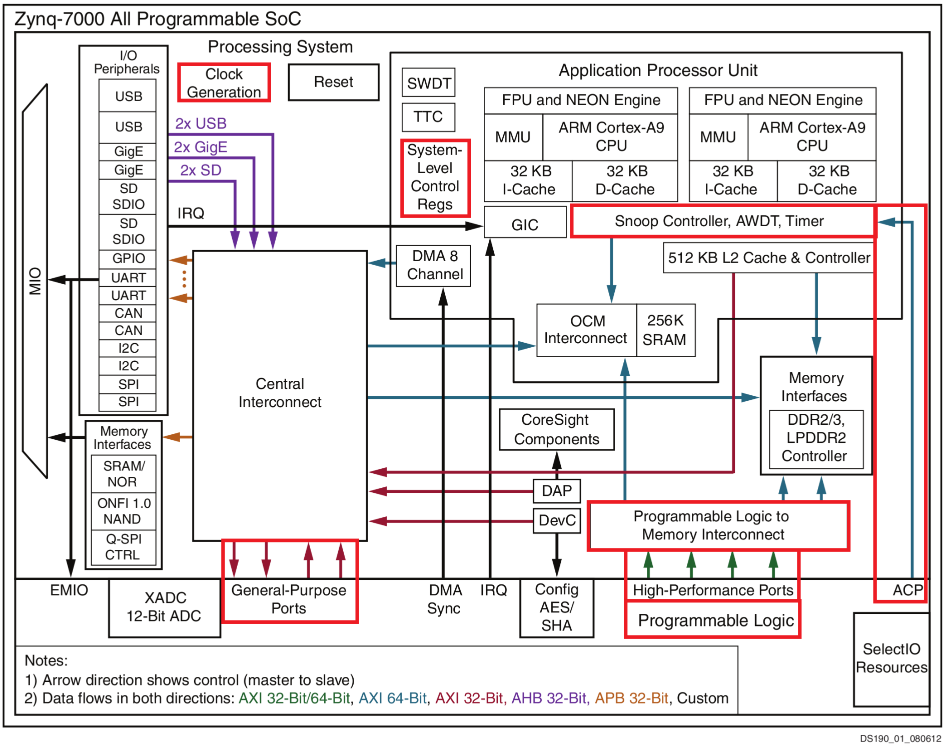


Figure 2.3: Xilinx Zynq-7000 SoC overview [32]

## 2.3 Xilinx Zynq-7000 System-on-Chip

Zynq-7000 is a family of SoC platforms composed of a powerful dual-core ARM® Cortex™ A9 MPCore™ based Processing System (PS) and a 28nm Xilinx Programmable Logic (PL) on a single die. The PS also includes on-chip memory, external memory interfaces and a variety of peripherals (Figure 2.3). The system offers the scalability and flexibility of an FPGA, while providing performance, power and ease of use typically associated with Application-Specific Integrated Circuits (ASICs) and Application-Specific Standard Products (ASSPs) [32]. The use of a powerful general-purpose processor allows developing both bare-metal and Linux applications.

Tightly integrated on the same die is the PL part, which is used to extend the PS. It is based on 28nm Artix-7 or Kintex-7 FPGA fabrics and comes in different sizes to accommodate the needs of various use-cases and applications. Artix-7 devices offer lower power and lower cost, targeting high-volume applications, while the Kintex-7 ones are used for high-performance and high I/O throughput [31].

The Zynq architecture enables software programmability in the PS and implementation of custom logic in the PL. The software updates and the hardware customization – whether static or dynamic, partial of full reconfiguration – can

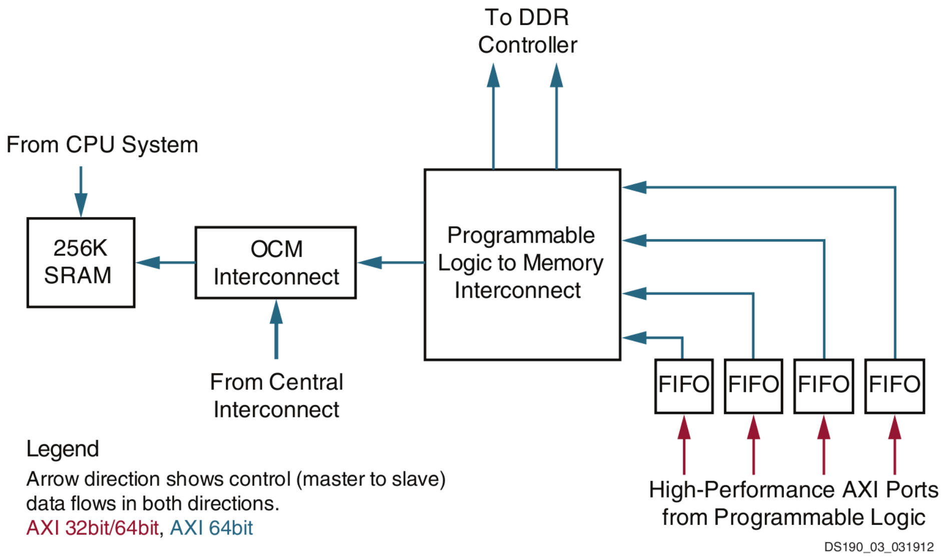

To DDR Controller
From CPU System
256K SRAM
OCM Interconnect
Programmable Logic to Memory Interconnect
From Central Interconnect
FIFO
FIFO
FIFO
FIFO
Legend
Arrow direction shows control (master to slave) data flows in both directions.
AXI 32bit/64bit, AXI 64bit
High-Performance AXI Ports from Programmable Logic
DS190_03_031912

Figure 2.4: PL Interface to PS Memory Subsystem [32]

be accomplished under control of the ARM processing system [31]. This opens a broad range of possibilities for hardware-software co-design. The integration of the PS with the PL allows levels of performance that the two-chip solutions (e.g. an ASSP with an FPGA) cannot match due to their limited I/O bandwidth, latency and power budgets [32].

Figure 2.3 highlights the parts of the system that are used for configuring and communicating with the custom logic implemented in the PL. Zynq is a processor-centric system, with the CPU coordinating the work of all peripherals. Zynq adopted the AMBA bus with AXI interface as the primary means of communication between the PS and the modules implemented in PL. The PS features two 32-bit general-purpose (GP) AXI Master ports to access the AXI Slave peripherals in PL. Usually these peripherals contain a set of internal configuration and status registers which are memory mapped to CPU's address space.

Zynq features the ability to share processor memory (internal and external) with the programmable logic, achieving high bandwidth with low latency. The peripherals that implement the AXI Master interface can access processor memory using the following ports [32]:

- Two 32-bit AXI Slave interfaces
- Four 64-bit/32-bit configurable, buffered AXI Slave interfaces with direct access to DDR memory and On-Chip Memory (OCM), referred to as High-Performance slave ports (HP).
- One 64-bit Slave Accelerator Coherency Port (ACP) for coherent access to CPU memory.

The two highest performance interfaces between PS and PL are the HP and the ACP. The HP interfaces connect the PL to the memory interconnect via a FIFO controller. Two of the three output ports go to the DDR memory controller and the third goes to the dual-ported OCM (Figure 2.4). The Accelerator Coherency Port (ACP) provides connectivity between the CPU and a potential accelerator function in the PL. It directly connects to the Snoop-Control Unit (SCU) of the ARM Cortex-A9 processors, enabling cache-coherent access to CPU data in the L1 and L2 caches [32].

Zynq allows fine tuning of the system power consumption via dynamic clock gating. The different clocks in the system can be configured from software by modifying the System Level Control Registers (SLCR) (Figure 2.3). The PL and PS are on separate power domains, allowing to shut down the PL completely, when not in use. The PL needs to be reconfigured after each power-on. The user should take PL configuration time into consideration when using this power saving mode. The PS cannot be turned off, but can be clocked down to 20 MHz. If the CPU delegates a very time-consuming task to the accelerator, it can go to a low-power sleep mode and be woken up by an interrupt, when the accelerator is finished. Furthermore, for single-threaded applications it makes sense to disable one of the cores to save energy.

# Chapter 3

# Implementation

This work explores the techniques and tools which are used to improve the system's energy-efficiency. My approach selects a particular application and develops a special-purpose accelerator which offloads the CPU by implementing the most computationally intensive part of the algorithm in hardware. The accelerator is developed with the main focus on energy-efficiency, which in addition may or may not improve the performance.

The purpose of this chapter is to present the complete design flow in detail (see Figure 1.2 on page 13) and tools used for implementing the hardware accelerator. Since the target device is a Xilinx Zynq platform, the Xilinx tool-chain is used. It is composed of Vivado HLS (VHLS), for designing the accelerator, PlanAhead with Xilinx Platform Studio (XPS) and ChipScope, for assembling and configuring the target device, and the Xilinx Software Development Kit (SDK) for modifying the original software to work on the accelerated system.

The first step of the design flow (Figure 1.2) is described in my previous paper [14]. That work analysed the Standard Performance Evaluation Corporation (SPEC) benchmark suite to identify the SPEC2006 implementation of the Lattice Boltzmann methods (LBM) algorithm as the best candidate for acceleration. LBM is a class of computational fluid dynamics (CFD) methods for fluid simulation [24]. It is the computationally most important part of a larger code which is used in the field of material science to simulate the behaviour of fluids with free surfaces, in particular the formation and movement of gas bubbles in metal foams [21].

The most computationally intensive part of the algorithm is the *performStreamCollide* method. This method maps very well onto a hardware implementation, because it is self-contained, *i.e.* does not call other methods, and consumes more than 99% of algorithm's total execution time.

This chapter describes the steps of the design flow (Figure 1.2) that start after the function to develop a hardware accelerator for is chosen:

- Pre-synthesis validation (Section 3.2)
- High-Level Synthesis (HLS) (Section 3.3)
- Post-synthesis verification (Section 3.4)
- System assembly and logic synthesis (Section 3.5)
- Software design (Section 3.6)

The High-Level Synthesis (HLS) process uses the function's source code to generate the RTL representation of the algorithm. The input to HLS should be adapted to pass the pre-synthesis validation, to make sure it does not contain constructs that cannot be synthesised, like recursive functions, dynamic memory allocation, system calls, etc. During the HLS phase, the tool is instructed to perform certain optimizations on the resulting RTL, like pipelining and loop unrolling, as well as defines accelerator's interface for communication with other modules in the system. The resulting hardware description then passes the post-synthesis verification against the source code, to prove it correctly implements the desired behaviour.

The complete system, containing the accelerator and all the necessary modules and connections is assembled and fed into the *logic synthesis* tool, which generates the bitfile to program the target FPGA device. Having the hardware system powered by the accelerator, the source code of the LBM algorithm is changed to use the accelerator for computing the `performStreamCollide` function.

## 3.1 Preparing the Application

The default data type for input and output arguments, used in SPEC2006 implementation of LBM, is a four-dimensional grid, 100x100x130x20 of double precision floating point numbers. In addition, the algorithm needs some margin space around the grid to be allocated, which makes the total size for one grid about 205MB. The method `performStreamCollide` operates on two grids, one source and one destination, which makes a total of 410 MB of memory needs to be allocated.

At first, I could not figure out how to access the DDR memory from the accelerator, so Block RAM (BRAM) was used to store the data. This architectural choice limited the size of the initial input set of application, because the maximum size of BRAM that can fit on Xilinx Zynq is only 560 kB, part of which is used internally by the accelerator. This restriction can be met using various techniques, like prefetching or double buffering, but these come with the additional difficulties when tracing the data movement in the algorithm.

Due to time constraints I decided to use a simpler method. The problem size has been scaled to 10x10x13x20 of *single* precision floating point numbers, which resulted in a total memory requirement of 266 kB. However the output of single precision computation is different when running on the CPU compared to floating

Listing 3.1: Remove dynamic memory allocation

```
/* dynamic memory allocation cannot be synthesised */
// char* foo = (char*)malloc(128 * sizeof(char));

/* use static memory allocation instead */
char _foo[128];
/* pointer in the original design using malloc should not be rewritten,
 * make it point to the existing static resource */
char* foo = &_foo[0];
```

point cores used in the accelerator. This mismatch prevented proper verification of the design, so I reverted back to double precision with a 10x10x5x20 grid. It resulted in a memory consumption of 281 kB. The maximum size of a BRAM module for Zynq is 256 kB, so I used two modules to store the grids.

Later in the design phase, I managed to access the DDR from the accelerator, but still kept the reduced size of the input set, to be able to evaluate how different memory systems affect performance and energy-efficiency.

## 3.2 Pre-synthesis Validation

Prior to synthesis, the C code should be validated by a test bench, which is nothing but a normal C program, containing all the functions above the synthesised one and a `main` function. The test bench should be self-checking, *i.e.* it should compare the output of the function to be synthesised with a "golden" result. It returns zero in case of success and a non-zero value, in case the outputs mismatch.

It is a good design practice to keep the functions used by the test bench in separate files from the functions to be synthesised. If a file contains both, it should be added to the project twice, once as a source file and once as a test bench file. All the input and output files used by the test bench should also be added to the project as test bench files.

Not every C/C++ function can be synthesised in VHLS. To be *synthesisable*, the function and all other functions it calls should contain the entire functionality of the design, avoiding system calls to the operating system. All its constructs should be unambiguous and of bounded size. The following is a list of constructs that cannot be synthesised:

- **System calls** cannot be synthesised, because they invoke OS routines which cannot be part of the final hardware design. For example `printf()`, `scanf()`, `sleep()`, `time()`, etc. should be removed from the functions to be synthesised. Also functions that handle files are not allowed, because there will be no concept of a file system in the final design. Access to external data should be performed via top-level function parameters or global variables.

- **Dynamic memory and functions.** To be able to synthesise a hardware implementation, the design must be self-contained, specifying all the required

Listing 3.2: Implementing unsupported casts

```c
union udouble {
	double d;
	u64 u;
};
union udouble ud;

u64 double2int(double d) {
	ud.d = d;
	return ud.u;
}

double int2double(u64 u) {
	ud.u = u;
	return ud.d;
}
```

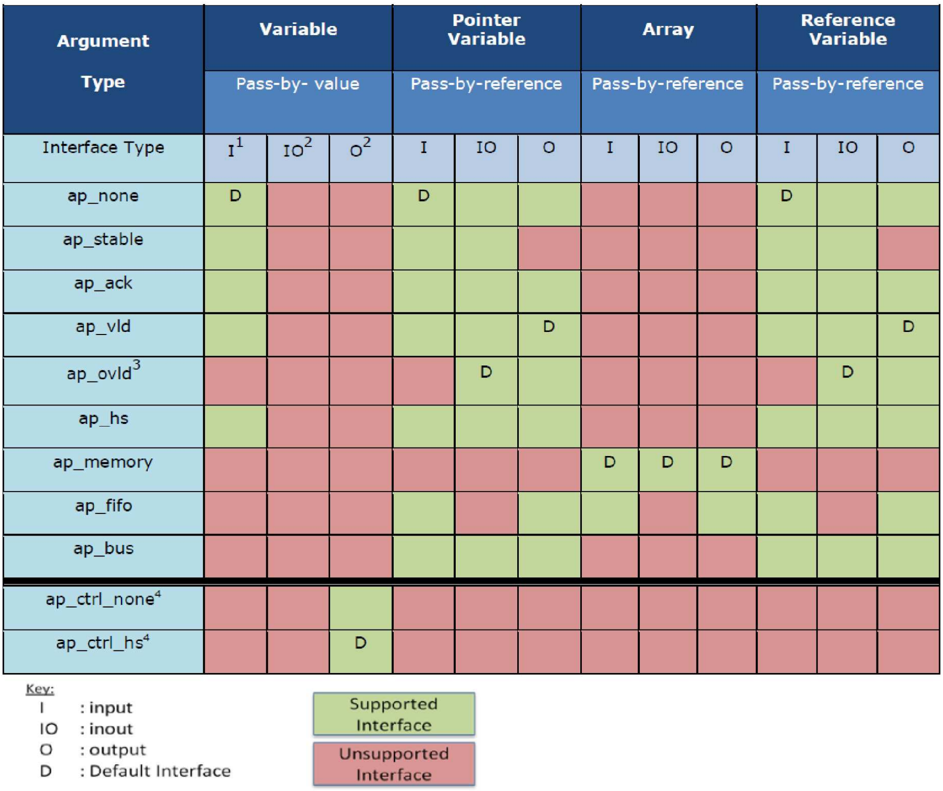
resources with fixed sizes. Dynamic memory allocation violates this constraint. Code that uses calls to `malloc()`, `calloc()`, `free()`, etc. should be adapted to use static allocation (see Listing 3.1). The same stands for virtual functions in C++, which cannot be used because the actual function to be executed is decided at runtime.

- **General Pointer Casting** - Pointer casting is not supported in general case, only between native C types. The LBM algorithm uses unsupported casting between `double` and `long int`. This casting is implemented using unions (see Listing 3.2).
- **Pointer Arrays** - Arrays of pointers are supported for synthesis if each pointer points to a scalar or an array of scalars.
- **Recursive Functions** - Recursive functions cannot be synthesised. This refers to functions which can form endless recursion as well as tail recursions, with a finite number of calls.

## 3.3 High-Level Synthesis (HLS)

High-Level Synthesis (HLS), also called behavioural or architectural-level synthesis, is an automated design process which generates RTL designs from behavioural specifications. These specifications are generally defined using a synthesisable subset of ANSI C, C++ and SystemC languages. The HLS tools also allow to define a cost function and a set of design constraints for area, performance, power consumption, etc. The goal is to generate a RTL design that implements the specified behaviour while satisfying the design constraints and optimizing the given cost function.

The RTL design of the LBM accelerator is built using the Xilinx Vivado HLS (VHLS) tool. It takes a C/C++ function as input and produces the equivalent



| Argument Type | Variable | | | Pointer Variable | | | Array | | | Reference Variable | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | Pass-by- value | | | Pass-by-reference | | | Pass-by-reference | | | Pass-by-reference | | |
| Interface Type | I[1] | IO[2] | O[2] | I | IO | O | I | IO | O | I | IO | O |
| ap_none | 🟩 D | 🟥 | 🟥 | 🟩 D | 🟩 | 🟩 | 🟥 | 🟥 | 🟥 | 🟩 D | 🟩 | 🟩 |
| ap_stable | 🟩 | 🟥 | 🟥 | 🟩 | 🟩 | 🟥 | 🟥 | 🟥 | 🟥 | 🟩 | 🟩 | 🟥 |
| ap_ack | 🟩 | 🟥 | 🟥 | 🟩 | 🟩 | 🟩 | 🟥 | 🟥 | 🟥 | 🟩 | 🟩 | 🟩 |
| ap_vld | 🟩 | 🟥 | 🟥 | 🟩 | 🟩 | 🟩 D | 🟥 | 🟥 | 🟥 | 🟩 | 🟩 | 🟩 D |
| ap_ovld[3] | 🟥 | 🟥 | 🟥 | 🟥 | 🟩 D | 🟩 | 🟥 | 🟥 | 🟥 | 🟥 | 🟩 D | 🟩 |
| ap_hs | 🟩 | 🟥 | 🟥 | 🟩 | 🟩 | 🟩 | 🟥 | 🟥 | 🟥 | 🟩 | 🟩 | 🟩 |
| ap_memory | 🟥 | 🟥 | 🟥 | 🟥 | 🟥 | 🟥 | 🟩 D | 🟩 D | 🟩 D | 🟥 | 🟥 | 🟥 |
| ap_fifo | 🟥 | 🟥 | 🟥 | 🟩 | 🟥 | 🟩 | 🟩 | 🟥 | 🟩 | 🟩 | 🟥 | 🟩 |
| ap_bus | 🟥 | 🟥 | 🟥 | 🟩 | 🟩 | 🟩 | 🟥 | 🟥 | 🟥 | 🟩 | 🟩 | 🟩 |
| ap_ctrl_none[4] | 🟥 | 🟥 | 🟩 | 🟥 | 🟥 | 🟥 | 🟥 | 🟥 | 🟥 | 🟥 | 🟥 | 🟥 |
| ap_ctrl_hs[4] | 🟥 | 🟥 | 🟩 D | 🟥 | 🟥 | 🟥 | 🟥 | 🟥 | 🟥 | 🟥 | 🟥 | 🟥 |

Key:
I : input
IO : inout
O : output
D : Default Interface

🟩 Supported Interface
🟥 Unsupported Interface

Figure 3.1: Data type and interface synthesis support [28]

Verilog code, that implements the same logic in hardware. This particular function is considered the top module and all the functions it calls are implemented as submodules. To avoid compatibility issues between different tools of the suite, the name of the top module should be in *lower case* characters. Otherwise, during logic-synthesis the PlanAhead tool produces an error that tells nothing about the cause of the problem and is it hard to figure out that the name of the module was the reason of failure.

The following sections describe different types of interface protocols and bus interface implementations that Vivado HLS can synthesise for a given module. Later the implementation details of three versions of the accelerator are presented, Default, Optim and Dual.

### 3.3.1 Interface Synthesis

When the method's source code is synthesised into an RTL module, the method's arguments are synthesised into RTL data ports. Interface synthesis is used to automatically add an interface protocol to the RTL data ports. The interface protocol could be as simple as an output valid signal indicating when an output is ready or it could include all the ports required to interface a BRAM [28].

The type of interface depends on the C argument. For example, the array arguments are the only ones that support a random access memory interface. This interface is used to directly connect to memory elements. If the memory is accessed via a bus interface, the argument should be a C pointer or a C++ reference variable. Figure 3.1 summarizes the types of interfaces which are supported for each type of C function argument. If no interface type or an unsupported type is specified for a port, the default one will be implemented as detailed in Figure 3.1.

The notes in Figure 3.1 are explained as follows [28]:

1. The concept of inputs and outputs is somewhat different between the C functions and RTL blocks. The following convention is used here for the purposes of explaining interface synthesis:

   - A function argument which is read and never written to, like an RTL input port, is referred to as an input (I)
   - A function argument which is both read and written to, like an RTL inout port, is referred to as inout (IO)
   - A function argument which is written and never read, like an RTL output port, is referred to as an output (O)

2. A standard pass-by-value argument cannot be used to output a value to the calling function. The value of an argument such as this can only be returned (or output from the function) by the function return statement.

   - Any pass-by-value function argument which is written to but never read, like an RTL output port, will be synthesised as an RTL input port with no fanout.
   - Any pass-by-value function argument which is written to and read, like an RTL inout port, will be synthesised as an RTL input port only.

3. The `ap_ovld` interface type is only valid for output ports.

4. The interface types `ap_ctrl_none` and `ap_ctrl_hs` are used to control the synthesis of function level interface protocols. These interface types are specified on the function itself (all other interface types are specified on the function arguments).

Table 3.1 summarizes the available interface protocols. There are two types of interface synthesis, the one that is performed on C function arguments and the one that is applied at the function or block level.

| Interface | Description |
|---|---|
| ap_none | The simplest interface with no associated control signals. The producer blocks are required to provide data to the input port at the correct time or hold it for the length of the transaction. The consumer blocks should read the output ports at the correct time. |
| ap_stable | Like ap_none it does not add any interface control ports to the design. The ap_stable informs the High-Level Synthesis (HLS) that the data applied to this port will remain stable during normal operation, but is not a constant value which could be optimized, and the port is not required to be registered. |
| ap_ack | Provides an acknowledge signal to say when data is consumed. |
| ap_vld | Provides a valid signal to indicate when the data is valid. |
| ap_ovld | The same as ap_vld, but can only be specified on output ports. This is a useful type for ensuring pointers which are both read from and written to, will only be implemented with an output valid port (and the input half will default to type ap_none) |
| ap_hs | This interface provides both an acknowledge and a valid signal. It is a superset of the ap_ack, ap_vld and ap_ovld interfaces. |
| ap_memory | Used to communicate with memory elements (RAMs, ROMs) when the implementation requires random accesses to memory locations. The memory interface cannot be stalled by external signals. It provides an indication of when output data is valid. |
| ap_fifo | Used when the access is performed only in a sequential manner. This interface allows the port to be connected to a FIFO, supports full two-way empty-full communication. |
| ap_bus | Used to communicate with a bus-bridge. The interface does not adhere to any specific bus standard but is generic enough to be used with a bus bridge which in-turn arbitrates with the bus system and is responsible to cache all burst writes. It supports standard and burst modes of operation. |
| ap_ctrl_none<br>ap_ctrl_hs | Used to specify if the RTL is implemented with block-level handshake signals or not. These signals specify when the design can start to perform its standard operation and when that operation ends. |

Table 3.1: Description of interface protocols supported in Vivado HLS

Standard port level interface synthesis is specified for each function argument. A function argument which is both read from and written to is synthesised in the following manner [28]:

- `ap_none, ap_stable, ap_ack, ap_vld, ap_ovld, ap_hs` – separate input and output ports. For example, if function argument `arg1` was both read from and written to, it would be synthesised as RTL input data port `arg1_i` and output data port `arg1_o` and any specified or default IO protocol is applied to each port individually.

- `ap_memory, ap_bus` – a single interface is created. Both these RTL ports support read and write.

- `ap_fifo` – read and write are not supported. There must be two separate function arguments for reading and writing.

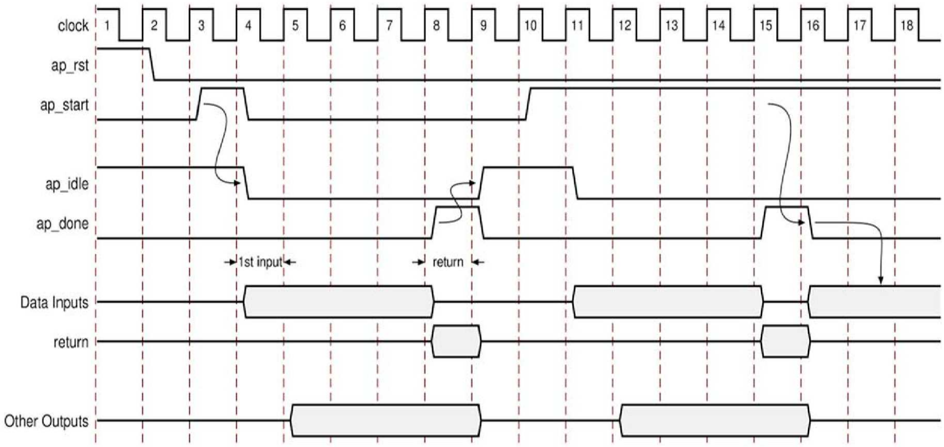


Figure 3.2: Behaviour of ap_ctrl_hs interface [28]

Interface modes `ap_ctrl_hs` and `ap_ctrl_none` are applied on the function or function return value to specify if the RTL is implemented with block-level handshake signals or not. These signals control when the block can begin execution, when it is ready for new data and when it completes. By default these signals are made external to the module, but could be as well made accessible through an internal control register.

The behaviour of the handshake signals created by the `ap_ctrl_hs` protocol is illustrated in Figure 3.2 and presented in the following [28]:

- After reset, the block will wait for `ap_start` to go high before it begins operation.
- Output `ap_idle` goes low when `ap_start` is sampled high.
- Data can now be read on the input ports. The first input data may be sampled on the first clock edge after `ap_idle` goes low.
- When the block completes all operations, any return value will be written to port `ap_return`. If there was no function return, there will be no `ap_return` port on the RTL block. Other outputs may be written to at any time until the block completes and are independent of this IO protocol.
- Output `ap_done` goes high when the block completes operation. If there is an `ap_return` port, the data on this port will be valid when the `ap_done` is high. The `ap_done` signal can therefore be used to show when the function return value is valid.
- The `ap_idle` signal goes high one cycle after `ap_done` and remains high until the next time `ap_start` is sampled high, indicating the block should once again begin operation.
- If the `ap_start` signal is high when `ap_done` goes high: the `ap_idle` signal will remain low, the block will immediately start its execution (or next transaction), the next input may be read on the next clock edge.

| **RTL Interface Protocol** | **Bus Interface Protocol** | | | | | | |
|---|---|---|---|---|---|---|---|
| | AXI4 Lite Slave | AXI4 Master | AXI4 Stream | PLB 4.6 Slave | PLB 4.6 Master | FSL | NPI |
| ap_bus | - | X | - | - | X | - | - |
| ap_fifo | - | - | X | - | - | X | - |
| ap_ctrl_hs<br>ap_none<br>ap_vld<br>ap_ack<br>ap_hs | X | - | - | X | - | - | - |
| ap_ovld<br>ap_memory | - | - | - | - | - | - | - |

Table 3.2: RTL port to bus interface mappings

Listing 3.3: *performStreamCollide* initial interface

```
void performStreamCollide(double[SIZE] srcGrid, double[SIZE] dstGrid);
```

## 3.3.2 Specifying the Bus Interfaces

In addition to standard interfaces described in the previous section, VHLS can also automatically add bus interfaces to the RTL design. The bus interfaces are added to the design during the "Export RTL" process, so they are not present in the RTL written after synthesis and do not appear in the synthesis reports.

The type of the bus interface depends on the protocol of the RTL port it is applied to. Table 3.2 is adopted from Vivado User Guide [28] and shows the list of the RTL interface ports and the available bus interfaces which can be connected to them. The `ap_memory` interface does not require a bus interface and can be directly connected to memories (BRAM). Any port with `ap_ovld` interface should be modified to be one of the supported types, for example `ap_hs`, or it cannot be connected to a bus interface.

## 3.3.3 Default Accelerator

The initial interface of the method `performStreamCollide` is shown in Listing 3.3. The method arguments are two arrays of `double` floating point numbers, representing the source and destination grids. By default, arrays are synthesised into memory ports (Figure 3.1). Their size should be fixed at compile time, otherwise the function cannot be synthesised. Connecting an external memory with fixed size directly to these ports results in a tight coupling of the system components and makes the system less flexible. To achieve a modularized and extensible architecture, the memory will be accessed via the bus. The address and location of the grids will be stored in internal configuration registers.

Listing 3.4: *performStreamCollide* with wrapper

```
 1  void performStreamCollide(volatile double *p_srcGrid, volatile double *p_dstGrid) {
 2     ...
 3  }
 4
 5  void lbm_acc(volatile double *bus,
 6     volatile u32 *srcAddr, volatile u32 *dstAddr) {
 7
 8     /* make sure the addresses are aligned */
 9     if (*srcAddr % sizeof(double) != 0 || *dstAddr % sizeof(double) != 0) return;
10
11     performStreamCollide(bus + (*srcAddr)/sizeof(double),
12                          bus + (*dstAddr)/sizeof(double));
13     }
14  }
```

Xilinx Zynq SoC is using the AMBA AXI bus interface for communication between modules in the system. The accelerator implements two AXI interfaces, an AXI master to access the memory and the AXI slave to give CPU access to its configuration registers.

#### 3.3.3.1 AXI Master Interface

Using the master interface, the Accelerator (ACC) accesses the memory controllers to keep the data. The unified bus standard allows to attach different memory controllers to the ACC, provided they all implement an AXI slave interface. This allows a flexible design, in which the memory system could be replaced, without modifying the ACC architecture.

In the current implementation of the system, the ACC will keep the data either in BRAM or in the DDR. In the former case, the ACC accesses the AXI slave BRAM controllers which in turn connect to the BRAM modules in the Programmable Logic (PL). The second option is to connect the ACC to the AXI slave ports of the Processing System (PS), which provide access to the DDR Controller.

The initial interface of the `performStreamCollide` (Listing 3.3) cannot be mapped to a `ap_bus` port. Figure 3.1 suggests that to implement a bus protocol, the function argument should be a pointer. This will allow both reading and writing to it. Vivado HLS (VHLS) requires the pointer to be declared `volatile` if it will be accessed multiple times. Thus, the first modification is to convert the *array* into a *volatile pointer* type (Listing 3.4 line 1). The modified interface remains compatible with the initial one.

In the C program, the pointer values of `srcGrid` and `dstGrid` are the memory locations of the source and destination grids respectively. However, when these arguments are mapped to bus ports, in the resulting RTL there is no information about the addresses of the grids. Both bus interfaces are implemented to have a base address of zero. This means the actual addresses of the grids should be explicitly added as an offset to `srcGrid` and `dstGrid` variables.

The ACC accesses the memory through a single bus interface. This requires changing the existing interface of performStreamCollide. To keep a clean coding style and ensure backward compatibility, the original interface is left unaltered, instead a wrapper lbm_acc is build around the original function (Listing 3.4). The volatile double *bus variable represents the master port of the ACC. The offsets specifying the addresses of source and destination grids are added to the bus variable and the results are passed as arguments to the performStreamCollide.

#### 3.3.3.2 AXI Slave Interface

The slave interface is used by the Processing System (PS) to set up and control the Accelerator (ACC). As previously discussed, the wrapper function should at least contain three arguments, the bus variable representing the master port and two pointer variables srcAddr and dstAddr that would contain the addresses of the source and destination grids respectively (Listing 3.4). Note that the value referenced by the pointer contains the actual data.

The address variables implement the ap_hs protocol, that provides both a valid and an acknowledge signal. By default each of them will be synthesised as separate AXI slave ports. To save pins and simplify the design, all configuration variables are bundled into a single AXI slave port. ACC stores these variables in internal registers, that can be accessed by the slave interface, specifying the address of the register.

The accelerator implements the so called “function level hand-shake protocol”. The behaviour of this protocol is detailed in Figure 3.2. It specifies three signals, which describe the status of accelerator’s Finite-State Machine (FSM). These signals are used by the PS to control when the ACC can start execution and to be notified when the operation has finished. By default, these signals are synthesised as external inputs and outputs. By specifying the register option on the ap_ctrl_hs protocol, the control signals can be accessed using the accelerator’s status register, which is also mapped to the AXI Slave interface. The control signals are:

- ap_start - input signal. When set to “high”, accelerator starts its execution.
- ap_idle - output signal. Is kept “low” during accelerator’s execution and “high” when it is idle.
- ap_done - output signal. Becomes “high” when the accelerator has finished its execution. Cleared on read.

#### 3.3.3.3 First Optimization

In the original SPEC implementation of LBM, the performStreamCollide is called multiple times in a loop, swapping the source and destination grids after each iteration (Listing 3.5). The call to performStreamCollide at line 11 is replaced by the call to the accelerator (Listing 3.6). To reduce the overhead of setting up

Listing 3.5: LBM original implementation

```
void LBM_swapGrids( LBM_GridPtr* grid1, LBM_GridPtr* grid2 ) {
   LBM_GridPtr aux = *grid1;
   *grid1 = *grid2;
   *grid2 = aux;
}

int main() {
   ...
   int t;
   for( t = 1; t <= param.nTimeSteps; t++ ) {
      performStreamCollide(*srcGrid, *dstGrid);
      LBM_swapGrids( &srcGrid, &dstGrid );
   }
   ...
}
```

Listing 3.6: Calling the accelerator

```
XLbm_acc_SetP_srcgrid(&acc, srcAddr);
XLbm_acc_SetP_dstgrid(&acc, dstAddr);
XLbm_acc_SetNsteps(&acc, param.nTimeSteps);
XLbm_acc_Start(&acc);
while (!XLbm_acc_IsDone(&acc));
```

and calling the accelerator in every iteration, the loop is moved inside the wrapper and the number of iterations is passed as an argument.

Notice that the swapping is done in a “hardware-friendly” way (Listing 3.7). Normally in software this is achieved by simply swapping the values of the variables (see `LBM_swapGrids` in Listing 3.5), but VHLS erroneously interprets it and “optimizes the unnecessary” swapping by just assigning one variable to another. The `srcAddr` and `dstAddr` are merged as if pointing to the same memory location, which is not the expected behaviour. Instead, the swapping is done explicitly, by changing the source and destination grid addresses every odd iteration (Listing 3.7 lines 12-18).

#### 3.3.3.4 Synthesising the Accelerator

Vivado HLS (VHLS) uses special directives to set up the synthesis process. They annotate function’s parameters, to specify the module’s interface, and pieces of code, to implement loop unrolling, pipelining and other optimization features. The VHLS directives can be specified in the source code, using `#pragma` pre-processor directives, or in the `directives.tcl` file of the solution. The Tcl file should normally be modified using the GUI, but one could as well directly create his own directives file and include it in the main `script.tcl` file of the solution.

Listing 3.8 presents the Tool Command Language (Tcl) script to configure the

Listing 3.7: Moving the main loop inside the wrapper

```c
void performStreamCollide(volatile double *p_srcGrid, volatile double *p_dstGrid) {
    ...
}

void lbm_acc(volatile double *bus,
    volatile u32 *srcAddr, volatile u32 *dstAddr, u32* nsteps) {
    int i;
    /* make sure the addresses are aligned */
    if (*srcAddr % sizeof(double) != 0 || *dstAddr % sizeof(double) != 0) return;
    for (i = 0; i < *nsteps; i++) {
        /* do the swap */
        if (i % 2 == 0) {
            performStreamCollide(bus + (*srcAddr)/sizeof(double),
                                 bus + (*dstAddr)/sizeof(double));
        } else {
            performStreamCollide(bus + (*dstAddr)/sizeof(double),
                                 bus + (*srcAddr)/sizeof(double));
        }
    }
}
```

Listing 3.8: Tcl configuration script

```tcl
 1  # implement the function level protocol for lbm_acc
 2  # and create a status register for the signals.
 3  set_directive_interface -mode ap_ctrl_hs -register "lbm_acc"
 4  # map the status register to the AXI Slave port
 5  set_directive_resource -core AXI4LiteS "lbm_acc" return
 6  # the "bus" parameter should implement the bus interface
 7  # and specifically the AXI Master.
 8  set_directive_interface -mode ap_bus -depth 36000 "lbm_acc" bus
 9  set_directive_resource -core AXI4M "lbm_acc" bus
10  # the other parameters are mapped to AXI Slave
11  set_directive_interface -mode ap_hs "lbm_acc" srcAddr
12  set_directive_resource -core AXI4LiteS "lbm_acc" srcAddr
13  set_directive_interface -mode ap_hs "lbm_acc" dstAddr
14  set_directive_resource -core AXI4LiteS "lbm_acc" dstAddr
15  # ... other accelerator's parameters are mapped in a similar way
```

synthesis process. For the sake of clarity, only the mapping for a few of accelerator configuration parameters is illustrated, the others are mapped in a similar way. The comments describe each command. The `depth` attribute at line 8 specifies the maximum number of accesses to the bus. For two $10 \times 10 \times 5 \times 20$ grids, each with two $2 \times 10 \times 10 \times 20$ margin spaces, the total number of bus accesses is 36000. This attribute is used by the post-synthesis verification described in the Section 3.4.

The design is synthesised by pressing the respective button in the VHLS GUI or calling the tool from the console:

```
$ vivado_hls -f <path to project>/solution/script.tcl
```

Listing 3.9: Source grid buffer

```
#ifdef USE_BUFFER
    uint8_t j;
    /* buffer SRC grid accesses */
    load_buffer:
    for (j = C; j < N_CELL_ENTRIES; j++) {
        srcBuffer[j] = LOCAL(srcGrid, j);
    }
/* redirect all accesses to buffer */
#define LOCAL(g, e) srcBuffer[e]
#endif
```

Listing 3.10: Appendix to the Tcl configuration script

```
# loop that is loading the buffer can be unrolled completely
set_directive_unroll "performStreamCollide/load_buffer"
# pipeline the main loop
set_directive_pipeline "performStreamCollide/main_loop"
# spend more effort to find the best schedule
config_schedule -effort high
# binding with ’-effort high’ produces negligible difference,
# but takes too much time to complete.
# ’-min_op’ option is used to minimize the number of adders and multipliers.
config_bind -effort low -min_op add,mul
```

### 3.3.4 Optimized Accelerator

Vivado HLS (VHLS) can be instructed to perform certain optimizations when synthesising the RTL. This is done by using directives, similar to the ones described in the previous subsection.

The LBM algorithm is implemented as a loop that computes a new value for every cell of the source grid and stores it in the destination grid. Because the computations for every cell are similar and to some degree independent, the algorithm could be optimized by a pipeline.

Pipelining is a technique that allows multiple instructions to simultaneously execute in the data path. The efficiency of the pipeline is limited by the number of inter-dependencies between instructions. To reduce those, memory accesses to the source grid are buffered in a small, fast, multi-ported local memory. In every iteration of the loop, the buffer contains the values of a small neighbourhood of the current cell (see Listing 3.9). The neighbours of the current cell are stored in consecutive memory locations, which would allow to read all the values in a single burst transaction. To specify a burst read or write, the memory should be accessed by the `memcpy` C function. Burst transactions are indeed faster than the equivalent number of single transactions. However, the simulations showed that using the burst read operation results in a less efficient design, because this transaction cannot be pipelined.

In addition to pipelining and buffering, the VHLS is instructed to perform scheduling and binding in the most efficient manner. Listing 3.10 presents the directives to perform these optimizations, which are appended to the `directives.tcl` file described in the previous subsection. Notice that the loops are addressed by labels.

### 3.3.5 Dual-Port Accelerator

Xilinx Zynq features a multi-ported DDR controller, which enables the Processing System (PS) and the Programmable Logic (PL) to have shared access to the DDR memory [32]. The DDR controller features four AXI slave ports for this purpose (Figure 2.3 page 22):

- One 64-bit port is dedicated for the ARM CPU(s) via the L2 cache controller and can be configured for low latency.
- Two 64-bit ports are dedicated for PL access.
- One 64-bit AXI port is shared by all other AXI masters via the central interconnect.

The multi-ported architecture of the DDR controller allows multiple masters to access the memory simultaneously or nearly simultaneously. This fact motivates an accelerator implementation that would feature two separate AXI master ports for reading from and writing to the memory. These ports could be connected to the two slave ports dedicated for PL access.

The interface of `performStreamCollide` naturally maps onto a dual-ported architecture. The accesses to source and destination grids are completely independent:

- The grids are stored in different memory locations
- The source grid is only read from and the destination grid is only written to.

A dual-ported architecture eliminates some of the data dependencies by introducing separate channels for reading and writing data. This optimization enables the pipeline to work more efficiently, since two memory accesses can occur simultaneously.

Listing 3.11 presents the top level function implementing the dual-ported architecture. Separate bus ports are used for the source and destination grids. The `directives.tcl` file is modified accordingly (Listing 3.12). Note that the `depth` attribute for each bus port is half the original one, since it represents the maximum number of accesses for only one grid.

Listing 3.11: Wrapper with dual-port architecture

```c
void performStreamCollide(volatile double *p_srcGrid, volatile double *p_dstGrid) {
    ...
}

void lbm_acc(volatile double *src_bus, volatile double *dst_bus,
    volatile u32 *srcAddr, volatile u32 *dstAddr, u32* nsteps) {
    int i;
    /* make sure the addresses are aligned */
    if (*srcAddr % sizeof(double) != 0 || *dstAddr % sizeof(double) != 0) return;
    for (i = 0; i < *nsteps; i++) {
        /* do the swap */
        if (i % 2 == 0) {
            performStreamCollide(src_bus + (*srcAddr)/sizeof(double),
                              dst_bus + (*dstAddr)/sizeof(double));
        } else {
            performStreamCollide(dst_bus + (*dstAddr)/sizeof(double),
                              src_bus + (*srcAddr)/sizeof(double));
        }
    }
}
```

Listing 3.12: Tcl configuration script for Dual ACC

```tcl
# implement the function level protocol for lbm_acc
# and create a status register for the signals.
set_directive_interface -mode ap_ctrl_hs -register "lbm_acc"
# map the status register to the AXI Slave port
set_directive_resource -core AXI4LiteS "lbm_acc" return
# the "bus" parameter should implement the bus interface
# and specifically the AXI Master.
set_directive_interface -mode ap_bus -depth 18000 "lbm_acc" src_bus
set_directive_resource -core AXI4M "lbm_acc" src_bus
set_directive_interface -mode ap_bus -depth 18000 "lbm_acc" dst_bus
set_directive_resource -core AXI4M "lbm_acc" dst_bus
# ... the other parameters are unchanged and omitted for brevity
```

## 3.4 Post-synthesis Verification

Verification in VHLS is composed of two steps. Pre-synthesis validation ( Section 3.2) which ensures that the C program correctly implements the desired functionality and the post-synthesis verification which proves the produced RTL code is correct.

Post-synthesis verification of the produced RTL design is performed by means of the co-simulation feature of VHLS. It uses the original C test-bench so there is no need to manually write a RTL one. Co-simulation is 2 to 3 orders of magnitude faster than ordinary RTL simulation [30].

A design can be verified using co-simulation if the following conditions hold:

- The C test-bench is self checking, returning 0 on success and non-zero on failure.

- It is required to use `volatile` qualifier on any function argument accessed multiple times. The exact number of accesses should be specified by the `depth` argument in the directives script.

- The synthesised function should implement the function-level hand-shake protocol.

When the code is adapted to meet the above conditions, the co-simulation may be started by using the respective button in the GUI or directly via the following command. The `verbose` option is useful for debugging purposes.

```
$ cd <project directory>/solution
$ vivado_hls -flow cosim -flow_args "-verbose"
```

## 3.5 Hardware Design

The system assembly is done in XPS. It is used to configure and connect the Zynq Processing System with the Intellectual Property (IP) cores from the Xilinx Embedded IP catalogue as well as with custom IPs, like the LBM accelerator.

When the RTL is successfully verified by the post-synthesis verification, the LBM accelerator can be exported in *pcore* format, to be used in XPS. This is done by pressing the respective button in VHLS GUI. During the export process, the bus interfaces are synthesised. The exported design should be copied to XPS' global or the project's local IP repository, so it appears in the list of available IPs. All the necessary components are added to the design and connected appropriately. To make sure that all connections are correct and there are no conflicts in the memory mapping of peripherals, one may use the Design Rule Check (DRC) feature of XPS. If no errors are found, a bitfile is generated and the design is exported in the SDK.

The CPU and the accelerator should share a memory space to store the source and destination grids, which is either the BRAM or the external DDR memory. The ACC can access the DDR through the AXI slave ports of the Processing System (PS) (Section 2.3). The two highest performance ports are the High-Performance slave port (HP) and the Accelerator Coherency Port (ACP). This results in three system architectures presented in this section:

- The BRAM implementation

- Accessing DDR with High-Performance slave port

- Accessing DDR with Accelerator Coherency Port

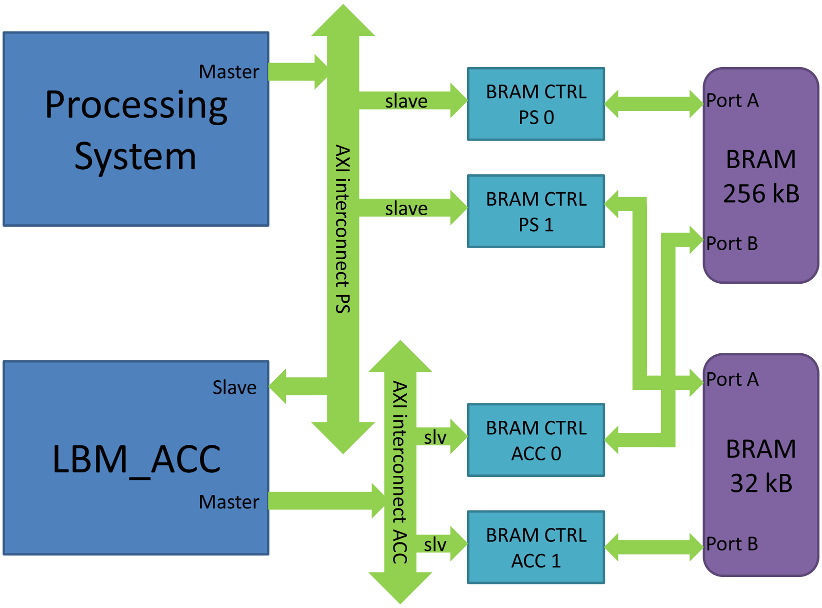


Figure 3.3: System architecture using BRAM

### 3.5.1 The BRAM Implementation

Every Zynq-7000 SoC has between 60 and 465 dual-port Block RAMs (BRAMs), each having 36 kB. Each BRAM has two independent ports that share nothing but the stored data, with a port width of up to 72 bits. The size of the input for the LBM accelerator is 281 kB (see Section 3.1). The maximum size of a single BRAM module for Zynq is 256 kB. To store the data, two BRAM modules mapped to consecutive addresses are added to the design.

Figure 3.3 shows the system architecture of the accelerator using the BRAM to keep the data. The BRAM can be connected to be bus with the help of the AXI BRAM controller. The dual-port nature of the BRAM, allows to share the same block between the CPU and the ACC. One port is connected to the controller accessed by the CPU and the other port is connected to the controller accessed by the ACC. Each AXI master in the system should have its own AXI interconnect. It can have several slave modules connected to it, but my experiments failed for a multi-master design.

The Processing System (PS) provides four clock and reset signals that can be used in the Programmable Logic (PL). It is important that all the components in the system are connected to the same clock source and its respective reset signal. This applies to the ACC, the AXI interconnects and the BRAM controllers. The maximum clock frequency is 200 MHz, but in practice it is limited by the maximum operating frequency of the accelerator.

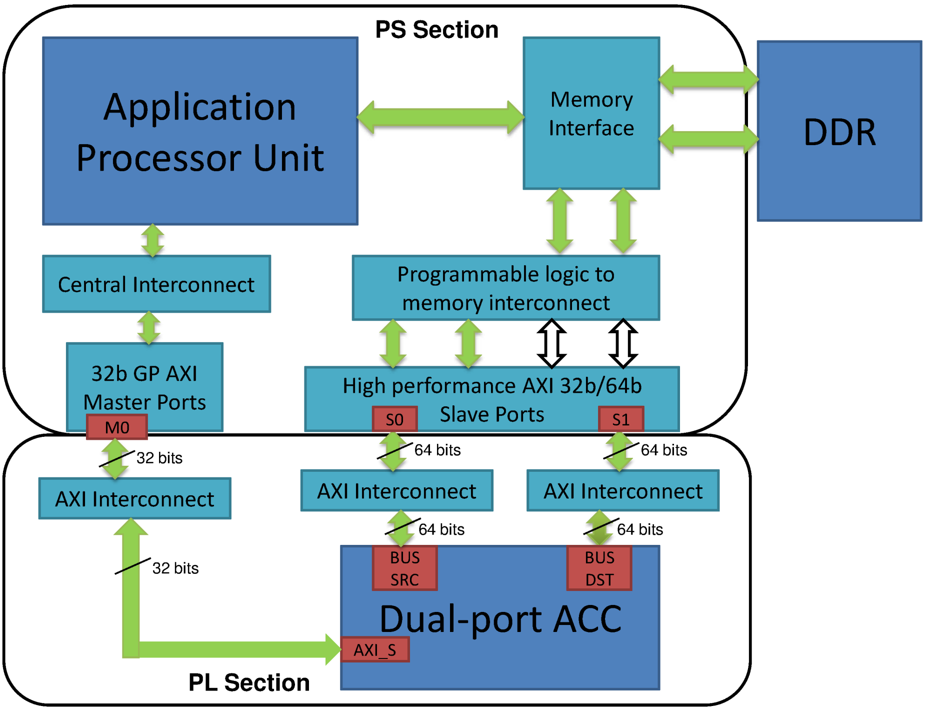


Figure 3.4: System architecture using HP

### 3.5.2 Accessing DDR with High-Performance Slave Ports

Figure 3.4 shows the block diagram of the dual-ported accelerator connected to the DDR memory (the single ported version is connected in a similar way). Xilinx Zynq features four configurable 32-bit/64-bit AXI slave interfaces optimized for high bandwidth access from PL to external memory. These interfaces access the memory interconnect which has two separate channels to access the DDR controller. This allows for both single and dual-ported implementations of the accelerator to efficiently access the data in the DDR.

While connecting to a BRAM controller using the accelerator's AXI master interface needs no special customization, accessing the HP ports does not work by default. I could not find any documentation that would explicitly describe how to access the HP ports. As far as the existing documentation is concerned, the HP ports are regular AXI slave ports and they should be accessed no different than an AXI slave BRAM controller. To analyse the problem, I consulted a reference design.

The "Zynq-7000 All Programmable SoC: Concepts, Tools and Techniques (CTT)" [29] contains a reference design of an AXI Central DMA (CDMA) that is connected to the HP ports of the PS. In this system, AXI CDMA acts as a master device to copy an array of data from the source buffer location to the destination buffer

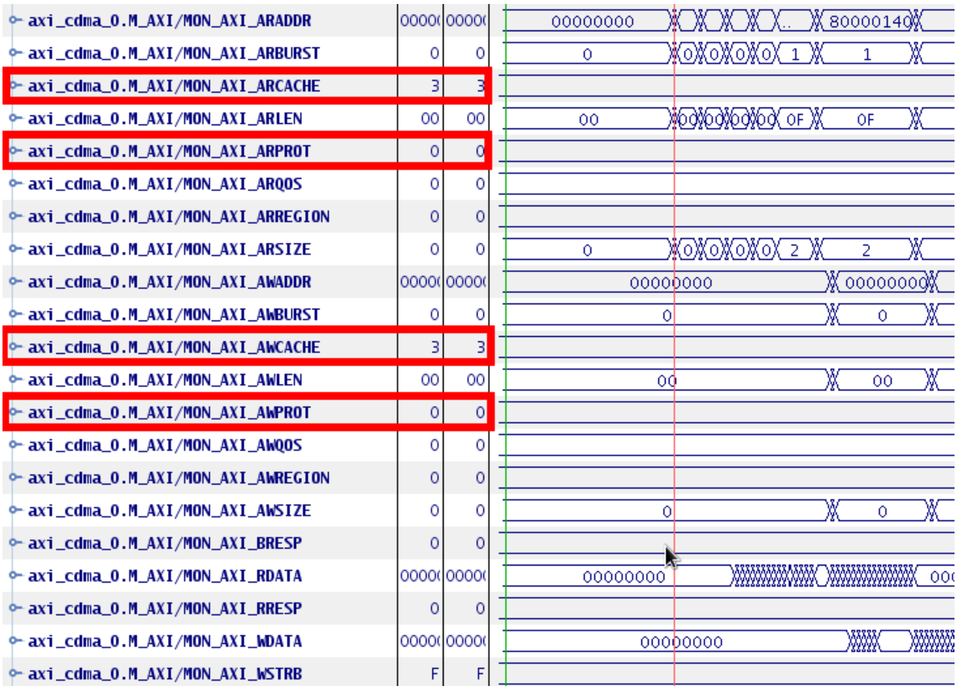

axi_cdma_0.M_AXI/MON_AXI_ARADDR
axi_cdma_0.M_AXI/MON_AXI_ARBURST
axi_cdma_0.M_AXI/MON_AXI_ARCACHE
axi_cdma_0.M_AXI/MON_AXI_ARLEN
axi_cdma_0.M_AXI/MON_AXI_ARPROT
axi_cdma_0.M_AXI/MON_AXI_ARQOS
axi_cdma_0.M_AXI/MON_AXI_ARREGION
axi_cdma_0.M_AXI/MON_AXI_ARSIZE
axi_cdma_0.M_AXI/MON_AXI_AWADDR
axi_cdma_0.M_AXI/MON_AXI_AWBURST
axi_cdma_0.M_AXI/MON_AXI_AWCACHE
axi_cdma_0.M_AXI/MON_AXI_AWLEN
axi_cdma_0.M_AXI/MON_AXI_AWPROT
axi_cdma_0.M_AXI/MON_AXI_AWQOS
axi_cdma_0.M_AXI/MON_AXI_AWREGION
axi_cdma_0.M_AXI/MON_AXI_AWSIZE
axi_cdma_0.M_AXI/MON_AXI_BRESP
axi_cdma_0.M_AXI/MON_AXI_RDATA
axi_cdma_0.M_AXI/MON_AXI_RRESP
axi_cdma_0.M_AXI/MON_AXI_WDATA
axi_cdma_0.M_AXI/MON_AXI_WSTRB
00000000
80000140
0
1
00
0F
2
3
F

Figure 3.5: ChipScope waveforms for AXI CDMA

location in DDR system memory.

To analyse the traffic on the master port of the CDMA I used the ChipScope [26] tool. During system assembly in XPS, a ChipScope AXI monitor is added to the design and attached to the AXI master port of the CDMA to intercept the values of all associated signals. The relevant section of the waveforms is illustrated in Figure 3.5. The highlighted signals are different than the defaults implemented by the Vivado HLS:

```
# ACC AXI Master port          # AXI CDMA Master port

AXI_CACHE = 0b0000             AXI_CACHE = 0b0011
AXI_PROT  = 0b010              AXI_PROT  = 0b000
```

The `AXI_PROT` is used to set the `ARPROT` and `AWPROT` signals. These bits are known as the Non-Secure, or NS bits, and are defined in the public AMBA AXI bus protocol specifications [4]:

- `AWPROT[1]`: Write transaction - low is Secure and high is Non-secure
- `ARPROT[1]`: Read transaction - low is Secure and high is Non-secure

All bus masters set these signals when they make a new transaction, and the bus or slave decode logic must interpret them to ensure that the required security

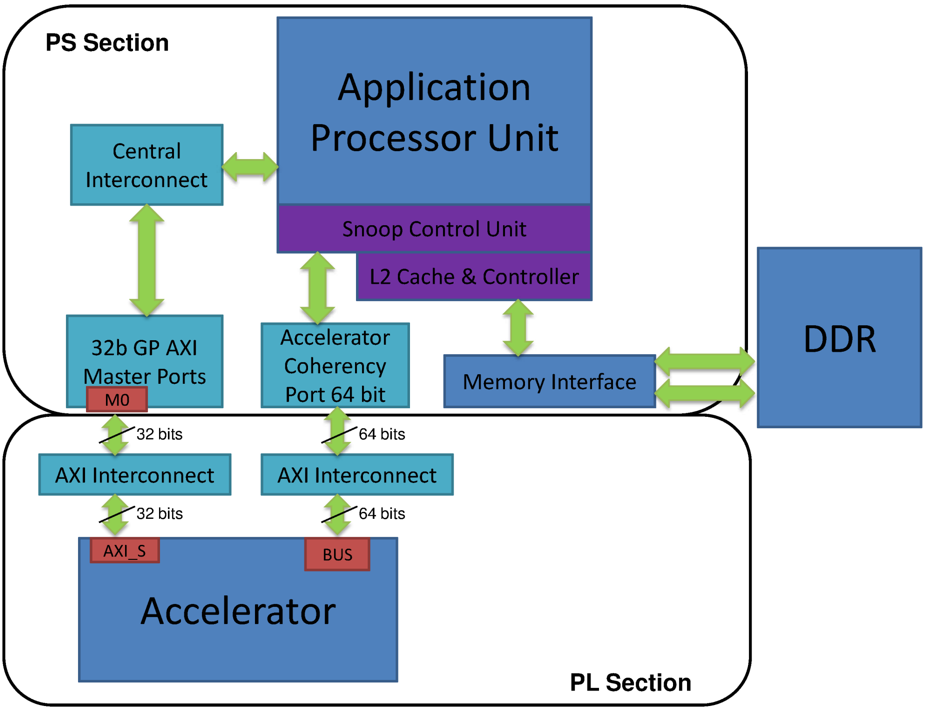


Figure 3.6: System architecture using ACP

separation is not violated. All Non-secure masters must have their NS bits set high in the hardware, which makes it impossible for them to access Secure slaves. The address decode for the access will not match any Secure slave and the transaction will fail [4]. Apparently the HP ports are defined as Secure and this explains why the default AXI Master port of the ACC failed to access these slave ports.

Xilinx recommends that master devices drive their `AW/RCACHE` outputs to `0b0011` to allow the AXI interconnect core to pack data while performing width conversion and to allow store-and-forward in datapath FIFOs [27].

After setting the right values for `AXI_CACHE` and `AXI_PROT` parameters of the AXI Master interface, the ACC worked as expected and was able to access the DDR using the HP ports of the Processing System.

### 3.5.3 Accessing DDR with Accelerator Coherency Port

The Zynq-7000 Accelerator Coherency Port (ACP) is a 64-bit AXI Slave interface that directly connects the PL to the Snoop-Control Unit (SCU) of the ARM Cortex-A9 processors, enabling cache-coherent access to CPU data in the L1 and L2 caches. The ACP provides a low latency path between the PS and the PL-based accelerator when compared with a legacy cache flushing scheme [32].

The read and write requests performed on the ACP behave differently depending on whether the request is coherent or not. ACP requests behaviour is as follows [5]:

- **ACP coherent read requests** – An ACP read request is coherent when `ARUSER[0] = 1` and `ARCACHE[1] = 1` alongside `ARVALID`. In this case, the SCU enforces coherency.
  - When data is present in one of the Cortex-A9 processors, the data is read directly from the relevant processor and returned to the ACP port.
  - When data is not present in any of the Cortex-A9 processors, the read request is issued on one of the Cortex-A9 MPCore master ports, along with all its AXI parameters, with the exception of the locked attribute.
- **ACP coherent write requests** – An ACP write request is coherent when `AWUSER[0] = 1` and `AWCACHE[1] = 1` alongside `AWVALID`. In this case, the SCU enforces coherency.
  - When the data is present in one of the Cortex-A9 processors, the data is first cleaned and invalidated from the relevant CPU.
  - When the data is not present in any of the Cortex-A9 processors, or when it has been cleaned and invalidated, the write request is issued on one of the Cortex-A9 MPCore AXI master ports, along with all the corresponding AXI parameters with the exception of the locked attribute.
- **ACP non-coherent read/write requests** – An ACP read/write request is non-coherent when `A(R/W)USER[0] = 0` or `A(R/W)CACHE[1] = 0` alongside `A(R/W)VALID`. In this case, the SCU does not enforce coherency, and the write request is forwarded directly to one of the available Cortex-A9 MPCore AXI master ports.

The `A(R/W)USER[0]` bit defines if the memory region accessed by the AXI master is shared by several processors (value 1) or is used only by a single processor (value 0). The bits `A(R/W)USER[4:1]` have the following meaning [5]:

- `0b0000` – Strongly ordered: All memory accesses to Strongly Ordered memory occur in program order. These address locations are *not held in a cache* and are treated as Shared memory locations.
- `0b0001` – Device: Designed to handle memory-mapped peripherals. These address locations are *not held in a cache.*
- `0b0011` – Normal Memory Non-Cacheable: Designed to handle normal memory. Memory accesses conform to Weakly Ordered model of memory ordering.
- `0b0110` – Normal Memory Write-Through
- `0b0111` – Normal Memory Write-Back no Write-Allocate
- `0b1111` – Normal Memory Write-Back Write-Allocate

The last three options are used when *cached* access is required and define the respective cache policy.

Figure 3.6 illustrates the accelerator connected to the ACP port. It is configured to use the cache and to issue coherent memory transactions by setting the `A(R/W)USER = 0b11111` and `A(R/W)CACHE = 0b0011`.

Listing 3.13: Simple HelloWorld application for Zynq

```
/*
 * helloworld.c: simple test application
 */

#include <stdio.h>
#include "platform.h"

void print(char *str);

int main()
{
    init_platform();
    print("Hello World\n\r");
    cleanup_platform();
    return 0;
}
```

## 3.6 Software Design

The LBM application will be running bare-metal rather than as a Linux program. This results in more precise performance and power measurements, because no other application or OS routine will interfere with LBM. The software design is performed using Xilinx Software Development Kit (SDK). It is used to compile a software executable that could run on Xilinx Zynq. To be able to run the application on the target device, a Standalone Board Support Package (BSP) is added to the project. It contains all the necessary libraries for running the application bare-metal along with configuration headers, that specify the addresses of the peripherals.

A simple “HelloWorld” Xilinx C application is created and linked to the previously added BSP (Listing 3.13). This program provides the basic functionality of initializing the system, writing some text to the UART and cleaning up. It will serve as a stub for the LBM application. LBM’s source files are adapted to be able to run on the accelerated system. This mostly concerns the `main` function from `main.c` which is presented in the Methodology chapter.

# Chapter 4

# Methodology

The hardware accelerator is implemented on the Zedboard [35], that features a Xilinx Zynq$^{\text{TM}}$-7000 All Programmable SoC. The ARM Cortex$^{\text{TM}}$-A9 together with the FPGA on the same chip provide a powerful and flexible platform for implementing high-performance accelerator applications. The OS running on the board is Xillinux [34], a Linux distribution based on Ubuntu 12.04 and customized for the Zedboard.

The development is done on a host machine, with Intel® Core$^{\text{TM}}$2 Quad CPU Q9400 2.66GHz and 4 GB of main memory. It is running an CentOS 6.4 i686 Linux and Xilinx Design Suite 14.2. Power measurements are performed using an Agilent 34410A multimeter.

The Zedboard features a UART and a JTAG controller to communicate with the host computer. These controllers have a USB interface, so they can be connected using simple USB cables. The UART is used for transferring messages between Zynq and the development computer. The UART is configured with a baud rate of 115200 bps, 8 data bits, no parity and one stop bit. The host computer is running `minicom` [15] to communicate via UART. The JTAG controller allows to program and debug the Zynq from the host computer. The cable driver that comes with the Xilinx tool suite does not work, so the Digilent plug-in should be installed [9].

## 4.1 Power Measurements

### 4.1.1 Measurement Setup

Zedboard features a $10m\Omega$, 1W current sense resistor. Header J21 straddles this resistor to measure the voltage across it for calculating power. Every time this voltage is sampled, the power is computed using the following formula:

$$P = (V_{in} - V_{zed}) * (V_{zed}/R) \tag{4.1}$$

where $V_{in}$ is the input voltage 12V, $V_{zed}$ is the voltage measured across the resistor having $R = 10m\Omega$.

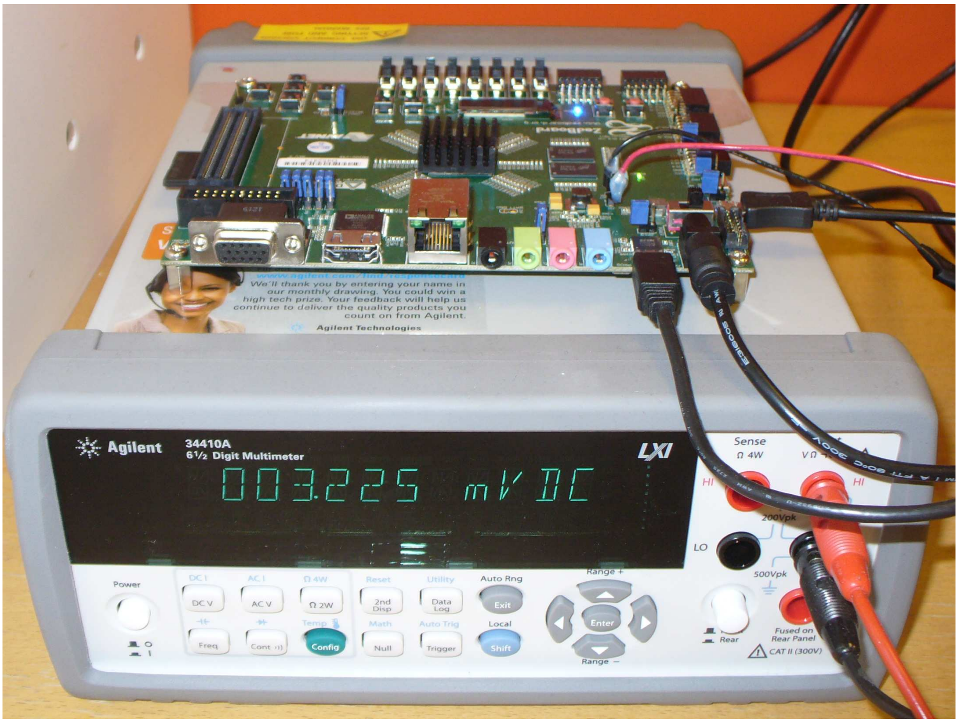


Figure 4.1: Power measurement setup

Figure 4.1 illustrates the power measurement setup. For sampling the voltage, an Agilent 34410A multimeter is used. It is connected to the host machine through a USB cable. The multimeter is installed as a Linux character device. It is programmed by means of Standard Commands for Portable Instruments (SCPI) language [19]. To read the voltage measurements a *bash* script is executed (see Listing 4.1). It sends a SCPI instruction, requesting a sample and outputs the response. The start and end times are logged to measure the execution time. The script runs continuously until it is interrupted by an external signal. Notice that the interrupt should be "trapped" to allow a graceful termination of the script. The sequence inside the while loop should be atomic, otherwise the device file may become corrupt, leading to a "broken pipe" error.

After the measurement session is over, the output file is parsed to calculate the average power consumption:

$$P_{avg} = (\sum_{i=1}^{n} P_i)/n \tag{4.2}$$

where $P_i$ is the power computed for every voltage sample, and $n$ is the total number of samples.

Listing 4.1: Script to read the measurements

```
#!/bin/bash

trap ’setit’ SIGINT SIGTERM SIGQUIT
setit() {
   stop="1"
}
echo "start "$(date +%s.%N)
while [ "$stop" != "1" ]; do
   echo MEAS? > /dev/usbtmc1 #request a sample from the multimeter
   voltage=‘cat /dev/usbtmc1‘ #read the response
   echo $voltage
done
echo "stop "$(date +%s.%N)
```

### 4.1.2 Measuring Idle Power

Zedboard is not equipped for fine grained power measurements. It contains only one current-sense resistor to measure the power of the complete board. To compare the power consumption of different configurations of the system, the total power consumption is broken down into *idle* and *active* parts.

The *active* power consumption is the one that will be used in all the energy and EDP calculations. It is an approximation of the power added by a given configuration of the system to the baseline *idle* power.

The *idle* power consumption serves as a common reference for all the measurements in the experiments presented in this work. To measure *idle* power the following steps are taken after the board is powered on:

1. The FPGA is programmed using the bitfile.
2. All the enabled peripherals are initialized and their clocks are powered on.
3. One of the ARM cores is disabled and the second one is clocked down to the minimum value of 20 MHz.
4. The FPGA clock is turned off.

The CPU cannot be turned off completely, because it is the central master of the system and coordinates the functionality of the whole design.

### 4.1.3 Measuring Application Power

The application code is running on Zynq bare-metal. To be able to debug it, Xilinx Microprocessor Debugger (XMD) is used. Listing 4.2 shows the Tcl file used by `xmd` to setup the Zynq. The lines starting with “#” are comments explaining every instruction. After the Tcl file is sourced from the XMD console, it starts a `gdb` (GNU Debugger) server. To measure the power consumed by the Zedboard when executing LBM, the measurement script (Listing 4.1) should be launched at the

Listing 4.2: XMD commands script

```
# program the FPGA
fpga -f <path to bitfile>/system.bit
# initialize the peripherals
source <path to hw folder>/ps7_init.tcl
ps7_init
# download the elf file
dow <path to executable>/lbm.elf
```

Listing 4.3: Debug with GDB

```
# read the executable
file <path to executable>/lbm.elf
# connect to xmd's gdb server, by default it runs on port 1234
target remote localhost:1234
# put a breakpoint just before the algorithm starts
break main.c:100
# provide the shell commands to execute on hitting this breakpoint
commands
# remember to run it as a background process
shell <path to script>/get.sh > output.log &
# immediately continue execution
continue
end
# breakpoint just after the algorithm ends
break main.c:150
commands
# kill the script
shell killall get.sh
continue
end
# set program counter to 0
set $pc=0
# launch the executable
continue
```

beginning of the algorithm and killed at the end. This can be achieved by using `gdb` and executing shell scripts when certain breakpoints are hit. Let's say the algorithm starts at line 100 and finishes at line 150 of `main.c` file. Listing 4.3 shows how to connect to `xmd`'s `gdb` server and setup the breakpoints for controlling the measurement script. This file should be sourced from the `gdb` console. The script will log all the voltage measurements together with the start and end times. It is post-processed to compute the average power consumption.

Listing 4.4: Compiling the application for different HW

```
int main() {
    /* ... code omitted for brevity ... */
#ifdef USE_BRAM
    /* Store grids in BRAM */
    src_bus = (double*)XPAR_BRAM_1_BASEADDR;
#else
    /* Store grids in main memory */
    /* Add the offset for stack and heap */
    src_bus = (double*)XPAR_PS7_DDR_0_S_AXI_BASEADDR + 0xA00000;
#endif
    src_bus += MARGIN; /* Allocate margin space */
    /* Set address for the destination grid */
    dst_bus = src_bus + GRID_SIZE;
    srcGrid = &src_bus;
    dstGrid = &dst_bus;
    /* ... code omitted for brevity ... */
#ifdef USE_ACC
    /* Setup the accelerator */
    XLbm_acc_SetP_srcgrid(&acc, (u32)*srcGrid);
    XLbm_acc_SetP_dstgrid(&acc, (u32)*dstGrid);
    XLbm_acc_SetNsteps(&acc, param.nTimeSteps);
    /* Start the accelerator */
    XLbm_acc_Start(&acc);
    /* Wait until the accelerator has finished execution */
    while (!XLbm_acc_IsDone(&acc));
#else
    int t;
    for(t = 1; t <= param.nTimeSteps; t++){
        performStreamCollide(*srcGrid, *dstGrid);
        LBM_swapGrids(&srcGrid, &dstGrid);
    }
#endif
    /* ... code omitted for brevity ... */
}
```

## 4.2 Running on Different Hardware

To compare the performance and energy efficiency, the application is run on four hardware configurations:

1. On ARM with data in DDR (main memory)
2. On ARM with data in BRAM
3. On Accelerator (ACC) with data in DDR
4. On ACC with data in BRAM

The code is parametrized using C preprocessor directives to either use ARM or ACC, and either keep data in DDR or in BRAM. Listing 4.4 is an extract from the

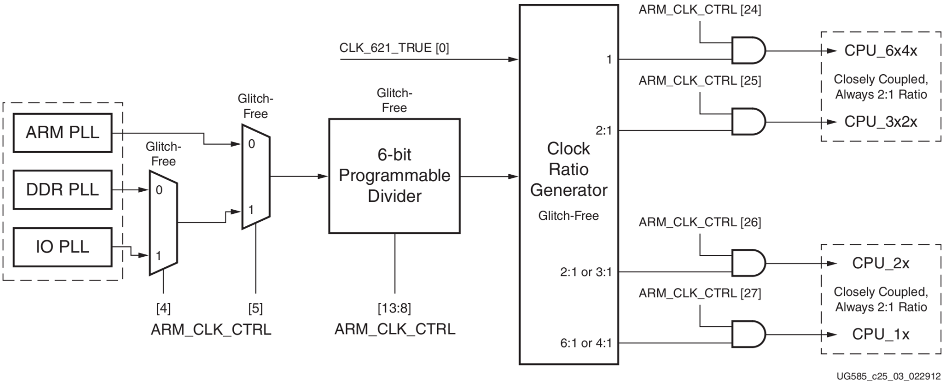


Figure 4.2: CPU clock generation and domains [33]

`main` function, showing the parametrization. Lines 18 to 25 show how the accelerator is called by the CPU. First the source and destination addresses are configured, along with the number of iterations the accelerator should perform. The CPU then writes to the configuration register, to instruct the accelerator to start execution. Afterwards, the CPU polls the status register until the *done* bit is set high. Since the application is running bare-metal, there is no memory protection mechanism and the CPU can access any memory location. The programmer is responsible for ensuring that the code does not overwrite the stack and heap segments of the C program. In this example, an offset of 10 MB is added to the DDR base address.

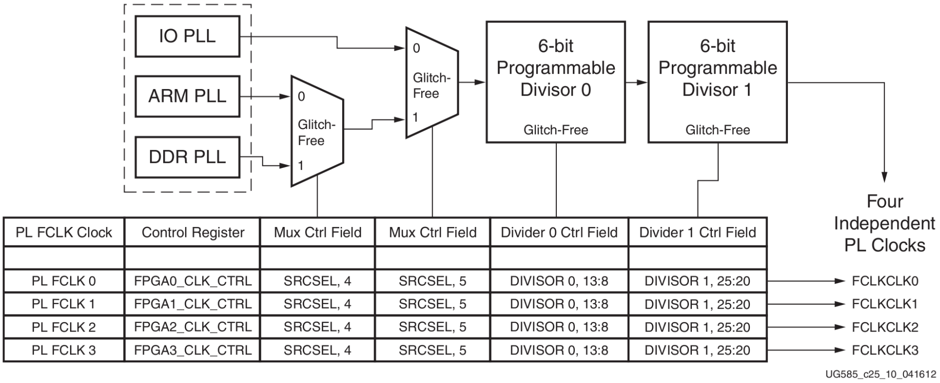
## 4.3 Tweaking Energy-Efficiency

Energy Delay Product (EDP) is used to measure the system's energy-efficiency. It is based on total execution time of the application and the power consumption of the hardware system. The previous chapter focused on developing an energy-efficient architecture, but clock speed has also a big impact on a component, both when it is idle and fully operational. The speed of various clocks in the system can be changed from software using *dynamic clock gating*.

In the current setup, two clocks are used to tweak the performance and power consumption, the CPU (`ARM_CLK`) and the FPGA clocks. Zynq provides 4 different clocks that can be used in the PL substrate, named `FPGA_CLK[0..3]`. The `FPGA_CLK0` is used for all core components: ACC, BRAM controllers and AXI interconnects.

Clocks are generated by Phase Lock Loops (PLL). By applying different divisors, one could dynamically change the clock frequency. Figures 4.2 and 4.3 show the generation of ARM and FPGA clocks.

To minimize the power consumption of the system, regardless of whether the accelerator is used or not, one of the two cores of the CPU is disabled, since the LBM application is single-threaded.



| PL FCLK Clock | Control Register | Mux Ctrl Field | Mux Ctrl Field | Divider 0 Ctrl Field | Divider 1 Ctrl Field |
|---|---|---|---|---|---|
| | | | | | |
| PL FCLK 0 | FPGA0_CLK_CTRL | SRCSEL, 4 | SRCSEL, 5 | DIVISOR 0, 13:8 | DIVISOR 1, 25:20 |
| PL FCLK 1 | FPGA1_CLK_CTRL | SRCSEL, 4 | SRCSEL, 5 | DIVISOR 0, 13:8 | DIVISOR 1, 25:20 |
| PL FCLK 2 | FPGA2_CLK_CTRL | SRCSEL, 4 | SRCSEL, 5 | DIVISOR 0, 13:8 | DIVISOR 1, 25:20 |
| PL FCLK 3 | FPGA3_CLK_CTRL | SRCSEL, 4 | SRCSEL, 5 | DIVISOR 0, 13:8 | DIVISOR 1, 25:20 |

Figure 4.3: PL clock generation [33]

When running the application on the accelerated system, the CPU busy-waits for the accelerator to finish its execution. To save power, the CPU is dynamically clocked down during that period. It is important to mention that the ACC cannot be clocked down when the algorithm runs on the CPU and keeps the data in BRAM, since both BRAM and ACC are connected to the same clock source. Xilinx Zynq allows software control over the clocks and other parameters of the PS by means of SLCR. Different combinations of clock speeds for CPU and FPGA are used to explore the design space to find the most energy-efficient configuration. Listing 4.5 shows the extract from the `main.c` that illustrates the functions used to control the system clocks using SLCR. Note that before SLCR can be modified, it should be unlocked. Dynamic clock gating is controlled by the *power control register* of the System Control Coprocessor CP15. This register is not memory mapped and should be accessed using assembly language.

Listing 4.5: Controlling the system clocks from software

```c
void disable_core() {
    /* unlock SLCR */
    Xil_Out32(SLCR_BASEADDR + SLCR_UNLOCK, 0x0000DF0D);
    /* shutdown core #1 */
    u32 a9_cpu = Xil_In32(SLCR_BASEADDR + A9_CPU_RST_CTRL);
    Xil_Out32(SLCR_BASEADDR + A9_CPU_RST_CTRL, apply_mask(a9_cpu, A9_CLKSTOP1,
         A9_CLKSTOP1));
    /* lock SLCR */
    Xil_Out32(SLCR_BASEADDR + SLCR_LOCK, 0x0000767B);
}

void slowdown_clk() {
    /* unlock SLCR */
    Xil_Out32(SLCR_BASEADDR + SLCR_UNLOCK, 0x0000DF0D);
    /* enable dynamic clock gating */
    u32 pwr_ctrl = 0;
    asm volatile ("mrc p15, 0, %0, c15, c0, 0" : "=r"(pwr_ctrl) : : "cc");
    pwr_ctrl |= 1;
    asm volatile ("mcr p15, 0, %0, c15, c0, 0" : : "r"(pwr_ctrl) : "cc");
    pwr_ctrl = 0;
    asm volatile ("mrc p15, 0, %0, c15, c0, 0" : "=r"(pwr_ctrl) : : "cc");
    /* increase the ARM clk divider */
    u32 arm_clk_ctrl = Xil_In32(SLCR_BASEADDR + ARM_CLK_CTRL);
    orig_arm_div = arm_clk_ctrl & DIVISOR;
    arm_clk_ctrl = apply_mask(arm_clk_ctrl, DIVISOR, arm_div);
    Xil_Out32(SLCR_BASEADDR + ARM_CLK_CTRL, arm_clk_ctrl);
    /* increase the FPGA clk divider */
    u32 fpga0_clk_ctrl = Xil_In32(SLCR_BASEADDR + FPGA0_CLK_CTRL);
    orig_fpga0_div0 = fpga0_clk_ctrl & DIVISOR0;
    fpga0_clk_ctrl = apply_mask(fpga0_clk_ctrl, DIVISOR0, fpga0_div0);
    Xil_Out32(SLCR_BASEADDR + FPGA0_CLK_CTRL, fpga0_clk_ctrl);
}

void restore_clk() {
    /* restore fpga0_div */
    u32 fpga0_clk_ctrl = Xil_In32(SLCR_BASEADDR + FPGA0_CLK_CTRL);
    fpga0_clk_ctrl = apply_mask(fpga0_clk_ctrl, DIVISOR0, orig_fpga0_div0);
    Xil_Out32(SLCR_BASEADDR + FPGA0_CLK_CTRL, fpga0_clk_ctrl);
    /* restore arm_div */
    u32 arm_clk_ctrl = Xil_In32(SLCR_BASEADDR + ARM_CLK_CTRL);
    arm_clk_ctrl = apply_mask(arm_clk_ctrl, DIVISOR, orig_arm_div);
    Xil_Out32(SLCR_BASEADDR + ARM_CLK_CTRL, arm_clk_ctrl);
    /* disable dynamic clock gating */
    u32 pwr_ctrl = 123;
    asm volatile ("mrc p15, 0, %0, c15, c0, 0" : "=r"(pwr_ctrl) : : "cc");
    pwr_ctrl = apply_mask(pwr_ctrl, 1, 0);
    asm volatile ("mcr p15, 0, %0, c15, c0, 0" : : "r"(pwr_ctrl) : "cc");
    pwr_ctrl = 123;
    asm volatile ("mrc p15, 0, %0, c15, c0, 0" : "=r"(pwr_ctrl) : : "cc");
    /* lock SLCR */
    Xil_Out32(SLCR_BASEADDR + SLCR_LOCK, 0x0000767B);
}
```

# Chapter 5

# Results

This chapter presents the results for performance and energy-efficiency of the experiments conducted using different configurations of the hardware. The performance is calculated as the inverse of the execution time. The most commonly used metrics for evaluating the energy-efficiency are the Energy ($E$), Energy Delay Product ($EDP$) and Energy Delay Squared Product ($ED^2P$), which depend on power ($P$) and execution time ($T$) and are calculated as follows:

$$E = P \times T \tag{5.1}$$

$$EDP = E \times T \tag{5.2}$$

$$ED^2P = E \times T^2 \tag{5.3}$$

The choice of $E$ gives and advantage to systems that stress energy consumption over performance. The $EDP$ favours systems that value both performance and energy consumption, and $ED^2P$ favours high performance processors whose design allocates a large expenditure of energy in return for small improvements in performance.

The accelerator presented in this work is designed with the focus on both energy consumption and performance. This fact determined the choice of Energy Delay Product as the metric used in the energy-efficiency calculations. In the following the *energy-efficiency* refers to the inverse of the EDP.

The system can be logically divided into computation and communication parts. In the following I will refer to *computation* part as either CPU or any of the three versions of the Accelerator (ACC) (Default, Optim and Dual). *Communication* will refer to the bus system and the storage device, BRAM or DDR. The terms *CPU* and *ARM* are used interchangeably.

BRAM can only be accessed using an AXI BRAM controller connected to an AXI interconnect. The CPU can access the DDR directly through the memory controller with optionally disabling the L2 cache. To access the DDR, the ACC can use the HP or ACP attached to the AXI interconnect. All the modules on Programmable Logic (PL) (ACC, interconnect, BRAM controller) are powered by

| **Configuration** | **CLK (ns)** | **Power** | **Latency gain** | **Energy gain** | **EDP gain** |
|---|---|---|---|---|---|
| Default | 8.75 | 3915 | 1.00 | 1.00 | 1.00 |
| Default | 4.55 | 10298 | 0.99 | 0.37 | 0.37 |
| Buffer | 8.75 | - | - | - | - |
| Buffer, pipeline 155 | 18 | 4062 | 1.59 | 1.53 | 2.44 |
| Buffer, pipeline 155, scheduler effort high | 18 | 3824 | 1.59 | 1.63 | 2.59 |
| Buffer, unroll 2, pipeline 310 | 18 | 4608 | 1.59 | 1.35 | 2.15 |
| Buffer, unroll 3, pipeline 465 | 18 | 5401 | 1.58 | 1.14 | 1.81 |
| No buffer, unroll 3 | 18 | 5105 | 0.73 | 0.56 | 0.40 |
| Dual default | 8.75 | - | - | - | - |
| Dual default | 4.55 | 15067 | 1.87 | 0.48 | 0.91 |
| Dual, pipeline 201 | 8.75 | 3257 | 7.25 | 8.72 | 63.30 |
| Dual, pipeline 453 | 4.55 | 10338 | 13.95 | 10.84 | 73.75 |
| Dual, buffer, pipeline 443 | 8.75 | 5972 | 23.96 | 15.70 | 376.41 |
| Dual, buffer, pipeline 192, scheduler effort high | 8.75 | 5748 | 23.96 | 16.32 | 391.08 |
| Dual, buffer, pipeline 443 | 4.55 | 21000 | 44.40 | 8.72 | 367.61 |
| Dual, buffer, pipeline 192 | 4.55 | 20371 | 44.40 | 8.53 | 378.96 |

Table 5.1: Characteristics of different ACC implementations

the same clock source. In the following I refer to BRAM *frequency* as the frequency of BRAM controller and that of the AXI interconnect.

To make a fair comparison, the results are grouped by the different computation and communication setups in the following categories:

- **CPU** - BRAM vs DDR + L2 cache vs DDR no cache
- **ACC** - BRAM vs DDR + L2 cache vs DDR no cache
- **BRAM** - ACC vs CPU
- **DDR** - ACC vs CPU

## 5.1 Accelerator Implementations

Synthesis of the LBM accelerator takes just a minute in Vivado HLS. In addition to the RTL description of the module, the tool also generates reports that contain information about the minimum clock cycle period, latency, area and power estimations. This information allows exploring the design space and evaluating different optimization techniques very fast, without the need to physically implement the design in hardware, which may take a couple of hours. Before performing system assembly and logic synthesis, the accelerator passes the post-synthesis verification, which may take another 20-30 minutes.

The purpose of this work was to accelerate an application of choice, while minimally altering its source code. Since changing the internal implementation is discouraged, a wrapper was built to change the interface of the accelerator. Two architectures were designed, one having a single master port to read and write the data and the other one with separate ports for source and destination grids. Since the source grid is only read from and the destination grid only written to, the two ports are synthesised as unidirectional.

Table 5.1 shows the characteristics of the two architectures and the impact of various optimization techniques. The data in the table is based on the post-synthesis reports generated by the VHLS. The red rows represent designs that would not fit on the target device, and the green ones show the most energy-efficient configurations of the single and dual-port architectures. The last three columns, show the gain with respect to the Default ACC, single-port with no special optimization directives applied. For example, the "Latency gain" is the latency of Default ACC divided by the latency of that specific configuration. The number next to "pipeline" shows the number of steps it contains, and the number next to "unroll" specifies how many times is the main loop of the ACC unrolled. The "buffer" optimization refers to the small local memory that buffers the accesses to the source grid. The "scheduler effort high" option instructs the HLS to spend more time to produce a better scheduling of operations.

The results presented in Table 5.1 reveal a couple of observations on how different settings and features of the accelerator affect its performance and energy-efficiency.

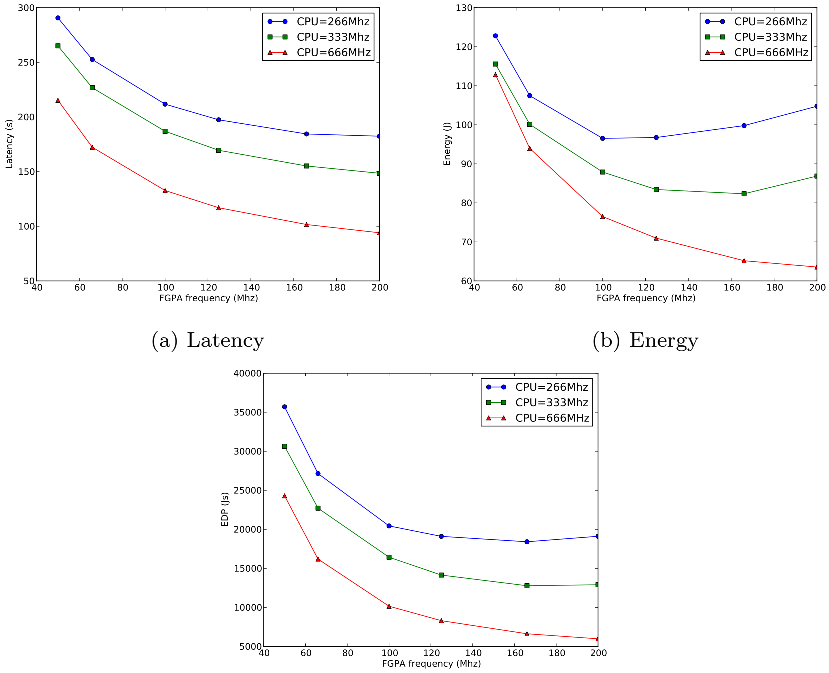
The first observation is that increasing the clock frequency may indeed improve performance but may as well result in a worse energy-efficiency. Only for the "dual, pipeline" configuration, the EDP is improved when the clock speed is increased. For the "Default" accelerator, the latency is not changed when increasing the clock, which determined a worse energy-efficiency.

The second finding is that unrolling the loop does not increase the performance, but only adds to the power consumption. The scheduler cannot extract more parallelism from the unrolled loop, because of the data dependencies.

The third observation is that sometimes the optimizations can increase the minimum clock period, due to dependencies in the pipeline. This phenomenon happens in case of the single-port architecture. Even with a slower clock, the resultant design runs faster and consumes less energy. I would expect VHLS to stall the pipeline, when a dependency prevents the instruction to be fed in. Stalling could be locally inefficient, but would avoid increasing the clock period, thus resulting in a globally more efficient design.

The next observation is that the dual-port implementation does not in itself make the accelerator perform better. The benefits of such an architecture are seen when the pipeline is introduced. Having different ports for reading and writing removes some of the data dependencies in the data-flow graph, thus allowing the pipeline to work more efficiently. This results in a 14x speed-up and 73.75x better energy-efficiency compared to the default, single-ported architecture.

Finally, introducing the small local memory for buffering the accesses to the source grid increases the parallelism and lowers the access latencies. It gives the pipeline even more freedom and results in an impressive 44.4x gain in performance and 379x gain in energy-efficiency.



(a) Latency

(b) Energy

(c) Energy Delay Product

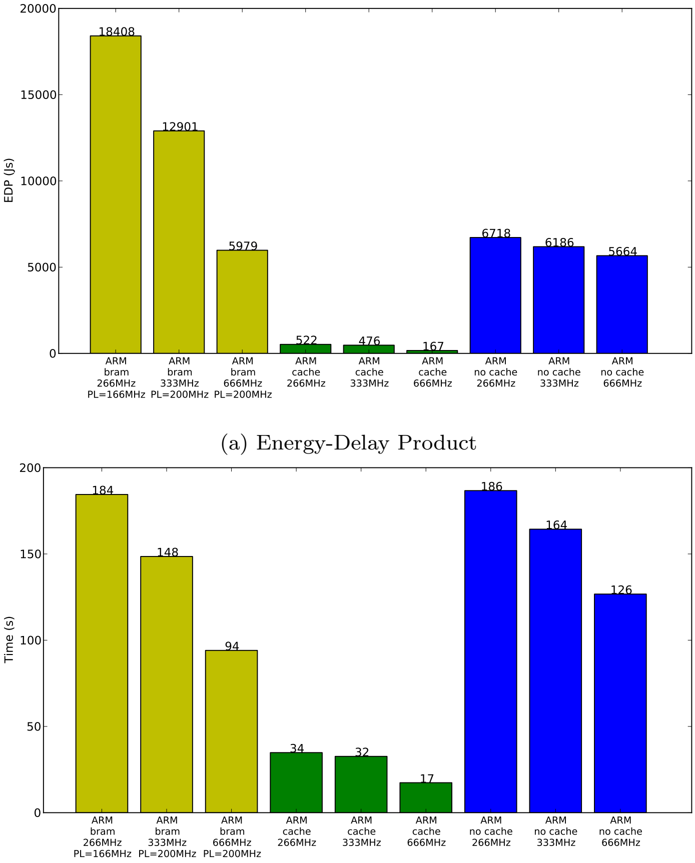
Figure 5.1: Results for CPU with data in BRAM

## 5.2 LBM Running on ARM

The results in this and the following sections are based on the physical power measurements of the system as detailed in Section 4.1. The purpose of this section is to evaluate the initial system, without the accelerator. It discusses the performance and energy-efficiency of the LBM application running on the CPU while keeping the data either in BRAM or DDR. While accessing the DDR, L2 cache is optionally disabled to match the cache-less BRAM configuration.

To compare the BRAM configuration with the DDR ones, I identified the most energy-efficient combinations of CPU and BRAM frequencies (see Figure 5.1c). The optimum for CPU running at 266 MHz is reached when the BRAM is at 166 MHz. In case of CPU at 333 MHz, there a negligible increase in EDP, when speeding up the BRAM from 166 to 200 MHz, so the latter result is chosen, in favour of better performance in terms of latency (see Figure 5.1a). Having these results, I can assume that the most energy-efficient ratio between CPU and BRAM frequencies is about 1.6:1, which means the CPU at 666 MHz would require BRAM to run twice its maximum speed to reach the optimum.

Figure 5.2 compares the results for EDP and latency of different storage options for LBM running on the CPU. For BRAM, the most energy-efficient configurations



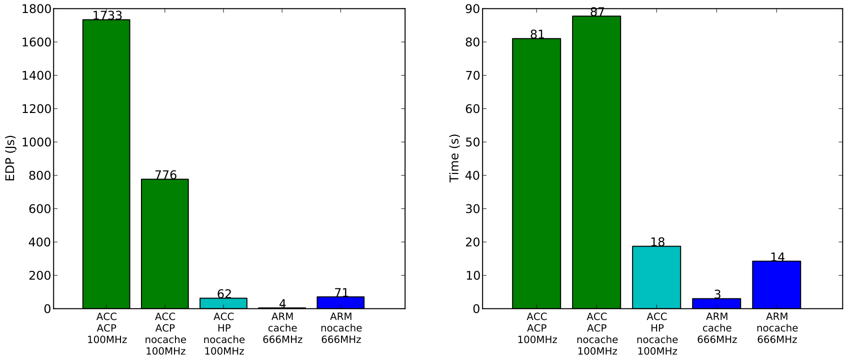
(a) Energy-Delay Product

(b) Latency

Figure 5.2: Results for CPU with data in BRAM or DDR

are chosen. These charts clearly show the supremacy of the L2 cache setup. With CPU at 666 MHz it achieves a 34x better energy-efficiency and 7.6x better performance compared to accessing the DDR directly and avoiding the cache. The reason for such a great improvement is the reduced latency of the L2 cache and the small size of the input set, that was scaled down to 281 kB from 405 MB to fit into BRAM. The LBM algorithm makes 30000 iterations of grid updates, which means after just 1 iteration the complete input resides in CPU's 512 kB L2 cache. To compete with such performance, the ACC should be able to access the CPU's memory hierarchy.

Analysing the EDP of the BRAM and DDR cache-less configurations (Figure 5.2a), it is of no surprise that the BRAM performs worse. The reason is that the DDR cannot be removed from the system, so when running on the BRAM, the unused DDR still adds to the power consumption. Even with such huge drawback,



(a) Energy Delay Product (EDP)

(b) Latency

Figure 5.3: Results for ACC using the ACP

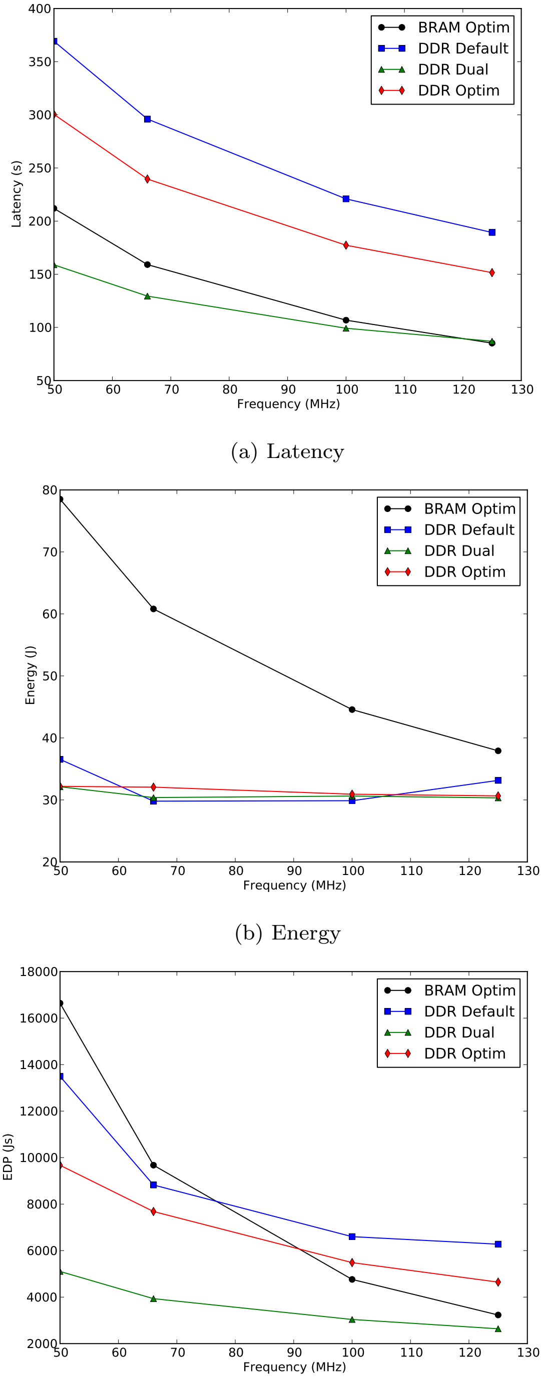
the BRAM scores almost the same EDP as DDR with CPU running at 666 MHz. This is due to BRAM's 34% better performance (Figure 5.2b). This fact is quite surprising, since the DDR has two ports and runs at 533 MHz and BRAM has only one and runs at 200 MHz. Smaller latencies for all CPU frequencies suggest that the BRAM implementation would also be more energy-efficient, if the DDR could be completely switched off. Switching to dual-port BRAM could potentially improve the performance and energy-efficiency even further.

After analysing the performance and energy-efficiency of various configurations of the initial system, I found that the BRAM performs better than the DDR, but none of the two is even close to the speed and energy-efficiency of the system using the L2 cache. To account for this difference, in the next sections, the ACC will be compared to the CPU while keeping the same memory system.

## 5.3 LBM Running on ACC

This section explores performance and energy-efficiency of different memory systems that can be used with the ACC, together with a comparison of the three types of the ACC: Default, Optim and Dual.

From the results of Section 5.2, it is obvious that using the L2 cache greatly improves application's performance and energy-efficiency. To access the CPU's memory hierarchy from PL, the ACP is the only solution, which enables coherent access to CPU's caches. I thought the accelerator connected to the ACP will behave like ARM's third core, and enjoy the benefits of a cached architecture. In practice, however, the accelerator achieves no performance gain, when using the ACP, actually getting much worse results compared to DDR cache-less access through the HP (see Figure 5.3). The reason could be that the ACP only ensures the coherency, but does not actually use the memory hierarchy. In other words, when looking for data, it also checks the caches, invalidates the memory locations when appropriate, but does not bring the data to cache after a miss. This may



(a) Latency

(b) Energy

(c) Energy Delay Product

Figure 5.4: Results for different ACC implementations

explain that using the ACP in case of LBM is actually an unnecessary overhead, and that is it much faster to ensure the coherency in software. Only two operations need to be done, flush the cache after the CPU prepared the data and invalidate it when the ACC has done its work.

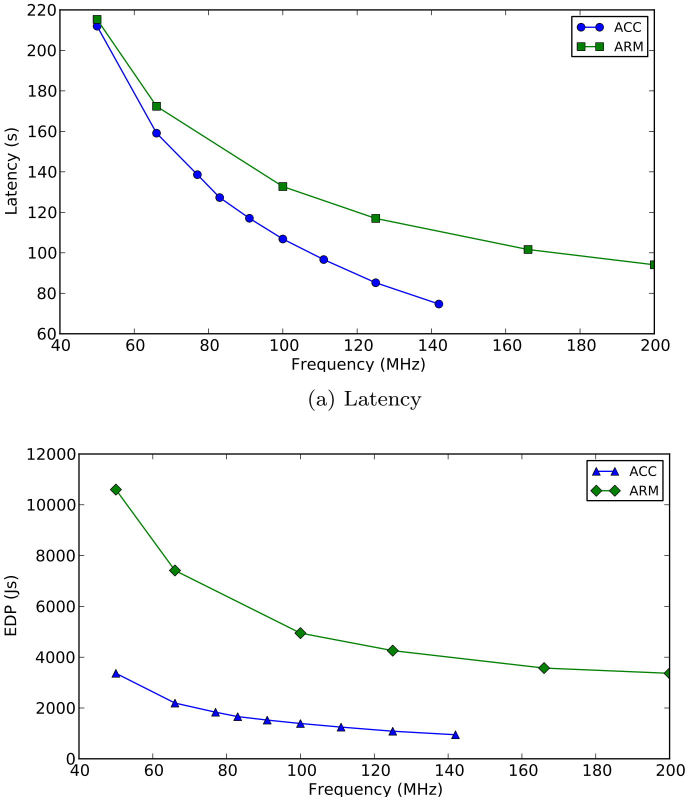
Figure 5.4 compares the metrics for the three ACC implementations accessing the DDR, and the DDR vs BRAM configurations for the Optim ACC. These results confirm the findings in the previous section, that the BRAM implementations perform better both in terms of energy-efficiency and performance. With respect to EDP, it could have been even lower, if the DDR could be shut down completely.

The energy consumption of the BRAM implementation (Figure 5.4b) is linear with respect to the frequency of the PL. However, the DDR implementations show almost constant energy consumption across ACC versions and operating frequencies. The reason is that the power consumption and performance of the BRAM is dependent on the PL frequency, while the DDR is not influenced by this factor. This explains the fact that the BRAM implementation of the Optim ACC performs worse at lower frequencies than its DDR version. However at about 90 MHz, the BRAM starts to improve energy-efficiency compared to DDR. At peak frequency the BRAM version is 1.4x more energy-efficient and 1.8x faster than the DDR version.

When the clock speeds are matched, Optim performs better than Default. It proves that pipelining is indeed an effective technique of increasing the performance and improving the EDP. However, when synthesised in VHLS, the Optim accelerator had a maximum clock frequency of 50MHz, while Default was synthesised with a target clock of 100MHz. Comparing these configurations, the Optim is actually 1.5x worse in energy-efficiency and 1.4x slower in performance. This fact shows that *optimizations which lead to a slower clock should be avoided, since they may results in a overall slower device.* It was not the case for the Dual ACC.

Out of the three ACC implementations, the Dual performs best. Separating the read and write ports, resulted in less data dependencies, making the pipeline more effective. Since the DDR has two ports for accessing the memory, the ACC could issue read and write operations simultaneously.

Of course the improvements in EDP and latency in the practical implementation do not match the estimations from VHLS presented in Table 5.1, because they represent different things. The results in Table 5.1 are based on simulations, while the results in this Section are based on physical power measurements "out of the wall", of the complete development board. These measurements are offset by the power consumption of an "idle" system as discussed in the Section 4.1.2 of the Methodology Chapter. It is not possible to get a fine-grained, per-component power consumption on Zedboard for a more precise analysis of the energy-efficiency.



(a) Latency

(b) Energy Delay Product

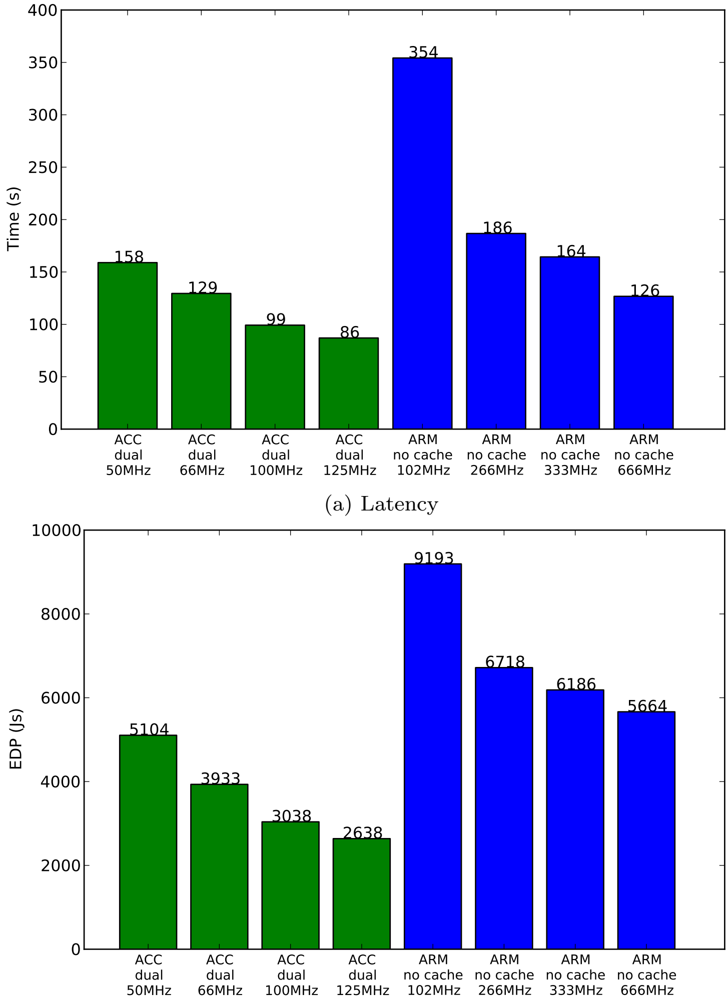
Figure 5.5: Results for LBM with data in BRAM

## 5.4 LBM with Data in BRAM

Figure 5.5 presents the results of running the application on both ACC and ARM while keeping data in BRAM. These metrics were collected in the first, BRAM-based implementation of the system (see Section 3.1). It uses the OPTIM version of the ACC, which was synthesised with a target clock frequency of 50MHz, but produced correct results up to 142MHz. The CPU frequency is kept constant at its maximum value of 666MHz.

A limitation of the design is that the ACC, the AXI interconnect and the BRAM controllers have all the same clock source. This prevents running the interconnect and BRAM at the maximum speed of 200MHz. Even with such constraint, the ACC is more energy-efficient than the CPU in all configurations of the two.

Comparing the ACC at 142MHz and the CPU at 666MHz with the interconnect and BRAM at 200MHz, the former achieves a 3.5x increase in energy-efficiency while improving the latency by 21%.



(a) Latency

(b) EDP

Figure 5.6: Results for LBM with data in DDR

## 5.5 LBM with Data in DDR

Section 5.3 confirmed the results of HLS estimations, that the DUAL implementation of the ACC is indeed the fastest and the most energy-efficient. Experiments in Section 5.2 show that down-clocking the CPU does not improve the EDP in case of LBM, and that a custom accelerator connected directly to DDR can hardly compete with CPU using the complete memory hierarchy. Based on these findings, this section compares the EDP and latency results for CPU without the L2 cache and the DUAL ACC connected to DDR through the HP port. The L1 caches are still enabled and provide the CPU with shorter memory-access delays.

The Dual ACC, just like the CPU, benefits from the dual-port DDR architecture. It allows two memory operations to be issued simultaneously. Figure 5.6 shows the results for latency and EDP for various clock frequencies of the ACC and CPU. When both units run at their peak performance, the ACC proves to be 1.5x faster and 2.15x more energy-efficient. If clock speeds are matched at 100 MHz, the ACC is 3.6x faster and 3x more energy-efficient.

# Chapter 6

# Conclusion and Future Work

## 6.1 Conclusion

This thesis evaluated how energy-efficiency of a system can be improved using special-purpose accelerators. A literature survey on energy-efficient accelerators was conducted. In contrast to the related work [18, 22, 23], that used simulators to evaluate the energy savings, the results in this thesis were based on a physical implementation of a hardware accelerator using the Xilinx Zynq SoC platform. The accelerator was developed with the focus on energy-efficiency.

This section summarizes this work's findings against the research questions that were formulated in Section 1.2.

**1. How can Xilinx tools assist in developing an energy-efficient hardware accelerator?**

This work defined the complete design work flow for developing an accelerator for an application of choice. The accelerator was implemented on Xilinx Zynq using the Xilinx ISE System Edition with Vivado HLS.

Xilinx Vivado HLS was used to develop the hardware accelerator. It eases the development process by working on a higher abstraction level – the behavioural specifications. The tool provides quality of results that rivals hand-coded RTL, while decreasing the design time by a factor of 10 [30].

Vivado HLS enables early prototyping and evaluation of the design. The post-synthesis reports estimate the utilization, power, throughput and latency of the synthesised design. These metrics allow to evaluate the impact of various optimization techniques on the energy-efficiency of the accelerator.

Vivado HLS accepts as input a synthesisable subset of C/C++/SystemC programs. If the algorithm is too complex and cannot be completely synthesised, it can be broken down into simpler modules and the remaining non-synthesisable logic can be implemented manually in RTL or left in software and executed by the CPU.

The main drawbacks of Vivado HLS are poor documentation and the difficulty to debug. The tool may fail with a generic error message, that does not give any clue what was the reason of the failure. The solution is either asking the community (which is very limited) or trial-and-error. The lack of a detailed documentation resulted in additional difficulties when implementing the LBM accelerator. The reference designs for Vivado HLS are quite simple and do not address the issues of connecting the accelerator to the DDR directly or using the cache hierarchy.

**2. How can different optimization and architectural choices affect the performance and energy-efficiency of the accelerator?**

This thesis explored the effects of different optimization techniques on the accelerator's performance and energy-efficiency. The following findings are based on the estimations from Vivado HLS post-synthesis reports:

- Increasing the clock frequency may indeed improve performance, but may as well result in a worse energy-efficiency.
- Pipelining always improves performance and energy-efficiency. The improvement is very much influenced by the dependencies in the algorithm.
- Having separate ports for reading and writing removes some data dependencies and boosts the effectiveness of the pipeline. This results in a 13.95x gain in performance and 73.75x gain in energy-efficiency.
- A small local memory to buffer read accesses enhances the pipeline even further reaching a 44.4x gain in performance and 379x gain in energy-efficiency.

**3. How can performance and energy-efficiency be evaluated on Zynq?**

The accelerator was implemented and evaluated on the Zedboard development board that features a Zynq-7000 SoC. Zedboard is not equipped for fine grained power measurements. It contains only one current-sense resistor to measure the power of the complete board. Zynq does not feature any energy counters that would assist in evaluating the energy-efficiency of the system. This work proposed a methodology to break down the total power consumption to estimate the energy savings the accelerator brings.

**4. How can the custom accelerator improve the energy-efficiency of the complete system?**

This thesis evaluated the energy-efficiency of the accelerated system under different architectural configurations of the system and the following conclusions were drawn:

- Optimizations which lead to a slower clock should be avoided, since they result in a overall slower device.
- Pipelining is a very good optimization technique, unless it results in a slower clock.

- With equivalent memory systems, the Accelerator achieves better performance and energy-efficiency than the CPU implementation.
  - For BRAM implementation, the ACC achieves 3.5x increase in energy-efficiency and improves latency by 21%, even if accelerator's frequency is 4.7x less than that of the CPU.
  - For accessing DDR directly, the dual-ported ACC proves to be 1.5x faster and 2.15x more energy-efficient, even if the ACC runs at a frequency 5.3x less than the CPU. If the clock frequencies are matched at 100 MHz, the accelerator is 3.6x faster and 3x more energy-efficient.
- The CPU using the L2 cache is the most energy-efficient configuration of the system. Enabling the cache results in 7.6x better performance and 34x better energy-efficiency. Such a great improvement is due to the reduced size of the input data set, that can completely fit into the L2 cache. This result illustrates that the LBM as a member of Structured Grids makes good use of the cache hierarchy.
- The ACC can access the caches through the Accelerator Coherency Port, but performs much worse than accessing the DDR directly through the High-Performance slave ports. This may be explained by the fact that the ACP only ensures coherency, without actually using the cache hierarchy. In other words, it scans the caches on read, but does not write the data in the cache after a miss.

As a general conclusion, moving to a heterogeneous multi-core system, containing special-purpose accelerators proved to be an effective solution to increase the system's performance and energy-efficiency.

## 6.2 Future Work

The results of this work clearly show that the cache hierarchy greatly improves performance and hence energy-efficiency for a member of the Structured Grids dwarf. To be able to accelerate such a system, the main step in future work is to make the accelerator use the complete cache hierarchy, as if it was the third core of the CPU.

The next step would be analysing the algorithms of different members of the Structured Grids dwarf to identify commonalities which could be implemented as hardware accelerators. These accelerators should be configurable to a degree that they cover several use-cases without increasing their complexity too much. If the accelerator becomes much bigger, it consumes more power and the gain in energy-efficiency may be lost. Instead, another specialized core can be implemented.

# Appendix

Attached to this thesis are the Vivado HLS and the PlanAhead projects for the different configurations of the system. The directory structure is presented below:

- `planAhead` – The PlanAhead project folder
  - `lbm_acp` – The Optim Accelerator accesing the data in DDR using the Accelerator Coherency Port
  - `lbm_ddr_default` – The Default Accelerator accesing the data in DDR using the High-Performance Slave Ports.
  - `lbm_ddr_optim` – The Optim Accelerator accesing the data in DDR using the High-Performance Slave Ports.
  - `lbm_dual` – The Dual Accelerator accesing the data in DDR using the High-Performance Slave Ports.
  - `lbm_bram_optim` – The Optim Accelerator accesing the data in BRAM
  - `get.sh` – The `bash` script to log the voltage measurements.
  - `measure.c` – The C program to post-process the voltage measurements log file and calculate the execution time and average power consumption.
- `vivado_hls` – The Vivado HLS project folder
  - `lbm_dual` – The Dual Accelerator
  - `lbm_optim` – The Optim Accelerator

In each PlanAhead project, of particular interest are the following files and directories:

`lbm_dual.sdk/SDK/SDK_Export/` – The location of the software source files.

`lbm_dual.sdk/SDK/SDK_Export/proc_module_hw_platform/system.bit` – The bitstream to program the FPGA.

`lbm_dual.sdk/SDK/SDK_Export/hw/ps7_init.tcl` – The Tcl file sourced in the `xmd` console to setup the system.

`lbm_dual.srcs/sources_1/edk/proc_module/pcores/` – The folder containing the accelerator IP in *pcore* format.

`xmd.tcl` – This file is sourced in the `xmd` console to program and setup the Accelerator.

`gdb.in` – This file is sourced in the `gdb` console to upload the application executable file and control the measurement scripts.

# Bibliography

[1] Ahn, I., Goundling, N., Sampson, J., Venkatesh, G., Taylor, M., and Swanson, S. Scaling the Utilization Wall: The Case for Massively Heterogeneous Multiprocessors. Tech. rep., Computer Science and Engineering, University of California, San Diego, 2009.

[2] Ahn, J. H., Dally, W. J., Khailany, B., Kapasi, U. J., and Das, A. Evaluating the Imagine Stream Architecture. In *Proceedings of the 31st annual international symposium on Computer architecture* (Washington, DC, USA, 2004), ISCA '04, IEEE Computer Society, pp. 14–.

[3] Akass, C. ARM wrestles dark silicon. `http://www.theinquirer.net/inquirer/feature/1598659/arm-wrestles-dark-silicon`, 2010.

[4] ARM. AMBA AXI and ACE Protocol Specification. `http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ihi0022d/index.html`, 2013.

[5] ARM. Cortex-A9 Technical Reference Manual. `http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.subset.cortexa.a9/index.html`, 2013.

[6] Asanovic, K., Bodik, R., Catanzaro, B. C., Gebis, J. J., Husbands, P., Keutzer, K., Patterson, D. A., Plishker, W. L., Shalf, J., Williams, S. W., and Yelick, K. A. 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, 2006.

[7] Chung, E. S., Milder, P. A., Hoe, J. C., and Mai, K. Single-Chip Heterogeneous Computing: Does the Future Include Custom Logic, FPGAs, and GPGPUs? In *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture* (Washington, DC, USA, 2010), MICRO '43, IEEE Computer Society, pp. 225–236.

[8] Colella, P. Defining Software Requirements for Scientific Computing, 2004. presentation.

[9] Digilent USB-JTAG Plugin. http://www.digilentinc.com/Products/Detail.cfm?NavPath=2,66,768&Prod=DIGILENT-PLUGIN, 2012.

[10] Dwarf Mine. http://view.eecs.berkeley.edu/wiki/Dwarfs.

[11] Ebeling, C., Cronquist, D. C., and Franklin, P. RaPiD - Reconfigurable Pipelined Datapath. In *Proceedings of the 6th International Workshop on Field-Programmable Logic, Smart Applications, New Paradigms and Compilers* (London, UK, UK, 1996), FPL '96, Springer-Verlag, pp. 126–135.

[12] EEMBC. Embedded Microprocessor Benchmark Consortium (EEMBC). http://www.eembc.org, 2012.

[13] Esmaeilzadeh, H., Blem, E., St. Amant, R., Sankaralingam, K., and Burger, D. Dark silicon and the end of multicore scaling. In *Proceedings of the 38th annual international symposium on Computer architecture* (New York, NY, USA, 2011), ISCA '11, ACM, pp. 365–376.

[14] Fiodorov, A. Towards Improving Energy-Efficiency with Special-Purpose Accelerators. Tech. rep., Norges teknisk-naturvitenskapelige universitet (NTNU), 2012.

[15] Minicom - friendly serial communication program. http://alioth.debian.org/projects/minicom, 2008.

[16] Moore, G. E. Cramming More Components onto Integrated Circuits. *Electronics 38*, 8 (Apr. 1965), 114–117.

[17] Owens, J. D., Luebke, D., Govindaraju, N., Harris, M., KrÃijger, J., Lefohn, A., and Purcell, T. J. A Survey of General-Purpose Computation on Graphics Hardware. *Computer Graphics Forum 26*, 1 (2007), 80–113.

[18] Sampson, J., Venkatesh, G., Goulding-Hotta, N., Garcia, S., Swanson, S., and Taylor, M. Efficient complex operators for irregular codes. In *High Performance Computer Architecture (HPCA), 2011 IEEE 17th International Symposium on* (2011), pp. 491–502.

[19] SCPI. Standard Commands for Programmable Instrumentation (SCPI) Consortium. http://www.ivifoundation.org/scpi/, 2013.

[20] Standard Performance Evaluation Corporation (SPEC). http://www.spec.org/, 2006.

[21] SPEC2006. Lattice Boltzmann Method (LBM). http://www.spec.org/cpu2006/Docs/470.lbm.html, 2012.

[22] Venkatesh, G., Sampson, J., Goulding, N., Garcia, S., Bryksin, V., Lugo-Martinez, J., Swanson, S., and Taylor, M. B. Conservation cores: reducing the energy of mature computations. In *Proceedings of the fifteenth*

*edition of ASPLOS on Architectural support for programming languages and operating systems* (New York, NY, USA, 2010), ASPLOS '10, ACM, pp. 205–218.

[23] Venkatesh, G., Sampson, J., Goulding-Hotta, N., Venkata, S. K., Taylor, M. B., and Swanson, S. QsCores: trading dark silicon for scalable energy efficiency with quasi-specific cores. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture* (New York, NY, USA, 2011), MICRO-44 '11, ACM, pp. 163–174.

[24] Wikipedia. Lattice Boltzmann Method (LBM). `http://en.wikipedia.org/wiki/Lattice_Boltzmann_methods`, 2012.

[25] Wu, L., Weaver, C., and Austin, T. CryptoManiac: a fast flexible architecture for secure communication. In *Proceedings of the 28th annual international symposium on Computer architecture* (New York, NY, USA, 2001), ISCA '01, ACM, pp. 110–119.

[26] Xilinx. ChipScope Pro. `http://www.xilinx.com/tools/cspro.htm`, 2012.

[27] Xilinx. LogiCORE IP AXI Interconnect (v1.06.a). `http://www.xilinx.com/support/documentation/ip_documentation/ds768_axi_interconnect.pdf`, 2012.

[28] Xilinx. Vivado Design Suite User Guide: High-Level Synthesis. `http://www.xilinx.com/support/documentation/sw_manuals/xilinx2012_2/ug902-vivado-high-level-synthesis.pdf`, 2012.

[29] Xilinx. Zynq-7000 All Programmable SoC: Concepts, Tools and Techniques (CTT). `http://www.xilinx.com/support/documentation/sw_manuals/xilinx14_3/ug873-zynq-ctt.pdf`, 2012.

[30] Xilinx. Vivado Design Suite. `http://www.xilinx.com/products/design-tools/vivado/index.htm`, 2013.

[31] Xilinx. Zynq-7000 All Programmable SoC. `http://www.xilinx.com/zynq`, 2013.

[32] Xilinx. Zynq-7000 Overview. `www.xilinx.com/support/documentation/data_sheets/ds190-Zynq-7000-Overview.pdf`, 2013.

[33] Xilinx. Zynq-7000 Technical Reference Manual. `http://www.xilinx.com/support/documentation/user_guides/ug585-Zynq-7000-TRM.pdf`, 2013.

[34] Xillinux: A Linux distribution for the Zedboard. `http://xillybus.com/xillinux/`, 2012.

[35] Zedboard. `http://www.zedboard.org/`, 2012.