FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

Field-Configurable GPU

Pedro Rodrigues de Castro

Mestrado Integrado em Engenharia Eletrotécnica e de Computadores

Supervisor: Professor José Carlos Alves

July 20, 2021

Abstract

This thesis proposes a solution on the topic of high performance computing and the field of study of accelerated data processing.

One well known platform that targets this issue is the GPU. These devices present a SIMT (Single Instruction, Multiple Data) architecture which are ideal to process high amounts of data in parallelizable processes.

So, using them as an inspiration, a GPU micro-architecture was implemented on an FPGA in order to test if the performance would be promising.

In order to optimize as much of the resources of the FPGA (Field Programmable Gate Array) and the performance of the application, this micro-architecture can be reconfigured with different parameters and hardware modules using a software tool created for this project.

This software includes a compiler that interprets an assembly-like program and implements it according to the developed architecture. This way, only the resources needed to execute the application are allocated to the architecture.

To end the design process of this tool, the program constructs the Hardware Description Language (HDL) files with the reconfigured architecture for the given user specification.

The combination of the architecture with the program provides a reconfigurable and flexible platform that enables writing programs for a specific application, while still being easier to program and modify compared to manual HDL instantiating.

To test the design, an FCNN (Fully Connected Neural Network) was implemented using the project's design process and tools. Its performance was compared to other processors, as to obtain a baseline for the benchmark. The results showed a speedup of up to 80 and 7 times compared to an ARM and Intel processor, respectively.

Even though further optimizations could be done to this example in terms of performance, these results showed potential benefits in process acceleration, specially in the embedded systems area.

The tools developed also allow the use cases to scale up depending on the size of the application.

Resumo

Esta tese propõe uma solução no tópico de computação de alta performance e o campo de estudo de processamento de dados acelerado.

Um dos dispositivos mais conhecidos atualmente que é usado nesta área é a Unidade de Processamento Gráfico (GPU). Este é ideal para a aceleração de computações em tarefas que envolvem o processamento de muitos dados em paralelo. Isto é possível devido à sua microarquitectura SIMT (Instrução Única, Múltiplos Dados).

É por este motivo que este projeto toma esta plataforma como inspiração e tenta implementá-la num FPGA (*Field Programmable Gate Array*).

Tirando partido do poder de reconfiguração dos FPGAs, o projeto remove as partes de hardware que não serão usadas na aplicação do utilizador. Isto otimiza os recursos disponíveis do circuito integrado, tal como reduz os tempos de transmissão de dados entre registos.

A ferramenta desenvolvida no contexto da tese compila um programa, escrito de forma semelhante a *Assembly*, para a arquitetura proposta.

No fim do processo de design, a ferramenta constrói os ficheiros HDL (*Hardware Description Language*) que contêm os módulos necessários para a arquitetura conforme os detalhes dados pelo utilizador.

A combinação da arquitetura com o programa disponibiliza ao utilizador uma maior facilidade de modificações e de reconfiguração à sua aplicação comparado com o instanciamento manual.

O design foi testado com a implementação de uma Rede Neuronal Completamente Conectada (FCNN). Para ter uma referência de comparação de resultados, o mesmo algoritmo foi executado em dois processadores. A implementação em FPGA demonstrou uma aceleração do processo 80 e 7 vezes maior do que um processador ARM e Intel, respetivamente.

Apesar deste exemplo ter alguma margem para mais otimizações, os resultados obtidos já são promissores, especialmente na área de sistemas embebidos.

As ferramentas desenvolvidas também permitem o escalamento da aplicação, dependendo do seu tamanho.

Acknowledgements

I would like to thank my supervisor Prof. José Carlos Alves for his guidance and curiosity throughout all the thesis. I would also like to acknowledge the constant support and advises of my family and my colleagues.

Pedro Rodrigues de Castro

"What I cannot create, I do not understand"

Richard Feynman

Contents

List of Figures

List of Tables

Abbreviations and Symbols

ALU	Arithmetic Logic Unit
ASIC	Application Specific Integrated Circuit
AVX	Advanced Vector Extension
BCE	Basic Computing Element
CPU	Central Processing Unit
CUDA	Compute Unified Device Architecture
FCNN	Fully Connected Neural Network
FPGA	Field-Programmable Gate Array
GPGPU	General Purpose Graphical Processing Unit
GPU	Graphical Processing Unit
HDL	Hardware Description Language
IC	Integrated Circuit
ISA	Instruction Set Architecture
LUT	Look Up Table
MIPS	Microprocessor without Interlocked Pipelined Stages
MMU	Memory Management Unit
MNIST	Modified National Institute of Standards and Technology
PC	Program Counter
RAM	Random Access Memory
ROM	Read Only Memory
RTL	Register Transfer Level
SIMD	Single Instruction, Multiple Data
SIMT	Single Instruction, Multiple Threads
SM	Streaming Multiprocessor
SMT	Simultaneous Multi-threading

Chapter 1

Introduction

1.1 Context

Modern computing has come a long way since its foundation and, more then ever, depends on speed and reliability. As larger and more complex programs are created, the need to keep up with the performance requirements tighten. For this purpose, a area known as accelerated computing has become a topic more refined and complex to rival the constantly increasing demands of data processing.

Not only does this field of study investigate application acceleration through software, but it also contemplates new, or purpose specific, hardware implementations.

Such an implementation is the GPU (Graphical Processing Unit). This device is designed for the purpose of processing large amounts of graphical data. However, it is also possible to use it's power to compute other tasks that also require parallel data computations. In recent years, this gave rise to a new and dedicated device for this motive, the GPGPU (General Purpose GPU), where the graphic pipeline hardware was removed and some types of data processing was optimized.

Another well known devices for application acceleration are the FPGAs or the ASICs. These two make full use of the hardware configuration to output the intended results. Although, in terms of raw performance, the ASIC is superior to the FPGA, the price of fabrication is much larger if the desired batch size is small. Therefore, in this thesis, the main focus will be turned to the FPGA, as it is a cheaper platform for prototyping and testing.

The FPGA allows the reconfiguration of its own hardware structure. Meaning, the device may be reconfigured, fine tuned or re-purposed for other implementations, in contrast to the ASIC, which after fabrication cannot be rewired.

1.2 Objectives

The FPGA is a versatile device and the GPU is a well known architecture which enables the parallelization of calculations for large sums of data. Having mentioned these two devices, the aim of this thesis is to implement and evaluate the performance of a GPU architecture on an FPGA.

More specifically, as the intention is not to focus on graphic rendering, the micro-architecture in mind is the GPGPU, which removes the unnecessary graphical processing hardware.

However, this project also has the purpose of allowing for a more flexible execution of the GPGPU architecture. This could have an impact from adaptable data sizes, to reduced instruction sets for the ALUs (Arithmetic Logic Unit).

A program will interconnect the user's application program and the architecture, abstracting him from the the instruction set and the lower level implications.

1.3 Motivation

Having a reconfigurable architecture may tackle a few problems in computation and embedded systems. Making purpose specific architectures can improve significantly in terms of used area of the IC, speed and/or power consumption.

In applications where speed is key, the space available in the chip can be managed to enable the fastest performance possible. From creating an operation that would normally be unavailable in normal CPUs or GPUs, to repeating the same physical blocks to increment the data throughput.

Certain devices are more limited in area, being the available number of logic units or having to share space with another digital system in the same FPGA or ASIC. Having full control over the sizes of the physical blocks or the memory and bus sizes can help improve space efficiency.

Regarding the power consumption, most embedded systems have a tight restriction on battery life. Furthermore, with IoT (Internet of Things) devices being widely used for more, and more demanding, applications (such as edge computing) this should be kept in mind.

All these factors must be taken into account while implementing accelerated computing applications. However, the convenience and the comfort of a high level overview, much like the programming of a normal CPU application, must still be present in the tools to be developed.

Chapter 2

State of the Art

2.1 Accelerated Computing

With the rise of machine learning, computer vision and other big data applications, computation speed with high data throughput is a great concern. A small list of alternatives to tackle this subject will follow.

For specific applications, the best solution could always be considered the implementation of an ASIC. These are tailored to solve a specific problem that raises from a specific application. Its hardware is, in the best case, fully optimized for speed, space, power or all of those. However, once designed and fabricated, there is no way of adapting its configuration. This can even cause bugs to be deployed without having a way to fix them without replacing the component with a newly designed one.

Much like an ASIC, the FPGA can be configured for the specific task. Yet, it can be reconfigured and re-purposed, contrary to the ASIC. The structure of the FPGA is a grid of Configurable Logic Blocks (CLB), containing Look Up Tables, Flip-Flops and other components, that can be rewired to a specific pattern. Although it may not be able to achieve such a good performance compared to its counterpart, due to the overhead in chip area, it allows for a good prototyping platform.

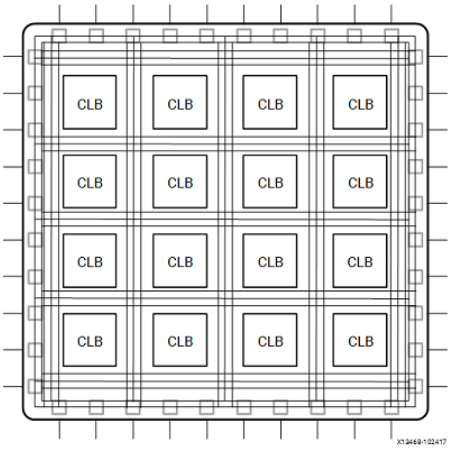

Figure 2.1: Simplified overview of the FPGA architecture [1]

The DSP (Digital Signal Processor) is a microprocessor that sees usage in audio and image processing, besides other communication tasks. It is a processor with hard-wired signal processing algorithms, optimizing the transistor use and clock cycles for the required operations. Nonetheless, it trades this optimizations for flexibility. In most cases, it is used as a pre/post-processor for other connected ICs.

In the specific area of machine learning and computer vision, a few solutions for have emerged recently, the TPU (Tensor Processing Unit) and the VPU (Visual Processing Unit), respectively. Both enable the implementation of some AI (Artificial Intelligence) algorithms and techniques to enhance the speed of the task. These units can present a similar micro-architecture to the GPU, as they are both SIMD (more on this topic will be presented latter on, 2.3). However, the way they are implemented may be restrictive for the algorithm needed. This is due to the hardware design of the devices, which might only be capable of performing a set of algorithms. Besides, the network parameters need to be quantized when compiling the program to the hardware, which may reduce the precision of the implementation [13, 14].

The GPU (or the GPGPU) is another platform with an architecture capable of performing data processing acceleration. The output bandwidth is very high for the same operation compared to a CPU. Its disadvantage is that, even though it has a programmable interface, it can't run purely sequential tasks without loosing its strength in performance.

Last but not least is the CPU. Up until a certain point in history, the CPU's speed was bound by the operations per clock cycle. This would mean that the speed would improve by decreasing the clock cycles taken to perform operations and by increasing the clock frequency, depending on both architectural and technological improvements.

Over the years, as Moore's law dictates, the speed was increasing. However, the speed of the CPUs were reaching their limits with only a single core due to the power wall problem. The heat dissipation of the chip could not keep up with the heating of the system, ending up damaging it, or even burning the IC. The industry resisted implementing multi-core systems, as it required a lot of effort, not only in terms of hardware, but also on more complex compilers.

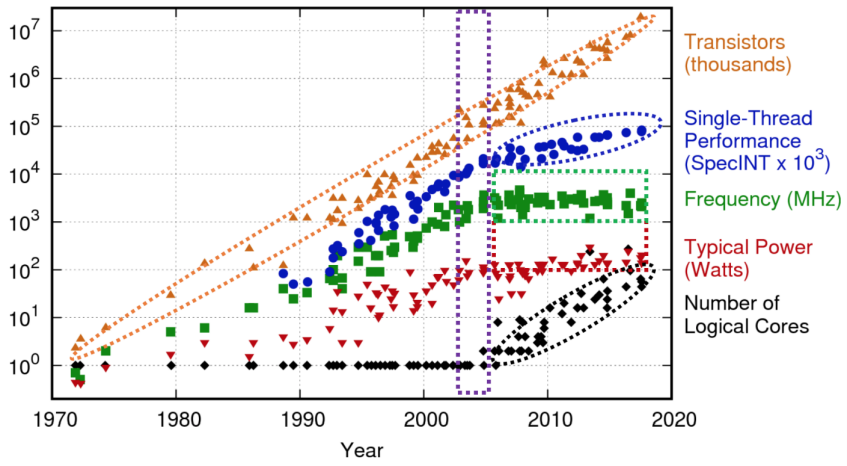

Figure 2.2: 42 Years of Microprocessor Trend Data [2]

Ultimately, the power wall was hit around 2004, which forced the trend of multi-core CPUs. The CPU became a generic processing unit with several cores to execute parallel tasks. This multi-core architecture can also be seen in modern GPUs and even DSPs.

The speedup gained from this multi-core system, however, is bounded by the executed program. The Amdahl's law, stated in Equation 2.1, bounds the maximum speedup gained in a multi-core system depending on the parallelizable execution time of a program, being N the number of cores and F the parallelizable part of the program (between 0 and 1, being 1 a fully parallelizable program).

$$Speedup = \frac{1}{(1-F)+\frac{F}{N}} \tag{2.1}$$

In modern computing, the cores themselves can have different computation powers. So, each chip is bounded by a N Base Core Equivalents (BCE) for all cores, having each core consuming R BCEs. Therefore, there are N/R physical cores per chip [3].

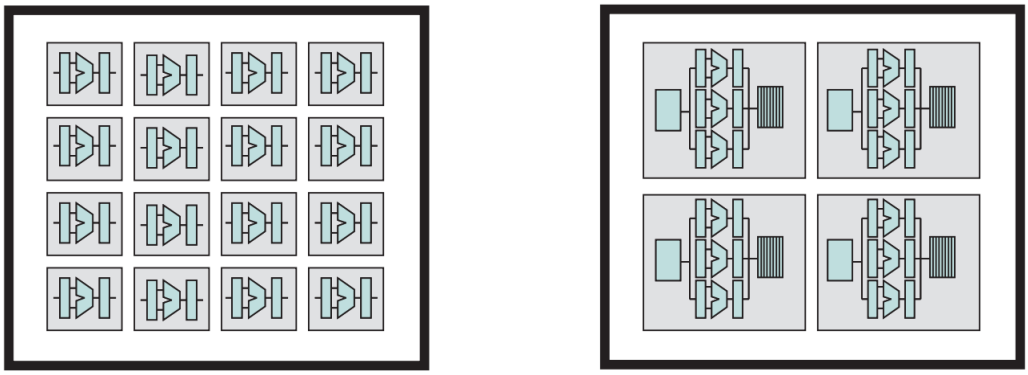

Figure 2.3: Left: Homogeneous sixteex 1-BCE cores. Right: Homogeneous four 4-BCE cores [3]

Note: This is a simplified schematic and assumes the area as the limiting resource.

In this case, the serial fraction of the program uses a core at rate $Perf(R)$. Typically, $Perf(R) = \sqrt{R}$ [3]. The parallelizable segment of the program uses N/R cores at rate $Perf(R)$ each. For now it is being considered that the BCEs are equally distributed amongst the physical cores, making them homogeneous (or symmetric) cores.

$$Serial\ time = \frac{1-F}{Perf(R)} \tag{2.2}$$

$$Parallel\ time = \frac{F}{Perf(R) \times \frac{N}{R}} = \frac{F \times R}{Perf(R) \times N} \tag{2.3}$$

With these new factors, Amdahl's law becomes:

$$Homogeneous\ Speedup = \frac{1}{\frac{1-F}{Perf(R)} + \frac{F \times R}{Perf(R) \times N}} \tag{2.4}$$

This is what makes the distinction between the GPU and the CPU processing power. Even though the GPU has more cores than the CPU, each has less power compared to the CPU cores.

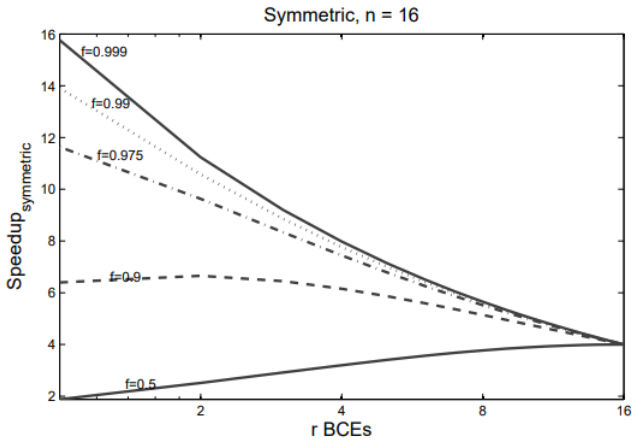

Figure 2.4: Example of the speedup achieved when distributing sixteen BCEs over the R value with different parallelization values [3]

The more parallelizable the program is, the more it benefits from having multiple cores. In contrast, the performance is degraded in programs with a serial execution compared to processors with fewer cores.

An alternative to the homogeneous multi-core chip is the heterogeneous (or asymmetric) architecture. This has a bigger core that performs the serial operation and distributes the remaining BCEs between other smaller cores with one BCE each.

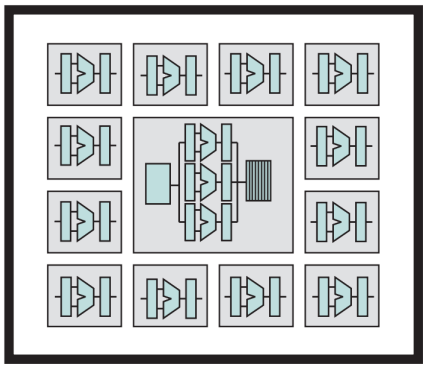

Figure 2.5: Heterogeneous one 4-BCE core and 12 1-BCE cores (Total of 16 BCEs) [3]

$$Heterogeneous\ Speedup = \frac{1}{\frac{1-F}{Perf(R)} + \frac{F}{Perf(R)+N-R}} \tag{2.5}$$

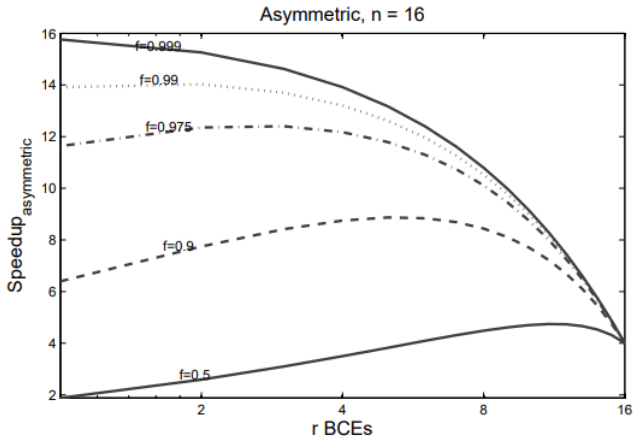

Figure 2.6: Example of the speedup achieved when splitting the sixteen BCEs over a larger core and several 1-BCE cores [3]

Depending on the BCE count and the parallelizable section of the program, a sweet spot in the distribution of the BCEs becomes apparent (for example, with $F = 0.9$ and $N = 16$, the optimized distribution is around $R = 7$). Furthermore, the overall performance of the heterogeneous architecture is better in comparison to the homogeneous.

More on this topic, there is a technique named dynamic multi-core chips, that bundles the BCEs in groups at run-time, in order to take full advantage of all the available cores at every scenario. With this, the speedup achievable is:

$$Dynamic\ Speedup = \frac{1}{\frac{1-F}{Perf(R)} + \frac{F}{N}} \tag{2.6}$$

If the hardware was ideal, this would be the most optimized case of all described above, as the own architecture can change in accordance to the current needs of the application.

2.1.1 Final remarks on Accelerated Computing

All the devices covered until now can be merged together, on the one hand, to cover up their disadvantages, and on the other hand, to make the most out of their strengths. In fact, this is common in High Performance Computing (HPC).

However, distributing different devices has a cost. As a matter of fact, one of the current major bottlenecks that is being faced is the memory bandwidth limitation. In other words, some applications' speed is not actually bound by the computation power, but by the memory transmission speed.

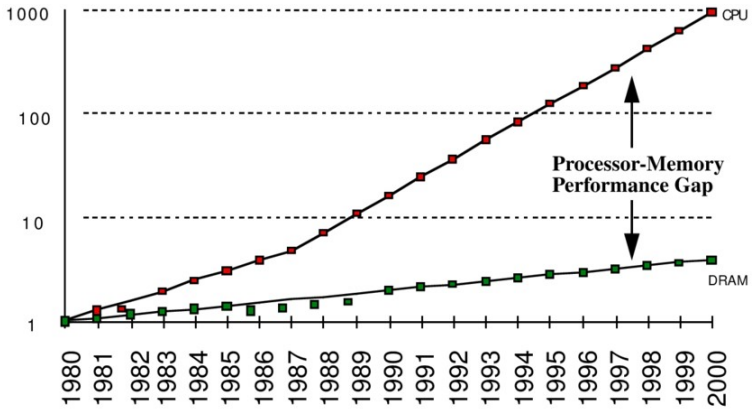

Figure 2.7: Processor-Memory performance gap [4]

This problem affects all memory transmissions within the same IC and becomes even more evident when integrating different chips communicating to each other. For this reason, in most accelerated computing applications, the constant exchange of information is generally avoided, and whenever possible, the memory closest to the processing units is used (namely the lower level caches and registers).

Technologies like the 2.5D or 3D packaging emerged to try to mitigate this issue, by implementing the different ICs in the same packaging [15]. This method reduces the time of data exchanges and increases bandwidth.

The roofline model attempts to specify if the application's speed limitation comes from memory bandwidth or the program's complexity [16, 17, 5]. In this model, the application is quantified into an arithmetic intensity in flops (floating point operations per second) per byte (Gflop/B). The bandwidth of the memory is given in gigabytes per second (GB/s) and the processing performance in Gflops. With this information, the plotted graphic (exemplified in Figure 2.8) shows that if the arithmetic intensity is lower than the intersection of the two lines, the program's performance is being bounded by the memory's bandwidth. If the arithmetic intensity is beyond that point, the program's limitation is the processor's speed.

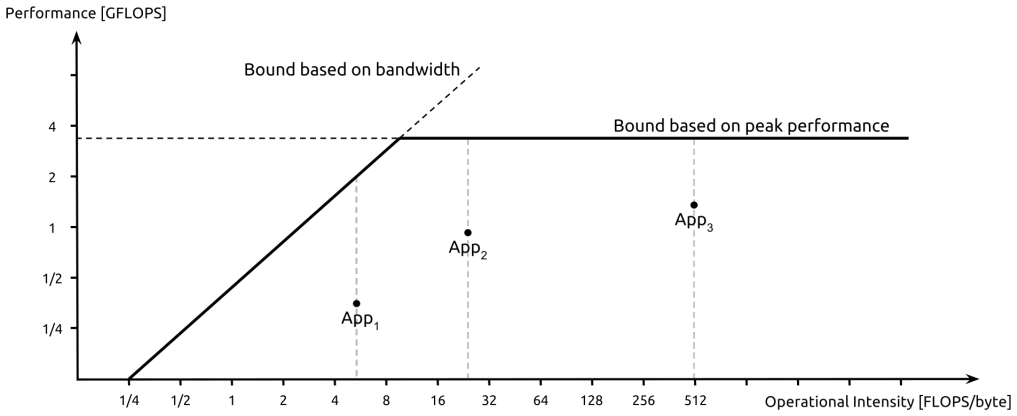

Figure 2.8: Example of the roofline model for 3 applications. App_1 is bounded by the memory bandwidth while the other two apps are bound by the processor's peak performance [5]

2.2 Processor Architecture

Before diving deeper in the GPU architecture, it should be made clearer how a processor functions and its core components. The contemplated architecture that will follow is the Harvard processor architecture due to it's simplicity. Furthermore, this type of design can be found in a traditional DSP architecture and other micro-controllers, which, considering the topic at hand, shows some relevance. Modern computers are based on a derived version of this architecture, using the some properties of the Von Neumann architecture, named modified Harvard architecture, which allows instruction memory to be accessed as data [18, 19].

The Harvard architecture distinguishes itself owing to its physical separation of instruction memory and data memory. This way, the program instructions are located in ROM (Read-Only Memory) while the data memory is located in RAM (Random Access Memory). This way, it is less likely that the program will suffer memory corruption due to the incorrect usage of the data memory. Furthermore, and more importantly, this distinction forces the existence of two separate buses, making it possible for the processor to fetch instructions and read/write data at the same time.

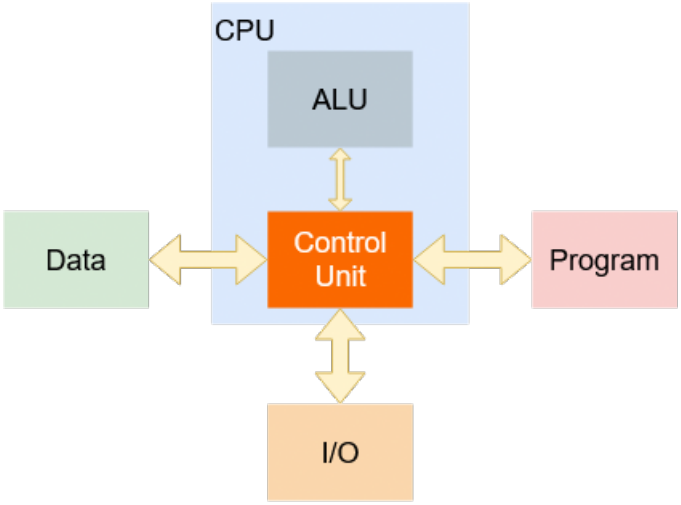

Figure 2.9: Harvard architecture overview

The control of the system is performed by a central unit, which performs the data or instruction fetches and stores/loads the needed information from/to the respective register. This registers reside in the ALU and are used by it to perform the operations or store the results. There are also special registers that store other information regarding the operation performed, such as the result of a conditional statement or the overflow of a mathematical operation. If the programmed application checks the state of this flags, it is the control unit's task to jump to the correct instruction address depending on the boolean value.

The jump operation is performed over the Program Counter (PC) register, which resides inside the control unit. In the normal operation of the program, at the execution of each instruction the PC is incremented to the next address. However, if the jump instruction is being performed, an address offset is added to the PC. This may depend on the architecture itself, as the PC might be set to a specific address instead of being added as an offset.

Besides the control flow instructions, such as the jump, performed in the control unit, the majority of the operations execute in the ALU. All these mentioned instructions are part of the ISA (Instruction Set Architecture). The instructions are decoded and only executed by the intended modules. This selection is normally done by multiplexers that enable the block which must perform the operation [6].

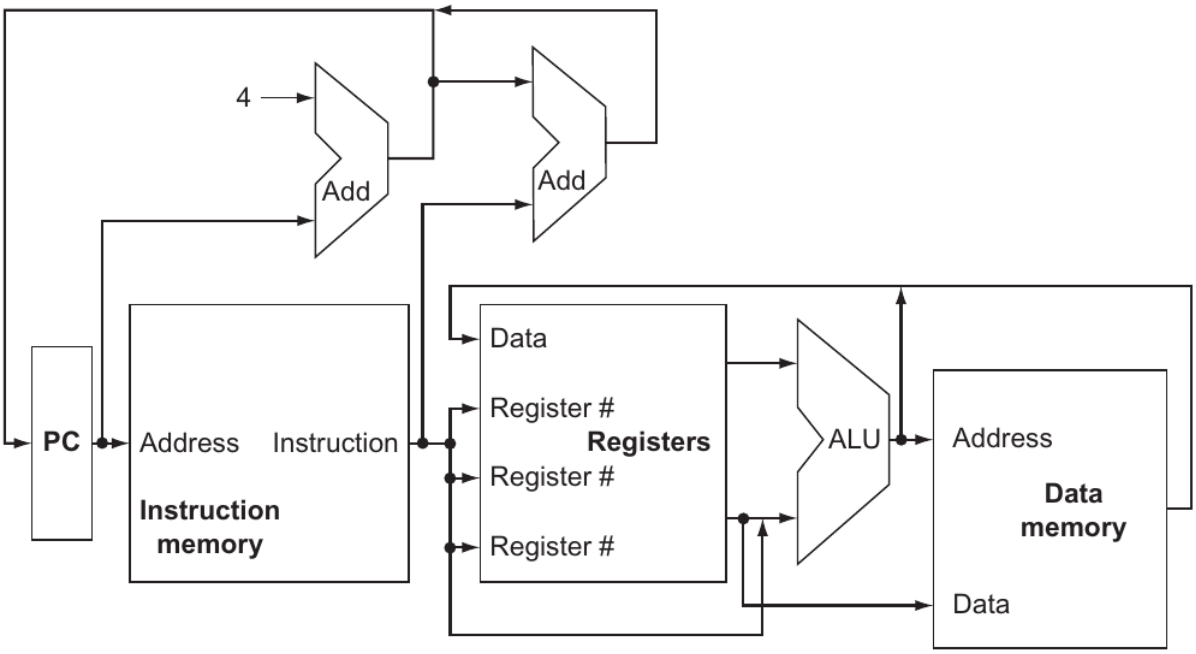

Figure 2.10: Abstract view of the implementation of the MIPS subset [6]

The ISA may have several classifications based on the implementation [12, 20, 21]:

- **Stack machines — 0-Operand**: The operations are performed on the elements on top of the stack.
- **Accumulator machines — 1-Operand**: The operations are performed on the top element of the stack and are implicitly accumulated on a special register.
- **General Purpose Registers (GPR)**:
  - **Register-Register (or Load-Store) — 3-Operand**: The data is moved from memory to the registers before the operation.
  - **Register-Memory — 2-Operand**: One operand is in memory when performing the operation while the other is already loaded to the register.
  - **Memory-Memory — 3-Operand**: The operation has to fetch all data when performing the instruction.

These implementations affect the performance or the program size. The example in Table 2.1 shows the program size depending on the implementation used, being the Memory-Memory the lowest in program memory usage. However, the Mem-Mem has to fetch instructions from memory, making it slower. The Reg-Reg has a bigger program size, but all data is in register when operating, making it faster than the previous. The Reg-Mem is the middle ground between those two. It should be noted that not all ISA classifications have only n number of operands in its instructions, as there are some operations that might need another number of operands. Besides, a computer architecture is not limited to only one of these types of instruction sets. It may be relevant to add different variants of operations to have programs aimed for various optimizations.

Table 2.1: Example of the $C = A + B$ operation

Stack	Accumulator	Reg-Reg	Reg-Mem	Mem-Mem
Push A	Load A	Load R1, A	Load R1, A	Add C, A, B
Push B	Add B	Load R2, B	Add R1, B	
Add	Store C	Add R3, R1, R2	Store C, R1	
Pop C		Store C, R3		

The usual instructions found in most ISAs include:

- **Load/Store instructions**: Move data between memory and registers.
- **Jump or branch instructions**: Jump to an address in the instruction memory. They can be unconditional jumps or a jump instruction that is only performed if a flag is set or if the operands check the condition.
- **Comparison instructions**: These instructions perform comparisons between registers and normally write to a flag register.
- **Boolean logic instructions**: These can be bitwise comparators, comparisons between registers or shift registers.
- **Mathematical instructions**: Such as additions, subtractions, multiplications, etc. The ISA may have different sets of instructions for different data types and sizes.

2.3 A Brief History of the GPU

As the name implies, the GPU's main purpose is for graphics processing and rendering. This processor is normally integrated in a graphics card. However, the first graphics cards were not the processing units of today.

Before having a dedicated device for graphic processing, this task was performed in a software pipeline (Figure 2.11) by the CPU. Not only would this generate lesser quality graphics, but also cause a heavier workload and potentially slowing the other processes down. For the hardware interface, the image display would resort to transistor-transistor logic (TTL) chips [22, 7].

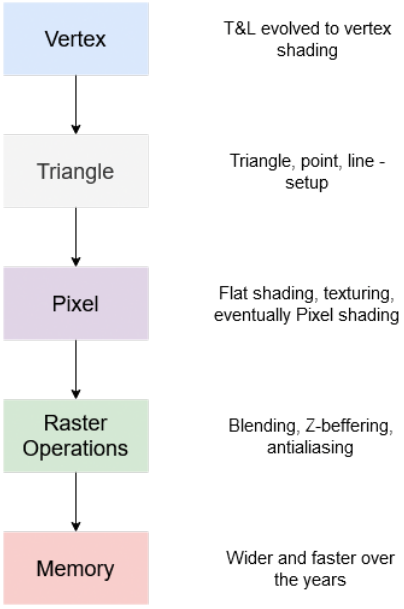

Figure 2.11: The traditional fixed-function graphics pipeline (Modified from [7])

To keep up with the demand for higher quality graphics, the pixel count was increasing. Moreover, the graphics pipeline would also evolve, adding new calculations to the already existing ones. These factors would increase the workload of the CPU even further.

It was in the '90s that a dedicated devices started appearing. Some of the latter stages of the pipeline were transferred to hardware. However, the CPU was still required for the first stages.

As the evolution continued, some segments of the pipeline started to be divided into smaller sections, taking advantage of parallelizable calculations such as the texture [7].

It was not until 1999 that the pipeline was fully implemented in hardware. From this point onward, the pipeline itself started evolving, adding new stages and parallelizations to the process.

In 2001, the pipeline was made programmable. In conjunction with the graphics hardware, the programmers had the option to program segments of the pipeline. Only one year latter, the first true programmable GPU was created. This featured per pixel operations, allowing fine grain customization of the image processing [7]. Nonetheless, the tools to program this devices were crude and unrefined, often resorting to modified graphical APIs for the specific application. Needless to mention that this tools made it hard to take full advantage of the board.

2.4 The modern GPU Architecure

Note: Before proceeding with this section, the terms regarding the GPU micro-architecture components will normally be expressions coined by NVIDIA. This is due to the fact that they are the most commonly found idioms when referring to the subject. More can be found in the Appendix A.

Finally, in 2006, the last step towards the architecture of the modern GPU was done. The featured Unified Shader Design (Figure 2.12) transformed the device into a new and more generic micro-architecture.

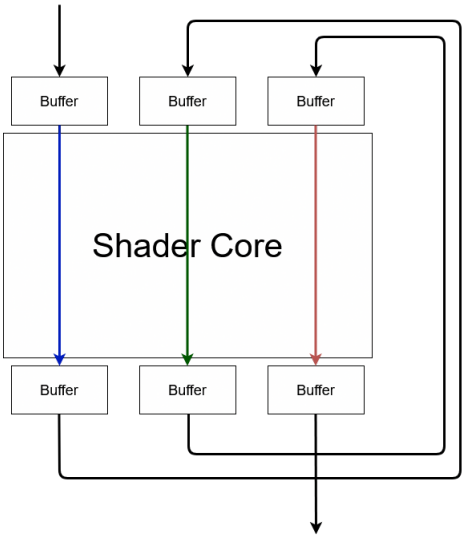

Figure 2.12: Unified hardware shader design (Modified from [7])

The main feature that allows the modern GPUs to achieve such high throughput is the SIMD (Single Instruction, Multiple Data) micro-architecture. As the name implies, a single instruction of the program executes in several ALUs concurrently, outputting the same number of data elements as ALUs. These processing cores, called CUDA cores by NVIDIA, are capable of generic calculations similar to the instructions commonly found in an CPU.

Figure 2.13: NVIDIA®'s Streaming Multiprocessor [8]

The Unified Shader Architecture groups several CUDA cores in lots. These lots are called streaming multiprocessors (SM), and the cores inside it share the same SIMD instruction. A program, or thread, that executes inside the SM is called a warp (or wave front in AMD's terms [23, 24]). The GPU has several SMs, enabling it to have multiple programs independent of each other, or segments of the same program.

In the Figure 3.2, the tensor cores shown are proprietary hardware that performs certain operations faster to the detriment of precision and is used in modern NVIDIA GPUs.

2.4.1 Memory

The GPU has several levels of memory. This has two reasons, the first being the scope of the layer(s) that should have access to it. The second is due to speed limitations caused by the distance between the memory and the component accessing it.

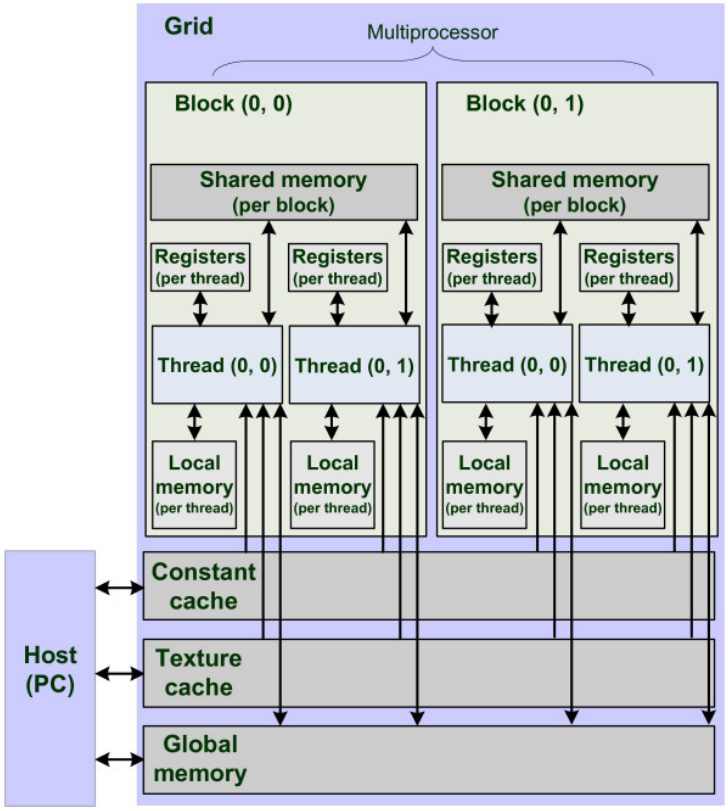

Figure 2.14: CUDA Memory hierarchy [9]

The memory levels ordered by access speed (from faster to slower) in a GPU are the following:

- **Registers**: Only available for the individual core.
- **Shared memory or level 1 (L1) cache**: Accessible by all cores within a SM.
- **Local memory**: Memory accessible by a single thread (or core). However, it is not on chip memory, making it slower than shared memory.
- **Constant memory**: Read-only memory, resides in device memory.
- **Texture memory**: Read-only memory, resides in device memory and is optimized for 2D spacial locality.
- **Global memory**: Shared by all the Streaming Multiprocessors.

It is worthy to mention that there are differences in bandwidth between layers. The order of bandwidth from larger to smallest is: Shared memory/L1 cache, Texture/constant memory, Global memory, Local memory, Registers. The throughput of the global memory may vary depending on the device capabilities, however, making it less precise to order the bandwidth [25].

2.4.2 The GPGPU

The final transition specialized in accelerated computation was done in 2010 with the advancement of the GPGPU. Now, the platform was set for big data applications, without the graphical specific hardware. This modification alleviated some chip space, allowing the placement of more cores for more data processing or more memory.

More importantly, the cores of the GPGPU (and the GPU) evolved from a SIMD micro-architecture to a SIMT (Single Instruction, Multiple Threads). This meant that the cores are more developed, and don't need to run the exact same instructions. However, the execution of branching instructions may aggravate the performance of the device.

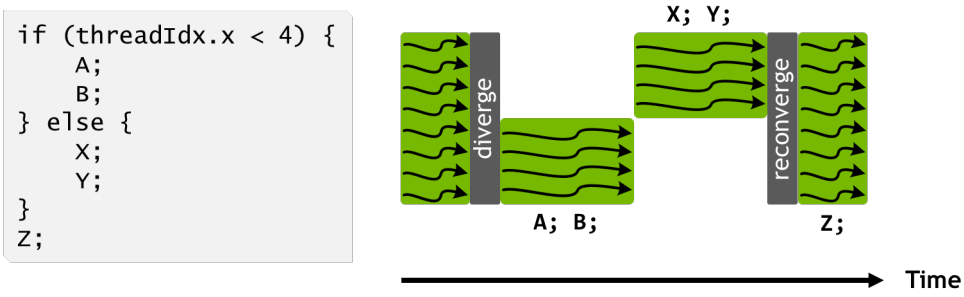

Figure 2.15: Branching effect [8]

If a program has a conditional statement that would cause the threads within a warp to branch, some of the cores will have to remain inactive until the other cores finish executing their conditional statement. This may cause a considerable slow down in the program execution. This is the reason why it should be avoided to use conditional statements in GPU programs.

2.5 Hardware Improvement Techniques

2.5.1 Speed improvements

This section will discuss some methods to improve the speed in applications data show a lot of Data-Level Parallelism (DLP).

2.5.2 Pre-processing

A simple but very effective technique is to pre-process the incoming data with dedicated hardware. This may very from application to application, and the computations performed may range from simple signal processing, to more complex data paths.

As this is just a method, it can be implemented as purely combinational circuits to sequential circuits, as shown with the DSP. The task of this component is not to process the data completely, but to simplify part of the input into more digestible data for the rest of the hardware.

2.5.3 SIMD

The SIMD architecture previously mentioned when discussing the GPU's architecture was one of its variations. There are two variations in regards to this topic [12].

The first one, is an execution pipeline of data operations. It is easier to compile and understand then its counterparts. However, when conceived, it was considered too expensive for microprocessors due to the cost of sufficient DRAM bandwidth.

The second variation is called a Vector Extension. Nowadays, it is found in most instruction sets of devices which support multimedia applications. Currently more known as AVX (Advanced Vector Extension) in the x86 architecture, it started as MMX (Multimedia Extensions) in 1996, and was followed by SSE (Streaming SIMD Extensions) shown on Figure 2.16. Other architectures like ARM or RISC-V present their own version of vector extension as SVE (Scalable Vector Extension) and RVV (RISC-V Vector Extension), respectively [12, 8, 26, 27].

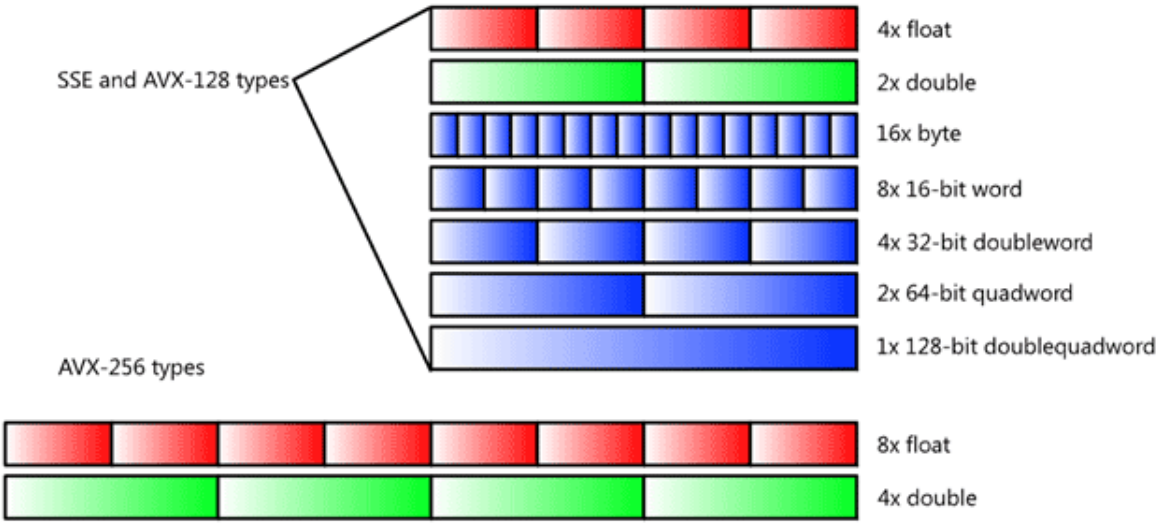

Figure 2.16: Intel® AVX and Intel® SSE data types [8]

This method takes advantage of the already implemented registers of the ALU. It creates an array in said register with smaller sized data types. It performs the operation needed, generating an output register that will also contain the results in an array (Figure 2.17).

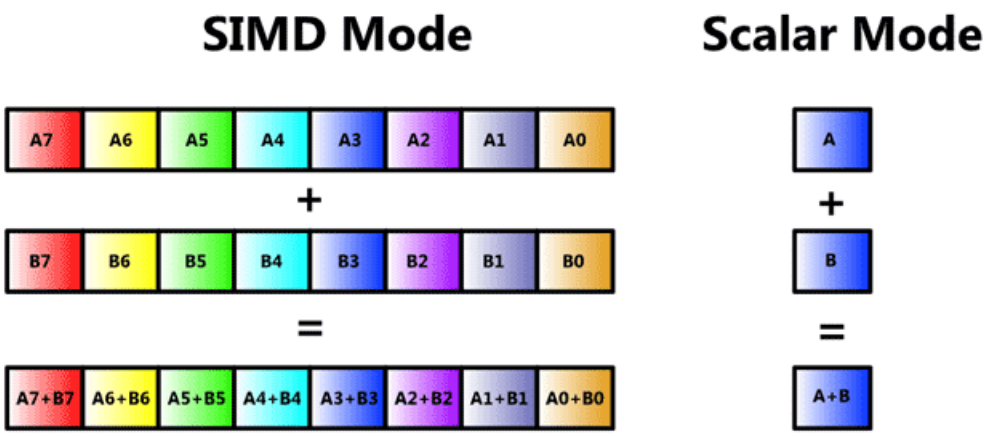

Figure 2.17: SIMD versus scalar operations [8]

This method is implemented with its own set of instructions, and cannot be performed in unsupported hardware. This means that the hardware used for the array calculations is not the same as with the scalar operations. This certainly adds performance in regards to speed, but it trades it for chip area.

It should be kept in mind that this technique is not mutually exclusive with SIMT and SMT, as it can complement them depending on the hardware design and further improve data-level parallelism processing.

2.5.4 SMT

The SMT (Simultaneous Multi-Threading) provides self-contained cores that can run independently from each other and separate the tasks of each executing thread. Not only can this bring faster processing speeds for data-level parallelism, but also for tasks. This is currently seen in most CPU's. Even though it requires more complex operative systems (with complex and, preferably, scalable schedulers) and compilers, it has improved the speedup of computation in the past few decades, with this topic still in development [28].

2.5.5 SIMT

The SIMT (Single Instruction, Multiple Threads) can be considered an hybrid of the SIMD and the SMT. Although the cores execute the same instruction, they may be able to branch and halt execution while the other threads execute the necessary branching program. Although this might be more inefficient in resource usage, it makes the system more adaptable.

This type of implementation can be seen in modern GPGPU's as previously mentioned in 2.4.2.

2.5.6 Superscalar Processors

The superscalar processor enables the execution of several instructions at a time, accelerating the execution of the program. This also enables the Out-of-Order execution (also know as OoO) which

mitigates the slower memory speeds. As shown in Figure 2.7, the major bottleneck in computing is the memory speed compared to processing speed [29, 30].

With In-Order execution, this would mean the ALU would have to wait while the registers did not have the new data.

In contrast, the Out-of-Order execution executes instructions if the memory is available, and only has to wait in cases where the needed memory is still being fetched and no operation can be done in the mean time. This can be achieved through a combination of the organization of the instructions made at compile time and the supporting hardware, which must be able to execute more than one instruction in parallel.

In modern CPUs, this OoO execution is also achieved by making the CPU itself reorganize the execution of the instructions, making it less dependent on the compiler and more reliable when having several processes running in the same machine.

2.5.7 DSP in the FPGA

Most FPGA's already have built in DSPs to speedup some very common applications. As a matter of fact, most neural networks and computer vision applications need the addition and multiplication operations. To accelerate the process and reduce consumed area of the application in the FPGA, the designers of these chips integrate a set number of DSPs, depending on the model. The operation of the DSP is translated to the following equation:

$$P = B \times (A + B) + C \tag{2.7}$$

However, this can also be reconfigured, depending on the FPGA, replacing additions with subtractions or a self feeding loop, with $C = P$. This is better visualized in Figure 2.18.

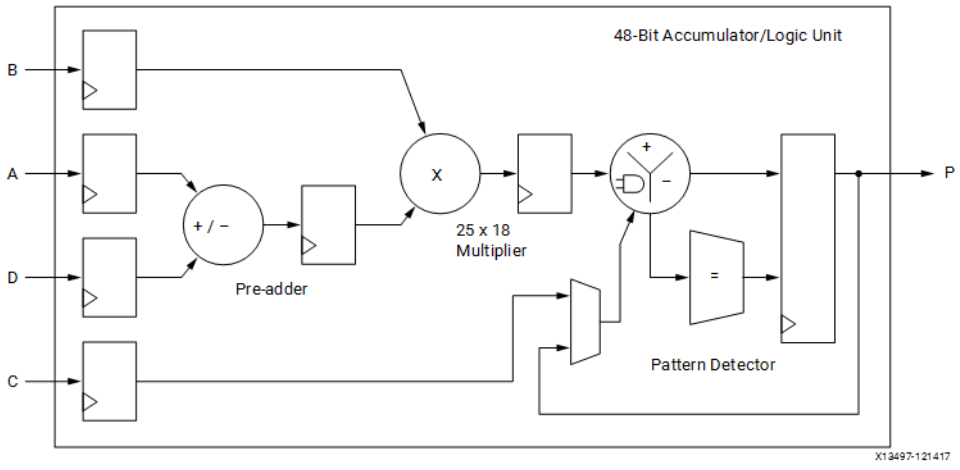

Figure 2.18: Xilinx® DSP48 block [1]

2.6 Power Consumption Improvements

There are two approaches when dealing with power consumption. The first is when designing the architecture, and the second is performed at run-time. The run-time solutions also need supporting hardware in order to operate, but change dynamically depending on the run-time conditions.

Some of the techniques presented next are basic techniques used in digital design overall, while others are more complex and specific for sequential processing units.

2.6.1 Design/Fabrication low-power techniques

2.6.1.1 Module based power reduction

A very common method is to pipeline the processing in several stages. In this context of processing units, there can be two types of pipelining at different levels of abstraction. At the lower level, the instructions or the pre-computation can be pipelined, meaning part of the operation is performed at each clock cycle until the final stage is reached. At the higher level of abstraction, the SIMD (or SIMT/SMT) cores can perform the operations like previously, passing the data from core to core.

The pipeline method has several advantages. Firstly, it can allow a higher clock speed in instructions where the propagation delay was a bottleneck. And second, it can improve the power consumption, as their will be less dynamic power consumption from the switching transistors.

Pipelines in processors have a disadvantage, however. Modern pipelines use speculative execution to attempt to predict the branching in the execution. In addition to the increase in complexity, if the prediction result is incorrect, the pipeline stalls and energy is wasted [31].

Another method is the parallel replication of a block and the desynchronization of each. This means that each block takes turns when processing the data at every clock cycle, allowing to greatly reduce the clock frequency. By copying several similar blocks, the occupied area is increased. This technique can also be implemented in conjunction with the pipeline, creating a hybrid. However, when dealing with multi-core architectures, this can be achieved at a higher level of implementation with scheduling of the warp execution, making it less relevant in this specific field of study.

Either integrated in the same IC or connected to the main chip with an interconnection, the pre-computation of incoming (or outgoing) data, using combinational or sequential logic, not only reduces the power consumption, but also the processing speed.

Most circuits, and especially processors, don't have all parts of circuit operating at all times. So, a method to reduce the power consumed is gating the clock, that is, turning of the clock to the components that are inactive, reducing the energy lost in switching the clock line [32].

Lastly, when executing a program, the ALU might not need the result of an operation immediately after dispatching the instruction. As such, the ALU could have what is called slack scheduling, allowing some instructions to take more than a clock cycle to perform while it runs other instructions of the program. There can be several ways to achieve this, including a smaller frequency for the hardware executing the slower operation or a pipeline, as previously discussed [33, 34, 35].

2.6.1.2 Interconnect power reduction

The transmission of data across a chip is also a factor to consider when optimizing the energy efficiency of the system [10].

Normally, the transmission of data is done in buses that have components attached and listening to it. But, most data exchanges are done from one component to another, meaning, there is no need for the other components to listen for data in the bus if it is not addressed to them. By segmenting the bus into several parts and disconnecting the lines, if the transmission is not needed beyond to a certain point of the line, energy consumption can be reduced.

Another alternative is to create hardware interfaces between communicating modules that splits the signal into two lines, each with a complementary differential voltage compared to one another. This voltage is also considerably lower than the rest of the chip, hence the better results in power consumption. This technique is called a low-swing bus. Still, when considering its implementation in an FPGA, there is no control over this component, being the fabricator's decision to add such a feature.

When a line on the bus switches states, the line's capacity needs to be charged/discharged. In common implementations, the line is connected to the source or ground. However, with the adiabatic bus, some charge can be recycled by redirecting the charge from a line that is transitioning from 1 to 0, to a line that is transitioning from 0 to 1. Even if not all charge can be reused, the improvement is significant. The most energy is saved when an equal number of lines rise and fall at the same time.

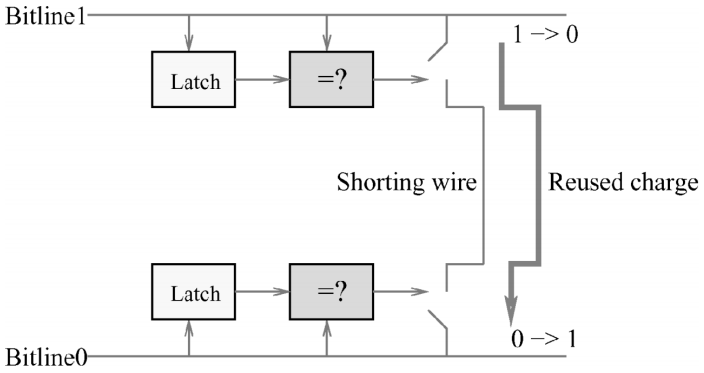

Figure 2.19: Two lines switching states in an adiabatic bus [10]

2.6.2 Run-time low-power techniques

Depending on the current executing program in a processor, it may not be justifiable to run at maximum processing power. In order to improve this, power management units can be implemented [10, 36, 37].

One way of doing this is to simply turn off any component that is not being used, (much like clock gating, but this time completely blocking the power). This could bring an additional delay whenever the turned off module would need to be repowered.

Another approach is to scale the frequency and the voltage supply of the component. There are three types of power losses in the CMOS technology.

- **Energy lost through leakage currents**: Highly dependent on the specific CMOS technology and size
- **Power lost through short circuit**: When the transistors in a logic gate are switching from one state to another, due to the ON/OFF time of the transistors, a short circuit happens.
- **Dissipated power from switching**: Energy lost when charging or discharging the load's capacitance.

The last two mentioned power losses happen dynamically, meaning it depends on how many logic transitions the systems has overall. The approximate and simplified equation for the power lost from dynamic operation is:

$$Dynamic\ Power \approx V_{DD}^2 \times f_{CLK} \times C_{Load} \tag{2.8}$$

So, the system depends on both the clock frequency and the supply voltage. There is however a limitation on the modification of these two parameters. First and foremost, the voltage cannot be lower than the threshold of the given technology. Then, if the voltage is lowered, it should be noted that the clock frequency also needs to lower. This happens because voltage reduction increases the delay [38, 39].

Because of this, as the supply voltage decreases, the power decreases as well, with the increase in the time per operation (TOP).

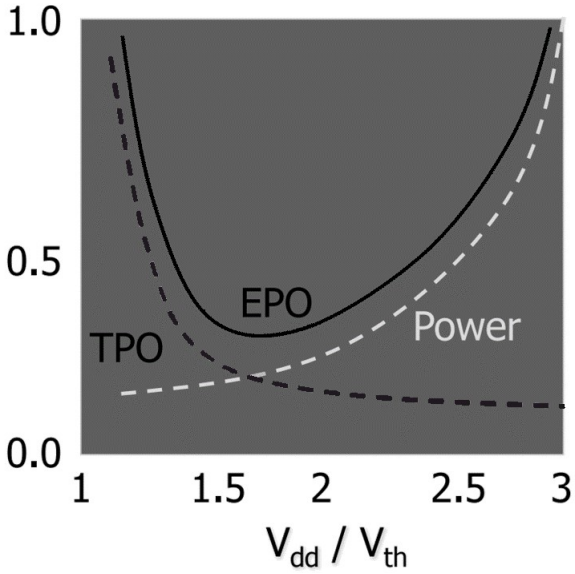

Figure 2.20: The EPO (Energy per Operation) in function of the ratio of the power supply voltage and the technology's threshold voltage

These types of run-time power management can be inserted at different levels. At a lower level, a local management unit can be used to control the specific operation of a single module. At a higher layer, a global management unit can regulate or send information to the local management units to control groups (or individual) cores, processors or cluster of processors.

Chapter 3

Architecture

Inspecting the problem at hand the steps to create a solution should be the following

1. Create and test a sample architecture for the intended purpose of the project
2. Generate RTL architecture files from a user's program and parameters

As in any computer architecture, the first step should be to create the basic structure while having into consideration the instruction set that will be implemented in it. This allows to better organize what should be needed from the hardware, and better adapt the resources demanded to perform those operations.

As the context of the thesis is to implement the architecture into an FPGA, it should also be noted that the resources used in the architecture include already implemented blocks, such as DSPs and RAM/ROM. These help improve the speed of the architecture, as purpose specific components already integrated into FPGA's board are more performant than replicating them with the Look Up Tables (LUT).

In order to achieve a more flexible design capable of being as generic as possible, the intended architecture for this project was SIMT. This makes it possible to have branching executions of the different threads running in the cores. Although this is more convenient and advantageous in certain cases, the general rule of branching in SIMT architecture still applies, being that branching should be used as little as possible.

In the first iteration of the project, a few basic blocks were defined. These include the main control unit, that decodes the instruction, and the core itself.

3.1 First iteration

In the first iteration there were a few flaws that would limit the architecture in terms of performance.

The main focus of the control unit was to decode the instructions from ROM and to handle the program counter, being just increment it or loading it with and address to jump to.

This would mean that the core activation management was made inside each core. In comparison to a centralized management, sharing information between each core and the control unit will need more wiring. This is not the best solution in order to make the project scalable, as it will require more logic to perform the same operations.

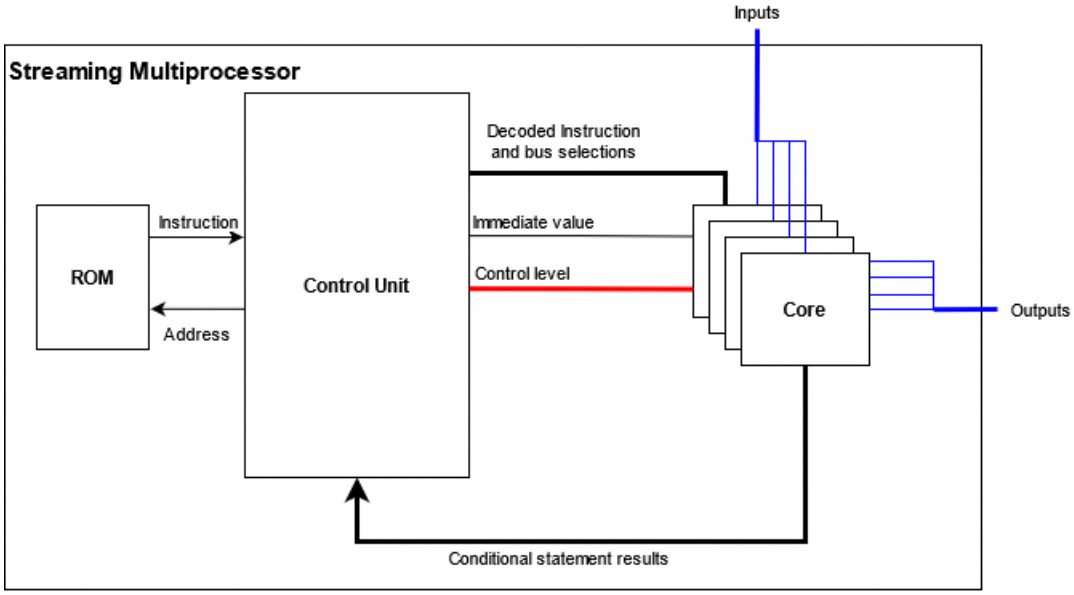

Figure 3.1: Streaming Multiprocessor schematic, first iteration

The branch control was performed with a level counter. For each new branch, the level would increase in the control unit. Each core would keep track of its own branching level. For example, if a core stopped executing in the level 4, it would remain inactive while the level in the control unit was higher than 4. Each core would also communicate to the control unit if it was available to perform a jump. That way, if every core agreed to perform the jump, no unnecessary branch would be read, avoiding wasted time reading instructions that would not be used.

The branching implementation, however, had a major flaw. When returning to a previous branch, the progress that a core had already executed within that branch was lost. This would mean constant repetition of conditional statements to verify the same conditions tested for the previous branches and slow down the execution.

The first iteration was also intended to be used mostly with 3 operands in the instructions (2 for the operands and 1 for the register to store the result). However, this restricts the usage of some operations in the present DSP unit of the FPGA. In case any MACC operations should be required, the last register of the DSP needs to be active . Instead of using external registers to the DSP for this purpose, it is more performant and convenient to use the internal register at the output signal.

3.2 Final version

From the first iteration, a few points were improved. Firstly, the cores had to be made as lightweight as possible. This turns the project easier to scale up, as doing this would only involve increasing the number of cores (of course, still taking into account the resources for the target FPGA). So,

by removing the branching from the core, the central control unit has the information associated with every core to decide if a branch is necessary or if a jump should occur. This topic will be discussed in further detail in the subsection 3.3.

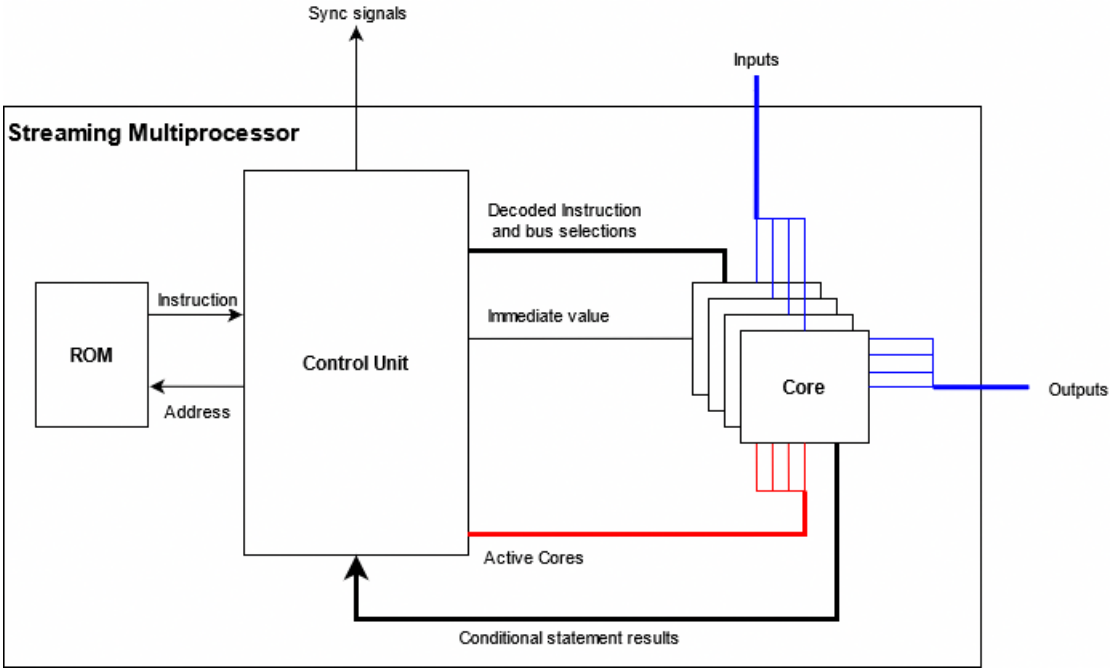

Figure 3.2: Streaming Multiprocessor schematic

When a core is inactive, the instructions and the write enable signals are turned off. This is the effect of having transparent latches, which are enabled simply by the active core signal corresponding to the thread. Moreover, when a core is also inactive, the comparator signals the control unit has if the core is available to jump, has it would not perform any operation either way.

To have a method of synchronization between outside structures and the Streaming Multiprocessor (SM), a signal of N bits were created. With the current instructions available, only one bit is active per instruction, and the bit remains in a high state for exactly one clock cycle.

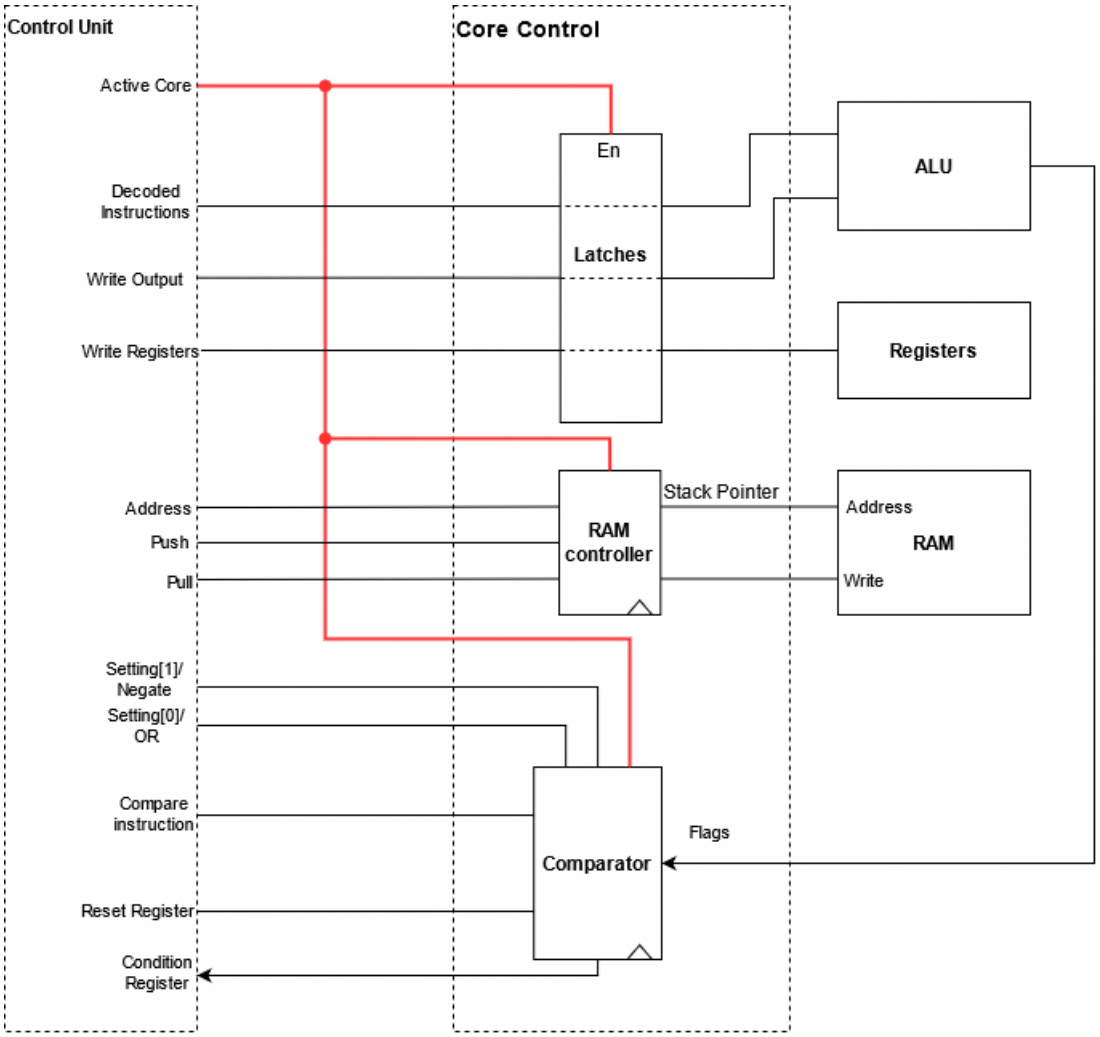

Figure 3.3: Control flow interactions with the Core

It was also important to have more than one memory type. In this new iteration, a RAM was added to each core. The individual RAM control is provided through the Core Control Unit, instead of the main one. This way, the core only inserts or modifies its RAM content if it is active. The stack pointer is also individual to each core, meaning that when a branch occurs, the stack may increase/decrease differently to each core. It should be noted however, when using this, stack consistency should be kept in check when writing a program. Otherwise, it may lead to the stack overflowing.

More on this point, as the implementation is application specific and the parameters are fine tuned by the user, no memory protection was implemented. So, elements can be pushed onto the stack when it is completely full. This will cause the stack pointer to return to beginning of the stack, which may lead to the impossibility to pop previous elements. However, if the right stack size is specified by the user the given application, and with the right memory consistency program, this should not pose a problem. Besides, this reduces the amount of logic per core, ever so slightly.

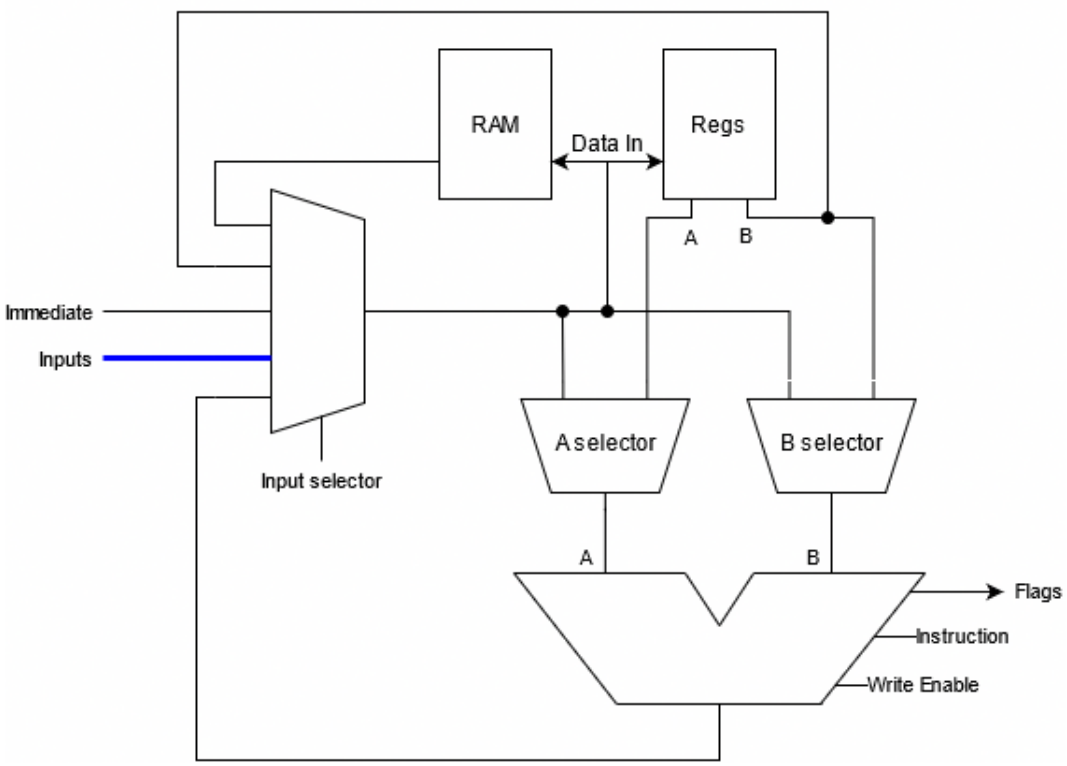

Figure 3.4: Data flow of a Core

With this design, the core consists of components like memories, multiplexers, a few control logic and the ALU. In conjunction with the generator program, the unused memory types are also removed or reduced to save as much space per core as possible.

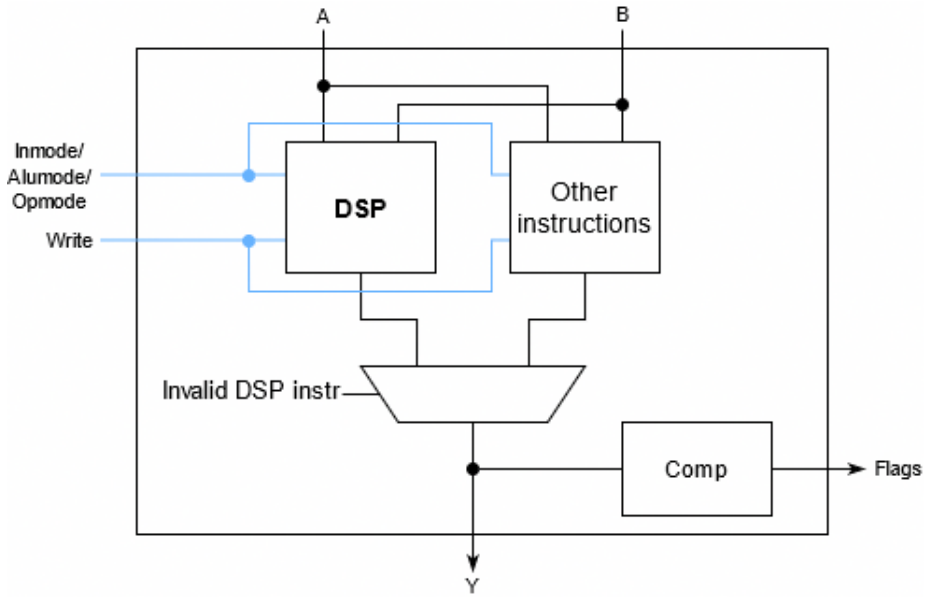

Figure 3.5: ALU schematic

Another stronger point for this architecture's flexibility is the input multiplexer configuration. There is a multiplexer that selects a data source to the data bus. With this, operations can be

performed with input or RAM data, sparing a few clock cycles to load the data into registers. The line is defaulted to the ALU output, as it should be one of the most important data.

Table 3.1: Opmode values and respective outputs for the DSP48E1[11]

Opmode							**Output**
6	5	4	3	2	1	0	
Z			Y		X		
xxx			xx		00		0
xxx			01		01		Mult
xxx			xx		10		Output (P)
xxx			xx		11		A:B
xxx			00		xx		0
xxx			01		01		Mult
xxx			10		xx		All bits to 1
xxx			11		xx		C
000			xx		xx		0
001			xx		xx		PCIN
010			xx		xx		Output (P)
011			xx		xx		C
100			10		00		Output (P)
101			xx		xx		17-bit shift (PCIN)
110			xx		xx		17-bit shift (Output)
111			xx		xx		xx

Note: Some operations require specific conditions to be met.

Looking at the DSP operation mode codes in Table 3.1, there is one case where the behaviour is undefined (Opmode[6:3]=111). As such, it is possible to take advantage of this extra unused instructions to provide more operations to the ALU without using extra signals. This is why the DSP is a component of the overall ALU. It is now possible to insert new and custom operations to the ALU in a separate file.

Another product of this decision is the usage of a single register in the control unit for the Inmode, Opmode and Alumode. Although the DSP has available registers to store these signals, they were disabled in order to use all the instructions available in the ALU, and to create consistency across the cores and the ALU.

3.3 Branching

One of the main features of this design is the integration of a SIMT architecture. This enables the usage of conditionals and execution of different instructions depending on the information of each core altogether.

It should be noted that compared to the case of a single core architecture, the branching doesn't have the same implications as in a SMT architecture. While in a single core architecture, a branch would mean the program counter would jump to an address in case a condition was met, in a SIMT, this would only occur if all the cores are in an agreement. If not, the cores that signaled they were available to jump would need to be turned off, while the others execute the instructions.

To optimize the FPGA resources as much as possible, an individual stack was made for the branching and to jump to subroutines. As the width of the addresses and the number of cores are very likely to be different, it is better to save memory space by separating them. It is also simpler to implement the control for such a structure.

Before performing the branching, a setup instruction necessitates to be performed, initializing the branching stack. The width of the stack corresponds to 2 times the number of cores present in the design.

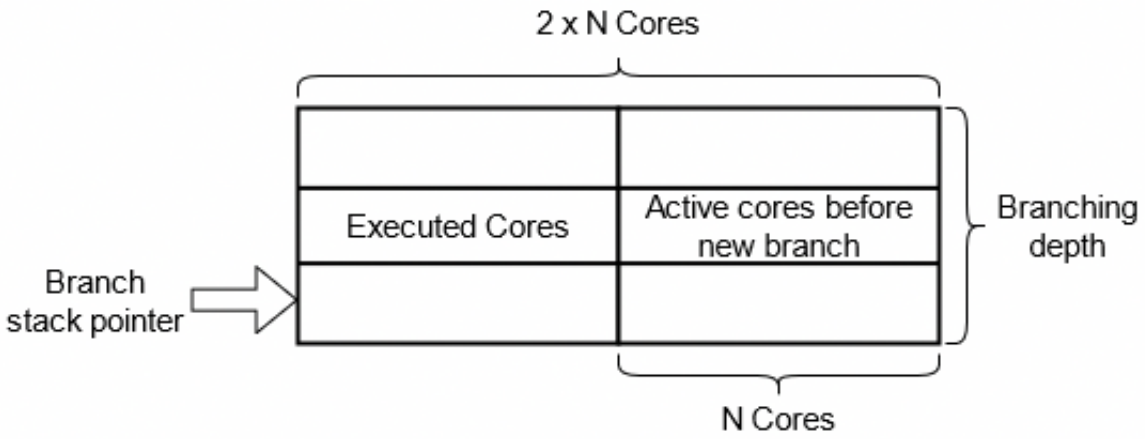

Figure 3.6: Branching stack

On the right side, the currently active cores are pushed onto it when setting up the procedure. With this information, it is possible to have nested branches, as the information before the branch began is now available.

On the left side of the stack, are the cores that have already executed a segment of the branch. This emulates an if/else statement. If the first conditions to jump have not been met by a certain core, this core executes the following instructions. This core then becomes inactive, ignoring any other execution of the subsequent branches.

Any time there is an update on the branching, it is checked if the left side of the top of stack is the same as the right side. This case means that all the cores that branched have do not need to perform any other operation. If this condition is met, a jump is performed to the end of the branching were a final instructions pops the stack, and decrements the pointer (if the address is already at the end of the stack it sets the empty register). At this point, the active cores are the same as the ones before the branching.

A program segment that would use the branching would look something like the Listing 3.1. The syntax will be detailed in a latter point, but this is presented here to show the functionality of the branches.

```
    new_branch
    add             reg_0, reg_1
    compare         $0, !negative
    add             reg_2, reg_3
    compare         $0, !negative
    branch          Branch_2
Branch_1:
    ...
    update_branch   End_of_branch
    compare         $0, !zero
    branch          Branch_3
Branch_2:
    ...
    update_branch   End_of_branch
Branch_3:
    ...
End_of_branch:
    end_branch
```

Listing 3.1: Example of a program with 3 branches

The previous program has 3 branches and acts similarly to an "if"/"else if"/"else" statement. When performing the compare instructions, it is possible to mix normal instructions as the conditional register will hold its value until a branch is executed.

After a branch is executed, an update instruction checks if all the cores are available to jump to the end of the branching, closing it and resuming the program. The last branch should not need to run this instruction.

The conditions for the "else if" should be written after the update instruction, followed by another branch to the next "else" address.

3.4 Subroutine execution

A common occurrence in computer programming is a repetition of the same structure of code. To avoid this, and to save some ROM space, a subroutine system was implemented.

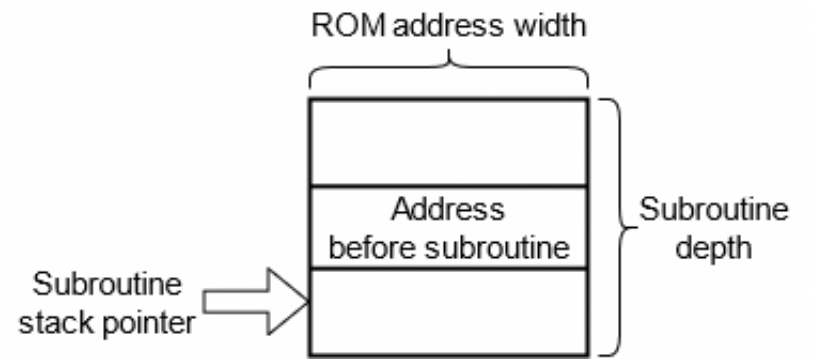

Figure 3.7: Subroutine stack

The subroutine jump is a simpler stack implementation compared to the branching one. Its respective stack stores the address where a subroutine was called. Every subroutine should have a return from subroutine instruction at the end, which pops the address that it needs to return to. By having a stack, a subroutine can call another subroutine, allowing for nested statements.

3.5 Instructions

The architecture and the generator program were designed to make it easier to create new instructions. With this in mind, the instruction decoding was made automatic.

When referring to the instructions, however, it should be emphasized that the architecture considers the application's reconfiguration and exact memory utilization. This means that the instructions may not be very efficient in terms of addressing space. In common CPU architectures, for example, the addressing space and the instruction composition are very coherent with each other. So, in this case, the configuration of every instruction needs to be mapped to the exact parameters and usage of the current application, which will be discussed in a latter chapter.

3.5.1 Decoding

From the architecture side, new states need to be added to the state machine in order to correctly decode the operands depending on the instruction, as shown in Figure 3.9. In case the whole operation code and arguments fit inside the instruction width, the state remains in the Fetch state. If the instruction does not fit in a single memory cell, then it needs to change to a new state to continue with the decoding. A new state is used for every new type of decoding the program requires, until every operand is completely decoded. The last state of the decoding process can then redirect the state machine to another state, instead of going to the default Fetch, redirecting the flow to a Jump or Branch state, depending on the purpose of the instruction.

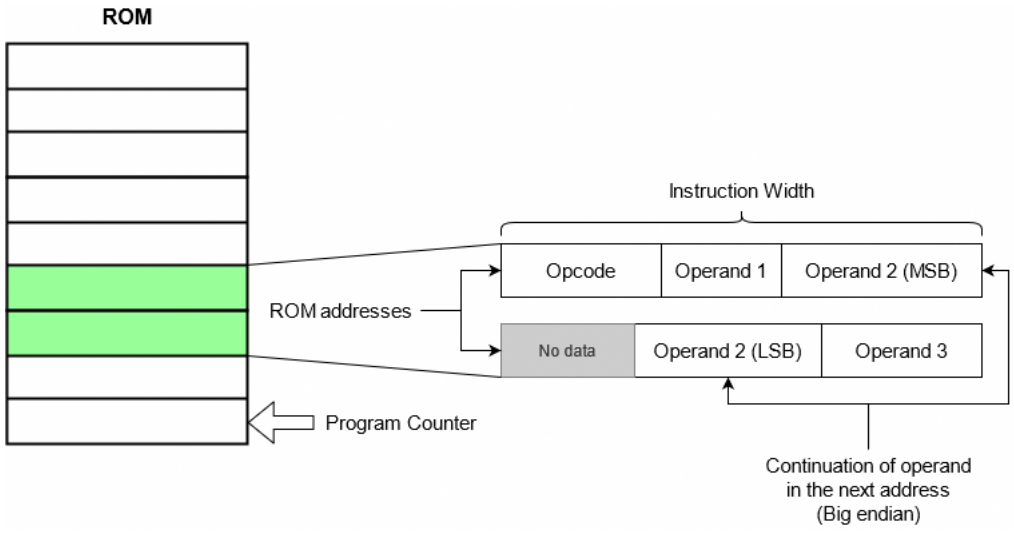

Figure 3.8: ROM instruction storage. Example of an instruction that takes 2 ROM addresses.

This process has a disadvantage however. Even if the two instructions share exactly the same operands, they will need individual decoding states for each, as the final state transition may be different. This may cause a bigger state machine, then a manually created one.

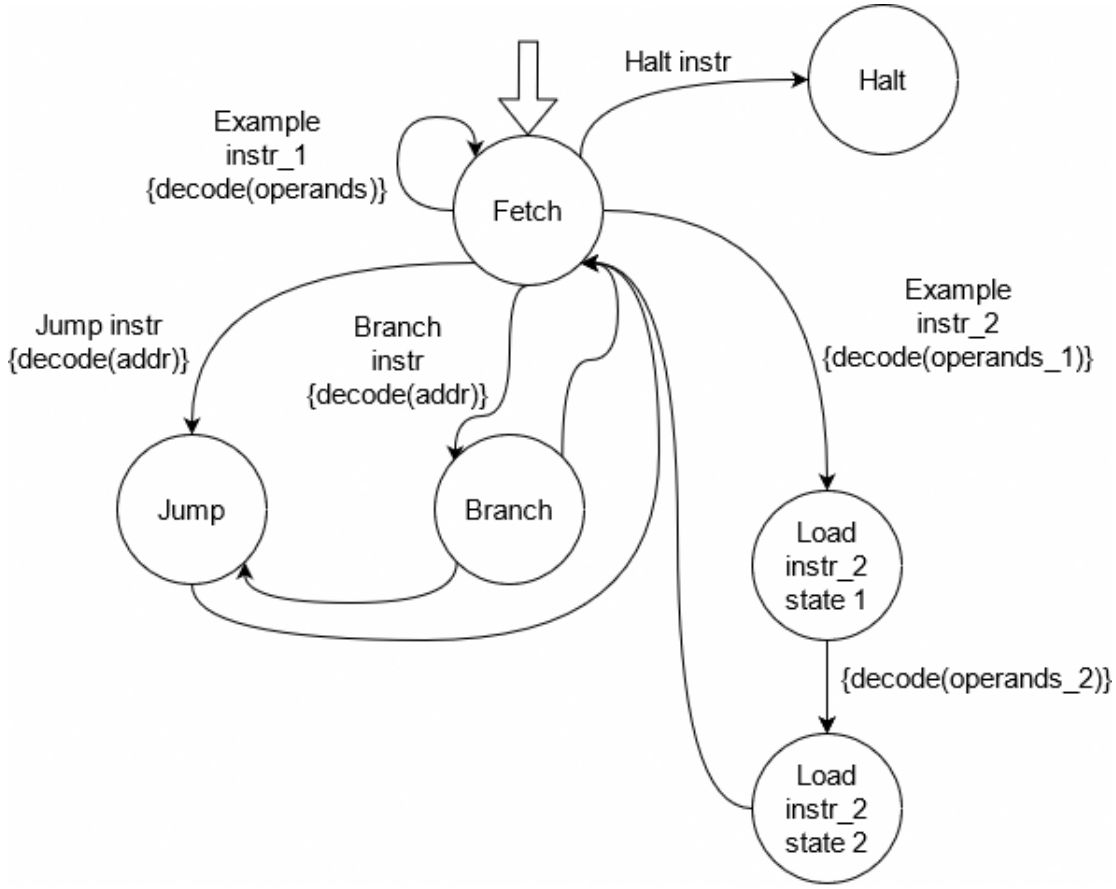

Figure 3.9: Decoding state machine. The example instr_1 only takes a single ROM address, and as such, doesn't need a new state, contrary to instr_2, which occupies 3 ROM addresses.

The decoding itself is simply a matter of filling the right registers with the right information. The operands in the ROM are organized as big endian. When an instruction does not fit in a memory address, and an operand breaks in chunks, its continuation goes to the leftmost side of the next address line, and so on for every other operand.

This is shown in Figure 3.8, where the *Operand 2* is divided into two, and is inserted into the address line on the side of the *Operand 3*.

The number of address lines taken for an instruction depends on the size of its operands, the instruction width set by the user and the total number of instructions used (which translates to the operation code width).

3.5.2 Instructions and Instruction types

As the architecture is somewhat flexible to different types of instructions, these can take a different number and types of arguments.

Most of the instructions consist in assigning values to the correct registers. Some registers of this registers are pipelined, meaning that the effects of changing values will only be noticeable in the next clock cycle.

From Table 3.2 a few more details should be given about some registers.

The Alumode, Opmode and Inmode are inputs specific to the DSP48E1 of the Xilinx series 7. But, has Opmode has a few undefined behaviours for some intructions, a wrapper module (the ALU) was created in order to use this instructions for other custom ones. The register that writes to the output register of the ALU will also select the between the DSP and the output register of the extra instructions depending on the Opmode value. If it is a valid value for the DSP, the write output will write in the DSP register, else, it will write the additional instructions.

The compare instruction uses a feedback loop to evaluate its new value with its previous one. To change the logic operation between the new flag comparison and the previous, the compare settings register was created. It is a two bit register, where bit 1 is used to negate the currently evaluated flag (if set to 1) and bit 0 performs the OR operation with the previous flag (if set to 1, else performs AND). After a branch is performed, be it successful or not, the reset compare register resets the compare output to 1. As such, it is recommended to use the default AND operation at the first compare of a branch.

Most of the instructions have unique operands and operand order assigned to them. A few of the more regular instructions have 2 register addresses, 1 register address and one stack address or a program address (in which to perform a jump). Combining this factor, and the fact that some instructions require different assignments at the end of their decoding, the state machine needs to have specific states for the full decoding of the instructions. This will depend on the instruction width, as well as the width of the operands and the instructions used, as some might share similarities with others.

Table 3.2: Registers used in the control unit to manage the operations and information circulating in the warp.

Register	**Purpose**	**Defaults**	**Output**
Alumode/Opmode/Inmode	Selects operations perfomed in the ALU	-	Yes
Write to outputs	Writes to the output registers of the ALU	0	Yes
Active cores	Currenctly active cores	-	Yes
Sync signal selector	Select the warp external signal (for synchronization purposes)	0	Yes
Immediate	Immediate value to load registers or for operations	-	Yes
Input data selector	Selects the source of the data for the data bus	0 (ALU selected)	Yes
A selector/B selector	Selects the A/B input of the ALU between the registers or data bus	0 (Register bank)	Yes
A address/B address	Register bank addresses	-	Yes
Write to regs	Writes the A address register with the data bus information	0	Yes
RAM address offset	Offset to read/write data compared to the stack pointer private to each core	-	Yes
Push stack	Push information from the data bus to the stack	0	Yes
Pop stack	Pops the information of the stack	0	Yes
Compare instruction	Signals the flag to use for the conditional operation	0 (No compare)	Yes
Compare settings	Settings used for the compare instruction	-	Yes
Reset compare	Resets the conditional register of the cores	0	Yes
Program counter to be loaded	Register used to perform jumps	-	No
Load immediate program counter	Signals the control unit to load the program counter with the program counter to be loaded	0	No
Loop counter	Counts decrementing loops. Can be used in conjunction with the RAM address offset	-	No
Data to subroutine stack	Data used to load the top of the address stack	-	No
Write to subroutine stack	Writes the A address register with the data bus information	0	No
State	Current state of the finite state machine	Fetch	No

Note: The register naming is a simplification and the same names might not be applied in the actual architecture.

Chapter 4

RTL generation program

The RTL generation follows several stages before the final result is stored in files. The files are subsequently stored in directories which make it easier to manually modify or import to the IDE of choice.

The steps were mostly done sequentially by the following order:

1. Read configuration file
2. Program parsing
3. Update program information
4. Assemble machine code file
5. Module generation based on instructions used
6. Module files generation

The first step consists only of reading a configuration file written in JSON (JavaScript Object Notation). The Go standard library already supports parsing this file format, which makes the interchange of data more convenient.

Table 4.1: Adjustable parameters in the configuration file.

Parameter	**Type**
Cores	int
Instruction width	int
Stack depth	int
Default data width	int
Address stack depth	int
Inputs	int
Branches	int
Loop register width	int
Sync bits	int
RAM content file	string

Note: The parameters have default values built in the program, but the used parameters should still be declared in the configuration file.

The other steps will be described in the following subsections.

4.1 Program parsing

The program parser is used to turn a program written similarly to assembly into a data structure, which is the assembler transforms into machine code that is able to be correctly interpreted by the hardware architecture. Furthermore, the parser can also extract some other relevant information about the design.

The parser, however, was not written by hand. A tool called PEG (Parser Expression Grammar) can evaluate user specified rules, written in their own syntax, to generate a program. This program is able to parse strings to the desired data structure and execute instructions. In this case, the Go library used in this program is called Pigeon [40].

```
WORD <- [a-zA-Z_]+ ([0-9] / WORD)* {
    // The second return argument is an error (in this case, none)
  return string(c.text), nil
}
```

Listing 4.1: PEG rule basic example

In this example, a word is defined by starting with at least one letter or underscore and can be followed by any other digit or word. This means that things such as "Example1", or "_example_2" are valid, but "3_example" is not.

This rules can then be used in other sets of rules. In the following case, a label is defined as a word with a colon, any unimportant spacing characters (such as space or tab) and a new line (represented as NL).

```
LABEL <- label:WORD ':' _? NL* {
    // Type conversion to type Label
  l := Label(label.(string))

  // If there are duplicate labels, return error
  if _, exists := declaredLabels[l]; exists {
    return nil, fmt.Errorf("Label %s already exists", label.(string))
  }

  // Add label to the map of existing labels
  declaredLabels[l] = currentSignificantLine

  return l, nil
}
```

Listing 4.2: PEG rule composite example

The usage of the Pigeon library might not produce an optimized parser for the specific application in terms of resource usage or computational speed. However, it allows for a more maintainable approach to create the program. To add a new feature or to modify the syntax of the program, only a few set of rules should need to be changed.

Although there are a lot of variants, the syntax of the language resembles that of an assembly program. The entry point is the top of the file. The labels are instanced as previously mentioned in the example, and do not need to be called by the program instructions (and in the absence of comments, can be used to organize and document the code). Immediate numeric values are signaled using the dollar sign and flags are signaled with a exclamation mark.

```
    // Defining a register starts with a dot
    .register_1 8 output
    .register_2 8
    // A label can be called to perform jumps in the address of the program
Label_example:
    // Instruction that uses two defined registers
    add register_1, register_2
    // The exclamation point marks a flag to use in the compare instruction
    compare $0, !flag
    // A jump to the previously declared label
    jump Label_example
```

Listing 4.3: Program example

The program data is stored in a structure with an ordered array of lines. The lines can be either a label or variable declaration, or an instruction. After the error checking, the ordered array is iterated over to give each label an address and to calculate the total program size.

The program size is variable, as the instruction operands may overflow the current instruction width, forcing a new line to be created as seen in Figure 3.8. This can lead into a cascading effect, as the program size also influences the program addressing width. This may overflow any jump operation, which in turn changes the program size.

As such, this process is repeated until the size of the program remains the same through the last two iterations. By now, the labels also have a fixed address given to them, resolving the issue of any incorrect label address.

The previously referenced error checking occurs in two instances, during and immediately after the parsing. The errors detected are not very complex, but provide some level of guidance to the user.

The list of errors caught during the parsing are as follows:

- Wrong syntax
- Repeated register definition
- Repeated label definition
- Unrecognized flags

After the parsing of the file into the correct data structure, the final error check is performed, raising the following warnings and interrupting the program execution.

- Unknown instruction
- Incorrect operand number

- Incorrect operand type
- Usage of undeclared label/register

The error checking for incorrect instructions is done after the parsing. The Pigeon library in Go interrupts program execution immediately after an anomaly occurs. By checking this type of error after the parsing, it is possible to expose all unknown instructions that may have been used instead of just one.

At this point, it is known what should be used in the architecture design, so all parameters associated with it are updated. This includes the number of used registers and instructions in the program. But before proceeding with the Verilog generation, the program is converted into machine code.

This procedure involves translating the given code for the instruction to the machine code and inserting it to the right offset of the corresponding ROM address. The instruction code might change by rerunning the assembler program, as the structure that holds the information is a hashmap and not an ordered array.

Then, the operands are also inserted into the address as it is described in the Figure 3.8.

4.2 RTL generation

Part of the objective of having a reconfigurable architecture was to cut any unnecessary part of the architecture. This was achieve, to a certain extent, by checking the dependencies of instructions, and removing, not only the instruction and the decoding of the operation itself, but also the modules and registers associated with it.

Having less content written on the files also makes it easier to inspect the architecture, and fine tune it as needed.

With this intent, a Verilog library implemented in Go was created to generated a more organized code structure for the RTL design.

Although Go is not an object oriented programming language, it implements a similar paradigm, the interfaces. This might mean that UML (Unified Modeling Language) class diagrams are not be the most accurate way to describe the interface and data type interactions with each other. However, it is the closest UML type of diagram to what it is intended.

The program was organized in two structures. The module components and the behaviour data structure.

The module's components are the backbone for the generation of the HDL files. This portion consists of signals and other modules instantiation and its connections.

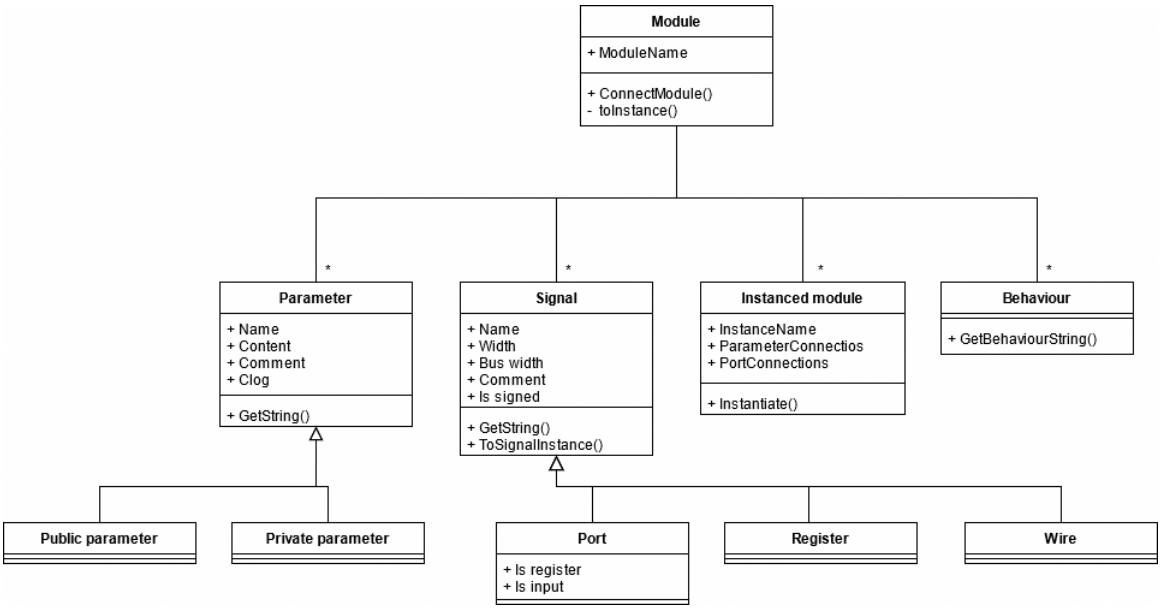

Figure 4.1: Module class diagram

The Go language accepts dynamic type verification and assertion for its variables. For that reason, when a component (like the parameters for instance) have generalizations spawning from them, those sub classes do not need to have a different attribute or method given to them. This can be useful to check what type of variable is being assigned to a connection or instantiation. But, there are several alternatives for this same problem without using variable type assertions. This was the proposed one, as to take advantage of the language capabilities.

The behaviour modeling is done by the second part of the HDL generator. A behaviour can be defined as a assign or a always block (It should be noted that a generate or a initial block of verilog would also be in this category, although the program was not extended to those cases).

An assign can also be inserted into a always block, making it a statement inside of it. This also makes its code declaration different. Moreover, it can also be a blocking or non-blocking assign. For that reason, the assign data structure has a pointer to the always block it is inserted in. In the case it is a null pointer, it is an assign outside the always block. If the always block has an item in the sensitivity list (@ statement), it becomes a non-blocking assign (<=). In the last case, it becomes a clocking assignment (=).

The assign has two structures, one for the left side and the other for the right side. The left side only accepts, signal instances. A signal instance can be the entire signal, a truncation of a certain width of a variable, or a bus selection. The right side accepts this, as well as operations with signals or parameters. The if's and switches are also considered statements, as they can be inserted into blocks (for cycles should also be implemented in this scope, as their structure is very similar). This can be self contained or nested.

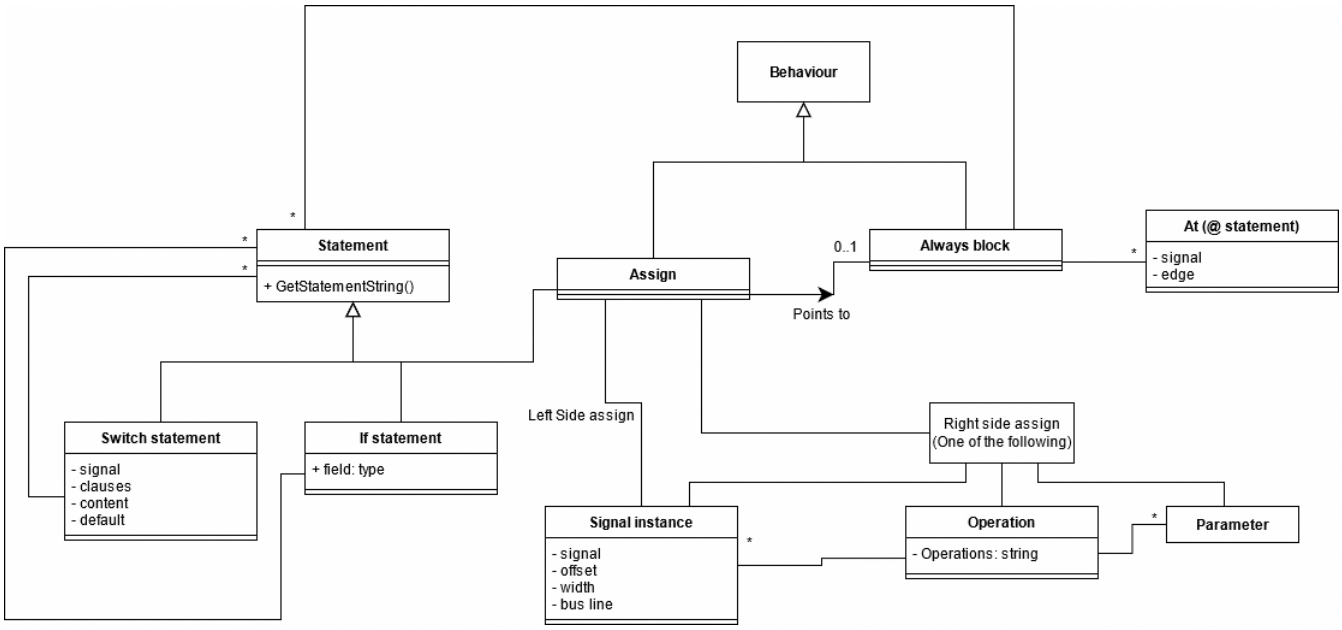

Figure 4.2: Behaviour class diagram

Unfortunately, this behaviour implementation is flawed, as a tree data structure would have been more suited to tackle the issue. Instead, it was implemented as an array. The tree would allow for a recursive implementation that could be scaled up for a larger implementation or project.

A defect in the Signal data type is the lack of buses given to the same signal. As it currently stands, the program only allows a single bus given to a signal, represented by a single attribute in the interface.

4.2.1 Module generation

Although most of the structure is generated using the aforementioned Verilog library, the final file is created using templates [41]. This is an easier alternative to allow some tweaks to the final file and grants a bypass to the limitations imposed by the library implementation.

An intermediary interface calls the methods of the module components and behaviours to generate a string corresponding to the correct data type. This interface makes the connection between the module data structure and the templates.

The modules are created in a single file each, to better organize the content of the architecture.

Some of the modules were created manually (such as the ALU, as it requires the DSP instantiation). This files are simply copied to the created project and placed into a different folder, as to distinguish this type of files, from the automatically generated ones.

4.3 Creating instructions

The creation of new instructions was intended to be done in a external file. In spite of that, it was not possible due to how the program was implemented. The program parser/assembler and the

module creation became two separate entities, and the code base would require a major refactoring to accommodate this feature.

As it was implied with the previous statement, to apply a new instruction, there should be two modifications in the program.

The modification done in the assembler, inside the *instructions.go* file, requires the "mapped instructions" map to be given a new entry. This new entry can have another "operand property" associated to it. This variable type describes the order, the type, and the size of the operands for the instruction.

The second modification is done in a file with the same name, but, this time, inside the modules directory. The "instruction content" variable is also a map that stores the content of the corresponding instruction. The name has to be consistent to the previously given name. Here, there are three attributes given to the instruction.

Firstly, the "operand distribution" structure contains the signals to which the ROM content will be distributed (the order should also be the same to the assembler file). This structure also contains any assigns to signals at the end of the instruction decoding. For example, a write enable to the registers or the ALU after the decoding of the data path selection.

The other two attributes for the new instruction are the "initial statements", which specify the statements that should be performed before any more decoding, and an "additional states" attribute. The last one, redirects the state machine to a given state after the full decoding of the instruction, being specially useful in a jump instruction.

Chapter 5

Results

5.1 Architecture and Program generator

In the previous chapter, the decisions for the architecture design were discussed. This time, its limitations should be displayed.

One major problem of having a reconfigurable architecture is the lack of efficiency in memory addressing and data width. This makes it more difficult to create coherence in data types. In modern CPU architectures that follow the Modified Harvard architecture model (Similar to the Von Neumann Model) it is possible to insert almost any type of data on the data bus. This enables operations with program counters, instructions or almost any other register. In a multi-core system, it might be more difficult to create a consensus between the information, specially if the flow of the information is from the cores to the control unit. But, this type of feature could allow for adaptable loops or the usage of a generic control stack in the control unit. Furthermore, due to the same reason, a single control flow stack in the control unit could be very inefficient. If the width of the types of data to be stored in the stack are too discrepant, it would cause a lot of unused space in the memory.

In this regard, a component that this type of architecture would benefit greatly would be a Memory Management Unit (MMU). Although such a unit is out of the scope of this project, due to its dimensions and time needed to construct it, this component would enable the coherent access to the memory in the system. This could also include the interchange of information between cores and memory hierarchies, without compromising the data integrity.

The generator program also has a few limitations, besides the ones exposed in the previous chapter. The program was not able to fully optimize the wire and register width of the data. Currently, the program generator only uses the maximum data width as a reference, and uses it for the registers, wires and RAM. The program could truncate or expand buses and wires in order to reduce the space used per register or RAM address to use the strictly necessary memory space.

5.2 CUDA Comparison

The CUDA framework allows the compilation of C/C++ code to run on a GPU and take advantage of it's architecture. So, as it is aligned with the interest of this thesis, a comparison should be drawn between the two of them. This will not be a complete series on CUDA programming, as that would be beyond the scope of the section. That being said, the examples presented will only cover a small portion of the topic.

5.2.1 Kernel programming

In CUDA, the intended parallelizable part of the program can easily (on most cases, that is) be programmed calling a simple function called a kernel.

The GPU MMU also allows the cores to share memory between each other in a coherent manner. This makes it easier to reference variables that are shared across all cores, accessing it using the coordinates assigned to the cores. The cores can have 3 axis per coordinate: x, y and z; and 2 types of coordinates: one associated to the grid position and the other to the block. When the kernel is called, Listing 5.2, the dimensions for the grid and the blocks are defined. This variables can have 3 axis each, so, for each core (or more accurately, thread) there are 6 coordinates. Having these amount of freedom permits an adaptability for a wide range of applications in processing data with 3, or less, dimensions.

Taking a look at a basic example and a simple parallelization problem, a N by N matrix multiplication[42, 43].

```
__global__ void matrixMult(int *A, int *B, int *C, int N){
    int row = blockIdx.y * blockDim.y + threadIdx.y;
    int col = blockIdx.x * blockDim.x + threadIdx.x;

    int accumulator = 0;
    if (row < N && col < N) { // In case the block dimension is bigger than the
    matrix
        for (int i = 0; i < N; i++) {
            accumulator += A[row * N + i] * B[i*N + col];
        }
    }
    C[row * N + col] = accumulator;
}
```

Listing 5.1: CUDA Kernel for the matrix multiplication

```
    // Variables declaration of type dim3
    dim3 threads_per_block(N, N);
    dim3 blocks_per_grid(1, 1);
    matrixMult<<<blocks_per_grid, threads_per_block>>>(A, B, C, N);
```

Listing 5.2: CUDA Kernel function call

CUDA makes it available to every single core its own thread and block ID. With this, and a shared memory, it is possible to access the location of a memory address given just an array pointer.

This is contrary to the project's architecture, which does not support pointers, as the address lines and registers are not connected to the data bus.

The fact that there is no shared memory also makes it impossible to have multiple cores access different addresses in the same memory space.

When calling the kernel, the dimension of the block can be specified, making it very easy to scale the project. Not only that, this allows for dynamic scaling, as the dimensions specified do not need to be constant values at compile time. This is due to the usage of a CPU in combination with a flexible GPU architecture. Having different types of memory inside the GPU architecture allows it to dynamically allocate the needed space for the application, and reserve the right amount of cores for the task. It also let's multiple applications run on the same device. This point, however, is not very relevant for the current project.

In contrast to CUDA, the project's tools output would require a lot more manual tinkering.

With the current architecture, it is possible to stream and process the data at the same time. Nevertheless, as to draw a comparison between the CUDA model, it is assumed that the information is already loaded into memory before processing it.

```
1  Setup_load_loop:
2      begin_loop          $[N]
3  Load_Memory:
4      load_in             temp_reg, $4
5      push                temp_reg
6      load_in             temp_reg, $5
7      push                temp_reg
8      loop                Load_Memory
9  Load_MACC_instruction:
10   load_op       $0, $37, $0
11 Setup_mult_loop:
12   begin_loop      $[N]
13 Multiply:
14   pop                 temp_reg
15   exec_op_loop      temp_reg
16   pop                 temp_reg
17   loop        Multiply
18 Output_result:
19   save_to_reg         output_reg
```

Listing 5.3: Base program for the matrix multiplication

5.2.2 Memory management

CUDA provides intelligent allocation, based on the GPU model and resources, both in compilation-time and in run-time.

```
    int *A, *B, *C;
    // Allocate Memory in CPU and GPU - The memory will be transferred
    automatically
    cudaMallocManaged(&A, N*sizeof(int));
    cudaMallocManaged(&B, N*sizeof(int));
    cudaMallocManaged(&C, N*sizeof(int));

    // Kernel execution
    matrixMult<<<blocks_per_grid, threads_per_block>>>(A, B, C, N);

    // Synchronize data on CPU and GPU after kernel execution
    cudaDeviceSynchronize();
```

Listing 5.4: CUDA automatic memory management

This example uses managed allocation, but CUDA also provides manual allocation of memory and can even target specific memory levels. This means a fully optimized program for a particular architecture of GPU by detailing the resource locations for a performance boost.

The main limiting factor when using the GPU to accelerate computations, is the memory transfer bandwidth and latency. This problem is evident when the data needs to be passed down from a CPU to the GPU.

The FPGA has an advantage in this regard, as it can take the raw data directly and pre/post-process it within the FPGA itself. Not only that, it might not even need a software layer, which usually detriments the performance of the task.

5.3 Example use case

To illustrate the program and architecture, it was showcased an example application. The objective is to implement a Fully Connected Neural Network (FCNN) using the design process of this project.

5.3.1 The FCNN

The FCNN is a trained model from the MNIST dataset, a classic example of neural networks. In order to train the model, the Tensorflow framework from the Python programming language was used.

The MNIST dataset is a set of monochromatic handwritten digits from 0 to 9 and the objective of the neural network is to correctly identify the corresponding digit. Given the simplicity of the problem, there should not be any need for a hidden layer, so the total amount of neurons are a number arbitrated for the input layer and 10 neurons for the output. In this example, 20 input neurons were used to demonstrate the application.

The FPGA chosen for the application was a from the Zynq-7000 series. As this SoC tend to have smaller FPGAs, their resources may not be sufficient to store all the weights of the network,

the images were shrunk to 20x20 pixels, giving a total of 400 pixels, instead of the 784 pixels (28x28).

The activation function chosen for the output of the first layer was a ReLU (Rectifier Linear Unit). It a simpler activation function that can be implemented using the current operations in the architecture and demonstrates the branching effect.

The implementation is done by integrating the SM with other blocks that select the signal data. This reduces the size of the multiplexers and the logic inside the SM. The pixel select and layer 2 input select blocks pick the respective input by counting the number of synchronization signal pulses. This selection could be performed using multiplexers inside the SM. However, this would cause large multiplexer address spaces. Moreover, this way, it is easier to implement an outside buffer to control the intake of the pixels.

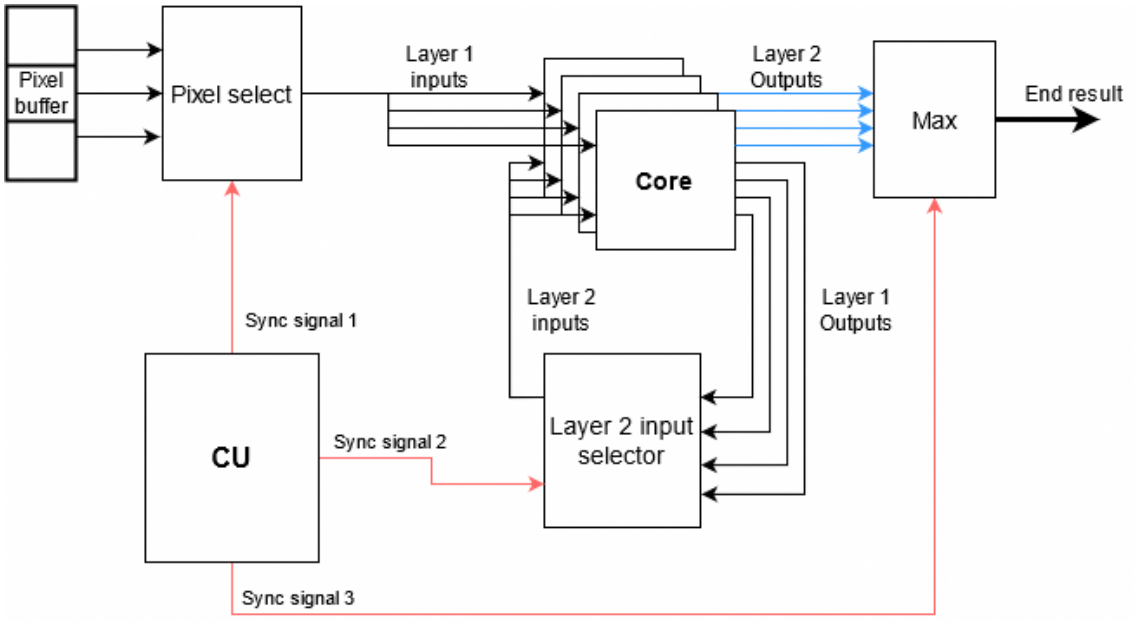

Figure 5.1: FCNN Hardware implementation

The total of 30 neurons are implemented using a SM of 20 cores. The second layer depends on the output of the first. So, after the 20 cores process the first layer, the 10 last cores are turned off to execute the second layer.

5.3.1.1 Neural network modifications

Even though Tensorflow already has the MNIST dataset available, the implementation needed differs a bit from the classic. Tensorflow uses floating point weights and biases for its networks. But, these are very computational intensive and the current architecture does not support this type of operations. As such, the trained network needs to be adjusted accordingly.

The first problem comes from the resolution of the trained network. The final weights and biases normally range between -2 and 2, with decimal points. That means that is not possible to

directly translate those to the FPGA. So, the minimum and maximum of the overall network were obtained and with this information, the weights were distributed linearly between the two values.

Using this method, the choice for the quantization bits was 8 for the weights. It should provide enough resolution to solve the MNIST dataset, whilst not occupying much space per weight in the memory.

The second problem comes from the possible overflowing of the registers when performing the MACC operations needed for the neurons. In order to guarantee no overflow occurs when performing a sum, one extra bit should be provided. For example, summing 8 bits with 8 bits would provide 9 bits. So, in a multiplication the number of bits of the end result should double the input's. When performing a lot of consecutive MACC operations, this would scale to an exorbitant register size needed to loose no information (there is no such overflow problem with float operations, as the resolution and scale change according to the magnitude of the numbers, although some information may be lost in the process).

Firstly, the input was quantized from 8 bits to 2 bits. This resolution change has to be somewhat drastic because the first layer of the neural network will have to make a MACC for every single pixel in the image, which can lead to a quick overflow. Fortunately, as the weights are typically symmetrical to zero, there is less probability of having an overflow. Besides, as the pictures fed into the network are monochromatic and have low resolution, having such a low resolution should not constitute any issue.

Secondly, the 16 bit result from the layer 1 is then piped through a special instruction that requatized the output to 8 bits. However, for simplicity and performance, the quantization is not done using the same method above described. This is done through a simple shift right, which removes the least significant bits from the results. This is not the most robust method, compared to the linear distribution between the minimum and maximum value, but the network still predicts the digit accurately.

5.3.1.2 RTL modifications

The architecture itself needs to be slightly tinkered with to be able to take the most out of it.

As mentioned previously, the requantization after the first layer was given a special instruction in order to accelerate the performance. This was achieved using an invalid DSP operation code, making a shift right when the given code is assigned to the registers.

The weights and biases are loaded into the RAM from a file, and follow the structure shown in Figure 5.2. The memory is organized to enable its access of the weights in a sequential order using the instruction with the loop iteration as an offset.

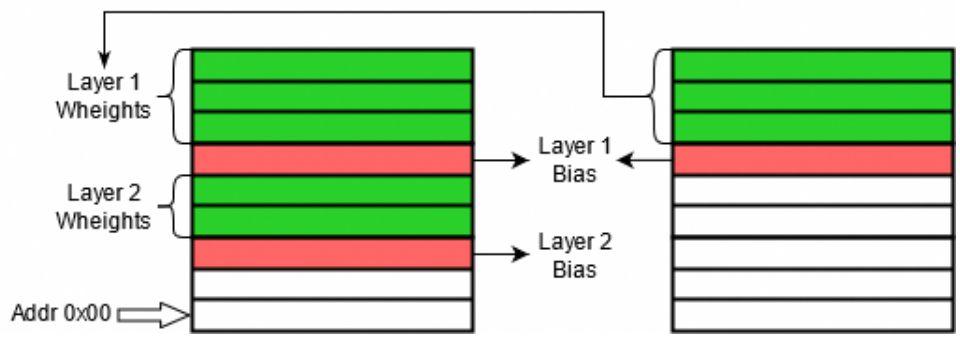

Figure 5.2: RAM content. The right stack is the content of a core that only implements the first layer, while the left one executes in all layers.

It is noteworthy that the instruction width was set to 12 bits, as most of the instructions did not take more than this space.

5.3.2 Program implementation

The program consists of 3 parts. Two of them are the layer matrix multiplication and its setup, while middle one is the requantization and the ReLU.

The program begins with a variable declaration, it is important that the output order is correctly implemented, as it translates to the peripheral output of the SM. A temporary register is used to store a intermediate results. This register is not actually needed, as the other registers could be used to store temporary values as well. However, this results in fewer transitions at the outputs, avoiding unnecessary register switching.

```
1 Vars_init:
2   .res_layer_1 16 output
3   .res_layer_2 16 output
4   .temp_reg 16
```

Listing 5.5: Variable declaration

The entry point of the applications begins by enabling all cores and resetting the DSP register to 0. Then, the multiply and accumulate operation is loaded and the loop begins. The loop is set to the 400 pixels of the image and to be able to multiply each pixel by its respective weight, the operation inside the loop uses the loop value as an offset. Every time a MACC is complete, the synchronization signal 0 is set in order to move to the next pixel.

After the loop is complete, the bias is added to the end result.

```
1 Reset_ALU_reg_1:
2   activate_all
3   load_op       $0, $0, $0
4   exec_op       temp_reg, temp_reg
5 Start_layer_1:
6   load_op       $0, $37, $0
7   begin_loop      $400
8 Layer_1_mult:
```

```
  load_in       temp_reg, $4
  exec_op_loop_offset temp_reg, $0
  sync_signal     $0
  loop        Layer_1_mult
Add_bias_1:
  save_to_reg     temp_reg
  load_op       $0, $51, $0
  exec_op_stack     temp_reg, $401
```

Listing 5.6: Program entry point and layer 1 calculation

Having the complete result of the first layer, the requantization and activation function process are merged into an if/else statement, as only the positive numbers need to be requantized. The load operation has the operation code assigned for the shift right of the DSP register, setting up this shifted output for the next instructions.

The branching itself is done by checking if the negative flag is not true ($2 translates to "`not negative`"). If this branch condition is true, the output is loaded into the layer 1 output register. If not, 0 is loaded into the register instead.

```
Load_requantize_operation:
  load_op       $0, $127, $0
ReLU:
  new_branch
  compare       $2, !negative
  branch        Load_output
  load_imm      res_layer_1, $0
  update_branch   End_ReLU
Load_output:
  save_to_reg     res_layer_1
End_ReLU:
  end_branch
```

Listing 5.7: Requantization and ReLU

The process for the second layer is much similar, being the only difference the previous activation of the first 10 cores, which correspond to the second layer neurons. Lastly, the jump reverts the program to the beginning. This is equivalent to an infinite while loop.

```
Reset_ALU_reg_2:
  load_op       $0, $0, $0
  exec_op       temp_reg, temp_reg
Start_layer_2:
  activate_cores    $1023
  load_op       $0, $37, $0
  begin_loop      $20
Layer_2_mult:
  load_in       temp_reg, $5
  exec_op_loop_offset temp_reg, $401
  sync_signal     $1
  loop        Layer_2_mult
```

```
13 Add_bias_2:
14   save_to_reg     temp_reg
15   load_op        $0, $51, $0
16   exec_op_stack   temp_reg, $422
17   save_to_reg     res_layer_2
18   sync_signal     $2
19   jump_addr      Reset_ALU_reg_1
```

Listing 5.8: Layer 2 calculation and jump to the beginning of the program

5.3.3 Resource usage

The resource usage of the FPGA were the following:

Table 5.1: Implementation resource usage of the FCNN in the FPGA of the Zynq-7000 (Device name Z-7010)

	Used	**Usage (Percentage)**
LUT	4746	27%
LUTRAM	2560	42.7%
FF	1062	3%
DSP	20	25%
IO	11	20.4%
BUFG	5	15.6%

The most important aspect to point out is the automatic usage of the LUTRAM of the Vivado tools. This increases the possible speed of the clock and as the FPGA has enough resources, this will be used for the reference, as the example is putting more emphasis on performance.

In case the ROM and RAM were forced to the BRAM components already present in the FPGA (using the keywords *rom_style/ram_style="TRUE"*) the clock period would need to be decrease by 2ns (the next section will present the actual clock speed). Nonetheless, the LUTRAM would not be used in the implementation, sparing some of the more performant memory for another application.

5.3.4 Performance

The estimated clock speed in the Vivado synthesis and implementation tools was above 140MHz (7.2ns clock period). Knowing that an execution of the network takes 2594 clock cycles to complete, the network can output results at a frequency of around 53KHz (18.7μs between outputs).

To compare the performance of the implemented architecture for the problem, the same algorithm was replicated in C++ and compiled with the g++ program. This language was chosen, as it normally provides fairly fast programs and has a math focused library (Eigen) with optimizations implemented to further improve the compiled binary [44].

The program was then tested in two devices. One was a normal personal computer processor (Intel i7-8565u, 18GHz, 4 cores) and an ARM processors of the Zynq-7000 series (in a Red Pitaya board, with the device name Z-7010). This two devices provide a baseline for the performance of the FPGA implementation, both in the context of normal computing and in the embedded systems area. In this last case, it is also relevant to understand the speedup gained from the new hardware design, as both the ARM and FPGA belong in the same IC.

The C++ program was tested with two methods, using the time UNIX utility and the gprof (GNU profiler). The program was neural network was iterated a numerous amount of times in order to have a better average time of the network execution.

Unfortunately, a GCC error made it impossible to execute the program in the Zynq processor without removing the output prints. This system call can add a lot of overhead to the execution time. As such, it will be assumed that the program took two thirds of its time performing the actual program, and not the printing functions. This number was estimated from comparing two programs, one with and the other without prints, on the Intel processor.

On the processor Intel, the time taken for the program execution oscillated from 90μs to 100μs per image.

On the ARM processor's side, the time utility estimated a time of 2.9ms (4.3ms with prints) per image, while the profiling program estimated a 1.1ms (1.7ms with prints) execution time. To have a better comparison with the FGPA, the last time will be used to draw a comparison, as it is the best result.

So, having the results, the program running in the ARM processor is almost 60 times slower than the FPGA and the Intel processor around 5 times. This is a very promising result, even when using a not completely optimized hardware architecture.

5.3.5 Optimizations

After the base example executed as intended and a foundation for other result was set, it is clear that the design could be further optimized.

With only three small modifications the performance was accelerated by 30%, two of which involve creating new and custom instructions.

The first, although not as focused on performance as the other, is to create an instruction to modify the Opmode register individually. The instruction $load\_instr$ takes three arguments, but the Inmode and the Alumode are not actually used in this program, and are always set to 0. So, the decoding of the three operands was not possible in 1 clock cycle for the current instruction width. By substituting this instruction with $load\_opmode$ instead, the ROM usage will decrease, and so will the logic used for the decoding and the clock cycles needed.

The other two optimizations were targeted to reduce the amount of clock cycles taken inside the loops. In the unaltered program, each loop takes 6 clock cycles.

Firstly, the instruction $exec\_op\_loop\_offset$ needs two arguments, taking a total of 16 bits for the whole instruction:

1. Operation code : 5 bits (20 unique instructions)
2. Register addressing : 2 bits (3 registers)
3. RAM offset : 7 bits (512 RAM depth)

As the decoding only evaluates a ROM address line at a time, having 16 bits of instruction width will allow the decoding of this instruction in one clock cycle.

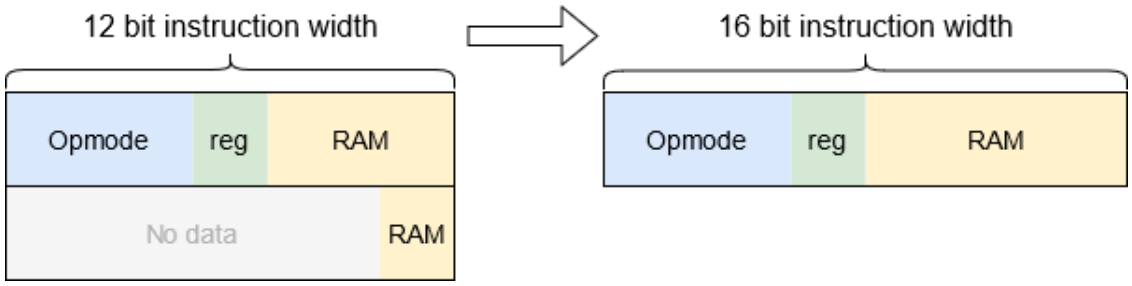

Figure 5.3: Fixed ROM address overflow.

The next step was merging the *load_in* and the *sync_signal* into a single operation, as the synchronization is simply to set a bit to one in order for the exterior modules to coordinate with the SM. So these two operation were combined into *load_input_and_sync*, which takes all the arguments from the previous instructions.

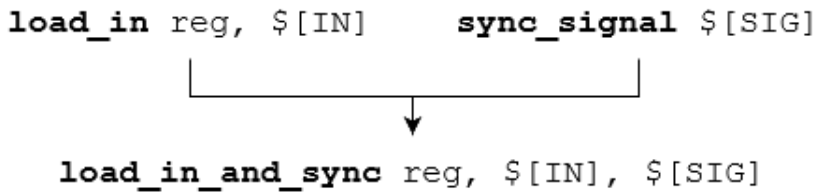

Figure 5.4: New instruction created by merging two existent instructions.

```
  begin_loop      $400
Layer_1_mult:
  load_in_and_sync  temp_reg, $4, $0
  exec_op_loop_offset temp_reg, $0
  loop        Layer_1_mult
```

Listing 5.9: Layer 1 weight multiplication with the instruction

With these final two modifications, the clock count per loop came down to 4, instead of 6. As the program takes most of it's time inside the loops, the clock cycles reduced to perform the image analysis were roughly a third, going to 1740 clock cycles.

The overall resource usage and clock constrain remained approximately the same, making the output frequency of the module to 77KHz, or 13.0μs per execution.

Now, the ARM and Intel processors were slower almost 85 and 7 times, respectively, compared to the optimized design.

5.3.6 Scalability

This specific problem can scale up in several different ways.

One, for instance, is the image size increment. With the current implementation, this would essentially mean an increase in the number of loops and RAM depth.

As most of the execution of the program is performed inside loops, the clock cycles needed for the whole program are approximately 4 times the number of inputs of each layer, which correspond to the total number of loops in every layer.

So, by increasing the number of pixels in the image, the first layer's input will increase on a 1 to 1 proportion, and so will the number of weights.

The FCNN could be expanded in two other ways. Increasing the number of layers or the number of neurons.

When incrementing the number of neurons, only the number of cores could end up increasing. That would mean more overall resource usage, not only in the number of cores, but also the weights the next layer needs. The speed processing speed, however, would only increase due to the next layer having more inputs. This would mean that the increased layer itself would perform in the same amount of clock cycles as the previous design.

The second scaling alternative could be tackled in different ways. The current implementation is one of them. The cores used translate directly the maximum number layer size, and the cores are activated as needed per layer. As every core is symmetrical, this can also mean that some RAM space is unutilized.

Another alternative to implementing the FCNN could be to split the layers into different SMs. This way, there would be no need to put cores in a SM on idle mode. With this pipelined method, the latency between the input and the processed result could remain approximately the same compared to the previous implementation, but the throughput could be increased.

There is, however, a caveat. Not only would the SMs need to have some synchronization method between them, but also, some entire SMs could end up needing to remain idle for an amount of time depending on the amount of inputs/outputs. In case a layer has more inputs than outputs, the SM would need to remain inactive while it awaits new data from the previous layer.

The last modification that the FCNN could suffer can be in the weights and image width. Most of this process would be manual and require previous tests to understand the adequate data width resolution to have no overflows.

Lastly, the requantization instruction performed in intermediary layers might also require to be adjusted to the new data and weights width.

Chapter 6

Conclusions and future work

6.1 Objective's fulfilment

The overall project can be considered to be fairly close to its initial objectives. The architecture should provide enough power to be able to compete with other embedded systems like microprocessors, while still maintaining some ease of usage and flexibility.

Implementing an application specific architecture purely in RTL is usually the best option performance and resource wise. The tools developed sacrifice part of these benefits, but allow for a more expansible and organized code base.

Unfortunately, the program generator lacks in optimization and reconfiguration features. Things like the register usage or memory sizes should be automatically optimized to the bare minimum for the application. Or, at the very least, configurable through the generator program. At the current point, these types of optimizations need to be done manually in the HDL files. Luckily, the files follow a consistent order, so, finding and changing the needed components is easier.

Other convenient missing features include: more command line instructions for the program, to make the user interaction simpler; expandable instruction list (without having the need to recompile the program) through a configuration file; a simpler and more automatic way to integrate modules provided by the user. The generator program should have been able to provide more reconfiguration power. Not only that, it should also be much more optimized.

6.2 Future work

As it was stated, the project could still profit from further refinement. As it stands, all parts of the project have room for improvement, even if ever so slightly.

Firstly, one fundamental modification that could be done to the architecture is to implement a Modified Harvard model, allowing access to different types of data and registers all across the streaming multiprocessor. This would make it more malleable to more circumstances and contexts. Besides, the data and control bus widths could be optimized to reduce the resources used and the clock propagation time, while still maintaining the best possible data resolution.

In addition to this, having a shared memory across all cores can also shift the programming paradigm with a tool like this. Using pointers and pointer arithmetic for each thread makes the memory exchange simpler.

Furthermore, it would be interesting to enhance the current architecture to transform it into a super-scalar processor. Capable of performing more than one instruction at a time without raising conflicts in the instruction execution.

Another interesting aspect to explore would be a compiler capable to adapt to a reconfigurable hardware design. With a higher level syntax, emulating a language such as C for instance, the development speed and the versatility would increase.

Appendix A

NVIDIA Terms

Table A.1: Conversion from terms used in [12] to official NVIDIA/CUDA and AMD jargon

Type	More descriptive name used in the book	Official CUDA/NVIDIA term	Book definition and AMD and OpenCL terms	Official CUDA/NVIDIA definition
Program abstractions	Vectorizable loop	Grid	A vectorizable loop, executed on the GPU, made up of one or more "Thread Blocks" (or bodies of vectorized loop) that can execute in parallel. OpenCL name is "index range." AMD name is "NDRange".	A grid is an array of thread blocks that can execute concurrently, sequentially, or a mixture.
	Body of Vectorized loop	Thread Block	A vectorized loop executed on a multithreaded SIMD Processor, made up of one or more threads of SIMD instructions. These SIMD Threads can communicate via Local Memory. AMD and OpenCL name is "work group".	A thread block is an array of CUDA Threads that execute concurrently together and can cooperate and communicate via Shared Memory and barrier synchronization. A Thread Block has a Thread Block ID within its Grid.
	Sequence of SIMD Lane operations	CUDA Thread	A vertical cut of a thread of SIMD instructions corresponding to one element executed by one SIMD Lane. Result is stored depending on mask. AMD and OpenCL call a CUDA Thread a "work item."	A CUDA Thread is a lightweight thread that executes a sequential program and can cooperate with other CUDA Threads executing in the same Thread Block. A CUDA Thread has a thread ID within its Thread Block.

Table A.2: Conversion from terms used in [12] to official NVIDIA/CUDA and AMD jargon (cont. 1)

Type	More descriptive name used in the book	Official CUDA/NVIDIA term	Book definition and AMD and OpenCL terms	Official CUDA/NVIDIA definition
Machine object	A Thread of SIMD instructions	Warp	A traditional thread, but it contains just SIMD instructions that are executed on a multithreaded SIMD Processor. Results are stored depending on a per-element mask. AMD name is "wavefront."	A warp is a set of parallel CUDA Threads (e.g., 32) that execute the same instruction together in a multithreaded SIMT/SIMD Processor.
	SIMD instruction	PTX instruction	A single SIMD instruction executed across the SIMD Lanes. AMD name is "AMDIL" or "FSAIL" instruction.	A PTX instruction specifies an instruction executed by a CUDA Thread.
Processing hardware	Multithreaded SIMD processor	Streaming multiprocessor	Multithreaded SIMD Processor that executes thread of SIMD instructions, independent of other SIMD Processors. Both AMD and OpenCL call it a "compute unit." However, the CUDA Programmer writes program for one lane rather than for a "vector" of multiple SIMD Lanes.	A streaming multiprocessor (SM) is a multithreaded SIMT/ SIMD Processor that executes warps of CUDA Threads. A SIMT program specifies the execution of one CUDA Thread, rather than a vector of multiple SIMD Lanes.
	Thread block scheduler	Giga thread engine	Assigns multiple bodies of vectorized loop to multithreaded SIMD Processors. AMD name is "Ultra-Threaded Dispatch Engine".	Distributes and schedules thread blocks of a grid to streaming multiprocessors as resources become available.
	SIMD Thread scheduler	Warp scheduler	Hardware unit that schedules and issues threads of SIMD instructions when they are ready to execute; includes a scoreboard to track SIMD Thread execution. AMD name is "Work Group Scheduler".	A warp scheduler in a streaming multiprocessor schedules warps for execution when their next instruction is ready to execute.
	SIMD Lane	Thread processor	Hardware SIMD Lane that executes the operations in a thread of SIMD instructions on a single element. Results are stored depending on mask. OpenCL calls it a "processing element." AMD name is also "SIMD Lane".	A thread processor is a datapath and register file portion of a streaming multiprocessor that executes operations for one or more lanes of a warp.

Table A.3: Conversion from terms used in [12] to official NVIDIA/CUDA and AMD jargon (cont. 2)

Type	More descriptive name used in the book	Official CUDA/NVIDIA term	Book definition and AMD and OpenCL terms	Official CUDA/NVIDIA definition
Memory hardware	GPU Memory	Global Memory	DRAM memory accessible by all multithreaded SIMD Processors in a GPU. OpenCL calls it "Global Memory."	Global memory is accessible by all CUDA Threads in any thread block in any grid; implemented as a region of DRAM, and may be cached.
	Private Memory	Local Memory	Portion of DRAM memory private to each SIMD Lane. Both AMD and OpenCL call it "Private Memory."	Private "thread-local" memory for a CUDA Thread; implemented as a cached region of DRAM.
	Local Memory	Shared Memory	Fast local SRAM for one multithreaded SIMD Processor, unavailable to other SIMD Processors. OpenCL calls it "Local Memory." AMD calls it "Group Memory".	Fast SRAM memory shared by the CUDA Threads composing a thread block, and private to that thread block. Used for communication among CUDA Threads in a thread block at barrier synchronization points.
	SIMD Lane registers	Registers	Registers in a single SIMD Lane allocated across body of vectorized loop. AMD also calls them "Registers".	Private registers for a CUDA Thread; implemented as multithreaded register file for certain lanes of several warps for each thread processor.

References

[1] What is an FPGA? URL: https://www.xilinx.com/html_docs/xilinx2017_4/sdaccel_doc/gac1504034293050.html.

[2] Matan Rusanovsky, Re'Em Harel, Lee-Or Alon, Idan Mosseri, Harel Levin, and Gal Oren. *BACKUS: Comprehensive High-Performance Research Software Engineering Approach for Simulations in Supercomputing Systems*. October 2019.

[3] Mark D Hill and Michael R Marty. Amdahl's Law in the Multicore Era. page 6.

[4] David Patterson, Thomas Anderson, Neal Cardwell, Richard Fromm, Kimberly Keeton, Christoforos Kozyrakis, Randi Thomas, and Katherine Yelick. A case for intelligent RAM. *Micro, IEEE*, 17:34–44, April 1997. doi:10.1109/40.592312.

[5] Roofline model, June 2020. Page Version ID: 963987092. URL: https://en.wikipedia.org/w/index.php?title=Roofline_model&oldid=963987092.

[6] David A Patterson and John L. Hennessy. *Computer Organization and Design: The Hardware/Software Interface*. 5 edition.

[7] Chris McClanahan. History and Evolution of GPU Architecture. page 7, 2010.

[8] Inside Volta: The World's Most Advanced Data Center GPU, May 2017. URL: https://developer.nvidia.com/blog/inside-volta/.

[9] Laiq Hasan, Marijn Kentie, and Zaid Al-Ars. DOPA: GPU-based protein alignment using database and memory access optimizations. *BMC research notes*, 4:261, July 2011. doi:10.1186/1756-0500-4-261.

[10] Vasanth Venkatachalam and Michael Franz. Power reduction techniques for microprocessor systems. *ACM Computing Surveys*, 37(3):195–237, September 2005. URL: https://doi.org/10.1145/1108956.1108957, doi:10.1145/1108956.1108957.

[11] 7 Series DSP48E1 Slice User Guide (UG479). page 58, 2018.

[12] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*. The Morgan Kaufmann Series in Computer Architecture and Design. Morgan Kaufmann, 5 edition, 2011.

[13] J. Dean, D. Patterson, and C. Young. A New Golden Age in Computer Architecture: Empowering the Machine-Learning Revolution. *IEEE Micro*, 38(2):21–29, March 2018. Conference Name: IEEE Micro. doi:10.1109/MM.2018.112130030.

[14] A. Sodani, R. Gramunt, J. Corbal, H. Kim, K. Vinod, S. Chinthamani, S. Hutsell, R. Agarwal, and Y. Liu. Knights Landing: Second-Generation Intel Xeon Phi Product. *IEEE Micro*, 36(2):34–46, March 2016. Conference Name: IEEE Micro. doi:10.1109/MM.2016.25.

[15] Liang Wang, Gill Fountain, Bongsub Lee, Guilian Gao, Cyprian Uzoh, Scott McGrath, Paul Enquist, Sitaram Arkalgud, and Laura Mirkarimi. Direct Bond Interconnect (DBI®) for fine-pitch bonding in 3D and 2.5D integrated circuits. In *2017 Pan Pacific Microelectronics Symposium (Pan Pacific)*, pages 1–6, February 2017.

[16] J. W. Choi, D. Bedard, R. Fowler, and R. Vuduc. A Roofline Model of Energy. In *2013 IEEE 27th International Symposium on Parallel and Distributed Processing*, pages 661–672, May 2013. ISSN: 1530-2075. doi:10.1109/IPDPS.2013.77.

[17] G. Ofenbeck, R. Steinmann, V. Caparros, D. G. Spampinato, and M. Püschel. Applying the roofline model. In *2014 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 76–85, March 2014. doi:10.1109/ISPASS.2014.6844463.

[18] Modified Harvard architecture, January 2021. Page Version ID: 1002160162. URL: https://en.wikipedia.org/w/index.php?title=Modified_Harvard_architecture&oldid=1002160162.

[19] Harvard architecture, March 2020. Page Version ID: 943976392. URL: https://en.wikipedia.org/w/index.php?title=Harvard_architecture&oldid=943976392.

[20] Instruction set architecture, January 2021. Page Version ID: 1003219780. URL: https://en.wikipedia.org/w/index.php?title=Instruction_set_architecture&oldid=1003219780.

[21] Daniel Page. *Practical Introduction to Computer Architecture*. Texts in Computer Science. Springer London, London, 2009. URL: http://link.springer.com/10.1007/978-1-84882-256-6, doi:10.1007/978-1-84882-256-6.

[22] Thomas ScottCrow. Evolution of the Graphical Processing Unit, December 2004. URL: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.142.368&rep=rep1&type=pdf.

[23] AMD Corp. AMD ISA. URL: http://developer.amd.com/wordpress/media/2012/10/R700-Family_Instruction_Set_Architecture.pdf.

[24] AMD Corp. AMD RDNA ISA. URL: https://developer.amd.com/wp-content/resources/RDNA_Shader_ISA.pdf.

[25] NVIDIA Corp. CUDA C Programming Guide. URL: https://developer.download.nvidia.com/compute/DevZone/docs/html/C/doc/CUDA_C_Programming_Guide.pdf.

[26] RISC-V Vector Extension Intrinsic Support - SiFive. URL: https://www.sifive.com/blog/risc-v-vector-extension-intrinsic-support.

[27] V for vector: software exploration of the vector extension of RISC-V, May 2020. URL: https://www.european-processor-initiative.eu/v-for-vector-software-exploration-of-the-vector-extension-of-risc-v/.

[28] SIMD < SIMT < SMT: parallelism in NVIDIA GPUs. URL: https://yosefk.com/blog/simd-simt-smt-parallelism-in-nvidia-gpus.html.

[29] Dirk Grunwald, Mary Jane Irwin, and Trevor Mudge. Kool chips workshop. Monterey, California, December 2000. URL: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.23.2424&rep=rep1&type=pdf.

[30] John Paul Shen and Mikko H. Lipasti. *Modern Processor Design: Fundamentals of Superscalar Processors*. Waveland Press, July 2013. Google-Books-ID: ffQqAAAAQBAJ.

[31] S. Manne, A. Klauser, and D. Grunwald. Pipeline gating: speculation control for energy reduction. In *Proceedings. 25th Annual International Symposium on Computer Architecture (Cat. No.98CB36235)*, pages 132–141, July 1998. ISSN: 1063-6897. doi:10.1109/ISCA.1998.694769.

[32] J. Shinde and S. S. Salankar. Clock gating — A power optimizing technique for VLSI circuits. In *2011 Annual IEEE India Conference*, pages 1–4, December 2011. ISSN: 2325-9418. doi:10.1109/INDCON.2011.6139440.

[33] Jason Casmira and Dirk Grunwald. Dynamic Instruction Scheduling Slack. December 2000.

[34] Brian Fields, Rastislav Bodık, and Mark D Hill. Slack: Maximizing Performance Under Technological Constraints. page 12.

[35] Yau Chin, J. Sheu, and D. Brooks. Evaluating techniques for exploiting instruction slack. In *IEEE International Conference on Computer Design: VLSI in Computers and Processors, 2004. ICCD 2004. Proceedings.*, pages 375–378, San Jose, CA, USA, 2004. IEEE. URL: http://ieeexplore.ieee.org/document/1347949/, doi:10.1109/ICCD.2004.1347949.

[36] T. D. Burd and R. W. Brodersen. Energy efficient CMOS microprocessor design. In *Proceedings of the Twenty-Eighth Annual Hawaii International Conference on System Sciences*, volume 1, pages 288–297 vol.1, January 1995. doi:10.1109/HICSS.1995.375385.

[37] Thomas D. Burd and Robert W. Brodersen. *Energy Efficient Microprocessor Design*. Springer Science & Business Media, 2002. Google-Books-ID: kmOtPUV8ijsC.

[38] M. Hamada, M. Takahashi, H. Arakida, A. Chiba, T. Terazawa, T. Ishikawa, M. Kanazawa, M. Igarashi, K. Usami, and T. Kuroda. A top-down low power design technique using clustered voltage scaling with variable supply-voltage scheme. In *Proceedings of the IEEE 1998 Custom Integrated Circuits Conference (Cat. No.98CH36143)*, pages 495–498, May 1998. doi:10.1109/CICC.1998.695026.

[39] M. Igarashi, K. Usami, K. Nogami, F. Minami, Y. Kawasaki, T. Aoki, M. Takano, S. Sonoda, M. Ichida, and N. Hatanaka. A low-power design method using multiple supply voltages. In *Proceedings of 1997 International Symposium on Low Power Electronics and Design*, pages 36–41, August 1997. doi:10.1145/263272.263279.

[40] Martin Angers. mna/pigeon, November 2020. original-date: 2015-04-05T14:55:20Z. URL: https://github.com/mna/pigeon.

[41] template - The Go Programming Language. URL: https://golang.org/pkg/text/template/.

[42] An Even Easier Introduction to CUDA, January 2017. URL: https://developer.nvidia.com/blog/even-easier-introduction-cuda/.

[43] Matrix-Matrix Multiplication on the GPU with Nvidia CUDA | QuantStart. URL: https://www.quantstart.com/articles/Matrix-Matrix-Multiplication-on-the-GPU-with-Nvidia-CUDA/.

[44] Eigen, June 2021. URL: https://eigen.tuxfamily.org/index.php?title=Main_Page.