NATIONAL UNIVERSITY OF SCIENCE AND TECHNOLOGY POLITEHNICA OF BUCHAREST

Doctoral School of Electronics, Telecommunications and Information Technology

Decision no. 126 from 09-11-2023

PhD Thesis Summary

Eng. Mihai ANTONESCU

Algorithmical and Architectural Improvements for a Map-Reduce Hardware Accelerator

Optimizări algoritmice şi arhitecturale pentru un accelerator hardware de tip Map-Reduce

---

THESIS COMMITTEE

Prof. Dr. Ing. Gheorghe Brezeanu National University of Science and Technology Politehnica of Bucharest	President
Prof. Dr. Ing. Gheorghe Ştefan National University of Science and Technology Politehnica of Bucharest	PhD Supervisor
Prof. Dr. Ing. Corneliu Burileanu National University of Science and Technology Politehnica of Bucharest	Reviewer
Prof. Dr. Ing. Aurel-Ştefan Gontean Politehnica University Timişoara	Reviewer
Prof. Dr. Ing. Dan Nicula Transilvania University of Braşov	Reviewer

BUCHAREST 2024

Table of contents

Chapter 1
Introduction

1.1 Presentation of the field of parallel computing

The field of parallel computing comprises nowadays the majority of computation. Single core machines have been relegated to a secondary position, being used in embedded computations where computing power is less important than other considerations. Within this field, over the decades, a shift in paradigms and approaches has been observed. Most current parallel solutions for intense computation fall into two main categories: off-the-shelf machines repurposed for the application in question and application specific circuits (implemented as ASIC or in FPGA). These machines are then connected to a host computer forming a heterogeneous system.

There is no one-size-fits-all solution, each solution being uniquely affected by the combination of memory (in terms of both size and speed), single core power and multi/many core scalability.

Current generation multi-core workstations are fundamentally derived from single core chips, grouped together in a specific geometry. These solutions offer high single core performance but cannot scale well into the thousands of cores range. They are usually power hungry, with recent advances tackling this problem by combining performance and economy cores on the same chip. A recent trend for these machines sees adding AMX (Advanced Matrix eXtensions) in addition the already solidified AVX (Advanced Vector eXtensions) hardware.

The most commonly used acceleration solution today, consists of GPUs used in the context of general computing. As the name suggests, being a primarily graphics-oriented device, it is not naturally suited to general purpose computation. However, given the ease of access to these devices and the efforts put forth by Nvidia and AMD, big improvements have been made into using this technology for various computations. Given the ubiquitousness of machine learning in today's computing landscape, current generation GPUs have added dedicated hardware blocks (tensor cores) for the most commonly used functions. Tensor cores have become a standard of modern GPUs, Nvidia offering tutorials and tips on how to best use them [1], being evaluated from multiple perspectives such as performance [2] [3] and numerical behaviour [2] [4]. Using tensor cores improves energy efficiency in AI applications (for which they were designed) and potentially for other use cases (for example [5]), if code can be written in such a fashion as to utilize them. The equivalent technology used by AMD is called Matrix Core.

As the default acceleration solution, GPUs have impacted all computational domains. The scientific community has treated this subject from multiple points of

view: actual computation, memory management and transfers, data embedding and encoding. Heterogeneous computing techniques can be seen at [6] and optimization techniques for GPUs are detailed at [7].

When even greater performance is desired, FPGA implemented hardware structures can be used for acceleration of custom algorithms or specific computationally intensive section. A step beyond lies acceleration using custom ASICs, a good example of this approach being the TPU (Tensor Processing Unit).

1.2 Scope of this thesis

This thesis will focus on improving the "Connex" general-purpose many-core accelerator and the development of a new parallel accelerator system, targeting FPGA implementation. This accelerator is designed to be connected as tightly as possible to a host computer/processor in order to offload data intensive tasks from it.

In order to validate the various architectural improvements, algorithms that make use of these improvements have been written and tested.

Architectural improvements will include replacing the Reduce network with a Scan-Permute-Pack network, making the accelerator fully parametrizable and reconfigurable, IO interface generalizations, adding floating point support and others.

The three main algorithms developed for this machine will be the AES, the Transpose algorithm for square matrices and the Fast Fourier Transform (FFT) in different configurations.

1.3 Content of this thesis

This thesis details the work I have contributed on improving a custom many-core Map-Reduce parallel architecture. This previous work is detailed in Chapter 2.

Chapter 3 explores the implementation of the Scan-Permute-Pack network that was added to encompass the Reduce network and offer new functionalities.

Chapter 4 explains the implementation of parametrization and reconfigurability as used in this architecture.

Chapter 5 is concerned with using this reconfigurability for IO improvement.

Chapter 6 and Chapter 7 are concerned with improving computational power inside each cell, specifically by adding a stack mechanism and floating-point support.

Smaller miscellaneous hardware improvements are detailed in Chapter 8 and changes to the ISA are detailed in Chapter 9. These changes range from changing the ISA format to adding new instructions.

Synthesis results are presented and discussed in Chapter 10.

Chapter 11, Chapter 12 and Chapter 13 are concerned with application-level improvements: AES, computing matrix transpose and computing FFTs.

A second set of miscellaneous improvements are presented in Chapter 14, this time focussed on testing and system software.

Chapter 15 presents conclusions together with future work.

Chapter 2
Previous Form of the Map-Reduce Accelerator

2.1 Theoretical considerations

Dissimilarly to serial machines, parallel computing hardware has had a completely different birth, childhood and maturity in the present days. Serial machines followed a normal development process, comprised of the following stages: (1) computational model - the Turing Machine [8], (2) abstract model - Harvard/Von Neumann model, (3) market growth and large-scale manufacturing, (4) architecture - x86, ARM, etc.

On the other hand, parallel machines followed a completely different track, starting life as ad-hoc groupings of serial machines. Abstract machine models were eventually developed but the underlying computational model for parallel computing was not given proper attention. There is however a computational model that fits the parallel approach published at around the same time as Turing's, the Kleene model of partial recursive functions [9]. Based on this model, the core architecture of the Connex accelerator was defined and implemented, as described in [10], [11], [12].

The computing structure is defined by the composition rule from the mathematical model and as shown in the above papers, the Map-Reduce abstract model is born.

2.2 Accelerator core

Given the previously mentioned abstract model, the accelerator architecture is based around these two types of operations.

- Map operations that take a vector of input data and output a vector, performing the same operation on all the data.
- Reduce operations that take a vector of input data and output a scalar.

Map operations are performed in an array of simple, cellular like, computing machines, collectively named "Array". Reduce operations are performed through a log-depth tree structure, a network of very simple cells, collectively called "Reduce". The Reduce network is fed data and commands (the function it is to perform) by the Array. In order to control operations in the Array, a third structure is required, the "Controller". The resulting basic structure is presented below in Figure 2.1.

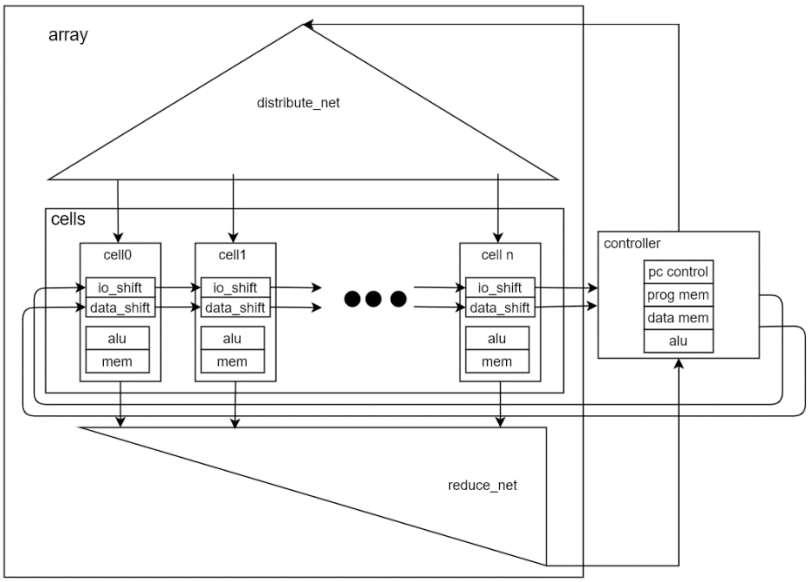

Figure 2.1 *Original Connex accelerator core structure (adapted from [12])*

2.3 Instruction Set Architecture (ISA)

One complete instruction (32b) for this machine is comprised of two normal Assembler (ASM)-like instructions (16b): one that is executed in the Controller and one that is executed in each active cell of the Array.

Each instruction is made of three components:

- OPCODE (5b) - used to select the desired operation;
- OPERAND (3b) - used to select the operand that is fed into the ALU alongside the accumulator;
- VALUE (8b) - scalar value that can be interpreted as the operand value.

A more detailed description of all operations can be found in [13], chapter 2.1.3.

2.4 Accelerator system - hardware

The accelerator system was also developed at [13]. It is comprised of the following main structures: Accelerator, ARM Cortex-A9 (Processing System), used as a HOST and a DMA engine.

2.5 Accelerator system - software

In order for this hardware system to be used, an appropriate software stack is needed. My colleagues developed a version of this software stack based on the PYNQ Linux kernel available at [14]. The software stack was made up of: Hardware interface, Linux Kernel, PYNQ system, Jupyter Notebook and Python environment.

Using this setup, linear algebra operations were performed on the accelerator. Data was generated inside the Python environment, sent to the accelerator (together with commands), computed inside the accelerator and finally extracted and displayed.

Chapter 3
Architectural Improvement: Scan-Permute-Reduce Network

This chapter describes the creation, implementation, testing and integration of the multifunction scan circuit described in [16] and expanded upon in [17].

3.1 Scan and Reduce

Reduce operations take an input vector and output a single value. On the other hand, scan operations take an input vector and output another vector.

Scan operations perform the desired operation at each step and output the partial result. As such, they are also known as Prefix operations. The final output of the scan operation is in fact the result outputted by the reduce operation.

3.2 The Beneš-Wacksman permutation network

The original circuit structure I considered for scan type operations was that of the permutation circuit developed by Vaclav E. Benes [18] and optimized by Abraham Waksman [19]. This circuit allows all permutations of its inputs to be performed, given the appropriate control bits. An example of a 16-input permutation network is shown in Figure 3.1. Coloured in red are unnecessary cells (as shown by Abraham Waksman in [19]) which will be present as simple pipeline registers without the multiplexers that enable the permutation function. Coloured in green are the input and output layers and in blue and yellow the two subnetworks, as per the recursive definition. Each permutation cell is made up of two multiplexers, receiving both of the input operands, each outputting one of the two data outputs.

3.3 Implemented functions

The following functions have been implemented in the final network: reduce add, reduce min, reduce max, reduce and bitwise, reduce or bitwise, reduce xor bitwise, scan add prefix, scan xor bitwise prefix, permute, pack.

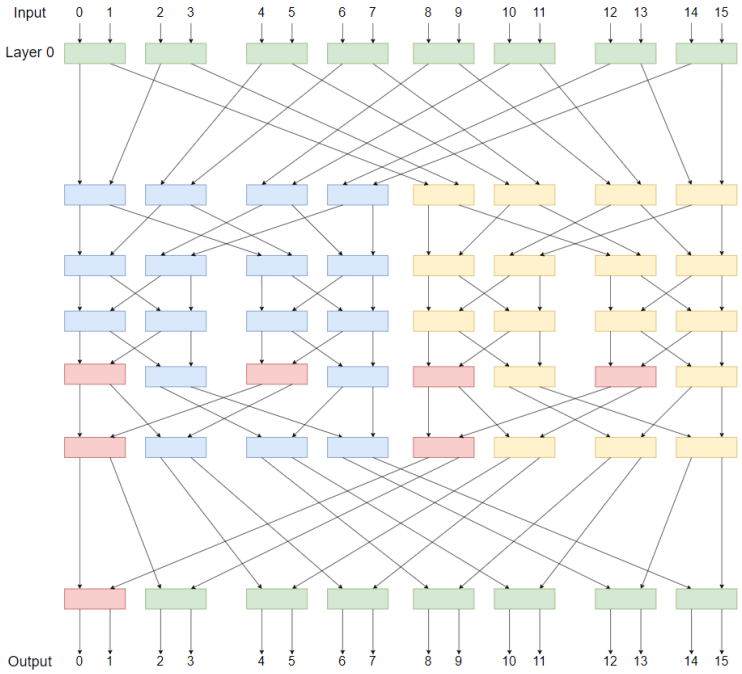

Figure 3.1 *Benes-Wacksman permutation circuit - 16 input example*

3.4 Multi-Function Scan-Permute-Pack-Reduce circuit: hardware description

Based on the structure and form provided by the Benes-Wacksman permutation circuit, a new circuit was developed in order to provide multiple other operations. This multi-function scan circuit was published in [16] and expanded upon in [17]. While the networks form was kept unchanged, its cell's structure was severely modified to accommodate new functionality. The internal structure of the multifunctional cell is shown in Figure 3.2.

Depending on each cell's position and on the functions the network is asked to perform, its internal structure can vary significantly. This variation manifests itself in the complexity of the control signals generator circuit and in the presence or absence of some hardware blocks.

The two data movement operations implemented, Permute and Pack, require control bits that act as selections for the two internal multiplexers in each cell. Each cell requires one control bit in order to function properly. Since the network is pipelined, and in order to reduce physical wiring complexity, the control bits are passed together with the data they are supposed to guide, each cell "consuming" one control bit.

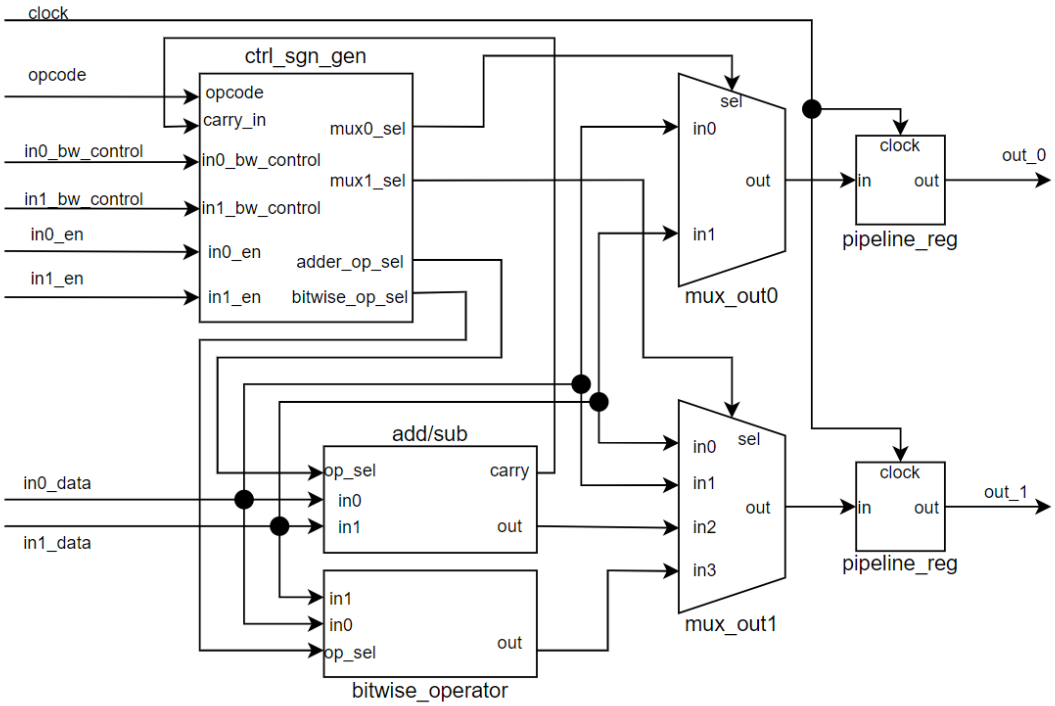

Figure 3.2 *Multifunctional cell internal structure example*

3.5 Sequential version

A sequential version of this network was also developed and tested, using only one layer of permutation/computation cells and two multi-input multiplexers for each cell. It is to be used when space is more important that performance. In the final accelerator this version was discarded

3.6 Control bits generation

The generation of all needed control bits is a complex operation. It is split into two stages: generation of control bits for each cell and assembly of the control bits going backwards through the network (based on the path the bits will take) in order to obtain control words. Code was written in C++ to perform this function.

3.7 Testing

In order to prove that this circuit correctly performs the proposed operations, a SystemVerilog testbench was designed and written. This testbench generates input data for each network input, feeds this input to both the hardware network and a non-synthesizable golden model and finally compares the two outputs in order to validate functionality. Separate test functions for each operation were developed. Transitions between functions were also taken into account. The circuit passed all proposed tests.

3.8 Synthesis results

Synthesis results for the multifunction network (Table 3.1) are presented below. Synthesis was done targeting the ZYNQ7020 SoC found on a PYNQ-Z2 board.

Table 3.1 *Multifunctional network synthesis result*

Nr inputs	Nr cells	LUTs	REGs	Average LUTs/cell	Average REGs/cell
8	20	1240	1459	62	73
16	56	3813	3921	68	70
32	144	10208	10368	71	72
64	352	25894	26252	74	75
128	832	63389	64146	76	77
256	1920	151180	152627	79	79

3.9 Integration into the Map-Reduce accelerator. The Map-Scan-Permute-Pack-Reduce architecture

This network was developed having the accelerator described in Chapter 2 in mind, specifically to come as a complement (or an improvement over) to the Reduce network it already contains. As such, the reduce output can connect directly into the Controller and all scan outputs can connect back into the Map/Cells section.

3.10 Conclusions

In this chapter, the development of our multi-function (Scan, Permute, Pack) circuit was presented. This process involved conception, starting from the Benes-Waksman permutation network, deciding on and implementing different functionalities, testing each of these functions and finally integrating this design in the complete accelerator structure.

This network can now perform multiple operations (as described in Chapter 3.3), including all operations that the previous Reduce network could perform.

The use of macros and parametrization helps this circuit achieve a high degree of flexibility, which in turn helped its integration in the accelerator structure.

For the Permute operation, there is a downside, the fact that computing the control bits is a complicated process that cannot be trivially accelerated.

A second downside, although much smaller, is in regard to the Pack operation, which requires a Prefix Add on the enable bits present in each cell, before it can be run.

Similarly, other 2-input variable, associative functions that support the inverse operation can be implemented using this network shape, although circuits may not be the most efficient implementation for all possible functions.

Chapter 4
Architectural Improvement: Parametrization and Reconfigurability

4.1 Rewriting and parametrization

The way in which the accelerator was initially written, albeit functional, was not conformant with current good practices for software/hardware development. As such, the entire code base was rewritten in an easier to understand and follow format.

With this occasion, an opportunity for advanced parametrization and reconfigurability has arisen. We use the name parameters to refer to those macros that can be changed and affect the underlying structure of the machine. Macros related to numbering (for example ISA macros that number each instruction opcode) and macros that can be derived from other macros are not considered parameters.

The most important parameters present are those pertaining to the number of cells, the amount of memory present in each cell and if specific hardware features are present or not.

4.2 Hardware reconfigurability and conditional synthesis

Many hardware blocks were integrated into the accelerator in such a way as to be conditionally synthesised. Depending on a multitude of factors, the code base offers a trade-off between advanced hardware capabilities on one side and circuit size and complexity on the other.

We view reconfigurability as one of the most important concepts brought about by the explosion in FPGA size increase and adoption. On our architecture, this is very important as it allows targeting hardware for specific applications while still being able to bear the name "general purpose". In the context of an application, this accelerator can obtain good results while not being fully reconfigurable (switching accelerators mid application would be slow) and not using the same amount of hardware resources as multiple dedicated accelerators for each subtask. In addition to this, hardware resources not used for specific features can in turn be converted into more array cells that will increase the degree of parallelism available.

Chapter 5
Architectural Improvement: IO Size Generalization

5.1 Previous work

Both input and output interfaces are on 64 bits with additional control signals connecting FIFOs and the propagation control mechanism. Since each cell requires 32 bits of data, 2 cells can be serviced at any one time, hence creating the dual-cell. A further improvement (presented in [13]), created the "quad-cell", in order to have a total throughput of 64 bits per cycle. Figure 5.1 exemplifies this concept.

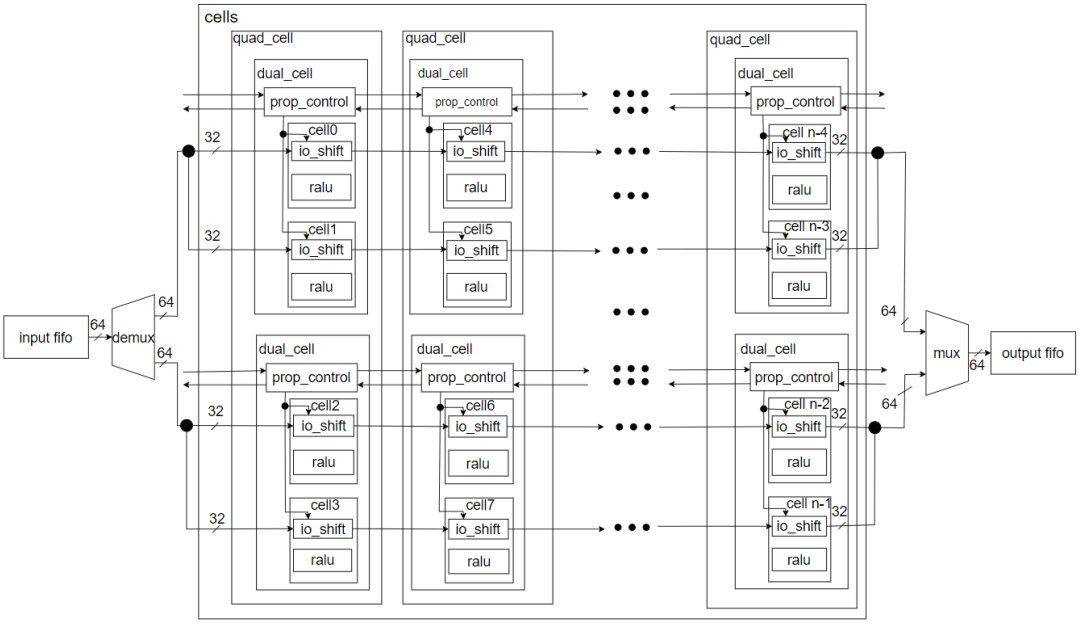

Figure 5.1 *IO quad cell organization*

5.2 Improvements

While the form proposed in Figure 5.1 was both functional and efficient, in order to keep in line with the concept of parametrization and reconfigurability presented in Chapter 4, a generalization of this concept was in order. Thus, the multicell/partial_multicell structure was created (see Figure 5.2).

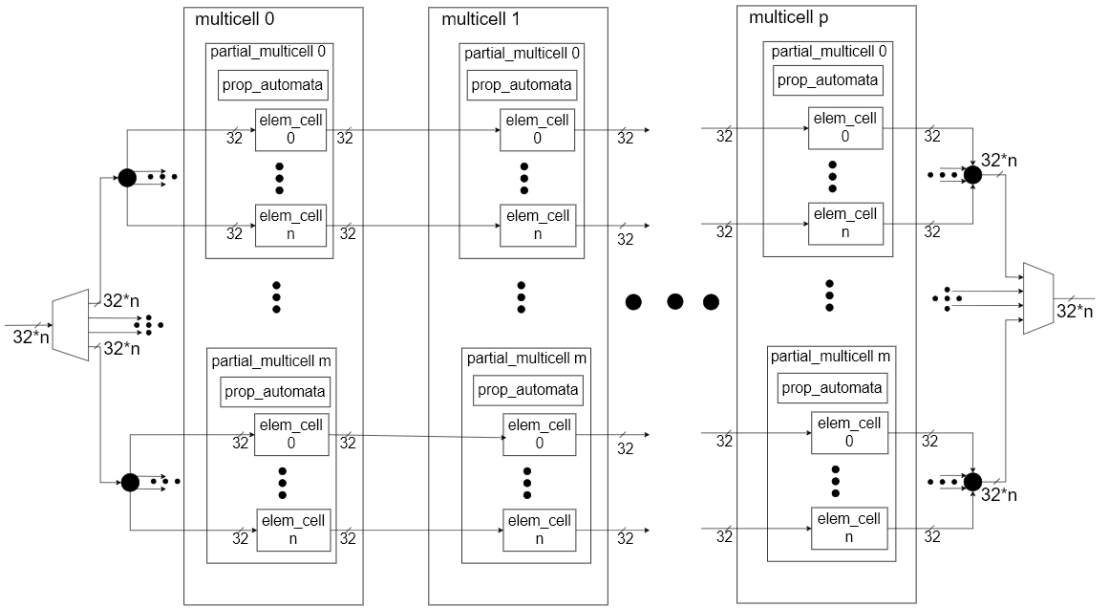

Figure 5.2 *IO multicell organization*

Cells are grouped into partial multicells. All cells from within a partial multicell are loaded in parallel. Multiple partials form a multicell. Each partial multicell from within one Multicell is loaded and unloaded sequentially in relation to the other Partials. As such, partial multicells become a generalization of dual-cells and multicells become a generalization of quad-cells. This will help support large data busses at the system level.

5.3 Conclusions

While, for the current setup, this generalization is not strictly needed and can be considered in excess (IO interface is at maximum 64 bits due to AXI Stream and system level connections), considering the perspective of having other interfaces and an unknown number of bits that can be parallelly transported, I view this improvement as important.

Secondly, this flexibility can be leveraged to help the FPGA placement and routing algorithms. While not yet tested, I theorize that for the same number of cells, some arrangements can be more beneficial in terms of achievable clock speed.

Additionally, this organization allowed some of the options regarding Array decoder location as explained in Chapter 8.3.

Chapter 6
Architectural Improvement: Accumulator and Stack Processor

6.1 Introduction

Processors fall into three major categories:

- Register based - operations can occur between any two registers;
- Accumulator based - operations can occur between the Accumulator register and a second operand;
- Stack based - operations occur between the Top of Stack (TOS) and a second operand on the stack (usually, data right beneath it).

On our machine the register-based variant was discarded as the ISA format was initially on 16b, too small to contain all desired opcodes and a significand number of addressable general-purpose registers. Because of this reason and because performance was not majorly impacted in a negative way, an accumulator-based approach with a very large register file (used as second operand) was chosen.

Accumulator processors free up space in the instruction format, but unfortunately, often require large amounts of load/store operations. On the other hand, a pure stack processor is more restrictive regarding its second operand (it must also be on the stack), but cuts even deeper into instruction format, as no operand address is usually needed. Operations are performed between the Top of Stack (TOS) and the Under Top of Stack (UTOS). An usual operation consumes both of these operands places the result as the new TOS. From a sequential point of view, this can be seen as a double pop (2 operands) followed by a push (result).

6.2 Implementation

A stack processor is not inherently better or worse than an accumulator processor and in order to combine advantages from both approaches, the two concepts were combined into an efficient hybrid machine.

An accumulator-based approach has proven highly efficient for this accelerator on many problem classes and there was no need to completely discard it. In addition to it, the stack mechanism allows storing of data for future use in an address independent manner and was chosen for its simplicity in both hardware design and usage. The resulting hardware structure is shown in Figure 6.1. Implemented stack specific operations are shown in Figure 6.2.

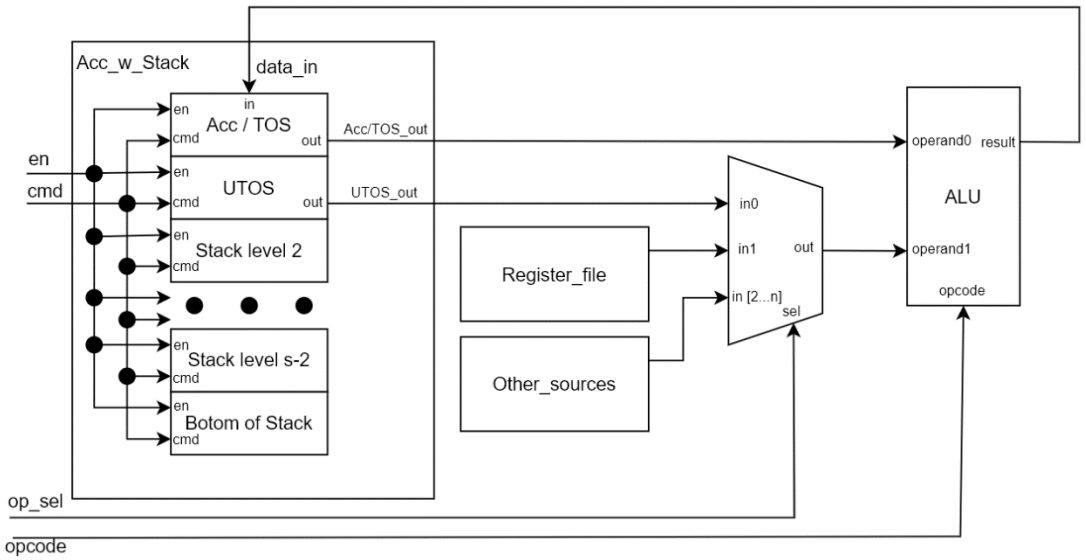

Figure 6.1 *Stack processor hardware design*

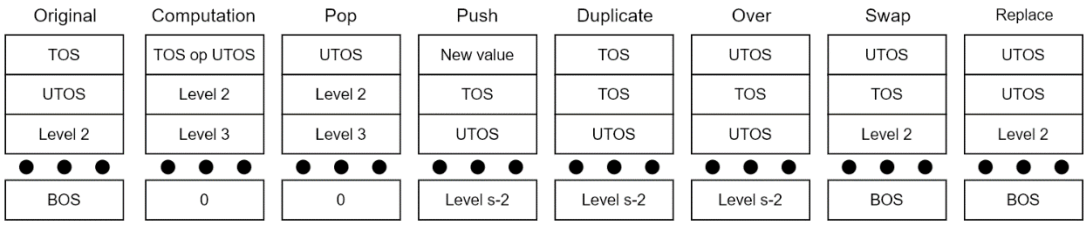

Figure 6.2 *Stack operations*

This improvement can also be seen as adding a new "addressing mode" for operand selection.

Testing the proposed architecture proved successful, with many use cases in which working with a stack improved function execution time. In use cases found thus far, it has proven helpful for small improvements, chipping 1 instruction at a time, off the accumulator only code. Over the course of larger programs and multiple nested loops these small improvements can become significant.

Further testing is required in order to properly measure and judge the impact of this addition to the architecture.

Chapter 7
Architectural Improvement: Floating Point Support

7.1 Floating point operation description

Floating Point (FP) operation support has been added to the accelerator architecture.

All Floating Point operations are designed as microprograms. Each such microprogram requires a number of clock cycles (see Table 7.1) until it is completed. As the FP unit is not a separate coprocessor, no other computations can be done while FP operations are in progress. Floating point operations with denormalized numbers, 0, infinity, NaN are also supported.

Table 7.1 *Floating point operations - clock cycles per operation*

addition	7
subtraction	7
multiplication	7
division	57

7.2 Floating point additional hardware

In both Array and Controller all added hardware is identical.

In order to respect the concepts of reconfigurability and conditional synthesis, floating point (FP) support is governed by two parameters (one for Controller and one for Array). These parameters control if all FP operations are supported and if all FP circuitry is present or not.

From the hardware point of view, FP operations require the barrel shifter/rotater be present and a separate dedicated floating-point module. The floating-point module contains a priority encoder, some registers, multiplexer, some additional data processing logic and some control logic. This logic controls multiplexer selects, register loads, and all control inputs for the barrel shifter/rotater. In regard to the barrel shifter/rotater, an additional multiplexer had to be added.

From a circuit size point of view, the floating-point unit is almost as large as all the other computation related circuitry in each cell (excluding the DSP primitive used) and will require optimization in a future version.

7.3 Conclusions

Floating Point operations were implemented and tested for both Array and Controller.

These operations are written as microprograms as each cell does not have a FP coprocessor. This allowed reuse of already present hardware resources.

Additional hardware had to be designed and integrated into the accelerator in order for FP operations to function. The ALU and local shift register from each processing core were also used in the microprograms which in turn reduced the amount of additional hardware needed.

Floating point operations with denormalized numbers, 0, infinity, NaN are supported.

The currently implemented floating point mechanism has one main drawback, the fact that is occupies a large number of LUTs.

Chapter 8
Architectural Improvements: Miscellaneous Improvements

8.1 Global rotate via global shift register (GSR)

The Global Shift Register (GSR) is used to move data between neighbouring cells by shifting data to the right(initially). The GSR is distributed along the accelerator, one shift register cell in each array cell. The current GSR cells are connected to their right neighbours (sends data to the right), to their left neighbours (receives data from the left) and to the accumulators from the cells they inhabit. The accumulator connection is bidirectional allowing both load and store operation.

In order to better use this resource, multiple improvements were made. Firstly, bidirectional data shifting was implemented. While storage elements remain the same (one register per array cell), a multiplexer was added for left/right data selection. With this hardware framework in place, rotate operations were also added. The resulting hardware structure can be seen in Figure 8.1.

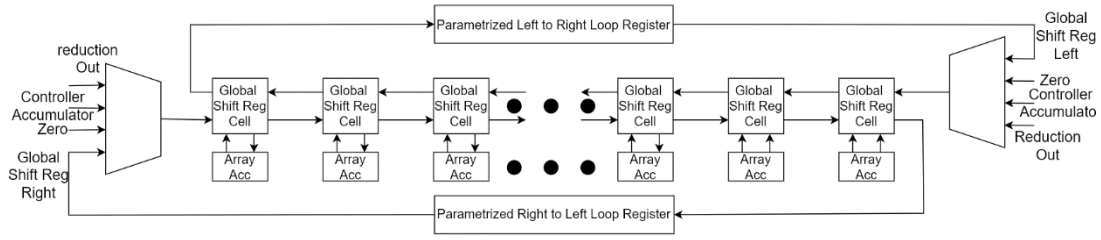

Figure 8.1 *The Global Shift Register - improved*

Additional instructions were also added in order to use these new hardware capabilities.

8.2 Barell shifter for local Shift/Rotate operations

Shift/Rotate operations can also happen at the cell level, using data present in the Accumulator. This type of bit shifting is very useful for multiple algorithms and was implemented in both Controller and Array. Local shift/rotate operations required additional hardware in order to be performed, specifically a Barrel Shifter (for rotate) and a Bar Generator followed by “and” gates (for shift operations).

8.3 Array decode location

While initially, array instruction decoding happened in each computational cell, it can now happen in one of four locations: (1) Controller: only one decoder is present for all cells; (2) Multicell level; (3) Partial Multicell level; (4) Elementary Cell level: one decoder per cell. The trade-off present here is between consumption of area (multiple decoders results in a larger area) or higher congestion through the distribute net (if decoding is done once and instructions are expanded into commands that need to be transported). Further study is required concerning which of these approaches is optimal, and how the term "optimal" is defined regarding this subject.

8.4 DMA mechanism for controller memory

While not as frequently used as the Array memory DMA, my colleagues and I decided to also implement a DMA mechanism for the Controller memory. This mechanism is still useful when relatively large amounts of data are to be passed to the Controller and the program path used for function parameters would prove to be too slow. Such cases include filter values for convolutions or precomputed keys for cryptographic algorithms (ex: AES). It functions very similarly to its Array counterpart described in [13] and [15].

8.5 Adding address registers

Both Controller and Array support indirect addressing modes based on the AddrReg register. It holds the base address with which offsets are added to obtain the final physical address. This addressing mechanism was initially the same for both Array and Controller.

The indirect addressing modes it provides are valuable for 2 reasons. Firstly, some programs (most intense computation) use clear addressing patterns and are greatly sped up by this mechanism, especially by indirect addressing with AddrReg increment (for strided access). Secondly, having a base address for data stored by each program (or by each primitive function call) allows IO to happen on half of the Array memory while the other half is used for computation. Thus, using this method, all addressing must be done in a relative manner inside all programs written.

A problem arises when computation has multiple data access patterns, each following a different rule. A second problem appears when multiple primitive function calls are to process the same data, but the first one modifies the base address register. In this situation, subsequent functions use an improper base address In the Controller, as it is a single entity, the most flexible (and space consuming) solution was adopted. This consists of having a base address register block and adding one additional instruction for switching which specific register is active.

In the Array, as each cell has its own AddrReg, having a block in each cell, together with all the additional circuitry needed for control and selection would increase circuit size unnecessarily. A simple 2-layer stack structure was used.

Chapter 9
Architectural Improvements: ISA Upgrades

9.1 New instruction format

The first and also most notable improvement was increasing the instruction size from 16b to 32b. The size of the opcodes remains the same but having a larger value that can be loaded helps with increasing all data processing and accelerator data sizes.

This instruction format was written in a parametrizable fashion, such that future growth (if needed) is easily supported. Macros were used everywhere format and size information was needed.

9.2 Meaning and organization of opcodes

All opcodes are generated using macros such that numbering and value assignment is done automatically at elaboration.

For each family of opcodes (control or data operations), the exact location where opcodes are executed (Controller or Array) also affects the precise numbering. The same opcode can be used for different operations in Controller vs in Array.

One opcode can have up to 4 meanings, based on operand type and on where it ends up being executed. Both the assembler software tool and the hardware synthesis tool use the same files in order for each instruction to perform its desired function.

9.3 Instruction opcode compression

In the process of improving this architecture, new instructions had to be added and at one point no more opcodes were left free for new assignations. This led to reducing the number of used opcodes by grouping together similar instructions.

Originally, each ASM instruction had its own opcode, which left very few open codes, especially for control type instructions. The adopted solution was to group similar instructions together and use parts of the "value" field as a continuation of the opcode, a type of secondary opcode. This method was adopted for jump/branch instructions, stack control, spatial selection, fixed right shifting, special load/store, global shifting and others.

9.4 Jump/Branch

Two main upgrades were done to the jump/branch mechanism: adding the "decrement register" and the "compare value register". Originally, all branches were done with accumulator comparison.

The decrement register is used for branch with decrement instructions and allows decrement and increment with settable values. It should be noted that the decrement register is always subtracted, but it can hold negative values such that this subtraction becomes an addition. This register is an extremely valuable feature as it allows cleaner code that can be more easily understood. In some cases, the loop counter "i" is also needed to compute addresses or other information for the array, and specific counting direction or even increment or decrement with values different from "1" are needed. This allows strided access in Array cell memory to be easily done via the Controller, without needing additional Controller memory load/store steps.

All previous branch types depended on the accumulator and on the flags that it governs: zero, carry, negative. For maximum flexibility regarding writing loops, I wanted to also take into consideration the "if(accumulator == value)" case. As such, the "value register" was born. Both equals and does not equal cases are taken into account, each having its own jump/branch type ASM instruction. A secondary improvement was made by allowing the value register to be incremented or decremented by the decrement register and be itself checked for zero or negative. This created another level of parallelism inside the Controller, uncoupling the accumulator from actual loop controls. An additional feature developed was having a parametrizable number of value registers, now also seen as loop counter registers. This was done to further improve control over loops, especially in the case of nested for loops.

9.5 Swap accumulator and memory location

A new instruction was added to the ISA, one that swaps data between the Accumulator and a location from memory. This can be viewed as a load combined with a store instruction, both performed in the same clock cycle and is used for optimization purposes.

Adding this new instruction required very small hardware changes in the decoder area. The rest of the data paths and everything else needed was already in place, a proper control mechanism being the only thing missing.

9.6 Shift/Rotate with fixed amount

In the original architecture only right shifting with "1" could be performed. Left shifting could be done by multiplication with powers of two. This proved to be a limiting factor in some applications.

For many applications, required shifting is always done with the same amount, usually 8b. Because of how common this number is, relatively low hardware requirements and the fact that this increases shift speed by a factor of up to 8 in some cases, a special "shift with fixed amount" instruction was added to the ISA. Everything

related to this mechanism is parametrizable, its presence can be enabled or disabled and the shift amount (if values other than "8" are needed) can also be set at synthesis. As this mechanism is completely separated from the Barell Shifter, if all shifting in the application is done using the same amount, a lot of area can be saved.

9.7 Global Shift/Rotate with or without bool

The Global Shift Register operated independently of cell status, specifically it did not take into account if the Array cells were active or not. I devised and implemented three types of shifting for the GSR (in both directions)

- **Shift without Bool**
  Bool value is not taken into account when shifting, all cells perform the operation. This was the version previously implemented.
- **Shift with Bool and add "0"**
  If the current cell is inactive, its value stays the same. If it is active and if the previous cell is inactive, previous cell value is ignored and a "0" is inserted.
- **Shift with Bool and hold**
  If the current cell is inactive, its value stays the same. If it is active and if the previous cell is inactive, previous cell value is ignored and the current cells value is held. This effectively duplicates values found at boundary locations between active and inactive cells.

From a hardware point of view, the internal multiplexer of each GSR cell grew in size, this being the major downside of adding these operations.

9.8 Program load at any address

Accelerator programming is usually done at the start of computation and initially programs could only be loaded at address 0. To improve programmability, small changes were made such that the program could be stored at any address in the instruction memory.

9.9 Address register operations

For even more efficient usage of the Address register (in addition to what was described in Chapter 8.5), additional operations were added for changing its value. These operations are additions between the current register and data coming from one of two sources: either the Accumulator or the "value" bits of the current instruction.

These instructions allowed great flexibility for address control as it allowed offsets either computed in the Accumulator or from instruction value, to be directly added to the AddrReg without any additional load or store operations.

Chapter 10
Synthesis Results

10.1 Results

This chapter presents synthesis results for different machine configurations, the most important of which are shown in Table 10.1 and Table 10.2 below. Conditional synthesis is employed and based on desired functionalities, the architecture is generated with or without specific blocks. Synthesis was done targeting a Zynq 7020 SoC, specifically a Pynq-Z2 board using Vivado 2022.2.2.

Additional parameters that are not subject to this test are:

- Memory sizes: data memory 1024 words, instruction memory 2048 words, input, output, program in memories: 1024 words;
- Controller is fully synthesized, having a floating-point unit, selectable address register and a data stack of depth 4;
- Array instruction decoder is set in the Controller;
- Data size: 32b.

Table 10.1 *Synthesis results: Map-Reduce*

Nr. cells	Slice LUTs	Slice REGs	F7+F8 Muxes
8	9675	3827	761
16	11997	5337	765
32	17012	8654	752
64	26710	15152	743
128	46889	28031	734

Table 10.2 *Synthesis results: Map-Scan-Permute-Pack, maximal Array cells*

Nr. cells	Slice LUTs	Slice REGs	F7+F8 Muxes
8	17683	6662	843
16	28512	11837	918
32	50957	23959	1024
64	97481	50703	1258
128	206631	110611	1772

10.2 Conclusions

From synthesis data, the following conclusions can be drawn:

- LUTs are the main resource that gets consumed with growing the accelerator size.
- The number of BRAMs and DSPs available on each board also limits the number of computational cells that can be generated. This however is not the hardest restriction. For high-end FPGA devices, DSP units are in the low thousands and BRAM blocks in the several hundreds range. RAM can also be obtained from LUTs as distributed RAM. UltraRAM blocks are also an option.
- Sizes of 256-1024cells can be synthesized on current generation high-end FPGAs, depending on particular architecture customisation. The parametrization and reconfiguration mechanism described in Chapter 4 proves very valuable in optimizing for size for specific classes of applications that may or may not require the presence of all possible functional blocks.
- The new IO mechanism, all the Controller improvements and the block design needed to achieve system level functionality occupy roughly 11k LUTs. This is a relatively large number for designs with a small number of cells but becomes less and less relevant as the number of Array cells increases.
- In terms of LUT usage, a maximal cell is roughly five times as large as a minimal cell.
- Floating point support is relatively large in terms of resource usage and should be avoided if the application permits it. Fixed point computation is preferable.
- Floating point hardware support requires size optimizations.
- The multifunctional network itself, also occupies a large amount of space even with conditionally synthesized cells. Its size is: $n \times (2 \times \log_2 n - 1)$ network cells (roughly four times the number of cells of the reduce-only network).

Current description is a mix of structural and behavioural descriptions, and its prime objective is to provide a proof of concept for the architecture. In a subsequent development phase, the circuit will undergo size optimization techniques.

Chapter 11
Application-Level Improvements: AES Algorithm

11.1 Introduction

This chapter details the implementation of the Advanced Encryption Standard (AES) algorithm on our custom Map-Scan-Permute accelerator architecture. This work was presented at [20].

The Advanced Encryption Standard (AES) is a block cypher that was chosen by the United States government to supersede the Data Encryption Standard (DES). The AES is a symmetric key block cypher that supports multiple key sizes, specifically 128b, 192b and 256b. Multiple variants of the algorithm also exist, such as : Electronic codebook (ECB), Cipher block chaining (CBC), Cipher feedback (CFB), Output feedback (OFB), Counter (CTR)[21].

11.2 Implementation and results

The AES algorithm has two main stages: the key expansion and round computations. The key expansion is used to transform the cypher key into multiple keys, one used per round of encryption. Key size can be 128b, 192b or 256b, each key size dictating the number of rounds the algorithm is performed for.

In our work, we have currently implemented a variant where each block of text is encrypted on one cell, a pure SIMD approach, using the key computed and stored in the Controller.

The first stage of the algorithm, the key expansion is done on the Controller as to reduce energy consumption. For the next steps, the Controller is used for loop and branch control, for memory address precomputing (when applicable) and for distributing the expanded key to all cells. In the Array cells, data is stored as 8b per memory location. The SubBytes step uses a look-up table stored in each cell. The Shift Rows and the Mix Columns simply rearrange data and the Add Round Key is a basic xor operation between data and received key.

The AES algorithm has been implemented for all three key sizes and for two modes of operation: ECB and CTR. Latency results of these implementations are shown in Table 11.1. The key expansion stage adds 5459 (128b key), 5741 (192b key), 8106 (256b key) additional clock cycles.

Additionally, transfer times can be negated by using the mechanisms described in [13] and [15].

Table 11.1 *AES 128 ECB - latency results in clock cycles[20]*

	Number of cells			
Number of blocks	16	32	64	128
16	9353	n/a	n/a	n/a
32	18942	9353	n/a	n/a
64	37688	19086	9353	n/a
128	75180	37832	19358	9353
256	150164	75324	38104	19886
512	300132	150308	75596	38632

According to [20] "AES192 follows the same general trends, with an increase of roughly 21% compared to AES128. Similarly, AES256 has an increase in execution time of 42% compared to AES128. The same trends are followed by both the ECB and CTR variants.

Compared to the ECB variant, CTR grows by roughly 1.5%, 1.3%, and 1%, depending on key size. This is observed over all combinations of number of blocks and number of computational cells. The CTR mode only adds a few more operations at the beginning and the end of the ECB mode."

In terms of throughput per core, if all devices taken into account are scaled to the same frequency, the proposed accelerator is situated below other solutions found on the market due to the very simple nature of its computational unit. We are however in the same order of magnitude as other solutions, achieving 2.7 MB/s per core at a scaled frequency of 1.6 GHz. GPU implementation from [22] offer 5.2 MB/s per core with modern CPU implementations having performances of over 10 MB/s per core at the scaled frequency.

However, based on the simplicity of our processing element and on previous power estimations, we should be able to obtain roughly 2x Gbps/W compared to GPU solutions.

11.3 Conclusions

The AES algorithm has been implemented on our accelerator and this implementation offers decent, but not good performances in terms of throughput and latency. The current implementation was done with a pure SIMD approach in mind.

Based on the nature of the AES algorithm, the accelerator's hardware resources are not used to their fullest, specifically, the multifunctional network is not used and as such, acceleration is limited. Compared to a single core machine, acceleration is obtained through the separation of Control from data-oriented computation and through the fact that the machine has multiple computational cores.

Chapter 12
Application-Level Improvements: Square Matrix Transpose

12.1 Introduction

The transpose operation is one of the basic data movements required in linear algebra. It involves switching data between a matrix's lines and columns, such that line0 becomes column0, line1 becomes column1 etc. Time complexity of the transpose algorithm on a pure serial machine is in O(n^2), passing sequentially through all the elements in the matrix.

12.2 Map-Permute approach to the transpose

Our approach is structured around the permute function from our multifunctional network. Using it, a complete size "n" vector can be transposed at a time, thus reducing time complexity of the transpose algorithm to O(n). Inside the Array, data is stored into each Cell's memory as shown in Figure 12.1 for a 16×16 matrix in a 16 cell Array. The term "vertical vector" is used for columns and the term "horizontal vector" for lines.

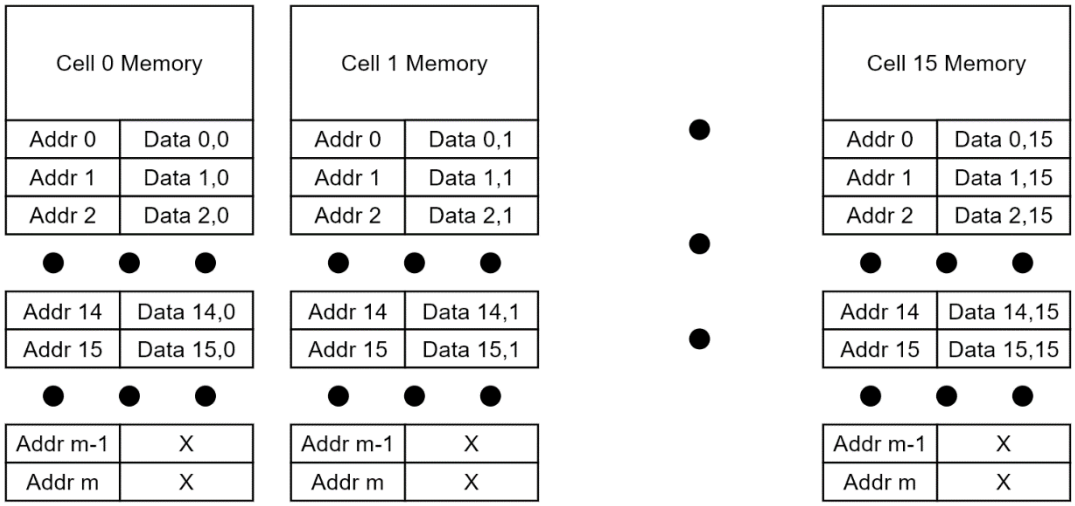

Figure 12.1 *Array data arrangement for a 16x16 matrix on 16 cells*

Our approach is based on walking broken diagonals, specifically the main diagonal. Data from one broken diagonal is processed at a time, in total "n" data values. At each

iteration of the main loop, each cell will output one data point and receive and store another. For a transpose algorithm that processes "n" values at a time, the main problem that needs to be addressed is how to avoid collisions at both reading and writing. One cell can output only one data value and one cell can only receive one data value, per iteration. For this reason, broken diagonals were chosen. This access pattern perfectly fulfils the above mentioned requirement but will require some address computation logic in order to properly read/write data from the appropriate locations. The algorithm is structured in 3 steps and is exemplified in Table 12.1:

- Loading a broken diagonal;
- Permutation;
- Storage of a broken diagonal.

Table 12.1 *Map-Permute transpose example*

	step 1	0 5 a f	4 9 e 3	8 d 2 7	c 1 6 b
Initial matrix:	step 2	0 5 a f	3 4 9 e	2 7 8 d	1 6 b c
0 1 2 3 4 5 6 7 8 9 a b c d e f	step 3	0 x x x x 5 x x x x a x x x x f	0 4 x x x 5 9 x x x a e 3 x x f	0 4 8 x x 5 9 d 2 x a e 3 7 x f	0 4 8 c 1 5 9 d 2 6 a e 3 7 b f

With this approach, permutation bits need to be precomputed in advance, stored in the main memory and loaded at the appropriate time. After a proof-of-concept implementation, optimizations were considered and implemented.

This approach led to a time complexity of O(n) with a constant of around 13.

In the case of transposing matrices that are not square, padding with 0 is required until a square form is reached.

Chapter 13
Application-Level Improvements: Fast Fourier Transform (FFT)

13.1 Introduction and steps

This chapter details the design, functioning and performance of the Fast Fourier Transform algorithm on our custom Map-Scan-Permute architecture. This work was presented in [23].

Considered one of the backbone algorithms of modern computing the FFT is used nowadays whenever the Discrete Fourier Transform needs to be computed and computing it from the definition proves too slow. This algorithm reduces time complexity from $O(n^2)$ to $O(n \times \log_2(n))$.

The best-known variant of computing the FFT is the Cooley-Tukey algorithm and is based on recursively computing smaller and smaller transforms [24].

By using complex numbers, the real and imaginary part was stored in two successive memory locations and operations actually required multiple steps. Addition required two addition operations while complex multiplication required four multiplications, one addition and one subtraction. On our architecture 8.8 fixed point complex numbers were used. Multiplication itself (being in fixed point) was done in two steps: actual multiplication and right shift by 8.

13.2 FFT state of the art

As a very important algorithm in modern computing, FFT has been implemented on all currently available platforms: GPUs, FPGAs, DSPs.

Surveys regarding FFT implementations can be found at [25] and [26].

Work regarding the FFT algorithm on previous versions of the accelerator can be seen in [27] and [28].

13.3 FFT implementation and variants

As seen in Chapter 13.1, the FFT requires two types of operations:

- Data movement;
- Actual computation: addition, multiplication.

In adapting this algorithm to our architecture, each of these two categories had to be taken into account. Computation is done in each Array cell and data movement is done through the multifunctional network, specifically, by using the permute function.

All computation was done using fixed point 8.8 arithmetic as a functional floating-point unit was not developed at the time. Switching to floating point should not be troublesome, however obtained results will suffer as these operations require six clock cycles to be performed (see Chapter 7).

Algorithm validity was tested by comparing obtained results to a reference C++ implementation.

For proper hardware testing, multiple variants were developed, for smaller/larger data sizes and for different modes of data organization. Vertical organization refers to data held within one Array cell memory, at multiple addresses, while horizontal organization refers to data being held within multiple Array cells, at the same address.

Variants are as follow:

- Pure serial - a purely serial monocore implementation
- "Dual core" variant - Controller + 1 Array cell
- Many-core variant: vertical. It is the many-core generalization of the previous variant. Comprised of Controller and as many Array cells as desired. The FFTs computed in each cell are completely independent of one another.
- Many-core variant: horizontal. This variant makes use of all the cells and the inherent parallelism of our hardware structure to improve the time it takes to solve one FFT. Operations are performed in each cell and data movement, the butterfly permutations, are done using the multifunctional network, specifically, the permute function. This setup offers the lowest latency of all variants discussed. It is, however, not as throughput efficient as the vertical setup. This logarithmical decrease in performance comes from the latency induced by the multifunctional network.
- Many-core variant: rectangle. The rectangular version comes into play when the number of samples is greater than the number of Array cells. Data is organized in a similar fashion to the horizontal version, but it now occupies multiple memory locations. In this situation, a combination of the vertical and horizontal approaches is used.
- Many-core variant: square. In addition to the previous variant, if the number of memory lines occupied in each cell is equal to the number of cells in the array (the rectangle is actually a square), another, more efficient algorithm could be employed.
- Multiple small rectangles/squares organization. This organization allows fine control over the throughput/latency compromise. Using less cells to compute one FFT will increase throughput, but also latency, while using more cells will decrease both.

13.4 FFT simulation results

In this section, results from simulating our multiple variants of FFT setups are shown. All simulations were done targeting FPGA implementation, specifically a ZYNQ7020 System on Chip. Throughput and latency accelerations are calculated in

reference to a purely sequential single core machine with a similar instruction set. All results are shown in clock cycles and not actual frequencies in order to express architectural improvements and not technological improvements. Transfer operations are not included as at the point of testing that part of the system was not yet fully functional. Additionally, transfer times can be completely negated as described in [13]. Evaluation results are obtained through direct simulation or estimations based on ASM code metrics and direct simulation results.

Results discussed in this chapter can be summarized in Table 13.1, showing a FFT with 256 samples in multiple configurations on a 64-cell machine. This is done in order to show the versatility of our machine and the throughput versus latency compromise.

Table 13.1 *FFT results: multiple versions with 256 samples*

Number of array cells	Samples per FFT	FFTs in parallel	Cycles per sample	Through put acc	Latency acc	Latency in clock cycles	Data config.
64	256	64	4.2	95.9	1.5	69752	Vertical
256	256	1	3.5	114.0	114.0	908	Horizontal
64	256	1	12.1	33.1	33.1	3121	Rectangle
64	256	2	10.7	37.5	18.7	5508	Rectangle
64	256	4	9.4	42.8	10.7	9659	Rectangle
64	256	8	8.1	49.5	6.1	16702	Rectangle
64	256	16	6.8	58.6	3.6	28257	Rectangle
64	256	4	4.5	89.2	22.3	4640	Square

13.5 Comparisons with other systems

"For comparison, the FFT 4096 was tested on three other systems: local testing (on an Intel I7770HQ, 8CPU, 2.8 GHz) of a CPU implementation of the FFTW algorithm [29] local testing (on an Nvidia GeForce GTX1050, 640 cores, 1.354 GHz) of a GPU implementation [30] and FPGA Xilinx IP implementation (targeting ZYNQ 7020 SoC, Vivado default settings) [31].

Frequency scaled, the CPU implementation achieved roughly 126 Mega samples per second, the GPU 175, and our proposed general-purpose accelerator 207. Something to note is that our implementation is in fixed point, while the CPU and GPU operate in floating point, and as such an approximately 3×performance degradation is expected when switching to floating point. Regarding fixed vs floating point computation, it should be noted that with larger FFT sizes, fixed point additions and multiplications accumulate errors and will no longer give the correct result. Another thing to note is that compared to the GPU, our machine uses 64 cores instead of 640 and would be much more energy efficient. This also shows our architectural benefits as the computing cores are much more efficiently used. The Xilinx IP configurator (on default settings for fixed point FFT) offers a latency of 14465 cycles, roughly half of what our machine offers. However, this IP targets only FFT computation, and is not a general-purpose architecture."[23]

Comparisons between our accelerator and other FFT solutions from literature places us between GPUs and FPGAs. This is to be expected as FPGA implementations are tailored to the problem at hand, thus offering lower latencies and lower area usage, at a price in versatility.

13.6 Conclusions

FFT, one of the backbone algorithms in digital signal processing pertains for acceleration on our architecture. The work presented improves and expands previous work shown in [27] and [28].

The parallel patterns that this algorithm uses are mainly Map (for computation) and Permute, for which our machine is perfectly suited. The algorithm itself also has a high degree of operational intensity (by which we understand the amount of computation done per number of bytes transferred) and is also inherently parallelisable to using all or almost all available processing cells.

Multiple data organization patterns have been explored and proven functional and efficient, with each being suited to a specific scenario or to specific application requirements. Vertical data organization in which each cell individually computes a FFT is best when throughput acceleration is the only target. When latency is hard constrained, the horizontal setup utilises all array cells for only one FFT and offers the best latency accelerations achievable on this architecture. Combinations of the two styles, in addition to being able to limit the number of cells attributed to each FFT gives users fine control over the throughput-latency compromise.

A novel way of computing the FFT, specifically designed for this kind of Map-Permute machine was also developed and tested. It shows that good throughput can be achieved without paying a large price in latency, by designing appropriate software solutions that make efficient use of the hardware resources and of other algorithms that are well suited to this machine. In the square matrix organization of samples, permutation network latency is hidden by using the transpose operation, an operation that can be efficiently pipelined, whereas the rectangular setup could not.

Chapter 14
Miscellaneous Improvements

14.1 Study: Avoiding Latencies of Log-Depth Parallel Computational Patterns

Based on the previously described algorithms and on our experience using the accelerator, a study has been done regarding techniques for the reduction of latencies pertaining to the Scan-Permute-Reduce network.

We have identified four methods with which latency can be avoided or hidden, based on four applications developed. This study was presented and published at [32]. These methods mainly revolve around pipelining, either in its direct form or different variants. Another important aspect discussed is the fact that in some situations, a completely new algorithm can emerge, dramatically improving performance. The algorithms and solutions for matrix-vector multiplication (pipelining), square matrix transpose (block grouped pipelining), FFT (algorithmic improvements) and, pooling (mixed function pipelining) are discussed.

14.2 Testbench: program and IO support

The original experimental setup was inflexible and an improved testbench, that could more easily process both programs and data needed to be written. For ease of use, a simple mechanism for comparatively testing the output file against a golden model file was also developed.

14.3 Testbench: printing and debug options

Debugging Verilog simulations is a very arduous task. In order to simplify this process, a complex printing to console mechanism was developed. Both continuous and final printing of selectable hardware resources is supported.

14.4 Macro parsers

As this project is highly parametrized, different architectures can easily be generated with minimal changes to a set of parameters. Changing the ISA was also necessary on several occasions, which in turn produced a complete renumbering of usable opcodes. In order to easily follow macro values a macro parser was needed.

These parsers require input files in Verilog/SystemVerilog containing the macros whose values we want to see. The output file generated is a map/dictionary comprised of pairs: "macro_name macro_value".

14.5 Petalinux and system

The previous version of the complete accelerator system used a Linux image distributed by TUL, creators of the PYNQ-Z2 board for which we targeted implementation. In order to have complete control over the software layer, our team decided to rebuild the Linux image from scratch. Xilinx recommends building using Petalinux, a set of software tools used specifically for embedded Linux development.

With complete access to the underlying operating system, multiple low-level changes were possible. The most important of these changes is in reserving address spaces for use only by the accelerator runtime environment and application, specifically removing them from the main memory used by all other applications.

While image building and deployment proved troublesome, the Linux image was successfully booted on the PYNQ-Z2 board. Using SSH, remote login via ethernet was also possible.

As a first step, applications were written via Vitis SDK and then moved and tested on the board. Another improvement was the addition of a complete on-board build environment so that applications can be developed and tested locally. Writing basic test applications proved troublesome as some drivers were missing and had to be rewritten.

The Petalinux system deployed is still running and used for testing different versions of both the hardware accelerator itself, as well as of low-level SDK components (specifically runtime environment and application build system).

Chapter 15
Conclusions

15.1 Objectives and results

My research was dedicated to improving the parallel processing capabilities of a heterogeneous Map-Reduce system. Improvements were done in regard to both hardware and software.

From a hardware point of view, three categories of improvements arise:

- readability, reconfigurability, parametrisation;
- small improvements and bugfixes;
- large improvements requiring dedicated hardware blocks.

Each of these categories received its due attention and time which resulted in a powerful Map-Scan-Permute accelerator that can be easily reconfigured to fit application needs and space constraints.

Large improvements were made in order to allow floating point computation and Scan and Permute type operations. Introduction of the stack concept into our machine also helped reduce the size (and the execution time) of critical loops. Different types of shift and rotate instructions were also developed for both Array and each individual cell.

While some additions were made as a response to a specific problem or situation, some were also made in order to future proof the design. Such is the case for the generalization of Array IO, some of the added branch instructions or the Controller level DMA mechanism. Furthermore, numerous bugs were found and fixed.

From an application point of view, assembler code was written to both test the newly developed features and emphasise them by using them in computation. The AES algorithm has been adapted for our accelerator, a novel transpose algorithm was developed, specifically suited to the Map-Permute type architecture and was then used to efficiently compute the Fast Fourier Transform. The FFT algorithm was also implemented on our architecture and subsequently tested in multiple forms. The specific FFT form to be selected and used in applications, will depend on application needs in terms of throughput and latency.

This system was then synthesized and implemented on a PYNQ-Z2 SoC in order to prove its functionality. A Linux image was built to run on the ARM Cortex A9 in order to build and test actual C/C++ applications.

To summarize, a fully functional heterogeneous computing system has been developed, improved and implemented, a system that offers good performances in terms of parallelism, scalability, latency and power consumption.

15.2 Original papers

1) Mihai Antonescu, Gheorghe M. Stefan, "Politehnica" Univ. of Bucharest, Romania: “Multi-function Scan Circuit”, International Semiconductor Conference CAS2020; IEEE conference proceedings; doi: 10.1109/CAS50358.2020.9268048.

2) Mihai Antonescu, Mihaela Maliţa, Gheorghe M. Ştefan “Latency Hiding of Log-Depth Scan and Reduce Networks in Heterogeneous Embedded Systems”, 29th International Symposium for Design and Technology in Electronic Packaging (SIITME), 2023, IEEE conference proceedings, accepted for publication.

3) Mihai Antonescu, Gheorghe M. Ştefan, "Multi-Function Scan Circuit for Assisting the Parallel Computational Map Pattern", Romanian Journal of Information Science and Technology (ROMJIST), Vol 27, Nr. 1 of 2024, accepted for publication.

4) Andreea-Cătălina Pietricică, Mihai Antonescu, George-Vlăduț Popescu. "Evaluation of AES Cryptographic Algorithm on a General-Purpose Map-Scan Accelerator", International Semiconductor Conference CAS2023, IEEE conference proceedings, DOI: 10.1109/CAS59036.2023.10303705.

5) Mihai Antonescu, Mihaela Maliţa, “FFT on a Heterogeneous System with a General-Purpose Map-Scan Accelerator”, Romanian Journal of Information Science and Technology (ROMJIST), accepted for publication.

6) M. Antonescu, C. Bîra, “Discrete Gravitational Search Algorithm (DGSA) applied for the Close-Enough Travelling Salesman Problem (TSP / CETSP)”; International Semiconductor Conference CAS2019 Sinaia România; IEEE conference proceedings; DOI:10.1109/SMICND.2019.8923719.

7) M. Vasile, S. Martoiu, N. Boukadida, M. Antonescu, A. Ulmamei, G. Stoicea, R. Hobincu, C. Iordache and on behalf of the ATLAS TDAQ collaboration, “FPGA implementation of RDMA for ATLAS readout with FELIX at high luminosity LHC”, published 20 May 2022 • © 2022 IOP Publishing Ltd and Sissa Medialab, Journal of Instrumentation, Volume 17, May 2022, DOI 10.1088/1748-0221/17/05/C05022.

15.3 Original contributions

The major original contributions of my work can be split into two categories: hardware improvements and assembler programs written.

On the hardware side, the following were done:

- The Reduce network was improved as to support Scan operations, Permutations and the Pack operation;
- Parametrization and conditional synthesis can now be used to customize the accelerator as needed by the applications that it will run;

- The IO mechanism was improved to support external interfaces of different sizes and to allow custom cell grouping;
- The accumulator architecture was improved to support stack operations. The stack structure was added as an improvement to the accumulator;
- The Global Shift Register was updated to allow improved communication between neighbouring cells;
- A Barell Shifter was added to each cell as to allow rotate and shift operations;
- Improvement to the address register mechanism for improved memory addressing;
- ISA was updated and compressed;
- Additional Branch type instructions were added for improved program control;
- Other miscellaneous instructions were added;
- Bugfixes for any identified problems.

On the algorithmic side, the following algorithms were developed and tested:

- Algorithm for computing AES encryption;
- Algorithm for computing the square matrix transpose;
- Algorithm for computing the FFT.

The testbench used for simulation and testing was completely rewritten and a simulation flow was developed in order to allow rapid algorithm testing and debugging.

Above the hardware level, the accelerator was integrated on a PYNQ-Z2 board. The ARM processor on the ZYNQ system was used as the Host for our accelerator. A Petalinux distribution was compiled from scratch and customized for our needs and the accelerator was physically tested.

From these contributions, the papers presented in Chapter 15.2 were written.

15.4 Future work

As this project is still in development, a lot of work is still in need of being done. In the following, I will briefly describe domains of interest and focus:

In the near future, there are two directions of work:

- Consolidation and bug fixing - testing and fixing bugs is needed in order to progress, together with a thorough consolidation of the work done thus far, specifically writing of proper documentation.
- Second proof of concept full application - the first full application to be done as a proof of concept was matrix multiplication [13]. At this point, the software stack has changed in almost every aspect and as such, a second test is needed. This test should pass through all layers of the software stack, from the highest layer of C/C++ user applications, all the way down to the assembler library of functions with computation done on the FPGA implemented accelerator.

In the medium future, there are the following tasks that need doing:

- Development of the assembler primitive functions library;
- Development of the C/C++ function library that calls upon the primitive function library;
- Software tool development: assembler, profiler, tuner, debugger, IDE with GUI, runtime environment, continuous integration for git, installation scripts, cross compilation support etc;
- Hardware support for on board debugging;
- Hardware circuit optimizations - size reduction in LUTs and REGs coupled with higher working frequencies;
- Testing a larger accelerator, having more computational cells (128, 256 etc);
- Adding additional instructions and hardware features;
- Testbench development and proper testing of the machine. Testing should also be done with regard to different architecture configuration parameters and their combinations;
- Full stack application development and testing.

In the far future, the following developments are possible:

- Multiple ways to connect to the accelerator, multiple interfaces with the host computer;
- Clusters of accelerators - given their nature and shape, the Map-Reduce structure can also be scaled at the multiple accelerators level.

Bibliography

[1] Nvidia, Tensor core usage tips: https://developer.nvidia.com/blog/optimizing-gpu-performance-tensor-cores/, accessed on 20.05.2023.

[2] W. Sun, A. Li, T. Geng, S. Stuijk and H. Corporaal, "Dissecting Tensor Cores via Microbenchmarks: Latency, Throughput and Numeric Behaviors," in IEEE Transactions on Parallel and Distributed Systems, vol. 34, no. 1, pp. 246-261, 1 Jan. 2023, doi: 10.1109/TPDS.2022.3217824.

[3] Hồ, Khoa & Zhao, Hui & Jog, Adwait & Mohanty, Saraju. (2022). Improving GPU Throughput through Parallel Execution Using Tensor Cores and CUDA Cores.

[4] P. M. Basso, F. F. d. Santos and P. Rech, "Impact of Tensor Cores and Mixed Precision on the Reliability of Matrix Multiplication in GPUs," in IEEE Transactions on Nuclear Science, vol. 67, no. 7, pp. 1560-1565, July 2020, doi: 10.1109/TNS.2020.2977583.

[5] John W. Romein, "The Tensor-Core Correlator", in Astronomy and Astrophysics, Vol. 656, Article A52, December 2021, doi: https://doi.org/10.1051/0004-6361/202141896

[6] Sparsh Mittal and Jeffrey S. Vetter. 2015. A Survey of CPU-GPU Heterogeneous Computing Techniques. ACM Comput. Surv. 47, 4, Article 69 (July 2015), 35 pages. https://doi.org/10.1145/2788396.

[7] Pieter Hijma, Stijn Heldens, Alessio Sclocco, Ben van Werkhoven, and Henri E. Bal. 2023. Optimization Techniques for GPU Programming. ACM Comput. Surv. 55, 11, Article 239 (November 2023), 81 pages. https://doi.org/10.1145/3570638.

[8] A. M. Turing, On computable numbers with an application to the Eintscheidungsproblem. Proceedings of the London Mathematical Society 42, 1936.

[9] S. Kleene, "General Recursive Functions of Natural Numbers," Mathematische Annalen 112:727–742. 1936.

[10] G. M. Ştefan, M. Maliţa: "Can One-Chip Parallel Computing Be Liberated From Ad Hoc Solutions? A Computation Model Based Approach and Its Implementation", 18th International Conference on Circuits, Systems, Communications and Computers (CSCC 2014), Santorini Island, Greece, July 17-21, 2014, 582-597.

[11] M. Maliţa, G. M. Ştefan, D. Thiebaut: "Not Multi-, but Many-Core: Designing Integral Parallel Architectures for Embedded Computation", ACM SIGARCH Computer Architecture News, 35 (5)32-38, Dec. 2007. Special issue: ALPS '07 - Advanced low power systems; communication at International Workshop on Advanced Low Power Systems held in conjunction with 21st International Conference on Supercomputing, June 17, 2007 Seattle.

[12] Mihaela Maliţa, Gheorghe Ştefan: "The Berkeley Motifs and an Integral Parallel Architecture", in ROMJIST, vol. 12, no. 1, 2009, pag. 35-49.

[13] George-Vladuţ Popescu, " Architectures and Structures for Heterogeneous Computing - Improvements in Data Transfer for a Heterogeneous Computing System", PhD Thesis, UPB, Bucharest 2023.

[14] PYNQ Z2 overview, [online] Available: https://www.tulembedded.com/fpga/ProductsPYNQ-Z2.html, accessed on 24.07.2023.

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

[16] Mihai Antonescu, Gheorghe M. Stefan, "Politehnica" Univ of Bucharest, Romania: Multi-function Scan Circuit. CAS2020; International Semiconductor Conference IEEE conference proceedings; doi: 10.1109/CAS50358.2020.9268048.

[17] Mihai Antonescu, Gheorghe M. Stefan, "Multi-Function Scan Circuit for Assisting the Parallel Computational Map Pattern", Romanian Journal of Information Science and Technology (ROMJIST), Vol 27, Nr. 1 of 2024, accepted for publication.

[18] Vaclav E. Benes: Mathematical Theory of Connecting Networks and Telephone Traffic. New York: Academic Press, 1965.

[19] Abraham Waksman. 1968. A Permutation Network. J. ACM 15, 1 (Jan. 1968), 159–163. https://doi.org/10.1145/321439.321449.

[20] Andreea-Cătălina Pietricică, Mihai Antonescu, George-Vlăduț Popescu. "Evaluation of AES Cryptographic Algorithm on a General-Purpose Map-Scan Accelerator", International Semiconductor Conference CAS2023, IEEE conference proceedings, DOI: 10.1109/CAS59036.2023.10303705.

[21] "Recommendation for Block Cipher Modes of Operation" (PDF). NIST.gov. NIST. p.9. Archived (PDF) from the original on 29 March 2017, [online] Accessed at: https://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-38a.pdf, on 29.07.2023.

[22] J. Ma, X. Chen, R. Xu and J. Shi, "Implementation and Evaluation of Different Parallel Designs of AES Using CUDA," 2017 IEEE Second International Conference on Data Science in Cyberspace (DSC), Shenzhen, China, 2017, pp. 606-614, doi: 10.1109/DSC.2017.19.

[23] Mihai Antonescu, Mihaela Maliţa, "FFT on a Heterogeneous System with a General-Purpose Map-Scan Accelerator", at Romanian Journal of Information Science and Technology (ROMJIST), accepted for publication.

[24] Cooley, James W.; Tukey, John W. (1965). "An algorithm for the machine calculation of complex Fourier series". Mathematics of Computation, 1965, Vol 19 (no. 90): 297–301. doi:10.2307/2003354. JSTOR 2003354.

[25] Mario Garrido "A Survey on Pipelined FFT Hardware Architectures", Journal of Signal Processing Systems, vol 94, Jul. 2021.

[26] Konguvel Elango and Mu Kannan "A Survey on FFT/IFFT Processors for Next Generation Telecommunication Systems",Journal of Circuits, Systems and Computers, vol. 27, 03.2018.

[27] I. Lörentz, M. Maliţa, R. Andonie, "Fitting FFT onto an energy efficient massively parallel architecture" Proceedings of the Second 617 International Forum on NextGeneration Multicore/Manycore Technologies, IFMT '10, 8:1–8:11, 2010.

[28] Calin Bira, Liviu Gugu, Mihaela Malita, Gheorghe Stefan, "Maximizing the SIMD Behaviour in SPMD Engines", Proceedings of 619 the WCECS 2013, Oct 2013.

[29] M. Frigo and S. G. Johnson, "The Design and Implementation of FFTW3," in Proceedings of the IEEE, vol. 93, no. 2, pp. 216-231, Feb. 2005, doi: 10.1109/JPROC.2004.840301.

[30] Alexander Hurd, 2018, "fftw-cufftw-benchmark", Available: https://github.com/hurdad/fftw-cufftw-benchmark, Commit cfc8aa8, Accessed on 08.05.2023.

[31] AMD-Xilinx, PG109 (04.05.2022): Fast Fourier Transform v9.1 LogiCORE IP Product Guide, [online] Available at: https://docs.xilinx.com/r/en-US/pg109-xfft/Fast-Fourier-Transform-v9.1-LogiCORE-IP-Product-Guide, Accessed on 08.05.2023.

[32] Mihai Antonescu, Mihaela Maliţa, Gheorghe M. Ştefan "Latency Hiding of Log-Depth Scan and Reduce Networks in Heterogeneous Embedded Systems", 29th International Symposium for Design and Technology in Electronic Packaging (SIITME), 2023, IEEE conference proceedings, accepted for publication.