

TECHNICAL UNIVERSITY OF CRETE

DIPLOMA THESIS

---

Efficient reconfigurable architectures for  
Backprop algorithm using High-Level  
Synthesis framework

---

*Author:* Ioannis S. Doitsinis

Thesis Committee:

*Supervisor:* Prof. D. Pnevmatikatos

Committee member: Prof. Ap. Dollas

Committee member: Doctor Gr. Chrysos



June 2018



# Acknowledgement

First of all i would like to thank my advisor, Prof. Pnevmatikatos for his help and guidance. Also, i would like to thank the members of this thesis committee, Prof. Dollas and Dr. Chrysos for their comments and advice. I couldn't thank more Dr. Chrysos and Phd candidate Ch. Vatsolakis for all their help and attention all these months, i really appreciate what you've done for me. Finally a big thank to my family and friends for their support.



# Abstract

The need for designing more efficient processing units has led researchers to seek solutions on heterogeneous system architectures, especially now that typical CPUs reach their physical boundaries as far as the number of integrated transistors and clock speed. These heterogeneous systems try to use various processing units like GPUs, ASICs and FPGAs, alongside the CPU in order to go through specific cost-efficient tasks, optimizing the overall performance. Systems integrated on FPGAs that accelerate the execution of a program are commonly called hardware accelerators.

MachSuite is a set of benchmark programs, i.e., of the most widely used hardware accelerators. It was developed in order to help researchers evaluate accelerator-centric architectures.

First, in this thesis, we analyze and describe the MachSuite benchmark. Next, we introduce a novel hardware-based architecture for one of the described applications, i.e., a Back-propagation algorithm which is used for training artificial neural networks. The proposed architecture offers a more efficient hardware-based execution exploiting hardware parallelism. Finally, we optimize the architecture of our design and evaluate it under a theoretical aspect and on a real system.



# List of Figures

1.1	Block diagram of FPGA architecture . . . . .	16
1.2	Arrangement of Slices within the CLB . . . . .	17
1.3	A heterogeneous system . . . . .	17
3.1	All the MachSuites accelerators demonstratively . . . . .	23
3.2	A selection of stencils used in various scientific applications	28
4.1	A simple Neural Network . . . . .	31
4.2	Threshold estimation for a neuron . . . . .	32
4.3	Sigmoid activation function of a neural network . . . . .	33
4.4	Plot of a Sigmoid function . . . . .	33
4.5	A Simple Neural Network . . . . .	34
4.6	Backpropagation algorithm flowchart . . . . .	38
5.1	Hardware-based architecture for function Matrix Vector Product Input Layer. In the red cycles are the modules that do the double precision addition and double precision multiplication . . . . .	44
5.2	Diagram of the whole Backprop algorithm. Each function is represented as a different module. The outputs of each function are stored in Block RAMs. The last function, i.e., update_weights takes as inputs all the intermediate matrices, calculates the new weights and biases and feeds them back to the start. . . . .	45
5.3	An example of a 5 stage pipeline structure . . . . .	47
5.4	Examples of the array partition technique . . . . .	48
5.5	Block diagram of the introduced architecture (Clusters). The bold arrows represent the assignment of the generated input matrices to FIFO structures. The intermediate matrices are also set to FIFO and as well the last cluster returns the new weights and biases with FIFOs to the start.	51
5.6	RACOS design architecture . . . . .	54
6.1	Timing of the original Backprop using the default input dataset(65KB) . . . . .	59
6.2	Latency and throughput in clock cycles for the original Backprop using the default input dataset(65KB) . . . . .	60
6.3	Memory Utilization of the original Backprop using the default input dataset(65KB) . . . . .	60
6.4	Instantiation of the algorithms functions using the default input dataset(65KB) . . . . .	61

6.5	Comparison of the initial statistics of the original algorithm and the final after the optimization . . . . .	65
6.6	The report of Vivado HLS for cluster 1 after the best optimization showing the total utilization percent on the Virtex 6 . . . . .	73
6.7	The report of Vivado HLS for cluster 2 after the best optimization, showing the total utilization percent on the board . . . . .	76
6.8	The report of Vivado HLS for cluster 3 after the best latency optimization showing the total utilization percent on the board . . . . .	78
A.1	This fixed module does an addition between two 64 bit operators and is an essential part of Backprops architecture	101
A.2	The design of a Block RAM as it is implemented by Vivado HLS tool. It is a simple straightforward design, with the main body ( <i>ram_reg</i> ) and some modules (an AND module and a Mux) to regulate the write enable and read from the RAM and some registers to store the data before the RAM output. . . . .	101



# List of Tables

3.1	Memory Utilization of all the original and without any optimization algorithms of MachSuite. The highlighted ones are more suitable for researching purposes due to their size. . . . .	29
6.1	Dataflow directive over the top function of the original architecture . . . . .	62
6.2	Loop unrolling of Exponential function of the original architecture (Previous refers to the Dataflow directive above)	62
6.3	Loop unrolling on RELU function of the original architecture (Previous refers to the loop unrolling directive above) . . . . .	62
6.4	The three functions of the original architecture after applying pipeline directives to the first two and loop unrolling directive to the third (Previous refers to the loop unrolling directive above) . . . . .	63
6.5	Back-propagation functions of the original architecture, all with pipeline directives. Those that don't have an II indication have the default II=1(Previous refers to the directives above) . . . . .	64
6.6	Update_weights function of the original architecture with pipeline directives (Previous refers to the directives on the table above) . . . . .	64
6.7	For-loops of the Update_weights function of the original architecture (Previous refers to the table of directives above)	65
6.8	The second architecture of the clusters and the accumulative performance in comparison to the original architecture . . . . .	66
6.9	The clustered architecture with the addition of FIFO directives, the comparison of Cl1+Cl2+Cl3 with the original algorithm and the difference between the Cl1+Cl2+Cl3 with FIFOs design and the without FIFOs one . . . . .	66
6.10	Cluster 1, comparison of the original form together with the optimized design, initial optimization . . . . .	67
6.11	All the top function's directives of Cluster 1, initial optimization . . . . .	67
6.12	The matrix_vector_product_with_bias_second_layer function of the first cluster, initial optimization . . . . .	67

6.13	Loop unrolling of Exponential function (Previous refers to the table of directives above) . . . . .	68
6.14	RELU function of cluster 1, initial optimization . . . . .	68
6.15	Comparison of the original cluster 2 and optimized design, initial optimization . . . . .	68
6.16	Top function of Cluster 2, initial optimization . . . . .	69
6.17	get_delta_matrix_weight2 function of cluster 2, initial optimization . . . . .	69
6.18	Pipeline on the last three functions of cluster 2, initial optimization . . . . .	70
6.19	Comparison of the original cluster 3 and optimized design, initial optimization . . . . .	70
6.20	Top function of Cluster 3 with all the added directives (Initial refers to the unoptimized cluster 3), initial optimization . . . . .	70
6.21	get_delta_matrix_weight1 function with pipeline directive, initial optimization . . . . .	71
6.22	Pipeline directives on every for-loop of the Update_weights function of Cluster 3, initial optimization . . . . .	71
6.23	The total area results of all clusters for the first optimization policy and the comparison with the available resources on a Virtex 6 FPGA. . . . .	72
6.24	Best latency optimization in comparison with the original unoptimized cluster 1 . . . . .	73
6.25	First and best latency optimization for Cluster 1 . . . . .	73
6.26	matrix_vector_product_with_bias_second_layer function of the first cluster, best latency optimization . . . . .	74
6.27	RELU function of cluster 1, best latency optimization . . . . .	74
6.28	add_bias function of cluster 1 , best latency optimization . . . . .	74
6.29	Pipeline directives on the top functions body of cluster 1 (Previous refers to the added directives shown on the table above), best latency optimization . . . . .	75
6.30	Best latency Optimization in comparison with the original unoptimized cluster 2 . . . . .	75
6.31	Best latency optimization of Cluster 2 in comparison with the first one . . . . .	75
6.32	get_delta_matrix_weight2 function of cluster 2. We showed in Table 6.17 from subsection 6.3.1 the results that a pipeline with II=32 directive had on this function. Now, we demonstrate the further optimization of this function with a different II option . . . . .	76
6.33	Top function of cluster 2 after the use of pipeline directives on the functions body (Previous refers to the added directives shown on the table above) . . . . .	77
6.34	Best latency optimization in comparison with the original unoptimized cluster 3 . . . . .	78

6.35	Best latency optimization of Cluster 3 in comparison with the first one . . . . .	78
6.36	The Dataflow directive effect over the update_weights function . . . . .	79
6.37	The Pipeline directive that was applied in the Top function of cluster 3 for the best latency optimization (Previous refers to the added directives shown on the table above) .	79
6.38	Pipeline with II=1 directive on get_delta_matrix_weight1 function (Previous refers to the added directives shown on the table above), best latency optimization . . . . .	79
6.39	The total area results of all clusters for the best latency optimization policy. . . . .	80
6.40	Test . . . . .	80
6.41	Comparison of all the execution times . . . . .	80
6.42	Worst case scenario for each cluster of the not optimized clustered architecture. The theoretical values are calculated on a Virtex 6 FPGA running at 125 MHz with Vivado HLS . . . . .	85
6.43	Best average case scenario for the not optimized clustered architecture. The theoretical values are calculations of the Vivado HLS on a Virtex 6 FPGA running at 125 MHz .	86
6.44	Execution times for 10 concurrent applications per policy	86
6.45	Best case scenario for the best latency optimized clustered architecture. The theoretical values are calculated for a Virtex 6 FPGA running at 125 MHz with Vivado HLS .	87
6.46	The execution times for the two architectures WITHOUT optimizations(The clusters design has only FIFOs to the inputs and outputs) . . . . .	88
6.47	The percent of BRAM utilization for the two architectures WITHOUT optimizations. With BOLD are the cases where the BRAM utilization is overflown as it was calculated for a VIRTEX 6 board that contains 832 BRAMs	88
6.48	The execution times for the two architectures OPTIMIZED(First optimization) . . . . .	88
6.49	The percent of BRAM utilization for the two architectures OPTIMIZED (First optimization). With BOLD are the cases where the BRAM utilization is overflown as it was calculated for a VIRTEX 6 board that contains 832 BRAMs	89
6.50	Software runtime for CL1+CL2+CL3 compared with the equivalent of the THEORETICAL hardware runtime Optimized (Best optimization) . . . . .	89
6.51	Summary of area utilization for all architectures. . . . .	90
6.52	Summary of execution time for all architectures and speed up in account of the first Architecture(No optimization) .	94

6.53	Summary of all times for all Architectures and the software run for each input dataset. (A, B refers to the Approaches of the Second Architecture ) . . . . .	95
6.54	Estimated energy consumption of the 2nd architecture on hardware and software . . . . .	95
6.55	The Dataflow appliance on the original Backprop and on the Clustered architecture . . . . .	97
6.56	The FIFO directives on the Clustered architecture . . . . .	97
6.57	The total optimization of all the pipeline directives that were applied on the original Backprop and on the Clustered architecture . . . . .	98
6.58	The total optimization of all the partially unroll directives that were applied on the original Backprop and on the Clustered architecture . . . . .	98

# Contents

<b>Acknoledgments</b>	<b>3</b>
<b>Abstract</b>	<b>5</b>
<b>List of Figures</b>	<b>7</b>
<b>List of Tables</b>	<b>9</b>
<b>1 Introduction</b>	<b>15</b>
1.1 FPGAs . . . . .	15
1.2 Hardware Accelerators . . . . .	16
1.3 High Level Synthesis . . . . .	17
1.4 MachSuite . . . . .	18
1.5 Contribution . . . . .	18
1.6 Thesis Structure . . . . .	19
<b>2 Related Work</b>	<b>21</b>
<b>3 Overview of MachSuite</b>	<b>23</b>
3.1 Memory profiling of MachSuite . . . . .	28
<b>4 Backprop implementation</b>	<b>31</b>
4.1 Neural Networks . . . . .	31
4.1.1 Sigmoid Neuron . . . . .	32
4.2 Backpropagation . . . . .	33
4.2.1 Code structure of the Backprop algorithm . . . . .	35
4.2.2 Algorithm Flowchart . . . . .	37
<b>5 Architecture Implementation</b>	<b>41</b>
5.1 Memory analysis . . . . .	41
5.2 First Backprop architecture . . . . .	42
5.2.1 Architecture . . . . .	42
5.2.2 Hardware optimizations . . . . .	46
5.3 Second Backprop architecture . . . . .	50
5.3.1 Architecture of clusters . . . . .	50
5.3.2 Hardware optimizations . . . . .	51
5.4 Final System Integration . . . . .	53
5.4.1 Best optimization of clustered architecture . . . . .	55

<b>6</b>	<b>Results</b>	<b>59</b>
6.1	Baseline performance of Backprop . . . . .	59
6.2	First Architecture . . . . .	59
6.2.1	First Architecture Performance . . . . .	61
6.3	Second Architecture . . . . .	65
6.3.1	Initial performance results for clustered version . . . . .	66
6.3.2	Best optimization of clustered version . . . . .	72
6.4	Integration . . . . .	85
6.5	Result Comparison and analysis . . . . .	90
<b>7</b>	<b>Conclusions and future work</b>	<b>99</b>
<b>A</b>		<b>101</b>
	<b>Bibliography</b>	<b>103</b>

# Chapter 1

## Introduction

Since the invention of the first computer, the computing industry is trying to find out ways to build faster, better and non-power hungry machines to serve the increasing demands in the various IT workloads.

For the last 50 years the exponential improvement of chip technologies followed the Moore's law, as defined by Gordon Moore the co-founder of Intel and Fairchild Semiconductor. Moore's law is an observation which says that the amount of components (transistors) per integrated circuit is doubling approximately every 18 months. However, many researchers in the semiconductor industry, including Gordon Moore, claim that this law will not be applicable anymore, as the transistor is exponentially decreasing in size and it is reaching its physical boundaries.

Transistors nowadays are so small that the width of their channel is only a few atoms long. This fact creates a few challenges. Firstly, modern chips have so many transistors integrated that they have reached limitations related to the amount of power they use. In other words Dennard scaling has ended. This, also, leads to cooling problems, too. Secondly, quantum mechanics start to play a significant role in these sizes where a phenomenon called quantum tunneling (an effect about electrons jumping across a potential energy barrier) creates engineering problems and sets a physical limit to the transistors size in which they work properly.

There have been some promising approaches to solve those complications and one of them is in the field of heterogeneous systems, like hardware accelerators. In this thesis we will describe the implementation of hardware accelerators on Field Programmable Gate Arrays (FPGAs) as an attempt to develop different architectures that are flexible and can achieve performance and efficiency similar to ASIC.

### 1.1 FPGAs

A Field-Programmable Gate Array (FPGA) is an integrated circuit designed to be configured by a designer after the manufacturing process. They serve as the building blocks of reconfigurable computing and their advantage is that they are sometimes significantly faster than generic CPUs for specific applications due to their parallel nature. The ability of

reconfiguration is what makes them so popular as they combine the speed of hardware with the flexibility of software. FPGAs are broadly used on applications, like digital signal processing, computer vision, cryptography and process acceleration. The designers use the FPGAs to accelerate certain parts of an algorithm and they share part of the computation between the FPGA and a generic processor, which is this thesis' theme.

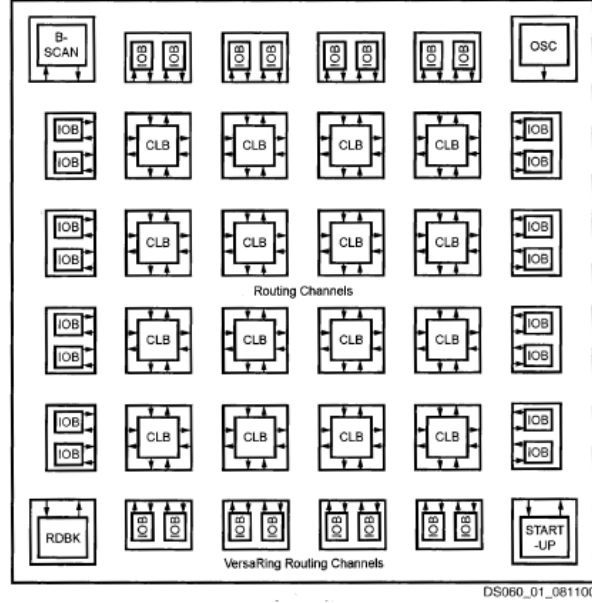


Figure 1.1: Block diagram of FPGA architecture

The basic architecture of an FPGA, as shown in [Figure 1.1](#), consists of Configurable Logic Blocks (CLBs), routing channels, input/output blocks, and also some elements that are not represented, like Digital Signal Processor slices (DSPs), and Block Rams (BRAMs). CLBs are surrounded by input/output blocks (IOBs) for communicating with external devices. Each CLB ([Figure 1.2](#)) consists of slices and each slice contains 4 Look-Up Tables (LUTs) and 8 Flip-Flops (FF) that can be configured to perform either combinational or sequential logic. The general FPGA structure allows for arbitrary configuration, so designers can connect the logic elements however necessary.

## 1.2 Hardware Accelerators

Nowadays, some applications have become so computational intensive that have high execution times on conventional CPUs. This led researchers to design heterogeneous hardware systems that utilize GPUs and FPGAs alongside the main CPU. These systems try to take advantage of concurrency and efficiency, scheduling either a complete or parts of the mapped algorithm to be computed on a specific part of the system. [Figure 1.3](#) [19] shows an example of such system. So, the



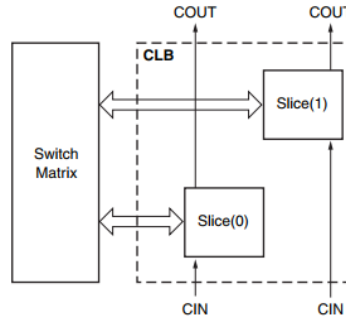


Figure 1.2: Arrangement of Slices within the CLB

hardware that performs the acceleration is separate of the main CPU and is called hardware accelerator.

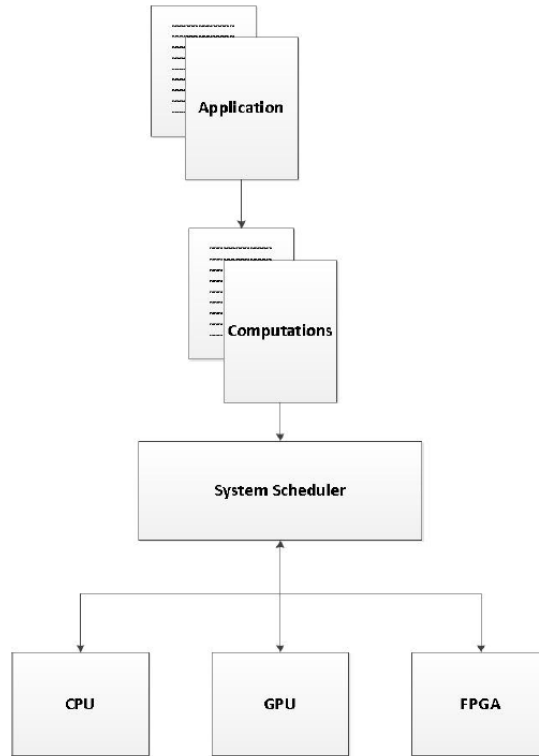


Figure 1.3: A heterogeneous system

FPGAs can be very useful in such cases and they can be programmed with small algorithms that accomplish the acceleration. A simple method to develop accelerators is the use of High Level Synthesis.

## 1.3 High Level Synthesis

High Level Synthesis (HLS) is an automated process that interprets an algorithm description in a high-level-language and produces the digital

hardware that implements that algorithm description. In other words, there are special tools like Vivado HLS that transform a C/C++ code program into register transfer level (RTL) hardware description language (HDL) implementation, which can be afterwards loaded on an FPGA and be evaluated. The goal of HLS is to give the hardware designers the ability to efficiently build, verify and optimize hardware, exploiting the flexibility and ease that a higher level of abstraction language provides, while the tool automatically implements the RTL design.

## 1.4 MachSuite

MachSuite [17] is an open-source benchmark that provides High Level Synthesis(HLS) synthesizable C code intended for accelerator-centric research. It is a collection of 19 benchmarks spanning 12 different kernels, written to cover a diverse set of applications and to include the most popular algorithms that are being used in hardware accelerator architectures. The lack of a standard and well-defined benchmark led researchers to many problems in the past, as there are many different algorithms with different coding styles and approaches for the same kernels making it hard to compare the results. MachSuite provides standardization by implementing the most popular kernels in a simple way and using realistic input data in order to help researchers to make advances.

## 1.5 Contribution

The contribution of this thesis is described below:

- We provide a detailed description and an analysis of the memory utilization for all MachSuite Benchmarks accelerators using the Vivado HLS tool.
- We describe the functionality for a MachSuite Benchmark algorithm, i.e., Backprop algorithm, which is an artificial neural network training method.
- We propose two different parallel reconfigurable architectures for the Backprop algorithm. The first architecture is based on simple HLS directives. The second architecture is based on the split of the algorithm into three parallel and independent pipelined modules (clusters).
- We apply various hardware optimizations for the two different proposed architectures of Backprop algorithm, e.g., pipelining and loop unrolling.

- We integrate the two proposed architectures into the RACOS application.
- We compare the two different proposed architectures for the Backprop algorithm and we evaluate their performance results.

## 1.6 Thesis Structure

In chapter 2 of this thesis, we present the related work in the field of hardware acceleration, as well as, the hardware acceleration applications on FPGAs. In chapter 3 we will provide an overview of the MachSuite Benchmark and in chapter 4 we will describe Backprop algorithm. Chapter 5 describes the hardware architectures and the optimization techniques we used. Chapter 6 presents and compares the performance results and finally chapter 7 concludes the thesis and provides comments for future work.



# Chapter 2

## Related Work

There are several works in the literature studying FPGA-based accelerators. All these studies are based on the key advantages of FPGAs that are parallelism, high energy-efficiency and flexibility.

There are previous works about Neural Networks and machine learning techniques, which are developed on FPGA-based accelerators. These works have been widely used on image, text and voice recognition, machine translation, scene analysis, encryption and other applications. Convolutional Neural Networks (CNN) [24], that are widely employed for image recognition, have been implemented with FPGAs accelerators due to their high performance and reconfigurability that exceeds performance of generic processors. Other types of Neural Networks like Recurrent Neural Networks [10] are developed extensively on FPGAs due to the flexibility and parallelization that they provide. An issue for the FPGA-based applications is the limited memory bandwidth provided from an FPGA platform especially when the deep learning techniques get more complex and require more computational throughput. Therefore, to tackle this problem there have been introduced methods for analyzing the computational throughput and memory requirements of a CNN design and optimizing it using the roofline model [24].

Microsoft has turn attention on FPGA-based accelerators to make search engine algorithms (Bing) more efficient and competitive [23]. The growth of the World Wide Web increased the difficulty for a search engine to fetch rapidly the search data. To carry out that task, machine learning algorithms need to be deployed, but they are slow with large datasets on general purpose CPUs. That's why Microsoft develops new generations of FPGA-based accelerators and optimizes the searching algorithms to exploit parallelism with promising results on real world search activities.

Data centers face an exponential increase in the amount of traffic that they have to serve due to cloud computing. Typical servers with general CPUs lose ground due to their excessive power consumption. On the other hand, GPUs although they provide significant performance advantages, they can be used as fixed resources in the Infrastructure-as-a-Service (IaaS) model, leading to under utilization problems [9]. Accelerators on FPGAs can be ideal for utilization on data centers as they are re-programmable and they can adapt to new workloads and new needs. Also, studies have shown that they can achieve multiple times better performance per watt than CPU/GPU

---

implementations for the same applications [14].

In order to train a neural network, a Backpropagation algorithm is used. The model of the Backpropagation Neural Network (BPNN) finds applications in various scientific fields, from classification and image recognition, to medicine and chemistry. Indicatively, some examples will be provided. The ability of a BPNN for pattern recognition makes it a useful tool for financial applications. BPNN are used to analyze financial data, to find patterns and to predict stock values [12]. In the field of pharmaceuticals, researchers use BPNN techniques to predict the effect of drug experiments on lab rats [6]. An other application of BPNN is on the field of geology and geo-engineering. BPNN are trained to identify seismic waves from recorded seismic data [11]. They are also used for mapping and predicting earth slope movement and ground movement around tunnels [13]. Finally, BPNN can find applications in the field of medicine. Researchers use neural networks to recognize X-ray images and to extract features that lead to the diagnosis of diseases [16].

Finally, we can see some examples where hardware accelerators with the combination of hardware optimizations using High-Level Synthesis can be useful. Electrocardiogram (ECG) analysis is a method for detecting heart diseases in the field of medical health care. Machine learning algorithms are used to analyze the data volume of this method and detect patterns that assess the health status of a patient. Researchers try to find efficient ways to improve performance of this method using hardware based accelerators developed with High Level Synthesis. A case study for arrhythmia detection, which is a heart disease, on an ECG medical database, used a machine learning algorithm, i.e., a Support Vector Machine (SVM) algorithm. SVM was re-structured using HLS to be transfered on HW accelerators [21]. This study, also, explored the use of HLS directives, in order to improve performance of this specific method and managed to achieve up to 94% latency improve compared to the original algorithm.

# Chapter 3

## Overview of MachSuite

This chapter describes all the accelerators that are included in MachSuite as shown in the following table. All the algorithms are characterized by simplicity to help other researchers experiment. As we see in [Figure 3.1](#), many of the algorithms provide two distinct versions for the same kernel. These algorithms solve the same problem with different ways or characteristics in order to provide more diversity.

Kernel/Algorithm	Description	Berkeley Dwarf
AES/AES	AES encryption	Combinational logic
BACKPROP/BACKPROP	Neural network training	Unstructured grids
BFS/BULK	Breadth-first search	Graph traversal
BFS/QUEUE	Breadth-first search	Graph traversal
FFT/STRIDED	Fast Fourier transform	Spectral methods
FFT/TRANPOSE	Fast Fourier transform	Spectral methods
GEMM/NCUBED	Matrix multiplication	Dense linear algebra
GEMM/BLOCKED	Matrix multiplication	Dense linear algebra
KMP/KMP	String matching	Finite state machines
MD/KNN	Molecular dynamics	N-body methods
MD/GRID	Molecular dynamics	N-body methods
NW/NW	DNA alignment	Dynamic programming
SORT/MERGE	Sorting	Map reduce
SORT/RADIX	Sorting	Map reduce
SPMV/CRS	Sparse matrix/vector multiplication	Sparse linear algebra
SPMV/ELLPACK	Sparse matrix/vector multiplication	Sparse linear algebra
STENCIL/STENCIL2D	Stencil computation	Structured grids
STENCIL/STENCIL3D	Stencil computation	Structured grids
VITERBI/VITERBI	Hidden Markov model estimation	Graphical models

Figure 3.1: All the MachSuites accelerators demonstratively

### AES

The Advanced Encryption Standard is a block cipher designed as a replacement for the Data Encryption Standard (DES) algorithm. This implementation of the algorithm processes data blocks using cipher keys of 256 bits in a series of substitution phases. AES is very useful on both hardware and software implementations, because of its parallelizability, and it uses byte-oriented arithmetic operators and transformations small enough for lookup tables. MachSuites implementation of AES provides a lookup table optimization for the primary Substitution-box (S-box)

---

which is a basic component of symmetric key algorithms in cryptography.

### **BACKPROP**

Artificial Neural Networks (ANN)[17],[20] are a widely used machine learning technique, with applications on various fields like computer vision, stock market prediction, data classification and many others. Inspired by the way biological systems, like the human brain, work, neural nets are composed of a large number of interconnected elements (neurons), which are organized in multiple layers that feed-forward information and have some weights labeled between every neuron (or node). The ANN, usually, consist of three layers, the layer of input units, one hidden layer and the layer of output units. Data flows forward through these layers and we observe the outputs. Training a neural network is an expensive task, as it requires the iterative updating of a large number of parameters (weights, biases) to finally fit the desired outputs. Their ability to extract patterns and detect trends makes a trained ANN an "expert" in a specific category, which leads to many applications. Backpropagation (Backprop) is a common method to train an ANN. It compares the outputs of an untrained network to the desired ones and it calculates an error value for each element in the output layer using a loss function. Then this error value is propagated backwards towards the input layer, updating weights on every node throughout the network. Finally, this process continues iteratively until the error is minimized and the network considers to be trained.

### **BFS/BULK**

Breadth-first search algorithms are used for traversing graphs or trees and they are build blocks for other algorithms like path finding, maximizing flows in networks, etc. Its characteristic is that, this type of algorithms traverse a tree or a graph, layer by layer and its key advantages are its parallelizability and that it can process massive data sets efficiently. The BFS/BULK[17] is a data oriented implementation of breadth-first search algorithms and uses a brute-force, data parallel method, which is typically used on Single Instruction, Multiple Data (SIMD) and vector architectures. Another feature of BFS/BULK is that the structure of the graph input affects massively the execution behavior. In more detail, the mesh graphs often create memory overflowing problems and overestimate typical graph diameter. MachSuite creates using the R-MAT[7] algorithm, which is a simple method to create quickly realistic graphs with a few parameters, a low-diameter and scale-free graph.

### **BFS/QUEUE**

A common implementation of BFS uses a queue algorithm to dynamically track the current horizon. In this way we trade off lower memory bandwidth requirements for increased bookkeeping. The BFS/QUEUE [17] variant creates an identical solution to BFS/BULK but with a



different node traversal order, showing notably different computational characteristics.

### **FFT/STRIDED**

The Fast Fourier Transform is an algorithm that computes the Discrete Fourier Transform on signals, from their original form to the frequency domain and vice versa. Applications of FFT kernels are used in almost all fields that use sinusoidal signals, such as engineering, physics, applied mathematics, and chemistry. It is the most common hardware accelerator found in literature. The canonical Cooley-Tukey "butterfly" method, which is contained in MachSuite's implementation of FFT, is characterized by a wide range of strided access patterns and nested, triangular loop structures. The MachSuite benchmark provides a straightforward, iterative implementation of a 1024-point, complex FFT.

### **FFT/TRANSPOSE**

FFT transpose is an optimization of FFT kernel that computes a series of small-radix FFTs that consist of transpose operations in order to reduce memory costs on modern acceleration architectures. This technique trades off data manipulation overhead for improved locality and it is optimized for a single, fixed-size FFT. MachSuite studies the core structure of a well-tuned GPU code and provides a 512-point, complex FFT that uses an 8-point small-radix FFT.

### **GEMM/NCUBED**

GEMM stands for GEneral Matrix to Matrix Multiplication and it multiplies two input matrices. Matrix multiplication is likely the most useful building block found in numerical software and it is important for any linear algebra package. It provides high computational density, an easily manipulated mathematical structure with extremely regular behavior and it is a common target for automatic and hand-tuning. GEMM/NCUBED [17] is a naive,  $O(n^3)$  algorithm for dense matrix multiplication provided as a well-understood reference point.

### **GEMM/BLOCKED**

Blocking is a well-known optimization technique that exploits effectively the memory hierarchy. Usually matrix multiplication algorithms use a blocked loop structure that aims to improve memory locality by commuting the arithmetic to reuse all of the elements in one block before moving onto the next. MachSuite implements a different version of GEMM that uses a fixed 8-factor blocking component.

### **KMP/KMP**

String searching algorithms are used in various applications, from packet filters to scientific codes (DNA pattern matching problems). The

---

Knuth- Morris- Pratt algorithm is a fast string matching technique with running time proportional to the sum of the length of the strings. The key improvement in KMP is a small, precomputed data structure that predicts the next position in the input string where a match could be found after a mismatch. This way the algorithm bypasses re-examination of previously matched characters. MachSuite embodies a KMP implementation with both the matching and precomputation steps.

### **MD/KNN**

Molecular dynamics simulations are a category of n-body problems that are essential for computational chemistry packages. The Molecular dynamics packet includes methods of studying the atoms movements. While most MD codes include a variety of iterated equations, the most CPU intensive component is normally the calculation of the potential[2] (the force field between the atoms), which is order  $O(n^2)$ . Both MachSuites MD benchmarks compute Lennard-Jones potentials, which is a commonly used approximation to the Van der Waals interactions between all pairs of atoms. The strength of these interactions die out as a sixth-order polynomial function of distance, thus, most simulations further approximate the force calculation by only considering nearby pairs of points. MD/KNN[17] (k- nearest neighbors) uses well-defined, fixed-length neighbor lists to track the relevant molecular interactions.

### **MD/GRID**

Another version of MD, which is widely used by many computational chemistry packages[3], replaces distinct neighbor lists with a 3-D grid. In this way, force calculations are computed on all particles in the current and adjacent grid cells. This technique trades off bookkeeping overhead to track and iterate over grid cells for improved memory locality and enables memory partitioning. MachSuites both MD/GRID[17] and MD/KNN codes use the same input set and *agree* within 0.1%.

### **NW**

The Needleman-Wunsch is an algorithm used in bioinformatics to align DNA or protein sequences. It compares two nucleotide or amino acid sequences and finds out structural or functional similarities. The algorithm is a dynamic programming method that divides the problem into separate smaller sub problems and optimizes a similarity score between two strings. MachSuites implementation of NW is a wavefront computation that populates a square similarity score matrix as it runs. Finally, the optimal alignment is reconstructed by materializing this score matrix.

### **SORT/MERGE**

Mergesort is a simple and efficient comparison algorithm invented by John von Neumann. A sorting kernel serves as a building block for many

other algorithms. Although merge sort is an outdated algorithm tends to be popular on parallel platforms due to its simple structure and low data dependencies. MachSuite[17] includes an iterative implementation of an 4096 long integer array sort.

### **SORT/RADIX**

Radix sort is a typical non-comparison-based algorithm, that sorts data by rearranging integer representations (keys) based on the same significant position and value. Radix sort is often used when handling input sets with small value ranges and in parallel contexts. Non-comparison sorts operate in a way so that they exploit properties of the value domain to lower computational complexity. MachSuites version sorts an integer array by comparing 4-bits blocks at a time.

### **SPMV/CRS**

Sparse matrix-vector multiplication methods, i.e.,  $y = Ax$ , are widely used. A sparse vector is a vector that its most elements are zeros. Sparse matrices often appear for solving partial differential equations or computing properties on high-diameter graphs. Storage and manipulating sparse matrices/vectors requires different algorithms in order to exploit the properties of sparsity. In more detail, the multiplication is identical to the dense version, but organizing and tracking the nonzero elements changes dramatically the computational characteristics. SPMV/CRS[17] uses Compressed Row Storage format for storing the nonzero elements, creating one-dimensional arrays that contain all the information for the sparse matrix. Since sparse matrix operations often depend heavily on the structure and density of the input matrix, MachSuite provides a test matrix [1] as a proxy for the behavior of an iterative solver.

### **SPMV/ELLPACK**

Ellpack is an alternative to the Compressed Row Storage (CRS), contiguous nonzero external storage format. Ellpack in order to sustain regularity in access pattern trades off memory overhead. This way it pads out each row of its nonzero matrix to the maximum length of any row, filling the empty cells with zeros in order to enable sequential access.

### **STENCIL/STENCIL2D**

Stencil codes are a class of iterative kernels, which update array elements according to some fixed patterns. These codes can be found in many applications on computer vision and scientific simulations. The inner loop of a stencil code is a fixed-size computational 2D or 3D template, which performs a sequence of sweep across a large input grid. The size and the shape of the stencil itself varies across applications.

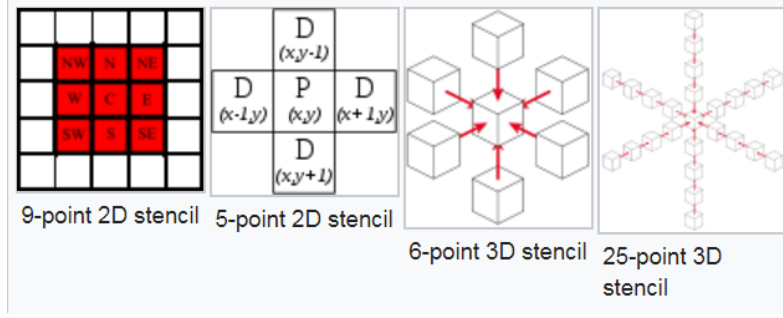


Figure 3.2: A selection of stencils used in various scientific applications

The majority of codes use a unit-stride motion across the data. MachSuite contains a typical, 9-point, 2D stencil that applies a 3x3 filter to an input array.

### STENCIL/STENCIL3D

Stencils can be found with different grid configurations and dimensions. As the grid changes, the execution characteristics change, too. To demonstrate the difference between surface and volume stencils, MachSuite also provides a 7-point star-shaped, 3D stencil.

### VITERBI

Hidden Markov models are machine learning techniques that have unobserved (hidden) states. They are widely used as stochastic models with applications ranging from information coding to pattern recognition and bioinformatics. The Viterbi algorithm is a dynamic programming method for finding the most likely sequence of hidden states called the Viterbi path based on a set of observations and a pair of probability matrices. Viterbi exhibits high computational density. MachSuite provides an implementation that computes probabilities on a Hidden Markov model.

## 3.1 Memory profiling of MachSuite

In the [Table 3.1](#) we summarize, what resources each of the algorithms would require if programmed on a Virtex6 FPGA. We used the vivado HLS xilinx tool to calculate those quantities. These measurements are upon the original and unoptimized algorithms that are contained in the benchmark. Most of the algorithms are not memory demanding due to their simplicity and mostly because they are designed this way to achieve small memory footage (around 32KB which is an average cache size) for research reasons. [Table 3.1](#), also, shows all the available memory of the targeted FPGA.

According to [Table 3.1](#), a few algorithms (Backprop, FFT/Strided, FFT/Transpose, MD/Grid and MD/KNN) are more suitable for

	BRAM-18K	DSP48E	FF	LUT
AES	4	0	479	1071
<i>BACKPROP</i>	<i>34</i>	<i>112</i>	<i>34956</i>	<i>38593</i>
BFS/BULK	0	0	381	618
BFS/QUEUE	2	0	255	642
<i>FFT/STRIDED</i>	<i>0</i>	<i>56</i>	<i>4213</i>	<i>7548</i>
<i>FFT/TRANPOSE</i>	<i>33</i>	<i>157</i>	<i>19871</i>	<i>49365</i>
GEMM/BLOCKED	0	14	1166	1848
GEMM/NCUBED	0	14	1070	1880
KMP	0	0	327	541
<i>MD/GRID</i>	<i>0</i>	<i>42</i>	<i>7605</i>	<i>11731</i>
<i>MD/KNN</i>	<i>0</i>	<i>42</i>	<i>6745</i>	<i>9858</i>
NW	0	0	485	968
SORT/MERGE	4	0	423	484
SORT/RADIX	0	0	311	837
SPMV/CRS	0	14	1203	1894
SPMV/ELLPACK	0	14	1082	1861
STENCIL/2D	0	4	127	174
STENCIL/3D	0	8	467	565
VITERBI	64	3	1548	2931
<b>Available</b>	<b>832</b>	<b>768</b>	<b>301440</b>	<b>150720</b>

Table 3.1: Memory Utilization of all the original and without any optimization algorithms of MachSuite. The highlighted ones are more suitable for researching purposes due to their size.

researching purposes due to their relative large size. For this thesis we arbitrarily picked Backprop algorithm because of its big size and its function-oriented structure.



# Chapter 4

## Backprop implementation

In this chapter we will describe the functionality of Backpropagation algorithm but first, a little more about Artificial Neural Networks.

### 4.1 Neural Networks

A neural network ([Figure 4.1](#)) is a set of interconnected "neurons" arranged in layers, i.e., the input layer, one or more hidden layers and the output layer, where we get the outputs of the network. There are connections between each neuron in every layer, as shown in [Figure 4.1](#), that show how the information flows. Also, some weight values are attributed to each connection and these weights control the signal between neurons. The NN is fed with a defined input set and it generates the observed outputs. The goal of the neural networks is to implement tasks that seem easy for a human but are very difficult and complicated of a computer, like image recognition for example.

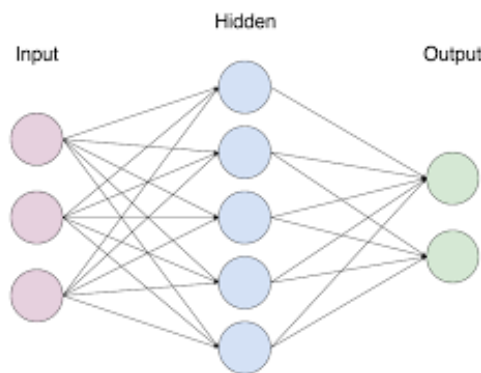


Figure 4.1: A simple Neural Network

The neural network concept, while being a simple idea, can be very complex with hundreds of neurons and many layers. They can learn to decide what an output will be based on the inputs. This "learning" process is possible for a NN as it can adapt and change the weight

values in order to subsequently change the output results. A more precise representation of a neural networks neuron is as follows:

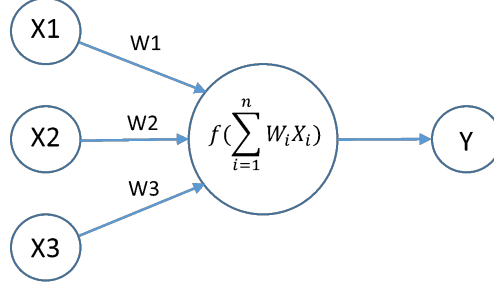


Figure 4.2: Threshold estimation for a neuron

$X_1, X_2, X_3$  are the input values and  $W_1, W_2, W_3$  are the weights of a specific neuron.  $Y$  is the neuron's output. In order to decide for  $Y$  to be either 0 or 1 the sum  $\sum_i w_i x_i$  must be greater or lower than a threshold value or else bias ( $-b$ ).

$$Y = \begin{cases} 0 & \text{if } \sum_i w_i x_i \leq -b \\ 1 & \text{if } \sum_i w_i x_i > -b \end{cases}$$

The value of threshold( $b$ ) affects massively the output. If a big positive value is set for a threshold, then the neuron will give "1" as the output and similarly a big negative threshold will cause a "0" as output. This model of neural network, though, has a disadvantage. A small change to the weight or the bias of one neuron can lead to a big change to the overall output of the neuron and change unpredictable the overall output of the network. In more detail, a marginal change to these values can make a big difference producing a "1" for a "0" and vice versa. The equation above is generic and shows how an output of a neuron is calculated.

#### 4.1.1 Sigmoid Neuron

We cannot have an unpredictable node (neuron) behavior in order to successfully train a neural network. We need the small changes to the weights and the biases, to lead to small changes to the outputs so that we can gradually and repeatedly train the network to give the correct output.

To counter the problem, actual neural networks use a different type of neurons, the sigmoid neurons, where the sum  $\sum_i w_i x_i$  is multiplied on every node with an activation function  $\sigma()$  that has a "sigmoid" shape. This function defines the output of the neuron and is essential for the learning process.

The activation function or the transfer function  $\sigma(net)$  as shown in the [Figure 4.3](#) is:

$$\sigma(net) = \frac{1}{1 + e^{-net}}$$



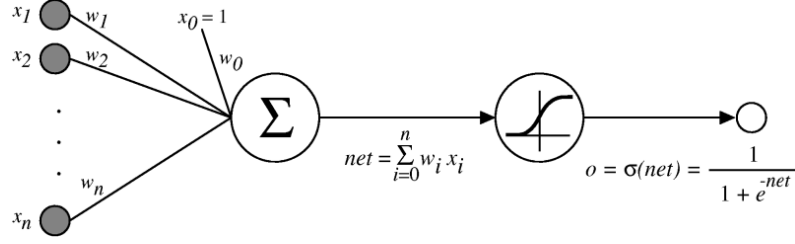


Figure 4.3: Sigmoid activation function of a neural network

This equation is widely used as activation function. It is non-linear (has a sigmoid shape) and it is in the range of (0,1). This function takes real numbers and "squashes" them into range between 0 and 1, while the "sigmoid" shape creates a smooth transition.

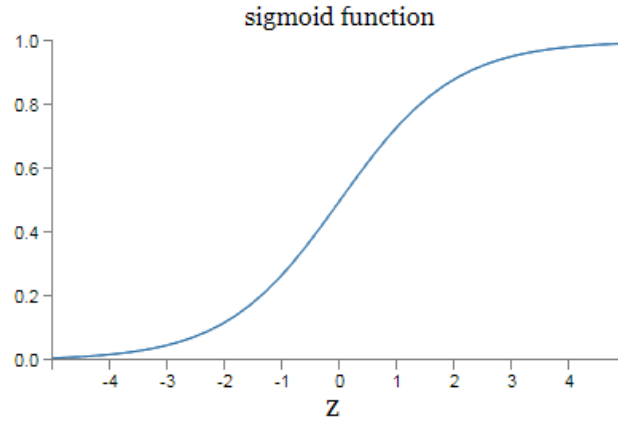


Figure 4.4: Plot of a Sigmoid function

If we put it more explicitly it becomes

$$\sigma(net) = \frac{1}{1 + e^{-\sum w_i x_i - b}} \quad (4.1)$$

Given that the sum  $\sum_i w_i x_i + b$  is a large positive number then  $e \approx 0$  and  $\sigma(net) \approx 1$ . Similarly when  $\sum_i w_i x_i + b$  is a large negative number, the  $e \rightarrow \infty$  and the neurons output is approximately 0. Last, when  $\sum_i w_i x_i + b$  has a value neither too big nor too small,  $\sigma(net)$  can have any output between 0 and 1. This property is very useful as it creates a smoothness effect that we need for the output variation in order to train a network. Also, it makes a nice interpretation of the firing rate of a actual biological neuron.

## 4.2 Backpropagation

Backpropagation is a common algorithm used to train artificial neural networks. This method is used for calculating the error contribution, or the error function, of every neuron of the network after a set of input

data has been given throughout the network. This error information is propagated backwards layer by layer through the network updating each neuron weight and bias until the network is considered to be trained. Backpropagation method requires a known desired output for each input in order to find the error function and to train the network. [18] [15]

To explain more thoroughly this process lets first consider a simple network, like in Figure 4.5

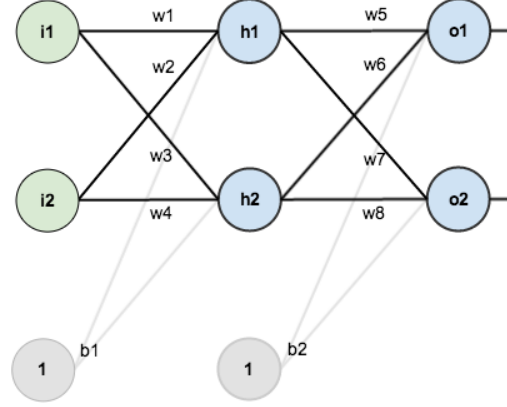


Figure 4.5: A Simple Neural Network

$i_1, i_2$  are the input neurons,  $w$  the weight of each connection,  $h_1, h_2$  two neurons of the hidden layer,  $o_1, o_2$  the output neurons and  $b_1, b_2$  the biases that also have weight attributes. The Backpropagation algorithm works in two phases.

1. Forward input propagation until the network generates an output. Then, the calculation of the error using the known target outputs takes place.
2. Backpropagate the error by starting from output layer and updating every weight attribute until input layer.

Considering the first phase, we need to show how the input propagates through the network. For example, for the simplest network that we defined previously, Figure 4.5, the net input and output of each hidden neuron layer is calculated using equation 4.1. Next, we do the same for the output neurons and this leads us to the net output of the network.

After the calculation of the network outputs, we can calculate the error function using the equation

$$E_{total} = \frac{1}{2} \sum (target - output)^2 \quad (4.2)$$

After the error calculation, we can start the second phase by backpropagating this error and updating the weights. This process will

change slowly the overall output so that to be closer to the target, thus minimizing the  $E_{total}$ .

In our example [Figure 4.5](#) we have to find out the change of  $W_5$  in order to decrease the total error by a little. In other words we need to find the partial derivative of  $E_{total}$  with respect to  $W_5$ .

$$\frac{\partial E_{total}}{\partial W_5}$$

This partial derivative can be written as [\[15\]](#)

$$\frac{\partial E_{total}}{\partial W_5} = -(target_{o1} - out_{o1}) * out_{o1}(1 - out_{o1}) * out_{h1} \quad (4.3)$$

Finally, when we have the error contribution of the specific weight, we subtract this value from the current weight to get the new weight using also a new variable  $\alpha$ , i.e., the learning rate.  $\alpha$  is the speed that the whole network needs to be trained. If  $\alpha$  is high, the network will be trained fast but it will not be very accurate. The new weight will be:

$$W'_5 = W_5 - \alpha * \frac{\partial E_{total}}{\partial W_5} \quad (4.4)$$

We continue and calculate in the same way  $W_6$  but we don't update the new weights yet. We need the old weights to calculate correctly the hidden layer weights. Finally, this procedure updates all the weights and the biases of all neurons. Next, we feed forward the same input in order to take a different output with smaller error. This process takes place iteratively for many thousands times until the error is sufficiently small, thus, the network considers to be trained.

### 4.2.1 Code structure of the Backprop algorithm

The original algorithm is implemented in a single file the `backprop.c`. There are also three more files, which are important for the algorithm to run. The `generate.c` generates the initial values of the weights, the biases, the `training_data` and the `training_targets` using a random generation function, i.e., `rand()`, and stores them into matrices with predefined sizes. The other two files are, the `backprop.h`, which defines the matrices sizes and helps with customization, and the `support.h`, which has general definitions, instructions and macros.

The `generate.c` file calls the function `backprop()` with parameters all the initialized matrices that constitute the input of the algorithm.

The Backprop algorithm is structured in functions that divide the computations and provide comprehension to the programmer. The original version has a main function "`backprop()`" that calls all the others.

The algorithms functionality is described in the following functions:

1. *matrix\_vector\_product\_with\_bias\_input\_layer()* is the first function that is called and it gets as input all the first layers weights and biases. It calculates the internal algorithm values based on the typo:  $\sum_i w_i x_i + b$ .
2. *RELU()* function gets the above value as input and calculates the activation and deactivation function of the first layer neurons.
3. *matrix\_vector\_product\_with\_bias\_second\_layer()* function gets as input the weighs, the biases of the hidden layer and the activation function of the first layer and it gives the sum for the hidden layer. Then *RELU()* is called again, thus, the activation and the deactivation functions are calculated for the hidden layer.
4. *matrix\_vector\_product\_with\_bias\_output\_layer()* function takes as input all the weights and the biases of the output layer neurons and the previously calculated activation function and it produces the last sum. Next, *RELU* gets that sum value and gives the last activation and deactivation functions.
5. *soft\_max()* function has as input the activation value for the output layer and "squashes" it in the range  $[0,1]$ .
6. *take\_difference()* function takes as input the output of *soft\_max()* comparing it with the training targets and giving the error of the networks outputs.
7. *get\_delta\_matrix\_weights3()* function starts the backpropagation part. It gets as input the output error and the activation function of the hidden layer and produces the new values of the output layer weights, the delta weights3.
8. *get\_oracle\_activation2()* function takes the output error and the deactivation function of the hidden layer nodes and it finds the new values for the biases of the hidden layer, i.e., the *oracle\_activations2* variable.
9. *get\_delta\_matrix\_weights2()* function gets the *oracle\_activation2* and the activation function of the input layer and computes the delta weights2.
10. *get\_oracle\_activation1()* function, again, gets the *oracle\_activation2* and the deactivation function of the input layer and finds the *oracle\_activation1*.
11. *get\_delta\_matrix\_weights1()* finds the last variable that needs to be computed, i.e., the delta weights1. This function uses the *oracle\_activation1* function and the input data of the network.

12. `update_weights()` function, finally, updates all the `weights(1,2,3)` with the delta `weights(1,2,3)`, the `biases(1,2)` with `oracle_activations(1,2)` and the `biases3` with the error value of the outputs.

### 4.2.2 Algorithm Flowchart

In this section we present the flowchart of the MachSuites Backpropagation algorithm [Figure 4.6](#).

First, the algorithm generates random values, which initialize the first input of weights and biases. It, also, generates the training data, which are the target values and the input data of the neural network.

Next, this input is given to the algorithm and the process begins. Based on the [Equation 4.1](#), the activation function for each neuron of the first layer is computed and stored in a matrix. Then the algorithm uses this matrix and the same equation to calculate the activation function for all the hidden layer neurons and it stores again those values on a different matrix. The third step calculates in the same way the activation function for the output layer neuron and stores it to another matrix. This matrix contains the original output of the untrained network.

After that, the algorithm calculates the total error ([Equation 4.2](#)) using the targets that were set at the beginning and the last matrix that contains the original outputs. This is the last step of propagation phase and the backpropagation starts.

Firstly, in the backpropagation phase, the new biases of the output layer are calculated in a matrix called "oracle activation(3)". This uses the partial derivative of the activation function "deactivation", which is based on the following equation:

$$deactivation(\sum_i w_i x_i + b) = activation(\sum_i w_i x_i + b) * (1 - activation(\sum_i w_i x_i + b)) \quad (4.5)$$

The "oracle activation" function is the product of the previously calculated error and the d(e)activation function. This is the crucial information that is backpropagated and trains the network.

Next, the algorithm computes the new weights of the output layer, i.e., the "delta weights", with the use of the previous "oracle activation" and the activation function of the hidden layers nodes.

Next, the algorithm proceeds back to the hidden layer after the calculation of the oracle function and the delta weights of the output layer. There, the algorithm computes the "oracle activation(2)", the delta weights and the activation function of the input layer.

## 4.2. BACKPROPAGATION

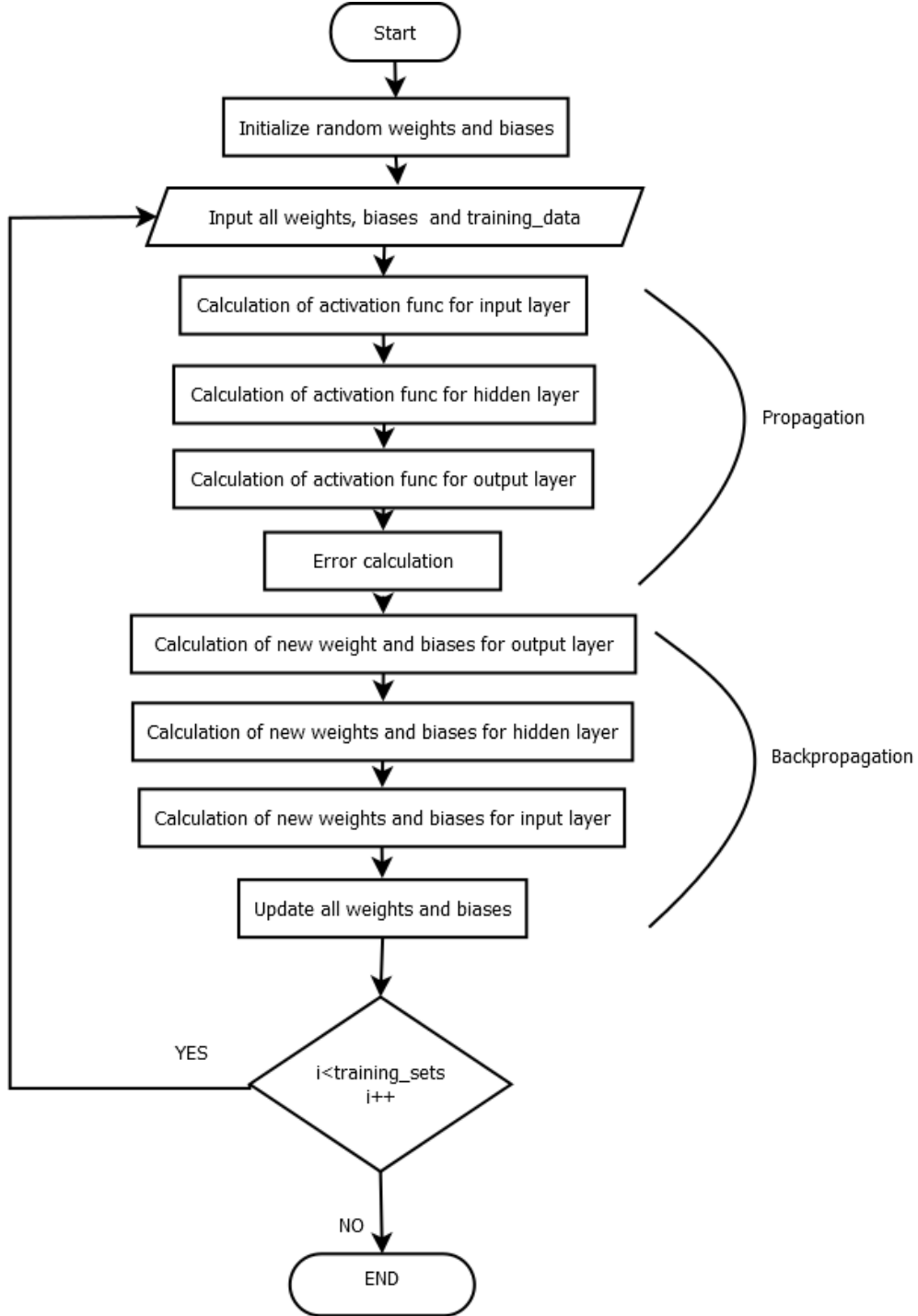


Figure 4.6: Backpropagation algorithm flowchart

Then, the final "oracle activation(1)" for the input\_layer is found as:  $OracleActivation(1) = OracleActivation(2) * dactivation(1)$  and the "delta weight(1)" as:  $DeltaWeight(1) = OracleActivation(1) * InputSet$ .

Last, the algorithm updates all the weights and the biases after the calculation of all the parameters. The delta weight matrices are used to

update the old weights and the oracle activation functions for the old biases. As we pointed before in [Equation 4.4](#), the new weight is:

$$\text{New Weight} = \text{Old Weight} - \text{Learning Rate} * \text{Partial Derivative of Error Sunction}$$

The partial derivative is calculated *a priori* in each oracle activation function so we have all we need to proceed the update. In the same way, all the biases are updated. At this point the first training set considers to be completed.

The above process runs iteratively within a for-loop for all the training sets that we defined initially until the produced error is small and the network is trained.





# Chapter 5

## Architecture Implementation

### 5.1 Memory analysis

The subject of this thesis is to map an algorithm on hardware using high level synthesis. In order to do that, we need firstly to describe the memory requirements of an FPGA. This will provide comprehension about the memory units that are utilized by the HLS tool and further knowledge about the functionality of an FPGA. Secondly, we need to analyze each application of the MachSuite benchmark to find the more memory demanding one, so that our research will be challenging. We did that process in chapter 3 and we chose baackprop algorithm for our experiments.

For the needs of this thesis we used as a reference device a Virtex6 Xilinx FPGA (XC6VLX240T). The Virtex6 FPGA consists of 37680 total slices, 832 Block RAMs 18KB and 768 DSP48E. Each slice contains 4 LUTs and 8 Flip Flops so the total amount of available Flip Flops is 301440 and of LUTs is 150720.

Block RAMs 18KB have a true dual-port configuration meaning that they consist of 18KB storage area and two independent access ports, which are used for reading or writing. Vivado HLS can automatically analyze the design and choose the number of ports on a BRAM that will maximize the data rate. The designer has also the option to select accordingly, using directives. Read operation takes 2 clock cycles to complete as an output register is interposed, whereas write operation takes 1 clock cycle. Block RAMs are also flexible and can be configured for example 16K x 1, 8K x 2, 4K x 4, 2K x 9, 1K x 18 or 512 x 36.

DSP48E is a Digital Signal Processor slice embedded inside virtex 5 and 6 FPGAs that can support some functions like multiply, multiply accumulate (MACC), multiply add, one- or n-step counter, logic operations(AND,OR) and more.

There are two main components that constitute the logic resources on an FPGA, LUTs and Flip-Flops. A Look Up Table (LUT) is basically a table that determines what the output for any given input(s) is. In more detail, it is a truth table that effectively defines how the combinatorial logic behaves. Virtex6 uses 6 input LUTs. Flip-Flops (FF) are circuits that have two states and can store information. One FF represents one bit of information. Both LUTs and FF logic resources are grouped in

slices to create configurable logic blocks.

## 5.2 First Backprop architecture

In order to compile, simulate and synthesize the Backprop algorithm into RTL (Register-Transfer Level) verilog code we used the Xilinx Vivado HLS tool. First, we simulated the C code of the algorithm with the HLS tool in order to confirm that the algorithm works fine, i.e., it produces the correct output (small error). Next via vivado HLS we chose the function that will be synthesized into verilog (top function). In the first architecture, the main function *backprop(...)* was selected as top function and it was synthesized. After synthesis all the needed hdl files were produced. Next, we co-simulated the C code (testbench, libraries) with the verilog code in order to verify the RTL design. Last, we exported the RTL system for use in other Xilinx tools, like Vivado.

### 5.2.1 Architecture

This section presents the reconfigurable architectures for the backprop algorithm.

#### Matrix Vector Product Input Layer

The first function that is mapped on hardware is *Matrix\_vector\_product\_with\_bias\_input\_layer()*. This function takes as input three double precision matrices. The Biases1[ ], the Weights1[ ] and the training\_data[ ] matrices are the three input matrices and the output is activations1[ ] matrix. The output matrix is double-precision type, too.

Its functionality is to add and multiply elements of these input matrices within two nested for-loops in order to produce the output matrix. The tool by default implements the multiplication and addition operations using DSP48E macro cells. It uses fixed numbers of DPS48E, that depend on the type of operation and the precision type of the variables. In any case 3 DSP48Es are utilized for addition of double type variables (dadd) and 11 DSP48Es for multiplication of double type variables (dmul).

The main body of this function is this nested four loop.

```
for(j = 0; j < nodes_per_layer; j++){
    activations[j] = (TYPE)0.0;
    for (i = 0; i < input_dimension; i++){
        activations[j] += weights[j*input_dimension + i] * tr_data[i];
    }
}
```

The Vivado HLS tool implements this piece of code in the following way. Firstly the tool will implement modules that will do simple operations, between integers, like (addition, multiplication, logical comparison etc). With these modules and as well with registers and multiplexers, the tool will implement the following part of the code:

```
j = 0, i = 0
j < nodes_per_layer, i < input_dimension
j++, i++
j*input_dimension + i
```

Next, the tool uses two fixed modules, one that does a simple addition of two double-precision (64-bit) elements and one that does a multiplication of the same kind of elements (Appendix). These two components are the part of the logic that it will implement this part:

```
weights[j*input_dimension + i] * input_sample[i]
activations[j] + (weights[j*input_dimension + i]} * tr_data[i]);)
```

The tool, finally, uses registers, multiplexers and modules for addition and comparison to implement the for-loop structure itself. It uses these logic components to control the operations inside the loop structure and to make them iterate until the exit condition is met. In the same way is the outer for-loop mapped in the hardware.

```
for (i = 0; i < input_dimension; i++)
{}
```

Figure 5.1 shows the whole Matrix Vector Product Input Layer function, that was piece by piece described, as it is mapped in the RTL. The figure shows a general idea of how the various components are differentiated with a special shape and are interconnected. In the red cycles are presented the two components that do the double precision addition and multiplication.

## BLOCK RAM

Between the core functions of the algorithm, intermediate matrices are mapped to store the in-between data. The tool utilizes all these matrices as Block RAMs

Block RAMs are fast and simple memory blocks and that's why the tool uses them automatically for the main matrices of the algorithm. Naturally, the number of BRAMs a matrix uses, depends on its size. The minimum amount of RAM that can be assigned to a matrix is 2 x 18KB RAM because Virtex 6 FPGAs contain 36KB Block RAM modules divided in two separate 18K independent blocks [5]. The only downside about them is the limited port number that they have and the small available amount of them on an FPGA (Appendix).

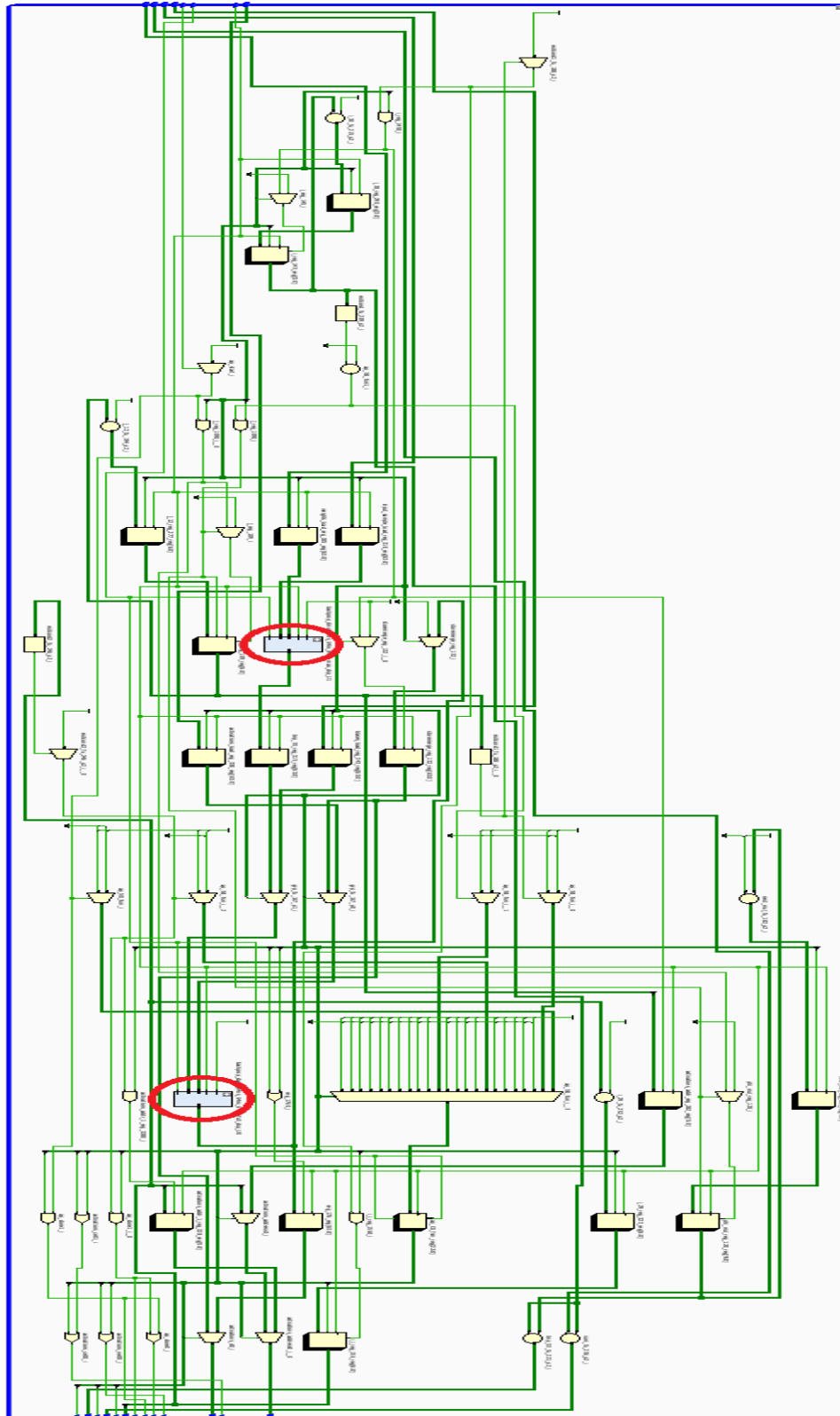


Figure 5.1: Hardware-based architecture for function Matrix Vector Product Input Layer. In the red cycles are the modules that do the double precision addition and double precision multiplication

### The whole algorithm

All the rest functions of the mapped algorithms follow the same functionality of the `Matrix_vector_product_with_bias_input_layer` function. Each function is mapped as an independent hardware module. Secondary matrices are mapped as Block RAM in between the functions. The architecture of the whole Backprop algorithm is presented in Figure 5.2.

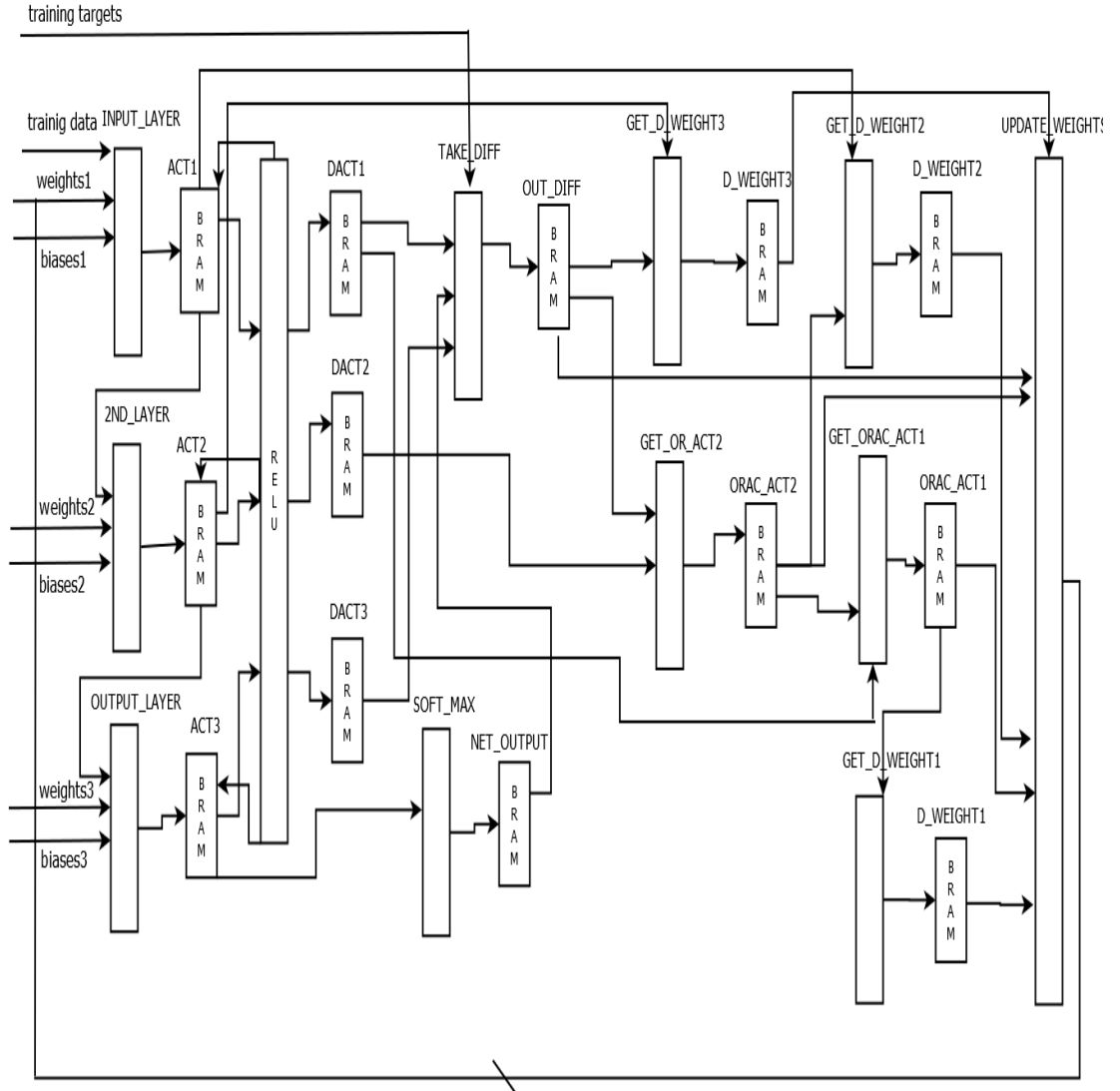


Figure 5.2: Diagram of the whole Backprop algorithm. Each function is represented as a different module. The outputs of each function are stored in Block RAMs. The last function, i.e., `update_weights` takes as inputs all the intermediate matrices, calculates the new weights and biases and feeds them back to the start.

### 5.2.2 Hardware optimizations

Vivado HLS attempts to optimize latency and throughput for the mapped architectures. First, the tool attempts to minimize the interval between new inputs(throughput). Second, it attempts to minimize the latency and finally it tries to map the architecture on hardware using the less possible resources. It also takes advantage of the parallelism, and when it is possible it schedules functions to run concurrently.

Vivado HLS tool provides many options for manual optimization of the mapped design in order to improve the overall performance. This takes place with the use of special directives. There are three different techniques which can be used to optimize performance:

#### 1. Latency Optimization

Latency can be shortened by using special directives on a function or on a block of code, setting the min and max latency that is desired. Another method is also, loop manipulation, like loop unrolling, flattening and merging that reduce loop transition overheads.

#### 2. Throughput Optimization

For throughput improvement there is the Dataflow directive that is applied on the top-level function, which maps the other functions and loops to operate in parallel. Another method is the use of pipeline directives which offers concurrent/parallel execution of functions and loops. These two methods work best in combination with each other.

#### 3. Array Optimization

Finally, arrays can create bottlenecks due to memory and port accesses, which result in latency and throughput delay. We can eliminate those bottlenecks with the use of directives for partitioning and reshaping arrays.

We used mainly loop unrolling and pipelining directives for mapping our algorithm, as it consists mostly of functions that have nested for-loops. We, also, kept the clock frequency around 8 ns as that is the target frequency of our FPGA. Additionally, we tried to have the optimum trade-off between area and execution time as the memory utilization grows with the latency improvement. The throughput of the algorithm is attempted to be maximized with the Dataflow and pipeline directives but data dependencies hinder this attempt.

Vivado HLS keeps by default all loops rolled. The operations in each iteration use the same hardware. The tool gives the opportunity to unroll a loop either partially or fully. Below we can see what a loop unrolling in C code means. [4]

```
int n;
for (n = 0; n < 100; n++)
{
    b=a[n]+b;
}
```

Listing 5.1: A rolled loop

```
int n;
for (n = 0; n < 100; n +=2)
{
    b=a[n]+b;
    b=a[n+1]+b;
}
```

Listing 5.2: Partially Unrolled loop

In this example lets say that  $a[ ]$  is mapped in a Block-RAM. The rolled loop runs for 100 times and it requires an adder and a single port BRAM to read  $a[ ]$ . The other loop is partially unrolled by a factor of 2, which means that it will take half of the iterations to complete, but here the trade-off is that it needs two adders and a dual port BRAM.

There is the option to fully unroll a loop, but except of the multiplication of the hardware utilization there is one more issue. For the above example, if we unroll the loop by factor 4, we will need four adders and a 4-port BRAM that performs four reads per cycle. As there is not that type of BRAM available, the array must be partitioned.

Pipelining is a technique that improves throughput while mapping different operations in parallel. This technique exploits the way a processor is divided in stages (five stages in our example for simplicity) the Instruction Fetch (IF), Instruction Decode (ID), Execution (EX), Memory Access (MEM) and register Write Back (WB) stages, and loads instructions in every clock cycle [Figure 5.3](#).

Instr. No.	Pipeline Stage						
1	IF	ID	EX	MEM	WB		
2		IF	ID	EX	MEM	WB	
3			IF	ID	EX	MEM	WB
4				IF	ID	EX	MEM
5					IF	ID	EX
Clock Cycle	1	2	3	4	5	6	7

Figure 5.3: An example of a 5 stage pipeline structure

In that way each stage does not stay idle until the instruction is done but executes the next , minimizing the throughput at 1 cycle.

In Vivado HLS when applying a pipeline directive in a function or in a loop, it fully unrolls all loops in the hierarchy and as a result it uses high percentage of hardware. With the help of the loop unrolling directive we can control this "time vs space" trade-off. The pipeline directive by default attempts to load a new instruction every clock cycle, in other words to keep the Initiation Interval (II) at 1 cycle. In some cases though, this requirement cannot be met, usually when some instructions

try to load or store multiple times simultaneously in a Block RAM, that has only two ports (bottleneck). Also, data dependencies between loop iteration may result stalling the pipeline. Then the tool increases this II accordingly, resulting delay in throughput by some clock cycles. The total clock cycles are given from:  $(\text{Loop Trip Count} - 1) * II + \text{Iteration latency}$ .

The limited BRAM port problem can be solved by manipulating an array. As we stated before arrays are mapped in BRAMs. There is the `ARRAY_PARTITION` directive that partitions arrays into smaller structures, providing in that way more data ports. There are different options for the partition. The partition can be block or cyclic by a factor (usually by 2) or complete as it is shown in Figure 5.4.

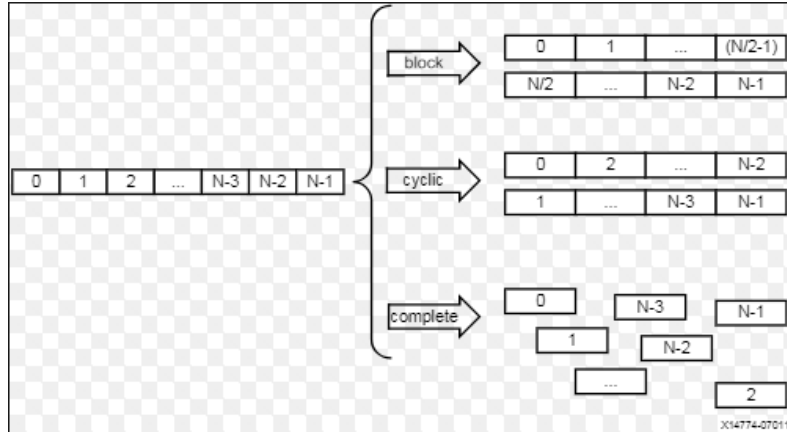


Figure 5.4: Examples of the array partition technique

### Optimizations on original Backprop

Next, we will describe the directives that are used on every function as long with the details of the optimization.

First, we used the *Dataflow* directive that is applied on the top function. Generally in C all operations take place sequentially. Dataflow directive analyzes the data flow and tries to enable a function or a loop to start execution before the previous one completes all of its operations. It can achieve concurrency this way optimizing throughput and latency. This optimization takes place by adding memory channels at the size of the variables, which help the data to flow between tasks, but with a small area overhead. These channels can be configured and set to ping-pong buffers of FIFO buffers. In some cases we chose FIFO buffers to minimize the memory used by the channels.

Next function we'll analyze is *exponential()*. This function is called many times in the algorithm so its optimization is quite important. We *fully unrolled* this loop. Pipeline directive could not be used in this case because the double precision division and double precision addition operations (the body of the loop) created a critical path resulting longer



clock time (13ns with target at 8ns).

```

TYPE exponential(TYPE x){
    int n=16; sum=1;
    for (int i = n - 1; i > 0; --i )
    {
        sum = 1 + x * sum / i;
    }
}

```

Listing 5.3: Backpropagations function "exponential"

The next function we optimized is *RELU()*. This function is called many times and needs to have improved latency for some area penalty. We chose to use *pipeline* directive on this function, but the tool calculated that due to data dependencies the II should be 16 for those issues to be resolved.

Next, we worked for three functions *matrix\_vector\_product\_with\_bias\_input\_layer()*, *matrix\_vector\_product\_with\_bias\_second\_layer()* and *matrix\_vector\_product\_with\_bias\_output\_layer()*. We used *pipeline* directives in the first two and *loop unrolling* for the third one. In the first function we had no issues. A bottleneck conflict appeared in the second function and the tool computed that the number that solved the conflict was 2. So we set II=2 in the directives options. The third was unrolled with factor 4 in order to have the best trade-off between latency and area.

The next mapped functions are these that do the backpropagation part, i.e., *take\_difference()*, *get\_delta\_matrix\_weights3()*, *get\_oracle\_activations2()*, *get\_delta\_matrix\_weights2()*, *get\_oracle\_activations1()* and *get\_delta\_matrix\_weights1()*. Some of the functions had no dependency or bottleneck issues and were *pipelined* achieving also a small percentage improvements on latency. Others, like the larger functions, i.e, *get\_delta\_matrix\_weights2()*, *get\_delta\_matrix\_weights1()* and *get\_oracle\_activations1()*, all had dependency constrains that needed to be resolved with II=32 in the pipeline options.

The *weights\_upgrade()* function is especially interesting because it is the biggest in functionality, as it consists of many for- loops, some of them nested, that update the new values to the matrices. We used *pipeline* directive on most of them. The directives were applied on the outer loop of the nested ones so that the tool unrolls completely all the loops in the hierarchy for pipelining.

Finally after the use of the directives we described above, we managed all together to achieve the best trade-off between latency and area as it was our goal. More details in chapter 6.

## 5.3 Second Backprop architecture

The general idea for our new architecture design of Backprop algorithm was to break the algorithm into three parts that will become three independent IP cores. These cores can be loaded on different PRRs in the same FPGA, are interconnected with FIFO interfaces to stream the data and are executed in concurrency targeting, in that way, better performance.

### 5.3.1 Architecture of clusters

Backprop algorithm is structured in functions. The idea is to group these functions into three clusters, trying also to make them utilize about the same area. Each cluster will also have its inputs and outputs to FIFO modules in order to achieve concurrency while streaming the data from one cluster to another. This FIFO assignment can easily take place using directives in Vivado HLS. The queue structure has an input to write the data and two signals, the *empty* signal to show when the queue is empty and the *read* signal for reading input data. This addition to our design naturally creates an increase to the total area, because the FIFOs require area( registers and multiplexers).

Figure 5.5 shows the diagram of our new architecture.

Instead of having the algorithm in one piece, we now have it in three independent parts, of course without changing its functionality. The bold arrows represent the FIFO interconnection from the initial input of the algorithm. The connection between the clusters is also FIFO, as well as the feedback to the start to continue the loop until the backpropagation is done.

Each cluster maps some functions together into a single module for each case. So in the beginning for cluster 1, the top level function has become the *matrix\_vector\_product\_with\_bias\_input\_layer()*. The first cluster also maps the propagation phase of the algorithm which consists of the following functions: *exponential()*, *add\_bias\_to\_activations()*, *RELU()*, *matrix\_vector\_product\_with\_bias\_second\_layer()* and *matrix\_vector\_product\_with\_bias\_output\_layer()*.

The second cluster has as top function the *soft\_max* function and contains almost all the backpropagation stages. The second cluster contains: *take\_difference()*, *get\_delta\_matrix\_weights3()*, *get\_oracle\_activations2()* and *get\_delta\_matrix\_weights2()* functions.

The last cluster is *get\_oracle\_activations1()* function and it mainly updates the weights and the biases. It contains : *get\_delta\_matrix\_weights1()* and *update\_weights()* functions.

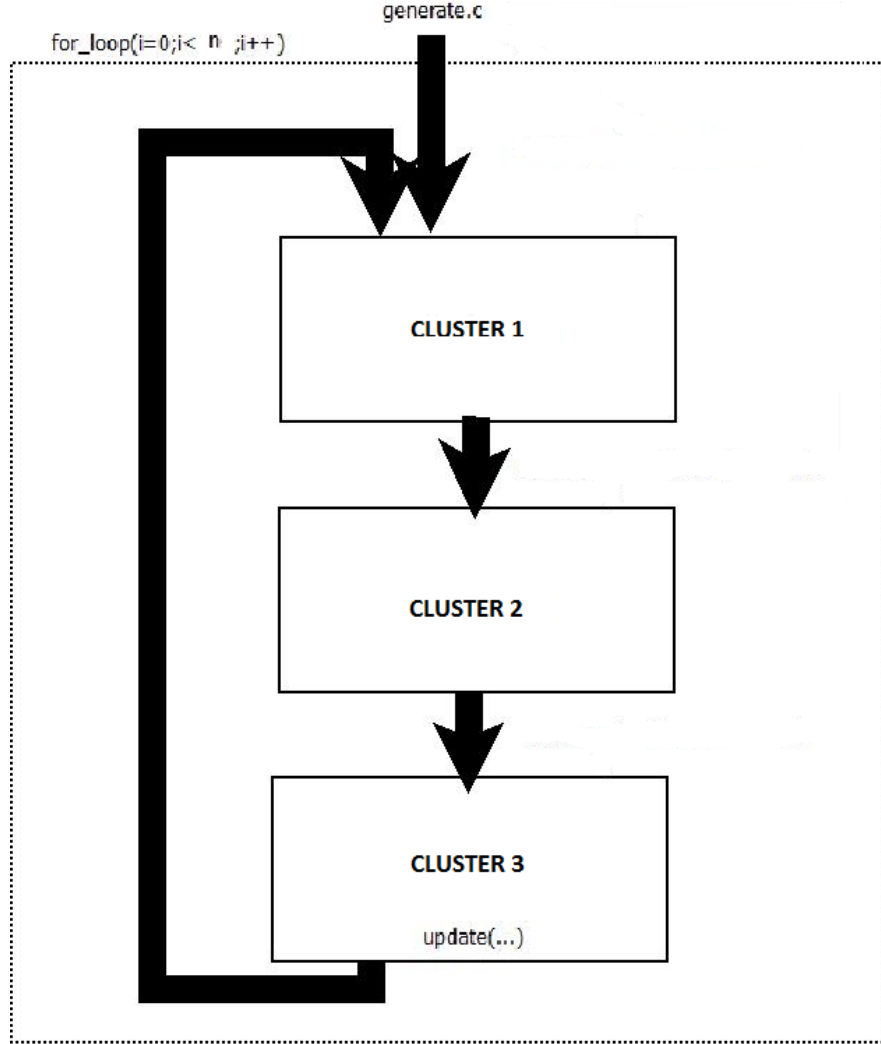


Figure 5.5: Block diagram of the introduced architecture (Clusters). The bold arrows represent the assignment of the generated input matrices to FIFO structures. The intermediate matrices are also set to FIFO and as well the last cluster returns the new weights and biases with FIFOs to the start.

### 5.3.2 Hardware optimizations

The *FIFO* directive addition was our first optimization, which allows the data to stream between the clusters. This design was essential for our architecture, as we need the data to stream in order to achieve parallel execution of the clusters. FIFOs also are required for the mapping of our system on a hardware platform, which is described in the next section. Our policy was again to choose pipeline and loop unrolling directives that provide about the same percentage of acceleration for the same amount of area increase.

### CLUSTER 1

We mention first of all that all inputs and outputs of all three clusters were set to *FIFO* interface.

In respect to the **first** cluster we used *Dataflow* directive to its top Function, i.e., *matrix\_vector\_product\_with\_bias\_input\_layer()* and *pipeline* directive on its function's body. Then the side functions *matrix\_vector\_product\_with\_bias\_second\_layer()*, *add\_bias()*, *matrix\_vector\_product\_with\_bias\_output\_layer()* and *RELU()* were *pipelined* or *unrolled*.

The *matrix\_vector\_product\_with\_bias\_second\_layer()* function as it is the larger in terms of iterations had impressive results. It consists of two for-loops that are nested and the *pipeline* directive unrolled fully these loops in order to schedule the instructions to be fetched at every cycle. Also, we set the option *II=2* on the directive in order to resolve the problem with the bottleneck of loading operation of an array.

Next function that was optimized is *RELU()*. We stated before that this function is called many times so its optimization was highly efficient. We used here the *pipeline* directive. As the function's loop was fully unrolled, a carried dependency constrain came along on the store operation for *activations[ ]* variable inside the loop, as it is shown in the list of code below. The tool automatically calculated that the target of the pipeline directive should be *II=16* in order to get that issue resolved.

```

TYPE RELU( ) {
    int n=1;
    for( i = 0; i < size; i++) {
#pragma HLS PIPELINE II=16
        dactivations[i] = activations[i]*(1.0-activations[i]);
        activations[i] = 1.0/(1.0+exponential(activations[i]));
    }
}

```

Listing 5.4: Example of a pipeline directive on "RELU" function

Finally we unrolled the inner loop of the last function *add\_bias()*. The loop contains a simple add operation and we *unrolled it for factor 4* in order to have the best trade off.

### CLUSTER 2

As far as the **second** cluster, we used *pipeline* directive to all its functions and *Dataflow* directive to the top function. More specifically the *Dataflow* directive didn't make any latency improvement but decreased a bit the used resources (2% FFs and 3% LUTs). This is based on the fact that the tool automatically schedules the functions to operate in parallel when ever possible so the throughput is optimal. This directive adds memory channels (with FIFO buffers) between tasks to maximize data flow, fact that explains the area utilization decrease.

This cluster has also a function that maps almost all the clusters complexity. This function is *get\_delta\_matrix\_weights\_2()* and was

optimized with a *pipeline* directive. Although, there was a carried dependency constrain, causing the directive to have  $II=32$ , we kept this  $II$  option to 32 in order to have the greatest trade off for latency.

The other three functions *get\_delta\_matrix\_weights\_3()*, *get\_oracle\_activations\_2()* and *take\_difference()* were also *pipelined* with no issues whatsoever, giving accumulatively the last piece of improvement.

### CLUSTER 3

Finally, the **third** cluster has also mostly been pipelined. Typically, for the top function (*get\_oracle\_activation1()*) was used the *Dataflow* directive, while the inputs and the outputs were applied with *FIFO* interface directives and also the main body of the function was *pipelined*. There was an issue in pipelining the main body of this top function, which contains two nested for- loops. There was a carried dependency constrain issue when pipelining the outer loop, that needed to be resolved with the option  $II=32$  of the directive. This directive also created too much area so the trade off wasn't efficient enough. So, we decided to pipeline only the inner loop that gave a better trade off with only a small bottleneck issue which was resolved with the  $II=4$  option.

The second function of this cluster is *get\_delta\_matrix\_weight1()* which was also *pipelined*. There was also an issue here, as the load operation could not be scheduled due to few memory ports.  $II=32$  solved this issue, leading to some latency cycles loss.

Finally, there is the *update\_weights()* function which embodies more than 60% of the clusters functionality. It contains 12 for loops, and 6 of them are double nested loops. We used the *pipeline* directive to all of them, leading to a satisfying speedup for a reasonable area trade off. More details about the above will be given in Chapter 6.

## 5.4 Final System Integration

For the last part of this thesis we tried to combine our work with an already existent application about hardware accelerators developed from researchers in our laboratory.

This application is called **RACOS**(Reconfigurable ACcelerator OS)[22] and in a nutshell is a system that produces an interface between software and FPGA hardware and provides the capability to a user to load\unload compatible accelerators on the Partial Reconfiguration Regions (PRR) of a FPGA and evaluate them. The system is capable of loading multiple accelerators on a FPGA and scheduling them for execution using four policies: *Simple*, *In Order*, *Out of Order* and *Forced*. The first two policies consider the submission order, while the other two target to reduce the number of reconfigurations.

Figure 5.6 shows the hardware architecture of this system [8].

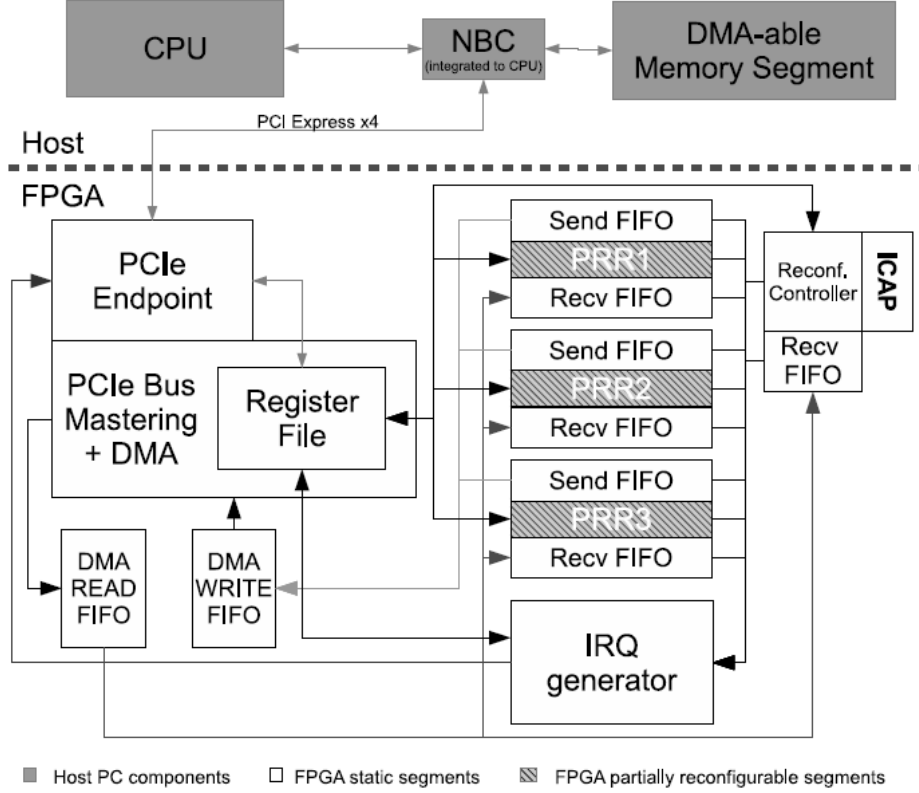


Figure 5.6: RACOS design architecture

The figure is divided into two parts, that represent the two components of the design. The Host is a typical desktop computer (gray blocks) and the other part is an FPGA board that is connected to the host PC with the PCI express interface. Data can be transferred from the Host to the FPGA and vice versa via DMA (Direct Memory Access) transactions. The FPGA is divided into two segments, the static and the reconfigurable as it is highlighted in the figure. The static part contains all the necessary components for loading and accessing the accelerators, which are the PCIe and DMA that control the data I/O, the IRQ generator that sends interrupts to the CPU informing for the current state, the Reconfiguration Controller that schedules the data into the PRRs via the ICAP port and various FIFOs and buffers that temporarily save and transfer data. Finally, the Partially reconfigurable segments are the area where the accelerators are loaded and executed.

We used this system to load and test our backprop accelerator. We tested the original architecture with and without optimizations, the second architecture (Clusters) with and without optimizations and the third design of the clustered version with best optimization for the latency. The results will be presented in Chapter 6.

### 5.4.1 Best optimization of clustered architecture

In this last section we describe our **second Approach** on the second architecture. We optimized the clustered version of Backprop algorithm trying to achieve the maximum latency improvement. We scrutinize each cluster to find that point where the latency speed up is maximum.

#### Cluster 1

With regard to the first cluster, we analyzed further from the Vivado HLS result report the latency estimations for each function and each loop. Two of the functions, `RELU()` and `matrix_vector_product_with_bias_second_layer()` are the most time-consuming. In section 5.3.2 we described that some of the applied pipeline directives had an Initiation Interval (II) different than 1 clock cycle, which is the target. Also, some loops were not fully unrolled. We agreed with this compromise because we did not concern about the area increase.

For this design, we moved towards minimizing the latency of the mapped architecture. Taking this in mind, we applied on the two functions, `matrix_vector_product_with_bias_second_layer()` and `RELU()`, *pipeline* directives enforcing the tool to achieve  $II=1$  in combination with *Array partition* directives on the arrays that created those issues. We also *fully unrolled* `add_bias()` functions for-loops and in that way we hit a ceiling on minimizing latency but for a radical area increase.

#### Cluster 2

The same logic stands also for cluster 2. We kept all the already applied directives and added a few more on the places where we previously decided that it did not worth it. The main improvement came for the `get_delta_matrix_weights2()` function, which initially had an issue with the pipeline directive. For our first optimization policy we decided to keep the  $II=32$  option as the tool suggested, solving the issue and keeping in that way the best trade off. For our second policy we needed all the latency improvement so we enforced the tool to achieve  $II=1$  no matter how much area must be utilized. Finally we used *pipeline* directives on the top function over all the for-loops (even the small ones) trying to find the point where the latency optimization is max. The results are shown in subsection 6.3.2.

#### Cluster 3

A lot of effort was put on cluster 3, too, and especially in `update_weights()` function that has many loops and operations. Our policy was again to keep the efficient optimizations and try to find out ways to maximize the speed up adding new ones or improving the existing ones.

For the first policy, we observed that some loops could not be pipelined efficiently because of data dependencies and array bottlenecks, compelling us to apply  $II > 1$  or to avoid pipelining them altogether. In some cases we enforced the  $II=1$  option but in other ones this did not work. We talk about for-loops, where pipeline directives created a significant decrease of the clock speed due to critical path issues that were created by operations like division and multiplication. So now we decided to manually manipulate them. For example the following loop created a clock increase (13ns from 8ns) because of data dependencies inside the loop when the pipeline directive tried to fully unroll it.

```
for(i=0; i < input_dimension; i++){
    for(j = 0; j < n_p_l; j++){
        weights1[i*n_p_l + j] -= (d_weights1[i*n_p_l + j] * l_r);
        norm += weights1[i*n_p_l + j]*weights1[i*n_p_l + j];
    }
}
```

Listing 5.5: Example of a loop that created data dependencies before optimization

We firstly *flattened* manually the above loop, then we applied manual *unrolling with factor 8* and finally we managed to solve the critical path problem with the  $II=4$  option in the pipeline directive.

```
for(i=0; i < input_dimension*n_p_l; i+=8){
#pragma HLS PIPELINE II=4
    weights1[i] -= (d_weights1[i] * l_r);
    weights1[i+1] -= (d_weights1[i+1] * l_r);
    weights1[i+2] -= (d_weights1[i+2] * l_r);
    weights1[i+3] -= (d_weights1[i+3] * l_r);
    weights1[i+4] -= (d_weights1[i+4] * l_r);
    weights1[i+5] -= (d_weights1[i+5] * l_r);
    weights1[i+6] -= (d_weights1[i+6] * l_r);
    weights1[i+7] -= (d_weights1[i+7] * l_r);
    norm += weights1[i]*weights1[i]
           + weights1[i+1]*weights1[i+1]
           + weights1[i+2]*weights1[i+2]
           + weights1[i+3]*weights1[i+3]
           + weights1[i+4]*weights1[i+4]
           + weights1[i+5]*weights1[i+5]
           + weights1[i+6]*weights1[i+6]
           + weights1[i+7]*weights1[i+7];
}
```

Listing 5.6: The previous loop manually manipulated



Finally we applied *pipeline* directives to every for-loop of the cluster 3 to ensure that everything is at max latency efficiency.



# Chapter 6

## Results

### 6.1 Baseline performance of Backprop

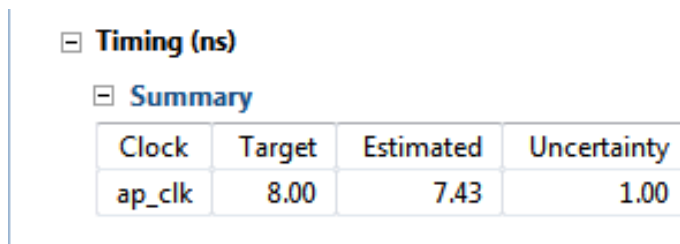
The MachSuite benchmark provides fully functional and synthesizable code for each hardware accelerator that it describes. We used the provided input data set for Backprop algorithm to conduct our experiments and to evaluate our introduced architectures.

For the first architecture of Backprop the input dataset is provided along with the algorithm. A `generate.c` file produces the initial input that is given to the algorithm. The weight and the bias matrices for the input, the hidden and the output layer are randomly produced, as well as the training data. The training target matrices are filled with ones. The size of all matrices is arbitrarily chosen by the MachSuite developers and the total amount of the input dataset ranges around 65 KB.

We used the execution results of this original algorithm, with this specific input dataset as a reference point for comparison with our mapped architectures.

### 6.2 First Architecture

First, the Xilinx Vivado HLS tool was used to compare the performance of the proposed hardware-based architecture of the algorithm vs. its software-based solution. This tool creates automatically the hardware architecture of the algorithm and gives information about the performance and memory configuration of the mapped hardware architecture.



The image shows a screenshot of the 'Timing (ns)' summary table from the Xilinx Vivado HLS tool. The table has four columns: 'Clock', 'Target', 'Estimated', and 'Uncertainty'. There is one row of data for 'ap\_clk' with a target of 8.00, an estimated value of 7.43, and an uncertainty of 1.00.

Clock	Target	Estimated	Uncertainty
ap_clk	8.00	7.43	1.00

Figure 6.1: Timing of the original Backprop using the default input dataset(65KB)

The first result that is shown on the synthesis report of the tool is the timing, [Figure 6.1](#). The target device is a Virtex 6 device that uses clock at 125 MHz, i.e, 8 ns clock cycle. The tool estimated that the mapped algorithms will have a 7.43 ns clock cycle, which is acceptable.

#### ▣ Latency (clock cycles)

##### ▣ Summary

Latency		Interval		Type
min	max	min	max	
99507915	99507915	99507916	99507916	none

Figure 6.2: Latency and throughput in clock cycles for the original Backprop using the default input dataset(65KB)

Second, the report shows the latency of the algorithm. The default input dataset (65 KB) was used to test the algorithm. The tool calculated that the mapped algorithm takes 99.507.915 cycles (of 8ns each), which means about 0,796 sec to finish execution. Interval shows the clock cycles that are needed for this design with this specific input dataset to accept a new set of input data.

##### ▣ Summary

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	-	0	808
FIFO	-	-	-	-
Instance	-	112	31219	36406
Memory	34	-	0	0
Multiplexer	-	-	-	1379
Register	-	-	3737	-
<b>Total</b>	<b>34</b>	<b>112</b>	<b>34956</b>	<b>38593</b>
<b>Available</b>	<b>832</b>	<b>768</b>	<b>301440</b>	<b>150720</b>
<b>Utilization (%)</b>	<b>4</b>	<b>14</b>	<b>11</b>	<b>25</b>

Figure 6.3: Memory Utilization of the original Backprop using the default input dataset(65KB)

Finally the report displays the memory utilization of the produced hardware. [Figure 6.3](#) shows the total use of the hardware resources in terms of BRAM\_18K, DSP48E, FFs and LUTs. The hardware-based mapping of Backprop algorithm needs 34 BRAMs, 112 DSP48Es, 34956

Flip Flops and 38593 LUTs. The report shows also, the amount of those components that are available on the Virtex6 board as well the utilization percent. We can also see more thoroughly how that memory is used. The Expression line on the report shows the area resources, that some small operations, i.e., add, sub, or, etc, need, to be mapped on hardware. The Memory line refers to the RAM space, that is used mainly for the input matrices and for some intermediate matrices, which are used for storing the data as the algorithm runs. The report also shows the usage of some extra multiplexers and registers for addressing and storing some matrices in the main function. Finally, the Instance line, which takes the most memory, represents the algorithms functionality divided in functions. In more details, [Figure 6.4](#) shows how the tool maps each function of the algorithm.

Detail

Instance

Instance	Module	BRAM_18K	DSP48E	FF	LUT
<a href="#">grp_backprop_RELU_fu_768</a>	backprop_RELU	0	14	5799	6811
<a href="#">grp_backprop_RELU_1_fu_778</a>	backprop_RELU_1	0	14	4857	5474
<a href="#">backprop_dadddsub_64ns_64ns_64_5_full_dsp_U71</a>	backprop_dadddsub_64ns_64ns_64_5_full_dsp	0	3	446	797
<a href="#">backprop_dmul_64ns_64ns_64_7_max_dsp_U72</a>	backprop_dmul_64ns_64ns_64_7_max_dsp	0	11	341	212
<a href="#">grp_backprop_matrix_vector_product_with_bia_fu_812</a>	backprop_matrix_vector_product_with_bia	0	14	1320	1355
<a href="#">grp_backprop_matrix_vector_product_with_bia_1_fu_784</a>	backprop_matrix_vector_product_with_bia_1	0	14	1859	2089
<a href="#">grp_backprop_matrix_vector_product_with_bia_2_fu_799</a>	backprop_matrix_vector_product_with_bia_2	0	14	1327	1378
<a href="#">backprop_mux_3to1_sel2_64_1_U73</a>	backprop_mux_3to1_sel2_64_1	0	0	0	64
<a href="#">backprop_mux_3to1_sel2_64_1_U74</a>	backprop_mux_3to1_sel2_64_1	0	0	0	64
<a href="#">backprop_mux_3to1_sel2_64_1_U75</a>	backprop_mux_3to1_sel2_64_1	0	0	0	64
<a href="#">backprop_mux_3to1_sel2_64_1_U76</a>	backprop_mux_3to1_sel2_64_1	0	0	0	64
<a href="#">grp_backprop_soft_max_fu_758</a>	backprop_soft_max	0	14	5430	6671
<a href="#">grp_backprop_update_weights_fu_734</a>	backprop_update_weights	0	14	9840	11363
Total	13	0	112	31219	36406

Figure 6.4: Instantiation of the algorithms functions using the default input dataset(65KB)

### 6.2.1 First Architecture Performance

In this section we will describe the contribution of each directive that we applied on the original version of Backprop algorithm architecture.

First, we used the *Dataflow* directive ([Table 6.1](#)). As we stated before, this directive adds channels between tasks to allow the data to flow. The interesting thing about that assignment in our case is that it lead to a small decrease of the total latency (64385 cycles  $\sim$  0.06%) with an appreciable decrease in area. -14 BRAMs (-12.5%), -4569(-12.8%) FFs and -5132(-13.3%) LUTs. The lack of a great latency improvement is due to the tools compiler automatically tries to schedule the functions to run concurrently when data is ready. The area decrease is because of slight

## 6.2. FIRST ARCHITECTURE

modifications all over the design, mainly multiplexers and intermediate arrays were subtracted and replaced with the FIFO configuration of the channels.

	LATENCY(cycles)	BRAM	DSP48	FF	LUT
INITIAL	99507915	34	112	34956	38593
DATAFLOW	99443530	34	98	30387	33461
CHANGE	-64385	0	-14	-4569	-5132

Table 6.1: Dataflow directive over the top function of the original architecture

The next function was `exponential()`, where we used the *loop unrolling with factor 4* directive. The choice of factor in this directive was according to the best trade-off between latency and area. We managed to save accumulative almost 3 million clock cycles ( $\sim 3\%$  improvement) of the latency with the cost of 500 FFs ( $\sim 1\%$ ) and 6081 LUTs ( $\sim 18\%$ ). All that area was used by the tool to construct the new fully unrolled structure of `exponentials()` main loop (Table 6.2).

	LATENCY(cycles)	BRAM	DSP48	FF	LUT
PREVIOUS	99443530	34	98	30387	33461
UNROLLING f=4	96838301	34	98	30887	39542
CHANGE	-2605229	0	0	+500	+6081

Table 6.2: Loop unrolling of Exponential function of the original architecture (Previous refers to the Dataflow directive above)

Then we added *pipeline with II=16* on the for-loop of `RELU()` function (Table 6.3) due to a carried dependency constrain in it. The number of the II was calculated by the tool and gives us the best trade-off. We gained about 6 million cycles latency improvement ( $\sim 6\%$ ) with area increase 12 BRAMs ( $\sim 35\%$ ), 28 DSP48Es ( $\sim 29\%$ ), 5699 FFs ( $\sim 15\%$ ) and 8322 LUTs ( $\sim 21\%$ ). Pipelining with II=1 here wasn't a feasible choice, as while trying this directive to achieve II=1, it results a radical increase of DSP48, FF and LUT utilization (about 4-5 times more).

	LATENCY(cycles)	BRAM	DSP48	FF	LUT
PREVIOUS	96838301	34	98	30887	39542
PIPELINE II=16	90428652	46	126	36586	47864
CHANGE	-6409649	+12	+28	+5699	+8322

Table 6.3: Loop unrolling on RELU function of the original architecture (Previous refers to the loop unrolling directive above)

Next, [Table 6.4](#) shows that we optimized the three functions, `matrix_vector_product_with_bias_input_layer`, `matrix_vector_product_with_bias_second_layer` and `matrix_vector_product_with_bias_output_layer` by applying *pipeline* and *unrolling* directives. The first, after *pipelined with II=1*, managed to achieve  $\sim 3\%$  acceleration for about the same percentage of area penalty. In the second was applied *pipeline with II=2* (due to array bottlenecks) and due to the many iterations that it has in its loops, achieved  $\sim 12.5\%$  latency decrease for 22% increase in FFs and 14% LUTs. The third when pipelined created way more area penalty than latency benefit so we chose to *unroll by factor 4* its loop, gaining about 0.4% time for about the same area percent increase.

	LATENCY(cycles)	BRAM	DSP48	FF	LUT
PREVIOUS	90428652	46	126	36586	47864
INPUT II=1	88593109	46	140	40527	50409
SECOND II=2	78939760	46	154	49130	57891
OUTPUT f=4	78907812	46	154	49304	58137
CHANGE	-11520840	0	+28	+12718	+10273

Table 6.4: The three functions of the original architecture after applying pipeline directives to the first two and loop unrolling directive to the third (Previous refers to the loop unrolling directive above)

Next, we will describe the functions that map the backpropagation part and we will present them together in [Table 6.5](#). *Pipeline* directives were used for all of them. The `take_difference()`, `get_oracle_activations2` and the `get_delta_matrix_weights3()` functions were pipelined with no problem (II=1). For the rest functions appeared conflicts on data dependencies that were resolved increasing the II option of the directives. Thus, `get_delta_matrix_weight2` and `get_oracle_activation1` functions had II=32 due to loop-carried dependencies (a constrain in loops that one iteration must wait for the previous one to finish) and `get_delta_matrix_weight1` had II=32 due to array bottlenecks. All these directives gained accumulatively about 22% acceleration with 13% BRAM, 62% DSP48E, 31% LUT and 23% FF increase utilization.

Later, there is the `Update_weight()` function that is the most computational intensive function. [Table 6.6](#) shows the total gain that the *pipeline* directives caused on the function.

More specifically, the `Update_weights` function contains 12 for-loops and all of them were pipelined. [Table 6.7](#) shows in details the pipeline directives for each for-loop. The loops that are nested have an (N) indicator next of their name in the table. Loops like number 5 and 7 have the more iteration and thus came from their optimization the most speed up. After optimizing all these for-loops, we can see that this function alone gained almost 48 million clock cycles that is about 50% speed up to

## 6.2. FIRST ARCHITECTURE

PIPELINE	LATENCY(cycles)	BRAM	DSP48	FF	LUT
PREVIOUS	78907812	46	154	49304	58137
TAKE_DIFF	78756711	46	154	49297	58163
MATRIX_W3	78435112	48	165	49817	58456
OR_ACT2	77907970	48	168	50569	59350
MAT_W2 II=32	71535811	50	190	52790	61279
OR_ACT1 II=32	62457852	50	229	62710	69948
MAT_W1 II=32	61165099	52	251	64870	71733
CHANGE	-17742713	+6	+97	+15566	+13596

Table 6.5: Back-propagation functions of the original architecture, all with pipeline directives. Those that don't have an II indication have the default II=1(Previous refers to the directives above)

PIPELINE	LATENCY(cycles)	BRAM	DSP48	FF	LUT
PREVIOUS	61165099	52	251	64870	71733
UPDATE_W	13247500	52	254	86882	91600
CHANGE	-47917599	0	+3	+22012	+19867

Table 6.6: Update\_weights function of the original architecture with pipeline directives (Previous refers to the directives on the table above)

the original algorithm, for the cost of 3 DSP48, 22012 FFs( $\sim 60\%$  more) and 19867 LUTs( $\sim 50\%$  more) on the original memory requirements of the algorithm (34956 FFs, 38593 LUTs).

Finally, Backprop algorithm was about 86.3% faster, from **796 ms** to **108.3 ms** based on 8 ns clock period. The throughput of the algorithm is the same with the latency. Data dependencies allowed for no concurrency so the only performance improve came from the latency speed up. The penalty for this improvement was the increase of 52% for **BRAM**, 126% for **DSP48E**, 148.5% for **FF** and 137% for **LUT** as it is presented in [Figure 6.5](#)



PIPELINE	LATENCY(cycles)	BRAM	DSP48	FF	LUT
PREVIOUS	61165099	52	251	64870	71733
Loop 1 (N)	55949450	52	254	67921	74749
Loop 2	55780563	52	254	68055	74899
Loop 3 (N)	50980153	52	254	71398	76786
Loop 4	50399630	52	254	71425	76801
Loop 5 (N)	35112582	52	254	77998	82412
Loop 6	34925627	52	254	78068	82499
Loop 7 (N)	16882648	52	254	82832	87533
Loop 8	16256482	52	254	82850	87579
Loop 9 (N)	14880148	52	254	84323	89209
Loop 10	14536925	52	254	84361	89289
Loop 11 (N)	13488509	52	254	86063	91333
Loop 12	13247500	52	254	86882	91600
CHANGE	-47917599	0	+3	+22012	+19867

Table 6.7: For-loops of the Update\_weights function of the original architecture (Previous refers to the table of directives above)

Latency (clock cycles)			
		solution1	solution2_opt
Latency	min	99507915	13247500
	max	99507915	13543182
Interval	min	99507916	13247501
	max	99507916	13543183

Utilization Estimates		
	solution1	solution2_opt
BRAM_18K	34	52
DSP48E	112	254
FF	34956	86882
LUT	38593	91600

Figure 6.5: Comparison of the initial statistics of the original algorithm and the final after the optimization

## 6.3 Second Architecture

The second architecture without any directive is a little slower than the original one (about 10 million cycles  $\sim 10\%$  that is about 80 ms) as the data is stored in arrays before exiting and after entering the next module. That process needs, also, extra area for storing and reading the data.

### 6.3. SECOND ARCHITECTURE

	LATENCY(cycles)	BRAM	DSP48	FF	LUT
ORIGINAL	99507915(796 ms)	34	112	34856	38593
CLUSTER 1	29882627(239 ms)	34	70	16484	18088
CLUSTER 2	9812600(78.5 ms)	28	14	7820	9925
CLUSTER 3	70024800(560 ms)	86	28	12913	14211
CL1+CL2+CL3	109720027(877.5 ms)	148	112	37056	42842
CHANGE	+9762112(81.5 ms)	+108	0	+2200	+4249

Table 6.8: The second architecture of the clusters and the accumulative performance in comparison to the original architecture

#### 6.3.1 Initial performance results for clustered version

The addition of *FIFO* directives had a small enhancement in speed up (2.8 million cycles  $\sim 2\%$ ) despite the fact that the data is now streamed continuously between clusters. This result is not impressive for our algorithm because for the most part this algorithm has a sequential execution with many data dependencies. There was also a slight decrease on resources.

	LATENCY(cycles)	BRAM	DSP48	FF	LUT
ORIGINAL	99507915	34	112	34856	38593
CLUSTER 1	29024595	34	70	16363	18093
CLUSTER 2	9748541	28	14	7721	9917
CLUSTER 3	68103139	86	28	12668	14189
CL1+CL2+CL3	106876329	148	112	36752	42221
CHANGE	+7368414	+108	0	+1896	+3628
FIFO- NO FIFO	-2843698	0	0	-304	-621

Table 6.9: The clustered architecture with the addition of FIFO directives, the comparison of Cl1+Cl2+Cl3 with the original algorithm and the difference between the Cl1+Cl2+Cl3 with FIFOs design and the without FIFOs one

#### CLUSTER 1

##### Overall performance of Cluster 1

Table 6.10 shows the overall optimization of cluster 1 as well the total percentage of speed up for the equivalent change in area.

CL1	ORIGINAL	OPTIMIZED	CHANGE
LATENCY	239 ms	20 ms	-92%
BRAM_18K	34	36	+5%
DSP48E	70	112	+60%
FF	16484	30795	+86%
LUT	18088	32153	+77%

Table 6.10: Cluster 1, comparison of the original form together with the optimized design, initial optimization

### Cluster 1 in detail

Table 6.11 shows all the applied directives on the TOP function of Cluster 1. The *Dataflow* and *FIFO* directives were applied over the whole top function and the *pipeline* directive was applied on the top functions body, which is a for- loop.

CLUSTER 1	LATENCY(cycles)	BRAM	DSP48	FF	LUT
INITIAL	29882627	34	70	16484	18088
FIFO	29042595	34	98	16363	18093
DATAFLOW	29036958	34	98	16202	18055
PIPELINE	27190193	34	84	20155	20482
CHANGE	-2692434	0	+14	+3671	+2394

Table 6.11: All the top function's directives of Cluster 1, initial optimization

The applied directives on the top function managed to gain 2.6 million clock cycles ( $\sim 9\%$ ) for 14 DSP48Es( $\sim 20\%$ ) 3671 FFs ( $\sim 22\%$ ) and 2394 LUTs( $\sim 13\%$ ) Table 6.11.

Next, we used *pipeline* directive on the `matrix_vector_product_with_bias_second_layer` function. In this case the Initial Interval option was set  $II=2$  due to limited memory ports of the BRAMs where the arrays are stored. The pipelining gave 35% latency improvement with the penalty increase in area of 2 BRAMs(+5%), 14 DSP48Es(+16%), 8603 FFs(+42%) and 7477 LUTs(+50%) Table 6.12.

CLUSTER 1	LATENCY(cycles)	BRAM	DSP48	FF	LUT
PREVIOUS	27190193	34	84	20155	20482
PIPELINE $II=2$	17536844	36	98	28758	27959
CHANGE	-9653349	+2	+14	+8603	+7477

Table 6.12: The `matrix_vector_product_with_bias_second_layer` function of the first cluster, initial optimization

Next is Exponential() function. We used here, as with the original backprop the *loop unrolling with factor 4* directive. The choice of factor

### 6.3. SECOND ARCHITECTURE

in this directive was according to the best trade-off between latency and area. We managed to save accumulative almost 2.5 million clock cycles ( $\sim 14\%$  improvement) of the latency with the cost of 289 FFs ( $\sim 1\%$ ) and 3851 LUTs ( $\sim 14\%$ ). (Table 6.13).

	LATENCY(cycles)	BRAM	DSP48	FF	LUT
PREVIOUS	17536844	36	98	28758	27959
UNROLLING f=4	14973729	36	100	29047	31810
CHANGE	-2563115	0	+2	+289	+3851

Table 6.13: Loop unrolling of Exponential function (Previous refers to the table of directives above)

Next is RELU() function. Here, it was used a *pipeline* directive with II=16 (loop-carried dependencies on a division operation) that saved about 12 million cycles, which is a further 81% speed up, reaching our optimum time for only 12 DSP48Es (12% more), 1748 FFs (6% more) and 343 LUTS (1% more) Table 6.14.

CLUSTER 1	LATENCY(cycles)	BRAM	DSP48	FF	LUT
PREVIOUS	14973729	36	100	29047	31810
PIPELINE II=16	2566477	36	112	30795	32153
CHANGE	-12407252	0	+12	+1748	+343

Table 6.14: RELU function of cluster 1, initial optimization

Finally for cluster 1 there is the add\_bias() function. We tested for that function both, loop unrolling with various factors and pipeline directives and none of them provided improvement that was efficient enough. So we didn't use any directive.

## CLUSTER 2

### Overall performance of Cluster 2

Table 6.15 presents the total latency optimization for Cluster 2

CL2	ORIGINAL	OPTIMIZED	CHANGE
LATENCY(cycles)	9748541(78.5 ms)	2178721(17 ms)	-78%
BRAM_18K	28	34	+21%
DSP48E	14	50	+257%
FF	7721	10894	+41%
LUT	9917	12675	+27%

Table 6.15: Comparison of the original cluster 2 and optimized design, initial optimization

**Cluster 2 in detail**

The **second** cluster is the smallest one. The top function `soft_max()` wasn't optimized further, except of the *Dataflow* and *FIFO* interface directives. This function is very small and everything we tried wasn't efficient enough. So, with the *Dataflow* and *FIFO* directives we had about 0.6% latency decrease along with 3% less FFs and 2% less LUTs (Table 6.16).

CLUSTER 2	LATENCY(cycles)	BRAM	DSP48	FF	LUT
INITIAL	9812600	28	14	7820	9925
FIFO	9748541	28	14	7721	9917
DATAFLOW	9742118	28	14	7569	9698
CHANGE	-70482	0	0	-251	-227

Table 6.16: Top function of Cluster 2, initial optimization

Next, we applied *pipelining* with  $II=32$  (due to loop carried dependencies) to the `get_delta_matrix_weight2()` function (Table 6.17) and it gave us 64% latency speed up for 7% more BRAMs, 157% more DSP48Es, 23.5% FFs and 18% LUTs.

CLUSTER 2	LATENCY(cycles)	BRAM	DSP48	FF	LUT
PREVIOUS	9748541	28	14	7569	9698
PIPELINE II=32	3001219	30	36	9877	11630
CHANGE	-6747322	+2	+22	+2308	+1932

Table 6.17: `get_delta_matrix_weight2` function of cluster 2, initial optimization

The last three functions `get_delta_matrix_weights_3()`, `get_oracle_activations_2()` and `take_difference()` were also *pipelined* with no dependency errors and bottlenecks, giving accumulatively 24% further improvement to the latency with 13% more BRAMs, 39% more DSP48s, 10% FFs and 7.5% more LUTs as it is shown in Table 6.18.

### 6.3. SECOND ARCHITECTURE

CLUSTER 2	LATENCY(cycles)	BRAM	DSP48	FF	LUT
PREVIOUS	3001219	30	36	9877	11630
ORACLE_ACT2	2549077	32	50	10570	12154
MATRIX_W3	2182285	34	50	10791	12451
TAKE_DIFF	2178721	34	50	10894	12675
CHANGE	-822498	+4	+14	+1017	+1045

Table 6.18: Pipeline on the last three functions of cluster 2, initial optimization

### CLUSTER 3

#### Overall performance of Cluster 3

Table 6.19 shows the comparison of the original performance of cluster 3 and the optimal one, showing also the percentage of change.

CL3	ORIGINAL	OPTIMIZED	CHANGE
LATENCY(cycles)	70024800(560 ms)	7908318(63.2 ms)	-89%
BRAM_18K	86	92	+7%
DSP48E	28	42	+50%
FF	12913	40523	+213%
LUT	14211	35991	+153%

Table 6.19: Comparison of the original cluster 3 and optimized design, initial optimization

#### Cluster 3 in detail

For cluster 3 we followed the same policy as with the previous two clusters and we applied *Dataflow* and *FIFO* directives. Also, we used *pipeline* with *II=32* directive for the loop of the top functions body, which is `get_oracle_activations1` function. All these optimizations contributed the first 10% speed up but with a small penalty in area, 0.6% more FFs and 1% more LUTs Table 6.20.

CLUSTER 3	LATENCY(cycles)	BRAM	DSP48	FF	LUT
INITIAL	70024800	86	28	12913	14211
FIFO	68103193	86	28	12668	14189
DATAFLOW	68085673	86	28	12540	14009
PIPELINE II=32	61314895	86	28	12999	14371
CHANGE	-8709905	0	0	+86	+160

Table 6.20: Top function of Cluster 3 with all the added directives (Initial refers to the unoptimized cluster 3), initial optimization

The next function of this cluster is `get_delta_matrix_weight1`. This was *pipelined* with  $II=32$  (due to array bottlenecks) and has gained an extra 2% latency improvement and caused  $\sim 0.6\%$  FF and  $\sim 0.3\%$  LUT increase [Table 6.21](#).

CLUSTER 3	LATENCY(cycles)	BRAM	DSP48	FF	LUT
PREVIOUS	61314895	86	28	12999	14371
PIPELINE $II=32$	60089461	86	28	13089	14417
CHANGE	-1225434	0	0	+90	+46

Table 6.21: `get_delta_matrix_weight1` function with pipeline directive, initial optimization

Finally, for the update weights function, after the implementation of *pipeline* directives to all for- loops of the functions body we managed to achieve 70% latency improvement for 7% more BRAMs, 0% more DSP48E, 88% more FFs and 41% more LUTs as it is shown in [Table 6.22](#).

CLUSTER3	LATENCY(cycles)	BRAM	DSP48	FF	LUT
PREVIOUS	60089461	86	28	13089	14417
Loop 1 (N)	56238749	86	42	16333	16786
Loop 2	56056520	86	42	16486	16803
Loop 3 (N)	49586515	86	42	22627	19461
Loop 4	49179638	88	42	22653	19507
Loop 5 (N)	26810845	90	42	34275	30943
Loop 6	26652003	90	42	34300	30988
Loop 7 (N)	9807010	92	42	38431	33496
Loop 8	9539362	92	42	38448	33623
Loop 9 (N)	8663501	92	42	38448	33623
Loop 10	8332842	92	42	39150	34520
Loop 11 (N)	8084589	92	42	39956	35406
Loop 12	7908318	92	42	40523	35991
CHANGE	-52181143	+6	+14	+27434	+21574

Table 6.22: Pipeline directives on every for-loop of the `Update_weights` function of Cluster 3, initial optimization

### Summary for the first latency optimization and throughput

The final results for the first optimization of our architecture about the execution times are:

- Cluster 1, before optimization **239 ms** and after **20 ms**
- Cluster 2, before optimization **78.5 ms** and after **17 ms**
- Cluster 3, before optimization **560 ms** and after **63.2 ms**

The latency summary of all the clusters is **100.2 ms**, with the execution time of the three unoptimized clusters being **877.7 ms** which is about **88.6 %** speed up. The equivalent execution time for the original Backprop optimized is **108.3 ms** making the clustered version **8.1 ms** faster( $\sim 8\%$ ).

	BRAM	DSP48	FF	LUT
CLUSTER 1	36	112	30795	32153
CLUSTER 2	34	50	10894	12675
CLUSTER 3	92	42	40523	35991
CL1+CL2+CL3	162	204	82212	80819
<i>AVAILABLE</i>	<i>832</i>	<i>768</i>	<i>301440</i>	<i>150720</i>
BALLANCE	670	564	219228	69901

Table 6.23: The total area results of all clusters for the first optimization policy and the comparison with the available resources on a Virtex 6 FPGA.

### 6.3.2 Best optimization of clustered version

This section describes the different optimization policy, which we applied on the clustered version of Backprop algorithm. The second policy aims to maximize the latency optimization, independently to the area increase but still to fit inside a Virtex6 FPGA. We used this policy to find the best execution time for the algorithm.

We tried to optimize latency by changing the options on the pipeline directives, where there was some kind of dependency or bottleneck issues and the tool estimated different II value  $> 1$ . Also we fully unrolled the loops that we previously decided not to.

#### Cluster 1

##### Overall performance of Cluster 1

Table 6.24 shows the contribution of our best optimization over the initial cluster 1 without any directives.

Table 6.25 compares the results of the first optimization vs. the second one (best latency optimization) and the percentage of change.

The final results show that the first cluster initially was executed in 29.882.627 clock cycles, i.e., **239ms**, the first optimization decreased latency to 2.566.477 cycles, or, **20 ms** execution time and the best improvement that we managed to achieve is 1.987.948 clock cycles, or, **15.9ms** which means that we had **93%** speed up from the original cluster 1.

Figure 6.6 shows the report of the vivado HLS tool, where the utilization estimates show that, even though we didn't take into



CL1	Initial Design	Best Optimization	CHANGE
LATENCY(cycles)	29882627 (239ms)	1987948 (15.9ms)	-93%
BRAM_18K	34	44	+29%
DSP48E	70	368	+425%
FF	16484	100060	+507%
LUT	18088	99242	+448%

Table 6.24: Best latency optimization in comparison with the original unoptimized cluster 1

CL1	1st Optimization	Best Optimization	CHANGE
LATENCY(cycles)	2566477 (20ms)	1987948 (15.9ms)	-25%
BRAM_18K	36	44	+22%
DSP48E	112	368	+228%
FF	30725	100060	+225%
LUT	32153	99242	+209%

Table 6.25: First and best latency optimization for Cluster 1

consideration the area in our last effort for optimization, the final architecture still fits in the board.

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	-	0	580
FIFO	-	-	-	-
Instance	-	368	95464	96579
Memory	44	-	128	13
Multiplexer	-	-	-	1422
Register	-	-	4468	648
Total	44	368	100060	99242
Available	832	768	301440	150720
Utilization (%)	5	47	33	65

Figure 6.6: The report of Vivado HLS for cluster 1 after the best optimization showing the total utilization percent on the Virtex 6

### Cluster 1 in detail

For cluster 1 we didn't change the *Dataflow* of the *FIFO* directives so we took them for granted and continued from there. We changed for function

matrix\_vector\_product\_with\_bias\_second\_layer the *pipeline* II option from 2 to 1 combined with an *Array Partition* directive as it is presented in [Table 6.26](#)

CLUSTER 1	LATENCY(cycles)	BRAM	DSP48	FF	LUT
PIPELINE II=2	17536844	36	98	28758	27959
PIPELINE II=1	17334894	36	98	30796	29569
CHANGE	-201950	0	0	+2038	+1610

Table 6.26: matrix\_vector\_product\_with\_bias\_second\_layer function of the first cluster, best latency optimization

Then, the biggest change happened in RELU() function. Initially the tool suggested *pipeline* directive with *II=16* in order to resolve a carried dependency constraint. This constrain was issued on a double-precision division operation. Enforcing the tool to achieve the II option the tool utilized a large amount of DSP48, FF and LUT to create multiple instances of this operation, optimizing further the latency but with a massive cost on area as it is shown in [Table 6.27](#).

CLUSTER 1	LATENCY(cycles)	BRAM	DSP48	FF	LUT
PIPELINE II=16	2816477	36	112	30795	30153
PIPELINE II=1	2106449	42	339	84333	86644
CHANGE	-710028	+6	+227	+53538	+56491

Table 6.27: RELU function of cluster 1, best latency optimization

The next change was in add\_bias function, where we *fully unrolled* the previously, partial with factor 4 unrolled loop, as presented in [Table 6.28](#)

CLUSTER 1	LATENCY(cycles)	BRAM	DSP48	FF	LUT
UNROLLING F=4	2655107	42	121	34129	33135
UNROLLING FULL	2085748	42	141	34906	34856
CHANGE	-569359	0	+20	+777	+1721

Table 6.28: add\_bias function of cluster 1 , best latency optimization

Finally, we applied pipeline directives to all for- loops that are contained in the top function, which previously in the first optimization policy we decided that they didn't give a good trade off, achieving our best time optimization ([Table 6.29](#)).

CLUSTER 1	LATENCY(cycles)	BRAM	DSP48	FF	LUT
PREVIOUS	2081347	42	359	88917	91362
TOP FUNCTION	1987948	44	368	100060	99242
CHANGE	-93399	+2	+9	+11143	+7880

Table 6.29: Pipeline directives on the top functions body of cluster 1 (Previous refers to the added directives shown on the table above), best latency optimization

## Cluster 2

### Overall performance of Cluster 2

Table 6.30 shows the best overall latency improvement of the second cluster which firstly ran in 9.748.541 clock cycles, or, **78.5ms**. The first optimization achieved 2.178.721 cycles, or, **17ms** and the last finally achieved an execution time at 1.367.570 cycles, i.e, **11ms** which is 85% acceleration from the original (Table 6.31).

CL2	Initial Design	Best Optimization	CHANGE
LATENCY(cycles)	9748541(78.5ms)	1367570(11ms)	-85%
BRAM_18K	28	34	+21%
DSP48E	14	254	+1714%
FF	7721	64450	+735%
LUT	9917	66763	+573%

Table 6.30: Best latency Optimization in comparison with the original unoptimized cluster 2

CL2	1st Optimization	2nd Optimization	CHANGE
LATENCY(cycles)	2178721(17 ms)	1367570(11ms)	35%
BRAM_18K	34	34	0%
DSP48E	50	254	408%
FF	10894	64450	492%
LUT	12675	66763	426%

Table 6.31: Best latency optimization of Cluster 2 in comparison with the first one

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	-	0	2708
FIFO	-	-	-	-
Instance	-	254	56212	60344
Memory	34	-	0	0
Multiplexer	-	-	-	2148
Register	-	-	8238	1563
Total	34	254	64450	66763
Available	832	768	301440	150720
Utilization (%)	4	33	21	44

Figure 6.7: The report of Vivado HLS for cluster 2 after the best optimization, showing the total utilization percent on the board

### Cluster 2 in detail

Cluster 2 is the smallest of the three in area and the fastest in runtime and it didn't have a lot of margin for further optimization. The most work was done in function `get_delta_matrix_weight2` which has the most computation complexity of the cluster.

The *Dataflow*, *FIFO* interface directives and *pipeline* directives with  $II=1$  were again unchanged. Thus, we describe only the further optimization. In `get_delta_matrix_weight2` function we had to face a carried dependency issue for the pipeline directive that concluded having  $II=32$  (double precision multiplication). In this case, though, we forced the tool to achieve  $II=1$  in this function gaining almost 35% speed up with the following results in [Table 6.32](#)

CLUSTER 2	LATENCY(cycles)	BRAM	DSP48	FF	LUT
PIPELINE $II=32$	3376219	30	36	9877	11630
PIPELINE $II=1$	2321606	34	250	62229	64183
CHANGE	-1054613	+4	+214	+52352	+52553

Table 6.32: `get_delta_matrix_weight2` function of cluster 2. We showed in [Table 6.17](#) from [subsection 6.3.1](#) the results that a pipeline with  $II=32$  directive had on this function. Now, we demonstrate the further optimization of this function with a different  $II$  option

Previously, we decided for our first optimization policy not to apply any directive on the top function of this cluster as nothing was efficient enough. However, for the second policy we used *pipeline* directives on the top functions body gaining a small percentage of latency improve as it is shown in [Table 6.33](#).

CLUSTER 2	LATENCY(cycles)	BRAM	DSP48	FF	LUT
PREVIOUS	2321606	34	250	62229	64183
PIPELINE	1367570	34	254	64450	66763
CHANGE	-954036	+0	+4	+2221	+2580

Table 6.33: Top function of cluster 2 after the use of pipeline directives on the functions body (Previous refers to the added directives shown on the table above)

### Cluster 3

#### Overall performance of Cluster 3

[Table 6.34](#) shows the best overall latency improvement of the third cluster. The original cluster 3 was executed in 70.024.800 clock cycles (**560ms**), after the first optimization the execution time was 7.908.318 cycles (**63.2 ms**) and we achieved for the best case an execution time at 5.508.096 clock cycles (**44ms**) which is 92% acceleration from the original.

[Table 6.35](#) shows the comparison of our first optimization of the cluster 3 with the final optimization.

### 6.3. SECOND ARCHITECTURE

CL3	Initial Design	Best Optimization	CHANGE
LATENCY(cycles)	70024800(560ms)	5508096(44ms)	-92%
BRAM_18K	86	102	+18%
DSP48E	28	145	+417%
FF	12913	70602	+462%
LUT	14211	61078	+329%

Table 6.34: Best latency optimization in comparison with the original unoptimized cluster 3

CL3	1st Optimization	2nd Optimization	CHANGE
LATENCY(cycles)	7908318(63.2 ms)	5508096(44ms)	-30%
BRAM_18K	92	102	+9.2%
DSP48E	42	145	+245%
FF	40523	70602	+74.2%
LUT	35991	61078	+71.5%

Table 6.35: Best latency optimization of Cluster 3 in comparison with the first one

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	-	0	994
FIFO	-	-	-	-
Instance	-	145	59166	52179
Memory	102	-	128	3
Multiplexer	-	-	-	4694
Register	-	-	11308	3208
<b>Total</b>	<b>102</b>	<b>145</b>	<b>70602</b>	<b>61078</b>
Available	832	768	301440	150720
<b>Utilization (%)</b>	<b>12</b>	<b>18</b>	<b>23</b>	<b>40</b>

Figure 6.8: The report of Vivado HLS for cluster 3 after the best latency optimization showing the total utilization percent on the board

#### Cluster 3 in detail

The point from where we started the further optimization of cluster 3 is the point where we stopped the 1st optimization.

We tried the *Dataflow* directive on the `Update_weights` function as we present in [Table 6.36](#).

For the top function of this cluster that is `get_oracle_activation1()`, we kept the *Dataflow* and *FIFO* interface directives. This function had

CLUSTER 3	LATENCY(cycles)	BRAM	DSP48	FF	LUT
First Opt	9158318	92	42	40523	35991
DATAFLOW	8559293	92	84	56821	46400
CHANGE	-599025	+0	+42	+16298	+10409

Table 6.36: The Dataflow directive effect over the update\_weights function

an issue with one of the for- loops in our first optimization and we decided to pipeline it with  $II=32$  (due to iteration dependencies). Now, we used *pipeline with  $II=1$*  in combination with an *Array Partition* directive on the array that created the issue as it is shown in Table 6.37.

CLUSTER 3	LATENCY(cycles)	BRAM	DSP48	FF	LUT
PREVIOUS	8559293	92	84	56821	46400
TOP FUNCTION	6274359	94	109	68199	55417
CHANGE	-2284934	+2	+25	+11378	+9017

Table 6.37: The Pipeline directive that was applied in the Top function of cluster 3 for the best latency optimization (Previous refers to the added directives shown on the table above)

In the last Table 6.38, we see the final difference that the change from  $II=32$  to  $II=1$  in the *pipeline* after an array partition did to the get\_delta\_matrix\_weight1 function.

CLUSTER 3	LATENCY(cycles)	BRAM	DSP48	FF	LUT
PREVIOUS	6274359	94	109	68199	55417
PIPELINE $II=1$	5508096	102	145	70602	61078
CHANGE	-766263	+8	+36	+2403	+5661

Table 6.38: Pipeline with  $II=1$  directive on get\_delta\_matrix\_weight1 function (Previous refers to the added directives shown on the table above), best latency optimization

### Summary for the best latency optimization

The final results for the best optimization of our architecture for the execution times are:

- Cluster 1, before optimization, **239 ms**, after first optimization, **20 ms** and after best, **15.9 ms**.

### 6.3. SECOND ARCHITECTURE

- Cluster 2, before optimization, **78.5 ms**, after first optimization, **17 ms** and after best, **11 ms**.
- Cluster 3, before optimization **560 ms**, after first optimization, **63.2 ms** and after best, **44 ms**.

The latency summary of all the clusters after first optimization is **100.2 ms** and after the best is **70.9 ms**. The throughput of our algorithm is now at maximum and equals latency.

	BRAM	DSP48	FF	LUT
CLUSTER 1	44	368	100060	99242
CLUSTER 2	34	254	64450	66763
CLUSTER 3	102	145	70602	61078
CL1+CL2+CL3	180	767	235112	227083
<i>AVAILABLE</i>	<i>832</i>	<i>768</i>	<i>301440</i>	<i>150720</i>
BALLANCE	652	1	66328	<b>-76363</b>

Table 6.39: The total area results of all clusters for the best latency optimization policy.

CL1+CL2+CL3	BRAM	DSP48	FF	LUT
First	162	204	82212	80819
Second	180	767	235112	227083
CHANGE	+11%	+275%	+186%	+180%

Table 6.40: Test

In [Table 6.39](#) we collected all the area results of the second optimization. We see that our policy while the most time optimum, created more area in LUT units than the available on the Virtex 6 board, which we have as a target device. [Table 6.41](#) contains the results of the execution times for all three architectures about the clustered version of backprop. The column "original" refers to the unclustered version of backprop, with and without optimizations.

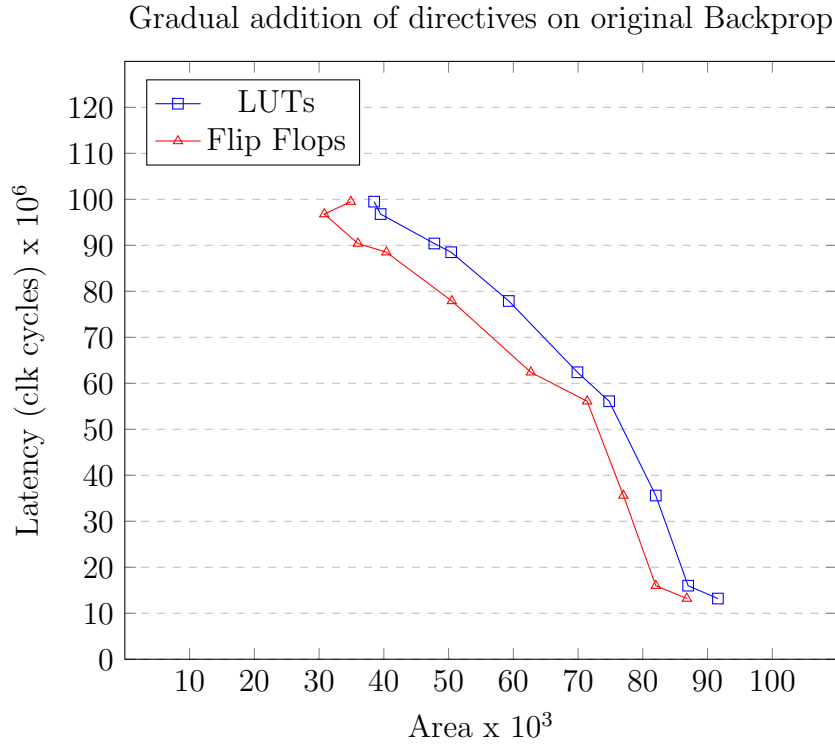
optimization	Cluster 1	Cluster 2	Cluster 3	Cl1+Cl2+Cl3	Original
No	239 ms	78.5 ms	560 ms	877.5 ms	796 ms
FIRST	20 ms	17 ms	63.2 ms	100.2 ms	108.3 ms
SECOND	15.9 ms	11 ms	44 ms	70.9 ms	

Table 6.41: Comparison of all the execution times



## Charts

In this section we will demonstrate via charts how does the design change latency over area and we will explain why we used our different optimization policies for the clustered version of Backpropagation. We use as indication units the number of Flip Flops and LUTs as the other ones, i.e., BRAMs and DSP48Es have small variations and are not representative of the logic.



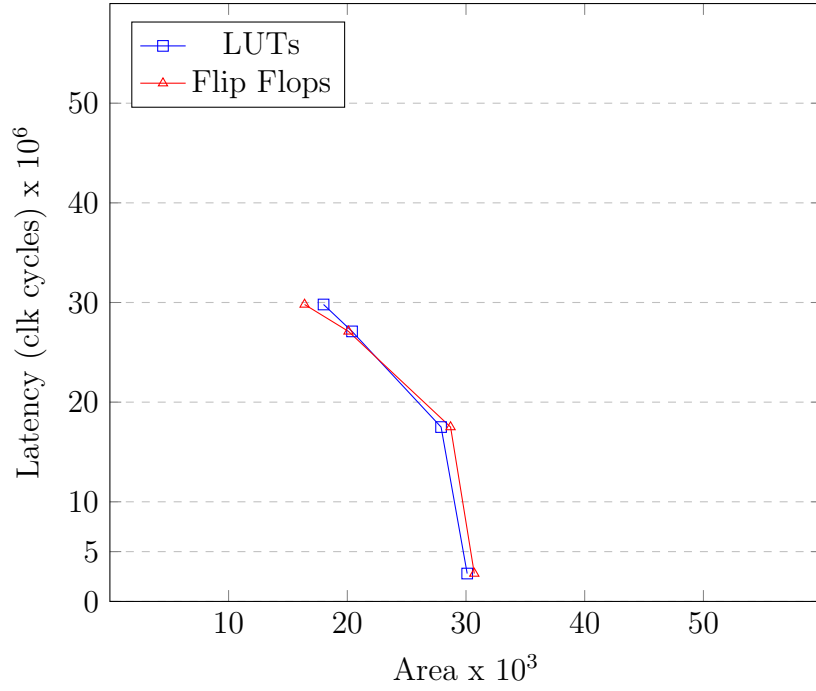
*Each pair of points on the graph represents an addition of a directive on the architecture until the last two pairs, which show the overall optimization that we achieved. The initial decrease on area and as well in latency (at about 95 million clock cycles) is due to the appliance of the *Dataflow* directive. From then on, as we gradually add directives, the area follows a linear increase with respect to the latency decrease until the last pair of points, where is the last directive we added.*

## Cluster 1

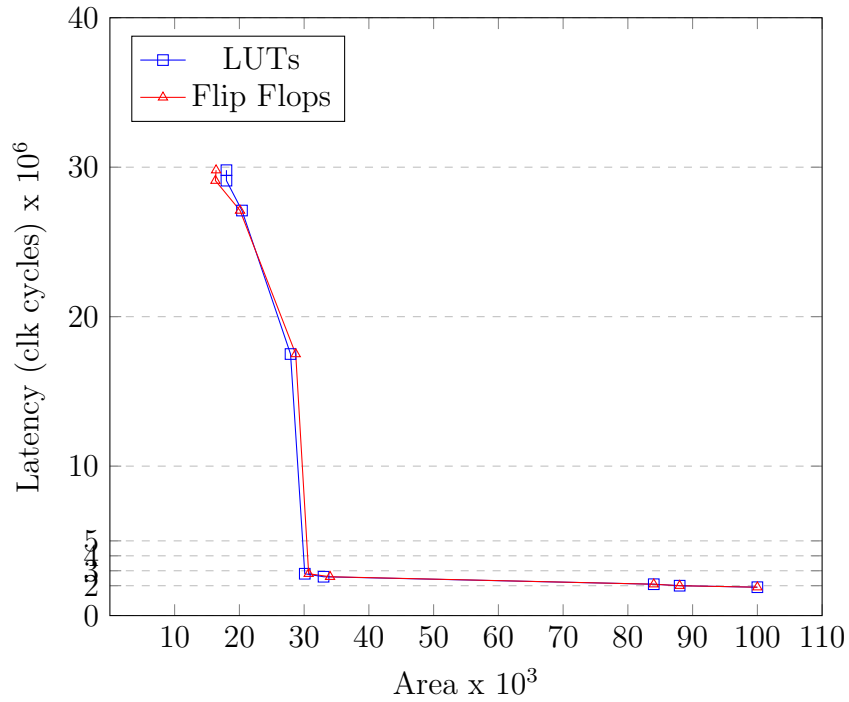
The next two charts are about Cluster 1 and about our policy for the first and best optimization, which we applied on it. In other words, we tried to extract the optimum point for our first optimization policy from the curve that the latency and area correlation follows. *Each pair of points*

represent an addition of directives as we have described in detail in the previous chapters.

Gradual addition of directives on Cluster 1 for the first optimization



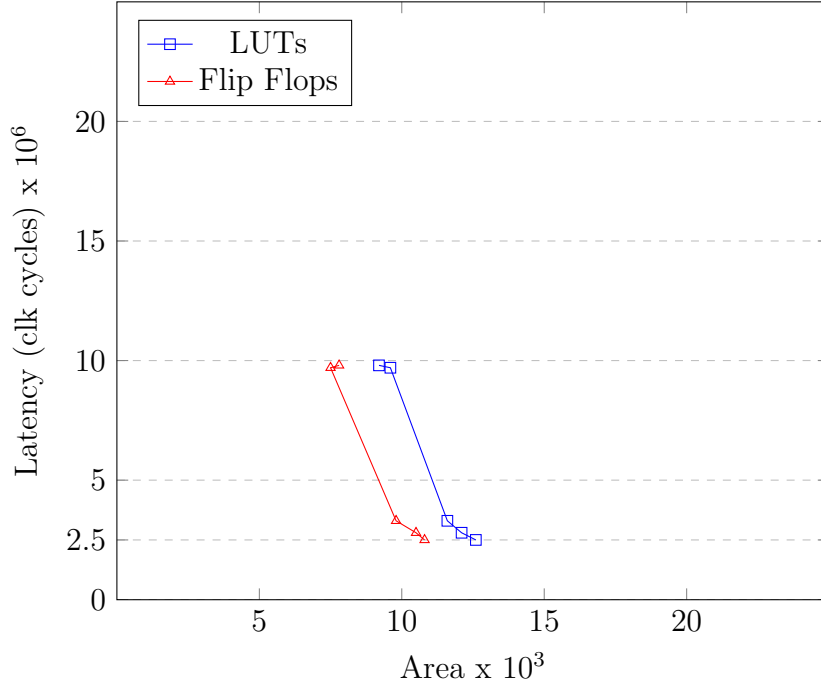
Gradual addition of directives on Cluster 1 for the best optimization



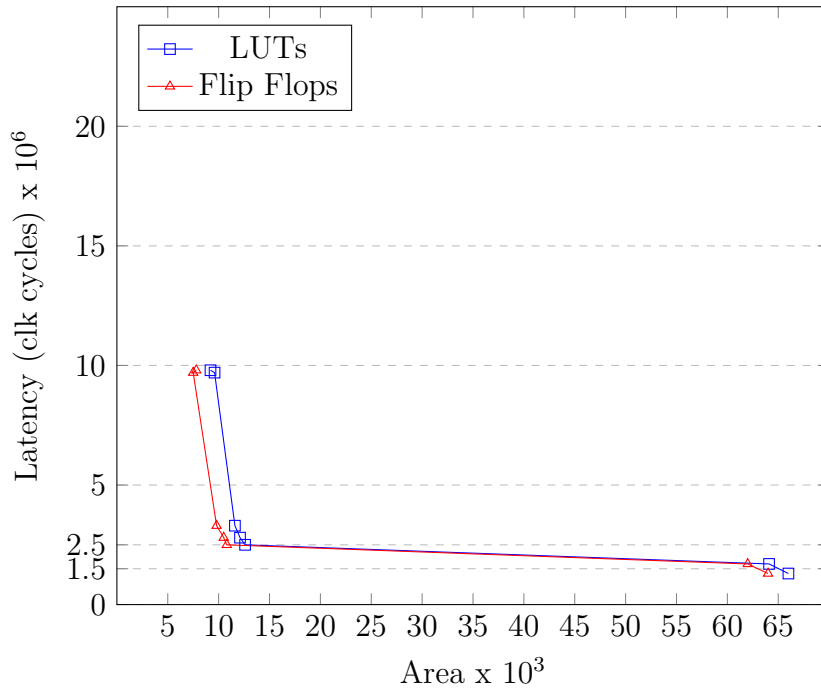
We observe that when the latency is decreased and reaches about 2-3 million clock cycles, the further we proceed in optimizing it, creates much more area overhead so the trade-off is disadvantageous. That's the reason why we stopped our first optimization at this point.

## Cluster 2

Gradual addition of directives on Cluster 2 for the first optimization



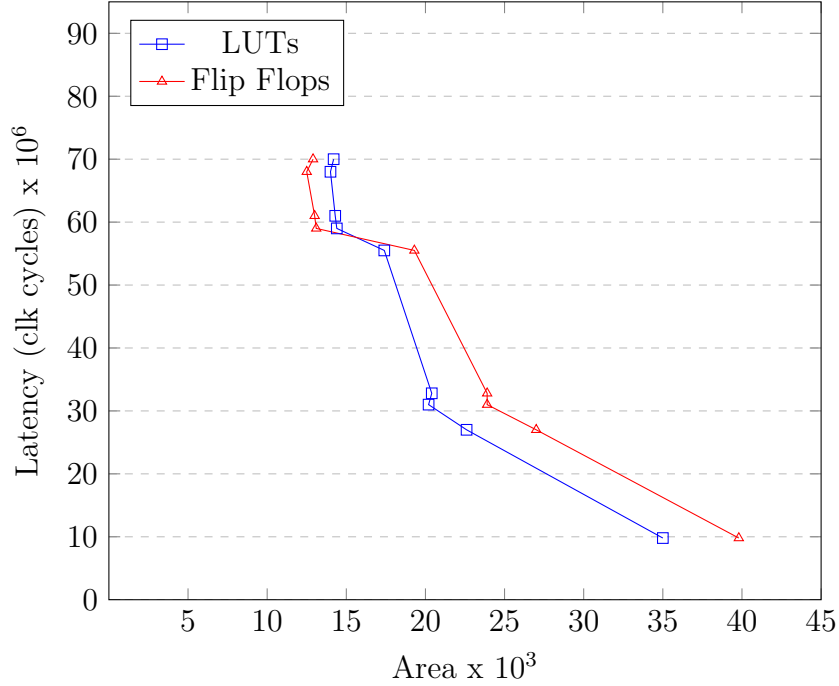
Gradual addition of directives on Cluster 2 for the best optimization



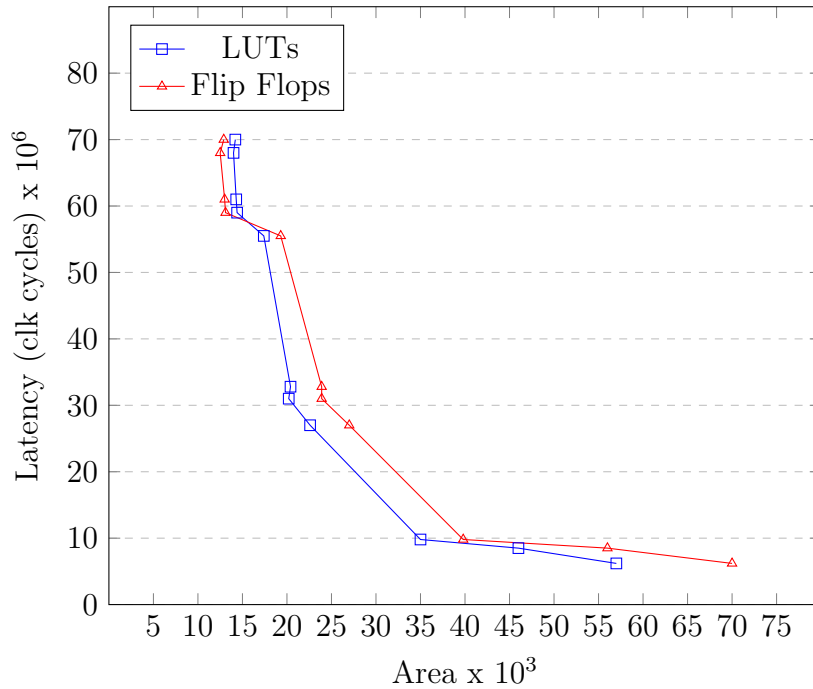
The optimum point, at where the trade-off is the best, is at about 2.5 million clock cycles. From there every additional directive caused a huge area penalty for a small further speed up, that's why we chose that point for our first policy optimum.

### Cluster 3

Gradual addition of directives on Cluster 3 for the first optimization



Gradual addition of directives on Cluster 3 for the best optimization



The optimum point at where the trade-off is the best is at about 10 million clock cycles. We notice that from then on the area and especially the amount of Flip Flops are increased rapidly and so we chose that point for our first policy optimum.

## 6.4 Integration

In this section we will present the integration and testing of our work on RACOS, which is a system developed by a member of our lab [8]. As we described in section 5.4, RACOS gives us the capability to load one or multiple accelerators in an FPGA and schedule them to be used by one or more applications following four policies, Simple, In Order, Out of Order and Forced.

### Single application use

Firstly, we tested the second architecture (clusters) without any optimizations. For the worst case scenario, we measured the total time that is needed for a cluster to be initialized and executed on an empty FPGA. This worst time includes, the time necessary for the clusters, to be transfered on the FPGA, to be loaded in the Partial Reconfiguration Region, the time that it takes for the data to be fetched in the memory of the FPGA and the execution time for the cluster. We did that procedure 10 times for each cluster and the average times are represented in Table 6.42.

	THEORETICAL	INTEGRATED
CL1	239ms	1214ms
CL2	78.5ms	1062ms
CL3	560m	1564ms
Total	877.5	3840ms

Table 6.42: Worst case scenario for each cluster of the not optimized clustered architecture. The theoretical values are calculated on a Virtex 6 FPGA running at 125 MHz with Vivado HLS

We did also a second test trying to get the best time results. For this case we got the same measurements, but for a "Hot" system. This means that we had already loaded the accelerator and the data in the system, so we measured the net execution time. We did that process again for 10 instances for each cluster and we have the results in table Table 6.43.

With our experiments on RACOS we managed, in the best case scenario, to approximate the theoretical results that we got from the hardware simulation of Vivado for the execution times of the clusters architecture of Backprop. The actual system was 62.5 ms slower

	THEORETICAL	INTEGRATED
CL1	239ms	253ms
CL2	78.5ms	103ms
CL3	560s	584ms
Total	877.5ms	940ms

Table 6.43: Best average case scenario for the not optimized clustered architecture. The theoretical values are calculations of the Vivado HLS on a Virtex 6 FPGA running at 125 MHz

than our theoretical times (Table 6.43), fact that is explained by the reconfiguration delay cost of the FPGA.

### Use of Multiple applications

Next, we tested the systems capability for 10 applications (the same application was executed 10 times with different input datasets) running simultaneously and scheduling the accelerators (clustered architecture) according to the four policies. The difference between the policies is that they try to reduce the number of the reconfigurations, so that the latency penalties are reduced. The "Simple" policy has the most reconfigurations and "Forced" the less. Each application uses all three clusters, i.e. the whole accelerator Table 6.44.

Policy	Execution
Simple	29200ms
In Order	16400ms
Out of Order	8700ms
Forced	5900ms

Table 6.44: Execution times for 10 concurrent applications per policy

We did the same experiment to test the softwares performance. We ran the software of the accelerators on three cores on a general CPU (Intel i7, at 3 GHZ), corresponding to the three PRRs on the FPGA, for the same 10 applications. The results are **339 ms** execution time, which is again, many times faster than the most optimal policy for the hardware. From this experiment we can see the impact that the reconfiguration cost has, considering the difference between the "Simple" and the "Forced" policies. The reconfiguration cost explains also the difference between the software and hardware performances.

### Best latency optimization

Finally, we tested the second architecture with the best latency optimization. We measured, on a hot system, the average results of 10 executions for each cluster on RACOS, and we show in [Table 6.45](#) the best performance that we achieved.

	THEORETICAL	INTEGRATED
CL1	15.9ms	39ms
CL2	11ms	33ms
CL3	44ms	68ms
Total	70.9ms	140ms

Table 6.45: Best case scenario for the best latency optimized clustered architecture. The theoretical values are calculated for a Virtex 6 FPGA running at 125 MHz with Vivado HLS

We measured the time that it takes for the software of this particular architecture to be executed on an Intel i7-950 at 3 GHz. The software completes execution in **93 ms**. From [Table 6.45](#) we can conclude that in theory we managed to exceed software for **22.1ms**, but the actual execution time on the hardware was **47ms** slower. The reconfiguration delay of each cluster on the Partially Reconfigurable Region of the FPGA causes this difference.

### Systems behavior for big input

Backprop and all MachSuite algorithms are designed to have relatively small inputs, e.g, a few KB, namely the size of an average cache memory. The size of Backprops all input matrices is about 65 KB. In this thesis we tested the algorithm with bigger inputs, indicatively in the order of MB, so that we can test the theoretical and actual behavior our our design.

The input dataset is stored in the BRAMs on an FPGA. There is a finite amount of available BRAMs on an FPGA and in our case for the Virtex 6 there are 832 BRAM 18KB. That is about 14,97 MB. When the input dataset for any accelerator is increased, the only quantity that is substantially increased is the amount of BRAMs that are needed to store that data.

1. The next two tables ([Table 6.46](#), [Table 6.47](#) ) contain the results of the two architectures without optimization for the different amounts of input datasets.

## 6.4. INTEGRATION

	CL1	CL2	CL3	CL1+CL2+CL3	No Clusters
65 KB	239 ms	78.5 ms	560 ms	877.5 ms	796 ms
1 MB	2198 ms	1248 ms	6569 ms	10015 ms	44400 ms
2 MB	3748 ms	2341 ms	12526 ms	18615 ms	122700 ms
3 MB	6068 ms	3785 ms	20106 ms	29959 ms	230500 ms
4 MB	7770 ms	4915 ms	26203 ms	38888 ms	342800 ms

Table 6.46: The execution times for the two architectures WITHOUT optimizations(The clusters design has only FIFOs to the inputs and outputs)

	CL1	CL2	CL3	CL1+CL2+CL3	No Clusters
65 KB	4 %	3.3 %	10.3 %	17.6 %	4 %
1 MB	42 %	35.5 %	<b>143 %</b>	<b>220.5 %</b>	41.8 %
2 MB	83.1 %	71.3 %	<b>286.3 %</b>	<b>440.7 %</b>	83.1 %
3 MB	<b>151.9 %</b>	<b>128.6 %</b>	<b>509.3 %</b>	<b>789.8 %</b>	<b>149.9 %</b>
4 MB	<b>163.9 %</b>	<b>140.6 %</b>	<b>570.9 %</b>	<b>875.4 %</b>	<b>163.9 %</b>

Table 6.47: The percent of BRAM utilization for the two architectures WITHOUT optimizations. With BOLD are the cases where the BRAM utilization is overflown as it was calculated for a VIRTEX 6 board that contains 832 BRAMs

We can see from the results that although the second architecture (Clusters) created memory overflow on the FPGA (fact that causes delay because the data must be fetched from the external memory) the original algorithm is significant slower (up to 9 times). This means that the streaming design was significantly faster when tested for bigger inputs.

2. [Table 6.48](#) and [Table 6.49](#) show the same results for the first optimization versions of the two architectures.

	CL1	CL2	CL3	CL1+CL2+CL3	No Clusters
65 KB	20 ms	17 ms	63.2 ms	100.2 ms	108.3 ms
1 MB	171 ms	229 ms	679 ms	1079 ms	1740 ms
2 MB	320 ms	428 ms	1331 ms	2079 ms	4870 ms
3 MB	510 ms	696 ms	2123 ms	3329 ms	9130 ms
4 MB	661 ms	906 ms	2769 ms	4336 ms	13590 ms

Table 6.48: The execution times for the two architectures OPTIMIZED(First optimization)

We can see, firstly, from [Table 6.49](#) that the optimized versions of both architectures created no more BRAM overflow on the



	CL1	CL2	CL3	CL1+CL2+CL3	No Clusters
65 KB	5.2 %	4 %	12.2 %	21.4 %	6.2 %
1 MB	43.7 %	35.8 %	<b>144.9 %</b>	<b>224.4 %</b>	44.4 %
2 MB	84.8 %	71.8 %	<b>287 %</b>	<b>443.6 %</b>	85.3 %
3 MB	<b>153.6 %</b>	<b>129 %</b>	<b>510.5 %</b>	<b>793.1 %</b>	<b>153.8 %</b>
4 MB	<b>165.6 %</b>	<b>141.1 %</b>	<b>572.1 %</b>	<b>878.8 %</b>	<b>166.1 %</b>

Table 6.49: The percent of BRAM utilization for the two architectures OPTIMIZED (First optimization). With BOLD are the cases where the BRAM utilization is overflown as it was calculated for a VIRTEX 6 board that contains 832 BRAMs

FPGA than the unoptimized ones. Secondly, with the applied optimizations the difference between clustered and original design was drastically reduced but the clustered design is still up to 3 times faster(4MB).

3. We calculated the software execution times for the same accelerators with the same input datasets (1MB,2MB,3MB,4MB), running on an Intel i7-950 at 3 GHz (Table 6.50). These results are very close to the most optimized hardware architecture as we can conclude from Table 6.48 but still software is up to 2.7 times faster.

	Software	Hardware	Difference
65 KB	93 ms	70.9 ms	- 23%
1 MB	393.7 ms	1079 ms	+ 174%
2 MB	1136.3 ms	2079 ms	+ 83%
3 MB	2113.8 ms	3329 ms	+ 57.5%
4 MB	3196.7 ms	4336 ms	+ 35.5%

Table 6.50: Software runtime for CL1+CL2+CL3 compared with the equivalent of the THEORETICAL hardware runtime Optimized (Best optimization)

## 6.5 Result Comparison and analysis

In this section we will describe graphically the results of our work and provide our conclusions.

### Area comparison

We begin showing in [Table 6.51](#) the summary for the area utilization of each architecture.

Backprop	BRAM	DPS48E	FF	LUT
First HW Architecture (No optimization)	34	112	34856	38593
First HW Architecture (Optimization)	52	254	86882	91600
Second HW Architecture (A Approach)	162	204	82212	80819
Second HW Architecture (B Approach)	180	767	235112	227083

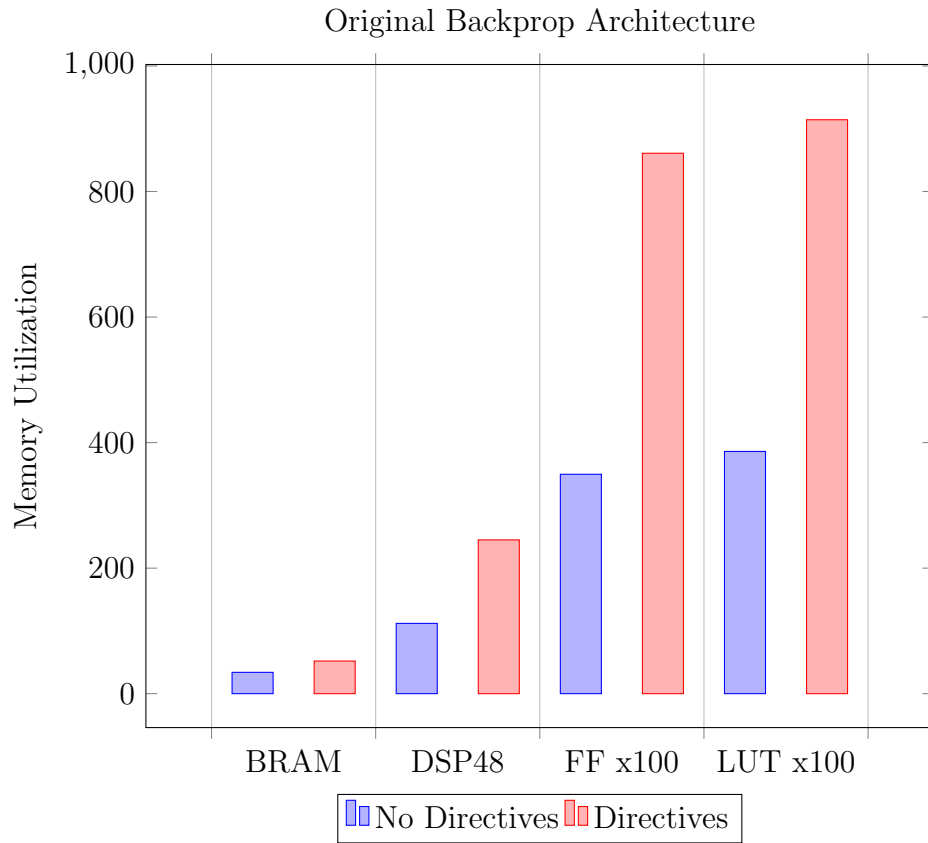
Table 6.51: Summary of area utilization for all architectures.

We conclude from [Table 6.51](#) that:

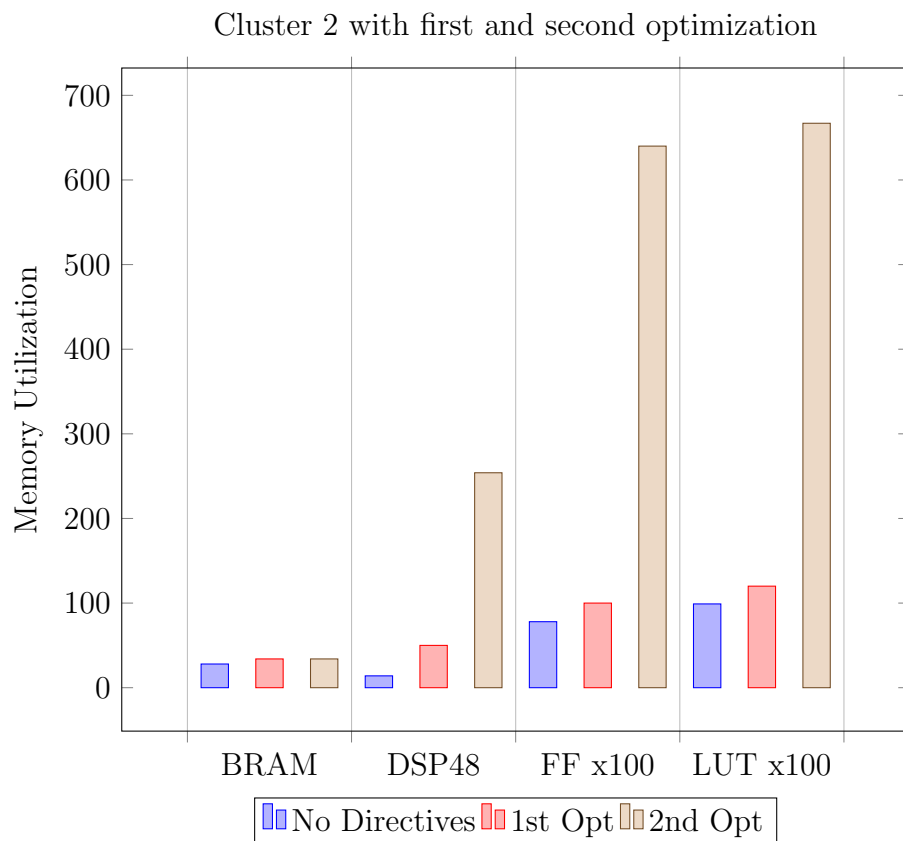
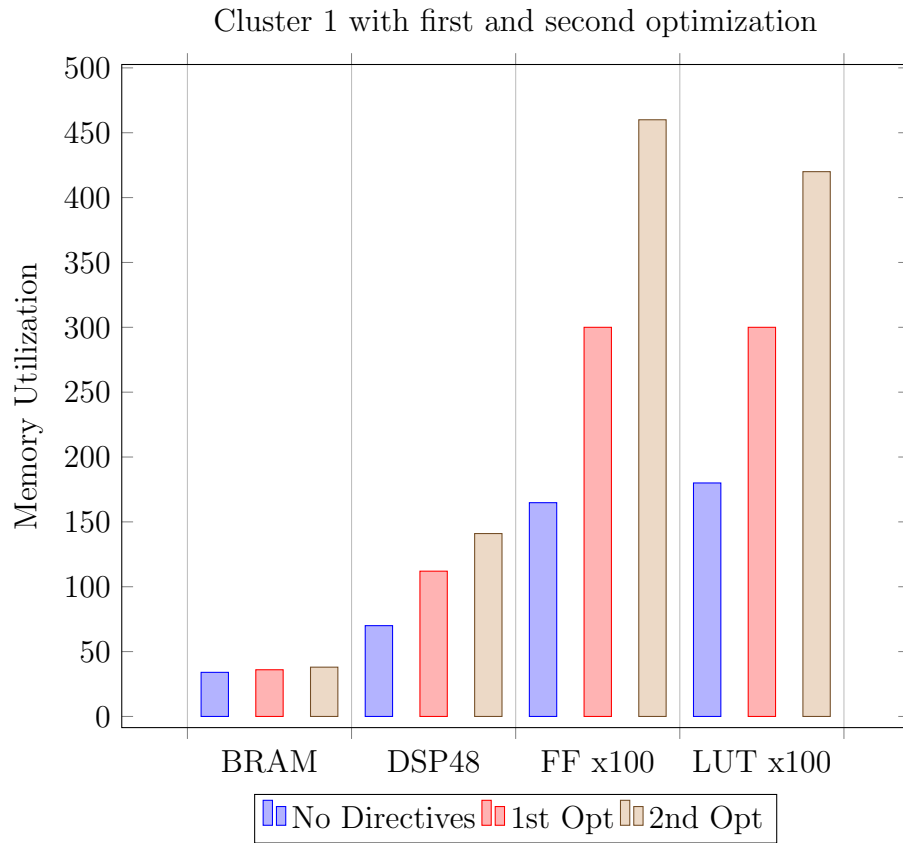
A) The first HW architecture with (Optimization) and the second HW architecture (A Approach) have about the same area utilization except of the BRAMs. The second architecture has way more BRAMs because in the design with the clusters we added arrays in each cluster to store the streamed data from the FIFOs.

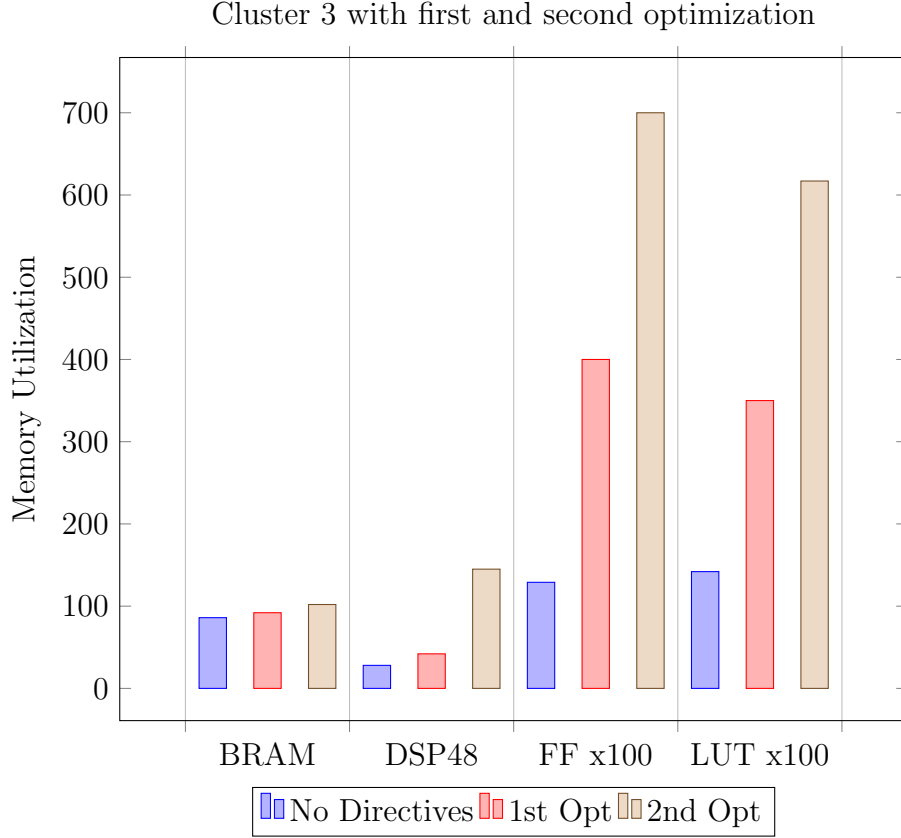
B) Comparing the second architecture A and B approaches we see that the B approach utilized a massive amount of resources to achieve the small speed up that it gained compared with the A approach.

Next, we show the area utilization with charts. The first chart is about the memory utilization of the original algorithm for all available memory units. The FF and LUT quantities are represented two orders of magnitude lower so that the other quantities are visible.



The next three charts have all the results of the memory utilization about the Clusters. We present the area variations for the design without directives and the design of the first and the second optimization.





### Latency results and comparison

Starting with the optimizations of the original algorithm and the second architecture (the clusters) we have:

1. The original Backprop became **86.3%** faster with the addition of directives on HLS.
2. The clustered design (*plus FIFOs*) after the appliance of the *same* directives as the original design became **88.6%** faster than the clustered design without optimizations.
3. The clustered design after achieving the best latency with directives, became **91.9%** faster than the clustered design without optimizations.
4. The clustered design, finally, in comparison with the original design after having the same hardware optimizations but with FIFOs that streamed the data gave us [Table 6.52](#):

The clustered design is **8.1 ms or 8%** faster than the original, meaning that our architecture achieved a small percentage of concurrency between clusters via streaming the data.

As long as we have tested our architecture on a real system, it is worthwhile to compare the actual results of the hardware with the software respectively.

Backprop	Time	Speed up
First HW Architecture (No optimization)	796 ms	1
First HW Architecture (Optimization)	108.3 ms	7.3
Second HW Architecture (A Approach)	100.2 ms	7.9
Second HW Architecture (B Approach)	70.9 ms	11.2

Table 6.52: Summary of execution time for all architectures and speed up in account of the first Architecture(No optimization)

For the clustered version (without optimization) we had initially calculated from Vivado HLS the execution time at **877.5ms** which is our theoretical base. When tested on RACOS we got execution time at **940ms** (Table 6.43). Those results are pretty close, having the real system **62.5ms** slower. The reconfiguration overhead that needs to be paid on the real system, justifies the latency difference.

Our second experiment, that included multiple runs of the accelerator for ten applications, aimed to demonstrate, the impact of the reconfiguration cost to the total performance. So, the first policy, i.e., the "simple" one, where the system has to pay the reconfiguration cost EVERY time required **29200ms** to finish, in comparison with the last policy, the "forced" one, where the system pays that cost only once (one time for each cluster) needed **5900ms** to complete. We notice that because the net execution time of the accelerator is generally small(a few hundred ms) the partial reconfiguration overhead, while small, has a noticeable impact. However, the software execution time of the same experiment on the i7 processor was **339 ms**, which is 17 times faster.

Finally, the best theoretical time that we managed to achieve for the three clusters together, is a total **70.9 ms**, from **877.5 ms** which was the execution time before the optimization. The results for the same architecture on RACOS were **140 ms** execution time. Meanwhile, the software execution time that we had initially calculated for the three clusters is **93ms**. So, comparing this software time with the best theoretical result, we see that theoretically we managed to exceed software in speed for **22.1 ms**. However, the actual execution time on a physical system is **47 ms** slower than the software.

A final useful calculation and comparison would be the estimation of the energy consumption for the two cases. A) The energy consumption of our 2nd architecture as it was executed on the real system on the FPGA. B) The analogous energy consumption of the 2nd architecture on software.

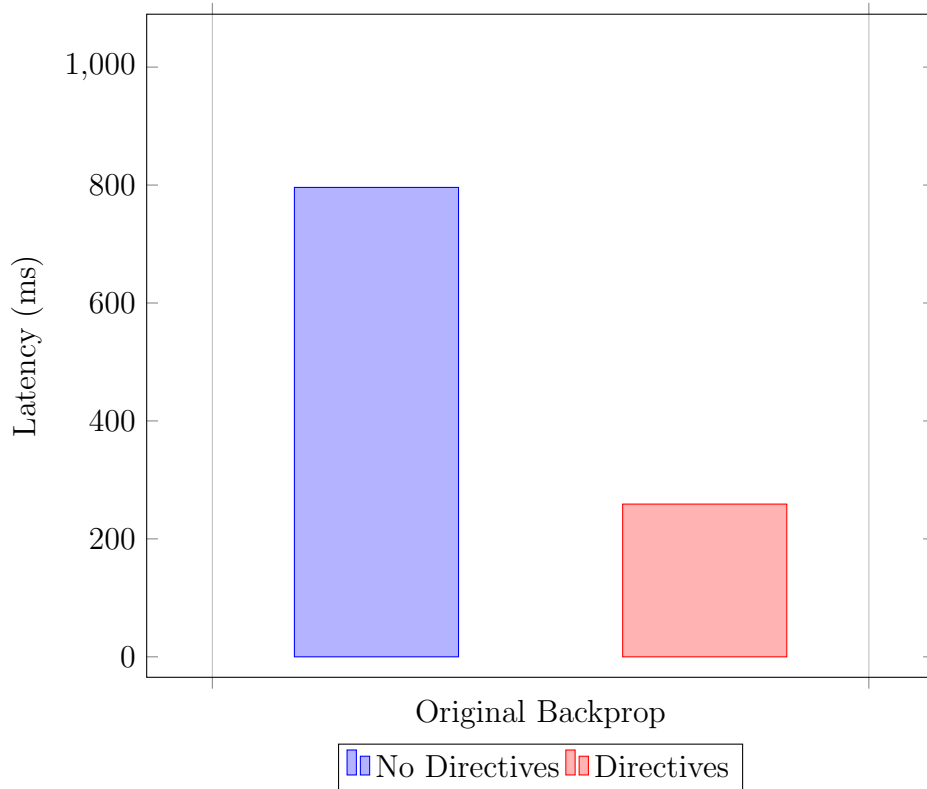
Dataset	Software	1st Archit	2nd Archit (A)	2nd Archit (B)
	Time	Time	Time	Time
65KB	93 ms	108.3 ms	100.2 ms	70.9 ms
1MB	393.7 ms	1740 ms	1521 ms	1079 ms
2MB	1136.3 ms	4870 ms	3256 ms	2079 ms
3MB	2113.8 ms	9130 ms	4960 ms	3329 ms
4MB	3196.7 ms	13590 ms	6936 ms	4336 ms

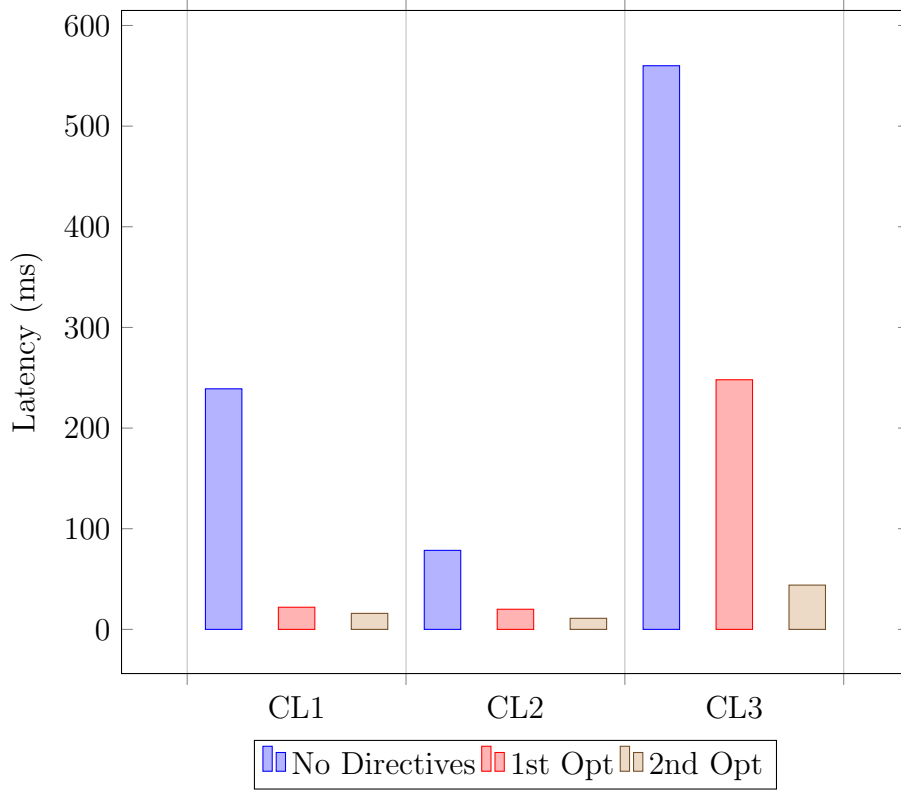
Table 6.53: Summary of all times for all Architectures and the software run for each input dataset. (A, B refers to the Approaches of the Second Architecture )

Given that the average energy consumption of the i7 CPU is about 130 W and the average energy consumption of the FPGA is about 9.5 W, we can calculate the energy performance of our architecture on Joules (W\*s).

	Energy consumption
Hardware	0.00037 J
Software	0.00335 J
Difference	x 9

Table 6.54: Estimated energy consumption of the 2nd architecture on hardware and software





### Conclusions on the use of directives

Finally, some conclusions about the choices we made for the specific directives that we used.

- **Dataflow**

The Dataflow directive is for optimizing the overall throughput. It ensures that every function and every loop will start operating as soon as the data is available (task-level pipeline). It adds memory channels between tasks in order to keep the data "flowing". In our case of the second architecture these memory channels were set to FIFO interfaces. The overall effect of this directive in our algorithm was to make changes all over the design, gaining some percent of area resources and a small amount of latency improve. The throughput of our algorithm was already at maximum and equal the latency due to default settings of the Vivado tool that exploits concurrency where ever possible.

- **FIFO**

The FIFO interface directive specifies the inputs and outputs of the clusters to be implemented as a FIFO in order to stream data during the dataflow optimization. This directive was applied only to the clustered architecture in order to stream data between the clusters. This addition caused a small latency speed up and a small area decrease due to the replacement of some registers and



	Latency	area			
Dataflow		BRAMs	DSP48	FF	LUT
Original	-0.06%	-12.5%	-	-12.8%	-13.3%
Cl1+Cl2+Cl3	-0.02%	-	-	-1.2%	-1%

Table 6.55: The Dataflow appliance on the original Backprop and on the Clustered architecture

arrays with FIFO structures. The overall performance improve that caused on the clustered architecture was 8% , i.e. 8ms to the total timing.

	Latency	area			
FIFO		BRAMs	DSP48	FF	LUT
Cl1+Cl2+Cl3	-2%	-	-	-0.8%	-1.4%

Table 6.56: The FIFO directives on the Clustered architecture

- **Pipeline**

The Pipeline directive improves throughput by allowing the concurrent execution of operations within a loop or function. When applied on a function it unrolls all loops in the hierarchy causing a drastic latency improve but for the same reason it causes a big increase in the area that is required. Our algorithm consists mainly of for-loops so we used this directive extensively. When trying to pipeline a loop some issues may appear. *Bottlenecks* may appear when many memory accesses are occurring at the same time on a Block RAM that has limited ports (2-port BRAMs). This issue is resolved with the addition of an Array Partition directive, but in some cases this solution was not efficient because this directive created too much area utilization penalty. Array partition was used mainly in our second policy for the best latency optimization. An other issue that may appear is a *carried dependency constrain* between loop iterations. In this case an operation on a loop must finish before the next iteration starts. A solution to this would be to create multiple instances of this operation and apply array partition on the operants, but with huge resources penalty. Both issues are resolved with efficacy by increasing the Initiation Interval (II) in the pipeline directive (loosing a few clock cycles). The run time of a pipelined loop is  $(\text{Loop trip count} - 1) * \text{II} + \text{Depth}$ . (Depth= number of cycles needed to complete one iteration)

- **Loop unrolling** This directive unrolls loops to create multiple independent operations rather than a single collection of operations. This causes the area utilization to increase because multiple copies of the logic in a loop are required. This directive also

## 6.5. RESULT COMPARISON AND ANALYSIS

	<b>Latency</b>	<b>area</b>			
Pipeline		BRAMs	DSP48	FF	LUT
Original	-77.5%	+17.6%	+114%	+142%	+113%
Cl1+Cl2+Cl3	-85.5%	+9.5%	+57%	+121%	+76%

Table 6.57: The total optimization of all the pipeline directives that were applied on the original Backprop and on the Clustered architecture

usually requires array partition. As we stated above the pipeline directive unrolls fully a loop. In some cases though, a full unrolled loop isn't the most efficient choice, due to the trade-off between latency and area. In our algorithm some functions, that are called many times, were partially unrolled (with factors 2 and 4) in order to achieve the best trade -off between area and latency.

	<b>Latency</b>	<b>area</b>			
Loop Unroll		BRAMs	DSP48	FF	LUT
Original	-11.1%	+35.2%	+25%	+17.7%	+34.3%
Cl1+Cl2+Cl3	-2.4%	-	-	+1.5%	+11.6%

Table 6.58: The total optimization of all the partially unroll directives that were applied on the original Backprop and on the Clustered architecture

# Chapter 7

## Conclusions and future work

### Summary

In conclusion of the thesis we can say that we provided a detailed description of all MachSuites accelerators. More specifically, we elaborated what the Backprop algorithm is, how it functions and why we chose it for our thesis.

Secondly, we created an analysis of the memory requirements for every MachSuite benchmarks algorithm. We demonstrated also, what is, for a specific target board, the actual area utilization of every accelerator in detail.

Afterwards, we analyzed the architecture of the original Backprop accelerator and we described the way the HLS tool maps the commands in C language into hardware, deploying the four available resource units (BRAM18K, DSP48E, Flip Flops and Look Up Tables). We also introduced a different architecture for this current algorithm, where we split the algorithm into three parts, that improves its efficiency and exploits parallel execution.

We used the HLS tool to apply latency optimization directives on both original and second architectures, achieving 86.3% speed up for the original architecture. We described in details the gradual latency improve contribution of each directive for both architectures.

For the second architecture we explored furthermore the optimization capabilities pointing out two baselines. First, we suggested an optimization technique that aimed for the best trade-off between latency and area. This first technique produced a total 88.6% speed up for the second architecture. Second, the best optimization technique aimed for the most optimal latency improve of the second architecture, regardless the resources utilization and was made in order to test the capabilities of the hardware in comparison with the software. This technique gave us a 91.9% speed up which is the best one we can achieve ([Table 6.58](#)).

Our clustered architecture aimed on creating a "pipelined" version of the Backprop algorithm that streams the data. This version managed to be 8% faster than the original Backprop and couldn't be more optimal due to the nature of algorithm that has many data dependencies.

Finally, we tested our most latency optimal design of the second architecture (Clusters) of Backprop algorithm both on a hardware

---

simulator (Vivado HLS) and on a real hardware system (RACOS) and found that the performance on the real system was almost 2 times slower than the simulation.

### **Future Work**

This thesis can be used for possible future extensions including:

- The memory analysis of MachSuite that we provided can lead to further research on the other accelerators included in the benchmark. Also, the second architecture for Backprop algorithm that we introduced can help others to apply our technique on other applications on FPGAs.
- The optimized Backprop algorithm that we provided can be used unaltered to optimize applications on neural networks. In addition, the results of our latency optimization over Backprop algorithm can be generalized and be used on other hardware accelerators.
- Some of the issues and weaknesses we faced in this thesis due to limitations of the board that we used can be further analyzed and solved using a more modern FPGA board.

# Appendix A

Here some helpful figures.

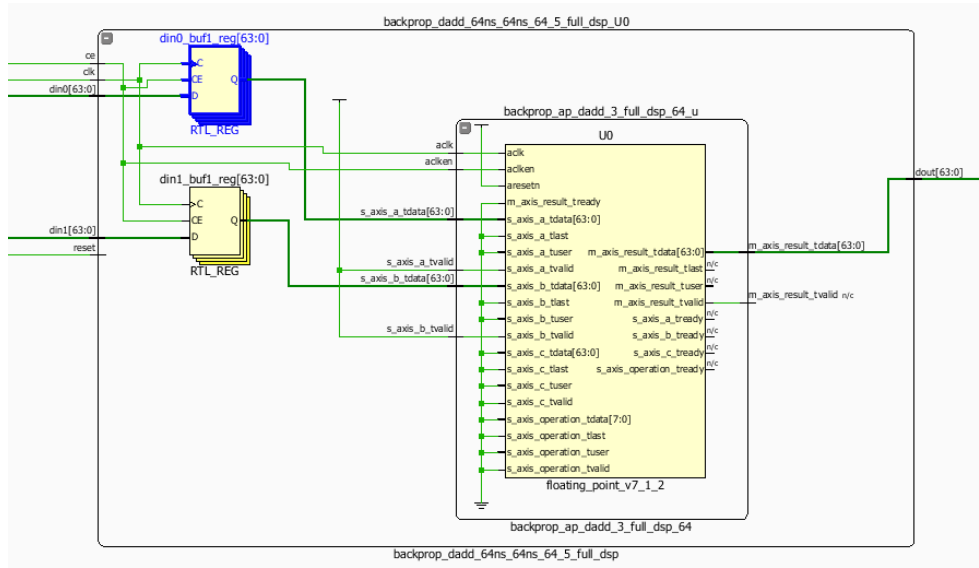


Figure A.1: This fixed module does an addition between two 64 bit operators and is an essential part of Backrops architecture

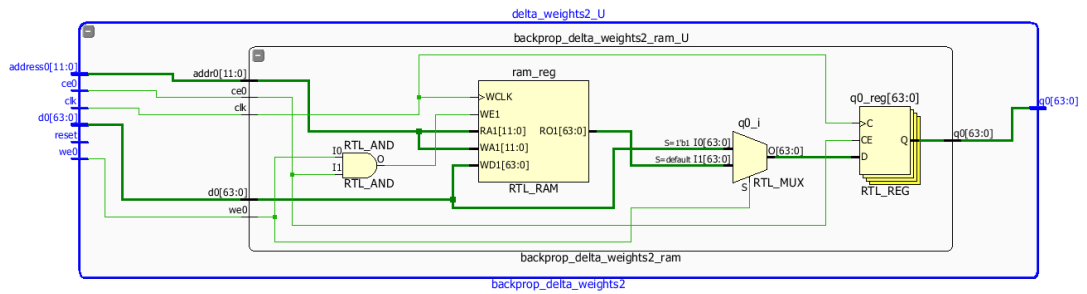


Figure A.2: The design of a Block RAM as it is implemented by Vivado HLS tool. It is a simple straightforward design, with the main body (*ram\_reg*) and some modules (an AND module and a Mux) to regulate the write enable and read from the RAM and some registers to store the data before the RAM output.

---

# Bibliography

- [1] Browse the Harwell-Boeing Collection. URL <http://math.nist.gov/MatrixMarket/data/Harwell-Boeing/>.
- [2] Force field(chemistry). URL [https://en.wikipedia.org/wiki/Force\\_field\\_\(chemistry\)](https://en.wikipedia.org/wiki/Force_field_(chemistry)).
- [3] LAMMPS Molecular Dynamics Simulator. URL <http://lammps.sandia.gov/>.
- [4] unrolling. URL [https://en.wikipedia.org/wiki/Loop\\_unrolling](https://en.wikipedia.org/wiki/Loop_unrolling).
- [5] Virtex-6 FPGA Memory Resources. URL [https://www.xilinx.com/support/documentation/user\\_guides/ug363.pdf](https://www.xilinx.com/support/documentation/user_guides/ug363.pdf).
- [6] G. L. Bin Lin. Application of back-propagation artificial neural network and curve estimation in pharmacokinetics of losartan in rabbit. 2015. URL <https://www.ncbi.nlm.nih.gov/pmc/articles/PMC4729999/>.
- [7] D. Chakrabarti, Y. Zhan, and C. Faloutsos. R-MAT: A Recursive Model for Graph Mining. 2004. URL <http://www.cs.cmu.edu/~christos/PUBLICATIONS/siam04.pdf>.
- [8] Charalampos Vatsolakis. A user-transparent system for virtualizing reconfigurable hardware accelerators, 2015. URL <http://dias.library.tuc.gr/view/62753>.
- [9] S. A. Fahmy, K. Vipin, and S. Shreejith. Virtualized FPGA accelerators for efficient cloud computing. In *Proceedings - IEEE 7th International Conference on Cloud Computing Technology and Science, CloudCom 2015*, 2016. ISBN 9781467395601. doi: 10.1109/CloudCom.2015.60.
- [10] Y. Guan, Z. Yuan, G. Sun, and J. Cong. FPGA-based Accelerator for Long Short-Term Memory Recurrent Neural Networks.
- [11] C. M. Hengchang Dai. Application of back-propagation neural networks to identification of seismic arrival types. 1996. URL <https://www.sciencedirect.com/science/article/pii/S0031920197000046>.

- [12] J. K. Jaiswal and R. Das. Application of artificial neural networks with backpropagation technique in the financial data. In *IOP Conference Series: Materials Science and Engineering*, 2017. doi: 10.1088/1757-899X/263/4/042139.
- [13] S. A. K. Neaupane. Applications of a backpropagation neural network in geo-engineering. 2003. URL <https://link.springer.com/article/10.1007/s00254-003-0912-0>.
- [14] C. Kachris and D. Soudris. A Survey on Reconfigurable Accelerators for Cloud Computing.
- [15] Mazur Matt. A Step by Step Backpropagation Example, 2015. URL <https://mattmazur.com/2015/03/17/a-step-by-step-backpropagation-example/>.
- [16] D. Pratiwi, D. D. Santika, and B. Pardamean. An Application Of Backpropagation Artificial Neural Network Method for Measuring The Severity of Osteoarthritis. *International Journal of Engineering & Technology IJET-IJENS IJENS I J E N S*, 11(03), 2011.
- [17] B. Reagen, R. Adolf, Y. S. Shao, G.-Y. Wei, and D. Brooks. MachSuite: Benchmarks for Accelerator Design and Customized Architectures. 2014. URL <http://www.eecs.harvard.edu/~shao/papers/machsuite.pdf>.
- [18] Rojas R. The Backpropagation Algorithm. In *Neural Networks*, chapter 7. 1996. URL <https://page.mi.fu-berlin.de/rojas/neural/chapter/K7.pdf>.
- [19] M. V. Ryan. FPGA Hardware Accelerators - Case Study on Design Methodologies and Trade-Offs. URL <http://scholarworks.rit.edu/theses/959/>.
- [20] C. Stergiou and D. Siganos. NEURAL NETWORKS. URL [https://www.doc.ic.ac.uk/~nd/surprise\\_96/journal/vol4/cs11/report.html#WhatisaNeuralNetwork](https://www.doc.ic.ac.uk/~nd/surprise_96/journal/vol4/cs11/report.html#WhatisaNeuralNetwork).
- [21] V. Tsoutsouras, K. Koliogeorgi, S. Xydis, and D. Soudris. HLS code transformation strategies and directives exploration for FPGA accelerated ECG analysis.
- [22] C. Vatsolakis and D. Pnevmatikatos. RACOS: Transparent Access and Virtualization of Reconfigurable Hardware Accelerators.
- [23] N.-Y. Xu, J. Yan, R. Gao, X. Cai, Z. Xia, and F.-H. Hsu. FPGA-based Accelerators for ” Learning to Rank ” in Web Search Engines.



- [24] C. Zhang, P. Li, G. Sun, Y. Guan, B. Xiao, and J. Cong. Optimizing FPGA-based Accelerator Design for Deep Convolutional Neural Networks. In *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays - FPGA '15*, 2015. ISBN 9781450333153. doi: 10.1145/2684746.2689060.