TÉCNICO LISBOA

VERSAT, a Compile-Friendly Reconfigurable Processor – Architecture

João Dias Lopes

Thesis to obtain the Master of Science Degree in

Electrical and Computer Engineering

Supervisor: Prof. José João Henriques Teixeira de Sousa

Examination Committee

Chairperson: Prof. Gonçalo Nuno Gomes Tavares
Supervisor: Prof. José João Henriques Teixeira de Sousa
Member of the Committee: Prof. Pedro Filipe Zeferino Aidos Tomás

November 2017

Abstract

This thesis describes Versat, a reconfigurable hardware accelerator for embedded systems, for which this work substantially contributed. Versat is a small and low power Coarse Grained Reconfigurable Array architecture (CGRA), which implements self and partial reconfiguration by using a simple controller unit. This approach targets ultra-low energy applications such as those found in Wireless Sensor Networks (WSNs), where using a GPU or FPGA accelerator is out of the question. Compared to other CGRAs, Versat has a smaller number of functional units interconnected in a full graph topology for maximum flexibility. The lower number of functional units is compensated by the ease of configuration and runtime reconfiguration. Unlike other CGRAs, Versat can map sequences of nested program loops instead of a single program loop at a time; self and partial reconfiguration happens between the nested loops. Versat can be programmed in assembly language and using a C++ dialect. Assembly programmability is a novel and useful feature for program optimization and working around compiler or architectural issues. This work contributed to the design of the whole system, with special emphasis on verification, debug, timing closure, multi-threading at the datapath level, boot loader coding, DMA, AXI interfaces, benchmark coding and regression testing. Results on a set of benchmarks show that Versat can accelerate single configuration kernels by 4x with one order of magnitude energy consumption improvement compared to a state-of-the-art processor. For multiple configuration kernels the improvements are even better: 20x acceleration with up to 2 orders of magnitude better energy efficiency.

Keywords: Reconfigurable Computing, Coarse Grained Reconfigurable Arrays, Embedded Systems, Low Power Systems

Resumo

Esta tese descreve o Versat, um acelerador de hardware reconfigurável para sistemas embebidos. O Versat é uma arquitetura de matriz reconfigurável de grão grosso e de baixa potência, que implementa reconfiguração auto-gerada e parcial usando um controlador. Esta abordagem visa aplicações de muito baixo consumo energético, como as encontradas em redes de sensores sem fios, onde o uso de GPUs ou FPGAs está fora de questão. Comparado com outras arquitecturas, o Versat possui um baixo número de unidades funcionais interligadas em topologia de grafo completo. O número de unidades funcionais é compensado pela facilidade de configuração e reconfiguração. Ao contrário de outras arquitecturas, o Versat pode mapear sequências de malhas aninhadas em vez de uma única malha; a reconfiguração auto-gerada e parcial ocorre entre malhas aninhadas. O Versat pode ser programado em assembly ou em C++. A programação em assembly é uma característica útil para optimização de programas e contorno de erros de compilação/arquitetura. Este trabalho contribuiu para o projecto de todo o sistema, com especial ênfase nas fases de verificação, depuração, restrições temporais, concorrência ao nível de circuito de dados, acesso directo à memória, interfaces AXI, programas para caracterização e testes de regressão. Resultados de caracterização mostram que o Versat pode acelerar algoritmos de configuração única até 4x com um consumo energético uma ordem de grandeza inferior comparado com um processador de referência. No caso de algoritmos com múltiplas configurações, os benefícios são ainda maiores: aceleração de 20x com até 2 ordens de grandeza de mais eficiência energética.

Palavras-chave: Computação reconfigurável, Matrizes Reconfiguráveis de Grão Grosso, Sistemas Embebidos, Sistemas de Baixa Potência

Contents

List of Tables

List of Figures

List of Acronyms

WSN Wireless Sensor Network

GPU Graphics Processor Unit

FPGA Filed Programmable Gate Array

CGRA Coarse Grained Reconfigurable Array

CPU Central Processing Unit

RISC Reduced Instruction Set Computer

VLIW Very Large Instruction Word

IC Integrated Circuit

API Application Programming Interface

SoC System on Chip

IP Intellectual Property

FFT Fast Fourier Transform

GPP General Prurpose Processor

ASIC Application-Specific Integrated Circuit

VLSI Very-Large-Scale Integration

AXI Advanced Extensible Interface

SPI Serial Peripheral Interface

DE Data Engine

FU Functional Unit

TLP Thread-Level Parallelism

DLP Data-Level Parallelism

ILP Instruction-Level Parallelism

AGU Address Generation Unit

ALU Arithmetic and Logic Unit

CM Configuration Module

DMA Direct Memory Access

CRF Control Register File

Chapter 1

Introduction

1.1 Motivation

Cloud Computing is a fashionable but well known topic, which has been studied since at least 1971 [1]. However, it can be argued that it is better to have distributed computing, storage and networking resources than to concentrate everything in a single node [2].

Edge computing is the logic response to cloud computing; the data is processed on the edge of the network, near its source [3]. This considerably reduces the bandwidth needed to transfer the data between the nodes and the data center. Edge Computing can benefit many applications and devices. In particular it is extremely useful in Wireless Sensor Networks (WSNs) [4].

A crucial problem of deploying computational resources on WSN nodes is power consumption. WSN nodes are normally battery operated and very price sensitive. Therefore, if one can deliver the same functionality with a smaller and more power efficient device, a more competitive product will be released.

Since WSN nodes are extremely constrained in terms of energy efficiency and price, they often employ inexpensive and low performance embedded processors. To attend the computational demands of certain applications, it is common to place dedicated hardware accelerators in the chip, used by these applications. However, the fact that these hardware blocks are not programmable increases the risk of design errors and prevents future updates. Ideally a programmable accelerator should be employed such as a Graphics Processing Unit (GPU) or a Field Programmable Gate Array (FPGA). However, these accelerators are much larger than the embedded processor cannot viably be used.

A more suitable accelerator is the Coarse Grained Reconfigurable Array (CGRA), as it can be made small and energy efficient. A CGRA is a collection of programmable Functional Units (FUs) and embedded memories connected by programmable interconnects. This structure is called the Reconfigurable Array. When programmed with a stream of configuration bits, the reconfigurable array forms hardware datapaths able to execute certain tasks orders of magnitude faster than a conventional Central Processing Unit (CPU). CGRAs are used as hardware co-processors to accelerate algorithms that are time/power consuming in regular CPUs.

Normally, the reconfigurable array is used only to accelerate program loops and the non-loop code

is run on a CPU. For this reason, CGRAs normally include a conventional processor. For example, the Morphosys architecture [5] integrates a small Reduced Instruction Set Computer (RISC) and the ADRES architecture [6] integrates a Very Large Instruction Word (VLIW) processor.

Despite being an obvious solution, in practice CGRAs have not been widely adopted in the industry. The main problems that were identified with existing the CGRAs and that motivated this work are the following: they are not made small enough, they have limited reconfiguration control and are difficult to program. Therefore, we propose some architectural improvements to address these problems which follow three basic ideas.

1.2 Solutions

The first idea is to use a low number of FUs which form a fully connected graph topology [7]. This simultaneously tackles the size and programmability problems (programmability increases with connectivity). Normally, graphs with constrained connectivity are employed in CGRAs to allow spatial compactness and a high frequency of operation. However, with a low number of FUs the full mesh topology becomes affordable. Integrated Circuit (IC) implementation results indicate that only 4.04% of the core area is occupied by the full mesh interconnect. Plus, a low power device will rarely choose to operate at a very high frequency, so the moderate frequencies achieved by the IC implementation should cover a vast range of applications. The reconfiguration process is *quasi-static*, i.e., it is more frequently reconfigured compared to static arrays such as [8], but much less frequently compared to dynamic arrays such as [6]. Although large datapaths cannot be build with few FUs, it is necessary to break large datapaths into several smaller ones. On the other hand, *any* datapath that does not require more FUs than existing can be build due to the full connectivity of the nodes. Finally, programming is drastically simplified as there is no need for *place & route* algorithms in compilation flow such as in FPGAs and other fabrics. Another interesting feature is the fact that a full mesh is such an easy to apprehend structure that assembly programming is enabled, which is invaluable for optimizations and compiler or hardware bug workarounds.

The second idea is to make the configuration register addressable at the word level and therefore enable fine partial reconfiguration [7]. The CGRA configuration is divided in spaces which correspond FUs. The configuration spaces are further divided in configuration fields which are individually addressable. Partial reconfiguration is useful to reduce the reconfiguration time to a minimum, sometimes exploiting the similarity between successive configurations. Reducing reconfiguration time has a dramatic influence on improving the performance, which can compensate a lower frequency of operation.

The third idea is to integrate a small programmable controller in the CGRA to manage reconfigurations and data transfers to/from external memory [9]. The controller is in charge of the main program flow of the accelerator, sequencing the configurations and using partial reconfiguration whenever possible. The controller can spawn data stream compute threads in the CGRA and data movement threads using a Direct Memory Access (DMA) unit. While these threads are running, the controller can prepare the next configurations.

1.3 Topic Overview

In this work we present Versat, a new reconfigurable hardware accelerator which is suitable for low cost and low power devices. Its architecture uses a small number of functional units and a simple controller. A smaller array limits the size of the data expressions that can be mapped to the CGRA, forcing large expressions to be broken into smaller ones which can be executed sequentially in the CGRA. Therefore, Versat requires mechanisms for handling large numbers of configurations and frequent reconfiguration.

Versat is meant to be used as a co-processor used by means of an Application Programming Interface (API) containing a set of useful kernels. Full application developers can use a commercial embedded processor to benefit from its rich ecosystem and drop in a Versat core for performance and power optimization. Versat programmers can create a set of kernels for application programmers use. In this way, the software and programming tools of the CGRA are clearly separated from those of the application processor.

1.4 Objectives

The main objective driving the development of the Versat architecture is to obtain useful acceleration at low power consumption and small silicon area. In fact, Versat can replace a number of dedicated hardware accelerators in a System on Chip (SoC), consisting in a smaller, more power efficient and safer to design system, as a programmable solution eliminates the development risk of designing dedicated hardware accelerators.

Digital signal processing applications are targeted for biometrics, speech recognition, artificial vision, security, etc. The overall goal of the project is to create an Intellectual Property (IP) core and a library of useful procedures. A clean procedural interface to a host processor or driver is also an objective. With such an interface hosts can have tasks executed in the CGRA by simply calling procedures and passing arguments.

To illustrate the applicability of the Versat core, two popular algorithms for embedded applications have been implemented: the Fast Fourier Transform (FFT) [10] and the K-Means Clustering algorithm [11]. The FFT is a central digital signal processing algorithm, used in many applications. The importance of the K-Means Clustering algorithm has been increasing with the demand for efficient classification of large datasets of multiple dimensions and clusters.

These two algorithms have been accelerated using Graphics Processor Units (GPUs) [12, 13, 14, 15] and using Field Programmable Gate Arrays (FPGAs) [16, 17, 18, 19, 20, 21]. Although these implementations have been shown to improve performance and energy consumption over General Purpose Processors (GPPs) implementations, they can hardly be used in ultra low energy scenarios such as WSNs. A few works have demonstrated that it is more efficient in terms of time and energy to run the FFT and K-Means Clustering algorithms in a distributed fashion [22, 23, 24].

1.5 Author's Work

The work here presented is the result of a few people's work. When I started working on this project, in the summer of 2014, there was a preliminary, non-functional version of the system. I contributed with many ideas: independent Address Generation Units (AGUs), which enabled multi-threading in the Versat reconfigurable array; improvement of the operation frequency by applying pipelining techniques; the boot loader Read Only Memory (ROM) software and its handshaking protocol with a host system. I also implemented the DMA, master and slave Advanced Extensible Interface (AXI) interfaces, contributed to the Application-Specific Integrated Circuit (ASIC) implementation, I wrote all assembly kernels used for tests and implemented a regression test system where it is easy to add or remove tests. As a result of this work, the following papers have been published: [25, 7, 9, 26, 11, 10].

1.6 Thesis Outline

This thesis is composed of 6 more chapters. In the second chapter, the advantages and disadvantages of other architectures are discussed. In the third chapter, the Versat architecture is fully described. In the fourth chapter, it is described how to program Versat. In the fifth chapter, two algorithms common in real applications are fully studied. In the sixth chapter, experimental results are presented. In the seventh and final chapter, our achievements are pointed out and directions for future work are outlined.

Chapter 2

Background

CGRAs have gained increasing attention in the last two decades both in academia and industry [5, 6, 27, 28, 29]. They are programmable hardware mesh structures that operate on word-length variables and are efficient in terms of operations per unit of silicon area. CGRAs nodes are generally ALUs with high-level operations such as addition, subtraction, multiplication, shifting, etc. [30]. The CGRA is a suitable architecture for low power systems. In this chapter, the relevant CGRA literature is reviewed. The chapter is organized in the following topics: interconnect topology, reconfiguration, address generation, heterogeneity versus homogeneity and compiler.

2.1 Interconnect topology

An important aspect of CGRAs is the interconnect topology [31]. Fully connected graph topologies have been avoided as they scale poorly in terms of area, wire delays and power consumption. Since large arrays have been preferred over smaller arrays, it has been important to keep a lean interconnection structure. However, in this work a relatively smaller array is used which permits the use a fully connected topology, as the area, frequency and power results demonstrate.

2.2 Reconfiguration

A critical aspect for achieving CGRA programmability is the dynamic reconfiguration of the array. Static reconfiguration, where the array is configured once to run a complete kernel, has poor flexibility [32]. Some arrays are dynamically reconfigurable but they only iterate over a fixed sequence of configurations [5, 6, 28]. These configurations must be moved to the CGRA from an external memory and stored inside the CGRA, which costs memory bandwidth and internal storage space. Following a fixed sequence of configurations means that it is impossible to conditionally choose the next configuration. Furthermore, in many cases, it is the host system that manages the reconfiguration [5, 6, 8]. This is innefficient as the host could be doing more useful tasks. It also makes programming and integrating CGRAs in an embedded system difficult.

In order to make the reconfiguration process efficient, full reconfiguration of the array should be avoided. In this work we exploit the similarity of different CGRA configurations by using *partial reconfiguration*. If only a few configuration bits differ between two configurations, then only those bits are changed. Most CGRAs are only fully reconfigurable [5, 6, 8] and do not support partial reconfiguration. The disadvantage of performing full reconfiguration is the amount of configuration data that must be kept and/or fetched from external memory. Previous CGRA architectures with support for partial reconfiguration include RaPiD [33], PACT [27] and RPU [34]. RaPiD supports dynamic (cycle by cycle) partial reconfiguration for a subset of the configuration bitstream, which implies that the loop body may take several cycles to execute. The partial reconfiguration process in PACT is reportedly slow and users are recommended to avoid it and resort to full reconfiguration whenever possible. In RPU, a kind of partial reconfiguration called Hierarchical Configuration Context is proposed to mitigate these problems. Data to make performance comparisons with these approaches have not been found but, compared to [33], the partial reconfiguration in this work happens between program loops instead of every clock cycle, and the whole loop body executes in only one cycle. Compared to [27], this partial reconfiguration is fast and used more frequently.

Moreover, in this approach the configurations are *sef-generated*. This way, the host does not have to manage the reconfiguration process and is free to perform more useful tasks.

2.3 Address Generation

Address generation is a well known topic. CPU architectures use Address Generation Units (AGUs) to compute memory addresses, in order to improve performance [35, 36, 37]. This computation is associated with load/store instructions.

The main contribution in [29] was the invention of an address generation scheme able to support groups of nested loops expressed in a single CGRA configuration. The idea, was to reduce the reconfiguration time and was inspired by the use of cascaded counters for address generation [38]. This represented a major improvement compared to other works that focus exclusively in accelerating the inner loops of compute kernels [39]. However, as this thesis shows, more can be done in terms of address generation for reducing the reconfiguration overhead.

2.4 Heterogeneity versus Homogeneity

The question of CGRA heterogeneity versus homogeneity in terms of the functional units is an important one. Some CGRAs are homogeneous [33], i.e., they only have one type of functional unit, whereas others are heterogeneous and support a diversity of functional units [40]. A careful analysis done in [41] has favored heterogeneous CGRAs, arguing that the performance degradation when going from a homogeneous to a heterogeneous architecture is greatly compensated by the better silicon area utilization and power efficiency of heterogeneous solutions. Hence, heterogeneous CGRAs have been adopted in this project.

2.5 Compiler

Compiler support is probably the most difficult aspect of CGRAs. CGRA compilers make use of standard compilation techniques, especially the well-known modulo scheduling approach used in VLIW machines [42], and of more exotic techniques like the place and route algorithms used in FPGA compilation [43].

One attempt to circumvent the compiler difficulties is to formulate CGRAs as vector processors [44, 45, 46]. In those approaches, instructions are the equivalent of small configurations, and their authors claim several orders of magnitude speedup in certain applications. However, the user has to work at a very low level to make use of vector instructions.

A new compiler for Versat has been developed [47]. Before, the use of standard compilers such as *gcc* or *llvm* for Versat has been investigated. However, it has been concluded that these compilers are good at producing sequences of instructions but not sequences of hardware datapaths. For this reason, it has been decided that a specific compiler was needed. The developed compiler is simple as it focuses on the tasks done in Versat's reconfigurable array while performing little or no optimizations on the sequence of instructions run by Versat's controller. There was no need to use complex *place & route* algorithms thanks to Versat's full mesh topology. The syntax of the Versat programming language is a subset of the C/C++ language with a semantics that enables the description of hardware datapaths. The compiler is not described in this thesis whose main thrust is the description of the architecture and its Very-Large-Scale Integration (VLSI) implementation.

Chapter 3

Architecture

Versat is designed for fixed-point signal processing and its architecture is shown in figure 3.1. It can be used by host processors in the same chip, as some procedures can run faster and at lower energy on Versat.

In order to reduce the dependency on the host CPU, Versat features a simple Controller used for algorithmic control, self-reconfiguration and data movement. This frees the host for more useful tasks. The controller is programmable and has an instruction memory where Versat programs (aka kernels) are stored. Control over the different modules is exerted using a Control Bus. Though not shown, there is also a serial divider accessible from the Control Bus.

Versat has a special unit that performs data computation, the Data Engine (DE). This unit is composed of Functional Units (FUs) interconnected by a full mesh. Hence, there is more than one way to make a given datapath, which is useful for simplifying the compiler.

The Configuration Module (CM) holds the configuration of the DE, i.e., it specifies the current datapath, and has another register where the next configuration of the DE can be prepared; it can also temporarily store tens of other complete configurations, which can be switched at runtime. The Controller can write partial configurations to the CM, and can also save and restore entire configurations from the configuration memory.

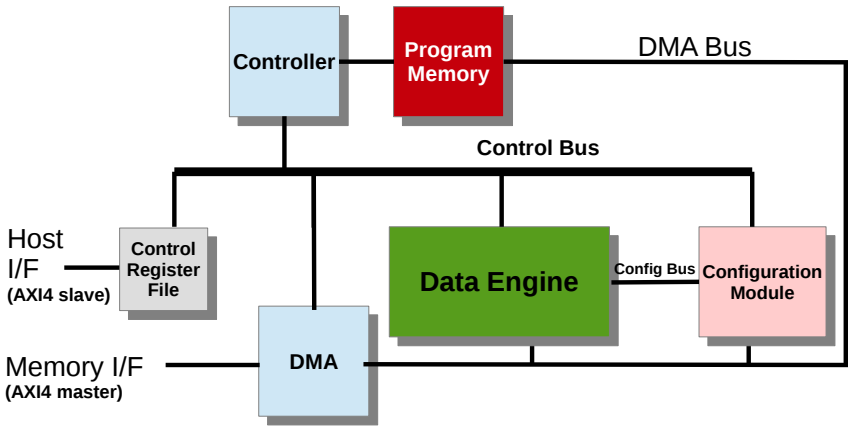

Figure 3.1: Versat top-level entity.

Versat has a DMA module so it can independently and efficiently transfer data, programs and configurations in and out of the device. The DMA drives a master Advanced Extensible Interface – AXI4. It is an interface designed by ARM, which derives from the Advanced Microcontroller Bus Architecture (AMBA).

To communicate with the host processor, Versat has a shared Control Register File (CRF). The CRF has two host interfaces that can be selected at compile time: a Serial Peripheral Interface (SPI) and a parallel bus interface. The SPI slave interface is used when the host system is an off-chip master device. The SPI interface is mainly used for debug and testing purposes. The parallel bus interface is used when the host is some embedded processor and may be supplied in the AXI4 Lite format.

3.1 Data Engine

The Data Engine (DE) has a fixed topology comprising 15 Functional Units (FUs) as shown in figure 3.2. The DE is a 32-bit architecture and contains the following configurable FUs: 4 dual-port embedded memories ($8kB$ each), 6 Arithmetic and Logic Units (ALUs), 4 multipliers and 1 barrel shifter. The output register of the FUs and the embedded memories are accessible by the Controller for reading and writing. (The embedded memory blocks are treated like any other FU by the Versat tools.)

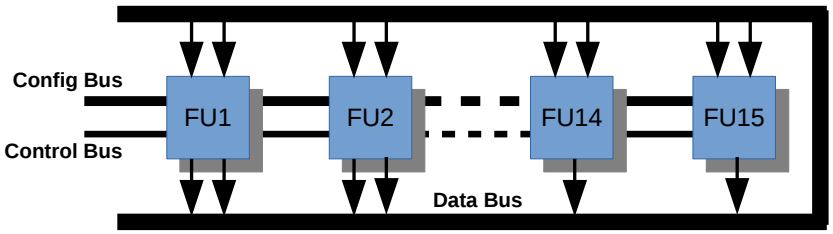

Figure 3.2: Data engine.

3.1.1 DE Structure

In the DE, the FUs are interconnected by a wide bus called the Data Bus. The Data Bus is simply the concatenation of all FU outputs. Each FU output contributes a 32-bit section to the Data Bus, except the embedded memories that contribute 2 sections with their 2 ports. According to the number of FUs of each type given before, the Data Bus has $2 \times 4 + 6 + 4 + 1 = 19$ sections of 32 bits. In addition to FU outputs, the Data Bus has 2 fixed sections containing the constants 0 and 1, which have been added because they are commonly used in datapaths. Each FU input can select any of these sections and there is also a selection that ignores the input value so that FU can be used as a Controller shared register. The FUs take their configurations from the respective configuration registers in the CM, whose outputs are concatenated in another wide bus denoted the Config Bus. There is a fixed set of high-level operations available in each FU. For example, an ALU can be configured to perform addition, subtraction, several logical functions, maximum, minimum, among others.

In figure 3.3, it is shown in detail how a particular FU is connected to the control, data and configuration buses. The FU, of type ALU, is labeled FU5 and has 2 pipeline stages. The last pipeline stage, denoted `pipeline register 1`, stores the output of the ALU, drives one section of the Data Bus and can be read or written by the Controller using the Control Bus (as shown in the figure). This feature enables the FUs to be used as Controller/DE shared registers. Each FU5 input has a programmable multiplexer to select one of the 19 sections of the Data Bus. Although the Config Bus is shown going to all FUs, in fact only the configuration bits of each FU are routed to it. These bits are called the *configuration space* of the FU. The configuration space is further divided in several *configuration fields*, which are 3 in this example: the selection of input A (5 bits), the selection of input B (5 bits) and the selection of the function (4 bits). The partial reconfiguration scheme works at the field level, so it is only possible to reconfigure these fields individually.

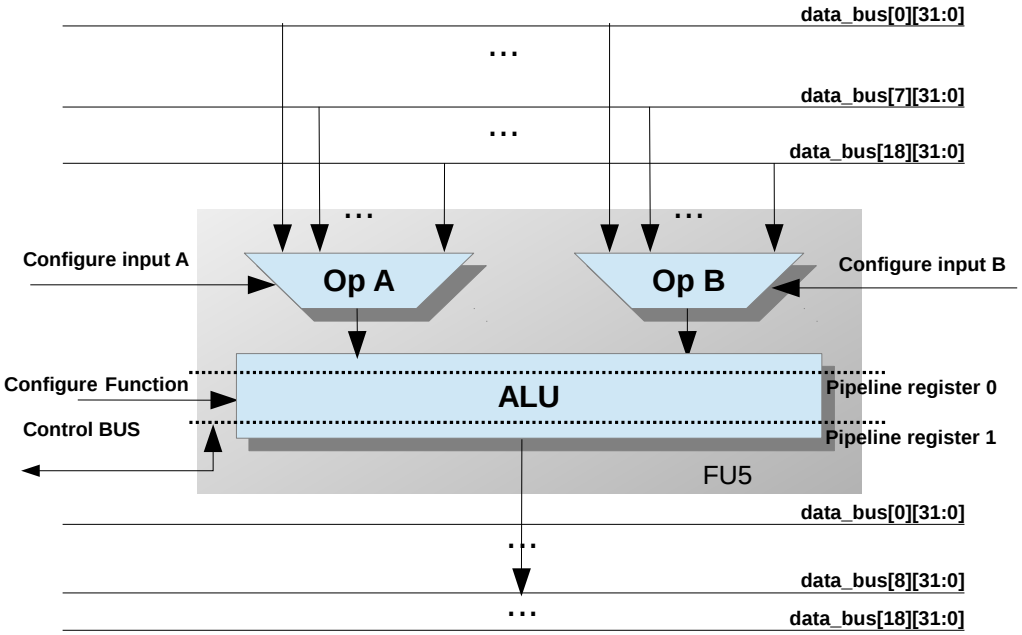

Figure 3.3: Functional unit detail.

Since any FU can select any FU output as one of its inputs, then the DE has a *full mesh topology*. This may seem exaggerated and unnecessary but this topology accomplishes 2 major goals: (1) *an intuitive assembly programmer's model is achieved*; (2) *the compiler design is greatly simplified*. In fact, assembly programmers need not remember or check what is connected to what since everything is connected to everything. Hardware datapaths can be manually built using store instructions that write to the configuration fields of the used FUs. A compiler can be developed with less effort as complex place and route algorithms, commonly used in CGRAs, become unnecessary with a fully connected topology.

Assembly programmability is a powerful feature. It may be used to optimize critical program sections or to work around bugs. In fact, hardware bugs, defects or failures may eventually be circumvented at post-silicon time using assembly code to avoid using the troubled parts of the architecture. Compiler problems can also be fixed by replacing the failing high-level code with assembly code.

In other approaches, such as in [6], a full mesh topology would be problematic because of the cycle

by cycle reconfiguration. It causes frequent switching of the interconnect network and consequent power dissipation. However, in Versat, reconfiguration does not happen every clock cycle. The interconnect consumes very little power since Versat is reconfigured only after a complete program loop or two levels of nested loops are executed in the DE. It may also be argued that a full mesh topology is large and limits the frequency of operation. However, our IC implementation results indicate that only 4.04% of the core area is occupied by the full mesh interconnect, while the core can work at a maximum frequency of $170MHz$ in a $130nm$ process. This is sufficient for many target applications. For example, in the multimedia space, some applications are required to work at an even lower frequency because of power and energy constraints.

Each configuration of the DE can implement one or more hardware datapaths. Multiple datapaths operating in parallel realize Thread-Level Parallelism (TLP). Datapaths having identical parallel paths implement Data-Level Parallelism (DLP). Finally, datapaths having long FU pipelines exploit Instruction-Level Parallelism (ILP).

In figure 3.4, three example hardware datapaths are illustrated. Datapath (a) implements a pipelined vector addition. Despite the fact that a single ALU is used, ILP is being exploited as the memory reads, addition operation and memory write are being executed in parallel for consecutive elements of the vector. Datapath (b) implements a vectorized version of datapath (a) to illustrate DLP combined with the ILP. The vectors to be added spread over memories M0 and M2, so that 2 additions can be performed in parallel. ILP and DLP can be further exploited as in datapath (c), whose function is to compute the inner product of two vectors: four elements are multiplied in parallel and the results enter an adder tree with an accumulator at the root. When an ALU is used as an accumulator the unused data input is used as a control input. As seen in the figure, this input is kept with the value 0, which specifies the accumulator function. Any positive value specifies this function while a negative value specifies that the ALU simply registers the data input.

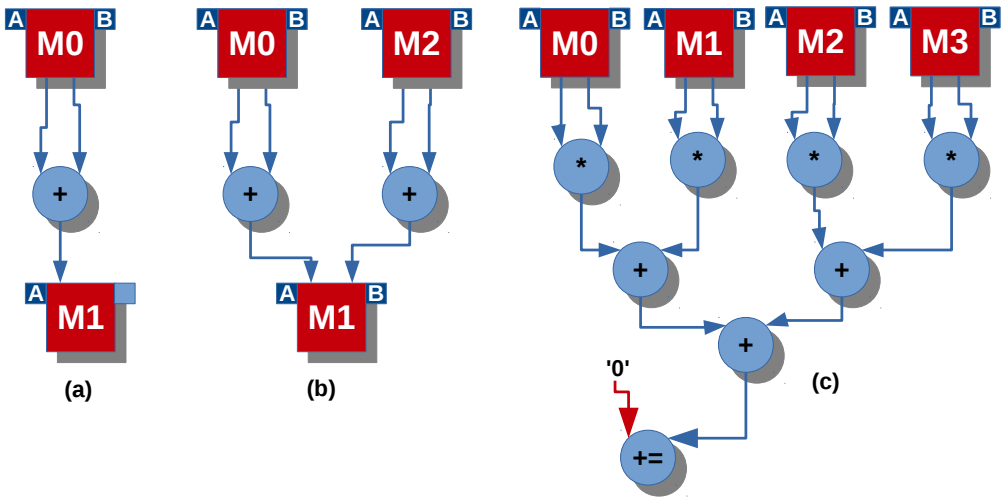

Figure 3.4: Data engine datapaths.

3.1.2 Address Generation Unit

Versat has 4 dual-port Random Access Memories (RAMs) of size 2048 words by 32 bits, which work normally as vector registers. Each port has an input, an output and an address input equipped with an Address Generation Unit (AGU), as shown in figure 3.5. The AGUs can be programmed to generate an address sequence for accessing data from the memory port during the execution of a program loop. Our AGU scheme is similar to the one described in [48], in the sense that both schemes use parallel and distributed AGUs. The AGUs support two levels of nested loops, with the restriction that the inner loop has a maximum of 32 iterations and the outer loop has a maximum of 2048 iterations. To compute longer loops reconfiguration is needed. AGUs can start execution with a programmable delay, so that circuit paths with different accumulated latencies can be synchronized. Each AGU can be operated independently from the other AGUs, which allows TLP in the DE.

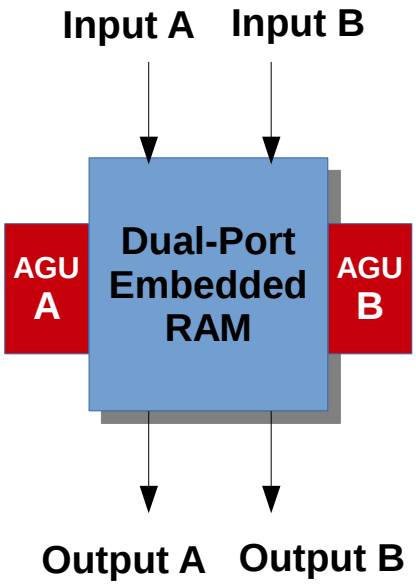

Figure 3.5: Dual-port embedded memory with AGUs.

Datapath (b) in figure 3.4, previously used to explain DLP, can also be used to explain TLP as follows. Suppose one block of vector elements to be added is placed in memory M0, and that its address generators M0-A, M0-B and M1-A are started (Thread 1). In parallel, one can move the next block to memory M2 and start AGUs M2-A, M2-B and M1-B (Thread 2). The user program can monitor the completion of these threads and restart them with new vector blocks. In this way, vectors that largely exceed the capacity of the DE memories can be processed in a continuous fashion.

The AGU parameters control the generation of address sequences and are described in Table 3.1. With these parameters the AGU can generate an address sequence for an embedded memory port. An example address sequence is shown in figure 3.6. Note that the AGU is enabled with a Delay of 2 clock cycles. It is enabled periodically for $Duty = 3$ cycles after every $Per = 5$ cycles. The initial value of the sequence is given by $Start = 10$ (in decimal notation). Every enabled cycle it is incremented by $Incr = 2$; in the last cycle of a period it is incremented by $Incr + Shift = -3$. This pattern repeats for

Table 3.1: Address generation unit parameters.

Parameter	Size (bits)	Description
Start	11	Memory start address. Default value is 0.
Per	5	Number of iterations of the inner loop, aka Period. Default is 1 (no inner loop).
Duty	5	Number of cycles in a period (Per) that the memory is enabled. Default is 1.
Incr	11	Increment for the inner loop. Default is 0.
Iter	11	Number of iterations of the outer loop. Default is 1.
Shift	11	Additional increment in the end of each period. Note that Per+Shift is the increment of the outer loop. Default is 0.
Delay	5	Number of clock cycles that the AGU must wait before starting to work. Used to compensate different latencies in the converging branches of the configured hardware datapaths. Default is 0.
Reverse	1	Bit-wise reversion of the generated address. Certain applications like the FFT work with mirrored addresses. Default is 0.

$Iter$ iterations, though this parameter is not illustrated in the figure.

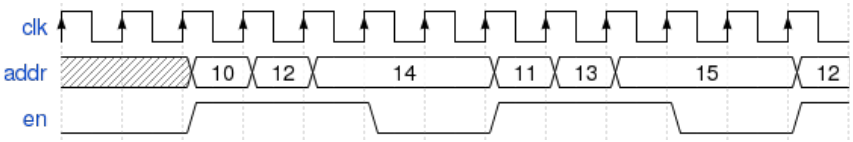

Figure 3.6: AGU output for $Delay = 2$, $Per = 5$, $Duty = 3$, $Start = 10$, $Incr = 2$, and $Shift = -5$.

The embedded memory ports have their own configuration fields: one for selecting/disabling the input or outputting the generated sequence (sel) and another for bypassing the AGU (Ext). If the input is disabled then the port is enabled for reading. If it is selecting a data source, then it is enabled for writing but also reads the word previously stored at that address. Normally an AGU is used for generating an address sequence for the memory port. However, the port can also be configured to output the generated sequence to the DE, where it can be used for general purposes. With this feature, data patterns, synchronization and control signals can be generated for the FUs. The configuration for bypassing the AGU ($Ext = 1$) enables the port to use as address values computed by datapath. In this case the port address is input by the other port of the same memory. This feature provides Versat with the capability of working with pointers.

In summary, the two memory ports can be independently configured to read and/or write the memory, each memory port can be read/written with an address sequence computed by the AGU or by some datapath in the DE, and a memory configured for writing also reads the previously stored word from the same address with one clock cycle of latency.

3.1.3 Arithmetic and Logic Unit

There are two types of Arithmetic and Logic Units (ALUs), denoted Type I and Type II: 2 Type I and 4 Type II. Both types have two inputs, A and B, and an output Y. A summary of the ALU operations is

given in Table 3.2. The ALUs have 4 configuration bits for the operation field and thus can support 16 different operations.

Table 3.2: ALU functions.

Operation	Type I	Type II (w/ feedback)
Logic OR	Y = A | B	Y = Y | B
Logic AND	Y = A & B	Y = Y & B
Logic XOR	Y = A $\oplus$ B	NA
Addition	Y = A + B	Y = (A<0)? B: Y+B
Subtraction	Y = B - A	Y = (A<0)? B: Y-B
Multiplexer	Y = (A<0)? B: 0	Y = (A<0)? B: Y
Sign extend 8	Y = A[7]...A[7..0]	NA
Sign extend 16	Y = A[15]...A[15..0]	NA
Shift right arithmetic	Y = A[31], A[31..1]	NA
Shift right logical	Y = '0', A[31..1]	NA
Signed compare	Y[31] = (A>B)	Y[31] = (Y>B)
Unsigned compare	Y[31] = (A>B)	NA
Count lead zeros	Y = CLZ(A)	NA
Signed maximum	Y=max(A,B)	Y = (A<0)? Y: max(Y,B)
Signed minimum	Y=min(A,B)	Y = (A<0)? Y: min(Y,B)
Absolute value	Y=|A|	NA

Type II ALUs use one of the configuration bits to create an internal feedback loop from the output to input A. With the other 3 bits, Type II ALUs can support 8 operations. If the feedback bit is set to '0', these operations are the same as for Type I ALUs. If the feedback bit is set to '1', the operation has input B and the current output as operands. In this case, input A becomes available for runtime control of the ALU, which is used for implementing conditional statements in operations ADD, SUB, MUX, MIN and MAX. For instance, the addition operation (ADD) can be turned into a conditional accumulate operation: the ALU accumulates the values coming to input B, if input A is not negative or simply registers input B otherwise. In another example, the signed minimum operation can be used to detect and register the minimum value among only certain elements of a sequence coming to input B, by using input A as a data qualifier. Often, an AGU is used to generate control sequences for the control input A of an ALU. The type II ALU is an original way to enable conditional execution in the DE.

3.1.4 Multiplier and Barrel Shifter

The multiplier produces a 64-bit result from two 32-bit operands and has two configuration parameters. One parameter allows selecting the lower or higher 32 bits of the result. The other parameter forces the multiply result to be left shifted by 1 bit. This configuration is useful when operands are in the Q1.31 fixed-point format, which is used in many DSP algorithms. By setting the first parameter to select the high part of the result and the second parameter to left shift it by 1, the multiplication of two Q1.31 operands also yields a Q1.31 result.

The barrel shifter can perform left and right shifts. Right shifts can be configured as logical (no

sign extension, feed '0' to the left) or arithmetic (with sign extension). One configuration parameter determines the shift direction (left or right) and another parameter sets the right shift type (arithmetic or logic). In one input of the barrel shifter is the value to be shifted and in the other input is the shift size (0 to 31).

3.1.5 Functional Unit Latencies

Each FU has a latency due to pipelining: 2 clock cycles for the ALUs, 3 for the multipliers and 1 for the barrel shifter and embedded memories. When configuring a datapath in the DE, it is necessary to take into account the latency of each branch, and compensate for any mismatches when branches with different latencies converge. To do this, the AGUs have the *Delay* parameter explained in Table 3.1. The branch latency is the sum of its FU latencies.

3.1.6 Data Engine Control

The DE is controlled using its Control and Status registers. The Control register structure is the following: bit 0 (init) resets the selected memory ports and bit 1 (run) enables the selected ports, resets the selected FUs and starts the DE; bits 2 - 20 select the memory ports and FUs to reset or enable. Recall there are 8 memory ports and 11 other FUs in a total of 19 FUs to control. The Status register structure is the following: bits 0 - 7 indicate which AGUs are running (logic '0') or idle (logic '1').

3.2 Configuration Module

The Configuration Module (CM) is composed of a Configuration Register File used to prepare the next configuration, the Configuration Shadow Register which holds the configuration being executed by the DE, and the Configuration Memory where frequently used configurations are stored (figure 3.7). Configurations stored in the configuration memory can later be used without modifications or they can be partially reconfigured before used.

The configuration register file is divided in configuration spaces which in turn are divided in configuration fields. Different configuration spaces differ on the number of fields and configuration fields differ on the number of bits. A full DE configuration is 672 bits. The configuration register file is addressable at the configuration field level and there are 118 configuration fields. This feature takes advantage of the fact that in most applications there is a high likelihood that a certain configuration or a similar one will be reused (time locality) to make partial reconfiguration effective.

The configuration shadow register holds an active configuration word for the DE, which is copied from the configuration register file whenever the Update signal is asserted by the Controller. Thus, the contents of the configuration register file can be changed while the configuration shadow register keeps the DE configured and running.

The configuration memory is a dual-port 64-position memory, where each position can store a full configuration and is, hence, 672 bits wide. Configurations can be loaded/stored from/to the configuration

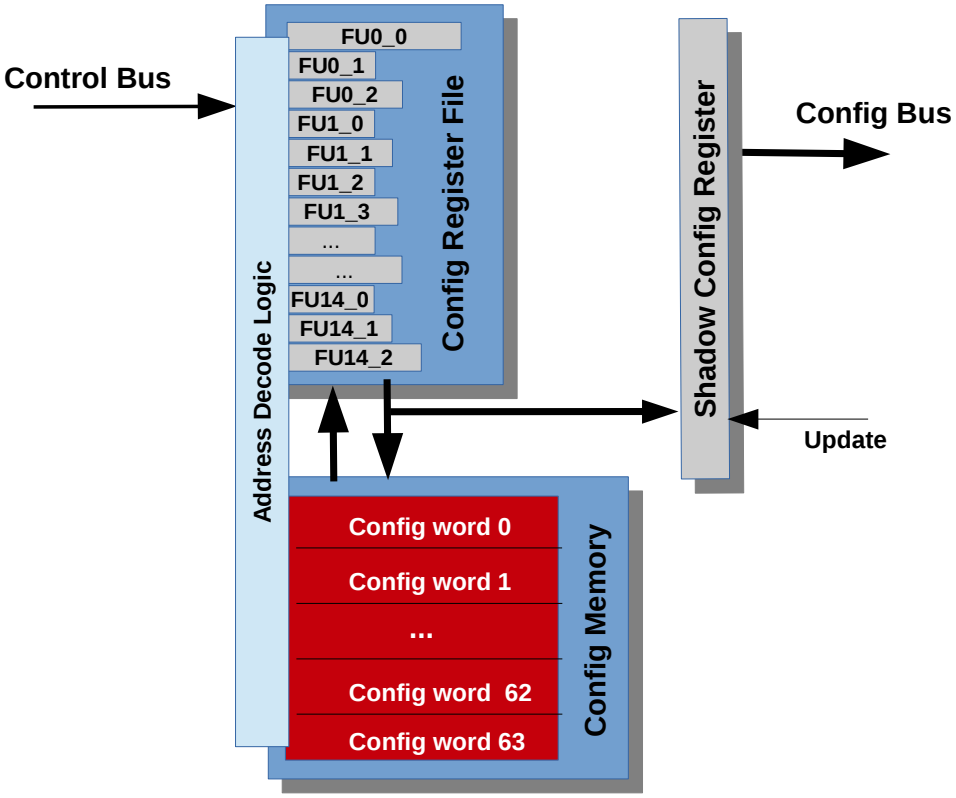

Figure 3.7: Configuration Module.

register file in just 1 clock cycle using one the memory ports. The other port is 32-bit wide and used to load and store configurations in the external memory using the DMA. This way, the configuration memory can be extended beyond the 64 configurations. This scheme is designed so that one can study the difference between working with pre-built configurations stored in external memory and generating configurations using the Versat controller.

3.3 Controller

The Versat Controller has a minimal architecture (figure 3.8) to support reconfiguration, data movement, algorithm control and host interaction. It contains 3 main registers: the Program Counter (PC), the Accumulator Register (RA) and the Address Register (RB). The PC contains the address of the next instruction as usual. Register RA is the destination of all operations that the controller performs, and is also often one of the operands (accumulator architecture). Register RB is addressable by the Controller and is used to store addresses for implementing indirect loads and stores.

The controller has an instruction set of only 16 instructions (`opcode` of 4 bits and `Imm`ediate value of 16 bits). These allow the controller to perform the following actions: loads/stores to/from the accumulator, arithmetic/logic operations and branches. There are three types of load instructions: of immediate constants, direct (from an immediate address) and indirect from an address stored in register RB. Store instructions can be direct or indirect.

The instruction set is outlined in Table 3.3. As usual, square brackets represent memory positions.

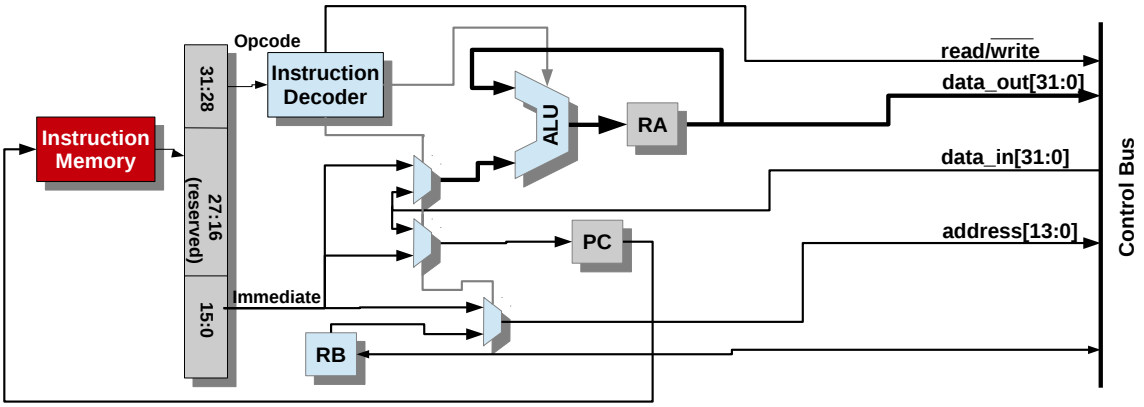

Figure 3.8: Controller.

For example, M[Imm] represents the contents of the memory position whose address is Imm. The PC is incremented for every instruction, except when indicated otherwise (branch instructions).

Table 3.3: Instruction set.

Instruction	Description
nop	No-operation
rdw	RA = M[Imm]
wrw	M[Imm] = RA
rdwb	RA = M[RB]
wrwb	M[RB] = RA
beqi	RA == 0? PC = Imm: PC += 1; RA = RA-1
beq	RA == 0? PC = M[Imm]: PC += 1; RA = RA-1
bneqi	RA != 0? PC = Imm: PC += 1; RA = RA-1
bneq	RA != 0? PC = M[Imm]: PC += 1; RA = RA-1
ldi	RA = Imm
ldih	RA[31:16] = Imm
shft	RA = (Imm $<$ 0)? RA $<<$ 1: RA $>>$ 1
add	RA = RA + M[Imm]
addi	RA = RA + Imm
sub	RA = RA - M[Imm]
and	RA = RA & M[Imm]

In order to reduce the critical path and increase the clock frequency, a pipeline register has been added between the instruction memory and the instruction decoder. The controller then takes 2 clock cycles to fetch an instruction, one from the memory itself and the other from the instruction register shown in figure 3.8. For simplicity, it executes every instruction fetched and thus branch instructions have 2 delay slots. The delay slots can be filled with useful instructions or with no operation (NOP) instructions. For instance, in a `for` loop, the delay slots can be used to increment the iteration count.

The boot loader software handles host procedure calls. The host writes the procedure parameters to the CRF and writes the program address to R0 which triggers a jump to the program address. During the execution of a typical program, the DMA and the DE are used multiple times. DMA and DE threads are spawned, hiding part of the controller execution time, as shown later. This architecture allows programs

to execute precise time delays using a `for` loop with a variable number of instructions in its body. These instructions may be useful ones (reconfiguration instructions, for instance) or just NOP instructions.

3.4 Divider

To perform divisions, Versat has a fixed-point serial divider, added as a controller peripheral, which takes 33 cycles to complete one division. It has shadow registers for the operands (dividend and divisor), as well as for the results (quotient and remainder). This means that it is possible to read the results and write next operands while a new division is already being computed. It has one configuration parameter that allows choosing between signed or unsigned division. This implementation was designed for low energy and small silicon area.

3.5 DMA

One of the crucial factors to guarantee acceleration is the rapidity at which data is moved in and out of Versat. Accessing data words from the external memory using load instructions is out of the question. Data must be moved in large blocks using a DMA engine to amortize the latency of the external memory device. The DMA engine is operated by the Versat controller and transfers a data burst from external memory into one of the Versat's memories (instruction, configuration, or data engine memory) or vice versa.

From a Versat program point of view, the DMA is memory mapped and the following DMA registers can be accessed: the external address register, the internal address register, the size register, the direction register, the status register and the start register. All registers, except the status register, are duplicated allowing the configuration of a new transaction while a previous one is running. The external address register holds the transfer start address in the external memory (32 bits), and the internal address register holds the transfer start address in Versat (14 bits written to a 32-bit register). The size register (8 bits) specifies the number of words to be transferred (256 words maximum, as per the AXI4 interface). The direction register indicates if the transfer is from Versat to external memory or vice-versa. After these registers have been configured, the program writes anything to the start register and the transfer begins. The contents of the status register tells the program whether the DMA is still busy, done or if an error has occurred.

3.6 Program memory

The Program Memory is divided in two parts: the boot ROM (256x32, $1kB$) and the instruction memory RAM (2048x32, $8kB$). They are addressable as a single memory: the first 256 addresses are used to address the boot ROM, while the others are used to address the instruction memory.

The boot ROM holds the code that allows Versat to communicate with the host processor. Basically, it is used for the host to call Versat kernels and to load/store values in any addressable memory position

within Versat without using the DMA. This includes loading programs issuing store instructions to the Program Memory. However, the Program Memory can not be read with load instructions. Its contents can only be executed.

3.7 Control Register File

The 16x32 Control Register File (CRF) is implemented with a dual-port register file (one port for the host and another for Versat). These registers are shared between the host processor and Versat, which are allowed to asynchronously read or write them.

3.8 Qualitative comparison with other architectures

Versat has some distinctive features which can not be found in other architectures: (1) it has a small number of FUs organized in a full mesh structure; (2) it has a fully addressable configuration register combined with a configuration memory to support partial configuration; (3) it has a dedicated controller for reconfiguration, DMA management and simple algorithm control – no general purpose RISC [5] or VLIW [6] processors are used.

CGRAs started as 1-D structures [33] but more recently 2-D square mesh FU arrays became more common [5, 6, 34]. The problem with square mesh topologies is that many FUs end up being used as routing resources, reducing the number of FUs available for computation and requiring sophisticated mapping algorithms [49]. Versat is an experiment on trading a lower FU count with a richer interconnect structure. As explained before, the silicon area occupied by the full mesh interconnect is less than 5% and the limits placed on the frequency of operation are normally irrelevant given the low energy budgets of target applications. In fact, a low energy consumption often imposes a frequency of operation which is well below Versat's maximum operating frequency.

As explained in [34], the reconfiguration time in CGRAs can easily dominate the total execution time. To counter this effect Versat takes partial reconfiguration to the extreme of using a fully addressable configuration register. This keeps the reconfiguration time to a minimum and contrasts with the more moderate hierarchical reconfiguration scheme proposed in [34].

Since it is crucial to have reconfiguration done quickly, Versat includes a small 16-instruction controller with low IO latency, practically dedicated to reconfiguration management. This controller is also used to manage data transfers and control the execution of acceleration kernels. In other architectures [5, 6, 34], more comprehensive processors are used, which can also run complex algorithms. However, combing complex application coding and accelerator control is difficult. Using its controller, Versat can completely take care of simple yet compute intensive kernels. Example kernels are FFT, DCT, Motion Estimation, and even Big Data algorithms such as K-Means Clustering. These kernels can simply be invoked by host processors, and they will run in parallel to completion, without requiring any external control.

Chapter 4

Programming

Versat can be programmed using a C/C++ subset, and the code can be compiled using the Versat compiler [47]. Versat can also be programmed in assembly language, given its easy to apprehend structure. To the best of our knowledge, Versat is the only CGRA that can be programmed in assembly. In this chapter, Versat programming is illustrated using its C/C++ dialect, which is easier for explaining the basic concepts. Also it will be explaind how to use Versat drivers from the host system point of view (where the host system is an embedded processor, for instance an ARM processor).

4.1 Basic programming

In order to outline the basics of Versat programming, a typical example program is given in this section. The program adds two memory interleaved vectors: one vector resides in the even addresses while the other vector resides in the odd addresses. The program is shown in figure 4.1 and comments are added for clarity.

The program starts in the `main()` function, as is usual in C/C++ programs, and proceeds immediately to loading Versat with data using the DMA. The DMA is configured (`dma.config()`) with the external and internal addresses, transfer size and direction. The `dma.run()` function sets the DMA running, but does not wait for its completion.

With the DMA running, the program proceeds to configure the DE. The configuration register file is cleared in order to start a new configuration (`de.clearConfig()`). Certain language constructs are interpreted as DE configurations and the compiler automatically generates instructions that write these configurations to the CM. This is the case of the ensuing `for` loop. The fact that memory ports are invoked in its body triggers this interpretation. Note that object or variable declarations are not yet supported by the compiler. Data must be referenced using the names of the memories or registers where they are stored. When variables are supported, the issue of recognizing DE configurations will need further research. For now, it can be conjectured that the presence of data arrays in a loop is sufficient for inferring a DE configuration. In the given example, the expression in the loop body configures the DE to read the two vectors from the two ports of memory 0, add their elements and place the result in

```
int main(){
        // initiate data transfer into Versat using DMA
        dma.config(1024, 256, 256, 1);
        dma.run();
        // clear configuration register and create new config
        de.clearConfig();
        for(j = 0; j < 128; j++)
                mem1A[j] = mem0A[j*2] + mem0B[1+j*2];
        // wait for data transfer to finish
        dma.wait();
        // run the data engine
        de.run();
        // configure DMA to transfer result back to memory
        dma.config(2048, 2048+128, 256, 2);
        // wait for data engine to finish
        de.wait();
        // transfer result back to memory using DMA
        dma.run();
        dma.wait();
}
```

Figure 4.1: Vector addition code.

memory 1 using its port A.

Next, the program checks whether the DMA has finished loading the data. The `dma.wait()` function blocks the Controller until the DMA is idle again. Note that the DMA has been running concurrently with the Controller, which has been configuring the DE.

Once the DE is loaded and configured, it is run by means of the `de.run()` function call. While it runs, the DMA is configured in advance to transfer the result of the vector addition to the external memory. Then the program waits for DE completion (`de.wait()`) and runs the DMA (`dma.run()`). It is necessary to wait for DMA completion (`dma.wait()`) before exiting the program, to guarantee that the result is stored in the external memory before control is passed to a host processor.

4.2 Self and partial reconfiguration

In this section, an example is presented to illustrate *self-generated partial reconfiguration* in Versat (figure 4.2). The example shows a `do-while` loop. These kind of loops are always executed by the Controller, because the DE has no means to stop conditionally.

A 2-level nested loop follows, where the body contains only expressions involving memory ports. This nested loop is therefore interpreted as a DE configuration and the compiler generates instructions that write this configuration to the CM. Note that the memory address expressions (between square brackets) use register R1, which is updated inside the `do-while` loop: R1 depends on R7 which is also

```
do {
        R1 = R8 + R7;
        for(j=0; j<R6; j++) {
                for(i=0; i<R14; i++) {
                        mem0B[R1+j*R13+i] =
                            (mem1A[R1+j*R13+i] * mem2B[1025+j*R2+R10*i]) -
                            (mem0A[R1+j*R13+i] * mem2A[1024+j*R2+R10*i]);
                        mem1B[R1+j*R13+i] =
                            (mem1A[R1+j*R13+i] * mem2A[1024+j*R2+R10*i]) +
                            (mem0A[R1+j*R13+i] * mem2B[1025+j*R2+R10*i]);
                }
        }
        de.wait();
        de.run();
        R7 = R7 + R6 + R6;
        R12 = R12 - 1;
} while(R12 != 0);
```

Figure 4.2: Self and partial reconfiguration code.

updated in the loop.

In fact, R1 is the start address for the data in memories 0 and 1, which corresponds to the Start parameter of the AGUs in the memory ports used (see Table 3.1). Thus, only the Start parameters in these AGUs need to be reconfigured, which means the DE is *partially reconfigured* in each iteration of the `do-while` loop.

Furthermore, the `do-while` loop is *generating* one DE configuration per iteration. Since this is done by the Controller, without any intervention of the host processor, it may be said that *Versat is self-reconfigurable*.

The `de.wait()` function call, after the nested loop, waits for the previous DE configuration to finish running. The following `de.run()` function call runs the DE again for the current configuration. While the DE is running, registers R7 and R12 are updated and the `do-while` loop goes on to the next iteration, provided the loop control register (R12) is non-zero. In the next iteration R1 is updated, the DE is partially reconfigured for the next run, and the process repeats all over again.

4.3 Versat Driver

Versat is designed to work as a co-processor for host systems. The host system needs a very simple driver, which consists of only a few functions for managing Versat. A program that uses the Versat driver as is shown in the figure 4.3.

The reduced number of functions present in the driver make Versat very easy to use by host pro-

```
#include “versat.h”
…
int main(void){
        // load Versat program
        versat_load(progExt_ptr, progInt_ptr, progSize);
        …
        // waits until Versat is ready
        versat_wait();
        // run the FFT kernel
        versat_run(FFT, x_ptr, w_ptr, X_ptr, Npts, Overlap, WindowSize);
        …
        // waits for the FFT execution to finish
        versat_wait();
        // process the FFT results
        …
        return 0;
}
```

Figure 4.3: C code using Versat drivers.

cessors. This is mainly because Versat is very independent, it can reconfigure itself, access data from the external memory and run simple control algorithms. However, Versat needs to be programmed separately.

To use the driver, first of all, the header file (`versat.h`) must be included in the source file of the host program willing to use Versat as an accelerator.

Second, the `versat_load()` function is called to load Versat kernel(s) into Versat's program memory. Parameter `progExt_addr` is the external memory address where the program resides and parameter `progInt_addr` is the internal (to Versat) memory address where it should be transferred; parameter `progSize` is the kernel size in 32-bit words.

Next, the `versat_wait()` function is called for waiting until Versat has finished the last command. In this case, it guarantees that the kernel code is already in the program memory, so that Versat is ready to execute the next command.

Then, the `versat_run()` function is called to start execution. A FFT kernel is used as the example. The first parameter, `FFT`, identifies the kernel to be run. It is actually the address of the kernel in Versat's program memory. Parameters `x_ptr`, `w_ptr` and `X_ptr` indicate the location of the input and output vectors in the external memory. The FFT is computed over a complex dataset of `Npts` points, using windows of size `WindowSize`, where two consecutive windows overlap by `Overlap` points.

While Versat is running, some host code could be put before calling the `versat_wait()` function again, as this code will execute in parallel with Versat if it does to take more time than the Versat kernel. When the `versat_wait()` function exits the FFT computation is done and one may add some post-processing code before exiting the host program.

4.4 Comparison with other architectures

Versat is the only known CGRA that can be programmed in assembly language. It supports configurations pre-compiled, i.e., is capable of using configurations that are stored in the external memory, by transferring them into its own configuration memory like other architectures that can only be full reconfigured by transferring configurations [5, 6, 8]. As it is able to perform *self and partial reconfiguration*, i.e., generate configurations on the fly and use highly sophisticated reconfiguration mechanisms to partially modify them, since they use to be similar to each other. Unlike some other architectures that use inefficient reconfiguration mechanisms and some of them are very recommended not be used [33, 27, 34].

In conclusion, Versat is a very capable architecture of perform *self and partial reconfiguration* on the fly. It works as a co-processor used to accelerate data computation, attached to a host system who calls it by using some procedures stored in the Versat's library (`versat.h`). It is quite independent from the host system for generating configurations and transfer the data to/from the external memory, freeing the host for other useful computations.

Chapter 5

Applications

The Versat architecture is suitable to be used in ultra low energy applications such as those found in Wireless Sensor Networks (WSNs), where using a GPU or FPGA accelerator is out of the question. A dedicated hardware accelerator may also be used but its lack of programmability can become a serious liability in the long run. In this chapter, two algorithms that can be used in such applications are presented: the Fast Fourier Transform and the K-Means Clustering algorithms. Their implementation on Versat demonstrates the versatility of the architecture.

5.1 Fast Fourier Transform

The Fast Fourier Transform (FFT) is an algorithm for efficiently computing the Discrete Fourier Transform (DFT) of a digitized signal. It is used in many signal processing applications such as filters, audio and video pre and post processing, as well as encoders and decoders (spectral analysis in general). There are many implementations of the FFT algorithm. In this work, the Radix-2 FFT algorithm has been chosen, due to its simplicity. In the next subsection, a formal description of the DFT is presented, and in the subsection after, the Cooley-Tukey Radix-2 FFT algorithm is described.

5.1.1 Discrete Fourier Transform

The Discrete Fourier Transform $X(k)$ of a discrete signal $x(n)$ having N samples is given in the following equation

$$X(k) = \sum_{n=0}^{N-1} W_N^{kn} x(n), \;\; 0 \leq k \leq N-1, \tag{5.1}$$

where the coefficients W_N^k are given by

$$W_N^k = e^{-j\frac{2\pi k}{N}}. \tag{5.2}$$

The most obvious way to compute $X(k)$ is to perform N complex multiplications ($4N$ real multiplications plus $2N$ real additions) and $N-1$ complex additions ($2N-2$ real additions). Because one needs to compute N points of the DFT, the algorithm complexity is $O(N^2)$. Attending to the coefficients

symmetry (equation 5.3) and periodicity (equation 5.4), it is possible achieve a better algorithm, and the Cooley-Tukey FFT Algorithm, explained in the next subsection, is just that.

$$W_N^{k+\frac{N}{2}} = -W_N^k \tag{5.3}$$

$$W_N^{k+N} = W_N^k \tag{5.4}$$

5.1.2 Cooley-Tukey Algorithm

The DFT computation for N points can be decomposed in two independent DFT computations of $N/2$ points, for N even, separating the points which have even indices from the ones that have odd indices. Equation (5.1) can thus be rewritten as

$$X(k) = \sum_{n=0}^{(N/2)-1} W_N^{k(2n)} x(2n) + \sum_{n=0}^{(N/2)-1} W_N^{k(2n+1)} x(2n+1). \tag{5.5}$$

From equation (5.2) it can be derived that $W_N^{2k} = W_{N/2}^k$ and equation (5.5) can be rewritten as

$$X(k) = \sum_{n=0}^{(N/2)-1} W_{N/2}^{kn} f_e(n) + W_N^k \sum_{n=0}^{(N/2)-1} W_{N/2}^{kn} f_o(n), \tag{5.6}$$

where $f_e(n) = x(2n)$ and $f_o(n) = x(2n+1)$. Hence, equation (5.6) can be written as

$$X(k) = F_e(k) + W_N^k F_o(k), \;\; 0 \leq k \leq N-1. \tag{5.7}$$

Since $F_e(k)$ and $F_o(k)$ are periodic with period $N/2$ and taking into account the coefficients' symmetry, according to equations (5.3) and (5.4), the previous expression can be rewritten as

$$\begin{aligned} X(k) &= F_e(k) + W_n^k F_o(k), \\ X\left(k + \frac{N}{2}\right) &= F_e(k) - W_n^k F_o(k), \;\; 0 \leq k \leq \frac{N}{2} - 1. \end{aligned} \tag{5.8}$$

Equation (5.8) proves that it is possible to decompose an N-point DFT into a weighted sum of *two* $N/2$-point DFTs. This procedure can be applied recursively until the simplest case of a 2-point DFT is reached. This happens after $log_2(N)$ recursion levels, which explains why this FFT algorithm is called *Radix-2*. In this algorithm there are $(N/2)log_2(N)$ complex multiplications and $Nlog_2(N)$ complex additions. Therefore, the Cooley-Tukey algorithm has complexity $O(Nlog(N))$.

A 2-point FFT computed in this way is shown in figure 5.1. This is called a butterfly diagram because of its shape. The input data is on the left, while the output data is on the right. All data are complex numbers. The edges represent the data flow. The edges apply the following weights to the data: W_N, -1 or 1 (by omission). Where edges merge, their corresponding values are added. For optimization, an edge having the coefficient -1 is simply subtracted from the other merging edge.

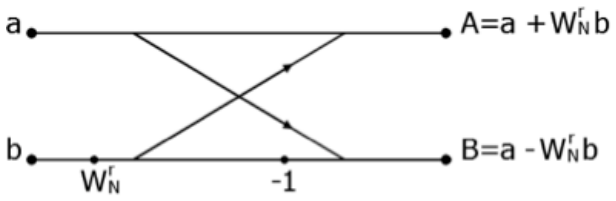

Figure 5.1: Butterfly diagram.

A graphical example of the Cooley-Tukey algorithm applied to an 8-point signal x is shown in figure 5.2. The recursion levels map to stages. In this example, there are $log_2(8) = 3$ stages. Each stage has distinctive groups of butterflies called blocks. The first stage has 4 blocks of a single butterfly, the second stage has 2 blocks of 2 butterflies and the third stage has a single block of 4 butterflies.

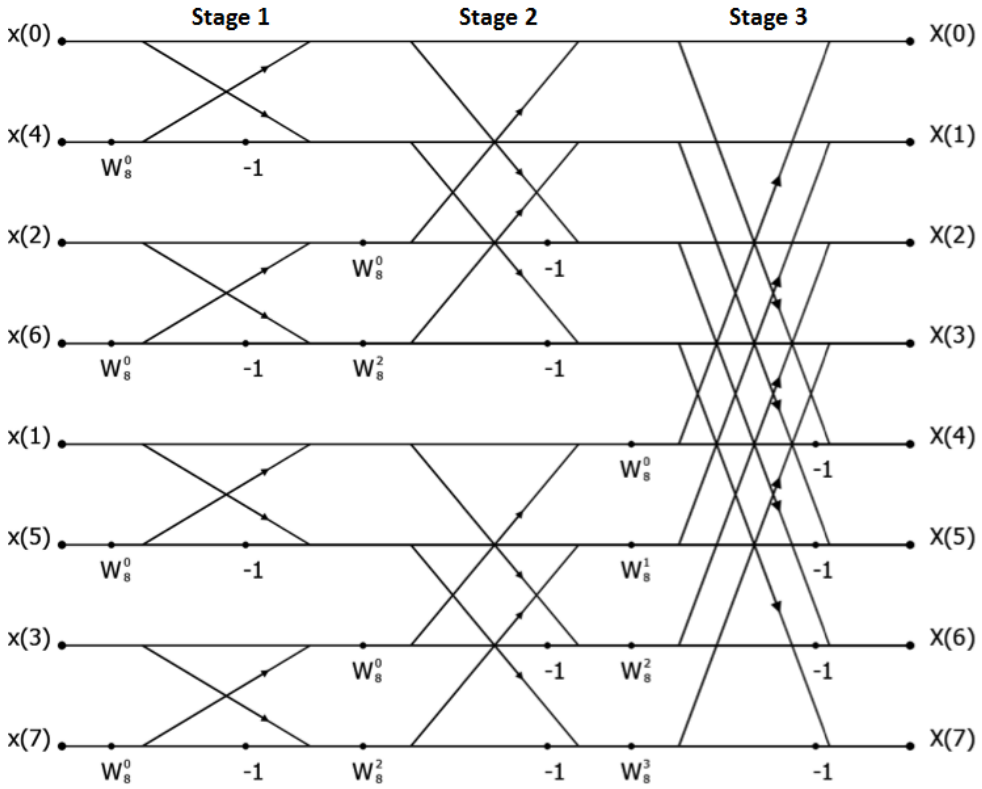

Figure 5.2: Cooley-Tukey agorithm appled to an 8-point signal x.

The Versat FFT kernel implements the Cooley-Tukey algorithm by splitting a basic butterfly structure in two steps:

1. Complex multiplication step: the datapoints stored at odd addresses are multiplied by the coefficients

2. Complex addition step: the datapoints stored at even addresses are added and subtracted from the results of the first step to yield 2 result datapoints

5.1.3 Implementation

The FFT kernel follows the algorithm given in the previous subsection, creating and partially reconfiguring two basic hardware datapaths that realize the *complex multiplication* and *complex addition* steps of the algorithm. In fact, two versions of each datapath have been created to enable ping-pong data processing. Another datapath is used to initially reorder the datapoints by reverting the bits of their addresses. The data format is 32-bit fixed-point in a Q1.31 organization. The kernel is written in the Versat assembly language and it is 873 instructions long.

The datapath to initially reorder the data is shown in figure 5.3. The AGUs of the source memories MEM2 and MEM3 (port A) read the datapoints with the Reverse parameter set to '1' (table 3.1), and the AGUs of the destination memories MEM0 and MEM1 (ports A) write them sequentially. Concurrently, the table of coefficients is copied from MEM2 to MEM0 (ports B), so that either can be accessed during the ping-pong processing.

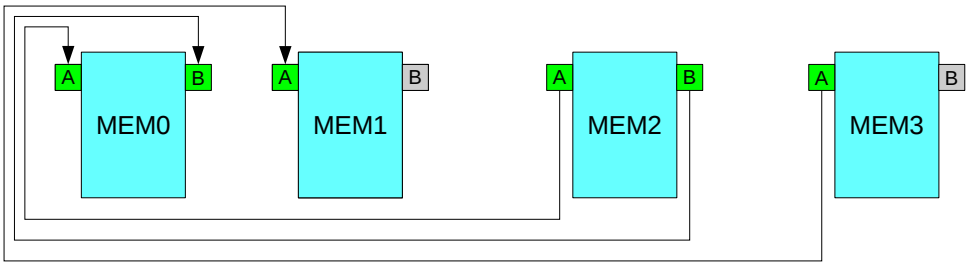

Figure 5.3: Datapath for read the initial datapoints with mirrored addresses.

The Versat controller plays an important role in complex kernels like the present FFT kernel. It is responsible for generating the datapaths, reconfiguring them and controlling the algorithm. Namely, it controls the outer loops over all stages and, in some cases, over all blocks in a stage.

First, the four datapaths are created and stored in the configuration memory. They are very similar two by two, swapping only the memories that are used for reading with the ones that are used for storing the data. This is done while data is being DMA transferred into the DE.

After reading a number of parameters from the CRF (number of datapoints, window and overlap sizes and addresses to read/write data in the external memory), as passed by the host, the program instructs the DMA to load the first datapoint chunk values in memories MEM2 and MEM3. Real parts go into MEM2 and imaginary parts into MEM3, not exceeding the 1024 lower addresses. Coefficient values are also DMA transferred into memory MEM2, occupying at most the 1024 higher addresses. Then the data is reordered and placed in memories MEM0 and MEM1 while the coefficients are copied to MEM0, using the reordering datapath explained before.

The datapath in figure 5.4 implements the complex multiplication step, reading the data from MEM0 (real part) and MEM1 (imaginary part) and coefficients from MEM2 (real and imaginary parts), and writing the result back in MEM0 and MEM1. Its configuration is loaded from the configuration memory into the configuration register file in one clock cycle, partially reconfigured, and shifted to the shadow configuration register in another cycle. The datapath forms a read-multiply-add-write pipeline with four multiplications and two additions in parallel in the respective stages, combining DLP and ILP.

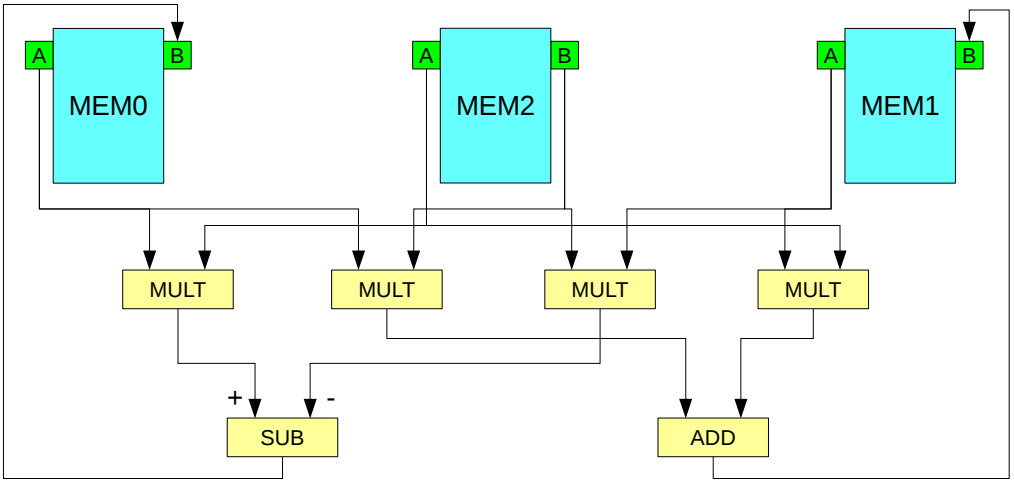

Figure 5.4: Datapath for the complex multiplication step.

Next, the datapath that performs the complex addition step, shown in figure 5.5, is loaded. This datapath performs the two complex additions in the butterfly diagram of figure 5.1. Note that MEM0 (real part) and MEM1 (imaginary part) have the original data in the even addresses and the original data multiplied by the coefficients in the odd addresses. Therefore the data in the even addresses is added and subtracted to the data in the odd addresses to produce the two butterfly outputs. The results are placed in MEM2 (real part) and MEM3 (imaginary part). The parallel additions exploit DLP while the read-compute-write pipeline exploits ILP.

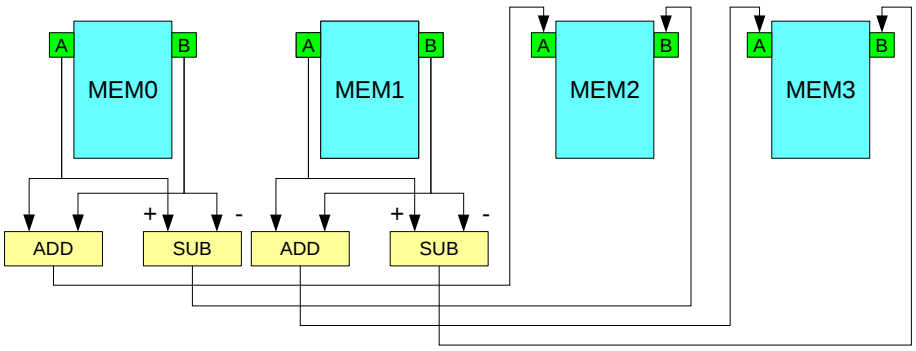

Figure 5.5: Datapath for the complex addition step.

After both steps are run, the controller increments the outer loop and the two steps are applied again, now from MEM2 and MEM3 to MEM0 and MEM1. This process is repeated until all stages are processed. Every time the kernel needs more data, the current results are stored in the external memory and new datapoints are loaded into the DE memories.

Because AGU parameter Period has only 5 bits (table 3.1), there is need to break the two step computations in several parts for some stages. This is because the two nested loops that can be executed in the DE are used to go over all blocks and all datapoints within a block. If there are more than $2^5 = 32$ butterflies within a block then the inner loop cannot be used. Only the iterations loop is used

and the DE needs to be reconfigured for each block. In order to save reconfiguration time, the controller uses partial reconfiguration. Additionally, if an FFT window cannot fit into the DE memories, the FFT computation is divided into several smaller FFTs and several other steps to merge the results.

After all stages are processed, the controller instructs the DMA to transfer the results to the external memory, using the respective address in the CRF passed by the host. Then, a new datapoint chunk is loaded and the whole process starts again, until all datapoints are processed and stored in the external memory. When this happens, the algorithm terminates. Before returning to the boot ROM memory, the controller clears register R0 in the CRF, which signals the host that Versat has finished execution.

5.2 K-Means Clustering

The K-Means algorithm is one of the simplest algorithms for performing the clustering task. Despite its simplicity, it is still one of the most widely used clustering algorithms, due to its ease of implementation and fast execution time.

5.2.1 Algorithm

The algorithm uses a centroid model. It separates the data into a set of clusters, each having a centroid represented by the mean vector of all the datapoints in the cluster. Each datapoint is classified as being in the cluster whose centroid is closest to it. The Euclidean distance is a common metric, though other types of metrics can be applied [50]. For simplicity, the Manhattan Distance (MD) is used. After an initial position is given to each centroid, the algorithm starts updating the position of the centers in an iterative fashion. Each iteration is divided in two main steps:

1. Assignment step: each datapoint is assigned to the nearest centroid, given the chosen distance metric
2. Update step: the centroids are recalculated; the new positions correspond to the mean of all the datapoints in each cluster

The algorithm ends when the centers are unchanged after an iteration.

5.2.2 Implementation

The K-Means Clustering kernel follows the algorithm given in the previous subsection, creating and partially reconfiguring the two basic hardware datapaths that realize the *assignment* and *update* steps of the algorithm. The kernel is written in the Versat assembly language and it is 691 instructions long.

As happens in the FFT kernel, the two datapaths are created and stored in the configuration memory. This is done while the first chunk of data is being transferred into the DE by the DMA. After reading the number of points, coordinates, centroids and respective external memory addresses from the CRF, as passed by the host, the program instructs the DMA to load the first datapoint chunk and initial centroid values in memories MEM0 and MEM1, respectively. The datapoint chunk size cannot exceed

2048x32bits, the size of MEM0 (starting at the lower address). The number of centroids times the number of coordinates is limited to 1024x32bits, half the size of MEM1 (starting at the second half). The other half is reserved for incrementally building the updated centroids. This is a hard limitation of this algorithm, which can only be overcome by using a larger embedded memory for the centroids, which costs silicon, or by streaming the centroids as done with the datapoints, which penalizes performance. However, in the target WSN applications, only a few centroids are used, making the current algorithm implementation acceptable.

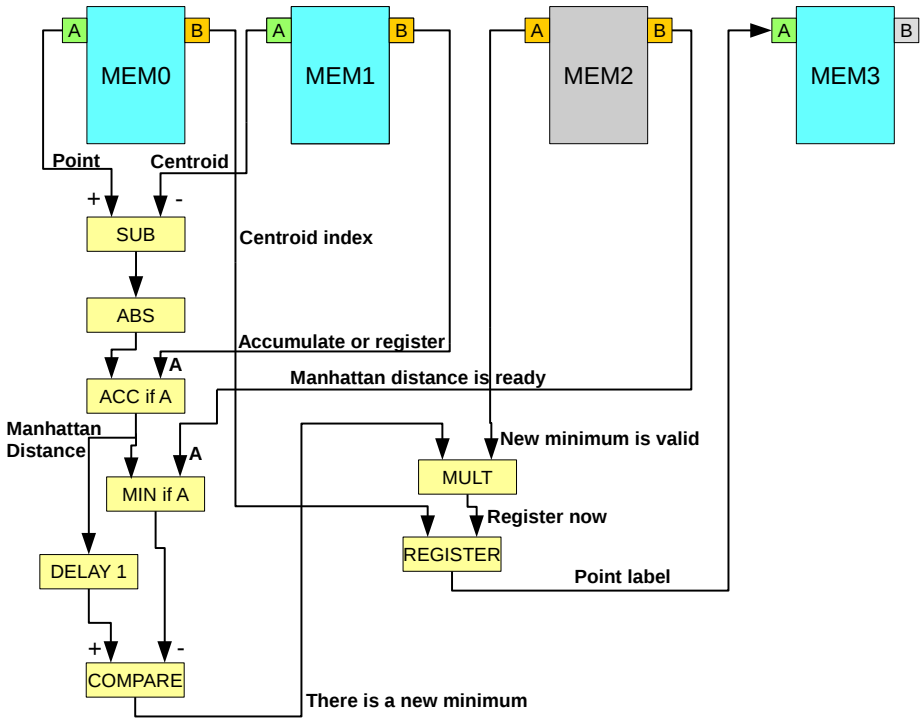

Figure 5.6: Datapath for the assignment step.

The datapath in figure 5.6 implements the assignment step. Its configuration is loaded from the configuration memory into the configuration register file and shifted to the shadow configuration register in two clock cycles, as in the FFT kernel. The DE is started and the first point is compared with all the centroids, coordinate after coordinate. The absolute value of the coordinate difference is accumulated in the type II ALU, configured with the *ACC if A* function. In this function, the A input instructs the ALU to either store the first absolute difference or accumulate the following ones. Input A is produced by the AGU from port B of MEM1, which is thus not used to generate memory addresses – this is on of Versat's novel feature as explained in section 3.1. If $A < 0$ then the ALU stores, otherwise it accumulates. After processing all coordinates, the MD is ready. The ALU configured as *MIN if A* checks whether the current centroid is the closest yet to the datapoint, by computing the minimum between the current MD and the previous minimum. Input A is generated by port B of MEM2 to synchronize this action. The current MD is delayed by 1 cycle using the barrel shifter configured as *DELAY 1* (i.e., no shift), so that it can be compared with the current minimum. If it is smaller, then it will be the next minimum, and the centroid

index, generated by port B of MEM0, is stored in the ALU configured as *REGISTER*, when signalled by port A of MEM2, which tells when is the new minimum valid. The logic AND between the existence of a new minimum and the moment when it is valid is implemented by a multiplier, since all the 6 ALUs have been used up. Note that port B of MEM0 and both ports of MEM2 need to delay their start instant considerably, waiting for all coordinates or all centroids to be processed, plus the datapath latency. This is not possible to achieve with the 5-bit *Delay* AGU parameter given in table 3.1, so it is the controller that starts these AGUs at the right time. Finally, the point label (closest centroid index) is stored in MEM3 after its port A receives a signal from the controller at the precise timing after all centroids are compared to the datapoint.

To process the next datapoint, the DE is partially reconfigured to advance the addresses of the datapoint and its label in ports A of MEM0 and MEM3, respectively. It is not possible to advance the datapoint without reconfiguration, as this would require our AGUs to support three levels of nested loops when they only support two. Their 2 levels are used to go through all centroids (*Iter* = number of centroids) and all coordinates for each centroid (*Per*=number of coordinates). Also note that we have *Shift*=-*Per* for the AGU of port A of MEM0, so that its generated address goes back to the first coordinate after the last one. The partial reconfiguration time for the next datapoint is hidden, as it occurs while the DE is running the current datapoint. This process repeats until all datapoints in the first chunk are processed.

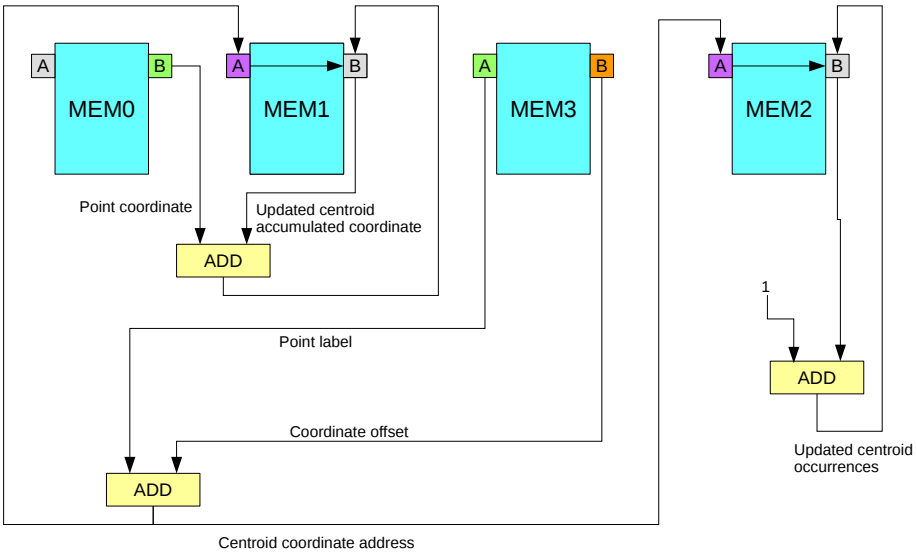

Figure 5.7: Datapath for the update step.

After the assignment step is done, the datapath for performing the update step, shown in figure 5.7, is loaded. In this datapath, the point coordinates, already loaded for the previous step, are read from port B of MEM0. The point labels (centroid indices), computed in the previous step, are read from port A of MEM3. Its port B is generating coordinate offsets synchronized with the datapoint coordinates. An ALU add labels and coordinate offsets to produce the accumulated coordinate address of the centroid

to update, which is fed to port A of MEM1. By configuration, port A can serve as the address for port B, another of Versat's novel features explained in section 3.1. Thus, the updated centroid accumulated coordinate is read from port B, added to the current point coordinate and stored back to port B at the same address. Each port has an input and an output from which can read an old value and write a new one to the same address. A similar loop is used for updating the number of occurrences of each centroid in MEM2, but it only updates the occurrences once per point. The number of occurrences for each centroid is stored in MEM2 and is incremented each time it is addressed, using the same address fed to port A of MEM1. The incrementer is implemented by another ALU. Note that any FU can select as input the commonly used constants 0 and 1, instead of a data bus entry. The two memory read-add-write self loops take 4 clock cycles due to the accumulated latencies of these operations. Therefore, the inner loop of the AGUs has size 4 ($Per = 4$), and the outer loop size equals the number of coordinates ($Iter =$ number of coordinates). Also note that port B of MEM1 and MEM2 are only enabled for writing in the last cycle of the 4-cycle inner loop ($Duty = 1$). After this nested loop runs, the DE is partially reconfigured to point to the next datapoint, as is done in the assignment step datapath. This process repeats until all datapoints in the first chunk are processed.

After the assignment and update steps are run, the next chunk of datapoints is loaded in MEM0 and the two steps are applied to the new chunk. This process is repeated until all datapoints are processed.

After all datapoint chunks are processed, the updated centroid accumulated coordinates reside in MEM1 and must be divided by the number of occurrences in MEM2 to yield the new centroid coordinates. Since the DE has no divider, the one that is coupled to Versat's controller is used to perform the divisions. While the divider is running, there is time to check whether the newly computed centroid coordinate differs from the one stored in MEM1, overwriting it if it does. If none of the new centroids is different from the stored ones, the algorithm terminates and the final centroids are DMA transferred to the external memory. If the point labels are also needed as a final result, then the assignment step must be run again for all datapoints, as the labels for each data chunk are not kept. This final loop represents a small overhead if the algorithm has many iterations. The labels produced during this extra iteration are DMA transferred to the external memory for each datapoint chunk.

Finally, the program clears register R0 in the CRF, which signals the host that Versat has finished execution.

Chapter 6

Results

In this chapter, experimental results for the Versat architecture are presented and discussed. First, the implementation results for Versat in FPGA, as well as in ASIC technology are presented. Then, performance and energy consumption results, both for simple and more complex computational kernels, are given. To end the chapter, Versat is compared to other CGRAs.

6.1 FPGA implementation results

The FPGA implementation results for Versat are given in Table 6.1. These results show that in terms of size, Versat is able to be implemented in the smallest FPGAs: in a Xilinx XC5VLX50T device of the Virtex V family (the one that is present on the Xilinx ML505 board), Versat occupies half of the logic resources; in an Altera Cyclone IV EP4CE22F17C6N device (present on Altera DE0 Nano board), Versat takes about 80% of the logic resources. The Xilinx implementation makes better use of the RAM resources for implementing the configuration memory than the Altera device. This explains why the Xilinx implementation consumes more RAM memory but less logic (LookUp Tables or LUTs), while the Altera device consumes more Logic Elements (LEs) and less RAM. This is also reflected in the frequency of operation, although the fact that two different architectures are being compared is also important; the Virtex V architecture can generally achieve higher frequencies of operation compared to the Cyclone IV architecture. Nevertheless, it is a fact that the current Versat implementation is not very well optimized in terms of the maximum frequency of operation, but more work can be done on this once it becomes a priority.

Table 6.1: FPGA implementation results.

Architecture	Logic	Regs	RAM(kB)	Mults	Fmax(MHz)
Cyclone IV	19366 LEs	4673	43.88	32 (9 bits)	64
Virtex V	12510 LUTs	4396	132.75	16 (18 bits)	102

6.2 ASIC implementation results

Versat has been designed using a UMC $130nm$ process. Table 6.2 compares Versat with a state-of-the-art embedded processor (ARM Cortex A9) and two other CGRA implementations (Morphosys [5] and ADRES [6]). The cores are compared in terms of the technology node (Node), silicon area (Area), embedded memory (RAM), frequency of operation (Frequency) and power consumption (Power). The Versat frequency and power results have been obtained using the Cadence IC design tools, and the node activity rate extracted from simulating an FFT kernel.

Table 6.2: ASIC implementation results.

Core	Node(nm)	Area(mm^2)	RAM(kB)	Freq.(MHz)	Power(mW)
ARM Cortex A9 [51]	40	4.6	65.54	800	500
Morphosys [5]	350	168	6.14	100	7000
ADRES [6]	90	4	65.54	300	91
Versat	130	5.2	46.34	170	132

Because the different designs use different technology nodes, it is difficult to compare the results in Table 6.2. In order to facilitate the comparisons, the results are scaled to the $40nm$ technology node and presented in Table 6.3. The scaling is performed as explained in [52].

Table 6.3: ASIC implementation results scaled to $40nm$.

Core	Area(mm^2)	RAM(kB)	Freq.(MHz)	Power(mW)
ARM Cortex A9 [51]	4.6	65.54	800	500
Morphosys [5]	2.19	6.14	875	800
ADRES [6]	0.79	65.54	675	40
Versat	0.49	46.34	553	41

The results show that Versat is the smallest core, and compared with the ARM processor it is $9\times$ smaller. The ADRES architecture is about twice the size of Versat and Morphosys is much larger, about half the size of the ARM processor. These differences can be explained by the different capabilities of these cores. While Versat has a 16-instruction controller and 11 FUs (excluding the memory units), ADRES has a VLIW processor and a 4x4 FU array, and Morphosys has a RISC processor and an 8x8 FU array.

In terms of embedded memory, Versat uses somewhat less memory than the ARM Cortex A9 or ADRES cores, but its memory size $(46kB)$ can be considered typical for an embedded processor. Morphosys uses a lot less memory, as it is designed to focus more on processing power and less on storage capabilities.

The ARM core operation frequency can be considered low as this is a power optimized version. Other versions of this core operating at higher frequencies exist, but the area footprint is larger and the power consumption is higher. Those versions are optimized for performance rather than power. Among the three CGRAs, Versat is the least optimized in terms of the working frequency. In fact, not too much effort has been put into achieving timing closure for a higher frequency. Notwithstanding, after analyzing

the critical paths, it became clear that there is plenty of room for optimization, so its frequency can be considered comparable to the other CGRAs.

As far as power is concerned, Morphosys consumes more than the ARM core. Again, this is the result of focusing in performance with a large array of FUs. The ADRES architecture seems well optimized for power, in spite of its cycle by cycle and progressive reconfiguration scheme. However, the acceleration that can be achieved with ADRES is not clearly documented in its publications [6]. Versat consumes about the same power as the ADRES core, but there is also room for improvement, given the little effort spent so far in low-level power optimization.

6.3 Execution results

In this section, Versat is compared with a state-of-the-art embedded processor in terms of performance and energy comsumption by running a set of example kernels on Versat and on the embedded processor. The results are divided in two parts: simple and complex kernels. In the simple kernels part, all tests operate on vector sizes of 1024, while in complex kernels part, larger data sets are used. All data is in 32-bit fixed-point format. A hardware timer has been used to measure the time in elapsed clock cycles. To end this section, results that show why self-generated configurations are better than pre-compiled stored configurations are presented.

In order to assess the performance of the Versat architecture, the Zybo Zynq-7000 ARM/FPGA SoC development board, featuring a Xilinx Zynq 7010 FPGA and a dual-core embedded ARM Cortex A9 system, has been used. Versat is connected as a peripheral to the ARM system using its AXI4 slave interface. The ARM system comprises a memory controller for accessing an external DDR3 module. Versat can also access this memory controller by connecting its AXI4 master interface to an appropriate AXI4 slave interface on the ARM system. The Zybo development board has been used only to measure the number of clock cycles for executing each kernel. The speedup was estimated using the following equation:

$$Speedup = \frac{t_{ARM} \times f_{Versat}}{t_{Versat} \times f_{ARM}}, \tag{6.1}$$

where t_{ARM} and t_{Versat} are the execution cycle count for the ARM and Versat, repectively, and f_{ARM} and f_{Versat} are their clock frequency in the $40nm$ process, according to table 6.3. The energy ratio was estimated by multiplying their execution time by their respective power consumption figure also given in table 6.3:

$$Energy\ Ratio = \frac{P_{ARM}}{P_{Versat}} \times Speedup. \tag{6.2}$$

6.3.1 Performance and energy consumption results for simple kernels

Results for the set of simple kernels are summarized in Table 6.4. These kernels use a single Versat configuration (no reconfiguration or reuse of data already in the accelerator), in order to get base values

for performance and energy consumption. In the next subsection, it will be shown that with massive reconfigurations and data reuse, performance and energy use improve significantly.

The results compare the performance of the Versat core with the performance of the ARM Cortex A9 core. The kernels are the following: `vadd` is a vector addition, `iir1` and `iir2` are 1^{st} and 2^{nd} order IIR filters and `cdp` is a dot (inner) product of two complex vectors. All kernels operate on Q1.31 fixed-point data with vector sizes of 1024.

For both systems, the program has been placed in on-chip memory and the data in the external DDR3 memory device. The *ARM* column denotes the execution cycle count for the ARM core. The *Versat* column gives the total cycle count for the Versat core, including data transfer, processing, control and reconfiguration. The *Control* column gives the unhidden control and reconfiguration cycles, that is, the number of these cycles that do not occur in parallel with the execution of the DE or DMA. The number of FUs used (column *#FUs*) and the code size in bytes (column *Code Size*) are also given for each kernel. The speedup and energy ratio have been obtained assuming the ARM and the Versat cores are running at the frequencies and power figures given in Table 6.3 for the $40nm$ technology node (equations 6.1 and 6.2). The speedup (column *Speedup*) is the ARM/Versat ratio of execution times. (The execution time is given by the cycle count divided by the frequency of operation.) The energy ratio (column *Energy Ratio*) is the energy spent by the ARM processor divided by the energy spent by the Versat core. The consumed energy is given by the execution time multiplied by the respective power consumption figure.

Table 6.4: Simple kernel results.

Kernel	ARM	Versat	Control	#FUs	Code Size	Speedup	Energy Ratio
`vadd`	14726	4517	36	4	152	2.25	27.44
`iir1`	18890	7487	26	5	220	1.74	21.22
`iir2`	24488	10567	26	11	332	1.60	19.51
`cdp`	25024	6673	26	14	408	2.59	31.59

From these results the main conclusion is that while the achievable speedups are modest, the energy gains are very significant for these single configuration kernels. This makes Versat a very attractive accelerator for high performance battery operated devices. The only requisite is that the vectors are long enough to justify the transfer of data in and/or out of Versat. Although not shown by the above results, the data transfer time dominates. For example, the `vadd` kernel processing time is only 1090 cycles and the remaining 3427 cycles account for data transfer and control.

The number of FUs used is low for the `vadd` and `iir1` kernels. The `vadd` kernel could perform multiple additions in parallel but it is not necessary as a single ALU is enough to hide the data transfer time if streaming the vectors. If the data were already in the DE then multiple ALUs in parallel would accelerate the processing time. The `iir2` and `cdp` kernels use more FUs as they require more computations per vector element.

The *Control* number of cycles is low for all examples, which shows that the configuration of the DE can be accomplished almost completely while the DMA is running. Configuration can also be done while

the DE is running, as will be shown in the next subsection where runtime reconfiguration is considered. In any case, the configuration overhead is low.

Given the simplicity of the examples, their code size, in the order of hundreds of bytes, is small. Many of these simple kernels can be placed in the $8kB$ program memory and invoked when necessary.

6.3.2 Performance and energy consumption results for complex kernels

In this subsection, three more complex kernels, that demand self and partial reconfiguration, are presented. The number of reconfigurations is high and the kernels operate for a long time on the data fetched from the external memory and/or produced by themselves. In these examples the data transfer time is less significant. The examples are the following:

- 1D-Convolution (conv-1D): very popular in applications such as Convolutional Neural Networks;
- Fast Fourier Transform (FFT): very common in digital signal processing (see section 5.1);
- K-Means Clustering algorithm (K-Means): widely used in Big Data applications (see section 5.2).

These examples are *parameterizable* and the parameters are passed by the host processor using the CRF. The algorithm that runs on Versat processes the parameters and generates configurations accordingly. This would be hard to achieve with statically compiled configurations and demonstrates the strength of self-generated configurations. Partial reconfigurations are equally important since they reduce considerably the reconfiguration time.

Results for 3 particular instances of the conv-1D, FFT and K-Means algorithms are detailed in Table 6.5. The conv-1D result was obtained by running the kernel on a dataset of 10^6 points applying 1-D convolution on a 256-point sliding window. The FFT result pertains a 16384-point window size with a 50% overlap over a dataset of 10^6 complex points. The K-Means result has been obtained for one iteration over a dataset of 1.36×10^6 points of 30 dimensions and 34 centroids.

Table 6.5: Complex kernel results.

Kernel	ARM	Versat	#FUs	Code Size	Speedup	Energy Ratio
`conv-1D`	2.26G	104.51M	13	668	14.95	182.32
`FFT`	1.28G	34.50M	12	3492	25.65	312.80
`K-Means`	9.02G	1.64G	12.5	2764	3.8	46.28

These results show that the speedup and energy efficiency improve with the complexity of the kernels when compared to the simpler kernels in the previous subsection. This has to do with the number of operations done in parallel in the DE, but also with the number of DE configurations that can be executed without fetching or saving new data in the external memory. The K-Means algorithm fetches a new datapoint chunk and applies 2 DE configurations: one for performing datapoint classification and the other to update the centroid positions. As for the FFT and conv-1D, after fetching a datapoint chunk, several configurations are applied corresponding to the several FFT stages, in case of the FFT and a window computation, in case of conv-1D. The datapaths for these kernels also expose a higher ILP and

DLP in its computations. Hence, the speedup and energy efficiency of the FFT and conv-1D are much higher when compared to the K-Means algorithm. All these algorithms illustrate the power of using the Versat CGRA compared to the ARM processor.

Conv-1D results

In the conv-1D kernel, the only parameterizable parameter is the window size. Versat/ARM speedup results are shown in figure 6.1. The window size is varied from 32 to 1024 and is increased by powers of 2. All datapoints, from the first data chunk, are multiplied by the sliding window coefficients and accumulated. Then Versat reconfigures itself to advance to the next window and the process repeats until the window is slid over all datapoints.

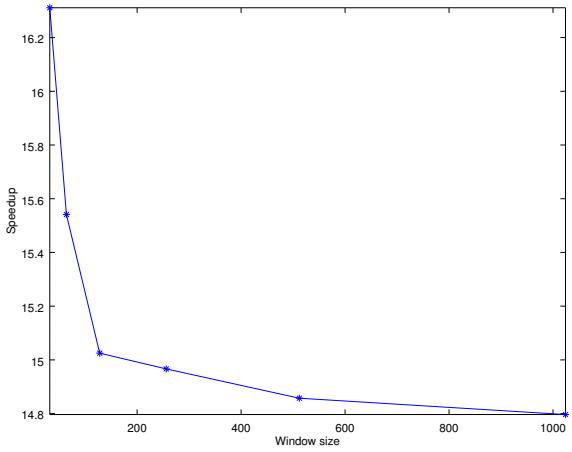

Figure 6.1: Conv-1D: speedup vs. window size.

As the sliding window size increases, the ARM core can better hide the time needed to store the computed values while the Versat core keeps its execution time linear on the window size. This causes a slight speedup drop, from 16.3 to 15.2, as shown in the figure. Due to the small size of the data chunks, the size of the internal memories is adequate for both cores.

FFT results

The FFT kernel can be parameterized with the following parameters: the number of datapoints, window size (must be a power of 2) and window overlap size. The algorithm computes the FFT successively, for the points in the sliding window, advancing the window for a number of points given by the window size minus the overlap size. In figure 6.2, the Versat/ARM speedup is shown as a function of the window size for 1 million datapoints and a half window (50%) overlap.

Initially, as the sliding window size reaches the capacity of the Versat memories, the speedup drops, while the ARM core uses its data cache and pre-fetch mechanism to sustain its performance. However, as the window size further increases, the ARM core reaches its own internal memory limitations for streaming data, and the speedup increases steadily again after that.

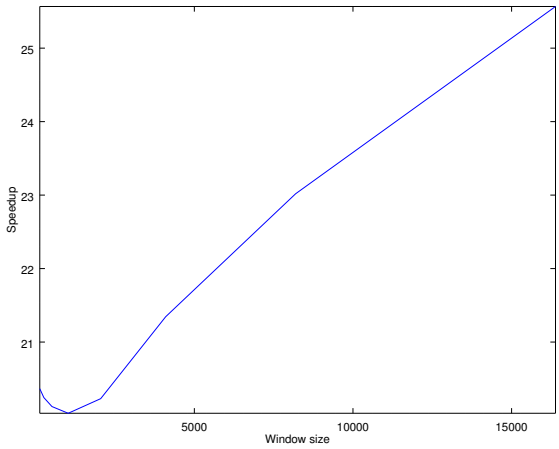

Figure 6.2: FFT: speedup vs. window size.

K-Means Clustering results

The parameters that can be passed to the K-Means Clustering kernel are the following: the number of datapoints, number of dimensions and number of centroids. In figure 6.3, it is shown how the time for one iteration varies with the number of datapoints for a fixed number of dimensions and centroids. The results are given for both cores using logarithmic scales.

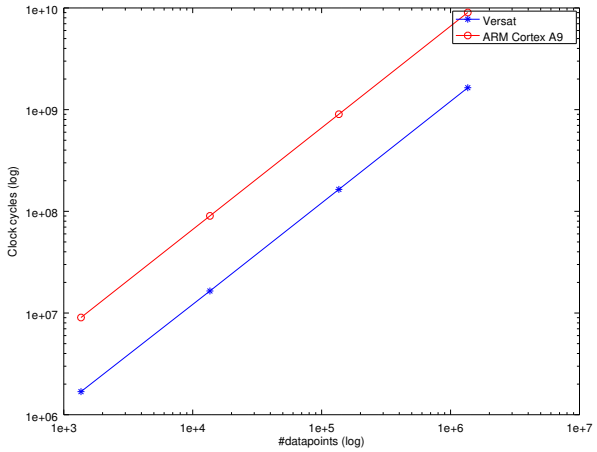

Figure 6.3: K-Means: iteration time vs. number of datapoints, for 30 dimensions and 34 centroids.

These results show that for both systems the execution time scales linearly with the number of datapoints. The Versat/ARM average speedup is 3.8, taking into account the number of clock cycles and the working frequencies of the cores (Table 6.3). Despite the modest speedup achieved in the K-Means Clustering algorithm, a considerable amount of energy can be saved.

Each point requires the DE to be reconfigured twice to apply the Assignment and Update steps. Since this algorithm is applied to millions of points then Versat is reconfigured millions of times at runtime. There is no reconfiguration overhead, as all reconfigurations are done while the DE or DMA is running.

6.3.3 Generated versus stored configurations

To show that using self-generated configurations has advantage compared to storing all configurations in the external memory, consider the K-Means Clustering kernel for instance.

This algorithm is especially useful when applied to large data sets. In the implementation above it has been chosen to apply it to millions of datapoints. Since for this algorithm Versat self-reconfigures for each point, then millions of configurations are needed. Since one configuration uses 672 bits then 672Mbits are needed only for configurations, much more than needed to store the program itself which generates the configurations.

For algorithms that require a number of configurations tied to the dataset size, pre-compiling and storing all configurations becomes simply not practical. Even for the FFT kernel that only requires 43 configurations, it can be proved that the self-generated version is marginally more efficient and has smaller memory footprint than the stored configurations version. Therefore, self-generated configurations is a good technique for CGRAs.

6.4 Comparing with other CGRAs

It is difficult to compare the Versat performance with the other CGRAs. Published results always omit important information, and it is hard to ascertain the conditions under which they have been obtained. A possible solution would be to replicate the other approaches in order to make fair comparisons. However, CGRAs are complex cores and implementing them from their descriptions in research papers, besides representing a formidable effort, is not guaranteed to yield trustworthy results, as some important details could be missed.

Notwithstanding, Versat can be compared with Morphosys for the FFT kernel, since it is reported in [53] that the processing time for a 1024-point FFT is 2613 cycles. Compared with the 12115 cycles taken by Versat, Morphosys is $4.6\times$ faster on this kernel at the same operating frequency. This is not surprising, since Morphosys has 64 FUs compared to only 11 FUs in Versat; Morphosys is $4.5\times$ the size of Versat according to Table 6.3. Using the frequency and power figures, it can be derived that Morphosys uses $2.66\times$ more energy than Versat for the FFT kernel.

Despite ADRES being one of the most cited CGRA architectures, comparisons with this architecture could not be made, since the execution times for the examples used in its publications have not been given in absolute terms.

Chapter 7

Conclusions

In this thesis we have described a new Coarse Grained Reconfigurable Array (CGRA) architecture, named Versat, and we have compared it with some other CGRA architectures. The main difference is that Versat uses self-generated and partial reconfiguration. These configurations are generated by an internal controller. The controller also takes care of data transfers to/from the external memory and simple algorithmic control. This allows Versat to independently run complex kernels such as the FFT and K-Means Clustering kernels, whose implementation has been presented in this thesis.

Versat is a minimal fixed-point CGRA with 4 dual-port embedded memories, 11 FUs, and a basic 16-instruction controller. Compared with other CGRAs with larger arrays, Versat requires a more sophisticated reconfiguration mechanism: the Versat controller can generate partial configurations and writes them field by field to a fully addressable configuration register file. The controller is also in charge of data transfers and basic algorithmic flows. This new architecture has several novel features: (1) one address generation unit per memory port, tightly coupled to it; (2) pointer support as values computed in the data engine can be used as memory addresses; (3) the address sequences produced by the address generators can be output and used in the data engine for any other purposes (data generators); (4) the ALUs support conditional and cumulative functions such as the accumulate and minimum functions, computed only if a condition is true. Feature (3) is especially useful for generating synchronization signals for the functional units in the data engine and feature (4) enables the execution of loops that contain if statements in their bodies.

Despite its limited compute resources, Versat is extremely flexible to program. In fact, it can be programmed like a traditional controller, where the program itself creates and partially reconfigures hardware datapaths in its data engine. The full mesh topology of the data engine is easy to deal with by programmers. Versat is capable of useful acceleration at low clock rates and, given its small size and parallelism, it can save orders of magnitude in terms of energy. It can be programmed in two different ways: in assembly language and using a C/C++ subset. To the best of our knowledge, Versat is the only CGRA architecture that can be programmed in assembly language. Assembling programmability provides ultimate control over all architecture details and is invaluable in software optimization, debug and system repair. Versat is designed to be a programmable alternative to dedicated hardware acceler-

ators, eliminating the risk of design errors. A software driver for host processors to use Versat has been developed. A host processor only needs a few function calls to program and run Versat kernels. The run function passes external memory data pointers and other specific parameters used in the Versat kernels. There is also a wait function used to check whether the kernel has finished. The host can run code in parallel with Versat by placing this code between the Versat run and wait functions.

7.1 Achievements

The results show that the Versat speedup and energy ratio improve with the kernel complexity. The kernel complexity depends on the number of datapaths (configurations) that can be run sequentially using the data already into the Versat memories. In fact, our results show that single datapath kernels achieve speedups in the order of $2\times$ and the gains in terms of energy are considerable, while multiple datapath kernels can achieve speedups an order of magnitude higher and save much more energy.

Unlike other CGRAs, which are designed to accelerate a single program loop, Versat is designed to accelerate a sequence of chained program loops, where the results produced in one loop are consumed by the next one. It has been explained that most of the times the next configuration can be generated while the DMA or the current configuration on the data engine is running. Because the Versat controller can generate configurations, these do not need to be stored in the external memory and then moved into Versat. In general, the code to generate configurations is much smaller than the configurations themselves.

Versat has been implemented in FPGA and in ASIC technology. In terms of silicon area, Versat is comparable to a basic low range CPU. Results on a VLSI implementation show that Versat is competitive in terms of silicon area, frequency of operation and power consumption. Versat is $9.4\times$ smaller than an ARM Cortex A9 processor and can achieve $553MHz$ of clock frequency, while the ARM core can run at $800MHz$ in the same $40nm$ technology node. Performance results on running an FFT kernel show that the Versat core can be $18\times$ faster and $220\times$ more energy efficient than the embedded processor. Running a K-Means Clustering algorithm, it is $3.8\times$ faster and consumes $46.3\times$ less energy. It should be clear from these numbers that GPUs and FPGAs cannot compete in this arena, and that the presented solution is useful in applications where cost and energy consumption are crucial.

Note that performance depends on how often such kernels are used in a complete application. In multimedia algorithms, for instance, between 40% to 80% of the execution time in a regular CPU can use an accelerator such as the one described here. This means that important overall speedups can be expected in practical applications.

This works has given rise to quite a few publications, one book chapter [7], and 5 conference papers [25, 9, 26, 11, 10], of which [26] won the best paper award.

7.2 Future Work

The work on Versat can be continued in many different fronts. First of all, many hardware optimizations can be done to improve the area, power and clock frequency even more. Reducing the configuration memory and applying clock gating techniques are two examples of what can be tried.

Floating point units are already being developed, as well as a Convolutional Neural Network (CNN) kernel for the Versat architecture, in two other master' thesis. By adding floating point units, algorithms that require this data type can be run on Versat. Since Versat can considerably accelerate a 1D-convolution kernel, CNN kernels that apply convolutions repeatedly with high data reuse are expected to have excellent performance and energy footprint compared to a CPU.

The Versat compiler needs to be upgraded and expounded. In fact many hew hardware features have been added since the compiler has been released. For example the compiler cannot make use of pointers in describing datapaths, which are already supported by the hardware. The same goes for using the address generators as general purpose sequence generators in datapaths. Moreover the compiler needs to support user declared variables, freeing the user to explicitly refer to hardware registers and memories.

Finally, more work on the architecture is needed. Namely, it is planned to increase the number of processing nodes (all FUs excluding memories), in order to move from accelerated embedded computing to high performance embedded computing or even super computing. This extended architecture has been called Deep Versat.

Bibliography

[1] J. E. White. Network specifications for remote job entry and remote job output retrieval at ucsb. *Network*, 1971.

[2] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, et al. A view of cloud computing. *Communications of the ACM*, 53(4):50–58, 2010.

[3] M. M. Gaber, J. B. Gomes, and F. Stahl. Pocket data mining. *Big Data on Small Devices. Series: Studies in Big Data*, 2014.

[4] P. Beckman, R. Sankaran, C. Catlett, N. Ferrier, R. Jacob, and M. Papka. Waggle: An open sensor platform for edge computing. In *SENSORS, 2016 IEEE*, pages 1–3. IEEE, 2016.

[5] M. hau Lee, H. Singh, G. Lu, N. Bagherzadeh, and F. J. Kurdahi. Design and Implementation of the MorphoSys Reconfigurable Computing Processor. In *Journal of VLSI and Signal Processing-Systems for Signal, Image and Video Technology*. Kluwer Academic Publishers, 2000.

[6] B. Mei, A. Lambrechts, J.-Y. Mignolet, D. Verkest, and R. Lauwereins. Architecture exploration for a reconfigurable architecture template. *Design & Test of Computers, IEEE*, 22(2):90–101, March 2005. ISSN 0740-7475. doi: 10.1109/MDT.2005.27.

[7] J. D. Lopes and J. T. de Sousa. Versat, a Minimal Coarse-Grain Reconfigurable Array. In *High Performance Computing for Computational Science, 12th International Meeting on*, 2016. Extended version to appear in a Springer LNCS book.

[8] R. Hartenstein, M. Herz, T. Hoffmann, and U. Nageldinger. Mapping Applications onto Reconfigurable KressArrays. In P. Lysaght, J. Irvine, and R. Hartenstein, editors, *Field Programmable Logic and Applications*, volume 1673 of *Lecture Notes in Computer Science*, pages 385–390. Springer Berlin Heidelberg, 1999. ISBN 978-3-540-66457-4. doi: 10.1007/978-3-540-48302-1_42. URL http://dx.doi.org/10.1007/978-3-540-48302-1_42.

[9] J. D. Lopes, R. Santiago, and J. T. de Sousa. Versat, a Runtime Partially Reconfigurable Coarse-Grain Reconfigurable Array using a Programmable Controller. In *II Jornadas Sarteco*, pages 561–569, Salamanca, Sept. 2016.

[10] J. D. Lopes and J. T. de Sousa. Fast Fourier Transform on the Versat CGRA. In *Accepted for publication in Proceedings of the Jornadas Sarteco*, pages ??–??, Sept. 2017.

[11] J. D. Lopes, J. T. de Sousa, and H. Neto. K-Means Clustering on CGRA. In *Accepted for publication in Proceedings of the 27th International Conference on Field-Programmable Logic and Applications, New Paradigms and Compilers*, FPL 2017, pages ??–??, Sept. 2017.

[12] K. Moreland and E. Angel. The FFT on a GPU. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, pages 112–119. Eurographics Association, 2003.

[13] M. Doggett, W. Heidrich, W. Mark, and A. Schilling. The FFT on a GPU.

[14] S. Che, J. Meng, J. W. Sheaffer, and K. Skadron. A performance study of general purpose applications on graphics processors. In *First Workshop on General Purpose Processing on Graphics Processing Units*, 2007.

[15] R. Farivar, D. Rebolledo, E. Chan, and R. H. Campbell. A Parallel Implementation of K-Means Clustering on GPUs. In *PDPTA'08 - The 2008 International Conference on Parallel and Distributed Processing Techniques and Applications*, pages 340–345, 2008. URL `http://dblp.uni-trier.de/db/conf/pdpta/pdpta2008.html#FarivarRCC08`.

[16] C. Chao, Z. Qin, X. Yingke, and H. Chengde. Design of a high performance FFT processor based on FPGA. In *Proceedings of the 2005 Asia and South Pacific Design Automation Conference*, pages 920–923. ACM, 2005.

[17] Z. Sun, X. Liu, and Z. Ji. The design of radix-4 FFT by FPGA. In *Intelligent Information Technology Application Workshops, 2008. IITAW'08. International Symposium on*, pages 765–768. IEEE, 2008.

[18] Z. H. Derafshi, J. Frounchi, and H. Taghipour. A high speed FPGA implementation of a 1024-point complex FFT processor. In *Computer and Network Technology (ICCNT), 2010 Second International Conference on*, pages 312–315. IEEE, 2010.

[19] W.-C. Liu, J.-L. Huang, and M.-S. Chen. KACU: k-means with hardware centroid-updating. In *Conference, Emerging Information Technology 2005.*, pages 3–, Aug 2005. doi: 10.1109/EITC.2005.1544347.

[20] H. M. Hussain, K. Benkrid, H. Seker, and A. T. Erdogan. FPGA implementation of K-means algorithm for bioinformatics application: An accelerated approach to clustering Microarray data. In *2011 NASA/ESA Conference on Adaptive Hardware and Systems (AHS)*, pages 248–255, June 2011. doi: 10.1109/AHS.2011.5963944.

[21] J. S. S. Kutty, F. Boussaid, and A. Amira. A high speed configurable FPGA architecture for k-mean clustering. In *2013 IEEE International Symposium on Circuits and Systems (ISCAS2013)*, pages 1801–1804, May 2013. doi: 10.1109/ISCAS.2013.6572215.

[22] T. Canli, A. Gupta, and A. Khokhar. Power Efficient Algorithms for Computing Fast Fourier Transform over Wireless Sensor Networks. In *Computer Systems and Applications, 2006. IEEE International Conference on.*, pages 549–556. IEEE, 2006.

[23] P. Sasikumar and S. Khara. K-Means Clustering in Wireless Sensor Networks. In *Proceedings of the 2012 Fourth International Conference on Computational Intelligence and Communication Networks*, CICN '12, pages 140–144, Washington, DC, USA, 2012. IEEE Computer Society. ISBN 978-0-7695-4850-0. doi: 10.1109/CICN.2012.136. URL http://dx.doi.org/10.1109/CICN.2012.136.

[24] X. Xu, R. Ansari, and A. Khokhar. Power-efficient algorithms for Fourier analysis over random wireless sensor network. In *Distributed Computing in Sensor Systems (DCOSS), 2012 IEEE 8th International Conference on*, pages 109–115. IEEE, 2012.

[25] J. T. de Sousa and J. D. Lopes. Versat, a Minimal Coarse-Grain Reconfigurable Array. In *High Performance Computing for Computational Science, 12th International Meeting on*, 2016.

[26] R. Santiago, J. D. Lopes, and J. T. de Sousa. Compiler for the Versat reconfigurable architecture. In *XIII Jornadas sobre Sistemas Reconfiguráveis*, Aveiro, January 2017.

[27] V. Baumgarte, G. Ehlers, F. May, A. Nückel, M. Vorbach, and M. Weinhardt. PACT XPP – A Self-Reconfigurable Data Processing Architecture. *The Journal of Supercomputing*, 26(2):167–184, 2003. ISSN 0920-8542. doi: 10.1023/A:1024499601571. URL http://dx.doi.org/10.1023/A%3A1024499601571.

[28] M. Quax, J. Huisken, and J. Van Meerbergen. A scalable implementation of a reconfigurable WCDMA RAKE receiver. In *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition, 2004*, volume 3, pages 230–235 Vol.3, Feb 2004. doi: 10.1109/DATE.2004.1269235.

[29] J. de Sousa, V. Martins, N. Lourenco, A. Santos, and N. do Rosario Ribeiro. Reconfigurable coprocessor architecture template for nested loops and programming tool, Sept. 25 2012. URL http://www.google.com/patents/US8276120. US Patent 8,276,120.

[30] J. L. Tripp, J. Frigo, and P. Graham. A survey of multi-core coarse-grained reconfigurable arrays for embedded applications. *Proc. of HPEC*, 2007.

[31] H. Park, Y. Park, and S. Mahlke. Polymorphic Pipeline Array: A Flexible Multicore Accelerator with Virtualized Execution for Mobile Multimedia Applications. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 42, pages 370–380, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-798-1. doi: 10.1145/1669112.1669160. URL http://doi.acm.org/10.1145/1669112.1669160.

[32] R. Hartenstein. Coarse Grain Reconfigurable Architecture (Embedded Tutorial). In *Proceedings of the 2001 Asia and South Pacific Design Automation Conference*, ASP-DAC '01, pages 564–570, New York, NY, USA, 2001. ACM. ISBN 0-7803-6634-4. doi: 10.1145/370155.370535. URL http://doi.acm.org/10.1145/370155.370535.

[33] C. Ebeling, D. C. Cronquist, and P. Franklin. RaPiD - Reconfigurable Pipelined Datapath. In *Proceedings of the 6th International Workshop on Field-Programmable Logic, Smart Applications, New Paradigms and Compilers*, FPL '96, pages 126–135, London, UK, 1996. Springer-Verlag. ISBN 3-540-61730-2. URL http://dl.acm.org/citation.cfm?id=647923.741212.

[34] L. Liu, D. Wang, M. Zhu, Y. Wang, S. Yin, P. Cao, J. Yang, and S. Wei. An Energy-Efficient Coarse-Grained Reconfigurable Processing Unit for Multiple-Standard Video Decoding. *IEEE Transactions on Multimedia*, 17(10):1706–1720, Oct 2015. ISSN 1520-9210. doi: 10.1109/TMM.2015.2463735.

[35] C. Van Berkel and P. Meuwissen. Address generation unit for a processor, Jan. 12 2006. URL http://www.google.la/patents/US20060010255. US Patent App. 10/515,462.

[36] S. B. Wijeratne, N. Siddaiah, S. K. Mathew, M. A. Anders, R. K. Krishnamurthy, J. Anderson, M. Ernest, and M. Nardin. A 9-GHz 65-nm Intel® Pentium 4 processor integer execution unit. *IEEE Journal of Solid-State Circuits*, 42(1):26–37, 2007.

[37] D. Cormie. The ARM11 microarchitecture. *ARM Ltd. White Paper*, 2002.

[38] S. M. Carta, D. Pani, and L. Raffo. Reconfigurable Coprocessor for Multimedia Application Domain. *Journal of VLSI Signal Processing Systems for Signal, Image and Video Technology*, 44(1-2):135–152, 2006. ISSN 0922-5773. doi: 10.1007/s11265-006-7512-7. URL http://dx.doi.org/10.1007/s11265-006-7512-7.

[39] B. De Sutter, P. Raghavan, and A. Lambrechts. Coarse-Grained Reconfigurable Array Architectures. In S. S. Bhattacharyya, E. F. Deprettere, R. Leupers, and J. Takala, editors, *Handbook of Signal Processing Systems*, pages 449–484. Springer US, 2010. ISBN 978-1-4419-6344-4. doi: 10.1007/978-1-4419-6345-1_17. URL http://dx.doi.org/10.1007/978-1-4419-6345-1_17.

[40] P. Heysters and G. Smit. Mapping of DSP algorithms on the MONTIUM architecture. In *Proceedings of the International Parallel and Distributed Processing Symposium, 2003*, pages 6–, April 2003. doi: 10.1109/IPDPS.2003.1213333.

[41] Y. Park, J. Park, and S. Mahlke. Efficient performance scaling of future CGRAs for mobile applications. In *International Conference on Field-Programmable Technology (FPT), 2012*, pages 335–342, Dec 2012. doi: 10.1109/FPT.2012.6412158.

[42] B. R. Rau. Iterative Modulo Scheduling: An Algorithm for Software Pipelining Loops. In *Proceedings of the 27th Annual International Symposium on Microarchitecture*, MICRO 27, pages 63–74, New York, NY, USA, 1994. ACM. ISBN 0-89791-707-3. doi: 10.1145/192724.192731. URL http://doi.acm.org/10.1145/192724.192731.

[43] V. Betz, J. Rose, and A. Marquardt, editors. *Architecture and CAD for Deep-Submicron FPGAs*. Kluwer Academic Publishers, Norwell, MA, USA, 1999. ISBN 0792384601.

[44] A. Severance and G. Lemieux. Embedded supercomputing in FPGAs with the VectorBlox MXP Matrix Processor. In *Hardware/Software Codesign and System Synthesis (CODES+ISSS), 2013 International Conference on*, pages 1–10, Sept 2013. doi: 10.1109/CODES-ISSS.2013.6658993.

[45] A. Severance, J. Edwards, H. Omidian, and G. Lemieux. Soft Vector Processors with Streaming Pipelines. In *Proceedings of the 2014 ACM/SIGDA International Symposium on Field-programmable Gate Arrays*, FPGA '14, pages 117–126, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2671-1. doi: 10.1145/2554688.2554774. URL http://doi.acm.org/10.1145/2554688.2554774.

[46] M. Naylor and S. Moore. Rapid codesign of a soft vector processor and its compiler. In *24th International Conference on Field Programmable Logic and Applications (FPL), 2014*, pages 1–4, Sept 2014. doi: 10.1109/FPL.2014.6927425.

[47] R. Santiago. Compiler for the VERSAT Reconfigurable Processor. Master's thesis, Instituto Superior Técnico, May 2016.

[48] N. Farahini, A. Hemani, H. Sohofi, S. M. Jafri, M. A. Tajammul, and K. Paul. Parallel Distributed Scalable Runtime Address Generation Scheme for a Coarse Grain Reconfigurable Computation and Storage Fabric. *Microprocess. Microsyst.*, 38(8):788–802, Nov. 2014. ISSN 0141-9331. doi: 10.1016/j.micpro.2014.05.009. URL http://dx.doi.org/10.1016/j.micpro.2014.05.009.

[49] D. Liu, S. Yin, L. Liu, and S. Wei. Polyhedral model based mapping optimization of loop nests for CGRAs. In *Design Automation Conference (DAC), 2013 50th ACM/EDAC/IEEE*, pages 1–8, May 2013.

[50] M. Estlick, M. Leeser, J. Theiler, and J. J. Szymanski. Algorithmic Transformations in the Implementation of K- Means Clustering on Reconfigurable Hardware. In *Proceedings of the 2001 ACM/SIGDA Ninth International Symposium on Field Programmable Gate Arrays*, FPGA '01, pages 103–110, New York, NY, USA, 2001. ACM. ISBN 1-58113-341-3. doi: 10.1145/360276.360311. URL http://doi.acm.org/10.1145/360276.360311.

[51] W. Wang and T. Dey. A survey on ARM Cortex A processors. http://www.cs.virginia.edu/~skadron/cs8535_s11/ARM_Cortex.pdf. Accessed 2016-04-16.

[52] S. Borkar. Design challenges of technology scaling. *IEEE Micro*, 19(4):23–29, Jul 1999. ISSN 0272-1732. doi: 10.1109/40.782564.

[53] A. H. Kamalizad, C. Pan, and N. Bagherzadeh. Fast parallel FFT on a reconfigurable computation platform. In *Computer Architecture and High Performance Computing, 2003. Proceedings. 15th Symposium on*, pages 254–259, Nov 2003. doi: 10.1109/CAHPC.2003.1250345.