POLITECNICO DI TORINO

Master Degree in Electronic Engineer

Master Degree Thesis

Hardware Accelerator Sparsity Based for Ultra-Low Power Quantized Neural Networks based on Serial Multipliers

Supervisors

Prof. Maurizio MARTINA

Dr Francesco CONTI (ETH Zurich)

Candidate

Eleonora FERRARA

December, 2020

Summary

Nowadays, applications such as Deep Learning Enabled Internet of Things have become a new way to look to the future of monitoring, control and automation of the reality around us. In this thesis, the focus is on Edge Computing IoT, that pushes the analytic part from servers to sensors and portable devices by cutting off the need for data transmission and broad bandwidth. However, a problem has to be fixed in order to get the possibility to employ this new technology in whatever application from Industry 4.0 to biomedical, from autonomous driving to smart cities and so on. This important issue is the huge power consuming that this kind of applications requires. In the previous work related to the Multi-Precision Bit-Serial Hardware Accelerator IP for Deep Learning Enabled IoT designed in collaboration with ETH of Zurich, this aim was reached, partially, thanks to the utilization of the PULP system, elaborated from ETH of Zurich researchers, that will be introduced better further in this thesis. This open source platform supports the HWPE (Hardware Processing Engine) interface, which has been integrated as SMAC Engine yet. It exploits the cache memory hierarchy in order to reduce the latency and the power consuming related to the handling of a huge data quantity during the convolution operations of multilayer neural networks, such as CNN (Convolutional Neural Networks). The focus of this thesis is to get a Serial-MAC-engine (SMAC Engine), that is a 65 nm hardware accelerator with 8 or 4 bits parallelism for activations and 8, 6 or 4 bits for weights, optimized in terms of power consuming, paying attention to the trade off with the performances. The bit serial multiplication approach used before for this system respects the state-of-the-art low-power standards. The consumption of 0.58Pj/MAC only makes the power budget of the order of mW, suitable for IoT. However, there is the possibility to achieve a lower power consuming focusing on a particular issue that involves the matrices of data during the convolution process, that is *sparsity*. In this way, it can be good to reduce the usage of data memory during the operations using some compression techniques that will be explained further.

Acknowledgements

I am grateful to prof. Maurizio Martina and prof. Francesco Conti for giving me the opportunity to carry on this thesis work with an important technical support, thanks to their willingness to listen to my work updates and my doubts during the thesis period and to their excellent knowledge of the matter this thesis work treats.

I would like to thank also Maurizio Capra for the technical and moral help that gave me at each step of this thesis work, for each kind of doubt I had.

A very great thanks goes to my family, that was always present during both the good and the difficult moments, not only in this last thesis period but during all the university journey. They always gave me the strength to go on although the bad moments I have passed through.

I would like also to thank Francesco for giving me strength and encouragement in this last two years and during the thesis journey.

I couldn't forget the support all my friends gave me during my life and for the good moments passed together. First of all, I say thank to my schoolmates and friends Angela, Giulia, Marta, Lorenza, Salvo and Marco. They have always believed in me also when I didn't do. Then, I need to thank my lifelong friends Giulia, Sara, Veronica, Manuela, Sara because they helped me become who I am today.

Table of Contents

1 Introduction 1
- 1.1 General Principles . . . 1
- 1.2 Introduction to chapters . . . 4

2 Deep Neural Networks and Convolutional Neural Networks 5
- 2.0.1 Basic concepts . . . 5
- 2.0.2 The origins of Convolutional Neural Networks . . . 9
- 2.0.3 The Convolution Operation . . . 10
- 2.0.4 Convolutional Neural Networks basics . . . 12
- 2.0.5 Architecture of Convolutional Neural Networks . . . 13

3 Sparsity 18
- 3.1 Hardware Accelerators Sparsity Based in Convolutional Neural Networks . . . 18
  - 3.1.1 Eyeriss . . . 20
  - 3.1.2 EIE . . . 21
  - 3.1.3 NullHop . . . 21
  - 3.1.4 SCNN . . . 23
- 3.2 Sparsity Based SMAC-Engine Basics . . . 25
  - 3.2.1 Data flow . . . 25
  - 3.2.2 Compression of data . . . 26
  - 3.2.3 Weights sharing . . . 27
- 3.3 Analysis of Sparsity for some Convolutional Neural Networks . . . 29

4 SMAC-Engine for Sparse CNNs 33
- 4.1 Hardware Design of the Accelerator for sparse data volumes . . . 33
  - 4.1.1 Data organization in TCDM . . . 33
  - 4.1.2 Decompression System . . . 35
  - 4.1.3 SMAC-Engine . . . 41
  - 4.1.4 Compression System . . . 42
- 4.2 Control Unit . . . 45

4.2.1 COUNTERS 45
4.2.2 FSM Low-Level 46

5 Integration on PULP HWPE 50
5.1 PULPissimo system and HWPE 50
5.1.1 The Hardware Processing Engine 50
5.1.2 Integrating SMAC-Engine sparsity based in a HWPE 53

6 Analysis Results 58
6.1 QuestaSim Simulation Results 58
6.1.1 Testbench Results 58
6.1.2 Testbench Setup 63
6.2 Synopsys Design Compiler logic synthesis 64
6.2.1 Synthesis setup 64
6.2.2 Synthesis Results 65

7 Conclusions and future improvements 67

Bibliography 69

Chapter 1

Introduction

1.1 General Principles

During the last years, the Artificial Intelligence took place in lots of applications fields, such as autonomous driving, object detection, speech recognition and other tasks that normally are considered "human-only". The AI, in fact, gave the possibility to computers to take intelligent decisions, and this is made possible mainly thanks to two fundamental concepts inspired by this computer science issue: *Machine Learning* and *Deep Learning*. These two methods have some differences between them, mainly related to the application fields. However, the Deep Learning is a sub-branch of Machine Learning and it is based on unsupervised learning, while the other on supervised learning. The first concept means that machines learn without the need of human instructions, but only by means of neural network training. While the second one means that they use structured data sets. The only similarity of these two branches of the AI is the huge quantity of data on which the machines are trained to take decisions.

Nowadays, Deep Learning is the branch of AI attracting a very large number of researchers and practitioners due to its extremely wide range of applications, from 4.0 industry to healthcare, IT security and many others. The particularity of the DL approach is that the system acts in multi-layer neural networks, such as DNN (Deep Neural Networks), which combine different algorithms and have been inspired by the human brain behaviour and structure. In fact, since a neuron is considered to be the elementary computational element of our brain, which is composed of billions of them, a DNN consists of several layers (the higher the number the deeper the network), each containing some neurons contributing to the computation of the output result. At each layer, the input is controlled for an other characteristic and the system uses this practice to decide how to characterize

the data concerned. This is useful when not all the aspects of the objects can be categorized previously. In this way, only the system can recognize the adequate distinguishing features and any related changes.
During the training process, a DNN tries to properly tune the weights and biases parameters based on the so-called hyper-parameters, like the learning rate, the number of hidden layers, the number of neurons and so on. The weights and biases parameters are the ones that will ultimately be used during inference to perform the specific tasks the DNN has been developed for.
The CNN (Convolutional Neural Networks), introduced by LeCun et al. from 1998, is one of the most important typology of DNN, and has been designed to process images. However, the computer vision application is not the only one in which CNN operate, but many other areas are equally involved. The real change has been introduced in 2012 with AlexNet which won the ImageNet Large Scale Visual Recognition Competition (ILSVRC). Meantime, some conditions have changed thanks to these type of multi-layer networks. In fact, a great quantity of labeled training data are available now, such as ImageNet which contains millions of images and ten thousands of classes. Moreover, the training of complex models, deep and with lots of weights and connections, needs high computational power systems like GPU, with millions of cores and GB of internal memory which reduced the training time from months to days. An other change is the implementation of Relu as activation functions instead of sigmoids, which could be problematic during back-propagations of gradients.
Finally, the development of several open source frameworks tailored for DNN applications, such as Tensorflow, Caffe, PyTorch, Keras and many others, further enlarged the accessibility to newcomers thus broadening their evolution and diffusion even more.
The use of DNNs brought to the possibility in optimizing new technologies such as Internet of Things, where big data centers receive data from edge nodes sensors that, typically, are further processed by the same data center. But these neural networks introduced the new concept of elaborating directly in loco the images or other informations they acquire, making a previous inference computation at the edge node level. This brings to the possibility of getting higher performance transfers of huge quantity of data, and in order to do it specialized hardware solutions, such as ASIC, FPGA or complex SoCs, have been thought. For example, thanks to the intrinsic post-fabrication programmability, FPGAs allow for a greater flexibility in terms of workloads handling. In fact, this flexibility enables the user to reconfigure the data path easily, even during run time, using partial reconfiguration. This approach is completely different from the one used by GPU, in which there is a parallel architecture (SIMT) that allows a reduced mapping of the workload if the parallelism is not enough high. So, FPGAs result in lower performance efficiency, besides the fact that the additional complexity around their compute resources to

facilitate software programmability is power-hungry.

However, higher integration solutions such as SoC, where we can find processor unit, memory hierarchies and elaboration unit all together, are more affordable. For instance, the ETH of Zurich and the University of Bologna proposed a paradigm where an ultra-low power multicore SoC called PULP (Parallel Ultra Low Power) can be augmented by specialized accelerator units, called Hardware Processing Engines (HWPE), to greatly accelerate specific functions such as the DNN inner kernels.

1.2 Introduction to chapters

The chapters of this thesis are organized in this way:

1. *Deep Neural Networks and Convolutional Neural Networks.* This chapter gives an insight into DNNs and, more in particular, CNNs about which hystorical origins and mathematical concepts are properly explained.

2. *Sparsity.* This chapter has been concerned in order to explain the work process, from the study of jet exploited accelerator architectures for sparse data to the one employed in this thesis work.

3. *SMAC-Engine for sparse CNNs.* In this chapter will be shown the hardware architecture of this SMAC-Engine sparsity based, properly defining structures and functions of its main blocks.

4. *Integration on PULP HWPE.* This chapter has been used to explain how the SMAC-Engine with the new mechanism of handling sparse data has been integrated inside the HWPE accelerator, taking place in the engine part.

5. *Analysis results.* This chapter introduces the analysis on timing, area and power of this system.

Chapter 2

Deep Neural Networks and Convolutional Neural Networks

2.0.1 Basic concepts

The Deep Learning is a branch of the Machine Learning, and it introduces respect to the last one the possibility to extract from some objects, like images or texts, informations which, typically, only the human perception can extrapolate. In fact, the Deep Learning has some elements reminding the structure of a brain, with neurons and axons. In the following figure is shown this analogy.

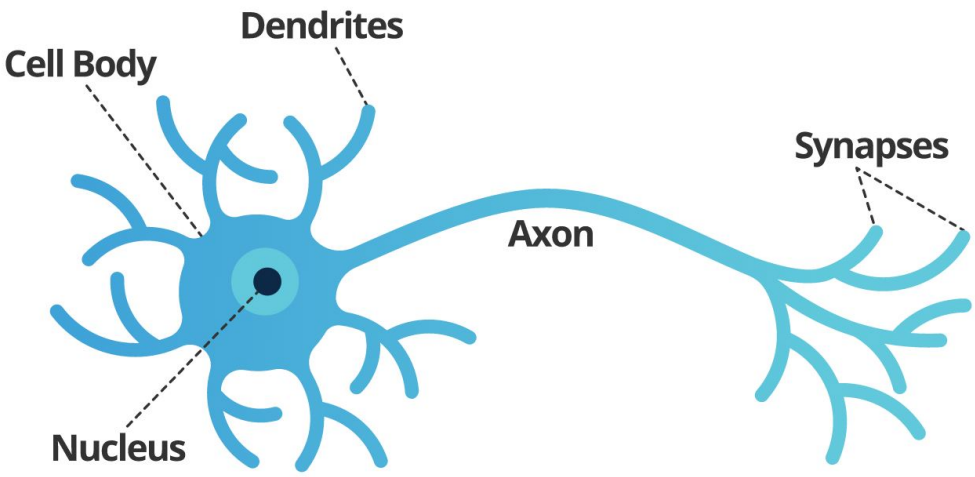

Figure 2.1: Human brain's neurons

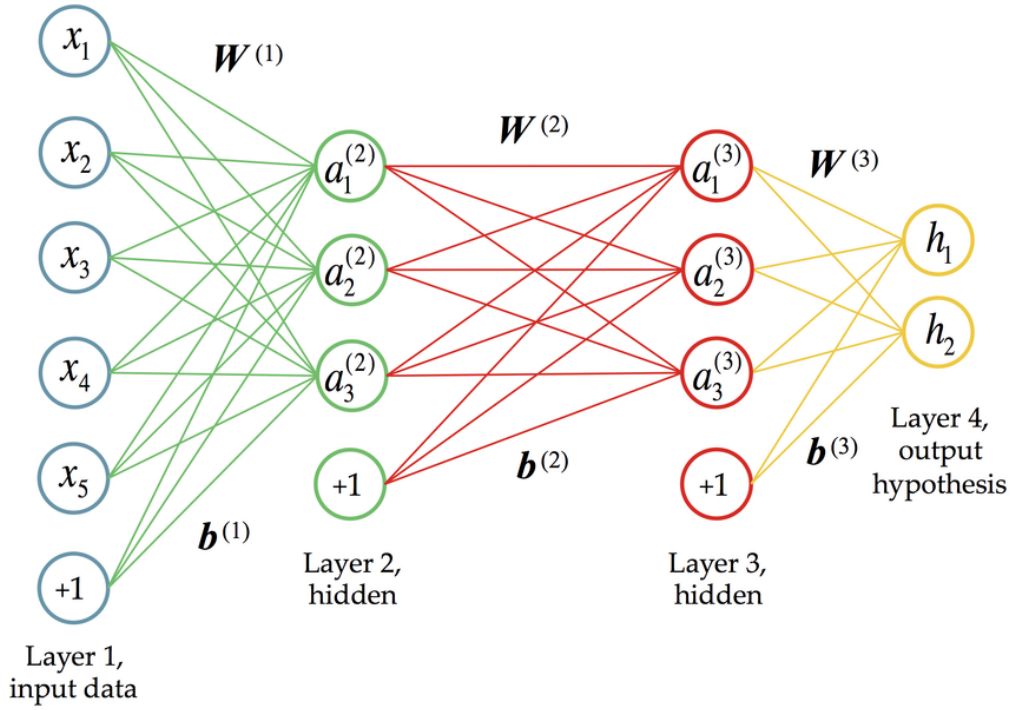

Figure 2.2: DNN structure [14]

So, as it can be seen from Figure2.1 and from Figure2.2 the Neural Network

has lot of processing units called *neurons* that are similar to the ones present in the human brain. In particular, the input layer ($features$) is assimilated to the dendrites in human brain's neural network; while the hidden layer is considered as the cell body and lies between the input layer and the output layer, which is like the synaptic output in the brain. In the hidden layer the artificial neurons take a set of inputs based on synaptic weight, which is the amplitude or strength of a connection between nodes. These weighted inputs generate an output through a transfer function to the output layer. The mathematical computation of the output of a j_{th} hidden neuron is the following one:

$$a_j = f\left(\sum_{ß} x_i W_{ij} + bias\right) \tag{2.1}$$

In the equation 2.5 there are 2 parameters, W (weight) and $bias$, important for the definition of the output. In fact, the weights are used as multiplicative factor applied to the input signal, and can be positive or negative (*inhibitory* and *excitatory* synapses), greater or less than 1 (*amplifying* or *attenuating* synapse). This makes clear the nature of this function, that is a weighted sum among all the input features. The bias, instead, is important to adjust the working point of the neuron itself. While x_i are the input activations, the output activations are obtained by means of a non-linear transfer function, called *activation function*. There is the possibility to choose among different of these functions, and below are listed some of them:

- Sigmoid function:
$$f(a) = \sigma(a) = \frac{1}{1 + \exp(-a)} \tag{2.2}$$
- Hyperbolic Tangent function:
$$f(a) = \tanh(a) = \frac{e^a - e^{-a}}{e^a + e^{-a}} \tag{2.3}$$
- Rectified Linear Unit (ReLU):
$$f(a) = \max(0, a) \tag{2.4}$$
- Leaky ReLU:
$$f(a) = max(0.1a, a) \tag{2.5}$$

Nowadays, the ReLU is the most used activation function and is defined as the positive part of its argument:

$$f(x) = x^+ = \max(0, x) \tag{2.6}$$

where x is the input to a neuron. This is a ramp function, and it was introduced for the first time in a dynamic network by Hahnloser et al. in 2000, citing strongly biological reasons (*activation potential*). In 2011 it has been proved that this function is good to train deep neural network in an efficient way. This because, throwing away the linearity part of the function that relates input to outputs, the output activations reach a certain sparsity level that carries to a better approximation of very complex and deep neural networks. From 2018, this function is the most used in DNN.
Moreover, other activation functions like the sigmoid and the hyperbolic tangent were a problem for the training of very complex deep neural networks, because of the vanishing gradient descent and the slower back propagation. These, in fact, are the most used algorithm during training process in order to derive the parameters that work best for the designed network, so need to be optimized in order to obtain good performances in terms of training.

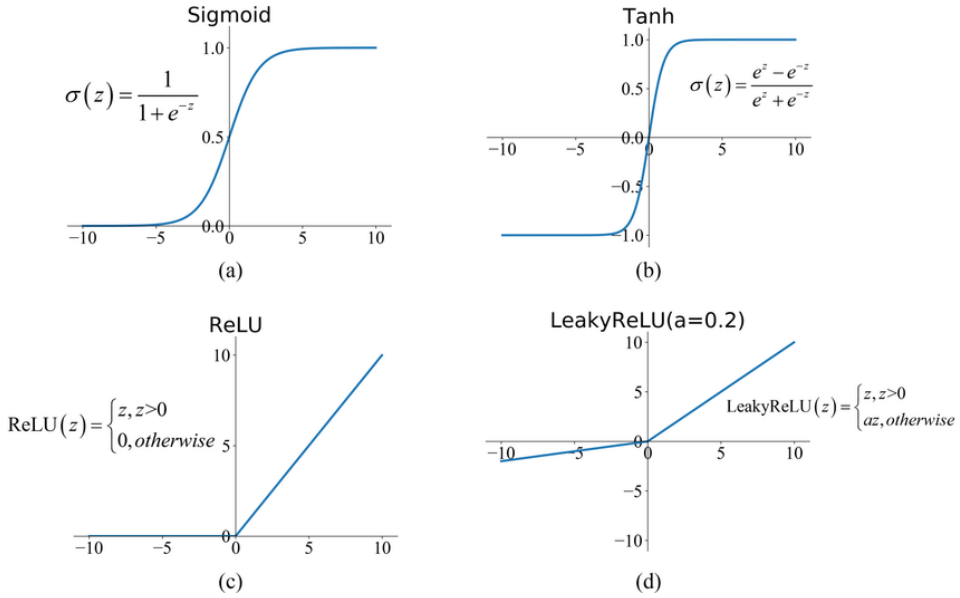

Figure 2.3: Activation functions [5]

Thanks to this structure, the artificial neural networks can emulate the complex functions that only human brain could implement.
Some models of deep learning are made of different elaboration steps, each one extracting the representation of the step before. In the supervised deep learning, which means that the machine can automatically make previsions about the output of a system thanks to some previous examples of input – output obtained, the most used neural network is the multi-layer one, so called *Deep Neural Network*.

2.0.2 The origins of Convolutional Neural Networks

The most important typology of DNN for image classification is the CNN - Convolutional Neural Network or ConvNet. This kind of Neural Network has its origin in the studies carried out by the two neurophysiologists David Hubel and Torsten Wiesel, who collaborated for several years during the last century. The object of their studies was the responsivity of neurons in cat's brain to some images projected in precise locations on a screen. The discovery (1962) was that neurons in the early visual system were excited more strongly by certain patterns of light respect to others [6].
The part of the brain that inspired the CNN architecture was the primary visual cortex, V1. This last one is the first area of the brain that elaborates input images, which are then shaped by light arriving in the eye and stimulating the retina, the light-sensitive tissue in the back of the eye. The image then passes through the *optic nerve* and a brain region called the *lateral geniculate nucleus*. These two anatomical regions carry the signal from the eye to V1.
Three properties of V1 have been used to inspire CNN:

1. V1 is arranged in a spatial map. While mirroring the image in the retina, it has a two dimensional structure. So, convolutional networks capture this property by having their features defined in terms of two-dimensional maps.

2. V1 contains many simple cells. A simple cell's activity can to some extent be characterized by a linear function of the image in a small, spatially localized receptive field. The detector units of a convolutional network are designed to emulate these properties of simple cells.

3. V1 also contains many complex cells. Differently from simple cells, complex cells are invariant to small shifts in the position of the feature. This inspires the pooling units of convolutional networks.

The neuroscience has not exposed precisely for the *training* of convolutional neural networks. We can list different studies like the one carried on by Marr and Poggio, 1976, in which was contemplated the model structures with parameter sharing across multiple spatial locations date back to early connectionist models of vision, but not using the modern back-propagation algorithm and gradient descent. For example, the neocognitron (Fukushima, 1980) incorporated most of the model architecture design elements of the modern convolutional network but relied on a layer-wise unsupervised clustering algorithm.
Then, *Lang and Hinton* (1988) introduced the use of back-propagation to train TDNNs (Time-Delay Neural Networks). These are one-dimensional convolutional networks applied to time series, but this is not inspired by biological motivations. After this success with back-propagation, *LeCun et al.* (1989) developed the modern

convolutional network by applying the same training algorithm to 2D convolution applied to images.

2.0.3 The Convolution Operation

As the name of Convolutional Neural Network suggests, the basic algorithm that these networks employ is the *convolution function*. Convolution is a specialized kind of linear operation. So, Convolutional Neural Networks are simply neural networks that use convolution in place of general matrix multiplication in at least one of their layers. In mathematics (in particular, functional analysis), convolution is a mathematical operation on two functions (f and g) that produces a third function (f * g) that expresses how the shape of one is modified by the other. It is defined as the integral of the product of the two functions after one is reversed and shifted. And the integral is evaluated for all values of shift, producing the convolution function.

$$y = f * g \tag{2.7}$$

In the time domain, the convolution can be seen as a mathematical operation that, given two functions, gives a third one computed as in the following:

$$y = (f * g)(t) = \int f(\tau)g(t-\tau)d\tau \tag{2.8}$$

However, in machine learning applications the input is usually a multidimensional array of data, and the kernel is usually a multidimensional array of parameters that are adapted by the learning algorithm. Typically, we use convolutions over more than one axis at a time. For example, if we use a two-dimensional image I as our input, we probably also want to use a two-dimensional kernel K:

$$S(i,j) = (I * K)(i,j) = \sum_m \sum_n I(m,n)K(i-m, j-n) \tag{2.9}$$

Convolution is commutative, meaning we can equivalently write:

$$S(i,j) = (K * I)(i,j) = \sum_m \sum_n I(i-m, j-n)K(m,n) \tag{2.10}$$

The convolution operation is the basis for the invariant linear systems theory. The meaning of this is explained in the following:

- **Linear System**. A system is linear when the transform function applied by the system respects the "Principle of Superposition of Effects"; this property is explained by the equation below:

$$L[a_1x_1(t) + a_2x_2(t)] = a_1L[x_1(t)] + a_2L[x_2(t)] \tag{2.11}$$

- **Invariant System**. A system is invariant when the variation in the input (in time domain or in spatial coordinates) is repeated by the output in the same way. If

$$T[x(t)] = y(t) \tag{2.12}$$

then

$$T[x(t - \Delta)] = y(t - \Delta) \tag{2.13}$$

Application of convolution to digital filters

The convolution is an operation that finds its functionality in different application fields. In particular, among them there are the following two:

- digital elaboration of images (convolutional filters);
- digital signal processing, in which the filtering in the frequency domain results in a convolution between two functions in the time domain, that is a multiplication in the frequency domain.

Looking at the second one, doing a discretization of the time in which the signal is sampled by the Linear System, the convolution becomes a simple sum between the multiplications of a same weighting function with different time instants of the signal in input to be elaborated. Doing the integral, instead, we obtain a smoothed estimation of the signal, that after the filtering will be free from noise and other less important informations [11].
So, in this case an analogy has to be done. In fact, in both the applications are used filters to modify the signals or the images, but are both related to different domains. While for digital signal processing are used some probability density functions in the time domain to change the spectrum of the input signal maybe filtering some noise, the filters used in the processing of images are called Kernels and are used as masks for images in order to recognize borders, corners and so on. Furthermore, going deeper with the features characterization in CNN it's possible to obtain detailed informations set about the images under filtering.

The other important analogy is that, as the digital filters for signal processing in the frequency domain need to be done in an LTI (Linear Time Invariant) system, digital filtering for images needs a Linear and Invariant System that brings to the architecture of the Convolutional Neural Networks, that will be explained further in this chapter.

2.0.4 Convolutional Neural Networks basics

CNN took place in the computer vision field with higher strength in 2012, after the challenge ILSVRC (ImageNet Large Scale Visual Recognition Competition), which is an image classification competition. AlexNet was the Convolutional Neural Network that won that competition, introducing some relevant conditions respect to the previous CNN of LeCun (1989). These conditions are summarized below:

- Big Data: there was the availability of dataset of huge dimensions, like ImageNet, which contains over 15 millions labeled high-resolution images with around 22,000 categories. ILSVRC uses a subset of ImageNet of around 1000 images in each of 1000 categories. In all, there are roughly 1.2 million training images, 50,000 validation images and 150,000 testing images [9].

- GPU computing: there is the need of using powerful computational systems, and the GPU with multiple cores and GB of internal memory carried to reduce the time of training from few months to few days.

- Vanishing (or exploding) gradient: the gradient back propagation is a problem on very deep networks if the sigmoid function is used as activation function. So, the problem was solved with the introduction of ReLU activation function, which make several times faster the training than the equivalent *tanh*.

Main Characteristics of CNN

There are three important motivations to choose CNN instead of other neural networks, and below are defined which ones and why.
First of all, the fact that the kernel (filter) is smaller than the input image implies that some connections between the output and all the inputs are sparse, so they are not connected. This property is called *sparse connectivity*. Typically, the kernel is used to detect corners, edges and other features that could be repeated over all the image in input. This means that if we have to store in a memory the parameters that are used to accomplish the convolution is not a problem for the energy consumption and for the efficiency of the operation, because we need fewer pixels of weights in order to do this over thousand of pixels of the input image. So, also fewer operations need to be done. This is a good optimization respect to traditional neural networks, where layers use matrix multiplication by a matrix of parameters that connect all the inputs with all the outputs in a linear way, making a dense connection between them.

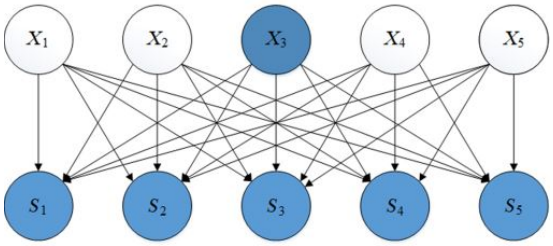

Figure 2.4: Dense Connections [15]

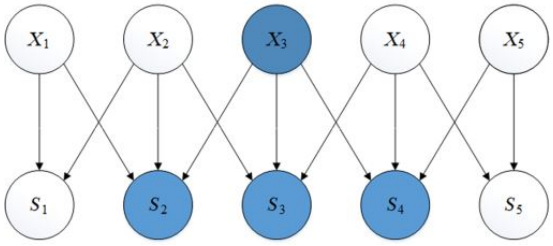

Figure 2.5: Sparse Connections [15]

The second important issue that brings to choose CNN better than traditional neural networks is the *parameter sharing*, which is a consequence of the fact that the same features are detected over all the image in input. In fact, while typically in traditional NNs the same filter is not used for other inputs but only once, in this case the weights of a kernel are tied to different input positions. So, the CNN learns no more than one set of features each time it does the convolution.
The last motivation that makes CNN so efficient is the *equivariance* introduced by the convolution operation, which is something we have seen yet reminding to the concept of Invariant Systems. In this case, the domain is not time but space. So, differently from the former where the changes at the output of the system were related to time delay, here the variation we observe at the output is tied to a spatial translation. So, the same change is obtained both in input and output image. Mathematically, a function f(x) is equivariant to a function g if:

$$f(g(x)) = g(f(x)). \tag{2.14}$$

2.0.5 Architecture of Convolutional Neural Networks

Typically, the input images of a CNN are composed of three layers, namely RGB (Red, Green and Blue). Each pixel of these three layers contains values and are arranged spatially along the width, height and depth (channels). The goal of the architecture of CNN is to learn the classification of the image figured out by these

pixels values with the features recognition. Operations like convolution, pooling and ReLU are applied to squeeze and stretch them along the depth. An example of this is shown in Figure2.6.

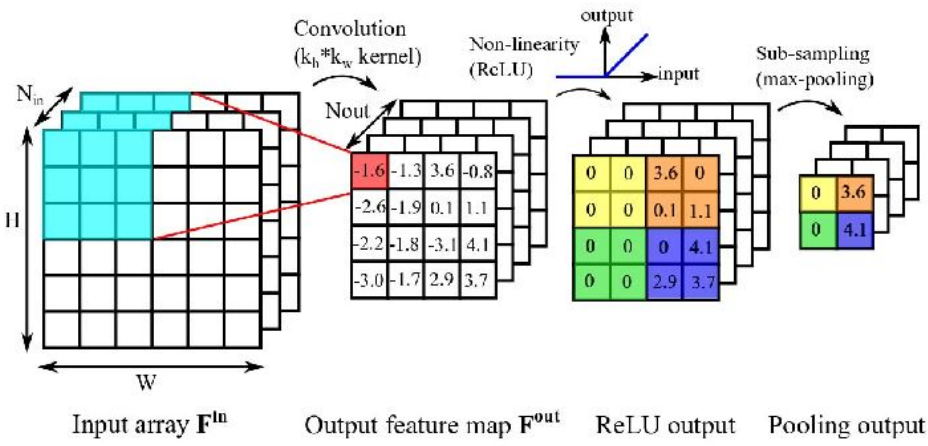

Figure 2.6: 2D convolution [1]

In fact, it is observable that after the convolution layer the channels at the output are more than those of the input. These are called *feature maps*, and are related to different filters applied at the same input activations. From their name it figures out that each channel at the output extrapolates a different feature of the input image over all the pixels that compose it.

Performing Convolution

The convolution operation consists, usually, in these three steps:

1. taking the input volume, superimposing a filter on it, starting from the upper leftmost position;

2. performing the element-wise multiplication and adding up the result, which is also known as a Multiply and Accumulate (MAC) operation, to obtain the output value;

3. moving the filter by one position and repeat the operation until all the output elements are calculated.

The mathematical definition is reported below:

$$y[k_{out}][w_{out}][h_{out}] = \sum_{k_{in}=0}^{n_c^{[l-1]}} \sum_{i=0}^{f^l} \sum_{j=0}^{f^l} x[k_{in}][w_{out}+i][h_{out}+j] \cdot W[k_{out}][k_{in}][i][j] \quad (2.15)$$

where:

$$0 \leq k_{out} \leq n_c^l, \quad (2.16)$$

$$0 \leq w_{out} \leq n_w^l, \quad (2.17)$$

$$0 \leq h_{out} \leq n_h^l, \quad (2.18)$$

The dimensions of the output, considering the 3D shape, follow a particular rule, that is, if the input dimensions are n_w^{l-1} x n_h^{l-1} x n_c^{l-1} and the number of filters of dimensions f^l x f^l is n_f^{l-1}, then the output volume will have dimensions n_w^l x n_h^l x n_c^l, where:

$$n_w^l = n_w^{[l-1]} - f^l + 1 \quad (2.19)$$

$$n_h^l = n_h^{[l-1]} - f^l + 1 \quad (2.20)$$

$$n_c^l = n_f^{[l-1]} \quad (2.21)$$

Stride and Padding Techniques

This rule changes if we consider two techniques more:

1. Zero padding;
2. Striding.

The first is applied when there is interest in preserving informations on the edge of the image. In fact, as we can see from Figure2.7, the dimensions of the image is enlarged by a further crown of zeros that will be helpful in considering the activations at the edge more than once, so including their values in more than one output pixel.

Here, the rule for dimensions is the following one:

$$n_w^l = n_w^{[l-1]} + 2p - f^l + 1 \quad (2.22)$$

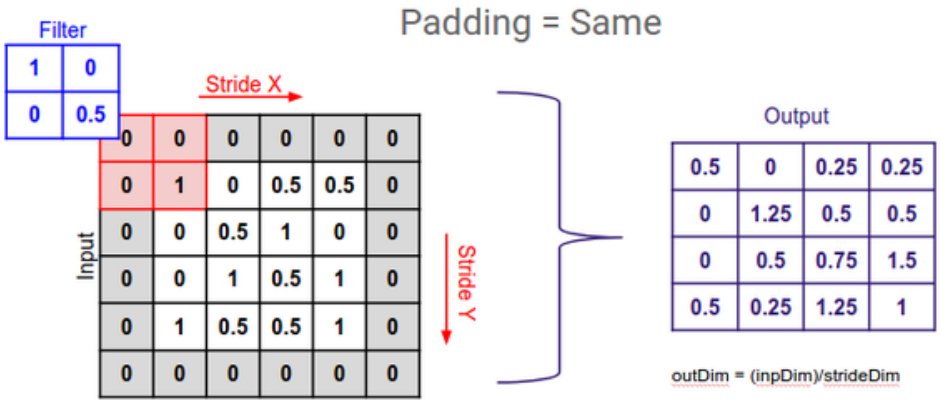

Figure 2.7: Zero Padding Technique [13]

$$n_h^l = n_h^{[l-1]} + 2p - f^l + 1 \tag{2.23}$$

$$n_c^l = n_f^{[l-1]} \tag{2.24}$$

where p can assume different values; if we don't consider padding it will be 0 (*valid convolutions*), while in other cases (*same convolution*) the padding amount is adjusted in order to have the same output size as the input; so,

$$p = \frac{f^l - 1}{2} \tag{2.25}$$

The second technique we have nominated is the *striding*. It consists in doing a larger step each time we want to slide the filter. So, instead of moving it by only 1 pixel at a time, it slides by s pixels. This is useful if we want to get a lower overlapping of the output values in a layer. In Figure2.8 is represented graphically how it operates.

The rule to obtain the output sizes is the following one:

$$n_w^l = \left\lfloor \frac{n_w^{[l-1]} + 2p - f^l}{s} + 1 \right\rfloor \tag{2.26}$$

$$n_h^l = \left\lfloor \frac{n_h^{[l-1]} + 2p - f^l}{s} + 1 \right\rfloor \tag{2.27}$$

$$n_c^l = n_f^{[l-1]} \tag{2.28}$$

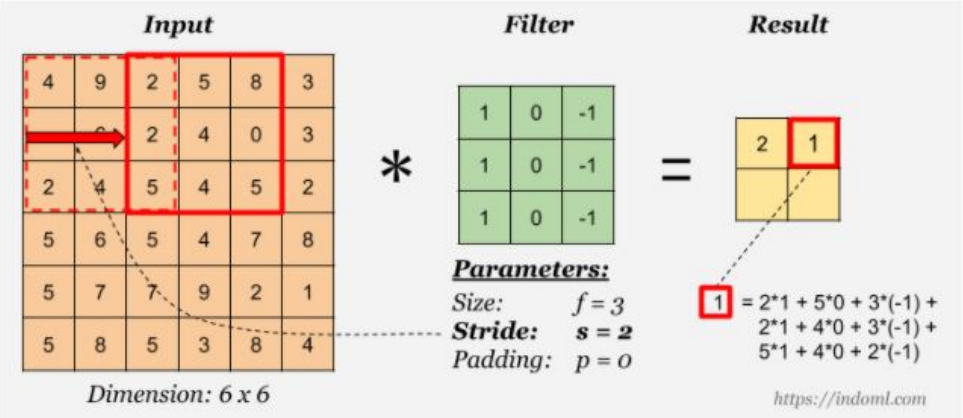

Figure 2.8: Striding Technique

Architecture

Each layer of a Convolutional Neural Network is composed of the following stages:

1. Convolutional stage (CONV);
2. Pooling stage;
3. Fully Connected (FC) stage.

Typically, after the convolution step is introduced the non-linearity by means of ReLU (Rectified Linear Unit) function, which has been introduced previously in this chapter.

The Pooling layer is the one which helps to extrapolate well the features of the image without taking care of the small translations of it. In fact, the operation done by the Pooling is, typically after the ReLU, a statistical summary of the pixels around a certain location. For example, the *max pooling* reports the maximum output within a rectangular neighborhood. So, the pooling adds invariance to local translations at the output of the layer in which lies, because after a translation the most of the pooled outputs do not change. An other important motivation to use pooling functions is to obtain output feature maps of the size required by the network, regardless of the input size.

The last stage, the FC (Fully Connection), is a dense connection that is used for the final classification of the image.

Chapter 3

Sparsity

In Section 1.2 it was introduced the definition of ReLU function, that is used to obtain activation values. An important characteristic that this last operation adds to CNNs is the sparsity of its feature maps. So, starting from this last concept a new kind of hardware architecture has been implemented in this thesis design respect to the previous one, in order to handle the sparse matrices of data.

Sparsity is the condition in which, for instance, a matrix of data could be if most of its values are exactly at zero. So, considering the MAC (multiply and accumulate) operations that typically are employed in the convolution operations in CNNs, this attribute could come in handy while performing it, avoiding to do multiplications with zero values. In this way, different techniques have been explored during the last years by hardware accelerators supporting CNNs. In the following paragraph these solutions will be explained properly.

3.1 Hardware Accelerators Sparsity Based in Convolutional Neural Networks

Typically, the huge amount of data that CNN and, more in general, DNNs should use is really power-consuming and reduces performances of the hardware that handle the inference process. In fact, in such systems typically there is an on-chip engine that is involved in convolution operations and an off-chip main memory to access each time is required by the algorithm. The bandwidth and the number of loading cycles related to the accessing in memory could be very energy-consuming and it's the main motivation that brings to find solutions in order to make CNNs accelerators affordable for the intended use, for example in the IoT field. For instance, if it would be implemented in wearable devices for medical aims it should be a low-power system in order to have a longer time-life.

An other improvement, less important in terms of energy consumption but however helpful, is the reduction of multiplications with zero values, which wouldn't make any contribution to the final results.

Sundry solutions for sparsity based accelerators have been implemented by researchers in the CNNs field, but it's still difficult to recognize the most affordable one in terms of power consumption and performances, because it depends also on the kind of Neural Network that is supported in the particular architecture.
Below, are reported the final results in terms of power consumption and throughput reached by some hardware accelerators:

- *Eyeriss* processes convolution layers at 35 frames/s and 0.0029 DRAM accesses/MAC for AlexNet at 278 mW [4];
- EIE has a processing power of 102 GOPS/s working directly on a compressed network, corresponding to 3TOPS/s on an uncompressed network, and processes FC layers of AlexNet at $1.88x10^4$ frames/sec with a power dissipation of only 600mW [8];
- $NullHop$, exploiting sparsity, achieves an efficiency of 368%, maintains over 98% utilization of the MAC units, and achieves a power efficiency of over 3 TOPS/s/W in a core area of 6.3 mm^2 [2];
- $SCNN$ improves both performances and energy by a factor of 2.7x and 2.3x respectively, over a comparably provisioned dense CNN accelerator [12];

Now, it will be explained in each case which kind of techniques have been implemented for sparse data and how they should impact to the power consumption and performances of the system.

3.1.1 Eyeriss

This accelerator uses as sparsity solution the compression of data technique called *RLC* (Run Length Compression), used in order to exploit zeros in feature maps and save DRAM bandwidth. In Figure3.1 is shown an example of RLC, in which the data is divided in 2 parts: *Run* and *Level*. The first encodes the length of zero sequences, that in this case could reach 31 zeros, while the second term is related to the value of non-zero activation that is separated from the next one by the number of zeros in the Run term. In this case, the data are packed into a 64 bit word, where at the end there is a 0 term saying if the data compression comes to the end.

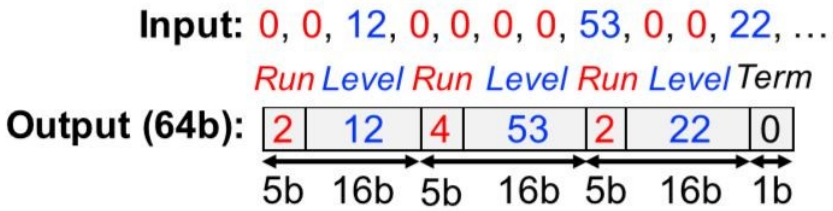

Figure 3.1: RLC example [4]

The compression of data allows to reduce the necessary capacitance of data volume in the DRAM, that is the main memory linked to the accelerator. The input data of each layer of a CNN, except for the first layer, are written in the DRAM in the compressed format.

How is the stream of data between the accelerator and the DRAM?
In order to recognize the real sequence of the activation values, the accelerator needs an RLC decoder, which decompresses the data coming from the DRAM that are, further, written into the GLB (Global Buffer). After the convolution is done, the computed output feature maps are read from the GLB, processed by the ReLU module where appropriate and compressed by the RLC encoder. This compression is good in terms of communication bandwidth with the DRAM, where data are then stored. In fact, this saves lot of energy linked to the loading and storing cycles of data, as well as the space of the DRAM.

What is the role of Global Buffer?
The Global Buffer is a local memory that interfaces the external DRAM, and it contains 100 kB of input feature maps and partial sums/output feature maps, while 8 kB of filter weights for the next processing step, in order to save the energy linked to the external memory access and storage.

The space is divided into 25 banks, each of which is a 512 bit x 64 bit (4 kB) SRAM, where the PE array can access simultaneously input feature maps and partial sums.

3.1.2 EIE

This is another hardware accelerator handling sparse data. Below, are reported some important issues related to this kind of accelerator:

- dynamic input vector sparsity;
- static weight sparsity;
- weight sharing.

The basic idea is to compress the network by means of *pruning*, using the technique of *Deep Compression* [7]. This technique consists in removing from the network all connections with weights below a threshold, and then retraining the sparsified network to lain with the final weights. This is useful to reduce the accesses to cache memory during the multiplication of matrix by vectors that, typically, are accomplished during CNNs.

Moreover, it could help bringing all the weights of the sparse network in an on-chip SRAM. In fact, each processing element has its own SRAM with prestored weights which processes the activations incoming.

The following results have been reached with this approach:

- Operating directly on compressed networks enables the large neural network models to fit in on-chip SRAM, which leads to 120x energy savings compared to accessing from external DRAM.
- Exploitation of the dynamic sparsity of activations to save computation. It saves 65.16% energy by avoiding weight references and arithmetic for the 70 % of activations that are zero in a typical deep learning application.
- Method of both distributed storage and computation to parallelize a sparsified layer across multiple PEs, which achieves load balance and good scalability.

3.1.3 NullHop

The technique used in this case is called *Sparsity Map* (SM), where a 3D mask of '0' and '1', together with a Non-Zero Value List (NZVL), has the same number of values as for the activations in input in the feature maps. The pixels that are

equal to '1' indicate that the activations in that positions are non-zero values. The length of NZVL can be variable and it contains all the non-zero activation values, seeking the order reported in Figure 3.2.

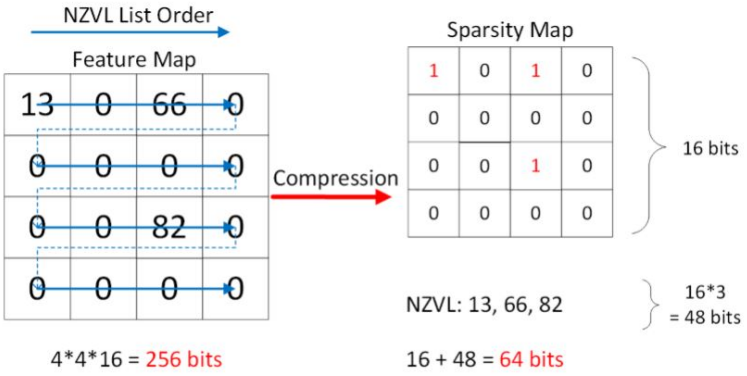

Figure 3.2: SM example [2]

This accelerator can implement one convolutional stage followed by a ReLU and max-pooling stage, which can be disabled if necessary.
An important feature that make this technique affordable it's that it avoids the redundant MACs and wasting of clock cycles skipping over zeros in the input CNN layers.

How is the data stream handled on the NullHop architecture?
Two streams of data come from the external memory, with a parallelism of 32 bits. One of them is for the input configuration that comes from microcontroller, while the other one is for the output control signals like clock, reset and bus handshake signals. The input feature maps and the kernel values for the current convolutional layer are stored in two independent SRAM blocks. The output feature maps are then pushed off-chip to the external memory, and streamed back to the local SRAM for the next convolutional layer.
During the computation of the output maps, the values of the input features are never decompressed but decoded.

In the following lines, it will be described how the processing pipeline is organized. The *Input Decoding Processor* (IDP) generates pixels from the reading of a portion of the compressed input feature maps, and these are then passed to the *Compute Core Module* (CCM). These pixels are, typically, all non-zero and this helps in skip over MACs with zero activations, that would give a negligible contribute to the final result.
The IDP also forwards the pixels positions (row, column, input feature maps index) to the CCM, where it's placed the *Pixel Allocator*, which allocates each incoming

pixel to a *Controller*. Each of these manages the operation of a subset of the MAC blocks and submits the appropriate read requests to the Kernel Memory banks. All MAC blocks under the same Controller receive the same input pixel from their Controller, but weights from different kernels, producing pixels in different output feature maps. Then, the pixel stream compression block streams off-chip the output feature maps.

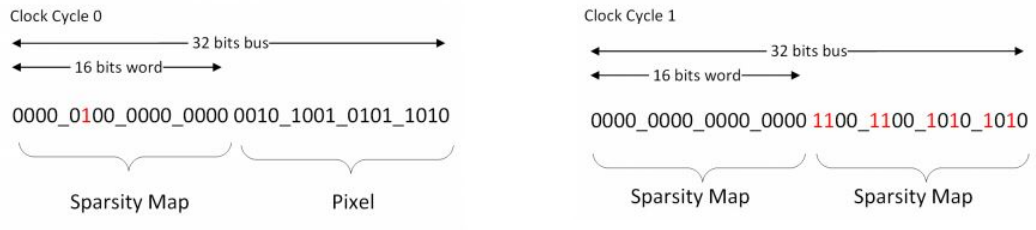

Figure 3.3: Format of the words in NullHop [2]

3.1.4 SCNN

The Sparse CNN accelerator architecture is an inference architecture that exploits both weights and activation sparsity to improve the performance and power of DNNs, and its usage is intended to optimize the computation of the convolutional layers.
SCNN employs both an algorithmic dataflow that eliminates all multiplications with a zero and a compressed representation of both weights and activations through almost the entire computation.
The processing element (PE) has a multiplier that accepts a vector of weights and a vector of activations (Cartesian Product). Moreover, in order to reduce data accesses the activation vectors are reused in an input stationary fashion while being multiplied with a series of weight vectors. Finally, only non-zero weights and activations are fetched from the input storage arrays and delivered to the multiplier.
Also SCNN, as CNN, accelerators can accumulate the partial products generated by the multipliers. A scatter accumulator array, to which coordinates and products are sent, is used to sum the partial products.
Final results obtained by the implementations of SCNN are:

- efficient compressed storage and delivery of input operands;
- high reuse of the input operands in the multiplier array;
- no time spent on multiplications with zero operands.

The elimination of zero values multiplications is done by applying Cartesian Product dataflow, which exploits both weights and activation reuse while delivering non zero values to the multipliers. So, the choice of dataflow has an effect on area and energy-efficiency of an architecture.
In this work, the dataflow used is called *PlanarTiled-InputStationary-Cartesian Product-sparse*, or PT-IS-CP sparse, which enables reuse patterns that exploits the sparsity of data involved.

In the microarchitecture of SCNN are present two types of RAMs for both input activations and output activations, respectively IARAM and OARAM, and there is the possibility to swap them when the output becomes the input for the next layer of convolution. This permits to preserve energy and efficiency during the operations because a reduced number of accesses to external DRAM are needed.
When the activations vector I and weights vector F in input to the processing elements are fetched in their compressed form from their respective buffers, then are dispatched to an F x I multiplier array which accomplish the Cartesian product of these vectors. This means that a partial sum is obtained by the multiplication of every activation with every weight.

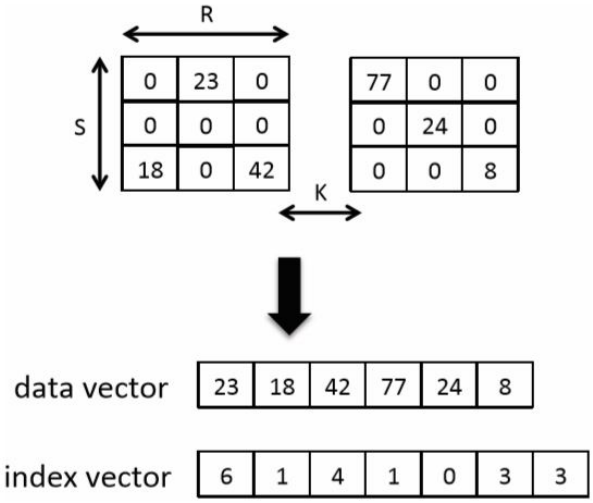

Figure 3.4: Weights compression [2]

The compression algorithm consists in defining a vector of indices and a vector of values, where the indices indicate the number of zeros that separate two non zero activations, while the values are the non null activations itself. The Figure 3.4 represents an example of this kind of compression.

3.2 Sparsity Based SMAC-Engine Basics

The previous implementations of sparsity based accelerator architectures have inspired this thesis work. Despite the differences between them that will be introduced in Section 3.2.1, mainly related to the kind of data flow implemented, this accelerator handles the sparsity of feature maps in each layer of convolution looking at the main techniques that brought the previous works to reach good results in terms of power consumption and MACs operations per clock cycle. In particular, as explained further in sections 3.2.2 and 3.2.3, the compression of data (activations) and weights reuse are the two techniques taken into account.

3.2.1 Data flow

Typically, the operation of convolution in CNNs are accomplished by using 2D maps of kernels sliding over the input feature maps, and the order with which data are fetched from the activations memory is the following one:

```
for z in range(k_ch):
   for x in range(f):
      for y in range(f):
          out_activation_value+ = act[z][x][y] * weight [z][x][y]
```

The difference in this thesis work is that the convolutions with weights are done in the direction of the channels, so in the z direction. The convolution result is the same but the difference is the data flow. The operations are carried on volumes of data in input. In fact, the activations of a convolutional volume are read by the memory TCDM in the z direction as shown in the following figure:

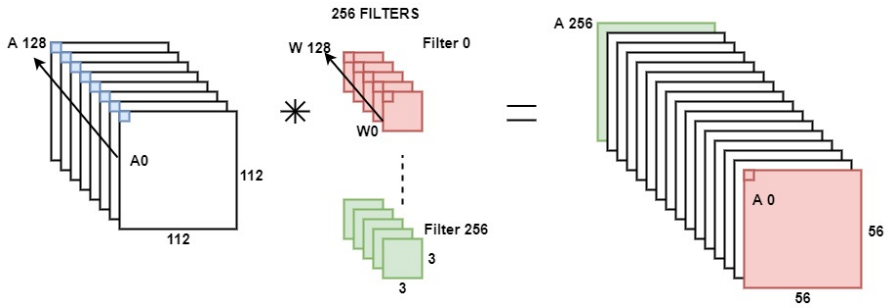

Figure 3.5: 2D convolution with 3D volumes of data

The order of the activation positions (x,y) inside the TCDM are contiguous, and are saved here in the direction of ascending x, y and z. For the z values there is an internal loop for the indexing of the TCDM. The outer loop is the one for x, while the loop for y is the outermost, as reported below:

```
for y in range(f):
   for x in range(f):
      for z in range(k_ch):
          out_activation_value+ = act[z][x][y] * weight [z][x][y]
```

3.2.2 Compression of data

The compression of data used in this thesis work is something similar to the one reported in Eyeriss accelerator, explained in paragraph 3.1.1.

This technique consists in compress data in the z ascending direction, as represented in Figure 3.5. The result from each compression is a data on 14 bits, where the 8 bits in the MSB part are related to the non-zero activation, while the 6 bits in the LSB part are related to the length of zero sequences. The number of zeros in this last element defines the offset between two non-zero activations, that is further decoded in order to obtain the convolution between the right values of both activation and weights.

While reading the output of the convolution operation, the compression follows this rule (shown also in Figure 3.6):

1. the first data considered is related to the position corresponding to z = 0, so belonging to the first feature map, and it is reported in the MSB part of the compressed data even if its value is zero;

2. the following data seek the compression algorithm, which defines that if there is a sequence of zero higher or equal to 1, it is written in the LSB part of the compressed word as "zero count";

3. if there is a zero sequence with length higher than 63, the sequence is divided in more than one data compressed. For instance, if the length of zeros is 80 it will be distributed in 2 data compressed where the first one is "00000000 111111" and the second one is "00000000001111".

4. when two non-null activations are contiguous, the result of the compression will be, for example, "01010101 000000" and "1010101 0000000".

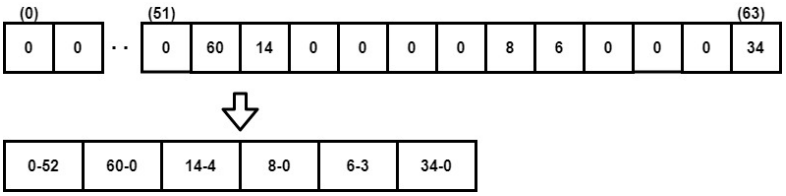

Figure 3.6: Activations compression example

3.2.3 Weights sharing

The concept of *weights sharing* was well exploited in the EIE work (Section 3.1.2), where each PE has its own SRAM cache in which weights are reused for different activation values. In our system, the reuse of weights on the internal engine was thought with the idea of reducing the memory access, and to exploit the parallel processing of the same weights position (x,y,z) for the activations shared between all the processing elements.

Analysis of loading cycles from TCDM

A previous analysis for the affordability of *weights sharing* in terms of energy consumption and performances has been done looking at the number of loading cycles that would be involved during the computation. In fact, considering a bandwidth of 224 bit for the interface with memory, and considering to load 64 filters or more at each convolution process, it was more affordable to consider a caching system inside the SMAC-Engine in which the weights are loaded only once; then, during the computation step in which the activation data compressed are taken, these values of weights are fetched in parallel among all the filters shared by the activations in input. Below, graphs and equations of this analysis have been reported.

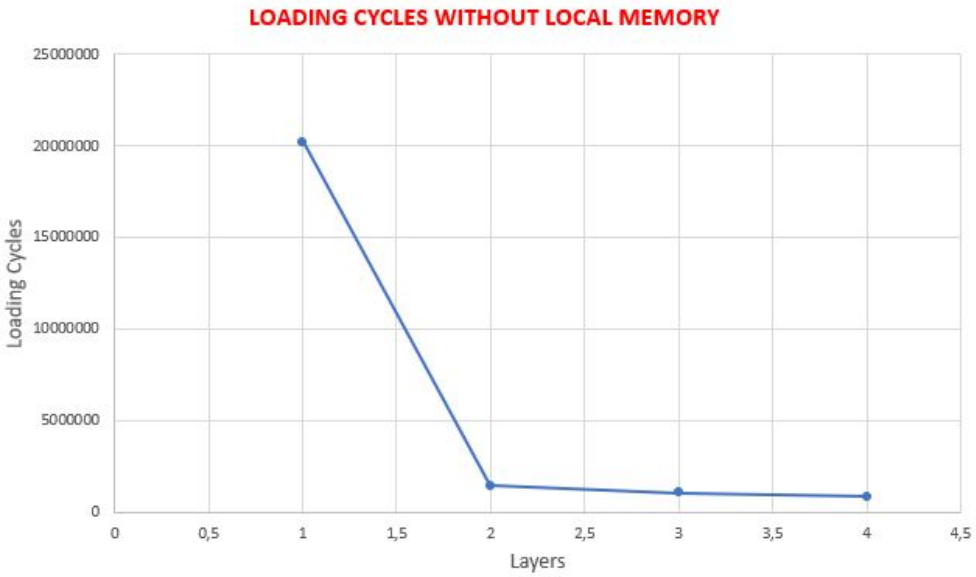

Figure 3.7

$$n_{cycles}^{nomemlocal} = \frac{n_{fil} \cdot dim_x \cdot dim_y \cdot dim_z \cdot 8bit \cdot vol_{conv}}{BW} \tag{3.1}$$

where vol_{conv} is the number of convolution volumes to be computed over all the input image, dim_x, dim_y, dim_z are the dimensions of a single filter, 8 bit is the parallelism of weights, and BW is the 224 bits bandwidth.

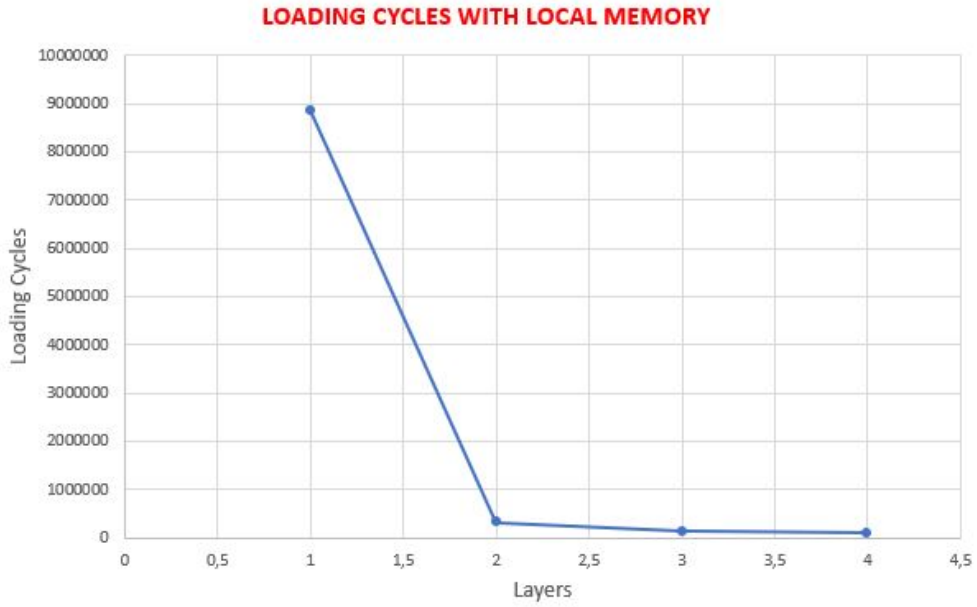

Figure 3.8

$$n_{cycles}^{memlocal} = \frac{n_{fil} \cdot dim_x \cdot dim_y \cdot dim_z \cdot 8bit}{BW} + dim_x \cdot dim_y \cdot dim_z \cdot vol_{conv} \tag{3.2}$$

where the first addend is the one used to define the number of cycles that are fetched directly from TCDM the first time, then are fetched all simultaneously from the SRAM banks inside the SMAC-Engine, as defined by the second addend.

So, looking at Figure 3.8 and Figure 3.7, the number of loading cycles is much lower for the second one of about 2 times. The CNN considered is the ResNet 18 and 34, that will be at the center of the further analysis as well.

For example, considering the third layer of ResNet 18, the following considerations could be done; if the fetch was done always from the external memory:

$$\frac{1}{2} \cdot \frac{3 \cdot 3 \cdot 128 \cdot 256 \cdot Pw}{BW} \cdot (dim\_out \cdot dim\_out) = 1032192 cycles \tag{3.3}$$

while, for the internal fetch:

$$\frac{1}{2} \cdot \frac{3 \cdot 3 \cdot 128 \cdot 256 \cdot Pw}{BW} + 36 \cdot (dim\_out \cdot dim\_out) = 17588 cycles \tag{3.4}$$

where Pw is for 8 bits (weight parallelism), BW is the bandwidth of 224 bits and dim_out is 14 (x and y size of output feature map).

3.3 Analysis of Sparsity for some Convolutional Neural Networks

Considering now the neural networks that could be implemented by this hardware accelerator, a previous analysis has been done on the sparsity of CNNs like ResNet, where the input of each convolutional stage in each layer was taken by means of a specific class in Python and analyzed in different ways. Moreover, the data set used to make the inference of this neural network is the Tiny ImageNet [10], containing 200 classes of the 1000 in ImageNet.

As reported in Figure 3.9 and Figure 3.11, a first analysis has been carried on about the percentage of zeros present in each feature map in input at each convolutional layer. This values range between 20 % and 80 % for both ResNet 18 and ResNet 34, and from this first analysis it can be seen that an implementation of the algorithm of compression could be useful for the activation values. In fact, a great part of these layers has more than 50 % of sparsity.
For what concerns the weights, their sparsity is something fixed and we can exploit better the sparsity coming from activations which are growing from the first layer to the last one as reported in the following figures.

A second analysis has been accomplished on the sequences of zeros that, as suggested before, are important in defining the offset between the non-null activations and for the compression of the data volume. As reported in Figure3.10 and in Figure3.12, the sequences are distributed among values between more or less 10 and 400, but in order to reach the final data on 14 bits, considering activations on 8 bits in the MSB part and the 6 bits of *zero count*, the following considerations has been done:

1. interface bandwidth between the external memory and the SMAC-Engine;
2. with the following calculation

$$avg\#zeros = \frac{sum_{seq}}{total_{seq}} \tag{3.5}$$

where sum_{seq} is the sum among the lengths of all the zero sequences and $total_{seq}$ is the number of zero sequences, both among all layers;

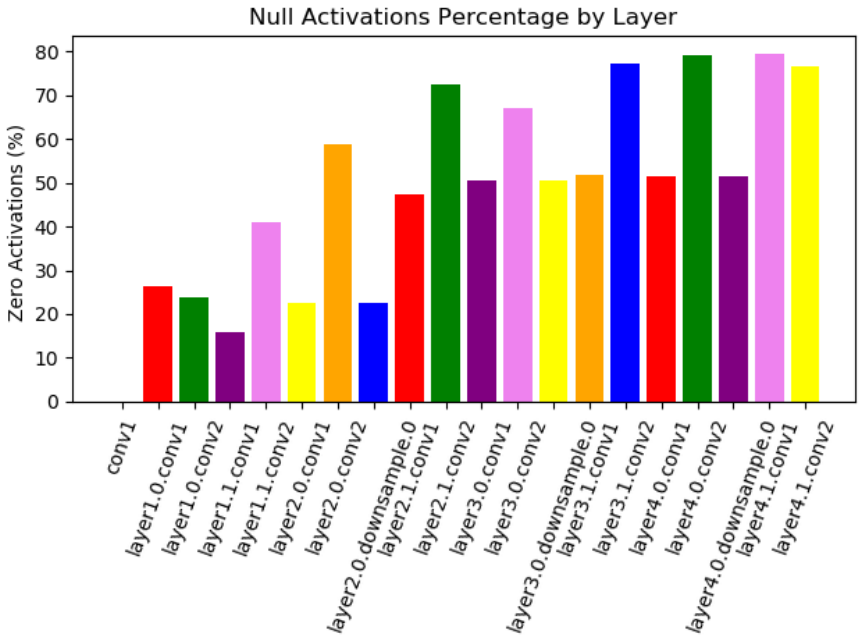

Figure 3.9: Sparsity Analysis for ResNet 18

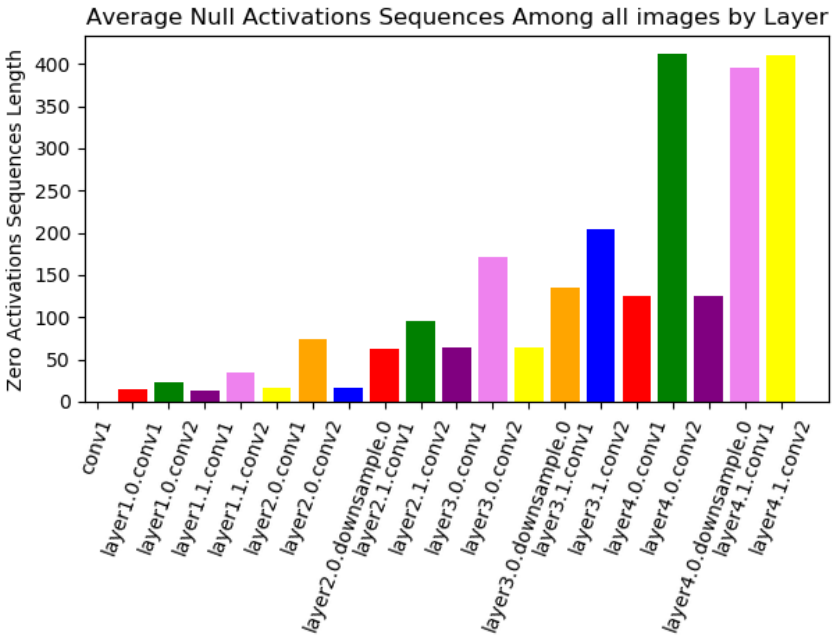

Figure 3.10: Average zero sequences in z direction

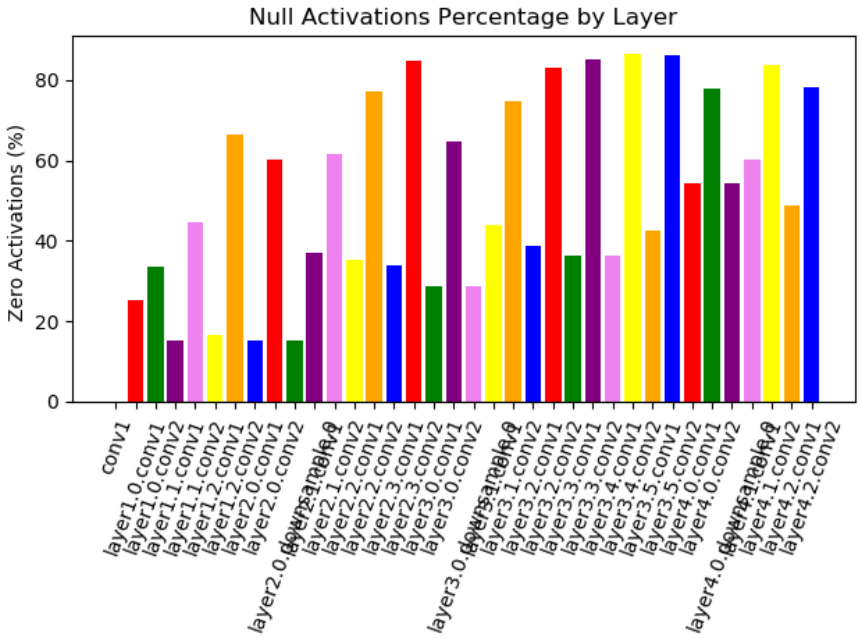

Figure 3.11: Sparsity Analysis for ResNet 34

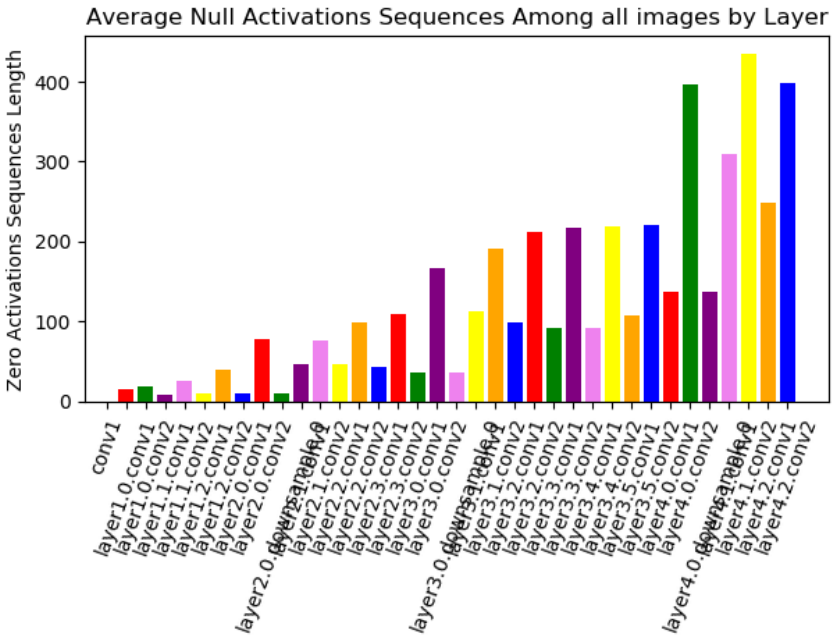

Figure 3.12: Average zero sequences in z direction

Starting from point 1, where it has been considered the architecture present in the dense SMAC-Engine, the usage of 16 activations at each convolution operations is maintained as well as the structure that handles it. So, considering a bandwidth not too much higher respect 128 bit and that was a multiple of 32 bits, 224 bits was perfect in order to consider activations on 8 bits and, as a consequence, 6 bits of zero count result. Now, it is possible to connect to point 2, where from the calculation came out that the value of 63 consecutive zeros was the best approximation as average length of zero sequences.

Chapter 4

SMAC-Engine for Sparse CNNs

4.1 Hardware Design of the Accelerator for sparse data volumes

The hardware implementation of what has been explained in Section 3.2 is properly defined in this chapter. The work on which this sparsity based system is integrated is a SMAC-Engine which has the scope to accomplish convolutions, so Multiply and Accumulate operations, in each layer of a convolutional neural network like the ones of the analysis reported in Chapter 3, ResNet18 and ResNet34.

In the following paragraphs it will be exposed the design units of the system handling sparse data volumes, that are the Decompressor Unit and the Compressor Unit. The first unit is involved in the operations of new activations fetching and decoding of the right position, obtaining also the right fetch of weights in the internal SRAM banks system present in each SMAC-Block PE. The second unit is the one which compress and obtain the data on 14 bits to be streamed off-chip to the external memory TCDM, organized as explained in Section 4.1.1.

4.1.1 Data organization in TCDM

The TCDM is considered in this work as a 7 bank SRAM system, with size 2^{32} x 32 bits, and is used to store kernel weights, input and output activations.
This kind of memory has been yet employed by the DORY (Deployment Oriented to Memory) algorithm as a L1 cache in which the tiling on the memory hierarchy (L_i) of data such as input/output activations and weights is considered in relation to a *layer analyzer* which optimizes and generates code to run the tiling loop,

orchestrate layer-wise data movement and call a set of *backend* APIs to execute each layer of the network, individually [3].

In this work, in order to test in a simpler way the whole system, this memory has been organized as explained in the following points:

- The weights are stored in the part of TCDM with $Base\_Address$ = 0x0 and $Offset$ = 28 x 2^{14}, with the organization shown in Figure 4.1.

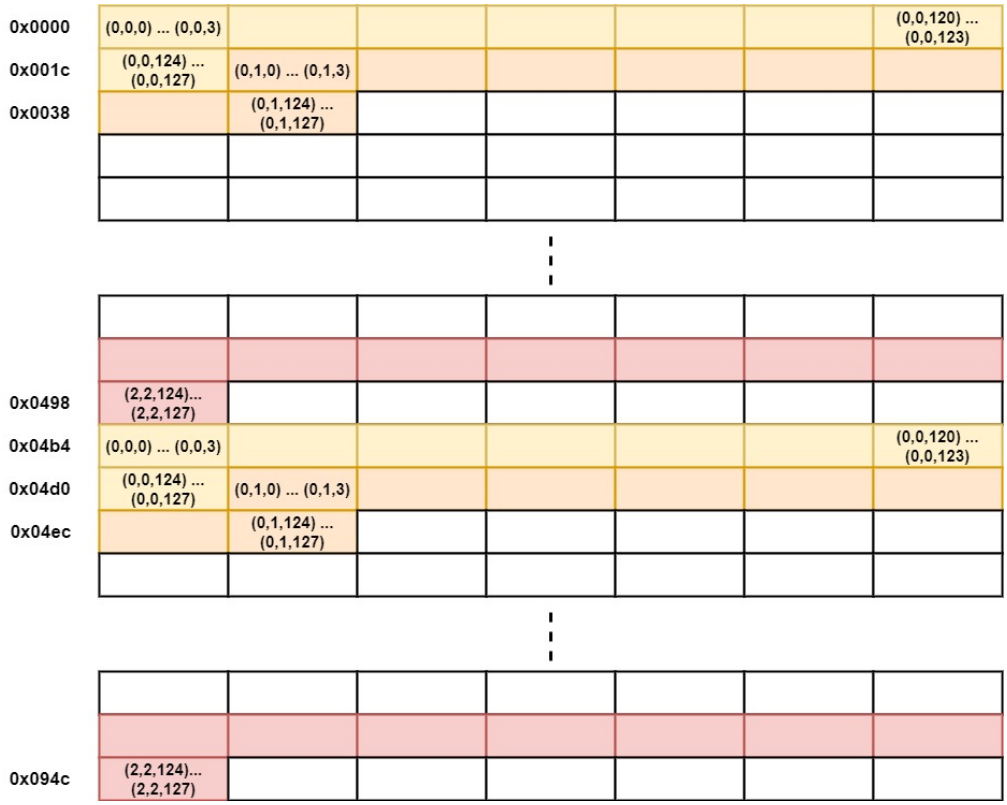

Figure 4.1: Weights Organization

- The activation inputs are stored in the region with Base_Address = 0x0 + 0x1c · 2^{14} and Offset = 0x1c · 2^7;
- The activation outputs are stored in the remaining part of the memory, in the region with Base_Address = 0x0 + 0x1c · (2^{14} + 2^7);

The activation values belonging to each position (x,y) are considered in their compressed form on 14 bits and the offset considered for them inside the memory is 0x70 in hexadecimal, so 112 in decimal value. This value has been get considering the sparsity that characterizes the layer of ResNet18 (3x3x128 convolution volumes)

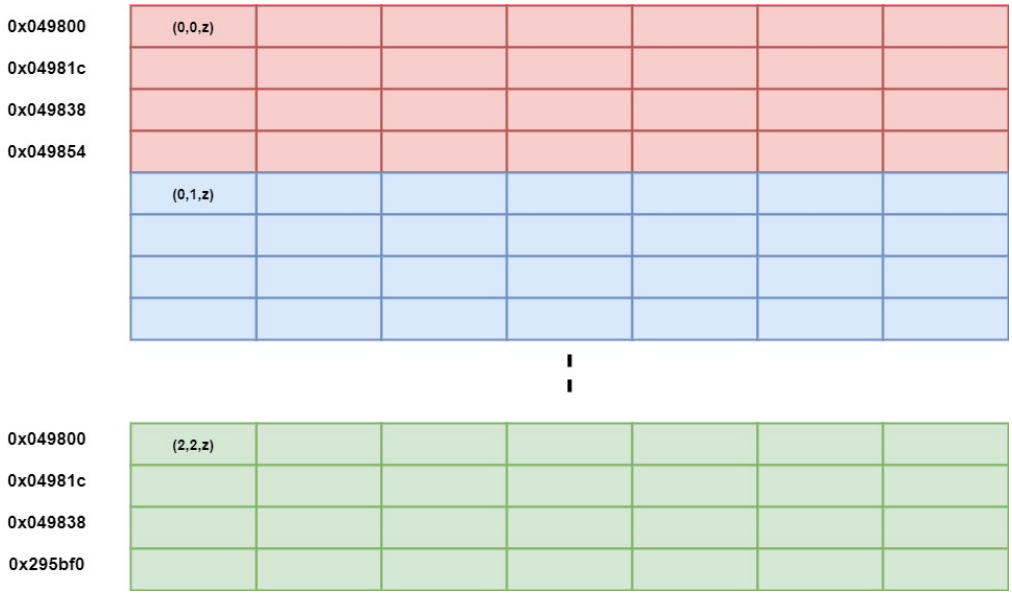

Figure 4.2: Input Activations Organization

involved in this example, that is 50%. So, as reported in Figure 4.2, the activations stored in TCDM are just the one related to a single convolution volume. This choice motivation lies in a simpler test and control for the address generation. The same volume is fetched until we obtain the final result, so the output activation values in the last position.

4.1.2 Decompression System

The main scope of this Decompression Unit is to accomplish for a right convolution operation, fetching the right weights and aligning them with the right activation values. In order to guarantee the good behaviour of this block, the following data path (Figure 4.3) has been implemented. Below, it will be explained the function of each block present in this Unit.

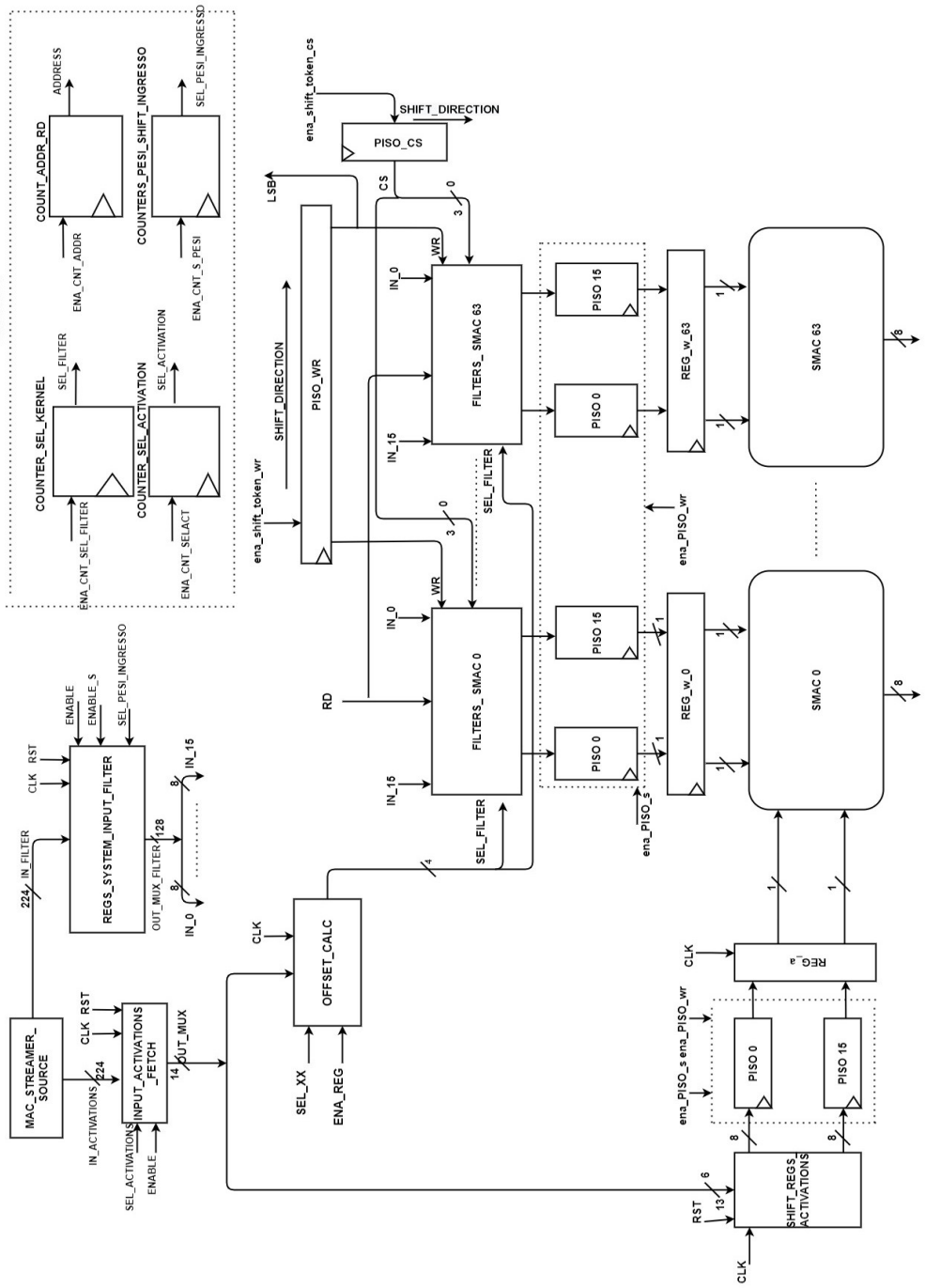

Figure 4.3: Decompression Unit

REGS_SYSTEM_INPUT_FILTERS

The function of this block is to align the packets of weights coming from the TCDM memory with a bandwidth of 224 bit, and dispatch them to the local memories introduced for the contention of the Kernel of the layer involved in the convolution. As reported in Figure 4.4 the data coming from the TCDM correspond to 28 weights belonging to a filter. The first of them is aligned with the MSB side of the "*Reg_weights_align*" register that has 896 bit of capacity. The last data incoming is pushed in the LSB part. This design decision is due to how data have been stored in the TCDM memory (Section 4.1.1). In fact, the weights of each filter are positioned in the ascending order with x and y, and for these data the compression has not been considered.

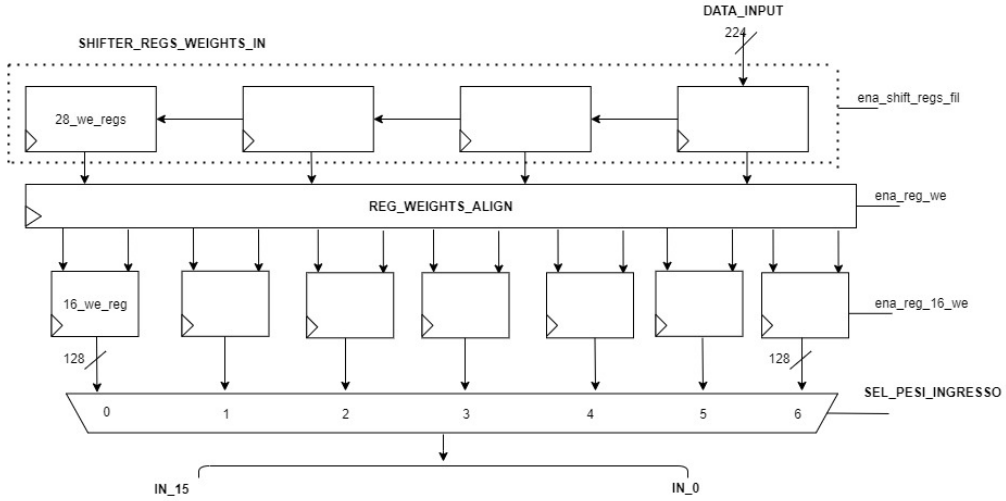

Figure 4.4: Input Weights Registers Block

SHIFTER_WR_TOKEN and SHIFTER_CS_TOKEN

During the writing of weights inside the SRAM banks system, an important role has been assumed by the shifter of the write token that will allow the writing of weights from SMAC_0 to SMAC_63. This operation, if the number of filters is 256, will be done 4 times and each time the write token goes to the 63^{rd} PE the CS token shifts to the next position in order to select the banks related to the next filters group.
While for the writing stage the CS signal shifts until the end of the SIPO register, in the reading stage the 4 CS signals are all at '1', because filters are read all simultaneously.

INPUT_ACTIVATIONS_FETCH

This block is used to select the activation values during the convolution algorithm execution. In the DP_ACT_REG are saved the 224 bits containing 16 data in their compressed format on 14 bits.
An important thing to underline is that the first activation value of the sequence related to the first position of the convolution volume is shifted yet in the shift register and the value of the first weight is read by the LSB byte at the 0x0000 address inside the RAM_INSTANCES block. This is due to how data have been compressed and are read from TCDM.

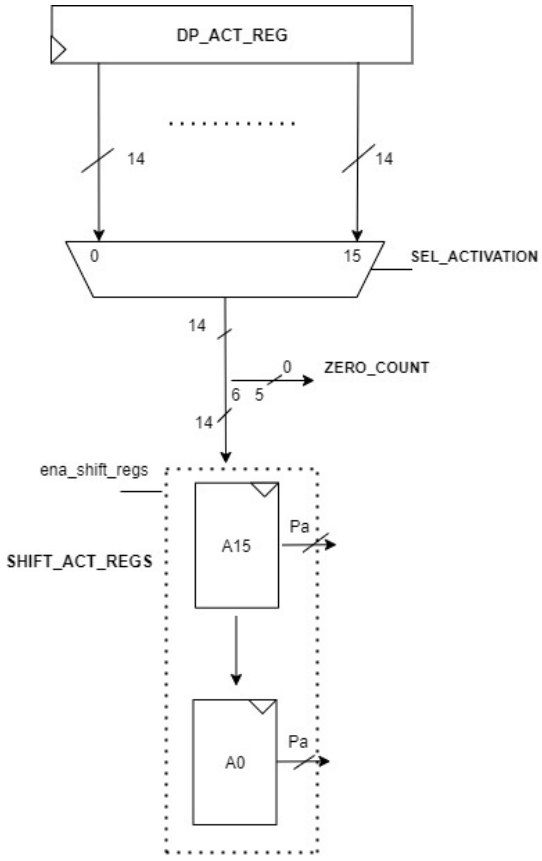

Figure 4.5: Input Activation Fetch Block

The 8 bit in the MSB part is the activation value, that is shifted in the SHIFT_ACT_REGS in Figure 4.5. The shift register is used in order to align the activations with the right value of weight coming from all the filters involved in the convolution. Below, is described the OFFSET_CALCULATION block that is used to verify the right position of weights and activations.

OFFSET_CALCULATION

As reported in Figure 4.7, the algorithm to calculate, for the present activation, the offset respect the next activation value is implemented in this way:

1. the value of zero count is read at the output of the INPUT_ACTIVATIONS_ FETCH unit;

2. as said before, for the first activation read in the convolution volume, the first weight is read by the first location in the local memory simultaneously in all the filters that are involved in the convolution over all the input image.

3. For the subsequent activations compressed in the 14 bit format, the position of weights to be fetched for the next activation value is calculated by means of a system that is composed of 2 adders and 1 subtractor units. The first adder sum the previous offset defined by the previous calculation saved in OFFSET_PREC register and the zero sequence length that separate the current activation read by the next one. Immediately after, this sum result is sum again with an other value that is "1" (to help the calculation of the offset). The 4 LSB of the last result are then read by the PRE_SEL_REG used to store the offset for the next weight value to be fetched from the local SRAM.

4. The offset stored in OFFSET_PREC register is the one defined respect the LSB byte at the current address of the SRAM instances, that could reach a maximum of 16 weights. So, the selection goes from 0 to 15.

5. The choice of SRAM with 128 bits single port inside FILTERS_SMAC$_i$ block is due to the fact that the zero count is done until 63 and, using an offset of 16 it was more affordable for the algorithm.

In Figure 4.6 the yellow region represents the weights positions in the (x,y) coordinate, for example, in the third level of ResNet18; the arrows define the order with which they are stored, from position (x,y,0) to position (x,y,127).

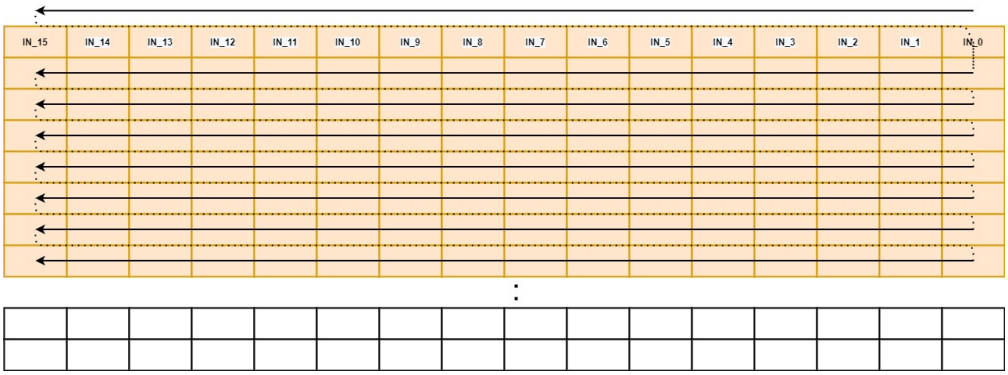

Figure 4.6: SRAM weights organization

FILTERS_SMACi

In this block, there is the same structure that is on the right side in Figure 4.7, but repeated 4 times (SRAM with shift registers). This because, leaving intact the structure of the dense SMAC-Engine where there was 4 accumulators for 4 different filters in each *SMAC-block*, this could be affordable while doing the calculation for each of these filters.
There are 64 units of this structure as reported in Figure 4.3, and in each of them the filters are organized in the following way:

- The filter number 1 in each FILTER_SMAC$_i$ is the one that, from the SMAC 0 to the SMAC 63, represents the positions from (x, y, 0) to (x, y, 63);

- The filter number 1 in each FILTER_SMAC$_i$ is the one that, from the SMAC 0 to the SMAC 63, represents the positions from (x, y, 64) to (x, y, 127);

- The filter number 1 in each FILTER_SMAC$_i$ is the one that, from the SMAC 0 to the SMAC 63, represents the positions from (x, y, 128) to (x, y, 191);

- The filter number 1 in each FILTER_SMAC$_i$ is the one that, from the SMAC 0 to the SMAC 63, represents the positions from (x, y, 192) alla (x, y, 255);

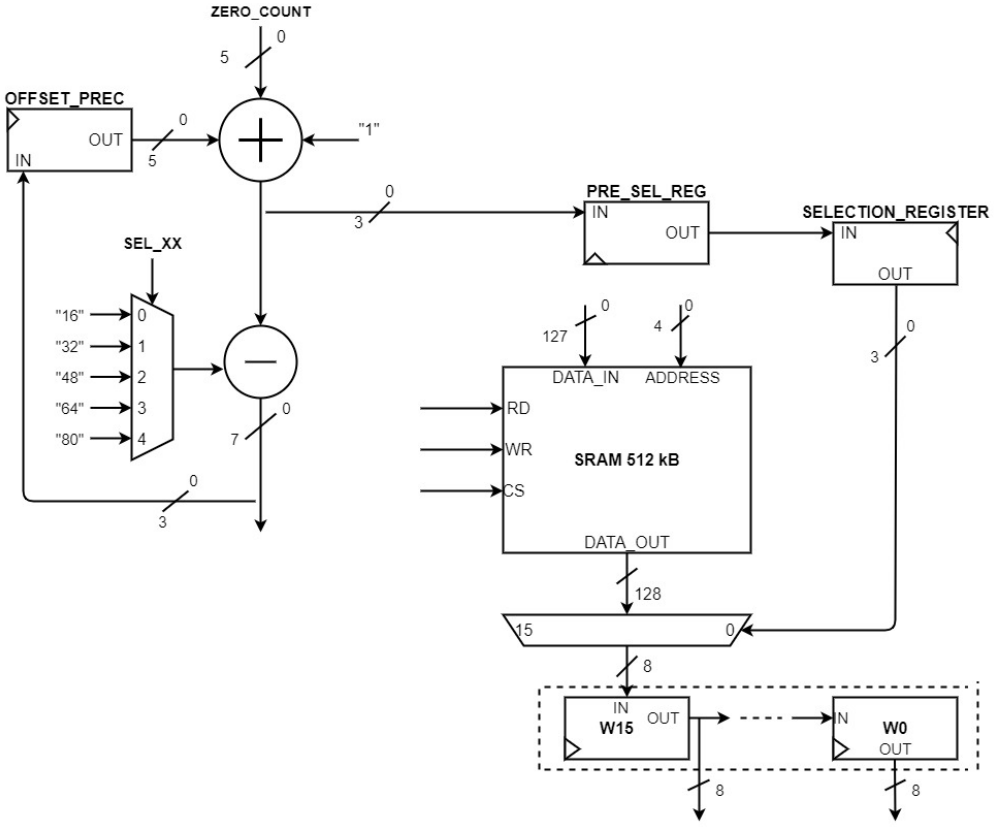

Figure 4.7: Offset Calculation Block

4.1.3 SMAC-Engine

This hardware architecture employs the convolution operation by means of an array of AND ports, *accumulators* and an *adder* to accomplish the Multiply and Accumulate operations but considering a serial input and not a parallel one. In this case, the parallelism considered for both activations and weights is 8 bits, and these shift from the LSB to the MSB during the convolution operations.
Moreover, inside each AC block there is one or more accumulator registers in which are accumulated the partial results of the multiply and accumulate operations. In particular, in the AC2 and AC3 blocks there are four accumulator registers that are used to save results from four different filters at a time. While in the first are accumulated the partial results during the shifting of bits, in the second one are accumulated the partial results coming from the entire convolution between 16 activations and 16 weights, among all bits. In AC1, instead, there is just one register to hold the results from the multiplication bit by bit of weights and activations.
The *neg_block*, moreover, is used to change sign of the activations in output while

at the output of the SMAC block there is a ReLU function handler in which a mux decides by means of the MSB of the output activation if the result must be clamped to zero.
Before the data is streamed off the SMAC block, a quantization is done in order to have a precision on 8 bits.

4.1.4 Compression System

In the compression system reported in Figure 4.8 the main scope is to execute the compression of all the output activations for the position (x,y) of the output feature map, that in a ResNet18, for instance, could variate between 64 and 512. The compression rules are described in Section 3.2.2, reporting also some examples. Below, are defined the blocks that constitute this system.

INPUT DATA ACTIVATIONS

The input is obtained by means of a 64 to 1 multiplexer that, a number of time equal to the number of filters used in each SMAC-block, are fetched from 0 to 63 coming with the order described in Section 4.1.2.

ZERO COMPARATOR

The zero comparator is used to detect null activations. This is important for the recognition of zero sequences and, when the zero count comes to 63, it stops and the value is compressed and shifted in the shift register. If the zeros are less than 63, and a new non-null activation comes into the system, the zero count stops and data are shifted.

ACTIVATION WRITING

Here, some ports are used in order to obtain the signal that enable the writing of the compressed data in the shift register. The system is reported in Figure 4.8, and it could be noticed that the output data come into a series of multiplexer in which the first defines which one of the output data from REG14 or REG14_1 can pass, and the second one chooses between a zero or the data compressed. This is due to the fact that sometimes the shifting of data can finish before the 16^{th} shift in the shift register, so it must be filled until the end of shifting with zeros in order to send the values in the right positions that then will be written in the TCDM memory. In fact, the fetching of data is done, further, in a sequential order and no zero values that have not been involved in the compression should be present in the middle of this sequence.

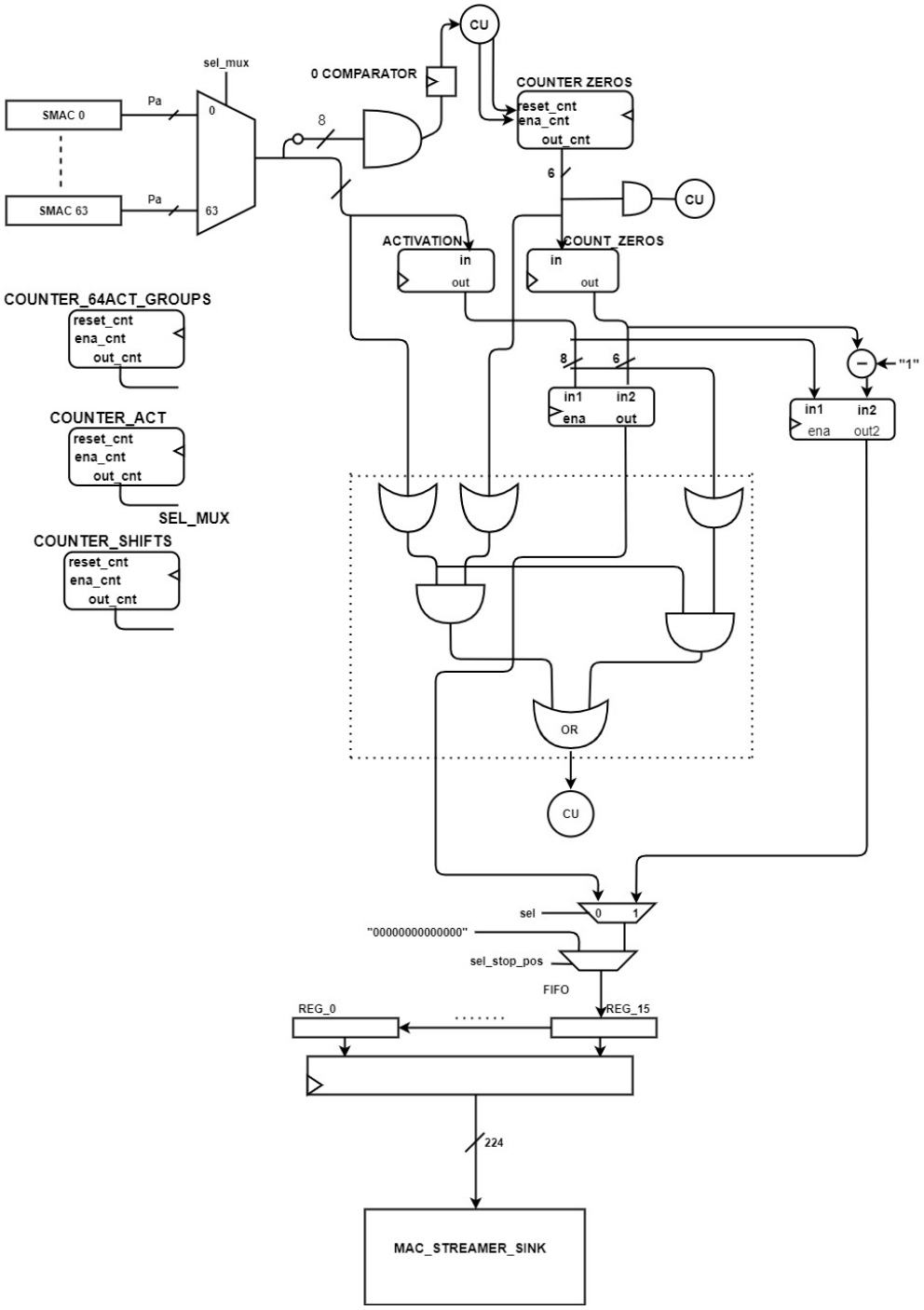

Figure 4.8: Compression Unit

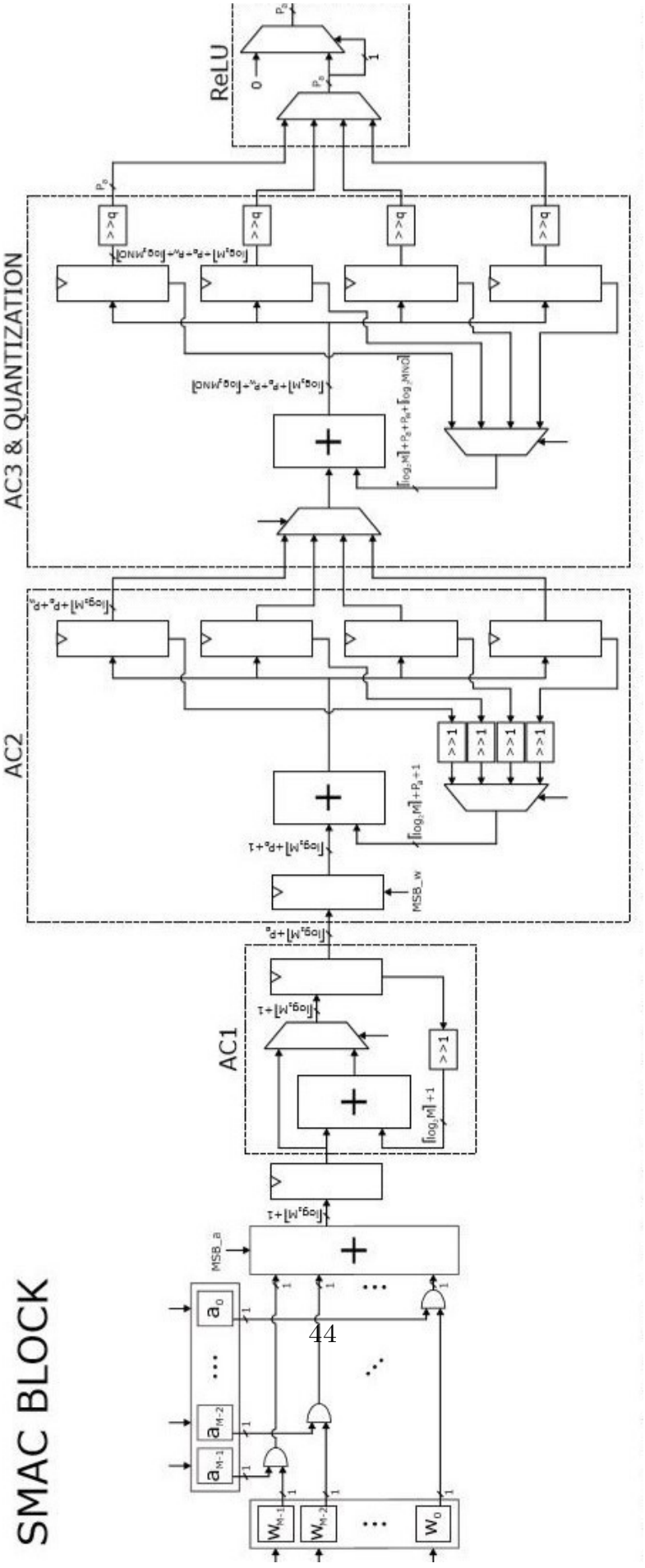

Figure 4.9: SMAC Block

4.2 Control Unit

The Control Unit is made up of two main components, whose functions are properly described below.

4.2.1 COUNTERS

The counters are involved in both the generation of signals for the evolution of FSM through its states and for some selection signal inside the data path.

COUNTERS-SMAC

- $FILTERS\_COUNTER$: this counter is used to select the internal accumulator in both the AC2 and AC3 blocks. In this case, it is used to select the proper accumulator register in each of the AC blocks. In case all the 4 filters in each SMAC Block are needed, the maximum value is set to "4".

- $CTRL\_CNT\_DONE\_QUANT$: this counter has to be programmed with the amount of shifting necessary to perform quantization. The shifting amount will vary depending on the number of operations done on a subset of 16 activations in the total volume of convolution.

- $CTRL\_CNT\_RELU\_MUX$: this counter is used as selector for the output activations that need to be clamped to zero with ReLU function. The value this counter is programmed with can be shared with $FILTERS\_COUNTER$, since the number of filter groups to deal with will be coincident with the number of selection signal values the multiplexer, before the ReLU block, is expected to switch among.

- $CTRL\_CNT\_IN\_VOL$: this counter controls how many convolution volumes have been computed and counts them each time the compression of each output position has been done, until the full convolution of the input volume is obtained.

COUNTERS-DECOMPRESSOR

- $FILTERS\_COUNTER$: here, this counter is used to select the right filter between the 4 RAM-instances in each FILTERS_SMACi. It is important both in the writing stage and in the reading stage. In the last one, it is used for the choice of the input data from RAM banks as well as for the selection of the right accumulator register, important during the convolution computation.

- *ADDRESS_GENERATOR*: this is used to index the Memory bank while writing and reading it. The MAX_VALUE_ADDRESS could differs, depending on the convolution layer. For example, in case of a filter of 3x3x64 the maximum address in the RAM-Instances is 36.
- *SEL_ACTIVATIONS_COUNTER*: this is used for the selection of the right 14 bit word in input from the "mac-streamer" block, that has a bandwidth of 224 bit, so the maximum value of the counter is MAX_VAL_SEL_ACT = "63".
- *COUNTER_SEL_FILTERS_IN* : this counter is used to select the input values of weights, 16 at each selection, with the right direction. So, they are selected from 0 to 6. These are selected from the output of the block described in Subsection 4.1.2.
- *SHIFT_FILTERS_COUNT*: this is used to count the shift of 28 weights in the input shift registers at the input of the block described in Subsection 4.1.2.

COUNTERS-COMPRESSOR

- *FILTERS_COUNTER* : in the compressor block, this counter is useful for the selection of the groups of activations at the output at the end of the convolution algorithm for the 16 activations involved. So, the counter recounts from 0 to 3, so MAX_FILTER_GROUP = "3".
- *ACTIVATION_Z_SELECTOR_COUNTER*: this is used to select the output activation related to the channel z, belonging to one of the filters group.
- *COUNTER_ZERO_SEQUENCES*: this is used in order to count the zeros inside each sequence related to the offset that defines the position of the non-zero activations.
- *COUNTER_SHIFTS_REGS_OUT*: this counter counts how many times a 14 bit word is written in the shift register until a 224 bit stream to send off-chip is obtained. So, the maximum value is "16".

4.2.2 FSM Low-Level

The FSM Low-Level is organized in the following groups of states:

- **Writing Weights States**: these states are the ones in orange in Figure 4.10, and are S0, S1, S2, S3, S4, S5, S6 and S7. Through these states the data of

weights belonging to different filters and coming from the TCDM are written in the local memory that is a collection of SRAM 32x128bits used as further cache system in order to make the less possible accesses to the TCDM memory.

- **Reading Weights and Activations States**: these states, the blue ones in Figure4.10, are involved in the reading step of both activations and weights, where only activations are read by the TCDM.
- **Computation States**: the computation states are similar to the ones in the previous work; the only things that are changed are the input signals used as controls to let the machine evolve through all states.
- **Compression States**: these are reported in Figure 4.11, and are involved in the last step related to the compression algorithm. It has been implemented as the last group of states before defining if the operations with the convolution volume or with the entire input data volume of that layer is finished.

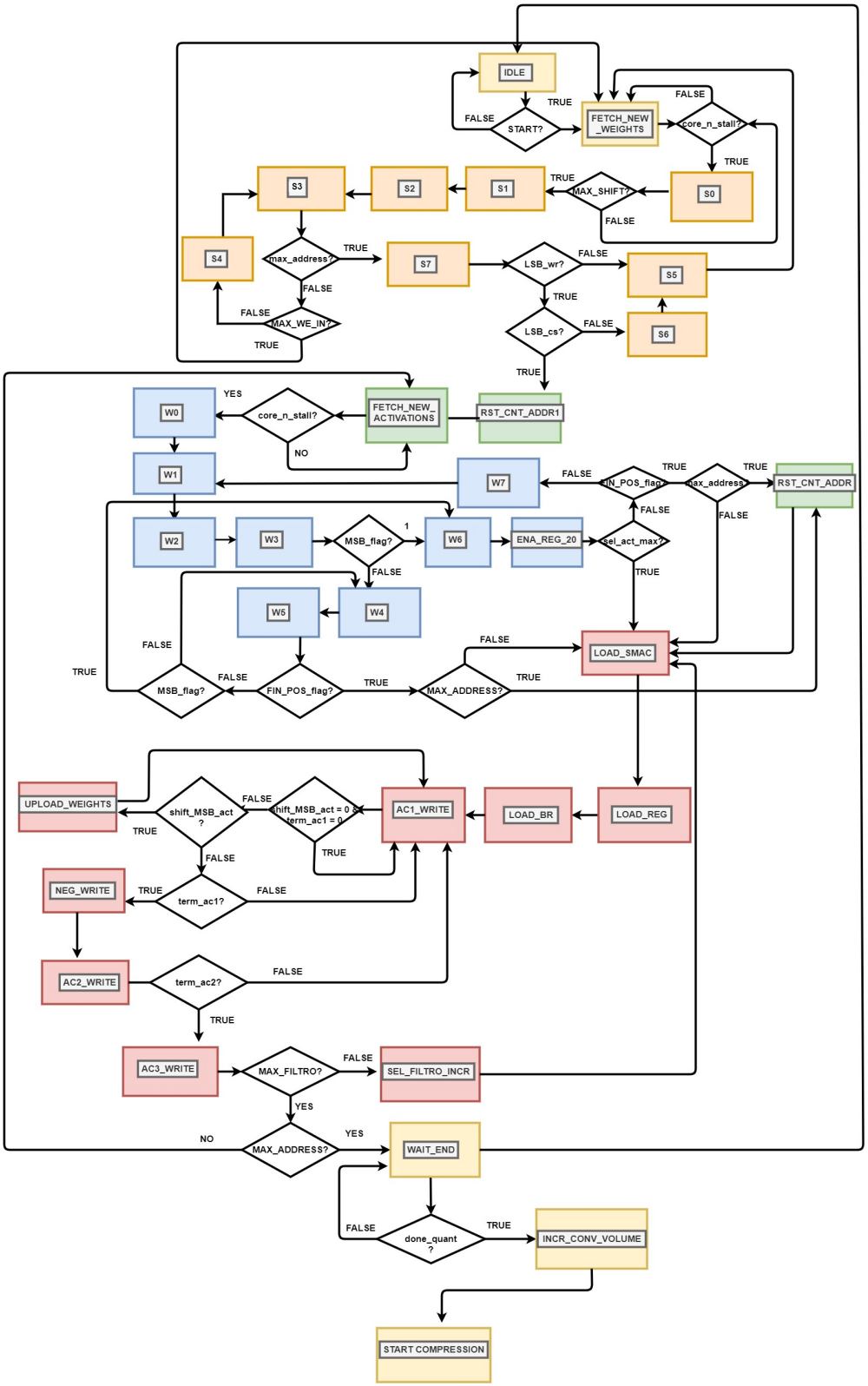
IDLE
START?
FETCH_NEW_WEIGHTS
core_n_stall?
S0
MAX_SHIFT?
S1
S2
S3
S4
max_address?
S7
LSB_wr?
S5
LSB_cs?
S6
MAX_WE_IN?
FETCH_NEW_ACTIVATIONS
RST_CNT_ADDR1
RST_CNT_ADDR
W0
W1
W2
W3
W4
W5
W6
W7
MSB_flag?
FIN_POS_flag?
MAX_ADDRESS?
ENA_REG_20
sel_act_max?
LOAD_SMAC
UPLOAD_WEIGHTS
shift_MSB_act ?
shift_MSB_act = 0 & term_ac1 = 0
AC1_WRITE
LOAD_BR
LOAD_REG
NEG_WRITE
term_ac1?
AC2_WRITE
term_ac2?
AC3_WRITE
MAX_FILTRO?
SEL_FILTRO_INCR
MAX_ADDRESS?
WAIT_END
done_quant ?
INCR_CONV_VOLUME
START COMPRESSION
TRUE
FALSE
YES
NO

Figure 4.10

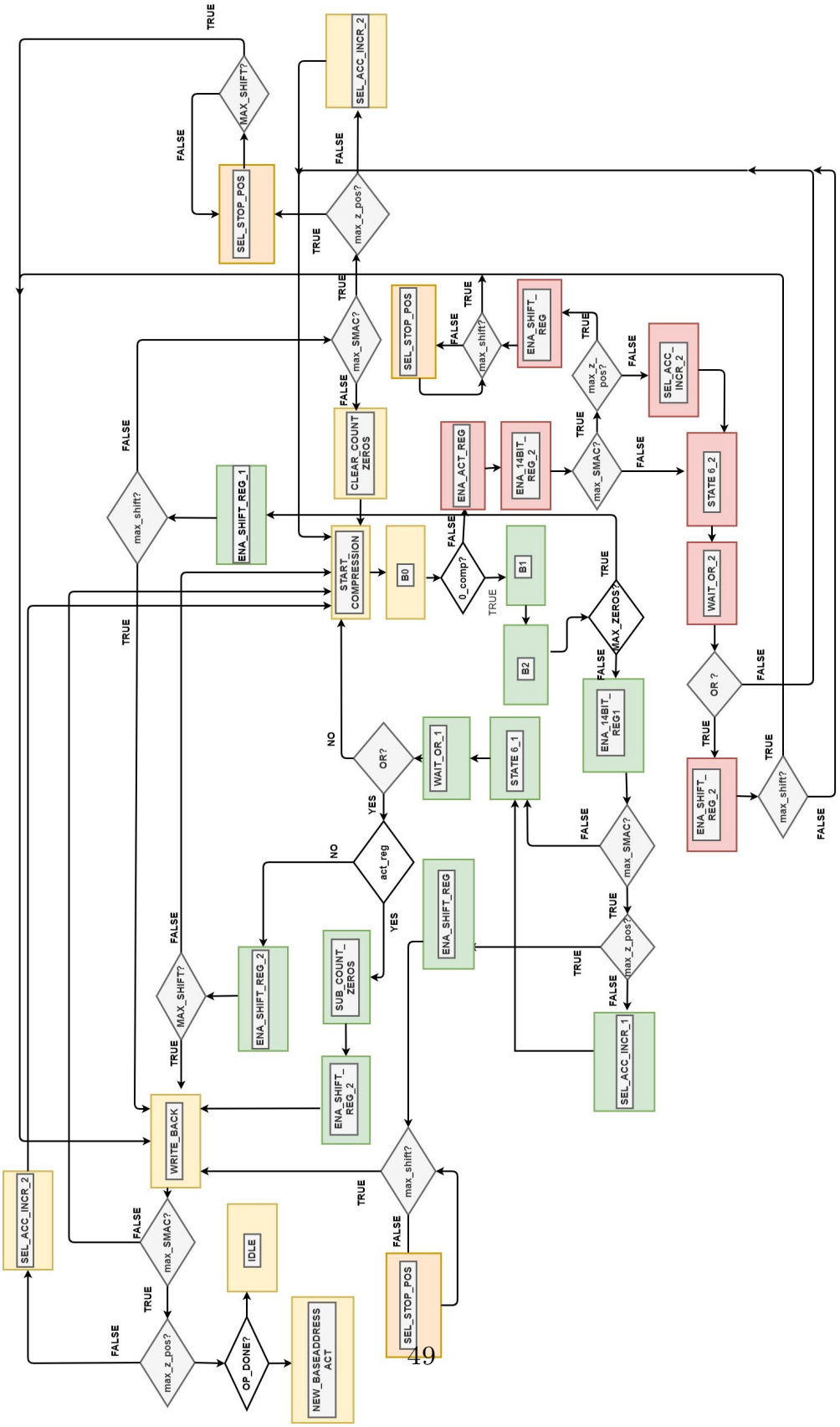
IDLE
START_COMPRESSION
B0
B1
B2
0_comp?
CLEAR_COUNT_ZEROS
ENA_ACT_REG
ENA_14BIT_REG_2
ENA_14BIT_REG1
ENA_SHIFT_REG
ENA_SHIFT_REG_1
ENA_SHIFT_REG_2
SEL_ACC_INCR_2
SEL_ACC_INCR_1
SEL_STOP_POS
SUB_COUNT_ZEROS
WRITE_BACK
WAIT_OR_1
WAIT_OR_2
STATE 6_1
STATE 6_2
NEW_BASEADDRESS ACT
OP_DONE?
OR?
OR ?
act_reg
max_shift?
max_SMAC?
max_z_pos?
MAX_ZEROS
MAX_SHIFT?
TRUE
FALSE
YES
NO

Figure 4.11

Chapter 5

Integration on PULP HWPE

5.1 PULPissimo system and HWPE

The design of the SMAC-Engine sparsity based is now integrated in a HWPE of a PULP platform, whose IPs are available open source.
An introduction of the HWPE's communication protocol with internal engine and with PULP is given in the next Sections, together with an explanation of how the integration with SMAC-Engine has been carried on.

5.1.1 The Hardware Processing Engine

The HWPEs have been developed as special-purpose and memory-coupled accelerators that live within the PULP system realized by Zurich ETH and the University of Bologna. This integration has the scope to increase the system performances and energy efficiency.

In figure 5.1 is shown the structure of an HWPE accelerator. The interfaces that made up this module are reported below:

1. a streamer interface, needed to interface the HWPE internal engine with the Tightly Coupled Data Memory (TCDM), a 64 kB memory organized in eight word-interleaved SRAM banks that is directly used by all the resources of the cluster without the need of an external DMA;

2. a control/peripheral interface, used to program the HWPE by means of a register file, a microcode processor and a control FSM. The peripheral interface is used to program the control unit.

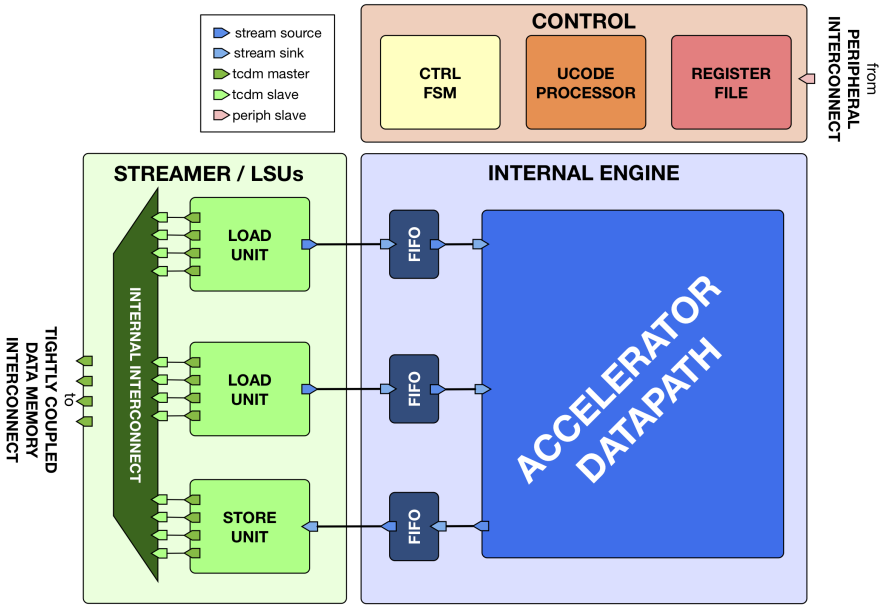

Figure 5.1

3. an internal engine, where the data path lies.

In the following, some further details regarding each of these sub-modules will be provided.

The streamer

The streamer is the module that carries the flow of data from or to the TCDM memory, so flowing from a *source* to a *sink* direction. This stream is handled by means of a protocol based on a *two signal handshake* and a data payload transferring. The protocol consists of the following rules:

1. At the clock cycle when both valid and ready signals are at '1', it occurs the handshake.

2. The data (multiple of 32 bits) and *strb* (indicating which of the bytes in the data are considered valid) can change their values or when the valid signal is deasserted or when, although the valid signal is at '1', the current cycle is the one following the handshake occurrence.

3. In order to avoid any deadlock while changing their states, the valid and ready signals are linked to each other in this way: while the assertion of valid doesn't depend on the state of ready, the assertion of ready depends in a combinatorial way on valid state.

4. The deassertion of valid can occur only in the cycle after a valid handshake occurs, in order to assure the correct consumption of valid data.

The interface between the HWPEs and the TCDM shared external memory is based, instead, on a TCDM protocol that connects a master to a slave by means of a request/grant handshake following the rules below:

1. At the clock cycle when both valid and ready signals are at '1', it occurs the handshake for both read and write transactions.
2. An r_valid must be asserted the cycle after a valid read handshake, and the r_data must be valid on this cycle. This is due to the tightly-coupled nature of memories: if the memory cannot respond in one cycle, it must delay granting the transaction.
3. While the assertion of *req* doesn't depend on the state of *gnt*, the assertion of *gnt* depends in a combinatorial way on the *req* state, always to avoid deadlock.

The control

The control module embeds three different sub-modules:

1. a control FSM, which will need to be designed from scratch as it will be specific for the developed accelerator;
2. a memory-mapped register file implemented with latches to save area and power, which includes two different set of registers:
   - *generic registers* (or job-independent): these registers store parameters that should not change during the execution of multiple jobs by the control module;
   - $job - dependent\ registers$: in these registers, parameters such as base addresses for each type of data (input activations, filters, output activations), controls for the engine such us the maximum value for the counters and so on are stored; their peculiarity, respect to the others, is that they can change at every new job and even when the HWPE is executing its tasks.
3. a microcode processor: the processor supports 2 different type of registers:
   - four R/W registers
   - twelve R/O to store parameters

The microcode task is defined by the $hwpe - ctrl - ucocode$, which implements the updating of the offsets respect to the base address of the type of data currently fetched from TCDM.

The engine

The current design of the accelerator is capable of handling sparse data of feature maps, and it has been properly integrated inside the PULP platform. In fact, it accomplishes the handshake with the streamer respecting the right protocol, defined in subsection 5.1.1. This system has been integrated inside HWPE by means of RTL description modules that let the interface with PULP be compliant with the bandwidth defined for the SMAC-Engine.

5.1.2 Integrating SMAC-Engine sparsity based in a HWPE

The repository in which are stored the developed SMAC-Engine *rtl* files with the ones describing the integration with HWPE is the *hwpe-mac-engine/rtl*. The other folders called *hwpe-stream* and *hwpe-ctrl* hold the files in which are described the resources of the HWPE, and have been changed in order to be used for the particular interface of the sparsity based SMAC-Engine.
In the following subsections, a detailed description about the RTL files and the HWPE files changes is reported, respect to the dense architecture of SMAC-Engine.

mac_engine.sv

In this module, there is the definition of two ports on 224 bits:

- source port a_i;
- sink port d_o.

Two source ports that were present in the standard implementation of the engine have been removed, and they were b_i and c_i. In the top view of the SMAC-Engine data path integrated as engine in the HWPE, the ports linked to the interface with streamer are the $input\_activations\_filters\_data$ on 224 bits (bandwidth of PULP), the $a\_i.valid$, associated to the $core\_n\_stall$ signal in SMAC-Engine, that assert the handshake with the TCDM memory when the data are valid to be elaborated by the engine, and the maximum value for counters that constitute the control unit of the SMAC-Engine and the parallelism of data of activations and weights, that have been set both to 8 bits for the test. In order to wake up the SMAC-Engine, the $core\_n\_stall$ has been used. Other signals coming from the low level control unit have been reported in the port assignment and communicate with the high level finite state machine described in mac_fsm.sv. The evolution of the FSM through its states is described further in this chapter.
Finally, two processes to generate the output valid signal as well as sample the output data in an output register have been defined before connecting them to the output data d_o.

mac_streamer.sv

In this module, the first change compared to the provided example was to extend the FIFO depth from 2 to 8, to better decouple the producer and the consumer and reduce the probability stall occurrence.
Furthermore, the $DATA\_WIDTH$ of source and sink stream interfaces have been extended from 128 bits (of the dense matrices based SMAC-Engine) to 224 bits (being always a multiple of 32 bits), that is the bandwidth of the new SMAC-Engine.
Another change, compared to the previous work, was to extend the "virtual" number of ports to the TCDM from eight to fourteen. These are considered *virtual* as it is like instantiating fourteen ports, but only seven of these are physically there: all these fourteen ports seems to be attached to the TCDM physically but they are not. This is necessary to handle 224 bits stream at the input and at the output. Indeed, a TCDM multiplexer can then be used to drive more input *virtual* TCDM channels into a smaller set of master ports (seven). Hence, together with the definition of a *virtual_tcdm* interface, a *hwpe_stream_tcdm_mux* has been allocated to handle this.
After defining the *virtual_tcdm* interface, the corresponding streams TCDM ports have been connected to the input of the multiplexer instantiated above and their DATA_WIDTH again extended from 128 bits to 224 bits. The unnecessary sources (b and c) have been removed from the multiplexer input. Finally, for both streams' FIFOs i_a_fifo and i_d_fifo, the DATA_WIDTH has again been extended to 224 bits, the FIFO_DEPTH to 8 and the parameter LATCH_FIFO has been set to 0, as the latter was not needed.

mac_package.sv

In this module, there is the definition of the *typedef_structures* defining blocks of signals that are employed to make possible the communication between the different resources of the HWPE, such as the engine, the streamer and the fsm.
However, other parameters have been defined here. First of all, the MAC_CNT_LEN definition has been left unchanged, as this will be a parameter employed by the *mac_ctrl.sv* module to define the dimension of the transaction size for both the weights and output activations. The transaction size is a quantity that is sent to the *mac_fsm.sv* and that helps this high-level FSM generating the correct information for the address generators in the streamer in order to obtain the right number of 224 bits words to be streamed.
The following parameters define the job-dependent register file addresses (or indexes to their content) and have been changed to match the quantities needed by the SMAC-Engine.

- MAC_REG_X_ADDR: this is the address to a register in the register file

storing the base address for the input activations in the TCDM;

- MAC_REG_W_ADDR: this is the address to a register in the register file storing the base address for the input weights in the TCDM ;
- MAC_REG_Y_ADDR: this is the address to a register in the register file storing the base address for the output activations in the TCDM;
- MAC_REG_CNT_PROG1: this is the address to a register in the register file with values like $max\_val\_in\_vol$;
- MAC_REG_CNT_PROG2: this is the address to a register in the register file with values like max_pos;
- MAC_REG_CNT1: at this address are stored some of the maximum values for counters in the control unit, which are max_act, max_zeros, max_groups_act, max_shifts, max_addr;
- MAC_REG_CNT2: at this address are stored some of the maximum values for counters in the control unit, which are max_shift_28we, max_sel_act, max_sel_fil, $max\_sel\_we\_in$, max_selxx.

mac_fsm.sv

The FSM high level has been implemented in order to handle the addressing of the TCDM, whose content has been defined by means of the $tb\_dummy\_memory.sv$. The addressing is divided into 3 steps:

1. firstly, the weights are fetched all together and stored in the local memory in SMAC-Engine;
2. secondly, during the computation of the convolution algorithm are fetched just the activations in input to the layer;
3. finally, during the compression of the results related to one output position, the 224 bits stream are sent off-chip to the TCDM memory until all the activations have been compressed.

Accordingly to this kind of algorithm, the "$hwpe - ctrl - ucode.sv$" has been implemented in order to define the offset during the updating of the address, distinguishing between input activations, output activations and weights.

In the following figure is reported the FSM high-level control flow.

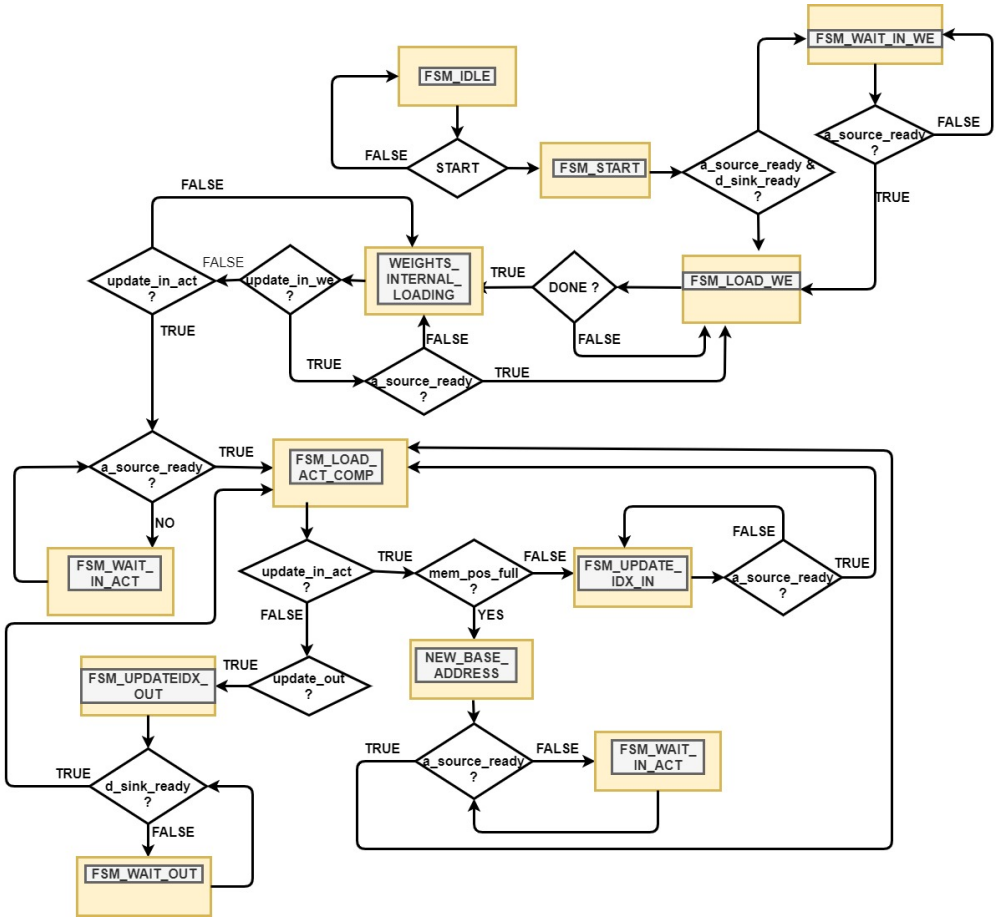

Figure 5.2: FSM High Level

mac_ctrl.sv

In this module, there is the integration of two blocks that are *mac_fsm.sv* and *hwpe_ctrl_ucode*. The first one has been described in the previous subsection, while the second one is the control related to the address generation, and it is used to change offset during the updating of the addresses.
It consists of just 2 states, IDLE and OFF_WORKING, respectively setting the default value at '0' for the offset registers related to the three types of data and updating them while the *mac_fsm* evolves through its states.

mac_top.sv

The *mac_top.sv* module is the top view of streamer, engine and control modules. These have been changed in their port declarations and the DATA_WIDTH has been set to 224 bits instead of 128 like in the previous work. The "enable" signal

has been set to '1', and any further adjustment occurred to this module.
Then, a module called *mac_top_wrap.sv* has been instantiated in order to be put inside the test bench to perform the needed simulations. In this module the interfaces protocols have been inserted.

Chapter 6

Analysis Results

In this chapter it will be reported both the analysis of the logic behaviour and the logic synthesis of the architecture under test. Respectively, the tools used in order to get the aimed results are *QuestaSim* 10.6*c* and *Synopsys Design Compiler*.

6.1 QuestaSim Simulation Results

In this Section, the simulations done with QuestaSim10.6 are reported and explained. In particular, from these simulations came out the proper functionality of each block described in Chapter 4, associated with the current states of the low level FSM. During the writing phase, data with a BW of 224 bits are fetched from TCDM with the address generation explained in Chapter 4. Then, the algorithm with which weights are aligned with the activations coming in input to the accelerator can be accomplished together with the one that implements the convolution operation between 16 activations and weights. At the end, the compression for the output activations at the position (x,y) can be carried on, sending to TCDM a 224 bits compressed activations data every time is required.

6.1.1 Testbench Results

WRITING PHASE

In this phase, the weights related to all the filters involved in the convolution layer are saved on a system of SRAM memory banks in which each SRAM of 1024 kB is instantiated for each filter. In Figure 6.1 is reported the stage in which a fetch of 224 bits of weights, so 28 weights, has been done. Then, they are aligned by means of a shift register with 4 registers with the subsequent data loaded on chip as reported in Figure 6.3, saved in a 896 bit register and then in 7 registers containing

16 weights each one. Then, in order to save them in the local SRAM banks, they are selected 16 weights by 16 weights from 0 to 6, so with a bandwidth of 128 bits.

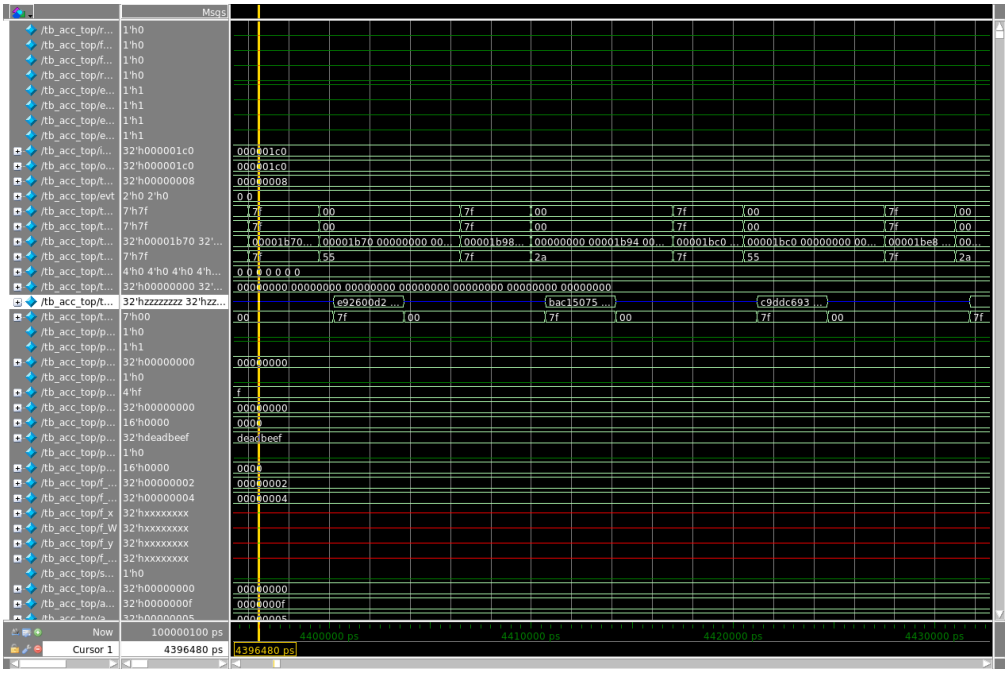

Figure 6.1: Weights fetching from TCDM

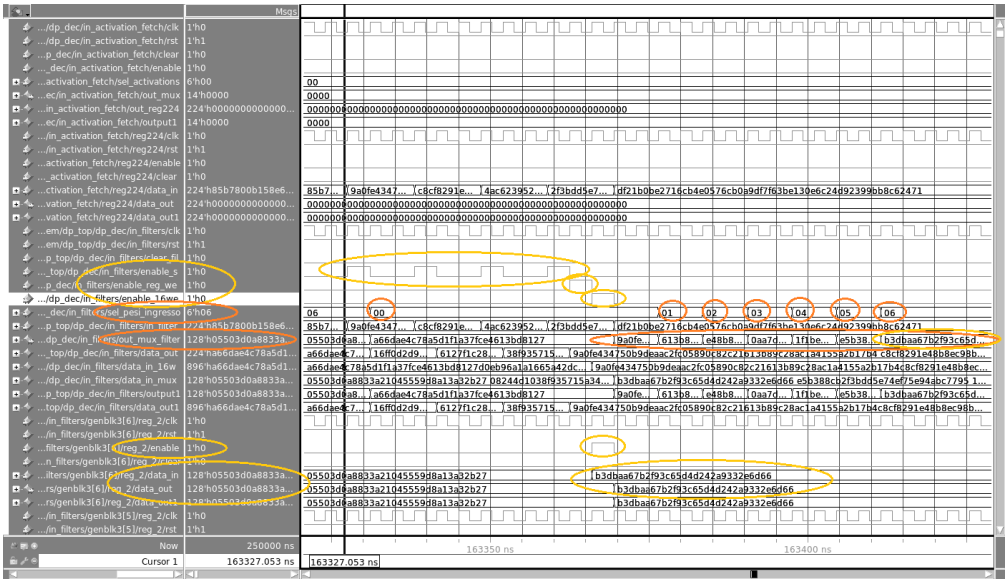

Figure 6.2: Weights loading in REGS_INPUT_SYSTEM

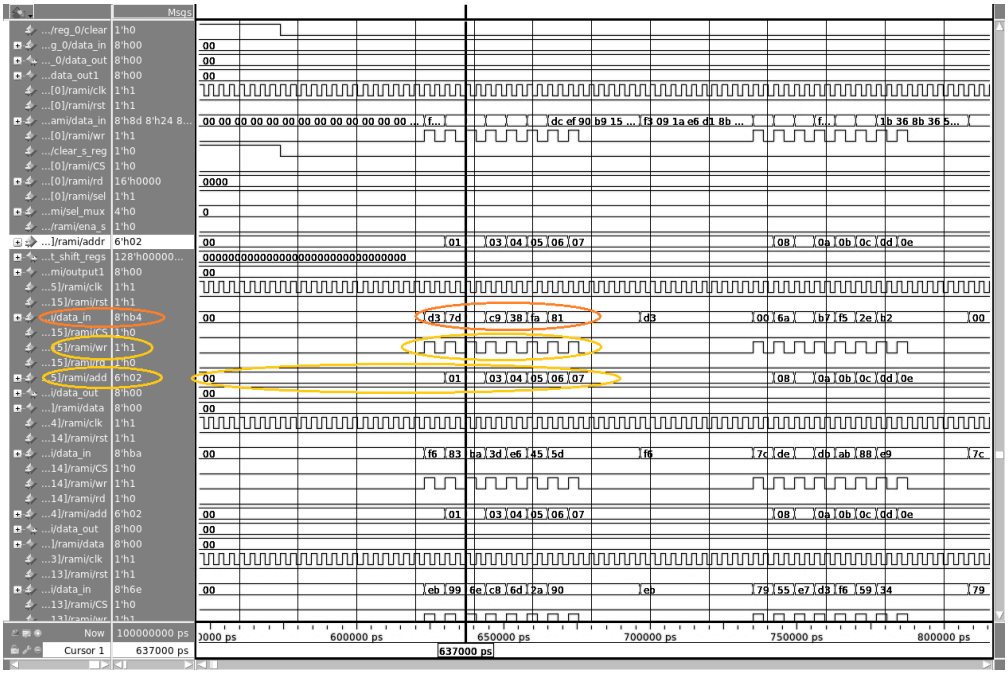

Figure 6.3: Weights loading in the SRAM banks

READING PHASE

During the reading stage, a particular algorithm is implemented and this is well reported in the analysis of Figure 6.4. As it is possible to notice, only one of the bytes in the 128 bits data at the output of the SRAM memory is read. This is controlled by an algorithm in which, by means of the sum between the number of zeros in the sequence and the offset respect to the first Byte at each address, the next weight position is detect for the next value of activation. Taking an example from the simulation in 6.4, in green are highlited the values of *zero count* and of the selection signal for the next weight position in a memory instantiation (one filter). The first is 30, so being the first data fetched from the first byte position in the address 0x0000 the value fetched after is at position 15 that is also the selection of the multiplexer at the output of the RAM.

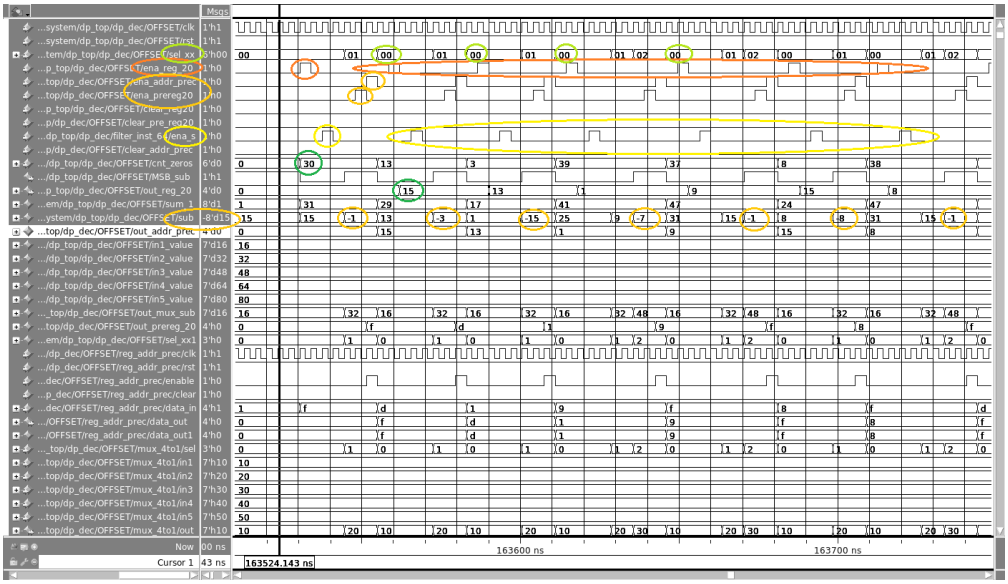

Figure 6.4: Offset detection

In Figure 6.5 the reading phase is shown from the memory point of view, and as reported here the *read signal* and the *chip select* assertion occurs in the clock cycle before the assertion of the *shift enable* signal.

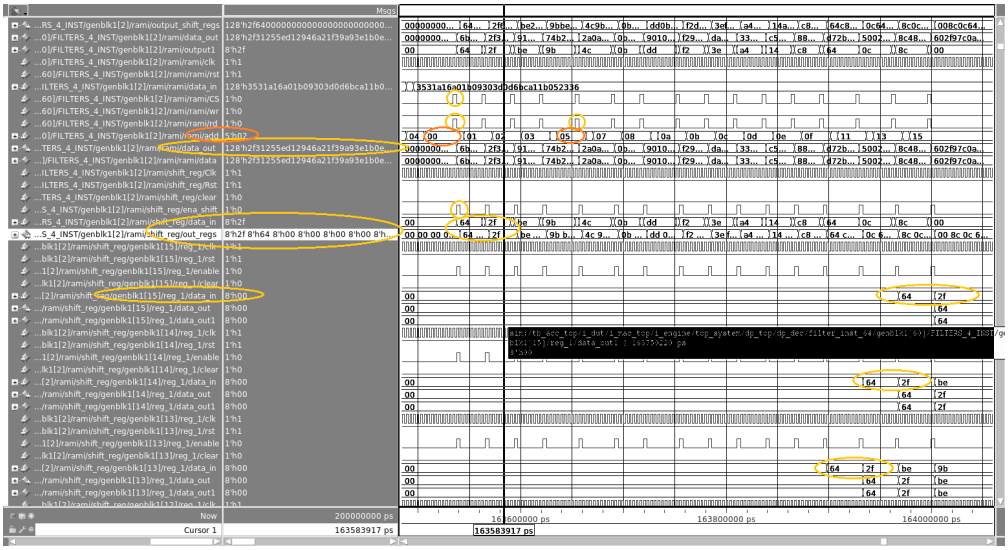

Figure 6.5: Reading of weights and alignment with activation values

COMPUTATION PHASE

After the alignment of 16 or less activations and weights have been done, the algorithm of convolution is accomplished, and its functionality is properly reported in Figure 6.6. The structure of the previous SMAC-Engine for dense matrices has been let unchanged, but the control signal for the evolution of states and the control of the data path are generated in a different way, maintaining the same flow of data and the same algorithm.

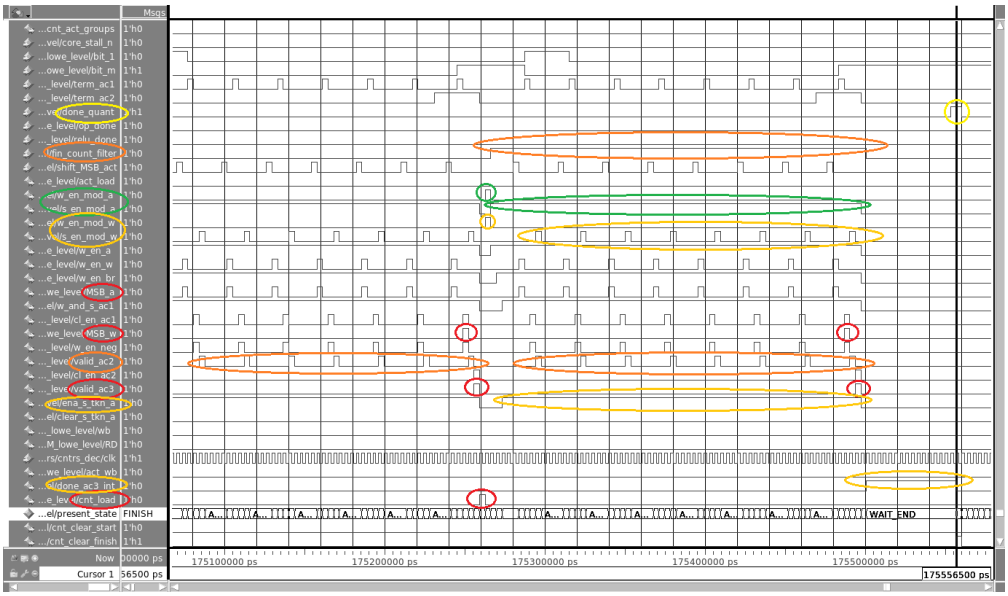

Figure 6.6: AC blocks

COMPRESSION PHASE

In the compression phase, data after the convolution of an entire convolution volume are compressed and, in the meanwhile, sent to the TCDM with 224 bits bandwidth. In Figure 6.4 is reported an example of how this compression operates. In fact, looking at the values signed in yellow, the sequence of the decimal digits "1 0 0 0" has been compressed in the 14 bit word "00000001 000011".

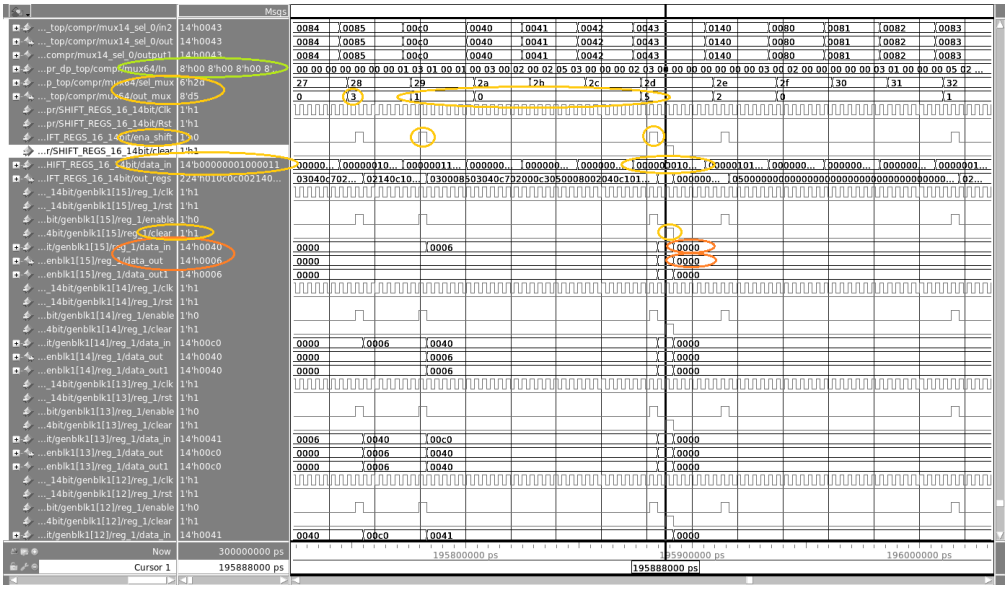

Figure 6.7: Compression

6.1.2 Testbench Setup

In order to obtain this analysis in QuestaSim 10.6c, the following steps have been followed:

1. set the *rtl* folder and load all the description files of the current architecture written in System Verilog;
2. set the *ips* folder and load all the description files of the hwpe-streamer, hwpe-control and hwpe-engine written in System Verilog;
3. set the Makefile in order to generate the libraries related to hwpe-streamer, hwpe-control, hwpe-engine and tech-cells-clock;
4. set the *sim* folder, go there and initialize the QuestaSim environment with the following command:

   source /software/scripts/init_questa10.6c

5. then, issue:

   make clean lib build

in order to execute the *Makefile*, which erases and recreates the libraries for the simulation;

6. finally, launch the simulation:

```
vsim -novopt -t 1ps hwpe_mac_engine_lib.tb_acc_top -L hwpe_ctrl_lib -L
hwpe_stream_lib -L tech_cells_generic_lib
```

6.2 Synopsys Design Compiler logic synthesis

6.2.1 Synthesis setup

In order to perform the logic synthesis with the software Synopsys Design Compiler, some adjustments have to be done. Since the technology used was the umc-65 nm in worst case, in order to clock gating cells to be correctly instantiated, the latch process in *cluster_clock_gating.sv* file has been substituted with the respective cell provided by the library, named LAGCEPM12R and to which the corresponding signals have been connected.
The next step was to change the adopted *script_syn.tcl* scripts onto which the commands given to the software tool are gathered. First of all, due to the IPs consisting in several files whose hierarchical dependence is not always straightforward, to the analyze command the autoread attribute with the corresponding path containing the ips has been added. In addition, the recursive attribute has been added when also the files in the sub-directories of the provided path needed to be analyzed. An example of this kind of command is the following:

```
analyze -f sv -lib WORK autoread -recursive ../ips/hwpe-stream/rtl
```

Another important constraint to add to this script was the one related to the latches with the commands:

```
set_multicycle_path 2 -setup -through [get_pins
    i_mac_top/i_ctrl/i_slave/i_regfile/
i_regfile_latch/hwpe_ctrl_regfile_latch_i/MemContentxDP_reg*/Q]

set_multicycle_path 1 -hold -through [get_pins
    i_mac_top/i_ctrl/i_slave/i_regfile/
i_regfile_latch/hwpe_ctrl_regfile_latch_i/MemContentxDP_reg*/Q]
```

These commands have been employed to specify to the synthesis tool that the employed latches will always work as registers, so they will never be transparent on the same cycle when their value changes.
Finally, to the compile_ultra command, the following attributes have been added:

```
compile_ultra -timing -gate_clock -no_autoungroup
```

to specify the software to use the clock gating cells and to not perform auto ungrouping. For further insights on the employed script it is suggested to check the provided *script_syn.tcl* script file.
As a side note, even though it has not been specifically employed for this last synthesis, the area and frequency values reported in the area comparison in chapter 4 have been gathered after writing a simple bash script, named auto_syn_script, able to iteratively perform the synthesis for a particular configuration, starting from a very relaxed time constraint, 10 ns, and then proceeding downwards in steps of 0.1 ns until the timing closure is violated. The violation triggers the stop of the script execution and provides the final text files onto which the respective area and timing values are reported.
For the logic synthesis of this system the following steps have been followed:

1. create the SYKA65_32X128X1CM2_tt1p2v25c.db for the library definition of the SRAM with 512 kB;

2. create the .synopsys_dc.setup file, which search for the file.db in order to take the synthesis components from the right libraries;

3. set the *syn* folder, go there and initialize the Synopsys environment issuing the following command:

   ```
   source /software/scripts/init_synopsys
   ```

4. issue the command to launch the logic synthesis:

   ```
   dc_shell-xg-t -f script_syn.tcl |tee out.log
   ```

6.2.2 Synthesis Results

The results of timing and area are the ones reported below:

- Area = 4342765,509191 μm^2;
- T_{ck}= 1.6 ns;

The throughput of the system is considered looking at the number of cycles involved in the algorithm for the third layer of ResNet 18 with convolution volumes of 3 x 3 x 128 for 256 filters:

$$\#\frac{MACs}{cycles} = \frac{1}{2} \cdot \frac{3 \cdot 3 \cdot 128 \cdot 256}{16044} = 9.19 MACs/cycle \tag{6.1}$$

$$\#\frac{MACs}{cycles \cdot T_{clk}} = 5.7 GMAC/s \tag{6.2}$$

These values have been obtained considering only reading, computation and compression phases involved in sparsity based convolution operations of a single volume of convolution.
An other fact to higlight is that, considering the fetching phase from the internal memory during convolution instead of the TCDM, the number of accesses to the last one memory is reduced from 1032192 cycles to 10532 cycles, as reported in chapter 3.

Chapter 7

Conclusions and future improvements

In this thesis work the convolution algorithm has been considered looking at the sparsity of data, and the results are different from those of an architecture in which dense matrices are involved. In fact, in the previous SMAC-Engine the result in terms of throughput was around 7.79 GMACs/s, but here the value has been reduced to 5.7 GMACs/s. Moreover, the area obtained of 4.34 mm^2, higher than the one for the system with dense data on 0.29 mm^2, is something expected while considering the algorithm of sparsity handling. However, the main aim in this work was to obtain a further reduction in terms of power consumption and reduction of the number of operations in which negligible values (zeros) would not contribute in the final result of convolution. Moreover, the fact that the fetching operations from memory have been reduced is good both for activations and, relatively, for weights. The choice of instantiating a 512 kB local memory to store weights before the convolution operation was related to the scope of reducing accesses to TCDM in which we exploit just one port with a bandwidth of 224 bits. So, over all the input convolution volumes it would be very high consuming to fetch data from TCDM. While with this structure it is possible a parallel fetching of 256 x 8 bit words. Finally, it is noticeable that the sparsity contribution is considered just during the reading of the internal memory when data are read in parallel over 256 filters and then dispatched to the SMAC-Engine.
The structure considered has been employed in order to test the algorithm of compression and the decoding of the compressed data, which reduce the memory capacity so lowering the power consumption. But in a future work, the idea was to obtain a structure in which the SMAC-Engine data path is doubled and there is the possibility to accomplish convolution over all the layers in a Convolutional Neural Network. The Neural Networks considered were ResNet 18, so dimensions

considered in the example tested are the same of the third layer of this Neural Network, but typically the dimensions of a Neural Network could reach 512 filters, so here comes the need in doubling the structure of SMAC data path, in which the activations shared are the same for all the filter weights stored in SRAM banks, which will be doubled too. Another improvement to do is to make this structure flexible for all the parallelism of weights (4 bits, 8 bits) and activations (4 bits, 6 bits, 8 bits) possible. This could be done yet in this structure, considering weights also on 4 bits; so, instead of fetching just 28 weights each time it could be considered 56 number of weights. But, at the moment, with the *bandwidth* used no more possibilities to make flexible this structure are available and the parallelism used is 8 bits for both data types.

Bibliography

[1] A. Aimar et al. «NullHop: A Flexible Convolutional Neural Network Accelerator Based on Sparse Representations of Feature Maps». In: *IEEE Transactions on Neural Networks and Learning Systems* 30 (2019), pp. 644–656.

[2] Alessandro Aimar et al. «NullHop: A Flexible Convolutional Neural Network Accelerator Based on Sparse Representations of Feature Maps». In: *IEEE Transactions on Neural Networks and Learning Systems* 30.3 (Mar. 2019), pp. 644–656. ISSN: 2162-2388. DOI: 10.1109/tnnls.2018.2852335. URL: http://dx.doi.org/10.1109/TNNLS.2018.2852335.

[3] Alessio Burrello et al. *DORY: Automatic End-to-End Deployment of Real-World DNNs on Low-Cost IoT MCUs*. 2020. arXiv: 2008.07127 [cs.DC].

[4] Y. Chen et al. «Eyeriss: An Energy-Efficient Reconfigurable Accelerator for Deep Convolutional Neural Networks». In: *IEEE Journal of Solid-State Circuits* 52.1 (2017), pp. 127–138. DOI: 10.1109/JSSC.2016.2616357.

[5] Junxi Feng et al. «Reconstruction of porous media from extremely limited information using conditional generative adversarial networks». In: *Physical Review E* 100 (Sept. 2019). DOI: 10.1103/PhysRevE.100.033308.

[6] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. http://www.deeplearningbook.org. MIT Press, 2016.

[7] Song Han, Huizi Mao, and William J. Dally. *Deep Compression: Compressing Deep Neural Networks with Pruning, Trained Quantization and Huffman Coding*. 2016. arXiv: 1510.00149 [cs.CV].

[8] Song Han et al. *EIE: Efficient Inference Engine on Compressed Deep Neural Network*. 2016. arXiv: 1602.01528 [cs.CV].

[9] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. «ImageNet Classification with Deep Convolutional Neural Networks». In: *Proceedings of the 25th International Conference on Neural Information Processing Systems - Volume 1*. NIPS'12. Lake Tahoe, Nevada: Curran Associates Inc., 2012, pp. 1097–1105.

[10] Y. Le and X. Yang. «Tiny ImageNet Visual Recognition Challenge». In: 2015.

[11] Fabio Neri Letizia Lo Presti. *L'Analisi dei Segnali*. Torino, C.L.U.T, 1992.

[12] Angshuman Parashar et al. *SCNN: An Accelerator for Compressed-sparse Convolutional Neural Networks*. 2017. arXiv: 1708.04485 [cs.NE].

[13] Ayeshmantha Perera. «What is Padding in Convolutional Neural Network's(CNN's) padding». In: (Sept. 2018). URL: https://medium.com/@ayeshmanthaperera/what-is-padding-in-cnns-71b21fb0dd7.

[14] Sara Sheehan and Yun Song. «Deep Learning for Population Genetic Inference». In: *PLOS Computational Biology* 12 (Mar. 2016), e1004845. DOI: 10.1371/journal.pcbi.1004845.

[15] Wei Wang et al. «Development of convolutional neural network and its application in image classification: a survey». In: *Optical Engineering* 58.4 (2019), pp. 1–19. DOI: 10.1117/1.OE.58.4.040901. URL: https://doi.org/10.1117/1.OE.58.4.040901.