Politecnico di Torino

DET - Department of Electronics and Telecommunications

Implementation of a Convolutional Neural Network Algorithm on FPGA Using High-Level Synthesis

Master's degree in Electronic Engineering

Supervisors:
Prof. Luciano Lavagno
Prof. Mihai Lazarescu

Candidate:
Rafael Campagnoli

Academic Year 2020/2021

Acknowledgments

I would like to thank professors Luciano Lavagno and Mihai Lazarescu for the full availability and support during all steps of the composition of this work. In difficult times like this I couldn't ask for more considerate and helpful advisors.

I would like to thank my parents, Noris and Fernando and my siblings, Pedro, Thais and Leticia, for all the love and support during these years, specially this last one, so that I could achieve the dream of getting my master's degree at Politecnico di Torino.

I would like to thank Luigi for the partnership in most of the projects throughout the course. Your friendship made me understand that two heads think better then two, and working in group can be the best way to resolve a difficult problem.

I would like to thank my *coinquilini* in Via Juvarra for making me fell at home in a foreign country.

These acknowledgments extends to everyone who was a part of this journey. I am very grateful to all for believing in me.

I dedicate the merit of this work to the benefit of all beings.

Abstract

The advancement of silicon technology is revolutionizing the world in terms of processing power. Algorithms and complex mathematical models that require much computing have been made feasible with ease in the last decade. One of such booming algorithms is the Convolutional Neural Network (CNN), which can make very complex predictions. However, with high computing complexity comes the drawback of high power consumption for processing, which is a significant concern for some applications.

This thesis analyzes the performance of a Field Programmable Gate Array (FPGA) implementation of a CNN algorithm used to predict the location of people indoors using infrared sensors data, an application that benefits from the high prediction power of the CNN but requires a low power implementation. The CNN was trained before using the Keras tool written in Python. In this work's contribution, the Keras model was translated into C++ code, then, the code was synthesized into Hardware Description Language (HDL) using High-Level Synthesis (HLS) tools, Vitis HLS and Vivado, and was finally implemented and simulated on several Xilinx FPGA chips. Furthermore, the HLS tools were used to explore the design space to optimize the design for cost, power and resources.

The design space exploration was performed in terms of processing parallelism, FPGA technology, and data type. A solution for the most sequential circuit was synthesized in parallelism exploration, with the minimum parallelism possible. The degree of parallelism was progressively increased with different solutions up to the highest degree of parallelism possible. The latency, power, and total energy solution of each solution were then evaluated and compared. Furthermore, the solution with the lowest energy consumption found was implemented in an FPGA chip of a more advanced technology that could run the algorithm at a higher frequency, assuming that a faster circuit would require lower energy, as the iteration time would be smaller. Lastly, this previous solution was implemented also

with a fixed-point data type, which loses computation precision but reduces the circuit complexity. It was observed that the usage of the fixed point implementation optimize energy consumption of the design by reducing the area and latency for calculation.

The hardware design efficiency was also compared with the implementation of the algorithm in software by running the algorithm and measuring the execution time on two different STM32 microcontrollers, one with a Floating Point Unit (FPU) and one without. The energy consumption of the software implementation was estimated by measuring the execution time and the active mode current consumption of the microcontrollers. As a result of this implementation, it was observed that modern microcontrollers with dedicated numeric computing units such as **STM32-L412** can perform very well, especially in terms of cost.

The analysis concluded that indeed the FPGAs can provide low-power solutions for the implementation of CNNs. Using higher degrees of parallelism for implementing the algorithm in an FPGA can drastically reduce the computation latency, which reduces the total energy consumption for calculation, but up to a certain point. A highly parallelized circuit can require more FPGA resources that consume current but may not necessarily speed up the processing. Higher parallelism require more resourceful FPGAs, increasing the implementation cost.

Contents

List of Figures

List of Tables

Acronyms

AI Artificial Intelligence.

API Application Programming Interface.

ASIC Application Specific Integrated Circuit.

CLBs Configurable Logic Blocks.

CNN Convolutional Neural Network.

CSV Comma-Separated Values.

DSP Digital Signal Processor.

FPGA Field Programmable Gate Array.

FPU Floating Point Unit.

GPU Graphical Processing Unit.

HDL Hardware Description Language.

HLS High-Level Synthesis.

II Initiation Interval.

NN Neural Network.

RAM Random Access Memory.

ReLU Rectified Linear Unit.

RTL Register Transfer Level.

CHAPTER 1

Introduction

1.1 Objectives

The main goal of this project was to explore the design space of the FPGA implementation of a convolutional neural network algorithm. The Artificial Intelligence (AI) processes data from infrared sensors to predict the indoor positioning of a person. A design space exploration should be performed to find a solution that best suits the problem in terms of power consumption and cost.

1.2 Previous Work

In previous works, the sensor group and High-Level Synthesis group performed experiments and research on capacitive sensor configuration, data acquiring systems, different Neural Network (NN) architectures, and also the neural networks quantization. The sensor group is highly focused on the whole system exploration. In each stage, some research and development can be performed.

1.3 Thesis Contribution

For this master's thesis, the stage of processing sensor data through a selected AI algorithm was chosen for exploration and optimization. The difficulty introduced by using a neural network to process the noisy infrared sensor data sets a significant drawback in a system that requires a low-power and low-maintenance design. However, the usage of

a dedicated process unit that is also low-power and low-cost can mitigate this drawback, leaving the final system with non-invasive sensors and a low-power controlling unit.

1.4 Thesis Structure

In Chapter 1 - Introduction, the main objectives and motivations of the thesis work are described. It is shown the previous work performed on the field by the Sensor Team Lab and the thesis contribution of the field.

In Chapter 2 - Bibliographic Research, the needed knowledge for understanding the concepts in this research is explained in detail. The references of the research performed during the work development are cited if the reader has a more profound interest in the topics presented.

In Chapter 3 - Implementation, the implementation of the software is described in detail. Firstly, it is explained the extraction of the parameters from a trained Convolutional Neural Network in the Keras environment, which was performed using a specific script. Then, the implementation in C++ of the convolutional neural network is explained, showing how the internal data flow and submodules of the software work.

In Chapter 4 - High-Level Synthesis and Simulations, the workflow of the HLS and its simulations are explained. Firstly, the hardware synthesis hypothesis targeting low power is described, and the proposed hardware synthesis directives are presented. Other comparisons and simulations performed, such as the FPGA vs. Microcontrollers comparison, are introduced.

In Chapter 5 - Results Analysis, the results of the previously described simulations are shown, as well as the analysis of the results achieved

In Chapter 6 - Discussion and Conclusion, the conclusion of the thesis is presented. Showing which previously made hypotheses made were achieved and what could be analyzed from the overall results.

In the Annexes section, the codes and scripts developed are shown in full.

CHAPTER 2

Bibliographic Research

2.1 Indoor person localization

Indoor people positioning can be beneficial for applications that require precise location data. Some examples are an intelligent supermarket that directs the customer precisely to where their desired groceries are at, smart homes, or an elderly care home that surveys the state of the patient 24 hours a day. Some of these applications, however, such as the older people monitoring, require the sensing of the person to be tag-less and to ensure privacy [1].

To achieve these requirements, the technology must be private, precise, and require little or no maintenance, which calls for a solution that involves non-invasive sensors, fast computation, low cost, and low-power consumption [2].

Following the previous work on the realization of this technology in capacitive sensors, this project intended to develop a technology that determines the indoor person localization using infra-red sensors data processed through a Convolutional Neural Network Algorithm that ensures the tag-less and privacy of the monitoring.

To achieve the low power and low-cost requirements, however, different solutions on hardware or software implementations of the CNN algorithm should be explored, which is the primary purpose of this work.

2.2 State of the Art

Recently, lots of articles have been published on the implementation of CNNs in FPGAs. Most of them show that the implementation of these algorithms is much more power-efficient and faster on FPGAs when compared to GPUs. Research also shows that the implementation of CNNs in FPGAs can deliver up to 2 times the energy efficiency of a GPU [3]. FPGAs are an exciting field for exploration due to their parallel architecture [4]. These findings reveal a promising field for applications that require a high processing complexity but require low energy consumption. The advancement of silicon and transistor technology has been lowering the price of chips over the last decades, and the tendency is to turn this technology more accessible, which makes the field of research of FPGAs implementation very promising for the following years.

2.3 Hardware Design

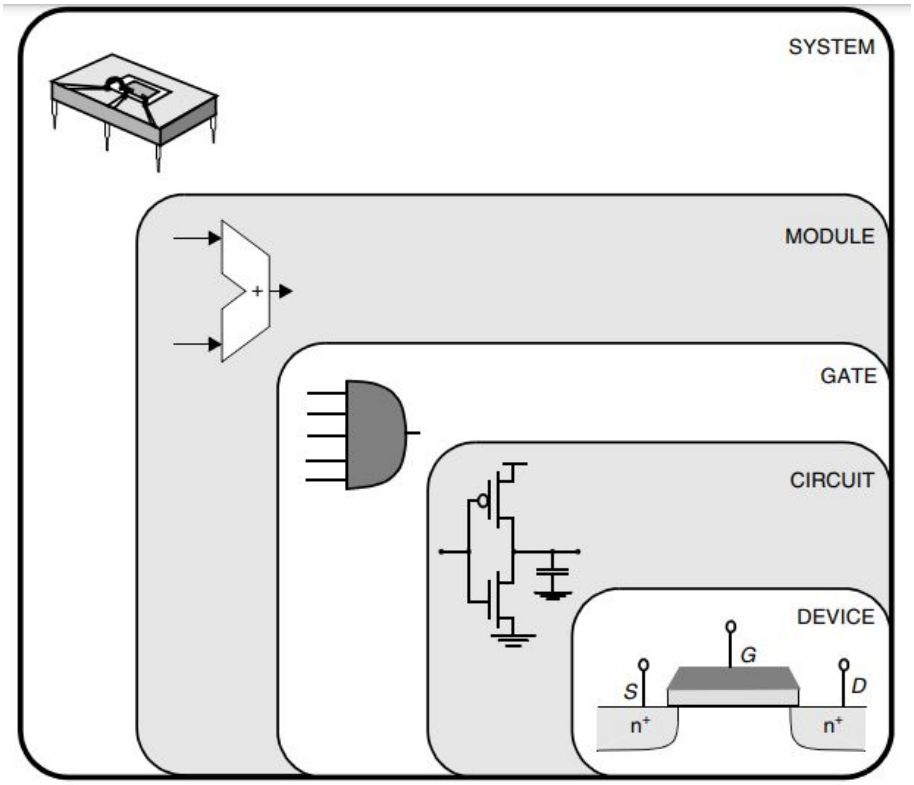

Figure 2.1: Different abstraction levels for electronic circuits design

There are multiple ways of approaching an electronic design problem. Electronic design can also be divided into layers of abstraction, from the minor device scale possible, the organization and placing of transistors, up to the system level, where a complete system is already designed. It must only be programmed to the requiring application [5].

The approach of this work was to perform a hardware design at the abstraction level. The electronic modules were organized in order to implement the specific application.

2.4 Machine Learning

Tom Mitchell defines Machine Learning with *"A computer program is said to learn from experience E with respect to some class of tasks T and performance measure P, if its performance at tasks in T, as measured by P, improves with experience E."* [6]. Machine learning has been used widely for a different range of problems that are too complex for a mathematical model to predict. The act of observing relevant data for inputs and outputs to a system and then learning through statistical modeling the probability of output given the input data set was decisive for the implementations of various applications in the last 20 years, and its demand is increasing. In general, machine learning algorithms can be divided into two groups: unsupervised learning and supervised learning.

In supervised learning, the artificial intelligence is trained by examples of data. A known set of input data paired with a known respective output data set is passed to the AI. The AI "learns" from that set of data, learning the input data patterns that generate the specific outputs. The training is achieved by a process known as backpropagation. In backpropagation, the error of the final output predicted by the artificial intelligence is sent back, and the AI adjusts its internal weights to minimize the error in future predictions. When this process is performed many times, the final prediction error tends to be reduced.

In unsupervised learning, data is presented to the AI without any labels so that the Machine will have to learn by itself. Unsupervised learning can also be called by the clustering algorithm, in which, inside a set of unlabeled data, the algorithm tries to create similar clusters of that data. In this way, when new data arrives at the algorithm, the artificial intelligence decides to which cluster of data this input belongs, based on the previous unsupervised learning.

2.4.1 Neural Networks

Neural networks are an ancient mathematical concept, but today are some of the most famous supervised learning techniques. It is based on the brain, which is a complex composition of neurons. Like that, the neural network algorithm is composed of a network of small mathematical units, the neuron. Each neuron performs a prediction based on its inputs following an activation function, which consists of the neuron's output. The neural network is a composition of layers of neurons. The first layer takes the data inputs and connects them to each neuron of the first layer. The subsequent layers take the outputs of the previous neurons and connect them to their neurons. This step is repeated until the output layer, where the final output for the neural network is calculated.

The neuron, the building block of the neural network, can be described as a computing block the follows (2.1)

$$y = f(\sum_{i=0}^{n}(W_i * X_i) + B) \tag{2.1}$$

Where y is the output vector, n is the number of inputs, f is the activation function, X_i is the input vector W_i is the weights matrix, and B is the bias.

In Figure 2.2, the functional diagram of the neuron structure can be observed. In Figure 2.3, an example of a neural network with the combined neurons can be observed.

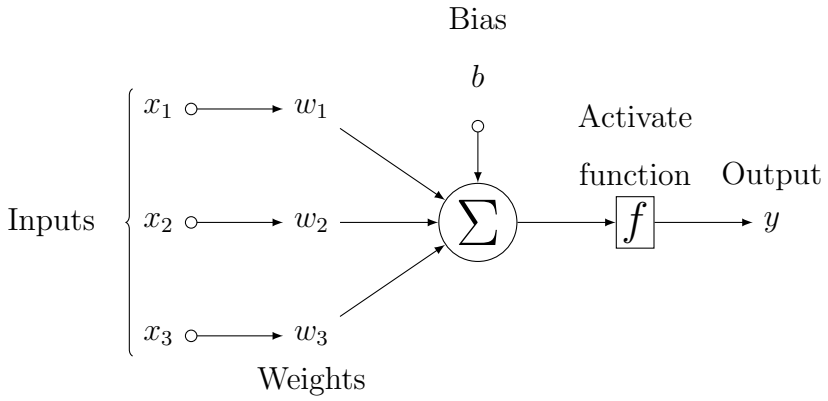

Figure 2.2: Neuron Structure

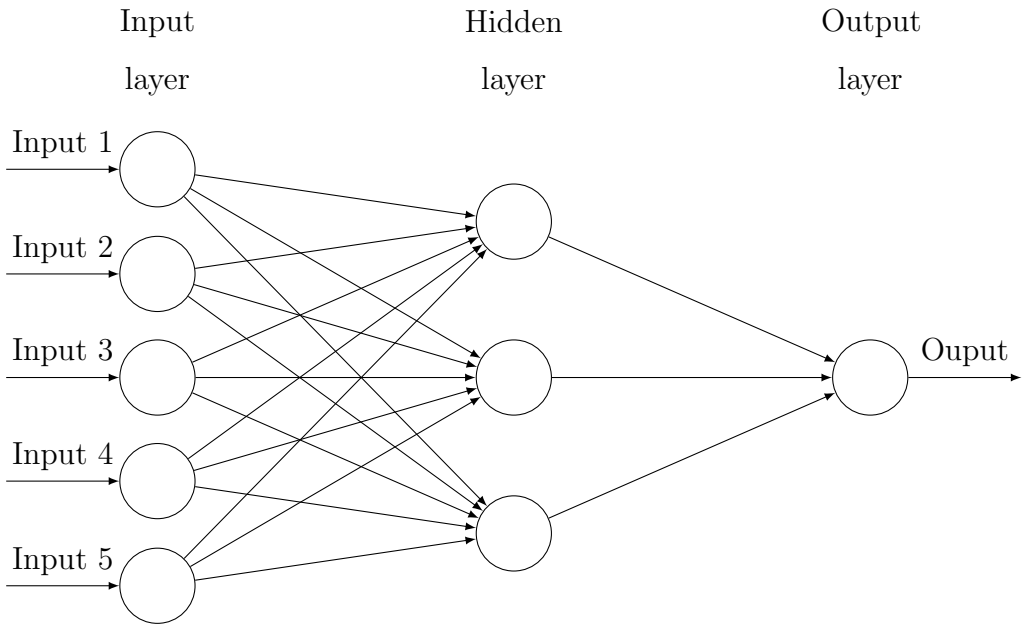

Figure 2.3: Neural Network Example

2.4.2 Convolutional Neural Networks

In the last decade, Convolutional Neural Networks, or Deep Neural Networks, have become the standard for various Computer Vision and Machine Learning operations [7]. solution of the neural networks problem Convolutional Neural Networks are a special kind of neural network that implements the hidden layers with different algorithms. They follow more complex algorithms and promise more complex predictions, also reducing the overall prediction error.

Training a Convolutional Neural Network, however, is very computationally demanding. The number of parameters is high, and therefore the number of weights and biases rise exponentially every time a layer is added to the architecture. The dataset size needed for training is also massive. Therefore, the usage of 2D CNNs was only made possible by the advancement of Graphical Processing Unit (GPU) parallel processing in the last years. Before that, only small-scale CNNs would be implemented. Nowadays, with the possibility of cluster computing, more and more complex artificial intelligence is being trained and implemented to perform complex tasks, such as autonomous driving. Besides the performance levels achieved, CNNs introduce the feature extraction and classification task into a single body [7].

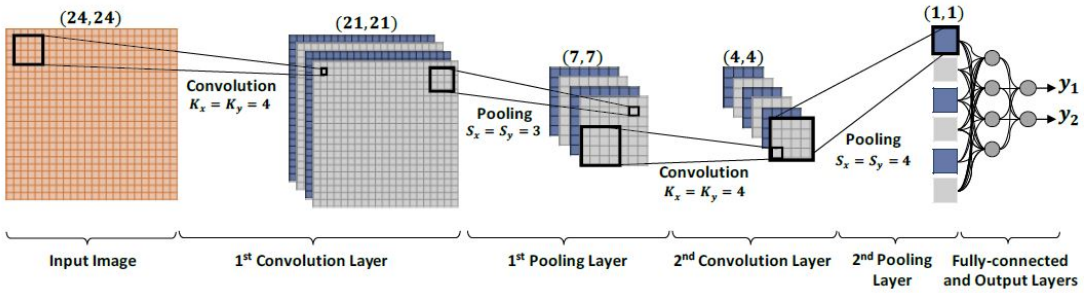

Figure 2.4: Illustration of a generic 2D Convolutional Neural Network

CNN's are usually implemented to process and predict 2D signals, such as images and video frames. The possibility to learn and predict from images has numerous applications, such as face recognition or the complete interpretation of a road (identification of traffic lights and signs). For the processing of 1D signals, such as data streams of single sensors, 1D CNNs were designed from the 2D CNN's perspective but performing the convolution between the filters and the input data in only one dimension. This algorithm has a significantly lower computational complexity, as the operations are simple array operations instead of matrix operations. 1D CNNs incorporate deep neural networks' prediction power but maintain a simple implementation in terms of calculation. These characteristics make 1D CNNs suited for real-time and low-power applications.

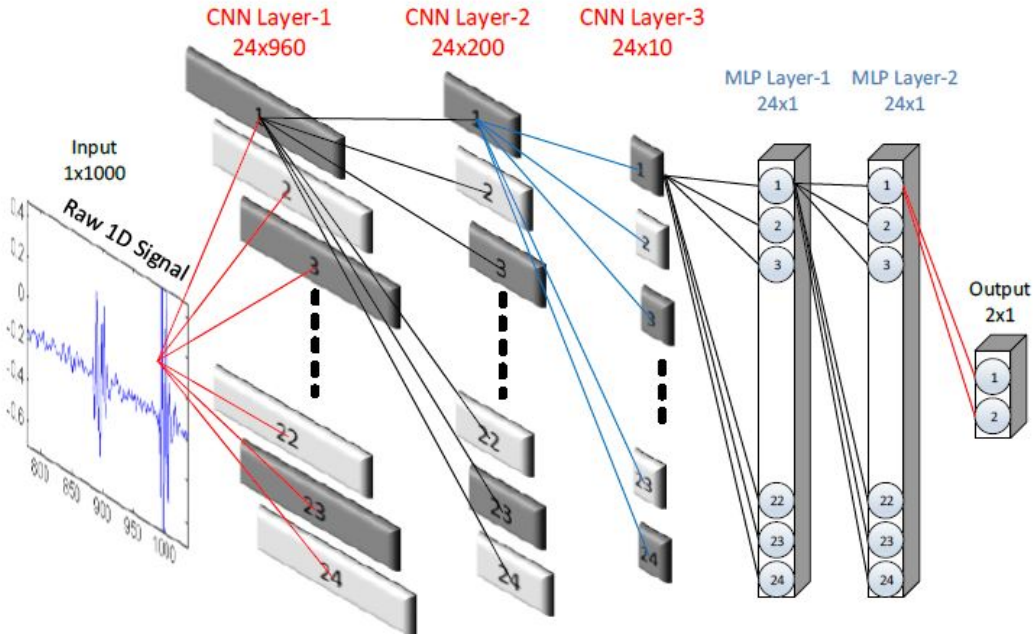

Figure 2.5: Illustration of a generic 1D Convolutional Neural Network

Convolution Layer

The convolution layer performs the deep learning part of the algorithm. With the feature of convoluting the input data into filters, instead of simply multiplying them by weights, the patterns learned by the AI can be more accurate. The price to pay with the addition of a convolution layer is the increasing processing complexity. The convolution layer of the algorithm implemented in this work is the 1D convolution algorithm, illustrated in Figure 2.6 and (2.2) [8].

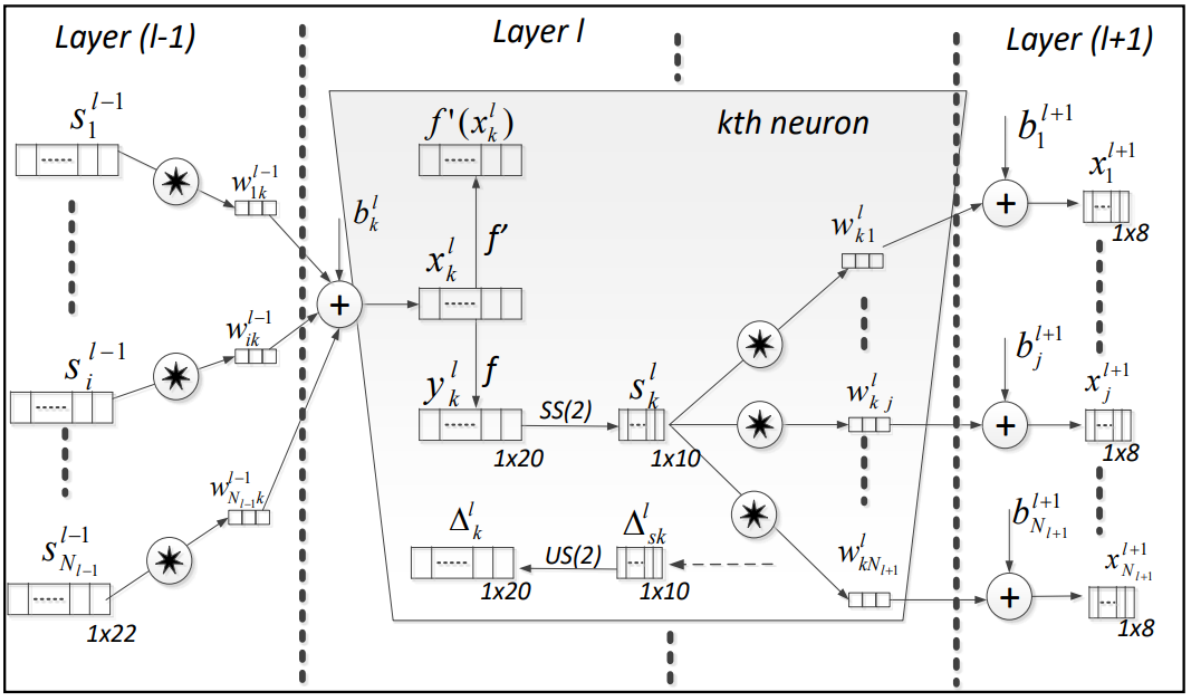

Figure 2.6: Example of 3 subsequent layers of the 1D CNN

$$x_k^l = b_k^l + \sum_{i=1}^{N_{l-1}} \text{conv1D}(w_{ik}^{l-1}, s_i^{l-1}) \tag{2.2}$$

In (2.2), it is described the calculation of the forward propagation from previous layer,$l-1$, to create the input of the kth neuron on the next layer, l. where x_k^l is the input, b_k^l is the bias of the kth neuron at layer l, and s_i^{l-1} is the output if the ith neuron in layer $l-1$. w_{ik}^{l-1} is the 1D kernel (also called filter) from the ith neuron in layer $l-1$ to the kth neuron in layer l. The ouput of each neuron can be obtained passing the input x_k^l through an activation function $f(.)$, such as 2.3

$$y_k^l = f(x_k^l). \tag{2.3}$$

In this kind of convolution, the resulting output size of the convolution is smaller than the size of the input. In order to achieve an output of the same size, the same padding configuration must be taken notice to the calculation. It simply means adding a null input (line full of zeroes) to the previous layer on the first and last data set. In this way, the kernel will be convoluted also on these null lines, generating at the end an output with the same size as the original data.

In the CNN used in this work, the input signal had a shape of 5 signals with 16 data inputs each, convoluted in one dimension through 32 different kernels. Once the objective was to implement an already trained CNN, the training and backpropagation algorithms will not be explored in this bibliographical research.

Pooling Layer

The pooling layer fundamentally performs an arithmetic operation on the previous convolution layer output to reduce the data size. It takes the output of each convolution kernel and reduces it by a defined algorithm, which can be to return the maximum value, the average value, or others. In the CNN implemented in this work, the average pooling was used.

2.5 Keras

Keras[9] is a deep learning Application Programming Interface (API) written in Python, running on top of the machine learning platform TensorFlow[10] . It is an interface optimized for the implementations of machine learning models. It provides the essential building blocks and abstractions for reading data, configuring the machine learning algorithm and its number of layers, neurons, filters, and other parameters. It is handy as it is portable and scale-able.

The CNN of this work was trained and developed in a Keras environment. In order to extract the trained CNN parameters, a simple python script can be applied.

2.6 FPGA

Field Programmable Gate Arrays, FPGAs, are programmable integrated circuits. They can bring speed and energy improvements both in high-end implementations and energy- or processing-constrained embedded devices, all of that while preserving programmability [11]. They consist in a matrix of Configurable Logic Blocks (CLBs) connected through programmable interconnects. This attribute differentiates FPGAs from Application Specific Integrated Circuit (ASIC), which are designed and manufactured in a permanent logic functionality.

Due to their programmability, FPGAs can be used in many different applications, from ASIC prototyping to task-specific processing in an embedded system. Since neural networks are highly-parallel processing algorithms, they can benefit a lot from the flexibility of FPGAs [11].

2.7 High-Level Synthesis

High-Level Synthesis, also referred to as C-Synthesis, is an automated algorithm to synthesize digital circuits from a C-code input into Hardware Description Language. It can be instrumental in combining the programmability of FPGA design with the facility that a programming language has to implement complex algorithms. In this work, the Vitis HLS tool was used to implement the convolutional neural network.

2.7.1 Vitis HLS

Vitis HLS is a high-level synthesis tool that converts C,C++ and OpenCL functions into hardwired logic circuits. The logic circuits are implemented through the usage of Random Access Memory (RAM)/Digital Signal Processor (DSP) blocks. The tool builds accelerated RTL IP to be further used by the Vivado Design Suite tool. Vivado then synthesizes and implements the RTL description into Xilinx FPGA chips. Vitis HLS also supports customization and specific optimizations to achieve the desired design objectives [12].

2.7.2 Vivado

Xilinx Vivado is a software environment for the development and implementation of HDL designs on FPGAs. It comes with an embedded simulator and IP integrator. It offers the possibility of synthesizing an IP generated from the Vitis HLS tool and for further implementation and routing of all the Xilinx FPGA chips. Vivado also enables power and timing analysis of the design, either in the synthesis phase or in the implementation phase.

2.8 Optimization

Vitis HLS allows the usage of different pragmas to indicate how the C++ code execution should be scheduled and allocated before being converted to Register Transfer Level (RTL). The usage of pragmas is highly recommended to achieve the desired optimization. The used pragmas in this project were:

- **Interface**: Defines which interface of the top main function should be implemented
- **Unroll**: Unrolls loop with the defined unrolling factor
- **Pipeline**: Pipelines loop with the defined pipelining factor. It can also be used with the -off directive to disable the HLS compiler optimizations on the loop
- **Partition**: Partitions array by a partition type and factor

2.9 STM microcontrollers

Microcontrollers are micro-processing devices that integrate memory and I/O functionalities inside a single chip. In the last decades, they were responsible for the revolution of embedded devices, and they still play an essential role today, especially with the advancements of the Internet of Things. For self-contained systems, with all the necessary memory and peripherals to work, they can be used in various applications requiring a simple control logic. Microcontrollers can be very cheap and still offer low-power and low-latency solutions, depending on design requirements.

In the development of this work, two different microcontrollers were used: the STM32-L152RE and the STM32-L412. STM32-L152RE uses an ARM Cortex M3 while STM32-L412 uses an ARM Cortex M4 and comes with a dedicated Floating Point Unit, which can be very useful for an application that requires many floating-point calculations such as the processing of a convolutional neural network.

CHAPTER 3

Implementation

In this chapter, the implementation steps of the CNN are described in detail. Starting from the weights extraction from the Keras model, the chapter then explains the main C++ function of the design, followed by the description of the functionality of its subfunctions.

3.1 Keras Model

The convolutional neural network for processing infrared sensor data was previously implemented as a Keras model, as illustrated in Figure 3.1. In this work's contribution, the CNN parameters and structure were extracted from the Keras model using a Python script and implemented in a C++ code, that was later converted into a Hardware Description Language model using High-Level Synthesis. Since the implementation was for the inference stage, the Dropout Layer and the Flatten layer were not considered.

For the extraction of the weights from the Keras model, the following Python script was used.

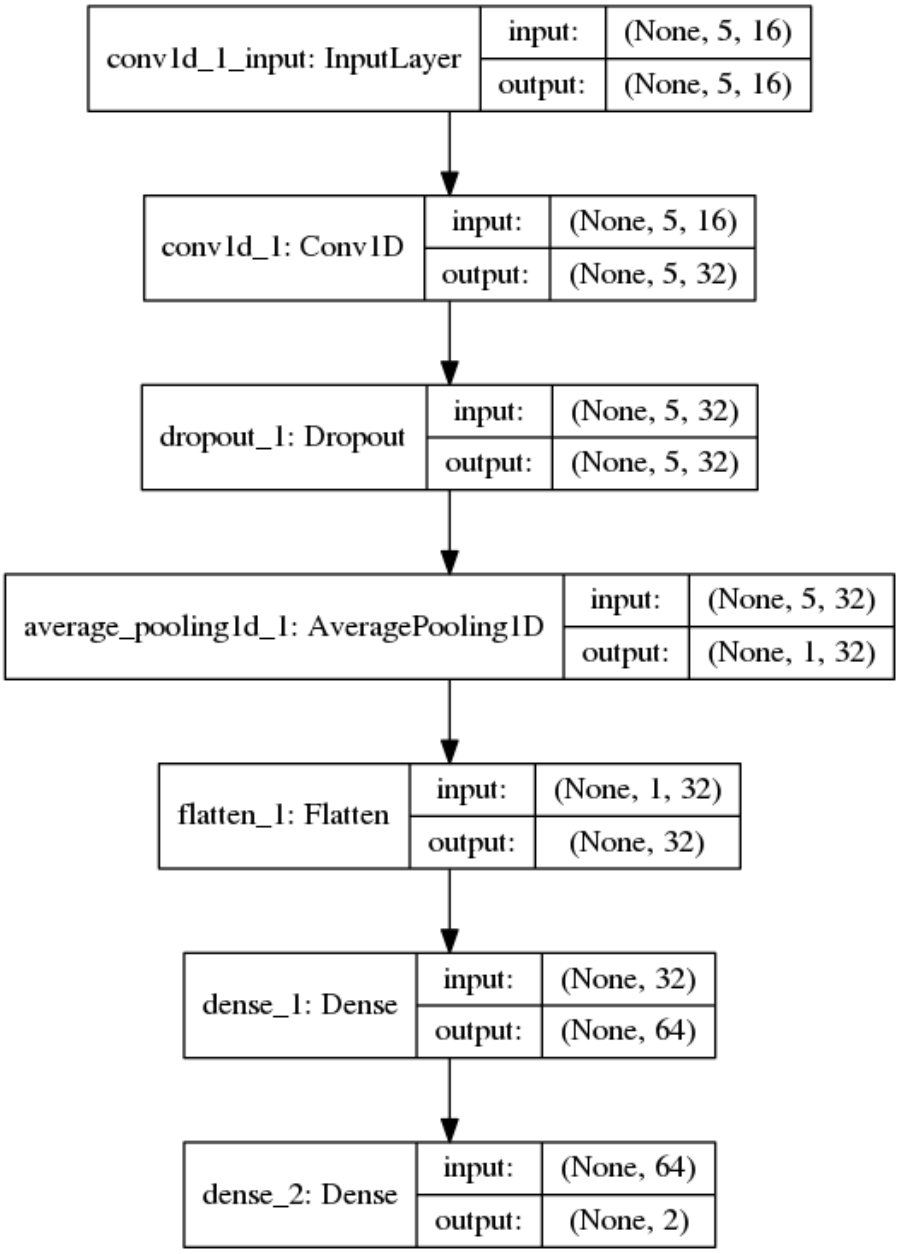

Figure 3.1: Keras model of the CNN

```
# -*- coding: utf-8 -*-


import numpy as np
np.set_printoptions(threshold=np.inf)

from keras.models import load_model
model = load_model('best_model.h5')


```

```
names = [weight.name for layer in model.layers for weight in layer.
    weights]
names = [sub.replace('/', '_') for sub in names]
names = [sub.replace(':0', '_') for sub in names]

#get the weights and store it in an inside variable
weights = model.get_weights()


#Print the weights into a .csv file
for name, weight in zip(names, weights):
    content1 = str(name)
    print(content1)
    weight.tofile(content1+".csv",sep=',',format='%10.5f')
```

Listing 3.1: script for extracting the weights from the Keras model

The script imports the given Keras model and stores its parameters internally. The script then converts the internal organization of the weight arrays to fit in the format of a Comma-Separated Values (CSV) file. The content of each weight is then written in a specific CSV file. In this way, the weights can be copied and pasted to the internal array variables of the C++ code.

3.2 Architecture

In this section, the architecture of the implemented convolutional neural network is described. It consists of the main function, which interconnects a window layer, a convolutional layer, a pooling layer, and two fully connected layers, the last one being the output layer.

3.2.1 Defines

3.2.2 Main Function

The main function of the code developed has two array inputs as interface:

- **datain[DATA_SIZE]** : array of size 16 for the reading of input data

- **op_out[OP_NEURONS]** : array of size 2 (x and y positions) for the output predicted by the CNN

Internally, the CNN is divided in 4 funcions that are sequentially interconnected:

- **conv_layer** : the convolution layer in which the data processed trough the kernels
- **pool_layer** : the layer that perform an average pooling from the data outputted by conv_layer
- **fc_layer** : A fully connected layer that process the data outputted from the pooling layer
- **op_layer** : the output layer, that reads the data of the previous fully connected layer into two neurons, predicting the final position data.

3.2.3 Convolutional Layer

Internally, the convolution layer is divided into two steps, the window buffer and the convolution loop.

The window buffer is the initial buffer that stores the data from the past five measurements. It consists of a shift register that shifts the received data every time data is read from the input FIFO.

The convolution loop is the first processing layer of the network. It takes the data from the window layer and performs a one-dimensional convolution with 32 filters, resulting in a 16x5 matrix. The Kernel size of each filter is 3. The convolution follows the padding 'same', which means that the result of the convolution of one filter has the same size as the input. In order to do that, each filter's calculation is performed with two additional rows with value 0 to the input data. In this way, each filter is convoluted with the input data 5 times, generating an output of a 5x32 matrix.

3.2.4 Pooling Layer

The pooling layer performs a data compression from the previous layer. It takes the 32x5 matrix and, for each column, calculates the average value, resulting in a 1x32 row vector.

3.2.5 Fully Connected

The fully connected layer is a neural network that takes the previous values and passes them through 64 neurons, each neuron with its own configured weight and bias with a ReLU activation function. Each neuron multiplies its inputs with its respective weights and then accumulates the result, adding a bias. This result is passed through a relu activation function, which displays the final output for each neuron.

3.2.6 Output Layer

The Output layer follows the same structure as the fully connected layer but with only two neurons. These neurons indicate each the final position coordinates, x and y, predicted by the convolutional neural network.

3.2.7 ReLU Function

The Rectified Linear Unit (ReLU) function is the activation function used for the entire CNN. It is defined as a function that outputs only the positive part of its argument.

$$f(x) = x^+ = \max(0, x) \tag{3.1}$$

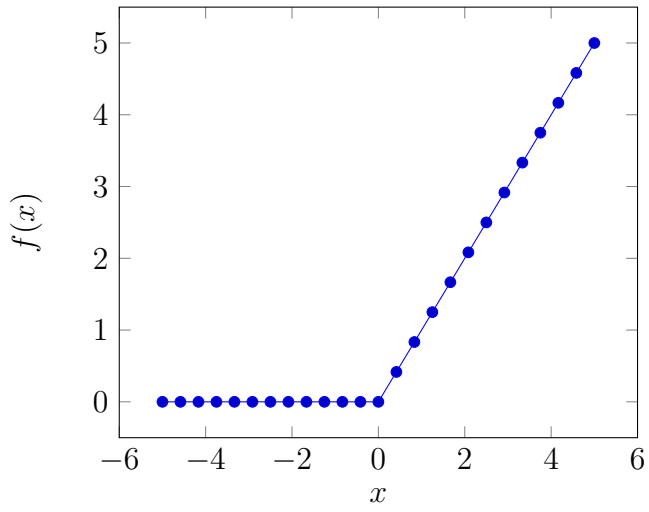

Figure 3.2: Plot of the ReLU activation function

3.3 HLS pragmas

To synthesize the developed C++ code, the software Vitis HLS was used. The tool compiles the code into RTL and provides pragmas used to optimize the hardware design. These optimizations can reduce latency, improve the throughput performance, reduce the area and the resource usage of the resulting RTL code. The pragmas may be inserted directly on the code or in an external TCL script [13].

The number of pragmas accepted by the tool is extensive and can help resolving many design optimization problems. The ones used in this design are described in detail in the following subsections.

3.3.1 INTERFACE

In a C/C++ code, the input and output interfaces of functions are implemented automatically through the linking stage of code compilation. In a physical circuit, the data interface between main functions/modules must be implemented with a specific data transfer protocol and a control logic. The functionality of the **INTERFACE** pragma is to define which protocol must be synthesized from the described function arguments.

3.3.2 UNROLL

With the **UNROLL** pragma, it is possible to execute a loop in parallel instead of a single collection of operations. Loops inside functions are usually kept rolled. The HLS compiler executes the logic for one iteration of the loop and executes it the number of times the loop induction variable configures the loop. Using the **UNROLL** pragma, the logic synthesis implements the loop unrolled, which increases data access and throughput.

The pragma allows the unrolling to be fully or partially. Fully unrolling implements a copy of the loop for every loop iteration so that the entire loop can be executed in just one iteration. Partially unrolling implements the instances of the loop following a configurable parameter N. A loop with a partial unrolling configured with a given N parameter implements the loop N times.

3.3.3 PIPELINE

The **PIPELINE** pragma is used to configure the HLS compiler directives for pipelining. Its goal is to reduce the Initiation Interval (II) for a function or loop by concurrently executing the operations. The Initiation Interval is the number of clock cycles in which a function or a loop can process new inputs. The default compiler goal for optimization with the **PIPELINE** pragma is an Initiation Interval of one, as it can be observed in Figure 3.3. The target Initiation Interval can also be configured

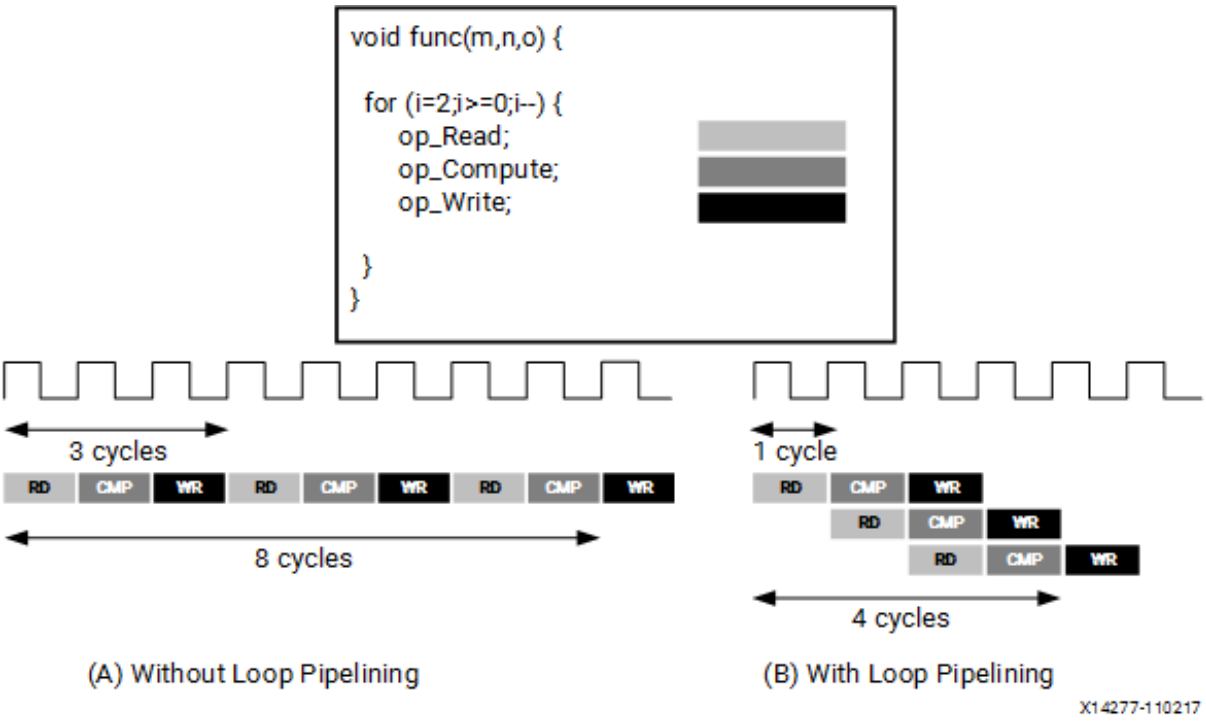

Figure 3.3: Pipelipe Pragma Illustration [12]

Another use of the pragma is the possibility to disable the compiler's automatic optimizations. This disabling is possible by passing the parameter "off" as an input to the pragma. This configuration is beneficial for a design that is focused on a smaller area for implementation.

3.3.4 Array Partition

The **array_partition** pragma partitions the data arrays into smaller or individual elements. Its use results in using smaller registers or memory elements instead of a large memory block. When used combined with pipeline or unroll, it can help to increase the

data throughput, or parallelism, of the design.

The partition can be configured in three types, cyclic, block, or complete. The cyclic partitioning creates smaller arrays interleaving elements from the original array. Therefore, the array is partitioned, putting one element into each new array before returning to the first array, repeating the cycle until the complete partitioning of the array. The block partitioning splits the array into equal blocks following a given parameter N. The complete partitioning, the default configuration, completely decomposes the array into individual elements.

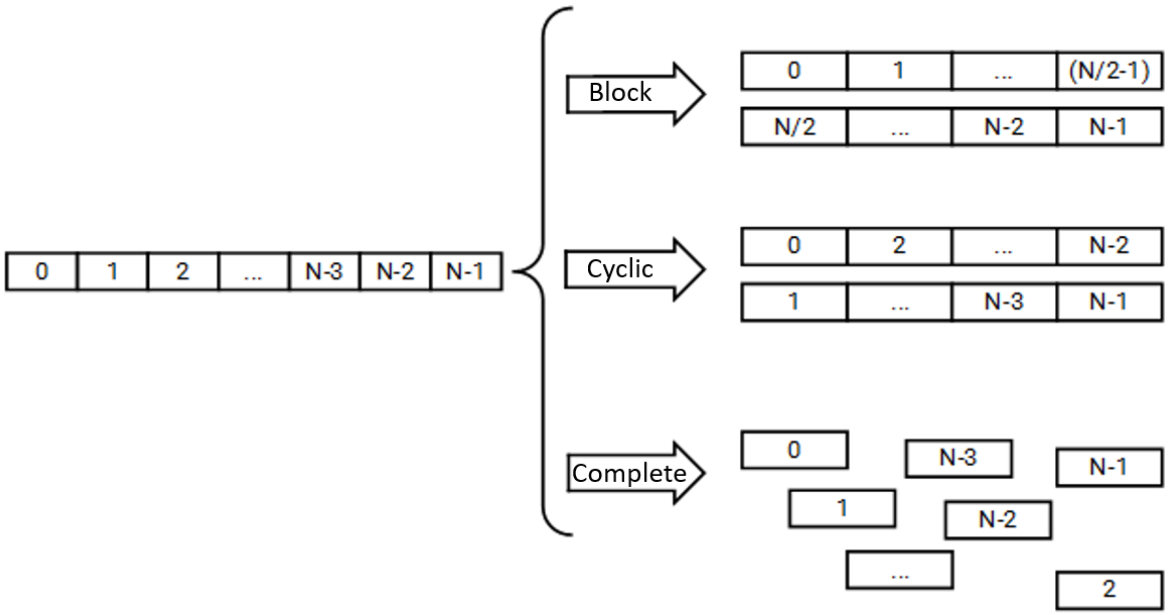

Figure 3.4: Array partitioning pragma illustration [14]

CHAPTER 4

High-Level Synthesis and Simulations

In this Chapter, the simulations performed are described in detail. Starting from the Hypotheses elaborated through the design development, it then describes how these hypotheses were tested and in which conditions the simulations were executed.

For the HLS implementation of the CNN, several different solutions were performed to explore the design space well. The different design space explorations were in parallelism, FPGA technology, data type, and performance comparison with microcontrollers.

4.1 Hypotheses

The first hypothesis for exploring the solutions was that a smaller, sequential implementation of the circuit would spend less static and dynamic power, which would result in a cheap, low-power, low-area implementation of the algorithm. The high parallelism implementation using more area would result in an expensive, large but faster circuit in latency.

Further hypotheses for exploration were in terms of different FPGA technologies, in which smaller technologies would consume less power, or different data-types (floating vs. fixed point), in which a fixed-point implementation of the calculations would reduce the device power with the cost of data precision.

The final hypothesis would be for the comparison between FPGAs and different CPUs. The assumption was that the hardware implementation of the algorithm would result in a faster and less energy spending circuit at the cost of more expensive technology.

All of the simulations performed and the further power analysis considered just

the internal power for calculation, and therefore did not consider the power of the host systems or the data transfer. These should also be taken into account in future work for implementations. In this work, the cost/performance ratio was not optimized for either microcontrollers or FPGAs.

4.2 Parallelism

In terms of the area-parallelism hypothesis, nine synthesis experiments followed by their respective simulations were performed to verify the circuit's behavior in terms of power in a growing gradient of parallelism. The first solution would disable all pipelining and parallelism optimizations to get the most sequential implementation possible. Other solutions would increase the parallelism up until the compiler would reach the maximum parallelism possible. The implementations were:

- **None**: All of the pipelining and array partitionings were disabled, resulting in a highly sequential circuit
- **Inner-Loops**: All of the most inner loops of the CNN were set to be unrolled and pipelined, leaving the other loops without compiled optimizations
- **Balanced-Inner**: The inner loops were set to be unrolled in a balanced way, meaning that all of them would be executed with the same unroll factor of 16
- **Half**: All of the loops of the design were unrolled by half with a factor of two
- **Inner-off**: Optimization of all of the inner-loops are disabled
- **Outer-Loops**:Optimization of the most outer loops are unrolled, while the inner loops are disabled
- **Conv-Layer**: Optimization of the whole convolutional-layer loops are unrolled and pipelined, while the other modules are disabled for optimization
- **OuterOff**: all of the inside loops are set for unrolling optimization pragma, except for the most outer one, which is turned off
- **Full**: All of the loops of the code are unrolled and optimized.

After the generation of the RTL models, the netlists were synthesized in Vivado and simulated for ten iterations of calculations. This step was performed in order to generate the switching activity of each design. This step was crucial in order to perform a more reliable power analysis.

After the synthesis and simulation, the power analysis was executed in Vivado for each solution.

4.3 FPGA Technologies

After comparing different kinds of parallelism, the implementation of the design on a smaller-transistor technology would be beneficial in terms of power, even if it would still increase the cost of the chip. Intending to test this assumption, the optimal solution from the previous step was implemented in an FPGA chip of the family Zynq Ultrascale+. After that, a comparison between the two different technologies was performed.

4.4 Data Types

For the data type simulation, the optimal solution from the subsection 4.2 was compared to a solution following the same directives of optimizations (complete unrolling of all loops), with the only difference of the data type used. The data type used for this simulation was a fixed-point implementation with the configuration of 4 bits for the signed integer part and 20 bits for the decimal part of the number. This implementation would allow the representation of data of integer numbers from -7 to +7 with a decimal digits precision of 10^{-6}, which is a little less than the precision of the floating-point implementation. The FPGA technology family used for this implementation was the Artix-7. A comparison between the precision of the final calculation from both floating and fixed-point implementations was performed. This step was performed to evaluate in practice the precision to be expected from the fixed-point design.

The fixed point configuration of this experiment was chosen arbitrarily as an implementation closer to the floating-point than the 16 bits implementation of the half-precision floating-point. Future work could explore this scenario.

4.5 FPGA vs Microcontrollers

As a final analysis, the previously analyzed implementations were compared to the efficiency of microcontrollers. Two microcontrollers were selected. One with and one without a floating-point unit. For the microcontroller energy consumption estimation running the algorithm, the following steps were performed for both of the microcontrollers:

- Firstly, the code used for the HLS implementations was used as a C++ code and implemented, via STM32 compiler, inside two STM32 microcontroller modules.
- In an infinite loop, before and after the CNN calculations, a command to toggle the state of an I/O was set.

```
1  /* Infinite loop */
2  /* USER CODE BEGIN WHILE */
3  while (1)
4  {
5    /* USER CODE END WHILE */
6
7    /* USER CODE BEGIN 3 */
8  HAL_GPIO_TogglePin(GPIOA, D9_Pin);
9  conv_layer(conv1_out, datain, conv_layer_weights,conv_layer_bias,
     CONV1_DATA_SIZE,CONV1_CHANNELS, CONV1_KERNEL_SIZE,
    CONV1_FILTERS, CONV1_STRIDE );
10 pool_layer(pool1_out, conv1_out, P1_SIZE, P1_CHANNELS,
    P1_KERNEL_SIZE, P1_STRIDE, P1_OUT );
11 fc_layer(fc1_out, pool1_out,fc1_layer_weights, fc1_layer_bias,
    FC1_NEURONS,FC1_CHANNELS);
12 fc_layer(op_out,fc1_out, op_layer_weights, op_layer_bias,
    OP_NEURONS,OP_CHANNELS);
13
14
15 }
16 /* USER CODE END 3 */
```

Listing 4.1: infinite loop for toggling the state of the I/O before and after calculation

- The state of the I/O was monitored in an oscilloscope, and the time for the toggle was measured, as it can be observed in Figure 4.1. By doing this, the time needed

by the CPU to calculate one iteration could be found.

Figure 4.1: Simulation performed for measuring the microcontrollers execution time

- The time acquired in the last step was used to calculate the energy spent by the CPU by multiplying it by the current draw value that could be found in the device's datasheet.

The selected microcontrollers for evaluation were the STM32L412, with an embedded Floating Point Unit, from the Arm Cortex-M4 32-bit family, and STM32L152 from the Arm Cortex-M3 32-bit family, which does not have a Floating Point Unit.

For better comparison, both of them executed the code wiht the same clock frequency of 32 MHz, which required a supply voltage of 1.8V. In order to estimate the total energy consumption, the datasheets [15] [16] were consulted and the run mode tables were extracted.

Table 5. Functionalities depending on the working mode (from Run/active down to standby) (continued)

Ips	Run/Active	Sleep	Low-power Run	Low-power Sleep	Stop		Standby	
					-	Wakeup capability	-	Wakeup capability
ADC	Y	Y	--	--	--	--	--	--
DAC	Y	Y	Y	Y	Y	--	--	--
Tempsensor	Y	Y	Y	Y	Y	--	--	--
OP amp	Y	Y	Y	Y	Y	--	--	--
Comparators	Y	Y	Y	Y	Y	Y	--	--
16-bit and 32-bit Timers	Y	Y	Y	Y	--	--	--	--
IWDG	Y	Y	Y	Y	Y	Y	Y	Y
WWDG	Y	Y	Y	Y	--	--	--	--
Touch sensing	Y	Y	--	--	--	--	--	--
Systic Timer	Y	Y	Y	Y	-	--	--	--
GPIOs	Y	Y	Y	Y	Y	Y	--	3 pins
Wakeup time to Run mode	0 µs	0.4 µs	3 µs	46 µs	< 8 µs		58 µs	
Consumption V_{DD}=1.8 to 3.6 V (Typ)	Down to 195 µA/MHz (from Flash)	Down to 38 µA/MHz (from Flash)	Down to 11 µA	Down to 4.6 µA	0.53 µA (no RTC) V_{DD}=1.8V		0.285 µA (no RTC) V_{DD}=1.8V	
					1.2 µA (with RTC) V_{DD}=1.8V		0.97 µA (with RTC) V_{DD}=1.8V	
					0.56 µA (no RTC) V_{DD}=3.0V		0.29 µA (no RTC) V_{DD}=3.0V	
					1.4 µA (with RTC) V_{DD}=3.0V		1.11 µA (with RTC) V_{DD}=3.0V	

1. The startup on communication line wakes the CPU which was made possible by an EXTI, this induces a delay before entering run mode.

Figure 4.2: Run modes for STM32L152xE (from datasheet)

Table 4. STM32L412xx modes overview

Mode	Regulator(1)	CPU	Flash	SRAM	Clocks	DMA and Peripherals(2)	Wakeup source	Consumption(3)	Wakeup time
Run	MR range 1	Yes	ON(4)	ON	Any	All	N/A	91 µA/MHz	N/A
	SMPS range 2 high							34 µA/MHz	
	MR range2					All except USB_FS, RNG		79 µA/MHz	
	SMPS range 2 low							28 µA/MHz	

Figure 4.3: Run modes for STM32L412xx (from datasheet)

From Figures 4.2 and 4.3, the values of current consumption per MHz in run mode were extracted and formed the Table 4.1 with the calculated parameters for the power supply.

STM32 POWER SUPPLY		
Solution	No-FPU	FPU
Component	STM32L152RE	STM32L412
Technology	Arm® Cortex®-M3 32-bit	Arm® Cortex®-M4 32-bit
Cost (Euro)	7.1	3.13
Frequency (MHz)	32	32
Supply voltage(V)	1.8	1.8
Run Mode Current (μA/MHz)	195	91
Active Current (mA)	6.24	2.91

Table 4.1: Power supply parameters for the STM32 microcontrollers

CHAPTER 5

Results Analysis

The following sections represent the design-space exploration performed for the implementation of the algorithm. The scenarios were divided between Hardware and Software implementations. On the Hardware section, different configurations of the implementation in the FPGA were explored, while in the Software section, the performance of the algorithm ran on different microcontrollers was analyzed.

5.1 Hardware implementations

5.1.1 Parallelism

The results of the parallelism hypothesis can be observed in Tables 5.1, 5.2 and 5.3.

From Table 5.1, it can be firstly observed that **None** implementation fits it a very small and cheap FPGA component and with a total energy consumption slightly higher then the **Inner-Loops** solution, even if the **Inner-Loops** reduces the total iteration latency by a half when compared to **None**.

In the **Balanced-Inner** solution fewer resources are used, but the dynamic power almost doubles in value indicating that it is not an interesting solution for exploration. This may be due the necessity of multiplexing and data storage needed for implementing this solution, as some of the loops are not unrolled completely, but just partially.

SMALL FPGA CHIPS			
Parallelism	None	Inner-Loops	Balanced Inner
Component	xc7a12tl	xc7a50tl	
Cost (Euro)	26.9	65.1	
Iteration Latency (μs)	1153	568	561
Usage/Available			
DSP	5/40	105/120	70/120
FF	2058/16000	31326/65200	28398/65200
LUT	4225/8000	25906/32600	26435/32600
BRAM	15/40	122/150	122/150
Frequency (MHz)	100	100	100
Slack (ns)	0.201	0.225	0.255
Supply voltage(V)	0.9	0.9	0.9
Dynamic Power (mW)	23	109	176
Static Power (mW)	59	69	69
Total Power (mW)	82	178	245
Energy (nJ)	89.9	101	137

Table 5.1: Results for small FPGA chips

From the implementations observed in Table 5.2, it can be firstly be observed that this higher degree of parallelism and resource usage does not come with advantages in terms of power and energy consumption.

The **Half** solution can be noticed by its usage of resources, which is quite less then the resources generally used by **Inner-Off**. The usage of LUTs, however, is higher. In the end, even sing less resources, and less dynamic power, the **Inner-Off** solution uses less energy for the calculation due to its faster iteration latency.

MEDIUM FPGA CHIP		
Parallelism	Half	Inner-Off
Component	xc7a75tl	
Cost (Euro)	105	
Iteration Latency (μs)	494	394
Usage/Available		
DSP	24/180	162/180
FF	26310/94400	27643/94400
LUT	40576/47200	20660/47200
BRAM	35/210	92/210
Frequency (MHz)	100	100
Slack (ns)	0.281	0.171
Supply voltage(V)	0.9	0.9
Dynamic Power (mW)	147	164
Static Power (mW)	86	87
Total Power (mW)	233	251
Energy (nJ)	115	98.9

Table 5.2: Results for medium FPGA chip

As it can be seen from Table 5.3, the implementation with the least energy expenditure in all of the design space exploration is the **OuterOff** solution. It was supposed that a slower and smaller circuit could spend less energy by having a lower static power, however, as it seems, parallelization can really pay off the higher static power, but up until a certain point. The Full solution has the highest degree of paralellization, it is faster, however the total energy spent for an iteration is higher than the OuterOff solution.

The effect of parallelization can only be observed in the Full and OuterOff implementation because most of the loops are pipelined and optimized, which drastically reduces the latency, by a factor of 100x when compared to the None implementation, even if the dynamic power used is very high when compared to all of the implementations. Mainly, what it can be concluded from this experiment is that the acceleration of the hardware calculation can reduce the total energy expenditure, at the cost of area. However the

loops must be pipelined and unrolled to exploit parallelism.

LARGE FPGA CHIP				
Parallelism	Outer-Loops	Conv-layer	OuterOff	Full
Component	xc7a200tl			
Cost (Euro)	212			
Iteration Latency (μs)	115	283	9.85	9.70
Usage/Available				
DSP	5/140	481/740	484/740	481/740
FF	40090/269200	56942/269200	88934/269200	66622/269200
LUT	97577/134600	101927/134600	66029/134600	132533/134600
BRAM	102/730	6/730	80/730	0/730
Frequency (MHz)	100	100	100	100
Slack (ns)	-0.322	-2	0.052	0.052
Supply voltage(V)	0.9	0.9	0.9	0.9
Dynamic Power (mW)	179	354	840	1073
Static Power (mW)	121	122	123	123
Total Power (mW)	300	476	963	1196
Energy (nJ)	34.5	135	9.49	11.6

Table 5.3: Results for large FPGA chip

5.1.2 FPGA Technologies

It can be seen from these experiments results in Table 5.4 that for the choice of the smaller-transistor technology, the investment does not really achieve a considerable improvement. Even if the dinamic power achieved is lower, the static power consumption for the UltraScale+ chip makes this optimization unnoticed in the final power and energy spenditure. On future work, some simulations could be performed with designs with a lower degree of parallelism with smaller UltraScale+ chips as well in order to verify the broad design space.

From the high slack of 7.87 ns observed in the implementation of the OuterOff solution on the UltraScale+ chip, it can be noticed that there is a high space for performance optimization. A higher frequency of 469 MHz could be achieved with this circuit, which would reduce the iteration latency and could reduce the total energy consumption, even

if the dynamic power could potentially rise. Further work could be performed in this implementation to observe the trade-offs in practice.

DIFFERENT FPGA TECHNOLOGY			
Parallelism	OuterOff	Full	OuterOffUltra
Component	xc7a200tl		xczu4ev UltraScale+
Cost (Euro)	212		880
Iteration Latency (μs)	9.85	9.70	5.52
Usage/Available			
DSP	484/740	481/740	485/728
FF	88934/269200	66622/269200	90344/175680
LUT	66029/134600	132533/134600	69258/87840
BRAM	80/730	0/730	80/256
Frequency (MHz)	100	100	179
Slack (ns)	0.052	0.052	-0.299
Supply voltage(V)	0.9	0.9	0.85
Dinamic Power (mW)	840	1073	1128
Static Power (mW)	123	123	322
Total Power (mW)	963	1196	1450
Energy (nJ)	9.49	11.6	8

Table 5.4: Results for FPGA chips from different technologies

5.1.3 Data Types

Firstly, from the precision evaluation in Table 5.5, it can be seen that the highest error found for the in the fixed point calculation is 0.002081, which represents 0.15% of the respective prediction of X. This analysis indicates that the calculation performed with can be a promising field for exploration, as the final relative error of this fixed point implementation is small and mantains the prediction reliable.

PRECISION EVALUATION						
	Float		Fixed		Error	
Calculation	X	Y	X	Y	X	Y
1	1.306774	2.244772	1.305695	2.243301	0.001079	0.001471
2	1.311916	2.293143	1.310211	2.291672	0.001705	0.001471
3	1.337289	2.268058	1.335510	2.267120	0.001779	0.000938
4	1.380041	2.223119	1.377960	2.222168	**0.002081**	0.000951
5	1.443046	2.199926	1.442215	2.198318	0.000831	0.001608
6	1.487756	2.217685	1.486862	2.216080	0.000894	0.001605
7	1.561531	2.256933	1.560120	2.255615	0.001411	0.001318
8	1.662190	2.284451	1.661438	2.283279	0.000752	0.001172
9	1.712496	2.281284	1.712006	2.279846	0.000490	0.001438
10	1.74027	2.302691	1.739807	2.300934	0.000463	0.001757

Table 5.5: Fixed-point precision evaluation

From Table 5.6, it can be seen that the usage of the fixed point implementation can continue the power optimization of the design by reducing the area and latency for calculation. This implementation resulted in the total energy for iteration of 10 times less than the floating point implementations.

From this analysis, it can be concluded that the fixed point implementation can still produce a significant result for the prediction of the position, with a small relative error. Depending on the application, this implementation maintains the reliability of the prediction algorithm.

DIFFERENT DATA TYPES			
Parallelism	OuterOff	OuterOffUltra	OuterOffFixed
Component	xc7a200tl	xczu4ev UltraScale+	xc7a200tl
Cost (Euro)	212	880	212
Iteration Latency (μs)	9.85	5.52	4.32
Usage/Available			
DSP	484/740	485/728	309/740
FF	88934/269200	90344/175680	25882/269200
LUT	66029/134600	69258/87840	14006/134600
BRAM	80/730	80/256	27/730
Frequency (MHz)	100	179	77
Slack (ns)	0.052	-0.299	-0.859
Supply voltage(V)	0.9	0.85	0.9
Dynamic Power (mW)	840	1128	98
Static Power (mW)	123	322	121
Total Power (mW)	963	1450	219
Energy (nJ)	9.49	8	0.945

Table 5.6: Results the different data types implementations

5.2 Software Implementations

From Table 5.7, it can be seen that microcontrollers can be an interesting solution for the processing of the algorithm in a low-power design. The **STM32-L412**, with a special Floating Point Unit turned out to be a good implementation of the design with a low cost of implementation, having an iteration energy slightly higher than the **None** solution. Furthermore, it can be seen that, with a higher investment in implementation, the iteration energy can be improved by a higher order, up until 1000 times lower in the **OuterOff-Fixed** fixed-point implementation when compared to the **STM32-L152RE** microcontroller. The higher investment in the technology may not pay off, however, as it can be observed in **OuterOff-Ultra** solution, which is implemented in a more expensive chip but delivers a final energy consumption that is very similar to the one in **OuterOff**.

MICROCONTROLLERS VS FPGAs						
Solution	No-FPU	FPU	None	OuterOff	OuterOff-Ultra	OuterOff-Fixed
Component	STM32-L152RE	STM32-L412	xc7a12tl	xc7a200tl	xczu4ev	xc7a200tl
Cost (Euro)	7.1	3.13	26.86	211.82	800	211.82
Iteration Latency (μs)	58600	20000	1153	9.85	5.52	4.32
Frequency (MHz)	32	32	100	100	179	77
Supply Voltage(V)	1.8	1.8	0.9	0.9	0.85	0.9
Total Power (mW)	11.2	5.24	82	963	1450	219
Energy (nJ)	658	105	94.5	9.49	8	0.945

Table 5.7: Results for microcontrollers compared to different FPGA solutions

The effect of the analysis can be better observed in Figure 5.1. The cost is displayed as a function of the energy consumption so that the energy saving in each solution could be observed. As it can be seen, the solutions show an inversely proportional pattern, the lower the cost, the higher the energy. Also, at a certain point, the energy saving comes with a much higher cost, in **OuterOffUltra**. Therefore, a solution for further reducing the energy consumption must come with other design losses, such as the precision loss in the **OuterOffFixed** solution.

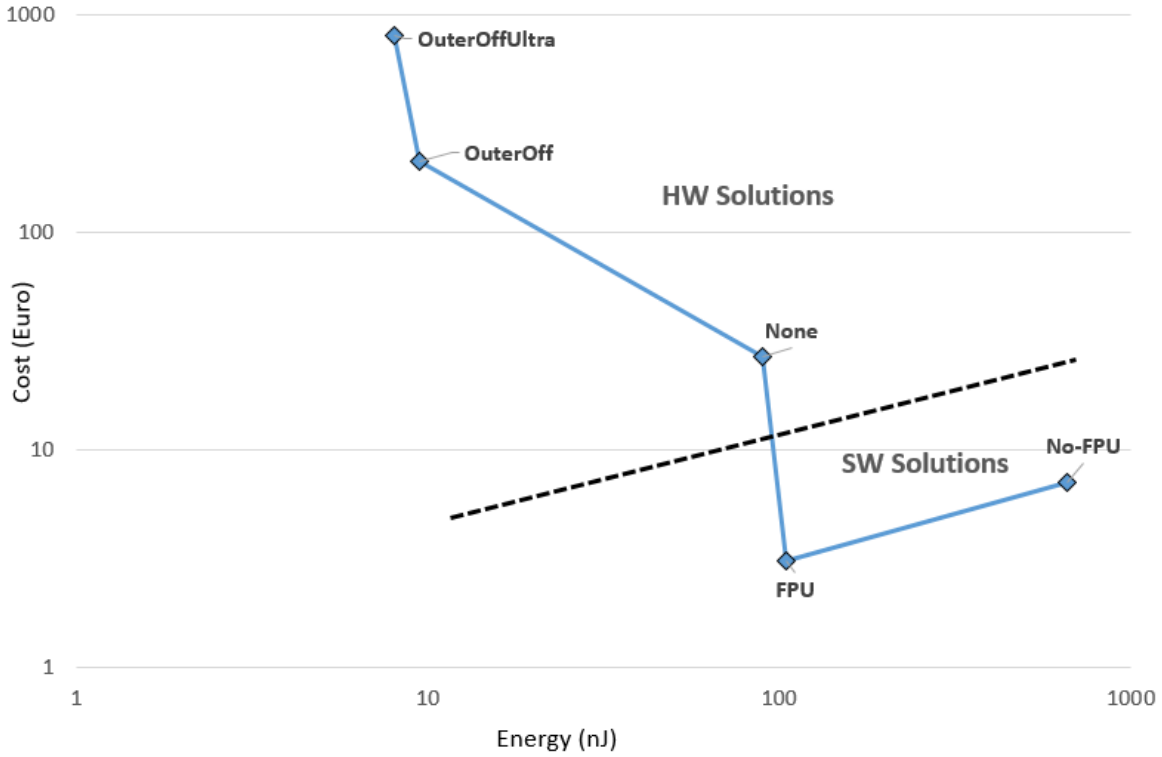

Figure 5.1: Cost vs Energy

CHAPTER 6

Discussion and Conclusion

6.1 Conclusion

As a preliminary conclusion, it can be said that High-Level Synthesis is a potent and practical tool to use in hardware design. The agility of configuring the pragma directives for HDL synthesis for several different solutions made this broad design space exploration possible.

In the light of the results from the performed experiments, it can be concluded that the main objective of the design space exploration was reached. The trade-offs of cost, power, area, and technology have been searched, and different solutions for the implementations were offered. Microcontrollers can offer a cheap and relatively low-power solution. However, FPGAs offer a more extensive space for design exploration, with much lower power implementations and the possibility for hardware capable of computing using a specific data type, such as the fixed-point implementation used in this project. An important drawback of FPGAs, however, is the higher unit cost.

In summary, Table 5.7 illustrates the whole cost/performance/energy/precision trade-off curve. It shows that modern microcontrollers with dedicated numeric computing units such as **STM32-L412** can perform very well, especially in terms of cost. In applications in which the energy consumption is a significant part of the total system cost, such as data centers or embedded devices with very high battery replacement cost and no independent power sources, the high price of FPGA technology could be compensated.

The main objective of the Thesis work of exploring the design space of the imple-

mentation of the convolutional neural network was achieved. Different solutions for the low power design were found, for low-cost and for high-end implementations.

6.2 Future Work

Some different scenarios arise from the evaluation performed in this thesis. Future work could be performed to explore the trade-off between fixed-point low power implementation and the precision of calculation of the floating-point. The fastest implementation of the algorithm in smaller FPGA technologies such as the UtraScale+ family for high-end applications could also be explored. Faster implementation of the algorithm means that the run-time is smaller, and therefore the total energy consumed for processing the calculation could be lower than the implementation in more prominent families.

References

[1] T. Kivimäki, T. Vuorela, P. Peltola, and J. Vanhala, "A review on device-free passive indoor positioning methods," *International Journal of Smart Home*, vol. 8, pp. 71–94, 01 2014.

[2] O. B. Tariq, M. T. Lazarescu, and L. Lavagno, "Neural networks for indoor human activity reconstructions," *IEEE Sensors Journal*, vol. 20, no. 22, pp. 13571–13584, 2020.

[3] C. Zhang, D. Wu, J. Sun, G. Sun, G. Luo, and J. Cong, "Energy-efficient cnn implementation on a deeply pipelined fpga cluster," in *Proceedings of the 2016 International Symposium on Low Power Electronics and Design*, ISLPED '16, (New York, NY, USA), p. 326–331, Association for Computing Machinery, 2016.

[4] F. U. D. Farrukh, T. Xie, C. Zhang, and Z. Wang, "Optimization for efficient hardware implementation of cnn on fpga," pp. 88–89, 2018.

[5] J. M. Rabaey, A. Chandrakasan, and B. Nikolic, *Digital Integrated Circuits*. USA: Prentice Hall Press, 3rd ed., 2008.

[6] T. M. Mitchell, *Machine Learning*. New York: McGraw-Hill, 1997.

[7] S. Kiranyaz, O. Avci, O. Abdeljaber, T. Ince, M. Gabbouj, and D. J. Inman, "1d convolutional neural networks and applications: A survey," *Mechanical Systems and Signal Processing*, vol. 151, p. 107398, 2021.

[8] S. Kiranyaz, A. Gastli, L. Ben-Brahim, N. Alemadi, and M. Gabbouj, "Real-time fault detection and identification for mmc using 1-d convolutional neural networks," *IEEE Transactions on Industrial Electronics*, vol. PP, pp. 1–1, 05 2018.

[9] F. Chollet *et al.*, "Keras." https://keras.io, 2015.

[10] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng, "TensorFlow: Large-scale machine learning on heterogeneous systems," 2015. Software available from tensorflow.org.

[11] M. Roukhami, M. T. Lazarescu, F. Gregoretti, Y. Lahbib, and A. Mami, "Very low power neural network fpga accelerators for tag-less remote person identification using capacitive sensors," *IEEE Access*, vol. 7, pp. 102217–102231, 2019.

[12] Xilinx, *Vitis High-Level Synthesis User Guide*.

[13] Xilinx, "Xilinx hls pragmas," 2021.

[14] Xilinx, "Sdsoc environment profiling and optimization guide," 2019.

[15] STMicroelectronics, *Ultra-low-power 32-bit MCU Arm®-based Cortex®-M3 with 512KB Flash, 80KB SRAM, 16KB EEPROM, LCD, USB, ADC, DAC*, 2021. Rev 10.

[16] STMicroelectronics, *Ultra-low-power Arm® Cortex®-M4 32-bit MCU+FPU, 100DMIPS, up to 128KB Flash, 40KB SRAM, analog, ext. SMPS*, 2021. Rev 8.

APPENDIX A

CNN code

```
#include "defines.h"
#include "weights.h"
#include "functions.h"
#include "Modules.h"

#include <hls_stream.h>




void CNN(volatile float op_out[OP_NEURONS],volatile float datain[
    DATA_SIZE]){
#pragma HLS INTERFACE ap_fifo port=datain
#pragma HLS INTERFACE ap_fifo port=op_out



    float pool1_out[P1_CHANNELS] = {0};

    float fc1_out[FC1_NEURONS] = {0};

    float conv1_out[CONV1_FILTERS][CONV1_CHANNELS] = {0};



  conv_layer(conv1_out, datain);
```

```
27  pool_layer(pool1_out, conv1_out);
28  fc_layer(fc1_out, pool1_out);
29  op_layer(op_out,fc1_out);
30
31
32  }
```

Listing A.1: CNN main function breaklines

APPENDIX B

CNN modules

```
#include "functions.h"
#include "ap_fixed.h"
#include <hls_stream.h>

void conv_layer(float out[CONV1_FILTERS][CONV1_CHANNELS],volatile float
    in[DATA_SIZE]) {


  static float window[5*DATA_SIZE]{};
    int  filter, row_offset,column_offset, channel_offset;
    float sum;


//shift register for storing the 16 values of datain in a window mode
    with the padding 'same': first and last rows are equal to 0
buffer:   for(int m = 0; m < DATA_CHANNELS; m++){
            buffer_label0:for (int n = 0; n < DATA_SIZE; n++){
              if(m == DATA_CHANNELS-1){
                window[m*DATA_SIZE + n] = in[n];

              }
              else
                window[(m)*DATA_SIZE + n] = window[(m+1)*DATA_SIZE + n];

              }

            }
```

```
26
27 //convolution layer with padding 'same'
28 convolution:  for (filter = 0; filter < CONV1_FILTERS; filter++){
29           conv_layer_label2:for (channel_offset = 0; channel_offset <
    CONV1_CHANNELS; channel_offset += CONV1_STRIDE ){
30             sum = 0;
31             conv_layer_label1:for(row_offset = 0; row_offset <
    CONV1_KERNEL_SIZE; row_offset++){
32               conv_layer_label0:for(column_offset = 0; column_offset <
    DATA_SIZE; column_offset++){
33
34                 if(!(((row_offset == 0) && (channel_offset ==0)) || ((
    row_offset == CONV1_KERNEL_SIZE-1) && (channel_offset ==
    CONV1_CHANNELS-1)) ) )
35                 sum += window[(channel_offset + row_offset - 1)*
    DATA_SIZE + column_offset]*conv_layer_weights[row_offset][
    column_offset][filter];
36
37                 }
38               }
39             out[filter][channel_offset] = relu(sum + conv_layer_bias[
    filter]);
40           }
41           }
42 }
43
44
45
46 void pool_layer(float out[CONV1_FILTERS],float in[CONV1_FILTERS][
    CONV1_CHANNELS]) {
47
48
49   int i, j;
50   float average;
51
52
53 pooling:         for(i = 0; i < CONV1_FILTERS ; i++){
54
55                 average = 0;
```

```
                pool_layer_label1:for (j = 0; j < CONV1_CHANNELS; j++){
                  average += in[i][j]/CONV1_CHANNELS;
                }
                out[i] = average;
            }

}



void fc_layer(float output[FC1_NEURONS], float input[FC1_CHANNELS]){

  float rel;

  NN: for (int i = 0; i < FC1_NEURONS; i++) {

      rel = 0;
        fc_layer_label2:for (int j = 0; j < FC1_CHANNELS; j++) {
          rel += input[j]*fc1_layer_weights[j][i] ;
        }
      output[i] = relu( rel  + fc1_layer_bias[i]);
    }
  }

void op_layer(volatile float out[OP_NEURONS], float input[OP_CHANNELS]){

    float rel, buff;

  output: for (int i = 0; i < OP_NEURONS; i++) {

        rel = 0;
          op_layer_label3:for (int j = 0; j < OP_CHANNELS; j++) {
            rel += input[j]*op_layer_weights[j][i] ;
          }
        out[i]= relu( rel  + op_layer_bias[i]);
      }
}
```

Listing B.1: CNN Modules breaklines

APPENDIX C

CNN Testbench

```
#include <stdio.h>
#include <string.h>
#include "ap_fixed.h"
#include <iostream>
#include <fstream>
#include <string>
#include <sstream>
#include <stdlib.h>

void CNN(volatile float op_out[2],volatile float datain[16]);

int main(){
  setvbuf(stdout,NULL,_IONBF,0);

  float datain[16]{};
  float op_out[2]{};
  float temp{};

  //read values from preprocessed data file and store it in Datain
  FILE *file = fopen("preprocessed.csv", "r");
  if(file == NULL) exit(1);
  char line[200];
  char *token;

  for(int i = 0; i < 10; i ++){       //perform the test 100 times
    fgets(line, sizeof(line), file);
    token = strtok(line, ",");
```

```
28      for (int col = 0; col < 16; ++col){
29        temp = strtod(token,NULL);
30
31        datain[col] = temp; // cast temporary value to floating point
32
33        token = strtok(NULL,",");
34
35      }
36
37      CNN(op_out,datain);
38      printf("iteration: %i \n", i);
39      printf("out1: %f \t out2: %f \n", op_out[0], op_out[1]);
40    }
41
42
43
44
45    return 0;
46  }
```

Listing C.1: Testbench for simulation breaklines