A REVIEW STUDY ON THE USE OF HIGH-LEVEL SYNTHESIS FOR IMPLEMENTING DEEP LEARNING ALGORITHMS IN FPGAs

A THESIS SUBMITTED TO
THE FACULTY OF ARCHITECTURE AND ENGINEERING
OF
EPOKA UNIVERSITY

BY

DANJELA RUÇI

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR
THE DEGREE OF MASTER OF SCIENCE
IN
ELECTRONICS AND COMMUNICATION ENGINEERING

OCTOBER, 2020

Approval sheet of the Thesis

This is to certify that we have read this thesis entitled "**A Review Study On The Use Of High-Level Synthesis For Implementing Deep Learning Algorithms In FPGAs**" and that in our opinion it is fully adequate, in scope and quality, as a thesis for the degree of Master of Science.

Dr. Ali Osman Topal
Head of Department
Date: October 02, 2020

Examining Committee Members:

Dr. Ali Osman Topal	(Computer Engineering)
Dr. Arban Uka	(Computer Engineering)
Dr. Julian Hoxha	(Computer Engineering)

I hereby declare that all information in this document has been obtained and presented in accordance with academic rules and ethical conduct. I also declare that, as required by these rules and conduct, I have fully cited and referenced all material and results that are not original to this work.

Danjela Ruçi

ABSTRACT

A REVIEW STUDY ON THE USE OF HIGH-LEVEL SYNTHESIS FOR IMPLEMENTING DEEP LEARNING ALGORITHMS IN FPGAs

Ruçi, Danjela

M.Sc. Department of Computer Engineering

Supervisor: Dr. Julian Hoxha

Because of the high precision that they offer, CNNs represent a very important model for systems that do image identification. However, such a task has high costs. For this reason, the current goal is to implement designs that are fast, but at the same time not costly. GPUs are an alternative, but they do not offer the best solution due to their large power consumption. FPGAs on the other hand, suit more with CNNs systems because they consume less energy and have a flexible structure. The difficult part for FPGA architectures is implementing CNN systems using HDL, which is not a platform on which to program; it is simply hardware-level code to describe components of hardware like registers and counters. With HLS, designers are now capable of using high-level languages like C or C++ to implement CNNs into FPGAs, because HLS "translates" or synthesizes the codes written in high-level languages into hardware-level code or RTL parameters. This thesis represents a review on the previous work done on the CNNs implementation on FPGAs using HLS and summarize the results obtained.

Keywords: Convolutional Neural Networks (CNNs), FPGA, High-Level Synthesis, Hardware Description Language, power consumption

ABSTRAKTI

NJË STUDIM TEORIK MBI PËRDORIMIN E SINTEZËS SË NIVELIT TË LARTË PËR ALGORITMET DEEP LEARNING NË FPGA

Ruçi, Danjela

Master Shkencor, Departamenti i Inxhinierisë Kompjuterike

Udhëheqësi: Dr. Julian Hoxha

Për shkak të saktësisë së lartë që ofrojnë, rrjetet neurale CNN përfaqësojnë një model shumë të rëndësishëm për sistemet që identifikojnë dhe klasifikojnë imazhe. Sidoqoftë, një detyrë e tillë ka kosto të larta. Për këtë aryse, qëllimi kryesor aktual lidhet me implementimin e modeleve që ofrojnë shpejtësi të lartë dhe kosto të ulët. Procesorët GPU janë një alternivë e mirë kundrejt këtyre kërkesave, por ato nuk ofrojnë zgjidhjen më të mire të mundshme për shkak të konsumit të lartë të energjisë. Nga ana tjetër, modulet FPGA përshtaten më së miri me rrjetet CNN sepse konsumojnë më pak energji se procesorët GPU dhe kanë një arkitekturë fleksibël. Vështirësia për dizenjuesit e moduleve FPGA është përdorimi i gjuhës HDL për implementimin e rrjeteve CNN. HDL nuk është një platformë mbi të cilën mund të programohet; ajo përfaqëson kod të nivelit hardware për të përshkruar komponentët hardware-ikë, siç janë regjistrat. Sinteza e nivelit të lartë (HLS) mundëson përdorimin e gjuhëve të nivelit të lartë, siç është gjuha C apo C++, për të implementuar rrjetet CNN në FPGA, pasi është HLS ajo që kujdeset për konvertimin e kodit të shkruar në gjuhë të nivelit të lartë, në kod të nivelit hardware. Ky punim përfaqëson një përmbledhje të punës që është studiuar dhe zhvilluar deri tani për implementimin e rrjeteve CNN në FPGA duke përdorur HLS.

Fjalë kyçe: rrjetet neurale (CNN), FPGA, sinteza e nivelit të lartë (HLS), gjuhë përshkruese hardware-ike (HDL), konsumim i energjisë

Dedicated to my family.

ACKNOWLEDGEMENTS

First and foremost, I would like to thank God Almighty for giving me the ability and knowledge to successfully complete this research study. Without His blessings, this achievement would not have been possible.

I would like to express my deep gratitude to my supervisor, Dr. Julian Hoxha, for his continuous guidance and useful suggestions throughout all the process of learning and writing this master thesis.

I am also thankful to the committee members for their thoughtful comments and recommendations on this master thesis.

My acknowledgement would not be complete without thanking my family, the forever biggest source of my strength. I would like to thank them for their great love and their continuous support throughout this entire process. I will always be grateful for you being my family.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF ABBREVIATIONS

HLS	High-Level Synthesis
RTL	Register-Transfer Level
SoC	System on a Chip
ASICs	Application-Specific Integrated Circuits
FPGAs	Field-Programmable Gate Arrays
HDL	Hardware Description Language
IC	Integrated Circuit
IP	Intellectual Properties
ML	Machine Learning
AI	Artificial Intelligence
DL	Deep Learning
NN	Neural Networks
CPU	Central Processing Unit
GPU	General Purpose Processing Unit

CHAPTER 1

INTRODUCTION

1.1. Introduction

In the past decades, deep learning (DL) has demonstrated its effectiveness and efficacy in resolving several problems related to real-world issues. The reason why DL is so useful and needed is related to its possibility to automatically adjust itself to new circumstances, as well as learn and improve itself. Convolutional neural networks (CNNs) are considered to be the state-of-the-art for DL algorithms.

Nowadays, CNNs are being used in numerous fields for several actions, like classifying images or identifying and recognizing objects. CNN approaches usually operate on a cloud server. Nevertheless, there exists a need for introducing CNNs to embedded systems, due to IoT emerging each day more. Such a task is particularly required for systems that collect huge amounts of data in real time. While CNNs are continuously being implemented to complex challenges, there are some difficulties regarding to low performance, latency and power consumption that are present on integrated systems that have CPUs or GPUs.

GPU architectures, due to their good performance and memory space, are considered as one of the most efficient tools in terms of the improvement of CNNs processes like training and classification. Still, their power consumption, which is a crucial metric for evaluating the throughput, is large. ASIC architectures on the other hand, have reached better performance consuming less power, but the time and costs needed for implementing them is high (Chen, Krishna, Emer, & Sze, 2016).

FPGAs have many favorable characteristics, which make them the most promising architectures for accelerating CNNs hardware, like good performance and low power consumption at a reasonable cost. Their highly efficient and flexible architecture enables managing various computing algorithms by the same time they try to accommodate

the device's memory resources. FPGAs are programmable modules which, in terms of efficiency, provide countless benefits. They also have characteristics like high velocity and low power consumption, that make them a good option for machine learning applications. While NNs are being transformed to reach out to more industries, it is helpful to have the flexibility that FPGAs offer. CNNs require high computational techniques and FPGAs offer a reasonable compromise between three parameters: cost efficiency, performance and power efficiency. FPGAs are also beneficial as large quantities of computing are being moved to Cloud, since FPGAs can be modified to different requirements users have

FPGAs provide better performance compared to CPUs. Regarding to power consumption, FPGAs offer higher efficiency in comparison with both CPUs and GPUs. However, the long time required for designing have restricted the utilization of FPGAs. Lately, the HLS tools have provided an automatic "translation" from high-level languages, like C or C++, into hardware description languages (HDL).

1.2. Aim of the study

For computers, image comprehension is a hard action. However, the task of image classification has a great importance on several applications for systems used almost every day, like security or medical field. Over the last several years, great improvement has been achieved in this field. Nowadays, CNNs provide the most successful solution to image understanding and classification.

Numerous systems have been proposed for achieving CNNs implementations that have effective performance. While CNNs require high computational techniques, FPGAs offer a reasonable compromise between three parameters: cost efficiency, performance and power efficiency.

The aim of this study is to develop a better understanding on the High-Level Synthesis implementation on FPGA architectures for Convolutional Neural Networks. Hardware description languages (HDL), like Verilog and VHDL, are an option for realizing this implementation, but coding for complex NNs is extremely complicated since

they are at a low abstraction level. Fortunately, the use of High-Level Synthesis simplifies all this process.

1.3. Thesis Organization

This master thesis is organized in five chapters.

In the first chapter, a brief introduction and the aim of the study is given.

In the second chapter, it is given a theoretical background on machine learning, deep learning, neural networks as well as a detailed description on convolutional neural networks. FPGA architecture is also described.

In the third chapter, it is documented a detailed introduction on High-Level Synthesis and a description of the *hls4ml* package.

In the fourth chapter, the literature review related to this work is given.

In the fifth chapter, there are given conclusions and future work.

CHAPTER 2

INTRODUCTION TO MACHINE LEARNING AND DEEP LEARNING

In this chapter is given a brief overview of machine learning and deep learning, as well as a description on Neural Networks. A detailed explanation on Convolutional Neural Networks is also included, followed by a paragraph that describes the FPGA architecture. Lastly, some reasons why FPGA technology is ideal for Neural Network implementations are listed.

2.1. Machine Learning Background

Machine learning (ML) is an artificial intelligence (AI) technology which allows different machines to implicitly learn and develop from experience without being programmed. ML is focused on the creation and development of programs which access content and information with the aim of learning for themselves. In other words, the general goal of ML is to recognize the data structure and integrate it into models that people can understand and use.

Even though ML is a branch of computer science, it is different from traditional existing techniques for computing. Algorithms used in traditional computing are collections of coded instructions that computers use for calculating or solving problems. Machine learning algorithms on the other hand are designed to permit computers to analyze input data and utilize statistical analysis to generate results which are within a particular range. Therefore, ML allows computers and machines to build structures through sample data so that processes like decision-making are done automatically based on input data.

Machine learning is a field which is evolving constantly. It has helped almost every user of technology nowadays. The technology of facial recognition makes social

media sites help users tag or share friends' images. The technology of optical character recognition (OCR) transforms text images into movable forms. Based on users' interests and using ML algorithms, recommendation mechanisms suggest what shows or movies to watch next.

2.1.1. Methods of Machine Learning

Tasks in machine learning are usually divided into broad categories, which are based on the way how the learning process is obtained or how the system gets a feedback on the learning.

Two of the most commonly accepted methods of machine learning are supervised learning, that "teaches" algorithms based on human-labeled sample data for input and output, and unsupervised learning, that does not "offer" labeled data to the algorithm to help it identify structure in the input data (Tagliaferri, 2017).

1. *Supervised Learning*

   In this type of learning, example inputs which are marked with their ideal output, are introduced to the computer. The goal of this approach is to enable the algorithm to "learn" and adjust the model by making a comparison between the real output and the "taught" one. For this reason, supervised learning utilizes patterns on certain data that is unlabeled in order to predict label values. A typical scenario of supervised learning is the use of past data to foresee future events that are likely to happen: tagged images of dogs are used to identify untagged images of dogs.

2. *Unsupervised Learning*

   In unsupervised learning, the algorithm is responsible for finding similarities between the input data. Unlabeled data is the ideal type of input for unsupervised learning. The aim of unsupervised learning is simple: finding hidden patterns within the same dataset and allowing the machine to explore the required representations for automatic data identification. Unsupervised learning is widely used in data transactions.

2.2. Deep Learning

Deep learning (DL) is an important method of ML that tries to imitate the way that human brains transform light and sound input into vision and hearing output. A deep learning model is similar to biological neural networks, which consist of many layers. In order to obtain data features, DL requires processing layers in cascade, where the output of one layer acts as input for the following one. Algorithms in DL are either *supervised*, which help to classify data, or *unsupervised*, which do pattern analysis.

Deep learning algorithms collect the most data among all machine learning algorithms that are developed. They have also beaten humans in certain tasks regarding to cognitive tasks. Due to these features, DL is now a potential approach in the field of artificial intelligence.

2.3. Neural Networks

Neural networks (NN) could be considered as the human brain's machine implementation. Similar to the way human brains are trained to learn, complete different tasks and produce things, like differentiating cats from dogs, neural networks can also "train" or learn and complete a task. In Figure 1 is shown a schematic representation of a neural network (Rao, 2020).

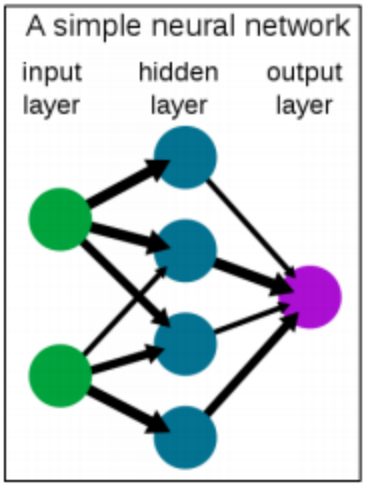

Figure 1. Schematic representation of a neural network

2.3.1. Neural network components

- *Neurons*: The spheres colored in green, blue and purple in Figure 1 represent the "neurons". Their goal is to process data which is transmitted to the neural network.

- *Layers*: There are at least three layers in a simple neural network: one input layer, one hidden layer and one output layer. More complex and deeper networks might have many hidden layers.

- *Weights and Biases*: The weights are represented by the arrows shown in Figure 1. They define the relative proportion of the relation among neurons. The mathematical relation that exists between the input, weights, biases and activation functions within a neural network are shown in Equation 1.

$$output = f(\sum(weight * input) + bias) \qquad \text{(Equation 1)}$$

where *f* is the activation function.

- Activations: The activation functions consist of tanh, sigmoid, softmax and others. These functions help in maintaining the values in the neural network within a limited range.

a) Sigmoid activation function gives an output that varies from 0 to 1, for any given input (as shown in Figure 2).
b) Tanh (hyperbolic tangent) activation function generates an output that has values from -1 to 1, for any given input. In comparison to the Sigmoid function, it has a steeper gradient (as shown in Figure 2).
c) Softmax activation function, in case of a network model that is focused in classification, produces a probability distribution as output (as shown in Figure 3).

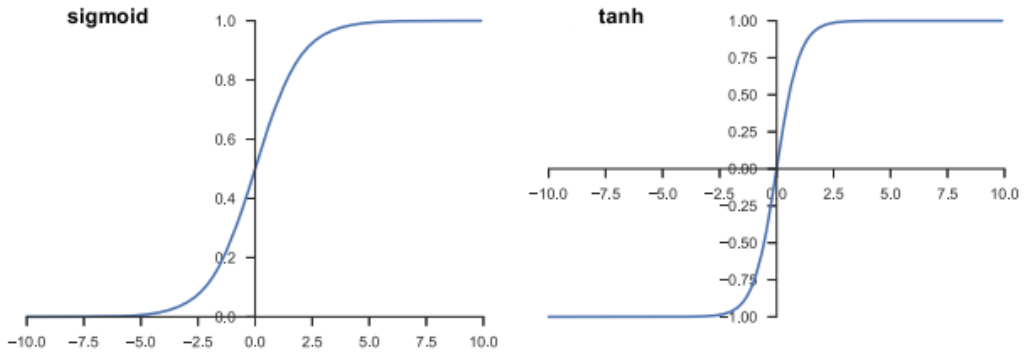

Figure 2. Sigmoid and Tanh activation functions

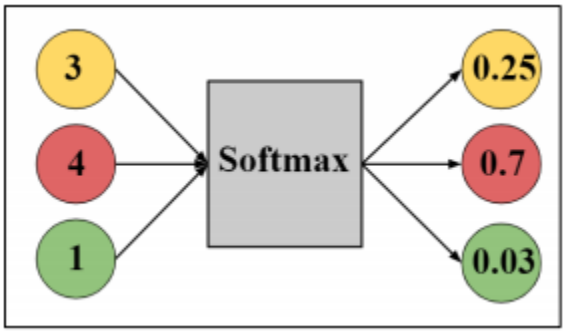

Figure 3. Softmax activation function

2.3.2. Training and Inference

One of the most effective approaches for neural network trainings is to provide it with a dataset, each input of which goes along with the expected output. For instance, a set of data that is composed of classified images of cats and dogs can be provided to a

network. The images of the dataset go from the input layer of the NN to the output layer of the NN, going via all the hidden layers. The value obtained in the output is compared to the expected value that has been made known to the network. An error that must be minimized to support the neural network foresee values as close as possible to the expected values, is calculated. It is of a great importance to emphasize that a NN is a mathematical model and the layers can communicate within only via numbers (Rao, 2020).

A common algorithm used in training NN is the *backpropagation algorithm*. This algorithm calculates a cost function (error function) depending on the comparison between the expected output and the obtained output, which is then utilized to modify the weights and the biases so that, for a given input, the obtained output is similar to the expected output. The method of going back to adjust the weights and the biases is called *backpropagation*. Its aim is to minimize the error as much as possible. If the NN is trained, the algorithm's prediction accuracy can be tested through the *inference process*. In the previous example, an unlabeled dog/cat image is given as input and it is tested the accuracy of output prediction.

2.3.3. Types of Neural Networks

There exist several categories of neural networks. Specific NNs deal with specific applications and operate with specific datasets. Many use images as inputs, whereas many others may choose a series of inputs.

The types of NNs are:

1. Deep Neural Networks (DNN), which have two or more hidden layers (more than 1).
2. Recurrent Neural Networks (RNN), which are helpful in context-relaying data prediction, like text generation.
3. Convolutional Neural Networks (CNN), whose ideal inputs are images.

2.4. Convolutional Neural Networks

A convolutional neural network is a category of the deep learning design that is responsible for analyzing data with grid patterns, like image data.

A CNN is a mathematical model which usually consists of three layer types: convolution layers, pooling layers and connected layers (Yamashita, Nishio, Do, & Togashi, 2018). The pooling and convolution layers are responsible for feature extractions, while connected layers are responsible for mapping the features extracted into a final output. The convolution layer has a crucial function in CNNs, which in fact consist of a stack of mathematical functions, like the convolution function, a specific linear operation.

2.4.1. Architecture of CNNs

As mentioned above, the architecture of CNN has many blocks: convolution layers, pooling layers and fully connected layers (Yamashita, Nishio, Do, & Togashi, 2018). A standard CNN architecture is composed of many layers of convolution followed by a pooling layer, which are then followed by one or many fully connected layers, as presented in Figure 4. The phase in which the data input is converted into an output via these layers is named *forward propagation*. Based on forward propagation and a loss function, the performance of a design under specific kernels and weights is determined.

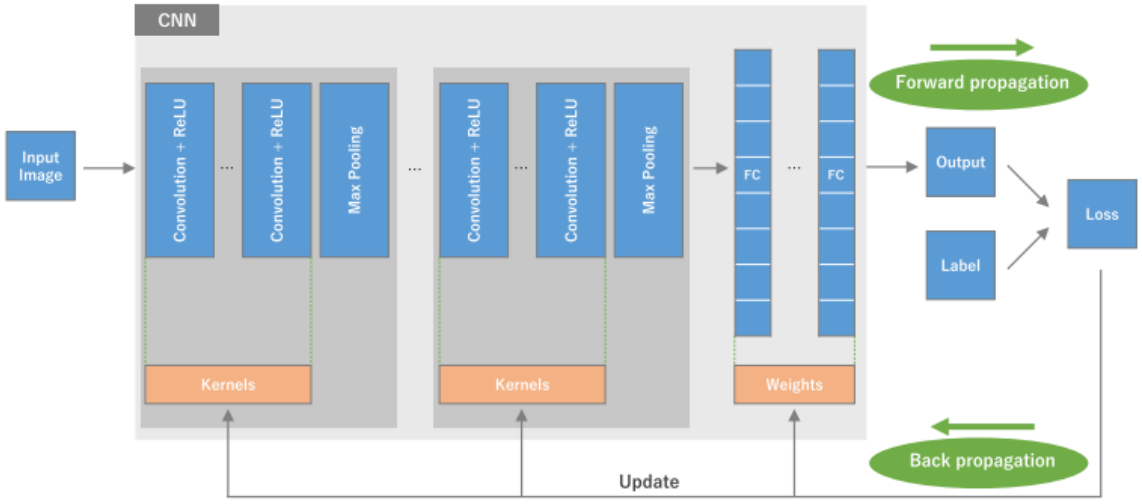

Figure 4. A visual representation of CNN architecture building blocks

2.4.1.1. Convolution layer

The convolution layer is a crucial element of the CNN architecture that extracts features, usually consisting of a combination of operations that might be linear (convolution operations) or nonlinear (activation function).

- *Convolution operation* is used for extracting features, with the application of a kernel (which represents a small array of numbers) in the input (which represents a numbers' array, called a tensor). An element-wise production is computed for each component of the kernel and the input tensor; and the value of the output is calculated as the sum of those products at the corresponding locations of the output tensor, known as a feature map. This process is illustrated in Figure 5 (Yamashita, Nishio, Do, & Togashi, 2018) and is redone with the use of multiple kernels to create an arbitrary amount of feature maps representing various input tensor characteristics, so that different kernels are seen as different extractors for the features.

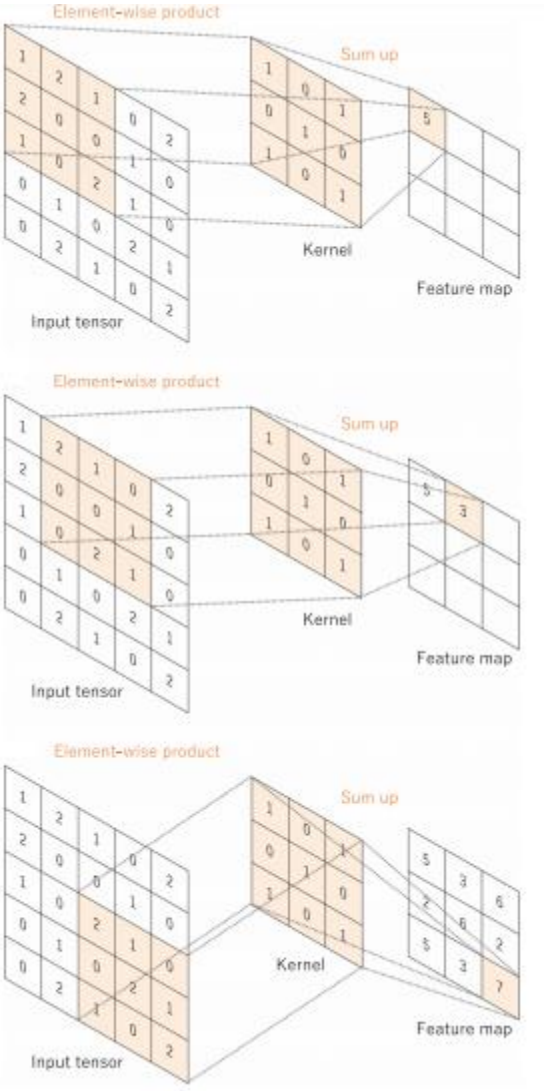

Figure 5. An illustration of the convolution process

There are two important hyperparameters that characterize the convolution process: the number of the kernels and their dimensions. The typical kernel size is 3 × 3, but it might also be 5 × 5 or 7 × 7. In the reported convolution operation, kernels' centers do not overlap the outermost input tensor element. The dimensions of the feature map are reduced in contrast to the input tensor. Better result is received by applying the zero-padding technique (columns and rows with 0s are added around the input tensor), as illustrated in Figure 6.

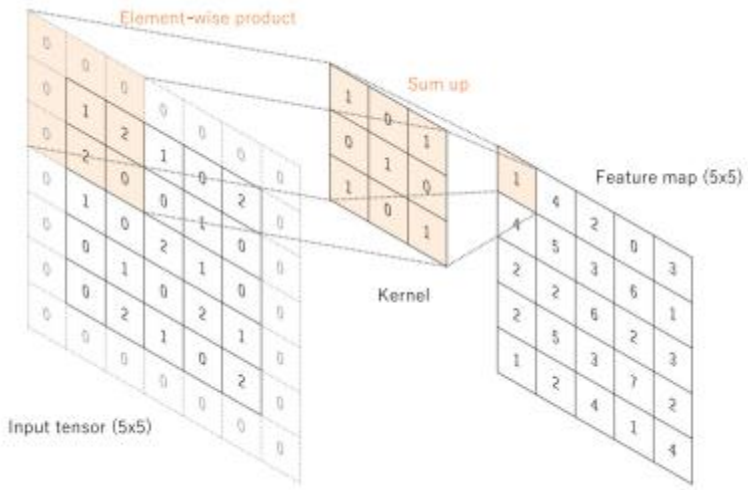

Figure 6. Convolution operation using the zero-padding technique

- *Nonlinear activation function* receives as input the output of the linear operation. The most used activation function is the ReLU (rectified linear unit). It calculates $f(x) = \max(0, x)$ and is represented in Figure 7. Other activation functions are the sigmoid function and the hyperbolic tangent (tanh) function, which are shown in Figure 2.

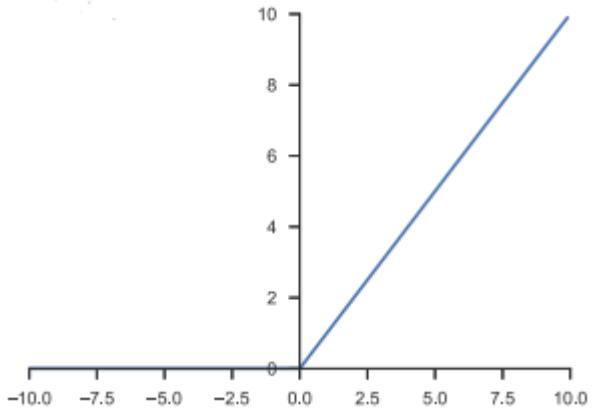

Figure 7. ReLU nonlinear activation function

2.4.1.2. Pooling layer

A pooling layer offers a standard subsampling procedure which decreases the dimensions of the feature maps, so that a translation invariance is added to minor shifts and the amount of corresponding trainable parameters is reduced.

The most common method of pooling processes is *max pooling*. Its job it to extract patches from the feature maps of the input in order to output the greatest value of each patch discarding other values. Figure 8 shows the process of max pooling. A widely used max pooling is the one with a filter of dimensions 2 × 2.

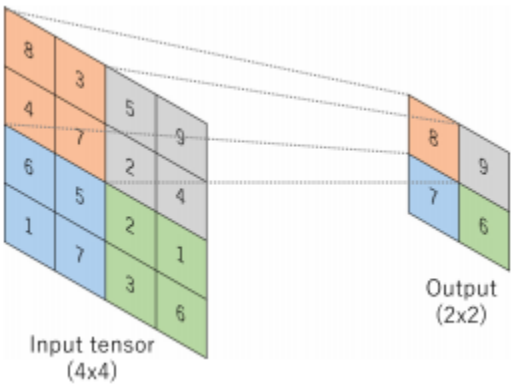

Figure 8. An example illustration of max pooling

2.4.1.3. Fully connected layer

The output of the convolution and the pooling layer, which are represented by feature maps, are usually flattened and linked to at least one fully connected layer. In every fully connected layer, each input has a corresponding output, to which it is connected by a trainable weight. As soon as the convolution layers' features are extracted and pooling layers' features are produced, they map with the resulting outputs of the neural network. A nonlinear function (sigmoid, tanh or ReLU) follows every fully connected layer, except from the last layer: its activation function is softmax (shown in Figure 3) and is different from activation functions that follow all the other layers.

2.4.1.4. Cost function

A cost function is also known as a loss function and evaluates the accuracy among the neural network's output predictions based on forward propagations. A widely used cost function is cross entropy.

2.4.1.5. Forward propagation and Backpropagation

The *forward propagation* represents the prediction phase. After the input data is read and processed by the network, an output value is obtained. The mathematical operation behind this process is given through Equation 1.

The *backpropagation* represents the training phase. In order to obtain a better performance, the network updates its parameters by making comparisons between the predictions or outputs of the network and the true values.

2.5. Field-Programmable Gate Arrays (FPGAs)

FPGAs are electronic devices that consist of an array which has customizable logical blocks. These logic blocks are linked through programmable connectors, that consist of packets of wires which run among the logic pieces in a vertical and in a horizontal way. New FPGA versions provide thousands logical blocks that can be configured, and at the same time, they have an amount of hardened operational modules that permit specific operations to be implemented quickly and effectively (Gschwend, 2020).

CNNs provide greater levels of accuracy compared to existing algorithms. However, they need enormous quantities of computing resources as well as memory access because of the huge amounts of parameters required by the convolution process. This leads to large power consumption and as a result, a computing obstacle for the CPUs (General Purpose Processors) (Hassan & Mostafa, 2020). As a consequence, hardware accelerators, like GPUs, FPGAs and ASICs, are being used in order to optimize the CNNs performance.

2.5.1. FPGAs over GPUs

General-Purpose Processor (GPU) architectures, due to their good performance and memory space, are considered as one of the most efficient tools in terms of the improvement of CNNs processes like training and classification. Still, their power consumption, which is a crucial metric for evaluating the throughput, is large.

2.5.2. FPGAs over ASICs

ASIC architectures have reached better performance than GPUs consuming less power, but the time and costs needed for implementing them is still high (Chen, Krishna, Emer, & Sze, 2016).

2.5.2. FPGAs and Neural Networks

FPGAs remain one of the most important technologies regarding to the CNN implementation since they offer a good performance and consume less power at a reasonable cost. Their highly efficient and flexible architecture enables managing various computing algorithms by the same time they try to accommodate the device's memory resources. FPGAs are programmable modules which, in terms of efficiency, provide countless benefits. They also have characteristics like high velocity and low power consumption, that make them a good option for machine learning applications. While NNs are being transformed to reach out to more industries, it is helpful to have the flexibility that FPGAs offer.

In the deep learning field, FPGAs are favored due to their task of inference. The training process helps a NN to determine the collection of weights and biases to best connect specific sets of inputs to their corresponding sets of outputs. The inference process is responsible for predicting an output based on the weights and biases, which are defined in the training process. While CNNs require high computational techniques, FPGAs offer a reasonable compromise between three parameters: cost efficiency, performance and power efficiency. FPGAs are also beneficial as large quantities of computing are being moved to Cloud, since FPGAs can be modified to different requirements users have (Majumder & Bondhugula, 2019).

With so many advantages that the implementation of neural networks or machine learning algorithms on FPGAs has, a question arises: what is the best way to do this implementation? Hardware description languages (HDL), like Verilog and VHDL, are an option, but coding for complex NNs is extremely complicated since they are at a low abstraction level. Fortunately, the use of High-Level Synthesis simplifies all this process.

CHAPTER 3

HIGH-LEVEL SYNTHESIS DESIGN

In this chapter it is given a short paragraph on Synthesis, which is then followed by a theoretical background on the High-Level Synthesis basics and benefits. Later in the chapter is described the *hls4ml* package and what Xilinx offers.

3.1. Introduction

The synthesis is a task which specifies the required behavior of a system, as well as the collection of objectives and constraints to satisfy. By *behavior* we refer to the way that the system and its elements cooperate with the environment they are in. Synthesis can occur at different levels of abstraction since structures can be represented in different levels of detail. By s*tructure* we refer to the collection of the elements that are interconnected to create the whole system. (McFarland, Parker, & Camposano, 1998). To recognize a specific behavior, there can be used various structures.

One of the main responsibilities that synthesis has, is to discover the structure that satisfies the best the constraints, such as power, cycle time limitations and area, while the other costs are significantly reduced. The aim might be, for example, to mitigate area while trying to attain the processing rate at its minimum.

There are different types of synthesis, each of which operates at a different level on the design hierarchy.

3.2. High-Level Synthesis

High-Level Synthesis (HLS) is an automated design process that will take an algorithmic definition as its input, so that it can construct the digital hardware that will implement the required function, (McFarland, Parker, & Camposano, 1998). For this reason, HLS is also called the *algorithmic level* of the design hierarchy. These algorithms are written in programming languages that are high-level languages. This level uses integer or/and bit strings and arrays as primary data types, rather than Boolean variables. The specifications of the input provide the necessary mappings from input sequences to output sequences.

There has been a movement towards automated synthesis in recent years, which aims to place the automated synthesis into higher levels of the design hierarchy. There are some considerable improvements that the implementation of HLS offers:

1. *Shorter design cycle*

   The more the design process is automated, the faster the company is able to reach the design's market window. In addition, the automation of processes reduces costs significantly, as much of the chip's cost is being developed during design.

2. *Less errors*

   By the time that the synthesis process is validated as correct, the probability that the final design will respond to the initial specification is very high. For new chips, this means less errors and less time for debugging.

3. *Self-documentation of the design process*

   A special feature of an automated system is that it keeps record of the decisions made and the reasons why, as well as the effects that those decisions had.

3.2.1. Historical background on HLS

HLS systems were first developed in the 1900s. The early generations of these systems resulted into a failure. However lately, due to the following reasons, we are seeing a progressively rising demand for innovative HLS solutions (Cong, et al., 2011):

1. *Nearly every SoC seems to have embedded processors*

   With the implementation of multiple micro-processors, memories and other units on a single chip, more and more software components are being included in the development of new embedded systems and devices. An automatic HLS flow helps designers to define the features of designs for embedded systems as well as SoC hardware logic in high-level languages, like C or C++. Thus, they can rapidly test various boundaries regarding hardware and software and explore trade-offs between area, power rand performance.

2. *A greater level of abstraction is required for the capacity of Huge Silicon*

   Abstraction of design is one of the most efficient strategies to control the complexity and to improve the performance of the design.

3. *The productivity of designs is improved by the reuse of behavioral IP*

   Behavioral synthesis has reduced the line-count in design parameters and at the same time, has the extra value of enabling an effective reuse of intellectual properties (IPs). Unlike RTL IP that has both well-defined interface protocols and micro-architecture, behavioral IP can be replaced by various implementing technologies and/or system requirements.

4. *Acceptance of high-level requirements is driven by verification*

   Transaction-level modeling (TLM) with C or C++ seems to be a common method for system-level verifications.

An increasing amount of FPGA and ASIC models and designs are being developed via HLS tools. This happens for two main reasons, which are listed as follows:

1. Less formal verification pressure is required when using HLS tools than in normal integrated circuits.
2. HLS tools are ideal for synthesis based on platforms as they help in achieving a higher quality of results (QoR).

Compilers for languages of high-level have had great success in practice since back to 1950s. The concept of automatic generation from high-level behavioral

requirements of circuit implementations rises inevitably with the growing complexity of integrated circuits (ICs) designs. Around the 1980s, the first efforts on HLS mostly consisted in research projects, where several tools were designed to attract attention to the technique and to try and work with different algorithms. However, almost all those tools made fairly simple predictions about the targeted platform and were not commonly used. Early marketing campaigns in the 1990s attracted significant interest among designers, but they struggled to achieve widespread acceptance due to low QoRs. Later HLS efforts have strengthened the use of it by increasing coverage of the input language and incorporation of the platform, as well as enhancing QoRs.

After 2000, both industry and academia have built a new generation of HLS tools (Cong, et al., 2011). Almost all these HLS tools concentrate on using C or C++ languages for design intent capturing. Compared to previous tools which recognize only HDL languages, the use of C or C++ languages makes the designers of algorithms and systems more open HLS tools. However, due to the fact that the C and C++ languages have complicated structures, like pointers and dynamic memory, that contribute to difficulty in synthesis, there have been continuous discussions on whether they are the best choice for HLS.

3.3. Methodology of HLS

HLS, usually referred to as *behavioral synthesis*, automatically compiles a high-level description of a design into a Register Transfer Level (RTL) implementation which matches several design constraints defined by the user. In comparison to RTL, the definition of HLS design is 'high level' in two aspects: design abstraction and specification language (Sun, Wang, Yin, Cavallaro, & Ly, 2012):

1. *High level of abstraction*

   The input of HLS is usually a design specification of the data flow. This level is higher than RTL since it gives the HLS tools the freedom to determine which tasks

to complete in each cycle of the clock and does not define one exact behavior at a cycle.

2. *High level of specification language*:

   The input of HLS is defined in languages such as C, C++, Java, Python or Matlab, allowing so the use of advanced data structures such as loops, arrays, classes, inheritance, overloading, polymorphism and so on. RTL, on the other hand, is represented by description languages.

RTL models produced by HLS are generally not human-readable. Despite this fact, there is a collection of reports provided by HLS tools that express in a quantitative way the predicted performance, timing and use of resources of RTL design. These reports are critical, and sometimes the only evidence in which designers are based on to change models and designs to produce more favorable results (Dai, et al., 2018).

3.3.1. Objectives of HLS

HLS's goal is not only to execute the description of the input, but also to discover parallelism from it and create a faster and cheaper small (micro) architecture. The micro architecture includes a datapath which is pipelined, as well as a description of the way data is processed through the datapath.

The HLS's output might consist of (Sun, Wang, Yin, Cavallaro, & Ly, 2012):

1. *Implementation of RTL*

   It contains the RTL netlist that includes the datapath, I/O, host and memory interfaces, the control logic and the libraries, scripts and synthesis constraints that are needed to synthesize the netlist of RTL.

2. *Feedback of the analysis*

   It contains GUI and performance bottleneck reports, high-level mapping of the source code to RTL, costs of hardware and so on. The aim is to help users in understanding and improving the micro architectures.

3. *Artifacts verification*

   It contains simulations on the reference model (test bench) to encourage users build and debug high-level language tests and use them again to verify RTL.

3.3.2. Constraints of HLS

There are some limitations/constraints which are user-specified, that help HLS create the wanted micro architecture. These limitations include (Sun, Wang, Yin, Cavallaro, & Ly, 2012):

1. *Target hardware*

   It contains all the tools that the design is developed for, such as the platform, frequency of the clock and the library of the technology. This information is used by the HLS to predict the timing of the sub-cycle and the datapath costs.

2. *Constraints on performance*

   These limitations consist of sampling rate of the input, production rate of the output, latency from input to output, intervals for loop initiation and latency of loops.

3. *Memory architecture*

   It defines how memory and its interfaces are mapped to multi-dimensional arrays, enabling HLS to build micro architectures that consist of multi-bank, multi-port and memories: internal and external.

4. *Constraints on interface*

   This contains the necessary logic to develop ports, protocols and handshake process on every host, input, output, and external memory interface. HLS produces the elements mentioned above in the netlist of RTL. This way, it is easily combined with different hardware components.

5. *Hierarchy of the design*

It helps to divide a design by implementing the concept of hierarchy in the description of the high-level input. This way, HLS can handle the complexity of the design by divide-and-conquer.

3.3.3. Benefits of HLS

To identify the ideal micro architecture behind a range of constraints, HLS is designed to explore through various algorithms and architectures. The key benefits of HLS arise from its use of languages of high abstraction and high specification level (Sun, Wang, Yin, Cavallaro, & Ly, 2012):

I. The advantages of modeling/designing at a high abstraction level:
   a. Authorizes focus to design essential features. Easy to explore other architectures.
   b. Easy to assess changes or modifications on algorithms.
   c. Easy to create the interfaces of memory, I/O and host, and the logic behind the pipeline and handshake.
   d. Easy to retarget constraints or performance of various hardware from the same description of the input.

II. The advantages of confirming at a high abstraction level:
   a. Easy to debug and test input definition features.
   b. Rapid simulation and available for free.
   c. Test suite can be reused to verify the RTL.
   d. Easy to obtain the coverage of the code.

III. The advantages of high-level specification language:
   a. Reuse of the existing code for confirmation and for design.
   b. The possibility to use different tools to develop software (such as Visual Studio).
   c. Offers various features of the advanced language, such as polymorphism.

The main benefits that HLS provides are listed as follow (XILINX, 2020):

1. Greater efficiency for hardware designers

Designers of hardware components can work on higher abstraction levels focusing on achieving a hardware high-performance.

2. Greater system efficiency for software designers

   Designers of software can optimize their algorithms' intensive parts on FPGAs.

3.4. HLS for FPGAs

FPGAs can generate circuits which have millions of memory units only for computing. The architecture of FPGA is adaptable, allowing greater optimizations for improved throughput. FPGAs consume less power and are considered efficient for embedded applications. The difficult part for FPGA architectures is implementing ML systems that are written in languages of higher level, like Python. HDL is not a platform on which to program; it is simply code to describe components of hardware like registers and counters. The main problem that arises is the lack of a direct translation from a high-level language to HDL.

An option to "fix" the issue of programming is by using HLS tools to build and develop programs for deployment in HDL. HLS tools permit the designers to use high-level languages instead of writing HDL code. So, HLS offers the ability to transform the descriptions of untimed software into optimized hardware designs that are cycle-accurate. As a result, it was identified as an important way to enhance the hardware design productivity. Through HLS, designers do not need to continuously deal with the details of low-level hardware description language (HDL) anymore. Instead, they can concentrate on choosing the best tradeoff between algorithm and microarchitecture from a single source of software design.

A significant distinction between the HLS tools of the current generation and their predecessors is that several tools are developed aiming FPGA implementation. In recent times, FPGAs have had a continuous improvement in speed and capacity, making

them an excellent platform for signal processing and communication applications. For this reason, several HLS tools are explicitly designed for FPGAs.

HLS tools' role in FPGA is presented in Figure 9. They translate the code written in a high-level language such as C, C++ or SystemC into HDL code. HDL code allows the RTL synthesis into a digital circuit, to finally deploy it on an FPGA.

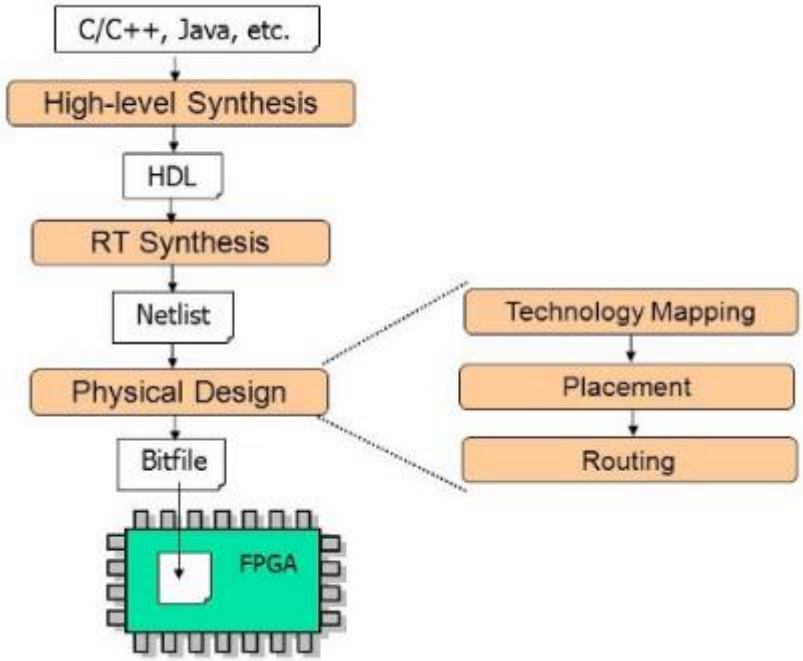

Figure 9. The role of HLS in FPGA systems

3.4.1. HLS Phases

The High-Level Synthesis process consists of three main phases (XILINX, 2020):

1. *Scheduling*

   This step decides the operations that will happen during every clock cycle relying on three components:

   - Clock cycle length
   - The needed time for completing an operation

- Optimization instructions specified by the users

For a longer duration of the clock, more operations are performed within a clock cycle and all operations could be performed in one clock cycle. On the other hand, for shorter durations of the clock, more clock cycles are required to perform all the operations. In other words, the high-level synthesis makes an automatic schedule for the operations to complete.

2. *Binding*

The binding phase specifies the hardware resources that will implement every scheduled operation.

3. *Extraction of control logic*

This phase makes the extraction of the control logic in order to construct a finite state machine (FSM) which will sequence each operation in RTL.

3.4.2. Synthesis of C code

The way in which HLS makes the C code synthesis is described as follows (XILINX, 2020):

- Synthetizing function arguments of highest level into RTL input/output ports
- Synthetizing C functions into blocks to fit in the hierarchy of RTL

  For C codes that contain a sub-functions hierarchy, the resulting RTL architecture will include a hierarchical structure of models and entities which will have a one-to-one relationship with the initial hierarchy of C functions.

- Keeping as *rolled* by default the loops in the C functions

  Once the loops are rolled, the logic for one loop iteration is generated by the synthesis process and the design of RTL uses this logic for all other iterations. Loops can also be unrolled with the use of optimization directives that will allow a parallel execution of all iterations. Also, loops may be pipelined by implementing a finite state machine.

- Synthetizing the C code arrays into RAM blocks for the final design of FPGA

  Arrays which are on the interface of the top-level, are implemented as access ports to RAM blocks that are located outside the design.

Depending on the defined default actions, constraints and optimizing directives, HLS generates an efficient implementation. Optimizing directives may be used in order to change and manage the default actions of the inner input/output ports and inner logic. This helps in the generation on different variants of the hardware architecture using a single C code.

To decide if the proposed architecture satisfies all the criteria, the HLS performance metrics may be obtained in the form of a *synthesis report* (XILINX, 2020). This report includes details for the metrics of performance that are listed as follows:

- Area: represents the quantity of hardware tools which are needed for implementing the design depending on the provided resources in FPGA.
- Latency: represents the amount of clock cycles needed for each value in the output to be calculated by the function.
- Initiation Interval (II): represents the total amount of clock cycles until the operation could allow new incoming input.
- Loop iteration latency: represents the total amount of clock cycles required for the completion of a loop iteration.
- Loop initiation interval: represents the total amount of clock cycles until the following loop iteration begins the data-processing.
- Loop latency: represents the total amount of clock cycles needed for executing all loop iterations.

3.5. The concept of hls4ml package

hls4ml is a software package to create HLS implementations of neural networks. The purpose of hls4ml package is to make an easy and accurate translation for machine

learning designs. In other words, hls4ml package is responsible for automatically converting a trained neural network into HLS code. The generated HLS design can either be used to develop an IP that can be inserted into more complicated structures, or can be used to build a kernel for CPU co-processing (Duarte, et al., 2018). Figure 10 illustrates a scheme of a typical workflow.

The section of the workflow evidenced in red shows the workflow of the regular software that is needed for a particular task to build a neural network. Before setting up a final model, this normal workflow includes a training phase and potential compression steps with tools like Keras and PyTorch. The blue part of the workflow represents the hls4ml section that transforms a design into HLS code which can later be run on an FPGA.

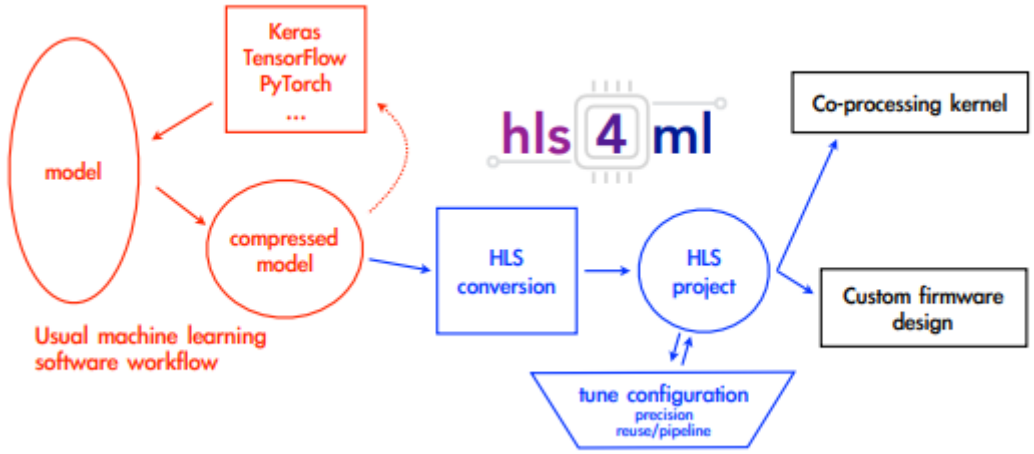

Figure 10. hls4ml workflow for translating a model into a FPGA implementation

At a higher level, the algorithm of FPGA design is unique from CPU programming, allowing individual operations to run in full parallel. This way, FPGAs can complete billions of instructions per second at lower power cost respective to CPUs and GPUs. But such operations use devoted resources and therefore they cannot be remapped in a dynamic way while running. The aim in designing an optimized implementation on FPGA is to balance the use of FPGA resources with reaching the desired algorithm's latency and throughput objectives. Important metrics for implementing FPGA include (Duarte, et al., 2018):

1. *Latency*: the overall time represented in units of "clocks" that is necessary to complete a particular iteration of the algorithm.
2. *Initiation interval*: often referred to as "II", expresses the amount of clock cycles needed before a new input is accepted by the algorithm. The initiation interval and the inference rate are inversely proportional of each other. As a result, input can be pipelined into the algorithm with a frequency of the initiation interval.
3. *Resource usage*: categories of FPGA resources are as follow: memory of onboard FPGA (BRAM), blocks of digital signal processing (arithmetic DSPs), registers and programmable logic (flip-flops and lookup tables).

The hls4ml tool consists of a variety of parameters that are configurable and might help users to discover and optimize their applications' latency space, initiation interval and usage of resources. The aim of hls4ml package is to enable the user to achieve this optimization via the FPGA design iteration and the translation of neural networks. Practically, the time needed to accomplish the translation of neural networks via hls4ml is reduced compared to the necessary time to develop a particular neural network model on an FPGA. At the same time, it can be used to quickly test machine learning (ML) algorithms lacking the need for dedicated support for the implementation of FPGA.

First, some terms and principles for deep and completely connected neural networks are introduced. In Figure 11 it is shown a neural network which has M layers. Each layer m has N_m neurons.

The input layer has N_1 input neurons and the output layer has N_M output neurons. The values of the output neurons vector are defined by x_m on each layer. For the m^{th} layer which is fully connected ($m > 1$) (Duarte, et al., 2018),

$$x_m = g_m (\boldsymbol{W}_{m,m-1} \boldsymbol{x}_{m-1} + \boldsymbol{b}_m) \qquad \text{(Equation 2)}$$

where $\boldsymbol{W}_{m,m-1}$ represents the weights matrix among layer m and layer $m-1$ of dimensions $N_m \times N_{m-1}$, $\boldsymbol{b}_m$ represent the bias values and g_m represents the function of activation for layer m. $N_m \times N_{m-1}$ multiplications are required to calculate the values of layer m neurons.

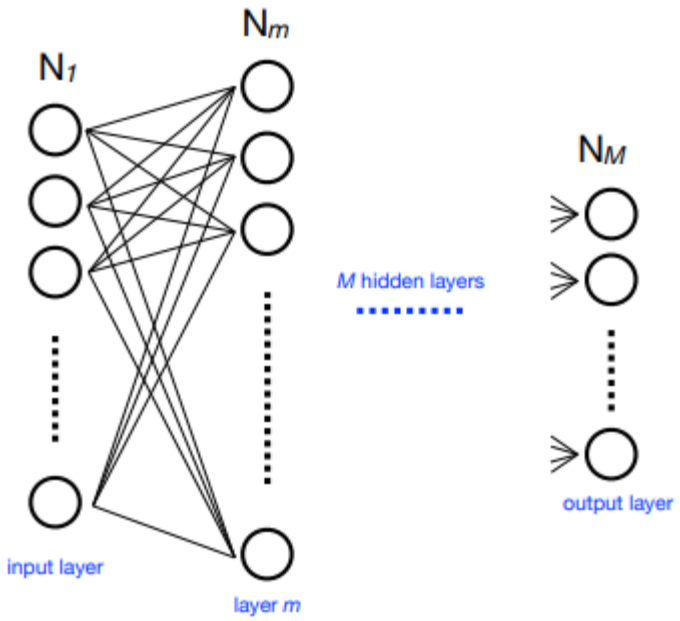

Figure 11. A visual representation of a deep neural network

In hls4ml, each layer x_m is calculated in an independent and sequent way. Following its initiation interval, the inference pipelined and allows a new set of data as its input. The overall number of multiplications that a certain neural network requires to infer is:

$$N_{multiplications} = \sum_{m-1}^{M} N_{m-1} \times N_m \qquad \text{(Equation 3)}$$

For a number of input values, activation functions that are non-trivial like hyperbolic tangent, sigmoid and softmax are precomputed and stored in BRAMs (Duarte, et al., 2018). The activation function of ReLU is applied in programmable logic. For any specified task, the impact of neural networks focuses on latency, throughput and usage of resource.

3.6. Vivado HLS

Vivado HLS (VHLS) is among the most known compilers for high-level synthesis. Produced by Xilinx Inc., it allows the designers to use features of high-level programming languages (even OO programming), to write the wanted codes. VHLS instantly converts the codes into languages of low level of abstraction, that describe features like registers, counters or state machines. The compilation process is usually affected by the use of scripted compiler instructions, which are nothing but VHLS explicitly interpreted meta-instructions. By default, the instructions are programmed to be completed simultaneously and fast. The role of Vivado HLS is shown in Figure 12.

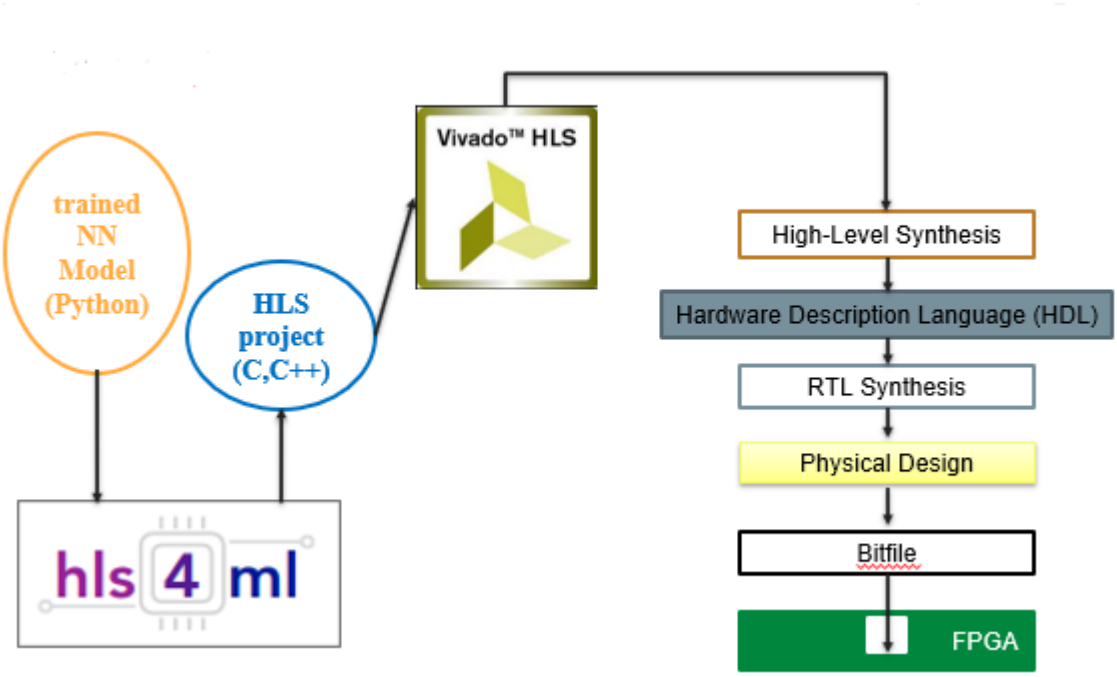

Figure 12. The "role" of Vivado HLS tool in the Synthesis process

CHAPTER 4

LITERATURE REVIEW

This chapter summarizes the literature review regarding to CNN implementation on FPGA architectures with the use of HLS. Several papers and articles have been studied, and the main ideas and results for the work done so far are included.

4.1. Methodology

The methodology used to conduct this master thesis is the review methodology. Several papers and articles are studied in order to develop a better understanding on High-Level Synthesis implementation on FPGA architectures for machine learning algorithms, especially for neural networks.

4.2. Related work

This section presents a description on the most recent work done for the CNN implementation on FPGAs.

The usage of AutoESL's AutoPilot HLS tool coupled with domain-specific system-level implementation developed by Xilinx is presented by (Cong, et al., 2011). The aim was to demonstrate the effectiveness of C-to-HLS synthesis solutions targeting multiple application domains. An experiment on a sphere decoder is done and the results show that the HLS solution can achieve 11-31% reduction in FPGA resource usage with improved design productivity compared to hand-coded design.

A model is designed by Microsoft for the CNN acceleration using many cards of FPGAs by (Ovtcharoc, et al., 2016). This design employs a controller of the highest

level to monitor the data flow with the use of a memory adapter. It contains several buffers for input data, one buffer for weighting the kernel, a huge collection of arrays of processing elements (PEA) and a section for redistributing the data. By using a DMA channel, it inserts the information from the PC storage into the buffers. The PEA blocks are used for calculating the dot product between the values of source buffer and the values of the weight buffer. The output of the dot product calculation is stored in the following input buffer. A schematic view of this model is given in Figure 12, whose main features are: (1) a software machine that can be customized, which at runtime could accept different layer configurations, (2) an effective mechanism which buffers data and a network for on-chip re-distributing, which reduces transmission traffic to off-chip storage, (3) a collection of processing elements (PEs) which are spatially spread and could be quickly scaled up to millions of units. This CNN accelerator can accept an image as input and process several layers of convolution, which are then transmitted to each PE array. In the end, the collected results are delivered to a dedicated NoC (network on chip), whose job is to redistribute the outcomes towards the input buffers for the following stage.

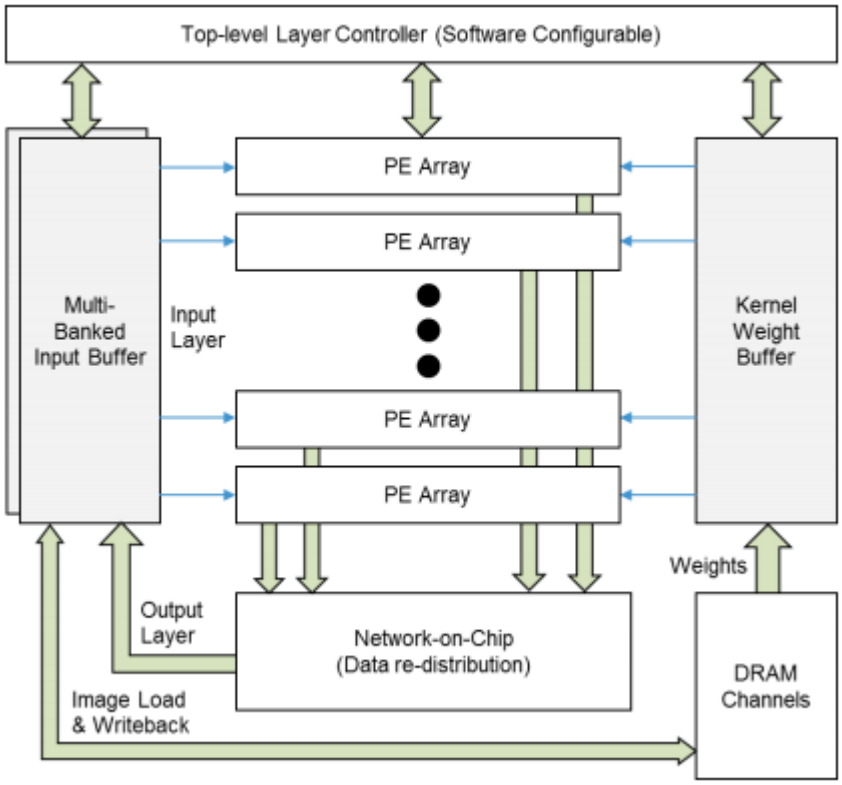

Figure 13. Design of the high-level of the CNN Accelerator

The model implemented by Microsoft is studied further by (Gschwend, 2020), by developing ZynqNet, which aims to make it operate for the training and the inference stages. ZynqNet is designed for SoC structures and not for server solutions and it contains two important elements: *ZynqNet CNN*, which is a strongly efficient and adapted CNN topology, and *ZynqNet FPGA Accelerator*, which is a design based on FPGA architecture used to evaluate the ZynqNet CNN. Even though the suggested approach tends to have some constraints because of a solely implementation via Cbased HLS, it appears to be optimistic. To handle the input data, the model utilizes a CLB (Circular Line Buffer), and to obtain data via the central memory, it utilizes an interface which is memory-mapped.

Two key accelerating techniques are introduced by (Hassan & Mostafa, 2020): (1) parallelism of layer resources and (2) pipelining within some layers. The implementation of CNN is done using the Xilinx SDSoC framework which includes the FPGA and the processor on a single board. The approach presented in this report aims to

reach a satisfying balance between the area, the speed and the design time, which is achieved.

A Long Short-Term Memory (LSTM) network targeting FPGA is implemented by (Rao, 2020). A LSTM model is first translated to HLS code, which is then given as input to an HLS tool to obtain reports and analyze the overall latency and resources required by this model. The LSTM model was successfully implemented, and further verification steps were taken. The first verification involved the verification between the KERAS LSTM model versus the HLS model generated by the framework. The second verification was performed by the HLS tool – Vivado HLS, that confirmed the model functionality between HLS and RTL.

A generic CNN accelerator for SoC is introduced by (Bjerge, Schougaard, & Larsen, 2020), who aimed an accelerated inference for various DL networks on a SoC structure. The given accelerator provides a flexible architecture that includes the HLS implementation via SystemC. It is capable of accelerating any Python-exported CNN and encourages a mixture of the CNN layers (convolutional, max-pooling and fully-connected). The approach was tested on a Xilinx Ultra96 frame and the used CNN is VGG16. In comparison with previous models, the outcome provided by this accelerator provided high precision during the training process. The inference was performed in 2 seconds while consuming an average amount of power, equal to 2.63W.

Machine learning models are implemented on HLS by (Dai, Zhang, Ustun, Young, & Zhang, 2018). In addition to Figure 9, Figure 13 shows the FPGA tool flow with HLS and the ML models proposed in this article. Experiments have proved that ML models can be trained to enable fast and accurate resource and timing estimations for HLS designs. They have also demonstrated that the proposed approach is able to dramatically reduce the estimation errors for different FPGA devices.

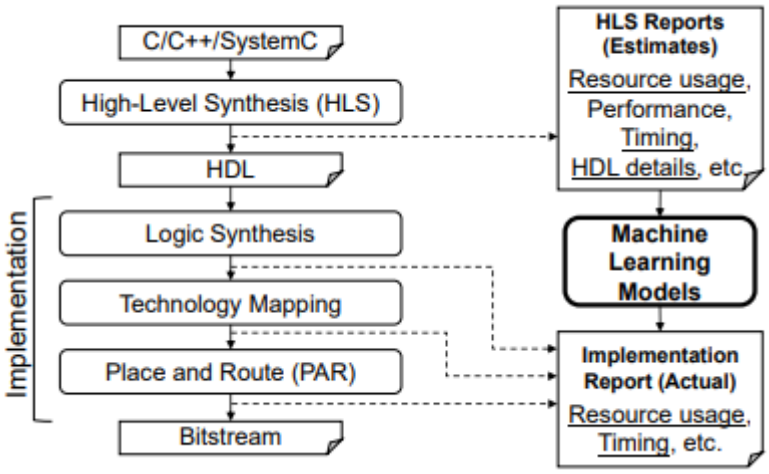

Figure 14. FPGA tool flow with HLS and the proposed ML models

The *hls4ml* software package is introduced and developed by (Duarte, et al., 2018), which is used to build machine learning models in FPGAs. They present a case study for NN inference in FPGAs focusing on a classifier for jet substructure. The results from this implementation have shown that the use of HLS increases accessibility across a broad user community and allows for a drastic decrease in firmware development time. The overall aim of this study is for hls4ml to be a general tool for translating many types of neural network architectures.

CHAPTER 5

CONCLUSIONS

This chapter summarizes the conclusions of the master thesis.

5.1. Conclusions

This thesis starts with detailed description on convolutional neural networks, as well as high-level synthesis process a review. Later, it focuses on the previous work done on the CNNs implementation on FPGAs using HLS

Throughout this work, it is seen that FPGAs have many favorable characteristics, which make them the most promising architectures for accelerating CNNs hardware, like good performance and low power consumption at a reasonable cost. Their highly efficient and flexible architecture enables managing various computing algorithms by the same time they try to accommodate the device's memory resources.

HLS has proven to be a great tool for writing code in high-level languages, avoiding so the hardware-description languages, which are significantly difficult to write for complex schemes. HLS tools translate the code written in a high-level language such as C, C++ or SystemC into HDL code. As a result, a significant distinction between the HLS tools of the current generation and their predecessors is that several tools are developed aiming FPGA implementation. In recent times, research has shown that FPGAs have had a continuous improvement in speed and capacity, making them an excellent platform for signal processing, communication applications and CNN implementation.

REFERENCES

[1] Bjerge, K., Schougaard, J., & Larsen, D. E. (2020). *A generic and efficient convolutional neural network accelerator using HLS for a system on chip design.* Aarhus: Aarhus University.

[2] Chen, Y.-H., Krishna, T., Emer, J. S., & Sze, V. (2016). Eyeriss: An Energy-Efficient Reconfigurable Accelerator for Deep Convolutional Neural Networks. *IEEE Journal of Solid-State Circuits*.

[3] Cong, J., Liu, B., Neuendorffer, S., Nouguera, J., Vissers, K., & Zhang, Z. (2011). High-Level Synthesis for FPGAs: From Prototyping to Deployment. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 473-491.

[4] Dai, S., Zhang, Y., Ustun, E., Young, E. F., & Zhang, Z. (2018). Fast and Accurate Estimation of Quality of Results in High-Level Synthesis with Machine Learning. *26th Annual International Symposium on Field-Programmable Custom Computing Machines.* IEEE.

[5] Dai, S., Zhou, Y., Zhang, H., Ustun, E., Young, E. F., & Zhang, Z. (2018). Fast and Accurate Estimation of Quality of Results in High-Level Synthesis with Machine Learning. *IEEE 26th Annual International Symposium on Field-Programmable Custom Computing Machines* (pp. 129-132). IEEE Computer Society.

[6] Duarte, J., Han, S., Harris, P., Jindariani, S., Kreinar, E., Kreis, B., . . . Wu, Z. (2018). Fast inference of deep neural neetworks in FPGAs for particle physics. *IOP Publishing for Sissa Medialab*.

[7] Gschwend, D. (2020). *ZynqNet: An FPGA-Accelerated Embedded Convolutional Neural Network.* Zurich: Swiss Federal Institute of technology Zurich.

[8] Hassan, R. O., & Mostafa, H. (2020). Implementation of deep neural networks on FPGA-CPU platform using Xlinx SDSOC. *Analog Integrated Circuits and Signal Processing*.

[9] Majumder, K., & Bondhugula, U. (2019). *A Flexible FPGA Accelerator for Convolutional Neural Networks.* Cornell University.

[10] McFarland, M. C., Parker, A. C., & Camposano, R. (1998). Tutorial on High-Level Synthesis. *Proeedings of the 25th ACM/IEEE Design Automation Conference* (pp. 330-360). IEEE.

[11] Ovtcharoc, K., Ruwase, O., Kim, J.-Y., Fowers, J., Strauss, K., & Chung, E. S. (2016). *Accelerating Deep Convolutional Neural Networks Using Specializd Hardware.* Microsoft.

[12] Rao, R. (2020). *Implementation of long Short-Term memory Neural Networks in High-Level Synthesis targetting FPGAs.* University of Washington (Master of Science in Electrical and Computer Engineering).

[13] Sun, Y., Wang, G., Yin, B., Cavallaro, J. R., & Ly, T. (2012). High-Level Design Tools for Complex DSP Applications. In R. Oshana, *DSP for Embedded and Real-Time Systems* (pp. 133-155). Elsevier.

[14] Tagliaferri, L. (2017). *An Introduction to Machine Learning.*

[15] XILINX. (2020). High Level Synthesis. In XILINX, *Vivado Design Suite User Guide* (pp. 5-7).

[16] Yamashita, R., Nishio, M., Do, R. K., & Togashi, K. (2018). Convolutional Neural Networks: an overview and application in radiology. *Springer*.

APPENDIX

The machine learning algorithms are usually developed in Python. This is the main reason that the *hls4ml* tool was developed: to convert the Python codes into HLS projects in C, C++ or SystemC.

Supposing that Python is already installed in the PC: to use the hls4ml tool, it is required to install it by the command `pip install hls4ml` through the command prompt. To be able to use this package, it is needed to import it in the project through the `import hls4ml` command.

Dependencies

Needed tools	**Aim**	**Source**
`numpy, h5py`	For the translation of Keras model files	http://www.numpy.org http://www.h5py.org
`pyyaml`	For configuration file parsing	https://pypi.python.org/pypi/PyYAML
`PyTorch`	For reading in Torch models	https://pytorch.org/
`scikit-learn`	For BDT architectures, includes dependencies on *numpy* etc	https://scikit-learn.org
`onnx`		https://github.com/onnx/onnx
`Xilinx Vivado HLS`	For converting the HLS project into HDL code, to be them implemented on FPGA architectures	https://www.xilinx.com/products/design-tools/vivado/integration/esl-design.html

Further information can be found at http://fastmachinelearning.org/hls4ml/

Vivado HLS

The minimum system memory recommendations for the Vivado Design Suite can be found on the link: https://www.xilinx.com/products/design-tools/vivado/memory.html

32-bit machines are not suitable for Xilinx devices.