POLITECNICO DI TORINO

Corso di Laurea Magistrale in Ingegneria Elettronica

Tesi di Laurea Magistrale

# Design of a Reconfigurable In-Memory Neural Network Accelerator


Relatori:
Prof. Maurizio Zamboni
Prof. Marco Vacca
Prof. Giovanna Turvani

Candidato:
Maurizio Spada

Dicembre 2019

# Acknowledgments

I would like to give special thanks to my parents and my brothers for supporting me during all these years and for believing in me.

I also thank all the people who helped me, especially during the most difficult moments.

I would like to thank all the professors I had during these years and in particular prof. Maurizio Zamboni, prof. Marco Vacca, prof. Mariagrazia Graziano and prof. Giovanna Turvani who encouraged me and helped me to express all my potential for this master's project.

I also want to thank Andrea Coluccio who supported me and helped me during these last months.

# Abstract

Neural networks (NNs) are nowadays widely used for many applications like speech and image recognition. This kind of neural networks has already exceeded human accuracy in many domains at the cost of high complexity from a computational point of view.

Usually, CPU and GPU are used to implement such algorithms. GPUs perform better than CPUs thanks to their high number of cores and their wider buses which allow increasing the throughput to the memory. However, even though GPUs can reach very high-performances for data-oriented algorithms, they consume a lot of power.

For low power applications, GPUs are out of the question, so neural network accelerators are designed to obtain high performances with low power consumption. This thesis work can be divided into two parts: in the first one the AlexNet algorithm is implemented on CPU and GPU and performances are compared.

In the second part, a reconfigurable In-Memory neural network accelerator is presented. This accelerator allows implementing the most common neural networks layers and it is characterized by very low power consumption.

The word "In-Memory" means the capability of executing simple logic operations inside the memory. An architecture based on Logic-in-Memory has several benefits mainly related to the possibility of executing some operations inside the memory itself without wasting energy for transferring data from the memory to the computational units.

## Hardware architecture

The hardware accelerator high-level scheme is shown in Figure 1.

The most important layers executed by neural networks have been implemented. In particular, the same hardware has been shared to execute convolutional, fully-connected and normalization layers. The ReLU activation function has been implemented with a multiplexer with two inputs which is driven by an AND gate.

The pooling layer, instead, requires dedicated hardware to compute the maximum value among a certain number of inputs.

In this project, two memories have been used to store data and weights.

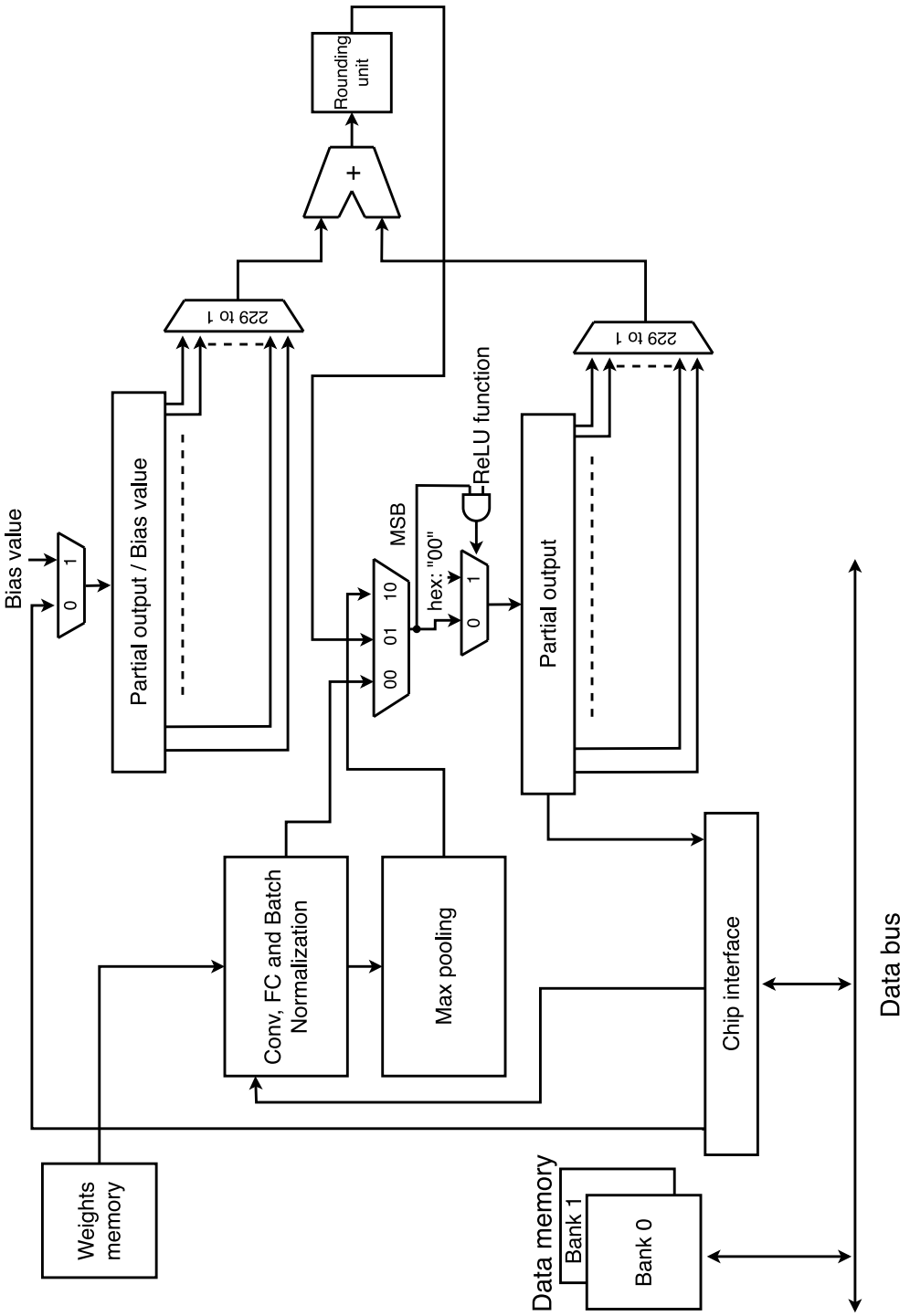


Figure 1: Hardware accelerator high-level scheme.

## Performance

This hardware accelerator is characterized by a very low power consumption with the possibility to increase its throughput replicating multiple times the whole structure.

Theoretically by multiplying the whole hardware (including the memory) n-times, it is possible to increase the performances on the single image being elaborated or it is possible to recognize multiple images simultaneously. In the latter case, the performance improvement is equal to the number of time the structure is replicated.

The synthesis has been performed with both 45 nm and 28 nm technological nodes and the results shown in Table 1 have been obtained:

| | **45 nm (max freq.)** | **28 nm (f = 285.71 MHz)** | **28 nm (max freq.)** |
|---|---|---|---|
| Area [$mm^2$] | 0.195 | 0.142 | 0.142 |
| Power [mW] | 41.89 | 22.42 | 43.69 |
| Frequency [MHz] | 285.71 | 285.71 | 588.23 |

Table 1: Synthesis results.

By using more advanced technologies the performances can be improved in terms of both maximum frequency and power.

After the synthesis the place & route has been executed using Cadence Innovus with the 45 nm technology.

The results are reported in Table 2:

| | **Post place & route** |
|---|---|
| Area [$mm^2$] | 0.190 |
| Power [mW] | 169.1 |
| Frequency [MHz] | 270.27 |

Table 2: Post place & route results.

The area and the frequency post place & route have decreased a little bit while the power consumption has increased by 4 times.

It is worth highlight the power estimation post place & route has been obtained by setting the inputs switching activity to 0.5.

# Table of contents

# List of figures

# Chapter 1

# State of the art

## 1.1 Introduction to Neural Network

Neural networks (`NNs`) are nowadays widely used for many applications like speech and image recognition. This kind of neural networks has already exceeded human accuracy in many domains at the cost of high complexity from a computational point of view.

Neural networks are based on neurons which are the smallest units in every network. A neuron performs a weighted sum of the inputs and then it applies a non-linear function called `activation function` (Figure 1.1).

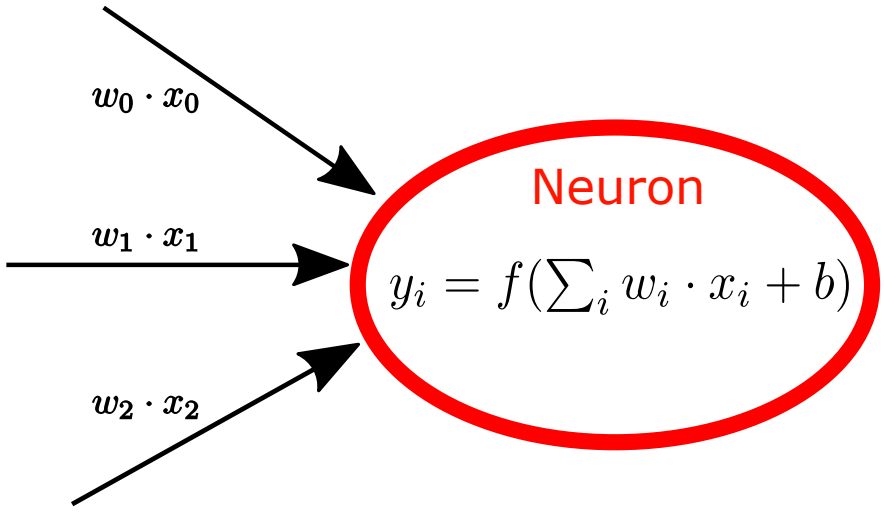


Figure 1.1: Neuron structure.

where $w_i$ are the weights and b is the bias.

There are several functions which can introduce non-linearity into a neural network. The most used ones are:

- `Sigmoid function`: this function is one of the historical non-linear functions and the output value is always in the range $[0 \div 1]$. It can be represented by the following equation:

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

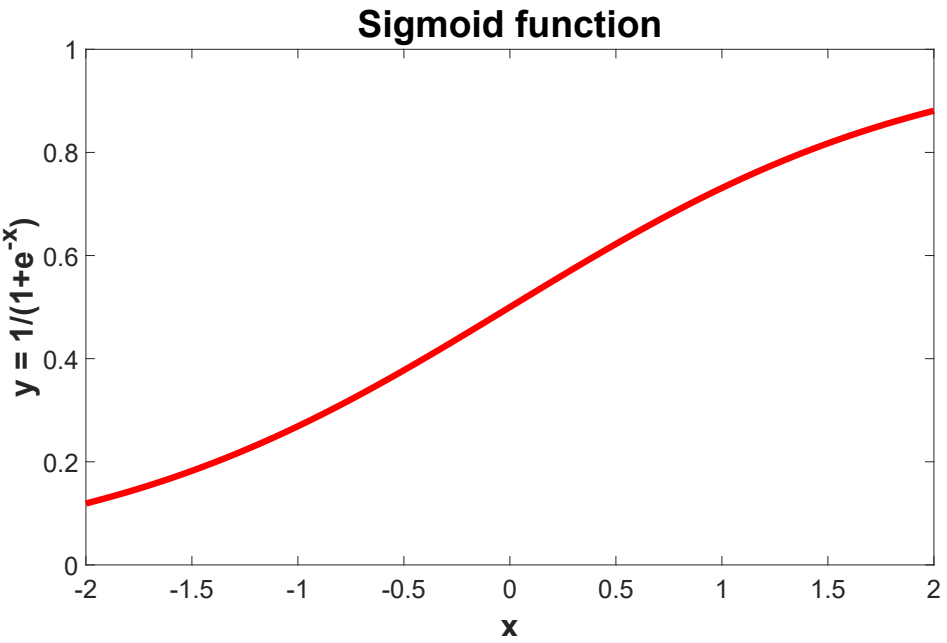


Figure 1.2: Sigmoid function.

- `Hyperbolic tangent`: the hyperbolic tangent function, as the sigmoid one, is one of the historical non-linear function and its output values are limited between $-1$ and $+1$. This function follows the following law:

$$y = tanh(x) = \frac{1 - e^{-2x}}{1 + e^{-2x}}$$

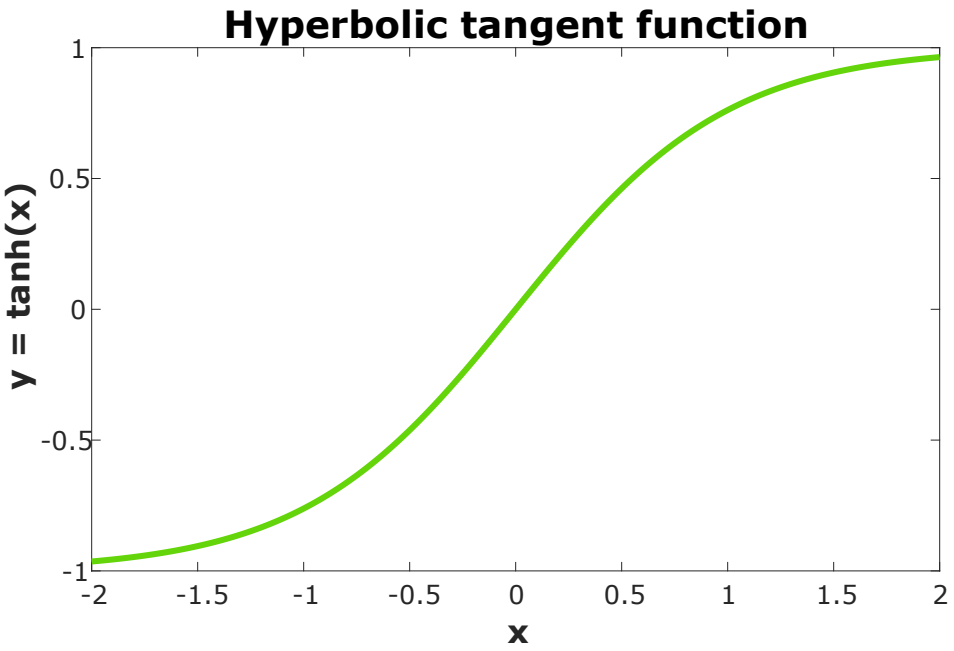


Figure 1.3: Hyperbolic tangent function.

- `Rectified Linear Unit function (ReLU)`: the ReLU function is very simple and allows fast training. For this reason, it has become very popular in recent years. It provides a null value for every negative input and the input itself if it is positive. The equation which describes this behaviour is:

$$y = max(0,x)$$

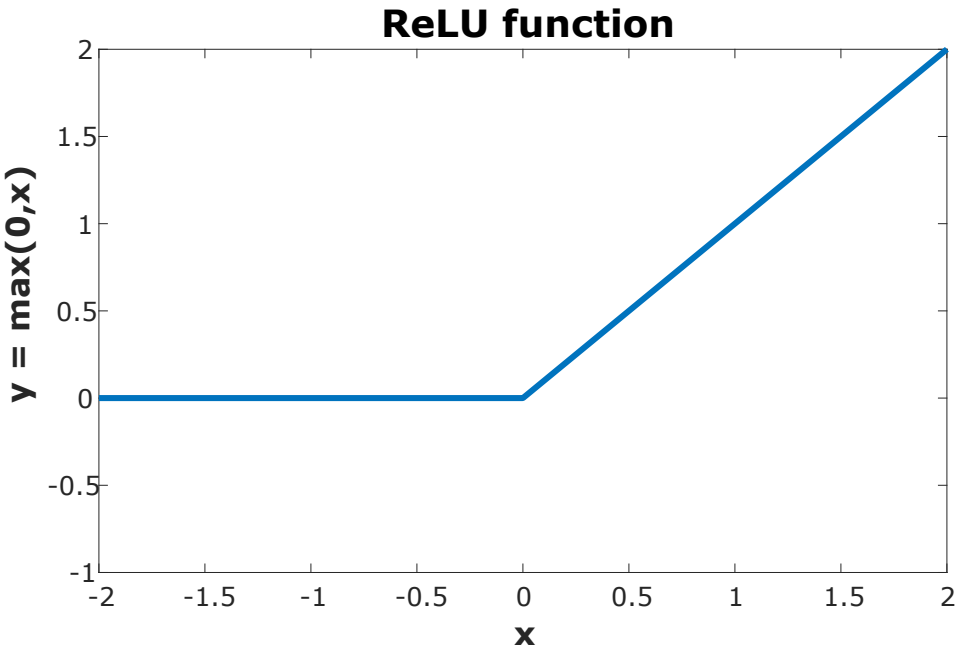


Figure 1.4: ReLU function.

- `maxout`: this function takes the max value of two intersecting linear functions.

The most simple neural network has three layers: an input layer, a hidden layer and an output layer. Each layer is made of by neurons and each one of them is connected to all the successive layer neurons (this is called `Fully Connected layer`).

When a neural network has more than one hidden layer, it is called a deep neural network (`DNNs`). Thanks to multiple hidden layers, DNNs are able to learn with more complexity and abstraction high-level features. For this reason, DNNs achieve superior performance in many tasks.

Every neural network has to be trained before performing the classification otherwise the obtained results would be wrong or not very accurate. Training a network means changing the values of the weights and biases in order to minimize the loss function which is the distance between the wanted result with respect to the obtained one. Once the training is finished the neural network can be used to evaluate different inputs by using the weights and biases determined during the training and this process is called `inference`.

## 1.2 Convolutional Neural Network

A Convolutional Neural Network (`CNN`) is a very common form of DNNs. Normally a DNN can be composed by only Fully Connected (`FC`) layers. However, this kind of choice implies a significant amount of storage and computation because every neuron of a layer is connected to every neuron of the successive layer. Thankfully, in many application, it is possible to use a reduced number of connections without affecting the final results and it is also possible to share the weights which means reducing the storage needed for their memorization.

CNNs use both fewer connections and weights sharing and in this way they are able to reach very high performances.

Generally, a CNN contains multiple *convolutional* layers which allow extracting high-level features of the input. Each convolutional layer generates a `feature map` (fmap) which contains the features extracted from that layer. Every fmap contains essential and unique information about the input data.

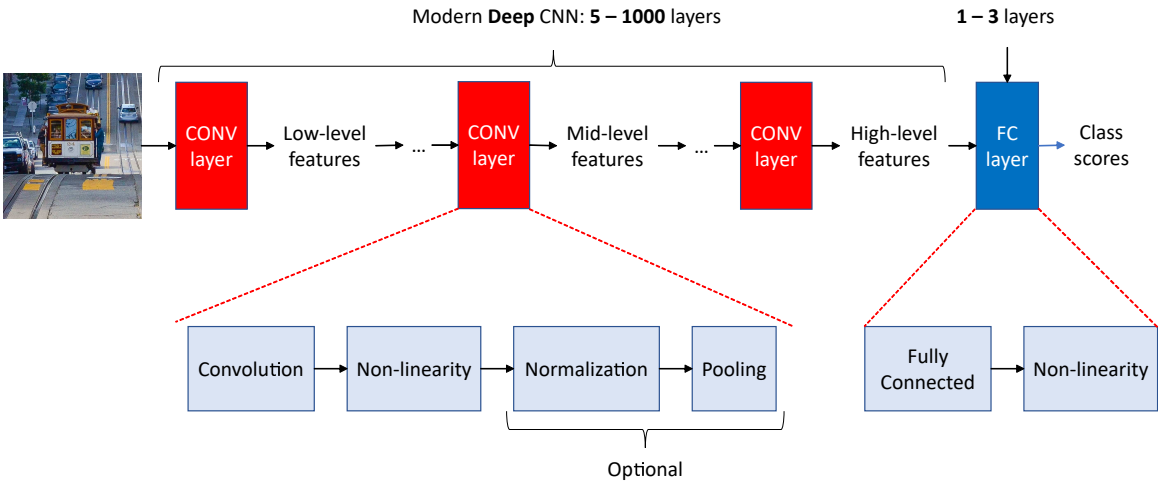


Figure 1.5: Convolutional neural network (example from [1]).

In Figure 1.5, it is possible to see the main layers used for an application like image recognition. In particular each `CONV` layer contains up to 4 sub-layers:

- `Convolution`: each layer which performs a convolution takes as input a set of 2-D *input feature maps (`IFMAPs`)* and every map is called *channel*. For an image there are 3 channels representing the RGB values. Every channel

is convolved with a distinct filter. Filters are organized also as a set of 2-D matrices which contain the weights determined from the training step. The results of each channel convolution are, then, summed together. The final result of a CONV layer is an *output feature map* `OFMap`. If more sets of filters are used it is possible to obtain multiple output channels. It is also possible to process together multiple IFMAPs as a *batch* in order to improve the filters weights reuse. The general equation which allows to compute every element of the OFMAP is the following ([1]):

$$\mathbf{O}[z][u][x][y] = \mathbf{B}[u] + \sum_{k=0}^{C-1}\sum_{i=0}^{S-1}\sum_{j=0}^{R-1}\mathbf{I}[z][k][Ux+i][Uy+j]\cdot\mathbf{W}[u][k][i][j]$$

$$0 \leq z < N, 0 \leq u < M, 0 \leq x < F, 0 \leq y < E$$

$$E = (H - R + U)/U, \quad F = (W - S + U)/U$$

where **O**, **I**, **W** and **B** are the ofmaps, ifmaps, filters and biases matrices respectively. $U$ is a given stride size. The other parameters meaning is specified in the following table:

| Shape parameter | Description |
|---|---|
| $N$ | Batch size of 3-D fmaps |
| $M$ | # of 3-D filters (# of ofmap channels) |
| $C$ | # of filter/ifmap channels |
| $W/H$ | ifmap plane width/height |
| $S/R$ | filter plane width/height (= W or H in FC layers) |
| $F/E$ | ofmap plane width/height (= 1 for FC layers) |

Table 1.1: Shape parameters of a convolutional/fully connected layer ([1]).

- `Non-linearity`: this layer applies a function (activation function) which introduces non-linearity into a neural network and usually this layer is placed after each convolutional or fully connected layer. Different types of activation function are available and they have been already analyzed in the previous section (section 1.1).

- `Normalization`: the goal of this layer is to control the input distribution across

layers. The idea is to normalize the input in order to reach zero mean and a unit standard deviation.

- `Pooling`: computations focused on reducing the dimensionality of a feature map. Since the pooling is applied to each channel separately it allows creating a network more robust and invariant to distortion and small shifts. For example, if max pooling is used with a stride of 2, the following output map is obtained:

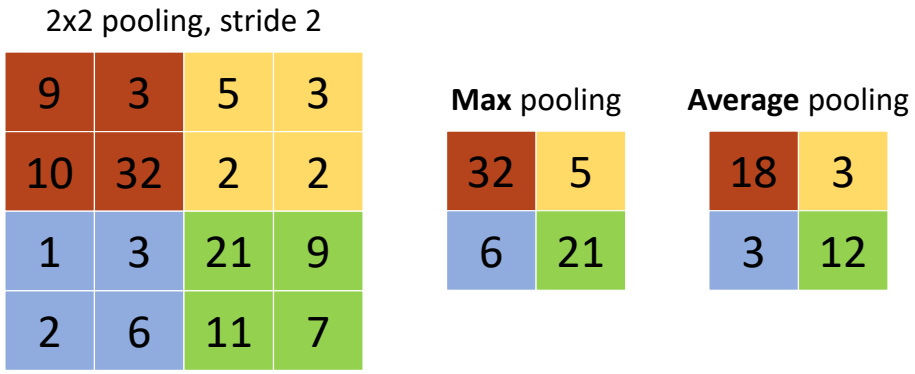


Figure 1.6: Different forms of pooling (figure from [2]).

Usually, a CNN contains from 1 to 3 `FC` layers placed at the end of the network itself. Each FC layer may contain a non-linearity layer other than the standard fully connected layer.

## 1.3 AlexNet neural network

AlexNet is the name of a CNN, designed by Alex Krizhevsky [11] which is able to process a very large dataset like ImageNet. This network is able to recognize 1000 different classes of images. This kind of task requires a very big neural network with a large number of weights.

### 1.3.1 Architecture model

The architecture model can be seen in Figure 1.7:

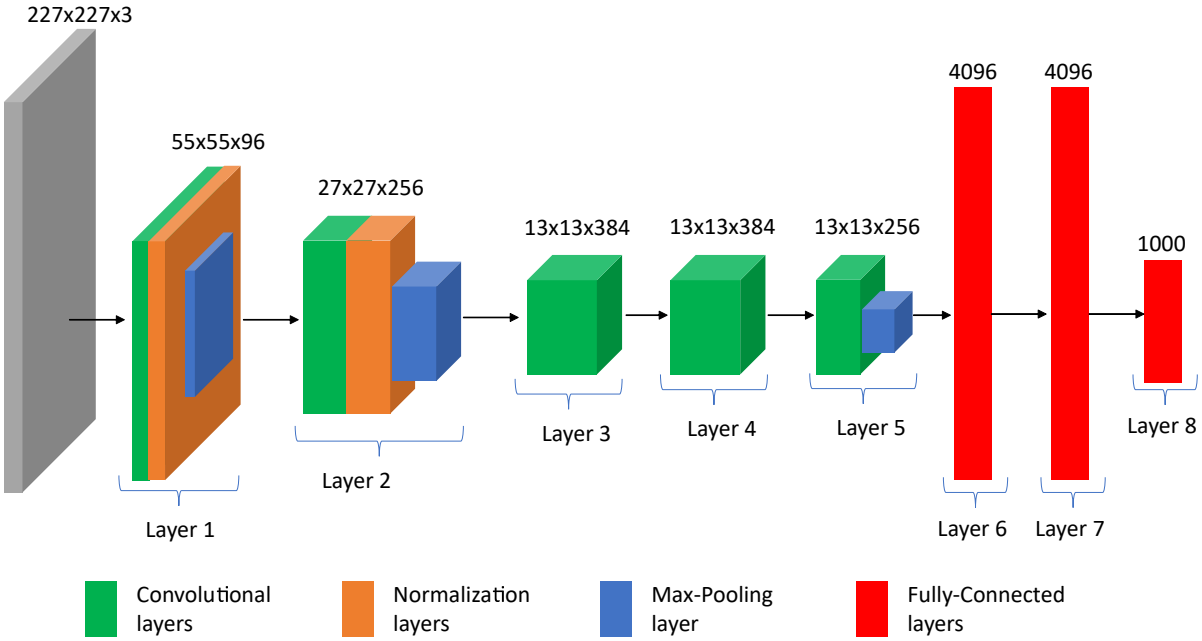


Figure 1.7: AlexNet architecture model (figure from [3]).

The activation function used in the AlexNet algorithm is the `ReLU` function and it is applied after each convolutional layer and each fully connected layer except for the last one which is connected to a 1000-way `softmax`. This last function produces a probability distribution over the 1000 classes labels. After the softmax, there is a max function which selects the output class label with the highest probability for each input image.

The layers organization is the following one:

- the first convolutional layer has as input a 227x227x3 image which is convolved with 96 distinct filters of size 11x11x3 with a stride of 4. The output dimension of a convolutional layer can be computed with the following formula:

$$W_O = \frac{W_I - F + 2 \cdot P}{S + 1}$$

$$H_O = \frac{H_I - F + 2 \cdot P}{S + 1}$$

$$D_O = K$$

where $W_I$, $H_I$, $K$ are respectively input matrix width, height and number of filters (also called kernels). $P$ is the padding while $S$ is the stride. $D_O$ is the output channel number. For the first convolutional layer the padding is equal to $P = 0$. Padding means adding "edges" of zeros (see Figure 1.8) to preserve the output image size avoiding information loss.

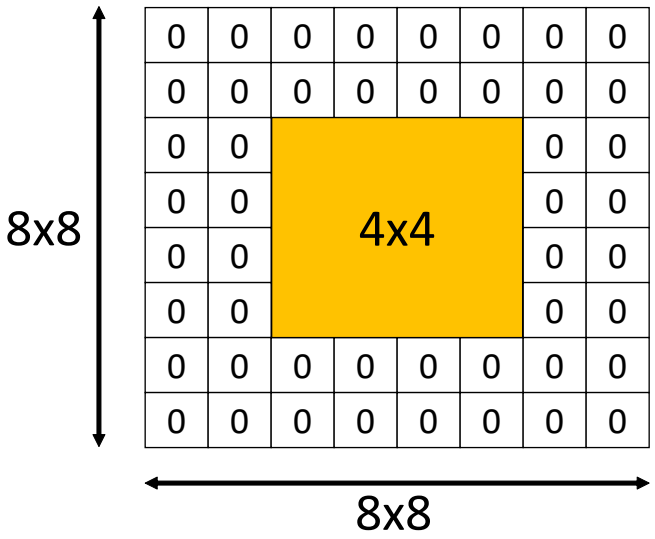


Figure 1.8: Padding example with $P = 2$ (two "edges" of zeros).

- The output of the first convolution is normalized with a `normalization layer` and then an `overlapping Max-Pooling` is applied (normalization and pooling layers are explained later)). The output of the max-pooling layer is a 27x27x96.

- The second convolutional layer takes as input the output of the previous pooling layer and convolves it with 256 distinct filters of size 5x5x96 with a stride of 1. The second convolution layer input is split into two parts (called groups) of size 27x27x48 each and the convolution of the two groups is executed in parallel. The padding for this layer is equal to $P = 2$.

- Also the output of the second convolutional layer is first normalized and then the pooling layer is applied. The output dimension, in this case, is 13x13x256.

- The third convolutional layer convolves the output of the previous layer with 384 distinct filters of size 3x3x256 with a stride of 1. The padding of this layer is $P = 1$.

- The fourth and fifth convolutional layers convolve the output of their previous layers with 384 (for the fourth layer) and 256 (for the fifth one) filters of size 3x3x256 with a stride of 1. Both convolutional layers inputs are split into two groups of size 13x13x192 which are executed in parallel. For these two convolutional layers, the padding is equal to $P = 1$.

- To the output of the fifth convolutional layer is applied a pooling layer and its output dimension is 6x6x256.

- The last three layers are all fully connected. They have 4096, 4096 and 1000 neurons respectively.

### 1.3.2 Local response Normalization layer

This layer normalizes the input distribution in order to reach zero mean and a unit standard deviation. In the AlexNet algorithm a local response normalization is used. The formula below is the one used for this kind of normalization:

$$b^i_{x,y} = \frac{a^i_{x,y}}{(K + \alpha \sum_{j=max(0,i-n/2)}^{min(N-1,i+n/2)} (a^j_{x,y})^2)^\beta}$$

As it is possible to see, it depends on several hyper-parameters like $\alpha$, $\beta$, $n$ and K which are set to the following values:

- $\alpha = 0.0001$;
- $\beta = 0.75$;
- $n = 5$, which is the 1-D window normalization size;
- $K = 1$.

N instead is the total number of kernels in the layer.

### 1.3.3 Max-Pooling layer

In the AlexNet algorithm, an overlapping max pooling is used. The only difference with respect to the one described in the previous section (section 1.2) is the overlapping. If the pooling windows is RxR and the stride `S` is greater or equal to `R`, then the pooling windows do not overlap. However, if the stride `S` is lower than `R` then an overlapping pooling is obtained. From a practical point of view an overlapping pooling looks like the following figure (Figure 1.9):

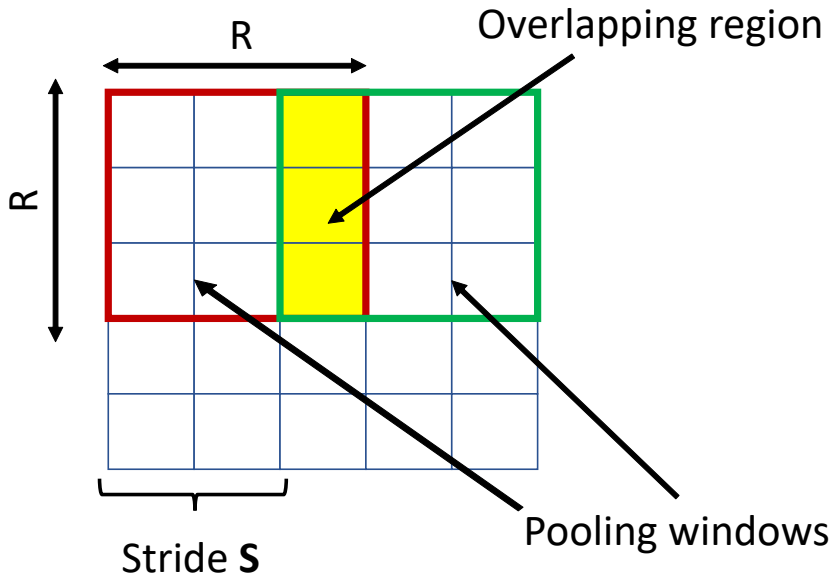


Figure 1.9: Overlapping pooling example.

### 1.3.4 Dropout

The `Dropout` operation allows to turn off some neurons in the hidden layers with a certain probability. In the AlexNet this probability is set to 0.5. This technique is used during the training of a neural network in order to speed up the loss function convergence.

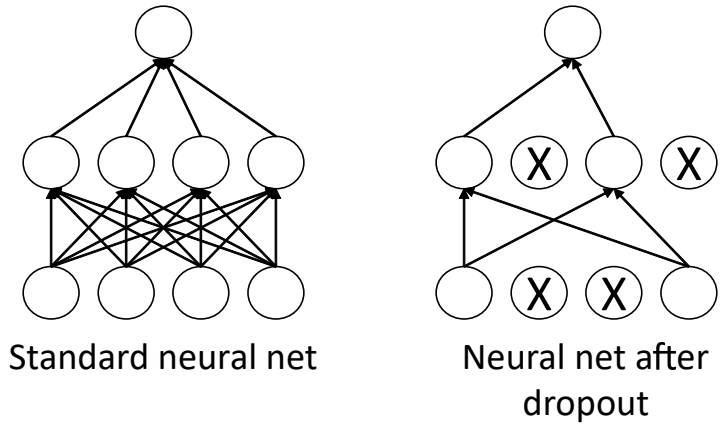


Figure 1.10: Dropout technique example.

### 1.3.5 Softmax

After the last fully connected layer a 1000 way `softmax` function is applied. This function transforms its input in a probability which is useful in some applications like image recognition where the most probable class should be predicted. It should be noted that if the application goal is just to know the identified classes then this layer is not implemented because it is sufficient to see the output of the last FC layer. The formula which implements the softmax function is:

$$y = \frac{e^x}{\sum_{j=1}^{k} e^{a_j}}$$

## 1.4 ZFNet

The ZFNet neural network [12] is an improved version of the AlexNet. This network has significantly improved the image classification error with respect to the AlexNet neural network.

The improvements introduced by the ZFNet are related to the first two convolutional layers: in particular, for the first one the filter size has been reduced from 11x11 to 7x7 and the stride has also been decreased from 4 to 2. For what concerns the second convolutional layer, everything is the same except for the stride which becomes 2 instead of 1 and the padding which is 0 in this case.

This two small modifications allowed the ZFnet to decrease the top-5 error by 1.7%.

The idea behind these small variations is that by reducing the filter size and the stride more information can be extracted by the first layer.

## 1.5 LeNet-5

The LeNet-5 neural network algorithm [13] is one of the most simple algorithms which is able to automatically classify hand-written digits.

Different versions of this network have been developed over time. However, the original model is the one shown in Table 1.2.

| Layer | Feature Map | Size | Kernel Size | Stride | Activation |
|---|---|---|---|---|---|
| **Input image** | 1 | 32x32 | - | - | - |
| **convolution** | 6 | 28x28 | 5x5 | 1 | tanh |
| **Average Pooling** | 6 | 14x14 | 2x2 | 2 | tanh |
| **convolution** | 16 | 10x10 | 5x5 | 1 | tanh |
| **Average Pooling** | 16 | 5x5 | 2x2 | 2 | tanh |
| **convolution** | 120 | 1x1 | 5x5 | 1 | tanh |
| **Fully-Connected** | - | 84 | - | - | tanh |
| **Fully-Connected** | - | 10 | - | - | softmax |

Table 1.2: LeNet-5 layers summarized.

The padding value is always 0 for every layer in the LeNet-5 algorithm.

The version implemented in this thesis is a little bit different with respect to the original one. The first difference is the activation function: the `tanh` function is very demanding from a computational point of view so it has been substituted by the ReLU. Another difference is that the third convolutional layer has been replaced with a fully connected one. For what concerns the pooling layer, the max-pooling layer has been used instead of the average one. In Table 1.3 the summary of the LeNet-5 modified version is reported.

| Layer | Feature Map | Size | Kernel Size | Stride | Activation |
|---|---|---|---|---|---|
| **Input image** | 1 | 32x32 | - | - | - |
| **convolution** | 6 | 28x28 | 5x5 | 1 | ReLU |
| **Max Pooling** | 6 | 14x14 | 2x2 | 2 | ReLU |
| **convolution** | 16 | 10x10 | 5x5 | 1 | ReLU |
| **Max Pooling** | 16 | 5x5 | 2x2 | 2 | ReLU |
| **Fully-Connected** | | 120 | | 1 | ReLU |
| **Fully-Connected** | - | 84 | - | - | ReLU |
| **Fully-Connected** | - | 10 | - | - | softmax |

Table 1.3: Summary of the LeNet-5 modified version.

This modified version will be used to test the correct behaviour of the hardware accelerator chapter 6.

## 1.6 GoogLeNet

The GoogLeNet [4] is a neural network used to recognized images like AlexNet, ZFNet and VGGNet. However, this network has significant improvements over ZFNet and AlexNet and at the same time, it has a relatively lower error rate compared with the VGGNet. In Figure 1.15 the GoogLeNet neural network model is shown.

The GoogLeNet algorithm introduces some new important concepts which are worth to be mentioned:

- The 1x1 convolution;
- Inception Module;
- Global Average Pooling.

### 1.6.1 The 1x1 convolution

The 1x1 convolution is used to reduce the computation needed by bigger convolutional layers. As a matter of fact, if the computation is reduced, the depth and the width of the network can be increased. To explain better why the computation is reduced, it might be useful to analyze an example. In Figure 1.11 a 5x5 filter is applied to an input feature map.

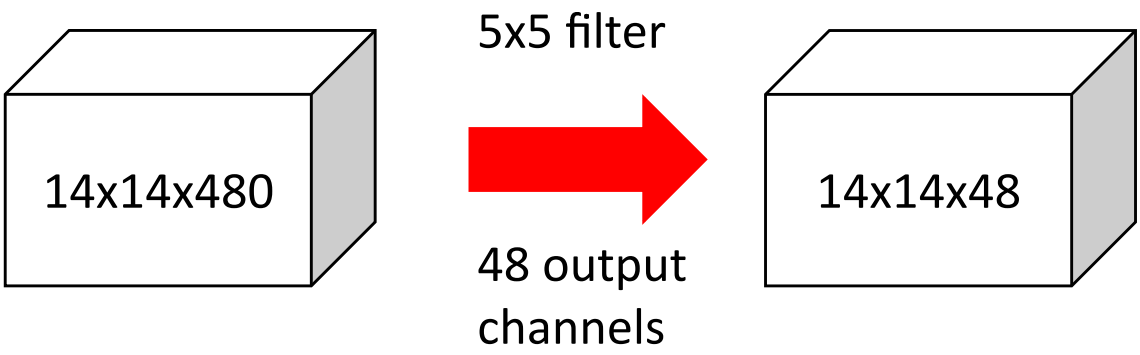


Figure 1.11: Convolutional layer without 1x1 filter.

The number of operations are

$$\#\ operations = (14 \cdot 14 \cdot 48) \cdot (5 \cdot 5 \cdot 480) = 112.9 \times 10^6$$

Now if a 1x1 filter is applied before the 5x5 one, like in Figure 1.12

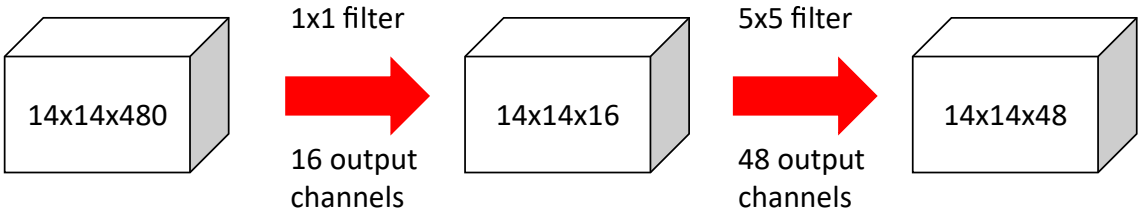


Figure 1.12: Convolutional layer with 1x1 filter.

the number of operations becomes

$$\#\ operations = (14 \cdot 14 \cdot 16) \cdot (1 \cdot 1 \cdot 480) + (14 \cdot 14 \cdot 48) \cdot (5 \cdot 5 \cdot 16) = 5.3 \times 10^6$$

which means that even if an additional layer has been inserted the total executed operations are more than 20 times less.

### 1.6.2 Inception module

An inception module is a computational block where filters with multiple sizes are applied to the same input and then their results are concatenated. In Figure 1.13 it is possible to see a naive version of the inception module.

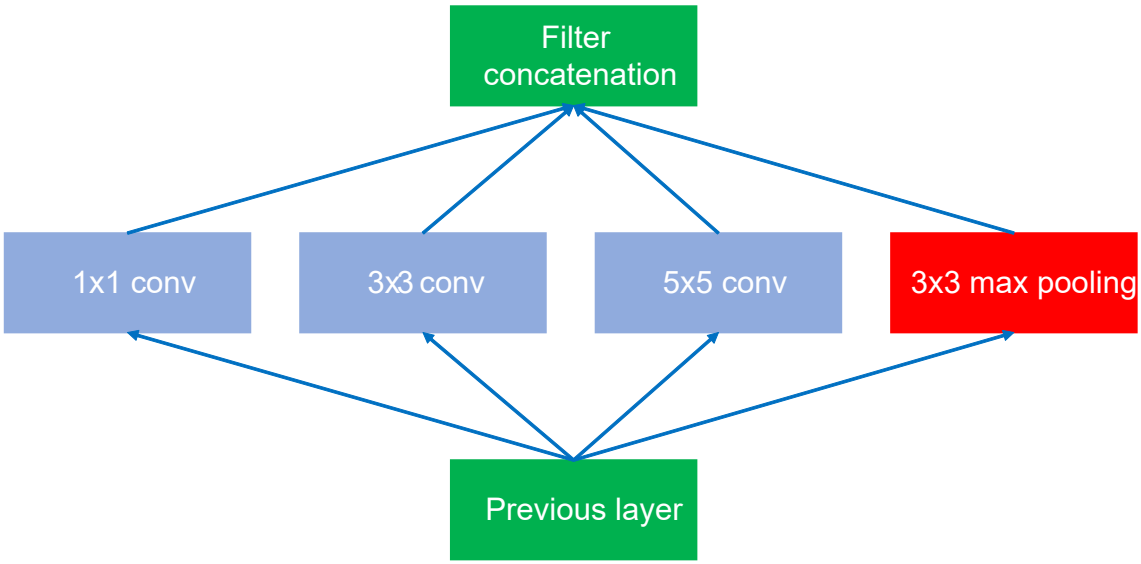


Figure 1.13: Inception module without dimension reductions.

By inserting 1x1 convolution the number of operations to be done are reduced as seen in subsection 1.6.1.

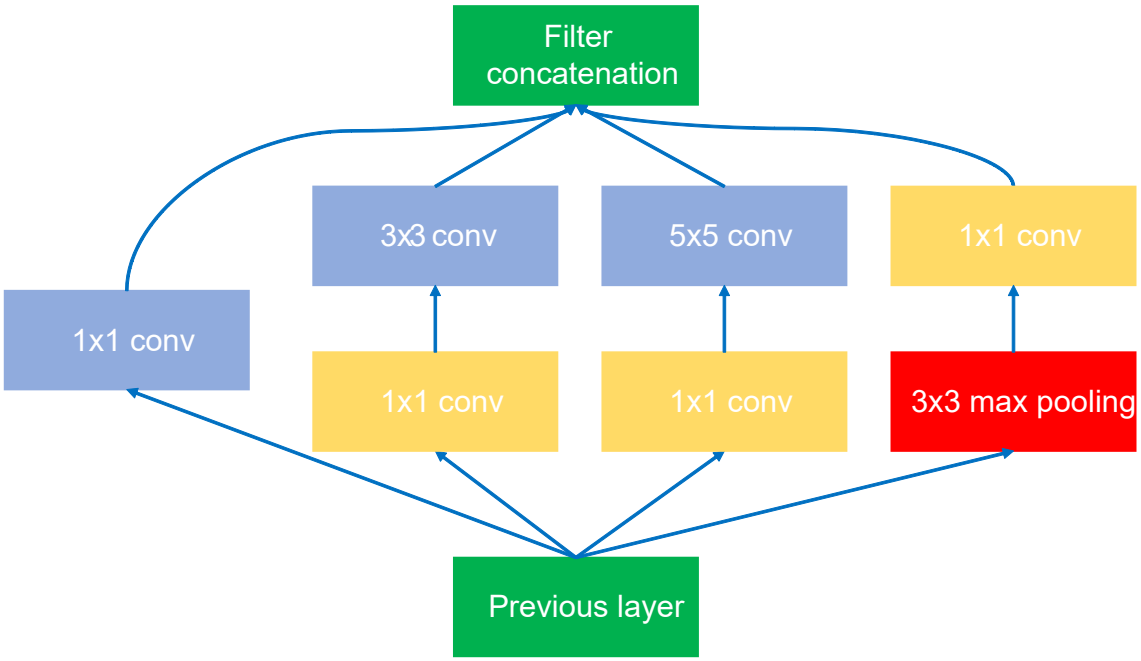


Figure 1.14: Inception module with dimension reductions.

### 1.6.3 Global Average Pooling

In classical neural network implementation like AlexNet, fully-connected layers are used at the end of the network. In GoogLeNet, global average pooling is used near the end of the network before inserting an actual fully-connected layer. The idea is that by using the average pooling there is not any weight to store which is a big advantage for such enormous networks. Moreover moving from an FC layer to an average pooling one allows improving the top-1 accuracy by about 0.6%. This means that no precision is lost when the fully-connected are substituted by an average pooling. Obviously, at the end of the network, at least one FC layer has still to be inserted followed by a softmax function to obtain the classification results.

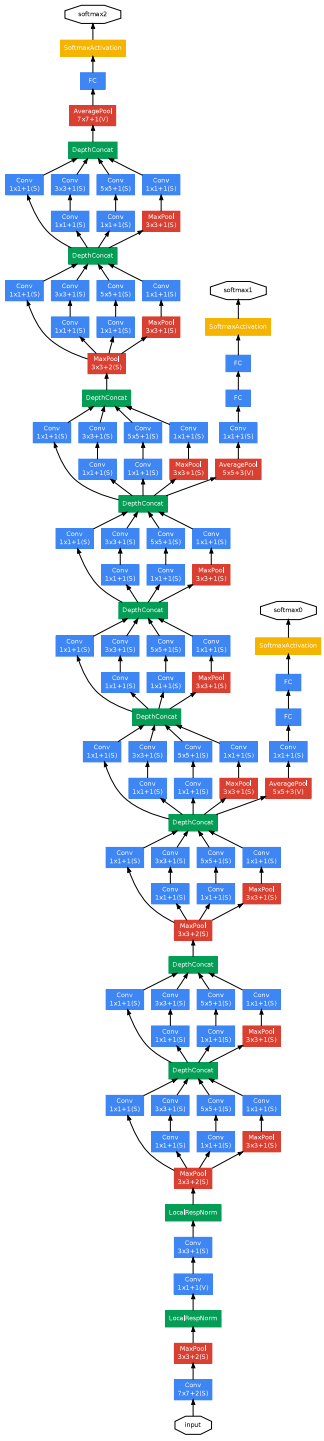


Figure 1.15: GoogLeNet neural network model [4].

## 1.7 VGGNet

The VGGNet [14] neural network has improved networks like ZFNet, AlexNet and GoogLeNet in classification tasks.

This network is based on a 3x3 filters in every convolutional layer. The idea is that by using 2 layers of 3x3 filters, a 5x5 area can be covered. If instead 3 layers of 3x3 filters are used, a 7x7 effective area is covered. This means that large-size filters such as the ones used in AlexNet, ZFNet and so on, are not needed.

Furthermore, by using smaller filters, the number of parameters to be stored are fewer. On top that having fewer parameters to be learnt during training allows a faster convergence and reduces the overfitting problem.

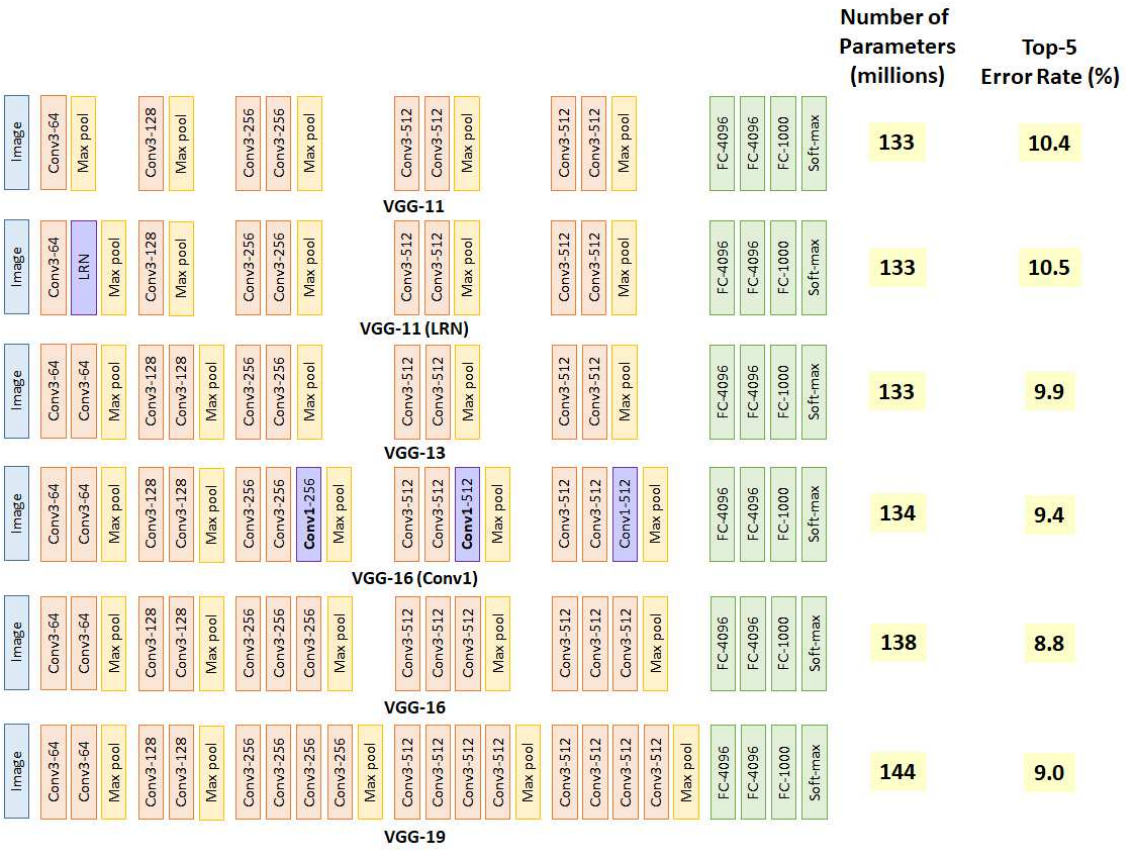


Figure 1.16: Different VGG layer structure using single scale (256) evaluation [5].

## 1.8 Residual Neural Network

In traditional neural networks, each layer feeds into the next layer. In residual neural network (ResNet) [6], on the other hand, each layer feeds into the next one and directly into the ones about 2-3 hops away. As a matter of fact, the basic block in such a network is a residual block like in Figure 1.17.

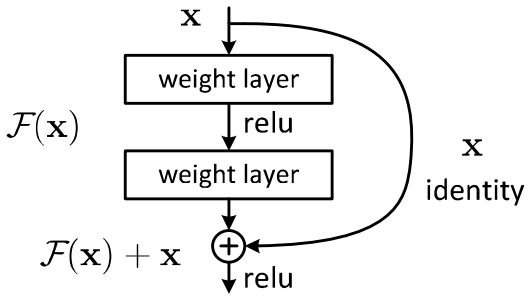


Figure 1.17: Residual block [6].

It is known that the accuracy of neural networks increases with an increasing number of layers. However, there is a limit to the layers which can be added and that result in an accuracy improvement. It might happen that very deep neural networks are able to learn very complex functions but due to vanishing gradients and high dimensionality problems, they are not able to learn simple functions like the identity one.

Furthermore, if the number of layers continues to be increased, the accuracy will start to saturate at some point and eventually it will start to degrade. This problem is called `degradation` and due to it, shallower networks might learn better than their deeper counterparts.

For this reason, a block like the one in Figure 1.17 is used. This residual connection connects the input to a certain layer output by an element-wise addition.

Essentially, residual blocks allow the flow of information from initial layers to last ones.

In Figure 1.18 there are some examples of `ResNet`.

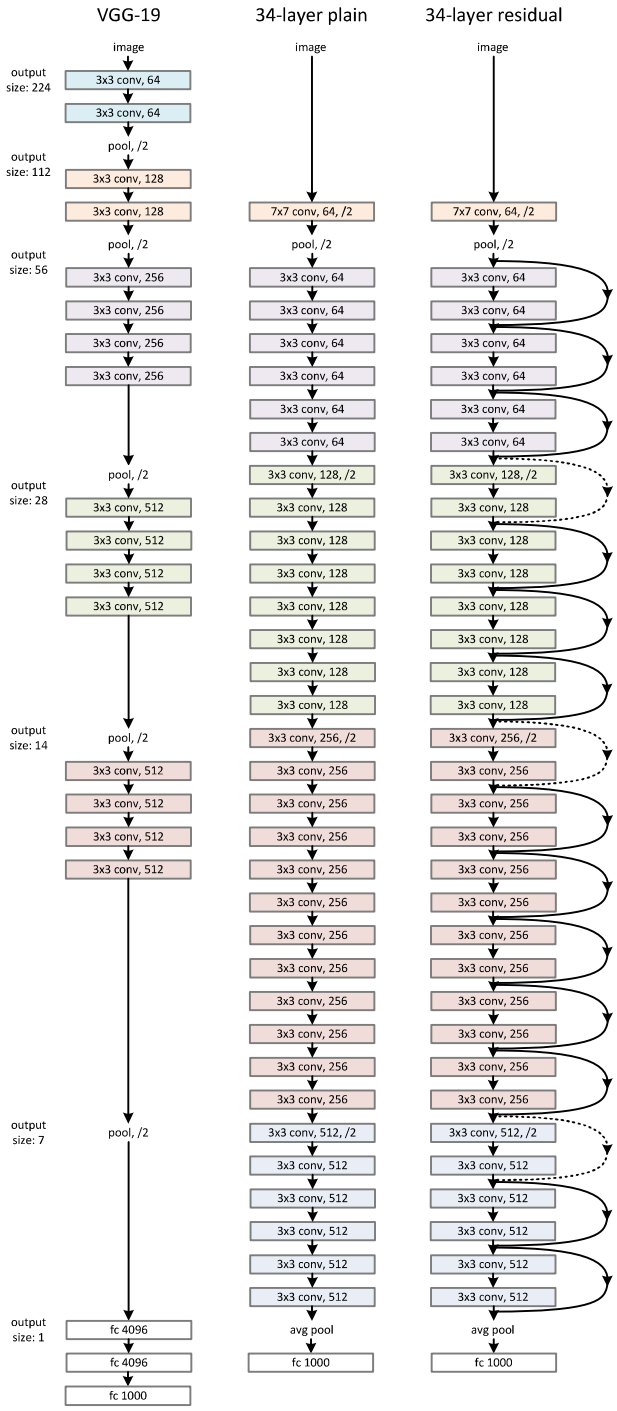


Figure 1.18: Figure 3. Example network architectures for ImageNet. Left: the VGG-19 model as a reference. Middle: a plain network with 34 parameter layers. Right: a residual network with 34 parameter layers. The dotted shortcuts increase dimensions [6].

## 1.9 Other Neural Networks

Nowadays there are a lot of different neural networks which in one way or the other use the layers and the principles seen before.

There are other two neural networks which are worth to be mentioned: `DenseNet` and `Recurrent neural network (RNN)`.

### 1.9.1 DenseNet

In standard Convolutional neural network, input images go through convolution to obtain high-level features. In ResNet, instead, the identity block is used to propagate the input from the initial layers to last ones with an element-wise addition.

In DenseNet neural network [7], on the other hand, each layer is connected to all the next ones through concatenation. It is like each layer receives a collective knowledge from all the preceding layer [15].

In Figure 1.19 a complete dense neural network with three dense blocks is depicted.

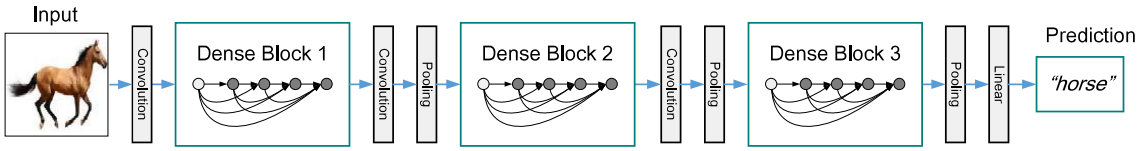


Figure 1.19: A deep DenseNet with three dense blocks [7].

In Figure 1.20 an example of a 5-layer dense block is presented. Each layer receives all preceding feature maps and thanks to this the network can be thinner and compact. As a matter of fact, the number of channels is usually fewer in this kind of network.

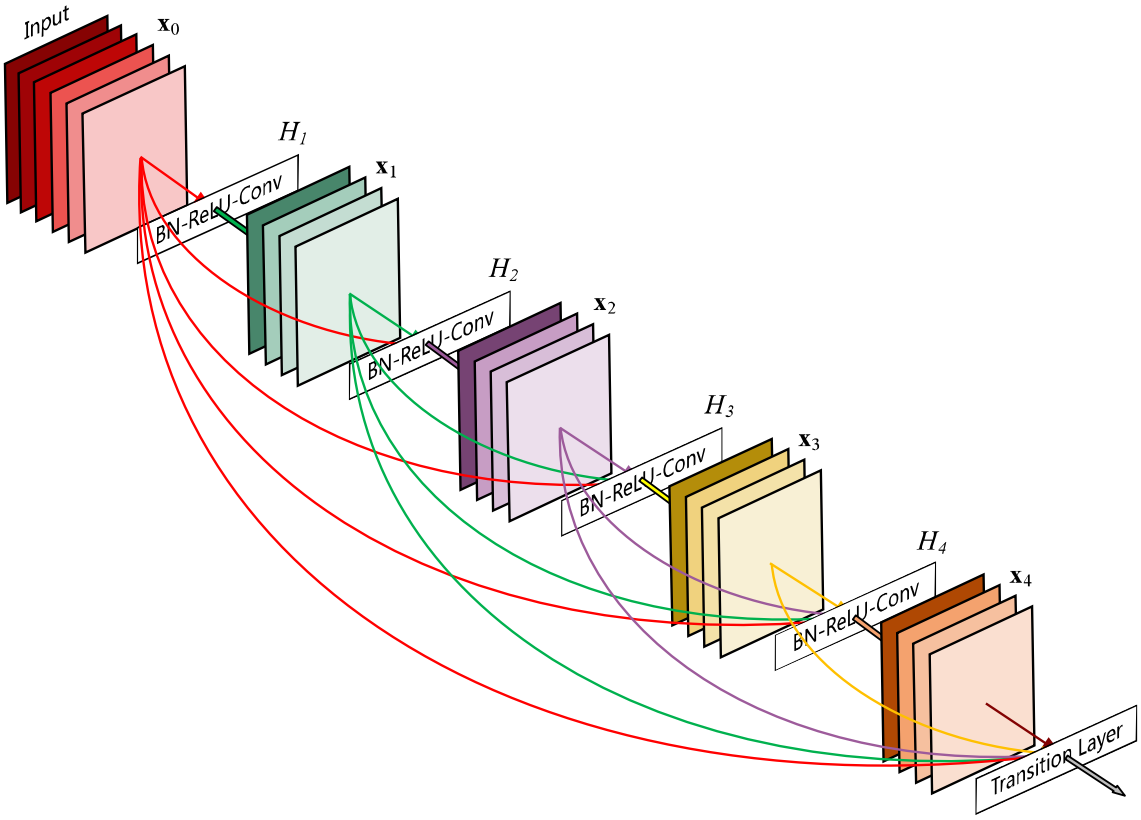


Figure 1.20: A 5-layer dense block. Each layer takes all preceding feature-maps as input [7].

### 1.9.2 Recurrent Neural Networks

Recurrent neural networks (`RNN`) [8] are used when it is necessary to take a series of input with no predetermined limit on size.

The difference with respect to a regular neural network called repeatedly is that RNNs are able to remember the past and then take decisions based on what they have learnt.

The idea is that the outputs are influenced by weights applied on inputs (just like regular neural networks) and by a hidden state vector which represents the context based on input/outputs history.

Due to this, the same input can produce a different output depending on the previous inputs in the series.

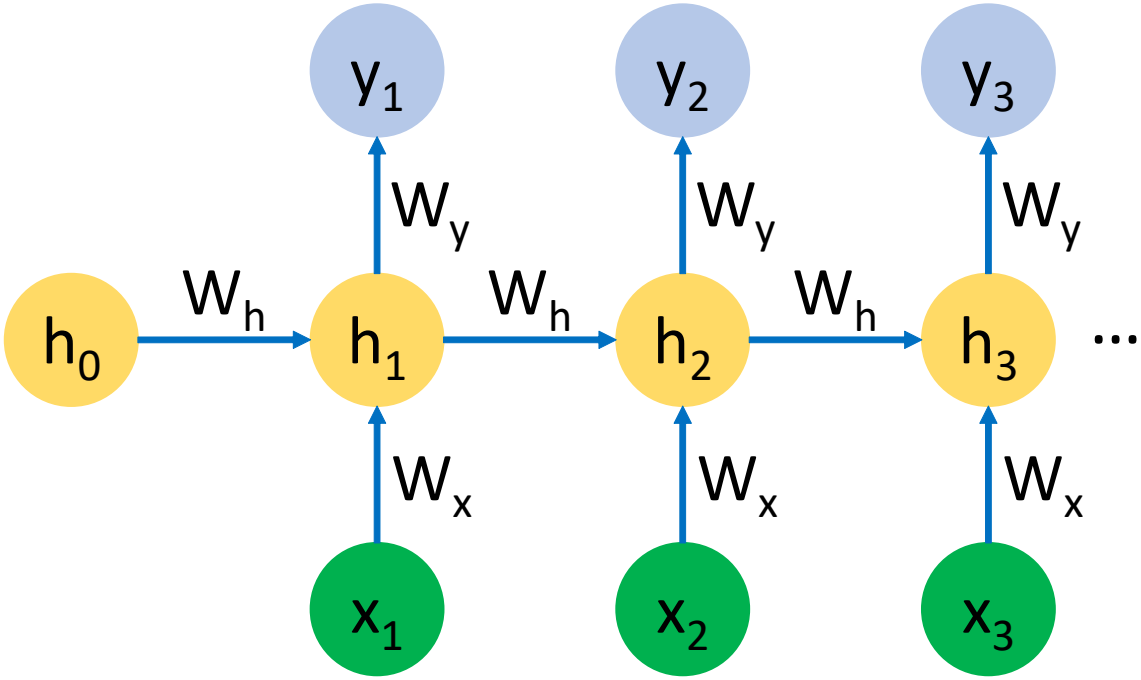


Figure 1.21: A RNN with a hidden state [8].

In Figure 1.21, an example of a Recurrent neural network is depicted. Hidden states are used to carry information from one input in the series to others.

## 1.10 Logic-in-Memory

Modern computing systems are based on von Neumann paradigm which consists of exchanging data between a CPU and a memory.

A CPU executes instructions on data, read from the memory, and then it writes back the results in the same memory. It is clear that this kind of mechanism is not efficient when the CPU has to deal with large quantities of data because a lot of reading/writing operations from/to the memory are executed.

Today, CPUs are very powerful and over the year their performances have increased thanks to a higher number of cores and a higher clock frequency. On the other hand, memories are not able to scale and improve at the same speed as the CPUs. For this reason, the memory bandwidth is becoming the main limitation of system performances.

Another important drawback of this type of paradigm is power consumption. Every access in memory consumes power which means that for data-intensive applications the amount of energy spent for memory accesses has a big influence on the overall power consumption.

A possible alternative to this paradigm is the Logic-in-Memory (`LiM`) approach [16]. Integrating simple logic directly inside the memory cell could cross the memory wall problem. Furthermore, since data are computed inside the memory there is no need to move them between the storage unit and the computing one.

This solution seems promising in theory but it is important to consider the operations to be executed in memory. First of all only basic operation can be implemented inside the memory and a different technology with respect to CMOS might need to be used for the memory implementation to obtain actual performance improvement.

Another important aspect is that not all algorithms can benefit from a LiM approach. If data are moved in different positions very often during the algorithm, it might be necessary to use several interconnections and this is not an ideal solution.

# Chapter 2

# AlexNet software implementation

## 2.1 TensorFlow

TensorFlow is one of the most successful open-source platforms for neural network development. A neural network in TensorFlow is expressed as a computation graph with nodes and edges: a node represents an operation while an edge represents the data consumed or produced by one or more nodes.

TensorFlow uses tensors to represent nodes. A tensor is an arbitrary dimensional array (which can represent a high order matrix or a vector) and each node can take more inputs and produce several outputs. There are two types of tensors: placeholders and variables. Variables are the parameters of the algorithm and TensorFlow keeps track of how to change these to optimize the algorithm. Placeholder, on the other hand, are objects that allow you to feed in data of a specific type and shape and depend on the results of the computational graph, such as the expected outcome of a computation.

In a Neural Network weights are represented by variables which TensorFlow automatically changes to minimize the loss function (when training is performed) while input data are represented by placeholders.

When training is performed, training data are usually first extracted and then fed to a model running on an accelerator. However if not optimized, while the CPU is preparing the data, the accelerator is sitting idle doing nothing and vice versa when the accelerator is training the model, the CPU is sitting idle. Since the training step time is the sum of both CPU pre-processing time and the accelerator training time, it is clear that one has to optimize in some way both of them to increase the performances of the algorithm.

One method to reduce this time is to overlap the preprocessing and the model execution of a training step: while the accelerator is performing training step `N`, the

CPU is preparing the data for step `N+1`. Thanks to this optimization it is possible to reduce the step time to the maximum (instead of the sum) between the training step time and the CPU pre-processing time. This method can be used also for inference and it is called `input pipeline`.

Furthermore, TensorFlow is capable to work in a heterogeneous environment with multicore CPU, GPU and Tensor Processing Unit (TPU) from mobile devices to production data centres.

## 2.2 Python code

The AlexNet Neural Network has been implemented using Python and the TensorFlow framework. The code has been divided into the following sections:

- AlexNet class;
- Dataset class;
- Data generator class;
- high-level file;
- Caffe classes.

### 2.2.1 AlexNet class

In this class the model of the AlexNet has been described: all the layers have been implemented using TensorFlow variables for the weights. This model is based on a Git-Hub open source project implemented by `Frederik Kratzert` [17].

At the beginning of the code the class `AlexNet` is declared and then the `init` function is defined. This function is executed every time a class is called because it is the constructor of a class. As it is possible to see from the code below, in this constructor the main features of the class are defined and the arguments of the class are stored into class variables.

```
import tensorflow as tf
import numpy as np

```

```
4
class AlexNet(object):
6 """Implementation of the AlexNet."""

8 def __init__(self, x, keep_prob, num_classes, skip_layer,
  weights_path='DEFAULT'):
10 """Create the graph of the AlexNet model.

12 Args:
  x: Placeholder for the input tensor.
14 keep_prob: Dropout probability.
  num_classes: Number of classes in the dataset.
16 skip_layer: List of names of the layer, that get trained from
  scratch
18 weights_path: Complete path to the pre-trained weight file, if it
  isn't in the same folder as this code
20 """
  # Parse input arguments into class variables
22 self.X = x
  self.NUM_CLASSES = num_classes
24 self.KEEP_PROB = keep_prob
  self.SKIP_LAYER = skip_layer
26
  if weights_path == 'DEFAULT':
28 self.WEIGHTS_PATH = 'bvlc_alexnet.npy'
  else:
30 self.WEIGHTS_PATH = weights_path

32 # Call the create function to build the computational graph of AlexNet
  self.create()
```

At the end of the constructor, another function is called: the `create()` function defines all the layers of the AlexNet.

```
  def create(self):
2 """Create the network graph."""
  # 1st Layer: Conv (w ReLu) -> Lrn -> Pool
4 conv1 = conv(self.X, 11, 11, 96, 4, 4, padding='VALID', name='conv1')
  norm1 = lrn(conv1, 2.5, 1e-04, 0.75, name='norm1')
6 pool1 = max_pool(norm1, 3, 3, 2, 2, padding='VALID', name='pool1')

8 # 2nd Layer: Conv (w ReLu)  -> Lrn -> Pool with 2 groups
  conv2 = conv(pool1, 5, 5, 256, 1, 1, groups=2, name='conv2')
10 norm2 = lrn(conv2, 2.5, 1e-04, 0.75, name='norm2')
  pool2 = max_pool(norm2, 3, 3, 2, 2, padding='VALID', name='pool2')
12
  # 3rd Layer: Conv (w ReLu)
14 conv3 = conv(pool2, 3, 3, 384, 1, 1, name='conv3')

16 # 4th Layer: Conv (w ReLu) splitted into two groups
```

```
conv4 = conv(conv3, 3, 3, 384, 1, 1, groups=2, name='conv4')

# 5th Layer: Conv (w ReLu) -> Pool splitted into two groups
conv5 = conv(conv4, 3, 3, 256, 1, 1, groups=2, name='conv5')
pool5 = max_pool(conv5, 3, 3, 2, 2, padding='VALID', name='pool5')

# 6th Layer: Flatten -> FC (w ReLu) -> Dropout
flattened = tf.reshape(pool5, [-1, 6*6*256])
fc6 = fc(flattened, 6*6*256, 4096, name='fc6')
dropout6 = dropout(fc6, self.KEEP_PROB)

# 7th Layer: FC (w ReLu) -> Dropout
fc7 = fc(dropout6, 4096, 4096, name='fc7')
dropout7 = dropout(fc7, self.KEEP_PROB)

# 8th Layer: FC and return unscaled activations
self.fc8 = fc(dropout7, 4096, self.NUM_CLASSES, relu=False, name='fc8')
```

Every layer has been defined later in the code as a function with appropriate arguments. For example in the following piece of code it is possible to see the `convolutional layer` function which takes as arguments the layer input, the filter width, the number of filters, the stride in both y and x-direction, a name, the padding type and finally the number of groups in which the convolutional layer should be divided.

```
def conv(x, filter_height, filter_width, num_filters, stride_y, stride_x, name,
padding='SAME', groups=1):
"""Create a convolution layer.
```

Moreover, each layer has its own weights which usually are defined after a Neural Network training. However, it is possible to find files which contain pre-trained weights. In this way, the inference can be performed without any training.

It is clear that these weights have to be loaded in the graph defined by TensorFlow. The function which allows doing this operation is the following one:

```
def load_initial_weights(self, session):
"""Load weights from file into network.

As the weights from http://www.cs.toronto.edu/~guerzhoy/tf_alexnet/
come as a dict of lists (e.g. weights['conv1'] is a list) and not as
dict of dicts (e.g. weights['conv1'] is a dict with keys 'weights' &
'biases') we need a special load function
"""
# Load the weights into memory
weights_dict = np.load(self.WEIGHTS_PATH, encoding='bytes',
      allow_pickle=True).item()
```

```
# Loop over all layer names stored in the weights dict
for op_name in weights_dict:

# Check if layer should be trained from scratch
if op_name not in self.SKIP_LAYER:

with tf.variable_scope(op_name, reuse=True):

# Assign weights/biases to their corresponding tf variable
for data in weights_dict[op_name]:

# Biases
if len(data.shape) == 1:
var = tf.get_variable('biases', trainable=False) #load weights only of non
      trainable layers
session.run(var.assign(data))

# Weights
else:
var = tf.get_variable('weights', trainable=False) #load weights only of non
      trainable layers
session.run(var.assign(data))
```

It is worth noting that in this class it is possible to define layers to be trained in the `SKIP_LAYER` variable: for these layers pre-trained weights will not be used which means that they have to be trained from scratch. However, the possibility to train just a certain number of layers is very interesting because if one has a particular set of images to be recognized, by training only the last Fully Connected layers, for example, it is possible to increase the accuracy for those specific images.

### 2.2.2 Caffe class

This class contains only the images categories names which can be recognized with the pre-trained weights loaded in the TensorFlow graph. These are just a few lines of this class in order to see how categories are defined:

```
class_names = '''tench, Tinca tinca
goldfish, Carassius auratus
great white shark, white shark, man-eater, man-eating shark, Carcharodon carcharias
tiger shark, Galeocerdo cuvieri
hammerhead, hammerhead shark
electric ray, crampfish, numbfish, torpedo
```

### 2.2.3 Data generator class

In this class, image data are prepared in order to use the `input pipeline` method to increase performances and reduce the amount of time to perform the inference.

```
class dataGenerator(object):
def __init__(self, val_path='val'):
self.all_images = self.generate_list(val_path)

def generate_list(self, val_path):
all_images = {}  #make a dictionary
for img_folder in os.listdir(val_path):
img = glob.glob(val_path + os.path.sep + img_folder + os.path.sep + '*.jpg')
#store images path for each category
all_images[img_folder] = img
return all_images

def get_next_image(self):
while True:
#choose one random category
img_class1 = random.choice(list(self.all_images))
#choose one random image from that category
img1 = random.choice(self.all_images[img_class1])
yield (img_class1, img1)
```

It is possible to see how the constructor method calls the `generate list` function which basically searches in the given path all the jpg images. A dictionary is created where at each key all the images which corresponds to that key are stored. For example if the path is something like:

*/category_name/image_name.jpg*

the key is the `category_name` while the images for each key are those in the folder itself. Moreover, every time a new image is requested through the function `get_next_image`, a random image of a randomly chosen category is picked.

### 2.2.4 Dataset class

Once the images are prepared, the tf.data API has to be used to build flexible and efficient input pipelines. The code from [18] which implements tf.data API has been modified and used in order to work with the AlexNet model. A detailed explanation on how to use this API can be found here [18] and here [19], while in the following

only the main features are highlighted and explained.

```
class Dataset(object):

def __init__(self, batch_size, generator=dataGenerator()):
self.next_element = self.build_iterator(batch_size, generator)

def build_iterator(self, batch_size, pair_gen: dataGenerator):
#batch_size = batch_size
prefetch_batch_buffer = 5

dataset = tf.data.Dataset.from_generator(pair_gen.get_next_image, (tf.string,
      tf.string))
dataset = dataset.map(self._read_image_and_resize)
dataset = dataset.batch(batch_size)
dataset = dataset.prefetch(prefetch_batch_buffer)
iter = dataset.make_one_shot_iterator()
element = iter.get_next()

return element

def _read_image_and_resize(self, label, img):
# read images from disk
#print("img: ", img, "+ label ", label, "\n")
img1_file = tf.read_file(img)
img1 = tf.image.decode_jpeg(img1_file)
#Set shape
img1.set_shape([None, None, 3])


#make transformation on the image
img_resized = tf.image.resize_images(img1, [227, 227])
img_centered = tf.subtract(img_resized, IMAGENET_MEAN)
#RGB -> BGR
img_bgr = img_centered[:,:,::-1]

return img_bgr, label
```

The main idea is to create a dataset of images and categories up to the `batch_size` dimension using the previously defined `Data generator class`. After that, an iterator is created in order to remember what element has been used and most important what is the next image to be evaluated.

Every time `next_element` is called, a new dataset of images of `batch_size` dimension is created thanks to the function `build_iterator`.

It has to be highlighted how each image is resized and converted in `BGR` format (instead of `RGB`) before being evaluated.

### 2.2.5 High level file

In the high-level file, all the classes are joined together and the AlexNet network is actually run and the results are printed on a terminal. First of all, some parameters have to be defined:

- num_classes: the classes number is fixed to 1000 and this value depends on the actual number of categories on which the network has been trained and on the dimension of the last fully connected layer (which means how many neurons the last layer has);
- batch_size: the batch size is the number of images to be evaluated in the same run of the algorithm. It can be seen as the fourth dimension of the input matrix (because there are already width, height and channel);
- train_layers: train layers are the ones to be trained. Usually only the last fully connected layers are selected. In this case, this array is empty because no training is performed;
- generations: number of times the algorithm is executed. This option is used only when training is performed.

```
batch_size = 100
num_classes = 1000
generations = 10000   #total steps
train_layers = []

generator = dataGenerator()
ds = Dataset(batch_size, generator)
next_batch = ds.next_element #next iterator

keep_prob = 1.0

# Initialize model
model = AlexNet(next_batch[0], keep_prob, num_classes, train_layers)

# Link variable to model output
score = model.fc8
prob = tf.nn.softmax(score)
#Test output

init = tf.global_variables_initializer()
#sess = tf.Session(config=tf.ConfigProto(log_device_placement=True))
```

```
sess = tf.Session()
sess.run(init)
# Load the pretrained weights into the non-trainable layer
model.load_initial_weights(sess)
img_input = model.X
```

In the first part of the code the needed classes are called and the model is initialized. After that the normalized version of the `softmax` function is applied at the output of the last fully connected layer.

It is important highlighting that in this phase the TensorFlow graph has been created but only when the session is instantiated and run the evaluation truly starts. For example to initialize all the variables of the graph the following code has been written:

```
init = tf.global_variables_initializer()
```

However, the actual initialization is not executed until the session is run. This means that the following lines have to be added:

```
sess = tf.Session()
sess.run(init)
```

These lines create a Session and then the `init` tensor is evaluated. TensorFlow is different with respect to the other programming languages: tensors are evaluated only when the run of the session is performed and only the tensors which are the arguments of the run function are actually evaluated. However, it is common that some tensors are the results of several operations on previous tensors. In those cases, TensorFlow is able to track backwards all the tensors that need to be evaluated and to compute the right results.

TensorFlow while running the algorithm performs its routines to optimize the code based on the CPU or GPU available. In order to allow TensorFlow to properly execute these routines (before actually evaluating the performance of the algorithm) the following code has been added:

```
for i in range(100):   ###used to allow TensorFlow to run its routines to optimize
      the CPU
output= sess.run(prob)
```

which means execute the algorithm 100 times before going on with the remaining code.

The final part of the high level file code is the following:

```
t = time.time()   #Time from epoch (1 january 1970 in seconds)
(output, output_prova,[img1,label]) = sess.run([prob,img_input, next_batch])
#output_prova = sess.run(img_input)
t_f = time.time() #Final time
for step in range(batch_size):

index = argmax(output,1)[step]
print(f"step {step}, label:{label[step]}, label_pred_argmax: {class_names[index]}
      ")
#plt.subplot(1, 2, 1)
#plt.imshow(img1[step].astype(np.uint8))
#plt.title(f'{label[step]}')
#plt.subplot(1, 2, 2)
#plt.imshow(output_prova[step].astype(np.uint8))
#plt.title(f'{class_names[index]}')
#plt.show()
print("\nFinal time [s]: ", t_f-t)
```

The algorithm is evaluated once again and its results are compared with the category name of each input image. Since also the `img_input` is evaluated it is also possible to print on screen the actual input picture of the model and the picture to which corresponds the category name compared with the AlexNet output. This is done just to see if the right images are evaluated each time.

### 2.2.6 How to select a GPU or a CPU to run the algorithm

On a typical system, there are multiple computing devices. In TensorFlow [20], the supported device types are `CPU` and `GPU`. They are represented as `strings`. For example:

- "/cpu:0": it indicates the CPU of the system;
- "/device:GPU:0": it corresponds to the GPU of the system, if there is any;
- "/device:GPU:1": it points to the second GPU of the system, etc.

However if a TensorFlow operation can be executed on either the CPU or the GPU, the latter device will be always used if not specified differently. To force the execution of the algorithm on the CPU the following line of code has to be added at the beginning of the code:

```
with tf.device("/cpu:0"):
```

To find out which devices the operations and tensors are assigned to, it is sufficient to create a session with `log_device_placement` which is a configuration option and set it to `True`. For example:

```
sess = tf.Session(config=tf.ConfigProto(log_device_placement=True))
```

This line of code will produce the following output:

```
Device mapping:
/job:localhost/replica:0/task:0/device:XLA_CPU:0 -> device: XLA_CPU device
/job:localhost/replica:0/task:0/device:XLA_GPU:0 -> device: XLA_GPU device
/job:localhost/replica:0/task:0/device:GPU:0 -> device: 0, name: Tesla K40c, pci
      bus id: 0000:08:00.0, compute capability: 3.5
IteratorToStringHandle: (IteratorToStringHandle):
      /job:localhost/replica:0/task:0/device:CPU:0
IteratorGetNext: (IteratorGetNext): /job:localhost/replica:0/task:0/device:CPU:0
conv1/weights/Initializer/random_uniform/RandomUniform: (RandomUniform):
      /job:localhost/replica:0/task:0/device:CPU:0
conv1/weights/Initializer/random_uniform/sub: (Sub):
      /job:localhost/replica:0/task:0/device:CPU:0
conv1/weights/Initializer/random_uniform/mul: (Mul):
      /job:localhost/replica:0/task:0/device:CPU:0
conv1/weights/Initializer/random_uniform: (Add):
      /job:localhost/replica:0/task:0/device:CPU:0
conv1/weights: (VariableV2): /job:localhost/replica:0/task:0/device:CPU:0
```

### 2.2.7 How to profile TensorFlow

Being able to profile a TensorFlow graph could be very useful to evaluate the performances of a Neural Network. Thanks to `timeline` module [21] this is quite simple: by adding just a few lines of code it is possible to create a `json` file with profiled data stored in Chrome trace format:

```
...
previous code
...
options = tf.RunOptions(trace_level=tf.RunOptions.FULL_TRACE)
run_metadata = tf.RunMetadata()
output = sess.run(prob, options=options, run_metadata=run_metadata)
fetched_timeline=timeline.Timeline(run_metadata.step_stats)
chrome_trace = fetched_timeline.generate_chrome_trace_format()
with open('timeline_CPU_input_pipeline_batch_size_5.json', 'w') as f:
f.write(chrome_trace)
```

Basically the algorithm is run with options and metadata like shown in the above code and then the `chrome_trace` is written in a `json` file. Once the file has been created, by going on the Chrome browser at the following address

chrome://tracing/

it is possible to load the json file and then something similar to the following timeline can be obtained:

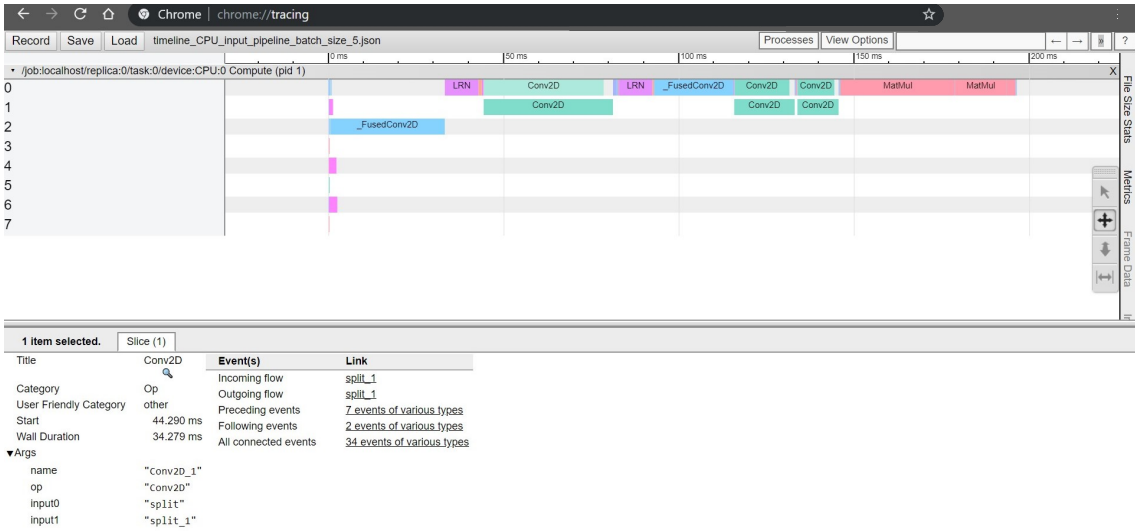


Figure 2.1: CPU profiling.

On the top there is the time axis in ms. To get more precise info about some operation it is sufficient to click on it. Also on the right side, there are simple tools: selection, pan, zoom and timing. On the left side for each process, it is possible to see how many threads are used and which are the operations executed by each one of them. In the image above there is just one process and eight threads: each horizontal track represent a thread on CPU.

While on the GPU by defaults TensorFlow uses as much memory as it can, on CPU it is mandatory to specify the number of threads and core processors to be used by the algorithm in order to evaluate in similar condition the performances. To do that before declaring a session it is sufficient to add the following configuration option:

```
session_conf = tf.ConfigProto(intra_op_parallelism_threads=8,
      inter_op_parallelism_threads=8)
sess = tf.Session(config=session_conf)
```

The best performance can be obtained by declaring both `intra_op_parallelism_threads` and `inter_op_parallelism_threads` equal to the maximum number of threads of the processor.

### 2.2.8 How to evaluate the inference time

In order to evaluate the inference time it is sufficient to use the `time.time()` function before and after the execution of the algorithm and then do a simple subtraction like in the following code:

```
t = time.time()   #Time from epoch (1 january 1970 in seconds)
output=sess.run(prob)
t_f = time.time() #Final time
print("\nFinal time [s]: ", t_f-t)
#print duration time on text_file
with open('duration_batch_size_5.txt','a') as f:
f.write(f"duration[s]: {t_f-t} \n")
```

However it is clear that this algorithm should be executed several times in order to compute the mean value among all the inference times. For this reason each inference time is written in a file opened in `append mode`. Moreover, an additional python script has been written in order to execute the algorithm multiple times. It is clear that another possibility could have been writing a for loop and execute multiple times the code above. However, executing the whole file together is more similar to an actual run of the algorithm. Remember also that TensorFlow does several optimizations when running the code, so the solution with a for loop could have given a no truthful result. The python script is actually very simple:

```
for j in range(100):
import high_level_file_5
importlib.reload(high_level_file_5)
```

Normally when a script is imported once, even if the import statement is written a second time it will not be executed again, while with the function `importlib.reload (python script name)` the compiler is forced to import again the script which means, in this case, execute again the AlexNet algorithm.

### 2.2.9 How to obtain power data of the GPU

Nvidia GPUs come with very handy developer tools which allow profiling deeply every algorithm. However, these tools work well only for codes based on CUDA API. Nevertheless thanks to some command-line tools provided by Nvidia itself, it is possible to measure some parameters of the GPU like temperature, power, used memory and so on. In particular the command `nvidia-smi` provides the following GPU parameters:

```
+-----------------------------------------------------------------------------+
| NVIDIA-SMI 418.67       Driver Version: 418.67       CUDA Version: 10.1     |
|-------------------------------+----------------------+----------------------+
| GPU  Name        Persistence-M| Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp  Perf  Pwr:Usage/Cap|         Memory-Usage | GPU-Util  Compute M. |
|===============================+======================+======================|
|   0  Tesla K40c           Off  | 00000000:08:00.0 Off |                    0 |
| 58%   75C    P0   157W / 235W |    294MiB / 11441MiB |     97%      Default |
+-------------------------------+----------------------+----------------------+

+-----------------------------------------------------------------------------+
| Processes:                                                       GPU Memory |
|  GPU       PID   Type   Process name                             Usage      |
|=============================================================================|
|    0     22113      C   /bin/mumax3                                 283MiB |
```

This is just an example where it is possible to see the power consumption. This means that in order to measure multiple times the power and then make an average among all the values, one needs to iterate this command every tot. amount of time. Fortunately an option for this command is `-lms numb` which basically means repeat every numb (expressed in ms) the command nvidia-smi.

There are still two problems: the command needs to be executed just when the algorithm is run, no sooner no later and furthermore when the option to repeat the command is given, the process related to that command will run forever if not stopped.

To solve all the above problems the following code can be used and can be implemented directly in the python script of the algorithm:

```
nvidia_cmd=subprocess.Popen(["nvidia-smi", "-lms", "1"], stdout=subprocess.PIPE,)
grep_cmd = subprocess.Popen(["grep", "MiB"], stdin= nvidia_cmd.stdout,
      stdout=subprocess.PIPE,) #second command
output = sess.run(prob)
#get process PID and kill it
```

```
pid = nvidia_cmd.pid
pid2=grep_cmd.pid
os.kill(pid, signal.SIGKILL)
os.kill(pid2, signal.SIGKILL)
if not nvidia_cmd.poll() and not grep_cmd.poll():
print ("Process correctly halted")
end_of_pipe=grep_cmd.stdout
with open('power_batch_size_5.txt','a') as f:
for line in end_of_pipe:
f.write(str(line)+"\n")
```

The first two lines basically represent the bash command:

$$>> nvidia - smi \quad -lms \;\; 1 \;\; |grep \;\; "MiB"$$

while the others represent just the python way to kill the process once finished and write the result on a file opened in `append` mode. For example the result of these lines of code may be something like this:

```
| 24%   47C    P0    63W / 235W |  11019MiB / 11441MiB |     13%      Default |
|    0     18625      C   python3.7                                  11006MiB |
| 24%   47C    P0    63W / 235W |  11019MiB / 11441MiB |     13%      Default |
|    0     18625      C   python3.7                                  11006MiB |
| 24%   47C    P0    63W / 235W |  11019MiB / 11441MiB |     13%      Default |
|    0     18625      C   python3.7                                  11006MiB |
| 24%   47C    P0    62W / 235W |  11019MiB / 11441MiB |     13%      Default |
|    0     18625      C   python3.7                                  11006MiB |
| 24%   47C    P0    65W / 235W |  11019MiB / 11441MiB |     13%      Default |
|    0     18625      C   python3.7                                  11006MiB |
```

It is worth highlighting that the GPU consumes power even if not used: this is called idle power. To obtain the true power consumption the idle power has to be subtracted to the measured one. By running the nvidia-smi command when nothing is running on the GPU the following power is obtained:

$$P_{Idle} = 20\,\mathrm{W}$$

As done previously to measure the inference time, a python script that runs a certain number of times the AlexNet algorithm has been written. However since each nvidia-smi command gives more than one sample, each iteration has to be separated in order to be able to do the average among all the samples. The script again is very similar to the previous one:

```
for j in range(10):
import high_level_file_5
with open('power_batch_size_5.txt', 'a') as f:
f.write("---------------------- \n")
importlib.reload(high_level_file_5)
with open('power_batch_size_5.txt', 'a') as f:
f.write("---------------------- \n")
```

The only difference with respect to the previous script for the inference time is that after each iteration the file is opened and two dash lines are added (the first time the script is imported the added line is only one).

There is more to say about the results obtained by this script: at each iteration, a different number of values are provided. Since at each iteration the measured values are very similar it is possible to select among all, the iterations where there is the highest number of samples and then compute the average among them in order to obtain a reasonable mean value. It has to be remarked that the mean value takes into account the number of samples so it is mandatory to first extract only the iterations with the same number of samples and then compute the average value.

To simplify the script inside each dash line two `W` letters have been inserted to obtain something like this:

```
|    0     18625      C   python3.7                                    11008MiB |
| 25%   50C    P0    69W / 235W |  11021MiB / 11441MiB |     17%      Default |
|    0     18625      C   python3.7                                    11008MiB |
------W---------W----
------W---------W----
```

and then the following bash command has been executed

```
cat power_batch_size_5.txt | grep W | tr -s ' ' |  cut -d 'W' -f 1 | cut -d ' ' -f
     5 > extracted_batch_size_5.txt
```

In this way the following result is obtained and saved in the file `extracted_batch_size_5.txt`:

```
63
63
63
62
65
------
64
63
```

```
63
```

Now Matlab scripts have been written to obtain the final data:

```
matrix_out = classify_power_values('extracted_batch_size_5.txt');
line_to_write = mean(matrix_out,2);
f_out = fopen('power_batch_size_5.txt','w');
for index = 1:(length(line_to_write))
fprintf(f_out,'%f\n', line_to_write(index));
end
fclose(f_out);
```

In this first script the function `classify_power_values` is called and it basically extracts from each file only the data with the same number of samples as discusses before. The code of this function is the following one:

```
function [matrix1] = classify_power_values(file_name)
%function Summary of this function goes here
%   Detailed explanation goes here
f = fopen(file_name);
i = 1;
n_max = 0;
k = 1;
vect2 = zeros(5,1);
matrix_f = zeros(n_max, 1);
temp = 0;
while ~feof(f)
line1 = fgetl(f);
if(strcmp(line1,'------'))
%new simulation data
if(temp == 0)
temp = 1;
i = 1;
n = length(vect2);   %% # of elements in the array
if(n > n_max)
n_max = n;
k = 1;
clear matrix_f;
matrix_f = zeros(n_max, 1);
matrix_f(:,k) = vect2;
k = k+1;
clear vect2;
elseif(n == n_max)
matrix_f(:,k) = vect2;
k = k+1;
clear vect2;
else
clear vect2;
end
```

```
34 end
   else
36 vect2(i) = str2double(line1);
   i = i+1;
38 temp = 0;
   end
40 end
   fclose(f);
42 matrix1 = matrix_f;
   end
```

Now this function extracts the data with the same number of samples and then the main script computes automatically the mean value among samples at the same position. This is done because the iterated nvidia-smi command gives samples over time so if the algorithm is iterated multiple times, it is possible to obtain an average value at each iteration and by joining together all the iterations, the power consumption over time can be computed.

# Chapter 3

# Comparison GPU vs CPU

Neural Network algorithms perform differently if a GPU or CPU is used. There are mainly two different performance indicators [22] which have to be considered when comparing a CPU and a GPU for neural network algorithms like AlexNet:

- Processing speed;
- Data transfer speed from the memory.

The processing speed depends on a various number of factors like cores number, clock speed (frequency) and last but not least number of instructions completed in each clock cycle (IPC, instructions per cycle). The IPC depends heavily on the application. The theoretical number is two to four instructions per cycle. However, the practical performance can be more or less close to the maximum based on if the processor is able to to get all necessary data on time. In the case of a memory-bound algorithm, the performance is easily reduced to a small percentage of the maximum.

For GPUs, processing units are highly different from general-purpose CPUs: the processing speed is much greater (mainly thanks to a large number of specialized cores though with lower frequency) and also the throughput to the memory is much greater due to wider buses usually.

This means that algorithms which are limited by processing speed and by memory can perform much better if executed on GPUs. Moreover, today's GPUs offer also double precision floating point capabilities.

To understand better all the differences listed before it might be useful to analyze the typical structure of an Intel i7 CPU and a NVIDIA Tesla GPU.

## 3.1 CPU structure

The generic structure of an Intel i7 CPU is shown in the Figure 3.1. There are four

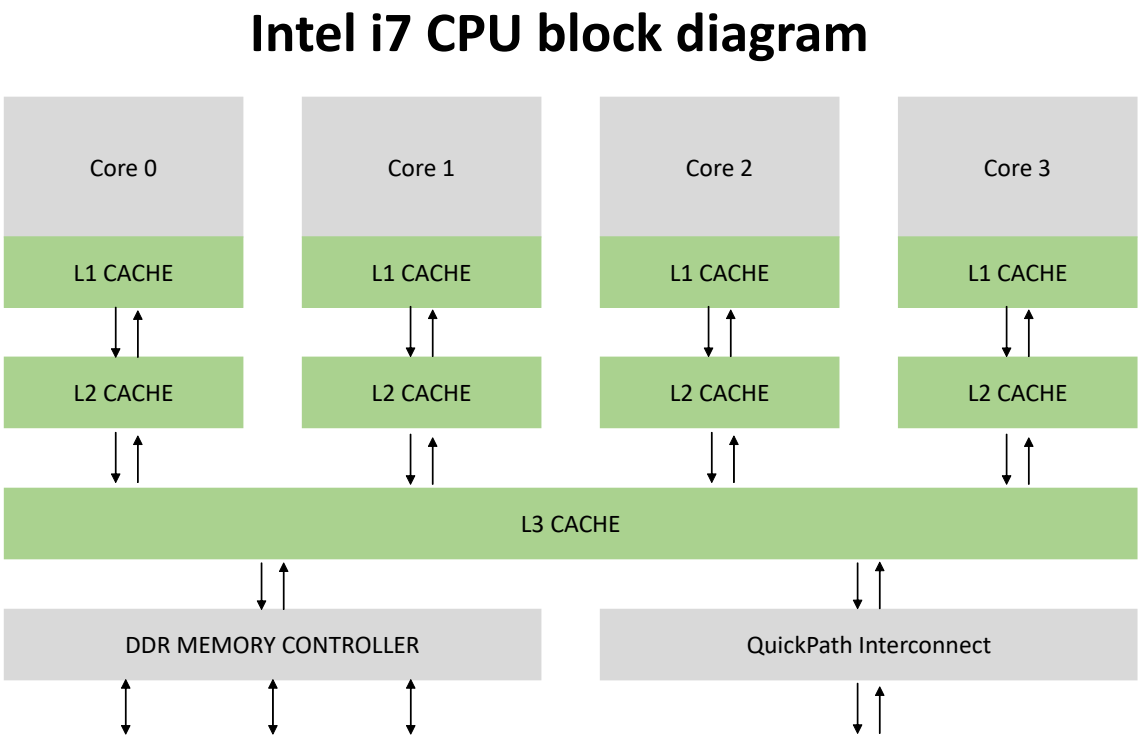


Figure 3.1: i7 CPU architecture.

cores, three levels of cache and a DDR memory controller. Everything in this scheme is standard except for the `QPI` (or `QuickPath Interconnect`). The `QPI` is a new serial bus introduced by Intel in 2008 in some desktop and server processors which basically replaces the `Front-Side Bus` and increases the scalability and the available bandwidth. The Front-Side Bus (`FSB`) is the one which transports data between CPU and Northbridge, knows also as Memory Controller Hub or Host Bridge. The Northbridge is responsible for tasks that require the highest performances.

In this comparison chapter, the Intel i7 930 has been used. The main specifications are:

- Clock frequency: 2.8 GHz (4 cores, 8 threads);
- DDR3 Memory bandwidth: 25.6 GB/s;
- DDR3 clock frequency: 1066 MHz;
- FSB bus width: 192 bits;

- Flop/cycle: 8 FP32 Flop/cycle and 4 FP64 Flop/cycle;
- Peak FP performance: 89.6 Gflops (FP32) and 44.8 Gflops (FP64).

There are few things to be highlighted here: FLOP means floating point operations and the Peak FP performance is computed by means of the following formula:

$$Peak\ FP\ Performance = \#\ cores \cdot f_{clk} \cdot Flops/cycle \tag{3.1}$$

Then based on which type of data is used (single precision 32 bits or double precision 64 bits) it is possible to obtain an indicator for single and double precision peak FP performance.

## 3.2 GPU structure

The generic architecture of a NVIDIA TESLA GPU is depicted in Figure 3.2. From the above picture it is possible to highlight the most important elements of a Tesla GPU architecture:

- Host interface and Compute work distribution: it takes instructions and data from Host CPU and its main memory respectively. Furthermore, it manages the threads of executions which means that it is responsible to assign groups of threads to processor clusters.
- TPC: it stands for texture/processor clusters which are the basic building block of a NVIDIA Tesla architecture. A single GPUs can have up to eight TPCs.

Each TPC contains two `Streaming multiprocessor (SM)`. As shown in the Figure 3.3 in each `SM` there are at least eight `Streaming Processors cores (SP)` and two special function units. In every TPC there is also a `Texture unit` and a memory: both are shared between the two SMs. SPs are often called `CUDA` core. In the NVIDIA TESLA K40c (the one used in this comparison) there are 2880 `CUDA` cores. This means that today's GPUs are well beyond the very simple scheme shown in Figure 3.2. The main specs of the NVIDIA TESLA K40c are:

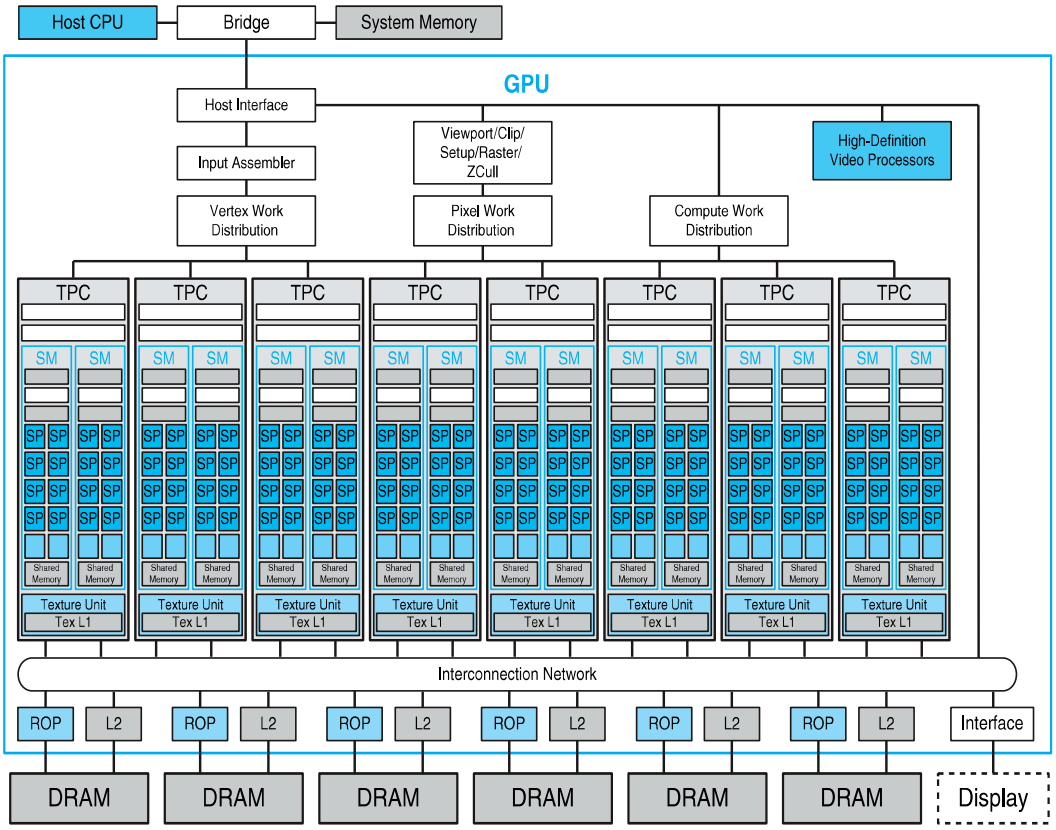


Figure 3.2: Tesla GPU architecture [9].

- Clock frequency: 745 MHz (2880 cores);
- Boost clock frequency: 876 MHz;
- DDR5 Memory bandwidth: 288.4 GB/s;
- DDR5 clock frequency: 1502 MHz;
- Interface bus width: 384 bits;
- Flop/cycle: 3 FP32 Flop/cycle and 1 FP64 Flop/cycle;
- Peak FP performance: 5.046 Tflops (FP32) and 1.682 Tflops (FP64).

As it can possible to see, though the clock is slower with respect to the CPU one, there are so many cores which allow obtaining much better performance. Another important element is the memory bandwidth which is one order of magnitude higher

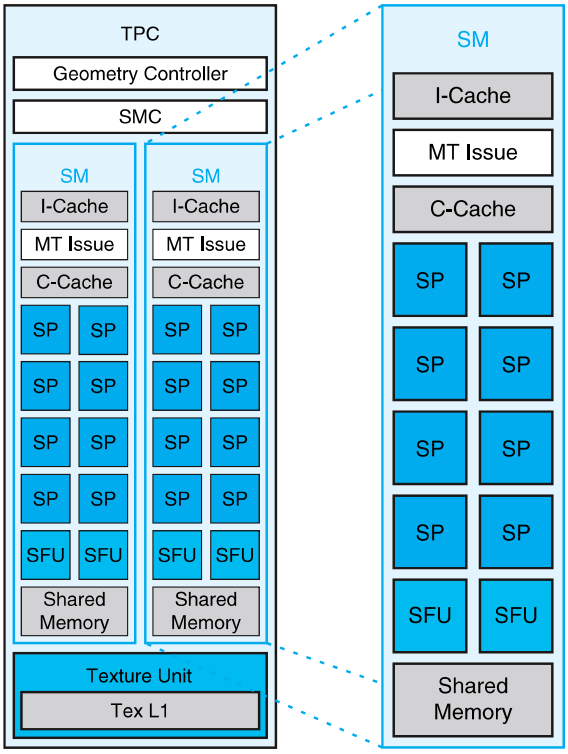


Figure 3.3: Tesla GPU architecture single cluster [9].

than the CPU one. Overall it is possible to say that the GPU will perform better than a CPU in every algorithm which is capable of using at the best all the cores and all the memory bandwidth.

## 3.3 AlexNet implementation: GPU vs CPU

The AlexNet neural network has been implemented on CPU and GPU and an extensive profiling of the algorithm has been performed with the code explained in the chapter 2. Performance indicators such as total and layer by layer delay, power consumption and frame per second have been evaluated.

For clarity purposes, the AlexNet layers are summarized in the Table 3.1 while the detailed explanation can be found in the subsection 1.3.1.

| Layer | Filter size | Output channel | Stride | Padding |
|---|---|---|---|---|
| **convolution 1 + ReLU** | 11x11 | 96 | 4 | 0 |
| **Normalization** | - | - | - | - |
| **Max Pooling** | 3x3 | - | 2 | - |
| **convolution 2 + ReLU** | 5x5 | 256 | 1 | 2 |
| **Normalization** | - | - | - | - |
| **Max Pooling** | 3x3 | - | 2 | - |
| **convolution 3 + ReLU** | 3x3 | 384 | 1 | 1 |
| **convolution 4 + ReLU** | 3x3 | 384 | 1 | 1 |
| **convolution 5 + ReLU** | 3x3 | 256 | 1 | 1 |
| **Max Pooling** | 3x3 | - | 2 | - |
| **Fully-Connected + ReLU** | - | 4096 | - | - |
| **Fully-Connected + ReLU** | - | 4096 | - | - |
| **Fully-Connected** | - | 1000 | - | - |
| **Softmax** | - | - | - | - |

Table 3.1: AlexNet layers summarized.

All the results in this section have been carried out with an Intel i7 930 as CPU and a Nvidia TESLA K40c as GPU.

The first comparison is related to the layer by layer delay ratio between CPU and GPU. Having in mind the different characteristics of GPUs and CPUs it is possible to foresee some results: in particular layers like convolution, normalization and fully-connected ones are likely to be slower on CPUs than on GPUs and by increasing the batch size this ratio is expected to grow higher. For the other layers, it is difficult to predict because they have relatively simple operations.

It is important to highlight that while GPUs performances do not change significantly from one run to the other, CPUs performances are strongly affected by the number of threads used at each run. Even though Tensorflow allows to set the number of cores and threads to be used it is not sufficient to guarantee that at each iteration the code uses always the same number of threads. In Figure 3.4 the number of threads used by the CPU to vary the batch size is shown.

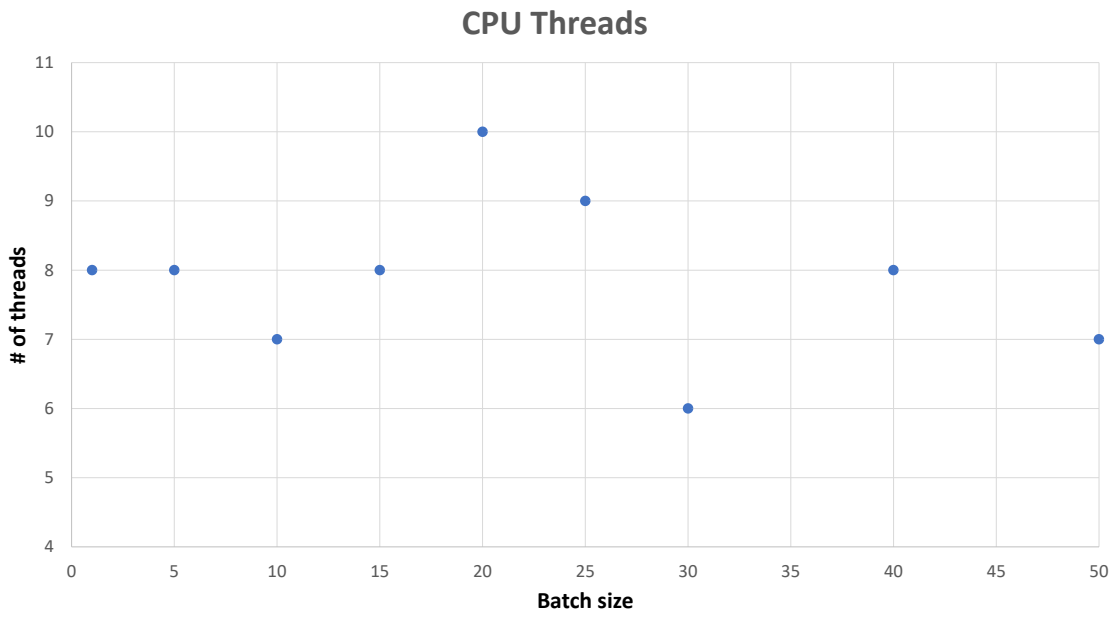


Figure 3.4: CPU number of threads to vary the batch size.

This variation of the number of threads affects the performances and it may cause variation among different batch sizes of the ratio between CPU and GPU layer by layer delay.

In the following table, the delay ratio between CPU and GPU is presented for each operation performed in the AlexNet including reshaping and split operations which are executed to adapt the dimension of a layer output to be accepted as input for the successive layer. Data for batch sizes 1, 5 and 10 are reported.

| Layer | CPU delay/GPU delay | | |
|---|---|---|---|
| | batch size = 1 | batch size = 5 | batch size = 10 |
| **convolution 1 + ReLU** | 6.10 | 21.17 | 27.09 |
| **Normalization** | 24.98 | 5.38 | 4.48 |
| **Max Pooling** | 7.88 | 5.21 | 17.47 |
| **Split** | 4.10 | 11.60 | 32.37 |
| **convolution 2** | 11.47 | 23.14 | 39.14 |
| **ReLU** | 15.00 | 5.58 | 6.86 |
| **Normalization** | 9.94 | 30.55 | 17.84 |
| **Max Pooling** | 5.59 | 4.16 | 4.90 |
| **convolution 3 + ReLU** | 10.46 | 20.39 | 19.42 |
| **Split** | 4.00 | 5.55 | 6.11 |
| **convolution 4** | 11.79 | 15.54 | 13.87 |
| **ReLU** | 4.50 | 13.00 | 6.75 |
| **Split** | 3.57 | 4.64 | 6.85 |
| **convolution 5** | 8.74 | 14.52 | 10.21 |
| **ReLU** | 3.4 | 11.00 | 9.00 |
| **Max Pooling** | 2.20 | 4.00 | 3.55 |
| **Reshape** | 0.54 | 0.73 | 0.89 |
| **Fully-Connected 6** | 25.25 | 38.92 | 16.04 |
| **ReLU** | 1.20 | 2.00 | 3.40 |
| **Fully-Connected 7** | 23.36 | 36.26 | 15.16 |
| **ReLU** | 1.20 | 2.00 | 3.40 |
| **Fully-Connected 8** | 13.74 | 18.91 | 10.54 |
| **Softmax** | 1.80 | 2.79 | 4.97 |

Table 3.2: AlexNet layer by layer delay ratio between CPU and GPU to vary the batch size.

As it is possible to see from Table 3.2, the general behaviour is that by increasing the batch size, the ratio increases which means that the GPU becomes more convenient to use for high batch sizes. However, there are some strange data: normalization layers are incoherent because sometimes the ratio increases and sometimes decreases significantly to vary the batch size and FC layers delay ratio also increases and decreases without an obvious reason. To understand better why this strange behaviour happens it is fundamental to analyze not only the ratio but also the delays of CPU and GPU layers separately.

What is important to understand from this table is that except for the reshaping, all the other operations are faster on a GPU. This result is very important because

it validates all the initial hypothesis on the performance indicators which have to be considered when comparing a CPU and a GPU for neural network algorithms (chapter 3). For what concerns the reshape operation, its time duration can be neglected if compared with all the other layers.

From now on, the delays of the most important layers are compared for both CPU and GPU implementation.

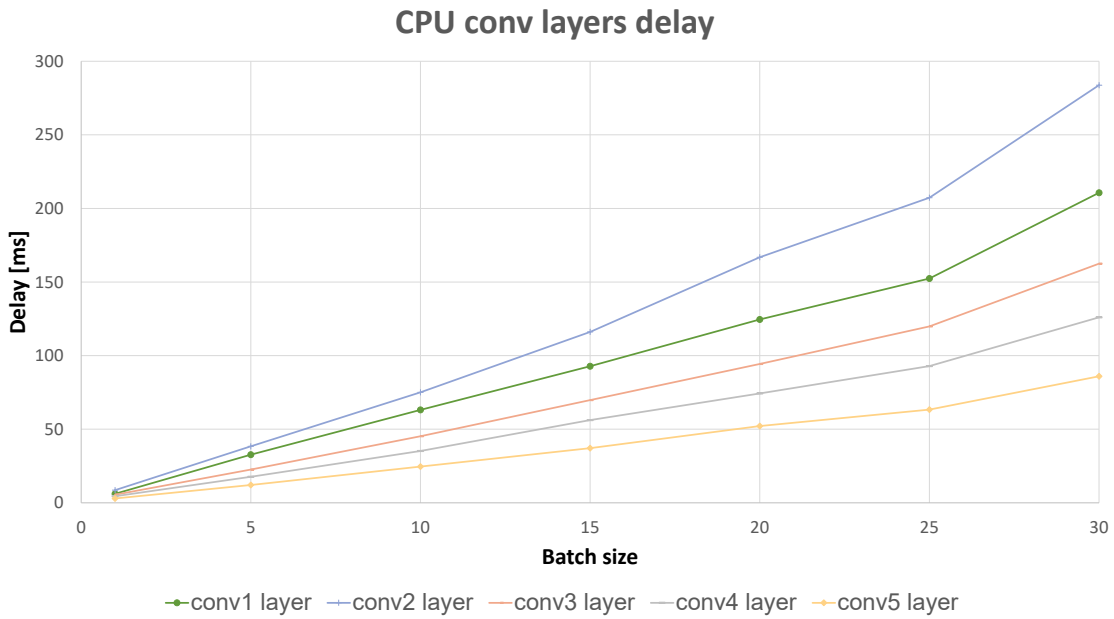


Figure 3.5: CPU convolutional layers delay vs batch size.

In Figure 3.5 it is possible to see how the convolutional layers delay increases almost linearly with the batch size. This behaviour is expected since the number of images on which the convolution is applied increases. However, it is worth highlighting how the convolutional layer 2 takes more time to end then all the others. This is due to the number of input and output channels: the second convolutional layer takes as input 96 channels which are higher than the input channels taken by the first one and produces 256 output channels which again is higher than the first one. Even so, this does not explain why the successive convolutional layers take less time to finish the convolution. If one analyzes in more details the AlexNet algorithm, it is straightforward to see that the second convolutional layer uses a 5x5 filter while the next ones use a 3x3 filter.

For what concerns the delay of other convolutions, convolution 3 is slower than the 4th one which is slower than the last one. This might seem strange because the last three convolutions are very similar. However, it has to be considered that convolution 2, 4 and 5 are actually split into two parts and executed simultaneously while convolution 1 and 3 are not. So, for this reason, the third convolution is slower than 4th and 5th ones, while the last one is faster than the 4th convolution because it has fewer output channels.

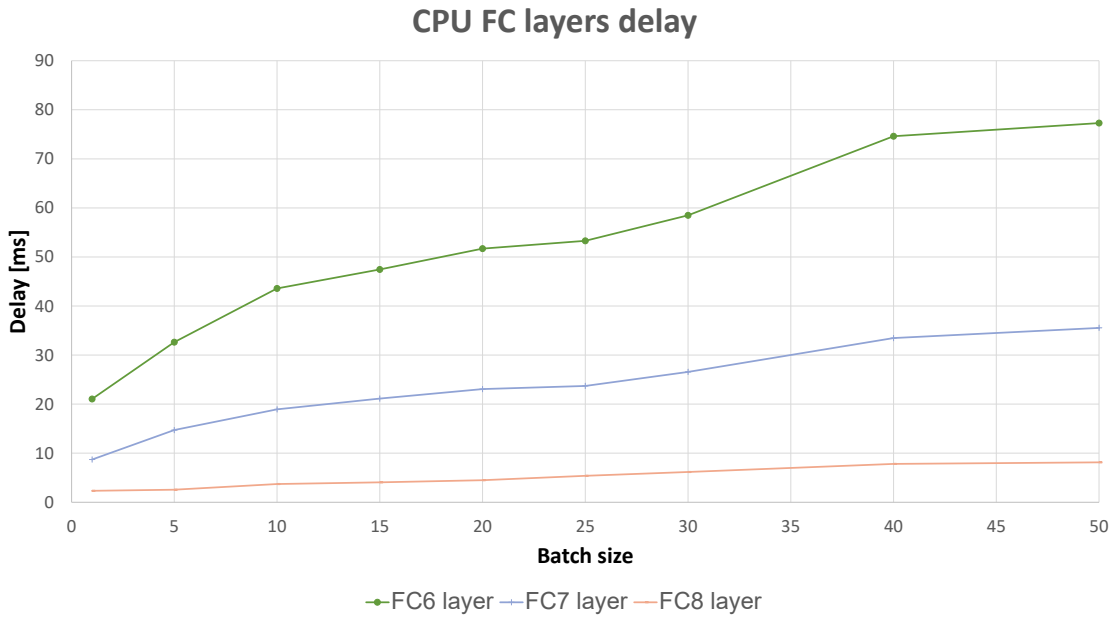


Figure 3.6: CPU FC layers delay vs batch size.

Figure 3.6 shows the delay of fully-connected (`FC`) layers to vary the batch size. It is straightforward to understand that by going towards the end of the algorithm the delay of the layers decreases because the dimension of the outputs decreases as well: the sixth FC layer takes more time than the seventh and eighth layer. This because of the input dimension of the FC6 which is bigger than the FC7, while FC8 has fewer output channels.

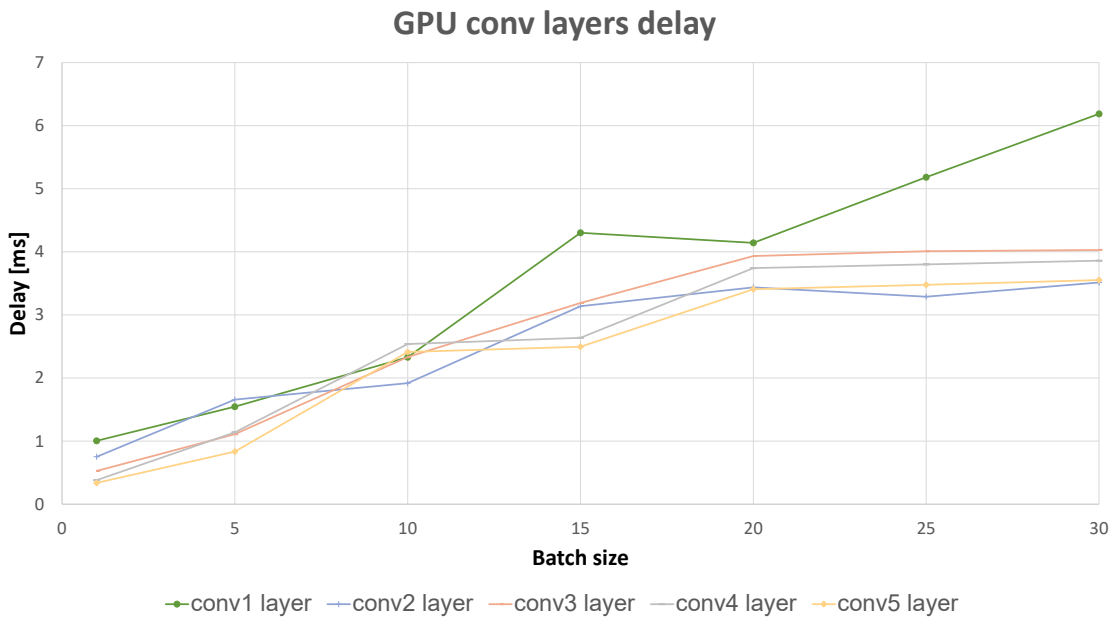


Figure 3.7: GPU convolutional layers delay vs batch size.

In Figure 3.7 it is possible to see the convolutional layers delay executed by a GPU. Differently from the CPU, all convolutional layers have a very similar delay except for the first one which diverges for high batch sizes due to the bigger filter size and the bigger input matrix.

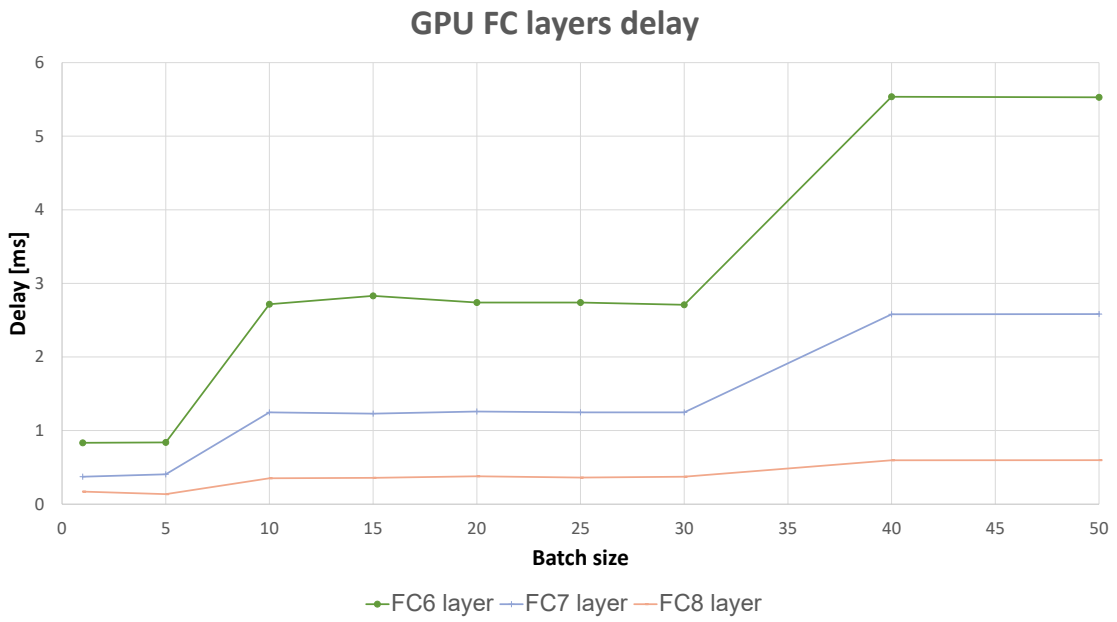


Figure 3.8: GPU FC layers delay vs batch size.

The behaviour of the FC layers (Figure 3.8) on GPU has a surprisingly different behaviour than the expected one. In fact, the delay of all the FC layers remains constant from a batch size equals to 10 to a batch size of 30. Then it increases a little bit and becomes constant again.

This strange behaviour is due to the high computation required by the FC layers which dominates the delay despite the actual dimension of the batch size. As a matter of fact, increasing the batch size means doing the same number of operations on more images but since the GPU has a highly parallelized structure, it can handle the additional operations simultaneously without weighing too much on the delay. Clearly, at a certain point, the batch size dimension will require the GPU to take more time to finish all the layer operation (as happens for bath size greater than 30) due to resources constraints.

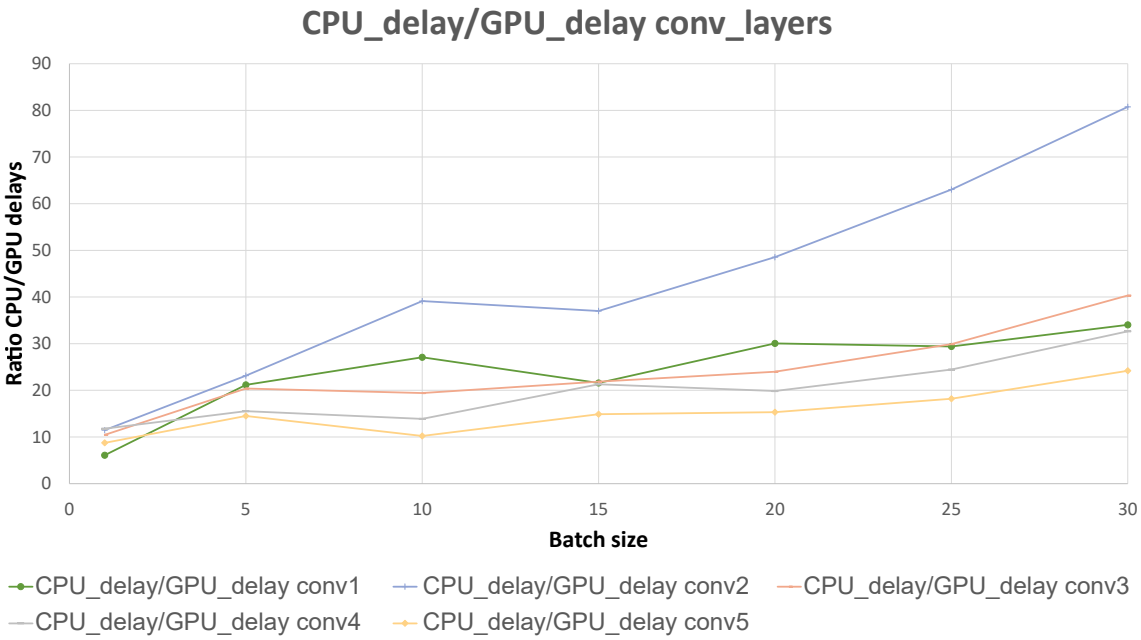


Figure 3.9: CPU convolutional layer delay/GPU convolutional layer delay.

By plotting the delay ratio between convolutional layers of CPU and GPU (Figure 3.9), it is possible to see that the first convolutional layer, due to its higher input dimension and its bigger filter size, has the highest ratio. This means that more than the others, it is the first layer which benefits more from a GPU implementation. Even so, the ratio of all convolutional layers delay is greater than 10 for every batch size analyzed and since it has an increasing trend it is possible to predict that for higher batch sizes the ratio will increase.

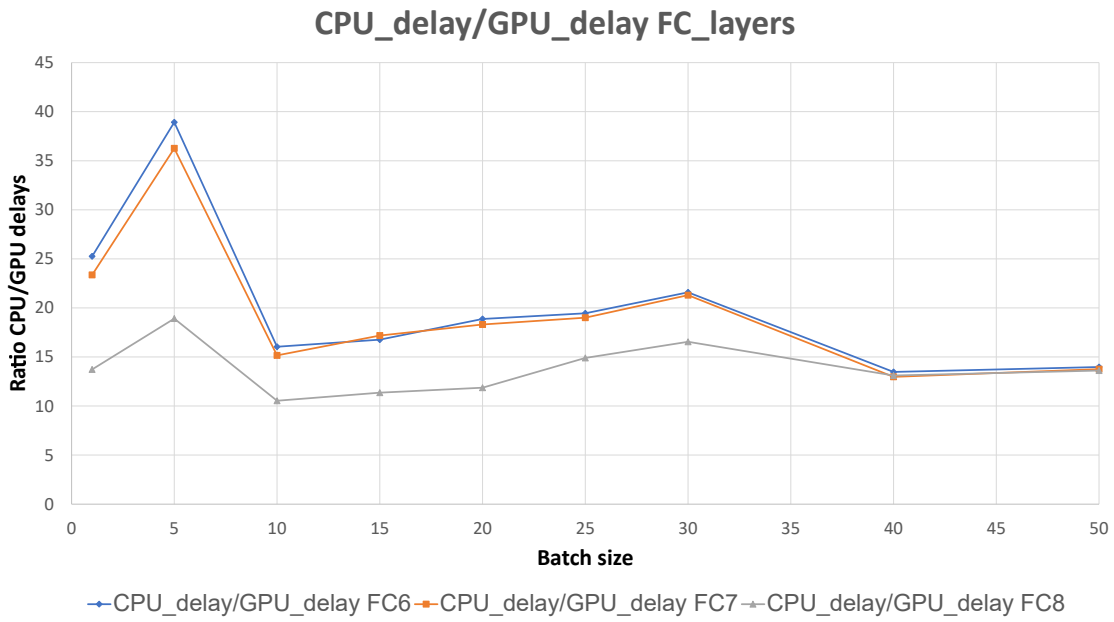


Figure 3.10: CPU FC layer delay/GPU FC layer delay.

The delay ratio concerning the FC layers is quite different from the convolutional ones. However, it has to be considered that this ratio is strongly affected by the FC layers behaviour of the GPU implementation which does not increase constantly. As a matter of fact, in Figure 3.10 the ratio has an alternating behaviour for each FC layer because while the GPU delay is constant, the CPU one always increases. For this reason, the ratios reach highest values only when the GPU delay is constant and decreases as soon as the GPU delay increases.

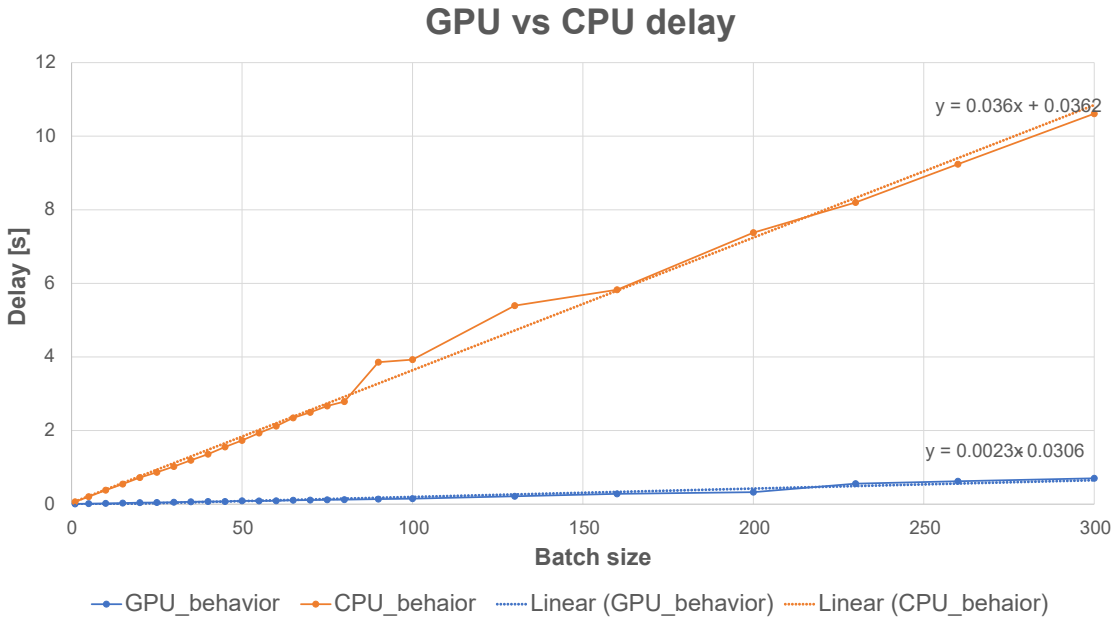


Figure 3.11: GPU and CPU delay vs batch size.

In Figure 3.11, the overall delay of the AlexNet implementation on GPU and CPU is compared to vary the batch size.

As expected the GPU is able to execute the whole algorithm much faster than the CPU. From the graph, it also seems that the total GPU delay is not affected too much by the batch size showing a linear behaviour with a very low angular coefficient.

On the other hand, the total CPU delay shows a more marked linear behaviour with a linear coefficient which is one order of magnitude higher than the GPU one. It has some irregularity around batch sizes 100 and 130 probably due to the number of threads which is not always constant.

It is worth highlighting that the angular coefficients of the linear approximations indicate how much time the CPU and the GPU implementation needs to execute the algorithm per batch sizes.

These coefficients are:

$$m_{CPU} = 0.036 \, \frac{\text{s}}{\text{batch size}}$$

$$m_{GPU} = 0.0023 \frac{\text{s}}{\text{batch size}}$$

This means that the GPU implementation is one order of magnitude faster than the CPU for all the batch sizes analyzed.

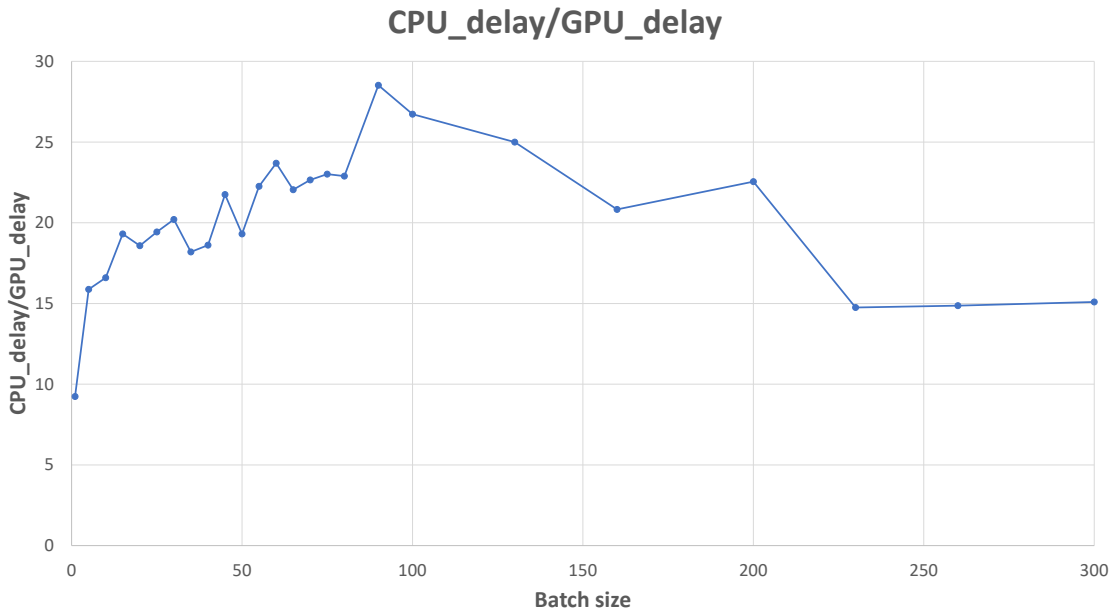


Figure 3.12: CPU delay/GPU delay vs batch size.

In Figure 3.12 the ratio between the CPU AlexNet total delay and the GPU one to vary the batch size is reported. The ratio is expected to be always increasing. However, there are some up and down due to the CPU delay that has some irregularity due to the number of threads which is not always constant while the GPU one increases linearly to vary the batch size.

To compute the AlexNet delay, several iterations have been considered for every batch size and then a mean value has been used. In the following figures, it is possible to see the delay distribution for batch size equals to 1 over 100 iterations for both CPU (Figure 3.13) and GPU (Figure 3.14).

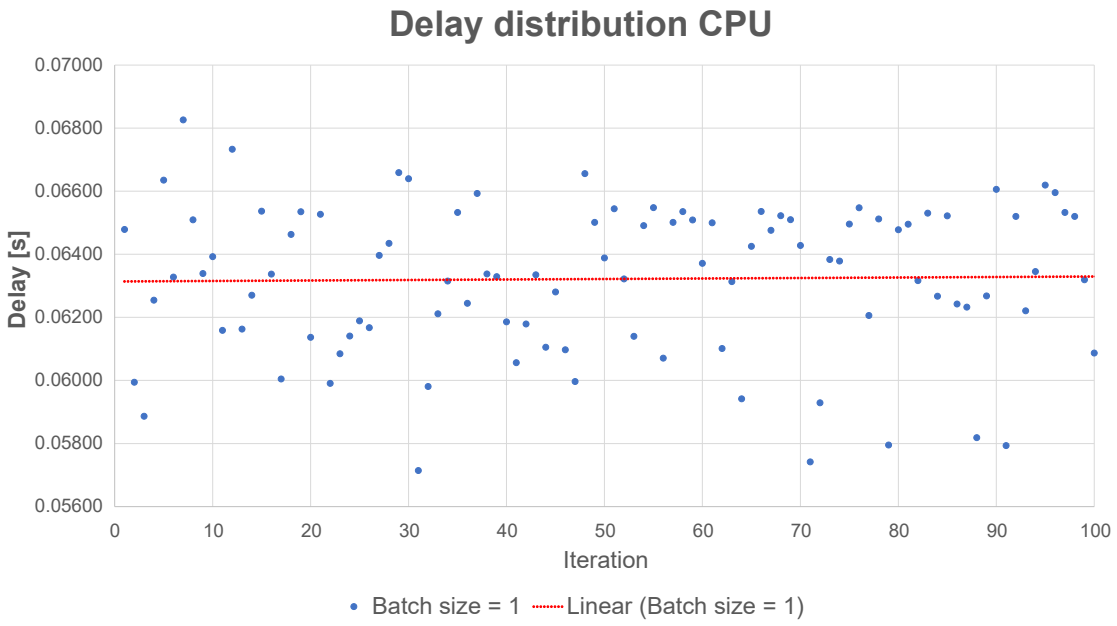


Figure 3.13: CPU delay distribution for batch size = 1.

For what concerns the CPU delay, it is possible to see that the linear interpolation between all the measured values can be considered constant among all the iterations. This means that the plotted distribution has more or less a fixed mean value.

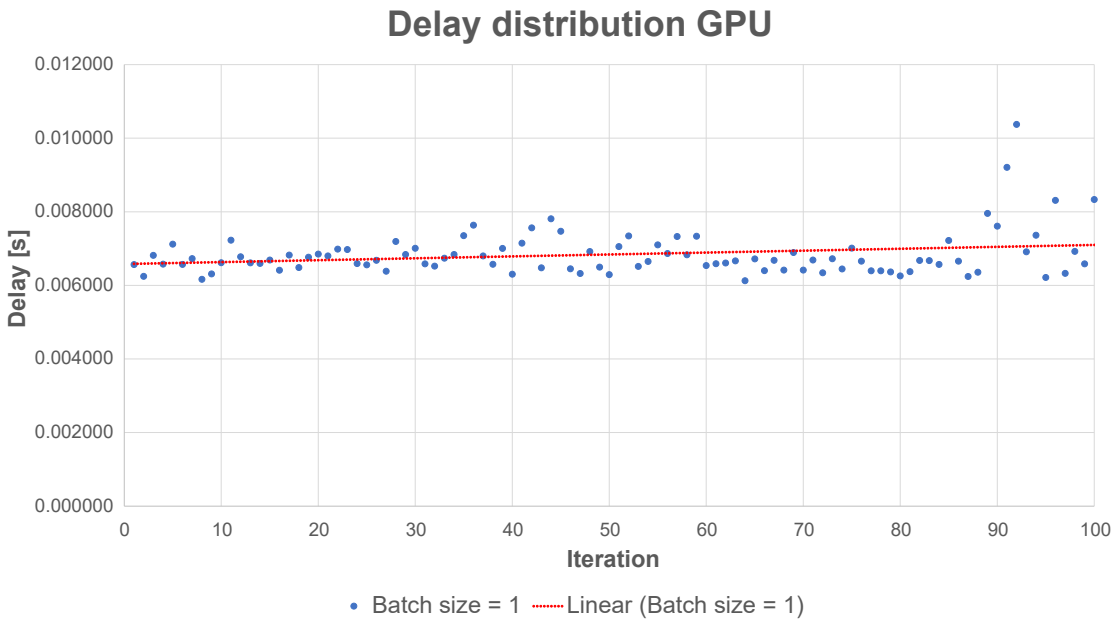


Figure 3.14: GPU delay distribution for batch size = 1.

The GPU delay distribution (Figure 3.14) shows a behaviour which is very similar to the CPU one except for the final iterations where there are some peak values which are much higher than the previous one. Due to those values, the linear interpolation is not a constant value but it is an increasing line. However, since there are few values which are far from the others, they can be considered as noise and can be neglected when computing the mean value.

To measure the performances it may be useful to compute how many multiply and accumulate operations (`MACs`) per second are executed from GPU and CPU and then compare them. This number expresses the efficiency of CPU and GPU for neural networks algorithms.

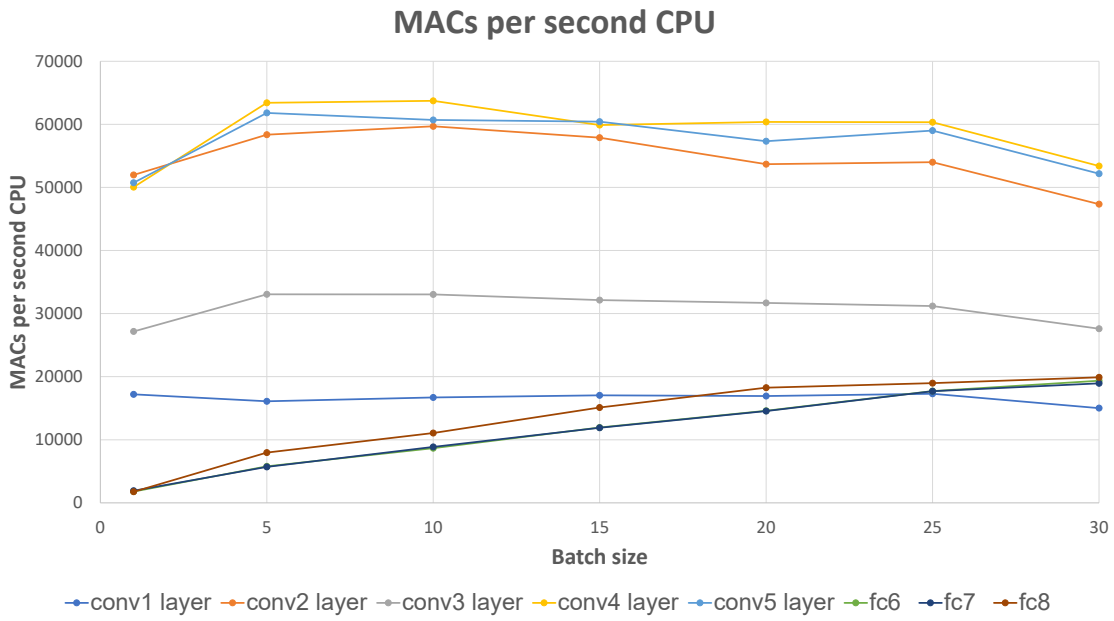


Figure 3.15: MACs per second CPU vs batch size.

In Figure 3.15 it possible to see how the number of MACs per second changes layer by layer. By increasing the batch size this number remains more or less the same although FC layers tend to increase their efficiency to vary the batch size. It has to be highlighted that the FC6 layer is overlapped with the FC7 layer because the efficiency of these two layers is almost the same.

However, this behaviour starts to change around batch size equals to 30 because the CPU has reached its maximum performance and the efficiency starts to drop.

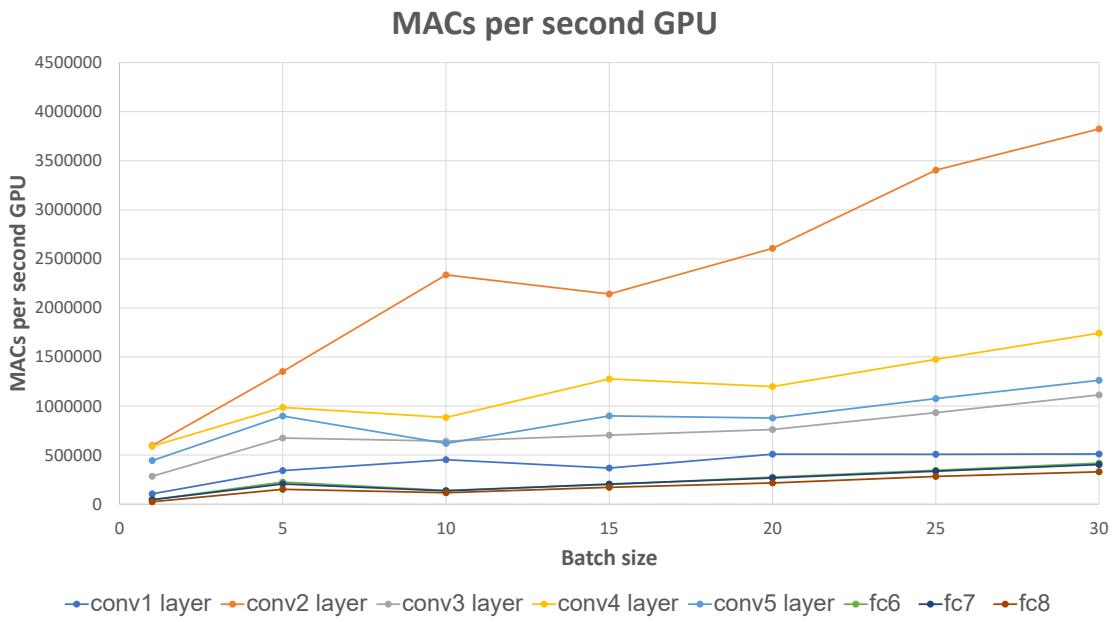


Figure 3.16: MACs per second GPU vs batch size.

For what concerns the GPU, the number of MACs per second increases by increasing the batch size. This behaviour means that the GPU has not reached its maximum performance even with batch size equals to 30. Also in this figure, the FC6 layer is overlapped with the FC7 layer because the efficiency of these two layers is almost the same.

This difference in terms of maximum performance between CPU and GPU is due to both the higher memory bandwidth and the higher number of cores of GPUs.

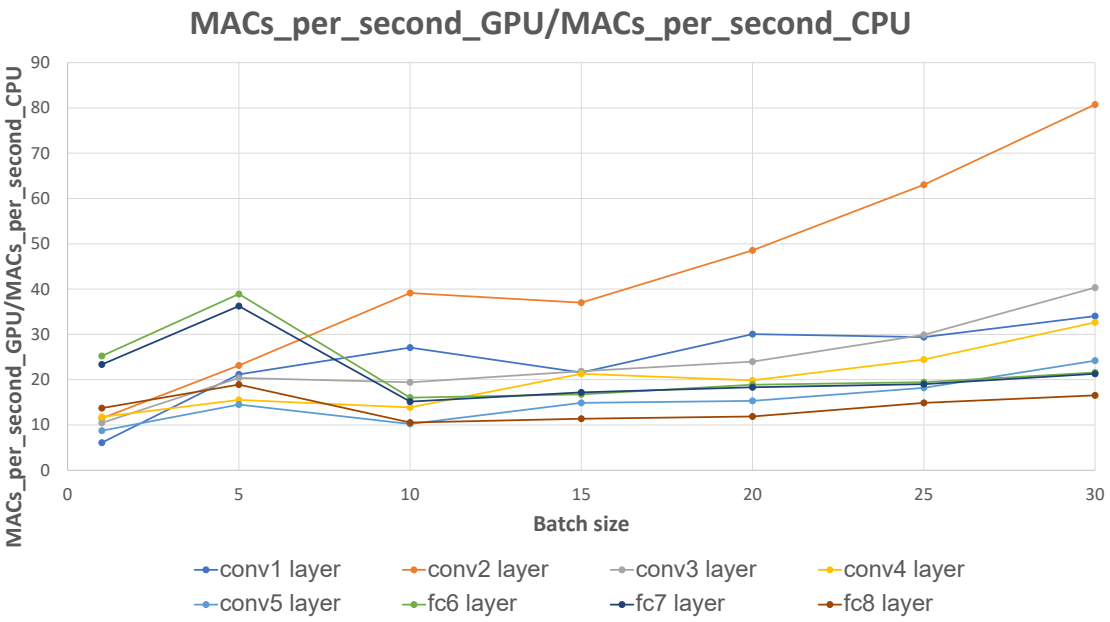


Figure 3.17: MACs per second GPU/MACs per second CPU vs batch size.

In Figure 3.17 it is possible to appreciate the ratio between the two previous graphs (Figure 3.15 and Figure 3.16). This ratio is deeply influenced by the GPU contribution which increases to vary the batch size while the CPU one is more or less constant.

It is worth highlighting that the ratio between the two efficiencies is always greater than 10 which underlines how good a GPU is for this type of algorithms.

Another figure of merit to evaluate the performance of a neural network algorithm, like AlexNet, which recognizes images, is the frame per second. This performance indicator can be computed as follows:

$$\frac{batch\ size}{delay}$$

where the denominator is the total delay of the algorithm of a particular batch size.

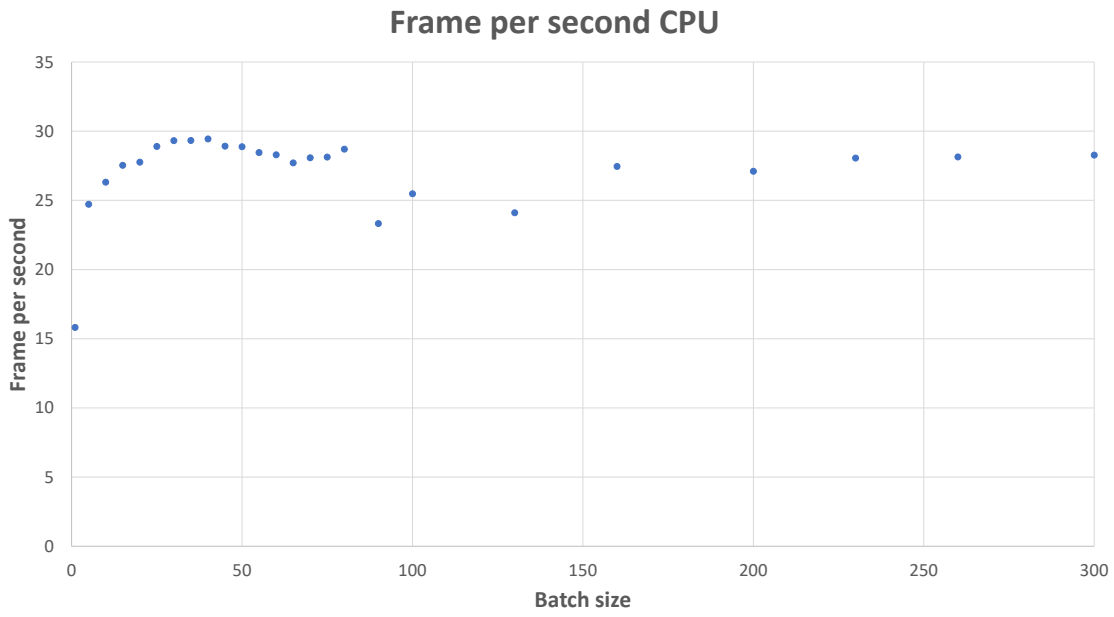


Figure 3.18: Frame per second CPU vs batch size.

In Figure 3.18, it is possible to see how the `FPS` are always below 30.

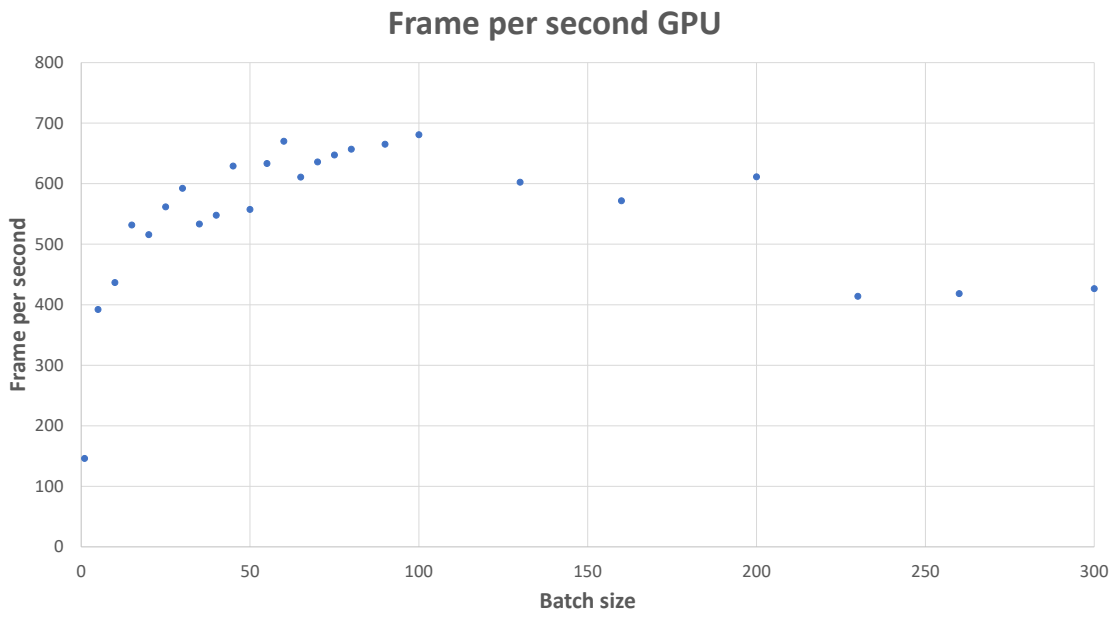


Figure 3.19: Frame per second GPU vs batch size.

The `FPS` for a GPU should be higher than the CPU ones since all the data analyzed until now say that the GPU performs better than the CPU for neural network algorithms.

As a matter of fact the maximum FPS value for a GPU is around 680. This means that the ratio between CPU FPS and GPU ones is:

$$FPS_{ratio} = \frac{FPS_{GPU}}{FPS_{CPU}} = 22.67$$

which means that the GPU is able to analyze almost 23 times the images analyzed by the CPU in the same amount of time.

The last performance indicators are the power and the energy needed to obtain all the performances seen until now.

Unfortunately only the power consumption of the GPU can be measured thanks to the `nvidia-smi` command. The CPU power consumption can not be measured without a specific instrument and even so there is always some background process which will affect the final measurements.

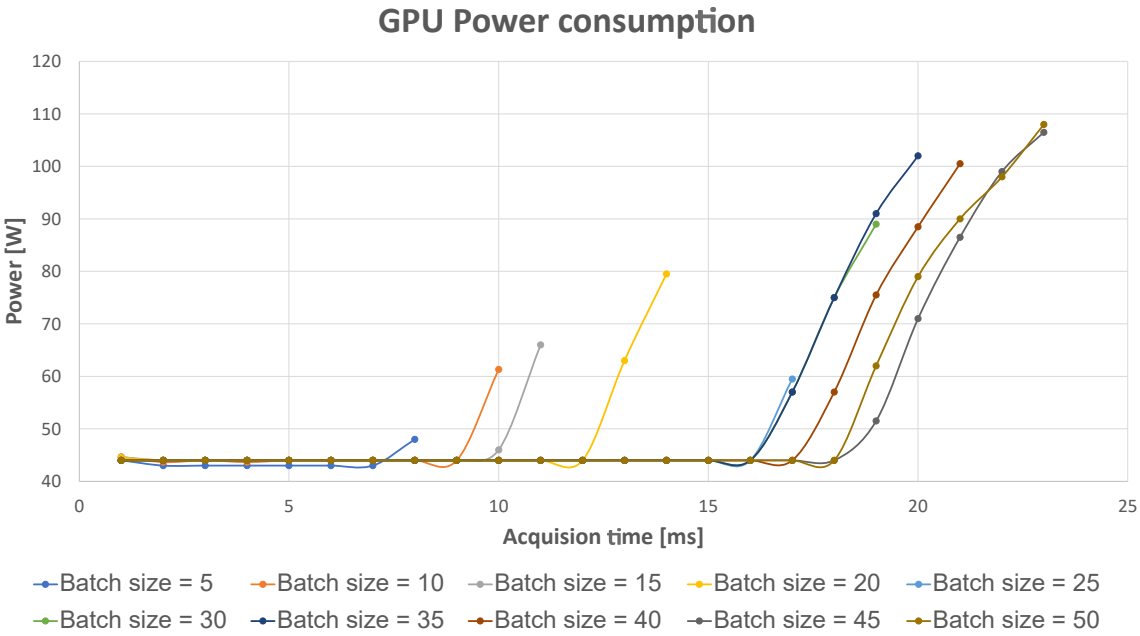


Figure 3.20: GPU power consumption vs batch size.

In Figure 3.20, the GPU power consumption to vary the batch size is plotted. As expected the power consumption increases with the batch size with peaks of almost 110 W with batch size equals to 50. It has to be remarked that this power is of the algorithm only: the idle power has been subtracted and no other programs were using the GPU while the measurement was carried out.

By multiplying the power consumption for the delay, it is possible to compute the energy. Since both the power and the delay increase with the batch size, the expected behaviour is an increasing one just as shown in Figure 3.21.

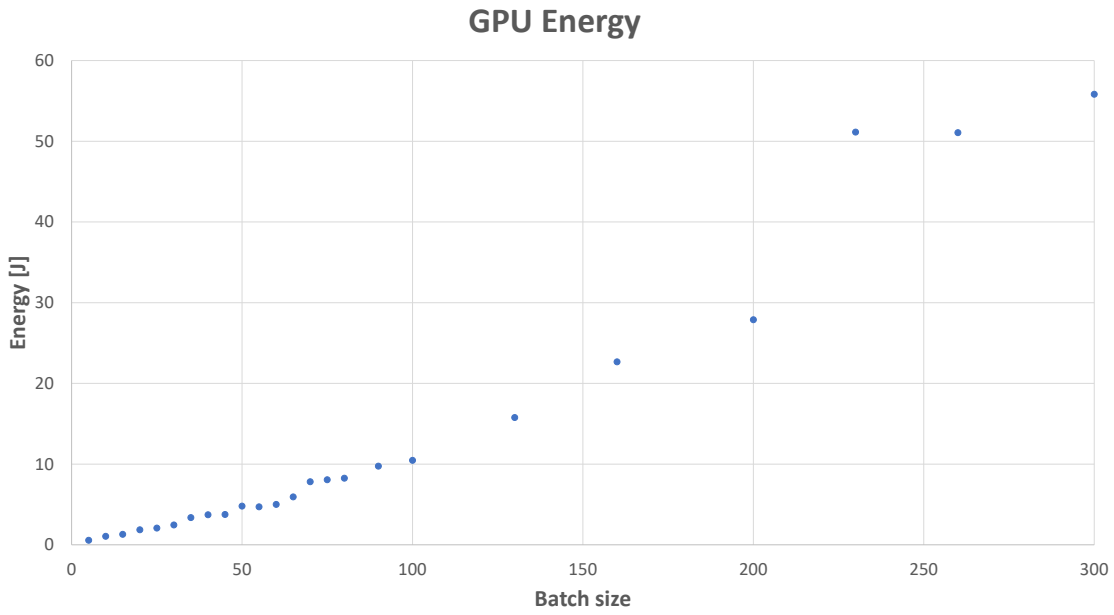


Figure 3.21: GPU energy vs batch size.

As a final comparison, this implementation is compared with another AlexNet software implementation with different GPU and CPU [23].

| **Platform** | **FPS max** | **Power [W]** | **Energy/images [J]** |
|---|---|---|---|
| Intel Xeon E5-2673 | 5 | 130 | 25.400 |
| Intel i7 930 (this implementation) | 30 | - | - |
| GTX Titan X | 769 | 250 | 0.325 |
| Tesla K40c (this implementation) | 680.81 | 148 | 0.180 |

Table 3.3: Comparison with other AlexNet implementations.

In Table 3.3 it is possible to see how this AlexNet implementation on a Tesla K40c is much more power-efficient: it consumes 148 W which is less than the Titan X implementation. However, it has to be noticed that this implementation has almost 100 FPS less than the Titan X one but it is more energy-efficient since the energy/image is almost twice less.

For what concerns the CPUs comparison, unfortunately only the FPS can be compared because there are no power data on this AlexNet implementation on a CPU.

The FPS of the Intel i7 930 are higher than the Xeon one and it is possible to estimate that the power consumption is higher too.

In conclusion, it is straightforward to see that a GPU implementation is better than a CPU one in everything: the FPS are higher, the power consumption is lower and it is more energy-efficient.

# Chapter 4

# Hardware implementation

Designing a reconfigurable HW accelerator based on logic-in-memory for neural networks requires an analysis of the most used neural networks to understand which are the layers to implement, all the parameters that change among all the algorithms and how many bits are needed to guarantee a satisfying accuracy.

Neural networks such as AlexNet, ZFNet, LeNet-5, GoogLeNet, VGGNet, DenseNet and ResNet have been analyzed. The idea is to find common characteristics among all the algorithms in order to design an HW accelerator as much optimized as possible for almost every network.

All the implemented layers can be executed separately which means that if a not implemented layer has to be executed at a certain point in the network, it is possible to execute the implemented layers on the designed accelerator and the not implemented ones outside (for example with a microprocessor).

The main constraint on this project is the power consumption. For this reason, all the design choices are made to keep as low as possible the power consumption.

The layers which have been implemented are: convolutional, batch normalization, fully-connected and max-pooling while as activation function, the ReLU has been chosen among all the possible ones.

## 4.1 Convolutional layer

The convolutional layer is probably the most used in every convolutional neural network. The Amdahl's law says that in an elaboration system it is important to optimize the most frequent case in order to obtain an overall increase in performance which is equal to:

$$Speed - up = \frac{1}{(1 - P + \frac{P}{S})}$$

where P stands for how many times the frequent operation is executed with respect to all the others and S indicates the improvement in terms of speed-up due to the optimizations done to the most frequent case.

As an example, if the convolution is executed just the 30% of the time and if one is able to improve the execution time of that operation by a factor of 2, the overall speed-up is:

$$Speed - up = \frac{1}{(1 - P + \frac{P}{S})} = \frac{1}{(1 - 0.3 + \frac{0.3}{2})} = 1.17$$

which is clearly not good. However, if the convolution is executed 70% of the time with the same value for S, the overall speed-up is:

$$Speed - up = \frac{1}{(1 - P + \frac{P}{S})} = \frac{1}{(1 - 0.7 + \frac{0.7}{2})} = 1.54$$

Which means that the new architecture is able to perform 1.5 times faster than the previous one.

By analyzing several neural networks, the following characteristics have come to light:

- Filter sizes values: 1x1, 2x2, 3x3, 5x5, 7x7, 11x11.
- Stride values: 1, 2, 4.
- Max feature maps number: 1024 (GoogLeNet).

It has to be highlighted that a 11x11 filter with a stride of 4 is used only by the AlexNet algorithm.

However, the ZFNet algorithm is basically a better evolution of the AlexNet, because it has better accuracy, with a smaller filter dimension (7x7 instead of 11x11) and a smaller stride value (2 instead of 4).

For these reasons the actual characteristics implemented by this accelerator are:

- Filter sizes: 1x1, 2x2, 3x3, 5x5, 7x7.
- Stride values: 1, 2.
- Max feature maps number: 1024 (GoogLeNet).

The core operation of a convolutional layer is the `MAC`. Since this accelerator will be based on logic-in-memory, it is important to think right from the beginning what types of logic will be implemented in the memory and what the surrounding logic will contain.

It is clear that inserting a multiplier directly in the memory is not convenient since it consumes a lot of power and it is slower. On top of that since a minimum of parallelization is needed it is straightforward to understand that inserting several multipliers and adders in the memory is not feasible.

Fortunately, a multiplier can be implemented in another way using the Modified Baugh-Wooley method.

### 4.1.1 Modified Baugh-Wooley

The Baugh-Wooley method consists in substituting a multiplier with basic logic gates, such as AND and NAND gates, and a number of adders which is equal to the highest number of bits between the two operands.

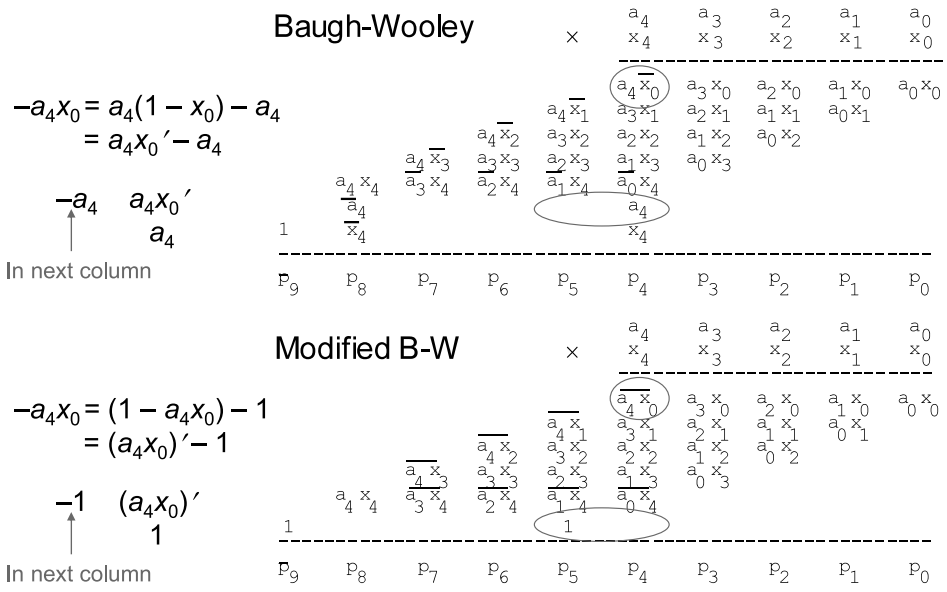


Figure 4.1: Baugh-Wooley method and its modified form.

Between the two methods shown in Figure 4.1, the Modified Baugh-Wooley has been chosen due to the lower number of adders. It is important to highlight that all the adders but the last one are no-signed adders which means that the sign extension

has to be done with zeros. The last adder instead has to be signed because in the last case a negative number has to be added.

### 4.1.2 LiM array

The logic-in-memory array is based on 9 processing elements with 3 additional adders and some registers. The design idea is to implement the possibility of executing up to a 7x7 filter with as low clock cycles as possible. However, it should be noticed that a 7x7 filter is executed just from the ZFNet algorithm and just once. For this reason, the LiM array has been designed to execute up to 5x5 filter as fast as it can with the possibility of executing a 7x7 one but with more clock cycles. All the design choices have been made to save power where possible without affecting too much the performances considering all the possible algorithms which can be executed on this accelerator.

In Figure 4.2 it is possible to appreciate the overall structure of the LiM array.

This type of structure is versatile because it is possible to decide how many blocks of this LiM array structure can be actually implemented in the memory. As an example, one can decide to implement the whole structure or just the processing elements.

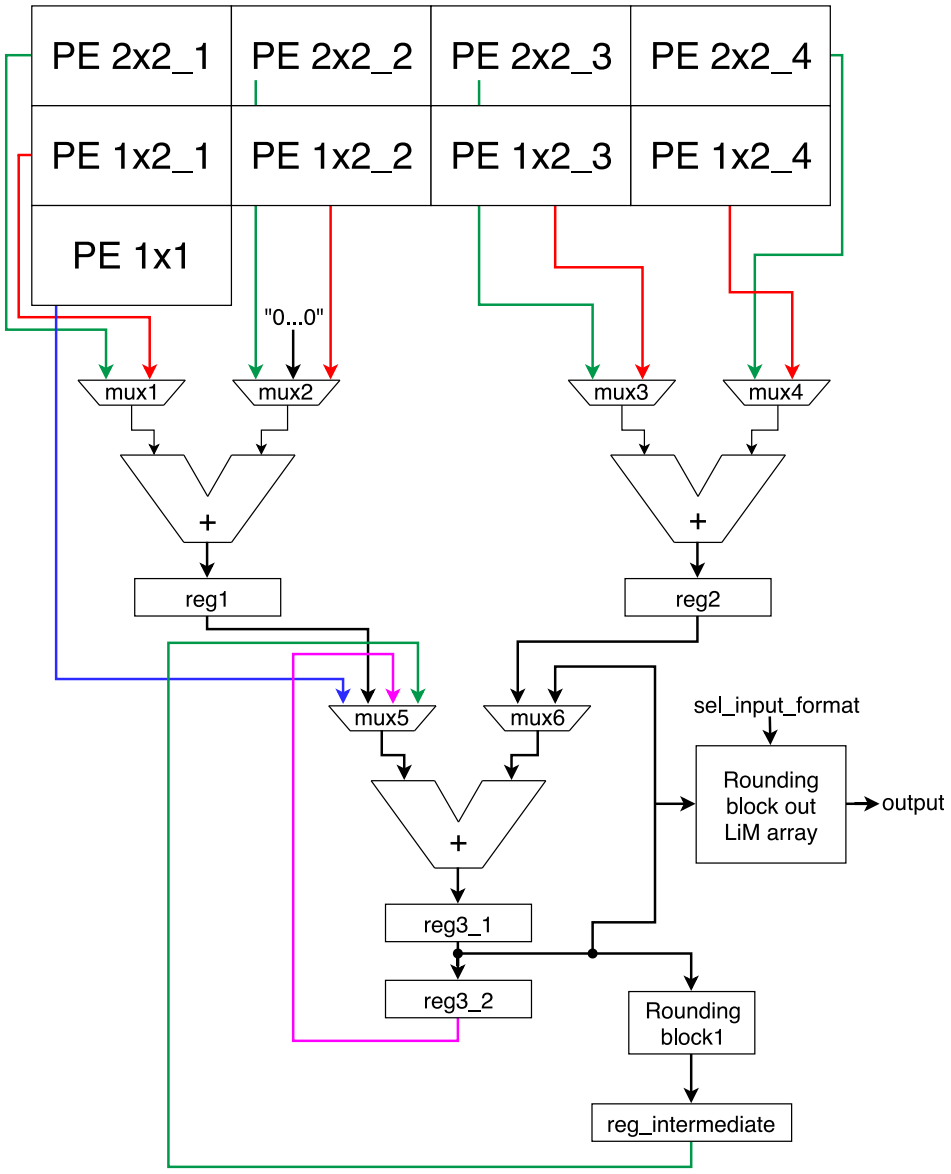


Figure 4.2: LiM array.

In the following, each block will be analyzed except for the rounding blocks which will be discussed in the section 4.6.

### Processing element 1x1

All the PEs are based on the Modified Baugh-Wooley method. In particular, the PE 1x1 is made of a block called filter and adders which are needed to compute the

final result.

It has to be highlighted that each PE can perform up to 4 MACs operations. In the specific case of a PE 1x1, only one multiplication is needed. As an example, a 4 bits PE 1x1 will be analyzed. The filter computes all the partial products which will be then added together. In Figure 4.3 it is possible to see all the logic gates needed to compute all 4 partial products.

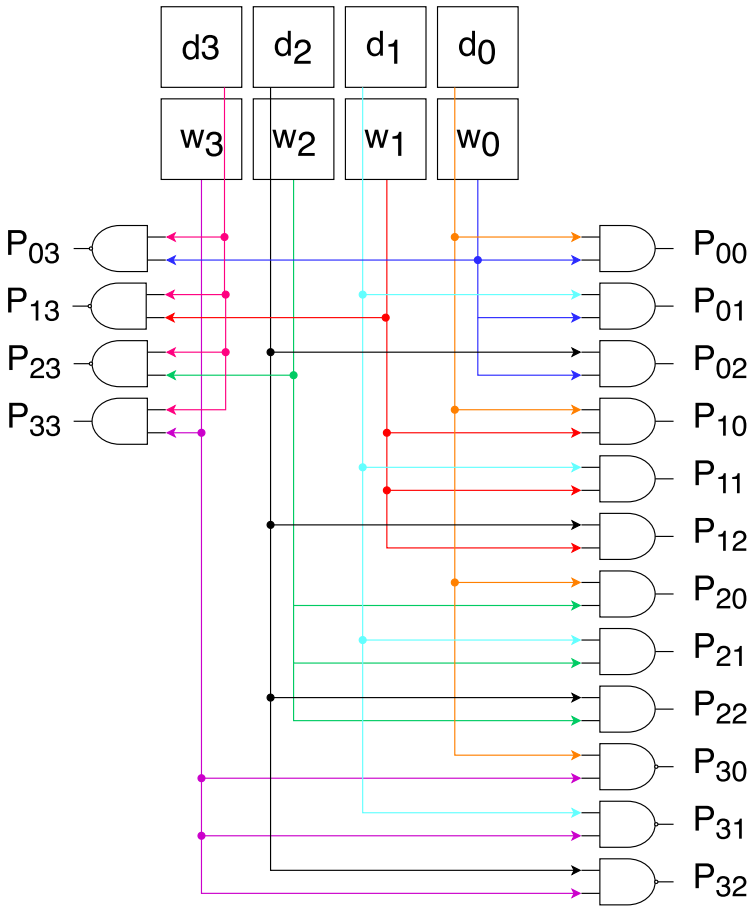


Figure 4.3: Basic cell 1x1 filter.

D3, d2, d1 and d0 represent each bit of the input datum while w3, w2, w1, w0 each bit of the input weight. For what concerns the partial products, the notation used is the following one:

$$P_{i,j}$$

where i represents the partial product number, while j represents the bit number of

each partial product.

All the partial product $P_i$ have to be added as shown in Figure 4.4. It has to be highlighted that each partial product is left-shifted of a number of position which is equal to the `i` index and then added together with all the others $P_i$.

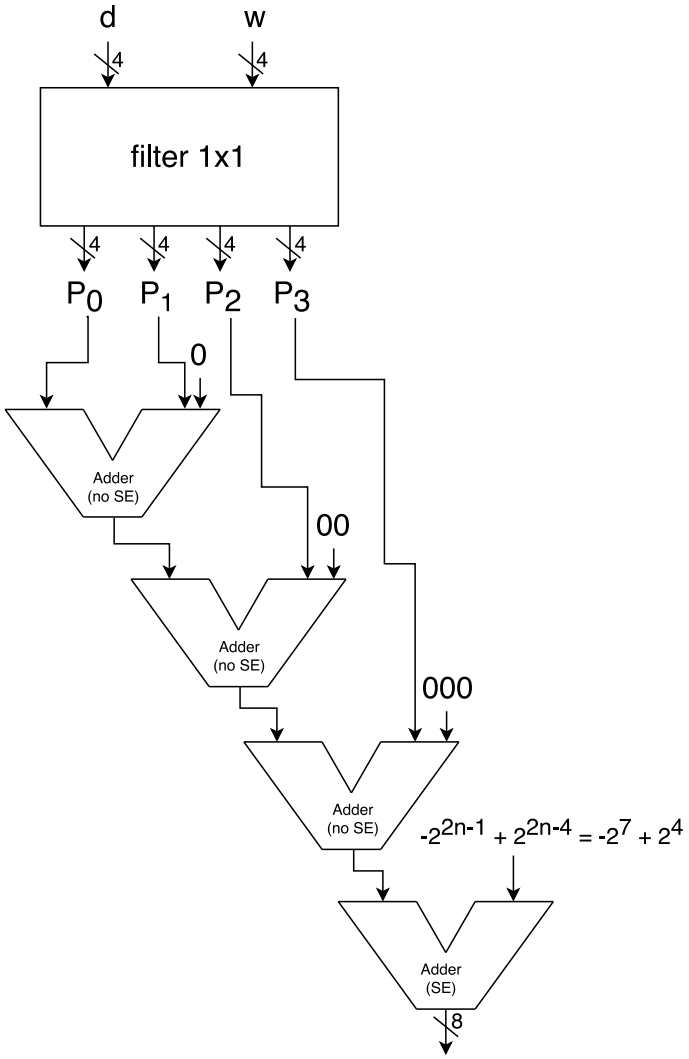


Figure 4.4: PE 1x1.

As mentioned in the subsection 4.1.1 all the adders but the last one are no-signed adders which means that the sign extension has to be done with zeros.

### Processing element 1x2

The PE 1x2 is very similar to the PE 1x1, in fact it is implemented as two 1x1 filters. In Figure 4.5 it is possible to see how each filter generates 4 partial products where each one is a 4 bits value.

Also in this case, an example with just 4 bits is used to make more understandable the overall structure.

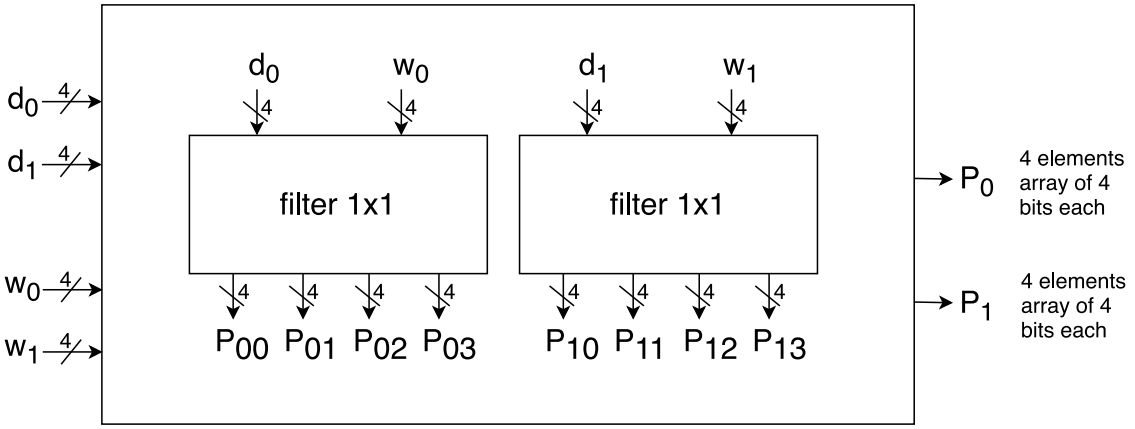


Figure 4.5: filter 1x2.

The convention used in Figure 4.5 is slightly different from the one used in the previous case. As a matter of fact, each partial product is denoted with:

$$P_{i,j}$$

where i represents the number of filters which has computed the partial product, while j represents the partial product number.

In this specific case, there are two 1x1 filters which means that all the $P_{0,j}$ represent the partial products generated by the first filter, while all the $P_{1,j}$ represent the ones generated by the second 1x1 filter.

It has to be noticed that this PE 1x2 implementation, is not based just on the replica of the PE 1x1, because if it were so, at the end it would have one additional adder to take into account the negative number which has to be added. Since the negative number to be added to complete each multiplication is known a priori, it is possible to join together two multiplications by adding properly all the partial

products and at the end it is sufficient to add one negative number which is two times the negative number to be added in case of a single multiplication.

This improvement allows saving one adder which is an important contribution in terms of both delay and power since theoretically this structure can be used with more than 4 bits.

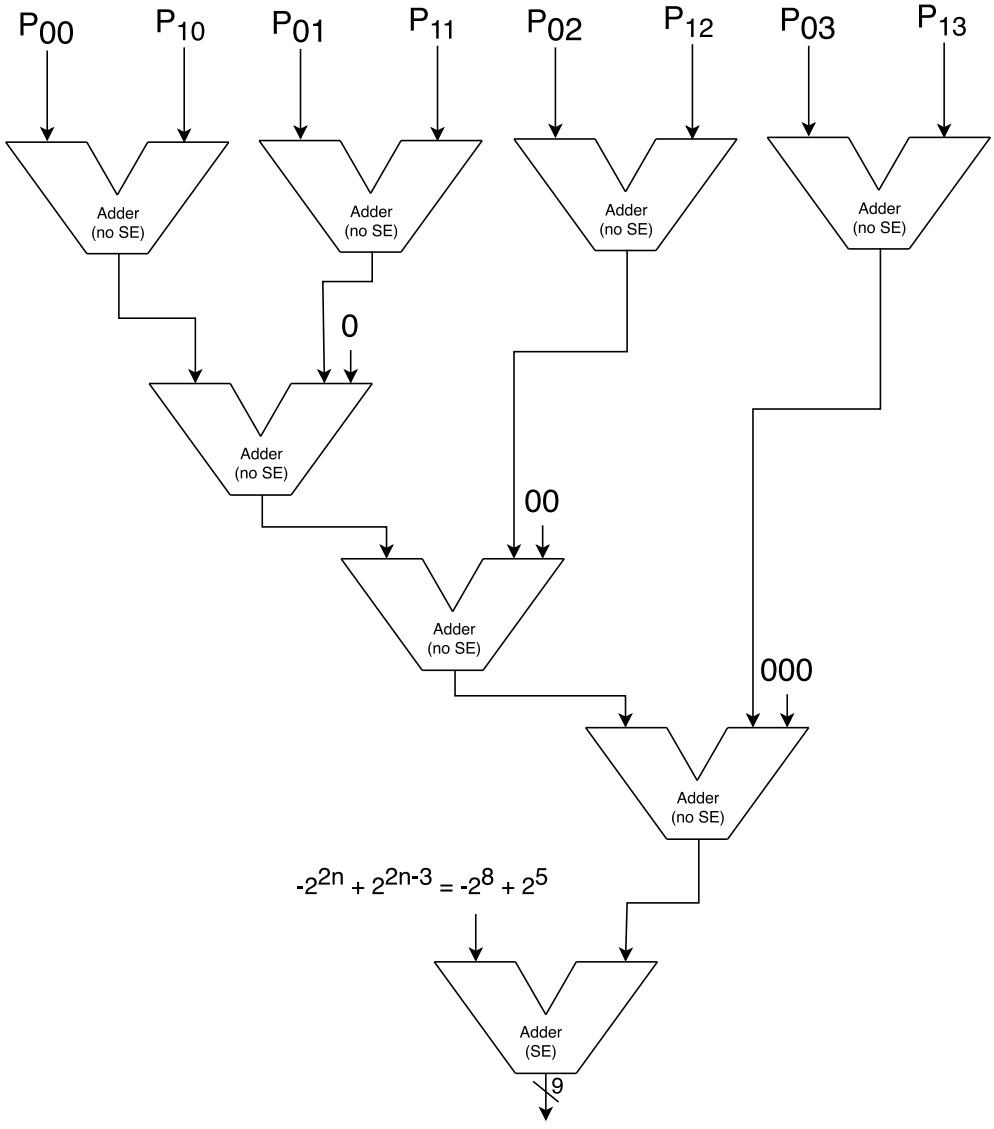


Figure 4.6: PE 1x2.

### Processing element 2x2

The PE 2x2 is based on two PE 1x2: in this case no further optimizations can be done and for this reason it is sufficient to replicate the structure of a PE 1x2 and adding a final adder to sum the two outputs coming from each PE 1x2.

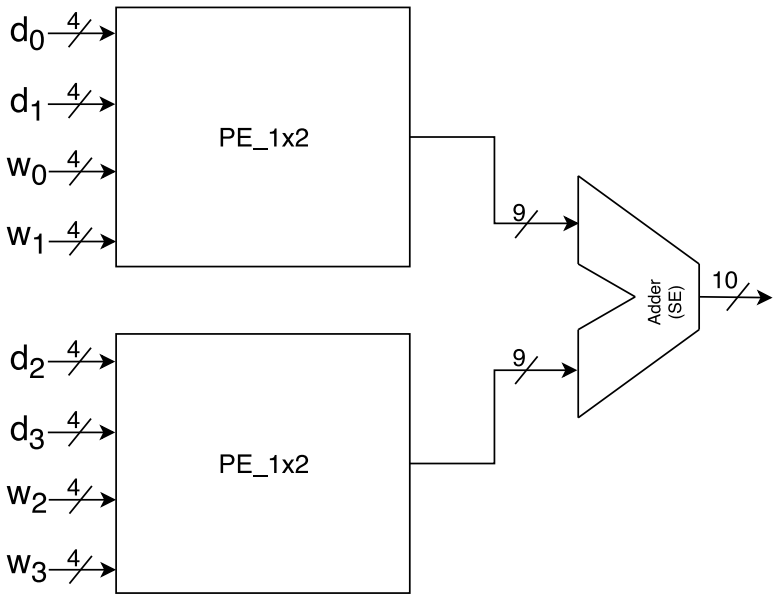


Figure 4.7: PE 2x2.

In Figure 4.7 it is possible to see 8 inputs: 4 are data inputs and 4 are weights input. Also in this example, each input has 4 bits.

It is worth highlighting that final adder is a `SIGNED` adder.

### 4.1.3 Convolutional surrounding logic

The surrounding logic has to provide the input to the LiM array and it has to handle the output of each convolutional layer. It has to be noticed that the surrounding logic has to handle multiple input channels and the insertion of zeros as padding.

In Figure 4.8 it is possible to see the surrounding logic of the LiM array. The input data come from a data memory which will be discussed later on. For now, it is sufficient to know that one row per time can be read by the memory and this row can go in any of the shift registers. Then the multiplexers which are at the input of each PE are properly controlled in order to use the right data input to compute the desired filter.

For what concerns the input weights, they are stored in a big register organized as a matrix with 7x7 basic cell of n-bit each. This matrix register can be controlled row by row which means that the input and the output of each row have $7 \cdot n \ bits$ where n is the number of bits of each cell. The number of rows and columns depends on the maximum dimension of the filter size.

The multiplexers in Figure 4.8 are needed also for the weights, but this time the connections come directly from the register matrix. In the end, the number of muxes and connections has to be doubled with respect to the one shown in the figure to take into account also the input weights of each PE.

The number of shift registers is 7 because this is the maximum dimension of the filter size. The width of each register file depends on the maximum dimension of the input image among all the neural network algorithms analyzed which is $229 \cdot n \ bits$ where n is the number of bits of each cell.

Practically these shift registers represent a copy of the data memory rows needed to perform the filter computation. From now on the shift registers will be considered as 229 cells of n bits each to simplify the dissertation.

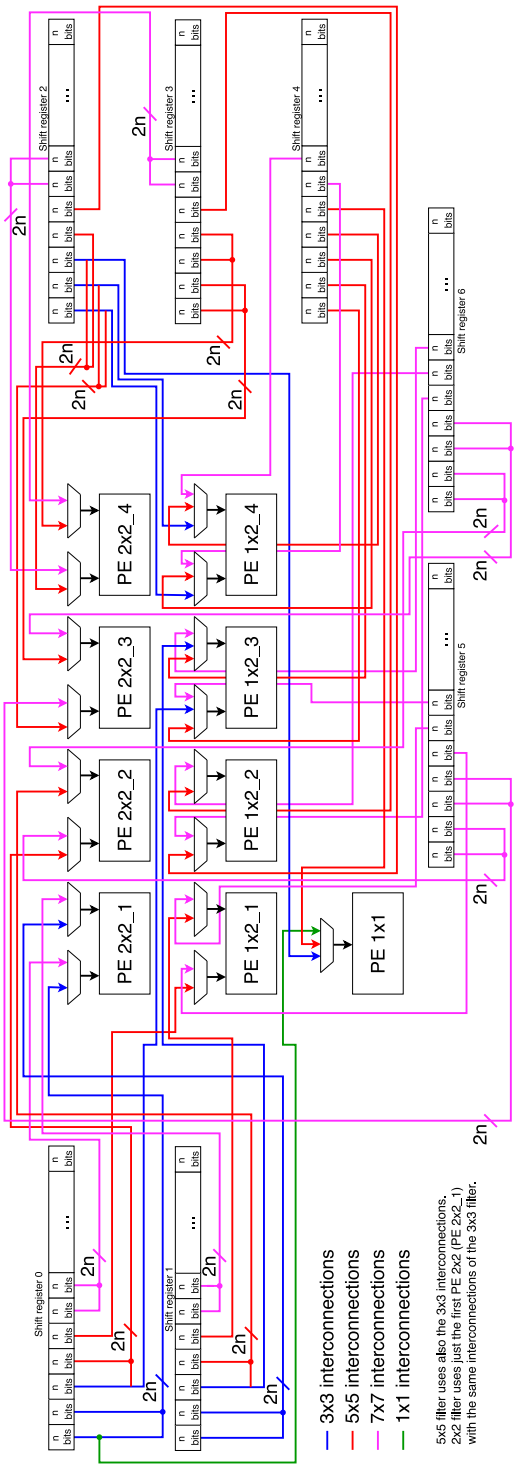


Figure 4.8: LiM array surrounding logic.

The idea behind the shift registers is to replicate the movement of the filter throughout the input matrix. If the filter moves from left to right of a number of position which is equal to the stride value until the end of the input matrix, the shift registers perform a left shift of stride positions. In this way, it is possible to connect just the first 7 cells of each shift registers simplifying also the connections to the input multiplexer of each PE.

In Figure 4.9, Figure 4.10 and Figure 4.11 it is possible to see the principle behind the shift register implementation.

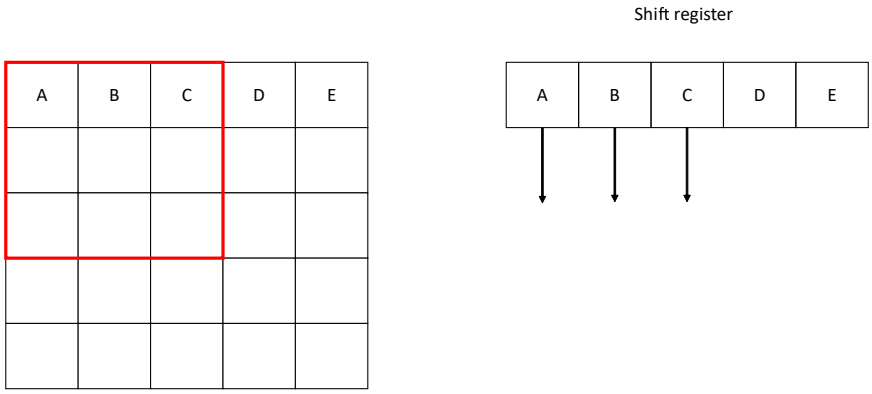


Figure 4.9: Convolutional layer step 1.

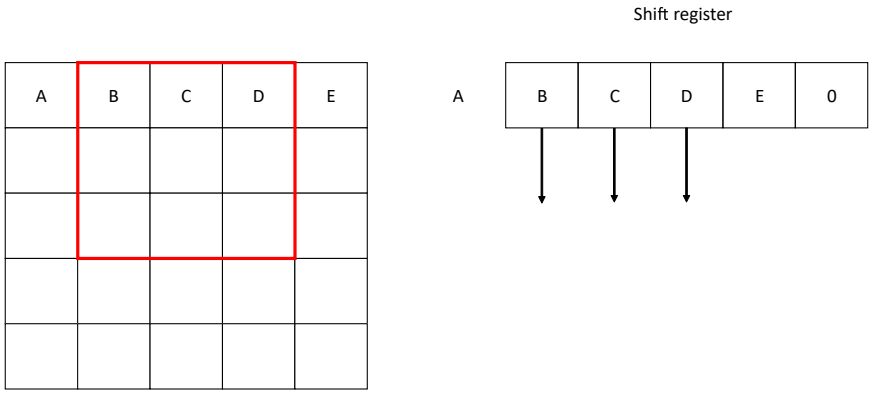


Figure 4.10: Convolutional layer step 2.

With the HW shown in Figure 4.8, the convolution of a single input channel can be performed. However a convolution might have multiple input channels, a bias to be added and zeros to be inserted as padding.

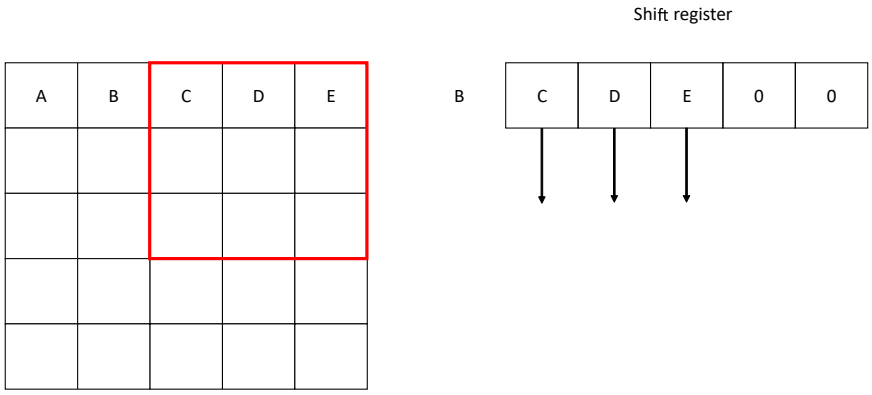


Figure 4.11: Convolutional layer step 3.

To be able to perform these additional operations, further hardware has to be added. In Figure 4.13 it is possible to see that two more registers, as well as an adder and additional muxes, have been added. In particular:

- Reg0_SL: in this register is loaded first the bias and then each partial result of the output of a convolutional layer. Considering an output channel, each element is obtained as a set of filters applied to all the input channels and then added together with the bias. For this reason, if this process is serialized, it is possible to use the Reg0_SL to store the bias values and all the partial result of each output channel. The right input of this register is selected by a multiplexer which selects between the bias value, contained in a register and then replicated 229 times, and a row read from the data memory.

- Reg1_SL: the partial results of each output channel are stored in this register. The convolutional logic described before is able to compute one datum per time which means that this register is divided into 229 cells of n bits each where each cell can be controlled separately. The convolutional logic computes a value which is stored in the appropriate cell of this register. Then this cell is added with the equivalent Reg0_SL cell through the additional adder and the output value is stored again in the same cell of Reg1_SL register.

- 2 muxes 229 to 1: these 2 muxes are used to select the two cells from Reg0_SL and Reg1_SL to be added.

- Adder: final adder which is used to sum together the values from Reg0_SL and Reg1_SL.

It is important to highlight that there are two muxes which select the right Reg1_SL input: the first one select among the convolutional logic output, the adder output and the max-pooling output which will be discussed later on. For what concerns the second mux, it is used to perform the ReLU activation function.

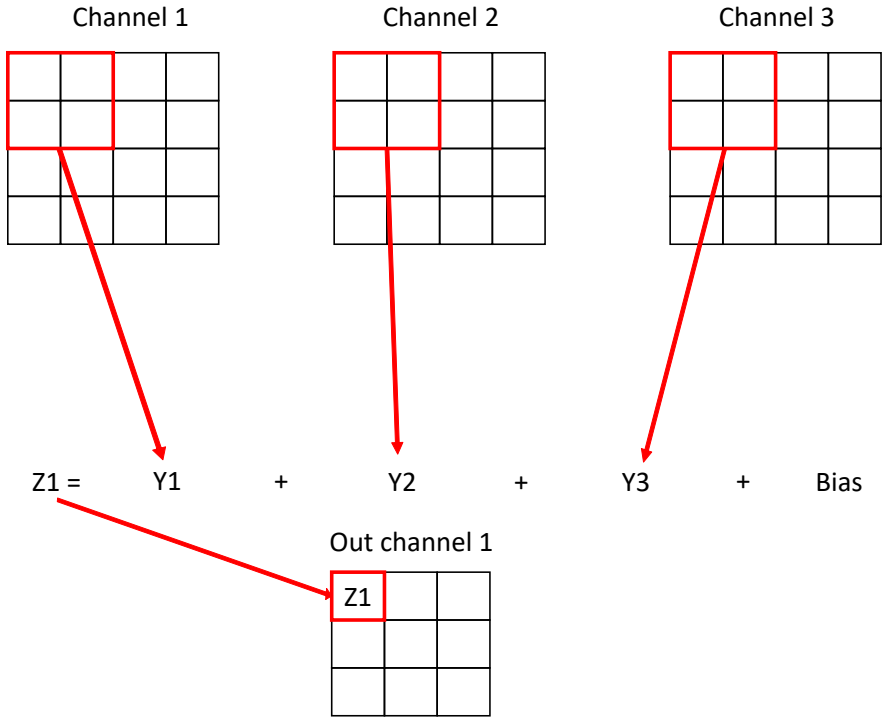


Figure 4.12: Convolutional layer with multiple input channels.

In Figure 4.12 it is possible to see how a convolutional layer with 3 input channels works: each input channel is convolved with its filter and then all the partial outputs are added together with the bias to generate an element for each output channel.

If the example in Figure 4.12 has to be executed with this HW accelerator the following operations would be performed:

- The bias value is stored in the Reg0_SL and the proper data and weights are read from the memory and then through the matrix register and the shift registers they are stored in the right PEs.

- $Y1$ is computed from the convolutional logic and then stored into the first cell of the Reg1_SL.
- $Y1$ is added with the bias stored in the Reg0_SL and the result is loaded in the same cell of the Reg1_SL.
- Once the entire row of the output channel is computed and stored in the Reg1_SL register, it is written in the data memory.
- By following all the previous points the whole output channel is computed and stored into the data memory.
- The input channel 2 is loaded into the memory and this time instead of loading the bias value in the Reg0_SL register, the first row computed in the previous steps and stored in the data memory is loaded.
- $Y2$ is computed and stored into the first cell of the Reg1_SL.
- $Y2$ is added with the first cell of Reg0_SL which contains $Y1 + bias$ and the result is stored again in the Reg1_SL register.
- Once again the whole output channel is computed and stored in the data memory. Each element is of the type $Y1 + Y2 + bias$.
- All these steps are repeated in order to add the results coming from the input channel 3.
- At the end, the data memory will contain the final output channel values.

From this brief explanation, it is clear that a data memory with 2 banks is needed: one bank is used to store the input channels and the other is used to store the output channels.

Now there is only one operation left to cover: the zero-padding. Zero padding means adding zeros in one or more rows and one or more columns. Fortunately implementing this operation is quite easy: if rows of zeros have to be written, it is sufficient to reset the Reg1_SL register and write it into the data memory. When columns of zeros have to be written instead, the Reg1_SL register can still be used: before starting the computation of a new row, this register is always reset. This

means that it already contains all zeros. To take into account the zero-padding for the columns, it is sufficient to write the first output value from the convolutional logic in the position $0 + padding$ in the Reg1_SL register. Thus the generic output $i$ will be written in the position $i + padding$. Once all the output channels values for one row have been written in the Reg1_SL register, the zero-padding in the columns at the right side of the computed data are already there thanks to the initial reset. Hence, at this point it is sufficient to write the Reg1_SL content in the data memory and proceed with the remaining computations.

It is worth highlighting that when performing the zero-padding of the columns, the Reg1_SL register cells which contain the padding values are not added with the Reg0_SL content in order to avoid the addition with the bias value stored at the first cycle in the Reg0_SL register.

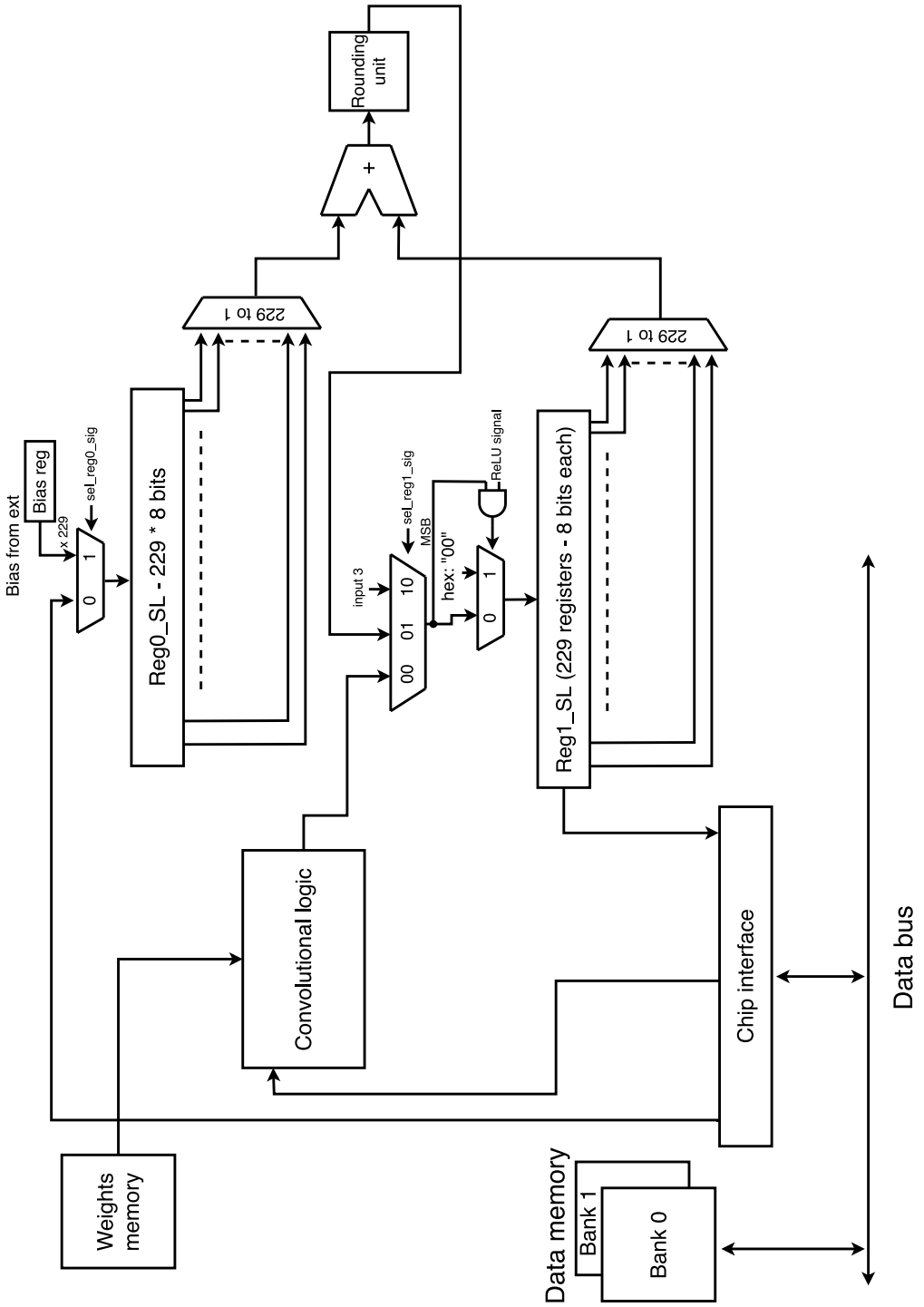


Figure 4.13: Convolutional block surrounding logic.

## 4.2 ReLU activation function

The ReLU activation function is very simple to be implemented in hardware. It is sufficient to check if the MSB of the datum is 1 (which means that it is a negative value) and then select the proper input to the Reg1_SL.

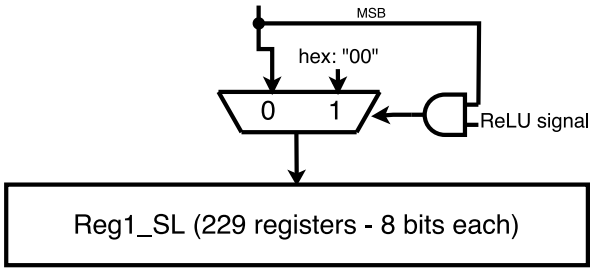


Figure 4.14: Hardware which performs the ReLU activation function.

In Figure 4.14 it is possible to see how the ReLU function is implemented: a 2 inputs multiplexer is used to select the right input of the Reg1_SL. However the ReLU function is not always used as activation function and in general, not all the layers use it. For example, the last fully-connected layer does not use any activation function.

For this reason, the ReLU function is applied only if the ReLU signal is equal to 1. This signal is generated by the control unit because it has to be asserted only if two conditions are satisfied:

- A ReLU external signal which must be active for the entire duration of the layer operations has to be asserted. This signal indicates whether or not the ReLU function has to be applied to that specific layer.
- The computation of the final input channel is being performed.

The problem is that in the Reg1_SL the partial result of a convolved channel per time is stored. This means that the ReLU signal can not be always asserted otherwise the activation function would be applied on each partial result and not only on the final ones as it is supposed to.

To solve this problem the control unit checks the input channel which is being processed and only if it is the last one the ReLU signal is asserted (if the activation

function has to be applied). The AND gate is used because the value to store is zero only if the ReLU function has to be applied and the input values are negative.

## 4.3 Fully-Connected layer

The first fully-connected (`FC`) layer is usually preceded by a flatten operation in order to transform the input structure which is a 3D matrix into a vector with only one dimension. In Figure 4.15 it is possible to see an example of a flattening operation.

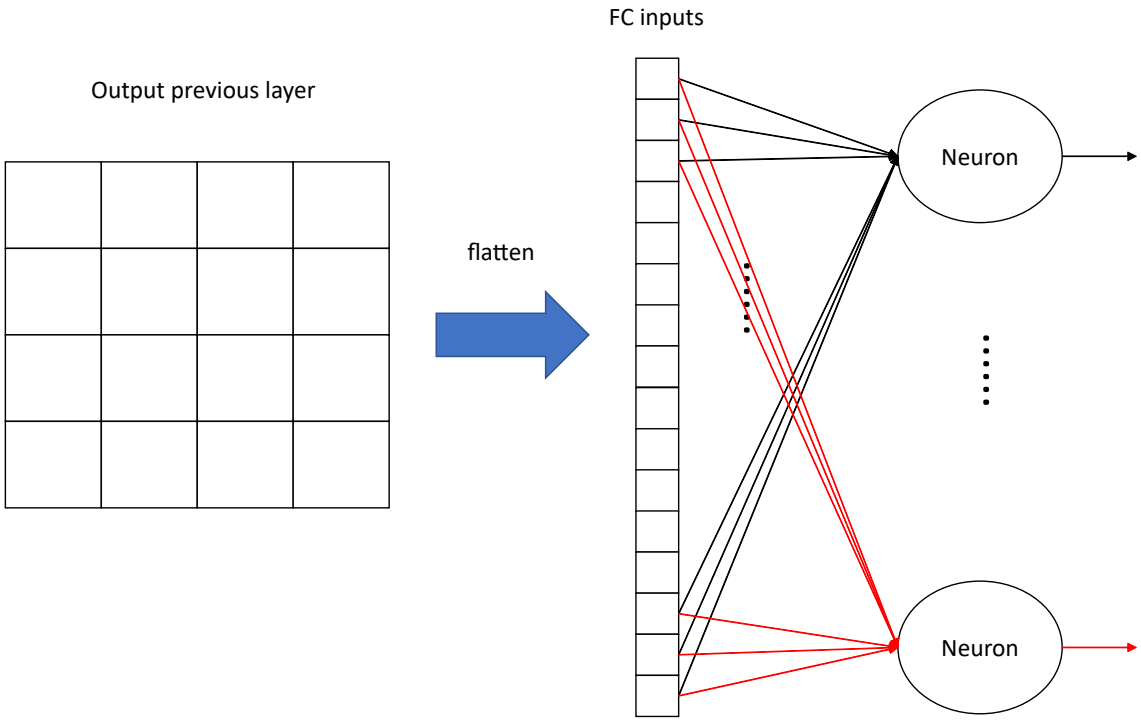


Figure 4.15: Flatten operation.

However, the flatten operation does not change the input data in any way. It is just a compression of all the input dimensions into a single one. This means that if the FC weights, instead of being ordered into a single vector, match the input data order before applying the flatten operation, it would be possible to delete the flatten operation and execute the FC layers as they were convolutional layers with weights that always change. Another important difference is that all the input values are multiplied for their equivalent weights and then they are added together to retrieve

just one single output. To obtain more outputs it is sufficient to use another set of weights. As a matter of fact, each output value of a FC layer represents a distinct output channel.

Theoretically, it is possible to implement the operations required by a FC layer with the same HW used for the Convolution. In particular, if the reg_intermediate register (Figure 4.2) is used as an accumulator, it is possible to compute the multiplication of each input value for its equivalent weight and store the partial sum into this register. For this reason, there is a rounding unit at the input of this register. If it were only for the convolution the rounding unit would not be necessary because the internal parallelism is such that the worst-case can be handled without any losses in precision. On the other hand, the FC layers can potentially do tons of additions which means that fully precision is not an option.

Assuming to use the same HW, there are still some issues to be solved. First of all the convolutional HW has been designed to compute the result of a particular filter applied to the input matrix. The main problem is that FC layers do not have filters because all the inputs are connected to all the neurons. It is clear that this can be seen as a unique giant filter but it is not feasible.

To solve this problem, one idea is to see the FC layers as a certain filter applied to all the inputs with different weights. For example, a filter 4x4 can be used in order to use as much PEs as possible. Differently from a convolution, the filters can not be overlapped in any case while covering the whole input matrix. For this reason, if a 4x4 filter is chosen then a stride of 4 has to be implemented as possible shift value for the shift registers.

From a graphical point of view an FC layer would be implemented as in Figure 4.16 and Figure 4.17 with a 4x4 filter with a stride of 4.


Figure 4.16: 4x4 filter used for FC layers - Part 1.

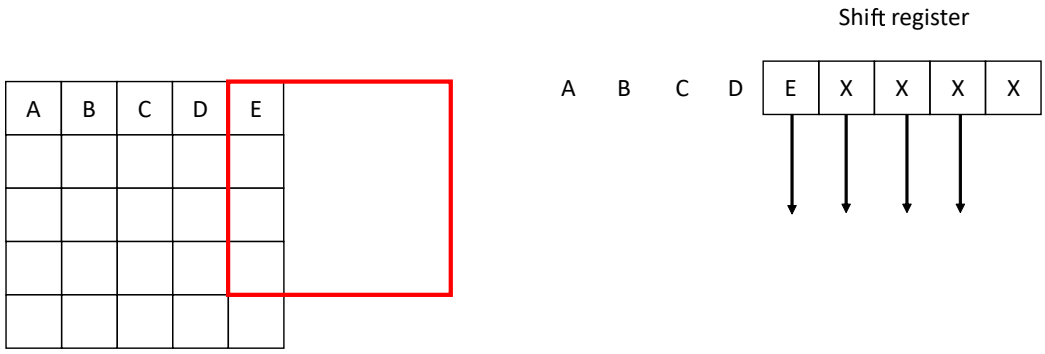


Figure 4.17: 4x4 filter used for FC layers - Part 2.

When shifting of 4 positions the filter might go outside the useful data of the input matrix, as shown in Figure 4.17. In this case, the wrong data could be used to compute the final result. However, in this HW implementation, the data memory is written row by row with the useful data and when the number of data is not enough to cover the entire row width, zeros are automatically inserted. With this little trick, the problem of using the wrong data is automatically solved.

However, when a filter 4x4 is used it is important to check if the input rows and the input columns are divisible by 4 because if they are not the computation of the last filter needs to be handled differently. If the number of rows is not divisible by 4 when computing the filters of the last rows only the row or rows which contain the actual data are read from the memory and stored in the shift register or registers. The other shift registers are reset to be sure they contain zeros and not other values. In this way when they are used for the multiplication, their contribution would be zeros and the result would not be affected. Similarly, when the input columns are

not divisible by 4 an additional, the last filter can be still computed because it would use the values shifted in the shift register. If these values are zeros their contribution would not affect either in this case the final result.

This method while more difficult from a control point of view, it is indeed the most effective one for this kind of architecture. Moreover, it is quite easy to check if the number is actually divisible by 4 by looking at the last two bits (least significant ones):

- 00: the number is divisible by 4.
- 01, 10, 11: the number is not divisible by 4. In this case, the rest could be 1,2 or 3.

It has to be noticed that it is sufficient to shift right the input number by two positions to perform the division by 4. While the dimension of the data memory is determined by the worst-case scenario in terms of input images, the weight memory must be as large as the data memory but also divisible by 4 in order to not make too complex the control unit of the FC layers.

The reason behind this constraint on the weight memory is to avoid trying to read inexistent rows or columns. In fact, if the weight memory had the same dimension of the data memory and an FC layer with an input equal to the max dimension of the memory was executed, at a certain point, the weight address generated by the Control unit would have addressed an inexistent location in the memory. This problem should never happen and while for the data memory it can not actually happen thanks to the shift registers and the check on the divisibility of the input rows and columns (as explained before), the weight memory is susceptible to this issue. There are two methods to solve this problem: the first one is to make the weight memory dimension divisible by 4; the latter is to make way more complex the control unit. Between the two solutions, the first one has been chosen.

It is important to highlight that no matter what are the values read from the additional locations of the weight memory, the final result is always right because their contribution is zero thanks to the multiplication with the values stored in the shift registers which are zeros.

## 4.4 Max-Pooling layer

The Max-Pooling layer can be implemented using blocks which compute the max value between 2 inputs. As done for the convolutional layer, it is important to analyze the most important neural networks algorithms in order to understand the hardware blocks which need to be implemented. In Table 4.1 the main parameters of the Max-Pooling layer of each neural network analyzed has been reported.

| **NN algorithm** | **Window size** | **Stride** |
|---|---|---|
| LeNet-5 | 2x2 | 2 |
| AlexNet | 3x3 | 2 |
| ZFNet | 3x3 | 2 |
| GoogLeNet | 3x3 | 2 |
| Vgg | 2x2 | 2 |
| ResNet | Average pooling | |
| DenseNet (for image net) | 3x3 | 2 |

Table 4.1: Max-Pooling parameters for different neural network (NN) algorithms.

From Table 4.1 it is possible to see that a window size of 2x2 or 3x3 can be used, while the stride is always equal to 2. Even so, there are some neural networks algorithms which use the average pooling instead of the one implemented in this HW accelerator. Nevertheless, it has been chosen to not implement the average pooling to keep as simple as possible the implemented HW which means that eventually the average pooling must be executed outside.

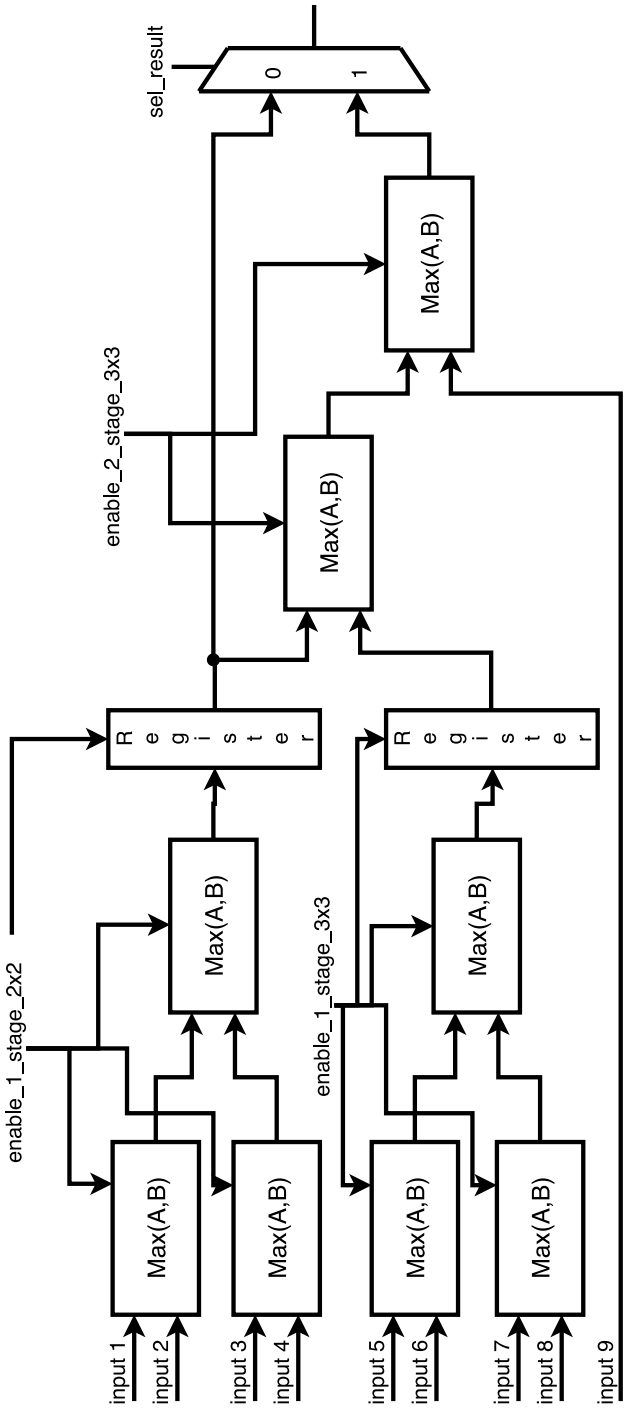


Figure 4.18: Max-Pooling HW implementation.

In Figure 4.18 the HW implementation of the Max-Pooling (`MP`) layer is shown. The maximum window size is 3x3 which means that in total the MP block has to accept up to 9 inputs. The MP operation is applied at each input channel and the windows are moved throughout each channel. For this reason, the shift registers used in the convolution layer can be re-used to store the inputs of the MP layer. Furthermore, a stride equals to 2 has already been implemented for the convolution so no changes have to be performed except for the extra connections from the shift register to the MP blocks. It has to be highlighted that only 3 shift registers are used because they contain all the 9 inputs needed in the worst case.

For what concerns the output of the MP layer, it goes in the proper cell of the Reg1_SL register.

Additional design choices have been made in order to save power where possible: if an MP operation with a window size of 2 has to be executed there is no need to keep active all the max blocks (which are combinatorial and so if the input changes they also change consuming more power). For this reason, there are the enable signals:

- Enable_1_stage_2x2: this signal is set to 1 whenever an MP operation has to be executed (the window size does not matter). When this signal goes to 1, all the max blocks controlled by this signal receive the correct inputs (otherwise they receive 0 as input in order to keep the output equal to 0 and save power when they are not used). The register enabled by this signal will then sample the output of a 2x2 MP window and will send it to the Reg1_SL through the final mux.

- Enable_1_stage_3x3: this command is used pretty much as the previous one with only one exception: it is asserted only when an MP operation with window size 3x3 has to be executed.

- Enable_2_stage_3x3: this signal is used to enable the last 2 max blocks in order to produce the right result for a 3x3 MP window and send it to the Reg1_SL through the final mux with the select signal set to 1.

With this optimizations, power can be saved when the MP layer is not executed and when an MP layer with a window size of 2 is executed. When a 3x3 MP window is

used, every block in Figure 4.18 is used so now power can be saved.

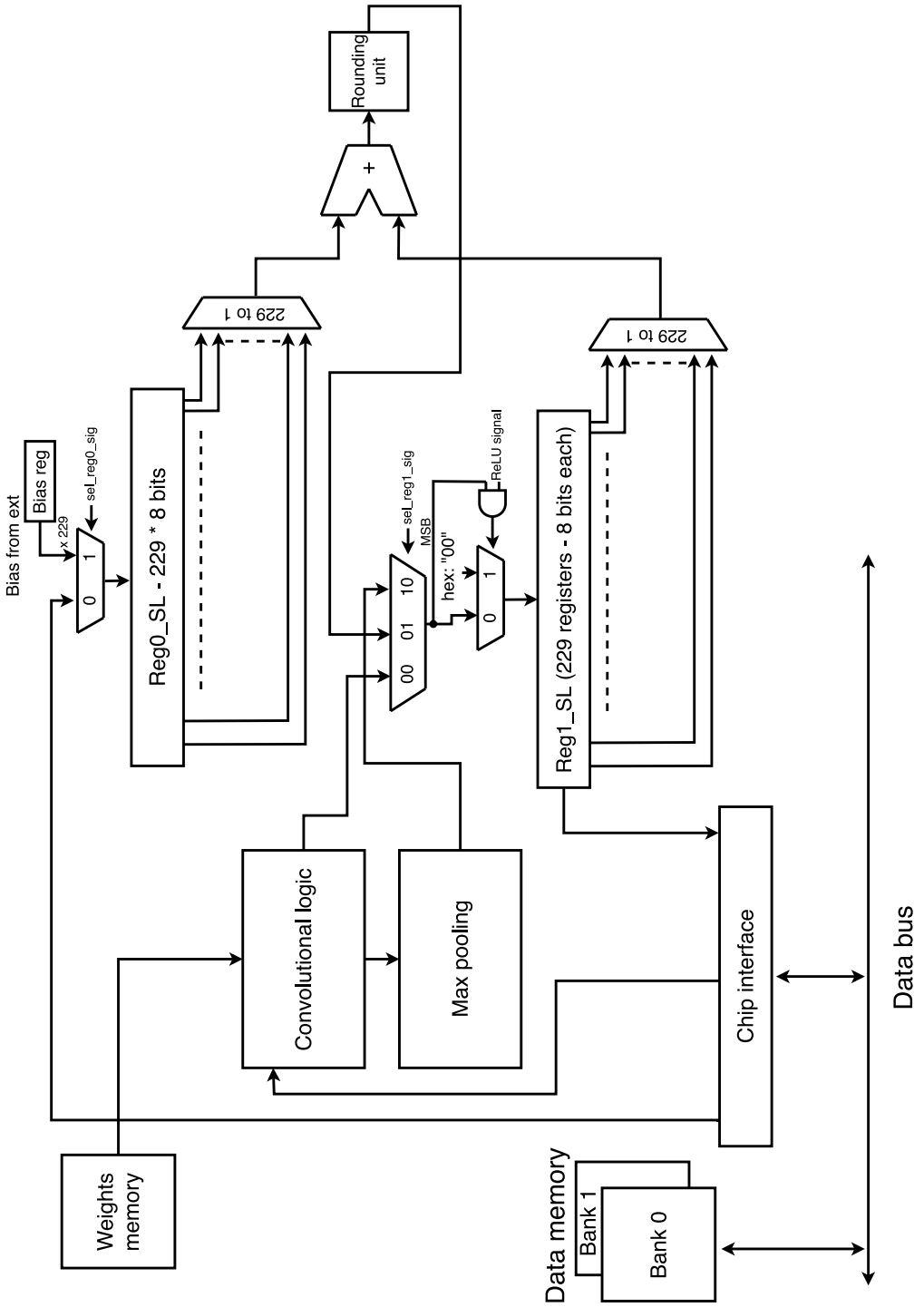


Figure 4.19: Map-pooling with convolutional logic high-level scheme.

## 4.5 Batch Normalization layer

There are several normalization layers which can be used with neural networks algorithms. Usually, they are executed after a convolutional layer but before the activation function. Most of the normalization layers are not trainable and they use division to compute the output value.

However, among all the normalization functions, there is one of them which is trainable: this normalization function is called `Batch normalization` (`BN`).

BN is very similar to a convolutional layer during inference because it is basically a multiplication of each input datum for a weight determined during training and then a bias, also determined during training, is added.

It is important to highlight that BN is applied at each value of each input channel and each normalized input channel becomes an output channel. This means that the number of output channels is equal to the number of input ones.

The BN formula is the following one:

$$y = \frac{\gamma}{\sqrt{Var(x) + \epsilon}} \cdot x + \left( \beta - \frac{\gamma}{\sqrt{Var(x) + \epsilon}} \right)$$

where $\gamma$ and $\beta$ are parameters determined with training, while $Var(x)$ and $E(x)$ are respectively the variance and the average value of all x values in a mini-batch size. A mini-batch size is a group of input images which are used during training and which are evaluated simultaneously. During inference what happens is that this formula can be seen as:

$$y = Weight * x + Bias$$

where both weight and bias values are constant throughout an entire channel but change among all the input channels. This means that each input channel has its own weight and a bias value. The main difference with respect to convolution is that there is not a summation through all the input channels to determine one single element of each output channel and that the bias value changes for each channel.

Fortunately with a little trick it is possible to execute the BN layer as it were a convolutional one: it is known that an element of a generic output channel of a convolutional layer is computed multiplying each input channel for a different filter and then add all the results of each channel convolution together with a bias value.

These filters change only if a different output channel is computed. If the output channels of the BN layer are interpreted as output channels of a convolutional one, it is sufficient to set the filter size as well as the stride to 1, in order to have that each value of each input channel is multiplied and added with a bias value. The key point here is that each set of filters which are used to compute the output channels contains only one filter which is different from zero (since it is a 1x1 filter each filter contains just one value). Which means that even if the sum through all the convolved channels is computed it will be still equal to a single input channel multiplied for the appropriate weights and added with the appropriate bias since all the other convolved channels have been multiplied by zero.

By writing down with formulas this concept, the following equation can be obtained:

$$\begin{pmatrix} F_{1,i,j} \\ F_{2,i,j} \\ ... \\ F_{C-1,i,j} \\ F_{C,i,j} \end{pmatrix} = \begin{pmatrix} W_1 & 0 & ... & 0 & 0 \\ 0 & W_2 & ... & 0 & 0 \\ ... & ... & ... & ... & ... \\ 0 & 0 & ... & W_{C-1} & 0 \\ 0 & 0 & ... & 0 & W_C \end{pmatrix} \cdot \begin{pmatrix} F_{1,i,j} \\ F_{2,i,j} \\ ... \\ F_{C-1,i,j} \\ F_{C,i,j} \end{pmatrix} + \begin{pmatrix} B_1 \\ B_2 \\ ... \\ B_{C-1} \\ B_C \end{pmatrix} \tag{4.1}$$

where $\hat{F}$ it the normalized version of a given feature map F in the CxHxW order (channel, height, width) obtained with the above matrix-vector operations for each spatial position i,j.

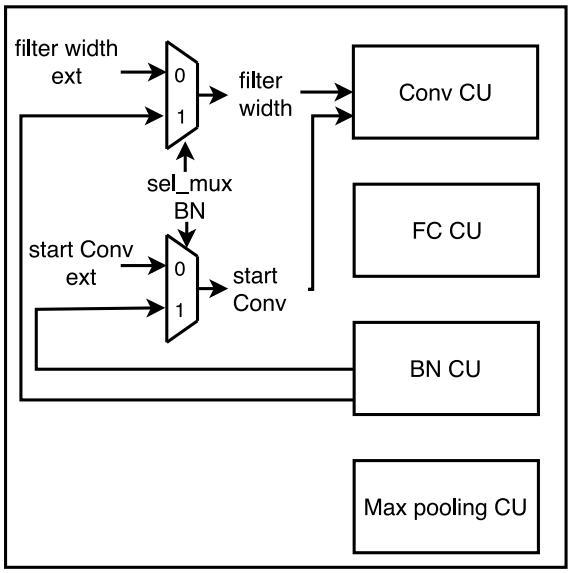


Figure 4.20: Batch Normalization.

In Figure 4.20 it is possible to see how the BN control unit basically gives the start signal to the convolutional one and set the filter size to 1. The stride is set from the external word and must be equal to 1 for this layer.

## 4.6 Rounding method

In the designed accelerator three distinct rounding units have been implemented. However, the only one which implements a proper rounding method is the one at the output of the LiM array, namely `Rounding block out LiM array`. The other two are based just on a saturator.

The Rounding block present at the output of the LiM array first apply a round half-up method to the input data in order to have the right number of fractional bit and then if the rounded value is too high or too small it is saturated to the maximum value representable. It has to be noticed that due to the nature of this block and due to the operations that it has to do, it can not be made parametric. As a matter of fact, the only way to round a value in such a way it has only the right number of integer and fractional bits is to know the fixed point format. For this reason, this unit can not be parametric as the other.

For what concerns the other two rounding units, they are parametric since in this

case, they are going to act only on the MSB of the input value. This different approach works because the values stored in the reg_intermediate register (Figure 4.2) are then used for other additions with numbers with the same fixed-point format. Moreover, additions do not increase the number of fractional bits but only the one of integer bits. This is true not only for the reg_intermediate register but also for the reg1_SL register (Figure 4.13) which is where the other rounding unit is placed. The idea for these two rounding units is to see if the value is too high or too small and then based on the result the input value will be saturated or the MSB will be just removed without any precision loss.

## 4.7 Data and weight memory

The memory system is shown in Figure 4.21. The data memory is made of two equal banks of size $229 \cdot 229$ cells of n bits each organized as a 2D matrix. It is worth highlighting that every parameter in this HW accelerator can be changed including the number of bits used for each cell and for representing data, the dimension of data and weights memories and many others. This because the architecture has been designed to be as generic as possible in order to handle every possible neural network with different input dimensions too. However, all the configurable parameters have been set to match the worst-case scenario among all the analyzed neural networks.

The data memory, as well as the weight one, is synchronous only for writing operation.

The data memory has a single port which acts as data input and data output port in order to decrease the number of pins required. Obviously, there is an address input as well as write and read signals. The single-port allows to read or write an entire row of the data memory which means that it has a parallelism of $229 \cdot n$ bits.

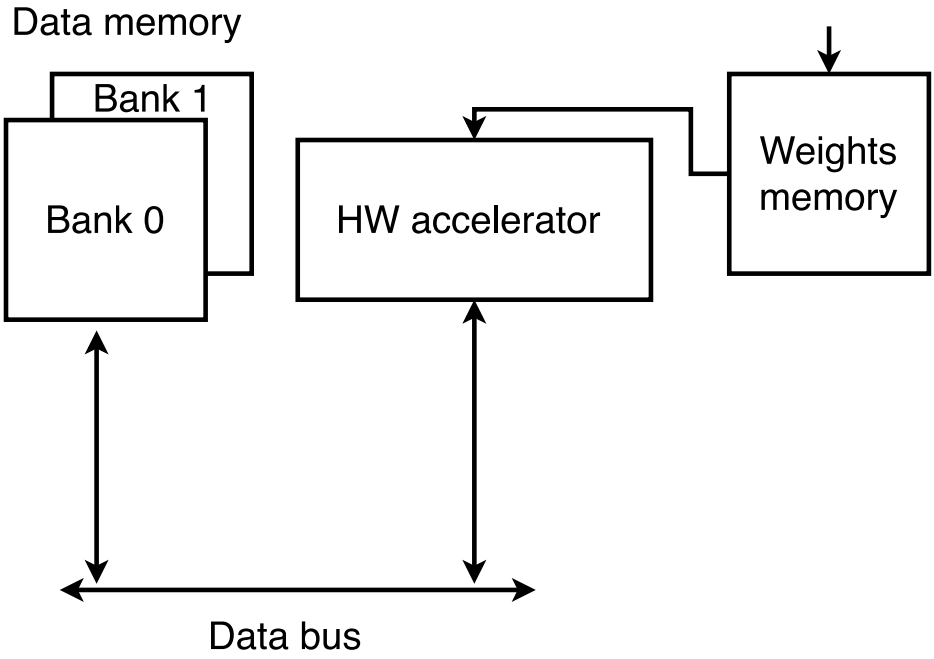


Figure 4.21: Memory system.

It has to be noticed that since there is one port which is shared for both input and output data a tri-state buffer has to be implemented in order to avoid conflicts on the data bus. The tri-state buffer implemented is shown in Figure 4.22.

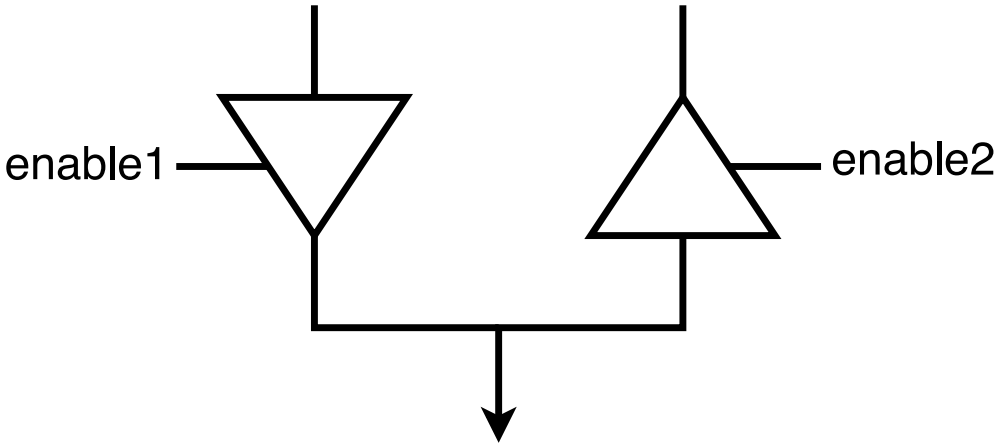


Figure 4.22: Tri-state buffer.

A tri-state buffer has to be implemented for each unit which is connected to the data-bus to avoid conflicts.

The weight memory is slightly different: first of all, there are two distinct ports, one for input data and one for output data. This memory has different parallelisms for reading and writing operations: when a writing operation has to be carried out, only one datum per time can be written in the memory. On the other hand, when a reading operation is executed, 7 data per time are red simultaneously.

The weight memory is organized as 2D matrix too, of size $232 \cdot 232$ because it has to be divisible by 4 and it has to be greater or equal to the dimension of the data memory. Moreover, the weight memory contains only one bank. The addresses for both writing and reading operations are divided into row and column. In the worst-case scenario for a convolutional layer, a 7x7 filter has to be used while for FC layers a 4x4 filter is always used with a stride of 4.

The main difference is that for a convolutional filter the weights are read just once for each input channel while FC layers need to read the weights for each computation. An assumption is made in this case: the convolutional weights are always stored in the upper left of the matrix.

However, for each computation of an FC layer, 4x4 weights have to be read from the memory. When another operation has to be performed, other 4x4 weights have to be read, but this time, only the columns change because the rows are always the same (at least until the end of the row is reached). Therefore for reading operations

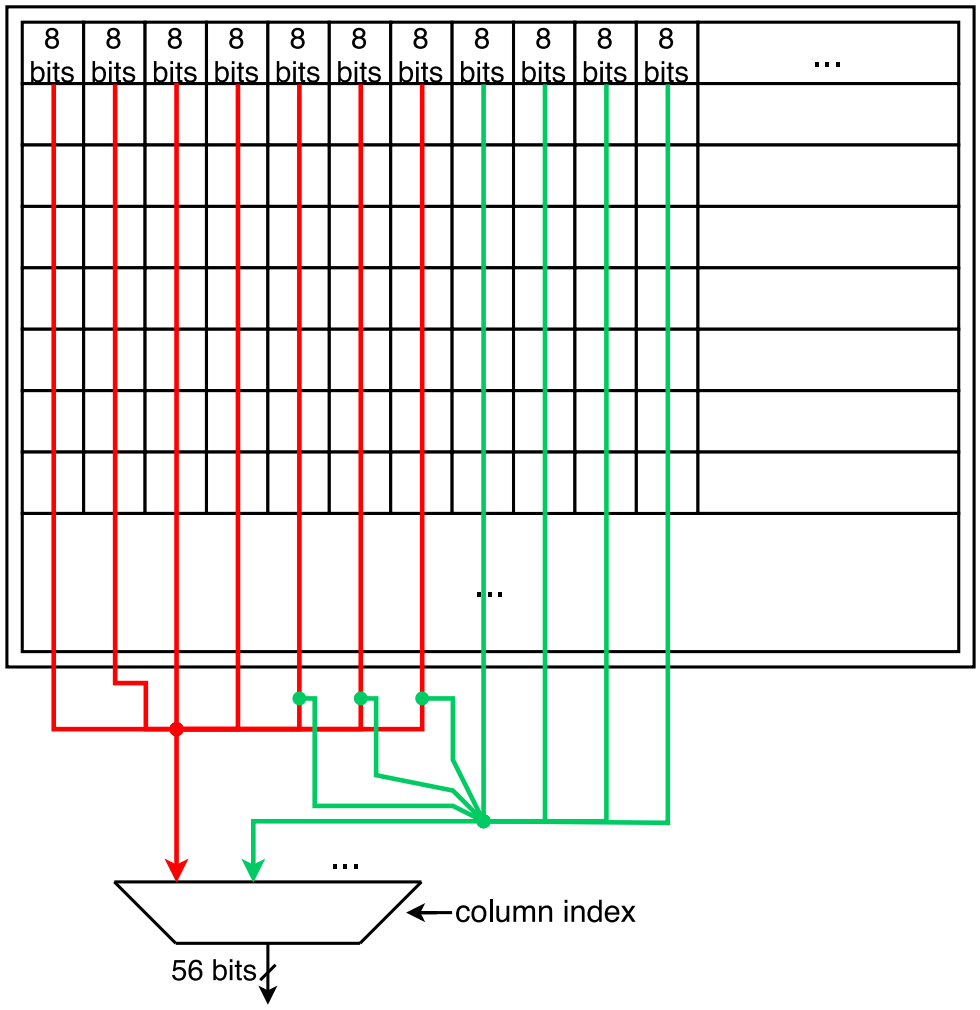


Figure 4.23: Weights memory: example of a reading operation.

changing the columns actually means moving of 4 cells to the right in a specific row. Evidently since in any case, 7 data are read at each reading operation, but from one column to the next one, there is a 4 cells shift, some data are read multiple times (Figure 4.23).

## 4.8 High-level scheme

By joining together all the implemented building blocks, the final hardware accelerator can be obtained.

In Figure 4.24 the high-level scheme of the whole accelerator is depicted.

As it is possible to see, a unique block is used for convolutional, fully-connected and batch normalization layers while the max-pooling one required additional logics to compute the maximum value given a certain number of inputs.

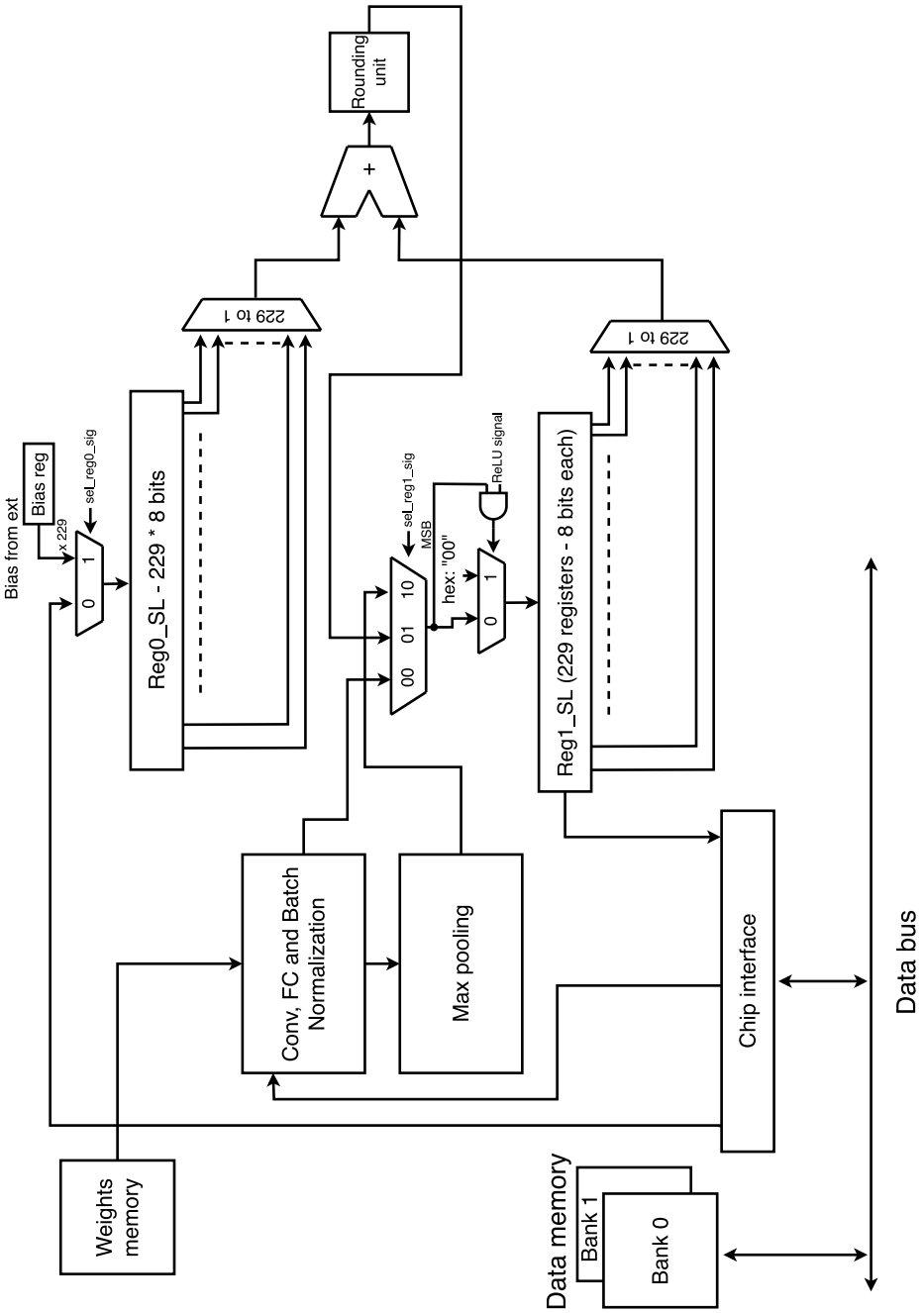


Figure 4.24: Hardware accelerator high-level scheme.

## 4.9 Chip parameters

In this section the configurable parameters which have to be provided as input are described:

- `start_conv`, `start_FC`, `start_max_pooling` and `start_BN` signals are used to select which operation has to be executed among all the implemented ones.

- `sel_input_format` signal is used for the rounding unit at the output of the LiM array. In particular if it is 1, it means that the input fixed-point format is Q3.5, otherwise, it is Q2.6. (The floating-point format used will be discussed in more details in the next chapter).

- `filter_width` is used to indicate the dimension of the filter to be used for the convolutional layers. It can be 1, 2, 3, 5 or 7.

- `input_rows` and `input_columns` parameters are used for the FC layers. They indicated how many rows and how many columns there are in the input. When the input of an FC layer is not a matrix (which happens for all the FC layers after the first one) then the input columns value indicates how many columns there are in the first row of the input.

- `output_width` is used to indicate how many values have to be computed for convolutional layers. Since the output of the convolutional layer is a matrix this parameter indicates either the number of rows or columns. It has to be noticed that this parameter has to be set also for the batch normalization layer because this layer is based on the convolutional one. The formula to compute this value is:
$$output\_width = \frac{input - filter\_width}{stride} + 1$$
It is worth highlighting that the `output_width` parameters has to be given without considering the padding zeros.

- `input_channels` and `output_channels` parameters indicate how many channels are given as input and how many of them the accelerator should compute as output.

- `padding` value indicates how many zeros have to be inserted as padding in the output. There are no constraints on its maximum value. However, looking at all the different neural network algorithms the maximum padding value does not exceed 4.
- `stride` parameter is used for convolutional layer and it can assume the values 1 or 2.
- `ReLU` signal indicates whether or not the ReLU activation function has to be executed.

# Chapter 5

# Software model of the designed accelerator

For verification purposes, it is important to write a software model which is able to emulate the designed hardware (HW). A software implementation generally is written with a high-level language to guarantee a faster simulation time which is very useful especially during the verification of the hardware.

Since neural networks models are implemented using fully-precision computation, a software model of the designed HW has been written without considering the limited number of bits. This first model is very useful because it allows checking if layer by layer everything in the HW accelerator works as supposed to.

One the correctness of the HW has been checked, the software model has been converted into a fixed-point one in order to understand how the accuracy changes to vary the number of bits and the adopted fixed-point format.

Both software models have been written with Matlab.

## 5.1 Fully-precision floating-point model

The first software implementation has been written without considering the limited number of bits to be used in an actual HW implementation. Each layer has been implemented in a parametric way (just like the HW) with functions in Matlab. In particular, the implemented functions are:

- `conv_function`.
- `FC_func`.
- `max_pool_func`.

The batch normalization layer has not been implemented since it is based on the convolutional one so it is sufficient to set the right parameters of the conv_function and use directly it.

## 5.2 Fixed-point model

From the fully-precision model, it is very simple to switch on a fixed-point one: it is sufficient to quantize each output to the number of bits at disposal adopting the same rounding technique used in HW accelerator. It is clear that all the inputs, as well as the weights and the biases value, have to be quantized to be given as input to either the HW accelerator or the fixed-point model.

The fixed-point model is quite useful since it allows to simulate different rounding methods and different fixed-point formats in order to find out which combination reaches the highest accuracy. Even so, it is fundamental to choose a neural network algorithm first to compare the results and the final accuracy of the fixed-point model with the floating-point one. The chosen neural network algorithm is LeNet-5.

## 5.3 LeNet-5 software model

The LeNet-5 neural network has been implemented first with Keras, which is an open-source library written in Python and capable of running on top of TensorFlow. Keras allows to implement, train and test a neural network very quickly. For instance, to define a model with Keras made of layers organized as a linear stack it is sufficient to write this line of code:

```
model = keras.Sequential()
```

Now that the model has been defined, it is possible to add layers to it in a very easy way. To define the whole LeNet-5 neural network the following lines have been written:

```
model.add(layers.Conv2D(filters=6, kernel_size=(5, 5), activation='relu',
      input_shape=(32,32,1)))
model.add(layers.MaxPooling2D())
model.add(layers.Conv2D(filters=16, kernel_size=(5, 5), activation='relu'))
model.add(layers.MaxPooling2D())
model.add(layers.Flatten())
```

```
model.add(layers.Dense(units=120, activation='relu'))
model.add(layers.Dense(units=84, activation='relu'))
model.add(layers.Dense(units=10, activation = 'softmax'))
```

Once the network has been defined, it has to be trained and then weights and biases values have to be extracted to be used with the Matlab model.

To extract biases and weights it is sufficient to use the `get_weights()` function. It has to be noticed that this function can be applied to a single layer so a for cycle is needed to extract all layers parameters. Therefore, the code required to extract the needed parameters will look similar to this:

```
for layer in model.layers:
intermediate_layer_model = Model(inputs=model.input, outputs=layer.output)
intermediate_output = intermediate_layer_model.predict(test['features'])
if(layer.name == model.layers[0].name):
np.save("conv1_input.npy", test['features'])
np.save("conv1_labels_input.npy", test['labels'])
np.save("conv1_weights.npy", layer.get_weights()[0])
np.save("conv1_bias.npy", layer.get_weights()[1])
np.save("conv1.npy", intermediate_output[:,:,:,:])
elif(layer.name == model.layers[1].name):
np.save("pool1.npy", intermediate_output[:,:,:,:])
elif(layer.name == model.layers[2].name):
np.save("conv2_weights.npy", layer.get_weights()[0])
np.save("conv2_bias.npy", layer.get_weights()[1])
np.save("conv2.npy", intermediate_output[:,:,:,:])
elif(layer.name == model.layers[3].name):
np.save("pool2.npy", intermediate_output[:,:,:,:])
elif(layer.name == model.layers[4].name):
np.save("flatten.npy", intermediate_output[:,:])
elif(layer.name == model.layers[5].name):
np.save("FC1_weights.npy", layer.get_weights()[0])
np.save("FC1_bias.npy", layer.get_weights()[1])
np.save("FC1.npy", intermediate_output[:,:])
elif(layer.name == model.layers[6].name):
np.save("FC2_weights.npy", layer.get_weights()[0])
np.save("FC2_bias.npy", layer.get_weights()[1])
np.save("FC2.npy", intermediate_output[:,:])
elif(layer.name == model.layers[7].name):
np.save("FC3_weights.npy", layer.get_weights()[0])
np.save("FC3_bias.npy", layer.get_weights()[1])
np.save("FC3.npy", intermediate_output[:,:])
print(intermediate_output.shape)
```

The code fragment above checks if the layer at each iteration corresponds to one of those implemented and based on which layer is, it will extract weights (`get_weights()`

`[0]`) and biases (`get_weights()[1]`). Some layers like the pooling ones do not have weight or biases to be extracted.

The partial outputs of each layer are also extracted in order to compare them with the Matlab model. This can be done by defining a sub-model which has the same input of the original one, but the output is the output of a specific layer. Then, the inference can be executed and the output can be actually evaluated.

It is worth highlighting that all the parameters and outputs are extracted as `numpy array`.

### 5.3.1 Weights and biases conversion

The data extracted from the Keras model of the neural network have to be converted in `txt` files to be used as inputs for the Matlab model and for the HW accelerator. A Matlab script which implements exactly this conversion has been written and the code is the following one:

```
conv1_weights = readNPY('conv1_weights.npy');
conv1_bias = readNPY('conv1_bias.npy');
fileID_w = fopen("conv1_weights.txt", 'w');
fileID_b = fopen("conv1_bias.txt", 'w');
for ch_out = 1:length(conv1_weights(1,1,1,:))
for row = 1:length(conv1_weights(:,1,1,1))
for column = 1:length(conv1_weights(1,:,1,1))
fprintf(fileID_w, "%1.5f ", conv1_weights(row, column, 1, ch_out));
end
fprintf(fileID_w, "\n");
end
fprintf(fileID_b, "%1.5f \n", conv1_bias(ch_out));
fprintf(fileID_w, "------------------------------ new ch weights\n");
end
fclose(fileID_w);
fclose(fileID_b);

conv2_weights = readNPY('conv2_weights.npy');
conv2_bias = readNPY('conv2_bias.npy');
fileID_w = fopen("conv2_weights.txt", 'w');
fileID_b = fopen("conv2_bias.txt", 'w');
for ch_out = 1:length(conv2_weights(1,1,1,:))
for ch_input = 1:length(conv2_weights(1,1,:,1))
for row = 1:length(conv2_weights(:,1,1,1))
for column = 1:length(conv2_weights(1,:,1,1))
fprintf(fileID_w, "%1.5f ", conv2_weights(row, column, ch_input, ch_out));
end
fprintf(fileID_w, "\n");
```

```
end
fprintf(fileID_b, "%1.5f \n", conv2_bias(ch_out));
fprintf(fileID_w, "--------------------------------new ch weights\n");
end
end
fclose(fileID_w);
fclose(fileID_b);


FC1_weights = readNPY('FC1_weights.npy');
FC1_bias = readNPY('FC1_bias.npy');
fileID_w = fopen("FC1_weights.txt", 'w');
fileID_b = fopen("FC1_bias.txt", 'w');
for ch_out=1:length(FC1_weights(1,:))
for ch_in=1: length(pool2(1,1,1,:))
j = 1;
for row=1:length(pool2(1,:,1,1))
for column=1:length(pool2(1,1,:,1))
fprintf(fileID_w, "%1.5f ",
      FC1_weights(ch_in+(j-1)*length(pool2(1,1,1,:)),ch_out));
j = j+1;
end
fprintf(fileID_w, "\n");
end
fprintf(fileID_b, "%1.5f \n", FC1_bias(ch_out));
fprintf(fileID_w, "--------------------------------new ch weights\n");
end
end
FC2_weights = readNPY('FC2_weights.npy');
FC2_bias = readNPY('FC2_bias.npy');
fileID_w = fopen("FC2_weights.txt", 'w');
fileID_b = fopen("FC2_bias.txt", 'w');
for ch_out=1:length(FC2_weights(1,:))
for ch_in =1:length(FC2_weights(:,1))
fprintf(fileID_w, "%1.5f ", FC2_weights(ch_in,ch_out));
end
fprintf(fileID_b, "%1.5f \n", FC2_bias(ch_out));
fprintf(fileID_w, "\n--------------------------------\n");
end
fclose(fileID_w);
fclose(fileID_b);


FC3_weights = readNPY('FC3_weights.npy');
FC3_bias = readNPY('FC3_bias.npy');
fileID_w = fopen("FC3_weights.txt", 'w');
fileID_b = fopen("FC3_bias.txt", 'w');
for ch_out=1:length(FC3_weights(1,:))
for ch_in =1:length(FC3_weights(:,1))
fprintf(fileID_w, "%1.5f ", FC3_weights(ch_in,ch_out));
```

```
78 end
   fprintf(fileID_b, "%1.5f \n", FC3_bias(ch_out));
80 fprintf(fileID_w, "\n-------------------------------\n");
   end
82 fclose(fileID_w);
   fclose(fileID_b);
84
   %%read input
86 fileID_data = fopen("input_conv1.txt", 'w');

88 for row = 1:length(in_conv1(1,:,1))
   for column = 1:length(in_conv1(1,1,:))
90 fprintf(fileID_data, "%2.5f ", double(in_conv1(1,row,column)));
   end
92 fprintf(fileID_data, "\n");
   end
94 fprintf(fileID_data, "-------------------------------\n");

96 fclose(fileID_data);
```

The readNPY function is available on GitHub and allows to convert a nunpy array into a multidimensional array which can be handled by Matlab. For each layer, a nested for cycle has been used to handle multiple channels. Furthermore, the input data extracted as well as the output of each layer contain multiple images so it is important to take this into account when converting the values.

### 5.3.2 Quantization function

The HW accelerator designed, as well as its Fixed-point Matlab model, uses a limited number of bits to represent data and parameters. For this reason, a quantization function is needed. Here is reported the Matlab function which performs the quantization:

```
   function [outputArg1] = quantizationFunc(value, n_bit_int, n_bit_fract)
2  %This function convert fractional number into number with
   if(value >= 0)
4  value = fix(value*2^(n_bit_fract)+0.5);
   else
6  value = value*2^(n_bit_fract);
   if(value == (fix(value)-0.5))
8  value = fix(value);
   else
10 value = round(value);
   end
12 end
```

```
output1 = value/(2^(n_bit_fract));
if(output1 > (2^(n_bit_int-1)-1/(2^n_bit_fract)))
outputArg1 = (2^(n_bit_int-1)-1/(2^n_bit_fract));
elseif(output1 <(-2^(n_bit_int-1)))
outputArg1 = (-2^(n_bit_int-1));
else
outputArg1 = output1;
end
end
```

First the number sign is checked: if it greater than 0 then the input value is rounded using the following formula:

$$round\_partial\_value(x) = fix(x \cdot 2^{n\_bit\_fract} + 0.5)$$

if instead the number was negative then the rounded value would have followed the following formula:

$$round\_partial\_value(x) = x \cdot 2^{n\_bit\_fract}$$

and then we can either use `fix` or `round` function to round the value.

The `fix` function rounds both positive and negative number towards 0, while `round` follows the round half-up scheme which means that if the number is $\geq 0.5$ then it is rounded to 1 otherwise to 0.

There is a different formula based on the sign of the input value. This just because the Matlab code is trying to model the HW: the rounding method implemented in HW is based on round half-up with a saturator. However, it is important to analyze what the HW actually does when rounding a negative number with this method. Suppose to have an input number like

$$input\_number = 11.01 \rightarrow -0.75$$

The objective is to obtain a rounded value equal to $-1$. Now with the method implemented in HW a 1 is added in the position after the truncation will happen. Moreover, this is a positive value which has to be added even in case of a negative number.

$$11.01+$$

$$00.10 =$$

$$---------$$

$$11.11$$

Then the number is truncated (in this case the truncation happens where the point is located) and the following result is obtained:

$$11 \rightarrow -1$$

Which is exactly the desired value. Thanks to this technique, from an HW point of view, always the same value is added either a positive value or a negative one is given as input. This technique works well due to the 2's complement which is used to represent the negative numbers.

The advantage is that a true rounding method is implemented for both positive and negative values with only one slightly difference: 0.5 and $-0.5$. Suppose to have the following numbers:

$$11.10 \rightarrow -0.5 \quad \text{and} \quad 01.10 \rightarrow +1.5$$

Now by applying the method explained before for the negative case, the following result would be obtained:

$$11.10+$$

$$00.10 =$$

$$---------$$

$$00.00 \rightarrow 00$$

while for the positive case:

$$01.10+$$

$$00.10 =$$

$$---------$$

$$10.00 \rightarrow 10$$

As it is possible to the 0.5 case is handled differently based on the sign of the input value: if the number is positive 0.5 is rounded towards 1; if it is negative $-0.5$ is rounded towards 0.

This is what happens in the HW implementation. To replicate this behaviour on Matlab two different functions have to be used and then in the case of a negative number the following code has to be added, in order to check the $-0.5$ case:

```
if(value == (fix(value)-0.5))
value = fix(value);
else
value = round(value);
end
```

Next, the following formula is applied:

$$round\_final\_value(x) = \frac{x}{2^{n\_bit\_fract}}$$

The remaining Matlab code implements the saturator.

Moreover, it is worth highlighting that the fixed-point format has to be established before calling this function.

### 5.3.3 Accuracy results and fixed-point format selection

To find the proper fixed-point format which does not decrease too much the final accuracy, tons of simulations have to be done because it is a matter of choosing the right format with the right number of bits.

To start with at least a proper number of bits, several papers on quantized neural networks have been read. From those, it is possible to say that if no particular quantization method (like the one based on logarithms) is used then 8 bits should be more than enough to guarantee a respectable accuracy.

Once the number of bits is fixed, the proper fixed-point format should be found. However, instead of going with a try and error approach, each input and each layer output of the floating-point implementation of the LeNet-5 has been analyzed.

The maximum input value is 255 while the maximum convolutional layer output is 215 and 96 is the maximum fully-connected layer one. Even so, all weights and bias are smaller than 0 which means that some fractional bits have to be taken into account otherwise each input would be multiplied by 0. Here is an example of weights and biases value for the first convolutional layer:

```
Weights example of the first convolutional layer
-0.13532 0.08340 -0.14414 0.02273 -0.14778
-0.04864 -0.09778 0.12905 0.13890 -0.10809
-0.09904 0.01753 -0.10780 -0.03043 -0.07611
-0.08719 -0.03902 -0.00248 -0.13409 0.11730
-0.01271 -0.02316 0.11619 0.16934 0.00220
Biases example of the first convolutional layer
-0.00003
-0.00082
-0.00102
0.00025
-0.00005
-0.00077
```

To solve this problem, there are two ways:

1. Increase the number of bits.

2. Find another solution.

Assuming the number of bits can not be changed (otherwise the HW accelerator would reach 16 or more bits internally), another solution has to be found. It is known that the basic operation of each layer in a neural network is a MAC (multiply and accumulate). If one is able to scale the input without affecting the final accuracy it would be possible to use still 8 bits without having weights and biases equal to 0. Fortunately, both addition and multiplication satisfy the distributive property which means that if a scale of a factor S is applied on both input and biases values (not weights ones) the output would be the right one scaled by S:

$$Output = W \cdot X + B$$

$$\frac{Output}{S} = \frac{W \cdot X + B}{S} = W \cdot \frac{X}{S} + \frac{B}{S}$$

By choosing $S = 100$ the max input value changes from 255 to just 2.55 which means that with 3 bits for the integer part all the input values can be represented. Assuming to have 8 bits at disposal, 3 can be used for the integer part and 5 for the fraction part meaning that the final fixed-point format is Q3.5.

It is important to notice here that apart from the first convolutional layer, all the others can be represented with only 2 integer bits (if divided by a factor S obviously). Even the first convolutional layer has a maximum value which is slightly greater than 2 ($215/100 = 2.15$).

The adopted fixed-point format is Q3.5 for the inputs and the weights of the first convolutional layer, while it is Q2.6 for all the other parameters. Even the output of the first convolutional layer is expressed with the Q2.6 format.

By using the format mentioned before a higher accuracy could be reached because more fractional bits are taken into account.

As a matter of fact by running the LeNet-5 quantized model with 300 test images and the previous fixed-point format an accuracy of 99.67% have been obtained which is amazing considering that the floating-point model, as well as the Keras model with double precision, had an accuracy of 98.67%.

# Chapter 6

# Verification

The verification step is one of the most important steps since it allows to verify the correct behaviour of the designed hardware. Creating a software model of the hardware allows comparing the results layer by layer in order to be sure that everything works as supposed to.

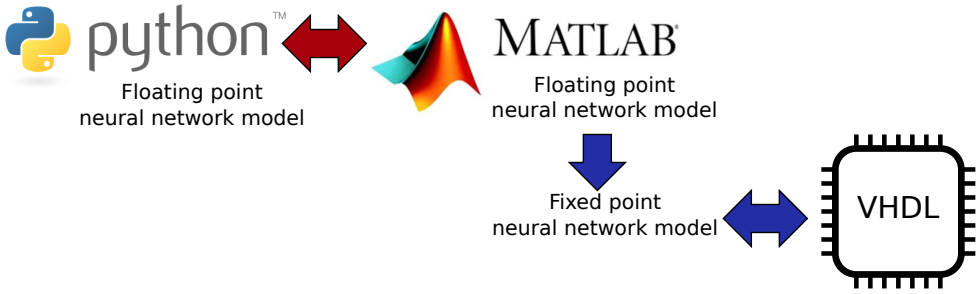


Figure 6.1: Verification flow [10].

In this specific case, the LeNet-5 has been tested with images coming from the `MNIST` database, so all the outputs here are referred to a particular neural network algorithm. Both Matlab and VHDL layer by layer outputs have been written on a file in order to make easier the verification step.

In the following the results of the second max-pooling layer are reported considering just the first two channels as an example:

```
Max-pooling 2 output (first two channels) -- Matlab fixed-point model
0.0000000     0.0000000     0.0000000     0.0000000     0.0000000
0.0000000     0.0000000     0.0000000     0.0000000     0.0000000
0.0000000     0.0000000     0.0000000     0.0000000     0.0000000
0.0000000     0.0000000     0.0000000     0.0000000     0.0000000
0.0000000     0.0000000     0.0000000     0.1718750     0.0000000
-------------------------
0.0000000     0.0000000     0.0000000     0.0000000     0.0000000
0.0000000     0.1718750     0.0000000     0.0000000     0.0000000
0.0000000     0.0000000     0.0000000     0.0000000     0.0000000
```

```
0.0000000      0.0000000      0.0000000      0.0000000      0.0000000
0.4375000      0.0000000      0.0000000      0.0000000      0.0312500
-------------------------
```

To be noticed how dash lines are used to separate the different output channels.

```
Max-pooling 2 output (first two channels) -- VHDL output
0.000000e+00     0.000000e+00     0.000000e+00     0.000000e+00     0.000000e+00
0.000000e+00     0.000000e+00     0.000000e+00     0.000000e+00     0.000000e+00
0.000000e+00     0.000000e+00     0.000000e+00     0.000000e+00     0.000000e+00
0.000000e+00     0.000000e+00     0.000000e+00     0.000000e+00     0.000000e+00
0.000000e+00     0.000000e+00     0.000000e+00     1.718750e-01     0.000000e+00
-----------------------------------------------------
0.000000e+00     0.000000e+00     0.000000e+00     0.000000e+00     0.000000e+00
0.000000e+00     1.718750e-01     0.000000e+00     0.000000e+00     0.000000e+00
0.000000e+00     0.000000e+00     0.000000e+00     0.000000e+00     0.000000e+00
0.000000e+00     0.000000e+00     0.000000e+00     0.000000e+00     0.000000e+00
4.375000e-01     0.000000e+00     0.000000e+00     0.000000e+00     3.125000e-02
-----------------------------------------------------
```

The two outputs are identical. It is clear that the final result is what matters the most because it tells which number image has been given as input.

In this case, the input image was the one reported in Figure 6.2.

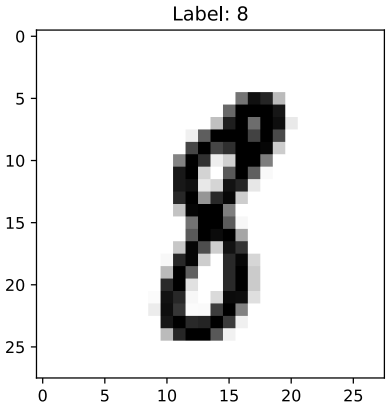


Figure 6.2: Input image.

The output of the last fully-connected layers are:

```
FC3 output -- Matlab fixed-point model
-0.0468750
0.0156250
0.0312500
-0.0156250
-0.0625000
-0.0312500
```

```
-0.0937500
-0.0781250
0.2031250
-0.0312500
```

The highest value from above is 0.2031250 in position 8 (from 0 to 9). This means that the recognized number is 8 which is correct.

```
FC3 output -- VHDL output
-4.687500e-02
1.562500e-02
3.125000e-02
-1.562500e-02
-6.250000e-02
-3.125000e-02
-9.375000e-02
-7.812500e-02
2.031250e-01
-3.125000e-02
```

The highest value from above is $2.031250e-01$ in position 8 (from 0 to 9). This means that the recognized number is 8 which is correct. The software and the hardware produce the same results and the highest value of the last layer output correspond to the input image which means that everything works as supposed to.

A Matlab script has been written in order to automatically compare the result of the HW with the image's label in order to see if the HW has recognized correctly the input image.

This script is actually very simple and it performs the multiplication by 100 of the HW output and then apply the `softmax` function. It has to be noticed that these last two operations are not mandatory.

```
FC3_out = "FC_3_output.txt";
input_matrix = zeros(1, 10);
fileID_d = fopen(FC3_out, 'r');
formatSpec = "%f";
numb_rows_ext = 1;
for i = 1:numb_rows_ext
line_d = fgetl(fileID_d);
prova = sscanf(line_d, formatSpec, [1 inf]);
input_matrix(i,1:length(prova)) = sscanf(line_d, formatSpec, [1 inf]);
end

out = softmax((input_matrix.*100)');
[max_value, max_index] = max(out);
predicted_value = max_index-1
```

```
fclose('all');

input_labels = readNPY("conv1_labels_input.npy");
correct_value = input_labels(300)
```

This code prints the predicted value from the HW accelerator and the correct one coming from the database `MNIST`. The index of the label is 300 just because in this case the tested image was the 300th in the `MNIST` database.

# Chapter 7

# Synthesis and Place & Route

In this chapter the synthesis and place & route results of the HW accelerator are reported.

## 7.1 Synthesis

The synthesis of the HW accelerator has been performed with Synopsys Design Compiler with two distinct CMOS technology nodes: 45 nm and 28 nm. For both technology nodes maximum frequency, area and power consumption have been computed.

### 7.1.1 45 nm technology node

The option used for the synthesis are:

- Clock period of 3.5 ns:

  ```
  create_clock -name MY_CLK -period 3.5 clk
  ```

- Clock uncertainty of 0.07 ns:

  ```
  set_clock_uncertainty 0.07 [get_clocks MY_CLK]
  ```

- Inputs and outputs delay:

  ```
  set_input_delay 0.5 -max -clock MY_CLK [remove_from_collection [all
  _inputs] clk]

  set_output_delay 0.5 -max -clock MY_CLK [all_outputs]
  ```

- `BUF_X4/A` input capacitance (3.4 fF) used as load for the outputs:

  ```
  set OLOAD [load_of NangateOpenCellLibrary/BUF_X4/A]
  ```

```
set_load $OLOAD [all_outputs]
```

Once the synthesis has been performed it is possible to use the commands `report_power`, `report_area` and `report_timing` to obtain respectively power consumption (in the worst case), area and maximum frequency.

It is worth highlighting how the `report_power` command computes the power in the worst case by setting all the inputs switching activities to 0.5. However the result of the worst case is quite important in this case because it is unfeasible to implement all the neural network algorithm and to compute for each of them the switching-activity-based power consumption. As a matter of fact the estimation of this more precise power consumption took 12 hours for a very simple neural network like LeNet-5 and the results are not so different with respect to the worst case to justify a so demanding analysis for all the neural networks algorithms.

In Table 7.1 the synthesis results are reported for 45 nm technology node.

| | **45 nm** |
|---|---|
| Area [$mm^2$] | 0.195 |
| Power [mW] | 41.89 |
| Power (LeNet-5) [mW] | 36.02 |

Table 7.1: Synthesis results.

The maximum frequency reached by this HW implementation is equal to:

$$F_{MAX} = 285.71\,\text{MHz}$$

### 7.1.2 28 nm technology node

For the 28 nm technology node, two distinct syntheses have been performed: in the first one the same frequency of the 45 nm technology node has been kept in order to see what benefits derives from a newer technology for what concerns the power consumption. In the second synthesis instead, the maximum frequency has been found to see how much faster the accelerator can get with newer technology.

In Table 7.2 it is possible to see the results of both syntheses.

| | **28 nm (f = 285.71 MHz)** | **28 nm (max freq.)** |
|---|---|---|
| Area [$mm^2$] | 0.142 | 0.142 |
| Power [mW] | 22.42 | 43.69 |

Table 7.2: Synthesis results.

With this newer technology the following maximum frequency has been reached:

$$F_{MAX} = 588.23\,\text{MHz}$$

As it is possible to see if the same frequency is kept the power consumption becomes half while with almost the same power consumption as in the older technology the maximum frequency can be doubled by changing the technology. So this means that this architecture critical path can benefit from newer technology implementation.

## 7.2 Place & Route

The synthesis results are very good to have an idea on how the designed hardware behaves in terms of power consumption and maximum frequency. However, in very complex designs these values may change significantly from the ones of the final chips.

To obtain a more accurate values (near to the actual ones) the place and route step has to be executed. This step involves deciding where to place all the components, circuitry and logic elements in a limited amount of space. Then the wires are connected to the placed components.

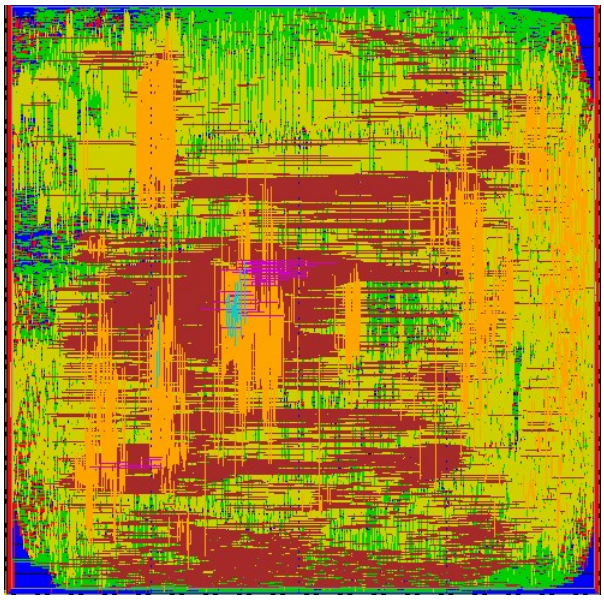

Figure 7.1: Physical chip of this hardware accelerator.

In Figure 7.1 the physical chip of the hardware accelerator is depicted.

In Table 7.3 the comparison of power, area and clock period between post-synthesis and post place & route are reported.

| | **Post-synthesis** | **Post place & route** |
|---|---|---|
| Area [$mm^2$] | 0.195 | 0.190 |
| Power [mW] | 41.89 | 169.1 |
| Frequency [MHz] | 285.71 | 270.27 |

Table 7.3: Synthesis vs place & route results for 45 nm.

The area and the frequency have decreased a little bit while the power consumption has increased of 4 times.

It is worth highlight the power estimation post place & route has been obtained by setting the inputs switching activity to 0.5.

Nevertheless this increasing in the power consumption might seem unjustified, but the clock signal and the interconnections in very complex design can reach an high percentage of the total power consumption and their contributions can be evaluated only after the place & route step.

Another important contribution to the final power consumption is the actual

load capacitance. As a matter of fact this value in post-synthesis simulation is fixed to 3.4 fF while after the place & route is evaluated as 0.616 nF. The capacitance computed by Innovus is closer to the real one since this is the final step in the design flow.

It has to be noticed that the clock period decreases of just 0.2 ns from post-synthesis to post place & route which means that the delay of all paths is well balanced and is not affected too much by the interconnections non-idealities.

# Chapter 8

# State of the art comparison

In this chapter, a comparison between this work and the state of the art has been performed. The parameters compared are:

- Area;
- power;
- execution time;
- FPS;
- energy per FPS;
- GOP/s/W;
- clock period.

One of the most used neural network for comparison purposes is the AlexNet. However, this HW implementation is not able to execute the AlexNet because it does not have the possibility to execute an 11x11 filter for the convolution. Even so, it is sufficient to add a proper control unit for the 11x11 case to solve this problem because the HW itself can execute every filter size.

In order to compute the execution time of distinct neural networks a series of parametric equations which are able to estimate the execution time of each layer have been extracted. Then with Matlab, all these equations have been implemented and a script which is able to compute all the parameters explained above has been written. It has to be underlined that this Matlab implementation is parametric which means that for every neural network parameters like execution time, FPS and energy per FPS can be computed.

It is clear that to have a precise estimation of the actual execution time it is important to consider also the time needed to write the data from the files to the memory and vice-versa.

The LeNet-5 has been implemented with this script and the execution time computed by Matlab is equal to the one computed with ModelSim. To have a fair comparison, the AlexNet has been implemented with this script and then compared with the state of the art. First of all a comparison with a previous thesis work [24] has been done. It is worth highlighting that the comparison has been made with synthesis result with a 45 nm technology node without considering how frequency and power are affected by the place & route (in the [24] these results are not available).

| | **This work** | **WINNER implementation [24]** |
|---|---|---|
| Area [$mm^2$] | 0.195 | 211.2 |
| Power [mW] | 41.89 | $70 \times 10^3$ |
| Execution time [ms] | $2.31 \times 10^3$ | 0.75 |
| FPS [1/s] | 0.43 | $1.32 \times 10^3$ |
| Energy per FPS [mJ/FPS] | 225.81 | 0.039 |
| Minimum clk period [ns] | 3.50 | 3.55 |

Table 8.1: Synthesis results compared with WINNER implementation [24].

It has to be noticed that the processing time of the different works analyzed is scaled to batch size equal to 1.

It is clear that from a power consumption point of view, this implementation is far more convenient since there are more than three orders of magnitude between the two compared power. However, WINNER implementation is a parallelized architecture which explains not only the high power consumption but also the extreme efficient architecture from an energy point of view. This happens because even if the power consumption is quite high, the WINNER implementation is able to finish the whole AlexNet much faster.

Another important parameter which is used to compare the performances between differences hardware implementations is the `GOP/s/W`.

This parameter is computed as the number of basic operations executed in one second divided by the power consumption. In this way it possible to compute the true performances of the accelerator without considering the time needed to

write/read data on/from a file.

Moreover, the way data are written in the memories could be different with respect to the one used in this architecture. In particular, larger memories can be used to store all the inputs channel and to store all the output channels avoiding the reading/writing operation of files every time a new channel is needed/computed. Another possible solution could be writing one output channel to a file while computing a new one without increasing the memory dimension.

These are only some solutions to mask or to avoid completely the time required to transfer data to/from the implemented memories.

For this reason, it is quite important to compute a parameter which evaluates the pure performance of the hardware in terms of operation per second and then divides it for its power consumption to obtain the efficiency of the accelerator.

Considering as the worst-case the first convolutional layer of the ZFNet but with just one input and output channels, it is possible to evaluate the GOP/s/W for this hardware accelerator. The time needed to write/read to/from a file has not been considered. In the following table (Table 8.2) this parameter is compared with other hardware implementations:

| | **This work** | **FPGA1** [23] | **FPGA2** [23] | **FPGA3** [23] | **FPGA4**[23] |
|---|---|---|---|---|---|
| Precision [N-bit] | **8** | 32 | 8/16 | 16 | 16 |
| GOP | **0.0012** | 1.33 | 30.9 | 30.76 | 1.45 |
| Power [W] | **0.042** | 18.61 | 25.8 | 9.63 | 30.2 |
| Performance [GOP/s] | **1.5** | 61.62 | 117.8 | 136.97 | 565.94 |
| Efficiency [GOP/s/W] | **34.9** | 3.31 | 4.57 | 4.22 | 22.15 |

Table 8.2: Comparison power and `GOP/s/W` between this work and other FPGA implementations [23].

where FPGA1 is Virtex-7 VX485T, FPGA2 is Stratix-V GSD8 FPGA3 is Zynq XC7Z045 and FPGA4 is Virtex-7 VC709.

From this new comparison, it is possible to see that the bottleneck of this work is the writing/reading operation to move data and weights from/to a file. As a matter of fact, the GOP/s/W of this architecture considering just the operations done from when the memories contain the right data and weights is the highest among all the analyzed implementations.

It has to be noted though that the results in Table 8.2 compares ASIC results

(this work) with FPGAs. To have a fair comparison other ASIC implementations should be analyzed (Table 8.3).

| | **This work** | **ShinDianNao [25]** | **Eyeriss [25]** | **Fulmine [25]** | **Origami[25]** |
|---|---|---|---|---|---|
| Technology [nm] | **45** | 65 | 65 LP | 65 LL | 65 |
| Area [mm$^2$] | **0.195** | 4.86 | 12.25 | 6.86 | 3.09 |
| Power [mW] | **42** | 320 | 278 | 24 | 93 |
| Performance [GOP/s] | **1.5** | 128 | 46 | 9.28 | 74 |
| Efficiency [GOP/s/W] | **34.9** | 400 | 166 | 618 | 804 |
| Efficiency per area unit [GOP/s/W/mm$^2$] | **178.97** | 82.30 | 13.55 | 90.09 | 260.19 |

Table 8.3: Comparison power and `GOP/s/W` between this work and other ASIC implementations [25].

where `LP` means Low Power while `LL` stands for Low Leakage.

From Table 8.3 it is possible to see that there are ASIC implementations far more efficient than this work in terms of `GOP/s/W`. From an area point of view, the smallest ASIC implementation among the analyzed ones is almost 16 times bigger than this work.

Hence, to take into account also the area another parameter can be computed: `efficiency per area unit` expressed as GOP/s/W/mm$^2$. As a matter of fact, if the area is taken into account, this solution is second only to the Origami architecture [25] in terms of efficiency per area unit.

## 8.1 Parallelization technique

The parallelization technique can be applied to this accelerator to improve the performances. In particular it possible to replicate the whole chip including the two memories. This is one of the strengths of this implementation: not only it is completely configurable for what concerns the number of bits and the specifications of each layer but it can be also parallelized based on how much area is at disposal and based on the power budget.

Theoretically if one parallelizes this architecture n-times, the execution time should decrease of the same amount. The AlexNet algorithm has in the worst case 384 distinct output channels. Since each output channel can be computed separately it is possible to execute each channel on a separate instance of the HW accelerator. For this reason, having more than 384 replicas is useless.

The Matlab script has been modified to take into account a parallelization of 384 times and every parameter has been recomputed. In Table 8.4 the parallelized accelerator has been compared with WINNER architecture.

| | **This work** | **WINNER implementation [24]** |
|---|---|---|
| Area [mm$^2$] | 74.88 | 211.2 |
| Power [mW] | $16.12 \times 10^3$ | $70 \times 10^3$ |
| Execution time [ms] | 9.8 | 0.75 |
| FPS [1/s] | 102 | $1.32 \times 10^3$ |
| Energy per FPS [mJ/FPS] | 1.56 | 0.039 |
| Minimum clk period [ns] | 3.50 | 3.55 |

Table 8.4: Parallelized accelerator compared with WINNER architecture [24].

As it is possible to see the execution time is improved by a factor of 235, which means that the performance improvement is not equal to the parallelization factor for this type of architecture.

It is straightforward to see that even with a parallelized architecture the performance in [24] can not be reached. However, it has to be noted that the power is 4 times smaller than WINNER implementation and the area is almost 3 times smaller.

If we increase the parallelization of a factor 4 (which means $4 * 384$) the same power consumption could be reached with an increase in terms of performance which is null in the case of AlexNet but which can be very important for a network like GoogLeNet which has up to 1000 output channels.

Supposing to use the parallelized version, it is possible to compare this work with other AlexNet models. One possible comparison is with a Floating-point implementation in terms of execution time, FPS and energy per FPS. To have a fair comparison, the execution time has to be scaled in order to represent the elaboration of a single frame. In Table 8.5 some state of the art results have been reported.

| | This work | GPU1 [23] | GPU2 |
|---|---|---|---|
| Execution time [ms] | 9.8 | 1.3 | 6.84 |
| FPS [1/s] | 102 | 769.0 | 680.8 |
| Energy per FPS [mJ/FPS] | 1.56 | 0.42 | 0.16 |
| Power [W] | 16.12 | 250 | 235 |

Table 8.5: State of the art comparison - Floating point vs this work.

Where GPU1 [23] is a GTX Titan X while GPU2's results are the one reported in section 3.3 and in this case a Nvidia Tesla K40c has been used.

It is clear that the GPU implementations are faster and more energy efficient but they consume more power. However, these are floating-point implementations.

In the next table (Table 8.6) a comparison with other fixed point work has been reported.

| | This work | FPGA1 [23] | FPGA2 [23] | FPGA3 [23] |
|---|---|---|---|---|
| Execution time [ms] | 9.8 | 21.61 | 20.10 | 2.56 |
| FPS [1/s] | 102 | 46.3 | 50.0 | 391.0 |
| Energy per FPS [mJ/FPS] | 1.56 | 8.68 | 7.68 | 0.2 |

Table 8.6: State of the art comparison - Fixed-point vs this work.

From these results, it is clear that this work is among the best in terms of efficiency thanks to low power consumption and a good parallelization.

All these considerations have been made by using the parallelization technique to improve the performance on a single image. However, this technique can be used to increase the number of images being elaborated simultaneously.

This means that in the same amount of time it is possible to compute a number of images which is equal to the parallelization factor.

So theoretically to match the same power consumption of the Winner architecture [24] a parallelization factor of 1666 should be used which means that the parallelized version is able to obtain the performances shown in Table 8.7.

| | **This work (k=1666)** | **WINNER implementation [24]** |
|---|---|---|
| Area [mm$^2$] | 324.87 | 211.2 |
| Power [mW] | $69.79 \times 10^3$ | $70 \times 10^3$ |
| Execution time [ms] | 1.39 | 0.75 |
| FPS [1/s] | 721.21 | $1.32 \times 10^3$ |
| Energy per FPS [mJ/FPS] | 0.13 | 0.039 |
| Minimum clk period [ns] | 3.50 | 3.55 |

Table 8.7: Parallelized accelerator (version 2) compared with WINNER implementation [24].

where `k` is the parallelization factor.

As it is possible to see there is still almost an order of magnitude of difference between the efficiency of the two architectures. The area has increased too and in this case, it is higher than the Winner implementation.

If compared with results in Table 8.6 and in Table 8.5 this architecture is much more efficient if parallelized with the second method. Obviously, the power consumption has increased as well but if it is not a problem then this could be another possible solution to increase the performances.

# Chapter 9

# Conclusions and future works

The reconfigurable in-memory accelerator for neural networks designed in this project is characterized by a very low power consumption without giving up too much on the performances. Multipliers have been substituted by simple gates and adders which means that they can be actually implemented in memory.

Furthermore, this hardware is completely reconfigurable and can execute the most important layers used in the most common Neural Networks.

It is clear that this accelerator has been designed for low power application, but based on the power budget one has, it is possible to parallelize this structure to increase the performances.

The parallelized version is among the best hardware implementations in terms of energy efficiency.

## 9.1 Future work

To improve the performances of this work several solutions can be tried:

- Implement the clock gating;
- Use a cache memory instead of an external data memory;
- Approximated arithmetic.

The clock gating can decrease significantly the overall power consumption and it can be applied to all the control units as well. In particular, the control unit hierarchy has been designed in such a way only a few machines are active per time. This means that the benefits of this technique can be quite important.

To improve the execution time a solution could be taking the data memory inside the accelerator and implement it as a cache memory. In this way, multiple rows could

be read at the same time decreasing significantly the time needed to store the data in the shift registers.

The performances of neural networks which use filter sizes bigger than 2x2 can benefit from an internal cache.

Furthermore, an approximated arithmetic can be used for multipliers and adders. This should increase the maximum clock frequency and at the same time, both the area and the power should decrease.

It is clear that the accuracy could be affected by this approximation but still, it might be interesting evaluating how much.

# Bibliography


[1] Vivienne Sze, Yu-Hsin Chen, Tien-Ju Yang, and Joel S Emer. Efficient processing of deep neural networks: A tutorial and survey. *Proceedings of the IEEE*, 105(12):2295–2329, 2017.

[2] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. Caffe: Convolutional architecture for fast feature embedding. In *Proceedings of the 22nd ACM international conference on Multimedia*, pages 675–678. ACM, 2014.

[3] Francisco J. Rodríguez, Antonio García, Pedro J. Pardo, Francisco Chávez, and Rafael M. Luque-Baena. Study and classification of plum varieties using image analysis and deep learning techniques. *Progress in Artificial Intelligence*, 7(2):119–127, Jun 2018.

[4] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott E. Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions. *CoRR*, abs/1409.4842, 2014.

[5] Sik-Ho Tsang. Review: Vggnet - 1st runner-up (image classification), winner (localization) in ilsvrc 2014, Sep 2019.

[6] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. *CoRR*, abs/1512.03385, 2015.

[7] Gao Huang, Zhuang Liu, and Kilian Q. Weinberger. Densely connected convolutional networks. *CoRR*, abs/1608.06993, 2016.

[8] Mahendran Venkatachalam. Recurrent neural networks, Jun 2019.

[9] Erik Lindholm, John Nickolls, Stuart Oberman, and John Montrym. Nvidia tesla: A unified graphics and computing architecture. *IEEE micro*, 28(2):39–55, 2008.

[10] Andrea Coluccio. In-memory binary neural networks. Master's thesis, Politecnico di Torino, 2019.

[11] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.

[12] Matthew D. Zeiler and Rob Fergus. Visualizing and understanding convolutional networks. *CoRR*, abs/1311.2901, 2013.

[13] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, Nov 1998.

[14] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

[15] Sik-Ho Tsang. Review: Densenet - dense convolutional network (image classification), Mar 2019.

[16] Giulia Santoro, Giovanna Turvani, and Mariagrazia Graziano. New logic-in-memory paradigms: An architectural and technological perspective. *Micromachines*, 10(6):368, May 2019.

[17] Frederik Kratzert. Finetuning alexnet with tensorflow. *Flair of Machine Learning - A virtual proof that name is awesome!*

[18] Uri Merhav. How to (quickly) build a tensorflow training pipeline. *Medium*, Nov 2018.

[19] Data input pipeline performance : Tensorflow core. *TensorFlow*.

[20] Using gpus : Tensorflow core. *TensorFlow*.

[21] Illarion Khlestov. How to profile tensorflow. *Medium*, Mar 2017.

[22] P. Maciol and K. Banas. Testing tesla architecture for scientific computing: The performance of matrix-vector product. In *2008 International Multiconference on Computer Science and Information Technology*, pages 285–291, Oct 2008.

[23] Huimin Li, Xitian Fan, Li Jiao, Wei Cao, Xuegong Zhou, and Lingli Wang. A high performance fpga-based accelerator for large-scale convolutional neural networks. In *2016 26th International Conference on Field Programmable Logic and Applications (FPL)*, pages 1–9, Aug 2016.

[24] Simone Domenico Antonietta. Weights in-memory neural network embedded ram. Master's thesis, Politecnico di Torino, 2019.

[25] F. Conti, R. Schilling, P. D. Schiavone, A. Pullini, D. Rossi, F. K. GÃ$\frac{1}{4}$rkaynak, M. Muehlberghuber, M. Gautschi, I. Loi, G. Haugou, S. Mangard, and L. Benini. An iot endpoint system-on-chip for secure and energy-efficient near-sensor analytics. *IEEE Transactions on Circuits and Systems I: Regular Papers*, 64(9):2481–2494, Sep. 2017.