POLITECNICO DI TORINO

Facoltà di Ingegneria dell'Informazione
Corso di Laurea in Electronic Engineering

Tesi di Laurea

Design and implementation of a hardware accelerator for Deep Convolutional Neural Networks

Relatori:
Prof. Guido Masera
Prof. Maurizio Martina

Candidato:
Osvaldo Genovese

Settembre 2018

Ringraziamenti

Un primo ringraziamento va ai miei relatori, il professore Guido Masera e il professore Maurizio Martina, per avermi guidato nello sviluppo di questa tesi e soprattutto per avermi fornito spunti interessanti nei momenti di difficoltà.

Grazie a tutti i ragazzi del VLSI lab che grazie alla loro simpatia hanno reso molto più leggere le giornate passate al pc e mi hanno sempre regalato consigli preziosi. In particolare grazie a Luigi, Ibrahim, Andrea e Giuseppe per tutte le giornate passate fuori ma soprattutto dentro il laboratorio.

Un grazie speciale a te, Silvi, che da quando ci siamo conosciuti non fai altro che spronarmi a dare il meglio di me in qualsiasi situazione. Sei sempre stata al mio fianco, nei momenti felici ma soprattutto in quelli più difficili.

Un grazie enorme alla mia famiglia, mio papà, mia mamma e mio fratello, che oltre a darmi la possibilità di raggiungere i miei traguardi non hanno mai smesso di credere in me, e sembra per fortuna, non smetteranno mai.

Sommario

In recent years,artificial intelligence and deep learning have become some of the most analized and discussed topics both in academic and R&D world. This happened mostly because of the increased interest regarding the analysis of big data; the ultimate goal was to implement always smarter automatic platforms in order to meet the rising demand.
In particular, most of the improvements in artificial vision tasks are reliyng on the development of neural networks, which can be defined as a series of algorithms that attempts to identify underlying relationships in a set of data. The *Neural* term was chosen since this network is able to correlate informations using a process that mimics the way the human brain operates. Neural networks have the ability to adapt to changing input so that the network produces the best possible result without the need to redesign the output criteria. In this work will be analyzed one of several possible neural network, called Deep Convolutional Neural Network. This structure is able to perform in a very efficient way matrix-matrix multiplication(convolutional product).The purpose of this network is mainly to provide a classification for a set of images in order to identify some previously defined classes. But in order to perform this classification in real time, the neural network has to compute a large amount of convolutional operations that can saturate the resources on our platform in terms of time and power. The ultimate goal of the recent applications in this field is to keep the accuracy of the classification really high(up to 99%) trying to reduce the power consumption and increasing the speed of execution.
In this thesis work it will be designed and implemented a Hardware accelerator able to perform the matrix-matrix multiplication, trying to exploit an energy-efficient convolution architecture. In order to achieve this result, as it will be explained later, it will be necessary to completely reschedule the dataflow of the convolutional steps. After a first implementation, a further optimization will be developed, trying to reduce as much as possible the internal parallelism of the structure in order to reduce the clock period and at the same time to increase efficiency.

Indice

Ringraziamenti I

Sommario II

1 Introduction **1**
1.1 The Deep Learning Context 2
1.2 Hardware acceleration and energy optimization 3
1.3 Purpose of the thesis and thesis outlines 4

2 Background on Convolutional Neural Network **5**
2.1 Previous work on CNNs 6
2.1.1 ILSVRC Dataset 7
2.2 Architecture of a CNN 7
2.2.1 Convolution Operation 8
2.2.2 Activation functions 11
2.2.3 Pooling Layer 12
2.2.4 Fully Connected Layers 13
2.3 Hardware acceleration of CNNs 14
2.3.1 FPGA vs ASIC structures 15
2.4 Methodology and research question 16

3 The proposed architecture **18**
3.1 Data Flow Diagram Description 19
3.2 Matlab Processing 20
3.3 Overall Architecture of the proposed Accelerator 22
3.3.1 Input Data Memory 25
3.3.2 Weight Memory 27
3.3.3 Datapath 28
3.3.4 Control Unit 31
3.3.5 Off-Chip Memory 32
3.4 Simulations 34

3.4.1 Modelsim Simulations 34

4 Hardware implementation and applications 40
4.1 Synthesis 41
4.1.1 Introduction to Synopsys Design Compiler 41
4.1.2 Reading VHDL source files 42
4.1.3 Applying constraints 43
4.1.4 Optimizing and Analyzing the reports 44
4.1.5 Saving the synthetized design 46
4.1.6 Simulation of the Netlist 46
4.1.7 Switching-activity-based power consumption estimation . . . 47
4.2 Architecture with reduced parallelism 49
4.3 Application of different weight filters 50
4.4 Comparison with previous Hardware Accelerators 53

5 Results and Conclusions 54
5.1 Proposed work summary 55
5.2 Future Works 56

Bibliografia 57

Capitolo 1

Introduction

This introductive chapter presents an overview of topics that will be analyzed in the following pages. Section 1.1 provides a brief summary on the so called "Deep Learning Context", a widely discussed topic in both research and engineering. Then CNNs algorithms are introduced in order to understand their working principles and the possible applications in engineering fields. Section 1.2 describes the meaning of "Hardware Acceleration" and how it can impact on the performance of the algorithm. A special focus is dedicated to the power management, crucial in real time applications. Finally Section 1.3 describes the purpose of this thesis and how it is organized.

1.1 The Deep Learning Context

In the last 20 years we have had a great explosion of the so called "big data" in multiple fields. The only possible way to analyze and point out answers from the much longer list of questions, was found in "Artificial Intelligence". An AI can be defined as a mix of software and hardware applications that are able to understand problems and solve them in a way that is similar to the one a human could choose. The full range of possible implementation is still illimitate; we could go from the interpretation of human language (speech recognition), to the analysis of a real-time video (image processing), to finally arrive to discern people's faces and behaviours. The AI field is still in constant evolution and inside this giant branch of study, we find Machine Learning (ML).
Machine Learning can be defined as a tecnique that gives to a defined AI the ability to progressively improve performances on a specific task without being explicitly programmed. In the usual way of programming, the software developer can create complex algorithm to perform a particular task, but all the steps have to be well known. The idea on which ML relies, consists of using a large Dataset and through simple functions be able to make decisions based on the data available. The result of a particular decision can be obtained following different mathematical formulas, like polynomial functions,statistical functions and so on. Each function can be linked to a particular behaviour,and the tecnique that will be analyzed in this thesis relies on a smaller branch of ML called Deep Learning(DL).

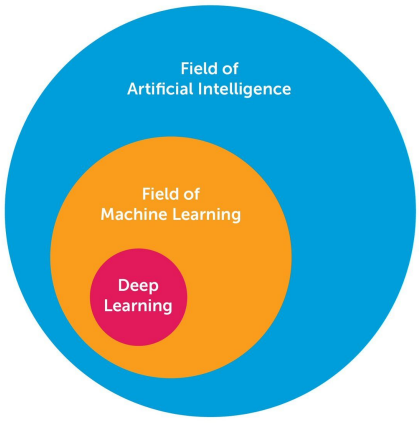

Figura 1.1. Deep learning inside Artificial Intelligence

In the last years,the Deep Learning field has become one of the most promising approach to solve a wide range of computing challenges. Every hour billions of

images or audio file are saved and stored inside big data centers of different companies. The amout of data to analyze is so impressive that the common goal is now to develop tecniques able to extract useful information from these datasets in an efficient way. One of the ML algorithm used to achieve this automatic extraction is the Neural Network approach.
In particular Convolutional Neural Network are a specific type of Artificial Intelligence that are modelled on the biological process of our visual cortex,and nowadays they outperformed all the previous networks in the image recognition and classification field.
CNNs are able to extract a particular pattern of information from an image and provide a classification even from an high level of abstraction. In the last few years the effectiveness of these algorithms increased at incredible rates. As a result, an always stronger effort has been carried out from researchers in order to keep this trend possible. Following this direction many Machine Learning frameworks have been developed in order to design and train Deep CNNs. Image processing is one of the interested fields on which CNNs had a massive impact. For this reason a lot of researchers are working to develop hardware solutions able to exploit the new achievable performances.
Following this direction, in the thesis the ultimate goal will be to optimize from an architecture point of view the internal computations of a generic convolutional accelerator.

1.2 Hardware acceleration and energy optimization

As we discussed before, the adoption of CNNs had an huge impact on the overall accuracy of image processing tasks, but everything has a prize and in this case we pay in terms of energy requirements in the overall computations. For this specific reason, nowadays it is true that we are trying to develop tecniques able to speed up the computation, but at the same time meeting the constraints on power consumption is also a primary goal. From a computational point of view CNNs represent a particul type of algorithm; they require a number of memory accesses and product operations that increases exponentially with the size of the whole network. This behaviour could lead to massive costs in terms of latency and working frequency, but on the other hand the major part of the patterns of CNNs are highly repetitive. This particular property can stimulate a pipelined and parallelized approach in order to boost the overall throughput.
Since the main operation of this kind of network is a convolutional product between

the pixels of the images and weights of the filter(almost 95% of computational power), the most solid way to increase the overall performances is always to optimize this kind of operations.
As the computational cost of CNNs rapidly increases, GPUs and application-specific integrated circuits specialized for parallel computation have been widely employed for real-time tasks; in this thesis we will discuss of both approaches even if the final implementation will be modelled using a specific application architecture, for reasons that will be cleared in the next chapters.

1.3 Purpose of the thesis and thesis outlines

The aim of this thesis is to implement and analyze a hardware accelerator able to compute the convolutional operations in a CNN layer. The ultimate goal is to achieve a significant energy-efficient architecture, without penalties in terms of speed and accuracy.
Since the main operations of a CNN are based on convolutional products, the first part of the work was to study the analytical expression of this operation and the different possible implementations of a consequent hardware architecture.
In order to better visualize the pattern of the thesis, below are reported the topics that will be analyzed in the future chapters.
In Chapter 2 will be provided a brief introduction on the evolution of Convolutional Neural Networks. After the analysis of previous works, it will be described the internal structure of a CNN with a particular focus on the possibility of hardware acceleration.
In Chapter 3 we will analyze the proposed architecture starting with the description of the Data Flow Diagram. Then it will be described the architecture developed with all the internal blocks. In the final part the hardware accelerator will be simulated with an highlight on the most important phases.
In Chapter 4 will be analyzed the synthesis phase, with the description of all the necessary steps. Then the main features (area,speed,power) will be extracted and the results will be compared with similar architectures. Finally a brief analysis of different kernel filters will be provided in order to understand how process different images.
In Chapter 5 will be outlined considerations on the work done and suggestions on the possible steps for future developments.

Capitolo 2

Background on Convolutional Neural Network

This chapter will present an overview of CNNs that is fundamental to understand the analysis in the following chapters. Section 1.1 describes the previous work and achievement obtained in the last years of reasearch in the development of Deep CNNs. Section 1.2 provides an introduction on the basic architecture of a CNN including a description of the functions of different layers. Then CNNs algorithms are introduced in order to understand their working principle and the possible applications in engineering fields. Section 1.3 analyze the meaning of hardware acceleration and the possibile choices in order to increase the overall performances.

2.1 Previous work on CNNs

Nowadays CNNs have become the most used platform for image processing and recognition, but this trend is due mainly to some incredible achievements obtained in the last few years. In order to analyze the evolution of convolutional network the starting point for all researchers should be the "Alexnet" network, developed and published in 2012 by Krizhevsky from the University of Toronto[1].
The main reason of the global success of this network is the brilliant victory at ImageNet Large Scale Visual Recognition Challenge (ILSVRC), a global competition where software programs compete to correctly classify and detect objects and scenes. In 2012, AlexNet significantly outperformed all the prior competitors and won the challenge by reducing the top-5 error from 26% to 15.3%. The second place top-5 error rate, which was not a CNN variation, was around 26.2%. The top-5 error can be defined as the percentage of times where the correct prediction is not in the first 5 choices of the classificator. The main variations from previous network were the multiple filter per layer and stacked convolutional layers.

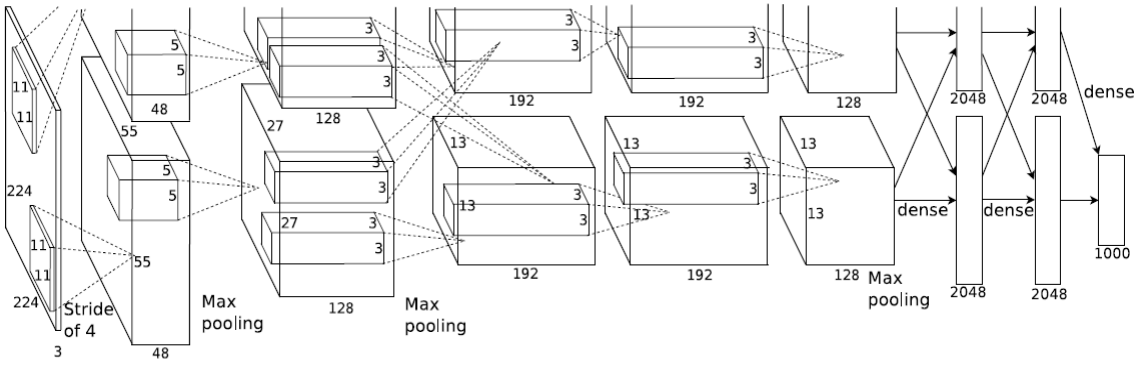

Figura 2.1. Alexnet Architecture [1].

After the success of Alexnet many networks were developed and one year later the so called "ZFNet" [2] was able to outperform Alexnet with a top-5 error rate of 14.8%.
At last, at the ILSVRC 2015, the so called Residual Neural Network (ResNet)[3] by Kaiming He introduced an architecture with "skip connections" and features heavy batch normalization. Such skip connections are also known as gated units or gated recurrent units and have a strong similarity to recent successful elements applied in Recurrent Neural Networks. Thanks to this technique they were able to train a with 152 layers achieving a top-5 error rate of 3.57% which beats human-level

performance on this dataset.
In 2017 an hardware accelerator was proposed by Yu-Hsin Chen [4]; *Eyeriss* optimizes for the energy efficieny of the entire architecture, including off-chip memory. This approach is similar to the one we will use in our network, since they propose an architecture with multiple processing elements exploiting a particular dataflow called Row Stationary(RS). This structure significantly reduces the data movement achieving great results in terms of speed and power.

2.1.1 ILSVRC Dataset

In order to test the proposed network it is necessary to define a generic Dataset from where all the images will be taken. In this thesis we will always work with the Dataset of the ImageNet Large Scale Visual Recognition Challenge (ILSVRC), since all the previous neural network were tested with it. In particular since it wasn't possible to test all the images with the different weight filter we will see just a small part of the Dataset of the 2017 but it will be enough to highlight all the main feautures in the proposed convolutional operations.

Figura 2.2. Alexnet Dataset

2.2 Architecture of a CNN

In machine learning, a Convolutional Neural Network (CNN, or ConvNet) is a class of deep, feed-forward artificial neural networks, most commonly applied to analyze visual imagery. Convolutional networks were inspired by biological processes where the connectivity pattern between neurons resembles the organization of the animal visual cortex.

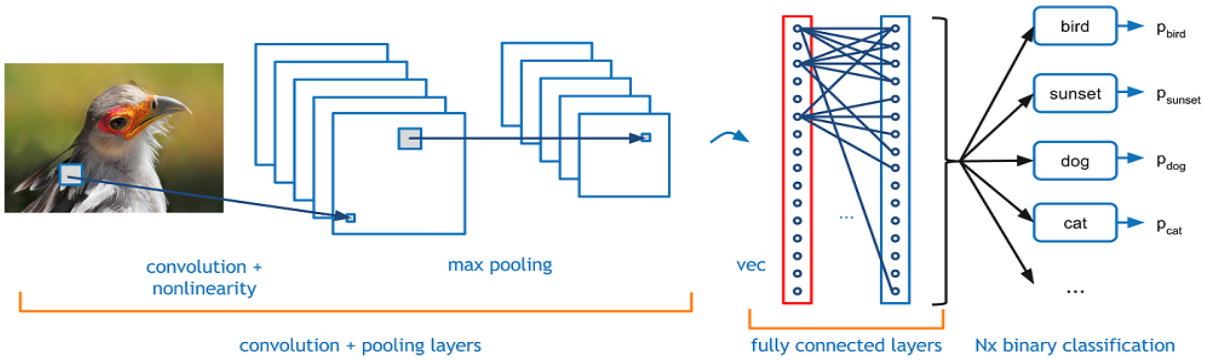

Figura 2.3. Convolutional Neural Network Architecture

Individual cortical neurons respond to stimuli only in a restricted region of the visual field known as the receptive fields. The receptive fields of different neurons partially overlap such that they cover the entire visual field.
The CNN algorithm is build by stacking multiple computation layers for feature extraction and classification. The latest CNNs achieve their high accuracy thanks to a very deep hierarchy of layers, which are able to convert the input image data into highly abstract representations called *feature maps*. Every CNN is able to perform four main operations, that corrispond to the basic building blocks,as shown in the list below.

- Convolutional Product
- Non Linearity (ReLU)
- Pooling
- Fully Connected Layer

In the following sections we will describe these operations,but the main focus will be the convolutional product operation since it is the one that will be optimized in the architecture developed.

2.2.1 Convolution Operation

The primary computation inside a CNN layer is a high-dimensional convolution. Before analysing in detail the operation, it is necessary to give some definitions about image processing. A generic image can be defined as a matrix of pixel values,

where each pixel has three different channels called Red,Blue,Green (each of 8 bit); in case of grayscale images, the channel is just one. A convolution operation is a matrix-matrix multiplication between an image, that is called input feature map (iFMAP),and a weight filter. A filter is a particular type of matrix that is able to extract embedded characteristics of an image; in the following chapters we will see how changing the coefficients of the filtering window it will be possible to highlight different aspects of the same image.

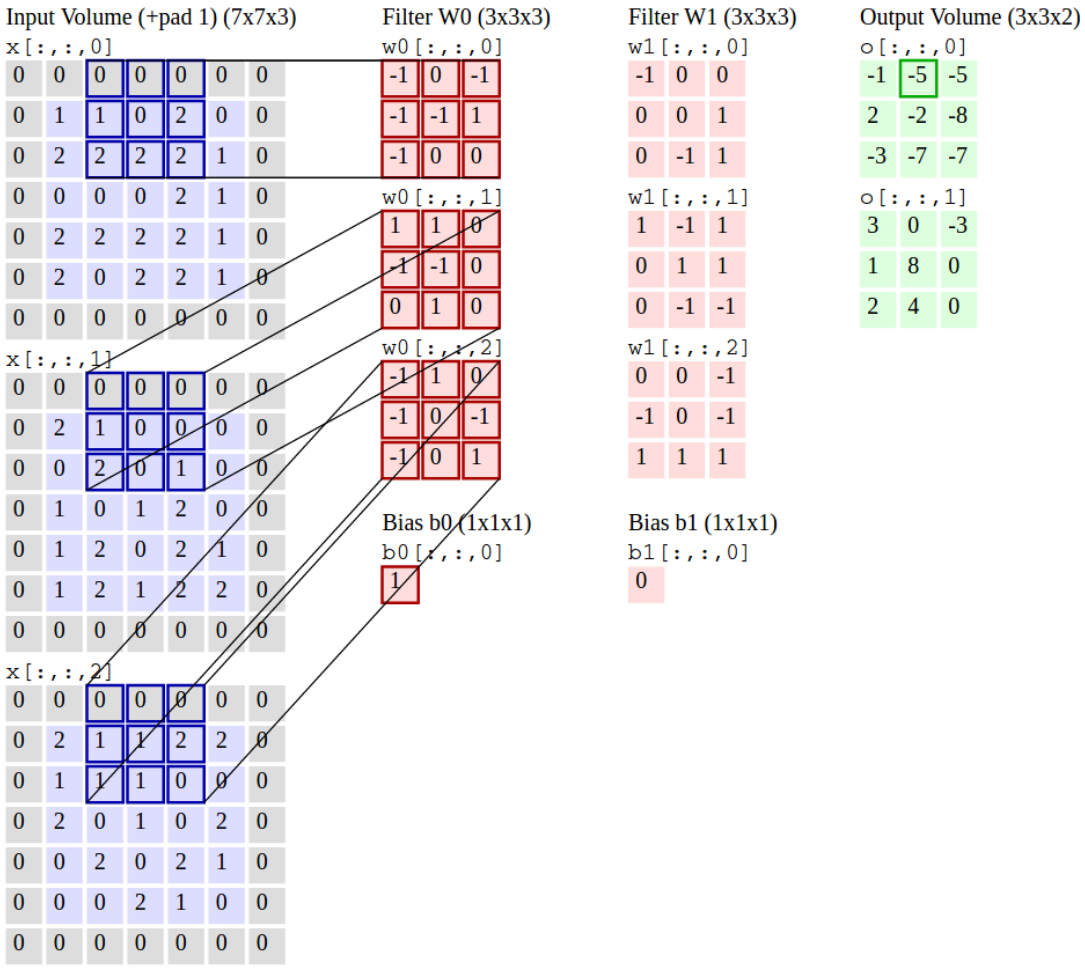

Figura 2.4. Convolution Example

The overall convolutional operation can be easily understood looking at the example above. Here we have an input feature map of 7x7x3 matrix while the

kernels are 3x3 matrix. The Kernel have to slide on the input matrix in order to correctly multiply all the pixels one by one. An important parameter in this operation is the sliding size called stride; it is defined as the space between each linear sample. Essentially, the stride is a metric for regulating the movement of various convolutional filters for pixel-wise operations across a given image space. In the following example the stride equal to 2.
As it is clear from the previous equations, the convolution operation can be computed as a series of pixel multiplications and iterative sums of partial product. In the image processing field the dimension of the convolution can be three-dimensional if we are looking to a RBG picture as in the previous example, but also two-dimensional, if we are analyzing picture in grayscale.
For sake of simplicity in the developed architecture the convolution operation is applied to grayscale image, but,as it will be explained later, it is really simple to convert a RBG image in grayscale through some rows of code in MATLAB.
The following formula defines the standard convolution operation applied in the two-dimensional case.

$$c_{xy} = \sum_{z=0}^{K-1} c_{xyz} = \sum_{z=0}^{K-1} \sum_{i=0}^{K-1} w_{zi} \times f_{(x+z)(y+i)} \tag{2.1}$$

In the previous equation f,w,c can be defined as the input feature map, the filter weight and the output feature map, respectively. If we try to apply this equation to a simple example we will have the following expanded coefficients:

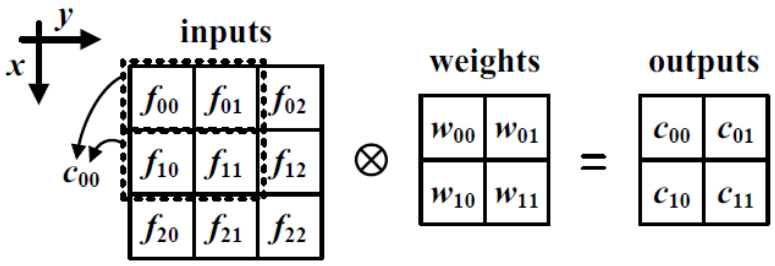

Figura 2.5. Input feature map & filter weight example.

$$c_{00} = w_{00}f_{00} + w_{01}f_{01} + w_{10}f_{10} + w_{11}f_{11} \tag{2.2}$$
$$c_{01} = w_{00}f_{01} + w_{01}f_{02} + w_{10}f_{11} + w_{11}f_{12} \tag{2.3}$$

$$c_{10} = w_{00}f_{10} + w_{01}f_{11} + w_{10}f_{20} + w_{11}f_{21} \tag{2.4}$$

$$c_{11} = w_{00}f_{11} + w_{01}f_{12} + w_{10}f_{21} + w_{11}f_{22} \tag{2.5}$$

The previous computation is applied with a weight filter 2x2 and a input feature map 3x3. From an analytical point of view it is possbile to compute the size of the output feature map using a simple equation:

$$oFMAP = (iFMAP - Filter + Stride)/Stride \tag{2.6}$$

It is plan to see that many input features along with the filter weights are loaded multiple times, this will result in large input memory accesses. Since thanks to previous works has been discovered that almost 90% of the power is related to these memory accesses, in the following chapter it will be proposed an architecture that is able to analyze every pixel of the input feature map just once and avoid all the redundancy loads.
As can be imagined in this case we will have the minumum number of memory accesses for each pixel, but of course we will pay in terms of parallelization; in fact it will be necessary to increase the number of resources that work at the same time. In our particular case we will see how we will need a number of parallel processing elements that grows linearly with the size of the kernel filter.

2.2.2 Activation functions

For sake of completeness, it is necessary to define the non linear unit, also called Rectified Linear Unit. In fact in almost all the CNNs after a convolutional layer it is placed a *ReLU*. The main function of this unit is introduce a non linearity in the system since the major part of the world data are non linear while all the convolutional operations are linear functions. Before *ReLUs* other functions were used, like the *tanh* or *sigmoid*; the main reason to use ReLU instead of the others is that it is possible to reduce the time spent training the network, without an evident loss of accuracy.

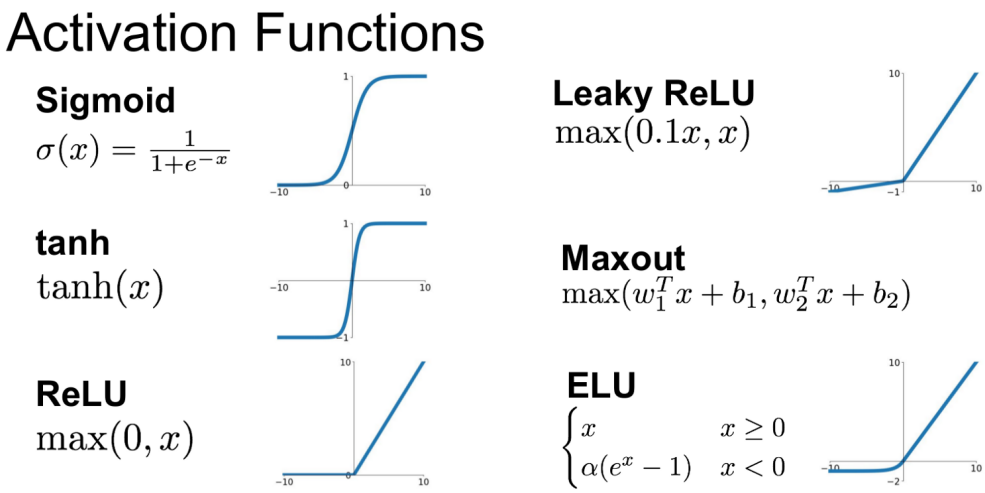

Figura 2.6. Activation Functions

2.2.3 Pooling Layer

The *Pooling* layer, also called sub-sampling layer, is usually an intermediate layer between the series of two convolutional ones. The physical implementation of this layer is quite similar to the convolutional operation, but in this case the filter is smaller (usually 2x2).
The most used type of pooling layer is the *Max-Pooling* one, which selects as output pixel the higher value in the analyzed window. After this operation the dimension of the output feature map is always reduced according to the size of the kernel.
The main results of this layer are basically two; it is able to reduce the dimension of the output feature map, with a significant saving in terms of storage of intermediate results. Besides the choice of the maximum value in the window allows to pass to the next layer only the most relevant features. In the following image is shown a simple example of max pooling operation:

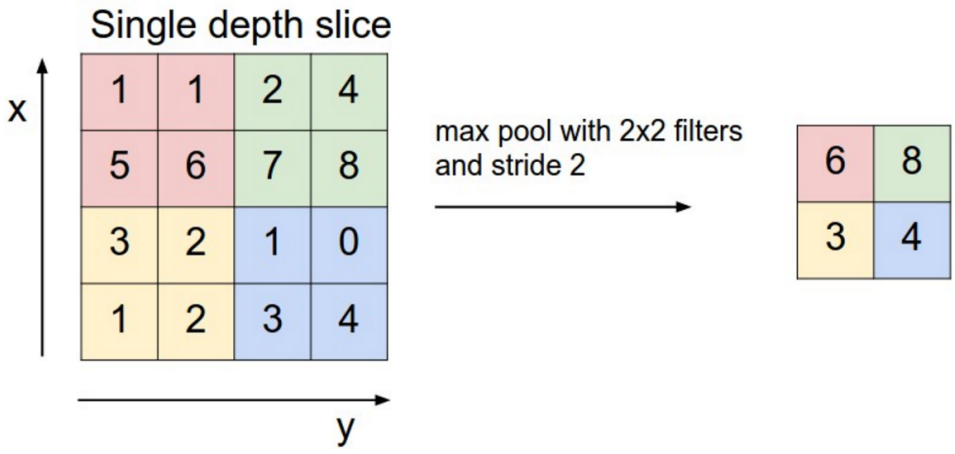

Figura 2.7. Max Pooling example

2.2.4 Fully Connected Layers

The *Fully Connected Layer* is basically composed of several units that are able to take the output of the previous layer as input (a max-pooling, a ReLu or a Conv) and determines which features most correlate to a particular class. Of course this layer is placed at the end of the CNN since it needs high level features to give the correct results. In the figure below we can see the differences between the connections of a FCL and a CL.

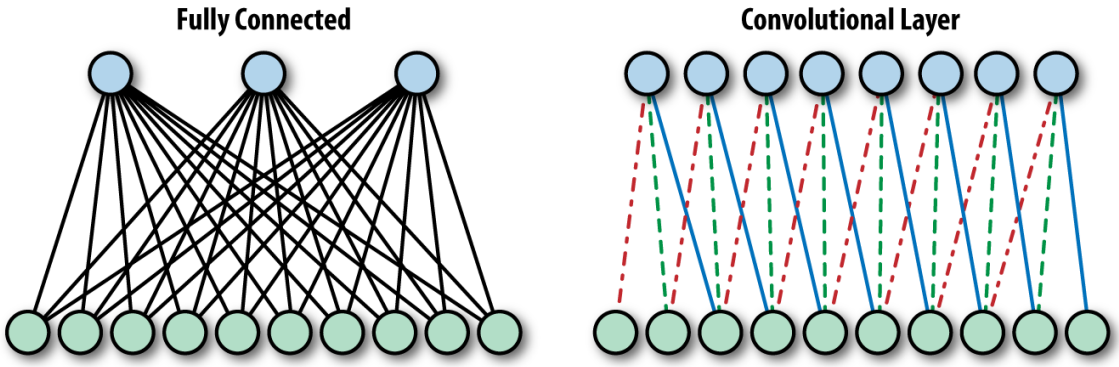

Figura 2.8. Fully-Connected VS Convolutional Layer

2.3 Hardware acceleration of CNNs

Convolutional neural networks have become the first solution for image recognition applications mainly because of their really impressive results in terms of accuracy; in the last few years the needy to make CNN available on mobile application and low power systems increased exponentially.
From this necessity grew up the concept of *Hardware Acceleration*, that could have many ways of interpretation. While in the usual way of thinking in the electronic research, the acceleration of a process means to be able to decrease the clock period, in the case of CNN tha main goal is to reduce the energy consumption meeting real-time constraints on speed and accuracy. One of the advantages of the convolutional operation is that it is highly repetitive and in some cases can be easily pipelined and parallelized.
In order to expoit the parallelism of the structure in the early days the first idea was to switch from CPU to GPU the processing of the networks. Even if the performances were good, the CNNs world was expanding in a very fast way, so it was necessary to develop new ideas to follow up on the increasing computational efforts of the networks. In the following figure we can see a general structure of an Hardware Accelerator; as we will explain in detail later, the main blocks are always the same. The convolutional operation is carried on by the Processing Elements (PE), while we have internal memories to keep track of the partial results and an interface that has to connect the on-chip memories with the off-chip ones.

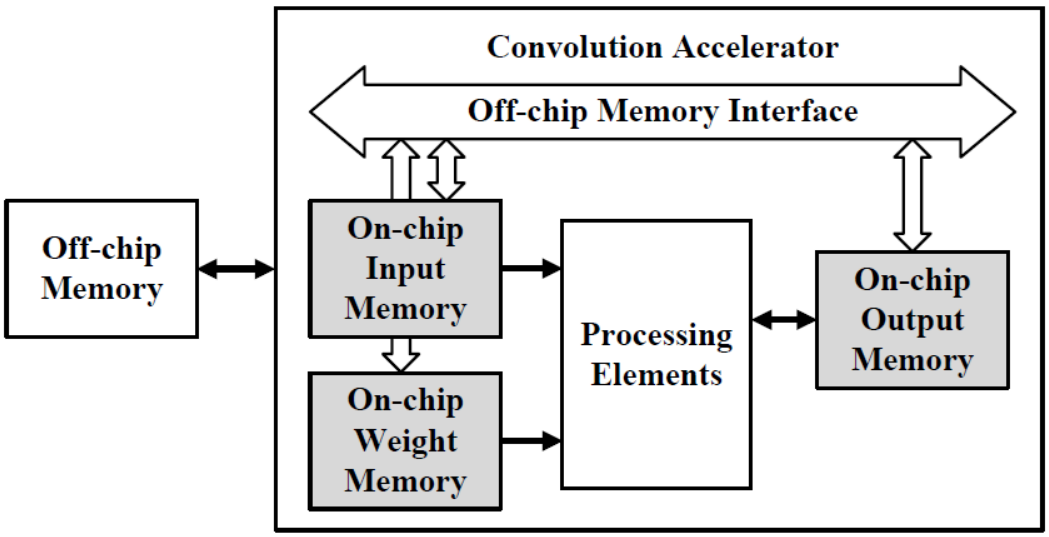

Figura 2.9. General Hardware Accelerator architecture.

Of course this structure has to be designed from scratch and it can have different implementation and cost problems. For this reason in the next paragraph are described two different implementation choices to further exploit the particular properties of these networks.

2.3.1 FPGA vs ASIC structures

Before analyzing the two main possibilities to implement a neural network, we have to define a FPGA and an ASIC structure. A Field Programmable Gate Array (FPGA) is an integrated circuit designed to be configured by a customer or a designer after manufacturing hence "field-programmable". The FPGA configuration is generally specified using a hardware description language (HDL); FPGAs contain an array of programmable logic blocks, and a hierarchy of reconfigurable interconnects that allow the blocks to be "wired together", like many logic gates that can be inter-wired in different configurations. Logic blocks can be configured to perform complex combinational functions, or merely simple logic gates. In most FPGAs, logic blocks also include memory elements, which may be simple flip-flops or more complete blocks of memory.
An Application-Specific Integrated Circuit (ASIC), is an integrated circuit (IC) customized for a particular use, rather than intended for general-purpose use. In the previous years of CNN development the ASIC solution was preferred since it was possibe to fully parallelize the whole architecture and be able to speed up a lot the different convolutional operations. But in the last period a lot of researchers decided to switch the implementation using FPGAs; the main reason of this big change is that ASIC processors have a huge non recurrent cost(NRE) and since the neural network world is in continuing expansion they needed a product that could have an higher flexibility. This characteristic was found in FPGA even if in some cases the performances were a little bit lower.

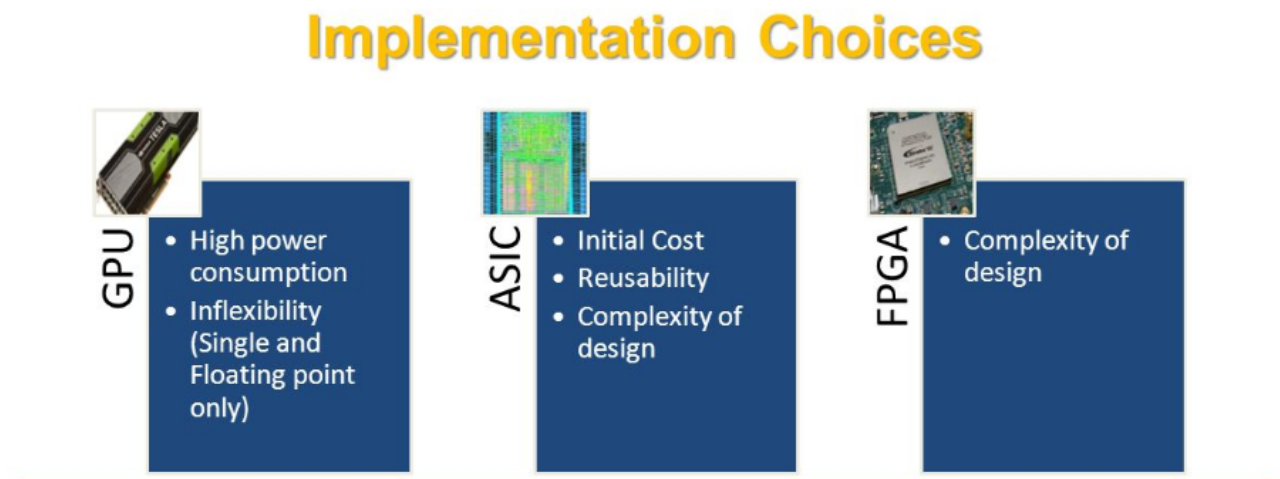

Figura 2.10. GPU vs ASIC vs FPGA

In this thesis we will try to exploit a solution implementing a ASIC architecture; this choice is due to the fact that in this early design phase we want to optimize a particular convolutional operation without many degrees of flexibility. For this reason in the next chapter it will be possible to see all the steps necessary to build from scratch an hardware accelerator. The starting point will be the DFG analysis, followed by the implementation of the code in an Hardware Description Language, in our case will be VHDL and Verilog; finally the whole structure will be simulated and synthetized in order to evaluate all the main features like the minimum clock cycle, the area of the chip and of course the power consumption.

2.4 Methodology and research question

As mentioned in Chapter 1, CNNs have become a very widely discussed topic in both companies and academic research. Lot of efforts have been done in order to provide improvement in accuracy, performance and energy efficiency of such algorthms. Despite the latest achievements in this field, the large amount of operations and memory accesses required to implement a CNN makes it a very difficult task. This is particularly true when this type of networks have to be implemented in hardware and critical scenarios where there is a particular need that regards power consumption, such as embedded devices and mobile applications.
Even though GPUs demonstrated to be efficient as accelerators for both training and inference of CNNs, such devices have an always higher power consumption, which makes them not suitable in the aforementioned power-constrained real-time

applications. For this reason, interest in special purpose accelerators has been growing in the last years. Indeed, this kind of devices can achieve massime saving in tems of power, and their architecture perfectly fit the massively parallel computation pattern of CNNs and neural networks in general. However, the process to design and implement such algorithms on ASIC structure can still be complex, since there is no pre-defined hardware structure, but the all network has to be designed and implemented from scratch.
Given these motivations, the aim of this work is to develop a hardware accelerator for Deep Neural Network, providing a continous description of all the necessary steps.
The methodology used is basically the design flow for every special purpose application; in the first phase the algorithm has to be analyzed and all the properties have to be derived from the DFGs. Then we need to choose how to define all the blocks to implement the analytical operation, and also how to reduce the internal connection and parallelism. Finally the architecture has to be simulated and synthetized in order to see if the ideal properties have been fully exploited. The final part is the analysis of the obtained results, comparing them with the latest results from similar architecture.

Capitolo 3

The proposed architecture

This chapter will present the developed architecture, with a focus on the code implemented in VHDL. Section 2.1 provides a brief analysis on the Data Flow Graph related to the convolutional operation and the possibility to reschedule it in order to optimize the memory accesses. Section 2.2 describes the Matlab code developed to fetch the architecture with the images and to show the final results. Section 2.3 provides an analysis in detail of the overall architecture with a particular focus on the main blocks and the different functions implemented. Finally Section 2.4 provides all the simulations necessary to understand the correct behaviour of the accelerator.

3.1 Data Flow Diagram Description

Before any type of hardware implementation, it is necessary to analyze the data dependencies inside the convolution algorithm thorugh a Data Flow Diagram. A DFD is a graphical representation of the "flow" of data through an information system, modelling its process aspects; it is often used as a preliminary step to create an overview of the system without going into great detail, which can later be elaborated. Applying the DFD to the convolutional operation in two-dimensione we will have the following figure.

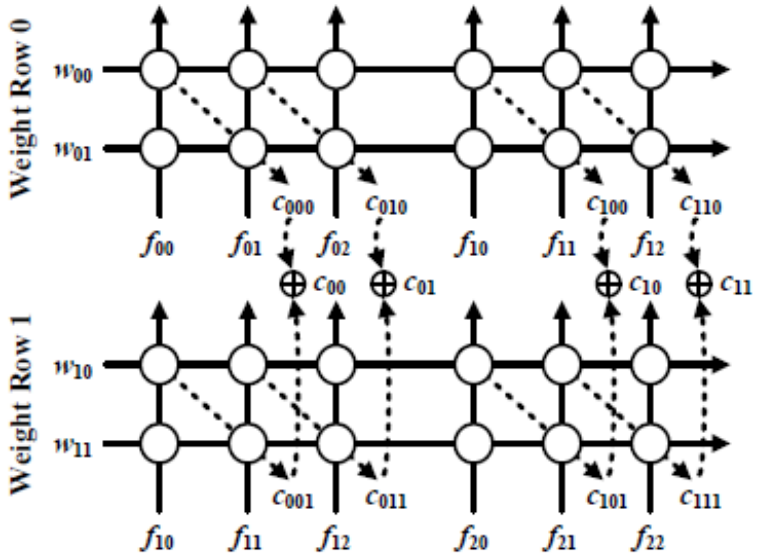

Figura 3.1. 2D Convolution [5].

For sake of simplicity all the pictures are referred to a 3x3 iFMAP and to a 2x2 weight filter, while every circle in th figures indicates a Multiply-and-Accumulate unit(MAC). In the previous picture two rows of filter weights are multiplied separately with the input features of the upper and lower dependence graphs, so the intermediate partial sums are obtained in separate ways. In this case it is necessary to aggregate the partial sums in successive operations. As mentioned in the previous chapter, a portion of the pixels in the input features are loaded multiple times inside the structure. Since almost 90% of the energy is consumed for memory accesses, while only a small part is devoted to the computational units, the overall efficiency can be enhanced by minimizing the redundant on-chip memory accesses.
In order to achieve this goal, the only way is to modify the Data Flow Diagram.

In the past few years many researchers developed different DFGs, one of the most interesting was proposed by Jihyuck Jo [5],where the undesired energy consumption due to redundancy is minimized allowing to load every pixel of the IFMAP just once. In the following figure it can be seen a rescheduled DFG that supports the previous described behaviour.

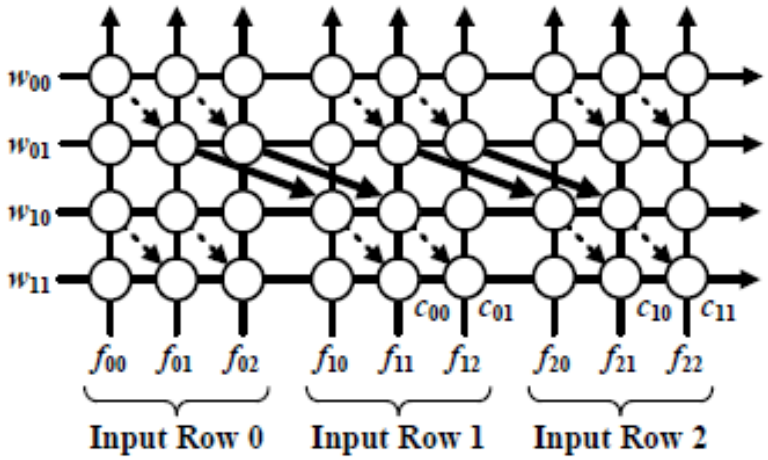

Figura 3.2. Rescheduled Data Flow Diagram [5].

The partial sums are not added immediately, but thanks to some on-chip memories are accumulated and wait for the execution of the next row. We will see in the next chapters that through this delay in the final computation of the partial sums the overall energy efficiency will be enhanced.

3.2 Matlab Processing

The previous architecture has been build for an input feature map of 128x128 pixels in grayscale (for simplicity just one channel) and supports kernel filter with a size 3x3. These choices were made for different reasons that we need to analyze in detail. From the DFG we know that even if the architecture is able to compute multiple operation in parallel, it is necessary to insert one pixel for each clock cycle and save inside the memories all the convolutional values computed for each row. Now it is clear why the size of the iFMAP impact in a linear way on the dimension of the on-chip memory. This issue could have a huge inpact if we consider that high resolution images would need always bigger memories. This problem can be bypassed

supposing that scaling in future tecnologies will be able to follow the increasing size of future images.
For this reason in the design phase it was decided to block the size of the image at 128x128 pixels; a much bigger issue have to be considered if we decide to increase the size of the weight filter. Analyzing the DFG in the previous section it can be noted that for each weight in the filter window it is necessary to implement a different processing element. If we use as example the first convolutional layer of Alexnet with a filter window of 11x11 pixel we will have 121 PEs that work in parallel.
Since our ultimate goal is to increase the efficiency of the structure it was decided during design phase to block at 3x3 the size of the weight filter. This choice has two main consequences,in fact from one side we will need much less processing elements e so we will reduce simultaneously power consumption and area, but at the same time we will not have a heavy reduction of the dimensions of the output feature map. The solution to this problem is to add an extra Max Pooling layer after the computation to further reduce the size of the oFMAP.
For these reasons in order to analyze different images without limit on their dimensions or colour, a simple script in Matlab has been developed. It is able to downsample every image to a 128x128 input feature map in grayscale.

```
%defining filter
filter = [-1 -1 -1 ; -1 8 -1 ; -1 -1 -1];
%  importing images
I = imread('gorilla.jpeg') ;
%  resizing images
Image_resize = imresize(I,[128 128]);
% showing the image
imshow(Image_resize);
%  Converting to gray scale
Image_grayscale = rgb2gray(Image_resize) ;
%inverting the matrix
Image_grayscale = Image_grayscale';
%showing the image
imshow(Image_grayscale);
% converting to hex
gray_hex = [dec2hex(Image_grayscale)];
% write the matrix into a txt file
dlmwrite('gorilla.txt',gray_hex,'delimiter','') ;
```

Figura 3.3. Pre-processing script.

Also at the end of the computation another Matlab script has been developed

to read the results written in a txt file and show the obtained output feature map.

```
% loading the oFMAP
A = load('results.txt');
B = int16(zeros(126,126));
k=1;

% loop to fill the matrix
for i=1:126
for j=1:126
     B(i,j)= A(k);
     k=k+1;
end
end

% adjusting the values
B = 255 -uint8(B);
% showing the image
imshow(B);
```

Figura 3.4. Post-processing script.

3.3 Overall Architecture of the proposed Accelerator

In previous researches have been analyzed different configurations for an hardware accelerator in a convolutional layer. It is possible to define three different architectural choices in transferring input and weight data from memories to the PE: *Broadcast*, *Forwarding*, and *Stay*.
The *Broadcast* approach requires a multi-port input memory able to read a different data every clock cycle and to send it to all the PEs at the same time, in order to exploits a full parallelism in the intermediate operations. The *Forwarding* approach uses the loaded data into a PE and passes them to a neighboring PE through some registers. The *Stay* scheme instead loads data inside a PE and keeps them for an entire convolutional operation.
In a similiar way we can define the output process in three main cases: *Aggregation*, *Migration*, and *Sedimentation*. The *Aggregation* scheme uses an hardwired adder tree in order to sum all the partial products coming from the PEs, this could be

problematic if the size of the convolutional operation increases too much. The *Migration* scheme pass the partial sum to a neighboring PE while the *Sedimentation* has an output memory for each PE.

Input	Weight	Output	Abbreviation
Broadcast	Broadcast	Aggregation	BBA
Broadcast	Forwarding	Sedimentation	BFS
Broadcast	Stay	Migration	BSM
Forwarding	Broadcast	Aggregation	FBA
Forwarding	Forwarding	Sedimentation	FFS
Forwarding	Stay	Aggregation	FSA
Stay	Broadcast	Aggregation	SBA
Stay	Forwarding	Migration	SFM
Stay	Stay	Migration	SSM

In Fig. 3.5 and 3.6 are defined how the different approaches can be implementend using Processing Elements(PEs) and On-chip Memories (OCMs).

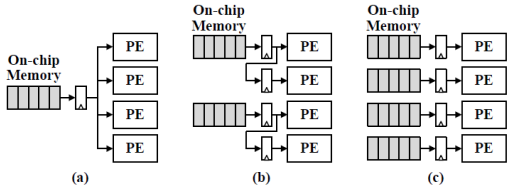

Figura 3.5. Input and weight Schemes. (a) Broadcast,(b) forwarding,(c) stay.

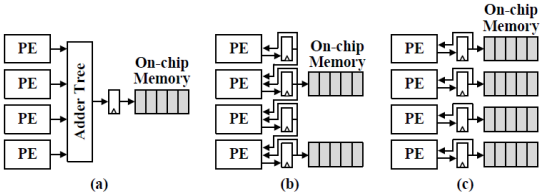

Figura 3.6. Output Schemes. (a) Aggregation, (b) migration, (c) sedimentation.

The selection of the load and store schemes is the key to realize an accelerator able to enhance particular features in the convolutional operation. In the proposed architecture the final choice was the so-called BSM structure (Broadcast-Stay-Migration) where we decided to apply the Broadcast approach for the loading of the iFMAP, the Stay approach to load the weight filter, and the Migration approach to collect the outputs. For what concerns the loading approach of the input feature map, the *Broadcast* one allow us to parallelize all the computations, and also reduces to just one the number of times we have to pick a pixel from the input memory. Instead for the weights, the stay approach is much better, since the kernel is not changing inside a full convolutional operation and it can be loaded just once. Finally we choosed a Migration approach for the output process, because passing the partial sum to a

neighboring PE is a good solution to enhance the energy efficiency, since the energy cost induced by an internal register transfer is much lower than memory operations. In the paper [5] has been shown how the BSM architecture is by far the best structure for what concern energy consumption between the various convolutional models.

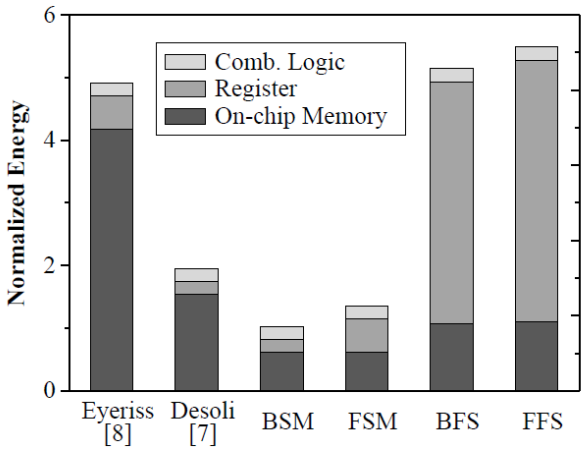

Figura 3.7. Normalized energy consumption for different convolutional models [5].

In order to exploit the properties that we described before on the BSM architecture, it was necessary to develop a special purpose structure able to complete the convolutional operation. The hardware architecture of our proposed accelerator is illustrated in the figure below, where it is possible to see all the different main blocks that will be described in detail in the following sections. As we can see the accelerator takes as input some images and stores them in the input memory. At the same time the weight memory is filled with the kernel filter weights. When the loading is finished, a start signal is send to the Control Unit that enables the Datapath with some Flag signals. After a predefined number of clock cycles the convolutional product of the image with the filter is over and the oFMAP is stored inside the Off-chip memory. At this point we can save this values and we can obtain the processed image.

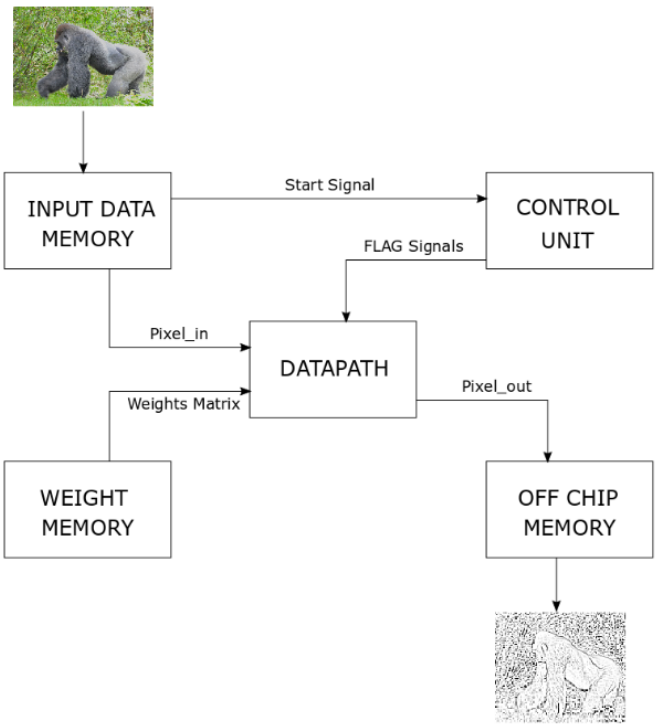

Figura 3.8. The proposed architecture.

3.3.1 Input Data Memory

The Input Data Memory (IDM) has two different tasks; firstly it has to load from a txt file all the pixels of the image(in our case is 128x128) and then it has to send all the pixels one by one to the Datapath where they will be processed. The inputs of this block are the main clock and an asynchronous reset signal, while as output beside the pixels, it has also a *start_sequence* signal that goes directly to the Control Unit in order to be sure that all the flags will be send in time. These flags are basically the write and the read enable necessary for the operations inside the on-chip memories

```
architecture behavioural of input_data_memory is

type rom_array is array (0 to m-1) of std_logic_vector(n-1 downto 0);    -- m = 16384
                                                                         -- n = 8
impure function read_data_ROM return rom_array is

    file infile1: text open READ_MODE is "./gorilla.txt";                -- read function from txt file
    variable buf1 :line;
    variable value1 : std_logic_vector (n-1 downto 0);
    variable ROM:rom_array:=(others=>(others=>'0'));
    variable I : integer range 0 to m ;

begin

        while not endfile(infile1) loop
           readline(infile1, buf1);
           hread(buf1, value1);
           ROM(I):= std_logic_vector(unsigned (value1) ); -- fill the matrix with the input feature values
           I := I+1;
        end loop;
    return ROM;
end function;
```

Figura 3.9. Read function of the iFMAP.

In our architecture the IDM is able to download just one image at time and then to flush it out when the output feature map is generated. Of course it could have been possible to save multiple images during the computation of the first one, but as said before, the goal is to keep the overall size of the memories small. For what concern the parallelism of the structure,every input pixel is represented on 9 bits, where 8 bit are for the actual representation and the first bit is for the sign. This is necessary because the multiplications inside the Datapath are signed since the values inside the kernel filter can be negative.

```
read_process: process (CLK,Rst)
variable address : integer range 0 to m;     -- m = 16384

begin
    if Rst = '1' then                          -- asynchronous reset
            address := 0;
            data_out <= (others=>'0');
            start_sequence <= '0';
    elsif (CLK'event and CLK = '1') then       -- reading process from input data memory
            if address = 0 then                --     ( 1 pixel x clock cycle)
               start_sequence <= '1';
               data_out <= '0'& ROM(address);
               address := address + 1 ;
            elsif(address >= m) then
               address:= 0;
            elsif (address < m and address > 0) then
               data_out <= '0' & ROM(address);
               address := address + 1 ;
            end if;
    end if;

end process read_process;
```

Figura 3.10. Output of the iFMAP.

3.3.2 Weight Memory

The Weight Memory (WM) works in a similar way of the IDM; it has as inputs the clock and the asynchronous reset, but since we have a *Stay* structure, all the weights have to be stored inside the PEs at the same time in order to exploits the full parallelism. In our case for a filter 3x3 we have nine outputs directly wired to the Datapath. Also for this memory the data are loaded from a txt file, but the parallelism of the outputs is different. In this case the outputs are on 7 bit, since the amplitude range of the weights is much smaller respect to the pixel; in the end using this structure we will have an internal parallelism of 16 bits that is quite acceptable for this kind of operations.

```
read_process: process (CLK,Rst)

variable address : integer range 0 to m;        -- m = 9
begin
    if Rst = '1' then
         address := 0   ;
         w_0 <= (others=>'0');
         w_1 <= (others=>'0');
         w_2 <= (others=>'0');
         w_3 <= (others=>'0');
         w_4 <= (others=>'0');
         w_5 <= (others=>'0');
         w_6 <= (others=>'0');
         w_7 <= (others=>'0');
         w_8 <= (others=>'0');
    elsif (CLK'event and CLK = '1') then
              if(address >= m) then
                    address:= 0;
              else                                        -- sending the first 7 bit
                    w_0 <= ROM(0)(6 downto 0);
                    w_1 <= ROM(1)(6 downto 0);
                    w_2 <= ROM(2)(6 downto 0);
                    w_3 <= ROM(3)(6 downto 0);
                    w_4 <= ROM(4)(6 downto 0);
                    w_5 <= ROM(5)(6 downto 0);
                    w_6 <= ROM(6)(6 downto 0);
                    w_7 <= ROM(7)(6 downto 0);
                    w_8 <= ROM(8)(6 downto 0);
              end if;
    end if;


end process read_process;
```

Figura 3.11. Output of the filter weights.

3.3.3 Datapath

The main computational unit of our Hardware accelerator is the Datapath (DP), that is mainly composed by three diffenrent blocks: Processing Elements (PE), On-chip memories and internal registers. The complete architecture is visible in fig. 3.12, while in the next sections the single blocks will be analyzed. From the picture is clear how the system works. Every PE has to operate with the current pixel and a predefined weight, of course since the weights for each image are always the same they can be saved just once.

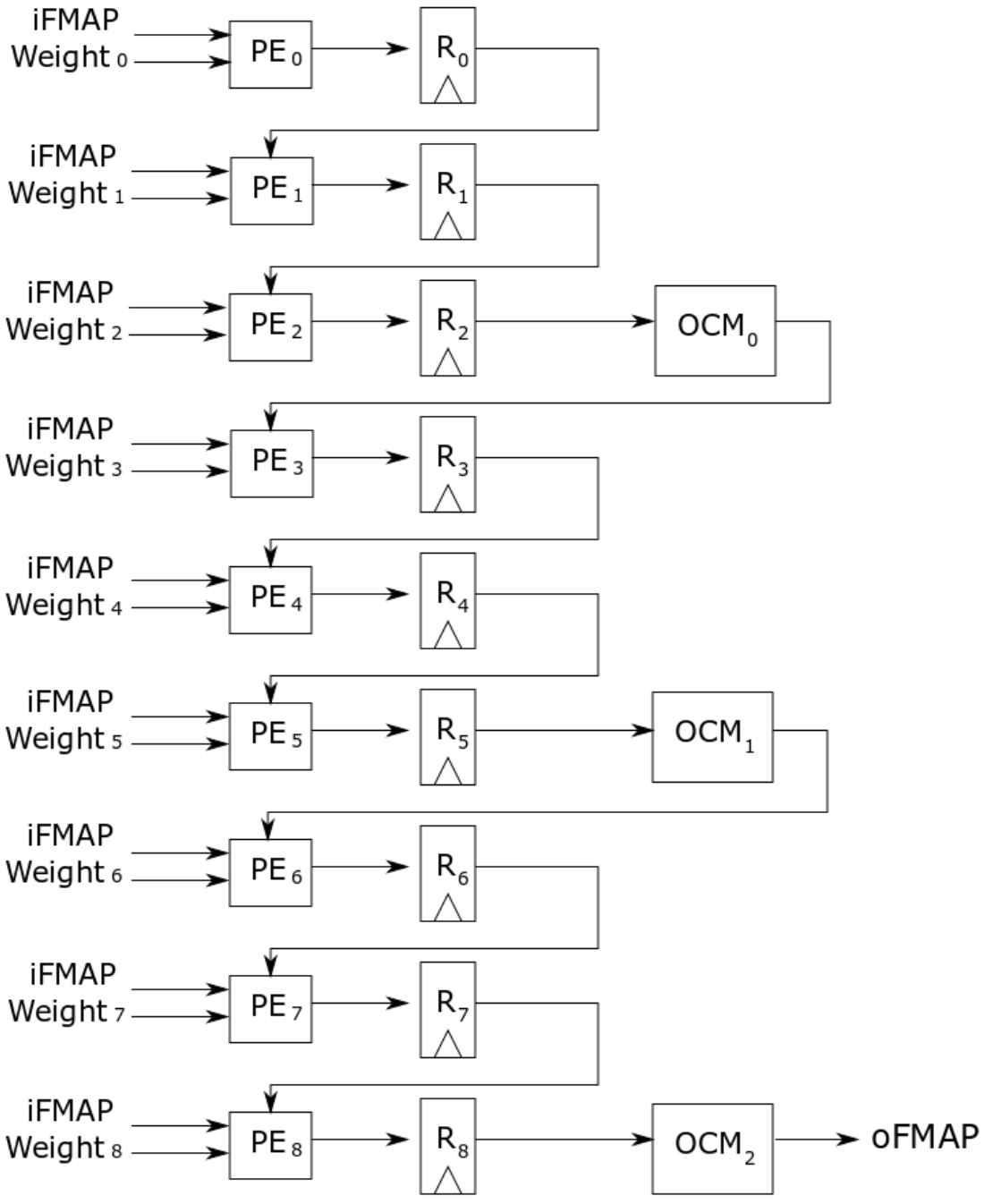

Figura 3.12. Datapath Block Diagram.

Processing Elements

This block is the heart of the computation inside the architecture, basically it implements a Multiply and Accumulate(MAC) operation. As it is clear from the Datapath image, there are nine PEs since the weight filter is 3x3 and we need nine convolutional product for each clock cycle. The internal parallelism of this unit is 16 bits, since the pixels are represented on 9 bits and the weights on 7 bits. Since the multiplication is the most complex operation, in the following sections we will present an architecture with a reduced internal parallelism that will allow to further reduce the computational power.

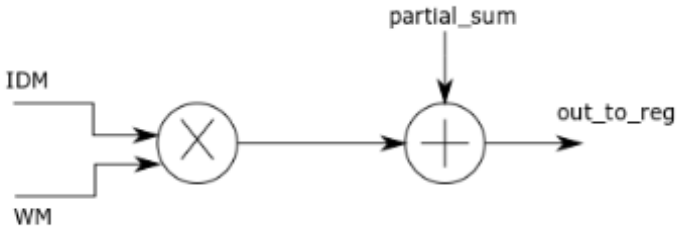

Figura 3.13. Processing Element.

On-chip memories

The on-chip memories are necessary since we decided to keep the intermediate results in storage and wait for the end of each row computations in order to sum up the partial results. As it is clear in the Datapath structure, we need three internal memories since the need to keep track of the partial value every three rows. For what concern the size of each memory, if we suppose to have as input feature map an image of 128x128 pixels, the internal memories will have to save (128-3+1)/1 = 126 values. This value can be computed using the equation 2.6 on the size of the oFMAP, if we use as filter size the value 3 and the stride equal to 1. The structure is similar to a standard FIFO, besides the clock and the reset; the memory is triggered by a Write Enable (WE) and a Read Enable(RE) that arrive from the Control Unit. The code structure of the memory is redundant on the controls flag since we had some problems during the synthesis and we needed valid addresses even when the machine was in idle state.

```
p_control : process (clock)

begin

  if (clock'event and clock = '1') then
    if reset = '1' then
       r_WR_INDEX <= 0;
       r_RD_INDEX <= 0;
    else                                          -- m = 126
      if (store_enable = '1') then                -- write address
         if r_WR_INDEX = m-1 then
            r_WR_INDEX <= 0;
         else
            r_WR_INDEX <= r_WR_INDEX +1 ;
         end if;
      else r_WR_INDEX <= 0;
      end if;
      if (output_enable ='1' ) then               -- read address
         if r_RD_INDEX = m-1 then
            r_RD_INDEX <= 0;
         else
            r_RD_INDEX <= r_RD_INDEX +1;
         end if;
      else r_RD_INDEX <= 0;
      end if;
      if store_enable = '1' then                   --write operation
         r_FIFO_DATA(r_WR_INDEX) <= data_in;
      end if;
      if output_enable = '1' then                  -- read operation
         data_out <= r_FIFO_DATA(r_RD_INDEX);
      end if;
    end if;
  end if;

end process p_control;
```

Figura 3.14. On-chip memories write and read operations.

3.3.4 Control Unit

The most complex unit in the architecture is without any doubt the Control Unit(CU), since it has been necessary to adapt all the internal flags of the machine to the re-scheduled DFG proposed at the beginning of the chapter.
The input of this unit are the main clock, the reset and the starting sequence signal from the IDM, while the outputs are all the WEs and REs destined to the on-chip memories plus an extra WE needed for the Off-chip memory(OCM). In the internal structure of this block we had to create some counters in order to keep track of the writing and forwarding sequences.

```
architecture behaviour of Control_Unit is

begin


On_chip_memory_2_write_process: process (clock)

variable counter : integer range 0 to m;      -- m = 16512
variable n : integer range 0 to p+2;          -- p = 126
                                              -- h = 16383
 begin

    if (clock'event and clock = '1')  then
        if reset = '1' then
            counter := 0;
            n:= 0;
        elsif  start_sequence = '1' then
            if  counter <= h then
                if ( counter >= ( p*n + 2*n) and counter <= ( p*n +2*n +1 )  then
                  store_enable_OM2 <= '0';
                else
                  store_enable_OM2 <= '1';
                end if;
                if ( counter = ( p + (p+2)*n ) ) then
                  n:= n+ 1;
                end if;
                counter := counter +1 ;
            elsif counter >h then
                store_enable_OM2 <= '0';
            end if;
        end if;
    end if;

end process On_chip_memory_2_write_process;
```

Figura 3.15. Generation of a generic write enable signal (WE).

3.3.5 Off-Chip Memory

The Off-chip Memory collects all the computed pixels that arrive from the Datapath and through a Write Enable signal, always managed from the CU, is able to write a txt file with the output feature map. This oFMAP will be sent to Matlab that will show the new image after the convolutional operation.
The structure of this memory is always similar to a FIFO, but a *End_Sim* signal is added in order to be sure that all the operations on the current image are finisced. In the next chapter we will see how this signal will be fundamental in order to have an estimation of the switching activity of the system and then of the power consumption.

```
read_process : process (clock,reset)

variable address_in : integer range 0 to m;  --m = 15876

begin

 if reset = '1' then
    RAM<=(others=>(others=>'0'));
    address_in := 0;
 elsif (clock'event and clock = '1')  and store_enable = '1'  then
    if store_enable = '1' then
      RAM(address_in) <= data_in;
      address_in := address_in + 1;
      if(address_in > m-1) then
         address_in:= 0;
         go_to_output <= '1';
      end if;
    end if;
 end if;

end process read_process;

output_process : process (go_to_output)

   file res_fp : text open WRITE_MODE is "./results.txt";
   variable line_out : line;
   variable I : integer range 0 to 15877 ;

 begin

      end_sim <= '0';
      if (go_to_output = '1') then
         for I in 0 to 15875 loop
         write(line_out, conv_integer(signed(RAM(I))));
         writeline(res_fp, line_out);
         end loop;
         end_sim <= '1';
      end if;
end process output_process;
```

Figura 3.16. Off-chip memory read and output operations.

3.4 Simulations

The simulation phase is one of the most intense and wide inside the design flow of every hardware architecture. The Hardware simulator that will be used in this thesis is mainly *ModelSim*. In the following sections we will briefly describe the simulation environment and we will analyze different waveforms in order to understand the timing diagram of the entire architecture.
ModelSim is a multi-language HDL simulation environment developed by Mentor Graphics; in our design we will mainly use only VHDL and Verilog as hardware description languages, but it supports also SystemC and SystemVerilog. One of the main advantages of this environment is the possibility of mixing in a transparent way blocks in VHDL and Verilog in one design. It gives us the possibility to simulate behavioural, RTL and gate-level code; the ModelSim debug environments broad set of intuitive capabilities make it the choice for both ASIC and FPGA design. In this thesis we will carry out the first solution, trying to design of an Application Specific Integrated processor(ASIC).

3.4.1 Modelsim Simulations

After the coding phase in VHDL of our hardware architecture, an intense phase of simulation was needed.In the following pages will be shown only simulations of the entire structure, but of course, in order to debug the whole architecture, a testbench for each basic block has been created and tested.
In order to easily repeat multiple simulations some simple scripts have been prepared,like this one.

```
vcom -93 -work ./work ../src/clk_gen.vhd
vcom -93 -work ./work ../src/input_data_memory.vhd
vcom -93 -work ./work ../src/weight_memory.vhd
vcom -93 -work ./work ../src/PE.vhd
vcom -93 -work ./work ../src/Register_N_bit.vhd
vcom -93 -work ./work ../src/on_chip_output_memory.vhd
vcom -93 -work ./work ../src/Control_Unit.vhd
vcom -93 -work ./work ../src/Datapath.vhd
vcom -93 -work ./work ../src/Off_chip_output_memory.vhd
vcom -93 -work ./work ../src/top_entity.vhd

vlog -work ./work ../tb/tb.v

vsim work.TestBench
```

Figura 3.17. Script per simulare i listati VHDL.

It is interesting to notice that while all the building blocks are in VHDL, the testbench is in verilog since , as was mentioned before, it is possible to write in different languages and then the simulator will be able to bind and planarize all the difference.
For sake of completeness in the following image we can see an extract from the TestBench, just to point out the differences between Verilog and VHDL.

```
module TestBench ();

    wire [8:0] pixel_in_i;
    wire [6:0] weight_0_in_i;
    wire [6:0] weight_1_in_i;
    wire [6:0] weight_2_in_i;
    wire [6:0] weight_3_in_i;
    wire [6:0] weight_4_in_i;
    wire [6:0] weight_5_in_i;
    wire [6:0] weight_6_in_i;
    wire [6:0] weight_7_in_i;
    wire [6:0] weight_8_in_i;
    wire [15:0] pixel_out_i;
    wire CLK_i;
    wire Rst_i;
    wire start_sequence_i;
    wire store_enable_ocom_in_i;
    wire END_SIM_i;

/*initial begin
   $read_lib_saif("../saif/library.saif");
   $set_gate_level_monitoring("on");
   $set_toggle_region(TE);
   $toggle_start;
end*/

    clk_gen CG(
          .END_SIM(END_SIM_i),
          .CLK(CLK_i),
          .RST_n (Rst_i));

    input_data_memory IDM(
        .CLK(CLK_i),
        .Rst(Rst_i),
        .start_sequence(start_sequence_i),
        .data_out(pixel_in_i));

    weight_memory WM(
        .CLK(CLK_i),
        .Rst(Rst_i),
        .w_0(weight_0_in_i),
        .w_1(weight_1_in_i),
        .w_2(weight_2_in_i),
        .w_3(weight_3_in_i),
        .w_4(weight_4_in_i),
        .w_5(weight_5_in_i),
        .w_6(weight_6_in_i),
        .w_7(weight_7_in_i),
        .w_8(weight_8_in_i));

    top_entity TE(
        .clock(CLK_i),
        .reset(Rst_i),
        .start_sequence(start_sequence_i),
        .w_0(weight_0_in_i),
        .w_1(weight_1_in_i),
        .w_2(weight_2_in_i),
        .w_3(weight_3_in_i),
        .w_4(weight_4_in_i),
        .w_5(weight_5_in_i),
        .w_6(weight_6_in_i),
        .w_7(weight_7_in_i),
        .w_8(weight_8_in_i),
        .pixel_data(pixel_in_i),
        .store_enable_OCM(store_enable_ocom_in_i),
        .new_pixel(pixel_out_i));

    off_chip_output_memory OCOM(
        .clock(CLK_i),
        .reset(Rst_i),
        .store_enable(store_enable_ocom_in_i),
        .data_in(pixel_out_i),
        .end_sim( END_SIM_i));

/*always @ ( END_SIM_i ) begin
  if (END_SIM_i) begin
  $toggle_stop;
  $toggle_report("../saif/top_entity_back.saif", 1.0e-10, "TestBench.TE");
  end
end*/

endmodule
```

Figura 3.18. TestBench in Verilog.

In the following images we can have a close look to the entire simulation; of course since a full convolutional operation requires 16384 clock cycles we need multiple images to highlight the main features. This number is mainly due to the size of the input feature map, since we have to analyze a pixel in each clock cycle and we have 128x128 = 16384 pixels. The extra 129 clock cycles are the internal latency of the architecture due the partial sum accumulation inside the on-chip memories.

In fig. 3.19 it can be noticed a full simulation of the structure; now let's try to analyze the different internal signals. From top to bottom we have the main clock and the reset, and the start_sequence signal, then there are all the input of the structure as the iFMAP's pixel and the weights. In the central part it can be seen all the control signals from the CU to the DP necessary to create the correct timing inside the structure. Finally we can see the End_Sim signal and the output of the structure. In the final clock cycle all the saved outputs will be send to a txt file; after that, Matlab will be able to read the results and show all the processing work done on the images.

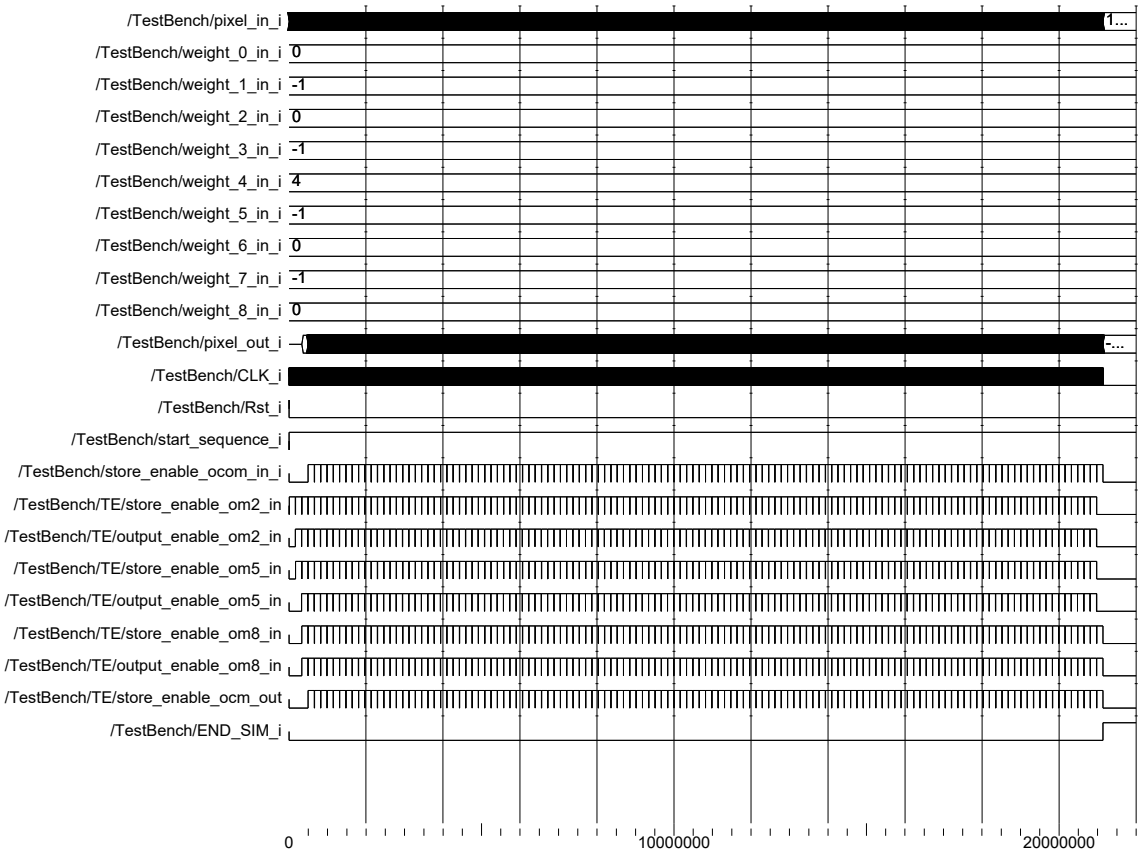

Figura 3.19. Full Simulation.

In fig 3.20 there is an highlight of the initial part of the simulation; if we focus on the reset and the start signals we can see how when the reset goes down, at the next rising edge of the clock the the start_sequence signal goes high and the input pixel and weights start to flow inside the architecture.

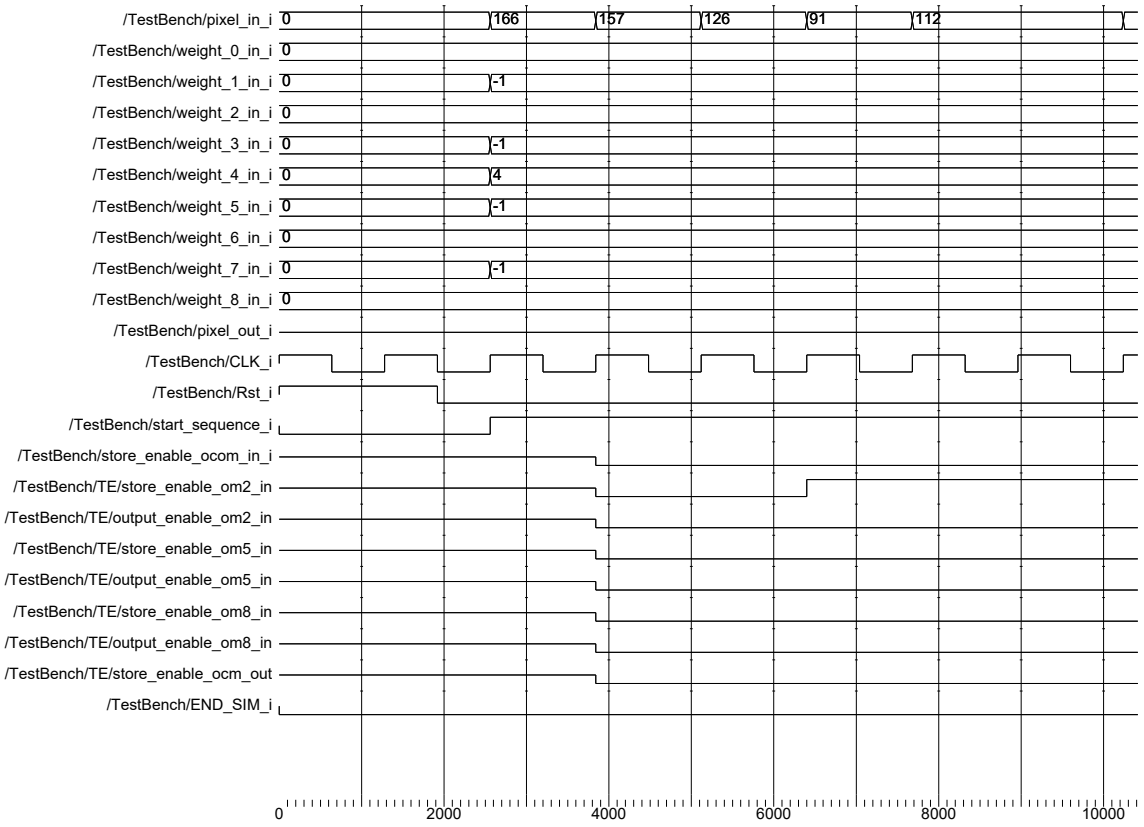

Figura 3.20. Start of the sequence.

The output pixel remains undefined for 387 clock cycles; this is the latency of the entire chain of registers and FIFO memories. The first valid value is 53 and it can be seen from the figure below.

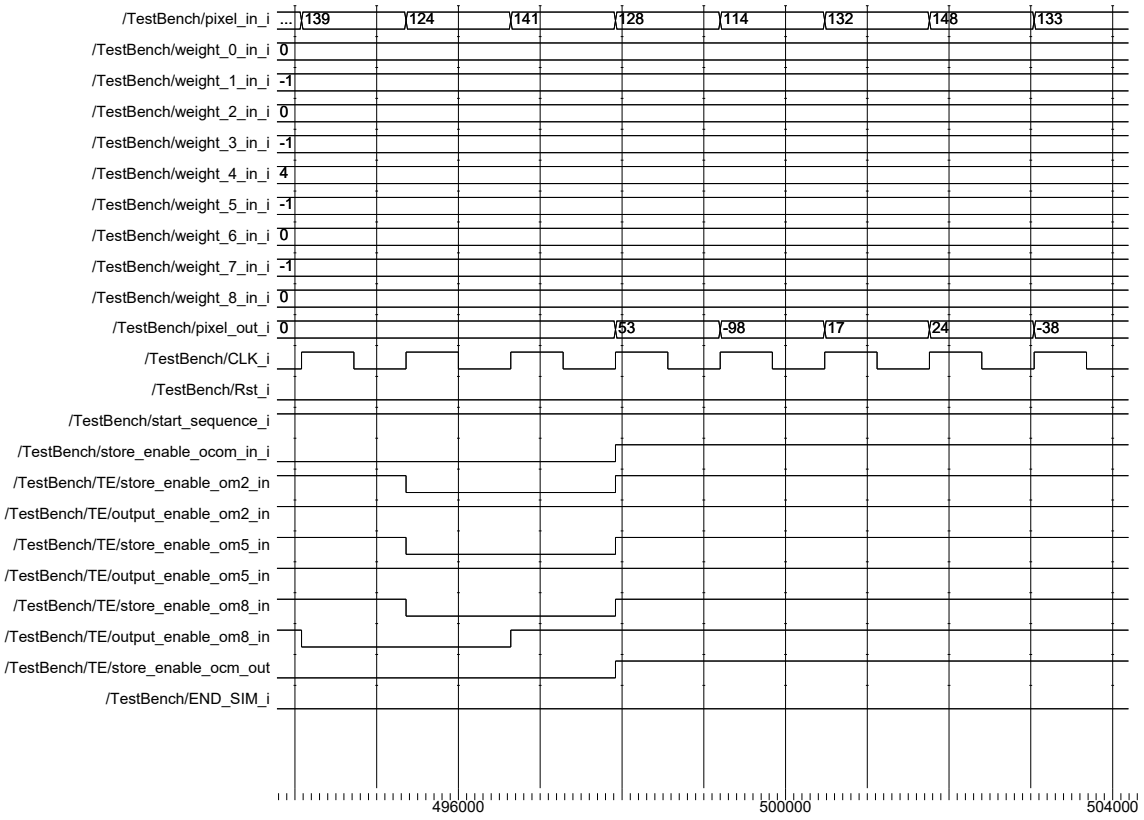

Figura 3.21. First output valid value (53).

Finally if we highlight the end of the simulation we can see how the End_Sim signal allows to block the internal clock and to write the final results when the last value is computed.

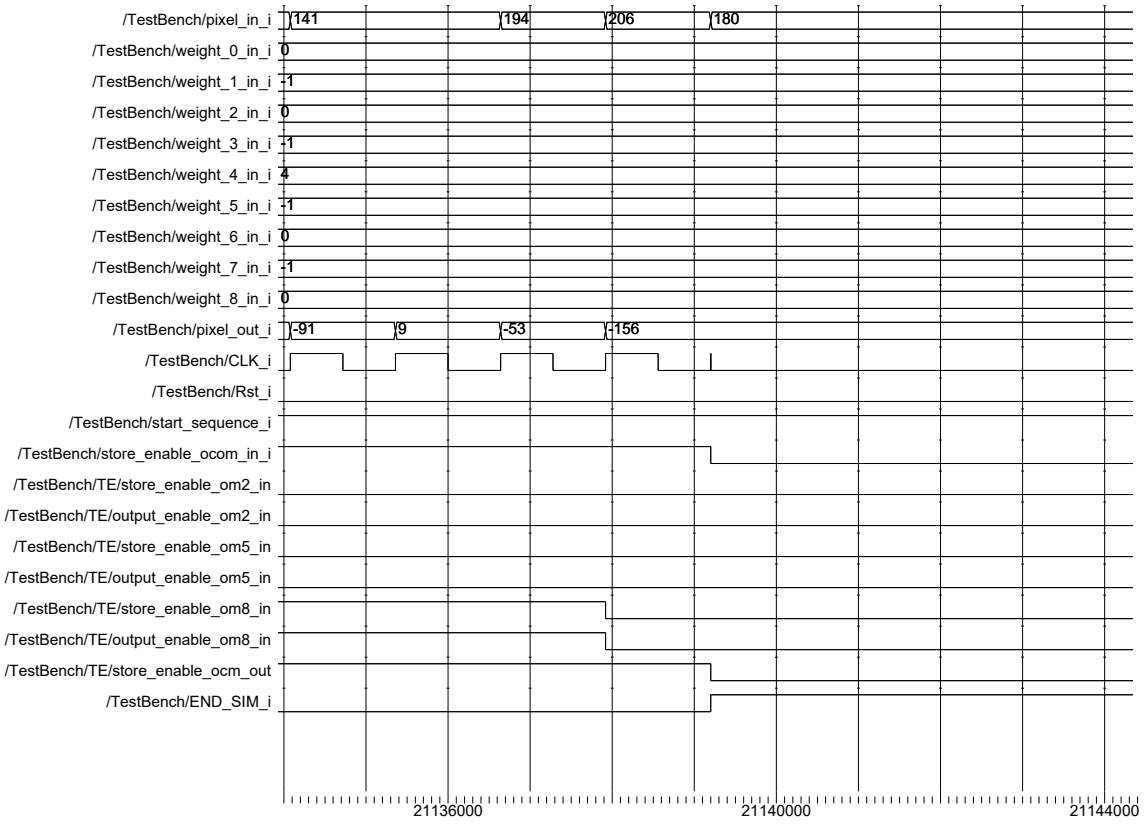

Figura 3.22. End of the simulation.

Capitolo 4

Hardware implementation and applications

This chapter describes all the different phases of the hardware implementation of our accelerator. Firstly,Section 4.1 provides all the steps required to synthetize the architecture through *Synopsys Design Compiler*. Then the analysis on area, power and speed is carried out. In Section 4.2 it is analyzed a different architecture with a reduced parallelism in order to compare different structures. Section 4.3 provides a description of the different weight filters tested and the resulting images coming out from the overall computation. Finally the last section provides a comparison with previous developed hardware accelerators.

4.1 Synthesis

Synthesis is a complex task consisting of many phases and requires various inputs in order to produce a functionally correct netlist. The following section presents the basic synthesis flow with Synopsys Design Compiler. In order to be able to generate the netlist it is necessary to have synthesizable and functionally correct HDL description of the hardware structure. The library that we will use to synthetize the previous described accelerator is the UMC 65nm, that is granted thanks to the partnership of the *United Mycroelectronic Corp* with the *Politecnico of Turin*.

4.1.1 Introduction to Synopsys Design Compiler

The Design Compiler tool is the core of the Synopsys synthesis products. Design Compiler optimizes designs to provide the smallest and fastest logical representation of a given function. It comprises tools that synthesize your HDL designs into optimized technology-dependent, gate-level designs. It supports a wide range of flat and hierarchical design styles and can optimize both combinational and sequential designs for speed, area, and power. In the following picture we can see an overview of how Design Compiler fits into the design flow.
Synthesis with Design Compiler include the following main tasks: reading the VHDL source files in the design, applying constraints, optimizing the design, analyzing the results and saving the obtained design. These tasks will be described in detail in the following sections.

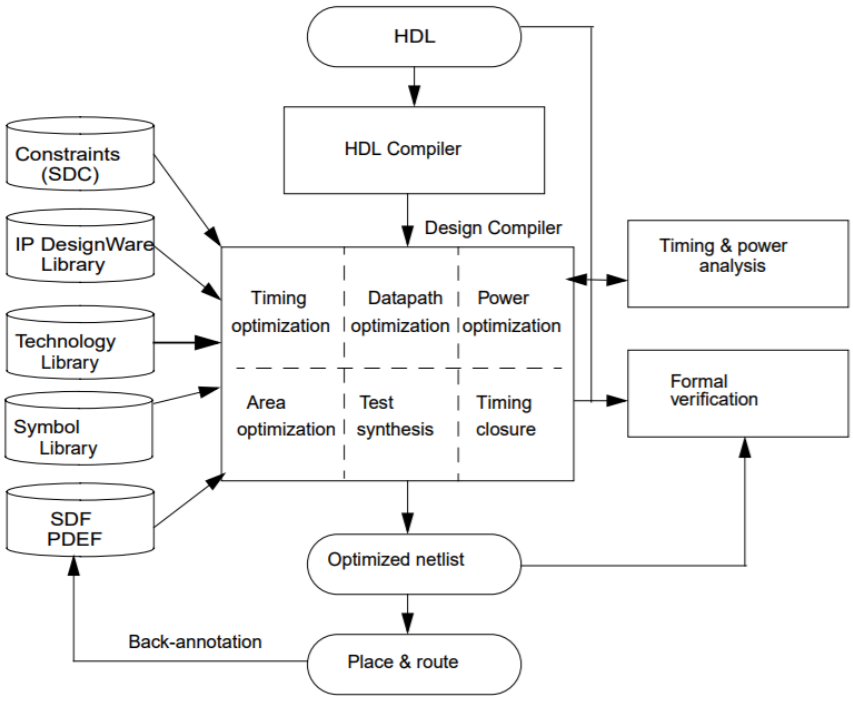

Figura 4.1. Design Flow

4.1.2 Reading VHDL source files

The first task in the synthesis is to read the design into Design Compiler memory. Reading in an HDL design description consist of two tasks: analyzing and elaborating the description. The analysis command (analyze) performs the following tasks: reads the HDL source, checks it for syntactical errors and creates HDL library objects in an HDL-independent intermediate format while saving these intermediate files in a specified location. The elaboration command (elaborate) instead translates the design into a technology-independent design from the intermediate files produced during the analysis, allows changing of parameter values (generics) defined in the source code, replaces the HDL arithmetic operators in the code with DesignWare components. The uniquify and link commands are necessary in order to deal with multiple instances of the same block.

```
analyze -f vhdl -lib WORK ../src/PE.vhd
analyze -f vhdl -lib WORK ../src/Register_N_bit.vhd
analyze -f vhdl -lib WORK ../src/on_chip_output_memory.vhd
analyze -f vhdl -lib WORK ../src/Datapath.vhd
analyze -f vhdl -lib WORK ../src/Control_Unit.vhd
analyze -f vhdl -lib WORK ../src/top_entity.vhd
set power_preserve_rtl_hier_names true
elaborate top_entity -arch behavior -lib WORK > ./elaborate.txt
uniquify
link
```

Figura 4.2. Script to analyze the HDL files

It is interesting to notice that in the elaboration reports we can see the number and the type of memory elements Design Compiler thinks it should infer. At this point, if the elaboration completed successfully, the design is represented in an internal, equation-based, technology-independent design format.

4.1.3 Applying constraints

The next task is to define the constraints of out design. Constraints are the instructions that the designer gives to Design Compiler in order to mark out the limits what the synthesis tool can or cannot do with the design or how the tool behaves. Usually this informations can be derived from the design and timing specifications. Firstly we have to set the constraints on the main clock and in order to simulate a real behaviour we have to define some uncertainty parameters assuming that in the architecture we could have jitter and skew problems. We have also to set output load available in the library in order to compute a timing analysis. In this particular case it has been chosen the input capacitance of a buffer.

```
create_clock -name MY_CLK -period 1.28 clock
set_dont_touch_network MY_CLK
set_clock_uncertainty 0.07 [get_clocks MY_CLK]
set_input_delay 0.5 -max -clock MY_CLK [remove_from_collection [all_inputs] clock]
set_output_delay 0.5 -max -clock MY_CLK [all_outputs]
set OLOAD [load_of uk65lscllmvbbr_120c25_tc/BUFM4R/A]
set_load $OLOAD [all_outputs]
```

Figura 4.3. Applying constraints

One of the goal in this first synthesis is to find the maximum possible frequency, so we have to force to 0 the clock period and then from the timing result we can

take the slack violation as minimum clock cycle.
After that, it possible to run a new synthesis with this minimum clock cycle and verify if the obtained slack is equal to zero without any violation.

4.1.4 Optimizing and Analyzing the reports

In the next phase we issue the compile ultra command in order to exploit all the possible optimizations that the tool is able to achieve. And finally we can write into some files the report on the area,the timing and the resources used by our accelerator.
In the final synthesis it has been found a minimum clock of 1.28 ns, so the derived maximum frequency was 781.2 MHz.

```
Path Group: MY_CLK
Path Type: max

Des/Clust/Port     Wire Load Model       Library
------------------------------------------------
top_entity         wl0                   uk65lscllmvbbr_120c25_tc

Point                                    Incr       Path
-----------------------------------------------------------
clock MY_CLK (rise edge)                 0.00       0.00
clock network delay (ideal)              0.00       0.00
input external delay                     0.50       0.50 f
w_8[5] (in)                              0.00       0.50 f
U642/Z (XOR2M8RA)                        0.08       0.58 r
U1169/Z (ND2M16RA)                       0.03       0.61 f
U397/Z (OAI22M6RA)                       0.03       0.64 r
U158/ICO (AD42M2RA)                      0.11       0.76 r
U300/S (AD42M4RA)                        0.16       0.91 f
U1247/Z (NR2M2R)                         0.05       0.96 r
U1246/Z (NR2M4R)                         0.03       0.99 f
U553/Z (AOI21M8R)                        0.05       1.04 r
U299/Z (INVM4R)                          0.03       1.06 f
U1618/Z (AOI21M2R)                       0.04       1.11 r
U1622/Z (XOR2M2RA)                       0.05       1.16 r
U1623/Z (AN2M6R)                         0.04       1.20 r
DP/Reg8/q_reg[9]/D (DFQM2RA)             0.00       1.20 r
data arrival time                                   1.20

clock MY_CLK (rise edge)                 1.28       1.28
clock network delay (ideal)              0.00       1.28
clock uncertainty                       -0.07       1.21
DP/Reg8/q_reg[9]/CK (DFQM2RA)            0.00       1.21 r
library setup time                      -0.01       1.20
data required time                                  1.20
-----------------------------------------------------------
data required time                                  1.20
data arrival time                                  -1.20
-----------------------------------------------------------
slack (MET)                                         0.00
```

Figura 4.4. Report timing.

In the fragment below we can see how the area of the synthetized architecture is 85812.85 μm^2.

```
Library(s) Used:

uk65lscllmvbbr_120c25_tc (File: /software/dk/umc65/Core-lib_LL_Multi-Voltage_Reg.Vt/synopsys/uk65lscllmvbbr_120c25_tc.db)

Number of ports:                          210
Number of nets:                         22387
Number of cells:                        21780
Number of combinational cells:          15332
Number of sequential cells:              6444
Number of macros/black boxes:               0
Number of buf/inv:                        785
Number of references:                     114

Combinational area:              41699.520826
Buf/Inv area:                     1013.760032
Noncombinational area:           44113.320971
Macro/Black Box area:                0.000000
Net Interconnect area:      undefined  (Wire load has zero net area)

Total cell area:                 85812.841797
```

Figura 4.5. Report area

From the previous simulation we also found that a complete convolutional operation on one input feature map takes 16384 clock cycles. Multiplying this value with the minimum clock cycles we can obtain the throughput of the architecture and analyze the consequences in real-time systems.
The full processing of one image will take 1.28 ns $\times$ 16384 = 2097152 ns; for sake of simplicity we can suppose that the whole operation takes 21 μs. This means that this hardware accelerator can analyze images with a throughput of more than 47 thounsands frames per second (FPS). Since even the most recent video-processing cameras have a maximum of 200 FPS there is no need to push too much on the frequency, but we can try to reduce it in order to decrease area and power consumption.
So the next step was to synthetize again the whole architecture with a frequency equal to $F_{max}/4$; since the dynamic power is linearly related to the frequency we expect a linear reduction of the overall power consumption. In the table below we can appreciate how the reduction is not exactly in linear scale, since we have some leakages in the structure,but its almost one third of the previous one.

Area[μm^2]	T_{clock}[ns]	F_{max}[MHz]	Power[mW]
80892.01	5.12	195.3	15.42

4.1.5 Saving the synthetized design

In the final step we can save the data required to complete the design and to perform the switching-activity power estimation. Then we have to export the netlist in Verilog and save the design in order to be simulated and verified.

```
ungroup -all -flatten
change_names -hierarchy -rules verilog
write_sdf ../netlist/top_entity.sdf
write -f verilog -hierarchy -output ../netlist/top_entity.v
write_sdc ../netlist/top_entity.sdc
```

Figura 4.6. Saving the current Design

4.1.6 Simulation of the Netlist

After multiple synthesis, in order to verify the final design of the architecture, a further simulation was necessary. In this case through another simple script it was possible to test the generated Netlist in Verilog. This step is always mandatory since even if the architecture seems to work correctly through the simulations in ModelSim, after the synthesis there could be some problems of undefined nets.

```
vlog -work ./work /software/dk/umc65/Core-lib_LL_Multi-Voltage_Reg.Vt/verilog/uk65lscllmvbbr_sdf21.v

vcom -93 -work ./work ../src/clk_gen.vhd
vcom -93 -work ./work ../src/input_data_memory.vhd
vcom -93 -work ./work ../src/weight_memory.vhd
vcom -93 -work ./work ../src/Off_chip_output_memory.vhd

vlog -work ./work ../netlist/top_entity.v
vlog -work ./work ../tb/tb.v

vsim -t 1ps -L ./work -sdftyp /TestBench/TE=../netlist/top_entity.sdf  work.TestBench
```

Figura 4.7. Script to simulate the Netlist

This situation is mainly due to the fact that while ModelSim applies as standard value 0 to all its undefined nets, in the synthesis if we have something initially undefined or not correctly clocked we could have some internal value in high impedance. If this happens we could have a propagation of the error inside all the pipeline and in the end the output of the entire system will be undefined.

In our case we had an issue with the control flags coming from the Control Unit; in fact since the Datapath was expecting always valid signals,when the machine was down, the internal on-chip memories weren't able to sample the data and all the internal registers were in high impedance. This problem was solved adding the possibility to reset to 0 the internal registers when the machine was in standby mode(waiting for a image).

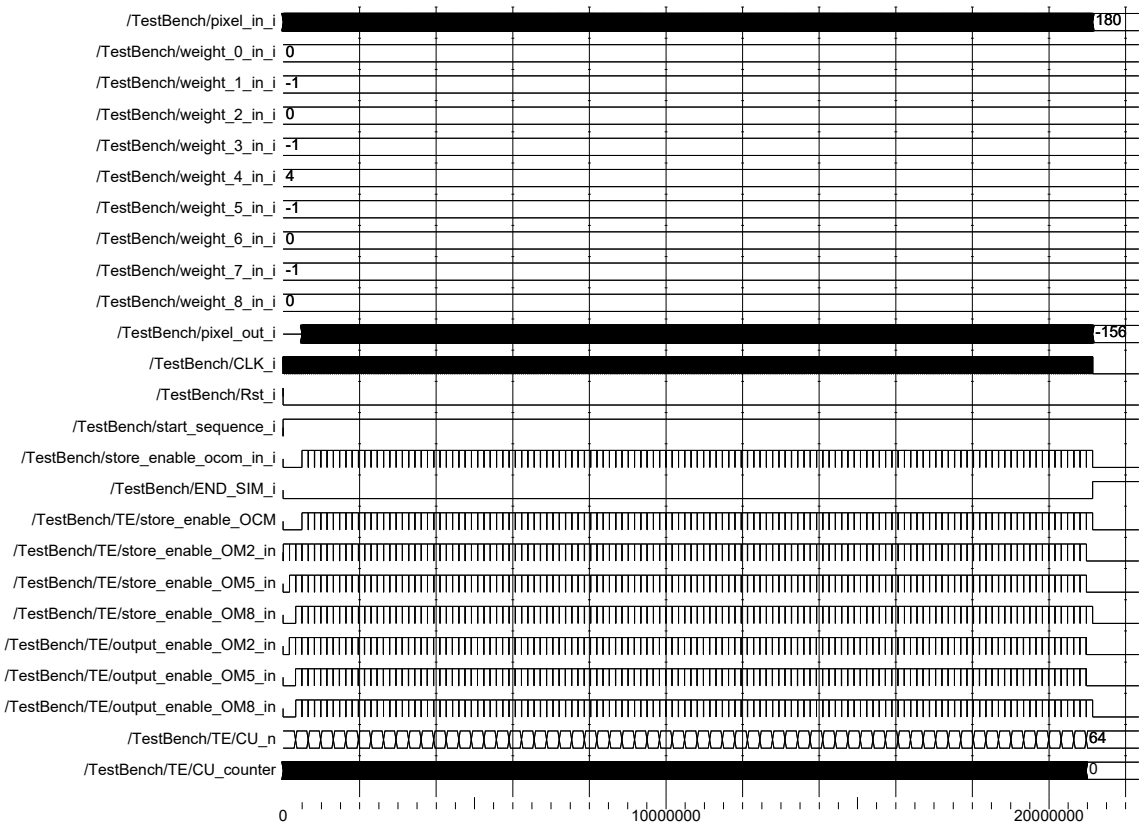

Figura 4.8. Simulation of the Netlist

4.1.7 Switching-activity-based power consumption estimation

The final step of the testing phase was the estimation of power consumption through a switching activity model implemented jointly by ModelSim and Synopsys. The switching activity of a system is defined as the probability that a node inside the architecture is changing its value in a determined clock cycle; so it is going from 0 to 1 or 1 to 0.

In order to complete this measurement we had to follow different steps. Firstly we had to create a file where Synopsys was able to extract the technological libraries. Then we had to modify the testbench in Verilog in order to add statements to get the switching activity.

```
initial begin
  $read_lib_saif("../saif/library.saif");
  $set_gate_level_monitoring("on");
  $set_toggle_region(TE);
  $toggle_start;
end

always @ ( END_SIM_i ) begin
 if (END_SIM_i) begin
 $toggle_stop;
 $toggle_report("../saif/top_entity_back.saif", 1.0e-10, "TestBench.TE");
 end
end
```

Figura 4.9. Code to obtain switching-activity

In the code its clear how the End_Sim signal is necessary in order to define a boundary in which the power simulation has to be performed. Finally we had to launch ModelSim and through another script it was possible to generate a file with the nets informations.

```
vlog -work ./work /software/dk/umc65/Core-lib_LL_Multi-Voltage_Reg.Vt/verilog/uk65lscllmvbbr_sdf21.v

vcom -93 -work ./work ../src/clk_gen.vhd
vcom -93 -work ./work ../src/input_data_memory.vhd
vcom -93 -work ./work ../src/weight_memory.vhd
vcom -93 -work ./work ../src/Off_chip_output_memory.vhd

vlog -work ./work ../netlist/top_entity.v
vlog -work ./work ../tb/tb.v

vsim -t 1ps -L ./work -sdftyp /TestBench/TE=../netlist/top_entity.sdf -pli /software/synopsys/syn_current/auxx/syn/power/vpower/lib-linux/libvpower.so work.TestBench
```

Figura 4.10. Script to compute the power analysis

In the end we had to go back to Synopsys design compiler and generate the report on the power consumption. As it is clear in the image below, the power consumption in out architecture is around 41.36 mW, of course this result has been computed using as reference the maximum frequency.

```
  Cell Internal Power  =  38.3965 mW   (93%)
  Net Switching Power  =   2.9621 mW    (7%)
                         ---------
Total Dynamic Power    =  41.3587 mW  (100%)

Cell Leakage Power     =   6.5132 uW


                 Internal         Switching           Leakage            Total
Power Group      Power            Power               Power              Power   (   %    )  Attrs
--------------------------------------------------------------------------------------------------
io_pad             0.0000           0.0000              0.0000             0.0000  (   0.00%)
memory             0.0000           0.0000              0.0000             0.0000  (   0.00%)
black_box          0.0000           0.0000              0.0000             0.0000  (   0.00%)
clock_network      0.0000           0.0000              0.0000             0.0000  (   0.00%)
register          36.5592           0.2099          3.1590e+06            36.7722  (  88.90%)
sequential         0.0000           0.0000              0.0000             0.0000  (   0.00%)
combinational      1.8371           2.7522          3.3542e+06             4.5927  (  11.10%)
--------------------------------------------------------------------------------------------------
Total             38.3963 mW        2.9622 mW     6.5132e+06 pW         41.3649 mW
```

Figura 4.11. Report Power

4.2 Architecture with reduced parallelism

In the following section a further optimization has been developed, in order to increase the performances in terms of speed and power without losing any accuracy in the results. Since the main operations of our convolutional accelerator were multiplications and addition, a intuitive way to decrease the complexity was to reduce the internal parallelism of the structure. The main problem was that the bits for the input feature map were already minimum, so instead of 7 bits for the weight filter we decided to work with only 5 bits. This solution will reduce the overall internal parallelism to 14 bits instead of 16 and in order to make a comparison all the previous steps were repeated.

Area [μm^2]	Clock Period [ns]	Max Frequency [MHz]	Power [mW]
79142.77	1.13	884.9	39.15

Changing the internal parallelism reduced of 13% the minimum clock cyle and reduced the overall area of around 8.4%. The total power consumption was reduced of 5.6% even if the frequency increased. Then if we try to synthetize this architecture with a clock cycle equal to 1.28(the maximum of the first architecture), we will have the following results.

Area [μm^2]	Clock Period [ns]	Frequency [MHz]	Power [mW]
73919.52	1.28	781.2	36.87

Decreasing the frequency of the system allows to the synthetizer to relax the constraints inside the architecture and it enables a save in terms of area and power. Of course the main drawback of the reduced parallelism solution is that the range of values that can be represented by the filter are between -16 and +15. As we will see in the next section the major part of the analyzed weight filter are inside this boundaries, but for some particular cases this representation is not enough and we have to stick with the original architecture.

4.3 Application of different weight filters

As we mentioned in the previous chapters, it is possibile to modify the weights inside a kernel filter in order to highlight particulare features of an image or a set of images. In this section we will analyze some pictures inside the Dataset of 2017 ILSVRC and we will apply different filters to our original architecture. Before that we can just define a couple of things about the structure af a *kernel filter*.
The convolutional kernel is a small matrix, in our case we consider a 3x3 matrix, with a defined number in each cell and a particular number in the middle of the matrix called *anchor* point.

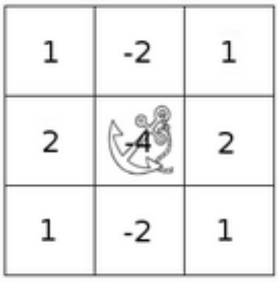

Figura 4.12. Anchor Point

This kernel slides over an image and the convolutional operation is computed; the *anchor* point is used to determine the position of the kernel with respect to the image. The anchor point starts at the top-left corner of the image and moves over each set of stride pixels sequentially. At each position, the kernel overlaps a few pixels on the image. Each overlapping pair of numbers is multiplied and added. Finally, the value at the current position is set to this sum. The main problem of this

structure are corners and edges, but as we exploited in our architecture, they can be simply ignored and preselected to 0 in order to go on with the overall computation.

-1	-1	-1
-1	8	-1
-1	-1	-1

Figura 4.13. Edge Detection Filter

-1	-2	-1
0	0	0
1	2	1

Figura 4.14. Sobel Filter

0	-1	0
-1	4	-1
0	-1	0

Figura 4.15. Laplacian Filter

Now we can analyze the effects on the images applying different kernel filters; the first kernel applied is called *Edge Detection* Operator,and it is clear how it tracks the edges inside the pictures.

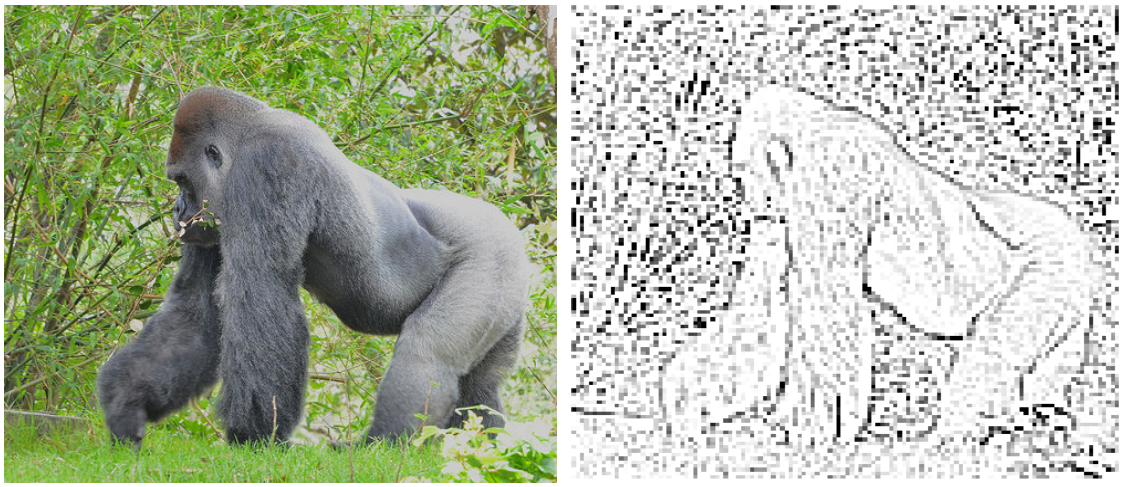

Figura 4.16. Application of the Edge Detection Operator

Another tested filter is the one defined as *Sobel*. This Operator is very effective against noise to noise since it has a smoothing effect.

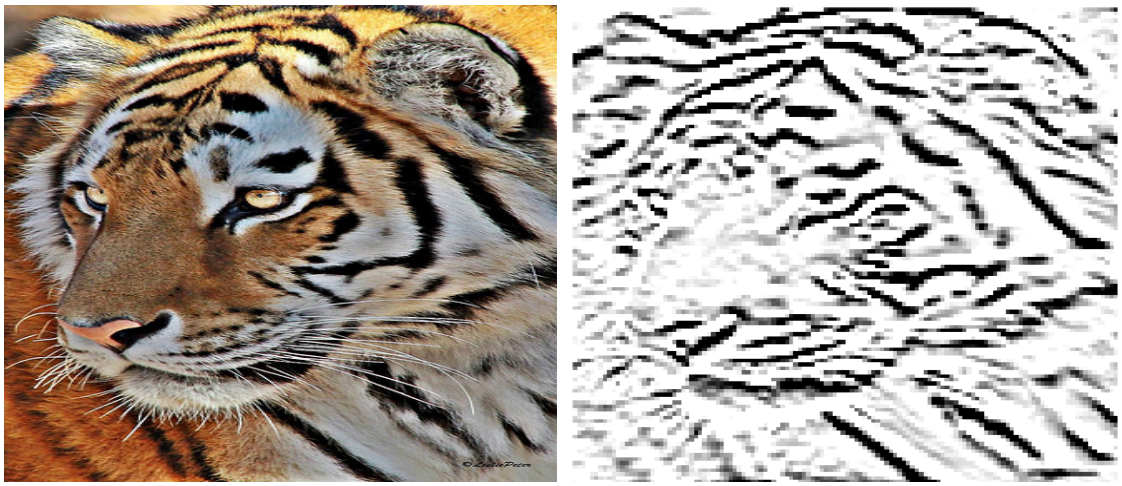

Figura 4.17. Application of the Sobel Operator

Finally we test the convolutional operation with the *Laplacian* Operator defined as the second derivative of the image. This Operator is much more sensitive to noise so it is used only for academic purpose.

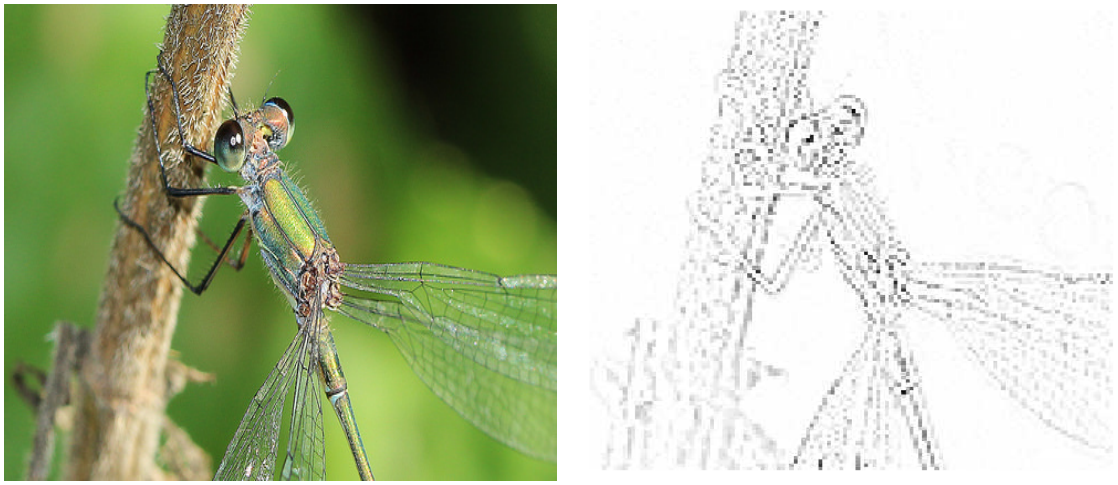

Figura 4.18. Application of the Laplacian Operator

The main goal of applying these filters is to be able to extract particular features to help the post-processing classification. As we mentioned before there will be necessary multiple stages of convolution in order to fully process a single image, but in any case if we are able to optimize a single layer, it will be the same for all the others.

4.4 Comparison with previous Hardware Accelerators

At the end of our work we can spend a few words on how this hardware accelerator can fit the giant branch of accelerators developed to boost Neural Networks from all points of view.
First of all we have to define its characteristic, so we can easily say that in order to make a fair comparison we will define a few parameters: the technological node, that in our case is the 65 nm UMC technology, the total area of the structure,the power consumption, the clock frequency and the bit width of our architecture that can be seen as the internal parallelism.
In the table below we have listed our feature comparing with some other papers that worked on the same subject.

Accelerators	The proposed structure	Cambricon-X[6]	EIE [7]	SCNN[8]
Tech. node[nm]	65	65	28	16
Area[mm^2]	0.0809	11.32	15.95	7.87
Power[mW]	41.37	1220	590	n/a
Clk Frequency[MHz]	781	1000	1200	1000
Bit width	16	16	16	16

The first thing to notice is that our structure seem to have much lower area and of course power consumption,but this is due to the fact that our hardware accelerator works mainly on one layer of a CNN while the others trained their accelerator on a complete CNN structure. If we wanted to make a fair comparison we would have to test our architecture in a full CNN environment; this wasn't possible because it would have required to build also a full network and not only a convolutional layer. In the future works could be very useful to try to exploit this direction, in order to make a much easier comparison with all the other similar works.

Capitolo 5

Results and Conclusions

This chapter provides an overall analysis of this thesis. Section 6.1 gives an overview of the proposed work, summarizing the methodology and the obtained results. Finally, Section 6.2 concludes the thesis by analyzing the problems of the proposed approach, proposing solutions to overcome such limitations and providing suggestions on future directions for this work.

5.1 Proposed work summary

Convolutional Neural Networks are playing an important role as a new emerging technology for computer vision. Thanks to their capability, they have become one of the most analyzed approaches in the last years for many application fields, including Big Data Analysis, mobile robot vision, video processing and so on. In such contexts, due to the huge amount of data to be processed or the need for real time execution, it is crucial to find techniques to speed up the computation. Moreover, the huge number of operations of CNNs makes it impractical to implement them on CPUs, so researchers and engineers have been looking for hardware accelerators able to provide fast implementation meeting the constraints on power consumption and accuracy. Among the accelerators proposed in the literature, ASIC processor demonstrated to be very effective low-power devices for hardware acceleration of CNNs.
The aim of the work presented in this thesis is to develop an hardware accelerator able to merge the demands in terms of speed and power through a careful analysis of the possible parallelization inside the CNN algorithm.
In particular in Chapter 3 is presented the proposed architecture with all the pre-synthesis simulations necessary to understand the behaviour of the algorithm. From this analysis it can be pointed out how changing the DFG of a particular hardware architecture is possible to exploit properties that will have a huge impact on the overall performances.
In Chapter 4 the synthesis flow is carried on with a particular focus on design compiler aspects that could be useful also for future works on the subject. The main result is that it was possible to complete the processing of an image in a very reasonable time compared to the possibilities in terms of FPS of modern cameras. In fact as we demonstrated our machine can work up to 45 thousand FPS; of course this result does not have to be considered in its absolute value, since in our case we analyzed just one layer of a CNN, but even if we suppose that a full structure will decrease of 100x the speed performances, we will still have more than 450 FPS to deal with. This value is above modern video-cameras that works usually at 120 FPS. THis result could lead to future implementation of structures able to meet constraints in real-time applications.

5.2 Future Works

Although efficient, the proposed architecture still has some limitations due to different reasons. On the one hand, at the moment the input features map that the machine is able to process are defined as 128x128 pixel images; this problem is not impossible to solve since the easiest solution could be to increase the size of the on-chip memories and maybe also try to use bigger filter kernel. But of course this could lead to a latency delay that has to be fully analyzed in order to meet the previous defined real-time constraints. On the other hand, the generated accelerator is able to process a fully convolutional operation, basically implementing the main function of a layer inside a Neural Network. But as we discovered using as example the AlexNet Network, modern CNN have more than 10 different layers, so a future optimization could be to test this architecture inside a particular framework in order to replace a single layer and analyze the overall performances.

Bibliografia

[1] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in neural information processing systems*, 2012, pp. 1097–1105.

[2] M. D. Zeiler and R. Fergus, "Visualizing and understanding convolutional networks," in *European conference on computer vision*. Springer, 2014, pp. 818–833.

[3] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 770–778.

[4] Y.-H. Chen, T. Krishna, J. S. Emer, and V. Sze, "Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks," *IEEE Journal of Solid-State Circuits*, vol. 52, no. 1, pp. 127–138, 2017.

[5] J. Jo, S. Kim, and I.-C. Park, "Energy-efficient convolution architecture based on rescheduled dataflow," *IEEE Transactions on Circuits and Systems I: Regular Papers*, no. 99, pp. 1–12, 2018.

[6] S. Zhang, Z. Du, L. Zhang, H. Lan, S. Liu, L. Li, Q. Guo, T. Chen, and Y. Chen, "Cambricon-x: An accelerator for sparse neural networks," in *The 49th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Press, 2016, p. 20.

[7] S. Han, X. Liu, H. Mao, J. Pu, A. Pedram, M. A. Horowitz, and W. J. Dally, "Eie: efficient inference engine on compressed deep neural network," in *Computer Architecture (ISCA), 2016 ACM/IEEE 43rd Annual International Symposium on*. IEEE, 2016, pp. 243–254.

[8] A. Parashar, M. Rhu, A. Mukkara, A. Puglielli, R. Venkatesan, B. Khailany, J. Emer, S. W. Keckler, and W. J. Dally, "Scnn: An accelerator for compressed-sparse convolutional neural networks," in *ACM SIGARCH Computer Architecture News*, vol. 45, no. 2. ACM, 2017, pp. 27–40.

[9] Y.-J. Lin and T. S. Chang, "Data and hardware efficient design for convolutional neural network," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 65, no. 5, pp. 1642–1651, 2018.

[10] F. Visin, K. Kastner, K. Cho, M. Matteucci, A. Courville, and Y. Bengio, "Renet: A recurrent neural network based alternative to convolutional networks," *arXiv preprint arXiv:1505.00393*, 2015.

[11] K. Kiningham, M. Graczyk, A. Ramkumar, and S. Stanford, "Design and analysis of a hardware cnn accelerator," *small*, vol. 27, p. 6.

[12] S. K. Kumaraswamy, P. Sastry, and K. Ramakrishnan, "Bank of weight filters for deep cnns," in *Asian Conference on Machine Learning*, 2016, pp. 334–349.

[13] G. Desoli, N. Chawla, T. Boesch, S.-p. Singh, E. Guidetti, F. De Ambroggi, T. Majo, P. Zambotti, M. Ayodhyawasi, H. Singh *et al.*, "14.1 a 2.9 tops/w deep convolutional neural network soc in fd-soi 28nm for intelligent embedded systems," in *Solid-State Circuits Conference (ISSCC), 2017 IEEE International*. IEEE, 2017, pp. 238–239.

[14] K. Ovtcharov, O. Ruwase, J.-Y. Kim, J. Fowers, K. Strauss, and E. S. Chung, "Accelerating deep convolutional neural networks using specialized hardware," *Microsoft Research Whitepaper*, vol. 2, no. 11, 2015.

[15] J.-H. Luo, J. Wu, and W. Lin, "Thinet: A filter level pruning method for deep neural network compression," *arXiv preprint arXiv:1707.06342*, 2017.