POLITECNICO DI TORINO

DIPARTIMENTO DI ELETTRONICA E TELECOMUNICAZIONI (DET)

Master's Degree in Electronic Engineering

Master's Degree Thesis

Development of hardware accelerators on FPGA for convolutional neural networks

Supervisor
Prof. Mario Roberto Casu

Candidate
Giulia Altamura

March, 2018

Summary

In recent years the field of *Deep Learning* has been rediscovered and deepened, in particular thanks to the development of *neural networks*, which are particular algorithms able to decide autonomously, based on certain inputs given by the system. The potentiality is given mainly by the possible application fields, which can range from speech recognition to autonomous driving of vehicles, from recognition of photos and videos to much more, such as artificial vision. However it should be specified that, although today the application of *Deep Learning* takes place in many sectors, its potential is still very wide.
One of the several kinds of neural networks are the *convolutional networks* (*ConvNet*), so called because of the particular application of convolutional levels which basically perform matrix-matrix multiplications. Given any input image, a *ConvNet* has the faculty and the purpose of providing the classification output to which the image belongs among the categories of classes available.
Lately the *ImageNet Large Scale Visual Recognition Challenge* (ILSVRC) has allowed the advent of new models of networks (such as *AlexNet*, *GoogleNet* and others) increasingly "deep" and therefore with an increasingly high computational load. This large amount of operations is usually performed on CPU or GPU, but now we try to run them on other devices, such as FPGA/ASIC, since the recent applications require more and more speed of execution and reduced power consumption.

In this thesis work an FPGA-based hardware accelerator for ConvNet is designed and implemented using a *Xilinx Virtex 7* device mounted on a quad-board *ProFPGA* system.
The verification tests that have been performed, have been made on the *AlexNet* model, even if what has been produced is applicable to every type of network. Starting from the *Caffe* framework, chosen for processing the desired network, the accelerator is designed to replace a particular library function called during the execution (*cblas_sgemm*) to perform the operation $C = A * B * alpha + C * beta$ on hardware, reducing the computational load at CPU level. Some improvements were then applied to increase accelerator efficiency.
During the implementation we realized that the communication system available was too slow to think about the whole of what we were planning, so we focused on

optimizing only the accelerator applying improvements time to time. First of all, a basic architecture was implemented that carried out the desired operation. At this point, memories and a state machine were first introduced to manage the calculation and then the architecture was parallelized to try to exploit the capacity of the device on which it was working and improving the performance of the architecture itself. In the last chapter it will be possible to verify the correctness of the work done and the collected data related to the various versions of the accelerator created.

Acknowledgements
Ringraziamenti

English Version

Special thanks goes first to *my parents*, who have always supported the importance of studying and have always been tempted me to try to do my best. Thanks to *my supervisor* for having followed me constantly and patiently during the production of this work. Thanks to those people, colleagues and friends outside of the Politecnico, who have accompanied me during this journey. It is a problem to list them all but I would like to thank especially *Marco Bassignana*, *Deborah Calabrese*, *Daniele Busacchio* and *Giovanni Fazio* for giving that support and that extra boost when I really needed it!

Fairness and satisfaction of the achieved goal I look at the new challenges I will have to face!

Versione italiana

Un ringraziamento speciale va prima di tutto ai miei genitori, che hanno sempre sostenuto l'importanza di studiare e mi hanno sempre spronata a cercare di fare del mio meglio. Grazie al mio relatore per avermi seguita costantemente e con pazienza durante la produzione di questo lavoro. Grazie a quelle persone, colleghi e amici al di fuori del Politecnico, che mi hanno accompagnata durante questo percorso. È un problema elencarli tutti ma in particolare ci tengo a ringraziare soprattutto Marco Bassignana, Deborah Calabrese, Daniele Busacchio e Giovanni Fazio per avermi dato quel sostegno e quella spinta in più quando ne avevo davvero bisogno!

Fiera e soddisfatta del traguardo raggiunto volgo lo sguardo alle nuove sfide che ci saranno da affrontare!

Contents

Summary I

Acknowledgements - *Ringraziamenti* III

List of Figures VI

List of Tables VIII

List of Acronyms IX

1 Introduction 1
- 1.1 An introduction to Deep Learning . . . 1
- 1.2 Purpose of this thesis . . . 3
- 1.3 Thesis's outlines . . . 4

2 Convolutional Neural Network 5
- 2.1 Overview and Architecture . . . 5
  - 2.1.1 Convolution . . . 6
  - 2.1.2 Non Linearity - ReLU . . . 9
  - 2.1.3 Pooling . . . 9
  - 2.1.4 Fully connected layer . . . 10

3 Caffe framework 11
- 3.1 Anatomy of a Caffe model . . . 12
  - 3.1.1 BLOBs, Layers, and Nets . . . 12
  - 3.1.2 Inside Caffe: understanding GEMM . . . 13
- 3.2 Caffe installation . . . 15
- 3.3 Inference on Caffe . . . 16

4 ProFPGA system 19
- 4.1 Devices . . . 20
  - 4.1.1 Motherboard . . . 20
  - 4.1.2 FPGA module: Xilinx Virtex XC7V2000T . . . 22

4.2 ProFPGA Builder . . . 22
4.2.1 The configuration file . . . 24
4.3 I/O: the Module Message Interface 64 . . . 27
4.3.1 The C program side . . . 28
4.3.2 The HDL side . . . 29
4.4 ILA Debug . . . 33
4.4.1 Creating Vivado Debug Project . . . 33
4.5 Working flow . . . 37
4.5.1 A simple test project . . . 37
4.6 Encountered problems . . . 39
4.6.1 PCIe connection . . . 39
4.6.2 Using an ILA core . . . 39

5 Hardware Accelerator Development 41
5.1 Operations flow description . . . 43
5.2 Used tool . . . 43
5.2.1 Designing with IP cores . . . 43
5.2.2 Implementation settings chosen . . . 45
5.3 Implemented design . . . 46
5.3.1 First basic design . . . 46
5.3.2 Simple Dual Port Memory introduction . . . 47
5.3.3 Architecture's parallelization . . . 49
5.3.4 Two MAC units . . . 50
5.3.5 Increasing Parallelization . . . 51

6 Results and Conclusions 52
6.1 Results's verification . . . 53
6.2 First basic design . . . 54
6.3 Simple Dual Port Memory introduction . . . 55
6.4 Architecture's parallelization . . . 57
6.5 Conclusions and future works . . . 60

Bibliography 61

List of Figures

1.1 Humans and AI [1] . . . 1
1.2 AI world [2] . . . 2
1.3 Mammal brain and Convolutional process [3] . . . 2
1.4 Whole system flow . . . 3

2.1 Standard Convolutional Neural Network [4] . . . 5
2.2 Convolution Operation [5] . . . 6
2.3 3D view of a convolution operation [6] . . . 7
2.4 Convolution Demo [5] . . . 8
2.5 Non linear functions . . . 9
2.6 Possible Pooling Operation [7] . . . 10
2.7 A three-layer fully connected [8] . . . 10

3.1 Data in Caffe: every layer takes input through bottom connections and makes output through top connections . . . 12
3.2 Diagram from Yangqing Jia's thesis: it's an analysis of the time spent for convolution using Alex Krizhevsky's deep learning architecture (*AlexNet*). You can see that 89% of the time is spent in *fc* e *conv* layers that are implemented using the GEMM library [9]. . . . 13
3.3 GEneral Matrix to Matrix Multiplication [9] . . . 14
3.4 Correct Caffe installation . . . 15
3.5 Networks from Model ZOO . . . 16
3.6 Input picture: cat.jpg [10] . . . 16
3.7 Inference . . . 17

4.1 Photo of the whole system [11] . . . 19
4.2 Profpga devices [12], [13] . . . 20
4.3 Board coordinate system [11] . . . 21
4.4 Configurable logic block screenshot . . . 22
4.5 ProFPGA builder graphical interface . . . 23
4.6 MMI 64 communication . . . 27
4.7 RTL *user_mmi64.vhd* . . . 32

4.8 Xilinx Programmer for ILA debugging . . . 33
4.9 Project confirm window . . . 34
4.10 Vivado GUI . . . 36
4.11 Work Directory . . . 37
4.12 Register File simulation . . . 37
4.13 Possible error message . . . 40

5.1 Operations flow . . . 43
5.2 IP setting . . . 44
5.3 Implementation settings . . . 45
5.4 First designed blocks . . . 47
5.5 *Simple Dual Port Memory scheme* . . . 47
5.6 ASM control chart . . . 48
5.7 Architecture with memory introduction . . . 49
5.8 Modified Simple Dual Port Memory scheme . . . 50
5.9 Parallelized architecture . . . 50

6.1 Comparison between *Caffe* and FPGA computation . . . 53
6.2 Occupied Area First Design . . . 54
6.3 Power consumption first design . . . 55
6.4 Occupied Area Second Design . . . 55
6.5 Power consumption second design . . . 56
6.6 Occupied Area TWO and FOUR MAC UNITS . . . 57
6.7 Occupied Area EIGHT and SIXTEEN MAC UNITS . . . 58
6.8 Power consumption Parallelized Designs . . . 59

List of Tables

3.1 Comparison of popular deep learning frameworks 11

4.1 FPGA Cell Content 22

5.1 Analysis of AlexNet Layers 42

6.1 Report Cell Usage First Design 54
6.2 Report Cell Usage Second Design 56
6.3 Report Cell Usage Parallelized Designs 58
6.4 Evaluation of the number of execution cycles 59

List of Acronyms

AI	Artificial Intelligence
BLAS	Basic Linear Algebra Subprograms
CNN	Convolutional Neural Network
DL	Deep Learning
FPGA	Field Programmable Gate Array
FSM	Finite State Machine
GEMM	GEneral Matrix to Matrix Multiplication
ILA	Integrated Logic Analyzer
ML	Machine Learning
MMI64	Module Message Interface 64
PCIe	Peripheral Component Interconnect express
ReLU	Rectified Linear Unit
TCP	Transmission Control Protocol

Chapter 1

Introduction

1.1 An introduction to Deep Learning

In recent years we have had a great explosion of data in multiple fields. In order to cope with the elaboration of these "big data" an answer was found in *Artificial Intelligence* (AI). We can define as AI a software or hardware application that thinks and solves problems as a human would do. Problems ranging from interpreting the language to distinguishing what is inside an image to understand and to discern different faces and people. What we have achieved so far is a narrow AI, that uses specific algorithms and techniques to solve some specific problems.

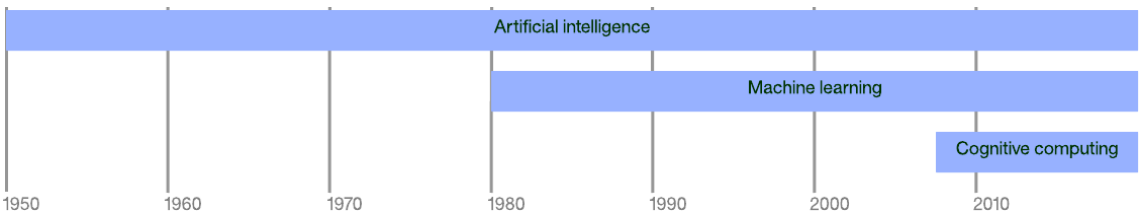


Figure 1.1: Humans and AI [1]

Inside the AI field, we find *Machine Learning* (ML), which consists of using a large data set and a number of classification algorithms to transform the normal way we are used to programming. In our normal way of programming, we create more and more complex algorithms, but we know exactly every single step of them. The underlying idea of creating classificators is to recover large amounts of data and simply create functions to understand which of these data we want and so to improve the hand-handed results and get a system that without writing all the algorithm, is able to make decisions based on the data it has available. Some of these classificators follow mathematical formulas that we know, such as straight lines, polynomial functions, statistical functions, and so on. Some of these are very

good at predicting some sort of behaviour too complex to write in an algorithm such as predicting the price of a house based on a historical series. Keeping on, as a branch of ML we find a particular technique of data learning, known as *Deep learning* (DL).

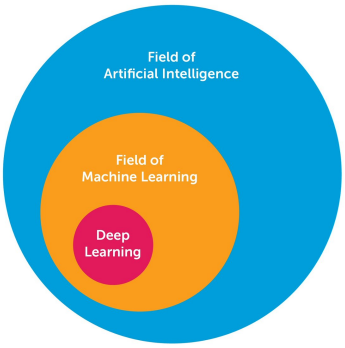


Figure 1.2: AI world [2]

The development of DL has taken place in parallel with the study in particular of *neural networks*. It is characterized by the effort to create a multi-level automatic learning model, in which the deepest levels take as inputs the outputs of the previous levels, transforming them and becoming more and more astonished. This intuition on learning levels gives the name to the whole field and it is inspired by how the mammal brain processes information and learns, responding to external stimuli.

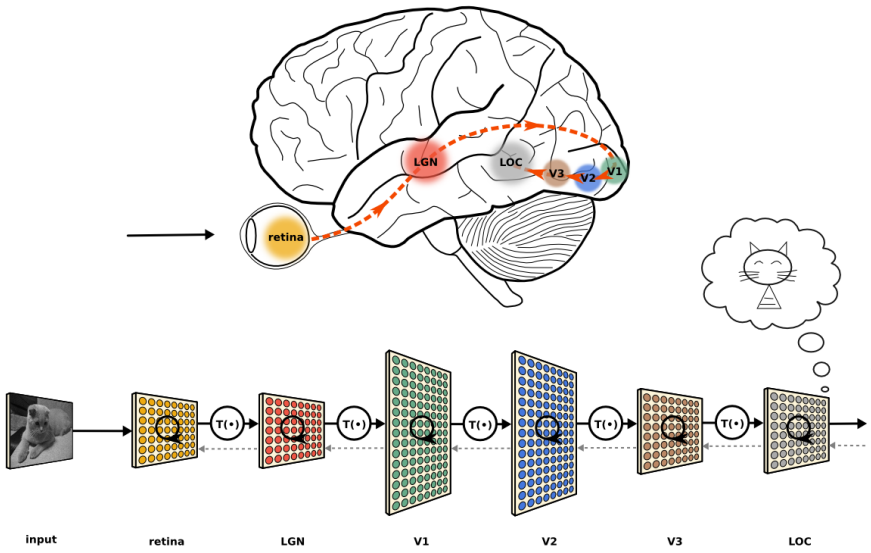


Figure 1.3: Mammal brain and Convolutional process [3]

Each level of learning corresponds to one of the different areas that make up the cerebral cortex. For example, the visual bark, which is responsible for image recognition, shows a sequence of sectors placed in hierarchy. Each of these sectors receives an incoming representation by means of flow signals that link it to other sectors. Each level in this hierarchy represents a different level of abstraction, with the more abstract features defined in terms of those of the lower level. When the brain receives images, it processes the perception of shapes (from the primitive ones to the increasingly complex ones) through various phases, such as edge detection. This is the case for hierarchical representation of the image in the growing abstraction level. Just as the brain learns by trying and activating new neurons by learning the experience, even in DL architectures, the extraction stages change based on input information received.

1.2 Purpose of this thesis

By giving an image as input to a Convolutional Neural Network (*ConvNet*), this will produce, as output, a class which identifies that image.
The aim of this thesis is to create a hardware accelerator which performs the convolution operation, the most critical operation in a ConvNet.
To give an idea of the computational load of a ConvNet, in the AlexNet model, for example, 90% of the processing time is occupied by convolution operations [14].
Furthermore the complexity of these network is strictly connected to their depth and one of the main problems is that this kind of operation requires a lot of memory.
We will try to implement an architecture that performs this operation in an optimized way, and to apply techniques to minimize the number of accesses to the external DRAM connected to the FPGA during convolution calculation. The dual purpose is to exploit as much as possible on-chip memory to reduce latency in accessing data and reduce the energy cost of accessing external memory.

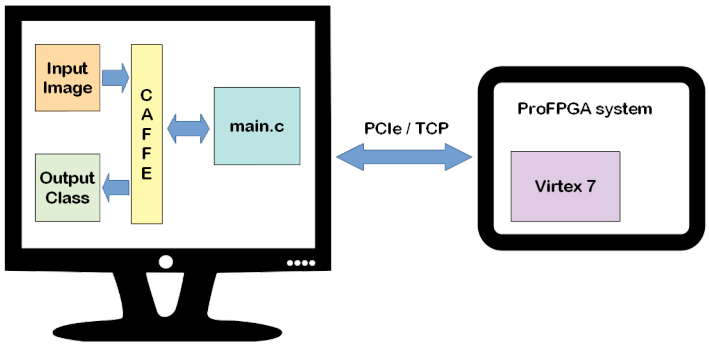


Figure 1.4: Whole system flow

The idea, as shown in figure 1.4, is to stop the execution of the *Caffe* framework by extracting some image data not yet processed and then give these ones as inputs to the FPGA that will perform the convolution operation and will return the processed data. At this point *Caffe* will read back the processed data and will finish the algorithm to give as result the image classification.

1.3 Thesis's outlines

In order to better visualize the content of this thesis, below are reported the topics that will be analysed in the various chapters.

In Chapter 2 will be provided an overview of convolutional networks which will explain its main features.

In Chapter 3 will be provided informations related to *Caffe*, the framework used. In particular we will see why it was chosen for this work, what are its characteristics, in particular how the *GEMM* library works, we will also see how to install it and to conclude will be provided a brief tutorial on how to use it for image classification/inference using Python as programming language.

In Chapter 4 we will see more in details the *ProFPGA* system, with all its hardware and software components, used to implement the accelerator. We will analyse the characteristics, the advantages offered, and the solutions found to the difficulties of usage. In the last section, a sort of guide will be provided to explain in a simple way how to use the system.

In Chapter 5 we will discuss the implementation of the accelerator's architecture and the various changes made during development to optimize it.

In Chapter 6 considerations will be given on the work done and suggestions on how to deepen it in future works.

Chapter 2

Convolutional Neural Network

2.1 Overview and Architecture

A *CNN* or *ConvNet* (Convolutional Neural Network) is so defined for the presence of at least one or more convolution layers then followed by fully connected layers as in a standard multilayer neural network. They are designed for input with a 2D structure as an image or a speech signal. These networks are easy to train and respect to a fully connected layers have fewer parameters with the same number of hidden units. Let's see the structure of a typical *ConvNet*:

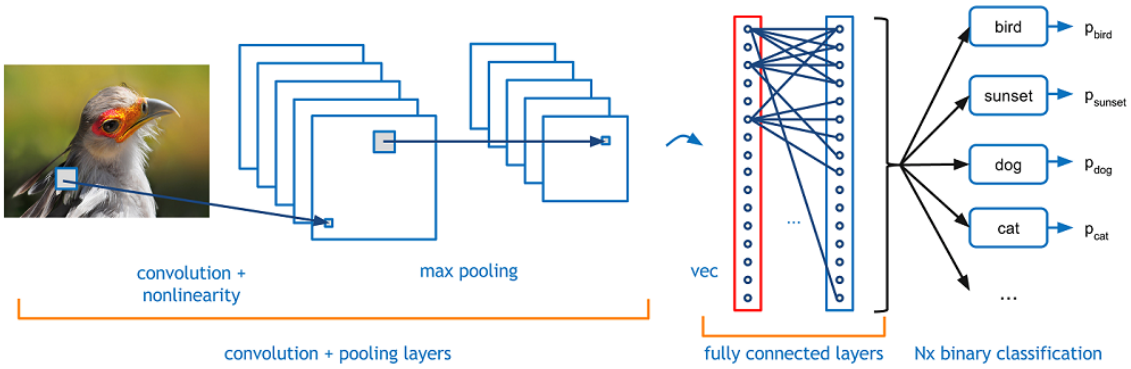


Figure 2.1: Standard Convolutional Neural Network [4]

We can highlight four main operations in a *ConvNet* as shown in the figure above:

1. Convolution
2. Non Linearity (ReLU)
3. Pooling or Sub Sampling
4. Classification (Fully Connected Layer)

These operations are the basic building blocks of every Convolutional Neural Network.

2.1.1 Convolution

Convolution is the operation to which we will pay more attention since it is the one we want to speed up on FPGA. Its purpose in *ConvNet* is to extract features from the input image.
An image can be seen as a matrix of pixel values. One from a standard digital camera will have three channel (RGB which stands for Red, Green and Blue) or just one if it is a grey-scale image.
Assuming to have, for example, a 7x7 image whose pixel values are only 0 and 1 (note that it would be a special case) and also another 3x3 matrix, the convolution can be computed as shown in the figure below:

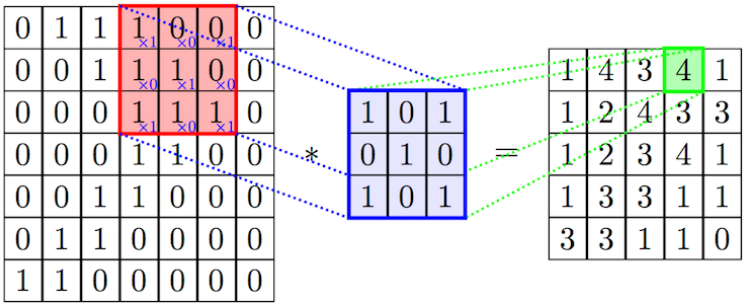


Figure 2.2: Convolution Operation [5]

The 3x3 matrix, called *filter* or *kernel* or *neuron*, is multiplied by a portion of the input matrix, until the values of all the elements of the output matrix, called *feature map*, are computed.
Obviously you can have different filters so perform different operations such as *edge detection*, *sharpen* and *blur* just by changing the numeric values (called *weights*) of the filter matrix before the convolution operation.
It is important to say that when dealing with large inputs such as images, it is not practical to connect neurons to all the neurons of the previous volume. Instead, we will link each neuron only to a local region of the input volume. Looking to the figure 2.3 you can see how each neuron in the convolutional layer (in blue) is connected only to a local region in the input volume (in red) spatially, but to the full depth (i.e. all colour channels). In this example you can note that there are 5 neurons along the depth, all looking at the same region in the input.

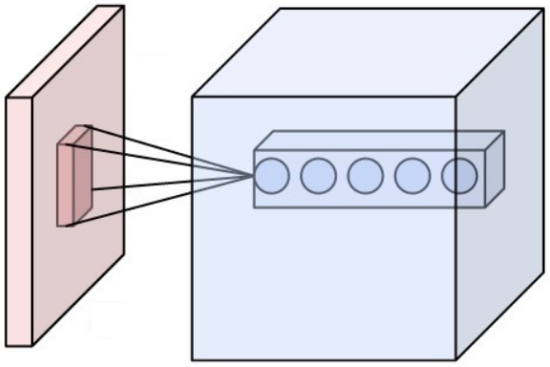


Figure 2.3: 3D view of a convolution operation [6]

But how many neurons are in the output volume or how are they arranged? Four hyperparameters come into play when deciding the size of the output volume:

1. The *depth*. It corresponds to the number of filters we would like to use, each learning to look for something different in the input as mentioned before.
2. The *receptive field size* of the Conv Layer neurons.We are talking about the dimension of the filters.
3. The *stride*. When we slide the filter along the input we have to decide of how many pixel per time we want do it.
4. The *zero-padding*. Sometimes it will be convenient to pad the input volume with zeros around the border, usually a border of one pixel size is used.

Summarizing we can say that a Convolutional Layer:

- Accepts a volume of size $W_1 * H_1 * D_1$
- Requires four hyperparameters:
  - number of filters **K**
  - their spatial extent **F**
  - the stride **S**
  - the amount of zero-padding **P**
- Produces a volume of size $W_2 * H_2 * D_2$ where:
  - $W_2 = (W_1 - F + 2P)/S + 1$
  - $H_2 = (H_1 - F + 2P)/S + 1$ (i.e. width and height are computed equally by symmetry)

– $D_2 = K$

To better visualize how work a 3D convolution operation we report the following example:

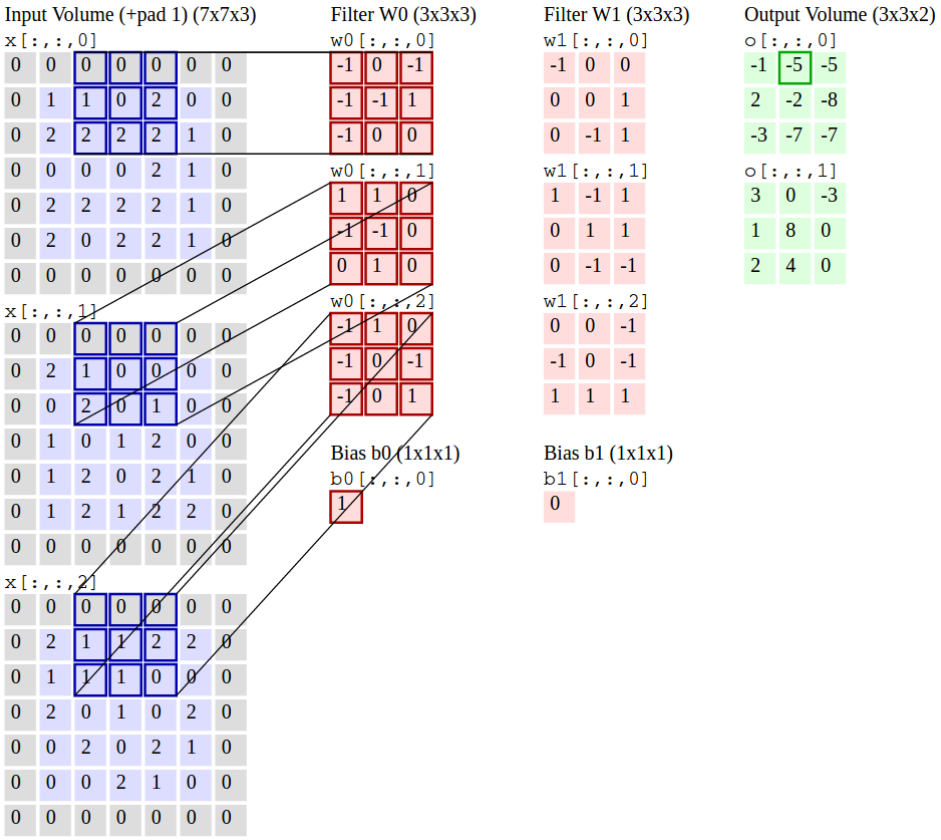


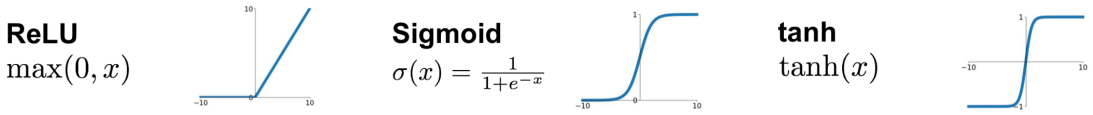
Figure 2.4: Convolution Demo [5]

The input volume (in blue) is of size W1=5, H1=5, D1=3, and the CONV layer parameters are K=2, F=3, S=2, P=1. This means that we have two filters of size 3×3 (in red), and they slide, or convolve, around the input of two pixel per time. Therefore, we can already compute the size of the output volume (in green) by applying the formula seen before obtaining a spatial size of (5 - 3 + 2)/2 + 1 = 3. Moreover, notice that a padding of P=1 is applied to the input volume, making the outer border of the input volume zero.

Anyway, we will see in chapter 3.1.2 the software approach to the convolution operation performed by the *cblas_sgemm* function used inside *Caffe* framework.

2.1.2 Non Linearity - ReLU

Usually, immediately after a convolutional layer, we find a non-linear unit, whose purpose is to introduce a non-linearity precisely, since most of the real-world data we would like our ConvNet to learn is non-linear, instead the convolution is a linear operation (element wise matrix multiplication and addition). The most popular non-linear function is the *Rectified Linear Unit* (*ReLU*), but it is not the only one. In the past, other functions were used such as *tanh* and *sigmoid*, but researchers found that the *ReLU* layers work much better since it is possible to train the network faster (thanks to computational efficiency) without there being a significant difference in accuracy.



(a) ReLU function (b) Sigmoid function (c) Tanh function

Figure 2.5: Non linear functions

As you can see from the figure above, the *ReLU* layer applies the function $f(x) = max(0, x)$ to all of the values in the input volume. Basically, this layer just replaces all negative pixel values in the feature map by 0.

2.1.3 Pooling

It is common to periodically insert a *pooling* level between the various layers of a ConvNet. This is done in order to progressively reduce the spatial dimension of the representation, the quantity of parameters and computation in the network, and therefore also to control over-fitting. Specifically, the pooling layer operates on each input depth slice and resizes it by generally applying the *MAX operation* (but this is not the only applicable function as you can see from the Figure 2.6).

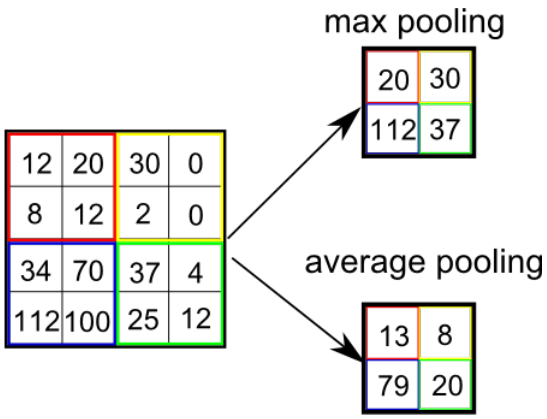


Figure 2.6: Possible Pooling Operation [7]

2.1.4 Fully connected layer

For regular neural networks and at the end of a ConvNet, the most common layer type we find is the *fully-connected* layer, in which neurons between two adjacent layers are simply fully connected between each other as shown in the figure below.

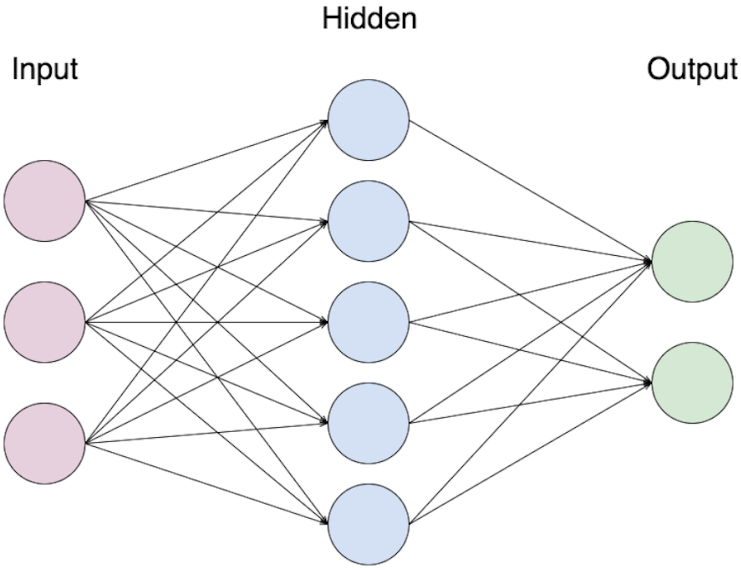


Figure 2.7: A three-layer fully connected [8]

Chapter 3

Caffe framework

Many frameworks have been developed to work in deep learning field. The table 3.1 provides an overview of some of the most popular frameworks used.
Caffe has been chosen among the available framework due to various reasons.
First of all it is a fully open source framework, maintained and developed by the *Berkeley Vision and Learning Center* (*BVLC*). In general, it provides all the necessary tools and good documentation on all the possible approaches to a neural network, so: training, testing and fine-tuning.
Even if, at the beginning, it was designed for design vision, then users also used and developed it for speech recognition, robotics, neuroscience and astronomy.
The implementation is completely C++ based, which eases integration into existing C++ systems and interfaces common in industry. A very useful fact also is that it offers pre-trained models for free research on *Model ZOO* [15] that you can easily use/download just going at the following link:
`http://caffe.berkeleyvision.org/model_zoo.html`.

Table 3.1: Comparison of popular deep learning frameworks

Framework	Core Language	Binding(s)	CPU	GPU	Pre-trained models
Caffe	C++	Python, MATLAB	√	√	A lot from *Model ZOO*
Tensor Flow	Python	-	√	√	Only *Inception*
Theano	Python	-	√	√	*Lasagne*
Torch	Lua	-	√	√	-

3.1 Anatomy of a Caffe model

3.1.1 BLOBs, Layers, and Nets

Caffe stores, communicates, and manipulates the information as **BLOBs** which represent the standard array and unified memory interface for the framework. A BLOB is generally used as a four dimensional ordered data structure: number, channels, height, and width. Furthermore the channels, height, and width usually describe a piece of data such as an image.

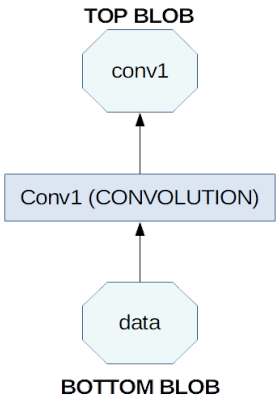


Figure 3.1: Data in Caffe: every layer takes input through bottom connections and makes output through top connections

Each **layer**/level type defines three critical computations: setup, forward, and backward. Levels have two key responsibilities for the functioning of the network as a whole: a step forward that takes input and produces output, and a step backward that takes the gradient with respect to the output and calculates gradients with respect to parameters and inputs, which at they are propagated to previous levels. The development of personalized levels requires the least effort due to the compositional nature of the network and the modularity of the code.
The **net** is a set of linked layers in a calculation chart. A typical network starts with a data layer that loads from the disk and ends with a loss level that calculates the goal for a task such as classification or reconstruction.
Models are defined in the clear protocol buffer scheme (*prototxt*) while the learned models are serialized as binary protocol buffer (*binaryproto*) *.caffemodel* files. The format of the model is defined by the protobuf scheme in *caffe.proto*.
Caffe speaks Google Protocol Buffer for the following strengths: binary strings of minimum size when serialized, efficient serialization, a human-readable text format compatible with the binary version, and efficient implementations of the interface in multiple languages, most notably *C++* and *Python*. All of this contributes to the

flexibility and extensibility of modelling in *Caffe* [16].

3.1.2 Inside Caffe: understanding GEMM

As already said in the previous sections, one of the main properties of *Caffe* is its modularity. Since to make deep learning with neural networks faster and more power efficient we need to focus on a function called *GEMM*, which is part of the *BLAS* (Basic Linear Algebra Subprograms) library. Let's go to analyse the C++ code that deals with the execution of convolution operations.

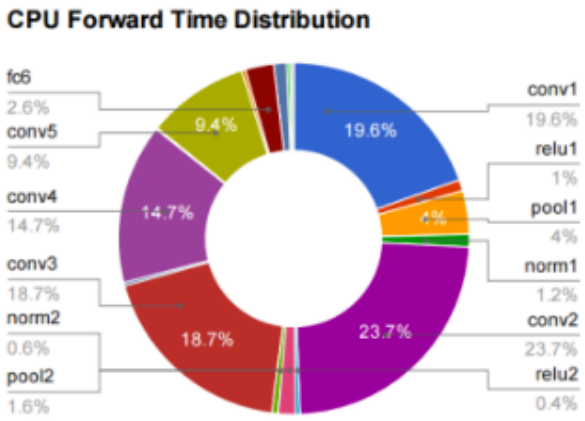


Figure 3.2: Diagram from Yangqing Jia's thesis: it's an analysis of the time spent for convolution using Alex Krizhevsky's deep learning architecture (*AlexNet*). You can see that 89% of the time is spent in *fc* e *conv* layers that are implemented using the GEMM library [9].

Specifically we will see more in detail the *cblas_sgemm* function which is called inside the *caffe_cpu_gemm* fuction from the *math_functions.cpp* file of *Caffe* reported below:

```
template<>
void caffe_cpu_gemm<float> ( const CBLAS_TRANSPOSE TransA,
  const CBLAS_TRANSPOSE TransB, const int M, const int N,
  const int K, const float alpha, const float* A, const float* B,
  const float beta, float* C)
  {
        int lda = (TransA == CblasNoTrans) ? K : M;
        int ldb = (TransB == CblasNoTrans) ? N : K;
        cblas_sgemm(CblasRowMajor, TransA, TransB, M, N, K, alpha,
        A, lda, B, ldb, beta, C, N);
  }
```

So what is *GEMM*? It stands for *GEneral Matrix to Matrix Multiplication*, and it essentially does exactly what it says, multiplies two input matrices together to get an output one [9].
The operation is defined as:

$$C := alpha * op(A) * op(B) + beta * C$$

where:

- $op(A) = A'$ if *TransA* is set, otherwise $op(A) = A$. For $op(B)$ is similar.
- *alpha* and *beta* are scalars.
- A, B and C are matrices: *op(A)* is an m-by-k matrix, *op(B)* is a k-by-n matrix, C is an m-by-n matrix as shown in figure 3.3.

A note about *Caffe* is that matrices are stored in row-major order in CPU but in col-major order in GPU. So *caffe_cpu_gemm* computes $C = A*B$ while *caffe_gpu_gemm* computes $C' = B' * A'$

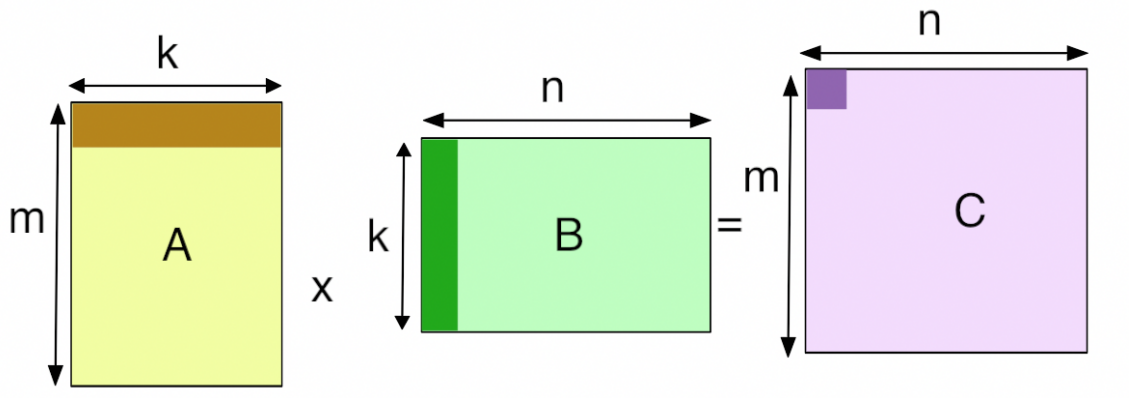


Figure 3.3: GEneral Matrix to Matrix Multiplication [9]

But how can *Caffe* perform a 3D convolution as a 2D matrix multiplication? Due to a previous linearization performed by the *im2col* operation.
And what are the two matrices A and B that are multiplied between them? A is the weight matrix or, in other words, the filter that is applied to the linearized 3D tensor B.

3.2 Caffe installation

It is possible downloading and installing *Caffe-BVLC* from `https://github.com/BVLC/caffe` (Master-version). But other prerequisites and packages are needed to use the framework. Something changes depending on which is your version of Ubuntu, moreover it is possible to train/run *Caffe*: only on GPU, only on CPU or on both.
For the versions of Ubuntu from 16 up, a large amount of documentation exists in case of any problem that can be found. You can found a very good manual for Ubuntu 16.04 at:
`https://github.com/BVLC/caffe/wiki/Ubuntu-16.04-Installation-Guide`.
We have worked with a previous version, Ubuntu 14.04. The installation was a little more difficult since it was necessary to install many non-existent dependencies in the system. In particular, for only CPU mode, it was necessary to *install*:

- General Dependencies:
  build-essential cmake git pkg-config, libprotobuf-dev, libleveldb-dev, libsnappy-dev, libhdf5-serial-dev, protobuf-compiler, libboost-all-dev, libgflags-dev, libgoogle-glog-dev, liblmdb-dev
- BLAS: libatlas-base-dev
- OpenCV: libopencv-dev
- Python: python-pip, python-dev, python-numpy

Once you've installed all the necessary dependencies, it is time to modify the *Makefile.config* file with the correct settings you want to use. In our case, it was enough to uncomment the line "*CPU_ONLY := 1*" which simply allows the use of *Caffe* only on CPU precisely.
At this point you can go to the *compilation* step launching the following commands:

- make all
- make test
- make runtest

At the end of the last command, if everything went as it should have been, you will can read a message saying: `[ PASSED ]`

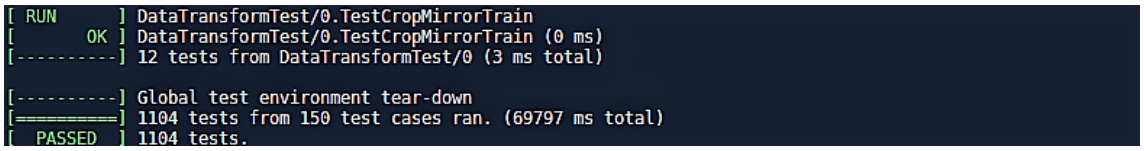
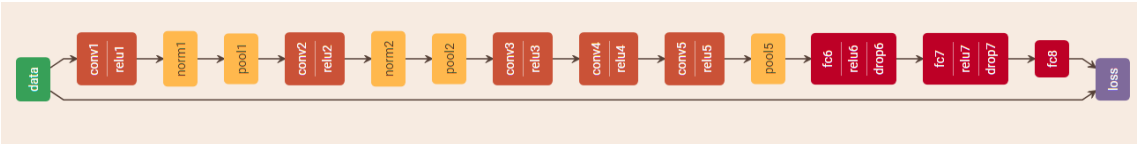


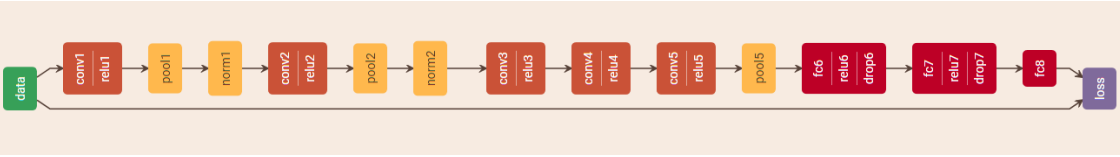
Figure 3.4: Correct Caffe installation

3.3 Inference on Caffe

As already mentioned, *Caffe* allows you to perform different tasks, but what we are interested in is to use it for image classification/inference.
To do this we can bypass the step for creating and training a network, instead we can use a pre-trained network model from the Model ZOO. For example, some of these network can be: *AlexNet* (figure 3.5a) and CaffeNet (figure 3.5b). Both networks have already been trained in *ImageNet* data set which has 1000 class of classification category.



(a) AlexNet diagram [17]



(b) CaffeNet diagram [18]

Figure 3.5: Networks from Model ZOO

From the pictures, you can notice how these two networks are almost equal. The only difference in fact resides at the beginning of the networks in which we find two levels reversed, specifically, *norm1* and *pool1*.

Figure 3.6: Input picture: cat.jpg [10]

Just loading the wanted network through a Python script, giving as input a certain image (figure 3.6), the chosen network will compute the final output as a class that identify the processed image as well as returning other important information (figure 3.7a and 3.7b).

Therefore you can see how the small difference between the two networks previously highlighted, let these come to a result, almost equal, but still different.

```
INFERENCE

Predicted class is:  285
Output label:  n02124075 Egyptian cat

Probabilities and labels:
[(0.32865819, 'n02124075 Egyptian cat'), (0.17557262, 'n02123159 tiger cat'), (0.12056047, 'n02123045 tabby, tabby cat'),
(0.094750918, 'n02119022 red fox, Vulpes vulpes'), (0.042105813, 'n02085620 Chihuahua')]
```

(a) AlexNet inference results

```
INFERENCE

Predicted class is:  281
Output label:  n02123045 tabby, tabby cat

Probabilities and labels:
[(0.3323881, 'n02123045 tabby, tabby cat'), (0.25875551, 'n02123159 tiger cat'), (0.12036911, 'n02124075 Egyptian cat'
), (0.078284144, 'n02127052 lynx, catamount'), (0.067682318, 'n02119022 red fox, Vulpes vulpes')]
```

(b) CaffeNet inference results

Figure 3.7: Inference

```
# loading libraries
import numpy as np
import matplotlib.pyplot as plt
from PIL import Image
plt.rcParams['figure.figsize'] = (10, 10)
plt.rcParams['image.interpolation'] = 'nearest'
plt.rcParams['image.cmap'] = 'gray'
import sys
caffe_root = '/home/galtamura/giulia/caffe/'
sys.path.insert(0, caffe_root + 'python')
import caffe
import os

caffe.set_mode_cpu()

# load the model
# model_def = caffe_root + 'models/bvlc_alexnet/deploy.prototxt'
# model_weights = caffe_root +
# 'models/bvlc_alexnet/bvlc_alexnet.caffemodel'
model_def = caffe_root +
'models/bvlc_reference_caffenet/deploy.prototxt'
model_weights = caffe_root +
'models/bvlc_reference_caffenet/bvlc_reference_caffenet.caffemodel'

net = caffe.Net(model_def, model_weights, caffe.TEST)

# load input and configure preprocessing
mu = np.load(caffe_root +
```

```
    'python/caffe/imagenet/ilsvrc_2012_mean.npy')
mu = mu.mean(1).mean(1)

transformer =
        caffe.io.Transformer({'data':net.blobs['data'].data.shape})
transformer.set_mean('data',mu)
transformer.set_transpose('data',(2,0,1))
transformer.set_raw_scale('data',225)
transformer.set_channel_swap('data',(2,1,0))

net.blobs['data'].reshape(1,3,227,227)
net.blobs['data'].reshape(1,3,227,227)

# load image in the data layer
im = caffe.io.load_image(caffe_root + 'examples/images/cat.jpg')
net.blobs['data'].data[...] = transformer.preprocess('data',im)

# INFERENCE
# compute
out = net.forward()
output_prob = out['prob'][0]

print 'Predicted class is: ', output_prob.argmax()

labels_file = caffe_root + 'data/ilsvrc12/synset_words.txt'
labels = np.loadtxt(labels_file, str, delimiter='\t')
print 'Output label: ', labels[output_prob.argmax()]

top_inds = output_prob.argsort()[::-1][:5]

print '\nProbabilities and labels:'
print zip(output_prob[top_inds],labels[top_inds])
```

Chapter 4

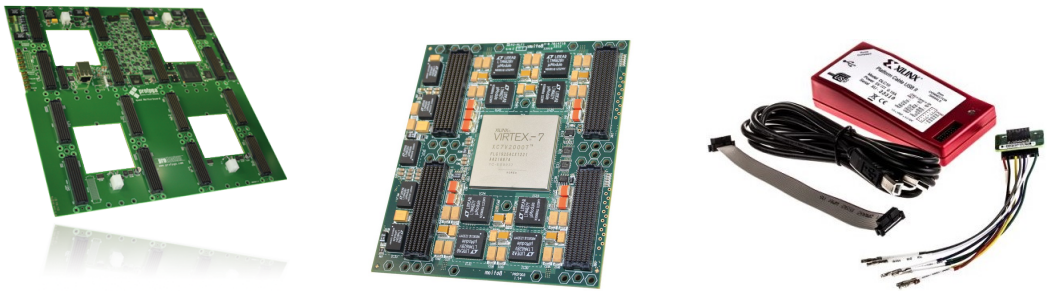
ProFPGA system

The quad V7 proFPGA system can be seen as a complete, scalable and modular multi FPGA prototyping solution. It can consist of several blocks such as a motherboard block, a FPGA module, a cable and so on. Moreover the system provides an extensive set of features and tools like remote system configuration, integrated self and performance test, automatic board detection, automatic I/O voltage programming, system scan and safety mechanism [12].

Figure 4.1: Photo of the whole system [11]

In particular, our proFPGA quad V7 system (figure 4.1) is equipped with a Xilinx Virtex 7 XCV2000T FPGA module (highlighted in red) and a Zynq7000

Z100 module (highlighted in blue), even if in this work thesis we will use only the Virtex one.
It is possible to transfer data, from the workstation to the user design inside the proFPGA and vice-versa, through the Module Message Interface 64 (MMI64). To debug and verify your own architecture is quite hard but possible thanks to the Integrated Logic Analyzer (ILA).
You have also to know that while the FPGA can be controlled remotely by means of simple commands, the system's motherboard can turned on/off only manually using the appropriate switch (highlighted in yellow).
The figure 4.2 shows the main components which make up our system. Let's take a closer look at the characteristics of these devices in the next sections and how they were used in this work.



(a) ProFPGA quad motherboard

(b) proFPGA Module V7 2000

(c) Xilinx programmer device

Figure 4.2: Profpga devices [12], [13]

4.1 Devices

4.1.1 Motherboard

It is possible to say that the motherboard (figure 4.2a) is the main component of our system as it constitutes its infrastructure. As reported in [19] the ProFPGA quad Motherboard offers several features. It provides:

- FPGA Modules: up to 4 proFPGA FPGA modules;
- Clock Management: 8 clock generators;
- Debug Interfaces: JTAG chain for Xilinx and Altera FPGA Modules;
- Host Data Exchange Interfaces: USB, PCIe 4Lane, Ethernet;
- Power Management: up to 1,2 kW;

- Board Detection and Power Protection: automatic daughter board, cable and FPGA modules detection and right power setting;

- and others.

In the proFPGA system we can identify two coordinates but since we use only one motherboard we will be interested to identify only one of the two. We are talking about the board coordinate system that describes the positions of the motherboard connectors allowing the detection of the proFPGA modules.
A single FPGA module can be connected to up to four connectors and a code is assigned to each connector as shown in the figure below.

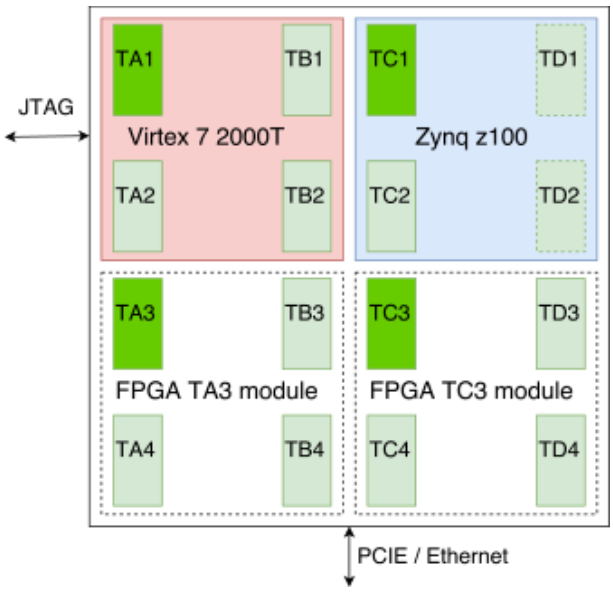


Figure 4.3: Board coordinate system [11]

As you can see, the coordinate code consists of two letters and a number. The first letter will be a **T** or a **B** that identify the top and bottom sides. In the representation there is only the letter **T** since we will use only the top side. The second letter (from **A** to **D**) and the number (from **1** to **4**) indicate respectively the column and row coordinates of the connectors.
The name of a module is identified by the code of its coordinate at the top left (highlighted in dark green in figure 4.3).
As said before we have two FPGA modules available even if we will use only the Virtex module. This one requires four connectors and in our configuration it uses the connectors **TA1**, **TA2**, **TB1** and **TB2**. Since the **TA1** is the top left connector code the Virtex 7 will be named *fpga_module_ta1*.
You can find more information about what was discussed in this section in [20].

4.1.2 FPGA module: Xilinx Virtex XC7V2000T

The logic core of the system is the proFPGA Xilinx Virtex XC7V2000T (shown in figure 4.2b). It is based on the Virtex 7 2000T, offers with its latest FPGA technology maximum capacity of up to 12 M ASIC gates alone in one FPGA [21], the World's Highest Capacity FPGA, according to Xilinx.
In the figure below it is possible to see the content of one of the main configurable logic block.

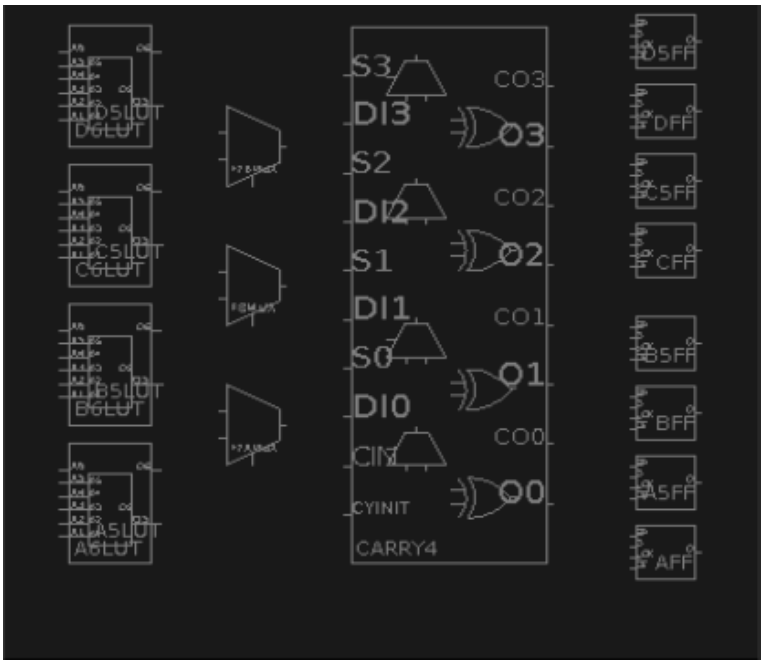


Figure 4.4: Configurable logic block screenshot

Instead in the following table are reported the main available cell:

Table 4.1: FPGA Cell Content

LUT	FF	BRAM	DSP
1221600	2443200	1292	2160

4.2 ProFPGA Builder

ProFPGA Builder is one of the software provided by ProFPGA. We will use it exclusively to generate the *profpga.cfg* file, which is the configuration file that will set up our FPGA and through which the bitstream described in the *user_mmi64.bit*

file will be loaded.
To launch the program, simply open a terminal and write the command:

```
> profpga_builder
```

A graphical interface will open and allow you to interact with the program (figure 4.5).

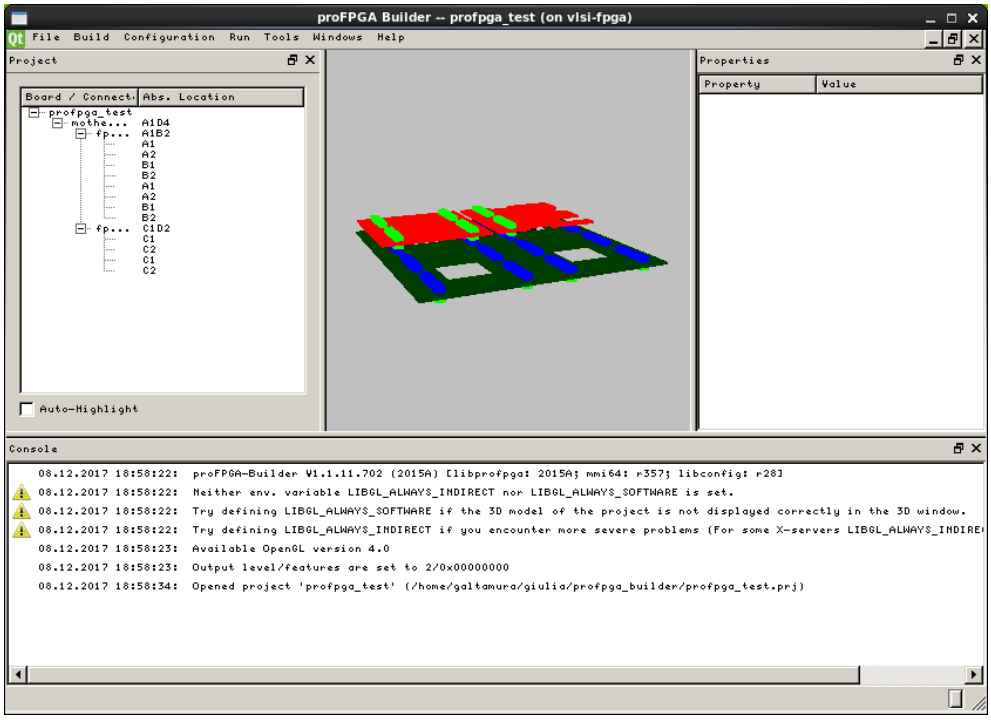


Figure 4.5: ProFPGA builder graphical interface

To use it is demand to create a project, in [22] you can find the description of all the required steps to create a project and hence the configuration file. More or less, the main operations you will need to perform will be:

- Enter location and name of the new project;
- Enter the ip address of the system you want to connect to (for our configuration `http://172.16.0.230`);
- Enter the location of the board description files;
- Specify the FPGA image file for each FPGA in the system using the "*Configuration* $\longrightarrow$ *Image Files*"menu command.

4.2.1 The configuration file

In addition to the FPGA bitstream generated by the *Vivado* tool, the configuration file is the basis for the use of the proFPGA system. Below is reported the configuration file generated for our system:

```
name = "profpga";
profpga_debug = 0;
debug = 0;
backend = "pcie";
backends :
{
   tcp :
   {
      ipaddr = "172.16.0.230";
      port = 0xD11D;
   };
   pcie :
   {
      device = "/dev/mmi64pcie0";
   };
};
system_configuration :
{
   sysconfig_match = "FIT";
   fpga_speedgrade_match = "FIT";
   motherboard_1 :
   {
      type = "MB-4M-R2";
      fpga_module_ta1 :
      {
         type = "FM-XC7V2000T-R2";
         speed_grade = 1;
         bitstream = "user_mmi64.bit";
         v_io_ta1 = "AUTO";
         v_io_ta2 = "AUTO";
         v_io_tb1 = "AUTO";
         v_io_tb2 = "AUTO";
         v_io_ba1 = "AUTO";
         v_io_ba2 = "AUTO";
         v_io_bb1 = "AUTO";
         v_io_bb2 = "AUTO";
```

```
};
fpga_module_tc1 :
{
   type = "FM-XC7Z100-R1";
   speed_grade = 1;
   v_io_ta1 = "AUTO";
   v_io_ta2 = "AUTO";
   v_io_ba1 = "AUTO";
   v_io_ba2 = "AUTO";
   boot_mode = "JTAG";
   usb_mode = "DEVICE";
   usb_id = "UNUSED";
   ps_npor = "SWITCH";
   ps_nsrst = "SWITCH";
   geth_config2 = "GND";
   geth_config3 = "LED1";
};
clock_configuration :
{
   clk_0 :
   {
      source = "LOCAL";
   };
   clk_1 :
   {
   source = "60MHz";
   multiply = 20;
   divide = 24;
   };
   clk_2 :
   {
   source = "125MHz";
   multiply = 8;
   divide = 10;
   };
   clk_3 :
   {
   source = "125MHz";
   multiply = 8;
   divide = 20;
   };
```

```
        clk_4 :
        {
        source = "125MHz";
        multiply = 8;
        divide = 40;
        };
      };
      sync_configuration :
      {
        sync_0 :
        {
          source = "GENERATOR";
        };
        sync_1 :
        {
        source = "GENERATOR";
        };
        sync_2 :
        {
        source = "GENERATOR";
        };
        sync_3 :
        {
        source = "GENERATOR";
        };
        sync_4 :
        {
        source = "GENERATOR";
        };
      };
   };
x_board_list = ( );
};
```

Starting from here, we highlight the settings to pay attention to:

- first of all, this file allows you to select which kind of connection, among Ethernet and PCIe, you would like to use. Watching at the line:

  ```
  backend = "pcie";
  ```

  in case you decide to use the TCP connection you have to simply modify *pcie* with *tcp*;

- second thing, if the physical position of the FPGA module changes, is sufficient to specify the new FPGA module name (figure 4.3).
- Third thing to note is the settings related to the clock. The *clk_0* will be the one that will be assigned to the HDL modules of the *ProFPGA* system and is set at a frequency of 100 MHz.
  The other clocks available in this case, from *clk_1* to *clk_4*, are those that will be supplied at the output then by the *profpga_clocksynch* modules represented in figure 4.7 and which can be set as desired and used by the user for their own designed module, in case a frequency is desired other than 100 MHz supplied to the rest of the system.
  Paying attention to *clk_1* for example, this will provide a system frequency of 50 MHz. It will also be necessary to modify the *constraint* file (located in the appropriate folder in reference to the project structure shown in the figure 4.11) to have a correct analysis according to the desired clock. In our case, it will suffice to specify that *clk_p[1]* must have a period of 20 ns (frequency 50 MHz) instead of 10 ns as for *clk_p[0]*.

4.3 I/O: the Module Message Interface 64

The communication between the PC with the running software and the HDL designs running inside the FPGA module on the proFPGA system is provided by the Module Message Interface 64 (*MMI-64*) [23].

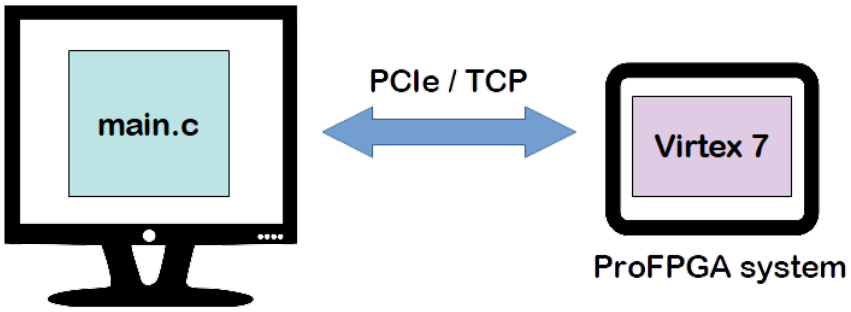


Figure 4.6: MMI 64 communication

From the picture above you can see that we can distinguish two side of the MMI64 communication: the C program and the HDL modules to be built in addition to the user's project.
It is possible to see the HDL modules as the implementation of a register file, whose dimensions are defined according to the functions that can/must be executed in

software by the C program. In particular we can have data of 8, 16, 32 or 64 bit length.
Let's take an example of which features the interface register file should possess. Suppose your project has 5 input signals and 7 output ones, all of 12 bits. Each one of these signals will be connected to a register, so it is necessary that the register file is composed of at least 8 registers ($8 > 7$). It's not necessary to have 16 ($16 > 7+5$) registers because from everyone of them it will be possible both read and write. Instead, as regards the data dimension, the registers must have a data length of at least 16 ($16 > 12$) bits.

4.3.1 The C program side

With reference to the figure 4.11 you can look the file *main.c* in the directory *test*.
As mentioned before, the MMI64 functions must be used according to the data size of the register file, which can be 8, 16, 32 or 64 bit. Specifically, the following functions are available:

To write registers	*To read registers*
mmi64_regif_write_8_ack();	mmi64_regif_read_8();
mmi64_regif_write_16_ack();	mmi64_regif_read_16();
mmi64_regif_write_32_ack();	mmi64_regif_read_32();
mmi64_regif_write_64_ack();	mmi64_regif_read_64();

Each one of this function requires four parameters: the module, the starting address of the register file, how many registers will be written and the pointer to the variable containing the data.

Below is reported an example to clarify the use of these functions:

```
//Array of 8 uint32_t data, corresponding to the instantiated
//register file
uint32_t wdata[8];
//Reading variable
uint32_t rdata32;
//You can write one to all the words of the register file
wdata[0] = 10;
wdata[1] = 5;
wdata[2] = 9;
//Write 3 words (10, 5 and 9), from address 0 till third
//register
status = mmi64_regif_write_32_ack(user_module, 0, 3, wdata32);
CHECK(status);
wdata[0] = 666;
//Write the value 666 in the address 5 (the sixth register)
```

```
status = mmi64_regif_write_32_ack(user_module, 5, 1, wdata32);
CHECK(status);
//Read 1 word at address 3
status = mmi64_regif_read_32(user_module, 3, 1, &rdata32);
CHECK(status);
```

4.3.2 The HDL side

The figure 4.7 represents the architecture that will be deployed on the FPGA. It is possible to distinguish different blocks:

- the *PROFPGA CTRL* is the one that allows the communication between the program C and the user module, so it is the interface between motherboard and Virtex.
- the *PROFPGA CLOCKSYNC* performs the synchronization, so it can be considered as a PLL. It is connected to a clock source from the motherboard and receives as multiplication and division values the ones written in the configuration file. There are four of these components.
- the *MMI64 M REGIF* is the interface with the usere module. At the time of its instantiation it is possible to define the data width and the number of registers that will constitute the *REG FF*.

It seems that the maximum values assignable to REGISTER_COUNT and REGISTER_WIDTH are respectively 16 and 64 ($16 \cdot 64 = 1024$). So, in this case, we can have up to a maximum of 16 I/O signals, each one of 64 bits. Below is the instantiation of the component with the maximum settable of the characteristics we have just discussed:

```
signal reg_addr : std_ulogic_vector(3 downto 0);
signal reg_wdata : std_ulogic_vector(63 downto 0);
signal reg_accept : std_ulogic;
signal reg_rdata : std_ulogic_vector(63 downto 0);

USER_REGIF : mmi64_m_regif
  generic map (
    MODULE_ID => X"FEEDBACC",
        REGISTER_COUNT => 16,
        REGISTER_WIDTH => 64
        )
```

For example, assuming you have a register file as *user module* of figure 4.7 described as follows:

```
signal data_in_s        : std_logic_vector(7 downto 0);
signal data_out_s       : std_logic_vector(7 downto 0);
signal sel_reg_s        : std_logic;
signal register_addr_s  : std_logic_vector(2 downto 0);
signal sel_mux_s        : std_logic_vector(2 downto 0);

component register_file is
  port (  Data_in : in std_logic_vector(7 downto 0);
          clock, reset: in std_logic;
          sel_reg: in std_logic;
          register_addr: in std_logic_vector(2 downto 0);
          sel_mux: in std_logic_vector(2 downto 0);
          Data_out: out std_logic_vector(7 downto 0)
       );
end component register_file;
```

The code of the process which manages the data exchange via the register file (*REG_FF*) could be something like this:

```
-- Accept commands always, you must obey!
reg_accept <= '1';
REG_FF : process(mmi64_clk)
  begin
  if rising_edge(mmi64_clk) then

  -- handle register transfers
    if reg_en='1' and reg_accept='1' then
      if reg_we='1' then -- write to registers
        reg_rvalid <= '0';
        reg_rdata <= (others=>'0');
    case reg_addr is
    when "0000" =>
      sel_reg_s <= std_logic(reg_wdata(14));
      register_addr_s <= std_logic_vector(reg_wdata(13 downto 11));
      sel_mux_s <= std_logic_vector(reg_wdata(10 downto 8));
      data_in_s <= std_logic_vector(reg_wdata(7 downto 0));
    when others =>
      sel_reg_s <= '0';
      register_addr_s <= (others => '0');
      sel_mux_s <= (others => '0');
      data_in_s <= (others => '0');
    end case;
      else -- read from registers
        reg_rvalid <= '1';
    case reg_addr is
    when "0000" =>
      reg_rdata(7 downto 0) <= std_ulogic_vector(data_out_s);
    when others =>
      reg_rdata <= (others => '0');
```

```
    end case;
      end if;
    else -- no transfer or not accepted
      reg_rvalid <= '0';
      reg_rdata <= (others=>'0');
    end if;

    -- reset values
    if mmi64_reset='1' then
      reg_rvalid      <= '0';
      reg_rdata       <= (others=>'0');
      sel_reg_s <= '0';
      register_addr_s <= (others => '0');
      sel_mux_s <= (others => '0');
      data_in_s <= (others=>'0');
    end if;
  end if;
end process REG_FF;
```

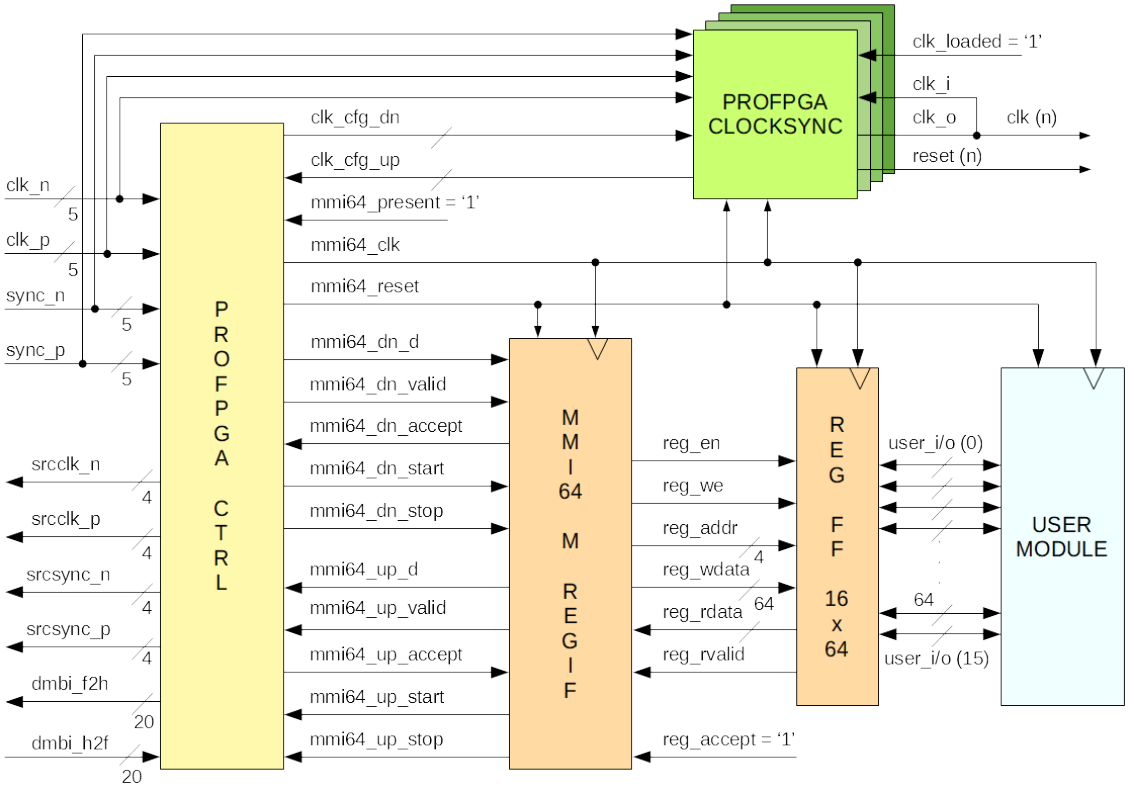


Figure 4.7: RTL *user_mmi64.vhd*

4.4 ILA Debug

As already explained, the *MMI64* interface allows to write and read the i/o signals of our VHDL top entity but despite this, it may be necessary, often indispensable, to check how effectively these signals are exchanged. For this purpose, *Vivado* provides a debugging tool named *Integrated Logic Analyzer* (ILA), a configurable IP core from *Xilinx*.
Below we will see the correct steps to generate and use an ILA core. However to do this is also required a physical device, as shown follows, the Xilinx programmer (Xilinx Platform Cable USB II represented in figure 4.2c). Specifically, this device will be connected on one side to the PC via USB cable, on the other to the proFPGA through a JTAG connection.
You can find further information about ILA-proFPGA debugging in [24], [25] and [26].

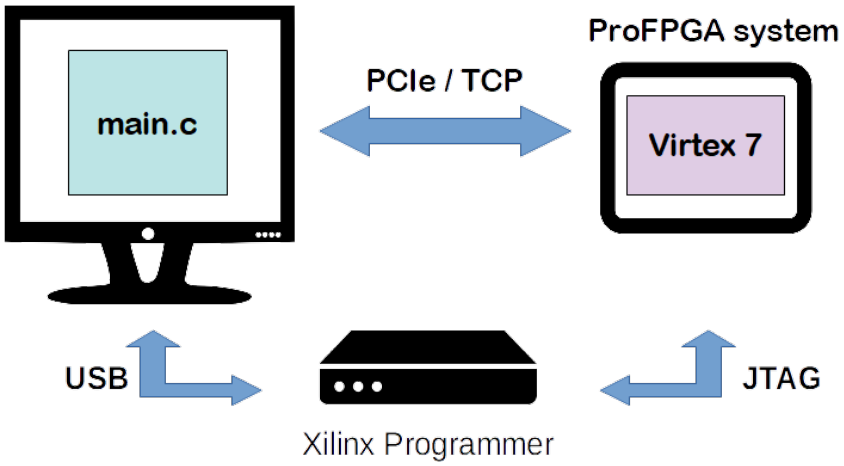


Figure 4.8: Xilinx Programmer for ILA debugging

4.4.1 Creating Vivado Debug Project

Once the above mentioned device is correctly connected and the drivers configured to use it, let's see which steps are required to generate an ILA core to debug our system:

1. Open *Vivado* and click on *"Create New Project"*.
2. Following the instructions, choose a name and the path in which to save the project.
3. Specify "*Post-synthesis Project*" as type.

4. At this point will appear the window *Add Netlist Sources*. Click on *"Add files"* and load, according to the structure in figure 4.11, the file:

   `profpga/vivado/output/user_mmi64_synthesized.dcp`

   Mark the option *"Copy sources into project"*.

5. The next window will be *Add Constraints (optional)*. As before, mark *"Copy sources into project"*, click on *"Add files"* and load:

   `profpga/vivado/constraints/user_mmi64.xdc`

6. Now, in the *"Default Part"* windows, leave everything as it is, ensuring that the selected device is correct (**xc7v2000tflg1925-2**).

If everything has been done correctly, you will see a confirmation window like the one below.

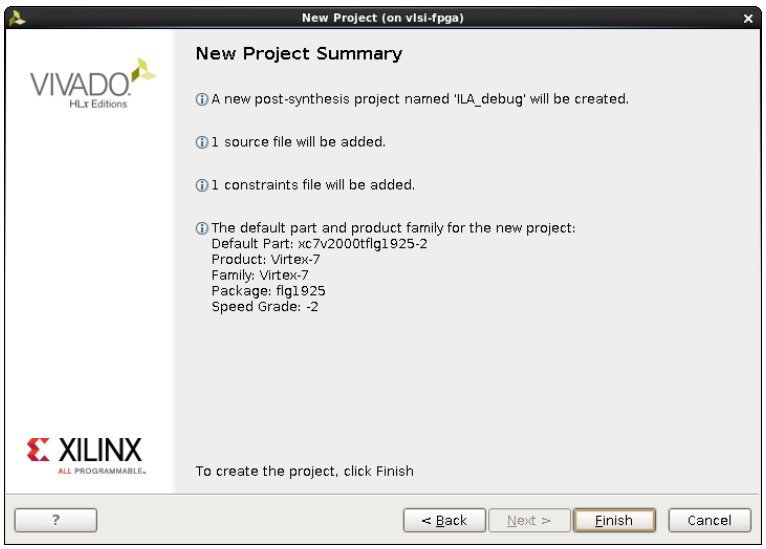


Figure 4.9: Project confirm window

The *Vivado* graphical interface will now open with our uploaded project:

- First we start to open the netlist by simply clicking on the button *"Open Sythesized Design"* highlighted in red in figure 4.10.
- At this point comes the most important phase needed to generate the ILA core, we have to choose the signals which we want to observe. To do that there are several ways, the one that I found most simple and understandable consists in selecting, marking for debugging, the signals directly from the schematic of the architecture.

- Once selected the signals you want to see, you have to click on *"Set Up Debug"*, a *"Synthesized Design"* option under *"Netlist Analysis"*. Be careful, because if you forget some signals you cannot add them as in any simulation, but you will have to repeat the ILA core creation steps.
- Click on *"Next"* in all the pop-ups. Just to know, it is possible to choose the sample of data path (the default is 1024) which is, the content of the signals at a certain moment. Since there is one sample every 3 MHz, it means that after a trigger you can follow the signals for roughly 340 microseconds. Consider that the more signals you choose to monitor, the more time will be required for the implementation, the more resources will be consumed in the FPGA.
- At this point it is possible to generate the bitstream simply by clicking the button highlighted in orange in figure 4.10. This operation will take time depending on the size and complexity of the architecture.

After completing this last step, the program will have generated the file we need in the following path:

`ILA_debug/ILA_debug.runs/impl_1/user_mmi64.bit`

You have to copy this file to the folder `profpga/test`. It is important to load the bitstream through the ProFPGA Builder software mentioned before.
Now, first, turn on the motherboard switching on the flip. At this point, go back to *Vivado* and follow the instructions below in the order presented:

1. Click on the *"Open Hardware Manager"* button highlighted in yellow.
2. On the *Vivado* command line write:

   ```
   > connect_hw_target
   ```

3. At this point we have to turn on the FPGA. Go back on the linux shell and launch the command:

   ```
   > profpga_run profpga.cfg --up
   ```

4. Click on *"Open Target"* under *"Hardware Manager"* then select *"Open New Target"*.
5. Select local host as server and 3 MHz as Jtag frequency.

Your system is finally connected successfully. You can debug the chosen signals of your architecture.
When you finished, through the *Vivado* line command write:

```
> close_hw_target
> disconnect_hw_server localhost:3121
```

Once the target is established, you can directly select it via *"Recent Target"* instead *"Open New Target"*, jumping the steps number 4 and 5.

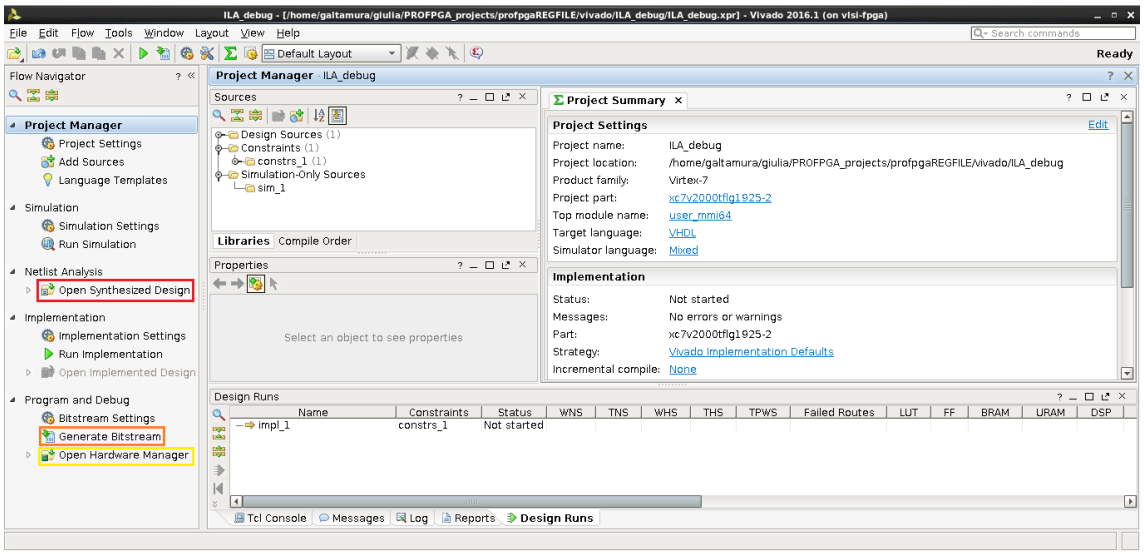


Figure 4.10: Vivado GUI

4.5 Working flow

Assuming to work with files in a folder organized as in the figure 4.11, let's see what you need to do for the implementation of your architecture and what are the various files with which we will work.

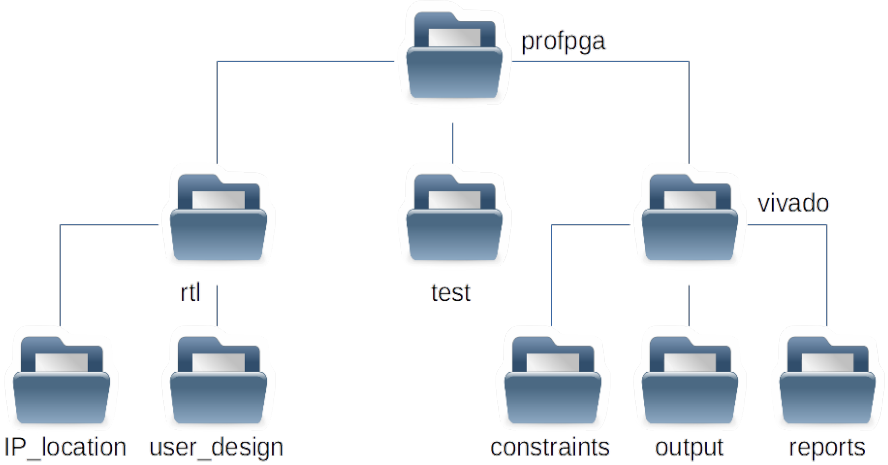


Figure 4.11: Work Directory

4.5.1 A simple test project

Initially, just to test the correct exchange of data between the PC and the FPGA, I realized a basic component, specifically a *register file*.
I described it as a set of three sub-component: 8 8bit-registers to save my data, 1 decoder in input to select where to save the data and 1 multiplexer to select the register from which take the output.
This four files will be put in the *user_design* directory. Obviously to be sure that the described implementation was written correctly, a test-bench of the same was created and simulated (see figure 4.12).

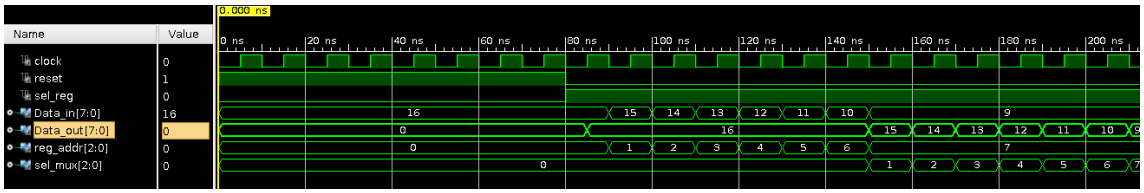


Figure 4.12: Register File simulation

At this point let's see the steps to follow to test the correct functioning of our user module.

1. Move in the *rtl* directory to modify the *user_mmi64.vhd* file including our design. The figure 4.7 shows the whole architecture that we will synthesized on the FPGA.

2. Correctly edited the file, go to the *vivado* folder.

3. First of all, you need to modify the *user_design.tcl* file, including the files that make up your architecture.

4. At this point all that remains is to launch the script *synthesized_me.sh*. If everything has been properly modified, the synthesis will advance till creating the netlist of your architecture that will be saved in the *output* folder. Otherwise, you should be able to read the errors and correct them.

5. Now that we have the synthesized netlist it is possible to set up the ILA core (follow the steps in section 4.4.1).

6. Once the correct behaviour has been verified and the bitstream generated, you must copy it in the *test* folder.

7. At this point the *main.c* file in the *test* foder must be adapted and compiled through the script *compile_me.sh*. If there are no errors nothing will be shown on the terminal and the *usertest* executable will be created. Every time you modify the *main.c* file, you must repeat this step.

8. Now you have everything to start the emulation so turn up the FPGA module and launch the command:

   ```
   > ./usertest profpga.cfg
   ```

   If you want to reboot the FPGA module just launch again the command:

   ```
   > profpga_run profpga.cfg --up
   ```

When you finish all the operation, remember to turn down the FPGA module launching:

```
> profpga_run profpga.cfg --down
```

and then switch off the motherboard.

4.6 Encountered problems

In this section we will try to give you useful information on the problems encountered during the use of the system.

4.6.1 PCIe connection

First of all, the use of the PCIe can be a little complicated because sometimes it is not recognised by the system. If the error:

```
Failed to scan mmi64 domain - status: E_MMI64_IDENTIFY_FAILED
```

occurs, the only way to correctly use it is to follow the protocol below:

- power up the proFPGA motherboard
- reboot the workstation
- power up the proFPGA module

4.6.2 Using an ILA core

In another case it is necessary to follow a precise sequence of commands to avoid problems: when creating the ILA core to debug the system. If the commands to be used are not execute in the correct way, like described in section 4.4.1, the Xilinx programmer will automatically disconnect from the PC, requiring the physical disconnection and reconnection of the device. To verify that the device is still connected to the PC, just simply launch the command:

```
> lsusb
```

If the programmer is correctly recognized, it will be shown something like this:

```
Bus 001 Device 003: ID 03fd:0008 Xilinx, Inc. Platform Cable USB II
```

Once verified that the device is actually connected you can follow the correct instructions previously provided for the creation of the core.

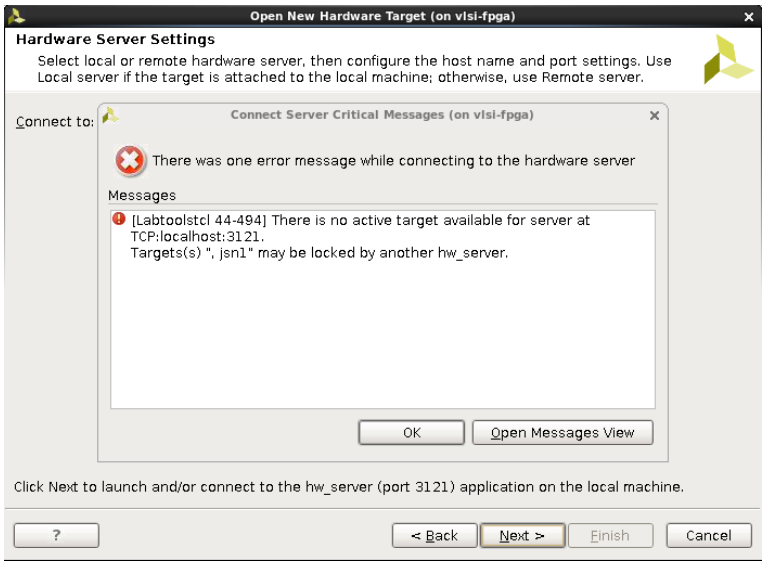


Figure 4.13: Possible error message

Chapter 5

Hardware Accelerator Development

We know that in a pre-trained ConvNet the main operation in the forward propagation path is the one of convolution. As already said, the implemented accelerator has the purpose to perform the GEMM operation in the best possible way.
Although each network performs multiplications of different matrices for each level and has a different number of levels from each other, being a C side of the system that manages the sending of data, it has been possible to create a "general" architecture always valid which repeats the same operation in loop. This architecture is made by simple elements such as registers, multiplexers and floating point units and is designed to perform the specific operation:

$$C'[m][n] = A[m][k] * B[k][n] * alpha + C[m][n] * beta$$

With reference to the theoretical notions previously reported, by means of some in-depth analysis, it has been possible to ascertain that:

- This operation is performed not only for convolutional levels but practically for almost all the levels of the network. The main difference lies in the fact that, usually, the convolutional levels operate a matrix-matrix multiplication, while other levels, such as, for example, max pooling, operate some other types of multiplications like vector-vector or vector-matrix or dot-vector multiplications.

- Looking at the equation above, as already said, the matrix A is identifiable as the weight matrix or, in other words, the filter that is applied to the portion of image that is being treated, represented instead by the matrix B.

- It has also been found that, according to the execution in *Caffe* of the *AlexNet* network, the GEMM function is recalled 20 times, eight of which to perform

the convolution. Although in figure 3.5a it is possible to see the representation of only 5 convolutional levels, these however coincide with the 8 function calls since three of these perform 1 convolutional level in two parts.

The table below shows the layers of the *AlexNet* network in which the function is called and for each of these the parameters: M (number of rows of matrices A and C), K (number of columns of matrix A and rows of matrix B), N (number of columns of matrices B and C), and the *alpha* and *beta* constants. The highlighted levels correspond to convolution layers:

Table 5.1: Analysis of AlexNet Layers

LV	M	K	N	alpha	beta
1	96	363	3025	1	0
2	96	1	3025	1	1
3	128	1200	729	1	0
4	128	1200	729	1	0
5	256	1	729	1	1
6	384	2304	169	1	0
7	384	1	169	1	1
8	192	1728	169	1	0
9	192	1728	169	1	0
10	384	1	169	1	1
11	128	1728	169	1	0
12	128	1728	169	1	0
13	256	1	169	1	1
14	1	9216	4096	1	0
15	1	1	4096	1	1
16	1	4096	4096	1	0
17	1	1	4096	1	1
18	1	4096	1000	1	0
19	1	1	1000	1	1
20	1000	1	1	-1	1

5.1 Operations flow description

In the diagram below, a representation of the whole operations flow is given, omitting the side of the accelerator that will be analysed later more in detail.

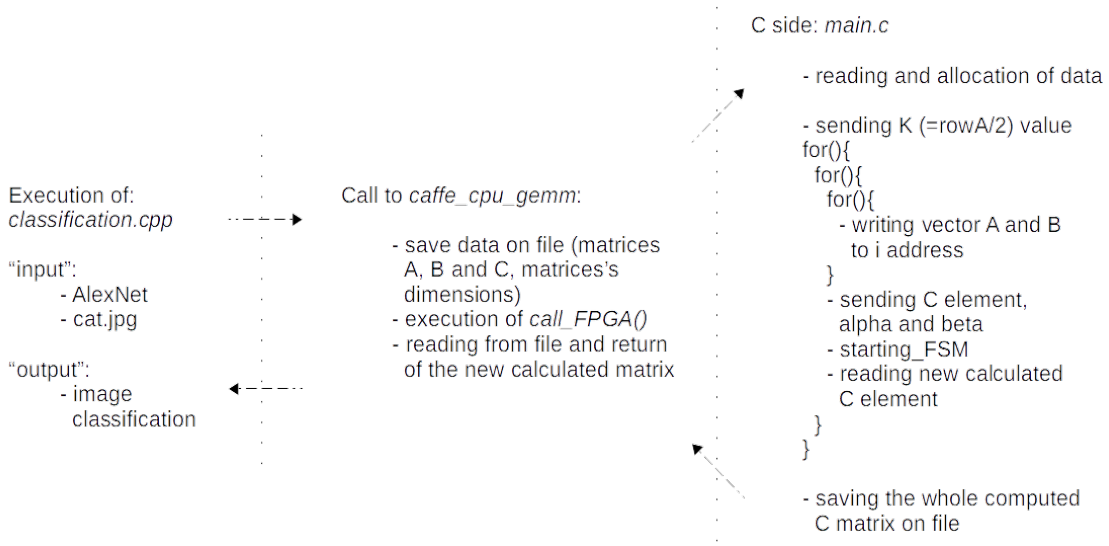


Figure 5.1: Operations flow

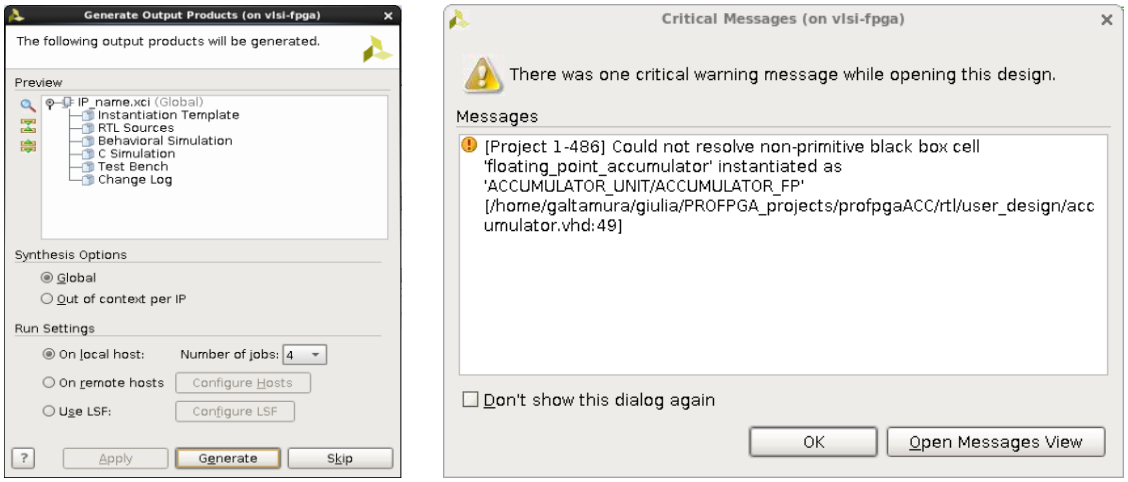
5.2 Used tool

Chapters 3 and 4 provide information regarding the framework chosen for this thesis work and the system on which the accelerator has been implemented. In any case, it is possible to say also that the operation of synthesis of the architecture was carried out through the *Vivado* tool. Thanks to this tool it was also possible to use IP cores customized specifically for the designed architecture (you can see in the section 5.2.1 how to use these correctly in a non-project mode).

5.2.1 Designing with IP cores

Some "pre-packaged" cores from *Vivado* were exploited to build the designed accelerator architecture. Specifically, floating point units (adders and multipliers) and memories were used.
Since in order to synthesize the architecture I inherited a script that runs *Vivado* commands in *non-project mode*, it was enough to insert simple commands to include the IPs in the project.
Prior to this, the desired IPs were customized using the graphical user interface. It is very important during this phase to select the "*global*" setting at the time of

confirmation to generate the IP, as shown in the figure below, otherwise this will be imported as black box.



(a) Global setting (b) Error Message in case of black box IP

Figure 5.2: IP setting

Returning to the changes to be made in the script (*vivado.tcl*), it was sufficient to make the following additions:

```
# Specify the creation of a project
create_project -in_memory -part xc7v2000tflg1925-2

# Adding the .vhd genereted description of the IP with all the other
# user file
source user_design.tcl

# Generate output products to be able to use it in the synthesis
file mkdir IP/IP_name
file copy -force ../rtl/IP/IP_name/IP_name.xci ./IP/IP_name
read_ip ./IP/IP_name/IP_name.xci.xci
generate_target all [get_ips IP_name]
set locked [get_property IS_LOCKED [get_ips IP_name]]
set upgrade [get_property UPGRADE_VERSIONS [get_ips IP_name]]
if {$locked && $upgrade != ""} {
upgrade_ip [get_ips IP_name]}
generate_target all [get_ips IP_name]
```

To have more information related to the usage of an IP you can see the documents [27] and [28].

5.2.2 Implementation settings chosen

Once the synthesis step is finished, the system can be debugged (see the section 4.4 for more detail) and implemented, in order to obtain the bitstream to be loaded into the FPGA.

Initially we tried to implement the design at a frequency of 100 MHz with the help of the "*ExtraTimingOpt*" setting, but since it was necessary to insert a decidedly high number of pipe stages, it was preferred to lower the frequency to 50 MHz. Done this it was enough to implement the design with the default settings already suggested by *Vivado*.

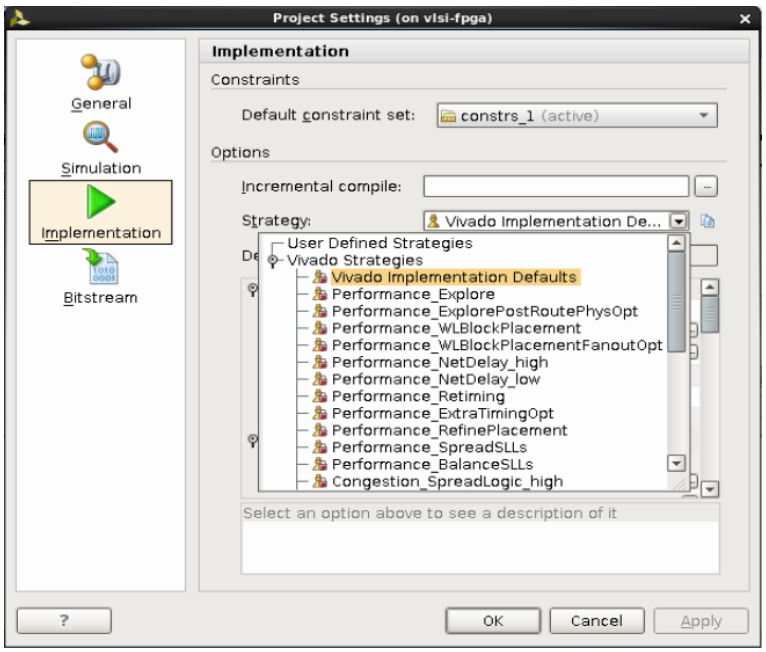


Figure 5.3: Implementation settings

5.3 Implemented design

5.3.1 First basic design

Knowing that we will be dealing with the matrices A (size MxK), B (size KxN) and C (size MxN), the first approach was to make a project based on the following simple iterative algorithm that allows the complete execution of the equation mentioned above:

```
for( m=0; m<rowA; m++)
        for( n=0; n<colB; n++){
                sum = 0;
                for( k=0; k<colA; k++)
                        sum += A[m][k]*B[k][n];
                C'[m][n]=sum*alpha+C[m][n]*beta;
}
```

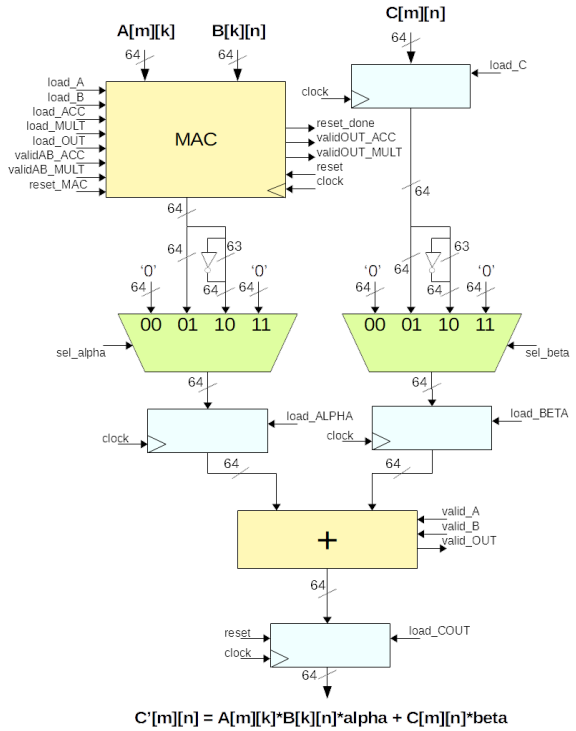
Considering the architecture shown in the figure 5.4a, we have:

- the *MAC* block that will carry out the innermost loop performing the multiplication and accumulation of the elements A and B;
- the multiplexers will allow the multiplication of the *sum* group for the constant *alpha* and of the element C for the constant *beta*;
- finally, the adder will give out the new calculated element C.

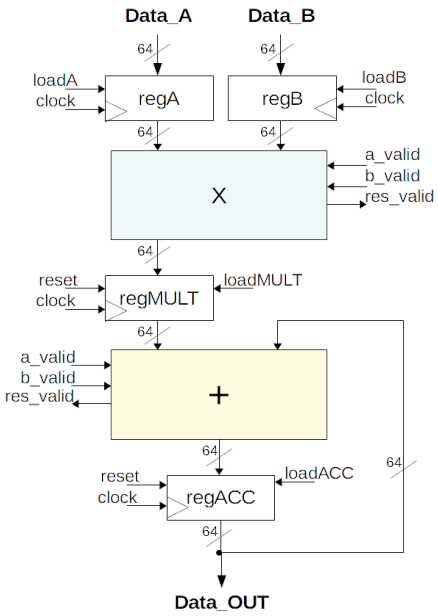
At last, these operations will be repeated until all the elements of the new calculated C matrix are obtained.
This structure fully functional but not very good especially for the latency of the communication system to synchronize the exchange of data. In any case, there would be no advantage to using this architecture instead of a CPU or a GPU because it did not perform well. Let's see how the choices have evolved to improve this first basic structure.

MAC unit

In the figure 5.4b is shown the MAC unit architecture. The floating point units have a clock signal because they are pipilened: in particular, the multiplier has two stage of pipeline, the adder instead just one. These decisions were reached after testing different configurations to avoid timing violations during the implementation step.



(a) First designed Accelerator architecture



(b) MAC architecture

Figure 5.4: First designed blocks

5.3.2 Simple Dual Port Memory introduction

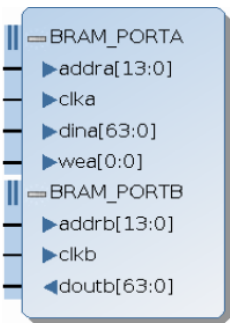


Figure 5.5: *Simple Dual Port Memory scheme*

Instead of sending to the architecture 3 data per time (element A[m][k], element B[k][n] and element C[m][n]), two memories (as shown in the figure on the side) have been introduced that will contain, from time to time, one, a vector of matrix A, and the other, a vector of the matrix B.
The chosen memories are Simple Dual Port and, as you can, they have two signal clock. This because we set the frequency of our architecture to 50 MHz due to the presence of the floating point units which require a certain computing time, and the frequency of the communication system to 100 MHz. So they allow to perform the write operation exploiting the first clock (100 MHz) and the read operation using the second clock (50 MHz).
After saving the two vectors in the two memories respectively, the *start* signal will be given to a FSM (reported in figure 5.6) which will process the data, calculating

an element of the new matrix C "automatically".

This improvement apparently allowed a gain of 30% from the point of view of timing performance thanks to the streamlining of the C side of the system. At this point, checks have been made that have highlighted the excessive slowness of the communication system, so it was decided to focus on optimizing only the computing architecture. So we will see, in the next paragraph, how we tried to exploit the availability of the FPGA with the parallelization of this structure.

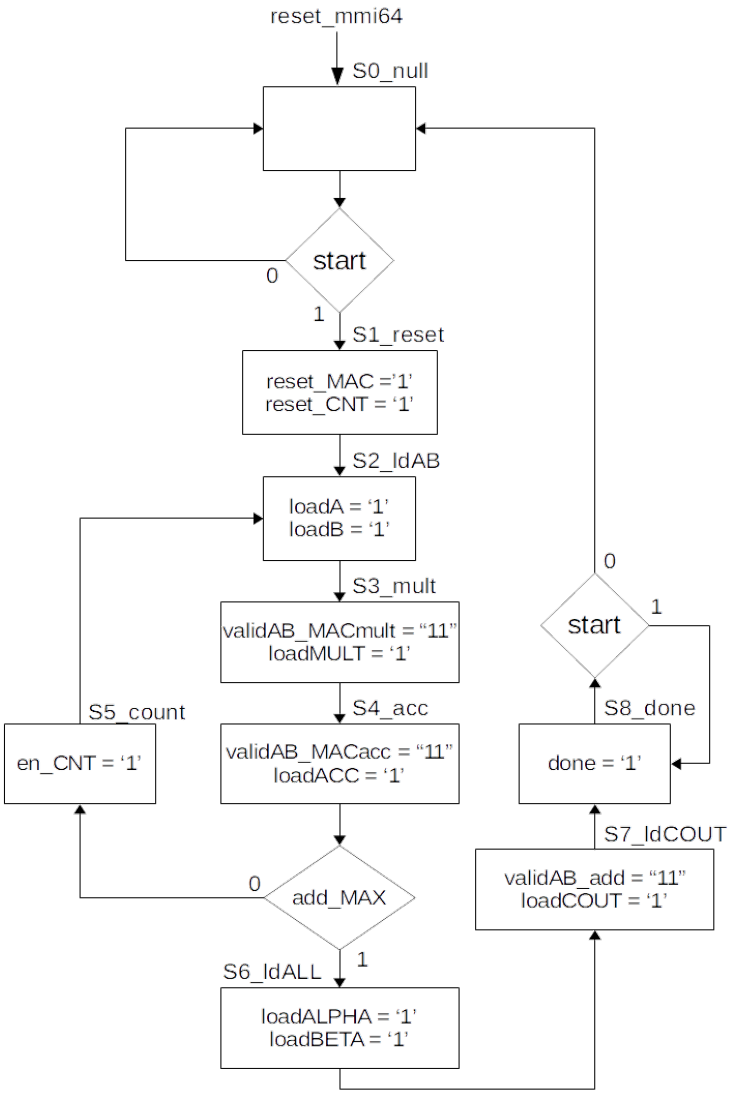


Figure 5.6: ASM control chart

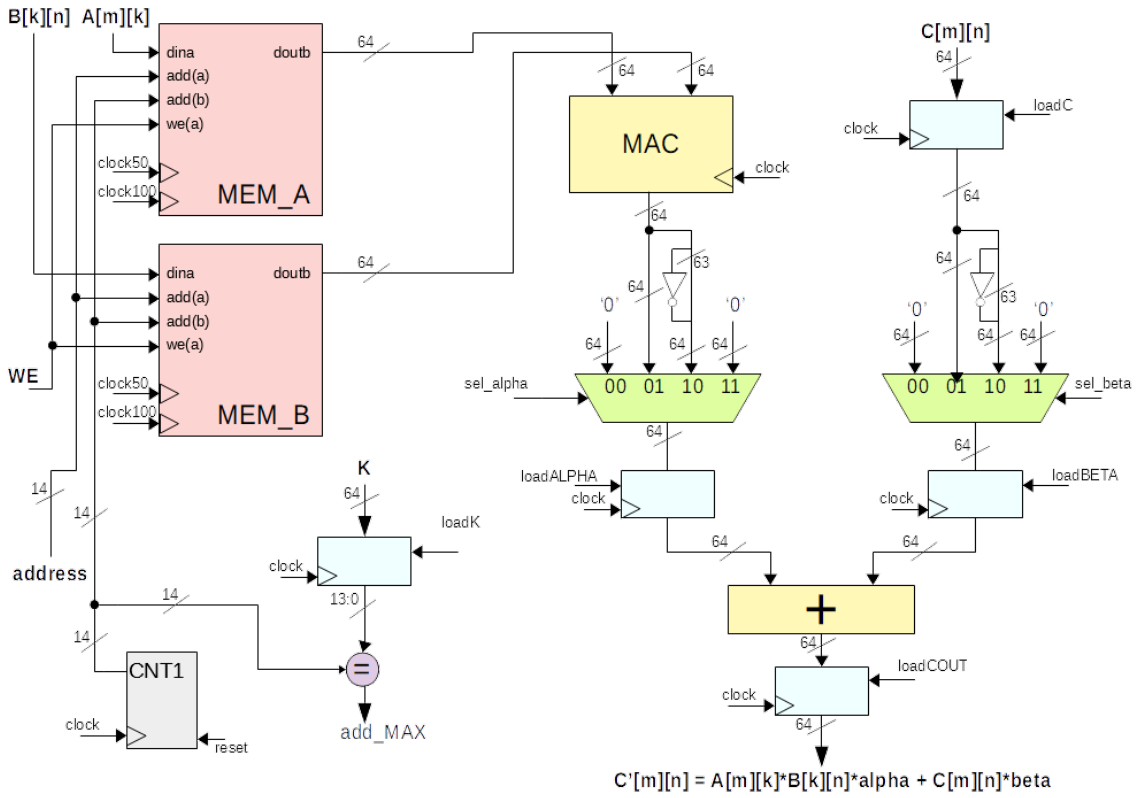
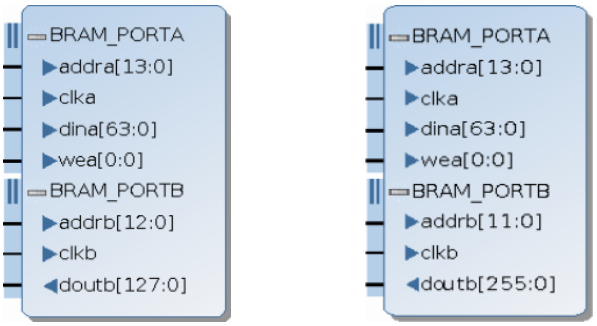


Figure 5.7: Architecture with memory introduction

5.3.3 Architecture's parallelization

As next step we modified a little the previous memories, increasing the width of the output port allowing to read more than one location per time. As you can see this also implies the reduction of the address bus that allows the reading of the data obviously. In this way, starting with duplicating the MAC unit it was possible to send, to each of these two units, half vector to be processed. As before, the writing operation to save the vectors on the two memories is managed from the C side, instead the read operation from the FSM.
The write operation includes:

- the load of the K value that will allow to the FSM to correctly manage the address to read;
- the load of the C element;
- the sending of the *start* signal to the FSM that will compute the new C element.



(a) Simple Dual Port 1 to 2

(b) Simple Dual Port 1 to 4

Figure 5.8: Modified Simple Dual Port Memory scheme

5.3.4 Two MAC units

In the figure below is reported the designed data-path, which is practically the same as before, just the MAC block has been duplicated and a further downstream adder has been added:

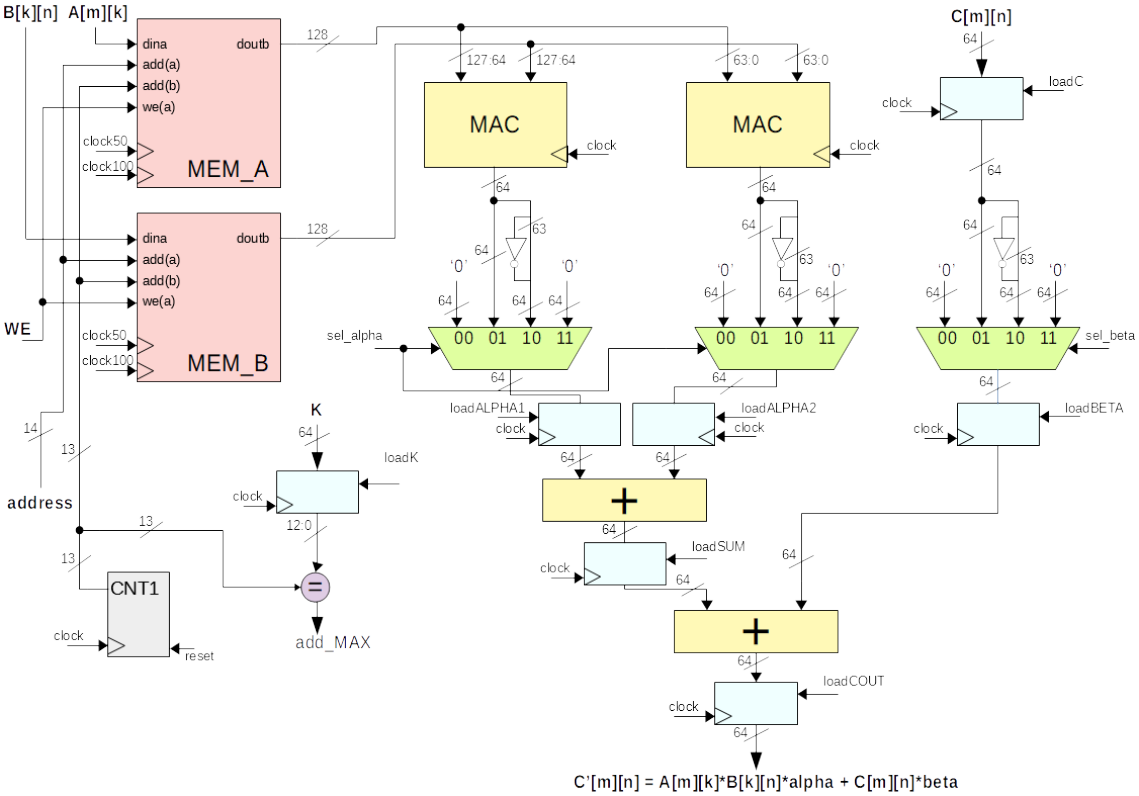


Figure 5.9: Parallelized architecture

As for the FSM instead, it was sufficient to insert a single state to manage the added adder downstream.

5.3.5 Increasing Parallelization

At this point, further parallelizing the architecture becomes quite simple. The changes that must be made from time to time are:

- customizing the two memories specifically and consequently also the counter which is necessary for the reading address of the same;
- duplicate the MAC units;
- duplicate adders and registers downstream;
- predict any state to manage the sums downstream

Like the two-unit MAC architecture described above, 4, 8 and 16 MAC drives have also been developed. In the next chapter, you will see information about the performance of the designed architectures discussed in this part.

Chapter 6

Results and Conclusions

The first intention of this work was to create a complete system, ie the accelerator also equipped with its communication system. During the implementation, however, it has been verified that the *MMI64* communication system is too slow for this purpose. Assuming for example to want to accelerate the third layer of the *AlexNet* network, to transfer, vector by vector, the only filter matrix A extracted from *Caffe*, which in this layer contains *384x2304* elements, the system takes about 6 seconds.

Once this was established, we focused on how to just optimize the accelerator, moving from a basic structure, to adding memories and finally to parallelization.
It has also to be said that, given the complexity of the floating point units working on 64 bits, has been chosen to decrease the frequency of the architecture to 50 MHz, compared to the frequency of the communication ProFPGA system (100 MHz), instead of complicating the architecture with an excessive number of pipe stages.

Below, in the next sections, you will find data collected from the different implemented configurations such as occupied area, speed performance and power consumption.

6.1 Results's verification

Despite the confirmation given by the *Caffe* framework on the correctness of the calibres carried out by the architecture, a graph is shown below for demonstration purposes.
Specifically, this test was performed on the calculation of the *128x169* elements of the matrix C of the *AlexNet* layer 12 made through the 4 MAC units architecture.

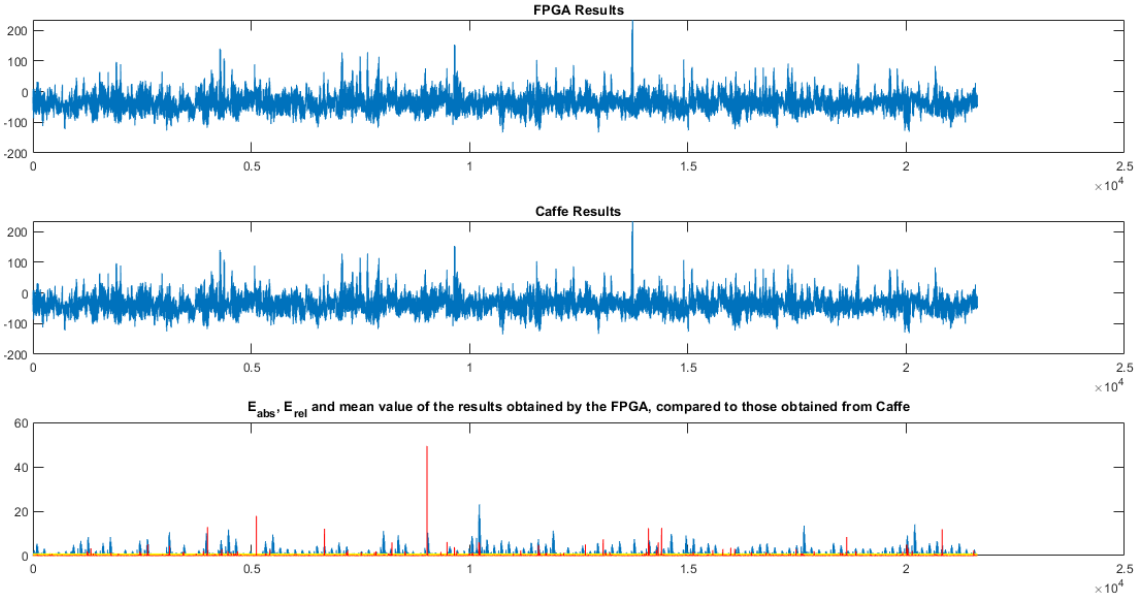
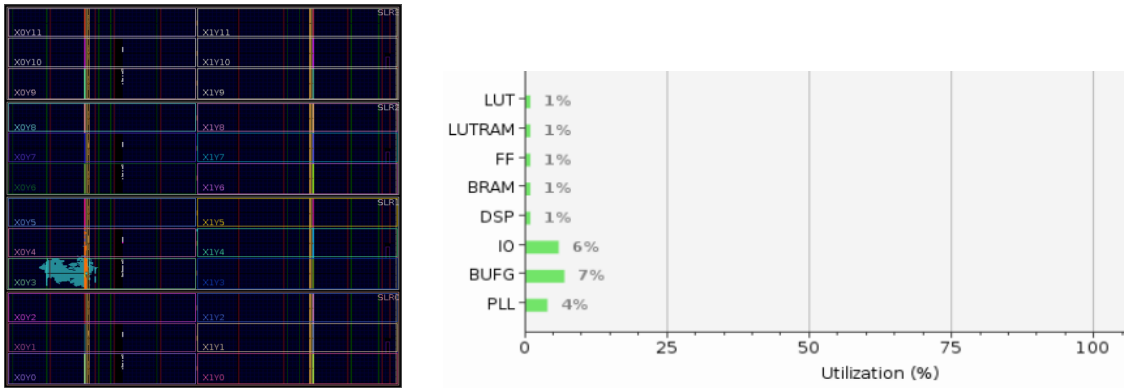


Figure 6.1: Comparison between *Caffe* and FPGA computation

In the first two panels it is possible to see the plot of the results calculated respectively by the FPGA and by *Caffe*. The two are almost identical, so to better appreciate the differences, in the last box you can see in blue the difference, in absolute value, element by element of the two calculated matrices, in yellow its mean value and in red the relative error. The latter is the one that shows us the actual correctness of the calculation performed, in fact, it is tending to zero (*0,3749* precisely), but it is also interesting to see where instead the calculations seem to have given different values thus bringing the average to that small deviation. Out of curiosity it has been found that the maximum difference between two elements is *23,0554*.
One of the reasons of this result can be the fact that while *Caffe* performs the calculations on 32 bits, the architecture uses 64 bit, with a greater precision therefore.

6.2 First basic design

For this first architecture (figure 5.4a) to which 3 data per time are sent directly from the *main.c*, without there being an actual FSM that takes care of managing the various signals, we can see the occupied area of the FPGA in the following 6.2a image (highlighted in light blue).



(a) Device (b) Percentage resources

Figure 6.2: Occupied Area First Design

Specifically, moreover, we report in the following table the number of components used:

Table 6.1: Report Cell Usage First Design

LUT	FF	BRAM	DSP
5325	5367	4	16

For this architecture the timing will not be commented because the control signals as well as data are sent through the writing functions of the C program, thus making everything decidedly too slow given the performance of the communication system.
As last, in the figure below is shown the power consumption of the designed architecture:

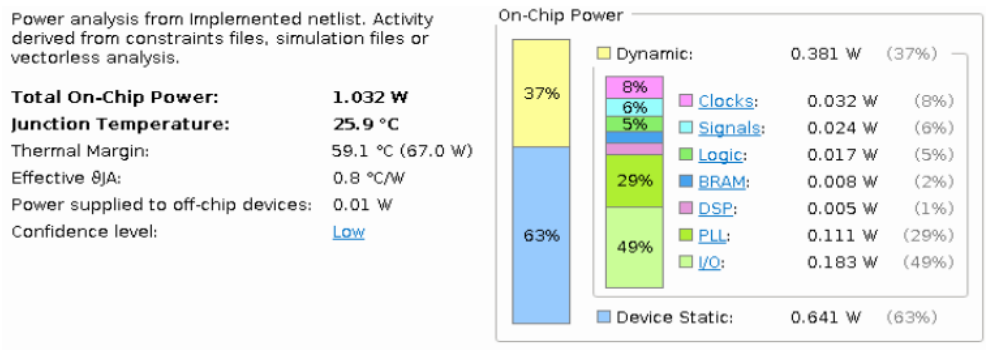
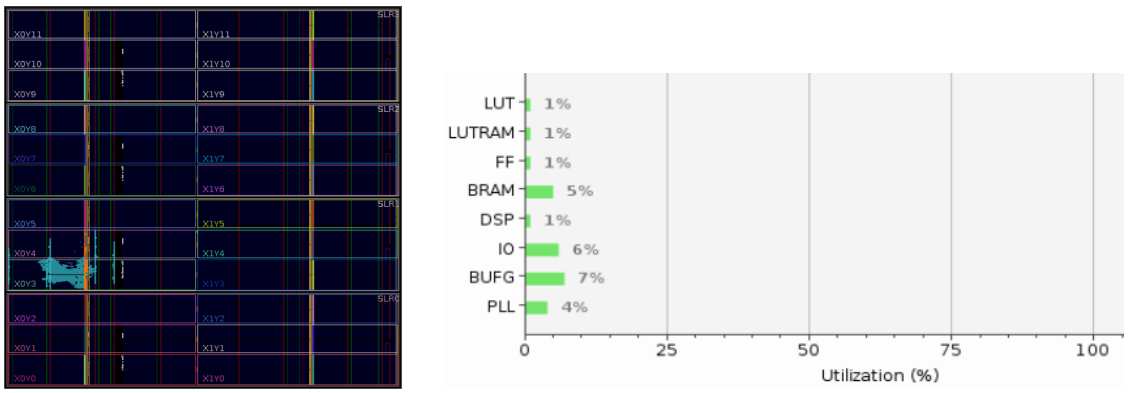


Figure 6.3: Power consumption first design

6.3 Simple Dual Port Memory introduction

Looking at the percentages of resource utilization reported in figure6.4b we can see, as could be expected, that the only parameter relating to the area that changes is that indicating the use of BRAMs. In fact, the only major change was the introduction of two *64x16K* memories and a FSM to manage the reading from the memories and the same calculation algorithm. The 14 addressing bits were decided according to the worst case, in this case on the basis of the vector containing the maximum number of elements (9216).



(a) Device (b) Percentage resources

Figure 6.4: Occupied Area Second Design

As before, we report in the following table the actual use of the most important cells.

Table 6.2: Report Cell Usage Second Design

LUT	FF	BRAM	DSP
5422	5448	61	16

About the speed performance of this algorithm, obviously regarding the latency we can say that it depends on the length of the two vectors that are being processed, as there is the loop of the MAC unit that will have to accumulate the sum of the products of all the elements. Once the loop is done and therefore having the final sum, the algorithm returns the final result in 3 cycles. You can see a more detail evaluation in table 6.4 in the next section.

As for the area, we can see small differences in power consumption. While the percentage of consumption given by the BRAMs had not been reported before, we can now see it appear.

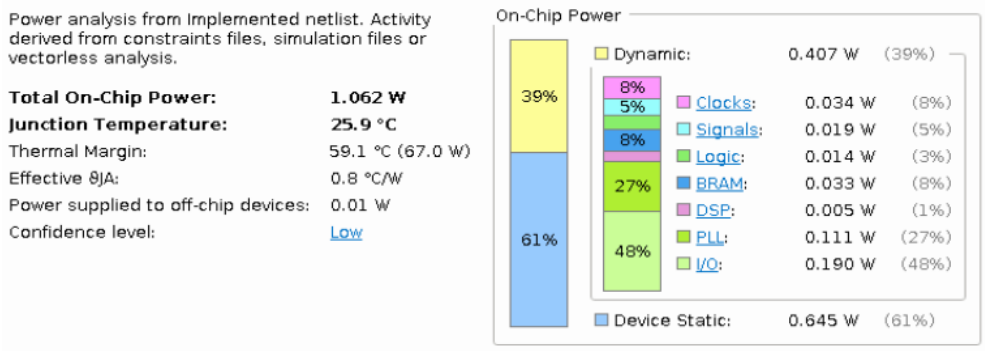
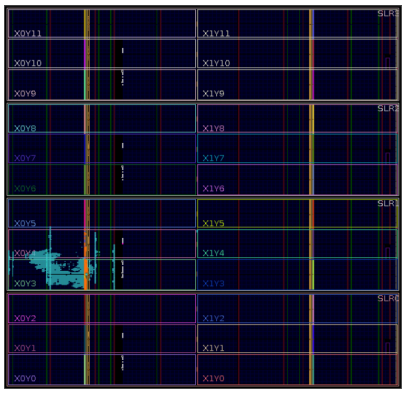


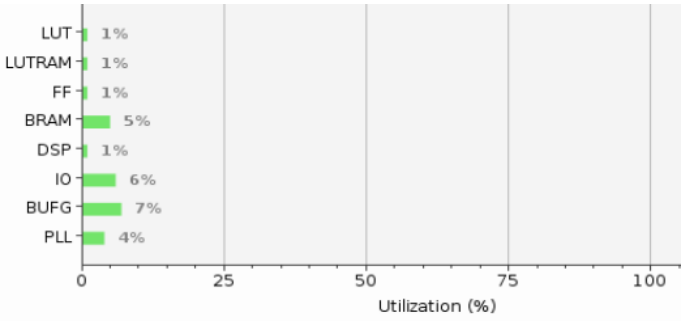
Figure 6.5: Power consumption second design

6.4 Architecture's parallelization

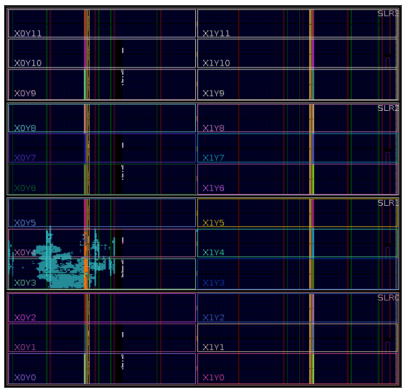
In this case, apart from the obvious increase of the various resources, we can see specifically from the table 6.3 how the number of DSPs doubles from time to time. This is because it is with these components that floating point architectures are built. We have 2 units in the MAC block (a multiplier and an adder) and one downstream of the accelerator (another adder), now having the architecture parallelized, in the first case for example, we have 2 MACs and 2 adders downstream, so, exactly double.



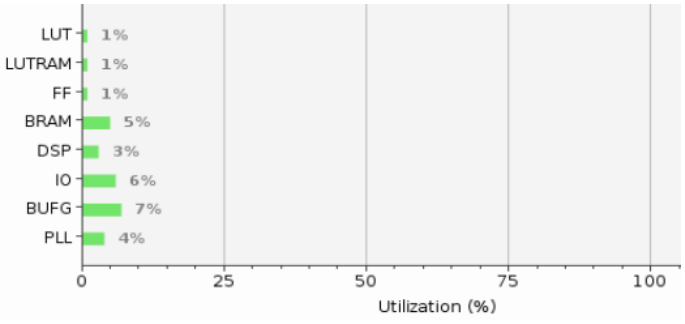
(a) Device 2 MAC units architecture



(b) Percentage resources 2 MAC units architecture



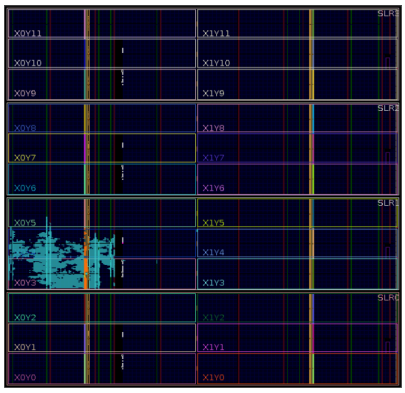
(c) Device 4 MAC units architecture



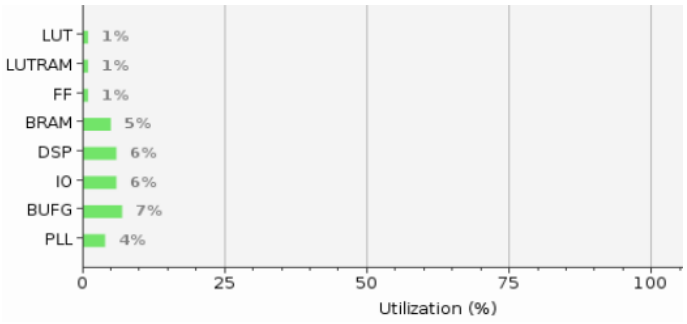
(d) Percentage resources 4 MAC units architecture

Figure 6.6: Occupied Area TWO and FOUR MAC UNITS

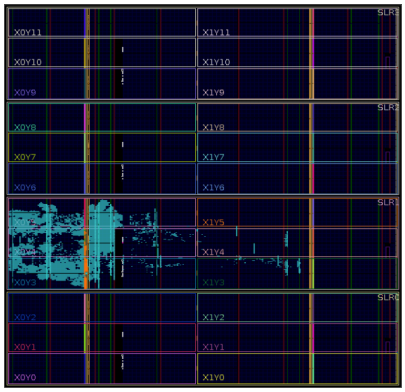
From the figures above and below we can see just the change in the percentage of use of the DSP components.



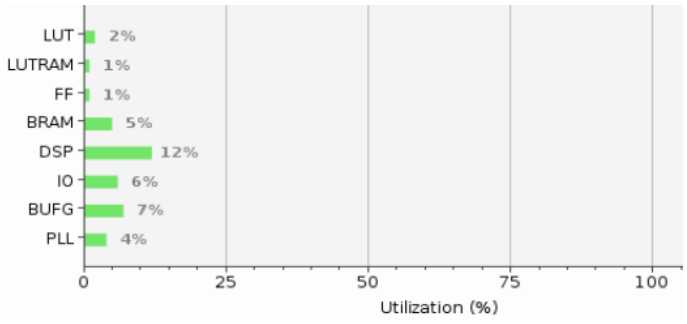
(a) Device 8 MAC units architecture



(b) Percentage resources 8 MAC units architecture



(c) Device 16 MAC units architecture



(d) Percentage resources 16 MAC units architecture

Figure 6.7: Occupied Area EIGHT and SIXTEEN MAC UNITS

Following the table with the main resources used for the different architectures:

Table 6.3: Report Cell Usage Parallelized Designs

MAC units	LUT	FF	BRAM	DSP
2	7197	5826	61	32
4	10647	6855	61	64
8	17898	8156	61	128
16	29908	11224	68	256

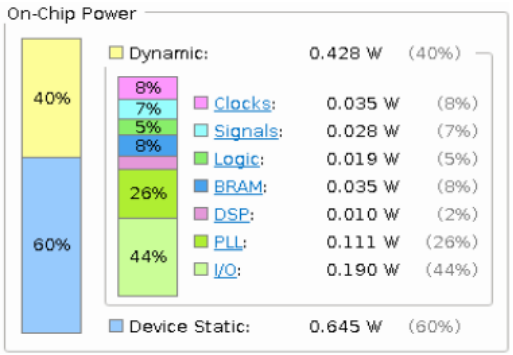
Regarding the speed performance of the algorithm, depending on the architecture, we will have the cycles spent in the loop greatly reduced by the increase of the MAC units working in parallel, paying only with additional cycles for the final calculation given by the adders tree that becomes more and more great as the parallelism of the whole structure increases.

Assuming that we have two vectors of 2304 elements, we see in the following table how the number of execution cycles necessary for the calculation of an element varies.

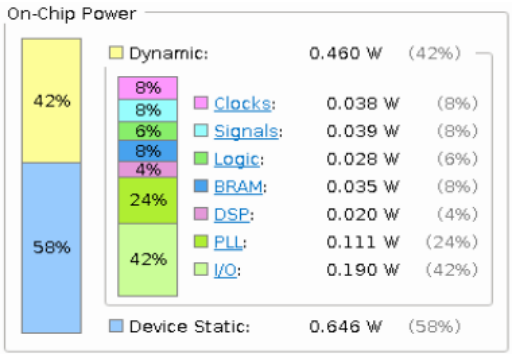
Table 6.4: Evaluation of the number of execution cycles

MAC units	Number of cycles (1 cycle = 20 ns)
1	9219
2	4604
4	2309
8	1158
16	569

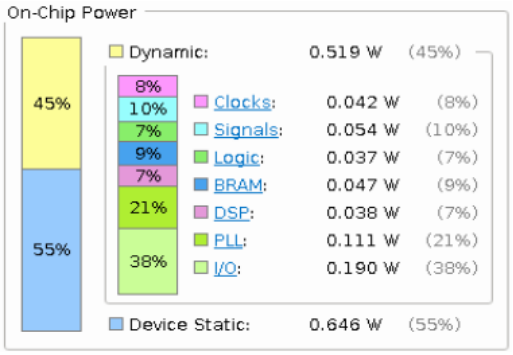
Also for the power we can see how the consumption varies in particular regarding the various *signals*, the *logic* and the *DSP* components. We can highlight how, while the static power remains almost unchanged, we have instead a significant variation in the dynamic power.



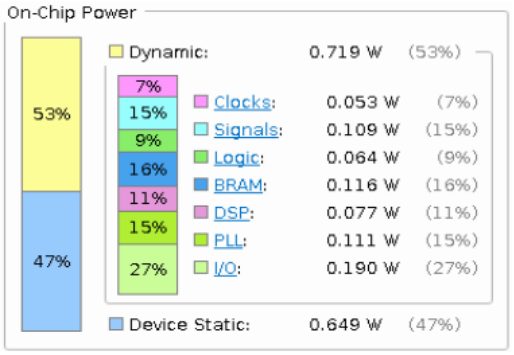
(a) TWO MAC UNITS power



(b) FOUR MAC UNITS power



(c) EIGHT MAC UNITS power



(d) SIXTEEN MAC UNITS power

Figure 6.8: Power consumption Parallelized Designs

6.5 Conclusions and future works

The ConvNet have proved to be an excellent solution for the image classification and other applications. We also know, however, that with the increase in the depth of these networks, the computational load increases exponentially, making them therefore not very "portable".

In this thesis, therefore, an architecture was developed that would implement the matrix multiplication operation carried out by a particular library function called by the CPU for the execution of the convolutional layers and not only.
The FPGA on which the architecture was implemented, a Xilinx Virtex 7, promised great things given its capacity, on the other hand the available communication system, ProFPGA *MMI64*, turned out to be very poor performance preventing a system optimization in a global sense.
So we concentrated on first doing something working and then optimizing by parallelizing the MAC units of the accelerator alone.

But the applied optimizations are obviously not the only ones possible. For any future work, therefore, it is possible, to begin with:

- given the bad performance of the communication system, trying to change it, thinking of something completely different;
- opting for a different type of parallelization. Instead of increasing the MAC units, allowing communication system, one might think of having even more "accelerator" units in such a way as to multiply several pairs of vectors at a time and then calculate more elements of the new C matrix in parallel.

Bibliography

[1] M. T. Jones, "A beginner's guide to artificial intelligence, machine learning, and cognitive computing," 2017.
[2] J. Le, "The 10 deep learning methods ai practitioners need to apply," 2017.
[3] B. Cipollini, "Deep neural networks help us read your mind," 2015.
[4] "A beginner's guide to understanding convolutional neural networks," 2016.
[5] S. Chatterjee, "Different kinds of convolutional filters," 2017.
[6] A. P. N. Iyer, "Overview of convolutional neural networks," 2017.
[7] "Trying to confirm average pooling is equal to dropping high frequency fourier coefficients using numpy," 2017.
[8] "Neural networks in javascript," 2016.
[9] P. Warden, "Why gemm is at the heart of deep learning." Blog, 2015.
[10] C. Bourez, "Deep learning tutorial on caffe technology : basic commands, python and c++ code," 2015.
[11] E. Garolla, "Adaptation and implementation of a lim architecture on a multi fpga system," 2017.
[12] PRO DESIGN Electronic GmbH, *FPGA Based Prototyping Solution*, 2017.
[13] "Fpga cpld platform cable usb ii."
[14] Y. Jia, "Learning semantic image representations at a large scale," *UC Berkeley Electronic Theses and Dissertations*, 2014.
[15] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. B. Girshick, S. Guadarrama, and T. Darrell, "Caffe: Convolutional architecture for fast feature embedding," *CoRR*, vol. abs/1408.5093, 2014.
[16] "Blobs, layers, and nets: anatomy of a caffe model."
[17] "Alexnet."
[18] "Caffenet."
[19] PRO DESIGN Electronic GmbH, *Quad V7 Prototyping System*, 2014.
[20] PRO DESIGN Electronic GmbH, *Hardware User Manual*, 2014.
[21] PRO DESIGN Electronic GmbH, *XC7V2000T FPGA Module*, 2013.
[22] PRO DESIGN Electronic GmbH, *Profpga Builder Software*, 2014.
[23] PRO DESIGN Electronic GmbH, *Getting Started with MMI-64*, 2013.
[24] PRO DESIGN Electronic GmbH, *Using Vivado ILA 2.0*, 2013.
[25] XILINX, *Vivado Design Suite User Guide - Programming and Debugging*, 2016.

[26] XILINX, *Vivado Design Suite Tutorial - Programming and Debugging*, 2016.
[27] XILINX, *Vivado Design Suite User Guide - Designing with IP*, 2014.
[28] XILINX, *Vivado Design Suite Tutorial - Designing with IP*, 2014.