POLITECNICO DI TORINO

Master's Degree in

Electronic Engineering

Master's Degree Thesis

Optimized VLSI architectures for efficient sparsity exploitation in Deep Learning

Academic supervisors:

Prof. Maurizio Martina

Candidates:

Matteo Pellassa

Michele Tomatis

Academic Year 2020/2021

Abstract

Artificial intelligence (AI) nowadays plays a predominant role in many areas including robotics, computer vision for medicine, autonomous driving and much more. However, this sector's algorithms are very sophisticated and also known to be both compute and memory-intensive. Techniques to improve efficiency, reducing the number of computations without losing accuracy, are becoming critical.

This thesis work focuses on the convolutional layer of Convolutional Neural Networks (CNNs) and aims to improve efficiency by avoiding useless computations. Starting from SqueezeFlow architecture, which employs PT-OS-sparse(planar tiles-output stationary) dataflow to exploit sparsity in the kernel matrices, we develop an architecture able to exploit sparsity in the input matrices and so employs PT-KS-sparse(planar tiles-kernel stationary) dataflow. A two-level memory hierarchy is also introduced to reduce latency and energy consumption in data retrieval.

An algorithm able to avoid useless computation is developed in python and then the hardware capable of executing it is described in VHDL and validated with ModelSim simulations. The proposed accelerator supports convolution with 3x3 kernels, up to 512 input channels and matrices size up to 254x254. The circuit is synthesized with Synopsys, with 65nm technology, obtaining a netlist able to work at 670MHz; the main metrics (area, throughput, power) are retrieved to be compared with state-of-the-art accelerators. Finally, several configurations are proposed, exploiting more accelerators working in parallel to improve the system's throughput.

Acknowledgements

We would like to thank Prof. Maurizio Martina for his infinity availability and his continuous support in this thesis work. Special thanks also go to the PhD students Beatrice Bussolino and Maurizio Capra: they have always given their assistance, without ever sparing their help and time.

Together they guided us through every step of the work and have always been prompt to give their aid, despite the difficulties linked to the pandemic which forced us to awkward videoconference meetings.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Generic introduction

Artificial intelligence (AI) nowadays plays a predominant role in many areas including robotics, computer vision for medicine, autonomous driving and much more. Convolutional Neural Networks (CNNs) have become one of the most used approaches to solve AI problems. In such networks, each layer generates a successively higher-level abstraction of the input data. The basic operation in a convolutional layer is the convolution between input channels (input matrices) and filters (matrices of weights, also called kernels). The output of this convolution is called output feature map (ofmap) and is a matrix that preserves essential yet unique information. The ofmaps, also called output channels, become in turn input for the next layer.

This sector's algorithms are very sophisticated and also known to be both compute and memory-intensive. Techniques to improve efficiency, reducing the number of computations without losing accuracy, are becoming critical; sparsity can be exploited to increase the performance of an accelerator. A sparse matrix is a matrix with zero values in it; in CNNs sparsity can affect both input channels and filters:

- Is demonstrated that is possible to prune weights matrices, removing redundant values, without affecting the accuracy of the net;

- The application of Rectified Linear Unit (ReLU) on the channels turn all the negative values in zeros, turning a dense matrix into a sparse one.

Since the multiply-accumulate (MAC) is the basic operation of the convolution, zeros in matrices (both input and weights) do not contribute to the output. A computation involving null values can be avoided, speeding-up the operation.

This thesis work had its starting point in the paper SqueezeFlow [1], which analyzes the sparsity in neural network models in order to decrease the number of operations required and also the number of accesses in memory. In particular, this paper employs planar tiles-output stationary (PT-OS) sparse dataflow on the case of sparsity in kernel matrices and applies specific rules to avoid doing unnecessary calculations with null values. Instead, in this thesis, we aimed to analyze the case with sparsity in input matrices and so to realize planar tiles-kernel stationary (PT-KS) sparse dataflow. The goal is to develop an architecture able to ignore null values present in input matrices, enhancing performance. Exploiting RLC compression format the zero values in the matrices aren't even stored, sparing space in memory. Furthermore, is introduced a memory hierarchy to reduce latency in data retrieval and related energy consumption.

Workflow

After studying how SqueezeFlow avoids unnecessary computation exploiting filter's sparsity, the first step was to develop an algorithm able to work with dense kernels and exploit sparsity in input matrices. The proposed accelerator supports convolution with 3x3 kernels, up to 512 input channels and matrices size up to 254x254. The algorithm is implemented at the software-level, realizing a python model of an accelerator able to perform nine MACs in parallel.

The next step is the hardware implementation of an architecture able to perform the algorithm. The circuit is divided into two main components: the Processing Element (PE) and the accumulator, which are almost independents and communicates through a FIFO. Both of them have their internal memory area, creating a memory hierarchy. The management of internal parallelism of the data in the accelerator was an object of study, to efficiently manage partial output data.

The accelerator must communicate with the external world to retrieve data. For this purpose is described an external interface, based on FIFOs, to correctly fetch data from external memory. A communication bandwidth of 88 bit was established to interface with it. The VHDL description of the hardware was validated with ModelSim simulations, comparing the result with those calculated by the python model. The external memory, where all the matrices are stores, is simulated using text files.

Once obtained a working model, the VHDL was synthesized with Synopsys, using 65nm libraries, to obtain a Verilog netlist. The netlist of the accelerator is then simulated to obtain its metrics: the accelerator is capable to work at 670MHz, the area of the single cell (PE + accumulator + FIFOs) is 1.63 mm^2. The power consumption obtained simulating the circuit with random matrices, is 21.4 mW . The throughput, that is the number of MAC performed in a second, is about 430 Mega Operations Per Second (MOPS).

Later, to increase throughput, several solutions with multiple PEs in parallel sharing a single accumulator were developed. The cost is the increase in the area and so a trade-off needs to be found. The circuits were validated with realistic matrices from a real CNN, VGG16. and both errors and performance are evaluated. In the thesis conclusion, the circuit's metrics are compared, both with SqueezeFlow and with the state-of-the-art accelerators. Several issues are then explained and possible future improvements are proposed and discussed.

This thesis is group work. Each phase of the work, from the first brainstorming sessions that led to the implementation of the algorithm to the final considerations and the validation of the netlists, was carried out by the two candidates together, in total collaboration. The writing of this document, which is the same for both, was also carried on together.

As regards the practical implementation of the performed work, is possible to find four macro-areas of "tasks":

- Python programming
- VHDL description of circuits and behavioural simulations
- Circuits synthesis and netlist validation
- Simulations and data analysis

The most important and challenging phase, namely the development of the circuits, their description in VHDL and their behavioural validation was implemented by both candidates. The python implementation of the algorithms and scripts, as well as the analysis of the circuit metrics, was mostly carried out by Matteo Pellassa. Everything about synthesis, Synopsys usage, metrics retrieving and, more in general, the work on the VLSI server was mainly done by Michele Tomatis. In the last part of thesis work, where the performance is analyzed using real data, Matteo

Pellassa focused on the simulative part and the throughput analysis while Michele Tomatis dealt with quantization and error retrieving.

1.2 Background

Artificial intelligence (AI) nowadays affects many aspects of our daily life and very often we use it without even noticing it. The term AI was coined by John McCarthy in the 1950s as "science and engineering of creating intelligent machines that have the ability to achieve goals like humans do" [2]. Figure 1.1 shows the subsets that make up AI.

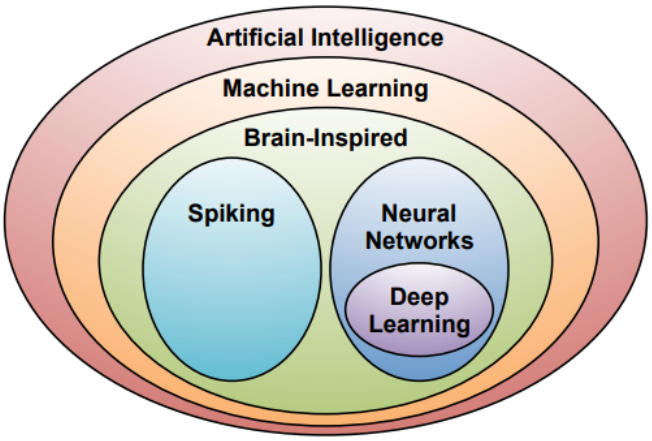
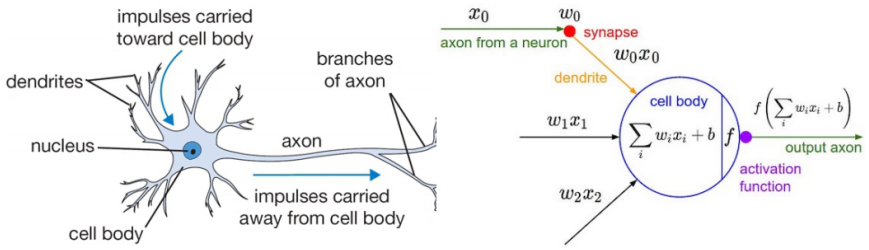


Figure 1.1. AI and relative subsections [2]

In particular, a large subset of AI is machine learning (ML). This term has been issued in 1959 by Arthur Samuel as "as the field of study that gives computers the ability to learn without being explicitly programmed" [2]. This definition allows highlighting the difference between ML and the standard programming method: normally, algorithms are designed and developed to solve specific problems and their efficiency depends on the experience and knowledge of the programmer. Otherwise, the machine learning algorithm (ML) is only realized once by the programmer and then is "trained", through the provision of a large number of incoming data, to learn how to solve a certain set of problems.

Also from Figure 1.1 it can be seen that within the ML it has a part on brain-inspired computation. Since the part of the human body destined for problem-solving is the brain, it is precisely here that the attention of researchers has been focused to try to replicate in an algorithm some aspects of how the human brain is thought to work. The basic element of the brain is the neuron that receives input signals (dendrites), performs calculations on the values received and provides an output signal (axon). These input/output signals are called activations. The output of a neuron (axon) can be connected to the inputs of many other neurons. This connection is made via synapses. The synapse scales the received input signal. This scaling is called weight, and the change in these values is the basis of the mechanism by which the learning of the human brain is believed to take place.

Thus the learning process is related to the regulation of these weights based on the values provided in the input while the overall model of the brain remains unchanged [9]. This concept is very important because by changing the values associated with synapses you can change the behaviour of the system and then the values provided in output without having to create a new model from zero.



(a) Physical structure of a neuron **(b)** Computational model of a neuron

Figure 1.2. Biological neuron and its neural network model. [9]

Computational model in artificial neural network

The mathematical model of the neuron is shown in Figure 1.2, part b. Each dendrite is associated with the product between synapse and axon from another neuron. The various dendrites are collected at the entrance of the neuron and added together; it is also included a bias term to include a possible offset.

Researches in computational neurobiology suggest that neurons are electrically excitable and is generated a pulse in output only if the overall input exceeds a certain threshold. To account for this behaviour in the model, a non-linear function called "activation function" is applied to the output of the neuron. Some of the most popular function are reported in Figure 1.3. The main solution adopted in the literature is the ReLU. The reason is due to the fact that it does not present major critical issues and also is very simple as it only requires a comparison between x and zero.

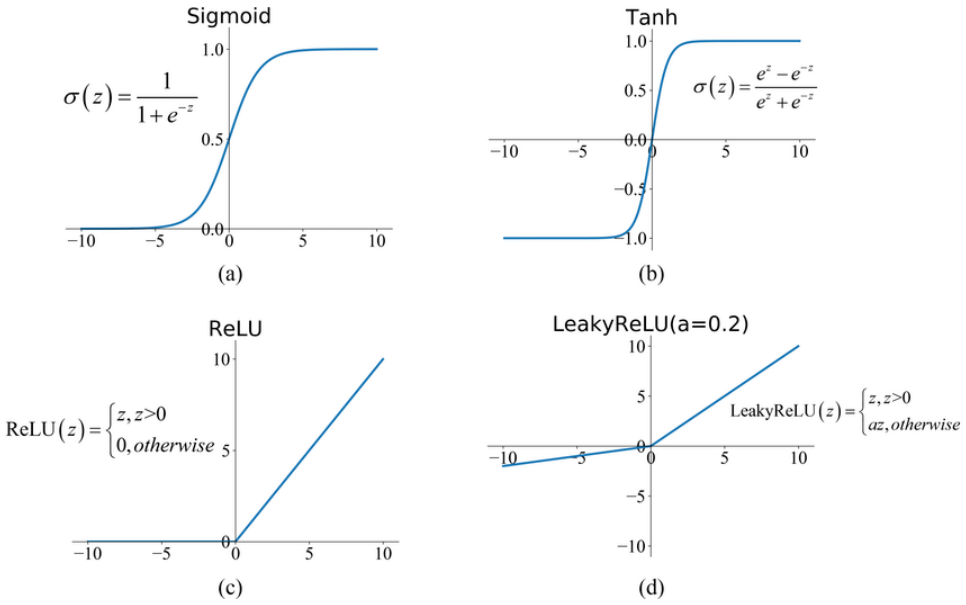


Figure 1.3. Activation functions and their equations [10]

Artificial neural networks are constructed as cyclic graph whose nodes represent the neurons.

These nodes are organized into three different types of layers: the input layer that receiving input, the output layer responsible for process the final result and a variable number of hidden layers. The number of hidden layers determines the depth of a neural network: if there are more than three hidden layers, the neural network is typically called a Deep Neural Network (DNN). Neurons are organized within these 3 types of layers but can connect in different ways. For example, a widely used configuration is fully connected (FC), where each neuron of layer l receives as inputs all the activations of layer l minus 1 and then performs a weighted sum of all its inputs:

$$O[c_o] = \sum_{c_i=0}^{C_i-1} W[c_0, c_i] \cdot I[c_i] + b[c_0] \tag{1.1}$$

where $0 \leq c_o < C_o$ $0 \leq c_i < C_i$ and Ci-Co are the number of neurons of layers l minus 1 and l respectively. Figure Figure 1.4 shows an example of two neural networks consisting of FC layers.

If the number of neurons increased in this case also the number of connections increased and this mainly means higher computation and storage requirements. It is also not always necessary to have each neuron of a layer connected to all neurons of the previous layer. For this reason another type of layer, called convolutional layer, has been introduced in order to limits the number of connections between adjacent layers.

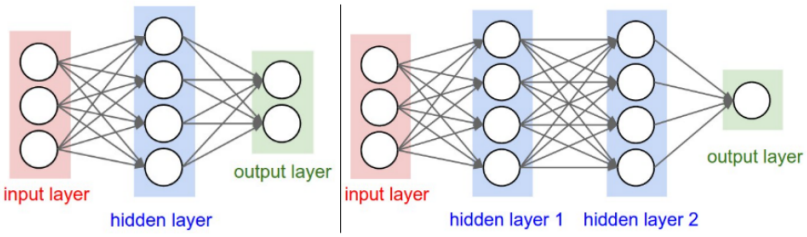


Figure 1.4. FC neural networks [9]

Convolutional Neural Networks

In 1998 a new architecture was proposed, known as Convolutional Neural Network (CNN) [3] to overcome the two main problems related to FC networks: parameter explosion and the fact that the order in which inputs are provided does not affect the results.

This second aspect deserves further explanation. Neural networks are used very often to solve tasks such as object detection and recognition. Images have a strong 2-D local structure: variables (or pixels) that are spatially or temporally nearby are highly correlated. This behaviour of strong correlation between neighbouring elements also allows extraction and combination of local features before recognize spatial or temporal objects by subsequent layers, because configurations of neighbouring variables can be classified into a small number of categories (e.g., edges, corners, etc.).

This concept has also been reported within neural networks where neurons are organized into 2D grids and a neuron of a specific layer does not receive all the activations from the previous one, but it is instead connected to a local and small receptive field of dimension $[H_k X W_k]$. This size of the receptive field is commonly called kernel size and the distance between adjacent kernel is defined by a stride parameter S.

Also concerning kernels, in convolutional networks, there is the concept of shared weights (or weight replication). This idea born from the fact that local receptive fields used to extract

elementary features are usually not only useful on a part of the image to be processed but instead can be applied throughout the image. Applying this concept to neural networks, all the neurons of a specific layer have the same matrix of weights, perform the same operation on a different part of the image and so detecting the same feature in different locations of the previous layer.

The output set of a layer is called a feature map, so several feature maps (with as many different weight matrix) are required to extract different features in a layer. The computations performed in a Conv layer involve an input feature map Ifm of size $[C_i \times H_i \times W_i]$, the weights W of size $[C_i \times C_o \times H_k \times W_k]$, and a bias term b of size $[C_o]$. The result of the computation is an output feature map Ofm of size $[C_o \times H_o \times W_o]$, computed as follows:

$$Ofm[c_o, h_o, w_o] = \sum_{c_i=0}^{C_i-1} \sum_{h_k=0}^{H_k-1} \sum_{w_k=0}^{W_k-1} W[c_i, c_o, h_k, w_k] \cdot Ifm[c_i, Sh_o + h_k, Sw_o + w_k] + b[c_0] \quad (1.2)$$

where $0 \leq c_o < C_o$, $0 \leq h_o < H_o$, $0 \leq w_o < W_o$, $0 \leq h_k < H_k$, $0 \leq w_k < W_k$

In addition to the convolution layer, there are other layers that are often used in CNN: non-linear activations, fully connected and pooling/subsampling layer. The first two have already been analyzed previously while pooling is now explained. The main purpose of this layer is

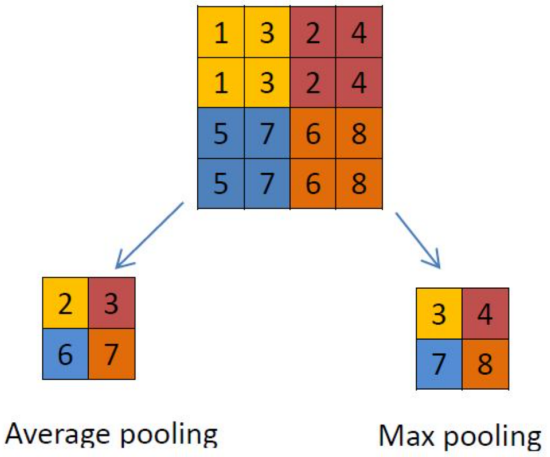


Figure 1.5. Example of MaxPooling and AveragePooling layers [4]

to reduce the size of the feature maps at the output of the convolutional layer and therefore reduce the computational costs and memory requirements for subsequent layers. To achieve this goal, pooling layers have receptive fields, similarly to convolutional layers, and for each of these groups, a single value is returned that takes into account the data statistic. In Figure 1.5 the two main pooling operations are shown: maximum and average. From the point of view of image recognition, the pooling layer is used to reduce the accuracy of feature maps and maintains the most essential information.

Training and Inference

Once the neural network has been built with the relative combination of layers, it is necessary to "train" the neural network to solve a certain class of problems by supplying a large amount of data, each time updating the weights and also any bias. This phase is known as training and

differs from the next phase which is called inference, in which the weights and bias values have already been determined and then by providing further inputs the network should be able to solve the proposed problem, such as the recognition of an image.

There are multiple ways to train neural networks; the main techniques are:

- Supervised learning, with a set of labelled input-output pairs, i.e., a set of inputs (data) with the corresponding expected output (labels). When training the network, for example to recognize images, the correct result is often known because it is given for the images used for training (i.e., the training set of the network). Then during the training phase, the inputs are provided to the neural network and the weights are updated based on the difference between the values provided at the output and the reference values.
- Unsupervised learning in which the training samples are not labelled and the goal is to find common patterns in the data.
- Another possibility is to use semi-supervised learning which is simply somewhere in between the two solutions above.
- Finally, there is the possibility of reinforcement learning in which aim is the creation of autonomous agents able to interact and make decisions in a given environment. This relationship produces cause-and-effect information that is evaluated by an interpreter to understand what the agent should have done to achieve the goal, correct decision. The goal of the agent is to maximize the fed back indication of the interpreter.

A widely used algorithm for NN training, in the case of supervised learning, is gradient back-propagation [5]. Firstly, the weights are initialized, usually randomly, and the inputs are supplied to the circuit for obtaining the results. These results and expected outputs are compared, and a loss (L) is calculated with a loss function, such as Euclidean distance or Mean Squared Error(MSE). The goal is to minimize this loss by applying the gradient method to modify the weights and the bias in the network, following a backward algorithm. The most used methods are Gradient Descent and Stochastic Gradient Descent [6].

1.3 Thesis organization

The chapters of this thesis follow the logical order described in the generic introduction. This is the introductory chapter. The next one is about the algorithm and describes the process which resulted in its final form. It highlights how sparsity can be exploited.

The three following sections describe in detail the hardware implementation of the accelerator. The first one is focused on the circuit which computes the convolution. The second describes how 2D matrices are accumulated to compute one output channel. The former defines the external interface and how the accelerator communicates with the outer world.

The sixth chapter is about synthesis and details the synthesis process. Here the main metrics of the circuit, which will be used later, are described and retrieved. The seventh chapter explore the possible configuration of the circuit. The performance is evaluated and shown with charts. In chapter eight the circuit is validated with data coming from a real neural networks and both precision and performance are analyzed.

The conclusive part of the thesis offers a review of the work. Here is realized a comparison with state-of-the-art accelerators. Moreover, are described the main problems of the architecture, with possible solutions. Possible future improvements are also discussed.

Chapter 2

Algorithm

This chapter describes the operations performed by the accelerator and how the implemented algorithm was obtained. After describing the generic algorithm of 3-D convolution, is analyze how scattering in the input matrices can be exploited to avoid useless computations and speed-up the convolution, highlighting the differences with the "sparse kernel approach" presented in SqueezeFlow [1]. Finally, the operations that the circuit execute to compute the 3-D convolution are described in detail.

2.1 3-D convolution

CNNs, as seen in Figure 1.2, are composed of several convolution layers, each of them performing high-dimensional convolution on a set of input data. Every layer provides the results to the following one, generating a successively higher-level abstraction; modern CNNs can achieve higher performances employing a very deep hierarchy of layers [2]. The focus of this thesis work is a single convolutional layer and thus on the 3-D convolution task.

The activations of the layer are a set of 2-D matrices, called *input channels*. Each channel is "convolved" with a distinct matrix, the *filter* (or *kernel*), generating a new matrix. Each element of the input channel it's added to its local neighbours, weighted (multiplied) by the kernel, to generate an output element: we call this operation 2-D convolution. Each filter belongs to a stack of 2-D filters, with a size equal to the number of channels (a filter for every input channel); this stack of filters is also called a 3-D filter. The results of 2-D convolutions of all the channels are added together to generate a single output matrix, the *output channel*. Finally, an activation function is applied to the output channel; in our case, the chosen function is the simple ReLU seen in Figure 1.3. More 3-D filters can be convolved with the same input channels to create more output channels. The 3-D convolution, graphically shown in Figure 2.1, can be described as:

$$O[m][x_o][y_o] = \sum_{n=0}^{N-1} \sum_{x_k=0}^{K_H-1} \sum_{y_k=0}^{K_W-1} K[m][n][x_k][y_k] \cdot I[n][x_i][y_i] \tag{2.1}$$

Where N is the number of input channels, M is the number of 3-D filters ($0 < m < M$) and K_W and K_H are respectively the width and the height of the filter.

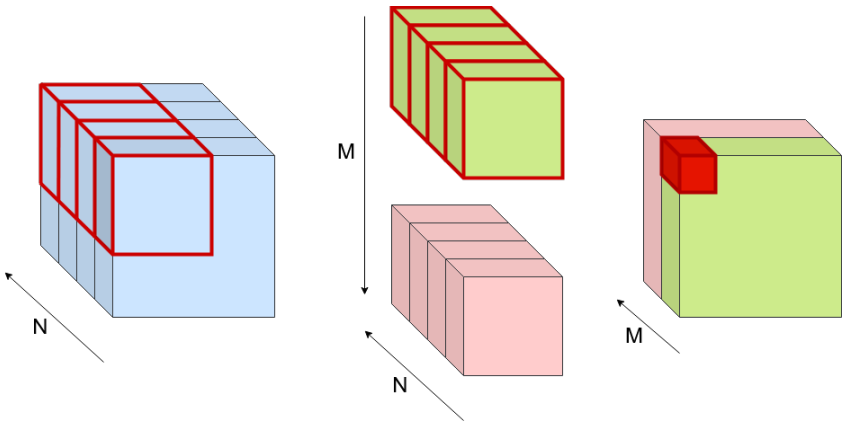


Figure 2.1. 3-D convolution with 4 input channels and 2 output channels. Different 3-D filters generate different output channels

The relationship between output, input and kernel matrices coordinates depends on the specific convolution type. Here is used the same type seen in squeezeflow paper: imagining to overlap the filter over the specific portion of input channel to be analyzed, the bottom right corner of the selected part has the same coordinates of the output element to be computed. The computation of 2-D convolution is defined as:

$$O[x][y] = \sum_{x_k=0}^{K_H-1} \sum_{y_k=0}^{K_W-1} K[x_k][y_k] \cdot I[x + x_k - (K_W - 1)][y + y_k - (K_H - 1)] \tag{2.2}$$

To better understand how our convolution works a graphical example is shown in Figure 2.2. In this example the filter is a 3x3 matrix and this will be the kernel size for now on.

This algorithm is a "full" convolution algorithm: the output location is computed also when the filter overlap outside the input matrix. In that case, the weights that are outside are multiplied by 0. With this method, the output matrices are bigger than the input one. This difference will be visible when dealing whit a real CNN, in chapter 8. Anyway, despite the algorithm is different, the basic operation is the same: the results is a matrix that is perfectly equal aside from the external values, which can be easily ignored.

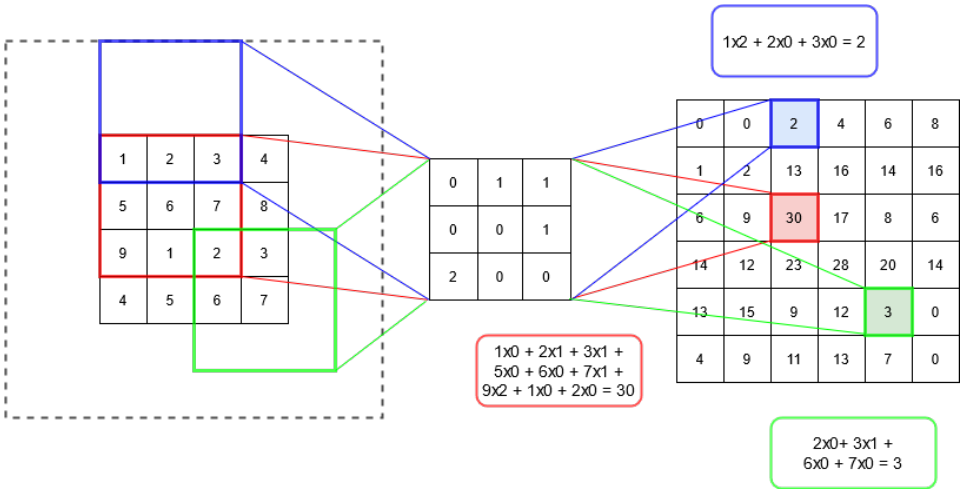


Figure 2.2. Example of 2-D convolution

2.2 Impact of sparsity

Sparsity, in CNNs, can be exploited to reduce the number of computation, memory accesses and storage requirements, increasing the performance of an accelerator. A sparse matrix is a matrix with zero values in it; in CNNs sparsity can affect both activations and weights:

- Is demonstrated (for example Cambricon-X [8]) that is possible to prune weights matrices, removing redundant values, without affecting the accuracy of the net;
- The application of ReLU on the activations turn all the negative values in zeros, turning a dense matrix into a sparse one.

These two sources of sparsity can lead to matrices with up to 70/80% of null values.

In section 2.1 is qualitatively described the algorithm but, to understand the impact of sparsity (and also to fully understand how it is implemented), the operations to be done must be better defined. First of all the 3-D convolution can logically be separated into two phases: 2-D convolution and accumulation. While accumulation is a very straightforward operation, here the focus is on the computation of 2-D convolutions.

In Figure 2.5 is shown the pseudo-code of the 2-D convolution. Each output location is computed iterating through the kernel matrix, extracting at each cycle the required input value and multiply it for the corresponding weight. So, for every output locations:

- 9 input values are read from memory
- 9 MACs are performed
- the result is stored in memory

The basic operation of the convolution is the multiply and accumulation (MAC). The number of computation to complete a 2-D convolution is proportional to the size of the output matrix:

$$N_{MAC} = 9 \cdot O_W \cdot O_H \tag{2.3}$$

where O_W and O_H are respectively the width and the height of the output while the 9 is due to the input portion that is involved for each output location (kernel dimensions K_W=3 and K_H=3). However, when one of the values to multiply is zero the operation is useless and can be avoided. These MACs can be skipped to save time and energy.

Squeezeflow paper proposes a simple modification to exploit sparsity in the filters. The new algorithm iterates only through not-zero values in the kernel matrices, reducing the computation needed proportionally to the sparsity factor:

$$O[x][y] = \sum_{i=1}^{N_k} K_i \cdot I[x + x_k - (K_W - 1)][y + y_k - (K_H - 1)] \tag{2.4}$$

$$N_{MAC} = N_k \cdot O_W \cdot O_H \tag{2.5}$$

where N_k is the number of not-zero kernel values.

This method doesn't involve intrusive changes to the basic logic. The only difference lie in the coordinates computation: since the convolution is no more regular they can not longer be derived from the loop indices. Activation coordinates need to be computed starting from those of the not-zero kernel locations.

This approach is perfect when dealing with weight sparsity but is not suitable to exploit sparsity in input matrices. In fact, when an output location is computed, is impossible to know in advance if the input needed for the computation is equal to zero. Is possible to skip the operation once the input value is read from memory, but in this way the performance gain is zero. Indeed, the input matrix is like a "passive actor" in the convolution, without the possibility to control the access to its locations. To exploit the input sparsity, i.e. to avoid its zero locations, a perspective change is needed. If each output element needs nine input values to be computed can also be said, looking at the problem from another point of view, that each input contributes to the computation of nine outputs locations. The way the input location affects the output matrix is shown in Figure 2.3: the filter superimposes every input location in nine different ways, computing every time a different output location. The colour of the 3x3 square which surrounds the input location (in red in the matrix) corresponds to the weight used in the computation. The circles, that are always in the bottom-right of the square, highlight the coordinates of the output. In Figure 2.4 there is another example, this time with numbers.

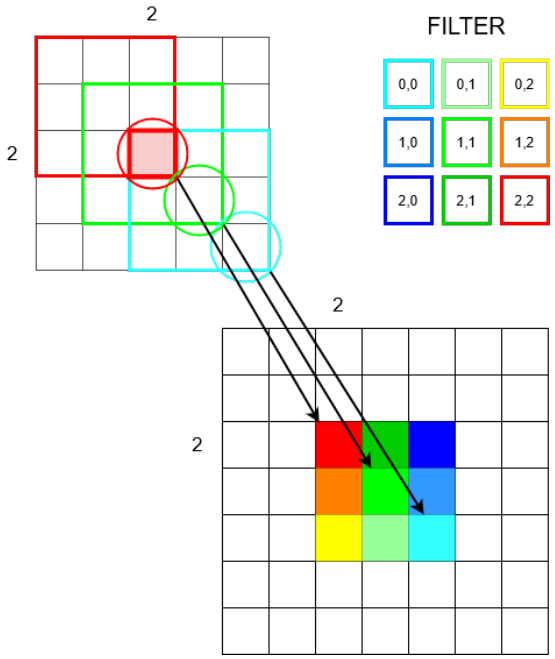


Figure 2.3. Depending on the weight the affected output location is different

The proposed algorithm iterate through the non-zero input locations(N_I) computing, for each one, the nine partial contributes, updating the output matrix. With this method, the sparsity in the activations matrices is perfectly exploited (zero values are not even stored in memory as we will see in the next pages), but a major problem arises: the number of memory accesses increases a lot. Unlike the previous approach, which provide a complete output location at every cycle (ready to be stored and then accumulated for 3D), the result of the MACs, in this case, are partial sums, which may need to be updated up to nine times. For this reason, a smaller memory is introduced inside the accelerator, where the partial results are stored until completion. This memory hierarchy allows mitigating the problem, reducing the latency and energy consumption caused by more frequent memory accesses. Details about this memory can be found in the chapter dedicated to the hardware, chapter 3.

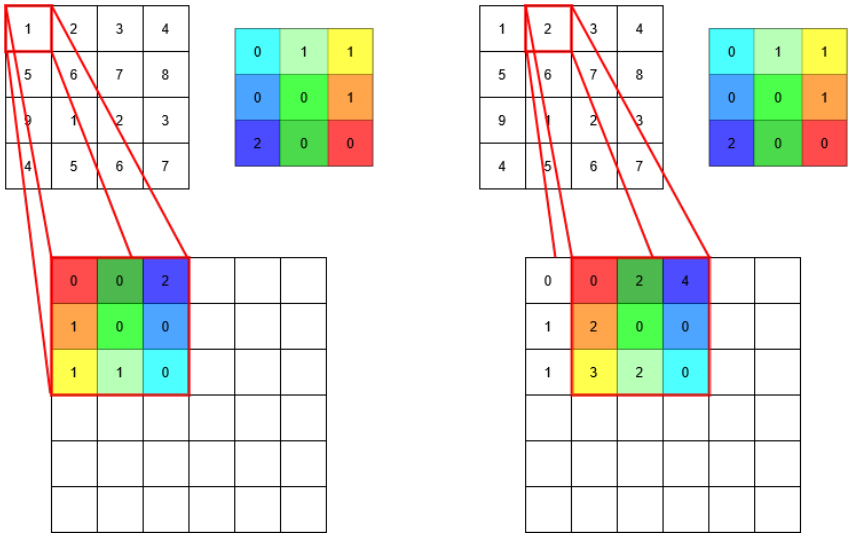


Figure 2.4. Every input affect nine output locations

```
for(m=0;m<out_channels;m++)
    for(n=0;n<in_channels;n++)
        for(oh=0;oh<Oh;oh++)
            for(ow=0;ow<Ow;ow++)
                for(kh=0;kh<Kh;kh++)
                    for(kw=0;kw<Kw;kw++)
                        output[m][oh][ow]+=input[n][oh+kh-(Kh-1)][ow+kw-(Kw-1)]*kernel[m][n][kh][kw]
```

conv_3D — conv_2D

Figure 2.5. Pseudocode of a Conv layer

2.3 Input compression format

Exploiting the sparsity of input matrices implies that the zero values are just skipped and there is no reason to store them. By using compression techniques it's possible to store only the non-zero elements and their coordinates (coordinates are needed because the structure is no more regular), reducing storage requirements and saving space. In literature there are plenty of compression techniques suitable for the purpose but, following the squeezeflow paper, Run Length Compression (RLC) is used in this work. RLC compress the matrices storing only the non-zero values and the number of zeros between them (implicit coordinate). An example of RLC applied on a simple matrix is shown in Figure 2.6. The original coordinates can be retrieved as follows:

$$x_i = x_{i-1} + (y_{i-1} + coord + 1)/L \tag{2.6}$$

$$y_i = (y_{i-1} + coord + 1)\%L \tag{2.7}$$

Where "coord" is the implicit coordinate and L is the length of the input matrix row. This technique involves an overhead caused by the coordinate information but, obviously depending on the degree of sparsity, the storage requirement can be reduced up to 60%.

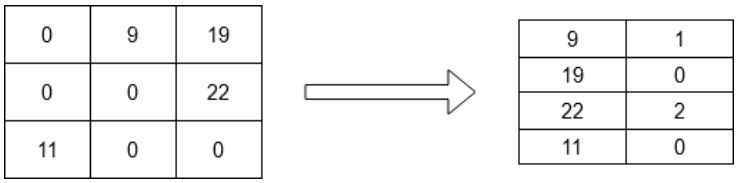


Figure 2.6. Example of RLC compression

Coordinates recovering is the first thing do be done at every algorithmic cycle because coordinates are needed to know which locations of the output matrix will be updated. Due to the complexity added by the compression, the circuit needs ad-hoc hardware to compute the coordinates of the activations (chapter 3).

2.4 Operations parallelism

The simplest way to speed-up the 2-D convolution is to perform more than one computation in parallel. When talking about the generic algorithm there isn't a real limit to the number of output locations that can be computed at the same time, being every calculation independent. Looking to this algorithm, instead, the operation parallelism must be carefully analyzed.

As it's now clear, every input sample contributes to nine different output locations, and the position of these contributes depend on the input and kernel coordinates. Every time a new contribute is computed the output location must be updated. Problems can occur, when exploiting parallelism, if data are not consistent, that is if two contributes to the same output locations are computed at the same time. Errors occur in this case because all the computations are done using the same old values.

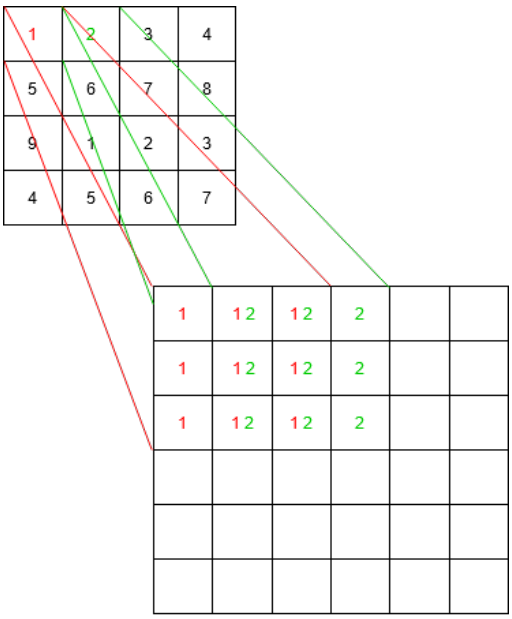


Figure 2.7. Two contiguous input locations need to update the same output portion

It's obvious that computations can be done in parallel only if involve independent (different) output locations. Not knowing in advance the coordinates of the next input locations, the only way to be sure to always perform independent operations is to compute in parallel only the contributes deriving by the same input location. The parallelism is then limited to nine computations at a time, each one involving the same input location and a different kernel value.

We will take full advantage of the parallelism limit by using nine different MAC units, each one multiplying the input value with a different weight from the 3x3 filter.

2.5 Data recycling

The typical convolution algorithms exploit the regular nature of the dense 2-D convolution to maximize data reuse, sharing input values between adjacent computation units. Also the Squeezeflow's "sparse kernel approach" take advantage of data sharing, though to a lesser extent. This is possible because adjacent output locations share some of the input values needed for their computation.

Something similar is also possible in this thesis work because, in some cases, different activations affects the same output locations; this happens when the activations are close in the matrix (Figure 2.7). The same reason we have seen previously in limiting maximum parallelism can now be an opportunity to save memory accesses. Consider taking full advantage of the parallelism, with nine output contributes computed at each algorithmic cycle: to correctly update the output matrix nine load and nine store operations are needed. Suppose now that some contributes computed at cycle $i+1$ are related to the same output locations of those computed at cycle i. Some computation of the cycle $i+1$ need the results of the cycle i: in this case, it would be convenient to directly reuse the data without storing and then reloading them.

Let's imagine the nine output locations to be computed organized in a square fashion, divided in three columns. There are 12 possible cases where computations can be "recycled" when moving from cycle i (red square) to cycle $i+1$ (green square), as shown in figure Figure 2.8. Indeed,

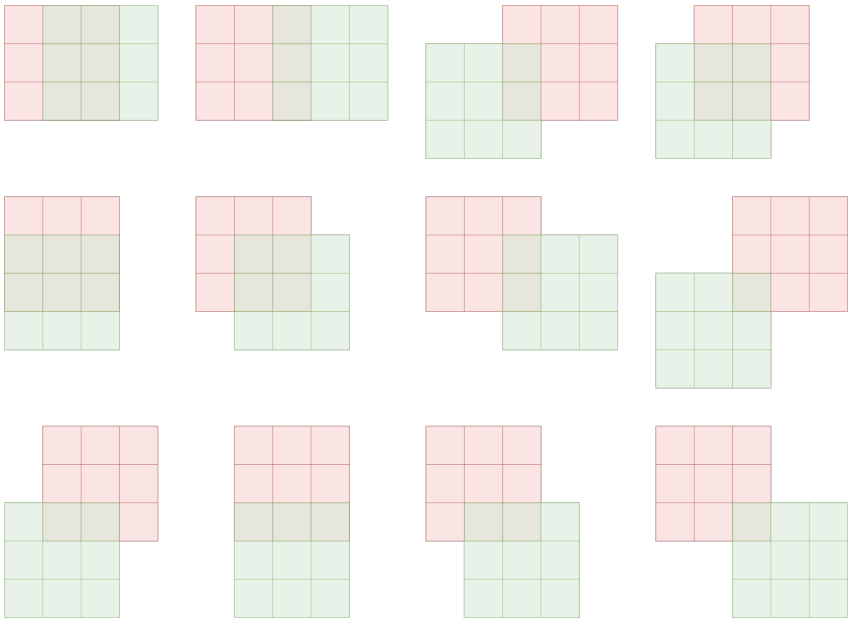


Figure 2.8. All possible recycle patterns

analyzing the recycle with realistic input matrix, i.e. quite big ones, only the configurations where the input location of cycle $i+1$ is on the same line of the one of cycle i occur. This reduce the recycle to two configurations, where whole columns are recycled, simplifying a lot the recycle management (Figure 2.9).

The implicit coordinate, together with the row of the current and next input locations, can be used to discriminate if it's possible to recycle and what to do. In particular, at every cycle, three possible cases can occur:

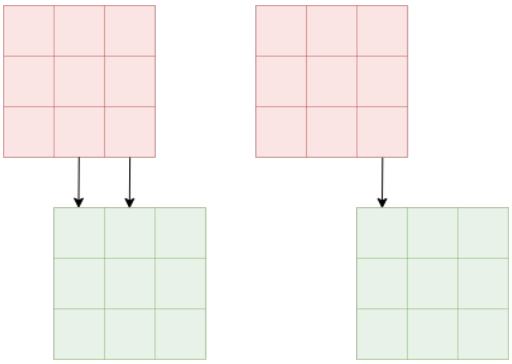


Figure 2.9. The two actual recycle cases

- implicit coordinate = 0; $x_{i+1} = x_i$: **maximum recycle**; the 2nd and 3d column of the computed output contributes are not saved and are used to compute the 1st and 2nd column of the next cycle, that are not loaded from memory
- implicit coordinate = 1; $x_{i+1} = x_i$: **partial recycle**; the 3d column of the computed output contributes is recycled to compute the 1st column of the next cycle
- all the other cases: **no recycle** is possible; all results have to be stored and at the next cycle is necessary to load from memory 9 partial output

Exploiting the recycle is possible to reduce the number of accesses to the partial results memory up to 30%, saving a lot of time. Of course, the lower the matrix sparsity the higher is the recycle impact on the performances: more dense matrices have more recycling opportunities.

2.6 Detailed algorithm and python model

The full algorithm, with all the previous considerations, can be logically described as:

- load a new not-zero activation
- extract the coordinates
- recycle previous results when possible
- load partial sums from internal memory
- compute the nine contributions
- update the output values, storing the results which will not be recycled

To decide which output elements to load from memory and which one to recycle from the previous computation is needed a comparison with the coordinates of the previous cycle. In the same way, it would be necessary to know the coordinates of the next activation to discriminate which partial result to store and which one can be immediately updated by the circuit. We prefer to reorganize the operations so that only the old coordinates are needed. The new order is:

- load a new not-zero activation
- extract the coordinates
- store the results computed in the previous cycle
- recycle previous results when possible
- load partial sums from internal memory
- compute the nine contributions

In Figure 2.10 is shown an example of a 2-D convolution as is computed with this algorithm, showing recycle. The filter consists of only values equal to one.

The algorithm is mapped in python to obtain a reference model to validate the circuit's results. Other than the 2-D convolution, are also implemented a function to compress activations and output matrices with RLC and a function to accumulate the convolution results. The matrices are represented with python's lists and are also printed on files to be read from ModelSim when the circuit is simulated. The files are divided in input files, filter files and output files and every file store a matrix, one value per row.

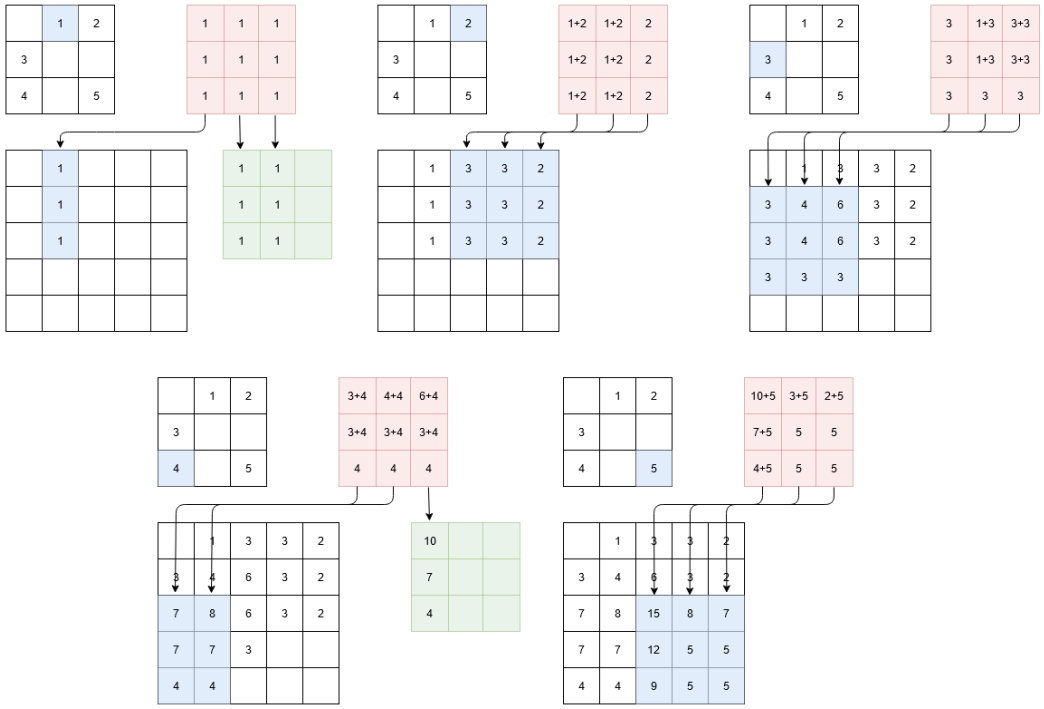


Figure 2.10. Convolution computed

2.7 Accumulation

The algorithm is now fully characterized regarding 2-D convolution, but not regarding the accumulation of the results across the channels. The results of the convolution between a single input channel and a filter are, in fact, partial contributes to the computation of an output channel. These contributes have to be summed together with the results of other 2-D convolutions to obtain the final value of the 3-D output channel. A new layer of memory is introduced to the hierarchy where to store, channel after channel, the partial results until the output channel is fully computed. The complete circuit will have a 3-level memory hierarchy:

- **main memory**: here are stored the activations, the filters and the output channels once fully computed;
- **intermediate memory**: here the output values are stored while they are accumulated;
- **internal memory**: this memory contains the partial results of the 2-D convolutions.

Considering that the output channels of a layer are used as input for another one is good to store the results in a compressed format. This operation synergizes very well with the application of ReLU to the output and is described in detail in section 4.1.

Chapter 3

Processing Element - 2-D convolution

This chapter describes the architecture of the circuit which perform the 2-D convolution, the Processing Element (PE). Firstly are analyzed all the convolutions parameters and how they affect the design. Then all the units which compose the PE are detailed, with their role in the convolution, their parallelism and how they are controlled by the FSM. Moreover, we fully characterize the internal memory used to store the partial values and we describe how we handle the generation of the results once they are fully computed.

3.1 Convolution parameters

There are plenty of variables regarding the convolutions and most of them affect the characteristic of the circuits. Here are described the choices made and how such parameters determine some of the circuit features.

The filter dimensions has already been chosen, opting for 3x3 matrices. This choice strongly affects the algorithm and the number of MAC that can be carried out in parallel (section 2.4). Now we have to choose the size of the input matrices; this parameter affects the dimensions of both the memories in our hierarchy. GoogLeNet [11] and VGG16 [12], two of the most important CNNs, elaborate images with input size 224x224. AlexNet [13], another important CNN, elaborates 227x227 images. Setting the maximum dimension of the input channel to 254x254,in order to have output 256x256 (following powers of two),the circuit should be able to support every CNN. The only limit lies in the maximum size of the input channels, convolutions with smaller matrices are always possible: the memories inside the circuit are simply underutilized. Considering that the biggest matrices elaborated by a CNN are the input of the first layer, if the circuit can process the first layer it will be able to process others as well.

The last parameter that affects the circuit's data parallelism is the maximum number of input channels that can be processed for a single convolution. This parameter affects the dimension of the accumulation memory, as described in section 4.4. To select a proper value the most common neural networks are once again inspected. VGG16, the CNN that will be used to validate this accelerator, present a maximum of 512 input channels in its layer; this is the chosen value for our accelerator.

In addition to those maximum parameters, that are fixed and set by construction, there are some others, customizable at run-time, which define the convolution to be executed. They are:

- input matrix dimension: needed to obtain the coordinates from the compressed input
- number of input channels: needed to control the accumulation of the results
- truncation mode: select the type of truncation applied to the output values (see section 4.4)

These parameters are set before the start of the convolution and are stored in proper registers in the configuration phase.

3.2 Architecture

The circuit is composed of two macro-areas: the control, including FSM and various utility circuits and the computation unit, composed by the MAC circuits and the internal memory. All the units are described below.

Coordinate Computation Unit

Coordinate Computation Unit (CCU) retrieves the coordinates of the activation being processed. It works in a recursive way, using the implicit coordinate field and the matrix's row length, stored in the configuration register. The formulas to obtain the coordinates from the RLC compressed input are:

$$y_i = y_{i-1} + coordinate + 1 \tag{3.1}$$

$$x_i = x_{i-1} \tag{3.2}$$

until $y_i <$ length, meaning that the values are on the same row. Otherwise the next value is on the next row of the uncompressed matrix and:

$$y_i = y_{i-1} + coordinate + 1 - length \tag{3.3}$$

$$x_i = x_{i-1} + 1 \tag{3.4}$$

To work properly the y_{i-1} register need to be initialized to the value "-1".

The CCU sub-circuit is composed of several adders and a subtractor, as shown in Figure 3.1. First of all the implicit coordinate is incremented by one and is added to the old value of the y coordinate. The sign of the difference between this value and the length of the row is used to discriminate if we reached a new row. If it is negative the new value is still in the same row. Instead, if it's positive a new row is reached and the row coordinate needs to be increased. Row and column signals (x and y) are represented with 8 bit because the biggest supported input channel is a 254x254 matrix. The output of adders and subtractor are instead 9 bit long. In Figure 3.2 is shown an example of the CCU working flow.

The implicit coordinate of the RLC compression is on 6 bit. Its parallelism is obtained in a statistic way: several random matrices were generated, with dimension 254x254 and different degrees of sparsity (between 30% and 70%). The maximum number of consecutive zeros in matrices is always less than 64 and so 6 bit are enough for its representation. Moving forward with thesis work we realized that this probably is not the optimal value and can lead to some problems in real cases when the sparsity is not uniform within the matrix (chapter 8). A greater number of bit, like 8, would be a better choice, although it requires more memory to be stored. Modify the CCU circuit would not be problematic but the RLC parallelism also affects the external interface and bandwidth (section 5.1); for this reason, the initial number of bit has not been changed. More detail about this problem, the implications and how to increase the coordinate parallelism can be found in the conclusion of this work.

Recycle unit

The recycle unit is a simple circuit that has to discriminate the situations in which is possible to reuse some values from the last computation. The control unit, based on the output collected here, selects which values read and store from/to internal memory.

The output signal is "00" when there is nothing to recycle, "01" in a partial recycle situation and "10" in when the recycle is maximum; the possible situations have been described in section 2.5. The circuit consists of comparators and simple logic behaviorally described in VHDL.

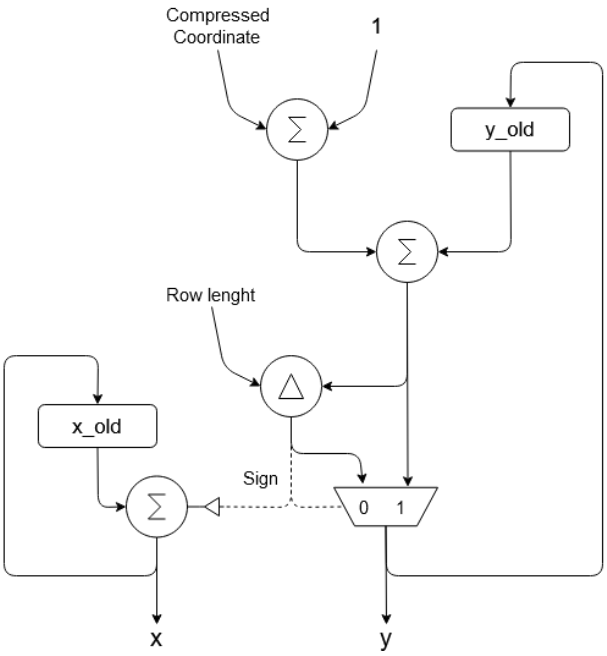
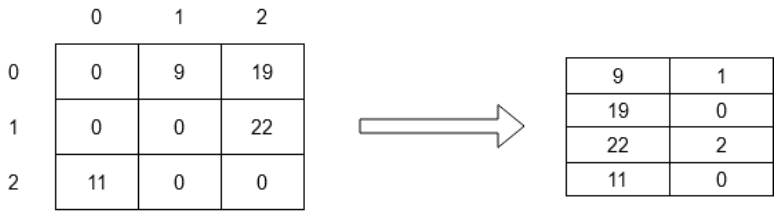


Figure 3.1. CCU scheme



row lenght	x_old	y_old	implicit coordinate	sum	diff	y	x
3	0	-1	1 (2)	1	-2	1	0
3	0	1	0 (1)	2	-1	2	0
3	0	2	2 (3)	5	2	2	1
3	1	2	0 (1)	3	0	0	2

Figure 3.2. CCU working flow

MAC units

The main operation of the convolution is the MAC and so the heart of the PE is the unit that performs this operation. The processing unit is composed of nine units that compute MACs, connected to the internal memory. Every unit contains a multiplier, which computes $I \cdot K_i$, and an adder, which accumulates the multiplier output to the partial sum from previous cycles. The MAC unit is shown in Figure 3.3.

Each unit is connected to one specific weight (K in the figure) from the 3x3 filter during all the

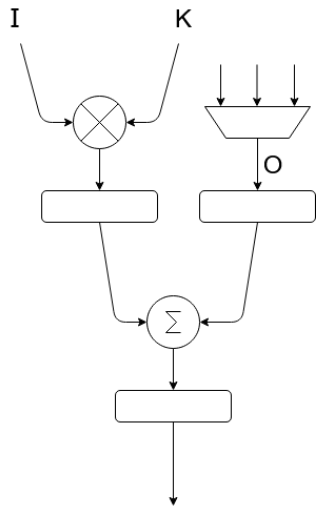


Figure 3.3. MAC unit

convolution process and receive a non-null input value at each cycle (I). The other value needed for the computation is the partial sum (O) coming from the memory and, in the case of recycle, some units receive as input the result of another one, to avoid useless memory accesses. The proper partial sum (recycled or coming from memory) is selected by a multiplexer, controlled by the CU. The whole computation circuit, with the connections between units, is in Figure 3.4.

The coordinates of the output affected by a specific unit depend only on the weight that is linked to it. Is possible to imagine the computation units as organized in a square fashion, with the unit in the top-left corner which computes the output whit the same coordinates as the input value. The other units compute the remainder of the square, every one with its offset. It's worth noting that, as seen in section 2.2, the "output square" computed by the circuit corresponds to the kernel matrix not exactly but symmetrically (respect the centre of the kernel). The top-left output location is computed starting from the bottom-right weight, the bottom-right one is computed with top-left weight, and so on. In Figure 3.5 there is the graphical example which clarify the concept and in Figure 3.6 a summary table. For every location, whose coordinates are on the edge of the table, is reported which weight from the kernel matrix computes that value.

Address selector and results multiplexer

Since the reading/writing is processed one value at a time, some logic is needed to always provide the correct address to the memory. Addresses are managed employing a multiplexer, the controller of which is provided by the CU.

In our case, with nine values managed in every cycle, we need a 9-way multiplexer and nine different addresses. The nine addresses, divided in column and row, are the combination of three columns values (y, y+1, y+2) and three rows values (x, x+1, x+2). Each one is connected to one entry of the mux (Figure 3.7).

The selection of the result of which MAC to save each time is also selected with a multiplexer, controlled by the same signal, to guarantee the correlation between the value and his address.

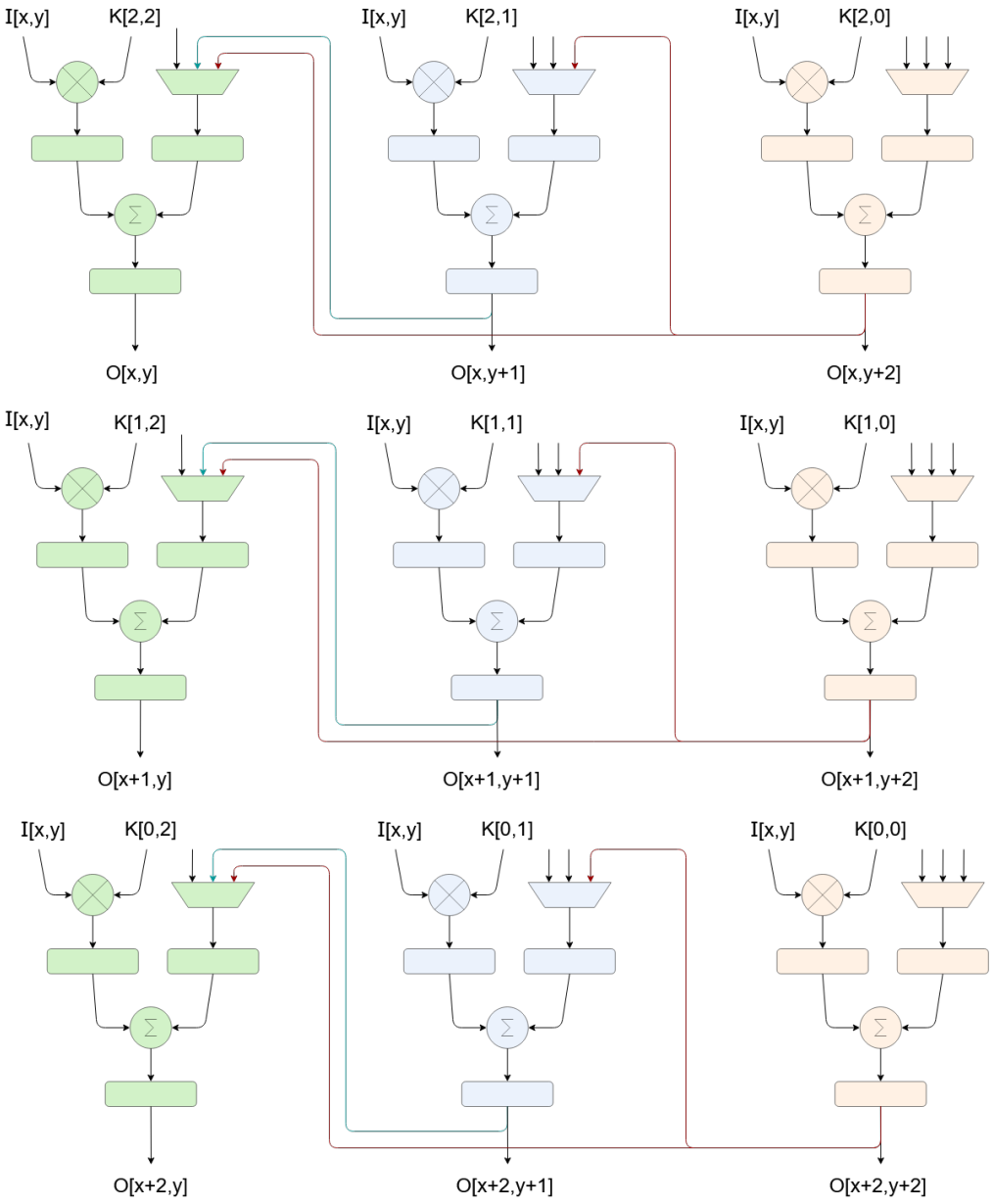


Figure 3.4. Full computation unit

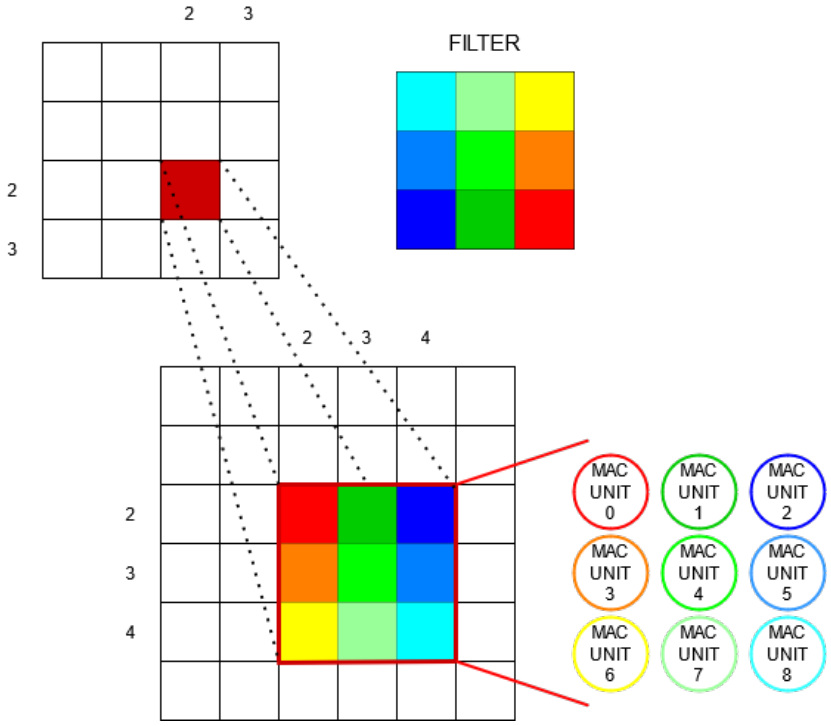


Figure 3.5. Weight - output correspondence

Y + \ X +	0	1	2
0	K[2,2]	K[2,1]	K[2,0]
1	K[1,2]	K[1,1]	K[1,0]
2	K[0,2]	K[0,1]	K[0,0]

Figure 3.6. Table of the output computation

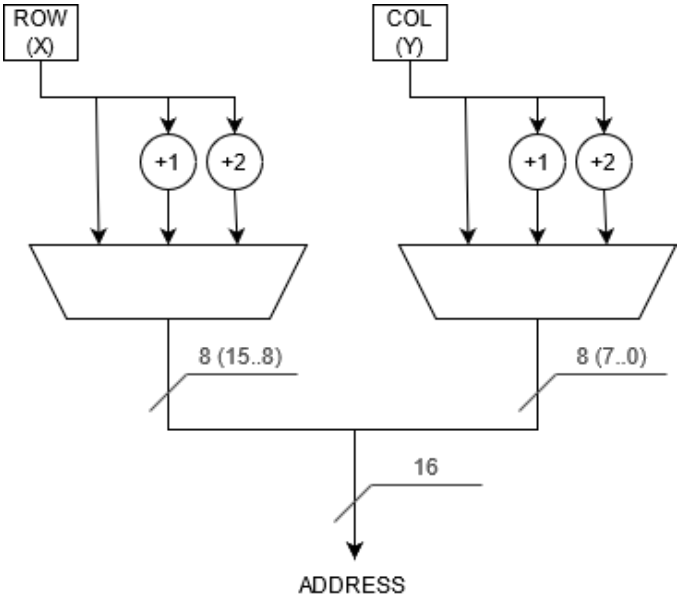


Figure 3.7. Coordinates selector

3.3 Parallelism

As mentioned above, the core of the PE is the MAC operation. The multiplication is done having inputs and weights on 16 bits; the correct parallelism of the multiplication result should be 32 bits. Actually, the values are on 16 bit but are signed numbers, so the MSB only provide informations about the sign of the number. Multiplying two signed numbers on 16 bit 31 bit are enough to represent the result. A simple demonstration can be found in Equation 3.5.

$$2^{n-1} \cdot 2^{n-1} = 2^{2n-2} \tag{3.5}$$

To complete the MAC operation it's necessary to carry out the addition between the multiplier output and the partial result of the output matrix. This partial output is stored in a support memory inside the PE so that it can be easily recovered. Therefore the parallelism of this memory depends on the parallelism adopted inside the MAC.

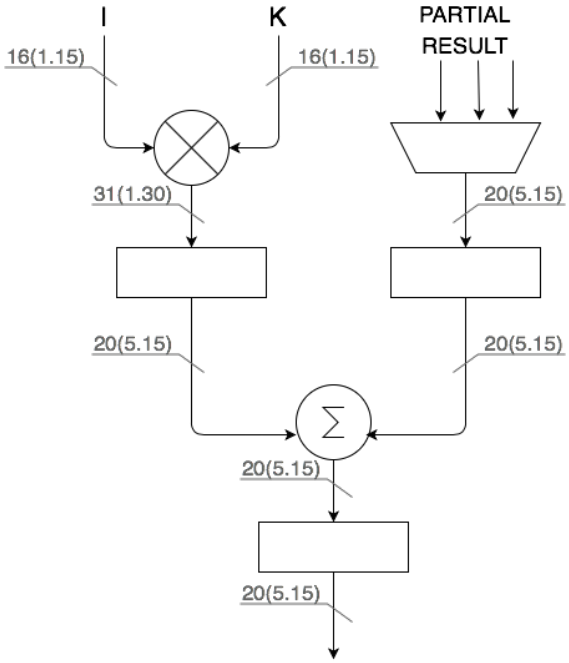


Figure 3.8. MAC unit parallelism

As shown in Figure 3.8, the choice made was to truncate the 15 LSB at the output of the multiplier to bring the parallelism back to input one, ie 16 bits. In this way the precision decrease but the dimension of the internal memory is greatly reduced. On the other hand, it was decided not to apply any truncation at the exit of the adder, to preserve resolution. The 16-bit parallelism will be restored only when the result is complete and ready to be accumulated to the output channel. It should also be remembered that each output location can be processed within the filter's range 9 times with a 3x3 kernel. This means that at most each output location will be taken from memory nine times to perform the MAC and therefore is possible to have at most nine additions for each output location. Knowing the worst case on the maximum number of sums and knowing the parallelism chosen at the adder input (16 bits) it's easy to obtain the parallelism needed to

avoid overflow. This clearly is also the parallelism of the internal memory.

$$16 + \lceil log_2(9) \rceil = 20bit. \tag{3.6}$$

An important observation still concerns the adder. This component performs a sum between data on 16 bit (multiplication output) and data on 20 bit (partial output from memory). To align the operands at the input of the adder it is necessary to carry out a 4-bit sign extension of the multiplier output. The data at the adder output is kept on the maximum parallelism, ie 20 bits, and saved in the support memory. A final truncation of 4 bit is performed to the results when they become "complete", i.e. they will no longer be involved in the computation.

A final important consideration concerns the format chosen to represent the data within the MAC. The input and weight values are represented by fixed point notation on N=16 bit:

$$(-1)^s \cdot m \cdot 2^{-f}. \tag{3.7}$$

where s is the sign bit, m is the N-1 bit mantissa, and f determines the location of the decimal point and acts as a scale factor. With f = 0, the weights and activations values have a range from -32768 to 32767. In our case, we have f = 15 so the range becomes from -1 to 0.999969482. This format is represented as 1.15, where the dot indicates the separation between the number of integer bits, (in this case only the sign), and the number of fractional bits (f = 15). As already said, the multiplication between two numbers with this format is represented on 31 bit, resulting in an output on 1.30 format.

This number of bits should then be further increased by 4 bits to correctly manage the various sums (max nine) for each output location; the maximum number of bit within the MAC rises to 35 bit, in the format 5.30. This has a strong impact on the MAC area but also on the power consumption. It was decided to reduce this number of bit truncating the low 15 bit at the output of the multiplier, thus passing from 1.30 to 1.15 format at the multiplier output and 5.15 out from the adder and inside the memory. The truncation of the 4 LSB to the "complete" results lead to an output format of 5.11. Knowing this is important to retrieve the correct decimal values from the bit representation.

The Figure 3.9 shows a graphical summary of the representation of the data within the MAC.

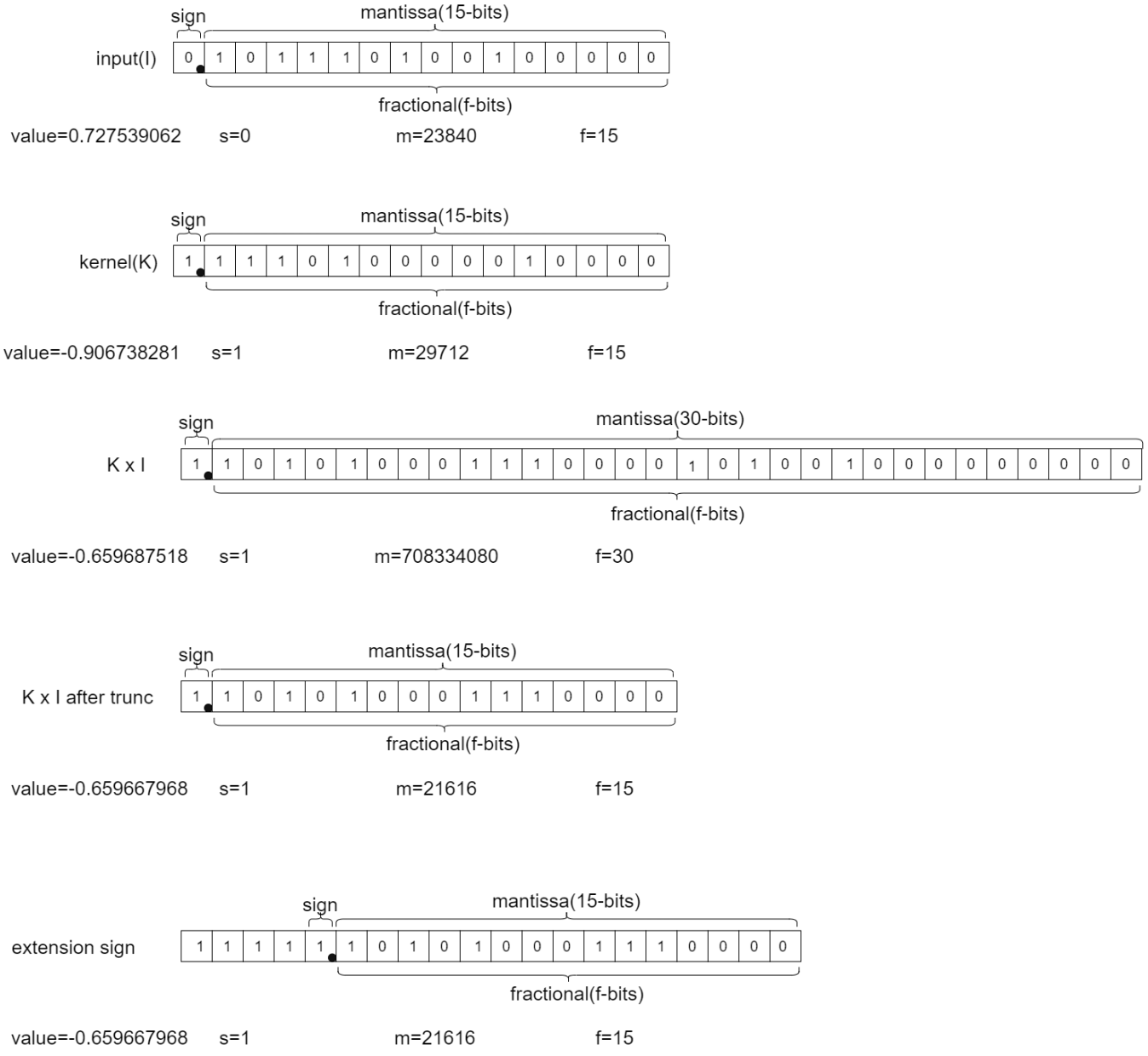


Figure 3.9. Various fixed point representation inside MAC

3.4 Internal memory

As mentioned in section 2.2, each iteration of the algorithm involves the partial update of nine locations of the output matrix. This implies having to save/retrieve these output values in multiple iterations until they are no longer affected by the weight matrix. The PE's internal memory serves this purpose.

The first problem was to understand the optimal sizing for the memory. A small memory allows to contain the explosion of the area and the power consumption but, on the other hand, it must be big enough to store the partial values needed to continue the convolution. So this memory must contain all the partial output results that may still be updated in subsequent computations. Each iteration involves a different input location and the 3x3 kernel matrix. The filter range affects three different output lines each time, as shown in Figure 2.3. This means that the memory should contain at least 3 complete rows of output. Sooner or later, the content of part of this memory will have to be overwritten with values belonging to other output rows. This can happen when all the values belonging to a row are fully computed and will no longer be involved in the convolution range between input and kernel.

For this reason, it was decided to consider a memory that can contain 4 complete lines of output. In this way is possible to recover the partial results needed for the convolution and, at the same time, download the contents of a fully-computed row to the accumulator. In Figure 3.10 are shown the internal organization of this memory and the various control signals necessary to manage the various operations. Here the architecture is explained:

- writing can take place in two different cases: to save the MAC results and to clean the portion of memory which has to be overwritten. Cleaning is needed because memory locations are always read before being written with new values.

- reading can take place in two different cases: reading partial results which are used in the MAC or to retrieve the definitive value and send them to the accumulator;

- the internal organization is divided into 4 separate memories, each of which is responsible for storing values relating to a specific output line. The control provides a 16-bit address separated into two parts: the upper part of the address (8 bits) represent the row in the output matrix and therefore in our case which of the 4 memories must be accessed. The remaining 8 bits indicate the column within the output matrix which in our case corresponds to the location within the single memory block selected. The lower part of the address can be directly used to access the location. To select the right block it is necessary to understand in which block is stored the row of interest. For this purpose, we introduce four registers, each one linked to one memory block, called "tag registers". These registers store the number of the row that is actually in the block. Comparing the upper part of the address with the contents of the registers is possible to select the correct memory block.

- The size of each memory block is related to the maximum possible size of the output matrices that it was decided to analyze, i.e. 256x256. For this reason, every block is 256 locations wide and can contain 256 different rows (tags). An address on 16 bit is provided. It is also important to underline the fact that having four memory blocks, therefore managing a maximum of four different output lines at the same time, but having much larger overall dimensions of the output matrices, along the algorithm these memories will contain different output lines and therefore it is also necessary to update, each time the single memory block must contain a new output row, by 4 positions the register associated with the single memory block in order to always maintain consistency between the values saved in the block and the corresponding row position to which those values refer. These 4 registers will be initialized

to 0,1,2,-1, respectively. The reason why the last register should be initialized to -1 and not the expected value 3 is due to the control of our circuit.

The memory is controlled by two different circuits: the PE control unit, which read values and store results, and the PE memory control unit, which detect when a block is complete and clear it. These two CU are mostly independent but need a basic form of communications. In details, the memory CU must know if a row is ready to be downloaded and understand which block contains it. On the other hand, the PE CU should be able to know if, when a new row is reached, there is an empty block to store the new values. The coordination is accomplished in a very simple way, using flip-flops (FFs). A set-reset FF is set by the PE CU(signal cu_mac shown on the left side in Figure 3.10) when a block is ready to be downloaded. In the same way, the set-reset FF is reset by memory CU when the block has been cleared. In this way the PE CU, reading the FF state, know if can overload the tag register and continue with the convolution. So, to recap:

- output FF SR=1: the various MAC units have completed calculations for one row of the output array.
- output FF SR=0: the completed output line, contained in a PE memory block, was correctly discharged outside the PE.

Concerning the tracking of the memory block to be downloaded outwards, the choice made was to associate with each block of memory a single bit, in our case contained in a FF, which signals precisely this situation. In particular, as Table 3.1 shows, in every moment only one of the four flip-flops contain "1". As the algorithm proceeds iterating through the activations new rows are reached and old rows become therefore complete and downloadable. The "1" is shared between FFs pointing to the complete row. At the beginning of the convolution the FF containing "1" is the FF3 so that, as soon as the calculations of the first row are completed, this "1" is shifted and then falls within FF0. FF0 is associated with the first block of memory, which contains the first completed output row.

current row	FF0	FF1	FF2	FF3
0	0	0	0	1
1	1	0	0	0
2	0	1	0	0
3	0	0	1	0
4	0	0	0	1
...	...	...	...	...

Table 3.1. FFs values respect to current input row analyze

To clarify, the whole working flow regarding the internal memory is shown in Figure 3.11 and Figure 3.12. In the upper part of the first matrix is shown the convolution. Then are represented, all the steps to obtain the output values. The vectors represent the four memory blocks, the rectangles are the TAG registers and the squares are the status FFs. The activations are enlighted with the same colour as the memory locations that they affect. From the second picture is clear what happen at the end of the convolution. The last three rows are downloaded from the internal memory in two tranches at the end of the computation.

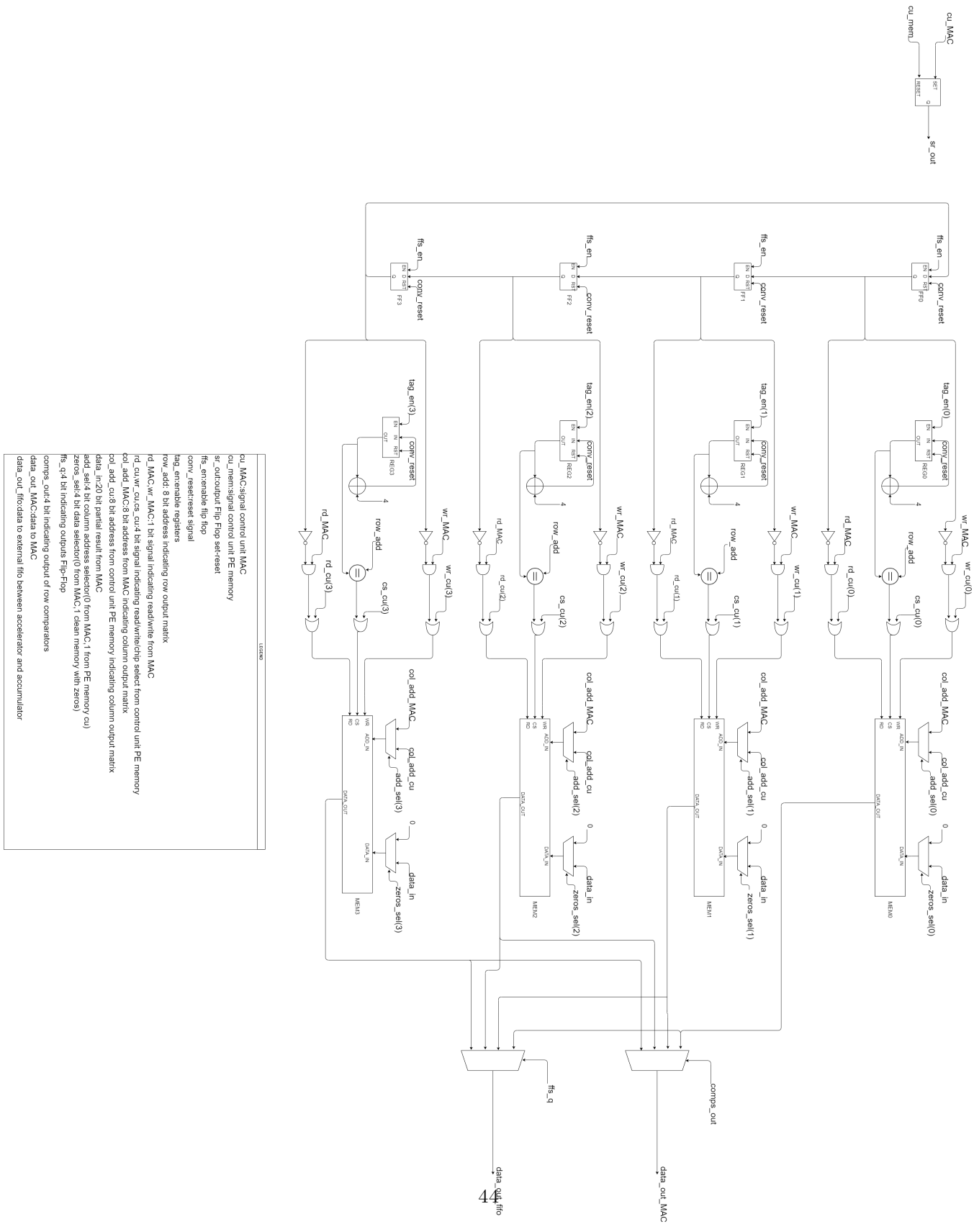


Figure 3.10. Memory architecture inside PE

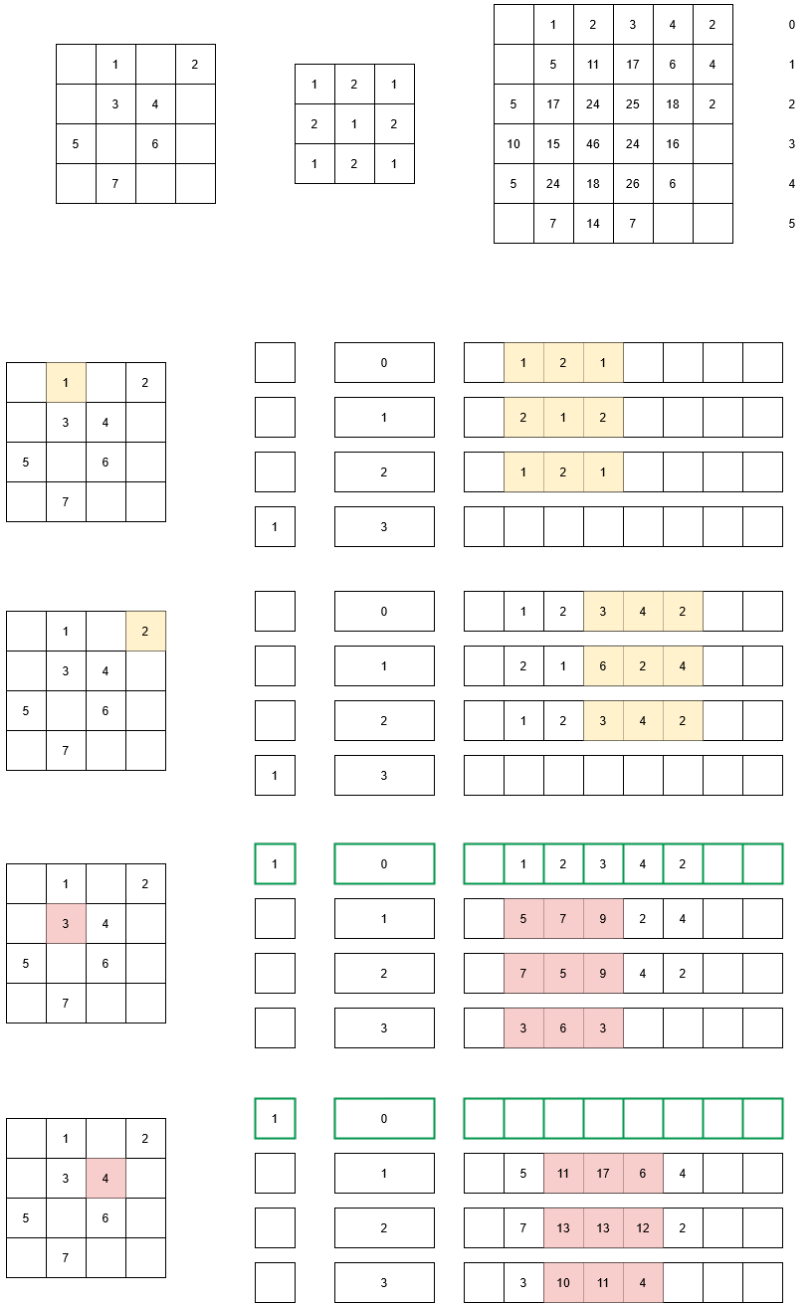


Figure 3.11. Memory working flow. In green the completed row ready to be accumulated.

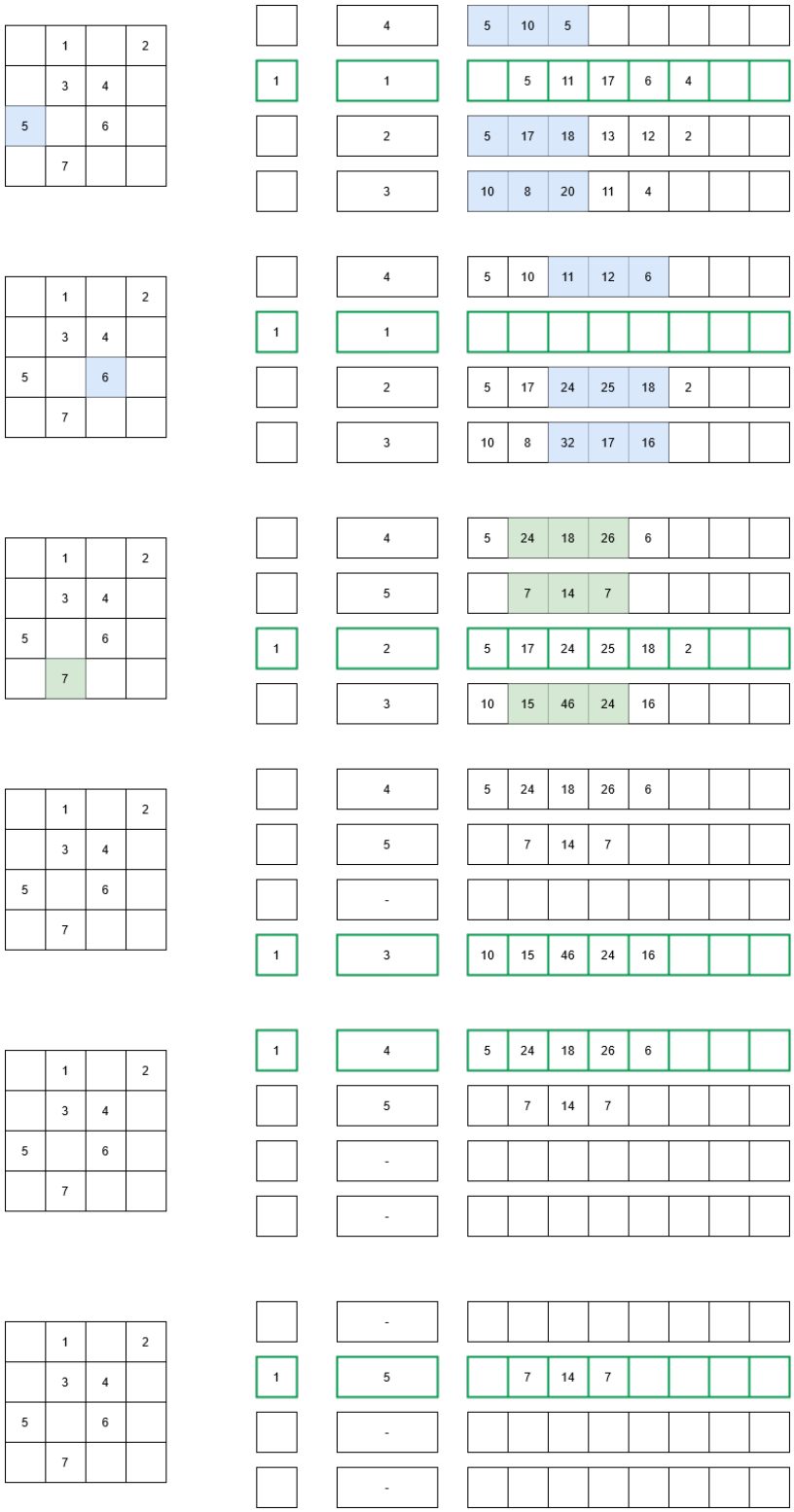


Figure 3.12. Second part, the last three downloading phases are highlighted

3.5 Control

The circuit is controlled with two control units, one to handle the computations and the other that takes care of the memory. Both are implemented as finite state machines.

3.5.1 Convolution CU

This is the more complex control of the two; it coordinates the PE circuit and makes convolution possible. Basically is divided into three moments:

- **Reset:** The reset phase takes place after the circuit is switched on. Not only all the configurable parameters and the variable has to be reset but also the internal memory; the operation can start only after the reset is complete;

- **Configuration:** In this step the user set the convolution parameters, that are stored in the configuration registers. The 3-D convolution start when a start signal is asserted;

- **Convolution loop:** This is the convolution stage when the circuit works to produce one output channel. In this phase is processed the required number of 2-D convolutions and the results are accumulated. Once the output channel is complete the circuit is ready to start over, either to compute another output channel or to start a new 3-D convolution with new parameters.

While the first two stages are very simple the third one, the convolution loop, is better described in the next paragraphs. The convolution's organization is shown in Figure 3.13.

The 3-D loop starts when the circuit detects the start signal and is shown in Figure 3.14; the algorithm is the one defined in chapter 2. The first operation is the loading of the weights in the MAC units; these values are used to compute the first 2-D output matrix and are replaced at every new 2-D convolution. Then, before the 2-D loop can start, a first input has to be loaded: this because the loading of new input and the computation shares the same state into the loop. Once the circuit enters the inner loop the operations continue until a 2-D convolution is over; due to the algorithm's nature, this happens when the last activation is processed. To understand when the activation matrix is finished a null location is added to its end. The null value can be used as spacer because the convolution exploit sparsity and so zero values are not part of the computations. When the null value is reached the end of the 2-D convolution is detected.

All the states of the 2-D convolution process are fully described below, following the logical order of the operations.

Computation and new input

In this state, a new input value is stored in the input register and the registers of the coordinates are uploaded. The computation (storing the results in the PEs output registers) can be done here as well, to avoid useless states.

Logically this is the first state of the loop but is not the first state of the algorithm because of the operations schedule. Indeed, after the loading of new input, there is the writing-in-memory stage, but it wouldn't make any sense to write in the first cycle of the algorithm, where no results are available yet. The algorithm starts then in the read-from-memory stage, preceded with a settling state, not part of the loop, which stores the first input.

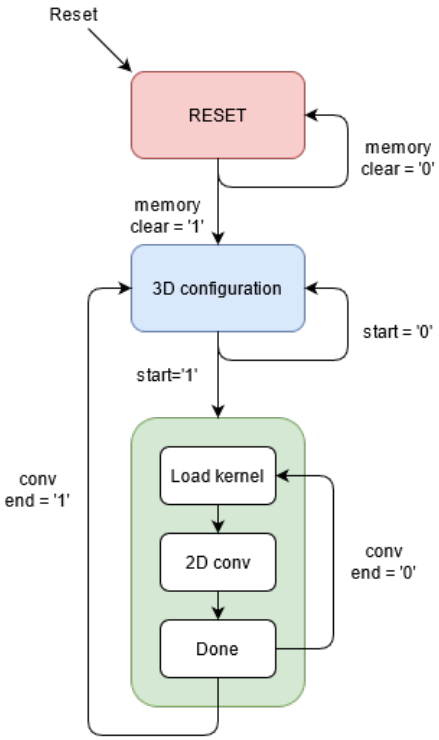


Figure 3.13. The three macro-states

Write dispatcher

From this state, needed to wait for the CCU computation of the new coordinates, starts a 3-way branch, controlled by the recycle signal. The branch destination, the write states, depends on the recycle condition.

It's worth noting that in the final cycle of the loop all the column must be written in memory because the computation is completed. In this case, the "STOP" signal is asserted and the execution flow is the "non recycle" one, regardless of the signals.

Write column states

These states[1] control the writing process. The writing, like the reading, is organized in columns, due to the nature of the recycle and, depending on the recycle, not all the states are executed. All the column are written when there is no possibility of reuse data; only two or one when recycling is possible.

In these states, the control provides a selection signal for the memory address and result's multiplexers, to write the results in the correct locations. The coordinates used to get those addresses are the ones relative to the previous cycle.

[1]In the diagram are represented only 3 states because is a logical representation. Actually, there are three sub-states for every column, for a total of nine states, one for every location to be written

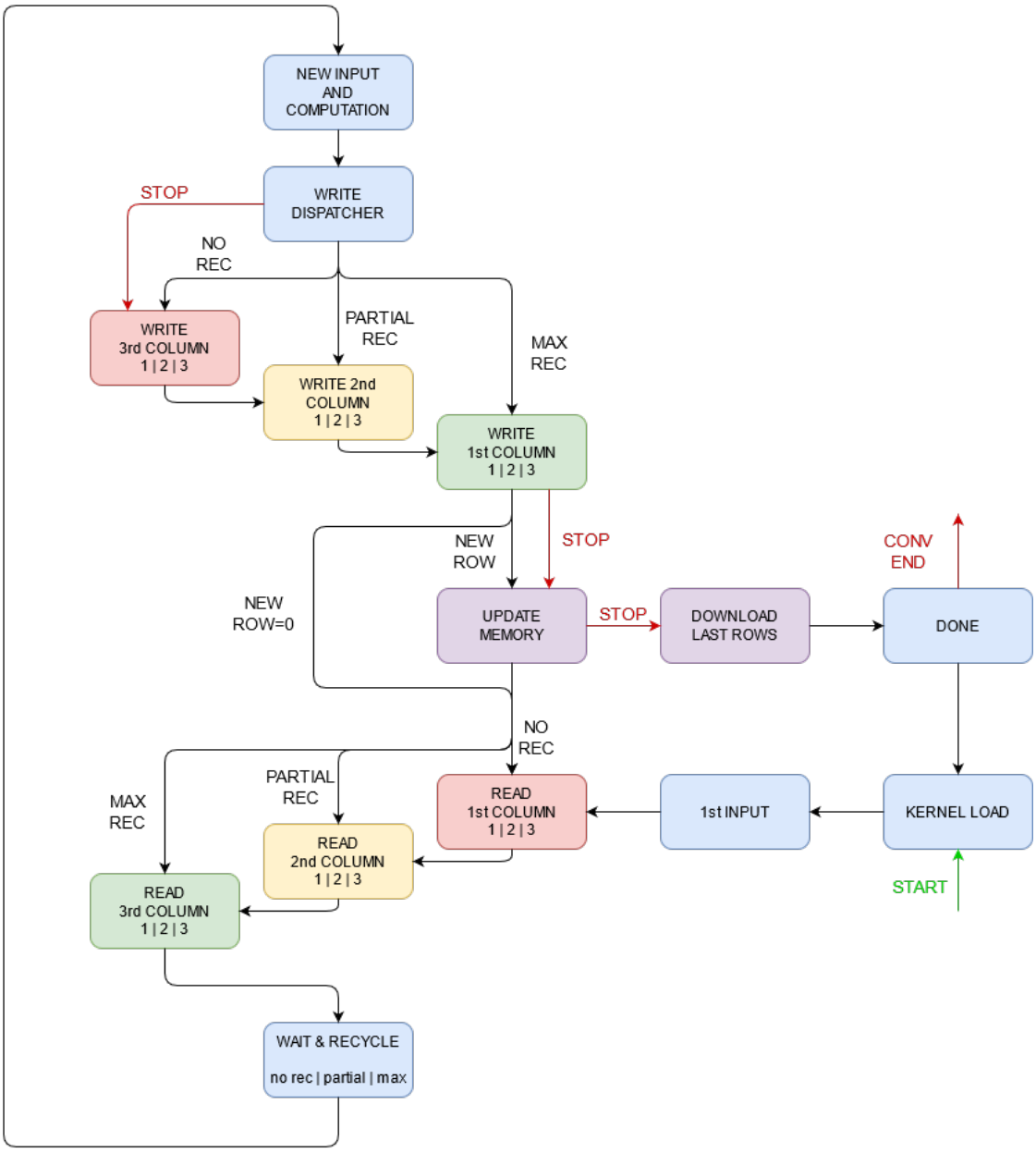


Figure 3.14. The FSM's accelerator(PE) algorithm

Read column states

These states are very similar to the previous ones, with the same logical organization. The only difference, being a reading operation, is that the circuit does not provide the data to the memory. Instead, the control must make sure that every MAC unit samples the correct value.

The reading state is also the entry point of the algorithm after the assertion of the "START" signal. In this case, all the column are read because no recycling can occur in the first cycle.

Since the memory is synchronous the output value is available at the processing unit whit a clock period delay. For example, while the FSM requires the third element of the second column from the memory, the memory output is sampled by the central MAC unit (always thinking about the square).

Wait & recycle

In this state (actually three states, one for each recycle condition) all the value that wasn't read from the memory are shared between MAC units. The FSM select the proper value for the recycle multiplexers and enable the correct registers. In addition, this state is exploited to save the last value from the memory, now available, in the bottom-right MAC unit.

Memory update

This series of states link the computation(PE) CU and the memory CU. As seen in section 3.4, we are sure that every time a new row is reached in the input matrix the oldest row in the memory contains complete values ready to be given in output. The new row condition is easy to detect from the x coordinate. When this occurs, the PE CU updates the oldest tag register with the value of the new row and at the same time signal to the memory CU that the oldest row is ready to be accumulated. The communications between the two CU takes place setting and clearing a bit in a FF.

The two CU are almost independent so is possible that when the new row is reached the oldest row in the memory has not yet been fully downloaded by the memory CU. In this case, to avoid errors, the convolution CU stops the circuit and wait for the memory.

When the circuit computes the contributions of the last activation all the values in the memory are fully computed. Before exiting the loop and start another convolution these last rows are downloaded from memory and the PE waits for the operation to be completed.

3.5.2 Memory CU

This control unit takes care of downloading the final values from memory and making them available at the output of the circuit. At the same time the row is cleared writing zeros in memory.

The FSM is triggered by the bit in the interface set-reset flip-flop and recognizes the row to process checking the status flip-flops (FF0-FF3); the hardware is the one in Figure 3.10. The algorithm is very simple: when there is a row to process it reads the final values from memory and writes zeros in their places; the addresses are generated by a counter (CONT-COL), so values are processed in order. The FSM flowchart is shown in Figure 3.15.

An important observation concerns the download of these values from the memory to the outside (represented by a FIFO). It is necessary to check that the contents of the memory are fully-computed, that is, the values contained will no longer be overwritten, but it is also necessary to check that the FIFO destined to receive these values has at least one free location to receive a value, that is, it is not full. This explains the introduction of a "WAIT-NO-FULL" state which is used to wait, in the case of a full FIFO, that someone from the outside free places in the FIFO so that the memory inside the PEs can continue downloading.

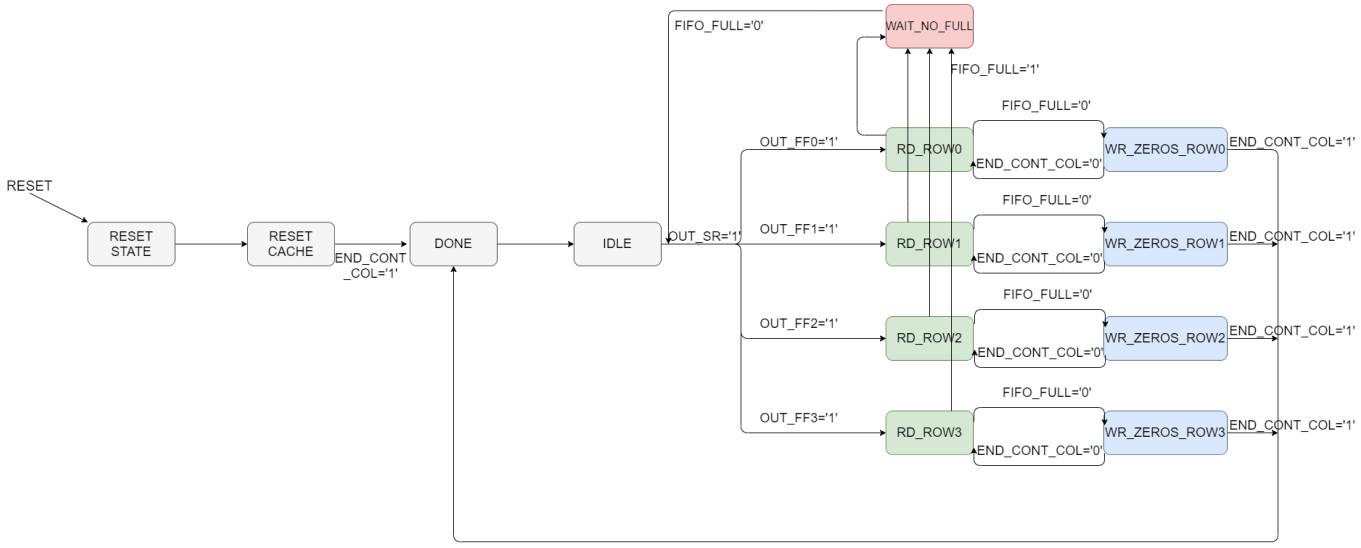


Figure 3.15. The FSM's memory PE algorithm

Chapter 4

Accumulator

This chapter is focused on the portion of the system which accumulates the values coming from the PE. The contributions of 2-D convolutions are accumulated, location by location, channel by channel, to accomplish the 3-D convolution typical of CNNs. Additionally, the ReLU activation function is performed on the final 3D results and the output activations are compressed, ignoring zero values, like the input.

4.1 Architecture

The circuit of the accumulator is very simple and once again based on the idea of a memory hierarchy. What is needed is a place to add and store, 2D-convolution after 2D-convolution, the values coming from the PE, until the 3D-convolution is over. In this way accesses from and to the main memory outside the complete architecture are saved, maximizing data reuse.

The memory in fact represents the output matrix and its dimensions depend on the size of the largest matrix processable by the accelerator. The maximum size is set to 256x256 and therefore this is the chosen memory size. Obviously is always possible to work with smaller matrices underutilizing the memory. Location parallelism is also needed to fully characterize the memory; parallelism is discussed in section 4.4. The memory's input port is connected to the output of an adder, which adds the contributions from the PE to the value already stored in memory. The memory address, essential to accumulate the values in the right position, is computed by the accumulator CU.

The last part of the circuit applies the ReLU function and integrate compression of the output values. ReLU function is the easiest activation function to apply, requiring only a multiplexer controlled by the MSB of the output value. Actually, integrating the output compression, the multiplexer is not even necessary because the zero values are not to be considered. Instead is used a counter, incremented every time the computed output is null, representing the implicit coordinate seen in section 2.3. The counter value is given in output together with the not null locations. The full accumulator circuit is shown in Figure 4.1.

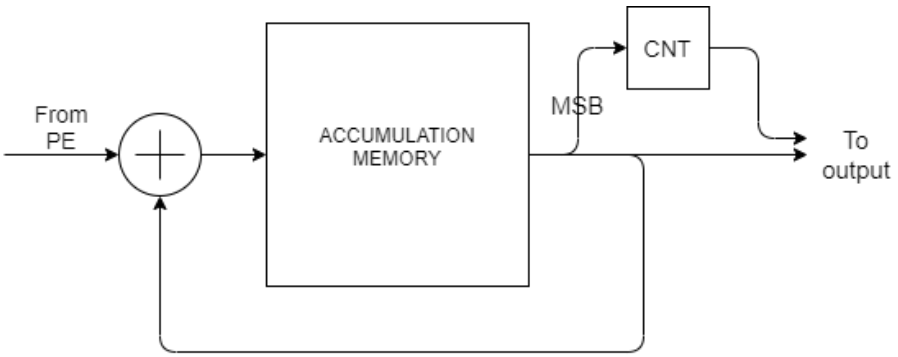


Figure 4.1. Accumulator circuit

4.2 Connection with PE

The accumulator's memory is much bigger than the PE's one and therefore slower (the synthesis data can be found in chapter 6). To not affect the accelerator's performance, instead of using a single, slower, clock frequency for the whole system it's decided to use two separate clock rates. For this reason, the accumulator cannot be directly connected to the PE and a FIFO is added between the two circuits (Figure 4.2).

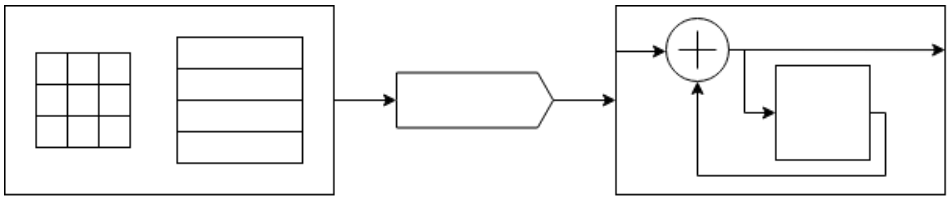


Figure 4.2. PE and accumulator connected with the FIFO

The PE produces one row of the output matrix at a time, as seen in section 3.4, so the FIFO dimension is chosen equal to the maximum row length (i.e. 256x16). This FIFO act as a storage for the PE's values and must be emptied before the following row of results is ready, otherwise the PE is forced to wait and performance drops.

The timing was analyzed in the worst-case situation, when the rows are the largest and their generation is very fast (largest matrix and maximum sparsity). It's found that when the accumulator's clock period is less than 2.5 times the PE's one, the accumulator can elaborate the incoming values and empty the FIFO before the next row is ready to be stored. Hopefully, the results of the synthesized circuit comply with this condition. Otherwise, a possible solution would be to use a dual-port (DP) memory to speed-up the accelerator, at the cost of complicating the control and increasing the memory area. Increasing the FIFO's dimension can also help but it's not conclusive: if the accumulator is not fast enough sooner or later the FIFO is filled by the PE.

4.3 Control

The accumulator needs to be controlled to works properly and the control unit, just like the PE's one, is an FSM. The main task of the control is to provide the correct addresses to the memory to do not lose the alignment between the memory locations and the values coming from the PE.

It's worth remembering that the results coming from the PE are not scattered and so to keep track of their coordinates it's enough to increment a counter every time a new value is read from the FIFO. Since the counter value is the only way to keep track of the right coordinates it's crucial to increment it at the right time, avoiding, for example, to read from the FIFO when it's empty.

Another important task is to keep track of the state of the convolution, because depending on the input channel being processed the operations to be done are different. The convolution state is tracked with a counter too, incremented every time a complete matrix from the PE is accumulated. Here are reported the tasks done in different phases of the convolution.

- **first channel:** read from the FIFO, store the FIFO's value in memory (select zero as the second addend, because memory is empty, with a multiplexer);
- **nth-channel:** read from the FIFO and memory, add the two values and store the result in memory;
- **last channel:** read from the FIFO and memory, add the two values; if the result is positive write in output the result and the coordinate, otherwise upgrade the coordinate counter;
- **first & last channel:** If a single input channel needs to be processed, it is necessary to read from the FIFO (and select zero as the second addend with a multiplexer) and save the result, if positive, direct in output, bypassing the memory.

The diagram of the FSM is shown in Figure 4.3. The combination of the "cnt_channel_zero" and "cnt_channel_end" signals is used to understand which input channel must be processed The "cnt_minor" signal is used to identify the specific case in which the first input channel to be processed also coincides with the last one.

Due to the system bandwidth (see section 5.1) that is on 88 bit, the values are saved in output four at a time. Since the number of values that will be saved is not predictable in advance, is possible that the potential of the band is not always satisfied. This happens when the number of values to be saved on output is not an integer multiple of 88 bits. For this reason, it is necessary to use "dummy" values which serve only to always guarantee the saving of values in multiples of 88 bits. These values are also a separator between different saved output channels, like the null values in the input matrices. Null values have been chosen as dummy values as only positive values are saved at the output (due to ReLU) and therefore null ones can work as spacers. This explains the meaning of the "WRITE_ZERO" and the"FIFO_WR_ZERO" states: they take care of adding the dummy values and fulfil bandwidth requirements.

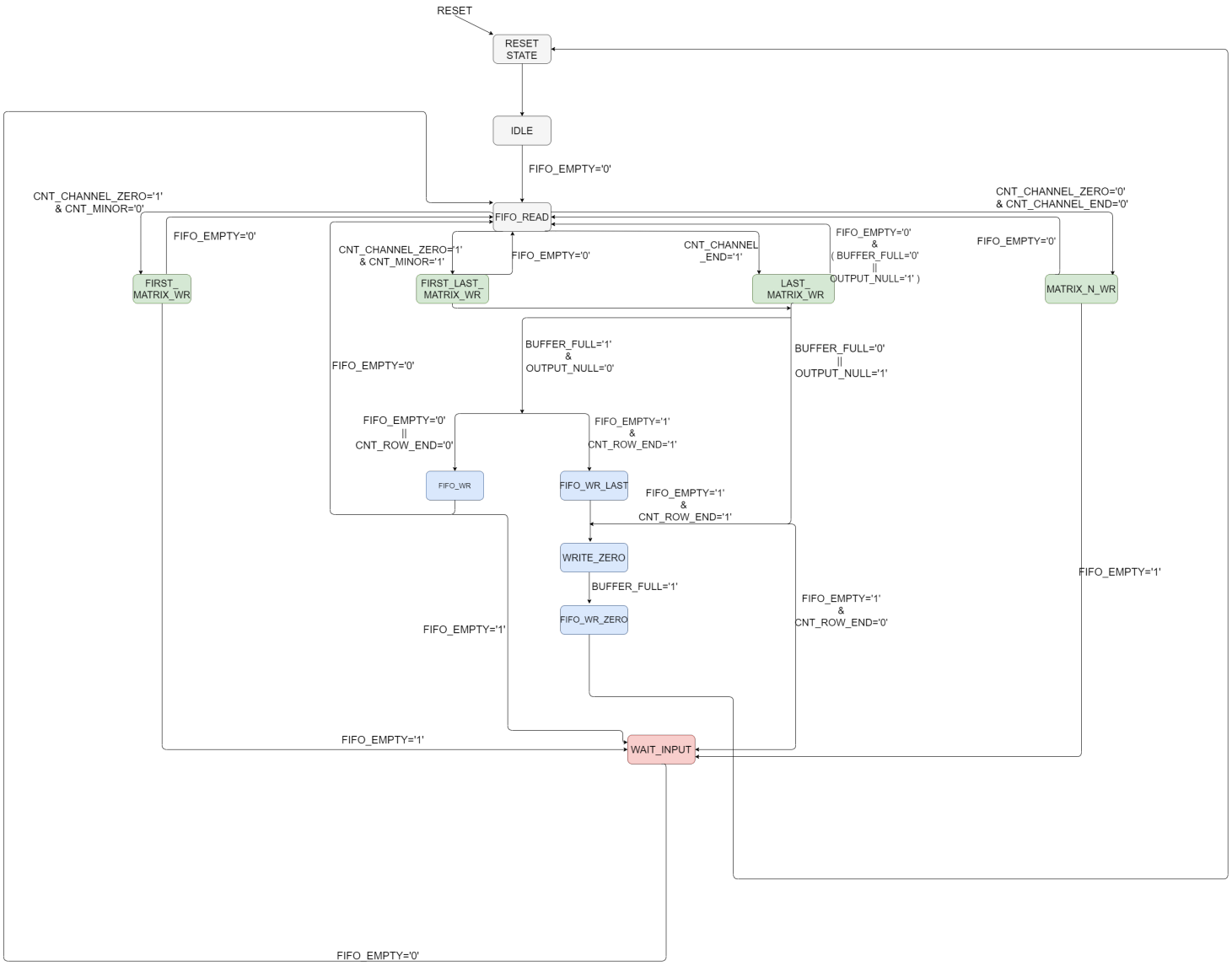


Figure 4.3. The FSM's accumulation algorithm

4.4 Parallelism and Truncation

In section 4.1 is described the memory which stores the partial 3D sum, but we do not talk about the width of its locations. The parallelism depends on two values: the width of the values coming from the PE and the maximum number of accumulations, i.e. the maximum number of input channel that the accelerator can elaborate.

To avoid saturation of the results the memory width is chosen to get correct results also in the worst-case condition. The worst-case for saturation is always add the maximum (or minimum) possible value. It's known that the values from the PE are represented on 16 bit and the maximum number of input channels is 512 (section 3.1). The worst case is adding together 512 times 2^{15}, resulting in the largest possible value of 2^{24}, which needs 25 bit to be represented. With this information is possible to fully characterize the memory, that is 256x256x25 bit, for a total of 200 kB [1].

The output of the circuit has to be brought back to 16 bit and so some bit must be ignored. The first and easiest idea is to truncate the 9 LSB of the output, avoiding overflow in any possible case. This solution works but strongly affects the precision of the accelerator, especially when working with few input channels (and so with small output values). To avoid overflow in the worst cases and improving accuracy at the same time, the truncation can be made dependant on the number of input channels. In Table 4.1 are reported the number of bit required to avoid overflow and the bit that can be safely truncated as the number of input channels varies. To implement this solution a priority encoder and a multiplexer are required. The multiplexer selects the desired portion of the 25 bit and is driven by the priority encoder. The priority encoder detects the most significant "1" in the number of input channels, allowing dynamic truncation.

input channels	required bit	taken bit
257-512	25	24..9
129-256	24	23..8
65-128	23	22..7
33-64	22	21..6
17-32	21	20..5
9-16	20	19..4
5-8	19	18..3
3-4	18	17..2
2	17	16..1
1	16	15..0

Table 4.1. Dynamic truncation

The two methods seen above ensures correct results in every situation but can be too pessimistic. Analyzing various convolutions with various number of input channels we found out that the worst-case approach cut way more bit from the result than was actually needed to avoid overflow. However, it's also true that for our tests are used matrices with randomly generated values; nothing prevents from having particular configurations where the worst-case situation can be reached. In light of the above considerations, we opted for a dynamic solution that leaves to the user the freedom to choose how many bit truncate, making the circuit as flexible as possible.

[1] The only limit to the number of input channel processable by the accelerator is related to the parallelism of the accumulator memory

In the configuration phase, while setting the convolution parameters, the user can choose the desired truncation from a predefined set. Four different truncation modes (15..0, 16..1, 17..2, 24..9) are provided, but both the number of possibilities and the kind of truncation are easily editable acting on a register and a multiplexer.

In Table 4.2 is shown the average error on a convolution with four input channels, with values taken from a real CNN. The errors are obtained comparing the ideal output (the result of an algorithm where no type of truncation is ever performed) with the output from the accelerator. Some configurations shown in the table are not included in the circuit. More details on precision and errors can be found in chapter 8, but is clear that the type of truncation highly affects the performance.

truncated bit	error [%]
0	2.2
1	2.5
2	3.7
4	9
8	125
9	237

Table 4.2. Effect of truncation on the error

Chapter 5

Complete circuit

In the previous chapter we shown the core of the accelerator: the PE, the accumulator circuit and how they are connected. We also analyzed the internal memory hierarchy, which values we store, where we store them and their lifetime inside the circuit. To complete its task the accelerator must communicate with the outer world, that is the external memory where all the matrices are stored.

In this chapter is described the external interface between the accelerator and the external memory. Is also described what the bandwidth is and how is selected the chosen value of 88 bit per cycle. Finally is also described the control unit that manages the data exchange between memory and accelerator and how the VHDL model of the system was validated performing ModelSim simulations and comparing the results with the python model.

5.1 External interface and bandwidth

To work the circuit needs to download values from an external memory, where all the data are stored. For the purposes of this work we do not fully characterize this memory; in fact the circuit will interact with text files. However, for the point of view of the accelerator, the interface is with a RAM type memory, large enough to store all the matrices involved in the convolution. About the organization of data in this memory it is possible to think, without going into details, of a division in three parts: input area, where are stored the activation matrices; filter area, where are stored the matrices which make up 3-D filters and output area, where are stored the result matrices.

To interact with this memory the circuit must provide an address and a read/write signal, other than data to be written when is needed. This task is up to an high-level control unit, which is responsible for generating signals with the correct timing to always feed the accelerator with the data to process and to memorize the results when available. This CU is described further on in this chapter.

When dealing with such a generic memory is not possible to make assumptions on its timing; the only known thing is that is a very big memory and thus with a big latency. For this reason, in CNN accelerators, the interface with the main memory is not synchronized with the computation timing. The access to memory occurs "in blocks", with more than a single word read at every cycle; the number of bit read at every access is called bandwidth. So what we need to do is to choose the bandwidth value that more suits our system. To choose the bandwidth we make some considerations:

- BW must be a multiple of the external parallelism; every input/output is represented with 22 bit (16 for location and 6 for its coordinates);
- To load a kernel matrix we have to read 144 bit (9 values on 16 bit);
- The circuit is not so data greedy, so we don't want to exceed 100 bit per cycle, that is an acceptable low value for the bandwidth [14].

The first two considerations are self-explanatory; for last one is we need to remember the flow chart in section 3.5. Every input location is elaborated within 12-20 clock cycles, depending on the recycle. Moreover, both kernel loading and output storing have a minor impact on the bandwidth. We use the same filter matrix through the whole 2-D convolution, meaning that another filter is not needed for microseconds. The output have a not negligible impact on the bandwidth but impact only in the processing of the last channels. Starting from these facts we chose a value of 88 bit for the bandwidth, meaning 4 input/output locations at the same time. This value, besides, enable to load a kernel matrix with only two memory cycles (88 bit mean five weights from the filter).

Not only the interaction with the memory happens with more than a location at a time, but also the external interface can be totally independent respect to the computation in terms of clock signal. To correctly interface the circuit with the memory a storage system is needed. We introduce FIFO memories, in a similar way to what has been done to connect the PE to the accumulator (section 4.2). There are an input FIFO, between external memory and PE's input, and an output FIFO, between accumulator's output and external memory. These FIFO are designed to contain up to a whole row of the maximum matrix, i.e. 256 locations. To simplify the interface with the memory these FIFO have not parallelism of 16 bit(256 locations) but have a parallelism equal to the bandwidth (64 locations of 88 bit). In this way the value read from memory can be directly written in them. All the FIFO used in this thesis works are realistic: for synthesis are used 65nm libraries provided by Faraday.

To interface these 88 bit width FIFOs to the circuit some minor modifications have been done to the datapath. At the input side of the PE is added a multiplexer, which fetch at every cycle the correct value from the 88 bit word read from the FIFO. At the output side of the accumulator is added a layer of four 22 bit registers: every time a new value has to be written in output is instead written in a free register; when all the registers are busy their content is downloaded in the output FIFO. A clarifying image is in Figure 5.1.

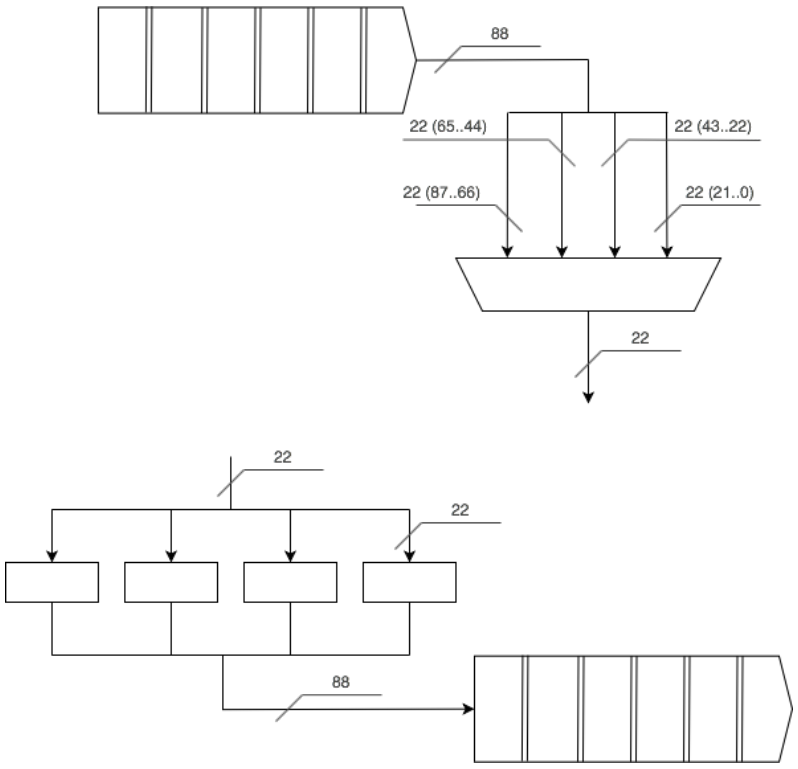


Figure 5.1. Bandwidth management. Input interface (top) and output interface (bottom)

Both input and output channels interface was discussed, but there are others data that are loaded from external memory: filter's weights. To add a FIFO for each of the nine weights is not optimal, not only for the area but also because, as said before, the kernel are updated very rarely. For this reason, instead of FIFOs, is added a layer of registers to contain the weight of the next cycle (the weights that are processed are saved inside the PE). To understand when a new filter needs to be loaded from memory is used a synchronization bit saved in a flip-flop. This bit is equal to zero at reset and when is zero the CU load a new filter in the registers, setting the bit to '1'. When the PE needs new weights samples the registers, loading the new filter and clears the bit. In this way the CU, reading the zero, understand that a new filter can be fetched from memory and stored in the interface registers. The circuitry is shown in Figure 5.2.

The final circuit, now fully characterized in all his features, is shown in Figure 5.3. Is composed by the PE, the accumulator, the internal FIFO between them, the input and output FIFOs and the weight's registers layer (the register layer, as the circuitry to interface with 88 bit bandwidth, is not shown). This is the complete form of the accelerator, ready to be tested.

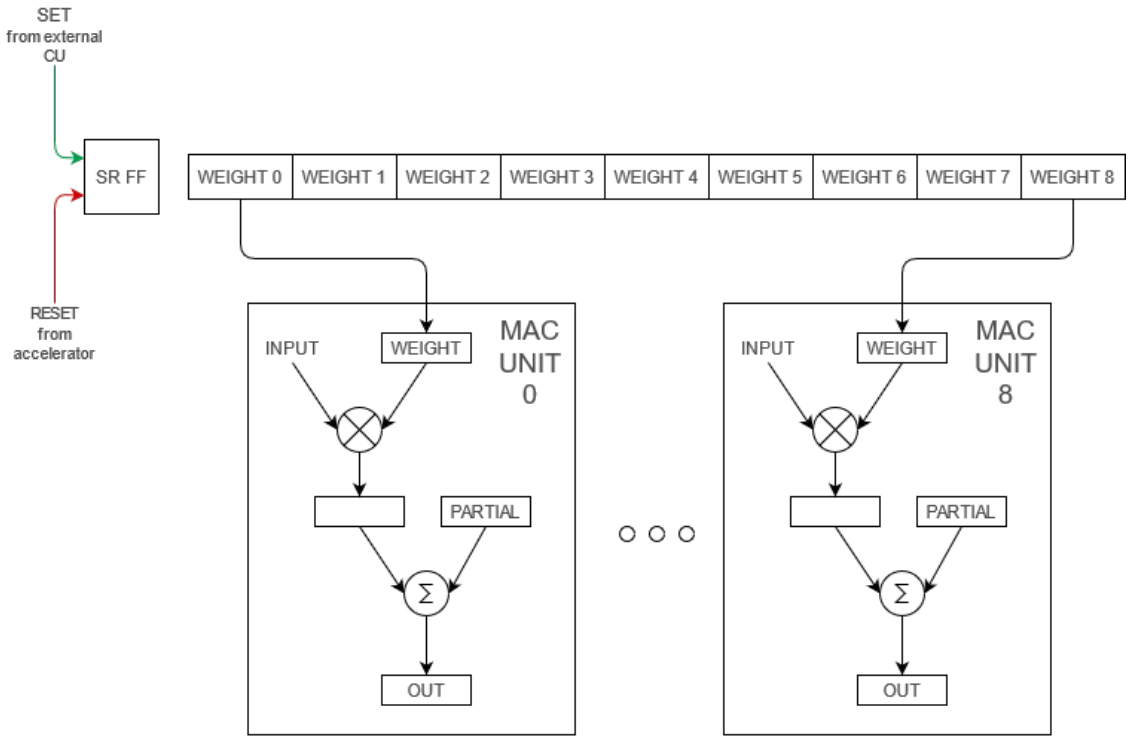


Figure 5.2. The layer of registers where the incoming weights are stored

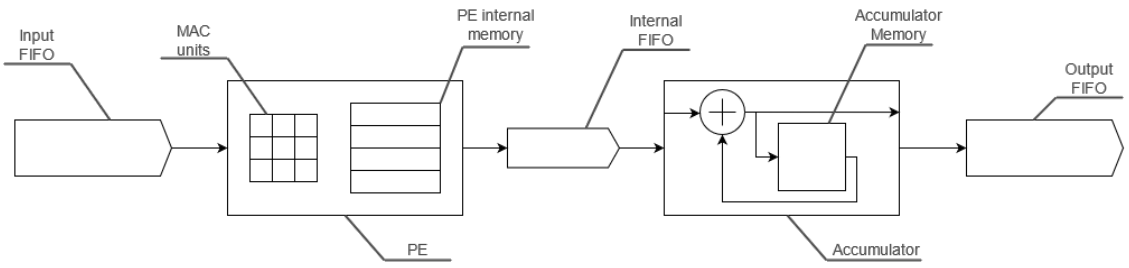


Figure 5.3. The circuit ready to be synthesized

5.2 Top level CU

The interaction between accelerator and external memory is arranged by the external control. It generates the correct signals and specially the correct address to point to the desired location in memory.

To track which data are elaborated at a given time in the convolution this CU uses indexes, other than addresses. Using together indexes and addresses is possible to obtain the full address. The indices are divided in "n" indexes and "m" indexes. In general "n" indexes point the matrix within the computation of same output channel while "m" indices keep track of which output channel is being computed. For the sake of clarity here are listed all the parameters with their role and how they change during the operations:

- **input address**: points to the current locations inside the input matrix. It's updated with every input read and is incremented by 4 (four word are read at a time). When the end of the channel is reached it is reset.

- **input n**: indicates which input channel to read. It's updated every time the end of the current matrix is reached and it's reset at the end of the last input channel.

- **input m**: in case of more output channels every input channel is convolved many times. This index keeps track of how much time the input channels have been read.

- **kernel address**: in theory it would be used to address the weight inside the 2-D filter. Actually it's not needed because all the weights are read from memory together when a new filter is loaded.

- **kernel n**: points to the 2-D filter inside the current 3-D filter. It's updated every time a new filter is loaded and it's reset when the last filter of the stack is loaded, pointing to the first filter of the next channel.

- **kernel m**: indicates the current 3-D filter.

- **output address**: points to the next empty memory location to fill with the results of the convolution. It's increased by 4 every time a new block of results is stored in memory. It's reset when a full output channel is written; pointing to the start of the next matrix.

- **output m:** indicates to which output channels the results belong to. It's updated when a full output channel is written in memory.

Is worth noting that the activations, kernel and output retrieving are (almost) independent tasks; indexes belonging to different matrices are not correlated. For example, is possible to simultaneously store in the input FIFO data from the 2nd input channel (for the 3rd time maybe) when loading the 3rd filter of the 2nd channel and storing in memory results belonging to the 1st output channel, previously computed and saved in the output FIFO. A graphical example is shown in Figure 5.4.

The high-level CU has four main tasks: starting a new 3-D convolution, loading a new filter, providing new input and storing pending output. Every task has a certain priority and is executed under certain conditions:

- **Start-up**: when the accelerator is ready to work the CU provides the first filter matrix, downloading it from the main memory, and the start signal. This is the state with the highest priority: if the circuit doesn't start to work there is no convolution.

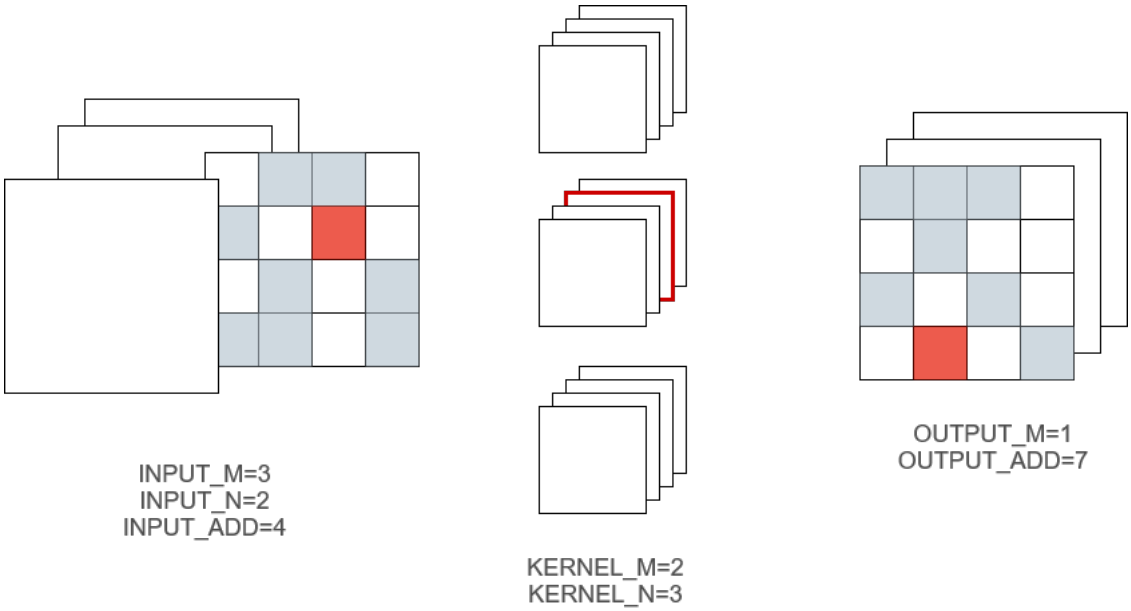


Figure 5.4. Indexes graphical example. The indexes should start from 0, here starts from 1 for sake of clarity.

- **New filter**: when the register's layer that contains the next filter is empty the CU fetch a new filter from memory (if there are other filters to elaborate) and fills the registers. This is the second state in priority because the accelerator detects the missing kernel as condition to terminate the convolution.
- **Load input**: when the input FIFO is not full the CU loads new activations, taken from the memory, in it. This state has the lowest priority, together with the store state, and is executed alternatively to it.
- **Store result**: when the output FIFO is not empty the CU read four values from it and store them in the main memory. This state has the lowest priority, together with the load state, and is executed alternatively to it.

In figure Figure 5.5 is shown a schematic with the CU's state evolution. This figure shows the control related to the basic architecture consisting of an accelerator and an accumulator. A couple of remarks are needed: flag item is used to allow alternate execution between load input and store result. To distinguish different matrices at the output, it was necessary to insert barrier values in order to clearly separate the values belonging to different output matrices. Since only positive values are saved at the output (as the ReLU is applied), null values have been used to indicate this separation. This explains the control indicated in the flow diagram "any read value is zero". Finally it is important to observe the end_sim signal handling. This signal is used, when all the output matrices have been saved, to block the circuit switches and correctly record the power consumption. For further observations see section 6.3.

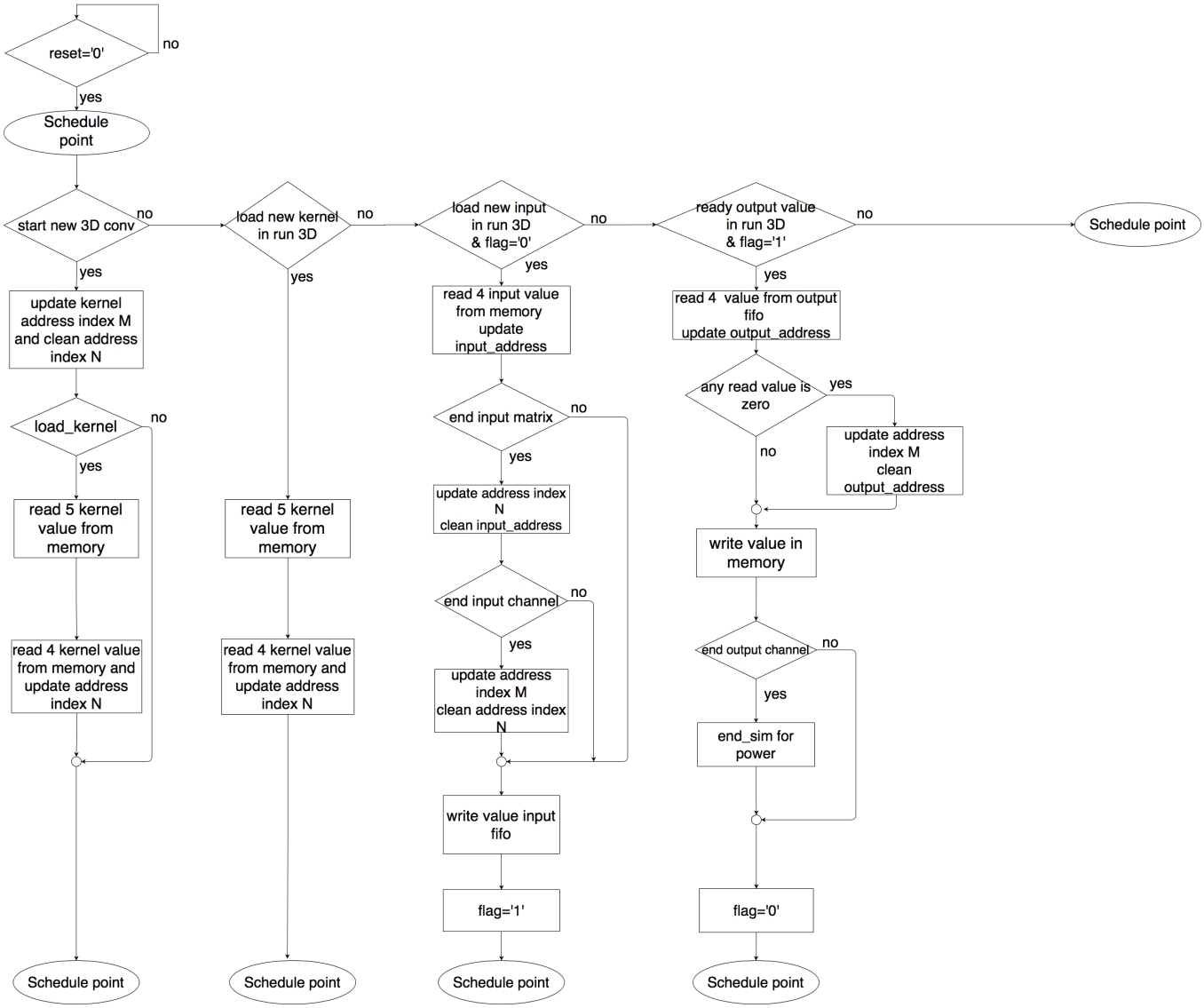


Figure 5.5. Flow diagram CU top level

5.3 Testbench and validation

Once defined the structure of the control unit is possible to simulate the circuit and validate the results, comparing them with the values from the python model. Before talking about the simulation process and the obtained results it is necessary to describe the testbench.

Testbench

Usually a testbench should read the signals from the Design Under Test (DUT - the accelerator in this case) and provide the stimuli. In the case of our accelerator the testbench should mimic the memory, reading the address and the control signals from the circuit external interface (top level CU) and providing/storing the correct values one clock cycle later. However, although stricter, a process like this brings some complications in this particular case. First, the top-level CU is specific for the circuit under test, with a PE and an accumulator. This means that to analyze different circuits (i.e. with different parallelism, chapter 7) would be necessary different control circuits, to be written again every time. Second, the data are not stored in a real memory but in text files and only the testbench can directly interact with files. Interacting with files starting from an address is not convenient and need a way to retrieve the file name would be needed. Furthermore, the external memory is an abstract entity and there is not a definition on how data are organized within it.

For these reason a little trick is employed: instead to define a real FSM inside the DUT we decide to embed the external control in the testbench through a VHDL process. In this way all the problems are solved at the same time:

- A process inside the testbench is way more easy to describe then a complete FSM and consequently is possible to create a script to automatically adapt it to different circuits, without wasting time rewriting it;
- The testbench can directly manage files. The file name is easily retrieved from the current state (input/kernel/output) and from the indexes seen above, without the need of conversions or memory mapping.

The script to automatize testbenches generation is written in python. The structure of the testbench is briefly described in section 7.4, once that the possible circuit configurations have been clarified.

Validation

The circuit is validated simulating the circuit with ModelSim and, in parallel and with the same input, computing the convolution with the python model. In this way is possible to compare the results from the simulation with the reference provided by software. The theoretical model is the python script described in section 2.6; the organization of the files that mimic the memory is represented in Figure 5.6. The matrices are generated randomly and with various sparsity factor comprised between 0.3 and 0.7. The size of the matrices can theoretically reach 256x256 but, due to the high time required, almost all the simulation use 50x50 input matrices.

The flow followed to validate the circuit is the following:

- Definition of the convolution parameters (input size, channels length, truncation mode);
- Generation of input and kernel matrices;
- Input compression;

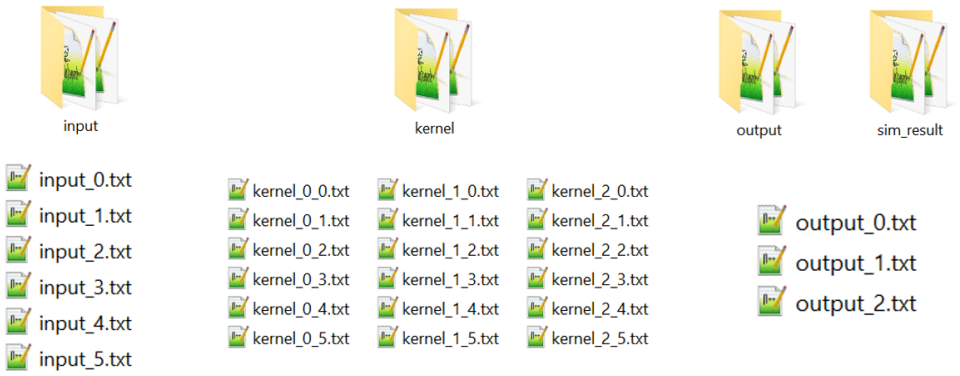


Figure 5.6. Files organization for a convolution with five input channels and three output channels

- Computation of results;
- Printing matrices on files;
- Launch ModelSim;
- Compile VHDL files;
- Run simulation;
- Compare results (using diff function)

And is the flow followed for every simulation.

Chapter 6

Synthesis and metrics

6.1 Synthesis

As previously explained, the circuit architecture was described in VHDL and subsequently simulated through Modelsim. To validate the simulation results, a comparison was made via software with python. This workflow covered the first part where the hardware model was simulated and validated.

The next step, which is explored in this chapter, is to synthesize the complete architecture, derive the area and timing metrics, simulate and validate the netlist and finally derive the power consumption. The synthesis was performed using the Synopsys 2018 tool with the UMC 65nm LL (Low Leakage) technology. The synthesis of the memory models used (FIFOs and internal memories) was performed using libraries for UMC technologies provided by Faraday.

The synthesis was carried out using a basic architecture with a single PE and a single accumulator. The architecture also contains an input and an output FIFO, necessary for interfacing with the external world, as seen in chapter 5. In Table 6.1, the overall area of basic architecture and the contribution of the various subsystems are reported. The following area values are obtained with a clock of 1.5ns for the PE and 3ns for the accumulator circuit (further details in section 6.2).

circuit	area [mm^2]
total	1.63
PE	0.084
accumulator	1.46
fifo_PE_accum	0.017
input_fifo	0.035
output_fifo	0.035

Table 6.1. Area 1accel-1accum and relative contribution

The first observation that comes naturally when looking at the table concerns the predominant contribution of the accumulator area. This is mainly because the accumulator is the part of the circuit that contains the largest memory (See section 4.1). The input and output FIFOs have a greater area than the FIFO between PE and accumulator. These memories have the same number of locations but what varies is the word size: as explained in the previous chapter, the input-output FIFOs have 22-bit parallelism while the FIFO between PE and accumulator presents parallelism of 16 bit.

These results obtained from the synthesis have aroused some observations regarding the organization of the architecture. As previously mentioned, the area values, shown in Table 6.1, are related to a single PE and accumulator structure. If we take this basic system and replicate it N-times in parallel, this would allow to linearly increase the computation that can be performed in parallel but at the cost of increase the area of N times. Since the predominant contribution of the area is due to the accumulator, the idea pursued is to consider several PEs in parallel sharing a single accumulator. All the details about the parallelism of the accelerator can be found in chapter 7.

6.2 Maximum frequency

The area values reported above were obtained with a period of 1.5ns (666MHz) for the PE and 3 ns (333MHz) for the accumulator. To correctly obtain the maximum frequency different synthesis was carried out, generating different timing report with different clock period to get the slack as close as possible to zero. The lower limit for the PE is found at 1.5 ns. The accumulator clock is chosen so that it is in full ratio to the PE clock and obviously without violations of the critical path. The minimum value that respects these constraints is 3ns.

After obtaining the area and timing metrics the synthesized netlist is simulated with the timing constraints. The netlist check is carried out with input matrices with different sizes and sparsity, randomly generated by python, and the simulation results are compared with the values obtained by the software. This process is followed both to ensure the correct behaviour of the synthesized netlist and to obtain switching activity information of the nodes under different conditions. The switching activity is essential to generate power reports and to understand how the input characteristic affect power and energy consumption.

6.3 Power

The power values are derived using the switching activity obtained in different working conditions. It is important to observe that,as the size of the matrices varies, the number of clock cycles necessary to complete the convolution changes while there are no effects on the power values. This is because power is mediated on the time, so the bigger number of operation does not influence it; however, the difference is big in terms of energy. Sparsity, on the other hand, slightly influence power because influence the recycling opportunity and so the number of memory accesses. In Table 6.2, is reported the overall power consumption of basic architecture and the contribution of the various subsystems. The values shown in the table indicate the power values by adding together the dynamic and static part.

circuit	power [mW]
total	21.447
PE	14.806
accumulator	3.623
fifo_PE_accum	1.832
input_fifo	0.602
output_fifo	0.197

Table 6.2. Power 1accel-1accum and relative contribution

It is important to say that these power values are obtained by inserting a signal in the simulation that block the switching of the clock signal once the necessary computations are done. Since the simulation time can be greater than the time needs to perform the calculations it is necessary to derive the power taking into account only the switches necessary to obtain the results.

The main contribution to power consumption is related to the PE circuit and the reasons are multiple. A first factor is the working frequency: as previously mentioned, the PE is faster and works at a double frequency compared to the accumulator. Clock frequency has a direct influence on power and it's reasonable to think that the PE consumes more than the rest of the circuit. Another important observation concerns the composition of the architecture. Any logic circuit can be characterized by a combinational part and/or a sequential part. This circuit contains many elements of memory as the computations to be performed do not depend only on the current inputs but also from "data history". The memory hierarchy, implemented in the architecture to try to decrease the latency in data retrieval and the related power consumption, still represents the main portion of consumption: about 60% of the total, which goes up to 87% considering also the basic elements of memory (registers). This is in line with the current situation of Deep Neural Networks algorithms in which the main part of the power consumption is due to continuously reading/writing data to/from memories. In the PE the number of memory accesses is very high as partial outputs are continuously saved and recovered from the internal memory. In the other memories, the data traffic is much less and this explains how, despite having a much larger memory, the power consumption of the accumulator is smaller.

Is interesting to observe the difference in consumption between the various FIFOs of the architecture. The internal FIFO, which connects the PE to the accumulator, has the highest power consumption despite being the smallest. This is explained thinking about the data in the circuit. Due to the nature of this accelerator in the external FIFOs, both at the input and output side, the matrices are sparse and only not null values are stored. In the FIFO at the accelerator output, the internal FIFO, all the values are saved, even the null ones. This because, despite being useless in the accumulation, the null values are essentials to maintain the correct alignment of the data. The greater number of data to be saved coincides with a greater number of accesses

and so leads to greater consumption. A future improvement of this work could be the research for a solution to avoid useless accumulation, even if they aren't so impactful. Looking at the external FIFOs we notice a greater power consumption in the input FIFO compared to the output one. This is explained by remembering the behaviour of 3-D convolution: an output channel is the result of the convolution of all the input channels with the filters. The FIFO in which the least values are written/read is the FIFO at the output of architecture as it contains only positive values and it's read/written only once in the whole 3-D convolution and so accesses are limited. The power consumption of the input FIFO is higher because it contains several input matrices for each 3-D convolution, even if they are sparse.

To demonstrate all these arguments related to power consumption, in the Figure 6.1 the data flow in the architecture is reported to be able to obtain an output matrix relating to a complete 3D convolution. A number of sparse (factor Si) input matrices (with dimension Li x Li) equal to in_channels and a number of dense kernel matrices (with dimension Lk x Lk) also equal to in_channels will be provided at the input of architecture. The same number of arrays will pass at the output of main part of the accelerator (PE) but with slightly increased size due to the convolution between input and kernel. All these matrices will pass unaltered in the FIFO between accelerator and accumulator(FIFO_ACCEL_ACCUM) and will then be finally added together in the accumulator. At the output of the accumulator, only one matrix will appear having the same dimensions as the accumulated matrices but, after passing through the ReLU, it will present a sparsity factor (So) such that only non-null values will be saved in the output FIFO.

A final observation concerns the registers. For most of them, clock gating could be implemented so that these registers sample the input data only at specific times established by the control. Masking the clock to the registers when they are unused could bring huge benefits in terms of power consumption, at very little cost.

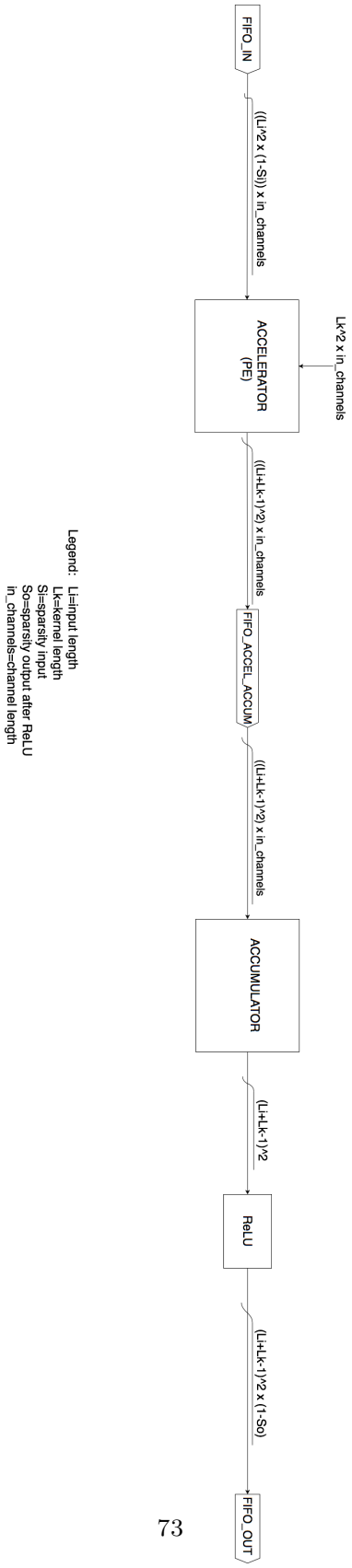


Figure 6.1. Traffic data inside complete architecture

6.4 Throughput

Another fundamental metric that characterizes the circuit is the throughput. Usually, the throughput indicates how much results are computed in the time unit, but for an accelerator, a metric like this makes the comparison difficult. This because a lot of parameters affect the ratio with which the output matrices are generated, such as the CNN type or the approach to the convolution. Since the MAC operation is common to every 3-D convolution, the number of MAC performed in a second is used as a metric for performance and is called Mega Operations Per Second (OPS).

The number of operation of a convolution depends only on the convolution parameters (matrices size, sparsity, number of channels). To complete a convolution are performed nine MACs, inside the PE, for every non-zero activation and then, for every input channel, a number of accumulations (sum) equal to the output size. Since a MAC is a sum and a multiplication together, the sums inside the accumulator are considered as $\frac{1}{2}$ operation. The number of MAC required to complete the convolution is:

$$MACs = output\_channels \cdot (tot\_activations \cdot 9 + \frac{1}{2} \cdot output\_size \cdot input\_channels) \tag{6.1}$$

This formula, retrieved from the algorithm, has been verified through simulation. Once known the number of operations, to compute the throughput is needed the convolution time that is obtained through simulation. Throughput was obtained in a totally automated way: once set the convolution parameters, the number of operation is computed with Equation 6.1 and then the simulation is started. With the help of a signal from the testbench is possible to acquire the time needed to complete the convolution and then the throughput is computed with the simple ratio between MACs and time. In Table 6.3 are reported the values of the throughput with different input size and sparsity. The throughput is expressed in MOPS.

Sparsity Input size	0.3	0.4	0.5	0.6	0.7
10x10	420	400	380	350	310
15x15	450	420	400	370	340
20x20	470	450	410	380	360
30x30	490	450	430	400	380
50x50	500	470	440	420	400

Table 6.3. Throughput with different sizes and sparsity factors

Throughput is not constant: it increases increasing the input size and decreases increasing sparsity. This is not a surprise because it's known, from section 2.5, that with very small matrices the recycle is compromised due to the shortness of rows. In the same way, increasing sparsity the opportunities for recycling are reduced because there are fewer values. In both cases the computation is longer and, with the same number of operations, the throughput is lower. The value chosen as the reference is the one related to a 30x30 matrix with a sparsity factor of 0.5 and is about 430 MOPS.

Chapter 7

Accelerator Parallelism

In chapter 6 is described the syntheses process and are reported the major metrics that characterize the circuit. In this chapter, we focus on throughput, which is the main metric for performance and is very low compared with state-of-the-art accelerators [16] [17] [18]. We describe how to improve the number of operations exploiting parallelism, how to work with more circuits in parallel, what are the limits and what is the optimal circuit configuration. Finally, we explore what happens when the workload is not uniform among all the PEs and how to avoid error in such a situation.

7.1 The idea

As seen in section 2.4, the number of MACs that can be carried out in parallel is limited by the size of the filter. This means that the single convolution cannot be speed-up any further and the only alternative is to add more circuits in parallel. The simplest idea is to replicate the entire architecture, as shown in Figure 7.1, with each circuit processing a different output channel; the flow of operation is shown in Figure 7.2. Each circuit process the same input channel, convolving it with a filter from different blocks: another advantage of this configuration is the possibility to share the input FIFO among the PEs. In this way, the throughput increases linearly. Together

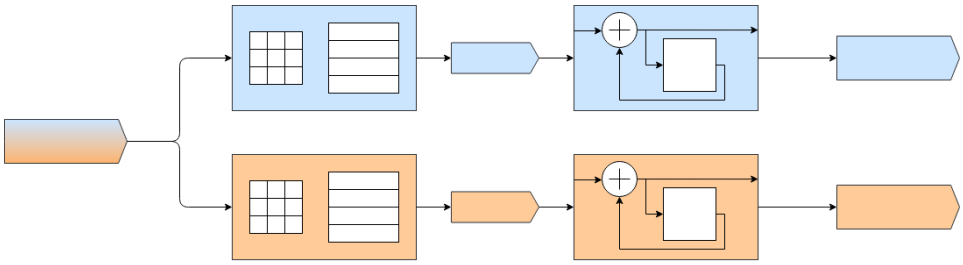


Figure 7.1. The accelerator composed by two circuits in parallel

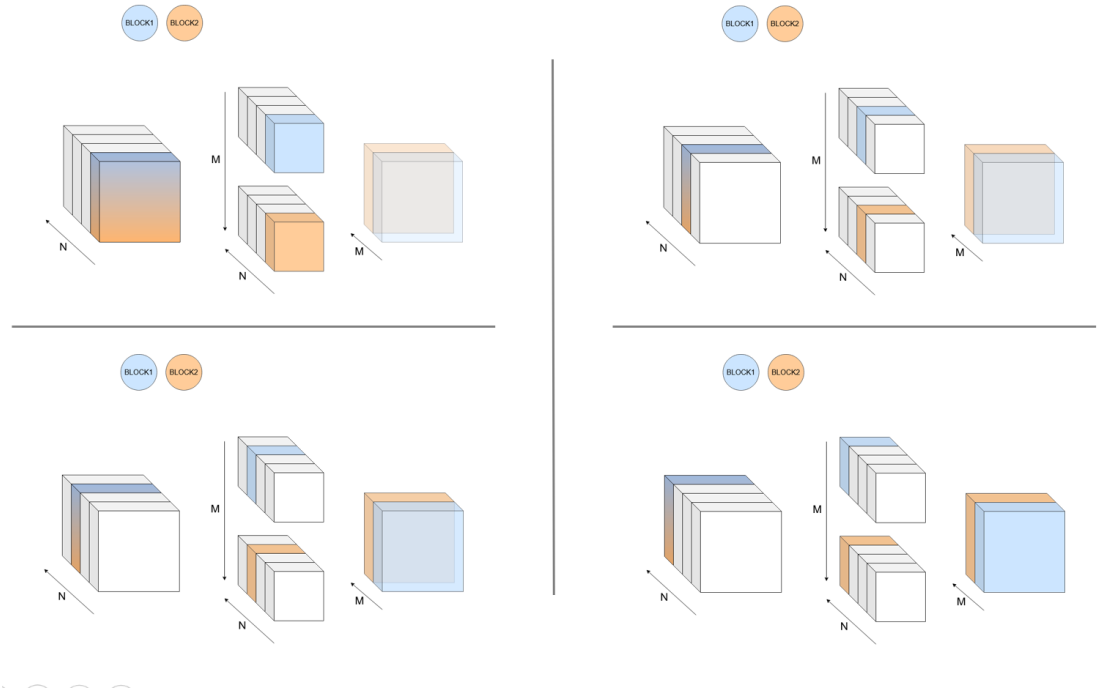


Figure 7.2. Convolution flow with two circuits in parallel

with the throughput also the area increases linearly, carried by the accumulator, soon getting too big. In Table 7.1 is reported the trend in function of the number of circuits.

No of circuits	Throughput [MOPS]	Area [mm^2]
1	430	1.63
2	860	3.26
4	1720	6.52
8	3540	13.04

Table 7.1. Throughput and area varying the number of circuits

To improve performances without increasing too much the area is possible to adopt another circuit's configuration. The PE is the portion of the circuit that is responsible for most of the computation. The accumulator, which is also very large due to the large memory inside it, just has to add together the contribution coming from the PE. The idea is to replicate only the PEs and use a single accumulator to serve several of them, adding the various contributions before the accumulation task (Figure 7.3). The drawbacks of this solution are the presence of an adder

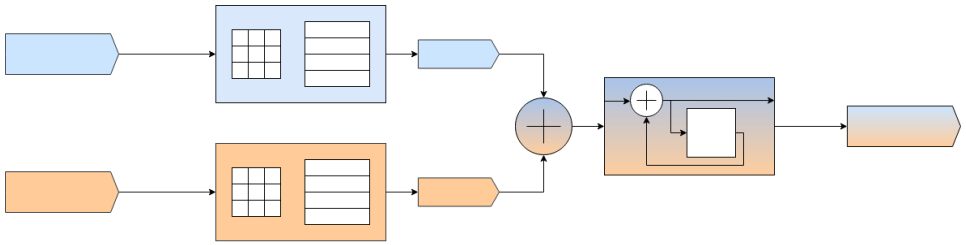


Figure 7.3. The accelerator composed by two PEs sharing one accumulator

layer before the accumulator circuit (negligible contribution) and a slight decrease in performance, which are not the ideal ones. The main advantage is that increase in area is strongly reduced. In Table 7.2 are reported the data about the area with the comparison respect to the previous solution.

No of PEs	Area [mm^2]	Area comparison
1	1.63	-
2	1.77	-46%
4	2.04	-69%
8	2.59	-80%

Table 7.2. Area as a function of the number of PE and comparison

Every PE now contributes to the elaboration of the same output channel and the operation flow is in Figure 7.4. It's worth noting that the number of 2-D convolutions executed in parallel it's equal to the number of PE and do not depend on the number of accumulators. This is obvious also looking at the two very qualitative examples gave in the pictures. The number of convolutions is the same in the two cases and also the number of steps required to complete the computations of the two output channels. Ideally, the time required to complete the tasks would also be the same but, due to some problems related to sharing a single accumulator, this is not true. The PEs work in parallel and process different input channels. Depending on the channel, and more specifically on the degree of sparsity and on how the activations are spread across the matrix, the execution time can vary; more the matrices differ,greater is the difference in the generation of the output by the PEs. This is not a problem when each PE has its own accumulator, but,when the accumulator is shared,the results of the different convolutions must be synchronized to avoid errors in accumulation. The synchronization across the channels is granted by saving the results in the FIFOs, but this means that the accumulator must always wait for all the FIFO to have values inside them. If even one PE does not have the results ready yet all the others must wait, with performance degradation. The more are the PEs,the bigger is the degradation and this, together with the limited bandwidth, places a limit on the number of PEs that can share a single accumulator, as it's described in the next paragraphs. Anyway, the performance decrease is very small (10% in the very worst case) and can be ignored in the first place. This consideration is valid

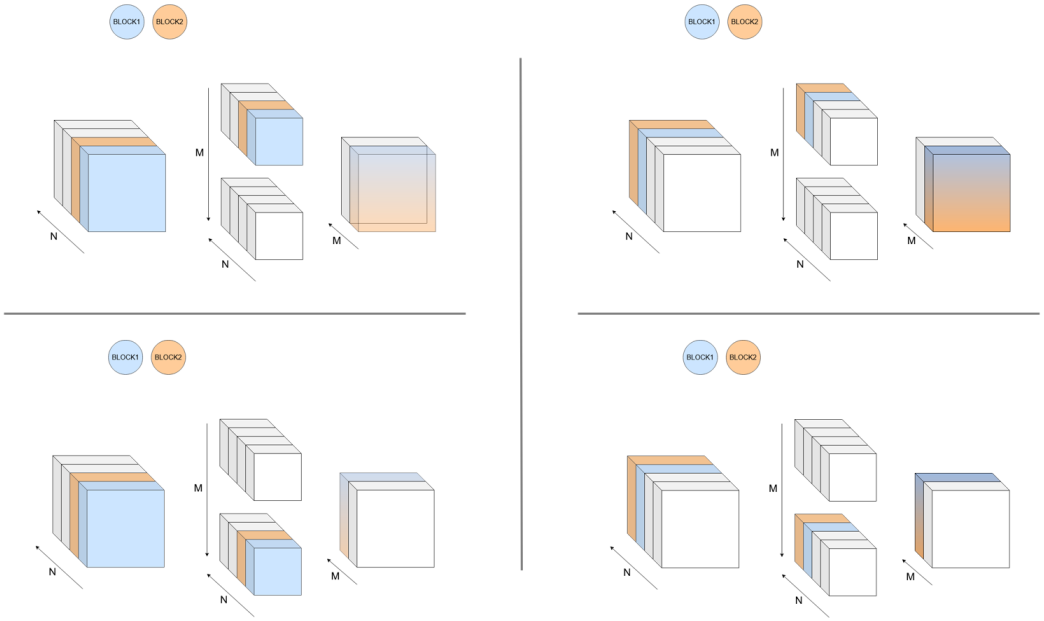


Figure 7.4. Convolution flow with two PEs sharing one accumulator

with uniform matrices but, as is shown in chapter 8, dealing with real matrices this contribution becomes bigger.

7.2 Area vs throughput vs power and bandwidth limit

Here are reported some graphics and tables showing the circuit's parameters in function of the PEs number. This full characterization shows how the parameters change as the parallelism change and allows us to choose the best circuit configuration. **N.B.** these values are computed using random matrices as circuit inputs and are therefore theoretical; in section 8.3 are shown values obtained with actual matrices coming from CNNs and is made a comparison with the ideal values reported here.

Firstly in Table 7.3 there is a summary of the metrics retrieved by the synthesis in chapter 6. These values are related to the circuit composed by a single PE and it's own accumulator, which is the "ideal" configuration, used as a reference. These metrics slightly depends on matrix size

Circuit	Area [mm^2]	Power [mW]	Throughput [MOPS]
PE	0.08	14.81	about 430
Accumulator	1.46	3.62	
Internal FIFO	0.02	1.83	-
External FIFO (in)	0.03	0.60	-
Internal FIFO (out)		0.20	-

Table 7.3. Circuit recap

(specially whit small matrices) and sparsity; they were retrieved with 30x30 input matrices with a sparsity factor of 0.5. 30x30 may seem a small matrix but increasing furthermore the size does not affect that much the values; on the contrary, greater matrices increase a lot the time required to complete the simulations.

As described above, sharing a single accumulator seems to be the best idea to improve the throughput of the accelerator. In fig Figure 7.5 the values seen in Table 7.2 are plotted and the graph in Figure 7.6 show the big area difference. The area is linear in both cases and so, the

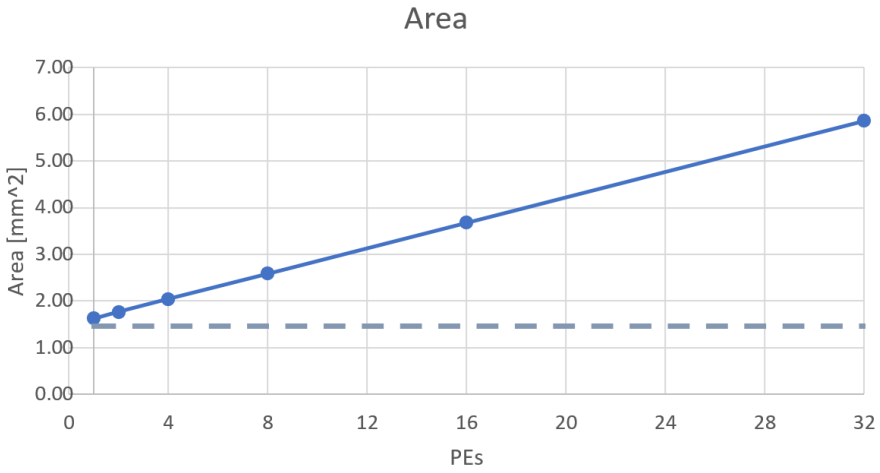


Figure 7.5. Area in function of number of PEs; the dashed line is the offset due to accumulator area

more the PEs in parallel the greater the gain.

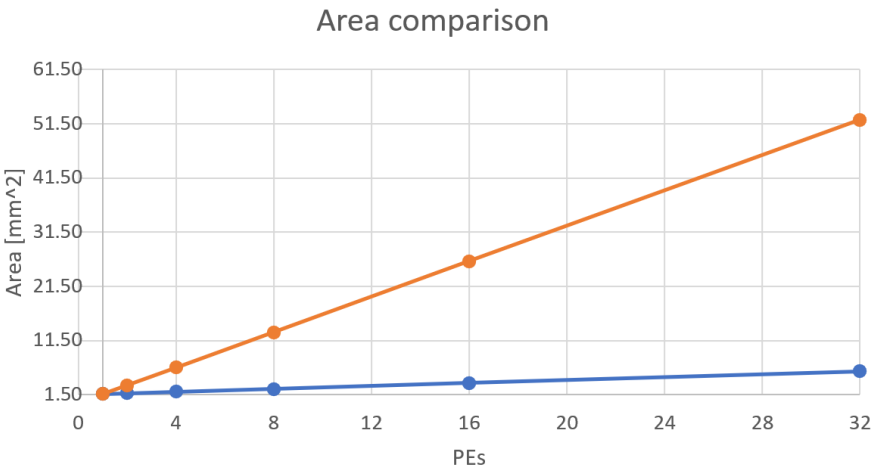


Figure 7.6. Comparison between areas; in blue the area with a single accumulator, in orange the area obtained replicating both PE and accumulator

Let's now analyze how the throughput varies with the number of PEs. In Figure 7.7 is shown the ideal trend of throughput as the PEs increase. Together in the same graph is shown the throughput over the area: since the area increases little, this number is not constant but increase with the number of PEs. As said previously, due to the circuit configuration the throughput decrease increasing the number of PEs. In Figure 7.8 are plot several trends in function of the

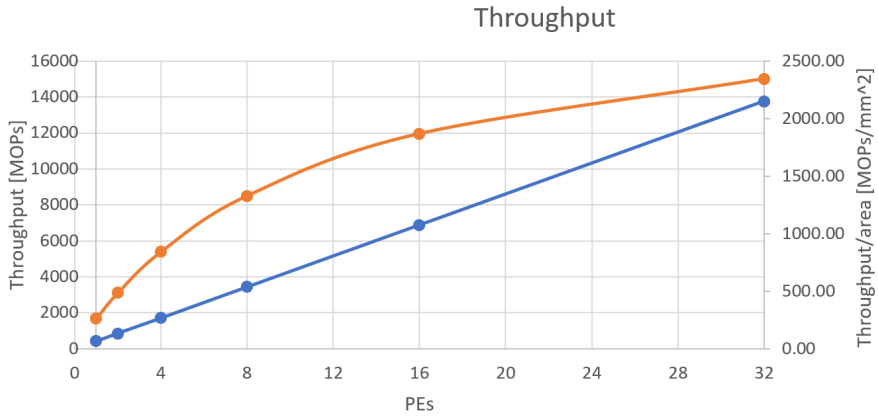


Figure 7.7. Throughput in function of number of PEs; in blue the throughput, in orange throughput density; ideal values

number of PEs, each one with a different sparsity range across the channels. The data are obtained simulating convolutions with 30x30 input matrices and 3 input channels per PEs (for instance, for the circuit with 4 PEs, the convolution has 12 input channels), to always have all the PEs working. To limit the variance, the simulation is processed three times (with different input data of course) for every circuit; the values are then averaged to obtain the plotted data. In Table 7.4 are reported the numeric data for sake of clarity. As expected, when there are few PEs performance is the ideal one. Increasing the number of PEs working in parallel performance degrades when the channels

Sparsity range No of PEs	0.5	0.45-0.55	0.4-0.6	0.3-0.7
1	437	433	433	442
2	866	872	830	837
4	1730	1717	1667	1625
8	3459	3363	3364	3116
16	6880	6704	6384	6256

Table 7.4. Throughout and sparsity

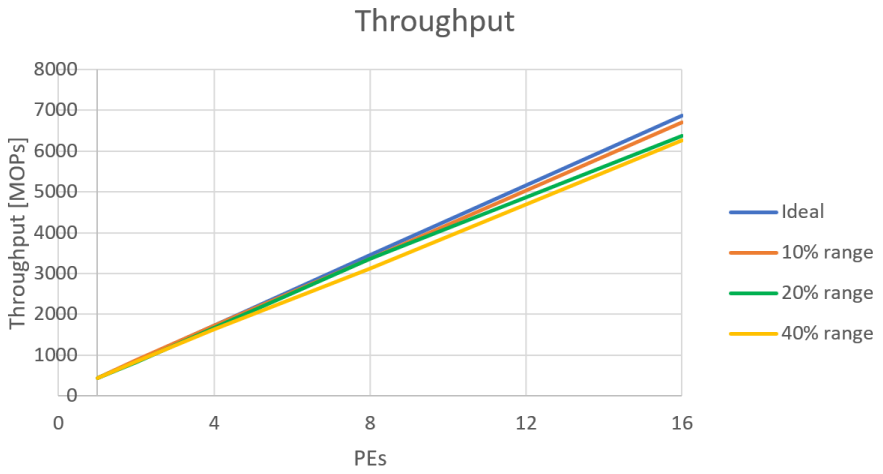


Figure 7.8. Throughput in function of number of PEs with sparsity difference

differ in sparsity. When the sparsity is fixed (in this case equal to 0.5, but is not important) the throughput is the ideal one. Another consideration to be done is that these matrices are random but the values inside them are evenly distributed. Since is not directly the sparsity but the non-uniform distribution of values (different sparsity generate different distribution inside matrices) to cause the degradation of the performance,it's important to analyze how the circuit works with matrices taken from real CNNs. This analysis is in chapter 8.

Uneven loads are not the only factor to limit the maximum number of PEs in the circuit. Another factor to keep in mind is related to the external interface: more PEs in parallel means more convolutions carried on together, on different input channels. Since the external interface has a fixed bandwidth there is a limit beyond which the accelerator can no longer be fed in time. This limit is not hard to be found starting from the bandwidth of 88 bit, considering the external clock the same as the internal one. Remembering the algorithm,is known that every PE needs a new activation every 10-20 clock periods; with four values provided to the PE every time, in the worst case there is a 40 cycles window before the circuit runs out of data to process. Considering only the write-in-FIFO states of the external CU, are needed three cycles to write a stack of values in every FIFO (two cycles to read from memory and write in FIFO and one cycle for the ineffective storing state). This means that in the worst-case condition the limit would be around 13 PEs. Luckily the worst-case condition is almost impossible to reach (and the bottleneck would be in the PE-accumulator interface in that case). In the average situation, every input value requires 18 clock periods to be processed. The useful time to provide new values rises to 72 cycles, meaning a maximum of 24 PEs (72/3); considering also the other states of the CU,a reasonable limit can be around 20 PEs. This number is verified through simulation of circuits with different parallelism and the first problems occur around 20 PEs, as expected. In Figure 7.9 there is a graph showing how performance drops when the accelerator can no longer be fed properly.

Is not possible to increase the throughput by adding more PEs in parallel, but is still possible to increase performance by adopting a hybrid solution between the two circuits seen in Figure 7.1 and Figure 7.3. Implementing more accumulators more output channels are processed in parallel, with PEs belonging to different circuits which process the same input channels (Figure 7.10). In this way, the input FIFOs are filled simultaneously (PEs could also share the same FIFO) and throughput can be furthermore increased. Obviously employing more accumulators the area increase very much. Due to the infinite circuit configurations which can be "created" replicating

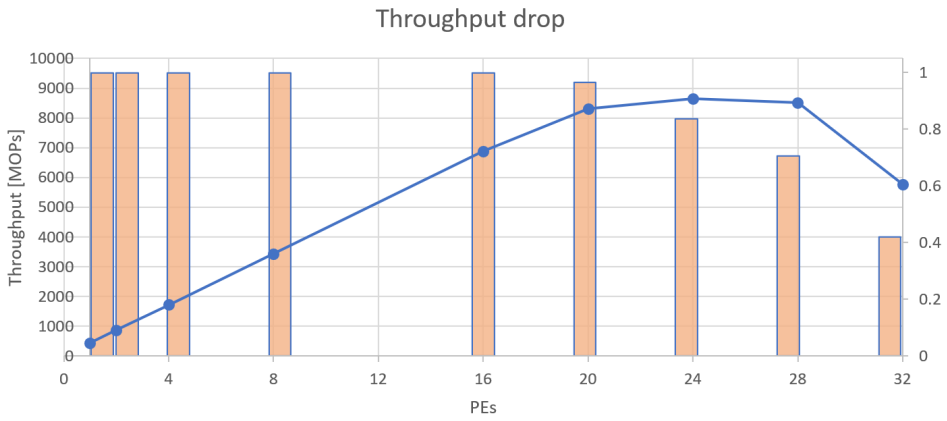


Figure 7.9. Performance drop when PE limit is excedeed. The orange bars represent the performance drop in percentage.

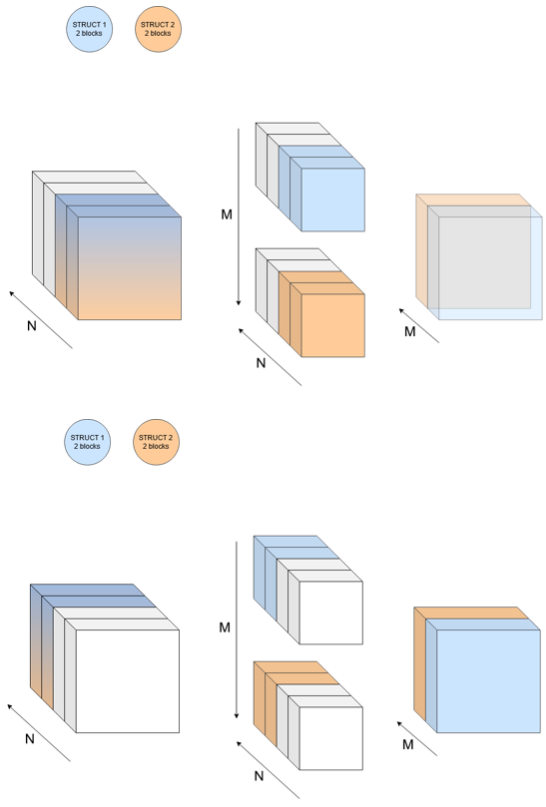


Figure 7.10. Hybrid circuit with two accumulators serving two PEs each

structures with an arbitrary number of PEs, here are not reported data about a specific circuit. The metrics for a specific accelerator can be easily retrieved, once the parameters have been set,

looking at Table 7.3:

$$Throughput = 430 * N_{PE} * N_{struct} \quad [MOPS] \tag{7.1}$$

$$Area = (N_{PE} * (PE + FIFO_{in} + FIFO_{int}) + Accum + FIFO_{out}) * N_{struct} \tag{7.2}$$

$$Area = (N_{PE} * (0.08 + 0.03 + 0.02) + 1.46 + 0.03) * N_{struct} \quad [mm^2] \tag{7.3}$$

Where N_{PE} is the number of PEs sharing one accumulator and N_{struct} is the number of times the circuit is replicated. The input FIFOs can be theoretically shared between PEs, slightly reducing area:

$$Area = N_{PE} * 0.03 + (N_{PE} * (0.08 + 0.02) + 1.46 + 0.03) * N_{struct} \quad [mm^2] \tag{7.4}$$

In Figure 7.11 there is a simple example of an accelerator composed of two structs with four PEs each and in Table 7.5 are reported its metrics.

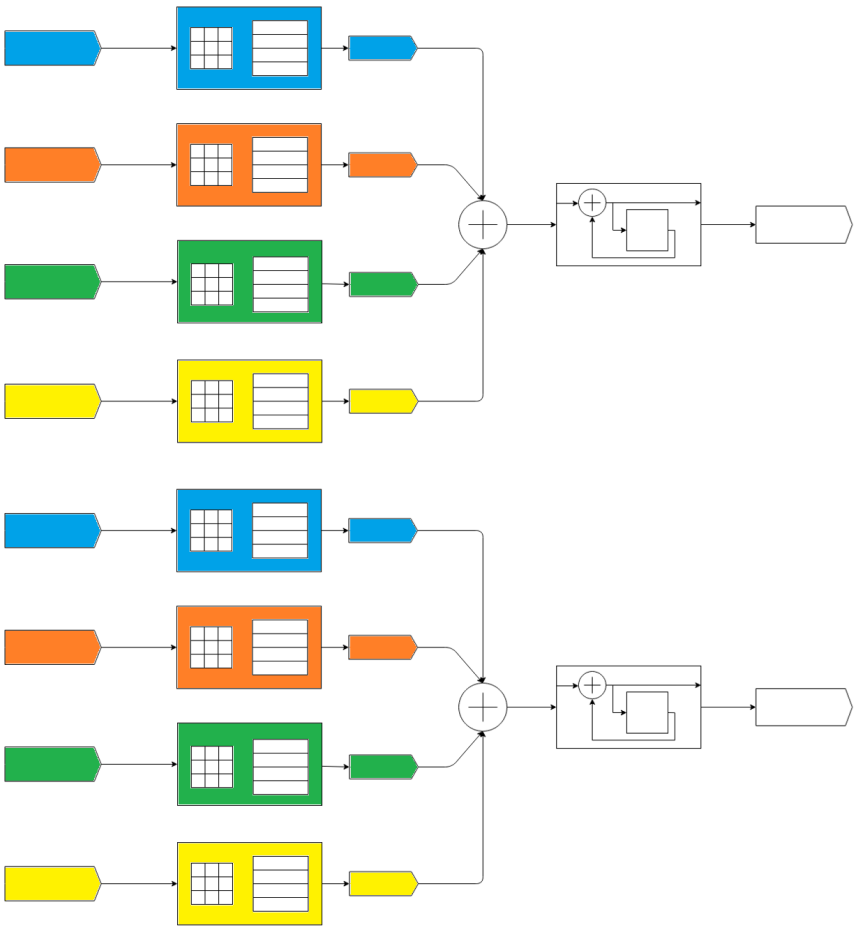


Figure 7.11. 4x2 accelerator, same colors share the same input channel

The last metrics to analyze is power. There is not much to say beyond what has already been seen in chapter 6. To analyze the power impact of the circuit varying the PEs,two metrics are introduced: power density, self-explanatory, and energy efficiency, that is the energy required for

Area	4.08 mm^2
Area (shared FIFOs)	3.84 mm^2
Throughput	3.44 GOPS
Power	118.86 mW

Table 7.5. Metrics for a 4x2 accelerator

each MAC. Energy efficiency is expressed in pJ/MAC and can be computed starting from power and throughput:

$$Efficiency = Power/Throughput \quad [\frac{J/s}{MAC/s} = \frac{J}{MAC}] \tag{7.5}$$

In Figure 7.12 are shown power and power density. Energy efficiency gets better increasing the

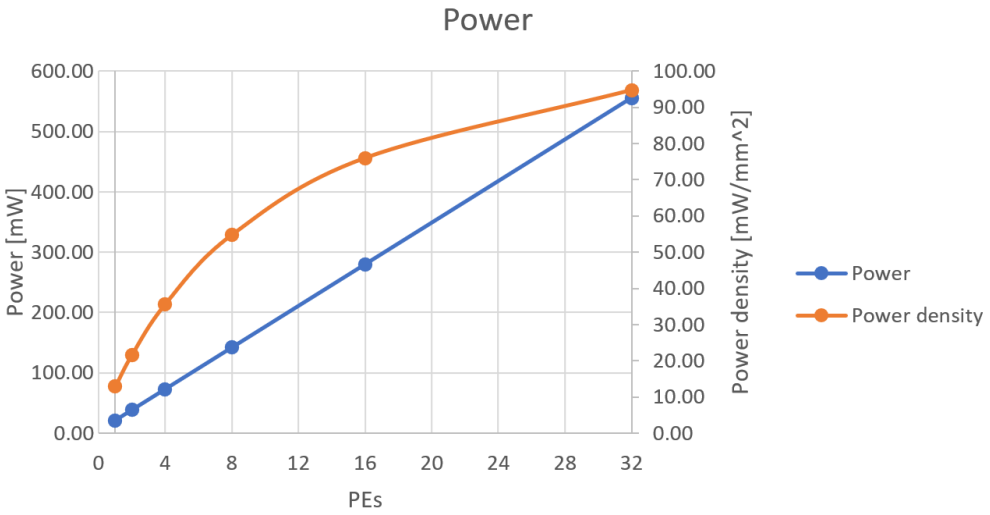


Figure 7.12. Power (blue) and power density (orange) as functions of the number of PEs

number of PEs; starts from 40.74 pJ/MAC with one PE up to 33 pJ/MAC with sixteen PEs. The values are in Table 7.6.

No of PEs	Energy efficiency [pJ/MAC]
1	40.74
2	36.61
4	34.55
8	33.52
16	32.75

Table 7.6. Energy efficiency

7.3 Unbalanced work

Processing more channels in parallel (both input or output channels) can lead to situations where some circuits have no data to work with. This happens when the number of channels is not multiple of the number of PEs or struct. In detail, when the input channels are not a multiple of the PEs,in the last convolution session some PEs doesn't elaborate any data. When the number of output channels is not a multiple of the number of replicated structs, during the computation of the last output channels there are paused structs. A qualitative example is in Figure 7.13. In the top-left pane all the PEs work. In the top-right pane is represented the 2nd convolution session with only two more input channels left: the 3rd and 4th PE of the structs have note data to work with. The bottom panes shown the elaboration of the last output channels; the input channels are blue only, meaning that only one of the two structs works. In the bottom-left corner all the PEs of the blue struct work; in the bottom-right pane only two PEs of a single struct process data.

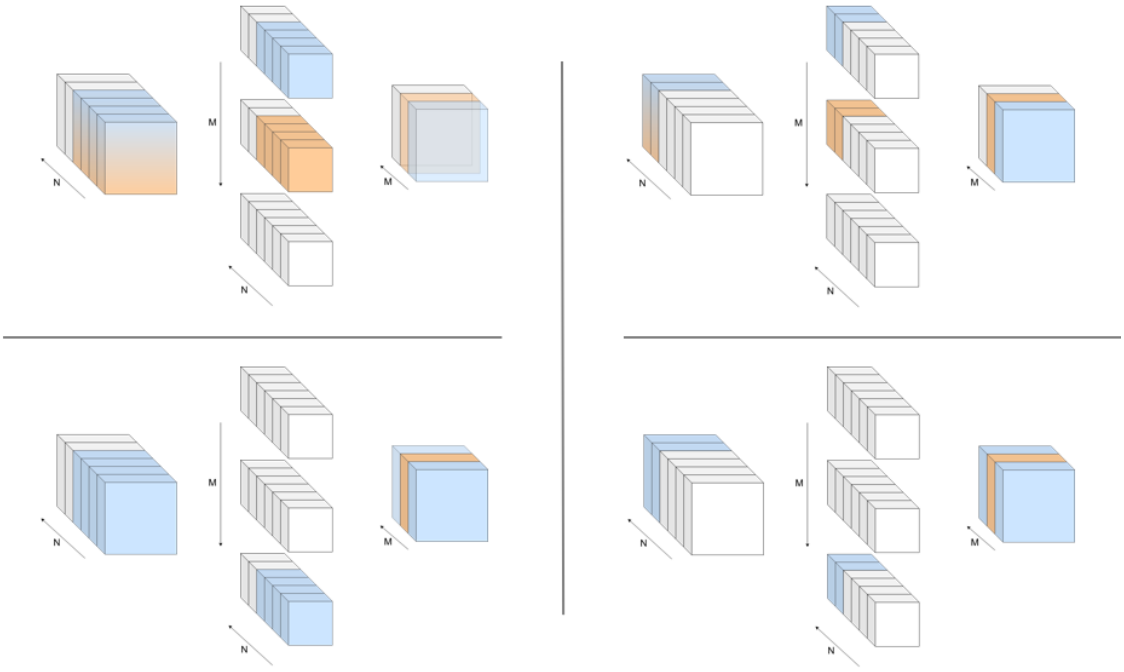


Figure 7.13. Unbalanced work control in case of 4PEs

To obtain a behavior like this, required to be able to complete any type of convolution, some modification are made to the circuit.

If the number of input channels is not a multiple of the number of PEs, will be reached a point where there are no longer enough input matrices to make all the PEs work. To be sure that the architecture works also in this situation, it is necessary to mask the PEs no longer have any matrix to process, allowing the other PEs to correctly combine the results of the last matrices.

"Masking" a PE means identifying a specific situation in which the iternal FIFO(between PE and accumulator) is empty, i.e. the results of the PE have already been accumulated and so the other PEs don't have to wait values from it. The PE is actually stopped, i.e. it has no longer

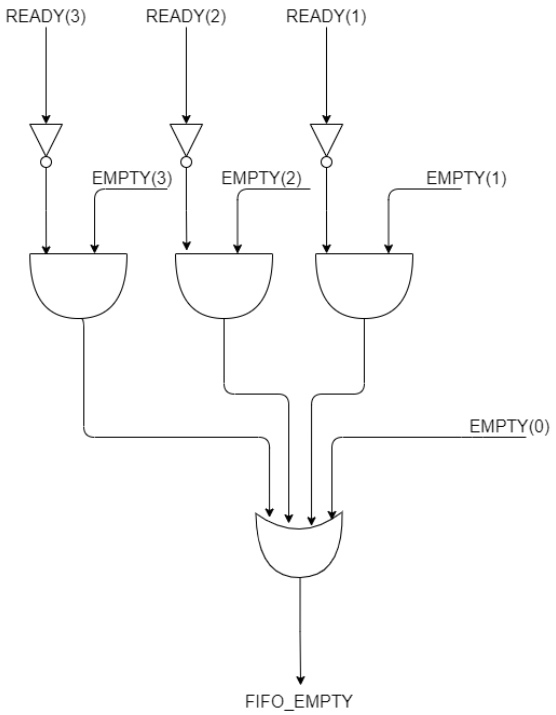


Figure 7.14. Control management for unbalanced work in case of 4PEs

received a start signal to restart a new 2-D convolution. In Figure 7.14 is shown the circuit contained in the accumulator to correctly manage the masking, in the case of 4 PEs, and the Table 7.7 shows the truth table of the key component of this control.

empty	ready	not(ready)	out
0	0	1	0
0	1	0	0
1	0	1	1
1	1	0	0

Table 7.7. Truth table empty & not(ready)

The ready signal is used by the PE to indicate that it is ready to start a new convolution. The empty signal, on the other hand, corresponds to the internal FIFO outgoing from the PEs. The combination of these two signals is enough for the accumulator to understand in which situation the PEs are. If empty is zero, the same value must be propagated to the output of AND gate, i.e. if the FIFO is not empty, there are certainly still values to accumulate and therefore the PE must not be masked.

The critical case is managed when empty = '1' i.e. internal FIFO in the architecture is empty, but it is necessary to understand the reason for this situation:

- **ready=0**: The PE is still processing some values and therefore the FIFO is empty because the consumption of data was faster than the filling with new results by the PE. In this case

the PE is still working and must not be masked. The FIFO_EMPTY signal must go to '1' to force the accumulator to wait in order to synchronize all the PEs.

- **ready=1**: The PE is stopped because has finished the convolution and is waiting to start a new one. The combination of this situation together with empty = '1' makes it clear that the PE must be masked as it is no longer producing results and is not receiving more values. In this situation FIFO_EMPTY must remain at zero since the PE is now "transparent" from the accumulator point of view.

In Figure 7.14, is reported the case with 4 PEs. The OR is used to collect the status of the various PEs. The first accelerator in the architecture, i.e. PE0, deserves a particular observation. Even in the case of unbalanced work, it is the only PE that always works because non-integer multiples imply having at least a matrix remainder that this PE will elaborate. For this reason, the accumulator needs only the empty signal of the FIFO out of PE0 and not consider its ready signal. This concerns the control part but to complete the "masking" of the PEs it is still necessary to make sure that they do not pollute the combination of the other PEs. For this reason has been introduced a MUX for each PE, controlled by the empty and ready signals, to provide a null value to the accumulator when the related PE is no longer working. These muxes can be found in Figure 7.15 where is shown the management of the architecture in the case of 4 PEs. If both empty and ready are '1' the MUX select zero so that the PE does not alter the result of the accumulation.

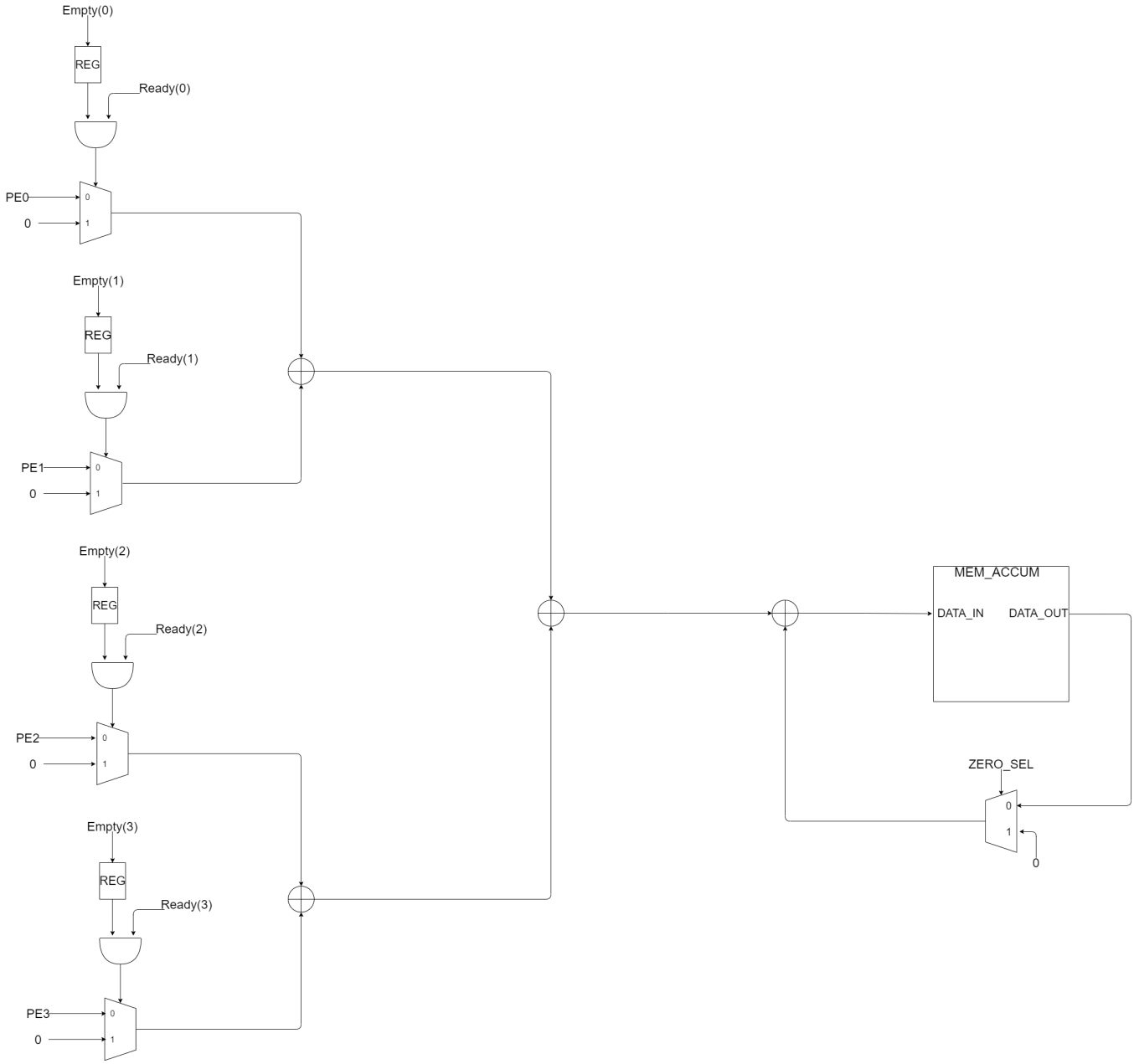


Figure 7.15. Accumulator architecture in the case of 4 PEs with unbalanced work managing

7.4 Testbench generator

Now that the structure of the parallel circuit is clear,is possible to explain in detail the parameters of the testbench, his features and how is possible to automatize the testbench generation.

Let's start with the circuit. Is possible to have several PEs in parallel working with the same accumulator. This structure can be replicated at will to increase performance. What is needed is to correctly feed with data the accelerator with a proper testbench. In section 5.2 the structure of the control is described, in the case of a simple, mono-PE circuit. To address the values correctly are used indexes, for input filter and output matrices. When the parallelism of the circuit change also the indexes are influenced:

- **Input:** input index points to the input channel to read. When there are more PEs everyone should receive data from different channels; PEs from different circuits receive the same input values. For this reason, there is a number of input indexes equal to the number of PEs (per circuit);
- **Filter:** kernel indexes points to the filter to be used in convolution. Every PE performs a different 2D convolution, even sharing the input channel. There is an "n" kernel index for every individual PE but only one "m" kernel index for every accumulator;
- **Output:** output index points to the location where to store the output channel. Every circuit computes a different output channel. There is an output index for every accumulator.

Is very important how the updating of these indexes occurs. When there is a single PE the input and kernel indexes are increased by one, pointing to the next matrix for the subsequent cycle. When there are more PEs the indexes must be updated accordingly to point to the correct matrices. In Figure 7.16 is shown a graphical example. It takes into account also imbalanced work situation. In this example the testbench works with a 2x2 circuit and so use:

- two input indexes, one for each input FIFO to be fulfilled (two "n" and two "m" indexes);
- two "m" kernel indexes and four "n" kernel indexes;
- two output indexes.

The structure of the testbench is a process that has four branches. Every branch fulfills one of the possible tasks of the control (section 5.2), using the indexes to select correct data. The structure is really regular: the tasks are always the same, independently of the circuit structure; the indexes are strictly correlated to the circuit parameters. For this reason is possible to write a script that, given in input the accelerator parameters (number of PE per accumulator, number of circuits in parallel), automatically write a testbench file able to correctly pilots the circuit. The VHDL signals and component are easily generalized using the "generate" construct.

Being the VHDL circuit described in a parametrical way, using the testbench generator script we were able to create and simulate every possible version of the accelerator just launching some python script, speeding up enormously the whole validation process.

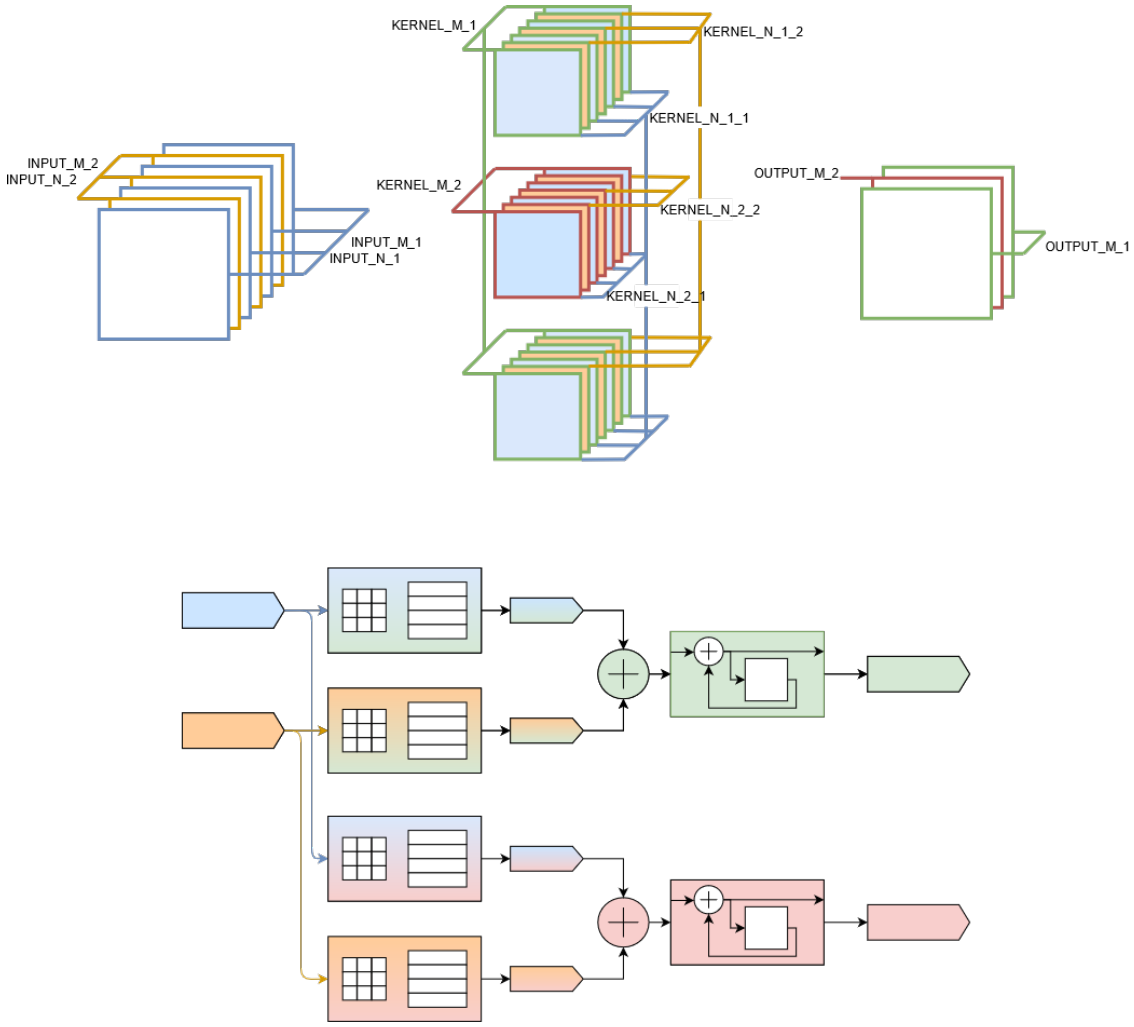


Figure 7.16. The indexes of the testbench with a 2x2 circuit

Chapter 8

Netlist validation with realistic data

In this chapter are analyzed the performance of the accelerator when working with real data. The matrices used for the validation come from a real CNN, VGG16 [12], and are extracted with a famous machine learning framework: PyTorch [15]. In the first chapter, the features of VGG16 are described and we explain how the validation work is carried out. Then both the precision of the results and the performance of the circuit in terms of throughput are investigated. The values obtained from verification are used to understand the quality of the work and to analyze where future improvement could be done.

8.1 VGG16

VGG16 is composed of several layers, as can be seen in Figure 8.1. Starting from an RGB (three input channels) 224x224 input image, a series of convolutions and pooling layers allow extracting features reducing the size of matrices.

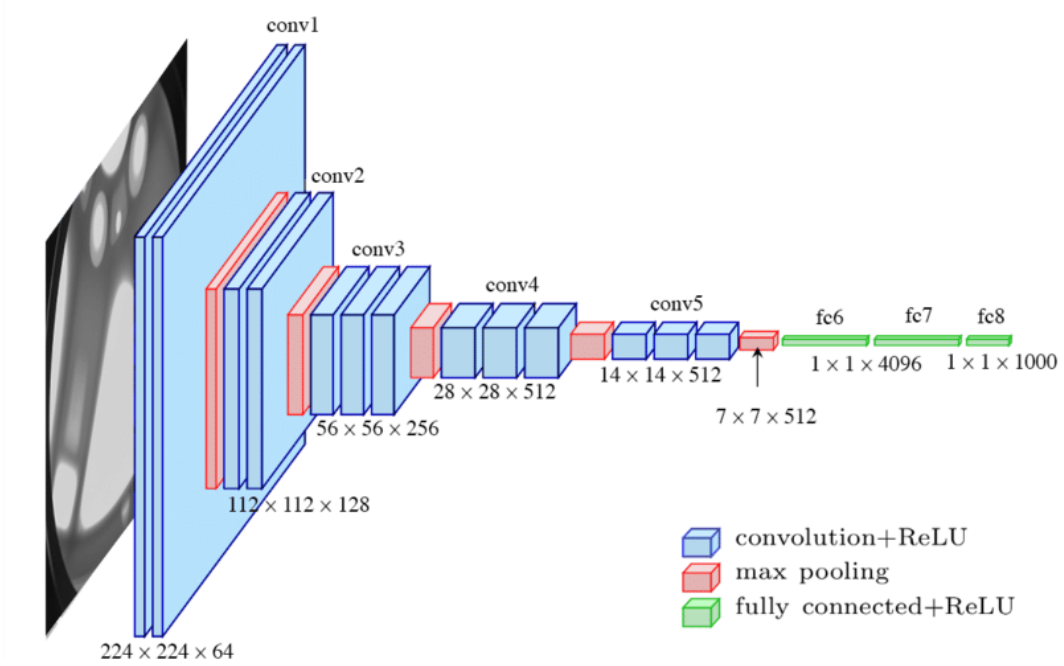


Figure 8.1. VGG16 [12]

The accelerator of this thesis work supports only the convolution + ReLU layer of VGG16 and so is not possible to execute the whole network. Anyway, to validate the results, we don't need to simulate the whole neural network from the input image to the output matrices: the individual convolutional layers are executed, one at a time. Then, for every layer, the output matrices from the circuit are compared with the expected results. In this way there are plenty of different working conditions for the circuit and, above all, the matrices are not randomly generated, allowing realistic results. The layers details are shown in Table 8.1.

The workflow to validate the circuit, to be repeated for each layer, is the following:

- Extract the tensor (set of input, weight and output matrices) with pytorch;
- Discover the maximum values in the matrices to know how to quantize values;
- Compress the input matrices and compute the number of MAC to be performed;
- Create the files to feed the testbench;
- Simulate the circuit obtaining the output channels and the simulation time;
- Compare the simulation results with the expected results from torch;

Layer	Size	In channels	Out channels	Sparsity
1_1	224	3	64	0
1_2	224	64	64	0.43
2_1	112	64	128	0.36
2_2	112	128	128	0.55
3_1	56	128	256	0.46
3_2	56	256	256	0.63
3_3	56	256	256	0.67
4_1	28	256	512	0.69
4_2	28	512	512	0.77
4_3	28	512	512	0.81
5_1	14	512	512	0.82
5_2	14	512	512	0.80
5_3	14	512	512	0.81

Table 8.1. Convolutional layers of VGG16. The sparsity is the average of the sparsity across all the input channels

- Compute the throughput.

It's clear that is impossible to perform the whole 3-D convolution for every layer (simulating a convolution with 512 input channels and 512 output channels will take ages) but also only a few output channels could provide enough information. The problem is that also the simulation of a single complete output channel (that, remember, imply a number of 2-D convolution equal to the input channels) is very long and therefore not feasible. Luckily is possible to split the validation phase into two different and independent tasks: precision and performance.

- To verify the correctness of the results is mandatory to compute a complete output channel. Hopefully, we know, thanks to validation done with random values, that the output of the circuit matches with the python results. Knowing this it's possible to substitute the matrices computed with software to the results from the circuit and avoid the Modelsim simulations. In this way is also possible to generate as many output channels as wanted.

- To evaluate the performance is not needed to generate a complete output channel. The power is the energy mediated over time, to increasing the number of input channel make the simulation longer but doesn't change the power consumption. The only effect generated by the incomplete convolution is the incorrectness of results but is not a problem because only the throughput it's being evaluated.

This is precisely what is done in the next chapter. In section 8.2 and section 8.3 it's explained how to separate the two validation phases and the results are shown.

8.2 Precision

As mentioned earlier, one of the main problems with CNNs concerns the memory footprint, the energy cost of memory accesses and, to a lesser extent, the energy required for computations. One of the most used methods to reduce the storage cost and computations requirements is quantization. In this way, it is possible to reduce the precision of the data processed along the architecture with a saving in terms of bit parallelism to be maintained. Of course, this loss of precision must be analyzed and kept under control to minimize the error between the reconstruction of the quantized data at the architecture output and the theoretical reference value provided by PyTorch.

The inputs supplied to the architecture are fixed-point number on 16 bits-two's complement with an integer part of NI bits and a fractional part of NF bits; the NF value can be seen as a scale factor that determines the position of the decimal point. The first problem encountered concerned the optimal choice of NI and NF: having a fixed number of significant bit (15) the precision that is possible to achieve is limited. Greater the number to be represented greater is the number of bits required to represent its integer part and, consequently, lesser are the bits dedicated to the fractional part. We also need to remember that the resolution is equal to 2^{-NF}. Obviously, values coming from the same layer are used for the same computation and so are quantized in the same way.

To find the optimal quantization an analysis of the input and kernel values was carried out for each layer and the results obtained are reported in the Table 8.2. When the number of bit is negative means that the value don't need integer bits to be represented (basically is lower than zero). In such cases the quantization leads to a multiplication. The first observation is that the values obtained are very different between inputs and kernels with a variation even between different layers. This is evident by the different number of bit needed for the integer part. This behaviour is due to the fact that kernels and inputs have very different ranges of values and that they change from one layer to the next. The table also shows the differentiation between the two following cases:

- **worst case quantization**: the number of bits (integer part) required to represent, without saturation, even the largest value of the layer. In this way precision is sacrificed as the number of fractional part bits is reduced. This situation is much more penalizing when there are few big values (which determinate the quantization) and a lot of values that would require less bit for the integer part.
- **average quantization**: the number of integer part bits is chosen based on the average of the input values. This improves the resolution but some values may not be representable as they would require a greater number of integer part bits. These values need to be saturated and this can lead to a greater error if there are a lot of exceding values.

This variation of the number NI involves a change of NF since the sum of these two values is fixed to the parallelism chosen for the inputs, ie 16 bits. Taking into account that one bit is reserved for the sign, the number NF can be easily obtained from the table by making $15 - NI$. This variation of NF makes the fixed-point representation dynamic. Usually, the values that are involved in computation together must have the same representation, to be aligned during sum. In the case of this circuit, anyway, all the sum are performed after a first multiplication between input and kernel. This means that a different representation of input and kernel is allowed and will result in a different scale factor after the multiplication.

This concerns the processing of input data after which, as indicated in section 3.3, in various points of the architecture the parallelism is reduced as alone changing from floating point to fixed point, without reducing bit-width, does not reduce the energy or area cost of the memory. These two concepts (scale factor NF and bit-width) are closely related to each other and both affect the

Layer	quantization type	Int.bits I	Int.bits K
1_1	worst	2	1
1_1	mean	0	-2
1_2	worst	4	0
1_2	mean	-1	-4
2_1	worst	5	0
2_1	mean	1	-4
2_2	worst	5	-1
2_2	mean	0	-5
3_1	worst	6	0
3_1	mean	1	-5
3_2	worst	6	-1
3_2	mean	1	-5
3_3	worst	6	0
3_3	mean	1	-5
4_1	worst	7	-1
4_1	mean	1	-5
4_2	worst	6	-1
4_2	mean	1	-6
4_3	worst	6	-2
4_3	mean	0	-6
5_1	worst	6	-1
5_1	mean	0	-6
5_2	worst	7	-1
5_2	mean	0	-6
5_3	worst	6	-2
5_3	mean	0	-6

Table 8.2. Statistical analysis on the weights and activations dynamics

output values. An example demonstrating this link is shown in the Figure 8.2. This figure shows

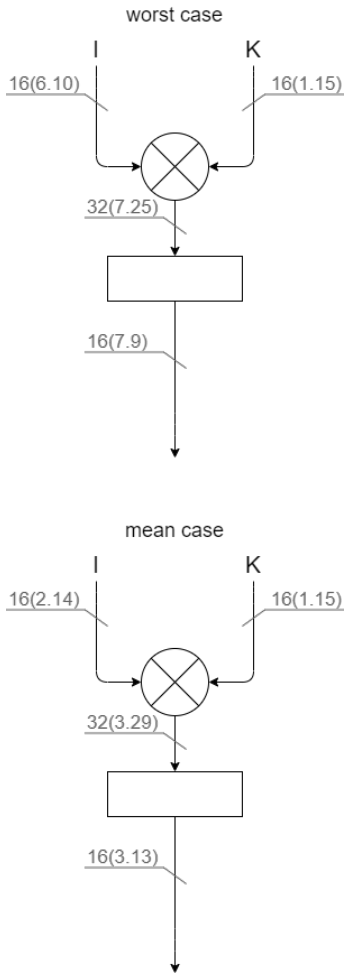


Figure 8.2. Worst NI choice and mean NI choice for layer 2.1

the case of layer 2_1 of the VGG16 network indicating the difference between the worst case and the mean case for the choice of NI and the relative consequence when it is necessary to reduce the parallelism at the output of the MAC multiplier. The choice of the subdivision between the NI part and the NF part must be consistent with the range of input values, otherwise the risk is to eliminate a lot of useful dynamic with the reduction of parallelism.

For example, if most of the input activations are numbers < 1, it is convenient to use the mean configuration as it is only one bit of integer part is enough to represent such small numbers. If the worst configuration is chosen, which includes 6 integer bits, the fractional part would be penalized with the subsequent reduction of parallelism and 5 integer bits would be kept,which will never be exploited as the dynamic is restricted to numbers < 1 .

Python simulations have been performed to find the error in the case of worst and mean

quantization. The mean error relative in output (M_E_R) is obtained with:

$$M\_E\_R = \frac{1}{N\_matrix} \cdot \sum_{m=0}^{out\_channel} \cdot \frac{1}{row \cdot column} \sum_{i=0}^{row} \sum_{j=0}^{column} \frac{matrix\_2D[i][j] - matrix\_ref[m][i][j]}{matrix\_ref[m][i][j]} \tag{8.1}$$

where N_matrix represents the number of output matrices for a specific layer, row and column indicate the dimensions of each matrix and matrix_2D indicates the results of the model while matrix_ref is the theoretical result provided by pyTorch. The Table 8.3 shows the relative errors in output for each layer. Observing the values of the table it is evident that both the quantization

Layer	quantization type	mean error relative [%]
1_1	worst	6.9
1_1	mean	232
1_2	worst	44.8
1_2	mean	94.4
2_1	worst	54.9
2_1	mean	96.2
2_2	worst	41.6
2_2	mean	99.6
3_1	worst	79.6
3_1	mean	99.6
3_2	worst	66.3
3_2	mean	98.3
3_3	worst	79
3_3	mean	99.2
4_1	worst	81.5
4_1	mean	98.3
4_2	worst	75.5
4_2	mean	99.9
4_3	worst	57.1
4_3	mean	99.9
5_1	worst	82.3
5_1	mean	99.7
5_2	worst	94.6
5_2	mean	99.9
5_3	worst	79
5_3	mean	99.9

Table 8.3. Mean error relative for each layer

types lead to big errors, that are quite similar. This because, though increasing the resolution, the "mean case" quantization introduces the saturation of the biggest values, increasing the error.

To effectively decrease the error a trade-off is needed between the two limit cases of quantization. The optimal number of integer bits for each layer can be found with a refined analysis of the dynamic of the input matrices. To understand which is the best choice we analyze the layers studying the distribution of the values in the matrices. In Figure 8.3, Figure 8.4, Figure 8.5 and Figure 8.6 is shown the distribution of activations (in orange) and weights (in blue) for each layer. On the x-axis, there is the number of bits required to represent the integer part of the number

while the y-axis represents the number of occurrences in the layer, in percentage. From the

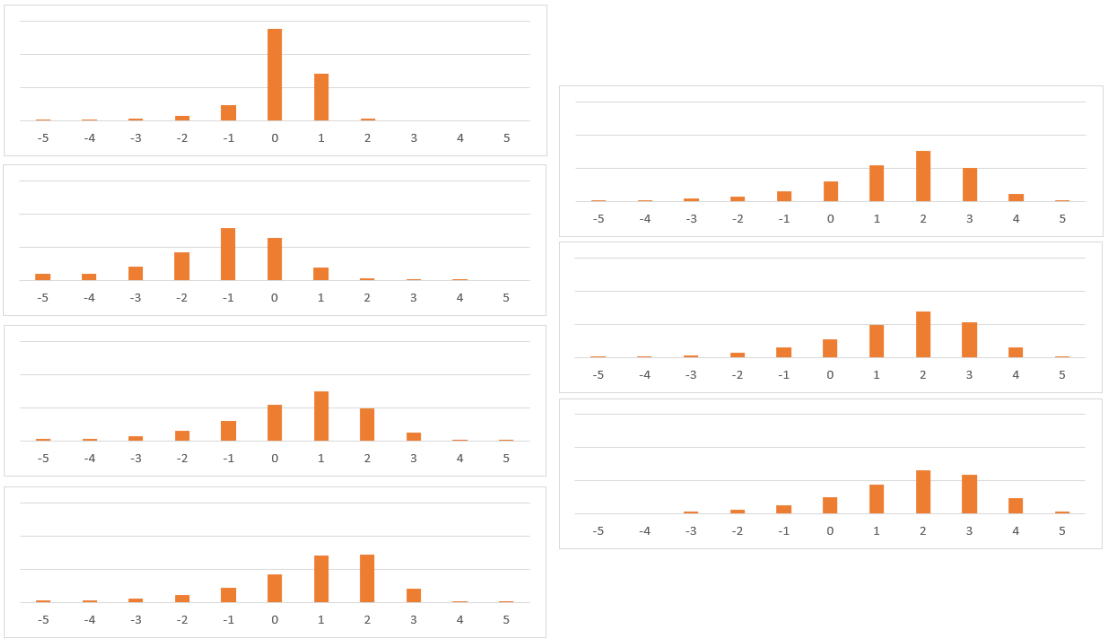


Figure 8.3. Input dynamic, percentage per number of bits, layers 1, 2 and 3

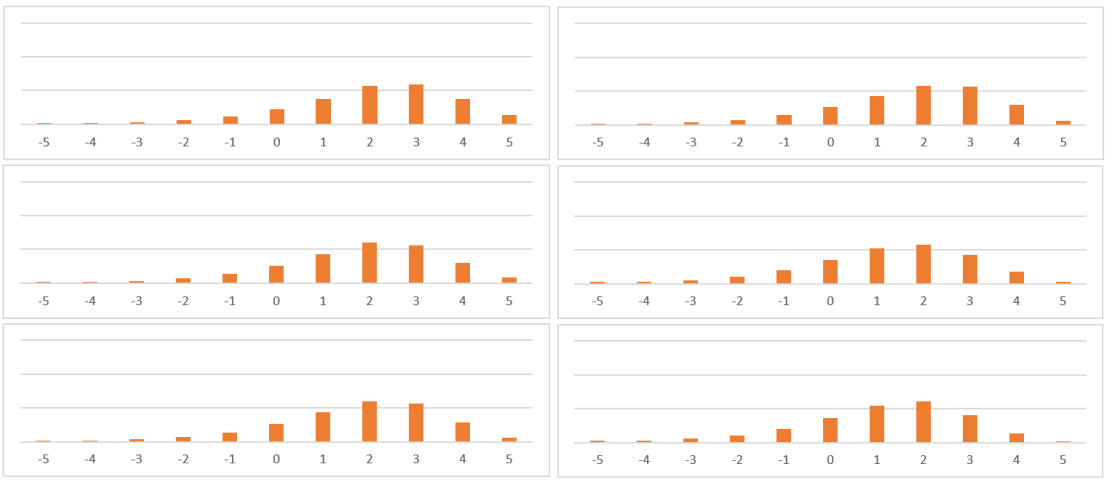


Figure 8.4. Input dynamic, percentage per number of bits, layers 4 and 5

graph is clear how much bits to use in every case. In Table 8.4 are reported the errors per layer using the best trade-off to quantize. Note how the number of bit used is coherent with the graphs about dynamic.

Searching for the trade-off between the two cases (mean, worst) effectively allows to significantly reduce the errors in output from each layer even though the architecture continues to keep,

Layer	Int bits I	Int bits K	error [%]
1_1	1	0	4.5
1_2	2	-1	12
2_1	3	-2	15.8
2_2	3	-2	14.2
3_1	4	-2	17.8
3_2	4	-3	19.6
3_3	4	-3	27.9
4_1	5	-3	21.7
4_2	5	-4	24.9
4_3	5	-3	24.6
5_1	4	-4	32.3
5_2	4	-4	35.4
5_3	4	-4	23

Table 8.4. Smart quantization and relative error

Figure 8.5. Kernel dynamic, percentage per number of bits, layers 1, 2 and 3

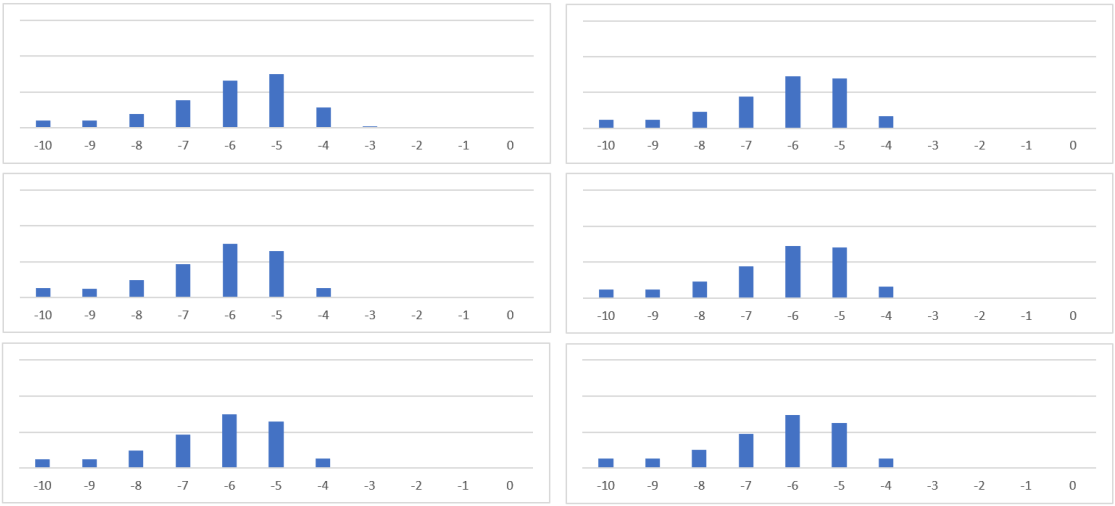


Figure 8.6. Kernel dynamic, percentage per number of bits, layers 4 and 5

within it, significant reductions of the parallelism. This is further proof of the importance of the fixed point representation choice and its close link with bit-width. Another possibility, that is not analyzed here, could be to slightly increase the parallelism inside the architecture by evaluating the further reduction of errors and the drawback of the increase in the area.

8.3 Performance

A first overview of performances can be found in chapter 6 and chapter 7. The biggest difference between the matrices used in those chapters and the matrix in VGG16 is the uniformity. In figure Figure 8.7 there is an example taken from the last convolutional layer of VGG16 compared with matrices with the same size but randomly generated. The sparsity is equal to 75% in both cases, but the values are distributed in very different ways. This difference is even more evident with bigger matrices, as in Figure 8.8.

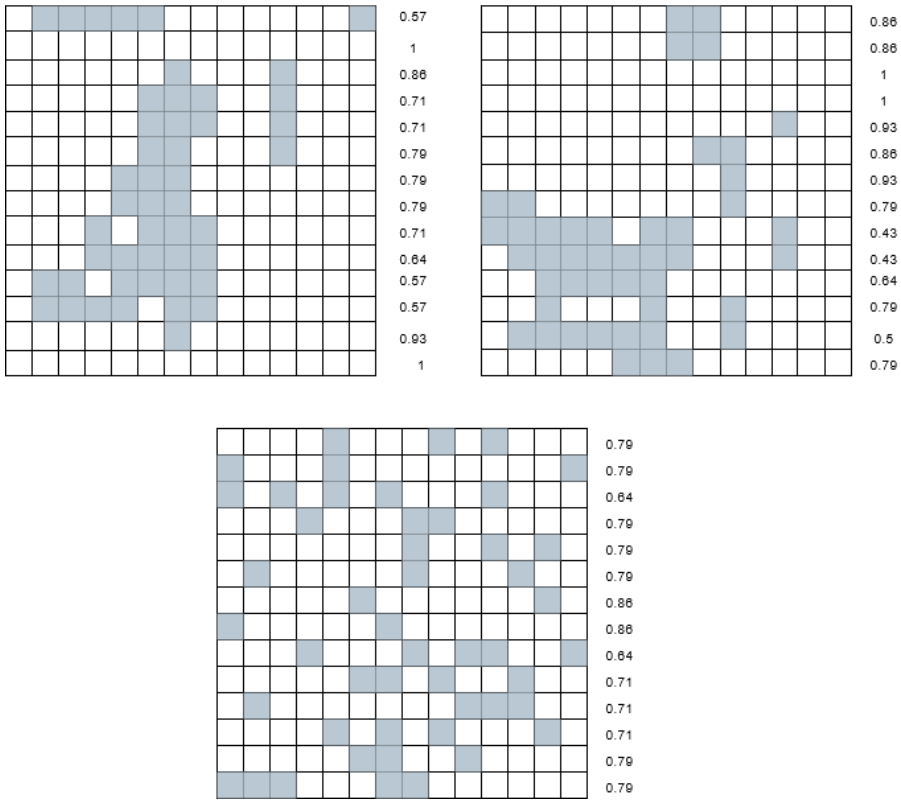


Figure 8.7. Differences between random matrix (bottom) and real matrices (top). In gray the not-null locations. The numbers on the right represent the sparsity of the single row.

Has already been seen how the difference in sparsity impact on the throughput when exploiting more PEs in parallel, because the faster PEs are forced to wait for the results coming from the slower ones. The PEs works row by row so is not the overall sparsity but the sparsity of the single row to influence the circuit. For this reason, we expect worse performance when working with real matrices because there are more possibilities to have different speeds between PEs.

In Table 8.5 are reported all the data about throughput obtained simulating the convolutional layers of VGG16 network. As previously said, only a subset of input channels is processed to speed-up the simulation; for every layer the number of channels used in the simulation is reported. Having two sets of input for the CNN,the simulations are repeated for each of them; in the table there are two rows per layer. The weights do not affect performance in any way so having only a set is not a problem. Note that there is again a column about sparsity and the values are different from Table 8.1. This is the average sparsity of the channels used in the simulation. The

Layer	Channels	Sparsity	MOPs (per PE)		
			1 PE	4 PEs	16 PEs
1_1	3	0	630	630	630
		0	630	630	630
1_2	16	0.1	-	-	-
		0.41	-	-	-
2_1	16	0.29	548	473	425
		0.36	537	448	431
2_2	16	0.63	438	322	285
		0.53	519	407	358
3_1	48	0.44	495	424	382
		0.49	510	436	372
3_2	48	0.65	440	360	299
		0.61	487	405	364
3_3	48	0.66	442	355	320
		0.66	470	398	336
4_1	48	0.71	414	351	248
		0.7	432	398	339
4_2	48	0.76	375	342	278
		0.76	383	371	322
4_3	48	0.82	317	302	260
		0.81	329	326	303
5_1	48	0.82	291	286	255
		0.76	361	346	285
5_2	48	0.78	331	323	270
		0.79	322	318	283
5_3	48	0.8	314	307	230
		0.79	324	309	280

Table 8.5. Throughput for every layer and different circuits. Layer 1_2 is too long to be simulated

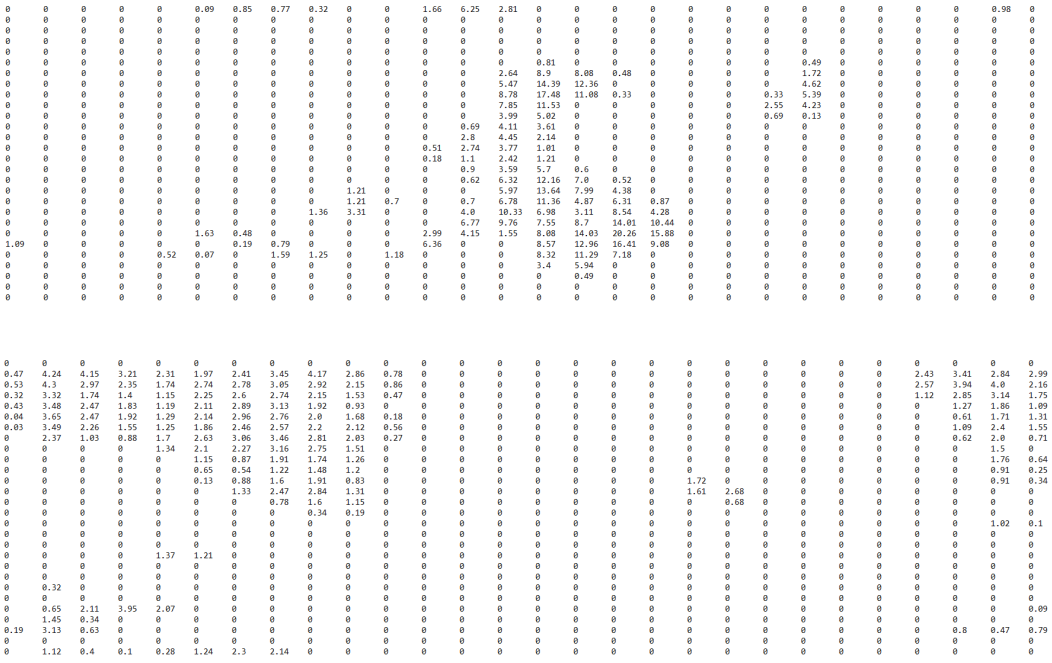


Figure 8.8. Non-uniformity in bigger matrices; taken from layer 4_1

throughput is reported in three different conditions: the circuit with only one PE, which assures the best possible individual throughput, the circuit with 16 PEs, which have the maximum possible throughput overall and the circuit with 4 PEs, that places between the two. The latter circuit allows simulation with more channels per PE, making it possible to have results less influenced by the specific matrices.

Looking at the values from the first column, they are similar to the ones obtained after synthesis with random matrices, and in some case even slightly better (is reported the table of chapter 6). This is not strange: in the case of a single PE, the non-uniformity does not worsen

Sparsity Input size	0.3	0.4	0.5	0.6	0.7
10x10	420	400	380	350	310
15x15	450	420	400	370	340
20x20	470	450	410	380	360
30x30	490	450	430	400	380
50x50	500	470	440	420	400

Table 8.6. Throughput table from synthesis chapter

performance because there are not channels elaborated in parallel. Instead, grouping values near each other, increase the recycling opportunities, making the convolution faster. On the other hand, the concentration of the values in a circumscribed region inside the matrix leads to other

areas where the rows are almost empty. This cause problems also for the configuration with only one PE and his accumulator: if the row is poor of values it will be processed very fast. In this cases is possible that the accumulator is not fast enough to accumulate the values of the row (that, we remember, is not sparse when exiting the PE), causing a bottleneck between the PE and the accumulator. This behaviour is visible with big matrices, where the rows are long and slow to be accumulated. The extreme limit is the presence of totally empty rows. As seen in section 3.4 the entire system of data storing and accumulation order is based on processing rows one by one, without "holes". For this reason, an empty row cannot be accepted: in the compression phase, at least one "dummy" value must be added to the empty row. The value must be the smallest possible to minimize the introduced error. The value is "1" considering the binary representation and its absolute value depends on the chosen quantization. In Figure 8.9 is show an example with a "5.11" quantization.

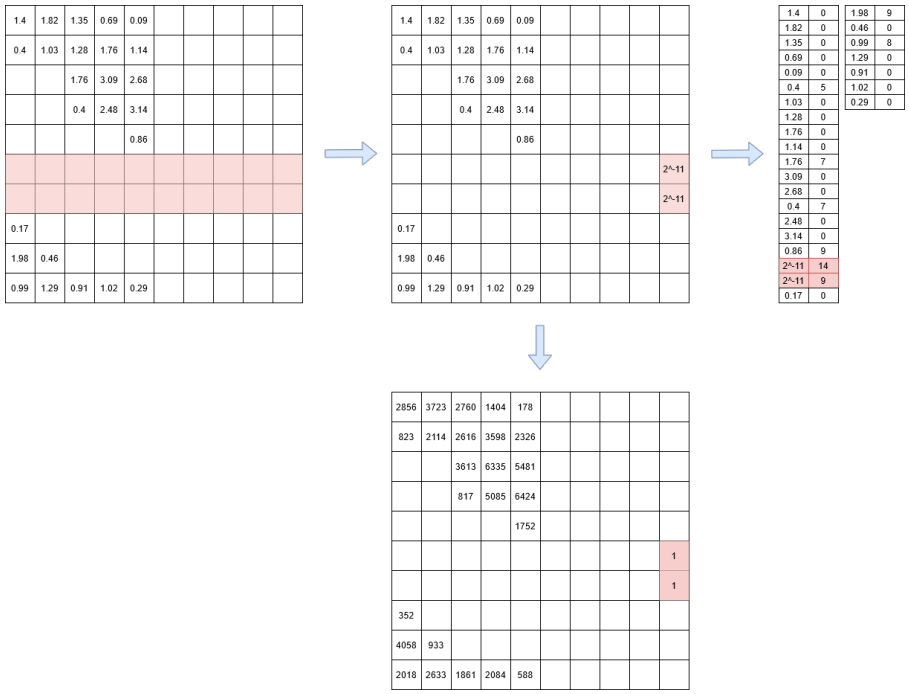


Figure 8.9. Empty rows in a matrix

When the number of PEs increases the situation change. The performance is now worse with respect to the values found with random and uniform matrices: in this case, also the matrices with the same sparsity have different configurations, increasing the differences in elaboration time. More the differences between matrices more the performance drop. The drop for the last layers of VGG16 is not so big: the performance is already very low due to the smallness of the matrices and the huge sparsity factor. For the first and intermediate layers, however, the performance drop is bigger than the one found with random matrices, with decreases up to 40% (for random matrices was about 10%). In Figure 8.10 the performance drop can be evaluated, for both the data set. This brings other variables on the choice of the best circuit to use.

Until this moment the throughput has always been the focus of the analysis and, looking at how it varies, we saw that the more the sparsity less the circuit's performance. This claim can sound strange thinking about the fact that the whole work is focused on exploiting sparsity to avoid useless computation. In fact, in this case, is not all about throughput. Increasing sparsity

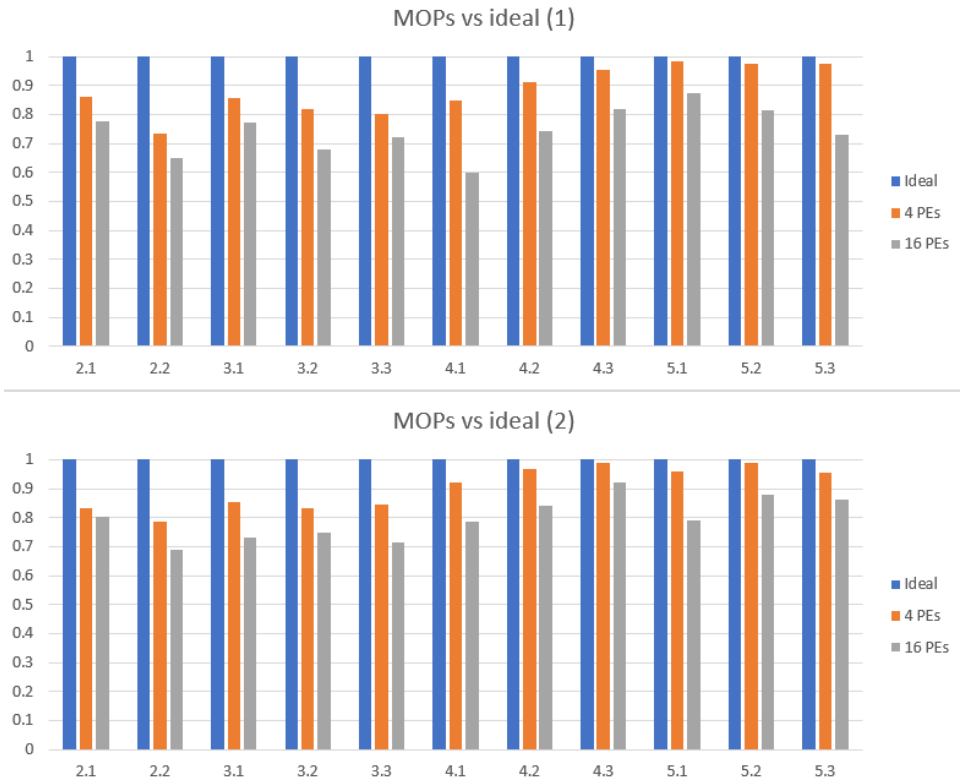


Figure 8.10. Graphical representation of the performance drop

for sure increase the average time needed to process the input values but, obviously, the number of values to be processed decrease. So the operations per seconds are reduced but with less operation to perform the convolution is speed up. In Table 8.7 is reported the number of spared computations (MAC) when exploiting sparsity with our algorithm. This is only a qualitative comparison but can clearly show how the initial purpose is accomplished.

As usual, the table has a row per layer; the numbers reported are the number of operations needed to compute an output channel. These values are not simulation results but are computed starting from the input channels and their sparsity. The table has entries for both the set of input at our disposal. There isn't a division per circuit because the number of operation to be done is independent of the structure of the accelerator. In Figure 8.11 and Figure 8.12 is reported a graphical representation of the data in Table 8.7.

In both the images are reported the operation when the sparsity is not exploited (dense, blue) and the number of computations when exploiting sparsity with the two set of data at our disposal (sparse1, orange and sparse2, gray). It's easy to see that going deeper inside the layers the acceleration increase; this is because the sparsity is greater in the last layers. The computation reduction follow the sparsity, as is shown in Figure 8.13.

Layer	Dense MACs	Sparsity	Sparse MACs	Ratio
1_1	1'430'016	0	1'431'366	1
		0	1'431'366	1
1_2	30'507'008	0.44	17'952'800	0.59
		0.43	18'218'138	0.6
2_1	7'626'752	0.31	5'409'846	0.71
		0.38	4'888'989	0.64
2_2	15'253'504	0.61	6'554'628	0.43
		0.50	8'068'509	0.53
3_1	3'813'376	0.42	2'307'805	0.61
		0.49	2'048'173	0.54
3_2	7'626'752	0.65	2'952'563	0.39
		0.62	3'201'800	0.42
3_3	7'626'752	0.67	2'816'078	0.37
		0.67	2'807'645	0.37
4_1	1'906'688	0.69	677'052	0.36
		0.69	680'841	0.36
4_2	3'813'376	0.76	1'100'502	0.29
		0.78	1'037'682	0.27
4_3	3'813'376	0.82	898'839	0.24
		0.81	937'899	0.25
5_1	953'344	0.84	231'514	0.24
		0.80	263'050	0.28
5_2	953'344	0.80	263'851	0.28
		0.81	255'202	0.27
5_3	953'344	0.81	258'829	0.27
		0.81	256'561	0.27

Table 8.7. Throughput for every layer related to sparsity

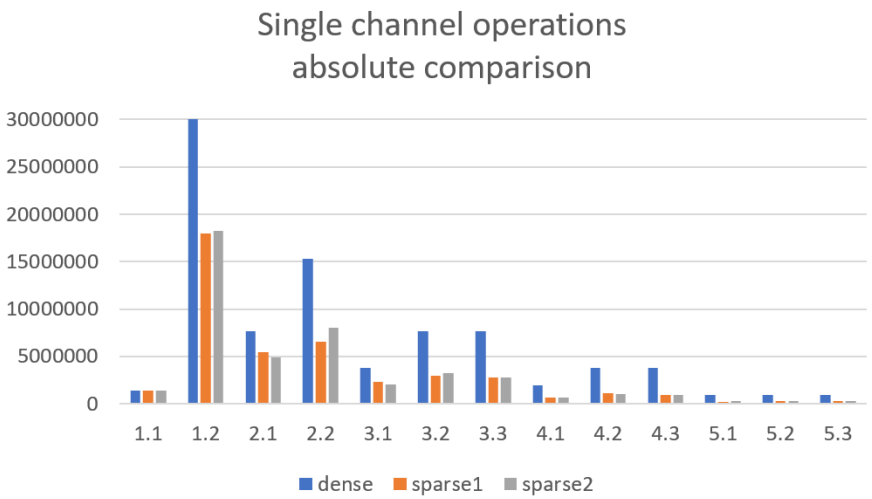


Figure 8.11. Computation reduction due to sparsity

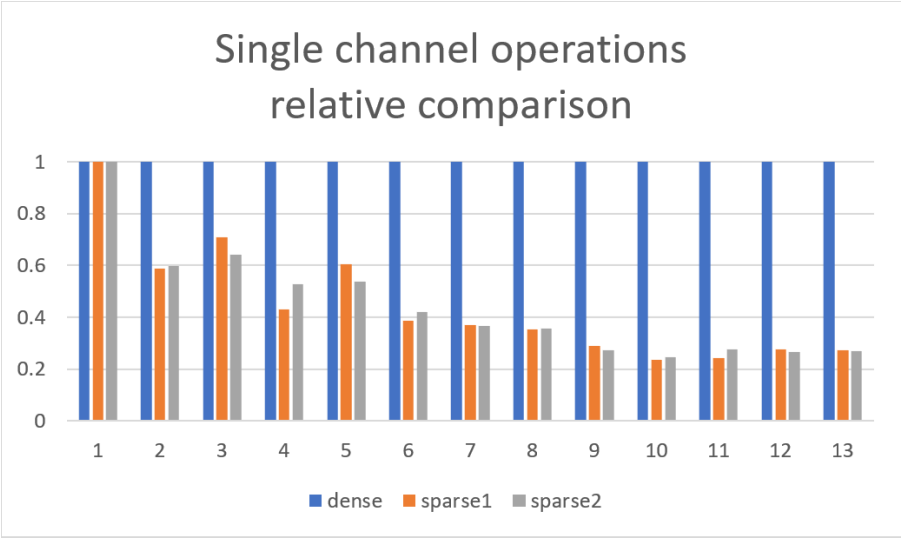


Figure 8.12. Computation reduction ratio

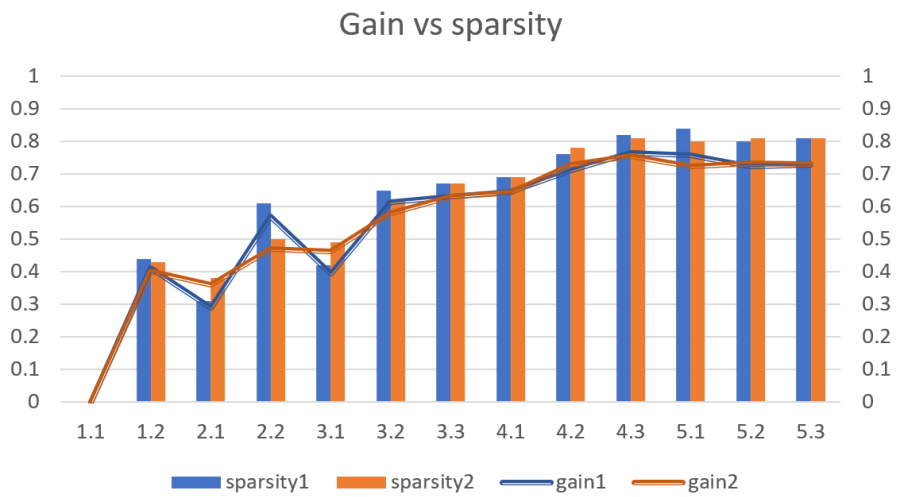


Figure 8.13. Gain vs sparsity

Chapter 9

Conclusion

Firstly a confrontation was made with the starting point of this thesis that is the paper Squeeze-Flow [1] ,also analyzing the limits of this comparison.

A more general comparison is then made by analyzing others state-of-the-art CNN accelerators, to evaluate the differences in the main metrics between our work and the circuits used nowadays. Moreover, several ideas on possible improvements are proposed. These are the main features that could be introduced continuing the work or suggestions about how to avoid issues that occurred during the accelerator's development.

9.1 Squeezeflow comparison

As previously said, the starting point of this thesis work was the SqueezeFlow paper [1], so we try to make a first comparison with respect to this architecture. SqueezeFlow reports an area of about $1.28mm^2$ for the logic elements(including $0.80mm^2$ MAC arrays) and about $3.50mm^2$ for the memory elements; the total area is about $4.80mm^2$. The basic Architecture developed in this thesis, that is with one PE and one accumulator, has a total area of $1.63mm^2$ of which about $1.5mm^2$ are related to the memories and the remaining part is logic.

Unfortunately, make a direct comparison is not so easy, because the two architectures are different. Furthermore, the algorithms behind the convolution process differ: SqueezeFlow performs the calculation in a "definitive" way, one location of output matrix at a time while the algorithm developed in this work iterates through input matrix locations. As already seen, this perspective change brings to the introduction of the memory hierarchy. In particular, the predominant part of the area of this thesis architecture is due to the accumulator, but it does not contribute to the basic 2D convolution. For this reason should be better to only take into account the PE in the comparison.

On the other hand, also the different parallelism has to be taken into account: SqueezeFlow architecture works with an array of 64 MAC units in parallel, whereas in our basic architecture only 9 MACs are computed together (equal to the number of kernel locations and maximum possible value). Also, in SqueezeFlow every MAC unit contain four multiplexers, instead of only one in our circuit. Given the strong difference in the number of MACs involved in the two cases,it is decided to make a comparison considering the area of the single MAC unit. To obtain the area of a single unit for SqueezeFlow the total area of the array's MAC is divided by the number of elements obtaining an area associated with the single MAC of $0.80/64 = 0.0125mm^2$. This is not very accurate since the area of the units also contains interconnections (in SqueezeFlow is present a network-on-chip architecture), which have a big impact on the total area.

Regarding the thesis architecture, to generate a more detailed area report the single MAC circuit is synthesized and it is possible to determine that it occupies an area of $0.005mm^2$. In every case the SqueezeFlow architecture presents a bigger area. As said above, the difference could be largely related to the massive amounts of interconnections from a MAC unit to the others.

A similar approach can be used for power consumption. To derive the power of the single MAC, it was necessary to divide the power associated with the processing unit($280mW$) by the number of MACs contained inside(64). The result obtained is a consumption of about $4.4mW$ associated with each MAC. Regarding the thesis architecture, the analyzed value is about $1.5mW$ for each MAC.

Unfortunately it is not possible to carry out a more detailed comparison with the SqueezeFlow architecture as the basic ideas are very different and substantially impact all the main metrics. In particular, it was not possible to set a comparison on performance as the paper does not report clear values in this regard but merely makes a comparison in relative terms with respect to some particular architectures. For this reason, the following paragraph shows a comparison with other examples in the literature which, despite being architectures different from the one developed in this thesis, report results with which it is possible to set a more complete confrontation considering different aspects.

9.2 State-of-the-art comparison

As seen in chapter 6, the main metrics that characterize an accelerator are area, throughput and power. To compare the implemented architecture with other examples present in the literature, it would be enough to compare these metrics. In this way, however, the comparison risks not being meaningful (as just stated in the comparison with SqueezeFlow) as the architectures are profoundly different from each other both at the algorithmic level of processing the values (which is then reflected in the architecture and therefore in the area) and in the operating frequency (which has a direct influence on power, other than on throughput). For this reason, "unitary" metrics have been introduced to make comparison independent from the specific architectures. These values are obtained from the ratio between other primary metrics, like throughput over area, energy efficiency or power density.

The first metric considered is the throughput over area ratio. Figure 9.1 shows the comparison

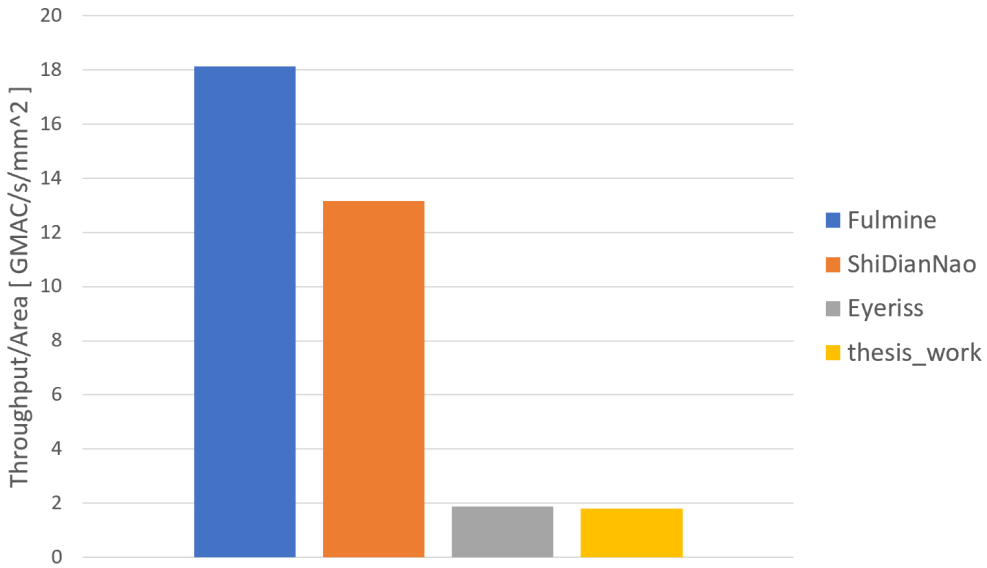


Figure 9.1. Throughput area ratio

between some architectures present in the literature (Fulmine [16], ShiDianNao [17], Eyeriss [18]) and our architecture, in the version with 16 PEs. It can be seen immediately that, apart from Eyeriss, other architectures perform better. The limitation to the number of MAC arises both from the maximum limit of 9 MACs within the PE(section 2.4) but also from the limited number of PEs that can be used in parallel(section 7.2). To improve the situation could be possible to rethink the convolution algorithm (albeit continuing to exploit the sparsity) or to optimize the control unit, decreasing the number of average clocks cycles required to process a value from the sparse input matrix. Another possibility could be to try to reduce the footprint of the area. In this case, the situation is more delicate as the main contribution to the area is due to the memory of the accumulator that is already shared by 16 PEs. The only possibility could be to bring a portion of this on-chip memory out of the chip, but in this case, the risk is to increase too much energy consumption. (accessing an off-chip memory is very expensive [14]).

The second metric is still linked to the MAC as it constitutes the basic operation performed by the architecture. It is shown in Figure 9.2, with the name of energy efficiency. Is defined as the energy needed to perform a complete MAC, the definition is in Equation 7.5. The elaborated

architecture has a higher value than the proposed examples and this is not optimal as this ratio should be as low as possible to reduce the consumption required for each operation performed. An opportunity to improve the situation could be to modify the internal parallelism of the architecture in order to reduce the area and consequently also the power consumption.

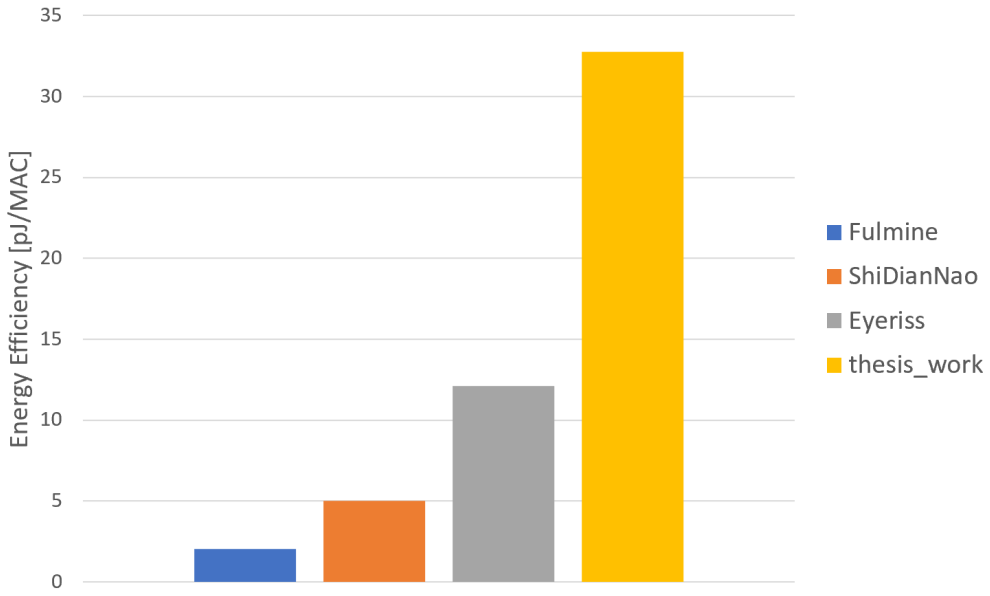


Figure 9.2. Energy Efficiency

Finally,the last quantity analyzed is the power density in Figure 9.3. Also in this case the value is higher than the examples present in the literature. The main explanation may be linked to the fact that our area is limited compared to the other examples while the power involved is comparable. This implies a higher power density.A possible solution could be to increase the architecture footprint to decrease this density, however this choice must be justified with, for example,a substantial increase in performance otherwise the risk is to improve the power density but worsen other metrics. Another possibility is to analyze the power consumption and try to reduce it to improve its density.In this case the focus should be mainly on the memories as this is where the main part of consumption is concentrated.

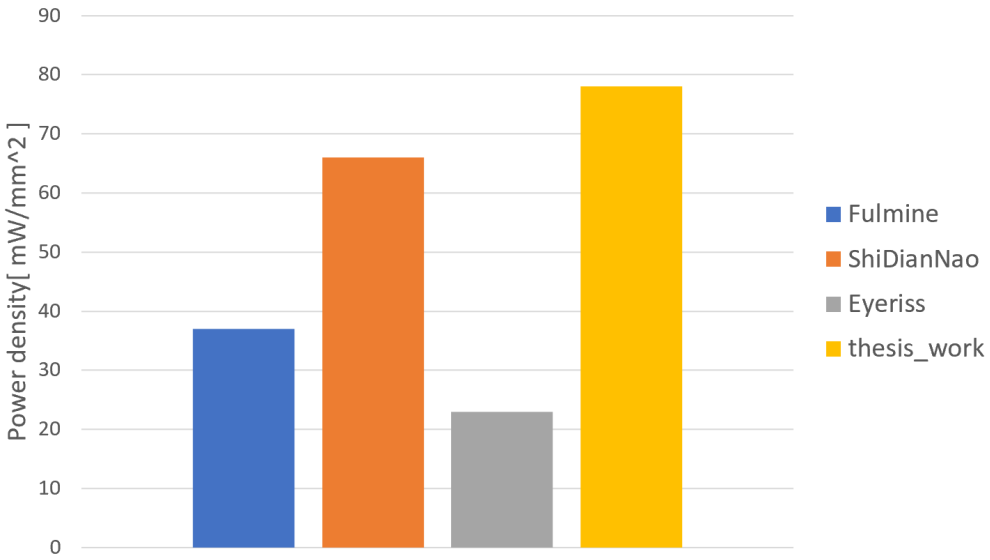


Figure 9.3. Power density

9.3 Problems and possible improvements

9.3.1 Matrices encoding

As analyzed in section 7.2, an important factor limiting performance is bandwidth. Considerations on performance limits will not be reported here but an analysis will be made on how it might be possible to overcome these limitations. The bandwidth has been set at 88 bits but the amount of values that can be contained in these 88 bits could change if the dimensions connected to each input change. As already mentioned, each input value is represented on 22 bits, where 16 bits represent the value and 6 bits the offset to retrieve the correct position of this input within the sparse matrix. One possibility would be to reduce the size of the inputs value. The other would be to choose a different way to encode the position of the inputs, to reduce the amount of bits needed to store the coordinate.

The first possibility would result in a loss of precision and is not analyzed here. The second possibility could help reduce the size associated with each input without losing precision. The choice to encode the position of the inputs on 6 bits arises from an analysis of the sparsity of the matrices and on the maximum number of consecutive zeros that can occur between two non-null values. This analysis led to choosing 6 bits as worst-case. A possible alternative could be the solution adopted in the NullHop architecture [7] in which a 3D mask is used to indicate non-null values for each input matrix. Basically a bit is associated to each value of the input matrix and this bit is 1 if the value of input is non-null otherwise the bit is zero.This solution is on average more efficient in encoding data than the RLC solution.The main difference is that the solution adopted by NullHop also associates a bit with null input values while the RLC solution associates 6 bits only to non-null values.

Adopting the NullHop coding would allow to eliminate the 6 bits of position coding and instead it would be necessary to download the 3D mask one piece at a time. For example, with a 100 * 100 matrix, the input values would be downloaded on 16 bits (instead of 22 bits) so with 88 bits of bandwidth you could download 5 input values at a time (before they were 4). Portion of the mask corresponding to this matrix consisting of $100 \cdot 100 \cdot 1 = 10000$ bits. With 88 bits of bandwidth, it would take 114 cycles of overhead to download this mask. The best solution would be to interlace the input download with the download of this mask in order to mitigate the overhead. It is obvious to remember that, changing the input encoding, it would then be necessary to adapt the decoding hardware component accordingly. In addition, an extra memory component would be needed to store the 3D mask.

Another difficult regard RLC encoding with 6 bits has been encountered when working with real matrices: the non uniformity of such matrices was not taken into account when selecting the number of bit to encode the coordinate. When the matrices are not uniform,the number of consecutive zeros increase and can, in some cases, overcome the 6 bits. Another compression method could solve also this problem.

9.3.2 Pipeline and parallel architecture

Another aspect that could improve the situation is to reduce the critical path of our architecture with the pipeline technique. As for the accelerator, the critical path concerns the input value involved in the multiplication operation with the kernel inside the MAC, so an additional register should be entered for the input value. As for the accumulator, pipe registers could be inserted in the adder cascade necessary to combine the results of the different accelerators.

The other problem with this architecture is the degradation of performance when you have a single shared accumulator for a certain number of accelerators. The penalty is due to the fact that it is necessary to synchronize the data of the different accelerators before accumulating them. If different accelerators process input matrices with very different sparsity, the throughput of these

accelerators will be very different and to ensure synchronization the faster accelerators will have to wait for the slower ones. The only way to improve this situation is to try to supply, each time to the different accelerators, input matrices with very similar sparsity.This does not completely eliminate the problem since even with equal sparsity, the input matrices can have different disposition of the values and therefore different possibility of recycling between two cycles (section 2.5), however it allows to reduce the throughput displacement of the different accelerators.

9.3.3 Low power techniques and memory optimization

The confrontation between this architecture and state-of-the-art accelerators shows not optimal values for the metrics of our circuit. This is also due to the lack of any optimization in our architecture. Some little and not-intrusive modification can improve a lot the power behaviour of the system. Clock gating can mask the clock to the sequential components where they are not used, reducing power consumption. Another trick which doesn't modify the circuit behaviour reducing power is using transistor with high threshold voltage in the not critical paths. There are plenty of other modification which can improve the circuit's metrics, with greater or lesser impact on the circuit design; future improvements in this sense could originate much better results in comparison.

Further improvement could also be carried out using different libraries in synthesis which, using smaller transistors, could improve the area, power and frequency metrics. The components which more than any other influence the performance are memories: they occupy most of the area, consume most of the power and constitute the critical path. Any, also little, improvements in the memories could have a big impact on the system in general.

9.3.4 More sparsity

Another limitation is that this architecture only removes the computations related to null inputs, exploiting sparsity only if is related to input matrices. The optimal solution could foresee to exploit also sparsity in kernels avoiding to perform null calculations in both situations. This would involve modifying the PT-KS dataflow with the possibility of overcome the limit on the number of MACs within the PE, which must correspond to the size of the kernel.

Another situation in which sparsity is not exploited is the transfer of data from PE to accumulator, when all the results are take into account, also the null values. Finding a smarter way to synchronize the value to be accumulated to the values in the accumulator memory,the null values in the 2-D output matrices could be ignored.

9.3.5 Kernel flexibility

This accelerator is meant, from the very first moment, to work with filter matrices with 3x3 size, and this parameter is fixed by construction. A new features that would be really interesting and would made the circuit even more flexible would be to make the kernel size changeable.

From an architectural point of view would not be so hard to implement such a functionality. Just like the memories and the input matrices size, the number of MAC units in the circuit would set the maximum limit to the kernel size: using a smaller kernel would underutilize the architecture.

The difficult is related to the control: the FSM which pilot the hardware is strictly correlated to the kernel size. To implement the variable kernel functionality a way should be found to make control flexible as well.

9.4 Final considerations

This last chapter allows us to make a general analysis of the work done. The heart of this work was the realization of a hardware architecture able to work with sparse inputs, avoiding computations with null values, and dense kernel. As is clearly reported in Figure 8.12 in section 8.3, this purpose has been achieved, with a lot of useless computations simply ignored.

The drawback of the implemented algorithm is the higher amount of partial results, which translates into an increase in memory accesses. The mechanism for recycling the results between the MAC units and the implemented memory hierarchy allows good management of the values by limiting the power consumption and the latency in the recovery of these values, mitigating the problem.

The division of the circuit into two different parts (PE and accumulator) limits the accesses to the external memory during the computation of 3-D output channels. Being the two parts almost independent, the core of the architecture, the PE where the MACs are processed, is not slowed down. The interface with the external matrix is characterized by a fixed bandwidth, which amount is determined by the amount of data being transferred. Our architecture maximizes the reuse of data, allowing a bandwidth lower than 100 bit per cycle.

The footprint of the circuit is strongly influenced by the accumulator area while the parallelism potential is quite limited as it capped to the number of weights of the filter matrix. To achieve a performance-area trade-off, architectures with a shared accumulator are used. The number of PEs working in parallel is limited by the bandwidth and, in the case of real input matrices, by the performance degradation introduced when different matrices are processed in parallel.

Finally, when compared with state-of-the-art accelerators, our architecture does not show good metrics. However, it must be considered that in this work we focused on a working architecture able to exploit sparsity. No kind of optimization is taking into account regarding power consumption, area or operating frequency. As seen in chapter 8, instead, the results are correct and the sparsity is fully exploited.

Appendix

Appendix A

In this appendix some important details are reported to obtain the synthesis of the architecture. In particular, the steps to synthesize using external libraries for memories. An example is also shown in which, using these libraries, the timing is altered and consequently it was necessary to insert registers to reduce the critical path and restore the initial situation.

A.1 synthesis process

In the first instance, the procedure carried out to obtain the area reports was the standard one followed in some degree courses.

As a brief summary, the main steps foreseen by the standard procedure are reported:indication of the technology with which the synthesis is to be carried out, indication of all the necessary vhdl files and clock period at which the synthesized netlist must work. Once the synthesis is obtained using the Synopsis tool, information is provided on whether the set clock period is violated by the architecture or not. In the event of a violation, the synthesis must be re-performed with a higher clock in order to eliminate the timing violations and obtain a consistent indication on the area.

As already said previously, the memories present in the architecture play a very important role on the area and therefore it was decided to replace the vhdl that described the memories with instances of library memories provided by Faraday. The parameters of the memories, such as parallelism of the location and number of address bits, obviously remain unchanged but using library files,instead of behavioral descriptions in vhdl,allows to obtain more realistic area values.

The first change required in this case is to modify the .synopsys_dc.setup file to add the various libraries needed to perform the new synthesis.The other main change to be made is to modify the vhdl files of the memories by introducing an instance of the library component inside. The rest of the procedure remains standard and in this way the area values reported in chapter 6 were obtained.

A.2 Timing analysis with libraries

The library files used in the synthesis concern all the memories present in the architecture, i.e. accelerator memory, accumulator memory, FIFO input, FIFO output and the FIFO between PE and accumulator.

Using these libraries means, in addition to obtaining more realistic area values, also introducing non-idealities into the system as it is necessary to take into account characteristics such as the timing necessary to read or write data that are not normally present in the vhdl description. This aspect is very important because if this library timing is added to the critical path it could happen to have a violation of the clock for which it is necessary to make the architecture work at lower frequencies. In particular, the main problems concerned the interface between the input

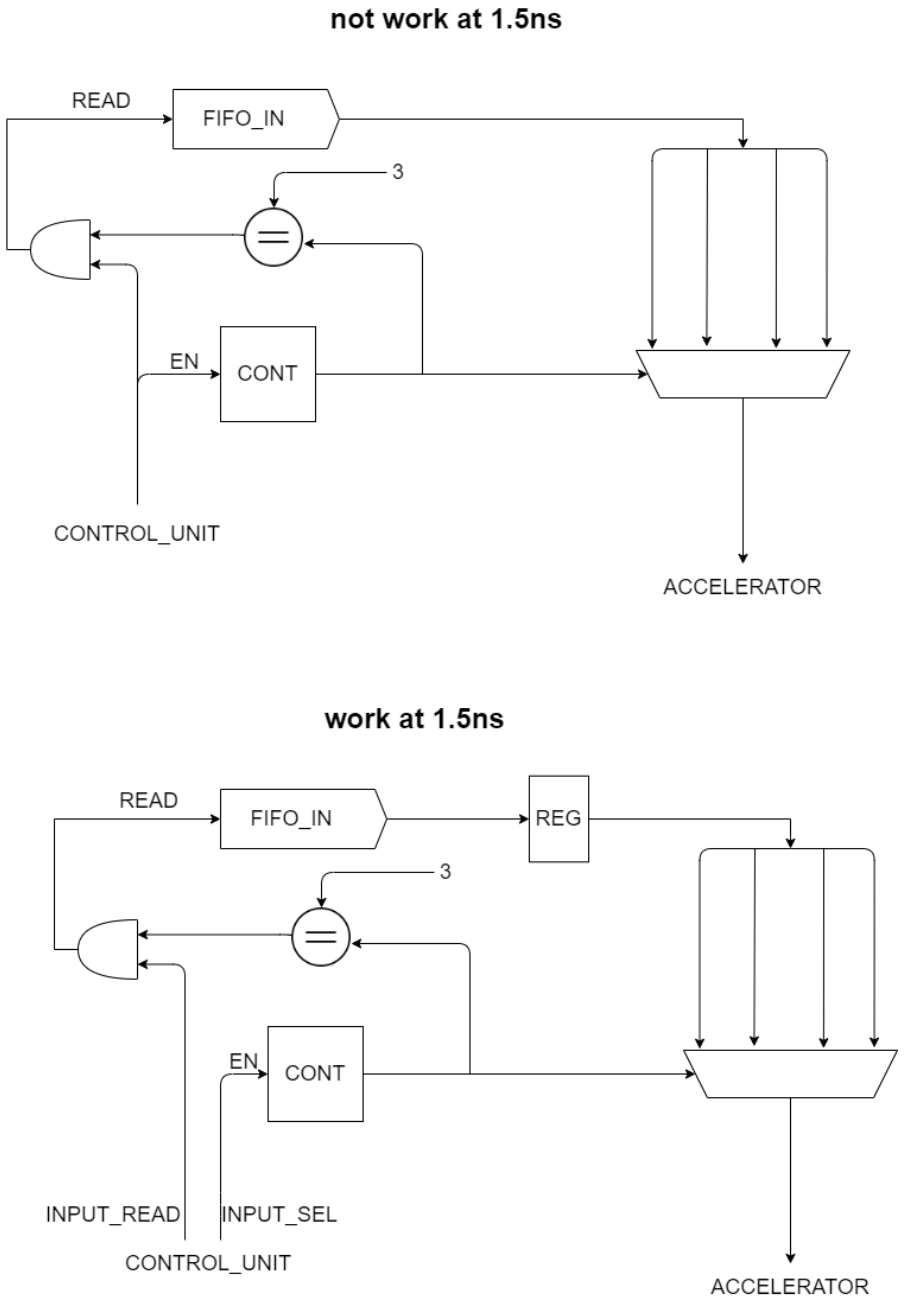


Figure A.1. Changes to maintain working frequency accelerator

FIFO and the accelerator. As previously said, these memories require a certain time between read request and when the data is actually supplied and in this specific case this time was found in $0.94ns$. This time added to the critical path present in the accelerator exceeded the value $1.5ns$ which was the operating frequency of the accelerator. To solve this problem, as it is also shown in the Figure A.1,a register has been introduced between the FIFO input and the accelerator in

order to break the critical combinational path and re-establish the operating frequency of $1.5ns$ for the accelerator.

It is important to underline the fact that the register allows to reduce the combinatorial path but introduces a delay equal to a clock cycle. For this reason it was necessary to divide the reading of the input fifo from the driving of the multiplexer.

Appendix B

In the chapter 8,an analysis was carried out on the architecture output relative error with respect to the VGG16 reference values. In this appendix, on the other hand, the absolute error is taken into consideration. In particular, a statistical analysis of this error was carried out to understand its distribution in the various layers VGG16.

B.1 statistics error distribution

To carry out this error analysis, we followed the same approach already used for the precision in the chapter 8, i.e. using the python model of the architecture to simulate the complete VGG16 layers and obtain the results of all the output channels. The python results were then compared, location by location, with the reference values VGG16 to obtain indications on the absolute error using the following formula:

$$absolute\_error = matrix\_ref - matrix\_2D\_python \tag{9.1}$$

The goal was then to analyze these errors in order to understand where they concentrated more, that is, what was the error value that happened most often. In the beginning,we collected these errors by keeping them separate for each output matrix of each layer VGG16. However, each layer has many output channels (up to 512) and therefore a table or graph would be needed for each of them. It is easy to understand that it is necessary to adopt an alternative solution to represent these errors in a compact way.

It was decided to join together the analysis of all the output matrices relating to a layer and indeed,in the following images, each curve represents a specific convolutional layer of the VGG16 network. All the layers that reported a similar distribution of the error were grouped in the same graph.

The title of the graph means the analysis of the error between the reference values VGG16 ("real" values) and the values coming out of the architecture model taking into account the various internal precision losses ("reconstruct" values).

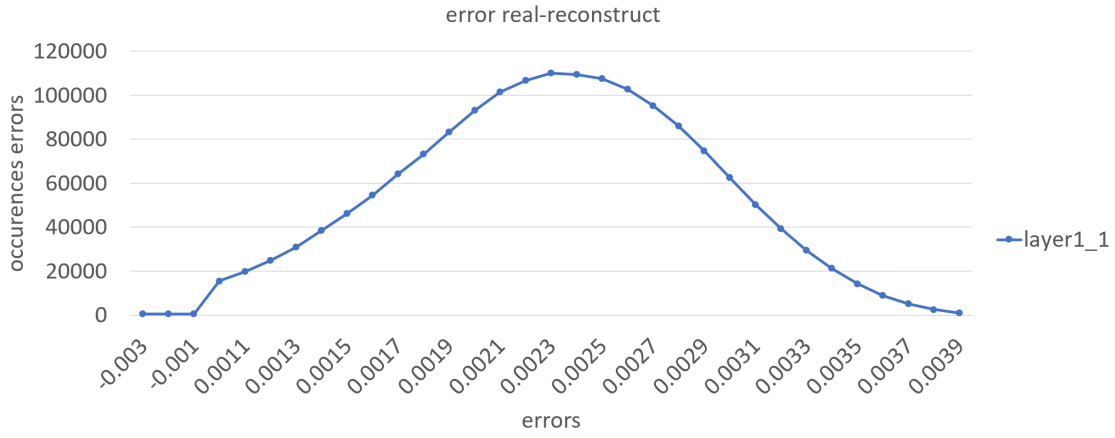


Figure B.1. Error distribution for convolutional layer 1_1

A couple of interesting observations emerge from the analysis of these graphs:

- The distribution of these errors between one layer and the next is very different in the first layers (at least or up to 7) but then tends to overlap gradually more and more. This behavior

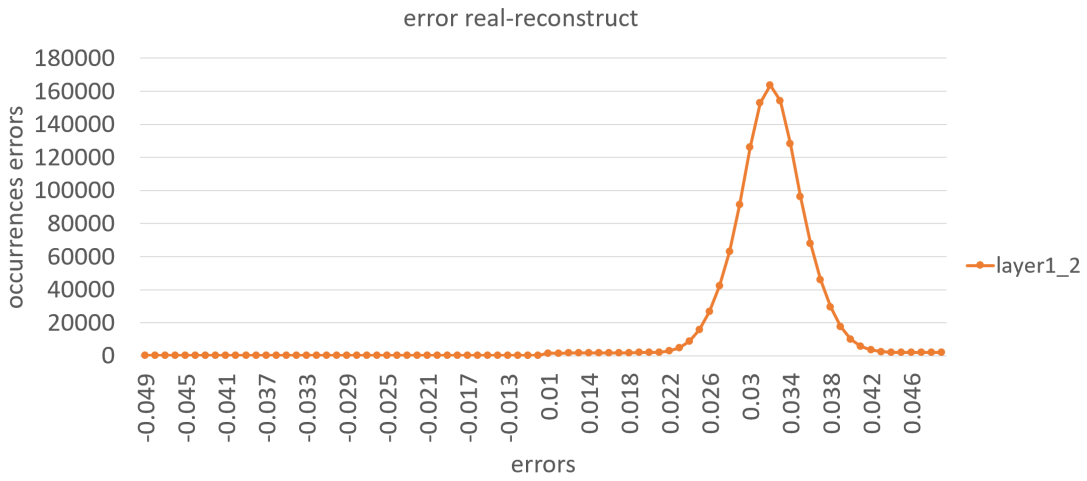


Figure B.2. Error distribution for convolutional layer 1_2

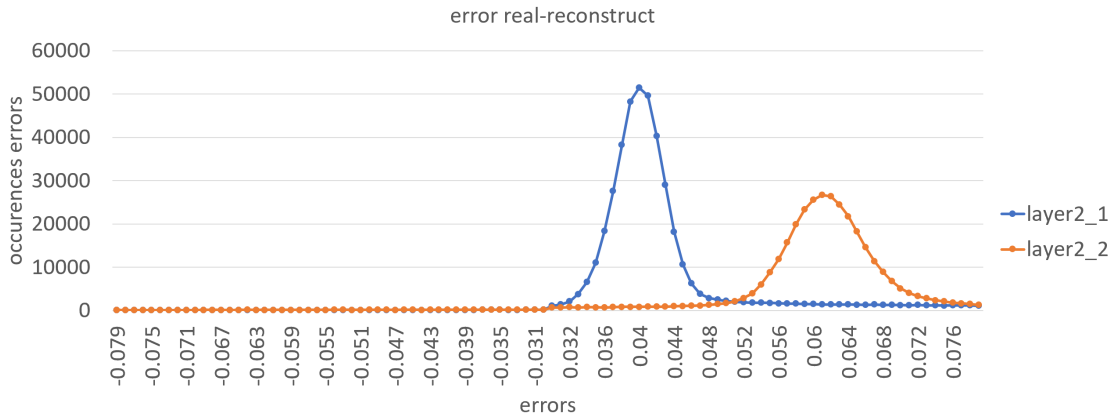


Figure B.3. Error distribution for convolutional layers 2_1 and 2_2

can derive from the fact that each convolutional layer tends to "refine" the results of the previous layer, therefore also the error tends to become uniform.

- Linked to the concept of sparsity is the peak of each curve. The difference between the maximum numerical values of the curves of the first layers (up to 14) and the final layers is very high. This is due to the fact that,as move from one layer to the next,the sparsity increases. As previously mentioned, in a real case such as VGG16,the distribution of sparsity is not random but tends to concentrate in precise areas of the input matrices.This means that there will be portions of the null input matrix that will not be processed by the architecture and which will be reflected in null zones in the output matrices. Therefore the number of non-null values in output decreases and consequently the possible quantity of errors in output is reduced as the null output zone that has not been processed by the architecture cannot have introduced errors with respect to the reference values.

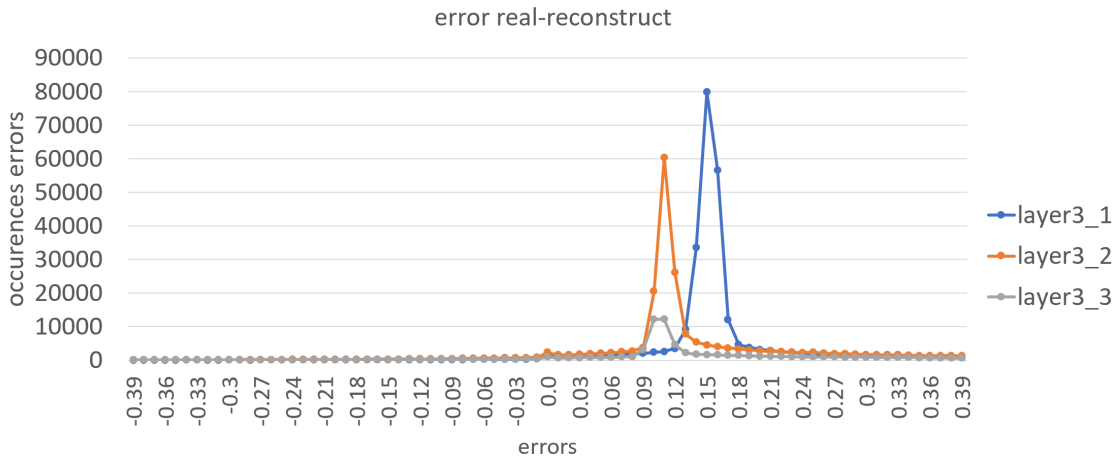


Figure B.4. Error distribution for convolutional layers 3_1, 3_2 and 3_3

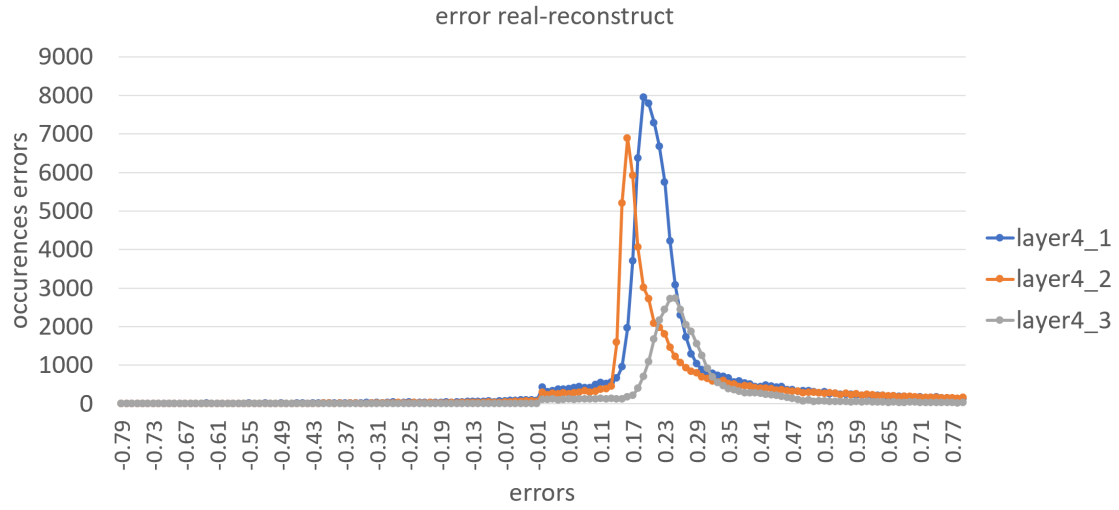


Figure B.5. Error distribution for convolutional layers 4_1, 4_2 and 4_3

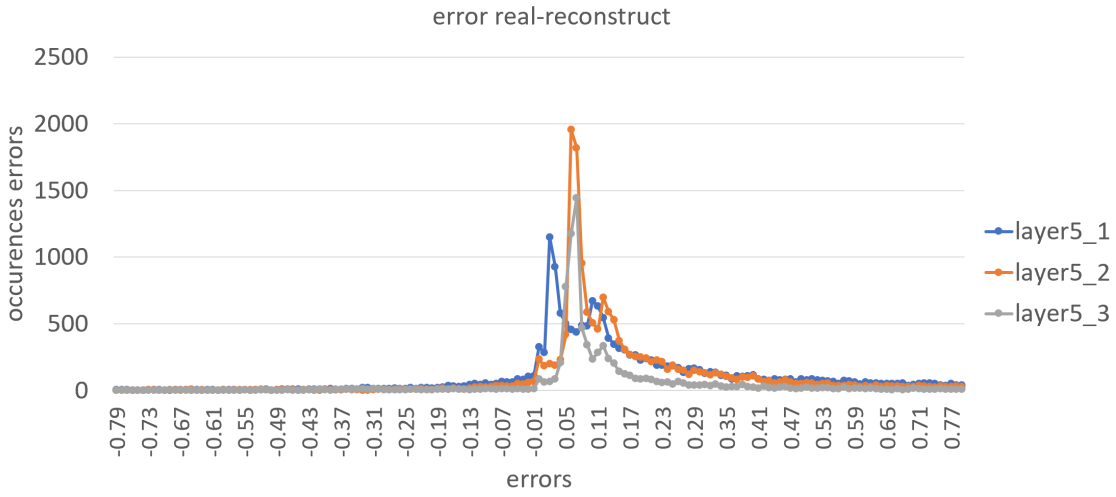


Figure B.6. Error distribution for convolutional layers 5_1, 5_2 and 5_3

Bibliography

[1] J.Li, S.jiang, S.Gong, J.Wu, J.Yan, G.Yan, X.Li "SqueezeFlow: A Sparse CNN Accelerator Exploiting Concise Convolution Rules" Proceedings of the IEEE, vol. 68, pp. 1663– 1677, Nov 2019

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

[3] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," Proc. IEEE, vol. 86, no. 11, pp. 2278–2324, Nov. 1998

[4] CNN bases,URL:"https://kunicom.blogspot.com/2017/08/27-cnn-basic.html"

[5] Al-Masri A. How Does Back-Propagation in Artificial Neural Networks Work?, URL: "https://towardsdatascience.com/how-does-back- propagation-in-artificial-neural-networks-work-c7cad873ea7"

[6] Batch, Mini Batch & Stochastic Gradient Descent, URL: "https://towardsdatascience.com/batch-mini-batch-stochastic-gradient-descent-7a62ecba642a"

[7] A.Aimar, H.Mostafa, E.Calabrese, A.Rios-Navarro , R.Tapiador-Morales,I.Lungu, Moritz B.Milde, F.Corradi , A.Linares-Barranco, Shih-Chii Liu, T.Delbruck "NullHop: A Flexible Convolutional Neural Network AcceleratorBased on Sparse Representations of Feature Maps",2018.

[8] S. Zhang, Z. Du, L. Zhang, H. Lan, S. Liu, L. Li, Q. Guo, T. Chen, and Y. Chen, "Cambricon-x: An accelerator for sparse neural networks," in Proc. 49th Annu. IEEE/ACM Int. Symp. Microarchitecture, 2016, pp. 1–12.

[9] A. Karpathy, "Stanford university cs231n: Convolutional neural networks for visual recognition," vol. 1, 2018.

[10] Junxi Feng, Xiaohai He, Qizhi Teng, Chao Ren, Honggang Chen and Yang Li , "Reconstruction of porous media from extremely limited information using conditional generative adversarial networks" pp.1-14,2019.

[11] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, "Going deeper with convolutions," in Proc. IEEE Conf. Comput. Vis. Pattern Recognit., 2015, pp. 1–9.

[12] K. Simonyan and A. Zisserman, "Very deep convolutional networks for largescale image recognition," in Proc. 3rd Int. Conf. Learn. Representations, San Diego, CA, USA, May 2015. [Online]. Available: http://arxiv.org/abs/1409.1556

[13] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in Proc. Int. Conf. Neural Inf. Process. Syst., 2012, pp. 1097–1105.

[14] Kevin Siu, Dylan Malone Stuart, Mostafa Mahmoud, and Andreas Moshovos "Memory Requirements for Convolutional Neural Network Hardware Accelerators" University of Toronto,pp1-11

[15] Pytorch documentation,URL: "https://pytorch.org/docs/stable/index.html"

[16] F. Conti, R. Schilling, P. D. Schiavone, A. Pullini, D. Rossi, F. K. Gurkaynak, M. Muehlberghuber, M. Gautschi, I. Loi, G. Haugou, ¨S. Mangard, and L. Benini, "An iot endpoint system-on-chip for secure and energy-efficient near-sensor analytics," CoRR, vol. abs/1612.05974, 2016.

[17] Z. Du, R. Fasthuber, T. Chen, P. Ienne, L. Li, T. Luo, X. Feng, Y. Chen, and O. Temam, "Shidiannao: Shifting vision processing closer to the sensor," in 2015 ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA), pp. 92–104, June 2015

[18] Y. Chen, T. Krishna, J. S. Emer, and V. Sze, "Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks," IEEE Journal of Solid-State Circuits, vol. 52, pp. 127–138, Jan 2017