Escola Tècnica Superior d'Enginyeria Informàtica
Universitat Politècnica de València

Design and verification of a Systolic Array Multiplier

DEGREE FINAL WORK

Degree in Computer Engineering

Author: Pablo Andreu Cerezo
Tutor: José Flich Cardo
Experimental Tutor: Carles Hernandez Luz

Course 2019-2020

Resum

La intel·ligència artificial intenta resoldre molts dels problemes als quals s'enfronta la societat moderna. Però, perquè aquesta aconsegueixi guanyar popularitat i arribar a estar present en molts dels aspectes de les nostres vides, és necessari el desenvolupament de xips eficients dedicats a la inferència en xarxes neuronals , tenint la multiplicació eficient de matrius com a component essencial per a aquesta tasca.

Per això, en aquest treball s'afronta el disseny, verificació i caracterització d'un multiplicador de matrius compatible amb *AXI*. En aquest treball analitzem el fonament de la multiplicació de matrius de manera sistòlica, fins els passos necessaris perquè aquest sigui funcional dins el paradigma descrit. Com a resultat obtenim el nostre propi nucli RTL compatible OpenCL capaç de multiplicar matrius en una FPGA i d'aquesta manera fàcilment desplegable a un *ASIC*.

Paraules clau: Matriu sistòlica, FPGA, Verilog, FloPoCo, AXI, Vitis, OpenCL

Resumen

La inteligencia artificial intenta resolver muchos de los problemas a los que se enfrenta la sociedad moderna. Pero, para que esta consiga ganar popularidad y llegar a estar presente en muchos de los aspectos de nuestras vidas, es necesario el desarrollo de chips eficientes dedicados a la inferencia en redes neuronales, siendo la multiplicación eficiente de matrices esencial para esta tarea.

Por ello, en este trabajo se afronta el diseño, verificación y caracterización de un multiplicador de matrices compatible con *AXI*. Nos proponemos analizar desde el fundamento de la multiplicación de matrices de manera sistólica, hasta los pasos necesarios para que este sea funcional dentro del paradigma descrito. Como resultado obtenemos un kernel *RTL* lanzable desde OpenCl capaz de multiplicar matrices en una *FPGA* y siendo este fácilmente desplegable en *ASIC*.

Palabras clave: Matriz sistólica, FPGA, Verilog, FloPoCo, AXI, Vitis, OpenCL

Abstract

Artificial intelligence aims to solve much of the problems of the contemporary society that we live in. But, in order for it to be ever so prevalent, the development of efficient inference-specific chips is needed, being matrix multiplication at the core of neural network inference.

So, in this work, the design, verification and characterization of an AXI-Compliant matrix multiplier will be reviewed. We range from the systolic array paradigm, to the needed steps and modules to make it fully functional, finally reaching an open-CL launchable matrix multiplication kernel that can be tested on FPGA and can be easily ported for ASIC usage.

Key words: Systolic array, FPGA, Verilog, FloPoCo, AXI, Vitis, OpenCL

Contents

Contents **v**
List of Figures **vii**
List of Tables **viii**

1 Introduction **1**
1.1 Motivation 1
1.2 Objectives 2
1.3 Expected impact 2
1.4 Methodology 3
1.5 Memory structure 3

2 Background **5**
2.1 The Traditional CPU Model 5
2.2 The Need for Accelerators (Heterogeneous Computing) 7
2.2.1 Floating Point Unit 7
2.2.2 Systolic Array Organization 9
2.2.3 Quantization 11
2.3 Implementation Tools 12
2.3.1 FPGA Tools 12
2.3.2 Libraries 12

3 State of the art **17**
3.1 Widespread Dedicated Neural Network Accelerators 17
3.1.1 Nvidia Tensor Cores 17
3.1.2 Google TPU 18
3.1.3 Graphcore IPU 20
3.2 Widespread FPGA Neural Network Accelerator designs 21
3.2.1 HLS Designs 21
3.2.2 Xilinx DPU core 22
3.3 Proposed design place on the current state of the art 23

4 Architectural design **25**
4.1 Basic element: MAC unit 27
4.1.1 Internal MAC unit architecture 27
4.1.2 Data loading process 29
4.1.3 Flow Control Mechanism 30
4.2 Our systolic array design 30
4.2.1 Overview example of the process of the proposed matrix multiplication unit 31
4.2.2 Systolic Array control submodules 34
4.3 AXI 36

5 Performance Evaluation **37**
5.1 Testing methodology 37
5.1.1 Simulation of the design on Vivado 37
5.1.2 Simulation of the finished result on Vitis 37

5.2 Results and discussion 39
5.2.1 Performance 39
5.2.2 Resources analysis 41
5.2.3 FloPoCo tuning and its performance impact 43
5.3 Conclusion 46
6 Conclusions **47**
6.1 Connection between the Computer Science degree and the presented work 48
7 Future work **49**
7.1 Improvements on current design 49
7.2 Pipelining usage improvement 49
7.3 A matrix caching mechanism for partial multiplications 50
7.4 Variable precision arithmetic 51
Bibliography **53**

Appendix
A Definitions, terminology and acronyms **55**

List of Figures

2.1 The MIPS processor data-path. 5
2.2 The AMD Zen microarchitecture. 6
2.3 Single precision floating point representation. IEEE-754 standard. 8
2.4 bfloat16 floating point representation. 9
2.5 Matrix multiplication example. 9
2.6 Systolic array organization. Matrix multiplication example. Initialized system. 10
2.7 Systolic array organization. Matrix multiplication example, first step. . . . 10
2.8 Systolic array organization. Matrix multiplication example, second step. . 10
2.9 Systolic array organization. Matrix multiplication example, third step. . . 10
2.10 Systolic array organization. Matrix multiplication example, fourth step. . 10
2.11 A quantization example. 11
2.12 Single precision floating point representation. The FloPoCo way. 13
2.13 Passively cooled Alveo U200 card. 15

3.1 Mixed precision multiplication with NVIDIA Tensor Cores. 17
3.2 Half precision floating point representation. IEEE-754 binary16 standard. 18
3.3 Google's TPU version 1 system architecture overview. 18
3.4 Google's TPU version two and three system architecture. 19
3.5 Coral edge for production TPU models as per march 2020. 20
3.6 IPU architectural overview diagram. 21
3.7 DPU Top-Level Block Diagram. 23

4.1 System representation of the proposed design. 26
4.2 MAC operator sub-module interface and basic operation. 27
4.3 FPU module internal 28
4.4 FPU loading mechanism. First cycle. 29
4.5 FPU loading mechanism. Second cycle. 29
4.6 FPU loading mechanism. Third cycle. 29
4.7 Valid-Ready data transfer timing diagram example. 30
4.8 An example of a matrix multiplication. 31
4.9 An example of a matrix multiplication. First step. 32
4.10 An example of a matrix multiplication. Second step. 32
4.11 An example of a matrix multiplication. Third step. 32
4.12 An example of a matrix multiplication. Fourth step. 33
4.13 An example of a matrix multiplication. Fifth step. 33
4.14 An example of a matrix multiplication. Sixth step. 33
4.15 An example of a matrix multiplication. Seventh step. 34
4.16 An example of a matrix multiplication. Eight step. 34
4.17 Variable FIFO queue depth representation. 35

5.1 Results of the Vitis hardware emulation. 38
5.2 Vitis application timeline results. 38
5.3 Performance impact of different A multiplications against a B matrix loaded. 41

5.4 Percentage of utilization of Alveo U200 resources. 42
5.5 Resource usage of Systolic array module and FIFO queue subsystem depending on size of the Systolic Array. 43
5.6 Performance bottleneck of the default FloPoCo pipelining. 45
5.7 FLOPS per target frequency for an 8×8 Systolic Array. 45
5.8 Ideal maximum FLOPS per target frequency and pipelining usage for an 8x8 Systolic Array. 46

7.1 Sub-matrix multiplication example for MAC unit pipelining. 50
7.2 Rudimentary proposed Systolic Array caching mechanism. 50

List of Tables

4.1 Valid-Ready control flow mechanism possible states. 30

5.1 Size and performance comparison of the proposed design. 40
5.2 Resource usage of Alveo U200 according to size of the design. 42
5.3 Percentage of resources taken by the Systolic Array part of the design. . . 43
5.4 FloPoCo initial approach synthesis and performance results. 44
5.5 FloPoCo with tuned target frequencies synthesis and performance results. 44

CHAPTER 1
Introduction

Neural network inference is at the core of machine learning inference and training. With the recent advances and the current trends of adapting almost everything to machine learning, accelerating those critical workloads becomes extremely important.

For accelerating AI (Artificial Intelligence) workloads, floating point matrix multiplication becomes essential and it is considerably complex computationally speaking and non-trivial to parallelize for large matrices. AI has been accelerated by the usage of parallelization, but is still far from being perfectly efficient. Therefore, alternative chip architectures for solving this efficiency problem have been developed over the last few years.

One potential architecture for multiplying matrices is the Systolic Array design. The Systolic Array organization is what our design uses as its foundation as is one of the most efficient architectures for matrix multiplication with data reuse. A complete explanation of the Systolic Array architecture, our selected Systolic Array design and an in-depth view onto the challenges, performance and design choices of our design are exhibited in this work.

1.1 Motivation

The market for AI hardware acceleration has been consistently rising for the past years, but there are still some ways in which it can be greatly improved. When we proposed this design we looked at the existing alternatives, and, even though this field is extremely saturated with innovation, the Open-Source alternatives for implementing this were far fewer than expected, making every one of the existing ones difficult to implement on within our existing objectives. Those core objectives where:

- For the design to be Open Source.
- For the design to be easily converted into ASIC (Application-specific integrated circuit), with this meaning that our design must be DSP-independent.
- For the design to be easily adapted for its use as a RISC-V arithmetic co-processor.

We couldn't find existing projects that met that criteria, mainly because the use of DSPs on FPGA designs is incredibly prevalent due to the current trend of HLS (High Level Synthesis) design instead of the traditional RTL (Register Transfer Logic) that is much more easily translated into silicon.

Therefore, we took this as an opportunity to create a brand new innovative matrix multiplication unit to contribute to the AI acceleration field. Indeed, we designed a sys-

tem to be considered as a proof of concept so that can later be developed into a competitive matrix multiplier. This opportunity was great for me personally as hardware design has been always at my heart and further expanding my knowledge of the field is one of my passions. So this project will let me learn some industry-standard ways of communication such as the AXI protocol and enrich my understanding of FPGA programming.

Another great motivator for this work was the possibility to work in an investigation environment, with great coworkers and mentors that offered great support and let me fully develop my skills in the best possible way. Being this an opportunity to grow both personally and professionally.

1.2 Objectives

So, once we had the motivation figured out, we defined the following objectives:

- The design had to be Open Source and certifiable.
- The design had to be made on RTL instead of HLS.
- The design must multiply single precision floating point matrices.
- The design had to implement a Systolic Array for efficient matrix multiplication.
- The design must be able to be attached to an AXI bus for processor communication.

With those objectives in mind, we thoroughly analysed the different flavours of Systolic Array implementations and AXI bus attachments and opted for defining some secondary objectives, listed next:

- The design had to implement FloPoCo as an arithmetic library, with arithmetic libraries being easily interchangeable. This allows for a flexible adaptation to custom precision formats and different floating point libraries.
- The design had to be able to be easily configurable for different precision arithmetic and mixed-precision arithmetic (potentially achieved by the use of FloPoCo).
- The design had to be tested and implemented on the Alveo U200 platform.
- The Systolic Array design should be implemented as Google's TPU v1 implemented it, thus, reducing the required AXI bus bandwidth.
- At the end of the design, it must be tested and benchmarked, making sure there was no possible case of unplanned matrix multiplication failure.
- The design must be able to easily be expanded for more complex capabilities, so a modular design must be provided.

1.3 Expected impact

Related with the expected impact this design could have on our modern society the possibilities are endless. AI is starting to become ubiquitous and has endless applications. From object tracking or recognition to autonomous driving. All those options could be accelerated with a design like the one proposed here, although with some major improvements on performance and size.

A possible development with this chip would be that of autonomous vehicles, or self driving vehicles, as well as a human-machine interface, that, if installed on a train could give the train driver some insight on what to perform on each moment, warning him on possible dangerous situations even if those are not yet visible for such driver. Indeed, the goal is to use the developed multiplier on the H2020 SELENE European Project, which is lead by the advisors of this project and where the author of the project will further collaborate and extend the work presented here. The SELENE project aims to bring AI support to three challenging use cases, ranging from automated trains to autonomous driving.

Regarding the UN (United Nations) sustainable development goals that this project help further we shall outline a couple:

- Good Health and Well-Being (Goal 3)
- Sustainable Cities and Communities (Goal 11)

Those are selected although with artificial intelligence the possibilities and areas of application are endless. We selected those goals as they perfectly suit with traditional AI applications.

Good Health and Well-Being are selected as they can be directly related with self driving cars, potentially leading to less road accidents and road clutter. Therefore, quality of life is greatly improved. Sustainable cities and Communities are selected as smart cities are becoming increasingly prevalent. Huge possibilities are envisaged on the front of stoplight automation that automatically meets traffic demands or detects accidents.

1.4 Methodology

For the methodology some early decisions had to be made. Firstly, the arithmetic library that we wanted to use. In that regard we chose FloPoCo due to it being entirely RTL and having selectable and adaptable pipelining. It met our Open Source and RTL objectives and provided single precision and mixed precision arithmetic support.

In terms of the AXI compatibility we chose the Xilinx's AXI wrapper submodule as it is a complete and fully tested component. Using the Xilinx AXI wrapper also enabled us to use the XRT and Vitis kernels to test our design on a higher level.

As for testing we chose to test the output with a simple Python script we developed and to use the Xilinx Vivado integrated waveform analysis tools to perform intra- and inter-module debugging.

As for the design of the main part of the work (the systolic Array) we developed self-sufficient MAC (Multiply ACcumulate) submodules that communicate to each other using the Valid-Ready interface. We chose this design as it is timing-agnostic inside the systolic array, and thus, simpler. It provided us backpressure support without any needed top-level control logic.

1.5 Memory structure

This document is structured as follows.

In chapter 2 we present all the concepts that are necessary to understand this work. From hardware concepts such as the function of a Floating Point unit and the most common Floating Point formats to the software libraries and tools used for this project.

In chapter 3 we present the most prevalent AI accelerators in both the ASIC and FPGA fields.

In chapter 4 we present the design of our matrix multiplier solution. From a general diagram of the inner workings of the module, to an in depth explanation of every submodule of the Systolic Array design.

In chapter 5 we thoroughly benchmark our design from various perspectives. We asses the size, achievable frequency and performance for various real-world and theoretical cases.

In chapter 6 we try to summarize the work done and put it into perspective with the objectives presented on this chapter, assessing the relation between this work and the Computer Science degree that it is presented in.

Finally in chapter 7 we present some possible optimizations that can be performed on this design to truly extract the maximum possible performance out of the FloPoCo generated operators and make it truly competitive.

CHAPTER 2
Background

In this chapter, we introduce all the basic concepts needed to understand this work. We describe the IEEE754 floating point standard and the specific format used in this work. We also introduce the systolic array architecture and show an example of a systolic array multiplication process to further understand its inner workings. Then we introduce the tools used to develop our design. Further, we present the tools that enable running our design on a modern PCI-compatible FPGA and the programming paradigm used.

2.1 The Traditional CPU Model

Since the early processors, the CPU (Central Processing Unit) has applied the same computing paradigm: First, reading data from memory and storing it into registers. Then, forwarding data into a Computing Unit with an opcode to obtain the desired result. Results are stored on a register for further computations to be performed. Finally, data is written back into main memory.

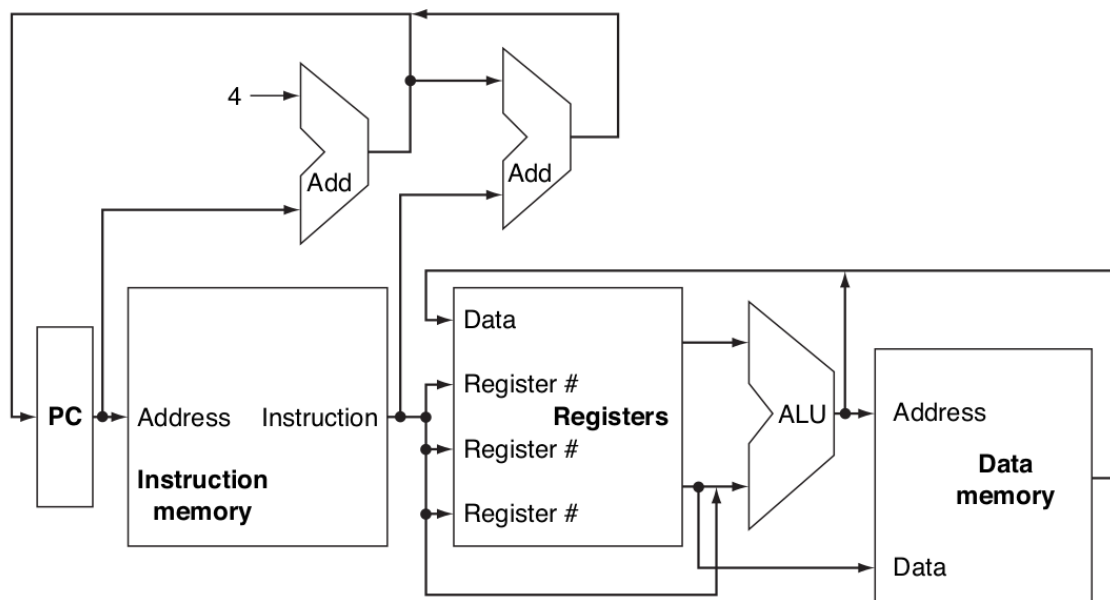

Figure 2.1: The MIPS processor data-path.

Source: [1, p. 302]

On Figure 2.1 we can observe the aforementioned process. We have a PC (Program Counter) register that points to the current instruction address to be run. Then, with that address we fetch the instruction from memory (Instruction memory in this case). Next, with the data that is coded on this instruction, this simple processor can do the following:

- Update the PC with a new instruction address. This is commonly called jumping to a new address. In Figure 2.1 we can observe the wire going from Instruction memory to the upper rightmost adder. There, the position of the program counter will be updated **n** positions in comparison from its previous address. Therefore, jumping to a new address with new instructions to be executed.
- Read data from a register and store it on the data memory.
- Read data from a register, perform an operation on the ALU (Arithmetic and Logic Unit). Storing it later on a register. Both registers (Source and Destination), and the operation to be performed on its data are encoded onto the instruction.

Figure 2.1 is obviously a simplified version of what we can see on modern processors, but illustrates all the principal parts of one. Modern processors have multiple arithmetic units, multiple cores, an entire memory hierarchy below the register level and many more optimizations.

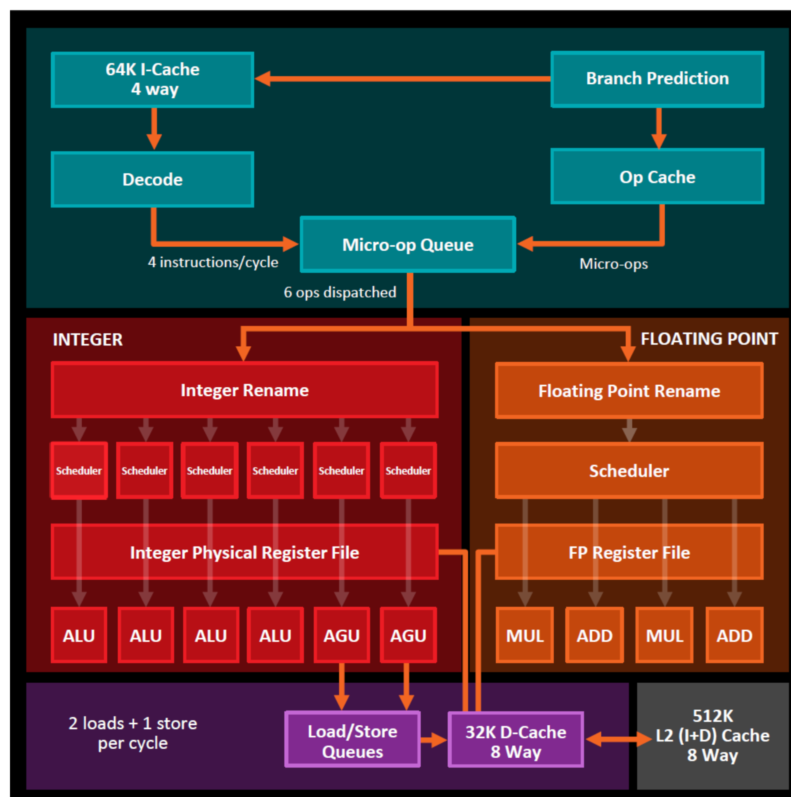

Figure 2.2: The AMD Zen microarchitecture.

Source: [2, p. 7]

On Figure 2.2 an example of a modern processor microarchitecture is shown. As we can see, it has a complex cache subsystem and speculative instruction execution, denoted by the integer and floating point register rename and schedulers. Those follow the algorithm described on [3]. But, the basic layout presented on Figure 2.1 is still there. This

means that we still have an Instruction Fetch, Decode and the operands passing through an ALU and being stored onto registers.

2.2 The Need for Accelerators (Heterogeneous Computing)

With the simple model described on Figure 2.1 we are heavily constrained on data parallelism (we can only compute as many operations as operators are available). This is indeed logical, but, the number of resources needed to make a sequential program essentially "Semi-parallel" is significant. The support for parallelism is described by Tomasulo on its famous paper [3]. It basically works by dynamically checking dependencies between instructions on a sequential program and executing in *parallel* those that do not depend on each other.

This process, however, is still an inefficient process, taking a great deal of die space on a chip for diminishing single core performance returns as a consequence of Moore's law [4, Section: Costs and curves] slowing on recent times.

This is the reason why application specific chip designs are increasingly emerging on these days. Specific compute units offer greater efficiency and speed gains on specific workloads and are becoming a *de facto* complement to modern CPUs.

In this specific workload accelerator area: Graphical Processing Units (GPUs), Tensor Processing Units (TPUs) and FPGAs are among the current options. They offer significant efficiency and power advantages in certain workloads vs a traditional CPU due to their inherently different design. That design is much less general purpose but accelerates a limited range of functions greatly due to their dedicated hardware to do so. Notice accelerated functions can also be performed by traditional CPUs, but in a much slower and less power efficient manner.

2.2.1. Floating Point Unit

Most applications using accelerators rely on real number processing. This is the notable case of artificial intelligence applications. For this reason we focus on accelerators that target floating point numbers.

A floating point unit is a computing unit dedicated to perform floating point operations. Floating point is a representation method for real numbers, where we have a base and an exponent coded into it, enabling the representation of huge numbers with a limited set of bits.

This enables the representation of larger than 2^{bits} numbers. Specifically, IEEE-754 single precision lets us represent numbers on the range of $\approx (-2^{127}, 2^{127})$. However, this range makes the representation of large exact numbers impossible. Nevertheless, this format is still worth being used for real numbers.

Floating point numbers can be represented in a wide variety of ways. From the basic ones described in this section and shown in Figure 3.2, to more complex ones such as the FloPoCo one described on subsection 2.3.2 and the one used by Google described on Figure 2.4. **In this project, we select the FloPoCo method.**

The IEEE-754 Floating Point Standard

IEEE-754 defines a wide variety of bit lengths and methods of defining floating point numbers on a binary representation. Those methods are very similar, and for the pur-

poses of this work, understanding the 32-bit floating point representation is enough. This representation is called the Single Precision Floating Point number representation.

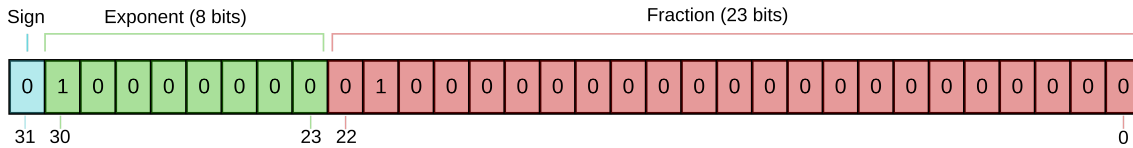

Figure 2.3: Single precision floating point representation. IEEE-754 standard.

The sign bit just represents the sign of the number, being zero for a positive floating point number and one for a negative one. Special values such as $+0$, -0, $+\infty$ and $-\infty$ are supported.

The exponent is a number represented using 127-excess format with two's complement. This means that a value of "00000000" is equal to -127 and a value of "11111111" is equal to 128. A zero exponent value equals to "01111111".

The fraction bits represent a number in the 1.fraction format, being the first bit the 2^{-1} and so on.

As an example, the following procedure is the decoding process of the number coded in Figure 2.3:

- Sign: The sign bit is decoded as positive.
- Exponent: The exponent bits are "10000000" or $2^7 = 128$. By encoding the 128 to excess 127, the following result is presented $128 - 127 = 1$. Making the final exponent 1.
- Fraction: Following the method described before we have a fraction equal to "01000-00000..." or $0 \times 2^{-1} + 1 \times 2^{-2} = 0.25$. But, per IEEE-754 standard definition, the mantissa number starts with an implicit bit set to one [5, p. 19], so the real mantissa is 1.25.

If we put all the previous numbers together we get $2^1 \times 1.25 = 2.5$, obtaining its decimal number representation.

With the previously explained format, special numbers such as zero, infinity, NaN (Not a Number) and numbers close to zero are not coded. For those, the IEEE-754 standard defines special combinations of bits to represent them. One of those combinations are the de-normalized, close-to-zero numbers. Those are coded with an exponent value of all zeros. With the fraction component codification being 0.fraction instead of the previous 1.fraction. This enables the representation of very small numbers.

Also, setting all bits to one on the exponent field has special meanings. An exponent of all ones with an all-zeros mantissa represents infinity, being -infinity possible with a negative sign field. Furthermore, an all-ones exponent with any mantissa bit set to one is considered NaN (not a number).

Bfloat16 number representation

The *bfloat16* format is an alternative representation format to the conventional IEEE754 half precision format shown on Figure 3.2. This format emerged by the need of energy-efficient methods with lower precision accuracy requirements.

This format is specially thought to be easily interoperable with the IEEE754 single precision format shown on Figure 2.3. This interoperability is given by both formats us-

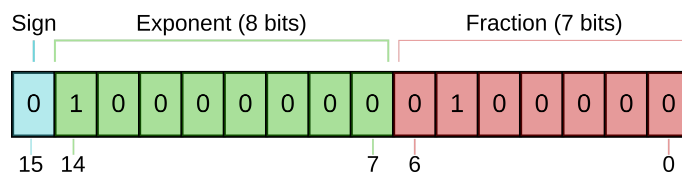

Figure 2.4: bfloat16 floating point representation.

ing the same number of bits on the exponent field. Making the conversion between the IEEE754 single precision and bfloat16 as trivial as truncating the lower bits of the mantissa and thus, sacrificing a bit of precision. The inverse is also trivial, for a conversion between bfloat16 and IEEE754 single precision, a mantissa padding with zeroes suffices.

This format makes mixed precision arithmetic functionally the same as single precision arithmetic inside an operator by eliminating the need for number translation hardware, being multiplication on *bfloat16* and addition on IEEE754 single precision possible and widely implemented with this system.

bfloat16 is widely used on neural network workloads due to its high range of values (as a result of its eight bit exponent). According to Google "it provides a better training and model accuracy than the IEEE half-precision representation" [6, TPU Versions]

2.2.2. Systolic Array Organization

In this work we deploy a systolic array module. Such arrays represent a highly efficient method to perform matrix multiplication operations and are used at the core of AI applications. For that reason, we describe here the basic foundations of systolic arrays, describing our slightly different implementation on section 4.2.

A systolic array works by flowing data from memory through an array of connected compute units, therefore reusing data read from memory. In our case, the matrix multiply operation, compute units are made of MAC (Multiply ACcumulate) units. By reusing data read from memory, this organization is much faster and efficient than a CPU. In particular, the Google TPU v1 was 62 times more efficient in performance per watt in comparison with a traditional CPU [7, p. 7].

We illustrate an example of the matrix multiplication on Figure 2.5 performed on a systolic architecture.

$$A = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \times B = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} = C = \begin{bmatrix} 7 & 10 \\ 15 & 22 \end{bmatrix}$$

Figure 2.5: Matrix multiplication example.

A first analysis of the systolic architecture shown on Figure 2.6 shows us four compute units interconnected on a systolic array manner. Each compute unit takes two values, one from top and one from left and multiply them, storing the result on a built-in accumulator register. Notice that each compute unit will operate with the input values when they are both available.

Then, the data fed to the unit is passed onto the next unit. Making the data flow inside the systolic array without the need to use store operations and named registers.

As shown on Figure 2.7 the data is fed to the systolic array from top and left, flowing on the direction indicated by the arrows. So the red data is multiplied to the blue data on

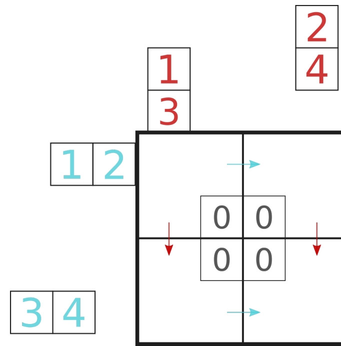

Figure 2.6: Systolic array organization. Matrix multiplication example. Initialized system.

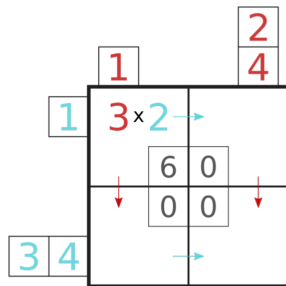

Figure 2.7: Systolic array organization. Matrix multiplication example, first step.

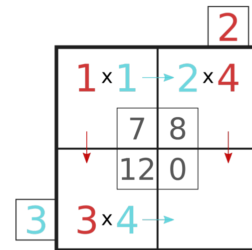

Figure 2.8: Systolic array organization. Matrix multiplication example, second step.

each unit and flows downwards, while the blue data will flow rightwards. The result of the performed multiplications is accumulated into each units memory as shown above.

Then, on Figure 2.8 the systolic data flow can be seen clearly. The value 2 that was on the upper leftmost unit flows into the upper rightmost unit, being multiplied by the value 4 coming from above. This process will be repeated with all data fed into the matrix and is a symmetrical process.

This "symmetry" means that the same process is occurring on every compute unit in an identical manner, making conceptually the scaling of this units extremely simple. Larger size arrays means further reusing data and reducing access to memory.

In this example the compute units perform multiply and accumulate operations, but they could perform a wide range of possible operations. What should be noted from this example is the way the data flows inside the array, and not the specific operation performed inside each compute unit.

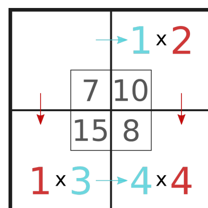

Figure 2.9: Systolic array organization. Matrix multiplication example, third step.

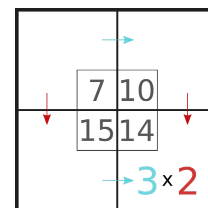

Figure 2.10: Systolic array organization. Matrix multiplication example, fourth step.

Finally, on Figure 2.9 and Figure 2.10 the last two steps of this process can be seen. The data flowing outside the array could be fed onto another array or simply discarded, being that outside the scope of this example and being implementation-dependant.

On section 4.2 our specific implementation of the systolic array architecture will be introduced.

2.2.3. Quantization

Quantization consists on converting the standard 32-bit values into 8-bit values by mapping them inside a range. Quantization is useful to neural networks as values inside a neural network tend to stay between a small range, for example between -8.342f to 23.354f. Using quantization, we map the least value to 0 and the maximum value to 255 so that instead of having a smooth function we have a less-smooth but precise-enough representation.

This quantization method is supported and widely used on multiple AI acceleration designs, such as the Nvidia Tensor Cores [8, Turing Tensor Cores] as well as in the Google TPU v1 (see subsection 3.1.2) and has become an incredibly efficient mechanism for number representation in neural networks.

By using this method, each quantized step would be of 0.12381549523f, so loosing some precision is possible, but in neural networks, high precision needs tend to be overshadowed by the large exponent disparity of weights.

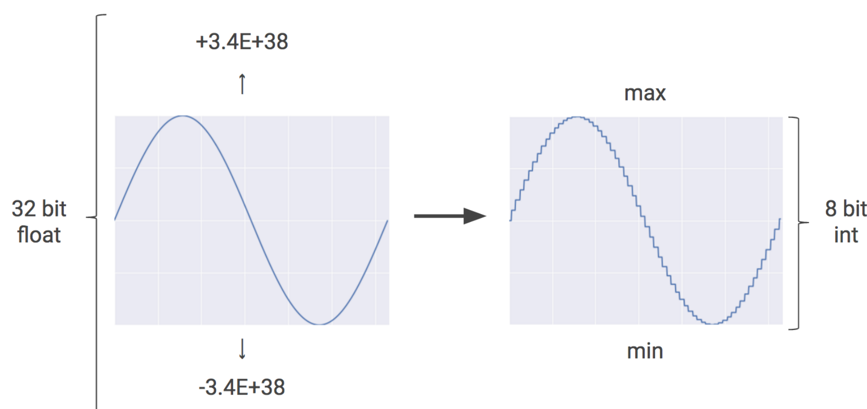

Figure 2.11: A quantization example.

Source: [9, Quantization in neural networks].

By using quantization we are obviously losing some precision. This is merely a consequence of using less bits to represent a range of numbers. This precision loss is not significant when dealing with non-zero values. However, we often use zeros as padding when dealing with systolic arrays or to initialize certain values or represent certain decisions.

This loss of precision can be observed on the previously mentioned quantization example with 8 bits over [-8.342, 23.354]. Where zero would be the integer 67 and represent -0.047f. Here we can see how the loss of precision on the zero over a large quantity of numbers can cause a large drift on the final values.

The zero value problem only arises due to the non-zero representing value on the quantized values, this can easily be solved by forcing one quantized value to zero or can be ignored. The path of ignoring the non-precise zero values seems to not cause many issues with the most common neural networks and algorithms where a few decimal places in precision drift is not that important, though depending on the different rounding policies and bit lengths used on the quantization phase, the results can vary wildly [10].

In this project we selected the FloPoCo floating-point unit to implemenent our FPU units. This allows us to play with different quantization algorithms and methods in our systolic array solution. Indeed, this is one of the grand goals we pursue in this project as we plan to use the provided solution in the H2020 SELENE project.

2.3 Implementation Tools

In this section we describe the software tools and libraries used for the project. We target the deployment of our systolic array in FPGAs. Therefore, we describe here the tools used to program FPGAs as well as the available libraries to ease our designs.

2.3.1. FPGA Tools

As of FPGA tools we use both the Xilinx Vivado IDE tool and the Xilinx Vitis IDE tool. In the next paragraphs, we will describe both of them.

Xilinx Vivado is a hardware design and verification software that we use extensively in our design. Vivado enables us to organize the code, view schematics, calculate achievable frequencies for different pipelined configurations and see the internal signal values for a specific time in our design.

With this software, in addition to the high expressive power of System Verilog, we verify our design with extensive test bench runs. Vivado simplifies debugging hardware by setting custom waveform configuration files. Those let us debug and compare individual signals in specific critical times and check whether the result of a module was as expected.

"Xilinx Vitis Unified Software Platform is a tool that combines all aspects of Xilinx software development into one unified environment"[11]. This translates into a tool that lets the FPGA programmer design circuits on a HLS (High Level Synthesis) programming language like Cpp or OpenCL. Those circuits, that we call kernels, can be launched from an OpenCL code running on the host machine. This, in theory, makes the use and design of FPGA much easier and comparable to a GPU accelerated workflow for the programmer.

HLS design is, in theory, a simpler and faster design process than traditional RTL design. In practice, there is a handful of pragmas that you can use in order for Vitis to infer an efficient design. The need, however, for maximum efficiency, configurability and control is what motivated us to design our circuit on traditional RTL languages. This RTL design gives us much more control to leverage the resources that are available to us on the FPGA silicon, giving our design plenty more reconfiguration potential.

Vitis, in addition to being able to infer RTL designs from HLS descriptions, is able to package RTL designs as kernels to be enqueued from the OpenCL code on the host machine.

2.3.2. Libraries

FloPoCo

"FloPoCo (Floating-PointCores, but not only) is an open-source C++ framework for the generation of arithmetic data-paths. It provides a command-line interface that inputs operator specifications, and outputs synthesizable VHDL." [12]

We extensively used this framework to generate the core of our systolic array, the MAC (Multiply ACcumulate) arithmetic unit. This arithmetic unit is composed of FloPoCo generated operators interconnected with HDL.

We specifically use FloPoCo for its enormous configuration potential and frequency specific pipelining. This frequency specific pipelining lets us design a circuit where we dictate a target frequency for the FloPoCo operators and the tool automatically generates latches inside those operators when necessary and in a balanced way.

Using this characteristic we can leverage the full high frequency potential of our FPGA or ASIC design moving the critical path outside of the arithmetic units without much design effort on the designer part, ensuring simple, efficient and well performing arithmetic operators.

We chose to use FloPoCo instead of conventional DSPs due to its configuration potential. It allows us to perform the following tasks in an IP core-less, and thus FPGA vendor independent way:

- Choose target frequency: The designer just needs to select a target frequency and FPGA family and the custom pipelining is done automatically by the tool. Making the design simpler and much more easy to reconfigure and fine-tune for our specific needs and targets.

- Choose input and output precision: Due to our design aiming to target neural network inference, lower than single precision floating point or mixed precision floating point can be needed. FloPoCo lets us perform those tasks in an easily parameterizable way. Making the design of a floating point multiplier with input precision of 32 bits and output precision of 16 bits almost trivial. It is as simple as setting the input fraction precision to 23 bits and input exponent precision to 8 bits. Then, setting the output fraction precision to 7 bits and exponent precision to 8 bits. Calling then FloPoCo with those parameters a fully pipelined design is completed and ready to be used in our design.

FloPoCo Floating-point Format

The FloPoCo floating point format is slightly different from the IEEE-754 single and double precision standard. In IEEE-754 single precision the bit definition is the one shown on Figure 2.3. In FloPoCo, a single precision floating point number binary codification would be as shown on Figure 2.12.

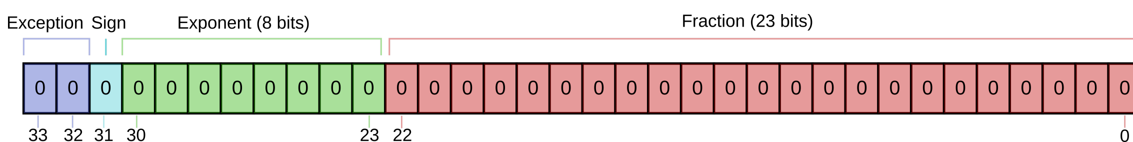

Figure 2.12: Single precision floating point representation. The FloPoCo way.

This format is used to avoid the special cases defined on subsection 2.2.1. Instead, special cases are coded on the upper two bits. This leads to a uniform floating point format of $1.Mantissa \times 2^{exponent}$. This uniformity provides speed due to the suppression of decoding mechanisms inside the operators.

The Exception field codification is as follows:

- "00": Zero, independently of the rest of the fields. Sign bit determines the sign of the zero. Positive and negative zeroes are supported.

- "01": Normal number, as described on subsection 2.2.1.

- "10": Infinity. Sign bit determines the sign of the infinite. Positive and negative infinities are supported.

- "11": NaN. The rest of the bits are ignored.

AXI

AXI (Advanced eXtensible Interface) [13] is part of the ARM AMBA specification that defines a parallel high-performance, multi-master, multi-slave communication interface, mainly designed for on-chip communication.

We use AXI for communicating our design with the external world, making it usable in a wide-range of environments. This bus lets the designer communicate with external memory in order to feed and write the processed data into the onchip and offchip memory.

Also, we need to consider that AXI defines several communication protocols that we use extensively internally and externally on our design. We will be using AXI for the external communications and AXI-Streaming interface for internal communications with the reader and writer module.

OpenCL

> OpenCL (Open Computing Language) is an open royalty-free standard for general purpose parallel programming across CPUs, GPUs and other processors, giving software developers portable and efficient access to the power of these heterogeneous processing platforms. [14]

Specifically we use OpenCL to leverage the parallel nature of FPGAs, being able to implement an RTL design as a callable OpenCL kernel. This makes the flow of information between host and kernel relatively transparent to the programmer. Further development on the inner workings of FPGA kernels will be presented on the following section.

The Xilinx Runtime

The Xilinx Runtime (XRT) [15] is a combination of userspace and kernel driver components. XRT supports PCIe based accelerator cards and provides an standardized software interface to Xilinx FPGAs. XRT lets us program the FPGA independently of its underlying configuration and technology (only applicable to Xilinx FPGAs), offering the needed abstraction to make the code easily portable between platforms such as the Alveo platforms or the Amazon F1 FPGA platform. With the Xilinx Runtime paired with the Vitis environment, HLS designs can be easily called and implemented in various platforms. It simplifies the use of external and internal memory thanks to the hardware abstraction provided by the XRT.

In addition to the previously mentioned aspects, once the kernels are compiled, they can be queued onto the execution environment by the Kernel Domain Scheduler (KDS), making the KDS the execution of the queued tasks onto the available compute units on an FPGA. This lets the programmer use as much space as available on the FPGA fabric, making efficient use of the available hardware.

The Alveo Platform

The Alveo platform is the PCI-compatible Xilinx FPGA lineup, supporting multiple workloads depending on the card selected. All these PCI compatible FPGA cards come with 100Gb Ethernet and in different configurations depending on the needed workload.

This Alveo card and ecosystem are designed to provide datacenters diferent kinds of performance gains on FPGA accelerated loads. From the U25 that aims to perform smart and fast packet switching and act as a smart NIC (Network Interface Card) to the U280, that has a great amount of LUTS, Registers and DSP Slices, as well as both HBM2 and DDR4 memory to perform a wide variety of tasks and be extremely flexible. Thus, leveraging the flexible nature of FPGA designs. The whole Alveo lineup specifications are published on the following reference [16].

This Alveo platform tries to leverage the fast design times of HLS (High Level Synthesis) to achieve fast design turnaround times and adapt to ever-changing demands on the business world. It achieves this by using OpenCL as it is widely understood, parallel, and scalable language that is also compatible with a wide variety of existing hardware such as GPUs and CPUs.

Figure 2.13: Passively cooled Alveo U200 card.

Source: Xilinx website.

CHAPTER 3
State of the art

In this chapter, the state of the art on current neural network accelerators is discussed. We range from the GPU tensor cores to the Google's TPU. We perform an overview of both ASIC chips and FPGA designs, describing each one's advantages and disadvantages.

3.1 Widespread Dedicated Neural Network Accelerators

In this section we present some of the widespread accelerators available in the market. They all have a strong correlation to the systolic array multiplication methodology described on subsection 2.2.2.

3.1.1. Nvidia Tensor Cores

The tensor core approach is the approach made by NVIDIA to tackle the problem of AI inference and training.

Tensor cores implement a wide array of optimizations for matrix multiplication. For instance, they offer mixed precision: FP16 and FP32. This means that an NVIDIA Tensor GPU can multiply two FP16 matrices and accumulate them into an FP32 one, loosing less precision than if the process were to be performed on the FP16 format solely. This process is depicted on Figure 3.1

Figure 3.1: Mixed precision multiplication with NVIDIA Tensor Cores.

Source: NVIDIA on [17, p. 20]

This FP16 multiplication also means a significant speedup over FP32 multiplication. The gains of this process are further increased if INT16 or INT8 formats are used. Those formats can be used to represent numbers on a range of floats in a process called quantization described on subsubsection 2.2.3.

The FP16 format shall be described as the same way of operation as described on subsection 2.2.1 but, instead of using the bit order described on Figure 2.3 (IEEE754 single precision), it uses the bit order described on Figure 3.2 (IEEE754 half precision).

Figure 3.2: Half precision floating point representation. IEEE-754 binary16 standard.

3.1.2. Google TPU

TPU v1

The google TPU is the systolic array implementation developed by Google on 2015. Since then, many iterations over the same architecture have been performed.

The first approach by Google with its TPU v1 was to create a systolic array multiplier with 256x256 as its size. This approach was to use the previously described quantization method in synergy with 8 bit integer operators to perform 65536 concurrent MAC operations.

Figure 3.3: Google's TPU version 1 system architecture overview.

Source: [9, RISC, CISC and the TPU instruction sets]

It also counted with a 700 MHz clock speed and 8 GiB of dual channel DDR3 RAM and was built on the old 28nm process, consuming only 40W and thus, being available as a PCIe accessory to the server market [9]. It performs the activation functions after the systolic array efficiently and with an enormous reuse of data, using previous iteration data for the next iteration without the need for that data to exit the TPU's memory subsystem.

In terms of the inner workings of the TPU v1 architecture seen on Figure 3.3, it is worth noting that, as the weights are modified just once per batch, the required bandwidth is lower and thus, they are stored on the outside DDR3 memory on model loading. Before computation, weights are read from main memory and moved into the "Weight FIFO". This buffer lets the TPU prefetch the next batches weights while still computing the previous batch results.

The "Unified Buffer" holds the activations. There we store the input data and the previous results to perform further computations. Due to its high usage, high bandwidth and low latency is required, so out of chip storage is discarded and fast 24 MiB memory is used for this purpose.

Once all weights are loaded into the weight FIFO and the activation buffer is full of input values, a control signal is emitted that loads a layer of weights into the MXU (Matrix multiply unit), propagating input values through the MXU and into the accumulators, into the activation. This activation applies a simple activation function from a list of activation functions that include the Binary step or ReLU, normalizing the result and moving the output to the Unified Buffer to be reprocessed.

TPU v2 and TPU v3

The second generation TPU, was announced on may 2017 and counted with two 128x128 systolic arrays instead of the previous 256x256 one. This change was to mitigate the inefficiencies presented by the filling and emptying of a systolic array presented on section 4.2.

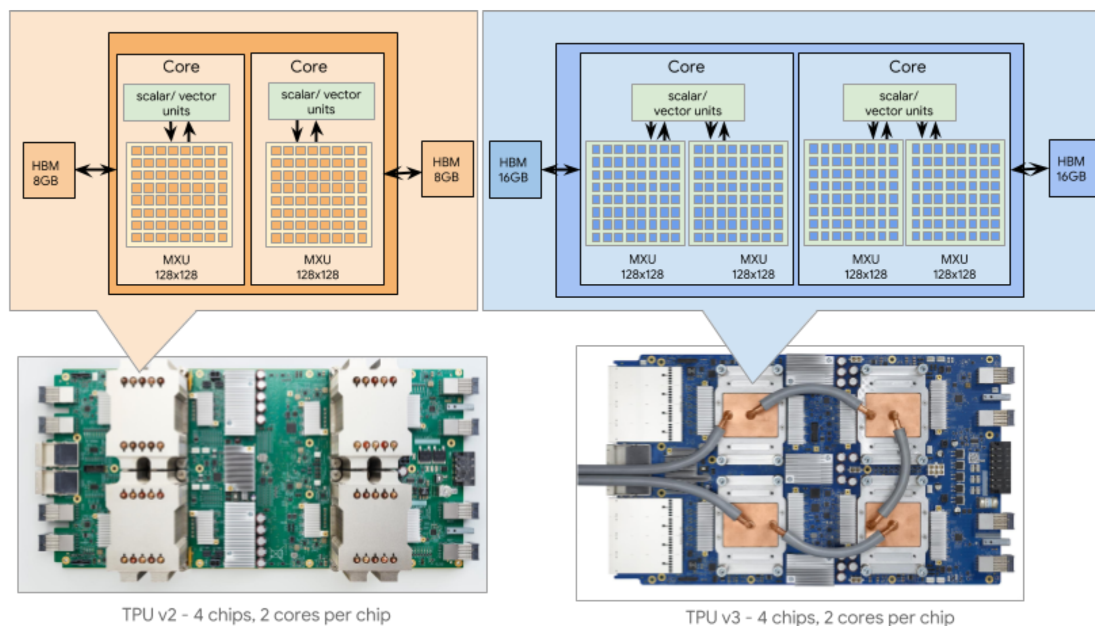

Figure 3.4: Google's TPU version two and three system architecture.

Source: [6, TPU Versions]

The last iteration of the TPU (The TPU v3) was presented on 2018 and featured two 128x128 MXU per core instead of the previous one 128x128 MXU per core. It also doubled HBM memory per core of 16 GB HBM compared to the previous 8 GB HBM of the TPU v2.

This TPU can be interconnected to train more complicated models in configurations that Google brands as TPU Pods. A TPU Pod is basically a cluster of TPUs connected over a dedicated network connection. This configuration allows for hosts with a TPU Pod connected to distribute the workload between the nodes of the cluster accordingly and efficiently, further accelerating workloads. TPU pod configurations can range from a single TPU v2 or v3 up to 64 TPU v2 or 256 TPU v3.

Edge TPU

The Google Edge TPU is a chip model that aims to perform inference on the edge. This means giving it a trained neural network and for it to perform real-time inference with it. This has endless applications, from face recognition on cameras, to real-time object tracking on an autonomous robot or synthesizing audio during a live musical performance.

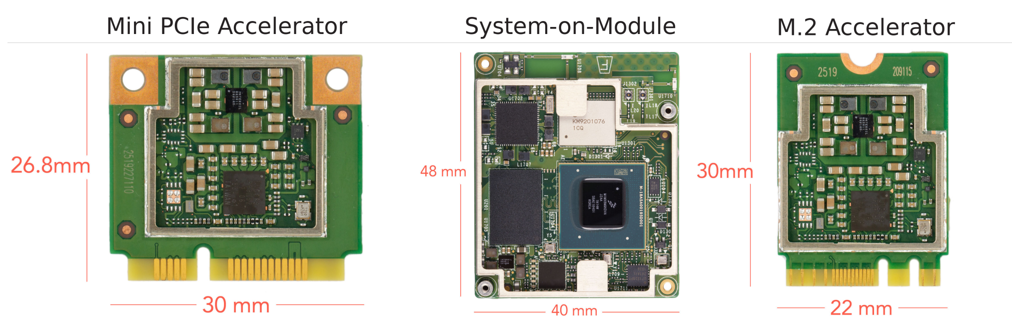

Figure 3.5: Coral edge for production TPU models as per march 2020.

Source: Coral for production models on [18]

The edge TPU manages to perform all these functions in an energy efficient manner (two TOPS per watt [18]) thanks to its Systolic Array architecture. Being available in a wide variety of form factors as an accelerator or as an standalone SoM (System on Module).

Coral Edge TPU has a peak throughput of four TOPS at two Watts. Offering a developer platform aside from the options shown on Figure 3.5 , being currently aiming to provide its accelerator as a surface mounted solution.

3.1.3. Graphcore IPU

"The Intelligence Processing Unit (IPU) is a novel, massively parallel platform recently introduced by Graphcore and aimed at Artificial Intelligence/Machine Learning (AI/ML) workloads"[19]

This Intelligence Processing Unit is designed for highly parallel and efficient execution of fine-grained operations, offering Multiple Instruction, Multiple Data parallelism and being distributed on its architecture. Each Intelligence Processing Unit contains 1216 elements that consist on one computing core and 256 KiB of local memory. Each of this elements is called a tile.

The IPU chip implements an interconnection network both between different IPU chips and between tiles inside a core. This interconnection consists of ten IPU link interfaces for IPU interconnection and two PCIe links for CPU interconnection as seen on Figure 3.6.

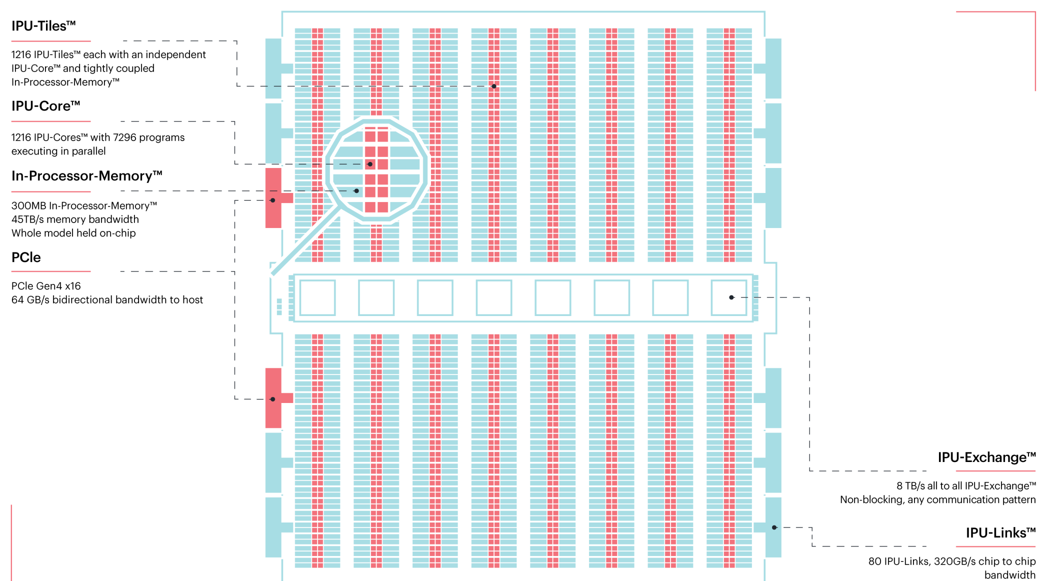

Figure 3.6: IPU architectural overview diagram.

Source: Graphcore's webpage, similar overview can be seen on [19]

The IPU manages the task of tile-independent code execution by not implementing shared memory. Instead the IPU implements 256 Kb of SRAM per tile, with that memory being exclusive, in opposition of the cache hierarchy of modern processors. This non-shared core-independent memory results on a great homogeneity on memory access times and great performance on irregular or random access pattern programs vs a traditional CPU or GPU.

The IPU is also capable of performing mixed-precision arithmetic in a FP32-FP16 manner, also incorporating systolic array-like structures called Accumulating Matrix Product (AMP) units presented on each tile. Those units offer the IPU a great matrix multiplication performance, being equivalent to a GPU. Graphcore claims better performance than some high-end commercial GPUs at the time of writting this document.

3.2 Widespread FPGA Neural Network Accelerator designs

3.2.1. HLS Designs

Xilinx Vitis unified platform

"The Vitis application acceleration development flow provides a framework for developing and delivering FPGA accelerated applications using standard programming languages for both software and hardware components."[11, p. 23].

Thus, the Vitis platform aims to provide easy development for the Xilinx Alveo lineup in different programming languages, such as C/C++ and OpenCL for the host application and C/C++, OpenCL C and RTL for the kernel. This, combined with the described tools on Figure 2.3.2 provides a modern flexible way of performing FPGA design and application development.

Intel/Altera FPGA Programming tools

The Intel FPGA ecosystem is extremely wide, offering a large array of tools for a large array of use cases. The main tools that we shall focus for this work are:

- Intel Quartus Prime software: This software is the Intel-equivalent at the Xilinx Vivado program used for this work. It offers the same features implemented in a slightly different manner. The details of the differences between Intel's approach to synthesis and place and route and Xilinx's approach are not relevant for this work, but shall be noted.

- Intel HLS Compiler: The Intel HLS compiler is the Intel solution to HLS hardware design, offering capabilities similar to those of the Vivado HLS solution and XOCC (Xilinx OpenCL Compiler).

- Intel FPGA SDK for OpenCL: This is the Intel solution to OpenCL kernel and host code paradigm. This solution competes with the Xilinx Vitis unified platform that we used extensively on our design.

Xilinx CHaiDNN HLS convolutional neural network accelerator

"CHaiDNN is a Xilinx Deep Neural Network library for acceleration of deep neural networks on Xilinx UltraScale MPSoCs. It is designed for maximum compute efficiency at 6-bit integer data type. It also supports 8-bit integer data type."[20]

CHaiDNN leverages both lower and fixed precision arithmetic and the HLS programming paradigm to obtain substantial performance. Performance figures, design, API and code can be found on the CHaiDNN Github repository at [20]

PipeCNN

PipeCNN is an FPGA HLS design that leverages the ease of use of OpenCL calls and design to perform Convolutional Neural Network computations. It works on Intel FPGAs and GPUs due to its OpenCL implementation.

Performance figures, design, API and code can be found on the PipeCNN Github repository at [21].

3.2.2. Xilinx DPU core

The Xilinx Deep Learning Processor Unit (DPU) core is an FPGA design core that is specially designed for convolutional network workloads.

It is highly parameterizable and flexible due to its FPGA based nature. Being the degree of parallelism a parameter and supporting most convolutional neural networks, such as VGG, ResNet, GoogLeNet, YOLO, SSD, MobileNet, FPN ... [22, p. 6].

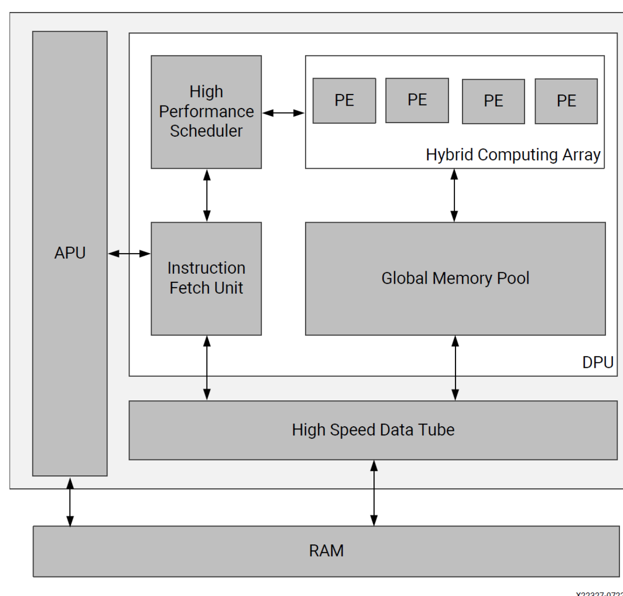

Figure 3.7: DPU Top-Level Block Diagram.

Source: DPU IP Product Guide PG338 [22]

As seen on Figure 3.7, the DPU is composed of several Processing Engines (PE), a global memory and control logic.

Those Processing Engines are highly optimized and able to use the Digital Signal Processor (DSP) slices available on the FPGA fabric. In addition to speeding up processing time with DSPs, Double Data Rate (DDR) clocking is used. In this specific case, one clock signal is used to drive the LUT based FPGA design, and the other clock signal is used to drive the DSP, resulting in a noticeable speedup in performance. [22, p. 12].

3.3 Proposed design place on the current state of the art

Although the proposed concept exists on a saturated market where multiple designs exist that perform similar tasks our design aims to provide DSP-less Open Source matrix multiplication. This is where the competition fades drastically. This occurs mainly due to DSPs offering greater performance than traditional LUT slices and designs aiming to provide the best possible performance for the least possible power consumption and size.

So for meeting the DSP-less and thus, ASIC compatible design, we sacrifice performance on the FPGA implementation, but, offer great configurability and the ability to fabricate our design inside a chip. I couldn't find a DSP-less, mixed and variable precision, AXI compliant, Open Source matrix multiplier design elsewhere. This might be due to the rising popularity of HLS, that tends to generate DSP heavy designs.

CHAPTER 4

Architectural design

For the architectural design of this project, an overview of the architecture will be shown, including the interconnection mechanism between modules and the intrinsic inner workings of our design.

In this chapter we will take a bottom up approach of the system overview presented on Figure 4.1. Thus, we first describe the inherent data dependencies of the Systolic Array and then, how our systolic control submodules fix them. The whole design of the systolic solution is shown in Figure 4.1. We will refer to this figure in different parts of this chapter.

Our systolic architecture will support two operating modes. First, the systolic array will receive specific commands to store a matrix of floating point numbers into the systolic grid. Each grid component will store a single value of the matrix. This matrix will be one of the matrices the array will multiply. Therefore, we pursue in our design a weight-centric approach where one of the matrices stays onsite in the systolic array. In the second operating mode the array will be horizontally fed with a matrix and this matrix will be multiplied on the fly by the stored one. The fed matrix will be injected horizontally and the resulting matrix will be flowing vertically and will output the systolic array from the bottom line of systolic components.

In both modes the matrices are injected always from left to right and sequentially. Each matrix row is injected to a systolic array row through the decoupling buffers and the associated FIFOs shown in the figure. Therefore, for a 16×16 systolic array, the native matrix multiplication operation will be performed by 16×16 input matrices. Of course, larger matrices can be multiplied by multiple runs on the systolic array.

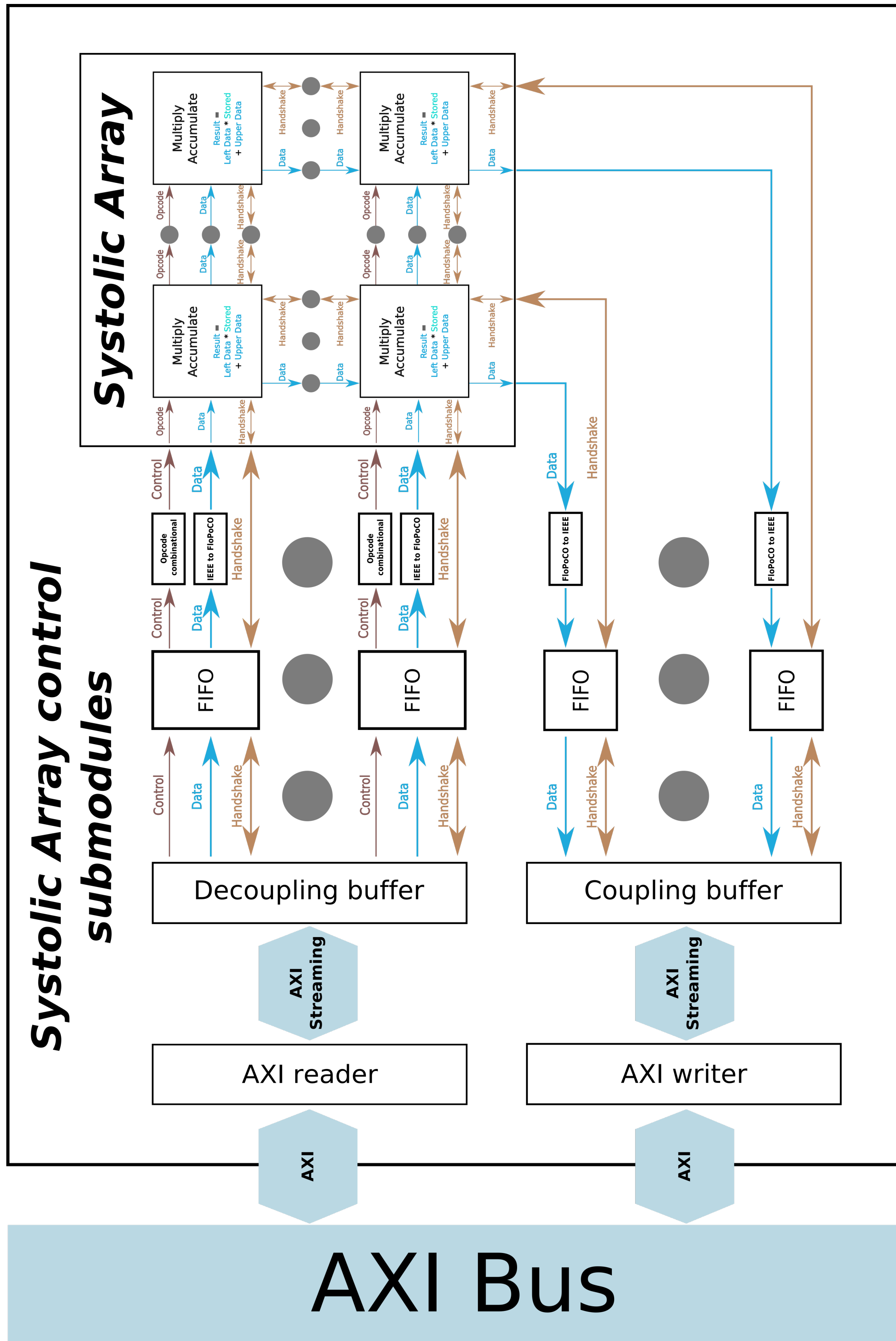

Figure 4.1: System representation of the proposed design.

4.1 Basic element: MAC unit

The multiply accumulate unit is the most basic unit in our design, and the one performing the computation. It is basically a unit that receives two numbers, multiplies one of these numbers (the one fed horizontally) by the one stored in the module, and then accumulates the result of that operation with the one fed vertically. The module interface can be seen on Figure 4.2 as well as its basic operation.

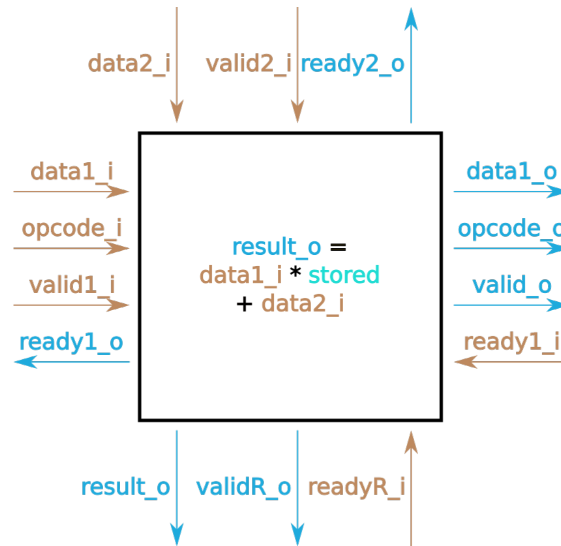

Figure 4.2: MAC operator sub-module interface and basic operation.

From the figure we can see that the MAC unit has four valid-ready hand-shake mechanisms. They implement the flow control that applies to every transaction issued from the MAC unit as a consumer and the MAC unit as a producer. We can observe the MAC unit behaves as a consumer to the left and upper part of the module, and as a producer to the lower and rightmost part of the module. Indeed, the MAC modules are built and connected forming a 2D grid as depicted in Figure 4.1.

Internally, MAC units have an accumulator, a multiplier, a register to latch the stored value into and some control logic to interpret the incoming opcode and decide whether to operate with its inputs or latch them in the stored register. Some processing of the internal opcode is also performed as described further on section 4.2. The data comming from the left will be then forwarded to the right side and the result produced by the module is sent downwards.

Internally, the MAC unit uses the FloPoCo number format and operators, being entirely reconfigurable to use FPGA-dependent DSP and different accumulator and multiplier designs as well as different precision or even mixed precision formats.

4.1.1. Internal MAC unit architecture

On Figure 4.3 a simplified version of the internal MAC module logic is presented. There we can see the complexity of the control logic. This control logic is needed due to the four-way synchronization of the MAC placement on the systolic array as seen on Figure 4.2.

The multiply and accumulate modules are taken straight out of the FloPoCo generated VHDL code and placed inside the Multiply Accumulate wrapper. This wrapper reads the opcode and the valid signals and decides whether to load the data coming from "Data1" into the "Store" register or into the multiplier.

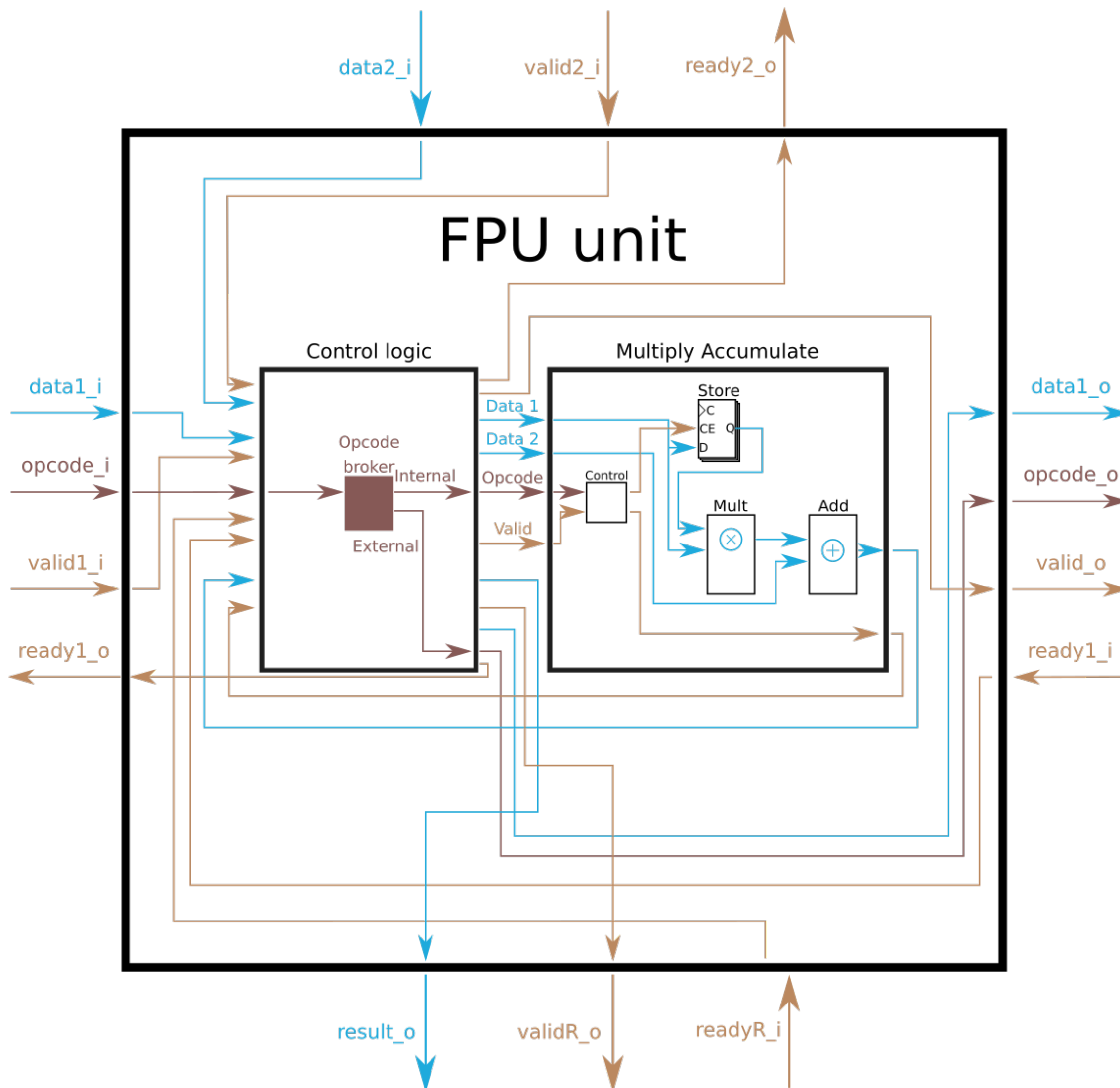

Figure 4.3: FPU module internal

This control logic inside the FPU also determines whether the result coming out of the "Add" operator is valid or not by counting the cycles that the Multiply-Accumulate pipeline takes and issuing a valid signal at the proper cycle.

This valid signal is processed by the FPU control logic and the add result latched to be dispatched into the lower module. The FPU control logic has several functions:

- Process the incoming opcode: This opcode has to be processed as we have three possible accepted opcodes for the FPU:
  - MAC: Tells the unit to perform a Multiply Accumulate with the stored data and the given "Data1" and "Data2".
  - Load: Tells the unit to load the data from "Data1" into the MAC "Store" register.
  - Forward Load: Tells the unit to forward the data and opcode coming from "Data1_i" and "opcode_i" to "Data1_o" and "opcode_o" for another unit to process. This opcode is used to bypass cells that should not store the value provided.
- Synchronize and latch the data coming from every side. IE: Both inputs and both outputs. This also means applying backpresssure when necessary.

4.1.2. Data loading process

As previously described our design uses two loading opcodes (Load and Forward Load) and one compute opcode. By using several opcodes, we reuse the horizontal channels between every FPU unit inside the Systolic Array for both matrix multiplication and loading.

This loading scheme is relatively simple. When a FPU unit receives a valid "Load" opcode it proceeds to load the data coming from "data1_i" inside the Multiply Accumulate latch and forwards to the right a non-op opcode. This non-op code is used to avoid the remaining units to perform any useless operation. On Figure 4.4 we can observe this loading process, there we name "Invalid" to the non-opcode.

If a unit receives a valid "Forward Load" opcode right after a load opcode it proceeds to send a "Load" opcode to the next unit with the received data. This case is the observed on Figure 4.5. This "Forward Load" to "Load" let's us chain FPUs and reuse the horizontal channels for load operations. The opcode attachment to each datum is performed by the Systolic Array control submodules shown on Figure 4.1.

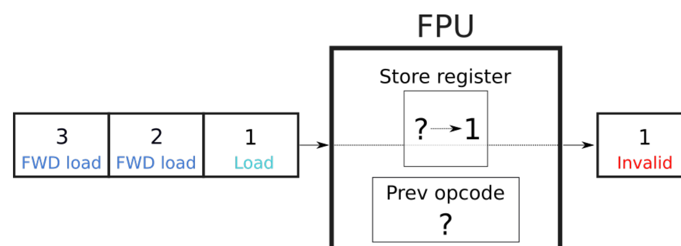

Figure 4.4: FPU loading mechanism. First cycle.

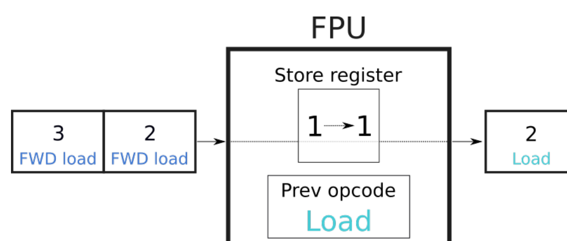

Figure 4.5: FPU loading mechanism. Second cycle.

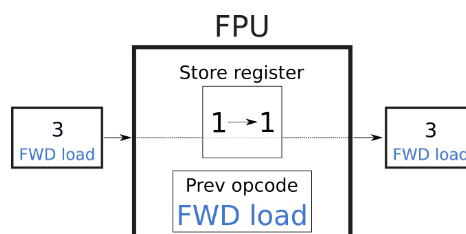

Figure 4.6: FPU loading mechanism. Third cycle.

On Figure 4.4, Figure 4.5 and Figure 4.6 all potential loading possibilities are shown. The transition between load and invalid opcode is represented on Figure 4.4. The conversion between load and forward load is represented on Figure 4.5. And the non opcode-modifying transition between a new forward load and a previous forward load is represented on Figure 4.6.

4.1.3. Flow Control Mechanism

The Floating Point units apply a flow control mechanism on every neighbour unit. For each one we use the valid-ready mechanism. In this mechanism, the data to be transmitted is synchronized by two signals: valid and ready. The valid-ready handshake is synchronous, and, as such, it must be checked at each cycle. We have four cases on the synchronization signals:

	Ready = 0	Ready = 1
Valid = 0	Nor the producer nor the consumer are ready, data on the attached dataline is not coherent.	The consumer is waiting for data from the producer.
Valid = 1	The producer data line contains valid data awaiting to be handshaked, backpressure is applied by the consumer.	Handshake occurs on both ends, both producer and consumer interpret this state as a correct transaction and update their states accordingly. If in the next edge this state is still on, a further transaction will happen.

Table 4.1: Valid-Ready control flow mechanism possible states.

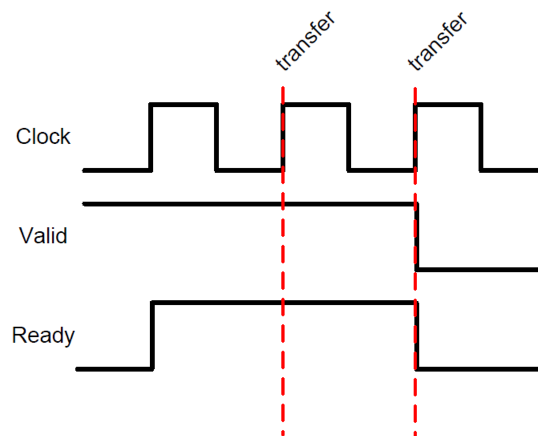

Figure 4.7: Valid-Ready data transfer timing diagram example.

Source: [23, Figure 3].

An example of a successful transaction as shown on the bottom right of Table 4.1 is shown on Figure 4.7. There we can observe that this mechanism is synchronized on the rising edge of the clock. If both the valid and ready signals are set to high on a given rising clock edge a transfer will occur on the attached data line. On the first rising clock edge of the figure we can observe that a transfer does not occur due to the ready signal being set to low on that specific cycle.

This mechanism has the advantage of relaxing the constraints on how the data has to be consumed from the components to which the accelerator is connected. It can be implemented with no need for registers or counters. This handshake mechanism is implemented on the FIFO interfaces and is further described on [23].

4.2 Our systolic array design

Our systolic array is composed by $N \times N$ MAC units interconnected without any additional logic. The absence of additional logic greatly promotes the scalability of the MAC unit.

To better understand the inner functioning of our systolic array multiplication unit, a running example will be shown on the next subsection. But first, some aspects of our implementation of the systolic array multiplication shall be clarified.

The systolic array multiplication shown on subsection 2.2.2 requires both matrices to be fed at the same time to the systolic array unit to perform the operation. That requires an outstanding amount of memory bandwidth, and thus, would imply a memory bottleneck in our approach. That's why a slight modification was performed on our systolic array inner working.

This slight modification implies that we only need one $n \times wordLength$ channel to feed data into our systolic array where n is the size of one matrix dimension. It consists on the addition of a load opcode to each MAC matrix cell, letting the systolic array store one matrix to be multiplied on. The loading mechanism has been described in subsection 4.1.2. This obviously means a slight increase on latency and a decrease on throughput, but, assuming the reuse of the loaded matrix, the memory bandwidth gains offset the slightly reduced throughput.

4.2.1. Overview example of the process of the proposed matrix multiplication unit

1	2
A	
3	4

x

1	2
B	
3	4

=

7	10
C	
15	22

Figure 4.8: An example of a matrix multiplication.

Here we have a simple example of a matrix multiplication where the leftmost matrix (A matrix) is multiplied by the middle matrix (B matrix) in order to produce the rightmost matrix (Result Matrix). This multiplication has already been performed on subsection 2.2.2, but due to performance requirements our unit performs it in a slightly different way.

In this example, the multiplication of the Figure 4.8 matrices will be performed in a similar way that our model processes it. For the sake of simplicity, this example will be laid out in steps. Each step is composed of several cycles and is considered a significant unit of work, where the state of the computation is significantly different from one step to another.

For the sake of understanding, we need to clarify two concepts:

- Loaded matrix: In our design the multiplicand matrix needs first to be loaded onto the systolic array to be multiplied. This will be referred as the B matrix on our $A \times B = C$ example.

- Multiplier matrix: This matrix shall be processed after the B matrix is loaded onto the systolic array internal memory. We can have N multiplier matrices multiplied by each loaded matrix. This will be referred as the A matrix from now on.

Now that the basic terms of a matrix multiplication are laid out, a trace of our systolic array multiplication will be shown.

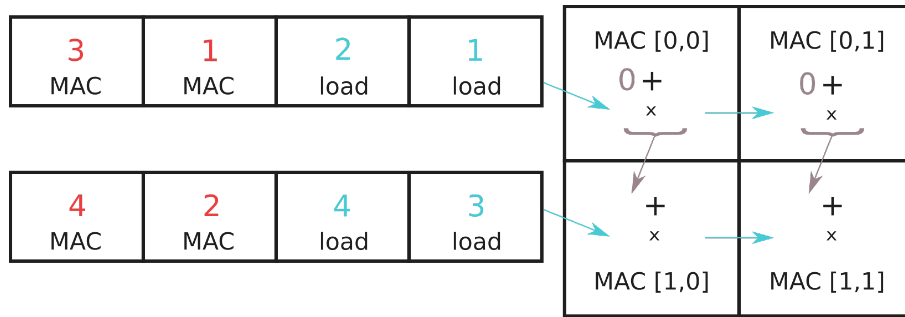

Figure 4.9: An example of a matrix multiplication. First step.

As shown in Figure 4.9, we first load onto the input queues the data to be processed. B matrix should be loaded on column-major order, while A matrices shall be loaded on row-major order. The load operation into the input queues can be done while we compute in the array. For the sake of description we will not take into account this overlap.

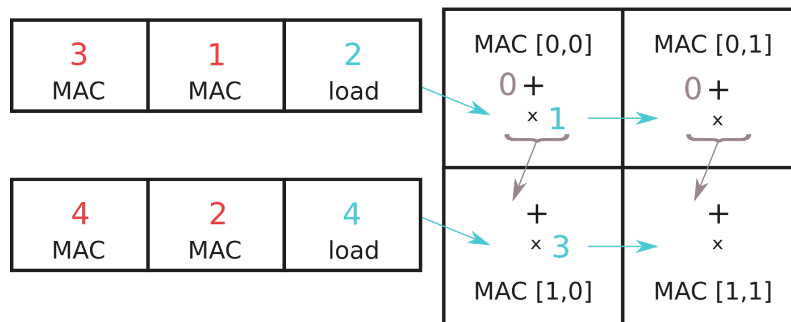

Figure 4.10: An example of a matrix multiplication. Second step.

Then the first batch of inputs is loaded, reading its operation code and performing it as specified. In Figure 4.10 we can observe that values 1 and 3 are loaded on MAC cells [0,0] and [1,0] specifically. In this design, the partial result going into the MAC operators on row zero is zero as they have no MAC units above.

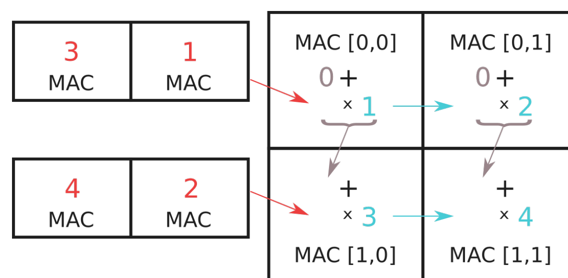

Figure 4.11: An example of a matrix multiplication. Third step.

Now, at the third step all the weights (B matrix) are loaded onto the systolic array. And the similitude between Figure 4.8 B matrix and Figure 4.11 is noticeable. When the B matrix is loaded, partial multiplications will be performed.

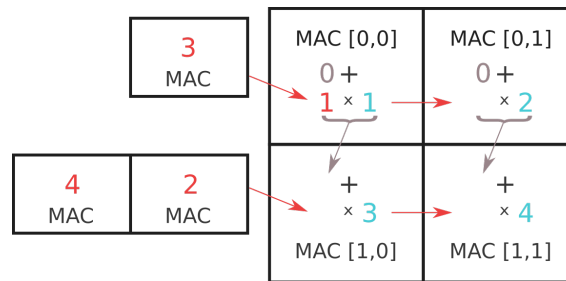

Figure 4.12: An example of a matrix multiplication. Fourth step.

As shown on Figure 4.12 we launch the first MAC operation on the first row but not the second row MAC operation. This is due to the staggered behaviour of a systolic multiplication. This staggering is due to the internal dependencies between MAC operations. One term of the MAC[1,0] sum is the result of the multiplication and accumulation of MAC[0,0]. Therefore the first MAC[0,0] operation shall be completed in order to launch the MAC[1,0] operation.

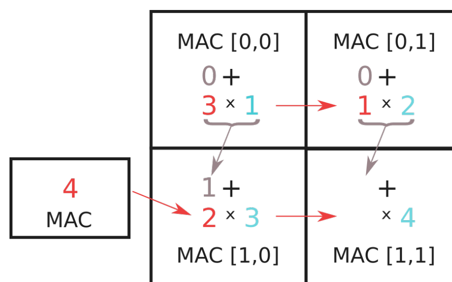

Figure 4.13: An example of a matrix multiplication. Fifth step.

Next, the MAC value is multiplied on cell [0,0] by the stored data and the result is passed onto [1,0] to be accumulated. Furthermore MAC value of [0,0] is passed onto [0,1] and [0,0] takes a new MAC value (Figure 4.13). This cache-less, uniform access time, data reuse is what really makes the Systolic Array architecture an specially efficient operation mechanism.

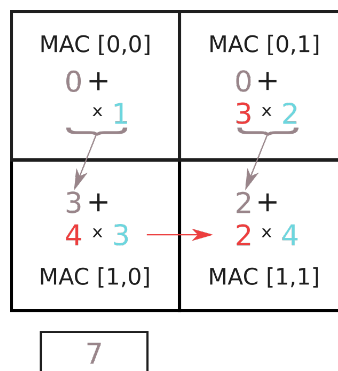

Figure 4.14: An example of a matrix multiplication. Sixth step.

On Figure 4.14 the first result of our matrix multiplication is shown. This result is $1 \times 1 + 2 \times 3$. If we refer to Figure 4.8 we can observe that this result corresponds to the data on the result matrix position [0,0]. This also happens to be the first consumed multiplicand data of our circuit. The previously mentioned data staggering can also be observed on the results.

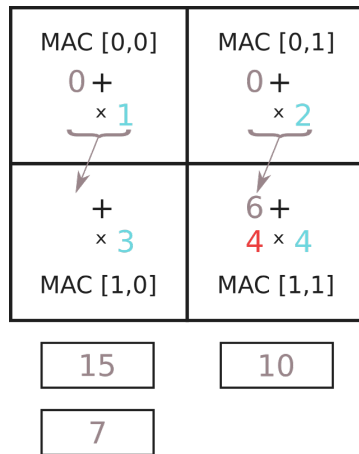

Figure 4.15: An example of a matrix multiplication. Seventh step.

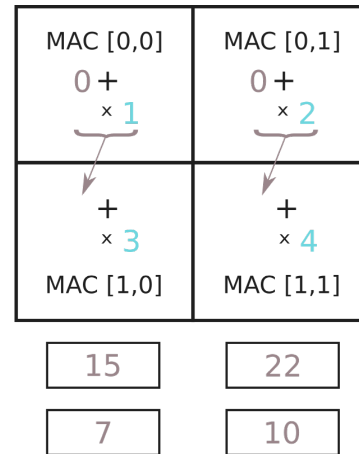

Figure 4.16: An example of a matrix multiplication. Eight step.

Lastly, in Figure 4.15 and Figure 4.16 the last two cycles are performed. This simple multiplication example shows us different traits of systolic array multiplication:

- Data reuse between operations is huge, being each element of the A matrix reused N times for an $N \times N$ matrix. In a conventional processor, the partial results should be stored on a register or on cache to be processed by the next operand, taking extra cycles and using extra power.
- Single matrix multiplications are really computationally expensive with this model. However, once you load a B matrix you can launch N A matrix multiplications against that loaded B matrix. The more back to back A matrices you launch against the loaded B matrix, the more the cost of filling and emptying the systolic array will be offset by the benefit of using it. This is really the typical use case in neural network inference settings, where weights are preloaded and then input data is inferred.

4.2.2. Systolic Array control submodules

The systolic array control submodules are the ones that help feed organized and properly formatted data into the systolic array, guaranteeing correct operation at the correct time. In the next subsections we describe several basic control components, decoupling buffers, FIFOs, and FloPoCo converters.

Coupling and decoupling buffers

The coupling and decoupling buffers serve a very important role on the bridging between the packets arriving to the module, and the row-independent data flowing into the systolic array.

The decoupling buffer splits the data coming from an AXI packet into the different needed data that go into the systolic array. It takes a 256-bit AXI packet and decomposes it into eight 32-bit data packets that flow into each row of the systolic array. These numbers are being used as an example, in the design the length of the systolic array is completely parameterizable.

Then each datum is synchronized independently to each FIFO queue for each row of the systolic array. If any row FIFO were to be full, the row synchronization would not

be produced and thus, backpressure would be applied. Indeed, the whole Systolic Array is interconnected with valid-ready synchronization and backpressure can be applied by any of the presented entities.

The control data is received into the Decoupling buffer and repeated and transmitted to each row. This is needed as all rows are working independently from each other in this particular design.

The coupling buffer works the same way as the decoupling buffer but it performs the opposite function. It receives individual data from each row, and, when all the rows have already fed data, it forms a packet and sends it into the AXI Writer control logic.

FIFO queues

FIFO queues, located between the decoupling buffer and the systolic array, are needed due to the staggered data consumption of the systolic array paradigm as described on subsection 4.2.1. These FIFO queues then, are very important for the correct working of the systolic array. Without them, the design would not work as only a transaction could be stored onto the systolic array.

The correct tuning of the length of these FIFOs is also very important, where the minimal length would not be equal for all. This is due to the staggering of consumption of data, where the upper FIFOs would need less length than the lower FIFOs. The minimal theoretical input FIFO length for a 8×8 matrix ranges from 0 at row 0 to 7 at row 7.

The output FIFO length works in a similar way than the input FIFO but with the lengths inverted. This means allocating a minimal theoretical input FIFO length of 7 for the leftmost matrix output and zero for the rightmost matrix output.

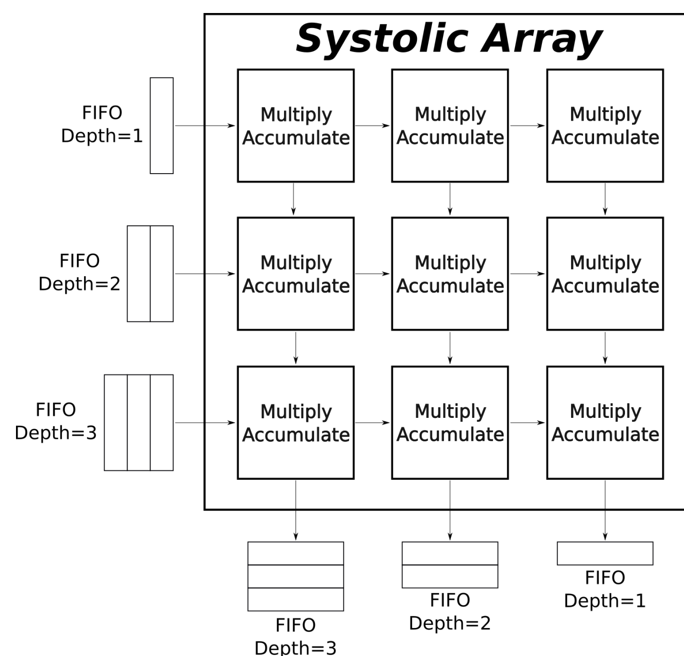

Figure 4.17: Variable FIFO queue depth representation.

This FIFO queue variable depth can be seen more graphically on Figure 4.17, where the top input queue is smaller than the bottom one and the rightmost output queue is smaller than the leftmost. This is due to the input staggering presented on Figure 4.12 and output staggering presented on Figure 4.15. In this example we used a slightly larger

than minimal FIFO queue depth as it gives us a small buffer and is more didactic than having a zero-depth queue as specified previously as minimal depth.

IEEE to FloPoCo and FloPoCo to IEEE

This modules are in charge of converting the standard IEEE single precision floating point format in the desired FloPoCo format, in this case the equivalent to IEEE single precision. They perform the transcoding between those two formats in a combinational way.

The format differences can be appreciated by comparing Figure 2.12 with Figure 2.3. But basically consist on eliminating all IEEE special number codifications and encoding some of those special values into the upper two bits. Some of those special values are the infinity values, zero and NaN.

Further explanation of the FloPoCo floating point number format can be seen on its corresponding section on subsection 2.3.2.

4.3 AXI

Communications from and to the Systolic Array are necessary as it follows the role of co-processor in a larger system. In order to perform this role, an standard communication protocol is strongly advised. Specifically, we chose the AXI interface (see Section 2.3.2) due to its tight coupling with the Xilinx ecosystem, and, in specific, with the Vitis environment. Also, the AXI protocol is the default protocol used and assumed by the embedded systems community.

AXI communication has been provided through several Xilinx modules that aim to integrate RTL designs into OpenCL callable kernels inside an FPGA runtime. The current module architecture supports several parameters through the AXI environment that define the inner workings of the Systolic Array:

- Matrix Size: Defines the size of the $N \times N$ matrix to be passed onto the implemented underlying hardware. Currently this number can only be lower or equal than the implemented matrix size, and lets us multiply matrices smaller or equal to $N \times N$ length.
- Number of matrices: Defines the number of A matrices of Matrix Size size to multiply against the loaded B matrix. Multiple back to back A matrix multiplications against a loaded B matrix is extremely encouraged due to the performance benefits of keeping the same matrix loaded.
- load b: Tells the systolic array if the first data to be passed through the AXI bus is a b matrix to be loaded onto the systolic array or are A matrices that should be multiplied by the currently loaded B matrix.
- AXI00_ptr0: AXI pointer of the memory address of the B matrix to load.
- AXI00_ptr1: AXI pointer of the memory address of the A matrix or matrices to multiply against B.
- AXI00_ptr2: AXI pointer of the memory address to store the C matrix.

Through these parameters passed as scalars to the module on ap_start (the module initializing signal) we control the number of transactions that need to occur and fully interface with the AXI protocol, and thus, the Vitis environment and the Xilinx Runtime.

CHAPTER 5
Performance Evaluation

In this chapter, we asses the performance of our systolic array module. For most of the analysis (frequency-related and resources-related) we use the whole system including the AXI interface and the systolic array modules (FIFOs, decoupling buffers, and FloPoCo units). For performance, however, we only analyze the systolic array subsystem. This allows us to decouple the efficiency of our design from the potential performance bottleneck of the memory attached to the systolic array, which is out of the scope of this project goal. Therefore, our performance results assume an ideal memory subsystem able to provide the required bandwidth. Notice also that the additional components in our system (FIFOs, decoupling buffers, FloPoCo units and AXI interface) would affect only on latency but not on throughput.

For the analysis related to clock speed and resources needed we selected the Alveo U200 platform due to it being available on our laboratory for testing and compatible with the Vitis openCL proof of concept.

5.1 Testing methodology

5.1.1. Simulation of the design on Vivado

In terms of the simulation of the design we used several tools. We heavily leveraged the Vivado simulation waveform analysis tool for module debugging. We also used the Vivado RTL analysis tool for checking our design in a simpler and human-comprehensible way.

The performance results shown on the next section are a mix of theoretical results obtained counting the cycles taken with the waveform analysis tool and simulated results obtained by dumping to a file the state of the simulation each cycle and post-processing that file with a simple python script to check for correctness and real-world performance.

All frequency results are obtained using Vivado Synthesis timing summary. This utility was also used to obtain the slack of our design at a given frequency and detect where the critical path is, and, if possible optimize it.

5.1.2. Simulation of the finished result on Vitis

The Vitis Unified Software Platform lets us launch RTL designs as OpenCL kernels using the AXI subsystem implemented in our work. Therefore, we can benchmark and verify the correctness of our design and the AXI subsystem in an easy manner and obtain preliminary results.

The Vitis simulation consists of the Host and Kernel codes. The Kernel code is our whole Systolic Array including the AXI wrapper. The Host code is a simple OpenCL code that loads up our kernel, creates two matrices "A" and "B", copies them into the FPGA memory, launches the FPGA kernel computation and retrieves the result to verify correctness and shows it on the screen.

The matrix sizes are 8×8 in order to speed up our hardware emulation compilation times and the results are shown on Figure 5.1. There we can observe the correctness of the results of the multiplication and, at the last line, the memory transactions that happened. Those transactions consist on two 8×8 matrices "A" and "B" transferred from host memory to FPGA memory (RD) and one result matrix transferred from FPGA memory to host memory (WR).

```
| 1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0|
| 1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0|
| 1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0|
| 1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0|
| 1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0|
| 1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0|
| 1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0|
| 1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0|
                    *
| 2.0  2.0  2.0  2.0  2.0  2.0  2.0  2.0|
| 2.0  2.0  2.0  2.0  2.0  2.0  2.0  2.0|
| 2.0  2.0  2.0  2.0  2.0  2.0  2.0  2.0|
| 2.0  2.0  2.0  2.0  2.0  2.0  2.0  2.0|
| 2.0  2.0  2.0  2.0  2.0  2.0  2.0  2.0|
| 2.0  2.0  2.0  2.0  2.0  2.0  2.0  2.0|
| 2.0  2.0  2.0  2.0  2.0  2.0  2.0  2.0|
| 2.0  2.0  2.0  2.0  2.0  2.0  2.0  2.0|
                    =
|16.0 16.0 16.0 16.0 16.0 16.0 16.0 16.0|
|16.0 16.0 16.0 16.0 16.0 16.0 16.0 16.0|
|16.0 16.0 16.0 16.0 16.0 16.0 16.0 16.0|
|16.0 16.0 16.0 16.0 16.0 16.0 16.0 16.0|
|16.0 16.0 16.0 16.0 16.0 16.0 16.0 16.0|
|16.0 16.0 16.0 16.0 16.0 16.0 16.0 16.0|
|16.0 16.0 16.0 16.0 16.0 16.0 16.0 16.0|
|16.0 16.0 16.0 16.0 16.0 16.0 16.0 16.0|
INFO: Test completed successfully.
XRT build version: 2.3.1301
INFO::[ Vitis-EM 22 ] [Time elapsed: 0 minute(s) 15 seconds, Emulation time: 0.0476112 ms]
Data transfer between kernel(s) and global memory(s)
SystollicArray_AXI_wrapper_1:m00_axi-DDR[1]          RD = 0.500 KB               WR = 0.250 KB
```

Figure 5.1: Results of the Vitis hardware emulation.

Figure 5.2: Vitis application timeline results.

Another perspective on this successful matrix multiplication is provided by running the Vitis analyzer tool on the hardware emulation run shown on Figure 5.1. The results of the application timeline analysis are shown on Figure 5.2. Those results show us the in-

credible inefficiency of small kernels launched in the Vitis environment. The entire simulation process (from kernel setup to kernel end) took 15.8 wall-clock seconds. Thankfully, we are only using the OpenCL kernel approach for AXI integration debugging so slow performance is not one of our main concerns.

Even though our full simulation took 15.8 seconds from kernel setup to finish, the actual kernel runtime was only 403 ms. This is still a very slow performance but helps us illustrate the time taken for runtime partial reconfiguration and kernel loading. Also, on the bottom of Figure 5.2 we can appreciate the host to FPGA memory transfers from both input matrices and the output matrix.

To summarize, the implementation of our design as an OpenCL kernel proved out to be successful but relatively slow. From this result we learned that larger kernels are strongly recommended to offset the time taken to setup the kernel environment on the Xilinx Runtime and multiple operations against the same kernel are recommended.

5.2 Results and discussion

We evaluate the design shown at Figure 4.1 with the MAC units using FloPoCo library operators tuned at a fixed frequency. First we provide results for performance and then we focus our attention on FPGA resource needs. Finally, we will provide an analysis of FloPoCo tuning effects on our design.

5.2.1. Performance

In terms of performance we analyze three cases: 1) scalability of our design in terms of size (ie: the performance benefit of implementing an 8×8 matrix instead of a 4×4 matrix); 2) performance of our design in terms of A multiplications performed against the same B matrix as described on subsection 4.2.1 and; 3) the frequency scaling of our design.

Frequency scaling analysis is not obvious. Ideally, performance should linearly increase as frequency increases, potentially reaching the maximum achievable frequency threshold (i.e: when the slack of our design is zero). However, performance will be affected by the FloPoCo unit, which allows us to achieve a particular frequency by redesigning its pipeline. This will impact our performance. In our study we take this into account and we analyze the maximum possible frequency for the Alveo U200 on each FloPoCo target frequency.

Indeed, the achieved performance will vary as the number of pipeline stages in FloPoCo will vary and not all of them will be efficiently used (unbalanced pipeline design is possible). Also, those pipeline stages are not efficiently used due to inherent data dependencies. Later, we further explain this problem, and at section 7.2 some possible solutions are discussed.

Scalability of the Systolic Array

For this analysis the selected target frequency and thus, pipeline depth, is set to 250 MHz. For the results, we show throughput measured as FLOPS/cycle instead of FLOPS/sec. So, we focus only on an ideal scalability analysis not considering the frequency effect, only our architectural design.

In Table 5.1 several performance figures are presented. The "Peak performance Flop/-Cycle" column represents the maximum possible performance of our Systolic Array when

Matrix Size	Peak performance Flop/Cycle	Worst case performance Flop/Cycle	Ideal Max Flop/Cycle
4x4	2.29	0.86	8
8x8	9.14	3.41	32
16x16	36.57	13.56	128
32x32	146.29	54.07	512

Table 5.1: Size and performance comparison of the proposed design.

all internal MAC units are working at the same time. This column's data is obtained by obtaining the maximum performance of a MAC unit theoretically by counting cycles (Each MAC unit takes seven cycles to process an input for this pipelining depth) and scaling it to the size of the Systolic Array.

Conversely to Peak performance, the "Worst case performance" column represents the worst possible performance of this Systolic Array taking into account infinite memory bandwith. This means that backpressure is not applied and this case just considers the loading of a "B" matrix and multiplication of a single "A" matrix against it. This contrasts with "Peak performance" where infinite "A" matrices are launched against a single "B" matrix. The results of this column where obtained by experimental testing. This experimental testing consists on a System Verilog testbench that writes selected signals into disk to be further post-processed by a python script.

Finally, the "Ideal Max Flop/Cycle" column consists on obtaining the actual maximum throughput of the operators taking into account only the FloPoCo units. The difference between the "Ideal Max" and "Peak performance" columns is the overhead that our synchronization (flow control) mechanisms is introducing.

With the results on Table 5.1 we can arrive to the following conclusions for a $N \times N$ array:

- Peak performance scales linearly with MAC units and quadratically with N. This means that the 8×8 matrix size has a 4x performance improvement over the 4×4 matrix size. This is due to 4x more MAC units available.
- Worst case performance does not scale linearly with matrix size. The larger the Systolic Array the worse the worst case performance gets. This means that the 8x8 matrix multiplication does not have four times more performance than the 4x4 on its worst case, instead it has 3.94 times more performance. This is due to a larger matrix taking longer to fill with loaded data and to empty once the MAC operation is performed.
- Ideal max Flop/Cycle scales linearly with MAC units, as peak performance does.

Performance impact when injecting consecutive A matrix multiplications

In this section, we analyze the performance impact of performing more **A** matrix multiplications against a loaded matrix (**B**).

The results shown on Figure 5.3 represent the expected logarithmic growth of throughput with an increasing number of A matrices launched against a preloaded B matrix. This growth is due to the offsetting of the computational time taken when performing the loading, filling and emptying the Systolic Array.

The "initial and interpolated B load" line as presented on Figure 5.3 present an scenario where n matrices are multiplied and two matrices are loaded, one at the beginning

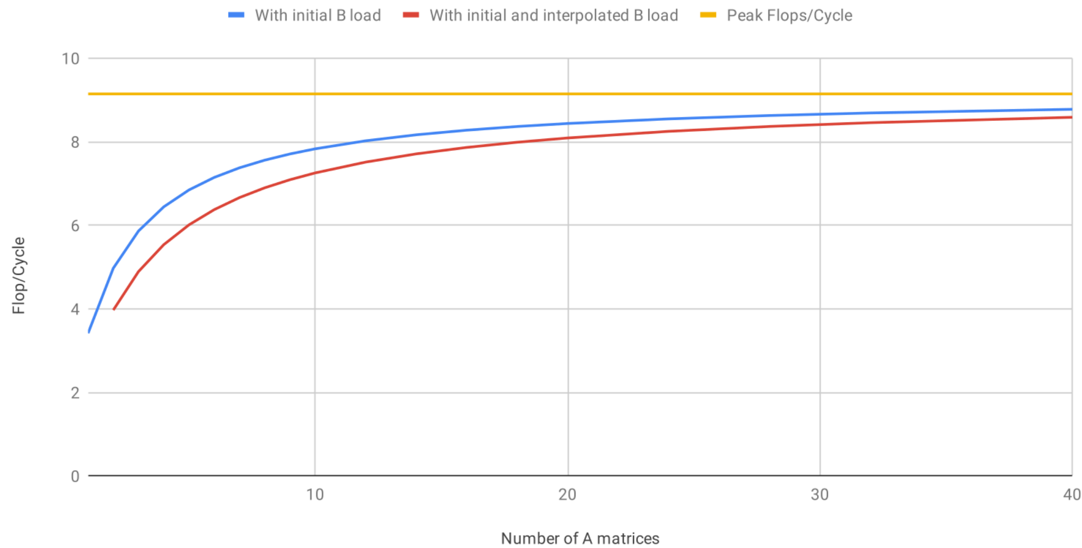

Figure 5.3: Performance impact of different A multiplications against a B matrix loaded.

of the computation and one right at the n/2 matrix. This line is here to present a common case that has yet to be discussed. This case is the case where multiple matrix operations are queued to be processed by our Systolic Array. This line represents the performance hit of multiplying two n/2 batches of matrices in respect to the multiplication of only one batch of n matrices.

This "initial and interpolated B load" line helps us illustrate the importance of keeping the pipeline full and shows the efficiency loss of not doing so. In specific, the main thing that this line highlights is that the performance of multiplying two batches of n/2 matrices is greater that the performance of multiplying a single n/2 size batch. This can be clearly seen by comparing the achieved FLOPS/cycle of the blue line at 10 matrices with the performance of the red line at 20 matrices. There we can clearly see that by keeping the pipeline full we obtain some of the performance back by pipelining the load of the second matrix while the first matrix multiplication is still being processed.

5.2.2. Resources analysis

In terms of resources we focus our attention on the resources needed by the FPGA implementation, which is mainly LUTs, Flip Flops, Carry8 and DSP48E2. For this analysis we target the full AXI-Integrated design with different matrix sizes and synthesized for the Alveo U200 board.

Being Vitis compatible is extremely important as it lets us design, validate and test our design seamlessly with its powerful environment. This means that all the PCI express logic and memory transfers are not dealt directly by us on the RTL, instead they are programmed on OpenCL and interfaced with the AXI RTL design through the Xilinx Runtime installed on the Alveo U200 FPGA.

As seen on Table 5.2 and more graphically on Figure 5.4 the resources needed for the design scale approximately linearly with the number of MAC units and quadratically

Matrix size	LUTs	Flip Flops	Carry8	DSP48E2
8x8	27971	24117	1676	133
16x16	109481	88038	6492	517
32x32	430691	339289	25724	2053

Table 5.2: Resource usage of Alveo U200 according to size of the design.

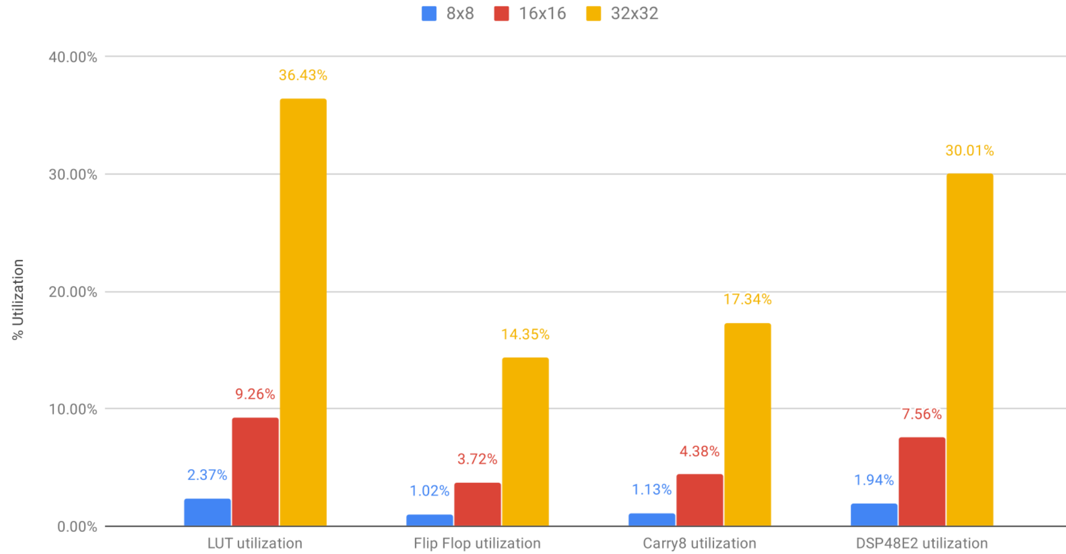

Figure 5.4: Percentage of utilization of Alveo U200 resources.

with the size of the Systolic Array. From 8×8 to 16×16 configurations there is four times more MAC units and the design is 3.91 times larger on LUT usage.

This observed non-perfect linear scaling is due to the asymmetry of the input and output FIFO queues and the constant AXI size for all designs. A deeper study into the resources needed for the FIFO queues reveals that the theoretical number of resources needed for a systolic array of size $N \times N$ is:

$$2 * \sum_{i=1}^{i=N} i = \frac{N(N+1)}{2} = N^2 + N$$

While the size of the systolic array scales N^2 and the AXI control logic barely varies on size.

As theoretically deduced and departing from those formulas we obtained the trend of resource growth depending on systolic array growth. This trend turned out to be an almost perfect fit (R=0.994) for the Systolic Array submodule and a perfect fit for the FIFO submodule as shown on Figure 5.5. The non-perfect fit for the Systolic Array could be due to several optimizations performed on synthesis in order to improve target frequency, power consumption or size, but the variation is so minor that further exploring is not deemed needed.

With these results we can observe that the main resource demanding component is the systolic array module, and not the the FIFO queues. As can be seen comparing the

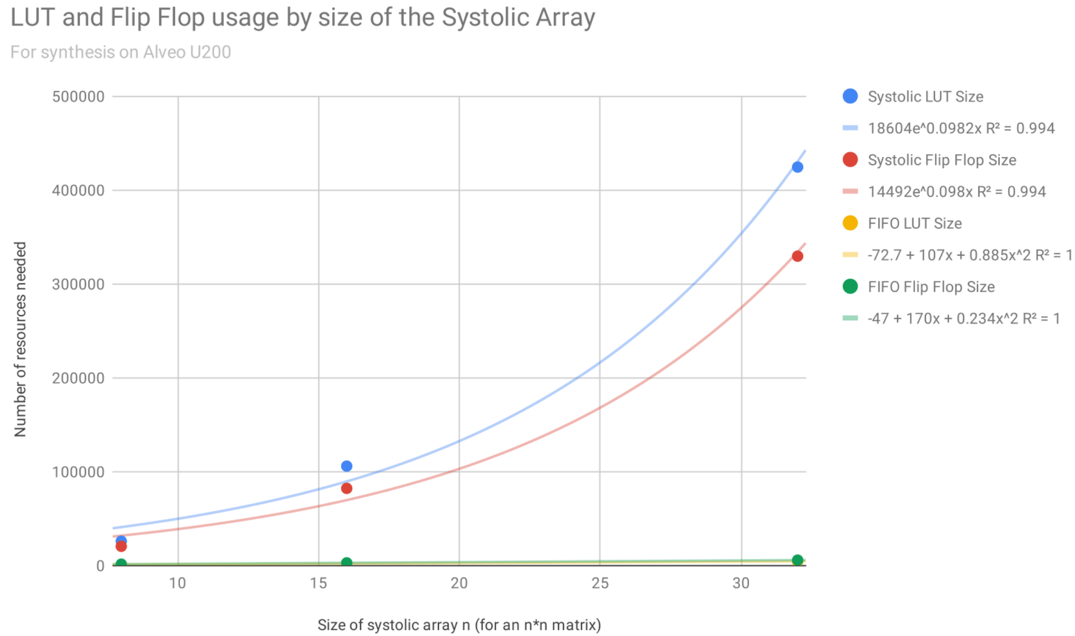

Figure 5.5: Resource usage of Systolic array module and FIFO queue subsystem depending on size of the Systolic Array.

Size of Systolic Array	LUTs used by Systolic Array	Flip Flops used by Systolic Array
8x8	92.36%	84.40%
16x16	96.61%	93.23%
32x32	98.52%	97.09%

Table 5.3: Percentage of resources taken by the Systolic Array part of the design.

growth trend formulas on Figure 5.5 the size of the systolic array scales with a greater quadratic number than that of the FIFO. So, the larger the size of the systolic array the higher the percentage of total size the Systolic Array will take in comparison with the control logic.

This previous statement can be checked with the results provided on Table 5.1. The results shown here prove that the part that performs the computations on our system is taking significantly more space than the control logic needed to run it, the greater the percent shown on Table 5.1, the more space efficient our system is.

5.2.3. FloPoCo tuning and its performance impact

In this section, the practical use of the FloPoCo configurable submodules is discussed, including our approach and implemented optimizations.

The FloPoCo utility is used in this project in two components:

- The AXI control subsystem: Here, the FloPoCo operators are used to calculate how many transactions should be passed onto the Xilinx AXI interface for each request. We used FloPoCo here because it is DSP-less and can meet all our needed potential frequency requirements, while DSPs cannot. The implemented FloPoCo operators in the AXI subsystem are integer squarers (x^2) and multipliers.

- The MAC units: In this module, FloPoCo operators are used to perform the core calculations of this work. Here a floating point multiplier and adder are implemented.

The FloPoCo implementation is achieved by providing to the FloPoCo module two parameters: the target FPGA family and the target frequency. In this sense, we had the additional problem of our FPGA family (Alveo) not being supported by FloPoCo as it is too new. To overcome this limitation we have chosen the most modern FPGA supported by FloPoCo and tested several target frequencies and implementations to determine if the pipelined realized by FloPoCo is optimal.

With that workflow in mind we proceeded to create a baseline FloPoCo target frequency study to maximise FLOPs. This consisted on testing different target frequencies for the FLoPoCo operators and their achieved throughput. Surprisingly, the achieved results do not scale well with frequency due to our design not fully exploiting the whole potential of the built-in operator pipelining due to inherent data dependencies between the different MAC operators. Further solutions for using that pipelining are presented on chapter 7.

FloPoCo target frequency	Synthesis achieved clock frequency	FLOP/S
400 MHz	348 MHz	1.82E+09
350 MHz	278 MHz	1.74E+09
300 MHz	227 MHz	1.78E+09
250 MHz	233 MHz	2.08E+09
200 MHz	160 MHz	1.43E+09

Table 5.4: FloPoCo initial approach synthesis and performance results.

According to the results presented on Table 5.4 the best target frequency is 250 MHz FloPoCo target, but, upon further inspection, an anomaly can be found between the target 250 MHz and 300 MHz. This anomaly consists on a higher FLoPoCo target frequency yielding lower synthesis results. Upon further inspection on the critical path of different target frequencies we discovered that the FloPoCo integer operators were behaving differently than the floating point ones, and were bringing the target frequency down.

Since the integer operators were only used once per AXI call and the floating point ones were used widely through our design, we decided to take the integer FloPoCo operators out of the critical path by increasing their FloPoCo target frequency and thus, their pipelining. With those optimizations we obtained the following results.

FloPoCo target frequency	Synthesis achieved clock frequency	FLOP/S
400 MHz	348 MHz	1.82E+09
350 MHz	307 MHz	1.92E+09
300 MHz	299 MHz	2.67E+09
250 MHz	299 MHz	2.67E+09
200 MHz	159 MHz	1.66E+09
150 MHz	154 MHz	1.93E+09

Table 5.5: FloPoCo with tuned target frequencies synthesis and performance results.

As shown on Figure 5.6 that compares the frequencies obtained on Table 5.4 and Table 5.5 the FloPoCo integer generated pipeline is not well balanced with the Floating Point pipeline, leaving for the user the job of tuning the different generated module frequencies to better leverage the maximum performance of the FPGA fabric.

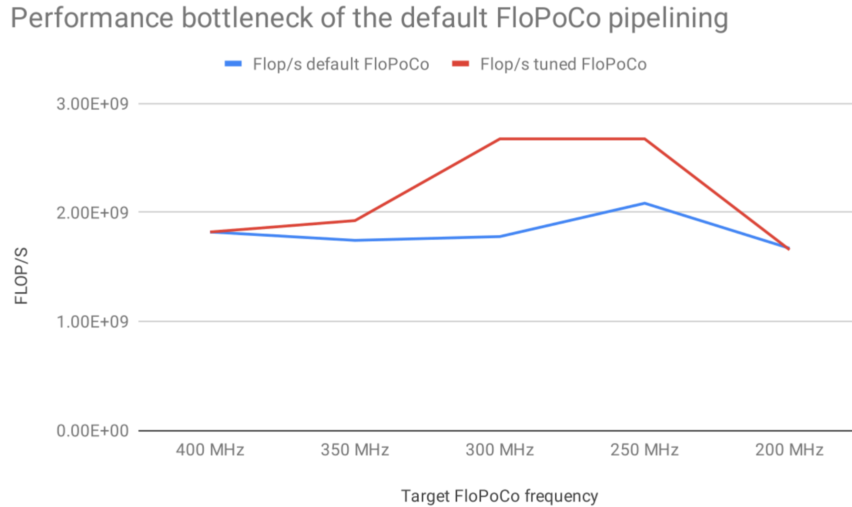

Figure 5.6: Performance bottleneck of the default FloPoCo pipelining.

This lack of consistency between target and obtained frequencies could also be the result of not selecting our synthesis FPGA family from the FloPoCo provided ones. But, as FloPoCo does not give support for our specific FPGA family this is our only choice if we want to use this tool. Further insights on this topic would be needed to determine if this is a localized issue for our target FPGA family or a recurrent FloPoCo issue.

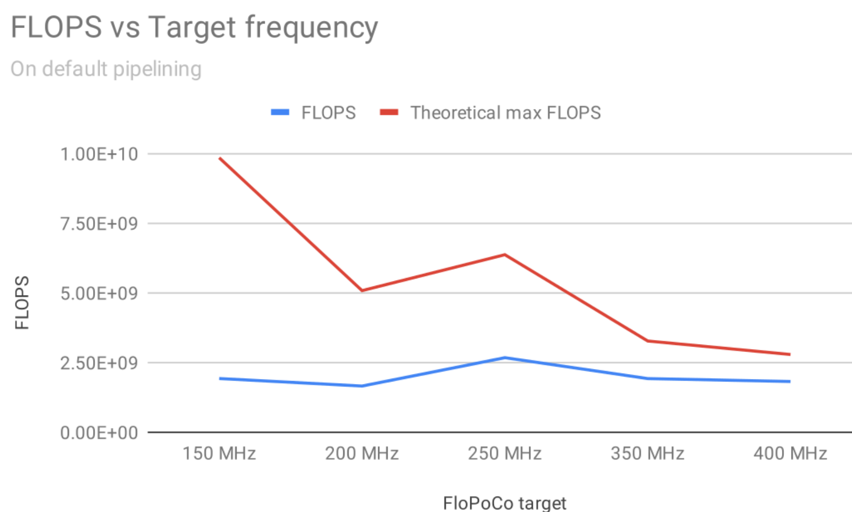

Figure 5.7: FLOPS per target frequency for an 8×8 Systolic Array.

In terms of ideal best performance of our design we calculated it by taking the latency of our MAC unit and obtaining the FLOP/cycle of our design by multiplying the FLOP/cycle of our MAC by the number of units on it. Once we obtain the maximum non-pipelined MAC throughput we must multiply it by our achieved synthesis frequency. This yields us the ideal max throughput of our design.

$$IdealMaxFLOPS = \frac{(MatrixSize)^2}{(MultiplierLatency + AdderLatency)} \times SynthesisFrequency$$

With this data we obtain the results displayed on Figure 5.7 that shows us that, as expected, the maximum theoretical throughput would be for the lower frequency and

minimal pipelining, as this pipelining is not used. But, the highest practical throughput is at 250 MHz target. This is due to the MAC pipelining overhead being offset by the internal synchronization mechanism speedup. This internal synchronization mechanism is what is causing us the difference between our projected maximum throughput and the obtained one.

It must be taken into account that a design that fully utilizes the internal MAC pipelining would be much faster as pipelining scales greatly with frequency and throughput is not affected by pipelining latency.

Another good insight into the current pipelining usage of our design could be gained from the graph shown on Figure 5.8, where we see the performance disadvantage of leveraging FloPoCo pipeline only as frequency scaling. Further work can be done to fully utilize the pipeline and improve significantly arithmetic performance. Although dramatically improving performance in this aspect would yield bottlenecks on the AXI subsystem.

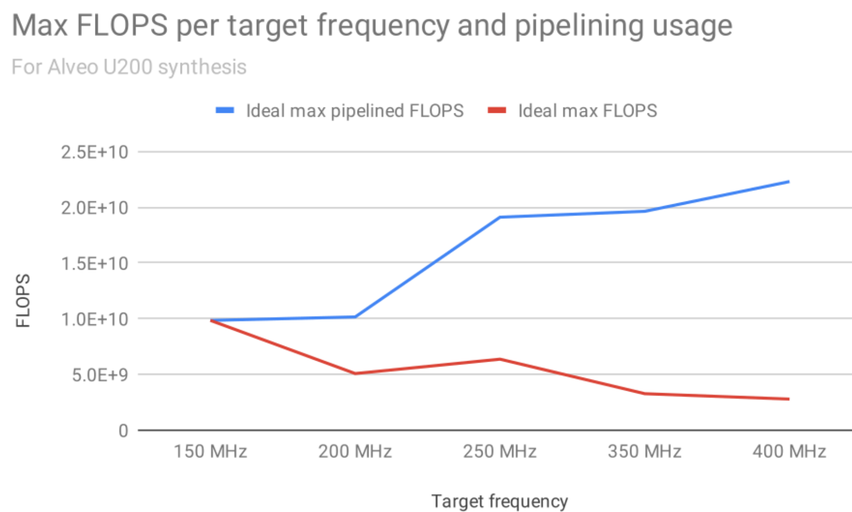

Figure 5.8: Ideal maximum FLOPS per target frequency and pipelining usage for an 8x8 Systolic Array.

5.3 Conclusion

As a general conclusion in our performance analysis, we can claim that the performance of our Systolic Array is a valid proof of concept for its systolic array organization and strategy and the design scales in resources and clock frequency. The current "naive" approach to the problem ended up providing simplicity and easeness of its design. The design can be further iterated to significantly increase its throughput as shown on the ideal maximum performance lines on the graphs shown on this chapter.

Further inspection on the ways that this performance can be increased is presented on chapter 7 (Future work).

CHAPTER 6
Conclusions

In this project we target the domain of matrix multiplication with a Systolic Array architecture for the purpose of accelerating machine learning inference workloads. We try to achieve this with a FPGA target for testing and with possible ASIC implementation in mind. Every design decision of this project was based on those principles and we strive to achieve decent performance out of our matrix multiplier.

With the previous objectives in mind we have designed, validated and evaluated a Systolic Array design that aims to be a matrix multiplication coprocessor. Offloading those incredibly expensive operations from the processor into our module to accelerate mixed general purpose and neural network workloads. We achieved this goal by using the industry standard AXI bus to receive requests from the main processor and to gather and write data into memory.

The resulting design is an highly configurable and flexible matrix multiplication coprocessor. It is easily configurable in precision, frequency and size and offers good scalability through an clear design which simplifies its use and understanding. It is completely FPGA independent as it does not bind its implementation to any FPGA resource (mainly DSPs), thus being completely RTL-based. Its use is made compatible as it uses the AXI interface, widely used and adopted.

This all led us to implement a design that achieves a decent 2.08 Gigaflops for an 8×8 matrix. This is a good starting point for performance and has the potential to scale up to 7.3 Gigaflops with the proposed optimizations in the Future Work chapter (chapter 7). This means that, even though we have yet to extract all the performance that is possible with the FloPoCo operator design, we have built a strong, scalable foundation to further improve and develop an efficient matrix multiplier. This will be developed in the framework of the H2020 SELENE project.

Even though the current design can be significantly improved, we managed to fulfil all the objectives that where initially proposed on section 1.2. It ended up being a DSP-less Open Source matrix multiplier that is accessed by the AMBA AXI standard bus.

Although this work ended up performing as expected and managed to achieve every proposed objective, we encountered a few struggles in the design process that required more time to invest.

The main problem that arose was a design problem. Initially, we wanted to implement a HLS kernel to feed our RTL kernel. We tried to perform this connection using a new Xilinx technology called Kernel to Kernel streaming, but, this technology ended up being in a transitional stage and causing us a lot of unforeseen problems. All Xilinx examples using this technology used the old version that was currently supported by our Alveo runtime, but, the Xilinx Vitis platform could not generate a kernel with the old version

of the kernel to kernel streaming platform and the Xilinx runtime did not support yet the new version of the streaming platform. So finally we abandoned that path and used the current path of one monolithic kernel.

This last problem taught me the value of carefully investigating the current state of a technology before using it and to never trust that a technology will work just because a big manufacturer is endorsing it. It definitely was a lesson on being overconfident on the work you are going to perform and the roadmap selected.

6.1 Connection between the Computer Science degree and the presented work

In terms of the connection between the computer science degree and the presented work, it is strong and has many sources. This degree helped me to structure my mind in a logic and more parallel way, this is not learned in any subject in specific, it is a mix of lessons learned in many many subjects. That parallel program oriented mind altogether with the proper code commenting skills learned on many subjects where the soft skills needed on this work.

But, in specific there are three subjects that are at the foundation of this work. The main one is Digital Design, where I learned the basics of RTL programming as well as the verification and implementation workflow with Vivado that I extensively used in this work.

Furthermore, in the Computer Architecture and Engineering subject I learned the fundamentals and culprits of processor design, understanding clearly the memory bottleneck that modern computation faces and thoroughly understanding the inner workings of the Google TPU v1. Finally, in Advanced Architectures I learned the importance of good flow control, back-pressure and in some ways of efficiently connecting components inside a design.

CHAPTER 7
Future work

The proposed design can be significantly improved on several directions that will be drafted on this chapter. Hereafter we present optimizations that range from improving the inefficient pipelining use shown on subsection 5.2.3 to improvements on the AXI-Systolic array interconnection and AXI bandwidth.

7.1 Improvements on current design

In terms of improving the current design there are a wide variety of options that present themselves, from experimenting with new arithmetic libraries to improving the internal module handshaking that is causing us a high number of delay cycles.

The more obvious path of improvement without any major change of our design is the improvement of current handshaking between the elements shown on Figure 4.1. In specific the most complicated and currently inefficient handshaking is the valid-ready subsystem inside each Multiply Accumulate submodule. This specific submodule ended up being the hardest one to implement of the whole work due to its 4-way synchronization (two inputs and two outputs) and currently does not support concurrent processing of data as to simplify the already complicated process of synchronization.

For further improvements on the MAC unit pipelining a complete rework of this submodule is needed. This is mainly due to its current simple design but slightly inefficient implementation.

7.2 Pipelining usage improvement

In order to increase the utilization of the MAC units data pipelining is needed. Currently our system uses the Adder and Multiplier pipelines just as frequency scaling, that's why our throughput does not scale well with pipelining stages Figure 5.7.

The described phenomenon is due to intrinsic data dependencies and thus, is impossible to solve for the current design, but several workarounds are possible.

The studied workaround will be presented with the matrices shown on Figure 7.1. Here we assume we have an internal systolic array size of four. Where previously we would load the B_1 matrix and launch the A_1 and A_3 matrices against it to create the C_1 result we now would interlace the A_1 and A_3 multiplications so they could be performed inside the Systolic Array on a pipelined fashion, effectively duplicating the throughput but requiring twice the memory bandwidth.

$$
\begin{array}{|cc|cc|}
\hline
1 & 2 & 3 & 4 \\
\multicolumn{2}{|c|}{A_1} & \multicolumn{2}{c|}{A_2} \\
5 & 6 & 7 & 8 \\
\hline
9 & 10 & 11 & 12 \\
\multicolumn{2}{|c|}{A_3} & \multicolumn{2}{c|}{A_4} \\
13 & 14 & 15 & 16 \\
\hline
\end{array}
\times
\begin{array}{|cc|cc|}
\hline
1 & 2 & 3 & 4 \\
\multicolumn{2}{|c|}{B_1} & \multicolumn{2}{c|}{B_2} \\
5 & 6 & 7 & 8 \\
\hline
9 & 10 & 11 & 12 \\
\multicolumn{2}{|c|}{B_3} & \multicolumn{2}{c|}{B_4} \\
13 & 14 & 15 & 16 \\
\hline
\end{array}
=
\begin{array}{|cc|cc|}
\hline
90 & 100 & 110 & 120 \\
\multicolumn{2}{|c|}{C_1} & \multicolumn{2}{c|}{C_2} \\
202 & 228 & 254 & 280 \\
\hline
314 & 356 & 398 & 440 \\
\multicolumn{2}{|c|}{C_3} & \multicolumn{2}{c|}{C_4} \\
426 & 484 & 542 & 600 \\
\hline
\end{array}
$$

Figure 7.1: Sub-matrix multiplication example for MAC unit pipelining.

This technique in conjunction with larger matrix sizes to offset the matrix loading overhead presented on Figure 5.3 would result in much improved numbers similar to those presented on Ideal max FLOPS in Figure 5.8.

This would greatly improve the throughput and efficiency of our operator but would require more complex synchronization and interpolation mechanisms and thus create a slightly larger control logic. Though this control logic size penalty would be greatly offset by the dramatic increase on throughput.

7.3 A matrix caching mechanism for partial multiplications

The previous proposed design and current design have a significant drawback in terms of memory movements for matrices larger than the Systolic Array grid. This drawback is easily appreciable with the example shown in Figure 7.1. It consists on the inherent data flow and reuse of already loaded submatrices.

For example, in order to obtain the "C_1" submatrix shown on Figure 7.1 in the most efficient manner we need to compute first the partial "C_1" and "C_3" submatrices with "A_1" and "A_3" multiplication into "B_1". But then we should load "B_3" into the weights and then launch the "A_2" and "A_4" matrices into it. Then we need to pull the previous partial result from memory and add the new result, storing it into memory again. This uses up our memory bandwidth needlessly and could be solved with a caching system inside the module.

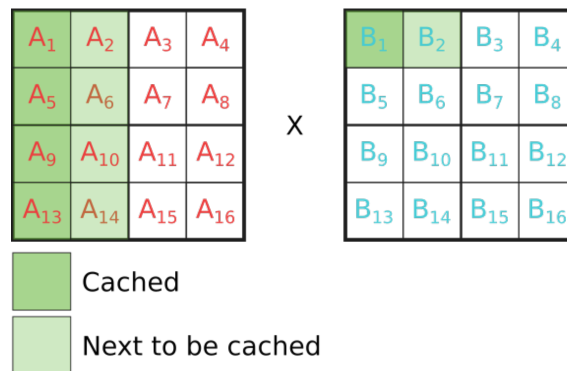

Figure 7.2: Rudimentary proposed Systolic Array caching mechanism.

This caching system could be implemented in several ways, but the simpler one seems to be to put a limit on the chunk size (Number of divisions of a matrix dependant on our systolic array size) of the submatrix multiplication that we can perform so we can cache an entire column of the "A" matrix and launch it against a "B" row, obtaining all the partial results of the desired "C" row as shown on Figure 7.2. This caching would be mapped on a non-cacheable memory segment, thus, guaranteeing the coherency with the rest of the memory hierarchy of the potential SoC.

This would end up saving us a great deal of "A" reloads from memory and thus freeing up bandwidth for other users of the AXI bus or to process all the resulting partial matrix multiplications yet to be added up. This method would use up $n \times wordSize$ bits of memory and would allow us saving $N \times N \times (N-1)$ transactions for an $N \times N$ matrix. So for the currently implemented method we would load up the Figure 7.2 "A" matrix four times and for this caching method we would load up the "A" matrix just once. We achieve a 250% decrease in AXI bandwidth usage for loading up "A" and "B" matrices in this example. The larger the matrices, the greater the size of the cache needed to cache up one column of the A matrix the larger the performance difference between both methods.

In general terms the reduction in bandwidth is equal to $\frac{N^3+N^2}{2\times N^2}$ for a needed cache size of $N \times wordSize$.

It needs to be noted that this caching strategy is simplistic and could be further improved adding more complexity to the system, but that's out of the scope of this work.

7.4 Variable precision arithmetic

As indicated on section 1.2 the design is compatible with different precision operators and mixed precision operators. The compatibility is tested to be working but not yet benchmarked due to the time constraints of this work. This variable precision arithmetic could result on smaller designs and improved throughput and latency of the operations. Also, power consumption reductions are expected as a result of a smaller design.

If implemented with the previous proposed changes, lower precision or mixed precision would allow us to achieve an smaller design or cram a couple more Systolic Array rows into the same size design.

Bibliography

[1] David A. Patterson and John L. Hennessy. *Computer organization and design: the Hardware/Software Interface*. Elsevier Science, 2011.

[2] M. Clark. "A new x86 core architecture for the next generation of computing". In: *2016 IEEE Hot Chips 28 Symposium (HCS)*. Aug. 2016, pp. 1–19. DOI: 10.1109/HOTCHIPS.2016.7936224.

[3] R. M. Tomasulo. "An Efficient Algorithm for Exploiting Multiple Arithmetic Units". In: *IBM Journal of Research and Development* 11.1 (Jan. 1967), pp. 25–33. ISSN: 0018-8646. DOI: 10.1147/rd.111.0025.

[4] G. E. Moore. "Cramming more components onto integrated circuits, Reprinted from Electronics, volume 38, number 8, April 19, 1965, pp.114 ff." In: *IEEE Solid-State Circuits Society Newsletter* 11.3 (Sept. 2006), pp. 33–35. ISSN: 1098-4232. DOI: 10.1109/N-SSC.2006.4785860.

[5] "IEEE Standard for Floating-Point Arithmetic". In: *IEEE Std 754-2019 (Revision of IEEE 754-2008)* (July 2019), pp. 1–84. ISSN: null. DOI: 10.1109/IEEESTD.2019.8766229.

[6] Google. *Cloud TPU System Architecture*. URL: https://cloud.google.com/tpu/docs/system-architecture (visited on 2020-03-11).

[7] Norman P. Jouppi, Cliff Young, Nishant Patil, et al. "In-Datacenter Performance Analysis of a Tensor Processing Unit". In: *Proceedings of the 44th Annual International Symposium on Computer Architecture*. ISCA '17. Toronto, ON, Canada: Association for Computing Machinery, 2017, pp. 1–12. ISBN: 9781450348928. DOI: 10.1145/3079856.3080246. URL: https://doi.org/10.1145/3079856.3080246.

[8] Emmett Kilgariff, Henry Moreton, Nick Stam, et al. *NVIDIA Turing Architecture In-Depth*. URL: https://devblogs.nvidia.com/nvidia-turing-architecture-in-depth/ (visited on 2020-05-11).

[9] Google. *An in-depth look at Google's first Tensor Processing Unit (TPU)*. 2017. URL: https://cloud.google.com/blog/products/gcp/an-in-depth-look-at-googles-first-tensor-processing-unit-tpu (visited on 2020-03-10).

[10] Moran Shkolnik, Brian Chmiel, Ron Banner, et al. "Robust Quantization: One Model to Rule Them All". In: *ArXiv* abs/2002.07686 (2020).

[11] *Vitis Unified Software Platform Documentation. Application Acceleration Development*. UG1393. Xilinx. Nov. 2019.

[12] Florent de Dinechin and Bogdan Pasca. "Designing Custom Arithmetic Data Paths with FloPoCo". In: *IEEE Design & Test of Computers* 28.4 (July 2011), pp. 18–27.

[13] *AMBA AXI and ACE Protocol Specification*. ARM IHI 0022E (ID022613). ARM. Feb. 2013.

[14] *The OpenCL™ Specification*. Version V2.2-11. Khronos® OpenCL Working Group. July 2019.

[15] *Xilinx Runtime (XRT) Architecture*. URL: https://xilinx.github.io/XRT/2019.2/html/index.html (visited on 2020-03-10).

[16] *Alveo Product Selection Guide*. URL: https://www.xilinx.com/support/documentation/selection-guides/alveo-product-selection-guide.pdf (visited on 2020-05-26).

[17] NVIDIA. *NVIDIA TESLA V100 GPU ARCHITECTURE*. Tech. rep. WP-08608-001_v1.1. NVIDIA, 2017.

[18] Coral. *System-on-Module datasheet*. URL: https://coral.ai/docs/som/datasheet/ (visited on 2020-03-25).

[19] Zhe Jia, Blake Tillman, Marco Maggioni, et al. *Dissecting the Graphcore IPU Architecture via Microbenchmarking*. Tech. rep. 1912.03413v1. Dec. 2019, p. 91. URL: https://www.graphcore.ai/hubfs/assets/pdf/Citadel%20Securities%20Technical%20Report%20-%20Dissecting%20the%20Graphcore%20IPU%20Architecture%20via%20Microbenchmarking%20Dec%202019.pdf.

[20] *CHaiDNN*. URL: https://github.com/Xilinx/chaidnn (visited on 2020-05-27).

[21] *PipeCNN*. URL: https://github.com/doonny/PipeCNN (visited on 2020-05-27).

[22] Xilinx. *Zynq DPU v3.2 product guide*. Tech. rep. PG338 (v3.2). Mar. 2020, p. 66. URL: https://www.xilinx.com/support/documentation/ip_documentation/dpu/v3_2/pg338-dpu.pdf.

[23] Christopher W. Fletcher. *EECS150: Interfaces: "FIFO" (a.k.a. Ready/Valid)*. URL: https://inst.eecs.berkeley.edu/~cs150/Documents/Interfaces.pdf (visited on 2020-04-21).

APPENDIX A
Definitions, terminology and acronyms

- FPGA: Field Programmable Gate Array.
- AXI: Advanced Extensible Interface.
- RTL: Register-transfer level.
- HDL: Hardware description language.
- ASIC: Application-specific integrated circuit.
- MAC: Multiply Accumulate.
- VHDL: Very High Speed Integrated Circuit Hardware Description Language.
- RTL: Register-Transfer Level.
- HLS: High Level Synthesis.
- OpenCL: Open Computing Language.
- NaN: Not a Number.
- PCI: Peripheral Component Interconnect
- MIPS: Microprocessor without Interlocked Pipelined Stages.
- ALU: Arithmetic and Logic Unit.
- PC: Program Counter
- XRT: Xilinx RunTime.
- KDS: Kernel Domain Scheduler.
- TPU: Tensor Processing Unit.
- MXU: Matrix multiply unit.
- SoM: System on Module.
- TOPS: Tera OPerations per Second.
- APU: Application processing unit.

- DSP: Digital Signal Processor.
- DDR: Double Data Rate.
- PE: Processing Engine.
- AMP: Accumulating Matrix Product.
- FloPoCo: Floating Point Cores.
- FLOPS: Floating Point OPerations per Second.
- NIC: Network Interface Card.