POLITECNICO DI TORINO

Master's Degree in Electronic Engineering

Master's Degree Thesis

WINNER

Weights In-memory Neural Network Embedded Ram

Supervisors:
Prof. Maurizio Zamboni
Prof. Mariagrazia Graziano
Ph.D. Giovanna Turvani

Candidate:
Simone Domenico Antonietta

April 2019

Acknowledgments

With these short rows, I want to thank people supported me in my thesis experience; I cannot explain how much they helped me, but I can remember them using this page.
A first thank to my parents, who permitted me to realize my dream to study in Politecnico di Torino, and a second one for my sister with the baby Emma, who cheered up her uncle also when he was tired or unhappy. Another special thank to Beatrice, who encouraged me in the worst moments, and a thought for all my friends who believed in me from the beginning.
Last but not least I want to remember my grandma, I cannot see her with my eyes but I know she helps me each day, and my grandpa, who told me many times that I could finish.
I don't know where I could be without the people I remembered here, but I know where I am now, that is the place I wanted to reach, so thank you for helping me to be here.

Abstract

Neural Networks can be a good solution to implement complex problems as image or speech recognition, so nowadays they are often used in many applications, like security, biomedicine and robotics. However Neural Networks are very big circuits, requiring high performance hardware.
The platform of choice for these circuits is a microprocessor, on which is possible to write software that implements the algorithm wanted. However, the capability of a microprocessor based on the standard Von Neumann architecture is severally affected by the memory bottleneck problem: the CPU spends the most of its time waiting the data provided by the main memory. This problem is more relevant in high performance systems. An important difference between Neural Networks and standard microprocessors is the organization of the memory: in the first case it is distributed on all the internal blocks of the system, in the second one it is placed into a unique block.
The solution proposed is the Weights In-memory Neural Network Embedded Ram (WINNER) architecture, an ASIC implementation of one of the most studied Neural Networks. The name WINNER is chosen because it is an innovative hardware solution that represents the starting point for a new generation of hardware Neural Networks, based on the same approach of the WINNER one.
The work of this thesis consists in the implementation of a Convolutional Neural

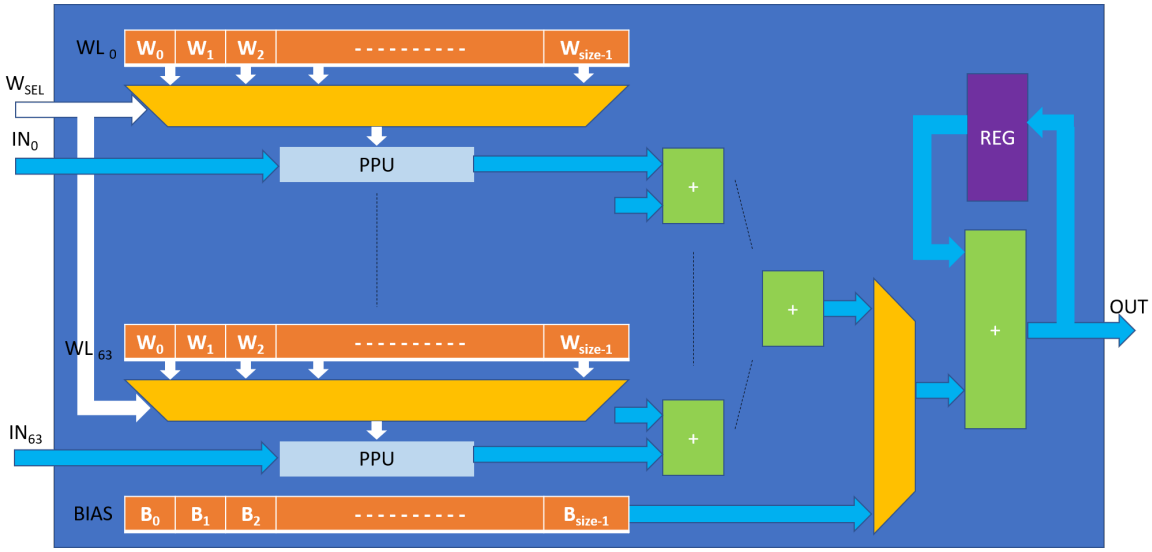


Figure 1: WINNER Neuron Block Scheme

Network for image recognition, the AlexNet, implemented on dedicated hardware. It is divided into two main blocks: the first one contains the network weights and implements the computational logic near them, while the latter controls the first one, generating the inputs and temporarily storing the internal output of the network.
The main innovation of this architecture is the hardware implementation of a Neural Network using the In-Memory approach: part of the computation is therefore evaluated inside the memory. Figure 1 represents the structure of a single neuron of the network composed of:

- the WL_0, WL_{63} and BIAS blocks, that are memory rows where network weights (W_0,...,W_{size-1}) and bias (B_0,...,B_{size-1}) are stored
- the trapezoidal blocks, that are multiplexers used to select the proper values, depending on the algorithm step executed
- the PPU blocks, that make a pre-evaluation of the memory data selected
- the adder blocks, that sum the PPU outputs together
- the REG block, that is a register that can store temporarily a partial result

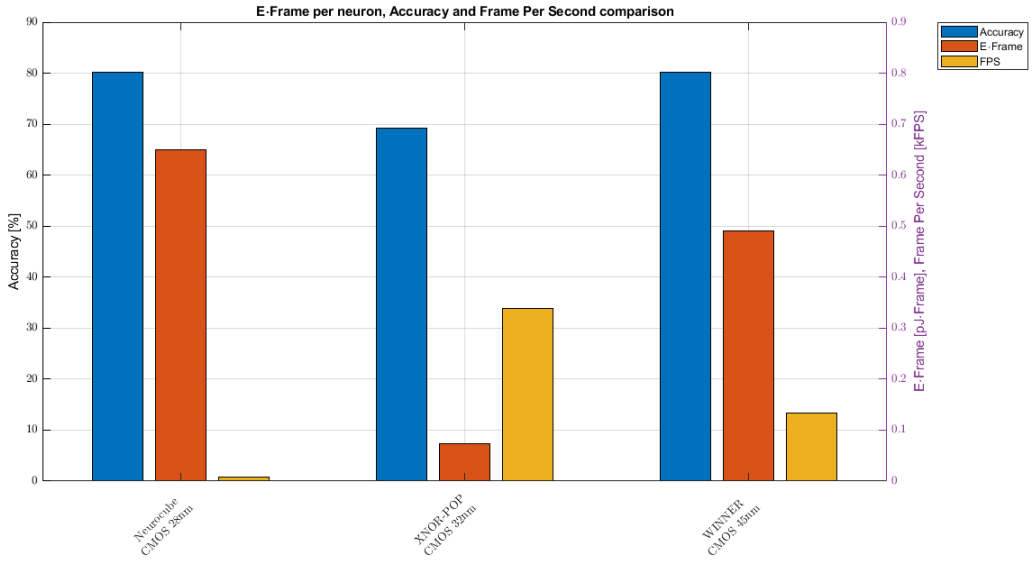


Figure 2: Comparison between the WINNER architecture and the main PIM ones

Each neuron, composed of 64 words, contains the weights needed for all the algorithm steps, so, fixing the step, the proper part of the word is selected.

The hardware obtained has a generic architecture, so it is possible to implement every type of neural network only changing the size of the words. The main reason why this organization is characterized by high performance relies on the fact that it combines the Logic In Memory approach with the Neural Network structure.
Figure 2 shows a comparison between the energy per frame obtained on the WINNER architecture and on other two in-memory solutions that implement the AlexNet: the Neurocube and the XNOR-POP. The picture reports also the accuracy reached by the networks and the Frame Per Second (FPS) used by the reference architectures; in order to compare the performance of this work with the state of the art. In particular, it is possible to see that the best performance is reached by the XNOR-POP; however, it implements an approximated version of the AlexNet, reaching around 10% less accuracy respect on the other two. The WINNER architecture, on the contrary, implements the full version of the AlexNet while using bigger transistors. The results are better than the Neurocube in every aspect. According to these considerations, it is possible to say that the WINNER approach represents a good starting point to implement big Neural Networks on a chip, while a large set of optimizations are still possible. For example, by using smaller transistors the energy efficiency can be greatly increased.

Table of contents

List of figures

Introduction

Technological applications requiring Neural Networks to recognize and classify images, sounds and input patterns are quickly increasing. These algorithms are increasing their complexity to implement more features, so their implementations need many resources to obtain good performance.

A big Neural Network is typically built with a software solution because it is possible to use a very large memory, the RAM, and it is simple to describe it using programming languages. However, this approach puts constraints on the performance obtained, because a Neural Network evaluates the data in a parallel way, using a distributed memory; in a software solution it is used a microprocessor system based on a Von Neumann architecture, characterized by the separation between memory and the elaboration unit, as shown in figure 3. In order to preserve the memory

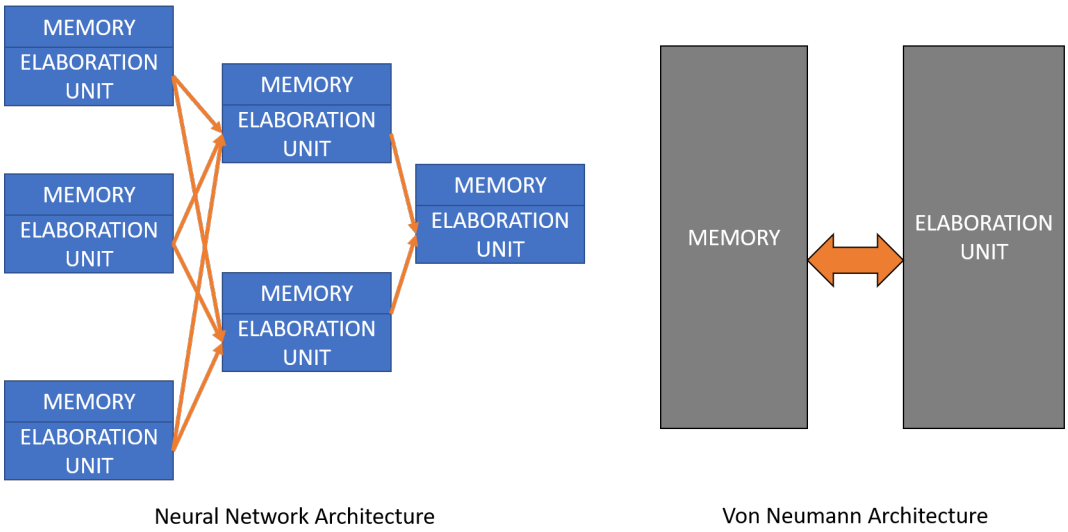


Figure 3: Neural Network and Von Neumann architectures

organization of a Neural Network, it is needed dedicated hardware that provides a simple way to reconfigure its memory, as for the RAM into the Von Neumann architecture.

The challenge proposed by the modern applications consists of the implementation

of a system similar to a brain, on which memory and computation are everywhere distributed, with the flexibility of a microprocessor. The Weights In-memory Neural Network Embedded Ram (WINNER) architecture wants to solve the problem described, providing an innovative hardware solution; the name WINNER is chosen because the purpose is to obtain an architecture that can be the starting point for a new generation of hardware Neural Networks with very high performance and computational capability.
The thesis is divided into 7 chapters:

- chapter 1, the State of the Art, contains the solutions adopted nowadays, describing the different technologies they use;
- chapter 2 shows the software implementation of the reference network chosen, the AlexNet, explaining the features added to make the hardware realization simpler;
- chapter 3 describes the hardware implementation using block schemes to show the component organization;
- chapter 4 illustrates the simulation process, reporting simulation images to explain how it is possible to test the hardware produced;
- chapter 5 contains the synthesis process used;
- chapter 6 shows all the results obtained after the synthesis step;
- chapter 7 reports the conclusions of the work done and analyses of the possible future improvements.

After these chapters, there are 3 appendices that describe in more details some arguments used in the project presented:

- appendix A illustrates the AlexNet model used;
- appendix B describes the MBE multiplier implementation;
- appendix C shows Taylor's series expansions.

Chapter 1

State of the art

The classical Von Neumann architecture provides a structure made up by memory, on which data and instructions are stored, and an elaboration unit, as represented in figure 1.1. This kind of approach is affected by an important problem: the bottleneck

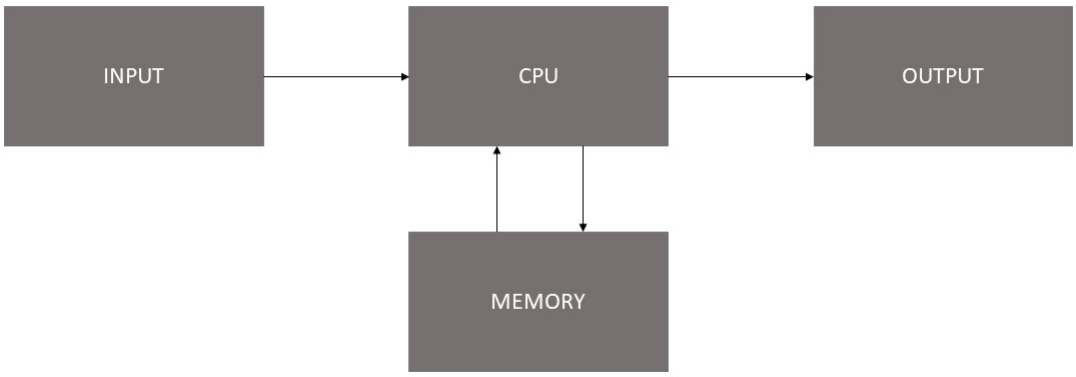


Figure 1.1: Von Neumann Architecture

between the two main blocks. In fact, what happens is that the Central Processing Unit (CPU) spends the most part of its time waiting for the data arriving from the memory, due to the fact that actual CPUs operate at a very high frequency and the memory storage capacity is very high, causing a non-negligible latency.
Engineers dedicate a lot of their time searching other possible architectures because of this strong performance limit. The ideal solution is to merge the two units into one big black-box containing logic and memory at the same time. This concept is called in literature Logic-In-Memory (LIM) or Processing-In-Memory (PIM), and consists of a big memory array, on which each cell or group of cells is made up by a part of memory and a part of logic.
There are several possible PIM implementations, one of these takes inspiration to the human brain: it is a unit with memory and evaluation mixed into the same

structure. An architecture that is organized as the brain is called Neuromorphic Architecture, and the network that implements is a Neural Network (NN).

1.1 Neuromorphic Architectures

The brain is probably the oldest calculator in the world. Its structure is created to reach very high parallelism and to have a low power dissipation, so it is a good example of energy-efficient architecture.
There are several internal components into the brain, but to understand how it works it is possible to consider only two of them: neurons and synapses.

In figure 1.2 is represented the scheme of a neuron: there are several arrows

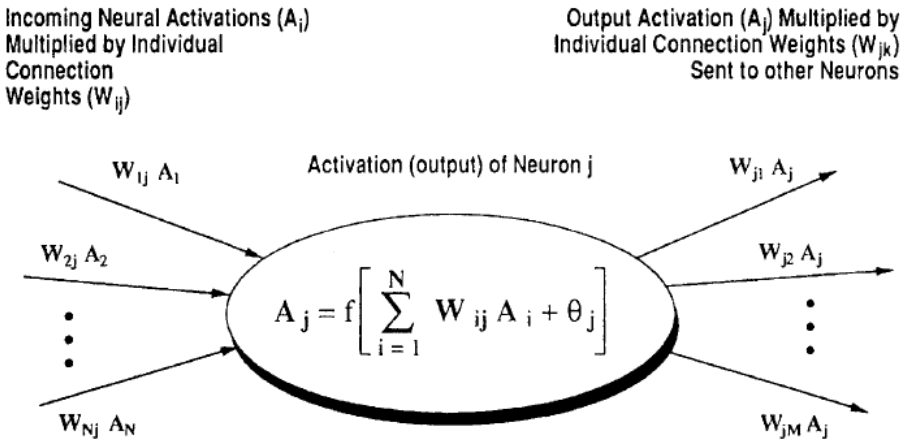


Figure 1.2: Schematic representation of a Neuron. Image adapted from [1]

in input and output, called synapses, each one with a specific weight (W). The neuron is drawn as a circle containing the activation function, that is the function to provide the output. The value generated is transported through the synapses to the next neurons, generating a network as in figure 1.3. The scheme obtained can be divided into different layers, each one containing neurons and synapses at the same topological level.
A neuron evaluates the stimuli received from its synapses and, considering them

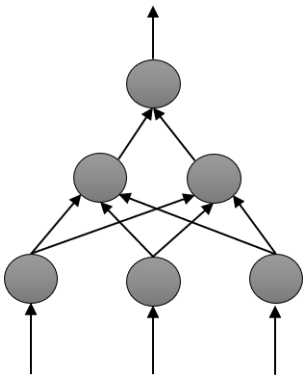


Figure 1.3: An example of Neural Network topology

with different weights, provides an output that becomes an input for another neuron passing through another synapse. This output consists of a spike produced only if the internal evaluation provides a result higher than the neuron threshold.
In this simple brain view there is a hierarchy of neuron: the meaning of the output provided by a neuron depends on its hierarchy level. According to this, it is simple to identify sequential layers, each one with a specific role; the last of them classifies the input stimuli into a class selected by a set of known classes. In this way, the brain can recognize known images, smells, sounds and situations.
The architecture described was the first inspiration to provide an alternative to the Von Neumann approach and took the name of Neural Network (NN); an example of a four layer NN is reported in figure 1.4.
In literature there are many NN types:

- **Artficial Neural Network (ANN)**, a NN with artificial neurons that use a certain mathematical function to provide a result; it is the most general category and each non-biological NN belongs to it. An ANN is made up by an input layer, an output layer, and one or more hidden layers.

- **Spike Neural Network (SNN)**, a NN that provides spikes instead of values as neuron outputs; this network is similar to a biological one. In that way the NN obtained can be artificial or biological, depending on how neurons are implemented.

- **Depp Neural Network (DNN)**, a NN that has more internal layers; its

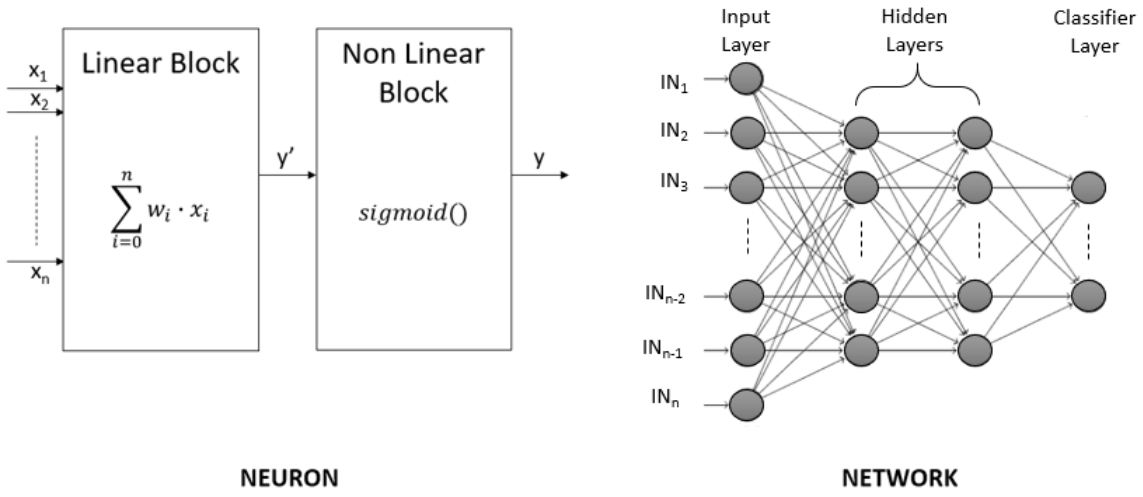


Figure 1.4: Representation of a generic neuron (on the left) and an example of possible NN implementation (on the right)

depth degree indicates the number of hidden layers that compose it. The pattern produced by an internal layer (called hidden layer) constitutes the feature map.

- **Fully Connected Neural Network (FC)**, a topological classification to identify a NN composed of layers characterized by neurons connected to each neuron of the next layer.
- **Binary Neural Network (BNN)**, an ANN that provides weights and output of a neuron in binary mode, indicating them with '0' or '1'.
- **Convolutional Neural Network (CNN)**, an ANN that uses more layers to classify the input pattern using the convolution operation on the input of each neuron. Typically a DNN uses different kinds of layer to select specific information on the input pattern and, elaborating them in other layers, it provides the final classification. It is very useful to replace a deep fully connected layer with a CNN because of the reduced number of neurons and connections.

A generic NN can be classified into more than one of these categories: they use different criteria to characterize that.
In an ANN the main characteristics of the brain are implemented using mathematical

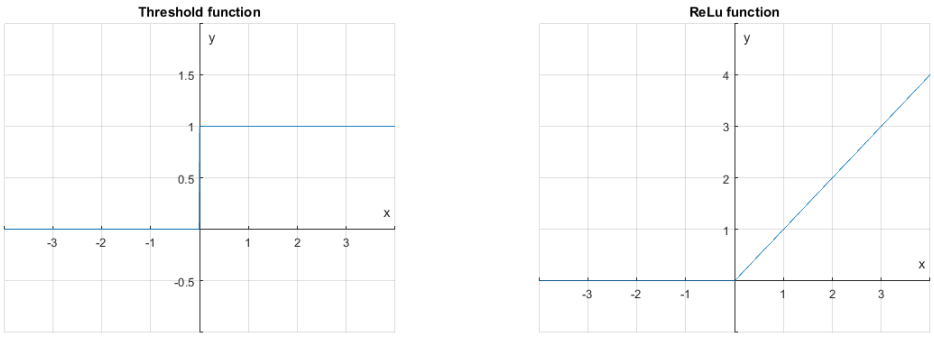
models; the most important mathematical contribution is on the evaluation of the neuron output. There are several models to obtain it:

- **Threshold function**: this function is the most similar to the biological approach because it provides a null output until the threshold set is reached. If this limit is equal to the result evaluated, the output becomes a pre-fixed value. This behaviour, represented in figure 1.5(a), is more useful when the output is '0' or '1', as in BNNs.

- **ReLu function**: it provides a null result for each negative input and a result equal to the input if it is positive, as in figure 1.5(b). It is commonly used in CNN to make the inter-layer values non-negative numbers.

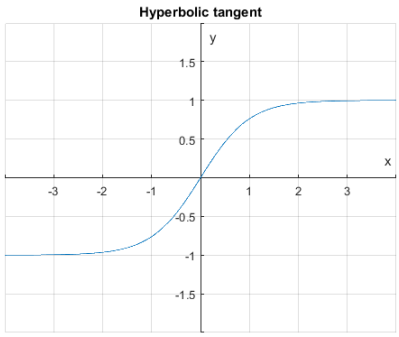
- **Hyperbolic tangent function**: the result of this function is limited between -1 and +1, so it provides a useful instrument to limit the hidden layer result range of a DNN; it is represented in figure 1.5(c). It belongs to the Sigmoid functions family, due to the graphical "S" form.

Another important part of a NN is the training: the process to assign the synapses weights. In a human brain the network continuously evolves, changing the weights due to a learning process that sometimes can happen. In an ANN is possible to simulate this behaviour using ON-LINE training, that consists on changing the weights to learn during the working phase of the network. Vice-versa, if the training is done only before starting to work, the approach is called OFF-LINE training.
As described for the human brain, the purpose of a NN is to recognize an input pattern and classify it in a category known. The most diffuse application is the image classification, which presents some critical points: the determination of the input images size and the number of output classifiers; increasing one of these two parameters, the network makes bigger, using more layers. This situation requires a NN capable to extract from the image analysed certain kinds of information, combining them to establish what is represented. This process is well implemented in CNNs, composed of several layers, each one with a specific purpose. They form together a complex structure capable to combine the partial detail extraction of the



(a)

(b)



(c)

Figure 1.5: Neuron output functions: (a) Threshold, (b) ReLu, (c) Hyperbolic tangent.

first layers to select one output category.
The main layers into a generic CNN are:

- **Convolutional Layers**: layers that implement filters with a fixed size; each filter is applied to a portion of the input pattern providing a result. Filters move with a specific stride; repeating this process many times, they cover all the input set. Each filter detects a particular characteristic, so its output is higher if the input set considered presents it.

- **Normalization Layers**: the result of convolution can reach high value, due to the fact that it is a sum of multiplications. To limit their output range it is possible to apply one normalization layer that, considering sequences of results,

makes them more uniform. It is typically applied after the first convolutional layers.

- **Pooling Layers**: in order to reduce the size of a layer, the pooling operation scans the input matrix as described for the Convolutional Layer. Each input set is replaced with only one value, usually computed as the average or the maximum of them.

- **FC Layers**: generally they are the last layers and they are used to combine all the previously information together providing the output classification.

There are many possibilities to implement a NN, from a pure software solution to a pure hardware one. If the algorithm is realized as a program running on a conventional Von Neumann architecture, the system is affected by the bottleneck of it, because the network becomes a simulation of a NN. The main consequence is a limitation on the energy-efficiency due to the reducing of the timing performance. For this reason, a pure hardware approach can produce a better result. In order to obtain high performance and maintain the power dissipation low, it's needed a short analysis on the main technologies available to realize a NN.

1.2 Technology choices

There are several technology alternatives to realize a neuromorphic architecture; the main one are:

1. Memristors-based;
2. MTJ-based;
3. PIM-based;
4. Many-core-based;
5. CMOS-based.

Each of these can introduce some advantages and disadvantages, analysed in the following.

Memristor

Memristors are electrical components capable to change their conductance respect on the amount of charge stored into them. To change the resistance of a memristor is sufficient to apply a current through it: in this way, a flow of charges passes through the device, varying the quantity of charge stored and causing a conductance change. The state assumed remains until another changing is externally imposed. This behaviour is very useful because it is possible to realize a component that implements a kind of non-volatile memory using only voltage and current, that are the simplest electrical quantities to manage.
This device can assume any resistance state, so it is an analogical component. In the digital world all the circuits are typically binary, so it is possible to use only two resistance values, one to represent the high level of a bit, the other one for the low level. In this way the standard CMOS memories can be replaced with memristor-based ones. In a second step, it is possible to use the analogical behaviour of a memristor to store in the same cell more than one bit, implementing a more compact layout.

General purpose neural memristor-based processor

Due to the fact that each NN has its specific requirement, as the number of neurons, the layer amount and the quantity of inputs, it is very hard to make a general purpose neural processor. This argument is fronted by the paper [11], that proposes a possible architecture to obtain this result.
The first problem to consider is the flexibility that the architecture must have; the solution shown is to divide the entire circuit into tiles of fixed length, connected with transmission gates (TGs) between them (figure 1.6) to provide the possibility to change the main parameters of the network according to what is needed. Implementing a NN with a memristor crossbar provides many advantages; the main one is to have synapses that are programmable with an external current and neurons that evaluate the output directly adding the current that flows into the synapses. It is possible to make it using a differential amplifier (figure 1.7), that has two input: the positive one is connected to all the synapses that have a positive weight, the negative one to the synapses associated to negative weights. In this way in the two lines

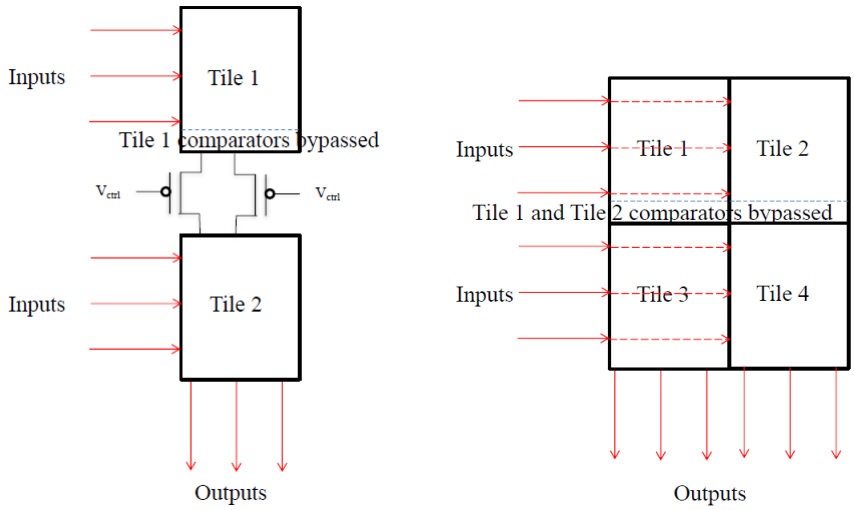


Figure 1.6: (a) Usage of the TG to connect two tiles. (b) Tiles combined to form a bigger array. [11]

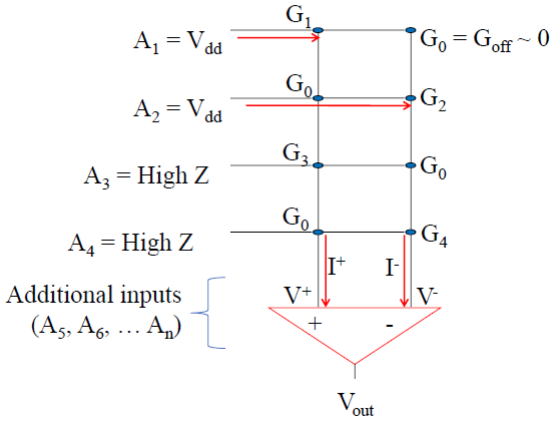


Figure 1.7: Differential amplifier used in the general purpose architecture. [11]

flow respectively the currents sum of positive synapses and the negative synapses one, so, making the difference between them using the differential comparator, it is possible to establish the result: if positive, the output will be a high logic level, on the contrary it will be a low logic level.

Power efficient structure for RRAM-based CNN

Many times it is useful to put together more crossbars, for example to implement a higher accuracy circuit, or to manage signed weights in a NN or simply to increment the size of the crossbar.
The approach proposed in [12] permits to provide these features without adding additional functions, saving area and power. The key element of the architecture presented is the decoder that selects the crossbar lines. A traditional decoder turns on all the transmission gate circuits during the evaluation phase, using the circuit in figure 1.8a. It is possible to modify this device according to the scheme shown in figure 1.8b, using a multiplexer: during computation, the V_{in} signal is selected, so, if it assumes a high logic level, the TG is enabled, and the extra port, connected to the added information, can propagate. In this way not all the cells are activated during the evaluation, but only the ones that have the V_{in} input at '1'. This modification

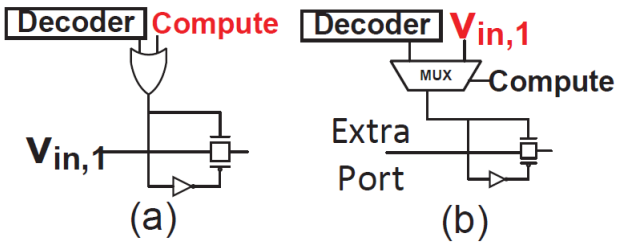


Figure 1.8: (a) Traditional decoder. (b) SEI decoder. [12]

permits to save the energy of all the cells that were enabled in the original structure and, at the same time, it allows the architecture to add shared informations between the rows, as the weight sign bit or the precision bias, obtaining a more flexible circuit.

MTJ

Magnetic Tunnel Junction devices (MTJs) are devices that can change their resistance due to an input current. To read the value stored on it, it is needed a read current, so it is possible to store information as in a traditional CMOS memory, but associating the bit value to the resistance set.
The structure of a perpendicular-MTJ (pMTJ) is shown in figure 1.9; the two external layers are made up by ferromagnetic materials, the internal layer is a very thin

oxide, in order to generate a tunnel junction. It is possible to change the resistance associated on it changing the magnetization of the two external layer: a parallel magnetization causes a low resistance, an anti-parallel one a high resistance. To use this device as a memory, the bottom layer is used to provide the magnetization current and the top one is used to store the information. The operation to write the bit value into the MTJ is a stochastic process, with a probability that increases with the amplitude of the current applied. Due to the fact that the information is associated to the resistance of the tunnel junction, the device consumes low power, because the tunnel current is very low, so the leakage is near-zero. This characteristic makes the MTJ very useful in applications as the NNs, that are made up by a high number of neurons, each one consuming power. There are several neural architectures that use

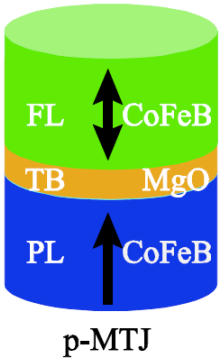


Figure 1.9: pMTJ structure: Free Layer (FL),Thin oxide Barrier (TB), Pinned Layer (PL). [2]

this kind of technology; in the following they are reported the most significant ones.

Fully connected Signle-Layer STT-MTJ based SNN

The architecture developed in [3] shows how to use MTJs to produce both neurons and synapses in a NN, as shown in figure 1.10. The NN implemented is a SNN designed for pattern recognition, in which the spiking neuron is realized with a MTJ and an inverting amplifier, implementing three operation mode, selected by three different transistors, as in figure 1.10b:

- stochastic writing mode: in this mode, the synaptic current is applied to the MTJ through the amplifier. The probability to switch the MTJ state from

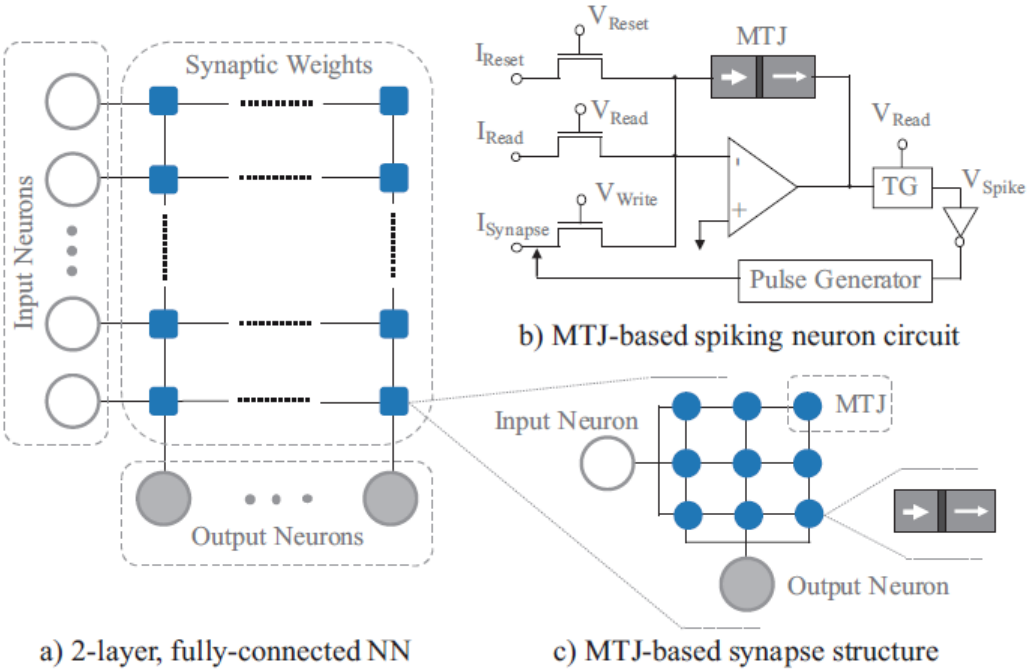


Figure 1.10: SNN architecture. [3]

parallel configuration, that is the default configuration, to the anti-parallel one, increases with the input signal amplitude; when the changing happens, a read operation is needed to propagate the output.

- read mode: a read current is applied in input and the output transmission gate is enabled, so, if the MTJ threshold is reached, the pulse generator is triggered, generating a spike.

- reset mode: after the reading step to transfer the information, the MTJ is reset to the parallel configuration, to initialize the circuit for a new evaluation.

For what concern the synapses, they are implemented as Compound Magneto-resistive Synapse (CMS), as shown in figure 1.10c. They are made up by N binary MTJ elements connected in a shunt configuration, so, from an external point of view, its behaviour is the same of a single synapse with a conductance that is the sum of the resistive device ones. The result is a synapse with N+1 discrete levels.

The study conducted in [3] tells that, using synapses with multiple MTJs, the precision of the NN increases; the same happens using a larger training set. However the first solution reaches a saturation faster than the second one, so it can be concluded that it is better to use a larger training set than increasing the number of devices for synapse. In figure 1.11 there is a graph with the data involved in this considerations. Moreover, the write operation is stochastic, so the learning phase can have

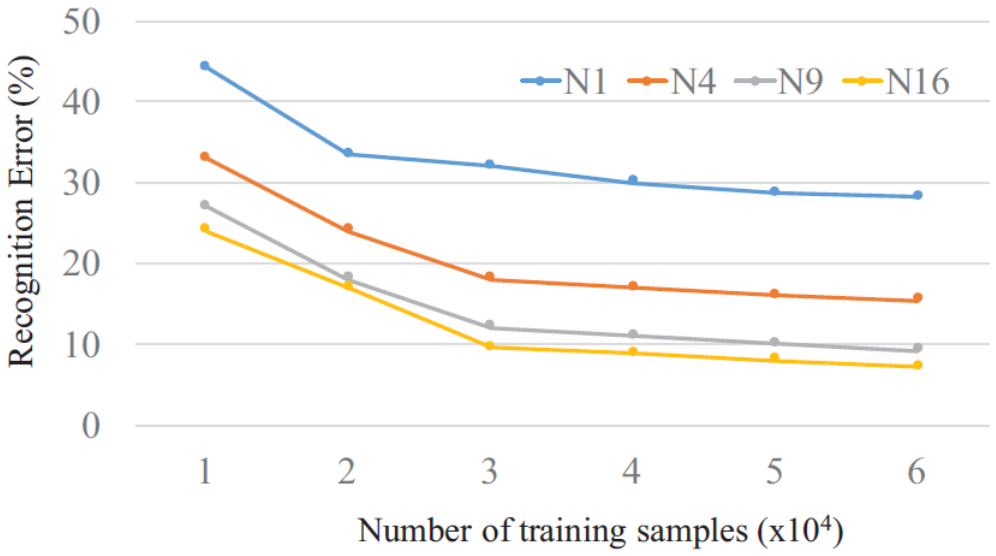


Figure 1.11: Recognition error of the SNN implemented as a function of the training set size, changing the synapse number of element. [3]

some problem due to the fact that not all the synapses are programmed correctly. Finally, it is important to consider the process variation in the fabrication process, because environmental conditions as temperature, pressure and humidity can change the behaviour of each MTJ device in the network respect to the others, causing a non-uniform response.

STT Magnetic Neuron for Low Power Neuromorphic Computing

The architecture described in [4] is made up by a hybrid structure, consisting on memristor synapses and MTJ neurons, as shown in figure 1.12. The structure implemented is a crossbar array; in this way the synaptic weight is associated to the

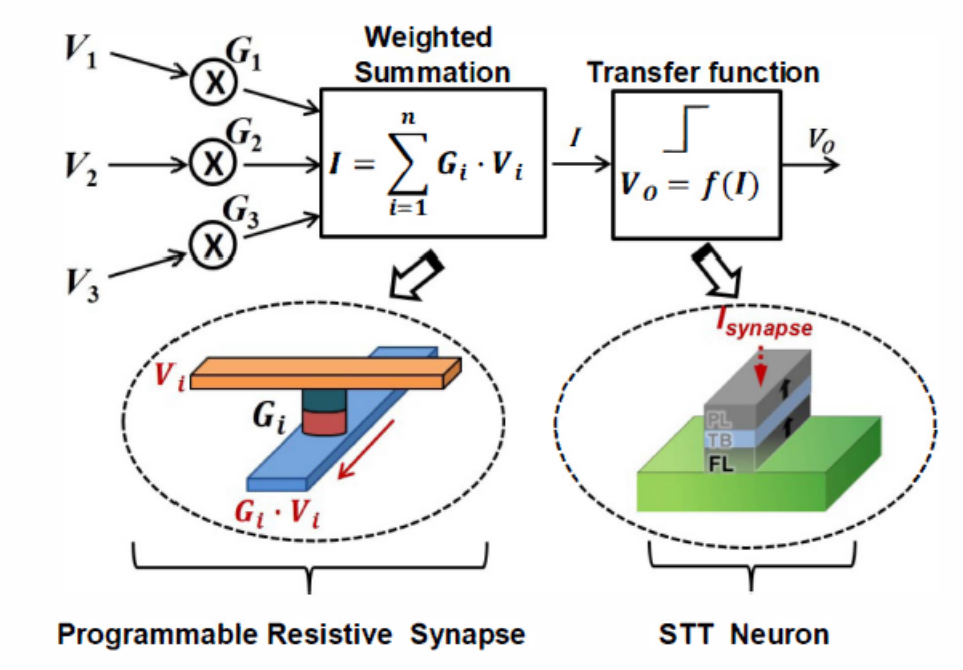


Figure 1.12: Synapses and neurons in the architecture proposed in [4].

conductance value at the cross point, where a memristor is placed. The neuron implements the transfer function using a couple of MTJs (one as reference and one for the neuron considered) to realize an input voltage divider for the threshold output amplifier, as represented in figure 1.13. The crossbar uses two rows for each input Vi to implement bipolar weights: when the input value is a logic '0', 0V are applied to both the lines; when the input is a logic '1', $+V_{dd}$ is applied to the positive row and $-V_{dd}$ to the negative one.

The use of memristors as synapses causes a bad dependence on the power consumption when the voltage supply changes: in a canonical synapse the relation is linear, in a memristor is quadratic. In order to solve this problem, that can vanish all the benefits introduced by MTJs on the energy consumed, it is added a bias row to the architecture, enabled by a PMOS transistor used as current source. In this way it is possible to split the evaluation operation in two phases: first the neuron is reset (the MTJ assumes the anti-parallel configuration), then the current is applied. This approach reduces the current of the synapses, so the energy-efficiency of the system increases.

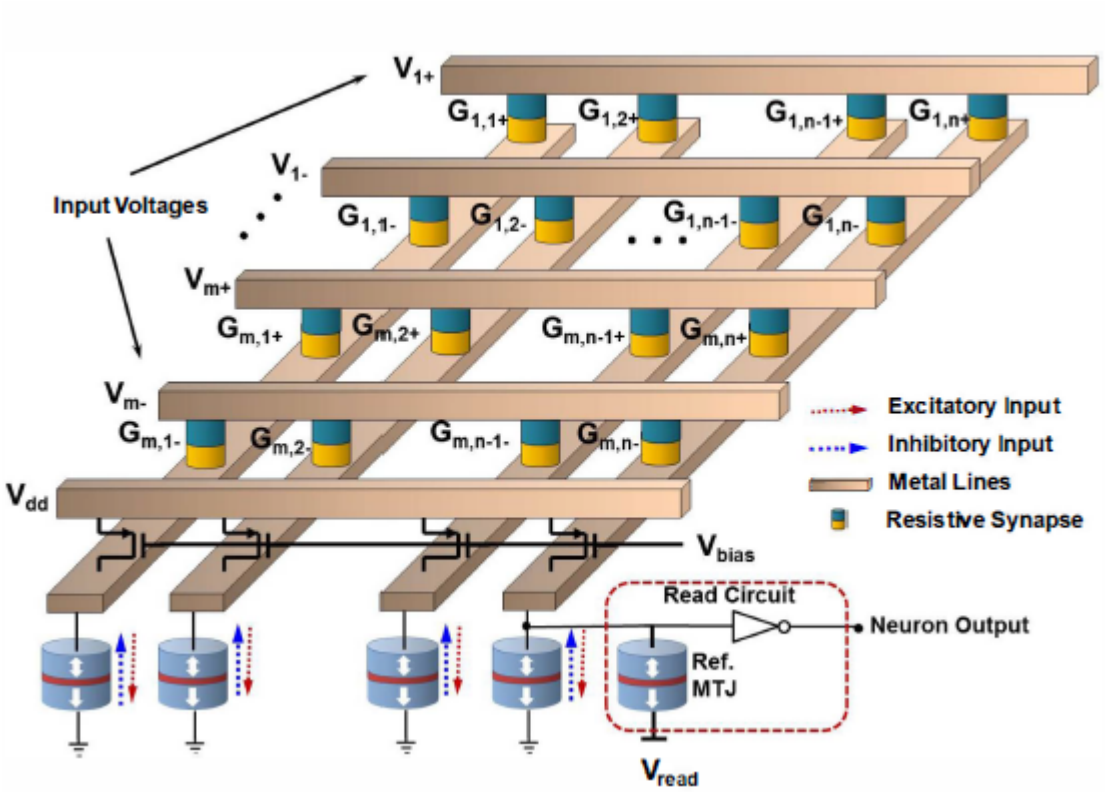


Figure 1.13: Neuromorphic architecture with memristors as synapses and STT-MTJ as neurons. [4]

The paper [4] compares this kind of architecture with a CMOS equivalent one, on which neurons and synapses are implemented with canonical registers, adders, multipliers and threshold units. The results, considering an iso-throughput implementation, is a better power consumption for the hybrid memristor/MTJ structure; the graph is reported in figure 1.14.

STT-RAM for Precision-Tunable General-Purpose NN Accelerator

MTJs are affected by process variations, that can cause different behaviour for the devices fabricated. This is a hard limit for the application of this technology in the conventional circuits; however in the approximate computing world this is not a problem, because the circuit realized has got a certain tolerance to errors and input noise, and process variations can be seen as the last one. Neural Network

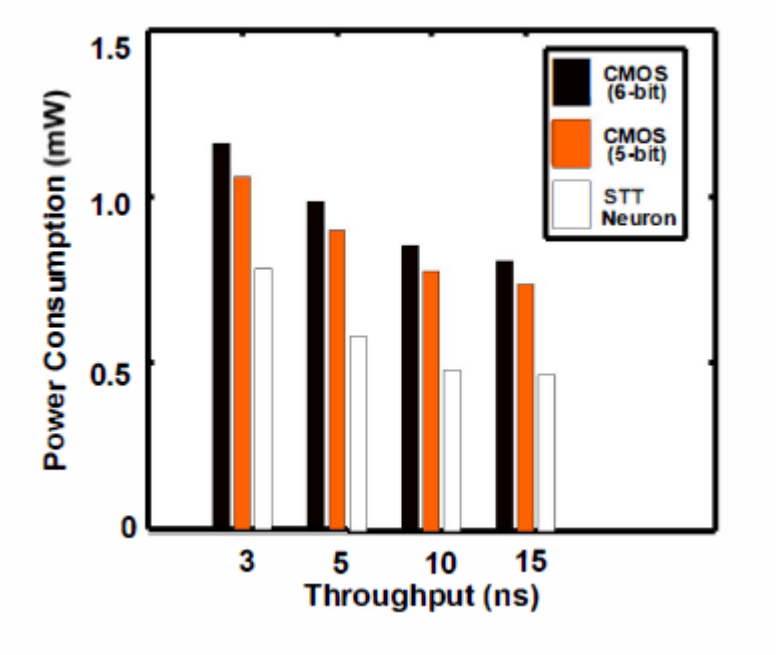


Figure 1.14: Iso-throughput comparison between the architecture proposed and a 45nm CMOS solution. A. Sengupta and K. Roy[4]

Accelerator (NNA) is a device that uses approximate computing, as all the NNs, to elaborate data, so the paper [5] proposes a way to use this characteristic to realize a more compact general purpose NNA.
To generate a compact design, the single cell is made up by two MTJs stacked, one large and the other small, in order to program and read them with the application of two distinguishable currents; the cell obtained is called Multi Level Cell (MLC). Due to the fact that one MTJ can store two values, with this structure is possible to store four values, so the layout is the half than a conventional one. However, the two MTJs are connected together, so they are forced to conduct the same current when a programming pattern is applied; this causes a two-steps write operation: first it is made a hard transition to switch both the MTJs, then a soft one to switch only the small. Also the read operation has to be in two steps, comparing the resistance with a first reference resistance (RR1) and, depending on the result of the comparison, with another one (RR2 or RR3). The description of this procedure is reported in figure 1.15. The using of this device provides a more compact layout, but it is slower than a traditional cell, because to read two bytes it is needed to access to the same

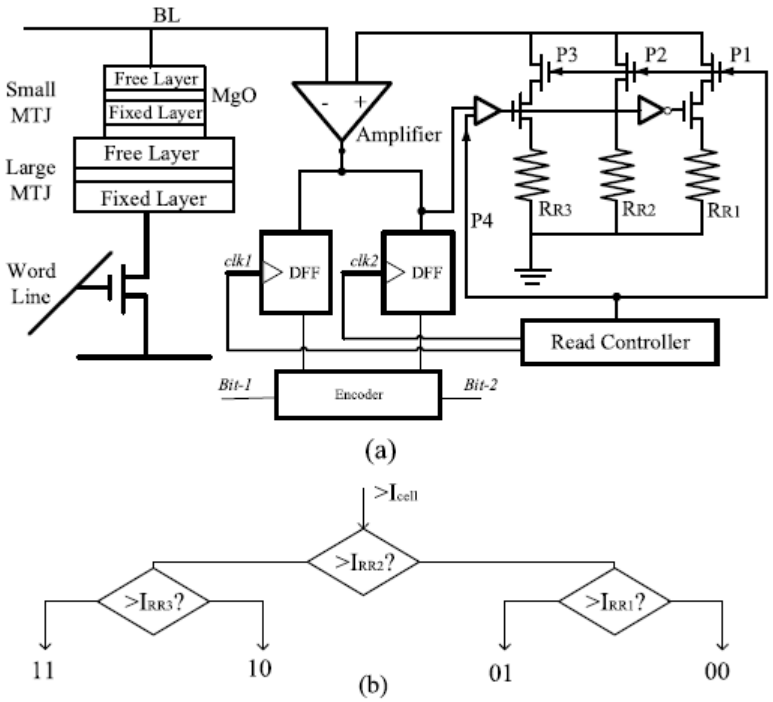


Figure 1.15: Description of MLC STT-RAM cell and its read procedure. (a) Structure of MLC. (b) Procedure of the MLC read operation.[5]

cell for two times. Moreover, the MLC memory sometimes can provide wrong results on the small MTJ due to the process variations and dynamic factors like thermal fluctuations during the access. These situations can be problems or not, depending on the application needed. For this reason, the MLC can also be used as a Single Level Cell (SLC), that is the conventional usage of a MTJ, or in a hybrid mode, that is the SR-MLC mode:

- MLC Mode: with the two MTJs is possible to store two bits (four resistance values), but two steps are needed to read and write the cell, and the reliability of the small bit information is low. In NNs it can be acceptable because the network accepts an input noise, but for other applications it is not a good result.

- SLC Mode: only the large MTJ is used, so only one step is needed to program and read the bit stored, providing better performance and reliability.

- SR-MLC Mode: this mode provides an intermediate behaviour between SLC and MLC: each two cell, three bits are stored, using the two small MTJs to store only one bit; in this way the reliability is improved, conserving the advantages on the compact layout.

These modes are represented in figure 1.16 and provide a very useful architecture for general purpose circuits, because it is possible to use it improving reliability, performance or to have a more compact layout. NNs use approximate computing,

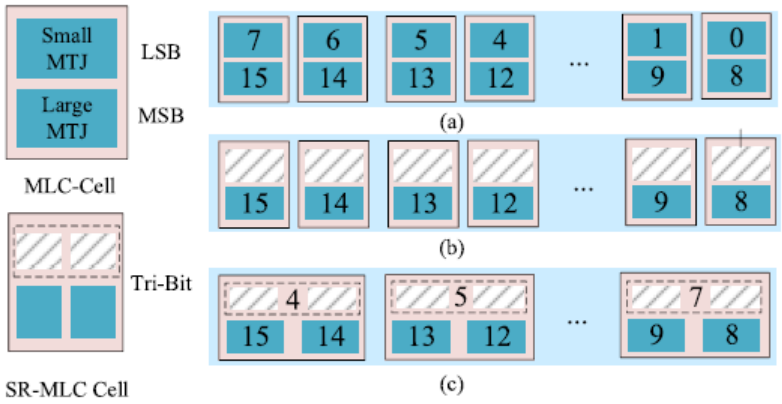


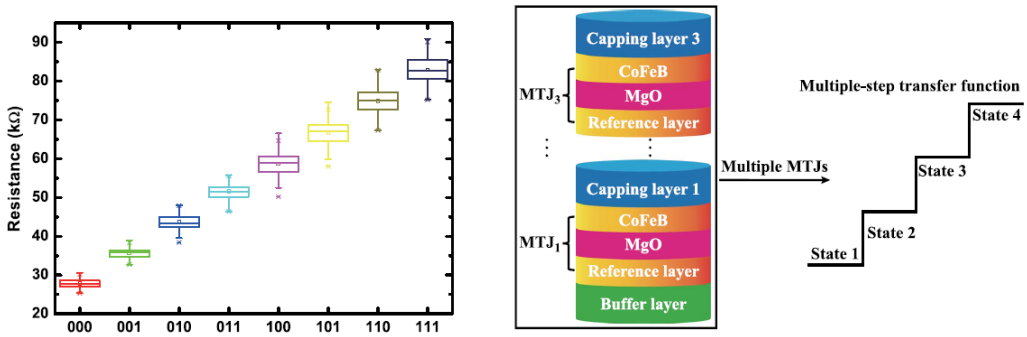
Figure 1.16: Data mapping in the reconfigurable MLC cell. (a) Bit mapping. (b) Low precision mode (SLC) mode. (c) Intermediate precision mode (SR-MLC mode). [5]

so there is a tolerance for the signals of each neuron. In general, the LSB is less useful than the MSB to generate a correct output, so, mapping the LSBs on the small MTJs, it is possible to obtain results that are similar or equal to the expected ones.

In order to make the architecture more general purpose, the datapath is divided into lanes, each one independent to the others, with the possibility to activate or deactivate part of the hardware, saving power. Also the multipliers implemented, that support a high parallelism, are made up by smaller multipliers, so it is possible to select the number of bits to make the computation and switch-off the part of the circuit that is not useful to provide it, improving energy savings. All this features make the hardware obtained a general purpose NNA, with the possibility to modify its behaviour in each moment.

ASANN based on CSS and CSN

The architectures analysed mainly use MTJs to have a binary behaviour. It is possible to obtain a MTJ device with multiple states stacking more of them. The component produced is called Compound Spintronic Synapse (CSS) if it is a synapse or Compound Spintronic Neuron (CSN) if it is a neuron. In paper [6] it is realized an All Spin ANN (ASANN), using CSN and CSS to implement a non-binary NN. In particular, a CSS made up by N MTJs is capable to assume 2^N discrete state; on the contrary, a CSN with the same number of element can have only N+1 states, due to the fact that it implements a step function, and the possible steps that can be made are N+1. Both the CSS and CSN behaviour are shown in figure 1.17. The



(a) CSS (b) CSN

Figure 1.17: (a) CSS with 8 discrete synaptic weights. (b) CSN that implements a multiple-step transfer function. [6]

work done in paper [6] is based on the same crossbar architecture used in [4] and implement the read, write and reset sequence described in [3], adapted to the ANN proposed. From the simulation results emerges that the number of MTJs that is possible to stack is limited due to the process variations: increasing the stack length, the accuracy decreases, but the ANN acquires higher input noise tolerance, so they act in two opposite directions. For this reason the result can be better or worse than the one obtained with a lower number of stacked element, depending on how much good is the technology used to realize the circuit.

PIM

A Processing In Memory solution is very useful when the architecture has a large number of elements, because data that are stored in memory are immediately available to the evaluation units. A structure of this type is appropriate in the case of NNs, when the quantity of neurons and synapses are very high, so the latency of the system could reach unacceptable values. The crossbar configurations seen previously are easy to implement in a PIM system, because they have a regular geometry, similar to the standard memory one. Another example of PIM is the Hybrid Memory Cube (HMC) [7], fabricated by Micron and made up by a 3D structure with a layer of CMOS logic on the bottom and more layers of DRAM on the top. The memory and the logic planes are divided into vaults to reduce the data latency, so each vertical column becomes a little datapath with a very close memory. This structure is shown in figure 1.18.

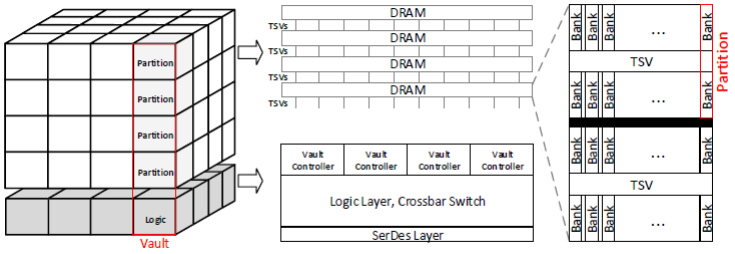


Figure 1.18: HMC Architecture. [7]

Neurocube

Neurocube is a programmable digital neuromorphic architecture based on the HMC [7], so it is a very dense PIM implementation of a NN. The HMC plane has a finite logic capacity, so each time is needed a NN with more layers, each one with a large number of neuron and synapses, it is better to evaluate only one layer, store the results and evaluate the next one, iterating the process, as in figure 1.19a. This approach provides a programmable and flexible way to realize NN, but the main disadvantage is the time spent to store output data, load a new layer on the HMC plane and apply inputs. However, theoretically is possible to implement each kind

of NN, independently from the complexity and the number of layers used. The Neu-

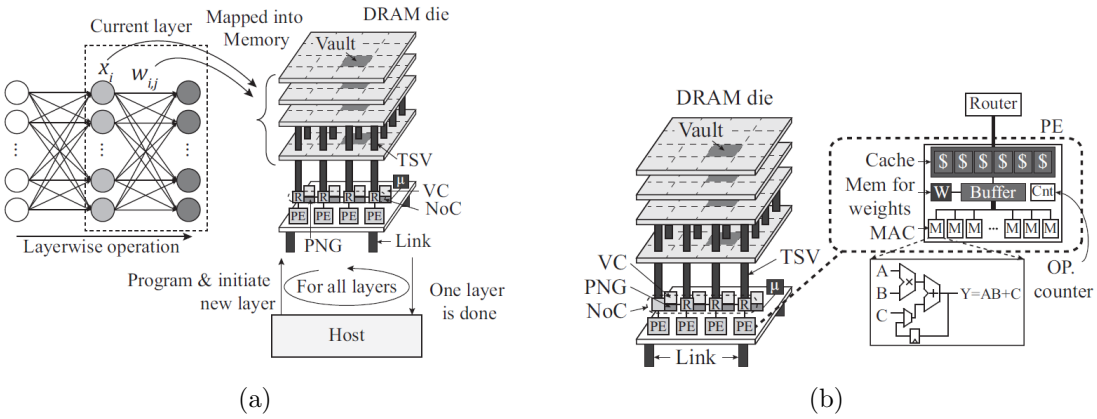


Figure 1.19: (a) Programming Neurocube Flow. (b) Neurocube Architecture. [8]

rocube is composed of a global controller, a programmable neurosequence generator (PNG) for DRAM, routers for a 2D-mesh network on chip (NoC), Processing Elements (PEs) and Vault Controllers (VCs) to manage the communication between PEs and the DRAM vaults. An external host can decide the parameters of the network programming the HMC with an external link. The implementation of a generic network is shown in figure 1.19b, on which PEs are made up by Multiply and ACcumulator blocks (MACs), a cache memory, a temporal buffer and a memory for the synaptic weights.

RTNN: Real-Time scheduling technique for convolution NN on 3D neuromorphic PIM architecture

The most important problem of the Neurocube [8] is the time to store data, load a new layer and apply inputs. The paper [9] proposes a scheduling technique to optimize the execution of CNN layers, obtaining an increasing of performance respect on the traditional Neurocube.

A CNN is made up by several deterministic processing procedure, each one associated to a functionality that can be subdivided into a certain number of tasks. The consequence is that it is possible to represent a CNN application with a data flow graph (DFG): each task represents a computation operation, such as convolution or pooling operation. For a computation task, the intermediate processing results

have to be synchronized in order to produce the correct result for the next task, so it is needed a rescheduling using a retiming function. The proposed technique is applied in a static way, because the scheduling is known after the training of the network; using this approach it is possible to not introduce more latency, so the benefits obtained are the maximum possible.
The scheduling algorithm uses the PE utilization ratio (U) to establish when the result obtained is good and the procedure can stop. The definition of the utilization ratio is reported in equation 1.1: e_i is the execution time for the task T_i, $\hat{p}$ is the number of steps required to complete one tasks set, N_{set} is the maximum number of tasks set allocable in $\hat{p}$ and M is number of PE available.

$$U = \frac{\sum_{i=1}^{n} e_i \cdot N_{set}}{\hat{p} \cdot M} \tag{1.1}$$

If the denominator of the equation is lower than the numerator it means that the utilization ratio is lower than the maximum one; the scope is to reach a value that is the closest possible to it. The process can be stopped when is reached a threshold, that assumes a predefined value set to the minimum utilization ratio acceptable.
In figure 1.20 are shown the steps to reach the maximum U possible considering the DFG shown in figure 1.20a, a threshold for U set to 0.92, a buffer budget of 9 and a buffer requirement for the tasks that is T_1=2, T_2=4, T_3=4, T_4=2, $T_{5]=2,T_6=1}$:

- the sum of the execution time is 9, so at most 2 sets of tasks can be allocated, otherwise the denominator becomes higher than the numerator. The utilization ratio obtained is 0.75; in order to preserve the data dependencies and remove synchronization overhead, the second set is released at time unit 4. In this way, $\hat{p}$ becomes 10, so it is possible to increase N_{set} to 4, obtaining U=0.9. According to this, another set is added, ending the operations at the time unit 14, reaching a U=0.96, higher than the threshold, so the process can be stopped. This situation is represented in figure 1.20b.

- due to the buffer requirement of the tasks, the remaining buffer budget is represented in figure 1.20c. The second step consists to allocate the remaining tasks, without violating the buffer budget, starting from the tasks that requires lower time units. The result is shown in figure 1.20d. The remained tasks

aren't allocable because of the buffer budget, so they are treated as infeasible tasks, generating the final scheduling represented in figure 1.20e.

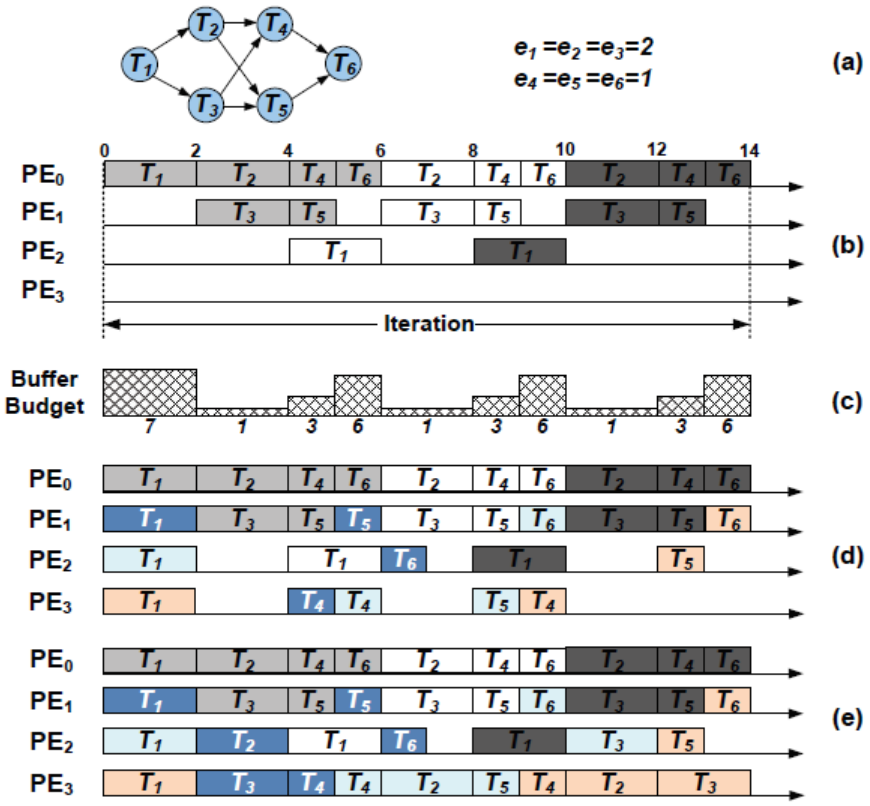


Figure 1.20: Generation of the scheduling. [9]

The conclusion about this scheduling algorithm is that the time used to run it is higher than the one used by other equivalent solutions, but the advantages in terms of scheduling optimization can be consistent, depending on the network implemented.

XNOR-POP

Paper [10] proposes an architecture capable to satisfy the constraints imposed by mobile devices, as the limited hardware resources and the low power budget.
The solution proposed is a CNN that uses the XNOR operation instead of the MAC one, called XNOR-Net; in this way is possible to reduce consistently the hardware,

because XNOR are very small logic gates, and also the power efficiency increases. The problem that can emerge by this circuit is that the results of a XNOR is only approximately the same of what a MAC provides, so the circuit is affected by a form of input noise, caused by this lost precision.
The XNOR-Net uses a stacked DRAM memory developed for mobile devices, called WideIO2 memory, represented in figure 1.21.

Figure 1.22 describes the behaviour of the network: each row of the DRAM is

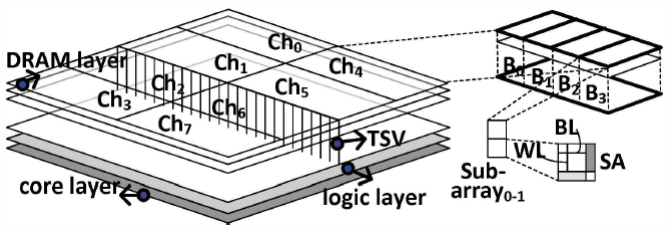


Figure 1.21: Wide IO2 DRAM architecture. [10]

associated to a block that provides the XNOR operation, then the result is latched to allow the starting of a new evaluation; the result goes through TSVs to the logic die, on which is processed with a population count, a pooling stage and finally reaches the Layer Output Buffer (LOB), implemented with a SRAM. When the computation is finished or the LOB is full, the pipeline stops and the output data are rewritten to the DRAM dies.

The results obtained by the implementations of different NNs, comparing with

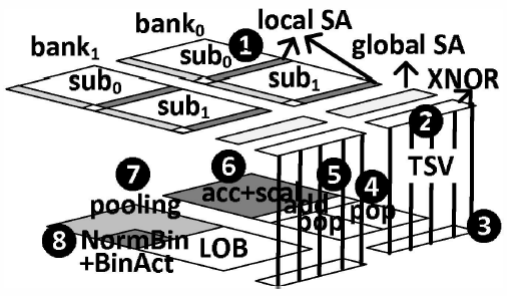


Figure 1.22: XNOR-POP flow. [10]

other accelerators, show that the performance of the XNOR-Net is about 10 times better and the power consumption is the 90%. However the disadvantage of this

solution is the accuracy reduction, but NNs are designed to tolerate some errors and noise, so, if the NN developed is robust to these negative effects, the architecture provided becomes better than the other traditional solutions.

Many-core

In order to implement a big NN, it is possible to implement an architecture made up by many neural cores, connected together to create a unique larger network. The main disadvantages of this solution are a reduction of performance when more chips communicates together and an increasing of the area occupied, because the network developed is distributed on more chips or PE. However there is an important advantage: the possibility to expand the NN, scaling up the number of neurons, layers and increasing the precision, without big constraints.

Loihi

Paper [13] presents a neuromorphic manycore processor fabricated by Intel to implement a SNN in a 14nm CMOS technology. The processor is made up by:

- 128 Neuromorphic cores, each one implementing 1024 neural units, grouped in sets that share configuration and state variables, updated each algorithmic step;
- 3 embedded x86 processor cores;
- Off-chip communication interface that hierarchically extend the mesh in 4 planar directions to other chips;
- Asynchronous NoC to manage communication between cores.

The features implemented are several; the main ones are the possibility to add stochastic noise to a neuron synaptic response, the configurable and adaptable synaptic, axon and refractory delays, the configurable dendritic tree processing, the neuron threshold adaptation in support of intrinsic excitability homeostasis, the scaling and saturation of synaptic weights in support of “permanence” levels that exceed the range of weights used during inference.

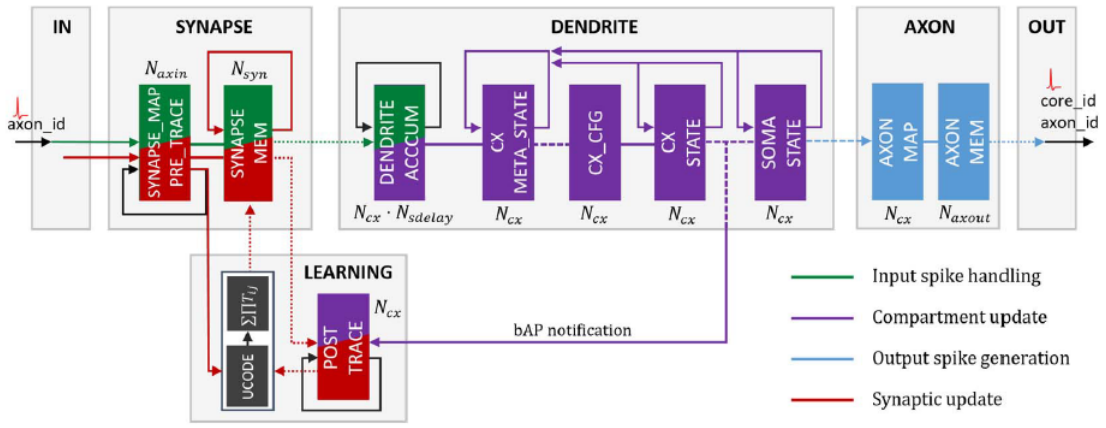


Figure 1.23: Loihi core architecture.[13]

The architecture of a single core is drawn in figure 1.23, on which there are several blocks:

- Synapse unit processes all incoming spikes and reads out the associated synaptic weights from the memory;
- Dendrite unit updates the state variables u and v of all neurons in the core;
- Axon unit generates spike messages for all fanout cores of each firing neuron;
- Learning unit updates synaptic weights using the programmed learning rules.

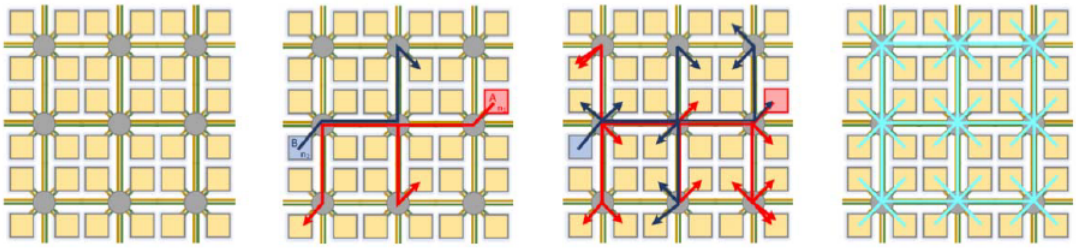


Figure 1.24: Mesh operation: first box, initial state for time-step t; second box, neurons n_1 and n_2 in cores A and B generate spikes; third box, spikes from all other neurons firing on time-step t in cores A and B are distributed to their destination cores; fourth box, each core advances its algorithmic steps to t+1. [13]

All the cores are put in a mesh structure on which they can exchange spikes. The communication is completely asynchronous and event driven, with a synchronization

barrier: when a core receives it, the communication with the near cores starts, propagating the spikes received to the internal neurons; after that phase, all the cores increase the own time-step of one unit. This asynchronous behaviour is possible due to the fact that in a SNN, inside a single time-step, the result doesn't change with the order of the received spikes. The detailed representation of the mesh operation is reported in figure 1.24.

DaDianNao

The purpose of the paper [14] is to realize a structure capable to achieve high sustained machine-learning performance. The architecture proposed is called DaDianNao and is a multi-chip system made up replicating the DianNao accelerator, shows in figure 1.25. The "supercomputer" obtained is compared to a GPU-based

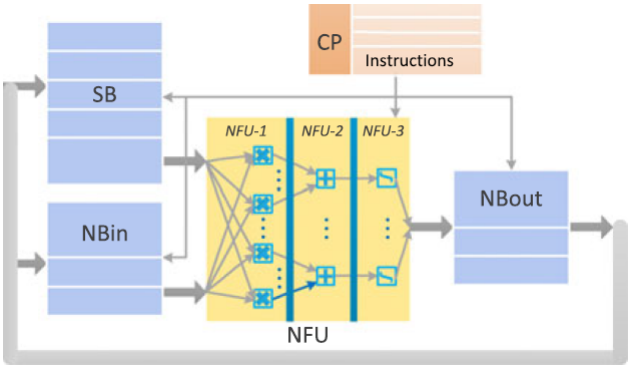


Figure 1.25: Block Diagram of the DianNao accelerator. [14]

approach in order to demonstrate that is possible to implement a NN with comparable or higher compute density (number of operations per second divided by the area) in a cheaper and more compact way.
To realize this purpose the entire structure is divided into tiles: each node contains 16 tiles and two central eDRAM banks; a tile contains a Neuron Functional Unit (NFU), four eDRAM banks and I/O interfaces to/from the central eDRAM banks. This structure is shown in figure 1.26.

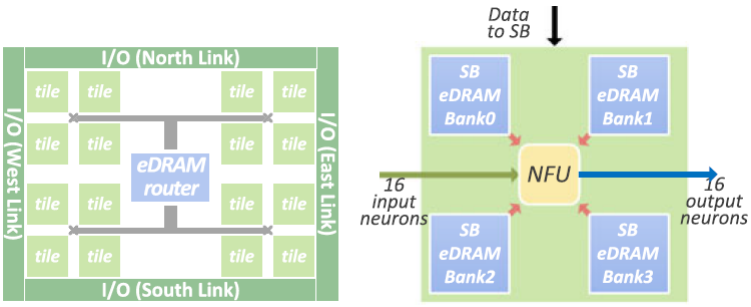


Figure 1.26: Tile-based organization of a node (left) and tile architecture (right). [14]

SpiNNaker

The paper [15] presents the SpiNNaker, a massively parallel computing engine designed to make easier the modelling and simulation of large-scale SNN of up to a billion neurons and a trillion synapses. It is made up by several nodes, each one equipped with a Chip Multi Processor (CMP), that is an ARM968 capable to simulate more than 1000 neurons, and a 128MB SDRAM.

The chip architecture (figure 1.27) is composed of synchronous islands containing 18 ARM968, surrounded by an asynchronous packet-switched communication infrastructure, that produces a Global Asynchronous Local Synchronous (GALS) system. This GALS makes local performance the highest possible. Moreover, all the chips can communicate with the external world with a 100-Mbit Ethernet interface.

TrueNorth

The TrueNorth architecture [16] is a big multi-chip architecture with many-core chips; each core implements 256 programmable neurons, 256 axons and 256*256 synapses, realized with a SRAM. The system take inspiration from the Macaque brain and it is implemented with crossbar arrays. It consists of several blocks, also shown in figure 1.28:

- A SRAM memory stores all the data for each neuron, a time-multiplexed neuron circuit updates neuron membrane potentials;
- A scheduler buffers incoming spike events to implement axonal delays;

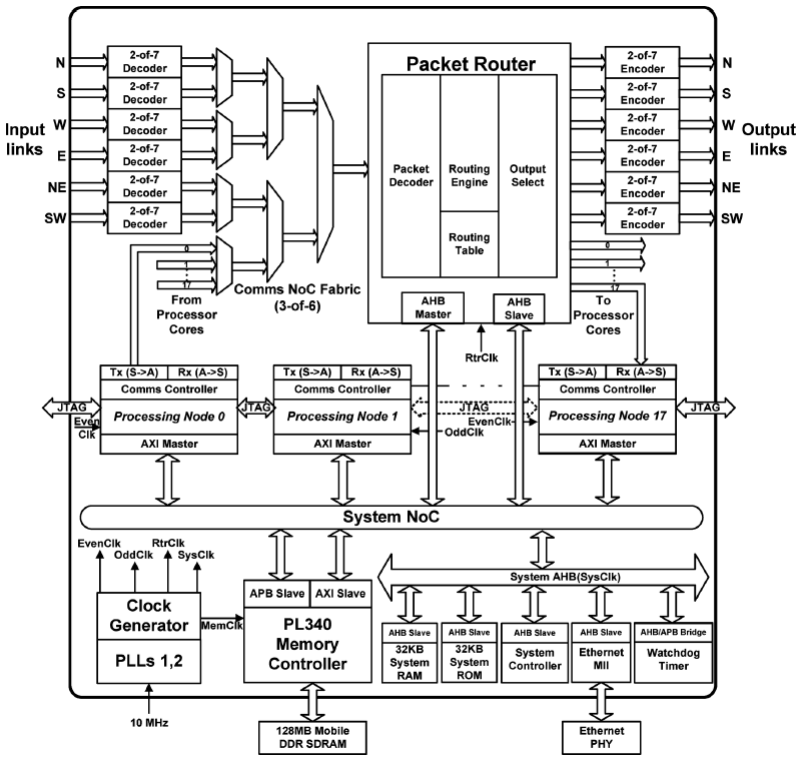


Figure 1.27: Organization of the SpiNNaker [15]

- A router relays spike events;
- An event-driven controller orchestrates the core operation.

The authors choose a NN to demonstrate how the architecture works: it is a NN that divides an input image into two other images, one with low resolution and another with high resolution. The first one is used with a part of NN to classify where it is detected a recognizable object (in the example proposed the network can detect only motorbikes and cars), the second one classifies what was detected.

CMOS

The CMOS technology is the oldest one, so it is not an emerging technology, but it is very important because the fabrication of a CMOS chip is simple and cheaper than a chip realized in other technologies. Respect on MTJs and memristors, the CMOS solutions are less compact in terms of layout and can have a higher power

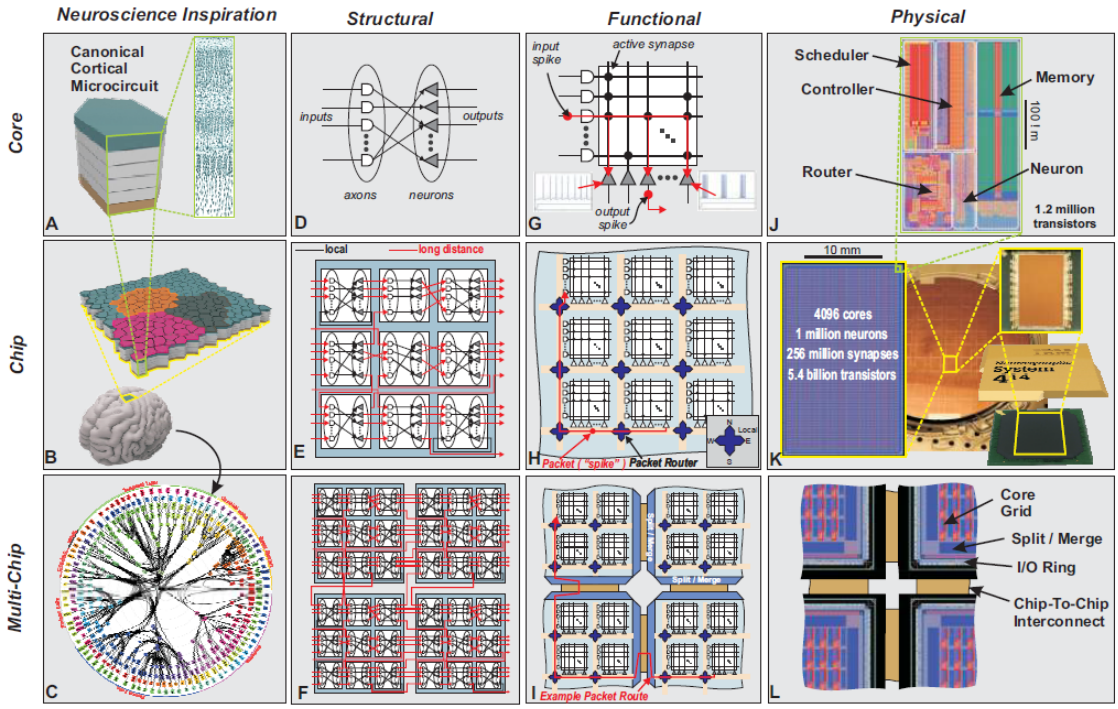


Figure 1.28: TrueNorth architecture, analysing in details multi-chip, chip and core level to the neuroscience, structural, functional and physical point of view. [16]

dissipation, due to the non-negligible leakage current. This is the reason why the research is moving from CMOS to other technologies, searching the ones that offer the best compromise between performance, power dissipation and area.
However, the using of CMOS remains important because it can be designed, simulated and fabricated without big problems, and a CMOS-circuit can be a starting point to make evaluations on how much good are other equivalent circuits produced. Moreover, it is one of the technologies that produce higher performance, so the big challenge is, after having selected the CMOS reference circuit, to produce an equivalent solution, with comparable performance and better compactness and/or power dissipation.

Embedded Crossbar Memory in a Digital Neurosynaptic Core

The purpose of [17] is to realize a neuromorphic core minimizing the active power consumption, meeting or exceeding real-time performance and neglecting the density

optimizations. The way choose to reduce the power is to update neurons in an event-driven manner, realizing asynchronous communications between blocks.

The architecture shows in figure 1.29 is made up by an input decoder with 1024

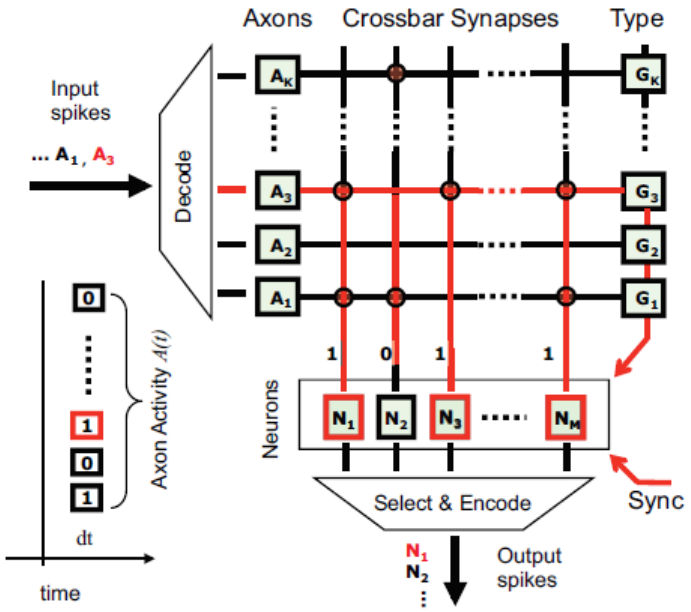


Figure 1.29: Internal blocks of the core include axons (A), crossbar synapses implemented with SRAM, axon types, that can be excitatory or inhibitory (G), and neurons (N). An incoming address event activates axon 3, which reads out that axon connections, and results in updates for neurons 1, 3 and M. [17]

axons, a 1024x256 SRAM crossbar, 256 neurons, and an output decoder.
The working principle of the circuit is based on two phases for each time-step:

- first phase: address-event are sent to the core one at a time as spikes that are sequentially decoded to the appropriate axon block. The axon activate the SRAM row on which it is connected; the row reads all the axon connections of its type, sending to the respective neurons only the ones that exist (they are classified with '1' if there is the connection, '0' otherwise). Neurons then update the membrane potential; when they end this operation, the axon block deselects the row, so a new address-event for the current time-step can be processed.

- second phase: each millisecond a synchronization event is sent to all the neurons, that check if its membrane potential is higher than a threshold and, in positive case, generate a spike, then reset the membrane potential to 0. Finally, spikes are sent as a sequence of address-event.

The behaviour so obtained consists on a set of neurons that accumulate received spikes increasing their membrane potential and synchronise the generation of output sparks each millisecond using a special barrier event. This behaviour doesn't match the real brain working principle for the second phase, because in the brain there isn't this form of synchronization; however, the authors decided to use this approach in order to ensure that the hardware and the software used were always in lock step at the end of each time-step.

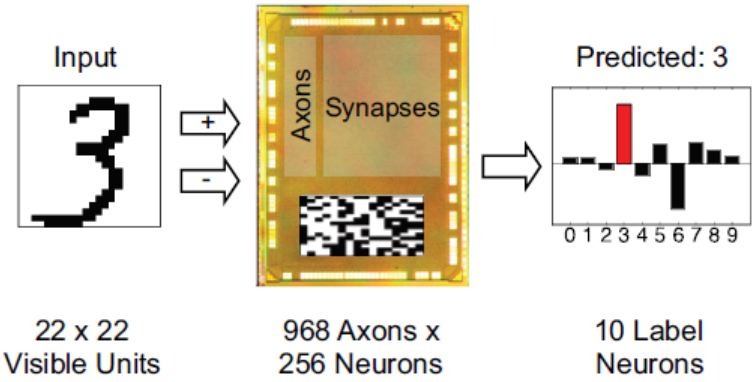


Figure 1.30: (left) Input figure to be classified. (middle) 16 x 16 grid of neurons spike in response to the digit stimulus. Spikes are indicated as black squares and encode the digit as a set of features. (right) An off-chip linear classifier trained on the features, and the resulting activation. Here, the classifier predicts that 3 is the most likely digit, whereas 6 is the least likely.[17]

An example of how the network works is reported in figure 1.30, on which the prediction of the input figure tells that '3' is the most probable number represented. The circuit realized is capable to implement a large number of NNs; the procedure to make the learning is an off-line training, then the weights are transformed in a hardware compatible format, using the type axon properties, that can be excitatory (+) or inhibitory (-).

ROLLS

The paper [18] presents the ROLLS (Reconfigurable On-Line Learning Spiking) neuromorphic processor, a device that can be used both for research experiment in computational neuroscience and to develop application solutions as NNs. This is possible because it implements analogical and digital circuit to emulate exactly the behaviour of a brain; in this way, all the parts that compose a brain can be replicated and simulated by the circuit, obtaining a good platform for neuroscience researches. At the same time, neurons and synapses developed use the same configuration and behaviour of the NNs, so it is possible to realize a generic neural network on it, using also the digital circuits that allow the chip to communicate with the external world. The architecture in figure 1.31 comprises analogical synapses that produce

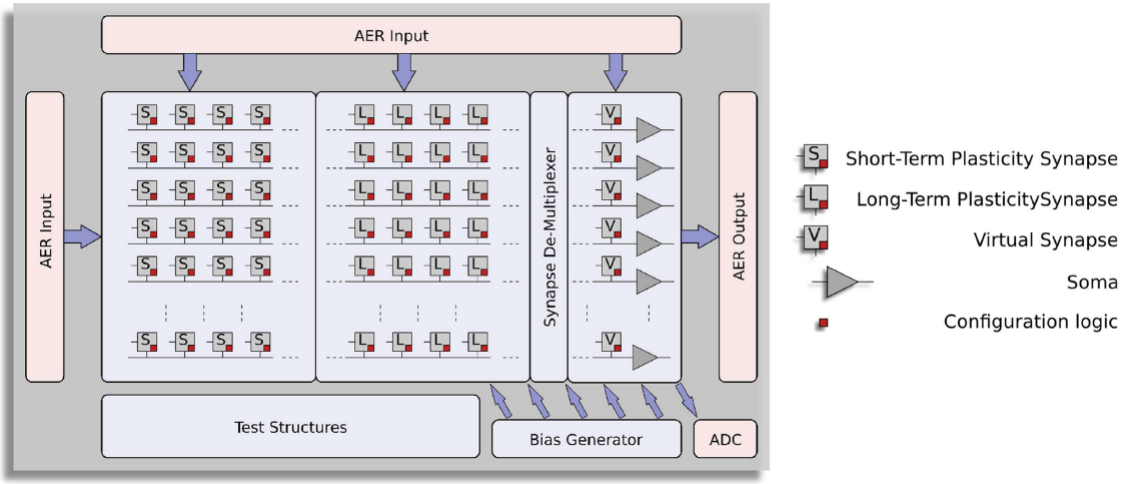


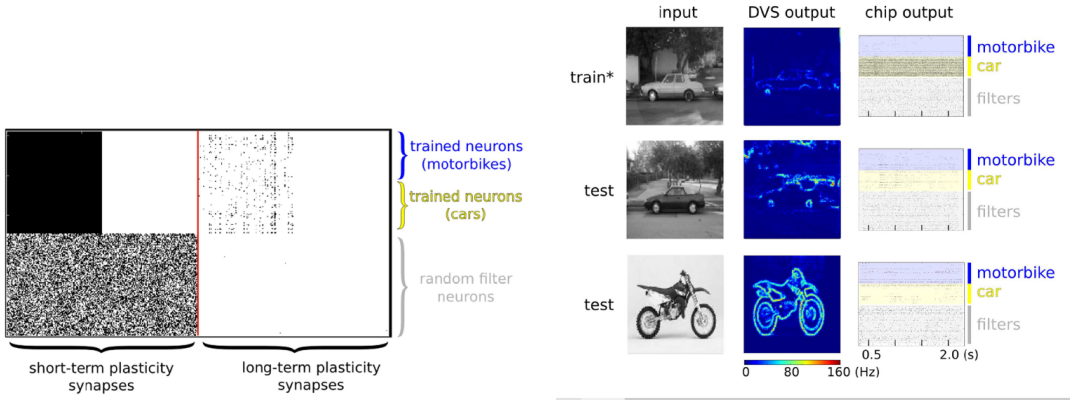
Figure 1.31: ROLLS architecture., with the two synapses arrays (emulating the long-term and short-term synapses of a brain), an additional row of synapses (the virtual ones) and a row of neurons (somas). There are peripheral circuits as asynchronous digital AER (Addres Event Representation) blocks, ADCs and a programmable on-chip bias generator. [18]

biologically realistic response properties and spiking neurons that can exhibit several realistic behaviours; these kinds of neurons are very good to implement also SNNs, because they work already with spikes.

The circuit implements 256 neurons, an array of 256x256 learning synapses to model the long-term plasticity mechanism, another array of 256x256 programmable

synapses to model the short-term plasticity, 256x2 row of linear integrator filters that are "virtual synapses" to model excitatory and inhibitory behaviours and a group of peripheral I/O to communicate with the external world.

The authors show an example of NN implementation on the ROLLS chip (figure



(a) (b)

Figure 1.32: (a) Neurons in the NN implemented to recognize only motorbikes and cars; only the top half part of the neurons is used because it is sufficient to demonstrate the right behaviour of the NN. [18]

1.32) that can distinguish if the input image contains a car or a motorbike. After the training and using only the half of the neurons available, the network produces right results.

A Neuromorphic Event-Based Neural Recording System using ROLLS

Paper [19] proposes a possible usage of the ROLLS architecture; in particular, it wants to demonstrate that is possible to convert biological audio signals into asynchronous digital events to be applied on a low-power SNN.

To apply real biological signals, a group of birds were anaesthetized and exposed to two different songs, so the data were measured from electrodes applied on them. The songs selected were two: the bird's own song (BOS) and a reversed version of it (REV). The SNN was tested applying different input sets; the network produced the main frequency of the song as output, then determined if it was a valid song or not applying a threshold. The results, shown in figure 1.33, have produced an

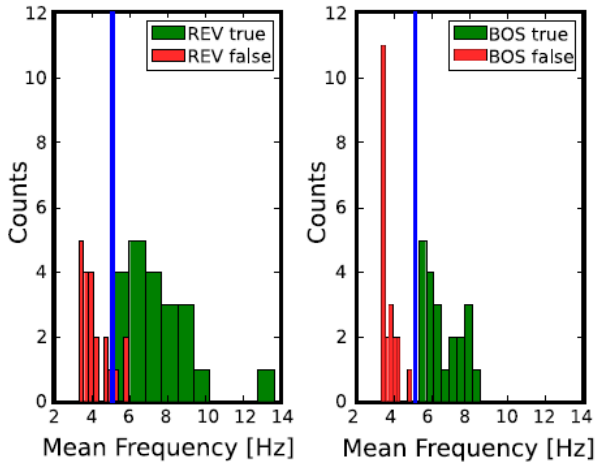


Figure 1.33: Results of the application of BOS and REV songs to the SNN implemented. [19]

error only in three cases of the REV song recognition, generating a false positive, obtaining an accuracy near 96%.

HICANN

HICANN (High Input Count Analog Neural Network) [20] is a chip that mixes analogical and digital circuits to realize a network made up by realistic neurons that can be connected together with other chips of the same type to expand the computational capability of the system.

Figure 1.34 shows the Analog Network Core (ANC) structure, on which the neu-

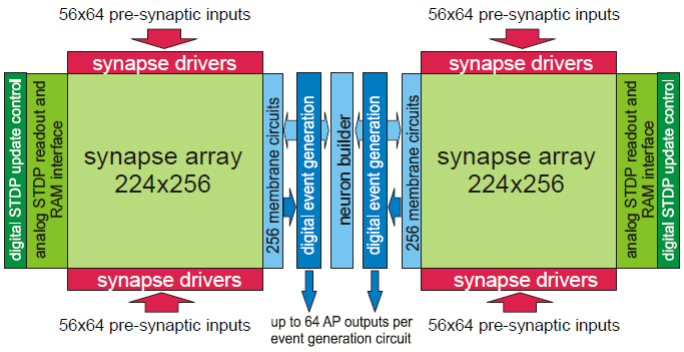


Figure 1.34: Schematic diagram of the ANC. [20]

rons are integrated with their synapses. To allow the possibility to have a variable number of synapse for all neurons, they are divided into several parts, called dendrite membranes (DenMem), each one connected to 224 synapses. A circuit named neuron builder combines together neurons group of DenMem, that are configurable in 23 analog parameter inputs generated by analog memory cells placed between the DenMem circuit and the neuron builder. Figure 1.35 shows how a synapse is made

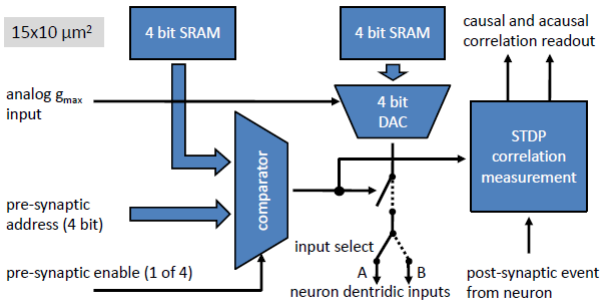


Figure 1.35: Schematic diagram of a synapse. [20]

up: it contains a four bit address decoder and it is connected to one out of four pre-synaptic enable signals, so each synapse in a certain column can receive 64 different input combination, but shared with the adjacent column. The synapse weight is realized with a current generated by a four bit multiplying DAC, that converts in an analogical signal the digital weight stored into a four bit SRAM; there is also an analogical input to control the maximum conductance (g_{max}) that a column of synapses can assume. Two adjacent columns can combine their synapses to reach a weight high resolution, due to the fact that they have the same inputs and is possible to program the g_{max} to be a fixed multiple of each other.
The current that exits to the synapse is transmitted to the neuron through the DenMem circuit, and it is encoded as a current pulse of defined length and an amplitude proportional to the synapse weight.

BrainScaleS

The purpose of [21] is to demonstrate the successful training of an analog neuromorphic system configured to implement a DNN. For this reason, the authors choose to

realize the network with a large number of HICANN [20], that implement analog neurons.

The BrainScaleS system (figure 1.36) consists of five cabinets, each containing

Figure 1.36: BrainScaleS system. [21]

four neuromorphic wafer-scale systems. Red cables shown in figure 1.36 provide the connectivity with a Gigabit speed. An additional rack hosts the infrastructures as power supplies, servers, control users and the network equipment.

Figure 1.37 shows the BrainScaleS wafer module, that is a silicon wafer with 384 HICANN chips. It comprises 48 reticles, each containing 8 HICANNs; each of them implements 512 neurons.

The network implemented is a feed-forward directed graph, with an input layer of 100 units used to represent the input pattern. Between the input and label layer there are two hidden layers composed by 15 units, and the weights are learned during several phases of training.

Figure 1.38 shows the results produced by the NN implemented after the training: on the left there are the pattern in input applied at different time, in the middle it

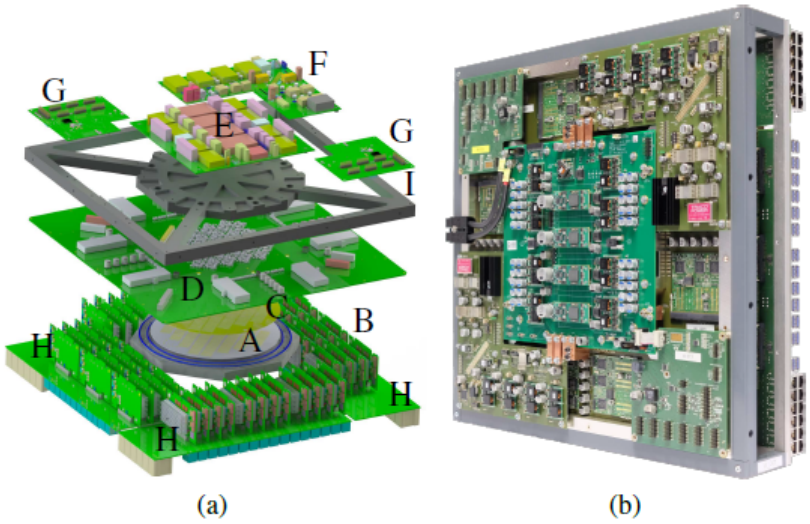


Figure 1.37: (a) BrainScaleS wafer module. A: wafer. B:48 FPGAs. C: positioning mask to align connectors from the wafer to the PCB. D: large main PCB. E,F: power supply. G: access to analog dynamic variables. H: edges on which there are connectors for inter-wafer and off-wafer/hosts communications. I: aluminium frame. (b) Fully assembled module. [21]

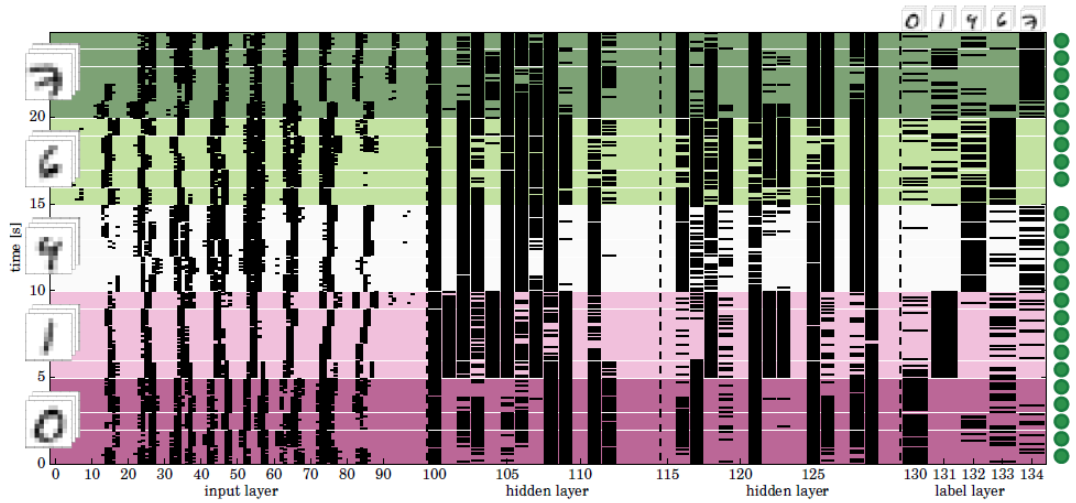


Figure 1.38: BrainScaleS results.[21]

is represented in black the neuron activities for the different layers and on the right there are the results of the classification layer.

Neurogrid

In order to realize a neuromorphic hardware architecture that is the closest possible to the real brain one, paper [22] proposes a completely analogical circuit, emulating all the neural elements except the soma with shared circuits. The usage of analogical circuits provides an optimization of the energy efficiency; another important parameter is made up by the performance of the circuit, so the interconnections between the neural arrays are organized in a tree network, maximizing the throughput. The

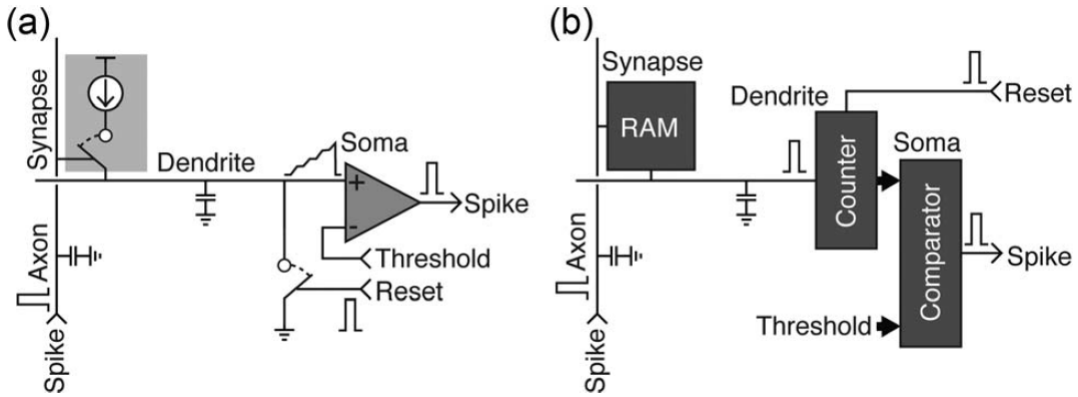


Figure 1.39: (a) Analog implementation of a neuron: spikes go through the axon reaching the synapse, that modulates the charges flowing through the dendrite, whose capacitance integrates them. The comparator (soma) compare the signal obtained with a threshold; if it is exceeded, a spike is produced and the capacitance is discharged (with the reset) to restart the cycle. (b) Classical digital implementation of a neuron, composed by a RAM, a counter and a comparator. [22]

Neurogrid project searches to reduce the number of transistors, sharing synapse and dendritic tree circuits; each Neurogrid board is divided into 16 Neurocores, each one capable to simulate up to 256x256 neurons.

The neural network elements can be implemented in a digital or analog way, as represented in figure 1.39. The approach adopted is the analog one for the reasons examined, but all the elementary blocks (axon, synapses, dendrites, somas) are implemented in a dedicated circuit, in order to emulate in the best way their real behaviours. To make easier the usage of this board, the project contains also a Graphic User Interface (GUI) to enable a user to change the model parameters, view spikes activity into the layers of the network implemented, plot spikes raster

from a selected layer and enter commands.

The performance, power and size results are reported in figure 1.40, on which there

	A (μm^2)	E (pJ)†	T (ps)	$A{\cdot}E{\cdot}T$	S	Synapse Spec
HICANN	436	198	2.9	250K	224	4-bit plastic
GoldenGate	256	4.0K	29.5	30.6M	1024	1-bit dedicated
Neurogrid	0.63	119	62.5	4.69K	4096	13-bit shared

† E includes dynamic and static power (for a 10 spike/s/neuron simulation).

Figure 1.40: Comparison between Neurogrid and other equivalent hardware solutions. [22]

is a table to compare the Neurogrid architecture with other equivalent projects, as the BrainScaleS [21] and the GoldenGate, an IBM project with the same purpose of the Neurogrid one. As shown, the circuit proposed has a more compact layout and consumes less power than the others, but with higher latency; however, analysing a general parameter, as the product of A,E and T, the best solution is the Neurogrid.

Neuromorphic Accelerator for Autonomous Robots

The paper [23] focuses on one of the possible application of a NN: the robotics. In particular, the purpose of the project is to provide a hardware architecture capable to recognize and classify inputs coming from sensors; in this way, a micro-robot can move autonomously in the space, without needing a remote driver. These kind of robots could be applied each time is needed to reach a place on which human people can't go, for example in natural disasters as earthquake, or simply for reconnaissance. A prototype of the robot is shown in figure 1.41, on which are indicated sensors, motors and the board.

In figure 1.42 is represented the general architecture of the circuit realized: the ultrasonic sensors provide pulses directly to neurons through an array of synapses, that realize a time-to-digital conversion to provide valid data. These hidden layer neurons make a weighted sum of the inputs and produce a pulsed output sent to the output neurons, that, after a Winner Take All (WTA) operation, produce an action to move the robot straight, to the left or to the right. The feedback circuit predicts

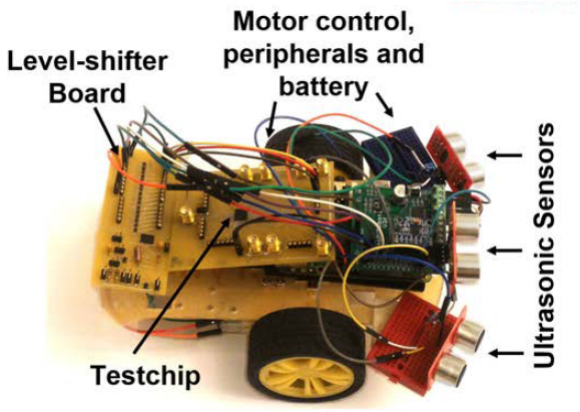


Figure 1.41: Micro-robot for which the NN is developed. [23]

the reward (i.e. avoid obstacles and cover maximum distance for area mapping), compute the loss function and updates the model (synaptic weights) for further explorations. In order to save power, the MAC operation is implemented in a time-

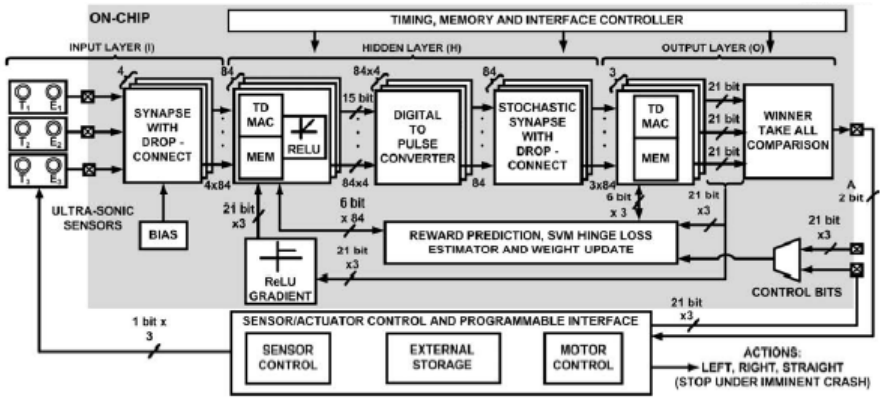


Figure 1.42: General architecture of the chip. [23]

way with a 21 bit counter, which multiplies the 6 bit input (x) from a pre-synaptic neuron to the 6 bit weight (w) of the synapse. The input is a pulse of width $T=x\cdot T_0$, generated by a digital to pulse converter; a Digitally Controlled Oscillator generates a frequency $F=w\cdot F_0$ and clock up/down the counter, evaluating $T\cdot F=x\cdot w$. The usage of the counter makes easily the implementation of negative weights, because it is sufficient to downcount to implement it.

1.3 Comparison between technologies

In order to compare the solution analysed, it is needed a parameter that is independent to the NN implemented; in fact the papers consider their specific NN, that in most cases is different from the others, causing a problem comparing them.
The starting point is to detect the interesting values reported: area, power and timing. These quantities are correlated, due to the fact that a circuit with a higher number of neurons probably uses higher power, higher timing and more area than another one, so it is necessary to create a different parameter that contains all the three cited, but that is independent on the number of neurons.
To do this, it is considered a variation of the power, the power per neuron, dividing the total power for the number of neurons. In this way, the value obtained considers not only the power used for a single neuron, but also the overhead that the insertion of a neuron causes to be managed. The results obtained are reported in figure1.43, on which are represented the parameters described for a subset of the paper analysed, due to the fact that not all of them show the quantities described.

The graph shows the three parameters for each solution, but to compare well the architectures proposed it is better to have only one parameter that combines them together, so it is possible to define the product energy-delay and the energy-area-delay one: the lower they are, the better is the solution associated on them. The figure 1.44 shows these results. It is important to note the behaviour of the technologies: by the graph emerges that in CMOS there are different behaviours, depending on the application (TrueNorth is the worst, but has a very high overhead to manage the large quantity of neurons), in Memristor technology the results are better due to the reduced area, MTJs seems to be the best due to their reduced area cost, and PIM solutions present the benefit of an optimized CMOS solution with lower area than it. The conclusion could be that MTJs technology is the best, but in the papers analysed there isn't a presentation of a complete NN built in MTJs, so the fabrication process could cause a decreasing of these theoretical performance.

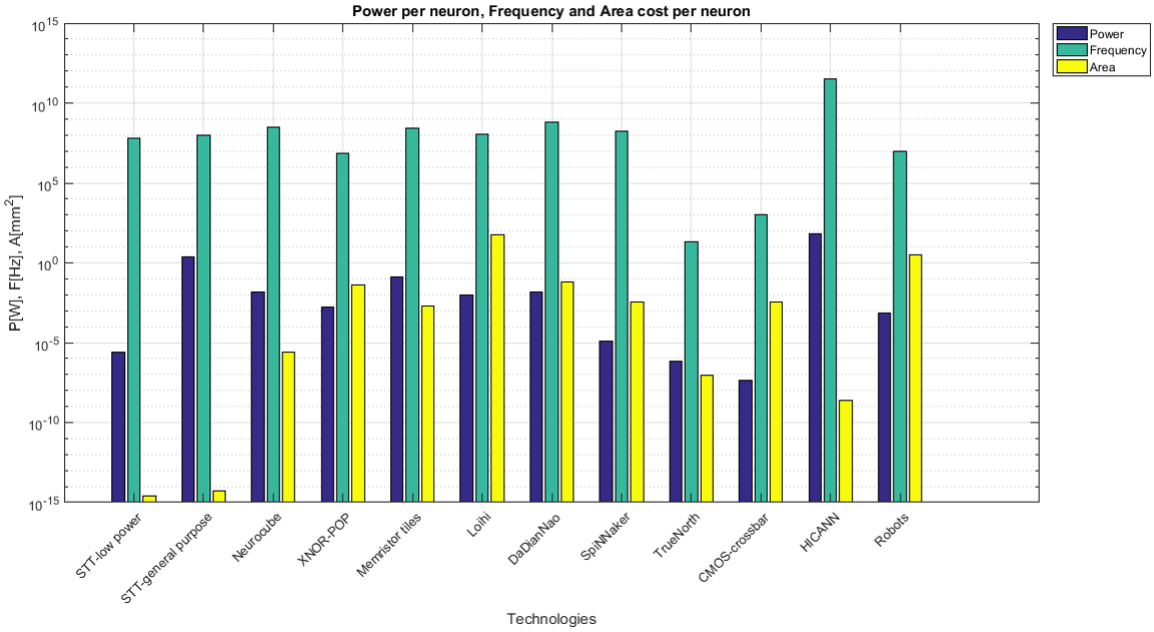


Figure 1.43: Power per neuron, frequency and area comparison between the main paper presented

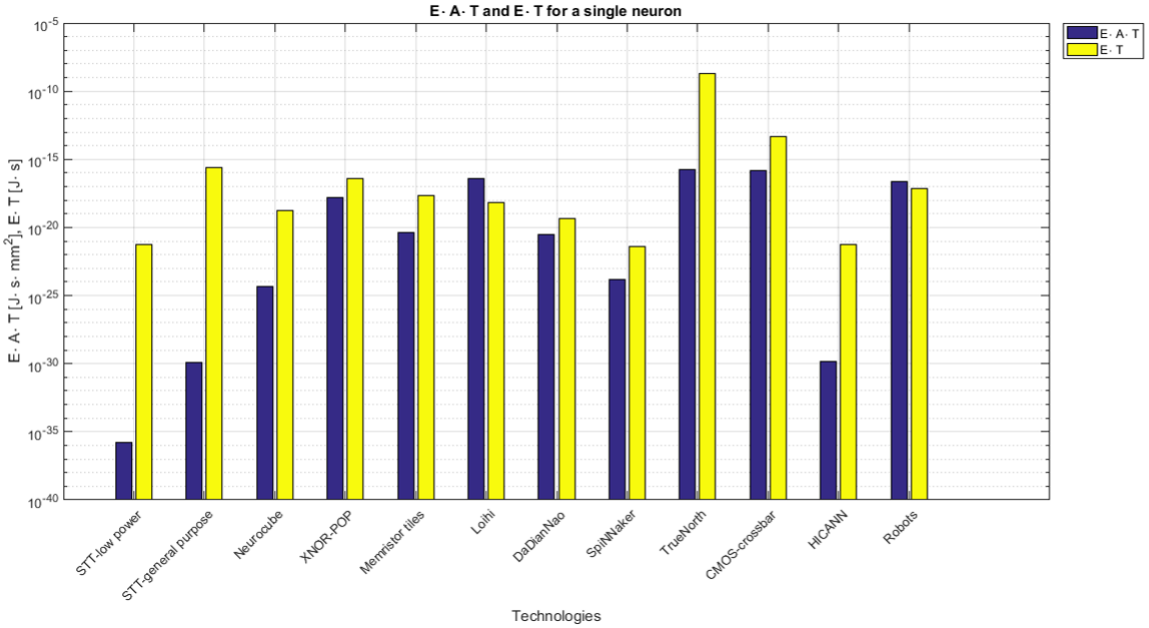


Figure 1.44: EAT and ET product for the solutions analysed

Chapter 2

Software Implementation

2.1 Metodology

In figure 2.1 is shown the design flow followed to produce the hardware, starting from a very-high-level software model. In particular, the starting point chosen is the Matlab environment, that implements by default a lot of pre-trained NNs. These models are provided with an abstract block view of the network, capable to show the top-view layer organization: which layers are present, how they are connected together and the parameters of each one. Before starting the hardware realization, it is useful to realize another representation of the NN model in a lower software level, in order to test all the intra-layer mechanisms. The programming language chosen for this step is the C++, that is a high level language with the possibility to realize a structure made up by objects and classes, similar to a hardware representation. In this way it is possible to realize an architecture that simulates what the circuit must do, implementing more features than the previous model, as, for example, the number of bit on which data are represented, and test them before the implementation, making it easier. In fact, if a changing is required after the hardware implementation, the software model can be modified to simulate the possibilities and to select the best one for the specific situation. The software reference model is very important because software simulations are generally less expensive in terms of computational cost and time, so this solution can provide a faster tool to develop the hardware. Finally, when the hardware is tested, the block synthesis is needed in order to characterize the circuit developed.

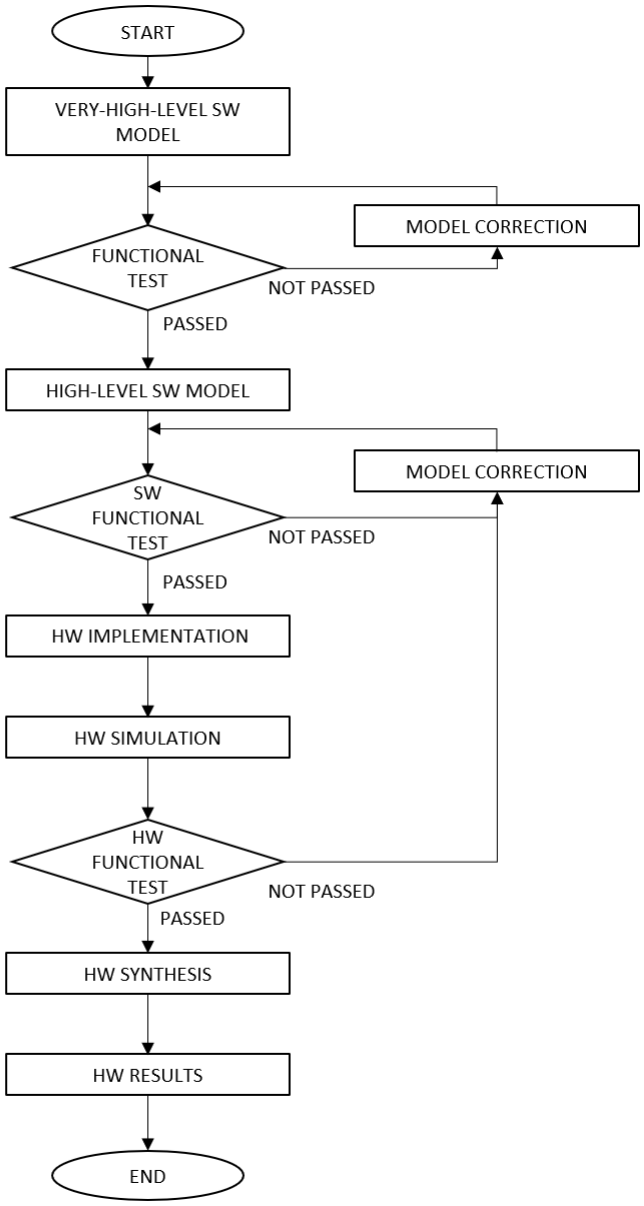


Figure 2.1: Design flow

2.2 Very Big NN: AlexNet

The pre-trained NN Matlab set contains different networks; one of the most interesting ones is the AlexNet, because it is a very useful NN to recognize images, due to its high input noise tolerance and the quantity of output classes (1000), as described

in more details in Appendix A. However, its main disadvantage is the very big size, including more than 34 millions of neurons; this causes a very hard hardware implementation and, also using a pure software solution, the computational cost and time required to get a classification can be very high.

2.3 Software implementation

The software realized implements an AlexNet with other features added in order to simulate what happens in hardware before realize it. In particular, the main important option consists in a parameter to set for each layer the number of bit used to represents input/output data and weights. In this way, the program produced is able to make the same image classification with different hardware parallelism, helping the choice for the hardware implementation.

General organization

The software executes the layers in order, reading an input .txt file, produced by the previous layer, and generating an output .txt file, that becomes the input for the next layer. In this way all the internal feature maps are available, but causing the software to be slower. Each layer is an object instantiated from its class, so it is needed to set its internal variables during the instantiation or using the setter functions before starting the layer evaluation. In a second step it is possible to start the computation function and check, according to its returning value, if the process is completed successfully or not. An example on how a layer is instantiated and managed by the main program is reported in list 2.1.

Listing 2.1: Example of the first convolutional layer management

```
/*------------Layer1-----------*/
int f_size[2]={11,11},i_size[4]={227,227,3,1},stride[2]={4,4},padding[4]={0,0,0,0};
int w_bits = 6,bits[2]={8,0};

convlayer layer1_conv1(f_size,96,"conv1_weights.txt","conv1_bias.txt",i_size,
        stride,padding,w_bits,bits);

std::ofstream out_file,log_file;
out_file.open("o_c1_file.txt");
log_file.open("log_file.txt");
```

```
12 if(layer1_conv1.elaborate(&out_file,"input_image_averaged.txt",&log_file))
       cout<<"c1: success!"<<endl;
14 else
       cout<<"c1: failure!"<<endl;
```

The classes structure of the program provides a good instrument to have a set of possible layers to implement that can be expanded in future, realizing a starting point for the high-level hardware simulation of all NNs, because if the layer classes are already present in the set of the ones implemented, it is needed only to instantiate them, however it is required to expand the set writing the new kind of layer. In both the cases, the program can work and realize the same behaviour of an hardware with the parallelism set without implement or simulate the hardware, that normally is a more expensive operation.

Classes description

All the classes are made up by the couple of header (.h) and source (.c) files, as the standard C++ implementation establishes. In the header files there are the declarations of functions and variables used in the source file: the first ones are divided in constructor, destructor, getters, to get the value of internal variables, setters, to set them, and the evaluation functions. In the source file it is present the body of these functions. There is one class for each kind of layer implemented and the class for the neuron, that is the base block on which several layers are built. On the following are described the classes realized; to see in more details the specific role of each layer in the NNs see the Appendix A.

Neuron

The Neuron class is a simple class that realizes a single neuron of a generic NN, so it is characterized by the internal variables that contain weights and the bias, both settable during the object instantiation and with setter functions. Their values can be read from an external class using the proper getter functions. There is an additional function that, passing the inputs as arguments, evaluates the output result of the neuron, making a convolution operation between inputs and weights,

due to the formula:

$$output = \sum_{i=1}^{n}(input_i \cdot weight_i) + Bias$$

The code that implements the operation described is reported in list 2.2.

Listing 2.2: Neuron elaboration function code

```
float output = 0;
for(unsigned int i=0; i<_weights->size(); i++)
{
    float tmp;
    tmp= ((*input)[i])*((*_weights)[i]);
    output += tmp;
}
output += _bias;
return output;
```

Convolutional layer

This is the most common layer in big NN as CNNs: the single layer applies a certain number of filters of a specified size to the input pattern, moving the filter to it with a parameter called stride, covering in a finite number of steps all the surface of the input matrix.
In the constructor the parameters to be initialized are:

- the filter size, that is a parameter made up by two integer values;
- the stride on the x and y directions;
- the padding, that is a frame of zeros added to the input pattern to realize an output of a specific dimension;
- the number of bits on which output results, weights and biases have to be represented;
- the number of filters contained by the layer;
- the size of the input pattern;
- the name of the files containing weights and biases.

All the quantities described above can be modified using the relative setter function. There is another function, called elaborate, that evaluates the outputs of the convolutional layer instantiated; using this function it is required to specify output file, input file and log file; the steps implemented are:

- to open the files and returns an error if the operation fails;
- to acquire weights, biases and inputs, adding the zero-padding frame is needed;
- to evaluate and write results on the output file: in this step there are loops to scan all the input pattern, on which there is the code to take the weights and biases needed in the current step, approximate them due to the number of bit previously set, use them to instantiate a neuron, to apply the inputs to the neuron, to represent with the correct number of bits the value generated and to saturate this number if it exceeds the range limited by the parallelism, applying also the ReLu function. A portion of this part of code is shown in list 2.3.

Listing 2.3: Part of code of the elaborate function in the convolutional layer class

```
std::vector <float> w,in;
// create the weight and input vectors for the current neuron
for(a=m*_input_size[2]/_input_size[3];a<(m+1)*_input_size[2]/_input_size[3];a++)
{
        for(int b=0;b<_filter_size[Y];b++)
        {
                for(int c=0;c<_filter_size[X];c++)
                {
                        float w_tmp =
                              weights[k][a-m*_input_size[2]/_input_size[3]][b][c];
                        w_tmp *= pow(2,_weights_bits);
                        w_tmp = round(w_tmp);
                        w_tmp /= pow(2,_weights_bits);
                        w.push_back(w_tmp);
                        float in_tmp = inputs[a][b+j*_stride[Y]][c+i*_stride[X]];
                        in.push_back(in_tmp);
                }
        }
}

float tmp,tmp_bias;
tmp_bias = biases[k];
tmp_bias *= pow(2,_weights_bits);
```

```
tmp_bias = round(tmp_bias);
tmp_bias /= pow(2,_weights_bits);

neuron tmp_neuron(&w,tmp_bias);
tmp= tmp_neuron.neuron_out(&in);

// limitation to the number of digits after the dot
tmp *= pow(2,_output_bits[1]);
tmp = round(tmp);
tmp /= pow(2,_output_bits[1]);

// limitation to the  number of digits before the dot
if(tmp>pow(2,_output_bits[0])-1)
        tmp = pow(2,_output_bits[0])-1;
else if(tmp<0)                          // implementation of the ReLu function
        tmp = 0;

if(tmp>mass)
        mass=tmp;

neuron_out_line.push_back(tmp);
*out_file << tmp << "\t";
```

Cross-channel normalization layer

This layer class implements the cross-channel normalization, a normalization between values placed in the same position during the output generation of the previous layer, but generated from different filters.
The internal variables of this class, readable and settable with getters and setters, are α, β, the window-channel-size (WCS), K and the input pattern size. There is an elaborate function that reads the input file specified in the arguments and, applying the normalization to its values, produces the output file with the name specified in the argument.

Max Pooling layer

This class realizes the pooling operation, that consists in taking a group of input values and replace them with only one value, that is the highest of them. This operation is needed to make a resize of the previous layer pattern, putting more in evidence the main characteristics individuated, that are the largest numbers. The parameters readable and settable with getter and setter functions are the X and Y

size of the pooling filter, the input pattern size and the stride used by the filter to move. An elaborate function scans, according to these parameters, the input values, producing the result in an output file. A portion of code reporting how the pooling operation works is reported in list 2.4.

Listing 2.4: Pooling operation

```
std::vector<float> in;
for(int b=0;b<_pool_size[1];b++)
{
        for(int c=0;c<_pool_size[0];c++)
        {
                in.push_back(inputs[k][b+j*_stride[0]][c+i*_stride[0]]);
        }
}
float pool_value = in[0];
for(unsigned int m=1;m<in.size();m++)
{
        if(in[m]>pool_value)
                pool_value = in[m];
}
pool_value = round(pool_value);
pool_out_line.push_back(pool_value);
o_file<<pool_value<<"\t";
```

Fully connected layer

This class implements a generic FC layer. The parameters settable are the input size and the softmax function enable, a flag that enable or disable a feature that implements the softmax function, a function that, using all the output values of the last FC layer, produces the probability that the classification executed is right. It operates with the formula:

$$y = \frac{e^x}{\sum_{i=1}^{n} e^{x_i}}$$

If this option is not set, a ReLu function is automatically applied after the FC layer. There is an evaluation function that, taking the input, weights, biases and output files as argument, produces the outputs after reading them instantiating the neurons required and representing the values obtained with the number of bits specified in the last two input arguments of the function (the first one is the number of bits for the integer part, the second one is for the decimal part).

Time requirement to classify an image

The approach chosen consists to open continuously files previously generated, to read them, to generate other output files and to close all, in order to leave the control of this process to another class. This flow requires less memory than other ones, but it is slower, due to the fact that intermediate results are continuously loaded and removed from the memory. This causes a time to classify an image that is around few minutes, depending on the pattern analysed. This time is higher than the one required for a Matlab script that uses around a half minute to take the classification, but implements more useful features, as the possibility to set the bit number for the internal values representation and the generation of the internal layer feature maps in files, useful when it is needed to test the hardware.

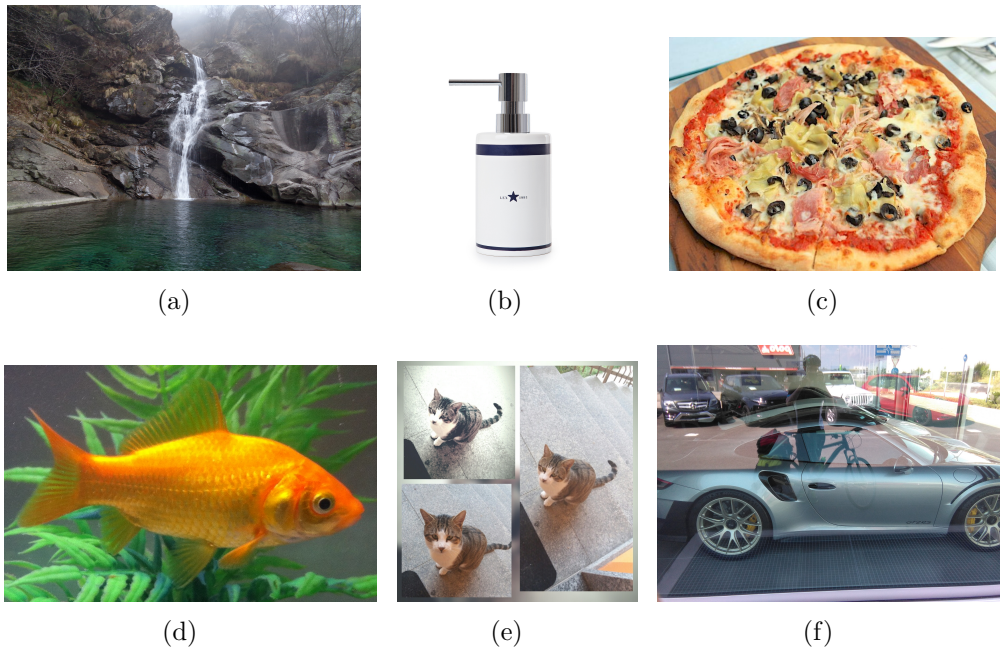
Parallelism of the data into the hardware

As already described, one of the purposes of the software realized is to implement the same features of an equivalent hardware before realize it to fix some requirements using shorter simulations. The most important point is the hardware representation of data, because the software can use floating point data and operations, but the hardware has got resources limits and constraints, so it is important to establish if it is possible to operate in fixed point and, in positive case, what is the number of bit required to correctly represent input, internal and output data. Before apply these steps to the software it is needed a short analyses of the situation:

- the software uses .txt input files, so it needs to convert an image in its pixel representation (it is described by three number for each pixel, indicating the quantity of Red, Green and Blue presents) and to write it into a .txt file before starting a classification;

- it is important to have some reference cases to make the comparison between the bits limited version and the non-limited one; the images chosen don't come from a database but are a mixture of images with high noise, so containing high difficult degree to classify them, and simpler figures. Examples of images used are reported in figure 2.2;

- it is important to reduce the number of bit in sequential steps to see what is the behaviour of internal data and if the output classification is affected or not from the changes.



(a) (b) (c)

(d) (e) (f)

Figure 2.2: (a) Cliff image covered by a waterfall (b) A commercial soap dispenser (c) Pizza (d) Goldfish (e) Three images of the same cat reproduced in different colours tone in the same image (f) Picture of a sports car with high noise

In table 2.1 are reported the results for some of the images previously shown with high difficult classification degree changing the number of representation bits. The numbers at the beginning of the columns are respectively the number of bits of: input integer part, input mantissa part, weight mantissa part. The integer part of weights is fixed to 2 bits because the weights can assume values in the range (-2;2). Only for fully-connected layers the 8 input bits are divided in 6 bits for the integer part and 2 for the mantissa one, due to the fact that the previous layers reduce the maximum representable value and the FC layers are the last ones, so they have to be more precise. Moreover, the input bits for the first layer have to be be represented in a signed way, but all the values produced by the internal layers are positive, because a normalization ReLu function is applied after them. This phenomenon is caused by the fact that the inputs are pre-elaborated with a simple subtraction, starting from

Layer	20-20-14	8-0-14	8-0-8	8-0-7	8-0-6	8-0-5
Cliff						
Conv 1	1470.09	255	255	255	255	255
Conv 2	351.466	255	255	255	255	255
Conv 3	248.321	247	249	247	242	222
Conv 4	126.463	127	130	128	137	122
Conv 5	112.187	114	112	106	112	113
FC 1	30	30	29	30	28	28
FC 2	12	12	12	12	10	11
FC 3	16	16	17	16	15	14
Class	Cliff	Cliff	Cliff	Cliff	Cliff	Cliff
Cat						
Conv 1	1591.3	255	255	255	255	255
Conv 2	452.973	255	255	255	255	255
Conv 3	268.13	255	255	255	255	255
Conv 4	161.037	161	163	164	159	179
Conv 5	197.089	199	199	192	202	161
FC 1	45	45	45	40	46	50
FC 2	8	8	8	9	9	10
FC 3	10	10	10	10	10	13
Class	Siamese Cat	Siamese Cat	Siamese Cat	Siamese Cat	Siamese Cat	Tub
Sports car						
Conv 1	1437.56	255	255	255	255	255
Conv 2	413.818	255	255	255	255	255
Conv 3	337.291	255	255	255	255	255
Conv 4	361.34	255	255	255	255	255
Conv 5	178.224	166	170	167	172	172
FC 1	32	33	33	33	32	37
FC 2	9	9	9	9	9	7
FC 3	16	16	16	16	15	12
Class	Sports car	Sports car	Sports car	Sports car	Sports car	Sports car

Table 2.1: Examples of classification results and internal maximum value for each layer changing the bit representation.

positive unsigned values and ending with signed numbers. To avoid this problem it is possible to represent on 8 bits all the layer inputs, also the first one, but using a signal that is capable to determine if the values are signed, unsigned or with a 2 bits mantissa. In this way the hardware produced is a generic component that can

be reused for each of the scenario described.

From the table 2.1, it is possible to conclude that the data can be represented with 8 bits of integer part and the mantissa can be neglected, approximating correctly the internal results. For what concerns the weights, their mantissa part can be represented with 6 bits, because the data reported show that with 5 bits some images produce wrong results, depending on the complexity. The data for 4 mantissa bits aren't reported because for the most part of images the results are wrong. The representation chosen consists in 8 bit of integer part for input and output data and, for weights, 2 bits of integer part plus 6 bits of mantissa.

Example of software output

In figure 2.3 is reported an example of classification made by the software; on each line is written an information about the layer evaluated in that moment, specified by the letters at the beginning of the row:

- cx: stays for convolutional layer of number x;
- crossx: stays for cross channel normalization layer of number x;
- px : stays for pooling layer of number x;
- fcx: stays for fully-connected layer of number x.

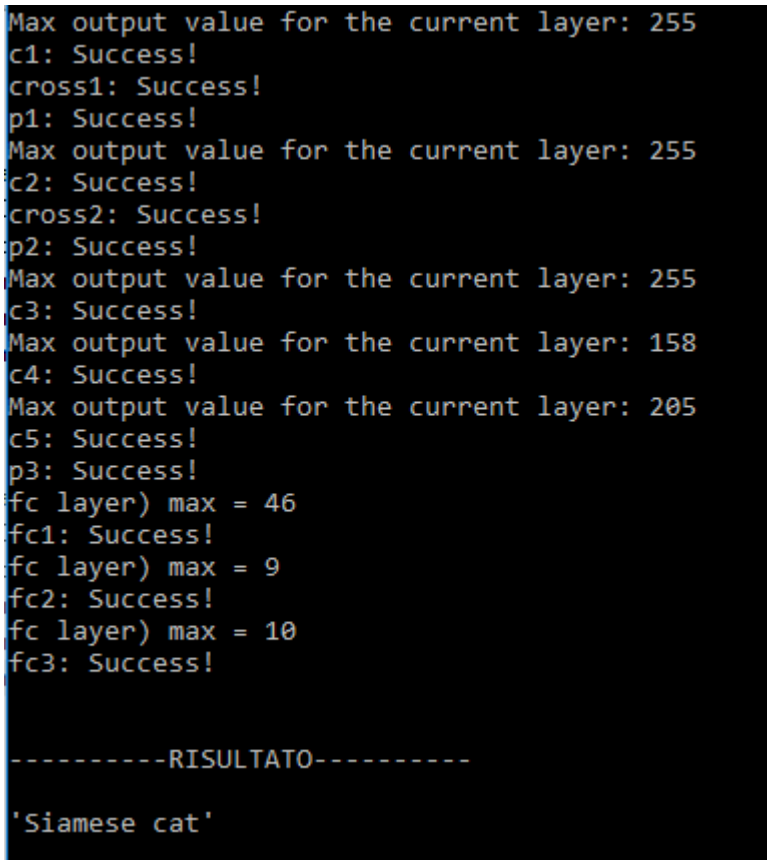


Figure 2.3: Example software window

Chapter 3

Hardware Implementation

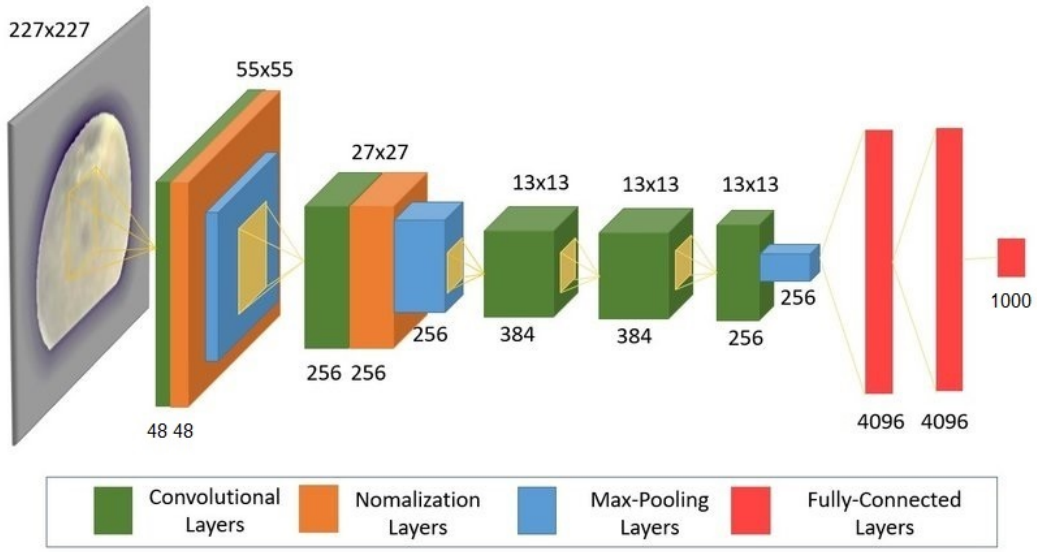


Figure 3.1: AlexNet 3D representation

Image recognition CNNs, as the AlexNet shown in figure A.1, are very interesting and useful, but are very big, causing high computational cost. If it is needed to realize an hardware to implement them, the starting point can be to create a chip with inside the entire network; this solution generates the best choice in terms of speed, because all the network is simultaneously working. Moreover, due to the fact that a layer produces valid outputs only after the previous layer makes the same, the flow of data is sequential, so it is possible to put pipeline registers between layers to make the critical path littler and increase the throughput. This scenario however isn't realizable, because this network is too much big to be implemented in a chip. In table 3.1 they are reported the quantities of data managed on each

Layer type	# filters	input size	# $inputs_{1neuron}$	# neurons	TOT_{mult}
Convolutional 1	96	227x227x3	363	290400	$105.42x10^6$
Convolutional 2	256	55x55x96	1200	186624	$223.95x10^6$
Convolutional 3	384	27x27x256	2304	64896	$149.52x10^6$
Convolutional 4	384	13x13x384	1728	64896	$112.14x10^6$
Convolutional 5	256	13x13x384	1728	43264	$74.76x10^6$
Fully Connected 1	-	13x13x256	9216	4096	$37.75x10^6$
Fully Connected 2	-	4096x1x1	4096	4096	$16.78x10^6$
Fully Connected 3	-	4096x1x1	4096	1000	$4.10x10^6$
TOT	-	-	62875	663368	$724.42x10^6$

Table 3.1: Hardware requirements for the AlexNet CNN

layer of the network; seen these data it is possible to note that the total number of neurons is more than half million and, considering that a single neuron requires one multiplier for each input, the total number of multipliers to be instantiated is over half a billion. Moreover, considering the high number of inputs and outputs for each layer, it is possible to understand that this architecture has too much internal and external interconnections; putting together these observations, neglecting other possible problems, it is possible to conclude that it is very hard to realize a single chip implementing all the network. In this way, the possibilities to realize this network become to use more chips, very hard solution due to the high number of inputs and outputs, or to serialize part of the evaluation, to use less area, power and interconnections. This last approach is the one chosen in the WINNER architecture, where there is a trade-off between a complete serial and a complete parallel circuit. In fact, if it is impossible to implement all the architecture due to power and area constraints, a solution could be to realize only a subset of neurons and reuse them to evaluate, in sequential steps, all the values required. This is a good choice, but the quantity of neurons to implement has to be determined according to certain considerations:

- to optimize area, this number must be the slowest possible: the littlest number is one, but in this way the time to realize one image classification becomes very high;

- the neurons implemented in a single layer are divided into a certain number

of types, that are the filter types: all the neurons of the same type have the same weights and bias, so, realizing one neuron for each kind of them, it is possible to reuse them without change these parameters;

- the neurons of the FC layers are realized in the same way of the convolutional ones, but typically they present a number of inputs many times higher, making difficult to analyse all the input pattern simultaneously.

Taking into account the facts analysed, the choice implemented into the WINNER architecture consists in 384 neurons, that is the highest number of filters in the same convolutional layer, to warranty the parallel evaluation of all the filters for a fixed input pattern. Each neuron is connected to the same inputs, that are 64, a number that is an integer power of 2, useful to optimize internally the neuron and higher enough to implement the serialization process also on the input evaluation without making the system too much slower. The steps number to evaluate one neuron output and all the neuron outputs for each layer are shown in table 3.2: in 52902 steps is possible to classify an image.

Layer type	$\#\ steps_{neuron}$	$\#\ steps_{layer}$	TOT_{steps}
Convolutional 1	6	3025	18150
Convolutional 2	19	729	13851
Convolutional 3	36	169	6084
Convolutional 4	27	169	4563
Convolutional 5	27	169	4563
Fully Connected 1	144	11	1584
Fully Connected 2	64	11	704
Fully Connected 3	64	3	192
TOT	-	-	52902

Table 3.2: Step number for the evaluation of a single neuron and a single layer

3.1 Block scheme

The standard block scheme of a generic NN layer is made up by a certain number of neurons that, combining the inputs together, produce the outputs, as shown in figure 3.2. This topological view doesn't specify how the neurons are implemented,

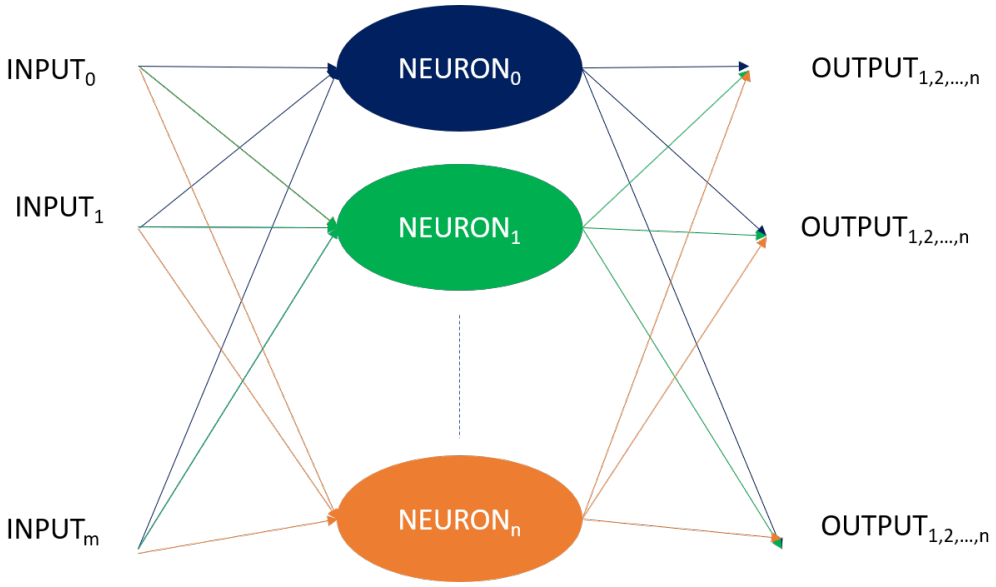


Figure 3.2: Topological view of a standard layer

so there are several possibilities. Considering that a neuron is a unit that implements a multiplication for each couple of input-weight and that the weights are fixed after the training for a specific NN, it is possible to conclude that an efficient structure can be the one reported in figure 3.3, representing two blocks:

- Weight-Block: it consists of a memory that contains all the weights of the NN, implementing internally the neurons that, making multiplications between external inputs and internal weights, produce the outputs required;

- In/Out-Block: it is composed of a memory that contains the input image and store the output of the Weight RAM to use them as input for the next layer, making also the padding and pooling operations if required.

With this architecture, a user has to initially load all the Weight-Block with the weights, then to load the In/Out-Block with the image to classify and to start the computation, waiting its end before loading another image. The In/Out-Block provides the inputs for the neural hardware implemented into the Weight-Block and store its outputs to manage the feature maps of the hidden layers.

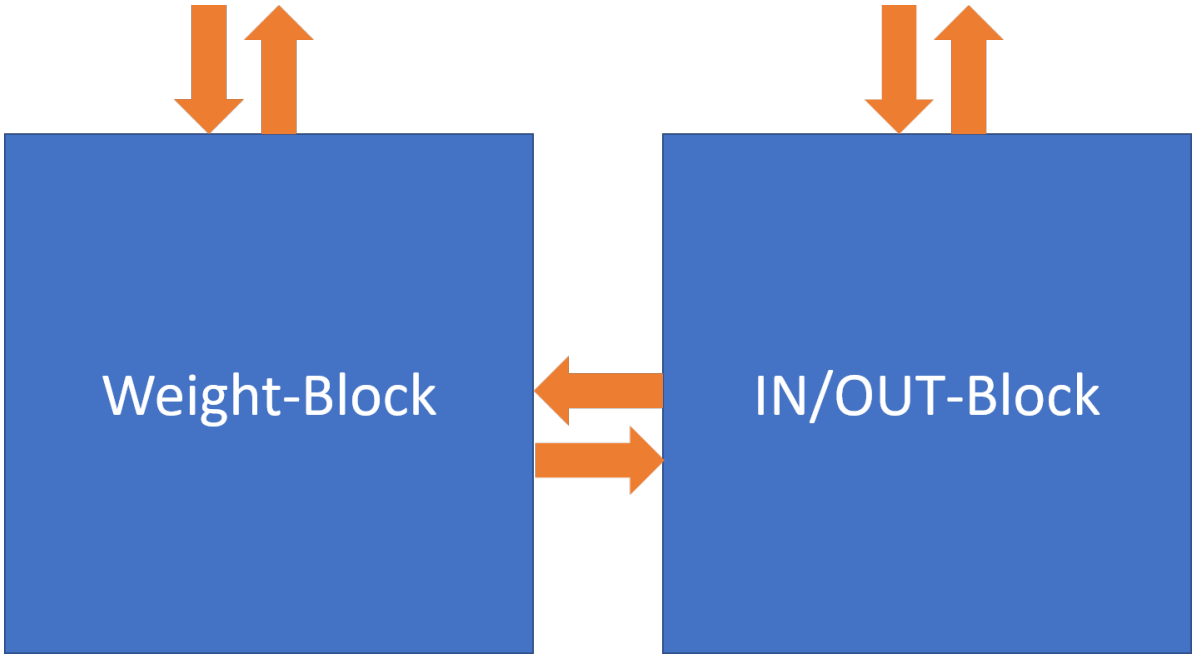


Figure 3.3: WINNER Block Scheme

3.2 Weight-Block

Block Scheme

This block is readable and writeable as a standard RAM and contains the weights; moreover it has got another input port to acquire the neuron inputs and another output port to produce their outputs, as shown in figure 3.4. The size of these ports are 64 Bytes for the input one, because each input is represented on 8 bits, and 384 Bytes for the output one, because there are 384 neurons implemented. The choice to use a memory to store the weights comes from the observation that normally the delay problems are caused by the fact that data and the computational blocks are far, so the consequence is that the critical path becomes very long. Using a memory, it is possible to allocate these blocks near the data stored, reducing the delay and improving the performance, implementing in this way a Logic In Memory or Logic Near Memory architecture.

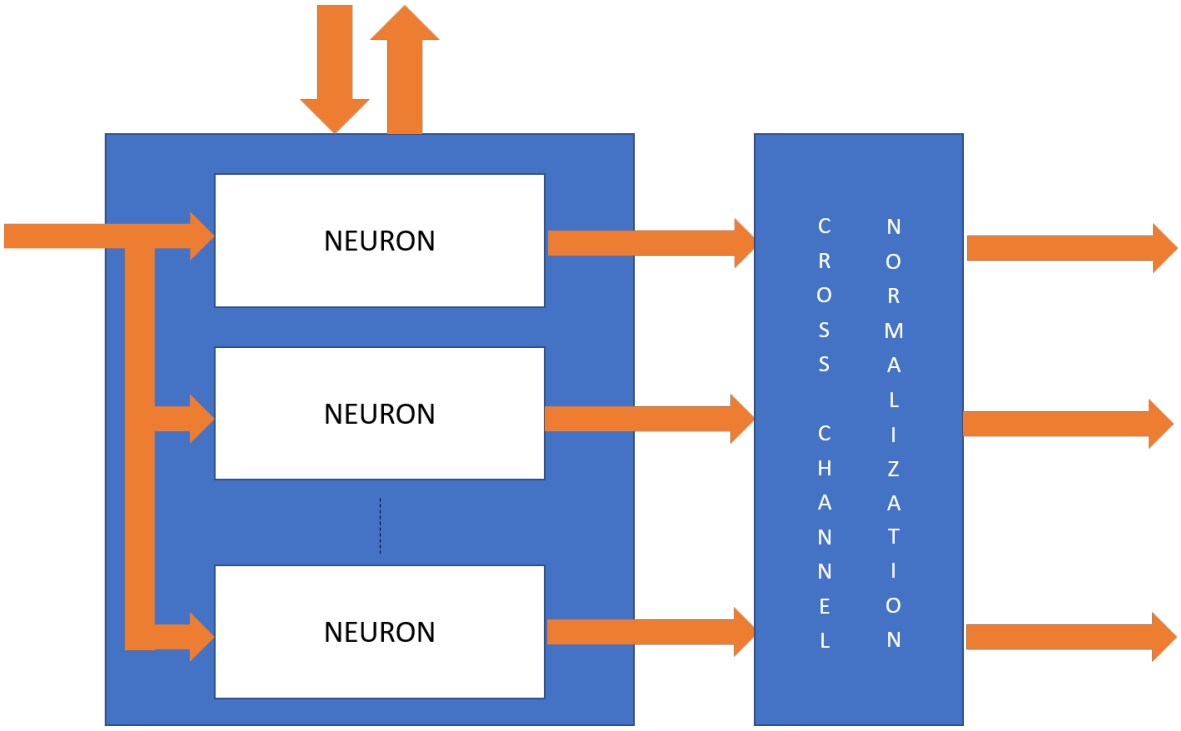


Figure 3.4: Weight RAM of the WINNNER architecture

Neuron

The most important part of the WINNER architecture is into the Weight-Block, because the neural computation is executed into this part of the system. For this reason, the block with the biggest relevance is the neuron, that can be implemented starting with the standard concept of neuron and then moving to a more innovative solution.

Standard Neuron

The standard implementation of a neuron comes from the convolution formula that it uses to provide the result:

$$y = \sum_{i=1}^{n} (w_i \cdot input_i) + Bias$$

On this formula all the inputs are multiplied by the respective weights and, finally, a term called Bias is added to the sum of these products. The easiest way to realize this behaviour is to implement a multiplier for each couple of input and weight and then to add all the results into an adder that sums also the Bias term with them, as shown in figure 3.5. However this structure has some problems:

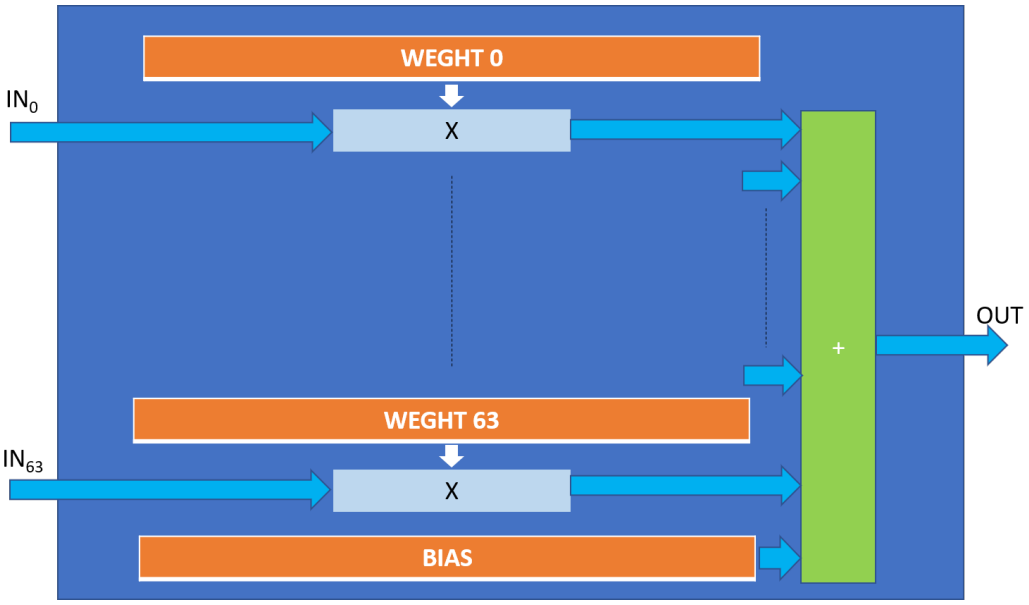


Figure 3.5: Block scheme of a standard neuron implementation

- standard multipliers can be slower respects on other kind of solutions;
- the adder has to add simultaneously all the inputs, that in the WINNER architecture are 64, causing one of the main sources of delay;
- this neuron can process no more than 64 inputs, but very big NNs as AlexNet have higher number of them, so this structure can't implement big networks;
- the weights used are fixed, so, if it is needed to implement a higher quantity of neuron types, it is required to stop the evaluation in a certain point, to reload all the Weight-Block with the new weights and restart the computation, causing a very high inefficiency of the system and a very low performance.

All these problems are solved using a non-standard implementation, introducing some blocks and optimizations that make the starting architecture more flexible and with better performance.

Optimized Neuron

The neuron implementation proposed in the WINNER architecture is made up by several blocks:

- **WordLine**: the neuron is part of the Weight-Block, so it contains several WordLines, each one constituted of all the weights needed to implement the NN selected, in the specific case the AlexNet. This means that a RAM row has got a certain number of weights to implement the input data serialization of a neuron and a certain number of these weights set to implement the neurons of each layer; considering the results previously shown in table 3.2 and the size of a weight, that is 1Byte (2 bit for the integer part and 6 bit for the decimal part), the row size is:

$$WL_{size} = 6 + 19 + 36 + 27 + 27 + 144 \cdot 11 + 64 \cdot 11 + 64 \cdot 3 = 2595 Bytes$$

  In figure 3.6 the WordLine are the blocks with the orange colour.

- **Selector**: in order to select the part of the wordline containing only the weight needed in a certain step, it is necessary a selector externally driven; it can be implemented with these two alternatives:
  - **Multiplexer**: this choice is more standard one but, considering that the wordline has a very large size (more than 2kB), it can be expensive in terms of area and power requirements;
  - **Wired-Or**: this solution uses a decoder to choose which bits to enable; all the bits with the same weight are connected to the same bitline, modelled as an OR operation between all of them, so the result is that only the bits enabled by the decoder take effect on it, obtaining the correct output.

  There is another selector, implemented as a multiplexer, to choose the input of the final adder between the sum of all the multiplications (weight times input)

or the Bias term. The first case is chosen if the result obtained is partial and it is needed to sum it with the result obtained adding other inputs; on the contrary, if there aren't other steps to generate the correct result, it is needed to sum the Bias, and this is the second scenario. In figure 3.6 the selectors are the yellow blocks: the horizontal ones are the wordline selectors, the vertical one is the selector between the sum result and the previous one.

- **PPU**: these blocks are the light blue ones in figure 3.6 and they are responsible of the generation of the partial products; in fact, a standard multiplier can be too much slower, so the idea is to implement a Modified Booth Encoder Multiplier with radix 2, generating the partial products near the wordline and using them in the adder chain that follows these blocks. To see in more details how the MBE works refer to Appendix B.

- **Adder**: they are needed to make correctly the convolution operation; there is an adder at the end of each PPU to evaluate the multiplication results and these values become the inputs for an adder reversed tree with each adder that sums two intermediate results. The final obtained number is one of the possible input of the last adder, that sums the previous sum value with it or with the Bias term, depending on the evaluation step.

- **Register**: it contains the previously evaluated result; if the output evaluation of a neuron isn't finished due to the high number of inputs, it is possible to separate in sequential steps the computation, using this register to temporarily store the partial sum produced.

The parallelism problem

As described in the software approach, it is important to choose a unique parallelism for the architecture; in this way it is possible to easily reuse it for the evaluation of all the layers included into the NN implemented. This choice causes the presence of an additional signal that sets in this three ways the parallelism:

- for the first layer, it sets the inputs as 8 bits signed.

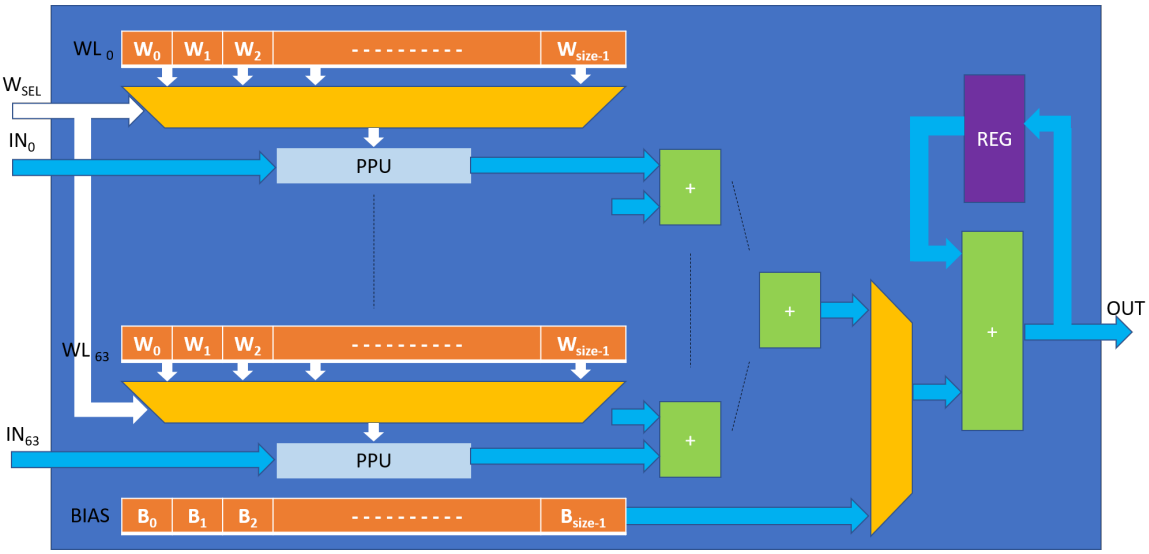


Figure 3.6: Block Scheme of the WINNER neuron implementation

- for the other layers, it sets the inputs as 8 bits unsigned, so the internal multipliers use an additional bit to evaluate the result if the MSB is '1'; in fact, using MBEs the result is signed, so the two inputs are always considered signed numbers, but, in this case, one of them is unsigned, so adding a '0' MSB the problem is solved. This solution is equivalent to add to the MBE result the other input (that is the weight) multiplied to the MSB.

- for the first FC layer, the inputs are 8 bits signed and the output 6 bits integer part plus 2 bits mantissa part, causing a shift of two bits in the result respect on the convolutional evaluations.

- for the other FC layers, both inputs and outputs have got 6 bits integer part and 2 bits for the mantissa, causing a shift of 4 bits in the result generation respect on the convolutional case.

Cross Channel Normalization Layer

This layer is used to normalize the values obtained analysing the inputs in a fixed step using the values provided by the adjacent channels contained into the Window Channel Size (WCS) to make the process. For this reason, the estimation of

this layer is put into the WINNER architecture after the neuron outputs into the Weight-Block, because, in this way, each time the filter outputs are available the normalization instantaneously is applied. To see more specification on this layer into the AlexNet refer to Appendix A. However, this kind of normalization is not

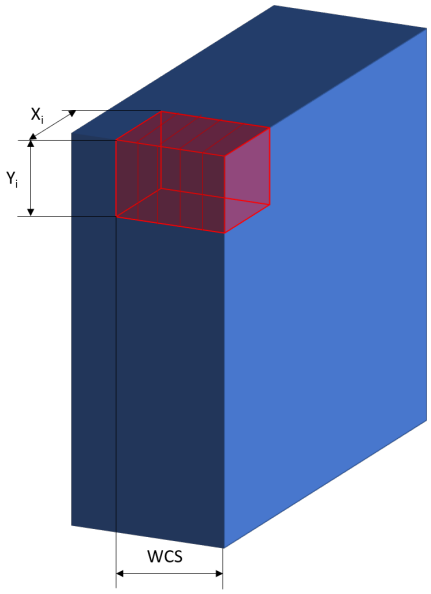


Figure 3.7: Example of a Cross-Channel Normalization Layer evaluation on a specific point of an output pattern

simple to be evaluated in hardware, due to the formula to be applied:

$$y = \frac{x}{(K + \alpha \cdot \sum_{i=1}^{n} x_i^2)^{\beta}}$$

In this evaluation, the critical point is the denominator, that has a power with coefficient 0.75, very hard to be evaluated in a small hardware circuit without floating point units. The solution is an approximation of the result, writing the Taylor series of the term $(1 + x)^{\beta}$, shown in Appendix C. In this way the computation becomes:

$$y = 1 + \beta \cdot x + \frac{\beta \cdot (\beta - 1)}{2} \cdot x^2$$

The result so obtained is an approximation of the real value, valid only near the point x=0, but, analysing the behaviour of the values involved, the term x can assume

the values between 0 and 6.25 (number obtained if all the values involved are the maximum possible on 8 bits, that is 255). This phenomenon causes a possible source of error in the NN implemented; in figure 3.8 are shown the approximations with the Taylor's polynomial changing its degree. The approximations are good only in

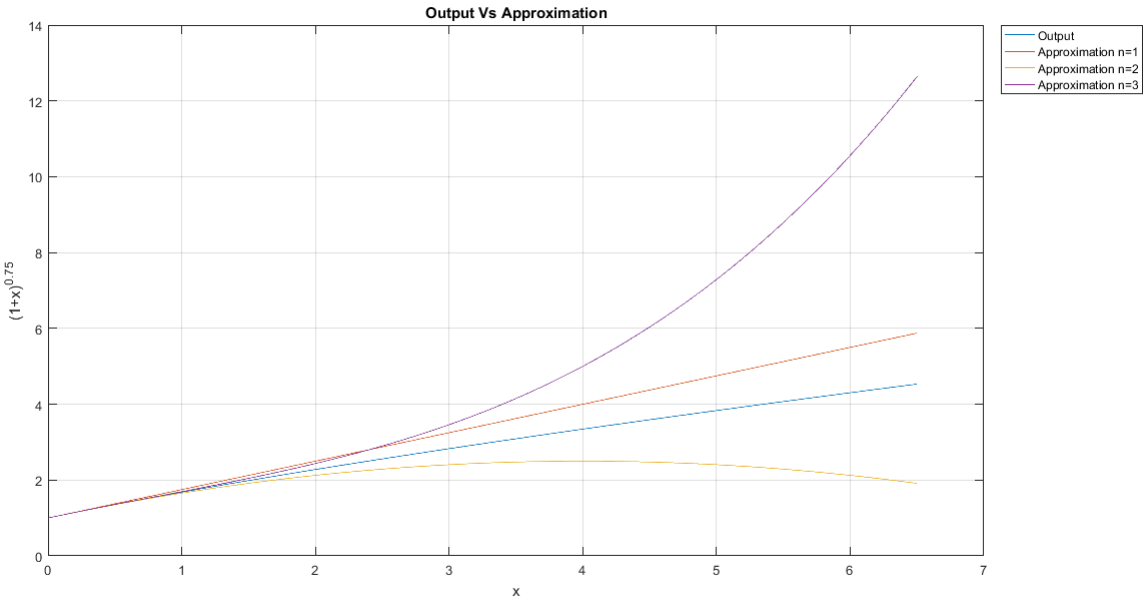


Figure 3.8: Taylor approximations of the cross channel formula denominator in the AlexNet values range

the range (0;1.5), then all the polynomials assume different behaviours. However, the cross channel normalization is applied only after the two convolutional layers to reduce the range of the values obtained, that tends to diverge and to saturate. Considering this, it is possible to say that, choosing a polynomial approximation that reduces the possible maximum value, if statistically the values that exceeds the range previously indicated are a little percentage of the total, the approximation can introduce a tolerable noise on the network that doesn't affect the result.
Using the software realized to simulate the hardware, this property was verified, obtaining that the values above the 1.5 threshold are around the 5% ,depending on the input image. With the same software, the classifications were tested to be sure that the approximation have no influence on them, and the results were the same of the ones obtained with the original formula.

3.3 In/Out-Block

This is the block containing all the input and output data from the layers; it stores the feature maps and it provides all the signals to manage the Weight-Block, according to the kind of layer and the parameters associated to it. Moreover, it can implement the Max-Pooling operation needed by certain kinds of layer and, if a zero-padding on the current pattern is required, can add it.

Block Scheme

The In/Out-Block is made up by a simple RAM with wordline of size 1 Byte because input, output and hidden layer values are represented on 8 bits, so it is readable and writeable as a normal RAM; the output of this memory is managed to implement or not a Max-Pooling and a zero padding. It has got two identical banks, because in each step it is required to have a place on which the inputs are taken and another place to store the outputs generated; for this reason, the two banks are alternatively used in read or write mode, due to the current step executed. The input parallelism is 384 Bytes because it is required to store, in the worst case, the outputs of all the 384 neurons implemented into the Weight-Block; the output parallelism is 64 Bytes due to the fact that the maximum inputs number for the Weight-Block is 64. Moreover, there is a unit, called Address-Bank selector, responsible to apply the mathematical rules needed in the current step to select correctly the read and write blocks and the addresses on which to write and to read; figure 3.9 shows the block scheme described.

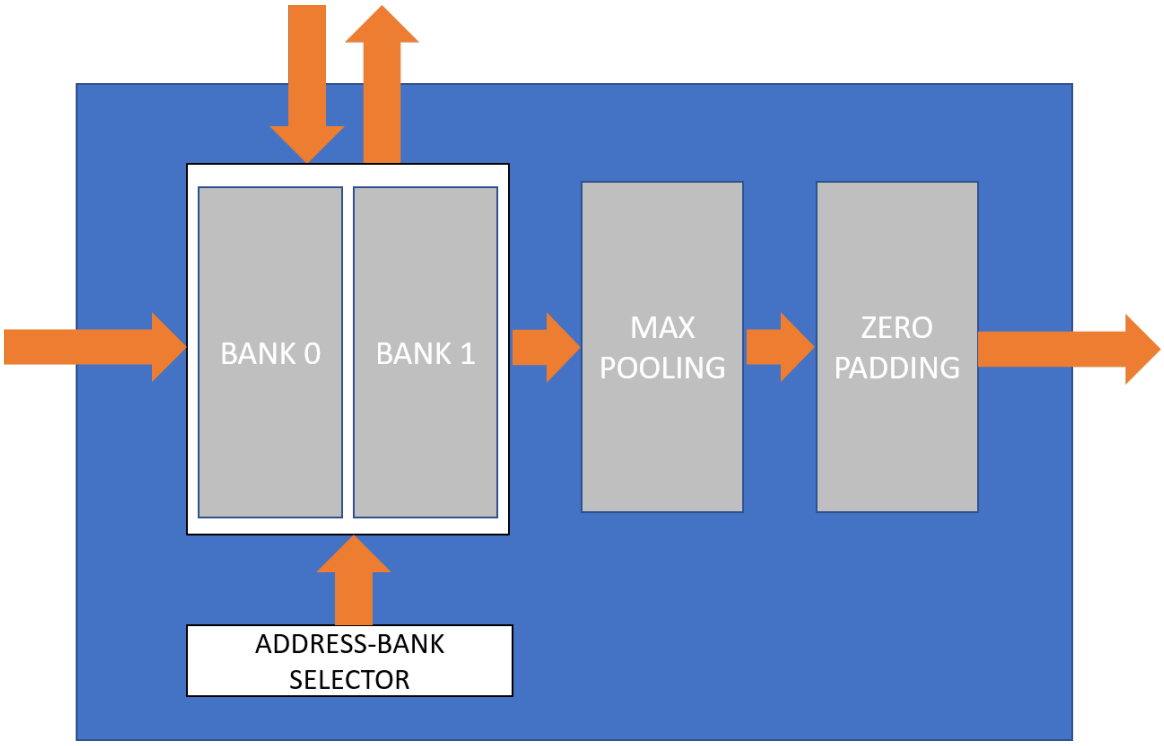


Figure 3.9: In/Out-Block block scheme

Chapter 4

Simulation

The simulation is one of the most important steps to realize an architecture, because it provides a way to see what happens into the circuit. Moreover, it is a debug instrument that permits to discover unwanted behaviours and to fix the bugs, providing a description code that implements correctly the theoretical one.
The software used to reach this purpose is QuestaSim, a Mentor tool that has many features, as the wave analyser, that shows the internal signals selected behaviour versus time as waveforms.
A possible problem during the verification of the architecture implemented is caused by the fact that there are a very high quantities of values. This very big collection of signals produces at each time-step partial results, so the verification process is very hard because in a fixed step not all the quantities have correspondence to the software solution. Considering this, it is possible to modify the software code illustrated in chapter 2 to show or store the partial products and sums during the evaluation of a hidden layer, for example, and, in this way, providing an instrument to compare and verify the internal results at each simulation step, making the bug-searching phase easier and faster.

4.1 Weight-Block

To test the Weight-Block is needed to consider the main blocks that compose it:

- the words;
- MBE multipliers;
- The adder reversed tree to sum them;

- the mechanism to select the actual correct Weight and the one to sum the previous partial value or the Bias term.

To verify the correct behaviour of these components, it is needed a test-bench capable to emulate the typical situation for this block, providing all the input signals that it requires during its normal working phase. In particular, the steps that this test-bench has to implement are the loading of the weights into the memory and the application of the correct input set, due to the specific time-step of the simulation; its complete behaviour is shown in figure 4.1.

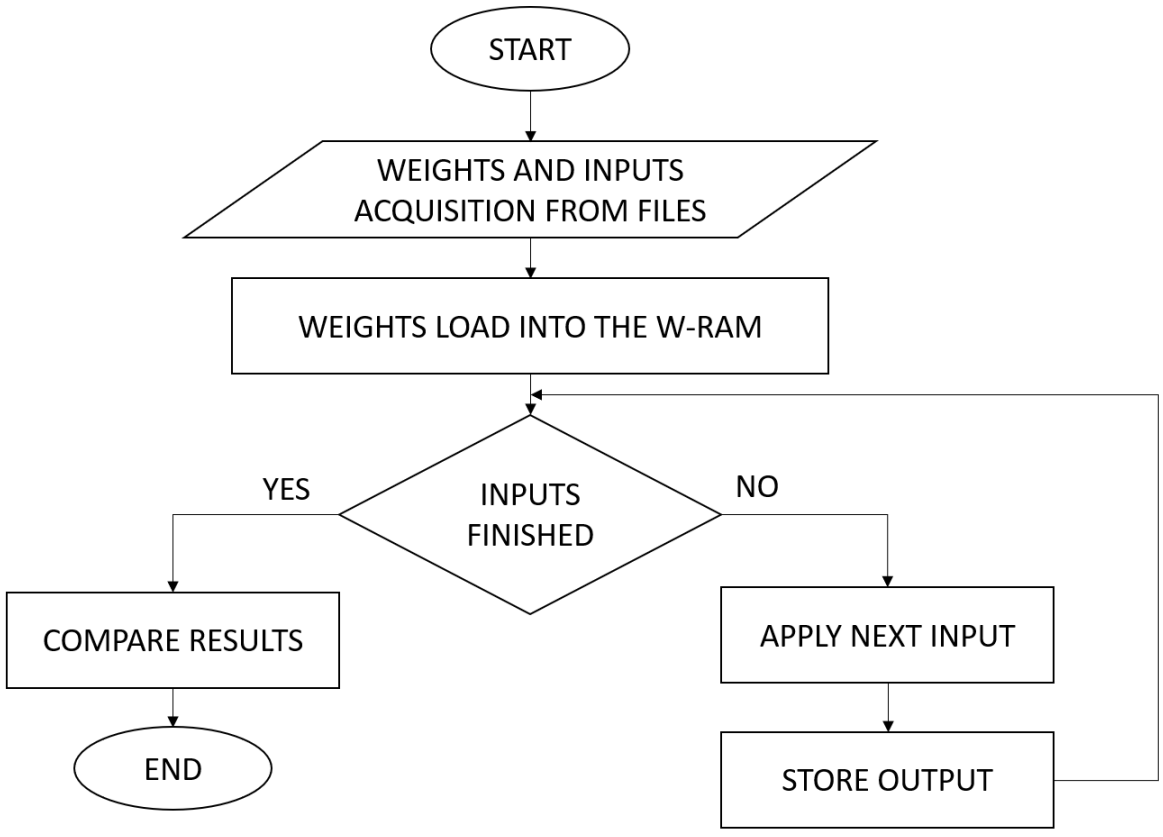


Figure 4.1: Test-bench flow-chart

Word loading

This is the first phase of the test-bench and it covers a very important role, because it initializes the Weight-Block loading all the weights required. The operations implemented are very simple: the input data is set, the address where storing the weight is updated and the Write Enable signal has to be active; in this way the first positive edge of the clock executes the write operation into the memory. Figure 4.2 shows an example with 64 weights loaded and the Write Enable signal put to the low value to stop the operation; this example uses a reduced number of bits for each word to make simpler the image. Note that the initial value of the words is a null number due to an initial reset of the memory.

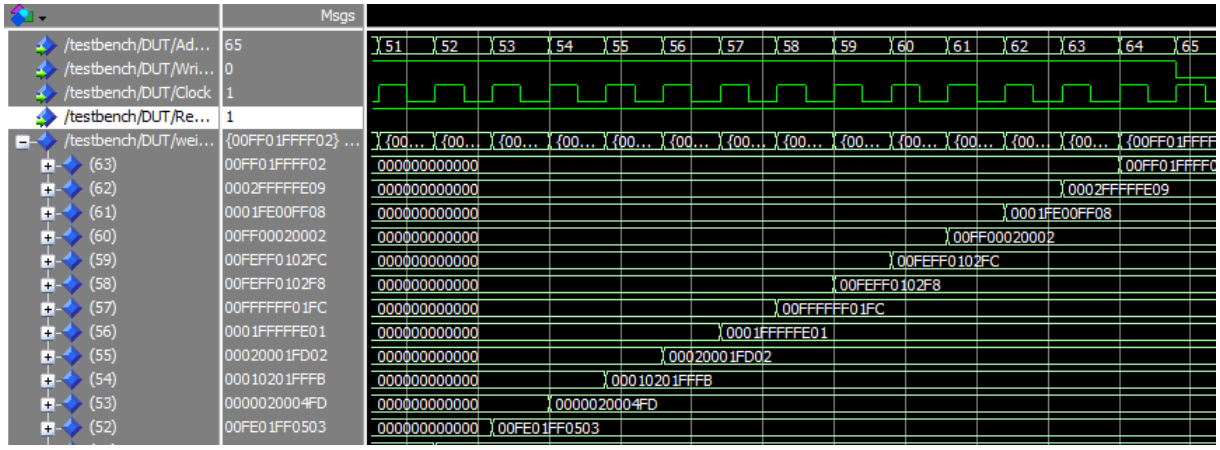


Figure 4.2: Weight loading example

MBE Multipliers

To test the MBE multipliers it is needed to see if the partial products produced by the four BEU units are right or wrong, so, for a fixed time simulation step, these numbers have to be consistent to the inputs, that are the correspondent weight and input data, as shown in figure 4.3, that shows the first one on the blue waveform, the second one on the yellow wave and the results with the white color. To verify if the simulation of the partial product produced is the expected one, it is possible to evaluate the theoretically result and compare them; as shown in figure 4.4, the two approaches provide the same results.

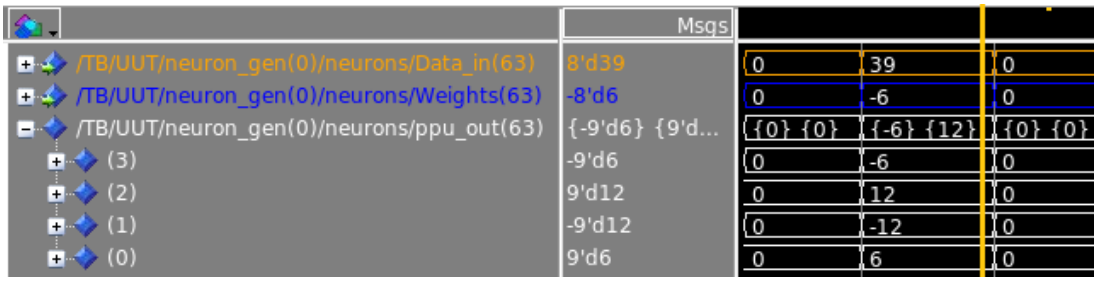


Figure 4.3: MBE partial product values in a fixed time instant

$D_{in} = 39_d = 00100111_b$ Weight = -6

0 0 1 0 0 1 1 1 (0)

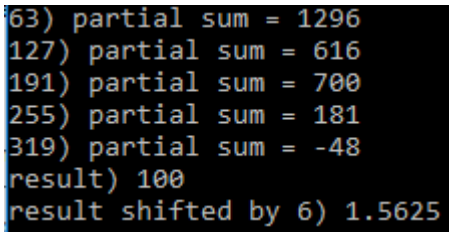
(a) (b) (c) (d)

(a) 001 → Partial product$_{(a)}$ = +Weight = -6
(b) 100 → Partial product$_{(b)}$ = -2 · Weight = +12
(c) 011 → Partial product$_{(c)}$ = +2 · Weight = -12
(d) 110 → Partial product$_{(d)}$ = -Weight = +6

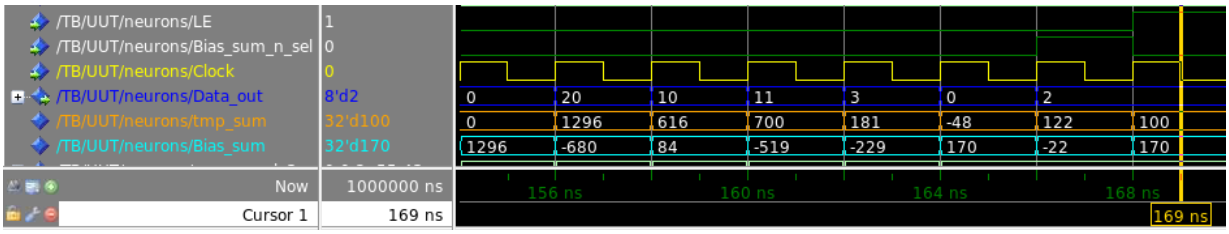
Figure 4.4: Theoretical evaluation of the partial product of the MBE

Adder reversed tree

In order to have a comparison instrument, the software illustrated in section 2 was modified, showing the partial additions equivalent to the ones generated by the hardware: each 64 input evaluations, the partial sum is shown using a signed bit representation without considering mantissa, that is the same format used in QuestaSim. In this way it is possible to compare the hardware and software results, checking if they match or not. In figure 4.5 it is represented an example of the process described. Each positive clock edge, the partial sum between the current 64 inputs and the previous result is evaluated, providing a temporarily result, indicated in figure 4.5b as *tmp_ sum*, that is the signal to compare to the software partial results of figure 4.5a. The last step before the valid output is characterized by the selection of the Bias to add to the previous result, so the signal *Bias_ sum_ n_ sel* becomes



(a)



(b)

Figure 4.5: (a)Software output for a specific time-step (b)Hardware output for the same simulation-step

'1' in presence of this step. In the next time-step, the signal *LE* becomes '1' to put in evidence the end of an evaluation and that the output is valid. The output signal is provided using the *tmp_ sum* signal, that now is containing the sum of the last partial result with the Bias, shifted properly to consider the right bits; the result so obtained is properly rounded.
For the case shown in figure 4.5, the software result is 1.5625, so, considering the 8-bit representation with only integer part, the hardware expected result is 2, that is the same value obtained from the simulation.

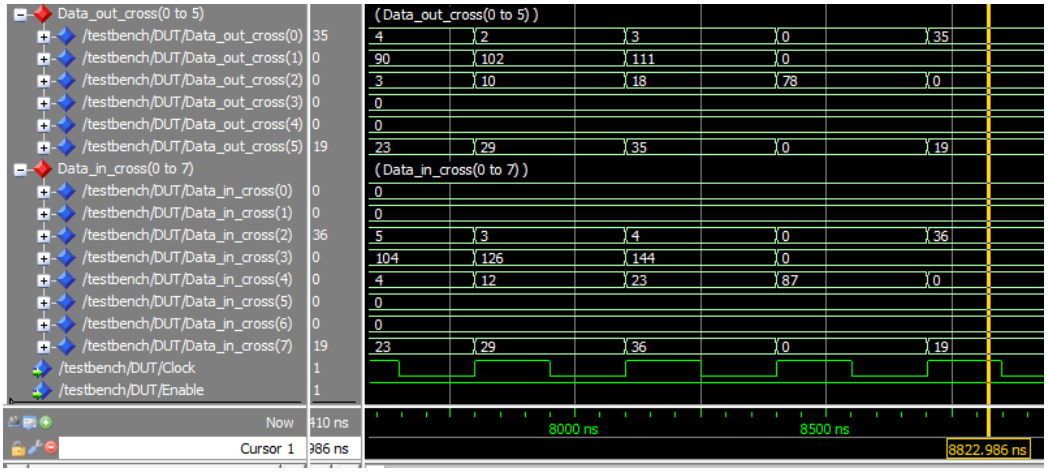
Cross-channel normalization layer

It is also important to verify the behaviour of this layer, because it is fundamental for the correct evaluation of the first and second convolutional layers. To test it, it is possible to use a specific test-bench that applies the inputs generated by the outputs of the two convolutional layers involved, comparing the outputs so obtained to the expected ones. In fact, the software used to emulate the hardware behaviour creates one .txt file for each feature map of the hidden layers, causing the possibility to access these values everywhere and always. In figure 4.6 it is shown an example

o_cross1_file.txt

```
1   0.999985   0.999985   2.99596 3.70678 4.31617 2.43744 3.07165 0   35.3157
56  0  0  0  0  0  8.98787 67.6487 89.7763 102.372 110.579 0  0   140.062
111   0  0  0  0  0  0  0  0  0  0  3.45294 9.74976 17.662  78.2806 0
166   0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  2.18315
221 0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
276 233 6.99486 5.99676 2.9996  0   12.9671 22.8192 28.6395 35.2533 0   18.884
```

(a)



(b)

results.txt

```
1  In the following there are the positions on which is present a wrong result:
2   Z) 6 Y) 54 X) 16  HW: 9 SW: 8
3   Z) 11 Y) 51 X) 10  HW: 5 SW: 4
4   Z) 15 Y) 19 X) 41  HW: 54 SW: 53
5   Z) 15 Y) 27 X) 15  HW: 43 SW: 42
6   Z) 15 Y) 32 X) 27  HW: 54 SW: 53
7   Z) 20 Y) 3 X) 27  HW: 33 SW: 32
8   Z) 34 Y) 4 X) 28  HW: 54 SW: 53
9   Z) 36 Y) 44 X) 9  HW: 27 SW: 26
10  Z) 52 Y) 43 X) 25  HW: 86 SW: 85
11  Z) 55 Y) 22 X) 26  HW: 54 SW: 53
12  Z) 65 Y) 20 X) 6  HW: 43 SW: 42
13  Z) 66 Y) 24 X) 15  HW: 40 SW: 39
14  Z) 77 Y) 13 X) 8  HW: 122 SW: 123
15  Z) 82 Y) 2 X) 38  HW: 79 SW: 78
16  Z) 89 Y) 8 X) 45  HW: 54 SW: 53
17 Total errors: 15
```

(c)

Figure 4.6: Cross-Channel Normalization Layer waveform example: (a) Software (b) Hardware (c) Test-bench used as debug instrument

of waveform matching between software and hardware and a case in which the test-bench is used as debug instrument to discover bugs, generating an output file containing information about the mismatching results.
In particular, the figure 4.6a is a collage of the first row of the first 6 channels that are part of the convolutional layer 1; the row portion shown corresponds to the same simulation steps analysed by the hardware in figure 4.6b. Moreover, in this last image it can be seen how the input data are modified by the cross-channel normalization and the presence of two additional initial values fixed to '0', to allow the normalization for the first two channels, that require respectively two and one values before them that aren't present, so in this way this possible problem is solved.

4.2 In/Out-Block

The testing of the In/Out Block has to be divided into the verification of each feature it implements, that are the following:

- The input loading, needed to initialize the memory before doing the computation;
- The storage of the intermediate results;
- The zero-padding operation, required for some layers;
- The Max-Pooling operation, needed for certain layers;
- The stride operation, that is the step on which the subsets of values are selected from the original input pattern;
- The output selection, that is the operation to put out the values required sequentially to cover all the current sub-set.

Initial input loading

This is the first phase needed to use correctly the In/Out-block, because the initial state of its internal memory is not known. For this reason, it is important to apply an initial reset, acting on a dedicated input signal, that forces all the bits to the

'0' null value. After doing this, it is possible to load into the memory the input pattern to classify. This operation requires to select the bank on which to write before starting writing and then, for each clock period, it is possible to change the input, that will be written into the address specified on the correspondent input. In figure 4.7 it is shown an example of the scenario described.

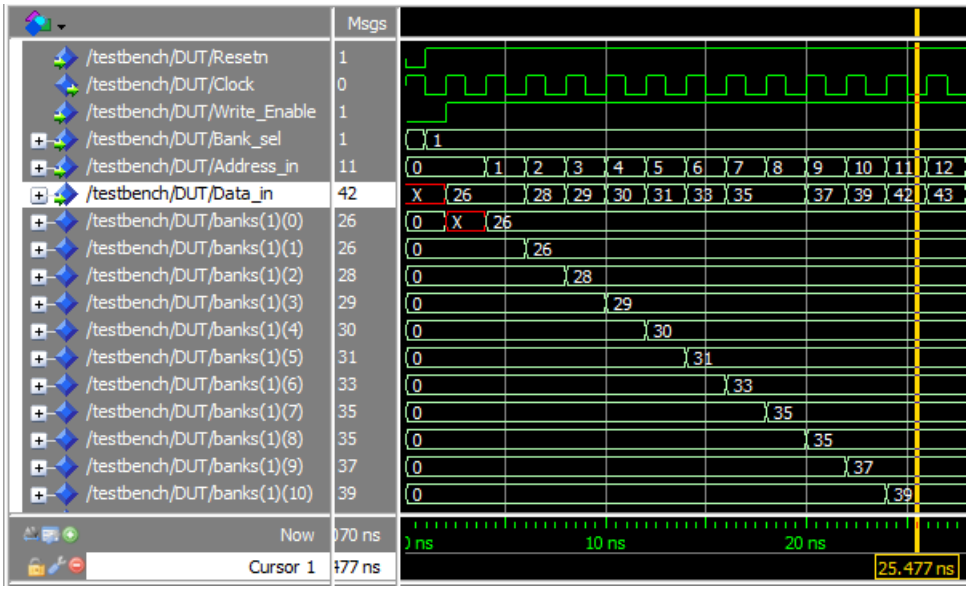


Figure 4.7: Example of an input loading phase for the In/Out Block RAM

Hidden layers result storage

An important feature to test is the correct storage of the inputs provided by the Weight-Block, because they are the inputs for the next layer it will evaluate.
In figure 4.8 they are reported pieces of the In/Out Block RAM with two sequential writing step applied: it is possible to see that the second step puts the values near the first ones; the input is maintained the same to obtain a more readable picture.
The addresses on which the first writing phase puts the input values are reported in table 4.1 and are obtained from the first layer parameters according to the formula:

$$Address = X_{size} \cdot Y_{size} \cdot FilterNumber$$

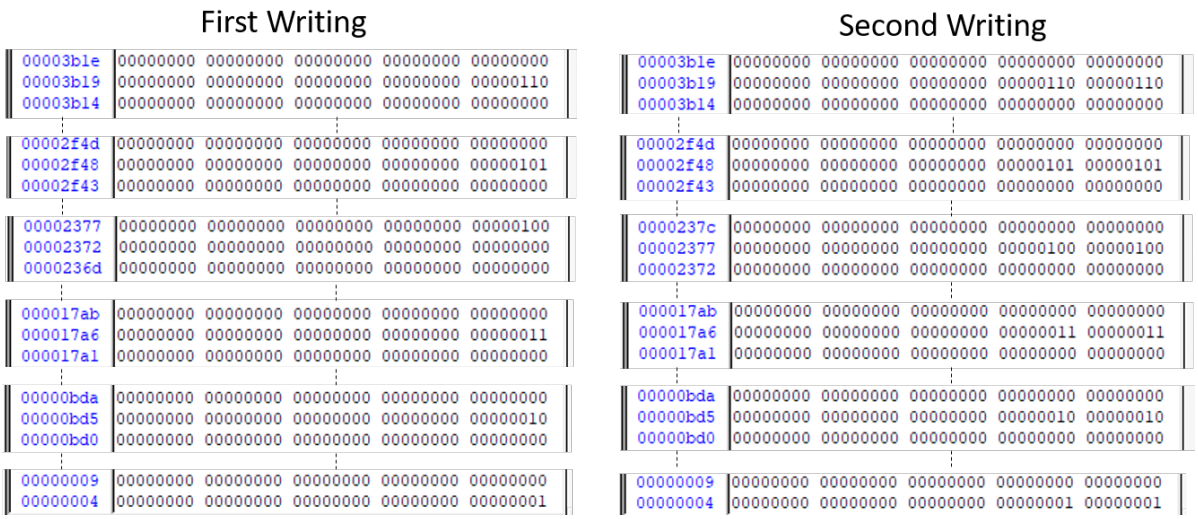


Figure 4.8: Example of the writing phase for the first hidden layer

For the first layer the formula becomes:

$$Address = 55 \cdot 55 \cdot N$$

To obtain the address for the next steps is needed to sum the current step to the base address obtained by the formula.

Filter Number	Memory Location [decimal]	Memory Location [hexadecimal]
0	0	0
1	3025	bd1
2	6050	17a2
3	9075	2373
4	12100	2f44
5	15125	3b15

Table 4.1: Memory base address for the first 6 filters

Zero-padding operation

The zero-padding operation consists to add a frame of zeros around the input pattern to warranty that the output of a certain layer has the specified size wanted. To know more details about this operation see Appendix A.

Figure 4.9 shows an example done on the first layer adding a frame of one null value

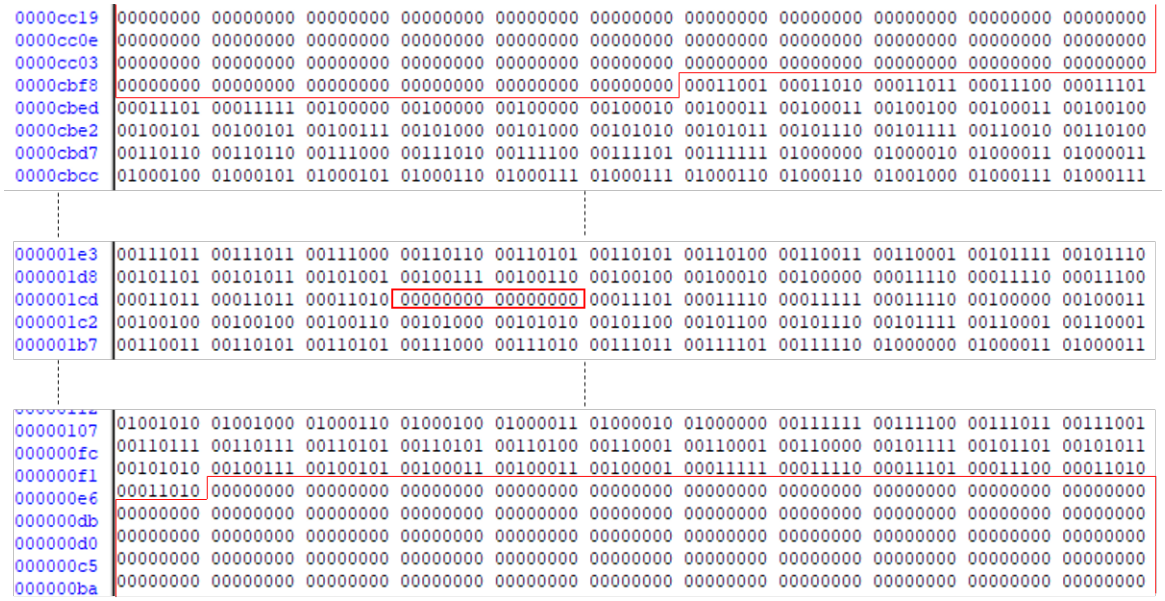


Figure 4.9: Zero-padding example with a frame of one '0'

around the input pattern; in the picture there are three situation analysed:

- The downer block of values represents the memory situation between the first memory address and the first input value; a long sequence of zeros is added in order to apply the zero-padding, and the first input is put after the last of them, into the address 230 ($e6_{hex}$), obtained by the fact that one row of size 229 (the first input block size plus the column zero frame, composed by one value at the beginning of the row and another one at the end) is added at the beginning of the pattern.

- The central block of values, representing the situation between two input rows, on which are present the last number of the previous one, that is the '0' of the zero-padding frame, and the first number of the next one, also zero due to the application of the zero-padding frame.

- The upper block of values shows the zero-padding frame added at the end of the input pattern block, that respects the same behaviour used by the initial frame block.

Max-Pooling operation

The Max-Pooling operation is required after certain convolutional layers to reduce the amount of output data without loss information about them; in Appendix A are available more details about this process.

In figure 4.10 it is reported an example of pooling for different input channels: the

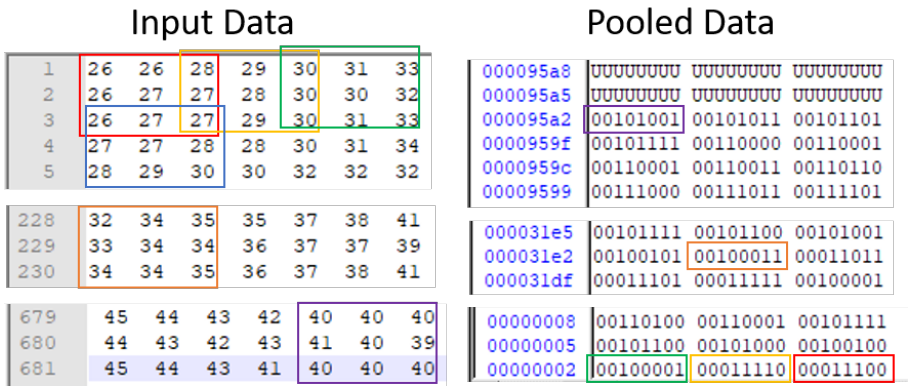


Figure 4.10: Max pooling example

left part of the image shows the inputs taken in different parts of the pattern, the right part shows the correspondent pooling outputs, ordered by the address. The input pattern for the example is the input of the NN, so it has a size of 227x227x3. To test the behaviour of the pooling in different situation, six blocks of data are analysed:

- the red one is the first block on which the pooling is applied; the maximum of them is 28, that is the one written in binary (00011100) in the right part of the picture.
- the yellow square put in evidence the second block of data, that starts two values after the first one due to the pooling stride factor (that is two); the maximum value is 30, that is the one reported in binary (00011110) in the second location of the right picture.
- the green section contains the third block analysed by the pooling, on which the output value has to be 33 (in binary 00100001).

- the blue block is the first pooling block on which the stride is applied on the Y direction, that corresponds to the first pooled value of the second row.

- the orange block is the first block of the second input channel, with a maximum value of 35, stored into the address 31e1 (12769_{dec}), obtained by the formula:

$$(\frac{X_{size} - Pool_{size}}{Pool_{stride}}+1)\cdot(\frac{Y_{size} - Pool_{size}}{Pool_{stride}}+1) = (\frac{227-3}{2}+1)\cdot(\frac{227-3}{2}+1) = 12769$$

- the violet one is the last block analysed of the last input channel, that has the maximum value equal to 41 (in binary 00101001).

Stride verification

Another important feature to test consists into the stride, that is the step characterized by two sequential subset of values extracted from the original input pattern to obtain the respective convolutional output. In figure 4.11 are shown the first

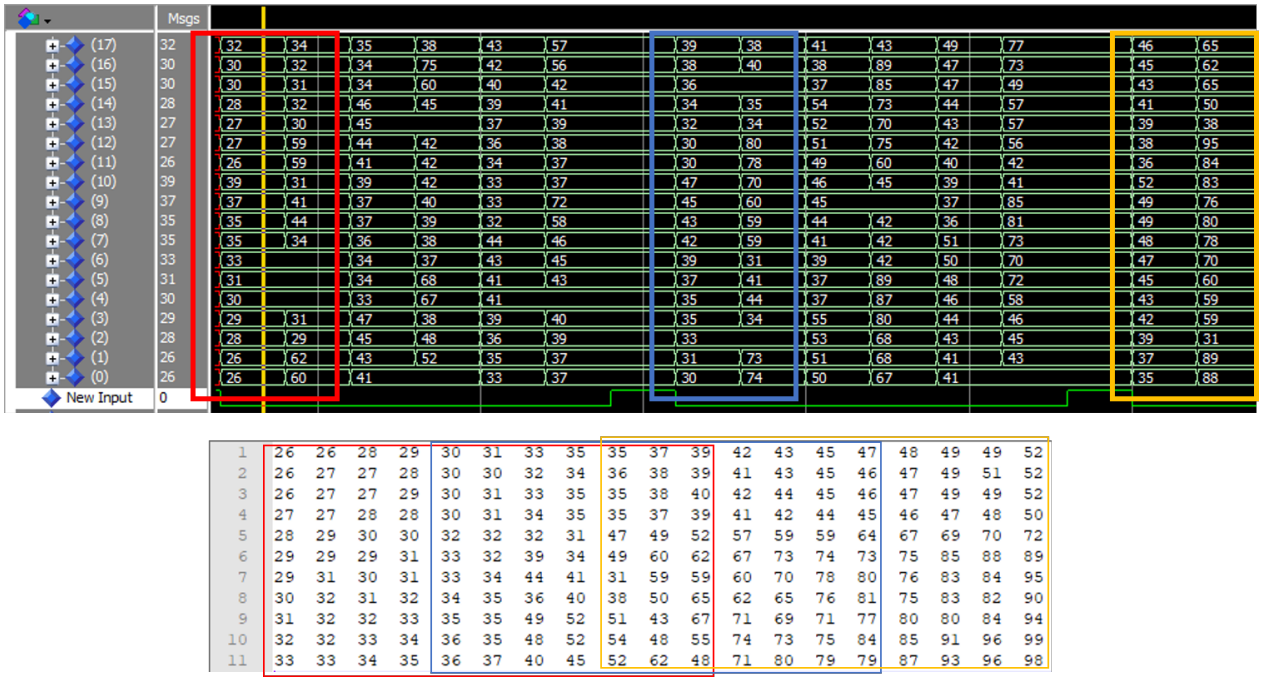


Figure 4.11: Sequential outputs of the In/Out Block: the squared blocks are the sequential subsets

eighteen outputs of the In/Out-Block and the *New Input* signal, that establishes that a new input is available and ready to process. Moreover, the picture shows the first subsets analysed, that are of dimension 11x11 and they are divided between them with a stride of four.

Outputs selection

The last feature to test consists in the fact that a generic input of the Weight-Block is typically too much big to be applied in one step, so it has to be divided into subsets used as inputs sequentially.

Figure 4.12 shows the first input of the first layer: the input channels are three and

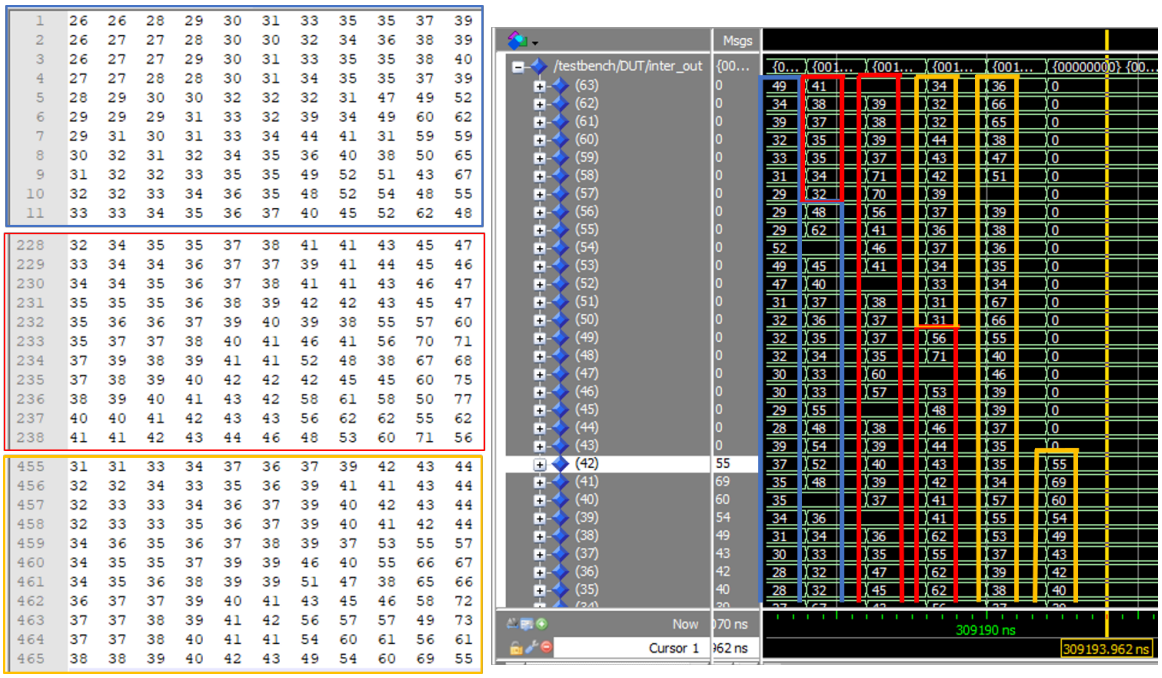


Figure 4.12: Outputs of a single evaluation: on the left there are the three input channel values, on the right the division in sequential subsets

the filter size is 11x11, so there are 363 input values to be evaluated. These inputs are divided into blocks of 64 data, that is the input parallelism of the Weight-Block, and the inputs not used in the last step are put to a null value.

Chapter 5

Synthesis

The synthesis step comes next to the simulation one because, after validating the architecture produced, it is important to evaluate the main parameters that characterize it: power, frequency and area. To obtain this purpose, it is needed a software capable to analyse the VHDL code previously produced and transform it into basic cells with known parameters; this tool, combining them together, can provide the quantities wanted as sum of the contribution of these blocks.
The program used is the *F-2011.09-SP3* version of Design Vision, a Synopsys tool with many useful features for the synthesis of digital circuits.

In figure 5.1 is represented the process flow used to make the synthesis. The compilation of the design is the first step, so it is needed a library to fix the technology node used on the analyses; the library used for this purpose is the Nangate 45nm one.
An important consideration to be evidenced is that the architecture proposed is a very large circuit, so it wasn't possible to synthesize it as it is, due to some constraints:

- the server used has got 128GB RAM, that is not large enough to load all the entire design;
- the performance are limited to the capability of the system, so a very large synthesis can use a very large time to be concluded.

For these reasons, the approach used is a bottom up process: all the smallest components are synthesised before, then the results so obtained are used to make the synthesis of the blocks that use them. Moreover, some components are too much large to be synthesised themselves, so, finally, for these blocks it is produced a result on a reduced version and then it is made a forecast of the expected behaviour. The results are generated combining together all the information so extracted.

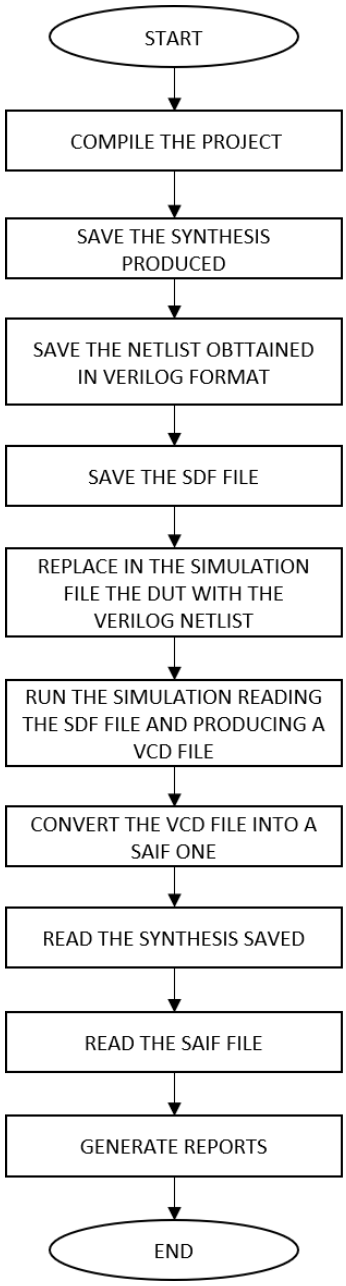


Figure 5.1: Synthesis flow-chart

Chapter 6

Results

6.1 Comparison between Adder-tree solution and a standard one

In order to establish if the adder-reversed tree solution is an improvement of the standard architectures, made up by adders not combined together with this topology, a generic neuron, that implements a convolution of 363 inputs with the respective weights, is generated using the two solutions. The two versions of this component are been synthesised to extract the parameters of power, area and critical path delay, useful to compare them.

Figure 6.1 shows the three quantities analysed for the standard version and the

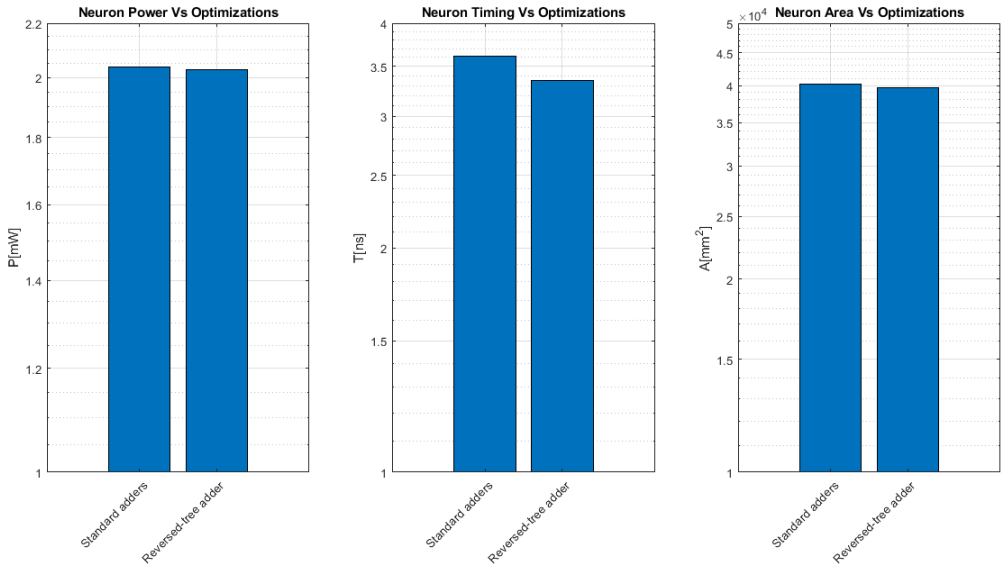


Figure 6.1: Comparison between a conventional solution and a reversed-tree one

adder-tree one: it is possible to see that the last solution has improvements on all the

parameters, so it can be concluded that it is a good optimization for the architecture proposed.

6.2 performance of the WINNER architecture

Power and Area Weight-Block estimation

The memory of the Weight-Block is a critical point during the synthesize step because it is a very large component: the Design Vision tool has got a limit to the number of iteration loop on the same object; this value can be increased, but the synthesis becomes so long and big in terms of RAM space that it is not possible to see the results before a crash of the software.
To avoid this problem, the parameters of this block are estimated for a different number of word Bytes and then the results are interpolated to provide a forecast of the expected behaviour. Figure 6.2 shows the power and area quantities for a neuron

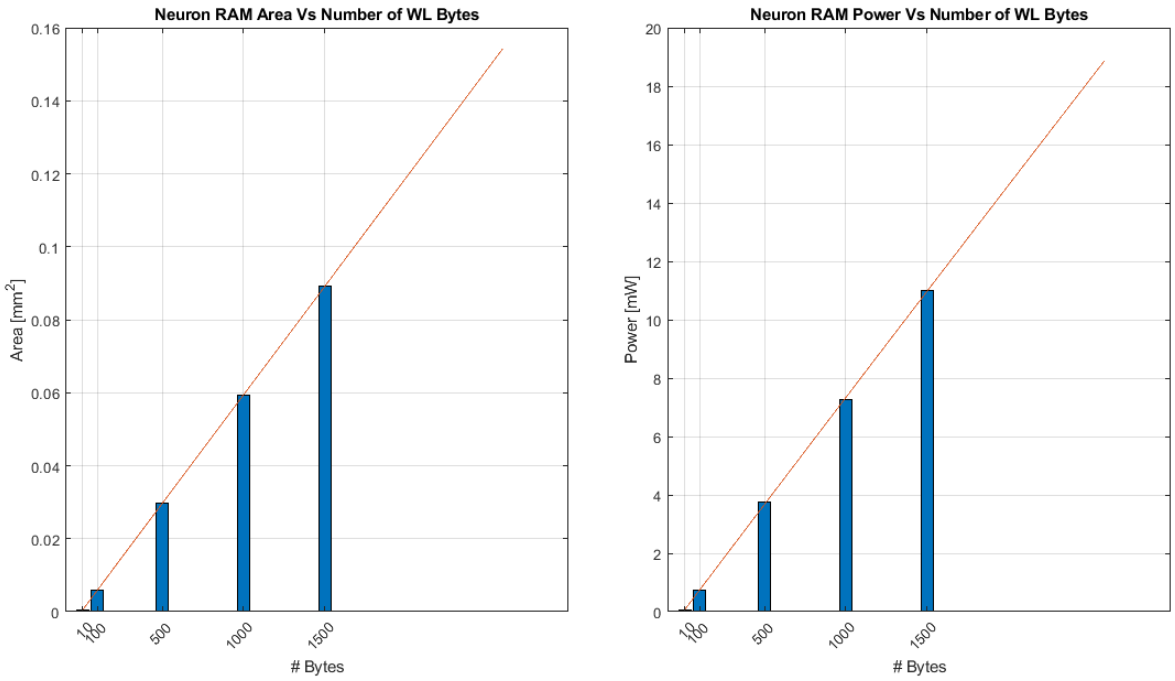


Figure 6.2: Power ans area forecasts for the RAM of single neuron

changing the word size: the columns in blue represent the results of the synthesis

made, the red line is the interpolation of them, used to provide the forecast.

In/Out Block

In figure 6.3 are reported the main contributions to the area and power into the In/Out-Block: the most important parts are the logic, composed by the circuit that establishes which outputs in which moment have to be selected, and the comparators, that forms together the pooling layer, used only after certain convolutional layers.

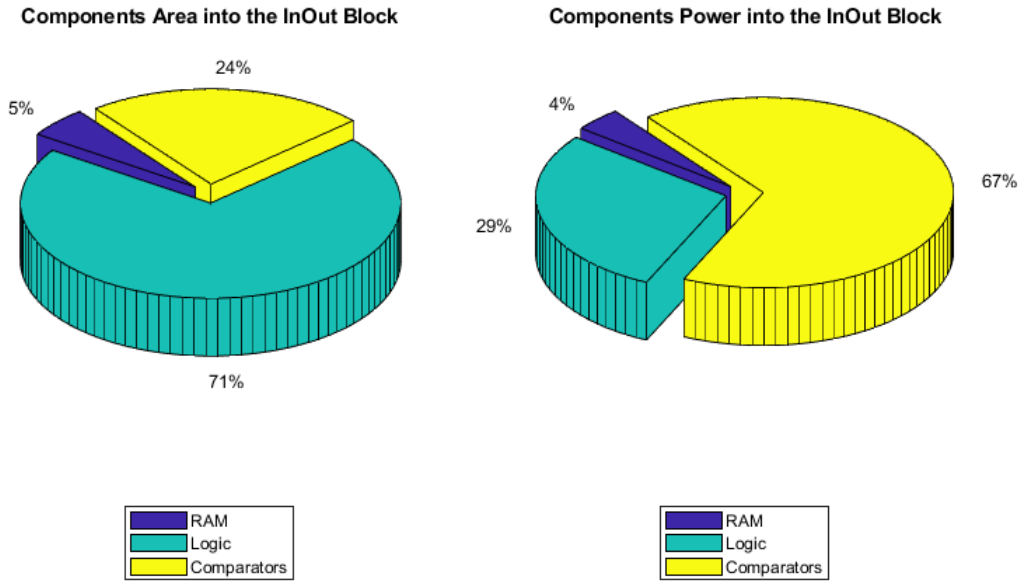


Figure 6.3: Power and Area of the In/Out Block internal components

MUX Vs Wired-OR Neuron

In order to establish if the optimization of the selectors is relevant or not into the architecture proposed, the power and area parameters of a neuron of the WINNER architecture realized with the two alternatives were analysed to be compared.

Figure 6.4 shows the contribution of the Weight-Block internal blocks to the total power of their neuron; in the following some observations are reported:

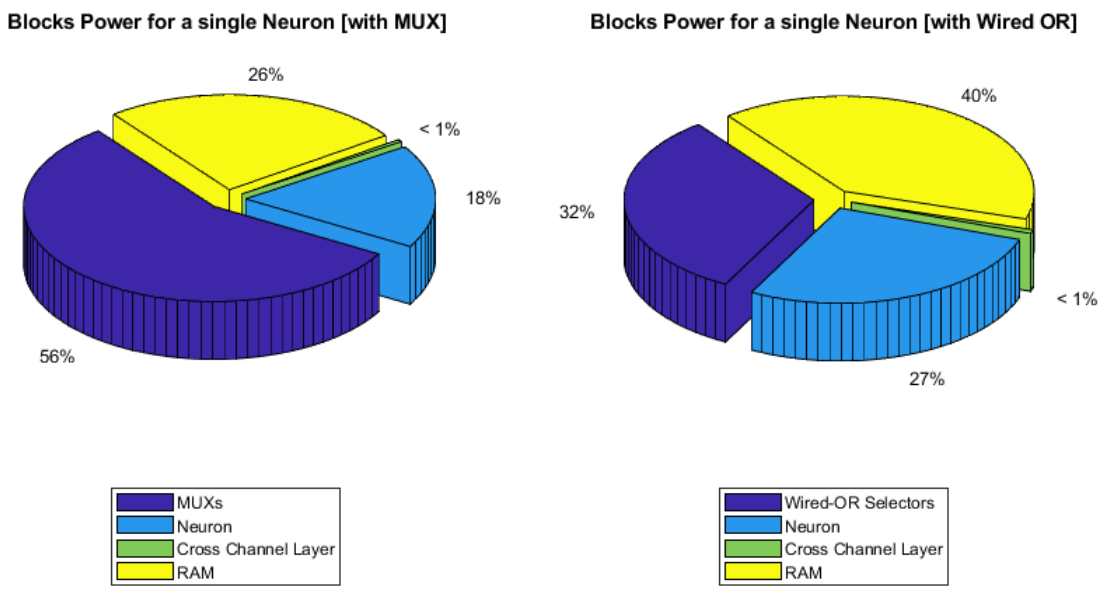


Figure 6.4: Power of the internal blocks of a neuron

- the contribution of the cross-channel layer is very small; this block is synthesised separately from the Weight-Block because of the Design Vision constraints already discussed, and the results obtained are divided for the number of neuron implemented to obtain the cost for the single one, that is the quantity reported into the picture.

- the main contribution in the MUX solution is made up by the MUX selectors, that are very big, due to the fact that have 2595 inputs; on the contrary, the main contribution with the wired-OR solution is made up by the memory, because this solution uses only one bitline decoder for all the words (implementing MUXs there is one decoder for each of them), reducing the hardware cost.

- the wired-OR solution uses an OR-chain to emulate, for synthesis requirements, the behaviour of a single wire with a transistor for each bit cell connected to it and a sense amplifier at the end of it, so the real parameters of this blocks are better then what is reported in the picture, that is an upper bound of it.

- the neuron has got a relevant part in terms of power contribution because it has many adders always working.

- the RAM power is the static power of the RAM, because the usage of the architecture consists into load one time the memory, consuming in this phase static and dynamic power, and then, without changing the memory values, make the computations, so there isn't any dynamic power dissipation during the normal working of the memory.

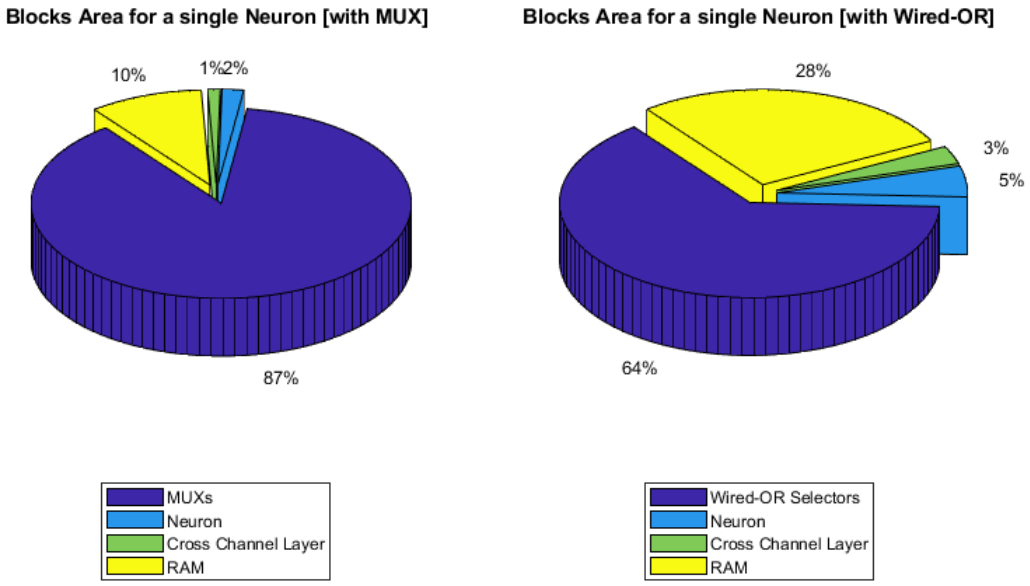


Figure 6.5: Area of the internal blocks of a neuron

Figure 6.5 shows the area contribution of the internal blocks of a neuron; the observations on how these results are provided are the same described for the power estimation. By the graph it can be seen that the main contribution is the selectors one; on the contrary, wired-OR solution this contribution is reduced.

Figure 6.6 shows the absolute values of power and area for the two solutions analysed: both the quantities are lower for the wired-OR implementation, so it can be concluded that this is a good optimization for the architecture proposed.

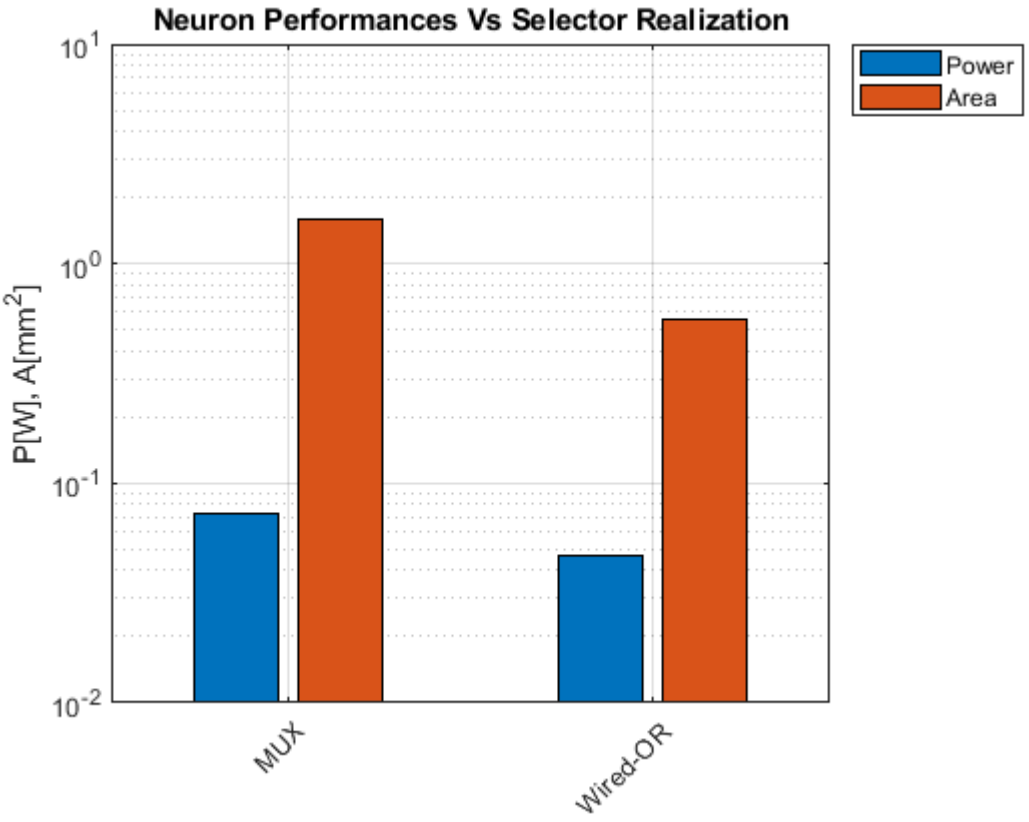


Figure 6.6: Power ans Area for a single neuron changing the type of selectors

Frame Per Second (FPS)

The Frame Per Second parameter is a very important quantity for convolutional NN as the AlexNet implemented, because it indicates how much input patterns can be classified in one second. In the case of the WINNER architecture, considering the clock with the same time period of the critical path, this parameter becomes:

$$FPS = \frac{1}{TOT_{steps} + S_{L1} \cdot (N_{ClkCross} - S_{N1}) + S_{L2} \cdot (N_{ClkCross} - S_{N2}) \cdot T_{clk}}$$

$$FPS = \frac{1}{(52902 + 3025 \cdot (51 - 6) + 729 \cdot (51 - 19)) \cdot 3.55 \cdot 10^{-9}} = 1326.5FPS$$

Where S indicates the number of steps for the execution of the layer number x (if the subscript is Lx) or the execution of a neuron of the layer number x (if the subscript

is Nx), $N_{ClkCross}$ the clock period required for the cross channel normalization layer to be executed and T_{clk} the clock period.
The number of clock periods used by the Cross Channel Normalization Layer is obtained by its critical path and it is equal to 51 clock periods; this layer is used only after the first two convolutional layers, so to the total number of steps required for the entire NN it has been added these two terms that consider the normalization layer. All the other quantities are extracted from table 3.2.

Power, Frequency and Area comparison

In order to compare the global performance of the WINNER architecture with the other technologies implementations analysed in the chapter , it is reported the same graph proposed at the end of it in figure 6.7, with the additional WINNER architecture results.

The WINNER architecture (the one evidenced with a red rectangular) has got a

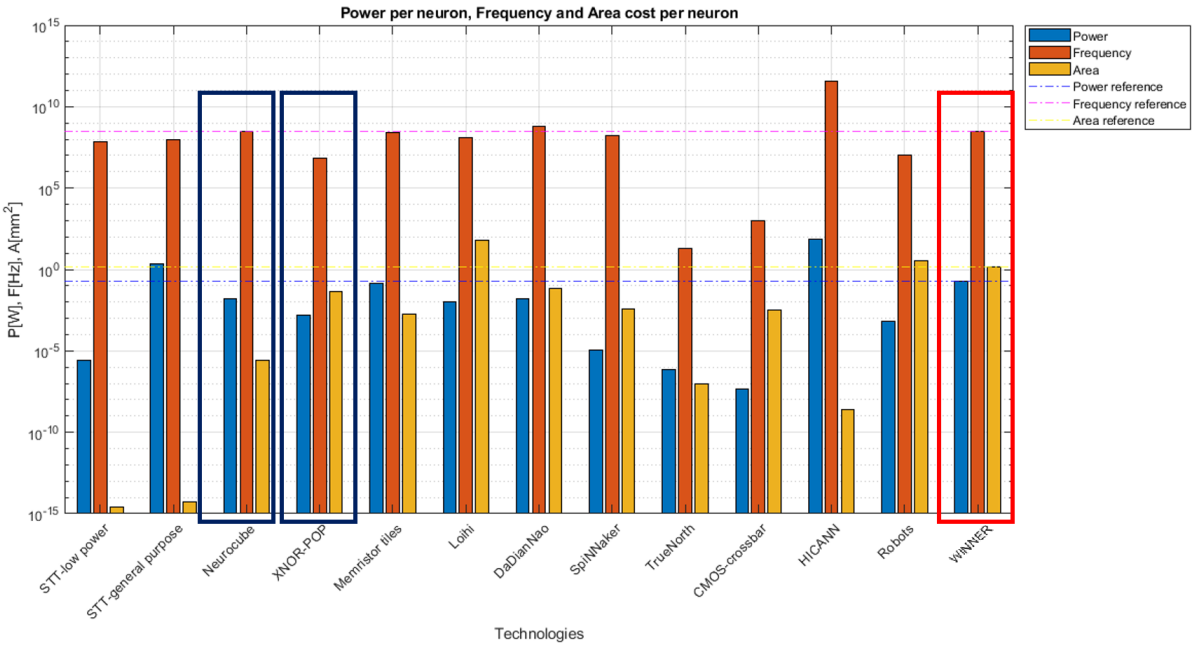


Figure 6.7: Power, Area and Timing comparison for a single neuron using a Wired OR selector

good frequency and a higher value of area and power respect on the most part of the

other solutions. The higher value of power and frequency are caused by the difference between the technology node on which are realized, the FPS reached and the type of NN that can be implemented, that in most cases have limited size. The reference architecture for the WINNER are the ones evidenced with a blue rectangular, because they have the same type of technology: in-memory CMOS.

EAT and ET comparison

Another comparison that can be done, using the results shown in figure 6.7, is the comparison of the energy-delay product and the energy-area-delay one; these quantities are good indexes of how much the solution proposed is good, because provide data that combine together all the performance information.

Figure 6.8 shows with a red rectangular the WINNER results and with blue ones the

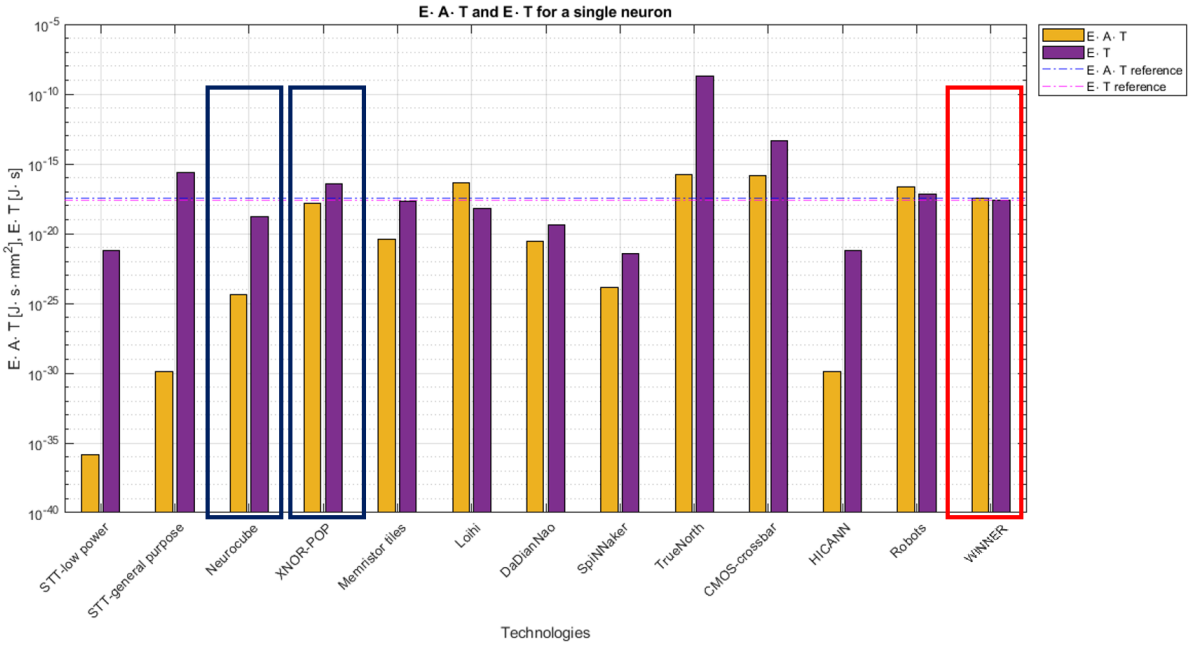


Figure 6.8: EAT and ET comparison for a single neuron using a Wired OR selector

Neurocube and XNOR-POP results, that are the main reference for the WINNER architecture, as already discussed. It can be seen that the two levels of EAT and ET are comparable to the XNOR-POP ones; remembering that these two solutions are realized with a different technology node, littler for the XNOR-POP, it is a good

result because, scaling the technology, the expected behaviour is an improving of the performance. The Neurocube has got better EAT and ET because it uses the HMC [7], that is a general purpose in memory device, to implement NNs, so the performance are good but the speed in terms of FPS is very low.

Energy per Frame

In order to obtain an application-independent comparison between the architecture realized and the other ones that are implemented using a Logic In Memory approach, the Energy per Frame quantities is analysed. Figure 6.9 shows the parameter described in association with the FPS and the accuracy reached by the network implemented, that is the AlexNet for all the three solutions reported. It is possible to see that the XNOR-POP has got the lowest value of $E \cdot Frame$, but implements an approximated version of the AlexNet, with an accuracy in the image recognition that is around 10% lower than the exact one. Moreover, the picture shows the technology node, that is very important to consider, because, reducing it, all the parameters previously analysed improve, generating a better behaviour.

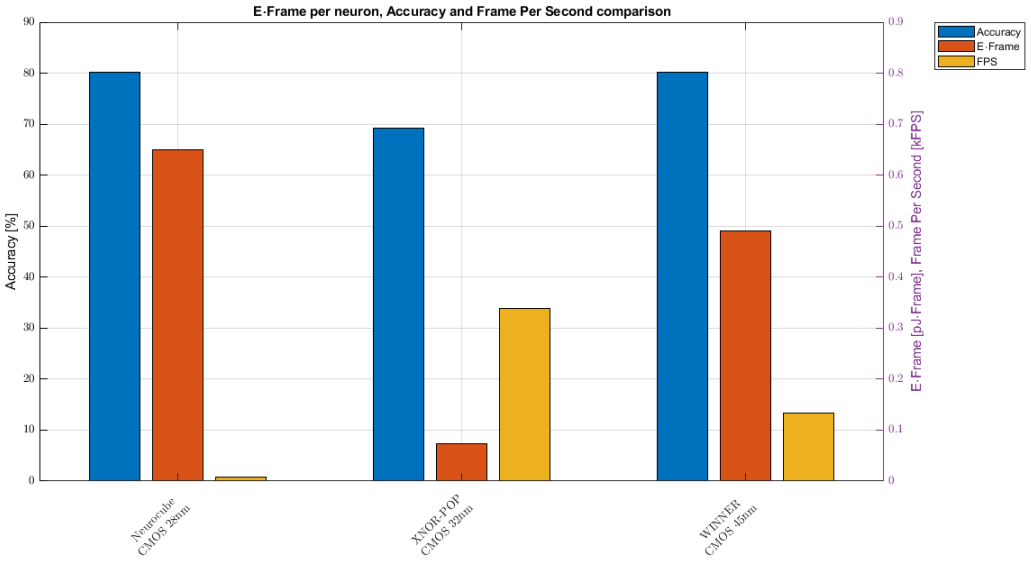


Figure 6.9: $E \cdot Frame$ comparison

Chapter 7

Conclusions and Future Works

The project presented is a starting point for a new way to implement Neural Networks: the combination between them and the Processing In Memory principle provides a good support to realize each kind of NN and make the architecture obtained easy to scale. Moreover, the PIM solutions, compared to the other ones, typically provide circuits with better FPS and with a higher capability to implement big NN. The comparison on the other main actual solutions shows that the WINNER architecture has the same advantages of a XNOR-POP one, but using a larger technology and implementing an exact computation of the network; in fact, the XNOR-POP adopts the pop-counting approximation to make the convolution operation. These differences are very important because a comparison between the WINNER and the XNOR-POP with the same behaviour would show different behaviours from what seen.
Considering the observations presented above, it is possible to understand that there are a lot of possible improvements, analyses and works starting from the WINNER architecture; the main ones are:

- the synthesis with a different library that uses smaller transistors, that could improve all the performance of power, area, frequency and FPS.
- the study of what happens in the WINNER architecture with the substitution of multipliers with approximated versions of them; this is a very interesting point because NNs have a good tolerance of the input noise, so make an approximated computation has the same result of increasing the input pattern noise, and can provide a solution with better performance.
- to realize a similar architecture with a different kind of technology, as MTJs, that seems to be a low power technology, but, for now, with reduced possibilities to implement a big hardware.

- the realization of a hybrid architecture that uses Memristors or MTJs to realize the part of memory and CMOS near it to make a part or all the evaluation process, realizing a PIM hybrid solution.

Appendix A

AlexNet Neural Network

AlexNet is a CNN developed by Alex Krizhevsky, who won in 2012 the ImageNet Large Scale Visual Recognition Challenge, a competition to produce the best image recognition NN. This network has got a very big topology and a large quantity of neurons, due to its depth, causing very good results but with an expensive computational cost. A graphical representation of the NN is reported in figure A.1, on which it can be seen that the network is made up by 8 main layers: the first five are convolutional, the last three fully-connected. Moreover, the input images to be processed must have a 227*227 size, with the three color channel (RGB).

In the following it is described in details the default Matlab implementation of the

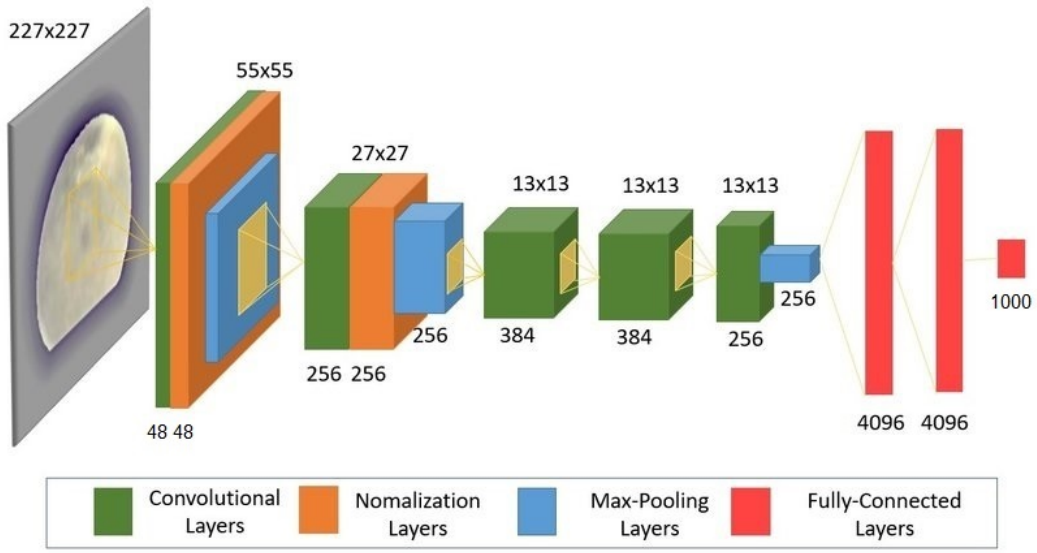


Figure A.1: AlexNet graphical representation. Image readapted from [24]

AlexNet, on which the first and the second convolutional layers are followed by a

normalization layer and a pooling one, in order to uniform the values and reduce the number of inter-layer results; also the last convolutional layer is followed by a pooling layer, and the fully-connected layers have got internally a dropout layer, that is useful during training to reach the result wanted in a faster way. The last FC layer has a softmax layer instead of dropout in order to obtain an output probability linked to the result provided.

A.1 Convolutional Layers

They apply a certain number of filters to subsets of the input pattern with the dimensions of the filter size, moving them with a specific stride to cover all the input set. For each of these movements each filter provides an output, that becomes the input for the next layer. Some layers consider for each filter all the input channels, others only a subset; typically the possibilities are to consider all the input channels in one block or in two. Moreover it is possible to add a frame of r rows and c columns of zeros to the input pattern, in order to maintain the output size equal to the input one, without information loss.
In figure A.2 are shown the graphical quantities described, and in table A.1 are reported the relative values for each convolutional layer.

# Layer	# Filters	$X_{filtersize}$	$Y_{filtersize}$	Stride	Padding	# Filter Blocks
1	96	11	11	4	0	1
2	256	5	5	1	2	2
3	384	3	3	1	1	1
4	384	3	3	1	1	2
5	256	3	3	1	1	2

Table A.1: Convoutional parameters for each layer

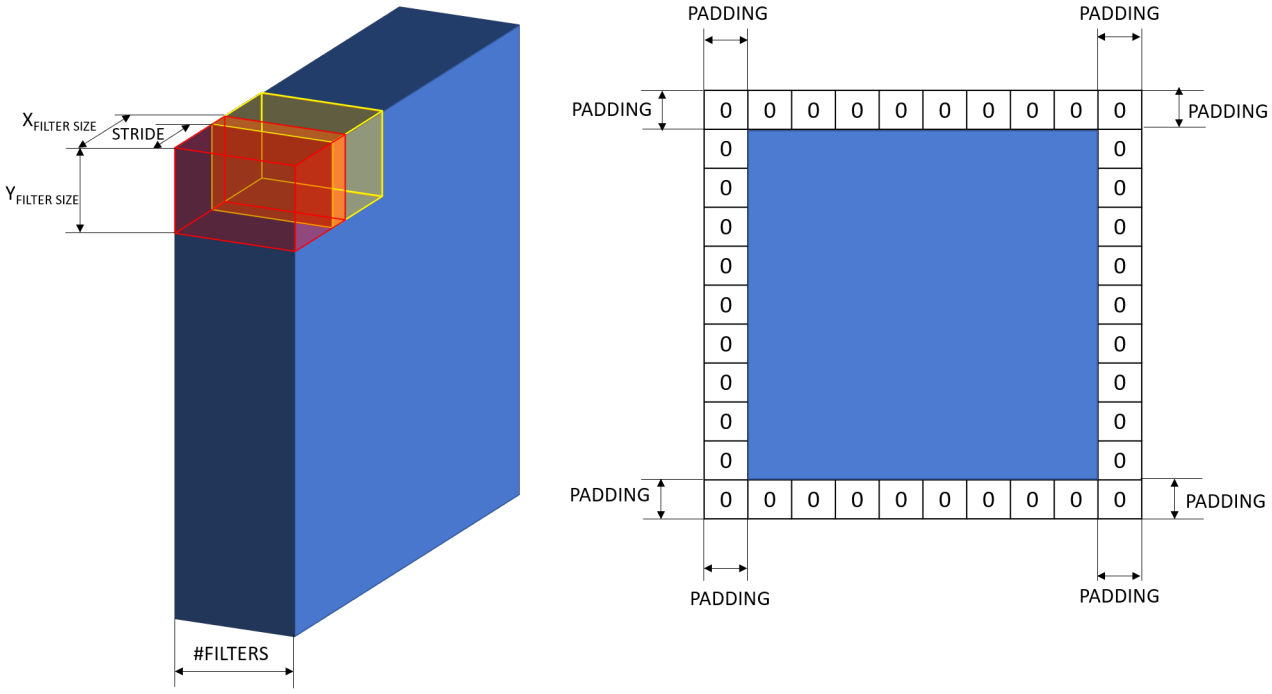


Figure A.2: Graphical representation of the convolutional parameters

A.2 Cross-Channel Normalization Layers

These layers are normalization layers, so they consider a series of input values and provide the same amount of results changed to make them more uniform. This evaluation considers the inputs with same position, but filtered by adjacent filters; for this reason it is called "cross-channel". The formula they use to make this normalization depends on the parameters α , β , K, that are real numbers set to 0.0001, 0.75 and 1 in AlexNet, and the window channel size, that means how much input channel to consider in the computation of an output, that is 5 for AlexNet:

$$y = \frac{x}{(K + \alpha \cdot \sum_{i=1}^{n} x_i^2)^\beta}$$

A.3 Max-Pooling Layers

The pooling operation consists on replacing a matrix of input values with the highest of them; this allows the computation to be less expensive and to put more in evidence the characteristics found by the filters applied. In fact, this step acts separately for each input channel, so it consider only groups of number extracted by the same previously filter. Moreover, it scans the input pattern with a fixed size and stride, as described for convolutional layers A.1. Figure A.3 shows an example on how a pooling layer acts on an input matrix; in AlexNet these parameters are:

- Pooling size = 3x3;
- Pooling stride = 2.

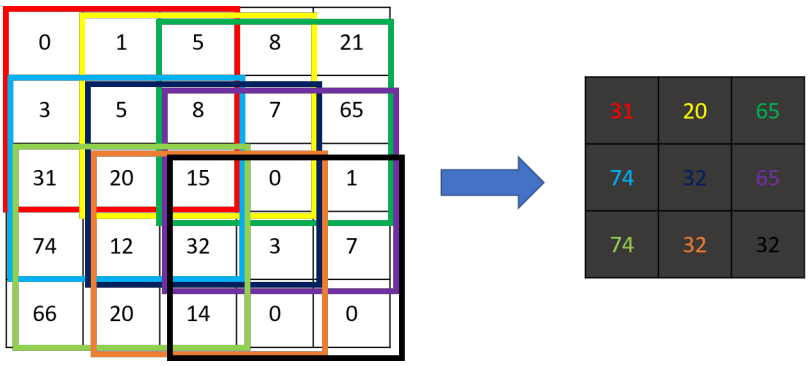


Figure A.3: Pooling operation on an input matrix of 5x5 with pooling size 3x3 and stride = 1, generating an output matrix of 3x3

A.4 FC Layers

These are the last three layers of the AlexNet, producing the classification result; in table A.2 they are reported the inputs amount and the number of neurons for each of them.

# FC layer	# inputs	# neurons
1	43264	4096
2	4096	4096
3	4096	1000

Table A.2: Inputs and neurons number for each FC layer on AlexNet

A.5 Dropout Layers

The dropout operation that is implemented in this layers consists in putting to '0' some inputs with a certain probability; this layer is used only during the training to reach the wanted result in a faster way. In the AlexNet analysed the probability parameter is set to 0.5, so half of the inputs are '0' during training.

A.6 Softmax Layer

This layer is the last one of the AlexNet and implements a softmax function, that is able to transform the output classification in a probability; this is useful in the cases on which the interest is on the reliability of the result produced, but in other cases it is required to know only the class identified, so this layer is not implemented in all applications. The softmax function consists in the formula below, on which k is the number of output classifiers:

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

Appendix B

Modified Booth Encoder Multiplier

MBE based multipliers are based on two parts: the first is used to obtain the partial products from the multiplicands and the second one to add these in an array of full adders.

B.1 Partial Product block

In order to obtain partial product, there is a specific unit that takes the two input numbers and, slicing the first one, makes the product between the slices and the second operand, generating the partial results to be added.
MBE multipliers are based on a Radix-2 approach, meaning that 3-bit slices on the first operand have to be analyzed, where two consecutive slices feature a 1-bit overlap. In this case, the number of bits used to represent the values is even (8), so the triplets of bits, that start with a first '0' added before the LSB, are completed. Vice-versa, in case of odd number of bits, a sign extension could be required.
Considering that the first triplet starts with an additional LSB of value '0', the number of slices obtained is 4, as shown in figure B.1 These triplets are used to

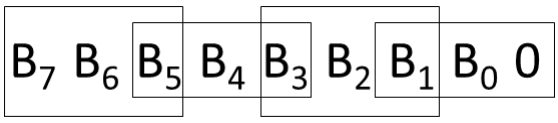


Figure B.1: MBE triplets generation

associate with a BEU (Booth Encoding Unit) the input multiplicand to its original value A, its complemented value $-A$, the doubles $2A$ and $-2A$ or zero, according

to the bits of the triplet, to realize a partial multiplication.
In table B.1 you can see how this associations are made, on which A is the second operand.

X_{j+1}	X_j	X_{j-1}	Operation
0	0	0	0
0	0	1	+A
0	1	0	+A
0	1	1	+2A
1	0	0	-2A
1	0	1	-A
1	1	0	-A
1	1	1	0

Table B.1: BEU truth table

Appendix C

Approximations with Taylor's series expansions

The Taylor's series expansion is a mathematical instrument to represent complex functions with a polynomial with a fixed degree: the higher it is, the more accurate is the approximation.
The result of this process is a new function that is valid only locally, so around a point fixed before starting the process. The starting function has to got some properties in order to apply the series expansion:

- continuous in the point x_0 on which the process is applied;
- derivable in the point x_0;
- derivable near the point x_0.

If these requirements are satisfied, the function can be rewritten using the equation of a polynomial with coefficients that are the n-order derivative evaluated in the point x_0 for the x^n coefficient:

$$y = \sum_{k=0}^{n} \frac{f^{(k)}(x_0)}{k!} \cdot (x - x_0)^k$$

In this way is possible to approximate some complex functions, used in Neural Network for the hidden layer results, in simpler ones, as for the exponent function described in appendix A to realize the cross-channel normalization, that becomes:

$$(1 + x)^\beta = 1 + \beta \cdot x + \frac{\beta \cdot (\beta - 1)}{2} \cdot x^2$$

Bibliography

[1] Alianna J. Maren, Craig T. Harston and Robert M. Pap,"Handbook of Neural Computing Applications", Academic Press, 1990.

[2] D. Zhang et al., "Energy-efficient neuromorphic computation based on compound spin synapse with stochastic learning," 2015 IEEE International Symposium on Circuits and Systems (ISCAS), Lisbon, 2015, pp. 1538.

[3] E. I. Vatajelu, L. Anghel, "Fully-connected single-layer STT-MTJ-based spiking neural network under process variability", *Proc. IEEE/ACM Int. Symp. Nanosc. Archit. (NANOARCH)*, pp. 21-26, Jul. 2017.

[4] A. Sengupta and K. Roy, "Spin-Transfer Torque Magnetic neuron for low power neuromorphic computing", *2015 International Joint Conference on Neural Networks (IJCNN)*, Killarney, 2015, pp. 1-7.

[5] L. Song, Y. Wang, Y. Han, H. Li, Y. Cheng and X. Li, "STT-RAM Buffer Design for Precision-Tunable General-Purpose Neural Network Accelerator", in *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 25, no. 4, pp. 1285-1296, April 2017.

[6] D. Zhang et al., "All Spin Artificial Neural Networks Based on Compound Spintronic Synapse and Neuron", in *IEEE Transactions on Biomedical Circuits and Systems*, vol. 10, no. 4, pp. 828-836, Aug. 2016.

[7] J. Schmidt, H. Fröning, U. Brüning. (2016). Exploring Time and Energy for Complex Accesses to a Hybrid Memory Cube. pp. 142-150.

[8] D. Kim, J. Kung, S. Chai, S. Yalamanchili and S. Mukhopadhyay, "Neurocube: A Programmable Digital Neuromorphic Architecture with High-Density 3D Memory", *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, Seoul, 2016, pp. 380-392.

[9] Y. Wang, R. Chen, R. Mao and Z. Shao, "Optimally Removing Synchronization Overhead for CNNs in 3D Neuromorphic Architecture", in *IEEE Transactions on Industrial Electronics*.

[10] L. Jiang, M. Kim, W. Wen and D. Wang, "XNOR-POP: A processing-in-memory architecture for binary Convolutional Neural Networks in Wide-IO2

DRAMs", *2017 IEEE/ACM International Symposium on Low Power Electronics and Design (ISLPED)*, Taipei, 2017, pp. 1-6.

[11] D. J. Mountain, M. R. McLean and C. D. Krieger, "Memristor Crossbar Tiles in a Flexible, General Purpose Neural Processor", in *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, vol. 8, no. 1, pp. 137-145, March 2018.

[12] L. Xia et al., "Switched by input: Power efficient structure for RRAM-based convolutional neural network", *2016 53nd ACM/EDAC/IEEE Design Automation Conference (DAC)*, Austin, TX, 2016, pp. 1-6.

[13] M. Davies et al., "Loihi: A Neuromorphic Manycore Processor with On-Chip Learning", in *IEEE Micro*, vol. 38, no. 1, pp. 82-99, January/February 2018.

[14] T. Luo et al., "DaDianNao: A Neural Network Supercomputer", in *IEEE Transactions on Computers*, vol. 66, no. 1, pp. 73-88, Jan. 1 2017.

[15] E. Painkras et al., "SpiNNaker: A 1-W 18-Core System-on-Chip for Massively-Parallel Neural Network Simulation", in *IEEE Journal of Solid-State Circuits*, vol. 48, no. 8, pp. 1943-1953, Aug. 2013.

[16] Paul A. Merolla et al., "A million spiking-neuron integrated circuit with a scalable communication network and interface", *science*, 08 aug 2014 : 668-673

[17] P. Merolla, J. Arthur, F. Akopyan, N. Imam, R. Manohar and D. S. Modha, "A digital neurosynaptic core using embedded crossbar memory with 45pJ per spike in 45nm", *2011 IEEE Custom Integrated Circuits*.

[18] Qiao Ning et al., "A reconfigurable on-line learning spiking neuromorphic processor comprising 256 neurons and 128K synapses", *Frontiers in Neuroscience 2015*.

[19] . Corradi and G. Indiveri, "A Neuromorphic Event-Based Neural Recording System for Smart Brain-Machine-Interfaces," in *IEEE Transactions on Biomedical Circuits and Systems*, vol. 9, no. 5, pp. 699-709, Oct. 2015.

[20] J. Schemmel, D. Briiderle, A. Griibl, M. Hock, K. Meier and S. Millner, "A wafer-scale neuromorphic hardware system for large-scale neural modeling", *Proceedings of 2010 IEEE International Symposium on Circuits and Systems*, Paris, 2010, pp. 1947-1950.

[21] S. Schmitt et al., "Neuromorphic hardware in the loop: Training a deep spiking network on the BrainScaleS wafer-scale system", *2017 International Joint*

Conference on Neural Networks (IJCNN), Anchorage, AK, 2017, pp. 2227-2234.

[22] B. V. Benjamin et al., "Neurogrid: A Mixed-Analog-Digital Multichip System for Large-Scale Neural Simulations", in *Proceedings of the IEEE*, vol. 102, no. 5, pp. 699-716, May 2014.

[23] A. Amravati, S. B. Nasir, S. Thangadurai, I. Yoon and A. Raychowdhury, "A 55nm time-domain mixed-signal neuromorphic accelerator with stochastic synapses and embedded reinforcement learning for autonomous micro-robots", *2018 IEEE International Solid - State Circuits Conference - (ISSCC)*, San Francisco, CA, 2018, pp. 124-126.

[24] Study and classification of plum varieties using image analysis and deep learning techniques - Scientific Figure on ResearchGate. Available from: https://www.researchgate.net/figure/Alexnet-architecture_ fig4_ 320511487 [accessed 26 Jan, 2019]