POLITECNICO DI TORINO

DIPARTIMENTO DI ELETTRONICA E TELECOMUNICAZIONI (DET)

Master Degree in Electronic Engineering

Master Degree Thesis

A SIMD Application Specific Instruction-Set Processor for Convolutional Neural Networks

Supervisors
Prof. Maurizio MARTINA

Candidate
Fabrizio FRISARI

October, 2019

Acknowledgements

First of all, I would like to thank Prof. Maurizio Martina at Politecnico di Torino. Always polite and available for any doubt or question. He quickly solved any issue with the server or the software. Truly inspirational.

A thank to the PhD candidate Riccardo Peloso, who constantly tried to help me in any circumstances. He followed me all along this work, he gave me a lot of inspiration for the development of the thesis and he helped me to solve many problems that arose in these months.

Thank also to the PhD candidate Maurizio Capra: he was there, ready to help me for any doubt (even though I did not want to annoy him too much) and he supported me in the meetings of the work with the Professor.

A huge thank to my parents and to my brother Daniele for all the sacrifices and the support. Surely you had a lot of patience. Now we can happily travel to Carcassonne.

Thank you Caterina, my lovely and unlimited source of strength. None of this was possible without you by my side. You never give up with me. Last but not least, you helped me with the dots in the image.

A thank to my friends and colleagues Angelo, Andrea and Mattia. They have been a great source of insights and they always were available for any doubt or question in all these years.

Thank to all my old friends, a constant part of my life, always there for me. You were my secret source of fun and inspiration.

Summary

Machine Learning (ML) is one of the greatest inventions in the last decades and it has completely changed the way of programming: with just a single program, the machine is able to learn by itself how to perform many different tasks. Machine Learning algorithms need a really large number of data for the learning process and great amounts of computational power; both are accessible only in the last decades and this is the reason why this technology has developed only in the last decades.

There is an area of ML called Neural Network (NN) that is often referred as *brain-inspired computation*, since the program emulates the capability of the human brain in learning and solving problems. Its basic block is the neuron: it receives an input element and outputs a nonlinear function. The latter is called activation function and, by connecting many of them in a network, it is possible to obtain a large computational unit, able to perform a specific task. The simplest NN has three layers: input layer, hidden layer and output layer. All those layers can have a large amount of neurons.

Really complex structures of neural networks are called DNN (Deep Neural Network) and define the DL (Deep Learning) area. Those networks have a great number of hidden layers and reach really high performances, with respect to simple NNs. They are used in many tasks, such as speech recognition, healthcare applications and computer vision problems, like object detection and image recognition. There are a lot of DNN solutions, one of the most popular is the Convolutional Neural Network (CNN). It is a very good model for computer vision applications.

In this work, a Convolutional Neural Network is created from scratch; the code is written in a mix of C/C++ code. Then, it is loaded in the Synopsys software *ASIP Designer*: it is a tool used for designing custom processors or programmable accelerators for different applications. Indeed, ASIP is the acronym of Application Specific Instruction-Set Processor and with this software it is possible to exploit this approach to realize custom processors. The tool uses its high level Hardware Description Language (HDL), called *nML*, to describe the architecture of the processor.

Starting from a default processor model, it is possible to modify it to realize a custom application. Moreover, ASIP Designer provides useful instruction profiler reports that emphasizes which instructions have the heaviest computational cost. This information can be used to easily optimize the software code and the structure of the hardware processor.

Starting from a simple CNN software solution, different implementations of the network are created. A Single Instruction Multiple Data (SIMD) approach is used to optimize and perform the convolution operation with few instructions. In the end, all the optimizations are adopted in a more complex CNN structure, smaller but similar to the most famous DNN models, such as VGG-16 and AlexNet. A final comparison of power consumption and area occupation is done for the different network implementations used in this work.

Contents

List of Tables

List of Figures

Listings

Chapter 1

Introduction

1.1 General principles

Nowadays, Machine Learning is one of the greatest recent invention and it is already changing the everyday life. ML is used by everyone, even without knowing it, and its influence will constantly increase in the next decades. It is a subfield of Artificial Intelligence (IA) and it has been defined for the first time by Arthur Samuel in 1959: "the field of study that gives computers the ability to learn without being explicitly programmed" [2]. This means a completely new way of programming: the single program is able to learn by *itself* how to do particular tasks, starting only from few inputs. Therefore, without being explicitly programmed, the algorithm can perform different applications. The border between humans and machines is thinner than ever and many researchers are exploring all the possibilities that a technology like this can give.

Even though Machine Learning is not a recent invention, it is largely used only in the last decades. One of the reason is the growth of technology, of the internet and automation. Basically, there are much larger data sets to exploit than ever before. Indeed, Machine Learning algorithms need a large amount of data for the learning process and those are accessible only in the recent years. They also need a huge amount of computational power that can be provided only with the recent technologies.

In 1959, Arthur Samuel, "an American pioneer in the field of computer gaming and artificial intelligence" [18], wrote a machine learning checkers program that was able to learn the good and bad board positions. The program increased its experience after playing thousands of games by itself and, in the end, it became a better player than Arthur Samuel himself. With today's technologies, the ML can learn to do almost everything. Some of the most important applications are:

- Computer vision: it is possible to detect and classify objects from images and videos (like video surveillance).

- Speech recognition and machine translation.
- Social media management and online advertising, like recommendations for services, interests, and products.
- Autonomous driving.
- Gameplay.
- Medical operations: disease detectors or devices that can help with health monitoring of the patients.
- Robotics.

In the future, ML will surely develop in more areas: for instance, it will possibly manage the traffic control or forecast the weather with precision. It could also improve cancer detection and have a huge impact in the world of finance [2].

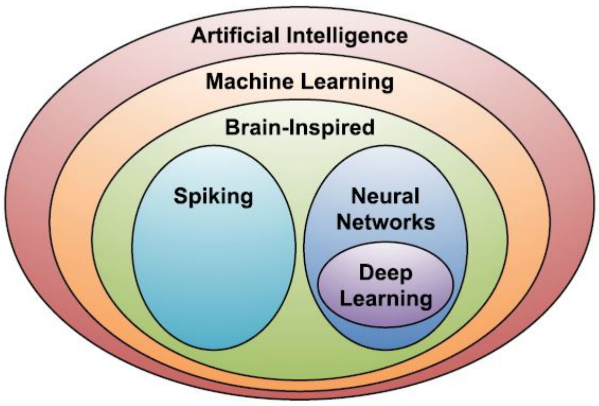


Figure 1.1: Diagram of artificial intelligence [1]

In Figure 1.1, there is a complete diagram of the AI. As already mentioned, Machine Learning is a subarea of Artificial-Intelligence; anyway, it can be observed an area of ML that is often referred to as *brain-inspired computation* [1]. It refers to programs that emulate, in some way, the human brain and its capability in learning and solving problems. A particular example is called Neural Network (NN), also shown in Figure 1.1. Its basic element is the neuron since it is also the principal computational element of the brain. A network is a composition of many neurons; thanks to this connection, the machine provides a large computation unit able to perform the specific task.

A particular subarea of the NN is the Deep Learning (DL). It is a particular case of a neural network: the basic elements of the structure are the same, but there are a huge amount of them to create a complex network that is called Deep Neural Network (DNN). This structure can provide very high performances and can perform almost any task. Today, DNNs are the most advanced ML architectures.

One of the most popular DNN structure is the Convolutional Neural Network (CNN) [3]. This architecture is the most used for computer applications, such as object detection and image recognition. In the last decades, many CNN models have been developed and they will be analyzed in the next chapters.

1.2 In the following chapters

The thesis will be organized in this way:

- Neural Network overview. In this chapter, the main concepts of the Machine Learning algorithms are analyzed in detail. It is explained the structure of a neural network and how it works. It is stressed the importance of *going deeper* and why the Convolutional Neural Network is one of the best DNN architecture. In the end, there is a comparison between the today's most popular Deep Neural Networks.
- ASIP Designer. This chapter is necessary to explain the Synopsys tool used in this work. It is called ASIP Designer and there is a brief description of how it works and what this tool can provide to the user. Since the software allows the realization of Application Specific Instruction-Set Processors, it is explained what this ASIP approach is; moreover, the latter compared with two other possible approaches, such as General Purpose (GPP) and Application Specific Integrated Circuit (ASIC).
- ASIP implementation. The starting point of the work is described here. This chapter highlights how the ASIP Designer tool is used and which processor core is chosen. Then, there is the description of the C/C++ code of a complete Convolutional Neural Network and its functions. There are, in total, five implementations of the network: exploiting the instruction reports of the Synopsys tool, many optimizations are performed to increase efficiency.
- Result and analysis. In this chapter, it is described how to stem the HDL of the processor from ASIP Designer and how to use it to simulate and synthesize the code. A brief comparison among the implementations is made.
- Final conclusion and future works. This chapter summarizes the obtained results and explains how the best implementation can be furtherly upgraded in future works.

Chapter 2

Neural Network overview

2.1 What is Machine Learning?

Machine learning develops computer programs that make use of several accessible data to learn for themselves. Another definition of ML is made by Tom Mitchell in 1998: "a computer program is said to learn from experience E with respect to some task T and some performance measure P, if its performance on T, as measured by P, improves with experience E" [2]. For instance, in the checkers playing program described in the chapter 1.1:

- the experience E represents the thousands of games that the machine played against itself, growing its experience;
- the task T is the task of playing the game;
- the performance measure P is the probability of winning a checkers match against new opponents.

There are a lot of different types of learning algorithms; the main two are the *supervised* and the *unsupervised* learning [3]. In the first case, the user teaches the system how to do a function, having some input data. In the latter, the machine finds previously unknown patterns in data set; basically, it learns by itself.

As regards supervised learning, the machine tries to predict the right output starting from a privuded labeled data. This means that the machine, after some training, analyzing a set of correct input-output pairs, learns to produce the correct outputs given new input data. Some examples of these algorithms are the housing price prediction application, the speech recognition, and the online advertisement.

The mapping between input and output can be made with many functions, such as logistic and linear regression. However, nowadays Neural Networks are much better than ML algorithms since they can be used for more complex structures.

NNs can afford a large number of non-linear input and parameters and thus they offer a larger computation [3].

2.2 Neural Networks

2.2.1 Introduction and basic concepts

The neural network emulates the behavior of the human brain. Scientists are still researching new information about how the brain works but is commonly known that its main element is the neuron. In Figure 2.1 there is a representation of a single neuron [22].

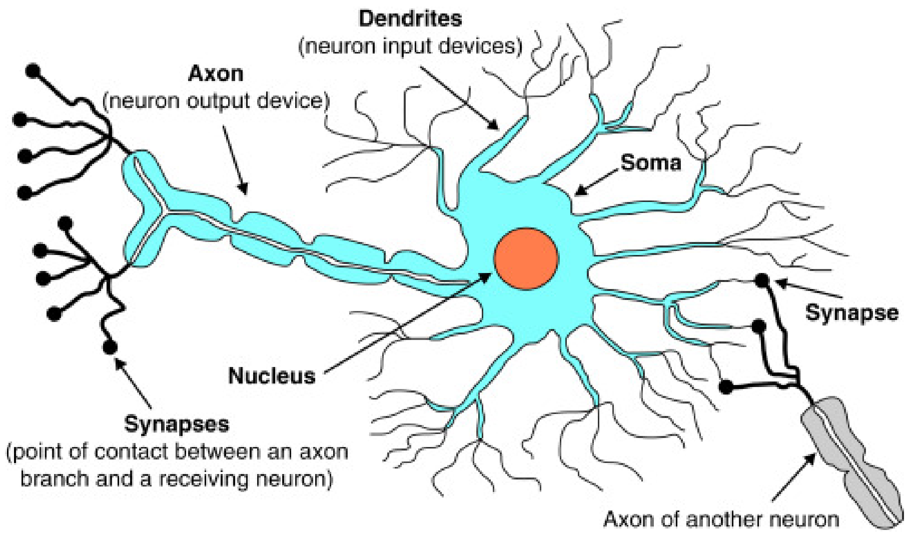


Figure 2.1: Representation of a neuron. [22]

There are billions of them in the brain, all connected together with the so-called *dendrites*, that are the neuron inputs. Instead, the output is the *axon*. The latter is connected with the dendrite of another neuron, forming the *synapse*. Therefore, the neuron receives the input signal entering via the dendrites, makes the computation and produces the output signal on the axon. Both signals are called *activation*.

One important feature to take into account is the capability of the synapse to scale the signal, as can be seen in Figure 2.2. That scaling parameter is called *weight* w_i and the network provides different outputs, for the same input, by changing this value [1]. Therefore, the structure of the machine is always the same, what changes is the response to the input stimulus. A scheme of the neural network is shown in Figure 2.3.

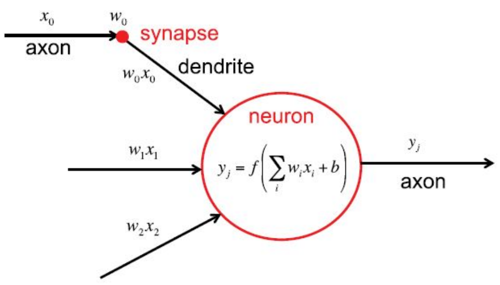


Figure 2.2: Connection to a neuron in the brain. [1]

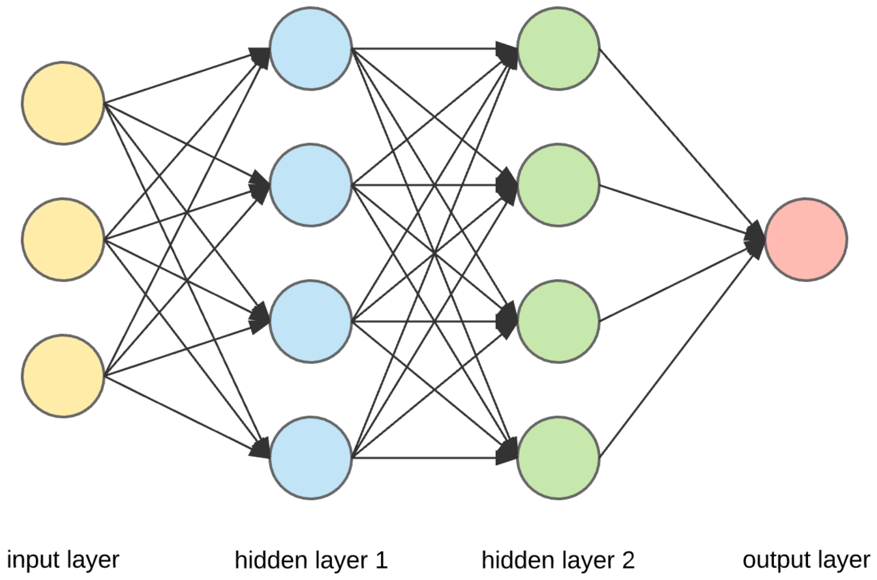


Figure 2.3: Example of a simple neural network. [23]

In this case, there are three input units that form the *input layer*. Then there is also the final layer, also called the *output layer* since it generates the final value of the network; between input and output, there are the *hidden layers*. They are so-called because the values of inputs and output of a neuron cannot be observed in that layer [3].

In Figure 2.2, inside the red circle, it is highlighted what normally a single neuron computes:

$$y_j = f\left(\sum_i w_i x_i + b\right) \tag{2.1}$$

where w_i are the weights, x_i are the input activations, while y_i are the output

activations. There is a bias factor b as well and usually its value is set to 1.
$f(\cdot)$ represent a non-linear function that is called *activation function*.

2.2.2 Activation functions

In Figure 2.4 there are some examples of non-linear activation functions.

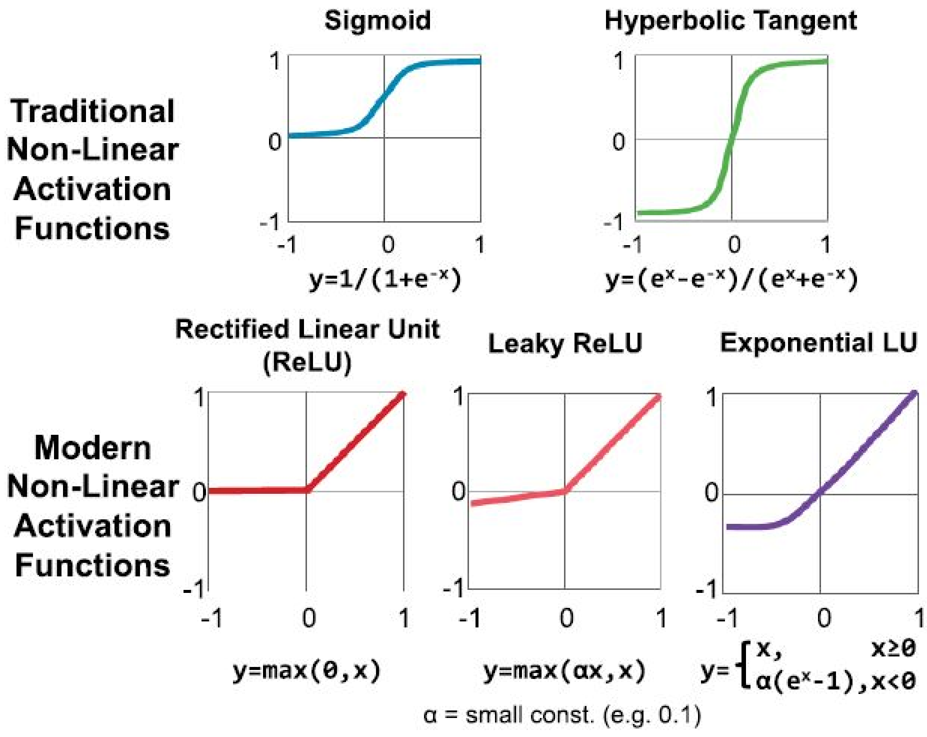


Figure 2.4: Nonlinear activation functions. [1]

First of all, the two traditional activation functions will be analyzed:

1. *Sigmoid function*:

$$y = \frac{1}{1 + e^{-x}} \tag{2.2}$$

It is one of the first functions used in literature. Indeed, it was used for the LeNet architecture [4], as described in the chapter 2.5. However, nowadays it is rarely used, especially for hidden layers. Indeed, this model is not really good for the training of the network: the gradient of the function can be small if x is large or small. This problem is called *vanishing gradient* and causes the gradient descent algorithm to be very slow [3].

Anyway, one peculiarity of the sigmoid function is that exists between 0 and 1: it can still be useful for probability predictions in the output layer.

2. *Hyperbolic tangent*:

$$y = \frac{e^x - e^{-x}}{e^x + e^{-x}} \tag{2.3}$$

This function has a range between -1 and 1. It can be seen as a shifted version of the previous algorithm that now it crosses the zero, indeed:

$$tanh(x) = 2\, sigmoid\,(2x) - 1 \tag{2.4}$$

This model is better for the training of the machine with respect to the sigmoid function: the derivatives are steeper and the *vanishing problem* is partially reduced (but still present) [3].

In the recent years, many nonlinear function has become popular and replaced the sigmoid and tangent functions in a lot of applications. Some of them are:

- *Rectified Linear Unit (ReLU)*:

$$y = max\,(0\,,\, x) \tag{2.5}$$

Nowadays, this is one of the most diffused activation function in the world, thanks to its simplicity and lower computational cost.

In particular, the derivative of the *rectifier linear unit* (ReLU function) is equal to 1 when x is positive; it is equal to 0 when negative. This allows a faster training than the two traditional algorithms. One disadvantage of this model is its behavior with negative values: it is impossible to fit or train the negative inputs properly since those values are all immediately set to zero [3].

- *Leaky ReLU*:

$$y = max\,(\alpha x\,, x) \tag{2.6}$$

where α is a small constant, usually set to 0.01. This function solves the main disadvantage of the ReLU since it has a better behavior when x is negative.

- *Exponential LU*:

$$y = \begin{cases} x & \text{if } \; x \geq 0 \\ a(e^x - 1) & \text{if } \; x < 0 \end{cases} \tag{2.7}$$

where a is a hyperparameter and has to be greater than zero. This function is used in the most recent works, such as a Deep Neural Network in 2016, and it results more accurate, less complex and faster than the previous ReLUs. It furtherly reduces the vanishing gradient problem and has mean activation close to zero [25].

All those activation functions are nonlinear. But what happens if a linear function, like the one in Figure 2.5, is used?

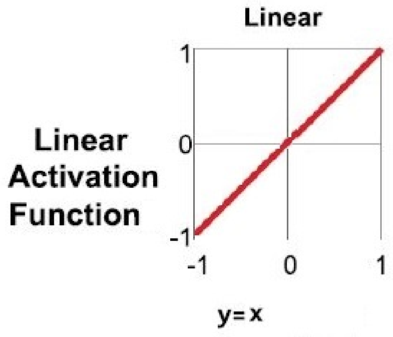


Figure 2.5: Linear activation function.

The range of the linear activation function, also called identity activation function, is not limited; it just outputs any input. There are two main problems: first, the learning algorithm would work with a constant gradient, since the derivative does not change. Second, even if the network is composed of a huge number of linear layers, the output would still be a linear function. Therefore, there is no reason to stack many layers, one is sufficient; in this way, without a deep network mechanism, it is impossible to detect some interesting features. The only application for a linear activation function is at the output layer of a machine that works on a regression problem. In the hidden layer, a nonlinear function is strictly necessary [3].

As can be seen, there is not always one activation function more valid than the others. For this reason, different functions can be adopted in different layers, depending on the requirement of a certain step.

An artificial neural network has an output that is a combination of all those activation functions by which every layer is composed, starting from the input layer towards the output layer. This is also known as *forward-propagation*.

2.2.3 Inference and Training

The best thing of a neural network is its straightforward implementation; usually, if a machine has to do thousand of things, as normally a human brain does, about the same number of different programs and algorithms have to be implemented to perform all those applications. However, in a NN, one single learning program is enough to execute the required tasks. It perfectly behaves like a brain and learns by itself how to process all the input data. This means that the program is always the same, even if the machine learns how to do thousand of new things.

There are two main processes to realize a NN:

1. *Training.*

2. *Inference.*

In the image recognition example, the machine can be taught with a training data set. Once it is ready, it can receive new input data, never analyzed before, and classify the object. In Figure 2.6 there is the comparison between the two processes.

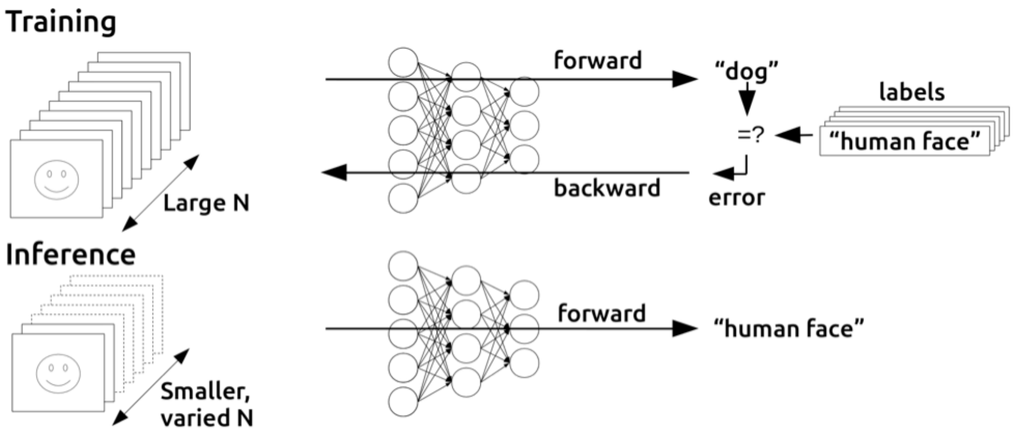


Figure 2.6: Comparison between training and inference. [24]

First of all, to *learn*, a machine has to determine the value of some important parameters, like weights and bias. This process is referred to as *training* of a network. In particular, the machine can be efficiently trained with the gradient descent algorithm [26].

To train the parameters w and b, a cost function is defined: it measures how well the algorithm is working on an entire training set. Then, the gradient descent algorithm runs iteratively and finds the value of the weights and of the bias that minimizes the cost function. In every iteration, the value of the weights are updated:

$$\begin{aligned} W &:= W - \alpha \cdot dW \\ b &:= b - \alpha \cdot db \end{aligned} \tag{2.8}$$

where α is the learning rate parameter (default value at 0.1), one of the hyper-parameters of a neural network. Its value determines how fast the model learns [3].

The parameters are updated with a value determined by the partial derivatives. Their value is evaluated with the *back-propagation* algorithm [26]. After all this process, the parameters are computed and the neural network is ready to receive new inputs and performs many tasks.

When the programs run with those defined optimal parameters, the process is called *inference*. In this case, the machine, starting from input data, performs a *forward-propagation* and evaluates the output, exploiting the trained weights. The forward-propagation is not so different from the back-propagation used for the training. Therefore, the techniques used for the inference can be useful to improve the training.

However, there main difference among the inference and training is the much more storage required by the training process. Indeed, with the back-propagation algorithm, the training needs the transitional outputs of the NN in order to make all the computations. Therefore, all the memory optimizations are welcome but they, of course, will have much more impact in the training.

2.3 Deep Neural Network

After looking at the basic blocks of a simple neural network with a single or few hidden layers, a *Deep Neural Network* can be now analyzed. All the characteristics of the previous chapters are still valid, the main difference is the number of hidden layers: in this case, it is really high. The direct consequence is the greater number of functions that the machine can learn.

Indeed, there are tons of neurons that now can be connected during the inference and with this great amount of activation functions it possible to extract a lot of high-level features. Starting from low-level ones, they are then recombined on the next layers to form higher-level features. In the end, all those information are useful to observe a particular and important aspect of the function [1]. Without DNNs, such performances cannot be achieved.

Today, in Deep Learning the number of network layers stays between five and a thousand. The hidden layer number has become a fundamental hyperparameter for the machine. These hyperparameters are important to characterize the learning algorithm [3]. Some of them are:

- The number of hidden layers, as just seen.
- The learning rate. It has been analyzed in the chapter 2.2.2; it is useful for the back-propagation in the training phase and determines how fast the main parameters change.
- The amount of iterations in the gradient descent algorithm.
- The number of neurons for each layer of the network.
- The activation function, seen in the chapter 2.2.2.

There are many other hyperparameters, depending on how complex is it the network, but those are the most important. They are so-called because they determine the value of the real parameters, such as weight and bias.

Nowadays, there are many applications for DL, like speech recognition, computer vision, and robotic tasks. The value of hyperparameters may change for any of them, there is not a universal solution that is always valid. Moreover, with the technological progress and innovation, the hyperparameters that today are perfectly tuned may change their optimal value in the future. Therefore, some form of flexibility in their use is required to avoid any problem and to adapt to the new technologies and applications.

Today, the development of DNNs is very fast thanks also to the Deep Learning frameworks available on the web [1]. Those are open-source libraries that can be exploited to realize a neural network without starting from scratch. The main frameworks are:

- *Caffe*, that supports MATLAB, C++, C and python;
- *Tensorflow*, that supports python and C++;
- *Torch* and *PyTorch*: the first supports Lua, C and C++; the latter is the evolution of Torch and it uses python.
- *Theano* and *Keras*, both support python.

The availability of such frameworks is beneficial for researchers, designers but also for hardware engineers. Moreover, they can exploit optimized software or hardware accelerators. Besides, there are some data sets, useful for image classification and computer vision applications, that are available. In this way, it is possible to estimate the accuracy of a DNN and to compare with other models and approaches.

2.4 Convolutional Neural Network

One of the most popular algorithms of a Deep Neural Network is the CNN (Convolutional Neural Network), also called ConvNet.

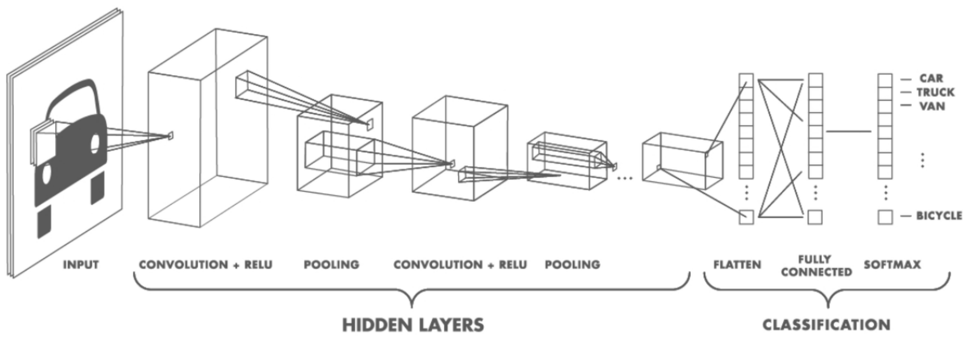


Figure 2.7: Architecture of a Convolutional Neural Network. [19]

In Figure 2.7 there is an example of CNN architecture and, as can be seen, it is generally made by these layers [19]:

- The Convolution Layer (CONV).
- The ReLu Layer (RELU), seen in the chapter 2.2.2.
- The Pooling Layer (POOL).
- The Fully Connected Layer (FC).
- The Softmax Layer, for classification problems. It is a generalized version of the logistic regression and it is used when the task is the classification of various classes, not just binary.

It is really used in computer vision problems, such as image classification, also called image recognition, and object detection. When the input of a computer vision is a colored image, it can get really big: indeed, three color channels are needed and their dimension become too large to deal with. Starting from a 1000x1000 pixel image, it is a megapixel; when colored it is 1000x1000x3, because of the RGB channels. Thus, the dimension of the input features is three million; if the first hidden layer has just 1000 units, the dimension of the weight matrix would be too big for a standard or Fully Connected Network. With that many parameters, it is difficult to prevent a neural network from overfitting [3]. Besides, computational and memory problems may arise in the training process of the machine. The convolution operation solves many of these problems, that is why the CNN is so popular.

In CNN there are more hyperparameters to take into account, such as:

- The size of the filter.
- The *padding*.
- The *stride*, to perform stridden convolutions.
- The *depth*, it represents the number of filters used in a certain stage of the network.
- The *channels*.

2.4.1 Convolution layer

The convolution is made between the input image and a filter, also called *kernel*. It is used because this operation may be useful to find some important features: for instance, it can be used as an edge detector. If an image has multiple channels, the filter has the same depth.

The *stride* is a fundamental hyperparameter in CNNs since it determines how the kernel convolves around the input feature map [1]. This hyperparameter defines the amount of cells by which the kernel has to shift in each iteration. An example is in Figure 2.8.

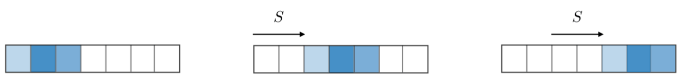


Figure 2.8: Slide of a vector with stride = 2. [1]

The stride is chosen in such a way that the output is an integer and not a fraction; if that is not the case, the output is rounded to the nearest integer. If the stride is greater than one, a shrink of the size of the feature map occurs [3].

As regards *padding*, it is another basic block of a Deep Neural Network. Choosing its value determines the dimension of the output matrix and it is critical in CNNs: indeed, after many steps of convolutions, the output feature map could get really small. In this case, detecting a new feature by using another kernel is very difficult. After few convolution steps, any information of the original input would be lost.

Besides, there is also a huge loss of information due to the pixels on the edge and the corner of the matrix: those values are just used once, and surely much less than the pixels in the middle, and thus they are not so relevant in the final computation of the output.

The padding can fix both of these problems, just by using a pad image.

There are mainly two choices for padding [3]:

- *Valid convolution*: no padding is applied ($p = 0$), so still there is a shrink of the image and thus of information.
- *Same convolution*: the input feature map is surrounded by zeros in all the borders (or some of them). The padding is done in such a way that the output dimension matches the input one ($p > 0$).

Usually, the size of the filter is odd, so that asymmetric padding is not required. In Figure 2.9 there is an example of *Same padding convolution* with $p > 0$.

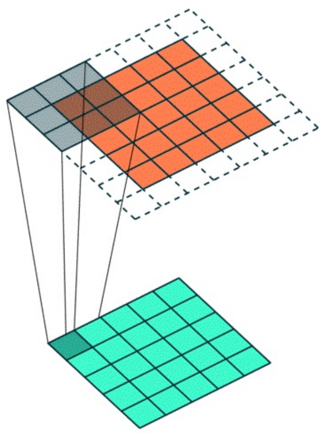


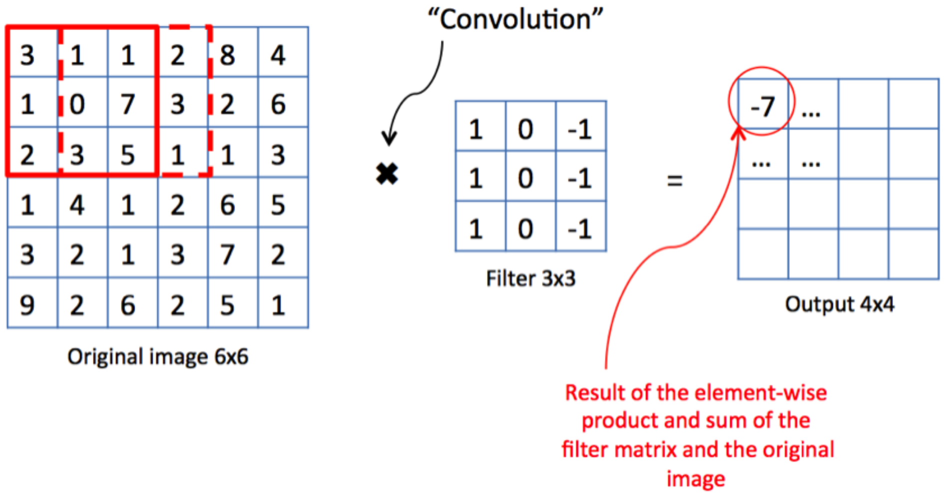
Figure 2.9: Convolution with padding = 1. [19]

As regards the convolution operation, an example is shown in Figure 2.10.

The operation is made between the 6x6 original input image and a 3x3 filter. An element-wise matrix multiplication is performed between the kernel and a 3x3 portion of the digital image; the sum of the result is saved in one element of the output matrix. Of course, to perform this operation, the size of the two matrices must match. Therefore, for a complete convolution operation, the kernel must shift, in this case, sixteen times. The output is a 4x4 matrix [20].

The general formula, for evaluating the dimension of the final output, is the following [3]:

$$O = \frac{(W - f + 2p)}{s} + 1 \tag{2.9}$$



"Convolution"
Original image 6x6
Filter 3x3
Output 4x4
Result of the element-wise product and sum of the filter matrix and the original image

Figure 2.10: Convolutional operation in detail. [20]

where:

- O is the output size (height/weight);
- W is the input size (height/weight);
- f is the filter size;
- p is the padding;
- s is the stride.

2.4.2 Pooling Layer

The Pooling Layer is largely used in a CNN to reduce the spatial dimensions (but not the number of channels) of the convolved feature map. This operation is really important since it allows to reduce the computational power and thus increases efficiency. Moreover, it reduces the overfitting problem and simplifies the extraction of important features. There are mainly two types of pooling [3]:

- Max pooling.
- Average pooling

In Figure 2.11 there is an example. In this case, starting from a 4x4 image, a kernel of size 2x2, with stride 2, is used to perform the pooling. It divides the input in different regions of the kernel size. The Max pooling returns the maximum

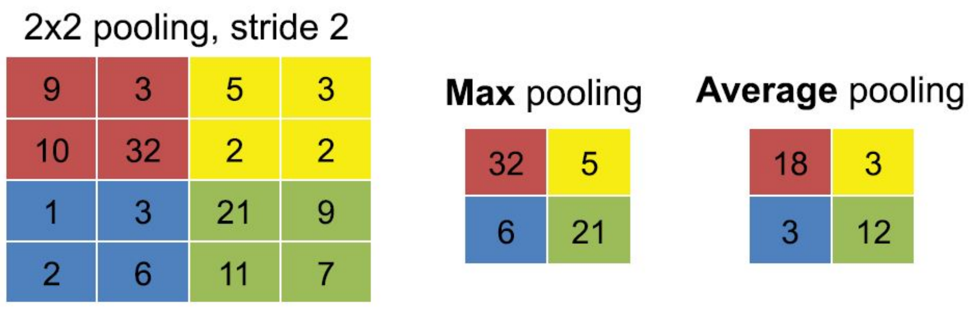


Figure 2.11: Various forms of pooling. [1]

value from every region. The Average pooling, instead, takes the average of all the values and returns it in the output matrix. As can be seen, the pooling performs a downsampling of the feature map, reducing its height and width. The depth of the input matrix is still the same [1]. Besides, the pooling operation does not increase the number of parameters to be learned by the machine: it is a fixed computation [3].

2.4.3 Fully Connected Layer

The last layer of a CNN is the Fully Connected one. It is a standard neural network composed of a certain number of hidden layers. So, it is a feed-forward neural network and the learning process is done with the back-propagation algorithm, as seen in the chapter 2.2.3. It is a cheap way to learn non-linear functions.

The input of this layer is the output of a CONV or POOL layer, but it must be a vector; since that output is a 3D matrix, it has to be flattened before entering in the final layer [3]. An example of Fully Connected Layer is shown in Figure 2.12.

In the FC layer, all the neurons of two different layers are connected together. This means that the output is composed of a weighted sum of the input activations. This can be very expensive in terms of computation and memory occupation. However, there are also the so-called *sparsely-connected* layer, in which not all the neurons are connected. Indeed, some weights are set to zero and there is no effect on accuracy [1].

Moreover, there is a technique in which only a limited number of weights contribute to the computation of the output. A particular case is the *weight sharing* approach, in which always the same set of weights contribute to the evaluation of the outputs [1]. Having these kind of structured sparsity can increase the performances of the network.

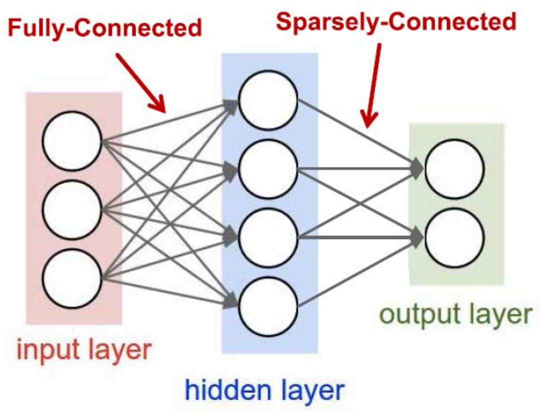


Figure 2.12: Fully connected versus sparse. [1]

2.4.4 1x1 Convolution

In complex architectures, the 1x1 Convolution block is heavily used. If the input image is, for instance, a single channel 6x6 matrix and it is multiplied by 1x1 filter of a certain value, the final result will be just a multiplication of the input for that value. However, this operation becomes useful when the input image has multiple channels.

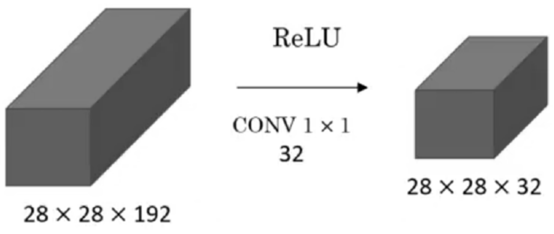


Figure 2.13: Example of 1x1 convolution. [3]

In the example in Figure 2.13, the input is a 28x28x192 volume. If the size of height and weight is too high, a Pooling Layer can be used to reduce their dimension. If instead, the aim is to shrink the number of channels, the 1x1 convolution is the best solution. The output depth dimension will be set by choosing the number of filters to be applied to the input [3].

This operation is also called *Network in network* and it is used in many DNN models, such as GoogLeNet architecture [7].

2.5 Models of DNN

The layers just described in the chapter 2.4 are the starting point for the creation of a complete neural network. Many CNN models have been created and understanding some of them is a good starting point to realize a new flexible network that works well for a certain task.

Some of these models are analyzed in this section, such as:

- *LeNet* [4]: it is one of the first CNNs, realized in 1989. Its purpose is digit classification of grayscale images, indeed the inputs have only one channel. The architecture of the LeNet-5 is shown in Figure 2.14.

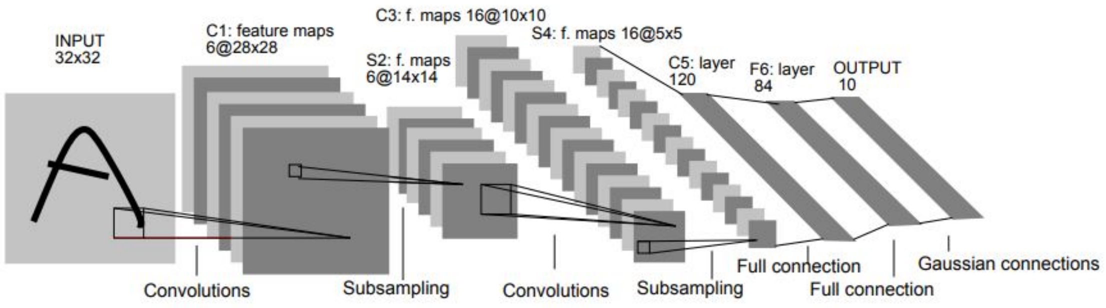


Figure 2.14: LeNet architecture. [4]

  The LeNet-5 has two CONV layers and, after both of them, there are 2x2 POOL layers. In the end, there are two FC layers and the Softmax layer for the classification. Since it is a really old structure, average pooling is used; moreover, the activation function is the sigmoid one. However, more modern and complex designs share these types of layers: alternation of convolution and pooling, fully-connected at the end.

- *AlexNet* [5]: it is similar to the LeNet architecture but much bigger. Its name stems from one of its author, Alex Krizhevsky, who also wrote the paper of this design. This architecture won the ImageNet contest in 2012 and it is a milestone for computer vision applications.

  As shown in Figure 2.15, the input is a 3D image and the network is composed of five CONV layers and three max-pooling layers. It is important to notice that at the first step, there is a high value of stride, equal to four, that shrinks the input image and reduces the computation. Moreover, ReLU nonlinearity is used in this design as an active function. In the end, there are three FC layers, then the classification is performed.

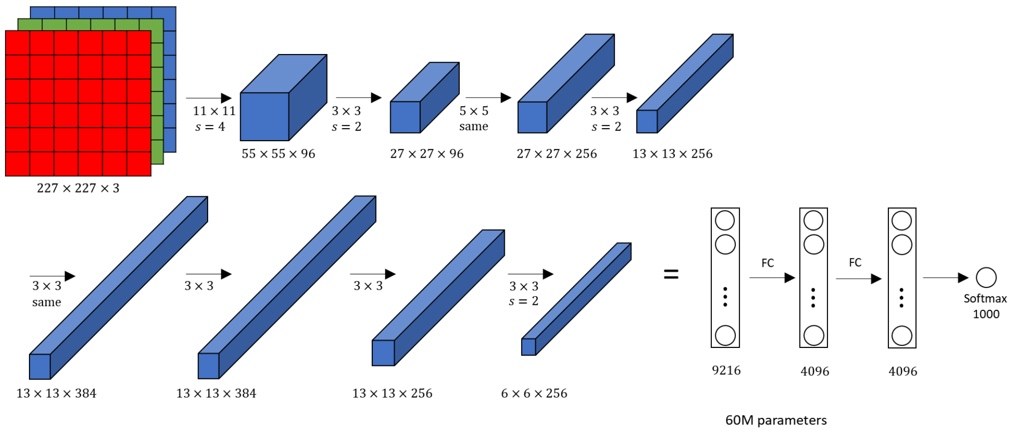


Figure 2.15: AlexNet architecture. [5]

- *VGG-16* [6]: this structure has a greater number of layers with respect to the first two solutions: sixteen layers, as can be seen in Figure 2.16. It has been realized in 2014 by K. Simonyan and A. Zisserman. It has a large amount of total parameters, even for today standards, and it is hard to train all of them.

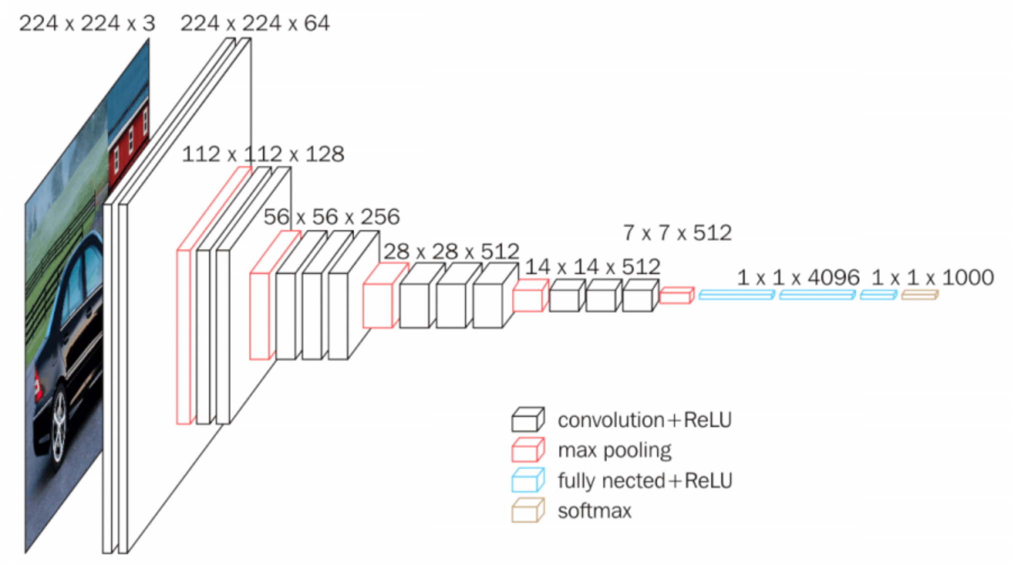


Figure 2.16: VGG-16 architecture. [6]

The reason why this model is so important is because, a few years ago, the cost of going deeper was increasing too much. The two authors proposed a new solution to realize 5x5 filters with a composition of smaller ones, containing fewer weights.

Therefore, even if this structure is really big, it is affordable and quite attractive because of its simplicity: there is an optimization in terms of hyperparameters. Indeed, all the CONV layers have the same stride, equal to one, the same 3x3 filters and the same padding as well. Moreover, the POOL layers, that are five in this model, are of the maximum type and use a 2x2 kernel with the stride equal to two. In the end, there are three fully-connected layers.

- *GoogLeNet* [7]: this structure has been proposed in 2014 and has 22 layers, greater than the VGG-16. This design is worth considering because of the *inception module*, represented in Figure 2.17.

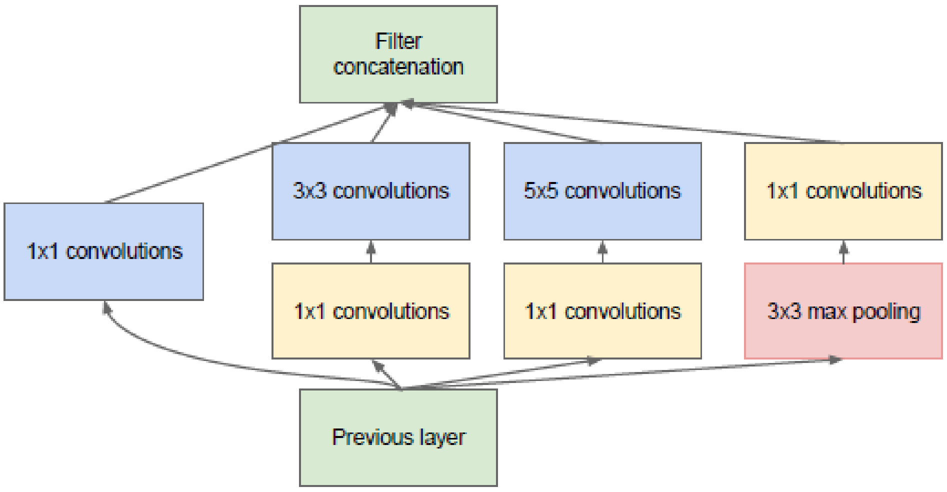


Figure 2.17: Structure of the inception module.

This structure reduces the number of parameters thanks to the parallel connections: there is the 3x3 max-pooling and also the convolutions with filter sizes 1x1, 3x3 and 5x5. At the output, all these modules are concatenated. In this way, the computation cost is reduced by one order of magnitude.

The total structure of the GoogleNet is in Figure 2.18.

The architecture is composed of nine inception layers, three CONV layers, and a final FC layer.

- *ResNet* [8]: also called Residual network, it realized in 2015. It took part in the ImageNet contest and it has been the first machine to overcome the human-level accuracy. This network exploits the *residual blocks*, that allows to do a skip connection (a *shortcut*): the activation, instead of being computed in a certain layer, can be postponed in a deeper layer in the network. This block is shown in Figure 2.19.

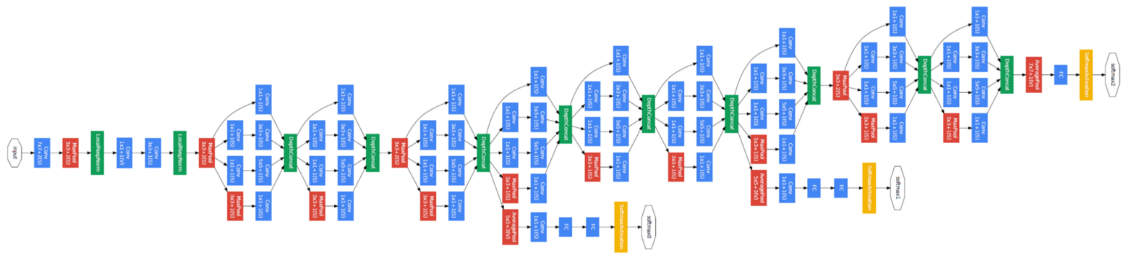


Figure 2.18: GoogLeNet architecture. [7]

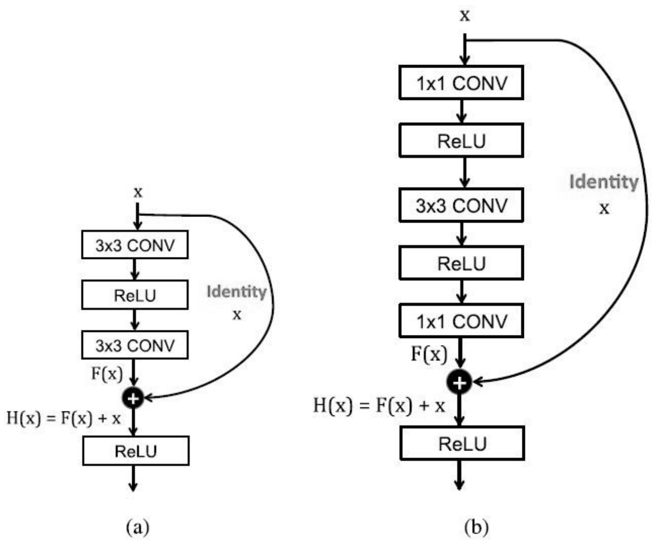


Figure 2.19: Shortcut module from ResNet. (a) Without bottleneck. (b) With bottleneck. [1]

Moreover, to reduce the number of weights, the *bottleneck* approach is used. In this case, instead of two CONV layers in the shortcut module, there are three layers (1x1, 3x3, 1x1).

The ResNet architecture is composed of one CONV layer, 16 residual blocks and then one FC layer.

In the Figure 2.20, a comparison between all the models just analyzed is shown. It is a diagram representing the results of the ImageNet Challenge from 2010 up to 2015. The best architecture, able to classify perfectly the images, even better than the human level, is the ResNet one.

Another important aspect to take into account is the increase of accuracy, between 2011 and 2012, thanks to one of the first deep learning algorithm used by AlexNet.

In Table 2.1, there are some interesting data to analyze the different DNN models.

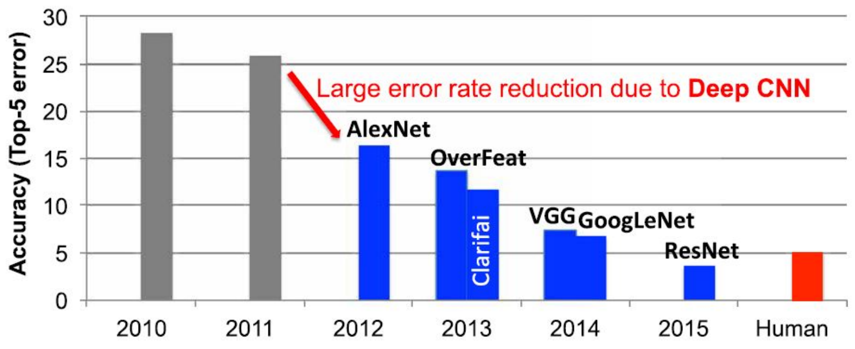


Figure 2.20: Results from the ImageNet Challenge. [1]

Table 2.1: Comparison between DNN models.

	LeNet 5	AlexNet	VGG-16	GoogLeNet	ResNet 50
Input Size	28x28	227x227	224x224	224x224	224x224
# of CONV layers	2	5	13	57	53
Filter sizes	2	5	13	57	53
Weights	2.6k	2.3M	14.7M	6.0M	23.5M
# of FC layers	2	3	3	1	1
Filter sizes	2	5	13	57	53
Weights	58k	58.6M	124M	1M	2M
Total weights	60k	61M	138M	7M	25.5M

First of all, the size of the filters changes on every layer, even in modern architectures. Therefore, the flexibility remains a priority [1].

As regards the number of total weights, the most modern DNNs are really deep networks and can afford many of them. In this way, they can provide a large amount of nonlinear function for the learning algorithm and can detect lots of features.

Moreover, in recent years the number and the importance of convolution layers is constantly increasing; it is the opposite for the FC layers. Therefore, nowadays the optimizations are made to improve the CONV operation [1].

Chapter 3

ASIP Designer

ASIP Designer is a Synopsys tool used for designing custom processors or programmable accelerators for different applications [9]. It is language-based and, starting from the description of the processor as input, it provides a Software Development Kit (SDK), that will be analyzed in the next paragraphs. Moreover, it is possible to make fast changes in the description of the processor model: it is quite simple to optimize the design for achieving a certain requirement.

The tool can be used for many applications, such as image and vision processors, wireless modem and medical devices: all systems where there are lots of signal processing but also strict low power requirements [9].

3.1 Design approach

In this section, three kinds of processor design will be analyzed. Starting from a pure hardware (HW) solution, up to a pure software (SW) approach, all the advantages and disadvantages of those will be investigated. In the end, the ASIP approach will be discussed. ASIP is the acronym of Application Specific Instruction-Set Processor and it is the starting point for the use of the Synopsys's tool ASIP Designer.

In Figure 3.1, there is the comparison between the three approaches in terms of efficiency (power, area and performance) and flexibility.

3.1.1 Hardware approach

The general hardware approach is the ASIC (Application Specific Integrated Circuit) design. It is a hardware solution in which the algorithms are described at RTL level, then made in silicon. ASIC is the best solution in terms of efficiency, power consumption and area occupation since it is a custom system made specifically for a certain function. However, it has really high complexity and also quite high design time; therefore, the realization cost is extremely high.

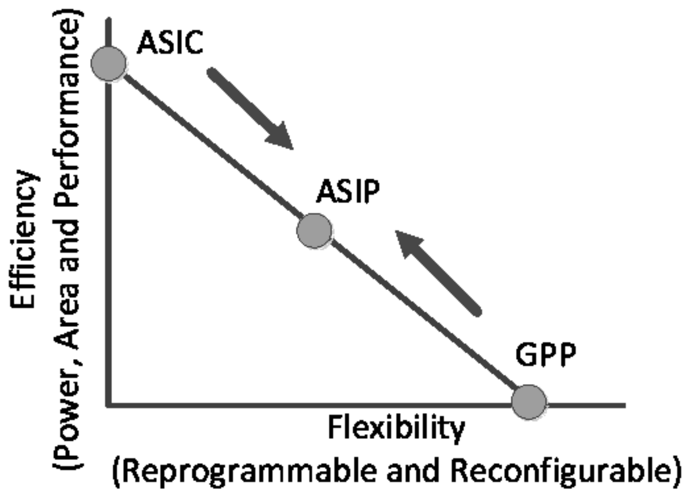

ASIC
ASIP
GPP
Efficiency
(Power, Area and Performance)
Flexibility
(Reprogrammable and Reconfigurable)

Figure 3.1: Test application structure

Another huge disadvantage is the lack of flexibility: once the custom function is realized, it is not feasible to change the algorithm; it would be necessary to design a new whole integrated circuit.

Therefore, the ASIC system is not always a possible solution: it is used only when there are few constraints for the specific applications and when there is a massive production that limit its costs.

3.1.2 Software approach

The general purpose microprocessor solution is the opposite of the ASIC approach. In this case, the design is not made for a single custom function but it is suitable for a large number of applications. It is a pure software solution in which the algorithms are implemented at a high level, in languages like C or Assembly.

There are no optimizations at hardware level: the design of this solution is not as expensive as the hardware one. This means also that the great advantage of the general-purpose system is the flexibility: the algorithm can be modified many times, without extra costs. However, this leads to lower performances, greater power consumption, and larger area occupation.

3.1.3 ASIP approach

The ASIP approach is the midway solution between the software and hardware ones. In this case, the designer is free to choose what to implement in hardware and what in software. This is important because sometimes the system has to be more efficient than a general-purpose processor, but still more flexible than the ASIC solution. Therefore, the idea is to start from a general-purpose processor and then modify it, adding custom instructions and removing the unused ones.

Besides, a key part of the implementation of an ASIP is to map the most complicated instructions to the hardware modules, while the less time-consuming functions to the software parts. In this way, the system can achieve much better performance.

3.2 ASIP Designer tool

ASIP Designer is a brand new Synopsys tool that allows the design of an ASIP system. In particular, it exploits the high-level hardware description language *nML* to describe the architecture of the processor [9].

The nML code defines the instruction set of the architecture and describes its instruction pipeline. Besides, it shows the bit-accurate behavior of the operations of the processor [9]. Since all the tools of ASIP Designer use the nML code, there is a full compatibility among the hardware implementation and the Software Development Kit (SDK).

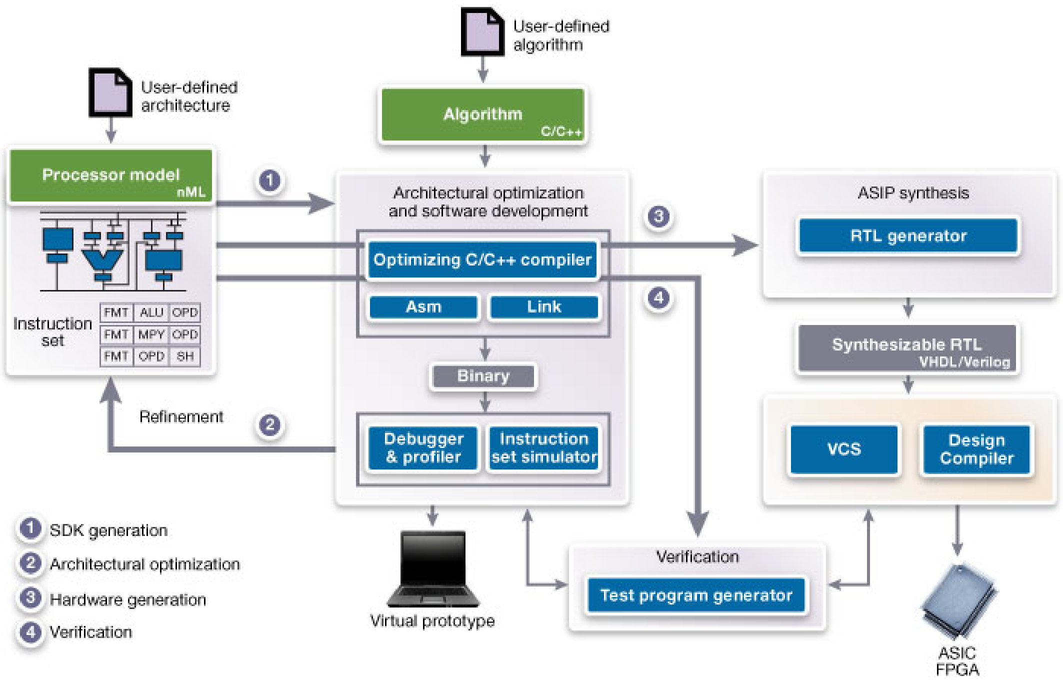


Figure 3.2: Test application structure. [9]

The aim of ASIP Designer is to describes ASIPs and hardware accelerators. Both share the most common techniques that accomplish high performance and low power: for instance, they exploit a "heavy use of specialized datapath elements and parallelism" [9]. These architectures are C/C++ processors and accelerators that exploit the software programmability of their applications.

3.2.1 ASIP features

ASIP Designer's technology supports many features [9], some of them are:

- It describes and modifies the ASIP architectures in the nML language.
- It has a unique Compiler-In-The-Loop technology that generates the SDK, which contains the following components:
  - An optimized compiler that has high-level code optimization and automatically adapts to the processor architecture. The compiler can support a wide range of them: from general-purpose processors to highly specialized ASIPs. Moreover, it supports these programming languages: C (that can be expanded with C++ classes and functions and exploit other data types), C++ and OpenCL C (that is the OpenCL kernel language).
  - A Linker that creates executable files from various object files.
  - An Assembler and Disassembler that converts the machine code from binary format to assembly and vice-versa. The assembly language is specified in the nML model of the processor.
  - An Instruction-Set Simulator (ISS) that gives both fast cycle-accurate and fast instruction-accurate simulations, using just-in-time compilation techniques. Both models are generated from the nML code of the architecture.
  - A Debugger that can be adopted simultaneously for the on-chip debugging (using JTAG) and for the instruction-set simulators.
- It has an RTL hardware generator that automatically translates the nML description of the processor into a synthesizable Verilog or VHDL code. The tool that performs this operation is called Go [13]. The HW implementation of the ASIP is efficient in terms of power consumption and area.
- It provides a support to verify the ASIP designs, such as some automatic test programs, created to analyze and diagnose certain ASIP's ability. Those are in C and assembly language.

Chapter 4

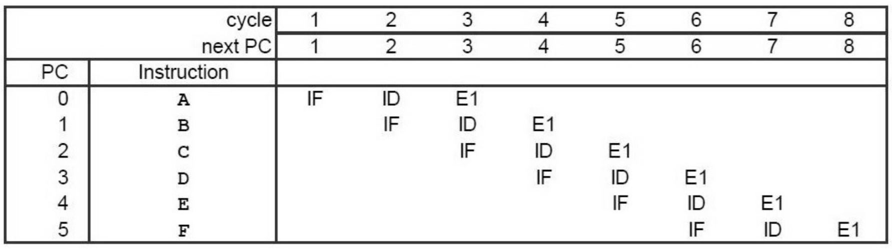
ASIP Implementation

The Synopsys tool ASIP Designer provides a large number of processors that can be used for any application. In this work, the *Tvec core* is used.

4.1 Tvec Processor

The Tvec processor is a more sophisticated version of the *Tmicro core*, also provided by the tool. The latter is a 16-bit microcontroller that is exploited by the software to show many concepts of processor modeling and tool capabilities, such as chip debugging and processor verification [10]. Moreover, it can be used as a basic element that may be developed to realize an application specific instruction-set processor.

Tmicro executes all the instructions in a pipelined architecture composed of three stages, as shown in Figure 4.1:



	cycle	1	2	3	4	5	6	7	8
	next PC	1	2	3	4	5	6	7	8
PC	Instruction								
0	A	IF	ID	E1					
1	B		IF	ID	E1				
2	C			IF	ID	E1			
3	D				IF	ID	E1		
4	E					IF	ID	E1	
5	F						IF	ID	E1

Figure 4.1: The instruction pipeline. [10]

1. Instruction Fetch (IF): in this stage, there is the fetch of a new instruction, taken from the program memory.

2. Instruction Decode (ID): the instruction, previously fetched, is now decoded.

3. Execute 1 (E1): during this stage, all the mathematical operations are performed.

The instructions of this processor are 16 bit wide and the core allows multi-word multi-cycle instructions [10]. The main features of the Tmicro architecture are:

- 16 bit ALU, that allows to perform different operations, such as integer arithmetic, bitwise logical and compare instructions. The data path of the ALU is shown in Figure 4.2.

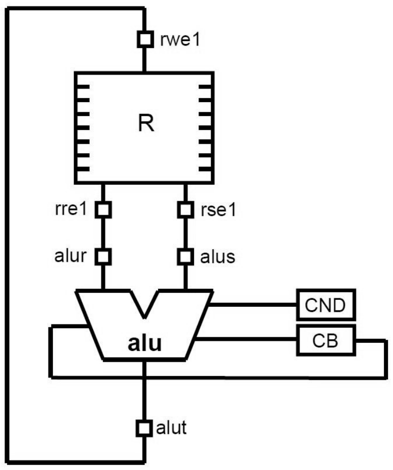


Figure 4.2: The ALU data path. [10]

- 16 bit shift unit. It supports the arithmetic shift right and the logical shift right and left. The shifter data path is in Figure 4.3.

- Register file with 8 registers.

- Register move instructions.

- Multiplier-Accumulator (MAC) unit.

- Many control instructions: jumps, subroutine calls and returns. It supports interrupts as well.

- 16 bit integer multiplier: 16-bit operands and the output on 32 bit.

- 16 bit division.

Besides, there are load and store instructions, also with indirect addressing. The Tmicro architecture provides two different memories: the Data Memory (DM) and

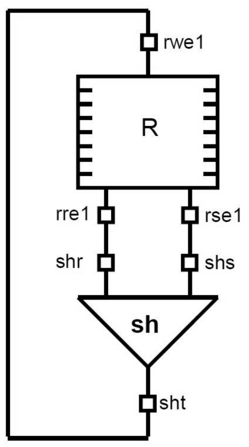


Figure 4.3: The shifter data path. [10]

the Program Memory (PM).

Those are just some of the default characteristics. The user can modify, add or remove any number of instructions to realize an application specific instruction-set. Indeed, this is exactly what it is done to realize the Tvec core, used during this work.

The main addition to the Tmicro core is the SIMD kind of operations. It stands for "Single Instruction Streams, Multiple Data Streams" [3]. The implementation of those types of instructions, also known as *vector instructions*, allows to operate with all the elements of a vector simultaneously. In this way, a faster and more efficient computation is given, when dealing with large amounts of data.

In the default configuration of the Tvec core, the vector type is 128 bit wide [11]. In particular, it contains 8 times the width of the 16-bit type *word*. Moreover, to perform SIMD instructions, it has been added, to the Tmicro core, a vector data-path, a vector register and a vector data memory (DMv). In fact, the new types of data are stored in the vector registers. In Figure 4.4, there is an example of a SIMD program.

Two vectors are loaded, at the same time, from the DM to the vector registers. Then, an element-wise addition is performed and the result is stored back in the memory. The addition is not the only allowed operation between two vectors. It is possible to perform:

- Element-wise bitwise vector operations, such as OR, AND, XOR and complement.
- Element-wise addition and subtraction of vectors;

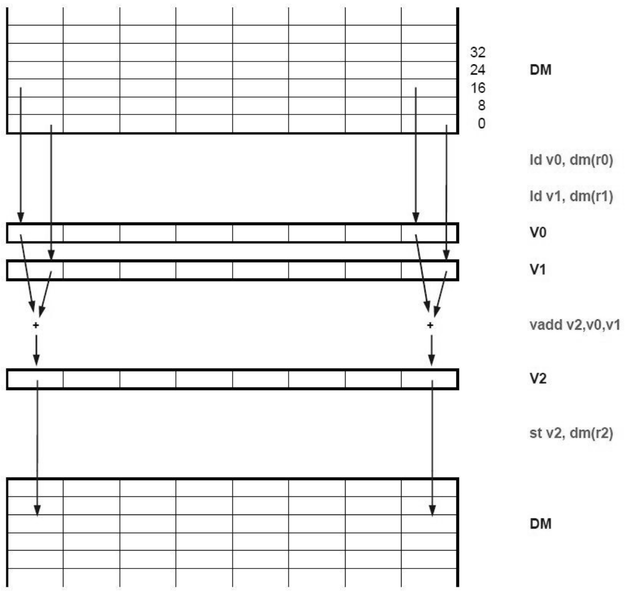


Figure 4.4: Execution of vector instructions. [11]

- Element-wise computation of maximum and minimum.
- Inter-element vector operations.

The data path of the vector unit, that computes all the previous operations, is shown in Figure 4.5.

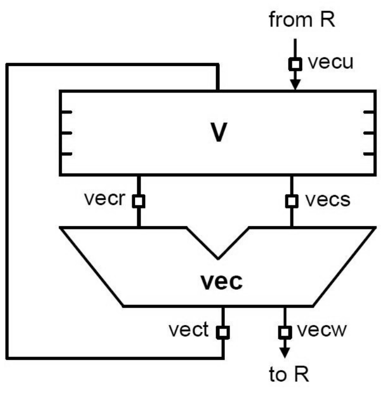


Figure 4.5: The data path of the vector unit. [11]

In Figure 4.5, *vecr* and *vecs* are the input transitories, while *vect* is the output computation transitory. Those transitories are used to connect the functional unit, called *vec*, to the vector registers.

There are two other transitories, that are *vecu* and *vecw*. The first is used when the operation has a scalar operand; the second when the computed output is a scalar [11].

4.2 Software Implementation

Once decided which processor core to use for the CNN, the next primary step is the description of a simple bi-dimensional neural network in a mixture of C/C++ code. This software implementation is parametric, therefore it is possible to modify every aspect of the network easily. Besides, the code covers only the inference process and not the training.

Eventually, this code will be executed in the Synopsys tool ASIP Designer. Since the Tvec core, used in this work, does not support many of the C and C++ libraries, the code is written in a simple way to avoid compiling problems. For this reason, the available frameworks, discussed in chapter 2.5, are not taken into account.

4.2.1 Fully-Connected Layer

First of all, a Fully-Connected layer function is created: it receives as input the matrix X and the weight matrix W. The other two inputs are the number of input and output neurons. With the latter values, it is possible to create a single FC layer having a variable number of neurons.

```c
void Fully_Connected_Layer(Image A[], int B[][SIZE], int D[][SIZE], int channel, int
    input_neuron, int output_neuron)
{
    int product = 0, final_result = 0, a=0, b=0;
    for(int i=0; i<output_neuron; i++)
    {
        for(int j=0; j<input_neuron; j++)
        {
            for(int k=0; k<channel; k++)
                product += dot_product((vint*)A[k].image[j], (vint*)B[j]);
        }
        final_result = relu_gradient(product+1);
        D[i][b] = final_result ; /
        final_result = 0;
        product = 0;
        b++;
    }
}
```

Listing 4.1: Fully Connected Layer code.

Each row of the matrix W contains the weights of a single neuron, that are multiplied by the input image afterwards. In particular, there is a dot product between the two; the bias is added to the output and then the ReLU activation function is performed. The final result is stored in an output matrix, whose size depends on the number of neurons.

It is possible to recall this function many times, in order to create as many hidden layers as the user desires.

4.2.2 Convolution Layer

The convolution layer is the most important part of the code. Indeed, the aim is to create a Convolutional Neural Network and the CONV layer is the most used layer in the whole network. This is the reason why multiple implementations of the same layer will be shown: since it is the layer that requires the largest computational unit, all the optimizations are done to improve this function.

The convolution function receives the input image and the filter. Moreover, there are a lot of parameters and hyperparameters that are necessary for a complete operation:

- Stride: by setting this value, it is possible to change the value of the stride and decide how to shrink the image.
- Padding: it is possible to choose between Same padding and Value padding.
- Kernel size: the CONV function supports 3x3 and 4x4 filters; it is possible to choose one of the two with a flag.
- Image size: it is possible to receive an image of any dimension.

The CONV function returns an integer value that represents the size of the output feature map, evaluated with the formula:

$$O = \frac{(W - f + 2p)}{s} + 1 \tag{4.1}$$

```
// Regular convolution function.
int Convolution(int conv_size,int kernel,int stride,int padding, Image &A, Image &B, Image &D)
{
    int convolute = 0, result, final_result;
    int x, y, a=0, b=0, z=0;
    int output = (conv_size - kernel + 2*padding)/stride + 1;
    int pad_size = conv_size+2;
    int temp_size = kernel*kernel;

    if(padding == 1)
    {
        int Pad[SIZE+2][SIZE+2];
        for(int i=0; i<pad_size; i++)
        {
            Pad[i][0]=0;
            Pad[0][i]=0;
            Pad[i][pad_size-1]=0;
            Pad[pad_size-1][i]=0;
        }
        for(int i=1; i<pad_size-1; i++)
        {
            for(int j=1; j<pad_size-1; j++)
                Pad[i][j]=A.getValue(i-1,j-1); //A[i-1][j-1];
        }

        for (int j = 0; j < stride*output; j+=stride)
        {
            for (int i = 0; i < stride*output; i+=stride)
            {
                x = i;
                y = j;
                for (int k = 0; k < kernel; k++)
                {
                    for (int l = 0; l < kernel; l++)
                    {
```

```
                    convolute += Pad[x][y] * B.getValue(k,l);
                        y++;
                    }
                    x++;
                    y = j;
                }
                final_result = relu_gradient(convolute);
                D.setValues(final_result , b, a);
                convolute = 0;
                b++;
            }
            b=0;
            a++;
        }
    }
    if(padding == 0)
    {
        for (int j = 0; j < stride*output; j+=stride)
        {
            for (int i = 0; i < stride*output; i+=stride)
            {
                x = i;
                y = j;
                for (int k = 0; k < kernel; k++)
                {
                    for (int l = 0; l < kernel; l++)
                    {
                convolute += A.getValue(x,y) * B.getValue(k,l);
                        y++;
                    }
                    x++;
                    y = j;
                }
                final_result = relu_gradient(convolute);
                D.setValues(final_result , b, a);
                convolute = 0;
                b++;
            }
            b=0;
            a++;
        }
    }
  return output;
}
```

Listing 4.2: Convolution Layer code.

4.2.3 Pooling Layer

The pooling layer is another important block of the network and it receives the convolved feature map from the CONV layer. Besides, there are some parameters to exploit also in this case:

- Stride.
- Image size.
- Kernel size.

To support the pooling function, two operations are defined: *findMean* and *findMax*. Both receive a 2x2 portion of the input feature map. In the first case, the average pooling is computed; in the latter, the maximum value is extracted. Next, The output feature map is stored in a new matrix, ready to be sent in the next layer.

In the end, also the pooling function returns the integer value of the output size of the matrix. In this case, it is evaluated with the formula 4.2:

$$O = \frac{W}{s} \tag{4.2}$$

```
int findMax(int mat[][N])
{
    int maxElement = 0;
    for (int i = 0; i < N; i++)
    {
        for (int j = 0; j < N; j++)
        {
            if (mat[i][j] > maxElement)
                maxElement = mat[i][j];
        }
    }
    return maxElement;
}

int findMean(int mat[][N])
{
    int sum = 0;
    for (int i=0; i<N; i++)
       for (int j=0; j<N; j++)
          sum += mata[i][j];
    return sum/(N*N);
}

// Pooling function.
int Pooling(Image &pool_H, Image &pool_D, int size, int kernel, int stride)
{
    int H_temp[2][2];
    int x, y, max, a=0, b=0;
    int output = size/stride;
  for (int i = 0; i < stride*output; i+=stride)
  {
    for (int j = 0; j < stride*output; j+=stride)
    {
      x = i;
      y = j;
      for (int k = 0; k < kernel; k++)
      {
        for (int l = 0; l < kernel; l++)
        {
                    H_temp[k][l]=pool_H.getValue(x,y);
          y++;
        }
        max = findMax(H_temp);
        x++;
        y = j;
      }
      pool_D.setValues(max, a, b);
      max = 0;
            b++;
    }
    b=0;
    a++;
  }
  return output;
}
```

Listing 4.3: Pooling Layer code.

In the code 4.3, there are also the functions *findMax* and *findMean*. In the simulated CNN, the *findMax* function is used.

4.2.4 Activation Function

The proposed CNN supports four different activation functions:

- ReLU gradient.

- Leaky ReLU.
- Hyperbolic tangent.
- Sigmoid function.

All these functions are described in the chapter 2.2.2. Having those different types of activation gives more flexibility to the neural network.

4.3 3D Convolutional Neural Network

In the next step of the CNN implementation, two other hyperparameters are added to the structure:

- Number of channels.
- Number of filters.

By using the two parameters above, it is possible to realize a 3D Convolutional Neural Network. First of all, an *Image* class is defined; it is used to increase the depth of the feature maps and of the kernels. The input image can now be colored, having the three RGB channels. The number of channels of the filter and of the image must match, otherwise, the convolution operation is not feasible.

Moreover, by choosing a number of filters greater than one, it is possible to detect more important features of the input image, in the same layer. The output image has as many channels as the number of input filters.

A new *Layer* functionis created. It receives as input two *Image* classes, one for the input image and the other one for the filters, and then performs the convolution and the pooling. In the main function, it is possible to create a layer of the network just by calling the Layer function.

In the end, it is possible to have multiple layers and the final FC layers. However, due to stack memory problems with the Tvec architecture, it is not possible to show a CNN with as many channels and filters as the most modern network models. The created CNN is a much smaller version of the VGG-16 network.

The simulations and the analysis are made on a CNN with two CONV-POOL layers and a final FC layer. The network is shown in Figure 4.6.

The input image is 18x18 large and has just one channel. In the first layer, the number of filters is 2; in the second layer is equal to 4. In the convolution layer, the padding is zero and the stride is equal to one. As regards the pooling layer, 2x2 portions of the matrix are considered and a max-pooling operation with stride $s = 2$ is performed.

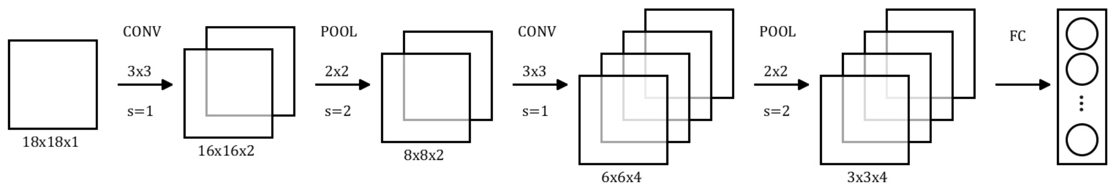


Figure 4.6: Convolutional Neural Network used for the simulations.

4.3.1 First Design

To realize the convolution operation, the Tvec architecture needs some new instructions. Indeed, the starting processor does not support the SIMD multiplication between two vectors, that is shown in Figure 4.7.

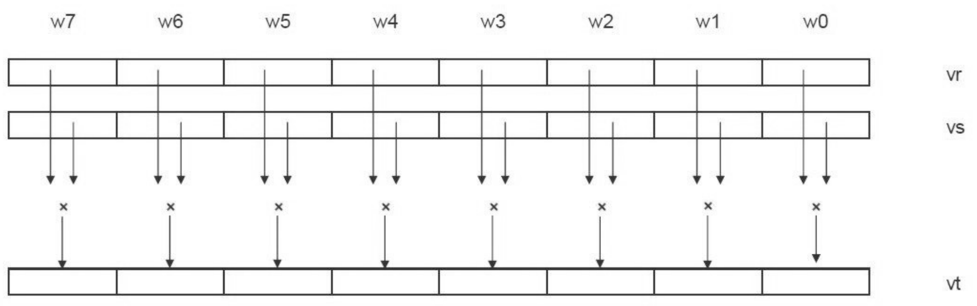


Figure 4.7: The SIMD multiplication between two vectors. [11]

The operands are two 8 bit vectors and the multiplication is saved in a final 8 bit vector [11]. It is possible to add this operation in the Tvec core by modifying three processor files:

- *tvec.h*: it is the primitive processor header file. It is a C++ file in which are declared all the data types and the functions that are used in the nML processor description. The data types are modeled as C++ classes and are used to declare storage in nML, while the functions are modeled as C++ functions.
- *tvec.p*: it is the primitive definition file. The user describes the behavior of the primitive functions using the Primitives Definition and Generation (PDG) language, based on the C language. This file is really important because the PDG tool exploits it to generate the C++ and the Verilog/VHDL files. The first is used by the compiler ISS, while the latter is used in the model created by the tool Go.

- *tvec.n*: in this file, there is the description of the whole processor architecture, written in the hardware description language nML.

The vector multiplier is declared in the primitive processor header file: it has two vword operands and a vword output. Its description is written in the tvec.p file, as shown below.

```
vword mul(vword a, vword b)
{
    word y;
    vword rv;
    for (int32_t i=0; i<VSIZE; i++)
        mulss(a[i], b[i], rv[i], y);
    return rv;
}
```

Listing 4.4: Description of vector multiplication in the PDG file.

There is a *for* loop where the basic multiplication *mulss*, already implemented in the Tvec core, is performed between the elements of the input vectors. Each multiplication is stored in the output vector. Therefore, the latter has the same size of the two input operands.

However, this is only the first step of the convolution; as a second step, a sum of all the elements of the output vector is needed [11]. This operation is performed as shown in Figure 4.8.

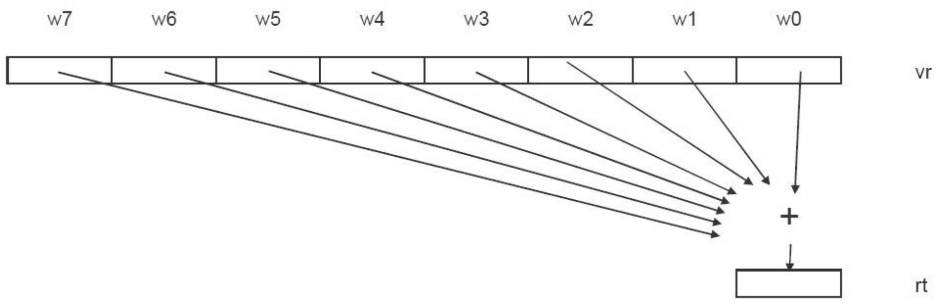


Figure 4.8: Summation of the elements of a vword. [11]

Those steps are repeated for every portion of the input image taken, while the filter remains the same. Then, depending on the number of channels and filters, other convolutions are performed in the same way. The output feature map will go through the next layers, as described in the previous chapters.

The first design of the convolutional layer is shown below.

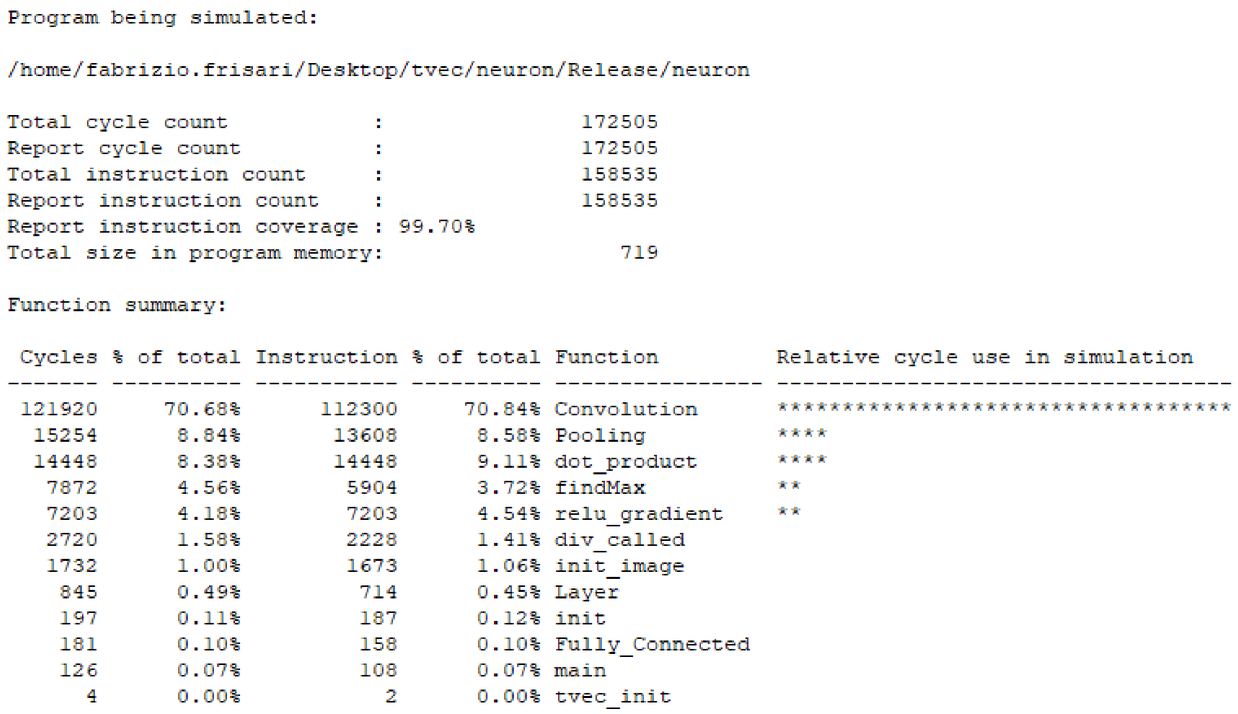
```
// Regular convolution function.
int Convolution(int conv_size,int kernel,int stride,int padding,Image &A, Image &B, Image &D)
{
    int X_temp[4];
    int W_temp[4];
    int convolute = 0, result, final_result;
    int x, y, a=0, b=0, z=0;
    int output = (conv_size - kernel + 2*padding)/stride + 1;
    int pad_size = conv_size+2;

    if(padding == 0)
    {
        for (int j = 0; j < stride*output; j+=stride)
        {
            for (int i = 0; i < stride*output; i+=stride)
            {
                x = i;
                y = j;
                for (int k = 0; k < kernel; k++)
                {
                    for (int l = 0; l < kernel; l++)
                    {
                        W_temp[l]=B.getValue(k,l);
                        X_temp[l]=A.getValue(x,y);
                        y++;
                    }
                    convolute += dot_product((vint*)X_temp, (vint*)W_temp);
                    final_result = relu_gradient(convolute);
                    x++;
                    y = j;
                }
                D.setValues(final_result, b, a);
                convolute = 0;
                b++;
            }
            b=0;
            a++;
        }
    }
  return output;
}
```

Listing 4.5: Convolution Layer of the first design.

In this case, the whole convolution is not a SIMD operation. Indeed, the maximum dimension of the vector type, defined in the Tvec core, is 8. Therefore, the convolution cannot be performed in a single step, with either 3x3 and 4x4 filters. The easiest solution is to temporarily store the rows of the input image and the filter in two different registers. The dot product is then performed between the two rows, with a SIMD multiplication, and saved in a variable. This step is repeated for each row and the value of the output variable is updated, summing up the previous values. Next, the activation function is applied to the result and the output value saved in the output feature map. At this point, a new portion of the image is taken, depending on the stride value, and the same steps are repeated.

The whole C/C++ code of the Convolutional Neural Network, with the first design of the convolution layer, is then imported in the tool ASIP Designer. After choosing the proper processor core (Tvec in this case), it is possible to compile the entire project and run the simulation. The tool provides an instruction report, shown in Figure 4.9.

The report gives information about the amount of memory and instructions used during the simulation. In particular, the simulated program requires a total number of instructions equals to 158535 and those occupy 719 bytes of the program memory.



```
Program being simulated:

/home/fabrizio.frisari/Desktop/tvec/neuron/Release/neuron

Total cycle count           :              172505
Report cycle count          :              172505
Total instruction count     :              158535
Report instruction count    :              158535
Report instruction coverage : 99.70%
Total size in program memory:                 719

Function summary:

 Cycles % of total Instruction % of total Function         Relative cycle use in simulation
------- ---------- ----------- ---------- ---------------- -----------------------------------
 121920     70.68%      112300     70.84% Convolution      ***********************************
  15254      8.84%       13608      8.58% Pooling          ****
  14448      8.38%       14448      9.11% dot_product      ****
   7872      4.56%        5904      3.72% findMax          **
   7203      4.18%        7203      4.54% relu_gradient    **
   2720      1.58%        2228      1.41% div_called
   1732      1.00%        1673      1.06% init_image
    845      0.49%         714      0.45% Layer
    197      0.11%         187      0.12% init
    181      0.10%         158      0.10% Fully_Connected
    126      0.07%         108      0.07% main
      4      0.00%           2      0.00% tvec_init
```

Figure 4.9: Instruction report of the first design.

Moreover, the total number of cycles is 172505.

The report also gives information about the computational cost, in terms of cycles and instructions, for every function of the program. The convolution is the most expensive function: it uses the 70% of the total instructions and cycles. Then, below the 10% there are the *Pooling* and *dot_product* functions as the second and third most expensive.

One of the best features of ASIP Designer is the possibility to analyze, in detail, the computational cost of each assembly instruction. Their total number is 719. The report, for each function of the code, provides a list of assembly instructions associated to their cost in terms of cycles and execution. For instance, the pooling function is performed in 93 assembly instructions, while the convolution in 148.

In the Appendix B there is the assembly instruction report of the convolution function. That report shows that the *if* statement, for padding $p = 0$, starts at the instruction 144 (of the total 719), which is a conditional jump. The loops, instead, start at 147. Inside the *for* loops, there are the most expensive instructions of the CNN. In particular, the load and store instructions, that prepare the operands of the dot product, sometimes require 7200 cycles to be executed. Right before the dot product operation, the compiler can insert a nop operation, because there are no other instructions to schedule at that time. Aftwerwards, the dot product function is called, with a *move immediate* and a *clid* instruction. Then, it requires the loading of the two operands and the data previously stored in the *convolute* variable.

To realize the dot product, four assembly instructions are enough: the load operation of the operands, the vector multiplication (*vmul*) and the final sum of the vector to a scalar variable (vsum).

4.3.2 Second Design

In this second design, the aim is to execute the convolution between the kernel and a portion of the image (of the same size of the kernel) with a single instruction, performing a complete SIMD operation. In this case, both the operands are temporary stored in two different vectors; their size depend on the dimension of the filter (that can be 3x3 or 4x4, in this algorithm).

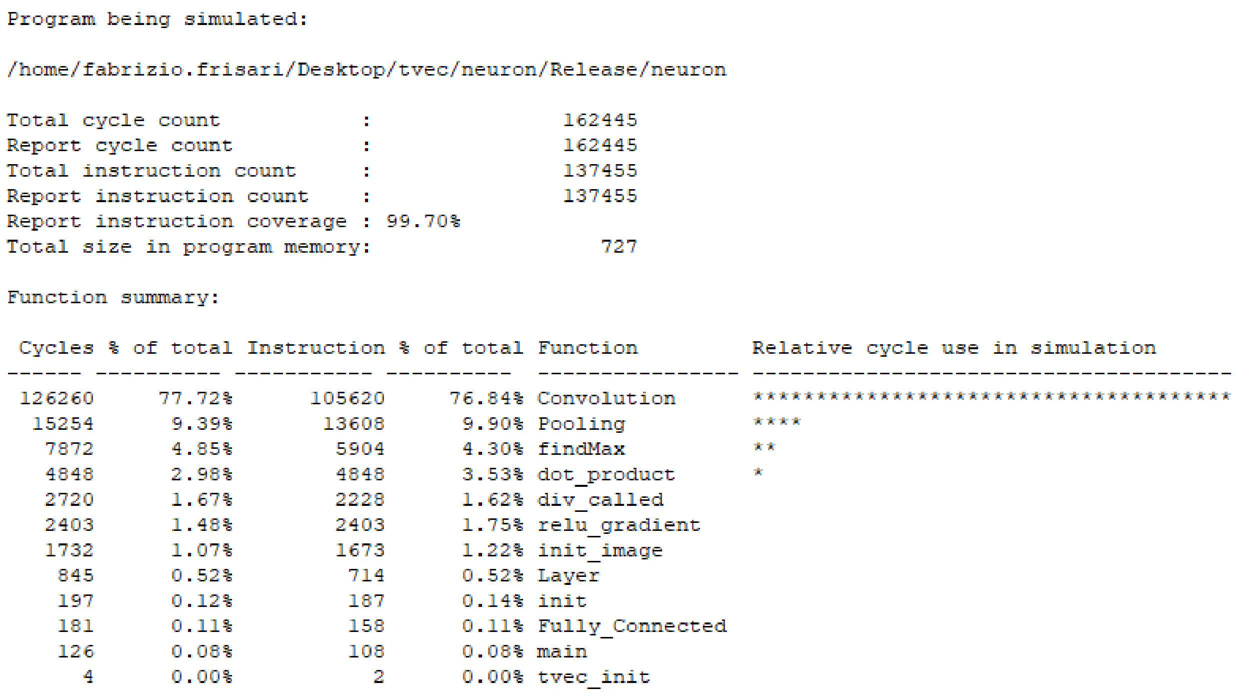
As seen in the previous design, in the standard configuration of the Tvec core the SIMD operations can be made between two vectors whose dimension is not greater than eight. Therefore, the hardware of the processor is modified, in order to increase the size of the vectors. The new dimension is sixteen, the double of the original, and in this way, it is possible to execute the convolution with either size of the filters. The C code of this design is shown below.

```c
// Regular convolution function.
int Convolution(int conv_size,int kernel,int stride,int padding, Image &A, Image &B, Image &D)
{
    int X_temp[16] = {0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0};
    int W_temp[16] = {0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0};
    int convolute = 0, result, final_result;
    int x, y, a=0, b=0, z=0;
    int output = (conv_size - kernel + 2*padding)/stride + 1;
    int pad_size = conv_size+2;
    int temp_size = kernel*kernel;

    if(padding == 0)
    {
        for (int j = 0; j < stride*output; j+=stride)
        {
            for (int i = 0; i < stride*output; i+=stride)
            {
                x = i;
                y = j;
                for (int k = 0; k < kernel; k++)
                {
                    for (int l = 0; l < kernel; l++)
                    {
                        if(i==0 && j==0)
                            W_temp[z]=B.getValue(x,y);
                        X_temp[z]=A.getValue(x,y);
                        z++;
                        y++;
                    }
                    x++;
                    y = j;
                }
                convolute += dot_product((vint*)X_temp, (vint*)W_temp);
                final_result = relu_gradient(convolute);
                D.setValues(final_result, b, a);
                convolute = 0;
                b++;
                z=0;
            }
            b=0;
            a++;
        }
    }
  return output;
}
```

Listing 4.6: Convolution Layer of the second design.

It is similar to the first design, but instead of temporarily store just a row of the operands, the whole matrices are stored in two vectors. In this way, it is fast and simple to perform the dot product between the two operands, to sum all the elements of the output vector in a single variable and to store it in the output feature map. The remaining part of the function is unchanged.

In Figure 4.10 there is the instruction report of the second design.



```
Program being simulated:

/home/fabrizio.frisari/Desktop/tvec/neuron/Release/neuron

Total cycle count           :               162445
Report cycle count          :               162445
Total instruction count     :               137455
Report instruction count    :               137455
Report instruction coverage : 99.70%
Total size in program memory:                  727

Function summary:

 Cycles % of total Instruction % of total Function         Relative cycle use in simulation
------ ---------- ----------- ---------- ---------------- --------------------------------------
 126260     77.72%      105620     76.84% Convolution      **************************************
  15254      9.39%       13608      9.90% Pooling          ****
   7872      4.85%        5904      4.30% findMax          **
   4848      2.98%        4848      3.53% dot_product      *
   2720      1.67%        2228      1.62% div_called
   2403      1.48%        2403      1.75% relu_gradient
   1732      1.07%        1673      1.22% init_image
    845      0.52%         714      0.52% Layer
    197      0.12%         187      0.14% init
    181      0.11%         158      0.11% Fully_Connected
    126      0.08%         108      0.08% main
      4      0.00%           2      0.00% tvec_init
```

Figure 4.10: Instruction report of the second design.

In this case, the convolution is still the most expensive function of the program. Anyway, the number of total instructions is lower of about 20k, while the total cycle count is reduced of 10k cycles. This is an important result: with a complete SIMD convolution, the program is optimized with respect to the first design.

However, by looking in details at the instructions of the CONV function, it is clear that the biggest contribution is still given by the load and store operations. At every iteration, the amount of data to load and store is really high. This can be the starting point for further improvements.

4.3.3 Third Design

The third design is made to reduce the number of store and load operations that the program has to perform for every dot product in the convolution function. This is crucial because, in a neural network with a large number of channels and filters, there are thousands of vectors to temporary store in the memory.

The proposed approach can be fully exploited when the stride is equal to one. In fact, in this case, considering a 3x3 filter (but the approach still works for a 4x4 kernel), each portion of the input image has six over nine elements in common with the previous one. Therefore, instead of saving the whole new matrix at each iteration, only three new values are stored.

However, in this way the order of the rows of the matrix changes; therefore, the kernel cannot just be saved once: it is stored in three different vectors (four, in the 4x4 case) and at each iteration, the right vector is taken for the dot product.

Another possible solution is to exploit shift operations to use only two vectors to store the matrices, one for the input feature map and the other one for the filter. In this way, at each iteration, the vector is updated by shifting left the old values that are still valid for the next computations, while the useless ones are removed. Then, the new values are added in the most right cells and the dot product can be performed. However, this implementation is not so efficient and thus it is discarded.

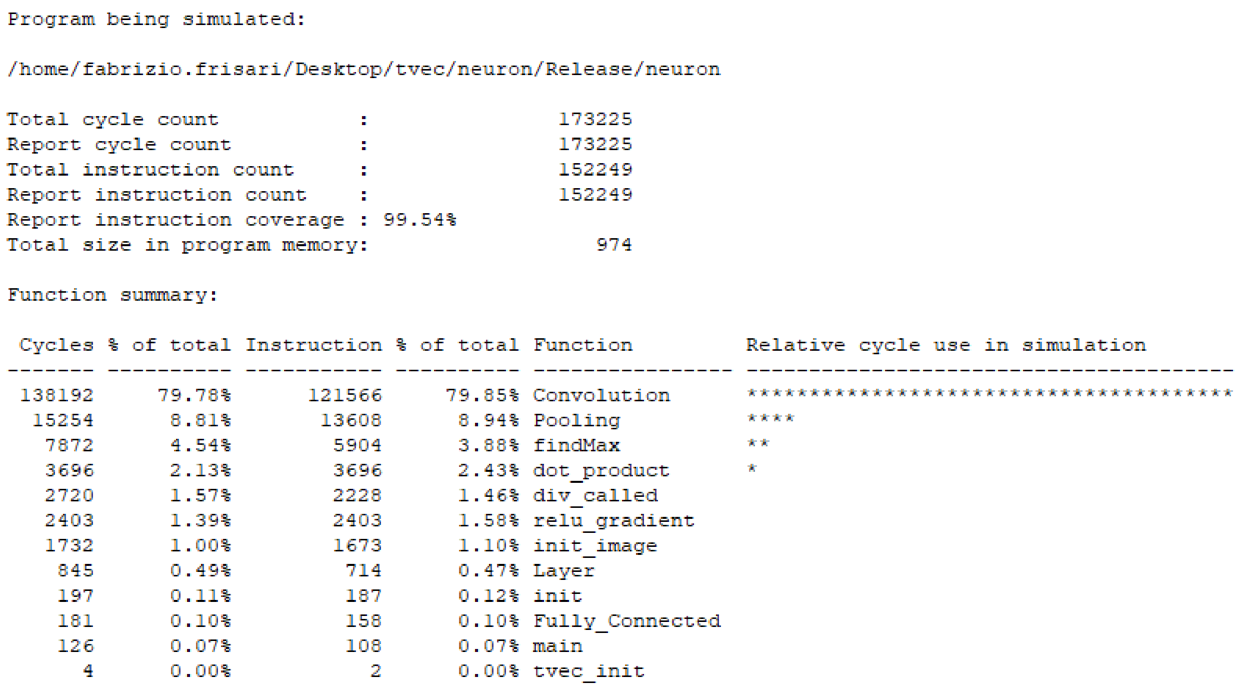
The C code of the convolution function is shown below.

```c
// Regular convolution function.
int Convolution(int conv_size,int kernel,int stride,int padding, Image &A, Image &B, Image &D)
{
    int X_temp[16] = {0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0};
    int X_final[16] = {0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0};
    int W_temp[16] = {0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0};
    int W_temp_2[16] = {0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0};
    int W_temp_3[16] = {0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0};
    int W_temp_4[16] = {0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0};
    int convolute = 0, result, partial, final_result;
    int x, y, a=0, b=0, z=0, c=kernel, d=2*kernel, e=3*kernel;
    int output = (conv_size - kernel + 2*padding)/stride + 1;
    int pad_size = conv_size+2;
    int temp_size = kernel*kernel;

    if(padding == 0)
    {
        for (int j = 0; j < stride*output; j+=stride)
        {
            x=0;
            for (int i = 0; i < stride*output; i+=stride)
            {
                y = j;
                for (int k = 0; k < kernel; k++)
                {
            if(i==0)
                    {
                        for (int l = 0; l < kernel; l++)
                        {
                            if(j==0)
                            {
                        W_temp[z]=B.getValue(x,y);
                        W_temp_2[c]=B.getValue(x,y);
                        W_temp_3[d]=B.getValue(x,y);
                        if(kernel == 4)
                        {
                            W_temp_4[e]=B.getValue(x,y);
                            e++;
                        }
                        c++;
                        d++;

                        if(c==temp_size)
                            c=0;
                        if(d==temp_size)
                            d=0;
                        if(e==temp_size)
                            e=0;
                            }
                    if(z==temp_size)
                                z=0;
                    X_temp[z]=A.getValue(x,y);
                            z++;
                            y++;
                        }
            x++;
            y = j;
                    }

                    if(i>0)
                    {
                        if(z==temp_size)
                            z=0;
                        X_temp[z]=A.getValue(x,y);
                        y++;
                        z++;
                    }
                }
                if(i>0)
                    x++;
                if(i==0 || i==kernel   || i==2*kernel || i==3*kernel)
                    convolute += dot_product((vint*)X_temp, (vint*)W_temp);

                if(i==1 || i==kernel+1 || i==2*kernel+1)
                    convolute += dot_product((vint*)X_temp, (vint*)W_temp_2);
                if(i==2 || i==kernel+2 || i==2*kernel+2)
                    convolute += dot_product((vint*)X_temp, (vint*)W_temp_3);
                if(kernel == 4)
                    if(i==3 || i==kernel+3 || i==2*kernel+3)
                        convolute += dot_product((vint*)X_temp, (vint*)W_temp_4);

                final_result = relu_gradient(convolute);
                D.setValues(final_result , b, a);
                convolute = 0;
                b++;
            }
            b=0;
            a++;
        }
    }
  return output;
}
```

Listing 4.7: Convolution Layer of the third design.

As can be seen, it is necessary to create four *W_temp* vectors that store the kernel in different orders. To perform the correct dot product, some *if* conditions are needed. Those can reduce the efficiency of the code by increasing the computational cost. Unfortunately, they are necessary in the software approach.

The instruction report of the third design is presented in Figure 4.11. The convolution function increases its cost in terms of cycles and instructions: it is about 80% of the total amount. However, the total cycle count is now increased with respect to the second design and it is similar to the first one. This is due to the *if* statements used in this configuration. Indeed, by looking at the assembly of the convolution function, it shows that the conditional jump instruction *jcr*, useful for choosing the right operand, needs 2400 cycles and 6720 instructions to be executed:



```
Program being simulated:

/home/fabrizio.frisari/Desktop/tvec/neuron/Release/neuron

Total cycle count           :               173225
Report cycle count          :               173225
Total instruction count     :               152249
Report instruction count    :               152249
Report instruction coverage : 99.54%
Total size in program memory:                  974

Function summary:

 Cycles % of total Instruction % of total Function         Relative cycle use in simulation
------- ---------- ----------- ---------- ---------------- ----------------------------------------
 138192     79.78%      121566     79.85% Convolution      ****************************************
  15254      8.81%       13608      8.94% Pooling          ****
   7872      4.54%        5904      3.88% findMax          **
   3696      2.13%        3696      2.43% dot_product      *
   2720      1.57%        2228      1.46% div_called
   2403      1.39%        2403      1.58% relu_gradient
   1732      1.00%        1673      1.10% init_image
    845      0.49%         714      0.47% Layer
    197      0.11%         187      0.12% init
    181      0.10%         158      0.10% Fully_Connected
    126      0.07%         108      0.07% main
      4      0.00%           2      0.00% tvec_init
```

Figure 4.11: Instruction report of the third design.

more than any other operation of the convolution function. Moreover, the load and store operations still have a huge impact on the overall cost.

4.3.4 Fourth Design

One way to improve the third design is to reduce the number of temporary vectors that are used to store the filter matrices. An efficient way to accomplish that result is to implement a structure called circular buffer. It also known as ring buffer and an example is shown in Figure 4.12.

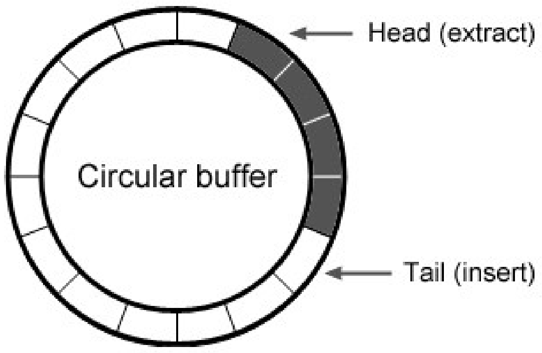


Figure 4.12: Structure of a circular buffer.

It is a data structure that exploits a single buffer of a fixed dimension that is treated as circular, even though it is a linear structure.

The circular buffer is organized in a FIFO (First In First Out) manner and uses a pointer to its inner locations: starting from an empty buffer, it is possible to fill it with new values, updating the pointer to the next element. If the FIFO is full and a writing operation is performed, the oldest value is removed and the pointer incremented.

The most efficient way to realize the ring buffer, in the convolution function, is probably a hardware implementation. Anyway, in this fourth design, a software version is realized. It is a simpler solution and it is worth checking if a good result is obtained without using the hardware accelerator. Indeed, the latter is solution is more invasive since requires another change to the Tvec architecture. In this way, it can be proved if the hardware accelerator solves the problems that may arise with the software solution and a comparison can be made.

The software solution is shown below.

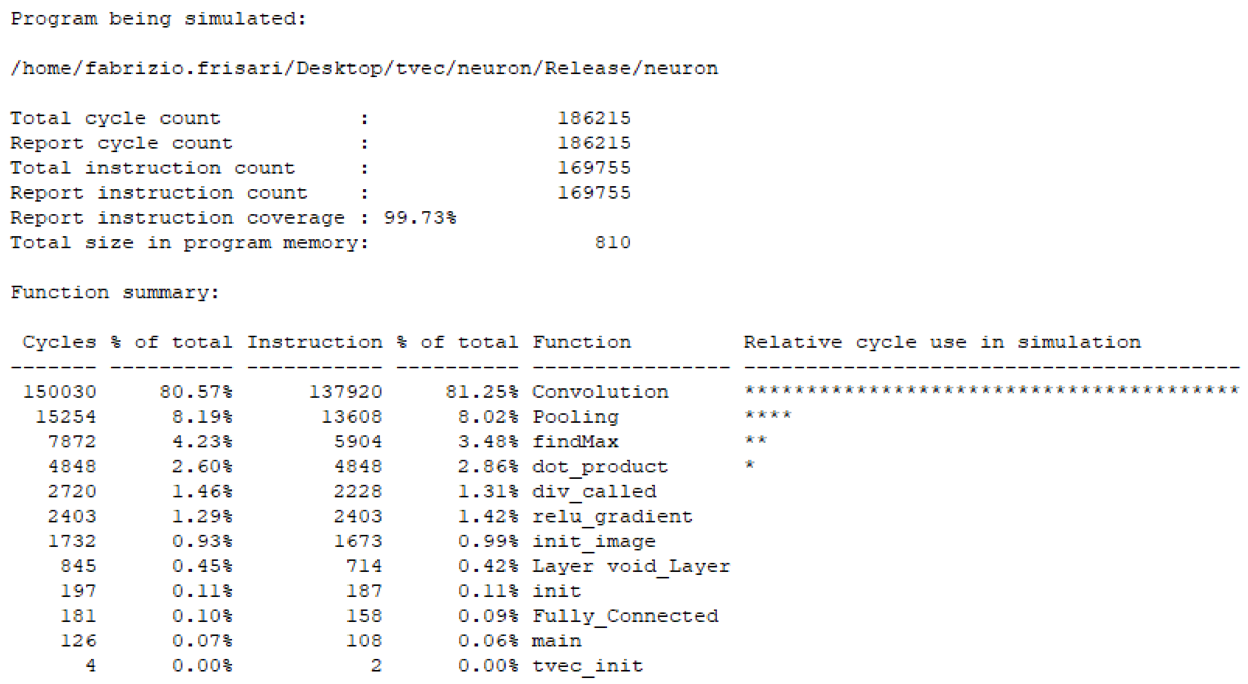
```
// Regular convolution function.
int Convolution(int conv_size,int kernel,int stride,int padding, Image &A, Image &B, Image &D)
{
    int X_temp[16] = {0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0};
    int X_final[16] = {0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0};
    int W_temp[16] = {0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0};
    int convolute = 0, result, partial, final_result;
    int x, y, a=0, b=0, z=0;
    int output = (conv_size - kernel + 2*padding)/stride + 1;
    int pad_size = conv_size+2;
    int temp_size = kernel*kernel;

    if(padding == 0)
    {
      for (int j = 0; j < stride*output; j+=stride)
      {
          x=0;
            for (int i = 0; i < stride*output; i+=stride)
            {
                y = j;
          for (int k = 0; k < kernel; k++)
          {
            if(i==0)
                      {
                          for (int l = 0; l < kernel; l++)
                          {
                              if(j==0)
                                  W_temp[z] = B.getValue(x,y);
                      if(z==temp_size)
                                  z=0;
                      X_temp[z] = A.getValue(x,y);
                              z++;
                              y++;
                          }
                          x++;
                          y = j;
                      }

                      if(i>0)
                      {
                          if(z==temp_size)
                              z=0;
                          X_temp[z] = A.getValue(x,y);
                          y++;
                          z++;
                      }
                  }

          if(i>0)
          {
              x++;
              for(int r = 0; r < temp_size; r++)
              {
                  if (z == temp_size)
                              z=0;
                  X_final[r] = X_temp[z];
                              z++;
              }
          }
          if(i==0)
              convolute += dot_product((vint*)X_temp, (vint*)W_temp);
          else
              convolute += dot_product( (vint*)X_final, (vint*)W_temp);
                final_result = relu_gradient(convolute);
          D.setValues(final_result , b, a);
          convolute = 0;
          b++;
            }
            b=0;
            a++;
        }
    }
  return output;
}
```

Listing 4.8: Convolution Layer of the fourth design.

The *X_final* vector is added in the code to realize a circular buffer. The image operand is eventually stored in that vector; then, it is performed, as usual, the dot product between the latter and the filter. When the operand of the dot product is the first portion of the input feature map, the circular buffer is not used.

In Figure 4.13 there is the report for this implementation.

The total cycle count is the largest among the considered designs. Therefore, a pure software implementation of the circular buffer is not a good solution. The detailed report of the convolution function shows a reduced impact of the *if* statements because of a larger contribution of the store and load instructions.



```
Program being simulated:

/home/fabrizio.frisari/Desktop/tvec/neuron/Release/neuron

Total cycle count            :              186215
Report cycle count           :              186215
Total instruction count      :              169755
Report instruction count     :              169755
Report instruction coverage  : 99.73%
Total size in program memory:                  810

Function summary:

 Cycles % of total Instruction % of total Function         Relative cycle use in simulation
------- ---------- ----------- ---------- ---------------- ----------------------------------------
 150030     80.57%      137920     81.25% Convolution      ****************************************
  15254      8.19%       13608      8.02% Pooling          ****
   7872      4.23%        5904      3.48% findMax          **
   4848      2.60%        4848      2.86% dot_product      *
   2720      1.46%        2228      1.31% div_called
   2403      1.29%        2403      1.42% relu_gradient
   1732      0.93%        1673      0.99% init_image
    845      0.45%         714      0.42% Layer void_Layer
    197      0.11%         187      0.11% init
    181      0.10%         158      0.09% Fully_Connected
    126      0.07%         108      0.06% main
      4      0.00%           2      0.00% tvec_init
```

Figure 4.13: Instruction report of the fourth design.

A hardware solution is necessary to accelerate the computation and to increase efficiency, as is shown in the next chapter.

4.3.5 Fifth Design

Although the software ring buffer did not bring interesting results, it is possible to exploit the feature of ASIP Designer to realize a hardware accelerator.

In this way, the functions with the largest computational cost can be mitigated and the machine will have better performances.

To realize a hardware circular buffer, the Tvec core is modified one more time. First of all, the primitive function of the ring buffer, called *barrel*, is declared in the primitive header file. Then, it is described in the *tvec.p* file, as shown below.

```
vword barrel(vword a, word z, word c)
{
    vword rv;
    word s=z;
    if(c==9)
    {
        for(int32_t i=0; i<9; i++)
        {
            if(s==9)
                s=0;
            rv[i]=a[s];
            s=s+1;
        }
    }
    else
    {
        for(int32_t i=0; i<16; i++)
        {
            if(s==16)
                s=0;
            rv[i]=a[s];
            s=s+1;
        }
    }
    return rv;
}
```

Listing 4.9: Circular buffer implementation.

The code above shows two different loops: the first is for 3x3 kernels, while the second for 4x4 ones. Then, the nML description of the processor is modified. In particular, the new function is added in the *vector.n* file, where there is the description of the vector operations supported by the Tvec architecture. The *barrel* function needs one vector and two variables as input and outputs a vector. The description is shown below:

```
opn vec_v(t: vt, u:ru, r:rr, s:vs)
{
    action {
    stage E1:
        vecs = s;
        temp_size = u;
        z = r;
        vect = barrel(vecs, z, temp_size) @vec;
        t = vect;
    }
    syntax : t "," u "," r "," s;
    image : "011"::t::u::r::s;
}
```

Listing 4.10: nML circular buffer description.

The operation is performed by the *vec* functional unit (discussed in the chapter 4.1) in the E1 stage of the pipeline. The source and destination vector registers are *vs* and *vt*, respectively, and these will store the input and output vectors of the function.

Then, there are the syntax and image attributes to consider. The first represents the assembly syntax for the instructions, while the second specifies the binary encoding for the corresponding instruction [12]. The *barrel* instruction is now fully described and it can be used in the final implementation of the convolution function, as shown below.

```
// Regular convolution function.
int Convolution(int conv_size,int kernel,int stride,int padding, Image &A, Image &B, Image &D)
{
    int X_temp[16] = {0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0};
    int X_final[16] = {0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0};
    int convolute = 0, result, partial, final_result;
    int x, y, a=0, b=0, z=0;
    int output = (conv_size - kernel + 2*padding)/stride + 1;
    int pad_size = conv_size+2;
    int temp_size = kernel*kernel;

    if(padding == 0)
    {
      for (int j = 0; j < stride*output; j+=stride)
      {
          x=0;
            for (int i = 0; i < stride*output; i+=stride)
            {
                y = j;
          for (int k = 0; k < kernel; k++)
```

```
            {
              if(i==0)
                      {
                          for (int l = 0; l < kernel; l++)
                          {
                              if(j==0)
                                   W_temp[z] = B.getValue(x,y);
                     if(z==temp_size)
                                   z=0;
                     X_temp[z] = A.getValue(x,y);
                              z++;
                              y++;
                          }
                          x++;
                          y = j;
                      }

                      if(i>0)
                      {
                          if(z==temp_size)
                              z=0;
                          X_temp[z] = A.getValue(x,y);
                          y++;
                          z++;
                      }
                  }

            if(i>0)
            {
                x++;
                barrel_op((vint*)X_temp, z, temp_size, (vint*)X_final);
            }
            if(i==0)
                convolute += dot_product((vint*)X_temp, (vint*)W_temp);
            else
                convolute += dot_product((vint*)X_final, (vint*)W_temp);

                  final_result = relu_gradient(convolute);
            D.setValues(final_result, b, a);
            convolute = 0;
          b++;
              }
              b=0;
              a++;
        }
    }
  return output;
}
```

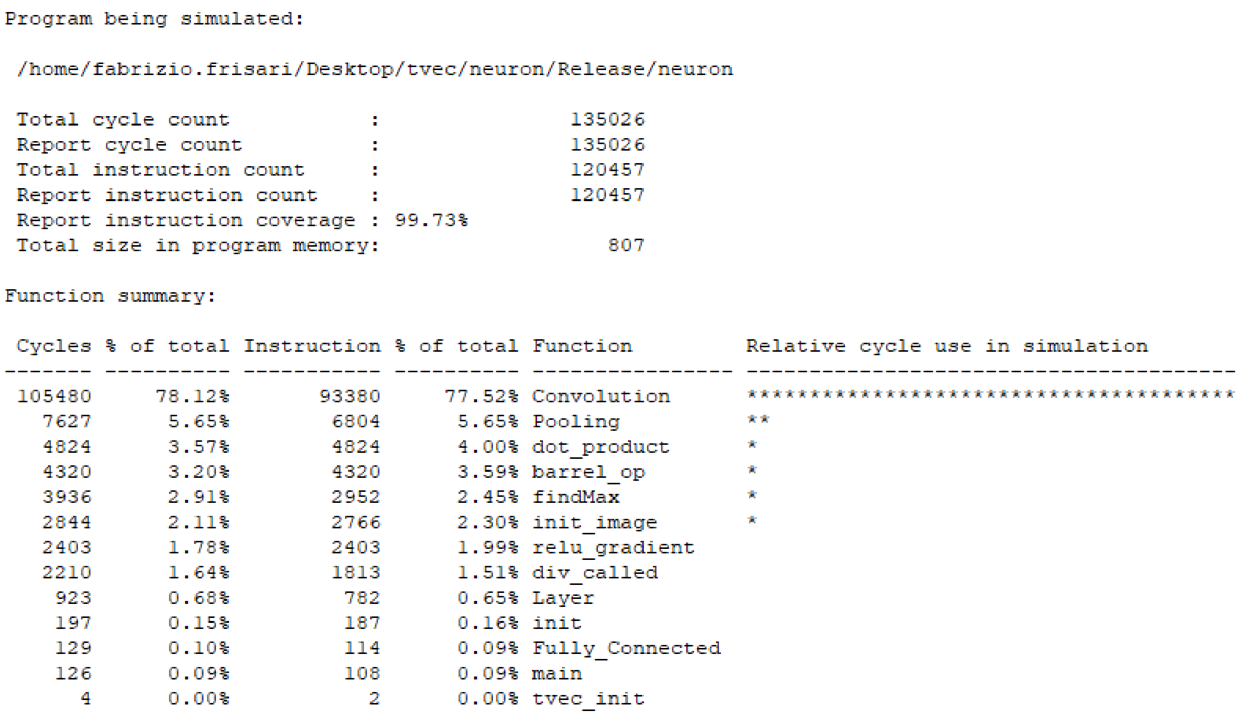
Listing 4.11: Convolution Layer of the fifth design.

The code is simpler because it is enough to call the new *barrel* function to exploit the ring buffer architecture.

As can be seen, inside the *for* loop, when the index i is equal to zero, the *barrel* function is not called. In that case, the first row portions of the input feature are taken: the image matrix is in perfect order and can be multiplied with the kernel. Therefore, there is no need to use the circular buffer and some instructions can be saved. In all the other cases, the ring buffer speeds up the execution, as can be noticed in the instruction report in Figure 4.14.

In this last design of the Convolutional Neural Network, the total cycle count is 135026 and the total instruction count is 120457.

In table 4.1 there is a comparison between the five implementation used in this work. A Figure of merit (FoM) that represents the total computational cost of the design is used. It is evaluated by multiplying the total cycle count, the total size in the program memory and the total instruction count; the result is normalized to the value obtained in the first design. This FoM is useful to compare the designs used in this work.



```
Program being simulated:

 /home/fabrizio.frisari/Desktop/tvec/neuron/Release/neuron

 Total cycle count           :              135026
 Report cycle count          :              135026
 Total instruction count     :              120457
 Report instruction count    :              120457
 Report instruction coverage : 99.73%
 Total size in program memory:                 807

Function summary:

 Cycles % of total Instruction % of total Function         Relative cycle use in simulation
------- ---------- ----------- ---------- ---------------- ---------------------------------------
 105480     78.12%       93380     77.52% Convolution      ***************************************
   7627      5.65%        6804      5.65% Pooling          **
   4824      3.57%        4824      4.00% dot_product      *
   4320      3.20%        4320      3.59% barrel_op        *
   3936      2.91%        2952      2.45% findMax          *
   2844      2.11%        2766      2.30% init_image       *
   2403      1.78%        2403      1.99% relu_gradient
   2210      1.64%        1813      1.51% div_called
    923      0.68%         782      0.65% Layer
    197      0.15%         187      0.16% init
    129      0.10%         114      0.09% Fully_Connected
    126      0.09%         108      0.09% main
      4      0.00%           2      0.00% tvec_init
```

Figure 4.14: Instruction report of the fifth design.

Table 4.1: Report comparison between the five implementations.

Design	Total Cycle Count	Total Instruction Count	Total Size in Program Memory	FoM
First	172505	158535	719	1
Second	162445	137455	727	0.82
Third	173225	152249	974	1.31
Fourth	186215	169755	810	1.30
Fifth	135026	120457	807	0.68

The fifth design, that uses a complete SIMD convolution function and exploits a hardware circular buffer, is the most efficient solution. Indeed, it has the lowest FoM, equal to 0.6. If a deeper CNN is used, the differences between these implementation can be even larger.

Chapter 5

Simulation and Synthesis

5.1 Creation of HDL

In this section, another tool of ASIP Designer, called Go, is exploited. It is a hardware description language (HDL) generator and it is a powerful tool that translates the description of the processor architecture into a synthesizable HDL: in fact, starting from the nML code, it is possible to obtain a Verilog or VHDL description, at the RTL (Register Transfer Level) [13].

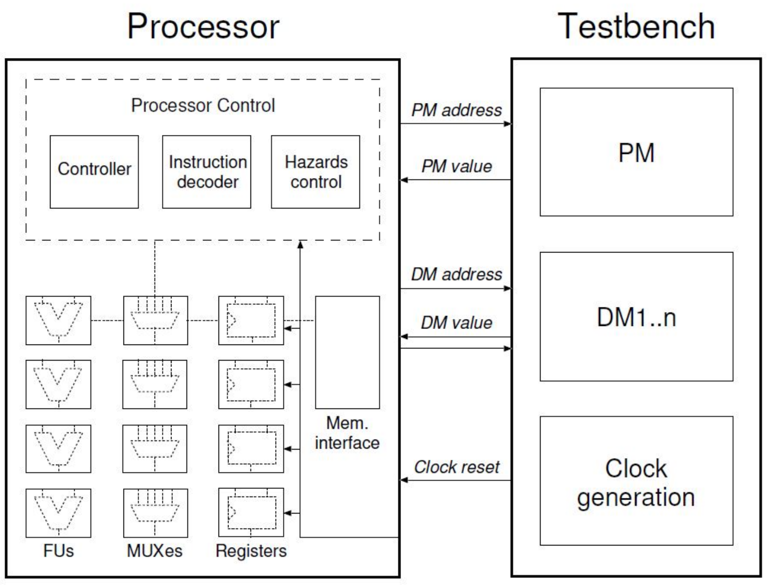


Figure 5.1: Schematic representation of the data path, generated by GO. Every rectangle in solid lines represents a separate entity. [13]

In Figure 5.1 there is a schematic of the data path generated by Go. As can be seen, it translates every single hardware unit and the final HDL code is complete. In the hdl directory, following the steps in Figure 5.2, it is possible to the read all the generated files.

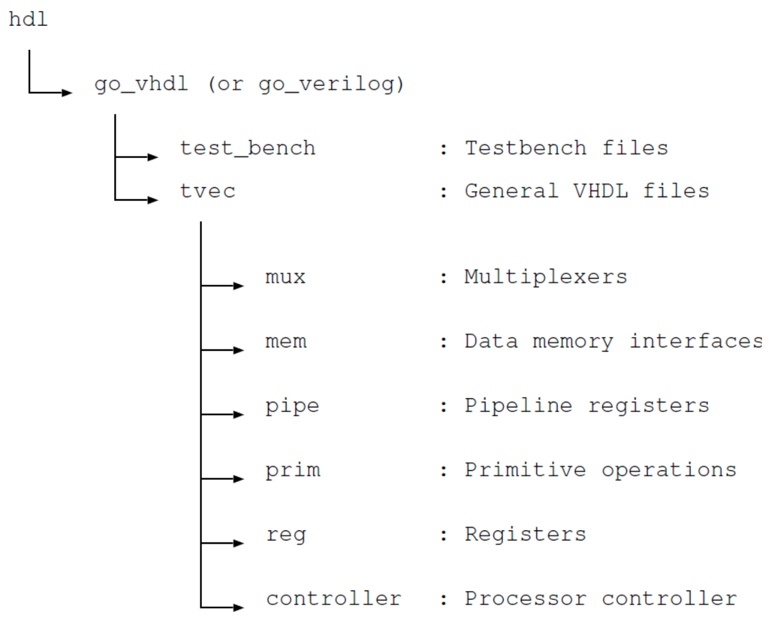

hdl
go_vhdl (or go_verilog)
test_bench : Testbench files
tvec : General VHDL files
mux : Multiplexers
mem : Data memory interfaces
pipe : Pipeline registers
prim : Primitive operations
reg : Registers
controller : Processor controller

Figure 5.2: The directory structure created by GO. [13]

First of all, in the Tvec subdirectory, it is possible to find the tvec.v file. It is the most important Verilog file, since it is the highest level module that describes the whole architecture and how all the blocks are connected [13]. Then, as can be seen in Figure 5.2, there are many folders:

- mux: in this subdirectory, the multiplexers of the nML description are implemented.
- mem: it has the implementation of a memory interface for every memory defined in the core. In the Tvec case, there are three memories: Program Memory (PM), Data Memory (DM) and Vector Data Memory (DMv).
- pipe: in this section, there are all the pipeline registers.
- prim: in this folder there are the functional units of the architecture.
- reg: it contains the register files and the physical registers of the Tvec processor.
- controller: this folder contains some important modules, such as the instruction set decoder and the hazard unit. The latter is important to prevent pipeline hazards, that are dangerous for the machine. Moreover, there is another

important file, named controller.v, that sets the new value of the program counter.

In the end, there is also the testbench directory that can be used for the simulation of the Verilog files.

5.1.1 Simulation

As seen in the paragraph 5.1, the Go tool of ASIP Designer generates the HDL of the Tvec processor. Besides, it provides the test-bench and a Makefile as well. Those are useful to simulate the HDL (Verilog, in this case) of the Tvec core. In particular, with the default settings of the Makefile, the tool allows to analyze, elaborate and simulate the RTL [13].

It is also possible to change some commands to simulate the architecture with Synopsy VCS(-MX), Mentor/Modelsim or Cadence. Those simulations can be compared with the ones performed by ASIP Designer in the instruction set simulator (ISS).

5.1.2 Synthesis

After the simulation, the last step of this thesis is the synthesis of the processor with Synopsys Design Compiler. The aim is to find the area, the power consumption and the maximum operating frequency of the implementation used in this work.

The Verilog HDL of the Tvec processor is loaded in Design Compiler; then all the files are analyzed and the top level module is elaborated. It is possible to apply some constraints to the design, such as the uncertainty of the clock signal. The memories are not synthesized. In the end, it is possible to compile the whole design and to read the reports of area, timing and power consumption.
These steps are repeated for the implementation used in this work:

1. The original Tvec processor provided by Synopsys.

2. The Tvec core with the addition of the vector multiplication

3. The processor with the double size of the vector types used in the third implementation.

4. The complete architecture used in the final design, with the implementation of the *barrel* function that realizes the hardware ring buffer.

The comparison between these four designs is in Table 5.1 and Table 5.2.
The maximum frequency operation is 243.90 MHz for all the four architectures used in this work. As regards the area, the final solution gives the minimum value.

Table 5.1: Area and frequency of the main four designs.

Design	Area [μm^2]	Frequency [MHz]
First	21736.99	243.90
Second	25925.42	243.90
Third	21694.96	243.90
Fourth	21102.31	243.90

Table 5.2: Power consumption of the main four designs.

Architecture	Internal Power [mW]	Switching Power [μW]	Leakage Power [μW]	Total Power [mW]
First	2.2013	81.8891	444.00	2.7272
Second	2.2012	82.0578	535.94	2.8192
Third	3.0892	97.0377	647.45	3.8337
Fourth	2.1992	82.8041	417.56	2.6995

In Table 5.2 there is the power report provided by Synopsys Design Compiler. It gives information about the static and dynamic power, where the latter gives the greatest contribution in all the implementations.

The final Convolutional Neural Network, used in the last implementation of the previous chapter, is the best solution also in terms of power consumption.

Chapter 6

Final conclusions and future work

6.1 Conclusions

This work showed how the Deep Learning algorithms may be created from scratch, in a C/C++ code, and used in the tool ASIP Designer. Among the possible architectures, the Tvec core is the best solution because it provides the SIMD operations, really useful when dealing with large amounts of data. In fact, Convolutional Neural Network may need lots of storage to save the feature maps and the kernels. The tiny CNN used in this work is not as deep as the most recent learning machines, but the obtained result are still valid and can be taken into account for future implementations.

Thanks to the reports provided by ASIP Designer, it was possible to perform on-going optimizations on both the hardware and software models. In the end, the best solution, in terms of required computational cost, is obtained with hardware accelerators that can execute vector operations and modules, such as the circular buffer, that reduce the total count of instructions.

Another advantage of this tool is the automatic generation of the HDL of the processor, with the Go tool. In this way, the architecture can be easily simulated and synthesized. The final design of the Tvec core requires less area and it is more efficient in terms of power and maximum operating frequency.

6.2 Future works

Future works may exploit or create more advanced architectures and optimize them with the help of ASIP Designer. In particular, Multiple Instructions Multiple Data (MIMD) type of cores may be used, instead of just SIMD processors. Besides, there will be more information about the features of Synopsys tool: it would become a standard approach for software and hardware engineers.

Moreover, in future it will be possible to use architectures compatible with more C/C++ libraries. In this way, more optimized descriptions of the architecture can be performed, in terms of total cycles and instructions count.

As regards the optimizations of a Convolutional Neural Network, by using more complicated architectures and with larger stack areas, it will be possible to analyze and improve deeper networks, similar to the most popular.

Exploiting ASIP Designer for Machine Learning algorithms can be a powerful tool to improve their efficiency and to increase the Deep Learning area of application.

Appendix A

Code of the Convolutional Neural Network

```
#define N 2
#define SIZE VSIZE
#define filter_1 2
#define filter_2 4
#define channel_1 1
#define channel_2 2

vint P[SIZE];
int W_FC[SIZE][SIZE];
int Y_FC[SIZE][SIZE];
int W_1[SIZE][SIZE];
int W_2[SIZE][SIZE];
int W_3[SIZE][SIZE];
int X_prova[SIZE][SIZE][SIZE];

//Function to initialize input and weights
void init(int image_size, int num_hidden_neuron, int vector_element_first)
{
    for (int i = 0; i < image_size; i++)
        P[i] = (i);
    int r = 0;
    // Weight for Fully connected layer
    for (int i = 0; i < num_hidden_neuron; i++)
    {
        for (int j = 0; j < vector_element_first; j++)
            W_FC[i][j] = (i+j-1);
    }
}

//Scalar product between 2 vectors
int dot_product(vint *pa, vint *pb)
{
        int product = 0;
        vint va = *pa;
        vint vb = *pb;
        vint dot_ab = mul(va,vb);
        product = sum(dot_ab);
        return product;
}

// Circular buffer
void barrel_op(vint *pa, int z, int temp_size, vint *pc)
{
        vint va = *pa;
        *pc = barrel(va, z, temp_size);
}

// Activation functions
int relu_gradient(int x)
{
  if(x > 0) return x;
  else return 0;
}
```

```

int leak_relu(int x)
{
  if(x > 0) return x;
  else return (int)0.01*x;
}

int sigmoid(int x)
{
  if(x > 1) return 1;
  else if(x < -1) return 0;
  else return x;
}

int tanh(int x)
{
    return tanh(x);
}


class Image
{
  public:
    Image();
    void init_image(int kernel, int t);

    void setValues(int value, int i, int k)
    {
        image[i][k] += value;
    }
    int getValue(int a, int b)
    {
      return image[a][b];
    }

  private:
    int image[SIZE][SIZE];
};

Image::Image()
{

        image[0][0] = 0;
}

void Image::init_image(int kernel, int t)
{
  for(int i=0; i<kernel; i++)
  {
    for(int j=0; j<kernel; j++)
    {
            image[i][j] = j+2*i-t-1;
    }
  }
}

// Regular convolution function.
int Convolution(int convolution_size, int kernel, int stride, int padding, //int channel, int
     filter,
                Image &A, Image &B, Image &D)//int D[][SIZE]) //  int A[][SIZE], int B[][SIZE],
     //int T[][SIZE][SIZE][SIZE], int S[][SIZE][SIZE],
{
    int X_temp[16] = {0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0};
    int X_final[16] = {0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0};
    int W_temp[16] = {0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0};
    int convolute = 0, result, partial, final_result;
    int x, y, a=0, b=0, z=0;
    int output = (convolution_size - kernel + 2*padding)/stride + 1;
    int pad_size = convolution_size+2;
    int temp_size = kernel*kernel;

    if(padding == 1)
    {
        int Pad[SIZE+2][SIZE+2];
        for(int i=0; i<pad_size; i++)
        {
                Pad[i][0]=0;
                Pad[0][i]=0;
                Pad[i][pad_size-1]=0;
                Pad[pad_size-1][i]=0;
        }
        for(int i=1; i<pad_size-1; i++)
        {
```

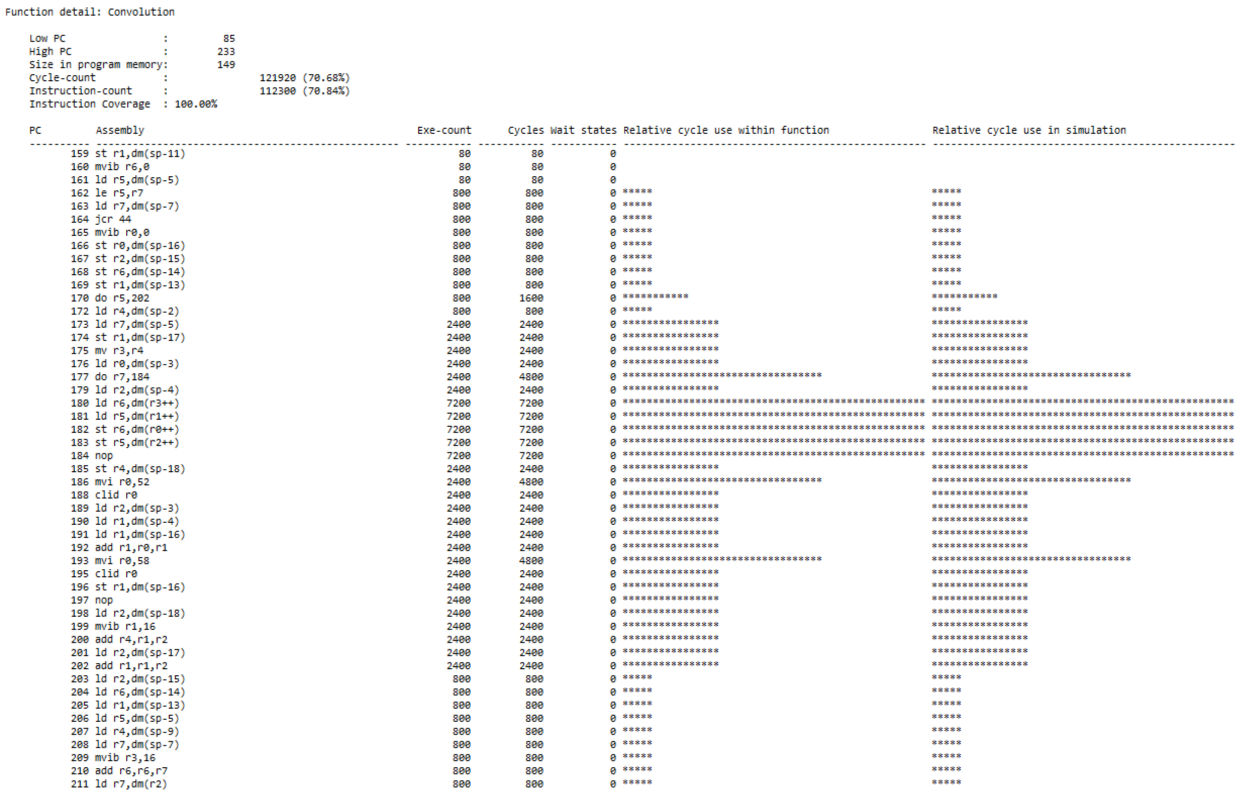
```
                  for(int j=1; j<pad_size-1; j++)
                      Pad[i][j]=A.getValue(i-1,j-1); //A[i-1][j-1];
          }

 // Fill output matrix: rows and columns are i and j respectively
          for (int j = 0; j < stride*output; j+=stride)
          {
              x=0;
              for (int i = 0; i < stride*output; i+=stride)
              {
                  y = j;
                  for (int k = 0; k < kernel; k++)
                  {
                      if(i==0)
                      {
                          for (int l = 0; l < kernel; l++)
                          {
                              if(j==0)
                                  W_temp[z]=B.getValue(x,y);
                              if(z==temp_size)
                                  z=0;
                              X_temp[z]=Pad[x][y];;
                              z++;
                              y++;
                          }
                          x++;
                          y = j;
                      }
                      if(i>0)
                      {
                          if(z==temp_size)
                              z=0;
                          X_temp[z]=Pad[x][y];;
                          y++;
                          z++;
                      }
                  }
                  if(i>0)
                  {
                      x++;
                      barrel_op((vint*)X_temp, z, temp_size, (vint*)X_final);
                  }
                  if(i==0)
                      convolute += dot_product((vint*)X_temp, (vint*)W_temp);
                  else
                      convolute += dot_product( (vint*)X_final, (vint*)W_temp);
                  final_result = relu_gradient(convolute);
                 D.setValues(final_result, b, a);
                  convolute = 0;
                  b++;
              }
              b=0;
              a++;
          }
      }

 if(padding == 0)
     {
        for (int j = 0; j < stride*output; j+=stride)
        {
             x=0;
              for (int i = 0; i < stride*output; i+=stride)
              {
                  y = j;
             for (int k = 0; k < kernel; k++)
             {
                if(i==0)
                      {
                          for (int l = 0; l < kernel; l++)
                          {
                              if(j==0)
                                  W_temp[z] = B.getValue(x,y);
                      if(z==temp_size)
                                  z=0;
                      X_temp[z] = A.getValue(x,y);
                              z++;
                              y++;
                          }
                          x++;
                          y = j;
                      }
                      if(i>0)
                      {
                            if(z==temp_size)
                                z=0;
                            X_temp[z] = A.getValue(x,y);
                            y++;
                            z++;
                        }
                    }
            if(i>0)
            {
                x++;
                barrel_op((vint*)X_temp, z, temp_size, (vint*)X_final);
            }
            if(i==0)
                convolute += dot_product((vint*)X_temp, (vint*)W_temp);
            else
                convolute += dot_product( (vint*)X_final, (vint*)W_temp);
                  final_result = relu_gradient(convolute);
            D.setValues(final_result , b, a);
            convolute = 0;
            b++;
              }
              b=0;
              a++;
          }
      }
  return output;
}

// Max Pooling
int findMax(int mat[][N])
{
    int maxElement = 0;
    for (int i = 0; i < N; i++)
    {
        for (int j = 0; j < N; j++)
        {
            if (mat[i][j] > maxElement)
                maxElement = mat[i][j];
        }
    }
    return maxElement;
}

// Average Pooling
int findMean(int mat[][N])
{
    int sum = 0;
    for (int i=0; i<N; i++)
       for (int j=0; j<N; j++)
          sum += mata[i][j];
    return sum/(N*N);
}

// Pooling function.
int Pooling(Image &pool_H, Image &pool_D, int size, int kernel, int stride)
{
    int H_temp[2][2];
    int x, y, max, a=0, b=0;
    int output = size/stride;
  for (int i = 0; i < stride*output; i+=stride)
  {
    for (int j = 0; j < stride*output; j+=stride)
    {
      x = i;
      y = j;
      for (int k = 0; k < kernel; k++)
      {
        for (int l = 0; l < kernel; l++)
        {
                    H_temp[k][l]=pool_H.getValue(x,y);
          y++;
        }
        max = findMax(H_temp);
        x++;
        y = j;
      }
      pool_D.setValues(max, a, b);
      max = 0;
            b++;
    }
    b=0;
    a++;
  }
  return output;
}

// Fully Connected Layer
void Fully_Connected_Layer(Image A[], int B[][SIZE], int D[][SIZE], int channel, int
     input_neuron, int output_neuron)
{
    int product = 0, final_result = 0, a=0, b=0;
    for(int i=0; i<output_neuron; i++)
    {
        for(int j=0; j<input_neuron; j++)
        {
            for(int k=0; k<channel; k++)
                product += dot_product((vint*)A[k].image[j], (vint*)B[j]);
        }
        final_result = relu_gradient(product+1);
       D[i][b] = final_result;
        final_result = 0;
        product = 0;
        b++;
    }
}

void Layer(Image A[], Image B[], Image C[], Image D[], int channel, int filter, int &image_size,
      int conv_kernel, int conv_stride, int conv_padding, int pool_kernel, int pool_stride, int
     num_layer)
{
    int conv_size = 0;
    if(num_layer == 1)
    {
        for (int p = 0; p < channel; p++)
             A[p].init_image(image_size, p);
    }
    for (int p = 0; p < channel; p++)
    {
         for (int t = 0; t < filter; t++)
            B[t].init_image(conv_kernel, t);
    }
    //CONVOLUTION
    for (int i = 0; i < channel; i++)
    {
         for (int j = 0; j < filter; j++)
            conv_size = Convolution(image_size, conv_kernel, conv_stride, conv_padding, A[i], B[
     j], C[j]);
    }
    //POOLING
    for (int j = 0; j < filter; j++)
        image_size = Pooling(C[j], D[j], conv_size, pool_kernel, pool_stride);
}



int main()
{
    Image H[filter_1];
    Image Y[filter_1];
    Image X[channel_1];
    Image W[filter_1];

    // Defs for convolution
    int image_size = 18;
    int conv_stride = 1, conv_kernel = 3, conv_padding = 0;

    // Defs for pooling
    int pool_stride = 2, pool_kernel = 2;

    // Defs for fully connected
    int kernel_size = 3;
    int num_hidden_neuron = 1, num_output_neuron = 1;
    int vector_element_first = 3, vector_element_second = 3;

    // Initialization of X and weights
    init(image_size, 3, vector_element_first);

    // LAYER 1
    Layer(X, W, H, Y, channel_1, filter_1, image_size, conv_kernel, conv_stride, conv_padding,
     pool_stride, pool_kernel, 1);

    //LAYER 2
    Image Y_2[filter_2];
    Image H_2[filter_2];
    Image W_2[filter_2];

    Layer(Y, W_2, H_2, Y_2, channel_2, filter_2, image_size, conv_kernel,
           conv_stride, conv_padding, pool_stride, pool_kernel, 0);

    Fully_Connected_Layer(Y_2, W_FC, Y_FC, filter_2, 2, num_hidden_neuron);

    return 0;
}
```

Listing A.1: Convolutional Neural Network.

Appendix B

Instruction profiler reports

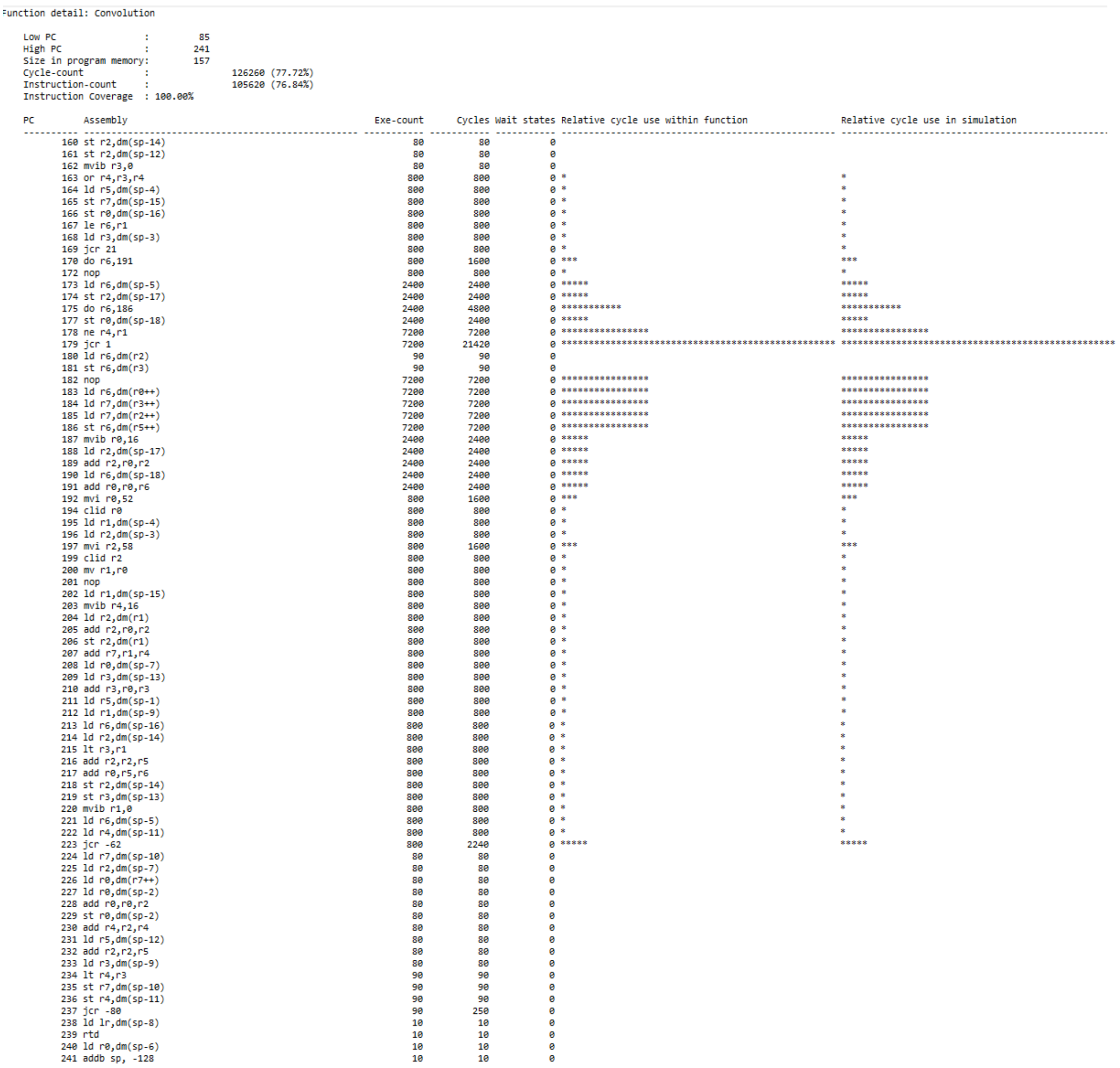
B.1 Convolution of the First Design



```
Function detail: Convolution

    Low PC               :        85
    High PC              :       233
    Size in program memory:       149
    Cycle-count          :                121920 (70.68%)
    Instruction-count    :                112300 (70.84%)
    Instruction Coverage  : 100.00%

    PC         Assembly                                   Exe-count      Cycles Wait states Relative cycle use within function                 Relative cycle use in simulation
    ---------- ---------------------------------------- ----------- ----------- ----------- -------------------------------------------------- --------------------------------------------------
           159 st r1,dm(sp-11)                                   80          80           0
           160 mvib r6,0                                         80          80           0
           161 ld r5,dm(sp-5)                                    80          80           0
           162 le r5,r7                                         800         800           0 *****                                              *****
           163 ld r7,dm(sp-7)                                   800         800           0 *****                                              *****
           164 jcr 44                                           800         800           0 *****                                              *****
           165 mvib r0,0                                        800         800           0 *****                                              *****
           166 st r0,dm(sp-16)                                  800         800           0 *****                                              *****
           167 st r2,dm(sp-15)                                  800         800           0 *****                                              *****
           168 st r6,dm(sp-14)                                  800         800           0 *****                                              *****
           169 st r1,dm(sp-13)                                  800         800           0 *****                                              *****
           170 do r5,202                                        800        1600           0 ***********                                        ***********
           172 ld r4,dm(sp-2)                                   800         800           0 *****                                              *****
           173 ld r7,dm(sp-5)                                  2400        2400           0 ****************                                   ****************
           174 st r1,dm(sp-17)                                 2400        2400           0 ****************                                   ****************
           175 mv r3,r4                                        2400        2400           0 ****************                                   ****************
           176 ld r0,dm(sp-3)                                  2400        2400           0 ****************                                   ****************
           177 do r7,184                                       2400        4800           0 *********************************                  *********************************
           179 ld r2,dm(sp-4)                                  2400        2400           0 ****************                                   ****************
           180 ld r6,dm(r3++)                                  7200        7200           0 ************************************************** **************************************************
           181 ld r5,dm(r1++)                                  7200        7200           0 ************************************************** **************************************************
           182 st r6,dm(r0++)                                  7200        7200           0 ************************************************** **************************************************
           183 st r5,dm(r2++)                                  7200        7200           0 ************************************************** **************************************************
           184 nop                                             7200        7200           0 ************************************************** **************************************************
           185 st r4,dm(sp-18)                                 2400        2400           0 ****************                                   ****************
           186 mvi r0,52                                       2400        4800           0 *********************************                  *********************************
           188 clid r0                                         2400        2400           0 ****************                                   ****************
           189 ld r2,dm(sp-3)                                  2400        2400           0 ****************                                   ****************
           190 ld r1,dm(sp-4)                                  2400        2400           0 ****************                                   ****************
           191 ld r1,dm(sp-16)                                 2400        2400           0 ****************                                   ****************
           192 add r1,r0,r1                                    2400        2400           0 ****************                                   ****************
           193 mvi r0,58                                       2400        4800           0 *********************************                  *********************************
           195 clid r0                                         2400        2400           0 ****************                                   ****************
           196 st r1,dm(sp-16)                                 2400        2400           0 ****************                                   ****************
           197 nop                                             2400        2400           0 ****************                                   ****************
           198 ld r2,dm(sp-18)                                 2400        2400           0 ****************                                   ****************
           199 mvib r1,16                                      2400        2400           0 ****************                                   ****************
           200 add r4,r1,r2                                    2400        2400           0 ****************                                   ****************
           201 ld r2,dm(sp-17)                                 2400        2400           0 ****************                                   ****************
           202 add r1,r1,r2                                    2400        2400           0 ****************                                   ****************
           203 ld r2,dm(sp-15)                                  800         800           0 *****                                              *****
           204 ld r6,dm(sp-14)                                  800         800           0 *****                                              *****
           205 ld r1,dm(sp-13)                                  800         800           0 *****                                              *****
           206 ld r5,dm(sp-5)                                   800         800           0 *****                                              *****
           207 ld r4,dm(sp-9)                                   800         800           0 *****                                              *****
           208 ld r7,dm(sp-7)                                   800         800           0 *****                                              *****
           209 mvib r3,16                                       800         800           0 *****                                              *****
           210 add r6,r6,r7                                     800         800           0 *****                                              *****
           211 ld r7,dm(r2)                                     800         800           0 *****                                              *****
```

Figure B.1: Instruction report of the first design.

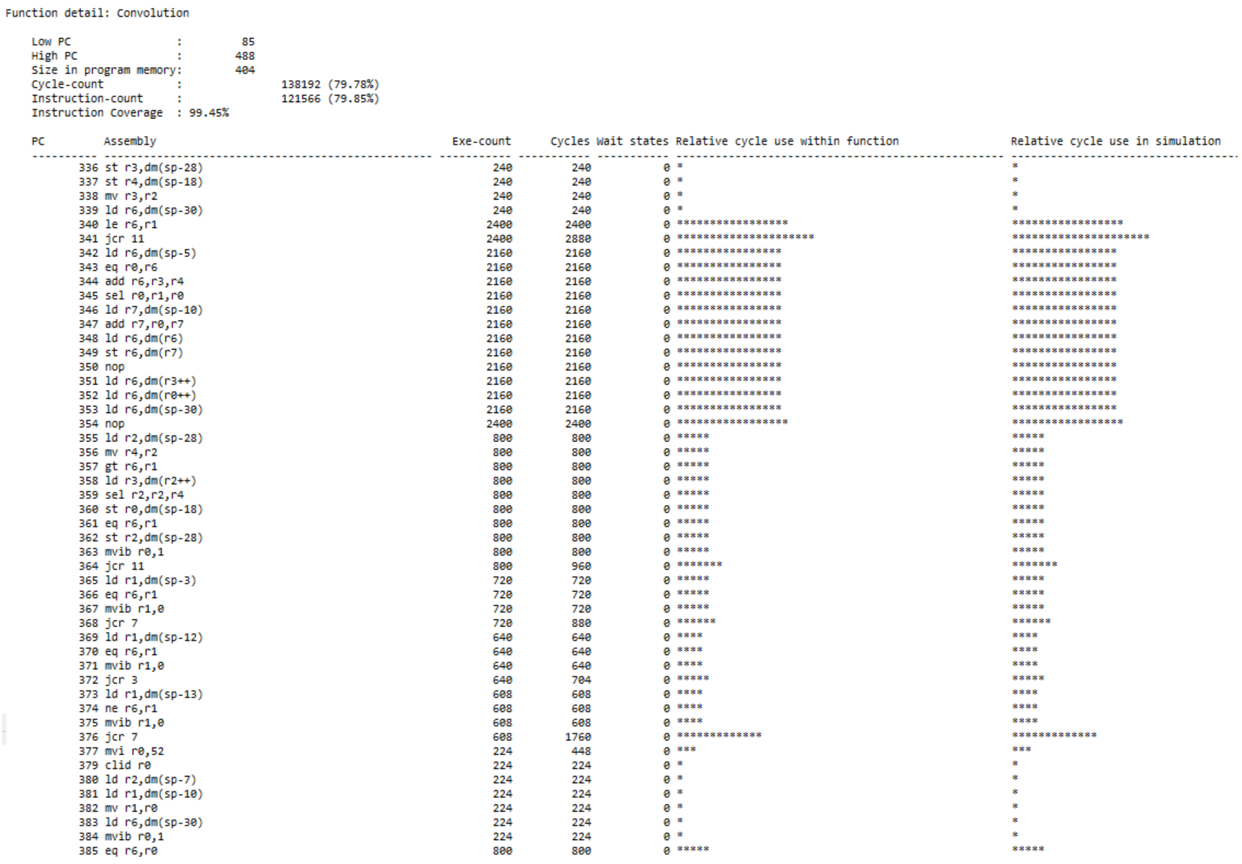
B.2 Convolution of the Second Design



```
unction detail: Convolution

   Low PC              :        85
   High PC             :       241
   Size in program memory:     157
   Cycle-count         :                 126260 (77.72%)
   Instruction-count   :                 105620 (76.84%)
   Instruction Coverage  : 100.00%

   PC        Assembly                        Exe-count      Cycles Wait states Relative cycle use within function                 Relative cycle use in simulation
   ---------- ------------------------------ ----------- ----------- ----------- -------------------------------------------------- --------------------------------------------------
          160 st r2,dm(sp-14)              80         80          0
          161 st r2,dm(sp-12)              80         80          0
          162 mvib r3,0                    80         80          0
          163 or r4,r3,r4                 800        800          0 *                                                  *
          164 ld r5,dm(sp-4)              800        800          0 *                                                  *
          165 st r7,dm(sp-15)             800        800          0 *                                                  *
          166 st r0,dm(sp-16)             800        800          0 *                                                  *
          167 le r6,r1                    800        800          0 *                                                  *
          168 ld r3,dm(sp-3)              800        800          0 *                                                  *
          169 jcr 21                      800        800          0 *                                                  *
          170 do r6,191                   800       1600          0 ***                                                ***
          172 nop                         800        800          0 *                                                  *
          173 ld r6,dm(sp-5)             2400       2400          0 *****                                              *****
          174 st r2,dm(sp-17)            2400       2400          0 *****                                              *****
          175 do r6,186                  2400       4800          0 ***********                                        ***********
          177 st r0,dm(sp-18)            2400       2400          0 *****                                              *****
          178 ne r4,r1                   7200       7200          0 ****************                                   ****************
          179 jcr 1                      7200      21420          0 ************************************************** **************************************************
          180 ld r6,dm(r2)                 90         90          0
          181 st r6,dm(r3)                 90         90          0
          182 nop                        7200       7200          0 ****************                                   ****************
          183 ld r6,dm(r0++)             7200       7200          0 ****************                                   ****************
          184 ld r7,dm(r3++)             7200       7200          0 ****************                                   ****************
          185 ld r7,dm(r2++)             7200       7200          0 ****************                                   ****************
          186 st r6,dm(r5++)             7200       7200          0 ****************                                   ****************
          187 mvib r0,16                 2400       2400          0 *****                                              *****
          188 ld r2,dm(sp-17)            2400       2400          0 *****                                              *****
          189 add r2,r0,r2               2400       2400          0 *****                                              *****
          190 ld r6,dm(sp-18)            2400       2400          0 *****                                              *****
          191 add r0,r0,r6               2400       2400          0 *****                                              *****
          192 mvi r0,52                   800       1600          0 ***                                                ***
          194 clid r0                     800        800          0 *                                                  *
          195 ld r1,dm(sp-4)              800        800          0 *                                                  *
          196 ld r2,dm(sp-3)              800        800          0 *                                                  *
          197 mvi r2,58                   800       1600          0 ***                                                ***
          199 clid r2                     800        800          0 *                                                  *
          200 mv r1,r0                    800        800          0 *                                                  *
          201 nop                         800        800          0 *                                                  *
          202 ld r1,dm(sp-15)             800        800          0 *                                                  *
          203 mvib r4,16                  800        800          0 *                                                  *
          204 ld r2,dm(r1)                800        800          0 *                                                  *
          205 add r2,r0,r2                800        800          0 *                                                  *
          206 st r2,dm(r1)                800        800          0 *                                                  *
          207 add r7,r1,r4                800        800          0 *                                                  *
          208 ld r0,dm(sp-7)              800        800          0 *                                                  *
          209 ld r3,dm(sp-13)             800        800          0 *                                                  *
          210 add r3,r0,r3                800        800          0 *                                                  *
          211 ld r5,dm(sp-1)              800        800          0 *                                                  *
          212 ld r1,dm(sp-9)              800        800          0 *                                                  *
          213 ld r6,dm(sp-16)             800        800          0 *                                                  *
          214 ld r2,dm(sp-14)             800        800          0 *                                                  *
          215 lt r3,r1                    800        800          0 *                                                  *
          216 add r2,r2,r5                800        800          0 *                                                  *
          217 add r0,r5,r6                800        800          0 *                                                  *
          218 st r2,dm(sp-14)             800        800          0 *                                                  *
          219 st r3,dm(sp-13)             800        800          0 *                                                  *
          220 mvib r1,0                   800        800          0 *                                                  *
          221 ld r6,dm(sp-5)              800        800          0 *                                                  *
          222 ld r4,dm(sp-11)             800        800          0 *                                                  *
          223 jcr -62                     800       2240          0 *****                                              *****
          224 ld r7,dm(sp-10)              80         80          0
          225 ld r2,dm(sp-7)               80         80          0
          226 ld r0,dm(r7++)               80         80          0
          227 ld r0,dm(sp-2)               80         80          0
          228 add r0,r0,r2                 80         80          0
          229 st r0,dm(sp-2)               80         80          0
          230 add r4,r2,r4                 80         80          0
          231 ld r5,dm(sp-12)              80         80          0
          232 add r2,r2,r5                 80         80          0
          233 ld r3,dm(sp-9)               80         80          0
          234 lt r4,r3                     90         90          0
          235 st r7,dm(sp-10)              90         90          0
          236 st r4,dm(sp-11)              90         90          0
          237 jcr -80                      90        250          0
          238 ld lr,dm(sp-8)               10         10          0
          239 rtd                          10         10          0
          240 ld r0,dm(sp-6)               10         10          0
          241 addb sp, -128                10         10          0
```

Figure B.2: Instruction report of the second design.

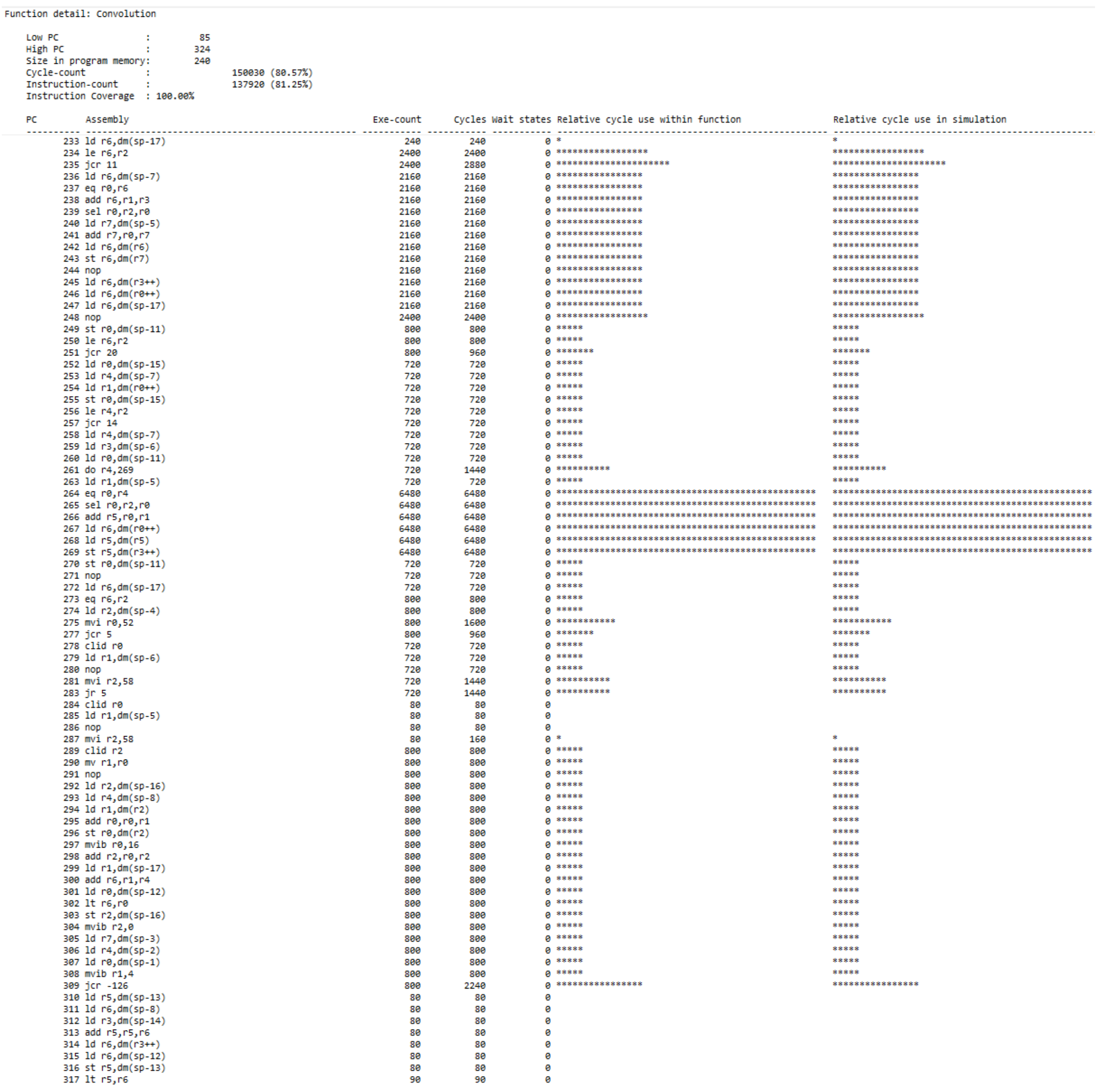
B.3 Convolution of the Third Design



```
Function detail: Convolution

    Low PC              :        85
    High PC             :       488
    Size in program memory:     404
    Cycle-count         :              138192 (79.78%)
    Instruction-count   :              121566 (79.85%)
    Instruction Coverage  : 99.45%

    PC         Assembly                                  Exe-count     Cycles Wait states Relative cycle use within function Relative cycle use in simulation
    ---------- ---------------------------------------- ---------- ---------- ---------- ------------------------------ ----------------------------------
           336 st r3,dm(sp-28)                                 240        240          0 *                              *
           337 st r4,dm(sp-18)                                 240        240          0 *                              *
           338 mv r3,r2                                        240        240          0 *                              *
           339 ld r6,dm(sp-30)                                 240        240          0 *                              *
           340 le r6,r1                                       2400       2400          0 *****************              *****************
           341 jcr 11                                         2400       2880          0 *********************          *********************
           342 ld r6,dm(sp-5)                                 2160       2160          0 ****************               ****************
           343 eq r0,r6                                       2160       2160          0 ****************               ****************
           344 add r6,r3,r4                                   2160       2160          0 ****************               ****************
           345 sel r0,r1,r0                                   2160       2160          0 ****************               ****************
           346 ld r7,dm(sp-10)                                2160       2160          0 ****************               ****************
           347 add r7,r0,r7                                   2160       2160          0 ****************               ****************
           348 ld r6,dm(r6)                                   2160       2160          0 ****************               ****************
           349 st r6,dm(r7)                                   2160       2160          0 ****************               ****************
           350 nop                                            2160       2160          0 ****************               ****************
           351 ld r6,dm(r3++)                                 2160       2160          0 ****************               ****************
           352 ld r6,dm(r0++)                                 2160       2160          0 ****************               ****************
           353 ld r6,dm(sp-30)                                2160       2160          0 ****************               ****************
           354 nop                                            2400       2400          0 *****************              *****************
           355 ld r2,dm(sp-28)                                 800        800          0 *****                          *****
           356 mv r4,r2                                        800        800          0 *****                          *****
           357 gt r6,r1                                        800        800          0 *****                          *****
           358 ld r3,dm(r2++)                                  800        800          0 *****                          *****
           359 sel r2,r2,r4                                    800        800          0 *****                          *****
           360 st r0,dm(sp-18)                                 800        800          0 *****                          *****
           361 eq r6,r1                                        800        800          0 *****                          *****
           362 st r2,dm(sp-28)                                 800        800          0 *****                          *****
           363 mvib r0,1                                       800        800          0 *****                          *****
           364 jcr 11                                          800        960          0 *******                        *******
           365 ld r1,dm(sp-3)                                  720        720          0 *****                          *****
           366 eq r6,r1                                        720        720          0 *****                          *****
           367 mvib r1,0                                       720        720          0 *****                          *****
           368 jcr 7                                           720        880          0 ******                         ******
           369 ld r1,dm(sp-12)                                 640        640          0 ****                           ****
           370 eq r6,r1                                        640        640          0 ****                           ****
           371 mvib r1,0                                       640        640          0 ****                           ****
           372 jcr 3                                           640        704          0 *****                          *****
           373 ld r1,dm(sp-13)                                 608        608          0 ****                           ****
           374 ne r6,r1                                        608        608          0 ****                           ****
           375 mvib r1,0                                       608        608          0 ****                           ****
           376 jcr 7                                           608       1760          0 *************                  *************
           377 mvi r0,52                                       224        448          0 ***                            ***
           379 clid r0                                         224        224          0 *                              *
           380 ld r2,dm(sp-7)                                  224        224          0 *                              *
           381 ld r1,dm(sp-10)                                 224        224          0 *                              *
           382 mv r1,r0                                        224        224          0 *                              *
           383 ld r6,dm(sp-30)                                 224        224          0 *                              *
           384 mvib r0,1                                       224        224          0 *                              *
           385 eq r6,r0                                        800        800          0 *****                          *****
```

Figure B.3: Instruction report of the third design.

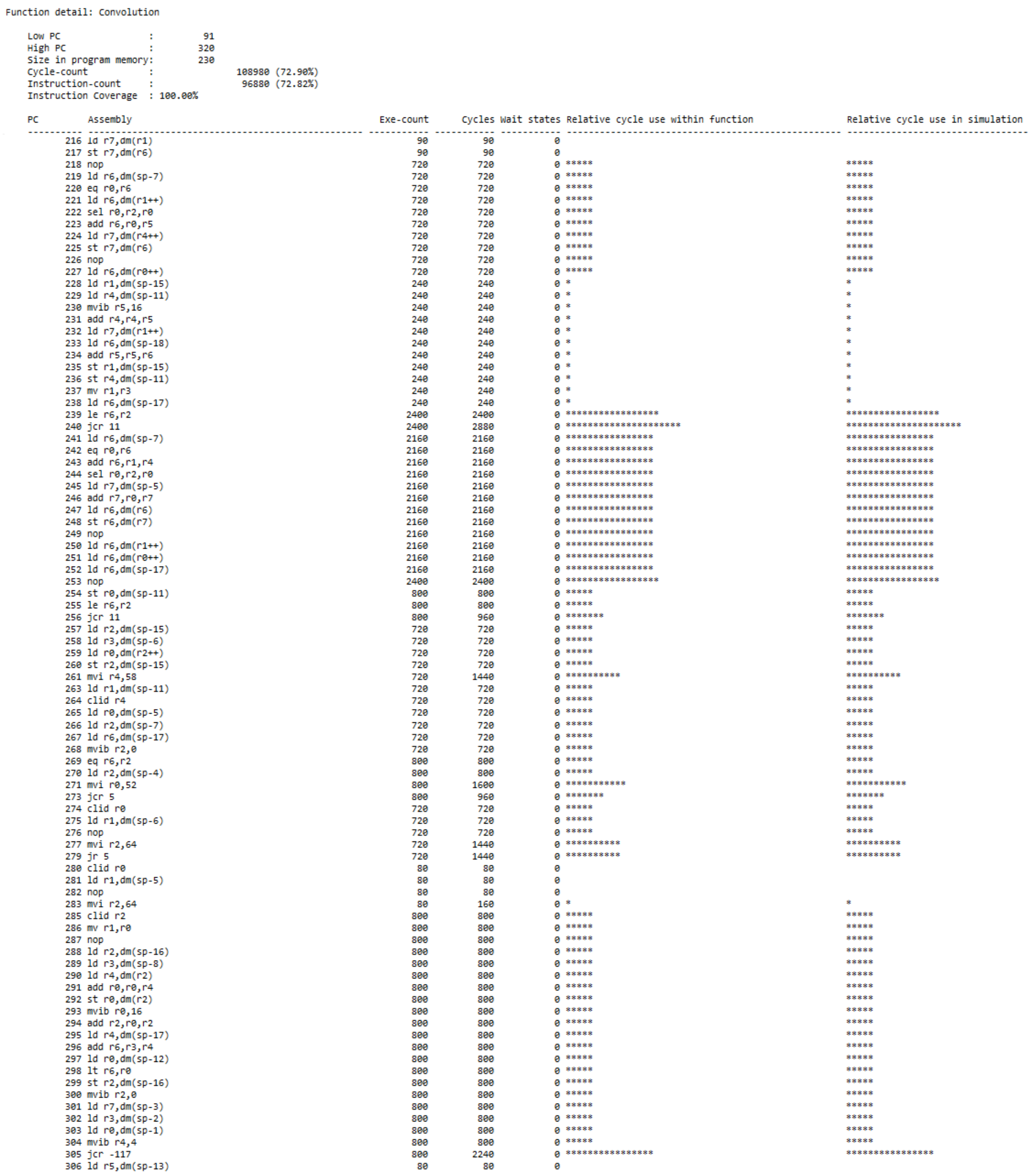
B.4 Convolution of the Fourth Design



```
Function detail: Convolution

    Low PC                 :         85
    High PC                :        324
    Size in program memory:         240
    Cycle-count            :                 150030 (80.57%)
    Instruction-count      :                 137920 (81.25%)
    Instruction Coverage   : 100.00%

    PC         Assembly                                              Exe-count      Cycles Wait states Relative cycle use within function                 Relative cycle use in simulation
    ---------- ---------------------------------------------------- ----------- ----------- ----------- -------------------------------------------------- --------------------------------------------------
           233 ld r6,dm(sp-17)                                              240         240           0 *                                                  *
           234 le r6,r2                                                    2400        2400           0 *****************                                  *****************
           235 jcr 11                                                      2400        2880           0 *********************                              *********************
           236 ld r6,dm(sp-7)                                              2160        2160           0 ****************                                   ****************
           237 eq r0,r6                                                    2160        2160           0 ****************                                   ****************
           238 add r6,r1,r3                                                2160        2160           0 ****************                                   ****************
           239 sel r0,r2,r0                                                2160        2160           0 ****************                                   ****************
           240 ld r7,dm(sp-5)                                              2160        2160           0 ****************                                   ****************
           241 add r7,r0,r7                                                2160        2160           0 ****************                                   ****************
           242 ld r6,dm(r6)                                                2160        2160           0 ****************                                   ****************
           243 st r6,dm(r7)                                                2160        2160           0 ****************                                   ****************
           244 nop                                                         2160        2160           0 ****************                                   ****************
           245 ld r6,dm(r3++)                                              2160        2160           0 ****************                                   ****************
           246 ld r6,dm(r0++)                                              2160        2160           0 ****************                                   ****************
           247 ld r6,dm(sp-17)                                             2160        2160           0 ****************                                   ****************
           248 nop                                                         2400        2400           0 *****************                                  *****************
           249 st r0,dm(sp-11)                                              800         800           0 *****                                              *****
           250 le r6,r2                                                     800         800           0 *****                                              *****
           251 jcr 20                                                       800         960           0 *******                                            *******
           252 ld r0,dm(sp-15)                                              720         720           0 *****                                              *****
           253 ld r4,dm(sp-7)                                               720         720           0 *****                                              *****
           254 ld r1,dm(r0++)                                               720         720           0 *****                                              *****
           255 st r0,dm(sp-15)                                              720         720           0 *****                                              *****
           256 le r4,r2                                                     720         720           0 *****                                              *****
           257 jcr 14                                                       720         720           0 *****                                              *****
           258 ld r4,dm(sp-7)                                               720         720           0 *****                                              *****
           259 ld r3,dm(sp-6)                                               720         720           0 *****                                              *****
           260 ld r0,dm(sp-11)                                              720         720           0 *****                                              *****
           261 do r4,269                                                    720        1440           0 **********                                         **********
           263 ld r1,dm(sp-5)                                               720         720           0 *****                                              *****
           264 eq r0,r4                                                    6480        6480           0 ************************************************   ************************************************
           265 sel r0,r2,r0                                                6480        6480           0 ************************************************   ************************************************
           266 add r5,r0,r1                                                6480        6480           0 ************************************************   ************************************************
           267 ld r6,dm(r0++)                                              6480        6480           0 ************************************************   ************************************************
           268 ld r5,dm(r5)                                                6480        6480           0 ************************************************   ************************************************
           269 st r5,dm(r3++)                                              6480        6480           0 ************************************************   ************************************************
           270 st r0,dm(sp-11)                                              720         720           0 *****                                              *****
           271 nop                                                          720         720           0 *****                                              *****
           272 ld r6,dm(sp-17)                                              720         720           0 *****                                              *****
           273 eq r6,r2                                                     800         800           0 *****                                              *****
           274 ld r2,dm(sp-4)                                               800         800           0 *****                                              *****
           275 mvi r0,52                                                    800        1600           0 ***********                                        ***********
           277 jcr 5                                                        800         960           0 *******                                            *******
           278 clid r0                                                      720         720           0 *****                                              *****
           279 ld r1,dm(sp-6)                                               720         720           0 *****                                              *****
           280 nop                                                          720         720           0 *****                                              *****
           281 mvi r2,58                                                    720        1440           0 **********                                         **********
           283 jr 5                                                         720        1440           0 **********                                         **********
           284 clid r0                                                       80          80           0
           285 ld r1,dm(sp-5)                                                80          80           0
           286 nop                                                           80          80           0
           287 mvi r2,58                                                     80         160           0 *                                                  *
           289 clid r2                                                      800         800           0 *****                                              *****
           290 mv r1,r0                                                     800         800           0 *****                                              *****
           291 nop                                                          800         800           0 *****                                              *****
           292 ld r2,dm(sp-16)                                              800         800           0 *****                                              *****
           293 ld r4,dm(sp-8)                                               800         800           0 *****                                              *****
           294 ld r1,dm(r2)                                                 800         800           0 *****                                              *****
           295 add r0,r0,r1                                                 800         800           0 *****                                              *****
           296 st r0,dm(r2)                                                 800         800           0 *****                                              *****
           297 mvib r0,16                                                   800         800           0 *****                                              *****
           298 add r2,r0,r2                                                 800         800           0 *****                                              *****
           299 ld r1,dm(sp-17)                                              800         800           0 *****                                              *****
           300 add r6,r1,r4                                                 800         800           0 *****                                              *****
           301 ld r0,dm(sp-12)                                              800         800           0 *****                                              *****
           302 lt r6,r0                                                     800         800           0 *****                                              *****
           303 st r2,dm(sp-16)                                              800         800           0 *****                                              *****
           304 mvib r2,0                                                    800         800           0 *****                                              *****
           305 ld r7,dm(sp-3)                                               800         800           0 *****                                              *****
           306 ld r4,dm(sp-2)                                               800         800           0 *****                                              *****
           307 ld r0,dm(sp-1)                                               800         800           0 *****                                              *****
           308 mvib r1,4                                                    800         800           0 *****                                              *****
           309 jcr -126                                                     800        2240           0 ****************                                   ****************
           310 ld r5,dm(sp-13)                                               80          80           0
           311 ld r6,dm(sp-8)                                                80          80           0
           312 ld r3,dm(sp-14)                                               80          80           0
           313 add r5,r5,r6                                                  80          80           0
           314 ld r6,dm(r3++)                                                80          80           0
           315 ld r6,dm(sp-12)                                               80          80           0
           316 st r5,dm(sp-13)                                               80          80           0
           317 lt r5,r6                                                      90          90           0
```

Figure B.4: Instruction report of the fourth design.

B.5 Convolution of the Fifth Design



```
Function detail: Convolution

    Low PC                :         91
    High PC               :        320
    Size in program memory:        230
    Cycle-count           :                108980 (72.90%)
    Instruction-count     :                 96880 (72.82%)
    Instruction Coverage  : 100.00%

    PC         Assembly                                             Exe-count      Cycles Wait states Relative cycle use within function                 Relative cycle use in simulation
    ---------- -------------------------------------------------- ----------- ----------- ----------- -------------------------------------------------- ---------------------------------
           216 ld r7,dm(r1)                                                90          90           0
           217 st r7,dm(r6)                                                90          90           0
           218 nop                                                        720         720           0 *****                                              *****
           219 ld r6,dm(sp-7)                                             720         720           0 *****                                              *****
           220 eq r0,r6                                                   720         720           0 *****                                              *****
           221 ld r6,dm(r1++)                                             720         720           0 *****                                              *****
           222 sel r0,r2,r0                                               720         720           0 *****                                              *****
           223 add r6,r0,r5                                               720         720           0 *****                                              *****
           224 ld r7,dm(r4++)                                             720         720           0 *****                                              *****
           225 st r7,dm(r6)                                               720         720           0 *****                                              *****
           226 nop                                                        720         720           0 *****                                              *****
           227 ld r6,dm(r0++)                                             720         720           0 *****                                              *****
           228 ld r1,dm(sp-15)                                            240         240           0 *                                                  *
           229 ld r4,dm(sp-11)                                            240         240           0 *                                                  *
           230 mvib r5,16                                                 240         240           0 *                                                  *
           231 add r4,r4,r5                                               240         240           0 *                                                  *
           232 ld r7,dm(r1++)                                             240         240           0 *                                                  *
           233 ld r6,dm(sp-18)                                            240         240           0 *                                                  *
           234 add r5,r5,r6                                               240         240           0 *                                                  *
           235 st r1,dm(sp-15)                                            240         240           0 *                                                  *
           236 st r4,dm(sp-11)                                            240         240           0 *                                                  *
           237 mv r1,r3                                                   240         240           0 *                                                  *
           238 ld r6,dm(sp-17)                                            240         240           0 *                                                  *
           239 le r6,r2                                                  2400        2400           0 *****************                                  *****************
           240 jcr 11                                                    2400        2880           0 *********************                              *********************
           241 ld r6,dm(sp-7)                                            2160        2160           0 ****************                                   ****************
           242 eq r0,r6                                                  2160        2160           0 ****************                                   ****************
           243 add r6,r1,r4                                              2160        2160           0 ****************                                   ****************
           244 sel r0,r2,r0                                              2160        2160           0 ****************                                   ****************
           245 ld r7,dm(sp-5)                                            2160        2160           0 ****************                                   ****************
           246 add r7,r0,r7                                              2160        2160           0 ****************                                   ****************
           247 ld r6,dm(r6)                                              2160        2160           0 ****************                                   ****************
           248 st r6,dm(r7)                                              2160        2160           0 ****************                                   ****************
           249 nop                                                       2160        2160           0 ****************                                   ****************
           250 ld r6,dm(r1++)                                            2160        2160           0 ****************                                   ****************
           251 ld r6,dm(r0++)                                            2160        2160           0 ****************                                   ****************
           252 ld r6,dm(sp-17)                                           2160        2160           0 ****************                                   ****************
           253 nop                                                       2400        2400           0 *****************                                  *****************
           254 st r0,dm(sp-11)                                            800         800           0 *****                                              *****
           255 le r6,r2                                                   800         800           0 *****                                              *****
           256 jcr 11                                                     800         960           0 *******                                            *******
           257 ld r2,dm(sp-15)                                            720         720           0 *****                                              *****
           258 ld r3,dm(sp-6)                                             720         720           0 *****                                              *****
           259 ld r0,dm(r2++)                                             720         720           0 *****                                              *****
           260 st r2,dm(sp-15)                                            720         720           0 *****                                              *****
           261 mvi r4,58                                                  720        1440           0 **********                                         **********
           263 ld r1,dm(sp-11)                                            720         720           0 *****                                              *****
           264 clid r4                                                    720         720           0 *****                                              *****
           265 ld r0,dm(sp-5)                                             720         720           0 *****                                              *****
           266 ld r2,dm(sp-7)                                             720         720           0 *****                                              *****
           267 ld r6,dm(sp-17)                                            720         720           0 *****                                              *****
           268 mvib r2,0                                                  720         720           0 *****                                              *****
           269 eq r6,r2                                                   800         800           0 *****                                              *****
           270 ld r2,dm(sp-4)                                             800         800           0 *****                                              *****
           271 mvi r0,52                                                  800        1600           0 ***********                                        ***********
           273 jcr 5                                                      800         960           0 *******                                            *******
           274 clid r0                                                    720         720           0 *****                                              *****
           275 ld r1,dm(sp-6)                                             720         720           0 *****                                              *****
           276 nop                                                        720         720           0 *****                                              *****
           277 mvi r2,64                                                  720        1440           0 **********                                         **********
           279 jr 5                                                       720        1440           0 **********                                         **********
           280 clid r0                                                     80          80           0
           281 ld r1,dm(sp-5)                                              80          80           0
           282 nop                                                         80          80           0
           283 mvi r2,64                                                   80         160           0 *                                                  *
           285 clid r2                                                    800         800           0 *****                                              *****
           286 mv r1,r0                                                   800         800           0 *****                                              *****
           287 nop                                                        800         800           0 *****                                              *****
           288 ld r2,dm(sp-16)                                            800         800           0 *****                                              *****
           289 ld r3,dm(sp-8)                                             800         800           0 *****                                              *****
           290 ld r4,dm(r2)                                               800         800           0 *****                                              *****
           291 add r0,r0,r4                                               800         800           0 *****                                              *****
           292 st r0,dm(r2)                                               800         800           0 *****                                              *****
           293 mvib r0,16                                                 800         800           0 *****                                              *****
           294 add r2,r0,r2                                               800         800           0 *****                                              *****
           295 ld r4,dm(sp-17)                                            800         800           0 *****                                              *****
           296 add r6,r3,r4                                               800         800           0 *****                                              *****
           297 ld r0,dm(sp-12)                                            800         800           0 *****                                              *****
           298 lt r6,r0                                                   800         800           0 *****                                              *****
           299 st r2,dm(sp-16)                                            800         800           0 *****                                              *****
           300 mvib r2,0                                                  800         800           0 *****                                              *****
           301 ld r7,dm(sp-3)                                             800         800           0 *****                                              *****
           302 ld r3,dm(sp-2)                                             800         800           0 *****                                              *****
           303 ld r0,dm(sp-1)                                             800         800           0 *****                                              *****
           304 mvib r4,4                                                  800         800           0 *****                                              *****
           305 jcr -117                                                   800        2240           0 ****************                                   ****************
           306 ld r5,dm(sp-13)                                             80          80           0
```

Figure B.5: Instruction report of the fifth design.

Bibliography

[1] V. Sze, Y. Chen, T. Yang, and J. S. Emer, *"Efficient processing of deep neural networks: A tutorial and survey,"* Proceedings of the IEEE, vol. 105, pp. 2295–2329, Dec 2017.

[2] A. Ng, "Machine learning on coursera."

[3] A. Ng, "Deep learning on coursera."

[4] Y. LeCun, et al., *"Handwritten digit recognition: Applications of neural network chips and automatic learning,"* IEEE Commun. Mag., vol. 27, no. 11, pp. 41–46, Nov. 1989.

[5] A. Krizhevsky, I. Sutskever, and G. E. Hinton, *"Imagenet classification with deep convolutional neural networks,"* Neural Information Processing Systems, vol. 25, 01 2012. W

[6] K. Simonyan and A. Zisserman, *"Very deep convolutional networks for large-scale image recognition,"* in Proc. ICLR, 2015.

[7] C. Szegedy, et al., *"Going deeper with convolutions,"* in Proc. CVPR, 2015, pp. 1–9.

[8] K. He, X. Zhang, S. Ren, and J. Sun, *"Deep residual learning for image recognition,"* in Proc. CVPR, 2016, pp. 770–778.

[9] "ASIP Designer: Design Tool for Application-Specific Instruction-Set Processors", Synopsys `https://www.synopsys.com/dw/ipdir.php?ds=asip-designer/`

[10] "Tmicro Core Processor Manual-ASIP Designer." Synopsys, Version M-2017.03.

[11] "Tvec core-ASIP Designer." Synopsys, Version L-2016.03.

[12] "The nML Processor Description Language." Synopsys, Version N-2017.09

[13] "Go User Manual nML to synthesizable HDL translation-ASIP Designer." Synopsys, Version N-2017.09.

[14] "Target Tmotion core-ASIP Designer." Synopsys, K-2015.12.

[15] K. He, X. Zhang, S. Ren, and J. Sun, *"Deep residual learning for image recognition,"* CoRR, vol. abs/1512.03385, 2015.

[16] "Convolutional Neural Networks for Visual Recognition", http://cs231n.github.io/convolutional-networks/

[17] Wikipedia, "Artificial neural network."

[18] Wikipedia, "Arthur Samuel."

[19] Matlab, "Introduction to deep learning: What are convolutional neural networks?."

[20] S. Barter, "Convolutional neural net in tensorflow."

[21] Tiny-dnn, free deep learning library written in C++. https://github.com/tiny-dnn/tiny-dnn

[22] "Biological and artificial neurons." https://www.sciencedirect.com/topics/engineering/neurons

[23] "Applied Deep Learning: Artificial Neural Networks." https://towardsdatascience.com/applied-deep-learning-artificial-neural-networks

[24] "Inference: The Next Step in GPU-Accelerated Deep Learning." https://devblogs.nvidia.com/inference-next-step-gpu-accelerated-deep-learning/

[25] Djork-Arné Clevert, Thomas Unterthiner, Sepp Hochreiter, "Fast and Accurate Deep Network Learning by Exponential Linear Units (ELUs)," Nov 2015.

[26] Robert Hecht - Nielsen, "Neural Networks for Perception," 1992.