electronics

MDPI

Article

Coarse-Grained Reconfigurable Computing with the Versat Architecture

João D. Lopes [1], Mário P. Véstias [2,*], Rui Policarpo Duarte [1], Horácio C. Neto [1] and José T. de Sousa [1]

[1] INESC-ID, Instituto Superior Técnico, Universidade de Lisboa, 1049-001 Lisbon, Portugal; joao.d.lopes@tecnico.ulisboa.pt (J.D.L.); rui.duarte@tecnico.ulisboa.pt (R.P.D.); hcn@inesc-id.pt (H.C.N.); jose.desousa@inesc-id.pt (J.T.d.S.)

[2] INESC-ID, Instituto Superior de Engenharia de Lisboa, Instituto Politécnico de Lisboa, 1959-007 Lisbon, Portugal

* Correspondence: mvestias@deetc.isel.ipl.pt; Tel.: +351-218-317-000

Abstract: Reconfigurable computing architectures allow the adaptation of the underlying datapath to the algorithm. The granularity of the datapath elements and data width determines the granularity of the architecture and its programming flexibility. Coarse-grained architectures have shown the right balance between programmability and performance. This paper provides an overview of coarse-grained reconfigurable architectures and describes Versat, a Coarse-Grained Reconfigurable Array (CGRA) with self-generated partial reconfiguration, presented as a case study for better understanding these architectures. Unlike most of the existing approaches, which mainly use pre-compiled configurations, a Versat program can generate and apply myriads of on-the-fly configurations. Partial reconfiguration plays a central role in this approach, as it speeds up the generation of incrementally different configurations. The reconfigurable array has a complete graph topology, which yields unprecedented programmability, including assembly programming. Besides being useful for optimising programs, assembly programming is invaluable for working around post-silicon hardware, software, or compiler issues. Results on core area, frequency, power, and performance running different codes are presented and compared to other implementations.

Keywords: reconfigurable computing; coarse-grained reconfigurable arrays; heterogeneous systems; low power systems

Citation: Lopes, J.D.; Véstias, M.P.; Duarte, R.P.; Neto, H.C.; de Sousa, J.T. Coarse-Grained Reconfigurable Computing with the Versat Architecture. *Electronics* **2021**, *10*, 669. https://doi.org/10.3390/electronics10060669

Academic Editor: Akash Kumar

Received: 7 February 2021
Accepted: 9 March 2021
Published: 12 March 2021

1. Introduction

Reconfigurable computing merges the flexibility of software with the performance of hardware. A reconfigurable architecture allows modifications to the computing datapath to improve the runtime of a particular algorithm. A suitable type of reconfigurable hardware is the Coarse-Grained Reconfigurable Array (CGRA) [1]. A CGRA is a collection of programmable functional units (FUs) interconnected by programmable switches. The FUs operate on data words of commonly used widths such as 8, 16, 32 or 64-bit words. When the CGRA is programmed it implements dedicated hardware datapaths that accelerate computations. Its high performance at low energy makes it a target architecture for both high-performance and embedded computing.

Fine-grained reconfigurable fabrics, such as Field Programmable Gate Arrays (FPGAs), also permit tailoring the underlying datapath. However, instead of word-level FUs, FPGAs use bit-level Look Up Tables (LUTs), making them too large and power-hungry for some target applications [2]. FPGAs integrate programmable coarse-grained blocks (like functional units, memory blocks, memory interfaces, and even processors) to alleviate the fine-level reconfigurability logic's burden fabric. This practice improves performance and reduces energy consumption but reduces flexibility.

CGRAs are generally used to accelerate program loops whose bodies contain array operations; the parts of the program which do not contain loops run on a classical pro-

cessor. For this reason, CGRA architectures typically feature a processor core [3,4]. With a higher granularity, CGRAs can operate at higher frequencies than FPGAs, providing higher performance, faster compilation, much lower reconfiguration times and lower energy consumption.

In this paper, we summarise some of the most relevant CGRAs, identifying their main architectural aspects. To fully understand the architectural options associated with CGRAs, we present Versat, a coarse-grained reconfigurable architecture for embedded systems. The Versat architecture is described, analysed, and experimental results are presented.

2. Coarse-Grained Reconfigurable Architectures

Before reviewing the existing coarse-grained reconfigurable architectures, it is essential to explain our understanding of what should be considered a coarse-grained reconfigurable architecture.

In general, a reconfigurable architecture [5] has some level of hardware reconfigurability that allows the adaptation of the datapath statically at compile time or dynamically at runtime. According to this definition, almost every computing system is reconfigurable, from a general or specific purpose processor, through FPGAs to dedicated application-specific integrated circuit (ASIC) devices.

A general processor reconfigures its arithmetic and logic unit (ALU) at an instruction level since it determines the configuration bits of the register file and the ALU for each instruction. On the other end, an ASIC can be designed without any configuration or with multiple alternative datapaths at different configuration levels. In any case, they are usually designed for a specific application where the need for of a configurable datapath is usually low.

The more configurability we add to a device, the more flexible it becomes to adapt the underlying architecture to a particular problem. However, the resources allocated to configurability reduce performance and increase area and energy consumption. So, one way to qualitatively classify reconfigurable architectures is through the granularity of the reconfiguration. The FPGA has the lowest granularity since it allows the reconfiguration of both logic and routing at the bit level. One can design and map high-level overlays with coarse reconfigurability on an FPGA, but the underlying device is always fine-grained. Coarse-grained reconfigurable devices adopt a higher reconfiguration level at or above functional units working with data words. Coarse-grained architectures [6,7] are those that use a bus interconnect and functional units, such as ALUs that perform more than just bitwise operations. The granularity level applies to all subsystems of the architecture, including computing, memory and interconnect. The reconfigurability of a CGRA is somewhere between an FPGA and an ASIC. In general, the reconfigurability of a CGRA architecture is enough to guarantee the efficient design of a specific domain of applications.

The granularity of a device does not have to be regular. For example, an FPGA contains different granularity levels—the low-level fabric logic and routing and a coarser level with DSP, BRAM (Block-RAM), memory interfaces and even processors. A coarse-grained architecture can also be multi-granular with a lower granularity logic inside an FU. However, what defines the granularity of a device is its general reconfiguration level.

Being reconfigurable does not mean it has to be generic. FPGAs are generic reconfigurable devices to implement any system, provided it has enough resources. However, even the FPGA can be application-oriented depending on the ratio between fine-grained logic and coarse-grained modules. Coarse-grained architectures can also be accelerators for a particular application domain. In this case, besides an array of reconfigurable functional units, additional domain-specific functions are added to the FUs or peripheral logic to turn the architecture for greater efficiency. Otherwise, these operations would have to be delegated to a CPU, increasing the application's execution time.

CGRAs have better performance, energy and integration density than fine-grained FPGAs at the cost of reduced flexibility for domain-specific applications. The remainder of this section presents a brief overview of the evolution of CGRAs.

A CGRA includes a mesh of reconfigurable functional units (RFU), the smallest reconfigurable unit. Data input and output can be controlled directly by a host processor or using address generator units (AGU) attached or not to the functional units (FU). The functional unit includes an ALU, input and output multiplexers to route input and output data, and optionally may include a local memory. Each functional unit connects to adjacent units to form a dedicated datapath. A configuration buffer statically or dynamically configures the FU.

The functional units' type and architecture, the interconnection architecture, and the level of reconfigurability define the CGRA. During the last years many different CGRA were proposed [8–11]. The following paragraphs present a brief overview of known CGRAs to contextualise the Versat CGRA.

The pioneering FPGAs were somehow limited in the data width, and the architecture of the functional units [12,13]. While the GARP architecture in [12] still designs the functional units as look-up tables, like in an FPGA, the reconfigurable units in CHESS [13] use a structure similar to an ALU.

While GARP and CHESS considered reconfiguration registers to store configurations, the RAW [14] CGRA included a dedicated memory to store configurations. These include the configuration bits of a reconfigurable network-on-chip, which turns the interconnection network very flexible and adaptable to a wide range of interconnection topologies. A different hierarchical interconnection topology was considered in KressArray [15]. The architecture includes address generators to stream data into the datapath configured in the CGRA. Another hybrid interconnection topology was implemented in PipeRench [16] with a mesh interconnection of linear arrays of functional units.

Soon, CGRA with larger data widths and closer to current CGRA configurations started to appear. MATRIX [17] considered an 8-bit datapath and a mesh of ALUs. ALUs are still fine-grained programmable, similar to an FPGA, but the interconnection topology has been improved, allowing connections to non-adjacent neighbour FU and networking operations, like data merge. This CGRA was followed by REMARC [18] that increased the data width to 16 bits and improved the FU integrating a register file with the ALU. MorphoSys [19] is another 16-bit mesh-topology CGRA similar to REMARC with an ALU extended with a shifter and a multiplier.

These earlier CGRA served as the background for more recent architectures. Besides increasing the number of functional units, the next generation of CGRAs is more domain-oriented. ADRES [20] targets embedded architectures and has typical ALUs as FUs, a register file and input/output multiplexers to forward data. A unique feature of ADRES is the possibility to configure a row of the mesh as a very long instruction. TRIPS/EDGE [21] is another proposal to increase the number of parallel instructions executed in the CGRA. A compiler was developed, together with the architecture, to execute up to 16 instructions in parallel. This approach was one of the first attempts to consider a CGRA for high-performance computing.

CGRAs can also target specific application domains. MORA [22] is a CGRA for media processing applications. It consists of an array of a 2D mesh of functional units with neighbor interconnections. Functional units has one scratchpad memory and an 8 bit datapath architecture with functional units optimized for the most common media processing operations.

Some CGRA architectures consider heterogeneous blocks. The reconfigurable array of BilRC [23] has generic ALU blocks, dedicated blocks for multiplication and shift and memory blocks. A full functional unit with ALU, multiplier and memory can be obtained with the tight interconnection of the three blocks.

Floating-point support is not common on CGRA architectures. However, some CGRAs have proposed the integration of floating-point units in the functional unit. FloRA [24], a 16-bit architecture has support for floating-point (FP) operations. The architecture does not include an FP unit in each FU. Instead, two FU must be combined to implement a

single-precision floating-point operation. Other CGRAs with floating-point capability have followed [25,26].

To improve the data rate of data input and output, AMIDAR [27], a recent CGRA architecture includes a direct memory access (DMA) module. It also includes the capacity for multiple configuration contexts and dedicated hardware support to implement branches.

An exciting novel approach for CGRA design was introduced in [28] that proposes a CGRA for general-purpose computing. It includes new constructions to support particular computation patterns like inner-products and new forms of loops.

Some CGRA target high-performance computing [29–34]. The main concern of a CGRA for high-performance computing is how to scale the interconnection network. Extending a mesh-based architecture is a non-scalable solution. Instead, these proposals consider a heterogeneous interconnection topology with a non-uniform distribution of data bandwidth.

A relatively small FU array at low frequencies reduces energy consumption and still provides enough computing power for many embedded systems. While most CGRAs target embedded systems, high-performance computing CGRA can also be designed by increasing the FU array and improve parallelism. The main challenge of these solutions is how to deploy them for general-purpose computing with high utilisation efficiency of the available resources. How can one map complete applications in a very regular, domain-oriented architecture? Can different kernels working simultaneously in different zones of the CGRA exploit the fast context switching of CGRAs?

Commercial applications of CGRA are still scarce due to the lack of maturity of design tools and productivity. Tool support determines the deployment of any CGRA. The success of a CGRA is highly dependent on the maturity of associated tools for compilation, simulation, verification and debug.

The following sections describe the Versat CGRA as a successful CGRA for embedded computing with dynamic context switching, fast data input and output and development tools.

3. Versat: A Coarse-Grained Reconfigurable Architecture for Embedded Systems

The design of the Versat CGRA started with the following observation—the compute-intensive inner loops typically accelerated in a CGRA tend to form clusters in the code, with the data produced in one loop consumed in the next loop. However, in between these loops, there is always code that is unsuitable for running on the CGRA. This code needs to run on the host processor. The frequent interventions of the host processor, which is also in charge of reconfiguring the array and transferring data, may result in a time overhead that cancels the CGRA acceleration.

A straightforward example is a two-level nested loop, where the inner loop is mapped to the CGRA and can be considerably accelerated. However, the host processor executes the outer loop control and does the following:

1. configures the array for the next inner loop
2. stores the computed data in the external memory
3. loads the new data in the array.

If the host processor cannot do this management fast enough, it may offset the performance gain of the CGRA. In such cases, executing the whole procedure on the host processor may turn out to be faster.

Previous work has targeted this problem by proposing CGRAs that can support nested loops. For example, the approach in [35] supports nested loops using select address generation units and has been successfully used in commercial audio codec applications.

This paper extends the work in [35] by adding a minimal programmable controller to the reconfigurable array, which can manage reconfiguration, data transfer and simple control code in between the accelerated program loops. This controller reduces the overhead caused by the host processor's frequent interventions and increases the CGRA tasks' granularity. It simplifies the host processor's programming, but requires program-

ming the CGRA rather than just reconfiguring it. The host processor is loosely coupled with the CGRA, whereas the CGRA controller is tightly coupled and micromanages the reconfigurable array.

The proposed architecture, Versat, uses a relatively small FU array coupled with the said programmable controller. A small array limits the size of the data expressions mapped to the CGRA, forcing the breaking down of large expressions to into smaller expressions executed sequentially on the CGRA. Therefore, Versat requires mechanisms for efficiently handling large numbers of configurations. Since Versat has its controller, it can generate its configurations rather than sourcing them from external memory. *Self-reconfiguration* is the most prominent feature of Versat. Storing routines that generate configuration sequences is much more efficient than storing the configurations themselves because most configurations are similar.

Partial reconfiguration exploits the similarity between CGRA configurations. If only a few configuration bits differ between two configurations, then only those bits are changed. Most CGRAs are only fully reconfigurable [3,4,36] and do not support partial reconfiguration. The disadvantage of performing full reconfiguration is the amount of configuration data kept or fetched from memory. Previous CGRA architectures with support for partial reconfiguration include RaPiD [37], PACT [38] and RPU [39]. RaPiD [37] supports dynamic (cycle by cycle) partial reconfiguration for a subset of the configuration bitstream, using smaller stored contexts. In PACT [38], one of the processors has to access the configuration memory, organised to make partial reconfiguration possible. However, this mechanism is reportedly slow, and the authors recommend avoiding it and resorting to full reconfiguration whenever possible. RPU [39] proposes a kind of partial reconfiguration called Hierarchical Configuration Context to mitigate these problems. However, none of the approaches reviewed supports self-reconfiguration and cycle by cycle reconfiguration is power-hungry.

This work proposes a partial reconfiguration scheme that uses a configuration register file and a configuration memory for temporarily storing full configurations. The configuration register file comprises registers of variable length, where each register corresponds to an atomic configuration field. Random access is allowed to the configuration fields, a more flexible approach than the hierarchical organisation of configuration memory contexts proposed in [39]. Moreover, user programs running on the Versat controller generate the configurations at runtime.

Versat cores are co-processors in an embedded system containing one or more application processors. These co-processors optimise performance and energy consumption during the execution of compute-intensive tasks. Application programmers can use them by simply calling special procedures in a program that runs on a host processor. A Versat API library provides this functionality. Since Versat programmers create the API library, the software and programming tools of the CGRA are separate from those of the application processor.

A compiler for Versat has been developed and presented in [40]. The compiler has restricted functionality but very efficiently accelerates tasks running on Versat. The syntax of the programming language is a subset of the C/C++ language, with a semantics that enables the description of hardware data. This paper does not describe the compiler, whose intent is describing the architecture and VLSI implementation.

3.1. Versat Architecture

Versat's architecture is outlined in Figure 1. At the top, one can see the Versat Controller and its Program Memory. The controller can access the various modules in the system through the Control Bus. The Controller executes programs stored in the Program Memory (9 kB = 1 kB boot ROM + 8 kB RAM). User programs are loaded in the RAM to execute algorithms which involve managing DE reconfigurations and DMA data transfers.

Versat user programs can use the Data Engine (DE) to carry out data-intensive computations. To perform these computations, the controller writes DE configurations to the

Configuration Module (CM) or restores configurations previously stored in the CM. The controller can load the DE with data to be processed or save the processed data back in the external memory using the DMA engine. The DMA engine also loads the Versat program and moves CGRA configurations between the core and external memory.

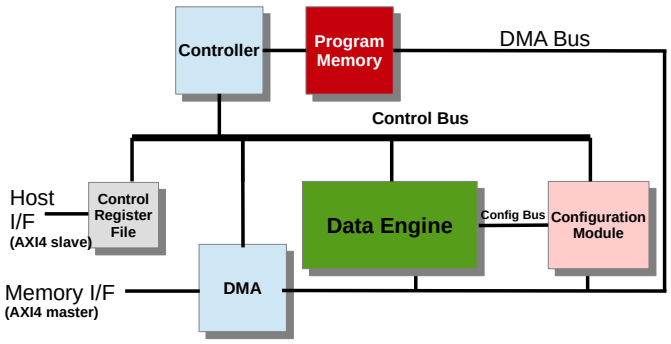

Figure 1. Versat top-level entity.

The Versat core has a host interface and a memory interface. Both interfaces use ARM's Advanced eXtensible Interface (AXI), a de facto standard for busing. A host system uses the host interface (AXI slave) to instruct Versat to load and execute programs. The host and the controller communicate using the Control Register File (CRF). The CRF is also used by Versat programs as a general-purpose (1-cycle access) register file. The memory interface (AXI master) is used to access data from an external memory using the DMA.

3.2. Data Engine

The Data Engine (DE) has a fixed topology and comprises 15 functional units (FUs), as shown in Figure 2. The DE is a 32-bit architecture, and the configurable FUs are the following: six Arithmetic and Logic Units (ALUs), four multipliers, one barrel shifter and four dual-port 8kB embedded memories, also treated as FUs for the sake of uniformity. The output register of the FUs is read/write accessible by the controller via the Control Bus.

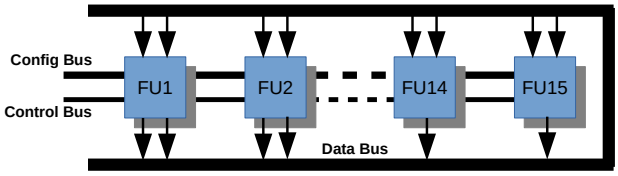

Figure 2. Data engine.

3.2.1. 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. An embedded memory contributes two sections to the Data Bus since it has two ports. Thus, according to the number of FUs of each type given before, the Data Bus has 19 sections of 32 bits. Each FU input can select any one of these sections. 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. A fixed set of word-level operations are available in each FU. For example, an ALU can be configured to perform addition, subtraction, logical AND, maximum and minimum and other functions.

Figure 3 shows how an example FU is connected to the control, data, and configuration buses. The example FU, of type ALU and labelled FU5, has two pipeline stages. The last pipeline stage, denoted `pipeline register 1`, stores the ALU output (output register), drives one section of the Data Bus and is also accessible by the Control Bus for reading and writing. The last 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. Only the configuration bits of each FU are routed from the Config Bus to that FU. These bits are called the *configuration space* of the FU. The configuration space is further divided in *configuration fields*. FU5 has three configuration fields: 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—only one field can be reconfigured at a time; it is impossible to change multiple fields simultaneously, except when loading a full configuration from the CM memory.

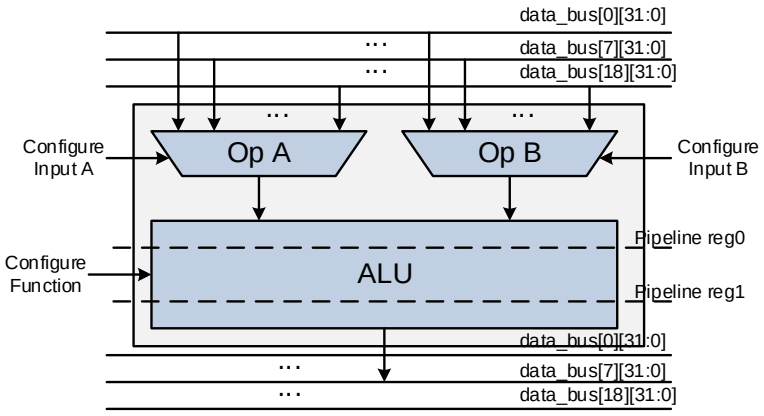

Figure 3. Functional unit detail.

Since any FU can select any FU output as one of its inputs, then the DE has a *full mesh topology*. This architecture may seem exaggerated and unnecessary, but this topology accomplishes two primary goals: (1) *programming in assembly becomes humanly possible* and (2) *the compiler design is greatly simplified*.

The assembly programmer does not need to remember or check what is connected to what. Note that everything connects 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 more easily as complicated Place&Route algorithms commonly used in CGRAs become unnecessary with a fully connected topology.

Assembly programmability is a powerful feature. It may be used to optimise critical program sections or to work around post-silicon bugs. More importantly, since CGRA compilers are still being researched and may have problems, assembly code may be used when the compiler fails.

In other approaches, such as in [4], a full mesh topology would be problematic because of the cycle by cycle reconfiguration. It would cause frequent switching of the (large) interconnect network and consequent power dissipation. In Versat, a full mesh becomes viable as reconfiguration does not happen every cycle. The interconnect consumes little power since Versat is reconfigured after a complete program loop or 2-level nested loop is executed in the DE. With a large number of FUs, a full mesh topology may result in a large circuit and limit the frequency of operation. Since Versat has only 15 interconnected FUs, its structure is still within practical limits. Our Integrated Circuit (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 170 MHz in a 130 nm process. This frequency is more than enough for many target applications. For example, in the multimedia space, most 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 multiple and parallel hardware datapaths. Multiple datapaths operating in parallel realise 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 4, three example hardware datapaths are illustrated. Datapath (a) implements a pipelined vector addition. Even though 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 vectorised version of datapath (a) to illustrate ILP+DLP. The vectors to be added spread over memories M0 and M2 so that two additions can be performed in parallel. ILP and DLP can be further exploited to produce efficient datapaths such as 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 followed by an accumulator. When the ALU is used as an accumulator, the unused data input is used for control. If the control input is zero or positive, the ALU accumulates; otherwise, the data input is passed through and registered at the output. This feature is useful when one does not wish to accumulate every element of the input sequence. In datapath (c) the accumulator's control input is forced to zero, which means it always accumulates.

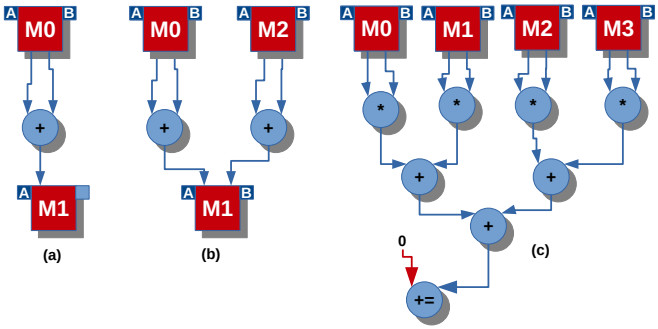

Figure 4. Examples of data engine datapaths. (**a**) Pipelined vector addition with source vector stored in a single memory; (**b**) pipelined vector addition with source vector stored in two memories; (**c**) pipelined implementation of the inner product of two vectors.

3.2.2. Address Generation

Each port of a dual-port embedded memory has one data input, one data output and one address input, as shown in Figure 5. In Versat, each port is equipped with an Address Generation Unit (AGU). An AGU can be programmed to generate the address sequence used to access data from the memory port, during the execution of a program loop in the DE. The AGU scheme is similar to the one described in [41], in the sense that both schemes use parallel and distributed AGUs. The AGUs in this work support two levels of nested loops. They can start execution with a programmable delay, so that circuit paths with different latencies can be synchronised. Each AGU can be operated independently of the other AGUs, which is instrumental for exploiting TLP.

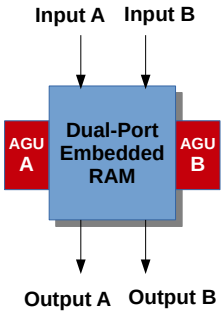

Figure 5. Dual-port embedded memory with Address Generation Units (AGUs).

Datapath (b) in Figure 4, used previously to explain DLP, can also be used to explain TLP, as follows. Suppose one block of vector elements to be added are placed in memory M0, and that AGUs M0-A, M0-B and M1-A are started (Thread 1). In parallel, one can move the next block to memory M2 and then start AGUs M2-A, M2-B and M1-B (Thread 2). The controller monitors the completion of Thread 1, restarts it with a new vector block and

proceeds to do the same for Thread 2, and so on. This process is a typical streaming scheme for vectors that vastly exceed the capacity of the Versat memories.

The AGU parameters are described in Table 1. An example address sequence is shown in Figure 6 to illustrate these parameters' use. Note that the AGU enable signal (`en`) is started with a two clock cycle delay (Delay = 2), has a period of 5 cycles (Per = 5), during which it is active for three cycles (Duty = 3). The initial value of the sequence is 10 (Start = 10 in decimal notation). Every enabled cycle the output address (`addr`) is incremented by 2 (Incr = 2); in the last cycle of a period, it is incremented by −3 (Incr+Shift = −3). This pattern repeats for Iter iterations, though the Iter parameter is not illustrated in the figure.

Table 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 converging branches of 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.

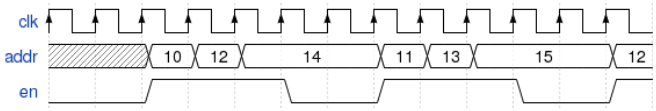

Figure 6. AGU output for *Delay* = 2, *Per* = 5, *Duty* = 3, *Start* = 10, *Incr* = 2, and *Shift* = −5.

The configuration of the input multiplexer present at each memory port determines what the port does. The following configurations are possible:

- READ: the port reads the memory; the address comes from the AGU or the other input port of the memory (external pointer).
- WRITE: the port writes the memory; the memory is also read at the same address before it is written; the Data Bus section to be written must be indicated; the address comes from the AGU or external pointer.
- DGU (Data Generation Unit): the sequence generated by the AGU is output instead of being used as the address. A DGU can be used for generating data patterns, synchronisation and control signals for the FUs.

3.2.3. ALU Functions

There are two types of ALUs—denoted Type I and Type II. The DE has 2 ALUs of type I and 4 ALUs of type II. A summary of the ALU operations is given in Table 2, where the inputs are denoted A and B, and the output is denoted Y. The ALUs have four configuration bits for the operation field and thus can support 16 different operations.

Table 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	—
Addition	Y = A + B	Y = A $\geq$ 0? Y+B: B
Multiplexer	Y = A $\geq$ 0? B: 0	Y = A $\geq$ 0? Y: B
Subtraction	Y = B $-$ A	Y = A $\geq$ 0? Y-B: B
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)	—
Count lead zeros	Y = CLZ(A)	NA
Signed maximum	Y= max(A,B)	Y = A $\geq$ 0? max(Y,B): Y
Signed minimum	Y= min(A,B)	Y = A $\geq$ 0? min(Y,B): Y
Absolute value	Y= $\lvert$A$\rvert$	NA

Type II ALUs use one of the function configuration bits to connect an internal feedback wire from the output to input A. The external port of input A becomes available for controlling the FU, as explained below. The remaining three function configuration bits are used to select eight operations from Table 2. If the feedback configuration bit is '0' then the type II ALU has the same behaviour compared to a type I ALU that selects the same function. If the feedback configuration bit is '1' then the recursive and conditional operations outlined in the 3rd column of the table apply.

The feedback operations in type II ALUs use input A for control. For example, the addition operation realises the following conditional operation—accumulate input B if input A is 0 or positive, or register input B otherwise. Another example is the minimum operation: compute the minimum value of a number sequence coming to input B if input A is 0 or positive. In this example, input A is being used to qualify the sequence elements for which the minimum should be computed. Input A can come from anywhere in the DE, but it usually comes from an AGU used to generate a control sequence.

3.2.4. Multipliers and Barrel Shifter

The multipliers take two 32-bit operands and produce a 64-bit result, of which only 32 bits are output. By configuration, it is possible to choose the upper or lower 32-bit part of the result. It is also possible to shift left the result by one position, which is useful to work in the Q1.31 fixed-point format, common in many DSP (Digital Signal Processing) algorithms.

For the barrel shifter, one operand is the word to be shifted, and the other operand is the shift size. The barrel shifter can be configured with the shift direction (left or right) and the right shift type (arithmetic or logic).

3.2.5. Functional Unit Latencies

Each FU type has a latency due to pipelining: two cycles for the ALUs, three cycles for the multipliers, one cycle for the barrel shifter and embedded memories. These individual FU latencies must be taken into account when building a datapath in the DE. The latencies

accumulate along the paths, and when two paths converge, it is necessary to make sure they have the same accumulated latency. This verification can be done using the *Delay* parameter of the AGUs, explained in Table 1.

3.2.6. Data Engine Control

The DE is controlled using its Control and Status registers. The Control Register structure is the following—bits 20 down to 2 are used for individually selecting the FUs (note there are 19 selectable units—8 memory ports and 11 other FUs.); bit 1 is used to enable the selected FUs—bit 0 is used to reset the selected FUs. The Status Register simply reports the state of the 8 AGUs in bits 7 down to 0: logic '0' indicates the AGU is running and logic '1' indicates the AGU is idle.

3.3. Configuration Module

The set of DE configuration bits is organised in configuration spaces, one for each FU. Each configuration space comprises multiple configuration fields, which are memory-mapped from the Controller point of view. The controller can change the contents of a single configuration field using a single store instruction. This implements partial reconfiguration. It is necessary to configure its FUs, one by one and field by field, to create a datapath. Sometimes it may take several cycles to configure a datapath. However, in real applications, most configurations can be obtained by incrementally changing a previously used configuration, which can be done parallel with the DE or DMA execution. Typically, a configuration can be done in a few clock cycles with a short or zero time overhead.

The Configuration Module (CM) is illustrated in Figure 7. It comprises a configuration register file, a configuration shadow register, a configuration memory and an address decoder.

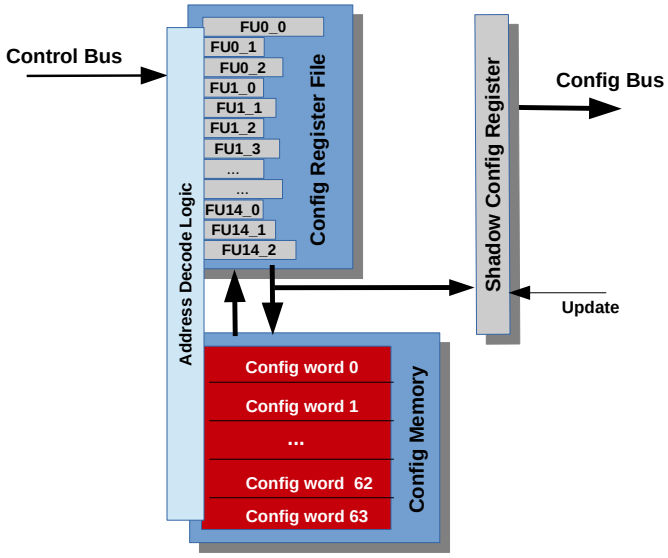

Figure 7. Configuration module.

The configuration register file has 15 configuration spaces, as many as configurable FUs in Versat. In Figure 7, FU_{ij} denotes the configuration field j of the configuration space i. As illustrated, the bit width of the configuration fields may differ. There are write-only addresses to access the configuration fields individually; read accesses to these addresses are ignored. The concatenation of all configuration bits from all configuration fields is called a configuration word. The configuration register file has 118 configuration fields, resulting in a 672-bit wide configuration word. There is also a special write address used to clear all bits in the configuration register file. Hence, zero is the reset and the default value of the configuration fields and, by design, this value selects a useful configuration. For example, the default value for the function configuration field of an ALU selects the

addition operation, which is a common operation. Building a DE configuration from the default configuration is about 40% faster than writing all its fields because many fields can be left with their default values.

The configuration shadow register holds the active configuration word of the DE, which is copied from the configuration register file whenever the Update signal is asserted by the controller (see Figure 7). Thus, the configuration register file contents can be changed while the configuration shadow register keeps the DE configured and running.

The configuration memory can hold up to 64 configuration words. There is a high likelihood that configurations in an application are reused, either as they are or with partial reconfiguration. Thus, it is useful to have a mechanism for moving configuration words between the configuration register file and the configuration memory, which works as follows. Each position of the configuration memory is memory mapped from the Controller point of view. Read access to any such position causes the configuration memory's contents to be loaded into the configuration register file; write access causes the configuration register file's contents to be stored to the configuration memory. No data is exchanged on the Control Bus during these reads or write accesses; these transactions are internal to the CM. Saving or loading configuration words takes only a single clock cycle.

3.4. Controller

Versat uses a minimal controller for reconfiguration, data transfer and simple algorithmic control. This controller is not meant to replace a more general host processor, which can run complex applications while using Versat as an accelerator. On the contrary, the Versat Controller is designed to simplify the interactions between the host processor and Versat, freeing the host from the burden of micromanaging the accelerator.

The Controller instruction set contains just 16 instructions for the following actions: (1) loads and stores; (2) basic logic and arithmetic operations; (3) branching. The controller has an accumulator architecture with a 2-stage pipeline shown in Figure 8. The architecture contains three central registers: the accumulator (RA), the address register (RB), used for indirect loads and stores, and the program counter (PC). There is only one instruction type, as illustrated in Figure 8. The controller is the master of a simple bus called the Control Bus, which can be accessed using the four self-explanatory signals shown in the figure. Register RB can be accessed as if it were a peripheral of the Control Bus.

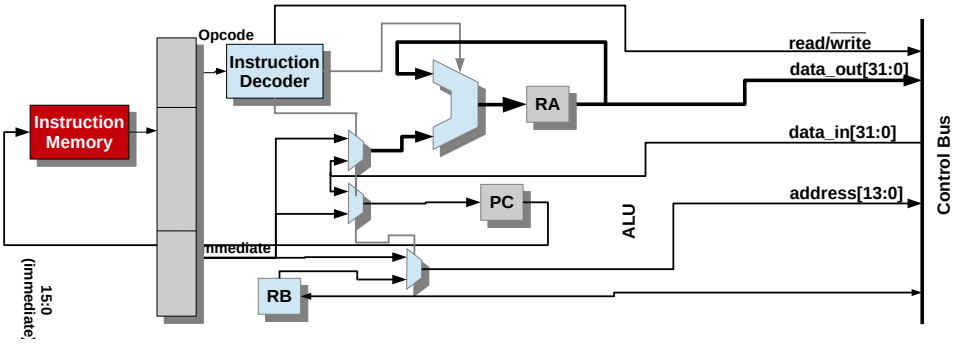

Figure 8. Controller.

The instruction set is outlined in Table 3. As usual, brackets represent memory positions. For example, (Imm) represents the contents of the memory position whose address is Imm. The PC is incremented for every instruction, except for branch instructions. For these instructions, the PC is loaded with the branch target.

Table 3. Instruction set.

Mnemonic	Description
rdw	RA = (Imm)
wrw	(Imm) = RA
rdwb	RA = (RB)
wrwb	(RB) = RA
ldi	RA = Imm
ldih	RA[31:16] = Imm
beqi	RA == 0? PC = Imm: PC + = 1; RA = RA − 1
beq	RA == 0? PC = (Imm): PC + = 1; RA = RA − 1
bneqi	RA != 0? PC = Imm: PC + = 1; RA = RA − 1
bneq	RA != 0? PC = (Imm): PC + = 1; RA = RA − 1
add	RA = RA + (Imm)
addi	RA = RA + Imm
sub	RA = RA − (Imm)
shft	RA = (Imm < 0)? RA=RA<<1: RA=RA>>1
and	RA = RA & (Imm)
xor	RA = RA ∧ (Imm)

3.5. Qualitative Comparison with Other Architectures

Versat has some distinctive features, which cannot be found in other architectures: (1) it has a small number of FUs organised in a full mesh structure; (2) it has a fully addressable configuration register combined with a configuration memory to support partial self-configuration; (3) it has a dedicated and minimal controller for reconfiguration, DMA management and simple algorithm control—no RISC [3] or VLIW [4] processors are used.

CGRAs started as 1-D structures [37], but more recently, 2-D mesh FU arrays became more common [3,4,39]. The problem with 2-D 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 [42]. Versat is an experiment on trading the FU count for 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 generally irrelevant, given the target applications' low energy budgets. A low energy consumption often imposes a frequency of operation which is well below Versat's maximum operating frequency (see Section 5).

As explained in [39], the reconfiguration time in CGRAs can easily dominate the total execution time. To tackle this problem, Versat resorts to self-generated configurations to avoid loading configurations from the external memory. These are fine and partial reconfigurations to minimise the reconfiguration time. This powerful reconfiguration mechanism contrasts with the moderate hierarchical reconfiguration scheme proposed in [39].

Since it is crucial to have reconfiguration done quickly, Versat includes a minimal 16-instruction controller, practically dedicated to reconfiguration management, and can write to the configuration register file in a single clock cycle. This controller is also used to manage data transfers and control the execution of acceleration kernels. In other architectures [3,4,39], more comprehensive processors are used, which can also run complex algorithms. However, combining complex application coding and accelerator control is difficult. Using its controller, Versat can completely take care of compute-intensive

yet straightforward kernels. Example kernels are FFT [43], DCT, Motion Estimation, and even big data algorithms such as K-Means Clustering [44]. Host processors can invoke these kernels, and they will run in parallel and to completion, without requiring any external control.

4. Programming

Versat can be programmed using a C/C++ subset, and the code can be compiled using the Versat compiler [40]. 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 section, Versat programming is illustrated using its C/C++ dialect, which is easier for explaining the basic concepts. The programming tools will not be explained as they are not the main focus of this article.

4.1. Basic Programming

A typical example program is given in this section to outline the basics of Versat programming. The program adds two memory interleaved vectors—one vector resides in the even-numbered addresses while the other vector resides in the odd-numbered addresses. The program is shown in Figure 9 and comments are added for clarity.

```
int main(){
        //initiate data transfer into Versat using DMA
        dma.config(1024, 256, 256, 1);
        dma.run();
        //clear configuration register and create new DE configuration
        de.clearConfig();
        for(j = 0; j < 128; j++)
                mem1A[j] = mem0A[j*2] + mem0B[1+j*2];
        //wait for DMA 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 9. Vector addition code.

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 with the external and internal addresses, transfer size and direction using the `dma.config()` method. 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 using the `de.clearConfig()` method. Certain language constructs are interpreted as DE configurations, and the compiler automatically generates instructions that write these configurations to the CM. An example is the case of the ensuing `for` loop. The fact that memory ports are invoked in its body triggers this interpretation. Note that the compiler does not yet support object or variable declarations. Data must be referenced using the names of the memories or registers where they are stored. When variables are supported, the issue of recognising DE configurations will need further research. It can be conjectured that data arrays in a loop are sufficient for inferring a DE configuration. In the example given, 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 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.

Once the DE is loaded and configured, it is run employing the `de.run()` function call. While it runs, the DMA is configured to transfer the vector addition to 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 (see Figure 10). The example shows a do-while loop. The controller always executes this kind of loops because the DE has no means to stop conditionally.

```
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 10. Self and partial reconfiguration code.

A 2-level nested loop follows, where the body contains only expressions involving memory ports. Therefore, this nested loop is 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 updated.

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 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. The next iteration R1 is updated; then the DE is partially reconfigured for the next run, and the process repeats all over again.

5. Results

In this section, experimental results for the Versat architecture are presented and discussed. First, integrated circuit implementation results are outlined. Second, performance and energy consumption results on simple computational kernels are given. Third, performance and energy consumption results in more complex kernels are presented. A comparison between Versat and other CGRAs is made to end this section.

5.1. Integrated Circuit Implementation Results

Versat has been designed using a UMC 130nm process. Table 4 compares Versat with a state-of-the-art embedded processor (ARM Cortex A9) and two other CGRA implementations (Morphosys [3] and ADRES [4]), in terms of the technology node (N), silicon area (A), embedded memory (M), frequency of operation (F) and power consumption (P). The Versat frequency and power results have been obtained using the Cadence IC design tools. The power figure has been obtained using the node activity rate extracted from simulating an FFT kernel.

Table 4. Integrated circuit implementation results.

Core	# FU	DW (Bytes)	N (nm)	A (mm^2)	M (kB)	F (MHz)	P (mW)
ARM [45]	—	4	40	4.6	64	800	500
Morphosys [3]	64	2	350	168	6	100	7000
ADRES [4]	16	4	90	4	64	300	91
MORA [22]	22	1	130	1.75	16	1000	—
BilRC [23]	152	2	90	11.9	16	415	—
TRANSPIRE [26]	8	1	28	0.27	32	50	—
Versat	15	4	130	5.2	46	170	132

Because the different designs use different technology nodes, it is difficult to compare the results in Table 4. The results are scaled to the 40 nm technology node to facilitate the comparisons, and presented in Table 5. The scaling is performed as explained in [46].

Table 5. IC results scaled to 40 nm.

Core	#FU	DW (Bytes)	A (mm^2)	M (kB)	F (MHz)	P (mW)	Area/#FU/DW$\times 10^3$
ARM [45]	—	4	4.6	64	800	500	—
Morphosys [3]	64	2	2.19	6	875	800	17.1
ADRES [4]	16	4	0.79	64	675	40	12.3
MORA [22]	22	1	0.19	16	3200	—	8.6
BilRC [23]	152	2	2.57	16	941	—	8.5
TRANSPIRE [26]	8	1	0.77	32	35	—	96.3
Versat	15	4	0.49	46	553	41	8.1

Different CGRAs have different configurations with different data widths and number of functional units. Therefore, we considered a metric given by the ratio between area, number of functional units and data width in bytes: (Area/#FU/DW column in Table 5).

The results show that Versat is the smallest core. Compared with the ARM processor it is 9$\times$ smaller. The ADRES architecture is about 50% larger than Versat and Morphosys is about twice bigger. MORA and BilRC are close to Versat, but Versat has multipliers in all computational functional units. The large area of TRANSPIRE is in part due to

the utilization of floating-point units, considered essential by the authors for Internet of Things applications.

Versat uses somewhat less embedded memory than the ARM Cortex A9 or ADRES cores, but its memory size can be considered typical for an embedded processor, as can be observed from the memory utilization of the other architectures. Morphosys uses a lot less memory, as it is designed to focus more on processing power and less on storage capabilities.

The frequency of operation of the ARM core can be considered low as this is a power optimised version. Other versions of this core operate at higher frequencies but have a larger area footprint and higher power consumption. Those versions are optimised for performance rather than power. Among the five CGRAs, Versat is the least optimised in terms of the working frequency. Considering the application of Versat for embedded systems, not too much effort, like pipelining and routing, has been put into achieving timing closure for a higher frequency, since we wanted to keep a low area and power. Notwithstanding, after analysing the critical paths, it became clear that there is plenty of room for optimisation, 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 on performance with an extensive array of FUs. The ADRES architecture seems well optimised for power, with its cycle by cycle and progressive reconfigurations. However, the acceleration achieved with this core is not documented in its publications [4].

Versat is therefore a very compact, flexible and power efficient CGRA architecture.

5.2. Performance and Energy Consumption Results for Simple Kernels

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 to assess the performance of the Versat architecture. Versat is connected as a peripheral of the ARM system using its AXI-4 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 AXI-4 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 results for a set of simple kernels are summarised in Table 6. These kernels use a single Versat configuration (no reconfiguration or reuse of data already in the accelerator) to get base values for performance and energy consumption. In the next section, it will be shown that with massive reconfigurations and data reuse, it becomes even more advantageous to use Versat.

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 1st and 2nd 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.

The program has been placed in the on-chip memory and the data in the external DDR3 memory device for both systems. The ARM cycles column denotes the execution cycle count for the ARM core. The Versat cycles columns give the total cycle count (All), including data transfer, processing, control and reconfiguration, and the unhidden control cycle count (Ctr), which means the control and reconfiguration cycles that do not occur in parallel with the DE or DMA. The number of FUs used (# FUs), and the code size in bytes (Size) are 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 5 for the 40 nm technology node. The speedup (SU) is the ARM/Versat ratio of execution times. In turn, the execution time is given by the cycle count divided by the respective frequency of operation. The energy ratio (ER) is the energy spent by the ARM processor divided 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. Simple kernel results.

Kernel	ARM Cycles	Versat Cycles		#FUs	Size (Bytes)	SU	ER
		All	Ctr				
`vadd`	14,726	4517	36	3	152	2.25	27.48
`iir1`	18,890	7487	26	5	220	1.74	21.27
`iir2`	24,488	10,567	26	8	332	1.60	19.54
`cdp`	25,024	6673	26	10	408	2.59	31.61

The main conclusion is that while the achievable speedups are modest, the energy gains are significant for these single configuration kernels. These results make 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 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, in this case, a single ALU is enough to hide the data transfer time for streaming the vectors. If the data were already in the DE, then more ALUs in parallel could have been used. The `iir2` and `cdp` kernels use more FUs as they require more computations per vector element.

The *Ctr* number of cycles is low for all examples, which shows that DE configuration can be accomplished almost wholly while the DMA is running. Configuration can also be done while the DE is running, as shown in the next section where runtime reconfiguration is considered. In any case, the configuration overhead is low. Given the simplicity of the examples, their code size is small in the order of hundreds of bytes. Many of these simple kernels can be placed in the 8 kB program memory and invoked when necessary.

5.3. Performance and Energy Consumption Results for Complex Kernels

In this section, two 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 or produced by themselves. In these examples, the data transfer time is less significant. The examples are a K-Means Clustering algorithm [44], widely used in big data applications, and a Fast Fourier Transform (FFT) [43], very popular in digital signal processing.

These examples are parameterisable, 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 capability would be hard to achieve with statically compiled configurations and demonstrates the strength of self-generated configurations.

The parameters that can be passed to the K-Means Clustering kernel are the numbers of data points, dimensions and centroids. The algorithm iterates over all points until it converges and finds a cluster for each point. The centroids, whose positions are updated in every iteration, represent the cluster centres. In each iteration, all points are compared to all centroids. Each point requires the DE to be reconfigured twice. Since this algorithm is applied to millions or more points in many practical situations, Versat is reconfigured millions of times. There is no reconfiguration overhead, as all reconfigurations are done while the DE is running. Figure 11 shows how the time for one iteration varies with the number of data points for a fixed number of dimensions and centroids. The results are given for the ARM core and Versat, using logarithmic scales.

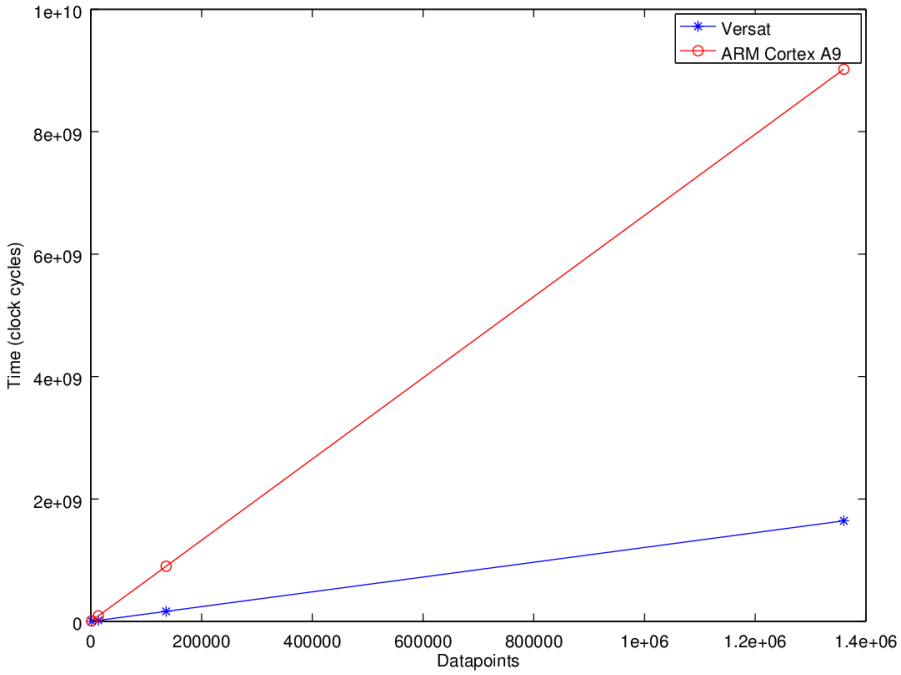

Figure 11. K-Means: iteration time vs. number of datapoints, for 30 dimensions and 34 centroids.

These results show that the execution time scales linearly with the number of data points for both systems. These numbers show a Versat/ARM average speedup of 3.8, taking into account the number of clock cycles and the cores' working frequencies.

The FFT kernel uses the Radix-2 Cooley-Tukey algorithm and can be parameterised with the following parameters—the number of data points, window size (must be a power of 2) and window overlap size. The algorithm computes the FFT for the sliding window points, advancing the window by the window size minus the overlap size. Figure 12 shows the Versat/ARM speedup as a function of the window size for 1 million datapoints and a half window (50%) overlap.

As the sliding window size reaches the Versat memories' capacity, the speedup drops, while the ARM core can still use its data cache and pre-fetch mechanism to sustain its performance. However, when the window size further increases, the ARM core reaches its internal memory limitations for streaming data and the Versat/ARM speedup increases steadily after that.

Results for two particular instances of the K-Means and FFT algorithms are detailed in Table 7. The K-Means result has been obtained for one iteration over a dataset of 1.36×10^6 points of 30 dimensions for 34 centroids. The FFT result pertains a 16384-point window size with a 50% overlap over a dataset of 10^6 complex points.

Table 7. Complex kernel results.

Kernel	ARM Cycles	Versat Cycles	#FUs	Size (Bytes)	SU	ER
`kmeans`	9.02 G	1.64 G	9	2764	3.8	46.28
`fft`	1.28 G	34.50 M	8.5	3492	25.57	311.79

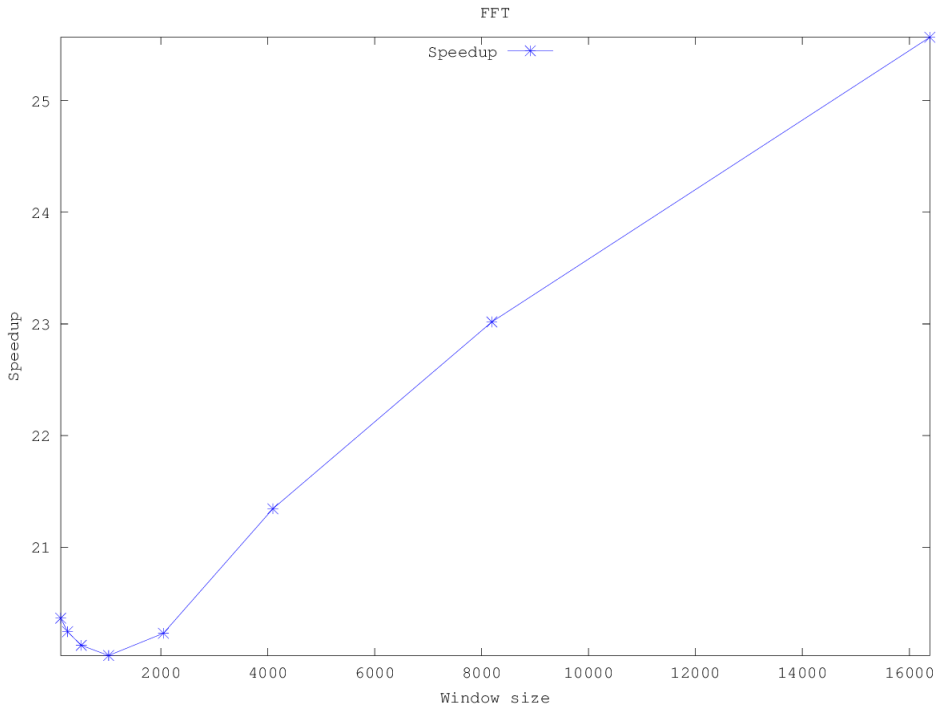

Figure 12. Fast Fourier Transform (FFT): speedup vs. window size.

These results show that the speedup and energy efficiency is more sizeable with more complex kernels than with the simpler kernels in Section 5.2. This improvement has to do with the number of operations done in parallel in the DE and 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 two DE configurations; one configuration performs datapoint classification, and the other configuration updates the centroid positions. As for the FFT, after fetching a datapoint chunk, several configurations are applied corresponding to the several FFT stages. The FFT datapaths also expose a higher ILP in these computations. Hence, the speedup and energy efficiency of the FFT is much higher when compared to the K-Means algorithm. Both these algorithms illustrate the power of using the Versat CGRA compared to the ARM processor.

5.4. Performance Comparison with Other CGRAs

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

Notwithstanding, Versat can be compared with Morphosys for the FFT kernel since it is reported in [47] that the processing time for a 1024-point FFT is 2613 cycles. Compared with the 12,115 cycles taken by Versat, Morphosys is 4.6× faster on this kernel at the same operating frequency. This fact is not surprising, since Morphosys has 64 FUs compared to only 11 FUs in Versat; Morphosys is 4.5× the size of Versat according to Table 5. Morphosys uses 2.66× more energy than Versat for the FFT kernel, as shown by the frequency and power figures in this table.

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 are never given in absolute terms.

6. Conclusions

In this paper, a novel CGRA architecture called Versat has been presented as a case study of CGRAs architectures. A detailed description of the CGRA architecture has been presented for a better understanding of the subject. Versat is a minimal CGRA with four embedded memories and eleven FUs interconnected by a full mesh. It features a small 16-instruction controller, used primarily for runtime self-generated partial reconfiguration. The controller is also responsible for running simple algorithms and managing data transfers using an embedded DMA module.

The controller is not a replacement for a host processor capable of executing arbitrarily complex algorithms. Versat is an accelerator for host processors. Nevertheless, with its minimal controller, Versat can execute high-level procedures, freeing the host from micromanaging the CGRA. Hosts can interact with Versat through an API containing a set of useful kernels.

Versat has fewer functional units compared to other CGRAs. However, its fully interconnected topology provides unprecedented programmability for supporting various algorithms. Its structure is so simple that it can be programmed in assembly language. To the best of our knowledge, Versat is the only CGRA that supports assembly-level programming, which is key to mitigating development risks in a production environment. Versat can also be programmed in a C/C++ dialect for greater productivity. Despite its small size, it can accelerate computations and improve the energy efficiency by orders of magnitude compared to general-purpose processors.

A Versat program generates myriads of configurations on the fly, saving configuration storage space and memory bandwidth. Other CGRAs use pre-compiled configurations, which is much more limiting. To reconfigure quickly, Versat supports partial reconfiguration. Configurations are stored in a register file, where each register stores an individually accessible configuration field. Additionally, a configuration memory stores the most frequently used complete configurations.

The results of a VLSI implementation show that Versat is competitive in terms of silicon area (1.62× smaller than ADRES [4]), and energy consumption (2.66× lower compared with Morphosys [3]). Performance results show that Versat can accelerate computational kernels by up to one order of magnitude and save up to two orders of magnitude in energy compared to an ARM Cortex A9 system. For example, for an FFT kernel, Versat is 25.57 times faster and consumes 311.79 times lower energy than the said ARM system.

Author Contributions: Investigation, J.D.L., M.P.V., R.P.D., J.T.d.S. and H.C.N.; methodology, J.D.L., M.P.V., R.P.D., J.T.d.S. and H.C.N.; validation M.P.V., J.T.d.S. and H.C.N.; writing—review and editing, J.D.L., M.P.V., R.P.D., J.T.d.S. and H.C.N. All authors have read and agreed to the published version of the manuscript.

Funding: This work was supported by national funds through Fundação para a Ciência e a Tecnologia (FCT) with reference UIDB/50021/2020.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. De Sutter, B.; Raghavan, P.; Lambrechts, A. Coarse-Grained Reconfigurable Array Architectures. In *Handbook of Signal Processing Systems*; Bhattacharyya, S.S., Deprettere, E.F., Leupers, R., Takala, J., Eds.; Springer: Cham, Switzerland, 2010; pp. 449–484.
2. Qiu, J.; Wang, J.; Yao, S.; Guo, K.; Li, B.; Zhou, E.; Yu, J.; Tang, T.; Xu, N.; Song, S.; et al. Going Deeper with Embedded FPGA Platform for Convolutional Neural Network. In Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA'16), Monterey, CA, USA, 22 February 2016; ACM: New York, NY, USA, 2016; pp. 26–35. [CrossRef]
3. Hau Lee, M.; Singh, H.; Lu, G.; Bagherzadeh, N.; Kurdahi, F.J. Design and Implementation of the MorphoSys Reconfigurable Computing Processor. *J. Vlsi Signal-Process. Syst. Signal Image Video Technol.* **2000**, *24*, 147–164. [CrossRef]
4. Mei, B.; Lambrechts, A.; Mignolet, J.Y.; Verkest, D.; Lauwereins, R. Architecture exploration for a reconfigurable architecture template. *Des. Test Comput.* **2005**, *22*, 90–101. [CrossRef]

5. Estrin, G. Organization of Computer Systems: The Fixed plus Variable Structure Computer. In *Papers Presented at the May 3–5. 1960, Western Joint IRE-AIEE-ACM Computer Conference*; IRE-AIEE-ACM'60 (Western); Association for Computing Machinery: New York, NY, USA, 1960; pp. 33–40. [CrossRef]
6. Hartenstein, R.W.; Hirschbiel, A.G.; Riedmuller, M.; Schmidt, K.; Weber, M. A novel ASIC design approach based on a new machine paradigm. *IEEE J. Solid-State Circuits* **1991**, *26*, 975–989. [CrossRef]
7. Chen, D.C.; Rabaey, J.M. A reconfigurable multiprocessor IC for rapid prototyping of algorithmic-specific high-speed DSP data paths. *IEEE J. Solid-State Circuits* **1992**, *27*, 1895–1904. [CrossRef]
8. Tehre, V.; Kshirsagar, R. Article: Survey on Coarse Grained Reconfigurable Architectures. *Int. J. Comput. Appl.* **2012**, *48*, 1–7.
9. Wijtvliet, M.; Waeijen, L.; Corporaal, H. Coarse grained reconfigurable architectures in the past 25 years: Overview and classification. In Proceedings of the 2016 International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation (SAMOS), Agios Konstantinos, Greece, 17–21 July 2016; pp. 235–244. [CrossRef]
10. Liu, L.; Zhu, J.; Li, Z.; Lu, Y.; Deng, Y.; Han, J.; Yin, S.; Wei, S. A Survey of Coarse-Grained Reconfigurable Architecture and Design: Taxonomy, Challenges, and Applications. *ACM Comput. Surv.* **2019**, *52*. [CrossRef]
11. Podobas, A.; Sano, K.; Matsuoka, S. A Survey on Coarse-Grained Reconfigurable Architectures From a Performance Perspective. *IEEE Access* **2020**, *8*, 146719–146743. [CrossRef]
12. Hauser, J.R.; Wawrzynek, J. Garp: A MIPS processor with a reconfigurable coprocessor. In Proceedings of the 5th Annual IEEE Symposium on Field-Programmable Custom Computing Machines Cat. No.97TB100186, Napa Valley, CA, USA, 16–18 April 1997; pp. 12–21. [CrossRef]
13. Marshall, A.; Stansfield, T.; Kostarnov, I.; Vuillemin, J.; Hutchings, B. A Reconfigurable Arithmetic Array for Multimedia Applications. In Proceedings of the 1999 ACM/SIGDA Seventh International Symposium on Field Programmable Gate Arrays (FPGA '99), Monterey, CA, USA, 22 February 1999; Association for Computing Machinery: New York, NY, USA, 1999; pp. 135–143. [CrossRef]
14. Agarwal, A.; Amarasinghe, S.; Barua, R.; Frank, M.; Lee, W.; Sarkar, V.; Srikrishna, D.; Taylor, M. The Raw Compiler Project. In Proceedings of the Second SUIF Compiler Workshop, Stanford, CA, USA, 21–23 August 1997, pp. 21–23.
15. Hartenstein, R.W.; Kress, R.; Reinig, H. A new FPGA architecture for word-oriented datapaths. In *Field-Programmable Logic Architectures, Synthesis and Applications*; Hartenstein, R.W., Servít, M.Z., Eds.; Springer: Berlin/Heidelberg, Germany, 1994; pp. 144–155.
16. Goldstein, S.C.; Schmit, H.; Budiu, M.; Cadambi, S.; Moe, M.; Taylor, R.R. PipeRench: A reconfigurable architecture and compiler. *Computer* **2000**, *33*, 70–77. [CrossRef]
17. De Hon, M. MATRIX: A reconfigurable computing architecture with configurable instruction distribution and deployable resources. In Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines, Napa Valley, CA, USA, 17–19 April 1996; pp. 157–166. [CrossRef]
18. Miyamori, T.; Olukotun, K. REMARC: Reconfigurable Multimedia Array Coprocessor. IEICE Transactions on Information and Systems. Volume E82-D, 1998; pp. 389–397. Print ISSN: 0916-8532. Available online: https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.64.6917&rep=rep1&type=pdf (accessed on 5 March 2021).
19. Lu, G.; Singh, H.; Lee, M.H.; Bagherzadeh, N.; Kurdahi, F.; Filho, E.M.C. The MorphoSys Parallel Reconfigurable System. In *Euro-Par'99 Parallel Processing*; Amestoy, P., Berger, P., Daydé, M., Ruiz, D., Duff, I., Frayssé, V., Giraud, L., Eds.; Springer: Berlin/Heidelberg, Germany, 1999; pp. 727–734.
20. Mei, B.; Vernalde, S.; Verkest, D.; De Man, H.; Lauwereins, R. ADRES: An Architecture with Tightly Coupled VLIW Processor and Coarse-Grained Reconfigurable Matrix. In *Field Programmable Logic and Application*; Cheung, P., Constantinides, G.A., Eds.; Springer: Berlin/Heidelberg, Germany, 2003; pp. 61–70.
21. Burger, D.; Keckler, S.W.; McKinley, K.S.; Dahlin, M.; John, L.K.; Lin, C.; Moore, C.R.; Burrill, J.; McDonald, R.G.; Yoder, W. Scaling to the end of silicon with EDGE architectures. *Computer* **2004**, *37*, 44–55. [CrossRef]
22. Chalamalasetti, S.R.; Purohit, S.; Margala, M.; Vanderbauwhede, W. MORA—An Architecture and Programming Model for a Resource Efficient Coarse Grained Reconfigurable Processor. In Proceedings of the 2009 NASA/ESA Conference on Adaptive Hardware and Systems, San Francisco, CA, USA, 29 July–1 August 2009; pp. 389–396. [CrossRef]
23. Atak, O.; Atalar, A. BilRC: An Execution Triggered Coarse Grained Reconfigurable Architecture. *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.* **2013**, *21*, 1285–1298. [CrossRef]
24. Lee, D.; Jo, M.; Han, K.; Choi, K. FloRA: Coarse-grained reconfigurable architecture with floating-point operation capability. In Proceedings of the 2009 International Conference on Field-Programmable Technology, Sydney, NSW, Australia, 9–11 December 2009; pp. 376–379. [CrossRef]
25. Feng, F.; Li, L.; Wang, K.; Han, F.; Zhang, B.; He, G. Floating-point operation based reconfigurable architecture for radar processing. *IEICE Electron. Express* **2016**, *13*, 20160893. [CrossRef]
26. Prasad, R.; Das, S.; Martin, K.J.M.; Tagliavini, G.; Coussy, P.; Benini, L.; Rossi, D. TRANSPIRE: An Energy-Efficient TRANSprecision Floating-Point Programmable ArchItectuRE. In Proceedings of the 23rd Conference on Design, Automation and Test in Europe (DATE'20), Grenoble, France, 9–13 March 2020; pp. 1067–1072.
27. Wolf, D.L.; Jung, L.J.; Ruschke, T.; Li, C.; Hochberger, C. AMIDAR Project: Lessons Learned in 15 Years of Researching Adaptive Processors. In Proceedings of the 2018 13th International Symposium on Reconfigurable Communication-Centric Systems-on-Chip (ReCoSoC), Lille, France, 9–11 July 2018; pp. 1–8. [CrossRef]

28. Dadu, V.; Weng, J.; Liu, S.; Nowatzki, T. Towards General Purpose Acceleration by Exploiting Common Data-Dependence Forms. In Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'52), Columbus, OH, USA, 12–16 October 2019; pp. 924–939. [CrossRef]
29. Sato, T. DAPDNA-2 a dynamically reconfigurable processor with 376 32-bit processing elements. In Proceedings of the IEEE Hot Chips XVII Symposium (HCS), Stanford, CA, USA, 14–16 August 2005; pp. 1–24.
30. Zhu, M.; Liu, L.; Yin, S.; Wang, Y.; Wang, W.; Wei, S. A reconfigurable multi-processor SoC for media applications. In Proceedings of the 2010 IEEE International Symposium on Circuits and Systems, Paris, France, 30 May–2 June 2010; pp. 2011–2014. [CrossRef]
31. Das, S.; Sivanandan, N.; Madhu, K.T.; Nandy, S.K.; Narayan, R. RHyMe: REDEFINE Hyper Cell Multicore for Accelerating HPC Kernels. In Proceedings of the 2016 29th International Conference on VLSI Design and 2016 15th International Conference on Embedded Systems (VLSID), Kolkata, India, 4–8 January 2016; pp. 601–602. [CrossRef]
32. Prabhakar, R.; Zhang, Y.; Koeplinger, D.; Feldman, M.; Zhao, T.; Hadjis, S.; Pedram, A.; Kozyrakis, C.; Olukotun, K. Plasticine: A reconfigurable architecture for parallel patterns. In Proceedings of the 2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA), Toronto, ON, Canada, 24–28 June 2017; pp. 389–402. [CrossRef]
33. Käsgen, P.S.; Weinhardt, M.; Hochberger, C. A Coarse-Grained Reconfigurable Array for High-Performance Computing Applications. In Proceedings of the 2018 International Conference on ReConFigurable Computing and FPGAs (ReConFig), Cancun, Mexico, 3–5 December 2018; pp. 1–4. [CrossRef]
34. Podobas, A.; Sano, K.; Matsuoka, S. A Template-based Framework for Exploring Coarse-Grained Reconfigurable Architectures. In Proceedings of the 2020 IEEE 31st International Conference on Application-Specific Systems, Architectures and Processors (ASAP), Manchester, UK, 6–8 July 2020; pp. 1–8. [CrossRef]
35. De Sousa, J.; Martins, V.; Lourenco, N.; Santos, A.; do Rosario Ribeiro, N. Reconfigurable Coprocessor Architecture Template for Nested Loops and Programming Tool. U.S. Patent 8,276,120, 25 September 2012.
36. Hartenstein, R.; Herz, M.; Hoffmann, T.; Nageldinger, U. Mapping Applications onto Reconfigurable KressArrays. In *Field Programmable Logic and Applications*; Lecture Notes in Computer Science; Lysaght, P., Irvine, J., Hartenstein, R., Eds.; Springer: Berlin/Heidelberg, Germany, 1999; Volume 1673, pp. 385–390.
37. Ebeling, C.; Cronquist, D.C.; Franklin, P. RaPiD—Reconfigurable Pipelined Datapath. In *Field-Programmable Logic Smart Applications, New Paradigms and Compilers*; Hartenstein R.W., Glesner M., Eds.; Springer: Darmstadt, Germany, 1996; Volume 1142, pp. 126–135.
38. Baumgarte, V.; Ehlers, G.; May, F.; Nückel, A.; Vorbach, M.; Weinhardt, M. PACT XPP—A Self-Reconfigurable Data Processing Architecture. *J. Supercomput.* **2003**, *26*, 167–184. [CrossRef]
39. Liu, L.; Wang, D.; Zhu, M.; Wang, Y.; Yin, S.; Cao, P.; Yang, J.; Wei, S. An Energy-Efficient Coarse-Grained Reconfigurable Processing Unit for Multiple-Standard Video Decoding. *IEEE Trans. Multimed.* **2015**, *17*, 1706–1720. [CrossRef]
40. Santiago, R.; de Sousa, J.T.; Lopes, J.D. Compiler for the Versat Architecture. In Proceedings of the XIII Jornadas de Sistemas Reconfiguráveis, Aveiro, Portugal, 30–31 January 2017; pp. 41–48.
41. Farahini, N.; Hemani, A.; Sohofi, H.; Jafri, S.M.; Tajammul, M.A.; Paul, K. Parallel Distributed Scalable Runtime Address Generation Scheme for a Coarse Grain Reconfigurable Computation and Storage Fabric. *Microprocess. Microsyst.* **2014**, *38*, 788–802. [CrossRef]
42. Liu, D.; Yin, S.; Liu, L.; Wei, S. Polyhedral model based mapping optimization of loop nests for CGRAs. In Proceedings of the 2013 50th ACM/EDAC/IEEE Design Automation Conference (DAC), Austin, TX, USA, 29 May–7 June 2013; pp. 1–8.
43. Lopes, J.D.; de Sousa, J.T. Fast Fourier Transform on the Versat CGRA. In Proceedings of the Jornadas Sarteco, Malaga, Spain, 19–22 September 2017; pp. 1–8.
44. Lopes, J.D.; de Sousa, J.T.; Neto, H. K-Means Clustering on CGRA. In Proceedings of the 27th International Conference on Field-Programmable Logic and Applications, New Paradigms and Compilers (FPL 2017), Ghent, Belgium, 4–8 September 2017; pp. 1–4.
45. Wang, W.; Dey, T. A Survey on ARM Cortex A Processors. Available online: http://www.cs.virginia.edu/skadron/cs8535s11/armcortex.pdf (accessed on 16 April 2016).
46. Borkar, S. Design challenges of technology scaling. *IEEE Micro* **1999**, *19*, 23–29. [CrossRef]
47. Kamalizad, A.H.; Pan, C.; Bagherzadeh, N. Fast parallel FFT on a reconfigurable computation platform. In Proceedings of the 15th Symposium on Computer Architecture and High Performance Computing, Sao Paulo, Brazil, 12 November 2003; pp. 254–259. [CrossRef]